Verilog-to-C-Compiler: Simulator Generator

A Thesis Submitted
in Partial Fulfilment of the Requirements
for the Degree of
Master of Technology

by

Anand Vivek Srivastava

to the

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

January, 2007

CERTIFICATE

Certified that the work contained in the thesis entitled "*Verilog-to-C-Compiler: Simulator Generator*", by "*Anand Vivek Srivastava*", has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

(Prof. Rajat Moona)
Professor,
Department of Computer Science and Engineering,
Indian Institute of Technology,
Kanpur.

January, 2007

Contents

List of Tables

Abstract

With the increase in complexity of hardware models, we need better simulation tools to keep up with the needs of the designers. While Verilog, like other hardware specification languages, is especially suitable for hardware description, it does not compare favorably to general purpose languages, like C or C++, when efficiency of execution is considered. Moreover, it is very difficult to write hardware specification in a general purpose language. In addition, Verilog or other HDL descriptions are suitable for hardware synthesis.

A Verilog to C compiler allows us to use Verilog for hardware description, and still be able to exploit the efficiency of a general purpose language. This also allows us to reuse the large repository of hardware descriptions that are available as HDL models. This work consists of creating a compiler for Verilog RTL which generates an executable model. The executable model simulates the behavior of the hardware component described by the Verilog RTL model. An event based simulator is generated, which can be executed natively.

Keywords: Verilog, compiler, simulator, hardware model, system modeling

Chapter 1

Introduction

1.1 Objective

This work tries to converts a synthesizable behavioral Verilog model to an equivalent C program, which can be compiled using a standard C compiler and run natively. The primary aim of this exercise is to speed up the relatively slow process of simulation. The objective of this thesis is to parse the Verilog code and create an executable model that simulates the behavior of the hardware component described by the Verilog RTL model.

This work is part of a larger project[2, 3], that intends to create an Integrated Development Environment for system modeling. The project uses Sim-nML[4] for processor model description, that is used by various retargetable tools like code profiler[5], cache simulator[5], assembler[6], dis-assembler[7] and functional simulator[8]. The Sim-nML processor model is converted into C code that can be executed to simulate the processor specified. Since, RTL description of hardware components is typically available as an HDL model, the C code for processor model simulation cannot use them directly. This makes the process of system simulation very slow as during the runtime logic simulation tools are used to simulate the hardware. If hardware behavioral models can be converted into C code programmatically, they can be integrated with the processor models easily.

1.2 Motivation

With hardware getting increasingly complicated, using software tools for designing and testing is necessary. With the increase in complexity of hardware models, we need better simulation tools to keep up with the needs of the designers. A faster simulation tool increases the efficiency of the designers, by reducing the time taken by a design-test-debug cycle. The more efficient the simulator, the more complex hardware model it can execute in feasible time intervals, resulting in shorter product design time. It can reduce costs incurred in acquiring more processing power. Therefore, there is always an incentive to create a faster simulator.

While Verilog, like other hardware specification languages, is especially suitable for hardware description, it does not compare favorably to general purpose languages, like C or C++, when efficiency of execution is considered. Moreover, it is very difficult to write hardware specification in a general purpose language. With the kind of complexity the hardware designs have today, it would be quite inefficient to do so manually. In addition, Verilog or other HDL descriptions are suitable for hardware synthesis.

A Verilog to C compiler, gets us the best of both worlds. This allows us to use Verilog for hardware description, and still be able to exploit the efficiency of a general purpose language. This also allows us to reuse the large repository of hardware descriptions that are available as Verilog RTL models.

1.3 Survey of Related Works

There have been innumerable attempts at writing compilers/converters that convert Verilog models into another models. Lots of compilers convert Verilog to other hardware specification languages like VHDL or SystemC[18], and vice versa. Such conversions are usually done for compatibility reasons or for speed gains. Other incentives for converting Verilog models into other forms include using the target as intermediate representations that can be used as an input for another program. Some such works are discussed below.

1.3.1 Verilator

Verilator[9] is a compiler for synthesizable Verilog into C++ with a SystemC[18] wrapper. The project was first conceived in 1994 by Paul Wasson at the Core Logic Group at Digital Equipment Corporation. The initial design was used to merge Verilog code to a C based CPU model of the Alpha processor and simulated in a C based environment. In 1995, the software started being used by DEC for Multimedia and Network processor development, with Duane Galbi taking over active development. In 2002, Wilson Snyder, did a complete rewrite of Verilator in C++, with a SystemC mode. The software is freely downloadable under the GNU public license. It runs 4-10 times faster than some commercial simulators. Verilator aims at creating executable models of CPUs for embedded software design teams.

Verilator supports a very restrictive synthesizable subset of Verilog. A C++ /SystemC wrapper has to be manually written to execute the generated code. It is a two state simulator, so Xs and Zs are approximated to either 0 or 1. Verilog functions and Verilog tasks are only partially supported by Verilator. The support for Verilog arithmetic operations is partial, and division and multiplication are 32-bit operations. Delays and timing controls are ignored as it would be in any synthesis tool.

1.3.2 vl2mv

Vl2mv[11] was created as a Master's thesis at University of California, Berkeley in 1994. This tool converts a Verilog model to an equivalent automata. The target language is formally defined as BLIF-MV[12], which is a language designed for describing hierarchical sequential systems with non-determinism. This project is now part of a larger project VIS[13] that attempts to implement formal verification, synthesis, and simulation of finite state systems. The project is now a joint project of University of California at Berkeley, the University of Colorado at Boulder, and the University of Texas, Austin. By converting Verilog models into BLIF-MV, it allows the models to be used by VIS. The tool supports most of Verilog 1995 constructs. The compiler is written using C, along with lex and yacc tools. Both VIS and vl2ml are freely downloadable from the project website[13].

1.3.3 Carbonized RTL Machine Objects

Carbon Design System[14] has commercial products that convert Verilog or VHDL RTL models into cycle accurate and register accurate forms. The target representation is called Carbonized Machine Objects, which can be used by other products like Carbon's VSP^{TM} (Virtual System Prototyping)[15] and RealView^{TM}[16]. RealView allows the RTL model to be seen as a cycle accurate SystemC models. The Carbonized Machine Objects can also be called independently by a C++/SystemC wrapper, which works as a test bench. Since the RTL models are converted into cycle accurate models, the performance of these models is better than equivalent inter-cycle accurate event-based simulators.

1.4 Overview

In this work, a compiler has been created that converts Verilog RTL model to a C program. The C program is compiled to get a executable model for the hardware described by the Verilog program. The generated simulator can be run natively independent of the compiler tool. This approach allows a much faster simulation compared to conventional Verilog logic simulators.

This thesis aims at being able to provide fast software simulation for hardware models. Therefore, only a synthesizable subset of Verilog is supported. Moreover, *Verilog-to-C-compiler* creates a behavioral RTL simulator and does support gate level modeling. Independent work that converts Sim-nML processor models into executable C models has been carried out. The C hardware models generated using this tool can be easily integrated with the C processor models, allowing the large repository of Verilog hardware models to be reused with the processor models.

The generated model is an event based simulator, with an execution model similar to the hardware execution model. The only significant deviation from the hardware execution model is that the delays and delay controls. The delays are are non-synthesizable, hence are not supported.

1.5 Organization of the Report

The rest of the thesis is organized as follows. **Chapter 2** gives an overview of the high level design of *Verilog-to-C-compiler*. The design decisions taken for the back-end of the compiler have also been explained in this chapter. **Chapter 3** provides an insight into the implementation details of the software. This chapter also explains the general organization of the various components of the software. Finally, in **Chapter 4** we conclude with results and future work.

Appendix A outlines the Verilog constructs that are supported, along with those that are partially supported or not supported. **Appendix B** describes in some detail the parse tree data structures. **Appendix C** outlines the general purpose data structures that have been implemented.

Chapter 2

Verilog-to-C-compiler Design

Verilog-to-C-compiler compiles Verilog behavioral model and generates a simulator in general purpose programming language C. The design of this tool is similar to most modern day compilers. The high level software design of this tool is represented by the pipe and filter design pattern[19, 21, 22]. Figure 2.1 represents the high level design of the tool.

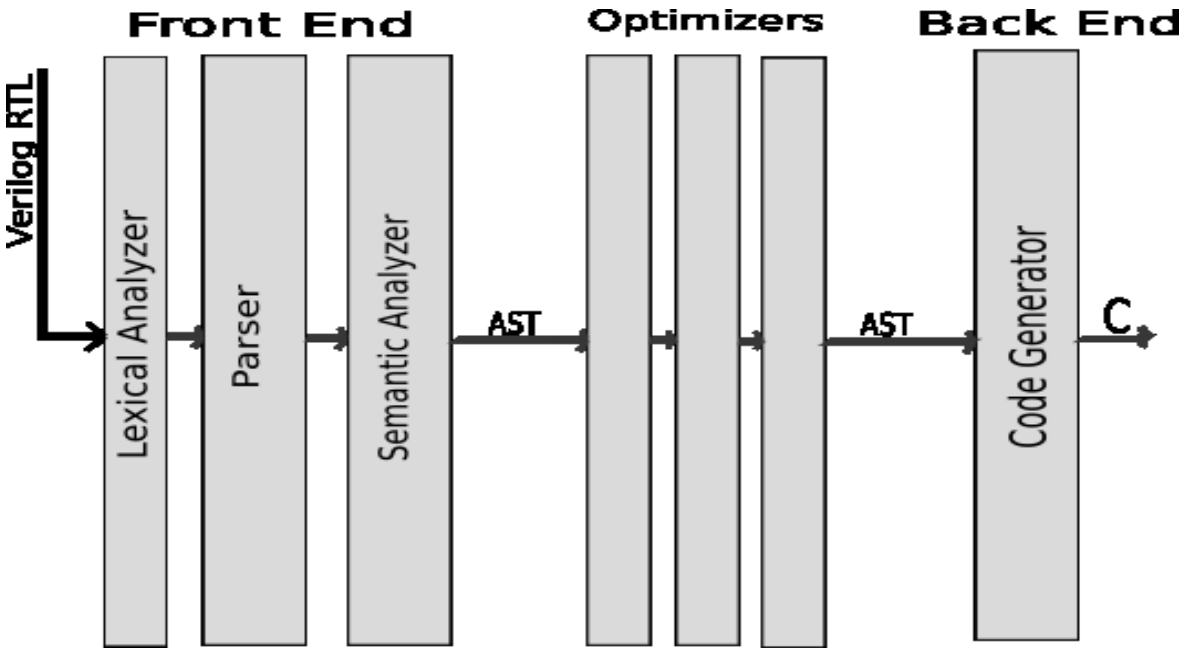

Figure 2.1: Compiler Module Diagram: Pipe and Filter Architecture

2.1 Front end

2.1.1 Lexical Analyzer

The lexical analyzer, the parser and the semantic analyzer form the front end of the compiler. The lexer accepts a Verilog file and breaks it down to a stream of tokens. Ill-formed tokens are captured here and an error is reported. The lexical analyzer associates a context with each token passed to the parser. The context consists of the starting line number, end line number and source filename. This context is used for error reporting and debugging at all steps of compilation. The lexical analyzer handles the definition and usage of Verilog macros, and therefore, works as a preprocessor as well.

