Уважаемый читатели! Автор, по просьбе читателей, непрерывно вносит дополнения в описание разделов. Эти дополнения более полно описывают варианты применения языка Verilog. В этом дополнении приведены описания двух групп утверждений, первые - позволяют выполнять циклы, а вторые - позволяют генерировать компоненты, устанавливаемые в проект.

2.31. Утверждения выполнения цикла

Есть четыре типа утверждений, которые позволяют выполнять циклы. Эти утверждения позволяют контролировать число повторений циклов. Можно управлять выполнением циклов, то есть делать их ноль раз, один раз, или большее количество раз.
forever *(навсегда)* – заданный цикл утверждений выполняется непрерывно.
repeat *(повторение)* - утверждение выполняется фиксированное число раз. Если результат выражения, описывающее число повторений цикла, будет оценен как неизвестное значение или как высокий импеданс, то число повторений для такого цикла будет определено как нуль, и никакие утверждения из этого цикла не должны выполняться.
while *(до тех пор, пока)* - утверждения выполняются до тех пор, пока выражение не становится ложным. Если выражение с самого начала трактуется как ложное, то цикл не должен выполняться вообще.
for *(для)* - такой цикл выполняется в соответствии с заданными для цикла утверждениями следующим образом:
a) выполняется назначение, которое обычно производит инициализацию переменной, управляющей числом выполняемых циклов.
b) производится оценка выражения, которое ограничивает число циклов, и если в результате этой оценки будет определено, что необходимо выполнить еще несколько циклов, то произойдет выполнение следующего цикла, и затем будут произведены действия, описанные в пункте «c». А в том случае, когда число циклов, которые необходимо выполнить будет оценено как нуль, то цикл должен завершиться. Если результат выражения будет оценен как неизвестное значение или как высокий импеданс, то это должно быть обработано как нуль циклов.
c) Выполняется то назначение, которое указано в записи цикла для переменной управления циклом. Оно изменяет значение переменной управления циклом, затем повторяется шаг «b».
В примере 2.111 показан синтаксисис для различных утверждений выполнения цикла.

```
function_loop_statement ::= forever function_statement
    | repeat ( expression ) function_statement
    | while ( expression ) function_statement
    | for ( variable_assignment ; expression ; variable_assignment )
function_statement
loop_statement ::= forever statement
    | repeat ( expression ) statement
    | while ( expression ) statement
    | for ( variable_assignment ; expression ; variable_assignment )
      statement
```

Пример 2.111. Синтаксисис для утверждений выполнением цикла.

Далее приведены примеры применения трех выриантов утверждений для выполнения цикла. Цикл **forever** должен только использоваться вместе с управлением по времени, с задержками и т.д. или отключающим утверждением - **disable**, и, поэтому, этот пример был представлен выше.
Применение утверждения **repeat,** показано в примере 2.112. В этом случае, с помощью цикла с использованием утверждения **repeat**, операторов суммирования и сдвига выполняется уможитель.

```
parameter size = 8, longsize = 16;

reg [size:1] opa, opb;
reg [longsize:1] result;

begin : mult
  reg [longsize:1] shift_opa, shift_opb;
  shift_opa = opa;
  shift_opb = opb;
  result = 0;

  repeat (size) begin
    if (shift_opb[1])
      result = result + shift_opa;
      shift_opa = shift_opa << 1;
      shift_opb = shift_opb >> 1;
    end
end
```

Пример 2.112. Применение утверждения **repeat** для организации цикла.

Применение утверждения **while,** показано в примере 2.113. В этом случае, показан подсчет числа логических 1 в rega.

```
begin : count1s
  reg [7:0] tempreg;
  count = 0;
  tempreg = rega;

  while (tempreg) begin
    if (tempreg[0])
      count = count + 1;
      tempreg = tempreg >> 1;
    end
end
```

Пример 2.113. Применение утверждения **while** для организации цикла.

