Circuits and Systems
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ens.ewi.tudelft.nl/

CAS-MS-2009-16

M.Sc. Thesis

VHDL to SystemC: The Design of a Translator

Gert Jan Schoneveld

Abstract

VHDL and SystemC are both languages to describe or model circuits and systems. Reasons could exist for wanting to translate a model in VHDL to an equivalent model in SystemC. A system in SystemC can be needed for modeling a system with a software part, for a faster simulation, or because some tools only support SystemC.

This thesis presents a tool that performs this translation from VHDL to SystemC. The tool is constructed like a regular compiler: It consists of a front-end that reads and analyzes VHDL code and a back-end that generates SystemC code. The front-end came from the FreeHDL project and we have made the back-end ourselves. The back-end generates SystemC code by traversing the tree that comes from the front-end.

We have validated that the tool is suitable for simulation purposes and we have verified that the tool translates VHDL without problems in most cases we have seen in the wild.

VHDL to SystemC: The Design of a Translator

Thesis

submitted in partial fulfillment of the
requirements for the degree of

Master of Science

in

Computer Engineering

by

Gert Jan Schoneveld
born in Landsmeer, The Netherlands

This work was performed in:

Circuits and Systems Group
Department of Microelectronics & Computer Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Delft University of Technology

Delft University of Technology
Department of
Microelectronics & Computer Engineering

The undersigned hereby certify that they have read and recommend to the Faculty of Electrical Engineering, Mathematics and Computer Science for acceptance a thesis entitled **"VHDL to SystemC: The Design of a Translator"** by **Gert Jan Schoneveld** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dated: 27 August 2009

Chairman: _____
prof.dr.ir. A.J. van der Veen

Advisor: _____
dr.ir. T.G.R. van Leuken

Committee Members: _____
dr.ir. A.J. van Genderen

_____

Abstract

VHDL and SystemC are both languages to describe or model circuits and systems. Reasons could exist for wanting to translate a model in VHDL to an equivalent model in SystemC. A system in SystemC can be needed for modeling a system with a software part, for a faster simulation, or because some tools only support SystemC.

This thesis presents a tool that performs this translation from VHDL to SystemC. The tool is constructed like a regular compiler: It consists of a front-end that reads and analyzes VHDL code and a back-end that generates SystemC code. The front-end came from the FreeHDL project and we have made the back-end ourselves. The back-end generates SystemC code by traversing the tree that comes from the front-end.

We have validated that the tool is suitable for simulation purposes and we have verified that the tool translates VHDL without problems in most cases we have seen in the wild.

Acknowledgments

I would like to thank prof.dr.ir. A.J. van der Veen for proof-reading and for teaching me the art of scheduling, and I would like to thank Roël Seedorf for proof-reading.

Gert Jan Schoneveld
Delft, The Netherlands
27 August 2009

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

The hardware description languages VHDL [4] and SystemC [6] are used to describe or model circuits and systems. There are various situations where a VHDL model needs to be translated to SystemC. A system in SystemC can be needed for modeling a system with a software part, for a faster simulation, or because some tools only support SystemC.

1.1 The project

The goal of this thesis project is a tool that performs the translation of VHDL to SystemC automatically.

We limit ourselves to the VHDL constructs we have learned during our studies. It is a very limited subset of the official VHDL, but not of the VHDL we are seeing in the wild. The world seems to use only 'our' constructs. An example is the large VHDL model of an Intel 8051 that we use in Section 5.1.1.

Exotic constructs like aliases, guards, blocks as well as most attributes are not supported.

1.2 Alternatives

The different problems that our tool tries to solve may have other solutions. It is for example possible to model a system with a software part using a SystemC and VHDL Co-simulation [17].

1.3 Thesis Overview

This section describes the structure of this report. Chapter 2 presents a historical overview on the topic. Chapter 3 gives the structure of the tool and the reasons behind it. Chapter 4 is about the implementation of the mapping to SystemC. The usefulness of the tool is addressed in Chapter 5. Chapter 6 gives a summary and some ideas for the future. The manual in Appendix A is not part of the story, but could be needed if you want to use the tool. (And as the designer I hope you do.)

2 Historical overview

There has been a lot of research on the parsing of VHDL and the interoperability between VHDL parsers and the tools that want to use them. The VHDL language is very poorly defined and it still contains a large number of ambiguities in the grammar [16]. This makes it very hard to write and maintain your own VHDL parser.

In the early 1990s an IEEE workgroup was formed to develop the VHDL Intermediate Format (VIF). The purpose of this (planned to be) standardized intermediate representation was to improve the interoperability between the VHDL parsers and the simulators or other tools [15]. VIF never reached completion.

Because the interoperability still needed improvement the University of Cincinnati started a similar project in the late 1990s. They called their intermediate representation the Advanced Intermediate Representation with Extensibility / Common Environment (AIRE/CE) [19]. AIRE/CE also never reached completion but derivatives of it are used in all parsers that we examined in this project.

At the beginning of the 21st century the Università di Verona has designed a VHDL to SystemC translator for their "Environment for Functional Test Generation" [11][12]. They have designed and documented a large number of mappings from various VHDL constructs to SystemC. Their efforts were of great value in the design of our tool.

3 The Structure of the Tool

The tool is structured as any other compiler. It consists of a front-end that reads and analyzes the VHDL input and a back-end that generates SystemC output. Figure 3.1 shows this.

In section 3.1 the choice for the front-end is explained. Section 3.2 gives information about the tree that is built by the front-end. Section 3.3 shows the structure of the back-end. It explains in general how the information in the tree is being processed and how this information is ending up at the right location in the output files.

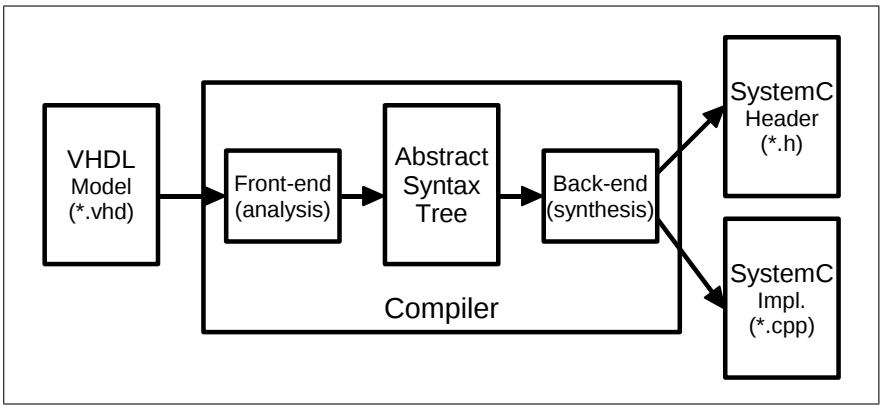


Figure 3.1: The structure of the translator

3.1 The Front-end

As front-end of the tool the existing analyzer from the FreeHDL project is used. Because of the large number of ambiguities in the official VHDL grammar [16] writing our own front-end would have taken too much effort. Now by choosing an existing front-end we have been able to focus on the development of the back-end of the tool.

