

БЛОКИРУЮЩИЕ И НЕБЛОКИРУЮЩИЕ ПРИСВАИВАНИЯ

Лекционное занятие



Блокирующие и неблокирующие присваивания

Для **блокирующего присвоения** используется символ **=**

Для **неблокирующего назначениями**, используют символ **<=**

Блокирующее присваивание обычно приводит к реализации комбинационных логических схем после того, как синтезировали код. Напротив, **неблокирующее присваивание** обычно приводит к последовательным схемам после синтеза.

Когда происходит назначение сигналов с помощью **блокирующего назначения**, значение сигнала обновляется, как только выполняется строка кода.

Напротив, сигналы, использующие **неблокирующий метод**, не обновляются сразу после назначения. Вместо этого используется планирование назначения для обновления (по фронту сигнала).

Блокирующие и неблокирующие присваивания

```
always_ff @(posedge clk)
begin
    n1 <= d; // неблокирующее
    q <= n1; // неблокирующее
end
```

1. Использование `always_ff`
`@(posedge clk)` и неблокирующего
присваивания для моделирования
последовательной логики.

```
assign y = s ? d1 : d0;
```

2. Использование непрерывного
присваивания для моделирования
простой комбинационной логики.

Блокирующие и неблокирующие присваивания

```
always_comb  
begin  
    p = a ^ b; // блокирующее  
    g = a & b; // блокирующее  
    s = p ^ cin;  
    cout = g | (p & cin);  
end
```

3. Использование `always_comb` и блокирующего присваивания для моделирования более сложной комбинационной логики.

4. Не рекомендуется присваивать значение одному и тому же сигналу в разных операторах `always` или непрерывных присваиваниях.

КОМБИНАЦИОННАЯ ЛОГИКА

Лекционное занятие



```
module fulladder(  
    input  logic  a, b, cin,  
    output logic  s, cout);  
    logic  p, g;  
    always_comb begin  
        p = a ^ b;           // блокирование  
        g = a & b;           // блокирование  
        s = p ^ cin;         // блокирование  
        cout = g |(p & cin); // блокирование  
    end  
endmodule
```

ПОЛНЫЙ СУММАТОР с помощью
always/process

p и **g** - промежуточные
сигналы для вычисления **s** и
cout.

Внутри одного оператора
блокирующие присваивания
выполняются в порядке их
записи.

Здесь эквивалентом `always_comb` было бы `always @(a, b, cin)`,
но `always_comb` лучше, поскольку позволяет избежать ошибок, связанных
с недостающими в списке чувствительности сигналами.

ОПЕРАТОР CASE

Лекционное занятие



Оператор `case`

Оператор `case` используется для выбора блока кода для выполнения на основе значения выбранного сигнала в проекте.

Когда прописывается оператор `case`, указывают входной сигнал для мониторинга и оценки. Значение этого сигнала сравнивается со значениями, указанными в каждой ветви инструкции `case`.

Как только будет найдено соответствие значению входного сигнала, будет выполнена ветвь, связанная с этим значением.

Оператор **case**

```
case (<variable>)
```

```
    <value1> : begin
```

```
// Эта ветвь выполняется, когда <переменная> = <значение1>
```

```
    end
```

```
    <value2> : begin
```

```
// Эта ветвь выполняется, когда <переменная> = <значение2>
```

```
    end
```

```
    default : begin
```

```
// Эта ветвь выполняется во всех остальных случаях
```

```
    end
```

```
endcase
```

Приведенный
фрагмент кода
показывает общий
синтаксис
оператора **case**.

```
always_comb begin
```

```
  case (addr)
```

```
    2'b00 : begin
```

```
      q = a;
```

```
    end
```

```
    2'b01 : begin
```

```
      q = b;
```

```
    end
```

```
    3'b10 : begin
```

```
      q = c;
```

```
    end
```

```
    default : begin
```

```
      q = d;
```

```
    end
```

