2011

Synthesizable SystemC to VHDL Compiler Design

Rui Chen

THE FLORIDA STATE UNIVERSITY

COLLEGE OF ENGINEERING

SYNTHESIZABLE SYSTEMC TO VHDL COMPILER DESIGN

By

RUI CHEN

A Thesis submitted to the
Department of Electrical and Computer Engineering
in partial fulfillment of the
requirements for the degree of
Master of Science

Degree Awarded:
Fall Semester, 2011

Rui Chen defended this thesis on November 3, 2011.

The members of the supervisory committee were:

Uwe Meyer-Baese
Professor Directing Thesis

Simon Foo
Committee Member

Ming Yu
Committee Member

The Graduate School has verified and approved the above-named committee members, and certifies that the thesis has been approved in accordance with the university requirements.

ACKNOWLEDGMENTS

As the author of this thesis, I am keenly aware that it represents the fruition of not only my own work, but also the support which other individuals and organizations have lent me over the years, and for which I am profoundly grateful.

First, I would like to send my deepest gratitude to my supervisor, Dr. Uwe Mayer-Baese, whose precious guidance, support support and encouragement were pivotal in establishing my self-confidence in this endeavor. Also, I would like to thank Dr. Simon Foo and Dr. Ming Yu for spending their valuable time to be my committee members. I would like to thank Dr. Mei Su, my undergraduate advisor, who gave me great confidence and led me to the research fields. I would like to thank Dr. Yao Sun, I'd really appreciate the discussions with him in the college years. I would like to thank my roommates and friends, Wanjun Cao, Chongyue Yi, Luyang Wang, and Peipei Zhang. Without their help, I can hardly persist to the final stage. Finally, I would thank my parents for giving me the great opportunity to study at FSU, it is the greatest experience in my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ANTLR	ANother Tool for Language Recognizer
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
AST	Abstract Syntax Tree
BNF	Backus–Naur Form
CAD	Computer Aided Design
EDA	Electric Design Automation
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
IDE	Integrated Development Environment
IR	Intermediate Representation
RTL	Register Transfer Level

ABSTRACT

Efficient hardware/software co-design framework greatly facilitates not only the design but also the verification early in the embedded system deign cycle. With these electric design automation (EDA) tools, the hardware can be concisely and efficiently modeled at a higher abstraction level better than with the more traditional hardware description languages (HDLs). SystemC is such a hardware/software co-design language, which allows hardware models to be written in C++. Consequently, it is naturally needed to consider a hardware implementation for the verified SystemC models.

To solve this bottleneck problem, two design flow methodologies, namely, direct synthesis method and indirect synthesis method, are proposed in the literatures. Since the indirect synthesis method applies the traditional HDL synthesis technology as part of its design flow, it is considered as framework solution. As result, the compiler tool, which translates SystemC to VHDL, is presented in the thesis. The details of equivalent constructs between these two languages are illustrated. Besides performing obvious translation of many SystemC elements, the compiler focuses on the automated translation of control structures and complicated expressions. Finally, a series of digital application modules in SystemC were tested. Simulations and RTL views of the translated VHDL models yielded the expected results.

CHAPTER 1

INTRODUCTION

Part of embedded system developments, electric design automation (EDA) tools are always facing the dilemma between the fast design-to-market time and accurate and complete design description from users. As the core building brick, compiler is not only a bridge between software and hardware, but also highly affects the quality of the generated code. For example, with embedded C compilers, the code translation from C to assembly is very high efficient. The executive efficiency of C code is 95% functionally equivalent to the efficiency of corresponding hand-written assembly code. Unless you need to consider the time-critical tasks for the real-time systems, these compilers can greatly facilitate your design speed. With the rapid developments of application specific integrated circuits (ASICs) and filed field programmable gate arrays (FPGAs) these days, the software and hardware co-design platforms are attracting more and more attentions. The behind-scene idea is pretty simple, **one language for all designs**, which means the modeling capability at the different levels (including system level, algorithmic level, behavior level, register transfer level (RTL) level, and Gate level) should be integrated into one language. In this sense, our compiler translating synthesizable SystemC into VHDL is part of this big idea.

As known to all, compiler design is one of the most challenging engineering problems, but can be expressed in a simple way. It is a Turing machine, which generates the best matched output corresponding to the certain input pattern. In details, a compiler consists of a front end, which translates source code into some intermediate representation, and a back end, generating target code, which best represents the design intentions (shown in Figure 1.1). During the different phases of the compila-

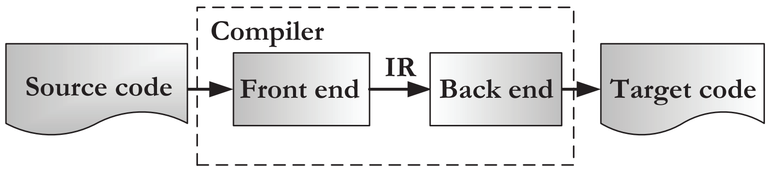


Figure 1.1: General structure for a compiler

tion process, one level or multilevel optimizations can be applied to increase the code efficiency and minimize the code size. Based on the levels of source language and target language, compilers can be classified as commonly-known compilers (compiling high level code to low level code), decompilers (compiling low level code to high level code), and translators (compilation between two high level languages). The developed SystemC to VHDL translator belongs to the translator category. In terms of the compilation passes, the compiler can also be divided into two classes, one-pass compiler and multi-pass compiler. The number of passes are generally related to the complexity of optimizations and the quality of the generated target code. One-pass compiler is quite less time consuming in generating the executable code, while multi-pass compiler can highly improve the quality (code size and executive efficiency) of the target code. In this sense, the multi-pass compiler with abstract syntax tree (AST) as the most common intermediate representation (IR) is adopted in the thesis.

1.1 SystemC

Although SystemC got the C label in its name, but it is actually a C++ class library with SystemC constructs. Since standardized in 2005 [16], it has been a favorable hardware design language in the software/hardware co-design field due to the simulation speed and flexible interface to the software design over the other languages (shown in Figure 1.2). When it comes to the synthesis, SystemC, like C++, is pretty

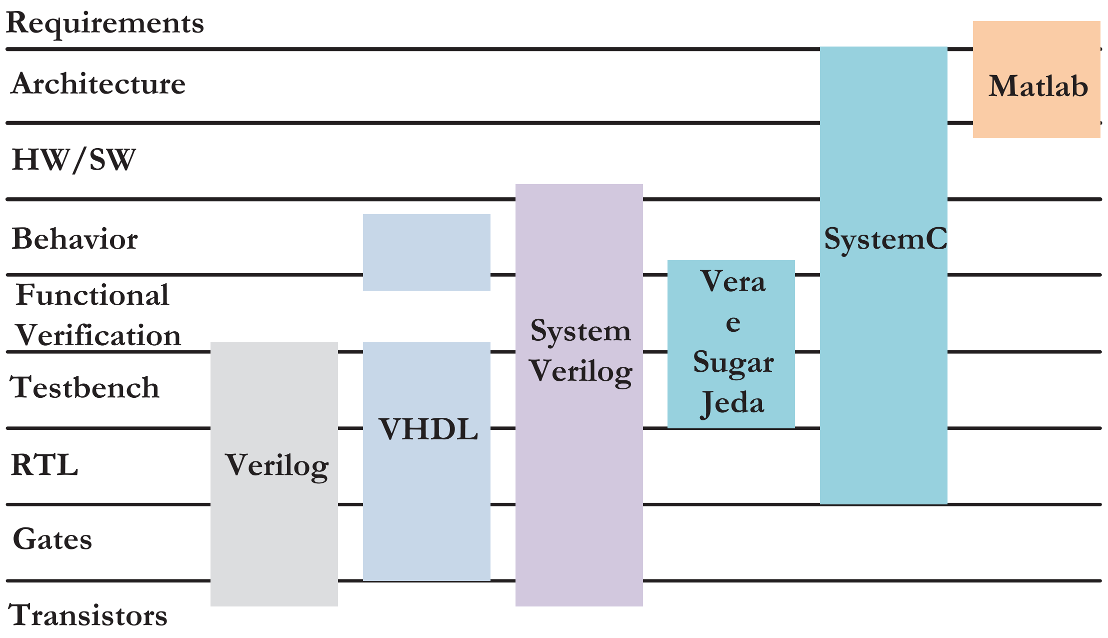


Figure 1.2: SystemC compared with different modeling languages

hard to get analyzed by the EDA tools as result of many language features, such as classes and templates. So, when considering hardware implementations of the SystemC designs, the Open SystemC Initiative (OSCI) synthesizable subset standard is considered [4][6]. Contrast to the more advanced direct synthesis, which compiles the system specification into the implementation netlist, the indirect synthesis design flow strategy, adopted in the thesis, translates the SystemC model into RTL Verilog/VHDL first and then uses the conventional RTL synthesis technology. As a result, the the indirect synthesis methodology make the existing EDA tool (such as Synopsys Design Compiler) better serve to generate the better netlist and further the GDSII [1].

