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, возросла 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 — один 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.

Схематично изобразим этот модуль:

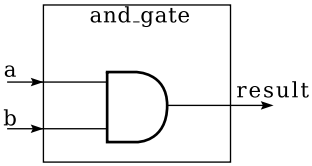


Рис. 1.1: Структура модуля «and_gate»

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 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

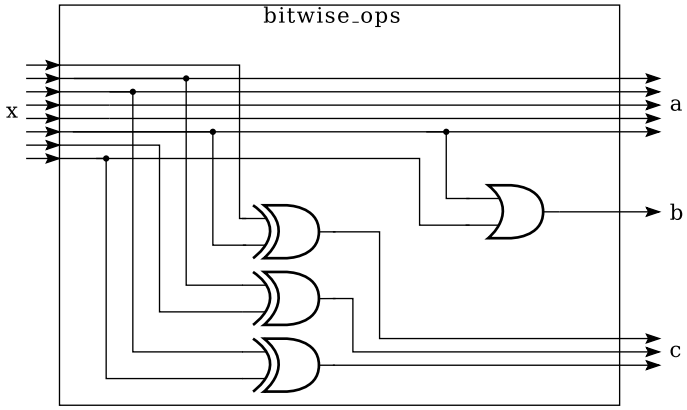


Рис. 1.2: Структура модуля «bitwise_ops»

Впрочем, реализация ФАЛ с помощью логических вентилей не всегда представляется удобной. Допустим нам нужно описать таблично-заданную ФАЛ. Тогда для 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 @(x_bus) begin
11   case (x_bus)
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» она подключается к объединению проводов «x2», «x1» и «x0» с помощью оператора assign как показано на Рис. 1.3.

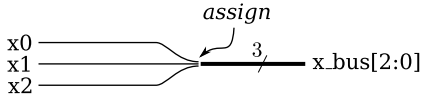


Рис. 1.3: Действие оператора **assign**

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 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 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

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 конструкцию 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. **??** показано принятое в цифровой схемотехнике обозначение дешифратора.

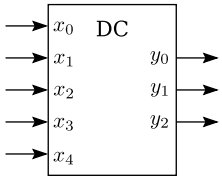


Рис. 1.4: Графическое обозначение дешифратора

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     2'b00:   y <= a;
12     2'b01:   y <= b;
13     2'b10:   y <= c;
14     2'b11:   y <= d;
15     default: y <= a;
16   endcase
```

```
17  end
18
19  endmodule
```

Листинг 1.10: Описание мультиплексора на языке Verilog

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

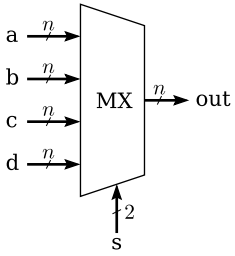


Рис. 1.5: Графическое обозначение мультиплексора

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 ви-

да: $f(s_1, s_0, a_0, b_0, c_0, d_0) = y_0$ 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:

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

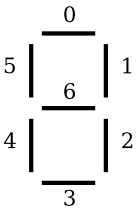
1.3 Задание лабораторной работы

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

1. Ввод информации происходит с переключателей SW[9:0];
2. SW[3:0] должны обрабатываться дешифратором «DC1», согласно индивидуальному заданию;
3. SW[7:4] должны обрабатываться дешифратором «DC2», согласно индивидуальному заданию;
4. Реализовать дешифратор «DC-DEC», преобразующий число, представленное в двоичном коде в цифру, отображаемую на семисегментном индикаторе. Руководствоваться при этом нужно следующими соображениями:
   - Семисегментный индикатор подключается к шине

HEX0[6:0]

- Диоды на семисегментном индикаторе загораются при подаче на них низкого напряжения (0 - горит, 1 - не горит)
- Соответствие линий диодам семисегментного индикатора приведено ниже:



5. С помощью мультиплексора реализовать следующую схему подключения:
   - Если SW[9:8] = 00, на дешифратор DC-DEC поступает выход DC1;
   - Если SW[9:8] = 01, на дешифратор DC-DEC поступает выход DC2;
   - Если SW[9:8] = 10, на дешифратор DC-DEC поступает выход логической функции f;
   - Если SW[9:8] = 11, на дешифратор DC-DEC поступает SW[3:0].

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.

1.4 Варианты индивидуальных заданий

1. - Логика работы дешифратора DC1:
     Кодирует количество переключателей SW[3:0] в положении «0».
   - Логика работы дешифратора DC2:

Логическое "ИЛИ"сигналов с переключателей SW[7:4] с числом «0101».

- Функция f:
  $f = SW[0]||(SW[1] \oplus (SW[2]\&SW[3]))$

2.
- Логика работы дешифратора DC1:
  Кодирует количество сочетаний «01» на SW[3:0]
- Логика работы дешифратора DC2:
  Логическое "И"сигналов с переключателей SW[7:4] с числом «1101»
- Функция f:
  $f = (SW[0] \oplus SW[1])||(SW[2] \oplus SW[3])$

3.
- Логика работы дешифратора DC1:
  Кодирует количество сочетаний «10» на SW[3:0]
- Логика работы дешифратора DC2:
  Логическое "НЕ"сигналов с переключателей SW[7:4]
- Функция f:
  $f = SW[0]\&SW[1]\&(\neg SW[2])\&(\neg SW[3])$

4.
- Логика работы дешифратора DC1:
  Кодирует количество переключателей SW[3:0] в положении «1».
- Логика работы дешифратора DC2:
  Число с переключателей SW[7:4], сдвинутое на 1 двоичный разряд влево.
- Функция f:
  $f = (SW[0]||SW[1]||SW[2])\&SW[3]$

5.
- Логика работы дешифратора DC1:
  Кодирует количество переключателей SW[3:0] в положении «0».
- Логика работы дешифратора DC2:
  Логическое "исключающее ИЛИ"сигналов с переключателей SW[7:4] с числом «0111».
- Функция f:
  $f = (\neg SW[0]) \oplus (\neg SW[1])||(SW[2]\&SW[3])$

6.
- Логика работы дешифратора DC1:
Кодирует количество сочетаний «01» на SW[3:0]
- Логика работы дешифратора DC2:
Логическое "НЕ"сигналов с переключателей SW[7:4]
- Функция f:
$f = (\neg SW[0])||(\neg SW[1])||(\neg SW[2])||(\neg SW[3])$

7.
- Логика работы дешифратора DC1:
Кодирует количество сочетаний «11» на SW[3:0]
- Логика работы дешифратора DC2:
Логическое "И"сигналов с переключателей SW[7:4] с числом «1001»
- Функция f:
$f = (SW[0]\&SW[1]) \oplus (SW[2]||SW[3])$

8.
- Логика работы дешифратора DC1:
Число с переключателей SW[3:0] - число 3 (десятичное).
- Логика работы дешифратора DC2:
Логическое "исключающее ИЛИ"сигналов с переключателей SW[7:4] с числом «1000».
- Функция f:
$f = ((\neg SW[0])\&SW[1])||SW[2]||SW[3]$

9.
- Логика работы дешифратора DC1:
Число с переключателей SW[3:0], делённое на 2 без остатка.
- Логика работы дешифратора DC2:
Логическое "И"сигналов с переключателей SW[7:4] с числом «1010»
- Функция f:
$f = (SW[0] \oplus SW[3])\&(SW[1]||SW[2])$

10.
- Логика работы дешифратора DC1:
Кодирует количество сочетаний «010» на SW[3:0]
- Логика работы дешифратора DC2:
Логическое "исключающее ИЛИ"сигналов с переключателей SW[7:4] с числом «0011».

- **Функция f:**
  $f = SW[0]||SW[1]\&SW[2]||SW[3]$

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:

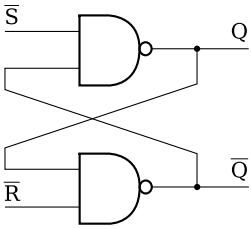


Рис. 2.1: Структура RS-защелки

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.

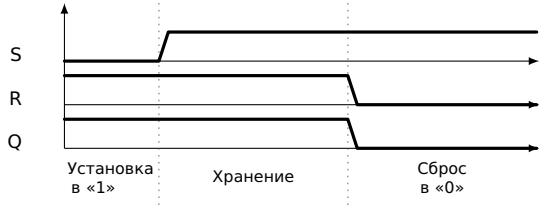


Рис. 2.2: Временная диаграмма работы RS-защелки

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

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

лов.

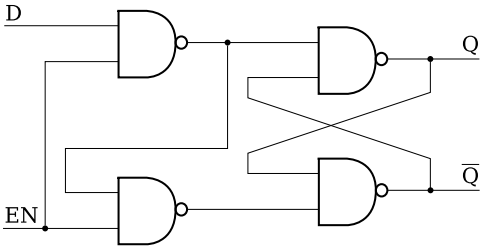


Рис. 2.3: Структура D-защелки

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.

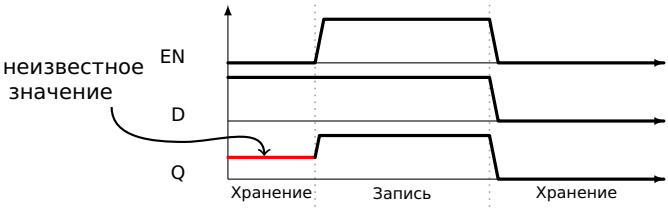


Рис. 2.4: Временная диаграмма работы D-защелки

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

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

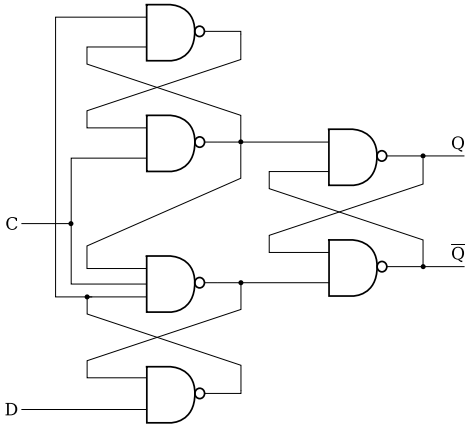


Рис. 2.5: Структура D-триггера

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

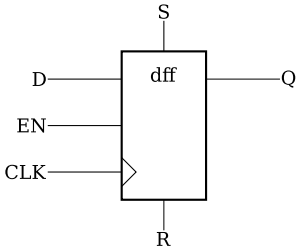


Рис. 2.6: Графическое обозначение D-триггера

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

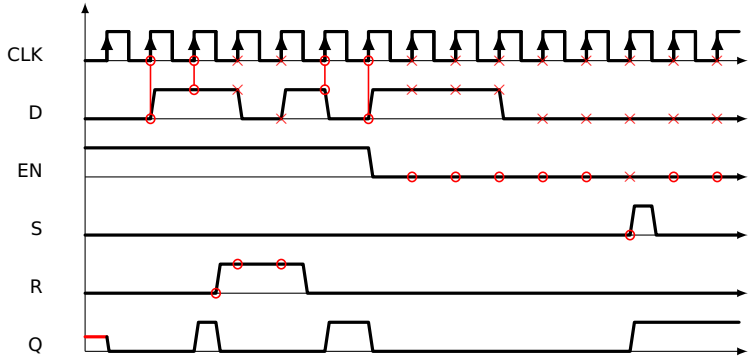


Рис. 2.7: Пример работы D-триггера

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_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

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

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 в разных блоках always на языке Verilog

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:

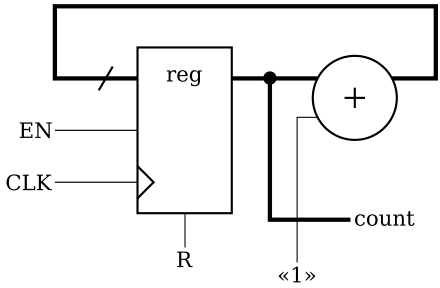


Рис. 2.8: Структура восьмибитного счетчика

На временной диаграмме ниже хорошо видно как работает счётчик:

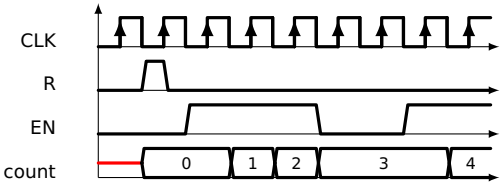


Рис. 2.9: Пример работы регистра

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 восьмибитного счетчика на языке Verilog

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:

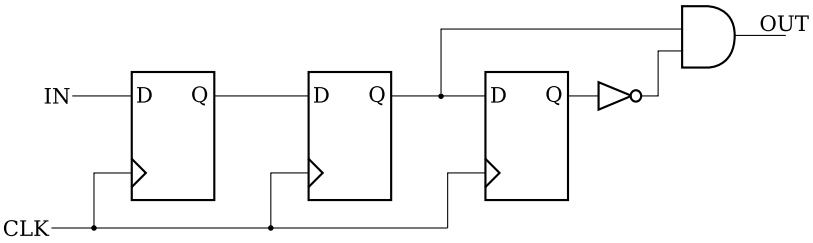


Рис. 2.10: Схема выработки синхронного импульса из асинхронного события

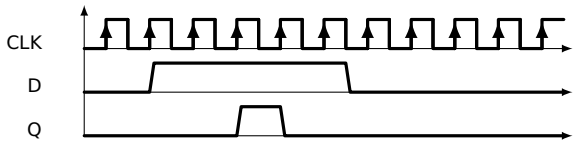


Рис. 2.11: Временная диаграмма работы схемы синхронизации

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

Сигнал OUT в таком случае подключается к входу EN 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 отладоч-

ной 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 Варианты индивидуальных заданий

1. Событие:
   Количество сочетаний «01» на переключателях SW[9:0] не менее 4.

2. Событие:
   Количество переключателей SW[9:0] в положении «1» не

менее 5.

3. Событие:
   На переключателях закодировано число больше 10, но меньше 20.

4. Событие:
   Четное количество переключателей SW[9:0] в положении «1».
   (нуль – четное число)

5. Событие:
   Симметрия переключателей SW[9:0] относительно середины.

6. Событие:
   Число на SW[9:5] больше числа на SW[4:0] как минимум в 2 раза.

7. Событие:
   Асимметрия переключателей SW[9:0] относительно центра.

8. Событие:
   Количество сочетаний «101» на переключателях не менее 2.

9. Событие:
   Количество переключателей SW[9:0] в положении «1» не более 3.

10. Событие:
    Нечетное количество переключателей SW[9:0] в положении «1».

2.4 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 семисегментный индикатор? Как называется эта цифровая схема?
6. Продемонстрируйте код, реализующий индивидуальное задание.
7. Покажите в коде лабораторной работы код, реализующий счётчик.
8. Что такое сигнал синхронизации?

Лабораторная работа №3 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:

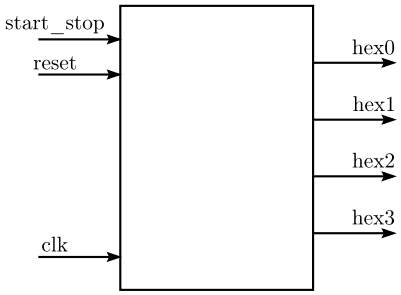


Рис. 3.1: Общая схема секундомера

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
11 endmodule
```

