Лекция 13

9,2 Типы покрытия

Покрытие является общим термином для измерения прогресса для завершения проверки проектов.

Ваш моделирования постепенно закрашивают области проекта , по мере того как вы пытаетесь Охватить все легальные комбинаци. Инструменты покрытия собирают информацию во время моделирования, а затем выполняется пост-обработка, чтобы создать отчет о покрытии.

Этот отчет можно использовать для поиска дыр в покрытии а затем изменить существующие тесты или создавать новые, чтобы заполнить дыры. Этот повторяющийся процесс продолжается, пока вы неудовлетворены уровнем покрытия.

Когда вы собираете информацию о том , какие функции были покрыты, Вы выполняете «покрытие проекта». Например, просматривая план проверки D-триггера можно было отметить не только хранение данных, но и как он сбрасывается в Определенное состояние

9.2.1 Покрытие кода

Самый простой способ измерить прогресс тестирования это покрытия кода.

Здесь вы измеряете, сколько строк кода были выполнены (Покрытия строк), какие пути исполнения кода и выражения были выполнены (покрытия путей), какие из одно-битных переменных имели значения 0 или 1 (покрытия переключений), и какие состояния и переходы в автомате были посещены (FSM покрытия). Вам не нужно писать никакого дополнительного кода HDL. Инструменты проектирования автоматически это делают на основе анализа исходного кода и добавления скрытого кода для сбора статистики. Затем запускают все тесты, и инструмент покрытия кода создает базу данных.

Многие симуляторы включают инструмент проверки покрытия кода. Пост-обработка этих инструментов преобразует базу данных в читаемую форму. Конечный результат является мерой того, насколько тесты осуществили проверку кода. Обратите внимание, что это в первую очередь касается анализом кода устройства , а не testbench.

Непроверенный код может скрывать аппаратные ошибки, или может быть просто быть избыточным.

Покрытие кода показывает, насколько тщательно тестом осуществляется "внедрение" проектной спецификации, а не план проверки. Если даже ваши испытания достиг 100% покрытие кода, ваша работа еще не окончена

Функциональные покрытия привязаны к концепции проекта и иногда называются "Спецификациями покрытия", а покрытие кода измеряет «конструкцию» реализации. Но что произойдет, если блок кода отсутствует в проекте . Покрытие кода не может поймать эту ошибку, а функциональное может.

Что делать, если вы сделали ошибку, а ваш тест не поймал? Что еще хуже, если в реализации не хватает функции. Следующий модуль описывает Dff

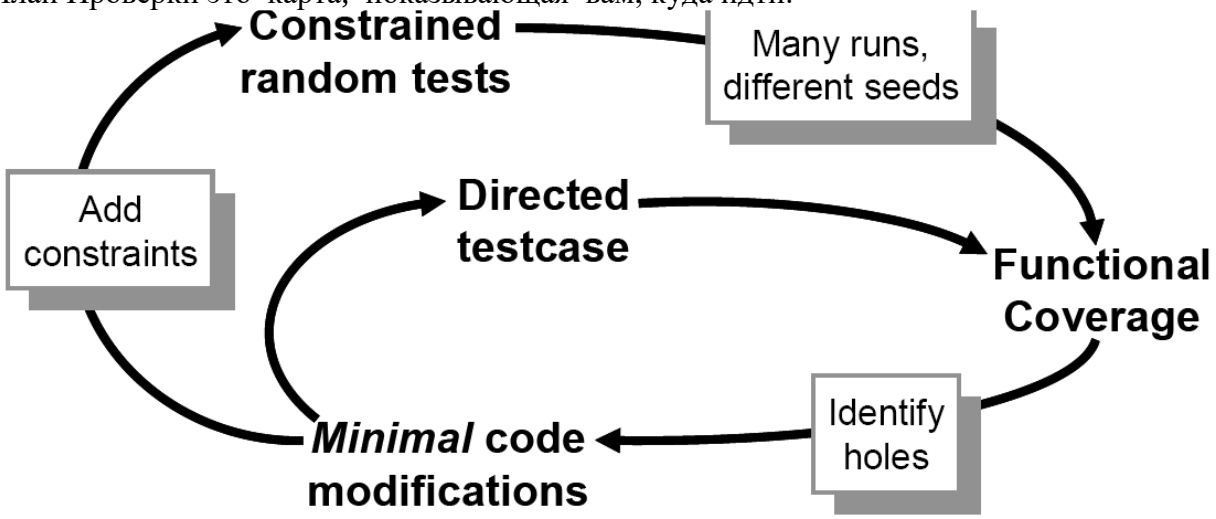
Вы видите ошибку?

```
module dff(output logic q, q_l)
input logic clk, d, reset_l);
always @(posedge clk or negedge reset_l) begin
q <= d;
q_l <= !d;
end
endmodule
```

Dff модель неполна - отсутствует путь -логика Сброса случайно пропущена. Инструмент покрытия кода сообщит, что каждая линия была осуществляться, но модель не была реализована правильно.

