

ОСНОВЫ

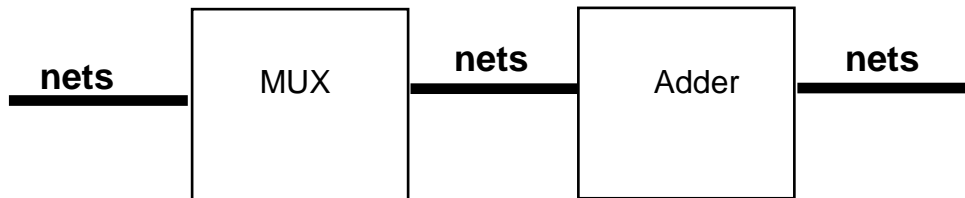
VerilogHDL/SystemVerilog

(синтез и моделирование)

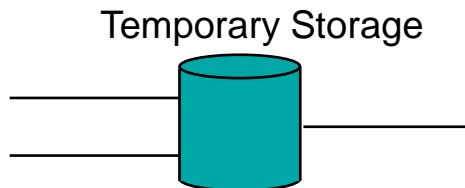
Группа типов данных Variable

Типы данных (напоминание)

- Группа типов данных Net – представляет физическую связь между элементами структуры



- Группа типов данных Variable – представляет элементы для временного хранения данных



Типы данных группы Variable

Типы данных	Для чего используется	Поддержка синтеза
reg	Переменная. Для знакового представления используйте reg signed.	Y
integer	Знаковая переменная (обычно 32 бита)	Y
time	Без знаковое целое (обычно 64 бита) используется для хранения времени при моделировании	N
real	Переменная с плавающей запятой, двойной точности	N
realtime	Переменная с плавающей запятой, двойной точности, используемая с time	N

*Тип **reg** не соответствует физическому триггеру*

Использование типа данных Reg

- ❑ Тип данных Reg могут иметь:
 - ✓ Выходы модуля
 - ✓ Внутренние сигналы модуля
- ❑ С типом данных Reg можно использовать:
 - ✓ Вектора
 - ✓ Массивы
 - ✓ Все операторы, которые использовали для типа данных Net
- ❑ Используется **только** в процедурных блоках

Процедурные блоки

Два типа процедурных блоков

❑ initial

- ✓ Используется для инициализации начальных значений

❑ always

- ✓ Используется для задания алгоритма работы устройства

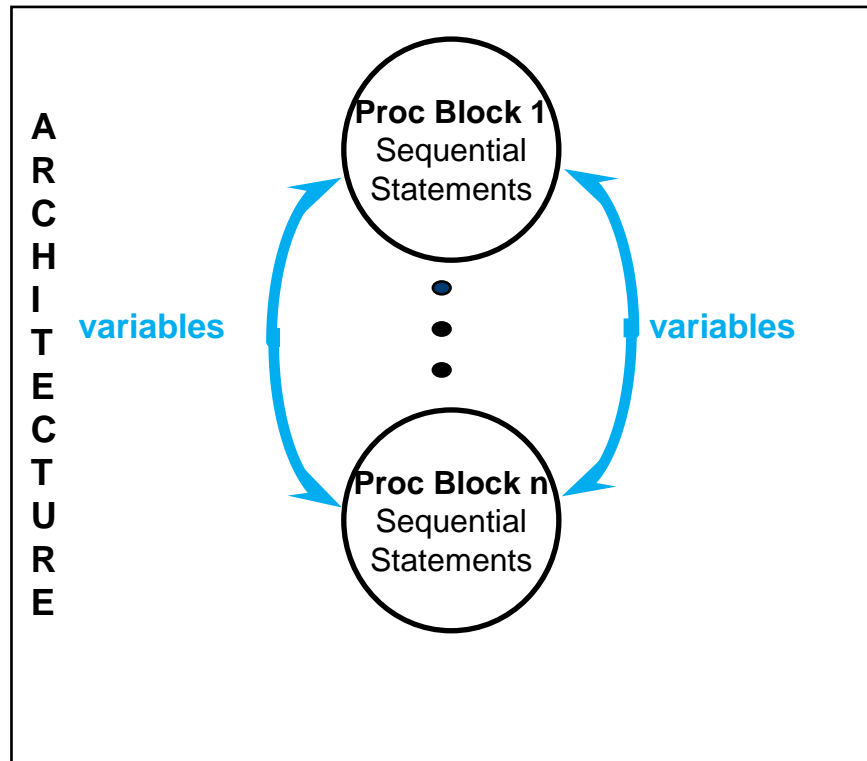
❑ Каждый always и initial блок - отдельный процесс

❑ always и initial не могут использоваться рекурсивно

❑ Особенности назначений **в рамках процедурных блоков**

- ✓ **LHS** - тип данных из группы **variable** (reg, integer, ...)
- ✓ **RHS** - типы данных групп net и variable или вызовы функций.

Процедурные блоки



- ❑ Каждый процедурный блок выполняется параллельно со всеми остальными блоками и непрерывными назначениями
 - ✓ Последовательность использования процедурных блоков `always/initial` не имеет значения
- ❑ Внутри процедурного блока операторы выполняются последовательно (`begin ... end`)
 - ✓ Последовательность использования операторов внутри процедурного блока имеет значение.

Выполнение процедурных блоков

- ❑ Все процедурные блоки (процессы) во всех модулях проекта выполняются вместе
 - ✓ Выполнение начинается в момент времени 0
 - ✓ Текущий цикл выполнения заканчивается когда процедурный блок достигает одного из условий
 - Выполнен последний оператор блока initial
 - Выполняется блокирующее (Blocking) назначение с задержкой
 - Достигнут оператор с управлением событиями Event control
 - Достигнут оператор Wait statement
 - ✓ После завершения всеми процессами выполнения текущего цикла (на данном временном шаге), модельное время увеличивается (переход к следующему временному шагу)

- ❑ Текущий цикл выполнения (текущий временной шаг) используется для определения поведения модели (описания модуля)
 - ✓ Simulation tool: Текущий временной шаг выполнения процесса соответствует времени моделирования
 - ✓ Synthesis tool: Порождает функциональность соответствующую физической аппаратуре.

Процедурный блок `initial`

- ❑ Содержит **поведенческое** описание
- ❑ Каждый `initial` выполняется начиная с момента времени 0
 - ✓ **выполняется только один раз** и больше не выполняется
- ❑ Если блок содержит более одного оператора, то для группировки операторов необходимо использовать ключевые слова `begin` и `end` (либо `fork join`)
- ❑ Примеры использования
 - ✓ Инициализация сигналов **для синтезируемых описаний и описаний тестов**;
 - Любая функциональность, которая должна быть активирована один раз
- ❑ Обратите внимание:
 - ✓ хотя блок `initial` выполняется только один раз, но операторы внутри блока могут продолжать работать все время моделирования

Процедурный блок `always`

- ❑ Содержит поведенческое описание
- ❑ Каждый блок `always` выполняется начиная с момента времени 0
 - ✓ выполняется постоянно (циклически)
- ❑ Если блок содержит более одного оператора, то для группировки операторов необходимо использовать ключевые слова `begin` и `end` (либо `fork` `join`)
- ❑ Примеры использования
 - ✓ **для синтезируемых описаний и описаний тестов**
 - Любой процесс или функциональность, выполнение которых должно повторяться циклически

Примеры

```
...  
initial
```

```
    clk = 1'b0;
```

```
always
```

```
    #1 clk = ~clk;
```

```
...
```

```
...  
initial  
begin
```

```
    clk = 1'b0;
```

```
end
```

```
always  
begin
```

```
    #1 clk = ~clk;
```

```
end
```

```
...
```

```
...  
initial
```

```
    clk = 1'b0;  
    dat = 0'b0;
```

```
always
```

```
    #1 clk = ~clk;  
    #2 dat = dat & dat;
```

```
...
```

```
...  
initial  
begin
```

```
    clk = 1'b0;  
    dat = 0'b0;
```

```
end
```

```
always  
begin
```

```
    #1 clk = ~clk;  
    #2 dat = dat & dat;
```

```
end
```

