Введение в Verilog – язык описания цифровых схем.

Николай К.

Специально для проекта http://marsohod.org

Введение

В этой статье – пять уроков Verilog – языка описания цифровых схем. Этот язык используется для проектирования логики микросхем FPGA или CPLD и так же ASIC.

Статья ни в коей мере не претендует на полноту описания языка Verilog. И, конечно, она не может заменить множества книг, которые Вам придется прочитать, если Вы всерьез заинтересуетесь / займетесь проектированием цифровых микросхем. Тем не менее, надеюсь, эти уроки помогут начинающим разработчикам, ведь здесь, в краткой форме, изложены основные понятия и принципы языка.

Причина написания статьи - информации на русском языке о Verilog очень мало. Основные источники информации о языке Verilog – англоязычные.

Читатель должен быть подготовлен. Вы должны иметь хотя бы общее представление о традиционных языках программирования типа C или Паскаль.

Основа для написания этой статьи – уроки Verilog от Tim Miller, взятые с http://opengraphics.org. Тем не менее, эта статья не просто перевод с английского на русский. Я добавил один урок, другие отредактировал, переработал и дополнил, а так же вставил иллюстрации в виде схем и временных диаграмм.

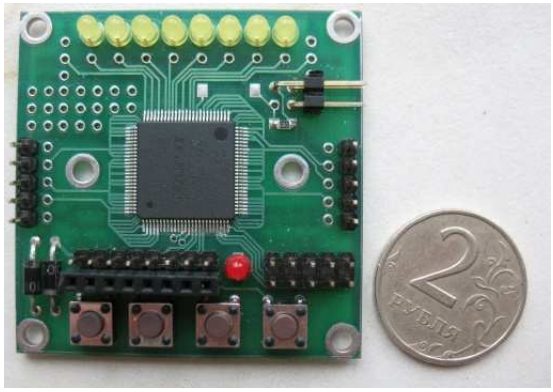
При написании этой статьи так же использовались следующие книги:

1) “The Verilog Hardware Description Language”, Fifth Edition. Donald E. Thomas, Philip R.Moorby.
2) “Verilog HDL. A Guide to Digital Design and Synthesis”, Samir Palnitkar.

Что бы начать использовать полученные знания на практике Вам нужна среда разработки и программирования – например, Altera Quartus II (http://altera.com).

На сайте http://marsohod.org Вы так же найдете пошаговые инструкции для Quartus II: как создать проект, нарисовать цифровую схему или описать ее на Verilog, как откомпилировать проект и зашить его в ПЛИС.

Сайт http://marsohod.org представляет простую макетную плату **Марсоход** с ПЛИС Altera для Ваших экспериментов – и это проект с открытыми исходниками!



На сайте опубликавана схема этой платы, а так же исходники всех проектов, сделанных на ее основе: машинки, роботы, игрушки.

Урок 1. Базовые типы источников сигналов.

Verilog - язык описания цифровых схем. На первом уроке познакомимся с базовыми типами источников сигнала используемыми в языке.

Пожалуй было бы не плохо начать наше обсуждение с понятия сигнал (*signal*). Сигналы – это электрические импульсы, которые передаются по проводам (*wire*) между логическими элементами схемы. Провода переносят информацию не производя над ней никаких вычислений. В цифровой схеме сигналы важны для передачи двоичных данных.

Базовый тип источника сигнала в языке Verilog – это провод, *wire*. Таким образом, если у вас есть арифметическое или логическое выражение, вы можете ассоциировать результат выражения с именованным проводом и позже использовать его в других выражениях. Это немного похоже на переменные, только их (как провода в схеме) нельзя пересоединить на лету, нельзя поменять назначение. Значение провода (*wire*) – это функция того, что присоединено к нему.

Вот пример декларации однобитного провода в “программе” Verilog:

```
wire a;
```

Вы можете ему назначить другой сигнал, скажем сигнал “*b*”, вот так:

```
wire b;
assign a = b;
```

Или вы можете определить сигнал и сделать назначение ему одновременно в одном выражении:

```
wire a = b;
```

У вас могут быть провода передающие несколько бит:

```
wire [3:0] c; //это четыре провода
```

Провода передающие несколько бит информации называются “шина”, иногда “вектор”. Назначания к ним делаются так же:

```
wire [3:0] d;
assign c = d; //“подключение” одной шины к другой
```

Количество проводов в шине определяется любыми двумя целыми числами разделенными двоеточием внутри квадратных скобок.

```
wire [11:4] e; //восьмибитная шина
wire [0:255] f; //256-ти битная шина
```

Из шины можно выбрать некоторые нужные биты и назначить другому проводу:

```
wire g;
assign g = f[2];  //назначить сигналу "g" второй бит шины "f"
```

Кроме того, выбираемый из шины бит может определяться переменной:

```
wire [7:0] h;
wire i = f[h];  // назначить сигналу "i" бит номер "h" из шины "f"
```

Вы можете выбрать из сигнальной шины некоторый диапазон бит и назначить другой шине с тем же количеством бит:

```
wire [3:0] j = e[7:4];
```

Так же, в большинстве диалектов Verilog, вы можете определить массивы сигнальных шин:

```
wire [7:0] k [0:19];  //массив из двадцати 8-ми битных шин
```

Еще существует другой тип источника сигнала называемый регистр: *reg*. Его используют при поведенческом (*behavioral*) описании схемы. Если регистру постоянно присваивается значение комбинаторной (логической) функции, то он ведет себя точно как провод (*wire*). Если же регистру присваивается значение в синхронной логике, например по фронту сигнала тактовой частоты, то ему, в конечном счете, будет соответствовать физический D-триггер или группа D-триггеров. D-триггер – это логический элемент способный запоминать один бит информации. В англоязычных статьях D-триггер называют *flipflop*.

Регистры описываются так же как и провода:

```
reg [3:0] m;
reg [0:100] n;
```

Они могут использоваться так же, как и провода в правой части выражений, как операнды:

```
wire [1:0] p = m[2:1];
```

Вы можете определить массив регистров, которые обычно называют "память" (*RAM*):

```
reg [7:0] q [0:15];  //память из 16 слов, каждое по 8 бит
```

Еще один тип источника сигнала – это *integer*. Он похож на регистр *reg*, но всегда является 32х битным знаковым типом данных. Например, объявим:

```
integer loop_count;
```

Verilog позволяет группировать логику в блоки. Каждый блок логики называется “модулем” (*module*). Модули имеют входы и выходы, которые ведут себя как сигналы *wire*.

При описании модуля сперва перечисляют его порты (входы и выходы):

```
module my_module_name (port_a, port_b, w, y, z);
```

А затем описывают направление сигналов:

```
input port_a;
output [6:0] port_b;
input [0:4] w;
inout y; //двунаправленный сигнал, обычно используется

         //только для внешних контактов микросхем
```

Позже мы увидим, что выход модуля может быть сразу декларирован как регистр *reg*, а не как провод *wire*:

```
output [3:0] z;
reg [3:0] z;
```

Еще проще можно сразу в описании модуля указать тип и направление сигналов:

```
module my_module
(
    input wire port_a,
    output wire [6:0]port_b,
    input wire [0:4]w,
    inout wire y,
    output reg [3:0]z
);
```

Теперь можно использовать входные сигналы, как провода *wire*:

```
wire r = w[1];
```

Теперь можно делать постоянные назначения выходам, как функции от входов:

```
assign port_b = h[6:0];
```

В конце описания логики каждого модуля пишем слово *endmodule*.

```
module my_module_name (input wire a, input wire b, output wire c);
assign c = a & b;
endmodule
```

Последний тип источника сигнала, о котором мы поговорим на этом уроке – это постоянные сигналы или просто числа:

```
wire [12:0] s = 12; /* 32-х битное десятичное число, которое будет
“обрезано” до 13 бит */
wire [12:0] z = 13’d12; //13-ти битное десятичное число
wire [3:0] t  = 4'b0101; //4-х битное двоичное число
wire [3:0] q  = 8'hA5;   // 8-ми битное шестнадцатеричное число A5
wire [63:0] u = 64'hdeadbeefcafebabe;  /*64-х битное
шестнадцатеричное число */
```

Если точно не определить размер числа, то оно принимается по умолчанию 32-х разрядным. Это может быть проблемой при присвоении сигналам с большей или меньшей разрядностью.

Числа – это числа. Они могут использоваться во всяких арифметических и логических выражениях. Например, можно прибавить 1 к вектору “*aa*”:

```
wire [3:0] aa;
wire [3:0] bb;
assign bb = aa + 1;
```

Понятно, что такое выражение сложения в коде на Verilog в конечном счете обернется аппаратным сумматором внутри чипа.

Урок 2. Иерархия проекта.

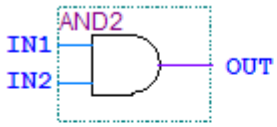
Мы уже знаем, что такое модуль.

