"A Verilog to C Compiler."
RSP 2000, Paris, June 2000.

DJ Greaves
University of Cambridge*

Abstract

This paper describes a compiler which converts from Verilog to C. The output is then compiled to machine native code and tends to execute faster than native mode Verilog simulation because the compiler preserves only the synthesis semantics, not the simulation semantics, of Verilog and also performs logic minimisation. Busses of up to 32 or 64 bits can be modelled as C integers whereas larger busses are automatically split. We describe the motivation, method and quality of the results.

1 Introduction

In this paper we describe the design of a Verilog to C converter program and describe experience with it. The program, called VTOC, takes a synthesisable Verilog module and generates a semantically equivalent ANSI C code section. Parts of the Verilog language which are not normally compilable to hardware are not compiled to C by the VTOC compiler, although there is limited support for meta-commands such as `$display`.

The purpose of the VTOC compiler is to enable sections of Verilog to be used by non-hardware engineers and people without Verilog tools or licences in functional emulations of that hardware. Typically, the output from VTOC will be included as one of the modules in a system simulator or emulator.

C and Verilog have a great deal in common, especially in the syntax and semantics of their operators. Verilog has additional operators for selecting bits from vectors and concatentating expressions to form vectors, but does not support multi-dimensional arrays, pointers or recursion [1]. The statement commands are similar, but there are many small differences, especially in the '`case`' statement. Major differences are that Verilog has two assignment operators (blocking and non-blocking). Verilog also has continuous assignments and multiple threads.

* djg@cl.cam.ac.uk

A possible approach would be to macro-expand Verilog into C and wrap the result in a threads package which was augmented with special support for the Verilog simulation cycle. Such an approach can either preserve the full simulation semantics or just the cycle semanatics.

Our approach is to use standard Verilog preprocessing and compilation algorithms as embodied in Verilog hardware compilers, such as CV2 [2] and CV3 [3], to elaborate the input language into a set of assignments. (The difference between CV2 and CV3 is the richness of the source Verilog that can be compiled: both support all common RTL constructs, but CV3 handles conditional event control and selective unwinding of '`while`' loops.) After further processing, these become assignments between integer C variables in the target C environment. In this way, we generate a section of C which can be run without a special threads library and which emulates the operation of synthesised hardware.

An advantage of our approach is that speed is gained since logic minimisation is performed and simulation-level detail is thrown out. Potential disadvantages of our approach are that handling of uncertain values, temporary bus fights, Verilog meta-language commands, such as '`$display`' and other simulation-level operations is degraded. Since we do not use a threads package, but instead use a single thread to emulate a large hardware section, combining separately-compiled sections at the link-editor stage can be problematic. This is discussed in section 4.

2 Processing Stages

Figure 2 shows the processing stages used by the compiler. The compiler input stage performs Verilog macro preprocessing and parsing. It also implements basic identities which simplify the language, such as converting '`always`' into '`intial while (1)`', and converting '`for`' into its equivalent '`while`' form. Basic checking of multiply defined identifiers and missing modules is performed.

