Verilog - accelerating digital design

Gerard M Blair

ABSTRACT

At first glance, Verilog is simply a language for digital hardware simulation - but in practice it has become the linchpin for a complete design flow from concept to digital component. This article describes the ideas behind the language and its growing role in digital design.

1. Overview

Verilog is a hardware description language. This means that it is a (sort of programming) language which allows a designer to describe a component in text rather than as a schematic. Thus, while a 4-bit comparator could be drawn as a schematic of four symbols (of a one-bit comparator) connected by wires, it can also be written in Verilog as in Figure 1.

```
module four_bit_comp (gt_out, A, B);
output       gt_out;
input  [3:0] A, B;

wire   [2:0] carry;
supply0      gnd;

one_bit_comp b0 (carry[0], A[0], B[0], gnd),
             b1 (carry[1], A[1], B[1], carry[0]),
             b2 (carry[2], A[2], B[2], carry[1]),
             b3 (  gt_out, A[3], B[3], carry[2]);

endmodule
```

Figure 1: Verilog four-bit comparator (like a schematic)

Verilog allows hardware to be described in a wide variety of styles. For instance, the comparator in figure 1 is written as four calls to a sub-module: conceptually, a direct textual representation of a schematic with no greater or lesser "design content". Instead we could write it in terms of the function it performs, as in figure 2 where the *(A > B)* is a Verilog relational operator which is true (HIGH/1) if A is greater then B and false (LOW/0) otherwise. Thus the *function* is directly expressed in the programming language.

```
module four_bit_comp (gt_out, A, B);
output       gt_out;
input  [3:0] A, B;

assign gt_out = (A > B);

endmodule
```

Figure 2: Verilog four-bit comparator (functional)

With this representation, the function of the module is very clear, and very easy to write. The clarity is a vital aid to design quality since clear code is easier to debug, maintain and upgrade. For instance, suppose the module were to be changed to output the larger of the two inputs – consider the changes needed in the schematic. The Verilog can be rewritten as figure 3 where *(<expression>) ? <statement> : <statement>;* is the same syntax as the *if-statement* in the language *C*). Thus in Verilog, the new module is created by changing the right-hand-side of one assignment and the output from a single bit to a bus.

```
module four_bit_larger (larger, A, B);
output [3:0] larger;
input  [3:0] A, B;

assign larger = (A > B) ? A : B;

endmodule
```

Figure 3: Verilog four-bit comparator returning larger input

In itself, this is a useful method of design capture, favoured by designers with a strong computer science background; but the majority of digital designers still feel more comfortable with schematics. The true virtues of Verilog, however, lie elsewhere in that:

- many different levels of description can be used within the same design environment - which leads to very effective design flows
- all such descriptions can be verified through simulation, and the same simulation can be used for all levels of description and ultimately for testing the fabricated component
- there are now synthesis tools which allow very simple descriptions to be automatically translated into gate-level netlists: neither figure 2 nor 3 requires further design effort

In the following section we will look at the history of Verilog and of its main rival, and then return to the virtues outlined above.

2. Birth, growth and rivalry

Verilog was developed at a time when designers were looking for tools to combine different levels of simulation. In the early 1980s, there were switch-level simulators, gate-level simulators, functional simulators (often written ad-hoc in software) and no simple means to combine them. Further, the more-widespread, traditional programming languages themselves were/are essentially sequential and thus "semantically challenged" when modelling the concurrency of digital circuitry.

Verilog was created by Phil Moore in 1983-4 at Gateway Design Automation and the first simulator was written a year later. It borrowed much from the existing languages of the time: the concurrency aspects may be seen in both Modula and (earlier) Simula; the syntax is deliberately close to that of C; and the methods for combining different levels of abstraction owe much to Hilo (from Brunnel University, UK).

In 1989, Gateway Design Automation (and rights to Verilog) were purchased by Cadence who put Verilog in the public domain in the following year. This move did much to promote the use of Verilog since other companies were able to develop alternatives tools to those of Cadence which, in turn, allowed users to adopt Verilog without dependency on a single (primarily workstation-tool) supplier. In 1992, work began to create an IEEE standard (IEEE-1364) and in December 1995 the final draft was approved. Thus Verilog has become an international standard - which will further increase its commercial development and use.