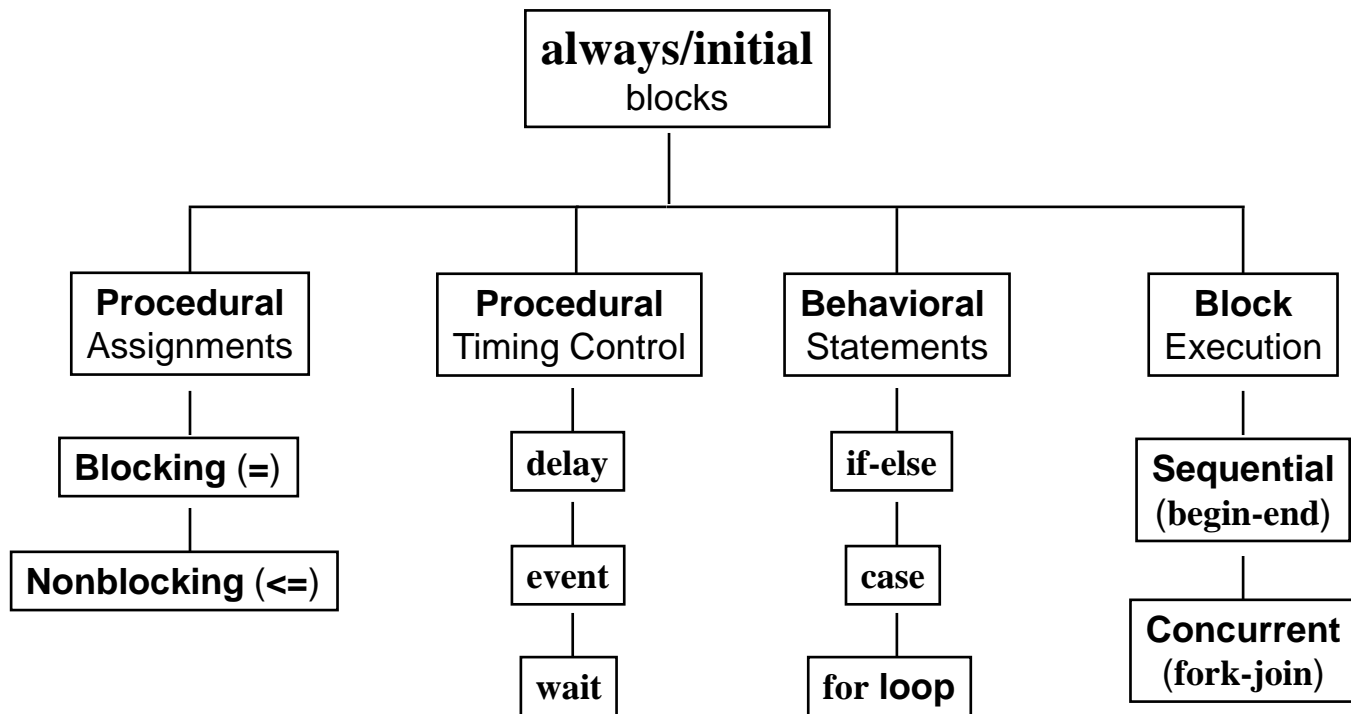
```
...
```

Именование процедурных блоков

- ❑ Процедурному блоку может быть присвоено имя
 - ✓ Необходимо использовать `begin ... end` (fork ... join) даже с одним оператором
 - ✓ после `begin (fork)` добавляется имя блока
- ❑ Преимущества
 - ✓ Позволяет ссылаться на процедурные блоки по именам
 - ✓ Позволяет декларировать локальные объекты для процедурного блока
 - ✓ Позволяет в системе моделирования осуществлять мониторинг процедурного блока по имени

```
initial  
begin : clock_init  
    clk = 1'b0;  
end  
  
always  
begin : clock_proc  
    clk = ~clk;  
end
```

Особенности блоков **always/initial**



Выполнение блоков (Block Execution)

❑ Два способа выполнения блоков

- ✓ Sequential
- ✓ Parallel

❑ Sequential (последовательные блоки)

- ✓ Операторы между **begin** и **end** выполняются последовательно
- ✓ Если имеется несколько операторов внутри блоков **initial** или **always** и вы хотите чтобы они выполнялись последовательно, операторы должны быть сгруппированы (обрамлены) ключевыми словами **begin** и **end**
- ✓ **Поддерживается синтезом**

❑ Parallel (параллельные блоки)

- ✓ Операторы между **fork** и **join** выполняются параллельно
- ✓ Если имеется несколько операторов внутри блоков **initial** или **always** и вы хотите чтобы они выполнялись параллельно, операторы должны быть сгруппированы (обрамлены) ключевыми словами **fork** и **join**
- ✓ **Не поддерживается синтезом**

Управление событиями Event Control

- ❑ Задается выражением `@(<event>)`
 - ✓ Выражение приостанавливает выполнение процедурных операторов до наступления события (до изменения значения выражения) у переменной из списка `<event>`
- ❑ Выражение `@(<event>)` может использоваться только в процедурных блоках `initial` или `always`
- ❑ Обеспечивает управление, чувствительное к изменениям выражения:
 - ✓ `@(clk)` – выполнение приостанавливается до любого изменения `clk`
 - ✓ `@(posedge clk)` – выполнение приостанавливается до появления фронта `clk`
 - Фронт определяется как один из переходов : `0=>1`, `0=>x`, `0=>z`, `x=>1`, `z=>1`
 - ✓ `@(negedge clk)` – выполнение приостанавливается до появления спада `clk`
 - Фронт определяется как один из переходов : `1=>0`, `1=>x`, `x=>0`, `z=>0`
- ❑ Для проверки нескольких событий используют логическое ИЛИ:
 - ✓ Запятая (,) в Verilog '2001: `@(<event>, <event>, ...)` ; or в Verilog 95

Управление событиями и список чувствительности

- ❑ Использование `@(<event>)` в начале блока `always` позволяет контролировать момент начала выполнения блока
 - ✓ Для начала выполнения блока `always` требуется наступление события
 - ✓ Блок `always` становится “чувствительным” к переменным в списке `<event>`
- ❑ Список `<event>` называют – списком чувствительности (Sensitivity List)
- ❑ Использование управлением событиями поддержано средствами синтеза

Формат

```
always @(sensitivity_list) begin  
    -- Statement_1  
    -- .....  
    -- Statement_N  
end
```

Пример

```
// Процесс запускается если  
// значение любого из входов  
// a, b, c или d изменяется  
always @(a, b, c, d) begin  
    y = (a ^ b) & (c ~ | d);  
end
```

Два типа RTL процессов

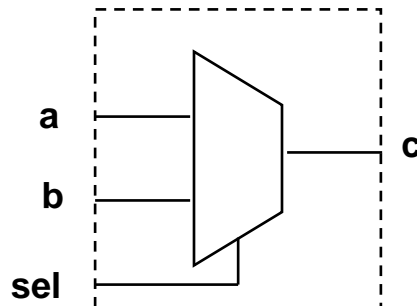
- Комбинационный процесс

- Чувствителен ко всем сигналам в процессе

```
always @ (a, b, sel)  
always @ *
```

** - добавить все входы*

Список чувствительности включает все входы комбинационной цепи

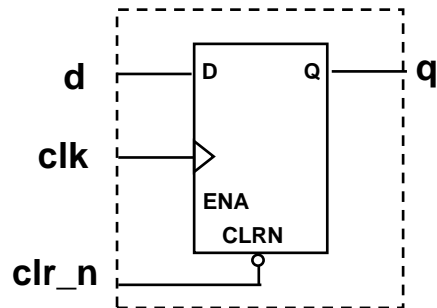


- Тактовый (регистровый) процесс

- Чувствителен к тактовым сигналам и сигналам управления

```
always @(posedge clk, negedge clr_n)
```

Список чувствительности не включает d вход, а только тактовый сигнал и сигнал асинхронного сброса



Неполный список чувствительности

- ❑ Неполный список чувствительности в процедурном блоке **always** может привести к различиям в при функциональном и временном моделировании (моделировании после синтеза)
 - ✓ Большинство средств синтеза предполагают полный список чувствительности (**могут его дополнить**)
 - ✓ Средства моделирования:
 - При функциональном моделировании используется неполный список
 - При временном – результаты синтеза, в котором этот список полный.

```
always @ (a, b)  
y = a & b & c;
```

Ошибка – при моделировании поведение не соответствует 3-входовому AND

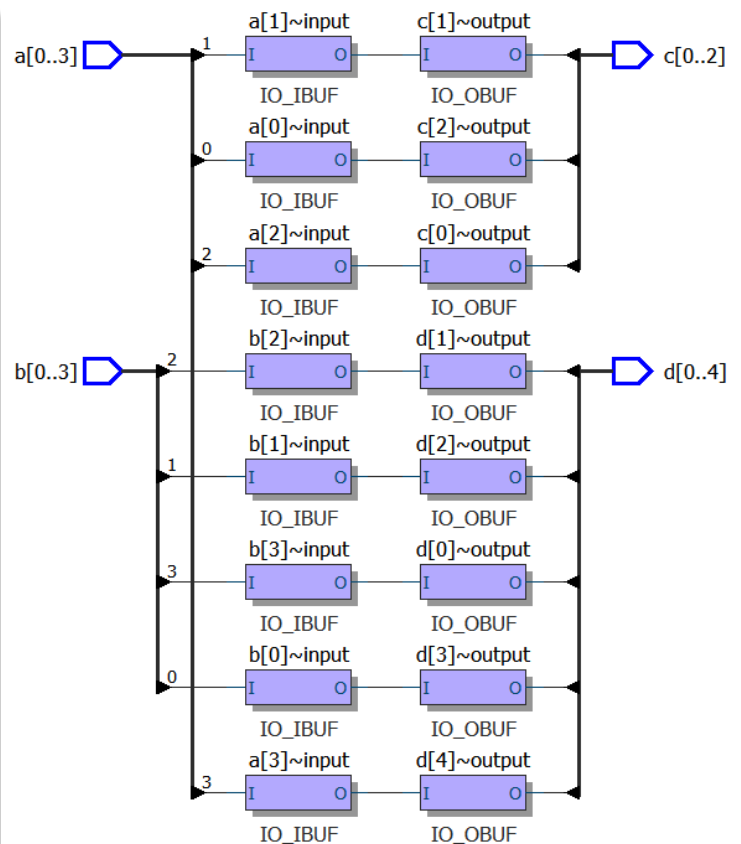
```
always @ *  
y = a & b & c;
```

Использование @* позволяет решить проблему

Пример обращение к элементам вектора

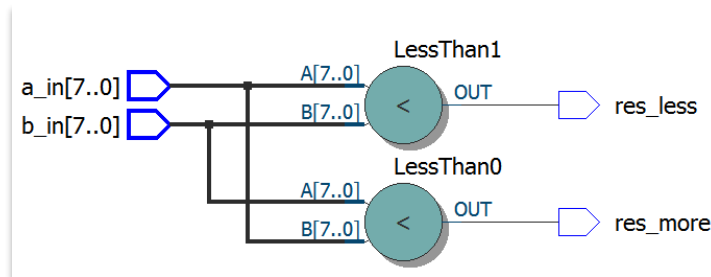
```
1 module al_ex8(a, b, c, d);
2   input [0:3] a, b;
3   output reg [2:0] c;
4   output reg [4:0] d;
5
6   always @*
7   begin
8     c = a[0+:3];
9     d[3 -:4] = b;
10    d [4 -:1] = a[3+:1];
11  end
12 endmodule
```

```
1 module ex8(a, b, c, d);
2   input [0:3] a, b;
3   output [2:0] c;
4   output [4:0] d;
5
6   assign c = a[0+:3];
7   assign d[3 -:4] = b;
8   assign d [4 -:1] = a[3+:1];
9 endmodule
```



