Module 20

Coding Design Behavior Algorithmically

cadence

This module just briefly introduces testbenches and focuses primarily on the difference between behavioral and RTL modeling. It uses a bus interface controller model to illustrate these differences.

Module Objective

In this module, you:

- Model a design behavior algorithmically and are introduced to a testbench concept

Topics

- Comparing Behavioral and RTL modeling
- Example Bus Interface Controller Model
- Testbench Introduction

Your objective is to model a design at the behavioral level of abstraction and at the register-transfer level of abstraction. To do this, it is probably beneficial to first look at some simple examples.

Verilog Supports Multiple Levels of Abstraction

Most examples herein are coded at the Register Transfer Level (RTL).

- Coded in terms of registers values at each clock cycles.
- People tend to think of RTL as the “synthesis subset”:
  - Using only the constructs and templates accepted by synthesis.

```
always @(posedge clk)
 begin
  d_out <= d_tmp;
  d_tmp <= d_in;
 end
```

Verilog also supports the *behavioral* level of abstraction.

- No regard for an actual hardware implementation.
  - All constructs usable.
- Useful for algorithmic or architectural exploration and for testbench.

```
always @(d_in)
 d_out <= repeat(2)
  @(posedge clk)
   d_in;
```


Almost all examples you have seen here up to this point are coded at the Register Transfer Level (RTL). That means that the behavior is coded in terms of registers and the values they have at any particular clock cycle. People tend to think of RTL as the synthesis subset, but though the constructs and templates accepted by synthesis are definitely RT-level constructs, RTL coding can include non-synthesizable constructs.

Verilog also supports the behavioral level of abstraction, in which you code behavior with no regard for an actual hardware implementation, so can utilize any Verilog construct. The behavioral level of abstraction is useful for exploring design architecture, and especially useful for developing testbenches.

Comparing Behavioral and RTL Modeling

RTL Modeling

- Defines model architecture
- Clocks generally required
  - Activity synchronized to clock edge
  - Models timing with clock cycles
- Data stored in “registers”
- Contains code constructs and style supported by synthesis standard
- Used for all code to be synthesized

Behavioral Modeling

- Defines model behavior
- Clocks generally omitted
  - Activity synchronized to any event
  - Models timing with delay control
- Data stored in any manner
- Contains any Verilog construct
- Used for system-level modeling and for testbench

RTL does not necessarily mean the “synthesis subset” but many people think of it that way.

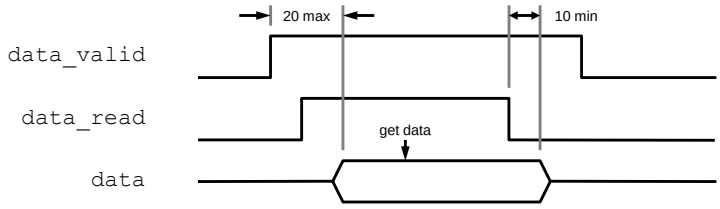
cadence

Verilog supports multiple levels of abstraction, from a stochastic system-level model for architectural exploration down to a switch-level model. People most often use the loosely defined behavioral level or the better-defined register-transfer level.

- The behavioral level defines only the model behavior. It can define the behavior totally algorithmically, it can “step” the algorithm by a series of ordered events, and it can provide inter-event delays for visualization purposes and for latency and throughput calculations. The behavioral level can store data in any manner and use any Verilog construct. Your testbench is an example of a behavioral model.
- The register-transfer level (RTL) defines the model in terms of sequential storage units referred to as registers and the synchronized transfer of data between the registers. It defines clocks, variables to represent the storage elements, and processes that react to active edges of the clock.

Since the advent of logic synthesis, the RT level has come to commonly mean the usage of an HDL in a structured manner from which a logic synthesis tool can infer combinational and sequential logic. Hence, the standards defining the synthesizable subset of the HDL have become de facto definitions of the RT level of abstraction.

Example Bus Interface Controller Model



- Data available not more than 20 after `data_valid` asserts
- Data retained not less than 10 after `data_read` de-asserts

```
// Behavioral model
always
  begin
    wait (data_valid);
    data_read = 1;
    #20;
    local_buffer = data;
    data_read = 0;
    #10;
    wait (~data_valid);
  end
```

- Defines model behavior
- Clocks generally omitted
- Data stored in any manner
- Contains any Verilog construct

388

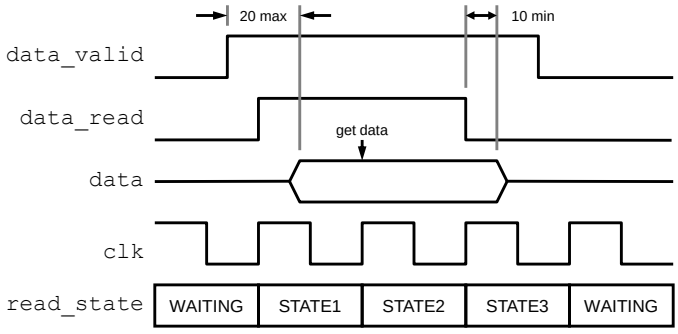
cadence

The waveform diagram depicts the Bus Interface Controller specification:

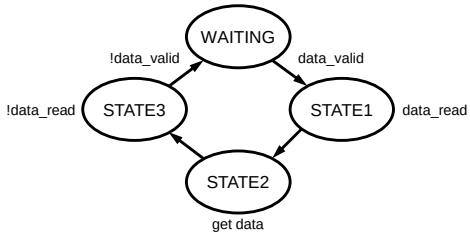
- The data_valid input indicates that new data is available.
- The data_read output indicates the model is ready to receive the data.
- The data input appears no later than 20ns after data_valid asserts.
- The model can drop the data_read output any time after capturing the data.
- The data_valid input drops to indicate the end of the transaction.

The behavioral model implements this behavior without regard to clocks.

Bus Interface Controller Implementation



Clock period not less than 20



The RTL model synchronizes its behavior to clock edges and defines a state machine to step through the behavior. The state machine steps through four states, stepping on the positive clock edge:

- The WAITING state waits for the data_valid input and transitions to STATE1.
- The STATE1 state asserts the data_read output and transitions to STATE2.
- The STATE2 state captures the data input and transitions to STATE3.
- The STATE3 state drops the data_read output and transitions back to WAITING.

RTL Procedures Review

Sequential Logic

- Storage is implied.
- Outputs are a function of current inputs and storage variables.
- Procedure is sensitive to clock edge and optionally to reset edge.
- Nonblocking assignments.

```
always @*
  if (sel == 0)
    y = a;
  else if (sel <= 5)
    y = b;
  else
    y = c;
```

Combinational Logic

- Storage is not implied.
- Outputs are a function of current inputs.
- Procedure is sensitive to all inputs.
- Blocking assignments.

```
always @(posedge clk)
  if (rst || count==9)
    count <= 0;
  else
    count <= count + 1;
```


The definition of combinational logic is that if you account for propagation delay then you know what the outputs are based solely upon the state of the inputs, i.e., it has no internal storage. This requires that the simulator evaluate the inputs and update the outputs upon any transition of any input. You can guarantee this by doing two things:

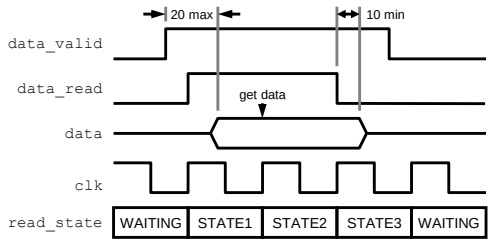
- You form a list of event expressions containing all nets and variables that the procedural block reads, or use the Verilog 2001 implicit event expression list wildcard (*) character.
- You ensure that each execution of the procedural block assigns a value to all variables that the block writes.

The left combinational block does both. It uses the implicit event expression list, and with the last else branch ensures that the d variable gets a value if it did not already get one. Note that this block uses blocking assignments.

The definition of sequential logic is that it must somehow imply storage. If it evaluates the inputs upon any transition of any input, but fails to update some output for some combination of inputs, then it implies storage for that output to hold its value until it is next assigned. It implies latch storage because it is not sensitive to any particular input edge. If the block evaluates its inputs and updates its outputs only upon a particular input edge, then it implies storage to hold all outputs until that next particular input edge. It implies flip-flop storage because it is sensitive to a particular input edge.

The right sequential block implies flip-flop storage because it updates its outputs only upon a particular input edge. Note that this block uses nonblocking assignments to prevent clock/data races.

Bus Interface Controller RTL




```
// FSM
always @(posedge clk or posedge reset)
  if (reset == 1)
    read_state <= WAITING;
  else
    case (read_state)
      WAITING: if (data_valid)
                  read_state <= STATE1;
      STATE1 : read_state <= STATE2;
      STATE2 : read_state <= STATE3;
      STATE3 : if (!data_valid)
                  read_state <= WAITING;
    endcase
```

```
// Constants and variables
localparam integer WAITING = 0,
                   STATE1  = 1,
                   STATE2  = 2,
                   STATE3  = 3;
reg [7:0] local_buffer;
reg [1:0] read_state;
reg       data_read;
```

```
// Data store
always @(posedge clk)
  if (read_state == STATE1)
    local_buffer <= data;
```

```
// Output decode
always @(read_state)
  data_read = read_state == STATE1
            || read_state == STATE2;
```

391

cadence

To code the RT-level model, we focus separately upon its four features:

- Firstly, we declare the model's constants and variables. We declare constants for the FSM states, a variable to hold the captured data, a variable to hold the current state, and a variable to hold the value of the data_read output.
- Second, we define the FSM. We make a sequential procedural block sensitive to the positive clock edge and sensitive to the positive edge of an asynchronous reset. The reset sets the state to WAITING, and then in a case statement, we step through the states, each step dependent on the current state and some steps depending also on the value of an input.
- Third, we define a sequential procedural block to capture incoming data upon the active clock edge that occurs while in the STATE1 state. We could have merged this feature into the FSM block but chose to not do so because it is not really part of the FSM.
- Fourth, we define a combinational block to generate the data_read output as a function of the current state. We could also have merged this feature into the FSM block but chose to not do so because it also is not really part of the FSM.

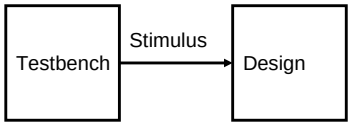
Testbench Introduction

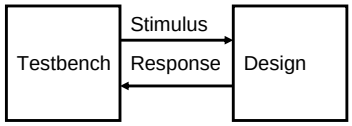
Sophisticated Testbench

- Models operational environment
- Complex randomized stimulus
- Interacts with design
- Response checking during simulation

Simple Testbench

- Just stimulates the design
- Simple stimulus generation
- No interaction with design
- Response checking is a post-simulation activity





392

cadence

The majority of the project development effort is typically focused upon design verification.

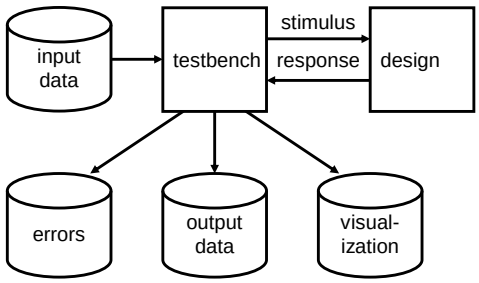
Testbenches tend to start off simple but quickly become sophisticated.

- A simple testbench might declare nets and variables, connect them to a DUT instance, provide stimulus to the DUT, and dump the stimulus and response to a file for later analysis. A simple testbench is appropriate for a small simple test case and not appropriate for a large complex design.
- A sophisticated testbench will at least check the DUT response spontaneously. It would ideally dynamically react to the DUT response, that is, communicate with the DUT. A truly powerful testbench would as much as practical mimic the expected DUT operating environment.

Testbenches and Files

Verilog can read and write external files:

- Input data generated by some other tool.
- Output data to be analyzed later.



Your testbench can read data from a file, for example a file of test patterns that some other program or tool such as a system modeling tool created.

Your testbench can write data to a file, for example:

- A record of what was tested and what failed.
- Stimulus and response data for later analysis.
- Pattern (image) data for a visualization application.

Module Summary

Now you can model design behavior algorithmically.

In this module, you learned to:

- Describe behavioral modeling, which is used for system-level modeling and for testbenches. The behavioral model stores the data in any manner, uses any Verilog construct, and generally omits clocks.
- Describe RTL modeling architecture, which is used for all code that you are going to synthesize. The RTL model includes the required clocks, data stored in registers, code constructs, and style supported by the synthesis standard.
- Write simple and sophisticated testbenches.

cadence

You can now model a design both behaviorally and at the register-transfer level. This module just briefly introduced testbenches and focused primarily on the difference between behavioral and RTL modeling. It used a bus interface controller model to illustrate these differences.

Module Review

1. What is behavioral modeling used for?
2. What is RTL modeling used for?
3. Explain the difference between the procedures that represent combinational logic and the procedures that represent sequential logic.
4. Suggest a way to define the constant values that represent FSM states.

cadence

This page does not contain notes.

Module Review Solutions

1. What is behavioral modeling used for?
   - Behavioral models are not restricted to synthesis-supported constructs, so can use any construct to model a system and to code a testbench.
2. What is RTL modeling used for?
   - The register-transfer level of abstraction has come to mean the use of synthesis-supported templates to represent digital hardware.
3. Explain the difference between the procedures that represent combinational logic and the procedures that represent sequential logic.
   - The outputs of combinational logic are a function purely of the current inputs, with no implied storage. The procedure is sensitive to all inputs and typically uses blocking assignments.
   - The outputs of sequential logic are a function of at least one storage variable and potentially also inputs. The procedure is sensitive to a clock edge and optionally sensitive to a reset edge, and must use nonblocking assignments for the storage variables.
4. Suggest a way to define the constant values that represent FSM states.
   - The most prevalent and a synthesis-supported way to define FSM state values is to use the parameter or localparam constructs.

cadence

This page does not contain notes.

Labs

There are no labs in this module.

cadence

This page does not contain notes.