At present, there is standards activity to extend Verilog beyond purely digital circuits. This includes Verilog-MS for "mixed signal specification" and Verilog-A for "analog" design; the latter was recently approved (June 1996) by the board of Open Verilog International and is now under consideration by the IEEE. In addition, work is underway to automate the proof of "equivalence [between] behavioural and synthesizable specifications" (see the Cambridge web site below) to which Verilog readily lends itself.

While Verilog emerged from developments within private companies, its main rival came from the American Department of Defence (DoD). In 1981, the DoD sponsored a workshop on hardware description languages as part of its Very High Speed Integrated Circuits (VHSIC) program, and the outcome formed a specification for the VHSIC hardware description language (VHDL) in 1983. Because this was a DoD programme, there were initially restrictions its dissemination, until 1985 when the development was passed on to IEEE whose standard (IEEE 1076) was formally accepted in 1987.

There is, of course, the question as to which language is better. And this, of course, is a hard question to answer without causing excitement and rebuttals from the marketing departments of the less-preferred language. However, the following points featured in a recent debate in the VHDL and Verilog news groups.

The main factor is the language syntax – since Verilog is based on C and VHDL is based on ADA:

- Verilog is easier to learn since C is a far simpler language. It also produces more compact code: easier both to write and to read. Furthermore, the large number of engineers who already know C (compared to those who know ADA) makes learning and training easier.
- VHDL is very strongly typed, and allows programmer to define their own types although, in practice, the main types used are either the basic types of the language itself, or those defined by the IEEE. The benefit is that type checking is performed by the compiler which can reduce errors; the disadvantage is that changing types must be done explicitly.

Verilog has two clear advantages over VHDL:

- it allows switch-level modelling - which some designers find useful for exploring new circuits
- it ensures that all signals are initialized to "unknown" which ensures that all designers will produce the necessary logic to initialize their design - the base types in VHDL initialize to zero and the "hasty" designer may omit a global reset

VHDL has two clear advantages over Verilog:

- it allows the conditional instancing of modules (*if/for ... generate*). This is one of those features that you do not miss until you have used it once - and then you need it all the time. Many Verilog users recognize this lack and create personal pre-processing routines it implement it (which negates some of the advantages of a language standard).
- it provides a simple mechanism (the *configure* statement) which allows the designer to switch painlessly between different descriptions of a particular module. The value of this is described in the next section.

Selecting a design language, however, cannot be done by considering the languages in isolation. Other factors must include the design environment, the speed of simulation and the ease with which the designer can test-and-debug the code: the design environment is crucial. Verilog includes the Programming Language Interface (PLI) which allows dynamic access to the data structure. For the expert user this gives a degree of control which few simulators (if any) can match. For the tool-designer it encourages the development of better design environments with tools such as customized graphical waveform displays, or C-language routines to dynamically calculate delays for timing analysis.

Pragmatically, both languages have a large installed base and design-investment – thus a designer needs to know both. However, the market place is now being won by Verilog: the latest figures (EDAC's market research) give Verilog a nearly 2:1 lead over VHDL in tools' revenue.

3. A Possible Design Flow

The initial importance of Verilog is that it supports a hierarchical design style coupled with mixed-mode simulation. To understand this, consider the question of how does the designer know that the design does what it is supposed to? The answer is: *through simulation.* To understand how Verilog provides this verification, let us consider a possible design flow.

The first step in converting a design specification into a hardware design is to decide how to test it. For a digital design, this requires the creation of full set of test vectors which exercise every stated feature in the specification and a corresponding set of output vectors; together these are often known as the "golden" vectors. Of course, it is unrealistic to suppose that these will be written correctly on the first pass. And as the design progresses, the test vectors, and the specification itself, will need to be upgraded. However, the point remains that by using Verilog, the test vector generation, the component design and the verifying simulation are all conducted within a single unified language.

