Краткий конспект книги
Стемпковский А.Л., Семенов М.Ю. «Основы логического синтеза средствами САПР Synopsys с использованием Verilog HDL: учебное пособие». – М.: МИЭТ, 2005 -140 с.

Verilog HDL (Hardware Description Language) –
язык описания поведения цифровых устройств, был разработан фирмой Gateway Design Automaton как внутренний язык симуляции.
Cadence приобрела Gateway в 1989 г. и открыла Verilog для общего использования.
В 1995 г. был определен стандарт языка - Verilog LRM (Language Reference Manual), IEEE Std 1364-1995,
Последняя редакция стандарта - в 2001 г.
Подмножество языка, используемое для синтеза: IEEE Std. 1364.1 – 2002.
***
Существует три типа конструкций Verilog HDL:
- поддерживаемые средствами синтеза ИС);
- неподдерживаемые средствами синтеза;
- игнорируемые средствами синтеза.
Сами средства синтеза могут иметь какие-либо *особенности* по использованию синтезируемых конструкций в конкретную технологию,
некоторые конструкции могут *поддерживаться* одними средствами синтеза и *не поддерживаться* другими.
***
Синтаксис Verilog HDL
Исходный текстовый файл = последовательность символов, чувствительных к регистру:

- пробелы *(\b, \t, \n)* и комментарии *//, /*....*/* (игнорируются);
- операторы: *y = &a;* // унарный
  y = a+b; // бинарный
  y = c ? a : b; // триадный
- числа: *<**разрядность**>'<**формат**><**значение**>;*
  2'b10; *// 2-х битовое бинарное число;*
  12'habc; *// 12-ти битовое шестнадцатеричное;*
  -4'd5; *// 4-х битное десятичное отрицательное;*
  8'o7; *// 8-ми битное восьмеричное;*

x *//неопределенное состояние*
z *// высокий импеданс:*
8'hax; // 4 мл. бита – неопределнность
4'b0z; // эквивалентно 4'b000z
0.001; *// или 1e-3 – вещественные (real) не синтезируются*
***
Синтаксис Verilog HDL
- строки (string) – последовательность 8-ми битных ASCII-символов, которые обычно испся для вывода информации при моделировании;

- идентификаторы – уникальные имена объектов, к которым можно обращаться в проекте (имена модулей, цепей, переменных);
- ключевые слова – определяют специальные языковые конструкции, должны быть написаны в нижнем регистре

input *clk;* // ***input*** – ключевое слово, *clk* – идентификатор
reg *q*; // ***reg*** - ключевое слово, *q* – идентификатор
wire *select*; // ***wire*** - ключевое слово, *select* – идентификатор
***
Ключевые слова
always

and
assign
begin
buf
bufif0
bufif1
case
casez
casex
cmos
deassign
default
defparam
disable
event
edge
else
end
endcase
endmodule
endfunction
endprimitive
endspecify
endtable
endtask
for
force
forever
fork
function
highz0
highz1
if
initial
inout
input
integer
join
large
macromodule
medium
module
nand
nmos
not
notif0
notif1
negedge
nor
or
output
parameter

pmos
posedge
primitive
pull0
pull1
pullup
pulldown
rcmos
reg
release
repeat
rpmos
rnmos
rtran
rtranif0
rtranif1
strong0
strong1
supply0
supply1
scalared
small
specify
specparam
strength
table
task
tran
time
tranif0
tri
tri0
tri1
tranif1
trior
triand
trireg
vectored
wait
wand
weak0
weak1
while
wire
wor
xnor
xor
***

Типы данных

Используемые значения переменных (алфавит):

Verilog обрабатывает всего *четыре* значения переменной:

1) “1” - логическая единица
2) “0” - логический ноль
3) “z” - состояние высокого импеданса (исп-ся при моделировании)
4) “x” - неизвестное логическое состояние (исп-ся при моделировании
***

Цепи (net data type) - для представления реальных соединений в схеме, (например, соединения между вентилями или подключенными модулями).
К цепям необходимо постоянно прилагать непрерывное воздействие, т.е. цепи не могут хранить присвоенную им величину.
Цепь принимает значение выходной величины источника (драйвера), к которому обязательно должна быть подключена.
Обозначения:

wire *out_buf;* // декларация цепи типа wire
tri *out_tri_buf;* // tri – несколько источников сигналов
wand, wor, (для моделирования монтажных функций)
tri0, tri1, (для моделирования резисторов к 0 или 1)
trireg (для моделирования емкостных цепей)

***

Переменные (variable data type) - для обозначения элементов языка, способных хранить данные, т.е .переменная сохраняет величину до тех пор, пока ей не будет присвоена другая величина

reg *q_ff;*
/* объявление *q_ff* хранящих свое значение между процедурными присваиваниями */
/* для описания аппаратных регистров и комбинационной логики*/
integer *i_count*; // объявление целочисленных переменных со знаком
// только для манипуляций с числами
time /*для объявления и хранения временной переменной – текущего времени моделирования*/
***

Векторы

Цепи и переменные типа reg могут объявляться как векторы, разрядность вектора указывается в квадратных скобках после ключевого слова, причем левое всегда старший значащий разряд (MostSB), правое – младший (LeastSB):

wire *[7:0] vector_net*; // 8-ми битный вектор типа wire
reg *[1:8] vector_var*; // 8-ми битный вектор типа reg
vector_net[3]; // 3-ий бит вектора *vector_net*
vector_var[5:3]; // 5, 4 и 3 биты вектора *vector_var*
***

Массивы
Массивы определяются для цепей и переменных, которые могут быть скалярами и векторами. Каждый элемент массива (слово) может быть как однобитным (скаляр) так и многоразрядным (вектор)
reg *array_a[0:7];* // массив из 8-ми однобитных переменных типа reg
reg *[31:0] array_b[0:15];* // массив 16-ти 32-х битных векторов типа reg
// обращение к 32-битному слову с адресом 10 из массива array_b:
wire*[31:0] from_array_b=array_b[10];*
NB: <u>важно не путать</u> массивы и векторы.
*Вектор – **<u>один</u>** элемент, который может иметь разрядность больше единицы.*

*Массив - **множество элементов**, которые могут иметь разрядность 1 бит или n бит, м.б. многомерными*
***
Примеры массивов:

объявление памяти:
reg *mem_1bit [0:1023];* // память из 1K однобитных слов
reg *[7:0] mem_8bit [0:1023];* // 1K восьмибитных слов - векторов типа reg
Многомерные массивы:
wire*[3:0] farray_c [0:255] [0:127];* // двумерный массив их 4-х битовых
// слов типа wire
/* выборка 3-го и 2-го битов элемента с адресом [255] [127] из двумерного массива array_c */
wire *[1:0] from_array_c=array_c [255] [127] [3:2]:*
***

Параметры:
ключевое слово ***parameter***

для объявления констант внутри модуля
parameter *WIDTH=8;* // объявлена константа разрядности слова
parameter *ADDR=32;* // константа разрядности адреса
// объявление параметризированной памяти:
reg *[WIDTH-1:0] memory [0:ADDR-1];*
***

2.3. Системные функции и директивы
Запись: **$<ключевое слово>**
Примеры:
$display(v1,v2,…,vn) // для вывода на экран значений переменных, // строк и выражений
Форматы вывода:
%d или %D *– вывод на экран в десятичном формате*
%b или %B *– вывод на экран в двоичном формате*
%o или %O *– вывод на экран в восьмеричном формате*
%h или %H *– вывод на экран в шестнадцатеричном формате*
%m или %M *– вывод на экран в иерархического имени*
%t или %T *– вывод на экран в текущего времени моделирования*

$display *(“Synopsys course”);*
>Synopsys course
reg *[7:0] addr;*
$display *(“Address is %b”, addr);*
>Address is 10101010;
***

Директивы компилятора:
Запись:

`<имя макроса>

Примеры:

`include // для включения какого-либо файла или фрагмента в // исходный код

`define *WIDTH 32;* // для определения текстового макроса
reg *[WIDTH-1:0] data* // объявление разрядности [31:0]
***
Директивы компилятора:
`timescale <единицы измерения>/<точность>
// аргумент ***<единицы измерения>*** *определяет, в каких единицах*
// будет проводиться моделирование и каких единицах будут
// измеряться задержки
// аргумент ***<точность>*** *определяет с какой точностью эти задержки будут округляться*
`timescale *100ns/1ns*
module *example;* //декларация модуля
reg *test_signal;* //декларация тестового сигнала
initial
begin
test_signal=1'b0;
#5 test_signal=1'b1; // 5 ед. времени соответствуют
// 5*100ns = 500ns
end
endmodule
***
Модуль - *основная иерархическая единица Verilog-описания.*
Модуль обеспечивает необходимое взаимодействие с другими модулями посредством интерфейсных портов (входы, выходы или двунаправленные порты)
Для построения иерархической структуры разрешается подключение других модулей внутри описываемого модуля (подключение модулей "instance")
module *<имя_модуля> (<список внешних портов модуля>) <декларация параметров>*
<объявление входных портов>
<объявление выходных портов>
<объявление двунаправленных портов>
<объявление цепей>
<объявление переменных>
<подключение ячеек>
<подключение модулей нижнего уровня>
<непрерывные присваивания - assign>
<always и initial – блоки и поведенческие конструкции>
<функции и процедуры>
endmodule
***
Порты *– обеспечивают интерфейс модуля с внешним окружением, которое может взаимодействовать с модулем через указанные порты .*
Ключевые слова декларации портов:
input // входной порт
output // выходной порт
inout // двунаправленный порт
Пример: объявление модуля и портов D-триггера

module *d_flip_flop (d, clk, q);*

input *d;*
input *clk;*
output *q;*
reg *q;*
< код с описанием функций модуля>
endmodule
***

Правила подключения портов:

– все объявленные порты имеют по умолчанию тип ***wire****. Входные и двунаправленные порты могут быть только* ***wire,*** *т.к. они транслируют сигналы от внешних источников в данный порт. Если выходной порт хранит присваиваемую ему величину, то его следует назначить как* ***reg****.*

При создании иерархических описаний:

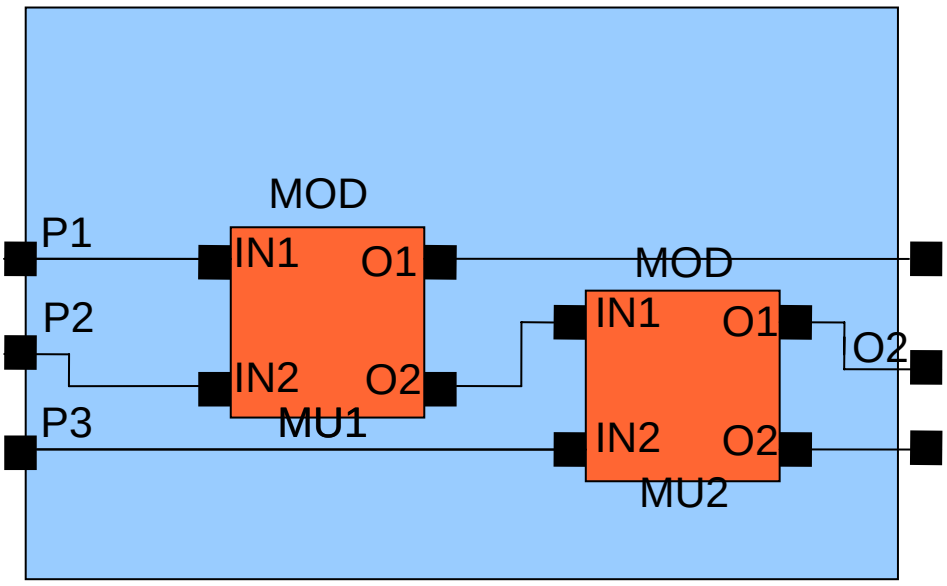
- *входы (input) внутри модуля могут быть только* ***wire****, но подключаться извне могут как к цепям, так и к переменным;*
- *выходы (output) внутри модуля могут быть* ***wire****, так и* ***reg****, а подключение извне может быть только к цепям;*
- *двунаправленные порты (inout) внутри модуля могут быть только цепью* ***(wire)****, и подключаться извне должны только к цепям*

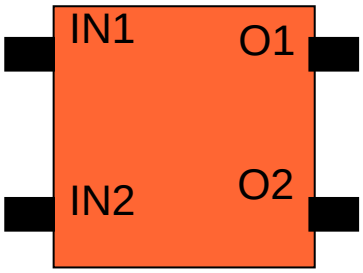
***

Пример объявления типов портов и сигналов:

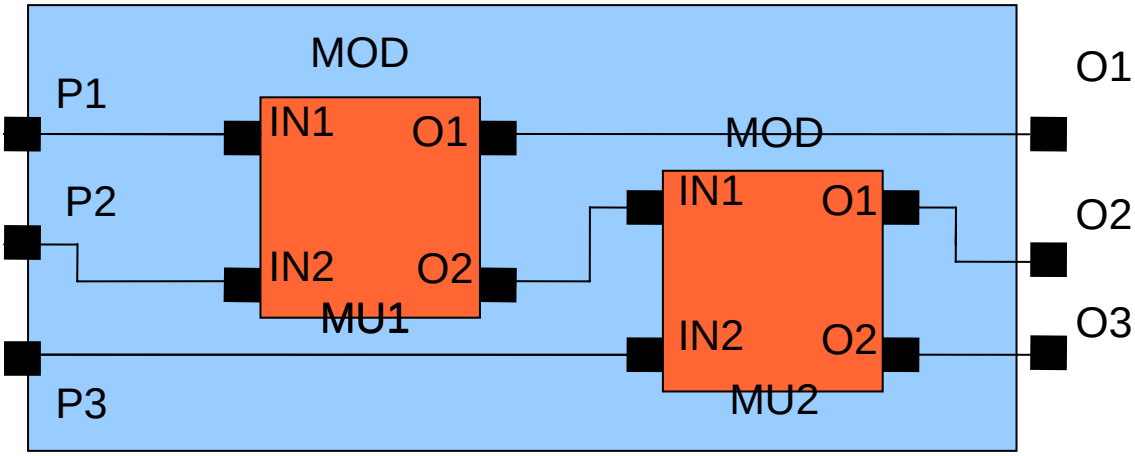
module *d_flip_flop (*
input *wire d,*
input *wire clk,*
output *reg q*
);
<внутреннее тело модуля>
endmodule
***

Иерархия






```
module MOD (IN1, IN2, O1, O2);
input IN1, IN2;
output O1, O2;
///<------------->
endmodule
```




```
module TOP_LEVEL (P1, P2, P3, O1,O2,O3);
input P1, P2, P3;
output O1,O2,O3;
wire net1;
MOD MU1 (P1, P2, O1, net1);
MOD MU2 (.IN1(net1), .IN2(P3), .O1(O2), .O2(O3));
///<------------->
endmodule
```

одноразрядный полусумматор:

```
module half_adder (a, b, co, s);
        input a;
        input b;,
        output co;
        output s;
        <внутреннее тело модуля>
endmodule
```

Пример объявления типов портов и сигналов:

```
module d_flip_flop (
        input wire d,
        input wire clk,
        output reg q
```

```
);
        <внутреннее тело модуля>
endmodule
```

Создание иерархического описания (подключение (instantiation) - модулей нижнего уровня описания):
первый способ: внешние сигналы следуют в таком же порядке, что и списке внешних портов для этого модуля;
второй способ (предпочтительнее): порты подключаются по имени, порядок записи м.б. любым.
Пример: 2-х битовый сумматор.

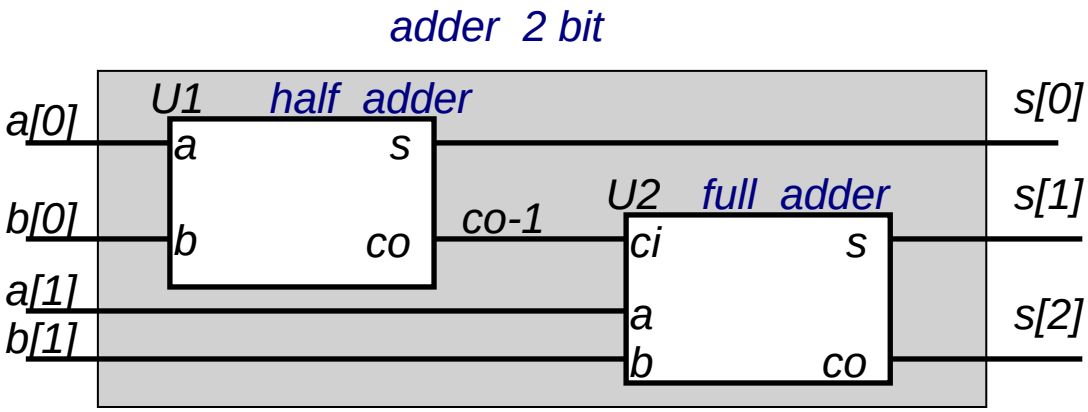
```
module adder_2bit (a, b, s);
        parameter W=2;

        input [W-1:0] a;        // 1-ый операнд
        input [W-1:0] b;        // 2-й операнд
        output [W-1:0] s;       // результат

        wire co_1;              // внутренняя цепь
// по порядку портов:
        half_adder U1(a[0], b[0], co_1, s[0]);
// по имени портов:
        full_adder U2 (.ci(co_1), .a(a[1]), .b(b[1]), .co( s[2]), .s(s[1]));

endmodule
```

схема:



Возможно, что какие-либо порты будут не подключены (например, дополнительные выходы для отладки). В этом случае:
- при подключении по порядку следует отделять место их нахождения запятыми:
half_adder U1(a[0], b[0], , s[0]); .// выход co не подключен
- при подключении по имени - не указывать
full_adder U2 (.ci(co_1),. a(a[1]), .b(b[1]),. s(s[1]));

Получение доступа к любому внутреннему сигналу (при моделировании) осуществляется путем обращения по иерархическому имени:
adder_2bit.co_1 //обращение к сигналу co_1
adder_2bit.U1.<идентификатор цепи внутри полусумматора> //обращение к цепи внутри полусумматора

При подключении модулей параметры могут быть переопределены:
- явно оператором ***defparam***
- неявно непосредственно при подключении модуля

```
module parallel_reg (d, clk, q);
        parameter WIDTH=4;          // разрядность регистра = 4
        input [WIDTH-1:0] d;        // вход данных
        input clk;                  // тактовый вход
        output [WIDTH-1:0] q;       // выход регистра

        reg [WIDTH-1:0] q;          // переменная

                <внутреннее опсание модуля>

endmodule

//  описание модуля верхнего уровня иерархии
module top_level (...);
parallel_reg U1 (.d(net_d1), . clk(clk), . q(net_q1));
defparam U1.WIDTH=8;      // переопределение новое значение WIDTH=8
...
parallel_reg #(16) U2 (.d(net_d2), . clk(clk), . q(net_q2));

endmodule
```

ВЫРАЖЕНИЯ, ОПЕРАНДЫ И ОПЕРАТОРЫ

Выражения – конструкции языка, представляющие собой комбинации операндов и операторов для получения определенного результата.
Операнды:
- константы,
- переменные биты векторной переменной (вектора),
- цепи, биты векторной цепи (вектора),
- элемент массива,
- вызов функции, которая возвращает одно из перечисленных выше значений операнда.

Операторы – указывают, какие действия следует проводить с операндами.

Список операторов:

Операнды объединения и повторения:

```
// объявление операндов
reg         op1;
reg[1:0]    op2;
reg[2:0]    op3;
// пусть op1=1'b1; op2=2'b11; op3=3'b001;
res1= {op1,op2};            // результат res1=3'b1_11 тоже, что 3'b111 для удобства /
                            //компилятором знак "_" игнорируется
res2= {op1,op2[0],op3[2:1]};    // результат res2=4'b1_1_00;
res3= {2{op3}};             // результат res3=6'b001_001 (два раза op3);
res4= {op1,{2{op1,op3}}};   // результат res4=9'b1_1001_1001;;
```

Арифметические операторы:
- унарные (знаковые), отрицание "**!**" и свертка
- бинарные: сложение (+), вычитание (-) , умножение (*), деление (/),
возведение в степень (**), вычисление по модулю (%).

Если хотя бы один операнд равен ***x*,** то и результат всего выражения ***x.***

```
reg[3:0] op1, op2, op3;
integer op4, op5;

// пусть op1=4'b0101; op2=4'b0010; op3=4'b00x; op4=5, op5=3;
res1=op1+op2;           // res1=4'b0111=4'd7;
res2=op1+op3;           // res2=4'bxxxx;
res3=op1-op2;           // res3=4'b0011=4'd3;
res4=op1*op2;           // res4=4'b1010=4'd10;
res5=op4/op5;           // res5=1 (дробная часть отбрасывается);
res6=op1%op5;           // res6=2 (остаток от деления).
```

Операторы отношения (>, >=, <, <=)
Возвращают значение "0", если заданное отношение ложно и "1", если истинно. Если хотя бы один бит операндов содержит ***x*** или *z*, то результат будет равен неопределенности, т.е. ***x***.

Логические операторы И (&&) и ИЛИ (||):
всегда возвращают 1 бит (0 или 1, или, если хотя бы один бит операндов содержит ***x*** или *z*, то возвращают ***x***.).

Операторы равенства / неравенства (==, !=).

Эти операторы сравнивают операнды побитно, дополняя недостающие разряды нулями. Возвращают логическую “1”, если выражение истинно и логический “0”, если ложно, или, если хотя бы один бит операндов содержит ***x*** или ***z***, то возвращают ***x***.
Условные операторы ***(===, !==)*** сравнивают побитно операнды, которые в т.ч. могут содержать ***x*** или ***z***. Они не являются синтезируемыми и возвращают 0 или 1.

Побитовые операторы: **~(побитовое НЕ), &(побитовое И), | (побитовое ИЛИ) , ^(побитовое исключающее ИЛИ), ~^(побитовое исключающее ИЛИ-НЕ) .**
Выполняют операции с битами из каждого разряда, т.ч. и с ***x*** или ***z,*** дополняя недостающие нулями. Возвращают значения, имеющие разрядность самого большого операнда. Правила побитовых операций задаются соответствующими таблицами истинности:

Операторы свертки.
Выполняют побитовую операцию над векторным операндом. Возвращают один бит.

reg*[3:0] op1;*

// пусть op1=4’b0101;
res1=&op1; *// свертка по И: 0&1&0&1;* *res1=1’b0;*
res2=|op1; *// свертка по ИЛИ: 0|1|0|1;* *res2=1’b1;*
res3=~|op1; *// свертка по ИЛИ–НЕ: ~(0|1|0|1)* *res3=1’b0;*
res4=^op1; *// свертка по исключающему ИЛИ: 0^1^0^1* *res3=1’b0;;*
res5=op4/op5; *// res5=1 (дробная часть отбрасывается);*
res6=op1%op5; *// res6=2 (остаток от деления).*

Операторы сдвига <<, >>, <<<, >>>.
Осуществляют сдвиг векторного операнда вправо или влево.
Логические операторы сдвига **<<** и **>>,** арифметический сдвиг влево **<<<** сдвигают так, что освобождающиеся позиции (биты) заполняются нулями.
Арифметический сдвиг вправо **>>>** в зависимости от типа результата заполняет освободившиеся позиции нулями (переменные без знака) или старшего значащего бита (тип данных со знаком):

reg*[3:0] op1, res1;*
reg signed *[3:0] op2, res2;*

// пусть op1=4’b0001, op2=4’b1000;
res1=(op1<<2); *// результат без знака* *res1=4’b0100;*
res2=(op2>>>2); *// результат со знаком* *res2=4’b1110;*

Условный оператор (триадный)
<условие>?<выражение_1>:< выражение_2>;
Если условие истинно, то выполняется *выражение_1,* в противном случае - *выражение_2,* причем, если результат условия – неопределенность ***x*** или состояние с высоким импедансом ***z***, то результаты выражений 1 и 2 сравниваются побитно и возвращается x, в той позиции, где биты различны и то, значение бита, где они совпадают.
// 16-ти битный тристабильный буфер:
bus_addr [15:0] = en ? addr [15:0] : 16’bz;
// мультиплексор 2 в 1:
Res sel ? op1 : op0;
Приоритеты операторов:

По степени убывания от высшего к низшему (при равенстве приоритетов операторы выполняются слева по порядку записи):

1. + — *!* ~ (унарные)
2. **
3. * / %
4. + — (бинарные)
5. << >> <<< >>>
6. < <= > >=
7. == != === !==
8. & ~&
9. ^ ^~ ~^
10. | ~|
11. &&
12. ||
13. ?:

Рекомендуется использовать скобки.

Непрерывные присваивания (continuous assignments).
Используются для присваиваний каких-либо величин цепям.
Ключевое слово ***assign*** (может отсутствовать)
assign*(сила сигнала)#(задержка) <присваивание>*
По умолчанию необязательный параметр «сила сигнала» (drive strength) – “strong 1” (сильная 1) и“strong 0” (сильный 0). Необязательным также является параметр «задержка» (delay), который является не синтезируемым (не нужно указывать задержки при написании синтезируемого кода).
Правила:
- результатом присваивания (т.е. записанные слева) должны быть скалярные и векторные сигналы типа «цепь» или конкатенация скалярных и векторных цепей;
- непрерывные присваивания всегда отслеживают любые изменения операндов и результат после указанной задержки принимает соответствующее им значения;
- операнды в непрерывных присваиваниях (т.е. справа в записи) м.б. векторами или скалярами, причем как цепями, так и переменными.

Пример
явное непрерывное присваивание:
wire *res1;*
assign *res1=op1|op2;*

неявное непрерывное присваивание с задержкой на 10 ед. времени:
wire *#10 res1=op1|op2;*
(если изменения на входах происходят быстрее указанной задержки, то они не передаются).

Пример: 4-х разрядный сумматор

1. Структурное описание 4-х разрядного сумматора.

```
// Описание одноразрядного полусумматора:
module half_adder (a, b, co, s);
        input a, b;             // входы
        output co;              // перенос разряда
```

```
        output s;               // результат
assign s=a^b;
assign co=a&b;

endmodule

// Описание одноразрядного сумматора:
module full_adder (ci, a, b, co, s);
        input ci;               // входной перенос
        input a, b;             // входы
        output co;              // перенос разряда
        output s;               // результат


assign s=a^b^ci;
assign co=(a&b)|(a&ci)|(b&ci);

endmodule

//Структурное описание 4-х разрядного сумматора с подключением полусумматора и
одноразрядных сумматоров:

module adder_4bit (a, b, sum);

        input [3:0] a, b;       // входные операнды
        output [4:0] sum;       // результат
// объявления внутренних цепей
        wire co_1;              // внутренняя цепь переноса с 0-го разряда
        wire co_2;              // внутренняя цепь переноса с 1-го разряда
        wire co_3;              // внутренняя цепь переноса с 2-го разряда
//подключение полусумматора и сумматоров по имени портов:
        half_adder U1 (.a(a[0], .b(b[0]), .co(co_1), .s(sum[0]);
        full_adder U2 (.ci(co_1), .a(a[1]), .b(b[1]), .co(co_2), .s(sum[1]));
        full_adder U3 (.ci(co_2), .a(a[2]), .b(b[2]), .co(co_3), .s(sum[2]));
        full_adder U4 (.ci(co_3), .a(a[3]), .b(b[3]), .co(sum[4]]), .s(sum[3]));

endmodule
```

2. Описание 4-х разрядного сумматора с использованием непрерывных присваиваний:

```
module adder_4bit (a, b, sum);
        parameter WIDTH=4;          // разрядность = 4
        input [WIDTH-1:0] a, b;     // входные операнды
        output [WIDTH-1:0] sum;     //результат

assign sum=a+b;

endmodule
```

Пример: мультиплексор 4 в 1.

```
// используются непрерывные присваивания и вложенный условный оператор
```

```
module mux4_1 (op0, op1, op2, op3, sel, res);
        input op0, op1, op2, op3;      // входы мультиплексора
        input [1:0] sel;          // управляющие входы
        output res;               // выход мультиплексора
// принцип работы мультиплексора:
// если sel[1:0]=2'b00, то res=op0;
// если sel[1:0]=2'b01, то res=op1;
// если sel[1:0]=2'b10, то res=op2;
// если sel[1:0]=2'b11, то res=op3;

assign res=sel[1] ? (sel[0] ? op3 : op2) : (se[0] ? op1 : op0);

endmodule
```

ПОВЕДЕНЧЕСКИЕ КОНСТРУКЦИИ

Структурные конструкции *initial* и *always.*

Число конструкций ***initial*** и ***always*** в модуле Verilog не ограничено.
Они выполняются параллельно и независимо от порядка записи.
Не допускается вложенность ***initial*** и ***always***.
Каждая структурная конструкция ***initial*** выполняется только один раз, начиная с нулевого момента времени.

Структурная конструкция ***initial*** несинтезируема и обычно используется при моделировании в фалах тестовых проверок функционирования (*test bench* файлах) для инициализации сигналов, формировании воздействий и других процессов, которые должны быть выполнены только один раз.

Пример: фрагмент тестового файла, формирующий входное воздействие для сигнала reset и время моделирования.

```
module design_testbench
…
reg reset;                // объявление переменной
// формирование входного воздействия для сигнала reset:
initial
begin
        reset=1'b1;
        #100 reset =1'b0;
end

// задание момента времени для окончания процесса моделирования
initial
#1000 $finish;
…
endmodule
```

Структурные конструкции ***always*** непрерывно выполняют последовательность определенных внутри них действий. Повторяются циклически, начиная с нулевого момента времени.
Если в модуле есть несколько конструкций ***always***, то каждая из них выполняется параллельно с другими и независимо от других.

Пример: фрагмент тестового файла, формирующий входное воздействие для тактовых импульсов clk.

```
module design_testbench
…
reg clk;                    // объявление переменной
// установка начального состояния сигнала clk :
initial
clk=1'b0;

// формирование переключающих воздействий для сигнала clk:

always
#50 clk=~clk;           // период переключения равен 100 единиц времени
…
endmodule
```

Внутри структурных конструкций ***initial*** и ***always*** могут быть использованы другие поведенческие конструкции и операторы языка Verilog. При этом эти конструкции могут быть сгруппированы с использованием синтезируемого последовательного блока

```
begin
…
end
```

в этом случае операторы выполняются последовательно в порядке записи внутри блока или несинтезируемого, (используемого в test bench) параллельного блока

```
fork
…
join
```

Для привязки моделирования к моменту временили к каким-либо событиям используется временной (timing control) или событийный (event control) контроль.

Временной контроль процедурных операций

#delay <процедурное присваивание/оператор>;
где *delay* – число, обозначающее задержку присваивания в единицах, указанных в директиве ***`timescale***…

Событийный контроль процедурных операций

Событием в данном контексте называется какое-либо изменение значения или состояния цепи или переменной. Для обозначения событийного контроля используется символ @:

@ (событие) <процедурное присваивание/оператор >;

Обозначения переходов сигналов из одного состояния в другое:

- переходы ***posedge****: 0->1, 0->x, 0->z, x->1, z->1*
- переходы ***negedge****: 1->0, 1->x, 1->z, x->0, z->0*

Если требуется осуществлять контроль нескольких событий, то исп-ся оператор ***or***. Ключевое слово ***or*** не приводит к появлению каких-либо дополнительных вентилей в схеме и только перечисляет события или сигналы, при изменении которых происходят

процедурные присваивания. Сигналы, указанные при перечислении событий, называют ***списком чувствительности***. Допускается при перечислении использовать не оператор ***or***, а запятую, или, если нужно при описании комбинационной логики отследить изменения всех входных сигналов - *.

Пример: процедурные присваивания с событийным контролем.

```
module design_RTL (...)
...
reg q, sum                  // объявление переменных
// процедурное присваивание происходит по переднему (положительному) фронту сигнала clk:

always @ (posedge clk)
q=d;

// процедурное присваивание происходит при изменении значений сигналов a или b):

always @ (a or b) // эквивалентно always @ (a, b) или always @*
sum=a+b;
...
endmodule
```

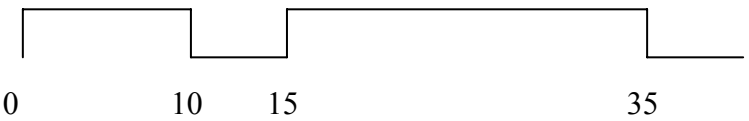
Другой тип событийного контроля – несинтезируемый оператор ***wait*** задерживает выполнение присваивания до тех пор, пока условие не будет истинным.

wait *(условие) <процедурное присваивание/оператор >;*

Процедурные присваивания (procedural assignments)

Если к цепям следует прикладывать непрерывные воздействия (continual assignments), (см. выше), то к переменным (типа ***reg, integer, time, real, realtime***) следует применять процедурные присваивания. Присваивание значений переменным происходит под действием временного или событийного контроля внутри конструкций ***initial*** или ***always*** , а также в процедурах или функциях. Результатом процедурного присваивания может быть скаляр, вектор, отдельные биты или часть векторных переменных, элемент массива, конкатенация переменных или их частей.
Существует два вида процедурных присваиваний: блокирующие (blocking, обозначаются знаком равенства =) и неблокирующие (non-blocking, обозначаются комбинацией знаков равенства и меньше <=).
Блокирующие присваивания выполняются в том порядке, в каком они определены в последовательном блоке ***begin-end.***
Неблокирующие процедурные присваивания выполняются параллельно (одномоментно) в последовательном блоке ***begin-end***.

Пример: организация сигнала с заданными изменениями во времени.



а) блокирующие присваивания:

```
initial
begin
        tst = 1'b0;
        #10 tst = 1'b0;
        #5 tst = 1'b1;
        #20 tst = 1'b0;
end
```

б) неблокирующие присваивания:

```
initial
begin
        tst <= 1'b0;
        #10 tst <= 1'b0;
        #15 tst <= 1'b1;
        #35 tst <= 1'b0;
end
```

Блокирующие процедурные присваивания целесообразно использовать при описании комбинационных схем:

Пример: вычисление выражения *(a&b)|(c&b)*

```
module design RTL_comb (a, b, c, d, result);
        input a, b, c, d;
        output result;
        reg a_b, c_d;           //внутренние переменные
        reg result;             // выходная переменная
// последовательное выполнение процедурных блокирующих присваиваний обеспечивает
// result = (a&b)|(c&b)
always @ (a or b or c or d)
        begin
                a_b = a & b;
                c_d = c & d;
                result = a_b|c_d;
        end
endmodule
```

В данном примере выход *result* объявлен как переменная типа ***reg***, т.к. этот сигнал определяется при помощи процедурного присваивания внутри структурной конструкции ***always.***

Неблокирующие процедурные присваивания целесообразно использовать при описании последовательностных (триггерных) схем:

Пример: 2-х битный сдвиговый регистр с входом d и выходом q (выход первого триггера подан на вход данных второго триггера) с тактирующим сигналом clk.

```
module design RTL_seq (d, clq, q);
        input d, clk;
        output q;
        reg q1;                 //внутренняя переменная
        reg q;                  // выходная переменная
// неблокирующие присваивания выполняются параллельно, порядок записи не имеет
// значения
always @ (posedge clk)
        begin
```

```
            q1 <= d;
            q2< = q1;
            q <= q2;
        end
endmodule
```

Условные поведенческие конструкции *if-else*

Конструкция ***if-else*** применяется при описании триггеров, срабатывающих по фронту, защелок, комбинационных логических схем.
Синтаксис:
if (<условие>)
<процедурная операция, если условие истинно>
else
< процедурная операция, если условие ложно >

Проверка нескольких условий
if (<условие1>)
<процедурная операция, если условие 1 истинно>;
else if (<условие2>)
<процедурная операция, если условие 2 истинно>;
else if (<условие3>)
<процедурная операция, если условие 3 истинно>;
else
< процедурная операция, по умолчанию>;

Укороченная запись: в одной строке, когда отсутствует ***else*** – операция
if (<условие>) <процедурная операция, если условие истинно>;

Пример: мультиплексор 2 в 1.

```
module mux2_1 RTL (op0, op1, select, out_mux);
        input op0, op1;         //входы данных
        input select;           //пуравляющий вход
        output out_mux;         //выход
        reg out_mux;            // переменная
// конструкция принятия решений
always @ (op0 or op1 or select)
        if (select ==1'b0)
                out_mux = op1;
        else
                out_mux = op0;
endmodule
```

Если по условию должна выполняться группа операторов, то они группируются в блоки ***begin-end***. Допускается вложение ***if-else*** конструкций, при этом считается, что оператор ***else*** относится к ближайшему ***if***-оператору.
Пример:

```
module up_down_counter (...);
...
// вложенная инструкция if-else
always @ (op0 or op1 or select)
```

```
        if (enable_count)
                if (up_down)
                        q <= q + 1'b1;
                else    //ветвь else  относится к условию if (up_down)
                        q <= q - 1'b1;
...
endmodule
```

Поведенческие конструкции ветвления *case, casez, casex*.

При множестве условий удобнее использовать ключевые слова ***case, endcase*** и ***default***.
В конструкциях ***case*** выражение сравнивается с приведенными значениями побитно. Они используются при разработке синтезируемого кода.
Синтаксис:
case (<выражение>)
значение 1: < процедурная операция 1 >;
значение 2: < процедурная операция 2 >;
значение 3: < процедурная операция 3 >;
…
default: < процедурная операция, если выражение не равно ни одному из значений >;
endcase

Если требуется выполнение нескольких процедурных операций при определенных значениях выражений, то их следует сгруппировать в последовательном блоке ***begin-end.***

Пример: описание простейшего арифметико-логического устройства (АЛУ) с использованием конструкции ветвления.

```
module alu (op1, op2, op_type, result);
        parameter ADD = 2'b00;
        parameter SUB = 2'b01;
        parameter MUL = 2'b10;
        input [7:0] op1, op2;           //входные операнды
        output reg [15:0] result;       // переменная, выход  АЛУ
// конструкция принятия решений
always @ (op1 or op2 or op_type)
        case (op_type)
        ADD:            result = op1 + op2;
        SUB:            result = op1 - op2;
        MUL:            result = op1 * op2;
        default:         result = 16'b0;
        endcase
endmodule
```

Приведенный выше пример "*mux4"* описания мультиплексора 4 в 1 с использованием непрерывных присваиваний ***assign*** имеет более понятный код, если использовать конструкции ветвления:

```
module mux4_1 (op0, op1, op2, op3, sel, res);
        input op0, op1, op2, op3;       // входы мультиплексора
        input [1:0] sel;                // управляющие входы
        output res;                     // выход мультиплексора
        reg res;                        // объявление типа переменная
```

```
// принцип работы мультиплексора:
// если sel[1:0]=2'b00, то res=op0;
// если sel[1:0]=2'b01, то res=op1;
// если sel[1:0]=2'b10, то res=op2;
// если sel[1:0]=2'b11, то res=op3;

//assign res=sel[1] ? (sel[0] ? op3 : op2) : (se[0] ? op1 : op0);
always @ (op0 or op1 or op2, or op3 or sel)
        case (sel)
        2'b00 :         res = op0;
        2'b01 :         res = op1;
        2'b10 :         res = op2;
        defualt :       res = op31;
        endcase
endmodule
```

Разновидности casex (casez) используются, чтобы не анализировать состояния ***x*** и ***z*** (только ***z***) битов, рассматривать их как безразличные (don't care) и, соответственно, не анализировать их. Для обозначений состояний ***x*** и ***z*** может использоваться символ ***?*** Значение ***?*** также не анализируется.

Пример:

```
module decoder (...)
        ...
always @ (op0 or op1 or op2, or op3 or addrl)
        case (addr)
        4'b1??? :       res = op3;
        4'b01?? :       res = op2;
        4'b001? :       res = op1;
        4'b0001 :       res = op0;
        endcase
...
endmodule
```

Если старший значащий бит сигнала *addr* равен логической 1,то на выход поступает сигнал *op3*, а остальные биты не анализируются. Иначе, если два старших бита равны 2'b01, то на выход поступает сигнал *op2* и т.д.

Поведенческие конструкции циклов *repeat, for, while, forever*

Все конструкции циклов могут быть использованы только внутри структурных блоков ***initial*** и ***always***.

Несинтезируемая конструкция ***repeat:***

```
repeat (число повторений)
<процедурная операция>;
```

Повторяет выполнение процедурных присваиваний или операторов определенное число раз. В конструкции ***repeat*** может использоваться событийный или временной контроль, для группировки нескольких процедурных присваиваний или операторов исп-ся последовательный блок ***begin-end***.

Пример: формирование паузы, длительностью 8 периодов тактовой частоты.

```
initial
begin
...
// формирование ожидания длительностью 8 периодов clk
repeat (8)
@ (posedge clk);
```

...
End
Цикл ***forever***:
forever *<процедурная операция>;*
Не содержит каких-либо условий и циклически повторяет процедурные присваивания или операторы до окончания симуляции. Группировка нескольких присваиваний осуществляется при помощи блока ***begin-end***.
Цикл ***forever*** как правило используется совместно с конструкциями временного или событийного контроля. Если такой контроль отсутствует то цикл будет повторяться постоянно с нулевой задержкой и оставшаяся часть Verilog-описания не будет выполнена (т.е. произойдет зацикливание).
Пример:
initial
begin
clk = 0; // начальная инициализация переменной
// формирование последовательности тактовых импульсов clk с периодом 100
forever
    # 50 clk = ~clk;
end

Цикл ***while***:
while *(условие)*
    <процедурная операция>;
В конструкции ***while*** повторение выполняется до тех пор, пока указанное условие не станет ложным. Если условие не определено (т.е. ***x*** или ***z***), то это трактуется как 0 ине происходит ни одного повторения. Группировка нескольких присваиваний осуществляется при помощи блока ***begin-end***.
Пример: использование цикла ***while*** в test-bench для формирования паузы длительности 8 тактов.
initial
begin
...
count = 8; // начальная инициализация переменной

// переменная count декрементируется по каждому фронту clk до тех пор, пока // не станет равной 0
while *(count > 0)*
    *@ (**posedge** clk) count = count - 1;*
...
end

Цикл ***for***:
for *(начальное присваивание; условие; шаг присваивания)*
    <процедурная операция>;
Конструкция ***for*** повторяет выполнение процедурных присваиваний или операторов такое число раз, которое задается при помощи трех характеристик:
- начальное присваивание – определяет начальное значение повторения;
- условие – определяет состояние, когда повторения должны быть завершены;
- шаг присваивания – определяет величину изменения индекса повторения обычно при помощи декрементировния или инкрементирования.
Группировка нескольких присваиваний осуществляется при помощи блока ***begin-end***.
Конструкция ***for*** – синтезируема.

Пример: описание многоразрядного сумматора, в котором цикл ***for*** используется для определения логической функции i-того бита результирующей суммы.

```
module adder (a, b, co, sum);
        parameter [WIDTH=8];        // разрядность
        input [WIDTH-1:0] a, b;     // входные операнды
        output co;                  // выходной перенос
        output [WIDTH-1:0] sum;     // результирующая сумма
// объявление переменных
reg co;
reg [WIDTH-1:0] sum;
integer i;                          // индекс для использование в конструкции for
// описание функциональной части сумматора:
aways @ (a or b or co)
begin
        co = 1'b0;

        for ( i = 0; i < WIDTH ; i = i +1 )
                begin
                sum [i] = a [i] ^ b [i] ^ co;
                co = ( a [i] & b [i]) |( a [i] & ^ co) |( b [i] & ^ co);
        end
end
endmodule
```

Функции *(function)* и процедуры *(task)*

Используются, когда требуется повторить один и тот же функциональный фрагмент в различных частях кода.

Синтаксис:

```
function ([MSB:LSB]) <имя функции>;
        <декларация входов, переменных>;
        begin
                <процедурные операции>;
        end
endfunction
```

Функция должна удовлетворять следующим требованиям:

- не допускается использовать внутри функции временного или событийного контроля , т.е. конструкций **#, @** или ***wait***;
- внутри функций не допускается вызов процедур, т.е. конструкций ***task;***
- функция возвращает только одну переменную, которая имеет тоже название, что и функция;
- функция должна иметь хотя бы один входной аргумент;
- функция не должна содержать аргументов, продекларированных как ***input*** как ***output***
- функция не должна содержать неблокирующие (non-bloking, <=) присваивания;
- функция может иметь локальные декларации переменных (т.е. переменных типа reg, integer, real и т.д.), но не может иметь деклараций цепей;
- функция не должна содержать ***always*** и ***initial*** блоков;

При объявлении функции происходит неявное объявление переменной с тем же именем и той же разрядностью, что и для функции. Через эту регистровую переменную возвращается значение функции. Вызов функции осуществляется присваиванием какой-либо этой функции с аргументами (параметрами функции), определенными в скобках.

Пример: использование функции в описании умножителя.

```
module multiplier (in1, in2, mult);
        input [7:0] in1, in2 ;          // входные операнды
        output [15:0] mult ;            // результирующее произведение
        reg [15:0] mult ;               // переменная

function [15:0] mult_func;
        input [7:0] a, b;
        mult_func = a * b ;
endfunction

always @ (in1 or in2)
        mult = mult_func (in1, in2);

endmodule
```

Процедура ***task***

Синтаксис:

```
task  <имя процедуры>;
        <декларация входов, выходов, двунапрвленных портов>;
        <декларация переменных>;
        begin
                <процедурные операции>;
        end
endtask
```

Процедуры должны отвечать следующим требованиям:
- допускается использовать внутри процедуры конструкций временного или событийного контроля **#, @** или ***wait***;
- процедуры разрешают вызов как других процедур, так функций***;***
- процедура может иметь как ни одного так и более одного аргументов, продекларированных как ***input*** как ***output***
- процедура не возвращает переменные, а передает их через объявленные выходы;
- процедура может иметь локальные декларации переменных (т.е. переменных типа reg, integer, real и т.д.), но не может иметь деклараций цепей;
- процедура не должна содержать ***always*** и ***initial*** блоков;

Вызов процедуры осуществляется указанием имени процедуры с перечислением входных, выходных и двунапрвленных портов, указанных в скобках.
Пример: использование процедуры.

```
module operation_sum_sub (in1, in2, result_sum, result_sub);
        input [7:0] in1, in2 ;                  // входные операнды
        output [8:0] result_sum ;               // результат сложения
        output [8:0] result_sub ;               // результат вычитания
        reg [8:0] result_sum, result_sub ;              // переменные

task sub_sum_task;                              // объявление процедуры
        input [7:0] a, b;                       // входные аргументы процедуры
        output [8:0] sum, sub ;                 // выходные аргументы процедуры
        begin
                sum = a + b ;
                sub = a – b ;
```

```
        end
endtask

always @ (in1 or in2)
        sum_sub_task (in1, in2, result_sum, result_sub);     // вызов процедуры
endmodule
```

В этом примере аргументами, передаваемыми в процедуру являются *in1* и *in2*. Когда происходит вызов процедуры, то на входы *a* и *b* передаются соответственно *in1* и *in2*. Когда процедура выполнена, из нее передаются выходные аргументы *sum* и sub, которые присваиваются, соответственно, переменным *result_sum* и *result_sub*.
Особенностью процедур является то, что они могут непосредственно использовать переменные, объявленные в модуле, а не в процедуре.
Пример: использование процедуры для работы с переменными, объявленными вне.

```
module design_testbench;
        reg clk, reset_b ;                  // переменные
// объявление процедуры установки начальных значений переменных
task init_setting;                                  // объявление процедуры
        begin
                clk = 1'b0 ;
                reset_b = 1'b1 ;
        end
endtask
initial
        begin
                init_setting;    // вызов процедуры установки начальных значений переменных
                ...
        end
endmodule
```

Замечание 1. Как правило, всеми средствами синтеза поддерживаются следующие конструкции:

наименование	ключевые слова
- определение модуля	***module - endmodule***
- внешние порты	***input, output, inout***
- параметры	***parameter***
- цепи и переменные	***wire, tri, reg***
- подключения модуля	*module instantiation*
- подключение примитива	*gate instantiation*
- функции и процедуры	***function, task***
- непрерывные присваивания	***assign***
- структурные конструкции	***always***
- последовательные блоки	***begin-end***
- условные конструкции	***if-else***
- конструкции ветвления	***case, casex, casez***
- конструкции цикла	***for***

Замечание 2. Средствами синтеза не поддерживаются следующие операторы:
- операторы условного равенства с учетом ***x*** или ***z***: *===* и *!== (в реальной схеме нет возможности сравнивать **x** или **z**)*

- оператор возведения в степень ** (поддерживается ограниченно: только когда один из операндов равен 2)