2.1.2 Syntactic Parser

The parser accepts the stream of tokens generated by the lexical analyzer and matches them with the grammar rules. Syntax errors, along with their context, are reported at this stage. Each supported Verilog construct is internally modeled as a C++ class. The Verilog constructions in the program are converted into objects of corresponding classes from the parse tree data structure. The objects together form an internal tree structure, referred to in here as the parse tree.

2.1.3 Semantic Analyzer

The semantic analyzer makes changes to certain parts of the parse tree and creates an abstract syntax tree (AST). While creating the AST, the semantic analyzer performs another round of error checking for catching errors that could not be caught by the parser. The semantic analyzer also performs certain other tasks that could not be performed by the parser. These include identifier name and scope resolution. The AST obtained is in general simpler that the parse tree, since it need not capture the syntax of all source language constructs. The AST is used as an intermediate representation between the front end and the back end of the compiler.

2.2 Optimizers

The optimizers performs multiple traversals of the AST, making changes to the AST, but while maintaining all its properties.

A number of optimizations are implemented in the *Verilog-to-C-compiler* which increase the efficiency and readability of the generated code. The optimizations module of the tool is based on the visitor design pattern[19, 20]. The motivation of using the visitor design pattern is to separate the traversal code from the optimizations code, thereby facilitating easy extendability.

Since the structure of the internal representation remains unchanged, optimization routines can be called any number of times and in any order. A certain order of application of optimizations may be better than the other, depending upon the input code. The optimizations that have been incorporated in the tool are as follows.

2.2.1 Parameter replacement

In this optimization, module parameters are replaced by the corresponding value. This can be done only when it is ensured that the value is not overridden (at module instantiation). Parameters declared with the `localparam` keyword can always be replaced, since they can never be overridden. Parameter replacement is done as early as possible so that other optimizers work more effectively. Parameters declared with the `‘define` macro, however, get expanded by the lexical analyzer itself.

2.2.2 Constant Folding

Expressions that can be evaluated at compile time, are evaluated and replaced by their value. This reduces the size of the AST and at times will allow other optimizations to work better. Expressions that are folded would be of one of the following kinds.

- Expressions where all subexpressions are constants while some are not. Therefore, a unary operation with a constant operand and a binary operation where both operands are constants qualify for this group. The arithmetic, logical or bitwise operation will be performed by the compiler and the result would replace the expression. This process is performed in a bottom-up fashion allowing complex

but constant expressions to be evaluated in a single iteration. The following is an example of this kind of optimization.

```
value1 = 4'b1001 & 4'b11;      ⟹      value1 = 4'b1;
```

- Expressions where some subexpressions are constants. Expressions of this kind can be evaluated only in certain cases, where the value depends on the constant parts alone. The following are some examples of such expressions.

```
value1 = value2 || 1'b1;      ⟹      value1 = 1'b1;
value1 = value2 && 1'b0;      ⟹      value1 = 1'b0;
```

2.2.3 Dead code removal

If it can be guaranteed at compile time that a portion of code would not be executed, the code can be dropped. A common example of dead code is unused code like uninitiated modules and functions or tasks that are not invoked. Similarly, if-else statements with constant condition expressions can be simplified. If the expressions evaluates to true, the entire statement can be replaced with the true statement block. If it evaluates to false, it would be replaced by the false statement block. If there is no false statement block and the conditional expression evaluates to false, the entire if statement is dropped.

The following is an example of parameter replacement optimization and dead code removal.

```
parameter WIDTH = 14;
if(WIDTH<32)                    ⟹      display("GOOD");
        $display("GOOD");
else
        $display("BAD");
```

Dead code removal reduces the target code size and the final binary size. This optimization has a minor performance advantage on the execution speeds of generated code by avoiding cache being filled with non-useful code. By simplifying

the if statements some jump statements are also avoided. Since the dropped code is not processed by the code generator module, this optimization increases the efficiency of the compiler as well.

2.3 Back end: Code generator

Verilog-to-C-compiler uses C constructs to model the behavioral constructs of source Verilog code. The constructs that Verilog shares with C, are mapped trivially to the equivalent structures. The Verilog if-else statement is an example, which gets mapped to the C if-else statement. The Verilog constructs that have no direct equivalent in C, have to be simulated by using a combination of C constructs.

It is noteworthy, that the data model of Verilog is significantly different from C. While each bit of a C integer value can have only two states, represented by a 0 and a 1, Verilog HDL value set consists of four states, represented by 0, 1, x and z. The four states represent the possible states a hardware wire or a hardware register can take. A value of 0 means low voltage and value of 1 represents high voltage, which are complimentary to each other. The X and Z values represent an unknown logic state and a high impedance value, respectively. A wire without a driver is denoted by a value Z. When, a value of Z is used in expressions, it may cause an X value on the output port. Also, while the range of all C data types are fixed and are constants, Verilog data types can be much wider. Therefore, a Verilog register data type might not fit within a C integer. To overcome the differences in the data model of the two languages, this tool includes an Integer library (see appendix C.7). This library can handle integers of arbitrary width, supports the four possible state values and all Verilog operators (see appendix A.2 for complete list of supported operators).

Moreover, a C variable would not change its value until it is explicitly assigned another value. This does not hold true for Verilog net data types. These are equivalent to a hardware wire, the output value of which changes continuously as the input value changes (with or without a delay). Therefore, the value on a Verilog wire has to be evaluated continuously.

Similarly, while a Verilog `always` block is intuitive when describing hardware, it lacks a close equivalent in C. For an exact equivalent the `always` block should be

executed in an infinite loop. However, if more then one always blocks exist, then it is not possible for a single threaded process on a uniprocessor machine.

The generated model is event based, and so does not differ much from the hardware execution model. The only significant deviation from in the hardware execution model is that the delays and delay controls are not supported and hence are ignored. *Verilog-to-C-compiler* in the end supplies ANSI C code. The generated code is compiled to get native machine code, that can be run independently.
The generated code has the following sections.

Includes : Various include directives that include some standard C header file and some libraries provided by *Verilog-to-C-compiler*.

Top module : Generated code corresponding to the top module from Verilog source (along with all instantiated modules)

Initialization : A function responsible for initializing various global constructs

Driver : The main function that acts as the driver and runs the target code as would be done by a Verilog simulator.

The code generator takes in the AST generated by the semantic analyzer and output the target code (C program). The C program that is generated can be compiled using a standard ANSI C compiler, and after linking it to the Integer library (described in section C.7) can be run natively on the machine.

2.3.1 Modules and Module Instantiations

Verilog modules do not get any representation in the generated code. All declarations at module scope are mapped to the global scope in C code. Comments are included in the generated code to mark the beginning and end of a module. Expect for the top module, all modules have their corresponding code generated as many times as they are instantiated. Uninstantiated modules, except for the top module, are dropped. For all instantiated modules, the full hierarchical name of the instance is prefixed to the name of its members. If no instance name is specified during module instantiation, an

auto—generated dummy name is provided to each instance. The following example shows how Verilog modules would look like in the generated C program.

Verilog code	C code
module sub (pvalue1);	
...	
endmodule	
module main ;	/* module _main start */
...	...
sub(value2);	/* module _main__dummy1 start */
	...
	/* module _main__dummy1 end */
...	...
endmodule	/* module _main end */

In the example above, no code is initially generated corresponding to the `sub` module. Code generation starts with code for top module `main`. The code for the top module includes code for instantiated modules, once for each instantiation. The name `dummy1` is the auto-generated name assigned to the module instantiation.

2.3.2 Variable Declarations

Variable declarations from module scope get mapped to global scope in generated code. The data type of the declared variable would be either a standard C `int` data type or a pointer to the `Integer` class (see appendix C.7) depending of the kind of values the variable may take. Variables declared as ports in instantiated modules get replaced by the argument that connects to it, at module instantiation. Therefore, port declarations from instantiated modules do not find any corresponding declaration in target code. Following is an example that shows how Verilog variable declarations are translated to C code.

Verilog code	C code
module sub (pvalue1);	
reg [1:0] pvalue1 ;	
reg [1:0] value1 ;	
...	
endmodule	
module main ;	/* module _main start */
reg [40:0] value1 ;	Integer* _main_value1 ;
reg [1:0] value2 ;	int _main_value2 ;
...	...
sub(value2);	/* module _main__dummy1 start */
	int _main__dummy1_value1 ;
	...
	/* module _main__dummy1 end */
...	...
endmodule	/* module _main end */

As can be seen in the example, variables `value1` and `value2` from `main` module are generated as global variables. However, port variable `pvalue1` from instantiated module does not get generated. Non-port variable `value1` in module `sub` also gets converted to a global variable but the top module name and instance name are prefixed to its name.