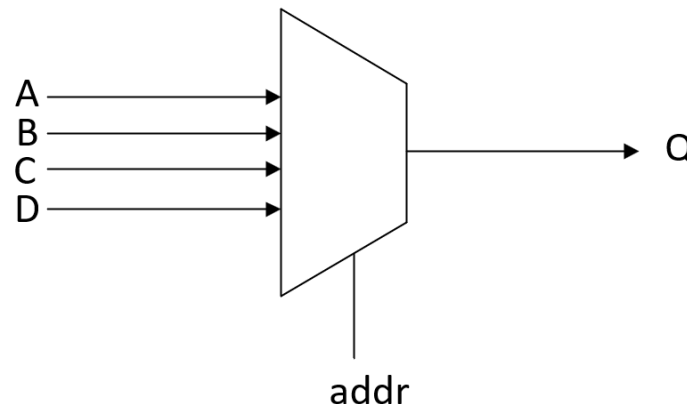
```
  endcase
```

```
end
```

Оператор case

В данном примере используют **блокирующее** присваивание, так как моделируют **комбинационную логику**, и **неблокирующее** присваивание **внутри блока always_comb не рекомендуется**.

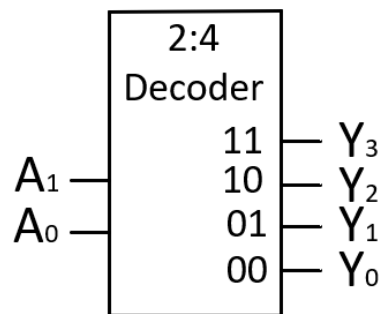
Мультиплексор четыре к одному



Дешифраторы

Дешифраторы – это комбинационное логическое устройство, которое предназначено для преобразование двоичного кода в необходимый сигнал управления в какой-либо системе исчисления на одном из выходов

В общем случае у дешифратора имеется N входов и 2^N выходов. Он выдает **"1"** строго на один из выходов в зависимости от набора входных значений.



Дешифратор 2:4

A ₁	A ₀	Y ₃	Y ₂	Y ₁	Y ₀
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Оператор **case**

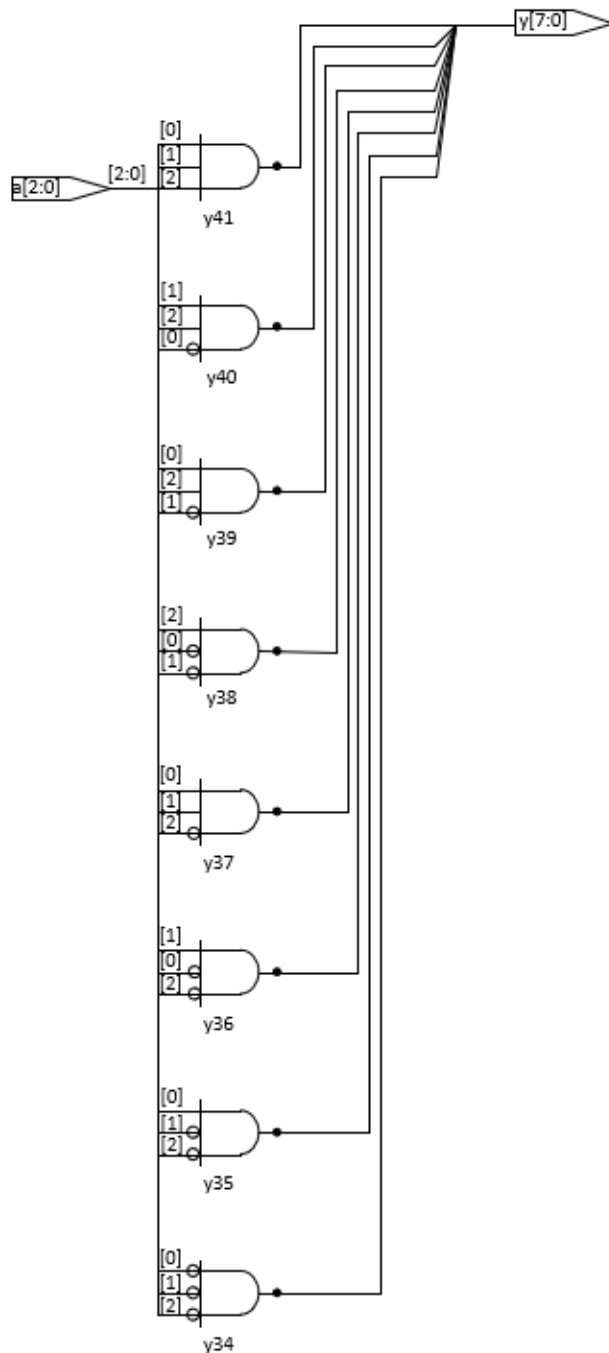
Условие **default** в данном случае не нужно для синтеза, поскольку перечислены все возможные сочетания входов, но оно полезно для симуляции на случай, если какой-либо из входов равен **x** или **z**.

```
module decoder3_8(  
    input    logic [2:0]    a,  
    output   logic [7:0]    y);  
always_comb  
    case(a)  
        3' b000: y = 8' b00000001;  
        3' b001: y = 8' b00000010;  
        3' b010: y = 8' b00000100;  
        3' b011: y = 8' b00001000;  
        3' b100: y = 8' b00010000;  
        3' b101: y = 8' b00100000;  
        3' b110: y = 8' b01000000;  
        3' b111: y = 8' b10000000;  
        default: y = 8' bxxxxxxxx;  
    endcase  
endmodule
```