В проекте, особенно сложном, бывает много модулей, соединенных между собой. Прежде всего, нужно заметить, что в проекте всегда есть один модуль самого верхнего уровня (*top level*). Он состоит из нескольких других модулей. Те в свою очередь могут содержать еще модули и так далее. Не обязательно, чтобы все модули были написаны на одном языке описания аппаратуры. Совсем наоборот. Довольно удобно и наглядно иметь модуль самого верхнего уровня выполненным в виде схемы, состоящей из модулей более низкого уровня. Эти модули могут быть написаны разными людьми, на разных языках (Verilog, VHDL, AHDL, и даже выполнены в виде схемы). На самом деле – это все дело вкуса и возможностей компилятора (синтезатора), а так же требований заказчика.

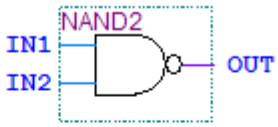
Поскольку у нас уроки Verilog, то будем все же рассматривать проект с точки зрения программирования на этом языке.

Итак, внутри тела любого модуля, можно объявлять экземпляры других модулей и потом соединять их друг с другом проводами.

Предположим у нас есть следующие примитивные модули:



AND2	IN1 = 0	1	x	z
IN2 = 0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x



NAND2	IN1 = 0	1	x	z
IN2 = 0	1	1	1	1
1	1	0	x	x
x	1	x	x	x
z	1	x	x	x

Вы видите графическое представление самых простых логических элементов и ниже их таблица истинности – значение логической функции на выходе при заданных значениях на входах. Слева изображен двухвходовый логический элемент **И**. На его выходе единица, если на первом **И** на втором входе единица. Справа изображен двухвходовый логический элемент **И-НЕ**. На его выходе ноль, если на первом **И** втором входах единицы.

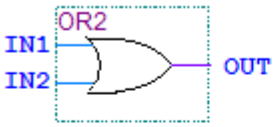
Обратите внимание, что на входе может быть не только логический ноль или единица, но и так же неопределенное значение **x**, или вход может быть подключен к двунаправленному сигналу (*inout*), находящемуся в высокоомном состоянии **z**. Эти значения так же указаны в таблице истинности для полноты картины.

Эти логические модули можно было бы описать на языке Verilog следующим образом:

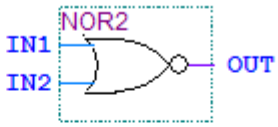
```
module AND2(output OUT, input IN1, input IN2);
assign OUT = IN1 & IN2;
endmodule

module NAND2(output OUT, input IN1, input IN2);
assign OUT = ~(IN1 & IN2);
endmodule
```

Однако, вот такие описания делать не нужно. Это базовые элементы (*gates*) и они наверняка должны быть в стандартных библиотеках синтезатора.

Вот еще пара важных логических элементов:



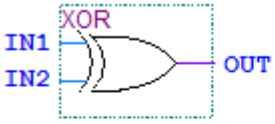
OR2	IN1 = 0	IN1 = 1	IN1 = x	IN1 = z
IN2 = 0	0	1	x	x
IN2 = 1	1	1	1	1
IN2 = x	x	1	x	x
IN2 = z	x	1	x	x



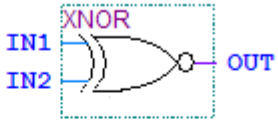
NOR2	IN1 = 0	IN1 = 1	IN1 = x	IN1 = z
IN2 = 0	1	0	x	x
IN2 = 1	0	0	0	0
IN2 = x	x	0	x	x
IN2 = z	x	0	x	x

Слева логический элемент **ИЛИ**. На выходе единица, если на первом **ИЛИ** втором входе единица. Справа – логический элемент **ИЛИ-НЕ**. На выходе ноль, если на первом **ИЛИ** втором входе – единица.

Следующие важные примитивы связаны с «отрицанием равнозначности»:



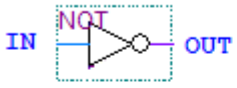
XOR	IN1 = 0	IN1 = 1	IN1 = x	IN1 = z
IN2 = 0	0	1	x	x
IN2 = 1	1	0	x	x
IN2 = x	x	x	x	x
IN2 = z	x	x	x	x



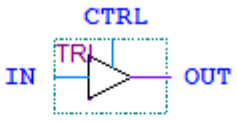
XNOR	IN1 = 0	IN1 = 1	IN1 = x	IN1 = z
IN2 = 0	1	0	x	x
IN2 = 1	0	1	x	x
IN2 = x	x	x	x	x
IN2 = z	x	x	x	x

Элемент слева (**XOR**) имеет на выходе ноль если на обоих входах одинаковое значение (либо оба входа ноль, либо оба входа единица). Элемент справа (**XNOR**) - тоже самое, только с инверсией. На выходе единица, если либо оба входа ноль, либо оба входа единица.

Напоследок еще пара очень важных базовых элемента:



NOT	OUT
IN = 0	1
IN = 1	0
IN = x	x
IN = z	x

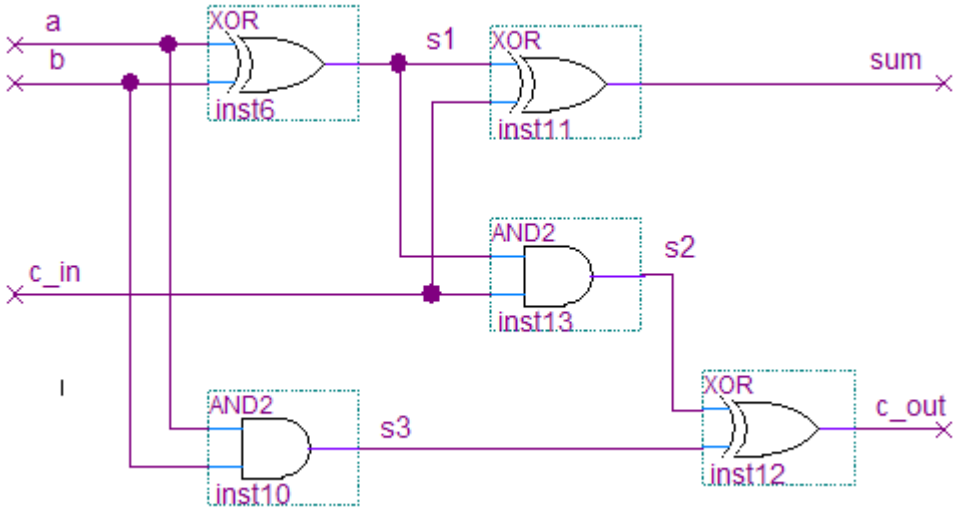


TRI	CTRL = 0	CTRL = 1	CTRL = x	CTRL = z
IN = 0	z	0	L	L
IN = 1	z	1	H	H
IN = x	z	x	x	x
IN = z	z	x	x	x

Слева элемент **НЕ**. На его выходе значение противоположное входному значению. Если на входе единица, то на выходе ноль. И на оборот.

Справа - буфферный элемент, использующийся для двунаправленных сигналов (*inout*). Этот элемент «пропускает» через себя входной сигнал, только если на входе CTRL есть управляющая единица. Если на входе CTRL ноль, то элемент «отключается» от выходного провода переходя в высокоомное состояние. Такие элементы вообще-то используются только для выводов цифровых микросхем. Иными словами, использовать двунаправленные сигналы (*inout*) правильнее всего только в модуле самого верхнего уровня.

Итак, мы знаем про основные базовые логические элементы – и это тоже модули. Используем их в модуле более высокого уровня. Сделаем однобитный сумматор по вот такой схеме:



Этот сумматор складывает два однобитных числа *a* и *b*. При выполнении сложения однобитных чисел может случиться «переполнение», то есть результат уже будет двухбитным (1+1=2 или в двоичном виде 1'b1+1'b1=2'b10). Поэтому у нас есть выходной сигнал переноса *c_out*. Дополнительный входной сигнал *c_in* служит для приема сигнала переноса от сумматоров младших разрядов, при построении многобитных сумматоров.

Посмотрите, как можно описать эту схему на языке Verilog, устанавливая в теле модуля экземпляры других модулей. Это описание на уровне элементов (*gate-level modelling*). Мы установим в наш модуль 3 экземпляра модуля XOR и два экземпляра модуля AND2.

```
module adder1(output sum, output c_out, input a, input b, input
c_in);
wire s1,s2,s3;

XOR  my_1_xor(  .OUT (s1),  .IN1 (a), .IN2 (b)  );
AND2 my_1_and2( .OUT (s3), .IN1 (a), .IN2 (b) );

XOR my_2_xor(  .OUT (sum),  .IN1 (s1), .IN2 (c_in)  );
AND2 my_2_and2( .OUT (s2), .IN1 (s1), .IN2 (c_in) );

XOR my_3_xor( .OUT (c_out),  .IN1 (s2), .IN2 (s3) );

endmodule
```

Порядок описания экземпляра модуля такой:

- Пишем название модуля, экземпляр которого нам нужен.
- Пишем название конкретно этого экземпляра модуля (по желанию).
- Описываем подключения сигналов: точка и затем имя сигнала модуля, затем в скобках имя провода, который сюда подключен.

