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 помощью 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 — одни из наиболее распространённых на текущий момент. И начнём мы с разработки наиболее простых цифровых устройств — логических вентилей.

1.2 HDL описания логических вентилей

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	$a \mid b$
0	0	0
0	1	1
1	0	1
1	1	1

Таблица 1.2: ИЛИ

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

Таблица 1.3: Исключающее 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.

Схемотично изобразим этот модуль:

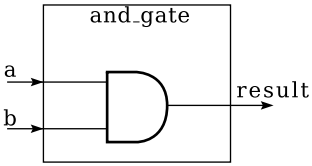

Рис. 1.1: Структура модуля «and_gate»

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 вентилей

Условие будет выполняться либо когда *не* выполнено условие *«c»*, либо когда одновременно выполняются условия *«a»* 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

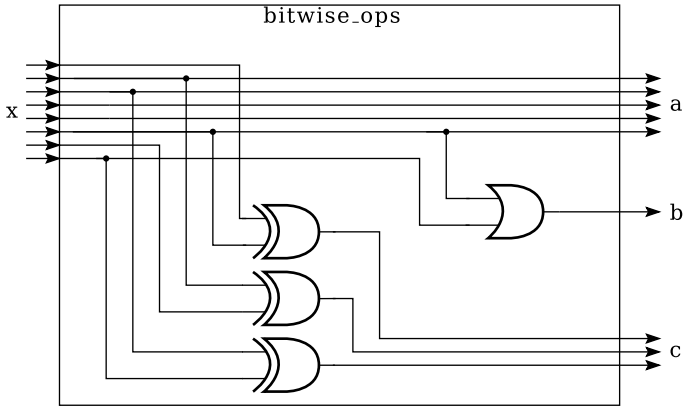

Рис. 1.2: Структура модуля «bitwise_ops»

Впрочем, реализация ФАЛ с помощью логических вентилей не всегда представляется удобной. Допустим нам нужно описать таблично-заданную ФАЛ. Тогда описания этой функции при помощи логических вентилей 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.

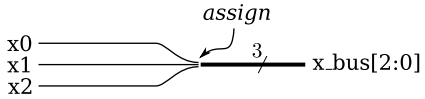

Рис. 1.3: Действие оператора **assign**

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

Verilog HDL описывает цифровую аппаратуру, которая су-

ществует 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 назвали 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 выолнения 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 назвать блоками, или частями будущего цифрового устройтсва.

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:

$$y_0 = f(x_2, x_1, x_0);$$
$$y_1 = f(x_2, x_1, x_0);$$
$$y_2 = f(x_2, x_1, x_0);$$
$$y_3 = f(x_2, x_1, x_0).$$

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.

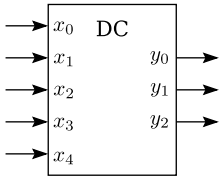

Рис. 1.4: Графическое обозначение дешифратора

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

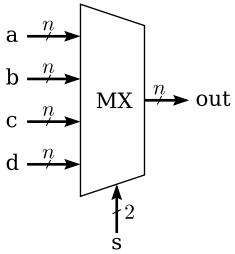

Рис. 1.5: Графическое обозначение мультиплексора

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:

$$
\begin{array}{llll}
f(00\mathbf{0}000) = 0 & f(010\mathbf{0}00) = 0 & f(1000\mathbf{0}0) = 0 & f(11000\mathbf{0}) = 0 \\
f(00\mathbf{0}001) = 0 & f(010\mathbf{0}01) = 0 & f(1000\mathbf{0}1) = 0 & f(11000\mathbf{1}) = 1 \\
f(00\mathbf{0}010) = 0 & f(010\mathbf{0}10) = 0 & f(1000\mathbf{1}0) = 1 & f(11001\mathbf{0}) = 0 \\
f(00\mathbf{0}011) = 0 & f(010\mathbf{0}11) = 0 & f(1000\mathbf{1}1) = 1 & f(11001\mathbf{1}) = 1 \\
f(00\mathbf{0}100) = 0 & f(010\mathbf{1}00) = 1 & f(1001\mathbf{0}0) = 0 & f(11010\mathbf{0}) = 0 \\
f(00\mathbf{0}101) = 0 & f(010\mathbf{1}01) = 1 & f(1001\mathbf{0}1) = 0 & f(11010\mathbf{1}) = 1 \\
f(00\mathbf{0}110) = 0 & f(010\mathbf{1}10) = 1 & f(1001\mathbf{1}0) = 1 & f(11011\mathbf{0}) = 0 \\
f(00\mathbf{0}111) = 0 & f(010\mathbf{1}11) = 1 & f(1001\mathbf{1}1) = 1 & f(11011\mathbf{1}) = 1 \\
f(00\mathbf{1}000) = 1 & f(011\mathbf{0}00) = 0 & f(1010\mathbf{0}0) = 0 & f(11100\mathbf{0}) = 0 \\
f(00\mathbf{1}001) = 1 & f(011\mathbf{0}01) = 0 & f(1010\mathbf{0}1) = 0 & f(11100\mathbf{1}) = 1 \\
f(00\mathbf{1}010) = 1 & f(011\mathbf{0}10) = 0 & f(1010\mathbf{1}0) = 1 & f(11101\mathbf{0}) = 0 \\
f(00\mathbf{1}011) = 1 & f(011\mathbf{0}11) = 0 & f(1010\mathbf{1}1) = 1 & f(11101\mathbf{1}) = 1 \\
f(00\mathbf{1}100) = 1 & f(011\mathbf{1}00) = 1 & f(1011\mathbf{0}0) = 0 & f(11110\mathbf{0}) = 0 \\
f(00\mathbf{1}101) = 1 & f(011\mathbf{1}01) = 1 & f(1011\mathbf{0}1) = 0 & f(11110\mathbf{1}) = 1 \\
f(00\mathbf{1}110) = 1 & f(011\mathbf{1}10) = 1 & f(1011\mathbf{1}0) = 1 & f(11111\mathbf{0}) = 0 \\
f(00\mathbf{1}111) = 1 & f(011\mathbf{1}11) = 1 & f(1011\mathbf{1}1) = 1 & f(11111\mathbf{1}) = 1
\end{array}
$$

Лабораторная работа №2
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), соединенных по следующей схеме:

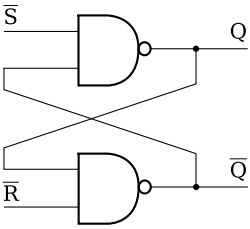

Рис. 2.1: Структура RS-защелки

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 базисе 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.