ДЕШИФРАТОР 3:8

Оператор **case**

Синтезированная схема модуля decoder3_8



Оператор case

Оператор `case` используется для описания дешифратора семисегментного индикатора по таблице истинности. Выполняет различные действия в зависимости от значения его входных данных. Если все возможные сочетания входных данных определены он синтезируется в комбинационную логику, **в противном случае получится последовательная логика**, т.к. выход сохранит свое предшествующее значение в неопределенных случаях.

```
module sevenseg(  
    input    logic [3:0]    data,  
    output   logic [6:0]    segments);  
always_comb  
    case(data)  
        0: segments = 7' b111_1110;  
        1: segments = 7' b011_0000;  
        2: segments = 7' b110_1101;  
        3: segments = 7' b111_1001;  
        4: segments = 7' b011_0011;  
        5: segments = 7' b101_1011;  
        6: segments = 7' b101_1111;  
        7: segments = 7' b111_0000;  
        8: segments = 7' b111_1111;  
        9: segments = 7' b111_0011;  
        default: segments = 7' b000_0000;  
    endcase  
endmodule
```

ДЕШИФРАТОР СЕМИСЕГМЕНТНОГО
ИНДИКАТОРА

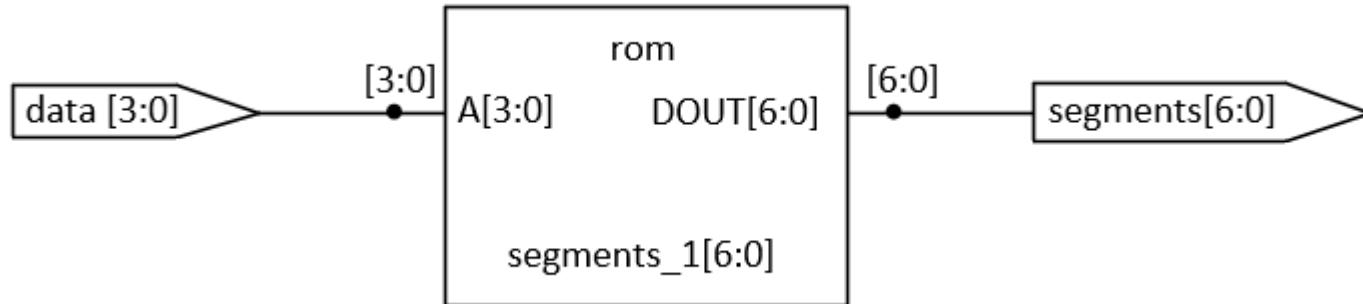
Оператор **case**

case проверяет значение **data**; если **data** равно "0" ,
выполнится действие после
двоеточия, т. е. установка
segments в 1111110.