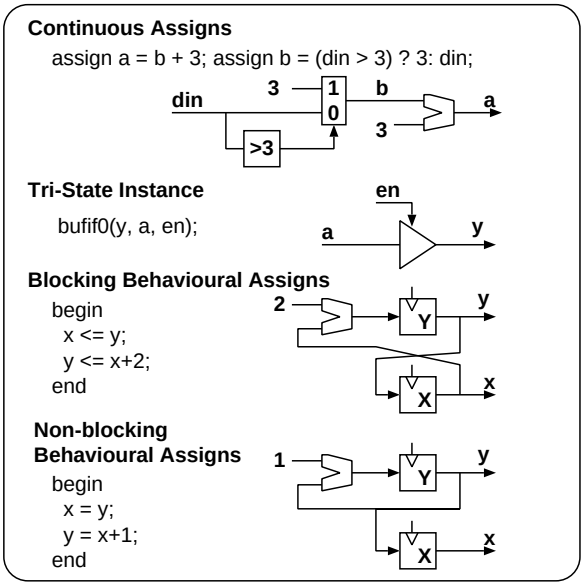

Figure 1. Examples of Verilog's Assignment Statements

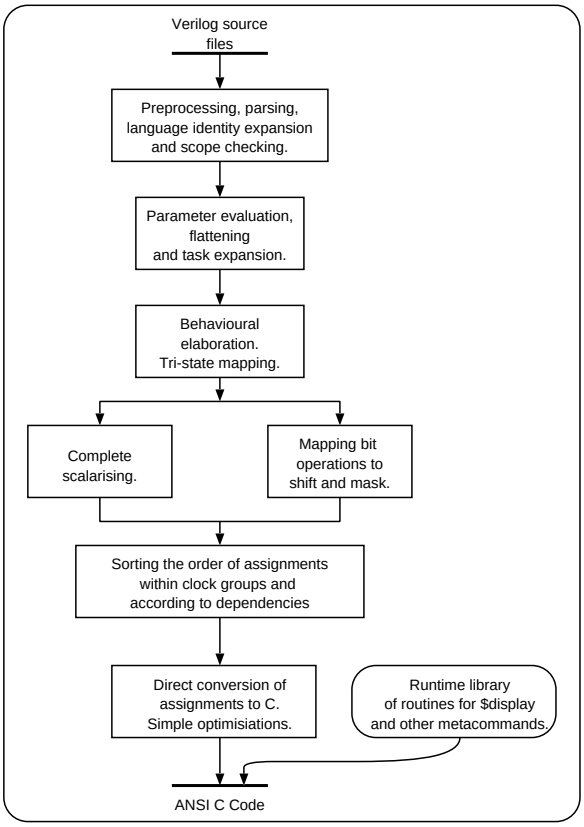

Figure 2. Processing stages in the Verilog to C compiler.

2.1 Parameter Expansion and Flattening.

The compiler can handle either a single module or a heirachy where a top-level module includes instances of sub-modules. Where a heirarchy of modules is used, the lowest, leaf modules are implemented only in RTL or behavioural Verilog. Specify blocks, used in Verilog to describe complex primitives such as flip-flops and other functions, are not supported: instead, the user must write equivalent leaf modules in RTL for any such primitives. Built-in gates are converted to continuous assigns. The compiler flattens such heirachies to give a single module whose signature is the same as the original top-level module, but where the RTL and behavioural code from all the intermediate modules has been combined. This stage of processing evaluates parameters and parameter overrides and textually expands tasks. Functions are elaborated in the next stage since they cannot be textually expanded here since Verilog has no statement-as-expression or '`valof-resultis`' construct.

In the flattening stage, bodies of multiple instances of the same sub-module are copied out and concatenated by the compiler, but with renaming of their components to avoid clashes. Global renaming is also possible via command-line flags so that the user can combine separately compiled sections of generated C without clashes in the link editor.

2.2 Behavioural Elaboration

The resulting, large, simplified module is compiled by behavioural elaboration ([1, 2]) to an unordered set of assignments. Verilog can be regarded as having four major types of assignment, as illustrated in figure 1. A fifth type, known as behavioural continuous assignments, are not handled for logic synthesis.

Logic synthesis converts block assignments, whose textual order is generally significant, into non-blocking assignments, whose order is not significant. A list of non-blocking assignments are a register transfer level (RTL) hardware description. Once blocking assignments have been removed by the behavioural elaboration, there are three remaining classes: clocked, tri-state and continuous. There is an irony in that C essentially only supports blocking assignment (order is sensitive in C) whereas the reduction to cycle-semantics generated by the logic synthesis removes these and generates non-blocking assigns. The solution is the dependency sorting described in section 2.4.

Each of the remaining three types of assignment has left and right-hand expressions, but the clocked assignments have also a clock expression and possibly an asynchronous reset or preset expression and the tri-state assignments have an associated enable expression.

