RTL C-Based Methodology for Designing and Verifying a Multi-Threaded Processor

Luc Séméria[1], Andrew Seawright[2], Renu Mehra[1], Daniel Ng[3], Arjuna Ekanayake[4], Barry Pangrle[1]

[1]Synopsys, Inc. [2]O-In Design Automation, Inc. [3]Broadcom, Inc.
{lucs,renu,pangrle}@synopsys.com andrew@0-in.com dng@broadcom.com [4]arjuna@stanfordalumni.org

Abstract – A RTL C-based design and verification methodology is presented which enabled the successful high speed validation of a 7 million gate simultaneous multi-threaded (SMT) network processor. The methodology is centered on statically scheduled C-based coding style, C to HDL translation, and a novel RTL-C to RTL-Verilog equivalence checking flow. It leverages improved simulation performance combined with static techniques to reduce the amount of RTL-Verilog and gate-level verification required during development.

Categories – B.5.2 [**Register-Transfer-Level Implementation**] Design Aids: *Automatic synthesis, Hardware description languages, Optimization, Simulation,Verification*.

General Terms -- Design, Verification, Performance, Languages.

Keywords – C/C++, RTL, design, verification, formal equivalence checking.

1. INTRODUCTION

With increasing design complexity, different techniques have been developed to bridge the gap between the amount of logic that can be put on a chip and the design and verification effort necessary to build such a chip. Design reuse and moving to higher levels of abstraction are two promising techniques to boost productivity. For the development of high-performance processors, the benefits of standard behavioral synthesis techniques are not obvious. Nevertheless, the need for an efficient and fast design entry language becomes even more crucial. In this paper, we present how a register transfer level (RTL) C/C++ design methodology is used to accelerate the design and verification of a complex processor.

C and C++ can be efficiently compiled enabling fast simulation and a flexible way to scale the number of concurrent simulations. In the case of a processor, a fast synthesizable model also represents a perfect instruction set simulator, which facilitates the early development of software as well as system verification early in the design process.

Using a programming language for hardware implementation has several challenges. The first challenge consists of defining a fast and accurate representation of the hardware. Speed is a must to reduce simulation time compared to RTL Verilog. RTL C also has to be accurate so that the design can be debugged early on, at the source level. In this paper, we present our RTL C coding style. To increase simulation speed, our C model is statically scheduled and can use ANSI C types. For higher accuracy, a C++ object library is used to represent bit-accurate data types. Random initialization of variables is also performed to check reset behavior.

The second challenge consists of integrating a C-based synthesis flow into the overall ASIC design flow. This can be done by automatically translating the RTL C description into RTL hardware-description language (HDL), or directly into gates. This translation step represents an extra step in the ASIC design flow, and can add additional risks. C to HDL translation tools are fairly recent and have not yet been validated on many large production designs. Moreover, evolving and competing C/C++ standards result in various interpretations making it difficult to develop and use such tools.

A C lint tool is used at the design entry to catch the most common coding style errors early on in the design process before C to HDL translation. Further, formal equivalence checking is commonly used to verify the different synthesis steps. Our methodology provides a novel formal equivalence checking flow to compare the RTL C and the HDL generated. It uses a separate and additional C to HDL translation step, followed by equivalence checking between the two translated RTL HDL models.

A final challenge is to provide a method to integrate a C-based design core with IP blocks coded using traditional RTL HDL descriptions. Co-simulation is used to integrate RTL C models with other IP Verilog models. Co-simulation is also used to validate the RTL Verilog and netlists within our C testbench environment.

The contributions of this paper are the following. We present a verification flow for C-based design. The design entry is bit-accurate RTL C model that is automatically translated into HDL for synthesis. Equivalence checking is used to formally verify the RTL C and HDL models are equivalent. Simulation of the design is also performed at each step of the design process.

