Digital logic simulation: event-driven, cycle-based, and home-brewed

Clive "Max" Maxfield,
Intergraph Computer Systems

Logic simulation is an interesting beast. Most engineers know what it does and what it's for, but few understand its subtle ramifications.

Several techniques exist for verifying the functionality of a digital circuit, ranging from the "suck-it-and-see" (that is, build-it-and-see-if-it-works) approach, through various flavors of logic simulation, to formal verification technology. To increase your delectation, logic simulation comes in several flavors, including event-driven, cycle-based, and a variety of "home-brewed" approaches.

Traditional logic simulation is conceptually simple. First, you describe the topology of the circuit, including the components you are using and the connections among them, along with a set of test vectors, or stimuli, to apply to the design. When you pass the circuit to the simulator, it accesses a predefined model library to determine the functionality and timing of each component and then constructs a virtual circuit in the computer's memory. The simulator then applies the test vectors to the circuit and reports the results. In the early 1970s, designers described the circuit as a textual netlist and represented the test vectors and the simulation results as files of tabular values. As computers became more powerful, graphical methods for capturing the schematic and stimuli and displaying the results superseded these time-consuming and error-prone techniques (Figure 1).

You can describe the circuit at many levels of abstraction, including flow charts, state diagrams, high-level hardware-description languages, and libraries of parameterized modules (Reference 1).
Also, you can represent the simulation models for the gate-level components in this article at different levels of abstraction.

Designers of early logic simulators often based them on the concept of "simulation primitives" (simple logic gates and registers) and had to represent any other models as collections of these primitives. Later simulators used a plethora of proprietary HDLs, but the industry has now largely standardized on two main "tongues," Verilog and VHDL, which are described, respectively, by the IEEE 1364 and 1076 standards (Figure 2).

Both Verilog and VHDL can describe circuits at different levels of abstraction—from primitive switches and gates to behavioral representations. (I dislike the term "behavioral," because you can use any level of abstraction to describe the behavior of a circuit. I prefer the term "algorithmic" to describe high-level representations, but "behavioral" has become the de facto standard.) In some respects, VHDL is more powerful than Verilog at the behavioral level; however, most of today's synthesis tools cannot accept anything more abstract than register-transfer-level descriptions, which tend to level the playing field between the two languages. Neither Verilog nor VHDL covers the complete switch and gate levels. Although both languages can represent the functionality of these primitive elements, the languages' success in modeling sophisticated timing effects varies. Also, neither language can completely handle the esoteric delay models required by deep-submicron technologies.

**Event-driven simulators**

The most common form of logic simulation is event-driven, in which the simulator sees the world as a series of discrete events. When an input value on a primitive gate changes, the simulator evaluates the gate to determine whether this change causes a change at the output and, if so, schedules an event for some future time (Figure 3).

Most event-driven logic simulators allow you to attach minimum, typical, and maximum delays to each model (Figure 4). When you run the simulator, you can select one of these delay modes, and the simulator uses that mode for all of the gates in the circuit. Also, some simulators allow you to select one delay mode as the default and then force certain gates to adopt another mode. For example, you might set all the gates in your datapath to use minimum delays and all the gates in your control path to use maximum delays, thereby allowing you to perform a "cheap and cheerful" timing analysis.

One problem facing the creators of simulation models is that delay specifications are becoming more complex over time. In the early 1970s, it was common for data books to specify identical delays for all of the input-to-output paths associated with a simple gate. However, over time, the books more accurately specified these delays, and each input-to-output path now typically has its own delay for both rising and falling transitions at the output (Figure 5).

Another problem facing designers is that each tool, such as those for simulation and synthesis, typically has its own model library, and these tools often return different delays. One thing that must happen in the not-so-distant future is for diverse tools to use common libraries. Another trend is that the timing and functionality portions of the models are becoming separate and distinct entities. I can envision a day when all of the tools use a common timing model that returns different levels of accuracy, depending on the information you feed it. Thus, in the prelayout part of the design cycle, the model would return delays at one level of accuracy, and the delays would become increasingly...
accurate as more information becomes available throughout the design process.