Аналогично проверяются
другие значения **data** вплоть
до 9.

Условие **default** (по
умолчанию) – удобный
способ определить выход для
всех случаев, не упомянутых
явно.

Оператор **case**



Синтезированная схема модуля `sevensseg`

ОПЕРАТОР IF

Лекционное занятие



Оператор **if**

Оператор **if** - это условный оператор, который использует логические условия для определения, какие блоки кода выполнять.

Всякий раз, когда условие оценивается как **true**, выполняется ветвь кода, связанная с этим условием.

Приведенный фрагмент кода демонстрирует базовый синтаксис оператора **if**.

```
if (<expression1>) begin
    // Код для выполнения
end
else if (<expression2>) begin
    // Код для выполнения
end
else begin
    // Код для выполнения
end
```

Оператор **if** использует логические условия, чтобы определить, какие строки кода выполнять.

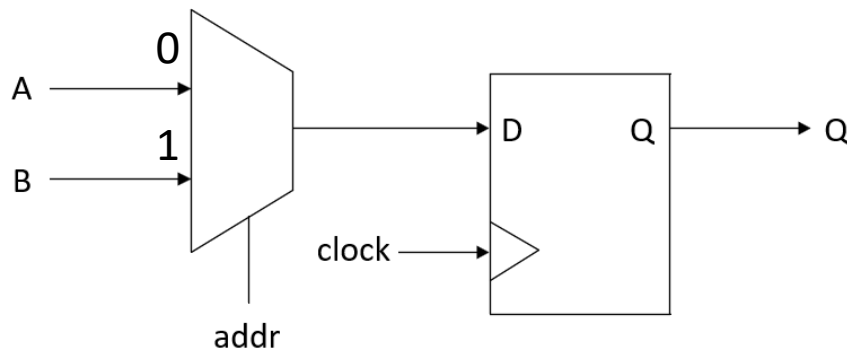
Выражения **<expression1>** и **<expression2>** вычисляются последовательно, и код, связанный с выражением, выполняется, если оно принимает значение **true**.

Оператор `if`

Единственное исключение из этого возникает, когда ни одно из выражений не является истинным. В этом случае будет выполняться код в ветке `else` .

Код, связанный с каждой ветвью, может включать любой допустимый код `SystemVerilog`, включая дополнительные инструкции `if`. Этот подход известен как вложенные инструкции `if`.

В примере будет рассматриваться мультиплексор с тактовой частотой, используя асинхронно сбрасываемый триггер общего типа для регистрации выходных данных мультиплексора.



Оператор **if**

```
always_ff @(posedge clock, posedge reset)
begin
    if (reset) begin
        Q <= 1'b0;
    end
    else begin
        if (addr) begin
            Q <= b;
        end
        else begin
            Q <= a;
        end
    end
end
end
```

Первый оператор **if** используется , чтобы установить выход триггера в 0b, когда активен сброс.

Когда сброс не активен, то передний фронт тактового сигнала запускает блок **always_ff** . Ветвь **else** первого оператора **if** используется, чтобы зафиксировать это условие.

Второй оператор **if** необходим для моделирования поведения схемы мультиплексора.

Оператор **if**

```
always_ff @(posedge clock, posedge reset)
begin
    if (reset) begin
        Q <= 1'b0;
    end
    else begin
        if (addr) begin
            Q <= b;
        end
        else begin
            Q <= a;
        end
    end
end
end
```

Когда сигнал **addr** равен **1b**, происходит присваивание выходу триггера значения входа **b**.

