10. Ассерции

10.1. Последовательности (Sequence)

SystemVerilog предоставляет возможность описания специфических последовательных выражений или последовательностей булевых выражений с явными временными взаимоотношениями между ними. Для определения совпадения последовательностей, булевы выражения вычисляются в каждой последовательной точке считывания значений, определяемой синхросигналом, связанным с последовательностью.

Простейшее последовательное поведение является линейными. Линейная последовательность - это конечный список булевых выражений SystemVerilog в линейном порядке возрастания времени. Говорят, что линейная последовательность имела место на конечном интервале следующих друг за другом синхрособытий, если первое логическое выражение принимает значение «истина» на первом синхрособытии, второе логическое выражение получает значение «истина» при поступлении второго синхрособытия, и так далее, вплоть до последнего логического выражения, которое должно принять значение «истина» на последнем синхрособытии. Примером простой линейной последовательности является булево выражение. Его присутствие на указанном синхрособытии означает, что оно принимает в значение «истина» в данный момент времени.

Более сложное последовательное поведение в SystemVerilog описывается с помощью последовательностей (sequence). Последовательность – это упорядоченная совокупность булевых выражений, которые включают множество нулей, конечное или бесконечно число линейных последовательностей. Если хотя бы одна из линейных последовательностей, входящих во множество, имеет место в конечном интервале последовательных синхрособытий, то считается, что последовательность произошла на данном интервале времени.

Базовое выражение последовательности: «за a на следующем синхротакте следует b» на SystemVerilog может иметь вид:

```
a ##1 b
```

В этом примере, «##1» обозначает задержку длиной в один синхротакт между последующим булевым выражением.

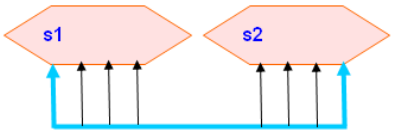
В SystemVerilog, каждый элемент последовательности может быть либо булевым выражением, либо другой последовательностью. В терминах последовательности, булево выражение – это упрощенный вырожденный случай последовательности длиной в 1. Таким образом, выражение конкатенации последовательностей

```
s1 ##1 s2
```

обозначает, что последовательность s2 начинается на синхротакт после завершения последовательности s1, как это изображено на рис 10.1. Для перекрывающихся последовательностей

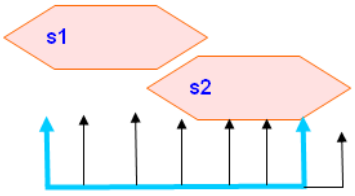
```
s1 ##0 s2
```

последовательность s2 начнет свое выполнение в синхротакте, в котором последовательность s1 завершит свое выполнение (рис 10.2).



s1 ##1 s2

Рис. 10.1 Конкатенация последовательностей



s1 ##0 s2

Рис.10.2. Перекрытие последовательностей.

Хотя последовательности очень полезны для описания временных взаимоотношений между выражениями, важным является их оформление в виде элементов языка, таким образом, чтобы они могли быть повторно использованы или на них можно было ссылаться. SystemVerilog определяет такой элемент языка – **sequence**. Кроме перечисленных возможностей, последовательности могут быть использованы при построении других последовательностей или как часть свойств параллельных ассерций (рассматриваются ниже).

10.1.1. Булевы выражения в последовательностях

Используемые в последовательностях выражения вычисляются с использованием значений переменных, которые представлены в выражении. Результат такого выражения имеет булев тип и интерпретируется точно также в условие в операторе if. Существует ряд ограничений для выражений, которые применяются в последовательностях и ассерциях.

В булевых выражениях, составляющих последовательности, не разрешены: вещественные данные (shortreal, real и realtime), строки, события, указатели, классы, ассоциативные и динамические массивы.

> Выражения позволяют включать вызовы функций, но существуют некоторые семантические ограничения. Функции, которые появляются в выражениях, не могут содержать выходной аргумент ссылочного типа ref (разрешаются const ref), Функции должны быть автоматические (или защищены от статической информации).

Массивы фиксированного размера, упакованные или нет, могут быть использованы полностью, в виде диапазона или по отдельным элементам. Индекс массива может быть задан константой, параметром или переменной. Следующий пример показывает некоторые допустимые формы сравнения членов структур и объединений. Пусть выполнено определение:

```
typedef int [4] array;
typedef struct { int a, b, c, d } record;
union { record r; array a; } p, q;
```

Тогда выражения p.a == q.a или p.r == q.r является легальными

Примеры использования массивов в выражениях:

```
logic [7:0] arrayA [0:15], arrayB[0:15];

arrayA == arrayB;
```

```
arrayA != arrayB;
arrayA[i] >= arrayB[j];
arrayB[i][j+:2] == arrayA[k][m-:2];
(arrayA[i] & (~arrayB[j])) == 0;
```

Переменные в выражениях должны быть статическими переменными проекта или вызовами функций, которые возвращают значения допустимых типов. Статические переменные, объявленные в программах, интерфейсах или блоках синхронизации могут быть также использованы в ассерциях. Если ссылка на статическую переменную декларируется в задаче, то значение такой переменной может быть прочитано в любой момент времени, независимо от вызова функции.

Разрешены все операторы, которые могут быть использованы с допустимыми в последовательностях типами данных, кроме операторов присвоения, инкремента и декремента. SystemVerilog содержит операторы из языка Си, такие как +=, ++ and --. Они не могут быть использованы в вычислении выражений, которые используются в ассерциях.

10.1.2. Синтаксис

Последовательность может быть декларирована в модуле, интерфейсе, программе, блоке синхронизации, пакете и области единицы компиляции.

SystemVerilog синтаксис для декларирования и создания копий последовательностей:

```
// Декларация
sequence sequence_identifier [ ( [ tf_port_list ] ) ] ;
    { assertion_variable_declaration }  // Будут рассмотрены ниже
    sequence_expr ;
endsequence [ : sequence_identifier ]

// Создание копии последовательности
 sequence_identifier [ ( [ list_of_arguments ] ) ]
```

Необязательный список формальных параметров позволяет описывать последовательности, как общие временные взаимоотношения. Формальные аргументы заменяются актуальными, которые передаются в последовательность при создании ее копии. Например, последовательность

```
sequence seq1 (a, b);
      a ##2 b;
endsequence
```

представляет последовательность из двух выражений. Когда создается копия последовательности

```
seq1(e,f)
```

актуальные аргументы e и f заменяют определенные в последовательности формальные аргументы (a и b, соответственно), таким образом, временные взаимоотношения между e и f будут выглядеть

```
e ##2 f
```

Таблица 10.1 объединяет некоторые из операций, которые могут быть использованы в последовательностях.

Таблица 10.1. Операции последовательностей

Операции	Синтаксис	Объяснение
Конкатенация	seq1 ##1 seq2	Последовательность seq2 начинается со следующего синхротакта после завершения seq1
Перекрытие	seq1 ##0 seq2	Последовательность seq2 начинается в том же синхротакте, в котором заканчивается последовательность seq1
Обнаружение конца последовательности	seq1 ##1 seq2.ended	Последовательность seq2 завершается на один синхротакт после завершения последовательности seq1, независимо от момента начала последовательности seq2
Повторение	seq1[*n:m]	Последовательность seq1 повторяется минимум n и максимум m раз. Может формировать множественное совпадение последовательностей
Обнаружение первого совпадения	first_match(seq1)	Если последовательность seq1 имеет множество совпадений, используется первое из них и игнорируются оставшиеся
ИЛИ	seq1 or seq2	Формирует последовательность, которая возваращет значение "истина", если обраружена хотя бы одна из последовательностей seq1 или seq2
И	seq1 and seq2	Формирует последовательность, которая возваращает значение "истина", если одна последовательность появляется до или после появления другой последовательностиMatches
Пересечение	seq1 intersect seq2	Возваращает значение "истина" в синхроциклах где выполняются обе последовательности seq1 и seq2
Уточнение условием	cond throughout seq	Условие cond имеет значение "истина" для каждого синхротакта seq
В рамках	seq1 within seq2	Последовательность seq1 начинается вместе или после начала последовательности seq2 и заканчивается вместе с ней или до ее завершения

Актуальные аргументы, заменяющие формальные, могут быть представлены:

- Идентификатором
- Выражением
- Выражением событийного контроля
- Верхним уровнем, как $(upper range as $)

Переменные, используемые в последовательностях, не являющиеся формальными аргументами, обрабатываются согласно области видимости, для границ, в которых последовательность декларируется. Последовательности могут содержать синхрособытия, например.

```
sequence s1;
      @(posedge clk) a ##1 b ##1 c;
endsequence

sequence s2;
      @(posedge clk) d ##1 e ##1 f;
endsequence

sequence s3;
      @(negedge clk) g ##1 h ##1 i;
endsequence
```

В примере последовательности s1 и s2 вычисляются по переднему фронту clk, а s3 – по заднему.

Пример декларации последовательности, содержащей аргументы:

```
sequence s20_1(data,en);
     (!frame && (data==data_bus)) ##1 (c_be[0:3] == en);
endsequence
```

Последовательность s20_1 не включает описания синхронизации. В этом случае, синхросигнал наследetncz от внешнего источника, такого как оператор property или assert.

К последовательности можно обращаться по ее имени, допускается использование иерархического имени.

При описании задержек в последовательностях могут быть использованы операторы повторения:

1) Последовательное повторение (Consecutive repetition ([*)). Обозначает повторение последовательности. Например,

```
a ##1 b [*3] ##1 c   соответствует  a ##1 b ##1 b ##1 b ##1 c
```

Для описания конечного, но неограниченного числа повторений используется символ доллара ($). Например,

```
a ##1 b [*1:$] ##1 c
```

2) Повторение переходом (Goto repetition ([->)). В качестве операндов чаще используются булевы выражения, а не последовательности. Описывает итеративные совпадения булевых выражений по синхрособытиям, которые не обязательно являются последовательными и заканчиваются на последней итерации, возвращающей значение «истина». Например,

```
a ##1 b [->2:10] ##1 c   соответствует  a ##1 ((!b[*0:$] ##1 b) [*2:10]) ##1 c
```

3) Непоследовательное повторение (Nonconsecutive repetition ([=)). Подобно повторению переходом, за единственным отличием, что проверка повторений не обязательна должна заканчиваться на последнем выражении, возвращающем значение „истина”. Например,

```
a ##1 b [=2:10] ##1 c    сообтветствует      a ##1 ((!b [*0:$] ##1 b) [*2:10]) ##1 !b[*0:$] ##1 c
```

10.1.3. Локальные переменные в последовательностях

Последовательности описывают временные взаимоотношения между сигналами. Тем не менее, для того, чтобы описать какое-то сложное поведение, необходимо явно сохранять значения в некоторых точках последовательности, таким образом, чтобы к ней можно обращаться позже. Рассмотрим последовательность, такую как «когда данные поступают в конвейер, они будут переданы наружу через 3-5 синхротактов, и увеличены на 1»

```
sequence s5;
    int d;
   @(posedge clk) (valid, d = data) ##[3:5] (dout == (d+1));
endsequence
```

Метка правильного значения сигнала, когда данные поступают на конвейер, и 3-5 тактов позже, на выходе из конвейера dout, должны быть равны входным данным, увеличенным на 1.

Значение сигнала, поступающее в конвейер, сохраняется, а через 3-5 синхротактов, выход конвейера dout, должен быть равным входному значению, увеличенному на 1. Захват данных, выполненный в последовательности, позволяет избежать построения отдельного автомата или другого вспомогательного кода для захвата данных и необходимости редактирования его для каждой последовательности. Это также позволяет избежать необходимости сообщать другим инструментам, что вспомогательный код не является частью проекта, что упрощает использование таких инструментов.

10.1.4. Событие последовательности (Sequence events)

Копия конструкции sequence может быть использовано в событийном выражении для управления выполнением процедурных операторов, основанных на положительном результате сравнении последовательностей. Это позволяет точке завершения именной последовательности sequence переключать множественные действия в других процессах.

Когда копия последовательности (sequence) описывается в событийном выражении, то процесс, выполняющий событийный контроль должен блокироваться, пока указанная последовательность не достигнет конечной точки.

Последовательность достигает своей точки завершения, всякий раз, когда происходит совпадение всей последовательности. Процесс продолжает выполнение после области наблюдения (Observe region), в которой была обнаружена конечная точка последовательности.

Пример использования последовательности для событийного управления:

```
sequence abc;
    @(posedge clk) a ##1 b ##1 c;
endsequence

program test;
    initial begin
       @ abc $display( "Saw a-b-c" );
       L1 : ...
    end
```

```
endprogram
```

Когда именная последовательность abc достигнет конечной точки, блок initial в программе test будет разблокирован. Выполнится оператор $display, выводящий строку "Saw a-b-c", и следующий за ним оператор с меткой L1. Конечная точка последовательности sequence рассматривается как переключение, являющееся событием.

Для синхронизации используется точка выхода из последовательности, а точка входа игнорируется. Аргументы должны быть статическими, автоматические переменные приведут к ошибке.

10.1.5. Управление с проверкой выполнения последовательности (Level-sensitive sequence controls).

Выполнение процедурного кода может быть приостановлено, пока статус завершения последовательности не примет значение истина. Это реализуется путем использования чувствительного к уровню оператора wait совместно (в конъюнкции) с встроенным методом triggered, который возвращает статус текущего конца именной последовательности (листинг 10.1.). Метод triggered возвращает значение «истина», если данная последовательность достигает конечной точки в определенный момент времени (текущий шаг моделирования), и ложь в противном случае. Статус переключения для последовательности устанавливается в области наблюдения (Observe region) и продолжается в течение оставшегося временного шага (до следующего времени моделирования).

