

ОСНОВЫ

VerilogHDL/SystemVerilog

(синтез и моделирование)



# Параметры

---

# Параметры - Parameters

---

- ❑ Символическому имени присваивается некоторое значение
- ❑ Два типа параметров:
  - ✓ parameter
  - ✓ localparam
- ❑ Во время компиляции преобразуется в константу
- ❑ Значение может быть изменено на этапе компиляции
  - ✓ Исключение: Local parameters (**localparam**)

```
parameter size = 8;
```

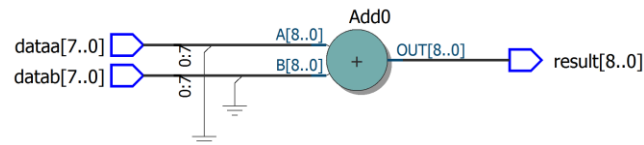
```
reg [size-1:0] dataa, datab;
```

# Использование параметров

```
1 module unsigned_adder
2   #(parameter WIDTH=8)
3   (
4     input [WIDTH-1:0] dataa,
5     input [WIDTH-1:0] datab,
6     output [WIDTH:0] result
7   );
8
9   assign result = dataa + datab;
10
11 endmodule
```

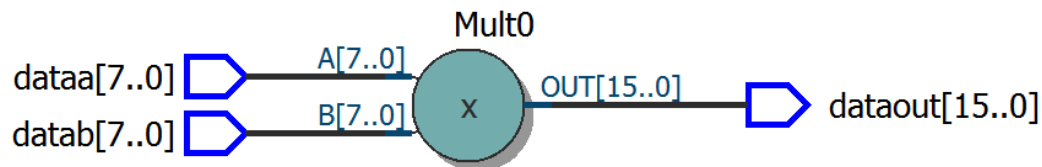
```
1 module unsigned_adder_ (dataa, datab, result);
2   parameter WIDTH=8;
3
4   input [WIDTH-1:0] dataa;
5   input [WIDTH-1:0] datab;
6   output [WIDTH:0] result;
7
8   assign result = dataa + datab;
9
10 endmodule
```

```
1 module unsigned_adder__ (dataa, datab, result);
2   localparam WIDTH=8;
3
4   input [WIDTH-1:0] dataa;
5   input [WIDTH-1:0] datab;
6   output [WIDTH:0] result;
7
8
9   assign result = dataa + datab;
10
11 endmodule
```



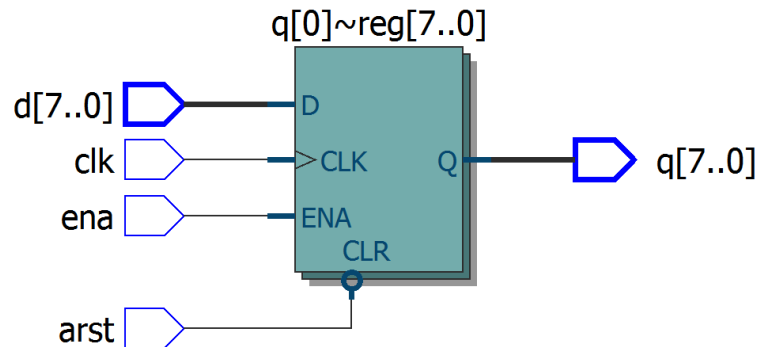
# Умножитель

```
1 module param_mult
2   #(parameter WIDTH=8)
3   (
4     input [WIDTH-1:0] dataa,
5     input [WIDTH-1:0] datab,
6     output [2*WIDTH-1:0] dataout
7   );
8
9   assign dataout = dataa * datab;
10
11 endmodule
```



# Регистр

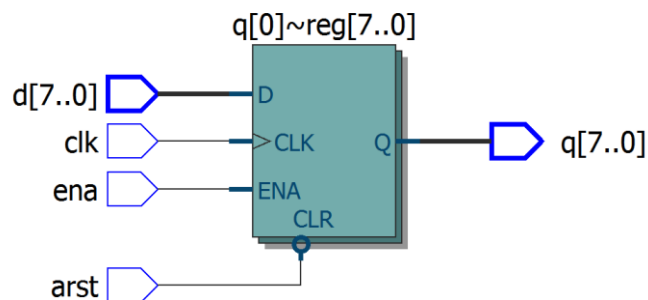
```
1 module rgstr (arst, ena, clk, d, q);
2   parameter width_rg = 8;
3
4   input clk, arst, ena;
5   input [width_rg-1:0] d;
6   output reg [width_rg-1:0] q;
7
8   always @(posedge clk, negedge arst)
9     if (arst==1'b0) q <= {width_rg{1'b0}};
10    else if (ena) q <= d;
11
12 endmodule
```