1.2 VHDL

In the conventional hardware design field, VHDL and Verilog are two powerful and dominant hardware design languages. Although they have many things in common, and comparably equivalent at the most times, they are still two different languages with different focus levels (shown in Figure 1.3). It is well acknowledged that VHDL

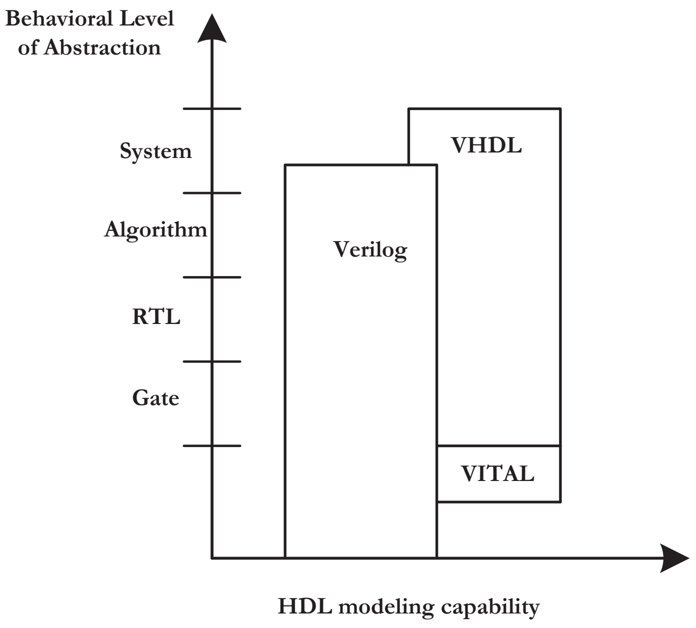


Figure 1.3: HDLs modeling capability comparison

good at system design, while Verilog has special language features for gate modeling. Based on this reason, VHDL is chosen as the target HDL language. Also the standards [14][15], especially the latest synthesizable subset of VHDL `IEEE Std 1076.6-2004`, are taken as reference to design the compiler.

[1]A database file format which is the de facto industry standard for data exchange of integrated circuit or IC layout artwork

1.3 Background and related work

Although compiler design, or compiler construction, is black art behind various kinds of EDAs or integrated development environments (IDEs), it not only alleviates the learning curve for new design language, but also increases the design efficiency to some degree. So it is quite common to see new compilation techniques year by year. The recent hot topic related to hardware and software co-design is high level synthesis, which aims to create a uniform framework that makes one language is enough for embedded system design. Obviously, SystemC, other than C or C++, is the best choice. However, it seems not a easy job. So, as simplified versions of SystemC, C and C++ got much analysis [5][8]. After years' efforts, the commercial tool from Synopsys finally made breakthroughs in 2004. Right now, Synopsys has two high level synthesis tools, namely, Synphony C Compiler and Synphony Model Compiler. Besides the Synopsys tools, there are still a bunch of commercial tools available, such as AutoPilot of AutoESL, Catapult C of Mentor Graphics. It also should be noted that these tools not only support SystemC, but also support C/C++.

In the open source community, the achievements are quite less profound, which makes the high level synthesis still mysterious to various application engineers. With open source tools sc2v [3], gsc, Pinapa [9], and PinaVM [7] as pioneers, the similar but different developing strategy is deployed in the thesis.

1.4 Problem formulation

Hardware/software co-design framework has become an important methodology in the rapid prototyping of embedded systems. The application of SystemC for such modeling has gained more and more supports among users; furthermore, a convenient and easy-to-learn synthesis strategy for SystemC to the hardware implementation would definitely strengthen the capability of SystemC in the design flow. In this thesis, the dedications for this problem can be expressed as:

- Identify the subset of SystemC that can be described using synthesizable VHDL. Then design and implement a tool to translate this subset to the synthesizable VHDL models.

The proposed compiler structure is shown in Figure 1.4.

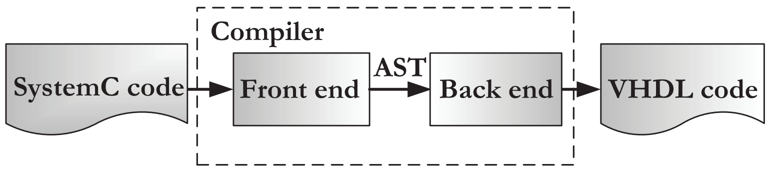


Figure 1.4: Proposed structure for SystemC to VHDL compiler

In the context of the thesis, synthesizable VHDL, IEEE1076.6- 2004 [15], is applied. The version of SystemC used in the thesis is SystemC 2.2 as described in [6].

1.5 Thesis Organization

In this section, the framework of the thesis is presented as follows:

- Chapter 2 analyzes SystemC as a language and identifies the input restrictions. The functionally equivalent constructs are discussed one by one.

- Chapter 3 provides the details of the implementation of the compiler tool using the ANother Tool for Language Recognition (ANTLR) toolset in Java.

- Chapter 4 contains the testing results for the developed compiler with three typical digital designs. The post-translation simulation results will also be provided to verify the correctness of the translations.

- Chapter 5, the final chapter, will conclude by presenting an overview of the results of this thesis and highlights some important points. The further prospects for the compiler developments is also presented.

CHAPTER 2

IDENTIFYING THE SIMILARITY

From the point of the low level netlist or even the GDSII, SystemC and VHDL are still two high level languages, which means they have many features in common. Before getting to the details of the compiler construction, it is quite necessary to identify the similar structures first. Firstly, a small example in SystemC is presented in Listing 2.1.

```
#include "systemc.h"
SC_MODULE (my_model)
{
        sc_in<sc_logic> input1;
        sc_in<sc_logic> input2;
        sc_out<sc_logic> output1;
        sc_out<sc_logic> output2;
        SC_CTOR  (my_model)
         {
             SC_METHOD ( process );
              sensitive << input1 << input2;
         }
          void process( )
          {
                  sc_logic my_var1, my_var2;
                   my_var1 = ~input1;
                   my_var2 = ~input2;
                   output1 = input1 & my_var2;
                   output2 = input2 & my_var1;
           }
};
```

Listing 2.1: A small SystemC program

In terms of a successful translation, it always means the best solution in target language can be found to match with the design intentions of source code. In this sense, a potential translation option (shown in Listing 2.2) as target is meaningful and helpful for developing the compiler.

```
library IEEE;
use IEEE.std_logic_1164.all;
entity my_model is
               port(
               input1: in STD_LOGIC;
               input2: in STD_LOGIC;
               output1: out STD_LOGIC;
               output2: out STD_LOGIC;
               );
end my_model;

architecture my_arch of my_model is
begin
  process( input1, input2)
        variable my_var1,my_var2: STD_LOGIC;
  begin
        my_var1 :=  not input1;
        my_var2 :=  not input2;
        output1 <= input1 and my_var2;
        output2 <= input2 and my_var1;
  end process;
end my_arch;
```

Listing 2.2: The possible translation of the sample SystemC program

With the small example, the general structural comparison between the two languages (shown in Figure 2.1) can be easily derived. the comparison would assist the compiler to solve the scope issues.

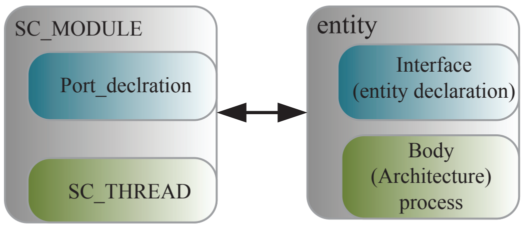


Figure 2.1: Structural comparison between the two languages

Actually, this is the traditional style, one of the two design styles. The other one is recommended style, which separates the declaration and implementation, just like the normal C++ code would do [2]. The two design styles seems quite similar to each other, actually it is. But when separating the declaration and implementation, the compiler must handle the include directive in the preprocessing phase, which leads to the extra pass of the compiler construction. For example, sc2v use two different passes to parse the header and source source file separately with two different compiler, namely `sc2v_step1` and `sc2v_step2`. The reason is that some lexical tokens are ambiguous meanings in either header or source file (like `<<` could be either "shift left" or "two left angle brackets"). In the thesis, the recommended design style as well as traditional design style are both supported. However, for simplifying the compiler design, the include directives are manually operated instead of preprocessing by the compiler.

Another application of include directive is for the hierarchy design, which means the implementation is constructed in several modules. In SystemC, the hierarchy design is conducted as follows: firstly, construct the submodules in different headers and source files; then include the submodule declaration headers into the main module header; finally, make the port association in the CTOR construct of the main module. The VHDL design is also quite similar, the difference is you use the package directive instead of include directive. Normally, package declaration of submodules is specified in a sperate VHDL source file for convenience.

2.1 Data types

Aside from the general structure, the main concern is the data types of the language. Since SystemC is a class library to C++, it inherits from all the built-in data types of C++. However, as a hardware design language, it also supports many hardware features. The data types shown in SystemC in Table 2.1 are derived from [6].