The tri-state assignments are then converted to continuous assignments by collating on the left-hand side net name and then building a multiplexor using the conditional expression operator. This leaves only two types of assignment for further processing.

As in many Verilog compilers, Verilog arrays can be converted to sets of registers if desired. Verilog only has one dimensional arrays: each location is typically a number of bits wide. The conversion to scalar form is controlled by command-line options or embedded comments in the source file, with the default being that arrays smaller than 64 in length are converted to individual C variables and others map to C arrays. When scalarising is enabled (see next section), the array locations may also be split into separate variables, or the array may be split into separate arrays, one per bit.

2.3 Bit-level processing.

As shown in figure 2, alternative schemes are possible for handling the bit-level operators supported by Verilog: expressions can either be scalarised in the compiler or left in vector form. The choice is normally determined by a command-line flag to the compiler. When vectored, broadside registers of up to the target C integer size in width (32 or 64 bits) are held in one C variable. When scalarised, each C variable holds just one bit and the right-hand side of expressions is made only of simple Boolean operators. In the vectored form, the right-hand sides of the generated expressions will include further C operators, including addition, subtraction, multiplication, division, modulus and left and right shifts by variable amounts. If a register or bus is wider than the target integer size then it is forced to be scalarised into a number of single bit locations.

Logic minimisation is performed in these stages. A number of algorithms are possible. Since multi-level logic is not being generated, straightforward Espresso is a good choice for scalarised signals. The minimisation of non-scalared, general C expressions appears to be a vast subject: our current compiler only implements peephole minmisiation, based on identities: for example '$a + 0 \equiv 0$' and '$(1)?t : f \equiv t$'.

2.4 Dependency Sorting

In Verilog, blocking assignments which are triggered from a common event, such as a clock edge, take place in parallel. Owing to the serial thread of execution found in C, assignments which take place in parallel in the Verilog implementation must either be placed in an order such that the same overall effect is achieved or else an intermediate variable must be introduced to model the parallelism. An example of an intermediate variable being introduced is shown in section **??**. The intermediate variable acts like the master portion of a D-type flip-flop in a hardware environment whereas the variable which is externally visable acts like the slave.

The continuous assignments and the non-blocking assignments have exactly contrary requirements on their dependencies and the same sorting algorithm is used for both, but with the comparison predicate negated in one of the runs.

For non-blocking assignments, which are clocked in parallel off the same clock, they must be sorted such that the effects of an earlier assignment *do not* influence the right-hand sides of any subsequent assignments. If this were to happen, it would give an effect akin to clock skew, where the effect of changing the output of one flip-flop is experienced at the input to a second flip-flop before that second flip-flop receives it clock edge. If the sorting algorithm cannot produce an ordering, owing to cyclic dependencies, then it breaks the cycle by introducing the above-mentioned intermediate variables.

For continuous assignments, it is intended by the designer that all combinatorial paths and loops have been executed to their fullest extent, with all outputs settling down, before the next clock cycle. This requires an order where if a variable which is assigned occurs in the right-hand side of another continuous assignment, then this variable should be placed before the other in the output sequence. If the sorting algorithm cannot find such an order, owing to a combinatorial loop in the logic, then it prints a warning and generates an arbitrary ordering. In most sensible designs, this will have little influence. In a bad design, where the loop inverts and is fully enabled under certain circumstances (i.e. settings of inputs and state variables) then the C code will produce an oscillating value with a half cycle once per call, mirroring what the hardware would do.

2.5 Conversion to C and C-level Optimisation

A single ANSI C output file is generated per compilation. A report file is also generated. The output file (or files from multiple compilations if used) need(s) to be linked against the file '`ttvtoc.c`' which provides various runtime system resources, such as the code for '`$display` for printed output and `$read_memh` for initialising an array from a data file.