The rest of this paper is organized as follows. Related work is presented in Section 2. The C/C++-based design methodology is then introduced in Section 3. Our RTL C coding style is described in Section 4. In Section 5, we present our framework for equivalence checking between HDL and RTL C. Section 6 is our result section: the simulation performance of our RTL C model is presented along with examples of bugs found in our design using equivalence checking.

2. RELATED WORK

Several C/C++ coding styles have been used in the past two decades in the industry [17,24] as well as in the research community to describe hardware both for modeling and synthesis [7,20]. In general, the C/C++ language is both extended and restricted [15]. It is extended to support hardware data-types such as bit-vector, 3-state logic, etc. and, sometimes, to support parallelism using communicating processes and reactivity. On the other hand the C/C++ lan-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DAC 2002, June 10-14, 2002, New Orleans, Louisiana, USA.

Copyright 2002 ACM 1-58113-461-4/02/0006...$5.00.

guage is also restricted to prevent the usage of non-synthesizable constructs.

C/C++ programs are sequential whereas hardware is parallel by nature. Several techniques are usually used to represent hardware. The main difference is on the implementation of parallelism and reactivity. Both parallelism and reactivity may be implicit using a run-time simulation kernel as in SystemC [12,18], Cynlib [5], Spec C [9] or Esterel C [21]. The semantics may then be synchronous/cycle-based, asynchronous/event-driven, or a mix of both. This run-time behavior of the system is well suited for higher levels of abstractions, at the behavioral or system level or to model mixed hardware/software systems at the transaction level. However, it is not optimal at the RT level. There is an overhead for scheduling the different tasks, with the risk of running each task more than once at each clock cycle, which is not very efficient. This is addressed by static scheduling techniques. Different static coding styles have been used in the industry and in commercial simulation and synthesis environments. Such environments include RTL-C from Cynergy [4] and CycleC from C Level Design (acquired by Synopsys) [2]. They are a good fit for processor design.

A synthesizable subset of the language is also defined. Some research has been done on synthesizing floating point variables [19], pointers, and dynamically allocated memory [23]. Using these techniques, recursions could also be synthesized. These features mainly make sense at the behavioral or system level. In the case of C++, the synthesis of user-defined classes (i.e. classes that are not part of the hardware C++ library) and methods remains an open issue.

The synthesis path itself can be implemented in two different ways. One method consists of translating the RTL C description into an RTL HDL description [2,3,6,7]. This RTL description is then synthesized using existing synthesis tools. Synopsys CoCentric SystemC Compiler [11] may also synthesize hardware directly without the intermediate step of generating HDL code.

3. OVERALL METHODOLOGY

In this section, we present an overview of our C-based design and verification methodology. This methodology has been used in the development of a simultaneous multi-threaded (SMT) network service processor. The processor executes MIPS compatible code and is capable of forwarding 25 million packets per second – enough performance for 10 gigabit applications such as web switches, edge routers, traffic shapers and network storage servers. The design has about 7M gates. The core of the processor which accounts for 60%-70% of the overall design, is implemented in C at the RT level. The processor uses TSMC 0.15μ process and is targeted for 300MHz.

An overview of our C-based methodology is shown in Figure 1. Our design entry is a statically scheduled RTL C model. It is integrated with our C testbench environment for simulation. Both ANSI C fundamental types and bit-accurate types are supported. ANSI C fundamental types provide faster simulation whereas bit-accurate types model the hardware types more precisely.

The C code is translated into Verilog for implementation using a commercial tool. An internal lint tool has been developed to check the coding style before C to HDL translation.