Листинг 10.1. Использование методов последовательностей

```
sequence abc;
    @(posedge clk) a ##1 b ##1 c;
endsequence
sequence de;
    @(negedge clk) d ##[2:5] e;
endsequence
program check;
    initial begin
        wait( abc.triggered || de.triggered );
        if( abc.triggered )
            $display( "abc succeeded" );
        if( de.triggered )
            $display( "de succeeded" );
        L2 : ...
    end
endprogram
```

В представленном примере блок initial проверяет оператором wait завершение последовательности abc или de. Когда, хоть одно из вычисленных выражений примет значение «истина», оператор wait разблокирует процесс, выведет сообщение и продолжит выполнение следующих операторов, начиная с метки L2.

10. 2 Ассерции в SystemVerilog

Ассерции SystemVerilog разработаны для предоставления возможности описания поведения проекта в явной и однозначной манере. Ассерции в основном используются для проверки поведения проекта. Кроме того, ассерции могут быть использованы для получения информации о функциональном покрытии и генерации входных стимулов для проверки. Ассерции реализуются подобно другим блокам проекта и активны для всего моделирования. Программа моделирования отслеживает изменения ассерций, таким образом, чтобы собрать функциональное покрытие данных.

SystemVerilog имеет встроенные эффективные унифицированные ассерции, которые могут быть использованы для моделирования и формальной верификации. Основные преимущества использования заключаются в следующем:

- Легко настраиваются, потому что строятся на подобном языке и синтаксисе.
- Код ассерции небольшой по размеру, благодаря автоматическому концептуальному пониманию управляющей логики проекта
- Простое создание связей между ассерциями и testbench без необходимости специального интерфейса
- Настройка и управление выводимыми сообщениями, возможность представления нескольких уровней ошибок
- Возможность связи между Verilog и Си-функциями
- Возможность избежать рассогласования между моделированием и формальным вычислением с явно определенной семантикой распределения работ

- Возможность повышения производительности процесса верификации.

Семантика SVA определена так, что вычисление асерций гарантирует эквивалентность между моделированием, которое основано на событии (event-based), и асерциями, основанными на синхротакте (cycle-based). Это означает, что различные инструменты будут интерпретировать поведение SVA одним и тем же способом. Кроме того, унификация ассерций с кодом проектирования и верификации, повышает мощность инструмента ассерций, благодаря непрерывному процессу. В частности, SystemVerilog позволяет ассерциям связывать информацию с testbench, а testbench реагировать на состояние ассерций, без использования дополнительного интерфейса.

SystemVerilog поддерживает (имеет) два типа ассерций: прямые (процедурные) и параллельные.

- Прямые ассерции используют семантику моделирования событий и выполняются, как операторы в процедурном блоке. Прямые ассерции, в первую очередь, предназначены для использования в моделировании.
- Параллельные ассерции основаны на семантике синхронизации и захвате значений переменных в определенные моменты времени. Одной из задач ассерций в SystemVerilog является обеспечение общей семантики для ассерций, чтобы они могли быть использованы для управления различными проектами и инструментами верификации.

Реальная мощность SVA, для моделирования и формальной верификации, заключается в возможности краткого описания последовательного поведения и вычисления этих значений в дискретных точках процесса моделирования, обычно по синхротакту. Концепции и компоненты, составляющие параллельные ассерции, могут рассматриваться как множество слоев, из которых каждый строится на лежащем ниже слое (рис 22.3).

Базовая функция ассерций описывает множество ситуаций поведения, которые как ожидается должны иметь значение истины для данного проекта или компонента. Слой булевых выражений является самым базовым и описывает значение элементов в конкретной точке времени, в то время как слой последовательностей строится на булевом и описывает временные взаимосвязи между элементами на протяжении периода времени. Слой свойств использует последовательности для описания реального поведения, а слой ассерций связывает его поведение с инструментами верификации и проектирования.

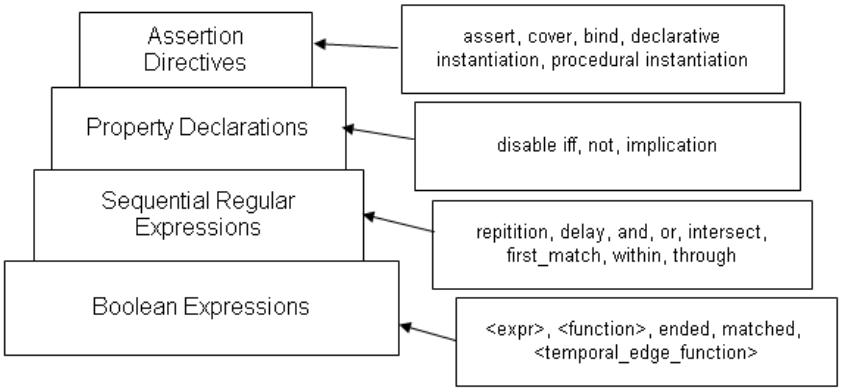


Рис. 10.3 Слои построения ассерций

10.2.1 Выборка значений переменных (Sampling)

Для обеспечения устойчивости между системой моделирования и инструментами формальной верификации, применяющими синхронное моделирование, параллельные ассерции в SystemVerilog используют выборку значений сигналов для вычисления выражений. Эти образцы значений сигналов определены как значения сигналов в конце последнего шага моделирования перед появлением синхрособытия. Таким образом, может быть получен предсказуемый результат, независимо от внутренней реализации очередности и вычисления событий симулятором.

Явное обозначение считывания значений сигналов по отношению к фронту синхросигнала является внутренней частью поведения ассерций SystemVerilog и делается возможным, в частности, потому что ассерции упрощают другую часть языка SystemVerilog. Существующий в Verilog механизм планирования был расширен для поддержки выборки значений сигналов и обеспечения механизмов связи, обеспечивающих качественное взаимодействие ассерций с остальной частью языка. Эти расширения планирования позволяют симулятору и другим инструментам применять существующий алгоритм оптимизации для выполнения ассерций, точно также как для моделирования проекта и TestBench.

Отдельный язык ассерций не может применять этот механизм непосредственного считывания значений, потому что требуемые для этого связи не являются частью Verilog. В некоторых случаях такое поведение может быть аппроксимировано в Verilog, но вероятность появления гонок и мультисинхронные системы делают невозможным гарантирование слаженности между инструментами формальной верификации и моделирования во всех случаях, без выполнения улучшения механизма планирования.

Возможность возникновения гонок – это недостаток семантики планирования событийного моделирования в Verilog, который позволяет нескольким событиям случаться в заданный период моделирования. Поскольку инструменты синтеза и формальной верификации используют синхронный подход к проекту, условия гонок всегда разрешаются согласно той же семантики считывания значений, что используют ассерции в SystemVerilog. Это означает, что ассерции, дают пользователю пост-синтезный взгляд на проект во время до-синтезной RTL-верификации, исключая возможность возникновения различного поведения проекта до и после синтеза.

10.3. Прямые (процедурные) ассерции

Прямые ассерции используются в процедурном коде в любом месте в конструкциях initial и always.

Синтаксис прямых асерций:

```
assert ( expression ) [[ pass_stmt ] else fail_stmt ];
```

Выражение вычисляется сразу, при выполнении оператора. Операторы в ветвях pass и fail, если они присутствуют, выполняются мгновенно после вычисления выражения.

Процедурный код testbench может проверять значения сигналов проекта, переменных и выполнять необходимые действия, в случае обнаружения ошибки. Например, в проекте выполняется проверка запроса шины request и ожидается, что ответ будет получен двумя тактами позже. Тестирование этой ситуации можно выполнить с помощью оператора if.

// Проверка сигнала с помощью оператора if

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

Оператор ассерции более компактен, чем оператор if. При этом значение условия обратно, записываемому в if. Другими словами, если условие имеет значение «истина», то выполнение ассерции было успешно.

```
// Простая процедурная асерции
bus.cb.request <= 1;
@bus.cb;
a1: assert (bus.cb.grant == 2'b01);
// остальной код теста
```

Если сигнал grant получен правильно (grant=2'b01), работа теста будет продолжено. Иначе будет выдано следующее сообщение.

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

Сообщение говорит, что в линии 7 файла test.sv, асерция top.t1.a1 начала выполнение в момент 55ns для проверки сигнала bus.cb.grant и сразу же обнаружила ошибку.

Процедурные асерции имеют ветви then- и else-, при необходимости, в них могут быть добавлены операторы.

```
// Создание пользовательского сообщения об ошибке
a1: assert (bus.cb.grant == 2'b01)
    else $error("Grant not asserted");
Если сигнал grant не имеет ожидаемого значения, то будет выведено соответствующее сообщение.
"test.sv", 7: top.t1.a1: started at 55ns failed at 55ns
Offending '(bus.cb.grant == 2'b1)'
Grant not asserted
```

SystemVerilog имеет четыре функции для вывода сообщений: $info, $warning, $error и $fatal, которые могут быть использованы только в операторе ассерции. Можно использовать ветвь then для записи количества успешных выполнений оператора ассерций.

```
// Создание пользовательского сообщения об ошибке
a1: assert (bus.cb.grant == 2'b01)
        grants_received++;              // Количество успешных выполненийт
    else
        $error("Grant not asserted");
```

Операторы прямых ассерций выполняют тестирование указанных условий, в момент своего выполнения в процедурном коде. Выражение интерпретируется тем же способом, что и условие в операторе if. Если значение выражения равно X, Z или 0, оно рассматривается, как «ложь» и ассерция провалилась (fail), иначе – «истинна» и ассерция прошла (pass).

Оператор, связанный с успешным выполнением ассерции записывается первым, называется pass-оператором и выполняется, если значение выражения имеет значение «истина». Pass-оператор может, например, записывать число успешных проходов в файл отчета (log), но может вообще отсутствовать. Если оператор для ветви успешного выполнения ассерций не указан, то никакие действия в этом случае выполняться не будут. Оператор, ассоциирующийся с веткой else, называется fail-оператор и выполняется, если значение выражения «ложь». Присутствие этого оператора также является необязательным. Указанные действия выполняются мгновенно, после вычисления ассерции.

Необязательная метка оператора создает именную конструкцию, и она может быть выведена с помощью символа форматирования %m. Например,

```
assert_foo : assert(foo) $display("%m passed"); else $display("%m failed");
```

Поскольку ассерция – это оператор, который может принимать значение «ложь», то падение ассерции должно иметь связанный с ним уровень серьезности. По умолчанию, такой уровень серьезности равен error. Задать или явно указать его можно, включая одну из следующих системных задач:

```
$fatal  - фатальная ошибка.
$error  - ошибка времени выполнения.
$warning – предупреждение времени выполнения, о которое может быть обработано в манере,
определяемой системой моделирования.
$info  - означает, что ассерции не назначен определенный уровень серьезности.
```

Все эти задачи должны выводить сообщения, обозначающие уровень серьезности падения ассерции и информацию, которая должна включать следующее:

- Имя файла и номер линии оператора ассерции.
- Иерархическое имя оператора ассерции, если он имеет имя или границы видимости ассерции, если метки нет.

Для инструментов моделирования, эти задачи должны сообщать время моделирования, в которое она вызывалась. Каждая системная задача может также включать дополнительную описываемую пользователем информацию, используя тот же самый формат представления данных, что и оператор Verilog $display. Если ветвь else содержит более чем одну из этих системных задач, то каждая из них должна быть описана указанным способом. В примере задача уровня серьезности выполняется не в момент падения ассерции, но тем не менее выводит реальное время обнаружения ошибки.

```
time t;
always @(posedge clk)
    if (state == REQ)
      assert (req1 || req2)
            else begin
                  t = $time;
                   #5 $error("assert failed at time %0t",t);
            end
```

Если ассерция падает в момент времени 10, сообщение об ошибке будет выведено в момент времени 15, но при этом содержать оно будет “assert failed at time 10”.

Вывод информационных сообщений и предупреждений, может контролироваться опциями, формируемыми командами. Поскольку fail-операторы, также как и pass-операторы, является легальными процедурными операторами SystemVerilog, то они могут быть также использованы для формирования условий выполнения ассерций в другой части testbench.

```
assert (myfunc(a,b)) count1 = count + 1; else ->event1;
assert (y == 0) else flag = 1;
```

Листинг Пример прямой ассерции

```
//++++++++++++++++++++++++++++++++++++++++++++++++
// DUT With Immediate assertions
//++++++++++++++++++++++++++++++++++++++++++++++++
module assert_immediate();

reg  clk, grant, request;
time current_time;
```

```
initial begin
  clk = 0;
  grant   = 0;
  request = 0;
  #4 request = 1;
  #4 grant = 1;
  #4 request = 0;
  #4 $finish;
end

always #1 clk = ~clk;
//================================================
// Assertion used in always block
//================================================
always @ (posedge clk)
begin
  if (grant == 1) begin
    CHECK_REQ_WHEN_GNT : assert (grant && request) begin
      $display ("Seems to be working as expected");
    end else begin
      current_time = $time;
      #1 $error("assert failed at time %0t", current_time);
    end
  end
end

endmodule
```

Результат моделированя

10.4. Параллельные ассерции

Другой тип ассерций – параллельные ассерции, выполняют проверку сигнала в течение всего процесса моделирования. При описании таких ассерций используется синхронизация.