# Регистр (Правильно ?)

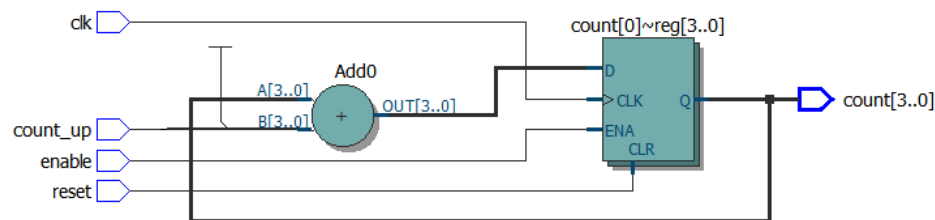
```
1 module rgstr (arst, ena, clk, d, q);
2   parameter width_rg = 8;
3
4   input clk, arst, ena;
5   input [width_rg-1:0] d;
6   output reg [width_rg-1:0] q;
7
8   always @(posedge clk, negedge arst)
9     if (arst==1'b0) q <= {width_rg{1'b0}};
10    else if (ena) q <= d;
11
12 endmodule
```

```
1 module rgstr_ (arst, ena, clk, d, q);
2   parameter width_rg = 8;
3
4   input clk, arst, ena;
5   input [width_rg-1:0] d;
6   output reg [width_rg-1:0] q;
7
8   always @(posedge clk, negedge arst)
9     if (arst==1'b0) q <= 1'b0;
10    else if (ena) q <= d;
11
12 endmodule
```



# Счетчик двоичный

```
1 module param_cnt
2   #(parameter WIDTH=4)
3   (
4     input clk, enable, count_up, reset,
5     output reg [WIDTH-1:0] count
6   );
7
8   always @ (posedge clk or posedge reset)
9   begin
10    if (reset)
11      count <= 0;
12    else if (enable == 1'b1)
13      count <= count + (count_up ? 1'b1 : -1'b1);
14    end
15  end
16 endmodule
```





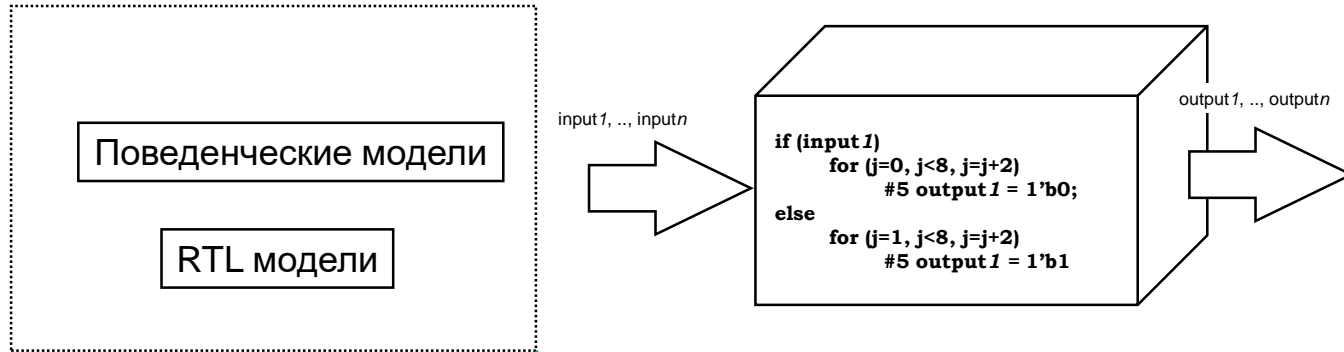


# **Иерархическое проектирование**

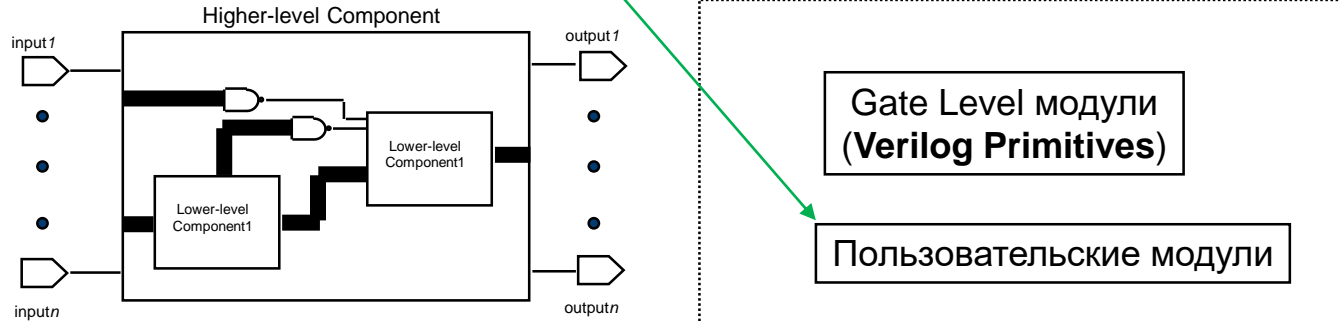
---

# Иерархическое описание

## Поведенческое описание



## Структурное описание















# Gate-Level примитивы

□ В языке Verilog определены примитивы 2 типов:

✓ Gate Type и Switch Type

□ Примитивы Gate Type

Primitive	Name	Function	Primitive	Name	Function
	<b>and</b>	n-input <b>AND</b> gate		<b>buf</b>	n-output buffer
	<b>nand</b>	n-input <b>NAND</b> gate		<b>not</b>	n-output buffer
	<b>or</b>	n-input <b>OR</b> gate		<b>bufif0</b>	tristate buffer lo enable
	<b>nor</b>	n-input <b>NOR</b> gate		<b>bufif1</b>	tristate buffer hi enable
	<b>xor</b>	n-input <b>XOR</b> gate		<b>notif0</b>	tristate inverter lo enable
	<b>xnor</b>	n-input <b>XNOR</b> gate		<b>notif1</b>	tristate inverter hi enable

# Поддержка Gate-level примитивов в пакете QII

---

## ☐ Section 7—Gates and Switches:

Section	Construct	Quartus II Support
7.2	<code>and</code> , <code>nand</code> , <code>nor</code> , <code>or</code> , <code>xor</code> , <code>xnor</code> Gates.	Supported
7.3	<code>buf</code> and <code>not</code> Gates	Supported
7.4	<code>bufif1</code> , <code>bufif0</code> , <code>notif1</code> , <code>notif0</code> Gates	Supported
7.5	MOS Switches	Not supported
7.6	Bidirectional Pass Switches	Not supported
7.7	CMOS Switches	Not supported
7.8	<code>pullup</code> and <code>pulldown</code> Sources	Not supported

# Обращение к Gate примитивам

---

- Формат для обращения к Gate-level примитивам:

```
gate_name #(delay) instance_name [array range] (terminal, terminal,...);
```

- ✓ *gate\_name* - имя примитива ( AND, NOR, BUFIF0...)
- ✓ # (delay) - задержка примитива – одна или список (задавать не обязательно)
- ✓ *instance\_name* - имя экземпляра (задавать не обязательно)
- ✓ array range – количество экземпляров примитива (если один, то задавать не обязательно)
- ✓ (*terminal*, *terminal*,...) – список сигналов, сопоставляемых с выводами

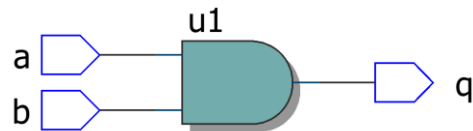
# Сопоставление сигналов выводам Gate примитивов

- Для примитивов возможно только позиционное сопоставление сигналов и выводов.
- Общее правило: первый вывод – выход, затем – входы;

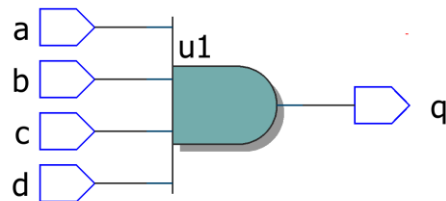
Gate примитив		Выводы и их порядок перечисления
and or xor	nand nor xnor	1-выход, 1 или более входов
buf	not	1 или более выходов, 1 - вход
buffif0 buffif1	notif0 notif1	1-выход, 1-вход, 1-вход управления

# Использование примитивов (примеры)

```
module gate_prim2 (q, a, b);  
input  a, b; output q;  
  
and u1  (q, a, b);  
  
endmodule
```

