The *CSYN* Verilog Compiler and Other Tools.

David Greaves

University of Cambridge, Computer Laboratory, Cambridge, UK. djg@cl.cam.ac.uk

Abstract. The CSYN Verilog compiler was written by Dr Greaves in early 1994 as a vehicle for research in logic synthesis algorithms and to support experimental extensions to the Verilog language to test high-level specification techniques. A basic version of CSYN is in use at a number of local companies for industrial FPGA design. This paper describes CSYN and its use with Xilinx devices for teaching. To extend this work, we are defining formal semantics for Verilog, both for simulation and compilation into hardware. This paper reports the performance of CSIM, an X-windows Verilog simulator based on the formal simulation semantics and expresses the desire for a general purpose semantics for Verilog, which can help prove the equivalance of different implementations of a module.

1 Background

The Systems Research Group of the University of Cambridge Computer Laboratory has for ten years or more owned and used a set of CAD tools based around the 'Cambridge HDL' heirarchic net list format [1]. The tools were first written to support internal research projects, but they have also been commercialised from time to time and used by a dozen or so local companies and they have been used extensively in teaching.

In recent years, limited logic synthesis capabilties were added to the tools using enhanced ABEL as the input specification language. ABEL's infix operators for addition and complex combinatorial expressions so became available. In addition, a local language FDL (functional definition language) was added to allow behavioural specification of standard, non-synthesised parts, such as RAMs, FIFOs or clock modules and test wrappers. Hence the human input to the CAD system was split over three source languages which necessarily resided in different source files, owing to different parsers and front-end processing.

A few years ago, the author became interested in the Verilog language and so the process of moving the tools to Verilog began. Like VHDL, Verilog is able to integrate these three forms of specification using a common syntactic structure, and indeed the forms can be intermixed at the fine grain of individual declarative statements within a Verilog 'module'. The module is the heirachic building block, and can be anything form a gate to a microprocessor or complete system.

Today the tools run on Unix or other Posix machines (such as Linux) using an X-windows interface for the simulators and command line or makefile driven interfaces for the remainder. About half of the designs generated with the tools are targeted at Xilinx devices and the remainder include standard cell,

PCB board models, MACH devices or abstract designs with their own research content.

The following programs have been written:

- CSYN - Verilog compiler.
- CSIM - Verilog simulator.
- XSIM - Cambridge HDL/FDL simulator.
- CVAUX - Verilog flattener.
- CVXNF - Verilog to Xilinx net list format convertor.
- LCATOV - Xilinx logic cell array to Verilog convertor.
- CPAL - Verilog to JDEC pal compiler.
- TRANS - Verilog multi-level logic minimiser.

2 CSYN **Verilog Compiler**

CSYN is described in the CSYN user manual [2]. The input to CSYN is a set of Verilog files, including the design proper and standard libraries. The libraries contain reusable building blocks, such as a Huffman or Manchester coder, which can be compiled for any target technology, and target technology-specific definitions such as input/output pads or special buffers which drive things such as the Xilinx global clock nets.

The output from CSYN is a heirarchic Verilog net list known as a `vnl` file where each module in the output corresponds to a module in the input source (CSYN can also generate Cambridge HDL). A `vnl` file is the subset of Verilog language obtained by deleting the RTL and behavioural declarations: `assign, initial` and `always`, and disallowing the entity declarations `integer, reg, time, event` etc.. This leaves only the `wire` and `tri` data types and the only significant construct is structural instantiation of a submodule.

The heirarchy in a `vnl` file can be flattened when necessary using an extra program called 'CVAUX'. A related program, CVXNF will flatten and generate Xilinx net list format (xnf) for input into the Xilinx tools. In this mode, it takes an additional command line argument to specify a technology library, such as 'xi4000', which contains appropriate macros which are expanded to generate the xnf.

