Verilog HDL

Содержание

1. Введение в Verilog HDL

2. Лексические соглашения

Исходные текстовые файлы языка Verilog представляют собой последовательность лексических элементов, состоящих из одного или нескольких символов. Расположение элементов исходного кода имеет свободный формат, т.е. пробелы, символы табуляции, пустые строки синтаксически безразличны. Однако пробелы и пустые строки очень важны для структурирования исходного кода, что повышает его читабельность. В языке имеют место следующие лексические элементы:

- Операторы
- Пробелы и комментарии
- Числа
- Строки
- Идентификаторы
- Ключевые слова

Эти лексические элементы будут рассмотрены в этом разделе.

2.1 Операторы

Операторы Verilog HDL могут иметь от одного до трех символов и будут подробно рассмотрены в параграфе 4.2 «**Операторы языка Verilog**». Операторы могут быть унарными и бинарными, а также есть один оператор который имеет три операнда – условный оператор. Унарные операторы располагаются слева от своего операнда. Бинарные операторы располагаются между операндами. Условный оператор имеет два символа, разделяющие операнды.

2.2 Пробелы и комментарии

К пробелам в языке относятся собственно пробелы, символы табуляции, пустые строки. Verilog HDL игнорирует эти символы, кроме случаев, когда они разделяют лексические элементы. Также пробелы и символы табуляции не игнорируются в строках.

В языке Verilog приняты две формы для ввода комментариев. Однострочные комментарии начинаются с символов // и заканчиваются концом строки. Многострочные комментарии или иначе блочные комментарии начинаются с символов /* и заканчиваются символами */. Многострочные комментарии не могут быть вложенными, но однострочный комментарий внутри блочного допустим.

2.3 Числа

Числа или иначе константы могут определяться в десятичном, шестнадцатеричном, восьмеричном или двоичном форматах. В языке Verilog предусмотрены две формы для записи чисел. Пкервая форма представляет собой простое десятичное число как последовательность цифр от 0 до 9 и опционально может предварятся символами плюса или минуса. Вторая форма представления чисел имеет следующий формат:

<size> <base_format> <number>

Поле **size** содержит десятичные цифры и указывает разрядность константы в битах. Это поле опционально и может опускаться в случаях если разрядность константы заранее определена (Допустим производится присвоение константного значения переменной, разрядность которой задана при ее объявлении). Если разрядность не определена, то разрядность принимается по умолчанию равной 32 (32 в большинстве систем синтеза, в некоторых разрядность по умолчанию может отличаться). Второе поле **base_format** содержит букву, определяющую формат представления числа (десятичный, двоичный ...). Эта буква предваряется символом одиночной кавычки ('). Формат представления определяется следующими буквами: d – десятичный; h – шестнадцатиричный; o – восмеричный и b – двоичный. Допускается вводить эти буквы как прописными, так и строчными. Последнее поле **number** содержит цифры допустимые для выбранного формата: от 0 до 9 для десятичного формата; от 0 до 7 для восмеричного; 0 и 1 для двоичного; от 0 до 9 и буквы a, b, c, d, e, f для шестнадцатеричного. Буквенные обозначения могут вводится как строчными, так и прописными буквами.

Числа имеющие знаковое представление предваряются опциональными символами плюса и минуса перед полем **size**. Эти символы перед полем **number** недопустимы. Запись -8'h5A эквивалентна записи -(8'h5A), а запись 8'h-5A вызовет синтаксическую ошибку.

Помимо указанных цифр в поле **numbers** могут присутствовать буквы x, X, z, Z и символ (?) (В поле **size** они недопустимы). Буквы x и X обозначают неизвестное (неопределенное) состояние, т.е. состояние соответствующих битов неизвестно. Буквы z и Z обозначают состояние высокого импеданса – z-состояние или отсутствие источника (драйвера). Символ ? эквивалентен символу z и предназначен для лучшей читабельности кода в случаях когда это состояние безразлично (don't-care). Существует возможность сокращать запись числа: например 8'b1 будет эквивалентно записи 8'b11111111, но запись 8'b01 будет эквивалентна записи 8'b00000001. В силу того, что такие сокращенные записи могут мешать читабельности кода, употреблять их без особой надобноси не следует, пожалуй допустимыми можно считать записи 8'b0, 8'b1, 8'bx, можно записать и 16'hx – это эквивалентно 16'hxxxx. В остальных случаях лучше делать подробную запись числа.

Для улучшения читабельности допускаются еще два приема: допускается вставлять пробелы и символы табуляции между полями записи, например 8'hB6 можно записать как 8 'h B6. Вставлять пробелы внутри поля **number** не допускается, но можно всталять символ подчеркивания (_), для разбиения числа на некоторые группы. Примеры 83465 можно записать как 83_465; 8'b01100111 можно записать как 8'b0110_0111. Символ подчеркивания в поле **size** или перед полем **number** недопустим.

2.4 Строки

Строками в языке Verilog являются последовательностями символов, заключенных в двойные кавычки и располагающиеся в одной линии (одной строке исходного кода). Строки могут использоваться в языке как операнды в выражениях представляя собой последовательность 8-и разрядных (байтовых) ASCII-кодов, где один ASCII-символ представляется как одна буква, цифра или сперциальный символ. Более подробно об использовании строк в языке рассказывается в пункте 4.3.3 "**Строки**".

2.5 Идентификаторы, ключевые слова, системные имена

Идентификаторы, ключевые слова и системные имена строятся по одним и тем же правилам, поэтому они и объединены в данный параграф. Общие правила следующие:

- Идентификаторы не должны совпадать с ключевыми словами;
- Идентификаторы не могут начинаться с символов ($) или (` – апостроф);
- Идентификатор не может начинаться с цифры;
- Длина идентификатора по стандарту не ограничена, но может ограничиваться в конкретном компиляторе;
- Ключевые слова всегда пишутся строчными буквами;
- Системные имена начинаются с символа ($);
- Директивы компилятора начинаются с символа (` – апостроф);
- Компилятор различает строчные и прописные буквы.

2.5.1 Идентификаторы

Идентификаторы используются в языке Verilog в качестве символических имен для обозначения переменных, констант, модулей, функций, задач и т.д. и могут использоваться для обозначения этих объектов в любом месте описания. Идентификатор представляет собой последовательность из букв, цифр, символов доллара ($) и символов подчеркивания (_) с учетом изложенных выше правил. Другие символы могут использоваться только если идентификатор начинается с символа \ (escaped identifiers) – в этом случае он может содержать все печатные символы. Заканчивается такой идентификатор пробелом, символом табуляции или новой строкой.

Замечание:

Обычные идентификаторы могут содержать только буквы латинского алфавита, Для использования букв другого алфавита необходимо идентификатор начинать с символа `\`. *Подобные идентификаторы могут не поддрживаться некоторыми компиляторами.*

Поскольку компилятор различает строчные и прописные буквы, можно ввести допустим такой: Module – это слово совпадает по произношению с ключевым словом ***module***, но компилятор ошибки не выдаст, поскольку слово начинается с прописной буквы. Такая возможность есть, но строго не рекомендуется вводить идентификаторы, отличающиеся от ключевых слов только регистром, это потенциально приведет к путанице в будущем и говорит о низкой квалификации разработчика.

Идентификаторы по возможности должны отражать свойства объекта к которому они принадмежат, это повышает самодокументированность кода. Например лучше написать Shifter, чем Sftr: в первом случае понятно, что идет речь о сдвиговом регистре, а что имелось в виду во втором случае не очень. Следует избегать в качестве идентификаторов одиночных букв, особенно a, b, c, d, e, f, x, z, h, b, o – в любом регистре они сопадают с резервированными обозначениями. К сожалению к языку Verilog не разработано чего-либо похожего на «Венгерскую нотацию» как в языке C, поэтому давать какие-либо общие рекомендации трудно, но есть несколько простых правил, улучшающие читабельность кода:

- Поскольку все ключевые слова прописываются строчными буквами, идентификаторы имен переменных, модулей и т.п. лучше начинать с прописной буквы а продолжать строчными.
- Идентификаторы – имена портов или параметров – лучше писать прописными буквами целиком, в любом месте кода будет понятно с чем мы имеем дело – с портом или с обычной переменной.
- Имена портов или переменных на котором действуют сигналы с активным уровнем в лог.0 лучше начинать со строчной буквы n (negative), это помогает избегать ошибок, связанных с некорректным активным уровнем сигнала (значением переменной). Вместо буквы n можно использовать символ подчеркивания (_).
- Желательно разработать для себя и дополнительные правила, которые помогут вам или другому разработчику легче разбираться в исходном коде.

2.5.2 Ключевые слова

Ключевыми словами начинаются предопределенные идентификаторы, используемые для определения языковых конструкций. Ключевое слово предваряемое символом \ ключевым словом уже не является и компилятором интерпретируется как обычный идентификатор. Использовать этот прием строго не рекомендуется (см. пункт выше). Все ключевые слова прописываются только строчными буквами. В документе ключевые слова прописываются полужирным курсивом. Ниже приведен список ключевых слов языка Verilog.

always	***event***	***or***	***strong1***
and	***for***	***output***	***supply0***
assign	***force***	***parameter***	***supply1***
begin	***forever***	***pmos***	***table***
buf	***fork***	***posedge***	***task***
bufif0	***function***	***primitive***	***time***
bufif1	***highz0***	***pull0***	***tran***
case	***highz1***	***pull1***	***tranif0***
casex	***if***	***pullup***	***tranif1***
casez	***initial***	***pulldown***	***tri***
cmos	***inout***	***rcmos***	***tri0***
deassign	***input***	***reg***	***tri1***
default	***integer***	***release***	***triand***
defparam	***join***	***repeat***	***trior***
disable	***large***	***rnmos***	***trireg***
edge	***macromodule***	***rpmos***	***vectored***
else	***medium***	***rtran***	***wait***
end	***module***	***rtranif0***	***wand***
endcase	***nand***	***rtranif1***	***weak0***
endmodule	***negedge***	***scalared***	***weak1***
endfunction	***nmos***	***small***	***while***
endprimitive	***nor***	***specify***	***wire***
endspecify	***not***	***specparam***	***wor***
endtable	***notif0***	***strength***	***xnor***
endtask	***notif1***	***strong0***	***xor***

2.5.3 Системные имена и директивы компилятора

К системным именам относятся имена системных задач и функций предопределенные в стандарте Verilog HDL. Все системные имена начинаются с символа ($), поэтому спутать с обычным идентификатором их трудно (идентификатор не может содержать этот символ в начале). Полный список системных имен здесь не приводится, поскольку они будут подробно рассмотрены в разделе "**Системные функции и задачи**". Системные имена всегда вводятся строчными буквами. В документе системные имена выделяются полужирным шрифтом, например **$display**.

К директивам компилятора относятся предопределенные стандартом идентификаторы, предваряемые символом апострофа (`). Директивы компилятора вводятся строчными буквами. Директивы компилятора будут подробно рассмотрены в разделе "**Директивы компилятора**". В документе они также выделяются полужирным шрифтом, например **`define**, **`include**.

3. Типы данных

Данные в языке Verilog предназначены для сохранения состояний (регистры) и для передачи состояний между моделируемыми объектами (цепи).

3.1 Возможные состояния

В языке Verilog данные могут принимать одно из четырех возможных состояний:

1 – представляет логическую 1 или значение «истинно» (true)
0 – представляет логический 0 или значение «ложно» (false)
z – представляет состояние высокого импеданса
x – представляет неизвестное логическое состояние

Почти все типы данных в Verilоge принимают одно из этих состояний за небольшим исключением: такой тип данных как event вообще не удерживает состояний, а тип trireg может принимать z-состояние только в начальный момент времени, до первой инициализации – в дальнейшем в это состояние он больше не переходит.

3.2 Регистры и цепи

Регистры и цепи являются двумя наиболее важными группами данных. Эти группы различаются по способу назначения данных и их удержания. Также они представляют собой различные логические структуры.

3.2.1 Цепи

Цепи представляют типы данных предназначенных для соединения между собой структурных объектов, например, логических вентилей. Цепи не удерживают своего состояния (за исключением типа trireg) их состояние дожно непрерывно удерживаться (continious assignment) выходом с логического вентиля или комбинационной схемы. Если к цепи не подключен источник (driver) то цепь переходит в состояние высокого импеданса. Исключение составляет цепь типа trireg которая удерживает предыдущее состояние (но это не регистр!).

3.2.2 Регистры

Регистры представляют собой элементы хранения данных. Ключевое слово для данных этого типа – ***reg***. Регистры сохраняют свое состояние от одного назначения до другого. Операция назначения состояния действует как триггер который изменяет состояние элемента хранения данных. По умолчанию регистровые данные принимают неизвестное состояние (x).

3.2.3 Объявление цепей и регистров

Цепи объявляются по следующим синтаксическим правилам:

nettype [drive_strength][1] [expand_range] [delay] <list_of _variables>;

[1] Элементы объявления заключенные в квадратные скобки являются необязательными

nettype – тип цепи, может принимать значения ***wire***, ***wand***, ***wor***, ***tri*** и т.д.
drive_strength – определяет «мощность» источника (драйвера).
expand_range – определяет разрядность цепи, по умолчанию цепь одно-разрядная.
delay – определяет задержку распостранения по цепи в установленных временных единицах или иначе указывает задержку от момента назначения состояния цепи до его фактического установления.
list_of_variables – список объявляемых переменных, разделяемых запятой.

Есть отличия при объявлении переменной типа ***trireg***:

trireg [charge_strength] [expand_range] [delay] <list_of _variables>;

trireg – ключевое слово.
charge_strength – определяет «емкость» источника.

Регистры определяются иначе:

reg [range] <list_of _reg_variables>;

reg – ключевое слово.
range – определяет разрядность регистровой переменной, по умолчанию регистр одноразрядный.
list_of_reg_variables – список объявляемых регистровых переменных, разделяемых запятой.

Ключевые слова определяющие тип цепи приведены ниже:

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

«Мощность» определяется одним из следующих выражений:
drive_strength: (<strength1>, <strength0>) или (<strength0>, <strength1>), где strength1 и strength0 опрределяют «мощность» источника в состояниях лог.1 и лог.0 соответственно.
Ключевые слова определяющие «мощность» следующие:

strength1:	***supply1***	***strong1***	***pull1***	***weak1***	***highz1***
strength0:	***supply0***	***strong0***	***pull0***	***weak0***	***highz0***

Комбинации (***highz1***, ***highz0***) или (***highz0***, ***highz1***) являются недопустимыми. По умолчанию «мощность» цепи определяется как (***strong1***, ***strong0***).

«Емкость» источника определяется так:
charge_sterngth: (<capacitor_size>), где capacitor_size определяется одним из следующих ключевых слов:

capacitor_size:	***small***	***medium***	***large***

По умолчанию «емкость» устанавливается как (***medium***).

Замечание:
«мощность» или «емкость» источника могут иметь практическое значение при моделировании схемы с целью получить наиболее объективные результаты, большинство систем синтеза эти объявления игнорируют.

Объявления разрядности для цепи и регистра несколько отличаются:

Цепь: [<msb>:<lsb>] ***scalared*** [<msb>:<lsb>] ***vectored*** [<msb>:<lsb>]
Регистр: [<msb>:<lsb>]
msb – старший значащий бит (most significant bit)
lsb – младший значащий бит (least significant bit)

Ключевые слова ***scalared*** и ***vectored*** определяют скалярная это цепь или это вектор. Различие заключается в следующем: скалярность подразумевает, что разработчик может использовать (читать или устанавливать) отдельные биты или часть (группу) битов. Векторизованная цепь может быть прочитана или установлена лишь целиком. По умолчанию цепь устанавливается как ***scalared***.

Границы диапазона msb и lsb представляются либо в числовой форме (константы) или в виде константных выражений – т.е. все входящие в выражение операнды являются константами. Кроме того границы диапазона не могут представляться отрицательными или дробными числами. При этом msb может быть меньше или равно lsb.

Замечание:
По стандарту разрядность цепей или регистров не ограничивается, но системы синтеза могут иметь ограничения на разрядность переменных.

Многоразрядные цепи и регистры подчиняются арифметике по модулю 2 в степени n, где n – количество разрядов.

Временные задержки (delay) будут подробно рассмотрены в пункте XX.

3.2.4 Примеры объявлений цепей или регистров

В ледующем листинге приведены примеры объявлений цепей и регистров.

Листинг3-1:
```
wire a, b, c;          // Объявление трех одноразрядных цепей a,b,c
wire [7:0] busa;       // Объявление 8-разрядной скалярной цепи busa
wire [0:9] busb;       // Объявление 10-разрядной скалярной цепи busb
wand vectored  [7:0] andbus;      // Объявление 8-разрядной
                                  //векторизованной цепи andbus
reg (highz1, strong0)[3:0] opdrain;     // Объявление 4-разрядного
                                        //регистра с открытым
                                        // коллектором opdrain
reg [Nbit - 1:0] dout;      // Объявление Nbit-разрядного регистра dout
trireg (small) [7:0] treg;  // Объявление 8-разрядной скалярной цепи
                            // treg
```

3.3 Типы цепей

Существует несколько различных типов цепей – все эти типы предствлены в следующих секциях.

*3.3.1 Типы **wire** и **tri***

Цепи типов ***wire*** и ***tri*** предназначены для соединения элементов. Эти типы идентичны по синтаксису и по их функциональному назначению. Различие в ключевых словах показывает лишь их назначение в модели. Цепи типа ***wire*** обычно используются для того, чтобы показать, что цепь имеет лишь один драйвер, в то время как тип ***tri*** используется чтобы показать, что у цепи несколько источников – обычно это трех-стабильные шины. Логические конфликты в случае нескольких источников приводят к неопределенному состоянию, за исключением случаев, когда лишь один источник находится в состоянии лог.1 или лог.0, тогда как остальные находятся в z-состоянии. Это наглядно демонстрируется таблицей истинности (табл. 3-1) для случая цепи с двумя драйверами. По вертикали отображены состояния одного источника, по горизонтали – другого. Необходимо заметить, что «мощность» источников одинакова.

Табл. 3-1 Типы wire и tri

	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	z

*3.3.2 Типы **wand/triand** и **wor/trior***

Типы ***wand***, ***triand***, ***wor***, ***trior*** используются для моделирования цепей с разрешающей логикой. Эти типы позволяют разрешать конфликты в цепях с несколькими источниками. Цепи ***wand*** или ***triand*** создают такую конфигурацию, что результирующее состояние будет лог.0 в случае если хотя бы один из источников находится в лог.0. Эти типы идентичны по синтаксису и по функциональному назначению (также как для типов ***wire*** и ***tri***). Ниже приведена таблица истинности для цепи ***wand*** или ***triand*** с двумя источниками равной «мощности» (табл. 3-2).

Табл. 3-2 Типы wand и triand

	0	1	x	z
0	0	0	0	0
1	0	1	x	1
x	0	x	x	x
z	0	1	x	z

Цепи ***wor*** или ***trior*** создают такую конфигурацию, что результирующее состояние будет лог.1 в случае если хотя бы один из источников находится в лог.1. Эти типы идентичны по синтаксису и по функциональному назначению (также как для типов ***wire*** и ***tri***). Ниже приведена таблица истинности для цепи ***wor*** или ***trior*** с двумя источниками равной «мощности» (табл. 3-3).

Табл. 3-3 Типы wor и trior

	0	1	x	z
0	0	1	x	0
1	1	1	1	1
x	x	1	x	x
z	0	1	x	z

3.3.3 Тип ***trireg***

Цепи типа ***trireg*** сохраняют свое состояние и используются для моделирования цепей с сохранением заряда (отсюда и термин «емкость»). Цепи этого типа могут быть в одном из двух состояний:

- В состоянии когда хотя бы один источник находится в состояниях лог.1, лог.0 или неизвестном состоянии – это состояние распространяется по цепи. Это состояние с подключенным драйвером (driven state). «Мощность» в этом состоянии может быть ***supply***, ***strong***, ***pull*** или ***weak*** в зависимости от мощности источника.
- В состоянии когда все источники находятся в состоянии высокого импеданса. Цепь удерживает последнее состояние когда источник был подключен, высокоимпедансное состояние не распространяется по цепи этого типа. «Мощность» «драйвера» цепи в этом состоянии может быть ***small***, ***medium*** или ***large***.

В случае конфликта источников подключенных к цепи правило разрешения аналогично представленному в табл. 3-1, за тем исключением, что вместо z-состояния цепь будет удерживать состояние предшествующее установке всех источников в состояние высокого импеданса.

3.3.4 Типы ***tri1***, ***tri0***, ***supply1***, ***supply0***

Типы ***tri1*** и ***tri0*** моделируют цепи с резистивной подтяжкой (pull-up, pull-down) к состоянию лог.1 и лог.0 соответственно. Когда все источники цепи такого типа находятся в состоянии высокого импеданса, то цепь устанавливается в соответствующее лог. состояние с «мощностью» ***pull***. Таблица истинности в остальных состояниях идентична табл. 3-1 (если «мощности» драйверов по крайней мере ***strong***).

Типы ***supply1*** и ***supply0*** моделируют источники питания подключенные к этой цепи. Тип ***supply1*** моделирует источник питания Vdd или Vss, а ***supply0*** моделирует подключение к «земле». Цепи подобного типа никогда не подключаются к выходу вентиля и к оператору назначения состояния, поскольку их «мощность» выше «мощности» любого другого источника.

Замечание:

Все перечисленные типы цепей кроме типов ***wire*** *и* ***tri*** *могут иметь практическое значение при моделировании схемы с целью получить наиболее объективные результаты, большинство систем синтеза эти типы приводят к допустимым типам* ***wire*** *и* ***tri****. Использование этих типов может привести к различным результатам в процессе моделирования и реального функционирования схемы. Применение вышеперечисленных типов допустимо в случае если точно известно, что система синтеза правильно интерпретирует эти типы и платформа на которой разрабатывается устройство позволяет реализовать цепи подобных типов.*

3.4 Элементы памяти

В языке Verilog эмлементы памяти моделируются как массивы регистров. Подобные массивы могут быть использованы для моделирования ПЗУ, ОЗУ или регистровых файлов. Каждый регистр в массиве определяется как элемент или слово и адресуется одномерным индексом. Многомерных массивов в Verilog HDL по стандарту 1995г. не предусмотрено. Ниже представлен синтаксис массива регистров.

reg [range] <memory_name> [<const_expr> : <const_expr>];

Здесь **range** определяет разрядность элемента массива, как и для регистра он может объявляться в возрастающем и убывающем диапазоне (это важно – игнорирование возрастание или убывание диапазона может приводить к ошибкам), может представлять собой константные выражения. Имя массива – **memory_name** – подчиняется всем правилам написания идентификаторов. Интерес представляет размерность массива заключенная в квадратные скобки. Выражения **const_expr** могут представлять из себя любую положительную константу или константное выражение. В отличие от разрядности регистра возрастание или убывание диапазона не играет никакой роли.

Существует некоторое ограничение: нельзя непосредственно обратится к отдельному биту или группе битов элемента массива. Для этого необходимо скопировать элемент в отдельный регистр и лишь тогда обратиться к соответствующим битам регистра. Эта ситуация представлена на листинге 3-2.

Листинг3-2:

```
reg [7:0] temp;              // Объявление 8-разрядного регистра
reg [3:0] data;              // Объявление 4-разрядного регистра
                             // приемника
reg [7:0] mem [0:255];       // Объявление массива

// Инициализация массива
- - -
- - -
// Чтение битов 0-3 из 25 элемента массива mem в регистр data
temp = mem[24];              // Читаем элемент массива в регистр temp
data = temp[3:0];            // Копируем нужные биты в регистр data
// data = temp[0:3]          - Ошибка! Нарушен порядок
                             //возрастания/убывания диапазона
                             // регистра temp
```

Обращение к массиву без указания индекса является ошибкой, обращение всегда ведется к конкретному элементу массива. Индекс может быть любым выраженем, если его результат положительная целая величина.

3.5 Переменные типов *integer* и *time*

При моделировании цифровых устройств могут быть очень полезны такие переменные языка Verilog как ***integer*** и ***time***. Конечно можно использовать отдельный регистр для вычисления текущего времени в процессе моделирования, но удобнее использовать переменные типа ***integer*** или ***time***, так как это повышает

самодокументирование кода. Синтаксис для объявления переменных этого типа приведен ниже.

time <list_of_variables>; - объявление переменных типа ***time***;
integer <list_of_variables>; - объявление переменных типа ***integer***.

Переменные типа ***time*** используются в случаях, когда в процессе моделирования требуются какие-либо манипуляции с временными отсчетами для диагностики и устранения ошибок. Этот тип данных обычно используется совместно с системной функцией **$time**, размерность этой переменной 64 бита.

Переменные типа ***integer*** обычно используются как переменные общего назначения, для хранения отсчетов, которые не связаны с каким-либо аппаратным регистром. Разрядность переменных типа ***integer*** 32 бита.

Замечание:
Некоторые системы синтеза могут ограничивать размерность переменных типа ***integer*** *и* ***time****.*

Допускается использование массивов переменных типа ***integer*** и ***time***. Пример объявления массивов приведен ниже:

integer a[1:64]; // массив из 64 переменных типа ***integer***
time change_history[1:1000]; // массив из 1000 переменных типа ***time***

Значения этим переменным присваиваются также как и регистровым переменным. Переменная типа ***time*** ведет себя точно также как 64-х разрядная регистровая переменная. Это беззнаковая переменная и подчиняется беззнаковой арифметике. Пременные типа ***integer*** напротив подчиняются знаковой арифметике, что приводит к различным результатам по сравнению с 32-х разрядной регистровой переменной. Регистровые переменные подчиняются беззнаковой арифметике. Простой пример: мы можем присвоить регистровой переменной отрицательное значение – допустим 16-и разрядной переменной мы присвоили значение -24 (-16'd24), а в результате в следующей арифметической операции в этой переменной окажется число 65512 – целое положительное число! С переменной типа ***integer*** этого не произойдет – в арифметической операции будет использовано именно -24. Об этом важном различии необходимо помнить при операциях с регистровыми и целыми переменными.

3.6 Вещественные числа

Кроме переменных типа ***integer*** и ***time*** Verilog HDL поддерживает использование вещественных констант и переменных. Объявление подобных переменных выполняется с помощью ключевого слова ***real***. За некоторым исключением эти переменные могут использоваться в тех же случаях что и переменные типа ***integer*** и ***time***.

real <list_of_variables>; // Синтаксис объявления переменной типа ***real***

Основные отличия переменных типа ***real***:

- Не все операторы языка Verilog могут быть использованы с переменными типа ***real***.
- Не допускается объявления диапазона для этих чисел.
- По умолчению значение этих переменных равно нулю.

3.6.1 Определение вещественных чисел

Вещественные числа могут определяться как десятичной нотации (например 12.47), так и в научной нотации (например 38e8, что обозначает 38 умноженное на 10 в степени 8). Вещественные числа определенные с десятичной точкой должны иметь хотя бы одну цифру с каждой стороны от десятичной точки. Ниже приведены некоторые примеры корректных реальных чисел:

1.2
0.1
2394.26331
1.2E12 (Символ экспоненты может быть как e так и E)
1.30e-2
0.1e-0
23E10
29E-2
236.123_763_e-12 (Символ подчеркивания игнорируется)

Пример неверного определения вещественных чисел – опущена цифра с одной из сторон от десятичной точки:

.12
3.e7
.2E-7

3.6.2 Операторы и вещественные числа

Результатом применения логических операторов и операторов отношения к вещественным числам является однобитовое скалярное значение. Не все операторы могут быть использованы с вещественными числами. В следующем разделе приведена таблица операторов и указано, какие операторы могут быть применены, а какие нет. Также вещественные константы и переменные не могут быть применены в следующих контекстах:

- Дескрипторы фронта/среза (***posedge***, ***negedge***) не могут быть применеы к вещественным переменным.
- Выбор отдельного бита (bit-select) или группы битов (part-select) для переменной, объявленной как ***real***.
- Вещественные числа не могут быть индексами для выбора отдельного бита или группы битов.
- Не может быть массивов вещественных чисел.

3.6.3 Преобразование вещественных чисел

Вещественные числа преобразуются к целому типу по следующему правилу: преобразование идет к ближайшему целому числу, т.е. числа 35.7 и 35.5 будут преобразованы к числу 36, а число 35.2 – к 35. Подобное преобразование имеет место когда вещественное число присваивается к переменной целого типа. В разделе «**Системные функции и задачи**» обсуждаются системные функции для точного преобразования вещественных чисел.

3.7 Параметры

Параметры языка Verilog не относятся ни регистрам ни к цепям. Параметры не могут быть переменными – это всегда константы. Объявляются параметры следующим образом:

parameter <list_of_assignments>;

Замечание:
Некоторые системы синтеза позволяют указывать диапазон в объявлении параметра.

При объявлении параметр всегда должен быть проинициализинован – **list_of_assignmens** содержит выражения присваивания, разделенные запятой, где с правой стороны должна находиться константа или константное выражение. При этом константное выражение может содержать как константы, так и ранее определенные параметры. Примеры объявления параметров приведены в листинге 3-3.

Листинг3-3:

```
parameter msb = 7;                // Определяет msb как константу равную 7
parameter e = 25, f = 9;          // Объявление 2-х констант
parameter r = 5.7;                // Объявление константы типа real
parameter byte_size = 8, byte_mask = byte_size - 1;
parameter average_delay (r + f) /2;
```

Хотя параметры представляют собой константы, на этапе имплементации (синтеза) их значения могут быть изменены, что позволяет модифицировать экземпляры модулей. Параметры могут быть модифицированы с помощью выражения defparam или при создании экземпляра компонента. Типичным применением параметров является указание задержек и разрядности переменных.

4. Выражения в языке Verilog

4.1 Операнды в выражениях языка Verilog

Этот раздел описывает операторы и операнды доступные в Verilog HDL, а также как использовать их в формальных выражениях. Выражения представляют собой конструкции, которые комбинируют операнды с операторами для получения результата, который представляет собой функцию от значений операндов и семантического смысла операторов. Кроме того, выражением может быть любой разрешенный операнд – допустим, группа битов регистра. Существуют некоторые ограничения при написании константных выражений – константные выражения могут содержать только константы или предварительно определенные параметры, но могут содержать любые операторы, приведенные в табл. 4-1. При использовании в выражениях переменных типа ***integer*** и ***time***, они воспринимаются так же как и переменные типа ***reg***.

В языке Verilog можно использовать операнды следующих типов:

- числа (включающие и вещественные)
- цепи
- регистры, ***integer*** и ***time***-переменные
- одиночные биты цепи
- одиночные биты регистров
- группу битов цепи
- группу битов регистра
- элементы памяти (массива)
- результат конкатенации
- вызовы пользовательских или системных функций, возвращающих любое из перечисленных выше значений

4.2 Операторы языка Verilog

Символы операторов Verilog HDL похожи на те, что используются в языке C. Список операторов приведен в табл. 4-1.

Табл. 4-1 Список операторов Verilog HDL

Символ	Назначение	Использование с типом real
{}	Конкатенация (concatenation)	Не допустимо
+ - * /	Арифметические (arithmetic)	Допустимо
%	Модуль (modulus)	Не допустимо
> >= < <=	Отношения (relational)	Допустимо
!	Логическое отрицание (logical negation)	Допустимо
&&	Логическое И (logical and)	Допустимо
||	Логическое ИЛИ (logical or)	Допустимо
==	Логическое равенство (logical equality)	Допустимо
!=	Логическое неравенство (logical inequality)	Допустимо
===	Идентичность (case equality)	Не допустимо
!===	Неидентичность (case inecuality)	Не допустимо
~	Побитовая инверсия (bit-wise negation)	Не допустимо
&	Побитовое И (bit-wise and)	Не допустимо
|	Побитовое ИЛИ (bit-wise or)	Не допустимо

Символ	Назначение	Использование с типом real
^	Побитовое исключающее ИЛИ(bit-wise exclusive or, xor)	Не допустимо
^~ ~^	Побитовая эквивалентность (bit-wise equivalence, xnor)	Не допустимо
&	Редукционное И (reduction and)	Не допустимо
~&	Редукционное НЕ-И (reduction nand)	Не допустимо
|	Редукционное ИЛИ (reduction or)	Не допустимо
~|	Редукционное НЕ-ИЛИ (reduction nor)	Не допустимо
^	Редукционное исключающее ИЛИ (reduction xor)	Не допустимо
~^ ^~	Редукционное НЕ исключающее ИЛИ (reduction xnor)	Не допустимо
<<	Сдвиг влево (left shift)	Не допустимо
>>	Сдвиг вправо (right shift)	Не допустимо
<<<[2]	Циклический сдвиг влево (arithmetic left shift)	Не допустимо
>>>	Циклический сдвиг вправо (arithmetic right shift)	Не допустимо
?:	Условный оператор (conditional)	Допустимо

4.2.1 Приоритет бинарных операций

Приоритет бинарных операций (а также оператора ?:) подчиняется тем же правилам, что и приоритет аналогичных операций в языке C. Verilog HDL содержит также несколько операций не представленных в языке C. Ниже представлен список операторов Verilog, упорядоченных по их приритету.

+ - ! ~ (унарные) Наивысший приоритет
*** / %**
+ - (бинарные)
<< >> <<< >>>
< <= > >=
== != === !===
& ~&
^ ^~ ~^
| ~|
&&
||
?: Нижайший приоритет

Операторы находящиеся в одной строке имеют равный приоритет, строки упорядочены от наивысшего приоритета к наименьшему. Все операторы с равным приоритетом выполняются в порядке слева-направо, исключение составляет операторы ?: которые выполняются справа-налево.

Пример: A + B – C В этом примере значение B прибавляется к A, затем C вычитается из результата.

Когда операторы имеют различный приоритет, сначала выполняются операторы с наивысшим приоритетом, а затем с меньшим.

Пример: A + B / C В этом примере значение B будет поделено на значение C и результат будет добавлен к A.

Приоритет выполнения операций может быть изменен с помощью круглых скобок.

[2] Операции циклического (арифметического) сдвига определены в стандарте 2000г. Не все системы моделирования и синтеза поддерживают эти операции.

Пример: (A + B) / C Рерультат будет совершенно другой в отличие от предыдущего примера. Сначала будет вычислена сумма A и B и лишь затем результат будет поделен на значение C.

4.2.2 Знаковые и беззнаковые числа в выражениях

Операнды могут выражаться как безразмерные числа, или они могут иметь размер. В выражениях Verilog интерпретирует числа в виде sss'f nnn (размерные числа, sss – размер числа в битах) как беззнаковые, запись в такую переменную числа со знаком приведет к тому, что в выражении будет использовано не отрицательное число, а дополнительный код представленный уже как целое положительное число (см. пример в п. 3.5). Еще один пример приведен ниже, он показывает, что аналогичная ситуация может возникнуть и в случае использования переменной типа ***integer***.

```
integer Inta;
Inta = -12 / 3;       // Результат равен -4
Inta = -'d12 / 3;     // Результат равен 1431655761
```

Не будем забывать, что по умолчанию целое число имеет 32 разряда.

4.2.3 Арифметические операции

К бинарным арифметическим относятся **+ - * / %**. Операция деления отсекает от целого числа дробную часть. Для получения остатка используется операция **%**. Эта операция воспринимает знак первого операнда. В следующих примерах представлены различные результаты этой операции.

Операция	**Результат**	**Комментарий**
10 % 3	1	10/3 дает остаток 1
11 % 3	2	11/3 дает остаток 2
12 % 3	0	12/3 не дает остатка
-10 % 3	-1	Результат совпадает со знаком первого операнда
11 % -3	2	Знак второго операнда игнорируется
-4'd12 % 3	1	-4'd12 воспринимается как целое положительное число с остатком 1

Унарные арифметические операции (+, -) имеют приоритет перед бинарными. Если один из операндов в арифметических операциях имеет неопределенное значение, то и результат операции будет неопределен (x).

4.2.4 Операции отношения

Следующие примеры иллюстрируют операторы отношения:

```
a < b      a меньше чем b;
a > b      a больше чем b;
a <= b     a меньше или равно b;
a >= b     a больше или равно b;
```

Все эти выражения возвращают лог.0 если приведенное отношение ложно (false) или лог.1 если отношение истинно (true). Если один из операндов имеет неопределенное значение, то и результат будет неопределен. Все операторы отношения имеют одинаковый приоритет и более низкий, чем приоритет арифметических операторов. Следующий пример иллюстрирует смысл этого правила:

```
a < size -1          // Эта конструкция идентична
a < (size -1)        // этой, но
size – (1 < a)       // эта конструкция отличается
size – 1 < a         // от этой
```

Заметим, что в кострукции size – (1 < a) операция отношения вычисляется первой, а затем результат (0 или 1) вычитается из переменной size. В следующем же выражении сначала size уменьшается на 1, а затем результат сравнивается с a.

4.2.5 Операторы сравнения

Операторы сравнения имеют более низкий приоритет, чем операторы сравнения. В следующем примере иллюстрируются операторы сравнения.

```
a === b          // a равно b, включая x и z
a !== b          // a не равно b, включая x и z
a == b           // a равно b, результат может быть неизвестен
a != b           // a не равно b, результат может быть неизвестен
```

Все четыре оператора имеют одинаковый приоритет. Они сравнивают операнды бит в бит, заполнением нулями в случае если операнды имеют различную разрядность. Так же как и операторы отношения они возвращают 0 если false и 1 если true. Если хотя бы один операнд содержит x или z, то операции == и != возвращают неопределенное значение. Операции === и !== сравнивают операнды с учетом x и z, поэтому всегда возвращают либо 0, либо 1.

Замечание:

Некоторые системы синтеза могут выдавать ошибку на операции === и !==.

4.2.6 Логические операции

Операторы логического И (**&&**) и логического ИЛИ (||) тесно связаны между собой. Результат логического сравнения может принимать значение «истинно» (true, 1) или «ложно» (false, 0), или если один из операндов имеет неопределенное значение, то и результат будет неопределен. Допустим, один из операндов имеет значение равное 10, а второй равен нулю. Результат логического И будет равен 0, так как второй операнд равен нулю. Результат логического ИЛИ, однако будет равен 1, так как первый операнд отличен от нуля. Verilog HDL воспринимает операнд равный 0 как «ложный», в то же время если операнд не равен нулю (не обязательно равен 1, например 10), то он воспринимается как «истинный». Именно это и произошло в приведенном примере: операция (true **&&** false) привела к результату false, а операция (true || false) привела к результату true.

Приоритет операции **&&** больше чем приоритет ||, но оба они меньше, чем приритет операций отношения и сравнения. Рассмотрим следующий пример:

a < size – 1 && b != c && index != lastone

Три суб-выражения объединены по логическому И без использования круглых скобок. Хотя такая конструкция вполне допустима, применять ее не следует из-за ухудшения читабельности кода. Следующий пример читается гораздо легче:

(a < size – 1) && (b != c) && (index != lastone)

Третьим логическим оператором является унарный оператор логического отрицания **!**. Оператор логического отрицания преобразует ненулевое («истинное») значение в 0 («ложно»), а нулевое («ложное») в 1(«истинно»). Неопределенное значение приведет к неопределенному результату. Наиболее часто этот оператор используется в следующей конструкции:

if (**!**inword) эта запись эквивалентна записи if (inword == 0)

4.2.7 Побитовые логические операции

Побитовые операции выполняют манипуляции с операндами бит в бит, т.е. например 0-й бит первого операнда взаимодействует с 0-м битом второго; 1-й бит с 1-м и т.д. Если один из операндов имеет меньшую разрядность, то недостающие биты заполняются нулями. В следующих таблицах представлены результаты побитовых операций – значения результирующих битов в зависимости от битов операндов.

Табл. 4-2 Побитовое отрицание (~ - унарная операция)

0	1
1	0
x	x

Табл. 4-3 Побитовое И (**&**)

	0	1	x
0	0	0	0
1	0	1	x
x	0	x	x

Табл. 4-4 Побитовое ИЛИ (|)

	0	1	x
0	0	1	x
1	1	1	1
x	x	1	x

Табл. 4-5 Побитовое исключающее ИЛИ (^)

	0	1	x
0	0	1	x
1	1	0	x
x	x	x	x

Табл. 4-6 Побитовое исключающее НЕ ИЛИ (~^ ^~)

	0	1	x
0	1	0	x
1	0	1	x
x	x	x	x

Следует обратить внимание, что операторы **&** и **&&**; | и || различаются принципиально. Операторы **&&** и || обычно используются в конструкциях с условиями (if, например), а **&** и | для маскирования отдельных битов переменной.

4.2.8 Операторы редукции

Все операторы редукции являются унарными, и результатом всех операций является одноразрядное значение – 0, 1 или x. Алгоритм работы всех операторов следующий: на первом шаге вычисляется значение результата операции между 0-м и 1-м битом операнда, на следующем шаге вычисляется результат между предыдущим результатом и 2-м битом операнда и т.д. пока не будет вычислен результат по всем битам. На каждом шаге результат определяется по таблицам истинности 4-3 – 4-6. Необходимо заметить, что унарные операторы редукции NAND и NOR вычисляются точно также как операторы AND и OR, а полученный результат инвертируется.

В таблице 4-7 приведены результаты унарных операций **&**, |, **~&** и ~| различных комбинаций битов операнда. В таблице 4-8 приведены результаты операций ^ и ~^.

Табл. 4-7 Результаты унарных операций **&**, |, **~&** и ~|

Разряды операнда	**&**	**|**	**~&**	**~|**
Все в лог.0	0	0	1	1
Все в лог.1	1	1	0	0
Часть в лог.0, часть в лог.1	0	1	1	0

Табл. 4-8 Результаты унарных операций ^ и ~^

Разряды операнда	**^**	**~^**
Нечетное число бит в лог.1	1	0
Четное число бит в лог.1 (или ни одного)	0	1

4.2.9 Синтаксические ограничения

В языке Verilog накладываются два ограничения для того чтобы защитить файлы описания от типографской ошибки, которые крайне тяжелы для обнаружения. Ошибка заключается во вставке ошибочного пробела. В следующем примере первая строка не соответствует второй:

1. a & &b; a | |b;
2. a && b; a || b;

В первой строке введен дополнительный символ пробела между символами & и |. Это может быть замысел разработчика, а может ошибочный ввод пробела. Для того,чтобы предотвратить подобную неоднозначность Verilog требует разделить бинарную и унарную операцию круглыми скобками. В силу этого требования первая строка примера является ошибочной. Поэтому правильно надо записывать так:

a & (&b); a | (|b);

4.2.10 Операторы сдвига

К операторам сдвига относятся операторы <<, >>, <<< и >>>. Первые два оператора выполняют простой сдвиг влево или вправо на количество позиций указанных операндом в правой части выражения, два других выполняют циклический или иначе арифметический сдвиг. В следующем примере иллюстрируется механизм действия этих операторов.

Выражение	**Результат**
8'b01001111 << 3	8'b01111000
8'b01001111 <<< 3	8'b01111010
8'b01001111 >> 3	8'b00001001
8'b01001111 >>> 3	8'b11101001

Простые операторы сдвига заполняют освободившиеся позиции нулевыми значениями, циклические операторы сдвига выполняют сдвиг «по кольцу».

4.2.11 Условный оператор

Условный оператор имеет три операнда, разделенные двумя операторами в следующем формате:

cond_expr **?** true_expr **:** false_expr;

Если условие (cond_expr) вычисляется как «ложное» (false), то в качестве результата будет использовано выражение false_expr. Если условие «истинно» (true), то будет использовано выражение true_expr. В случае если условие неопределенно, то оба выражения false_expr и true_expr сравниваются бит в бит, по правилам указанным в табл. 4-9, для того,чтобы вычислить окончательный результат. Если разрядность операндов различна, то наименьший из двух приводится к разрядности большего, при этом недостающие биты добавляются слева и заполняются нулями.

Табл. 4-9 Результат неопределенного условия

?:	0	1	x	z
0	0	x	x	x
1	x	1	x	x
x	x	x	x	x
z	x	x	x	x

Следующий пример иллюстрирует типичное применение условного оператора для реализации шины с тремя состояниями:

wire [15:0] busa = drive_busa **?** data **:** 16'bz;

Если drive_busa равен лог.1 то на шине busa устанавливается значение data. Если drive_busa неизвестен, то неизвестное значение устанавливается и на шине busa. В случае если drive_busa находится в состоянии лог.0, то шина busa находится в состоянии высокого импеданса.

4.2.12 Конкатенация

Операция конкатенации является одной из наиболее удобных и мощных операций в языке Verilog. Суть ее заключается в слиянии нескольких переменных в единое целое, единую переменную, с которой можно производить любые другие операции. Необходимо отметить два момента: операция конкатенации обладает наивысшим приоритетом по сравнению с любой другой операцией вне символов конкатенации (**{}**), но операции заключенные внутри фигурных скобок имеют еще больший приоритет (Операции внутри фигурных скобок это недокументированное свойство языка, главное в этом случае, чтобы в результате внутренней операции результат получил определенную разрядность). Вторым моментом является тот факт, что операция конкатенации недопустима с вещественными числами. Синтаксис операции приведен ниже:

{<expr_1>, <expr_2>, … <expr_n>};

Операция может содержать несколко повторяющихся элементов, для сокращения записи используется множитель, который указывает сколько раз повторяется данный элемент:

{4{w}} эквивалентно {w, w, w, w}

Множитель может быть целой, неотрицатьной константой или константным выражением.

Также в операции могут использоваться внутренние объединения:

{{a, b, c}, {3{d, e}}} эквивалентно {a, b, c, d, e, d, e, d, e}

Результат операции слияния может использоваться в любом случае в качестве операндов или в качестве вектора (переменной) которой присваивается значение. Это широко используется для случаев, когда функция должна вернуть несколько значений.

4.3 Операнды

4.3.1 Битовая адресация цепей и регистров

Битовая адресация позволяет использовать в качестве операнда отдельный бит многоразрядной цепи (в дальнейшем «шины») или регистра. При этом индекс (адрес) бита может быть выражением (результат этого выражения должен быть, разумеется, положительным целым числом, включая и 0). Индекс в данном случае выступает как операнд для адресации конкретного бита.

acc[index] - выбирается бит регистра (шины) acc на который указывает index;

Следует обратить внимание на возрастание/убывание диапазона регистра (шины). Ниже приведен простой пример:

```
reg [7:0] acc1;
reg [0:7] acc2;
acc1 = 8'b10000000;
```

acc2 = 8'b10000000;

- acc1[0] вернет значение 0 (lsb);
- acc2[0] вернет значение 1 (msb);

Если значение индекса неопределенно (а также он в z-состоянии) или индекс выходит за рамки диапазона, то выражение вернет неопределенное значение.

Замечание:

Некоторые системы синтеза могут выдавать ошибку в случае выхода индекса за границу диапазона, хотя по стандарту это вполне допустимая операция. В каждом конкретном случае необходимо строго следить, чтобы такая ситуация не сложилась. Кроме того, если система синтеза игнорирует эту ошибку, то вполне возможен «подводный камень», связанный с изменением разрядности шины - разработчик уменьшил разрядность шины, а диапазон индексов не изменил, что может привести уже к логической ошибке (синтаксически эта ситуация не диагностируется!).

Операция адресации может применяться не только к отдельным битам, но и к группе последовательных битов регистра или шины (групповая адресация или part-select). Группа битов выбирается с помощью следующей синтаксической конструкции:

reg_bus_name [msb_expr : lsb_expr];

В данном случае **reg_bus_nam**e – имя регистра или шины, а **msb_expr** и **lsb_expr** старший и младший значащий бит группы соответственно. Оба эти выражения должны быть константными и возвращать целое положительное число. В этой операции также имеет значение возрастание и убывание диапазона. Если регистр (шина) объявлены с возрастанием диапазона, то и групповая адресация должна быть с возрастанием диапазона и наоборот. Несоблюдение этого требования приводит к синтаксическим ошибкам.

Совет:

В каждом конкретном проекте следует придерживаться простого правила: если вы начали объявлять шины и регистры в убывающем диапазоне то так и продолжайте, это избавит вас и ваших коллег, от большой путаницы в дальнейшем.

4.3.2 Адресация элементов памяти

Ранее уже обсуждался вопрос объявления массивов. В этом пункте обсуждается адресация к элементам массива (элементам памяти). В следующем примере объявляется массив mem, который содержит 1024 байта:

reg [7:0] mem [0:1023];

Для того, чтобы обратиться к отдельному байту используется следующий синтаксис:

mem[addr_expr]; где **addr_expr** – любое выражение, возвращающее целое, положительное число.

Обращение к группе байтов запрещено, так же как запрещено напрямую обратиться к отдельному биту элемента памяти (см. листинг 3-2 в п.3.4). Возможна ситуация представленная в следующем примере:

```
reg [31:0] reg_name;          // Объявление 32-х разрядного регистра reg_name;
reg mem_name [31:0];          // Объявление массива из 32-х бит;
reg_name[8];                  // Адресуемся к 8-у биту регистра;
mem_name[8];                  // Адресуемся к 8-у элементу массива;
reg_name[5:0];                // Адресуемся к группе битов регистра;
mem_name[5:0];                // Получаем ошибку!
```

Совет:

Для того, чтобы избежать подобной ситуации существуют два правила. Первое: необходимо писать самодокументированный код, имена переменных должны отражать их назначение. Второе: необходимо избегать битовых массивов, слишком легко спутать с регистром, системы синтеза выделят равное количество ресурсов как для регистра `reg_name`*, так и для массива* `mem_name` *в приведенном примере.*

Еще один маленький штрих: в качестве индекса можно использовать элемент этого-же массива – косвенная адресация. Пример:

```
mem_name [mem_name[3]];
```

В объявлении массива возрастание/убывание диапазона элементов массива игнорируется, а в диапазоне разрядности элемента – нет. Если индекс (адрес элемента массива неопределен, находиться в z-состоянии или выходит за рамки диапазона, то операция адресации вернет неопределенное состояние.

4.3.3 Строки

Строковые операнды воспринимаются как константные целые числа, состоящие из последовательности 8-разрядных ASCII-кодов, один байт на символ. Любой оператор Verilog может манипулировать строковыми операндами как единичным цифровым значением:

```
reg [8*14:1] stringvar;
stringvar = “Hello world”;        // Выполнена инициализация переменной stringvar
```

Переменная будет сохранена в виде числа

```
00_00_00_48_65_6c_6c_6f_20_77_6f_72_6c_64
```

Для строковых переменных допустима операция конкатенации:

```
stringvar = {stringvar, “!!!”};
```

Результат будет следующим:

48_65_6c_6c_6f_20_77_6f_72_6c_64_21_21_21 Hello world!!!

Еще одной операцией является операция сравнения (==, **!=**), которая позволяет сравнивать две строки. За исключением этих трех операций (присваивания(=), конкатенации (**{}**) и сравнения, все остальные операции особой пользы не приносят, хотя ошибок не вызовут (точнее, если и вызовут, то не потому, что переменная строковая). Со операцией слияния связан один «подводный камень», он иллюстрируется следующим примером:

reg [8*10:1] string1, string2;
string1 = “Hello”;
string2 = “world!”;

if ({string1, “ “, string2} == “Hello world!”) **$display**[3](“Ok!”);

В этом примере “Ok!” получено не будет, поскольку операция слияния также воспринимает строковую переменную как целое число определенной разрядности. В левой части оператора сравнения окажется:

00_00_00_00_00_48_65_6c_6c_6f_20_00_00_00_00_77_6f_72_6c_64_21

Произошло слияние трех строковых пременных, причем две из них имели разрядность 8*10 = 80 бит, в целом число получилось из 168 бит (неинициализированные разряды – нули, тоже вошли в результат). Строка “Hello world!” занимает 8*12 = 96 бит. Разумеется эти строки никогда не совпадут.

Строковая переменная может быть пустой “”, что эквивалентно нулевому значению.

[3] $display – системная функция вывода сообщений на консоль (см. раздел «Системные функции и задачи»)

5. Иерархические структуры

5.1 Понятие иерархических структур

Verilog HDL поддерживает иерархическое описание проекта. На самом верхнем уровне лежит модуль, содержащий экземпляры модулей следующего, более низкого, уровня иерархии, а также связи между ними. Модули более низкого уровня в свою очередь тоже могут содержать экземпляры модулей следующего уровня и т.д. По стандарту ограничений на количество уровней нет, но количество уровней может быть ограничено системой синтеза. К иерархическим объектам языка относятся:

- Модули (modules)
- Макромодули(macromodules)
- Встроенные примитивы (primitives)
- Примитивы определяемые пользователем (user defined primitives)

5.2 Модули

5.2.1 Формальный синтаксис

Объявление модуля заключено между двумя ключевыми словами ***module*** и ***endmodule***. Синтаксис приведен ниже:

module <module_name> [<list_of_ports>];
    <module_items>
endmodule

Имя модуля **module_name** является обязательным, это имя под которым модуль представлен в описании. Список портов **list_of_port** является опциональным, он определяет порядок портов при создании экземпляра модуля (иногда говорят экземпляра компонента). Список чаще всего опускается в testbench-модулях, предназначенных для моделирования какого-либо объекта – другого модуля, в этом случае testbench-модуль выступает как модуль высшего уровня иерархии.

Объекты модуля **module_items** перечислены ниже:

- Объявления параметров (parameter declaration) – всегда следуют за списком портов.
- Объявления входных портов (input declaration).
- Объявления выходных портов (output declaration).
- Объявления двунаправленных портов (inout declaration).
- Объявления цепей (net declaration).
- Объявления регистров (reg declaration).
- Объявления переменных типа ***time*** (time declaration).
- Объявления переменных типа ***integer*** (integer declaration).
- Объявления переменных типа ***real*** (real declaration).
- Объявления переменных типа ***event*** (event declaration).
- Экземпляры примитивов (primitive instantiation).
- Экземпляры модулей (module instantiation).
- Переопределение параметров (parameter override).

- Непрерывные присвоения значений переменным (continious assignments).
- Specify блоки (specify blocks).
- Установки начальных состояний (initial statements).
- Процесс-блоки (always blocks).
- Объявление задач (процедур, tasks).
- Объявление функций (functions).

Объявления парааметров, цепей, регистров, переменных типа ***time***, ***integer***, ***real*** были приведены в разделе 3 **Типы данных**. Объявления переменных типа ***event*** и их применение является темой для отдельного раздела. В следующих пунктах будут разобраны объявления портов, создание экземпляров модулей и примитивов, specify блоки, блоки установки начальных состояний, процесс-блоки, объявления функций и задач.

5.2.2 Объявления портов

Порты модуля объявляются с помощью следующих ключевых слов ***input***, ***output***, ***inout***, для того чтобы определить какие порты являются входными, какие выходными, какие двунаправленными, а также разрядность портов. Кроме того, если не указан список портов при объявлении модуля, то эти объявления определяют порядок портов при создании экземпляра этого компонента. Для testbench-модулей входные и выходные порты не объявляются, поскольку эти модули не синтезируются. Для синтезируемых модулей обязательно должен быть объявлен хотя бы один выходной порт. Общий синтаксис объявления портов приведен ниже:

<port_direction> [<port_range>] <port_name>, [<port_name>]….;

Направление порта **port_direction** определяется ключевыми словами ***input***, ***output***, ***inout***. Опциональное поле **port_range** определяется точно также, как диапазон для регистровых переменных и цепей (ключевые слова ***vectored*** и ***scalared*** не применяются). Имена портов ***port_name*** представляют собой идентификаторы, они также должны быть в списке портов. Пример объявления модуля и его портов приведен в листинге 5-1.

Листинг5-1:

```
module dreg(DIN, DOUT, CE, RST, CLK);
  parameter NBRB = 8;
  input [NBRB – 1:0] DIN;
  input CE, RST, CLK;
  output [NBRB – 1:0 ] DOUT;
  reg [NBRB – 1:0 ] DOUT;
  ----
  <dreg statements>
  ----
endmodule
```

В этом примере объявляется модуль dreg (data register) со входными портами DIN, CE, RST, CLK и с одним выходным портом DOUT. Кроме того, объявлен параметр NBRB (number of bit), инициализированный значением 8, что определяет разрядность входной и выходной шины DIN и DOUT. Следует обратить внимание, что шина DOUT

объявлена дважды: как выходной порт и как регистр. Это необходимо, чтобы указать, что выходной порт DOUT является выходом регистровой переменной DOUT (т.е. DOUT является регистром, не цепью). Для выходов цепей переопределение не нужно, состояние этих портов определяется комбинационной схемой.

Совет:

Идентификаторы портов, параметров лучше всего составлять из прописных букв, имена внутренних переменных можно прописывать строчными буквами, но начинать имя с прописной. Ключевые слова в языке `Verilog` *всегда прописываются строчными буквами. Это не требование синтаксиса по отношению к портам и переменным, но позволяет в любом месте кода знать, с чем мы имеем дело: с портом, переменной, ключевым словом. Не надо забывать, что* `Verilog` *различает строчные и прописные буквы.*

5.2.3 Создание экземпляров модулей

Модули являются основной иерархической единицей Verilog HDL и поэтому объявление одного модуля внутри другого не допускается. Однако, можно создать экземпляр (instance) другого модуля который был объявлен в другом месте. Это и обеспечивает иерархическую структуру проекта. Модуль для которого не существует экземпляра в каком-либо другом модуле называется модулем верхнего уровня иерархии (top-level module). Очень часто это testbench-модули. Формальный синтаксис экземпляра модуля приведен ниже:

<module_name> [<parameter_value_assignment>] <module_instances>;

Здесь **module_name** обозначает имя модуля экземпляр которого создается. Опциональное поле **parameter_value_assignment** предназначено для модификации параметров модуля. Синтаксис этого поля выглядит следующим образом:

parameter_value_assignment: #(<expr_1>, <expr_2>, … <expr_n>)

Поле **module_instances** содержит список экзампляров модуля **module_name** с установленными параметрами. Синтаксис этого поля выглядит так:

module_instances:
<inst_name> (<list_of_connection>), … <inst_name> (<list_of_connection>)

Имя экземпляра **inst_name** должно быть уникальным в пределах модуля. Особого внимания требует список соединений – **list_of_connection**. В простейшем случае, список состоит из имен цепей, следующих друг за другом в порядке, определенном порядком портов модуля. Инена соединительных цепей (не регистров!) разделяются запятыми. Этот вариант списка соединений с позиционным соответствием – цепь подключается к тому порту, который находится на той же порядковой позиции. Хотя это и наиболее распространенный метод, но не всегда он удобен. Другим вариантом является ключевое соответствие: для каждой цепи прямо указывается порт к которому цепь должна быть подключена. Для этого используется простая конструкция:

.<port_name> (<net_name>) – следует обратить внимание на точку в начале записи.

В листинге 5-2 приведен пример создания трех экземпляров компонента dreg (см. лист. 5-1)

Листинг5-2:

```
module reg_bank(DIN, DREG1, DREG2, DREG3, RG1, RG2, RG3, RST, CLK);
  input [7:0] DIN;
  input RG1, RG2, RG3, RST, CLK;
  output [7:0] DREG1;
  output [5:0] DREG2;
  output [3:0] DREG3;

  dreg Register_1 (DIN, DREG1, RG1, RST, CLK);
  dreg #( 6) Register_2 (DIN[5:0], DREG2, RG2, RST, CLK);
  dreg #( 4) Register_3
    (.CLK(CLK), .RST(RST), .DIN(DIN[3:0]), .DOUT(DREG3), .CE(RG3));
endmodule
```

Из этого примера можно увидеть следующие моменты:

- Register_1 создается с параметром NBRB по умолчанию – 8, что определяет 8 разрядов выходного порта DOUT подключенного к шине DREG1. Используется позиционное соответствие цепей и портов модуля dreg.
- При создании Register_2 параметр NBRB был переопределен, он стал равен 6, и это определило 6-и разрядный порт DOUT модуля dreg. Следует обратить внимание, что имя параметра не указывается, т.е. параметры переопределяются только по принципу **позиционного соответствия**. Также используется позиционное соответствие при определении списка соединений. Интересным моментом является тот факт, что используются не все разряды шины DIN, а только первые шесть разрядов. Можно не указывать группу битов подключенных к этому порту, но во-первых это наверняка вызовет предупреждение системы синтеза, а во вторых к порту будут подключены разрады с 0-го по 5-й, что в данном случае и требовалось. При других условиях этот вариант не приемлем.
- При создании Register_3 также был переопределен параметр NBRB – ничего нового тут нет. В отличие от предыдущего варианта в списке подключений используется ключевое соответствие. На первый взгляд это более тажеловесная конструкция, но она дает два важных преимущества: первое – порядок подключения цепей не завмсит от порядка следования портов,а портов у модуля может быть очень много, могут добавляться новые порты или удаляться ненужные. В этом случае мы застрахованы от ошибок связанных с порядком следования портов модуля, а эта ситуация практически не распознается системой синтеза, но наверняка приводит к логическим ошибкам. Для примера можно в Register_1 поменять местами цепи RST и RG1 – система синтеза не выдаст даже предупреждения, но схема окажется абсолютно неработоспособной. Второе преимущество: при позиционном соответствии нельзя из списка пропустить хотя бы один сигнал, даже если в этом экземпляре этот **выход** (пропуск входного сигнала не допускается) не нужен. В случае ключевого соответствия этой проблемы нет.

Еще одно маленькое дополнение: если входной сигнал не должен изменяться в процессе работы схемы, пропустить мы его не можем, но можем установить фиксированное значение (0 или 1) непосредственно в списке соединений. Это может вызвать предупреждение системы синтеза, но в данном случае с ним придется мирится.

Есть еще один механизм для переопределения параметров. В этом случае используется выражение ***defparam***. В листинге 5-3 приведен пример который позволяет переопределить параметр NBRB в Register_1 и Register_2.

Листинг5-3:

```
module annotate;
  defparam
    reg_bank.Register_1.NBRB = 6,
    reg_bank.Register_2.NBRB = 4;
endmodule
```

Использование такой конструкции перопределило параметры регистров, сделав их 6 и 4 соответственно. Модуль annotate включается в проект на равных правах с другими модулями, но имеет отличие: у него нет ни входных ни выходных портов, нет списка портов. Тем не менее в просессе синтеза он участвует. Преимуществом такого метода является то, что он не затрагивает те модули в которых должны измениться параметры, он может быть введен в тестовых целях, а затем исключен из проекта. Недостатком же является тот факт, что меняя параметры, определяющие разрядность портов он не может изменить разрядность шин, которые к этим портам подключены. Добавление такого модуля как в лист. 5-3 к проекту вызовет предупреждения системы синтеза по поводу несоответствия разрядности портов и подключенных к ним шин. Этот случай можно обойти вводом дополнительных параметров в модуль reg_bank, которые позволят изменить разрядность шин подключенных к портам и переопределить их в модуле annotate. Однако в этом случае придется вносить изменения в самом модуле reg_bank, что как раз и нежелательно. В других случаях, например при переопределении временных задержек, никаких проблем не возникнет.

5.2.4 Создание экземпляров примитивов

Более подробное изложение того, что такое примитивы будет представлено в параграфе **5.4 Примитивы**. В данном пункте будет изложено краткое описание, того, что понимается под примитивом и создание экземпляра примитива в модуле.

Под понятием «Примитив» понимается отдельная иерархическая единица, аналогичная модулю в том отношении, что на любом уровне иерархии она занимает равные права с модулем. В отличие от модулей на примитивы накладывается ряд ограничений. Основные из них два: у примитива не может быть более одного выхода, причем одноразрядного. Второе: для примитивов не определены параметры. Также при создании экземпляра невозможно использовать ключевое соответствие, только позиционное. Но есть и небольшое преимущество: на экземпляр примитива может быть определена задержка, в то время как в модуле задержки определяются в теле модуля (константами или параметрами). При создании экземпляра примитива не обязательно указывать собственное имя экземпляра. Общий синтаксис при создании экземпляра следующий:

<primitive_name> [<delay>] [<inst_name>] <list_of_connection>;

Как и вслучае с модулем, можно определить сразу несколько примитивов (т.е. [<inst_name>] <list_of_connection>, [<inst_name>] <list_of_connection>;). Примитивы делятся на две группы: встроенные примитивы языка и примитивы определяемые пользователем. Нет никакой разницы при создании экземпляра примитива из той или другой группы. В листинге 5-4 приведен пример в котором были созданы три примитива реализующие функцию Y = (A and B) or (C and D).

Листинг5-4:

```
module comb_block(A, B, C, D, Y);
  input A, B, C, D;
  output Y;
  wire x1, x2;
  and (x1, A, B), (x2, C, D);
  or #5 (Y, x1, x2);
endmodule
```

В примере было реализовано два примитива ***and*** и один ***or***. Как видно, примитивы как правило реализуют простые логические функции. В силуэтого обстоятельства, в некоторых источниках, подобные примитивы часто называют вентилями, что близко к их истинному назначению и реализации. В примере есть один момент, который никогда раньше не рассматривался: а именно выражение #5. В предыдущем пункте аналогично переопределялся параметр (см. лист. 5-2). В данном случае это совершенно другая операция – установка задержки на примитиве. Это выражение воспринимается как задержка на 5 временных единиц (что такое временная единица, где и как она определяется будет рассказано в разделе «**Директивы компилятора**». Необходимо отметить следующее: временные задержки играют роль только в процессе моделирования, для того чтобы получить более реальную картину событий в реальном устройстве. Системы синтеза всегда игнорируют подобные выражения, задержки в реальном устройстве будут другими, определенными в процессе имплементации.

Существует несколько правил по которым прописывается список соединений примитива, но главное из них одно – выход примитива (вентиля) всегда стоит на первом месте, все остальные порты являются входами. Ни один порт не может быть двунаправленным (есть, правда, одно исключение, но это не синтезируемый примитив). В примере все примитивы безымянные, логические функции, которые они реализуют слишком тривиальны и определяются названием примитива.

Совет:

Не стоит давать собственные имена экземплярам встроенных примитивов, чтобы не засорять пространство имен. Для примитивов определенных пользователем имена давать можно для улучшения читабельности кода.

5.2.5 Процесс-блоки (always-блоки)

Блоки подобного типа здесь названы в силу того, что они реализуют некий процесс, который может повторятся многократно. В литературе их часто называют always-блоками по ключевому слову, определяющему такой блок. Вкратце можно сказать следующее – в подобном блоке всегда есть некоторое выражение, в правой части которого стоит переменная, блок выполняет процесс **всегда** (always), когда эта переменная изменяет свое значение или происходит какое-либо событие (event). Формальный синтаксис always-блока приведен ниже.

always [<event_control>]
[***begin*** | ***fork***] [: <block_name>]
<block_statements>;
[***end*** | ***join***]

Опциональное поле ***event_control*** содержит т.н. список чувствительноси блока, т.е. список событий (events) при возникновении которых блок начнет выполняться. Если список отсутствует, он создается автоматически по именам переменных в правой части выражений ***block_statements***.

Замечание:
Некоторые системы синтеза могут выдавать предупреждения на подобную ситуацию. Это не критично, но лучше все же заполнить это поле, чтобы не засорять итоговый отчет по результатам синтеза.

Если в блоке только одно выражение, то ключевые слова ***begin*** – ***end*** и ***fork*** – **join** можно опустить (хотя и оставить их вполне можно). Какую именно пару можно применять и когда будет подробно рассказано в разделе «**Поведенческое описание**». Чаще всего применяется пара ***begin*** – ***end***, что определяет блок с последовательным выполнением операторов. Опциональное поле **block_name** позволяет присвоить блокам ***begin*** – ***end*** и ***fork*** – **join** собственные имена, что иногда бывает полезно (см. тот же раздел).

Наиболее важным в описании always-блоков занимает список чувствительности. Формальный синтаксис списка чувствительноси следующий:

event_control: @ (<event_1> **or** <event_2> **or** … <event_n>)

в более позднем стандарте языка Verilog (2000 г.) предусмотрена и длугая форма:

event_control: @ (<event_1> **,** <event_2> **,** … <event_n>)

События event_1 – event_n представляют собой неявную форму переменной типа ***event***. Для описания этих событий используются два ключевых слова ***posedge*** и ***negedge*** для того, чтобы указать с каким фронтом сигнала (переменной) данное событие связано: с возрастающим (из лог.0 в лог.1 – ***posedge***) или спадающим (из лог.1 в лог.0 – ***negedge***). Эти ключевые слова можно не указывать, в этом случае событием станет любое изменение переменной.

Замечание:
Возрастающий фронт может определяться и как переход переменной из неопределенного состояния в состояние лог.1, а спадающий - как переход из лог.1 в неопределенное состояние.

Ключевые слова ***posedge*** и ***negedge*** всегда применяются при создании регистровой логики для фиксации фронта или среза тактового сигнала. Пример списка чувствительности для регистра dreg приведен ниже:

@ (***posedge*** CLK ***or posedge*** RST)

В данном случае регистр будет фиксировать фронт тактового сигнала CLK, и фронт сигнала сброса RST (хотя это и не тактовый сигнал, подобная конструкция вполне допустима). В листинге 5-5 приведен полный код регистра dreg, усеченная версия которого была приведена в листинге 5-1.

Листинг5-5:

```
module dreg(DIN, DOUT, CE, RST, CLK);
  parameter NBRB = 8;
  input [NBRB – 1:0] DIN;
  input CE, RST, CLK;
  output [NBRB – 1:0 ] DOUT;
  reg [NBRB – 1:0 ] DOUT;
  always @(posedge CLK or posedge RST)
    begin
      if (RST == 1) DOUT <= 'b0;
      else
        if (CE == 1) DOUT <= DIN;
    end
endmodule
```

В примере применен оператор **if** в несколько упрощенной форме – в этой форме он полностью аналогичен оператору if в языке C. Запись алгоритма в такой форме называется поведенческим описанием. В разделе «***Поведенческое описание***» будет подробно рассмотрен синтаксис операторов if, case и операторов цикла. В примере есть еще два момента на которые следует обратить внимание: это начальная инициализация регистра DOUT без указания его разрядности 'b0 – разрядность однозначно определяется параметром NBRB; и новый оператор <=. Этот оператор носит название внеблочного назначения (оператор = это блочное назначение или присваивание). Смысл этого оператора заключается в том, что значение переменной будет присвоено только в момент выхода из блока. Эти операторы не должны повторятся для одной и той же переменной в одной ветви алгоритма, чтобы избежать неопределенного состояния. Оператор = выполняется в тот же момент где он встретился и поэтому может многократно повторятся в одной и той же ветви. Ветвью алгоритма называется путь по которому алгоритм может завершится. Попав на одну ветвь процесс не может перейти на другую параллельную ветвь. Такими параллельными ветвями в алгоритме стали выражения под первым оператором **if** и выражения под оператором **else**. Одновременно эти две ветви не будут выполнены никогда, поэтому конфликта не будет.

Простейший пример использования always-блока выглядит следующим образом:

always #half_period CLK = ~CLK;

Эта запись предназначена для моделирования тактового сигнала CLK с периодом 2 * half_period. Эта конструкция не синтезируема, поскольку системы синтеза игнорируют временную задержку, но она широко применяется в testbench-модулях. В общем случае always-блоки применяются для реализации регистровой логики: регистров, счетчиков, конечных автоматов, но могут использоваться для реализации и комбинационной логики: мультиплексоров, дешифраторов, перемножителей и т.д. Практически все синтезируемые конструкции в Verilog HDL так или иначе используют эти блоки. В рамках одного модуля количество блоков не ограничивается.

5.2.6 Блоки установки начальных состояний

Блоки установки начальных состояний вводятся ключевым словом ***initial***. Главное отличие таких блоков от always-блоков в том, что они выполняются один раз в самом начале моделирования, на момент времени 0. Эти блоки не поддерживаются системами синтеза (как правило), но широко используются в testbench-модулях. Формальный синтаксис такого модуля выглядит так:

initial
[***begin*** | ***fork***] [: <block_name>]
<block_statements>;
[***end*** | ***join***]

Списка чувствительности в таких блоках нет, поскольку он выполняется только один раз в начале моделирования. Как видно из его названия, наиболее часто эти блоки применяются для установки начальных состояний. Однако вводя задержки и операторы присвоения, можно сформировать некоторую временную диаграмму. В одном модуле может быть несколько initial-блоков, их количество не ограничивается. Чаще всего в блоках используется пара ***begin*** – ***end***, хотя пара ***fork*** – ***join*** тоже допустима. Разумеется допустимы и все операторы разрешенные для подобных блоков. В листинге 5-6 приведен пример initial-блока который реализует простую временную диаграмму.

Листинг5-6:

```
  initial
    begin
      Sig = 0;
      #5 Sig = 1;
      #5 Sig = 0;
      #10 Sig = 1;
      #10 Sig = 0;
      #15 Sig = 1;
      #15 Sig = 0;
    end
```

В начальный момент времени 0 переменной Sig было присвоено нулевой значение. Через 5 временных единиц ее значение стало равным 1, еще через 5 единиц она вернулась в нулевое состояние и так далее. Этот блок задал поведение переменной на интервал времени в 60 временных единиц, далее переменная перешла в нулевое состояние и в нем осталась. Подобная временная диаграмма может использоваться в процедурах моделирования проекта. Для того чтобы избежать конфликтных ситуаций, значение переменной лучше изменять только в одном блоке. В разных блоках (в том числе и в always-блоках) значение переменной можно изменять, но эти изменения не должны перекрываться по времени. Часто делается так: создается только один initial-блок, где пременным присваивается только начальное состояние в нулевой момент времени. Дальнейшие операции с переменной выполняются в одном или нескольких always-блоках, при этом списки чувствительности (списки событий) блоков не должны перекрываться.

Более подробное описание блоков начальной установки будет сделано в разделе «**Testbench-модули**».

5.2.7 Specify-блоки

Блоки подобного типа используются для спецификации временных задержек по путям (pathes) модуля. Пути обычно определяются для комбинационной логики и представляют собой маршрут от какого-либо одного терминала до другого. В листинге 5-4 был рассмотрен комбинационный блок реализующий функцию Y = (A and B) or (C and D). Путями в этом блоке являлись маршруты от входных сигналов A, B, C и D до выходного сигнала Y:

A -> x1 -> Y;
B -> x1 -> Y;
C -> x2 -> Y;
D -> x2 -> Y;

Specify-блоки позволяют установить задержки распространения по каждому из путей. Разумеется это играет роль только в процессе моделирования, в процессе синтеза эти задержки игнорируются (некоторые системы синтеза могут даже вообще не поддерживать подобные конструкции, будут выдавать ошибки). Формальный синтаксис этих блоков следующий:

specify
    <specify_items>
endspecify

Вопрос об использовании specify-блоков заслуживает отдельного разговора, и в данном разделе не рассматривается.

5.2.8 Присвоение значений переменным

Значения могут присваиваться переменным не только внутри блоков, но и непосредственно в теле модуля. В зарубежной литературе фигурирует термин continious assignments, который говорит о том что данной переменной значение присваивается непрерывно, при любом изменении операндов в правой части выражений. Разумеется подобное присвоение можно выполнять только для цепей. Переменные других типов в таких выражениях недопустимы. Различают два варианта подобных присвоений: присвоение непосредственно в месте объявления переменной и присвоение после объявления переменной. Ниже приведен формальный синтаксис подобных присвоений.

<nettype> [<drive_strength>] [<expand_range>] [<delay>] <list_of_assignments>;

assign [<drive_strength>] [<delay>] <list_of_assignments>;

В первом варианте все поля кроме последнего являются теми же самыми, что были рассмотрены в пункте 3.2.3. Поле **list_of_assignments** содержит не просто список переменных, но список переменных с присвоенными им значениями. Эти значения могут быть константами, выражениями, результатом функции, операциями конкатенации, условным операторами. Во втором случае используется ключевое слово assign и отсутствует поле **expand_range**, поскольку переменная должна быть объявлена ранее. Не стоит указывать поля **drive_strength** и **delay** в двух местах, т.е.

при объявлении и в процессе присвоения, в этом случае может иметь место непредсказуемый результат, это в стандарте не обговаривается и разные системы моделирования могут по разному решать этот конфликт.

В листинге 5-7 – 5-9 приведены примеры модуля полностью аналогичные примеру в листинге 5-4.

Листинг5-7:

```
module comb_block(A, B, C, D, Y);
  input A, B, C, D;
  output Y;
  wire x1, x2;
  assign x1 = A & B, x2 = C & D;
  assign #5 Y = x1 | x2;
endmodule
```

Листинг5-8:

```
module comb_block(A, B, C, D, Y);
  input A, B, C, D;
  output Y;
  wire x1 = A & B, x2 = C & D;
  assign #5 Y = x1 | x2;
endmodule
```

Листинг5-9:

```
module comb_block(A, B, C, D, Y);
  input A, B, C, D;
  output Y;
  assign #5 Y = (A & B) | (C & D);
endmodule
```

В этих примерах продемонстрированы три приема построения модуля. Какой из вариантов выбрать решает сам разработчик в соответствии со своими пристрастиями.

5.2.9 Объявление задач (процедур)

Задачи как и функции предназначены для выполнения каких-либо общих процедур в нескольких различных местах описания, а также для разбиения больших процедур не группу процедур меньшего объема для повышения читабельности кода. Входные, выходные и двунаправленные сигналы могут служить значениями аргументов как для функций, так и для задач. В этом пункте рассматривается формальное описание задач и их основные свойства. Основные свойства задач следующие:

- Задачи могут содержать элементы временного контроля – задержки, события.
- Из тела задачи можно вызвать другую задачу или функцию.
- Задача может не иметь аргументов или несколько аргументов любого типа.
- Задача не возвращает значений.

Формальный синтаксис объявления задачи начинается с ключевых слов ***task*** и ***endtask***. Задача должна быть объявлена в теле модуля по следующим правилам:

task <name_of_task>
<task_var_declaration>
<statement_or_null>
endtask

Имя задачи **name_of_task** представляет собой уникальный в пределах модуля идентификатор. Объявление переменных ***task_var_declaration*** включает в себя следующие поля:

- Объявления параметров.
- Объявления входных переменных.
- Объявления выходных переменных.
- Объявления регистровых переменных.
- Объявления переменных типа ***time***, ***integer***, ***real***, ***event***.

Все эти переменные объявляются по тем же самым правилам, что и модульные объявления. В поле **statement_or_null** могут располагаться блоки ограниченные парами ***begin*** – ***end*** и ***fork*** – ***join***, могут располагаться непрерывные присвоения, а может и вообще ничего не быть (все присвоения сделаны при объявлении переменных). В листинге 5-10 приведен пример использования задачи для разрешения конфликта при записи данных в регистры (аналогичный примеру в лист. 5-2).

Листинг5-10:

```
module reg_bank(DIN, DREG1, DREG2, DREG3, RG1, RG2, RG3, RST, CLK);
  input [7:0] DIN;
  input RG1, RG2, RG3, RST, CLK;
  output [7:0] DREG1;
  output [5:0] DREG2;
  output [3:0] DREG3;
  reg CE1, CE2, CE3;
  always @(RG1 or RG2 or RG3)
    begin
      resolve (RG1, RG2, RG3, CE1, CE2, CE3);
    end
  dreg Register_1 (DIN, DREG1, CE1, RST, CLK);
  dreg #( 6) Register_2 (DIN[5:0], DREG2, CE2, RST, CLK);
  dreg #( 4) Register_3
    (.CLK(CLK), .RST(RST), .DIN(DIN[3:0]), .DOUT(DREG3), .CE(CE3));

  task resolve
    input I1, I2, I3;
    output O1, O2, O3;
    begin
      if (I1 && !I2 && !I3)
        {O1, O2, O3} = 3'b100;
      else  if (!I1 && I2 && !I3)
        {O1, O2, O3} = 3'b010;
      else if (!I1 && !I2 && I3)
        {O1, O2, O3} = 3'b001;
      else
        {O1, O2, O3} = 3'b000;
    end
  endtask
endmodule
```

В данном примере реализована задача под названием resolve, которая должна предохранять от некорректной записи в регистры. Если установлен только один из битов RG1, RG2, RG3, то это считается корректным кодом и выходные биты CE1, CE2, CE3 устанавливаются в соответсвии со входными битами. В случае если ошибочно установлены два или три входных бита, то все выходные биты устанавливаются в нулевое состояние, запрещая ошибочную запись в несколько регистров сразу. Следует обратить внимание, что переменные CE1 – CE3 объявлены регистровыми. Цепи в данном контексте использовать нельзя, поскольку вызов задачи располагается в always-блоке, т.е. не выполняется условие непрерывного присваивания значения CE1 – CE3 – цепи не удерживают своего состояния. Тело задачи может располагаться в любом месте модуля, может даже располагаться в отдельном файле – этот файл нужно включить в модуль по директиве **`include**. Об использовании этой директивы будет рассказано в разделе «**Директивы компилятора**». Есть еще два момента, связанные с объявлением и вызовом задачи: первый момент – при объявлении не указывается список аргументов задачи, порядок следования аргументов определен порядком списка входов и выходов задачи. Второе – при вызове задачи можно использовать только позиционное соответствие, попытка использовать ключевое вызовет ошибку.

5.2.10 Объявление функций

В языке Verilog функции используются аналогично задачам, но имеют ряд ограничений по сравнению с ними:

- Функции не могут содержать элементы временного контроля, они выполняются одномоментно.
- Из тела функции можно вызвать другую функцию, но задачу нельзя.
- Функция всегда должна иметь хотя бы один аргумент.
- Функция всегда возвращает одно значение.

Функции объявляются следующим образом:

function [<range>] <name_of_function>
    <function_var_declaration>
    <statement >
endfunction

Опциональное поле **range** определяет разрядность значения возвращаемого функцией. По умолчанию функция возвращает одноразрядное значение. Поле **function_var_declaration** идентично полю **task_var_declaration** в предыдущем пункте (Отсутствуют только объявления выходных и двунаправленных портов). Поле statement не может быть пустым, как минимум одно выражение должно быть – это присваивание неявной переменной, совпадающей с именем функции, значения которое должно быть возвращено. В листнге 5-11 приведен пример использования функции для той же цели, что и в предыдущем примере. Как видно из этого примера, функция возвращает трехразрядный вектор, определяющий состояние битов CE1 – CE3. Это типичный пример того, как с помощью функции вернуть несколько значений. По сравнению с задачей функция возвращает значения по другому механизму: в задаче это выполнялось за счет присваивания результата выходам задачи, а в данном случае происходит присвоение результата переменной resolve – это как раз та самая неявная переменная.

Листинг5-11:

```
module reg_bank(DIN, DREG1, DREG2, DREG3, RG1, RG2, RG3, RST, CLK);
  input [7:0] DIN;
  input RG1, RG2, RG3, RST, CLK;
  output [7:0] DREG1;
  output [5:0] DREG2;
  output [3:0] DREG3;
  reg CE1, CE2, CE3;

  always @(RG1 or RG2 or RG3)
    begin
      {CE1, CE2, CE3} = resolve (RG1, RG2, RG3);
    end
  dreg Register_1 (DIN, DREG1, CE1, RST, CLK);
  dreg #( 6) Register_2 (DIN[5:0], DREG2, CE2, RST, CLK);
  dreg #( 4) Register_3
    (.CLK(CLK), .RST(RST), .DIN(DIN[3:0]), .DOUT(DREG3), .CE(CE3));

  function [2:0] resolve;
    input I1, I2, I3;
    begin
      if (I1 && !I2 && !I3)
        resolve = 3'b100;
      else  if (!I1 && I2 && !I3)
        resolve = 3'b010;
      else if (!I1 && !I2 && I3)
        resolve = 3'b001;
      else
        resolve = 3'b000;
    end
  endfunction
endmodule
```

5.3 Макромодули

Verilog HDL включает в себя такую конструкцию как макромодуль. макромодули выполняют те же самые функции, что и обычный модуль, но в некоторых реализациях макромодули могут моделироваться гораздо быстрее. Когда симулятор компилирует экземпляр макромодуля он сливает его определение с определением модуля который этот экземпляр содержит. В результате не возникает границ между модулем и макромодулем и не выполняется соединение портов. Вместо этого он размещает определение макромодуля на том же самом иерархическом уровне, что и модуль в котором этот экземпляр создан. Этот процесс называется расширением макромодуля, а этот макромодуль называется расширенным. Макромодуль объявляется точно также, как и обычный модуль, но ключевое слово ***module*** заменяется на ключевое слово ***macromodule***. Экземпляры макромодулей создаются идентично экземплярам обычных модулей.

5.4 Встроенные примитивы

Комбинационные логические схемы могут моделироваться как с использованием непрерывных присвоений, так и с использованием логических вентилей и перключающих элементов. Моделирование с использованием логических вентилей имеет ряд преимуществ:

- Вентили обеспечивают максимально тесное соответствие между реальной схемой и ее моделью.
- Не существует операций непрерывного присвоения, эвивалентных двунаправленной передаче данных.
- В процессе синтеза обеспечивает более компактную и быстродействующую конструкцию, что может быть полезно для построения быстродействующих схем.

Недостатком подобного подхода является то, что вентили и перелючающие элементы имеют только один скалярный выход, поэтому они не могут быть источниками для цепей, объявленных с ключевым словом ***vectored***.

Подобные вентили и переключающие элементы, зарезервированные в стандарте языка Verilog, часто называют встроенными примитивами. Ниже представлен список ключевых слов, определяющих тип вентиля или переключающего элемента.

and	***buf***	***nmos***	***tran***	***pullup***
nand	***not***	***pmos***	***tranif0***	***puldown***
nor	***bufif0***	***cmos***	***tranif1***	
or	***bufif1***	***rnmos***	***rtran***	
xor	***notif0***	***rpmos***	***rtranif0***	
xnor	***notif1***	***rcmos***	***rtranif1***	

Поскольку целью данного документа является в первую очередь изучение синтезируемых конструкций, а также конструкций для их моделирования, то остановимся на разборе вентилей в первых двух столбцах списка. Остальные вентили и переключающие элементы системами синтеза как правило не поддерживаются. Более подробно об этих примитивах можно узнать в [1].

5.4.1 Вентили типов ***and, nand, or, nor, xor, xnor***

Вентили этих типов реализуют логические функции, соответствующие их названию. При создании экземпляров вентилей этой группы, первый порт в списке всегда является выходом (как и для всех примитивов, собственно говоря). Количество входов должно быть не меньше двух (может и больше, количество по стандарту не ограничивается, но может быть ограничено системой синтеза). Случай, когда количество входов больше двух, можно представить как каскадное включение двухвходовых вентилей, при этом суммарная задержка идентична задержке на одном вентиле. Такое представление полезно при создании многовходовых вентилей ***xor*** или ***xnor***. Вентили типов ***nand***, ***nor*** и ***xnor*** представляются так, как вентили ***and***, ***or***, ***xor***, но с инвертированным выходом. Создание экземпляров вентилей и переключающих элементов было рассмотрено в пункте 5.2.4 «**Создание экземпляров примитивов**». В таблицах 5-1 – 5-6 представлены таблицы истинности двухвходовых вентилей этих типов с учетом равной «мощности» входных переменных («Мощность» мы в

подробностях рассматривать не будем – эту информацию можно подчерпнуть из [1]. Это понятие игнорируется системами синтеза и бесполезно при моделировании синтезируемых конструкций).

Табл. 5-1 Вентиль ***and***

and	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

Табл. 5-2 Вентиль ***nand***

nand	0	1	x	z
0	1	1	1	1
1	1	0	x	x
x	1	x	x	x
z	1	x	x	x

Табл. 5-3 Вентиль ***or***

and	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

Табл. 5-4 Вентиль ***nor***

nand	0	1	x	z
0	1	0	x	x
1	0	0	0	0
x	x	0	x	x
z	x	0	x	x

Табл. 5-5 Вентиль ***xor***

xor	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

Табл. 5-6 Вентиль ***xnor***

xnor	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

*5.4.2 Вентили типов **buf, not, bufif0, bufif1, notif0, notif1***

Вентили этой группы представляют собой буферы и инверторы, с управляющим входом или без него. При создании экземпляров подобных примитивов на первом месте в списке портов стоит выход, на втором всегда один вход, на третьем (если вентиль управляемый) управляющий вход. Вентили типов ***buf*** и ***not*** являются неуправляемыми буферами и инверторами соответственно, поэтому управляющий вход у них отсутствует. Буферы часто используются в разрабатываемой конструкции для более равномерной нагрузки на выход другого вентиля (на один выход может быть подключено десятки входов, что не способстует высокому быстродействию схемы), а также для создания аппаратных задержек – в реальной схеме такой вентиль имеет реальную задержку, и система оптимизации не имеет права удалить его за «ненадобностью». Вентиль типа ***not*** – инвертор – используется по своему прямому назначению, а также для тех же целей, что и буфер. Имеет место одно существенное отличие буферов и инверторов от примитивов других типов – хотя вход всегда только один, но вот выходов может быть несколько. В данном случае последний в списке порт является входом, остальные – выходы.

Вентили остальных типов относятся к переключающим элементам, выход этих вентилей может устанавливаться в состояние высокого импеданса (выключение источника) по сигналу на его управляющем входе. Цифра в ключевом слове определяет активный уровень управляющего сигнала. В таблицах 5-7 – 5-12 приведены таблицы истинности вентилей этой группы.

В таблицах появились два новых обохначения состояний H и L. Символ H трактуется как «лог.1 или z-состояние», символ L как «лог.0 или z-состояние».

Табл. 5-7 Вентиль ***buf***

in	0	1	x	z
out	0	1	x	x

Табл. 5-8 Вентиль ***not***

in	0	1	x	z
out	1	0	x	x

Табл. 5-9 Вентиль ***bufif0***

bufif0	Control			
In	0	1	x	z
0	0	z	L	L
1	1	z	H	H
x	x	z	x	x
z	x	z	x	x

Табл. 5-10 Вентиль ***bufif1***

bufif1	Control			
In	0	1	x	z
0	z	0	L	L
1	z	1	H	H
x	z	x	x	x
z	z	x	x	x

Табл. 5-11 Вентиль ***notif0***

notif0	Control			
In	0	1	x	z
0	1	z	H	H
1	0	z	L	L
x	x	z	x	x
z	x	z	x	x

Табл. 5-12 Вентиль ***notif1***

notif1	Control			
In	0	1	x	z
0	z	1	H	H
1	z	0	L	L
x	z	x	x	x
z	z	x	x	x

5.5 Примитивы определяемые пользователем

В этом параграфе рассматривается технология создания и использования примитивов, создаваемых разработчиком – user defined primitives (UDPs). Экземпляры подобных примитивов создаются точно также как экземпляры встроенных примитивов. UDPs могут реализовывать как комбинационную логику, так и регистровую. UDPs реализующие регистровую логику используют входные значения и текущее состояние для того, чтобы определить новое состояние выхода. Для таких примитивов внутреннее состояние всегда соспадает с текущим состоянием выхода, они могут использоваться для моделирования D-триггеров (flip-flops) или триггеров-защелок (latches) с чувствительностью по фронтам или по уровню тактового сигнала соответственно. Все UDPs могут иметь только один выход, причем z-состояние выхода запрещено. Некоторые системы синтеза могут не поддерживать использование UDPs, или поддерживать только ограниченное их количество. Количество входов по стандарту не ограничивается, но могут быть ограничения систем синтеза.

5.5.1 Объявление примитивов разработчика

Поскольку примитив является самостоятельной иерархической единицей эквивалентной модулю, не допускается располагать его объявление внутри какого-либо модуля. Формальный синтаксис объявления UDPs представлен ниже:

primitive <UDP_name> (<output_terminal>, <input_terminal>, … <input_terminal>);
    <UDP_declaration>
    [<UDP_initial_statement>]
    <table_definition>
endprimitive

Объявление примитива заключается между парой ключевых слов ***primitive*** и ***endprimiteive***. **UDP_name** – имя (идентификатор) примитива, В круглых скобках далее следует список портов примитива. Порты примитива должны быть объявлены в поле

UDP_declaration по тем же самым синтаксическим правилам, что и в обычном модуле, за тем исключением, что все порты должны быть одноразрядными. Для регистровых примитивов добавляется регистровая переменная с именем идентичным имени выходного порта (**UDP_reg_declaration**).

UDP_declaration:
<UDP_output_declaration>
[<UDP_reg_declaration>]
<UDP_input_declaration>

UDP_output_declaration: ***output*** <output_terminal_name>;
UDP_reg_declaration: ***reg*** <output_terminal_name>;
UDP_input_declaration: ***input*** <input_terminal_name>, …;

Опциональное поле **UDP_initial_statement** добавляется только для регистровых переменных, при наличии такой необходимости. Начальное значение можно ввести одним из предложенных вариантов.

UDP_initial_statement: ***initial*** <output_terminal_name> = <init_val>;

init_val: 1’b0; 1’b1; 1’bx; 1; 0

Ядром примитива является таблица, фактически таблица истинности, оформленная по некоторым простым правилам. Таблица заключена между ключевыми словами ***table*** и ***endtable***. Различают таблицы для комбинационной и для регистровой логики.

table_definition:
table
<table_entries>
endtable

table_entries:
<combinational_entry> | <sequential_entry>

combinational_entry:
<level_input_list> : <output_state>;

sequential_entry:
<level_input_list> : <state> : <next_state>;

В таблице **level_input_list** представляет собой список состояний входных переменных для которого определяется состояние выхода. Элементы списка разделяются между собой пробелами (табуляцией) и представляют собой символы, представленные в таблице 5-13. Порядок следования входных переменных в таблице должен следовать порядку объявления портов примитива. Через символ двоеточия в примитивах, реализующих комбинационную логику, следует поле **output_state**, описывающее состояние выхода приданной комбинации входов. Если в таблице не обнаружено комбинации, соответствующей входам, то выход устанавливается в неопределенное состояние. Для примитивов, реализующих регистровую логику, добавляется дополнительное поле **state** описывающее текущее состояние выхода

примитива, в поле **next_state** – состояние в которое выход должен перейти. В регистровых примитивах могут комбинироваться входные состояния для регистровой и комбинационной логики. Поле выходного состояния может принимать значения 1, 0, x, X или – (символ (-) применяется только для регистровой логики и обозначает, что текущее состояние не изменяется.

Табл. 5-13 Символы входных воздействий

Симв.	Альтернатива	Назначение	Примечание
0		Состояние лог.0	Для всех видов логики комбинац. и регистровой
1		Состояние лог.1	
x	X	Неопределенное или z-состояние	
?		Безразличное состояние	
b	B	Состояние лог.0 или лог.1	
r	R, (01)	Фронт тактового сигнала	Только для регистровой логики
f	F, (10)	Срез тактового сигнала	
p	P, (01), (0x), (x1), (z1), (1z)	Фронт тактового сигнала с учетом неопределенных состояний	
n	N, (10), (1x), (x0), (0z), (z0)	Срез тактового сигнала с учетом неопределенных состояний	
*	(??)	Любое изменение тактового сигнала	

5.5.2 Пример реализации комбинационной логики

В листинге 5-12 приведен пример реализации мультиплексора 4 в 1 с использованием UDP.

Листинг5-12:

```
primitive mux41(O, I1, I2, I3, I4, S1, S2);
  input I1, I2, I3, I4, S1, S2;
  output O;
  table
  //     I1    I2    I3    I4    S1    S2    :     O
    0    0     0     0     b     b     :     0
    1    1     1     1     b     b     :     1
    1    ?     ?     ?     0     0     :     1
    ?    1     ?     ?     0     1     :     1
    ?    ?     1     ?     1     0     :     1
    ?    ?     ?     1     1     1     :     1
    0    ?     ?     ?     0     0     :     0
    ?    0     ?     ?     0     1     :     0
    ?    ?     0     ?     1     0     :     0
    ?    ?     ?     0     1     1     :     0
    x    ?     ?     ?     0     0     :     x
    ?    x     ?     ?     0     1     :     x
    ?    ?     x     ?     1     0     :     x
    ?    ?     ?     x     1     1     :     x
    ?    ?     ?     ?     x     ?     :     x
    ?    ?     ?     ?     ?     x     :     x
  endtable
endprimitive
```

В этом примере следует обратить внимание на использование символов ? и b. Эти символы позволяют в значительной мере сократить объем таблицы. Вообще говоря, таблица строится исходя из СДНФ (Совершенная Дизьюнктивная Нормальная Форма) логической функции. Однако размер СДНФ для приведенного примера будет очень

значительным. Можно построить таблицу используя МДНФ (Минимальная Дизьюнктивная Нормальная Форма), но для этого придется минимизировать СДНФ, что не всегда хочется делать. Использование символа b в первых двух строках позволило убрать шесть дополнительных строк (Все комбинации S1 и S2 для каждого списка входов мультиплексора в этих двух строках). Если же не использовать символ ?, то каждую из последующих восьми строк пришлось бы заменить четырьмя – в результате мы сэкономили 24 строки. В целом код получился на 30 строк короче, что полезно для лучшей читабельности кода. Фактически использование символов ? и b позволило нам в неявной форме минимизировать СДНФ. Последующие шесть строк таблицы строго говоря необязательны – при возникновении такой ситуации подходящей строки в таблице бы не нашлось и результатом все равно было бы неопределенное состояние. Однако, лучше внести эти строки, чтобы не возникло каких-либо неоднозначностей у челокека, который ваш код будет читать. Пример в данном случае простой, разработчик должен знать как ведет себя мультиплексор. Но если вы реализуете логическую функцию какого-либо «нестандартного» вида, то может возникнуть непонимание того, чего-же вы хотели сделать.

5.5.3 Пример реализации регистровой логики

В этом пункте рассматриваются два примера: пример реализации триггера-защелки с асинхронным сбросом и установкой (листинг 5-13) и реализацию Т-триггера с асинхронным сбросом и установкой (листинг 5-14).

Листинг5-13:

```
primitive latch(O, CLK, D, S, R);
  output O;
  reg O;
  input CLK, D, S, R;
  table
  //CLK   D     S     R     :     State :     O
    ?     ?     0     1     :     ?     :     0     // Reset latch
    ?     ?     1     0     :     ?     :     1     // Set latch
    ?     ?     1     1     :     ?     :     x     // Unknown result
    0     ?     0     0     :     ?     :     -     // No changes
    1     1     0     0     :     ?     :     1     // Latch logical
                                                    // one
    1     0     0     0     :     ?     :     0     // Latch logical
                                                    // zero
  endtable
endprimitive
```

В данном примере состояние выхода примитива практически не зависит от его текущего состояния, это частный случай, не правило. В общем случае текущее состояние влияет на результат, как показано в следующем примере.

Листинг5-14:

```
primitive ttrig(O, CLK, S, R);
  output O;
  reg O;
  input CLK, S, R;
  table
//CLK     S     R     :     State :     O
    ?     0     1     :     ?     :     0     // Reset trigger
```

```
    ?     1     0     :     ?     :     1     // Set trigger
    ?     1     1     :     ?     :     x     // Unknown result
    r     0     0     :     0     :     1     // Change state
    r     0     0     :     1     :     0     // Change state
    n     ?     ?     :     ?     :     -     // No changes
    *     x     ?     :     ?     :     x     // Unknown result
    *     ?     x     :     ?     :     x     // Unknown result
  endtable
endprimitive
```

Интересным моментом в данном примере является использование символов r и n. С первым все понятно – фиксируется возрастающий фронт. Во втором случае сделана попытка исключить любые вариации тактового сигнала, которые могут быть истолкованы как срез тактового сигнала. Этот подход, как мне кажется, более правилен с точки зрения того, что фиксированный срез (f) не закрывает все допустимые вариации сигнала – например, переход сигнала из лог.1 в неопределенное состояние. По логике формирования примитива это должно вызвать неопределенное состояние на выходе, что действительности не соответствует. Еще один момент, тактовый сигнал в подобных примитивах может быть только один. Последние две строки предназначены для того, чтобы отразить ситуацию, связанную с неопределенными значениями на входах сброса-установки.

5.6 «Черные» и «Белые» «ящики»

Под этими терминами понимают объект-модуль, созданный вами или другим разработчиком код которого запрещено или невозможно изменить (Разумеется, ничего невозможного нет, просто на это потребуется больше времени). Под невозможностью изменить код понимается ситуация когда модуль представлен в виде низкоуровнего описания (EDIF например) или вообще под руками модуля как такового нет (Допустим, что модуль содержит какую либо патентованную информацию, которая еще не оплачена). Модуль, содержимое которого вам не известно, но известен его внешний интерфейс (входы – выходы) и его функциональное назначение (описание модуля) называют «черным ящиком». Ваша задача заключается в грамотом встраивании подобного модуля в проект или моделировании схемы с таким модулем, если есть в наличии низкоуровневое описание. «Белым ящиком» называется модуль исходный код которого вы можете посмотреть, но менять его запрещено. Такая ситуация может сложиться в случае, если модуль входит в состав некоторой интегрированной библиотеки, обслуживающей несколько проектов и этот модуль уже используется в ряде проектов. В этом случае, правда, у вас есть возможность создать его копию под другим именем и делать с модулем все, что угодно. Другой причиной запрета может быть его жесткая привязка к платформе, на которой реализуется проект. Любые изменения могут повлечь за собой разрушение временных диаграмм, что приведет к значтиельным трудозатратам по их исправлению или может просто возникнуть отказ системы синтеза в имплементации модифицированного кода.

Как и для любого другого модуля для «ящиков» можно создать экземпляр, используемый в другом модуле. Может возникнуть ситуация, когда один или несколько входов-выходов для вашей задачи не нужны. Поскольку удалить их из тела «ящика» нельзя, то приходиться как-то эту ситуацию обходить. Экземпляр модуля-ящика обычно создается по следующим правилам:

- В списке соединений используется ключевое соответствие (для «черного ящика» порядок следования портов вам может быть не известен, а для «белого» используются не все выходные порты).
- Неиспользуемые входы пропускать нельзя, на них устанавливается неактивное значение.
- Неиспользуемые выходы можно пропустить, используя ключевое соответствие портов.
- Для «черных ящиков» представленных низкоуровневым кодом нельзя менять параметры, даже если в описании параметры есть.
- В проект должен быть внесен либо низкоуровневый код, либо подключена библиотека, либо непосредственно включен файл модуля (для «белого ящика»)

Во всех остальных отношениях экземпляр модуля-ящика создается по тем же синтаксическим правилам, что и экземпляр обычного модуля.

6. Поведенческое описание

Конструкции языка, изложенные в предыдущем разделе, представляют собой элементы иерархического или структурного описания проекта. Разработка и моделирование проекта с использованием логических вентилей, модулей и примитивов тесно связаны с логической структурой проекта, однако эти конструкции не обеспечивают необходимый уровень абстракции для описания комплекса высокоуровневых аспектов системы. Для решения этой проблемы используется поведенческое описание, позволяющее в полной мере использовать возможности языка для описания проекта на высоком уровне абстракции – т.е. переходу от описания логической структуры к описанию поведения объекта. Поведенческое описание широко используется в таких объектах как always-блоки, initial-блоки, задачи и функции (в функциях используется только поведенческое описание). Поведенческое описание характеризуется одним главным признаком – последовательным выполнением операторов. Это говорит о том, что такое описание может быть локализовано между парой ключевых слов ***begin*** – ***end***. Все что находится вне этой пары относится к структурному описанию. Основные опреаторы такого описания в целом очень похожи на выражения обычных языков программирования. К таким операторам относатся операторы ветвления (***if***), выбора (***case***, ***casex***, ***casez***), цикла (***for***, ***while***, ***repeat***, ***forever***). Такие операторы нельзя применять в структурном описании или внутри пары ключевых слов ***fork*** – ***join***. В этом разделе будут рассмотрены синтаксис этих операторов и их типичное применение.

6.1 Процедурные присвоения значений переменным

Процедурные присвоения предназначены для изменения значений таких переменных как регистры (***reg***), целые (***integer***), ***time***-переменные и элементы памяти (массивы регистров). Есть значительная разница между процедурным и непрерывным присвоением.

- Непрерывное присвоение изменяет состояние **цепи** немедленно при изменении состояний входных операндов.
- Процедурные присвоение изменяет состояние **регистровых** пременных под управлением некоторой процедуры в которой присвоение имеет место.

В правой части оператора присваивания может находится любое выражение, которое возвращает значение. Присваивание регистровой переменной (***reg***) несколько отличается от присваивания значений такой переменной как ***integer***. Отличие заключается в том, что регистровая переменная всегда беззнаковая, т.е. знаковое расширение воспринимается как целое положительное число. В случае когда выражение в правой части имеет иную разрядность, чем переменная, результат либо усекается слева до разрядности переменной, либо расширяется, путем заполнения недостающих позиций нулями.

Различают два вида процедурных присвоения: блочное процедурное присвоение и внеблочное процедурное присвоение. Эти виды присвоений определяют различное поведение процедуры в последовательных блоках. Блочное присвоение (=) выполняется немедленно, в том месте где оно встретилось в тексте, при этом переменная сразу меняет свое значение, и может быть изменена неоднократно в одной ветви алгоритма. Внеблочное присвоение (<=) изменяет значение переменной только в момент выхода из блока, поэтому, чтобы не было конфликтов, в одной ветви

алгоритма может быть только одна операция присвоения. В случае конфликта значение переменной будет неопределенным.

6.2 Оператор ветвления if

6.2.1 Синтаксис оператора ***if***

Оператор ветвления if широко применяется для реализации элементов регистровой логики, таких как регистры данных, сдвига, счетчиков, цифровых автоматов и т.д. Синтаксис этого оператора очень похож на синтаксис в языке С. Общий вид оператора выглядит так:

```
if (<cond_expr1>)
    [begin]
        <statements>;
    [end]
else if (<cond_expr2>)
    [begin]
        <statements>;
    [end]
else
    [begin]
        <statements>;
    [end]
```

Обязательной ветвью в этом операторе является ветвь после ключевого слова if. Ветви ***else if*** и ***else*** добавляются если они необходимы. Условные выражения cond_expr принимают значения **true** (результат cond_expr отличен от нуля) или **false** (результат равен нулю). Логика выполнения оператора следующая: если выполняется условие cond_expr1 (cond_expr1 != 0), то выполняется ветвь под оператором ***if***, а остальные ветви пропускаются. В случае когда условие не выполняется (cond_expr1 == 0) выполняется проверка условия cond_expr2. Ветвей ***else if*** в операторе может быть несколько. Если не выполняется предыдущее условие, то выполняется следующее. В случае когда не выполнилось ни одно из условий, то управление передается ветви ***else*** (если она есть). Когда этой ветви нет осуществляется выход из оператора. Ключевые слова ***begin*** – ***end*** обозначают границы последовательных блоков. Если в ветви только одно выражение, эти ключевые слова можно опустить. Ветви ***if***, ***else if*** и ***else*** иногда называют параллельными в том смысле, что всегда выполняется только одна из них. В случае когда разработчик применяет операцию внеблочного присвоения, любой переменной в одной ветви можно присвоить значение только один раз. Операторы ***if*** могут быть вложенными – внутри одной ветви оператора вполне может быть размещен другой оператор ***if***. В связи с этим есть один маленький «подводный камешек», он подробно будет рассмотрен в одном из следующих примеров.