Функциональные покрытия

Проекты становятся все более сложными, единственным эффективным способом их тщательной проверки является CRT. Такой подход освобождает
От скуки написания отдельных направленных тестов, по одному для каждой функции. Однако, если ваш testbench определяет случайное блуждание по пространству всех состояний, как вы узнаете, достигли ли цели?
Если вы используете случайные (впрочем и направленные) стимулы, вы можете оценить Степень проверки с использованием покрытия.
Функциональные покрытия является мерой с которой конструктивные особенности были реализованы в ходе испытаний. Начните с технических условий на проектирование и создания тест-плана с подробным перечнем того, что проверить и как. Например, если ваш проект подключается к шине, тесты должны проявлять все возможные взаимодействия между блоками и шиной, в том числе соответствующих состояний , задержек и ошибок
План Проверки это карта, показывающая вам, куда идти.



Используйте обратную связь для анализа результатов покрытия и решить, какие действия следует предпринять для того, чтобы обеспечить на 100% покрытия. Ваш первый выбор -запускать существующие тесты с большим числом ядер , а второй заключается в создании новых ограничений. Прибегать к созданию направленного тестов следует , только если это абсолютно необходимо.

Если вы используете только направленые тесты, план проверки ограничен. Если в спецификации проекта перечислено 100 фунций , вам достаточно
было написать 100 тестов.
С CRT, вы освобождаетесь от ручной записи каждой строки входных стимулов, но теперь вы должны написать код, который отслеживает эффективность теста.
Вы становитесь более продуктивным, ибо работаете на более высоком уровне абстракции. Вы перешли от настройки отдельных битов к описанию интересующих состояний устройства . Необходимость Достижения 100% функционального покрытия заставляет вас думать больше о том, что вы хотите наблюдать и как вы можете направить проект в эти состояния.

9.1.1 Сбор данных покрытия

Вы можете запускать тот же случайный testbench снова и снова, просто изменив Ядро случайной последовательности для создания новых стимулов. Каждое средство моделирования создает базу данных для функциональных покрытий , и фиксации следов от случайного блуждания. Вы можете объединить все это вместе, чтобы измерить общий прогресс в расширении функционального покрытия.

Затем необходим анализ данных о покрытии чтобы решить, как изменить тест. Если уровень охвата неуклонно растет, вам просто нужно запустить существующие испытания с новыми ядрами , или даже разработать больше тестов. Если рост покрытия начал замедляться, вы можете добавить дополнительные ограничения для получения более "интересных" стимулов. Когда вы достигнете плато, некоторые части устройства недоступны , поэтому нужно создать дополнительные тесты. Наконец, когда тестом покрыты около 100%, проверьте «степень ошибочности» . Если обнаружены еще не найденные ошибки, возможно вы упускаете истинное покрытие для некоторых областей устройства.

Каждый поставщик систем моделирования имеет свой собственный формат для хранения данных о покрытии, а также свои собственные инструменты анализа. Вы должны выполнить следующие действия с помощью этих инструментов.

- Запуск теста с несколькими ядрами . Для данного набора ограничений (и групп Покрытия), компилировать testbench и DUT в один исполнимый модуль.. Теперь вам нужно многократно запустить это случайный тест с ограничениями с различными ядрами случайной последовательности
- Проверка на успех / неуспех. Информация о Функциональном покрытии действительна только при успешном моделировании. Если налицо неуспех вызванный тем что есть ошибка, информации о покрытии должна быть уничтожена.
- Анализ покрытия по результатам нескольких запусков. Вы должны оценить, насколько успешно каждое ограничение набора, с течением времени. Если вы еще не достигли100% охвата районов, на которые направлены ограничения, но объем покрытия продолжает расти, запустите больше тестов с различными ядрами. Если уровень охвата стабилизировался, настало время изменить ограничения.

Только если вы думаете, что достижения последних нескольких тестов для одного определенного раздела может занять слишком много времени для случайного моделирования Вы должны использовать направленый тест. Даже тогда, продолжайте использовать случайный стимул для других разделов устройства, в иногда в таком случае этот "фоновый шум" находит ошибку.

9.2.3 Интенсивность обнаружения Ошибок

Косвенный способ измерения покрытия, это оценка интенсивности обнаружения новых ошибок. Вы должны отслеживать, как много ошибок вы нашли за или неделю, в течение всего срока реализации проекта. В начале, вы можете найти много ошибок непосредственно при создании testbench. Когда вы будете читать спецификацию , вы можете найти несоответствия, которые, будем надеяться фиксируются до RTL написания.