There are several VHDL-analyzers public available. Because our tool should merge dependant design units like an architecture and its entity (see Section 4.1), the search was limited to the VHDL-analyzers that contain a mechanism to make them aware of each other.

We have examined the analyzers from the GHDL, SAVANT and FreeHDL projects. In the following sections our experiences with them are described.

3.1.1 GHDL

GHDL is an open source VHDL simulator written in the language Ada [1]. It uses GCC as code generating back-end. The clear documentation attached to GHDL gives

only information for its users like the available command line switches. There is no documentation about the inner workings. We needed to understand this inner workings in order to be able to construct our own back-end for it.

GHDL has implemented 2 procedural interfaces for communicating with other tools. One of them is the Verilog Procedural Interface (VPI) [10]. By reading the specification we found out that VPI is only designed to behave as a simulation interface [5]. With VPI it is easy to intervene in a simulation, but it is not suitable as an interface between a parser and a back-end tool like for instance a synthesizer. In this project we wanted to do the latter and so we could not use VPI.

The other implemented interface is the VDHL Procedural Interface (VHPI) [7]. VHPI can just like VPI be used as a simulation interface but also as an interface to get model information. VHPI should theoretically be able to act as a standardized interoperability interface.

But after searching in the GHDL source code we had to conclude that only the parts of VHPI needed for controlling a simulation were implemented. So our tool could not use GHDL as front-end.

3.1.2 SAVANT

SAVANT is designed as a standalone VHDL analyzer [20]. The analyzed VHDL is accessible using a well documented plug-in mechanism. The documentation is even accompanied with a small example. SAVANT should also be able to generate C++ for their own simulator [21].

SystemC can be viewed as a simulator written in C++ and we thought that with some modifications the C++ generator could generate SystemC. But the whole C++ generator was missing in the distribution and that is why we abandoned SAVANT.

Looking back using our experiences with FreeHDL we have to acknowledge that this was a premature decision. Even if the C++ generator was included, it was probably too big to handle in this one person thesis project. Because of our misjudgement we have neglected the plug-in mechanism and the example that comes with it.

3.1.3 FreeHDL

The FreeHDL project has also a custom VHDL simulator [2]. The front-end is called the VHDL Analyzer and Utility Library or VAUL in short [3]. The standard back-end is a C++-generator that generates code to use with their own simulator. It is similar to the ideas behind SAVANT, but this back-end was working.

We viewed this analyzer as the only one that suited our needs and we started investigating the back-end. After a while we decided to print out the back-end on paper and it was until then that we saw that we really underestimated the size of it. The printer spilled out hundreds of pages containing tens of thousands lines of code. It was impossible to understand and to modify them all in our project.

We learned a lot about VAUL and the tree during this investigation of the back-end and that is why we decided to stay with VAUL. If we would have moved to SAVANT at this point our time with VAUL would have been a terrible waste of time.

At the end we have built our tool from the example that came with VAUL/FreeHDL. It is a back-end that generates Pseudo C and it does that with a manageable 2000 lines of code.

3.2 The Abstract Syntax Tree (AST)

The VAUL analyzer builds an Abstract Syntax Tree (AST) from the input and checks a large number of attributes including the types. The resulting tree is an annotated tree containing all information needed by our back-end. Figure 3.2 shows as example a subtree of a variable assignment statement. The annotation is not shown in the figure, but in reality every node has a field `subtype` saying that it is an integer in this case.

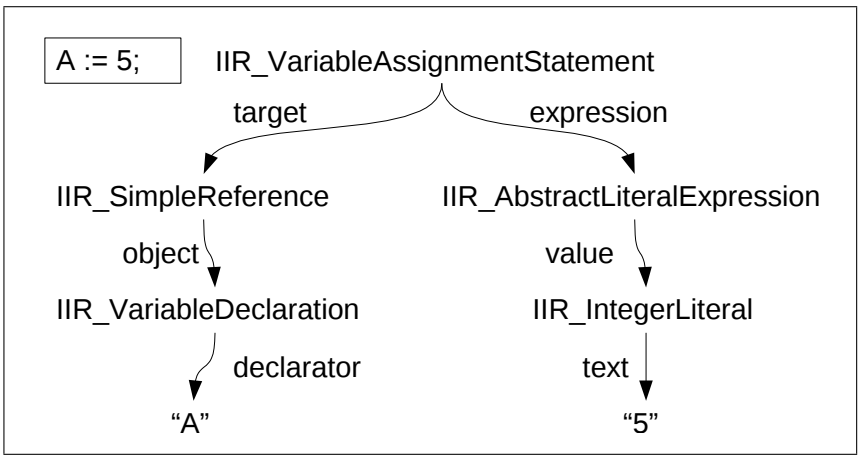


Figure 3.2: A statement in the tree

3.3 The Back-end

The back-end has to traverse the tree and while traversing it, it has to generate and send SystemC code to the 2 output files. The traversal of the subtree from Section 3.2 is shown in Figure 3.3.

On first sight it seems to be very difficult, but in reality it is not. The tree traversal is done recursively and a node only processes its direct children. Listing 3.1 shows the code to traverse the statement.

The SystemC code is sent to `*pActiveStream` in the listing. The pointer `pActiveStream` points to a `stringstream`. Using a pointer makes it possible to change the stream from above. This mechanism is used to send SystemC code to different places in the output files.

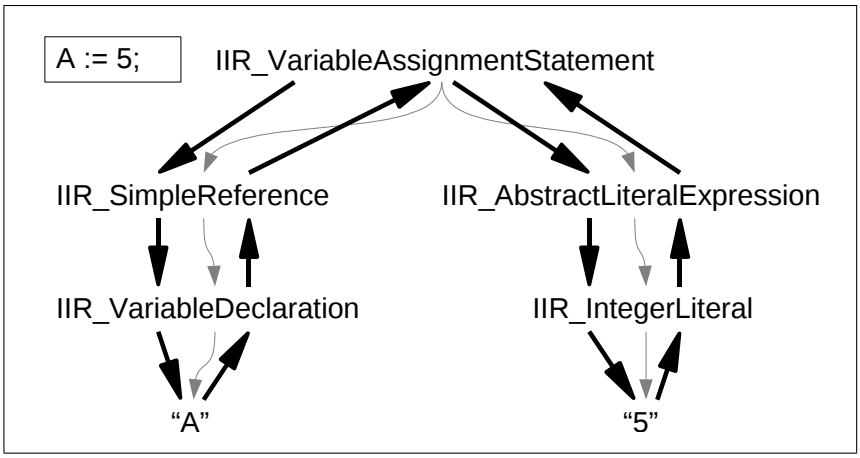


Figure 3.3: Traversing a node

Listing 3.1: Traversing and generating SystemC at the same time

```
void m_emit_stat(pIIR_VariableAssignmentStatement a)
{
  emit_lvalue_expr(a->target);
  *pActiveStream << " = ";
  emit(a->expression);
  *pActiveStream << ";" << endl;
}
```

Converting the Tree to SystemC

4

This chapter describes the implementation that maps the tree with the VHDL constructs to SystemC. During the implementation we had great help from the work done by the Università di Verona [11]. As already mentioned in Section 2, they have designed and documented a large number of mappings from VHDL to SystemC. If we would have wasted time designing their mappings ourselves we certainly would not have created the tool we have created now.