Table 2.1: Hardware data types supported in SystemC

SystemC Data Type	Description
`sc_int<const>`	width restricted integer
`sc_uint<const>`	width restricted unsigned integer
`sc_bigint<const>`	unrestricted integer
`sc_biguint<const>`	unrestricted unsigned integer
`sc_logic`	4-value logic
`sc_lv<const>`	`sc_logic` vector
`sc_bit`	bit
`sc_bv<const>`	`sc_bit` vector
`sc_fixed<>`	unrestricted fixed point
`sc_ufixed<>`	unrestricted unsigned fixed point

In VHDL, there are two standards related to the data types, namely, IEEE standard `1164-1993` [12] and IEEE standard `1076.3-1997` [13]. The first one introduces the `std_ulogic` and its subtype, `std_logic` inside the package `std_logic_1164`, while the latter one gives us more choices in handling the numeric and boolean values with bit and boolean types inside the package `numeric_std` and `numeric_bit`. The `std_logic_1164` and `numeric_bit` are the primitive packages; the `numeric_bit` is based on the `numeric_std` package. Before discussing the overlaps between these three packages, it is necessary to introduce the primitive `std_ulogc` type first. The nine different logic values indicate the signal's strength (shown in Listing 2.3); the values 'U', 'X' 'W' and '-' are considered as metalogic values, they define the behavior of the model itself rather than the behavior of the hardware being synthesized. The value 'U' represents the value of an object before explicitly assigned in the model; the values 'X' and 'W' represent forcing and weak values, respectively, for which the model is not able to distinguish between logic levels. And '-' is also called don't care value, which is used as "wild card" or "match all" token in `numeric_std` package.

```
TYPE std_ulogic IS (U, -- Uninitialized
                    X, -- Forcing Unknown
                    0, -- Forcing 0
                    1, -- Forcing 1
                    Z, -- High Impedance
                    W, -- Weak Unknown
                    L, -- Weak 0
                    H, -- Weak 1
                    - -- Dont care
                    );
```

Listing 2.3: The definition for `std_ulogic` in `std_logic_1164` package

Compared with the nine logic values of `std_ulogic`, bit and boolean are rather easy. The bit type has two choices, logic '0' or logic '1', while boolean also has two choices , true or false. So when you specify logic '1' and logic '0', the corresponding choices can be made from Table 2.2.

Table 2.2: Specify the logic '0' and '1' with the three different data types

logic **'0'**	bit value **'0'**
	boolean value **false**
	`std_ulogic` value **'0'** and **'L'**
logic **'1'**	bit value **'1'**
	boolean value **true**
	`std_ulogic` value **'1'** and **'H'**

It is easy to derive from the table that the `std_ulogic` can cover the data types, which bit and boolean can represent. Besides that, you can hardly specify the tri-state output with bit and boolean. So in practice, people choose `std_logic`, the resolved version of `std_ulogic` to specify the data types for convenience. In the thesis, the `std_logic` and `std_logic_vector` are used to translate the logic types, while the fixed point data types are simply ignored. In addition to the data types discussed

Table 2.3: Hardware data types between SystemC and VHDL

SystemC Data Type	VHDL counterpart
`sc_logic`, `sc_bit`	`std_logic`
`sc_int<const>`, `sc_bigint<const>`, `sc_bv<const>`, `sc_lv<const>`	`std_logic_vector`
`sc_uint<const>`, `sc_biguint<const>`	`unsigned std_logic_vector`

above, the compiler should also support the built-in C++ data types and user-defined data types. In the Table 2.4, the words `low` and `high` represent the lower and upper limits of the specified range, respectively. For example, the range of values for char is −128 to 127. The corresponding lower and upper limits for the VHDL integer can be low to −128 and high to 127, respectively.

Table 2.4: Built-int data types in SystemC

SystemC Data Type	VHDL counterpart
`char`	*integer* **range** (*low*(8) **to** *high*(8))
`int`	*integer* **range** (*low*(32) **to** *high*(32))
`long`	*integer* **range** (*low*(32) **to** *high*(32))
`bool`	*boolean*

The user-defined data types assist program developers to design in a more comfortable environment, for example, they can introduce the short names and new easy-to-identify data types. Since the target is VHDL, the user-defined data types can be generally divided into the two categories, namely, the scalar types and composite types. The scalar types consist of integer types, floating point types, physical types, and enumerated types, while composite types are composed of array types and record types. However, the floating point types and physical types are not supported by the synthesis program, only the rest four types are discussed here.

2.1.1 Scalar types

The most common use for integer types is to introduce the new data types, the syntax is normally like:

```
typedef type type_id;
```

Listing 2.4: Introduce new integer types

In the similar fashion, the corresponding VHDL type declaration is as follows:

```
    type type_id is type;
```

Listing 2.5: Corresponding VHDL type declaration

When introducing the new data types, one thing should be paid attention to is the scope. In SystemC, the scopes can be divided into the global and local scope, while the local scope can be further divided into the `sc_module` scope, `sc_method`(or `sc_thread`) subfunction scope. Correspondingly, in VHDL, there are four scopes, entity scope, architecture scope, subprogram scope and package scope. For here, just the single file compilation is discussed, so the file scope is not applied here. In sum, the scope comparisons of the two languages can be made in the following table.

Table 2.5: Scope comparisons between the two languages

SystemC Scope	VHDL counterpart
global	package
`sc_module`	entity
`sc_method`(or `sc_thread`)	architecture
global	package

It is also interesting to see the great similarity between SystemC and VHDL on enumeration types, but the semantic meanings are quite different. Compared with enumerations in VHDL, the SystemC enumerations is not really enumerations, but just pseudo enumerations. Also, the enumerations in SystemC is consistently assigned with an integer value from the beginning to the end. In comparison, the enumeration in VHDL is truly a value from a predefined set. In this sense, the SystemC enumeration type is a subset of VHDL enumeration type.

```
typedef enum type_id {element1, element2, ...};
```

Listing 2.6: Enumeration type in SystemC

```
type type_id is (element1, element2, ...);
```

Listing 2.7: Corresponding VHDL type declaration

In addition, it is also needed to note that the enumeration types are only applied in the package and entity scopes to assist the design.

2.1.2 Composite types

The composite types consist of array types and structure types. The array types in VHDL have two types, unconstrained types and constrained types, while the difference between the them is simply the upper and lower limit. In the thesis, only the constrained single and multiple dimension array types are considered. The translation can be easily made as the Listings 2.8 and 2.9. In the Listings, sup represents the upper limit, while inf represents the lower limit of array range. The indicators A and B are for the multiple dimension array version.

```
typedef array_type type_id [sup−inf+1];
typedef array_type type_id [supA−infA+1][supB−infB+1];
```

Listing 2.8: Both single and multiple dimension arrays in SystemC

Unlike SystemC, which just specifies the width of the vector (as shown above), VHDL needs to specify both the lower and upper bounds. Also, VHDL has two different indexing styles, namely, **to** and **downto**. Reverse range expression is applied here simply because the SystemC library uses the same convention of numbering vectors from right to left. And the corresponding VHDL translation is shown in Listing 2.9.

```
type type_id is array(inf to sup) of array_type;
type type_id is array(infA to supA, infB to supB) of
    array_type;
```

Listing 2.9: Corresponding VHDL array type declarations

In a similar way, let's start to look at the final data type, record type. The common structure types in SystemC is shown in Listing 2.10:

```
{
    subtype_indication identifier;
                :
    subtype_indication identifier;
};
struct type_id instance;
```

Listing 2.10: Structure types in SystemC

Since the struct is always associated with `type_id`, in practice, people always prefer to typedef the struct to actually introduce a new type. The modified structure declaration is shown in Listing 2.11.

```
typedef struct
{
    subtype_indication identifier;
                :
    subtype_indication identifier;
}type_id;
type_id instance;
```

Listing 2.11: More convenient structure types in SystemC

The above listing is also equivalent to apply the "**typedef** struct `type_id type_id`;" into the Listing 2.10.

```
type type_id is
    record
        identifier: subtype_indication;
                  :
        identifier: subtype_indication;
    end record;
instance: type_id;
```

Listing 2.12: Corresponding VHDL record type declarations

2.2 Statements

The most involved parts of translating SystemC to VHDL is to translate the statements. The statements in synthesizable subset of SystemC can be classified in Backus-Naur Form (BNF) as follows[6].

```
statement ::=
    labeled-statement
    | expression-statement
    | compound-statement
    | wait-statement
    | signal-assignment-statement
    | selection-statement
    | iteration-statement
    | jump-statement
    | declaration-statement
```

Listing 2.13: Statements classification in SystemC

While in essence, the major work is related to the two things, assignment statements and control flow statements.

2.2.1 Assignment statements

An assignment operation consists of three parts, the assigned identifier, assign operator and right hand side (RHS) expressions. Different from SystemC, the VHDL has two types of assign operators, `:=` and `<=`. The former operator goes with port and signal assignments, while the latter one is for variable assignments. The way of distinguishing the ports(signals) and variables during the translation is to firstly keep the declarations in different scope, and then check the type of assigned identifier in the code generation phase.

Although different, the assignment operators in SystemC are richer than VHDL, especially the compound assignment operators. The compound assignment operators are `=, *=, /=, %=, +=, -=, >>=, <<=, &=, ^=, |=`. The translation strategy is pretty straightforward, considering it as the combination of two operators, and disseminating it.

As for the RHS expressions, the logic between the two languages are quite the same, so the focus would be on the match of the operators. The operator comparison is shown in Table 2.6. In addition to the operators, there are many operators can

Table 2.6: Operator comparisons between the two languages

SystemC operators	VHDL counterparts	note
& | ∧ ∼	AND OR XOR NOT	bit-wise operators
|| && !	AND OR NOT	logic or operators
>> <<	srl sll	shift operators
== ! = < <= > >=	= / = < <= > >=	relational operators
+ -	+ -	unary operators
+ ∗ /	+ ∗ /	

hardly expressed in SystemC such as compound logic operators (nand nor xnor), and arithmetic and rotate shift operators (sla, sra, rol and ror). But the most impressive operator is modular operator, it has two variations, mod and rem. The difference between them can be shown in Listing 2.14:

```
A rem B = A − (A/B) ∗ B (in which A/B is an integer)
A mod B = A − B ∗ N      (in which N is an integer)
examples:
11 rem 4    =>    result: 3
−11 rem 4   =>    result: −3
9 mod 4     =>    result: 1
7 mod −4    =>    result: −1 (7 − 4∗2 =−1)
```

Listing 2.14: Two variations of modular operator

After the comparison, as the better choice, rem is chosen for '%'.