The second step is to break the problem down into smaller pieces. Hierarchical decomposition is the divide-and-conquer process of describing each module in terms of simpler modules so that function of the first module is *easily* understood in terms of their concerted action. This process can be seen as one of *complexity containment.* Whereas the full design may be too complex for the designer to grasp at once, its description in terms of a few simpler blocks is not. Each of the simpler blocks can then be implemented independently (indeed potentially by different designers) and rendered both understandable and less complex in terms of even smaller blocks, and so on until the simpler blocks each become so simple that it is easily understood in its entirety.

```
module complexModule ( ------- );
   --------
   Amodule A ( ------- );
   Bmodule B ( ------- );
   Cmodule C ( ------- );
   --------
endmodule

module testVecGeneration ( ------- );
   --------
   complexModule whole ( ------- );
   --------
endmodule
```

Figure 4: Hierarchical decomposition in Verilog

In Verilog this is done by writing sub-modules. A complex module is redefined as calls to sub-modules (as in figure 4) which are themselves described in Verilog. The equivalence of the new description to the original is then verified by simulation using the golden vectors. The sub-modules are each simpler than the complex module they jointly implement – and can be designed independently. While this process can be followed in other design environments, the point is that Verilog naturally supports it.

As each sub-module is developed, the ultimate test is to verify it in the context of the complete design and the golden vectors. This is practical because Verilog allows mixed-mode simulation. For instance, the designer of sub-module *Bmodule* can verify it using a simulation of *complexModule* with the modules *Amodule* and *Cmodule* in their original description (which is normally the fastest in terms of runtime). Thus the simple modules simulate the transformation by the golden vectors the hardware which surrounds the module under test.

Clearly this process would benefit from a simple mechanism to switch between different descriptions of the same module, which Verilog lacks; this is where an equivalent to the VHDL *configure* statement would be useful.

4. Synthesis

In recent years, however, the main importance of Verilog has become its use in synthesis.

Synthesis is a blanket term which refers to the automatic translation of HDL code into an equivalent netlist of digital cells. Essentially the synthesis tool is a collection of artificial intelligence (AI) programmes which interpret, optimize and retarget designs expressed in an HDL: they have captured design expertise which other designers can then apply automatically.

The first stage is for the synthesis tool to recognize structures in the Verilog code in terms of either abstract design concepts (such as finite state machines) or digital logic functions. Thus figures 2 and 3 complete the design of their respective functions: the detail of figure 1 (corresponding to the schematic) is no longer necessary. Some examples of other structures which can be recognized will be explained below.

The second stage is to apply various automated techniques to optimize the design, for instance:

- boolean logic reduction (including removing fixed signals)
- architectural selection (for instance, selecting an adder architecture)
- state machine minimization
- clock and signal distribution network generation
- automatic insertion of pipeline stages

and the criteria for optimization can be area, critical path delay, and (now even) power consumption. Furthermore, as the AI tools become more sophisticated (and as more designer expertise is captured within the tools) less detail is required in the design description and so design time is reduced.

The final stage is to map the design onto a specific technology. This is a key point: the design description *should* be totally independent from the technology which implements it. All the technology specific details are handled by the synthesis tool. Thus design porting is performed by changing a synthesis parameter.

In practice, the distinction between optimization and technology targeting is blurred since the characteristics of the cell-library affect the choice of the optimizer algorithms.

This level of automation dramatically increases the the speed of design, which is changing the normal design flow. Previously, there was a long delay between high-level architectural decisions and a realistic analysis of their consequences (since all the design work had to be performed by the designer). With synthesis, so much of the implementation work is performed automatically that it is feasible to explore a far wider range of architectural alternatives. Furthermore, the implementation of any one architecture can be similarly explored by varying parameters in the synthesis tool to optimize for such as parameters as: latency, clock frequency, area, delay, etc. In short, synthesis enables rapid design exploration which enable faster time-to-market of better design solutions.

To give a small sample of how synthesis works, we will consider a few "mappings" from Verilog onto digital logic. In each of the following, it is simply a matter of the synthesis tool recognizing structures in the underlying code for which it knows a digital-logic equivalent.

Figure 5 shows an alternative syntax for an *if-statement* to that in figure 3. Either of these forms imply a multiplexor. In the Verilog code, the outcome (the assignment to the variable *out)* is either *a* or *b* depending upon the value of *cnt;* in the corresponding hardware, the output of the multiplexor *(out)* is one of the two input signals depending upon the value of the control signal *cnt.* Thus, the synthesis tool will look for *if-statements* and (in some circumstances) transform them into a call to a multiplexor in the output netlist.