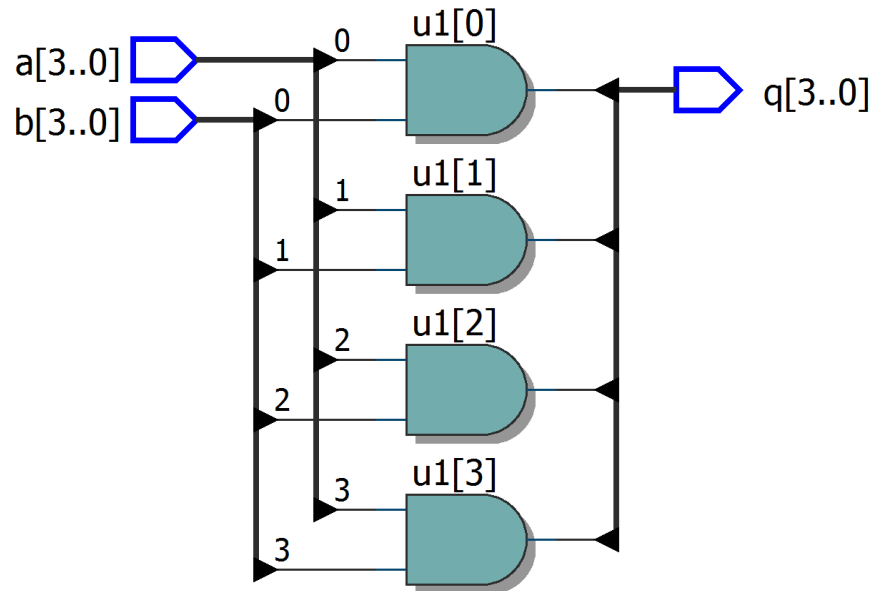


```
module gate_prim1 (q, a, b, c, d);  
input  a, b, c, d; output q;  
  
and u1  (q, a, b, c, d);  
  
endmodule
```



# Использование примитивов (примеры)

```
module gate_prim (q, a,b);  
input  [3:0] a, b;  
output [3:0] q;  
  
and u1 [3:0] (q, a, b);  
  
endmodule
```





## Задание задержки для Gate примитивов (используется при моделировании)

`gate_name` **#(delay)** `instance_name` [`array range`] (`terminal`, `terminal`,...);

Число элементов в списке	Задаваемые задержки
1	Задержка для фронта и спада сигнала
2	Задержка для фронта, задержка для спада сигнала
3	Задержка для фронта, Задержка для спада сигнала, Задержка выключения (перехода в Z состояние буфера)

```
module gate_prim_del (q, a, b);  
input  a, b; output  q;  
  
and #(3,7) u1  (q, a, b);  
  
endmodule
```

# Пример: описание полусумматора

## ❑ <gate\_name>

- ✓ and и xor

## ❑ #delay (только для примера)

- ✓ 2 time unit для and gate
- ✓ 4 time unit для xor gate

## ❑ instance\_name

- ✓ u1 для and gate
- ✓ u2 для xor gate

## ❑ Список выводов

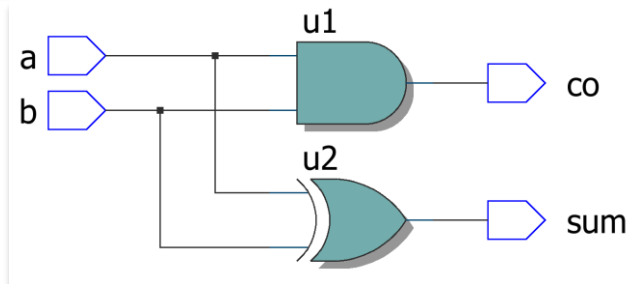
- ✓ (co, a, b) - (output, input, input)
- ✓ (sum, a, b) - (output, input, input)

```
module half_adder
(  output co, sum,
  input a, b );

  parameter and_delay = 2;
  parameter xor_delay = 4;

  and #and_delay u1(co, a, b);
  xor #xor_delay u2(sum, a, b);

endmodule
```



# Обращение к пользовательским модулям

---

## □ Формат:

```
comp_name  #(par_list) inst_name inst_array_range (port_list);
```

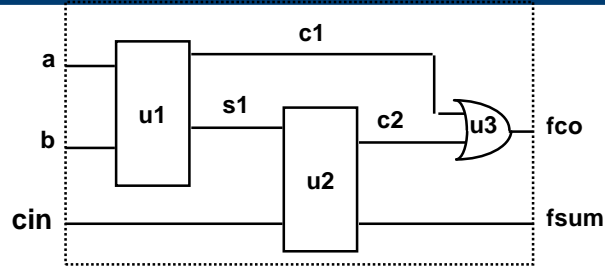
- ✓ *comp\_name* - имя модуля
- ✓  *#(par\_list)* – список значений, сопоставляемых с параметрами модуля (может быть упущен)
- ✓ *inst\_name* - имя экземпляра модуля
- ✓ *inst\_array\_range* – число используемых экземпляров модуля (если выводы – массивы соответствующего размера, то каждый модуль использует выводы с соответствующим индексом)
- ✓ *(port\_list)* - список сигналов, сопоставляемых с выводами модуля

# Сопоставление сигналов с выводами модуля

---

- ❑ Позиционное сопоставление – необходимо знать порядок перечисления выводов в модуле  
(signal, signal, ....)
- ❑ Сопоставление по именам – необходимо знать имена выводов модуля, порядок сопоставления может быть произвольным  
(.port\_name (signal), .port\_name(signal), ....)