Пример оператора сравнения

```
1 module al_ex6 (a_in, b_in, res_more, res_less );
2
3 input [7:0] a_in;
4 input [7:0] b_in;
5 output reg res_more, res_less;
6
7 always @*
8 begin
9     res_more = a_in > b_in;
10    res_less = a_in < b_in;
11 end
12
13 endmodule
```

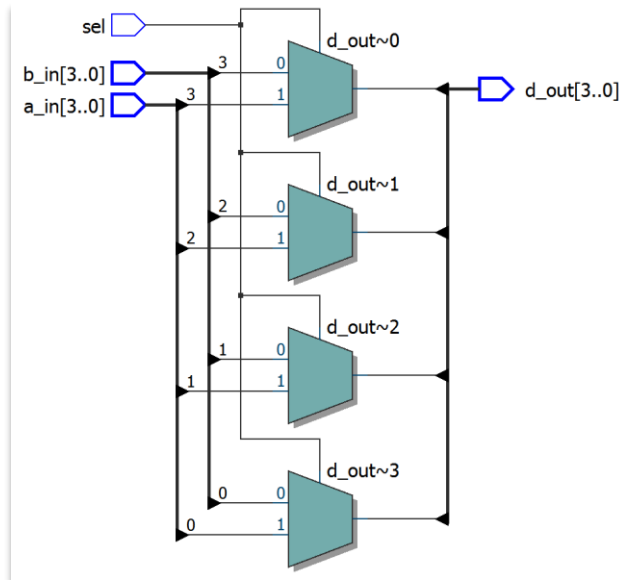


```
1 module ex6 (a_in, b_in, res_more, res_less );
2
3 input [7:0] a_in;
4 input [7:0] b_in;
5 output res_more, res_less;
6
7 assign res_more = a_in > b_in;
8 assign res_less = a_in < b_in;
9
10 endmodule
```

Пример оператора условного выбора

```
1 module al_ex_co (a_in, b_in, sel, d_out );
2
3 input  [3:0] a_in, b_in;
4 input  sel;
5 output reg [3:0] d_out;
6
7 always @*
8 | d_out = (sel=='b1)? a_in:b_in;
9
10 endmodule
```

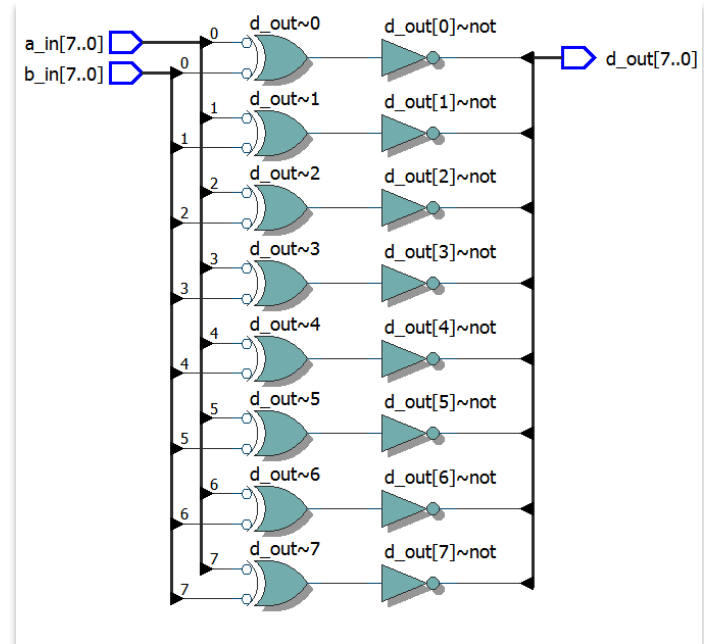
```
1 module ex_co (a_in, b_in, sel, d_out );
2
3 input  [3:0] a_in, b_in;
4 input  sel;
5 output [3:0] d_out;
6
7 assign d_out = (sel=='b1)? a_in:b_in;
8
9 endmodule
```



Пример оператора bitwise

```
1 module al_ex_bw (a_in, b_in, d_out );
2
3 input [7:0] a_in;
4 input [7:0] b_in;
5 output reg [7:0] d_out;
6
7 always @*
8 d_out = (~a_in ^ ~b_in );
9
10 endmodule
```

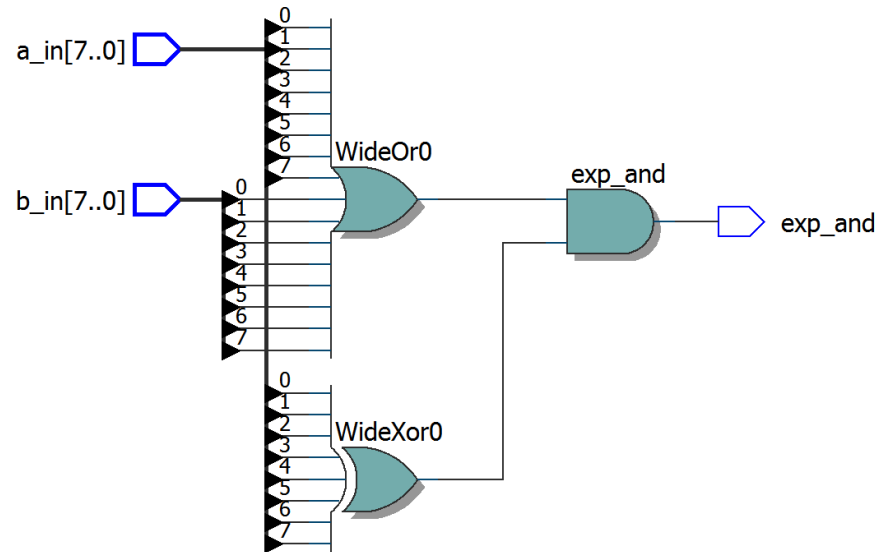
```
1 module ex_bw (a_in, b_in, d_out );
2
3 input [7:0] a_in;
4 input [7:0] b_in;
5
6 output [7:0] d_out;
7
8 assign d_out = (~a_in ^ ~b_in );
9
10 endmodule
```



Пример на все типы логических функций

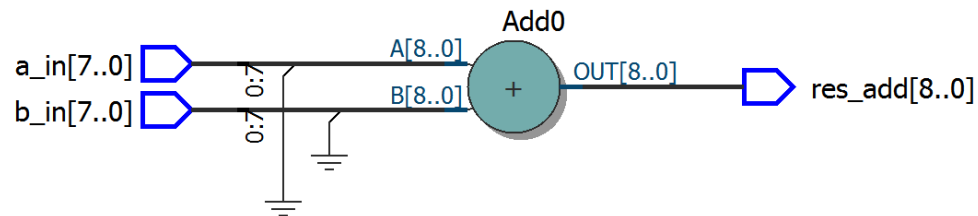
```
1 module al_ex8_ (a_in, b_in, exp_and );
2
3 input  [7:0] a_in, b_in;
4 output reg exp_and;
5
6 always @*
7     exp_and = ^a_in && (a_in | b_in);
8
9 endmodule
```

```
1 module ex8_ (a_in, b_in, exp_and );
2
3 input  [7:0] a_in, b_in;
4 output  exp_and;
5
6 assign exp_and = ^a_in && (a_in | b_in);
7 endmodule
```



Пример ADD

```
1 module al_ex1 (a_in, b_in, res_add );
2
3   input  [7:0] a_in, b_in;
4   output reg [8:0] res_add;
5
6   always @*   res_add = a_in + b_in;
7
8 endmodule
```

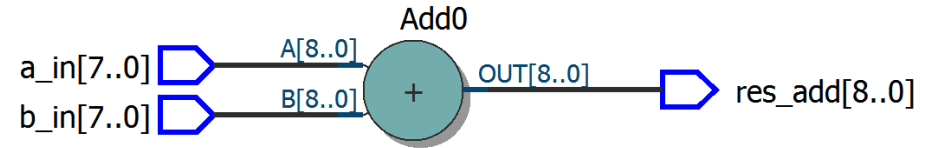


```
1 module ex1 (a_in, b_in, res_add );
2
3   input  [7:0] a_in, b_in;
4   output [8:0] res_add;
5
6   assign res_add = a_in + b_in;
7
8 endmodule
```

	0	1	2	3	4
Edit:/ex1/a_in	0				
Edit:/ex1/b_in	255				
sim:/ex1/res_add	255	256	257	258	259

Пример знакового сумматора

```
1 module al_ex1s (a_in, b_in, res_add );
2
3   input signed [7:0] a_in, b_in;
4   output reg signed [8:0] res_add;
5
6   always @* res_add = a_in + b_in;
7
8 endmodule
```