The output file generated by the compiler contains a single C routine with default name 'ttv' but this can be changed using the `-id` command-line flag to support multiple compilations. The top-level module has the same signature in C as the top-level module in Verilog had. Signals which enter or leave the top-level module are modelled as references (pointers) to external instances of integer variables. These instances are created by the user in a top-level simulation

wrapper.

```
   void ttv(unsigned int *out, un-
signed int *in, unsigned int *clk, ...)
```

Signals which are local to a module are instantiated as static, variables of the next largets width suported in C. Being static, they retain their value from one call to another.

In the C language object code, clocked assignments are normally enclosed inside '`if`' statements so that they are only executed when a clock edge is being simulated (section 3.3).

The order of components in the resulting C section is as follows:

1. Definitions of variables. All variables are statics or arrays of statics. There are several sources of these integers, including integers and '`reg`' variables in the user's program, state variables to reflect execution pointers, intermediate variables to manage parallel assignment and shadow variables to hold the previous value of nets for which edge detectors are needed. All are declared as '`unsigned`' except for those corresponding to integer declarations in the source Verilog. This is necessary to make the comparison operators work correctly, since only integers are signed in Verilog.

2. Assignments to variables representing combinatorial nets which are used as the D-inputs to flip-flops (i.e. which occur in the right-hand side of a blocking assigment after elaboration).

3. Assignments to the master intermediate variables or directly to slave '`reg`' variables if the dependency sorting prevented the need for a master.

4. Assignments from masters to slaves.

5. Assignments to combinatorial outputs, preceeded by assignments to intermediate nets which feed into combinatorial outputs. Buffered or inverted versions of clocked nets are here regarded as combinatorial, even though buffers and inverters do not combine signals.

6. Assignments from nets to their shadow variables for nets which are used in edge detectors.

Some assignments occur more than once: an example is where a net is both directly or indirectly the input to a D-type in the module and indirectly part of an output from the module that also depends on the state of D-types.

After the assignments to the slaves of the flip-flops (between points 4 and 5), the code for any meta-commands is provided so that they are executed in a state where all flip-flops have a consistent state. Such meta commands are typically enclosed inside a C '`if`' statment, since they will be conditional in the source Verilog and so do not happen every cycle. Also at this point, an include file is pulled in, where the user can insert print statements or other debugging information.

2.6 Variables of specific bit widths.

Variable declarations generated by the compiler can either use the native sizes availabe in C, which are char, short, long, and long long, or a richer set of types names 'uxx' where 'xx' is an integer denoting the number of bits. For example, '`u7`' is a seven bit, unsigned quantity The generation defaults to the 'u' form but can be toggled with the '-nou' command line flag. Macro definitions supplied to the C preprocessor subsequently convert the 'u' form to the next largest C type. One advanatage of the 'u' form is that it is directly understood by our companion C to Verilog compiler [4], and so enables a better closure should a design be converted between C and Verilog a number of times.

The compiler does not take any notice of the Verilog keywords '`vectored`' and '`scalared`' because these only change the simulation semantics and not the synthesis semantics.[1]

3 Rules used to convert Verilog to C.

Some of the rules for conversion to C are presented using the following examples where the Verilog on the left or top box is converted to the C on the right or bottom box.

3.1 Unary operators

Most Verilog operators map directly into C. However, Verilog has a number of unary operators which can take the individual bits of a bus and reduce them under one of the three operators, AND, OR and XOR. These must be expanded as the following example for AND shows:

`wire [3:0] a;`
`wire p = (&(f));`
	`{`
	`  p = 1 & (a & a>>1 &`
	`a>>2 & a>>3); }`

Neither C nor Verilog has an explicit boolean datatype that must be used for the control expression in the statements which use a condition expression, such as '`if`'. Instead any expression may be used. There is an implied reduction under the operator OR, found in both Verilog and C, which is inserted automatically in conditional statements such that if any bit is non-zero the expression acts as true. The VTOC compiler inserts the operator as an explicit construct in the input parse tree, but removes it again if possible when generating the output C code.

[1] Their main effect for simulation is to control whether all of a vector is uncertain if any part of it is. The compiler does not model uncertain values at runtime: the only use of '`x`' is to denote 'don't care' at compile time.