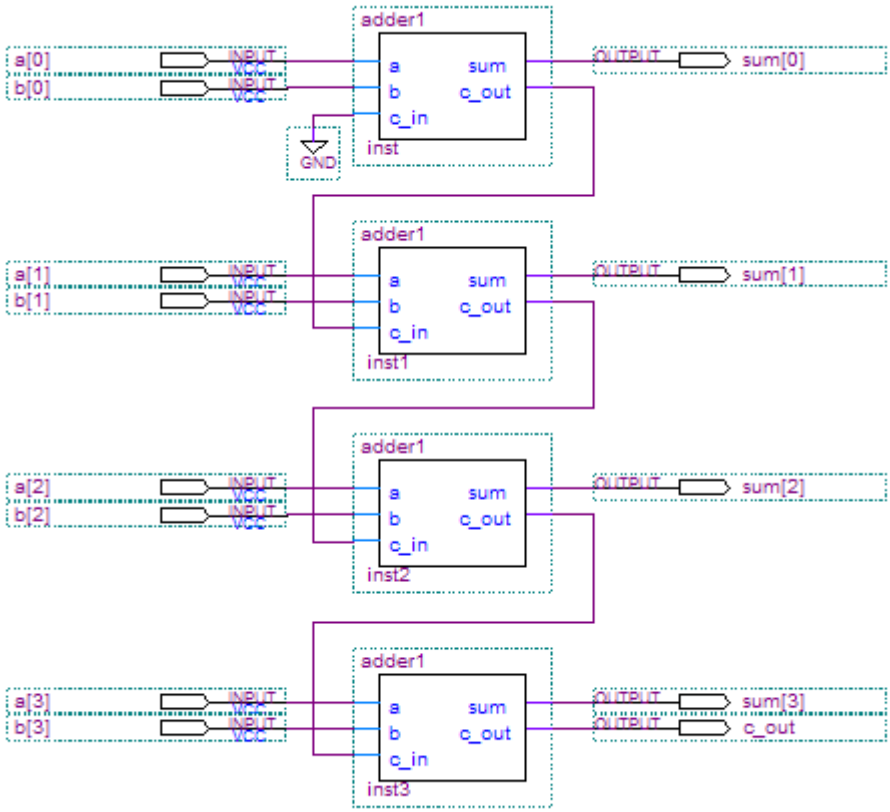
Конечно, совсем не обязательно описывать наш модуль так, как описано выше. Честно говоря, мне гораздо приятней видеть вот такой код однобитного сумматора:

```
module adder1(output sum, output c_out, input a, input b, input
c_in);
assign sum = (a^b) ^ c_in;
assign c_out = ((a^b) & c_in) ^ (a&b);
endmodule
```

Просто важно понимать, что существуют разные методы описания, и нужно уметь ими всеми пользоваться.

Теперь у нас есть однобитный сумматор и мы можем сделать, например, четырехбитный (с последовательным переносом)!

Вот так:



На Verilog это же будет выглядеть следующим образом:

```
module adder4(output [3:0]sum, output c_out, input [3:0]a, input
[3:0]b );
wire c0, c1, c2;
adder1 my0_adder1( .sum (sum[0]) , .c_out (c0), .a (a[0]), .b
(b[0]), .c_in (1'b0) );
adder1 my1_adder1( .sum (sum[1]) , .c_out (c1), .a (a[1]), .b
(b[1]), .c_in (c0));
adder1 my2_adder1( .sum (sum[2]) , .c_out (c2), .a (a[2]), .b
(b[2]), .c_in (c1));
adder1 my3_adder1( .sum (sum[3]) , .c_out (c_out), .a (a[3]), .b
(b[3]), .c_in (c2) );
endmodule
```

Таким образом, мы реализовали четырехбитный сумматор.
Мы получили его как модуль верхнего уровня *adder4*, состоящий из модулей *adder1*, которые, в свою очередь состоят из модулей примитивов AND2 и XOR.

Вообще, даже примитивы типа AND могут быть описаны на языке Verilog в виде нескольких связанных MOS или CMOS переключателей. Это уже уровень транзисторов (*switch level modeling*). Мы этим заниматься конечно сейчас не будем. Странно думать о транзисторах, если проект может использовать их тысячи и миллионы

Урок 3. Арифметические и логические функции.

Сейчас, мы уже знаем про модули, их входные и выходные сигналы и как они могут быть соединены друг с другом. На прошлом уроке я рассказал, как можно сделать многобитный сумматор. Нужно ли каждый раз, когда складываем два числа, делать такие сложные модули, как на том уроке? Конечно нет! Давайте познакомимся с основными арифметическими и логическими операторами языка Verilog.

С помощью комбинаторной логики посчитаем некоторые арифметические и логические функции. Программисты C/C++ будут чувствовать себя просто как дома.

- **Сложение и вычитание.**

Вот пример модуля, который одновременно и складывает и вычитает два числа. Здесь входные операнды у нас 8-ми битные, а результат 9-ти битный. Verilog корректно сгенерирует бит переноса (*carry bit*) и поместит его в девятый бит выходного результата. С точки зрения Verilog входные операнды беззнаковые. Если нужна знаковая арифметика, то об этом нужно отдельно позаботиться.

```
module simple_add_sub(
         operandA, operandB,
         out_sum, out_dif);

//два входных 8-ми битных операнда
input [7:0] operandA, operandB;

/*Выходы для арифметических операций имеют дополнительный 9-й бит
переполнения*/
output [8:0] out_sum, out_dif;

assign out_sum = operandA + operandB;
assign out_dif = operandA - operandB;

endmodule
```

- **Логический и арифметический сдвиг.**

Вот пример модуля, который выполняет сдвиги. В нашем примере результат для сдвига влево 16-ти битный (ну просто так, для интереса). Если сдвигать влево или вправо слишком далеко, то результат получится просто ноль. Часто для сдвига используются только часть бит от второго операнда, для того, чтобы сэкономить логику.

```
module simple_shift (
     operandA, operandB,
     out_shl, out_shr, out_sar);

// два входных 8-ми битных операнда
input [7:0] operandA, operandB;

// Выходы для операций сдвига
output [15:0] out_shl;
output [7:0]  out_shr;
output [7:0]  out_sar;

//логический сдвиг влево
assign out_shl = operandA << operandB;

/* пример: на сколько сдвигать определяется 3-мя битами второго
операнда */
assign out_shr = operandA >> operandB[2:0];

//арифметический сдвиг вправо (сохранение знака числа)
assign out_sar = operandA >>> operandB[2:0];

endmodule
```

- **Битовые логические операции**

Битовые операции в Verilog выглядят так же, как и в языке C. Каждый бит результата вычисляется отдельно соответственно битам операндов. Вот пример:

```
module simple_bit_logic (
       operandA, operandB,
       out_bit_and, out_bit_or, out_bit_xor, out_bit_not);

//два входных 8-ми битных операнда
input [7:0] operandA, operandB;

//Выходы для битовых (bit-wise) логических операций
output [7:0] out_bit_and, out_bit_or, out_bit_xor, out_bit_not;

assign out_bit_and = operandA & operandB;
assign out_bit_or  = operandA | operandB;
assign out_bit_xor = operandA ^ operandB;
assign out_bit_not = ~operandA;

endmodule
```

- **Булевые логические операции.**

Булевые логические операторы отличаются от битовых операций. Так же, как и в языке С, здесь значение всей шины рассматривается как ИСТИНА если хотя бы один бит в шине не ноль или ЛОЖЬ, если все биты шины – ноль. Результат получается всегда однобитный (независимо от разрядности операндов) и его значение "1" (ИСТИНА) или "0" (ЛОЖЬ).

```
module simple_bool_logic (
       operandA, operandB,
       out_bool_and, out_bool_or, out_bool_not);

//два входных 8-ми битных операнда
input [7:0] operandA, operandB;

// Выходы для булевых (boolean) логических операций
output out_bool_and, out_bool_or, out_bool_not;

assign out_bool_and = operandA && operandB;
assign out_bool_or  = operandA || operandB;
assign out_bool_not = !operandA;

endmodule
```

- **Операторы редукции.**

Verilog имеет операторы редукции. Эти операторы позволяют выполнять операции между битами внутри одной шины. Так, можно определить все ли биты в шине равны единице (*&bus*), или есть ли в шине хотя бы одна единица (*|bus*). Приведу пример:

```
module simple_reduction_logic (
       operandA,
       out_reduction_and, out_reduction_or, out_redution_xor);

//входной 8-ми битный операнд
input [7:0] operandA;

// Выходы для логических операций редукции
output out_reduction_and, out_reduction_or, out_reduction_xor;

assign out_reduction_or  = |operandA;
assign out_reduction_and = &operandA;
assign out_reduction_xor = ^operandA;

endmodule
```

А вот еще полезные операторы редукции:
`~|operandA` обозначает, что в шине нет единиц.
`~&operandA` обозначает, что некоторые биты в шине равны нулю.