Logic synthesised by CSYN is not mapped onto a specific target technology library. Instead, CSYN converts all RTL and behavioural constructs into a fixed set of gates, which are AND2, OR2, INV, DFF, TLATCH, MUX2. It is up to the software which processes the target `vnl` file to understand these gate types and fit the logic to the target device. However, the instantiated modules output from CSYN do not only include these gates, since the source Verilog may contain structural instantiations of library or other leaf modules such as the pads etc. mentioned earlier. Attempts to actually synthesise these leaf modules by CSYN is prevented by flagging their definition with a small extension to the Verilog language syntax: the keyword `primitive` may be introduced before the word `module` in each such definition, or alternatively the line `primitive everything` may be added to a file to mark as primitive all textually subsequent modules.

For primitive modules, CSYN ignores the module's body, if present, and does not flag 'output not driven' warning messages when the body is absent. However CSYN does cross-check the `input, output` and `inout` port direction statements for compatibility at each instantiation of the primitive module.

Using CSYN, the designer has control of the logic that will be synthesised through the way he uses continuous assignments and parenthesis. If part of the circuit is critical, it must be essentially hand coded in the source file using the more simple constructs of Verilog which have predictable synthesis paths, rather than using esoteric `for, case` or `while` constructs. Such control might be termed `in-band` when compared with other logic synthesisers which accept complex `out-of-band` annotations for each input and output, expressing a desire for low load or late arrival etc.. Advanced minimisation of logic functions using techniques based on *Expreso* are not considered appropriate within CSYN since the best optimisations are target technology specific and best implemented in subsequent vendor fitter tools.

Infact, CSYN does have a small set of command line parameters which modify slightly the way it synthesises structures such as adders, but these are generally left unused. Their use is to trade speed of operation against gate count.

2.1 CSYN operation

CSYN first parses the input Verilog files to build a lisp parse tree. It then selects the subset of sourced modules that will be present in the output (i.e. are being used) using a recursive tree walk from a command line specified root name, which must be one of the sourced modules.

It then uses simple algorithms to convert the supported behavioural constructs into continuous and behavioural assignments, generating for them a parse tree structure similar to that which they would have if they had been specified in this form to start with. D-type flip-flops or broadside registers are generated and inserted into the structure as though they had been structurally instantiated in the source file.

An algorithm which expands all busses into individual signal nets is then applied, and at the same time this algorithm rewrites the tree to convert all of the 'higher-order' logic functions, such as addition, bus comparison or unary reduction, into a small set of operators: namely AND, OR, XOR, INVERT and MUX2.

An algebraic simplifier then runs over the tree, reducing simple tautologies and applying other Boolean identities, but without applying cubic division or other more complicated minimisation.

The final stage is a recursive gate generating algorithm which walks over the tree replacing each node with a gate instantiation and allocating a signal identifier for its output. Before generating each gate, the gate generator checks whether it has already produced a gate of the same type and input connection pattern, and if so, uses the output from that instead. To make this associative search cheap, the gates are given a normalised input ordering and then hashed to provide a simple array index.

After this stage, the circuit is composed entirely of structural instantiations of modules and leaf gates. A port checker cross-checks the input/output/inout port direction across all submodule instantiations and checks that each signal has exactly one output driving it, unless it is of type `tri`.

2.2 CSYN **Synthesisable Constructs**

No contemporary Verilog compiler can convert every behaviourally expressable Verilog construct into gates. At the RTL level, CSYN supports the subset of the Verilog combinatorial operators shown in table 1 together with the unary reduction operators, bus concatenation and dynamic subscripting of busses.

Symbol	Function	Resultant width
¬	monadic negate	as input width
−	monadic complement (*)	as input width
!	mondaic logic not (*)	unit
*	unsigned binary multiply (*)	sum of arg widths
/	unsigned binary division (*)	difference of arg widths
%	unsigned binary modulus (*)	width of rhs arg
+	unsigned binary addition	input width plus one
−	unsigned binary subtraction	input width plus one
>>	right shift operator	input - shift amount
<<	left shift operator	input + shift amount
==	net/bus comparison	unit
! =	inverted net/bus compare operator	unit
<	bus compare operator	unit
>	bus compare operator	unit
>=	bus compare operator	unit
<=	bus compare operator	unit
&	diadic bitwise and	minimum of both inputs
↑	diadic bitwise xor	maximum of both inputs
↑ ¬	diadic bitwise xnor (*)	maximum of both inputs
|	diadic bitwise or	maximum of both inputs
&&	diadic logical and	unit
||	diadic locical or	unit
? :	conditional expression	maximum of data inputs

Table 1. Verilog Operators in order of Binding Power. Asterisked operators are not supported in current release of CSYN.

