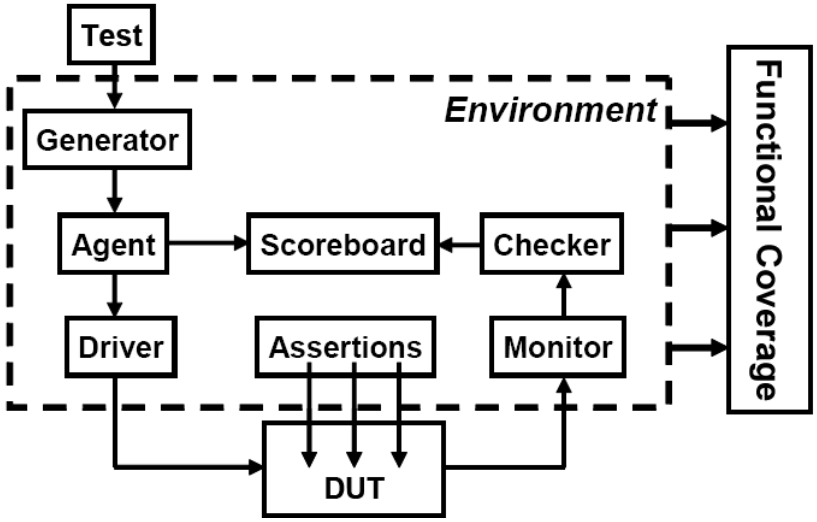
Лекция 11



M1:для последовательности случайно выбранных конфигураций

Выполнить // сценарный слой

M2: для последовательности случайно выбранных функций

Выполнить // функциональный слой

M3: для последовательности случайных транзакции

Выполнить // командный и сигнальный слои

Сформировать ожидаемый результат;

запустить тест;

сохранить результат и сравнить с ожидаемым;

контроль покрытия;

конец цикла M3;

контроль покрытия;

заполнение таблицы результатов и ее анализ ;

конец цикла M2;

контроль покрытия;

модифицировать список конфигураций;

конец цикла M1;

Ассерция это высказывание системного уровня, предназначенное для раннего обнаружения отклонения проекта от требований спецификации [36,47]. Если при выполнении действия заданного ассерцией утверждение, определенное в ней, не доказано, компилятор или симулятор выдают сообщение об ошибке. Ассерции можно разделить на три класса: мгновенные (immediate), параллельные и внешние. Мгновенные и последовательные вставляются в текст программ описания модулей. Последовательные

ассерции воспроизводятся при выполнении сеансов моделирования в соответствии с их локализацией в тексте программы. Параллельные ассерции также могут активизироваться при моделировании, но их главный смысл - проверка корректности описания на этапе компиляции и синтеза. Внешние ассерции встраиваются в TSTB и контролируют состояния портов проектируемого устройства.

Ассерции - блоки, добавляемые в исходный код проекта для наблюдения и управления поведением модели проекта [36, 47, 56] Ассерции создаются разработчиком или могут быть взяты из существующих библиотек для проверки типовых функций. Например, фирма Synopsys поставляет систему моделирования VCS с библиотекой SystemVerilog-ассерций.

Ассерции могут быть представлены операторами **if** для сообщений об ошибках, проявляющихся в процессе тестирования. Однако использование обычного условного выражения имеет ряд недостатков в сравнении с использованием специально введенной в SystemVerilog конструкции **assert**.

Во-первых, конструкция **assert** позволяет более компактно и явно представлять проверяемые опции и совмещать их с текстом, выводимым при обнаружении отклонений структуры или поведения от предполагаемых. Во-вторых, высказывание не входит в синтезируемое подмножество языка и соответствующая конструкция попросту игнорируется синтезатором, в то время как условные выражения, вводимые исключительно для контроля, необходимо перед синтезом вручную удалять из текста.

И, наконец, конструкция **assert** обеспечивает гибкое управление включением или блокировками некоторых проверок. В сеансах моделирования можно устанавливать дополнительные опции для разрешения или блокировки всех или выделенных ассерций (например, опция **ignore assertions** в системе моделирования QuestaSim).

Ассерции включаются в текст TSTB или ТсУ аналогично другим программным блокам (модулям, программам и т.п.) и активны на протяжении всего моделирования. Симулятор отслеживает, какие утверждения вызывались; при этом возможен сбор информации о функциональном покрытии этих конструкций. Как и другие процедурные операторы **процедурные ассерции** исполняются после исполнения предыдущего выражения в процедурном блоке always или initial, в котором они записаны. Инициализации **параллельной ассерции** выполняется по импульсу объявленного в ее декларации сигнала. Ассерция срабатывает в случае обнаружения ложности выражения.

Примеры 4.33 и 4.34 иллюстрируют два способа записи простейшей **процедурной ассерции** – с выражением **if**, и с использованием **assert**. Можно видеть, что текстуально конструкция **asser**t короче.

Пример 4.33. Контроль сигнала условным выражением

```
always ...
    bus.cb.request <= 1;
    @bus.cb;
    if (bus.cb.grant != 2'b01)
    $display("Error, grant != 1");
    // продолжение теста
```

Пример 4.34. Простая процедурная ассерция

```
always . . .
     bus.cb.request <= 1;
    @bus.cb;
    a1: assert (bus.cb.grant == 2'b01);
    // продолжение теста
```

Если сигнал **grant** назначен корректно, тест продолжается. Если же сигнал не получает требуемого значения, симулятор автоматически выдает сообщение типа

"test.sv", 7: top.t1.a1: started at 55ns failed at 55ns
Offending '(bus.cb.grant == 2'b1) '

Это сообщение означает, что в строке 7 программы **test.sv**, ассерция **top.t1.a1** была выполнена на 55 наносекунде модельного времени и проверила сигнал **bus.cb.grant**; при этом было обнаружено нессответствие результата требуемому.

Кроме предопределенных операций выполняемых при обнаружении несоответствия пользователь имеет возможность задать иные. В конструкциях процедурных ассерций можно задать специфические дополнительные операции. В том числе можно предусматривать альтернативные действия при обнаружении и не обнаружении несоответствий с использованием выражений типа **then**- и **else**-.

Пример 4.35. Задание сообщения пользователем

```
a1: assert (bus.cb.grant == 2'b01)
   grants_received++; //  операция при успехе
   else $error("Grant not asserted");
```

Если значение **grant** не соответствует ожидаемому, получим сообщение ошибке вида:

"test.sv", 76: top.t1.a1: started at 55ns failed at 55ns

Offending '(arbif.cb.grant == 2'b1)'
Error: "test.sv", 76: top.t1.a1: at time 55 ns
Grant not asserted

SystemVerilog допускает четыре функции для задания «важности ассерции», то есть действий при обнаружении несоответствия: $info, $warning, $error, $fatal. При моделировании, задавая опции симулятора, можно определить, продолжать ли моделирование или прекратить, если несоответствие утверждения обнаружено в ассерции, уровень важности которой равен или ниже заданного опцией. Подобно можно разрешить или запретить печать сообщения для ассерций в зависимости от уровня важности:

a1: **asser**t (bus.cb.grant == 2'b01)
$warning("Grant not asserted");

Если при запуске симулятора задано игнорирование ассерций (например, опция "ignore accertions" в симуляторах QuestaSim) с важностью **warning**, то обнаружение несоответствия условия в этом примере, равно как и для других ассерций с таким же и более низким уровнем важности, не останавливает моделирование, а с более высоким (**error** и **fatal**) прекращает.

Параллельные ассерции

Параллельные ассерции (concurrent assertions) записываются вне процедурных блоков и постоянно готовы к исполнению. Момент их исполнения определяется вкладываемой в выражение ассерции конструкцией **property** (свойство). В SystemVerilig конструкция **property** используется в различных случаях, в частности для задания последовательностей событий. В рассматриваемом в данном параграфе приложении **property** задает событие, вызывающее исполнение ассерции, и собственно проверяемое высказывание.

Пример 4.36 иллюстрирует простейшую форму параллельной ассерции.

Пример 4.36 Проверка корректности формирования подтверждения прямого доступа
assert property (@(posedge mr)
disable **iff (!hlda)** drq[3] |->dack[3] || drq[3] |->dack[3] || drq[3] |->dack[3] || drq[3] |->dack[3]
);

Такой текст может быть вставлен в модуль **cdma** примеров 4.14–4.16 .

Проверка выполняется по фронту сигнала **wr** (запись). Проверяется, что если в векторе запросов на обслуживание **drq** присутствует единица (есть запрос), то к моменту выполнения чтения будет установлен соответствующий сигнал подтверждения dack.

Необязательная в общем случае конструкция **disable iff** блокирует проверки – в данном примере проверки выполняются только при наличии ответа процессора о предоставлении прямого доступа **hlda**.

Property может быть определено вне конструкции **assert**, и включатся в ассерцию путем указания имении, как в примере 4.37.

Пример 4.37. Параллельная ассерция проверяет возможность неопределенного состояния сигнала

```
interface arb_if (input bit clk);
logic [1:0] grant, request;
logic reset;
  property request_2state;
    @(posedge) disable iff (reset)
    $isunknown(request) == 0;
  endproperty
request_2state: assert property (request_2state);
endinterface
```

Здесь ассерция включена в блок интерфейса и проверяет, что по фронту сигнала **clk** за исключением интервала сброса **reset** выходные сигналы арбитра не находятся в неопределенном (X или Z) состоянии.

Прямое Тестирование

сталкиваясь с задачей проверки правильности
проекта, вы, возможно, использовали прямые тесты. Используя этот подход, просматривают Спецификацию оборудования и пишут план проверки со списком тестов, каждый из которых направлен на набор соответствующих функций. Вооружившись этим планом пишут стимулирующие вектора, воспроизводят эти функции в тестирующей программе е Затем имитируют DUT с этими векторами и вручную просматривают полученные файлы отчетов и сигналы, чтобы убедиться, ТсУ делает то, что вы ожидаете. После того как тест работает правильно, ставится галочка в тест - план и переходят дальше.

Это поэтапный подход делает устойчивый прогресс, всегда популярен среди менеджеров, которые хотят видеть как проект продвигается вперед. Он также дает почти немедленные результаты, так как достаточна малая инфраструктура , если Вы ориентируйтесь на создание индивидуальных стимул-векторов. При достаточном количестве времени и штатного расписания, прямое тестирования подходит во многих случаях,.

Рисунок 1-1 показывает, как прямое тестирование постепенно охватывать функции определенные в плане верификации . Каждый тест ориентирован на очень специфический набор элементов ТсУ.

Имея достаточно времени, вы можете написать все тесты со 100% охватом всего плана проверки.

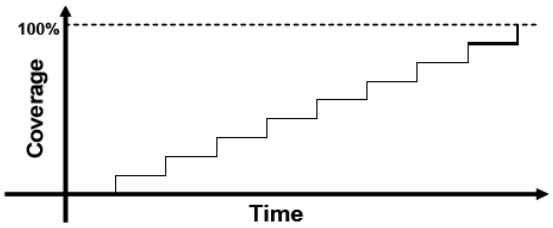


Рисунок 1-1 прогресс при прямом тестировании

Что делать, если недостаточно времени или ресурсов для выполнения направленного тестирования? Как видите, вы всегда можете делать продвижение вперед, но наклон остается прежним. Когда сложность конструкции удваивается, она занимает вдвое больше времени для завершения или требует в два раза больше людей. Разве такие ситуации желательны. Следует применить методологию, которая находит ошибки быстрее

Figure 1-2 Directed test coverage

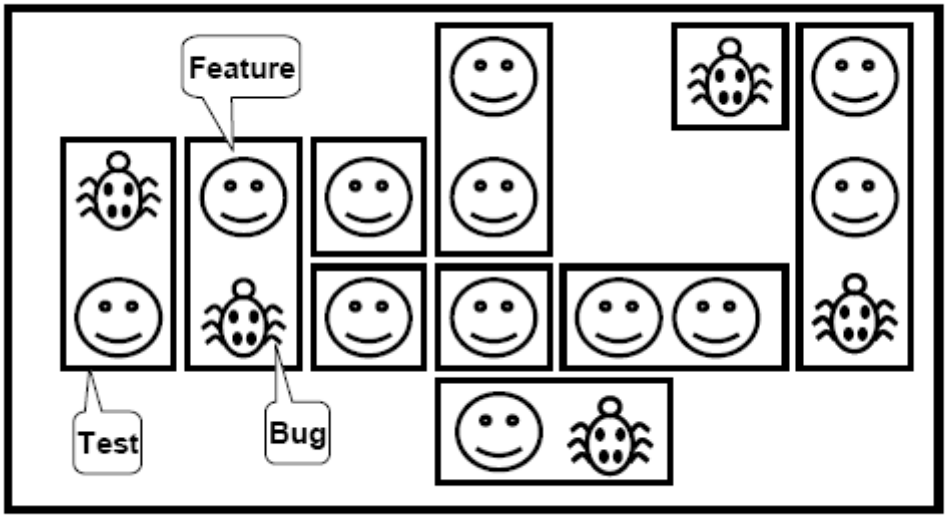


Рисунок 1-2 покрытия при прямом тестировании
Рисунок 1-2 показывает общее пространство проекта и функции, которые покрываются прямыми тестами . В этом пространстве множество функций, некоторые из которых. Вы должны написать тесты, которые охватывают все возможности и найти все шибки.