The rest of the chapter briefly summarizes the translation of the major language features of VHDL and identifies the difficulties beginning with the design units in section 4.1.

4.1 Design units

In VHDL a model consists of multiple design units. An *entity* declares the inputs and outputs of a component and the *architecture* models its behavior. To define common types in a model and to reuse procedures a *package* can be used.

One difficulty concerning the design units that had to be solved is that in SystemC the entity and the architecture are not separated like in VHDL. Although a SystemC component is split up in a header and a implementation file, the header file also contains information about the architecture like the declarations of its procedures and the connections to its subcomponents. As solution the tool traverses the tree of the entity again if the architecture is being processed.

Writing to the correct locations in the output files is done with streams (see Section 3.3). There are 4 locations where some things may need to be written an that is why 4 streams are stored globally. They are `templ`, `constructor`, `body` and `implementation`. Figure 4.1 shows where each stream is used for.

After walking the tree of a design unit, the procedure `write_streams(..)` writes the streams with the SystemC output to the header (.h) and implementation (.cpp) files.

4.1.1 The generated header file

The first thing that is printed in the header file is a guard to prevent multiple inclusions of the header. The guard looks like the one in Listing 4.1.

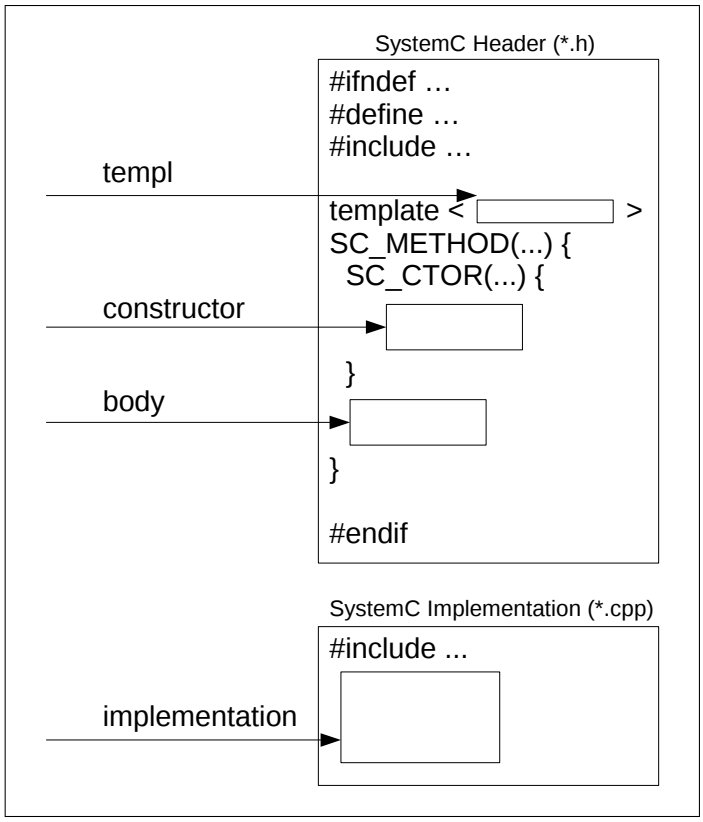


Figure 4.1: Available streams

Listing 4.1: Header guard

```
#ifndef MODULE_H
#define MODULE_H
  // ...
#endif /* MODULE_H */
```

After this the included headers are printed. One of them is of course `systemc.h` that declares the types and macros defining the language SystemC. The needed system headers and local headers are also printed as respectively `#include <file.h>` and `#include "file.h"`.

If the design unit is not a package a `SC_MODULE` is printed in the header. This is how a *module* is declared in SystemC. If there are generics declared in the entity then these will be printed just before the module as a `template`. The contents of the `constructor` stream is printed in the `SC_CTOR`. The remaining part of the module comes from the `body` stream.

4.1.2 The generated implementation file

There is also a .cpp file generated from the `implementation` stream. It is nothing more than a inclusion of the header of the module and the contents of the stream. All

required headers are included in the header of the module and so there is no need to include them in the implementation file.

4.2 Ports and Signals

In a module ports and signal are very common. A port declares an input or an output of a module and a signal declares an internal wire. SystemC has special structures to declare the ports and the signals. These are `sc_in`, `sc_out`, `sc_inout` and `sc_signal`. For example `sc_in<int >` declares an input of type int. The space in front of the `>` is generated to prevent collisions with types like `sc_int<4>`. Without space it would become `sc_in<sc_int<4>>` and that gives a syntax error because the `>>` is recognized as a right-shift operator.

It is not allowed in SystemC to declare a signal of an array type. An array of signals had to be generated for this case.

4.3 Subprograms

Functions and procedures (called *subprograms* together) are of great use in many algorithms. With them it is possible to design and test an algorithm in smaller reusable parts. In SystemC these are generated as ordinary member functions of the module. Nesting them is not allowed in SystemC. To remove the nesting the subprograms (and processes) are first generated in a temporary stream and at the end this stream is copied to the global implementation stream.

4.4 Processes

A process is the most important concurrent statement in a module. It contains a list of sequential statements that describe the behaviour of the module. It is the bridge between signals/ports and the algorithm that the module implements. In VHDL there exist some other types of concurrent statements like a concurrent signal assignment but with the exception of the component instantiation statement they are all converted by the parser to an equivalent process statement. This conversion is called *transmutation*.

There are 2 types of processes: one with a *sensitivity list* and one without. In SystemC terminology the first one is called a *method* and the second one a *thread*.

4.4.1 Method

A method is executed if there is an event on a signal in the sensitivity list. In SystemC this is specified by writing them to a `sensitive` object in the constructor like the example in Listing 4.2

Listing 4.2: Constructing a method

```
SC_METHOD(process);
sensitive << a << b;
```

The process itself is just an ordinary member function of the module like the procedure from Section 4.3.

4.4.2 Thread

The behaviour of a thread is slightly different from a process with a sensitivity list in VHDL. In both cases they are unconditionally executed at the start of the simulation, but in SystemC it is executed only once. In VHDL on the other hand the process starts at the beginning again after it has been completed. To mimic this behaviour in SystemC the contents of the process are put inside a never ending loop. This can be seen in the example in Listings 4.3 and 4.4. There you can also see the result of the transmutation that has been done by the parser.

Listing 4.3: Thread in VHDL

```
entity and2 is
  port (a,b: in bit;
        y: out bit);
end and2;

architecture dataflow of and2 is
begin
  y <= a and b;
end dataflow;
```

Listing 4.4: Thread in SystemC

```
// and2.h
#ifndef AND2_H
#define AND2_H
#include <systemc.h>
SC_MODULE(and2) {

  SC_CTOR(and2) {
    SC_THREAD(id_12);
  }
  sc_in<sc_bit > a;
  sc_in<sc_bit > b;
  sc_out<sc_bit > y;
  void id_12();

};
```

```
#endif /* AND2_H */

// and2.cpp
#include "and2.h"

void and2::id_12() {
  while(true) {
    y.write((a.read() & b.read()));
    wait(a.value_changed_event() | b.value_changed_event())
       ;
  };
}
```

4.5 Sequential Statements

Sequential statements describe the steps in an algorithm. A sequential statement can be as simple as a variable assignment or a procedure call and as complex as a case statement. Some simple ones were already implemented in the example that came with the parser so this section will only cover the more complex for loop statement, case statement, wait statement and signal assignment statement.

