

# Лабораторный практикум

«Проектирование цифровых устройств с помощью  
Verilog HDL»

# Лабораторная работа №1

## Введение в Verilog HDL

### 1.1 Возникновение языков описания цифровой аппаратуры

Цифровые устройства — это устройства, предназначенные для приёма и обработки цифровых сигналов. Цифровыми называются сигналы, которые можно рассматривать в виде набора дискретных уровней. В цифровых сигналах информация кодируется в виде конкретного уровня напряжения. Как правило выделяется два уровня — логический «0» и логическая «1».

Цифровые устройства стремительно развиваются с момента изобретения электронной лампы, а затем транзистора. Со временем цифровые устройства стали компактнее, уменьшилось их энергопотребление, возрасла вычислительная мощность. Так же разительно возросла сложность их структуры.

Графические схемы, которые применялись для проектирования цифровых устройств на ранних этапах развития, уже не могли эффективно использоваться. Потребовался новый инструмент разработки, и таким инструментом стали языки описания аппаратной части цифровых устройств (Hardware Description Languages, HDL), которые описывали цифровые структуры формализованным языком, чем-то похожим на язык программирования.

Совершенно новый подход к описанию цифровых схем, реализованный в языках HDL, заключается в том, что с помощью их можно описывать не только структуру, но и поведение цифрового устройства. Окончательная структура цифрового устройства получается путём обработки таких смешанных описаний специальной программой — синтезатором.

Такой подход существенно изменил процесс разработки цифровых устройств, превратив громоздкие, тяжело читаемые схемы в относительно простые и доступные описания поведения.

В данном курсе мы рассмотрим язык описания цифровой аппаратуры Verilog HDL — один из наиболее распространённых на текущий момент. И начнём мы с разработки наиболее простых цифровых устройств — логических вентилях.

## 1.2 HDL описания логических вентилях

Логические вентили реализуют функции алгебры логики: И, ИЛИ, Исключающее ИЛИ, НЕ. Напомним их таблицы истинности:

$a$	$b$	$a \cdot b$
0	0	0
0	1	0
1	0	0
1	1	1

Таблица 1.1: И

$a$	$b$	$a \oplus b$
0	0	0
0	1	1
1	0	1
1	1	0

Таблица 1.3: Исключающее ИЛИ

$a$	$b$	$a b$
0	0	0
0	1	1
1	0	1
1	1	1

Таблица 1.2: ИЛИ

$a$	$\bar{a}$
0	1
1	0

Таблица 1.4: НЕ

Начнём знакомиться с Verilog HDL с описания логического вентиля «И». Ниже приведен код, описывающий вентиль с точки зрения его структуры:

```

1 module and_gate(
2     input a,
3     input b,
4     output result)
5
6 assign result = a & b;
7
8 endmodule

```

Листинг 1.1: Модуль, описывающий вентиль «И»

Описанный выше модуль можно представить как некоторый «ящик», в который входит 2 провода с названиями «*a*» и «*b*» и из которого выходит один провод с названием «*result*». Внутри этого блока результат выполнения операции «И» (в синтаксисе Verilog записывается как «&») над входами соединяют с выходом.

Схематично изобразим этот модуль:

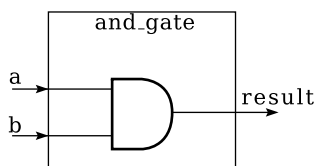


Рис. 1.1: Структура модуля «and\_gate»

Аналогично опишем все оставшиеся вентили:

```

1 module or_gate(
2     input a,
3     input b,
4     output result)
5
6 assign result = a | b;
7
8 endmodule

```

Листинг 1.2: Модуль, описывающий вентиль «ИЛИ»

```

1 module xor_gate(
2     input a,
3     input b,
4     output result)
5
6 assign result = a ^ b;
7
8 endmodule

```

Листинг 1.3: Модуль, описывающий вентиль  
«Исключающее ИЛИ»

```

1 module not_gate(
2     input a,
3     output result)
4
5 assign result = ~a;
6
7 endmodule

```

Листинг 1.4: Модуль, описывающий вентиль «НЕ»

В проектировании цифровых устройств логические вентили наиболее часто используются для формулировки и проверки сложных условий, например:

```

1 if ( (a & b) | (~c) ) begin
2     ...
3 end

```

Листинг 1.5: Пример использования логических вентиляей

Условие будет выполняться либо когда *не* выполнено условие «с», либо когда одновременно выполняются условия «а» и «b». *Здесь и далее под условием понимается логический сигнал, отражающий его истинность.*

В качестве входов, выходов и внутренних соединений в блоках могут использоваться шины — группы проводов. Ниже приведен пример работы с шинами:

```

1 module bus_or(
2     input  [7:0] x,
3     input  [7:0] y,
4     output [7:0] result);
5
6 assign result = x | y;
7
8 endmodule

```

Листинг 1.6: Модуль, описывающий побитовое «ИЛИ»  
между двумя шинами

Это описание описывает побитовое «ИЛИ» между двумя шинами по 8 бит. То есть описываются восемь логических вентилей «ИЛИ», каждый из которых имеет на входе соответствующие разряды из шины «x» и шины «y».