Variable declarations in tasks, functions or named blocks are mapped to local variables in corresponding constructs. The input ports in functions are declared in parameter scope for the corresponding function. The ports of tasks are dealt with in a way similar to the module instance ports, that is, they are verbatim replaced. This is done to capture the copy-in/copy-out behavior of task ports.

2.3.3 Always blocks

`Always` blocks map to C functions with no arguments. If the `always` block does not have an event control, it will be called infinite number of times till the program encounters a $finish system call. If an `always` block has an event control it will get called each time the event occurs. The C function generated for the `always` blocks contains code corresponding to the Verilog behavioral constructs in the block. The names of generated functions are internally generated and the name of the containing module or module instance is prefixed to this name. The prefix ensures a unique name for each block. The following code snippet shows the C code that corresponds to an always block.

Verilog code	C code
always	_main__dummy1_func3()
	{
...	...
	}

In the example above, the always block belongs to the `sub` module instantiated in top module `main`. The name `func3` is internally generated.

2.3.4 Initial blocks

Initialization is not synthesizable and therefore is ignored during code generation. However, as an extension, if `-DALLOW_INITIAL` flag is passed to the C++ compiler while compiling *Verilog-to-C-compiler* from source, `initial` statements are supported. The `initial` blocks behave like `always` blocks except that they are executed exactly once, before any `always` block. The order in which the `initial` blocks are executed is uncertain.

2.3.5 Continuous Assignments

Continuous assignments are modeled as always blocks with an event control added to make the block combinatorial. The assignment gets mapped to a function with no arguments just like the always block. The corresponding function is called whenever the value of the expression on the right hand side of the assignment changes. The function call causes the right hand side to be reevaluated and assigned it to the left hand side.

2.3.6 Delays and Delay controls

Delays are non-synthesizable, therefore they are ignored. Similarly, delay controls are also dropped. A warning is generated to inform the user that the program might not behave the way it was expected to, because of the ignored delays.

2.3.7 Named and unnamed blocks

Verilog block constructs (begin-end) get mapped to the equivalent C block, marked by an opening and closing brace enclosing a list of statements. The statements are executed in the same order in which they appear in the Verilog code. Named blocks, unlike unnamed blocks, are allowed to have variable declarations as well. These declarations appear as local variables with the block as their scope. The parallel blocks(fork-join) are also executed sequentially. However, the ordering may or may not be maintained.

2.3.8 Tasks and Task enable statements

No code is generated for task declarations. However, the task enable statements get verbatim replaced by the task body. The task ports, if any, are replaced by the arguments to the task enable call. The task body is generated for each invocation in a way similar to unnamed blocks. This ensures that the global module members are accessible to the task. The replacement of ports by arguments to task enable call, models the copy-in/copy-out behavior of the task ports. Since, the task enables are verbatim replaced task disable constructs are not supported. All tasks are assumed to

be reentrant, so each task has a local copy of its variables. Since, a task can have any number of output/in-out ports, it does not easily map to a C function.

2.3.9 Functions and Function calls

Function declaration is modeled as a C function with the arguments representing the input ports of the function. The return type of the C function is a C int or a pointer to an object of the Integer class. The return value serves the purpose of the output port for the function. All functions are assumed to be automatic (recursive), that is, each function has a local copy of its variables.

2.3.10 System tasks

While the system tasks are accepted by the parser, most of them are non-synthesizable, and will be ignored during code generation. The ones that are not ignored are as follows.

$finish : The $finish system call is modeled by the exit system call in C. When a program encounters a $finish system call, it is assumed to have run successfully and would cause an exit(0) statement to be called.

$display : The display statement is non-synthesizable but supported to allow debugging. It is modeled as a library function displayf, which uses C stdio.h library to output the message to standard output. The similar $displayb, $displayh and $displayo statements are also correctly handled.

2.3.11 Behavioral statements

Verilog constructs that have a direct C counterpart get modeled accordingly. A Verilog `if-else` statement is modeled as a C `if-else` statement. Verilog `for` and `while` loops are both modeled as C `while` loops. Verilog `case` statement is modeled as a C `switch` statement. The Verilog `repeat` statements causes the code for the statement block to be generated multiple number of times, depending on the counter expression.

Chapter 3

Implementation

Verilog-to-C-compiler has been implemented in C++, along with a few C libraries. All general purpose data structures are written as C++ template classes. The compiler makes use of a couple of open source compiler tools, Flex[25] and Bison[26]. The program has been developed on linux[23] platform, using GNU gcc compiler[24]. The implementation mostly follows the object oriented paradigm. Dynamic dispatch of functions is often exploited and polymorphic containers are frequently used.

Verilog-to-C-compiler has been designed and implemented to allow easy extendability and reusability. As the high level design shown in figure 2.1 suggests, any given module does not depend on modules that lie on its right. The dependence is restricted to one direction and for one level only. For example, making changes to the parser module would not require changes to be made to the lexical analyzer or the code generator. As an exception, the code generator module depends on the semantic analyzer module and not on the optimization modules. The new optimization modules can be plugged in or removed without any effect on the semantic analyzer or the code generator.

3.1 Parser module

The parser module contains the lexical analyzer, the syntactic parser and the parse tree data structures.

The lexical analyzer consists of a list of rules defining what constitutes a valid token (see appendix A.2 for list of recognized keywords). The rule list acts as an input to a lexical analyzer generator program, like Lex[27] or Flex. The lexical analyzer generator generates a C++ program, that scans the input Verilog program to generate a stream of tokens. The lexical analyzer also acts as a preprocessor, by expanding `` `define `` macros and taking care of conditional compilation directives (`` `ifdef ``, `` `ifndef ``, `` `else ``, `` `elsif `` and `` `endif ``). Table 3.1 contains the file needed by the lexical analyzer generator.

Filename	Description
parser/verilog.l	Contains rule list for Lexical Analyzer, and function definitions that are used by the lexer

Table 3.1: File for Lexical Analyzer

The syntactic parser, like the lexical analyzer, consists of a list of rules defining what constitutes valid syntax for the input Verilog program. Table 3.2 gives a brief description of the files responsible for parsing. Each rule also has an associated action that gets executed, whenever the rule matches a construct in the input. This rule list acts as input to a parser generator program, like YACC[27] or Bison. The parser generator creates a C++ program that uses the lexical analyzer to scan the input, and identifies the syntactic constructs in the input. The actions that are associated with the rules identify invalid syntax and report errors, if any. If there are no errors, a parse tree is constructed corresponding to the input.

Filename	Description
parser/Makefile	Provides targets to generate dependencies, compile and clean the parser module.
parser/verilog.h	Contains macro definitions that are used by the parser.

Continued on next page...

Filename	Description
parser/verilog.ypp	Contains grammar rules for syntactic parser and the associated actions.
parser/p_scope.h	Class definitions for P_scope and P_scope_stk classes. An object of P_scope class represents a valid scope for the parser. An object of P_scope_stk class holds the scope stack.
parser/p_context.h	Class and function definitions for P_context class. An object of this class is associated with each node of parse tree.

Table 3.2: Filelist for Syntactic Parser module

The parser module additionally implements C++ classes for each supported Verilog construct. Each class additionally includes an object of class `P_context`. The parse tree data structures form a complex class hierarchy that accurately corresponds to Verilog syntax. The parse tree generated by this module consists of objects of the parse tree data structure classes. A brief description of the files associated with parse tree classes is given in table 3.3. A more detailed description of each parse tree class is given in appendix B.

Filename	Description
parser/p_decl.cpp, parser/p_decl.h	Class and function definitions for P_decl, P_module, P_primitive, P_function, P_task, P_basicdecl, P_rangedecl, P_netdecl, P_paramdecl and P_cont_assign classes.
parser/p_decl_print.cpp	Print functions for declaration classes.
parser/p_delay.cpp, parser/p_delay.h	Class and function definitions for P_delay class.

Continued on next page...

Filename	Description
parser/p_desc.cpp, parser/p_desc.h	Class and function definitions for P_desc class.
parser/p_dst.h, parser/p_dst.cpp	Class and function definitions for P_dst class.
parser/p_expr.cpp, parser/p_expr.h	Class and function definitions for P_expr, P_event_expr and P_lval classes.
parser/p_expr_print.cpp	Print functions for P_expr, P_event_expr and P_lval classes.
parser/p_id.cpp, parser/p_id.h	Class and function definitions for P_id_range and P_range classes.
parser/p_inst.cpp, parser/p_inst.h	Class and function definitions for P_module_prim_inst, P_module_prim_inst_list, P_gate_inst and P_gate_inst_list classes.
parser/p_port.cpp, parser/p_port.h	Class and function definitions for P_port and P_port_connect classes.
parser/p_stmt.cpp, parser/p_stmt.h	Class and function definitions for P_stmt, P_proc_stmt, P_begin_end_stmt, P_if_else_stmt, P_case_stmt, P_forever_stmt, P_repeat_stmt, P_while_stmt, P_for_stmt, P_delay_control_stmt, P_event_control_stmt, P_assign_stmt, P_wait_stmt, P_fork_join_stmt, P_task_enable_stmt, P_sys_task_enable_stmt, P_disable_stmt, P_deassign_stmt, P_release_stmt, P_send_event_stmt classes.
parser/p_stmt_print.cpp	Print functions for statement classes.
parser/p_systask.cpp, parser/p_systask.h	Class and function definitions for P_systask class.

Continued on next page...

Filename	Description
parser/parser_datastructures.h	Header file that includes definitions of all parser classes. It also contains class definitions for data structures for temporary use by the parser.
parser/p_list_print.h	Function for printing lists of parser data structures.