Equivalence checking is performed between RTL C and Verilog HDL. We use our own translation tool to convert the RTL C model into a verification model in HDL. Then a commercial RTL to

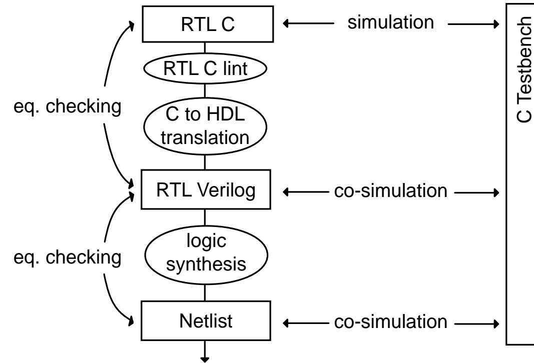

Figure 1: Verification Methodology

RTL equivalence checking tool [1,8,11,13] is used to verify the synthesized HDL against the verification HDL model. This step validates the HDL generated model against the C golden model. It validates the translation tool itself and also its interpretation of the C/C++ model: arithmetic operations, register mapping, and component instantiations.

Once the RTL verilog is generated and checked, our methodology follows the standard ASIC flow. Equivalence checking is used to formally validate the transformations performed on the design during synthesis (and scan insertion). The RTL verilog and the synthesized netlist are also verified using co-simulation with the C testbench environment.

4. RTL C MODELING

4.1 Coding style

The core of our processor is modeled in cycle-accurate C. This C model is statically scheduled. While static scheduling is more efficient, it requires more architecture work early on in the design process. In our static method, the design is partitioned in time into a set of slices. Figure 2 illustrates the use of slices in a simple example of a counter.

The slices represent the logic that is computed with specific ordering in the clock cycle. The design blocks contribute C functions called scheduling functions for execution in each of the slices. We found the static scheduling method to be a natural fit for efficiently modeling processor pipelines.

The design is described in C as a collection of logical blocks. The blocks are coded to define the imported, exported and internal signals and registers for the block, and to define the exported slice functions for the block. There is a methodology for reading signals between slices and between design blocks to ensure the design is proper and synthesizable.

In the RTL C simulation, the blocks are executed and communicate via the global scheduler which calls the slice functions of all the blocks in a simple static scheduling loop. Our methodology supports multiple instantiations of a given logical block.

In the verilog model for the complete design, the wiring of the instantiations of the logical blocks is generated automatically from a common file that describes the interconnect between the different logical blocks. A connectivity checker tool verifies that the connections described in the connectivity file correspond to the connections in the RTL C simulation model.

For synthesis, each logical block is individually translated (in isolation of the other blocks) into a Verilog module. This verilog module may contain hierarchy. Since logic synthesis doesn't perform scheduling of operations, module instantiation is directly inferred from the source code: C functions called inside slice scheduling functions become implemented as sub-modules.

4.2 Synthesizable subset

C and C++ are programming languages. As a result many of their features are of little use for hardware modeling, especially at the RT level.

In our methodology, unbounded loops and recursion are not supported. In order to optimize the synthesis results, pointers may only be used for parameters passed by reference in function calls. Operations on pointers, such as pointer arithmetic and type casting, are not permitted. Non-recursive data structures are allowed and embedded arrays and structures are supported. However, out-of-bound array accesses within structures are forbidden.

Structures are heavily used throughout our design. They are a convenient way of representing the set of input/output ports, registers and combinational signals for a given block. The example shown in Figure 2 uses several `structs`: `rCNT` represents the registers, `CNTimport` represents the imported signals and `CNTExport` the exported signals.

Since logic synthesis doesn't perform register allocation, registers are inferred directly from the source code. A naming convention is used for registers. They are synthesized by having a mandatory assignment from the `d` variable to the `q` variable conditional on the clock in the `RisingEdge` scheduling function (`CNTRisingEdge()` in Figure 2).

A naming convention is also used for ports and combinational logic signals. Ports may either be generated by analyzing the code [2] or by simply analyzing Import and Export signals.