3.2 Example of Input and output signals

This example shows how input and outputs are mapped to references whereas locals are mapped to statics. It also uses two simple gates, one of which is one of the builtin gates found in Verilog[2]

```
module A(a, b, p, q);
   input a, b;
   output p, q;
   wire abar =  a;
   and (p, abar, b);
   assign q = a & b;
endmodule
```

```
void ttv(u1 *a, u1 *b,
         u1 *p, u1 *q)
{
   static u1 abar;
   abar =  *a & 1;
   *p = abar & *b;
   *q = *a & *b;
}
```

When combinatorial paths through the top-level module are present, as in the example, the outputs will only be updated to reflect input changes each time the generated routine is called. Therefore, such paths might appear to have transparent latches in series with them, which are activated only as frequently as the routine is called.

3.3 Example of clocking

Many designs have a single master clock. Some designs have a single clock input, but internally divide this down to generate internal clocks of a lower rate. A third class of designs has more than one clock. In the first two cases, only a single clock signal is present in the top-level module.

The compiler can be operated in a mode, known as 'mode 1', where a command-line flag takes an argument which is the name of an identifier present in the top-level module which is to be treated as a clock input. When mode 1 is used, the clock input signal is not actually used. Instead, each time the generated C routine is called by a C thread, the system emulates the advance which would occur at the next active edge on that clock net. Therefore calls to the routine are clock cycles. All event control statements in the Verilog which are sensitive to this clock must have the same polarity (i.e. be all positive or all negative edge triggered).

The default mode is 'mode 0', where no top-level net is nominated as a clock net. In mode 0, the routine must be called sufficiently frequently not to alias the main (fastest) clock, or any other clocks which are fed in at the top-level.

[2]The Verilog shown in this example of a complete module does not include the housekeeping lines generated by the real compiler, which include the inclusion of "ttvinc", other header files and so on. These can instead be seen in Figure 4.

`always @(posedge clk)`
`  begin`
`  a = ~c;`
`  end`
	`}` Mode 0
	`  if (*clk != clk_hist)`
	`  {`
	`  a = ~c;`
	`  }`
	`  clk_hist = *clk;`
	`}`

`always @(posedge clk)`
`  begin`
`  a = ~c;`
`  end`
	`{` Mode 1
	`  a = ~c;`
	`}`

`always @(posedge clk)`
`  begin`
`  a = ~c;`
`  end`
	`{` Mode 2
	`  while(1)`
	`  {`
	`  a = ~c;`
	`  cx_barrier(*clk);`
	`  }`
	`}`

The final clock option generates 'mode 2' C code, which has the semantics expected by our CTOV compiler. In mode 2, there is only one clock and the body of the C code is enclosed in an infinite loop with a call to a threads or co-routine suspend function called "cx_barrier()" at one or more points in the loop. Again, the C-to-V compiler is able to convert this C code back to semantically-equivalent Verilog, but a threads library is needed to run more than one such section of code in parallel in the C environment.

3.4 Concatenations

Concatenations occuring in expressions are handled by introducing shifts, as shown here:

```
wire [3:0] a, b, c;
assign a = { b[3:2], c[3:2] };
```

```
{
   a = (b & 12) | ((c>>2) & 3);
}
```

Behavioural and continuous assignments to left-hand side concatenations of more than one item are handled by first assigning to an intermediate variable so that any side effects in right-hand side function calls only occur once and the effect of parallel assignment can be achieved if any element of the l.h.s. appears in the r.h.s.. Then the left-hand components are assigned to, one by one, with the right-hand argument shifted right by the appropriate amount. Here is an example using continuous assignments:

```
wire [3:0] a, b;
assign { b[3:2], a[3:2] } = a;
```

```
{
    t = a;
    a = ((t>>2) & 3) | (a & 12);
    b = (t & 12) | (b & 3);
}
```

3.5 Behavioural Elaboration.

Behavioural elaboration performs symbolic evaluation to mirror what a thread of execution would actually evaluate in a simulator. The results is a set of RTL assignments,

one per variable that is updated by the thread. Unlike the behavioural input code, RTL has the property that order of listing the transfer statements is not significant.