2.2.2 Control flow statements

The control flow statements are mostly involved with if-else statements, switch statements and iteration statements. They can stand alone to constitute a sequence of statements, or embedded into each other. However, the analysis style is pretty the same, the translation unit is the statement.

2.2.3 If-else statements

Depending on the truth value of one more conditions, the if-else statements executes a sequence of statements. the if-else statement in SystemC is shown in Listing 2.15:

```
if (condition)
    statement
[else
    statement]
```

Listing 2.15: If-else statement in SystemC

The above listing is pretty straightforward. But when in nested style, two things need to be considered. The first issue is the ambiguous match of if-else pair (aka dangling else problem). In the recently-invented languages, the ”end if” alike expressions are introduced to prevent the misleading situation. However, SystemC, as a class library to C++, inherits the if-else structure from C, which means it does not have the mechanism to handle the if-else ambiguity. So, the extended Backus–Naur Form method is introduced to identify if-else statement as a whole; this will be further discussed in the compiler design chapter.

The second thing is for the code generation. Let's look at the target if-else structure in VHDL first, which is shown in Listing 2.16.

```
if condition then
        sequential statements
{elsif condition then
        sequential statements}
[else
        sequential statemetns]
end if;
```

Listing 2.16: Corresponding VHDL if-else statements

As seen in Listing 2.16, a new keyword (elsif) is introduced, which aligns the nested single if statement with the above level if statement. Like we have presented in solving the if-else ambiguity, the if-else will be put as a whole in the syntactic analysis. But the target VHDL requires that if the next level if-else statement is just if statement, we should pull it into the one level above and apply the keyword elsif.

2.2.4 Switch statements

As another kind of selection statement, switch statements are fully equivalent to if-else statements in the language interpretation level. But in the hardware implementation, they are quite different, and different hardware logic block will be applied after the synthesis. The general syntax of switch statements in SystemC is shown in Listing 2.2.4:

```
switch(condition)
{
    case constant: {case constant: }
        statement-seq;
        break;
    {case constant: {case constant: }
        statement-seq;
        break;}
    [default : statement-seq]
}
```

The above listing is derived from [6]. Where, "case constant: case constant: " represents a case label list. With this structure, the multiple cases can be matched into the same result. However, VHDL handle the multiple-to-one mapping in another way. Let's first check the VHDL listing for case statements.

```
case expression is
    when choices =>
        sequential statements
    {when choices =>
        sequential statements}
        -- branches are allowed
    [ when others => sequential statements ]
end case;
```

Listing 2.17: Corresponding VHDL case statements

At a glance, it seems there is no corresponding part to the multiple-to-one statements. But the following code snippet may be a good example to show how.

```
case CNTL is
    when 3 | 15 =>
        Z <= A xor B;
    when others =>
        null;
end case;
```

Listing 2.18: Multiple-to-one code snippet in VHDL

It is easy to see that in VHDL the choices can consist of one or more options, which solve the multiple-to-one mapping issue. Furthermore, when there is no action in default statement in SystemC. A new keyword, null, is introduced to cover all the possible choices.

2.2.5 Iteration statements

Iteration statements are a big family, which consists of for loop, while loop, and do-while loop. However, in VHDL, the family is quite the same, which has three types of iteration schemes, namely, basic loop, while while loop and for loop. For convenience, only for loop and while loop are considered. Let's first check the syntax for these two loops.

```
//For-loop statement
for (for-int-statement; condition; variable increment){
     statement
     [if (condition) continue;]
     [if (condition) break;]
}
//While-loop statement
while (condition){
     statement
     [if (condition) continue;]
     [if (condition) break;]
}

//Do-while statement
do {
     statement
     [if (condition) continue;]
     [if (condition) break;]
}
while (expression);
```

Listing 2.19: Iteration statements in SystemC

For simplicity, the continue and break statements are put in an explicit way to match the next and exit statements in VHDL. For VHDL, it should be noted that identifier (index) in the for-loop statement is automatically declared by the loop itself, so there is no need to declare separately. Besides, the value of the identifier can only be read inside the loop and is not available outside.

```
--For loop statement
for identifier in range loop
     sequential statements
     [next [label] [when condition];
     [exit [label] [when condition];
end loop[ loop_label ];

--while loop statement
while condition loop
     sequential statements
```

```
    [next  [label] [when condition];
    [exit  [label] [when condition];
end loop[ loop_label ];

--loop statement, mapping of do-while loop
loop
    sequential statements
    [next  [label] [when condition];
    [exit  [label] [when condition];
    exit when not condition;
end loop;
```

Listing 2.20: Iteration counterparts in VHDL

2.3 Conclusion

Without applying the features, like classes and templates, the restricted subset of SystemC seems like C++ with hardware tag language. But as discussed it does not hurt the capability of accurately describing the hardware design. On the other side, it also greatly alleviates the compiler design. During the next chapter, ANTLR as the compiler generator toolset will be introduced. Besides, the construction process for the compiler will be shown step by step.

CHAPTER 3

COMPILER CONSTRUCTION

Although a compiler with commercial strength is still quite time consuming, compiler construction is not as horrible as it was used to be. At the present, there are many tools, named compiler generators (aka compiler compilers), facilitating the compiler design. To name a few, `lex/yacc` (or their GNU versions, `flex/bison`), ANTLR [10][11], Javacc are very good members of this family. For the open source tools, sc2v [3] and gsc, `lex/yacc` and keystone[1] are used respectively.

3.1 lex/yacc versus ANTLR

ANTLR[2] , with the full name of another tool for language recognizer, is used in the thesis. Since the wide popularity of `lex/yacc`, the comparison between these two languages are summarized as follows:

- ANTLR is written in Java, while `lex/yacc` is written in C. Although there is efficiency loss, a simple platform-independent synthesis tool can be built in the future. Also, Java has many advanced data structures to organize different data types, Javadoc can easily extract the structure comments to generate documents.

- ANTLR offers four kinds of grammar types, lexer, parser, tree, and combined lexer and parser. Unlike the separation of lex and yacc (communicating with each other through symbol table, yylval and yystype), this integral tool can greatly simplify the communication between lexer and parser. Besides, the flat AST can be easily generated after the parsing phase, while in yacc, the data structures and data operation methods should be written by the user to build the tree.

[1]A C++ front-end parse framework based on BTYACC, a variant of yacc, and treecc, a AST generator.

[2]The version 3 is used. According to the author, it has been rewritten over version 2. Several powerful features, such as LL(*) parsing strategy, auto-backtracking mode, and memoize option are introduced to enhance the ANTLR tool.

- ANTLR supports Extended Backus-Naur Form (EBNF) with LL(*) parsing strategy, while yacc just use LALR(1) with only one token lookahead.Due to this convenient feature, the nonterminal and terminal tokens have more convenient notations in the grammar specification. LL(*) parsing strategy also enables the arbitrary token lookahead. However, the associated time penalty is not negligible along with the increment of lookahead tokens. The compromised way could be specify the number lookahead tokens where you wan to apply.

3.2 Building front end

The front end is to convert the source program into some specific intermediate representation (IR). It also manages the symbol table for the back-end. The front end normally consists of two phases, lexical analysis (aka scanner) and syntactic analysis (aka parser).

Before implementing the grammar specifications, Let's get to know the structure of ANTLR file[3] , which is shown in Listing 3.1.

```
/** This is a document comment */
grammarType grammar name;
<<optionSpec>>
<<tokenSpec>>
<<attributeScopes>>
<<actions>>

/** doc comment */
rule1 : ... | ... | ... ;
rule2 : ... | ... | ... ;
...
```

Listing 3.1: The general structure for the grammar file

To better illustrate the implementation, a grammar example corresponding to the grammar structure above is shown in Listing 3.2:

```
/** This is a document comment */
grammarType grammar name;
grammar T;
options {
language=Java;
}
@members {
String s;
```

[3]In lex and yacc, their grammar structures are also quite similar to each other, except for many primitives in yacc, such as %left, %right, %nonassoc, %token, %type.

```
 9  }
10  r : ID '#' {s = $ID.text; System.out.println("found␣"+s);}
        ;
11  ID: 'a'..'z' + ;
12  WS: ('␣' |'\n' |'\r' )+ {skip();} ; // ignore whitespace
```

Listing 3.2: Grammar example in ANTLR

The line 10 in the Listing 3.2 is the grammar rule, while r is nonterminal token, ID is terminal token. The statements in the brace brackets are actions to the rule to the left. The line 11 and 12 are the lexer part for the simple grammar. Line 10 means only little case letters are accepted, while Line 10 means whitespace ("␣"), line feed ("\n"), carriage return ("\r") will be ignored with skip() method. Since this is not the iterative rule, only one ID ending with "#" can be found and output.

To better construct either scanner and parser, the knowledge of regular expression is necessary(shown in Table 3.1). A minor difference between flex and ANTLR should

Table 3.1: Common regular expression symbols

Symbol	Description
?	zero or one of the preceding element
*	zero or more of the preceding element
.	match any single character
[]	a character class that matches any character within the brackets. If the first character is circumflex(∧), it changes the meaning to match any character except the ones within the brackets
()	grouping the possible choices
|	The alternation operator; matches either the preceding regular expression or the following regular expression

be indicated here is that flex uses [a-z] to represent the class of little case literal tokens, while ANTLR uses ['a' .. 'z'].