Параллельные ассерции отличаются от прямых двумя важными аспектами. Во-первых, в дополнение к прямой записи в проекте, подобно операторам в блоках always или initial, параллельные ассерции могут быть реализованы декларативно, на уровне непрерывных операторов присвоения и реализации модулей (подобно непрерывному присвоению) за границами процедурных блоков. Второе отличие заключается в том, что параллельные ассерции позволяют описывать для проверки системы временное поведение, вместо комбинационных условий, как в прямых ассерциях

Синтаксис:

```
assert property ( sequential_expr_or_property )
        [[ pass_stmt ] else fail_stmt ]
```

Ключевое слово property отличает параллельную ассерцию от прямой ассерции. Последовательное выражение вычисляется с использованием значений сигналов и позволяет операторам pass/fail связываться с testbench. Поскольку ассерции являются внутренней частью языка, эти операторы могут использовать SystemVerilog в полном объеме для переключения событий, записи информации о покрытии, или других воздействий на порядок кода верификации, включая вызов Си-функций. Отдельный язык ассерций эффективен только в режиме “read-only”, позволяет наблюдать за поведением проекта, но не допускает элементов воздействия ни на проект, ни на testbench.

В следующем примере ассерция выполняет проверку, что сигнал арбитра не принимает значение X или Z, кроме ситуации инициализации устройства, когда reset=1 (листинг 10.2).

Листинг 10.2 Параллельная ассерция для проверки значения X/Z

```
interface arb_if(input bit clk);
    logic [1:0] grant, request;
    logic reset;

    property request_2state;
        @(posedge clk) disable iff (reset) $isunknown(request) == 0;
    endproperty

    assert_request_2state: assert property (request_2state);
endinterface
```

Параллельные ассерции описывают поведение, которое охватывает большой промежуток времени. В отличие от прямых ассерций, обработка модели основывается на сихронизации, таким образом, параллельная ассерция вычисляется только при появлении события синхронизации. При этом используется выборка значений переменных. Таким образом, независимо от внутреннего механизма обработки событий симулятором в результате вычислений может быть получен предсказуемый результат. Эта исполнимая модель соответствует также синтезируемой модели аппаратной реализации RTL-описание.

Значения переменных, используемых в ассерциях, собираются в подготовительный период (Preponed) временного слота, ассерции вычисляются в период наблюдения (Observe). Если переменная, используемая в ассерции, является входной переменной блока синхронизации, то ее значение считывается # 1step управлением. Любые другие типы выборок для переменных блоков синхронизации приведут к ошибке. Ассерции, использующие переменные блоков синхронизации, не создают свой способ выборки значений, а применяют указанный в блоках синхронизации

Временные модели, используемые в спецификациях параллельных ассерций, основаны на синхрособытиях и используют обобщенное понятие синхротакта. Определение синхронизации, как правило, создается пользователем и может изменяться от одного выражения к другому. Событие синхронизации - это отдельный момент времени, который сам не имеет длительности. Событие синхронизации может иметь место только один раз в течение любого времени моделирования и значения выборки сигналов для этого периода моделирования используются для вычисления параллельных ассерций. Рис. 10.4 представляет значения переменной, при изменении синхронизации. Значение сигнала req равны нулю в момент событий синхронизации 1 и 2. В событие 3, значение выборки переключается в единицу и остается в таком значении до синхрособытия 6. Значение переменной req в момент события 6 становится равным нулю, и остается таким для событий 7, 8 и 9. Синхрособытие 9 происходит вместе с переключением сигнала req, однако значение выборки для этого момента остается равным нулю, поскольку согласно семантике языка, значение считывается за один шаг(1step) до появления события.

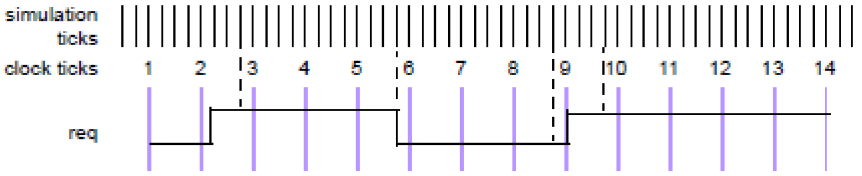


Рис. 10.4. Периоды получения значений

Выражение, используемое в ассерциях, всегда связано с определением синхронизации. Значения выборки используемые для вычисления изменяемых или булевых выражений или подвыражений, которые необходимы для обнаружения последовательности.

Для параллельных ассерций применяются следующие выражения:

- Важно гарантировать свободное от сбоев определение поведения синхросигнала. В противном случае могут быть получены ошибочные значения выборок.

- Если переменная, которая появляется в выражении синхронизации, также появляется в выражении с ассерцией, то ее значение в этих двух случаях может быть различным. Текущее значение переменной используется в выражении синхронизации, в то время как значение выборки переменной - внутри ассерции.

Выражения синхронизации, которое контролирует вычисление последовательности, может быть более сложным, чем просто имя одного сигнала. Для представления синхронизации могут быть использованы выражения вида (clk && gating_signal) и (clk iff gating_signal). Возможны другие, более сложные выражения. Однако, в целях обеспечения правильного поведение системы и соответствия как можно ближе семантике синхронного моделирования, сигналы в выражениях синхронизации должны быть свободны от рисков сбоев и должны иметь только одно переключение. Пример параллельной ассерции приведен ниже:

```
base_rule1: assert property (cont_prop(rst,in1,in2))
                $display("%m, passing");
            else $display("%m, failed");
```

Свойство может выполнять проверку одной или нескольких последовательностей, стартующих в различное время. Вычисление последовательности начинается с поиска совпадения начала последовательности на заданном синхрособытии. Чтобы определить наличие совпадения, соответствующие булевы выражения вычисляются, начиная с определенного синхрособытия и это продолжается при каждом последующем

синхрособытии, пока выражения имеют значение «истина». Если выражение примет значение «ложь», это означает, что совпадения нет.

10.5. Свойства (Properties)

Во многих случаях последовательности позволяют описывать поведение системы. Задавать проверку поведения можно с помощью свойств. Кроме того, свойства позволяют инвертировать смысл последовательности (т.е. последовательность не должна случиться), отключить вычисление последовательности, или применить последовательность к другим событиям.

Для определения свойств используется следующий синтаксис:

```
// Декларация свойств
property name [ ( formal_item {, formal_item } ) ] ;
        { assertion_variable_declaration }
    property_spec ;
endproperty [:name]
```

Описание свойств состоит из последовательностей или импликации последовательностей и других свойств.

```
property_spec ::=
    [clocking_event] [disable iff( expression )] [ not ] property_expr
property_expr ::= sequence_expr | implication_expr
```

Определение свойств выполняется в границах property-endproperty, и поддерживают формальные аргументы и аргумент, передающий точное поведение последовательности.

Например:

```
property p1;
    @(posedge clk) disable iff (test) not abort_seq;
endproperty
```

Оператор not инвертирует смысл последовательности abort_seq, таким образом, выражение будет иметь значение «ложь» при появлении последовательности abort_seq. Выражение disable iff отключает вычисление последовательности, если и только, если сигнал test=1. Обратите внимание на повторное использование ключевых слов disable и iff, которые применяются в других конструкциях SystemVerilog. Это свойство должно интерпретироваться как «пока сигнал test=0, проверять отсутствие появления последовательности abort_seq».

Другая полезная поведенческая концепция – «это исключается» (that of implication). Пусть необходимо проверить появление последовательности, но только после появления init_seq. Такое свойство может быть описано следующим образом:

```
@(posedge clk) init_seq |=> abort_seq;
```

где |=> неперекрывающийся оператор импликации. Это выражение свойств утверждает, что каждая успешно завершенная последовательность init_seq первого операнда подразумевает, что последовательность abort_seq из второго операнда начнется на следующем синхротакте. Для каждого такта, в котором успешное завершение последовательности первого операнда не присутствует, последовательность второго операнда не будет вычислена и свойство рассматривается как «истина».

Свойство определяет поведение проекта. Свойство может быть использовано для верификации, как спецификация предположения, проверка или покрытие. Для того чтобы использовать поведение для верификации, используются операторы assert, assume, или cover. Декларация свойства само по себе не производит результат.

Свойства property, как и последовательности, декларируется с необязательными формальными аргументам. Когда свойство вычисляется, то в него передаются фактические аргументы. Механизм передачи аргументов в свойства такой же, как и для передачи аргументов в последовательность. Последовательность получает расширение с реальными аргументами, заменяющими формальные аргументы.

Результат вычисления свойств либо «истина», либо «ложь». Существует семь видов свойств: последовательность (sequence), отрицание (negation), дизъюнкция(disjunction), конъюнкция (conjunction), if...else, импликация (implication) и создание копии (instantiation).

а) свойство-последовательность является последовательностью, вычисляемой в значение «истина», если и только если существует непустое совпадение последовательности. Последовательность, которая допускает пустое совпадение не разрешается для свойств. Поскольку совпадение существует, если и только если существует первое совпадение. Вычисление такого свойства - полное трансформирования его sequence_expr в first_match (sequence_expr). Как только первое совпадение sequence_expr обнаруживается, вычисление свойства рассматривается как «истина» и нет необходимости в поиске других совпадение.

б) Свойство-отрицание, имеет вид

```
not property_expr
```

При каждой попытке вычисления свойств вычисляется property_expr. Ключевое слово not обозначает, что при вычислении свойства возвращается противоположное значение для property_expr. Таким образом, если property_expr примет значение «истина», то not property_expr соответствует значению «ложь», и если property_expr оценивается как «ложь», то not property_expr будет «истина».

c) Дизъюнкция свойств имеет вид

```
property_expr1 or property_expr2
```

Свойство соответствует значению «истина» если и только если как минимум одно из свойств property_expr1 и property_expr2 «истина».

d) Конъюнкция свойств имеет форму

```
property_expr1 and property_expr2
```

Свойство оценивается как «истина», если и только если свойства property_expr1 и property_expr2 имеют значение «истина»

e) Свойство « if...else» имеет форму

```
if (expression_or_dist) property_expr1
```

или

```
if (expression_or_dist) property_expr1 else property_expr2
```

Свойство в первом случае оценивается как «истина», если и только если либо expression_or_dist принимает значение «ложь», либо property_expr1 – «истина». Во втором случае свойство возвращает значение истина, когда либо expression_or_dist и property_expr1 возвращают значение «истина», либо expression_or_dist вычисляется в «ложь», а property_expr2 – «истина».

f) Свойство-импликанта имеет форму

```
sequence_expr |-> property_expr
```

или

```
sequence_expr |=> property_expr
```

Копия именного свойства может быть использована в выражении property_expr or property_spec. В общем случае, использование копии является законным, если тело именного свойства property_spec может заменить его копию в property_expr or property_spec, подставляя реальные параметры, вместо формальных аргументов, и игнорируя локальные переменные. Так, например, если копия именного свойства используется как операнд property_expr в любом формирующем свойство операторе, то это свойство не должно включать выражение disable iff. Аналогичным образом, события синхронизации в именных свойствах должно соответствовать правилам мультисинхронной поддержки, в случае если свойство входит в выражение property_expr or property_spec, которое также управляется другим синхрособытием. Таблица 22.2 представляет списки операторов для последовательностей и свойств, начиная с самого высокого приоритета, и показывает сочетаемость для неунарных операторов.

Выражение disable iff может быть подключено к property_expr , чтобы сформировать property_spec.

```
disable iff (expression_or_dist) property_expr
```

Выражение disable iff называется выражением сброса (reset expression) и позволяет описывать приоритетный сброс. Для вычисления property_spec, оцениваются составляющие его property_expr. Если до завершения такого вычисления выражение сброса принимает значение «истина», то значение всего свойства property_spec – «истина». В противном случае, значение property_spec зависит от результата вычисления property_expr. Сброс выражения проверяется независимо для различных попыток вычисления property_spec. Значения переменных, используемых в выражении сброса, являются значениями переменных в текущем цикле моделирования, а не их выборками.

Выражения могут содержать ссылку на конечную точку последовательности с помощью метода переключения этой последовательности. Методы сравнение и обнаружения конца последовательности и локальные переменные не могут быть использованы в выражении сброса. Если значение выборки функции используется в выражении сброса, то синхронизация выборки должна быть явно описана в его списке актуальных аргументов.

Таблица 10.2 Операторы последовательностей и свойств предшественников и связи

Sequence operators	Property operators	Associativity
[*], [=], [->]		--
##		Left
throughout		Right
within		Left
intersect		Left
	not	--
and	and	Left
or	or	Left
	if...else	Right
	|->, |=>	Right

Свойства сами по себе никогда не вычисляются для проверки выражений. Их используют в операторе верификации, который устанавливает одну из следующих функций верификации:

- assert для описания свойств в качестве проверки, чтобы обеспечить, что свойство имеет место в проекте

- assume, для описания свойств в качестве предположения для среды

- cover для мониторинга вычисления свойств покрытия

10.6 Системные функции, используемые в ассерциях

Ассерции обычно используются для вычисления некоторых специфических особенностей реализации проекта, например, что определенный сигнал содержит только одну единицу. Следующие системные функции введены для описания функциональности ассерции:

- $onehot (<expression>) возвращает значение «истина», если только один бит выражения равен 1. $onehot0 (<expression>) возвращает значение «истина», если как минимум один бит выражения равен 1
- $isunknown (<expression>) возвращает значение «истина», если как любой один бит выражения равен X или Z. Это эквивалентно ^ <expression> === 'bx.

Все вышеперечисленные системные функции системы возвращают значение типа bit. Возвращение значения 1'b1 соответствует «истина», а значение 1'b0 – «ложь». Другая полезная функция, предоставляемая для логического выражения $countones, подсчитывает количество единиц в битовом векторе.

- $countones(expression)

Значения X и Z не засчитывается в число единиц.