После того, testbench запущен и работает, интенсивность потока ошибок изменяется по мере того как вы проверяете все модули в системе. Исправлена ошибка- скорость падает, мы надеемся, к нулю, так как проект приближается к завершению. Тем не менее, именно потому, что интенсивность обнаружения ошибки ставки стремится к нулю, вы еще не все сделали. Каждый раз, когда скорость «проседает», настало время, чтобы найти различные способы проверить крайние случаи.

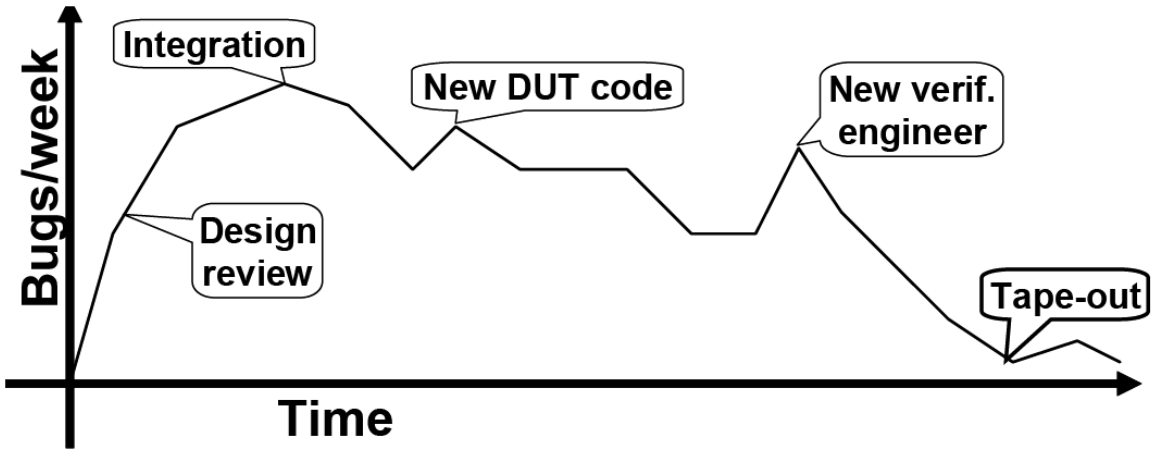


Рисунок 9-3 интенсивность обнаружения Ошибка в ходе выполнения проекта
Интенсивность обнаружения (и устроенения)ошибок, может варьироваться в во времени зависит от многих факторов, таких как фазы проекта, Текущие изменения в проекте, интеграция блоков, кадровые изменения, и даже график отпусков. Неожиданные изменения в скорости может сигнализировать о потенциальной проблеме. Как показано на рисунке 9-3, это не редкость, и следует продолжать поиск ошибки даже после того, как проект сдан , и даже после того, как он передан заказчику.

9,3 стратегии оценки функционального покрытия
Перед тем как написать первую строку тестового кода, вы должны оценить
Ключевые особенности проекта, крайние случаи, а также возможность отказов и повреждений. Думайте не только о значениях данных;
вместо этого следует продумать, как информация кодируется . В Плане должны быть представлены существенные состояния устройства .

9.3.1 Собирайте информации, а не данные
Классическим примером является FIFO. Как вы можете быть уверены, у вас полностью испытаны 1K памяти FIFO? Вы можете изменять значения индексов при чтении и записи, но возможно более миллиона комбинаций. Даже если вы
смогли смоделировать такое множество циклов, вы не захотите читать **coverage report**.
На более абстрактном уровне, FIFO может содержать от 0 до N-1 возможных значений.
А что если вы просто сравните индексы чтения и записи для оценки, насколько полон или пуст буфер FIFO? У вас останется 1K значений покрытия. Если ваш testbench
Выполнил 100 записей в FIFO, а затем выполнил на100 больше,
нужно ли проверять, бывало ли в FIFO 150 значений?
Крайними случаями FIFO являются «полный» и «пустой». Если вы можете вынудить FIFO перейти от «пусто» (состояние после сброса) до «полно» и обратно ,
Вы охватываете все уровни между ними. Также интересны состояния связанные с Изменением индекса между «все 1» и «все 0». Отчет о покрытии (coverage report) для этих случаев легко понять.
Вы могли заметить, что специфические состояния не зависят отразмера FIFO. Еще раз заметим: анализируем информацию, а не данные.
Диапазон сигналов с большим диапазоном (более нескольких десятков возможных значений)должен быть разбит на более мелкие диапазоны, а также граничные случаи. Например,ваш DUT может иметь 32-разрядную шину адреса, но вы, конечно, не нужно собирать 4000000000 отсчетов. Учитывайте наличие естественных подразделений, таких как память и IO пространство. Для счетчика, достаточно выбрать несколько интересных значений, и всегда старайтесь проверить переход от состояния « все 1» к 0.