Затем используется ветвь **else** вложенного оператора **if**, чтобы зафиксировать случай, когда сигнал **addr** равен **0b**.

```
module priorityckt(  
    input  logic [3:0]  a,  
    output logic [3:0]  y);  
    always_comb  
        if (a[3])      y <= 4' b1000;  
        else if (a[2]) y <= 4' b0100;  
        else if (a[1]) y <= 4' b0010;  
        else if (a[0]) y <= 4' b0001;  
        else           y <= 4' b0000;  
endmodule
```

СХЕМА ПРИОРИТЕТОВ

Оператор if

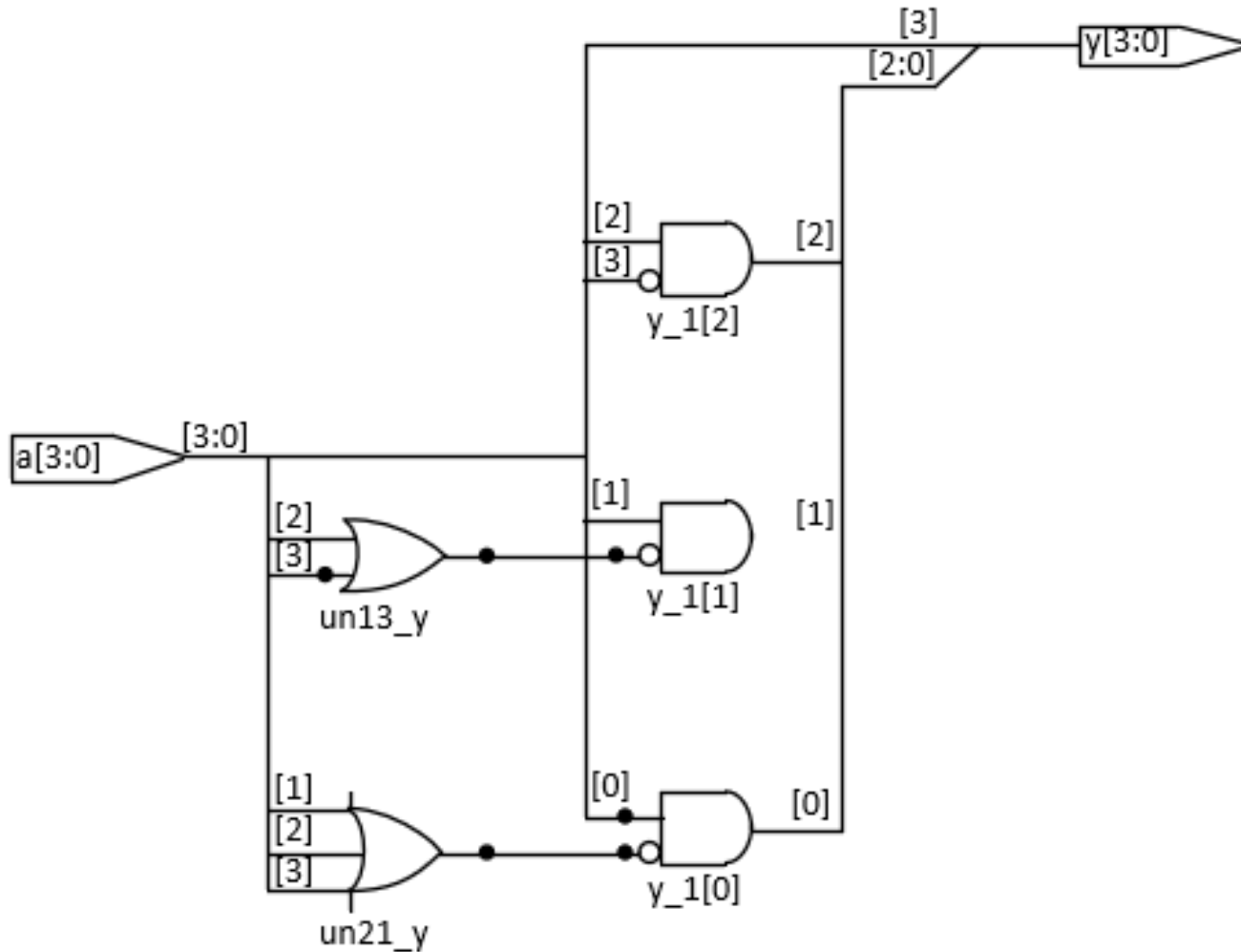
используется для описания схемы приоритетов.