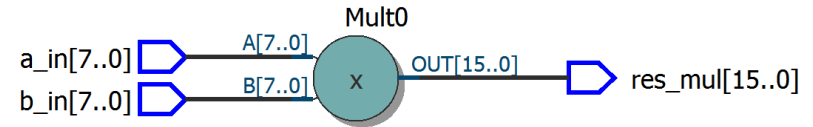


```
1 module ex1s (a_in, b_in, res_add );
2
3   input signed [7:0] a_in, b_in;
4   output signed [8:0] res_add;
5
6   assign res_add = a_in + b_in;
7
8 endmodule
```

/ex1s_tb/a_in	104	119	120	121	122	123	124	125	126	127	-128	-127	-126	-125	-124	-123	-122
/ex1s_tb/b_in	104	119	120	121	122	123	124	125	126	127	-128	-127	-126	-125	-124	-123	-122
/ex1s_tb/res_add	208	238	240	242	244	246	248	250	252	254	-256	-254	-252	-250	-248	-246	-244

Пример знакового умножителя

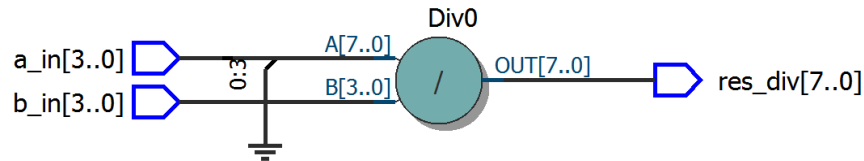
```
1 module al_ex2s (a_in, b_in, res_mul );
2
3   input signed [7:0] a_in, b_in;
4   output reg signed [15:0] res_mul;
5
6   always @* res_mul = a_in * b_in;
7
8 endmodule
```







```
1 module ex2s (a_in, b_in, res_mul );
2
3   input signed [7:0] a_in, b_in;
4   output signed [15:0] res_mul;
5
6   assign res_mul = a_in * b_in;
7
8 endmodule
```

Пример делителя с 4 знаками после запятой

```
1 module al_ex3_ (a_in, b_in, res_div );
2
3 input [3:0] a_in, b_in;
4 output reg [7:0] res_div;
5
6 always
7 begin
8     res_div = {a_in, 4'h0} / b_in;
9
10 end
11
12 endmodule
```



	Name	Value at 0 ps	0 ps	160,0 ns	320,0 ns	480,0 ns	640,0 ns	800,0 ns	960,0 ns
	a_in	U 1	1	2	3	4			
	b_in	U 0	0	1	2	3			
	W	U 15	15	2	1				
	f	F 0.9375	0.9375	0	0.5	0.3125			

Поведенческие операторы

- ❑ Описывают алгоритм работы
- ❑ Должны быть использованы в рамках процедурного блока

- ❑ Поведенческие операторы
 - ✓ if-else
 - ✓ case

Условный оператор if-else

❑ Тип 1:

- ✓ if (<expression>) true_statement(s);

❑ Тип 2:

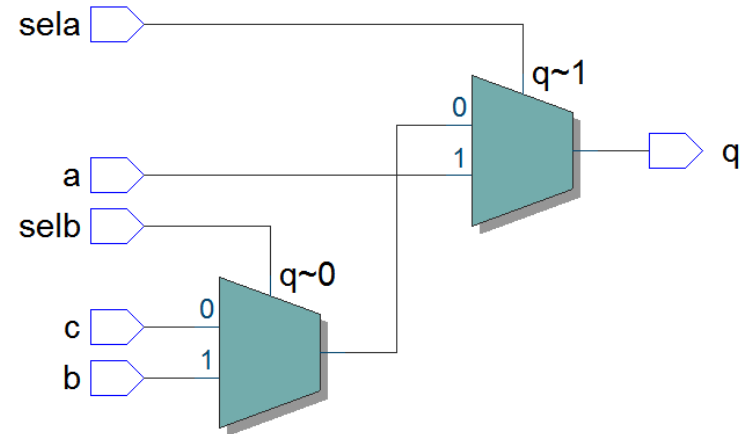
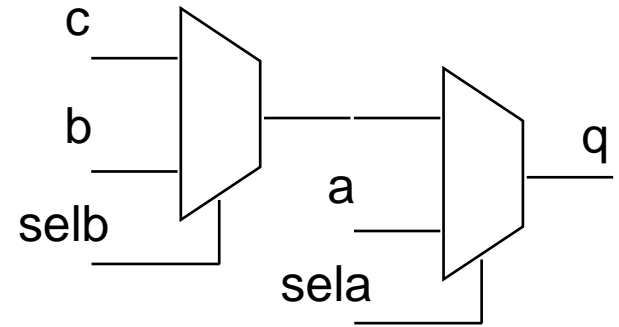
- ✓ if (<expression>) true_statement(s); else false_statement(s);

❑ Тип 3:

- ✓ if (<expression1>) true_statement(s) 1;
- ✓ else if (<expression2>) true_statement(s) 2;
- ✓ ...
- ✓ else if (<expression n >) true_statement(s) n;
- ✓ else default_statement(s);
 - Первое истинное условие вызывает выполнение связанных с ним операторов.
 - Условия оцениваются сверху вниз => приоритет условий
 - Если все условия ложны, тогда выполняются операторы, связанные с терминирующим “else”
 - Если есть несколько операторов _statement(**s**) - используется **begin end**

Пример

```
module ex_1 (  
  input sela, selb, a, b, c,  
  output reg q);  
  
  always @ (sela, selb, a, b, c) begin  
    //always @ * begin  
    if (sela)  
      q = a;  
    else if (selb)  
      q = b;  
    else  
      q = c;  
  end  
  
endmodule
```

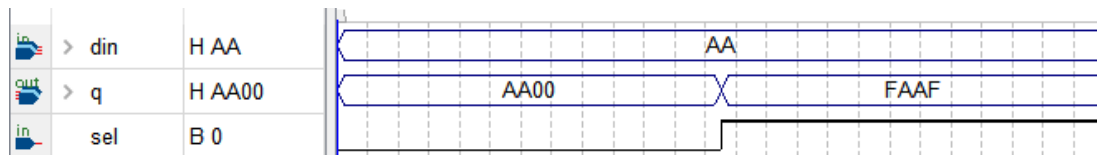


ВОПРОС: нужны ли здесь begin end ?

```
module ex_1 (  
    input sela, selb, a, b, c,  
    output reg q);  
  
    always @ (sela, selb, a, b, c) begin  
//always @ * begin  
        if (sela)  
            q = a;  
        else if (selb)  
            q = b;  
        else  
            q = c;  
    end  
  
endmodule
```


Пример

```
module ex_1 (  
    input sel,  
    input [7:0] din,  
    output reg [15:0] q);  
  
    always @ * begin  
        if (sel) begin  
            q[11: 0] = {din, {4{1'b1}} };  
            q[15:12] = 4'hf; end  
        else  
            q = din<<8;  
        end  
    end  
endmodule
```



Example

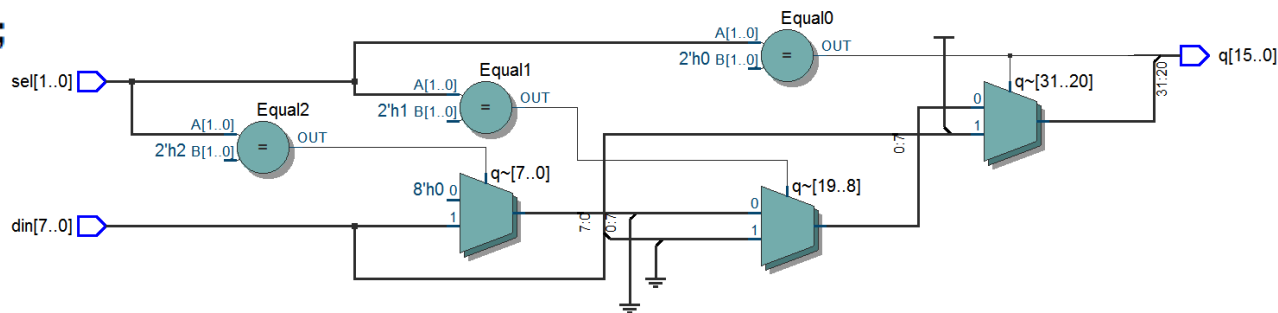
```
module ex_1 (
input  [1:0] sel,
input  [7:0] din,
output reg [15:0] q);
```

```
always @ * begin
  if (sel == 2'h0) begin
    q[11: 0] = {din, {4{1'b1}} };
    q[15:12] = 4'hf; end
  else if (sel == 2'h1)
    q = din<<8;
  else if (sel == 2'h2)
    q = din<<4;
  else
    q = 8'h00;
end
```

```
end
```

```
endmodule
```

in	> din	H AA	AA			
out	> q	H FAAF	FAAF	AA00	0AA0	0000
in	> sel	B 00	00	01	10	11



Оператор условного выбора case

- ❑ Формат оператора:

```
case {expression}
  <condition1> : {sequence of statements 1}
  <condition2> : {sequence of statements 2}
  ...
  default :      {sequence of statements  }
endcase
```

- ❑ Если {sequence of statements } содержит несколько операторов, то они группируются: **begin end**
- ❑ Условия оцениваются по порядку
- ❑ X и Z рассматриваются как логические значения
- ❑ **default** задает все другие возможные значения, которые не были явно указаны