With the handy symbols, a basic scanner for identifying the semantic components in SystemC can be easily constructed (shown in Listing 3.3):

```
Integer_literal
            :       DIGIT+
            |       '0x' (DIGIT | LETTER)+
            ;

Boolean_literal
```

```
            :         'true'
            |         'false'
            ;

StringLiteral
      :   '"' ( ~('\\'|'"') )* '"'
      ;

fragment
LETTER
            :         'a'..'z' | 'A'..'Z'
            ;

fragment
DIGIT       :         '0'..'9'
            ;

ID          :         LETTER (LETTER | DIGIT | '_')*
            ;

WS    :   (' '|'\r'|'\t'|'\u000C'|'\n') {$channel=HIDDEN;}
      ;

COMMENT
      :    '/*' ( options {greedy=false;} : . )* '*/' {
          $channel=HIDDEN;}
      ;

LINE_COMMENT
      : '//' ~('\n'|'\r')* '\r'? '\n' {$channel=HIDDEN;}
      ;

// ignore #line info for now
LINE_COMMAND
      : '#' ~('\n'|'\r')* '\r'? '\n' {$channel=HIDDEN;}
      ;
```

Listing 3.3: The basic scanner for SystemC

In the above listing, two kinds of comment styles, single line comment and multiple line comment, are support. The attribute "channel", instead of skip() method, is used here. The difference between them is skip() will not only hide the scanned token, but also throw it anyway, while `HIDDEN_CHANNEL` will not throw any token. The feature used here is fragment specifier, which help to build the lexical rule. The fragment rule will not generate any token as well.

In many cases, there is also need to transform or output the input token (like output ID in Listing 3.2). So the knowledge of the attributes of token is necessary. The common attributes of ANTLR used in the application are listed in Table 3.2 [10]: The attributes in Table 3.2 can also be used in the parser, while their function calls

Table 3.2: Predefined token and lexer rule attributes

Attribute	type	Description
text	String	The text matched for the token; translates to a call to getText().
type	int	The token type(nonzero positive integer) of the token such as `INT`; translates to a call to getType().
line	int	The line number on which the token occurs, counting from 1; translates to a call to getLine().
pos	int	The character position within the line at which the token's first character occurs counting from zero; translates to a call to getCharPostionInline().
index	int	The overall index of this token in the token stream, counting from zero; translates to a call to getTokenIndex().
channel	int	The token's channel number. The parser tunes to only one channel, effectively ignoring off-channel tokens. The default channel is 0(`Token.DEFAULT_CHANNEL`), and the default hidden channel is `Token.HIDDEN_CHANNEL`.
tree	Object	When building trees, this attribute points to the tree node created for the token; translates to a local variable reference that points to the node, and therefore, this attribute does not live inside the token object itself.

can be invoked in the code generator file as public methods.

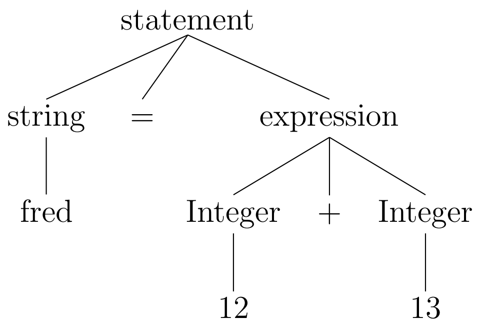
3.2.1 Parser

Parsing is the second stage of the compiler's front end. The parser works on the program as transformed by the scanner; it views the input as a stream of words, where each word has its annotation in the syntactic category. The parser will try to derive a set of rules to fitting these tokens input into the grammatical model of the source language specification. If the input is well fitted, an appropriate IR is generated; otherwise, the error and corresponding diagnostic information will be presented.

The parsing algorithms fall into two general categories, top-down parsing and bottom-up parsing. The examples for these two parsing algorithms are ANTLR and yacc, respectively. However, it is just a view-point difference: top-down construct the syntax tree with the root and grow the tree towards the leaves, while the bottom-up parser works in an contrary way. That's not the key point. The key difference is on how to settle the syntax ambiguity. ANTLR offers many attractive features, like semantic and syntactic predicates and auto-backtrack options. What's more, the most powerful LL(*) is provided to assist the compiler design. In comparison, for yacc, only %left, %right, %nonassoc can be used to solve the language ambiguity, and the number of lookahead token is quite restricted, only one. That's also the reason why ANTLR instead of yacc is chosen for the compiler generation.

A small syntax example
Before discussing the syntactic analysis, an excellent plug-in for designing grammar in Eclipse with ANTLR should be introduced first. It is ANTLRWorks a GUI development environment that sits on top of ANTLR and helps edit[10]. With its assistance, both concrete syntax and abstract syntax representations can be shown[4]. However, if the auto-backtracking option is used, the ANTLRWorks will does not work[5].

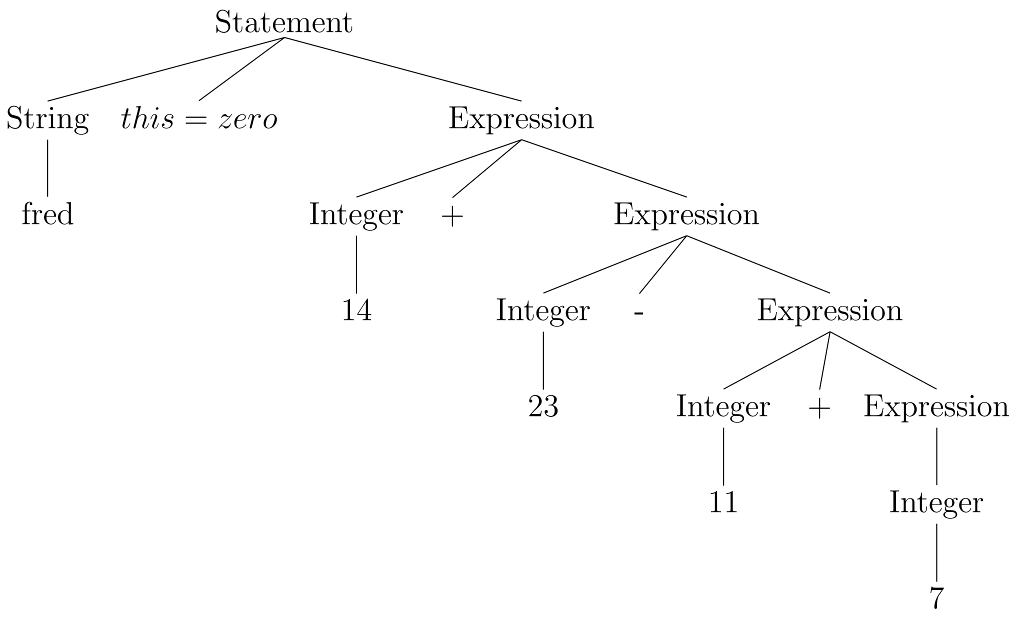
Let's assume parsing the input statement $fred = 12+13$; the possible AST would be looked as follows:



If go further, the LL(*) parsing strategy can be illustrated. $fred = 14+23-11+7$; the possible AST tree can be expressed in the following structure with right recursive rule.

[4]The difference between the two representations is that the abstract syntax extract the tokens, which have semantic meanings, to build the syntax tree, while concrete syntax tree is not.

[5]It is because that the ANTLRWorks did not get updated for newest ANTLR



Following the design logic, the grammar rules can be derived, and it is explicit to mimic the program flow of the source file.

3.3 Constructing back end

The term back end is sometimes confused with code generator because of the overlapped functionality of generating assembly code. Therefore, sometimes literatures use middle end to distinguish the generic and analysis and optimization phases in the back end from the machine-dependent code generators. In other words, the back end should include analysis, optimization and code generation these three phases in the broader sense.

To simplify the compiler design, the generic analysis and optimization are not fully considered. The main phase of the back end would be code generation.

3.3.1 Code generation

Code generation, to its name, is to emit the readily executable code to express the intention of the source code in the way of target language. The typical phases of code generation include instruction selection, instruction scheduling, register allocation and debug data generation if required. However, since the two high level languages are functionally equivalent, only the instruction selection needed to be considered in the translator design. With the corresponding constructs discussed in the Chapter 2, it is easy to finish the instruction selection. The rest is just to reprint what the collected information in the style of VHDL.

Although SystemC is quite different from the VHDL program both in the language arrangement and description style, but the translation can be put in a simple way.

Just tree walking several times and reprint the tree information and information stored in various kinds of JavaBeans. The whole printing process can be summarized as follows:

Print the library → Print the port definitions → Print the process

That is just for the case of one module with one thread. If multiple modules in hierarchical design, the complicated form from reference should be followed. Here, a small example as 1 bit full adder is taken as reference to explain the translation process.

```
#include "systemc.h"

SC_MODULE (BIT_ADDER){
   sc_in<sc_logic> a,b,cin;
   sc_out<sc_logic> sum,cout;
        SC_CTOR (BIT_ADDER){
        SC_METHOD (process);
        sensitive << a << b << cin;
   }

   void process(){
         sc_logic aANDb,aXORb,cinANDaXORb;

         aANDb = a.read() & b.read();
         aXORb = a.read() ^ b.read();
         cinANDaXORb = cin.read() & aXORb;

         sum = aXORb ^ cin.read();
         cout = aANDb | cinANDaXORb;
    }
  };
```

Listing 3.4: 1 bit full adder modeling in SystemC

The above listed code is for the 1 bit full adder design. During the translation, the port related definitions will be stored into the PortBean constructs, while process along with its sensitive list goes to the ProcessBean constructs. The `sc_logic` variables, affiliated with the process will be kept into the VariableBean with labels of function scope level. So basically, the elements for the source tree will be kept into different constructs, and subtle information for them will become its unique labels when translated into the target program. In this sense, these alongside constructs are always with AST for helping the translation. They are also called symbol tables.