- **Оператор условного выбора**

Язык С имеет оператор ‘? :’. С его помощью можно выбрать одно значение из двух по результату логического выражения. В Verilog тоже есть подобный оператор. Он фактически реализует мультиплексор. В данном примере на выходе мультиплексора окажется значение *operandA* если сигнал *sel_in* единица. И наоборот. Если входной сигнал *sel_in* равен нулю, то на выходе мультиплексора будет значение *operandB*.

```
module simple_mux (
       operandA, operandB, sel_in, out_mux);

//входные 8-ми битные операнды
input [7:0] operandA, operandB;

//входной сигнал селектора
input sel_in;

//Выход мультиплексора
output [7:0]out_mux;

assign out_mux = sel_in ? operandA : operandB;

endmodule
```

- **Операторы сравнения.**

Можно ли сравнивать "числа" (а точнее значения регистров или шин) в Verilog? Конечно можно! Вот такой пример (все сравнения происходят с беззнаковыми числами):

```
module simple_compare (
       operandA, operandB,

       out_eq, out_ne, out_gt, out_lt, out_ge, out_le);

//входные 8-ми битные операнды
input [7:0] operandA, operandB;

//Выходы операций сравнения
output out_eq, out_ne, out_gt, out_lt, out_ge, out_le;

assign out_eq = operandA == operandB;
assign out_ne = operandA != operandB;
assign out_ge = operandA >= operandB;
assign out_le = operandA <= operandB;

assign out_gt = operandA > operandB;
assign out_lt = operandA < operandB;

endmodule
```

Ну вот, в приведенных примерах были рассмотрены основные арифметические и логические операторы языка Verilog. Конечно, кое-что я упустил в этом описании, но базовые операции рассмотрены. Вы можете заметить, что некоторые операторы совсем не описаны. Это такие операторы, как умножение (*) или деление (/), или какой нибудь модуль, остаток от деления (%). Мне кажется такие вещи обычно лучше избегать для синтеза. Конечно они полезны для написания моделей симуляции. Некоторые чипы могут иметь встроенные умножители, если синтезатор знает об этом, он может использовать их. Тем не менее, лучше делать такую работу вручную – так больше возможностей контролировать возможности конкретного чипа.

Урок 4. Поведенческие блоки.

Мы уже познакомились с постоянным назначением сигналов, оно выглядит, например, вот так:

```
wire a,b,c;
assign c = a & b;
```

Постоянные назначения весьма полезны, но и они имеют недостатки. Такой код, когда его много, не очень легко читать. Чтобы сделать язык Verilog более выразительным, он имеет так называемые "*always*" блоки. Они используются при описании системы с помощью поведенческих блоков (*behavioral blocks*). Использование поведенческих блоков очень похоже на программирование на языке C. Оно позволяет выразить алгоритм так, чтобы он выглядел как последовательность действий (даже если в конечном счете в аппаратуре это будет не так).

Для описания поведенческого блока используется вот такой синтаксис:

```
always @(<sensitivity_list>) <statements>
```

<sensitivity_list> – это список всех входных сигналов, к которым чувствителен блок. Это список входных сигналов, изменение которых влияет выходные сигналы этого блока. "*Always*" переводится как "всегда". Такую запись можно прочитать вот так: "Всегда выполнять выражения <statements> при изменении сигналов, описаных в списке чувствительности <sensitivity list>".

Если указать список чувствительности неверно, то это не должно повлиять на синтез проекта, но может повлиять на его симуляцию. В списке чувствительности имена входных сигналов разделяются ключевым словом "*or*":

```
always @(a or b or d) <statements>
```

Иногда гораздо проще и надежней включать в список чувствительности все сигналы. Это делается вот так:

```
always @* <statements>
```

Тогда исправляя выражения в <statements> вам не нужно задумываться об изменении списка чувствительности.

При описании выражений внутри поведенческих блоков комбинаторной логики, с правой стороны от знака равенства, как и раньше, можно использовать типы сигналов *wire* или *reg*, а вот с левой стороны теперь используется только тип *reg*:

```
reg [3:0] c;
always @(a or b or d)
begin
c = <выражение использующее входные сигналы a,b,d>;
end
```

Обратите внимание, что регистры, которым идет присвоение в таких поведенческих блоках не будут выполнены в виде D-триггеров после синтеза. Это часто вызывает недоумение у начинающих.

Здесь мы делаем присвоение регистрам с помощью оператора "=", который называется "блокирующим". Для симулятора это означает, что выражение вычисляется, его результат присваивается регистру приемнику и он тут же, немедленно, может быть использован в последующих выражениях.

Таким образом, блокирующие присвоения используются для описания комбинаторной логики в поведенческих блоках. Не блокирующие присвоения будут описаны позднее – они используются для описания синхронной логики и вот уже там регистры *reg* после синтеза будут представлены с помощью D-триггеров. Не путайте блокирующие и не блокирующие присвоения!

Вы можете написать довольно много выражений связанных между собой, синтезатор переработает их и создаст, возможно, довольно длинные цепи из комбинаторной логики. Например:

```
wire [3:0] a, b, c, d, e;
reg [3:0] f, g, h, j;
always @(a or b or c or d or e)
begin
f = a + b;
g = f & c;
h = g | d;
j = h - e;
end
```

То же самое можно сделать по другому, вот так:

```
always @(a or b or c or d or e)
begin
j = (((a + b) & c) | d) - e;
end
```

На самом деле, после того, как проект будет откомпилирован, список всех сигналов проекта (*netlist*) может сильно сократиться. Многие сигналы, описанные программистом, могут исчезнуть – синтезатор выбросит их, создав цепи из оптимизированной комбинаторной логики. В нашем примере сигналы *f*, *g* и *h* могут исчезнуть из списка сигналов проекта после синтезатора, все зависит от того используются ли эти сигналы где-то еще в проекте или нет. Синтезатор даже может выдать предупреждение (*warning*) о том, что сигналу "*f*" присвоено значение, но оно нигде не используется – и такое тоже бывает.

Однако вернемся к нашим поведенческим блокам. Теперь с ними мы уже можем делать очень интересные вещи, такие как условные переходы, множественный выбор по условию, циклы и прочее.

Например, мы уже знаем как описать простой мультиплексор с помощью оператора "**?**", вот так:

```
reg [3:0] c;
always @(a or b or d)

begin
c = d ? (a & b) : (a + b);
end
```

А теперь можем написать это же, но по другому:

```
reg [3:0] c;
always @(a or b or d) begin
   if (d) begin
     c = a & b;
   end else begin
     c = a + b;
   end
end
```

Вместо параметра "*d*" может быть любое выражение. Если значение этого выражения истина (не равно нулю), то будет выполняться первое присвоение "*c = a & b*". Если значение выражения "*d*" ложь (равно нулю), то будет выполняться второе присвоение "*c = a + b*".

Если нужно сделать выбор из многих выриантов (это на языке схемотехники мультиплексор со многими входами), то можно использовать конструкцию *case*. Конструкции *case* очень похожи на *switch* из языка C.

Базовый синтаксис вот такой:

```
case (selector)
  option1: <statement>;
  option2: <statement>;
  default:  <if nothing else statement>;  //по желанию, но
желательно
endcase
```

А вот и простой пример:

```
wire [1:0] option;
wire [7:0] a, b, c, d;
reg [7:0] e;
always @(a or b or c or d or option) begin
case (option)
  0: e = a;
  1: e = b;
  2: e = c;
  3: e = d;
endcase
end
```

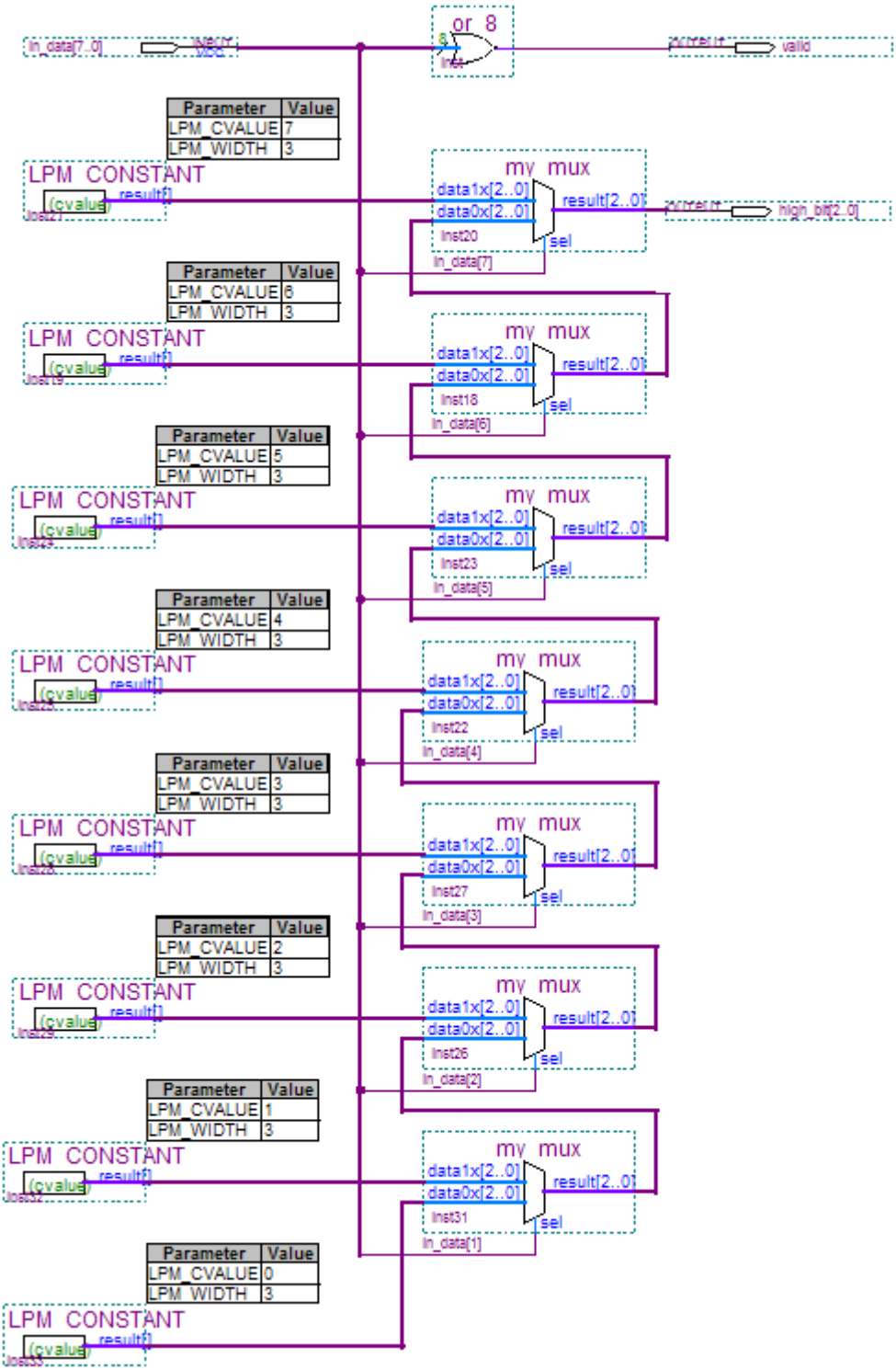
Поскольку входы у нас – это 8-ми битные шины, то в результате синтеза получится восемь мультиплексоров четыре-к-одному.