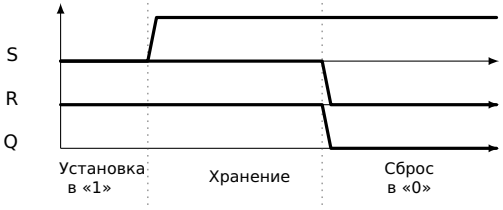

Рис. 2.2: Временная диаграмма работы RS-защелки

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 сигна-

лов.

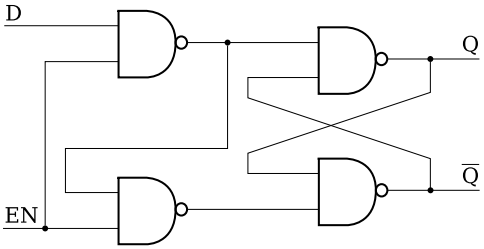

Рис. 2.3: Структура D-защелки

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.

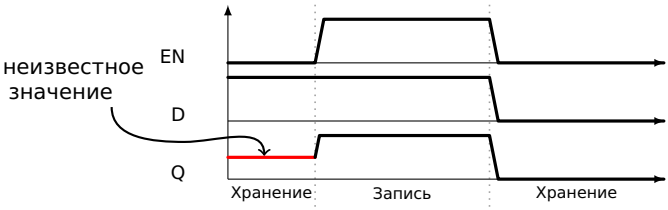

Рис. 2.4: Временная диаграмма работы D-защелки

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_beahv(
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:

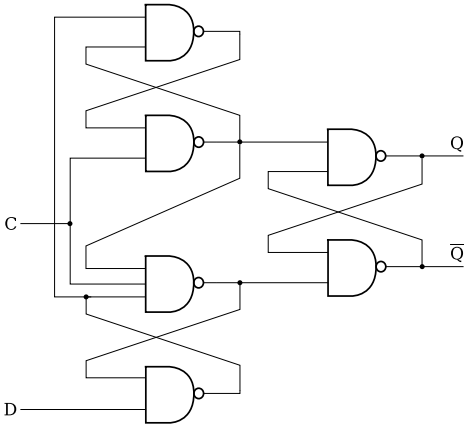

Рис. 2.5: Структура D-триггера

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

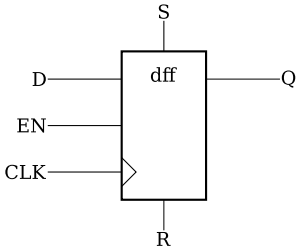

Рис. 2.6: Графическое обозначение D-триггера

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

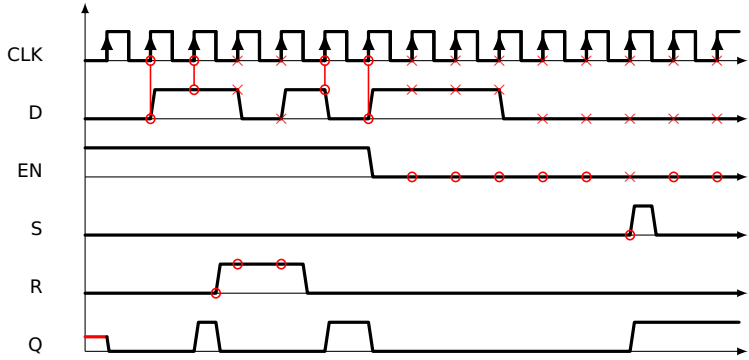

Рис. 2.7: Пример работы регистра

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_beahv(
 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» срабатывает 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 a разных блоках always на языке Verilog HDL

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

чик.

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:

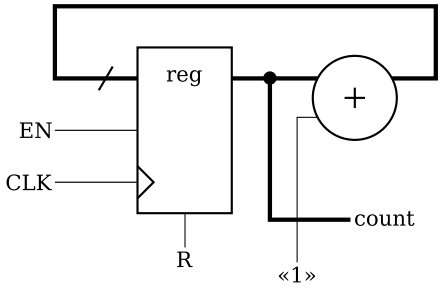

Рис. 2.8: Структура восьмибитного счетчика

На временной диаграмме ниже хорошо видно как работает счётчик:

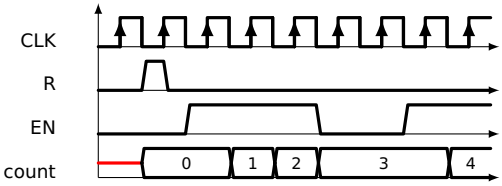

Рис. 2.9: Пример работы регистра

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 восьмиитного счетчика на языке Verilog HDL

Для того чтобы можно было подсчитывать события, а не переходы сигнала синхронизации из «0» в «1» понадобится ввести еще 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:

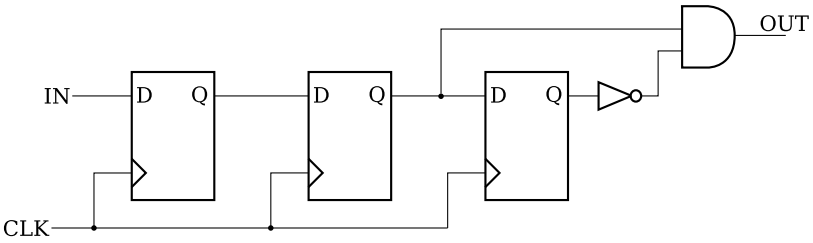

Рис. 2.10: Структура схемы синхронизации сигнала

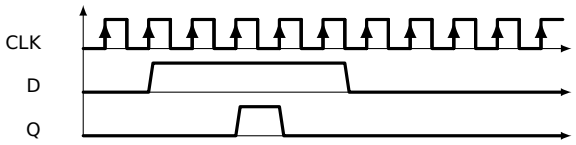

Рис. 2.11: Пример поведения схемы

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.

```
reg sw_event;
always @(SW) begin
    if ((SW[0] + SW[1] + SW[2] + SW[3]
       + SW[4] + SW[5] + SW[6] + SW[7]
       + SW[8] + SW[9]) > 4'd3) sw_event <= 1'b1;
    else sw_event <= 1'b0;
end

reg [2:0] event_sync_reg;
wire synced_event;
assign synced_event = event_sync_reg[1]
                    & ~event_sync_reg[0];

always @(posedge CLK50) begin
   event_sync_reg[2] <= sw_event;
   event_sync_reg[1:0] <= event_sync_reg[2:1];
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 №4 Секундомер

В прошлых лабораторных работах мы изучили базовые строительные блоки цифровых устройств. Теперь у нас уже достаточно знаний для реализации несложного, но функционально законченного цифрового устройства.

В данной лабораторной работе мы познакомимся с процессом проектирования полноценного цифрового устройства на примере разработки простого секундомера. Мы подробно, поэтапно, рассмотрим процесс проектирования, проиллюстрировав каждый этап графической схемой.

Для эффективного проектирования любого цифрового устройства нужно придерживаться некоторой «канвы» проектирования. Это поможет не запутаться и последовательно разобраться с вопросами, возникающими в ходе проектирования.

Начинать проектирование любого цифрового устройства следует с определения входов и выходов. Нужно понять какие данные будут входными для проектируемого устройства, и какие данные нам надо выработать и подать на выход.

В случае секундомера справедливы такие рассуждения:

Чтобы управлять работой секундомера нам понадобятся два входа: «старт/стоп» и «сброс».

Для отображения времени можно воспользоваться семисегментными индикаторами. Значит, для управления каждым из них понадобится семибитная шина, которая будет выходом нашего устройства.

Для отображения времени выделим 2 индикатора для отображения количества прошедших секунд и 2 индикатора для

отображения количества прошедших десятых и сотых долей секунды.

Значит выходом секундомера будут четыре семибитные шины для управления индикаторами.

В основе секундомера лежит счётчик. Работая, секундомер отсчитывает время, считая количество пришедших импульсов сигнала синхронизации, частота которого заранее известна.

Т.е. нам потребуется сигнал синхронизации со стабильной частотой.

Больше никаких входов и выходов не требуется.

Общая схема на данный момент выглядит так:

———picture———

Начнём описывать модуль на языке Verilog:

```
1  module stopwatch (
2  input start_stop,
3  input reset,
4  input clk,
5  output [6:0] hex0,
6  output [6:0] hex1,
7  output [6:0] hex2,
8  output [6:0] hex3);
9
10 endmodule;
```

Листинг 3.1: Описание входов и выходов модуля на языке Verilog HDL

Теперь приступим к описанию «внутренностей» модуля.

Чтобы реализовать секундомер, нам необходимо отсчитывать время.

Для отсчёта времени в цифровых устройствах считают количество прошедших импульсов синхронизации (тактов). Так как тактовые импульсы генерируются кварцевым генератором со стабильной, известной нам, частотой, то мы можем рассчитать количество импульсов, которое соответствует заданному времени.

Например, если в устройстве установлен кварцевый гене-

ратор на 26 МГц, то одной секунде соответствует 26 миллионов тактовых импульсов, а одной сотой секунды соответствует 260 тысяч тактовых импульсов.

Для того, чтобы отсчитать это количество импульсов подходит единственный из известных нам «строительных блоков» - счётчик:

———-picture———-

Как мы уже говорили, счётчик состоит из регистра и сумматора. Чтобы счётчик циклически отсчитывал одну сотую секунды его необходимо обнулить после того, как он отсчитает 260 тысяч тактовых импульсов. В этот же момент нужно выработать сигнал для остальной схемы, что прошла одна сотая секунды.

Из всех цифровых блоков, которые мы рассмотрели, для реализации задачи сравнения текущего значения счётчика с константой подходит только компаратор. На один из входов компаратора подадим текущее значение счётчика, а на другой вход - константу 260 000.

———-picture———-

Пока значения на входах компаратора будут отличаться, на выходе компаратора будет значение «0». Когда значения будут равны, компаратор изменит выход с «0» на «1», это и будет признак того, что прошло 0.01 секунды. Для того, чтобы можно было эффективно использовать сигнал «прошло 0.01с», этот сигнал должен иметь длительность равную 1 такту.

Этот же сигнал мы будет использовать для управления сбросом счётчика.

Итак, счётчик должен после достижения значения 260 000 принять значение «0», но переход должен случиться, как и все остальные переходы, в момент перехода тактового сигнала из «0» в «1».

Сброс, отвечающий таким условиям, называется «синхронный сброс».

Посмотрите, как будет выглядеть на временной диаграмме как будет работать счётчик если выход компаратора, подключить как сигнал синхронного сброса:

———-timing table———-

А так выглядит временная диаграмма, если подключить вы-

ход компаратора к входу асинхронного сброса триггера:

———–timing table———–

Выход компаратора, установившись в единицу, моментально сбросит счётчик и, так как значение счётчика изменилось, а значит, изменился и один из входов компаратора, выход компаратора сразу же перейдет в значение «0».

Обратите внимание, что длительность сигнала с выхода компаратора должна быть равна одному такту. Ведь в дальнейшем нам необходимо будет считать события «прошла одна сотая секунды», а значит подготовить сигнал единичной длительности, который соответствует этому событию (см. лабораторную работу №3).

Сигнал с компаратора, в случае, когда он подключен в виде синхронного сброса, полностью удовлетворяет этому условию, а значит нам не придется в дальнейшем вводить новые фрагменты схемы.

Как реализовать синхронный сброс в цифровом устройстве?

Для этого можно использовать мультиплексор. Схема будет выглядеть следующим образом:

———-picture———

Если подключить sync_reset к выходу компаратора, то когда счётчик достигнет порогового значения, выход компаратора изменится и переключит мультиплексор. Теперь на выход мультиплексора будет подаваться «0». Этот сигнал будет поступать на вход триггера, но запись нового значения произойдет только во время положительного фронта сигнала синхронизации.

Для общего сброса секундомера при нажатии кнопки «сброс» как раз можно воспользоваться входом асинхронного сброса регистра. Ведь при нажатии кнопки «сброс» можно обнулять регистр мгновенно.

Теперь надо выбрать правильный сигнал управления работой счётчика - сигнал разрешения работы (Enable, EN). Ведь счётчик должен начинать считать после нажатия кнопки «старт/стоп», а после её повторного нажатия должен останавливаться.

Для управления работой счётчика можно использовать сигнал, который будет единицей, пока счётчик должен работать и

нулём, если отсчёт времени остановлен. Как раз такой сигнал можно подать на вход разрешения работы регистра. Назовём этот сигнал «device_running».

Посмотрите, как выглядит остановка и запуск счётчика в таком случае:

————-timing table————–

К проектированию и описанию схемы, которая вырабатывала бы сигнал «device_running», мы вернемся позднее.

Стоит обратить внимание на следующий момент: что будет, если счётчик остановить в тот момент времени, когда его значение стало равно 259999?

Взгляните на временную диаграмму:

————-timing table—————

Для того чтобы не допустить такого поведения, можно немного изменить условие, запрещающее работу счётчика. Теперь мы будем дополнительно проверять сигнал с компаратора. И если счётчик в данный момент равен 259999, то запретить его работу будет невозможно.

Для решения этой задачи подойдет вентиль «или». Условие будет таким: «работа разрешена, если сигнал «device_running» равен единице **ИЛИ** когда текущее значение счётчика равно 259999».

Теперь схема счётчика выглядит следующим образом:

—————picture—————

Когда мы представили схему в виде набора цифровых блоков, мы можем описать её поведение на языке Verilog:

```
1  //регистр счётчика
2  reg [16:0] pulse_counter;
3
4  //описание компаратора
5  wire hundredth_of_second_passed =
6                (pulse_counter == 17'd259999);
7
8  //описание счётчика
9  always @(posedge clk or posedge reset) begin
10    if (reset) pulse_counter <= 0;
11        //асинхронный сброс
```

```
12
13    //сигнал разрешения работы счётчика
14    else if (device_running |
15                hundredth_of_second_passed)
16
17        //синхронный сброс
18        if (hundredth_of_second_passed)
19                pulse_counter <= 0;
20
21        //увеличение счётчика на единицу
22        else pulse_counter <= pulse_counter + 1;
23  end;
```

Листинг 3.2: Описание счетчика тактовых импульсов на языке Verilog HDL

Теперь, когда у нас есть счетчик, отсчитывающий такты и сигнализирующий о том, что прошла сотая доля секунды, мы можем отсчитывать сотые доли секунды.

Перед нами встаёт выбор.

Первый вариант – отсчитывать количество прошедших сотых долей секунды единственным счётчиком. Значение этого счётчика мы можем дешифрировать, чтобы выделить из него количество единиц, десятков, сотен и тысяч прошедших долей секунды, чтобы подать эти значения на дешифраторы семисегментных индикаторов:

————picture————

Второй вариант – использовать отдельные счётчики для сотых долей секунды, десятых долей секунды, целых секунд и десятков секунд.

Т.е. первый счетчик подсчитывает количество прошедших сотых долей секунды от 0 до 9, и, затем обнуляется, вырабатывая сигнал «прошла десятая доля секунды». Следующий счётчик, точно также считает уже десятые доли и вырабатывает сигнал «прошла одна секунда» и так далее.

————picture————-

Второй вариант для нас проще в реализации, компактнее, удобнее и понятнее.

Поэтому выберем именно его.

В качестве счётчиков подойдет уже описанная нами схема для подсчёта тактов, но с небольшими правками.

Счетчики подойдут нам потому, что функция их идентична – подсчёт событий с ограничением диапазона. Изменить нужно будет только разрядность счётчика с 16 на 4 и верхнюю границу счёта с 260000 на 9. Тогда счётчик будет выдавать признак переполнения (достижения границы отсчёта, когда его значение будет становиться) девяткой.

Еще одним моментом, о котором нужно позаботиться – длительность выходного сигнала.

Пока счётчик считал такты, его значение менялось каждый такт. Компаратор просто не мог принять значение 1 более чем на один такт. Теперь ситуация выглядит следующим образом:

————–timing table————-

Счетчик будет переключаться каждую 0,01 секунды, 0,1 секунды ,1 секунду или 10 секунд. И выход компаратора будет устанавливаться в 1 на всё время, которое потребуется для переключения счётчика из 9 в ноль. Т.е. в случае счётчика сотых долей секунды потребуется 259999 тактов.

Как выделить из всего времени, пока счётчик имеет значение «9» сигнал длительностью в один такт, который возникает в нужный момент времени? На временной диаграмме этот сигнал отмечен как «прошла 0,1с.»

Сигнал «прошла 0,1с» можно получить из сигналов, представленных на временной диаграмме следующим образом: «прошла 0,1с» правда, когда выход компаратора равен единице **И** «прошла 0,01с».

Схема счётчика практически не изменилась:

————————picture————————

Скорректируем описание её работы на Verilog:

```
1  //регистр счётчика
2  reg [3:0] hundredth_counter;
3
4  //описание компаратора
5  wire tenth_of_second_passed =
6         ((hundredths_counter == 4'd9) &
7           hundredths_of_second_passed);
```

```
 8
 9 //описание счётчика
10 always @(posedge clk or posedge reset) begin
11    if (reset) hundredths_counter <= 0;
12        //асинхронный сброс
13
14    //сигнал разрешения работы счётчика
15    else if (hundredth_of_second_passed)
16
17        //синхронный сброс
18        if (tenth_of_second_passed)
19               hundredths_counter <= 0;
20
21        //увеличение счётчика на единицу
22        else hundredths_counter <=
23               hundredths_counter + 1;
24 end;
```

Листинг 3.3: Описание счетчика сотых долей секунды на языке Verilog HDL

Счётчики десятых долей секунды, целых секунд и десятков секунд устроены абсолютно также. В описаниях изменятся только названия сигналов и регистров. Единственное в чем необходимо быть внимательным – это подключение сигналов. Для правильного подключения надо свериться со схемой, которую мы выбрали ранее.

Теперь вернёмся к вопросам, которые мы отложили ранее.

В нашем устройстве пока нет описания схемы, которая вырабатывает сигнал «device_stopped». Сигнал должен управляться кнопкой, поэтому как мы уже говорили, в лабораторной работе №3 потребуется схема, синхронизирующая сигнал, поступающий с кнопки с внутренним сигналом clk (тактовые импульсы).

Также сразу выделим из всего нажатия признак того, что кнопка была нажата, так, чтобы по длительности этот признак был равен одному такту.

Тогда схема будет абсолютно такой же, как и в лабораторной работе №3 и будет выглядеть следующим образом:

————————picture————————

Поведение такой схемы описывается на языке Verilog следующим образом:

```
1  reg [2:0] button_syncroniser;
2        wire button_was_pressed;
3
4  always @(posedge clk) begin
5     button_syncroniser[0] <= in;
6     button_syncroniser[1] <= button_syncroniser[0];
7     button_syncroniser[2] <= button_syncroniser[1];
8  end;
9
10 assign button_was_pressed <= ~button_syncroniser[2]
11                              & button_syncroniser[1];
```

Листинг 3.4: Описание схемы синхранизации на языке Verilog HDL

Эта схема и её описание подробно рассмотрены в лабораторной работе №3

Теперь нам нужно построить схему, которая по нажатию кнопки переключала бы сигнал «device_stopped» из «0» в «1» и из «1» в «0».

Что нам понадобится? Триггер, чтобы хранить значение «device_stopped». Чтобы менять значение на противоположное надо знать противоположное значение, значит, нужен инвертор. Событие должно случаться по сигналу «button_was_pressed», а значит речь, скорее всего, идет о входе разрешения работы триггера.

Немного подумав над этими вводными, нетрудно составить следующую схему:

————————picture————————

Временная диаграмма, которая соответствует работе этого устройства:

————————timing table————————

Описание поведения этой схемы на Verilog также не представляет сложности. Выполните его самостоятельно.

Теперь приведем полное описание секундомера (за исклю-

чением схемы, вырабатывающей сигнал «device_stopped»), выполненное на языке Verilog:

```
module stopwatch (
input start_stop,
input reset,
input clk,
output [6:0] hex0,  //индикатор сотых долей секунды
output [6:0] hex1,  //десятых долей секунды
output [6:0] hex2,  //секунд
output [6:0] hex3); //десятков секунд

// Часть I - синхронизация обработки
// нажатия кнопки «Старт/Стоп»
reg [2:0] button_syncroniser;
      wire button_was_pressed;

always @(posedge clk) begin
   button_syncroniser[0] <= start_stop;
   button_syncroniser[1] <= button_syncroniser[0];
   button_syncroniser[2] <= button_syncroniser[1];
end

assign button_was_pressed = ~button_syncroniser[2]
                            & button_syncroniser[1];


// Часть II - выработка признака «device_running»
// Самостоятельная работа студента!
reg device_running;



// Часть III - счётчик импульсов
// и признак истечения 0,01 сек
reg [16:0] pulse_counter;
wire hundredth_of_second_passed =
       (pulse_counter == 17'd259999);
always @(posedge clk or posedge reset) begin
   if (reset) pulse_counter <= 0;
```

```
38        //асинхронный сброс
39    else if (  device_running |
40               hundredth_of_second_passed)
41        if (hundredth_of_second_passed)
42               pulse_counter <= 0;
43        else pulse_counter <= pulse_counter + 1;
44       end
45
46
47  // Часть IV - основные счётчики
48  reg [3:0] hundredths_counter;
49  wire tenth_of_second_passed =
50        ((hundredths_counter == 4'd9) &
51          hundredth_of_second_passed);
52  always @(posedge clk or posedge reset) begin
53     if (reset) hundredths_counter <= 0;
54     else if (hundredth_of_second_passed)
55        if (tenth_of_second_passed)
56               hundredths_counter <= 0;
57        else hundredths_counter <=
58               hundredths_counter + 1;
59       end
60
61
62  reg [3:0] tenths_counter;
63  wire second_passed = ((tenths_counter == 4'd9) &
64                        tenth_of_second_passed);
65  always @(posedge clk or posedge reset) begin
66     if (reset) tenths_counter <= 0;
67     else if (tenth_of_second_passed)
68        if (second_passed) tenths_counter <= 0;
69        else tenths_counter <= tenths_counter + 1;
70       end
71
72  reg [3:0] seconds_counter;
73  wire ten_seconds_passed =
74        ((seconds_counter == 4'd9) &
75          second_passed);
76  always @(posedge clk or posedge reset) begin
77     if (reset) seconds_counter <= 0;
```