Table 3.3: Filelist for Parse Tree classes

3.2 Semantic Analyzer module

The semantic analyzer module consists of a parse tree to AST converter and semantic tree data structures. The parse tree to AST converter (Table 3.4) takes the parse tree as input and checks it for semantic consistency. This convertor also gathers certain information that could not be obtained by the parser. Identifier name resolution, which consists of resolving the data type and scope of an identifier, is performed at this stage. The symbols tables are populated accordingly in this pass.

Filename	Description
semantic_tree/Makefile	Provides targets to generate dependencies, compile and clean the semantic analyzer module.
semantic_tree/ parser_to_semantic_tree.cpp, semantic_tree/ parser_to_semantic_tree.h	Function definitions for the semantic analyzer. These functions act on the parse tree nodes and generate the AST.
semantic_tree/s_context.h	Class and function definitions for S_context, the context class associated with AST nodes

Continued on next page...

Filename	Description
semantic_tree/s_scope.h	Class definitions for S_scope and S_scope_stk classes. An object of S_scope class represents a valid scope for the semantic analyzer. An object of S_scope_stk class holds the scope stack.

Table 3.4: Filelist for Semantic Analyzer module

The semantic analyzer module also consists of class definitions of semantic tree data structures (Table 3.5, like the parse tree data structures. However, these data structures do not necessarily show similarity to Verilog syntax. The class definitions are made simpler to make the dependent modules simpler. Some constructs from parse tree gets mapped to another, logically equivalent but simpler construct in the AST. For example, a continuous assignment node (P_cont_assign) in the parse tree gets mapped into a always block node (S_proc_stmt) in the AST. Similarly, a declaration of a list of nets gets broken down into a list of declarations, with a net each. The AST consists of objects of the semantic tree data structure classes. The hierarchical nature of this tree is similar to the parse tree. Here is a brief description for the contained files, see appendix B for more detail.

Filename	Description
semantic_tree/s_decl.cpp, semantic_tree/s_decl.h	Class and function definitions for S_decl, S_module, S_primitive, S_function, S_task, S_basicdecl, S_rangedecl, S_netdecl, S_paramdecl and S_cont_assign classes.
semantic_tree/s_decl_print.cpp	Print functions for declaration classes.
semantic_tree/s_delay.cpp, semantic_tree/s_delay.h	Class and function definitions for S_delay class.
semantic_tree/s_desc.cpp, semantic_tree/s_desc.h	Class and function definitions for S_desc class.

Continued on next page...

Filename	Description
semantic_tree/s_dst.h, semantic_tree/s_dst.cpp	Class and function definitions for S_dst class.
semantic_tree/s_expr.cpp, semantic_tree/s_expr.h	Class and function definitions for S_expr, S_event_expr and S_lval classes.
semantic_tree/s_expr_print.cpp	Print functions for S_expr, S_event_expr and S_lval classes.
semantic_tree/s_id.cpp, semantic_tree/s_id.h	Class and function definitions for S_id_range and S_range classes.
semantic_tree/s_inst.cpp, semantic_tree/s_inst.h	Class and function definitions for S_mod_inst, S_mod_inst_list, S_gate_inst and S_gate_inst_list classes.
semantic_tree/s_port.cpp, semantic_tree/s_port.h	Class and function definitions for S_port and S_port_connect classes.
semantic_tree/s_stmt.cpp, semantic_tree/s_stmt.h	Class and function definitions for S_stmt, S_proc_stmt, S_begin_end_stmt, S_if_else_stmt, S_case_stmt, S_forever_stmt, S_repeat_stmt, S_while_stmt, S_for_stmt, S_delay_control_stmt, S_event_control_stmt, S_assign_stmt, S_wait_stmt, S_fork_join_stmt, S_task_enable_stmt, S_sys_task_enable_stmt, S_disable_stmt, S_deassign_stmt, S_release_stmt, S_send_event_stmt classes.
semantic_tree/s_stmt_print.cpp	Print functions for statement classes.
semantic_tree/s_systask.cpp, semantic_tree/s_systask.h	Class and function definitions for S_systask class.
semantic_tree/ semantic_tree_dst.h	Header file that includes definitions of all AST classes.
semantic_tree/s_list_print.h	Function for printing lists of AST classes.

Table 3.5: Filelist for AST classes

3.3 Util and Common modules

The util module provides general purpose data structures that are extensively used by all other modules. These data structures include polymorphic containers, like `list`, `set`, `stack` and `stable` (Symbol Table). These data structure have been implemented as C++ template classes to ensure they are usable for a variety of objects without compromising on type safety. They can be used for primitive data types, class objects or pointers. The dependence of this module over other modules is minimal allowing very easy reusability. Table 3.6 provides a brief description of the files that belong to the util module.

Filename	Description
util/Makefile	Provides targets to generate dependencies, compile and clean the util module. An additional install target installs the integer library, and some header files.
util/list.h	Class definitions for list_element, list and list_iterator classes
util/stack.h	Class definitions for stack class
util/st.h	Class definitions for stable (Symbol Table) and stable_iterator classes
util/set.h	Class definition for set class
util/integer.h, util/integer.c	Definition for struct Integer, and functions provided by Integer library
util/cint.h, util/cint.c	Functions for operations on C int data type
util/display.h	Contains function that handles the Verilog $display system call

Table 3.6: Filelist for Util module

The `list` class implements a doubly-linked list. References to both the start and end of the list are maintained, allowing addition and removal of elements from start and end in constant time (see appendix C.1). The `stack` class provides the basic stack functionality and internally uses the `list` class (see appendix C.3).

The `stable` class is a simple hash table implementation (see appendix C.4). The class requires a hash function which acts as a map for the key and the location where the record is stored. This hash function is used while comparing, inserting and finding elements. The `set` class is internally based on the symbol table class, with the key and the record being the same (see appendix C.6).

An `integer` library is also part of this module. The library supports integers of arbitrary width, with functions corresponding to all Verilog expression operators(see appendix C.7. Each bit in the integer can take four states similar to a Verilog integer. The library also provides Verilog unary reduction operations for C `int` data type.

The `common` module contains header files common to all other modules. These header files contain C macros and forward declarations of classes and functions. Logging facility that is used throughout the program is also part of this module, along with memory allocation, memory deallocation and string manipulation functions. A brief description of each file included in the module is given in table 3.7.

Filename	**Description**
common/Makefile	Provides targets to generate dependencies, compile and clean the common module.
common/all_classes.h	Forward declarations of all classes in the program.
common/extended_function_name.h	Class definitions of Extended_function_name. The class is used internally to hold names of function generated by the code generator module.
common/systask_defs.h	Contains the definitions of system tasks recognized by the parser.

Continued on next page...

Filename	Description
common/system_tasks.h	Class representing an entry in the table of recognized system tasks.
common/util.h, common/util.cpp	Provides utility functions for internal use by the program.
common/util_macros.h	Provides utility macros used extensively by other modules.

Table 3.7: Filelist for Common module

3.4 Interface module

The `interface` module defines the program interface that is used to drive the program. It consists of an `Interface` class, which has a couple of pure virtual functions. The `Interface` class is the parent class of `cli` class, which provides the command line interface to the program. This implementation provides an easy way to add a different interface, like a Graphical User Interface. Any new interface would need to inherit the Interface class, and implement the virtual functions. Table 3.8 provides a brief description if the files in the interface module.

Filename	Description
interface/Makefile	Provides targets to generate dependencies, compile and clean the interface module.
interface/interface.h	Class description for Interface class.
interface/cli.h	Class definition for Comman_line class.

Table 3.8: Filelist for Interface module

3.5 Optimization module

The `optimization` module consists of a traversal library and optimization functions corresponding to each optimization.

As mentioned earlier, the optimizations module is implemented as the visitor design pattern[19, 20]. The traversals library provides functionality to visit each node of the AST, independent of what needs to be executed on the node. The traversal is bottom up (post order) in nature.

An optimization routine consists of a function for each of the node that needs to be modified. Each optimization is implemented to make a pass at each node in the AST, and modify the ones required for the particular optimization. The optimization functions can communicate among themselves using members of a struct argument or using global variables. Table 3.9 lists the files included in the optimization module.

Filename	**Description**
optimizer/Makefile	Provides targets to generate dependencies, compile and clean the optimization module.
optimizer/execute_on_s.cpp	Contains function definitions for the AST traversals library.
optimizer/s_expr_o.cpp	Contains optimizations functions that act to AST nodes of S_expr kind.
optimizer/s_inst_o.cpp	Contains optimizations functions that act to AST nodes of S_mod_inst kind.
optimizer/s_stmt_o.cpp	Contains optimizations functions that act to AST nodes of S_stmt kind.

Table 3.9: Filelist for Optimization module

3.6 Code Generator module

The code generator module consists of code generator functions for each of the semantic tree data structure classes. The functions could be missing for the unsupported Verilog constructs (see appendix A.3). A top down traversal of the AST is made while generating corresponding C code for each tree node. Some additional code for including libraries is also generated, along with code for initializing and driving the generated model.

A couple of additional passes are made on the AST before invoking the code generation routine. The first iteration determines the data type of variables in generated code that would be optimal for a given Verilog variable. The second pass drops non-synthesizable Verilog constructs which do not get any code generated for them. This pass is made to increase the performance of the code generator and the readability and quality of generated code.

Filename	Description
code_generator/Makefile	Provides targets to generate dependencies, compile and clean the code generator module.
code_generator/ cint_compatibility.cpp	Provides functions that determine whether a node can fit in a C int data type or not.
code_generator/ code_generator.h, code_generator/ code_generator.cpp	Provides functions that are responsible for generating code for each node of the AST. Some supporting functions are also provided.
code_generator/ drop_non_synthesizable.cpp	Provides functions that drop some of the node from the AST for which no code would be generated by the code generator.
code_generator/ temp_var.h	Class definition for Temp_var class. The class represents a variable that is generated by the code generator, to store value temporarily.

