

Doing Behavioral Design the Right Way Minimizes Verification

Kirk Ober, Forte Design Systems, San Jose, CA
March 3, 2004

Introduction

Behavioral synthesis has weathered a few bumps in the road on its way to acceptance. The concept was simple enough: write an untimed behavioral specification, and then direct a behavioral synthesis tool to generate a family of cycle-accurate RTL architectures. Using this process, called *exploration*, designers could choose which of these architectures best fit their needs and use it in their path to gates. But this simple concept proved difficult to implement.

The early tools available in this space had most of the classic behavioral synthesis features like loop unrolling and datapath scheduling, but did not make it easy to do meaningful design exploration. For starters, they only accepted behavioral specifications in Verilog or VHDL, not a high-level, system-oriented language like C++ (where algorithms are typically developed) with the SystemC class libraries. They produced reports that were hard to read, and the directives were defined in a mixture of scripting commands and deeply embedded pragmas. The only control you had over the I/O interaction was by specifying one of three scheduling modes (remember terms like *superstate* and *free-floating*?). These modes applied to the whole specification, limited your control over specific I/O ports, and, hence, limited your ability to thoroughly explore the design.

But the biggest shortfall came in the area of verifying exploration results. Using the scheduling modes described above affected the RTL architectures dramatically. They could give the output ports of your generated architectures different output timing or even change their order of execution. Simply put, designers could not use the same testbench to simulate their behavioral specification and all of their generated architectures. The amount of time spent in verification ballooned because designers had to write unique testbenches for each RTL

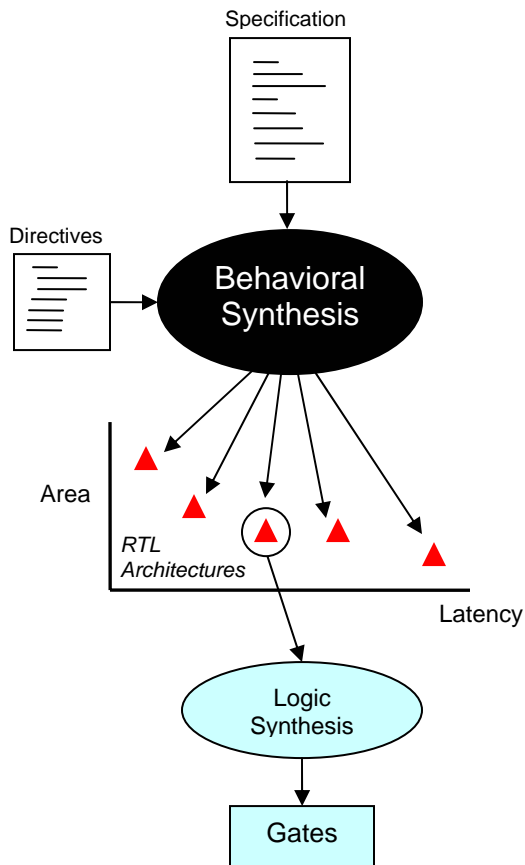


Figure 1 - Behavioral Exploration Flow

architecture. What’s worse, they were no longer really verifying anything — by using different tests they were not really proving the architectures had the same functionality as the original specification. The value of exploration diminishes considerably when designers cannot verify all of their generated architectures quickly and confidently.

This paper addresses ways to write your behavioral specification to achieve a more unified verification scheme. The goal is to use a single testbench before and after behavioral synthesis for all architectures.

Unified Verification Ingredients

Any behavioral specification must have the following to unify the verification process throughout exploration:

- *Handshaking* – Status signals for each data port tell the testbench when the design-under-test is ready to receive input data and when it has produced valid output data. This way the testbench can act on the status instead of needing any knowledge of what happens in specific cycles.
- *Protocol Blocks* – The sections of the specification where the I/O interaction is described must be marked as protocol blocks. These are, in effect, “don’t touch” blocks for the behavioral synthesis tool to preserve. This way I/O operations are the same in all of the generated RTL architectures. In addition to the single testbench argument made earlier, this ensures that the design will always be able

to interface properly with other chips in the system.

Inserting Handshaking

Using the designer’s directives, behavioral synthesis can create many architectures for a specification. One architecture may have a latency of one or two clock cycles with large area, while another might have a latency of ten clock cycles with a smaller area. Consider this simple SystemC specification, which does not use handshaking, and look at the impact it has on the design process:

```
SC_MODULE(cube) {
    sc_in< bool >      ck;
    sc_in< bool >      rst;
    sc_in< sc_uint<8> > in;
    sc_out< sc_uint<24> > out;

    void entry();

    sc_uint<8>      x;
    sc_uint<24> y;

    SC_CTOR(cube) {
        SC_CTHREAD(entry, ck.pos());
        watching(rst.delayed()==0);
    }
};

void
cube::entry()
{
    if (!reset) {
        out.write( 0 );
        wait(1);
    }

    while( true ) {
        x = in.read();
        y = x * x * x;
        out.write( y );
        wait(1);
    }
}
```

This specification reads the value of input port *in*, calculates its cube, and then writes that value to the output port *out*. The mathematic operation where the cube of the input value is

calculated, as well as the reading of inputs and writing of outputs, is available for a behavioral synthesis tool to schedule in a number of different ways. Figure 2 shows an exploration graph of several possible architectures that could be generated. In architecture A, the value on the output port is valid in one clock cycle; in architecture B, it takes two clock cycles to produce a valid output; and in architecture C, a very small area was achieved, but as a result it takes four cycles to produce a valid output.

At this point, you would want to verify that all of these generated architectures were functioning properly. To do so the designer would have to write three separate and distinct testbenches. Each would have to be written so that it expected the valid output in the correct clock cycle. Also, the testbench would have to know when the operation is complete so that it can force a new value on the input port at the correct time. Even for this simple example this is not a trivial matter. Designers would have to examine the behavioral synthesis report for each

architecture to determine the latency of the operation. For very large designs, this is neither practical nor efficient. As stated earlier, designers would be multiplying the amount of time spent in verification many times over.

The best approach is to instead introduce a handshaking scheme into the behavioral specification to indicate when inputs are ready for new data and when outputs have valid data. This way, one testbench can be written that will test the behavioral specification before synthesis, and all of the generated RTL architectures after synthesis.

A good handshaking scheme to use in behavioral synthesis is *ready/valid handshaking*. In ready/valid handshaking, two status ports are added for each data port in the algorithm. One is a *valid* signal to indicate when the port has valid data, and the other is a *ready* signal to indicate when the port is ready for new data. First, let's take a look at our example with ready/valid handshaking signals added in bold:

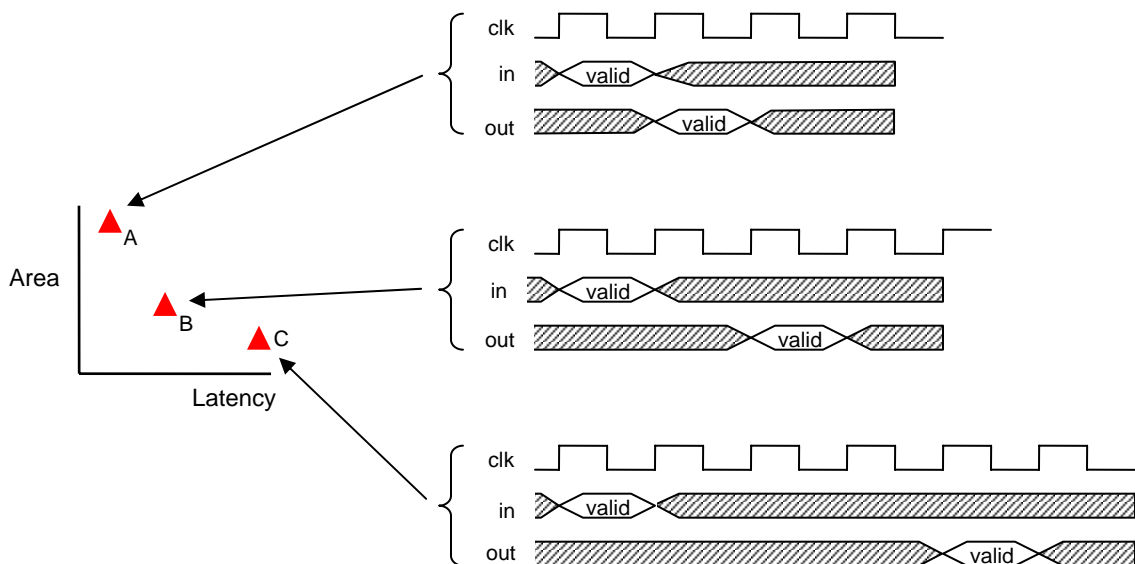


Figure 2 - Same Algorithm, Different Timing

```

SC_MODULE(cube) {
    sc_in< bool >      ck;
    sc_in< bool >      rst;
    sc_in< sc_uint<8> > in;
    sc_out< bool >     in_rdy;
    sc_in< bool >      in_vld;
    sc_out< sc_uint<24> > out;
    sc_in< bool >      out_rdy;
    sc_out< bool >     out_vld;

    void entry();

    sc_uint<8>      x;
    sc_uint<24> y;

    SC_CTOR(cube) {
        SC_CTHREAD(entry, ck.pos());
        watching(rst.delayed()==0);
    }
};

void
cube::entry()
{
    if (!rst) {
        out.write( 0 );
        out_vld = 0;
        in_rdy = 0;
        wait(1);
    }

    while( true ) {
        in_rdy = 1;
        do {
            wait(1);
        } while(!in_vld);

        x = in.read();
        in_rdy = 0;
        y = x * x * x;

        do {
            wait(1);
        } while(!out_rdy);
        out_vld = 1;
        out.write( y );
        wait(1);
        out_vld = 0;
    }
}

```

For input port *in*, ready signal *in_rdy* and valid signal *in_vld* were added. Ready signal *out_rdy* and valid signal *out_vld* were added for output port *out*.

When the specification begins its operation (at the start of the while loop), it begins by asserting the *in_rdy* signal. This tells the testbench that the specification is ready to start receiving valid input data. Then, by using a do/while loop, the specification waits until the testbench asserts the *in_vld* signal, indicating that the value presently on the input port is valid. The specification then reads the input, deasserts *in_rdy* to tell the testbench that it is busy processing the data and not ready to accept any new input data, and starts its cubing calculation.

When the specification is prepared to write the result of the calculation to the output port, it first makes sure that the testbench is ready to receive that result: a do/while loop is used to wait until the testbench asserts *out_rdy*. The specification then writes the result to the output port and asserts *out_vld* to tell the testbench that valid output data is present. It waits one clock cycle and then deasserts *out_vld*, ready to start another iteration of the while loop.

Obviously there is code inside the testbench that waits and acts on these handshaking signals being asserted by the specification. Figure 3 shows some timing diagrams for the handshaking interaction on each port.

Because of the ready/valid handshake, it doesn't matter how much time the internal circuitry of a given architecture takes to process data. A single testbench knows how to work on any architecture with any timing.

Establishing Protocol

There's one remaining area that must be nailed down in order to make the exploration of a behavioral algorithm reliable. Unified verification literally falls apart without properly

defined protocol blocks in the design specification. Remember that a behavioral synthesis tool will try to explore *every* part of the design by default, including the timing of input reads and output writes. This presents two problems. First, no single testbench can be written that can handle, say, one architecture where the input is read in one cycle and another where the input is read in two cycles. Second, once the architecture is synthesized to gates, there is no guarantee that it will interface