# Соединение выводов модуля



## Позиционно U1

**half\_adder (co, sum, a, b);**

co -> c1, sum -> s1,

a -> a, b -> b

## По именам U2

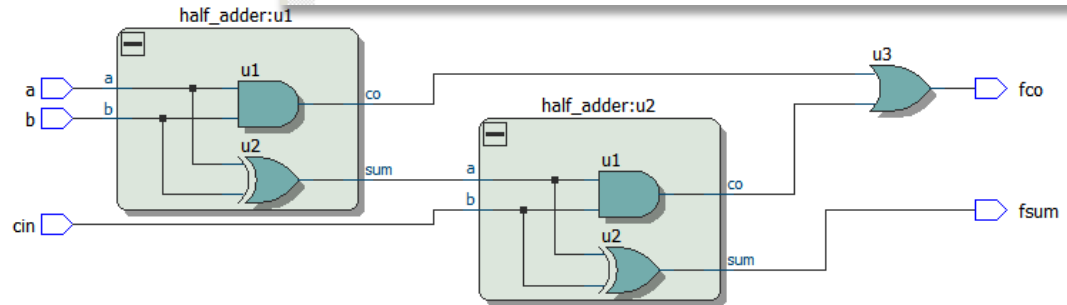
a -> s1,

b -> cin,

sum -> fsum,

co -> c2

```
module full_adder (  
    output fco, fsum,  
    input cin, a, b  
);  
    wire c1, s1, c2;  
  
    half_adder u1 (c1, s1, a, b);  
  
    half_adder u2  
        (.a(s1), .b(cin), .sum(fsum), .co(c2));  
  
    or u3(fco, c1, c2);  
  
endmodule
```



# Правила использования модулей

---

## Port Order Connections

```
module_name instance_name instance_array_range (signal, signal, ...) ;
```

## Port Name Connections

```
module_name instance_name instance_array_range  
  ( .port_name (signal) , .port_name (signal) , ... ) ;
```

## Explicit Parameter Redefinition

```
defparam heirarchy_path.parameter_name = value;
```

## In-line Implicit Parameter Redefinition

```
module_name # (value, value, ...) instance_name (signal, ... ) ;
```

## In-line Explicit Parameter Redefinition (*added in Verilog-2001*)

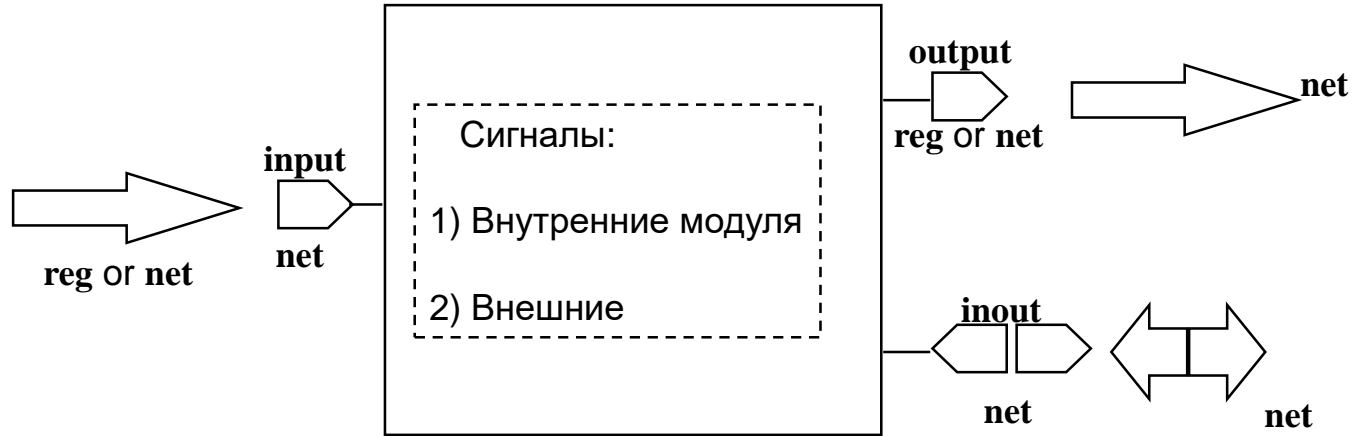
```
module_name # ( .parameter_name (value) ,  
  .parameter_name (value) , ... ) instance_name (signal, ... ) ;
```

# Использование модулей

---

- ❑ ***Gate-level modeling*** – использование встроенных в Verilog примитивов
  - ✓ and, nand, or, nor, xor, xnor
  - ✓ buf, bufif0, bufif1, not, notif0, notif1
- ❑ ***Module instantiation*** – использование созданных пользователем компонентов