Table 3.10: Filelist for Code Generator module

Chapter 4

Results and Conclusion

4.1 Experiments

A variety of tests were performed on the *Verilog-to-C-compiler* to test various aspects of the program.

4.1.1 Test environment

The tests were run on a computer with an Intel Pentium 4, with clock speed of 1.9GHz processor and 640MB of DDR memory, running Linux operating system (kernel version 2.6.11.4-20a). The *Verilog-to-C-compiler* and the generated C program were compiled using the GNU gcc (version 3.3.5) C++ and C compilers, respectively. The C++ compiler was passed the −DALLOW_INITIAL flag (for supporting initial blocks) , −g flag to allow debugging and −o3 flag for optimization. The C compiler was passed the −o3 optimization flag, while compiling the generated C program.

4.1.2 Test programs

The following test programs were run on the compiler to verify its correctness and measure the performance of the generated code.

- *Icarus Verilog Testsuite*

For testing the front end of the compiler, a testsuite maintained by Steve Wilson of Icarus Verilog[28] was used. The testsuite consists of 664 Verilog programs, each containing one or more modules. Each program consists of exactly one main file. However, some of the files include other files. Each test program checks a Verilog construction for correctness. The testsuite as a whole, provides an extensive coverage of syntax and semantics of various Verilog constructions. Not all programs in the testsuite are valid Verilog models, some of the programs are included to check error conditions.

- *Verilator Simulator Benchmark*

For testing the correctness of the back end (code generator) of the compiler and the performance of the generated simulator, Verilator simulator benchmark[29] was used.

The benchmark contains the following.

1. Synthesizable Verilog model of a 68K binary compatible, CRISC processor[30]. It consists of 24 modules, contained in 16 files and has nearly 3700 lines of code.

2. Synthesizable Verilog model of a simple asynchronous serial controller (SASC)[31]. This model consists of 3 files, containing 3 modules and approximately 350 lines of code.

3. A Verilog test bench consisting of 4 modules spread over 4 files and has approximately 120 lines of code.

The top module of the testbench consists of an initial block with a $finish system call following a delay control. Since, delay controls are ignored by the *Verilog-to-C-compiler*, the $finish system call causes the simulator to exit immediately. This

causes the the test bench to be non-synthesizable. As a work around, the initial block of the benchmark has been removed. A C driver function has been manually written that is responsible for controlling the simulation and eventually stoping it. This C function is responsible for calling appropriate functions and initializing the model. The simple driver function that is auto-generated by the tool can also be used in some special cases.

The experimental results have been reported in the following sections.

4.1.3 Correctness of *Verilog-to-C-compiler*

The modules described in chapter 2 have been tested independently, and also as a whole. In the tables in this section, the success count indicates the number of test cases that passed successfully, with or without warnings. The error count indicates number of test cases that caused the compiler to stop execution, gracefully. The crash count indicates the number of test cases that caused the compiler to stop unexpectedly. The unexpected result count is the number of test cases which passed when an error was expected or which failed when they were expected to pass. The warnings count is the total number of cases where a warning was raised. A single test case can cause multiple warning messages to be generated, which may or may not be of the same kind.

- *Parser*

The testcases were run with only the parser module in place. This tests the grammar specification of the parser and some of the functionality provided by the parse tree classes. Table 4.1 shows the results for the testsuite run for the parser module.

Success	659
Error(s) reported	5
Warning(s) reported	41
Crash count	0
Unexpected result(s)	0
Total	664

Table 4.1: Testsuite results for Parser module

The reported warnings include ignored compiler directives, like ‘timescale and ignored specify blocks. The reported errors include lexical errors and syntax errors.

Semantic Analyzer

The test cases were run with the parser and the semantic analyzer modules in place. This testsuite execution tests the semantic constraints put by the semantic analyzer and some functionality provided by the AST classes. Table 4.2 summarizes the results of the testsuite for the semantic analyzer.

Success	617
Error(s) reported	47
Warning(s) reported	411
Crash count	0
Unexpected result(s)	0
Total	664

Table 4.2: Testsuite results for Semantic Analyzer module

The error count and warning count shown in table 4.2, include the errors and warnings generated by the parser module (table 4.1). The reported errors include those caused by undeclared identifiers, or those accessed outside their scope. Variables declared more than once in the same scope also cause errors. Lastly, an error is reported if a unique candidate for the top module can not be identified. The reported warnings are due to implicit wire declarations.

Optimizers

The testsuite is run for the compiler front end, along with the optimization modules in place. The results have been summarized in table 4.3.

This run of the testsuite checks the optimization routines, and the traversal library for accuracy. It has been manually verified that the correct optimizations indeed take place where applicable. The two additional errors are division by zero errors, generated by the constant folding optimization module. However, it may be

Success	615
Error(s) reported	49
Warning(s) reported	5479
Crash count	0
Unexpected result(s)	0
Total	664

Table 4.3: Testsuite results for Optimizer modules

noted that the number of warnings shoots up drastically. Since, the optimization module drops non-synthesizable constructs, it generates a warning for each dropped construct. The new warnings correspond to dropped constructs, like delay controls and task disable statements.

Code generator

With the code generator included, the software becomes complete. So, the testcases for the code generator are the same as those for the complete application. Almost all test programs in Icarus Verilog testsuite are non-synthesizable. Since, this module only supports a synthesizable subset of Verilog, the testsuite could not be used for testing the complete application. To test this module the Verilator benchmark is used. The test program compiles without errors, but generates lots of warnings. Table 4.4 shows the test results for this module.

Parser warnings	23
Semantic Analyzer warnings	182
Code generator warnings	560
Total warnings	770
Unexpected warnings	0
Lines of code in Verilog program	4200
Lines of code in C program	5300

Table 4.4: Testsuite results for correctness of Code Generator module

	Generated Simulator	ModelSim
Number of simulation cycles executed	10,000,000	10,000,000
Total time taken for execution	8.9 seconds	42 seconds
Average number of cycles/second	1,120,000	238,000

Table 4.5: Comparative Benchmark results

The warnings generated by the parser are because of ignored compiler directives. The semantic analyzer warnings are caused by implicitely declared Verilog nets and ports. The warnings generated by code generator correspond to dropped non-synthesizable and unsupported constructs, chiefly delays and delay controls.

The size of the output program is of the same order as the input program. However, some functionality is provided to the generated program by the Integer library, the size of which is not included in count.

4.1.4 Performance of generated program

Providing a fast simulation is an important goal of this thesis. Therefore, the performance of the generated simulator is crucial to this work. For testing the performance of the generated simulator, Verilator simulator benchmark was used. The compiler successfully parsed the Verilog model and generated a C program. The C program was successfully compiled to get an executable file. The executable file was successfully run on the native machine, to simulate the Verilog RTL model.

For comparing the performance of the simulator generated by the *Verilog-to-C-compiler*, the same benchmark was run on **ModelSim SE PLUS 6.0**[32] as well. Optimizations were enabled for ModelSim as well as the C compiler that is used to compile the generated simulator. Table 4.5 shows the benchmark results for both the simulator generated by *Verilog-to-C-compiler* and ModelSim.

As can be seen from the table 4.5, the simulator generated by *Verilog-to-C-compiler* takes one-fifth the time to execute the benchmark Verilog model, than ModelSim.

4.2 Conclusion and Future Work

The primary objective of this thesis, of providing a fast simulator for synthesizable subset of Verilog has been achieved. The *Verilog-to-C-compiler* converts Verilog RTL to a C program. The generated C program can be compiled using a standard C compiler. The generated executable can be run to simulate the hardware model described by the Verilog RTL model. The performance of the generated simulator compares favorably to conventional Verilog logic simulators (see section 4.5 for detail).

The front end of the compiler has been designed, implemented and tested to work with a large subset of Verilog 2001 standard. This allows all kinds of Verilog programs to be accepted by the compiler. However, the back end works only for a more restrictive subset. Most non-synthesizable Verilog constructs are either partially supported or unsupported (see appendix A.3 for detailed list). As a workaround, some of these constructs can be modeled as synthesizable constructs. For example, a `trior` net is not synthesizable, but it can be modeled as an `or` operation on all the drivers. Such transformations may be supported in future by the compiler itself. Also, the new features introduced in the IEEE Verilog 2005 standard[17] have not been implemented in this software. These features may be incorporated in future.

More importantly, the simulator generator by *Verilog-to-C-compiler* currently is only slightly more efficient that conventional Verilog logic simulators. However, the performance of the product can be improved further. Only, some simple optimizers have been implemented in the compiler. The performance of the generated simulator can be improved further by implementing more optimizations, like loop unfolding and redundancy removal. The integer library (see appendix C.7) is an importance part of the generated simulator. The performance of the simulator can be improved by optimizing the integer library.

References

[1] A. V. Aho, R. Sethi, J. D. Ullman. *Compilers: Principles, Techniques and Tools*, Addison-Wesley Publishing Company

[2] Rajat Moona. *Processor Models for Retargetable Tools* rsp, p. 34, 11th IEEE International Workshop on Rapid System Prototyping (RSP'00), 2000.

[3] Subhash Chandra, Rajat Moona. *Retargetable Functional Simulator Using High Level Processor Models* vlsid, p. 424, 13th International Conference on VLSI Design, 2000.