Aside from building the constructs for representing the basic elements, various kinds of pseudo nodes are put into the AST to help identify the structures of subtrees. It sounds like street labels. When you just follow the map to search some certain location, the street labels would definitely help you find out. The compiler works in the exact same way. You search the AST branch by branch to find what you are interested, and then translate the corresponding part with the proper words. Besides using pseudo nodes for identifying the subtree, you can also use them to group the some sentences to optimize the source code. When analog to the map-street connection, it can be taken as a lot of similar apartments in a same community. As result, these street locations may share the same community name in the map to facilitate the search.

After acquiring the different information from the source program, the target code can be easily generated following the printing process.

3.4 Conclusion

Compiler construction can be divided into two main parts, front end and back end. Also, the compiler can be constructed one by one with verification of each step. The Flex/Bison is the most common compiler construction toolset, while ANTRL has many good features to facilitate the design. After extracting the different SystemC constructs information, it is easy to reprint the whole program in terms of VHDL. However, as discussed in the previous chapter, many input restrictions have been exerted on the input SystemC models. So if better pattern matching solutions can be came up with, the ability of the compiler to interpret the source program will be greatly enhanced.

CHAPTER 4

TESTING

The compiler described in the previous chapter has been implemented, and in order to provide some measure of validation for the tool, it will be tested through some representative examples. In this chapter, these test models with the simulations for the post-translation models are presented for comparison. In Section 4.1, the test models used to test the tool is introduced first. In Section 4.2, the simulations will be presented. In the final conclusion section, the features and limitations of the design will be stated in short.

4.1 Test Models

To verify the created tool three representative digital designs are tested through. The original SystemC models and the resulting VHDL models are presented and the post-translation simulation are shown. These three digital designs, which represent combinational logic, sequential logic, and FSM design style, are adopted from [1]. The three models were written in synthesizable SystemC model, conforming to the previously described input restrictions. All of these are useful hardware modules and can be expanded into larger and real digital applications. The characteristics of the three models are summarized in Table 4.1.

Table 4.1: Test models for the compiler

Model Name	Type	Size(lines of code)
1 bit full adder	combinational logic	28
Johnson counter	sequential logic	31
Mealy machine	FSM	68

4.1.1 1 bit full adder design

The first model used for testing is the most common design in combinational logic, the 1 bit full adder. The full demo code is listed in Listing 4.1:

```
#include "systemc.h"

SC_MODULE (BIT_ADDER)
{
     sc_in<sc_logic> a,b,cin;
     sc_out<sc_logic> sum,cout;

          SC_CTOR (BIT_ADDER)
          {
          SC_METHOD (process1);
               sensitive << a << b << cin;
          }

          void process1();
};

void BIT_ADDER::process1()
{
      sc_logic aANDb,aXORb,cinANDaXORb;

      aANDb = a.read() & b.read();
      aXORb = a.read() ^ b.read();
      cinANDaXORb = cin.read() & aXORb;

      sum = aXORb ^ cin.read();
      cout = aANDb | cinANDaXORb;
}
```

Listing 4.1: 1 bit full adder modeled in SystemC

The 1 bit full adder is my first test as it is very simple model and pretty much straight forward. It makes no use of the more intricate features of the tool and has a direct mapping strategy to VHDL.

4.1.2 Johnson counter

A common digital design always consists of combinational parts and sequential parts. So a Johnson counter, as a typical example, to test the software.

A Johnson counter is a digital circuit which consists of a series of flip flops connected together in a feedback manner. The circuit is special type of shift register where the complement output of the last flip flop is fed back to the input of first flip flop. This is almost similar to ring counter with a few extra advantages. When the circuit is reset all the flip flop outputs are made zero. For n-flip flop Johnson counter we have a MOD-2n counter. That means the counter has 2n different states.

The circuit diagram for a 3 bit Johnson counter is shown in Figure 4.1:

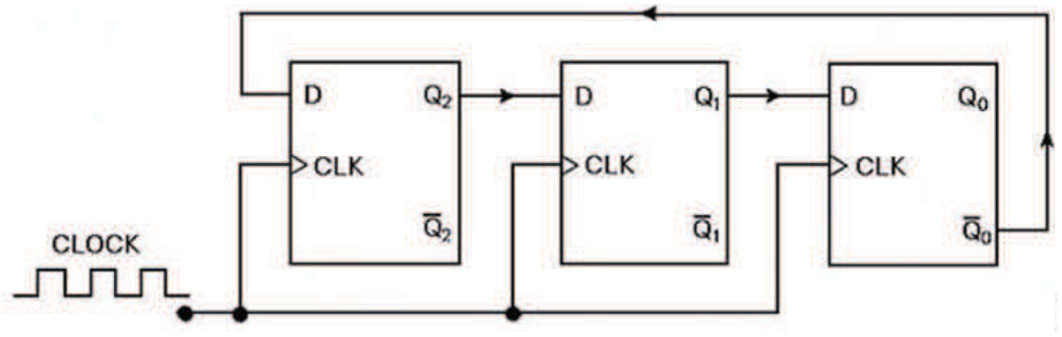


Figure 4.1: Johnson counter circuit diagram

The SystemC source program is listed in Listing 4.2

```
#include "systemc.h"

const int NBITS =3;

SC_MODULE (johnson_ctr) {
  sc_in<bool> clk , clear;

  sc_out<sc_uint<NBITS> > q;

  void prc_counter();

  SC_CTOR (johnson_ctr) {
    SC_METHOD (prc_counter); // Synchronous.
    sensitive << clk << clear;
  }
};

void johnson_ctr::prc_counter() {
  sc_signal<sc_uint<NBITS> > temp;
    //sc_uint<NBITS> temp;
  if (clk.posedge())
  {
          q = temp;
          if (!clear)
                temp = 0;
          else {
                //temp = (temp.range(NBITS-2, 0), !temp[
                    NBITS-1]);
                q = .concat(temp.range(NBITS-2, 0), !temp[
                    NBITS-1]);
```

```
            }
    }
}
```

Listing 4.2: Johnson counter modeled in SystemC

4.1.3 Mealy Machine

Finite state machine (FSM) is indispensable when developing large complicated and robust applications; it consists of two design types, Mealy machine and Moore machine. The difference between them is minor; a Mealy machine has outputs that depend on the state and input, while a Moore machine has outputs that depend on state only.

The SystemC model for Mealy machine is listed in Listing 4.3.

```
#include "systemc.h"

SC_MODULE (mealy) {
  sc_in<bool> clk, reset, a;

  sc_out<bool> z;

  //// One-cold encoding:
  //enum state_type {S0=0xE, S1=0xD, S2=0xB, S3=0x7};
  enum state_type {S0, S1, S2, S3};

  sc_signal<state_type> mealy_state, next_state;

  void prc_state();
  void prc_output();

  SC_CTOR (mealy) {
    SC_METHOD (prc_state);
    sensitive << clk << reset;

    SC_METHOD (prc_output);
    sensitive << mealy_state << a;
  }
};

void mealy::prc_state() {
  if (reset)
    mealy_state = S0;
  else
        if (clk.posegde())
        mealy_state = next_state;
}

void mealy::prc_output() {
  switch (mealy_state) {
    case S0:
      if (a) {
        z = 1;
        next_state = S3;
      }
      else
        z = 0;
      break;
    case S1:
      if (a) {
        z = 1;
        next_state = S0;
      }
      else
        z = 0;
      break;
    case S2:
      if (!a)
        z = 0;
      else {
        z = 1;
        next_state = S1;
      }
      break;
    case S3:
      z = 0;
      if (! a)
        next_state = S2;
      else
        next_state = S1;
      break;
  }
}
```

Listing 4.3: Mealy machine modeled in SystemC

4.2 Simulations

All the test models were translated using the tool and the resulting VHDL models were simulated. The test results are presented with both functional simulations and

RTL views.

The RTL view after elaboration and analysis under Quartus II is also shown in the Figure 4.3, which is essentially the logic gates mapping from SystemC code. The functional simulation also approves this simple translation.

```
--Code generated from SystemC file
--Author:  Rui Chen
--Tools: ANTLR and Eclipse

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity BIT_ADDER is
                port(
                a: in std_logic;
                b: in std_logic;
                cin: in std_logic;
                sum: out std_logic;
                cout: out std_logic
                );
end BIT_ADDER;

--architecture part
architecture arch of BIT_ADDER is

begin
process1:
        process( a, b, cin )
        variable aANDb: std_logic ;
        variable aXORb: std_logic ;
        variable cinANDaXORb: std_logic ;
        begin
        aANDb := (a and b);
        aXORb := (a xor b);
        cinANDaXORb := (cin and aXORb);
        sum <= (aXORb xor cin);
        cout <= (aANDb or cinANDaXORb);
        end process process1;
end arch;
```

Listing 4.4: 1 bit full adder generated in VHDL

With the synthesis result shown in Figure 4.4, it is obviously to know that a Johnson counter consists of a multiplexer and a D flip flop. The simulation also approves the result for the design.