4.3 Data types

Fundamental C/C++ data-types are limited to 8b, 16b, 32b and 64b lengths. The data-types used in hardware on the other hand may be of any length (e.g. 1b, 48b, etc.). In our methodology, two implementations of data types may be used. For the fast simulation speed, the hardware bit vector data types can be mapped to ANSI-C data types. For example 1-8 bits to `char`, 9-16 bits to `short`, etc. A slower but more accurate and elegant way of representing such datatypes is to use C++ classes (e.g. the SystemC `sc_uint<>` class) that implement the masking functions.

As shown in the file `common.h` in Figure 2, the SystemC library is used if `BIT_ACC` is defined. Otherwise, ANSI-C datatypes (`unsigned char`, `short`, `int` and `long long`) are used.

C++ objects can also be used to implement multi-state logic (e.g. 0, 1, X and Z states). However, as mentioned by Bening and Foster [14], the use of 'X' logic values is not advised at the RT level and should mainly be used at the gate level. Random initialization is a safer and more accurate way of representing the initial state on signals in hardware at the RT level.

Our implementation uses a modified version of SystemC that implements random initialization, in addition to zero initialization,

```
/* common.h */
#ifdef BIT_ACC    // use SystemC data types
#include <systemc.h>
typedef sc_uint<1> U1;
typedef sc_uint<9> U9;
typedef sc_uint<16> U16;
typedef sc_uint<32> U32;
typedef sc_uint<48> U48;
#else             // use ANSI C data types
unsigned char U1;
unsigned short U9;
unsigned short U16;
unsigned int U32;
unsigned long long U48
#endif

typedef struct { U9 d; U9 q; } R9;
```

```
/* counter.local.h */
typedef struct {
  U1 enableS0;
  U1 resetS0;
} CNTImport_s;

typedef struct {
  R9 counter;
} rCNT_s;
```

```
/* counter.export.h */
typedef struct {
  U8 resultS0;
  U1 overflowS0;
} CNTExport_s;

extern CNTExport_s CNTExport;
```

```
/* counter.c */
#include "common.h"
#include "counter.local.h"
#include "counter.export.h"
#include "io.export.h"

CNTImport_s CNTImport;
rCNT_s      rCNT;
CNTExport_s CNTExport;

void CNTRisingEdge() {
  if(system_clock) {
     rCNT.counter.q = rCNT.counter.d;
  }
}

/* slice 0 */
void CNTS0() {
  CNTExport.resultS0 =
             get_slice(0,7,rCNT.counter.d);
  CNTExport.overflowS0 =
             get_bit(8,rCNT.counter.d);
}

/* slice 1 */
void CNTImportS1() {
  CNTImport.enableS0 = IOExport.enableS0;
  CNTImport.resetS0 = IOExport.resetS0;
}

void CNTS1() {
 rCNT.counter.d = rCNT.counter.q; // default
  if(CNTImport.enableS0) {
     rCNT.counter.d = rCNT.counter.q + 1;
  }
  if(CNTImport.resetS0) {
     rCNT.counter.d = 0;
  }
}
```

Figure 2: Example of a counter module in RTL C code

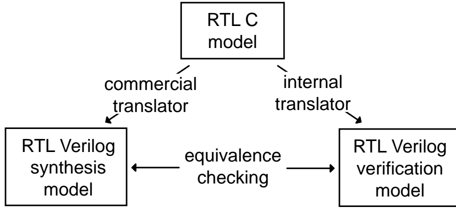

Figure 3: Formal equivalence checking between RTL C and Verilog

using a random seed. This is implemented in C/C++ using random assignments in constructor member functions. The seed can either be selected randomly for regressions or specified by the user to reproduce errors.

5. FORMAL EQUIVALENCE CHECKING

5.1 RTL C vs. Verilog equivalence checking

This section describes how we formally check that the generated verilog description matches the RTL C description. As we have seen in Section 3, current equivalence checkers can formally verify that two HDL models are equivalent. In this work, we do not try to replace existing equivalence checkers. Instead, we developed our own C to verilog translator tailored for our RTL C subset and coding style. Existing equivalence checking tools are then used to check that the verilog model used for synthesis is equivalent to our verification verilog model, as shown on Figure 3.