Дополнение:

Параллельные ассерции имеют несколько уровней:

- Boolean
- Sequences
- Property
- Property directive

Листинг Пример параллельной ассерции

```
//+++++++++++++++++++++++++++++++++++++++++++++++++
//   DUT With assertions
//+++++++++++++++++++++++++++++++++++++++++++++++++
module concurrent_assertion(
  input wire clk,req,reset,
  output reg gnt);
//=================================================
// Sequence Layer
//=================================================
sequence req_gnt_seq;
  // (~req & gnt) and (~req & ~gnt) is Boolean Layer
```

```
  (~req & gnt) ##1 (~req & ~gnt);
endsequence
//================================================
// Property Specification Layer
//================================================
property req_gnt_prop;
  @ (posedge clk)
    disable iff (reset)
      req |-> req_gnt_seq;
endproperty
//================================================
// Assertion Directive Layer
//================================================
req_gnt_assert : assert property (req_gnt_prop)
                 else
                 $display("@%0dns Assertion Failed", $time);
//================================================
// Actual DUT RTL
//================================================
always @ (posedge clk)
  gnt <= req;

endmodule

//++++++++++++++++++++++++++++++++++++++++++++++++
//   Testbench Code
//++++++++++++++++++++++++++++++++++++++++++++++++
module concurrent_assertion_tb();

reg clk = 0;
reg reset, req = 0;
wire gnt;

always #3 clk ++;

initial begin
  reset <= 1;
  #20 reset <= 0;
  // Make the assertion pass
  #100 @ (posedge clk) req  <= 1;
  @ (posedge clk) req <= 0;
  // Make the assertion fail
  #100 @ (posedge clk) req  <= 1;
  repeat (5) @ (posedge clk);
  req <= 0;
  #10 $finish;
end

concurrent_assertion dut (clk,req,reset,gnt);

endmodule
```

Листинг. Пример использования оператора ##

```
//++++++++++++++++++++++++++++++++++++++++++++++++
//   DUT With assertions
//++++++++++++++++++++++++++++++++++++++++++++++++
module hash_sequence();

logic clk = 0;
always #1 clk ++;

logic [2:0] req,gnt;
logic reset;
//================================================
// Shows basic usage of ## delay operator
//================================================
sequence req_gnt_1clock_seq;
  req[0] ##1 gnt[0];
endsequence
//================================================
// Shows range usage of ## delay operator
//================================================
```

```
sequence req_gnt_3to5clock_seq;
  req[1] ##[3:5] gnt[1];
endsequence
//================================================
// Shows zero delay usage of ## delay operator
//================================================
sequence req_gnt_0clock_seq;
  req[2] ##0 gnt[2];
endsequence
//================================================
// Shows sequence called within sequence
//================================================
sequence master_seq;
  req_gnt_1clock_seq ##1 req_gnt_3to5clock_seq ##1 req_gnt_0clock_seq;
endsequence
//================================================
//  Declare property for each of sequence
//  We may use more then one seuqnce in a property
//================================================
property req_gnt_1clock_prop;
  @ (posedge clk)
  disable iff (reset)
     req[0] |-> req_gnt_1clock_seq;
endproperty

property req_gnt_3to5clock_prop;
  @ (posedge clk)
  disable iff (reset)
     req[1] |-> req_gnt_3to5clock_seq;
endproperty

property req_gnt_0clock_prop;
  @ (posedge clk)
  disable iff (reset)
     req[2] |-> req_gnt_0clock_seq;
endproperty

property master_prop;
  @ (posedge clk)
  disable iff (reset)
     req[0] |-> master_seq;
endproperty

//================================================
// Assertion Directive Layer
//================================================
req_gnt_1clock_assert    : assert property (req_gnt_1clock_prop);
req_gnt_3to5clock_assert : assert property (req_gnt_3to5clock_prop);
req_gnt_0clock_assert    : assert property (req_gnt_0clock_prop);
master_assert            : assert property (master_prop);

//================================================
// Drive the input vectors to test assetion
//================================================
initial begin
  // Init all the values
  reset  <= 0;
  for (int i = 0; i < 3; i++) begin
    req[i] <= 0;
    gnt[i] <= 0;
  end
  @ (posedge clk);
  req[0] <= 1;
  @ (posedge clk);
  gnt[0] <= 1;
  req[0] <= 0;
  @ (posedge clk);
  gnt[0] <= 0;
  req[1] <= 1;
  @ (posedge clk);
  req[1] <= 0;
  repeat(3) @ (posedge clk);
```

```
    gnt[1] <= 1;
    @ (posedge clk);
    gnt[1] <= 0;
    req[2] <= 1;
    gnt[2] <= 1;
    @ (posedge clk);
    req[2] <= 0;
    gnt[2] <= 0;
    // Cause assertion to fail
    @ (posedge clk);
    req[0] <= 1;
    repeat(2) @ (posedge clk);
    gnt[0] <= 1;
    req[0] <= 0;
    #30 $finish;
  end

  endmodule
```

Результат моделирования

** Error: Assertion error.

Time: 23 ps Started: 21 ps Scope: hash_sequence.req_gnt_1clock_assert File: E:/Temp/BRAIN/assert1.sv Line: 66

** Error: Assertion error.

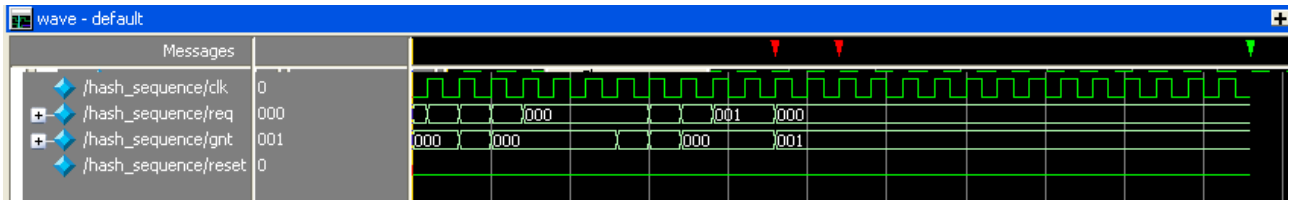
Time: 23 ps Started: 21 ps Scope: hash_sequence.master_assert File: E:/Temp/BRAIN/assert1.sv Line: 69

** Error: Assertion error.

Time: 27 ps Started: 23 ps Scope: hash_sequence.master_assert File: E:/Temp/BRAIN/assert1.sv Line: 69

** Note: $finish   : E:/Temp/BRAIN/assert1.sv(106)

Time: 53 ps Iteration: 0 Instance: /hash_sequence



Оператор $

Листинг. Пример оператора $

```
//+++++++++++++++++++++++++++++++++++++++++++++++++
//   DUT With assertions
//+++++++++++++++++++++++++++++++++++++++++++++++++
module unbound_assertion();
logic clk = 0;
logic req,reset, gnt;

always #1 clk ++;

//=================================================
// Sequence Layer
//=================================================
sequence req_gnt_seq;
  req ##[1:$] gnt;
endsequence
//=================================================
// Property Specification Layer
//=================================================
property req_gnt_prop;
  @ (posedge clk)
    disable iff (reset)
      req |-> req_gnt_seq;
```

```
endproperty
//================================================
// Assertion Directive Layer
//================================================
req_gnt_assert : assert property (req_gnt_prop);

initial begin
  $display("@%0dns Asserting reset",$time);
  reset <= 1;
  req <= 0;
  gnt <= 0;
  $display("@%0dns Deasserting reset",$time);
  #20 reset <= 0;
  // Make the assertion pass
  #100 @ (posedge clk) req  <= 1;
  $display("@%0dns Asserting req",$time);
  repeat (100) @(posedge clk);
  @ (posedge clk) req <= 0;
  $display("@%0dns Deasserting req",$time);
  repeat (2) @ (posedge clk);
  $display("@%0dns Asserting gnt",$time);
  gnt <= 1;
  @ (posedge clk) gnt <= 0;
  $display("@%0dns Deasserting gnt",$time);
  #100 @ (posedge clk) req  <= 1;
  $display("@%0dns Asserting req",$time);
  repeat (5) @ (posedge clk);
  req <= 0;
  $display("@%0dns Deasserting req",$time);
  #100 $finish;
end

endmodule
```

Операторы повторения

Repetition Operators

When a sequence needs to be tested for repetion, then repetition operators are used. They are of three types as below. The min value can be less then 0 and max value as sees in earlier page is $.

* Consecutive repetition ([*): Consecutive repetition specifies finitely many iterative matches of the operand sequence, with a delay of one clock tick from the end of one match to the beginning of the next. The overall repetition sequence matches at the end of the last iterative match of the operand.

* Goto repetition ([->): Goto repetition specifies finitely many iterative matches of the operand boolean expression, with a delay of one or more clock ticks from one match of the operand to the next successive match and no match of the operand strictly in between. The overall repetition sequence matches at the last iterative match of the operand.

* Nonconsecutive repetition ([=): Nonconsecutive repetition specifies finitely many iterative matches of the operand boolean expression, with a delay of one or more clock ticks from one match of the operand to the next successive match and no match of the operand strictly in between. The overall repetition sequence

Empty Sequence : An empty sequence is one that does not match over any positive number of clock ticks. The following rules apply for concatenating sequences with empty sequences. An empty sequence is denoted as empty, and a sequence is denoted as seq.

* (empty ##0 seq) does not result in a match.
* (seq ##0 empty) does not result in a match.
* (empty ##n seq), where n is greater than 0, is equivalent to (##(n-1) seq).
* (seq ##n empty), where n is greater than 0, is equivalent to (seq ##(n-1) 'true).

Листинг Пример [*

```
//+++++++++++++++++++++++++++++++++++++++++++++++++
//   DUT With assertions
//+++++++++++++++++++++++++++++++++++++++++++++++++
module repetition_assertion();

logic clk = 0;
always #1 clk ++;

logic req,busy,gnt;
//=================================================
// Sequence Layer
//=================================================
sequence boring_way_seq;
  req ##1 busy ##1 busy ##1 gnt;
endsequence

sequence cool_way_seq;
  req ##1 busy [*2] ##1 gnt;
endsequence

sequence range_seq;
  req ##1 busy [*1:5] ##1 gnt;
endsequence

//=================================================
// Property Specification Layer
//=================================================
property boring_way_prop;
  @ (posedge clk)
      req |-> boring_way_seq;
endproperty

property cool_way_prop;
  @ (posedge clk)
      req |-> cool_way_seq;
endproperty
//=================================================
// Assertion Directive Layer
//=================================================
boring_way_assert : assert property (boring_way_prop);
cool_way_assert   : assert property (cool_way_prop);

//=================================================
// Generate input vectors
//=================================================
initial begin
  req <= 0; busy <= 0;gnt <= 0;
  @ (posedge clk);
  req <= 1;
  @ (posedge clk);
  busy <= 1;
  req  <= 0;
  repeat(2) @ (posedge clk);
  busy <= 0;
  gnt <= 1;
  @ (posedge clk);
  gnt <= 0;
```

```
  // Now make the assertion fail
  req <= 0; busy <= 0;gnt <= 0;
  @ (posedge clk);
  req <= 1;
  @ (posedge clk);
  busy <= 1;
  req  <= 0;
  repeat(3) @ (posedge clk);
  busy <= 0;
  gnt <= 1;
  @ (posedge clk);
  gnt <= 0;
  #30 $finish;
end

endmodule
```

Листинг Пример [->

```
//++++++++++++++++++++++++++++++++++++++++++++++++++
//   DUT With assertions
//++++++++++++++++++++++++++++++++++++++++++++++++++
module goto_assertion();

logic clk = 0;
always #1 clk ++;

logic req,busy,gnt;
//==================================================
// Sequence Layer
//==================================================
sequence boring_way_seq;
  req ##1 ((!busy[*0:$] ##1 busy) [*3]) ##1 gnt;
endsequence

sequence cool_way_seq;
  req ##1 (busy [->3]) ##1 gnt;
endsequence

//==================================================
// Property Specification Layer
//==================================================
property boring_way_prop;
  @ (posedge clk)
      req |=> boring_way_seq;
endproperty

property cool_way_prop;
  @ (posedge clk)
      req |=> cool_way_seq;
endproperty
//==================================================
// Assertion Directive Layer
//==================================================
boring_way_assert : assert property (boring_way_prop);
cool_way_assert   : assert property (cool_way_prop);

//==================================================
// Generate input vectors
//==================================================
```

```
initial begin
  // Pass sequence
  gen_seq(3,0);
  repeat (20) @ (posedge clk);
  // Fail sequence (gnt is not asserted properly)
  gen_seq(3,1);
  // Terminate the sim
  #30 $finish;
end
//================================================
/// Task to generate input sequence
//================================================
task  gen_seq (int busy_delay,int gnt_delay);
  req <= 0; busy <= 0;gnt <= 0;
  @ (posedge clk);
  req <= 1;
  @ (posedge clk);
  req  <= 0;
  repeat (busy_delay) begin
   @ (posedge clk);
    busy <= 1;
   @ (posedge clk);
    busy <= 0;
  end
  repeat (gnt_delay) @ (posedge clk);
  gnt <= 1;
  @ (posedge clk);
  gnt <= 0;
endtask

endmodule
```

Листинг Пример [=

```
//++++++++++++++++++++++++++++++++++++++++++++++++++
//   DUT With assertions
//++++++++++++++++++++++++++++++++++++++++++++++++++
module noncon_assertion();

logic clk = 0;
always #1 clk ++;

logic req,busy,gnt;
//================================================
// Sequence Layer
//================================================
sequence boring_way_seq;
  req ##1 ((!busy[*0:$] ##1 busy) [*3]) ##1 !busy[*0:$] ##1 gnt;
endsequence

sequence cool_way_seq;
  req ##1 (busy [=3]) ##1 gnt;
endsequence

//================================================
// Property Specification Layer
//================================================
property boring_way_prop;
  @ (posedge clk)
```

```
      req |-> boring_way_seq;
endproperty

property cool_way_prop;
  @ (posedge clk)
      req |-> cool_way_seq;
endproperty
//================================================
// Assertion Directive Layer
//================================================
boring_way_assert : assert property (boring_way_prop);
cool_way_assert   : assert property (cool_way_prop);

//================================================
// Generate input vectors
//================================================
initial begin
  // Pass sequence
  gen_seq(3,0);
  repeat (20) @ (posedge clk);
  // This was fail in goto, but not here
  gen_seq(3,1);
  repeat (20) @ (posedge clk);
  gen_seq(3,5);
  // Lets fail one
  repeat (20) @ (posedge clk);
  gen_seq(5,5);
  // Terminate the sim
  #30 $finish;
end
//================================================
/// Task to generate input sequence
//================================================
task  gen_seq (int busy_delay,int gnt_delay);
  req <= 0; busy <= 0;gnt <= 0;
  @ (posedge clk);
  req <= 1;
  @ (posedge clk);
  req  <= 0;
  repeat (busy_delay) begin
   @ (posedge clk);
    busy <= 1;
   @ (posedge clk);
    busy <= 0;
  end
  repeat (gnt_delay) @ (posedge clk);
  gnt <= 1;
  @ (posedge clk);
  gnt <= 0;
endtask

endmodule
```

Системные задачи

Function $sampled returned the sampled value of a expression with respect to last clock event. When $sampled is invoked prior to the occurrence of the first clocking event, the value of X is returned. The use of $sampled in assertions, although allowed, is redundant, as the result of the

function is identical to the sampled value of the expression itself used in the assertion. But following types are usefull

- $rose returns true, when least significant bit changes to 1.
- $fell returns true, when least significant bit changes to 0.
- $stable returns true, if value did not change since last clock event to current clock event.
- $past returns value n clock events before.