Небходимо отметить, что вместо блоков ***begin*** – ***end***, допускается и применение параллельных блоков ***fork*** – ***join***. В параграфе этого раздела «**Последовательные и параллельные блоки**» будет подробно рассмотрены основные аспекты связанные с этими блоками.

6.2.2 Пример двоичного реверсивного счетчика

В листинге 6-1 представлен пример реализации двоичного реверсивного счетчика с параллельной загрузкой и асинхронным сбросом. Такое длинное название говорит о том, что счетчик может считать как с увеличением, так и с уменшением своего значения (реверсивность), начальный код в такой счетчик можно загрузить извне (параллельная загрузка) или асинхронно с тактовым сигналом сбросить его.

Листинг6-1:

```
module counter(DIN, COUNT, LOAD, CE, DIR, RST, CLK);
  parameter NBRB = 8;
  input [NBRB – 1:0] DIN;
  input LOAD, CE, DIR, RST, CLK;
  output [NBRB – 1:0 ] COUNT;
  reg [NBRB – 1:0 ] COUNT;
  always @(posedge CLK or posedge RST)
    begin
      if (RST == 1) COUNT <= 'b0;
      else
        if (CE == 1)
      begin
        if (LOAD == 1) COUNT <= DIN;
        else
          begin
            if (DIR == 1) COUNT <= COUNT + 1;
            else if (DIR == 0) COUNT <= COUNT - 1;
          end
      end
    end
endmodule
```

В этом примере используется три вложенных оператора ***if***. По ветке ***if*** первого (не вложенного) оператора выполняется сброс счетчика в нулевое значение. По ветви ***else*** этого оператора сначала проверяется разрешен доступ к счетчику или нет (первый вложенный оператор ***if***). Если условие CE == 1 не выполняется, то счетчик остановлен, и зарузить в него новое значение нельзя. Если условие выполнено (счетчик доступен) то проверяется какую операцию выполнять (второй вложенный оператор ***if***): загрузку нового значения (LOAD == 1) или выполнять счет. Если выбрана операция пересчета, то выясняется в каком напрвлении считать (третий вложенный ***if***). В случае если DIR равен 1, производится пересчет с увеличением значения переменной COUNT, а если DIR равен нулю, то значение COUNT уменьшается. Все эти операции выполняются по фронту сигнала CLK в случае если RST равен нулю. В данном примере используются внеблочные операторы присвоения. Однако, не будет ошибкой использование и блочных операторов. В данном случае это не критично, потому что, в каждой ветви используется только один оператор присвоения, одно выражение. Это частный случай, в целом, надо очень внимательно относиться к этим операторам. Пример того, как их использование может повлиять на работу схемы будет представлен в следующем пункте.

Этот пример можно рассматривать как шаблон для построения самых различных счетчиков: можно убрать, допустим, напавление пересчета и получим самый обычный счетчик, можно добавить дополнительный сигнал, управляющий направлением счета, тогда по одному сигналу будет выполняться инкремент счетчика, по другому декремент. Такой счетчик можно легко использовать в качестве регистра с

инкрементом/декрементом. Можно в строке COUNT <= COUNT + 1 заменить единицу входным вектором DIN и получить накапливающий сумматор.

6.2.3 Влияние операторов блочного и внеблочного присвоения

В листингах 6-2 и 6-3 приведены два примера, которые показывают, что может произойти при неправильном использовании этих операторов.

Листинг6-2:

```
module shiftreg(DIN, DOUT, CE, RST, CLK);
  parameter NBRB = 8;
  input [NBRB – 1:0] DIN;
  input CE, RST, CLK;
  output [NBRB – 1:0 ] DOUT;
  reg [NBRB – 1:0 ] DOUT;
  reg [NBRB – 1:0 ] Data1, Data2;
  always @(posedge CLK or posedge RST)
    begin
      if (RST == 1)
        begin
          DOUT = ’b0; Data1 = ’b0; Data2 = ’b0;
        end
      else
        if (CE == 1)
      begin
        Data1 = DIN;
        Data2 = Data1;
        DOUT = Data2;
      end
    end
endmodule
```

Листинг6-3:

```
module shiftreg(DIN, DOUT, CE, RST, CLK);
  parameter NBRB = 8;
  input [NBRB – 1:0] DIN;
  input CE, RST, CLK;
  output [NBRB – 1:0 ] DOUT;
  reg [NBRB – 1:0 ] DOUT;
  reg [NBRB – 1:0 ] Data1, Data2;
  always @(posedge CLK or posedge RST)
    begin
      if (RST == 1)
        begin
          DOUT <= ’b0; Data1 <= ’b0; Data2 <= ’b0;
        end
      else
        if (CE == 1)
      begin
        Data1 <= DIN;
        Data2 <= Data1;
        DOUT <= Data2;
      end
    end
endmodule
```

Эти примеры идентичны во всем, кроме использования операторов присваивания. Результат синтеза этих модулей окажется абсолютно различным. Рассмотрим операции присваивания в первом примере: Значение входной переменной DIN будет немедленно присвоено переменной Data, в следующей строке уже измененное значение Data1 присваивается переменной Data2, наконец измененное значение Data2 окажется присвоенным выходной переменной DOUT. Иначе говоря все три переменные Data1, Data2 и DOUT примут одно и то же значение входной переменной DIN. Разумеется, если в этом и состояла ваша цель, то можете все оставить и так. Хотя в этом случае можно было сделать проще и понятнее:

```
{DOUT, Data1, Data2} = {DIN, DIN, DIN};
```

Целью же этого примера было создание сдвиговой цепочки из трех регистров – переменные Data1 и Data2 не присутствуют среди выходов модуля. Очевидно, что первый вариант этой цели не удовлетворяет в отличие от второго. Во втором варианте происходит следующее: Оценивается значение переменной DIN, которое **будет** присвоено переменной Data1, оценивается текущее состояние Data1 (еще не измененное – из блока мы пока не вышли), которое **будет** присвоено переменной Data2, оценивается текущее состояние Data2, которое **будет** присвоено выходной переменной DOUT. При выходе из блока произойдет одновременное присвоение всех оцененных состояний соответствующим переменным. В результате переменная DOUT примет состояние Data2, переменная Data2 – состояние Data1, а переменная Data1 – состояние DIN. Иначе говоря, поставленная цель будет достигнута. Возникает один интересный вопрос: а можно-ли в первом варианте добиться того-же эффекта – ответ: можно! Для этого поменяем порядок следования операторов:

```
DOUT = Data2;
Data2 = Data1;
Data1 = DIN;
```

Хороший этот вариант или нет? Приемлемый, но не очень хороший. В данном случае пример простой и все как бы очевидно. Но если задача будет сложнее, то придется хорошо продумать в каком порядке расположить операторы присвоения. Возвращаясь ко второму примеру, можно сказать, что как бы мы ни меняли порядок следования операторов – результат будет неизменен.

Совет:

Реализуя схемы регистровой логики, лучше использовать внеблочное присваивание в тех случаях, когда в одной ветви присваивание выполняется однократно для одной и той же переменной. Использование блочного присваивания чревато некорректным поведением схемы. В случае, когда в пределах ветви одной и той же переменной значение присваивается несколько раз в разных местах алгоритма, наоборот, использование внеблочного присваивания недопустимо во избежание конфликтов. Для таких переменных лучше всего использовать тип ***integer****, это улучшает читабельность кода. Смешивание в одном алгоритме операций блочного и внеблочного присваивания допустимо только для разных переменных, для одной и той же переменной подобное может привести к ошибке, которая системами синтеза не диагностируется и найти такую ошибку крайне сложно.*

6.2.4 Использование вложенных операторов ***if***

В приведенных выше примерах использовались вложенные операторы ***if***. В целом, их использование не вызывает никаких затруднений, но существует небольшой «подводный камень». В приведенном ниже примере никакой ошибки не допущено:

Листинг 6-4:

```
module counter(DIN, DOUT, LOAD, CE, RST, CLK);
  parameter NBRB = 8;
  input [NBRB – 1:0] DIN;
  input LOAD, CE, RST, CLK;
  output [NBRB – 1:0 ] DOUT;
  reg [NBRB – 1:0 ] DOUT;
  always @(posedge CLK or posedge RST)
    begin
      if (RST == 1) DOUT <= 'b0;
      else
        if (CE == 1)
        if (LOAD == 1) DOUT <= DIN;
        else;                              // ***
    else
        DOUT <= DOUT + 1;
    end
endmodule
```

В этом примере реализован счетчик, работающий по следующему алгоритму: если не установлен сигнал сброса (RST), но установлены переменные CE и LOAD, то происходит загрузка счетчика. Если CE и LOAD сброшены, то счетчик выполняет инкремент значения DOUT по каждому фронту тактового сигнала. Следует обратить внимание на ключевое слово ***else***, обозначенное звездочками. Под этой веткой не стоит ни одного выражения, но ветка все равно существует. Если эту ветку убрать, то, разумеется, никакой синтаксической ошибки мы не получим – зато получим ошибку логическую. В этом случае выражение DOUT <= DOUT + 1 попадает под другой оператор ***if***. То есть эта строка будет принадлежать оператору ***if*** с условием проверки состояния переменной LOAD, в то время когда она должна принадлежать оператору с условием проверки переменной CE. Результат такой перестановки нетрудно представить – счетчик будет считать только когда CE установлен, а условие было, что счетчик ведет пересчет когда оба эти сигнала сброшены. В языке Verilog существует правило: ветви ***else*** и ***else if*** принадлежат ближайшему оператору ***if*** по направлению к началу блока. В силу этого правила и происходит такой эффект. В данном случае пустая ветка ***else*** просто необходима, чтобы вторая ветвь попала «на свое место». Следует обратить внимание на символ точки с запятой после ***else***: пустое выражение все же остается выражением и должно заканчиваться этим символом. Вывод из всего вышесказанного можно сделать такой: если используется ряд вложенных операторов ***if***, надо тщательно следить за ветвями ***else***, какая какому оператору принадлежит.

6.3 Операторы выбора case, casex, casez

Оператор выбора представляет собой похожую конструкцию как и оператор switch в языке C. Существует три формы этого оператора определяемые ключевыми словами ***case***, ***casex***, ***casez***. В этом параграфе будет рассмотрен синтаксис этих операторов, их назначение и возможные проблемы, связанные с их применением.

*6.3.1 Синтаксис операторов **case, casex, casez***

Все три формы синтаксически отличаются между собой только ключевым словом, а логически - своим поведением. Поскольку в этом пункте рассматривается синтаксис, то делать это мы будем на примере оператора ***case***.

Оператор выбора представляет собой частный случай оператора ветвления, его можно представить себе как оператор ***if*** с большим количеством веток ***else if***. В таком представлении это менее удобный оператор чем if, поскольку выбор выполняется при совпадении условия с одним из вариантов, которые представляют собой числовые константы, в то время как в операторе ***if*** анализируется условие по принципу «истинно-ложно». Другими словами, применять в условном выражении понятия «истина-ложь» в операторе ***case*** бесполезно. Пример: выражения типа (a == 5 && b <= 2 && !c) в качестве условия для оператора ***case*** бесполезно – оно может принять только состояния 0, 1 и x, а с выбором из этих значений справляется простейший оператор ***if***. Однако эти операторы нашли очень широкое применение в комбинационной логике в качестве дешифраторов и очень удобны в описаниях конечных автоматов.

Синтаксис оператора ***case*** выглядит так:

case (<cond_expr>)
    <const_cond_1> : <statement>;
    <const_cond_2> : <statement>;
    *
    *
    *
    <const_cond_n> : <statement>;
    [***default*** : <statement>;]
endcase

В этом операторе результат cond_expr по порядку сравнивается с константными условиями const_cond. В случае совпадения результата с одним из выражений, выполняется выражение statement соответствующее этому константному условию. Если не выполнилось ни одно из условий, то выполняется ветвь под ключевым словом ***default*** (если она есть – противном случае не выполняется ни одно из выражений). Заканчивается оператор ключевым словом ***endcase***. Выражения в каждой из ветвей могут быть простыми операторами, блоками операторов, ограниченные парами ***begin*** – ***end*** или ***fork*** – ***join***, или даже другими операторами ***case***, ***if***, ***for*** и т.д. Ветвь под ключевым словом ***default*** необязательна, но в ряде случаев просто необходима для правильной работы оператора. Еще одно важное замечание, константные условия не должны дублироваться, во всяком случае система синтеза проигнорирует дублированное условие, скорее всего предупредив об этом.

*6.3.2 Пример использования оператора **case***

В качестве типичного прмера будет представлен пример реализации дешифратора данных. В примере входная переменная DATA используется в качестве выражения, которое сравнивается с набором константных условий (cond_expr). Переменная DATA имеет четыре разряда, что должно соответствовать 16-и возможным состояниям. Однако в примере намеренно используются не все константные условия соответствующие этим состояниям. В примере также присутствует сигнал разрешения дешифрации EN. Дешифрация входного состояния должна выполняться только в

случае если EN равен 1. Результатом дешифрации выступает 8-и разрядная выходная переменная DOUT. В состояниях, которые не перечислены в списке костантных условий и в случае, когда EN находится в состоянии 0 все биты этой переменной должны быть равны нулю. Пример приведен в листинге 6-5 и в этом варианте ошибок не содержит.

Листинг6-5:

```
module encoder(DATA, DOUT, EN);
  input [3:0] DATA;
  input EN;
  output [7:0] DOUT;
  reg [7:0] DOUT;
  always @(DATA or EN)
    if (EN == 1)
      case (DATA)
        4'h1: DOUT <= 8'b0000_0001;
        4'h3: DOUT <= 8'b0000_0010;
        4'h5: DOUT <= 8'b0000_0100;
        4'h7: DOUT <= 8'b0000_1000;
        4'h9: DOUT <= 8'b0001_0000;
        4'hB: DOUT <= 8'b0010_0000;
        4'hD: DOUT <= 8'b0100_0000;
        4'hF: DOUT <= 8'b1000_0000;
        default: DOUT <= 8'b0000_0000;
      endcase
    else DOUT <= 8'b0000_0000;
endmodule
```

В этом примере необходимо обратить внимание на две строки, невнимание к которым может првести к серьезным проблемам. Это ветвь по умолчанию (***default***) в операторе ***case*** и ветвь ***else*** в операторе ветвления. Пропуск этих ветвей в данном случае приведет к тому, что наша **комбинационная** схема превратится в **регистровую**. Это призойдет потому, что не все ветви алгоритма при изменении входных переменных будут реализованы. А поскольку мы хотим реализовать комбинационную схему с помощью регистровой перменной (и это желание вполне законно), то должны учитывать любые изменения входных переменных, поскольку, если нужной ветви не окажется, то переменная будет сохранять свое предыдущее состояние. Пример: мы установили EN в логическую 1, а переменную DATA в состояние 4'h7. По логике работы алгоритма выходная переменная DOUT установится в состояние 8'h08. Затем мы установили EN в состояние лог.0. Схема, приведенная в примере установит на выходе состояние 8'h00, что и требуется. Но произойдет это за счет ветви ***else*** в операторе ***if***, поэтому, если ее пропустить, состояние выхода останется 8'h08, а это значит, что система синтеза внесла в схему триггер-защелку (latch) хотя наверняка предупредила об этом, а эта ситуация в рамках поставленной задачи недопустима. Полностью аналогичная ситуация сложится если пропустить условие по умолчанию в операторе ***case***. Не все возможные константные условия перечислены в операторе, поэтому если состояние перменной DATA не совпало ни с одним из условий, а ветки по умолчанию нет, то будет сохранено предыдущее состояние, т.е. опять будет внесена защелка. И всего вышеизложенного надо сделать следующий вывод: если мы хотим использовать регистровую переменную в комбинационной схеме, необходимо предусматривать ветви алгоритма на все возможные состояния входных переменных. В этом случае система синтеза не будет иметь оснований для того, чтобы ввести триггеры-защелки. В таких случаях ветвь ***else***

должна присутствовать всегда. Ветвь по умолчанию можно исключить если список константных условий полон, т.е. учитываются все возможные состояния. Однако, лучше на этой строке не экономить, в процессе разработки вы или ваши коллеги, могут посчитать ту или иную ветвь в списке ненужной, и в результате схема будет работать совершенно не так, как задумывалось. В регистровых схемах, разумеется, эти правила не обязательны, а нужные ветви вносятся только на основании предполагаемого поведения модуля.

*6.3.3 Использование операторов **casex** и **casez***

Эти два оператора необходимы в следующем случае: допустим часть битов условия должна игнорироваться, т.е. в сравнении они просто не участвуют. Обычный оператор ***case*** этого сделать не позволит, хотя значения x или z в условии синтаксической ошибки не вызовут (такое условие вызовет предупреждение, что эта ветвь никогда не будет выполнена). Для подобных целей как раз и используются операторы ***casex*** и ***casez***. Оператор ***casex*** игнорирует биты отмеченные как x или z, оператор ***casez*** игнорирует биты в z-состоянии. Вместо символа z в операторах можно использовать символ ?, что более наглядно показывает, что состояние данного бита безразлично. В листингах 6-6 и 6-7 приводятся два примера исользования операторов ***casex*** и ***casez*** соответственно.

Листинг6-6:

```
module mask_encoder(DATA, DOUT, EN);
  input [7:0] DATA;
  input EN;
  output [7:0] DOUT;
  reg [7:0] DOUT;
  always @(DATA or EN)
    if (EN == 1)
      casex (DATA)
        8'b1100_xx00: DOUT <= 8'b0000_0001;
        8'b0110_0xx0: DOUT <= 8'b0000_0010;
        8'b0011_00xx: DOUT <= 8'b0000_0100;
        8'b1001_x00x: DOUT <= 8'b0000_1000;
        8'b1010_x0x0: DOUT <= 8'b0001_0000;
        8'b0101_0x0x: DOUT <= 8'b0010_0000;
        8'b1111_xxxx: DOUT <= 8'b0100_0000;
        8'b0000_0000: DOUT <= 8'b1000_0000;
        default: DOUT <= 8'b0000_0000;
      endcase
    else DOUT <= 8'b0000_0000;
endmodule
```

В этом модуле выполняется простая операция: если в старшем полубайте условия биты установлены в 1 то соответствующие биты в младшем полубайте игнорируются при сравнении с кодом переменной DATA. Допустим, что состояние пременной DATA равно 8'b1010_1000 – переменной DOUT будет присвоено значение 8'b0001_0000. При переходе переменной DATA в состояние 8'b1010_0010 значение переменной DOUT не изменится, поскольку соответствующие биты в младшем полубайте игнорируются (маскируются). Однако, если состояние переменной DATA станет 8'b1010_0001, то будет выполнено условие по умолчанию, поскольку нулевой бит не замаскирован, и условию не соответствует. Такой дешифратор позволяет сделать, например, автоматическую сортировку поступающих данных по разным буферам.

В следующем листинге рассматривается пример использования оператора ***casez***.

Листинг6-7:

```
module data_encoder(DATA, DOUT, EN);
  input [7:0] DATA;
  input EN;
  output [7:0] DOUT;
  reg [7:0] DOUT;
  always @(DATA or EN)
    if (EN == 1)
      casez (DATA)
        8'b1???_????: DOUT <= 8'b0000_0001;
        8'b01??_????: DOUT <= 8'b0000_0010;
        8'b001?_????: DOUT <= 8'b0000_0100;
        8'b0001_????: DOUT <= 8'b0000_1000;
        8'b0000_1???: DOUT <= 8'b0001_0000;
        8'b0000_01??: DOUT <= 8'b0010_0000;
        8'b0000_001?: DOUT <= 8'b0100_0000;
        8'b0000_0001: DOUT <= 8'b1000_0000;
        default: DOUT <= 8'b0000_0000;
      endcase
    else DOUT <= 8'b0000_0000;
endmodule
```

В этом примере состояние выхода DOUT зависит от положения бита в состоянии лог.1, при этом игнорируются все биты, порядковый номер которых меньше, чем номер бита в лог.1. Такой дешифратор может использоваться в схемах автоматической обработки поступающих последовательных данных. В целом этот дешифратор работает также как и предыдущий. Отличие лишь в том, что в этом операторе не допускается использовать символ x. Также не допускается использование символов x, z и ? во входных переменных всех операторов ***case***.