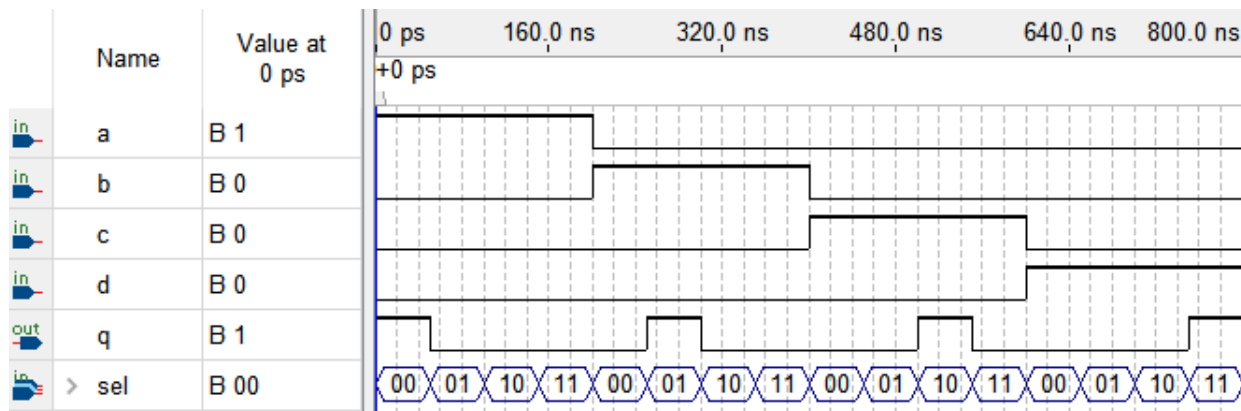
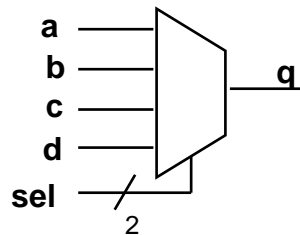
case Statements

- ❑ Verilog не требует (хотя это рекомендовано) чтобы:
 - ✓ Были рассмотрены все возможные значения
 - ✓ Все условия были уникальными (выбирается первое подходящее значение)
- ❑ Оператор case statements обычно синтезируется более эффективно если все условия уникальны (unique/parallel)
 - ✓ Наличие не уникальных условий приводит к появлению приоритета (как в if)
- ❑ Рекомендации
 - ✓ Рассматривайте все возможные значения (full case)
 - Не рассмотренные значения приводят к появлению триггеров-защелок
 - ✓ Определяйте выходы для всех условий
 - Если выход не определен для какого либо условия => триггер-защелка

Пример

```
module ex_1 (  
  input [1:0] sel,  
  input a, b, c, d,  
  output reg q);
```

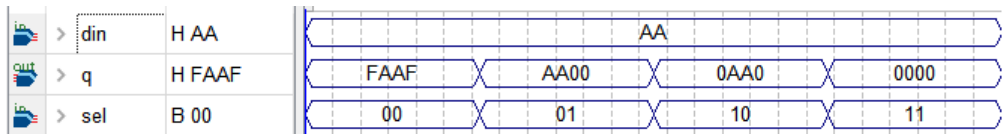
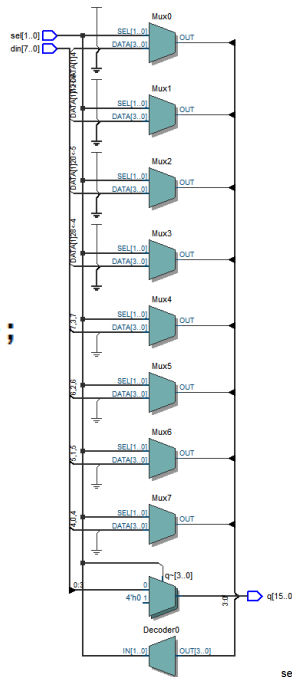
```
  always @ (sel, a, b, c, d)  
  //always @ *  
  case (sel)  
    2'b00 : q = a;  
    2'b01 : q = b;  
    2'b10 : q = c;  
    default : q = d;  
  endcase  
endmodule
```



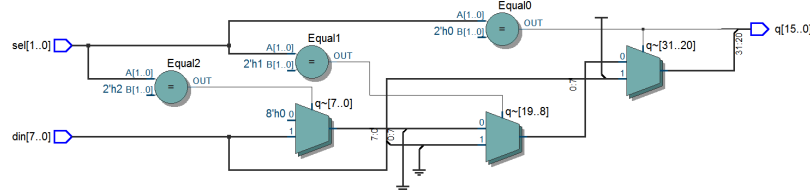
Пример

Без приоритизации

```
module ex_1 (  
  input [1:0] sel,  
  input [7:0] din,  
  output reg [15:0] q);  
  
always @ *  
  case (sel)  
    2'b00 : begin  
      q[11:0] = {din, {4{1'b1}}};  
      q[15:12] = 4'hf; end  
    2'b01 : q = din<<8;  
    2'b10 : q = din<<4;  
    default : q = 0;  
  endcase  
endmodule
```



```
module ex_1 (  
  input [1:0] sel,  
  input [7:0] din,  
  output reg [15:0] q);  
  
always @ * begin  
  if (sel == 2'h0) begin  
    q[11:0] = {din, {4{1'b1}}};  
    q[15:12] = 4'hf; end  
  else if (sel == 2'h1)  
    q = din<<8;  
  else if (sel == 2'h2)  
    q = din<<4;  
  else  
    q = 8'h00;  
end  
endmodule
```



Две дополнительные форм оператора case

□ casez

- ✓ **Z** и **?** Рассматриваются как «любое значение»

```
casez (encoder)
  4'b1??? : high_lvl = 3;
  4'b01?? : high_lvl = 2;
  4'b001? : high_lvl = 1;
  4'b0001 : high_lvl = 0;
  default : high_lvl = 0;
endcase
```

- if **encoder** = 4'b1z0?, then **high_lvl** = 3

□ casex

- ✓ **X**, **Z**, and **?** Рассматриваются как «любое значение»

```
casex (encoder)
  4'b1xxx : high_lvl = 3;
  4'b01xx : high_lvl = 2;
  4'b001x : high_lvl = 1;
  4'b0001 : high_lvl = 0;
  default : high_lvl = 0;
endcase
```

- if **encoder** = 4'b1zxx, then **high_lvl** = 3

Пример

```
module prio_encoder_case
(
    input wire [4:1] r,
    output reg [2:0] y
);

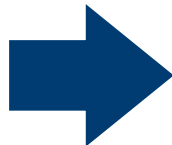
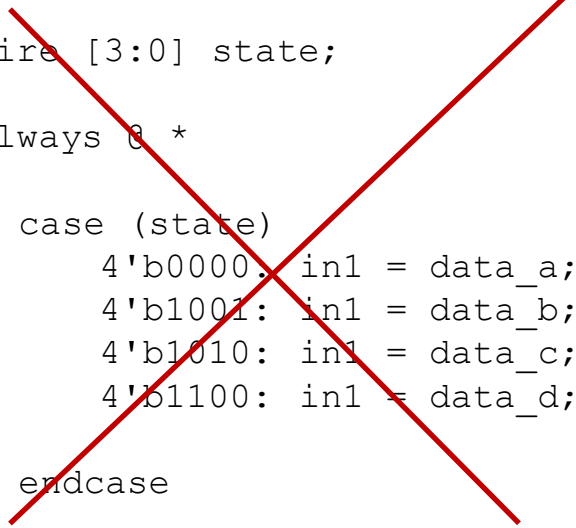
always @*
    case(r)
        4'b1000, 4'b1001, 4'b1010, 4'b1011,
        4'b1100, 4'b1101, 4'b1110, 4'b1111:
            y = 3'b100;
        4'b0100, 4'b0101, 4'b0110, 4'b0111:
            y = 3'b011;
        4'b0010, 4'b0011:
            y = 3'b010;
        4'b0001:
            y = 3'b001;
        4'b0000: // default can also be used
            y = 3'b000;
    endcase
endmodule
```

```
module prio_encoder_casez
(
    input wire [4:1] r,
    output reg [2:0] y
);

always @*
    casez(r)
        4'b1??? : y = 3'b100;
        4'b01?? : y = 3'b011;
        4'b001? : y = 3'b010;
        4'b0001 : y = 3'b001;
        4'b0000 : y = 3'b000;
    endcase
endmodule
```


Задание значения по умолчанию

```
...  
wire [3:0] state;  
  
always @ *  
  
    case (state)  
        4'b0000: in1 = data_a;  
        4'b1001: in1 = data_b;  
        4'b1010: in1 = data_c;  
        4'b1100: in1 = data_d;  
  
    endcase  
...
```



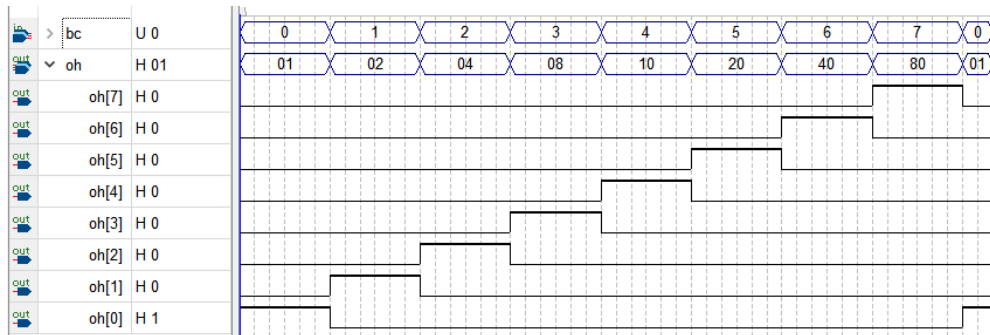
```
always @ *  
    case (state)  
        4'b0000: in1 = data_a;  
        4'b1001: in1 = data_b;  
        4'b1010: in1 = data_c;  
        4'b1100: in1 = data_d;  
        default: in1 = 32'b0;  
    endcase
```

```
always @ *  
begin  
    in1 = 32'b0;  
    case (state)  
        4'b0000: in1 = data_a;  
        4'b1001: in1 = data_b;  
        4'b1010: in1 = data_c;  
        4'b1100: in1 = data_d;  
    endcase  
end
```