Пример использования системных задач

```
module system_assertion();

logic clk = 0;
always #1 clk ++;

logic req,gnt;
//-------------------------------------------------
// Property Specification Layer
//-------------------------------------------------
property system_prop;
  @ (posedge clk)
      ($rose(req) && $past(!req,1)) |=>
         ($rose(gnt) && $past(!gnt,1));
endproperty
//-------------------------------------------------
// Assertion Directive Layer
//-------------------------------------------------
system_assert : assert property (system_prop);
//-------------------------------------------------
// Generate input vectors
//-------------------------------------------------
initial begin
  req <= 0;gnt <= 0;
  repeat(10) @ (posedge clk);
  req <= 1;
  @( posedge clk);
  gnt <= 1;
  req <= 0;
  @( posedge clk);
  // Make the assertion fail now
  req <= 0;gnt <= 0;
  repeat(10) @ (posedge clk);
  req <= 1;
  @( posedge clk);
  req <= 0;
  gnt <= 0;
  // Terminate the sim
  #30 $finish;
end

endmodule
```

Binary Operators

Binary operators take two operands or two sequence and produce a new sequence. Following are binary sequences operators.

- **and** : When both sequence are expected to match, and both the sequence start at same time, but are not expected to finish at same time. Then **and** operator is used.
- **intersect** : When both sequence are expected to match, and both the sequence start and end at same time. Then **intersect** operator is used.
- **or** : When one of the both sequence are expected to match. Then **or** operator is used.

Example : Binary Operators

```
//+++++++++++++++++++++++++++++++++++++++++++++++++
//   DUT With assertions
//+++++++++++++++++++++++++++++++++++++++++++++++++
module binary_assertion();

logic clk = 0;
always #1 clk ++;

logic req,gnt,busy;
//=================================================
// Sequence Specification Layer
//=================================================
sequence s1;
 $past(!req,1) ##1 ($rose(gnt) and $past(!gnt,1));
endsequence

sequence s2;
 $past(!req,1) ##1 $rose(gnt) ##1 $fell(gnt);
endsequence

sequence s3;
  req ##1 gnt ##1 busy ##1 (!req and !gnt and !busy);
endsequence

sequence s4;
  req ##[1:3] gnt;
endsequence
//=================================================
// Property Specification Layer
//=================================================
property and_prop;
  @ (posedge clk)
    req |-> s1 and s2;
endproperty

property intersect_prop;
  @ (posedge clk)
    req |-> s1 intersect s3;
endproperty

property or_prop;
  @ (posedge clk)
```

```
    req |-> s1 or s4;
endproperty
//================================================
// Assertion Directive Layer
//================================================
and_assert       : assert property (and_prop);
intersect_assert : assert property (intersect_prop);
or_assert        : assert property (or_prop);
//================================================
// Generate input vectors
//================================================
initial begin
  req <= 0;gnt <= 0;busy<=0;
  repeat(10) @ (posedge clk);
  req <= 1;
  @( posedge clk);
  gnt <= 1;
  req <= 0;
  @( posedge clk);
  busy <= 1;
  // Make the assertion fail now
  // OR will not fail
  req <= 0;gnt <= 0; busy <= 0;
  repeat(10) @ (posedge clk);
  req <= 1;
  repeat (2) @( posedge clk);
  req <= 0;
  gnt <= 1;
  @( posedge clk);
  @( posedge clk);
  req <= 0;gnt <= 0; busy <= 0;
  // Terminate the sim
  #30 $finish;
end

endmodule
```

Match Operators

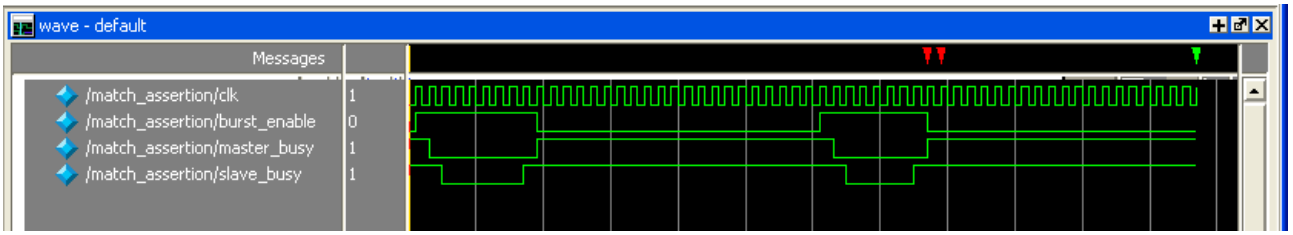
Match operators take two operands or two sequence and produce a new sequence. Following are match sequences operators.

first_match	**When we want to stop after first match of sequence, we use first_match match operator.**
throughout	When we want to check if some condition is valid over period of sequence, then **throughout** match operator is used.
within	When we want to check containment of one sequence in another sequence, we use **within** match operator.

Example : Match Operators

```
//+++++++++++++++++++++++++++++++++++++++++++++++++
//   DUT With assertions
//+++++++++++++++++++++++++++++++++++++++++++++++++
module match_assertion();

logic clk = 0;
always #1 clk ++;

logic burst_enable, master_busy, slave_busy;
//=================================================
// Property Specification Layer
//=================================================
property first_match_prop;
  @ (posedge clk)
    $rose(burst_enable) |=>
      first_match(burst_enable ##[0:4] !master_busy);
endproperty

property throughout_prop;
  @ (posedge clk)
    $rose(burst_enable) |->
      (burst_enable) throughout
          ( ##2 (!slave_busy && !master_busy) [*6]);
endproperty

property within_prop;
  @ (posedge clk)
    $rose(burst_enable) |=>
       (!slave_busy[*6]) within
           (($fell(master_busy)) ##1 !master_busy[*7]);
endproperty
//=================================================
// Assertion Directive Layer
//=================================================
within_assert     : assert property (within_prop);
throughout_assert : assert property (throughout_prop);
first_match_assert: assert property (first_match_prop);
//=================================================
// Generate input vectors
//=================================================
initial begin
  burst_enable <= 0; master_busy <= 1; slave_busy <= 1;
  @ (posedge clk);
  burst_enable <= 1;
  @ (posedge clk);
  master_busy <= 0;
  @ (posedge clk);
  slave_busy <= 0;
  repeat(6) @ (posedge clk);
  slave_busy <= 1;
  @ (posedge clk);
  burst_enable <= 0;
  master_busy <= 1;
  // Fail both the assertion
  repeat(20) @ (posedge clk);
```

```
  burst_enable <= 0; master_busy <= 1; slave_busy <= 1;
  @ (posedge clk);
  burst_enable <= 1;
  @ (posedge clk);
  master_busy <= 0;
  @ (posedge clk);
  slave_busy <= 0;
  repeat(5) @ (posedge clk);
  slave_busy <= 1;
  @ (posedge clk);
  burst_enable <= 0;
  master_busy <= 1;
  // Terminate the sim
  repeat(20) @ (posedge clk);
  $finish;
end

endmodule
```




```
# ** Error: Assertion error.
#    Time: 77 ps Started: 63 ps  Scope: match_assertion.throughout_assert File: E:/Temp/BRAIN/assert2.sv Line: 36
# ** Error: Assertion error.
#    Time: 79 ps Started: 63 ps  Scope: match_assertion.within_assert File: E:/Temp/BRAIN/assert2.sv Line: 35
# ** Note: $finish    : E:/Temp/BRAIN/assert2.sv(70)
#    Time: 117 ps  Iteration: 0  Instance: /match_assertion
```

Clocks in Sequence

As we had discussed ealrier concurrent assertion work on clock edges, so the sequence also. There are two ways clocks can be specified for a sequence to work.

- implied clock : In this clock is specified in property and sequence just uses that clock.
- explicit clock : In this clock is specified inside the sequence block and sequence uses this clock.

Till now we have seen implied clock, below example shows the explicit clock.

```
//++++++++++++++++++++++++++++++++++++++++++++++++
//    DUT With assertions
//++++++++++++++++++++++++++++++++++++++++++++++++
module clock_assertion(
  input wire clk,req,reset,
  output reg gnt);
//================================================
```

```
// Sequence Layer
// Here clock is specified inside the sequence
//================================================
sequence req_gnt_seq;
  @ (posedge clk)
  (~req & gnt) ##1 (~req & ~gnt);
endsequence
//================================================
// Property Specification Layer
// Still requires clock for the property
//================================================
property req_gnt_prop;
  @ (posedge clk)
    disable iff (reset)
      req |=> req_gnt_seq;
endproperty
//================================================
// Assertion Directive Layer
//================================================
req_gnt_assert : assert property (req_gnt_prop)
                 else
                 $display("@%0dns Assertion Failed", $time);
//================================================
// Actual DUT RTL
//================================================
always @ (posedge clk)
  gnt <= req;

endmodule

//++++++++++++++++++++++++++++++++++++++++++++++++
//   Testbench Code
//++++++++++++++++++++++++++++++++++++++++++++++++
module concurrent_assertion_tb();

reg clk = 0;
reg reset, req = 0;
wire gnt;

always #3 clk ++;

initial begin
  reset <= 1;
  #20 reset <= 0;
  // Make the assertion pass
  #100 @ (posedge clk) req  <= 1;
  @ (posedge clk) req <= 0;
  // Make the assertion fail
  #100 @ (posedge clk) req  <= 1;
  repeat (2) @ (posedge clk);
  req <= 0;
  #10 $finish;
end

clock_assertion dut (clk,req,reset,gnt);

endmodule
```

Sequence Arguments

A sequence can be declared with optional arguments, When a sequence is instantiated, actual arguments can be passed to the sequence. The sequence gets expanded with the actual arguments by replacing the formal arguments with the actual arguments.

There are type of arguments declaration.

- No Type : Here input arguments are not typed.
- With Type : Here input arguments are typed.

One of the big advantage of using sequence with argument, is it can be reused as shown in below example.

Example : Sequence Arguments

```
//+++++++++++++++++++++++++++++++++++++++++++++++++
//   DUT With assertions
//+++++++++++++++++++++++++++++++++++++++++++++++++
module propargs_assertion();
logic clk = 0;
logic req,gnt;
logic a,b;
//=================================================
// Sequence Layer with args (NO TYPE)
//=================================================
sequence notype_seq (X, Y);
  (~X & Y) ##1 (~X & ~Y);
endsequence
//=================================================
// Sequence Layer with args (NO TYPE)
//=================================================
sequence withtype_seq (logic X, logic Y);
  (~X & Y) ##1 (~X & ~Y);
endsequence
//=================================================
// Property Specification Layer
//=================================================
property req_gnt_notype_prop(M,N);
  @ (posedge clk)
      req |=> notype_seq(M,N);
endproperty

property a_b_type_prop(logic M, logic N;
  @ (posedge clk)
      a |=> withtype_seq(M,N);
endproperty
//=================================================
// Assertion Directive Layer
//=================================================
req_gnt_notype_assert : assert property (req_gnt_notype_prop(req,gnt));
a_b_type_assert       : assert property (a_b_type_prop(a,b));
//=================================================
// Actual DUT RTL
//=================================================
always @ (posedge clk)
  gnt <= req;

always @ (posedge clk)
  b <= a;
```

```
//+++++++++++++++++++++++++++++++++++++++++++++++++
//  Assertion testing code
//+++++++++++++++++++++++++++++++++++++++++++++++++
always #3 clk ++;

initial begin
  // Make the assertion pass
  #100 @ (posedge clk) req  <= 1;
  @ (posedge clk) req <= 0;
  // Make the assertion fail
  #100 @ (posedge clk) req  <= 1;
  repeat (2) @ (posedge clk);
  req <= 0;

  // Make the assertion pass
  #100 @ (posedge clk) a  <= 1;
  @ (posedge clk) a <= 0;
  // Make the assertion fail
  #100 @ (posedge clk) a  <= 1;
  repeat (2) @ (posedge clk);
  a <= 0;
  #10 $finish;
end

endmodule
```

Local Variables

One can have local variables inside a sequence or a property. This local variables can be assigned values and can be sampled later. Example below shows how this is done.