# Правила соединения выводов





# Задание параметров

---

- ❑ Если модуль нижнего уровня содержит параметры, то существует два метода задания значений параметров при использовании модуля
  - ✓ Параметры получают константные значения после компиляции
- ❑ Конструкция `defparams`
- ❑ Определение параметра в экземпляре модуля

# Конструкция Defparam

- Используется конструкция **defparam** включающая иерархическое имя переопределяемого параметра

```
module full_adder (  
    output fco, fsum,  
    input cin, a, b);  
  
    wire c1, s1, c2;  
  
    defparam u1.and_delay = 4, u1.xor_delay = 6;  
    defparam u2.and_delay = 3, u2.xor_delay = 5;  
  
    half_adder u1 (c1, s1, a, b);  
    half_adder u2 (.a(s1), .b(cin),  
        .sum(fsum), .co(fco));  
    or u3(fco, c1, c2);  
  
endmodule
```

```
module half_adder (  
    output co, sum,  
    input a, b  
);  
  
    parameter and_delay = 2;  
    parameter xor_delay = 4;  
  
    and #and_delay u1(co, a, b);  
    xor #xor_delay u2(sum, a, b);  
  
endmodule
```

# Определение параметров в экземпляре модуля

- ❑ Параметры можно определить в экземпляре модуля
- ❑ В версии Verilog '2001 и страше
- ❑ Рекомендованный метод

```
module full_adder (  
    output fco, fsum,  
    input cin, a, b  
);  
  
    wire c1, s1, c2;  
  
    half_adder #(4, 6)  
        u1 (c1, s1, a, b);  
    half_adder #(.and_delay(3), .xor_delay(5))  
        u2 (.a(s1), .b(cin), .sum(fsum), .co(fco));  
    or u3(fco, c1, c2);
```

**endmodule**

Позиционное  
сопоставление

Сопоставление по  
именам (рекомендовано)

Для использования значений, заданных в самом модуле (default value):  
при сопоставлении по именам используйте