This translation tool differs from existing C to verilog translation tools not only because it only supports a given coding style, but also because the generated code is targeted for verification rather than synthesis. The emphasis is on matching the bit-accurate C/C++ semantics rather than trying to optimize synthesis results. The implementation of the tool is presented below. Some results and examples of bugs found in the design and in existing translation tool are then presented in Section 6.2.

5.2 C to verilog translation

Our RTL C to HDL translation was implemented using the SUIF compiler framework [10, 25]. The different steps are shown on Figure 4 and are detailed in the rest of this section.

After preprocessing, the SUIF front-end pass generates an intermediate SUIF representation of the code. The front end is slightly modified to keep track of `typedef` constructs and internal data-types for bit-accuracy. The resulting SUIF intermediate representation is then transformed in each pass in order to remove all C constructs that do not make sense in verilog. The next steps represent source-to-source C transformations. They are followed by a C to verilog translation step and by verilog code generation. Each of these steps are defined in the following sub-sections.

- **Function inlining and forward substitution**

The first transformation on the source code consists of inlining functions that are defined (functions that are not defined are assumed to be mapped to components). Pointers may be used to pass parameters by reference. Forward substitution[1] is used to propagate the pointers' values. After propagation, loads (...=`*p`) and stores (`*p`=...) are replaced by direct variable assignments. Forward substitution is used instead of constant propagation [22] because

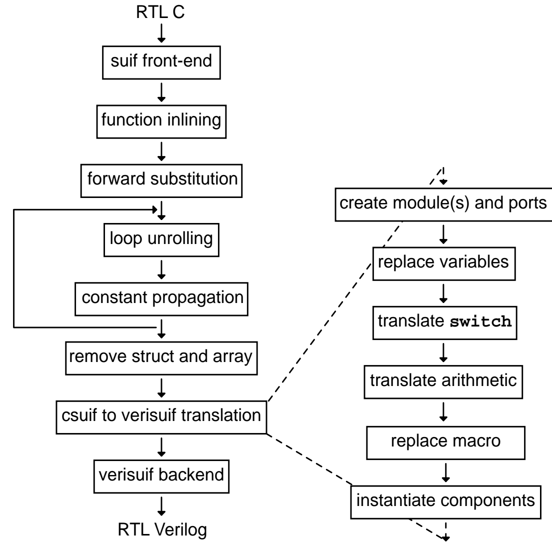

Figure 4: Implementation of xlc2v for the translation of RTL C code into verilog for equivalence checking

pointers to structure fields and array elements are defined using complex arithmetic operations (e.g. `p=&a[i].f`).

- **Loop unrolling and constant propagation/folding**

After function inlining, loop unrolling is performed and constant propagation [22] is used to propagate the value of the loop index (and other induction variables). In the case of nested loops where an inner loop depends on the indexes of the outer loops, multiple iterations are necessary.

- **Synthesis of structures and arrays**

At this stage all array accesses and structure accesses can be statically resolved. Structures are replaced by a set of variables that represent their different fields. The dot '`.`' in the `struct` is replaced by an underscore character '`_`'. Array variables are broken into a set of elements by appending the element index to the array variable name. For example, the field `rBLK.packet[2].header.byte[2]` corresponds to the variable `rBLK_packet2_header_byte2`. Name clashes are reported as errors.

The case of a structure variable passed by value or by reference in a function call is trickier to implement [2]. Instead of breaking the structure into a set of separate variables, a bit vector is created. Its size equals the sum of the widths of all the fields in the structure.

All of this is possible because pointer arithmetic and out-of-bound array accesses are not part of our synthesizable subset.

- **SUIF to verisuif translation**

This step takes a C intermediate representation and generates a verilog intermediate representation. The verilog IR and back-end used were developed by French et al. [16] and extended for our needs.

First the top-level verilog modules are created. In the case of a logic block, the output ports correspond to the Export signals. The input ports correspond to the Export signals assigned to Import signals. Another technique [2] is to map signals that are assigned but never read to output ports and signals that are read but never assigned to input ports. In the case of internal modules, parameters

1. Forward substitution is a pass that replaces a copy operation by a reevaluation of the expression. [22]

passed by value are mapped to input ports and parameters passed by reference (i.e. pointers) are mapped to output ports.

All variables are then uniquified and replaced by verilog `regs` with the correct bit-width. The next few passes modify the code itself. First the `switch` statements in C are translated to verilog `case` statements. Then, all arithmetic operations are replaced to match the SystemC bit-accurate semantics in verilog. All intermediate computations are performed on 64bit and their result is truncated at the final assignment (only `sc_uint<>` SystemC datatype is supported).

Several macros are used to represent verilog operations such as concatenation, bit selection and range selection. They are translated into their verilog equivalent. Functions that are not macros and not inlined are then mapped to components. Components are instantiated and temporary `regs` and wires are connected to their ports.

The resulting verisuif intermediate format is then translated into verilog using the verisuif back end.

6. RESULTS

In this section, we first present our RTL C simulation performance results to illustrate the advantage of using RTL C model in term of speed. Results are then presented for formal equivalence checking between RTL C and HDL. Several bugs found using equivalence checking are described.

6.1 Simulation results

The simulation performance results are shown in Table 1. The simulation example used here represents about one third of our chip, equivalent to a 2M gate design. These results are obtained on a Linux server with dual x86 Intel Xeon 730MHz processors. The same C testbench environment is used for all models. Co-simulation is used to validate the RTL Verilog model with the Synopsys VCS simulator [11] and the C testbench running in two separate processes. The Verilog model simulated is the synthesis model. We expect the SUIF-generated verification model would be slower.

Simulation Model	cycle per second	speed-up over HDL Verilog
Pseudo Cycle Accurate C	5864	150.0
RTL-C ANSI-C	1812	46.0
RTL-C Bit Accurate	127	3.2
Verilog RTL	39	1.0

Table 1: Simulation Performance

These results show that the RTL ANSI C model is about 46 times faster than the RTL verilog model and only 3 times slower than our pseudo cycle-accurate behavior model. Using C++ bit-accurate data types leads to a 14x slow down in terms of performance compare to ANSI C. The bit-accurate simulation could however be accelerated using simple compiler optimization techniques (e.g. to perform computations on 32b instead of 64b). The bit-accurate model remains however about 3x faster than verilog simulation. The Verilog results include the overhead of PLI and co-simulation using IPC. A faster co-simulation environment could be implemented using LWT and no PLI (e.g. using Synopsys DirectC [11]).

6.2 Equivalence checking analysis

Our implementation flow uses the tool presented in Section 5 for RTL C to HDL translation for generating a verification model. The C Level Design's System Compiler translation tool [2] is used for synthesis. Equivalence checking is then performed on the two RTL HDL codes using either Verplex Conformal LEC tool [13] or Synopsys Formality [11]. Results for the translation of some of the blocks of our chip are shown on Table 2. The number of lines for both the C and Verilog models are reported after removing comments and empty lines. The number of lines in Verilog RTL is from 2x to 7x more than the equivalent RTL C. This mainly comes from loop unrolling and also from matching the bit-accurate C semantics in Verilog. The translation time is measured on a Linux server with x86 Intel Xeon 730MHz processors. Even though the SUIF compiler framework was not optimized for performance, the translation time scales fairly well on our designs.

Block Name	C Lines	Verilog Lines	Translation time
C	2,597	5,034	1 min 43s
I	969	2,243	32s
L	1,258	7,737	50s
O	440	1,376	14s
P	22,409	150,845	32 min 11s
Q	6,199	35,583	6 min 19s
R	23,386	60,552	17 min 20s

Table 2: C to Verilog translation results for several logic blocks using internal tool xlc2v.