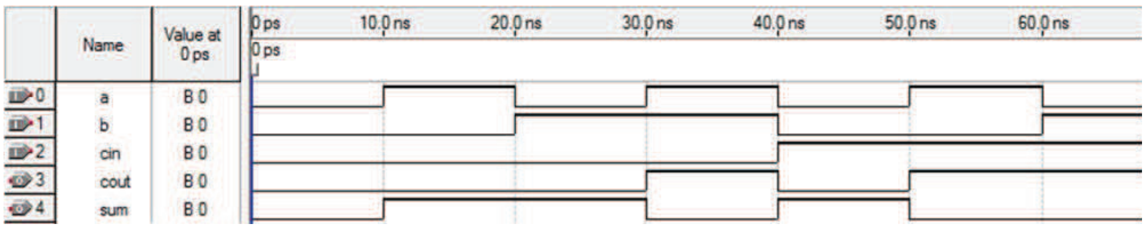


Figure 4.2: Functional simulation for 1 bit full adder

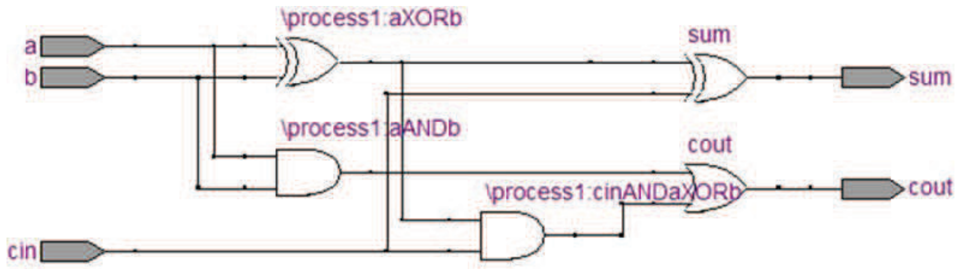


Figure 4.3: RTL view of 1 bit full adder

```
--Code generated from SystemC file
--Author:  Rui Chen
--Tools: ANTLR and Eclipse

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity johnson_ctr is
                port(
                clk: in std_logic;
                clear: in std_logic;
                q: out std_logic_vector ( 2 downto 0 )
                );
end johnson_ctr;

--architecture part
architecture arch of johnson_ctr is
signal t_cnt: std_logic_vector( 2 downto 0 );
constant NBITS: integer := 3;
begin prc_counter:
        process( clk, clear )
        begin
        if (rising_edge(clk)) then
```

```
                if (falling_edge(clear)) then
                        if ( not clear = '1' ) then
                                q <= (others => '0');
                        else
                                q <= (t_cnt((NBITS - 2)
                                   downto 0) & ( not t_cnt
                                   ((NBITS - 1)) ));
                        end if;
                end if;-- for negative list
        end if;-- for positive list
        end process prc_counter;
end arch;
```

Listing 4.5: Johnson counter generated in VHDL

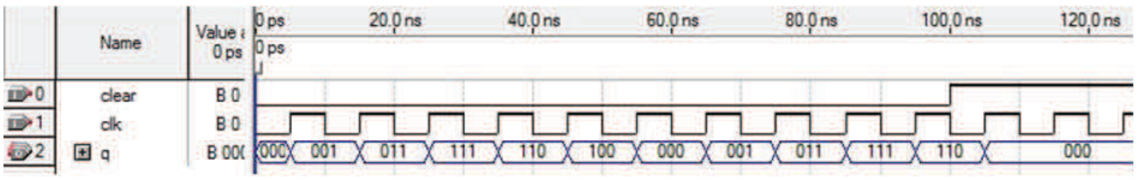


Figure 4.4: Functional Simulation for Johnson counter

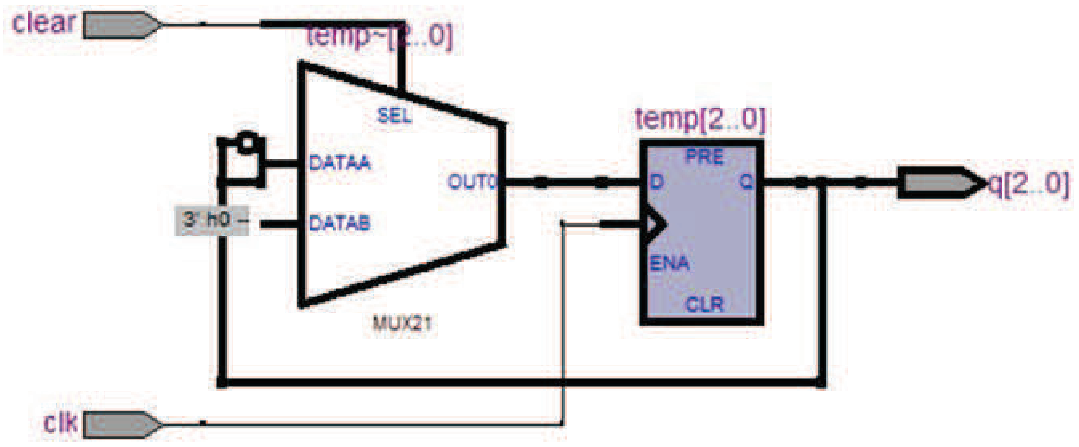


Figure 4.5: Synthesis result for Johnson counter

For some particular digital modeling, a FSM is always needed. The mealy machine is developed in the most typical way. And the simulation and RTL view also approves the software correctness.

```
--Code generated from SystemC file
```

```
--Author:  Rui Chen
--Tools: ANTLR and Eclipse

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity mealy is
                port(
                clk: in std_logic;
                reset: in std_logic;
                a: in std_logic;
                z: out std_logic
                );
end mealy;

--architecture part
architecture arch of mealy is
type state_type is (S0, S1, S2, S3);
signal mealy_state: state_type;
signal next_state: state_type;
begin
prc_state:
        process( clk, reset )
        begin
        if (rising_edge(reset)) then
                if (falling_edge(clk)) then
                        if (reset= '1') then
                                mealy_state <= S0;
                        else
                                mealy_state <= next_state;
                        end if;
                end if;-- for negative list
        end if;-- for positive list
        end process prc_state;
prc_output:
        process( mealy_state, a )
        begin
        case mealy_state is
                when S0 =>
                        if (a= '1') then
                                z <= '1';
                                next_state <= S3;
                        else
                                z <= '0';
```

```
            end if;

        when S1 =>
            if (a= '1') then
                z <= '1';
                next_state <= S0;
            else
                z <= '0';
            end if;

        when S2 =>
            if ( not a = '1' ) then
                z <= '0';
            else
                z <= '1';
                next_state <= S1;
            end if;

        when S3 =>
            z <= '0';
            if ( not a = '1' ) then
                next_state <= S2;
            else
                next_state <= S1;
            end if;

    end case;

    end process prc_output;
end arch;
```