Листинг 3.1: Описание входов и выходов секундомера

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 генератор 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:

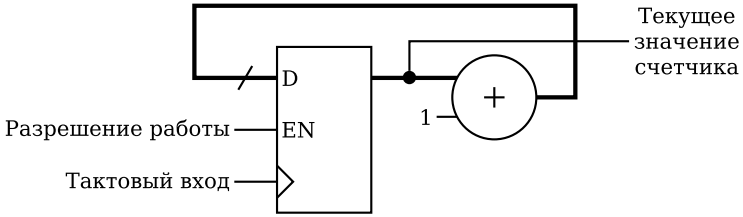


Рис. 3.2: Структура счетчика

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.

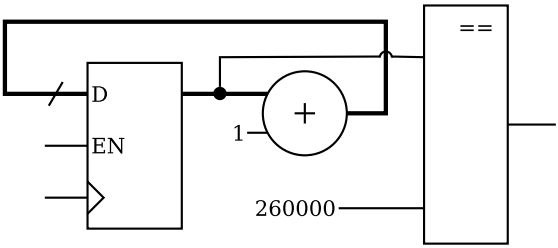


Рис. 3.3: Структура счетчика с компаратором

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:

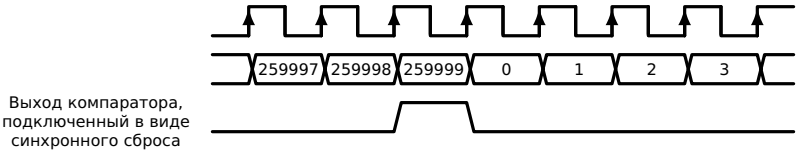


Рис. 3.4: Временная диаграмма работы счётчика с синхронным сбросом

А так выглядит временная диаграмма, если подключить выход компаратора к входу асинхронного сброса триггера:

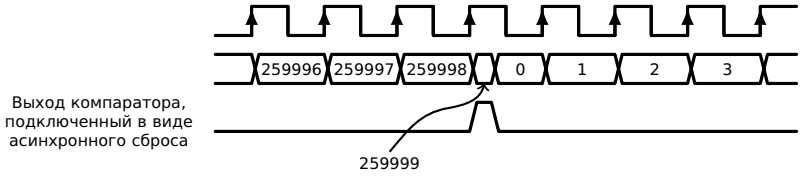


Рис. 3.5: Временная диаграмма работы счётчика с асинхронным сбросом

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 №2).

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:

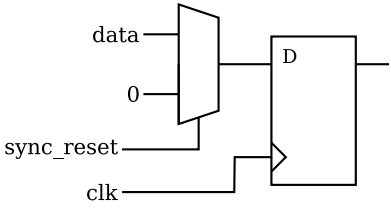


Рис. 3.6: Схема реализации синхронного сброса

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:

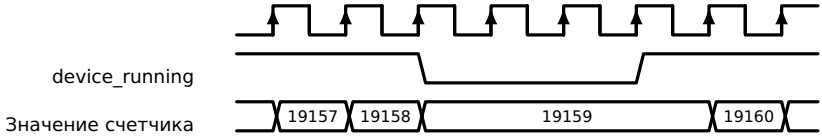


Рис. 3.7: Временная диаграмма работы сигнала device_running

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:

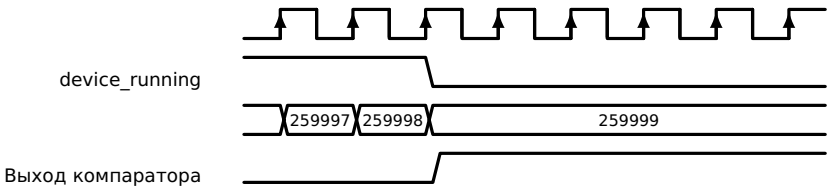


Рис. 3.8: Временная диаграмма работы счетчика

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:

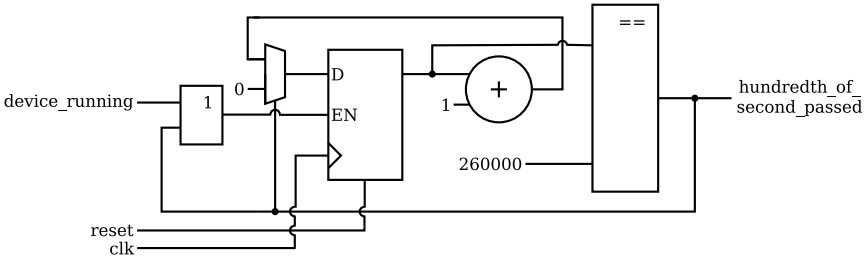


Рис. 3.9: Схема счетчика тысячных долей секунды

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 = 17'd0;
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    // асинхронный сброс
11   if (reset) pulse_counter <= 0;
12
13    // сигнал разрешения работы счётчика
14   else if (device_running |
15               hundredth_of_second_passed)
16
17        // синхронный сброс по достижению максимума
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

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:

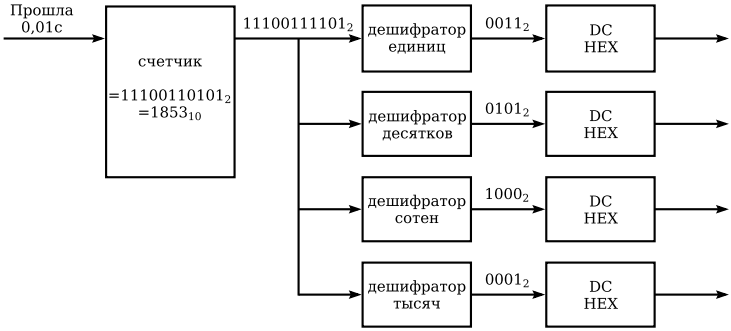


Рис. 3.10: Схема секундомера с дешифраторами разрядов

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.

