Лекция 1

Что такое ВЕРИФИКАЦИЯ

Представьте, что вам поручили строить дом для кого-то. С чего начать? Вы начинаете, выбирая двери и окна, выбирая краски и ковер цветов, или выбор сантехники? Конечно, нет!

Во=Первых

Вы должны рассмотреть, как владельцы будут использовать пространство, их бюджет, так что вы должны решить, какой тип дома построить. Перед тем как начать, изучать подробности SystemVerilog вы должны понять, как вы планируете верифицировать устройство и, как это влияет на структуры testbench.

Так же, как во всех домах есть кухня, спальни, и ванные комнаты, все тесты имеют некоторые общие структуры стимулов генерации и проверки реагирования

Важнейший принцип. который вы должны усвоить, как инженер -тестировщик: "Ошибки хороши." Не уклоняться от поиска следующей ошибки, и не звонить в колокола каждый раз, когда вы обнаружили новую, но всегда отслеживать каждую найденную ошибку. Вы должны внимательно насколько это возможно, осмотреть проект с разных сторон, чтобы извлечь все возможные ошибки сейчас, пока они еще легко исправимы .

Характерные черты SystemVerilog как Языка Проверка оборудования (HVL) которые отличают его от Языков Описание оборудования, таких как Verilog или VHDL являются

-случайные воздействия с ограничениями
- Функциональные покрытия
-СТРУКТУРЫ высшего уровня , особенно объектно-ориентированного программирование. динамические структуры данных
многопоточность и взаимодействия процессов
Поддержка HDL типов, таких как 4-значные состояния в Verilog
-- Тесная интеграция с событие-симулятор
Есть множество других полезных функций, но это позволит вам создать тест на более высоком уровне абстракции, чем вы сможете достичь с помощью HDL или языка программирования, таких как C.

www/accelera.org /System verilog 3.1 Language reference manual.

Процесс верификации

Какова цель верификации? ответ: «Поиск ошибок", будет лишь отчасти верным.

Цель аппаратного обеспечения является создание устройства, которое выполняет конкретную задачу, такую как DVD-плеер, сетевой маршрутизатор или процессор радиолокационного сигнала, на основе проектной спецификации. цель инженера-

тестировщика, состоит в том чтобы убедиться, что устройство может выполнить эту задачу успешно - то есть, проект есть точное представление спецификации. Ошибка это то, что вы получаете, при обнаружении расхождения. **Процесс верификации параллелен процессу создания устройства**.

дизайнер читает аппаратные спецификации блока, интерпретирует человеческий язык описания, и создает **соответствующее логическое описание** в машиночитаемой форме, как правило, RTL кода. Чтобы сделать это, он должен понимать формат ввода, функцию преобразования и формат вывода.

Всегда возможна двусмысленность в этой интерпретации, в том числе из-за неясностей в оригинальном документе, недостающих деталей , или противоречивого описания.

инженер-тестировщик, также должен прочитать аппаратные спецификации, создать план верификации, а затем выполнить его, чтобы провести испытания, показывающие что RTL код **правильно реализует функции**. Назначив более чем одного человека выполнять ту же интерпретацию, вы добавили избыточность процесса проектирования.

Работа инженера-тестировщика также состоит в чтении аппаратных спецификаций, чтобы сделать независимую оценку того, что они означают. Ваши тесты, а затем эксперименты RTL должны показать, что это соответствует вашей интерпретации.

Какие типы ошибок скрываются в проекте? Самый простой для обнаружения из них находятся на уровне блоков, модулей, созданных одним человеком.

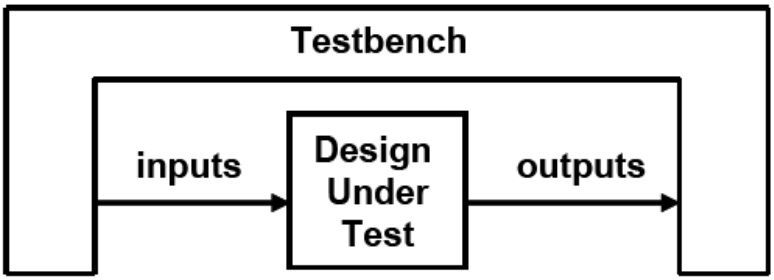
Правильно ли ALU складывают два числа? Разве каждый успешно ли шина завершает транзакцию ? Все ли пакеты прошли через часть сетевого коммутатора?

Почти тривиально писать направленные(=прямые) тесты, чтобы найти эти ошибки, если они содержатся полностью в пределах одного блока конструкции.

После уровня блоков, следующее место для поиска расхождений - границы между блоками. Интересные проблемы возникают, когда два или больше дизайнеров читая ТО же описание имеют различные толкования. Для данного протокола, какие сигналы изменяются и когда? Первый дизайнер строит драйвер шины и имеет один взгляд на спецификации, а второй строит приемник с немного иной точки зрения. Ваша задача состоит в нахождении в спорных районах общей логики и, возможно, даже помочь примирить эти две различные точки зрения.

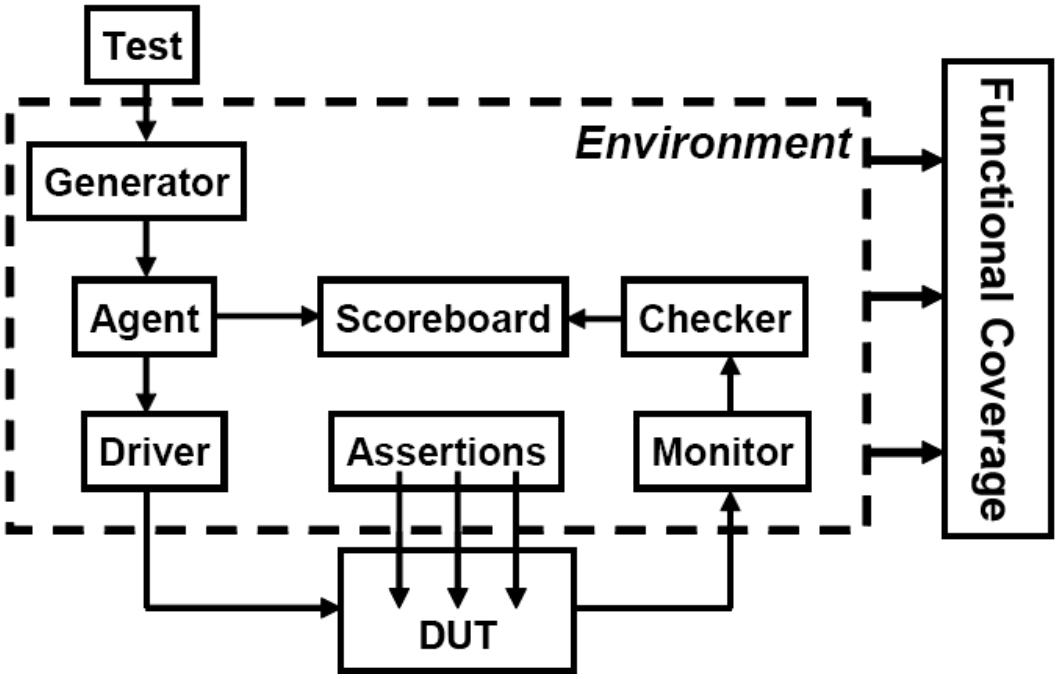
Для имитации одного блока, необходимо создать тесты, которые генерируют стимулы из всех блоков, что довольно муторно. Преимуществом является то, эти низкоуровневые модели исполняются очень быстро.

The testbench — design environment



К тому же вы можете обнаружить ошибки как в DUTтак и в testbench ПРИЧЕМ часто требуется БОЛЕЕ длинный код чтобы обеспечить стимуляции от отсутствующих блоков. Когда вы начинаете интегрировать блоки системы , они могут стимулировать друг друга, сокращая рабочую нагрузку. многоблочные модели позволяют раскрыть больше ошибок, но они и работают медленнее. На самом высоком уровне тестируемого устройства вся система тестируется, но производительность симуляции значительно снижается.

Ключевое понятие для любой современной методологии верификации это слоистые testbench. Хотя может показаться, что testbench в этой постановке более сложен, на самом деле это помогает реализовать Вашу задачу путем деления кода на более мелкие части, которые могут быть разработаны отдельно. Не пытайтесь писать одну процедуру, которая может случайным образом генерировать все виды стимулов, в том числе нелегальных, а также вводить ошибки многослойного протокола. Программа быстро становится сложной и не поддерживаемой.



Блоки testbench (внутри пунктирной линии) записываются в начале разработки. В ходе проекта они могут развиваться и вы можете добавить функциональности, но эти блоки не должны меняться для отдельных тестов.

Ваши тесты должны стремиться, чтобы все Блоки выполняли интересующие действия одновременно. Например Все порты ввода / вывода являются активными, Процессоры перемалывает данные а кэши пополняются. В совокупности таких действий, несомненно, происходят ошибки выравнивания данных и временные несогласования,.

После того как вы убедились, что DUT правильно, выполняет назначенные функции нужно посмотреть, как оно работает, когда есть ошибки. Может ли DUT поддерживать частичные транзакции, или транзакции с поврежденными данными или контрольным полем? Трудно просто перечислить все возможные проблемы, так же трудно определить как дизайнер должен справиться с ними .

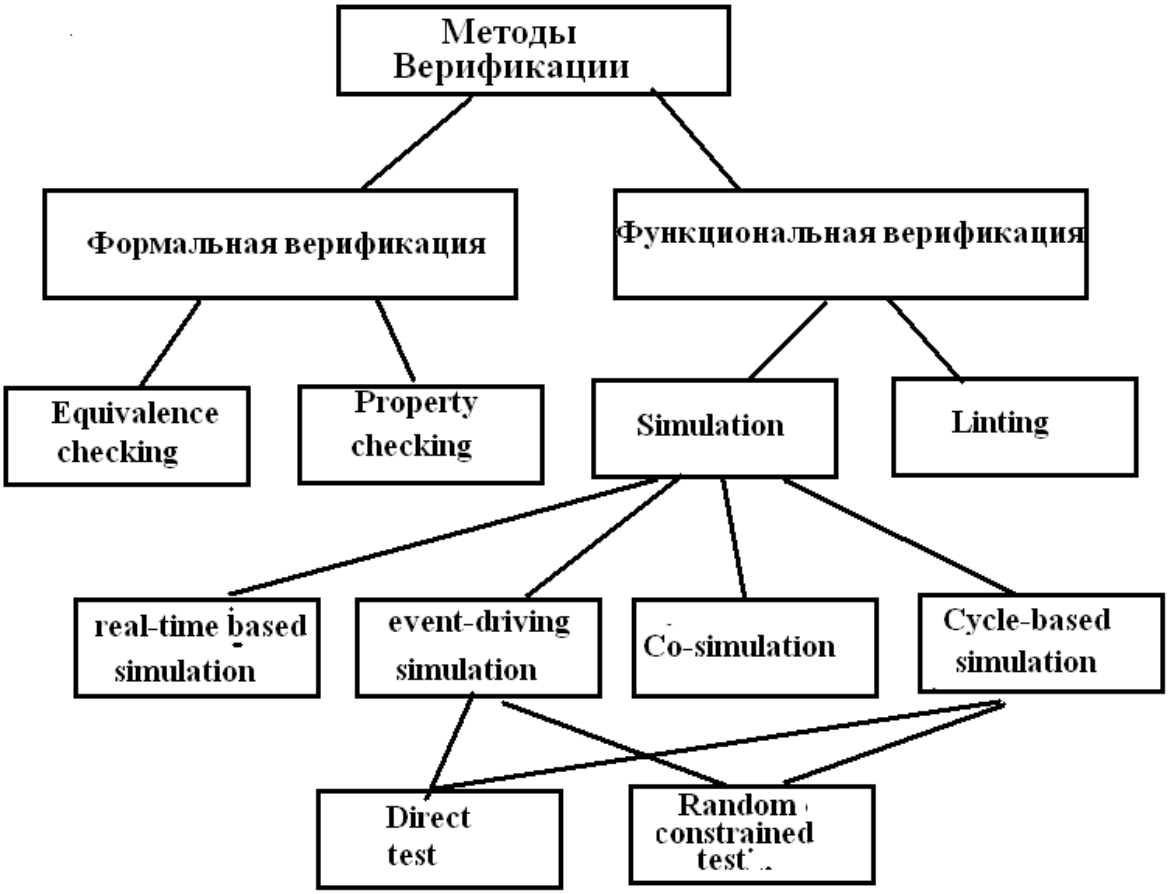
Ошибки вмешательств и обработки может быть самой сложной частью верификации .

По мере роста уровня абстракции абстракция, верификация бросает новые вызовы.

Вы можете показать, что отдельные ячейки в последовательности блоков маршрутизатора ATM правильны, но что если есть потоки различных приоритетов? Какая следующая ячейка должна быть выбрана не всегда очевидно на самом высоком уровне. Возможно, вам придется анализировать статистику из тысячи ячеек, чтобы убедиться, что совокупное поведение является правильным. И последнее замечание: вы никогда не сможете доказать, остались ли еще ошибки

И

Fron Janic Bergeron, Sinopsis Inc
"Writing testbenches using System Verilog

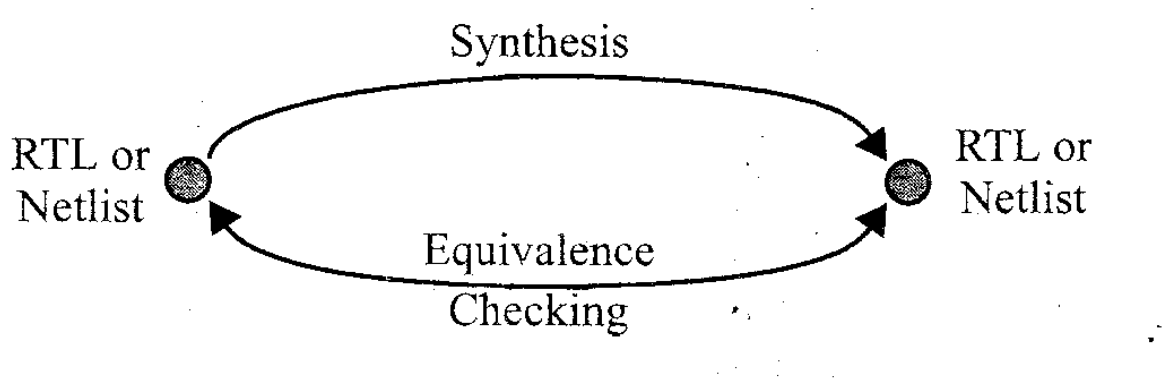


ФОРМАЛЬНАЯ ВЕРИФИКАЦИЯ математическое сравнение имплементации и спецификации или иных требований к устройству для определения не содержит имплементация нарушений спецификации

Примеры- некорректная таблица истинности, наличие недостижимых или тупиковых состояний автоматов, отсутствие некоторых соединений в синтезированной схеме

Применение формальной проверки подпадает под две широкие категориями : проверка эквивалентности и проверки property (свойства).

Проверка эквивалентности



В наиболее общем применении , проверка эквивалентности сравнивает два нетлиста чтобы некоторые элементы пост-обработки, такие как список соединений, цепочки сканирования, деревья тактовых сигналов или ручная модификация , не изменили функционирования схемы.

Другое популярное использование проверки эквивалентности – верифицировать , что список соединений правильно реализует оригинальный RTL код. Если полностью доверять инструменту синтеза, такая проверка не будет необходима . Тем не менее, инструменты синтеза это большие программные системы, которые зависят от правильности алгоритмов и библиотек . История показала, что такие системы подвержены ошибкам. В некоторых редких случаях, эта форма проверки эквивалентности используется для проверки вручную написанного RTL кода – правильно ли он представляет вентильный уровень

Проверки Эквивалентности может использоваться также для доказательства, что два RTL

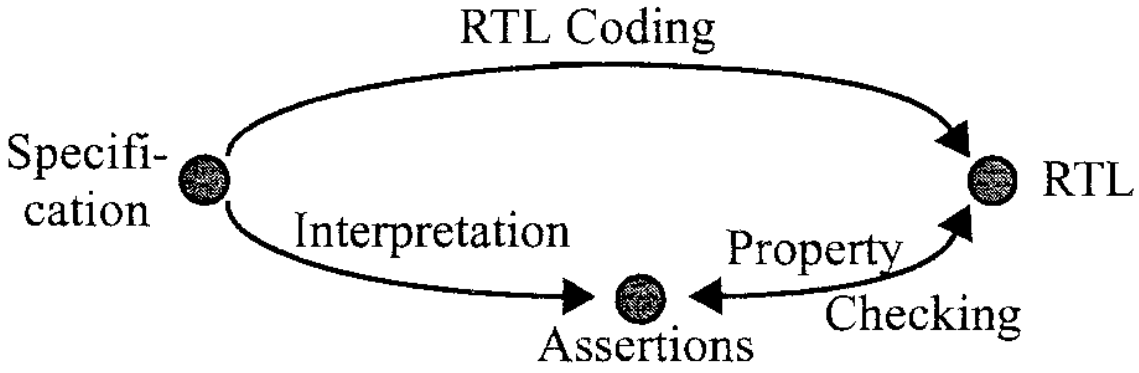
описания логически идентичны. Доказательство их эквивалентности позволяет избежать длительной работы моделирования регрессии, когда лишь незначительные нефункциональные изменений внесены в исходный код для получения лучших результатов синтеза..

Проверка properties (сущностных свойств)

Проверка сущностей это более современные путь технологии формальной верификации .В ней АССЕРЦИИ характеристик устройства формально доказываются или опровергаются. Например, все автоматы в конструкции могут быть проверены для недоступные или изолированные состояния . Более мощные средства Проверки сущностей могут , определять наличия тупиковых ситуаций при некоторых условиях

Другой тип Ассерций , что которые могут быть формально проверены относится к интерфейсам. В Спецификации понятия propertyes языка SystemVerilog указано что assertions об интерфейсах устройства фиксируются и интерпретатор пытается доказать или опровергнуть их. Например, assertion Может потребовать , что если сигнал ALE будет установлен , то либо сигнал DTACK либо ABORT в конце концов.

будут установлены ,



Reconvergence модель для контроля сущности показана на рисунке . Самым большим препятствием для технологии проверки свойств является определение спецификации доказываемой ассерции в интерпретации устройства . Только часть Ассерций , может быть доказана реально. Кроме того, для того чтобы были доказательства полезны, АССЕРЦИИ не должны быть тривиальными выдержками из поведения уже перехваченными RTL кодом. Они должны быть основаны на внешних требования, которым должен отвечать проект.

Функциональная Верификация

Функциональные Верификации проверяют проектной замысел.

Главной целью функциональной верификации является убеждение в том ,что устройство реализует предназначенные функции. Как показано на реконструктивной модель на рисунке , функциональной верификации связывает устройство с его спецификацией. Без функциональной верификации, пользователь вынужден верить, что преобразование специфицирующего документа в RTL код была выполнена без неправильного толкования назначений спецификации.

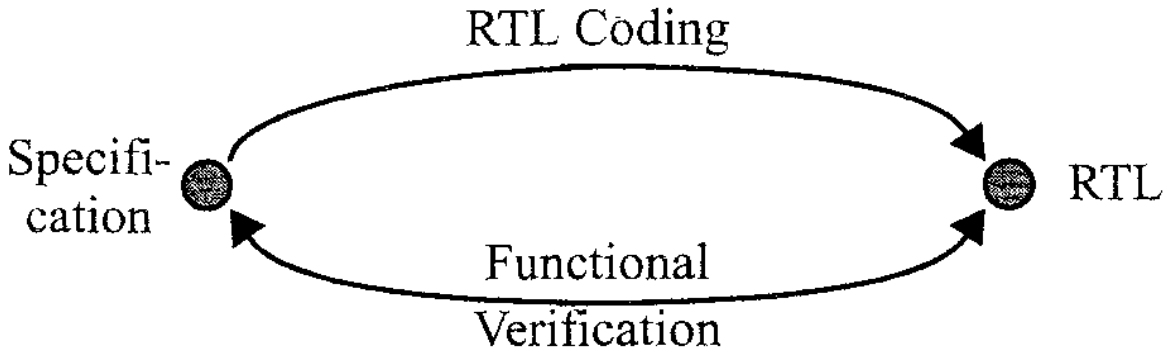


Рисунок 1-8.Функциональная проверка

Вы можете доказать наличие ошибок, но вы не можете доказать их лтсутствие

Функциональная проверка может быть выполнена с использованием трех взаимодополняющих элементарных подходов: черного ящика, белого ящика и серого ящика.

МОДЕЛИРОВАНИЕ

Моделирование является наиболее распространенным средством проверки технологии. Она называется "моделирование", поскольку она ограничена аппроксимацией реальности. Моделирование никогда не конечная цель проекта. Цель всех проектов, разработка аппаратной части, чтобы создать реальные физические проекты, которые Имитируют Ваш проекта до его реализации.

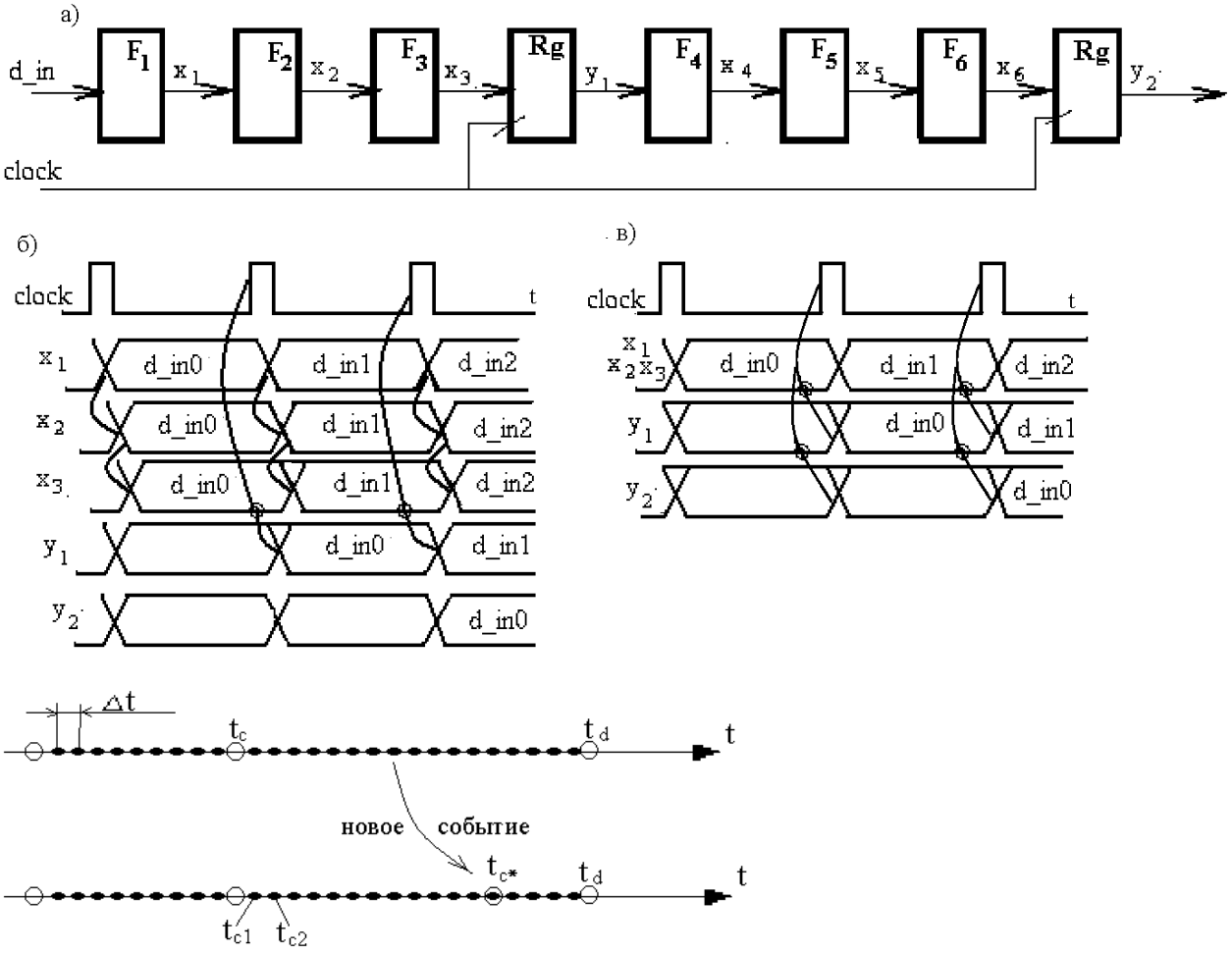
Написание тестов с использованием SystemVerilog дает возможность продавать их и получать прибыль. Эта технология проверки позволяет проектировщикам взаимодействовать с устройством , пока оно еще не производятся, и исправить недостатки и проблемы раньше.

Вы никогда не должны забывать, что моделирование является приближение к действительности

Цель testbench заключается в определении правильности организации тестируемого устройства DUT (ТсУ). Это достигается за счет следующих шагов.

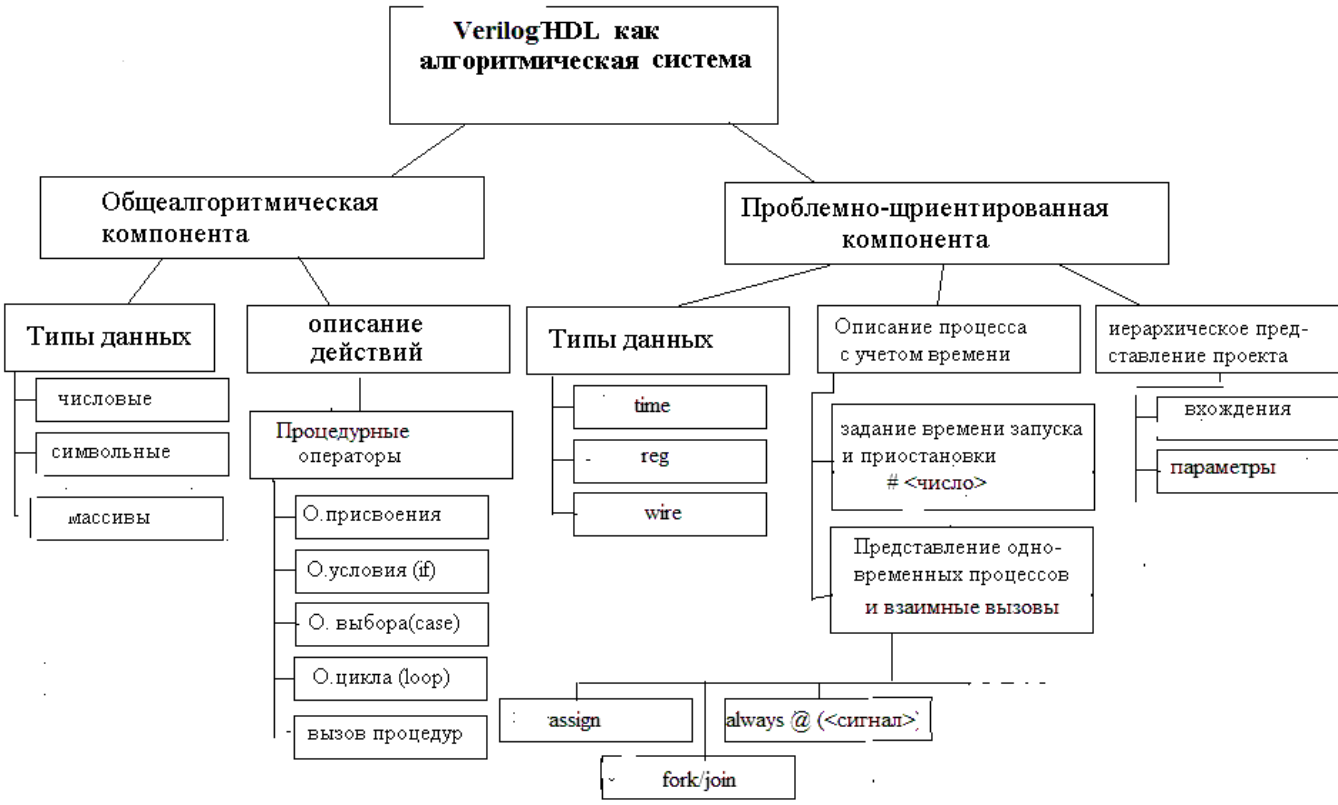
- Создание стимулов
- Подать стимул на ТсУ
- Зафиксировать реакцию
- Проверить правильность

Различают Событийное моделирование, цикло-базированное М, со-моделирование



Понятия прямого тестирования и квазислучайного.

Лекция 2


VerilogHDL как алгоритмическая система
Общеалгоритмическая компонента
Проблемно-щриентированная компонента
Типы данных
описание действий
Типы данных
Описание процесса с учетом времени
иерархическое пред-ставление проекта
числовые
символьные
массивы
Процедурные операторы
time
reg
wire
задание времени запуска и приостановки # <число>
вхождения
параметры
О.присвоения
О.условия (if)
О. выбора(case)
О.цикла (loop)
вызов процедур
Представление одно-временных процессов и взаимные вызовы
assign
always @ (<сигнал>)
fork/join

Концепции и базовые конструкции языка Verilog HDL

The types of lexical tokens in the language are as follows:

— White space
— Comment

— Operator
— Number
— String
— Identifier
— Keyword

2.3 Comments

The Verilog HDL has two forms to introduce comments. A *one-line comment* shall start with the two characters // and end with a new line. A *block comment* shall start with /* and end with */. Block comments shall not be nested. The one-line comment token // shall not have any special meaning in a block comment.

Operators

Operators are single-, double-, or triple-character sequences and are used in expressions.
Unary operators shall appear to the left of their operand. *Binary operators* shall appear between their operands. A *conditional operator* shall have two operator characters that separate three operands.

Numbers

Constant numbers can be specified as integer constants (exstly 32 bits) or real constants.

Nets

Next level

Expressions -declarations. assignments. controle expressions. functions,instances,
Clouse – не использьзуется в литературе по verilog
Constructs: block, assign, initial, always

Структура программы Verilog

<исходный текст> ::=
{! <Директива компилятора> !}

<описание> ::= < описание модуля> |! < описание примитива>
<описание модуля> ::=
module <имя модуля> « (< порт > {! ,< порт> !})» ;
{! <Декларация> |! < параллельный оператор>
!}
endmodule
|! macromodule « (< порт > {! ,< порт> !})» ;
{! <Декларация> |! < параллельный оператор>
!}
endmodule
<порт> ::=
<объявление соответствия порта>
|! . < имя порта> (« <объявление соответствия
порта>»)
<декларация> ::=

```
    < декларация параметров>
    |! < спецификация портов>
    |!  <декларация сигналов>
    |! <декларация времени>
    |! <декларация численных переменных>
    |! <декларация событий>
    |! <декларация логических ячеек>
```

<параллельные операторы> : == // continious statements

```
    < параллельное присвоение>
    |! <оператор вхождения модуля>
    |! <оператор блока>
    |! <оператор инициализации>
    |! <оператор постоянного повторения>
    |! <вызов подпрограмм>
```

Внутренние данные декларируются только один раз (задается имя и тип)
Порты декларируются трижды: в интерфейсном списке определяется имя, спецификация порта задает направление передачи (input, Output, или inout) и, (наконец, декларируется тип – wire по умолчанию)

```
 module bit_count_behave (inp, outp);
input inp;
output outp;
//wire [7:0] inp;
reg [3:0] number;
//wire [3:0]outp;
integer I;
reg[3:0] number=a+b;
  always #10 //@ (inp)/*      */
    begin
  number:= 0;
   for (i=0;i< 8;i=i+1)
      if  inp[i]='1' then
          number=number+1;
 end// always
assign outp=number;
endmodule;
```

В необязательном разделе "директивы компилятора" могут быть заданы некоторые параметры процедур компиляции и моделирования.

Наиболее употребительные директивы компилятора это 'timescale, 'define, 'include, 'ifdef, 'else, 'endif.

Тиы данных

Сигналы в VerilogHDL представляются в четырехзначном алфавите {0, 1, X, Z}. Однако для представления линий связи, к которым подключаются источники, характеризующиеся специфическими электрическими параметрами, например проходные ключи, схемы с открытым коллектором и подобные, драйверам сигнала могут быть приписаны дополнительные атрибуты – уровни силы (Strength Level) – которые доопределяют способ вычисления истинного значения сигнала на линии.

Переменные, представляющие реальные сигналы могут быть отнесены к одному из двух типов (ими массивам данных этих же типов **Цепи** и **Регистры**

Цепи представляют физические связи между структурными компонентами устройства. Сама по себе цепь не сохраняет состояние – она должна управляться драйвером, представленным оператором параллельного присвоения, причем имя драйвера совпадает с именем цепи. Это, собственно и отражается в принятом в языке Verilog названии операторов этого типа – «continious», то есть непрерывные, постоянно воздействующие на цепь.

Регистр – это обобщенное понятие, отражающее элементы, способные сохранять состояние. Программная модель предусматривает сохранение состояния от каждого присвоения до следующего.

<декларация цепи> ::=

<тип цепи> « signed» **«<право доступа>»**<диапазон>**»** «задержка>» <список переменных>;

|! <тип цепи> « signed» **«**<право доступа>»<диапазон>**»** «задержка>»

<список присвоений> ;

<тип цепи> ::=

wire | wand | wor | supply0 | supply1 | tri| tri0| tri1 |**triand** | **trior**| **trireg**

<право доступа> ::=

scalared | **vectored**

<диапазон> :=

[<константное выражение> : <константное выражение>]

<сила драйвера> ::=

(<Уровень силы 0>, <Уровень силы 1>)

Конструкция <сила драйвера> присутствует только в тех случаях, когда декларация переменной содержит присвоение, и если в тексте программы имеется несколько операторов присваивания значения этой переменной.

Уровень силы задает условия взаимного подавления сигналов от нескольких источников, подключенных к общей линии.

<декларация регистра> ::=
reg « **signed**» « «< право доступа>» <диапазон>»
<список переменных>;

```
reg a; // скалярная регистровая переменная
wire w1, w2; // декларация двух цепей
wire #10 w3= w1 && w2; //  совмещение объявления цепи и
присвоения:
       // w3 принимает значение логического И от значений
w1 и w2
       // через 10 единиц модельного времени после
изменения
       // любого из них
reg[3:0] v; //  4-х разрядный векторный регистр,
        // включающий v[3], v[2], v[1]  и v[0],
         // причем v[3] – старший разряд, а v[0] - младший
tri [15:0] busa; //  16-разрядная шина с тремя состояниями
reg signed [0:3] signed_reg; // 4-разрядный регистр,
          // код в котором
         //трактуется как число в диапазоне  от  -8 до  +7
```

Дополнительные типы данных

integer –

<декларация> := **integer** <список имен>;

Чаще используется для задания индексов и конфигурационных параметров. Если целый тип присвоен сигнальной переменной, то предполагается, что она представлена в реализации тридцатидвухразрядным кодом, и ее поведение подобно данным регистрового типа (присваивание только в процедурных операторах). Существенные отличие целых данных от регистровых — невозможность присвоения им неопределенных значений и невозможна прямая передача через порт

real - имеют свойства, подобные регистровым данным, но в языке вводится ряд ограничений на набор допустимых операций над ними *(см. разд. 3.3.3)*. Фактическое внутреннее представление определяется используемым для их интерпретации компьютером.

Строки

Строковые данные чаще всего используются для организации выдачи сообщений о возникновении тех или иных ситуаций в процессе моделирования, которые предусматривает разработчик. Строковая константа

записывается как произвольная последовательность символов, заключенная в двойные кавычки. Кроме стандартных символов клавиатуры применяются специальные конструкции управления выводом на дисплей, повторяющие соответствующие конструкции языка С: `"\n"` — перевод строки, `"\t"` — табуляция и т. д.

Если строка при исполнении программы может подвергаться модификации, то вводится переменная регистрового типа, конкретно — битовый вектор, число элементов которого достаточно для сохранения любого возможного значения строковой переменной. Каждый символ представлен восьмибитовым кодом ASCII. Специальных правил для декларации не вводится. Переменные типов wire и reg достаточной длины могут хранить строковые данные и над ними можно задавать такие же операции, как и с логическими данными

```
reg [8*17:1] stringvar, [8*(17+6):1] result_string;
  initial
    begin stringvar= "Simulation started";
        result_string={"Seance of", stringvar};
        $display(" %S", result_string);
    end;
```

Память

Reg [diapaz>] <name> [<diapaz>]

Время

Событие

Константы

Задаются либо директивой компилятора

`'define gate_delay=5`

Либо через декларацию параметров

<декларация параметров> ::=

parameter <имя>=<константное выражение>

«,<имя>=<константное выражение> »;

Программный модуль может содержать сколько угодно деклараций параметров. Тип параметра совпадает с типом константного выражения.

Примеры:

parameter x=25, f=9; // Два параметра (константы) целого типа;

parameter pi=3.14157; // Константа действительного типа;

parameter word_size=8, last_bit_number= word_size-1;

Параметры являются константами в данном Модуле, но их значение можно устанавливать в модуле высшего уровня иерархии при включении в иерархический проект

Синтаксис представления числовых констант имеет вид:

: «<Разрядность представления>» «<базовый формат>» <число>

<базовый формат> ::=

```
| 'd  // десятичный формат, цифры 0-9;
| 'h  // шестнадцатеричный формат, цифры 0-9, буквы a-f, x,z
| 'o  // восьмеричный формат, цифры 0-7, буквы x,z;
| 'b  // двоичный формат, цифры 0,1, буквы x,z;
```

5 'd 3 // пятибитовый код в десятичном представлении,
// эквивалентно 5 'b 00011.

12 'h 4x2 // двенадцатибитовый код в трехразрядном шестнадцатеричном
// представлении, эквивалентно 12 'b 0100xxxx0010.

6 'o z1 // шестибитовый код в двухразрядном восьмеричном
// представлении, эквивалентно 6 'b zzz001

Операции и выражения

Выражение — это конструкция, которая объединяет операнды и знаки операции для формирования результата.

Таблица7. *Символы операций Verilog*

Группа операций	Символы операций	Наименование	Применимость к данным типа `real`
Конкатенация	{ }		Нет
Арифметические операции	+, –	Унарные арифметические	Да
	+, –, *, /	Бинарные арифметические	Да
	%	Модуль числа	Нет
Операции сдвига	>>, <<	Сдвиг кода вправо или влево на заданное число разрядов	Нет
Операции отношения	>, >=	Больше, больше или равно	Да
	<, <=	Меньше, меньше или равно	Да
Операции сравнения	==, !=	Равно — не равно (особенности вычисления см. далее)	Да
	===, !==		Нет
Операция свертки	&, ~&	Свертка по И или И-НЕ	Нет
	|, ~|	Свертка по ИЛИ или ИЛИ-НЕ	Нет
	^	Свертка по "исключающему ИЛИ"	Нет
	^~ или ~^	Свертка по "исключающему ИЛИ-НЕ"	Нет

Поразрядные операции	~	Инверсия	Нет
	&, ~&	Поразрядное И (И-НЕ)	Нет
	|, ~|	Поразрядное ИЛИ (ИЛИ-НЕ)	Нет
	^	Поразрядное "исключающее ИЛИ"	Нет
	^~ или ~^	Поразрядное "исключающее ИЛИ-НЕ"	Нет
Логические операции	!	Логическая инверсия	Да
	&&	Логическое И	Да
	||	Логическое ИЛИ	Да
Условная операция	?	Если <условие>?, то <значение 1> : иначе <значение 2>	

(&A)&&(&b) a|b

Непрерывное присваивание

Приемником в операторе непрерывного присваивания может быть только переменная типа "цепь", скалярная или векторная. Синтаксис оператора параллельного присваивания имеет вид:

<оператор параллельного присваивания> ::=

assign « <сила драйвера> » «<задержка>» <присваивание> «,<присваивание>»;

<задержка> ::=

<параметр задержки>

| # (<параметр задержки>, <параметр задержки> «,<параметр задержки>»)

<параметр задержки> ::=

<Выражение времени>

| <Выражение времени> : <Выражение времени> : <Выражение времени>

```
module  delay_example;
  reg s1,s2,v;
  wire result;
    assign # ( 10,5,15)
                 result=(s1==1)? : 1'b1 : 1'bz,
                  result=(s2==1)? :  v : 1'bz;
    initial
         begin s1=0; s2=0;v=0;
                #10 s1=1;
                #20 s1=0;
                #25 s2=1;
                #15 v=1;
                 #10 v=0;
                #10 s2=0;
                #20  $finish;
          end
endmodule
```