Использование массива вместо case (оператор case)

❑ Преобразователь **binary code** в 1 из N

✓ Пример: 3-бит binary в 8 бит 1 из N



Case statement

```
module ex_1 (  
  input      [2:0] bc,  
  output reg [7:0] oh);
```

```
always @*
```

```
case (bc)
```

```
  3'd0 : oh = 8'h01;
```

```
  3'd1 : oh = 8'h02;
```

```
  3'd2 : oh = 8'h04;
```

```
  3'd3 : oh = 8'h08;
```

```
  3'd4 : oh = 8'h10;
```

```
  3'd5 : oh = 8'h20;
```

```
  3'd6 : oh = 8'h40;
```

```
  3'd7 : oh = 8'h80;
```

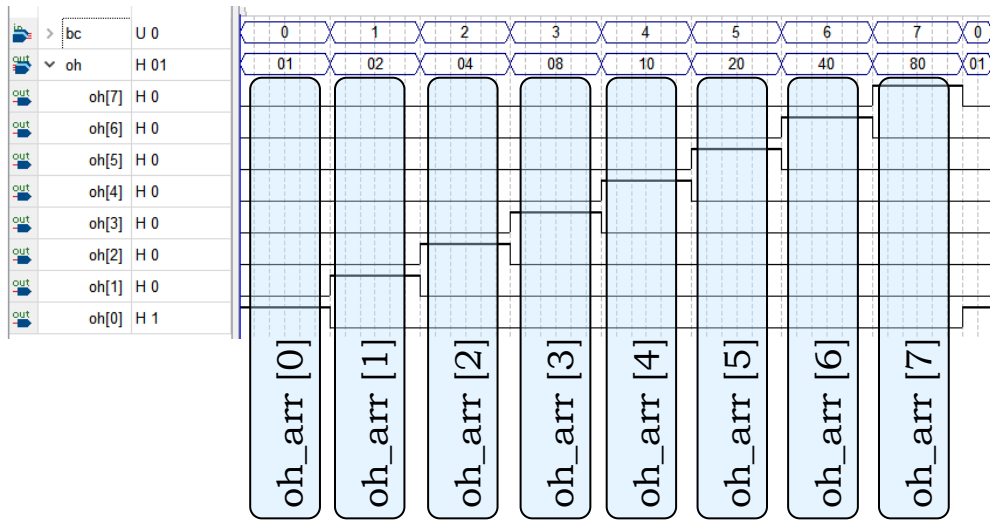
```
endcase
```

```
endmodule
```

Использование массива вместо case (массив)

❑ Преобразователь **binary code** в 1 из N

✓ Пример: 3-бит binary в 8 бит 1 из N



initialized array

```
module ex_1 (  
  input      [2:0] bc,  
  output reg [7:0] oh);  
  reg [7:0] oh_arr [0:7];
```

initial begin

```
    oh_arr[0] = 8'h01;  
    oh_arr[1] = 8'h02;  
    oh_arr[2] = 8'h04;  
    oh_arr[3] = 8'h08;  
    oh_arr[4] = 8'h10;  
    oh_arr[5] = 8'h20;  
    oh_arr[6] = 8'h40;  
    oh_arr[7] = 8'h80;
```

end

always @*

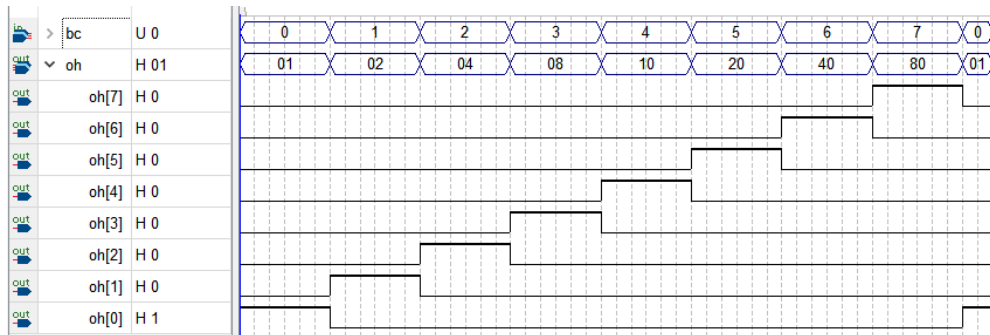
```
    oh = oh_arr[bc];
```

endmodule

Предложите более простой способ описания?

❑ Преобразователь **binary code** в 1 из N

✓ Пример: 3-бит binary в 8 бит 1 из N

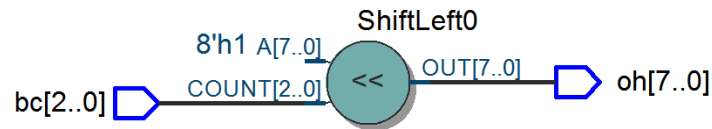


```
module ex_1 (  
  input      [2:0] bc,  
  output reg [7:0] oh);
```

```
  always @*
```

```
    oh = 1'b1 << bc;
```

```
endmodule
```

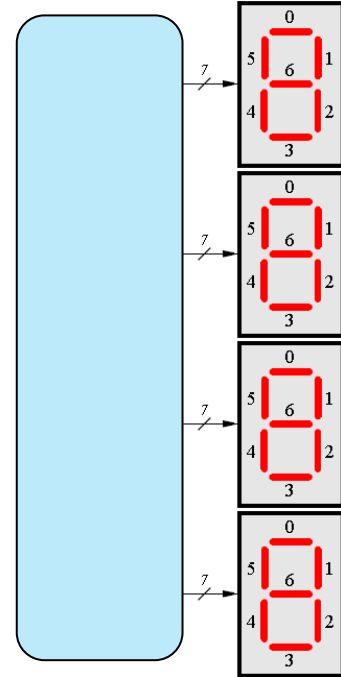


Использование массива вместо case

❑ Задание – разработать преобразователь **binary** (or BCD) в **7-сегментный** код для 4-х разрядов:

✓ Решение:

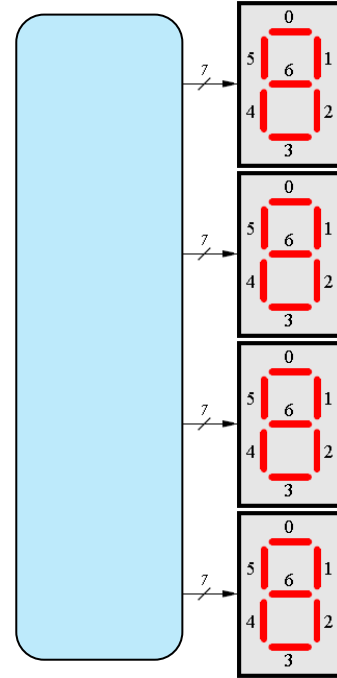
- Разработать модуль для одного разряда => использовать иерархическое (структурное) описание
- Использовать поведенческое описание (отдельный case для каждого разряда)
- **Использование инициализированного массива.**



Использование массива вместо case

```
module ex_1 (  
  input      [3:0] a,b,c, d,  
  output reg [6:0] ss_a,ss_b,  
  output reg [6:0] ss_c,ss_d);  
  reg [6:0] ss_arr[15:0];  
  initial begin  
    ss_arr[0] = 7'h40; //0  
    ss_arr[1] = 7'h79; //1  
    ss_arr[2] = 7'h24; //2  
    ss_arr[3] = 7'h30; //3  
    ss_arr[4] = 7'h19; //4  
    ss_arr[5] = 7'h12; //5  
    ss_arr[6] = 7'h02; //6  
    ss_arr[7] = 7'h78; //7  
    ss_arr[8] = 7'h00; //8  
    ss_arr[9] = 7'h10; //9  
    ss_arr[10] = 7'h08; //A  
    ss_arr[11] = 7'h03; //B  
    ss_arr[12] = 7'h46; //C  
    ss_arr[13] = 7'h21; //D  
    ss_arr[14] = 7'h06; //E  
    ss_arr[15] = 7'h0e; //F end
```

```
always @* begin  
  ss_a = ss_arr[a];  
  ss_b = ss_arr[b];  
  ss_c = ss_arr[c];  
  ss_d = ss_arr[d]; end  
  
endmodule
```



Задержка (Delay) в процедурных назначениях

Задержка (Delay) в процедурных назначениях

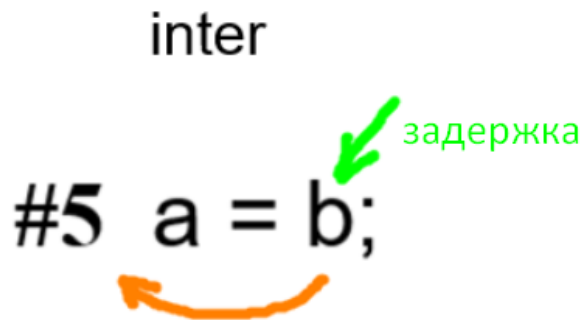
- ❑ Для задания задержки используется конструкция #<value>
- ❑ Варианты задания задержки
 - ✓ **Inter**-assignment Delay Control
 - ✓ **Intra**-assignment Delay Control
 - ✓ **Zero** Delay Control

Inter и Intra-Assignment Delay Control

- ❑ Зеленая стрелка показывает реализацию задержки
- ❑ Оранжевая стрелка показывает то, что выполняется сразу

inter

`#5 a = b;`



intra

`a = #5 b;`



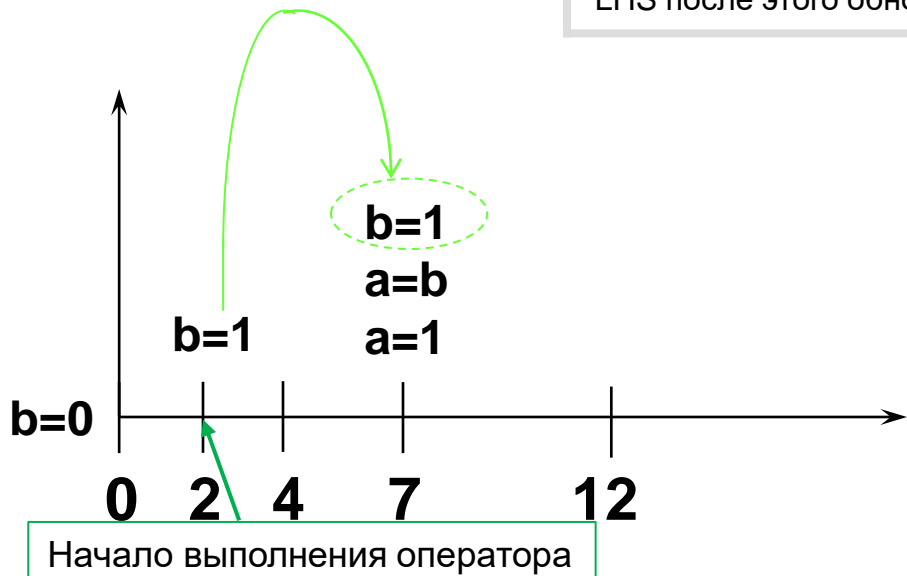
Inter-Assignment Delay Control

- Задерживает чтение|фиксацию (RHS), затем сразу запись (LHS)

#5 $a = b;$

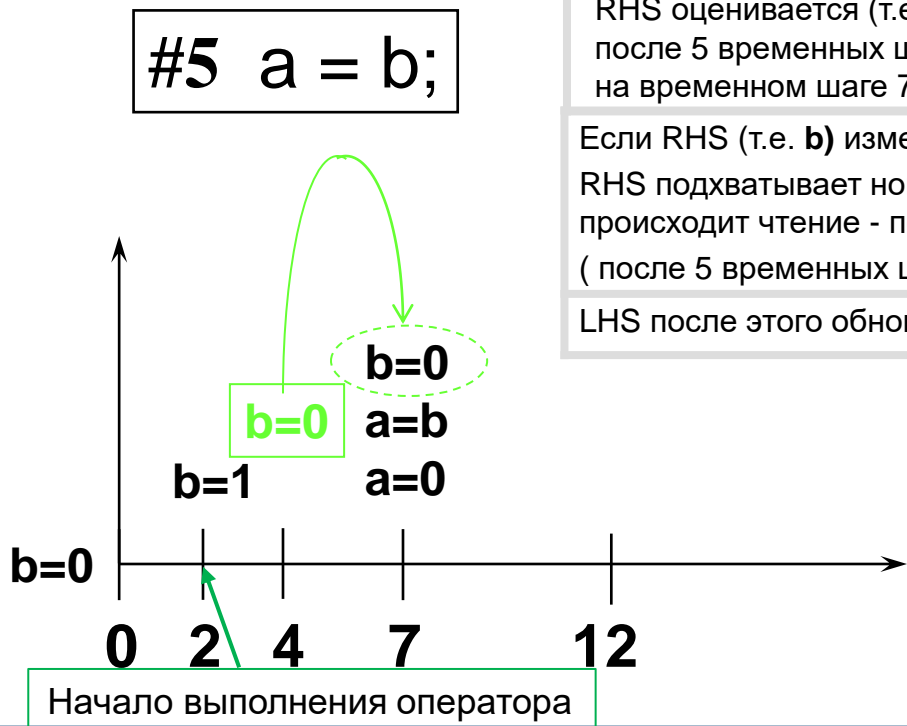
RHS оценивается (т.е. **b** читается) **после** окончания задержки,
(после 5 временных шагов от начала выполнения оператора,
т.е. на временном шаге 7)

LHS после этого обновляется сразу



Inter-Assignment Delay Control (продолжение)

- Задерживает чтение (RHS)



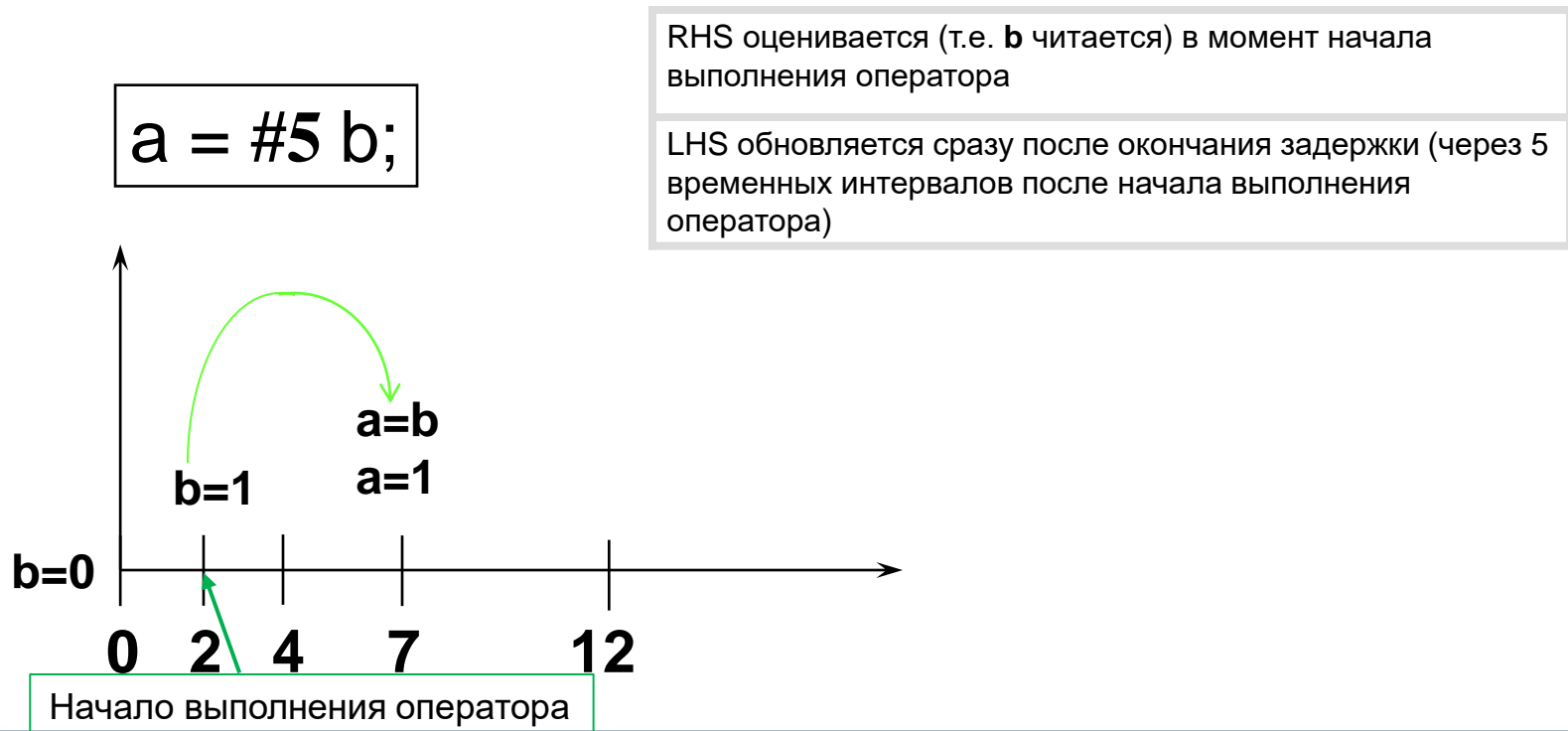
RHS оценивается (т.е. **b** читается) **после** окончания задержки, (после 5 временных шагов от начала выполнения оператора, т.е. на временном шаге 7)

Если RHS (т.е. **b**) изменяется до окончания задержки, => RHS подхватывает новое значение входа в момент когда происходит чтение - после окончания задержки – (после 5 временных шагов от начала выполнения)