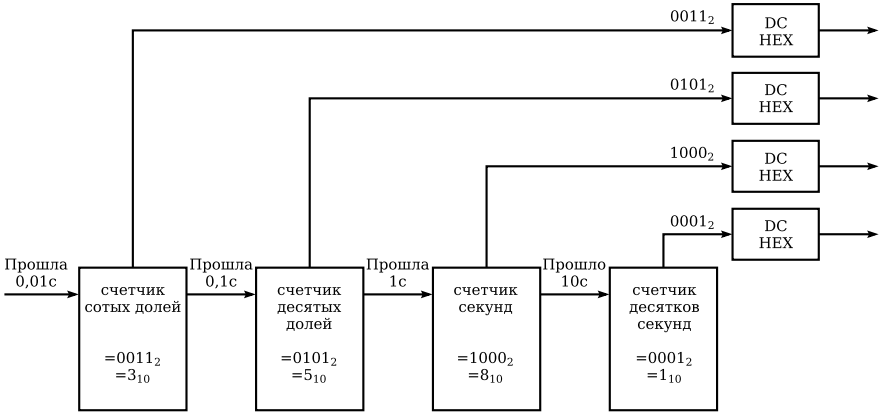


Рис. 3.11: Схема секундомера со счётчиками разрядов

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:

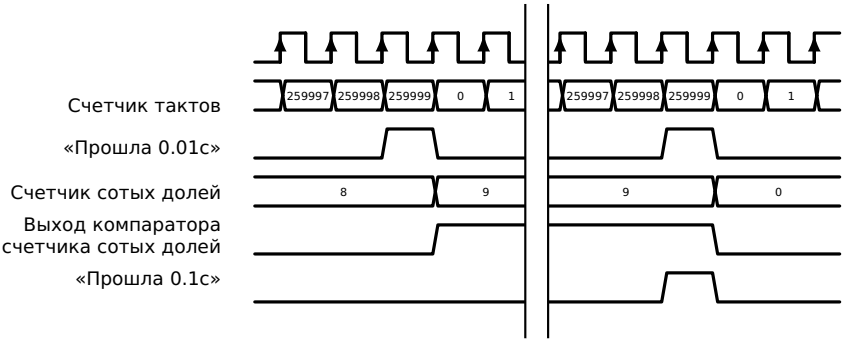


Рис. 3.12: Временная диаграмма работы секундомера

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:

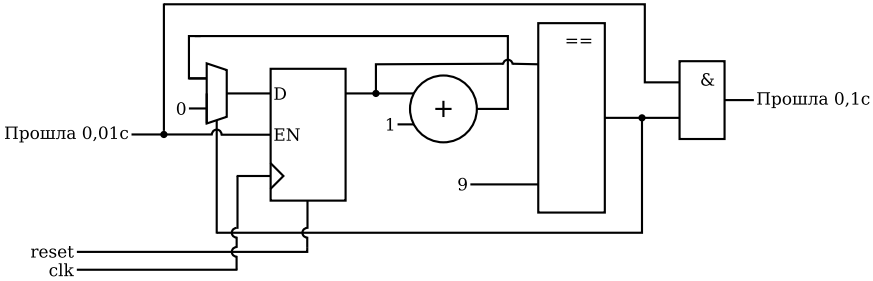


Рис. 3.13: Схема счётчика сотых долей секунды

Скорректируем описание её работы на Verilog:

```
// регистр счётчика
reg [3:0] hundredth_counter = 4'd0;

// описание компаратора
wire tenth_of_second_passed =
       ((hundredths_counter == 4'd9) &
         hundredths_of_second_passed);

// описание счётчика
always @(posedge clk or posedge reset) begin

   // асинхронный сброс
  if (reset) hundredths_counter <= 0;

   // сигнал разрешения работы счётчика
  else if (hundredth_of_second_passed)

       // синхронный сброс по достижению максимума
    if (tenth_of_second_passed)
           hundredths_counter <= 0;

       // увеличение счётчика на единицу
    else hundredths_counter <=
```

```
24              hundredths_counter + 1;
25  end
```

Листинг 3.3: Описание счетчика сотых долей секунды на языке Verilog

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 №2, 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 №2 и будет выглядеть следующим образом:

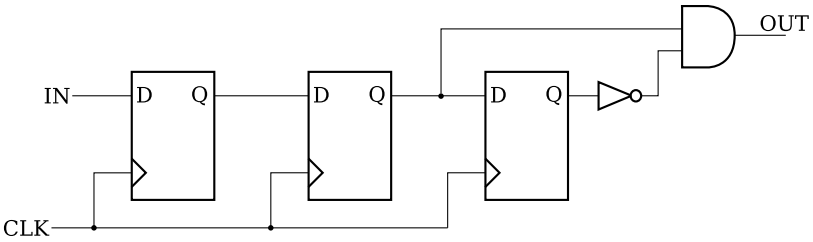


Рис. 3.14: Схема выработки синхронного импульса из асинхронного события

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 синхронизации на языке Verilog

Эта схема и её описание подробно рассмотрены в лабораторной работе №2

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:

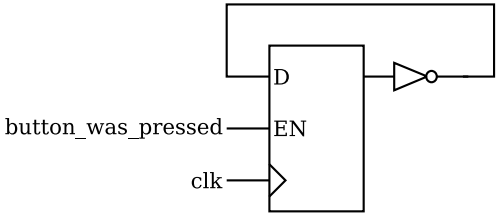


Рис. 3.15: Схема переключения сигнала «device_running»

Временная диаграмма, которая соответствует работе этого устройства:

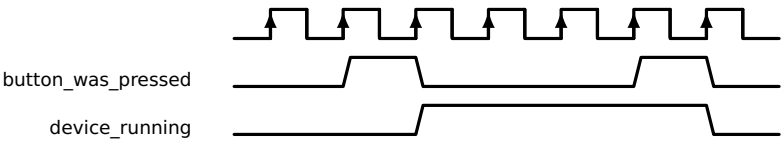


Рис. 3.16: Временная диаграмма переключения сигнала «device_running»

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

Теперь приведем полное описание секундомера (за исключением 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 «СтартСтоп»/
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!
```

```
27 reg device_running;
28
29
30
31 //  Часть III - счётчик импульсов
32 //  и признак истечения 0,01 сек
33 reg [16:0] pulse_counter =  17'd0;
34 wire hundredth_of_second_passed =
35       (pulse_counter ==  17'd259999);
36 always @(posedge clk or posedge reset) begin
37   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 = 4'd0;
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 =  4'd0;
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;
```

```
  else if (tenth_of_second_passed)
      if (second_passed) tenths_counter <= 0;
      else tenths_counter <= tenths_counter + 1;
end

reg [3:0] seconds_counter = 4'd0;
wire ten_seconds_passed =
       ((seconds_counter == 4'd9) &
         second_passed);
always @(posedge clk or posedge reset) begin
  if (reset) seconds_counter <= 0;
  else if (second_passed)
      if (ten_seconds_passed) seconds_counter <= 0;
      else seconds_counter <= seconds_counter + 1;
end

reg [3:0] ten_seconds_counter = 4'd0;
always @(posedge clk or posedge reset) begin
  if (reset) ten_seconds_counter <= 0;
  else if (ten_seconds_passed)
      if (ten_seconds_counter == 4'd9)
              ten_seconds_counter <= 0;
      else ten_seconds_counter <=
              ten_seconds_counter + 1;
end




//  Часть V - дешифраторы для отображения
//  содержимого основных регистров
//  на семисегментных индикаторах
reg [6:0] decoder_ten_seconds;
always @(*) begin
  case (ten_seconds_counter)
      4'd0:    decoder_ten_seconds <=  7'b0000001;
      4'd1:    decoder_ten_seconds <=  7'b1001111;
      4'd2:    decoder_ten_seconds <=  7'b0010010;
      4'd3:    decoder_ten_seconds <=  7'b0000110;
      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
116   assign hex3 = decoder_ten_seconds;
117
118   reg [6:0] decoder_seconds;
119   always @(*) begin
120     case (seconds_counter)
121       4'd0:    decoder_seconds <=  7'b0000001;
122       4'd1:    decoder_seconds <=  7'b1001111;
123       4'd2:    decoder_seconds <=  7'b0010010;
124       4'd3:    decoder_seconds <=  7'b0000110;
125       4'd4:    decoder_seconds <=  7'b1001100;
126       4'd5:    decoder_seconds <=  7'b0100100;
127       4'd6:    decoder_seconds <=  7'b0100000;
128       4'd7:    decoder_seconds <=  7'b0001111;
129       4'd8:    decoder_seconds <=  7'b0000000;
130       4'd9:    decoder_seconds <=  7'b0000100;
131       default: decoder_seconds <=  7'b1111111;
132     endcase
133   end
134
135   assign hex2 = decoder_seconds;
136
137   reg [6:0] decoder_tenths;
138   always @(*) begin
139     case (tenths_counter)
140       4'd0:    decoder_tenths <= 7'b0000000;
141       4'd1:    decoder_tenths <= 7'b1001111;
142       4'd2:    decoder_tenths <= 7'b0010010;
143       4'd3:    decoder_tenths <= 7'b0000110;
144       4'd4:    decoder_tenths <= 7'b1001100;
145       4'd5:    decoder_tenths <= 7'b0100100;
146       4'd6:    decoder_tenths <= 7'b0100000;
```

