

This section exemplifies the reasons for the book's success besides key content: paragraph summaries that highlight, diagrams that support, samples that illustrate, headings that guide, and succinct paragraphs that deliver. As editor I supported Janick's instincts for adding these features. This excerpt is from Writing Testbenches: Functional Verification of HDL Models authored by Janick Bergeron (www.janick.bergeron.com).

CHAPTER 4 **BEHAVIORAL HARDWARE DESCRIPTION LANGUAGES**

A proper verification engineer must break the “RTL mindset” that most hardware engineers, out of necessity, have grown into. To efficiently accomplish the verification task, you must be well-versed in behavioral (i.e. non-synthesizeable and highly algorithmic) descriptions. To use the behavioral constructs of VHDL or Verilog reliably and correctly, it is necessary to understand the side effects of the simulation algorithm and the limitations of the language - and ways to circumvent them. This understanding was not required to successfully write RTL models.

BEHAVIORAL VERSUS RTL THINKING

Many guide-
lines help code
RTL models.

All experienced hardware design engineers are very comfortable with writing synthesizable models. They conform to a well-defined subset of the VHDL or Verilog languages and follow one of a few coding styles. Numerous RTL coding guidelines have been published¹. They help designers obtain efficient implementations: low area, high speed, or low power. Guidelines, such as the ones

-
1. The IEEE will soon publish a standard set of guidelines for RTL coding. For Verilog, see “*IEEE P1364.1 Standard for Verilog Register Transfer Level Synthesis*” prepared by the Verilog Synthesis Interoperability Working Group of the Design Automation Standards Committee. For VHDL, see “*IEEE P1076.6 Standard for VHDL Register Transfer Level Synthesis*” prepared by the VHDL Synthesis Interoperability Working Group of the Design Automation Standards Committee.

shown in Sample 4-1, can help a novice designer avoid undesirable hardware components, such as latches, internal buses, or tristate buffers. More importantly, guidelines can also help maintain an identical behavior between the synthesizable model and the gate-level implementation, such as the ones shown in Sample 4-2.

Sample 4-1.
RTL coding
guidelines to
avoid undesir-
able hardware
structures

1. To avoid latches, set all outputs of combinatorial blocks to default values at the beginning of the block.
2. To avoid internal buses, do not assign *regs* from two separate *always* blocks (Verilog only).
3. To avoid tristate buffers, do not assign the value 'Z' (VHDL) or 1'bz (Verilog).

Sample 4-2.
RTL coding
guidelines to
maintain simu-
lation behavior

1. All inputs must be listed in the sensitivity list of a combinatorial block.
2. The clock and asynchronous reset must be in the sensitivity list of a sequential block.
3. Use a non-blocking assignment when assigning to a *reg* intended to be inferred as a flip-flop (Verilog only).

The adherence to the synthesizable subset and proper coding guidelines can be easily verified using a linting tool. (More details are in the section titled "Linting Tools" on page 22.) After several months of experience, the subset becomes very natural to hardware designers. It matches their mental model of a hardware design: state machines, operators, multiplexers, decoders, latches, clocks, etc.

Do not use RTL-
like code when
writing test-
benches.

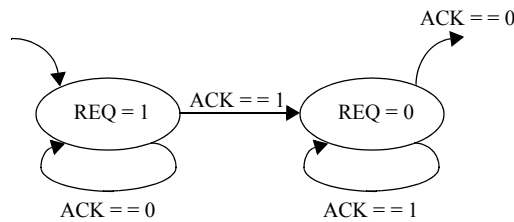
The synthesizable subset is adequate for describing the implementation of a particular design. I often claim that VHDL and Verilog are both equally poor at this task. The subset was dictated by the synthesis technology, not by someone with a warped sense of humor playing a practical joke on the entire industry. It was designed to describe hardware structures and logical transformations between registers, matching the capability of the logic synthesis technology.

However, this subset becomes quickly insufficient when writing testbenches that were never intended to be implemented in hardware. Both languages have a rich set of constructs and statements. If you have an “RTL mindset” when writing testbenches and limit yourself to using a coding style designed to describe relatively low-level hardware structures, you will not take full advantage of the language’s power. The verification task will be needlessly tedious and complicated.

Contrasting the Approaches

The example below shows a simple handshaking protocol. Your task is to write some VHDL or Verilog code that implements the simple handshaking protocol shown in Figure 4-1. It will detect that an acknowledge signal (ACK) has been asserted (high) after a requesting signal (REQ) has been asserted (high). Once the acknowledge is detected, the requesting signal must be deasserted, and then the requesting signal must wait for the acknowledge signal to be deasserted.

Figure 4-1.
State diagram
for
handshaking
protocol



RTL-Thinking Example. A hardware designer, with an RTL mindset, will immediately implement the state machine shown in Figure 4-1. The corresponding VHDL code is shown in Sample 4-3. This relatively simple behavior required 28 lines of code and two processes to describe, and two additional states in a potentially more complex state machine.

Sample 4-3.
Synthesizable VHDL code for simple handshaking protocol

```
type STATE_TYP is (... , MAKE_REQ, RELEASE, ...);
signal STATE, NEXT_STATE: STATE_TYP;

COMB: process (STATE, ACK)
begin
    NEXT_STATE <= STATE;
    case STATE is
        ...
        when MAKE_REQ =>
            REQ <= '1';
            if ACK = '1' then
                NEXT_STATE <= RELEASE;
            end if;
        when RELEASE =>
            REQ <= '0';
            if ACK = '0' then
                NEXT_STATE <= ...;
            end if;
        ...
    end case;
end process COMB;

SEQ: process (CLK)
begin
    if CLK'event and CLK = '1' then
        if RESET = '1' then
            STATE <= ...;
        else
            STATE <= NEXT_STATE;
        end if;
    end if;
end process SEQ;
```

Behavioral versus RTL Thinking

Focus on behavior, not implementation.

Behavioral-Thinking Example. A verification engineer, with a behavioral mindset, will instead focus on the *behavior* of the protocol, not its implementation as a state machine. The corresponding code is shown in Sample 4-4. The functionality can be described behaviorally using only four statements.

Sample 4-4.
Behavioral
VHDL code
for simple
handshaking
protocol

```
process
begin
    ...
    REQ <= '1';
    wait until ACK = '1';
    REQ <= '0';
    wait until ACK = '0';
    ...
end process;
```

Behavioral models are faster to write.

Modeling this simple protocol using behavioral constructs should require less than 10 percent of the time required to model it using synthesizable constructs. Not only is there less code to write (14 percent), but also it is simpler, requiring less effort to ensure that it is correct.

Behavioral models simulate faster.

Another benefit of behavioral modeling is the increase in simulation performance. Assuming that there is a long delay between a change in the request and the corresponding acknowledgement, the simulation of the synthesizable model would still execute the *SEQ* process at every transition of the clock (because that process is sensitive to the clock signal). The process containing the behavioral description would wait for the proper condition of the acknowledge signal, resuming execution only when the protocol is satisfied. If the acknowledge signal replies after a 10-clock-cycle delay, this represents a reduction of process execution from 40 in the synthesizable version to two in the behavioral one, or a 1900 percent increase in simulation performance.