In this example, the assignment to v has been subject to a further reordering, according to the dependency sorting rules of section 2.4, to be placed after v is used to set up d.

```
always @(posedge clk)
   begin
   a =  b & c;
   v <= a;
   d = v | c;
   end
```

```
{
  a =  b & c;
  d = v | c;
  v = a;
{
```

An example of compiling an addition in vectored form is as follows. Note that overflow in the fixed-field register variable is handled by the binary AND against a constant mask (3).

```
reg [1:0] sum;
always @(posedge clk)
sum <= sum + 1;
end
```

```
{
  sum = (sum + 1) & 3;
{
```

Here is the same example in scalared form

```
reg [1:0] sum;
always @(posedge clk)
    sum <= sum + 1;
    end
```

```
{
  sum_1 = 1 & (sum_0
     ^  sum_1);
  sum_0 = 1 &  sum_0;
{
```

Here is an example where an intermediate variable is needed in the C version to implement a swap. The intermediate variable name is generated by the compiler by appending '_t'.

```
always @(posedge clk)
    a <= b;
    b <= a;
    end
```

```
{
   a_t = b;
   b = a;
   a = a_t;
}
```

4 Multiple Compilations.

Linking together the output from multiple separate compilations is done using an automatically generated top-level C file and the normal system link editor. The top-level file contains instances of the modules generated by the separate compilations and instances of sets of variables which represent the wires between the modules. To achieve the behaviour of normal synchronous hardware, each module is passed references to a private set of the wire variables and changes in value are propagated between the sets after the single thread of execution has run the C subroutined generated from each compilation.

5 Performance and Conclusion.

To compare execution time of the generated C with the Verilog original, an example that finds all the primes up to 2^{17} using the sieve method is presented. Table 1 shows the runtime in seconds for the example under the two popular simulators from Cadence and with VTOC. The design was compiled to gate level and simulated again to give the second line of results. The design used 503 gates and a 32x8 SRAM. The run time for 12000 cycles of a 12000 line DSP/Viterbi unit is also shown.

Design	Cadence Verilog-XL 2.7.10	Cadence NC-SIM	CTOV (gcc)
Prime Gen Behavioural	105	11	1
Prime Gen Gate-level	5085	676	323
Viterbi Unit	113		3

Table 1. Run time in seconds for the 'Primes' example. UltrasparcII 200 MHz.

The program has met its design goals, in that it allows portable, licence-free versions of Verilog designs to be generated. It also provides a very-effective native-mode, cycle-based simulator, being more than 100 times faster than Verilog-XL. The compiler has been tested on large sections of Verilog that include commercial DSPs and custom digital filter blocks. A commercial product using our approach is available from TenTech[5] and a similar product is available from CAE Plus [6]. In future we need to add support for combinatorial exchanges between separately-compiled modules, which cannot be accurately modelled with the method described above. We can also add a simple library which will cause generation of tracing information for viewing with a standard graphical viewer, such as Cadence C-waves.

[1] "The Verilog Hardware Description Language" Donald E.Thomas and Philip Moorby. Published by Kluwer Academic Publishers. ISBN 0–7923-9126-8.

[2, 3] 'The CSYN Verilog Compiler' DJ Greaves. Presented at International Workshop on Field Programmable Logic, Oxford, 1st September 1995. Proceedings published by Springer Verlag LNCS. ISBN 3540 60294-1. See also www.cl.cam.ac.uk/users/djg/localtools/.

[4] "Cadence Native Mode NC-Verilog Compiler." www.cadence.com.

[5] "VTOC and CTOV Verilog to C Exchange Compilers." Download from www.tenisontech.com.

[6] ArchGen RTL-C compiler from CAE Plus. www.cae-plus.com