Давайте теперь рассмотрим циклы. Тут нужно сделать замечание, что циклы в Verilog имеют несколько иной смысл, не такой, как в языках C или Pascal. На языке C цикл обозначает, что некоторое действие должно быть выполнено последовательно раз за разом. Чем больше итераций в цикле, тем дольше он будет исполняться процессором. На языке Verilog цикл скорее описывает сколько экземпляров логических функций должно быть реализовано аппаратно. Чтобы синтез прошел успешно, циклы должны иметь заданное фиксированное число итераций – иначе синтезатор просто не сможет ничего сделать.

Рассмотрим простой пример – нужно определить номер самого старшего ненулевого бита вектора (шины):

```
module find_high_bit(input wire [7:0]in_data, output reg
[2:0]high_bit, output reg valid);

integer i;

always @(in_data)
begin
   //определим, есть ли в шине единицы
   valid = |in_data;

   //присвоим хоть что нибудь
   high_bit = 0;

   for(i=0; i<8; i=i+1)
   begin
     if(in_data[i])
     begin
       // запомним номер бита с единицей в шине
       high_bit = i;
     end
   end
end
endmodule
```

В приведенном примере цикл просматривает все биты "последовательно" от младшего к старшему. Когда будет найден ненулевой бит, то его порядковый номер запоминается в регистре *high_bit*. Когда цикл закончится, то регистр *high_bit* будет содержать индекс самого старшего ненулевого бита в шине (если конечно *valid* тоже не ноль). На самом деле, конечно, нужно представлять себе, что подобная запись цикла будет реализована в аппаратуре довольно длинной цепочкой из мультиплексоров. Вы должны представлять себе, что в этом примере цикл *for* - это примерно вот такая логическая схема:


in_data[7..0]
or_8
valid
Parameter
Value
LPM_CVALUE
LPM_WIDTH
7
6
5
4
3
2
1
0
3
LPM_CONSTANT
(cvalue)
result[]
my_mux
data1x[2..0]
data0x[2..0]
result[2..0]
sel
Inst20
Inst18
Inst23
Inst22
Inst27
Inst26
Inst31
Inst33
in_data[7]
in_data[6]
in_data[5]
in_data[4]
in_data[3]
in_data[2]
in_data[1]
high_bit[2..0]

Рассмотрим другой пример – нужно найти самый младший не нулевой бит в шине:

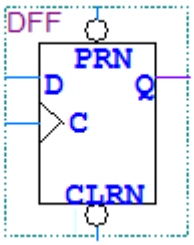
```
module find_low_bit(input wire [7:0]in_data, output reg
[2:0]low_bit, output reg valid);

integer i;

always @(in_data)
begin
   //определим, есть ли в шине единицы
   valid = |in_data;

   //присвоим хоть что нибудь
   low_bit = 0;

   for(i=7; i>=0; i=i+1)
   begin
     if(in_data[i])
     begin
       // запомним номер бита с единицей в шине
       low_bit = i;
     end
   end
end
endmodule
```

Теперь просмотр битов идет от старшего к младшим. Обратите внимание, что счетчик циклов определен как *integer* - это целое число со знаком. И так нужно, потому что сравнение ">=0" как раз и означает "не отрицательный". Переменная типа *reg* всегда положительна, значит здесь использоваться не может.

Урок 5. Синхронная логика.

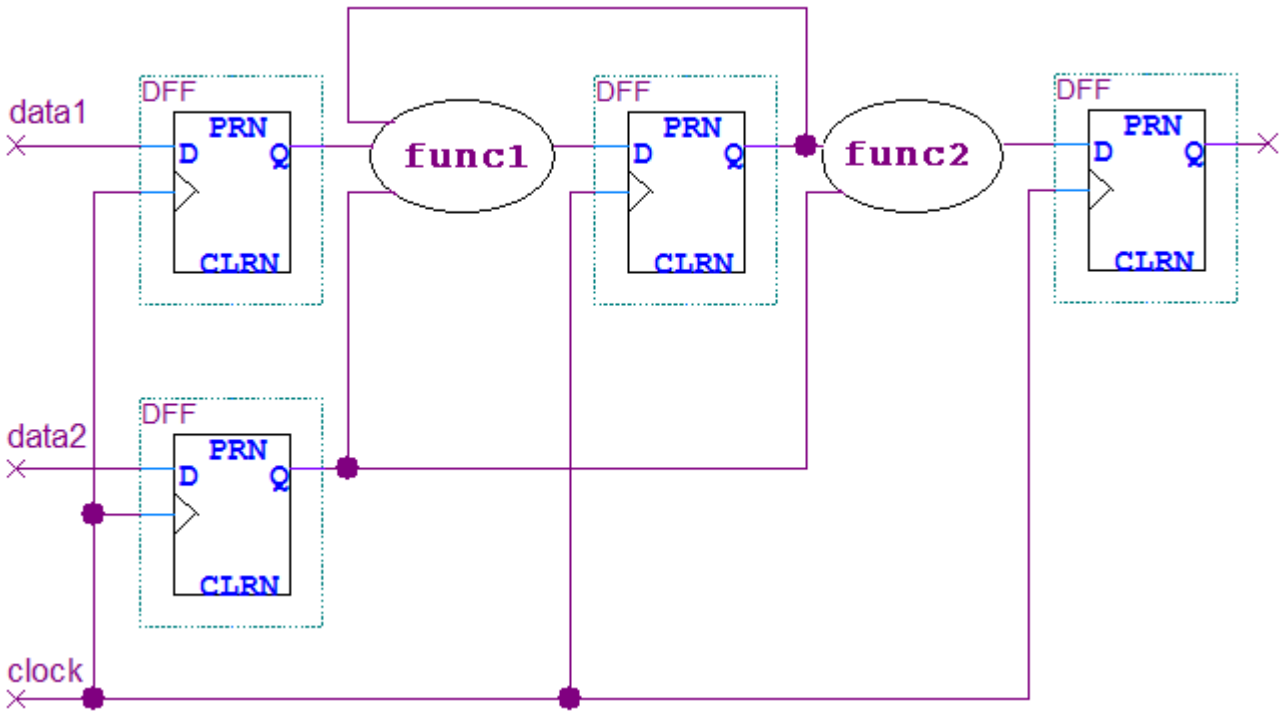
На предыдущих уроках мы уже познакомились с типами источников сигналов, узнали как установить экземпляры разных модулей в свой модуль и как соединить их проводами. Так же мы уже рассмотрели разные арифметические и логические выражения в поведенческом коде. Теперь нам осталось познакомиться с синхронной логикой и триггерами.

Существует несколько видов триггеров. Наиболее важные для практического применения – это D-триггера. Вот графическое изображение D-триггера:



D-Триггер (*flipflop*) – это специальный логический элемент, способный запоминать. Такой триггер запоминает логическое значение сигнала входа *D*, когда на втором входе *C* (обозначен треугольничком) появляется фронт сигнала. Фронт сигнала - это момент, когда входная линия переходит из состояния ноль в единицу. Один такой триггер запоминает один бит информации. Текущее значение записанное в триггер - на его выходе *Q*. Еще триггер может иметь сигналы ассинхронного сброса *clrn* или установки *prn*. По сигналу *clrn* = 0 триггер переходит в состояние ноль не зависимо от других входных сигналов *D* или *C*. Кое-что про триггера можно почитать в Википедии.

Все сложные цифровые схемы используют и комбинаторную логику и триггера. Вот типичный пример некой цифровой схемы:



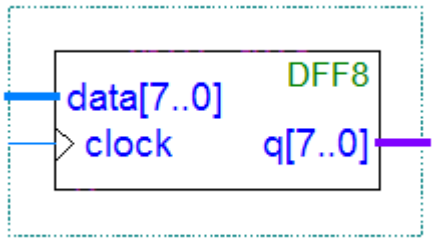
Логические функции *func1* и *func2* что-то вычисляют и по каждому фронту тактовой частоты результат вычисления запоминается в регистрах.