[4] V. Rajesh. *A Generic Approach to Performance Modeling and Its Application to Simulator Generator* Master's thesis, Department of Computer Science and Engineering, IIT Kanpur, August 1998.
http://www.cse.iitk.ac.in/users/simnml/docs/simnml.pdf

[5] Rajiv A. R. *Retargetable Profiling Tools and their Application in Cache Simulation and Code Instrumentation* Master's thesis, Department of Computer Science and Engineering, IIT Kanpur, December 1999.
http://www.cse.iitk.ac.in/users/simnml/thesis/rajiv.pdf

[6] Sarika Kumari. *Generation Of Assemblers Using High Level Processor Models* Master's thesis, Department of Computer Science and Engineering, IIT Kanpur, February 2000.
http://www.cse.iitk.ac.in/users/simnml/thesis/sarika.pdf

[7] Prithvi Pal Singh Bisht. *Generic Disassembler Using Processor Models* Master's thesis, Department of Computer Science and Engineering, IIT Kanpur, February

2002.
http://www.cse.iitk.ac.in/users/simnml/thesis/prithvi.pdf

[8] Surendra Kumar Vishnoi. *Functional Simulation Using Sim-nML* Master's thesis, Department of Computer Science and Engineering, IIT Kanpur, May 2006.
http://www.cse.iitk.ac.in/users/simnml/thesis/surendra.pdf

[9] Wilson Snyder, Paul Wasson and Duane Galbi. *Verilator - Convert Verilog code to C++/SystemC*
http://www.veripool.com/verilator.html

[10] S. T. Cheng et al. *Compiling Verilog into Timed Finite State Machines* International Verilog Conference'95.
http://vlsi.colorado.edu/~vis/doc/stcheng_verilog95.ps.gz

[11] S.-T. Cheng. *vl2mv Manual* Master Thesis, UCB ERL Technical Report M94/37
http://vlsi.colorado.edu/~vis/doc/stcheng_M94_37.ps.gz

[12] BLIF—MV Manual.
http://vlsi.colorado.edu/~vis/doc/blifmv/blifmv/blifmv.html

[13] VIS *Verification Interacting with Synthesis* Joint project of University of California at Berkeley, the University of Colorado at Boulder, and at the University of Texas, Austin. http://vlsi.colorado.edu/~vis/

[14] VSP^{TM} and RealView^{TM}. *Carbon's Virtual System Prototype and RealView*
http://carbondesignsystems.com/corpsite/products/html/vsp-product-brief.html

[15] VSP^{TM} Product Overview.
http://carbondesignsystems.com/corpsite/products/vsp-html-version.html

[16] SOC-VSP^{TM} Product Overview.
http://carbondesignsystems.com/corpsite/products/soc-vsp-html-version.html

[17] IEEE Verilog 2001 standard.
http://standards.ieee.org/announcements/verilog2001.html

[18] SystemC. http://www.systemc.org/

[19] E Gamma, R Helm, R Johnson, J Vlissides. Book: *Design Patterns: Elements of Reusable Object-Oriented Software*

[20] *Visitor Design Pattern - C++ implementation* http://www.swe.uni-linz.ac.at/research/deco/designPatterns/Visitor/visitor.abstract.html

[21] Frank Buschmann. *Building Software with Patterns* http://citeseer.ist.psu.edu/buschmann99building.html

[22] Pipes and Filters. http://vico.org/pages/PatronsDisseny/Pattern%20Pipes%20and%20Filters/

[23] Linux. *Unix-type open source operating system* http://www.linux.org/

[24] The GNU Compiler Collection. *Open source free C/C++ compiler* http://gcc.gnu.org/

[25] flex. *Fast Lexical Analyzer Generator* http://flex.sourceforge.net/

[26] Bison. *GNU parser generator* http://www.gnu.org/software/bison/

[27] Lex, Flex, YACC, Bison. http://dinosaur.compilertools.net/

[28] Icarus Verilog testsuite. http://sourceforge.net/projects/ivtest/

[29] Verilator Simulator benchmarks. http://www.veripool.com/verilog_sim_benchmarks.html

[30] Shawn Tan. *CRISC processor Verilog model.* http://www.opencores.org/cvsget.cgi?module=k68&tag= http://www.opencores.org/people.cgi/info/sybreon

[31] Simple Asynchronous Serial Controller http://www.opencores.org/projects.cgi/web/sasc

[32] ModelSim. http://www.model.com/products/default.asp

Appendix A

Supported Verilog constructs

A.1 List of recognized Verilog keywords

always	and	assign	automatic	begin
buf	bufif0	bufif1	case	casex
casez	cmos	deassign	default	defparam
disable	edge	else	end	endcase
endfunction	endmodule	endprimitive	endspecify	endtable
endtask	event	for	forever	force
fork	function	highz0	highz1	if
initial	inout	input	integer	join
large	localparam	macromodule	medium	module
nand	negedge	nmos	nor	not
notif0	notif1	or	output	parameter
pmos	posedge	primitive	pull0	pull1
pulldown	pullup	rcmos	real	realtime
reg	release	repeat	rnmos	rpmos
rtran	rtranif0	rtranif1	scalered	signed
small	specify	specparam	strong0	strong1
supply0	supply1	swire	table	task

teslaTimer	time	tran	tranif0	tranif1
tri	tri0	tri1	triand	trior
trireg	unsigned	vectored	wait	wand
weak0	weak1	while	wire	wor
xnor	xor			

A.2 Supported Verilog Operators

Unary Operators $+, -, !, \sim, \&, |, \sim \&, \sim |, \wedge \sim, \sim \wedge$

Binary Arithmetic Operators $+, -, *, /, \%, **$

Binary Logical Operators $\&\&, ||$

Binary Bitwise Operators $\&, |, \wedge, \sim \wedge, \wedge \sim, \sim \&, \sim |$

Shift Operators $<<, <<<, >>, >>>$

Comparison Operators $<, >, <=, >=, ==, != , ===, !==$

Ternary Operator $? :$

A.3 List of supported Verilog constructs

Lexical Constructs

Operators : (see appendix A.2)

White Spaces

Single line and Multiple line comments

Numbers : All decimal, binary, hexadecimal and octal formats supported.

Strings : Operations on strings are not supported.

Identifiers : All kinds of valid Verilog identifier names (Simple, Escaped and System names) are supported.

Keywords : (see appendix A.1)

Text Substitutions : (‘define macros)

Data types

Registers : Arbitrary width registers are supported.

Nets : Only wire type nets are supported, others are treated as wire. Implicit declarations are supported. Initialization is converted to continuous assignment.

Integers

Parameters : parameter and localparam is supported.

Behavioral modeling

Continuous assignments : Strength and delays are ignored.

Procedural assignments : Delay and event controls are ignored.

Conditional statement

Case statement : Supported for expression with width $\leq$ width of C int.

Loops : For, While and Repeat Loops are supported.

Blocks : Both named and unnamed blocks are supported. Parallel blocks are converted to serial blocks. Disabling of blocks is not supported.

Timing controls : Delay controls are ignored. Event controls are supported when they occur just under procedural blocks. Event controls other than just under always and initial blocks are not supported.

Tasks : Task disable statement is not supported.

Functions : Hierarchical access is limited to module scope.

Initial : Ignored. If compiler is compiled with -DALLOW_INITIAL flag, then Initial statements are supported.

Always : Supported with or without event controls.

Hierarchical structures

Modules : Macromodule treated same as module.

Module Instantiations

Ports : All input, inout and output ports are supported, though the direction is not enforced except for functions.

Hierarchical names : Supported, but hierarchical access across modules is not allowed.

Scope rules

A.4 List of unsupported Verilog constructs

Data types

Time

Real Numbers

Events

Parameter : defparam statement is not supported

Behavioral modeling

Loops : Forever loops are not supported

Gate and switch level modeling

Accepted by parser, but ignored by code generator.

User defined primitives

Accepted by parser, but ignored by code generator.

Specify blocks

Ignored by parser.

Appendix B

Class diagram of Parser module

The classes implemented by the parser module and a brief description of the Verilog construct each class represents is as follows. Figure B.1 shows the dependencies among top level classes.

P_dst: Parent class of all parse tree classes. Functions are implemented in this class that handle the context for each Verilog construction in the input Verilog description.

P_decl: Parent class of all declarations(see figure B.2).

P_stmt: Parent class of all behavioral statements(see figure B.3). Also represents an empty statement.

P_expr: Represents all kinds of Verilog expressions.

P_desc: Represents the Verilog model description. Consists of a list of modules and User Defined Primitives.

P_context: Represents the context associated with each construct. A context consists of start and end line numbers, along with the filename in which the construct appeared. If a construct spans multiple files, the file where the construct started is considered.

P_lval: Represents the subset of Verilog expressions that can appear on the left hand side of an assignment.

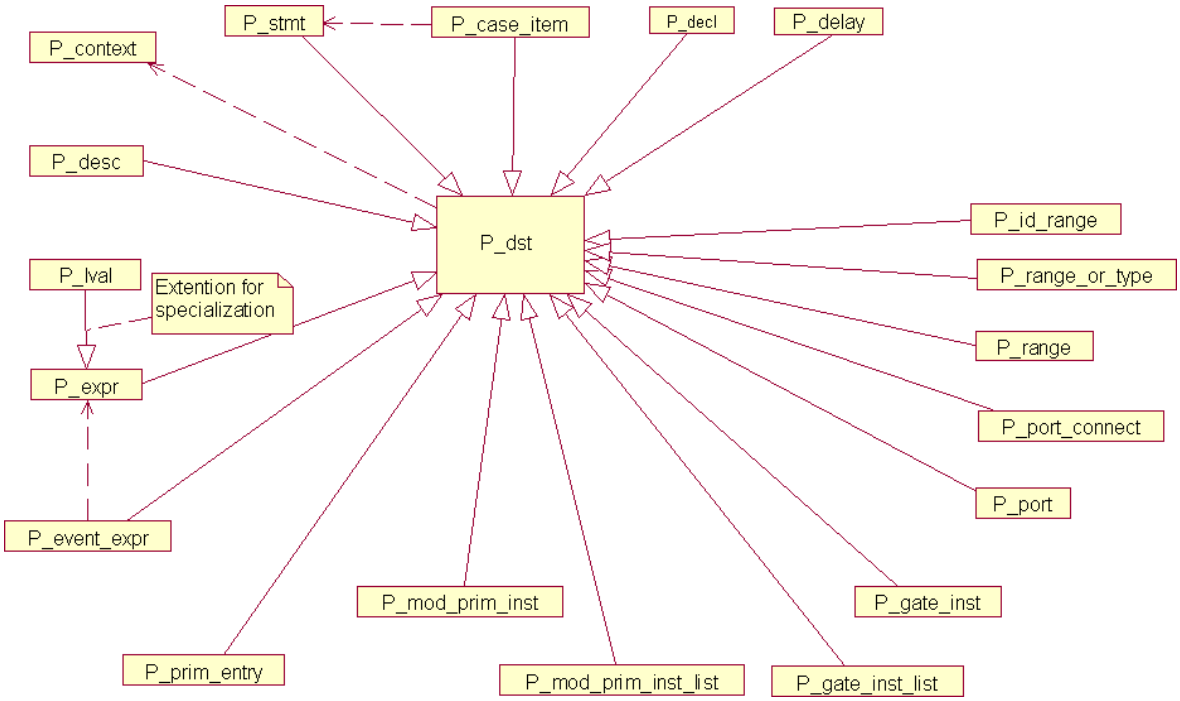

Figure B.1: Class diagram for parser datastructure

P_event_expr: Represents an expression that represents an event in Verilog.