```
78     else if (second_passed)
79         if (ten_seconds_passed) seconds_counter <= 0;
80         else seconds_counter <= seconds_counter + 1;
81        end
82
83  reg [3:0] ten_seconds_counter;
84  always @(posedge clk or posedge reset) begin
85     if (reset) ten_seconds_counter <= 0;
86     else if (ten_seconds_passed)
87         if (ten_seconds_counter == 4'd9)
88                ten_seconds_counter <= 0;
89         else ten_seconds_counter <=
90                ten_seconds_counter + 1;
91        end
92
93
94
95
96  // Часть V - дешифраторы для отображения
97  // содержимого основных регистров
98  // на семисегментных индикаторах
99  reg [6:0] decoder_ten_seconds;
100 always @(*) begin
101    case (ten_seconds_counter)
102        4'd0: decoder_ten_seconds <= 7'b0000001;
103        4'd1: decoder_ten_seconds <= 7'b1001111;
104        4'd2: decoder_ten_seconds <= 7'b0010010;
105        4'd3: decoder_ten_seconds <= 7'b0000110;
106        4'd4: decoder_ten_seconds <= 7'b1001100;
107        4'd5: decoder_ten_seconds <= 7'b0100100;
108        4'd6: decoder_ten_seconds <= 7'b0100000;
109        4'd7: decoder_ten_seconds <= 7'b0001111;
110        4'd8: decoder_ten_seconds <= 7'b0000000;
111        4'd9: decoder_ten_seconds <= 7'b0000100;
112        default: decoder_ten_seconds <= 7'b1111111;
113    endcase;
114 end
115 assign hex3 = decoder_ten_seconds;
116 reg [6:0] decoder_seconds;
117 always @(*) begin
```

```
118     case (seconds_counter)
119         4'd0: decoder_seconds <= 7'b0000001;
120         4'd1: decoder_seconds <= 7'b1001111;
121         4'd2: decoder_seconds <= 7'b0010010;
122         4'd3: decoder_seconds <= 7'b0000110;
123         4'd4: decoder_seconds <= 7'b1001100;
124         4'd5: decoder_seconds <= 7'b0100100;
125         4'd6: decoder_seconds <= 7'b0100000;
126         4'd7: decoder_seconds <= 7'b0001111;
127         4'd8: decoder_seconds <= 7'b0000000;
128         4'd9: decoder_seconds <= 7'b0000100;
129         default: decoder_seconds <= 7'b1111111;
130     endcase;
131 end
132 assign hex2 = decoder_seconds;
133 
134 reg [6:0] decoder_tenths;
135 always @(*) begin
136     case (tenths_counter)
137         4'd0: decoder_tenths <= 7'b0000000;
138         4'd1: decoder_tenths <= 7'b1001111;
139         4'd2: decoder_tenths <= 7'b0010010;
140         4'd3: decoder_tenths <= 7'b0000110;
141         4'd4: decoder_tenths <= 7'b1001100;
142         4'd5: decoder_tenths <= 7'b0100100;
143         4'd6: decoder_tenths <= 7'b0100000;
144         4'd7: decoder_tenths <= 7'b0001111;
145         4'd8: decoder_tenths <= 7'b0000000;
146         4'd9: decoder_tenths <= 7'b0000100;
147         default: decoder_tenths <= 7'b1111111;
148     endcase;
149 end
150 assign hex1 = decoder_tenths;
151 
152 reg [6:0] decoder_hundredths;
153 always @(*) begin
154     case (hundredths_counter)
155         4'd0: decoder_hundredths <= 7'b0000000;
156         4'd1: decoder_hundredths <= 7'b1001111;
157         4'd2: decoder_hundredths <= 7'b0010010;
```