Синхронно с каждым импульсом тактовой частоты схема переходит из одного устойчивого состояния в другое. Состояние всей схемы определяется текущими значениями в триггерах. В синхронной логике (в отличии от комбинаторной логики) обратные связи используются очень часто – то есть выход триггера может спокойно подаваться на его вход прямо или через другую логику.

Язык Verilog позволяет легко описать синхронные процессы.

Описание синхронной логики в поведенческом коде Verilog очень похоже на описание комбинаторной логики с одним важным отличием – в списке чувствительности *always* блока теперь будут не сами сигналы, а фронт тактовой частоты *clock*. Только в этом случае регистры, которым идет присвоение, будут реализованы в виде D-триггеров (*flipflops*).

Давайте посмотрим простой пример. Вот графическое представление восьмибитного регистра, запоминающего входные данные по фронту тактовой частоты:



Его представление в Verilog может быть такое:

```
module DFF8 (clock, data, q);
input clock;
input [7:0] data;
output [7:0] q;
reg [7:0] q;
always @(posedge clock) begin
    q <= data;
end
endmodule
```

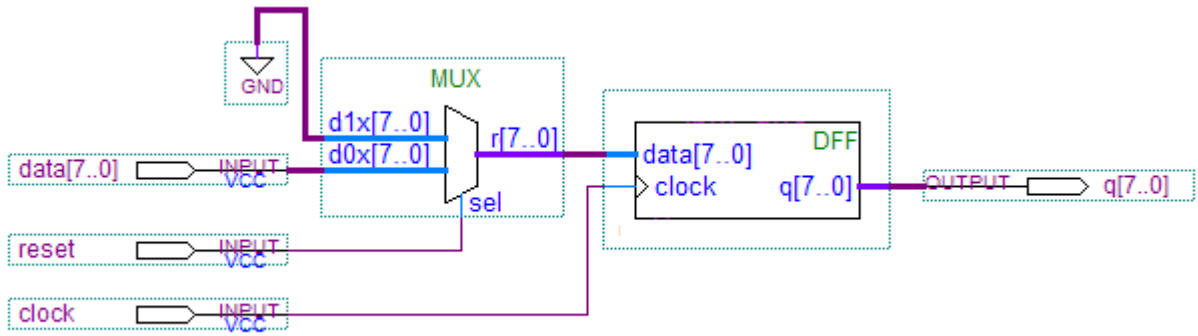
Здесь блок *always* чувствителен к фронту тактовой частоты: *posedge clock*. Еще, вместо *posedge* – положительного фронта тактовой частоты можно использовать *negedge* – отрицательный фронт. При использовании *negedge* запоминание в регистр происходит по перепаду сигнала *clock* с «1» в «0». Сам факт запоминания входных данных *data* в регистре *q* здесь описан оператором "<=". Это так называемое «неблокирующее присвоение».

Понятно, что на вход *in* такого регистра может быть подано, например, значение выражения какой нибудь комбинаторной функции, вычисленной в другом *always* блоке.

В синхронной логике значение входов триггеров важно только в момент запоминания в триггерах, в момент фронта тактовой частоты *clock*. В этом главное отличие от комбинаторной логики. Комбинаторная логика, описанная *always* блоками, производит «постоянное» вычисление своей функции по отношению к входным сигналам: как только меняется сигнал на входе, так сразу может меняться и сигнал на выходе.

Очень часто модули имеют дополнительные сигнал сброса. Они используются чтобы установить триггера схемы в некоторое исходное состояние. Сигналы сброса могут быть разных типов: синхронные и ассинхронные,

Вот пример реализации синхронного сброса регистра с активным положительным уровнем (*active-high*):

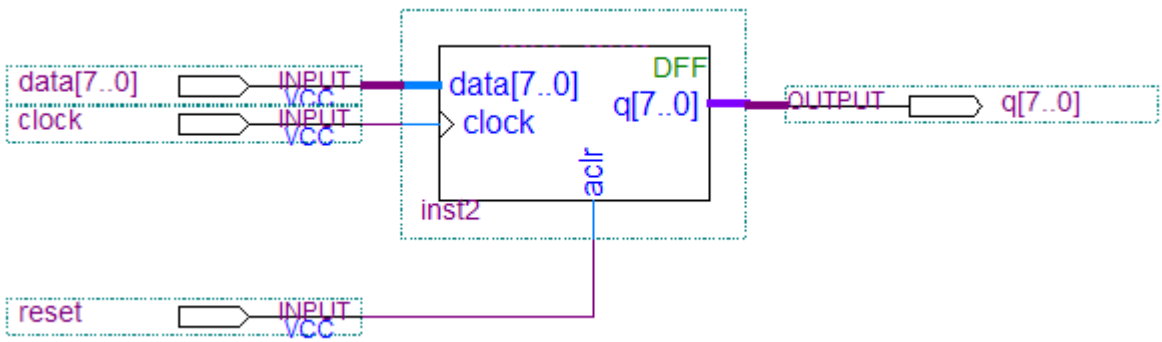


Вот так эту схему можно описать на языке Verilog:

```
module DFFSR (reset, clock, data, q);
input reset;
input clock;
input [7:0] data;
output [7:0] q;

reg [7:0] q;
always @(posedge clock) begin
  if (reset) begin
    q <= 0;
  end else begin
    q <= data;
  end
end

endmodule
```

Фактически, здесь при использовании синхронного сброса перед входом запоминающего регистра стоит мультиплексор. Если сигнал сброса ***reset*** активен ("единица"), то на вход регистра подается ноль. Иначе – загружаются другие полезные данные. Запись в регистр происходит синхронно с фронтом тактовой частоты *clock*.

Ассинхронный сброс с активным сигналом единица описывается следующим образом. Вот схема:



А вот представление на Verilog:

```
module DFFAR (reset, clock, data, q);
input reset;
input clock;
input [7:0] data;
output [7:0] q;
reg [7:0] q;
always @(posedge clock or posedge reset) begin
  if (reset) begin
    q <= 0;
  end else begin
    q <= data;
  end
end

endmodule
```

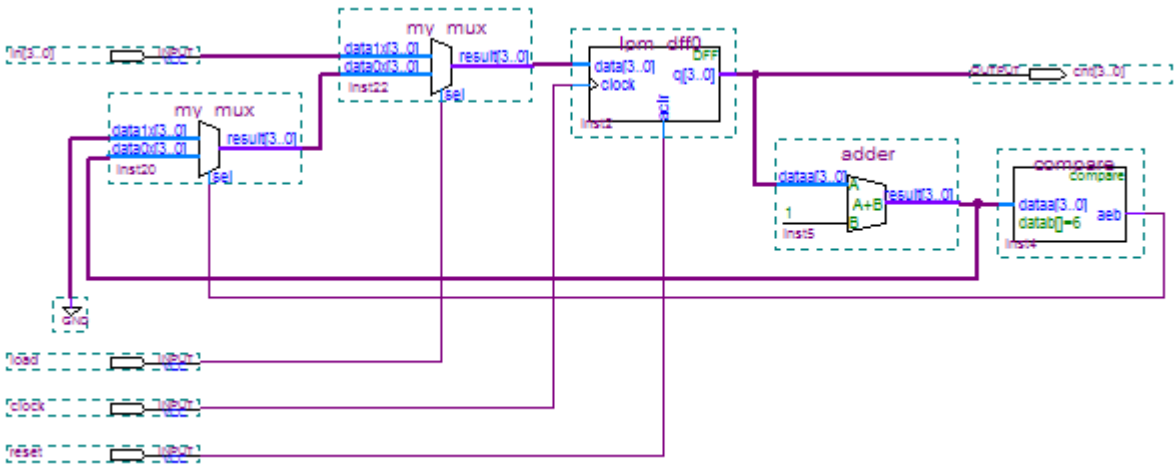
Обратите внимание, что сигнал *reset* теперь попал в список чувствительности для *always* блока.

Вообще-то для ассинхронных сбросов желательно всегда придерживаться вот такого стиля описания синхронной логики, как в приведенном выше примере. Сперва пишите "*if(reset)*" и все присвоения связанные с начальной инициализацией регистров. Потом пишите "*else*" и всю остальную логику. Дело в том, что синтезатор пытается в нашем коде выделить знакомые ему конструкции. Если написать иначе, хотя и логически правильно, то остается риск, что синтезатор чего-то не поймет и сгенерирует схему иначе, не используя все возможности триггера, не используя его ассинхронный сброс. В результате схема может стать больше (для ее реализации нужно больше логических элементов) и "медленнее" (работает стабильно на меньшей частоте).