In Verilog, behavioural statements must be included within an `initial` declaration or an `always` declaration. CSYN (release cv2) ignores the contents of `initial` statements, but they may be present and are vital when simulating the source file with a simulator (to generate resets and so on).

CSYN compiles only the following form of the Verilog `always` construct. It has the syntax

```
always @( <sensitivity-list> ) <behavioural-statement>
```

This statement causes execution of the behavioural statement each time an event in the sensitivity list occurs. The behavioural-statement is typically a `begin-end` block containing other statements. The only sensitivity lists supported are the `posedge` or `negedge` of a single clock net or the full *support* list of the following statement (block).[1]

[1] CSYN will next be extended to support asynchronous resets to its behavioural regis-

CSYN compiles behavioural constructs using a function $CC_{(\Sigma,\Delta)}$ which takes a section of source code whose extent is free from event control statements (such as `posedge`), a pair (Σ, Δ) consisting of a substitution list Σ and an update list Δ and returns a new such pair. Both the substitution list and the update list are mappings of each variable (`integer, wire` or `reg` etc.) that appears in the block to an expression. The substitution list gives the mappings of each variable to expressions at the current point in the program and the update list keeps track of variables which are to be finally updated on the next clock edge, but which are not necessarily changed at the current textual execution point. Both lists are of pairs of the form (v, e) where v is a variable and e is an expression mapped to it.

At entry to a source block (i.e. after any previous event control) the initial Σ passed to $CC_{(\Sigma,\Delta)}$ is set to map each variable directly to the register, input pin or other source that drives it. Δ is empty. $CC_{(\Sigma,\Delta)}$ will handle `begin-end` sequencing, `case, if-then-else, repeat` and both blocking and non-blocking assignments as follows:

- $CC_{(\Sigma,\Delta)}[\![v = x]\!] =$ (replace v with $E_\Sigma[\![x]\!]$ in Σ, Δ)
- $CC_{(\Sigma,\Delta)}[\![v <= e]\!] = (\Sigma, \text{append}((v, E_\Sigma[\![x]\!]), \Delta))$ where the routine append flags an error to the user if the resulting list has two entries for a given variable.
- $CC_{(\Sigma,\Delta)}[\![c_1 \ ; \ c_2]\!] = CC_{(S)}[\![c_2]\!]$ where $S = CC_{(\Sigma,\Delta)}[\![c_1]\!]$
- $CC_{(\Sigma,\Delta)}[\![\texttt{if } (e) \texttt{ then } c_1; \texttt{ else } c_2]\!] = (\Sigma', \Delta')$ where

$$
\begin{aligned}
j &= E_\Sigma[\![e]\!] \\
\Sigma' &= \{(v, x) \mid x = (j)?E_{\Sigma 1}[\![v]\!] : E_{\Sigma 2}[\![v]\!]\} \\
\Delta' &= \{(v, x) \mid x = (j)?E_{\Delta 1}[\![v]\!] : E_{\Delta 2}[\![v]\!]\} \\
(\Sigma_1, \Delta_1) &= CC_{(\Sigma,\Delta)}[\![c_1]\!] \\
(\Sigma_2, \Delta_2) &= CC_{(\Sigma,\Delta)}[\![c_2]\!] \text{ or } (\Sigma, \Delta) \text{ if the else clause is missing.}
\end{aligned}
$$

  The questionmark-colon is the usual conditional expression construct (found in C and Verilog).
- Verilog's `case` statement is handled by pre-converting to a series of `if`s in the obvious manner.
- Verilog's `repeat` statement is handled by pre-expanding the source parse tree the specified number of times (which must be a compile-time constant).