P_delay: Represents various kinds of delay specification.

P_id_range: Represents a Verilog identifier, along with an associated range, if any.

P_range_or_type: Represents the possible return values of a Verilog function.

P_range: Represents the lower and upper limits of Verilog range.

P_port: Represents Verilog ports for modules, tasks and functions.

P_port_connect: Represents the arguments passed while module instantiation that connects to the ports.

P_module_prim_inst_list: Represents a list of module or UDP instantiation.

P_module_prim_inst: Represents a single module or UDP instantiation.

P_prim_entry: Represents a single entry in a UDP table.

P_gate_inst_list: Represents a list of Verilog gate instantiation.

P_gate_inst: Represents a single Verilog gate instantiation.

Declaration classes

The P_decl is a generic class for Verilog declaration. The declaration class (P_decl) is extended by a number of classes (figure B.2), each subclass representing a kind of Verilog declaration. A brief description of the various declaration classes is as follows.

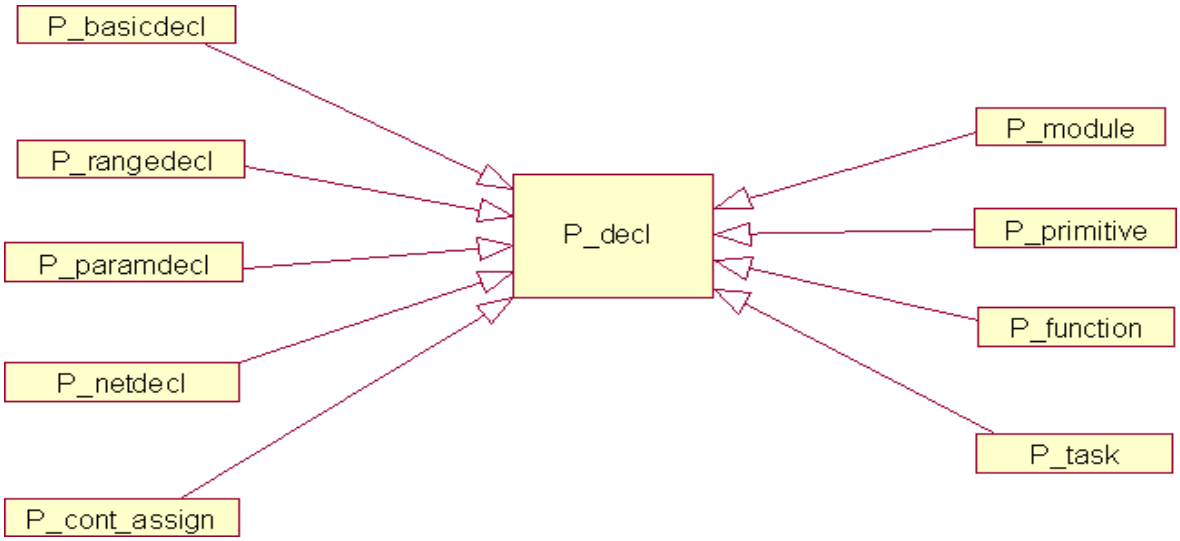

Figure B.2: Class diagram for declarations in parser datastructure

P_basicdecl: Represents a variable declaration with no range.

P_rangedecl: Represents a variable declaration with range.

P_paramdecl: Represents a parameter variable declaration.

P_netdecl: Represents declaration of a Verilog net.

P_cont_assign: Represents a continuous assignment to a Verilog net.

P_module: Represents declaration of a Verilog module.

P_primitive: Represents declaration of a Verilog User Defined Primitive(UDP).

P_function: Represents declaration of Verilog function.

P_task: Represents declaration of Verilog task.

Statement classes

The statement class (P_stmt) is inherited by a number of other classes, each subclass representing a kind of Verilog statement. Figure B.3 depicts the dependencies among the statement classes.

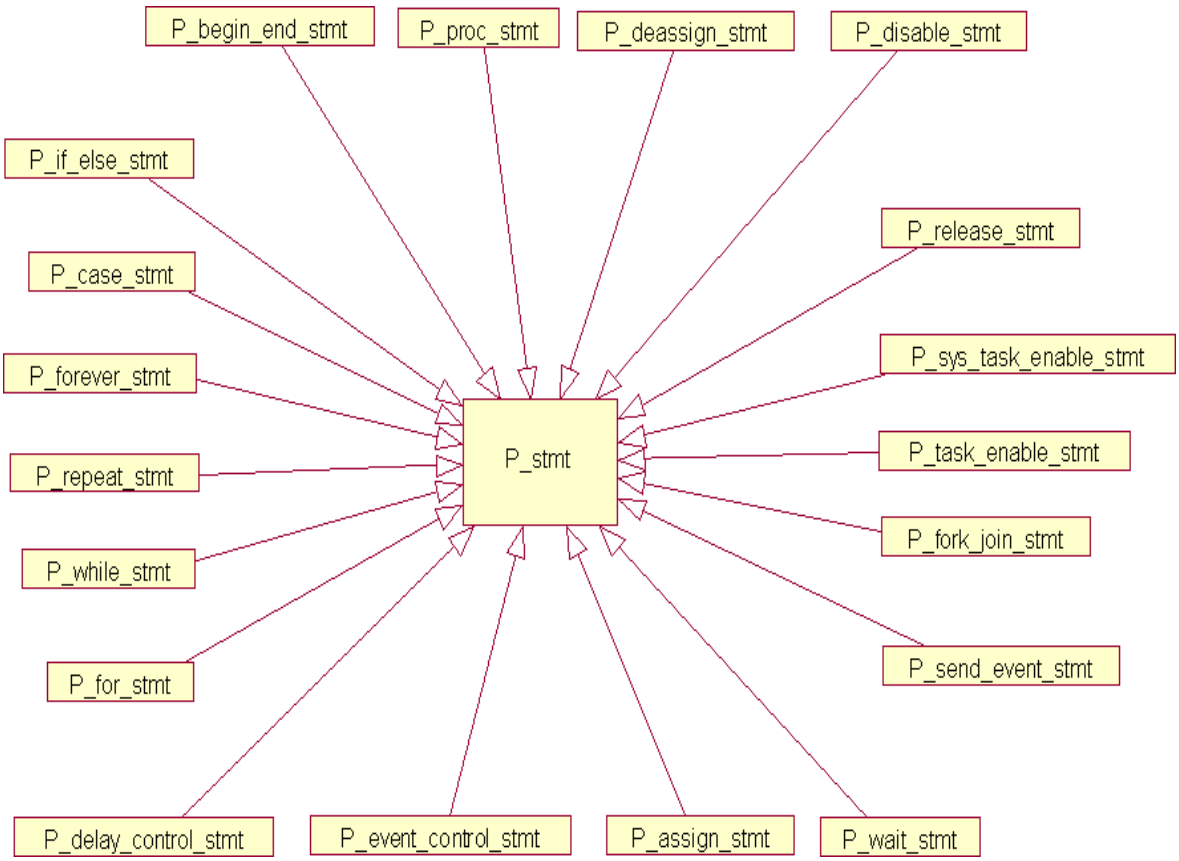

Figure B.3: Class diagram for statements in parser datastructure

A brief description of the various statement classes is as follows.

P_proc_stmt: Represents the Verilog initial and always blocks.

P_begin_end_stmt: Represents the Verilog sequential (being-end) block.

P_if_else_stmt: Represents the Verilog if-else statement.

P_case_stmt: Represents the Verilog case statement.

P_forever_stmt: Represents the Verilog forever loop.

P_repeat_stmt: Represents the Verilog repeat loop.

P_while_stmt: Represents the Verilog while loop.

P_for_stmt: Represents the Verilog for loop.

P_delay_control_stmt: Represents a generic Verilog statement with a delay control.

P_event_control_stmt: Represents a generic Verilog statement with a delay control.

P_assign_stmt: Represents the Verilog assignment statement.

P_wait_stmt: Represents the Verilog wait statement.

P_fork_join_stmt: Represents the Verilog parallel (fork-join) block.

P_task_enable_stmt: Represents the Verilog task enable statement.

P_sys_task_enable_stmt: Represents the Verilog system task call statement.

P_disable_stmt: Represents the Verilog task disable statement.

P_deassign_stmt: Represents the Verilog deassign statement.

P_release_stmt: Represents the Verilog release statement.

P_send_event_stmt: Represents the Verilog raise event statement.

All classes except the P_module_prim_inst_list and P_gate_inst_list classes have an equivalent AST class, with a name starting with a S_ instead of a P_.

Appendix C

Utility Classes

The util module implements various general purpose data structures. The important ones are as follows.