Example : Local Variables

```
//+++++++++++++++++++++++++++++++++++++++++++++++++
//   DUT With assertions
//+++++++++++++++++++++++++++++++++++++++++++++++++
module localvar_assertion();

logic clk = 0;
always #1 clk ++;

logic [7:0] portin, portout;
logic       in_en, out_en;
//================================================
// Sequence Layer
//================================================
sequence local_var_seq;
   logic [7:0] lport;
   (in_en, lport = portin) ##[1:5]
      (out_en and lport == portout);
endsequence
//================================================
// Property Specification Layer
//================================================
property local_var_prop;
  @ (posedge clk)
      in_en |-> local_var_seq;
endproperty
//================================================
// Assertion Directive Layer
//================================================
```

```
local_var_assert : assert property (local_var_prop);
//================================================
// Simple DUT logic
//================================================
always @ (posedge clk)
begin
  portout <= (portin < 4) ? portin : portin + 1;
  out_en  <= in_en;
end
//================================================
// Generate input vectors
//================================================
initial begin
  in_en <= 0; out_en <= 0;
  portin <= 0; portout <= 0;
  repeat (20) @ (posedge clk);
  for (int i =0 ; i < 8; i ++) begin
    @ (posedge clk);
    in_en <= 1;
    portin <= i;
    @ (posedge clk);
    in_en <= 0;
    portin <= 0;
  end
  #30 $finish;
end

endmodule
```

Calling Subroutine

Tasks, task methods, void functions, void function methods, and system tasks can be called at the end of a successful match of a sequence. The subroutine calls, like local variable assignments, appear in the commaseparated list that follows the sequence. The subroutine calls are said to be attached to the sequence. The sequence and the list that follows are enclosed in parentheses.

All subroutine calls attached to a sequence are executed at every successful match of the sequence.

Example : Calling Subroutine

```
//++++++++++++++++++++++++++++++++++++++++++++++++
//   DUT With assertions
//++++++++++++++++++++++++++++++++++++++++++++++++
module subroutine_assertion();

logic clk = 0;
always #1 clk ++;

logic [7:0] portin, portout;
logic       in_en, out_en;
//================================================
// Sequence Layer
//================================================
sequence local_var_seq;
```

```
   logic [7:0] lport;
   (in_en, lport = portin) ##[1:5]
      (out_en and lport == portout,
        $display ("@%0dns Input port %0d and Output port %0d match",
         $time, lport,portout));
endsequence
//================================================
// Property Specification Layer
//================================================
property local_var_prop;
  @ (posedge clk)
      in_en |-> local_var_seq;
endproperty
//================================================
// Assertion Directive Layer
//================================================
local_var_assert : assert property (local_var_prop);
//================================================
// Simple DUT logic
//================================================
always @ (posedge clk)
begin
  portout <= (portin < 4) ? portin : portin + 1;
  out_en  <= in_en;
end
//================================================
// Generate input vectors
//================================================
initial begin
  in_en <= 0; out_en <= 0;
  portin <= 0; portout <= 0;
  repeat (20) @ (posedge clk);
  for (int i =0 ; i < 8; i ++) begin
    @ (posedge clk);
    in_en <= 1;
    portin <= i;
    @ (posedge clk);
    in_en <= 0;
    portin <= 0;
  end
  #30 $finish;
end

endmodule
```

Multi Clock Support

There are times when we want to check signals or variables from two or more clock domains. As such SystemVerilog provides multi clock support in both **sequence** and **property** declaration.

Multi Clock Sequence

Multiclocked sequences are built by concatenating singly clocked subsequences using the single-delay concatenation operator ##1. This operator is nonoverlapping and synchronizes between the clocks of the two sequences. The single delay indicated by ##1 is understood to be from the end point of the first sequence, which occurs at a tick of the first clock, to the nearest strictly subsequent tick of the second clock, where the second sequence begins.

Below examples shows this is done.

Example : Multi Clock Sequence

```
//++++++++++++++++++++++++++++++++++++++++++++++++++
//   DUT With assertions
//++++++++++++++++++++++++++++++++++++++++++++++++++
module multi_clock_seq_assertion();
logic clk1 = 0;
logic clk2 = 0;
logic req;
logic gnt;
//==================================================
// Sequence Specification Layer
//==================================================
sequence multi_clock_seq;
   @(posedge clk1) req ##1 @(posedge clk2) gnt;
endsequence
//==================================================
// Property Specification Layer
//==================================================
property multi_clock_prop;
  @ (posedge clk1)
      req |-> multi_clock_seq;
endproperty
//==================================================
// Assertion Directive Layer
//==================================================
multi_clock_assert : assert property (multi_clock_prop);
//==================================================
// Here gnt is driven with respect to CLK2
//==================================================
initial begin
  #20 gnt <= 1;
  #120 gnt <= 0;
end
//++++++++++++++++++++++++++++++++++++++++++++++++++
//  Assertion testing code
//++++++++++++++++++++++++++++++++++++++++++++++++++
initial begin
  // Make the assertion pass
  req  <= 0; gnt <= 0;
  #100 @ (posedge clk1) req  <= 1;
  repeat (20) @ (posedge clk1);
  req <= 0;
  #10 $finish;
end

always #1   clk1 ++;
always #6.1 clk2 ++;

endmodule
```

Simulation : Multi Clock Sequence

Multi Clock Property

A multi clock sequence in itself is a kind of multi clock property. Below example is slit variation of multi clock sequence example seen earlier.

Example : Multi Clock Property

```
//+++++++++++++++++++++++++++++++++++++++++++++++++
//   DUT With assertions
//+++++++++++++++++++++++++++++++++++++++++++++++++
module multi_clock_prop_assertion();
logic clk1 = 0;
logic clk2 = 0;
logic req;
logic gnt;
//=================================================
// Property Specification Layer
//=================================================
property multi_clock_prop;
  @(posedge clk1) req |-> ##1 @(posedge clk2) gnt;
endproperty
//=================================================
// Assertion Directive Layer
//=================================================
multi_clock_assert : assert property (multi_clock_prop);
//=================================================
// Here gnt is driven with respect to CLK2
//=================================================
initial begin
  #20 gnt <= 1;
  #120 gnt <= 0;
end
//+++++++++++++++++++++++++++++++++++++++++++++++++
//  Assertion testing code
//+++++++++++++++++++++++++++++++++++++++++++++++++
initial begin
  // Make the assertion pass
  req  <= 0; gnt <= 0;
  #100 @ (posedge clk1) req  <= 1;
  repeat (20) @ (posedge clk1);
  req <= 0;
  #10 $finish;
end

always #1   clk1 ++;
always #6.1 clk2 ++;

endmodule
```

assert, assume and cover

As seen all the example earlier, a property in itself can not be used for checking a condition, it needs to used with verification statements like **assert**.

Followin are verification statements that can use a property.

- assert : This statement specifies if the propery holds correct.
- assume : This statement specifies property as assumption for the verification enviroment. This is more usefull with formal verification tools.
- cover : This statement monitors property for the sake of coverage. Coverage can be made to be reported at the end of simulation.

The assert, assume, or cover statements can be referenced by their optional name. If name is not given then compiler will give name for the sake of reporting as shown in example below.

Above verification statements can be specified in following places.

- initial or always blocks
- module
- interface
- program

Example : assert, assume and cover

```
//+++++++++++++++++++++++++++++++++++++++++++++++++++
//   DUT With assertions
//+++++++++++++++++++++++++++++++++++++++++++++++++++
module assert_assume_cover_assertion();
logic clk = 0;
logic req,gnt;
logic ce,wr;
logic [7:0] addr, data;
//=================================================
// Property Specification Layer
//=================================================
property req_gnt_prop;
  @ (posedge clk)
      req |=> gnt;
endproperty
//=================================================
// Check if address falls in range for a write
// operation
//=================================================
property addr_hit_prop(int min, int max);
  @ (posedge clk)
    (ce & wr) |-> (addr >= min && addr <= max);
endproperty
//=================================================
// Simple DUT with assert
```

```
//================================================
always @ (posedge clk)
begin
   gnt <= req;
   //================================================
   // Assert inside a always block
   //================================================
   req_gnt_assert : assert property (req_gnt_prop);
end
//================================================
// This is how you use "assume"
//================================================
req_gnt_assume  : assume property (req_gnt_prop);
//================================================
// This is how you use "assert"
//================================================
req_gnt_assert2 : assert property (req_gnt_prop);
//================================================
// This is how you use "cover"
//================================================
req_gnt_cover   : cover property (req_gnt_prop);
addr_hit_cover  : cover property (addr_hit_prop(1,5));
//++++++++++++++++++++++++++++++++++++++++++++++++
//  Assertion testing code
//++++++++++++++++++++++++++++++++++++++++++++++++
always #1 clk ++;

initial begin
  ce <= 0; wr <= 0; addr <= 0; req <= 0; gnt <= 0;
  data <= 0;
  // Make the assertion pass
  @ (posedge clk) req  <= 1;
  @ (posedge gnt);
  for (int i = 0; i < 10; i ++) begin
    @ (posedge clk);
    ce <= 1;
    wr <= 1;
    addr <= i;
    data <= $random;
    @ (posedge clk);
    ce <= 0;
    wr <= 0;
    addr <= 0;
  end
  @ (posedge clk);
  req <= 0;
  // Check
  #10 $finish;
end

endmodule
```

...

| sequence_expr |-> property_expr

| sequence_expr |=> property_expr

Therefore,

`sequence_expr |=> property_expr`
is equivalent to the following:

`sequence_expr ##1 'true |-> property_expr`

This clause is used to precondition monitoring of a property expression and is allowed at the property level. The result of the implication is either true or false. The left-hand operand *sequence_expr* is called the *antecedent*, while the right-hand operand *property_expr* is called the *consequent*.
The following points should be noted for `|->` implication:
— From a given start point, the antecedent *sequence_expr* can have zero, one, or more than one successful match.
— If there is no match of the antecedent *sequence_expr* from a given start point, then evaluation of the implication from that start point succeeds vacuously and returns true.
— For each successful match of antecedent *sequence_expr*, the consequent *property_expr* is separately evaluated. The end point of the match of the antecedent *sequence_expr* is the start point of the evaluation of the consequent *property_expr*.
— From a given start point, evaluation of the implication succeeds and returns true if, and only if, for every match of the antecedent *sequence_expr* beginning at the start point, the evaluation of the consequent *property_expr* beginning at the end point of the match succeeds and returns true.

Поддерживаются две формы имликации: перекрывающаяся использует операторо `|->` и неперекрывающаяся `|=>`. Для перекрывающейся импликации, если присутствуею предшествующая последовательность *sequence_expr*, то ее точка завершения будет началом для вычисления следующей последовательности *property_expr*. Для неперкрывающейся импликации, вычисление следующей последотельности *property_expr* начнеться со следующего такта, после завершения предыдущей.

Следующий пример иллюстрирует операцию передачу данных по шине от мастера к целевому устройству. Когда шина переходит в состояние передачи данных `data_phase`, то за этим может потребоваться несколько фаз передачи данных по блокам. При этом данные передаются по переднему фронту синхросигнала, когда устанавливается irdy и один из сигналов `trdy` или `stop`. В данном примере управление ведется низким уровнем и сигнал считается установленным, когда принимает значение нуля.

```
property data_end;
   @(posedge mclk)
      data_phase |-> ((irdy==0) && ($fell(trdy) || $fell(stop))) ;
endproperty
```

Each time a data phase is true, a match for `data_phase` is recognized. The attempt at clock tick 6 is illustrated in Figure 17-13. The values shown for the signals are the sampled values with respect to the clock. At clock tick 6, `data_end` is true because `stop` gets asserted while `irdy` is asserted.

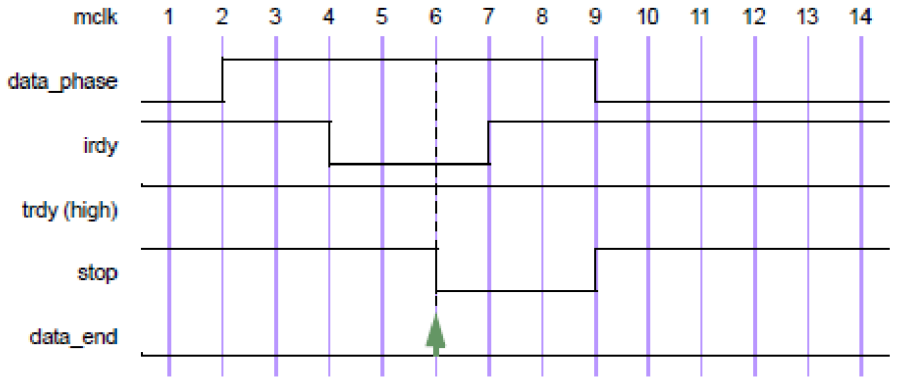


Figure 17-13—Conditional sequence matching

Рис 17.3 Условное совпадение последовательности

В следующей примере, `data_end_exp` используется для того, чтобы гарантировать, что сигнал `frame` не установлен (значение 1) в течении двух синхротактов после появления `data_end_exp`. Затее необходимо, чтобы на следующем такте сигнал `irdy` также будет сброшен(установлен в 1).

```
'define data_end_exp (data_phase && ((irdy==0)&&($fell(trdy)||$fell(stop))))
property data_end_rule1;
```

```
        @(posedge mclk)
            'data_end_exp |-> ##[1:2] $rose(frame) ##1 $rose(irdy);
    endproperty
```

Свойство `data_end_rule1` сначала вычисляет `data_end_exp` на каждом синхротакте для тестирования, что его значение «истина». Если его значение «ложь», значит все значение `data_end_rule1` имеет значение истина. Иначе, вычисляется следующая часть выражения `##[1:2] $rose(frame) ##1 $rose(irdy)`, что выражается в поиске в течение следующих двух сихротактов переключения сигнала `frame` в единицу. После этого на следующем синхротакте `irdy` также должна перейти в 1. Это иллюстрируется рисунком 17.14. На синхротакте 6 обнаруживается последовательность `'data_end_exp`, сигнал `frame` переключается в 1 на синхротакте 7. Поскольку это случается в период 1-2 тактов после `data_end_exp`, то это удовлетворяет условию и последовательность вычисляется дальше. На синхротакте 8 вычисляется `irdy`, последовательность имеет место, для попытки, которая началась на такте 6.