Several problems have been caught early on in our design as well as in the translation tool for synthesis. They are listed in Table 3. The types of bugs are sorted in two categories. The first category marked as *class D* represents the design and coding-style bugs that created incorrect behavior or made our translation tool fail. The second category marked as *class T* represents bugs found in C-Level System Compiler tool [2]. Overall, we found that System Compiler performed well. Using equivalence checking we were able to find the problems early in the project and easily work around them.

A detailed discussion of two of the bugs from this list follows.

Types of Bug	Class
Arithmetic mismatches between C and Verilog semantics leading to design bug	D
Bug in implementation of concat() macros	T
Variable passed by value to a function updated after the function is called	T
Out of bounds array accesses	D+T
Passing a structure field by reference through function call	T
Variable read before initialized	D
Parameter name mismatch between .h and .c (positional vs. name based disagreement during synthesis of module instances from function calls)	D
use of (x = 1) instead of (x == 1)	D
Variable assigned in multiple non-blocking statements	D
Use of m bit output from a function for a n-bit logic (m != n) during synthesis of function call	D
Incorrect syntax/usage of #ifdefs	D

Table 3: Types of Bugs Found

Example 1. *Arithmetic mismatches between C and Verilog are a common problem. With the options we have selected for synthesis, System Compiler generates a verilog code that "looks" very similar to the C code but that may not match its semantics. The typical error is something like the following operation:*

```
o1 = ((b4 + c4) > 0xF);
```

where `o1` *is a 1b variable and* `b4` *and* `c4` *are 4b variables. The carry is lost in the verilog generated for synthesis because the operation is performed on 4b instead of 32b in ANSI C (or 64b for SystemC).*

The correct Verilog code would look like:

```
o1 = (({1'b0,b4}+{1'b0,c4}) > 5'hF);
```

This problem also occurs with right shifts (>>) and other types of comparisons (e.g., <, >, ==, `!=`*). It is often found inside of the condition statement in* `if` *or* `(?:)` *constructs, which makes it very hard to debug during simulation.*

Example 2. *Bugs were also found in the System Compiler tool for parameters passed by reference in function calls for functions mapped to components. The following slice calls the function* `foo`*, mapped to a component.*

```
void BLKS1() {
    a.d = a.q;
    foo(a.d,&out);
    if(reset) a.d = 0;
}
```

The verilog code generated for synthesis is the following:

```
foo foo1(a_d, out);
always @(a_q or reset) begin
    a_d = a_q;
    if(reset) a_d = 0;
end
```

The problem in this case is that `foo(a.d,&out)` *reads the value of* `a.d` before *reset in the C code, whereas in the Verilog HDL code, it reads the value of* `a_d` after *reset. A correct Verilog code would have been:*

```
foo foo1(tmp_reg, tmp_wire);
always @(a_q or tmp_wire or reset)
begin
    a_d = a_q;
    tmp_reg = a_d;
    out = tmp_wire;
    if(reset) a_d = 0;
end
```

These types of bugs may be very hard to isolate using simulation. A workaround is to prevent the use of write-read-write sequences. This may sometimes be too restrictive and formal equivalence checking is the best way to make sure the code is correct.

7. CONCLUSION

The RTL-C design and verification methodology presented enabled the successful implementation and rapid validation of an extremely complex SMT network processor. The methodology provides a high speed RTL simulation model from a statically scheduled coding style and provides RTL-C to RTL-Verilog equivalence checking to further leverage the simulation performance advantage by reducing the amount of RTL-Verilog and gate simulation required during development.

