Journal of Zhejiang University SCIENCE
ISSN 1009-3095
http://www.zju.edu.cn/jzus
E-mail: jzus@zju.edu.cn

# A front-end automation tool supporting design, verification and reuse of SOC

YAN Xiao-lang (严晓浪)[1], YU Long-li (余龙理)[†1], WANG Jie-bing (王界兵)[†2]

([1]*Institute of VLSI Design, Zhejiang University, Hangzhou 310027, China*)

([2]*C-Sky Microsystems, Hangzhou 310032, China*)

E-mail: [†1]longli_yu@saianmicro.com; [†2]jiebing_wang@saianmicro.com

Received Mar. 18, 2003; revision accepted June 12, 2003

**Abstract:** This paper describes an in-house developed language tool called VPerl used in developing a 250 MHz 32-bit high-performance low power embedded CPU core. The authors showed that use of this tool can compress the Verilog code by more than a factor of 5, increase the efficiency of the front-end design, reduce the bug rate significantly. This tool can be used to enhance the reusability of an intellectual property model, and facilitate porting design for different platforms.

**Key words:** System-On-Chip, Verilog, HDL, Verification, Reuse
**doi:**10.1631/jzus.2004.1102 **Document code:** A **CLC number:** TN402

## INTRODUCTION

With ever increasing semiconductor chip complexity, HDL (hardware description language) is also evolving, as required by the evolving design and verification methodologies. The new HDL design methodology puts more emphasis on system level design, IP (intellectual property) models, design re-use and very deep submicron effects, etc. Such is the case for the emerging IEEE 1364-2001 Verilog-2001 standard (IEEE Std 1364, 2001; Cummings, 2001). Also, System-On-Chip (SOC) methodology requires system-level simulation, i.e., hardware-software co-design/verification in the early project stage. Mostly for this purpose, a group of design languages and extensions are proposed to raise the abstraction level for hardware verification. The list includes e, OpenVera, Superlog, System Verilog and a few other variants of C/C++.

Here we introduce an in-house developed HDL tool called VPerl (Verilog-Perl), which has been used successfully in implementing a 250 MHz 32-bit high-performance, low-power embedded microprocessor (CK520) in Hangzhou C-Sky Microsystems and VLSI Institute of Zhejiang University. Perl, as a programming language, not only has excellent text processing capability, but also offers database interconnectivity. A few elegant syntaxes offered by VPerl, as will be introduced later in this paper, will raise the abstraction level, reduce the code size, decrease bug rate and offer reusability, extensibility of the IP blocks. VPerl will be a great complement to not only Verilog-1995, but also Verilog-2001.

The following sections of the paper will discuss VPerl processing flow based on Verilog-1995 (IEEE Std 1364, 1995), VPerl for design, VPerl for verification and VPerl for reuse, respectively. In these discussions, the actual implementation of CK520 is used as the example.

## VPERL PROCESSING FLOW

VPerl transforms a Verilog template file called vp file into a Verilog file. A vp file is designed to let designers focus on the logic expressions, and the other parts such as port declaration, sensitive list declaration and register or wire data type declaration are automatically generated by the basic functions of Vperl (Davis and Mudge, 1995).

The transforming process is comprised of three stages: preprocessing, template analysis and HDL generation, as shown in Fig.1. Between these stages, useful data is exchanged with text file or through accessing database. Again, VPerl takes advantage of Perl's strong text processing capacity and rich internet resources.

At the first stage, VPerl template file is preprocessed. Preprocessing removes all the comments, processes Verilog directive keywords (for instance, "ifdef"), and formats the codes so that they can be readily handled at the next stage. Lexical check is also done at this stage. This template file should follow Verilog or VPerl syntax. At the second stage, Verilog lexis in the template file is analyzed, and necessary information is extracted and stored into an embedded database system–Berkeley Database. This information database contains the sources and sinks of an expression, the sensitivity list of a combinational block, etc. At the last stage, VPerl accesses the database and generates Verilog file. All VPerl syntax is substituted by associated Verilog code fragments.

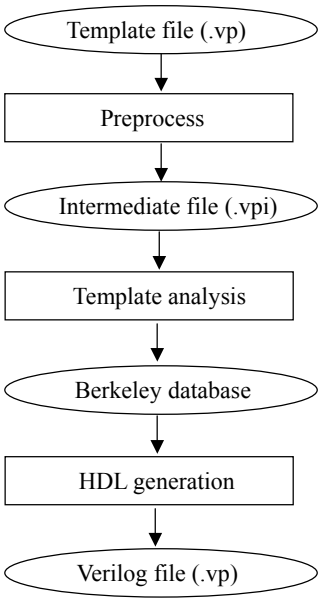


**Fig.1 VPerl processing flow diagram**

## VPERL FOR DESIGN

The use of VPerl consists of two steps. The first step is to find dependency for each Verilog module, for instance, the fact of A depending on B means B is instantiated in A. The outcome of this step is a makefile. The second step is to invoke VPerl processing engine that will expand a Verilog template file into a Verilog file.

Some basic features of VPerl are listed below.

i) Vperl handles the tedious parts of constructing a Verilog module, such as module declaration, port declaration, data type (register or wire) statements, and sensitive list statement in combinational logic, etc. These features not only reduce coding load, but also reduce bug rate. One example is that VPerl automatically analyzes the port definition through its source-and-sink database, and the designer can easily identify improper coding by looking at the port definition generated by VPerl. A sample template file is shown in Fig.2.

In Fig.2, all clauses starting with ampersand

```
// VPerl template example1
&module_beg;  // mark begin of module
&declare_ports;  // auto-declare ports
&declare_regs;   // auto-declare register signals
&declare_wires;  // auto-declare wire signals

assign data_sel[1:0] = src_sel ? src0[1:0] : src1[1:0];

&comb_begin;   // auto-declare sensitive list
   data_out[7:0] = 8'b0;
   case(data_sel[1:0])
      2'b00: data_out[7:0] = data_in[7:0];
      2'b01: data_out[7:0] = data_in[15:8];
      2'b10: data_out[7:0] = data_in[23:16];
      2'b11: data_out[7:0] = data_in[31:24];
   endcase
&comb_end;
&module_end;   // mark end of module
// end of example1
```

**Fig.2 A sample VPerl template file**

are VPerl syntax. VPerl will expand these clauses and generate corresponding Verilog code fragment. Through these clauses, designers will only concentrate on logic expression coding.

ii) VPerl has a powerful feature in instantiating sub-modules. VPerl can search the sub-modules automatically through external environment setting, instantiate them, and connect ports intelligently in the generated Verilog file (Bening *et al.*, 2001). An example is shown in Fig.3.

iii) VPerl provides an API interface to call other stand-alone programs. This greatly increases reusability of a Verilog Design–making HDL design more object-oriented. More details are covered in the section VPERL FOR DESIGN REUSE. It is worth mentioning that VPerl has integrated eperl, embedded Perl, which has been found rather useful. Fig.4 shows a piece of code that is included in our pad module.

VPerl will then instantiate the pad_cell 32 times, significantly reduce the duplicative work and avoid cut-and-paste errors.

Code compression ratio has been studied in CK520 project. We have seen 2 to 12 fold compression among different Verilog modules. More than 5 fold compression ratio can be achieved.

```
// VPerl template example2
// top-level module of CPU core

&module_begin;
&declare_ports;    // auto-declare ports
&declare_regs;     // auto-declare register signals
&declare_wires;    // auto-declare wire signals

&instance("ifu");    // instantiate instruction
                        fetch unit
&instance("decode"); // instantiate instruction
                        decode unit
&instance("execute"); // instantiate instruction
                        exec unit
&instance("data");      // instantiate data memory
                        access unit
&instance("wb");    // instantiate write back unit

&module_end;
```

**Fig.3 An example showing instantiation feature of VPerl**

```
...
<:
for($i=0; $i<32; $i++) {
  print "pad_cell i_gsb_$i (   \n";
  print "  .PAD (i_cpu_gsb[$i]),\n";
  print "  .C (pad_biu_gsb[$i]) \n";
  print ");                          \n";
}
:>
...
```

**Fig.4 Eperl support from Vperl**

## VPERL FOR VERIFICATION

VPerl provides a syntax:

&assert_error("… verilog expression…");

This will generate assertion-based error checking if enclosed Verilog expression is evaluated to be true (Bening and Foster, 2001), and directly help verification to find bugs during simulation run-time.

Another rather useful syntax is:

&assert_one_hot_error("…sel[x:o]…");

This is very useful for a high-performance design where one-hot state machine is constantly used.

Although only two syntaxes are listed here, VPerl leaves the door open for integrating more error checks.

## VPERL FOR DESIGN REUSE

Reuse is a prominent issue in the design of SOC (Keating and Bricaud, 2002). One important feature in VPerl is that it supports an API interface to call other stand-alone programs. By using this feature, objects can be created in the design with its characteristics and ports configured as needed, which is similar to creating an object in C++. We have employed a configurable synchronous FIFO generator script in CK520 project. A designer can call this script directly in the VPerl template file, and specify the depth, width, and flags they wish to have for the FIFO. Thus, the FIFO object is essentially reused throughout CK520 design.

The authors have overcome another challenge

during the design of CK520 by using VPerl. The goal is to port CK520 to a Xilinx Vertex-II FPGA to increase our confidence before the final tape-out. The problem is that in order to port to FPGA, several issues must be resolved: on-chip customer PLL to become Xilinx DCM, on-chip memory-compiled SRAM to become Xilinx Block SelectRAM, etc. Meanwhile, we constantly need to pull out internal wires to the FPGA external pins for digital analyzer observation. All these may require module port-interface changes, which ASIC design/verification team does not want to see. It is a common practice to separate ASIC and FPGA into two files, so one has to maintain the coherence between them. VPerl solves this problem by preprocessing "ifdef" construct and automatically generate port list, so that only one file needs to be maintained.

## CONCLUSION

As SOC design and verification becomes more complex, a good tool methodology becomes essential in increasing design efficiency, reusability and reliability. The authors believe that a tool like VPerl can greatly enhance the above areas in the SOC design and verification.

## ACKNOWLEDGEMENT

One of the authors, Dr. Jiebing Wang, sincerely thanks Mr. Dane Mrazek for his enlightening contributions to this work. All the authors thank the entire CK520 project team from C-Sky Microsystems and Zhejiang University for their trust in us and debugging of this tool.

## References

Bening, L., Hornung, B., Pflederer, R., 2001. Hardware Description Language-Embedded Regular Expression Support for Module Iteration and Interconnection. Proc. International Hardware Description Language Conference.

Bening, L., Foster, H., 2001. Principles of Verifiable RTL Design, Second Edition. Kluwer Academics, USA, p. 31-37.

Cummings, C.E., 2001. Verilog-2001 Behavioral and Synthesis Enhancements. HDLCON2001, Rev1.3.

Davis, B., Mudge, T., 1995. A Verilog Preprocessor for Representing Datapath Components. Proceedings of the 4th International Verilog Conference, p.90-98.

IEEE Std 1364, 1995. IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language.

IEEE Std 1364, 2001. IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language.

Keating, M., Bricaud, P., 2002. Reuse Methodology Manual, 3rd edition. Kluwer Academics.