```
147       4'd7:    decoder_tenths <= 7'b0001111;
148       4'd8:    decoder_tenths <= 7'b0000000;
149       4'd9:    decoder_tenths <= 7'b0000100;
150       default: decoder_tenths <= 7'b1111111;
151     endcase
152 end
153 
154 assign hex1 = decoder_tenths;
155 
156 reg [6:0] decoder_hundredths;
157 always @(*) begin
158   case (hundredths_counter)
159       4'd0:    decoder_hundredths <= 7'b0000000;
160       4'd1:    decoder_hundredths <= 7'b1001111;
161       4'd2:    decoder_hundredths <= 7'b0010010;
162       4'd3:    decoder_hundredths <= 7'b0000110;
163       4'd4:    decoder_hundredths <= 7'b1001100;
164       4'd5:    decoder_hundredths <= 7'b0100100;
165       4'd6:    decoder_hundredths <= 7'b0100000;
166       4'd7:    decoder_hundredths <= 7'b0001111;
167       4'd8:    decoder_hundredths <= 7'b0000000;
168       4'd9:    decoder_hundredths <= 7'b0000100;
169       default: decoder_hundredths <= 7'b1111111;
170   endcase
171 end
172 
173 assign hex0 = decoder_hundredths;
174 
175 endmodule
```

Листинг 3.5: Описание секундомера на языке Verilog

3.1 Задание лабораторной работы:

1. Изучить разработку к лабораторной работе.
2. Самостоятельно выполнить описание схемы, вырабатывающей сигнал «device_stopped».
3. Выполнить синтез и моделирование работы счётчика.

4. Продемонстрировать в результатах моделирования фрагменты временных диаграмм, приведенных в заработке.
5. Изучить работу устройства, реализованного в ПЛИС учебного стенда.

Лабораторная работа №4
Конечные автоматы

Когда мы проектировали секундомер, у нас возникла необходимость выделить признак того, что в данный момент секундомер работает. Для этого мы создали специальный регистр для хранения этого признака и описали схему управления этим регистром. В других блоках секундомера мы могли проверить признак работы (содержание регистра) и, в зависимости от него, разрешить, запретить или изменить работу.

Похожий подход применяется и в других случаях, когда необходимо выделить разные режимы работы цифрового устройства или обеспечить последовательное выполнение некоторых операций.

Действительно, ведь цифровое устройство существует целиком в один момент времени. Тогда как возможно добиться от него последовательного выполнения каких-то действий?

Вариант только один – описать ещё один блок, который будет управлять работой всей остальной части цифрового устройства.

Блок, который используется для этого, называется «конечный автомат».

Существует целый раздел математики, посвященный автоматам (и в частности конечным автоматам), который, вполне ожидаемо, называется «теория автоматов».

Но, так как нас, прежде всего, интересует прикладное применение конечных автоматов, мы рассмотрим их с точки зрения цифровой схемотехники.

В цифровой схемотехнике, конечным автоматом называется цифровая схема, которая отслеживает текущее состояние и обеспечивает переход от одного состояния к другому в зависимости от входных воз-

действий.

Например, в случае секундомера, который мы проектировали в прошлой лабораторной работе, конечный автомат, который мог бы управлять им, переключался бы между двумя состояниями: «остановлен» и «работает».

Для удобства проектирования конечный автомат представляют в виде графа. Вершины графа - это состояния, а рёбра графа - это воздействия, которые приводят к изменению состояния. Этот граф называют графом переходов конечного автомата.

Обычно граф переходов выглядит подобным образом:

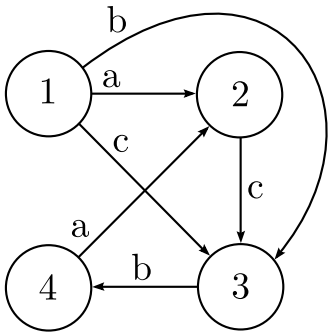


Рис. 4.1: Граф переходов конечного автомата

Граф переходов – это основной источник информации о конечном автомате. Он дает наглядное представление о том, как управляется и работает конечный автомат.

Обратите внимание – конечный автомат всегда прибывает в одном из своих состояний. Конечный автомат не может прибывать в двух состояниях одновременно. Чем-то граф переходов может напомнить вам игровую доску, где фишка находится на одном из полей и, в зависимости от выполненных условий, может перейти в одно из полей, соединённых с текущим.

Классическим примером, наглядно демонстрирующим работу конечных автоматов, может служить конечный автомат управления светофором. Попробуем разработать такой автомат.

Можно легко выделить состояния светофора: «горит зеле-

ный» - «мигает зеленый» - «горит желтый» - «горит красный» - «горит желтый и красный» - «мигает желтый».

Обратите внимание: автомат управления не может находиться в двух состояниях одновременно и, поэтому, чтобы описать ситуацию когда горят одновременно желтый и красный сигналы, нам потребовалось ввести новое состояние - «горит желтый и красный».

Обозначим все вершины графа – состояния конечного автомата:

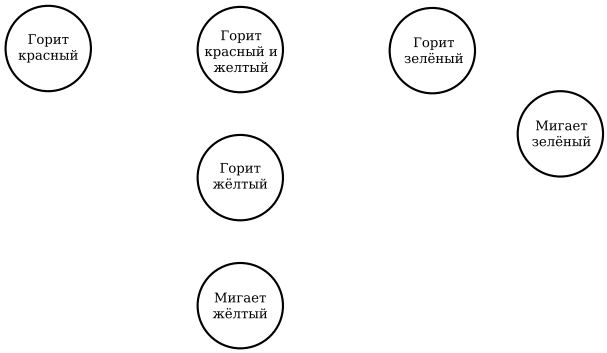


Рис. 4.2: Вершины графа переходов конечного автомата

Теперь стрелками отметим на графе все возможные переходы из каждого состояния.

Например, из состояния «мигает зеленый» светофор может переключиться в состояние «горит желтый» или, если регулирование закончилось, в состояние «мигает желтый».

Над каждой стрелкой подпишем условие, которое должно выполниться, чтобы переход произошел. Т.е. над стрелкой от «мигает зеленый» к «горит желтый» напишем условие: «прошло 5 секунд», а над стрелкой от «мигает зеленый» к «мигает желтый» напишем условие «конец регулирования».

В итоге получим полный граф переходов конечного автомата:

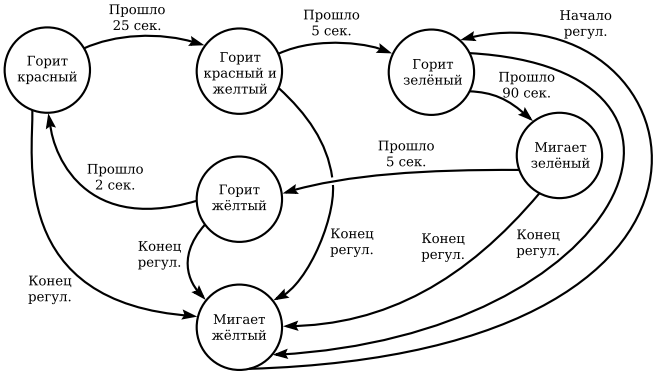


Рис. 4.3: Граф переходов конечного автомата для управления светофором

Изучите его работу. Проследите, как можно попасть в то или иное состояние.

Обратите внимание, что одни и те же входные воздействия могут приводить к разным переходам, если они возникают в разных состояниях конечного автомата.

Итак, для того чтобы реализовать в цифровой схемотехнике такое устройство, нам понадобятся регистр для хранения состояния (назовём его **«регистр состояния»**).

Также понадобится схема, которая будет принимать решение о том, какое состояние будет следующим. Посмотрите на граф: входом такой схемы должно быть текущее состояние и входные воздействия. Имея эту информацию можно определить, какое состояние будет следующим.

Регистр состояния, очевидно, синхронная схема. То есть, переход между состояниями может произойти только по положительному фронту тактового сигнала. Но как определить сам момент, когда необходимо переключиться? Ведь он зависит от входных воздействий – переход должен состояться, когда будут выполнены необходимые условия.

В такой ситуации поступают следующим образом: **регистр состояний меняет текущее состояние на следующее каждый такт вне зависимости ни от каких условий.** При этом следующее состояние, пока условия не выполнены, равно текущему. Таким образом, значение регистра не меняется,

пока не будут выполнены условия перехода.

Схема определения следующего состояния - асинхронная.

Входами этой схемы служат текущее состояние и все возможные входные сигналы (условия переходов). Выходом этой схемы будет следующее состояние.

Так как мы знаем все переходы, мы, фактически, определяем таблично заданную ФАЛ. Значит, эта схема является **дешифратором**.

Структура конечного автомата будет выглядеть следующим образом:

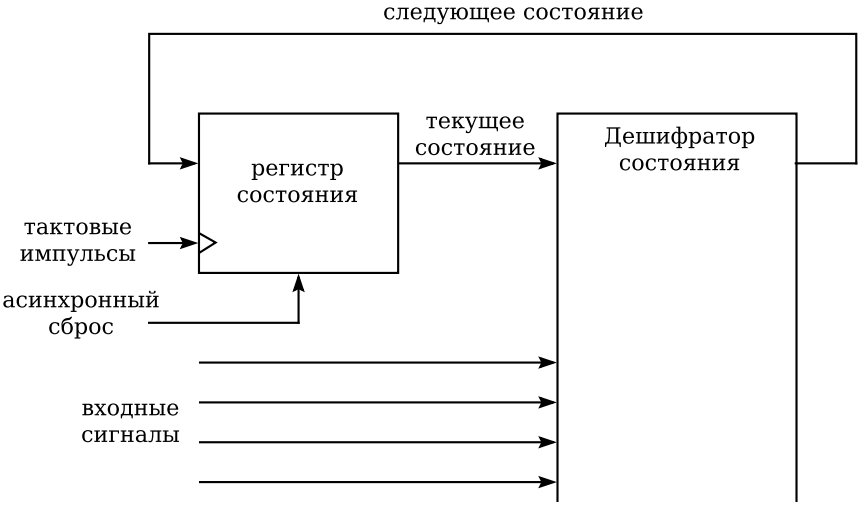


Рис. 4.4: Структура конечного автомата

Так как в составе конечного автомата присутствует триггер, **конечный автомат является синхронным цифровым устройством.** Состояния конечного автомата меняются по положительному фронту сигнала синхронизации.

Используя такую структуру в качестве основы, опишем на языке Verilog автомат, реализующий работу светофора.