9.3.2 измеряйте Только то, что вы собираетесь использовать
Сбор данных о функциональном покрытии может быть дорогим удовольствием , так что измеряйте только то что вы будете анализировать и использовать для улучшения ваших тестов. Ваш моделирование станет работать медленнее, так как дополнительно контролирует сигналы для функционального покрытия, но этот подход требует меньших усилий , чем анализ временных диаграмм и измерения покрытия кода. После завершения моделирования, база данных сохраняется на диске. Имея несколько testcases и несколько ядер случайных последовательностей, вы можете пополнять диск информацией о функциональном покрытии.
. Но если вы никогда не рассматриваете на заключительный **coverage report** , не следует выполнятm начальные измерения.
Есть несколько способов управления сбором данных о покрытии: при компиляции, включении экземпляра, при запуске. Вы можете использовать переключатели предоставляемые поставщиком симулятора , условную компиляции, подавление сбора данных о покрытии.
Последний из них менее желателен, так как после обработки отчетов заполняется разделы с 0% охвата, что создает трудности нахождения нескольких оставшихся.
9.3.3 Измерение полноты (возврат к рис)
Цель состоит в том, чтобы кодовое и функциональное покрытия оба были достаточно высокими . Но отдыхать рано!. Какова тенденция интенсивности обнаружения ошибок ? Существенные ошибки по-прежнему появляются ? Что еще хуже, они были найдены умышленно, или же ваш testbench Случайно наткнуться на ту или иную комбинацию состояний, которую никто не ожидал? С другой стороны, низкий уровень ошибкок может означать, что существующие стратегии выдыхаться, и вы должны рассмотреть разные подходы. Попробуйте различные подходы, такие как новые сочетания
блоков устройства и ошибки генераторов.

9,4 Простой пример функционального покрытия

Для оценки функционального охвата, начинают с плана проверки и пишут исполняемую версию его для моделирования. Ваш SystemVerilog testbench, фиксирует значения переменных и выражений. Эти деленные места называют точками покрытия (**cover poin**ts). Несколько cover points, которые отобраны в одно время (например, при завершении транзакции) составляют группу покрытия (covergroup).

Следующий фрагмент содержит транзакцию, данные могут принимать в 8 различных значений. testbench генерирует переменные порта случайным образом, а план проверки требует, чтобы каждое значение было оценено.

Пример 9-2 функциональное покрытие простого объекта

//Busifc- ранее продекларированный интерфейс

```
program automatic test(busifc.IB ifc);

  class Transaction;
    rand bit [31:0] data;
    rand bit [ 2:0] port;     // Eight port numbers
  endclass

  covergroup CovPort;
    coverpoint tr.port;       // Measure coverage
  endgroup

  Transaction tr = new;

  initial begin
    CovPort ck = new;         // Instantiate group

    repeat (32) begin         // Run a few cycles
      assert(tr.randomize);   // Create a transaction
      ifc.cb.port <= tr.port; //   and transmit
      ifc.cb.data <= tr.data; //   onto interface
      ck.sample();            // Gather coverage
      @ifc.cb;                // Wait a cycle
    end
  end
endprogram
```

.

Testbench собирает значения на порте в covergroup с именем CovPort. Восемь возможных значений, 32 случайных операций - сделал ваш testbench генерировать их всех? Вот часть покрытия файле отчета VCS.

Пример 9-3 отчет о покрытии для простого объекта

```
Coverpoint Coverage report
CoverageGroup: CovPort
    Coverpoint: tr.port
Summary
    Coverage: 87.50
    Goal: 100
    Number of Expected auto-bins: 8
    Number of User Defined Bins: 0
    Number of Automatically Generated Bins: 7
    Number of User Defined Transitions: 0

    Automatically Generated Bins

    Bin             # hits    at least
    ================================
    auto[1]         7         1
    auto[2]         7         1
    auto[3]         1         1
    auto[4]         5         1
    auto[5]         4         1
    auto[6]         2         1
    auto[7]         6         1
    ================================
```

Как вы можете видеть, testbench породил значения 1, 2, 3, 4, 5, 6 и 7, но никогда не выдавал 0. По крайней мере, колонка определяет, сколько циклов прошло прежде чем контейнер считается охваченным.