The function $E_\Sigma[\![]\!]$ is a function which rewrites an expression where the variables are replaced with their values in the current substitution set (which are potentially complex expressions containing only input variables).

The resulting pair of lists returned by $CC_{(\Sigma,\Delta)}$ are converted into an appropriate RTL assignments of new values for the variables. The pending update

---

ters. Currently flip-flops which require this must be structurally instantiated. CSYN does not yet generate transparent latches when a signal is missing from the support list, but this is no great loss, since most users of other Verilog systems run with this option disabled.

list will contain at most one entry for each v (whose type must be `reg`) and an error is flagged if it contains an entry for any variable also in the substitution list (otherwise the signal would be driven twice).

Warnings are issued if any variable is assigned more than once, for instance via both a blocking and non-blocking assignment in the same section of code. Assignments to the same variable from separate `always` blocks results in the output net being driven by more than one gate or flip-flop, which is spotted and flagged in the final net list cross-checking phase of CSYN.

2.3 An example of CSYN input and output

Here is an input file:

```
module ADDER(clock,in1,in2,out);
  parameter size = 4;
  input clock;
  input [size-1:0] in1, in2;
  output [size:0] out;
  reg [size:0] out;
  always @(posedge clock) out <= in1 + in2;
endmodule
```

Which can be compiled to give the following output

```
// CBG CSYN Verilog hdl system.  Release 2 Beta 5. (May 95)

module ADDER(clock, in1, in2, out);
  wire u10057, u10056, u10055, u10054, u10053, u10052, u10051,
       u10050, u10049, u10048, u10047, u10046, u10045, u10044,
       u10043, u10042, u10041, u10040, u10039, u10038, u10037,
       u10036, u10035;
  output [4:0] out;
  input [3:0] in2;
  input [3:0] in1;
  input clock;
  AND2  u10055(u10055, in1[2], in2[2]);
  AND2  u10056(u10056, u10047, u10048);
  OR2  u10057(u10057, u10055, u10056);
  BUF  u10035(u10035, u10057);
  XOR2  u10053(u10053, in1[0], in2[0]);
  DFF  u10054(u10054, u10053, clock, 1, 0, 0);
  BUF  u10058(out[0], u10054);
  XOR2  u10051(u10051, u10044, u10045);
  DFF  u10052(u10052, u10051, clock, 1, 0, 0);
  BUF  u10059(out[1], u10052);
  AND2  u10043(u10043, in1[1], in2[1]);
  AND2  u10044(u10044, in1[0], in2[0]);
  XOR2  u10045(u10045, in1[1], in2[1]);
  AND2  u10046(u10046, u10044, u10045);
  OR2  u10047(u10047, u10043, u10046);
  XOR2  u10048(u10048, in1[2], in2[2]);
  XOR2  u10049(u10049, u10047, u10048);
  DFF  u10050(u10050, u10049, clock, 1, 0, 0);
  BUF  u10060(out[2], u10050);
  XOR2  u10041(u10041, u10035, u10037);
  DFF  u10042(u10042, u10041, clock, 1, 0, 0);
  BUF  u10061(out[3], u10042);
  AND2  u10036(u10036, in1[3], in2[3]);
  XOR2  u10037(u10037, in1[3], in2[3]);
```

```
  AND2  u10038(u10038, u10035, u10037);
  OR2  u10039(u10039, u10036, u10038);
  DFF  u10040(u10040, u10039, clock, 1, 0, 0);
  BUF  u10062(out[4], u10040);
endmodule
```