N-входовая схема приоритетов устанавливает в значение **TRUE** тот из выходов, который соответствует наиболее приоритетному входу, равному **TRUE**.

Оператор **if** обязан быть внутри операторов **always**.

Если все возможные сочетания входов обработаны, то оператор синтезируется в комбинационную логику, иначе – в последовательную.

Оператор **if**



Синтезированная схема модуля `priorityckt`

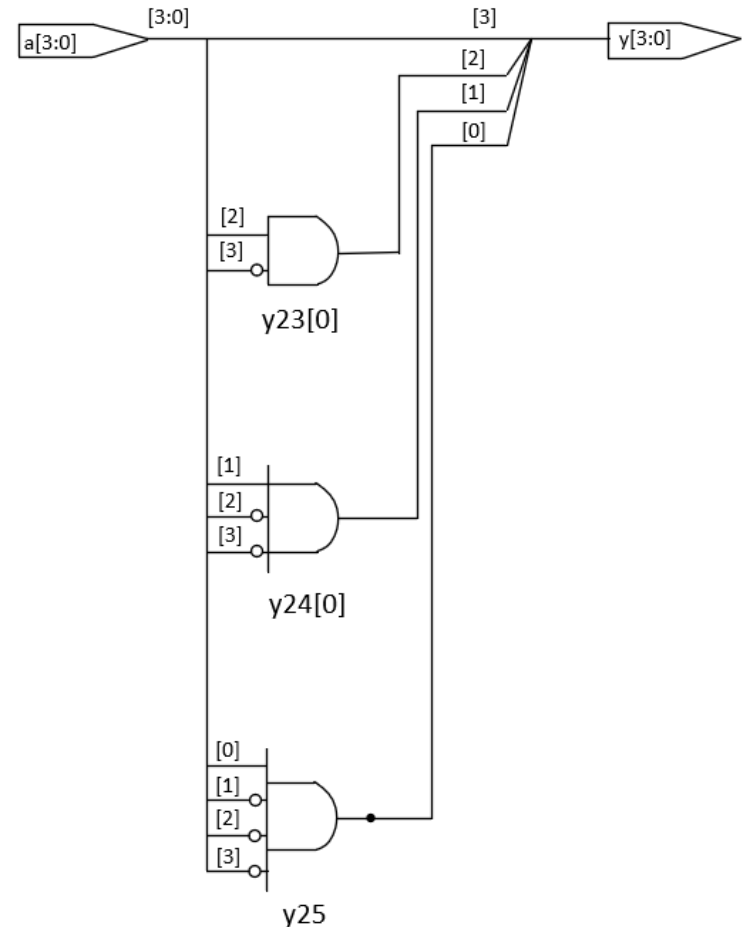
Таблицы истинности с незначащими битами

```

module priority_casez(
    input  logic [3:0]  a,
    output logic [3:0]  y);
always_comb
    casez(a)
        4' b1???: y <= 4' b1000;
        4' b01??: y <= 4' b0100;
        4' b001?: y <= 4' b0010;
        4' b0001: y <= 4' b0001;
        default: y <= 4' b0000;
    endcase
endmodule

```

СХЕМА ПРИОРИТЕТОВ С НЕЗНАЧАЩИМИ БИТАМИ



Синтезированная схема модуля
priority_casez

Оператор `always` используют для описания последовательных схем, потому что он сохраняет состояние переменных, если не было указано их изменить. Так же этот оператор можно использовать для поведенческого описания комбинационной логики.

```
module inv(  
    input  logic [3:0] a,  
    output logic[3:0] y);  
    always_comb  
        y = ~a;  
endmodule
```

ИНВЕРТОР с помощью `always`

Оператор `always_comb` исполняет выражения внутри, каждый раз, когда изменяется любой из сигналов в правой части `<=` или `=` оператора `always`.