При использовании шин можно в описании использовать конкретные биты шины и группы битов. Для этого используют квадратные скобки после имени шины:

```

1 module bitwise_ops(
2     input  [7:0] x,
3     output [4:0] a,
4     output      b,
5     output [2:0] c);
6
7 assign a = x[5:1];
8 assign b = x[5]  | x[7];
9 assign c = x[7:5] ^ x[2:0];
10
11 endmodule

```

Листинг 1.7: Модуль, демонстрирующий  
битовую адресацию шин

Такому описанию соответствует следующая структурная схема, приведённая на Рис. 1.2



Рис. 1.2: Структура модуля «bitwise\_ops»

Впрочем, реализация ФАЛ с помощью логических вентилях не всегда представляется удобной. Допустим нам нужно описать таблично-заданную ФАЛ. Тогда описания этой функции при помощи логических вентилях нам придётся сначала минимизировать её и только после этого, получив логическое выражение (которое, несмотря на свою минимальность, не обязательно является коротким), сформулировать его с помощью языка Verilog HDL. Как видно, ошибку легко допустить на любом из этих этапов.

Одно из главных достоинств Verilog HDL — это возможность описывать поведение цифровых устройств вместо описания их структуры.

Программа-синтезатор анализирует синтаксические конструкции поведенческого описания цифрового устройства на Verilog HDL, проводит оптимизацию и, в итоге, вырабатывает структуру, реализующую цифровое устройство, которое соответствует заданному поведению.

Используя эту возможность, опишем таблично-заданную ФАЛ на Verilog HDL:

```

1 module function(
2     input x0,
3     input x1,
4     input x2,
```

```

5   output reg y);
6
7   wire [2:0] x_bus;
8   assign x_bus = {x2, x1, x0};
9
10  always @(xbus) begin
11      case (xbus)
12          3'b000: y <= 1'b0;
13          3'b010: y <= 1'b0;
14          3'b101: y <= 1'b0;
15          3'b110: y <= 1'b0;
16          3'b111: y <= 1'b0;
17          default: y <= 1'b1;
18      endcase;
19  end;
20
21  endmodule;

```

Листинг 1.8: Пример описания таблично-заданной ФАЛ на Verilog HDL

Описание, приведённое выше, определяет  $y$ , как таблично-заданную функцию, которая равна нулю на наборах 0, 2, 5, 6, 7 и единице на всех остальных наборах.

Остановимся подробнее на новых синтаксических конструкциях:

Описание нашего модуля начинается с создания трёхбитной шины «x\_bus» на строке 7.

После создания шины «x\_bus», на она подключается к объединению проводов «x2», «x1» и «x0» с помощью оператора assign как показано на Рис. 1.3.

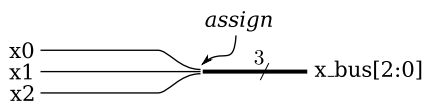


Рис. 1.3: Действие оператора **assign**

Затем начинается функциональный блок **always**, на котором мы остановимся подробнее.

Verilog HDL описывает цифровую аппаратуру, которая су-



шествует вся одновременно, но инструменты анализа и синтеза описаний являются программами и выполняются последовательно на компьютере. Так возникла необходимость последовательной программе «рассказать» про то, какие события приводят к срабатыванию тех или иных участков кода. Сами эти участки называли процессами. Процессы обозначаются ключевым словом **always**.

В скобках после символа @ указывается так называемый *список чувствительности процесса*, т.е. те сигналы, изменение которых должно приводить к пересчёту результатов выполнения процесса.

Например, результат ФАЛ надо будет пересчитывать каждый раз, когда изменился входной вектор (любой бит входного вектора, т.е. любая переменная ФАЛ). Эти процессы можно называть блоками, или частями будущего цифрового устройства.

Новое ключевое слово **reg** здесь необходимо потому, что в выходной вектор происходит запись, а запись в языке Verilog HDL разрешена только в «регистры» — специальные «переменные», предусмотренные в языке. Данная концепция и ключевое слово **reg** будет рассмотрено гораздо подробнее в следующей лабораторной работе.

Оператор **<=** называется оператором *неблокирующего присваивания*. В результате выполнения этого оператора то, что стоит справа от него, «помещается» («кладется», «перекладывается») в регистр, который записан слева от него. Операции неблокирующего присваивания происходят одновременно по всему процессу.

Оператор **case** описывает выбор действия в зависимости от анализируемого значения. В нашем случае анализируется значение шины «x\_bus». Ключевое слово **default** используется для обозначения всех остальных (не перечисленных) вариантов значений.

Константы и значения в языке Verilog HDL описываются следующим образом: сначала указывается количество бит, затем после апострофа с помощью буквы указывается формат и, сразу за ним, записывается значение числа в этом формате.