- пустое значение (`.and_delay( )`)
- или не используйте назначение параметра

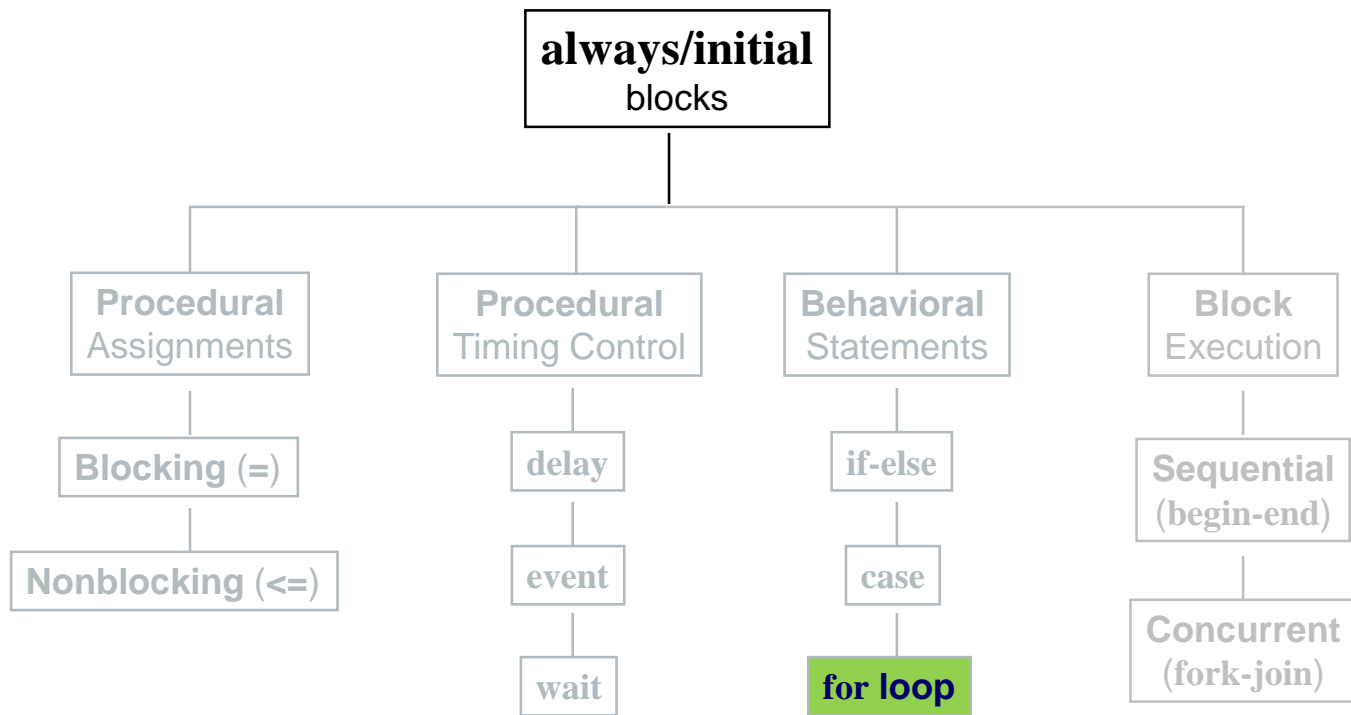


# Операторы цикла

---

# always/initial Blocks (Block Execution)

---



# Операторы цикла Loop

---

- ❑ **forever** loop – выполняется постоянно
  - ❑ **repeat** loop – выполняется определенное число раз
  - ❑ **while** loop – выполняется если выражение истинно
  - ❑ **for** loop – выполняется один раз в начале цикла и затем выполняется если выражение истинно
- ⇒ Операторы Loop используется для задания повторяющихся операций

# Операторы процедурных блоков

---

- Используются внутри процедурных блоков



- ✓ **if-else** statement

- ✓ **case** statement

- ✓ Loop statements

- **for** loop

- repeat loop

- while loop

- forever loop

Синтезируемые

Возможно синтезировать  
(лучше избегать в RTL)

Не синтезируемые

- **forever loop** – выполняется постоянно

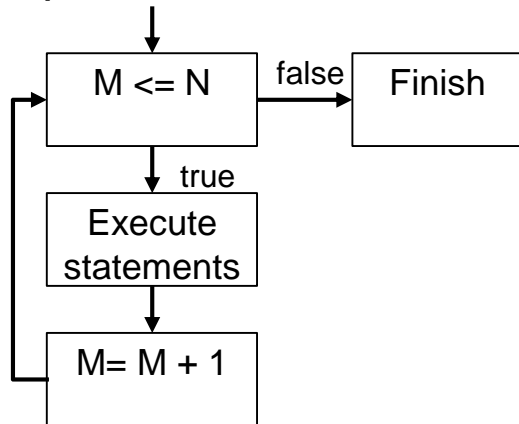
```
initial begin  
    clk = 0;  
    forever #25 clk = ~clk;  
end
```

Тактовый сигнал с периодом  
50 единиц времени



# repeat Loop

- **repeat loop** – выполняется определенное число раз



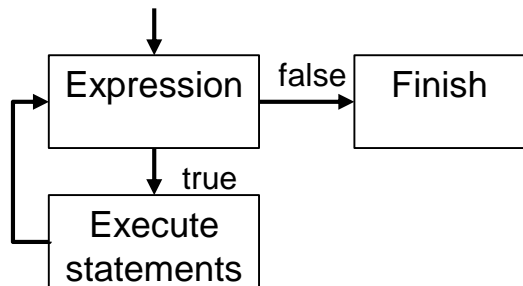
```
if (rotate == 1)
    repeat (8) begin
        tmp = data[15];
        data = {data << 1, tmp};
    end
```

Повторяет оператор сдвига  
8 раз

- М.б. синтезирован если количество выполнений фиксировано

# while Loop

- **while** loop – выполняется если выражение истинно



```
initial begin  
count = 0;  
while (count < 101) begin  
    $display ("Count = %d", count);  
    count = count + 1;  
end  
end
```

Считает от 0 до 100  
Выходит из цикла при count= 101

- М.б. синтезирован если количество выполнений фиксировано

# FOR Loop Statement

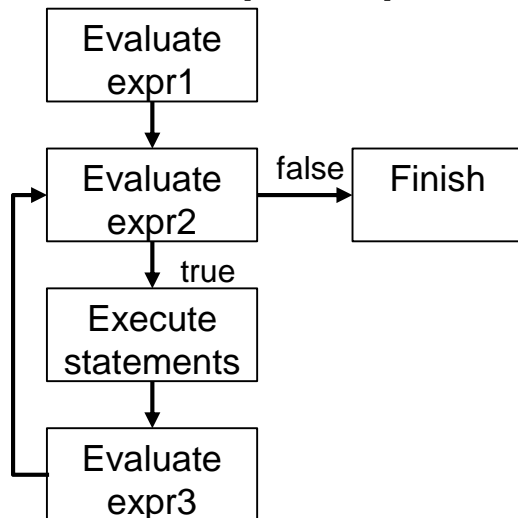
## ❑ `for(expr1; expr2; expr3) <statements>`

✓ Если несколько statements тогда

– **`begin <statements> end`**

## ❑ `<statements>` повторяется N раз

✓ N задано с помощью **`expr1; expr2; expr3`**



*integer i;*

...