Давайте рассмотрим еще один полезный практический пример. Предположим, что нам нужен счетчик по модулю 6, но чтобы исходное значение счетчика можно было бы загружать. Вот код описывающий такой счетчик:

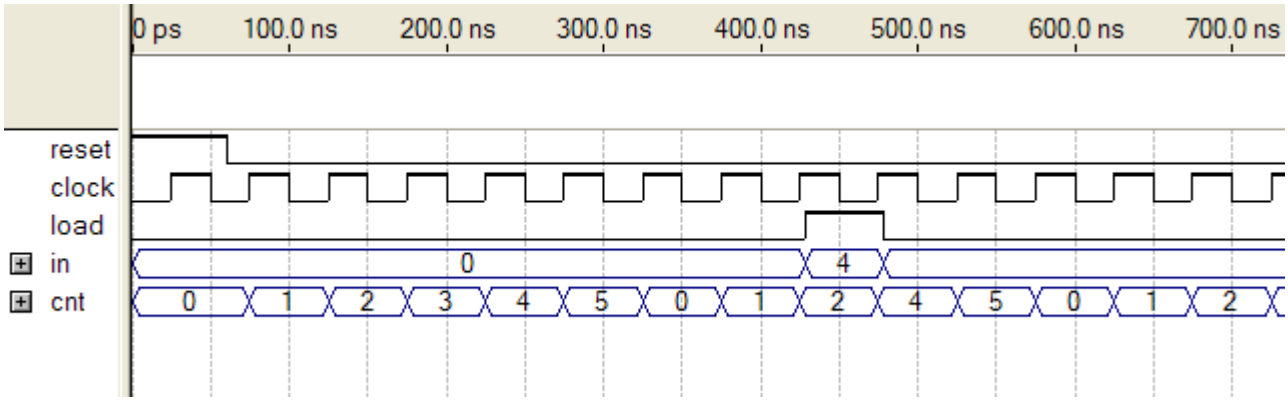
```
module mod_counter(
    input wire reset,
    input wire clock,
    input wire [3:0]in,
    input wire load,
    output reg [3:0]cnt
    );

parameter MODULE = 6;

always @(posedge clock or posedge reset)
begin
  if(reset)
    cnt <= 4'b0000;
  else
  begin
  if(load)
    cnt <= in;
  else
    if(cnt+1==MODULE)
       cnt <= 4'b0000;
    else
       cnt <= cnt + 1'b1;
  end
end

endmodule
```

В этом модуле у нас есть ассинхронный сброс счетчика, синхронная загрузка счетчика по условию *load* и синхронное увеличение значения счетчика до модуля. Эквивалентная схема такого модуля будет вот такая:



Здесь видно два мультиплексора, сумматор, компаратор и собственно регистр.

Оба модуля, и написаный на Verilog и выполненный в виде схемы, работают абсолютно одинаково:



На диаграмме симуляции видно, что если нет сигнала *reset*, то по каждому фронту тактовой частоты значение счетчика растет до модуля. Потом счет начинается сначала. Если появляется сигнал загрузки *load*, то значение счетчика перезагружается новым входным значением.

Теперь давайте посмотрим внимательнее на операторы присвоения языка Verilog. В случае синхронной логики присвоение обычно обозначает запись в регистр или триггер синхронно с фронтом тактовой частоты. Однако с точки зрения языка программирования Verilog мы опять подошли к этим двум понятиям: блокирующее и неблокирующее присвоение.

Блокирующее присвоение (с помощью оператора "=") называется так потому, что вычисления производятся строго в порядке, описанном в *always* блоке. Второе выражение вычисляется только после первого и результат второго выражения может зависить от результата первого.

Неблокирующее присвоение в *always* блоке (используется оператор "<=") обозначает факт одновременного запоминания вычисленных значений в соответствующих регистрах. Выражения в *always* блоке выполняются не последовательно, а одновременно. И присвоение, тоже произойдет одновременно.

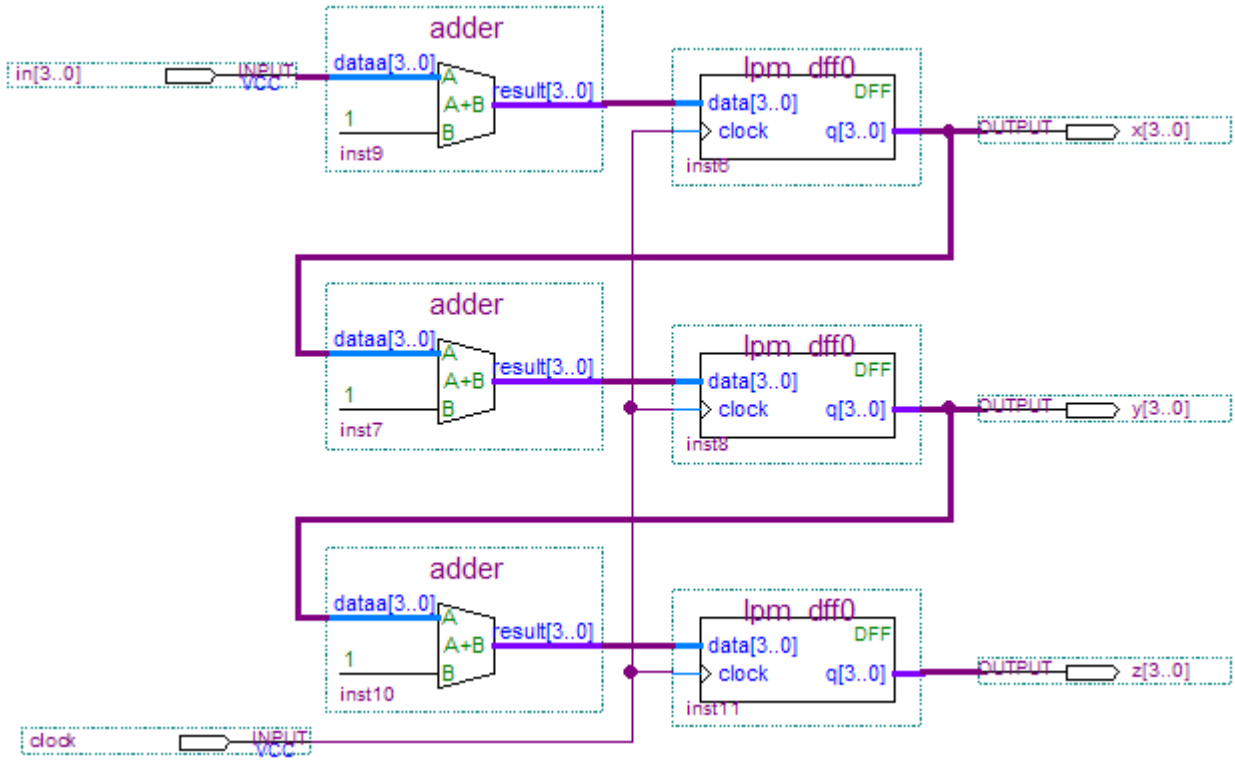
Обычно принято применять блокирующие присвоения при описании комбинаторной логики, а неблокирующие для синхронной логики. Так проще всего представлять себе описываемые алгоритмы (хотя оба типа присвоений могут соседствовать в одном *always* блоке).

Давайте попробуем поэкспериментировать с блокирующими и неблокирующими присвоениями.

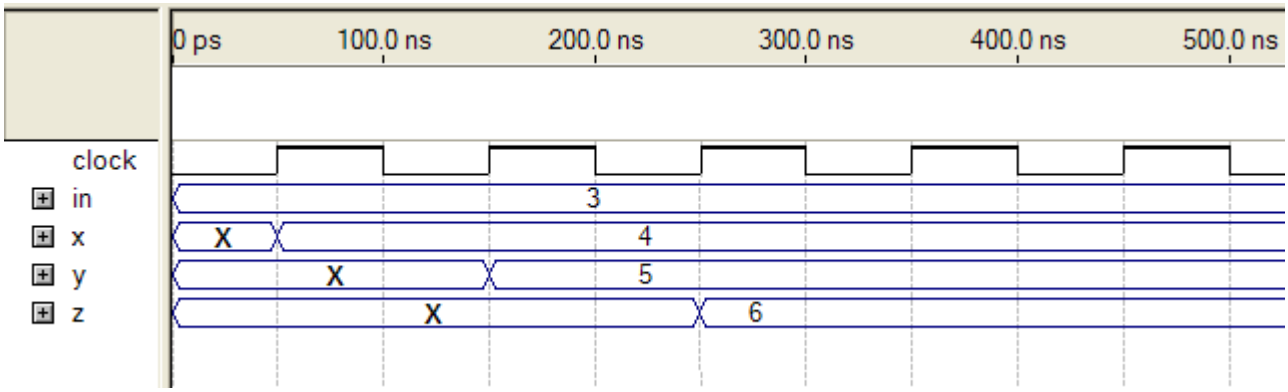
Рассмотрим вот такой пример:

```
module test(
  input wire clock,
  input wire [3:0]in,
  output reg [3:0]x,
  output reg [3:0]y,
  output reg [3:0]z );

always @(posedge clock)
begin
  x <= in + 1;
  y <= x + 1;
  z <= y + 1;
end

endmodule
```

Из этого кода синтезатор сделает три физических четырехбитных регистра на триггерах и три сумматора. Эквивалентная схема к такому коду будет вот такая:



Попробуем просимулировать такой модуль. Предположим исходное состояние триггеров неизвестно (значение X). Пусть, например, входное значение шины *in* равно 3 и модуль тактируется частотой *clock*.