Пример 9-4 Отчет о покрытие, простой объект, 100% покрытие

```
Coverpoint Coverage report
CoverageGroup: CovPort
    Coverpoint: tr.port
Summary
    Coverage: 100
    Goal: 100
    Number of Expected auto-bins: 8
    Number of User Defined Bins: 0
    Number of Automatically Generated Bins: 8
    Number of User Defined Transitions: 0

    Automatically Generated Bins

    Bin             # hits    at least
    ================================
    auto[0]         1         1
    auto[1]         7         1
    auto[2]         7         1
    auto[3]         1         1
    auto[4]         5         1
    auto[5]         4         1
    auto[6]         2         1
    auto[7]         6         1
    ================================
```

9,5 Анатомия группы покрытия

Группа покрытия похожа на класс - вы определяете один раз и затем экземпляр (один или несколько). Он содержит набор точек покрытия , опции, формальные аргументы ,и дополнительный переключатель. Группа покрытия включает в себя одну или несколько **точек покрытия** (данных), все из которых отбираются в одно время. Группы покрытия могут быть определены в классе или на уровне программы или модуля. В них можно может фиксировать видимые переменны, таких как программа / модуль переменных, сигналов от интерфейса или любой сигнал в устройстве (с помощью иерархической ссылки).

Группы покрытия внутри класса могут фиксировать переменные этого класса, а также данные от встроенных классов.

В SystemVerilog, вы должны определить **covergroup** на соответствующем уровне абстракции. Этот уровень может быть на границе между testbench и DUT, в операциях чтения и записи данных, в классе конфигурации среды, а вообще в любом месте, где необходимо. Выборка любой транзакции должна задерживаться , до тех пор пока транзакция фактически получена тестируемым устройством. Если Вы вводите ошибку в процессе передачи , заставляя ее прервать, необходимо уточнить , как вы относитесь к покрытию функции передачи. Вы должны использовать другую точку покрытия , которая создается только для обработки ошибок.

Класс может содержать несколько групп покрытия . Такой подход позволяет

Иметь отдельные группы, которые можно включать и выключать по мере необходимости. Кроме того, каждая группа может иметь отдельный спусковой механизм, позволяющий собирать данные многих источников.

9.5.1 Определение группы покрытия в классе

Группы покрытия могут быть определены в программе, модуля или класса. Во всех случаях нужно явно создать экземпляр его для начала выборки. Если группа определена в классе, вы не создаете отдельное название, когда его используете нужно только оригинальное название крышкой покрытия .

Пример 9-5 очень похож на первый пример в этой главе исключением того, что он включает группу покрытия класса транзактора, и, следовательно, не нуждается в отдельном имени экземпляра.

Пример (9-5) декларации функционального покрытия внутри класса

```
class Transactor;
  Transaction tr;
  mailbox mbx_in;
  covergroup CovPort;
    coverpoint tr.port;
  endgroup

  function new(mailbox mbx_in);
    CovPort = new;              // Instantiate covergroup
    this.mbx_in = mbx_in;
  endfunction

  task main;
    forever begin
      tr = mbx_in.get;          // Get next transaction
      ifc.cb.port <= tr.port; // Send into DUT
      ifc.cb.data <= tr.data;
      CovPort.sample();         // Gather coverage
    end
  endtask

endclass
```

9,7выборка Данных

Как покрытие собирает информацию?

При указании переменной или выражения в точке покрытия SystemVerilog создает ряд **«контейнеров»** для записи, сколько раз каждое значение получалось. Эти контейнеры являются основными единицами измерения для функционального покрытия . Если выбираете одно-битную переменную, максимум два контейнера будут созданы. Можно считать, что SystemVerilog заносит символ в той или иной контейнер каждый раз, когда происходит событие, объявленное в covergroup . В конце каждого моделирования, создается база данных со всеми контейнерами, в которых есть символ. Затем можно запустить инструмент анализа, который читает все базы данных и генерирует отчет с указанием результатов для каждой части проекта и для целостного представления.

9.7.1 Индивидуальные контейнеры и полный охват

Для того чтобы подсчитать покрытие для точки , вы должны сначала определить общее

число возможных значений, называемый также доменом. Там может быть одно значение для контейнера одной или несколько значений. Покрытие = число дискретных значений, деленное на число контейнеров в домене.
Покрытие 3-битной переменной представляет домен 0:7 и, как правило, разделен на восемь ячеек. Если во время моделирования Выборки принадлежали к семи контейнерам в отчете будет показано 7/8 или 87,5% охвата этой точки. Все точки одной группы объединены, чтобы показать покрытие для всей группы, а затем Все группы объединяются, чтобы дать общий процент охвата.
Это состояние для одной симуляции. Вы должны отслеживать покрытие во времени. Изучайте тенденции, таким образом, Вы можете видеть, надо ли продолжать моделирования или добавить новые ограничения или тесты. Теперь вы можете лучше предсказать, когда верификация проекта будет завершена.

9.7.2 автоматическое Создание контейнера
Как вы видели в отчете в примере 9-3, SystemVerilog автоматически создает контейнеры для точек покрытия . Он прослеживает в области выбранных выражений Для определения диапазона возможных значений. Для выражения, длиной N битов широкий, есть 2N значений. Вы также можете явным образом определить контейнер, как показано далее.

9.7.3 Ограничение на создаваемое число числа автоматических контейнеров
Опция группы покрытия **auto_bin_max** определяет максимальное число контейнеров для автоматического создания (значение по умолчанию 64) . Если область значений в точке покрытия переменной или выражения больше этого параметра, SystemVerilog делит диапазон на auto_bin_max контейнеров . Например, 16-битная переменная имеет 65536 возможных значений, так что каждый из 64 контейнеров охватывает 1024 значения.
Следующая программа использует первый пример данного раздела и добавляет в число точек покрытия вариант, который устанавливает auto_bin_max два контейнера . Отобранная переменная port по-прежнему, три бита, и домен восьми возможных значений.
первый контейнер занимает нижнюю половину диапазона 0-3, а другой верхнюю
.
Пример 9-11 Использование autu_nin_max

```
covergroup CovPort;
  coverpoint tr.port
    { options.auto_bin_max = 2; }  // Divide into 2 bins
endgroup
```

В Отчете о покрытии из VCS показаны два контейнера. Это моделирование Достигло 100% охвата, потому что восемь значений порта были назначены в 2 Контейнера .

Example 9-12 Report with `auto_bin_max` set to 2

```
Bin              # hits    at least
==================================
auto[0:3]        15        1
auto[4:7]        17        1
==================================
```

Пример 9-11 auto_bin_max использован в качестве опции для точки покрытия .

Вы можете также использовать его в качестве опции для всей группы.

```
covergroup CovPort;
  options.auto_bin_max = 2; // Affects port & data
  coverpoint tr.port;
  coverpoint tr.data;
endgroup
```

9.7.4 выборка выражений

Вы можете использовать и качестве наполнения покрытий выражения, но всегда проверяйте формат, в котором получаете результат вычисления выражения .

Пример 9-14

```
class Transaction;
  rand bit [2:0] hdr_len;      // range: 0:7
  rand bit [3:0] payload_len;  // range: 0:15
  ...
endclass

Transaction tr;

covergroup CovLen;
  len16: coverpoint (tr.hdr_len + tr.payload_len);
  len32: coverpoint (tr.hdr_len + tr.payload_len + 5'b0);
endgroup
```

9.7.7 Условные покрытия

для записи условного покрытия следует использовать ключевое **iff**

Часто это используют для отключения сбора во время сброса. Кроме того, вы можете использовать функции start и stop для управления отдельными экземплярами групп покрытия.

Example 9-19 Conditional coverage — disable during reset

```
covergroup CoverPort;
  // Don't gather coverage when reset==1
  coverpoint port iff (!bus_if.reset);
endgroup
```

Example 9-20 Using `stop` and `start` functions

```
initial begin
  CovPort ck = new;          // Instantiate cover group

  // Reset sequence stops collection of coverage data
  #1ns bus_if.reset = 1;
  ck.stop();
  #100ns bus_if.reset = 0; // End of reset
  ck.start();
  ...
end
```

9.7.9 покрытия Переходов
Вы можете определить состояние переходов для точки покрытия . Таким образом, можно указать не только какие интересные значения были отмечены , но и последовательности. Например,Вы можете проверить, переходил ли порт никогда не пошел от 0 до 1, 2 или 3.

Example 9-23 Specifying transitions for a cover point

```
covergroup CoverPort;
  coverpoint port {
    bins t1 = (0 => 1), (0 => 2), (0 => 3);
  }
endgroup
```

Вы можете быстро определить несколько переходов с помощью диапазонов. Выражение (1,2 => 3,4) создает четырех переходов (1 => 3), (1 => 4), (2 => 3) и
(2 => 4).
Вы можете задать переходы любой длины. Обратите внимание, что вы должны единожды выбирать каждое состояние в переходе. Таким образом, (0 => 1 => 2) отличается от (0 => 1 => 1 => 2) или (0 => 1 => 1 => 1 => 2). Если вам нужно повторить
Значения , как и в предыдущем примере, вы можете использовать сокращенную форму: (0 => 1 [* 3]=> 2). Чтобы повторно фиксировать переход из 1 3, 4 или 5 раз, используйте 1 [* 3:5].

9.7.11 Игнорирование значения
Для некоторых точек покрытия , вы никогда не получите все возможные значения. Например,3-битной переменная может быть использован для хранения всего шести значений, 0-5. Если вы используете создание автоматическую
Контейнера , вы никогда не выйдете за пределы 75% охвата. Есть два способа решить эту проблему. Вы можете явно определить в контейнере , что вы хотите, чтобы покрыть. Кроме того, вы можете позволить SystemVerilog автоматически
создать контейнер , а затем используя ignore_bins сказать, какое значение исключить из расчета функционального покрытия.