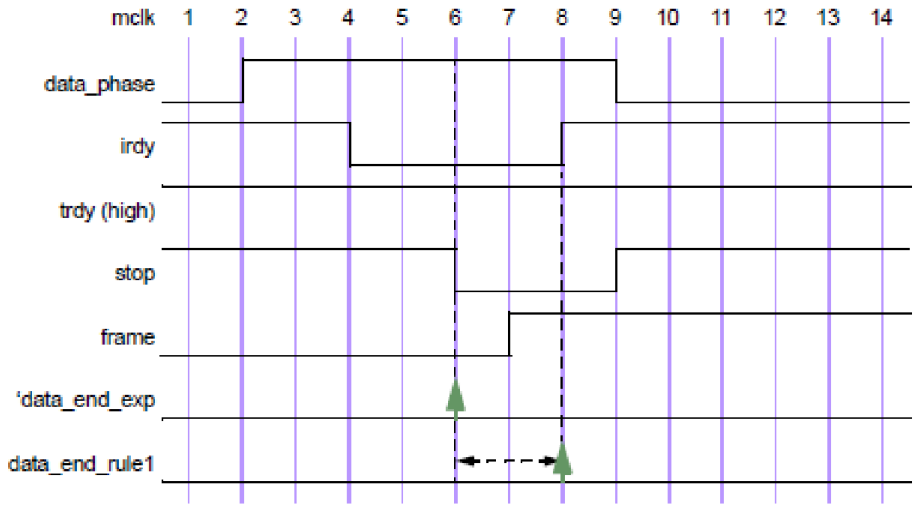


Figure 17-14—Conditional sequences

Рис. 17.14. Условные последовательности.

Обычно, ассерции ассоциируются с предварительным условием, таким образом, их проверка выполняется только после некоторого определенного условия. Как видно, в предыдущем примере оператор `|->` позволяет задавать такое предварительное условие в виде последовательностей, которые должны быть вычислены до вычисления всего свойства. Следующая модификация пример иллюстрирует, что произойдет, если убрать предварительное вычисление условий. Результат иллюстрируется примером 17.15.

```
property data_end_rule2;
@(posedge mclk) ##[1:2] $rose(frame) ##1 $rose(irdy);
endproperty
```

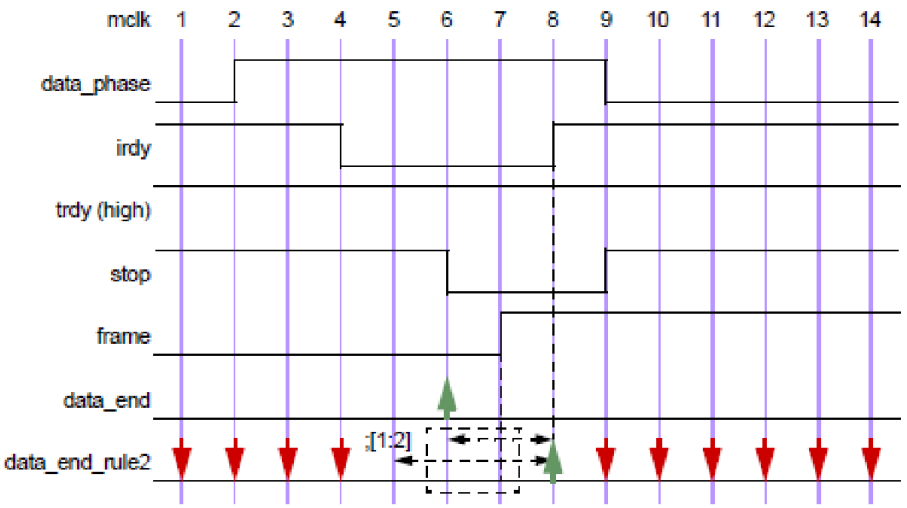


Figure 17-15—Results without the condition

Рис 17.15. Результат без условия

The property is evaluated at every clock tick. For the evaluation at clock tick 1, the rising edge of signal `frame` does not occur at clock tick 2 or 3; therefore, the property fails at clock tick 1. Similarly, there is a failure at clock ticks 2, 3, and 4. For attempts starting at clock ticks 5 and 6, the rising edge of signal `frame` at clock tick 7 allows checking further. At clock tick 8, the sequences complete according to the specification, resulting in a match for attempts starting at clock ticks 5 and 6. All later attempts to match the sequence fail because `$rose(frame)` does not occur again.
Figure 17-15 shows that removing the precondition of checking `'data_end_exp` from the assertion causes failures that are not relevant to the verification objective. It is important from the validation standpoint to determine these preconditions and use them to filter out inappropriate or extraneous situations.
An example of implication where the antecedent is a sequence follows:

```
(a ##1 b ##1 c) |-> (d ##1 e)
```

If the sequence `(a ##1 b ##1 c)` matches, then the sequence `(d ##1 e)` must also match. On the other hand, if the sequence `(a ##1 b ##1 c)` does not match, then the result is true.
Another example of implication is as follows:

```
property p16;
(write_en & data_valid) ##0
(write_en && (retire_address[0:4]==addr)) [*2] |->
##[3:8] write_en && !data_valid &&(write_address[0:4]==addr);
endproperty
```

This property can be coded alternatively as a nested implication:

```
property p16_nested;
(write_en & data_valid) |->
(write_en && (retire_address[0:4]==addr)) [*2] |->
##[3:8] write_en && !data_valid && (write_address[0:4]==addr);
endproperty
```

Multiclock sequence implication is explained in 17.12.

17.11.3 Property examples

Следующие примеры иллюсьрируют различные формы свойств:

```
property rule1;
   @(posedge clk) a |-> b ##1 c ##1 d;
endproperty

property rule2;
   @(clkev) disable iff (foo) a |-> not(b ##1 c ##1 d);
endproperty
```

Свойство `rule2` отрицает последовательность `(b ##1 c ##1 d)` в импликации. Синхросигнла `clkev` управлеяет порядком приема значений в свойстве.

```
property rule3;
   @(posedge clk) a[*2] |-> ((##[1:3] c) or (d |=> e));
endproperty
```

Свойство `rule3`: если a имеет место на двух тактах clk, то со второго такт‌t должна появиться последовательность c и случиться 3 раза, или появиться d, за которой на следующем такте e.

```
property rule4;
   @(posedge clk) a[*2] |-> ((##[1:3] c) and (d |=> e));
endproperty
```

Свойство `rule4`: says that if `a` holds and `a` also held last cycle, then `c` must hold at some point one to three cycles after the current cycle and, if `d` holds in the current cycle, then `e` must hold one cycle later.

```
property rule5;
@(posedge clk)
   a ##1 (b || c)[->1] |->
   if (b)
      (##1 d |-> e)
   else // c
      f ;
endproperty
```

Property `rule5` has `a` followed by the next occurrence of either `b` or `c` as its antecedent. The consequent uses **`if`**...**`else`** to split cases on which of `b` or `c` is matched first.

```
property rule6(x,y);
##1 x |-> y;
endproperty
property rule5a;
@(posedge clk)
a ##1 (b || c)[->1] |->
if (b)
```

```
rule6(d,e)
else // c
f ;
endproperty
```

Property `rule5a` is equivalent to `rule5`, but it uses an instance of `rule6` as a property expression.
A property can optionally specify an event control for the clock. The clock derivation and resolution rules are described in 17.14.
A named property can be instantiated by referencing its name. A hierarchical name can be used, consistent with the SystemVerilog naming conventions. Like sequence declarations, variables used within a property that are not formal arguments to the property are resolved hierarchically from the scope in which the property is declared.

Properties that use more than one clock are described in 17.12.

10.7 Параллельные и прамые ассерции

На практике, большинство прямых (sequential) ассерций выражаются в некоторой форме импликации, и таким образом требуют записи ассерций для описания предыдущего выражения для переключения ассерции. Как указывалось ранее, одним из ключевых преимуществ комплексных с языком проектирования асерций – простота встраивания ассерций в проект для инженеров-проектировщиков. Тем не менее, декларация ассерций, часто требует дополнительных усилий от проектировщика для эффективного их использования.

Рассмотрим модель управляющего автомата. Ассерции в этом типе проектов часто формируют «если это состояние, то должно произойти» и «если в состоянии ACK, если некоторая переменная foo равна 1, тогда регистр должен сохранять значение 0 в течении 5 синхротактов». Параллельная ассерция для проверки этого условия может быть записана следующим образом.

```
P1: assert property(
@(posedge clk) (st == ACK) && (foo == 1) |-> !req[*5]);
При этом RTL-код автомата должен выглядеть подобным образом:
always @(posedge clk)
case (st)
  ACK:
  if (foo == 1)
       begin  // в этом состоянии req=0 в течении 5 синхротактов
            ...
       end
  ...
endcase
```

Получается два фрагмента информации, которые дублируются в проекте и ассерции: синхронизация и переключение состояний. Вместо того, чтобы требовать от проектировщика дублирования этой информации, ассерции, являющиеся частью языка проектирования, могут быть процедурно встроены в RTL-код и автоматически выводить информацию при их использовании, например:

```
always @(posedge clk)
case (st)
  ACK:
  if (foo == 1)
   begin
        P2: assert property (!req[*5]);
        ...
   end
```

Эта прямая ассерция (P2) является семантически эквивалентной декларации параллельной ассерции (P1) приведенной выше, но гораздо проще в использовании и обслуживании. Процедурные встроенные ассерции используют полученные значения сигналов для оценки условий переключения, как делают декларативные ассерции. Правила логического вывода для предшественников включают в себя case и if-else операторы, что позволяет вводить в процедурные ассерции переключения произвольной сложности. В результате преимущество для процедурных ассерций является простым обслуживанием. То есть, если изменяется кода конечного автомата, условия переключения для ассерции автоматически обновляется, а пользователю придется модифицировать код вручную, чтобы обновить переключения, соответствующие декларативным ассерциям

10.7. Взаимодействие ассерций и Testbench

Как указывалось ранее, значительным преимуществом асерций комплексных с проектом и языком верификации – это возможность для ассерций передавать информацию в TestBench. В отдельном языке ассерций не существовало такого механизма коммуникаций. В SystemVerilog в ветвях pass и fail ассерции могут выполнять любые процедурные операторы, позволяя пользователю передавать в TestBench, что последовательность

выполнилась, обновлять счетчики покрытия или много другое, что может содействовать эффективности процедуры верификации, включая установку значений сигналов, вызов методов объектов класса, или даже вызов кода Си.

Пусть шины включает свойство «новый цикл шины не может начаться раньше, чем через 2 синхротакта после появления сигнала завершения цикла abort_cycle». Это свойство может кодироваться как

```
property wait_after_abort;

    @(posedge clk) abort_cycle |=> !cycle_start[*2];

endproperty

P3: assert property (wait_after_abort);
```

Когда появляется последовательность abort_cycle, свойство диктует, что сигнал cycle_start будет равен нулю в течение двух синхротактов. Свойство P3 кодирует это поведение и может быть использвано для мониторинга во время моделирования для проверки поведения, или в формальных инструментах, чтобы попытаться доказать, что поведение никогда не случиться.

Тем не менее, если это свойство определяет поведение интерфейса верхнего уровня, testbench будет генерировать транзакцию шины, которая должна подтвердить это поведение. Инженеры верификации, создавая testbench могут повторно использвоать информацию ассерций, как константу, которая помогает гарантировать, что генерируемые символы никогда не нарушат поведение. Такой testbench может выглядет подобным образом

```
program manual_stimulus_generator;
repeat(1000)
begin
   generate_transaction(addr,data);
   while(wait_cnt > 0)
       @(posedge clk) wait_cnt--;
 end
endprogram
```

Счетчик wait_cnt используется для описания задержки между транзакциями шины генерируемые the testbench. Последовательность abort_cycle может быть использована для установки этого счетчика, используя директиву покрытия.

```
cover property( abort_cycle ) wait_cnt = 2;
```

Использование директивы cover позволяет обнаруживать последовательность abort_cycle без формирования ошибки, если последовательность не присутствует. Если последовательность обнаружена, то счетчик wait_cnt устанавливается равным 2, заставляет testbench ждать 2 такта до генерирования следующей транзакции. Семантика формирования порядка обработки событий SystemVerilog гарантирует, что здесь не будет гонок между выполнением pass statement и testbench, что нельзя было бы гарантировать, если бы ассерции были реализованы с помощью другого языка.