```
1  ...
2
3  localparam YELLOW          = 3'b000;
```

```
 4  localparam YELLOW_BLINKING = 3'b001;
 5  localparam GREEN           = 3'b010;
 6  localparam GREEN_BLINKING  = 3'b011;
 7  localparam RED             = 3'b100;
 8  localparam RED_AND_YELLOW  = 3'b101;
 9
10  reg [2:0] state;
11
12  always @(posedge clk or posedge rst) begin
13    if (end_work) state <= YELLOW_BLINKING;
14    else begin
15      case (state)
16        YELLOW_BLINKING: if (start_work)
17          state <= GREEN;
18        GREEN: if (passed_90_seconds)
19          state <= GREEN_BLINKING;
20        GREEN_BLINKING: if (passed_5_seconds)
21          state <= YELLOW;
22        YELLOW: if (passed_2_seconds)
23          state <= RED;
24        RED: if (passed_25_seconds)
25          state <= RED_AND_YELLOW;
26        RED_AND_YELLOW: if (passed_5_seconds)
27          state <= GREEN;
28        default:
29          state <= YELLOW_BLINKING;
30      endcase
31    end
32  end
33
34  ...
```

Листинг 4.1: Конечный автомат, реализующий работу светофора

Итак, мы реализовали конечный автомат. Но как им пользоваться? Как с помощью автомата управлять работой цифровой схемы?

Возможно, вы уже догадались, что для управления работой цифрового устройства используют анализ текущего состо-

яния. Текущее состояние, в подавляющем большинстве случаев, и является выходом конечного автомата.

Анализируя состояние, можно переключать мультиплексоры, подавать (с помощью ФАЛ) необходимые входные воздействия или наборы данных, разрешать или запрещать работу блоков и модулей. Таким образом, можно существенно изменить поведение цифрового устройства.

Рассмотрим на конкретных примерах, как управлять работой светофора с помощью конечного автомата.

Управление светодиодами может осуществляться выходами компараторов.

Например, в состоянии «Горит зеленый», компаратор будет выдавать единицу, поступающую на зеленую лампочку.

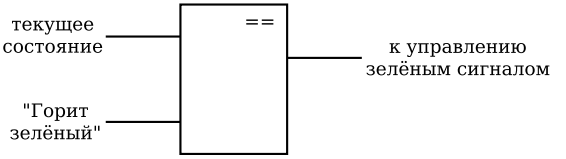


Рис. 4.5: Компаратор для управления светодиодами

В состоянии «Мигает зеленый» все немного сложнее.

Существуют разные варианты решения этой задачи.

Один из возможных – следующий: разработать схему, вырабатывающую периодические импульсы, и подключить через вентиль **И** к сигналу управления зеленой лампой светофора. При этом сам сигнал управления подавать уже не в одном, а в двух состояниях автомата.

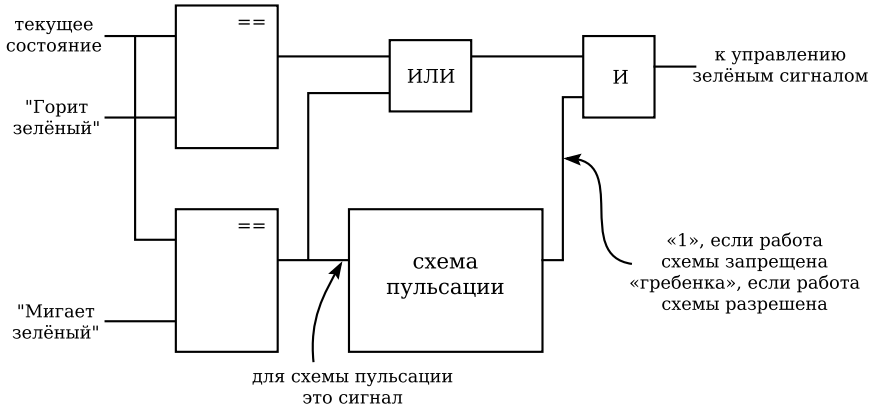


Рис. 4.6: Схема управления миганием зелёного светодиода

Схема работает следующим образом:

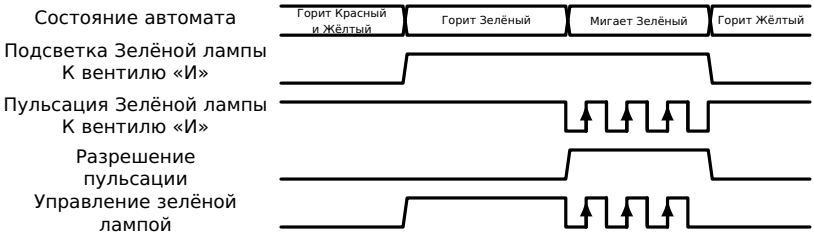


Рис. 4.7: Временная диаграмма работы схемы управления зелёным светодиодом

Схему пульсации можно использовать также для того чтобы мигать желтой лампой светофора.

Вспомним, что в цифровой схемотехнике нельзя закорачивать выходы цифровых блоков. Для того чтобы соединить выходы, необходимо использовать подходящие логические вентили.

Также вспомним, что автомат может прибывать только в одном своем состоянии.

Тогда общая схема будет выглядеть следующим образом:

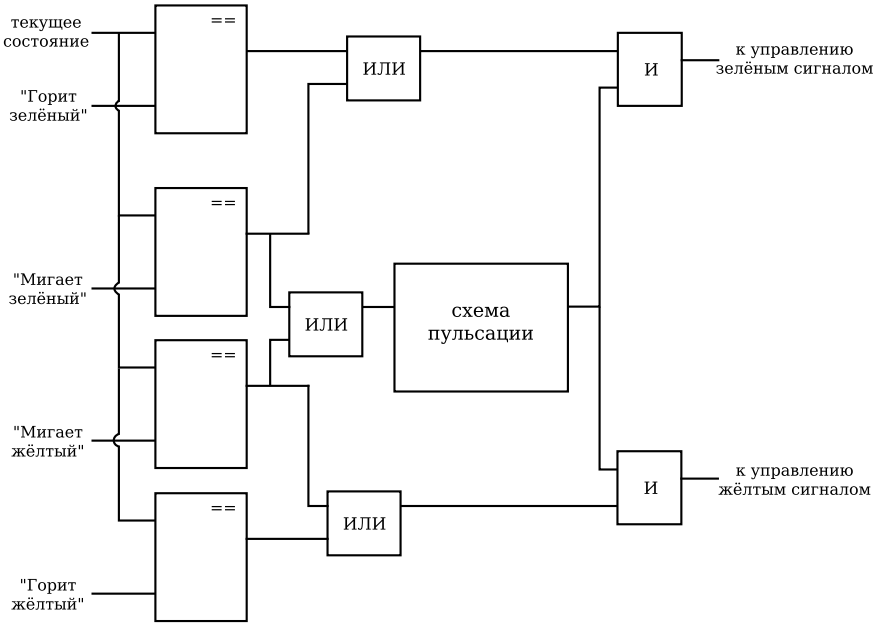


Рис. 4.8: Схема управления миганием светодиодов

Несмотря на внушительный вид это довольно простая схема:

Схема пульсации включается, когда автомат находится в состоянии «Мигает Зеленый» **ИЛИ** «Мигает Желтый».

Зеленая лампа подсвечивается, когда автомат находится в состоянии «Горит Зеленый» **ИЛИ** «Мигает Зеленый» **И** при этом выход схемы пульсации равен единице.

Для желтой лампы ситуация аналогична ситуации для зеленой лампы.

В форме поведенческого кода данная схема выглядит еще проще для понимания:

```
1  ...
2
3  wire green_light;
4  wire yellow_light;
5  wire blink_en;
```

```
 6 wire blink;
 7
 8 assign green_light  = (state == green)  |
 9                       (state == green_blinking);
10 assign yellow_light = (state == yellow) |
11                       (state == yellow_blinking);
12 assign blinking_en  = (state == green_blinking) |
13                       (state == yellow_blinking);
14
15 //код, описывающий поведение схемы пульсации
16 //выход схемы - сигнал blink
17 always @(posedge clk) begin
18   if (blinking_en = 1) then
19   ... // опустим описание
20 end
21
22
23 assign green  = green_light & blink;
24 assign yellow = yellow_light & blink;
25
26 ...
```

Листинг 4.2: Описание схемы пульсации лампы светофора

Хотелось бы сразу отметить некоторую особенность реализации конечного автомата в виде цифрового устройства. Взгляните на следующий граф переходов:

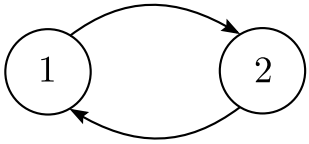


Рис. 4.9: Граф переходов конечного автомата с двумя состояниями

Несмотря на свою примитивность на его примере хорошо видно реакцию конечного автомата на внешнее воздействие.

Регистр состояния защелкивает новое состояние каждый такт, поэтому для того, чтобы автомат переключился из со-

стояния «1» в состояние «2», требуется, чтобы входное воздействие «a» имело единичную длительность.

В противном случае автомат переключится из состояния «1» в состояние «2», затем, на следующий такт автомат переключится из состояния «2» в состояние «1» и так далее, пока не закончится входное воздействие.

Сравните временные диаграммы ниже, чтобы лучше почувствовать разницу.

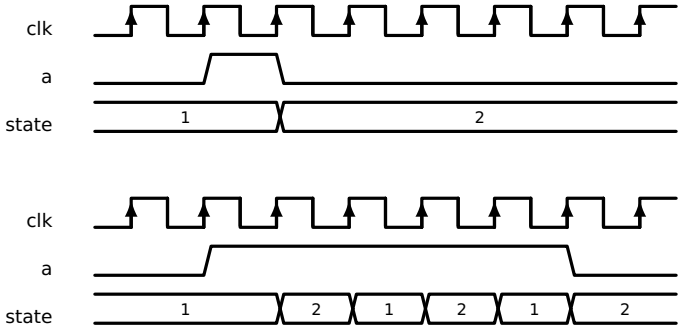


Рис. 4.10: Временные диаграммы конечного автомата с различной продолжительностью входного воздействия

Кроме разделения режимов работы конечные автоматы позволяют цифровым устройствам «выполнять» некоторые алгоритмы действий. Ведь мы можем и не ставить условий для перехода из некоторых состояний в другие. Посмотрите на граф ниже - он очень напоминает алгоритм программы.

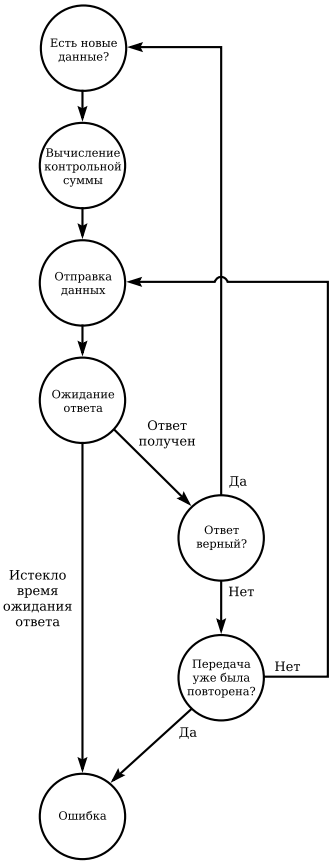


Рис. 4.11: Граф переходов, показывающий алгоритм выполнения программы

Мы вполне можем описать автомат, реализующий этот граф переходов:

```
1  ...
2
3  localparam IDLE = 3'b000;
4  localparam CALC_CHECKSUM = 3'b001;
5  localparam SEND_DATA = 3'b010;
6  localparam WAIT_ANSWER = 3'b011;
```

```
 7  localparam ANALYSE_ANSWER = 3'b100;
 8  localparam TRY_SECOND_TIME = 3'b101;
 9  localparam ERROR = 3'b110;
10
11  reg [2:0] state;
12
13
14  always @(posedge clk or posedge rst) begin
15      case (state)
16        IDLE:
17          if (new_data)
18            state <= calc_cheksum;
19
20        calc_cheksum:
21          if (checksum_calc_complete)
22            state <= SEND_DATA;
23
24        SEND_DATA:
25          state <= WAIT_ANSWER;
26
27        WAIT_ANSWER:
28          if (answer_recived)
29            state <= ANALYSE_ANSWER;
30          else if (wait_too_long)
31            state <= TRY_SECOND_TIME;
32
33        ANALYSE_ANSWER:
34          if (answer_is_ok)
35            state <= IDLE;
36          else
37            state <= TRY_SECOND_TIME;
38
39        TRY_SECOND_TIME:
40          if (already_tried)
41            state <= ERROR;
42          else
43            state <= SEND_DATA;
44
45        ERROR:
46          if (reset_error)
```

