<параллельные операторы> : == // continious statements

< параллельное присвоение>

|! <оператор вхождения модуля>

|! <оператор блока>

|! <оператор инициализации>

|! <оператор постоянного повторения>

|! <вызов подпрограмм>

Операторы *initial* и *always*

в языке Verilog различают два вида операторов — параллельные (по терминологии разработчиков языка "continuous" — непрерывные) и последовательные (названные разработчиками языка "procedural" — процедурные). Последовательные операторы выполняются друг за другом в порядке записи, параллельные же при любом изменении сигнала, используемого в качестве аргумента. Последовательные операторы должны быть локализованы в теле составных операторов.

Оператор инициализации initial запускается к исполнению единственный раз при начале моделирования и используется либо для задания начальных состояний в моделируемом устройстве, либо при записи программ моделирования (Test-Bench) для описания сигналов, поведение которых предопределено сценарием отладки на всей длине теста.

Синтаксис оператора инициализации определен следующим образом:

<оператор инициализации> ::= **initial** <оператор>

Вложенный оператор может быть составным. Не вдаваясь пока в детали определения составных операторов и блоков, отметим, что составной оператор представляет последовательность операторов, выполняемых друг за другом в порядке записи, ограниченную ключевыми словами **begin** и **end**. После исполнения всех действий, предписанных вложенными операторами, оператор "зависает" и не влияет на исполнение остальных программных конструкций.

Следующая программа представляет тест для проверки некоторого устройства, в данном примере представленного оператором параллельного присваивания.

```
module top;
reg [2:0] a;
wire [4:0] b;
assign #5 b ={c,&a,a};
// параллельный оператор, изменение a и c вызывает
// его исполнение через 5 единиц модельного времени

initial c=1'b1;
// нет необходимости в конструкции begin-end, потому
        // что вложен единственный оператор
initial
begin
a = 'b000;       // начальное значение
#100 a = 'b110;   // каждый оператор выполняется через 100 ед. модельного
              // времени после исполнения предыдущего
#100 a = 'b111;
#100 a = 'b110;
end
initial
begin // запуск системной функции отображения результатов
$monitor($time,, "a=%b, b=%b",a, b);
#1000 $finish;
end
endmodule
```

Операторы инициализации начинают исполняться "одновременно" (в смысле определенном дискретной событийной моделью, то есть их результаты недоступны для операторов, также имеющих нулевую отметку времени).

В результате моделирования будет получено:

```
0 a=000, b=xxxx
5 a=0000, b=10000
100 a=110, b=10000
105 a=110, b=10110
200 a=111, b=10110
205 a=111, b=11111
300 a=110, b=11111
305 a=110, b=10110
1000    $finish
```

Оператор постоянного повторения **always** также первый раз исполняется при начале моделирования, но затем повторяется каждый раз после завершения вложенного оператора. Синтаксис оператора имеет вид:

<оператор постоянного повторения> ::= **always** <оператор>

Если вложенный оператор составной, то действия повторяются после исполнения последнего оператора из числа вложенных в него. В связи с циклическим характером оператора **always**, он может иметь смысл только при наличии в его теле конструкций, предусматривающих его переход в состояние ожидания. Это может быть ожидание заданных моментов времени или иных событий в системе. Если не предусмотреть переход оператора в состояние ожидания, все другие операторы в программе будут заблокированы.

Например, конструкция

```
always a= ~a;
```

порождает бесконечный цикл с нулевой задержкой, в то время как

```
always #delay a= ~a;
```

представляет генератор импульсов единичной скважности и периодом `2*delay`, причем в интервале ожидания могут исполняться другие операторы, например, вызываемые изменением переменной `a`.

Инициализация процедурных операторов

В языке Verilog порядок исполнения операторов определяется не только и не столько порядком их записи. Предусмотрен широкий набор средств, определяющих условия, при которых оператор будет исполнен. Эти условия оформляются в виде префиксов операторов и выражений языка, в том числе логических и арифметических выражений, а также и выражений присваивания. Среди этих средств наиболее важное значение имеют *префиксы управления временем* и *префиксы событийного управления*

Ранее был рассмотрен ряд примеров использования префикса времени перед операцией присваивания. Префикс начинается с символа #, после которого записывается число, имя переменной времени или выражение. Однако действие этого префикса в различных ситуациях неодинаково.

Префикс времени перед оператором непрерывного присваивания означает вычисление нового значения на основе текущих значений аргументов и сохранение этого значения в буфере на время, заданное параметром префикса. Префикс перед словом **begin**, открывающим последовательный блок, означает задержку начала исполнения всего блока.

Если префикс предшествует последовательному оператору, то оператор исполняется после предыдущего в последовательности через временной интервал, заданный префиксом. Фактически, это означает приостанов исполнения программного блока. В качестве исходных данных для оператора используется значение переменных на момент начала его исполнения.

Заметим, что изменения аргументов могут быть произведены другими операторами, исполняемыми в "параллельно" инициированных блоках. Такие изменения учитываются параллельными операторами только после исполнения всех операторов имеющих равную метку времени или инициализированных общим событием.