***always @ (posedge clk)***

***for (i=1; i<=3; i=i+1)***

***begin <statements> end***

*reg [3:0] j;*

...

***always @ \****

***for (j=1; j<=3; j=j+1)***

***begin <statements> end***

# FOR Loop (сдвигающий регистр)

**module ex\_1**

**(input** clk, d,  
**output** reg q);

**reg** [3:0] val;

**integer** i;

**always@\*** begin

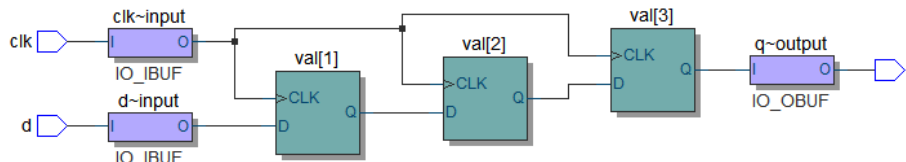
val[0] = d;

q = val[3];**end**

**always @ (posedge**  
clk)

**for** (i=1; i<=3; i=i+1)  
val[i] <= val[i-1];

**endmodule**



*val[0] = d;*

*q = val[3];*

*val[1] <= val[0];*

*val[2] <= val[1];*

*val[3] <= val[2];*

**module ex\_1**

**#(parameter wdt = 3)**

**(input** clk, d,

**output** reg q);

**reg** [wdt:0] val;

**integer** i;

**always@\*** begin

val[0] = d;

q = val[wdt];**end**

**always @ (posedge** clk)

**for** (i=1; i<=wdt; i=i+1)

val[i] <= val[i-1];

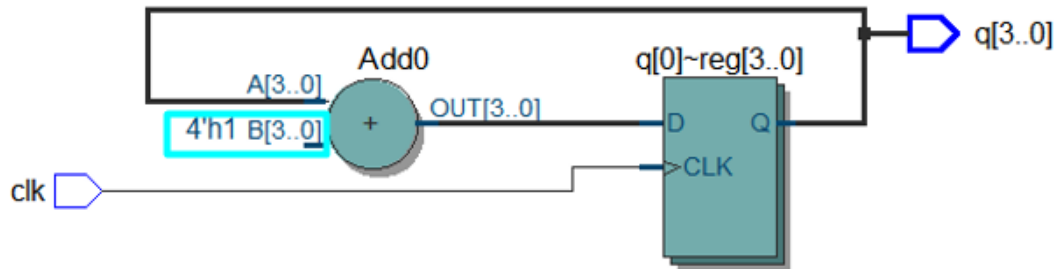
**endmodule**

# Что будет синтезировано?

```
module ex_1
(input clk,
 output reg [3:0] q);
 integer i;
 initial q = 4'd0;
 always @ (posedge clk)
   for (i=1; i<=3; i=i+1)
     q <= q + 1;
endmodule
```

Почему?

Т.к. будет понято компилятором как описание:



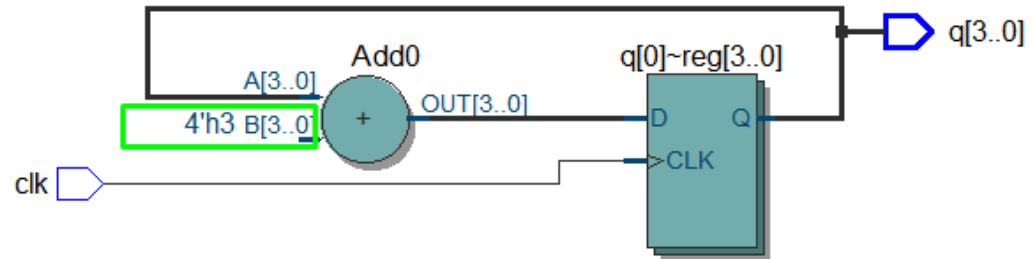
```
module ex_1
(input clk,
 output reg [3:0] q);
 integer i;
 initial q = 4'd0;
 always @ (posedge clk)
   q <= q + 1;
   q <= q + 1;
   q <= q + 1;
endmodule
```

# Что будет синтезировано?

```
module ex_1
(input clk,
 output reg [3:0] q);
  integer i;
  initial q = 4'd0;
  always @ (posedge clk)
    for (i=1; i<=3; i=i+1)
      q <= q + i;
endmodule
```

Почему?

Т.к. будет понято компилятором как описание:



```
module ex_1
(input clk,
 output reg [3:0] q);
  integer i;
  initial q = 4'd0;
  always @ (posedge clk)
    q <= q + 1;
    q <= q + 2;
    q <= q + 3;
endmodule
```