SystemVerilog включает возможность описания явного синхронного интерфейса между TestBench и проектом. Конструктор области синхронизации определяет, когда относительно синхронизации testbench будет захватывать и передавать тестируемому устройству (DUT) сигналы. По умолчанию, захват значений сигналов в testbench - это тоже самое, что захват значений с использованием ассерций, гарантирует, что testbench имеет доступ к тем же значениям, что и ассерции. Это не только позволяет избежать гонок между ассерциями testbench, ассерциями и тестируемым устройством, но также дает testbench представление о “post-synthesis» поведении устройства, с разрешением гонок, согласно правилам синтеза, во время RTL-верификации, выполняемой до синтеза.

10.8. Поддержка многодоменной синхронизации

Последовательности и свойства могут использовать многодоменную синхронизацию .

10.8.1. Последовательности с множественной синхронизацией (Multiply-clocked sequences)

Последовательности с множественной синхронизацией строятся путем конкатенации нескольких подпоследовательностей с помощью оператора конкатенации с единичной задержкой ##1. Этот оператор не перекрывает и не синхронизирует соединение синхросигналов двух последовательностей. Единичная задержка, обозначаемая ##1, понимается как задержка между окончанием первой последовательности, которая управляется первым синхросигналом, и ближайшим активным событием (или фронтом) второго синхросигнала, с которого начинается выполнение второй синхропоследовательности. Например,

```
@(posedge clk0) sig0 ##1 @(posedge clk1) sig1
```

Поиск последовательности начинается с сравнения sig0 после появления переднего фронта clk0. Затем ##1 выполняет задержку до ближайшего переднего фронта sig1. Поиск последовательностей завершается с окончанием последовательности sig1. Если clk0 и clk1 не идентичны, то синхрособытие переключается после ##1. Иначе, если clk0 и clk1 идентичны, синхронизирующее событие не изменяется после ##1 и представленная последовательность эквивалентна последовательности с одним синхросигналом:

```
@(posedge clk0) sig0 ##1 sig1
```

Когда выполняется конкатенация по-разному синхронизированных последовательностей, максимальная подпоследовательность с единичной синхронизацией необходима, чтобы допустить непустое совпадение. Поэтому, если последовательности s1 и s2 являются выражениями последовательностей без событий синронизаций, то тогда последовательность с множественной синхронизацией

```
@(posedge clk1) s1 ##1 @(posedge clk2) s2
```

допустима, если s1 или s2 совпадают с пустым словом. Событие переднего фронта синхросигнала clk1 применяется на всем протяжении поиска последовательности s1, в то время как передний фронт clk2 является событием синхронизации во время поиска последовательности s2. Если совпадение s1 – не пусто, то это событие является конечной точкой совпадения переднего фронта clk1. Синхронизация между конечной точкой последовательности и появлением переднего фронта clk2 после этого достигается с помощью оператора ##1. Передний фронт clk2 является начальной точкой поиска последовательности s2.

10.8.2. Свойства с мультисинхронизацией (Multiply-clocked properties)

Для свойств с мультисинхронизацей результат вычисляется также, как и для случая с единственным синхросигналом. Свойства с множественной синхронизацией могут быть описаны в несколько способов. Последовательность с мультисинхронизацией является сама по себе свойством с мультисинхронизацией. Например,

```
@(posedge clk0) sig0 ##1 @(posedge clk1) sig1
```

свойство с множественной синхронизацией. Если последовательность с множественной синхронизацией вычисляется как свойство, начинающееся в некоторой точке, вычисление возвращает значение истины, если и только если, существует совпадение в последовательности с множественной синхронизацией в этой точке.

Булевы операторы свойств (not, and, or) могут быть использованы свободно для комбинировании одно- и мульти-синхронные свойства. Значения операторов булевых свойств всегда единицы, также как случай отдельносинхронизируемых свойств. Например,

```
(@(posedge clk0) sig0) and (@(posedge clk1) sig1)
```

мультисинхронное свойство, но это не мультисинхронное последовательность. Это свойство возвращает true в точке, если и только, если две последовательности

```
@(posedge clk0) sig0
```

и

```
@(posedge clk1) sig1
```

обе совпадаются в начале этой точки.

Два неперекрывающихся оператора импликации |=> могут быть свободно использованы для создания мультисинхронного свойства из предшествующей последовательности и последующие свойства, что по-другому или мультисинхронизированы. Значение мультисихронизированной неперекрывающейся импликации подобна этой одно-синхронизируемой неперекрывающейся импликации. Например, если s0, s1 – последовательности с несинхронизируемым событием, тогда в

```
@(posedge clk0) s0 |=> @(posedge clk1) s1
```

|=> синхронизируются передним фронтом clk0 и передним фронтом clk1. Начиная с точки в которой импликация была вычислена, каждое совпадение s0 синхронизируется clk0, время движется вперед с конечной точки совпадения к ближайшей строго в будущем произошедшей по переднему фронту clk1, и из этой точки здесь должно выполняться совпадение s1 синхронизированного по clk1 между передним фронтом clk0 и передним фронтом clk1. Начиная с точки в которой вычисляется импликация, для каждого совпадения s0 синхронизированного clk0, время движется от конечной точки совпадения к ближащей строго случащейся в по переднему фронту clk1, и из точки здесь должно существовать совпадение с s1 синхронизированной clk1. Поскольку синхронизация между отдельными синхросигналами всегда требует строгое предшествование во времени, два оператора строящих свойства, что требуют специальное свойство с множественной синхронизацией для перекрывающейся импликацией |-> и если if/ if...else. Позже |-> перекрытие конца

предшественника с началом этой последующей, синхронизация для конца предшественника должна быть такая же как для начала следующей. Например, если clk0 и clk1 не идентичны и s0, s1, s2 – это последовательности с несинхронизированными событиями, тогда

```
@(posedge clk0) s0 |-> @(posedge clk1) s1 ##1 @(posedge clk2) s2
```

не правильное выражение, но

```
@(posedge clk0) s0 |-> @(posedge clk0) s1 ##1 @(posedge clk2) s2
```

правильное.

10.8.3. Синхропоток (Clock flow)

Хотя подсекция c, d обозначает событие синхронизации и v, w, x, y, z обозначают последовательности с несинхронизированными событиями. Синхропоток позволяет масштабировать события синхронизации для того, чтобы распространить его обычным способом через различные части мультисинхронных последовательностей и сокращают число мест в которых одинаковые события синхронизации должны быть описаны. Интуитивно, поток синхронизации обеспечивает, что в мультисинхронной последовательности или свойства масштабирую события синхронизации идущего слева направо через линейные операторы(т.е повторение, конкатенация, отрицание, импликация) и распределения операндова разветвленных операторов(например, коньюкция, дизъюнкция, пересечение, if … else), пока не будет заменено новым событием синхронизации.

Например,

```
@(c) x |=> @(c) y ##1 @(d) z
```

может быть записано в более простой форме

```
@(c) x |=> y ##1 @(d) z
```

потому что синхросигнал с понимается проходящим через | =>.

Синхропоток избавляет от необходимости записывать синхро события в позициях, где не допускается изменение синхронизации. Например,

```
@(c) x |-> @(c) y ##1 @(d) z
```

может быть записано как

```
@(c) x |-> y ##1 @(d) z
```

усилить ограничения, что синхронизация не изменится при переходе через | ->. Аналогично,

```
@(c) if (b) @(c) w ##1 @(d) x else @(c) y ##1 @(d) z
```

может быть записано как

```
@(c) if (b) w ##1 @(d) x else y ##1 @(d) z
```

усилить ограничения, что синхронизация не изменилась с булевыми условием b с начала, if и else ветви свойства. Синхропоток также делает соединенные отношения между конкатенацией и импликацией ясными для мультисинхронными свойствами:

```
@(c) x ##1 y |=> @(d) z
```

эквивалентно

```
@(c) x |=> y |=> @(d) z
```

и

```
@(c) x ##0 y |=> @(d) z
```

эквивалентно

```
@(c) x |-> y |=> @(d) z
```

Границы событий синхронизации соответствуют подвыражениям ограниченным скобками и, если подвыражение является последовательностью, также передается слева направо, прямо через скобки подвыражении. Тем не менее, границы видимости собитий синхронизации ограничиваются скобками.

10.8.4. Примеры

Примеры мультисинхронного описания (multiple-clock specifications):

```
sequence s1;
        @(posedge clk1) a ##1 b;          // последовательность с единственным синхросигналом
endsequence
```

```
sequence s2;
        @(posedge clk2) c ##1 d;          // последовательность с единственным синхросигналом
endsequence
```

1) Мультисинхрованная последовательность

```
sequence mult_s;
    @(posedge clk) a ##1 @(posedge clk1) s1 ##1 @(posedge clk2) s2;
endsequence
```

2) Свойство с мультисинхронной последовательностью

```
property mult_p1;
    @(posedge clk) a ##1 @(posedge clk1) s1 ##1 @(posedge clk2) s2;
endproperty
```

3) Свойство с именной мультисинхронной последовательностью

```
property mult_p2;
  mult_s;
endproperty
```

4) Свойства с импликацией мультисинхронизации

```
property mult_p3;
   @(posedge clk) a ##1 @(posedge clk1) s1 |=> @(posedge clk2) s2;
endproperty
```

5) Свойство с именными последовательностями, имеющими различную синхронизацию. В этом случае, если s1 содержит сихросигнал, то он должен быть идентичный с (posedge clk1). Подобным образом, если s2 содержит синхросигнал, он должен быть идентичный (posedge clk2).

```
property mult_p5
          @(posedge clk1) s1 |=> @(posedge clk2) s2;
endproperty
```

6) Свойства с импликацией, где предшественник и последователь являются именными мультисинхронными последовательсностями

```
property mult_p6;
       mult_s |=> mult_s;
endproperty
```

7) Свойство, использующее синхропоток и перекрывающуюся импликацию:

```
property mult_p7;
       @(posedge clk) a ##1 b |-> c ##1 @(posedge clk1) d;
endproperty
```

Здесь a, b и c синхронизируются передним фронтом clk.

8) Свойство, использующее синхропоток и if...else:

```
property mult_p8;
         @(posedge clk) a ##1 b |->
if (c)
        (1 |=> @(posedge clk1) d)
else
        e ##1 @(posedge clk2) f ;
endproperty
```

Здесь a, b, c и e синхронизируются передним фронтом clk.

10.5. Контрольные вопросы и задания

1. Создать последовательность, проверяющую одновременное выполнение последовательностей seq1 и seq2, управляемых передним фронтом синхросигнала clk.

2. Создать последовательность, проверяющую выполнение хотя бы одной из двух последовательностей seq1 и seq2, управляемых передним фронтом синхросигнала clk.

3. Написать параллельную ассерцию, выполняющую проверку значений A=01, set=1, по переднему фронту clk; выводящую сообщения в случае удачного и неудачного выполнения ассерции с указанием метки ассерции.

4. Написать прямую именную ассерцию, выполняющую проверку значений A=01, set=1, выводящую сообщения в случае удачного и неудачного выполнения ассерции с указанием метки ассерции.

5. Написать параллельную ассерцию, выполняющую проверку значений, что сигнал grant представлен унарным кодом, set=1, если reset не равен 1; выводящую сообщение об ошибке в случае неудачного выполнения ассерции.

6. Написать параллельную ассерцию, выполняющую проверку следования сигнала a=1, через 5 тактов после d=1 или b=1. Управляется задним фронтов clk. В случае, неуспешного выполнения ассерции выводить предупреждение.

7. Написать параллельную ассерцию, выполняющую проверку следования сигнала a=1, через 2-5 тактов после c=1 и b=1. Управляется задним фронтов clk. В случае, неуспешного выполнения ассерции выводить предупреждение.

8. Написать параллельную ассерцию, выполняющую проверку следования сигнала a=1 или b=1, через 3 тактов после c=1. Управляется передним фронтов clk. В случае, неуспешного выполнения ассерции выводить предупреждение.

9. Написать параллельную ассерцию, выполняющую проверку следования сигнала a=1, через 5 тактов после d=1 и через 8 тактов после b=1. Управляется передним фронтов clk. В случае, неуспешного выполнения ассерции выводить предупреждение.

10. Написать параллельную ассерцию, выполняющую проверку следования сигнала a=1 или d=1 и через такт после b=1. Управляется передним фронтов clk. В случае, неуспешного выполнения ассерции выводить сообщение о фатальной ошибке.

11. Создать последовательность, без перекрытия соединяющую последовательности a и b, обозначить последовательности с помощью формальных параметров, которые позволят их переопределить при использовании последовательности.

12. Создать последовательность, проверяющую последовательное через 2 такта друг за другом выполение условий a, b и c, управляемых передним фронтом синхросигнала clk.

13. Написать параллельную ассерцию, выполняющую проверку появления set=1 и через такт A=01, управляется передним фронту clk. Выводятся сообщения с указанием метки ассерции в случае удачного и неудачного выполнения ассерции.

14. Написать прямую ассерцию, выполняющую проверку значений grant=01 и request=1, выводящую сообщения в случае удачного и неудачного выполнения ассерции с указанием метки ассерции.

15. Написать параллельную ассерцию, выполняющую проверку значений, что сигнал grant содержит как минимум одну 1, если сигнал reset не равен 1; выводящую сообщение об ошибке в случае неудачного выполнения ассерции.

16. Написать параллельную ассерцию, выполняющую проверку следования сигнала a=1, через 2-5 тактов после c=1 и b=1. Управляется задним фронтов clk. В случае, неуспешного выполнения ассерции выводить предупреждение.

17. Написать параллельную ассерцию, выполняющую проверку следования сигнала a=1 или b=1, через 3 тактов после c=1. Управляется передним фронтов clk. В случае, неуспешного выполнения ассерции выводить предупреждение.

Выборка – значение переменной, полученное в момент появления синхрособытия.