LHS после этого обновляется сразу

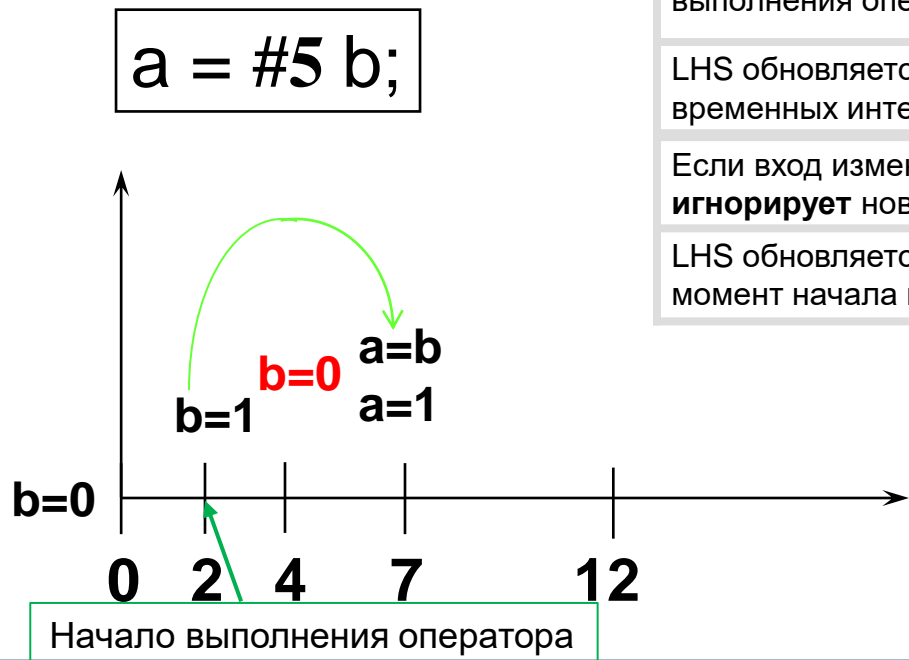
Intra-Assignment Delay Control

- Чтение|фиксация RHS сразу, задерживается запись LHS



Intra-Assignment Delay Control

- Чтение/фиксация RHS сразу, задерживается запись LHS



RHS оценивается (т.е. **b** читается) в момент начала выполнения оператора

LHS обновляется сразу после окончания задержки (через 5 временных интервалов)

Если вход изменяется до окончания задержки, RHS **игнорирует** новое значение

LHS обновляется тем значением RHS, которое было в момент начала выполнения оператора

Пример

```
reg [7:0] a, b;  
always  
    #10 a = b + 2;  
    b = a * 3;  
end
```

begin

// Inter-Assignment delay

1. **Wait 10;**
2. Read value of b and add 2 to it;
3. Assign the result to a;
4. Read the value of a and multiply by three;
5. Assign the result to b;

Пример

```
reg [7:0] a, b;  
always  
    a = #10 b + 2;  
    b = a * 3;  
end
```

begin

//Intra-Assignment delay

1. Read value of b and add 2 to it;
2. **Wait 10;**
3. Assign the result to a;
4. Read the value of a and multiply by three;
5. Assign the result to b;

Zero Delay Control

```
initial begin
```

```
  a = 0;
```

```
  b = 0;
```

```
end
```

```
initial begin
```

```
  #0 a = 1;
```

```
  #0 b = 1;
```

```
end
```

Все четыре оператора будут выполнены в момент времени 0

Так как используется конструкция #0, то операторы a = 1 и b = 1 будут выполнены последними.

- Обеспечивает способ управления порядком выполнения операторов в момент времени 0
- Не рекомендуется назначать различные значения одной переменной одновременно или в разных процессах

Sequential ⇔ Parallel блоки

Parallel блок

initial fork

#10 a = 1;

#15 b = 1;

#25 e = 1;

join

Время	Назначение
10	a = 1'b1;
15	b = 1'b1;
25	e = 1'b1;

Sequential блок

initial begin

#10 a = 1;

#15 b = 1;

#25 e = 1;

end

Время	Назначение
10	a = 1'b1;
25	b = 1'b1;
50	c = 1'b1;

Sequential ⇔ Parallel блоки

- Sequential и Parallel блоки могут быть вложенными

```
initial fork  
    #10 a = 1;  
    #15 b = 1;  
    begin  
        #20 c = 1;  
        #10 d = 1;  
    end  
    #25 e = 1;  
join
```

Time	Statement(s) Executed
10	a = 1'b1;
15	b = 1'b1;
20	c = 1'b1;
25	e = 1'b1;
30	d = 1'b1;

Типы процедурных назначений (блокирующие и неблокирующие)

Типы процедурных назначений

- ❑ Определено два типа процедурных назначений

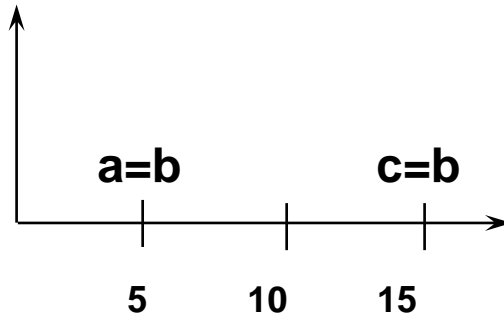
- ❑ Blocking (=) : Обновляет LHS назначение процедурного назначения **блокируя** другие выполнение других назначений процедурного блока
 - ✓ Значение RHS (inputs) защелкивается в момент выполнения оператора
 - ✓ Значение LHS (outputs) обновляется немедленно или после явно заданной задержки
 - ✓ Выполнение следующих операторов задерживается до полного выполнения blocking statement (до обновления значения LHS (включая задержку))

- ❑ Nonblocking (<=) : Обновляет LHS назначение процедурного назначения **не блокируя** другие выполнение других назначений процедурного блока
 - ✓ Значение RHS (inputs) защелкивается в момент выполнения оператора
 - ✓ Обновление значения LHS (outputs) планируется в конце временного шага или после явно заданной задержки
 - ✓ Выполнение следующих операторов не задерживается до полного выполнения nonblocking statement (до обновления значения LHS (включая задержку))

Blocking и Nonblocking Assignments

Blocking (=)

```
initial begin
  a = #5 b;
  c = #10 a;
end
```



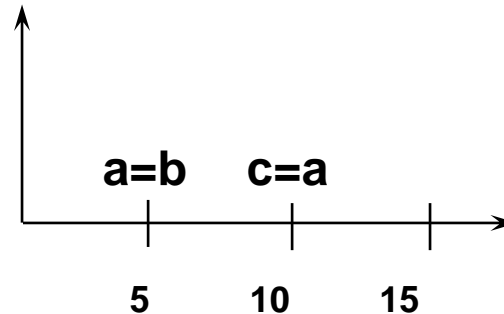
intra

$a = \#5 b;$

задержка

Nonblocking (<=)

```
initial begin
  a <= #5 b;
  c <= #10 a;
end
```



Выполнение Blocking & Nonblocking

```
initial begin  
  a = 1'b0; //Assgnmnt0  
  b = 1'b1; //Assgnmnt1  
  a <= #5 b; //Assgnmnt2  
  b <= #5 a; //Assgnmnt3  
end
```

*Это иллюстрация поведения
blocking and nonblocking
назначений*

1. Назначение 0 выполняется и **a** присваивается значение 0 (остальные назначения блокируются)
2. Назначение 1 выполняется и **b** присваивается значение 1 (остальные назначения блокируются)
3. Назначение 2 выполняется и планируется присвоить **a** текущее значение **b** (1) через 5 временных шагов
4. Назначение 3 выполняется и планируется **b** присвоить текущее значение **a** (0) через 5 временных шагов
5. Процесс заканчивается через 5 временных шагов и значение **a** обновляется до 1, а значение **b** обновляется до 0

Выполнение Blocking & Nonblocking

```
initial begin  
  a = 1'b0; //Assgnmnt0  
  b = 1'b1; //Assgnmnt1  
  a = #5 b; //Assgnmnt2  
  b = #5 a; //Assgnmnt3  
end
```

*Это иллюстрация поведения
blocking and nonblocking
назначений*

1. Назначение 0 выполняется и **a** присваивается значение 0 (остальные назначения блокируются)
2. Назначение 1 выполняется и **b** присваивается значение 1 (остальные назначения блокируются)
3. Назначение 2 выполняется и **a** присваивается текущее значение **b** (1) через 5 временных шагов (остальные назначения на это время блокируются)
4. Назначение 3 выполняется и **b** присваивается текущее значение **a** (1) через 5 временных шагов (остальные назначения на это время блокируются)
5. Процесс закончен

Вопрос: чему равны a и b?

Blocking

```
reg [7:0] a, b;
```

```
initial b = 0;
```

```
initial a = 4;
```

```
always @(a or b)
```

```
begin
```

```
    a = b + 2;
```

```
    b = a * 3;
```

```
end
```

Non-Blocking

```
reg [7:0] a, b;
```

```
initial b = 0;
```

```
initial a = 4;
```

```
always @(a or b)
```

```
begin
```

```
    a <= b + 2;
```

```
    b <= a * 3;
```

```
end
```


Ответ: чему равны a и b?

□ Текст сг Blocking

```
reg [7:0] a, b;
```

```
initial b = 0;
```

```
initial a = 4;
```

```
always @(a or b)
```

```
begin
```

```
    a = b + 2;
```

```
    b = a * 3;
```

```
end
```

$a = 0 + 2 = 2$

$b = 2 * 3 = 6$

Non-Blocking

```
reg [7:0] a, b;
```

```
initial b = 0;
```

```
initial a = 4;
```

```
always @(a or b)
```

```
begin
```

```
    a <= b + 2;
```

```
    b <= a * 3;
```

```
end
```

$a = 0 + 2 = 2$

$b = 4 * 3 = 12$

Continuous; Blocking; Non-Blocking assignments

	Continuous Assignment	Proc. Blocking Assignment	Proc. Non-Blocking Assignment
Operation	=	=	<=
Where	using stand-alone <i>assign</i> statement	inside of <i>always</i> and <i>initial</i>	inside of <i>always</i> and <i>initial</i>
Example	<pre>wire q; reg a, b; assign q = a & b;</pre>	<pre>wire q; reg a, b; always @(b) a = b + 5;</pre>	<pre>wire q; reg a, b; always @(b) a <= 5;</pre>
Valid LHS	net (wire)	reg	reg
Valid RHS	expression of net or reg	expression of net or reg	expression of net or reg
Evaluated	when any part of RHS changes	procedural execution	at the end of current time step