Возможные форматы:

- b – бинарный, двоичный;

- h – шестнадцатеричный;
- d – десятичный.

Немного расширив это описание, легко можно определить не одну, а сразу несколько ФАЛ одновременно. Для упрощения записи сразу объединим во входную шину все переменные. В выходную шину объединим значения функций:

```

1 module decoder(
2     input  [2:0] x,
3     output [3:0] y);
4
5 reg [3:0] decoder_output;
6 always @(x) begin
7     case (x)
8         3'b000: decoder_output <= 4'b0100;
9         3'b001: decoder_output <= 4'b1010;
10        3'b010: decoder_output <= 4'b0111;
11        3'b011: decoder_output <= 4'b1100;
12        3'b100: decoder_output <= 4'b1001;
13        3'b101: decoder_output <= 4'b1101;
14        3'b110: decoder_output <= 4'b0000;
15        3'b111: decoder_output <= 4'b0010;
16    endcase;
17 end;
18
19 assign y = decoder_output;
20
21 endmodule;
```

Листинг 1.9: Описание дешифратора на языке Verilog HDL

Теперь нам удалось компактно записать четыре функции, каждая от трёх переменных:

$$\begin{aligned}
 y_0 &= f(x_2, x_1, x_0); \\
 y_1 &= f(x_2, x_1, x_0); \\
 y_2 &= f(x_2, x_1, x_0); \\
 y_3 &= f(x_2, x_1, x_0).
 \end{aligned}$$

Но, если мы посмотрим на только что описанную конструк-

цию под другим углом, мы увидим, что это описание можно трактовать следующим образом: «поставить каждому возможному входному вектору  $x$  в соответствие заранее определенный выходной вектор  $y$ ». Такое цифровое устройство называют *дешифратором*.

На Рис. 1.4 показано принятое в цифровой схемотехнике обозначение дешифратора.

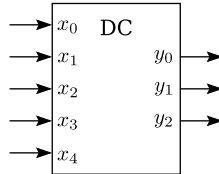


Рис. 1.4: Графическое обозначение дешифратора

Заметим, что длины векторов не обязательно должны совпадать, а единственным условием является полное покрытие всех возможных входных векторов, что, например, может достигаться использованием условия **default** в операторе **case**.

Дешифраторы активно применяются при разработке цифровых устройств. В большинстве цифровых устройств в явном или неявном виде можно встретить дешифратор.

Рассмотрим еще один интересный набор ФАЛ:

```

1  module decoder(
2      input [2:0] a,
3      input [2:0] b,
4      input [2:0] c,
5      input [2:0] d,
6      input [1:0] s,
7      output reg [2:0] y);
8
9  always @(a,b,c,d,s) begin
10     case (s)
11         3'b00:    y <= a;
12         3'b01:    y <= b;
13         3'b10:    y <= c;
14         3'b11:    y <= d;
15         default:  y <= a;

```

```

16   endcase;
17   end;
18
19   endmodule;

```

Листинг 1.10: Описание мультиплексора на языке Verilog HDL

Что можно сказать об этом описании? Выходной вектор  $y$  — это результат работы трёх ФАЛ, каждая из которых является функцией 6 переменных. Так,  $y_0 = f(a_0, b_0, c_0, d_0, s_1, s_0)$ .

Анализируя оператор **case**, можно увидеть, что главную роль в вычислении значения ФАЛ играет вектор  $s$ , в результате проверки которого выходу ФАЛ присваивается значение «выбранной» переменной.

Получившееся устройство называется *мультиплексор*.

Мультиплексор работает подобно коммутирующему ключу, замыкающему выход с выбранным входом. Для выбора входа мультиплексору нужен сигнал управления.

Графическое изображение мультиплексора приведено на Рис. 1.5

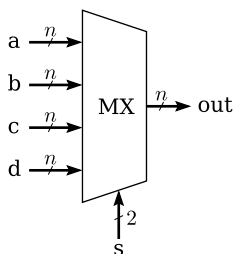


Рис. 1.5: Графическое обозначение мультиплексора

Особенно хочется отметить, что на самом деле никакой «проверки» сигнала управления не существует и уж тем более не существует «коммутации», ведь мультиплексор — это таблично-заданная ФАЛ. Результат выполнения этой ФАЛ выглядит так, как будто происходит «подключение» «выбранной» входной шины к выходной.

Приведём для наглядности таблицу, задающую ФАЛ для одного бита выходного вектора (число ФАЛ в мультиплексоре и,

следовательно, число таблиц, равняется числу бит в выходном векторе). Для краткости выпишем таблицу наборами строк вида:  $f(s_1, s_0, a_0, b_0, c_0, d_0) = y_0$  в четыре столбца.