```
47              state <= ERROR;
48
49          default: state <= ERROR;
50        endcase
51  end
52
53
54  ...
```

Листинг 4.3: Описание конечного автомата для приведенного графа переходов

Отметим некоторые особенности таких автоматов. Обратите внимание на условия переходов. Условия присутствуют практически во всех состояниях, даже если переход между ними линеен. Например, переход из состояния «calc_checksum» в состояние «send_data». Если для вычисления контрольной суммы требуется более одного такта, то без проверки выполнена ли операция нельзя покидать состояние вычисления контрольной суммы.

Разберем структуру цифрового устройства, которым будет управлять приведенный выше конечный автомат.

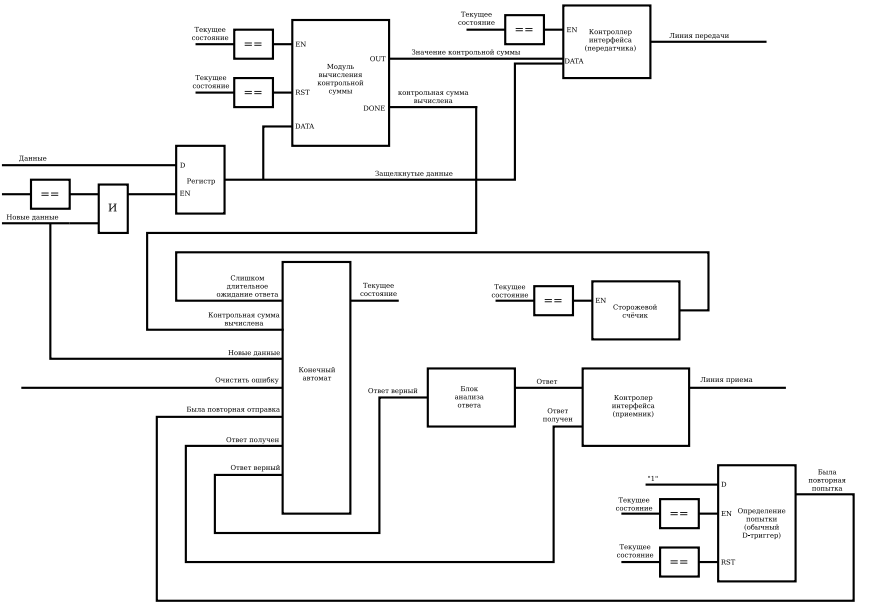


Рис. 4.12: Структура устройства для обмена данными

Как разработать подобную структуру?

Первое, что необходимо понимать, что блоки цифровых устройств существуют вне зависимости от того, используются ли они в данный момент или нет.

Теперь мы постараемся разбить устройство на блоки, определить функционал каждого из этих блоков и соединить их между собой.

Начинается разработка схемы с конечного автомата.

Наш конечный автомат уже разработан. Его мы помещаем в центр будущей структуры. Выходом конечного автомата служит текущее состояние. Именно его анализируют все остальные блоки устройства.

Теперь, когда мы поместили конечный автомат, приступим к проектированию периферии. Сначала регистр для того, чтобы защелкнуть и сохранить данные, которые в дальнейшем будем обрабатывать и передавать.

Когда необходимо защелкнуть данные? В самом начале ра-

боты. Т.е. в состоянии «idle», в момент, когда на вход пришел сигнал «new_data».

Идем дальше по алгоритму работы автомата. Теперь автомат переключился в состояние «calc_checksum». В структуру устройства необходимо добавить модуль, который будет считать контрольную сумму. Этот модуль будет производить вычисления, только когда автомат находится в состоянии «calc_checksum». После окончания вычисления контрольная сумма должна быть на выходе этого модуля и больше не должна меняться. Также этот модуль после завершения вычисления контрольной суммы должен выработать признак «checksum_calc_complete» для конечного автомата. Контрольную сумму можно сбросить в состоянии «idle».

Теперь автомат переходит в состояние «send_data». Для отправки уже должен быть сформирован пакет из данных и контрольной суммы (например, в виде объединенных шин).

Чаще всего контроллеры работают следующим образом. По сигналу разрешения записи данные с входа помещаются во внутренний регистр и, затем, начинается их отправка.

Именно поэтому мы использовали состояние, в котором наш конечный автомат проводит только один такт. Именно в этом состоянии происходит загрузка и запуск передатчика, а именно выработка сигнала «разрешение записи» для него.

Далее автомат попадает в состояние ожидание ответа. Для получения ответа, нам понадобится модуль приемника. Модуль приемника будет выставлять на шину значение принятых данных, а также он должен будет вырабатывать сигнал «answer_received» для работы конечного автомата.

Для функционирования конечного автомата нам также понадобится сигнал «waiting_too_long», который сигнализирует об отсутствии ответа в течение слишком большего времени. Чтобы выработать такой сигнал, поместим в устройство счётчик, который будет работать все время, пока автомат находится в состоянии «waiting_for_answer». Счётчик, при достижении максимального значения (которое мы зададим самостоятельно) будет вырабатываться сигнал «waiting_too_long». Обычно его называют «сторожевой счётчик».

Итак, если ответ получен, то его надо обработать. Мы раз-

работали конечный автомат так, что обработка должна произойти за один такт, ведь в состоянии «analyse_answer» нет условий, говорящих о конце обработки. Внесем в устройство модуль, анализирующий ответ, полученный модулем приемника.

Осталось только одно состояние, в котором мы не определили входные данные для конечного автомата. Это состояние «try_second_time». Нам понадобится схема, вырабатывающая сигнал «already_tried».

Это довольно простая схема: регистр, на вход которого подается единица. Регистр сбрасывается в состоянии «idle», а запись в него разрешена только в состоянии «try_second_time». Выход этого регистра и есть «already_tried».

Когда мы попадаем в состояние «try_second_time», мы видим, что в регистре записан «0», и переходим в состояние «send_data» для повторной отправки данных. В этот же момент времени происходит запись «1» в регистр. Теперь, если мы снова окажемся в состоянии «try_second_time», мы увидим в регистре «1» и не будем осуществлять повторную отправку данных.

Таким образом, мы закончили описание структуры нашего цифрового устройства. Мы знаем, из каких модулей оно должно состоять и как должен функционировать каждый из этих модулей.

В рамках данной лабораторной работы нас не интересует конкретная реализация модулей. Но стоит заметить, что начав описание тех или иных из них, разработчик может столкнуться с необходимостью добавить новые управляющие сигналы, ввести новые состояния конечного автомата или добавить новые связи.

Эти случаи вовсе не редкость. При проектировании цифровых устройств, практически каждый разработчик неоднократно корректирует изначально разработанную им структуру. Как правило, по мере получения опыта уменьшается количество подобных правок.

Вы увидели, как разрабатывается структура цифровых устройств, включающих конечные автоматы.

Вы убедились, что структуры, необходимые для реализации

тех или иных действий существуют одновременно, поэтому исполнение некоторых алгоритмов может быть связано с очень большими аппаратными затратами.

В таких случаях используют другие подходы к построению цифровых устройств, например, использование вычислительных ядер с программным управлением, с устройством которых вы познакомитесь в курсе «Микропроцессорные системы и средства».

Тем не менее, выполнение простых алгоритмов, например, алгоритмов управления с малым количеством ветвлений, часто перекладывается на конечные автоматы с целью минимизации программного кода и ускорения работы.