In the same manner, the synthesis tool also looks for simple arithmetic operations and transforms them to corresponding hardware. For instance, figure 6 illustrates that an addition in the Verilog code becomes a call to an adder in the resulting netlist. During this transformation, the logic architecture of the resulting adder will be chosen from several options so as to best fulfil the optimization criteria

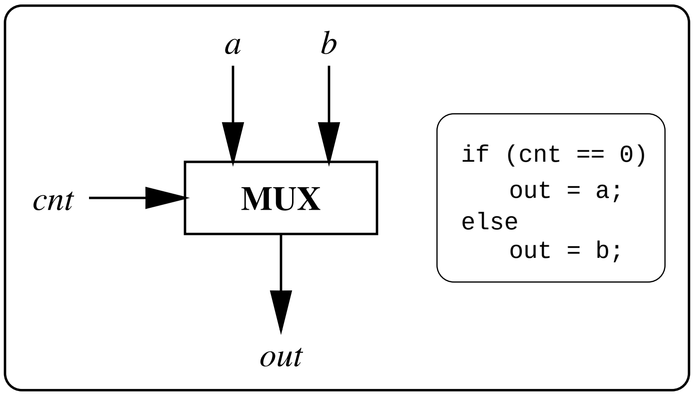

Figure 5: Verilog code for a multiplexor

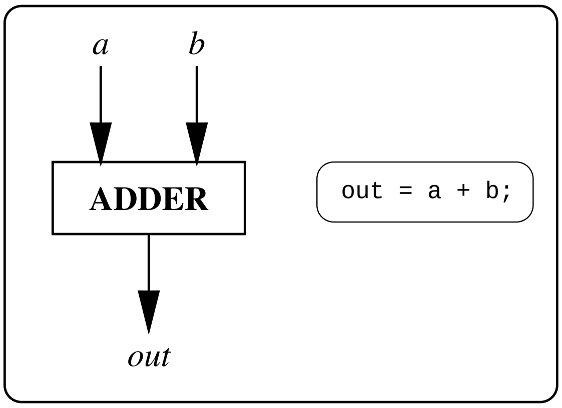

Figure 6: Verilog code for an adder

(high speed or low-area) of the synthesis run.

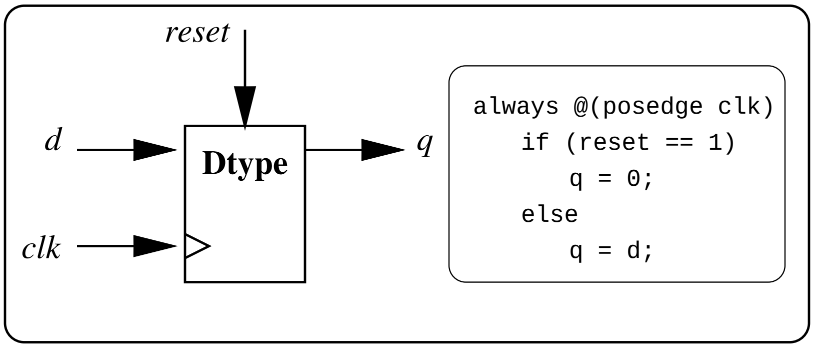

Figure 7: Verilog code for an edge-triggered register

The code in Figure 7, illustrates the syntax by which Verilog encodes events. The phrase *always @(posedge clk)* can be read as *whenever there is a positive edge on the clk signal, perform the following statement.* In this case the "following statement" is a conditional assignment of the variable *q* either to the input *d* or to zero. Because this *if-statement* is guarded by *@(posedge ---),* the synthesis tool knows to map it to a state-element and not to a multiplexor (as in Figure 5). The synthesis tools "recognizes" the pattern in the code of figure 7 as a "Dtype with synchronous reset".

As a final example, consider Figure 8. Here the syntax: *{ X, Y }* represents a concatenation of two variables. Thus if *X* were declared as a 3-bit register/wire and *Y* as a 4-bit register/wire, then *{ X, Y }* would be a 7-bit register/wire with the left-most 3 bits being those of *X* and the remainder being those of *Y.* If, in figure 8, *sReg* is declared as an (n-1) bit register, then the simple assignment of *{q, sReg} = {sReg, d}* within an *@(posedge ---)* represents an edge-triggered, serial-in, serial-out, n-bit shift-register.