Если код внутри оператора `always_comb` не является комбинационной логикой, то компилятор будет выдавать предупреждение.

Комбинационная логика*

```
module fulladder(  
    input  logic  a, b, cin,  
    output logic  s, cout);  
    logic  p, g;  
    always_comb begin  
        p = a ^ b;           // блокирование  
        g = a & b;           // блокирование  
        s = p ^ cin;         // блокирование  
        cout = g |(p & cin); // блокирование  
    end  
endmodule
```

ПОЛНЫЙ СУММАТОР С БЛОКИРУЮЩИМИ
ПРИСВАИВАНИЯМИ

Рассмотрим реализацию с
блокирующими
присвоениями

Комбинационная логика*

Значения **a**, **b** и **cin** изначально равны "0" .

Значения **p**, **g**, **s** и **cout** – "0" .

В какой-то момент **a** становится "1" , активируя оператор **always**.

Четыре блокирующих присваивания в следующем порядке:

1. $p \leftarrow 1 \oplus 0 = 1$

2. $g \leftarrow 1 \cdot 0 = 0$

3. $s \leftarrow 1 \oplus 0 = 1$

4. $cout \leftarrow 0 + 1 \cdot 0 = 0$

p и **g** получают свои новые значения до вычислений **s** и **cout** благодаря блокирующим присваиваниям.

Т.к. **s** и **cout** вычисляются из новых значений **p** и **g**, то они сразу верны.

Комбинационная логика*

```
module fulladder(  
    input    logic    a, b, cin,  
    output   logic    s, cout);  
    logic    p, g;  
    always_comb  
    begin  
        p <= a ^ b; // неблокирующий  
        g <= a & b; // неблокирующий  
        s <= p ^ cin;  
        cout <= g | (p & cin);  
    end  
endmodule;
```

ПОЛНЫЙ СУММАТОР С НЕБЛОКИРУЮЩИМИ
ПРИСВАИВАНИЯМИ

Другой случай, когда **a** из "0" становится "1", в то время как **b** и **cin** - "0". Четыре неблокирующих присваивания выполняются одновременно:

$$p \leftarrow 1 \oplus 0 = 1$$

$$g \leftarrow 1 \cdot 0 = 0$$

$$s \leftarrow 0 \oplus 0 = 0$$

$$\text{cout} \leftarrow 0 + 0 \cdot 0 = 0$$

Комбинационная логика*

```
module fulladder(  
    input    logic    a, b, cin,  
    output   logic    s, cout);  
    logic    p, g;  
    always_comb  
    begin  
        p <= a ^ b; // неблокирующий  
        g <= a & b; // неблокирующий  
        s <= p ^ cin;  
        cout <= g | (p & cin);  
    end  
endmodule;
```

ПОЛНЫЙ СУММАТОР С НЕБЛОКИРУЮЩИМИ
ПРИСВАИВАНИЯМИ

s вычисляется одновременно с p ,
т.о. используется старое значение p .

s остается равным "0" .

p изменяется с "0" на "1" .

Это изменение вызывает
исполнение оператора always во
второй раз:

$$p \leftarrow 1 \oplus 0 = 1$$

$$g \leftarrow 1 \cdot 0 = 0$$

$$s \leftarrow 1 \oplus 0 = 1$$

$$cout \leftarrow 0 + 1 \cdot 0 = 0$$

Комбинационная логика*

```
module fulladder(  
    input    logic    a, b, cin,  
    output   logic    s, cout);  
    logic    p, g;  
    always_comb  
    begin  
        p <= a ^ b; // неблокирующий  
        g <= a & b; // неблокирующий  
        s <= p ^ cin;  
        cout <= g | (p & cin);  
    end  
endmodule;
```

ПОЛНЫЙ СУММАТОР С НЕБЛОКИРУЮЩИМИ
ПРИСВАИВАНИЯМИ

`p` равно "1" , и `s` становится
равным "1" .

Неблокирующие присваивания
тоже приводит к правильному
ответу, но оператору `always`
приходится выполняться дважды.