Listing 4.6: Mealy machine generated in VHDL

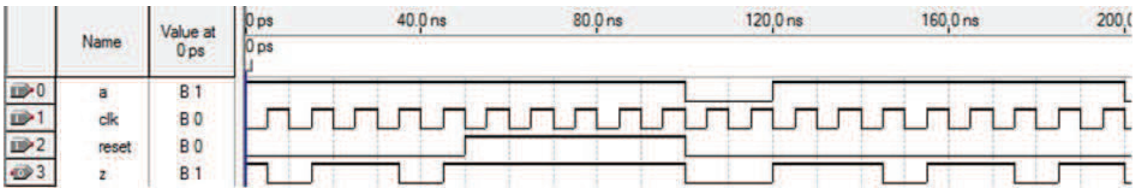


Figure 4.6: Functional Simulation for Mealy machine

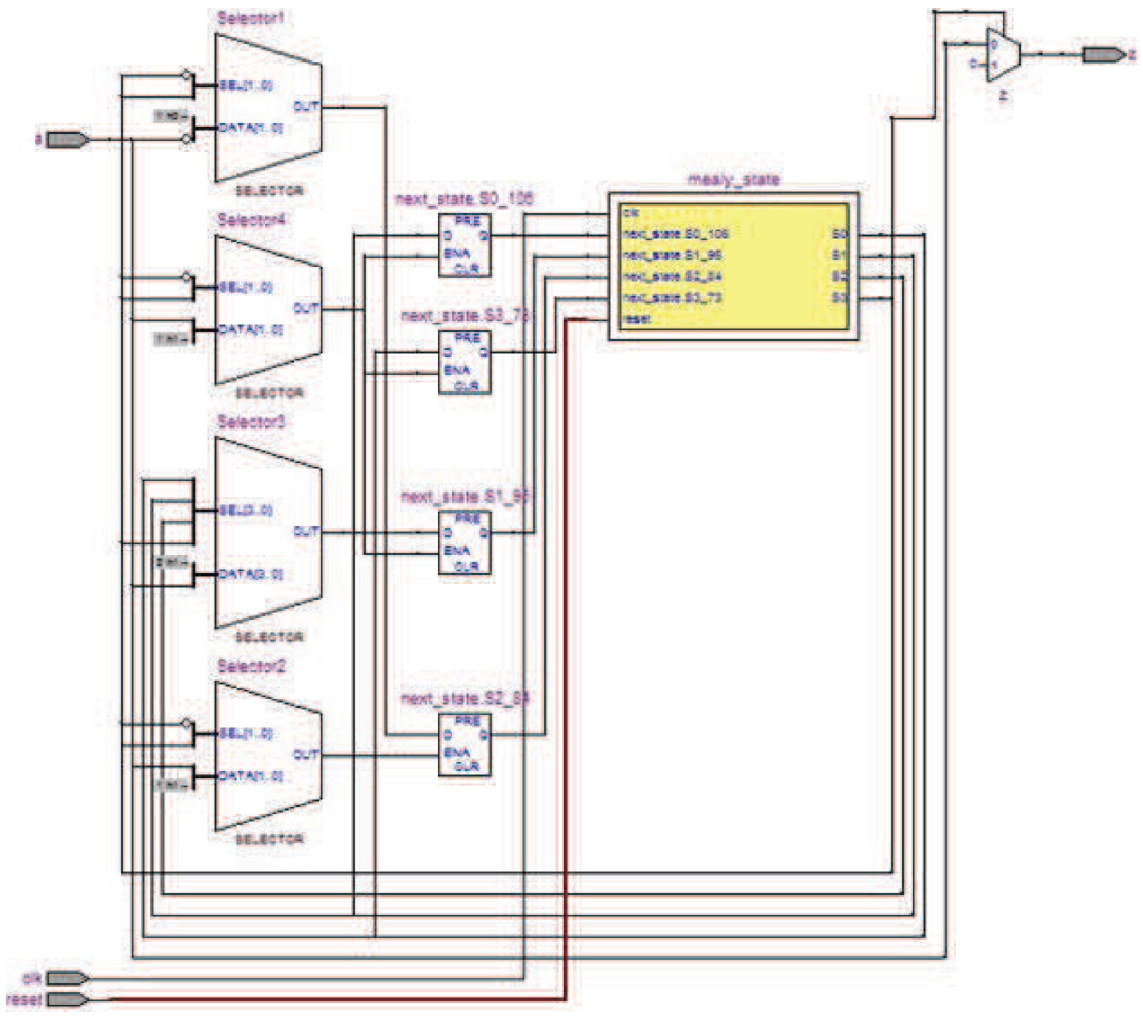


Figure 4.7: RTL view for the Mealy machine

4.3 Conclusion

Although the translations can be approved in many ways as you expect, it is still fairly fundamental of competitively less commercial strength to handle the various source code input styles. As discussed in the design part, the robust compiler should have the ability to handle different code writing flavors. It is always easier said than done, but it is future expectation of this compiler design.

CHAPTER 5

CONCLUSION

Compiler design of translating SystemC to VHDL allows the hardware synthesis with the separate need of developing EDA tool for synthesizing SystemC models. It adopts the indirect methodology to leverage existing technology for synthesis while using a modeling paradigm tailored for hardware/software co-design. The error-prone manual translation from a SystemC model to a synthesizable VHDL model can be reduced to a manual reduction of the SystemC model to conform to the tool restrictions.

In this chapter, the summary of the thesis will be presented first in Section 5.1, Then, the overview of the main contributions of the thesis and future prospects for the work will be discussed in Section 5.2 and 5.3.

5.1 Summary of the thesis

Compiling SystemC into VHDL can be accomplished by putting restrictions on the SystemC models. While the removals of many of the dynamic features of SystemC and the more abstract elements of C++ language reduces the effective expressions of these languages, however, they do not hurt the accuracy of describing the architecture of the hardware models. In this sense, the generated VHDL models are functionally equivalent to the input SystemC digital designs, and it also get proved via testing cases.

From module down to each data type, down to the individual variable operation, the different types of constructs for the two languages were compared in details. Like illustrated in Chapter 3, this kind of black art technique is basically reorganize the source from the point of target language. Thus, the different JavaBeans are designed to cope with the different constructs, such as ports, variables, signals, and etc. Also, JavaBean is also used for type checking and helping maintaining the consistency of the program. Although some features like sub-function has not been supported yet, they can be included into the future version of compiler.

ANTLR is a toolset for building the compiler, it does offers a lot of application programming interfaces (APIs) that you can access the scanned constructs. With identifying the different constructs, the several passes of tree walking has been preceded. As result, different parts of VHDL program are generated by each pass. Besides, some information, like line number, will also help to locate the translation errors.

To verify the compiler tool, a series of typical digital designs has been tested. The test models represent the real world application to some degree. The pre- and post-simulation results after the translation are identical. This is what was expected from the tool. Finally, it should be noted that the after- "place-and-route" simulations were not performed, because their results mostly differ from behavioral simulation results due to synthesis inference rules and targeted technology. Therefore, manual effort is needed to match them and this was not the focus of our work or of this thesis.

5.2 Main contributions

The main contributions of the thesis can be summarized as follows.

- The similarities of SystemC and VHDL are explored from the data types to the control flow structures. With each comparison, a feasible mapping developed and realized in the latter compiler construction. In this sense, the compiler can fulfill the fairly basic identification of the SystemC constructs and generate the corresponding code in the VHDL style.
- Certain restrictions should be exerted on the source code, such as removals of templates and classes. But as tested with the various typical kinds of digital designs, the effective expressiveness of the restricted SystemC does not get any loss.

5.3 Future directions

As known to all, no ends for the perfect. However, from the points of the input restrictions and code generation optimizations, future work on this topic can be conducted in the fields of restriction relaxations and tool improvements.

- Relaxation of the restrictions on the SystemC models
  - Floating and fixed point features. Since no standard implementation of a synthesizable fixed and floating point data type exists for VHDL, these two features are disabled in the tool. So with good idea on treating these two data types, the tool will be good at handling the sophisticated data types.
  - Classes and templates. The most valuable part of C++ is definitely class and template. Unfortunately, there is no feasible solution for it. But the goodness is, removals of them would not hurt the hardware description at all. But in the future, with adding these two translation features, the great hardware describing effectiveness can be expected.

- Improvements of the tool
  - Parallelism. Contrast to CPUs, FPGAs and ASICs are born to handle the parallel algorithms. So the constant propagation and constant folding mechanism should be applied to change the data path to make use of hardware resources.
  - Multi-target. The compiler tool only supports the output in VHDL. This was, after all, the primary target. Output back into SystemC could prove a valuable feature for validation of the model. It would allow the mapped translation to be tested in its original testing environment and any needed changes to the model could be implemented directly in this environment.

BIBLIOGRAPHY

[1] A.V. Aho. *Compilers: principles, techniques, & tools.* Addison-wesley, 2007.

[2] A.W. Appel and M. Ginsburg. *Modern compiler implementation in C*. Cambridge Univ Pr, 2004.

[3] J. Bhasker. *A SystemC Primer.* Star Galaxy Pub., 2004.

[4] D.C. Black, J. Donovan, B. Bunton, and A. Keist. *SystemC: From the ground up*. Springer Verlag, 2009.

[5] J. Castillo, P. Huerta, and J.I. Martínez. An open-source tool for systemc to verilog automatic translation. *Latin American applied research*, 37:53–58, 2007.

[6] K.D. Cooper and L. Torczon. *Engineering a compiler.* Elsevier, 2004.

[7] P. Coussy, A. Takach, M. McNamara, and M. Meredith. An introduction to the systemc synthesis subset standard. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 183–184. ACM, 2010.

[8] D.D. Gajski, N.D. Dutt, C.H.W. Allen, and Y.L. L. Steve. *High-level synthesis: introduction to chip and system design.* Kluwer Academic, 1992.

[9] E. Grimpe and F. Oppenheimer. Extending the systemc synthesis subset by object-oriented features. In *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 25–30. ACM, 2003.

[10] Open SystemC Initiative. *SystemC Synthesiable Subset Draft 1.3.* 2009.

[11] K. Marquet and M. Moy. Pinavm: a systemc front-end based on an executable intermediate representation. In *Proceedings of the tenth ACM international conference on Embedded software*, pages 79–88. ACM, 2010.

[12] K. Marquet, M. Moy, and B. Karkare. A theoretical and experimental review of systemc front-ends. *Verimag Research Report, Tech. Rep. TR-2010-4*, 2010.

[13] M.C. McFarland, A.C. Parker, and R. Camposano. The high-level synthesis of digital systems. *Proceedings of the IEEE*, 78(2):301–318, 1990.

[14] M. Moy, F. Maraninchi, and L. Maillet-Contoz. Pinapa: An extraction tool for systemc descriptions of systems-on-a-chip. In *Proceedings of the 5th ACM international conference on Embedded software*, pages 317–324. ACM, 2005.

[15] S.S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann, 1997.

[16] Douglas E. Ott and Thomas J. Wilderotter. *A Designer's Guide to VHDL Synthesis.* Springer Publishing Company, Incorporated, 1st edition, 2010.

[17] T. Parr. *The definitive ANTLR reference: building domain-specific languages.* Pragmatic Bookshelf, 2007.

[18] T. Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages.* Pragmatic Bookshelf, 2009.

[19] IEEE Computer Society. *IEEE 1164-1993: IEEE Standard VHDL Synthesis Packages.* 1993.

[20] IEEE Computer Society. *IEEE 1076.3-1997: IEEE Standard VHDL Synthesis Packages.* 1997.

[21] IEEE Computer Society. *IEEE 1076.6-1999: IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis.* 2000.

[22] IEEE Computer Society. *IEEE 1076-2002: IEEE Standard VHDL Language Reference Manual.* 2002.

[23] IEEE Computer Society. *IEEE 1076.6-2004: IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis.* 2004.

[24] IEEE Computer Society. *IEEE 1666-2005: IEEE Standard SystemC Language Reference Manual.* 2006.

BIOGRAPHICAL SKETCH

Rui Chen is from Jingzhou, Hubei in China. He graduated from the talented student program at School of Information Science and Engineering at Central South University with Bachelor Degree in Engineering in June 2009. In 2009 Fall, he came to Florida State University to pursue PhD degree. In the fall of 2011, he decided to graduate with Master of Science in Electrical Engineering and try to apply his knowledge to achieve more. He has a lot hobbies, like basketball, soccer, stamp-collecting. In the leisure time, he also loves reading history books.