Example 9-25 Cover point with `ignore bins`

```
bit [2:0] low_ports_0_5;      // Only uses values 0-5
covergroup CoverPort;
  coverpoint low_ports_0_5 {
    ignore_bins hi = {[6,7]}; // Ignore upper 2 bins
  }
endgroup
```

9.7.13 Покрытия автомата
Вы должны были заметить, что если группа покрытия используется для автомата , Вы можете использовать контейнеры для перечисления конкретных состояний, и переходов для дуг. Но это не означает, что вы должны использовать функциональное охват SystemVerilog для измерение Автомата . Вы должны были бы извлечь состояния и дуги вручную. Даже если вы сделали это правильно в первый раз, вы можете пропустить будущие изменения кода устройства. Вместо этого используйте встроенный в САПР

инструмент анализа покрытия кода, который извлекает состояния и дуги автоматически, избавляя вас от возможных ошибок.

.

9,8 Перекрестные покрытия

Точка покрытия записывает наблюдаемые значения одной переменной или выражения. Вы можете захотеть узнать не только то, что транзакция шины совершена, но также какие ошибки произошло во время этих операций, и их источник и назначение.
Для этого вам необходимо использовать перекрестное покрытие, чтобы оценить какие значения , были замечены В двух или более точках покрытия одновременно. Обратите внимание, что при измерении Пересекающихся покрытий переменной с N значений, и другой с М значениями. В SystemVerilog Необходимо NxM перекрестных контейнеров для хранения всех комбинаций.

9.8.1 Основы перекрестного покрытия

Конструкция **cross** в SystemVerilog определяет Комбинации. значений двух или более точек покрытия в группе. Выражение cross Содежит полько имена точек пократия или имя простой переменной. Если вы хотите использовать Выражения, иерархические имена или имена объектов , такие как handle.variable, необходимо указать выражение в **coverpoint** с меткой , а затем использовать ее в перекрестном объявлении.
Пример 9-28 создает точки покрытия tr.kind и tr.port. потом создается перекрестное покрытие двух точек , чтобы показать все комбинации. SystemVerilog создает в общей сложности 128 (8 x 16) контейнеры. Даже простое пересечение , может привести к очень большому числу контейнеров

Example 9-28 Basic cross coverage

```
class Transaction;
  rand bit [3:0] kind;
  rand bit [2:0] port;
endclass

Transaction tr;

covergroup CovPort;
  kind: coverpoint tr.kind;  // Create cover point kind
  port: coverpoint tr.port   // Create cover point port
  cross kind, port;          // Cross kind and port
endgroup
```

Example 9-29 Coverage summary report for basic cross coverage

```
Cumulative report for Transaction::CovPort
Summary:
    Coverage: 95.83
    Goal: 100

Coverpoint           Coverage     Goal         Weight
=========================================================
kind                   100.00      100         1
port                   100.00      100         1
=========================================================
Cross                  Coverage    Goal         Weight
=========================================================
Transaction::CovPort  87.50        100         1

Cross Coverage report
CoverageGroup: Transaction::CovPort
    Cross: Transaction::CovPort
Summary
    Coverage: 87.50
    Goal: 100
    Coverpoints Crossed: kind port
    Number of Expected Cross Bins: 128
    Number of User Defined Cross Bins: 0
    Number of Automatically Generated Cross Bins: 112

    Automatically Generated Cross Bins

    kind       port            # hits       at least
 ================================================
    auto[0]    auto[0]         1            1
    auto[0]    auto[1]         4            1
    auto[0]    auto[2]         3            1
    auto[0]    auto[5]         1            1
...
```

.

9.8.2 Маркировка перекрестных контейнеров покрытия

Если вы хотите более читабельным имена контейнеров перекрестные покрытие , вы можете пометить отдельные точки контейнера , и SystemVerilog будет использовать эти названия, при создании перекрестных контейнеров.

Example 9-30 Specifying cross coverage bin names

```
covergroup CovPortKind;
  port: coverpoint tr.port
   {bins port[] = {[0:$]};
   }
  kind: coverpoint tr.kind
   {bins zero = {0};        // A bin for kind==0
    bins lo   = {[1:3]};    // 1 bin for values 1:3
    bins hi[] = {[8:$]};    // 8 separate bins
    bins misc = default;    // 1 bin for all the rest
  }
  cross kind, port;
endgroup
```

9.8.3 Исключение перекрестных контейнеров

Чтобы уменьшить количество контейнеров , можно использовать **ignore_bins**. Для перекрестного покрытия охвата следует указать точку покрытия **binsof** и множество значений с опцией intersect, так как одна операция **ignore_bins** может смести многие индивидуальные контейнеры