Данная обработка снижает
скорость выполнения симуляции.

Комбинационная логика*

Недостатком **неблокирующих присваиваний** для моделирования комбинационной логики является следующее.

При симуляции может возникнуть неверный результат, если не обозначить промежуточные переменные в списке чувствительности.

Так же синтезаторы создадут правильную схему, даже если неверный список чувствительности приводит к неверной симуляции. Что приводит к несовпадению результатов симуляции и реального поведения аппаратуры.

Если бы список чувствительности оператора **always** был написан как **always@(a, b, cin)**, а не как **always_comb**, оператор не выполнялся бы повторно, когда изменяется **p** или **g**.

В этом случае **s** ошибочно остался бы равным "0" вместо "1" .

Последовательная логика*

```
module sync(  
    input  logic  clk,  
    input  logic  d,  
    output logic  q);  
    logic  n1;  
    always_ff @(posedge clk)  
    begin  
        n1 <= d; // неблокирующий  
        q <= n1; // неблокирующий  
    end  
endmodule
```

СИНХРОНИЗАТОР

Синхронизатор корректно смоделирован с использованием неблокирующих присваиваний. По переднему фронту тактового сигнала **d** копируется в **n1** в то же время, как **n1** копируется в **q**. Например, первоначальные значения **d** = "0" , **n1** = "1" и **q** = "0" . По переднему фронту тактового сигнала одновременно выполняются два присваивания. После прохождения фронта **n1** = "0" и **q** = "1" :

$$n1 \leftarrow d = "0" \quad q \leftarrow n1 = "1"$$

Последовательная логика*

```
module syncbad(  
    input  logic  clk,  
    input  logic  d,  
    output logic  q);  
    logic  n1;  
    always_ff @(posedge clk)  
    begin  
        n1 = d; // блокирование  
        q = n1; // блокирование  
    end  
endmodule
```

ПЛОХОЙ СИНХРОНИЗАТОР С
БЛОКИРУЮЩИМИ
ПРИСВАИВАНИЯМИ

По переднему фронту `clk`, `d` копируется в `n1`. Новое значение `n1` копируется в `q`, в результате чего значение `d` **ошибочно** оказывается и в `n1`, и в `q`.

Присваивания выполняются одно за другим, после фронта сигнала `q = n1 = "0"` .

1. `n1 ← d = "0"`

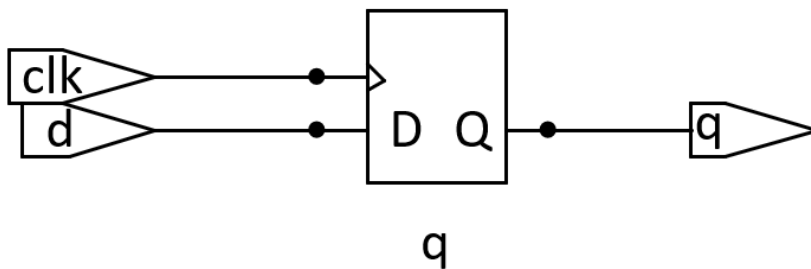
2. `q ← n1 = "0"`

Т.к. переменная `n1` не видна окружающему миру и не влияет на поведение `q`, синтезатор ликвидирует ее в процессе оптимизации.

Последовательная логика*

Для моделирования последовательной логики в операторе `always` рекомендуют пользоваться исключительно неблокирующими присваиваниями.

С помощью разных хитростей, например, изменения порядка присваиваний, можно добиться правильной работы блокирующих присваиваний, но они не дают никаких преимуществ, а лишь приносят риск нежелательного поведения.



Синтезированная схема для syncbad

Литература

1. Дэвид М. Хэррис, Сара Л. Хэррис Цифровая схемотехника и архитектура компьютера. Второе издание. Morgan Kaufman, 2013 г. – 1648 стр.
2. FPGA Tutorial. / [Электронный ресурс] / URL: <https://fpgatutorial.com/>

СПАСИБО ЗА ВНИМАНИЕ