C.1 list class

This is an implementation of a doubly linked list. Looking, adding and removing elements from both start and end have constant time complexity. Other operations have linear time complexity.

Members

- **first**: Points to the first element of list.
- **last**: Points to the last element of list.
- **length**: The number of elements in the list.

Functions

- **first_element**: Returns the first item in the list.
- **last_element**: Returns the last item in the list.

- **add_to_start**: Adds an element or another list to the start of list.
- **add_to_end**: Adds an element or another list to the end of list.
- **remove_from_start**: Removes and returns the first element in the list.
- **remove_from_end**: Removes and returns the last element in the list.
- **remove**: If the argument exists in the list, it is removed.
- **replace**: If the first argument exist in the list, it is replaced by the second argument. If multiple occurrences exist in the list, only the first occurrence would get replaced. Boolean value true is returned to indicate if the replacement was successfully done. A false value indicates that the element was not found in the list.
- **contains**: Checks if the argument exists in the list.

C.2 list_iterator class

This class is used to iterate over all elements in a list, without affecting it. The iterator becomes unpredictable if the list changes during iteration.

Members

- **l**: The list being traversed.
- **handle**: A pointer to the next element to be returned.
- **remain**: The number of elements left to be traversed.

Functions

- **has_more_items**: Returns true if there are more elements to be traversed.
- **next_item**: Returns the next item.

C.3 stack class

The stack class implementation uses the list class internally. All operations listed below have a constant time complexity.

Members

- **stk**: A pointer to the list object that holds the stack elements.

Functions

- **push**: Push an element into the stack
- **pop**: Removes and returns an element from top of stack
- **top**: Returns the element on top of stack. Same as a pop followed by a push. It is the same as a peak operation.
- **depth**: Returns the number of elements in the stack.
- **empty**: Returns true is the stack is empty, false otherwise.

C.4 stable class

The stable (Symbol Table) class implements a simple hash table. Lookup, insert and remove operations have on an average constant time complexity.

Members

- **hash**: A function pointer to the function used to compute hash values of a key.
- **compare**: A function pointer to the function used to compute two keys for equality.
- **bins**: A pointer to an array of bins that hold the records.
- **num_bins**: Number of bins currently allocated.

- **num_entries**: Number of entries in the table.
- **max_density**: The average number of records in each bin when an addition causes the hash table to be rehashed to a larger table.
- **reorder_flag**: If this flag is true, a record access will cause the accessed element to be bumped to head of its bin.
- **grow_factor**: The number of times the capacity of hash table grows when it is almost full.

Functions

- **lookup**: Returns true if the element is contained in the table.
- **insert**: Inserts a key along with a record.
- **remove**: Removes the element, if present.

C.5 stable_iterator class

This class is used to iterate over all elements in a symbol table, without affecting it. The iterator becomes unpredictable if the symbol table changes during iteration.

Members

- **table**: A pointer to the symbol table being iterated over.
- **index**: Index of the bin till which point the table has been traversed.
- **empty**: Pointer to the element in a bin till which point the table has been traversed.

Functions

- **next**: Returns the next element in the symbol table, NULL if no more element is present.

C.6 set class

This implementation of a set uses the stack class internally. Add, remove and find operations have a constant time complexity, on an average.

Members

- **s**: A pointer to the symbol table that holds the set elements.

Functions

- **add**: Add an element to the set.
- **remove**: Remove the argument from the set, if present.
- **find**: Returns true is the argument exists in the set.

C.7 Integer struct

The structure represents a Verilog variable. It can be used to save numbers of arbitrary width.

Members

- **bitcount**: The number of valid bits in the number.
- **data**[]: An array of C unsigned long, used to hold the data part of the number.
- **xz**[]: An array of C unsigned long, used as a vector to mark bits that are X or Z.

The following table explains how the value of a Verilog bit is interpreted in this datastructure.

Bit in xz	Bit in data	Verilog bit
0	0	0
0	1	1
1	0	x
1	1	z

C.8 Integer library

This library provides functions to manage the Integer struct described in appendix

Functions

- **newInteger**: Returns a new Integer with value 0.
- **fromInteger**: Returns a new Integer which is a copy of the argument.
- **fromString**: Returns a new Integer with value obtained from the string argument.
- **fromLong**: Returns a new Integer with value obtained from the C long argument.
- **fromUlong**: Returns a new Integer with value obtained from the C unsigned long argument.
- **fromInt**: Returns a new Integer with value obtained from the C int argument.
- **fromUint**: Returns a new Integer with value obtained from the C unsigned int argument.
- **copy**: Copies the second Integer argument to the first Integer argument.
- **copyLong**: Copies the second long argument to the first Integer argument.
- **bitSelect**: Copies a part of the first Integer argument to the second Integer argument.
- **logicalNot**: Invert the logical value in the argument.

- **arithmeticNot**: Performs a bitwise inversion on the argument (Each 0 bit is changed to 1, and each 1 bit is changed to 0).
- **neg**: Invert the arithmetic sign of the argument.
- **shiftLeft**: Perform a left shift operation on the argument.
- **shiftRight**: Perform a right shift operation on the argument.
- **getLong**: Returns a C long if the argument value can fit in it. Returned value is inaccurate otherwise.
- **toString**: Converts the Integer argument to a string.
- **isZero**: Returns true if the argument is zero.
- **isNegative**: Returns true if the argument is negative.
- **isPositive**: Returns true if the argument is positive.
- **hasXZ**: Returns true if the argument is contains at least one bit that is either X or Z.
- **equal2**: Checks two numbers for equality(Verilog == operator).
- **equal3**: Checks two numbers for case equality(Verilog === operator).
- **add**: Perform arithmetic addition on two number(Verilog + operator).
- **sub**: Perform arithmetic substraction on two number(Verilog − operator).
- **mul**: Perform arithmetic multiplication on two number(Verilog ∗ operator).
- **divide**: Perform arithmetic division on two number(Verilog / operator).
- **mod**: Find the arithmetic mod on two number(Verilog % operator).
- **exponent**: Find the value of first argument multiplied to itself second argument number of times(Verilog ** operator).
- **unaryNeg**: Finds the arithmetic negation of a number(Verilog − operator).
- **unaryNot**: Finds the logical negation of a number(Verilog ! operator).
- **unaryCompliment**: Finds the bitwise negation of a number(Verilog ∼ operator).
- **unaryAnd**: Finds the AND reduction of the number(Verilog & operator).

- **unaryOr**: Finds the OR reduction of the number(Verilog | operator).
- **unaryNand**: Finds the NAND reduction of the number(Verilog $\sim$ & operator).
- **unaryNor**: Finds the NOR reduction of the number(Verilog $\sim$ | operator).
- **unaryXor**: Finds the XOR reduction of the number(Verilog $\wedge$ operator).
- **unaryXnor**: Finds the XNOR reduction of the number(Verilog $\wedge \sim$ *or* $\sim \wedge$ operator).
- **logicalAnd**: Finds the logical AND of two numbers(Verilog && operator).
- **logicalOr**: Finds the logical OR of two numbers(Verilog || operator).
- **bitwiseAnd**: Finds the bitwise AND of two numbers(Verilog & operator).
- **bitwiseNand**: Finds the bitwise NAND of two numbers(Verilog $\sim$ & operator).
- **bitwiseOr**: Finds the bitwise OR of two numbers(Verilog | operator).
- **bitwiseNor**: Finds the bitwise NOR of two numbers(Verilog $\sim$ | operator).
- **bitwiseXor**: Finds the bitwise XOR of two numbers(Verilog $\wedge$ operator).
- **bitwiseXnor**: Finds the bitwise XNOR of two numbers(Verilog $\wedge \sim$ *or* $\sim \wedge$ operator).
- **lessThan**: Finds whether the first argument is less than the second argument(Verilog < operator).
- **lessThanEqual**: Finds whether the first argument is less than or equal to the second argument(Verilog <= operator).
- **greaterThan**: Finds whether the first argument is greater than the second argument(Verilog > operator).
- **greaterThanEqual**: Finds whether the first argument is greater than the second argument(Verilog >= operator).
- **leftShiftUnsigned**: Treats the first argument as an unsigned number and performs a left shift operation on it. The second argument give the shift count. The result is put into the third argument(Verilog << operator).

- **leftShiftSigned**: Treats the first argument as a signed number and performs a left shift operation on it. The second argument give the shift count. The result is put into the third argument(Verilog <<< operator).
- **rightShiftUnsigned**: Treats the first argument as an unsigned number and performs a right shift operation on it. The second argument give the shift count. The result is put into the third argument(Verilog >> operator).
- **rightShiftSigned**: Treats the first argument as a signed number and performs a right shift operation on it. The second argument give the shift count. The result is put into the third argument(Verilog >>> operator).

Appendix D

Using the *Verilog-to-C-compiler*

Following is a brief description of the usage scenario for *Verilog-to-C-compiler*.

`vl2c [-d] [-o <output_file>] [-t <top_module_name>] [-nomain] <input_files>`

−d	Debug mode, causes parser and semantic tree intermediates to be dumped
−t top_module_name	Overrides heuristic to find out top module. Mandatory when a unique top module can not be determined
−nomain	Instead of the main function, a simple function with name same as top module is generated
−o output_file	Compiler output gets saved to the output_file. Otherwise, the name of the first input_file is used.
input_files	A list of files that are to be compiled

To compile the generated C code, use the following command:

```
$ gcc -lm -linteger [-L <path/to/integer_library>] [-I
<path/to/util/headers>] [<c_wrapper_file.c>] [-o <output_filename>]
[-o3] <generated_file.c>
```

If the Integer library is installed to some system directory (use `make install` to do this), the -L and -I options can be avoided.

To execute the compiled binary for the simulator (The first step can be ignored if the integers library exists in the path):

```
$ export LD_LIBRARY_PATH = path/to/integer/library
$ ./output_filename
```