Оглавление

МГТУ им. Н.Э. Баумана

Факультет «Информатика и Системы Управления»

Кафедра ИУ-3 «Информационные системы и телекоммуникации»

ФЕДОРОВ СЕРГЕЙ ВЛАДИМИРОВИЧ

ЯЗЫК VERILOG

Электронное учебное пособие

МОСКВА

2009 год МГТУ им. Баумана

Аннотация

В пособии рассматриваются основные возможности языка описания аппаратного состава Verilog.

В настоящее время, в связи со значительным ростом степени интеграции цифровых схем, разработка цифровых схем осуществляется на высоком уровне абстракции, в том числе на специальных языках описания аппаратного состава. Язык Verilog – один из наиболее распространенных языков описания аппаратного состава, предоставляющий необходимые возможности для описания и моделирования цифровых схем.

Основное внимание в данном пособии уделено реализации синтезируемых описаний и соответствующему подмножеству языка. Рассмотрены основы создания тестовых модулей (тестбенчей) для моделирования в САПР. Пособие может использоваться в курсах, посвященных логическому проектированию схем и проектированию цифровых схем на программируемых логических интегральных схемах (ПЛИС).

1. Введение

1.1. Проектирование цифровых устройств на языках описания аппаратного состава

До появления микросхем высокой степени интеграции традиционным подходом к проектированию цифровых устройств был схемный ввод на уровне блоков, соответствующих стандартным микросхемам низкой степени интеграции и примитивов, таких, как логические вентили и триггеры.

В настоящее время, в связи со значительным ростом степени интеграции, произошел переход к описанию цифровых схем на высоком уровне абстракции. При этом часто используется подход “сверху вниз”, причем нижним уровнем описания часто является не описание функционирования модулей на уровне регистров и вентилей, называемое “уровнем регистровых передач” (RTL – register transfer level), а описание поведения схемы, ее логики работы, называемое “поведенческим описанием” (behavioral description level).

Для описания схем и их моделирования были разработаны так называемые языки описания аппаратного состава (HDL – hardware description language). Современные средства проектирования поддерживают моделирование и синтез логических схем, функционирование которых описано на этих языках.

Использование языков описания аппаратного состава обеспечивает ряд важных преимуществ:

- использование высокоуровневых описаний позволяет сократить сроки проектирования;
- спроектированные модули могут быть синтезированы для любой аппаратной платформы и технологической базы;
- возможности современных синтезаторов позволяют эффективно исследовать пространство возможных решений при синтезе и поиска компромисса по быстродействию, энергопотреблению и объему логики;

- упрощается поддержка, аудит и верификация проекта.

В настоящее время наиболее популярными языками, применяемыми для описания цифровых систем, являются языки Verilog HDL и VHDL.

В проектировании цифровых систем можно выделить несколько уровней детализации, которые одновременно соответствуют переходам между процессами разработки при реализации подхода “сверху вниз”:

- Требования к системе
- Формальное представление
- Архитектура
- Регистры и логика
- Вентили
- Транзисторы

Требования к системе формализуются в виде технического задания, технических требований и прочей проектной документации.

Формальное представление может быть осуществлено, например, в виде блок-схемы алгоритма, диаграммы потока данных в системе или конечного автомата. В настоящее время имеются САПР, поддерживающие такие представления системы и реализующие их автоматическую трансляцию в синтезируемые описания на языках описания аппаратного состава.

Однако пока что средства трансляции формального представления на более низкие уровни не позволяют получить гарантированно эффективных реализаций, так как при проектировании аппаратуры требуется учитывать ее особенности. Современные высокоуровневые средства описания могут вселить в неопытного разработчика надежду на “легкое” и “быстрое” проектирование, без необходимости разбираться с особенностями схемной реализации комбинационных функций, триггеров, устройств памяти и т.д..

Например, можно создать цифровой фильтр в Матлабе, экспортировать в Verilog и синтезировать в аппаратную реализацию. К сожалению, при таком подходе зачастую генерируются крайне неэффективные аппаратные реализации, которые требуют в дальнейшем значительной доводки. Правка кода на более низких уровнях зачастую не дает должного эффекта, так как грамотные технические решения должны приниматься уже на уровне формального представления, исходя из знания особенностей и ограничений аппаратуры. По этой причине начинающему разработчику следует сначала детально разобраться в особенностях аппаратной реализации на уровне триггеров и вентилей.

Описание на уровне архитектуры в Verilog обычно реализуется как поведенческое описание. При этом используются операторы, схожие с операторами обычных языков программирования – условный оператор, оператор выбора и т.д. Таким образом описывается поведение схемы и одновременно задается архитектура, как в части комбинационных функций, так и регистровой логики.

Например, следующий код:

```
always @(*)
begin
      a=b+d;
end;
```

читается как “всегда при изменении любого входного сигнала записать в a сумму текущих значений b и d” и приводит при переходе на более низкие уровни к синтезу сумматора (рис. 1).

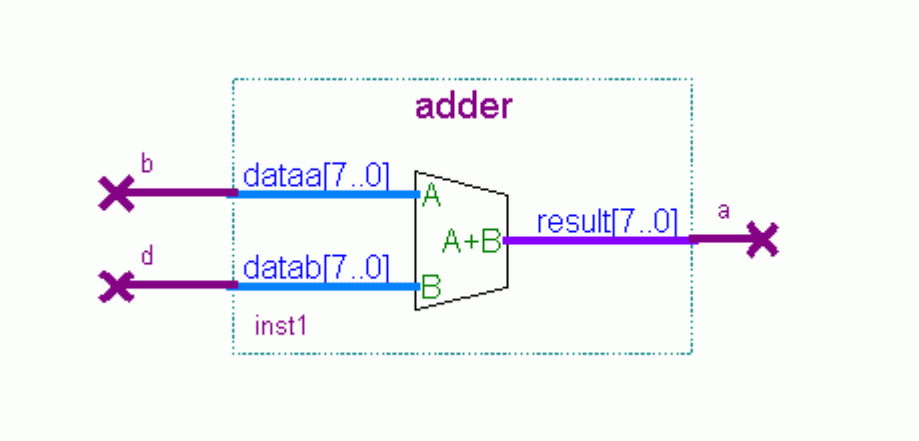

Рис. 1. Сумматор

Код:

```
always @(posedge clk)
begin
	a<=b+d;
end;
```

читается как “всегда при поступлении положительного фронта тактового импульса clk записывать в a сумму b и d в этот момент времени” и приводит к синтезу сумматора с регистром на выходе (рис. 2).

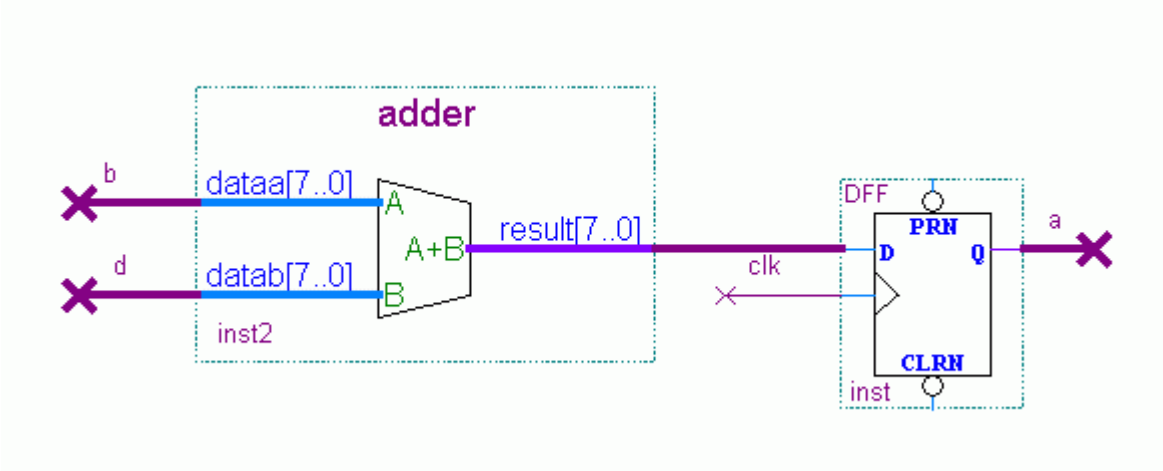

Рис. 2. Сумматор с регистром на выходе

Представление схемы в виде регистров и логики на Verilog реализуется на уровне регистровых передач. В сравнении с рассмотренным выше примером поведенческого

описания сумматора на уровне регистровых передач сумматор был бы описан логическими уравнениями, формирующими каждый разряд результата как комбинационную функцию от входов. Также к этому уровню следует отнести схемное представление.

Более низкие уровни соответствуют отображению схемы в аппаратный базис. Для программируемых логических интегральных схем (ПЛИС) это могут быть таблицы перекодировки или уравнения в базисе И-ИЛИ-НЕ, для базовых матричных кристаллов (БМК) – библиотечные элементы из библиотеки производителя микросхем и т.д. Как правило, получаемое при переходе на этот уровень представление схемы называется “списком соединений” (netlist). Список соединений как перечисление используемых базовых функциональных блоков (вентили, триггеры) и их соединений может также быть представлен на языках описания аппаратного состава. На Verilog этот уровень называется уровнем вентилей (gate level). Нижний уровень – уровень транзисторов (switch level) может использоваться при проектировании заказных микросхем и позволяет описывать схемы на уровне транзисторов.

Таким образом, язык Verilog позволяет описывать схемы на четырех уровнях: архитектуры (в виде поведенческого описания), регистров и логики (уровень регистровых передач), вентилей (списки соединений) и транзисторов.

В данном руководстве рассмотрены основные возможности языка по описанию и моделированию синтезируемых модулей на поведенческом уровне и уровне регистровых передач.

1.2. История Verilog

Язык Verilog был разработан в середине 80-х годов Филом Мурби в компании Gateway Design Automation как внутреннее средство описания цифровых схем для анализа и моделирования. В процессе развития языка было также разработано средство моделирования Verilog-XL, ставшее успешным коммерческим продуктом. Существенным

преимуществом языка Verilog по сравнению с существовавшими средствами моделирования явилось наличие интерфейса с языками программирования PLI, ориентированного на язык C и позволяющего вызывать код, реализованный на языке C из описания цифровой системы на языке Verilog. В 1989 году Gateway Design Automation была приобретена фирмой Cadence Design Systems. Для развития и популяризации языка в 1990 году Cadence сделала спецификацию языка общедоступной. В 1995 году был принят стандарт языка Verilog IEEE Std 1364-1995.

В настоящее время наиболее распространен и поддерживается большинством синтезаторов и пакетов моделирования язык, соответствующий скорректированному стандарту IEEE Std 1364-2001. Часто эта версия языка называется Verilog-2001. По сравнению с версией Verilog-1995 стандарт содержит большое количество модификаций, повышающих надежность кода и облегчающих проектирование.

В 2005 году был принят новый стандарт IEEE Std 1364-2005. Стандарт содержит незначительные уточнения спецификации языка в сравнении с Verilog-2001. В разделе, относящемся к PLI, сохранен только один тип интерфейса.

В развитие языка и расширение его возможностей для высокоуровневого описания и верификации систем в настоящее время разработан объектно-ориентированный язык SystemVerilog, стандартизированный как IEEE Std 1800-2005.

1.3. Архитектура ПЛИС

В данном руководстве проектирование на Verilog рассматривается в основном для обучения проектированию цифровых схем на программируемых логических интегральных схемах (ПЛИС). Рассмотрим основные особенности архитектуры ПЛИС.

ПЛИС – это полностью изготовленная специализированная интегральная схема, которая доспециализируется (программируется) разработчиком устройства. Основой ПЛИС

является массив функциональных преобразователей (ФП), каждый из которых позволяет реализовать программируемую пользователем комбинационную и (или) регистровую функцию.

В большинстве современных архитектур ФП выделяется комбинационная и регистровая часть. Наиболее распространенный триггер в регистровой части – это синхронный D-триггер с разрешением и асинхронным сбросом.

В комбинационной части используется три типа архитектур:

- на основе матрицы И-ИЛИ;
- на основе таблиц перекодировки (просмотровых таблиц);
- мультиплексорная.

Наиболее распространены первые два типа. Рассмотрим их более подробно.

Матрицы И-ИЛИ применяются большинством производителей только в ПЛИС низкой и средней степени интеграции. Матрица И-ИЛИ реализует программируемую ДНФ, которая формируется синтезатором из описания проекта пользователя.

Типичным представителем микросхем программируемой логики на основе архитектуры И-ИЛИ является группа семейств MAX3000/MAX7000 фирмы Altera (рис. 4).

В локальную матрицу соединений (матрицу И) поступают сигналы от глобальной матрицы соединений, причем на входе формируется прямое значение сигнала и его инверсия. На входах в каждый ФП (традиционно называемый макроячейкой в микросхемах на основе матрицы И-ИЛИ), формируется произвольная конъюнкция от сигналов в матрице И. Матрица выбора термов распределяет входные конъюнктивные термы в макроячейке, причем для реализации логических функций они поступают на входы элемента ИЛИ и элемента сложения по модулю 2. Таким образом, реализуется ДНФ с возможностью использования сложения по модулю 2, что позволяет реализовать в данных микросхемах

любую логическую функцию.

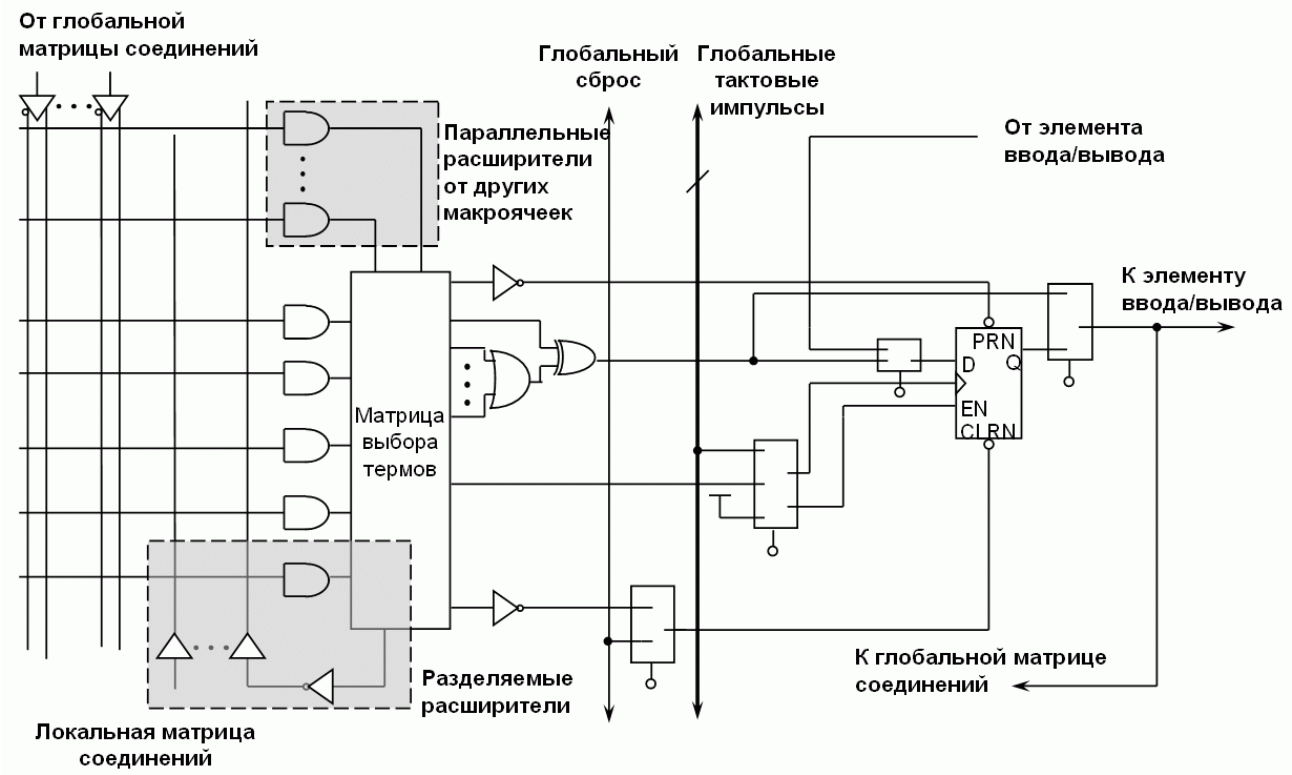

Рис. 4. Архитектура матриц И-ИЛИ. Макроячейка микросхемы семейства MAX7000A

Параллельные и разделяемые расширители связывают макроячейки между собой и позволяют более эффективно реализовывать комбинационные схемы.

Конфигурация реализуемой логической функции и коммутации сигналов в ФП (отмечена на рисунке мультиплексорами с отмеченными кругом входами управления) осуществляется на этапе программирования микросхемы и в режиме работы не изменяется. Например, в качестве источника асинхронного сброса триггера может использоваться сигнал с линии глобального сброса или сигнал, сформированный в матрице И.

Наиболее распространенной архитектурой в ПЛИС средней, высокой и сверхвысокой степени интеграции является архитектура на основе таблиц перекодировки. В настоящее время наиболее распространены 4-х входовые таблицы перекодировки. Такая таблица перекодировки представляет из себя запоминающее устройство емкостью 16 бит, в которое

записывается таблица истинности любой логической функции от 4-х переменных. Таким образом, любая комбинация 4-х входных переменных однозначно определяет адрес однобитной ячейки, в которой хранится значение функции.

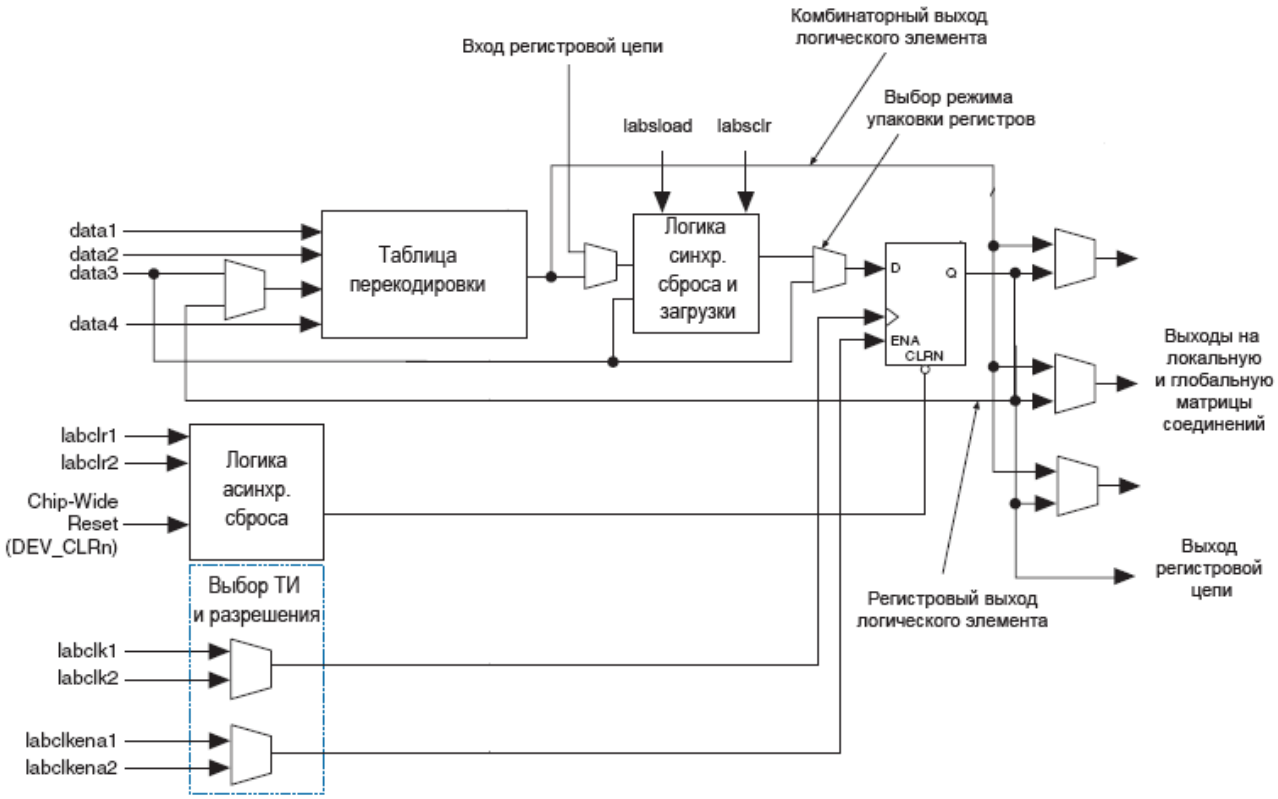

Рис. 5. Архитектура на основе таблиц перекодировки. Логический элемент микросхемы семейства Cyclone II.

Типичным представителем недорогих микросхем программируемой логики средней и высокой степени интеграции с такой архитектурой является группа семейств Cyclone фирмы Altera (рис. 5).

Входными сигналами в логический элемент из матрицы соединений являются сигналы данных (data1…4) и сигналы управления триггером (тактовые импульсы labclk, сигналы разрешения labclkena, сигналы синхронного сброса labsclr и загрузки labsload, а также сигналы асинхронного сброса labclr).

На выход может быть передан как комбинационный, так и регистровый выход логического элемента. В одном логическом элементе могут быть реализованы одновременно независимые комбинационная и регистровая функции (см. мультиплексор “Выбор режима упаковки регистров”).

В данном семействе логический элемент также имеет специальный режим работы, в котором он реализует один разряд двоичного сумматора с цепочным переносом в следующий логический элемент (на рисунке не показан).

В современных ПЛИС часто также имеются вспомогательные модули:

- модули оперативной памяти (статическое ОЗУ);
- аппаратно реализованные умножители;
- модули поддержки высокоскоростных интерфейсов приема-передачи данных (RapidIO, PCI Express и т.д);
- модули формирования тактовых импульсов на основе ФАПЧ.

При разработке проектов на языках описания аппаратного состава большая часть этих модулей настраивается и задействуется через вызов библиотечных функций, входящих в состав библиотек САПР ПЛИС. Исключением, пожалуй, является операция умножения, которая может быть преобразована большинством синтезаторов в задействование блоков аппаратных умножителей.

1.4. Пример реализации модуля на Verilog

Разберем в качестве примера файл dec.v. Модуль реализует модель железнодорожного семафора (рис. 6). Проходу поезда соответствует вход clr с активным низким уровнем. В момент прохода поезда загорается красный свет (состояние 0), далее по мере поступления тактовых импульсов на вход clk происходит смена состояний семафора в следующем порядке: желтый (состояние 1), желтый и зеленый (состояние 2), зеленый (состояние 3). По достижении состояния 3 семафор остается в нем до следующего прохода поезда.

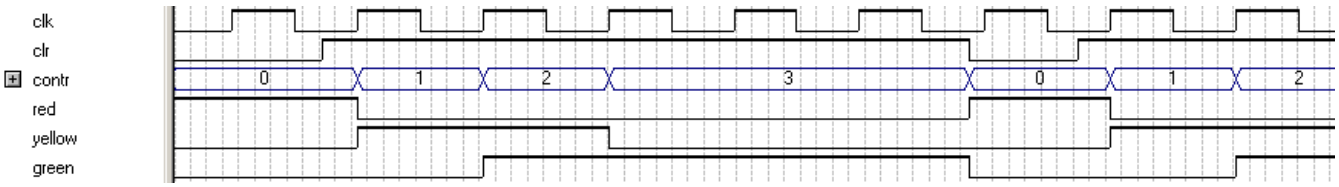

Рис. 6. Диаграмма работы семафора

Модуль реализован как синхронный счетчик до 3 со сбросом и специальным дешифратором номера состояния на выходе. Дешифратор комбинационно преобразует двоичный номер состояния, формируемый счетчиком, в значения трех выходных сигналов – red (красный), yellow (желтый) и green (зеленый).

```
1 module dec(clk, clr, contr, red, green, yellow);
2
3     //порты
4     input clk, clr;
5     output reg [1:0] contr;
6     output reg red, green, yellow;
7
8     //синхронная логика счетчика
9     always @ (posedge clk or negedge clr)
10    begin
11          if (!clr) contr<=0;
12          else if (contr!=3) contr<=contr+1;
13    end
14
15    //комбинационная логика декодирования
16    always @ (*)
17          case (contr)
18          2'b00: begin red=1'b1; yellow=1'b0; green=1'b0; end
19          2'b01: begin red=1'b0; yellow=1'b1; green=1'b0; end
20          2'b10: begin red=1'b0; yellow=1'b1; green=1'b1; end
21          2'b11: begin red=1'b0; yellow=1'b0; green=1'b1; end
22          endcase
23
24 endmodule
```

Строка 1.

Объявление модуля. Имя модуля - dec. Если это модуль верхнего уровня, имя должно соответствовать имени файла. Также в объявлении определены имена входных и выходных портов модуля.

Строки 4-6

Объявление входных и выходных портов. Определяет входы (input) и выходы (output) модуля для перечисленных в строке 1 имен портов. Входы и выходы могут быть как одноразрядными: clk, clr, red, green, yellow, так и многоразрядными (шинами): contr[1:0]. В объявлении шины индексы в квадратных скобках определяют диапазон индексов проводников, входящих в шину. Одновременно может быть определен тип выхода, в данном случае регистр (reg), по умолчанию, если тип не указан – проводник (wire).

Строка 9

Начало процедурного блока always, описывающего регистровую логику. При возникновении события, определенного в списке чувствительности @(…), однократно выполняются операторы процедурного блока. Если процедурный блок содержит более одного оператора, для их объединения используется блок begin…end.

В данном случае в списке чувствительности учитываются только сигналы clk и clr. При этом значение, которое формируется в регистре contr, от clk не зависит, а зависит от других узлов. Таким образом, clk – это вход синхронизации (тактового импульса), изменение которого приводит лишь к исполнению процедурного блока. Общепринято, что чувствительность к положительному/отрицательному фронту тактового импульса, определенная соответственно ключевыми словами posedge/negedge, ведет к синтезу триггеров, синхронизируемых по фронту.

Также в списке чувствительности указан сигнал асинхронного сброса clr, срабатывающий по отрицательному фронту. Читатель, знакомый с цифровой схемотехникой, заметит, что вход асинхронного сброса в триггерах активен не в момент фронта, а имеет активный низкий уровень (то есть, триггер сброшен и не реагирует на

тактовые импульсы, пока на входе асинхронного сброса установлен низкий уровень). Однако, такое описание является одной из особенностей применения языка Verilog – именно так принято описывать логику на основе синхронных триггеров с асинхронными управляющими сигналами.

Строка 11

Если сигнал clr находится в низком уровне (вне зависимости от того, что стало условием для выполнения процедурного блока – положительный фронт clk или отрицательный clr), в регистры contr записывается 0. Такое описание соответствует логике работы по уровню для асинхронного входа триггера.

Строка 12

Если сигнал clr находится в высоком уровне (неактивен), описываем логику работы счетчика. При срабатывании блока (а в данной ветви мы можем оказаться только при поступлении положительного фронта тактового импульса), содержимое регистра contr будет увеличено на 1, если оно еще не равно 3. В противном случае значение регистра не изменяется. Обратите внимание на используемый оператор неблокирующего присваивания <=. При описании регистровой логики рекомендуется использовать неблокирующие присваивания. Более подробное рассмотрение типов операторов присваивания и рекомендации по описанию комбинационных и регистровых схем приведены в разделах 2.3.11 и 3.1.

Строка 16

Начало процедурного блока, описывающего комбинационную логику. В отличие от регистровой логики, в комбинационной схеме значение на выходе может измениться при изменении любого входного сигнала. Это обозначается списком чувствительности @(*) и обычно используется при описании комбинационных схем.

Строки 17-22

Оператор выбора, реализующий функцию дешифратора. Обратите внимание на то,

что узлы red, green, yellow имеют тип reg, несмотря на то, что регистрами сами не являются (очевидно, что оператор case в данном примере описывает комбинационную логику без обратных связей). Аналогично строке 12 обратите внимание на используемые операторы блокирующего присваивания =. При описании комбинационной логики рекомендуется использовать блокирующие присваивания. Более подробное рассмотрение типов операторов присваивания и рекомендации по описанию комбинационных и регистровых схем приведены в разделах 2.3.11 и 3.1.

Строка 24

Модуль должен заканчиваться ключевым словом endmodule.

2. Язык Verilog

2.1. Лексические элементы

Язык Verilog чувствителен к регистру, за исключением особо оговоренных случаев.

2.1.1. Комментарии

Комментарии вводятся аналогично C/C++.
Однострочный комментарий открывается двойной косой чертой (//)
Многострочные комментарии заключаются между /* и */.

Пример:

```
assign a=c+d;     //реализуем сумматор - однострочный комментарий
/*еще один способ описать сумматор - данный комментарий
   требует больше одной строки */
always @(*)
begin
      a=c+d;
end
```

2.1.2. Числа

Целочисленные константы в Verilog задаются в десятичной, шестнадцатеричной, восьмеричной и двоичной системах счисления. В случае, если система счисления не задана, используется десятичная система счисления, при этом число трактуется как целое число со знаком.

Пример:

```
325 и 'd325 – десятичное число
'hFH – шестнадцатеричное число
'o34 – восьмеричное число
'b00101 – двоичное число
```

При задании значения может быть указана разрядность числа. Если разрядность явно не указана, разрядность устанавливается в соответствии с разрядностью выражения, в котором используется константа

Пример:

```
5'b00101 – пятиразрядное двоичное число
4'd12, 4'hC, 4'o14, 4'b1100 – одно и то же четырехразрядное число с разными основаниями
```

Также символом s перед указанием системы счисления может быть указано знаковое/беззнаковое представление, при этом для представления знаковых чисел используется дополнительный код.

Пример:

```
-4'h1, 4'shF - это одно и то же число, равное 1111 в двоичном виде и трактуемое в дополнительном коде как -1
```

При записи чисел по основаниям 2,8,16 могут использоваться символы x – значение не известно (любое значение) и z (или ?) – третье (высокоимпедансное) состояние. Примеры использования и разница между значения будут рассмотрены далее.

Пример:

```
4'bx011
8'hzF
```

В записи чисел регистр не важен. Для улучшения читаемости в записи числа может использоваться символ подчеркивания. Для корректного, читаемого и легко модифицируемого описания желательно использовать численные константы с явно заданными разрядностью и основанием системы счисления.

Также могут использоваться вещественные числа, при этом при компиляции они преобразуются в двоичные последовательности разрядности 64 – формат double по стандарту IEEE754

Пример:

```
1.3e-5
-0.1E12
0.314_159_265_E1
```

2.1.3. Строки

Строки задаются как последовательность символов, заключенная в двойные кавычки. При использовании строк в операторах они трактуются как беззнаковые целочисленные константы, представленные последовательностью 8-битных значений символов в коде ASCII. Строковые переменные должны объявляться как переменные типа reg и иметь разрядность, равную длине строки, умноженной на 8.

Пример:

```
reg [8*12:1] stringvar;
initial begin
      stringvar = "Hello world!";
      $display("%s is stored as %h",stringvar,stringvar");
end
```

В окне средства моделирования будет выведено

```
Hello world is stored as 48656c6c6f20776f726c64
```

2.1.4. Идентификаторы

Для задания идентификаторов могут использоваться английские буквы, цифры, знак $ и символ подчеркивания. Первым символом идентификатора не может быть цифра и символ $.

Пример:

```
din          //обычное задание идентификатора
shift_reg_in      //для улучшения читаемости можно использовать подчеркивания
node$657     //так можно нумеровать узлы, описывая списки соединений
_read        //так можно обозначить сигнал с активным низким уровнем
```

2.1.5. Ключевые слова

Ключевые слова Verilog приведены в Приложении 1. Заданные пользователем идентификаторы не должны повторять ключевые слова.

2.2. Типы данных

2.2.1. Набор значений

Используется 4 значения:

- 0 – логический ноль, ложь
- 1 – логическая единица, истина
- x – значение не известно (любое значение)
- z – третье (высокоимпедансное состояние)

2.2.2. Проводники

Тип wire представляет физический проводник в устройстве и используется для соединения вентилей и модулей. Также могут использоваться дополнительные типы проводников. Типы wand и wor позволяют присваивать переменной более чем одно значение, которые объединяются соответственно по логическому И и ИЛИ. Тип tri позволяет моделировать шину с третьим состоянием, причем в один момент времени все источники кроме, возможно, одного должны иметь значение z. Присвоение значений проводникам осуществляется в непрерывных присваиваниях assign.

Пример:

```
wire a,b;
wand d;

assign a=1'b1;
assign b=1'b0;
assign d=a;
assign d=b; //чему будет равен d?
```

2.2.3. Регистры

Тип reg представляет объект, который может хранить присвоенное ему значение от одного присваивания до другого. Присвоение значений регистрам осуществляется в процедурных присваиваниях в процедурных блоках always.

Пример:

```
reg a,b;
always @(a) b=a;
```

2.2.4. Входы, выходы и двунаправленные порты

Ключевые слова input, output и inout объявляют соответственно входы, выходы и двунаправленные порты модулей. Входные и двунаправленные порты имеют тип проводника (wire), выходные порты по умолчанию также имеют тип проводника, однако также могут явно конфигурироваться как reg, wand, wor и tri.

Пример:

```
module sample(a,b,c,d);
input a;          //вход a
output b;         //выход b, тип wire
output reg c;     //выход c, тип reg
inout d;          //двунаправленный порт d

      assign b = a;     //wire можно присвоить значение
                        //только в непрерывном присваивании
      always @(*)
      begin
            c=a;        //типу reg можно присвоить значение
      end               //только в процедурном присваивании
endmodule
```

Еще раз обратите внимание на разницу между применением типов reg и wire в приведенном примере! Определение и вопросы применения процедурных и непрерывных присваиваний

будут рассмотрены в п. 2.3.11.

2.2.5. Векторы

Приведенные выше объявления проводников и регистров создают скалярные (одноразрядные) переменные. Для объявления шин, многоразрядных регистров и портов используются векторы.

Пример:

```
wire [15:0] databus;
reg [15:0] dreg;
```

Первый указанный индекс соответствует старшему (наиболее значимому) биту, второй - младшему (наименее значимому).

Обращение ко всему вектору возможно по имени, например

```
dreg=databus;
```

Обращение к одиночному проводнику (bit-select) или поддиапазону (part-select) вектора осуществляется с указанием индекса/диапазона индексов

```
dreg[0]=…;
dreg[3:2]=…;
```

Для указания знакового представления используется ключевое слово signed

```
reg signed [15:0] dreg;
```

2.2.6. Массивы

Для объявления групп как скаляров, так и векторов используются массивы. Поддерживаются массивы с несколькими индексами. При обращении к массиву, в отличие от вектора, не допускается обращения ко всему массиву или к поддиапазону индексов. Обязательно указание значения каждого индекса.

Пример:

```
wire [15:0] data_src [7:0];   //массив из 8 16-ти разрядных шин
reg [15:0] dreg [1023:0];     //массив из 1024 16-ти разрядных регистров
```

Объявленный выше массив регистров трактуется как объявление памяти. Однако, для задействования аппаратно реализованных модулей памяти в ПЛИС лучше использовать библиотечные функции, входящие в состав библиотек САПР ПЛИС.

2.2.7. Тип integer

Переменные типа integer – переменные общего назначения, применяемые в основном в качестве индексов циклов, параметров и констант. По умолчанию имеют тип reg, при этом не ассоциируются с аппаратными регистрами. Данные в переменной типа integer трактуются как целое число со знаком, в то время как при объявлении переменной типа reg значение в ней трактуется как целое число без знака.

Ниже приведен полнофункциональный пример комбинационного модуля, определяющего количество единиц в 32-х разрядном двоичном числе.

Пример:

```
module countones(din,ones);

      input [31:0] din;
      output reg [5:0] ones;

      integer i;

      always @(*)
      begin
            ones=0;
            for (i = 0; i<32; i=i+1)
            begin
                  if (din[i]) ones=ones+1;
            end
      end
```

```
endmodule
```

2.2.8. Тип time

Используется только при моделировании, не используется в синтезируемых описаниях. Переменная типа time хранит 64-битное значение, которое считывается из системной задачи $time, возвращающей текущее время моделирования.

Пример:

```
integer i = 0;
time time_points [1023:0];
…
always @(*)
begin
      time_points[i]=$time; //сохраняем в массив моменты выполнения блока
      i=i+1;
      …
end
```

2.2.9. Параметры

Поддерживаются два типа параметров. Параметр типа localparam – это константа, объявляемая в данном модуле. Параметр типа parameter – это константа, которая может быть настроена при создании экземпляра модуля (см. 2.4). Это обеспечивает возможность создания параметризируемых модулей (например, с настраиваемой разрядностью, режимом работы и т.д.). После имени параметра указывается его значение по умолчанию, используемое, если в модуле, задействующем данный модуль, значение параметра не будет переопределено. Параметры могут объявляться с указанием разрядности для использования в качестве констант при присваивании проводникам и регистрам.

Пример:

```
parameter WIDTH = 16;
localparam [3:0] state0 = 4'b0101;
```

```
localparam [3:0] state1 = 4'b1100;
```

2.3. Операторы

2.3.1. Арифметические операторы

Операторы могут применяться в непрерывных и процедурных присваиваниях (см. 2.3.11), а также при определении параметров. Операторы + и – могут применяться и как бинарные, и как унарные операторы.

+a	унарный плюс
-a	унарный минус (смена знака числа в дополнительном коде)
a+b	Сложение
a-b	Вычитание
a*b	Умножение
a/b	Деление
a%b	остаток от деления (a по модулю b)
a**b	a в степени b

Примеры:

```
a=b/c;      //деление поддерживается при синтезе, однако такие сложные
            //функции лучше реализовывать с применением
            //оптимизированных библиотечных модулей
d=(d+1)%5;  //считаем по модулю 5
e=-d;       //унарный минус
```

Оператор вычисления остатка от деления при делении чисел со знаком возвращает результат со знаком первого операнда.

Поддержка различных операторов при синтезе должна уточняться в руководстве по

используемому синтезатору. Например, оператор возведения в степень может иметь такие ограничения на поддержку синтеза, как необходимость использования константы в основании или показателе.

2.3.2. Операторы сравнения

Операторы сравнивают два операнда одинаковой разрядности и возвращают однобитное значение 1 или 0. В случае, если сравниваются два операнда, явно заданные как знаковые числа (integer и signed reg или целочисленная константа без указания разрядности и основания), осуществляется сравнение чисел в дополнительном коде. В противном случае операнды трактуются как числа без знака.

a<b	a меньше b
a>b	a больше b
a<=b	a меньше или равно b
a>=b	a больше или равно b
a==b	a равно b, для операндов с x и z результат равен x
a!=b	a не равно b, для операндов с x и z результат равен x
a===b	a равно b, осуществляется сравнение x и z в операндах
a!==b	a не равно b, осуществляется сравнение x и z в операндах

Последние два оператора сравнения, называемые case equality, осуществляют побитовое сравнение операндов с сопоставлением значений x и z аналогично значениям 0 и 1. Результат этих операторов всегда определен и равен 0 или 1.

2.3.3. Побитовые операторы

Операторы осуществляют побитовое применение логической операции к операндам. Разрядность результата равна разрядности операндов.

~a	побитовое НЕ

a&b	побитовое И
a|b	побитовое ИЛИ
a^b	побитовое исключающее ИЛИ
a~^b	побитовое инверсное исключающее ИЛИ

При наличии в разрядах операндов значений x и z, в случае, если разряд второго операнда не определяет однозначно результат операции (например, для операндов со значениями 1 и x результат функции ИЛИ равен 1), результатом выполнения будет x в данном разряде.

2.3.4. Логические операторы

Операторы осуществляют логическое сравнение двух операндов и возвращают однобитное значение 1 или 0. Неоднозначность, которая может возникнуть при наличии в операндах значений x или z, приводит к формированию результата x.

!a	логическое НЕ
a&&b	логическое И
a||b	логическое ИЛИ

2.3.5. Операторы сокращения

Унарные операторы сокращения применяют одну побитовую операцию ко всем битам операнда и возвращают однобитное значение 1 или 0.

&a	сокращение по И
~&a	сокращение по И-НЕ
|a	сокращение по ИЛИ
~|a	сокращение по ИЛИ-НЕ
^b	сокращение по исключающему ИЛИ
~^b	сокращение по инверсному исключающему ИЛИ

При наличии в разрядах операнда значений x и z, в случае, если другие разряды не определяют однозначно результат операции (например, для функции ИЛИ один из разрядов равен 1), результатом выполнения будет x.

2.3.6. Операторы сдвига

Определены операторы арифметического и логического сдвига влево и вправо.

a<<b	логический сдвиг a влево на b позиций
a>>b	логический сдвиг a вправо на b позиций
a<<<b	арифметический сдвиг a влево на b позиций
a>>>b	арифметический сдвиг a вправо на b позиций

Арифметический и логический сдвиг влево выполняются одинаково, при этом в освобождающиеся разряд заносятся нули. При логическом сдвиге влево в освобождающиеся разряды заносятся нули, то время как при арифметическом сдвиге влево осуществляется знаковое расширение.

2.3.7. Операторы конкатенации

Оператор конкатенации {} группирует два и более операндов и возвращает их как вектор.

Пример:

```
wire [7:0] lobits, hibits;
wire [9:0] signextop,unsignextop;
wire [15:0] wordbits;

//реализуем группировку двух байт в 16 разрядов
assign wordbits={hibits, lobits};
//беззнаковое расширение разрядности до 10 бит
assign unsignextop={2'b0, lobits};
//знаковое расширение до 10 бит
assign signextop={lobits[7], lobits[7], lobits};
```

Возможно использование группы, образованной конкатенацией в качестве приемника при присваивании

```
assign {hibits, lobits} = a+b;
```

2.3.8. Операторы повторения

Оператор повторения {n{a}} формирует n копий операнда a, причем n – константа, не содержащая неопределенных значений.

Пример:

```
input a,b;
wire [7:0] y;
assign y={{5{a}},{3{b}}}; //в старшие 5 бит записываем a, в 3 младших - b
```

и для примера знакового расширения из предыдущего раздела:

```
//знаковое расширение разрядности до 10 бит
assign signextop={2{lobits[7]}, lobits};
```

2.3.9. Тернарный условный оператор

Тернарный условный оператор записывается как a?b:c, где a – условие, возвращающее однобитный результат, b – результат оператора, если a истинно, c – результат оператора, если a ложно.

2.3.10.Приоритет операторов

В таблице приведен приоритет операторов, начиная с самого высокого. Операторы в одной строке имеют одинаковый приоритет и оцениваются слева направо.

[]
()
+, -, !,~ (унарные)

**
*, /, %
+,- (бинарные)
<<, >>, <<<, >>>
<, <=, >, >=
==, !=, ===, !==
&, ~&
^, ^~, ~^
|, ~|
&&
||
?: (тернарный оператор)

2.3.11.Присваивания

Присваивания в Verilog делятся на процедурные и непрерывные.

Процедурные присваивания обновляют значения переменных под управлением конструкций управления потоком исполнения (процедурных блоков), в которых они используются.

При создании синтезируемых описаний такой конструкцией чаще всего является блок always @(), где в скобках определяется список чувствительности. Возможно перечисление всех переменных, от которых зависит описываемая схема, через ключевое слово or, но для конструкций, описывающих комбинационную логику, можно использовать оператор always@(*), что означает выполнение содержащихся в блоке операторов при изменении любой переменной, считываемой в блоке. Также может отслеживаться изменение уровня (фронт). Для этого используются ключевые слова posedge и negedge. В синтезируемых описаниях нельзя совмещать в одном списке чувствительности контроль по уровню и по фронту.

В процедурных присваиваниях приемником могут быть только переменные типа регистр (reg), а также родственные reg переменные типа integer, real и используемые при моделировании realtime и real.

Процедурные присваивания делятся на блокирующие и неблокирующие.

Выполнение блокирующего процедурного присваивания завершается до выполнения следующих за ним присваиваний. Блокирующее присваивание схоже с обычным описанием на языках программирования высокого уровня, где уравнения вычисляются последовательно. Для задания блокирующего присваивания используется символ =.

Пример:

```
wire [7:0] b;
reg [7:0] a,c,d;

always @(*)
begin
      c=b+d;
      c=a+c;
end
```

В данном примере c будет вычислено как сумма a, b и d (рис. 7).

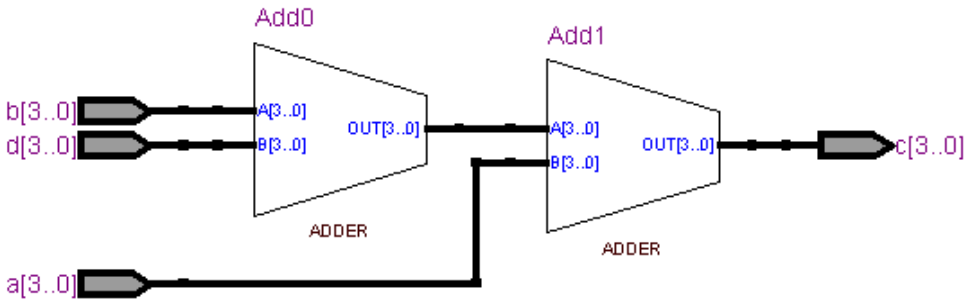

Рис. 7. Реализация блокирующих присваиваний

Неблокирующее процедурное присваивание выполняется, не блокируя другие присваивания (поток исполнения процедуры). Это позволяет более точно описать аппаратную реализацию, в которой все элементы (регистры, логика и все построенные на них модули более высокого уровня) работают параллельно. Неблокирующие присваивания в пределах одного процедурного блока выполняются в два этапа – сначала считываются текущие значения источников, и рассчитываются значения, которые должны быть присвоены приемникам. На втором этапе рассчитанные значения одновременно записываются в переменные-приемники. Для задания неблокирующего присваивания используется символ <=. В пределах одного процедурного блока не рекомендуется смешивать блокирующие и неблокирующие присваивания.

В случае, когда в один момент времени переменной присваивается значение более чем одним неблокирующим присваиванием, учитывается последнее из присваиваний.

Пример:

```
wire [7:0] b;
reg [7:0] a,c,d;

always @(*)
begin
      c<=b+d;
      c<=c+a;
end
```

В данном примере будет реализовано второе присваивание, что приведет к формированию комбинационной обратной связи. Результат синтеза такого описания приведен на рис. 8. Скорее всего, такая реализация не будет соответствовать ожиданиям разработчика.

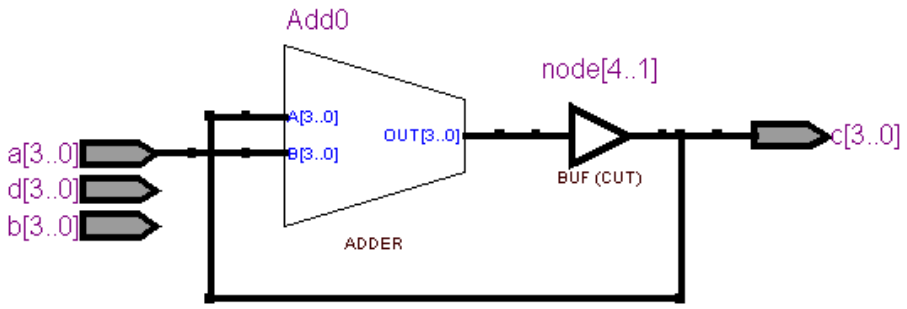

Рис. 8. Пример реализации некорректных неблокирующих присваиваний

Как правило, неблокирующие присваивания используются при описании регистровых схем. Особенности применения блокирующих и неблокирующих присваиваний при описании комбинационных и регистровых схем рассмотрены в 3.1.

Непрерывные присваивания осуществляются вне процедурного блока с использованием ключевого слова assign. Присваивание действует постоянно во время работы модуля. Приемником в непрерывных присваиваниях должна быть переменная типа проводник (wire) или родственного типа (например, output – по умолчанию имеет тип wire).

```
assign a=b+c;
```

Источниками как в процедурных, так и в непрерывных присваиваниях могут быть проводники, регистры и константы.

При использовании данных со знаком лучше использовать знаковые типы, так как в этом случае осществляется корректное расширение разрядности операндов. Следует избегать использования в одном выражении знаковых и беззнаковых операндов - это потенциальный источник ошибок. Если хотя бы один операнд является беззнаковым, операции над операндами в выражении осуществляются как над числами без знака!

2.4. Модули

2.4.1. Объявление модуля

Модули являются основными составляющими любого проекта на Verilog. Модуль представляет собой описание цифрового блока с определенным набором входов и выходов. На нижнем уровне модуль может описывать функционирование базовых элементов, таких, как вентили, на верхнем – систему в целом. В одном файле может быть объявлено несколько модулей, но обязательно должен присутствовать один модуль верхнего уровня с именем, совпадающим с именем файла.

Текущая спецификация языка поддерживает несколько вариантов объявления модуля для обеспечения совместимости с предыдущими версиями языка. Рассмотрим объявление модуля в соответствии со стандартом Veriog-1995, так как многие разработчики продолжают придерживаться старого стиля объявления модуля:

```
module test(a,b,c);
      input a,b;
      output [7:0] c;
      …     //реализация модуля
endmodule
```

В первой строке объявляется модуль с именем test и портами a,b и c. В следующих строках объявляется тип и разрядность портов модуля. Далее следует реализация. Объявление модуля завершается ключевым словом endmodule.

При создании параметризированного модуля объявляются параметры

```
module partest(a,b,c);
      parameter WIDTH=8;
      input a,b;
      output [WIDTH-1:0] c;
      …     //реализация модуля
endmodule
```

Версия Verilog-2001 позволяет объявлять тип и разрядность порта непосредственно в списке портов модуля, а также выделять параметры в отдельную структуру перед перечислением портов. Для модуля, объявленного выше, в версии Verilog-2001 объявление

будет выглядеть следующим образом:

```
module partest
#(parameter WIDTH=8)
(input a, input b, output [WIDTH-1:0] c);
      …     //реализация модуля
endmodule
```

2.4.2. Создание экземпляра модуля

Экземпляры модуля создаются как переменные типа модуля, при этом объявление допустимо в любом месте файла, но не в процедурных блоках. Для объявленного выше модуля partest могут использоваться следующие варианты создания экземпляра:

1) С параметрами по умолчанию (без настройки параметров), связь с портами по перечислению. Связь с портами осуществляется по порядку их перечисления.

```
wire f,g;
wire [7:0] res;
…
partest my_test(f,g,res);
```

2) С настройкой параметров, связь с портами по перечислению.

```
wire f,g;
wire [5:0] res;
…
partest (#6) my_test(f,g,res);
```

3) Связь с портами по имени (независимо от настройки параметров). Такое объявление может быть удобно, если не все порты модуля используются или удобно изменить порядок их перечисления.

```
wire f,g;
wire [5:0] res;
…
```

```
partest (#6) my_test(.a(f),.b(g),.c(res));
```

Если какой-то порт не используется, он не указывается в списке или указывается в списке с пустыми круглыми скобками. Порты, являющиеся выходами используемых модулей, могут связываться только с переменными типа “проводник” (wire, wand, wor и tri). Например, в приведенных выше примерах значение с выходного порта модуля partest присваивается переменной res типа wire.

2.5. Поведенческое описание

2.5.1. Блоки begin … end

Блоки begin … end используются для формирования блока операторов там, где допускается только один оператор по требованиям синтаксиса языка.

2.5.2. Циклы

В Verilog поддерживаются циклы for, while, forever и repeat. Оператор forever будет рассмотрен в разделе, посвященном моделированию.

Синтаксис:

```
for (<счетчик>=<нач. значение>; <условие продолжения>; <счетчик>=<выражение>)
begin
      <операторы>
end

while (<выражение>)
begin
      <операторы>
end

repeat (<количество итераций>)
begin
      <операторы>
end
```

Циклы for используются для повторного исполнения операторов или блока операторов. Как правило, счетчиком цикла является переменная типа integer. Циклы используются при моделировании и описании комбинационной логики (см. в 2.2.7 пример модуля, рассчитывающего количество единиц в двоичном слове).

Пример:

```
wire [7:0] xb;
reg [7:0] xg;
integer i;
always @(*) //расчет кода Грея для двоичного числа xb
begin
      xg[0]=xb[0];
      for (i=1; i<=7; i=i+1) xg[i]=xb[i]^xb[i-1];
end
```

Цикл while выполняется, пока выражение возвращает значение “истина”. Цикл repeat выполняется заданное число раз.

2.5.3. Условный оператор if…else

Условный оператор исполняет набор операторов или блок операторов в зависимости от результата оценки условия после if. В случае, если условие истинно, выполняются выражения, следующие после if, в противном случае выполняются выражения, следующие после else. Если после ключевых слов if и else следует более одного оператора, они объединяются структурой begin…end.

Поддерживаются вложенные формы if… else if … и сокращенная форма без else.

Синтаксис:

```
if (<выражение >)
begin
      <операторы>
end
```

```
if (<выражение >)
begin
      <операторы>
end
else
begin
      <операторы>
end

if (<выражение >)
begin
      <операторы>
end
else if (<выражение >)
begin
      <операторы>
end
…
else
begin
      <операторы>
end
```

Примеры:

```
reg [2:0] result;

always @(*) //if … else реализует мультиплексор 2 в 1
begin
      if (sel) result=a;
      else result=b;
end

always @(*) //if … else if реализует приоритетный шифратор
begin
      if (high_prio) result=3'b100;
      else if (mid_prio) result=3'b010;
      else if (low_prio) result=3'b001;
      else result=0;
```

```
end

always @(ena)     //if … без else, реализует защелку!!!
begin
      if (ena) result=a+b;
end

always @(posedge clk)   //if … без else, реализует регистр
begin
      if (ena) result<=3'b100;
end
```

Два последних примера реализуют схемы, приводящие к синтезу регистров.

В первом случае условием срабатывания блока always является изменение сигнала ena, но не изменение сигналов a и b. В результате синтезируется защелка – регистр, синхронизируемый по уровню.

Во втором примере условием срабатывания блока always является фронт сигнала clk, что соответствует синхронизации по фронту. Синтезируется регистр, синхронизируемый передним фронтом тактового импульса.

2.5.4. Оператор выбора case

Оператор выбора case позволяет осуществить множественный выбор из нескольких вариантов исходя из значения ключа. Если после значения ключа следует более одного оператора, они объединяются структурой begin…end.

Синтаксис:

```
case (<выражение (ключ)>)
<выбор1 (значение ключа)>:
      begin
            <операторы>
      end
<выбор2 (значение ключа)>:
```

```
        begin
                <операторы>
        end
        …
default:
        begin
                <операторы>
        end
endcase
```

Пример:

```
case (op_code)
3'b000: alu_out=a+b;
3'b001: alu_out=a-b;
3'b010: alu_out=-a;
default: alu_out=8'bx;
endcase
```

В данном примере для некоторого кода операции реализуются функции простейшего АЛУ, причем для кодов, не реализуемых в АЛУ (возможно, формирование результата при этом осуществляют другие модули), синтезатор будет формировать произвольное значение.

Значение ключа может быть константой или выражением. Также в одном варианте выбора может указываться несколько значений ключа, разделенных запятыми.

Хорошим тоном при проектировании синтезируемых комбинационных конструкций является обязательное использование варианта “default”, после которого перечисляются операторы, реализуемые в том случае, если значение ключа не совпадает ни с одним из перечисленных. Это позволяет избежать ситуации, когда значение узлу при выполнении блока не присваивается ни одним уравнением. В этом случае синтезируются защелки аналогично ситуации, рассмотренной для условного оператора.

В случае, если значение ключа включает символы x или z(?), выбор будет осуществлен, если в ключе в соответствующих позициях также имеются x и z(?).

Имеются варианты оператора casex и casez. При использовании оператора casex символы x и z(?) в определении значений ключа трактуются как битовые маски, для которых допустимо любое значение соответствующего бита ключа. Это может использоваться для сокращения объема записи.

Пример:

```
always @(*)
begin
      sel_ram=1'b0;
      sel_reg=1'b0;
      sel_io=1'b0;
      casex (address)
      8'b1xxxxxxx: sel_ram=1'b1;
      8'b00000xxx: sel_reg=1'b1;
      8'b01xxxxxx: sel_io=1'b1;
      endcase
end
```

В данном примере реализован простейший дешифратор адреса для некоторой карты памяти, в которой в разных областях расположено ОЗУ, регистры и порты ввода-вывода.

Следует отметить, что в данной записи мы отошли с целью сокращения объема от данной выше рекомендации использовать вариант “default”, однако все определяемые в данном операторе выбора переменные были инициализированы перед оператором case. Так как в данном случае используется блокирующее присваивание, по завершении выполнения case все переменные будут однозначно и правильно определены.

В операторе casez в качестве масок может использоваться только значение z(?), значение x не является маской, а рассматривается как точное значение наряду с 0 и 1.

3. Применение языка Verilog

3.1. Регистры, проводники и описание схем

Тип reg не обязательно синтезируется как аппаратный регистр! Непонимание этого аспекта зачастую приводит к непониманию того, как описывать регистровые и комбинационные схемы.

Сначала рассмотрим применение типов регистра и проводника для описания комбинационных схем. Допустим, мы имеем два узла типа проводника и требуется сформировать узел, равный конъюнкции a и b. Такую логику можно описать как процедурным, так и непрерывным присваиванием:

```
wire a,b;
wire wres;
reg rres;

assign wres=a&b;
always @(*)
begin
      rres=a&b;
end
```

Узел wres типа проводник формируется непрерывным присваиванием assign (в том смысле, что это присваивание непрерывно действует в процессе работы данного модуля).

Узел rres имеет регистровый тип, и смысл описания always@(*) в том, что при любом изменении переменных, входящих в правую часть операторов присваивания внутри блока always, осуществляется обновление значения узла rres, после чего он хранит это значение. Присваивание внутри блока always называется процедурным присваиванием (в том смысле, что узлу осуществляется присваивание во время выполнения процедуры).

Можно заметить, что оба описания синтезируют комбинационную схему, так как при изменении любой входной переменной значение выхода меняется в соответствии с

записанным уравнением.

Усложним схему, добавив третий узел с и реализовав мультиплексор:

```
wire a,b,sel;
wire wres;
reg rres;

assign wres=sel&a|~sel&b;
always @(*)
begin
      if (sel) rres=a;
      else rres=b;
end
```

В случае непрерывного присваивания не могут быть использованы высокоуровневые операторы, такие, как условный оператор, что требует описания схемы на уровне логических и арифметических операторов. Описание внутри процедурного блока может быть более высокоуровневым и абстрактным. Поэтому предпочтительным стилем описания схемы является описание с применением процедурных блоков.

Как правило, при описании комбинационных схем с применением блока always @(*) используются блокирующие присваивания. Это позволяет описать алгоритм вычислений подобно обычному языку программирования.

Однако для описания комбинационной схемы внутри процедурного блока следует придерживаться следующего правила – при выполнении блока always @(*), значение каждому вычисляемому узлу должно быть присвоено хотя бы один раз. Если значение узлу не будет присвоено, то узел должен быть равен предыдущему значению. В результате будет синтезирована схема с регистрами, синхронизируемыми по уровню – защелками. Синтезатор при этом может выдать предупреждение, так как в подавляющем большинстве случаев, особенно при проектировании на ПЛИС, разработчик описывает регистровую логику с применением специального синтаксиса (см. ниже) для задействования регистров, синхронизируемых по фронту. Рассмотрим пример:

```
wire a,ena;
reg rres;

always @(*)
begin
      if (ena) rres=a;
end
```

В случае, если ena изменилось и стало равно 1, rres присваивается значение a, во всех остальных случаях, даже если изменилось a, но ena равно 0, rres сохраняет предыдущее значение. Для реализации такой логики работы требуется защелка.

Так как в современных ПЛИС нет аппаратно реализованных защелок, применение такой конструкции повлечет реализацию защелки на комбинационной логике с обратными связями. Такая схема работает медленнее аппаратно реализованного регистра и имеет нестабильные временные характеристики, которые, к тому же, могут изменяться от компиляции к компиляции при изменениях взаимного расположения элементов.

Применение комбинационной логики с обратными связями при проектировании для ПЛИС является дурным тоном и свидетельствует либо о высшем пилотаже (которым не стоит заниматься без необходимости), либо о непонимании разработчиком ограничений и особенностей используемой аппаратной платформы.

В ПЛИС в качестве ресурсов для хранения значений используются регистры, синхронизируемые фронтом тактового импульса. Как правило, такой регистр также имеет управляющие сигналы асинхронного сброса и разрешения тактового импульса. Такие регистры описываются следующим образом:

```
wire a;
reg rres;
always @(posedge clk or negedge rst)
begin
      if (!rst) rres<=0;
```

```
        else if (ena) rres<=a;
    end
```

Обратите внимание на оператор неблокирующего присваивания (<=), используемый в данном примере.

При описании схем с применением триггеров, синхронизируемых по фронту, обычно используются неблокирующие присваивания в блоке always @(posedge clk). Однако приведенные в предыдущих разделах примеры показывают, что для описания комбинационной части схем удобно использовать блокирующие присваивания.

Поэтому при описании схем, в которых сложная комбинационная логика определяет значения на входах регистров, описание комбинационной логики и логики работы регистров удобно разделять по разным блокам always. Это позволяет избежать формирования промежуточных комбинационных значений с применением блокирующих присваиваний в блоке always @(posedge clk), что резко ухудшает читаемость проекта. Причиной этого является, во-первых, сложность различения переменных типа reg, которые будут реализованы как аппаратные регистры и промежуточных комбинационных узлов. Во-вторых, в комбинации блокирующих и неблокирующих присваиваний становится сложным отслеживание последовательности смены значений узлов

В качестве примера такого разделения модифицируем приведенный в 2.2.7 модуль расчета числа единиц в числе, добавив регистр на выходе:

```
module countones(clk,din,ones);
        input clk;
        input [31:0] din;
        output reg [5:0] ones;

        reg [5:0] comb_ones;
        integer i;

        always @(*)
```

```
	begin
		comb_ones=0;
		for (i = 0; i<32; i=i+1)
		begin
			if (din[i]) comb_ones=comb_ones+1;
		end
	end

	always @(posedge clk) ones<=comb_ones;

endmodule
```

Повторим сформулированные в этом разделе рекомендации:

- **При выполнении блока always @(*), реализующего комбинационную схему, значение каждому вычисляемому узлу должно быть присвоено блокирующими присваиваниями хотя бы один раз в любом состоянии схемы.**
- **При выполнении блока always @(posedge clk), реализующего регистровую схему, значение каждому вычисляемому узлу может быть присвоено неблокирующими присваиваниями не более одного раза в любом состоянии схемы.**

3.2. Примеры модулей

3.2.1. Дешифратор

```
//дешифратор 3->8 на основе оператора выбора
//объявление модуля Verilog-1995
module decode_case(din,dout);
	input [2:0] din;
	output reg [7:0] dout;

	always @(*)
	begin
		case (din)
		3'b000: dout=8'h01;
		3'b001: dout=8'h02;
```

```
            3'b010: dout=8'h04;
            3'b011: dout=8'h08;
            3'b100: dout=8'h10;
            3'b101: dout=8'h20;
            3'b110: dout=8'h40;
            3'b111: dout=8'h80;
            endcase
      end

endmodule

//параметризируемый дешифратор на основе операции сдвига
//объявление модуля Verilog-2001
module decode_shift
#(parameter WIDTH = 3)
(input [WIDTH-1:0] din, output reg [2**WIDTH-1:0] dout);

      always @(*) dout=1<<din;

endmodule
```

3.2.2. Синхронный счетчик до 5 со сбросом и разрешением

```
module bin_cnt
(
      input clk, enable, reset,
      output reg [3:0] count
);

      //срабатывает по фронту тактового импульса и асинхронного сброса
      always @ (posedge clk or negedge reset)
      begin
            if (!reset)
                  count <= 0;
            else if ((enable == 1'b1)&&(count!=5))
                  count <= count + 1;
      end

endmodule
```

Данный модуль реализует синхронный счетчик, который считает при поступлении положительных фронтов тактового импульса clk вверх до 5. По достижении значения 5 вне зависимости от тактового сигнала счетчик удерживает это значение до сброса. Асинхронный сброс осуществляется сигналом reset с активным низким уровнем.

Также в модуле используется вход разрешения тактовых импульсов enable, который задействует соответствующий порт триггеров при наличии этой возможности в аппаратуре. Если эта возможность не поддерживается аппаратно, сигнал разрешения формируется на комбинационной логике. Обратите внимание, что в списке чувствительности нет enable, так как триггер срабатывает только по тактовому импульсу или асинхронному управляющему входу (в данном случае сброса).

4. Моделирование

Для создания тестовых векторов и абстрактного описания схем используется более широкое подмножество языка, содержащее так называемые несинтезируемые конструкции. Оно включает в себя определение задержек, бесконечных циклов и т.д.. В предыдущем разделе были рассмотрены основы создания синтезируемых описаний модулей. В этом разделе будет рассмотрено, как описать поведение системы, в которой используется наш модуль и, таким образом, обеспечить возможность моделирования работы разработанного модуля.

4.1. Задание временных зависимостей

4.1.1. Задание задержек

Оператор задания задержки определяет задержку во времени выполнения следующего за оператора относительно предыдущего оператора. Задержка измеряется в шагах временной сетки, задаваемой директивой ‘timescale

Синтаксис:

```
#<задержка> <оператор>
```

Пример:

```
#5 a=a+b; //через пять шагов прибавить к a b
#4 a=a+c; //еще через четыре шага прибавить к a c
```

4.1.2. Задание событий

Оператор задания события нами уже рассматривался в описании списка чувствительности блока always. Рассмотренные ранее * и posedge clk – это события. При моделировании события могут определять условия для срабатывания процедурных блоков и однократного срабатывания отдельных операторов.

Синтаксис:

```
@<событие>
@(<событие or событие or …>)
```

Событие – это имя переменной, оператор детектирования фронта, имя события или *.

Пример:

```
@a b=a+c; //дождаться изменения a и при изменении a выполнить оператор
always @(a) b=a+c; //всегда при изменении a выполнять оператор
```

Для создания события используется следующий синтаксис:

```
event <имя события>;          //создаем событие
…
-><имя события>;              //активизируем событие
…
@(<имя события>) <операторы>  //контроль активизации события
```

Именованные события могут использоваться для взаимодействия параллельно функционирующих процедурных блоков. В одном блоке событие активизируется, в другом при его активизации запускаются операторы.

4.1.3. Использование задержек и событий внутри присваиваний

Рассмотренные задержки и события запускают выполнение операторов. Однако, они также могут использоваться внутри присваиваний для задержки присваивания. При этом считывание переменных осуществляется без задержки или контроля события. Рассмотрим их применение на примере задания задержек

Пример:

```
a = #5 a+b; //считать и сложить значения a и b, через пять шагов присвоить a
```

4.1.4. Оператор ожидания

Оператор wait переводит симулятор в состояние ожидания до того момента, когда оцениваемое выражение вернет результат “истина”.

Пример:

```
wait (!c) a = b;  //дождаться, когда c станет равным 0 и присвоить a значение b
```

4.2. Процедурные структуры

4.2.1. Блок always

Блок always постоянно выполняется во время моделирования. Все блоки always в модуле выполняются одновременно и параллельно. Операторы внутри блока выполняются последовательно для блокирующих присваиваний и параллельно для неблокирующих.

Так как блок always выполняется постоянно, он обычно используется с одним из рассмотренных способов задания временных зависимостей для определения момента следующего выполнения блока.

Пример:

```
always #5 clk=~clk;     //создаем тактовый импульс с периодом 10 шагов
```

4.2.2. Блок initial

Блок initial схож с блоком always, но выполняется только один раз в начале моделирования. Используется для начальной инициализации сигналов и задания последовательностей операторов, которые надо выполнить только один раз, например, покрывающих все время моделирования.

Пример:

```
initial clk=0;          //тактовый импульс начинается с нуля
always #5 clk=~clk;     //создаем тактовый импульс с периодом 10 шагов
```

Рассмотрим более сложный пример, в котором блок initial содержит полную развертку времени моделирования и инициирует изменения переменных, которые в приводят к выполнению блока always. В конце блока initial вызывается системная задача $stop, которая останавливает моделирование.

Пример:

```
initial
begin                    //момент времени (шаги)
      a=0;               //0
      #50 a=1;           //50
      #52 a=5;           //102
      #80 a=1;           //182
      #100 $stop         //282
end

always @(a)
begin
      d=e$a;
end
```

4.2.3. Поведенческое описание

Рассмотренные в разделе, посвященном синтезу операторы выбора, условные операторы, циклы и т.д. могут применяться и для создания несинтезируемых описаний. Рассмотрим в дополнение еще один оператор поведенческого описания, который может применяться только в несинтезируемых описаниях.

Цикл forever является несинтезируемой процедурной конструкцией, которая повторяется бесконечно (но внутри процедурного блока!). Приведем еще один пример создания тактового импульса на основе оператора forever и блока initial.

Пример:

```
initial
begin
```

```
        clk=0;
        forever #5 clk=~clk;
    end
```

4.3. Системные процессы и функции

Системные процессы и функции используются для управления процессом моделирования – остановки моделирования, ввода и вывода данных на экран, работы с файлами, формирования данных, в том числе моделирования случайных процессов и т.д. Имена системных процессов и функций начинаются со знака $. Рассмотрим некоторые из системных функций, входящих в стандарт языка.

4.3.1. Ввод/вывод на экран

Задачи $display, $write, $strobe и $monitor выводят данные на экран во время моделирования и используют одинаковый синтаксис. Задачи $display и $write выводят строку в момент их выполнения. Разница между ними заключается в том, что $display осуществляет перевод строки, а $write не осуществляет. Задача $strobe выполняет стробирование – значения данных оцениваются и выводятся на экран в конце текущего шага моделирования перед переходом к следующему шагу по временной сетке, то есть, в отличие от $display и $write все изменения сигналов, которые должны произойти на данном шаге, будут произведены до вывода. Задача $monitor выводит на экран строку при изменении любого параметра, который перечислен в списке параметров задачи (за исключением задач получения времени).

Перечисленные задачи по умолчанию выводят данные в десятичном виде. Есть версии всех задач с выводом по умолчанию данных в двоичном (на примере $display: $displayb), восьмеричном ($displayo) и шестнадцатеричном ($displayh) виде. При этом для любой функции может использоваться форматирование аналогично задание формата в функции fprint() в языке C.

Синтаксис:

```
$display("<строка форматирования>",<par1>,<par2>,…);
```

Пример:

```
initial
begin
      //вывод h в начале в шестнадцатеричном представлении
      $displayh("in the beginning h equals %h",h);
      //запуск мониторинга
      $monitor("h changed at %t, equals %h",$time, h);
      //остановка моделирования через 1000 шагов
      #1000 $stop;
end
```

В один момент времени может быть активна только одна задача мониторинга. Повторный вызов задачи мониторинга отменяет предыдущий. Имеются также две задачи $monitoron и $monitoroff, которые соответственно включают и выключают мониторинг в процессе моделирования.

4.3.2. Считывание времени

Задачи $time, $stime, $realtime считывают время в шагах сетки как 64-битное значение, 32-битное значение и число с плавающей запятой соответственно.

4.3.3. Управление процессом моделирования

Задача $stop останавливает моделирование.
Задача $finish останавливает моделирование и выходит из программы моделирования.

4.4. Директивы компилятора

Директивы компилятора начинаются со знака ` (обратный апостроф). Директивы обеспечивают возможность определения констант, условной компиляции, включения файлов, задания прагм, а также некоторые специфические функции, например, задание шага временной сетки или подтяжки к заданному уровню неприсоединенных портов

модуля.

4.4.1. Задание шага временной сетки

Директива `timescale задает единицу измерения и разрешение. Единица измерения (шаг сетки) определяет значения задержек, разрешение – точность представления результатов при моделировании. поддерживаются единицы измерения s,ms,us,ns,ps,fs – от секунд до фемтосекунд. Директива действует только в файле, где она определена. На другие файлы в проекте, в том числе на те, в которых определены вызываемые из данного файла модули, она не распространяется!

Синтаксис:

```
`timescale <единица измерения>/<точность>
```

Пример:

```
`timescale 1ns/100ps
…
#5 a=b;     //с задержкой в 5нс присвоить a значение b
```

4.4.2. Задание и использование макроса

Директива `define создает макрос, который в дальнейшем может использоваться для замены текста при компиляции. Определение может быть отменено директивой `undef. заменяет текст. timescale задает единицу измерения и разрешение. Директивы `ifdef, `ifndef, `else, `elsif и`endif используются для реализации условной компиляции в зависимости от того, определен или не определен макрос.

Синтаксис:

```
`define <имя макроса> <значение макроса>
`undef <имя макроса>
`ifdef <имя макроса> | `ifndef <имя макроса>
[`elsif <имя макроса>]
`else
```

```
`endif
```

Пример:

```
//определим базовую разрядность
`define WIDTH 8
//нужен вариант с удвоенной разрядностью
`define DOUBLE_WIDTH

//в зависимости от наличия макроса DOUBLE_WIDTH генерируем код
`ifdef DOUBLE_WIDTH
      wire [WIDTH*2-1:0] a;
`else
      wire [WIDTH-1:0] a;
`endif
```

4.4.3. Включение файлов

Включение файлов осуществляется директивой `include

Пример:

```
`include "libmodule.v"  //включаем содержимое файла libmodule.v
```

4.5. Создание тестбенча

Тестбенч (в прямом переводе - "испытательный стенд") – это описание поведения внешнего по отношению к тестируемому модулю мира, использующее тестируемый модуль как модуль нижнего уровня. Тестбенч подается на вход средства моделирования для проверки корректности работы тестируемого модуля. При этом при проектировании цифровых схем описание тестируемого модуля, как правило, синтезируемо, а описание поведения внешнего мира несинтезируемо.

Модуль, описывающий поведение внешнего по отношению к тестируемому модулю мира, выполняет две основные функции:
формирование входных сигналов для тестируемого модуля

контроль корректности работы

Формирование входных сигналов осуществляется с применением возможностей поведенческого описания и позволяет, например, изменять состояние внешнего мира в зависимости от выходов тестируемого модуля. Это значительно расширяет возможности моделирования по сравнению с моделированием с заданием тестового вектора в виде временной диаграммы.

Встроенные системные задачи предоставляют удобные возможности по контролю корректности работы. Возможно детектирование сбойных ситуаций и вывод сообщений о них, сохранение результатов моделирования в файл и т.д.

Создадим тестбенчи для приведенных в разделе 3.2 примеров модулей дешифратора и счетчика.

4.5.1. Тестбенч дешифратора

Для тестирования дешифратора требуется подать на него все возможные значения и проверить, что выход совпадает с требуемым. В данном тестбенче проверку правильности формирования выхода возложим на пользователя, так как большинство средств моделирования поддерживает отображение результатов моделирования (значений на входных, выходных и внутренних узлах тестируемого модуля) в виде временной диаграммы. В более сложных схемах можно сгенерировать эталонный отклик или считать его из файла и сравнивать с результатами, возвращаемыми модулем.

Пример:

Файл decode.v

```
module decode_shift
#(parameter WIDTH = 3)
```

```
  (input [WIDTH-1:0] din, output reg [2**WIDTH-1:0] dout);

        always @(*) dout=1<<din;

  endmodule
```

Файл decode_test.v

```
//установим временную сетку
‘timescale 1ns/1ps

/*
Тестбенчу не требуются входы, он сам формирует все сигналы
и передает их на вход тестируемого модуля.
Для тестирования комбинационной функции достаточно перебрать все
возможные комбинации значений на входе
*/
module decode_test;

localparam WIDTH = 4;

reg [WIDTH-1:0] enc_data;
wire [2**WIDTH-1:0] dec_data;

initial
begin
  //начинаем перебор значений с нуля
  enc_data=0;
  repeat (2**WIDTH)
  begin
        //выводим информацию о текущем состоянии
        //если бы мы использовали здесь $display,
        // значение enc_data могло бы не обновиться
        $strobe(“Input value: %d; output value: %b; time: %d”,
                    enc_data,dec_data,$time);
        //каждые 10нс инкрементируем значение на входе дешифратора
        #10 enc_data=enc_data+1;
  end
  //завершаем моделирование
```

```
    $finish
  end

  //создаем экземпляр функции, переопределяя параметр по умолчанию
  decode_shift #(WIDTH) test_dev(enc_data,dec_data);

  endmodule
```

Результат работы средства моделирования:

```
# KERNEL: Input value:  0; output value: 0000000000000001; time:                    0
# KERNEL: Input value:  1; output value: 0000000000000010; time:                   10
# KERNEL: Input value:  2; output value: 0000000000000100; time:                   20
# KERNEL: Input value:  3; output value: 0000000000001000; time:                   30
# KERNEL: Input value:  4; output value: 0000000000010000; time:                   40
# KERNEL: Input value:  5; output value: 0000000000100000; time:                   50
# KERNEL: Input value:  6; output value: 0000000001000000; time:                   60
# KERNEL: Input value:  7; output value: 0000000010000000; time:                   70
# KERNEL: Input value:  8; output value: 0000000100000000; time:                   80
# KERNEL: Input value:  9; output value: 0000001000000000; time:                   90
# KERNEL: Input value: 10; output value: 0000010000000000; time:                  100
# KERNEL: Input value: 11; output value: 0000100000000000; time:                  110
# KERNEL: Input value: 12; output value: 0001000000000000; time:                  120
# KERNEL: Input value: 13; output value: 0010000000000000; time:                  130
# KERNEL: Input value: 14; output value: 0100000000000000; time:                  140
# KERNEL: Input value: 15; output value: 1000000000000000; time:                  150
# RUNTIME: RUNTIME_0068 decode_test.v (31): $finish called.
# KERNEL: Time: 160 ns,  Iteration: 0,  TOP instance,  Process: #INITIAL#18_1.
# KERNEL: stopped at time: 160 ns
```

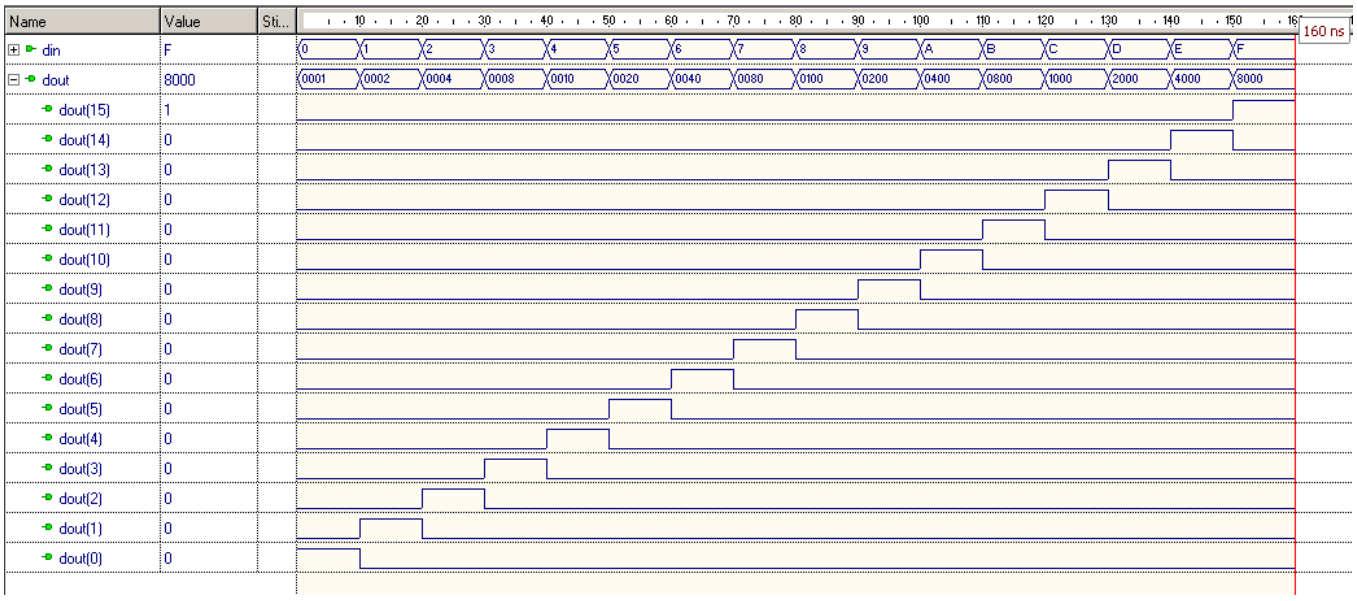

Рис. 9. Временная диаграмма работы дешифратора

4.5.2. Тестбенч счетчика

Для тестирования счетчика используем следующую методику:

- осуществим начальный сброс;
- разрешая тактовый импульс не на каждом такте, дадим достаточно тактов, чтобы досчитать до 5;
- убедимся, что при дальнейшем разрешении тактовых импульсов значение на выходе не изменяется
- осуществим сброс счетчика и убедимся, что счет опять начнется с 0

Пример:

Файл counter.v:

```
module bin_cnt
(
	input clk, enable, reset,
	output reg [3:0] count
);
```

```
	//срабатывает по фронту тактового импульса и асинхронного сброса
	always @ (posedge clk or negedge reset)
	begin
		if (!reset)
			count <= 0;
		else if ((enable == 1'b1)&&(count!=5))
			count <= count + 1;
	end

endmodule
```

Файл counter_test.v:

```
//установим временную сетку
`timescale 1ns/1ps

/*
Тестбенчу не требуются входы, он сам формирует все сигналы
и передает их на вход тестируемого модуля.
Для тестирования нашего счетчика предложим следующую методику:
- осуществим начальный сброс;
- разрешая тактовый импульс не на каждом такте, дадим достаточно тактов,
  чтобы досчитать до 5;
- убедимся, что при дальнейшем разрешении тактовых импульсов значение
  на выходе не изменяется
- осуществим сброс счетчика и убедимся, что счет опять начнется с 0
*/
module counter_test;

integer i;

//так как тактовый импульс, сброс и разрешение формируются
//в процедурных блоках, объявляем их как переменные типа регистр
reg clk,ena,rst;
//выход модуля, вне зависимости от внутреннего типа, может быть
//присвоен только  переменной типа wire
wire [3:0] result;

//формируем тактовый импульс с периодом 20нс в отдельном блоке initial
```

```
initial begin

	//задаем начальное значение clk
	clk=0;
	//до окончания моделирования каждые 10нс переключаем clk
	forever #10 clk = ~clk;
end

//формируем сигнал разрешения через такт в отдельном блоке initial
initial begin
	//сначала сбрасываем ena в 0
	ena=0;
	//далее по каждому отрицательному фронту clk меняем уровень
	//на противоположный (используется счетчиком на положительном фронте)
	forever @(negedge clk) ena=~ena;
end

//общая карта времени и сброс
initial
begin
	//осуществляем сброс
	rst=0;
	//снимаем сброс
	#10 rst=1;
	//смотрим, как считает счетчик до 5
	while (result<5) #20 $strobe("Counter value: %d", result);
	//ждем достаточно времени для прохождения нескольких разрешенных тактов,
	//чтобы проверить, удерживается ли значение 5 и сбрасываем счетчик
	#100 rst=0;
	//снимаем сброс
	#10 rst=1;
	//счетчик должен немного посчитать с нуля,
	//после чего завершаем моделирование
	#100 $finish;
end

//создаем экземпляр функции
bin_cnt test_dev(clk, ena, rst, result);

endmodule
```

Результат работы средства моделирования:

```
# KERNEL: Counter value:  1
# KERNEL: Counter value:  1
# KERNEL: Counter value:  2
# KERNEL: Counter value:  2
# KERNEL: Counter value:  3
# KERNEL: Counter value:  3
# KERNEL: Counter value:  4
# KERNEL: Counter value:  4
# KERNEL: Counter value:  5
# KERNEL: Counter value:  5
# RUNTIME: RUNTIME_0068 counter_test.v (55): $finish called.
# KERNEL: Time: 420 ns,  Iteration: 0,  TOP instance,  Process: #INITIAL#41_3.
# KERNEL: stopped at time: 420 ns
```

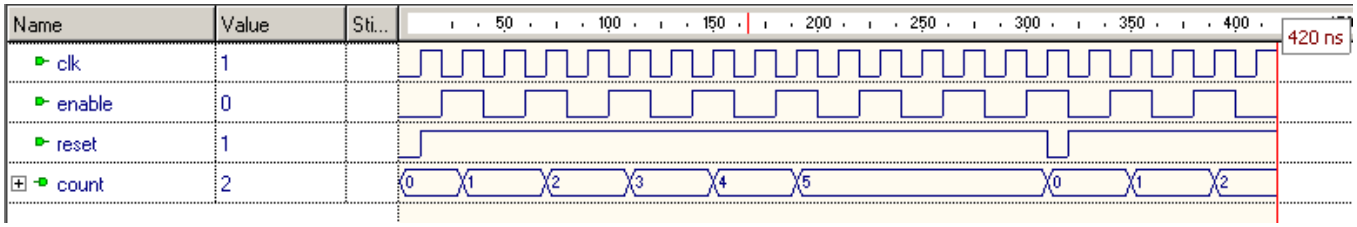

Рис. 10. Временная диаграмма работы счетчика

Приложение 1. Ключевые слова Verilog

```
always
and
assign
automatic
begin
buf
bufif0
bufif1
case
casex
casez
cell
cmos
config
deassign
default
defparam
design
disable
edge
else
end
endcase
endconfig
endfunction
endgenerate
endmodule
endprimitive
endspecify
endtable
endtask
event
for
force
forever
fork
function
generate
genvar
highz0
highz1
if
ifnone
incdir
include
initial
inout
input
instance
integer
join
large
liblist
library
localparam
macromodule
medium
module
nand
negedge
nmos
nor
noshowcancelled
not
notif0
notif1
or
output
parameter
pmos
posedge
primitive
pull0
pull1
pulldown
pullup
pulsestyle_oneve
nt
pulsestyle_ondet
ect
rcmos
real
realtime
reg
release
repeat
rnmos
rpmos
rtran
rtranif0
rtranif1
scalared
showcancelled
signed
small
specify
specparam
strong0
strong1
supply0
supply1
table
task
time
tran
tranif0
tranif1
tri
tri0
tri1
triand
trior
trireg
unsigned
use
vectored
wait
wand
weak0
weak1
while
wire
wor
xnor
xor
```

Контрольные вопросы

1. Опишите основные преимущества использования языков описания аппаратного состава.

Ответ

2. Перечислите уровни представления схемы в процессе разработки.

Ответ

3. Перечислите версии стандартов языка Verilog.

Ответ

4. Как задается разрядность чисел в языке Verilog?

Ответ

5. Как описать шину в Verilog?

Ответ

6. Объясните разницу между операторами == и ===.

Ответ

7. Чему равен результат операции 2’b10&2’bzx?

Ответ

8. Как осуществить знаковое расширение числа на 2 разряда?

Ответ

9. Объясните различие между непрерывными и процедурными присваиваниями.

Ответ

10. Объясните различие между блокирующими и неблокирующимим присваиваниями.

Ответ

11. Перечислите варианты подключения сигналов к портам модуля.

Ответ

12. Объясните, для чего применяется значение x в определении значения ключа в операторе casex.

Ответ

13. Объясните, почему при описании комбинационных схем в блоке always @(*) значение каждому вычисляемому узлу должно быть присвоено хотя бы один раз.

Ответ

14. Что такое тестбенч?

Ответ

15. Как задаются задержки в Verilog?

Ответ

Список литературы

1. В.Б.Стешенко. “ПЛИС фирмы “ALTERA”: элементная база, система проектирования и языки описания аппаратуры” - М.: Издательский дом “Додэка”, 2002. - 576 с.

2. С.Емец. “Verilog - инструмент разработки цифровых электронных схем” - М.: Компоненты и Технологии, №10, 2000 (http://www.compitech.ru/html.cgi/arhiv/01_02/stat_86.htm)