Example 9-32 Excluding bins from cross coverage

```
covergroup CovCovport;
  port: coverpoint tr.port
   {bins port[] = {[0:$]};
   }
  kind: coverpoint tr.kind
   {
    bins zero = {0};        // A bin for kind==0
    bins lo   = {[1:3]};    // 1 bin for values 1:3
    bins hi[] = {[8:$]};    // 8 separate bins
    bins misc = default;    // 1 bin for all the rest
  }
  cross kind, port {
    ignore_bins hi = binsof(port) intersect {7};
    ignore_bins md = binsof(port) intersect {0} &&
                     binsof(kind) intersect {[9:10]};
    ignore_bins lo = binsof(kind.lo);
  }
endgroup
```

Первый **ignore_bins** просто исключает контейнеры , где port=7 и любые значения переменной kind.. Так как kind это 4-битное значение, это заявление исключает 16 контейнеров . Второй более избирателен - он игнорирует контейнеры для которых port=0 , а kind= 9, 10 или 11, всего 3 контейнера .

Ignore_bins может использовать контейнеры, определенных в индивидуальных точках покрытия.
Ignore_bins lo использует имена контейнеров , чтобы исключить **kind.lo** то есть 1, 2 или 3.
Имена Контейнеров должны быть определены во время компиляции,.
Контейнеры hi_8, hi_a ... hi_f, и любые другие автоматически генерируюемые контейнеры не должны иметь имена, которые могут быть использованы во время компиляции в других операторах; эти имена создаются во время выполнения или во время создания отчета

9.8.4 исключение точек покрытия из метрики покрытия
Общее покрытие для группы определяется исходя из всех простых точек покрытия и перекрестных покрытий. Если вы отбираете переменную или выражение в coverpoint, и они будут использоваться в перекрестном покрытии , вы должны установить его вес до 0 потому что это не войдет общее покрытие.

Example 9-33 Specifying cross coverage weight

```
covergroup CovPort;
  kind: coverpoint tr.kind
   {bins kind[] = {[0:$]};
    weight = 0;              // Don't count towards total
  }
  port: coverpoint tr.port
   {bins zero = {0};
    bins lo   = {[1:3]};
    bins hi[] = {[8:$]};
    bins misc = default;
    weight = 5;              // Count in total
  }
  cross kind, port
    {weight = 10;}           // Give cross extra weight
endgroup
```

9.8.5 Слияние данных из нескольких доменов

9,9 **Опции покрытия (пропускаем!)**
9,10 параметризованные покрытия
Как начать писать, вы обнаружите, что некоторые из них очень близки к друг с другом. SystemVerilog позволяет параметризовать covergroup, так что вы можете создать общее определение, а затем указать несколько уникальных деталей, когда создаете его экземпляр. Но нельзя передавать переключатель (триггера) в экземпляр группы.
Вместо этого, следует оформить группу покрытие в классе и передать переключатель (триггер) в конструктор.
9.10.1 Передача параметров группа покрытия по значению
Пример 9-41 показывает группу покрытия , в которой используется параметр разделения диапазонана две половины. Просто передается середиее значение функции новой группы

Example 9-41 Simple parameter

```
bit [2:0] port;     // Values: 0:7

covergroup CoverPort (int mid);
  coverpoint port
   {bins lo = {[0:mid-1]};
    bins hi = {[mid:$]};
  }
endgroup

CoverPort cp;
initial
  cp = new(5);            // lo=0:4, hi=5:7
```

.

9.10.2 Передача параметров группы покрытия по ссылке

Вы можете задать отбираемую переменную с использованием передачи по ссылке. Здесь Вы хотите, чтобы крышка группу, чтобы попробовать значение на протяжении всего моделирования, а не
просто использовать значение, когда конструктор вызывается.

Example 9-42 Pass-by-reference

```
bit [2:0] port_a, port_b;

covergroup CoverPort (ref bit [2:0] port, int mid);
  coverpoint port {
    bins lo = {[0:mid-1]};
    bins hi = {[mid:$]};
  }
endgroup

 CoverPort cpa, cpb;
 initial
   begin
   cpa = new(port_a, 4);  // port_a, lo=0:4, hi=5:7
   cpb = new(port_b, 2);  // port_b, lo=0:1, hi=3:7
   end
```

9,13 Заключение

Когда вы переходите от написания направленных тестов, с «ручной» выработкой каждыго бита стимула, к случайным испытаниям с ограничением, вы наверное будете обеспокоены тем, что тесты больше вами не контролируются. Измеряя покрытие , особенно функциональное Покрытие , вы восстановить контроль, и знаете какие функции были проверены.

Корректная проверка функционального покрытия требует детального плана проверки и много времени на создани групп покрытия , анализа результатов и изменения тестов для создания надлежащие стимулы. Но эти затраты меньше, чем потребовалось бы
Для написания эквивалентных направленных тестов, а кроме того исследавние покрытия поможет лучше отслеживать успехи в проверке проекта