Early simulators could attach delays only to primitive elements. Thus, when you built a model, such as a multiplexer, you had to perform "distributed-delay" modeling, in which you distributed the delays over the primitive elements forming that model (Figure 6a). The problem was that data books specified delays only from the component’s inputs to its outputs, so the modeler had to fragment these delays and distribute portions of them throughout the model to achieve the correct total delays through each path. (The pioneer simulation-model writers were phenomenally good at solving simultaneous equations.) By comparison, modern simulators usually support pin-to-pin (pn-pn) delay specifications, which you can take straight from the data book and apply across the component's inputs and outputs (Figure 6b).

An argument exists that distributed-delay modeling handles narrow pulses and more closely simulates real designs than does pn-pn modeling. However, a model's contents usually bear only a passing resemblance to the internal structures of the physical device, so the way in which distributed-delay models handle narrow pulses is speculative at best.

Another interesting delay effect is that pulses can "shrink" or "stretch" as they pass through gates due to "unbalanced delays" on those gates (Figure 7). The rising delay is larger than the falling delay in this example. Therefore, a positive-going pulse applied to the input shrinks by the difference between the two delays. Similarly, a negative-going pulse applied to the input stretches by the difference. Also, "low-to-high" (LH) and "high-to-low" (HL) annotations apply to transitions at the output, so if the gate includes a negation (such as a NOT, NAND, or NOR), then the opposite effect occurs (Figure 8).

Now, consider how a simulator handles narrow pulses—those applied to the input that are narrower than the propagation delay of the gate. The first logic simulators targeted simple TTL devices at the board level. These devices, typically rejected narrow pulses, so the simulator represented them using an "inertial-delay" timing model (Figure 9). However, technologies such as ECL propagate pulses that are narrower than their propagation delays, as do devices such as delay lines. So, the next step was to allow the modeler to select between the inertial-delay model and a "transport-delay model" (Figure 10).

The problem with both the inertial- and transport-delay models is that they worked only in extreme cases. Over time, simulators began to use more sophisticated narrow-pulse-handling techniques, leading to the current state-of-the-art, three-band-delay model (Figure 11). With the three-band model, you can qualify each input-to-output delay with two values, r and p, which are percentages of the propagation delay for that path. If a pulse applied to the input is greater than or equal to p% of the propagation delay, then that pulse passes through the gate in a transport-delay mode. If the pulse is greater than or equal to r% of the propagation delay but less than p% of the delay, then that pulse appears at the output as an ambiguous, unknown value. Further, if the pulse is less than r% of the propagation delay, inertial-delay modeling rejects the pulse. Thus, the sophisticated three-band model allows the accurate modeling of any technology. Also, the three-band model provides backward compatibility with earlier modeling styles, because setting the r% and p% values to be 100% of the propagation delay results in a "pure-inertial" delay path. Similarly, setting the r% and
p% values to be 0% of the propagation delay results in a "pure-transport" delay path.

**Cycle-based simulators**

You can apply event-driven logic simulation using models with accurate timing data to almost every
digital design and get a good feel for what's going on in your circuit. However, one disadvantage of
event-driven simulation is that the simulation models can commandeer a lot of memory (several
thousand bytes per gate, in some cases) to hold the timing data. Also, these simulators may be
slower than you prefer. Thus, they may make it impossible to simulate a really large design in a
reasonable time.