2.4 Two examples of non-synthesisable modules

CSYN cannot synthesise the following useful phase-frequency comparator because the registered signals are updated in an edge sensitive way by more than one 'clock'. A good implementation of this circuit uses just 12 NAND gates, but the algorithm to generate this type of circuit is a research topic.

```
module PHASEFREQ(ref, loc, faster, slower);
  output faster;  // High when local oscillator is slower than ref
  output slower;  // High when local oscillator us too fast.
  input ref;   // Reference oscillator
  input loc;   // Local oscillator
  reg faster, slower;
  wire idle = ~(faster | slower);
  always @(posedge loc)
        begin
        if (idle) slower <= 1;
        faster <= 0;
        end
  always @(posedge ref)
        begin
        if (idle) faster <= 1;
        slower <= 0;
        end
endmodule
```

In addition, CSYN cannot synthesise the following pattern generator because there is an implied thread of control. The algorithm which would generate the flip-flops required for the program counter has not been written, and is perhaps difficult.

```
  reg din;
  initial din = 0;
  always
        begin
        @(posedge clk) din = 1;
        @(posedge clk) din = 1;
        @(posedge clk) din = 0;
        end
```

2.5 Transduction Minimisation

Philip Abbey of the Computer Laboratory has implemented the 'Transduction' technique for multi-level logic minimisation [4]. The input and output to his program are `vnl` files of equivalent functionality but where the output uses fewer gates (or is optimised under another metric). The transduction technique is applied heuristically to the combinatorial logic components of a circuit, leaving the flip-flop structure intact, and operates using the concept of 'permissible functions'. The permissible function at the output of a logic gate inside a multi-level logic circuit is simply a truth table indexed by the inputs to the combinatorial network and contains don't cares (X's). Heuristic modifications to the network

are applied to try to increase the number of don't cares, with the effect of increasing the probability that a given gate's output can either be covered by one of its inputs or by the output of another gate in the circuit. In either case, the gate has become unnecessary and so its output is replaced with a wire to the appropriate point.

Using the Transduction method seems to reduce the CLB count for a Xilinx target by about 10 percent. This improvement is on top of the optimisation already provided by the gate-sharing algorithm described in section 2.1, which is also about 10 percent. This shows that the tools supplied with the FPGA's can sometimes be enhanced using such techniques. I will present some figures at the workshop.

Table 2 shows parameters for a set of five Xilinx designs, each compiled for a Xilinx 3000 FPGA.

Design	Lines of Verilog	VNL lines	Combinatorial gates	D-types	CLBs
AAL-10	179	2459	1858	32	80
Nasty-J	234	1850	962	70	188
Jaglink	1092	1478	976	126	124
Xcoder	838	1579	963	142	136
Cdxlx	389	1376	563	71	62

Table 2. Parameters for a few Xilinx designs.

3 ECAD Practical Classes

The CSYN compiler has been used for teaching Verilog as part of a second year course at the Computer Laboratory. The students had available the Xilinx back end tools via a batch email server. They experienced design turn around times of about 3 hours on this batch queue, and so it was rather like submitting a design for foundry fabrication. The Xilinx tools returned to them the `lca` (logic cell array) file which they could load into our Xilinx teaching cards [3]. These cards have an FPGA and switches, LEDs and other peripherals. A local program, `lcatov` will convert a `lca` file into a Verilog module containing detailed post-layout timing information for each signal. This enabled the students to perform annotated post layout simulation to see how much slower their designs will run.

The largest designs performed by the students were lift controllers which required four Xc3064 devices. A typical design was a reaction timer or other game, using 150 or so CLBs (configurable logic blocks). As a final year project, one student implemented the ARM microprocessor.

4 The CSIM simulator

To date, the design flow mostly used with CSYN is to compile everything into Cambridge HDL and then perform simulations using XSIM [5]. The XSIM simulator consists of 33000 lines of *C* code and is a robust and easy to use tool. However, recently, Mike Gordon of the Computer Laboratory conducted a set of experiments on three commercial Verilog simulators (Verilog-XL from Cadence, ViperFree from InterHDL and Veriwell from Wellspring Solutions) in order to deduce a formal semantics for the execution model of Verilog [6]. The IEEE draft standard was also helpful [7].