Обратите внимание, что в качестве старших двух бит входного вектора для удобства записи и анализа мы выбрали переменные «управляющего» сигнала, а выделение показывает какая переменная «поступает» на выход функции  $f$ :

$f(000000) = 0$	$f(010000) = 0$	$f(100000) = 0$	$f(110000) = 0$
$f(000001) = 0$	$f(010001) = 0$	$f(100001) = 0$	$f(110001) = 1$
$f(000010) = 0$	$f(010010) = 0$	$f(100010) = 1$	$f(110010) = 0$
$f(000011) = 0$	$f(010011) = 0$	$f(100011) = 1$	$f(110011) = 1$
$f(000100) = 0$	$f(010100) = 1$	$f(100100) = 0$	$f(110100) = 0$
$f(000101) = 0$	$f(010101) = 1$	$f(100101) = 0$	$f(110101) = 1$
$f(000110) = 0$	$f(010110) = 1$	$f(100110) = 1$	$f(110110) = 0$
$f(000111) = 0$	$f(010111) = 1$	$f(100111) = 1$	$f(110111) = 1$
$f(001000) = 1$	$f(011000) = 0$	$f(101000) = 0$	$f(111000) = 0$
$f(001001) = 1$	$f(011001) = 0$	$f(101001) = 0$	$f(111001) = 1$
$f(001010) = 1$	$f(011010) = 0$	$f(101010) = 1$	$f(111010) = 0$
$f(001011) = 1$	$f(011011) = 0$	$f(101011) = 1$	$f(111011) = 1$
$f(001100) = 1$	$f(011100) = 1$	$f(101100) = 0$	$f(111100) = 0$
$f(001101) = 1$	$f(011101) = 1$	$f(101101) = 0$	$f(111101) = 1$
$f(001110) = 1$	$f(011110) = 1$	$f(101110) = 1$	$f(111110) = 0$
$f(001111) = 1$	$f(011111) = 1$	$f(101111) = 1$	$f(111111) = 1$

## Пример задания лабораторной работы

Пример формируется...

# Лабораторная работа №3

## Регистры и счётчики

Функции цифровых устройств, естественно, не сводятся к реализации разнообразных ФАЛ. Нам хотелось бы использовать цифровые устройства для обработки информации, вычислений. Но для осуществления этих возможностей нам недостаёт элемента памяти, который мог бы хранить промежуточные результаты. Ведь невозможно сделать калькулятор, если нет возможности сохранить вводимые числа и результат вычисления.

Элемент памяти — один из самых важных элементов цифровых устройств. Чтобы не делать ошибок при разработке цифровых устройств, необходимо понять место этого узла, его идею и инструменты языка Verilog, связанные с ним.

Первый элемент памяти, который мы рассмотрим — это **защелка** (англ. latch).

Защелка является основой всех элементов памяти. Она состоит из двух элементов И-НЕ (или из двух элементов ИЛИ-НЕ, в зависимости от базиса, выбранного при проектировании), соединённых по следующей схеме:

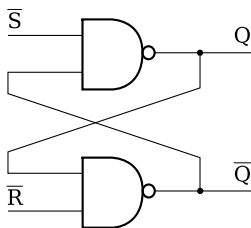


Рис. 2.1: Структура RS-защелки

У защелки два входа и два выхода. Входами являются сигналы «сброс» и «установка в единицу» или по-английски «reset»

и «set». В зависимости от элементов, из которых состоит защелка, полярность входных сигналов будет меняться. В базе И-НЕ сброс и установка происходят, когда соответственно сигналы R или S находятся в нуле, поэтому их обозначают как «не-сброс» и «не-установка», чтобы отразить этот факт. Выход защелки — это тот бит данных, который она хранит. Два выхода отличаются полярностью — один из них инвертирует хранимый бит. Ниже приведена таблица со всеми возможными комбинациями входных сигналов и временная диаграмма работы защелки.

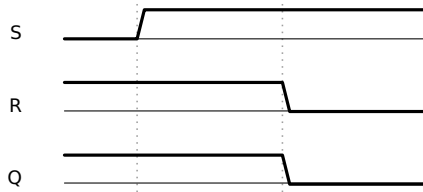


Рис. 2.2: Временная диаграмма работы RS-защелки

Опишем защелку на языке Verilog, опираясь на её структуру, которую мы рассмотрели выше. Нам понадобятся два входа, два выхода и два элемента И-НЕ, которые мы опишем с помощью операций И (оператор `&`) и НЕ (оператор `~`).

```

1  module latch_struct(
2  input nR,
3  input nS,
4  output Q,
5  output nQ);
6
7  assign Q = ~(nS & nQ);
8  assign nQ = ~(nR & Q);
9
10 endmodule;
```

Листинг 2.1: Описание RS-защелки на языке Verilog HDL

Элемент памяти нам, прежде всего, нужен для хранения данных. Для того, чтобы защелкой стало удобнее пользоваться, немного изменим схему подключения управляющих сигнала-

ЛОВ.

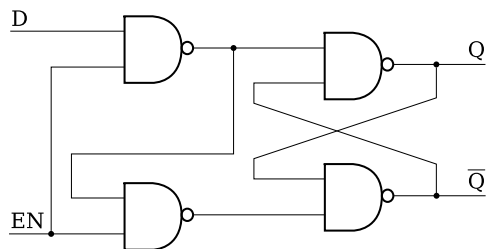


Рис. 2.3: Структура D-защелки

Защелка теперь будет работать следующим образом: при высоком уровне на входе «разрешить работу» («enable») данные со входа «данные» («data») будут проходить через защелку на выход, при низком уровне на входе «разрешить работу» защелка будет сохранять на выходе последнее значение со входа «данные», которое было до переключения сигнала «разрешить работу». Работа такой защелки показана на временной диаграмме ниже.

Как мы уже говорили, использовать структурные описания не всегда удобно. В большинстве случаев использовать поведенческое описание намного эффективнее. Поведенческое описание часто формулируется гораздо лаконичнее, и, так как его легче понять человеку, улучшается читаемость кода и уменьшается вероятность ошибок при его написании.

```
1 module d-latch_behav(  
2   input d,  
3   input en,  
4   output reg q);  
5  
6   always @(en, d) begin  
7       if (en) q <= d;  
8   end  
9  
10  endmodule;
```

Листинг 2.2: Поведенческое описание D-защелки на языке Verilog HDL



Если добавить к этой схеме еще две защелки, то можно привязать изменение «содержимого» защелки к переходу управляющего сигнала из «0» в «1», то получим следующую структуру:

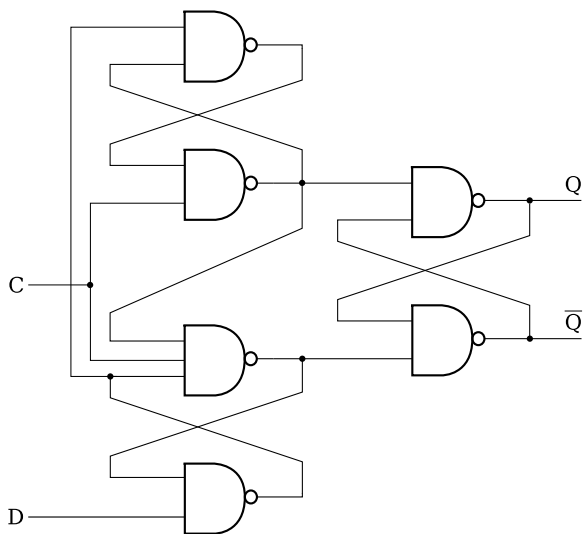


Рис. 2.4: Структура D-триггера

Эту схему можно немного доработать, введя управляющие сигналы сброса, установки в единицу и разрешения работы. Упрощенно такая схема изображается следующим образом.

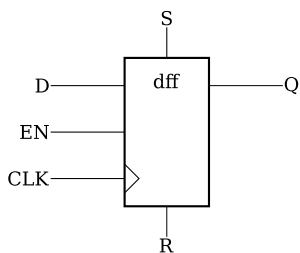


Рис. 2.5: Графическое обозначение D-триггера

Эта схема получила широчайшее применение в цифровой схемотехнике и называется d-триггер (от слова «data» — данные). Ниже приведена временная диаграмма работы d-

триггера.

Заметим, что сигнал *C* называют «тактирующим» сигналом или «сигналом синхронизации». Обычно в роли этого сигнала выступает сигнал от внешнего источника (чаще всего кварцевого резонатора) со стабильной частотой. А сами цифровые устройства, для работы которых необходим сигнал синхронизации, называют синхронными.

Сигнал синхронизации играет очень большую роль в цифровых устройствах. Прежде всего, он необходим для того, чтобы избежать непредсказуемого и нестабильного поведения триггеров в цифровых устройствах.

```
1 module d-flipflop_behav(  
2   input d,  
3   input clk,  
4   input rst,  
5   input en,  
6   output reg q);  
7  
8   always @(posedge clk or posedge rst) begin  
9       if (rst) q <= 0;  
10      else if (en) q <= d;  
11  end  
12  
13 endmodule;
```

Листинг 2.3: Описание D-триггера на языке Verilog HDL

В описании появилось новое ключевое слово `posedge`. Оно используется только в списке чувствительности блока `always` и означает событие перехода сигнала, имя которого стоит после этого ключевого слова, из состояния «0» в состояние «1».

Ключевое слово `posedge` было введено прежде всего для того, чтобы описывать схемы, содержащие триггеры. Ведь триггеры, как мы уже говорили, могут менять своё состояние только в момент положительного фронта (англ. *positive edge*) сигнала синхронизации.

Добавление в список чувствительности события `posedge rst` позволяет описать поведение триггера в момент асинхронного сброса: как только случается переход `rst` из «0» в «1» срабаты-

вает блок `always` и проверка условия `if (rst)` дает положительный результат, триггер сбрасывается в «0».

Если объединить несколько триггеров в группу, то получится то, что в цифровой схемотехнике называют «регистр».

```
1 module register_behav(  
2   input [7:0] d,  
3   input clk,  
4   input rst,  
5   input en,  
6   output reg [7:0] q);  
7  
8   always @(posedge clk or posedge rst) begin  
9       if (rst) q <= 0;  
10      else if (en) q <= d;  
11   end  
12  
13 endmodule;
```

Листинг 2.4: Описание регистра на языке Verilog HDL

Элементы памяти позволяют нам сохранять информацию для дальнейшей обработки или хранить готовый результат вычисления, хранить промежуточные результаты.

Запомните описание регистра. Оно используется при проектировании практически любого цифрового устройства с помощью Verilog.

Необходимо отметить важную концепцию языка Verilog. **Переменные типа `reg` могут быть изменены только в пределах одного блока `always`. Переменные доступны для проверки в любом из блоков, но изменять их значение можно только в одном из них.**

```
1 reg a;  
2 reg b;  
3  
4 always @(posedge clk) begin  
5     if (in < 5) a <= in;  
6 end  
7
```

```

8  always @(posedge clk) begin
9      if (n > 5) begin
10         b <= in;
11         a <= in - 5; //ошибка!!!
12     end
13     else b <= a;
14 end

```

Листинг 2.5: Пример присвоения значения переменной а разных блоках always на языке Verilog HDL

Одной из простейших, и в тоже время широко распространённой, цифровой схемой на основе регистров является счётчик.

Счётчик считает количество тактов, которое прошло с момента его обнуления.

Такая простая схема, тем не менее, используется практически в каждом цифровом устройстве. Как будет показано дальше, счётчик легко можно доработать таким образом, чтобы отсчитывались не такты, а какие-то события. Например, событиями могут быть: нажатие кнопки, принятие пакета данных, срабатывание датчика, выполнение какого-то условия (периодическое) и другое.

Итак, для того чтобы реализовать счетчик нам понадобится регистр и сумматор. Причем сумматор будет складывать значение, хранящееся в регистре с константой (в нашем случае единицей) а результат сложения будет поступать на вход регистра.

В результате получим следующую схему:

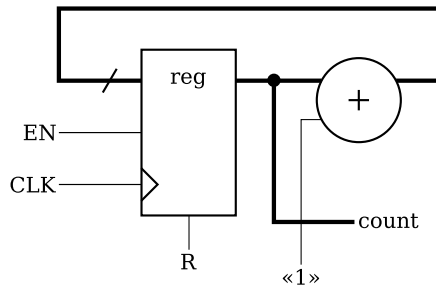


Рис. 2.6: Структура восьмибитного счётчика

На временной диаграмме ниже хорошо видно как работает счётчик:

Опишем поведение такого счётчика на Verilog.

```

1  module counter_8bit(
2  input clk,
3  input en,
4  input rst,
5  output reg [7:0] counter);
6
7  always @(posedge clk or posedge rst) begin
8      if (rst) counter <= 0;
9      else if (en) counter <= counter + 1;
10 end
11
12 endmodule;
```

Листинг 2.6: Описание восьмибитного счётчика на языке Verilog HDL

Для того чтобы можно было подсчитывать события, а не переходы сигнала синхронизации из «0» в «1» понадобится ввести ещё одну схему. Её смысл и назначение заключается в следующем: нам необходимо из асинхронного события получить синхронный сигнал единичной длительности. Тогда, подавая такой сигнал на вход enable счётчика, мы сможем считать количество произошедших событий.

Ниже представлена схема, позволяющая сделать это:

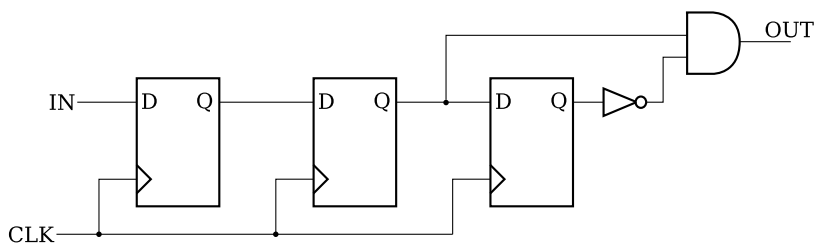


Рис. 2.7: Структура регистра на D-триггерах

Пример работы такой схемы:

Естественно такая схема работает только тогда, когда входной сигнал изменяется с частотой меньшей, чем частота синхронизации.

Сигнал out в таком случае подключается к входу enable счётчика.

## 2.1 Задание лабораторной работы

Описать на языке Verilog цифровое устройство, функционирующее согласно следующим принципам:

1. Ввод информации происходит с переключателей SW[9:0] и кнопок KEY[0], KEY[1]. Внешний источник сигнала синхронизации: CLK50;
2. KEY[1] должна функционировать как общий асинхронный сброс устройства;
3. При нажатии на KEY[0] записывать данные с SW[9:0] в десятиразрядный регистр;
4. Содержимое десятиразрядного регистра выводить на LEDR[9:0];
5. При нажатии на KEY[0] увеличивать 8-ми разрядный счётчик нажатий на 1, если произошло событие, указанное в индивидуальном задании студента;
6. Содержимое счётчика выводить в шестнадцатеричной форме на HEX0 и HEX1 (цифры с 0 до 9 и буквы A, B, C, D, E, F)

Выполнив описание модуля на языке Verilog необходимо построить временные диаграммы его работы с помощью САПР

Altera Quartus.

Привязать входы модуля к переключателям SW, отладочной платы, а выход к шине HEX0[6:0], получить прошивку для ПЛИС и продемонстрировать её работу.

## 2.2 Пример индивидуального задания

Событием является наличие 3 и более единиц на SW[9:0] в момент записи в регистр.

```
1  reg sw_event;  
2  always @(SW) begin  
3      if ((SW[0] + SW[1] + SW[2] + SW[3]  
4          + SW[4] + SW[5] + SW[6] + SW[7]  
5          + SW[8] + SW[9]) > '4d3) sw_event <= '1b1;  
6      else sw_event <= '1b0;  
7  end  
8  
9  reg [2:0] event_sync_reg;  
10 wire synced_event;  
11 assign synced_event = event_sync_reg[1]  
12                        & ~event_sync_reg[0];  
13  
14 always @(posedge CLK50) begin  
15     event_sync_reg[2] <= sw_event;  
16     event_sync_reg[1:0] <= event_sync_reg[2:1];  
17 end
```

Листинг 2.7: Решение индивидуального задания  
(фрагмент кода лабораторной работы)

## 2.3 Вопросы к защите лабораторной работы

1. Какие элементы памяти вы изучили в данной лабораторной работе?

2. Чем отличается RS-защелка от D-защелки?
3. Какие входы могут быть у триггера? Перечислите все и назовите их функции.
4. Какие блоки вашего цифрового устройства синхронные? Какие нет? Почему?
5. Какой фрагмент вашего кода описывает вывод значения счетчика на семи сегментный индикатор? Как называется эта цифровая схема?
6. Продемонстрируйте код реализующий индивидуальное задание.
7. Покажите в коде лабораторной код счётчика.
8. Что такое сигнал синхронизации?



# Лабораторная работа №6

## FLASH память

### 3.1 Виды энергонезависимой памяти

Ни один из блоков цифровых устройств, которые мы рассмотрели ранее не способен хранить информацию при отсутствии питания.

Чтобы решить эту проблему, на заре вычислительной техники, данные в цифровое устройство после подачи питания загружали с таких носителей, как перфокарты и, позже, магнитные ленты. Ещё позже для целей хранения информации при отсутствии питания были разработаны накопители на гибких магнитных дисках — дискетах и жёстких магнитных дисках — «HDD». На данный момент для хранения данных при отсутствии питания наиболее широко применяется FLASH-память.

Энергонезависимые накопители информации обладают как преимуществами, так и недостатками по сравнению с энергозависимой RAM-памяти.

Как правило, энергонезависимая память существенно уступает по скорости работы RAM-памяти. Это ограничение удалось преодолеть только недавно: в 2016 году была представлена постоянная память, где информация хранится в виде спина электрона. Такая память по скорости работы не уступает современной RAM-памяти, такой как DDR5. Но подобная память ещё долгое время будет оставаться недоступной для рядового пользователя из-за высокой стоимости и малой степени освоения технологии серийного производства 10нм.

## 3.2 Принципы работы FLASH-памяти

В качестве элемента хранения информации FLASH-память использует транзистор с плавающим затвором, где состояние затвора определяет бит хранимой информации, изображённый на Рис.??

Рассмотрим работу такого транзистора.

Как вы видите, он содержит два затвора: управляющий и плавающий. Плавающий полностью находится в диэлектрике и при этом способен накапливать электроны. От величины накопленного заряда меняется «лёгкость» с которой транзистор открывается — т.е. величина напряжения «управляющий затвор—исток», при которой через транзистор начнёт течь ток.

Для хранения информации используют следующий принцип (см.Рис.??): чтобы считать информацию, на управляющий затвор подаётся напряжение чтения — среднее между самым сильным (в диэлектрике нет электронов) и самым слабым (в диэлектрике максимум электронов). Если транзистор открывается, значит в плавающем затворе были электроны и мы считаем, что в нём записан «0», если не открывается, значит электронов в плавающем затворе нет и записана «1».

Осталось понять как можно «заставить» электроны попадать в плавающий затвор, ведь он изолирован диэлектриком. Не вдаваясь в подробности скажем, что если подать достаточно высокое напряжение «управляющий затвор—сток», то у электронов хватит энергии, чтобы «перескочить» диэлектрик и попасть в плавающий затвор. А если изменить полярность этого напряжения, то можно заставить электроны покинуть плавающий затвор (см.Рис.??).

Самое важное в этой идее то, что если электроны попали в плавающий затвор они не могут самостоятельно покинуть его через диэлектрик и будут оставаться там в течении многих лет. Таким образом и достигается сохранение записанной информации при отсутствии питания.

Теперь мы знаем, что для того чтобы записать или считать информацию из FLASH-памяти надо использовать большую разность потенциалов. Но на самом деле транзистор устро-

ен таким образом, что энергия, необходимая чтобы «загнать» электроны в плавающий затвор меньше энергии, необходимой, чтобы их «выгнать». Это делается чтобы при чтении значения электроны не покидали плавающий затвор.

При такой организации становится сложно обеспечить очистку каждого транзистора в отдельности, поэтому обычно стирается целая группа ячеек.

Из-за особенностей транзистора с плавающим затвором, которые мы рассмотрели можно выделить следующие характерные черты FLASH-памяти:

- Запись значения возможна только из логической «1» в логический «0»;
- Удаление информации возможно только из группы ячеек одновременно (блока);
- Удаление и запись информации приводят к деградации ячеек памяти;
- Чтение также приводит к деградации ячеек памяти, но в меньшей степени.

На Рисунке?? изображена общая структура FLASH-памяти. Как видно она практически не отличается от RAM-памяти: из ячеек строится матрица, контролируемая управляющим блоком. А сам управляющий блок обеспечивает коммуникацию с внешними устройствами, дешифрацию адреса и управление записью и чтением массива элементов памяти. Подключение FLASH-памяти и управление ей со стороны цифрового устройства полностью зависит от того, как реализован блок управления — доступ к содержимому FLASH-памяти может быть синхронный или асинхронный, по последовательной или параллельной шине, с разделением шин адреса и данных или без него.

### 3.3 Чип FLASH-памяти S29AL032D

Для практического знакомства с FLASH-памятью мы спроектируем контроллер микросхемы S29AL032D. Именно эта микросхема установлена на отладочной плате Altera DE1.

Основным источником информации о любой микросхеме

служат технические условия (англ. datasheet). В большинстве случаев в этом документе содержатся все необходимые сведения для использования микросхемы: электрические параметры, размеры и тип корпуса, информация о выводах, и многие другие сведения. В том числе datasheet содержит данные о протоколах информационного обмена.

В нашем случае микросхема уже подключена, поэтому из всего datasheet нас прежде всего интересует каким образом необходимо взаимодействовать с данной микросхемой, чтобы записать или считать данные.

Для разработки контроллера следует ознакомиться со следующими разделами документа:

- 11. Commands Definitions;
- 12. Write Operation Status;
- 17. AC Characteristics.

Далее будут приведены необходимые выдержки из документа, однако настоятельно рекомендуем ознакомиться с ним.

### 3.3.1 Операция чтения

Для чтения данных из чипа S29AL032D не требуется никакой дополнительной подготовки. Временная диаграмма чтения приведена в пункте 17.2 datasheet и представлена на Рис.??

Времена, указанные на диаграмме, приведены в Табл. 3.1

Обозн.	Описание	Min	Max
$t_{RC}$	Продолжительность цикла чтения	70нс	90нс
$t_{ACC}$	Задержка Адрес — Данные	70нс	90нс
$t_{CE}$	Задержка Выбор Чипа — Данные	70нс	90нс

Таблица 3.1: Временные характеристики операции чтения S29AL032D

Как мы уже знаем, единственным источником информации о времени для цифрового устройства может являться только сигнал синхронизации, заранее известной частоты. Приведем времена, упомянутые в Таблице 3.1 к тактовому сигналу частоты 50 МГц. Полученная временная диаграмма показана

на Рис. ??

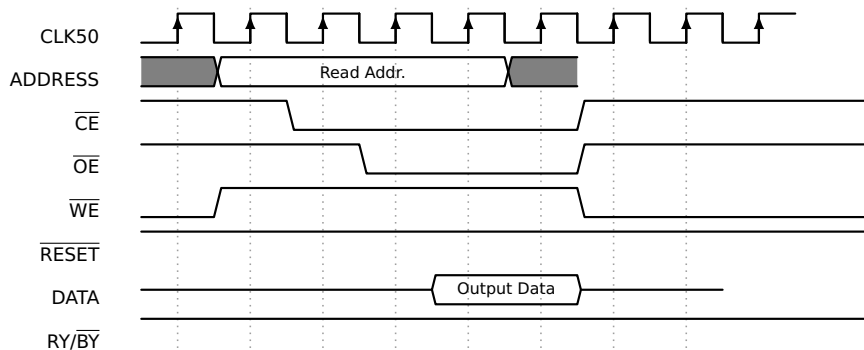


Рис. 3.1: Временная диаграмма операции чтения шины PCI

### 3.3.2 Операция записи

### 3.3.3 Операция очистки сектора

### 3.3.4 Проектирование контроллера S29AL032D