4.5.1 For loop Statement

A for loop is a special kind of loop that iterates through a range. The direction of this range can be ascending (up) or descending (down). Based on this direction the right SystemC loop is generated. Listings 4.5 and 4.6 show this.

Listing 4.5: For loop in VHDL

```
for v in 1 to 10 loop
  null;
end loop;

for v in 10 downto 1 loop
  null;
end loop;
```

Listing 4.6: For loop in SystemC

```
for(int v = 1; v <= 10; v++) {
  ;
}
for(int v = 10; v >= 1; v--) {
  ;
}
```

4.5.2 Case Statement

The case statement enables you to test multiple options at the same time. If the type is an enumeration or a scalar type like an integer the case statement can be written in SystemC as a `switch`, otherwise it is generated as a sequence of if-else statements.

4.5.3 Wait Statement

The execution of a thread can be suspended with a wait statement. It can be specified with a condition using `wait_until(..)`, with a timeout clause or with a sensitivity list with signals that triggers the next execution (both using `wait(..)`). The member function `.value_changed_event()` of a signal gives access to the event in SystemC.

4.5.4 Signal assignment statement

Signals and ports act as envelopes that need some extra effort to access the value. Writing a value to them is done in SystemC with the `.write(..)` member function.

Writing to a part of a signal (e.g., writing a single `sc_logic` in a `sc_signal< sc_lv<WIDTH> >`) is not allowed in SystemC. A solution is reading and storing the whole signal inside a temporary variable; modifying a part of the temporary variable; and writing the whole temporary variable back to the signal [8].

This solution is implemented twice in our tool. For simple signals the temporary signals are declared and initialized at the beginning of the process or subprogram and written back at the end. In the case that the signal is coming from an array of signals the solution is applied in the same line. This is done because the index that determines which signal from the array is written may be unknown at the beginning of the process or subprogram.

The second implementation is not used in all cases because a subsequent write to a part of a signal would destroy the previous writes. This destruction is caused by the delta delay between writing to a signal and a change of its value. The old value is read over and over again in subsequent writes and after changing a part of it, this value is written back. The result is that only the last write changes the signal.

The problem still persists in the case of the array of signals and that is one reason why we have written a disclaimer in the manual (see Section A.4).

Listings 4.7 and 4.8 show everything discussed in this section. `s` is a signal and `a` is an array of signals. For `s` the first implementation of the solution is applied. The signal is stored in the variable `__s__` at the beginning of the process and is written back at the end. If one bit is accessed in `a` the second implementation is applied in the same line using the variable `__tmp__`. The line is too wide to be visible as a single line in the Listing but its statement block (`{..}`) is still recognizable.

The described problem is visible in the example. In the second statement block with `__tmp__` the old value of `a[w]` is read again. If it is written back at the end of the statement block, the work done in the first statement block with `__tmp__` is lost.

Applying the first solution to the array of signals is not possible because the index variable `w` has the wrong value at the beginning of the process.

Listing 4.7: Partly written signals in VHDL

```
subtype BYTE is std_logic_vector(7 downto 0);
type BYTE_ARRAY is array (0 to 3) of BYTE;
signal s : BYTE;
signal a : BYTE_ARRAY;
--
pss: process
  variable v : integer;
  variable w : integer;
begin
  v := 2;
  w := 3;

  s(v) <= '1';
  s(v + 1) <= '1';

  a(w)(v) <= '1' ;
  a(w)(v + 1) <= '1' ;

end process;
```

Listing 4.8: Partly written signals in SystemC

```
void sas::pss() {
  int v;
  int w;

  while(true) {
    BYTE __s__ = s.read();

    v = 2;
    w = 3;
    __s__[v] = SC_LOGIC_1;
    __s__[(v + 1)] = SC_LOGIC_1;
    {BYTE __tmp__ = a[w].read(); __tmp__[v] = SC_LOGIC_1; a
       [w].write(__tmp__);}
    {BYTE __tmp__ = a[w].read(); __tmp__[(v + 1)] =
       SC_LOGIC_1; a[w].write(__tmp__);}
    s.write(__s__);
  };
}
```

4.6 Expressions

The translation of most expressions is not worth mentioning. The only thing that is interesting is *Constant Folding*.

4.6.1 Constant Folding

During the evaluation of expressions the tool folds integer constants along operators + and −. Constant folding means that operations on integer literals are evaluated at compile time. It should not be confused with constant propagation—this technique replaces identifiers with its constant value [13].

The implemented constant folding is a more sophisticated one that performs constant folding across nodes in the abstract syntax tree. For example `((x + 1)+ (y + 2))` is folded to `(x + y + 3)`

The main reason to implement constant folding was the unreadable expression that was the result of the translation of a range to a size. You specify the number of elements or bits in an array in VHDL with a range like this: `variable x: bit_vector(size - 1 downto 0 );` A literal translation would become `sc_bv<(size - 1)- 0 + 1> x;` The constant folding optimizes this to the more readable `sc_bv<size> x;`

For the implementation a class `FOLD_INFO` was designed with an integer field for the folded value and two lists with strings that could not be parsed as an integer. One list is for the strings that are added and the other for the strings that are subtracted from the integer value. These lists store for example the variables that are used in the expression. The class also has member functions for adding and subtracting two `FOLD_INFO` objects.

The folding is done in the recursive function `const_fold(..)` that adds, subtracts, or creates `FOLD_INFO` objects and returns one with all the information of the folded expression. This object is written to the active stream by another member function.

4.7 STD and IEEE libraries

Support for the STD and IEEE libraries is reached by checking for these declarations or types and conditionally printing adapted SystemC code. Checking is done with the function `is_declared_as(..)`. For example if the type is declared as `STRING` in library `STD` and package `STANDARD` a `.substr(..)` is printed inside the generation of a slice reference. Listing 4.9 shows this piece of code from the tool.

Listing 4.9: Checking for a type in the STD library

```
if(is_declared_as(sr->subtype, "STD/STANDARD/STRING"))
{
  *pActiveStream << ".substr(";
  emit_normalized_index(offset, er->left);
  *pActiveStream << ", ";
  emit_number_of_elements(sr->range);
```

```
  *pActiveStream << ")";
}
else
{
  // other code
}
```

4.8 Components

Besides describing a model behaviorally it can also be specified structurally using components. A structural SystemC model has other (smaller) models as member fields. Signals act as wires and they are connected to the components inside the constructor. The headers with the description of the components are included at the top of the file. Listing 4.10 shows a SystemC header file generated from a structural model.