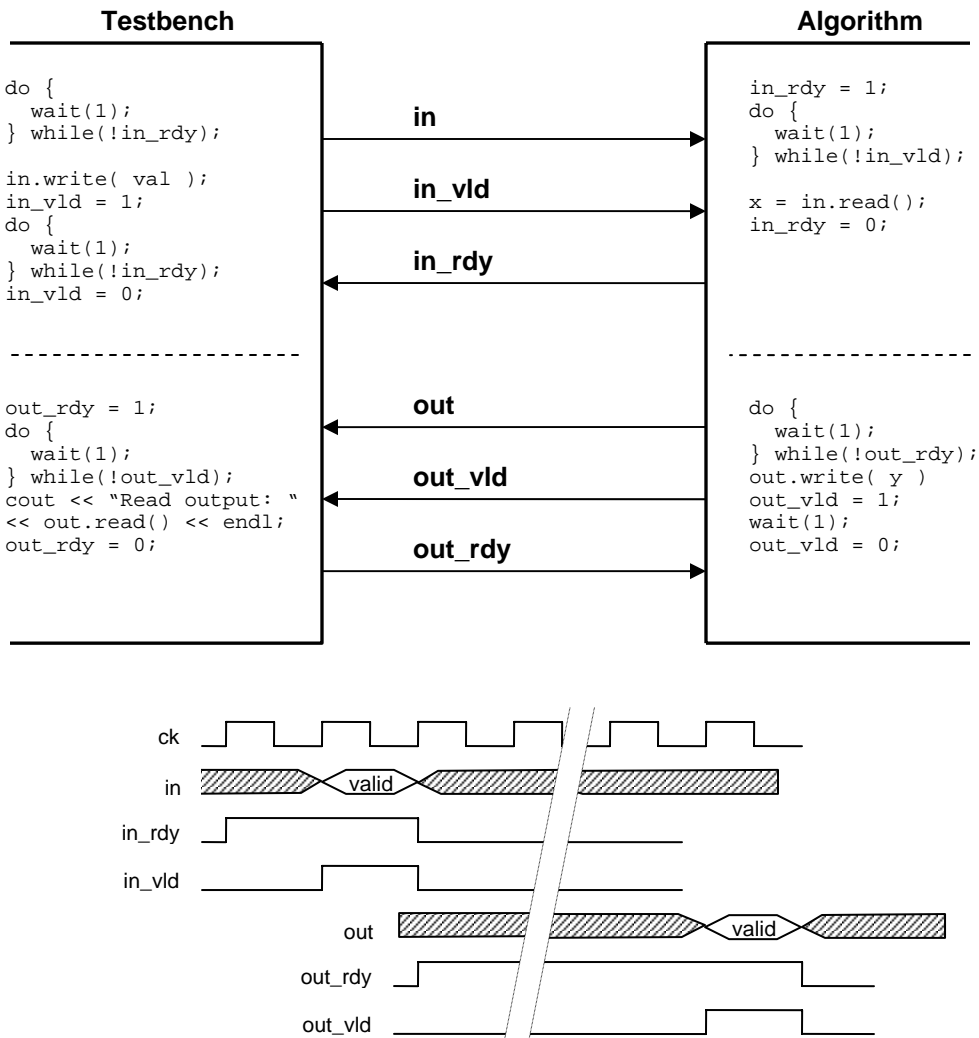


Figure 3 – Handshaking Interaction

properly with other chips in the system, which are going to expect only one style of protocol.

In our example, the code was written so that the input and output would be read and written in one clock cycle. It is possible that, if left unprotected, the behavioral synthesis tool could generate an architecture where the reading of the input took more than one clock cycle. Even with good handshaking, a single testbench could not do the job. The designer would have to have a testbench for architectures where the input needed to be valid for one clock cycle, and another testbench for architectures where the input needed to be valid for multiple cycles. The way to avoid this situation is to define protocol blocks — sections of your code that the behavioral synthesis tool will leave alone and will implement exactly as they are written.

In the few behavioral synthesis tools that support them, protocol blocks can be defined using inline directives. Each `wait()` statement inside a protocol block will translate to exactly one clock cycle in every generated architecture. Here's how the example might look with protocol blocks defined. **Note: each behavioral synthesis tool will have its own unique directives for defining protocol blocks. For instructional purposes, and to keep the code non-specific to a particular tool, this example uses generic //protocol begin/end pairs to signify protocol blocks:*

```
SC_MODULE(cube) {
    sc_in< bool >      ck;
    sc_in< bool >      rst;
    sc_in< sc_uint<8> > in;
    sc_out< bool >     in_rdy;
    sc_in< bool >      in_vld;
    sc_out< sc_uint<24> > out;
    sc_in< bool >      out_rdy;
```

```
    out_vld;

    void entry();

    sc_uint<8>      x;
    sc_uint<24> y;

    SC_CTOR(cube) {
        SC_CTHREAD(entry, ck.pos());
        watching(rst.delayed()==0);
    }
};

void
cube::entry()
{
    if (!rst) {
        // protocol begin
        out.write( 0 );
        out_vld = 0;
        in_rdy = 0;
        wait(1);
        // protocol end
    }

    while( true ) {
        // protocol begin
        in_rdy = 1;
        do {
            wait(1);
        } while(!in_vld);
        x = in.read();
        in_rdy = 0;
        // protocol end

        // This line free to be
        // scheduled
        y = x * x * x;

        // protocol begin
        do {
            wait(1);
        } while(!out_rdy);
        out_vld = 1;
        out.write( y );
        wait(1);
        out_vld = 0;
        // protocol end
    }
}
```

Anywhere an input is read or an output is written (along with any `wait()` statements that are involved) is placed inside a protocol block. The behavioral synthesis tool will preserve what's in the protocol blocks. The only part available for the tool to schedule is the multiplication operation where the cube of the input value is

calculated. The input and output interaction remains consistent.

Conclusion

The substance of the message here is to understand what parts of a behavioral specification should be scheduled and what parts should be left alone. By preserving the I/O protocol, exploration can be done quickly. After a family of RTL architectures has been generated, they can all be verified using a single testbench. This unified verification style lends itself to automation. A simple Makefile could be written to run the testbench on all the architectures and even compare the valid output values to some golden results. This saves verification engineers from having to write unique testbenches that expect values in specific cycles, and from spending unnecessary time evaluating whether the results are functionally the same as those for other architectures.

Protocol preservation also ensures that the design will maintain a consistent interface with other chips in the system, which may have been designed using conventional RTL synthesis and therefore will only work with one kind of interface.

Kirk Ober is a Senior Field Applications Engineer at Forte Design Systems. He has nine years experience in EDA, previously holding similar positions at Summit Design and Synopsys.