```
158         4'd3: decoder_hundredths <= 7'b0000110;
159         4'd4: decoder_hundredths <= 7'b1001100;
160         4'd5: decoder_hundredths <= 7'b0100100;
161         4'd6: decoder_hundredths <= 7'b0100000;
162         4'd7: decoder_hundredths <= 7'b0001111;
163         4'd8: decoder_hundredths <= 7'b0000000;
164         4'd9: decoder_hundredths <= 7'b0000100;
165         default: decoder_hundredths <= 7'b1111111;
166     endcase;
167 end
168 assign hex0 = decoder_hundredths;
169
170 endmodule
```

Листинг 3.5: Описание секундомера на языке Verilog HDL

3.1 Задание лабораторной работы:

Изучить разработку к лабораторной работе.

Самостоятельно выполнить описание схемы, вырабатывающей сигнал «device_stopped».

Выполнить синтез и моделирование работы счётчика.

Продемонстрировать в результатах моделирования фрагменты временных диаграмм, приведенных в заработке.

Изучить работу устройства, реализованного в ПЛИС учебного стенда.

Подготовить ответы на вопросы к защите лабораторной работы.

3.2 Вопросы к защите лабораторной работы

in progress

Лабораторная работа №6
FLASH память

4.1 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 этих целей 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-памятью.

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 недоступна для рядового пользователя из-за высокой стоимости.

4.2 Принципы работы FLASH-памяти

В качестве элемента хранения информации FLASH-память

использует транзистор с плавающим затвором. Состояние затвора определяет бит хранимой информации.

На Рис.**??** схематично изображена структура такого транзистора.

[Picture goes here]

Как вы видите, он содержит два затвора: управляющий и плавающий. Плавающий затвор — это полупроводника, который полностью окружён диэлектриком. При этом плавающий затвор способен накапливать электроны. От величины накопленного заряда меняется «лёгкость» с которой транзистор открывается — т.е. величина напряжения «управляющий затвор—исток», при котором через транзистор начнёт течь ток. Чем больше электронов находятся в плавающем затворе, тем «легче» открывается транзистор — ток начинает протекать через него при меньшем напряжении «управляющий затвор—исток».

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

[Picture goes here]

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 «выгнать» электроны наружу (см. Рис.**??**).

[Picture goes here]

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 устроен 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.

4.3 Особенности FLASH-памяти

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 (сектора);
- Удаление и запись информации приводят к деградации ячеек памяти;
- Чтение также приводит к деградации ячеек памяти, но в меньшей степени.

4.4 Структура FLASH-памяти

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 управле-

ния — доступ к содержимому FLASH-памяти может быть синхронный или асинхронный, по последовательной или параллельной шине, с разделением шин адреса и данных или без него.

[Picture goes here]

4.5 Микросхема 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 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, в первую очередь, интересует каким образом необходимо взаимодействовать с этой 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.

4.5.1 Проектирование контроллера S29AL032D

Как мы уже знаем, контроллер предназначен для обмена информацией с внешними цифровыми устройствами. Он дол-

жен предоставлять удобный, простой интерфейс и обеспечивать все необходимые взаимодействия с устройством. В таком случае другие блоки могут использовать один и тот же контроллер.

Чтобы начать разработку контроллера, нужно ответить важные вопросы: как должен работать наш контроллер и как он должен управляться?

Если мы хотим работать с памятью, то для нас наиболее важными являются операции записи и чтения данных. Тогда наиболее удобным для нас был бы уже знакомый интерфейс, похожий на RAM-память: данные для записи, данные для чтения, адрес и управляющие сигналы.

Теперь, когда мы определились с тем, как мы будем управлять контроллером, нам нужно понять как он должен взаимодействовать с самой микросхемой flash-памяти. Для этого изучим операции записи и чтения, описанные в datasheet S29AL032D.

4.5.2 Операция чтения

Чтение данных из микросхемы S29AL032D не требует никакой дополнительной подготовки. Временная диаграмма чтения приведена в пункте 17.2 datasheet и представлена на Рис.**??**

[Picture goes here]

Времена, указанные на диаграмме, приведены в Табл. 4.2

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 4.1: Временные характеристики операции чтения S29AL032D

Чтобы выполнить операцию чтения, нам нужно повторить эту временную диаграмму и соблюсти все временные интервалы. Но как это сделать?

Каким образом можно выдержать указанные временные интервалы?

Мы уже знаем, что единственным источником информации о времени для цифрового устройства может являться только сигнал синхронизации, частота которого заранее известна.

Привяжем времена, упомянутые в Таблице 4.2 к тактовому сигналу частоты 50 МГц, которым тактируется устройство. При этом учтём, что некоторые задержки могут быть равны нулю.

Полученная временная диаграмма показана на Рис. **??**

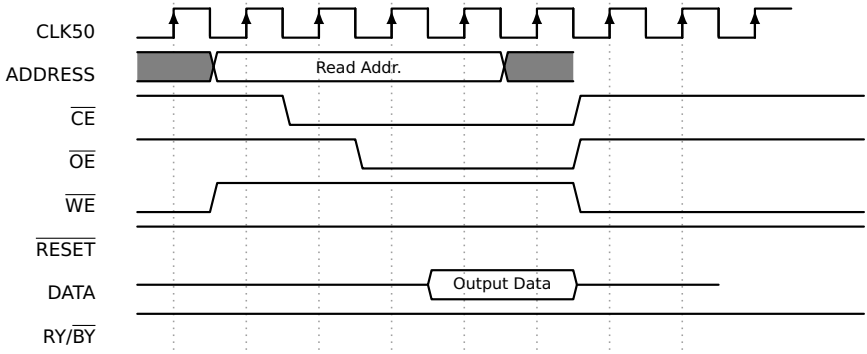

Рис. 4.1: Временная диаграмма операции чтения шины PCI

4.5.3 Операция записи

Обычно запись во flash-память — более сложная операция, чем чтение. Многие производители используют для записи специальные последовательности команд, защищая таким образом память от случайной записи.

Согласно datasheet S29AL032D (разделы 7 и 11) для того, чтобы осуществить запись нужного значения во flash-память, необходимо выполнить следующую последовательность из 4-х операций записи:

Записать данные AA по адресу AAA;
Записать данные 55 по адресу 555;
Записать данные A0 по адресу AAA;
Записать нужные данные по нужному адресу.

Временная диаграмма одной операции записи приведена в пункте 17.2??? datasheet и представлена на Рис.**??**, а её вре-

менные характеристики приведены в Табл. **??**.

[Picture goes here]

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 4.2: Временные характеристики операции чтения S29AL032D

Аналогично операции чтения, привяжем форму и времена временной диаграммы записи к тактовому сигналу, частотой 50 МГц. Полученная диаграмма, представлена на Рис. **??**

Также согласно datasheet, данные записываются не мгновенно. На то, чтобы провести операцию записи одного слова требуется порядка 11 мкс. Что приблизительно соответствует 550 тактам на частоте 50 МГц.

Также крайне важно, что при записи данных микросхема S29AL032D может менять значение с «1» на «0», но не наоборот!

Чтобы поменять значение с «0» на «1» требуется очистка целого фрагмента памяти, называемого сектором, либо полная очистка всей микросхемы!

Значит, для того, чтобы мы могли полноценно пользоваться микросхемой S29AL032D нам потребуется реализовать в контроллере функции очистки.

4.5.4 Операция очистки

Для очистки выбранного сектора необходимо выполнить следующую последовательность операций:

Записать данные AA по адресу AAA;
Записать данные 55 по адресу 555;
Записать данные 80 по адресу AAA;
Записать данные AA по адресу AAA;
Записать данные 55 по адресу 555;
Записать данные 30 по адресу сектора, который необходимо очистить.

Операция очистки сектора занимает существенное время,

и пока она не закончится, невозможно произвести запись или чтение из flash-памяти.

Операция полной очистки отличается только последним значением: для полной очистки данные 30 записываются по адресу AAA.

В datasheet на S29AL032D приведены следующие значения:

Очистка сектора - до NN мкс.

Полная очистка микросхемы - до NNN мкс.

4.5.5 Статус операции

Для того, чтобы контролировать завершение операций записи и очистки, а также отслеживать ошибки, которые могут возникнуть в процессе их выполнения необходимо получить информацию о статусе операции. Способы получения этой информации и её содержание приведены в разделе 12 datasheet. Далее мы отметим наиболее важные для нас моменты.

Так как на отладочном стенде Altera DE1, которым мы пользуемся для проведения лабораторных работ, не разведён сигнал BUSY микросхемы S29AL032D, то единственным способом получения статуса является чтение информации из адреса, по которому производилась запись.

Если операция удачно завершена — то будет получено значение, из указанного адреса. Для операции записи оно должно совпадать с тем, которое мы хотели записать, а для операции очистки должно содержать только единицы (8'hFF).

Если операция ещё не завершена, то биты [7:2] считанного значения будут содержать информацию о статусе операции. Расшифровка значений этих битов приведена в Табл. **??**

До окончания операции записи DQ[7] будет иметь значение противоположное записываемому («0» при очистке) - это основной признак того, что полученные данные отражают статус операции. Информация о битах статусного пакета приведена в Таблице **??**

Обратите внимание, что при повторном чтении некоторые биты статусного пакета меняют своё значение на противоположное. Это сделано, чтобы убедиться, что операция чтения выполняется корректно и микросхема не «зависла».

В информации о статусе операции есть важный признак: бит DQ[4] является признаком того, что время операции превысило максимально допустимое. Если этот бит принимает значение «1», то во время операции произошла какая-то ошибка. В подавляющем большинстве случаев это происходит при попытке записи в ячейку памяти, уже содержавшую какое-то значение.

4.5.6 Проектирование контроллера Flash (продолжение)

Теперь, когда мы познакомились с операциями, которые предстоит выполнять контроллеру, мы можем продолжить его проектирование.

Контроллер должен обеспечивать операции чтения, записи и очистки микросхемы. Для этого он должен последовательно обмениваться данными и производить проверку статуса операций. Значит в качестве его основы следует применить конечный автомат. Ведь именно конечный автомат позволяет нам разделить режимы работы и реализовать алгоритмы работы в цифровых устройствах.

Начнём проектировать конечный автомат с начального состояния - состояния бездействия. Будем постепенно наращивать его сложность и степень детализации, уточняя некоторые особенности.

[Picture goes here]

Из состояния бездействия возможны три различных перехода: операция чтения, операция записи и операция очистки.

[Picture goes here]

Теперь выделим основные этапы, которые присутствуют в этих операциях. Прежде всего нас интересуют сложные операции «запись» и «очистка».

Как уже говорилось, чтобы записать данные во flash-память требуется провести четыре обмена с flash-памятью. Но на этом нельзя заканчивать операцию, ведь необходимо дождаться окончания записи. Также надо учесть, что во время записи могут возникнуть ошибки.

Как мы уже говорили, для контроля статуса операции нам

нужно считать данные из адреса, по которому производится запись и проанализировать их. Отразим это в состояниях конечного автомата.

[Picture goes here]

Раньше мы не разделяли эти состояния и всё вместе называли «запись». Но, постепенно уточняя детали, мы разбили сложную операцию на более простые этапы.

Аналогично поступим с операцией очистки.

[Picture goes here]

Первое, что бросается в глаза - многократное повторение операций записи и чтения (которая используется при проверке статуса).

Также можно постараться выделить чтение и анализ статуса в отдельные состояния, общие для операций записи и очистки.

Тогда структура конечного автомата приобретает следующий вид:

[Picture goes here]

В состояниях W_n и E_n происходит запись значения во flash-память. В состояниях ST и R происходит чтение.

Можем ли мы выделить операции чтения и записи и реализовать их отдельно, чтобы затем использовать их как показано на графе переходов?

Чтобы понять это, сначала ответим на вопрос как вообще возможно реализовать эти операции в контроллере.

Для того, чтобы провести чтение, необходимо развернуть временную диаграмму, показанную на Рис.**??**. Тоже относится к записи: временную диаграмму записи, привязанную к тактовому сигналу, мы получили на Рис. **??**

Как мы уже обсуждали, схема которую можно использовать для разделения событий во времени — это конечный автомат. Например, чтобы реализовать операцию чтения, разобьём временную диаграмму чтения на этапы, и поставим каждому этапу в соответствие уникальное состояние, как представлено на Рис. **??**

Мы могли бы добавить эти состояния в конечный автомат, который мы уже начали проектировать, но тогда нам

пришлось бы каждое состояние чтения и записи разбить на несколько состояний. Это привело бы к ненужному усложнению структуры конечного автомата.

Вместо этого мы можем сделать отдельные небольшие модули, которые будут выполнять эти операции, и поместить автоматы в них. Например для операции чтения такой модуль будет управляться автоматом, представленным на Рис.**??**

Сигналом запуска для таких мини-автоматов будет признак того, что основной автомат находится в состоянии «чтение» или «запись».

Состояние «завершено» нужно для того, чтобы выработать сигнал окончания работы. Иначе «большой» автомат не будет иметь возможности «узнать» о том, что операция завершена и можно переходить в следующее состояние.

Теперь, когда мы оформили все основные идеи и общую структуру контроллера, можно преступить к его реализации на Verilog HDL.

Как всегда, начнём проектировать с интерфейса будущего контроллера - его входов и выходов. Так как контроллер будет обеспечивать доступ к памяти, мы хотели сделать его интерфейс похожим на интерфейс RAM-памяти. Но нам придётся ввести дополнительные сигналы для того, чтобы реализовать операцию очистки и индикацию ошибок.

Нам будет достаточно одного входа для адреса, так как мы не можем одновременно производить чтение и запись во flash-память.

Теперь опишем основной управляющий конечный автомат и мини-автомат чтения. Мини-автомат записи опишите самостоятельно.

Наладим связь между автоматами. Для этого определим управляющие сигналы (воздействия):

Начнём описывать исполняющую логику, которая будет задействована в различных состояниях: