# Оглавление

# Аннотация

В пособии рассматриваются основные возможности языка SystemVerilog.

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 SystemVerilog – это унифицированный язык для описания, спецификации и верификации аппаратуры, значительно расширяющий возможности традиционных языков описания аппаратного состава.

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). В настоящее время описание на поведенческом уровне, использующее такие операторы, как операторы цикла, условные операторы и т.д., также относится к уровню регистровых передач, если может быть автоматически синтезирована реализующая его схема.

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

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 и аудит проекта;
- обеспечивается возможность реализации сложных тестов для верификации проекта путем моделирования.

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 пакете Matlab, экспортировать в SystemVerilog и синтезировать в аппаратную реализацию. К сожалению, при таком подходе зачастую генерируются неэффективные аппаратные реализации, которые требуют в дальнейшем значительной доводки для повышения быстродействия и(или) уменьшения количества используемых вентилей. Правка кода на более низких уровнях зачастую не дает должного эффекта, так как грамотные технические решения должны приниматься уже на уровне формального представления и при проектировании архитектуры, 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.

Язык SystemVerilog обеспечивает возможность описания схемы на уровнях от архитектурного представления до представления на уровне транзисторов. Чаще всего для описания логики в SystemVerilog используются операторы, схожие с операторами обычных языков программирования – арифметические и логические операторы, условный оператор, оператор выбора и т.д. Так описывается поведение схемы и одновременно задается архитектура, как в части комбинационных функций, так и регистровой логики.

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. В SystemVerilog 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.

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

## 1.2. Архитектура ПЛИС

Языки описания аппаратного состава могут использоваться не только для проектирования логики заказных цифровых интегральных схем, но и для проектирования цифровых схем на программируемых логических интегральных схемах (ПЛИС). ПЛИС – это полностью изготовленная специализированная интегральная схема, которая доспециализируется (программируется) разработчиком устройства. Таким образом, обеспечивается возможность получить аппаратную реализацию обработки данных с характеристиками, близкими к заказной интегральной схеме, без необходимости проектирования самой интегральной схемы или реализации схемы на наборе дискретных логических микросхем стандартных семейств.

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 тип триггера 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 в настоящее время применяются в ПЛИС низкой степени интеграции. Матрица И-ИЛИ реализует программируемую ДНФ, которая формируется синтезатором из описания проекта пользователя. (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.

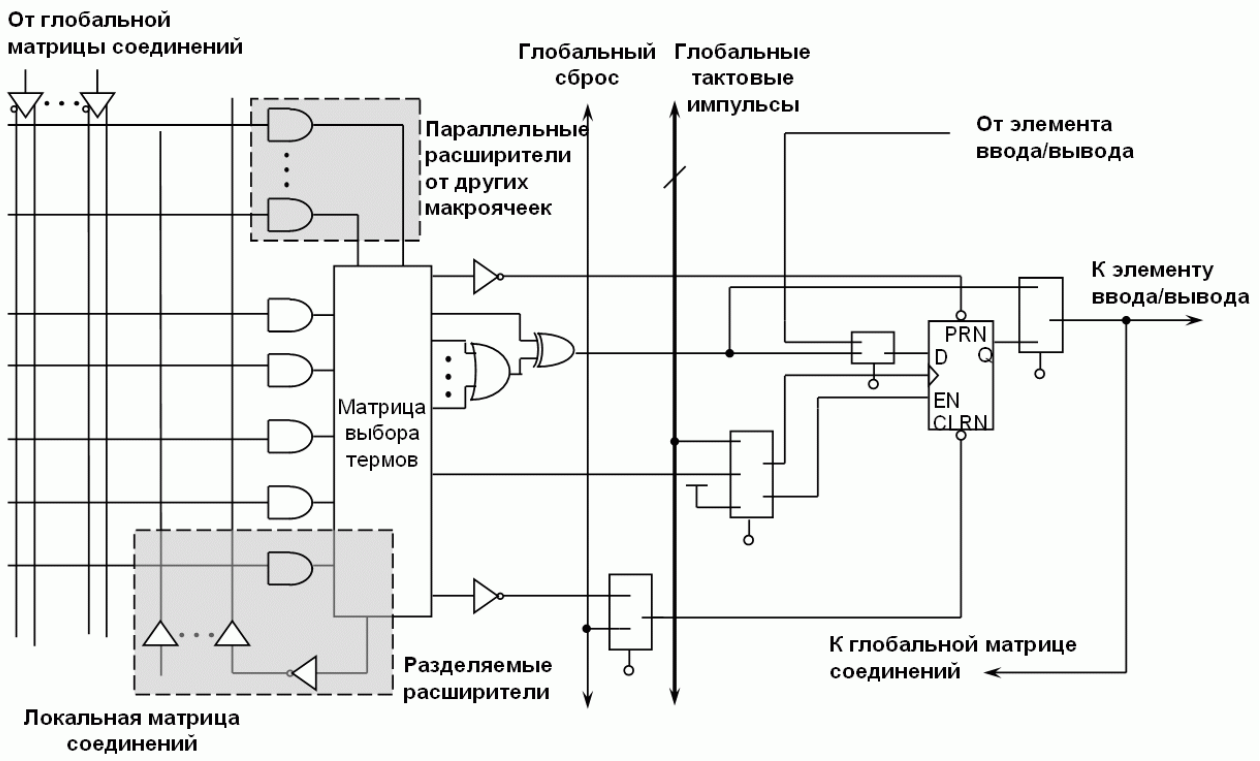


Рис. 4. Архитектура матриц И-ИЛИ. Макроячейка микросхемы семейства MAX7000A

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. Рассмотрим архитектуру на основе четырехвходовых таблиц перекодировки. Такая таблица перекодировки представляет собой 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.  Таблицы перекодировки выполняются на основе технологии статического ОЗУ, что обеспечивает возможность загрузки конфигурации в ПЛИС после включения питания и, в некоторых семействах, даже ее изменение в процессе работы устройства.

Типичным представителем недорогих микросхем программируемой логики средней и высокой степени интеграции с такой архитектурой является семейство Cyclone IV фирмы Altera. Архитектура функционального преобразователя этого семейства приведена на рис. 5.  Количество ФП в устройствах семейства Cyclone IV варьируется от 6 тысяч до 150 тысяч.

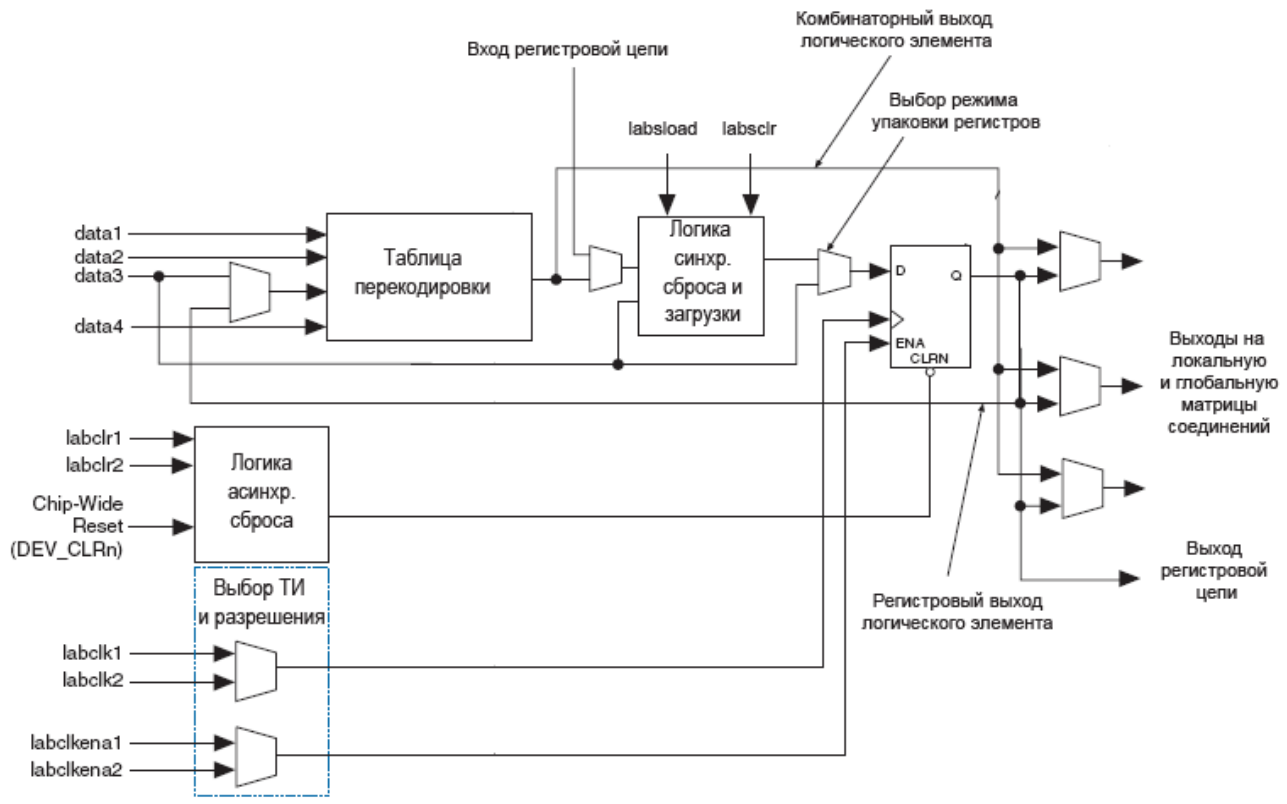


Рис. 5. Архитектура на основе таблиц перекодировки. Логический элемент микросхемы семейства Cyclone IV.

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 режим не показан).

В более современных семействах используются и другие, более сложные архитектуры ФП, однако для них обычно указывается примерная эквивалентная емкость в ФП с рассмотренной архитектурой, которая является де-факто базовой для различных производителей.

В 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);
- контроллеры оперативной памяти (DDR2, DDR3, DDR4 SDRAM)
- 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.

## 1.3. Пример реализации модуля на SystemVerilog

Разберем в качестве примера файл dec.sv. 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.

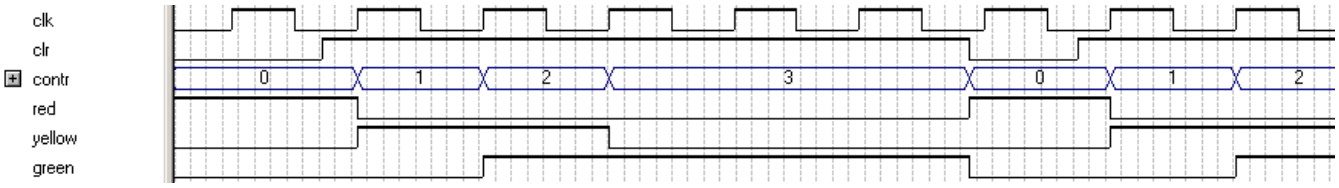


Рис. 6. Диаграмма работы семафора

Модуль реализован как синхронный двоичный счетчик до 3 со сбросом и специальный дешифратор 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(
2     input logic clk, clrn,
3     output logic red, green, yellow
4 );
5     //внутренние переменные
6     logic [1:0] contr;
7
8     //синхронная логика счетчика
9     always_ff @(posedge clk or negedge clrn)
10    begin
11          if (!clrn) contr<=0;
12          else if (contr!=3) contr<=contr+1;
13    end
```

```
14
15    //комбинационная логика декодирования
16    always_comb
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.

Строки 2-3

Список портов модуля. Определяет входы (input) и выходы (output) модуля. Входы и выходы могут быть как одноразрядными так и многоразрядными. Тип входов logic – это базовый тип языка SystemVerilog, используемый для описания цифровых сигналов.

Строка 6

Объявление внутренней переменной contr, в которой будет храниться значение счетчика состояний. Многоразрядные переменные и порты объявляются как векторы. В объявлении вектора индексы в квадратных скобках определяют диапазон индексов проводников, входящих в вектор. Первый индекс соответствует старшему, второй – младшему биту.

Строка 9

Начало процедурного блока always_ff, 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 фронты сигналов clk и clrn. Обратите внимание, что 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. При синтезе для хранения переменных типа logic, значения которых определяются в таком блоке, будут реализованы триггеры с синхронизацией по фронту clk.

Также в списке чувствительности указан сигнал асинхронного сброса clrn, 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 языка SystemVerilog – именно так принято описывать список чувствительности для реализации синхронных триггеров с асинхронными управляющими сигналами.

Строка 11

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

Строка 12

Если сигнал clrn 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 2.4.

Строка 16

Начало процедурного блока always_comb, описывающего комбинационную логику. В отличие от регистровой логики, в комбинационной схеме значение на выходе может измениться при изменении любого входного сигнала, поэтому список чувствительности у такого блока не определяется.

Строки 17-22

Оператор выбора, реализующий функцию дешифратора. Обратите внимание на то, что выходы red, green, yellow имеют тип logic, но, в отличие от переменных, описываемых в блоке always_ff, они не становятся регистрами, а соответствуют выходам комбинационной схемы. 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 2.4.

Строка 24

Модуль должен заканчиваться ключевым словом endmodule.

# 2. Основы языка SystemVerilog

## 2.1. Лексические элементы

Язык SystemVerilog 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_comb
begin
      a=c+d;
end
```

### 2.1.2. Числа

Целочисленные константы в SystemVerilog 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, как в примере выше, то она устанавливается в соответствии с разрядностью выражения, в котором используется константа или разрядностью приемника.

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. При этом синтезаторы обычно не поддерживают автоматический

синтез логики, реализующей операции с вещественными числами. Такая логика реализуется модулями, входящими в библиотеку функций.

Пример:

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

### 2.1.3. 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
```

### 2.1.4. Ключевые слова

Ключевые слова SystemVerilog приведены в приложении B стандарта языка. Заданные пользователем идентификаторы не должны повторять ключевые слова.

## 2.2. Типы данных

### 2.2.1. Набор значений

В SystemVerilog используется 4 значения для определения значений проводников и переменных:

- 0 – логический ноль, ложь
- 1 – логическая единица, истина

- x – значение не известно (любое значение)
- z – третье (высокоимпедансное состояние)

Тип logic определяет данные, которые могут принимать значения из набора {0,1,x,z} и используется задания значений проводников и переменных, описывающих сигналы в цифровой схеме.

Тип bit определяет данные, которые могут принимать значения из набора {0,1} и используется для высокоуровневого моделирования. Основанные на типе bit целочисленные типы могут использоваться для реализации высокоуровневых моделей, счетчиков циклов и т.д.

### 2.2.2. Проводники

Тип wire соответствует физическому проводнику в устройстве и используется для соединения вентилей и модулей, а также для описания простых комбинационных схем. 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, также значение проводника может быть определено портами модулей. Тип wire можно не указывать, можно указать тип данных logic, при этом компилятор исходя их контекста определит, что это проводник.

Пример:

```
wire a,b;
assign a=1'b1; //константа
assign b=a&c; //комбинационная логика
```

### 2.2.3. Переменные

Переменная представляет объект, который может хранить присвоенное ему значение. Для объявления переменной может использоваться ключевое слово var. Присвоение значений переменным осуществляется процедурными присваиваниями в процедурных блоках или одним непрерывным присваиванием assign. Ключевое слово var можно не указывать, а указать только тип данных, например, logic. В этом случае компилятор автоматически определяет переменную. Так как значения в непрерывных присваиваниях можно присваивать как проводникам, так и переменным, в дальнейшем мы будем использовать для описания переменных и в качестве замены проводников тип logic без указания ключевого слова var.

Пример:

```
logic a,b;
assign b = 1'b1;
always_comb b=a&c;
```

### 2.2.4. Входы, выходы и двунаправленные порты

Ключевые слова input, output и inout объявляют соответственно входы, выходы и двунаправленные порты модулей. Порты могут быть объявлены с указанием типа проводника или переменной. Если тип проводника или переменной не указан в объявлении порта, он может быть объявлен в модуле. Также порт может иметь тип интерфейса, в рамках которого могут быть определены его параметры, набор сигналов и методы работы с ними. В данном пособии интерфейсы не рассматриваются.

Пример:

```
module sample(input a, output b, output logic c, inout d);
logic b;
```

### 2.2.5. Массивы

Приведенные выше объявления проводников и регистров создают скалярные

(одноразрядные) переменные. Для объявления шин, многоразрядных регистров и портов, памяти и т.д. используются массивы.

В стандарте SystemVerilog определено два типа массивов – упакованные и неупакованные. У упакованного массива диапазоны индексов указываются после типа данных и до имени массива. Одномерный упакованный массив также называется вектором и обычно используется для описания многоразрядных регистров, шин и портов.

Пример:

```
logic [15:0] a, b;
```

Первый указанный индекс соответствует старшему (наиболее значимому) биту, второй - младшему (наименее значимому). Упакованный массив рассматривается как непрерывный набор бит, поэтому ему можно присваивать значение как многоразрядному регистру или шине.

Пример:

```
assign b = 5;
assign a = b - 2;
```

Обращение ко всему массиву возможно по имени, например

```
a = b;
```

Обращение к элементам вектора возможно как к одиночному проводнику (bit-select), так и к поддиапазону (part-select) с указанием, соответственно, индекса и диапазона индексов

```
dreg[0] = 1'b1;
dreg[3:2] = 2'b10;
```

По умолчанию значение вектора типа logic рассматриваются как целочисленные данные ез знака. Для  указания знакового представления используется ключевое слово signed.

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

Диапазон индексов неупакованного массива указывается после имени. В неупакованном массиве данные рассматриваются не как непрерывный набор бит, а как массив, состоящий из отдельных, не связанных элементов. SystemVerilog поддерживает неупакованные массивы фиксированного размера, динамические массивы, ассоциативные массивы и очереди. Однако, для синтезируемых описаний в настоящее время используются только неупакованные массивы фиксированного размера.

Поддерживаются массивы с несколькими индексами. При обращении к элементам массива сначала разворачиваются упакованные индексы, начиная с правого, а потом неупакованные.

Пример:

```
logic [15:0] data_src[7:0];   //массив из 8 16-ти разрядных шин
logic [15:0] dreg[1023:0];    //массив из 1024 16-ти разрядных регистров
logic [3:0][7:0] mm[0:3][0:9];      //можно и так
assign mm[3][9][2] = 8'd126;
```

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

### 2.2.6. Целочисленные типы данных

В таблице 2.1 приведены целочисленные типы языка SystemVerilog, в основе которых лежит тип logic. Эти типы могут использоваться при синтезе, однако, для точного описания логики желательно использовать типы минимально необходимой разрядности или разрядности, соответствующей разрядности памяти, умножителей и интерфейсов ввода-вывода, которая необязательно является степенью двойки.

Таблица 2.1. Целочисленные типы данных с четырьмя состояниями.

| Тип | Описание | Эквивалент |
|---|---|---|

| | | |
|---|---|---|
| logic | разрядность определяется пользователем, беззнаковое | |
| reg | разрядность определяется пользователем, беззнаковое | |
| integer | 32-х разрядное знаковое целое | logic signed [31:0] |
| time | 64-х разрядное знаковое целое | |

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.

В таблице 2.2 приведены целочисленные типы языка SystemVerilog, в основе которых лежит тип bit. Не рекомендуется использовать эти типы для описания синтезируемых конструкций и моделирования логики, которая будет синтезироваться. Эти типы похожи на соответствующие типы языка C, их можно использовать для высокоуровневого моделирования и несинтезируемых поведенческих описаний.

Таблица 2.2. Целочисленные типы данных с двумя состояниями

| Тип | Описание | Эквивалент |
|---|---|---|
| bit | разрядность определяется пользователем, беззнаковое | |
| byte | 8 бит, знаковое или символ ASCII | bit signed [7:0] |
| shortint | 16 бит, знаковое | |
| int | 32 бит, знаковое | |
| longint | 64 бит, знаковое | |

Для явного определения знакового и беззнакового представления целочисленных типов используются ключевые слова signed и unsigned соответственно. При использовании данных со знаком лучше использовать знаковые типы, так как в этом случае осуществляется 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!

Пример:

```
logic signed [13:0] data;     //знаковое 14-ти разрядное число типа logic
int unsigned count;           //беззнаковое 16-ти разрядное число типа bit
```

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

Пример:

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

      always_comb
      begin
            ones=0;
            for (int i = 0; i<32; i=i+1)
            begin
                  if (din[i]) ones=ones+1;
            end
      end

endmodule
```

### 2.2.7. 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 типа и 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

| | |
|---|---|
| +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, signed 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:

```
logic [7:0] lobits, hibits;
logic [9:0] signextop,unsignextop;
logic [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 logic a,b;
logic [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 SystemVerilog делятся на процедурные и непрерывные.

Процедурные присваивания обновляют значения переменных в процедурных блоках, в которых они используются. При создании синтезируемых описаний используются блоки always_comb, always_ff и always_latch, которые будут рассмотрены в п. 2.4. В процедурных присваиваниях приемником могут быть только переменные, а не проводники.

Процедурные присваивания делятся на блокирующие, неблокирующие и непрерывные процедурные. В дальнейшем будут рассматриваться только блокирующие и неблокирующие, которые обычно используются при описании синтезируемых конструкций. Важно понимать различие между ними.

Выполнение блокирующего процедурного присваивания завершается до выполнения следующих за ним присваиваний. Блокирующее присваивание схоже с обычным описанием на языках программирования высокого уровня, где обработка операторов

осуществляется последовательно. Для задания блокирующего присваивания используется символ =.

Пример:

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

always_comb
begin
      c=b+d;
      c=a+c;
end
```

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

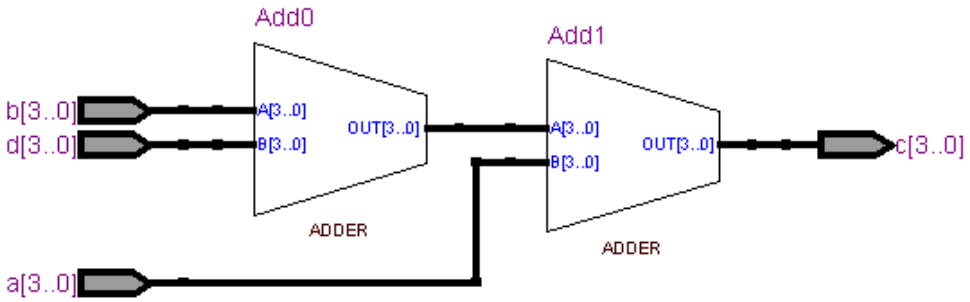


Рис. 7. Реализация блокирующих присваиваний

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 (оцениваются выражения в правых частях неблокирующих присваиваний). На втором этапе рассчитанные значения одновременно записываются в переменные-приемники. Для задания неблокирующего присваивания используется символ <=.

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

Пример:

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

always_comb
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.

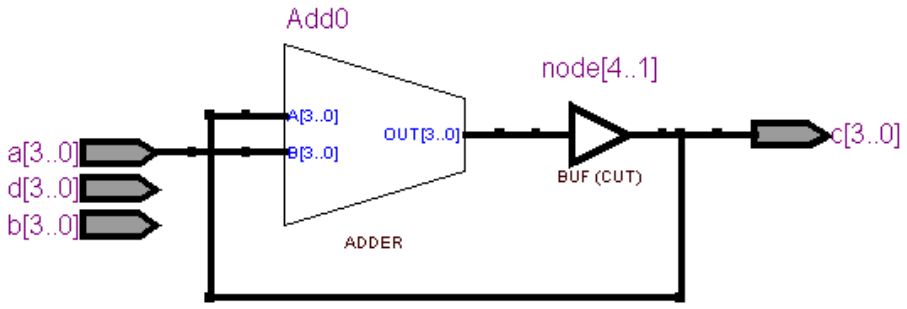


Рис. 8. Пример некорректного использования неблокирующих присваиваний

Как правило, неблокирующие присваивания используются при описании регистровых схем, так как одновременное выполнение всех присваиваний в блоке лучше отражает их логику работы. Особенности применения блокирующих и неблокирующих присваиваний при описании комбинационных и регистровых схем рассмотрены в п.2.4.

В пределах одного процедурного блока не рекомендуется смешивать блокирующие и неблокирующие присваивания, так как при этом становится сложно отследить

последовательность смены значений переменных и зависимости между ними. Кроме того, это ухудшает переносимость проекта, так как некоторые компиляторы не компилируют блоки, в которых одновременно используются блокирующие и неблокирующие присваивания.

Непрерывные присваивания осуществляются вне процедурного блока с использованием ключевого слова assign. Присваивание действует постоянно во  время работы модуля. Приемником в непрерывных присваиваниях может быть проводник или переменная.

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

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

## 2.4. Процедурные блоки

В синтезируемых описаниях на языке SystemVerilog используется три типа процедурных блоков: always_comb, always_latch и always_ff.

Блок always_comb определяет, что в нем описывается комбинационная схема. При моделировании блок выполняется в начальный момент времени и в случае изменения любых оцениваемых в нем значений. При синтезе это порождает комбинационную схему. Компилятор осуществляет контроль того, что при синтезе схемы, описываемой в блоке always_comb, не потребуется использовать элементы памяти – триггеры или защелки.

Рассмотрим в качестве примера два описания:

```
logic a,b,sel,res;
always_comb //правильно
begin
      if (sel) res=a&b;
      else res = a|b;
```

```
end

logic a,b,ena,res;
always_comb //ошибка!
begin
      if (ena) res=a&b;
end
```

В первом случае при изменении значений a,b или sel блок будет выполнен и значение переменной res будет присвоено. Таким образом, любое изменение значений a,b и sel приводит к пересчету значения переменной res, что соответствует поведению комбинационной схемы без памяти. Во втором случае значение переменной res при срабатывании блока и значении ena = 0 не будет изменяться. То есть, переменная res должна сохранить свое значение. Это соответствует поведению схемы с памятью. При детектировании в блоке always_comb пути, по которому значение переменной не присваивается, при компиляции будет выдана ошибка.

Блок always_latch описывает защелки. Рассмотренное выше описание, когда при ena =1 значение переменной зависит от оцениваемых значений других переменных и проводников, а при ena=0 фиксировано и не изменяется, соответствует поведению защелки – регистра, синхронизируемого по уровню.

```
always_latch //правильно
begin
      if (ena) res=a&b;
end
```

Так как в современных ПЛИС нет аппаратно реализованных защелок, применение такой конструкции повлечет реализацию защелки на комбинационной логике в ФП 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. Регистры, синхронизируемые фронтом, описываются в блоках always_ff.

```
logic a, ena, res;
always_ff @(posedge clk or negedge rstn)
begin
      if (!rstn) res<=0;
      else if (ena) res<=a;
end
```

Обратите внимание на оператор неблокирующего присваивания (<=), используемый в данном примере. При описании триггеров, синхронизируемых по фронту, в блоке always _ff обычно используются неблокирующие присваивания. При их применении сначала вычисляются правые части неблокирующих присваиваний, т.е. то, что будет записано в переменные, а потом осуществляется параллельная запись значений во все переменные, значения которых определяются в данном блоке. Это соответствует тому, что триггеры в синхронной схеме срабатывают одновременно по тактовому импульсу, а значения, которые при этом в них записываются, определяются исходя из их предыдущего состояния (того, в котором они находились до прихода тактового импульса), которое используется в правой части.

Важно понимать, что поведение реальной схемы несколько отличается от поведения блока при моделировании. В реальной схеме расчет правых частей неблокирующих присваиваний осуществляется комбинационной логикой и начинается после изменения значений на выходах триггеров-источников по предыдущему фронту тактового импульса, а приход нового фронта тактового импульса просто осуществляет параллельную запись

рассчитанных значений во все тактируемые им триггеры. В процессе моделирования при срабатывании блока always_ff сначала вычисляются правые части, а потом осуществляется одновременное присвоение значений переменным неблокирующими присваиваниями.

В отличие от описания комбинационных схем, при описании регистров в блоках always_ff и always_latch переменной можно не присваивать значение при срабатывании блока – это соответствует тому, что значение регистра не изменяется. Более того, при использовании неблокирующих присваиваний несколько одновременно выполняющихся неблокирующих присваиваний приводят к конфликту, как это было показано в п. 2.3.11. Таким образом, значение регистру может не присваиваться или присваиваться только одним неблокирующим присваиванием.

Рассмотрим разные варианты описания регистров.

1. Регистр, тактируемый передним фронтом тактового импульса, без дополнительных управляющих входов:

```
always_ff @(posedge clk) res<=a&b;
```

2. Регистр, тактируемый задним фронтом тактового импульса, без дополнительных управляющих входов:

```
always_ff @(negedge clk) res<=a&b;
```

3. Регистр, тактируемый передним фронтом тактового импульса, с входом асинхронного сброса с активным низким уровнем:

```
always_ff @(posedge clk or negedge clrn)
      if (!clrn) res<=0;
      else res <= a&b;
```

4. Регистр, тактируемый передним фронтом тактового импульса, с входом синхронного сброса с активным высоким уровнем:

```
always_ff @(posedge clk)
      if (sclr) res<=0;
      else res<=a&b;
```

5. Регистр, тактируемый передним фронтом тактового импульса, с входом асинхронной загрузки с активным высоким уровнем и входами разрешения и синхронного сброса:

```
always_ff @(posedge clk or negedge prn)
begin
      if (!prn) res<=1'b1;
      else if (ena)
      begin
            if (sclr) res<=0;
            else res<=a&b;
      end
end
```

Последнее описание требует детального понимания работы регистров в современной элементной базе.

Вход тактового импульса может упоминаться только в списке чувствительности, так как при проектировании синхронных схем тактовый импульс используется только для синхронизации работы триггеров и не должен использоваться в комбинационной логике, определяющей значения на остальных входах триггеров (за очень редкими исключениями, которые лежат вне рамок данного пособия). Использование тактового импульса в комбинационной логике может породить гонки между изменением сигналов на входе тактового импульса и других входах триггеров и является нарушением основных принципов проектирования синхронных схем, приводящим к синтезу неработоспособной схемы.

Асинхронные управляющие входы имеют приоритет над всеми остальными входами, поэтому в первом условном операторе в блоке анализируется вход асинхронной установки (в случае использования одновременно асинхронной установки и сброса приоритет должен быть у сброса) и только если он неактивен, выполняются остальные действия. Все асинхронные управляющие сигналы перечисляются в списке чувствительности, так как триггер реагирует на них независимо от тактового импульса.
Синхронные входы в списке чувствительности упоминаться не должны. При этом вход разрешения управляет всеми синхронными действиями, поэтому имеет наивысший приоритет среди синхронных входов. Обратите внимание, что его значение в примере,

приведенном выше, оценивается до оценки значения сигнала синхронного сброса.

Если приоритет синхронных входов должен по требованиям проекта отличаться от рассмотренного (например, синхронный сброс должен иметь больший приоритет, чем разрешение), схема может быть синтезирована, но при этом логика работы сигнала разрешения уже не будет функциональности входа разрешения триггера. Поэтому вход разрешения триггера не будет задействован, а сигнал разрешения будет подаваться в комбинационную схему, которая будет формировать значение на входе триггера. Эта схема будет реализовывать функцию не только от значений a и b, а от значений a,b, ena и res (ведь при ena = 0 на вход триггера надо будет подать его предыдущее значение), что усложнит логику:

```
always_ff @(posedge clk or negedge prn)
begin
      if (!prn) res<=1'b1;
      else if (sclr) res <= 0;
            else if (ena) res<=a&b; //ena – не сигнал на входе разрешения
end
```

Таким образом, можно сказать, что описание регистра является ”заклинанием”, правильное использование которого позволит эффективно использовать аппаратные возможности триггеров в целевой платформе, а неправильное может привести к ошибке в процессе компиляции, увеличить объем схемы, снизить ее быстродействие или просто привести к синтезу неработоспособной схемы.

Как было показано выше в примере с подсчетом количества единиц, для описания сложных комбинационных зависимостей удобно использовать блокирующие присваивания, а поведение регистровой логики лучше описывается неблокирующими присваиваниями. Поэтому при описании схем, в которых сложная комбинационная логика определяет значения на входах регистров, описание комбинационной логики и регистров можно разделить по блокам always_comb и always_ff. Имеет смысл разделять блоки, если требуется последовательное выполнение нескольких операторов для вычисления значений, что невозможно при использовании неблокирующих присваиваний.

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

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

	logic [5:0] comb_ones;

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

	always_ff @(posedge clk) ones<=comb_ones;

endmodule
```

В синтезируемом описании значение переменной может определяться только в одном блоке независимо от его типа. Представьте, что одновременно выполняются два блока и в них присваиваются разные значения одной переменной. В аппаратной реализации это соответствует короткому замыканию выходов двух схем.

Повторим сформулированные в этом разделе рекомендации:

- **При выполнении блока always_comb, реализующего комбинационную схему, значение переменной должно быть присвоено блокирующими присваиваниями хотя бы один раз в любом состоянии схемы.**
- **При выполнении блока always_ff, реализующего регистровую схему, значение**

**переменной может быть присвоено неблокирующими присваиваниями не более одного раза в любом состоянии схемы.**

- **Значение переменной может присваиваться только в одном процедурном блоке.**

## 2.5. Модули

Модули являются основными составляющими любого проекта на SystemVerilog. 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. В данном пособии используется последняя спецификация. Рассмотрим объявление модуля

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

В первой строке объявляется модуль с именем test, одноразрядными входами a и b и восьмиразрядным выходом c. Далее следует реализация. Объявление модуля завершается ключевым словом endmodule.

При создании параметризированного модуля перед именем модуля объявляется список параметров, возможно, с указанием значений по умолчанию:

```
module #(int WIDTH, DELAY = 0) partest
(
      input [WIDTH-1:0] a,b,
      output logic [WIDTH-1:0]
);
```

```
        …       //реализация модуля
endmodule
```

Параметр WIDTH не имеет значения по умолчанию, он должен быть определен при создании экземпляра модуля, для параметра DELAY определено значение по умолчанию 0.

Экземпляры модуля создаются как так же, как и переменные, с указанием имени модуля, при этом объявление допустимо в любом месте файла, но не в процедурных блоках. Приведем два примера создания экземпляра объявленного выше модуля partest:

1) Связь с портами осуществляется по порядку их перечисления, с параметрами по имени.

```
partest #(.WIDTH(8), .DELAY(1)) my_test(f, g, res);
```

2) Связь с портами и параметрами по имени.

```
partest #(.WIDTH(8), .DELAY(1)) my_test(.a(f), .b(g), .c(res));
```

Такое объявление может быть удобно, если не все порты модуля используются или удобно изменить порядок их перечисления. Если какой-то порт не используется, он не указывается в списке или указывается в списке с пустыми круглыми скобками.

## 2.6. Поведенческое описание

Рассмотрим конструкции, которые используются для описания логики работы схемы в процедурных блоках.

### 2.6.1. Блоки begin … end

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

### 2.6.2. Циклы

В SystemVerilog поддерживаются циклы for, while, do-while, foreach, forever и repeat. Операторы forever и repeat будут рассмотрен в разделе, посвященном моделированию, так как они обычно не используются при описании синтезируемых конструкций.

Синтаксис:

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

foreach ( <переменная> [<индекс>, <индекс>, …])
begin
      <операторы>
end

while (<выражение>)
begin
      <операторы>
end

do
      <операторы>
while (<выражение>)

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

Циклы for используются для повторного исполнения операторов или блока операторов. Как правило, счетчиком цикла является переменная типа int. Циклы используются при моделировании и описании комбинационной логики.

Пример:

```
logic [7:0] xb, xg;

always_comb //расчет кода Грея для двоичного числа xb
begin
      xg[0]=xb[0];
      for (int i=1; i<=7; i=i+1) xg[i]=xb[i]^xb[i-1];
end
```

Цикл foreach позволяет избежать определения циклов типа for, в том числе вложенных. При этом автоматически генерируются переменные для индексов и определяется диапазон их изменения. Пример вычисления количества единиц в двоичном числе, приведенный в п. 2.2.6, можно реализовать следующим образом:

Пример:

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

      always_comb
            foreach (din[i]) if (din[i]) ones=ones+1;

endmodule
```

Циклы while и do-while выполняются, пока выражение возвращает значение “истина”.

### 2.6.3. 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:

```
logic [2:0] result;

always_comb //if … else реализует мультиплексор 2 в 1
begin
      if (sel) result=a;
```

```
      else result=b;
end

always_comb //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_latch      //if … без else, реализует защелку
begin
      if (ena) result=a+b;
end
```

### 2.6.4. 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, что может привести к синтезу защелок 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(?).

Также 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_comb
```

```
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.

При определении оператора выбора могут использоваться ключевые слова priority и unique. Ключевое слово priority определяет, что первое совпадение отменяет оценку следующих вариантов выбора.  При моделировании осуществляется проверка и контроль на поступление  значений, не перечисленных в перечне вариантов и, если ни один из вариантов не соответствует входному значению, выдается ошибка.

Пример:

```
priority casez (code)
3'b1??: out=2'b11;
3'b01?: out=2'b10;
```

```
3'b001: out=2'b01;
endcase
```

Ключевое слово unique определяет, что все перечисленные варианты ключей являются взаимоисключающими.  При моделировании осуществляется проверка и контроль на поступление  значений. Если более чем один вариант соответствует значению ключа или ни один вариант не соответствует,  выдается ошибка. Это упрощает детектирование некорректного поведения схемы при моделировании.

Возможно применение обратной оценки в операторе case, когда ключом является константа, а вариантами выбора – переменная. Следующий пример иллюстрирует реализованный таким образом приоритетный шифратор, который определяет состояние системы в зависимости от значения битов управляющего слова. Если бит сброса установлен, значение остальных битов не учитывается и система находится в сбросе. Бит приема имеет более высокий приоритет, чем бит передачи. Для определения индексов битов используется директива компилятора `define (директивы компилятора SystemVerilog похожи на директивы препроцессора C и позволяют определять константы и реализовывать условную компиляцию), а состояние mode определено как перечислимый тип.

```
`define IX_RESET  0
`define IX_TX     2
`define IX_RX     5
enum {M_RESET, M_TX, M_RX} mode;
always_comb
begin
      priority case (1'b1)
      ctl[`IX_RESET]: mode=M_RESET;
      ctl[`IX_TX]: mode=M_TX;
      ctl[`IX_RX]: mode=M_RX;
      endcase
end
```

# 3. Применение языка SystemVerilog

## 3.1. Примеры модулей

### 3.1.1. Дешифратор

```
//дешифратор 3->8 на основе оператора выбора
//объявление модуля в стиле Verilog-1999
module decode_case(din,dout);
      input logic [2:0] din;
      output logic [7:0] dout;

      always_comb
      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 SystemVerilog
module decode_shift
#(int WIDTH = 3)
(input logic [WIDTH-1:0] din, output logic [2**WIDTH-1:0] dout);

      always_comb dout=1<<din;

endmodule
```

### 3.1.2. Синхронный счетчик до 5 со сбросом и разрешением

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

	//срабатывает по фронту тактового импульса и асинхронного сброса
	always_ff @ (posedge clk or negedge resetn)
	begin
		if (!resetn)
			count <= 0;
		else if ((enable)&&(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 абстрактного описания схем и задания тестовых воздействие используется  более широкое подмножество языка, содержащее так называемые несинтезируемые конструкции. Возможности несинтезируемого подмножества языка SystemVerilog, применяемого для верификации разрабатываемых модулей путем моделирования, предоставляют определение программ тестов, реализацию объектно-ориентированного подхода к описанию компонентов тестового окружения, контроль утверждений о свойствах схемы в процессе моделирования, определение требований к покрытию и анализ тестового покрытия и т.д.. С применением языка SystemVerilog реализована методология верификации UVM, которая определяет методологию верификации и библиотеку классов, используемых для формирования тестового окружения и управления программой теста. Рассмотрение подмножества, используемого для верификации, приведено в []. Описание UVM можно найти в [].

В данном пособии рассмотрены только базовые возможности. 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_ff. Рассмотренное ранее условие @( posedge clk) – это событие. 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
```

Обратите внимание, что при моделировании уже не обязательно использовать специальные типы блока always, определяющие синтез конкретной аппаратной реализации.

Для создания именованного события и работы с ним 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 без событий 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. Циклы repeat и forever

Рассмотренные в разделе, посвященном синтезу операторы выбора, условные операторы, циклы и т.д. могут применяться и для создания несинтезируемых описаний. Рассмотрим в дополнение еще два оператора, которые применяются только в несинтезируемых описаниях.

Цикл repeat позволяет повторить операторы указанное количество раз.
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.sv" //включаем содержимое файла libmodule.sv
```

## 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 используется средством 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_shift.sv

```
module decode_shift
#(parameter WIDTH = 3)
  (input logic [WIDTH-1:0] din, output logic [2**WIDTH-1:0] dout);

        always_comb dout=1<<din;

  endmodule
```

Файл decode_test.sv

```
//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;

    logic [WIDTH-1:0] enc_data;
    logic [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
      //останавливаем моделирование
      $finish;
    end

    //создаем экземпляр тестируемого модуля, переопределяя параметр по умолчанию
    decode_shift #(WIDTH) uut_inst(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.sv (31): $finish called.
# KERNEL: Time: 160 ns,  Iteration: 0,  TOP instance,  Process: #INITIAL#18_1.
# KERNEL: stopped at time: 160 ns
```

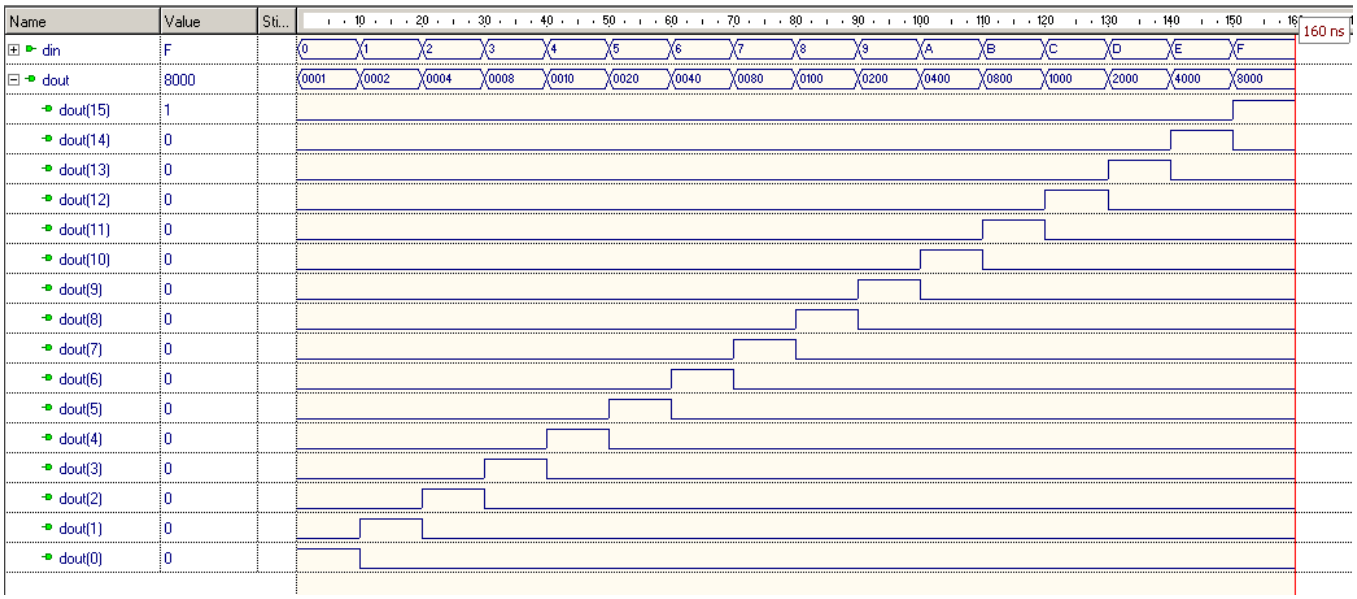


Рис. 9. Временная диаграмма работы дешифратора

### 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 bin_cnt.sv:

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

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

endmodule
```

Файл counter_test.sv:

```
//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;
logic clk,ena,rst;
logic [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 тестируемого модуля
    bin_cnt uut_inst(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.sv (55): $finish called.
# KERNEL: Time: 420 ns,  Iteration: 0,  TOP instance,  Process: #INITIAL#41_3.
# KERNEL: stopped at time: 420 ns
```

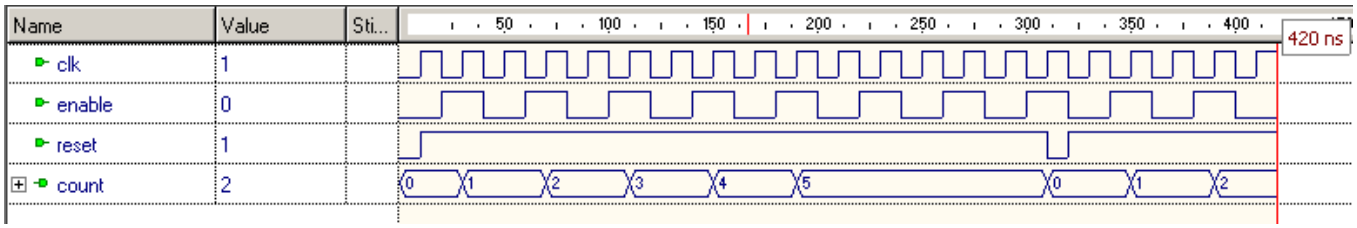


Рис. 10. Временная диаграмма работы счетчика