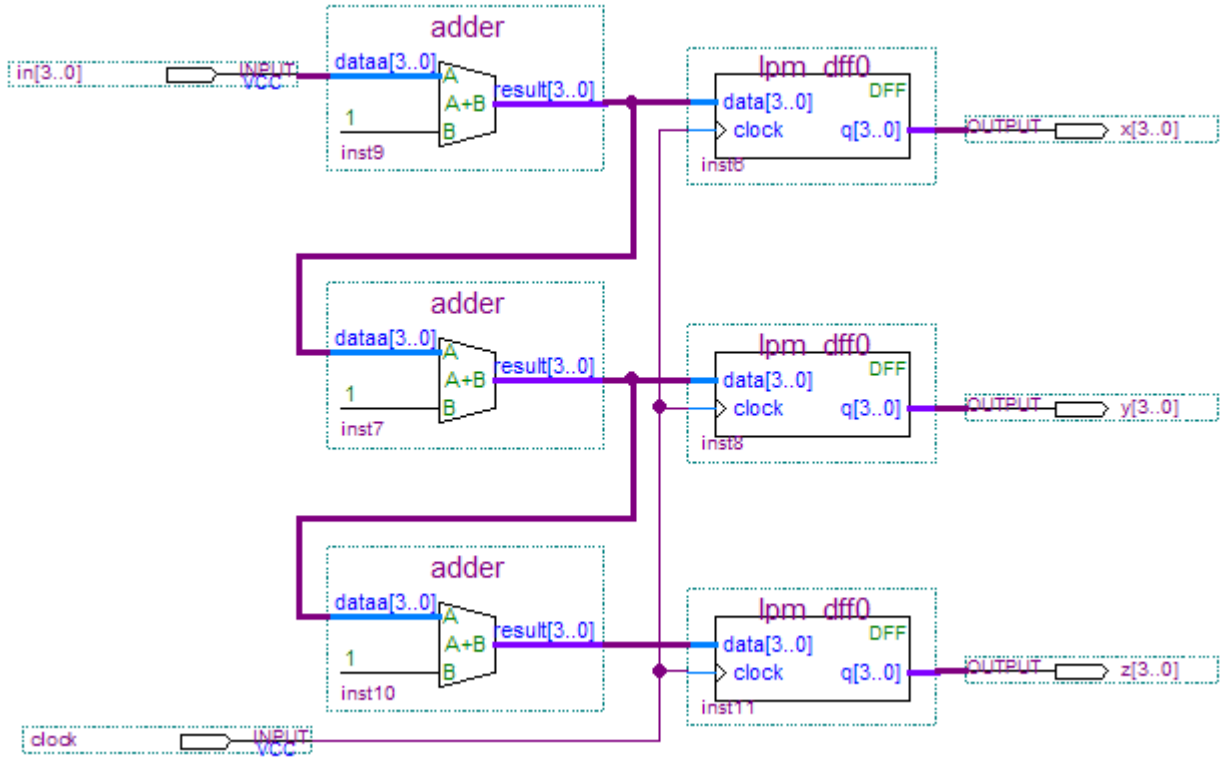
На этом рисунке виден результат симуляции.

Вместе с первым фронтом тактовой частоты *clock* в регистр *x* будет записано значение суммы *in+1* и оно равно четырем. В этот же момент времени в регистр *y* должна быть записана сумма *x+1*, но симулятор еще не знает текущего значения регистра *x*. Оно еще не определено. Только второй импульс тактовой частоты запишет в регистр *y* число 5. Присвоение в регистр *z* так же происходит синхронно с остальными присвоениями. Но только на третьем такте в регистр *z* запишется число 6, так как только к этому времени симулятору будет известно значение регистра *y*. В реальной схеме все работает точно так же. Сигнал на выходе *z* получается задержанным на 2 такта относительно выхода *x*.

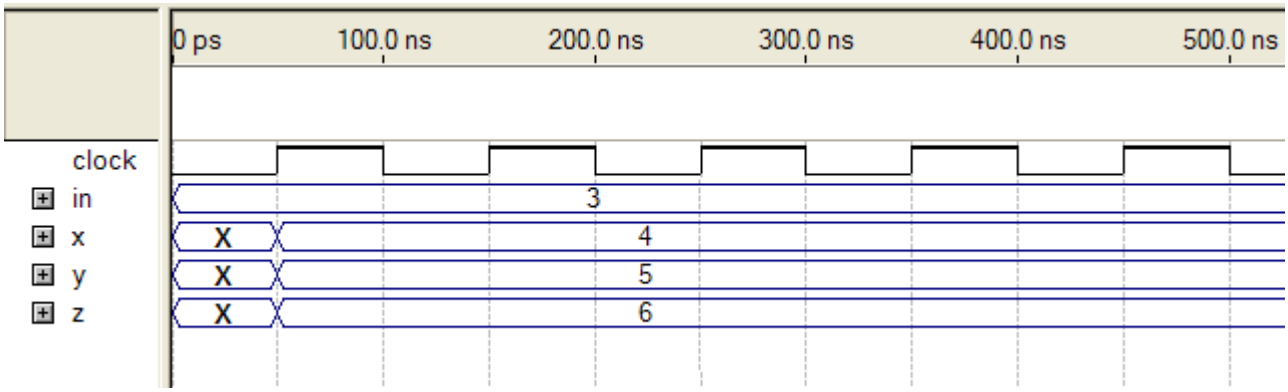
Попробуем теперь изменить наш модуль. Сейчас мы будем использовать блокирующее присвоение:

```
module test(
  input wire clock,
  input wire [3:0]in,
  output reg [3:0]x,
  output reg [3:0]y,
  output reg [3:0]z );

always @(posedge clock)
begin
  x = in + 1;
  y = x + 1;
  z = y + 1;
end

endmodule
```

Поведение модуля стало принципиально другим! И это неудивительно, ведь этот код описывает теперь уже другую цифровую схему. Регистры теперь соединены не последовательно один за другим, а как бы параллельно, только загружаемые значения разные:



Результат симуляции:



Поскольку мы использовали блокирующие присвоения, то это обозначает, что следующее выражение учитывает результат предыдущего (в порядке написания). Фактически в этом случае наша запись подразумевает следующее:

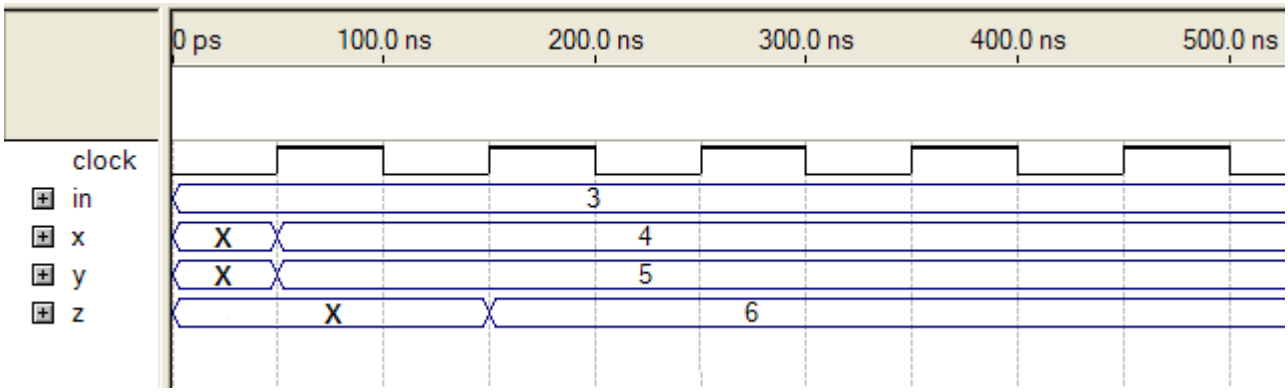
```
always @(posedge clock)
begin
  x = in + 1;
  y = in + 2; // y = x+1 = (in+1)+1
  z = in + 3; // z = y+1 = ((in+1)+1)+1
end
```

Мы так же получим три регистра, только в них будет записаны суммы одновременно уже в первом такте.

Давайте теперь попробуем изменить порядок строк наших выражений в *always* блоке. Неблокирующие присвоения даже не будем пробовать - результат не изменится. А вот с блокирующими присвоениями не так. Тут важна очередность выражений. Поменяем местами две строки, вот так:

```
always @(posedge clock)

begin
x = in + 1;
z = y + 1;
y = x + 1;
end
```

Получаем вот такой результат:



В регистр *z* будет помещено результирующая сумма на такт позже, чем в регистр *y*. И это правильно! По первому фронту тактовой частоты может быть вычисленно точно только *x* и *y*, но нельзя вычислить *z*. К моменту вычисления *z* по первому фронту значение *y* неопределено. А вот на втором фронте сигнала *clock* уже *z* будет посчитан.

Вот так. Блокирующие и неблокирующие присвоения действуют по разному.

Сейчас можно подвести некоторый итог: неблокирующие присвоения "<=" в языке Verilog позволяют описать многочисленные одновременные присвоения регистрам и учесть все задержки распространения данных внутри цифровой схемы.

Пожалуй на этом можно и закончить наш краткий курс по языку Verilog. Конечно за пять уроков невозможно рассказать все, но я думаю, что основные моменты были рассмотрены.

Надеюсь было не очень скучно.