ASIC Verification

SystemVerilog Assertions 1

Fall 2011
Meeta Yadav
Topics

- Introduction to assertions
- Types of assertions
- Properties
- Importance of SVA
- FIFO Example
Introduction to Assertions

What is an assertion?

Assertions are not really a new concept: In simple words they define behavior

Sample Assertion

A parent may tell a child
“there are 10 cookies in the cookie jar and during the day, if you are hungry, you may take at most one cookie! At the end of the day I want to see at least 9 cookies in the jar and ... and I’ll count the number of cookies every hour”
Introduction to Assertions

The parents statement translates into two properties

Property 1: “if hungry and there are more than 9 cookies then you must eat a cookie”. If a cookie is not eaten by the next hour clock, then it is an error.

Property 2: There should never be less than 9 cookies in the jar

Flowchart:
- If hungry and cookie count > 9, eat a cookie. Otherwise, don’t eat a cookie.
- If cookie count < 9, punish kids. Otherwise, continue as before.
Introduction to Assertions

Property 1: “if hungry and there are more than 9 cookies then you must eat a cookie”. If a cookie is not eaten by the next hour clock, then it is an error.

```vhls
property pGetAcookie;
@(posedge hour_clk) hungry&&cookie_count>9 |=> eat_1_cookie;
endproperty

apGetAcookie: assert property (pGetAcookie)
```

How do we express this behavior?
Property 2: There should never be less than 9 cookies in the jar.

How do we express this behavior?

```verilog
property pCookiesInJar;
@ (posedge hour_clk) not (cookie_count<9);
endproperty: pCookiesInJar
apCookiesInJar : assert property (pCookiesInJar) else punish_the_kid();
```
Introduction to Assertions

What is an assertion?

An assertion is a “statement of fact” or claim of truth made about the design by the design or verification engineer. If the claim is wrong then the assertion fails.

Representing Assertions

SystemVerilog assertions are represented as a property that defines intent and the property is enforced by using the assert statement.
What is an Assertion?

◆ An assertion is a statement that a certain property must be true
◆ Assertions are used to:
  ▪ Document the functionality of the design
  ▪ Check that the intent of the designer is met over simulation time
  ▪ Determine if verification tested the design (coverage)
◆ Assertions can be specified by
  ▪ The design engineer
  ▪ Verification engineer

If request has gone high then acknowledge should be asserted within 1 to 3 cycles

```
property pReqAck;
  (@posedge clk) request |-> ## [1:3] acknowledge;
endproperty: pReqAck
apReqAck : assert property (pReqAck)
```
Advantages of SystemVerilog Assertions

- Concise Syntax
- Ignored by Synthesis
- Can be disabled!
- Can have severity levels
- Perform Coverage
Types of Assertions

Types of assertions in SystemVerilog

1. Immediate assertions test for a condition at current time

   ```verilog
   always @(state)
   assert (state=$onehot) else $fatal;
   ```

   Generate a fatal error state variable is not a one-hot value

2. Concurrent assertions test for a sequence of events over multiple clock cycles

   ```verilog
   property pReqAck;
   (@(posedge clk) request |-> ## [1:3] acknowledge;)
   endproperty: pReqAck
   apReqAck : assert property (pReqAck)
   ```

A complex sequence can be defined very concisely
Immediate Assertions

- Executed once
- Usually placed inline with the code
- Looks a lot like if then else statements
- Not very useful
Immediate Assertions

- Immediate Assertions: An immediate assertion is a test of an expression the moment the statement is executed.

```
[name:] assert(expression)[pass_statement] [else fail_statement]
```

Syntax for Immediate Assertions

- May be used in initial and always procedures, tasks and functions.
- Performs a boolean true/false test:
  - If true execute the pass statement.
  - If false execute the fail statement.
- Evaluates the test at the instant the assert statement is executed.
Immediate Assertions

The statement associated with the success of the assert statement is the first statement, it is called the pass statement and is executed if the expression evaluates to true.

The action_block specifies what actions are taken upon success or failure of the assertion. The action block is executed immediately after the evaluation of the assert expression.

The statement associated with else is called a fail statement and is executed if the expression evaluates to false. The else statement can also be omitted.
Concurrent Assertions

- A concurrent assertion can test for a sequence of events spread over multiple clock cycles

```vhdl
base_rule1: assert property (property_specification) pass_statement else fail_statement;
```

**Syntax of concurrent assertions**

- `property_specification` describes a sequence of events
- Can be specified in always block, in initial blocks or stand-alone (like continuous assignments)

```vhdl
always @(posedge clock)
    if(state==FETCH)
        ap_req_gnt: assert property (p_req_gnt) passed_count++ else $fatal;
    property p_req_gnt;
    @(posedge clock) request ##3 grant ##1 !request ##1 !grant;
endproperty: p_req_gnt;
```
Assertion Actions and Messages

- The pass and fail statements can be any procedural statement
  - Print messages
  - Increment counters
  - Specify severity levels
- The pass statement is optional
  - If left off, then no action is taken when the assertion passes
- The fail statement is optional
  - The default is a tool-generated error message

```verilog
always @(negedge reset)
  fsm_reset_state: assert (state==FETCH) else $warning;
```

*Example of severity of assertion failure*
Assertion Severity Levels

◆ Since the assertion is a statement that something must be true, the failure of an assertion shall have a *severity* associated with it. By default, the severity of an assertion failure is *error*. Other severity levels can be specified by including one of the following severity system tasks in the fail statement:

- **$fatal**
  - Terminates execution of the tool
- **$error**
  - A run time error severity, tool continues execution
- **$warning**
  - Is a run-time warning, software continues execution
- **$info**
  - No severity, just print the message

```always @ (negedge reset)
  assert (state==FETCH) else $warning;
```

*Example of severity of assertion failure*
Overview of Properties and Assertions

- **Antecedent/Consequent**: To achieve the cause-effect of threads, SV introduces the concept of antecedent and consequent, with the two operators:
  - $|->$: Overlapped: The operator means that the resulting sequence starts in the same cycle as the causing sequence.
  - $|=>$: Non-overlapped: The operator means that the resulting sequence starts in the next cycle as the causing sequence.

```verilog
property p_handshake;
  @(posedge clk)
  request $\Rightarrow$ acknowledge ###1 data_enable ###1 done;
endproperty

aphandshake: assert property (p_handshake);
```

Property $p_{handshake}$ states that if the antecedent sequence $request$ is TRUE then the consequent must also be TRUE otherwise the property fails.
Overview of Properties and Assertions

```verilog
property p_handshake;
 @(posedge clk)
 request |=> acknowledge ##1 data_enable ##1 done;
endproperty

aphandshake: assert property (p_handshake);
```

Two active threads T1 and T2 for the handshake property
SystemVerilog Assertions in Verification Strategy

• Is the use of assertions a good verification strategy?
  ◆ Captures Designer Intent
  ◆ Allows protocols to be defined and verified
  ◆ Reduces time to market
  ◆ Greatly simplifies the verification of reusable IP
  ◆ Facilitates functional coverage metrics

YES!!!
SystemVerilog Assertions in Verification Strategy

- Assertions help capture designer intent (cont…)
  - Provide added visibility to the design and white box testing into its internal states

Types of assertions in a design

- Allows protocols to be defined and verified
  - By specifying temporal and combinational properties
  - Such assertions define the value of signals and the relationship between them
SystemVerilog Assertions in Verification Strategy

- Assertions reduce the time to market
  - ABV provides early detection of bugs
  - ABV facilitates debugging of failing tests as assertions can behave as monitors close to the source of errors

Design Errors vs time to market using ABV and FV
SystemVerilog Assertions in Verification Strategy

◆ Greatly simplifies the verification of reusable IP
  ▶ Previously defined assertions that express the input constraints and interface protocols of the IP block can be reused to help ensure proper integration
  ▶ Assertions can help ensure proper integration of the IP block within a larger system
  ▶ Typically, the verification of SoCs built using reusable IPs involves verifying the interfaces rather than the individual block’s functionality

◆ Facilitates functional coverage metrics
  ▶ Functional coverage is a measure of the how much functionality was exercised by verification
  ▶ Functional coverage metrics can be specified using SystemVerilog Assertions through the definition of user-defined coverage points using the `cover` directive
  ▶ For instance assertions that monitor the logic controlling a FIFO queue can be used to capture if the FIFO was ever full and whether extreme test corner cases were exercised
 Assertions Example: FIFO

1. When the FIFO is reset, the FIFO empty flag should be set and the full flag, wptr (write pointer), rptr (read pointer) and cnt (word counter) should all be cleared.
2. If the word counter (cnt) is greater than 15, the FIFO is full.
3. If the word counter (cnt) is less than 16, the FIFO is not full.
4. If the word counter is 15 and there is a write operation without a simultaneous read operation, the FIFO should go full.
5. If the FIFO is full, and there is a write operation without a simultaneous read operation, the full flag should not change.
6. If the FIFO is full, and there is a write operation without a simultaneous read operation, the write pointer should not change.
1. When the FIFO is reset, the FIFO empty flag should be set and the full flag, wp(tr (write pointer), rp(tr (read pointer) and cnt (word counter) should all be cleared.

```verilog
property reset_rptr0_wptr0_empty1_full0_cnt0;
 @(posedge clk)
   (rst |-> (rptr==0 && wptr==0 && empty==1 && full==0 && cnt==0));
endproperty
```
2. If the word counter (cnt) is greater than 15, the FIFO is full.

```vhdl
property full_fifo_condition;
    @(posedge clk) disable iff (rst)
    (cnt>15 |-> full);
endproperty
```
3. If the word counter (cnt) is less than 16, the FIFO is not full.

```vldb
property full_fifo_condition;
  @(posedge clk) disable iff (rst)
  (cnt<16 |-> !full);
endproperty
```
4. If the word counter is 15 and there is a write operation without a simultaneous read operation, the FIFO should go full.

```vhdl
property fifo_should_go_full;
@ (posedge clk) disable iff (!rst_n)
(cnt==15 && write && !read |=> full);
Endproperty
```
5. If the FIFO is full, and there is a write operation without a simultaneous read operation, the full flag should not change.

```
property full_write_full;
@ (posedge clk) disable iff (!rst_n)
(full && write && !read |=> full);
Endproperty
```
6. If the FIFO is full, and there is a write operation without a simultaneous read operation, the write pointer should not change.

```verilog
property full_write_wptr_no_change;
@ (posedge clk) disable iff (!rst_n)
(full && write && !read |=> $stable(wptr));
endproperty
```
Thank You