ACKNOWLEDGMENT

This work was done at Clearwater Networks. The definition of this design methodology involved the work of several members of the design and verification teams. We would like to thank Forrest Brewer, Steve Hatala, Jeff Handong, Dick Hessel, Jeff Huynh, Ed Jacobs, Phil Lowe, Enric Mussoll, Mario Nemirovsky, James Reilley, Nandu Sampath, Soumya Seshadri, Pierre-Xavier Thomas, Dan Williams, Tom Yeh. We would also like to thank the engineers at C-Level Design.

REFERENCES

[1] Avant! corp, Design Verifyer, http://www.avanticorp.com/

[2] C Level Design, C2HDL, http://www.cleveldesign.com/

[3] CoWare, N2C, http://www.coware.com/

[4] Cynergy System Design, http://www.cynergysd.com

[5] Forte Design Systems, Cynlib, http://www.forteds.com/products/cynlib.html

[6] Frontier Design, A|rt Builder, http://www.frontierd.com/

[7] IMEC OCAPI http://www.imec.be/ocapi/

[8] Mentor Graphics, Formal Pro, http://www.mentorg.com

[9] SpecC Technology Open Consortium http://www.specc.gr.jp

[10] Suif compiler framework http://suif.stanford.edu/

[11] Synopsys tools, http://www.synopsys.com/

[12] SystemC, http://www.systemc.org/

[13] Verplex, http://www.verplex.com

[14] Lionel Bening and Harry Foster, "*Principles of Verifiable RTL Design: a functional coding style supporting verificaion processes in Verilog*," 2nd ed., Kluwer Academic Publishers, 2001

[15] Giovanni De Micheli, "*Hardware Synthesis from C/C++*," proc. Design, Automation and Test in Europe, pp. 382-383, Munich, 1999.

[16] Robert French, Monica Lam, Jeremy Levitt, and Kunle Olukotun, "*A General Method for Compiling Event-Driven Simulations*," proc.. Design Automation Conference, June 95.

[17] Dan Joyce, Robert Stets, Andreas Nowatzyk, "*C++ Design, Verification and Automatic Conversion to Synthesizable Verilog on a Large Processor*," Proc. HDLCON, Santa Clara, Feb. 01.

[18] Stan Liao, Steve Tjang, Rajesh Gupta, "*An efficient Implementastion of Reactivity for Modeling Hardware in the Scenic Design Environment*," proc. Design Automation Conference, pp.70-75, June 97.

[19] H. Keding, M. Willems, M. Coors, H. Meyr, "*FRIDGE: A Fixed-Point Design And Simulation Environment*," proc. Design Automation and Test in Europe, pp. 429-435, 1998.

[20] David Ku and Giovanni De Micheli, "*High-Level Synthesis of ASICs under Timing and Synchronization Constraints*", Kluwer Academic Publishers, Boston, MA 1992.

[21] Luciano Lavagno and Ellen Sentovich, "*ECL: A Specification Environment for System-Level Design*," proc. Design Automation Conference, New Orleans, pp. 511-516, June 99.

[22] Steven Muchnick, "*Advanced Compiler Design and Implementation*," Morgan Kaufman Publishers, San Francisco, California, 1997.

[23] Luc Séméria, Koichi Sato, Giovanni De Micheli, "*Synthesis of Hardware Models in C with Pointers and Complex Data Structures*," IEEE trans. on VLSI, pp. 743-756, vol. 9, n. 6, Dec. 01.

[24] Kazutoshi Wakabayashi and Takumi Okamoto, "*C-Based SoC Design Flow and EDA Tools: An ASIC and System Vendor Perspective*," IEEE trans. on CAD, vol. 19, n. 12, pp. 1507-1522, Dec. 00.

[25] Robert Wilson, Robert French, Christopher Wilson, Saman Amarasinghe, Jennifer Anderson, Steve Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary Hall, Monica Lam, and John Hennessy "*Suif: An Infrastructure for Research on Parallelizing and Optimizing Compilers*", ACM SIPLAN Notices 28(9), pp.67-70, Sept. 1994.