One solution is to use a cycle-based simulator, such as SpeedSim from SpeedSim Inc (Westford,
MA). Cycle-based simulators are well-suited for evaluating classical synchronous designs, which
means that they cover a lot of ground ([Reference 2](#)). Consider a classical synchronous design as
comprising globs of combinatorial logic sandwiched between blocks of registers ([Figure 12](#)). Cycle-
based and event-driven simulators can conceptually accept the same netlist as input. However,
cycle-based simulators, such as SpeedSim, discard the timing information and convert the gate-level
netlists for the combinatorial blocks into "flat" Boolean equations. The end result is that a cycle-
based simulator uses memory efficiently and lets you quickly verify extremely large designs. You
sacrifice the ability to check timing, however, so you must perform some independent timing
verification, such as static-timing analysis.

However, the types of designs that are amenable to cycle-based simulation are also suitable for
static-timing analysis. One aspect of this form of simulation is its application to hardware/software
integration. In the not-so-distant past, I was working alongside a team designing a RISC hardware
accelerator. This project had two main facets: the hardware itself and the operating system. If you
verify a system of this complexity using an event-driven simulator, the best you can hope for is to
simulate a few tens of cycles of the system clock, which doesn't tell you much about the quality of
the system's software. Alternatively, you can build the board and then run the operating system, but
this approach pushes a large portion of the software debugging toward the end of the design cycle,
and some bugs will require hardware modifications, which can be an extremely expensive hobby.
For these reasons, it is preferable to debug the software alongside the hardware as early as possible
in the design cycle.

To solve this problem, the team spent a few weeks creating a relatively simple cycle-based simulator,
which allowed them to verify the entire system up to receiving the operating-system prompt on the
screen and executing simple commands. If we had attempted to do the same thing with an event-
driven simulator, we'd probably still be waiting for the results!

Home-brewed simulators can appear in a variety of different guises. Ed Smith, a programmer at
VeriBest Inc (Boulder, CO), provides an example of a simple home-brewed simulator, written in C, in
which the logic gates and registers are C functions. To create a circuit to simulate, you simply
declare a main function that calls and connects the primitive gates. For example, you could describe
and simulate the circuit for an 8-bit linear-feedback-shift register ([Reference 3](#)) as follows:

```c
#include 'models.c' #include 'sim.c' main() { /* circuit description */ xnor2 ("G1","q7","q1","w1");
    xnor2 ("G2","q7","q2","w2"); xnor2 ("G3","q7","q3","w3"); dff ("R0","clear","clock","q7","q0");
    dff ("R1","clear","clock","q0","q1"); dff ("R2","clear","clock","w1","q2");
    dff ("R3","clear","clock","w2","q3"); dff ("R4","clear","clock","w3","q4");
    dff ("R5","clear","clock","q4","q5"); dff ("R6","clear","clock","q5","q6");
    dff ("R7","clear","clock","q6","q7"); /* go simulate it */ simulate(stimulus, response); }
```
I've cut a few lines, such as those that determine the names of the stimulus and response files, for
the sake of brevity, but this portion is essentially all there is. The file "sim.c" contains the "simulate"
function, and the file "models.c" declares the "xnor2" and "dff" functions. You can download the
source code for this simulator, which occupies approximately 10 pg of C from
http://ro.com/~bebopbb. Root around under "free synthesis software and other interesting stuff."
Note, however, that we do not support this software, so you're on your own. Have fun with it, and let
me know what you think.

Clive "Max" Maxfield is a member of the technical staff at Intergraph Computer Systems
(Huntsville, AL, (800) 763-0242), where he gets to play with the company's high-
performance graphics workstations. In addition to numerous technical articles and papers,
Max is also the author of Bebop to the Boolean Boogie: An Unconventional Guide to
Electronics (ISBN 1-878707-22-1). To order, phone (800) 247-6553). You can reach
Maxfield via e-mail at crmaxfie@ingr.com.

References

1. Maxfield, Clive "Max," "LPMs: High-level design uses low-level techniques," EDN, May 9, 1996,
   pg 125.
2. Maxfield, Clive "Max," "To be or not to be asynchronous; that is the question," EDN, Dec 7, 1995,
   pg 157.