И наконец, префикс времени, расположенный перед записью операции в правой части оператора присваивания (такую запись называют "intra-assignment control", что можно перевести как управление изнутри присвоения), означает задержку присвоения результата приемнику, хотя при вычислении выражения также используются значения аргументов, существовавшие на момент инициализации оператора. (моделирование идеальной задержки

Перечисленные правила иллюстрируются программой, представленной в листинге и на рисунке, показаны результаты ее моделирования.

```
module  delay_example;
wire d;
reg a,b,c;
assign  #10 d=a;
initial
 begin a=0;
```

Процедурные (последовательные) операторы
Присваивания (блокирующин и неблокирующие)

```
always #10
      begin
           a_blocked=input;//(:= !!)
           b_blocked=a_blocked;
           a_non_blocked<= input #delta;
           b_non_blocked<= a_non_blocked;
      end
```

```
always  @ (posedge clock )
begin

  q1[0]=d;
  q1[1]=q1[0]; // повторители
  q2[0]<=d;
  q2[1]<=q2[0]; // регистр сдвига
end
```

3.3.6. Операторы принятия решений

Опенратор условия

```
    if ( <выражение>) <оператор>
       { [else if  <оператор>];
                 else
              }
```

<оператор варианта> ::=

<определитель оператора варианта>(<ключевое выражение>)

« <Вариант> «,<Вариант> » : <оператор>»

endcase

<определитель оператора варианта> ::= case| casez | casex

<вариант> := <константное выражение> | **default**

Исполняется лишь тот оператор, значение варианта которого совпадает со значением ключевого выражения. Если ни один из вариантов не совпадает с вычисленным значением ключевого выражения, то выполняется ветвь **default** или не выполняется ни один оператор. *В отличие от С*

выполняется только один вариант. Например, следующий оператор описывает дешифратор с трехразрядным адресным входом, входом разрешения и восемью выходами при представлении выбранного выхода низким уровнем сигнала.

```
case (en,in_code)
4'd8: {result = 8'b011111111;….}
4'd9:  result = 8'b101111111;
4'd10: result = 8'b11011111;
4'd11: result = 8'b11101111;
4'd12: result = 8'b11110111;
4'd13: result = 8'b11111011;
4'd14: result = 8'b11111101;
4'd15: result = 8'b11111110;
4'd0, 4'd1, 4'd2, 4'd3, 4'd4, 4'd5, 4'd6, 4'd7:
              result = 8'b11111111;
default result = 8'bx;
endcase
```

```
casez (ir)
5'b1????: request_code=5;
5'b01???: request_code=4;
5'b001??: request_code=3;
5'b0001?: request_code=2;
5'b00001: request_code=1;
5'b00000: request_code=0;
endcase
```

Операторы повторения

В языке Verilog определены четыре формы операторов повторения

<оператор повторения> ::=

forever <оператор> // for(, ,)

| **repeat** (<выражение>) <оператор>

| **while** (<выражение>)<оператор>

| **for** (<присвоение>; <выражение>; <присвоение>) <оператор>

.

Следующий листинг представляет описание последовательного умножителя., с последдовательным сдвигом кодов в регистрах множимого и множителя.

Листинг

```
module  multiply (clock,start, a,b,result,ready);
input clock,start;
input a,b;
output result,ready;
parameter size = 8, longsize = 16;
wire [size:1] a,b;
reg [size:1] opb;
reg ready;
reg [longsize:1] result;
reg [longsizые:1] opa;
 always @ ( posedge start)
    begin
    opa = a; // загрузка новых исходных данных
    opb = b;
    result = 0;
    ready= 0;
    repeat (size)   // повторить для всех разрядов
      @ ( posedge clock) // блок инициируется фронтом
//clock
       begin
          if (opb[1])  result = result + opa;
          opa = opa <<1;
           opb = opb>> 1;
       end
     ready=1;
    end
endmodule
```

Фрагмент, представленный далее, описывает процесс подсчета числа единиц в байте входных данных `in_data`. На каждом шаге алгоритма выполняется сдвиг кода аргумента, и, если в младшем разряде нуль, к результату прибавляется единица. Цикл прекращается, когда сдвигающий регистр очищен.

. . .

```
begin
  reg [7:0] shift_reg;
  result = 0;
  shift_reg = rega;
  while(shift_reg)
    begin
      if (shift_reg [0]) result = result + 1;
      shift_reg  = shift_reg>> 1;
    end
end
```

Этот же алгоритм может быть записан с использованием оператора `for`:

```
begin :count1s
  reg [7:0] shift_reg;
  result = 0;
  for (shift_reg = rega; shift_reg!=0; shift_reg =
shift_reg>> 1)
      if (tempreg[0]) result = result + 1;
end
```

В операторе **`for`** первое присвоение задает начальное значение переменной цикла, следующее за этим выражение — условие окончания (цикл прекращается, когда выражение дает нуль или не определено), а последнее (необязательное) — любые дополнительные присвоения, чаще всего преобразование переменной цикла.

Выражение вхождения

```
   Multiply (clock,start,a.b,
        result,ready),
 parameter  n=8,
              m= 16;
input a,b;
wire [n-1:0] a,b;

output result,ready;
reg [m-1:0] result;
reg ready;
….

module  mult8x8 (clk,strt, mult,factor,product);
input clk,strt;
input mult,factor;
output product;
wire [7:0] mult,factor;
reg [7:0] v1, v2, v3,v4,elder_part;
reg [8:0] middle_part;
wire [15:0]  product;
wire fin;
multiply #(4,8)
                    lower (.clock(clk),.start(strt),.a(mult[3:0]),.b(factor[3:0]),
        .result(v1),.ready(fin)),
  middle1(.clock(clk),.start(strt),.a(mult[7:4]),
        .b(factor[3:0]),.result(v2)),
  middle2(.clock(clk),.start(strt),
          .a(mult[3:0]),.b(factor[7:4]),.result(v3)),
  elder  (clk,strt,mult[7:4],factor[7:4],v4,);
always @(posedge fin)
 begin
   middle_part={1'b0,v2}+ {1'b0,v3}+ {5'd0,v1[7:4]};
   elder_part= {3'd0,middle_part[8:4]}+v4;
 end
assign
product={elder_part, middle_part[3:0],v1[3:0]};
end module;
```