The author took these semantics and implemented the CSIM simulator, based on XSIM. The performance of CSIM was compared against some other simulators for the test program in figure 1 and the resulting execution times on a SPARCstation 10 Model 30 are given in table 3. CSIM is clearly competitive in execution speed, but it is also very easy to use with its built-in interactive GUI.

Veriwell and Viperfree do not support the `vectored` Verilog range expansion mode which causes the simulator to model a bus as a single number, making the simulation run faster, but with restrictions on assignments to just parts of busses. However it is possible with all of the simulators to change the definition of `ctr` to be an integer. This reduced the simulation time by about 10 percent for Veriwell and Viperfree.

```
module CLK1MHz(o);
  output o;
  reg o;
  initial begin o = 0; forever o = #100 ~o; end
endmodule


module SIMSYS();
  wire x;
  reg [20:0] ctr;
  initial ctr = 0;
  CLK1MHz clk(x);
  always @(posedge x) ctr <= ctr + 1;
  initial #10_000_000 $finish;
endmodule
```

Fig. 1. Example program used to compare the simulators.

5 Future Directions

Currently we are using different rules for compiling and interpreting Verilog. These rule sets cover different subsets of the (richish) Verilog language. However, we observe that once the semantics of a language are well formulated, it becomes a fairly simple matter to implement the associated compilers and interpreters,

Simulator	Behavioural	Gate Level
Verilog XL	9	71
Veriwell	8	301
Viperfree	26	290
Xsim	n/a	89
CSIM	7	180

Table 3. Execution time in seconds of the test code before and after compilation on five simulators.

and so as semantics develop to cover a greater subset of the language, we can make our tools more powerful.

We have started the Verilog Formal Equivalence Project [8], funded by EPSRC to automate equivalence checking between different versions of a module (in VHDL these would be known as alternative *architectures*). We are especially interested in checking the equivalence of the behavioural and gate-level versions which are perhaps the input and output respectively of a synthesiser, or perhaps have come from different source files with hand recoding or isolated development. Our approach will be to define semantics for the language which can, as far as possible, be used both for compilation and simulation.

Aknowledgements are due to Olivetti Research Ltd, John Porter, Andy Harter, David Milway, Mark Hayter, Ian Pratt. Xilinx © is a trademark of the Xilinx Corporation. Verilog © is a trademark of Cadence Design Systems.

References

1. Newson, A., Milway D.: *Cambridge HDL and FDL Reference Manual* http://www.cl.cam.ac.uk/users/djg/ecadteach
2. Greaves, D.J.,: *The* CSYN *Verilog Compiler and other tools - Professional Reference Manual.* http://www.cl.cam.ac.uk/users/djg
3. Temple, S.,: *The Xilinx Teaching Board* http://www.cl.cam.ac.uk/users/djg/ecadteach
4. Muroga, S., Kambayashi, Y., Lai, H.C., Culliney, J.N.,: *The Transduction Method - Design of Logic Networks Based on Permissible Functions.* IEEE Transactions on Computers, Vol 38 No. 10 October 1989.
5. Newson, A., Milway D.: *The ORL XSIM Simulator Manual* On http://www.cl.cam.ac.uk/users/djg/ecadteach
6. Gordon, M.J.C.,: *The Semantic Challenge of Verilog HDL.* Tenth Annual IEEE Symposium on Logic in Computer Science (LICS'95), June 26-29, 1995, San Diego, California. On http://www.cl.cam.ac.uk/users/mjcg/Verilog.
7. IEEE 1364. *Section 5 - Scheduling Semantics. Draft Standard Verilog HDL.* Draft standards document. On http://www.cl.cam.ac.uk/users/mjcg/Verilog.

8. Greaves, D.J., Gordon, M.J.C,: *Checking Equivalence Between Synthesised Logic and Non-Synthesisable Behavioural Prototypes.* A three year EPSEC research project. On http://www.cl.cam.ac.uk/users/mjcg/Verilog.

This article was processed using the LaTeX macro package with LLNCS style