Listing 4.10: Structural SystemC model

```
#ifndef AND4_H
#define AND4_H
#include <systemc.h>
#include "and2.h"

SC_MODULE(and4) {
  SC_CTOR(and4) {
    gate1 = new and2("and2_gate1");
    gate1->a(a);
    gate1->b(b);
    gate1->y(s);

    gate2 = new and2("and2_gate2");
    gate2->a(c);
    gate2->b(d);
    gate2->y(t);

    gate3 = new and2("and2_gate3");
    gate3->a(s);
    gate3->b(t);
    gate3->y(y);

  }

  sc_in<sc_bit > a;
  sc_in<sc_bit > b;
  sc_in<sc_bit > c;
  sc_in<sc_bit > d;
```

```
  sc_out<sc_bit > y;
  sc_signal<sc_bit > t;
  sc_signal<sc_bit > s;
  and2 *gate1;
  and2 *gate2;
  and2 *gate3;

};

#endif /* AND4_H */
```

Similar to the difficulties with the signal assignment statement from Section 4.5.4 connecting parts of signals to components is not allowed. For signal parts that are used, new signals are created and they are connected to the partly used ones in newly created processes.

5 Validation and Verification

This chapter is about the *validation* and *verification* of the tool. Validation is checking if the tool meets the user's needs (i.e., is it the right thing?) and verification is checking if the tool is doing that without errors (i.e., is the thing right?).

The conclusion is that our tool meets most of the user's needs. The only thing we could not determine is if the output of our tool works well with a software part in SystemC. Our conclusion about the verification is that we do not have a formal proof that the tool ships without unintended errors but we are convinced that our focus on testing improved the quality dramatically.

5.1 Validation

As mentioned in Chapter 1 the tool is designed to convert VHDL to SystemC for the cases that a system in SystemC is needed for modeling a system with a software part, for a faster simulation, or because some tools only support SystemC.

In SystemC it should be easy to model a system with a software and a hardware part using Transaction Level Modeling (TLM). Implementing this whole design process with TLM was considered too time consuming for this project. There is also more to investigate about TLM, like the easiness of using it. See the future work in Section 6.

Ignoring TLM leaves us the need for simulation and the support for other tools. To validate that our tool is suitable for simulation we have converted a whole i8051 and simulated the resulting SystemC code (see Section 5.1.1). The support of other tools is ensured by sticking to SystemC output and not using any C++ tricks.

5.1.1 Simulating an Intel 8051

To validate that our tool can be used to simulate VHDL with the SystemC simulation kernel we have converted a VHDL model of an Intel 8051. The model is made by the University of California at Riverside [18].

In these simulations the 8051 runs a small software program that computes the Fibonacci numbers. It is missing the first number (0), but that is a bug in the software on the 8051.

The VHDL simulation is done with `ModelSim` and is shown in Figure 5.1. The figure does not show the beginning of the simulation because there is no output there. It is all initialization time during which the 8051 is busy writing zeros to its 128 bytes of internal memory.
The VHDL was converted using our tool to SystemC and after adding the signals to trace in a `sc_main`, the files where compiled with `g++` and linked with SystemC library

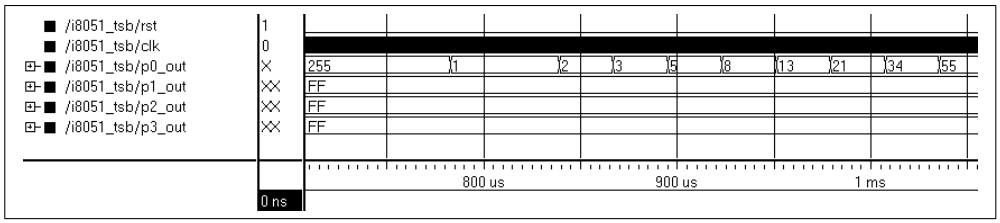


Figure 5.1: VHDL simulation of the original 8051

version 2.2.0. Running the created executable is simulating the model. The results that were visualized with `gtkwave` are visible in Figure 5.2. See the manual in Appendix A for the precise steps to do it yourself.

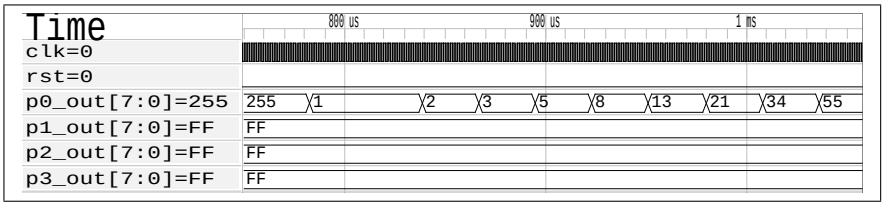


Figure 5.2: SystemC simulation of the converted 8051

The VHDL and SystemC simulation results are identical. We have shown that the SystemC code produced by our tool can be used for simulation.

5.2 Verification

Verification is—unlike the commonly forgotten validation—a very popular theme in the literature for some decades. The topics reach from the academic but impractical *formal verification* to the more useful *testing*. It was clear from the beginning that we were going to design a real program and thus we have limited ourselves to testing.

Testing was a major issue during the project. We have even made the tests before we started implementing a feature. This way of constructing software is called *Test-Driven Development*. We have also done *Regression testing*[14] to make sure that newly added code did not break the old one. The regression test was implemented with two `bash` scripts. One for converting a file with our tool and trying to compile the generated output with `g++` and the other for running the first script for all test files and reporting the number of failures. See Appendix B for the listings of the scripts. The test files themselves contain many of the VHDL constructs we learned during our studies.

6 Summary and Future Work

We have designed a tool that performs the translation of VHDL to SystemC automatically and we have shown in Chapter 5 that the tool is quite useful. The tool does not support all VHDL constructs. See Table A.1 for a list of the most common missing features. Supporting these constructs takes a lot of effort. We doubt it is worth the effort. That is why we will not give the popular advice to implement the missing features.

However, there are some things that came across during the project:

- The designers of SystemC have the rather stupid idea that the standard should evolve. Evolving is quite natural for software programs and also our tool evolved during the project to become better and better each week. But the purpose of standardization is that the standard is frozen so that tools of different suppliers made in different years can communicate with each other. This is not case with the evolving SystemC. For example the `sc_bit` and `sc_bv<WITDH>` that are used in this tool are already deprecated and will probably be removed from the standard in the next future [9]. If that happens the tool needs to be modified. And if something like this happens again in the future, then it needs to be modified again. And again and again until the standard becomes stable.

- As already mentioned in Section 5.1 it should be easy to model a system with a software and a hardware part using Transaction Level Modeling (TLM). The idea is that first the behavior of the whole system is modeled, then the system is split in a software and hardware part with *transactions* to communicate between the parts and as last step the software part is replaced by a microcontroller with the software in its rom, the hardware part by something that is synthesizable, and the transactions are replaced by busses or other real communication links.

  We did not have time to design a whole system with it, but we still want to know if it is useful in practice. We think it is a good idea to work this out and to compare it with the old way of designing with VHDL.

Bibliography

[1] *GHDL home page*, http://ghdl.free.fr/.

[2] *The FreeHDL Project*, http://freehdl.seul.org/.

[3] *VAUL - a VHDL Analyzer and Utility Library*, http://www-dt.e-technik.uni-dortmund.de/~mvo/vaul/.

[4] *Behavioural languages - Part 1-1: VHDL language reference manual*, 2004.

[5] *IEEE Std 1364 - 2005 IEEE Standard for Verilog Hardware Description Language*, 2006.

[6] *IEEE Std 1666 - 2005 IEEE Standard SystemC Language Reference Manual*, 2006.

[7] *IEEE Standard VHDL Language Reference Manual Amendment 1: Procedural Language Application Interface*, Sept. 5 2007.

[8] J. Bhasker, *A SystemC Primer*, Star Galaxy Publishing, 2004.

[9] David C. Black and Jack Donovan, *SystemC: From The Ground Up*, Springer, 2004.

[10] C. Dawson, S. K. Pattanam, and D. Roberts, *The Verilog Procedural Interface for the Verilog Hardware Description Language*, Proc. IEEE International Verilog HDL Conference, 26–28 Feb. 1996, pp. 17–23.

[11] Università di Verona, *An Overview on VHDL2SC1.0 Translator*.

[12] A. Fin, F. Fummi, and G. Pravadelli, *AMLETO: a multi-language environment for functional test generation*, Proc. International Test Conference, 30 Oct.–1 Nov. 2001, pp. 821–829.

[13] Dick Grune, Henri E. Bal, Ceriel J. H. Jacobs, and Koen Langendoen, *Modern Compiler Design*, John Wiley, 2002.

[14] C. Kaner, J. Falk, and H.Q. Nguyen, *Testing Computer Software*, Wiley, 1999.

[15] J. Lopez, G. Ricalde, A. Garcia, L. Entrena, J. Goicolea, and S. Olcoz, *Integrating tools in a VHDL framework*, Proc. VHDL International Users Forum. Spring Conference, 1–4 May 1994, pp. 154–162.

[16] Lubos Lorenc, Rudolf Schönecker, and Zbynek Krivka, *A Note on the Parsing of Complete VHDL-2002*, Workshop on Formal Models (Alica Kelemenová, Dusan Kolar, and Alexander Meduna, eds.), CEUR Workshop Proceedings, vol. 255, CEUR-WS.org, 2007.

[17] R. Maciel, B. Albertini, S. Rigo, G. Araujo, and R. Azevedo, *A Methodology and Toolset to Enable SystemC and VHDL Co-simulation*, Proc. IEEE Computer Society Annual Symposium on VLSI ISVLSI '07, 9–11 March 2007, pp. 351–356.

[18] University of California at Riverside, *Synthesizable VHDL Model of 8051*, http://www.cs.ucr.edu/~dalton/i8051/i8051syn/.

[19] John Willis, *Advanced Intermediate Representation with Extensibility / Common Environment*, 1999.

[20] P.A. Wilsey, D.E. Martin, and H. Hirsch, *The SAVANT project*, Proc. IEEE 1998 National Aerospace and Electronics Conference NAECON 1998, 13–17 July 1998, pp. 537–544.

[21] P.A. Wilsey, D.E. Martin, and K. Subramani, *SAVANT/TyVIS/WARPED: components for the analysis and simulation of VHDL*, Proc. International Verilog HDL Conference and VHDL International Users Forum IVC/VIUF, 16–19 March 1998, pp. 195–201.

Manual

A

This appendix is the manual for configuring (A.1) and using (A.2) the tool. The configuration takes some time but needs to be done only once. Also don't forget to check the known issues (A.3) and the disclaimer (A.4).

A.1 Configuration

The configuration procedure consists of 4 stages. First the tool needs to be *unpacked*, then it needs to be *set up* for the build. The *build* is next and after the build it is wise to *test* if the tool has been configured correctly.

A.1.1 Unpack

The tool is distributed in a compressed archive called `vtosc_dist.tgz`. The files are already in the folder `vtosc_dist` inside the archive. Unpack the archive using your favorite archiving software to the place where you want the tool to be generated. For instance in a command prompt you can run

```
tar xzf vtosc_dist.tgz
```

As said before the files are unpacked in the folder `vtosc_dist`. Inside you will find a build script, a folder with some test files, a folder with the source code of the back-end of the tool that has been written for this project and a compressed archive with the FreeHDL analyzer [2].

A.1.2 Setup

The `systemc` variable in the build script may need to be adapted to make the tool work in your environment. The variable contains the SystemC installation folder. Its current value is the default folder of the latest SystemC release (`/usr/local/systemc-2.2`). You should change this if you are using an older version of SystemC or if you have not installed SystemC in the default folder.

A.1.3 Build

If the build script contains the correct SystemC location, you have to run it:

```
cd vtosc_dist
./build
```

After running the build script there are two more things to find in the main folder. The folder `vlib` with the *std* and *ieee* vhdl libraries and the link `vtosc` to the compiler. Both are needed for the correct operation of the tool. More about them in the section about the usage of the tool (A.2).

A.1.4 Test

It is a good time to test the tool. Just go to the folder `test` and run the `test_all` script:

```
cd test
./test_all
```

All tests should pass. The most common cause for not passing is that SystemC cannot be found. If that is the case, make sure that the `systemc` variable in the `build` script is pointing to the correct folder and run the script again. (Don't worry, it is much faster the second time.)

If you want to test some more you can also convert and simulate the 8051 from Section 5.1.1:

```
cd i8051
./test_i8051
```

This simulation generates a `trace.vcd` file that can be viewed by `gtkwave` or similar programs. Note that the SystemC code to generate this trace file was made by hand and was included in the distribution. It was not generated by the tool. Keep this in mind if you want to simulate the files generated by the tool.

A.2 Usage

To use the tool from another folder 2 symbolic links need to be created. After their creation using the tool is straightforward.

A.2.1 Create Links

For the correct operation of the tool there are symbolic links needed to `vlib` and `vtosc`. Creating them is done by the following commands if `/home/vtosc_dist` is the main folder of the tool:

```
ln -s /home/vtosc_dist/vlib vlib
ln -s /home/vtosc_dist/vtosc vtosc
```

A.2.2 Run

If the links are made, running the tool is as simple as:

```
./vtosc file.vhd
```

The files that are generated depend on the name of the *entity* or *package* in that file. For every entity or package `name` there are always two files generated, namely `name.h` and `name.cpp`.

A.3 Known Issues

The most common unsupported constructs and possible solutions to resolve them are given in Table A.1. Everywhere the table reads signals, it applies to ports too.

Unsupported construct	Possible solution
Composite signal that is driven in parts by multiple processes	Write to helper signals inside the processes and let a newly inserted process write all helper signals to the composite signal.
Signal with multiple drivers that is needed for a bus	Replace the signal with its *resolved* cousin, like `sc_signal_rv<WIDTH>`.
Generate statement	Place multiple component instantiations inside an array. Construct a for-loop inside the `SC_CTOR` to connect them all.
Unconstrained array	Create a *template*.

Table A.1: Known issues and (hopefully) a way to resolve them

A.4 Disclaimer

There is no guarantee that the tool works correctly and that the generated files are correct. Some vhdl constructs are even known to give incorrect results (see Section A.3).

Test Scripts

B

The regression test was implemented with these two `bash` scripts. `test_file` for converting a file with our tool and trying to compile the generated output with `g++` and `test_all` for running the first script for all test files and reporting the number of failures.

Listing B.1: test_file

```
#!/bin/bash
SYSTEMC=/usr/local/systemc-2.2

if [ "$1" = "" ]; then
  echo  "ERROR: No input file specified!"
  exit 1
fi

echo "Converting $1 to SystemC..."
cmd="./vtosc "$@""
echo $cmd
$cmd
if [ $? -ne 0 ]; then
  echo "ERROR: Failed to convert $1 to SystemC!"
  exit 2
fi

ERROR=0
for f in $(./vtosc "$@" 2>&1 | awk '/.* .* writing .*/ {
   print($4)}'); do
  echo "Trying to compile the generated $f..."
  cmd="g++ -g -Wall -I$SYSTEMC/include -c $f"
  echo $cmd
  $cmd
  if [ $? -ne 0 ]; then
    ERROR=3
    echo "ERROR: Failed to compile the generated $f!"
  fi
done

exit $ERROR
```

Listing B.2: test_all

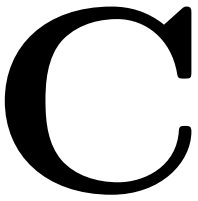
```
#!/bin/bash
echo "Testing multiple vhdl files with commandline options
   \""$@"\"..."

TESTS=(ttt_kort.vhdl ttt.vhdl component.vhdl expr.vhdl seq.
   vhdl datatype.vhdl libieee.vhdl pack.vhdl signl.vhdl
   spike.vhd spike_tb.vhd Adder.vhd Example.vhd Example_tb.
   vhd)

FAILURES=0
SUCCESS=0

for f in ${TESTS[@]}; do
  ./test_file "$@" "$f"
  if [ $? -ne 0 ]; then
    echo "FAILED"
    FAILURES=$(expr $FAILURES + 1)
  else
    echo "SUCCESS"
    SUCCESS=$(expr $SUCCESS + 1)
  fi
  echo
done
echo "TOTAL: "$SUCCESS"x SUCCESS & "$FAILURES"x FAILED"
exit $FAILURES
```

A Typical VHDL Model Converted



This Appendix shows the before and after listings of a typical Register Transfer Level (RTL) model in VHDL. The model is executing the following algorithm:
`if(a <= b)y = a + b; else y = a - b;`

Listing C.1: Input: Adder.vhd

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_signed.ALL;

ENTITY Adder IS
  PORT (A: in std_logic_vector(31 DOWNTO 0);
        B: in std_logic_vector(31 DOWNTO 0);
        Subtract: in std_logic;
        Sum: out std_logic_vector(31 DOWNTO 0);
        Sign: out std_logic
  );
END Adder;


ARCHITECTURE Behav OF Adder IS
  SIGNAL Tmp: std_logic_vector(31 DOWNTO 0);
BEGIN
  Sign <= Tmp(31);
  Sum <= Tmp;

  PROCESS(Subtract, A, B)
  BEGIN
    IF Subtract = '1' THEN
      Tmp <= A - B;
    ELSE
      Tmp <= A + B;
    END IF;
  END PROCESS;
END Behav;
```

Listing C.2: Output Header: adder.h

```
#ifndef ADDER_H
#define ADDER_H
#include <systemc.h>

SC_MODULE(adder) {
  SC_CTOR(adder) {
    SC_THREAD(id_18);

    SC_THREAD(id_19);

    SC_METHOD(id_21);
    sensitive << Subtract << A << B;

  }

  sc_in<sc_lv<32> > A;
  sc_in<sc_lv<32> > B;
  sc_in<sc_logic > Subtract;
  sc_out<sc_lv<32> > Sum;
  sc_out<sc_logic > Sign;
  sc_signal<sc_lv<32> > Tmp;
  void id_18();
  void id_19();
  void id_21();

};

#endif /* ADDER_H */
```

Listing C.3: Output Implementation: adder.cpp

```
#include "adder.h"

void adder::id_18() {
  while(true) {
    Sign.write(Tmp.read()[31]);
    wait(Tmp.value_changed_event());
  };
}

void adder::id_19() {
  while(true) {
    Sum.write(Tmp.read());
    wait(Tmp.value_changed_event());
```

```
  };
}

void adder::id_21() {
  if((Subtract.read() == SC_LOGIC_1)) {
    Tmp.write((A.read().to_int() - B.read().to_int()));
  }
  else {
    Tmp.write((A.read().to_int() + B.read().to_int()));
  }
}
```

Listing C.4: Input: Example.vhd

```
-- if(a <= b) y = a + b; else y = a - b;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY Example IS
  PORT (A: in std_logic_vector(31 DOWNTO 0);
        B: in std_logic_vector(31 DOWNTO 0);
        Clk: in std_logic;
        Reset: in std_logic;
        Y: out std_logic_vector(31 DOWNTO 0);
        Done: out std_logic
  );
END Example;

ARCHITECTURE Behav OF Example IS
  COMPONENT Adder
    PORT (A: in std_logic_vector(31 DOWNTO 0);
          B: in std_logic_vector(31 DOWNTO 0);
          Subtract: in std_logic;
          Sum: out std_logic_vector(31 DOWNTO 0);
          Sign: out std_logic
    );
  END COMPONENT;

  TYPE state_type is ( R, S0, S1a, S1b, S2);

  SIGNAL State, Next_State : state_type;

  SIGNAL Adder_A: std_logic_vector(31 DOWNTO 0);
  SIGNAL Adder_B: std_logic_vector(31 DOWNTO 0);
```

```
  SIGNAL Adder_Subtract: std_logic;
  SIGNAL Adder_Sum: std_logic_vector(31 DOWNTO 0);
  SIGNAL Adder_Sign: std_logic;

  SIGNAL RegA_In, RegA_Out: std_logic_vector(31 DOWNTO 0);
  SIGNAL RegB_In, RegB_Out: std_logic_vector(31 DOWNTO 0);
  SIGNAL RegY_In, RegY_Out: std_logic_vector(31 DOWNTO 0);

BEGIN

  ADDER_1: Adder PORT MAP(Adder_A, Adder_B, Adder_Subtract,
      Adder_Sum, Adder_Sign);


  STATES: PROCESS(State, A, B, RegA_Out, RegB_Out, RegY_Out
     , Adder_Sum, Adder_Sign)
  BEGIN
    CASE State IS
      WHEN R =>
        RegA_In <= A;
        RegB_In <= B;
        -- unused ADDER
        Adder_A <= RegA_Out;
        Adder_B <= RegB_Out;
        Adder_Subtract <= '0';
        -- still alive or unused register(s)
        RegY_In <= RegY_Out;
        --
        Done <= '0';
        Next_State <= S0;
      WHEN S0 =>
        Adder_A <= RegB_Out;
        Adder_B <= RegA_Out;
        Adder_Subtract <= '1';
        -- still alive or unused register(s)
        RegA_In <= RegA_Out;
        RegB_In <= RegB_Out;
        RegY_In <= RegY_Out;
        --
        Done <= '0';
        IF (Adder_Sign = '1') THEN
          Next_State <= S1b;
        ELSE
          Next_State <= S1a;
        END IF;
```

```
    WHEN S1a =>
      Adder_A <= RegA_Out;
      Adder_B <= RegB_Out;
      Adder_Subtract <= '0';
      RegY_In <= Adder_Sum;
      -- still alive or unused register(s)
      RegA_In <= RegA_Out;
      RegB_In <= RegB_Out;
      --
      Done <= '0';
      Next_State <= S2;
    WHEN S1b =>
      Adder_A <= RegA_Out;
      Adder_B <= RegB_Out;
      Adder_Subtract <= '1';
      RegY_In <= Adder_Sum;
      -- still alive or unused register(s)
      RegA_In <= RegA_Out;
      RegB_In <= RegB_Out;
      --
      Done <= '0';
      Next_State <= S2;
    WHEN S2 =>
      -- unused ADDER
      Adder_A <= RegA_Out;
      Adder_B <= RegB_Out;
      Adder_Subtract <= '0';
      -- still alive or unused register(s)
      RegA_In <= RegA_Out;
      RegB_In <= RegB_Out;
      RegY_In <= RegY_Out;
      --
      Done <= '1';
      Next_State <= S2;
  END CASE;
END PROCESS STATES;


MAIN: PROCESS(Reset, Clk)
BEGIN
  IF (Reset = '1') THEN
    RegA_Out <= (OTHERS => '0');
    RegB_Out <= (OTHERS => '0');
    RegY_Out <= (OTHERS => '0');
    State <= R;
```

```
    ELSIF (Clk'EVENT AND Clk = '1') THEN
      RegA_Out <= RegA_In;
      RegB_Out <= RegB_In;
      RegY_Out <= RegY_In;
      State <= Next_State;
    END IF;
  END PROCESS MAIN;


  -- connect output
  Y <= RegY_out;

END Behav;
```

Listing C.5: Output Header: example.h

```
#ifndef EXAMPLE_H
#define EXAMPLE_H
#include <systemc.h>
#include "adder.h"

SC_MODULE(example) {
  SC_CTOR(example) {
    ADDER_1 = new adder("Adder_ADDER_1");
    ADDER_1->A(Adder_A);
    ADDER_1->B(Adder_B);
    ADDER_1->Subtract(Adder_Subtract);
    ADDER_1->Sum(Adder_Sum);
    ADDER_1->Sign(Adder_Sign);

    SC_METHOD(STATES);
    sensitive << State << A << B << RegA_Out << RegB_Out <<
        RegY_Out << Adder_Sum << Adder_Sign;

    SC_METHOD(MAIN);
    sensitive << Reset << Clk;

    SC_THREAD(id_130);

  }

  sc_in<sc_lv<32> > A;
  sc_in<sc_lv<32> > B;
  sc_in<sc_logic > Clk;
  sc_in<sc_logic > Reset;
```

```
  sc_out<sc_lv<32> > Y;
  sc_out<sc_logic > Done;
  typedef enum { R, S0, S1a, S1b, S2 } state_type;
  sc_signal<state_type > Next_State;
  sc_signal<state_type > State;
  sc_signal<sc_lv<32> > Adder_A;
  sc_signal<sc_lv<32> > Adder_B;
  sc_signal<sc_logic > Adder_Subtract;
  sc_signal<sc_lv<32> > Adder_Sum;
  sc_signal<sc_logic > Adder_Sign;
  sc_signal<sc_lv<32> > RegA_Out;
  sc_signal<sc_lv<32> > RegA_In;
  sc_signal<sc_lv<32> > RegB_Out;
  sc_signal<sc_lv<32> > RegB_In;
  sc_signal<sc_lv<32> > RegY_Out;
  sc_signal<sc_lv<32> > RegY_In;
  adder *ADDER_1;
  void STATES();
  void MAIN();
  void id_130();

};

#endif /* EXAMPLE_H */
```

Listing C.6: Output Implementation: example.cpp

```
#include "example.h"

void example::STATES() {
  switch(State.read()) {
  case R:
    RegA_In.write(A.read());
    RegB_In.write(B.read());
    Adder_A.write(RegA_Out.read());
    Adder_B.write(RegB_Out.read());
    Adder_Subtract.write(SC_LOGIC_0);
    RegY_In.write(RegY_Out.read());
    Done.write(SC_LOGIC_0);
    Next_State.write(S0);
    break;
  case S0:
    Adder_A.write(RegB_Out.read());
    Adder_B.write(RegA_Out.read());
    Adder_Subtract.write(SC_LOGIC_1);
```

```
    RegA_In.write(RegA_Out.read());
    RegB_In.write(RegB_Out.read());
    RegY_In.write(RegY_Out.read());
    Done.write(SC_LOGIC_0);
    if((Adder_Sign.read() == SC_LOGIC_1)) {
      Next_State.write(S1b);
    }
    else {
      Next_State.write(S1a);
    }
    break;
  case S1a:
    Adder_A.write(RegA_Out.read());
    Adder_B.write(RegB_Out.read());
    Adder_Subtract.write(SC_LOGIC_0);
    RegY_In.write(Adder_Sum.read());
    RegA_In.write(RegA_Out.read());
    RegB_In.write(RegB_Out.read());
    Done.write(SC_LOGIC_0);
    Next_State.write(S2);
    break;
  case S1b:
    Adder_A.write(RegA_Out.read());
    Adder_B.write(RegB_Out.read());
    Adder_Subtract.write(SC_LOGIC_1);
    RegY_In.write(Adder_Sum.read());
    RegA_In.write(RegA_Out.read());
    RegB_In.write(RegB_Out.read());
    Done.write(SC_LOGIC_0);
    Next_State.write(S2);
    break;
  case S2:
    Adder_A.write(RegA_Out.read());
    Adder_B.write(RegB_Out.read());
    Adder_Subtract.write(SC_LOGIC_0);
    RegA_In.write(RegA_Out.read());
    RegB_In.write(RegB_Out.read());
    RegY_In.write(RegY_Out.read());
    Done.write(SC_LOGIC_1);
    Next_State.write(S2);
    break;
  }
}

void example::MAIN() {
```

```
  if((Reset.read() == SC_LOGIC_1)) {
    RegA_Out.write(sc_lv<32>(SC_LOGIC_0));
    RegB_Out.write(sc_lv<32>(SC_LOGIC_0));
    RegY_Out.write(sc_lv<32>(SC_LOGIC_0));
    State.write(R);
  }
  else if((Clk.event() && (Clk.read() == SC_LOGIC_1))) {
    RegA_Out.write(RegA_In.read());
    RegB_Out.write(RegB_In.read());
    RegY_Out.write(RegY_In.read());
    State.write(Next_State.read());
  }
}

void example::id_130() {
  while(true) {
    Y.write(RegY_Out.read());
    wait(RegY_Out.value_changed_event());
  };
}
```