Применение утверждения **for**, производится аналогично применению утверждения **while**, показанному в предыдущем примере – см. пример 2.113 и 2.114. Но тот же самый результат можно получить, если применить следующую запись, см. пример 2.115.

```
begin
 initial_assignment;

  while (condition) begin
    statement
    step_assignment;
  end
end
```

Пример 2.114. Применение утверждения **while** для организации цикла..

```
for (initial_assignment; condition; step_assignment)
  statement
```

Пример 2.115. Применение утверждения **for** для организации цикла..

2.32. Утверждения generate

2.32.1. Утверждения generate

В процессе разработки иногда бывает необходимо включить в состав проекта несколько однотипных компонентов или компонентов, имеющих какой либо параметр. Чтобы не описывать каждый такой компонент отдельно, можно прибегнуть к генерированию таких компонентов на этапе разработки проекта. Для этого используются ключевые слова **generate - endgenerate**. Все утверждения, которые необходимо сгенерировать, должны находиться между этими словами. Если для генерации чего либо используются переменные, то они называются **genvar** и они определены только на этапе генерации. Эти переменные не могут использоваться при симуляции проекта. Для всех утверждений – декларации переменных, функций, заданий (task), которые были выполнены при генерации действуют те же правила, что и для остальных утверждений. Для установки утверждений можно использовать условия. Все сгенерированные компоненты будут иметь свой уникальный индекс. И на них можно будет ссылаться в проекте.
Для того, чтобы выполнить взаимоподключения между элементами, внутри цикла генерации можно использовать следующие типы данных: **net**, **reg**, **integer**, **real**, **time**, **realtime**, и **event**. Эти компоненты, будучи сгенерированными, так же будут иметь свой индекс. И на них так же можно будет ссылаться в проекте.
Использование переопределения параметра, когда новое значение параметра задается в соответствии с порядком определения параметров в проекте или в соответствии с названием параметра «параметр = назначение значения» или как назначение при помощи утверждения

defparam, может также быть объявлено в пределах видимости цикла генерации. Однако, утверждение defparam будет действовать только пределах видимости цикла генерации или в пределах той иерархии, которая генерируется в данном цикле.
Задания (task) и объявления функций должны также работать только в пределах видимости генерации, но не в генерирующемся цикле. Сгенерированные задания и функции должны иметь уникальные названия идентификатора и на них можно в проекте.
При объявлении модуля или частей модуля, утверждении **generate** не разрешается использовать: параметры, локальные параметры, объявления входов и выходов, двунаправленных входов-выходов, и определять блоки.
Подключения к сгенерированным случаям модуля выполняются по тем же самым правилам, что и с остальными модулями.
Утверждения сгенерированные в цикле, создаются, при использовании одного из следующих методов: **generate-loop**, **generate-conditional**, или **generate-case**.
Синтаксис для реализации генерирации, приведен в примере 2.116

```
module_item ::= module_or_generate_item
   | port_declaration ;
                | { attribute_instance } generated_instantiation
                | { attribute_instance } local_parameter_declaration
                | { attribute_instance } parameter_declaration
                | { attribute_instance } specify_block
                | { attribute_instance } specparam_declaration
module_or_generate_item ::=
                { attribute_instance } module_or_generate_item_declaration
                | { attribute_instance } parameter_override
                | { attribute_instance } continuous_assign
                | { attribute_instance } gate_instantiation
                | { attribute_instance } udp_instantiation
                | { attribute_instance } module_instantiation
                | { attribute_instance } initial_construct
                | { attribute_instance } always_construct
module_or_generate_item_declaration ::=
                net_declaration
                | reg_declaration
                | integer_declaration
                | real_declaration
                | time_declaration
                | realtime_declaration
                | event_declaration
                | genvar_declaration
                | task_declaration
                | function_declaration
generated_instantiation ::=
                generate { generate_item } endgenerate
                generate_item_or_null ::=
                generate_item | ;
                generate_item ::=
                generate_conditional_statement
```

```
                | generate_case_statement
                | generate_loop_statement
                | generate_block
                | module_or_generate_item
generate_conditional_statement ::=
                if ( constant_expression ) generate_item_or_null [ else generate_item_or_null ]
generate_case_statement ::= case ( constant_expression )
                genvar_case_item { genvar_case_item } endcase
genvar_case_item ::= constant_expression { , constant_expression } :
                generate_item_or_null | default [ : ] generate_item_or_null
generate_loop_statement ::=
                for ( genvar_assignment ; constant_expression ; genvar_assignment )
            begin : generate_block_identifier { generate_item } end
genvar_assignment ::=
                genvar_identifier = constant_expression
generate_block ::=
                begin [ : generate_block_identifier ] { generate_item } end
```

Пример 2.116. Синтаксисис для реализации генерирации.

2.32.2. genvar – индексная переменная цикла генерации

Индексная переменная, которая должна только быть объявлена для использования в цикле генерации, объявляется как ***genvar*** и упоминается как ***genvar*** в остальной части этого раздела.
Синтаксис для генерации индексных переменных, дается в примере 2.117.

```
genvar_declaration ::= genvar list_of_genvar_identifiers ;
list_of_genvar_identifiers ::= genvar_identifier { , genvar_identifier }
```

Пример 2.117. Синтаксисис для реализации генерации индексной переменной **genvar.**

Переменная ***genvar*** должна быть объявлена в пределах того модуля, где эта переменная ***genvar*** используется. Переменная ***genvar*** может быть объявлена как внутри цикла генерации, где она используется, так и вне этого цикла. Переменная ***genvar*** – это целое число, которое является локальной переменной по отношению к циклу генерации к и она должна использоваться только в пределах цикла генерации, который использует эту переменную как индекс. Если какой-нибудь бит переменной ***genvar*** когда-либо будет установлен в значение X или Z, или если genvar будет задано отрицательное значение, то это должно быть оценено как ошибка.
Переменные ***genvar*** будут только определены в течение цикла генерации блоков, и они не существуют в течение моделирования проекта.
Значение ***genvar*** должно только быть определено генерирующимся циклом. Два цикла генерации, использующие одну и ту же переменную ***genvar***, не могут быть вложены один в другой. На значение ***genvar*** можно сослаться в любом контексте, где это требуется.

2.32.3. Цикл generate

Цикл **generate** может содержать одно или более объявлений переменных, модули, примитивы, определяемые пользователем, примитивы-вентили, непрерывные назначения, блоки инициации и блоки always. Все они могут быть многократно установлены в проекте, используя цикл **generate-loop**.
Индексная переменная цикла, используемая в генерировании цикла **generate-loop** должна быть объявлена как genvar.
Когда мы хотим организовать цикл, например вот такой:

```
genvar i,j;
generate for (i = 0; i < SIZE; i = i + 1)
```

то мы должны использовать два назначения переменной ***genvar***:
первое назначение – «i = 0», второе назначение – «i = i + 1».
Рекомендуется в одном цикле использовать только одну переменную, например так, как показано выше. Не рекомендуется в одном цикле делать назначения для разных переменных, т.е. делать вот так:

```
for (i= 0, j = 0; i < SIZE; i = i+1, j = i + 3)
```

При этом первое присвоение значения переменной всегда должно быть однозначным, т.е. не рекомендуется делать так:

```
for (i = i+1, i < SIZE; i=i+1)
```

или так:

```
for (i = j, i<SIZE; i = i + 1)
```

То есть, первое назначение ***genvar*** в цикле не должно содержать ссылку на индексную переменную, используемую в цикле справа.

```
module gray2bin1 (bin, gray);
  parameter SIZE = 8; // this module is parameterizable
      output [SIZE-1:0] bin;
      input [SIZE-1:0] gray;

      genvar i;

      generate for (i=0; i<SIZE; i=i+1) begin:bit
        assign bin[i] = ^gray[SIZE-1:i];
      end endgenerate
endmodule
```

Пример 2.117. Параметризированный конвертер двоичного кода в код Грея, генерация непрерывных назначений.

В примере 2.118 показан параметризированный конвертер двоичного кода в код Грея, такой же как и в предыдущем примере, но теперь выполняется генерация назначений **always.**

```
module gray2bin2 (bin, gray);
  parameter SIZE = 8; // this module is parameterizable
  output [SIZE-1:0] bin;
  input  [SIZE-1:0] gray;
  reg    [SIZE-1:0] bin;
  genvar i;
    generate for (i=0; i<SIZE; i=i+1) begin:bit
      always @(gray[SIZE-1:i]) // fixed part select
      bin[i] = ^gray[SIZE-1:i];
    end endgenerate
endmodule
```

Пример 2.118. Параметризированный конвертер двоичного кода в код Грея, такой же как и в предыдущем примере, но теперь выполняется генерация назначений **always**.

В примерах 2.119 и 2.120 – показаны параметризованные модули сумматоров, в которых используется цикл **generate**, для того чтобы генерировать примитивы-вентили XOR, AND, OR. В примере 2.119 использовано объявление цепи, имеющей два измерения. Это объявление дается вне цикла генерации, и оно необходимо для того, чтобы сделать подключения между примитивами-вентилями. А в примере 2.120 объявление цепи, необходимой для подключения примитивов делается непосредственно в цикле **generate**.

```
 module addergen1 (co, sum, a, b, ci);
  parameter SIZE = 4;
      output [SIZE-1:0] sum;
      output co;
      input [SIZE-1:0] a, b;
      input ci;
      wire [SIZE :0] c;
      wire [SIZE-1:0] t [1:3];

      genvar i;

      assign c[0] = ci;
      // Generated instance names are:
      // xor gates: bit[0].g1 bit[1].g1 bit[2].g1 bit[3].g1
      // bit[0].g2 bit[1].g2 bit[2].g2 bit[3].g2
      // and gates: bit[0].g3 bit[1].g3 bit[2].g3 bit[3].g3
      // bit[0].g4 bit[1].g4 bit[2].g4 bit[3].g4
      // or gates: bit[0].g5 bit[1].g5 bit[2].g5 bit[3].g5
      // Generated instances are connected with
      // multi-dimensional nets t[1][3:0] t[2][3:0] t[3][3:0]
      // (12 multi-dimensional nets total)

      generate
        for(i=0; i<SIZE; i=i+1) begin:bit
          xor g1 ( t[1][i], a[i], b[i]);
          xor g2 ( sum[i], t[1][i], c[i]);
          and g3 ( t[2][i], a[i], b[i]);
          and g4 ( t[3][i], t[1][i], c[i]);
          or g5 ( c[i+1], t[2][i], t[3][i]);
      end
      endgenerate
      assign co = c[SIZE];
endmodule
```

Пример 2.119. Параметризованный модуль сумматора, в котором используется цикл **generate**, для того чтобы генерировать примитивы-вентили XOR, AND, OR. В этом примере использовано объявление цепи, имеющей два измерения. Это объявление дается вне цикла генерации, и оно необходимо для того, чтобы сделать подключения между примитивами-вентилями.

```
 module addergen1 (co, sum, a, b, ci);
  parameter SIZE = 4;
      output [SIZE-1:0] sum;
      output co;
      input [SIZE-1:0] a, b;
      input ci;
      wire [SIZE :0] c;

      genvar i;

      assign c[0] = ci;

      // Generated instance names are:
      // xor gates: bit[0].g1 bit[1].g1 bit[2].g1 bit[3].g1
      // bit[0].g2 bit[1].g2 bit[2].g2 bit[3].g2
      // and gates: bit[0].g3 bit[1].g3 bit[2].g3 bit[3].g3
      // bit[0].g4 bit[1].g4 bit[2].g4 bit[3].g4
      // or gates: bit[0].g5 bit[1].g5 bit[2].g5 bit[3].g5
      // Generated instances are connected with
      // generated nets: bit[0].t1 bit[1].t1 bit[2].t1 bit[3].t1
      // bit[0].t2 bit[1].t2 bit[2].t2 bit[3].t2
      // bit[0].t3 bit[1].t3 bit[2].t3 bit[3].t3

      generate
      for(i=0; i<SIZE; i=i+1) begin:bit
        wire t1, t2, t3; // generated net declaration
        xor g1 ( t1, a[i], b[i]);
        xor g2 ( sum[i], t1, c[i]);
        and g3 ( t2, a[i], b[i]);
        and g4 ( t3, t1, c[i]);
        or g5 ( c[i+1], t2, t3);
       end
      endgenerate

      assign co = c[SIZE];
endmodule
```

Пример 2.120. Параметризованный модуль сумматора, в котором используется цикл **generate**, для того чтобы генерировать примитивы-вентили XOR, AND, OR. В этом примере объявление цепи, необходимой для подключения примитивов делается непосредственно в цикле **generate**.

Сгенерированные названия компонентов, установленных в результате генерации, в многоуровневом цикле генерирации, показаны в примере 2.121. Сгенерированное название для компонентов в каждом цикле генерации, создаются путем добавления строки к концу генерирующегося идентификатора блока для цикла. Эта строка представляет собой значение переменной , заключенной в квадратные скобки - "[genvars value]". Сгенерированные названия теперь представляют собой идентификаторы, которые могут использоваться в иерархических именах.

```
parameter SIZE = 2;
genvar i, j, k, m;
generate
  for (i=0; i<SIZE+1; i=i+1) begin:B1 // scope B1[i]
        M1 N1(); // instantiates B1[i].N1[i]
            for (j=0; j<SIZE; j=j+1) begin:B2 // scope B1[i].B2[j]
              M2 N2(); // instantiates B1[i].B2[j].N2
                for (k=0; k<SIZE; k=k+1) begin:B3 // scope B1[i].B2[j].B3[k]
                    M3 N3(); // instantiates B1[i].B2[j].B3[k].N3
                  end
              end
            if (i>0)
              for (m=0; m<SIZE; m=m+1) begin:B4 // scope B1[i].B4[m]
                M4 N4(); // instantiates B1[i].B4[m].N4
              end
       end
endgenerate

// some of the generated instance names are:
// B1[0].N1 B1[1].N1
// B1[0].B2[0].N2 B1[0].B2[1].N2
// B1[0].B2[0].B3[0].N3 B1[0].B2[0].B3[1].N3
// B1[0].B2[1].B3[0].N3
// B1[1].B4[0].N4 B1[1].B4[1].N4
```

Пример 2.121. Многоуровневый цикл **generate.**

2.32.4. Условная генерация с if-else-if

Выше мы рассмотрели случаи, когда генерация компонентов происходит безусловно. Но, кроме безусловной генерации компонентов, можно получить так же и условную генерацию компонентов. Для этого могут быть использованы выражения if-else-if. С их помощью могут быть так же сгенерированы конструкции, которые включают в себя модули, примитивы, определяемые пользователем, примитивы-вентили, непрерывные назначения, блоки инициализации и блоки always.

В примере 2.122 показан вариант описания модуля параметризируемого умножителя. Если хотя бы одним из множителей в этом умножителе имеет разрядность по входам a_width или b_width, определяемыми соответствующими параметрами, меньшую чем 8 бит, то в проект устанавливается модуль CLA_multiplier. Если оба множителя a_width и b_width больше или равны 8-ми битам, то в проект устанавливается умножитель WALLACE_multiplier.

```
module multiplier(a,b,product);
parameter a_width = 8, b_width = 8;
localparam product_width = a_width+b_width; // can not be modified
// directly with the defparam statement
// or the module instance statement #
input [a_width-1:0] a;
input [b_width-1:0] b;
output [product_width-1:0] product;

generate
  if((a_width < 8) || (b_width < 8))
```

```
    CLA_multiplier #(a_width,b_width) u1(a, b, product);
   // instance a CLA multiplier
  else
    WALLACE_multiplier #(a_width,b_width) u1(a, b, product);
    // instance a Wallace-tree multiplier
endgenerate
  // The generated instance name is u1
endmodule
```

Пример 2.122. Вариант описания модуля параметризируемого умножителя

2.32.4. Условная генерация с case

При использовании выражения **case** появляется возможность генерировать модули в которых выполняется функция выбора - «один из многих», аналогичная дешифратору. С помощью **case** могут быть так же сгенерированы конструкции, которые включают в себя модули, примитивы, определяемые пользователем, примитивы-вентили, непрерывные назначения, блоки инициализации и блоки **always**. Выражение **case** должно быть определенным во время разработки проекта. В примере 2.123 приведен случай генерации **case** для варианта, когда разрядность дешифратора меньше 3-х. В примере 2.124 показано описание для модуля памяти **dimm**.

```
generate
  case (WIDTH)
        1: adder_1bit x1(co, sum, a, b, ci);
            // 1-bit adder implementation
        2: adder_2bit x1(co, sum, a, b, ci);
            // 2-bit adder implementation
        default: adder_cla #(WIDTH) x1(co, sum, a, b, ci);
            // others - carry look-ahead adder
        endcase
// The generated instance name is x1
endgenerate
```

Пример 2.123. Генерация **case** для варианта, когда разрядность меньше 3-х.

```
module dimm;
parameter [31:0] MEM_SIZE = 8, // in mbytes
      MEM_WIDTH = 16;

      input [11:0] adr;
      input [1:0] ba;
      input rasx, casx, csx, wex;
      input [7:0] dqm;
      input cke;
      input [7:0] ds;
      inout [63:0] data;
      input [3:0] clk;
      wire rasb, casb, csb, web;
      wire [7:0] bex;

      genvar i;

      generate
```

```
      case ({MEM_SIZE, MEM_WIDTH})
      {32'd8, 32'd16}: // 8Meg 16 bits wide.
        begin
          for (i=0;i<4;i = i + 1)
            begin:word
              sms_16b216t0 p
              (.clk(clk), .csb(csx), .cke(cke), .ba(ba[0]),
              .addr(adr[10:0]),...rasb(rasx), .casb(casx),
              .web(wex),.udqm(dqm[2*i+1]), .ldqm(dqm[2*i]),
              ...dqi(data[15+16*i:16*i]), .dev_id(dev_id3[4:0])
              );
            end
            task read_mem;

            input [31:0] address;
            output [63:0] data;
            begin
              word[3].p.read_mem(address, data[63:48]);
              word[2].p.read_mem(address, data[47:32]);
              word[1].p.read_mem(address, data[31:16]);
              word[0].p.read_mem(address, data[15:0]);
            end
          endtask
        end
        // The generated instance names are word[3].p, word[2].p,
        // word[1].p, word[0].p, and the task read_mem
        //{32'd16, 32'd8}: - 16Meg 8 bits wide.
        begin
        for (i=0;i<4;i = i + 1)
          begin:byte
          sms_16b208t0 p
          (.clk(clk), .csb(csx), .cke(cke), .ba(ba[0]),
          .addr(adr[10:0]),
          ...rasb(rasx), .casb(casx), .web(wex), .dqm(dqm[i]),
          .dqi(data[8+8*i:8*i]),...dev_id(dev_id7[4:0])
          );
          end

        task read_mem;
        input [31:0] address;
        output [63:0] data;
          begin
            byte[7].p.read_mem(address, data[63:56]);
            byte[6].p.read_mem(address, data[55:48]);
            byte[5].p.read_mem(address, data[47:40]);
            byte[4].p.read_mem(address, data[39:32]);
            byte[3].p.read_mem(address, data[31:24]);
            byte[2].p.read_mem(address, data[23:16]);
            byte[1].p.read_mem(address, data[15:8]);
            byte[0].p.read_mem(address, data[7:0]);
          end
        endtask
        .....
       endcase
      endgenerate
      // The generated instance names are byte[7].p, byte[6].p,
      // byte[5].p, byte[4].p, byte[3].p, byte[2].p, byte[1].p,
      // byte[0].p and the task read_mem
endmodule
```

Пример 2.124. Пример описания для модуля памяти dimm.