Необходимо отметить один момент, принципиальный для операторов ***casex*** и ***casez***: необходимо внимательно следить, чтобы константные условия не дублировались. Простейший пример: условия 8'b01??_???? и 8'b0111_???? оказываются дублирующими, поскольку биты установленные в лог.1 во втором условии накрываются безразличным состоянием в первом. Самое неприятное, что система синтеза может не выдать даже предупреждения на эту ошибку – это уже проверено на XST Verilog (Xilinx ISE WebPack). Причина же заключается в том, что система их дублирующими не считает (!), по всей видимости воспринимая второе условие как частный случай первого. Как в этом случае ведет себя система синтеза просто не предсказуемо. Эксперимент показывает, что первое условие и будет выполняться, а второе просто игнорироваться.

6.4 Операторы цикла

К операторам цикла в языке Verilog относятся четыре оператора: ***for***, ***while***, ***repeat***, ***forever***. Все они предназначены для повтороения некоторого выражения или блока выражений один раз, многократно или вообще ни разу. Различаются они между собой особенностями выполнения блока выражений:

- Оператор ***forever*** выполняет блок выражений непрерывно, остановить его выполнение невозможно.
- Оператор ***repeat*** выполняет блок выражений фиксированное количество раз.
- Оператор ***while*** выполняет блок выражений до тех пор, пока его условие не примет значение false («ложно» или лог.0). Если условие на момент старта уже false блок не выполнится никогда – это оператор с предусловием.
- Оператор ***for*** выполняет блок выражений в три этапа:
  1) Инициализирует переменную цикла;
  2) Анализирует условное выражение, если результат равен нулю, то цикл завершается. Иначе выполняется блок выражений и переходит на третий этап;
  3) Выполняет выражение модифицирующее переменную цикла и возвращается на второй этап. Это также оператор с предусловием.

Какой из этих операторов выбрать должен решить разработчик исходя из перечисленных выше особенностей. В следующих пунктах будет подробно рассмотрен синтаксис этих операторов и области их применения, а также примеры использования.

6.4.1 Оператор ***for***

Синтаксис этого оператора выглядит следующим образом:

```
for (<var_init_assign>; <condition>; <var_update_assign>)
    [begin]
        <statements>;
    [end]
```

В этом операторе **var_init_assign** представляет собой выражение для инициализации переменной цикла (переменная должна быть объявлена заранее – обычно она имеет тип ***integer***). Условие выполнения цикла ***condition*** представляет собой булевое выражение, которое может принимать значения true («истина», не нуль) – в этом случае блок выражений будет выполнен или false («ложь», нулевое значение) – при этом цикл завершается. Переменная цикла может входить в состав условия, а может и не входить. Если условие представляет собой ненулевую константу, то цикл будет выполняться бесконечно. Выражение для модификации переменной цикла **var_update_assign** имеет смысл если переменная цикла входит в условие цикла. Все эти три поля являются обязательными, хотя могут быть и пустыми (имеется ввиду следующий вариант: ***for*** (;;)). Этом виде оператор ***for*** идентичен оператору ***forever***. Следует отметить тот факт, что ни одна система синтеза не поддерживает бесконечные операторы и операторы у которых число циклов выполнения не представляет собой константу. Подобные операторы могут быть применены, например, в качестве генераторов входных воздействий в testbench-модулях. В синтезируемых модулях их применять нельзя.

В листинге 6-8 представлен пример реализации комбинационного перемножителя с применением оператора ***for***.

Листинг6-8:

```
module mult(ADATA, BDATA, RESULT);
  parameter size = 8;
  input [size-1:0] ADATA, BDATA;
  output [2*size-1:0] RESULT;
  reg [2*size-1:0] RESULT;
  always @(ADATA or BDATA)
    begin : multblk
      integer ind;
      RESULT = 0;
      for (ind = 0; ind < size; ind = ind + 1)
        if (BDATA[ind])RESULT = RESULT + (ADATA << ind);
    end
endmodule
```

Это очень показательный пример абстрактного описания схемы. Реализация этого алгоритма на логических вентилях заняла бы гораздо больше места – примерно на порядок. В данном же случае мы получили весьма компактный и удобочитаемый код. Этот модуль реализует обычный алгоритм перемножения «столбиком», разумеется, для двоичных чисел.

Замечание:

*При работе на некоторых платформах, таких как современные ПЛИС, есть возможность сделать этот модуль еще более компактно. Многие ПЛИС содержат на кристалле интегрированные перемножители и системы синтеза могут их использовать. В коде применяется оператор перемножения *. По стандарту языка он может использоваться либо в константных выражениях, либо в выражениях, где второй операнд является степенью 2, либо в несинтезируемых модулях. Используя интегрированные перемножители можно перемножать любые целые переменные и в синтезируемом модуле.*

Следует обратить внимание, что блок ***begin*** – ***end*** имеет собственное имя – multblk. В данном случае это важно, поскольку внутри блока объявляется переменная ind (в неименованном блоке объявлять переменные запрещено). Переменные объявленные внутри блока считаются локальными, в то время как переменные внутри модуля объявляются глобально. С точки зрения системы синтеза это очень важный момент: глобальные переменные как правило реализуются физически в виде цепи или регистра, в то время как локальные переменные представляют собой некоторую абстракцию, предназначенную лишь для описания алгоритма и, как правило, физически не реализуются, либо реализуются в неявном виде. Типичный пример такой неявной реализации мы и наблюдаем в модуле перемножителя. В данном случае переменная цикла принимает значения от 0 до 7-и, что приводит к восьми операциям сложения, определяемые 8-ю значениями переменной. По этой самой причине операторы с бесконечным циклом или не постоянным количеством циклов физически не реализуемы. Переменную цикла можно объявить и вне блока, разницы не будет никакой, в данном случае она все равно будет реализована неявно. Однако объявление внутри блока все же более правильное решение: не засоряется пространство имен и улучшается читабельность кода, т.е. сразу ясно где эта переменная используется.

6.4.2 Оператор ***repeat***

Оператор ***repeat*** используется несколько реже чем оператор ***for*** по причине отсутствия в явном виде переменной цикла. Формальный синтаксис этого оператора представлен ниже.

repeat (<expression>)
    [***begin***]
        <statements>;
    [***end***]

Этот цикл будет выполнятся столько раз, сколько получится в результате вычисления выражения **expression**. Для того, чтобы подобный оператор был синтезируемым это выражение должно быть константным. В листинге 6-9 реализован тот же самый перемножитель, но с использованием оператора ***repeat***.

Листинг6-9:

```
module mult(ADATA, BDATA, RESULT);
  parameter size = 8;
  input [size-1:0] ADATA, BDATA;
  output [2*size-1:0] RESULT;
  reg [2*size-1:0] RESULT;
  always @(ADATA or BDATA)
    begin : multblk
      reg [2*size-1:0] Shift_a;
      reg [size-1:0] Shift_b;
      Shift_a = ADATA; Shift_b = BDATA;
      RESULT = 0;
      repeat (size)
        begin
          if (Shift_b[0])RESULT = RESULT + Shift_a;
          Shift_a = Shift_a << 1;
          Shift_b = Shift_b >> 1;
        end
    end
endmodule
```

В этом примере реализован тот же самый алгоритм перемножения «столбиком». Поскольку в явном виде переменной цикла нет, то используются операции сдвига на единицу в каждом цикле, что привело к некоторому увеличению объема кода. В этом варианте переменные Shift_a и Shift_b также в явном виде не реализуются. Определить такие переменные можно если компилятор анализирует код на предмет использования/ неиспользования объявленных переменных. Обычно появляется предупреждение, что переменная объявлена, но нигде не используется. Возникает вопрос: мы используем переменную в нескольких местах, а получаем предупреждение. Это предупреждение как раз и говорит о том, что реально этой переменной нет и она не используется, хотя и объявлена. Можно провести эксперимент: выведите переменную Shift_a в качестве порта, промоделируйте этот код и посмотрите какое значение принимает эта переменная.

6.4.3 Оператор ***while***

Операторы ***while*** довольно редко используются в синтезируемых конструкциях по той причине, что трудно бывает отследить за фиксированным количеством циклов. Однако, несмотря на то, что ранее говорилось по поводу неопределенного количества циклов, все же существует возможность построения синтезируемых конструкций с переменным количеством циклов. Подобный пример приведен в листинге 6-10. В этом примере используется оператор ***while*** для реализации того же перемножителя.

Формальный синтаксис этого оператора выглядит следующим образом:

while (<expression>)
    [***begin***]
        <statement>;
    [***end***]

Цикл будет выполняться до тех пор, пока условное выражение expressions не примет нулевое значение (т.е. не станет «ложным»).

Листинг6-10:

```
module mult(ADATA, BDATA, RESULT);
  parameter size = 8;
  input [size-1:0] ADATA, BDATA;
  output [2*size-1:0] RESULT;
  reg [2*size-1:0] RESULT;
  always @(ADATA or BDATA)
    begin : multblk
      reg [2*size-1:0] Shift_a;
      reg [size-1:0] Shift_b;
      Shift_a = ADATA; Shift_b = BDATA;
      RESULT = 0;
      while (Shift_b)
        begin
          if (Shift_b[0])RESULT = RESULT + Shift_a;
          Shift_a = Shift_a << 1;
          Shift_b = Shift_b >> 1;
        end
    end
endmodule
```

По сравнению с предыдущим примером в этом заменена буквально одна строка – вместо оператора ***repeat*** (size) помещена строка ***while*** (Shift_b). Однако, теперь есть крупное различие в процедуре перемножения. Если в первом случае выполнялось фиксированное количество циклов, то во втором могло выполнится от нуля до восьми циклов. Дело в том, что при очередном сдвиге Shift_b значение этой переменной могло стать равным нулю и следовательно выполнение этого оператора завершается. Налицо ситуация, когда заранее не известно сколько циклов будет выполнено, количество зависит только от значения BDATA, но ,конечно, не более 8-и (по количеству разрядов). Почему же такая ситуация стала возможной для синтеза? Да просто потому, что переменная Shift_b используется неявно, реально такой переменной нет. Система синтеза пытается понять, что от нее требует разработчик и по тому как она ее поняла строит какой-либо вариант реализации. Иначе говоря, в данном случае используется более высокий уровень абстракции, чем в предыдущих вариантах. Именно здесь находится очень крупный «подводный камень» связанный с тем, что система не может

понять или понять правильно желание разработчика. Для каждой системы синтеза предельный уровень абстракции различный. Например такая система как XST Verilog (ISE WebPack) не в состоянии реализовать этот пример, хотя система моделирования утверждает, что все в порядке. Поскольку оператор ***while*** задает, как правило, высокий уровень абстрактного описания, то его использование часто просто нежелательно, из соображений переносимости кода. Наиболее часто этот оператор используется в testbench-модулях, синтез которых не производится.

6.4.4 Оператор ***forever***

Этот оператор практически никогда не применяется в синтезируемых конструкциях. Чаще всего его применение просто не оправдано, всегда можно использовать другую конструкцию, выполняющую ту же задачу. Этот оператор можно применять совместно с оператором ***disable*** (см. следующий параграф) или прерывать его выполнение по какому-либо событию. Формальный синтаксис этого оператора выглядит так:

forever
    [***begin***]
        <statement>;
    [***end***]

Многие системы синтеза не поддерживают работу с операторами while и forever. К таким системам синтеза относятся XST Verilog и FPGA Express.

6.5 Оператор *disable*

Основным назначением этого оператора является прерывание какого-либо процесса. Допустим, идет выполнение какого-то поименованного блока, и на определенном этапе нужно прервать его выполнение. В определенной ветви алгоритма вводится этот оператор выполняющий прерывание процесса. Этот оператор почти никогда не применяется в синтезируемых конструкциях, по той же причине, что и операторы ***while*** и ***forever*** - они не поддерживаются многими системами синтеза. Все эти операторы более или менее широко используются только в testbench-модулях. Более подробно эти операторы будут рассмотрены в разделе "**Testbench-модули**".

6.6 Оператор wait

Оператор wait предназначен для ожидания какого-либо события и блок выражений, стоящий под этим оператором будет выполнен только когда это событие произойдет. Общий синтаксис этого оператора выглядит следующим образом:

wait (<condition>)
    [***begin***]
        <statement>;
    [***end***]

Выражение **statement** будет выполнено в случае если условие **condition** примет значение true (не равно нулю), в противном случае выражение выполнено не будет. Этот оператор также не применяется в синтезируемых модулях, но может широко использоваться в testbench-модулях для анализа выходных состояний тестируемого

устройства. В отличие от методов управления событиями (event control см. ниже) этот оператор позволяет использовать сложные условия, допустим единовременное совпадение состояний нескольких переменных, которые в один и тот же момент времени совпадать не должны. Более подробно этот оператор будет рассматриваться в разделе «**Testbench-модули**».

6.7 Последовательные и параллельные блоки

Для того чтобы обозначить группу выражений и определить порядок их выполнения, используются блоки обозначенные парами ключевых слов ***begin*** – ***end*** и ***fork*** – ***join***. Мы многократно использовали подобные блоки в предыдущих примерах. В этом параграфе будет подробно рассмотрен синтаксис этих блоков, в каком порядке выполняются выражения в блоках, зачем нужны именованные блоки. Последовательные блоки обозначаются ключевыми словами ***begin*** и ***end***, а параллельные блоки – ключевыми словами ***fork*** и ***join***.

6.7.1 Последовательные блоки

Формальный синтаксис последовательных блоков выглядит следующим образом:

begin : [<block_name>]
    [<block_declaration>]
    <block_statements>
end

Имя блока **block_name** представляет собой идентификатор и в общем случае не обязательно. Имя служит двум целям: первая – оператором ***disable*** можно прервать выполнение только именованного блока. Вторая – только в именованном блоке можно производить объявление локальных переменных в поле **block_declaration**. В этом поле можно объявлять переменные следующих типов:

- Параметры
- Регистровые переменные
- Целые переменные
- Вещественные переменные

Правила по которым эти переменные объявляются идентичны правилам объявления этих переменных в модуле. Необходимо сказать несколько слов о видимости подобных переменных. Локальные переменные как и переменные модуля являются статическими, т.е. будучи объявленными и инициализированными они сохраняют свое значение в интервалах между выполнением блока в разное время. Имя блока дает механизм доступа к этим переменным на любом этапе **моделирования**. Без использования имени переменная, объявленная в блоке, извне блока не видна. В случае если переменная блока совпадает по имени с переменной модуля, то переменная модуля «закрывается» переменной блока и из блока не видима. При этом системы синтеза никакой ошибки не выдадут, поскольку эти переменные объявлены в разных именованных объектах. Эта особенность удобна в плане расширения пространства имен, но все же может внести некоторую путаницу. Пользоваться этим приемом можно, но необходимо внимательно отслеживать модификации таких переменных, чтобы точно знать какая же переменная изменяется в том или ином месте. Более

распространен другой прием: переменная с одним и тем же именем может объявляться в разных блоках, но никогда не объявляться в модуле, простейший пример – переменная цикла, блоков может быть много, а переменные цикла в этих блоках имеют одно и то же имя, и при этом никакой путаницы обычно не возникает, пространство имен не засоряется.

Как уже говорилось, выражения внутри последовательного блока выполняются одно за одним, последовательно. В случае когда временные задержки не устанавливаются все так и происходит. Если же в блоке реализованы временные задержки, ситуация несколько меняется. Разумеется, выражения вродолжают выполнятся последовательно, но переход к следующему выражению выполнится только после выполнения предыдущего, т.е. с учетом задержки. В качестве подобного примера можно посмотреть листинг 5-6. Определенная сложность в таких блоках есть – при построении временной диаграммы время считается относительно к предыдущему выражению. Полное время выполнения блока расчитывается как сумма задержек по ходу выполнения алгоритма. Использование временных задержек не имеет смысла при синтезе модуля – системы синтеза игнорируют временные задержки. Можно, конечно, сказать, что они нужны для точного моделирования подобного модуля, но есть один момент, связанный с внеблочными присвоениями. Как уже говорилось, внеблочные присвоения выполняются на момент выхода из блока, что может привести к некорректным результатам моделирования (в лучшем случае просто будут игнорироваться). Отсюда следуют два вывода: первый – при моделировании синтезируемого модуля следует избегать временных задержек в операциях внеблочного присвоения; второй – при создании тестовых модулей, задающих временные диаграммы использование внеблочных присвоений недопустимо.

6.7.2 Параллельные блоки

Синтаксис параллельных блоков аналогичен синтаксису последовательных и отличается только ключевыми словами при объявлении блока:

fork : [<block_name>]
    [<block_declaration>]
    <block_statements>
join

Есть отличие при объявлении переменных блока: помимо перечисленных в предыдущем пункте в подобном блоке могут быть объявлены переменные типов ***time*** и ***event***. Параллельные блоки редко применяются для реализации синтезируемых модулей, поскольку не поддерживаются большинством систем синтеза (кстати переменные типов ***time*** и ***event*** тоже). Поскольку все операторы параллельного блока выполняются одновременно (конкурируют между собой), то использование этих блоков без механизмов управления временными задержками (Delay control) и управления событиями (Event control) особого практического смысла не имеет. Поэтому эти блоки и поддерживаюся в основном системами моделирования. Главное отличие параллельных блоков с временными задержками от последовательных блоков заключается в том, что задержки указывают относительное время от начала выполнения блока, а не от момента выполнения какого либо оператора. Опять посмотрим на листинг 5-6: в данном варианте блок последовательный, поэтому суммарное время выполнения составит 60 временных единиц (Подробнее, что такое временная единица рассказано в разделе «**Директивы компилятора**», директива

`timescale). Если в этом примере использовать параллельный блок то время его выполнения станет 15 временных единиц, по самой долгой задержке. В данном примере будет иметь место конфликт, так как через 15 единиц времени переменной Sig одновременно будут присвоены значения лог.1 и лог.0. Можно сделать вывод, что подобные конкурирующие конструкции использовать не стоит. Также не стоит делать и внеблочные присвоения.

6.7.3 Иерархические имена переменных

Последовательные и параллельные блоки могут быть вложены один в другой. В листинге 6-11 приведен пример многократного вложения блоков:

Листинг6-11:

```
module test();
  always @(posedge CLK or posedge RST)
    begin : mainblk
      reg Data_in, Strobe;
      if (RST)
        begin : rstblk
          integer int;
          ...
          ...
          fork : warnblk
            integer ErrCode, WarnCode;
            ...
            ...
          join
        end
      else
        ...
    end
endmodule
```

В этом примере сделано двойное вложение блоков: в блок mainblk вложен последовательный блок rstblk, в него в свою очередь вложен параллельный блок warnblk. Все три блока имеют свои собственные переменные. Выше говорилось, что на этапе моделирования можно получить доступ к любой переменной именованного блока. Это достигается использованием **иерархических имен** переменных. Правила построения иерархических имен просты: через символ точки (.) перечисляются все именованные объекты на пути к выбранной переменной. Допустим, нам надо из внешнего для модуля test объекта получить доступ к переменным всех вложенных блоков. Делается это так:

```
test.mainblk.Data_in                 // Доступ к переменной Data_in
test.mainblk.Strobe                  // Доступ к переменной Strobe
test.mainblk.rstblk.int              // Доступ к переменной int
test.mainblk.rstblk.warnblk.ErrCode  // Доступ к переменной ErrCode
test.mainblk.rstblk.warnblk.WarnCode // Доступ к переменной WarnCode
```

Следует обратить внимание, что доступ к переменным по иерархическим именам доступен только при моделировании кода, системы синтеза подобный доступ не поддерживают и выдают ошибку. Происходит это потому, что в реальном устройстве доступ через «крышу» не реализуем (Простая аналогия – если какой-либо сигнал внутри микросхемы не выведен на внешний контакт – читайте порт – то изменить его состояние непосредственно невозможно. В данном случае ситуация та же самая). Для систем синтеза доступ к переменным модуля возможен только через порты модуля.

7. Системные задачи и функции

В этом разделе кратко описываются системные задачи и функции, которые широко применяются при моделировании цифровых устройств. Системные функции служат различным целям, например, управление процессом моделирования, ввод/вывод форматированных данных, получение информации о реальном времени какого либо события – в общем они представляют мощный и удобный инструментарий разработчика. Системные функции никогда не применяются в синтезируемых кодах, поскольку чаще всего оперируют с несинтезируемыми объектами. Ниже представлен список наиболее часто применяемых системных функций. Необходимо отметить, что хотя такие объекты как системные функции и определены в стандарте языка, лишь некоторые из них в стандарте зарезервированы. Некоторые из системных функций могут не поддерживаться вашей системой моделирования, что может приводить к накладкам при переносе кода. Обязательным атрибутом системных функций является наличие символа ($) в начале их имени. Список системных задач и функций представлен ниже:

$bitstoreal	**$list**	**$rtoi**
$countdrivers	**$log**	**$save**
$display	**$monitor**	**$scale**
$fclose	**$monitoroff**	**$scope**
$fdisplay	**$monitoron**	**$showscopes**
$fmonitor	**$nokey**	**$showvariables**
$fopen	**$nolog**	**$showvars**
$fstrobe	**$printtimescale**	**$sreadmemb**
$fwrite	**$readmemb**	**$sreadmemh**
$finish	**$readmemh**	**$stime**
$getpattern	**$realtime**	**$stop**
$history	**$realtobits**	**$strobe**
$incsave	**$reset**	**$time**
$input	**$reset_count**	**$timeformat**
$itor	**$reset_value**	**$write**
$key	**$restart**	

В этом разделе задачи и функции разделены по их фунциональному назначению.

7.1 Функции ввода/вывода данных

7.1.1 Вывод данных на консоль системы моделирования

К этой группе относятся функции вывода той или иной информации на консоль системы моделирования. Это могут быть некоторые сообщения по ходу выполнения моделирования, сообщения об ошибочных ситуациях, некоторые числовые данные (текущий отсчет системного времени, номер теста и т.д.)

$display (P1, P2, P3, ...);

Эта системная задача предназначена для вывода на консоль форматированной информации в виде текстовой строки, подобно функции printf в языке C. P1, P2, P3 называются параметрами задачи и выводятся в том же порядке. После вывода всех

параметров задача добавляет символ новой строки. Параметры могут быть строковыми литералами с атрибутами форматирования, а также дополнительными символами предваряемые символом \. Дополнительные символы и атрибуты форматирования рассматриваются в пункте «**Формат вывода данных**» этого параграфа.

Пример: **$display** ("DReg value is %h", DReg);

$write (P1, P2, P3, ...);

Системная задача **$write** почти идентична по своему назначению задаче **$display**, но отличается тем, что не добавляет символа новой строки. В следующем примере будет выведено точно такое же сообщение, строка же будет переведена за счет символа (\n).

Пример: **$write** ("DReg value is %h\n", DReg);

$strobe (P1, P2, P3, ...);

Эта системная задача также предназначена для вывода форматированной информации на консоль, но ее главное отличие в том, что она всегда привязывается к конкретному событию и будет выводить информацию всякий раз как только это событие происходит в процессе моделирования. Вывод данных производится точно также, как и **$display**, включая все атрибуты форматирования и т.д.

```
Пример:     forever @ (posedge CLK)
                $strobe ("DReg value is %h", DReg);
```

Эта задача будет выводить состояние DReg строго по фронту тактового сигнала, и гарантировано, что это значение было установлено на момент прихода фронта.

$monitor (P1, P2, P3, ...);

Системная задача **$monitor** также действует подобно **$display**, но имеет следующие отличия: она выполняется всякий раз, когда изменяется значение одного или нескольких параметров, за исключением параметра определяемого функциями **$time**, **$stime** и **$realtime**; в один момент времени может действовать только одна задача $monitor с определенным списком параметров, в ходе моделирования может быть снова введена задача **$monitor** с другим списком, что отменит мониторинг предыдущих параметров; если два или более параметров изменят свое значение единомоментно, вывод их значений будет осуществлен в одной строке – задача выведет значения только один раз; задача может быть введена с пустым списком параметров, что отменит мониторинг предыдущих параметров. В отличие от **$strobe** эта задача выводит значения синхронно с метками системного времени (например возвращаемые функцией **$time**).

```
Пример:     $monitor ($time, , "DReg value is %h", DReg);
            #5 DReg = 8'h5A;
            #6.5 DReg = 8'76;

Результат:  5 DReg value is 5A
            7 DReg value is 76
```

$monitoroff

Эта задача предназначена для отлючения режима мониторинга, будучи введена, она не отменяет список параметров предыдущей задачи **$monitor**, но приостанавливает ее выполнение, что позволяет динамически управлять процессом вывода данных.

$monitoron

Эта задача предназначена для включения режима мониторинга, будучи введена, она возобновляет выполнение предыдущей задачи **$monitor**, после отключения мониторинга задачей **$monitoroff**. По умолчанию система моделирования на момент запуска имеет установленный флаг, разрешающий мониторинг (аналогично выполнению задачи **$monitoron**). Обе эти задачи вводятся без параметров.

7.1.2 Файловый ввод/вывод данных моделирования

К этой подгруппе задач и функций относятся задачи открывающие файл для ввода или вывода данных, для закрытия файла и собственно задачи ввода и вывода.

$fopen ("<name_of_file>");

Эта системная функция открывает файл с именем указанным в качестве параметра и возвращает дескриптор открытого для записи файла. Если дескриптор равен нулю, то открыть файл не удалось. Дескриптор представляет собой 32-х разрядное целое значение в котором лишь один бит установлен в лог.1 (кроме нулевого бита, он резервирован для вывода на консоль). Таким образом можно открыть до 31 файла. Использование такой системы позволяет выводить данные в несколько файлов одновременно с выводом на консоль как показано в примере.

Пример:
```
integer dat1, dat2, dat3, allfiles;
dat1 = $fopen("Data1.txt");
dat2 = $fopen("Data2.txt");
dat3 = $fopen("Data3.txt");
allfiles = 1 | dat1 | dat2 | dat3;
$fdisplay (allfiles, "DReg value is %h", DReg);
```

Значение DReg будет выведено во все три файла и на консоль системы моделирования.

$fopen ("<name_of_file>", "<access_modificator>");

Эта системная функция практически идентична предыдущей, но содержит модификатор доступа. Эта функция не прописана в стандарте 1995 г. и может не поддерживаться некоторыми системами моделирования. Есть также отличия и в возвращаемом дескрипторе. К сожалению, на данный момент не достаточно информации по дескриптору файла, поэтому для вывода лучше пользоваться первым вариантом. Модификаторы доступа бывают: r – файл открыт только для чтения, w – файл открыт для записи и чтения, a – файл открыт для дополнения и чтения. Пользуясь

этой функцией, однако лучше всего либо читать, либо создавать новый файл (w), либо добавлять информацию.

$fclose(<descriptor>);

После выполнения необходимых операций с файлом его необходимо закрыть. Для этого используется системная задача **$fclose**. С помощью этой задачи можно закрыть либо отдельный файл, дескриптор которого передается в качестве параметра, либо сразу группу файлов, передавая многоканальный дескриптор (как allfiles в предыдущем примере). Закрывая группу файлов необходимо убедиться, что все эти файлы еще не закрыты.

Пример: **$fclose** (allfiles);

Эта команда закроет все файлы, открытые в предыдущем примере.

$fdisplay (<descriptor>, P1, P2, P3, ...);
$fwrite (<descriptor>, P1, P2, P3, ...);
$fstrobe (<descriptor>, P1, P2, P3, ...);
$fmonitor (<descriptor>, P1, P2, P3, ...);

Эти системные задачи предназначены для вывода данных в файл, дескриптор которого передается в качестве первого параметра. Все эти задачи действуют идентично задачам **$display**, **$write**, **$strobe** и **$monitor**, поэтому подробно на них останавливаться не стоит. Вывод может осуществляться как в одиночный файл, так и в группу файлов используя многоканальный дескриптор.

Пример:
```
$fdisplay (dat1, "DReg value is %h", DReg);
$fwrite ((dat1 | dat 2), "DReg value is %h\n", DReg);
forever @(posedge CLK)
        $fstrobe (dat3, "DReg value is %h", DReg);
$fmonitor (allfiles, "DReg value is %h", DReg);
```

$readmemb ("<name_of_file>", <memory_name>);
$readmemb ("<name_of_file>", <memory_name>, <start_addr>);
$readmemb ("<name_of_file>", <memory_name>, <start_addr>, <end_addr>);
$readmemh ("<name_of_file>", <memory_name>);
$readmemh ("<name_of_file>", <memory_name>, <start_addr>);
$readmemh ("<name_of_file>", <memory_name>, <start_addr>, <end_addr>);

Эти системные задачи предназначены для чтения данных из указанного первым параметром файла и загрузки этих данных в указанный массив (блок памяти). Эти задачи могут быть выполнены в любое время в процессе моделирования. Текстовый файл может содержать следующие элементы:

- Пробелы, табуляцию, перевод строки
- Комментарии обоих типов
- Двоичные (для **$readmemb**) или шестнадцатеричные (для **$readmemh**) числа

Числа в файле не должны содержать разрядность и указатель формата (8'h5A будет неправильно, правильно 5A). Числа могут содержать символы x и X (неопределенное состояние), z и Z (состояние высокого импеданса), а также символ подчеркивания (_). Эти символы используются также как и в обычном описании чисел. Числа должны быть разделены пробелами, табуляцией или переводом строки. Для разделения чисел могут использоваться и комментарии. В этих задачах адресом является индекс массива, который используется для хранения данных. В процессе чтения числа последовательно присваиваются элементам массива. Разрядность чисел должна соответствовать разрядности элементов массива. Эти задачи могут не содержать адресов (индексов) массива с которых начинается и заканчивается запись. В этом случае желательно, чтобы длина массива совпадала с количеством чисел в файле. Запись начнется со стартового адреса по умолчанию – индекса в левой части объявления диапазона массива (если массив объявлен с диапазоном [0 : 255] то с индекса 0). В общем случае желательно, чтобы количество чисел в файле совпадало с количеством элементов массива между указанными адресами (Если адрес конечного элемента не указан, то за конечный адрес принимается индекс последнего элемента массива). В противном случае будет выдано предупреждение, а чтение будет продолжаться до тех пор, пока не будет заполнен массив или не будет обнаружен конец файла. Также диапазон адресов можно указать в самом файле данных используя символ (@) за которым следует шестнадцатиричное число. Пробелы между числом и символом (@) не допускаются. Такая спецификация указывает стартовый индекс массива. В файле данных может быть несколько подобных спецификаций. Если стартовый адрес указан и в файле данных и при вызове задачи, то адрес в файле данных должен укладываться в диапазон, определенный при вызове задачи, в противном случае будет выдана ошибка и процесс чтения будет прекращен. Ниже приведен пример составления файла данных (Пример1) и пример использования этих системных задач (Пример 2).

Пример 1:

```
// data.txt file
@A              // Start address is 10
01101101        // Number 1
01111101
01101001
* * * *
* * * *
11101101
01101101
01101111
01111001
01001101        // Number 256
```

Пример 2:

```
reg [7:0] mema [1:256];
reg [7:0] memb [1:256];
reg [7:0] memc [1:256];

$readmemb ("data.txt", mema);
$readmemb ("data.txt", memb, 1, 256);
$readmemb ("data.txt", memc, 20, 256);
```

Первый вызов задачи осуществит загрузку данных начиная с первого числа в файле, но начиная с 10-го элемента массива mema, часть данных останется незагруженной,

что вызовет соответствующее предупреждение. Второй вызов задачи загрузит все 256 чисел в массив memb. Третий вызов задачи вызовет ошибку, поскольку стартовый адрес указанный в файле лежит вне диапазона при вызове задачи.

7.1.3 Строковый ввод данных моделирования

$sreadmemb (<memory_name>, <start_addr>, <end_addr>,
<string1>, <string2>,... <stringN>);
$sreadmemh (<memory_name>, <start_addr>, <end_addr>,
<string1>, <string2>,... <stringN>);

Эти системные задачи действуют подобно задачам **$readmemb** и **$readmemh**, но производят загрузку массива не из файла, а из строк (stringN) заданных в списке параметров. Формат строк должен быть тот же самый как и в файле данных для задач **$readmemb** и **$readmemh**.

7.1.4 Формат вывода данных

С помощью системных задач, предназначенных для вывода данных можно выводить числа в различных представлениях, строки, отсчеты времени, иерархические имена блоков и т.д. Ниже представлен список атрибутов форматирования, предназначенных для вывода этих значений. Эти атрибуты располагаются в символьной строке (ограниченной двойными кавычками), для каждого атрибута (кроме %m и %M) в списке параметров задачи должно быть указано значение. Порядок параметров должен совпадать с порядком атрибутов в строке.

%h %H Отображение чисел в шестнадцатиричном формате;
%d %D Отображение чисел в десятичном формате;
%o %O Отображение чисел в восьмеричном формате;
%b %B Отображение чисел в двоичном формате;
%c %C Отображение чисел в формате ASCII символов;
%v %V Отображение "мощности" источника цепи;
%m %M Отображение иерархического имени;
%s %S Отображение строки;
%t %T Отображение времени в текущем формате;
%e %E Отображение вещественных чисел в экспоненциальном формате;
%f %F Отображение вещественных чисел в десятичном формате;
%g %G Отображение вещественных чисел в экспоненциальном или
десятичном формате в зависмости какая запись короче;

По умолчанию параметры выводятся в десятичном формате, например так как выводилось системное врема в примере для задачи **$monitor**.

Размер выводимых параметров в форматах %h, %H, %d, %D, %o, %O, %b и %B обычно определяется автоматически. Если необходимо вывести значение 16-и разрядной переменной, то в шестнадцатиричном формате будет зарезервировано 4 позиции в строке (макс. значение FFFF), в десятичном формате 5 позиций (65535), в восьмеричном 6 позиций (177777), в двоичном, разумеется 16. Незначащие нули слева опускаются только в десятичном формате, однако заменяются пробелами. В остальных форматах нули не пропускаются. Для того, чтобы пропустить незначащие нули или пробелы используют следующий прием: между символом (%) и спецификатором

формата вставляют 0 (например %0h, %0o, %0B), в данных выведенных в таком формате незначащие нули или пробелы будут пропущены.

Когда значение выводимой переменной содержит неизвестные состояние или состояние высокого импеданса то при отображении действуют следующие правила:

- В десятичном формате –
- Если все биты в находятся в неопределенном состоянии отображается одиночный символ x (строчной буквой).
- Если все биты в находятся в состоянии высокого импеданса отображается одиночный символ z (строчной буквой).
- Если лишь часть битов находится в неопределенном состоянии отображается одиночный символ X (прописной буквой).
- Если лишь часть битов находится в состоянии высокого импеданса отображается одиночный символ Z (прописной буквой).

В шестнадцатеричном и восьмеричном форматах –

- Каждая группа из 4-х бит представляется как одиночная шестнадцатиричная цифра; каждая группа из 3-х бит представляется как одиночная восьмеричная цифра.
- Если все биты в группе находятся в неопределенном состоянии, то эта цифра отображается символом x (строчной буквой).
- Если все биты в группе находятся в состоянии высокого импеданса, то эта цифра отображается символом z (строчной буквой).
- Если часть битов в группе находятся в неопределенном состоянии, то эта цифра отображается символом X (прописной буквой).
- Если часть битов в группе находятся в состоянии высокого импеданса, то эта цифра отображается символом Z (прописной буквой).

В двоичном формате каждая цифра отражается символами 0, 1, x или z.

Формат вывода «мощности» источника цепи (%v) отображается тремя символами. Первые два символа – мнемоника «мощности» источника. Третий символ отображает текущее логическое состояние. В списке параметров должна быть ссылка на имя цепи. Эта ссылка не может может быть выражением или ссылкой на одиночный разряд шины (например data[2] не допускается). Список мнемоник приведен ниже:

Strength Mnemonic	**Strength Name**	**Strength Level**
Su	Supply driver	7
St	Strong driver	6
Pu	Pull driver	5
La	Large capacitor	4
We	Weak driver	3
Me	Medium capacitor	2
Sm	Small capacitor	1
Hi	High impedance	0

В некоторых случаях мнемоника может отображаться в виде пары десятичных цифр – в этом случае мнемника отображает диапазон «мощности» в случае нескольких источников на цепи. Более подробно этот вопрос рассматривается в [1].

Текущее логическое состояние отображается символами

0 Значение лог.0
1 Значение лог.1
X Неизвестное состояние
Z Состояние высокого импеданса
L Состояние высокого импеданса или лог.0
H Состояние высокого импеданса или лог.1

При выводе иерархического имени используется формат %m. В списке параметров для вывода ничего не указывается, поскольку выводится иерархическое имя блока внутри некоторого модуля (вывод иерархических имен сигналов не поддерживается). Это полезно когда в проекте существует много эземпляров одного и того же модуля. В ирерархическом имени будет отображено имя конкретного экземпляра в котором встретился вызов задачи.

Формат %s подразумевает, что ASCII коды должны выводится в виде символов. Параметр ассоциированный с эти форматом воспринимается как последовательность байтовых ASCII-кодов, при этом каждые 8 бит представляются как одна буква. Если параметром является переменная, то ее значение выравнивается по правому краю диапазона, т.е. самый крайний бит справа является младшим значащим битом (lsb – least significant bit) последнего символа строки. Никаких символов окончания строки не предусматривается.

С помощью символа \ можно ввести в строку ряд специальных символов:

\n символ перевода строки;
\t символ табуляции;
\\ символ (\);
\" символ (");
\o символ определяемый тремя восьмеричными цифрами;
%% символ (%).

Пример: **$display** ("\\\t%%\n\"\123");

Результат: \ %
"S

7.2 Функции и задачи временного масштабирования

$time
$stime

Эта системные функции возвращают 64-х разрядное целое типа ***time*** которое отражает отсчет во временных единицах начиная с начала моделирования. Временные

единицы устанавливаются директивой **`timescale**. Возвращаемое функциями значение будет округлено к ближайшему целому числу.

Пример:
```
`timescale 10ns/1ns
module test();
parameter del = 1.55;
        initial begin
                $display ("Time =", , $time);
                #del $display ("Time =", , $time);
                #del $display ("Time =", , $time);
        end
endmodule
```

Результат:
```
Time = 0
Time = 2
Time = 3
```

Поскольку директивой **`timescale** временная единица определена как 10ns, то полученные отсчеты необходимо умножать на 10, чтобы получить отсчет в наносекундах. Разницы между этими системными функциями нет никакой, возможно имя **$stime** резервировано для будущего расширения возможностей этой функции.

$realtime

Эта функция также возвращает отсчет во временных единицах, но в виде вещественного числа. Точность возвращаемого значения зависит от второго параметра директивы **`timescale**.

Пример:
```
`timescale 10ns/1ns
module test();
parameter del = 1.55;
        initial begin
                $display ("Time =", , $realtime);
                #del $display ("Time =", , $realtime);
                #del $display ("Time =", , $realtime);
        end
endmodule
```

Результат:
```
Time = 0.0
Time = 1.6
Time = 3.2
```

$printtimescale ([<hierarchical_name>]);

Эта системная задача выводит на консоль размер и точность временных единиц, заданных для отдельного модуля. Параметр этой задачи является опциональным. Если он не задан, то результат будет выведен для модуля, заданного системной задачей **$scope**. Если же параметр передан, то выведенный результат будет относится к модулю, иерархическое имя которого передано в качестве параметра. Результат выводится в следующем формате:

```
Time scale of (<module_name>) is <unit> / <precision>
```

Пример:

```
`timescale 1ms/1us
module a_dat();
initial $printtimescale(b_dat.c1);
endmodule

`timescale 10fs/1fs
module b_dat();
        c_dat c1();
endmodule

`timescale 10ns/1ns
module c_dat();
        <statements>
endmodule
```

Результат: Time scale of (b_dat.c1) is 10ns / 1ns

$timeformat (<unit_numbers>, <precision_number>,
<suffix_string>, <minimum_field_width>);

Эта системная задача выполняет две функции: во-первых, как будет выводится информация под форматом %t; во-вторых определяет временные единицы при интерактивном вводе задержек. Параметр **unit_number** определяет временные единицы как и директива **`timescale**, но формат этого параметра другой. Этот параметр представляет собой целое число от -15 до 0. В следующем списке поставлено соответствие между форматом директивы **`timescale** и значением этого параметра.

Unit Number	**Time Unit**	**Unit Number**	**Time Unit**
0	1s	-8	10ns
-1	100ms	-9	1ns
-2	10ms	-10	100ps
-3	1ms	-11	10ps
-4	100us	-12	1ps
-5	10us	-13	100fs
-6	1us	-14	10fs
-7	100ns	-15	1fs

По умолчанию значение этого параметра устанавливается равным минимальному значению временной единицы в директивах **`timescale** в исходном коде. Параметр **precision** определяет количество знаков после десятичной точки в выводимом значении. По умолчанию этот параметр равен нулю. Третий параметр **suffix_string** определяет текстовую строку, выводимую непосредственно после значения временного отсчета (например, "ns"). По умолчанию это пустая строка. Последний параметр **minimum_field_width** определяет минимальное количество позиций отводимое под выводимое значение и текстовую строку, по умолчанию этот параметр равен 20. Установленные этой задачей значения будут действовать, пока не всретится другой вызов **$timeformat** с другими параметрами.

Пример:

```
`timescale 1ms/1us
module ctrl();
initial $timeformat(-9, 5, " ns", 10);
endmodule

`timescale 1fs/1fs
module a_dat();
initial
        #15043021 $display("Time = %t", $realtime);
endmodule
```

Результат: Time = 15.04302 ns

7.3 Функции и задачи управляющие процессом моделирования

$reset;
$reset (<stop_value>);
$reset (<stop_value>, <reset_value>);
$reset (<stop_value>, <reset_value>, <diagnostics_value>);

Эта системная задача предназначена для сброса системы моделирования в начальное состояние, т.е. временные отсчеты устанавливаются в нулевое состояние, все регистры и цепи устанавливаются в свое начальное состояние и начинается выполнение операций в блоках **initial** и **always**. Параметр **stop_value** определяет поведение системы моделирования после сброса, т.е. определяет переходит ли система к процессу моделирования немедленно или переходит в интерактивный режим. Нулевое значение этого параметра или отсутствие параметров вообще вызывает переход в интерактивный режим. Ненулевое значение вызывает перезапуск процесса моделирования. Параметр **reset_value** позволяет сохранить некоторое целочисленное значение после перезапуска системы моделирования. Поскольку все переменные принимают свое начальное состояние, другим способом это обеспечить нельзя. Сохраненное значение пожно получить воспользовавшись системной функцией **$reset_value**. Третий параметр **diagnostic_value** представляет собой целое число в качестве диагностического сообщения. Нулевое значение этого параметра подразумевает отсутствие сообщений.

$reset_value

Эта функция возвращает целочисленное значение, заданное параметром **reset_value** в предыдущем вызове задачи **$reset**.