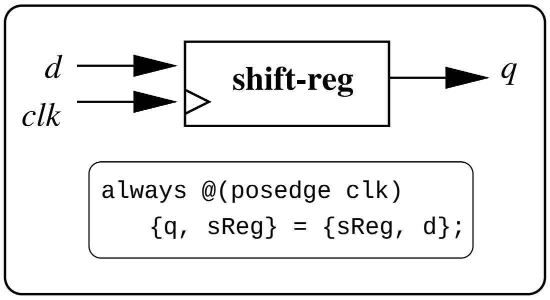

Figure 8: Verilog code for a shift-register

We can thus see that the complete encoding of quite complex hardware modules can be achieved in relatively simple text; however, there are pitfalls. The synthesis tool is merely a collection of AI routines - if the tool has not been coded with a particular piece of knowledge, then it becomes a programming challenge to "force" the synthesis tool to implement a design in the preferred manner. For instance, take the design of a simple module with three inputs *a, b* and *cnt* where the output *out* is equal to *a + b* if the control signal *(cnt)* is LOW, and to *a - b* if the control signal is HIGH.

```
module adder(out, cnt, a, b);
output [3:0] out;
input        cnt;
input  [3:0] a, b;

assign out = (cnt) ? a - b: a + b;

endmodule
```

Figure 9: simple coding of add/subtract unit

Figure 9 shows a simple piece of Verilog to encode this specification: depending on the value of *cnt, out* is assigned to either *a - b* or *a + b.* A synthesis tool (unless it has been specially programmed to catch this case) will interpret this conditional statement as a multiplexor selecting between the outputs of two separate modules; one which adds *a* and *b,* and one which subtracts.

Any competent designer, however, knows that a twos-complement subtraction can be performed by inverting the second operand and adding "1". With this in mind, the code can be rewritten as in figure 10 with *cnt* forming a conditional "add 1" (that is "add the value of *cnt*) and also conditionally inverting *b* by using XOR gates. This version will synthesis to a single adder module: without the extra subtractor.

```
module adder(out, cnt, a, b);
output [3:0] out;
input        cnt;
input  [3:0] a, b;

assign out = cnt + a +
             ({cnt, cnt, cnt, cnt} ^ b);

endmodule
```

Figure 10: optimized coding of add/subtract unit

The point is that although the synthesis tool can save enormous amounts of design time, the best results will only be obtained if the designer fully understands its methods – and its limitations.

5. Final comments

Textual capture of digital designs can be superior to schematic capture both in terms of clarity and design time (especially with the simpler syntax of Verilog). The design flow itself is enhanced both by the integrated simulation environment of behavioural to switch-level descriptions, and by the existence of synthesis tools which look after much of the previously-costly design details and which, in turn, allows greater exploration of alternative architectures.

For commercial designs, the optimal path to market is now to use synthesis and to turn a specification into a sythesizable description as quickly as possible. This allows rapid time to market. Then the design work continues to modify both the Verilog code and the target libraries to improve the component's performance. In this way, the first product release is achieved with full functionality at a early stage in the total design cycle. Furthermore, early market reaction may either suggest additional features for the second release, or bring the project to a early (but less costly) conclusion.

Although VHDL has two significant semantic advantages over Verilog (the *generate* and *configure* statements), Verilog is often preferred because of the relative simplicity of its syntax and the engineering "feel" that it is more closely connected (if desired) to the eventual hardware. It seems significant that Verilog is actually an anagram of "GI lover": although VHDL was mandated with the might of the American military establishment, the foot-soldiers prefer Verilog for its simple accessibility.

VERILOG RESOURCES - personal favourites

NewsGroup:
comp.lang.Verilog

WebSites:
http://www.cl.cam.ac.uk/users/mjcg/Verilog/
http://www.comit.com/˜rajesh/verilog

Book:
Verilog HDL by Samir Palnitkar - 1996 -
pub. Prentice Hall - ISBN 0-13-451675-3

Gerard M Blair is a Senior Lecturer at the Department of Electrical Engineering, The University of Edinburgh, The King's Buildings, Edinburgh, EH9 3JL, Scotland, UK – Tel: +44 131 650 5592 – Email: gerard@ee.ed.ac.uk