4.1 Задание лабораторной работы

1. Изучить разработку к лабораторной работе.
2. Реализовать конечный автомат, согласно индивидуальному заданию.
3. Ответьте на вопросы к защите лабораторной работы.

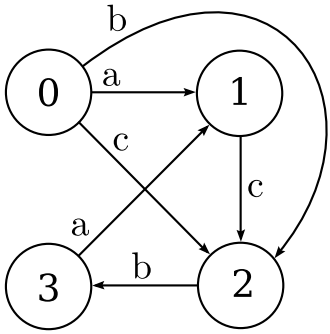
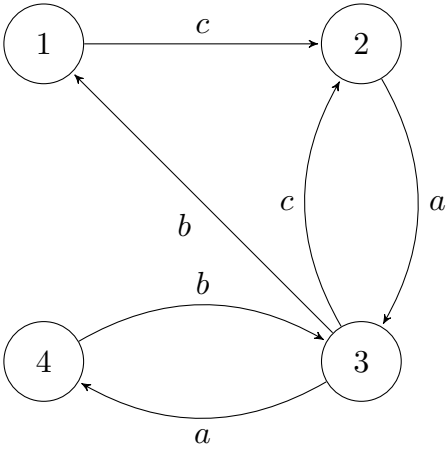


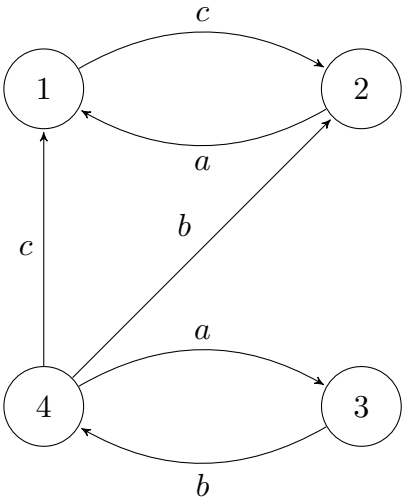
Рис. 4.13: Пример задания лабораторной работы

4.2 Варианты индивидуальных заданий

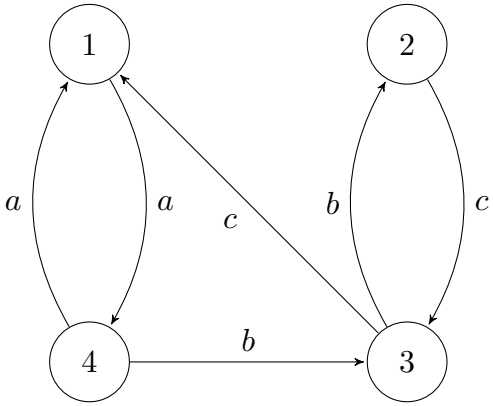
1. Граф конечного автомата:



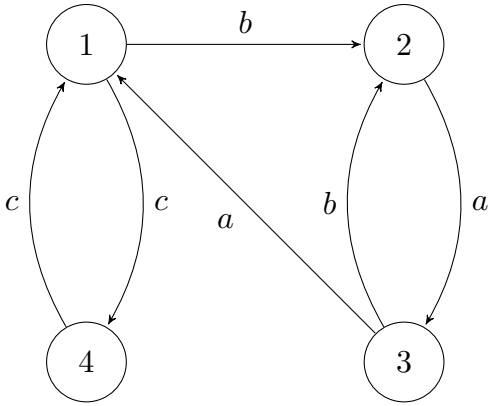
2. Граф конечного автомата:



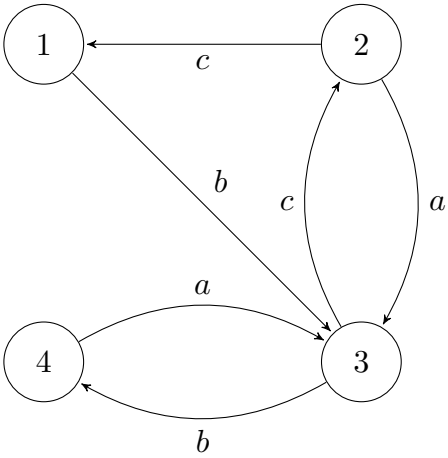
3. Граф конечного автомата:



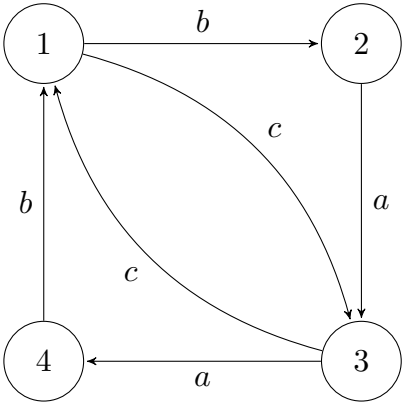
4. Граф конечного автомата:



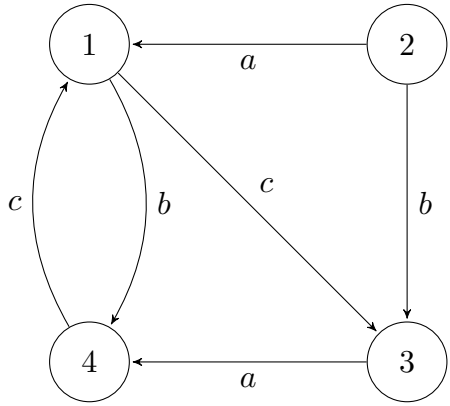
5. Граф конечного автомата:



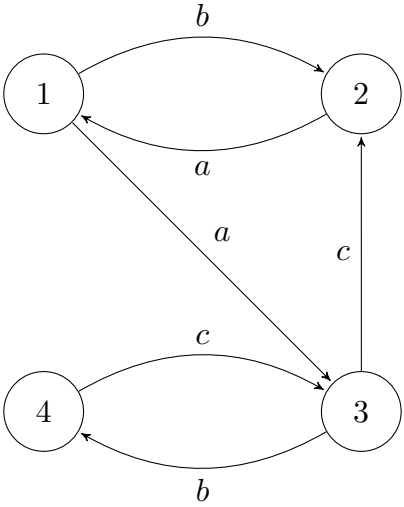
6. Граф конечного автомата:



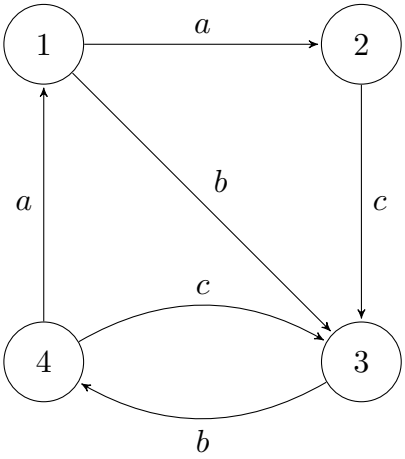
7. Граф конечного автомата:



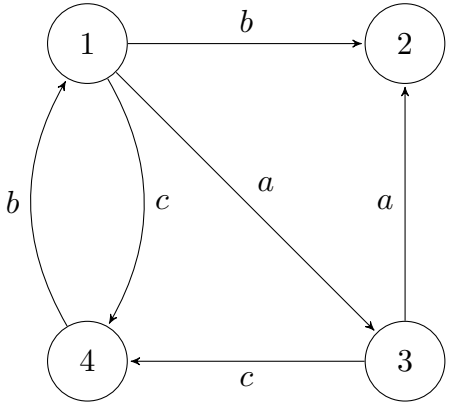
8. Граф конечного автомата:



9. Граф конечного автомата:



10. Граф конечного автомата:



4.3 Вопросы к защите лабораторной работы

1. Что такое конечный автомат?
2. Для чего конечные автоматы используются в цифровых устройствах?
3. Из каких цифровых блоков состоит конечный автомат?
4. Конечный автомат это синхронное или асинхронное устройство?
5. Работу какого цифрового блока конечного автомата определяет граф переходов?
6. Почему для верного функционирования конечного автомата важна длительность управляющих сигналов?
7. Изучите рисунок структуры цифрового устройства на стр.10. Назовите значение состояния автомата, анализируемое каждым из компараторов.

Лабораторная работа №5 RAM-память

Мы уже познакомились с многими блоками, из который состоят цифровые устройства. Последний базовый «строительный» блок, который мы рассмотрим в этом курсе — память с произвольным доступом (Random Access Memory).

RAM память в цифровых устройствах делится на три типа:

- Статическая память
- Динамическая память
- Энергонезависимая память

Статическая память по принципу работы похожа на набор регистров. Данные, которые были записаны в память, хранятся в ней, пока на цифровое устройство подается питание. Когда питание отключают, данные стираются из памяти.

Основным запоминающим элементом этого типа памяти является защелка. Такая память требует довольно большого количества ресурсов для своей реализации.

Динамическая память использует другой принцип хранения информации. В качестве элемента памяти в динамической памяти используются конденсаторы. Когда конденсатор заряжен — ячейка хранит «1». Когда разряжен — «0». Эта схема гораздо проще в реализации и требует существенно меньше ресурсов по сравнению со статической памятью. А значит на одном кристалле можно разместить большее количество памяти.

Но конденсатор неизбежно со временем разряжается через сопротивления утечки. Поэтому динамическая память требует постоянного обновления данных, которые были туда записаны. Обновление происходит благодаря внутренне-

му контроллеру, который проходит по всем адресам в памяти, считывая данные, а затем записывая эти же данные обратно и таким образом «обновляет» заряд конденсаторов.

Минусом динамической памяти является большое потребление тока, по сравнению со статической.

Наиболее известным представителем энергонезависимой памяти на данный момент является Flash память. Flash память базируется на транзисторах с плавающим затвором.

Выделяют два вида Flash памяти — NOR и NAND. Последняя получила наибольшее настроение.

У Flash памяти есть недостатки:

- высокая технологическая сложность и, соответственно, стоимость
- деградация ячеек памяти, при повторной перезаписи информации (сотни тысяч операций записи для коммерческих продуктов, миллионы - 2012 год и сотни миллионов - 2014 год)
- деградация ячеек памяти при чтении (аналогично записи)
- блочная организация (удалить можно только блока информации целиком)

В рамках нашего курса мы познакомимся со статической памятью и Flash-памятью, как наиболее часто применяющейся в качестве встроенной в цифровых устройствах.

Возможно, вы уже знаете основные принципы функционирования памяти в цифровой технике. Тем не менее, напомним, каким образом работает RAM память.

Сначала разберем, какие входы и выходы необходимы для полноценной работы памяти. ***Обратите внимание, что память является синхронным устройством и требует сигнала синхронизации***:

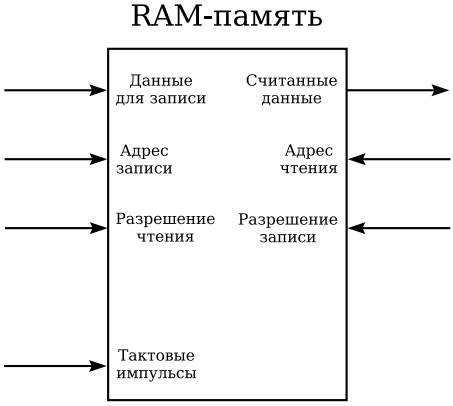


Рис. 5.1: Входные и выходные сигналы RAM-памяти

Как происходит запись данных в память и чтение из неё?

На вход «данные для записи» подаются данные, которые мы хотели бы записать в память. Одновременно с этим на вход «адрес записи» подается адрес ячейки, в которую мы хотели поместить входные данные. Запись происходит по положительному фронту сигнала синхронизации, когда вход «запись разрешена» находится в «1».

Посмотрите на временную диаграмму записи данных в память:

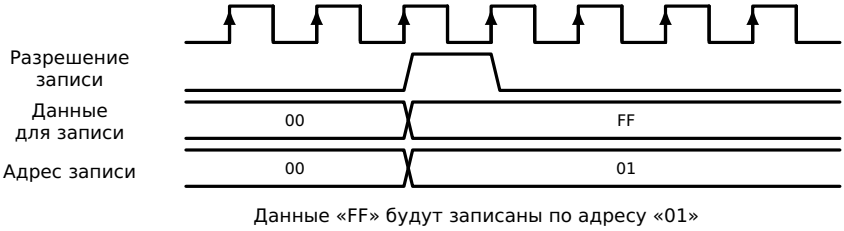


Рис. 5.2: Временная диаграмма записи в память

Чтение из RAM памяти происходит следующим образом: на вход «адрес чтения» подается адрес ячейки, из которой мы хотим получить данные, затем на вход «разрешение чтения» подается «1». По положительному фронту сигнала синхронизации данные выгружаются из памяти и становятся доступны для чтения:

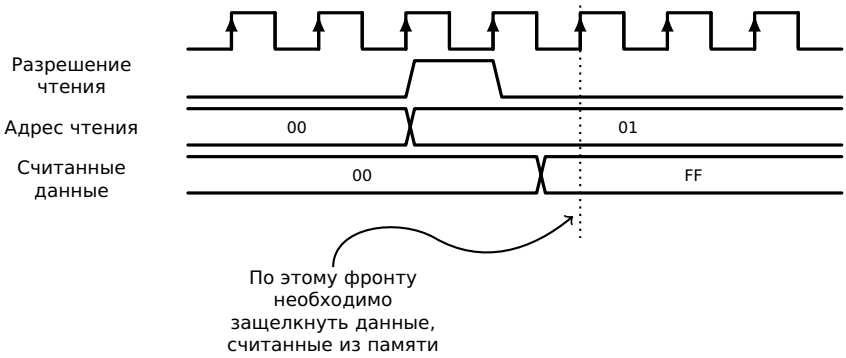


Рис. 5.3: Временная диаграмма чтения из памяти

Обратите внимание, что как в случае записи, так и в случае чтения, управляющие сигналы «запись разрешена» и «чтение разрешено» для однократной операции записи или чтения должны иметь длительность равную одному такту.

Посмотрите на временную диаграмму, демонстрирующую запись большого количества данных в память:

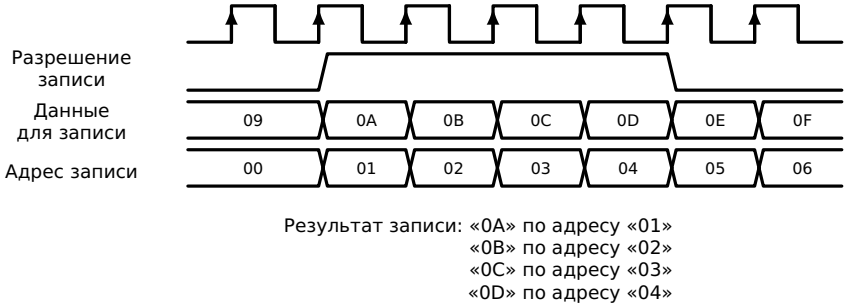


Рис. 5.4: Временная диаграмма последовательной записи данных

Обратите внимание на смещение считанных данных при последовательном чтении информации из памяти:

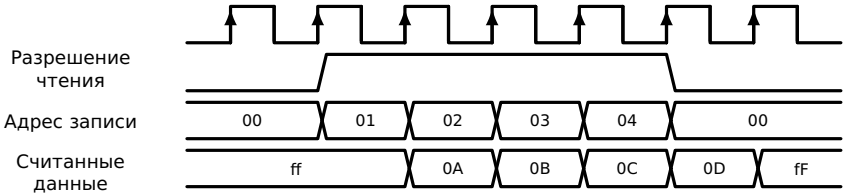


Рис. 5.5: Временная диаграмма последовательного чтения из памяти

Как же устроена статическая RAM-память?

Общую структуру статической памяти можно представить следующим образом:

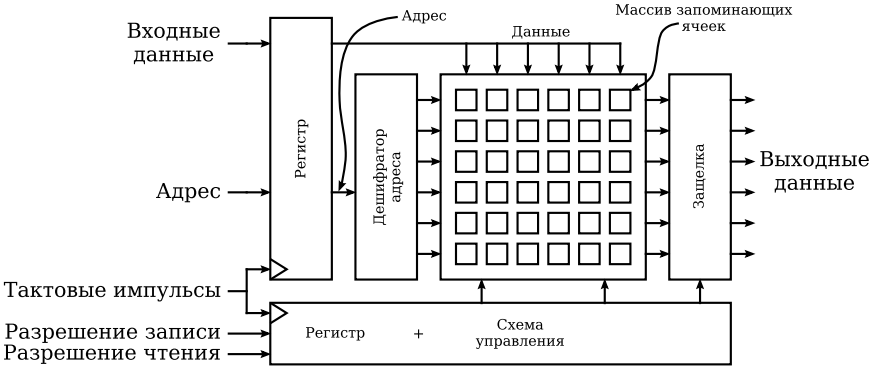


Рис. 5.6: Общая структура статической памяти

Как мы уже говорили выше, использование триггеров и регистров для хранения большого количества данных оказалось не эффективно. RAM память строится на основе массива запоминающих блоков.

Обычно каждый бит данных подается на столбец, а ячейка столбца, в которую произойдет запись, выбирается в зависимости от адреса. Т.е. нулевой бит, всегда подается на нулевой столбец, и, в зависимости от адреса, нулевой бит будет записан в конкретную ячейку нулевого столбца.

Чтение происходит подобным образом: адрес «выбирает» по одной ячейке из каждого столбца и из значений этих ячеек формируется вектор выходных данных.

Для того чтобы обеспечить управление работой массива памяти в ней присутствует схема управления и входной регистр для фиксации входных данных и адреса.

Обратите внимание, что схема управления тоже имеет регистр для хранения сигналов разрешения записи и разрешения чтения.

Данные, считанные из памяти поступают на защелку, поведение которой (разрешен ли приём или выбран режим хранения) также контролируется схемой управления.

На основе RAM памяти строятся некоторые другие виды памяти.

Из них очень часто в цифровых устройствах встречается буфер. Буферный блок представляет собой память, организованную следующим образом: новые данные всегда записываются в «конец» памяти, а считываются всегда из её «начала».

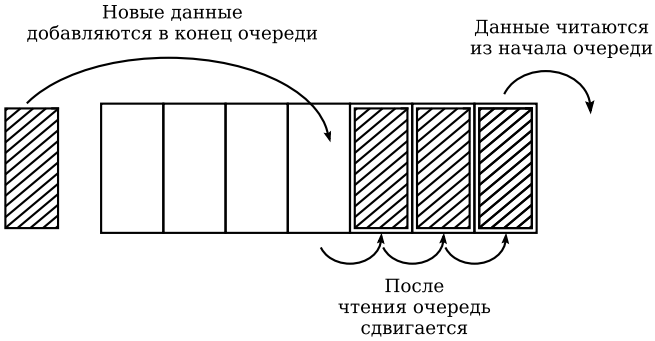


Рис. 5.7: Организация записи данных в буфер

В англоязычной литературе такой блок памяти носит название FIFO от словосочетания First In First Out – Первый Пришел Первый Ушел. Название FIFO настолько широко распространено, что оно практически вытеснило русское название «очередь».

Так как данные всегда записываются в конец очереди, а читаются всегда из её начала, то интерфейс FIFO не содержит адресной шины.

В основе FIFO, как мы уже говорили, лежит RAM память, а

очередь организуется за счет управляющего блока, специально подготавливающего адреса чтения и записи.

Запись происходит с нулевого адреса. После помещения информации в память, управляющий блок увеличивает адрес на единицу. Чтение происходит точно также, начиная с нулевого адреса. Когда происходит чтение, адрес чтения также увеличивается на единицу. Управляющему блоку необходимо следить, чтобы адрес чтения не «перегонял» адрес записи.

Все ячейки, адреса которых находятся «между» адресом чтения и адресом записи — заполнены, а все ячейки, адреса которых находятся «снаружи» этих адресов — пусты.

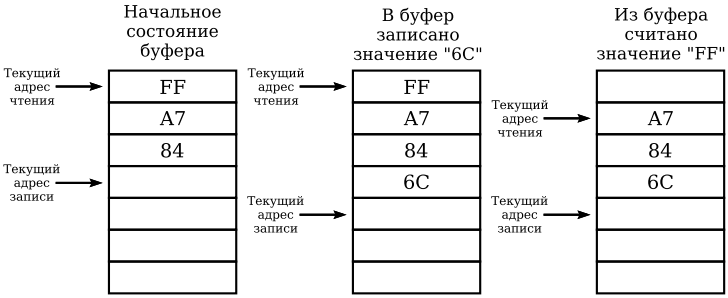


Рис. 5.8: Порядок чтения и записи в память

Также управляющий блок вырабатывает сигналы, описывающие состояние очереди: «пуста» - означает, что в очереди нет данных и «полна» - означает, что в очереди нет места для новых данных.

Подумайте, каким образом устройство управления может выработать эти сигналы?

Также часто в FIFO добавляют дополнительные признаки: «почти пуста» - означает, что в очереди только одно значение и «почти полна» - означает, что есть только одна свободная ячейка. Эти сигналы в некоторых случаях облегчают работу с FIFO.

С точки зрения управления записью и чтением, FIFO работает также, как и блок RAM памяти, т.е. требует сигналов разрешения записи и разрешения чтения.

FIFO часто применяют в качестве буфера, для того, что-

бы сохранить быстро поступающую информацию и, в дальнейшем, обработать её. Такой блок часто можно встретить в контроллерах интерфейсов, где обмен данными часто происходит «волнами».

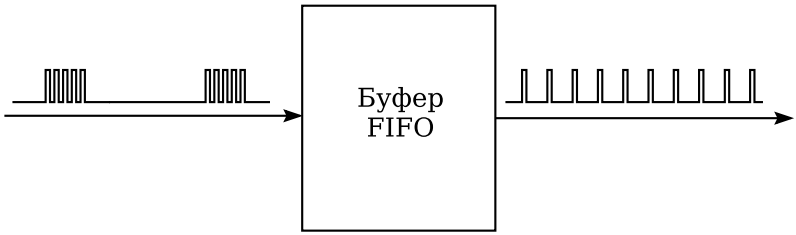


Рис. 5.9: Запись пакетов данных в FIFO

В цифровых устройствах для управления чтением из памяти и записью в неё часто используются конечные автоматы.

Продемонстрируем решение типовой задачи: спроектируем конечный автомат, задачей которого будет управление FIFO и передатчиком некоторого интерфейса. Автомат будет по готовности передатчика при наличии данных в FIFO выгружать из него новое значение и инициировать его передачу. Таким образом, мы реализуем буфер передачи.

Вот структурная схема устройства, которое мы хотим реализовать:

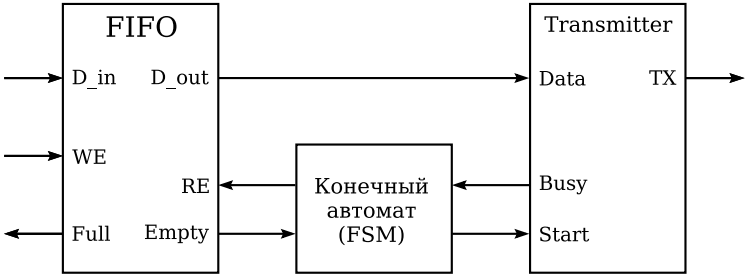


Рис. 5.10: Структура устройства управления передачей данных

Граф для конечного автомата будет выглядеть следующим образом:

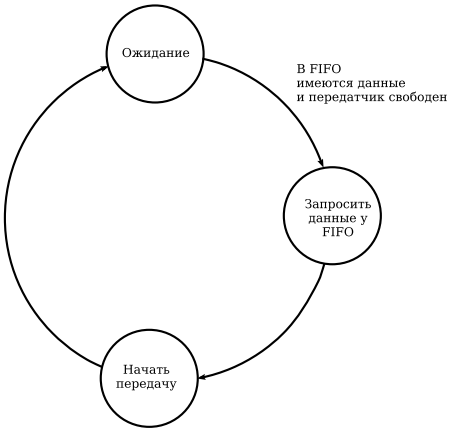


Рис. 5.11: Граф работы конечного автомата

В состоянии «ожидание» автомат ждёт готовности передатчика и появления данных в FIFO. Затем, когда данные появляются и передатчик свободен, автомат переходит в состояние «запросить данные у FIFO», затем сразу же в состояние «Начать передачу» и, затем, снова в состояние «ожидание».

Временная диаграмма будет выглядеть следующим образом:

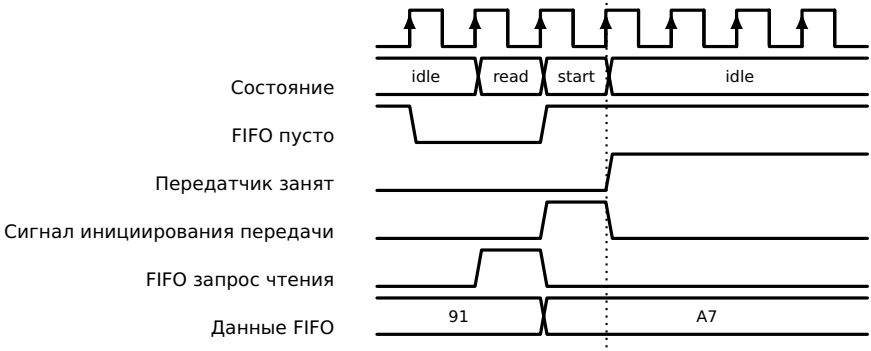


Рис. 5.12: Временная диаграмма последовательного чтения из памяти

Внимательно изучите временную диаграмму. Обратите внимание на то, каким образом спроектирован конечный ав-

томат. Отметьте, что сигнал инициирования передачи совпадает с состоянием «start», а сигнал запроса чтения для FIFO совпадает с состоянием «read». Также заметьте, что инициирование передачи происходит как раз в тот момент, когда данные с FIFO уже готовы для чтения.

Приступим к описанию нашего модуля, используя FIFO и передатчик, в качестве компонентов.

```
 1  module example_lab6 (
 2    input clk,
 3    input rst,
 4    input [7:0] data,
 5    input we,
 6    output full,
 7    output transmit_lane);
 8
 9  localparam idle = 2'b00;
10  localparam load = 2'b01;
11  localparam transmit = 2'b10;
12  localparam wait_trasnaction_to_complete = 2'b11;
13
14  reg [1:0] state;
15  wire [7:0] data_from_fifo_for_transmitter;
16  wire fifo_is_empty;
17  wire fifo_re;
18  reg transmitter_is_busy;
19  reg start_transaction;
20
21  always @(posedge clk) begin
22    case (state) is
23      idle:
24        if (fifo_is_empty |
25            transmitter_is_busy)
26          state <= idle;
27        else state <= load;
28      load: state <= transmit;
29      transmit: state <= idle;
30    endcase
31  end
32
```

```
33  assign fifo_re = (state == load);
34  assign start_transaction = (state == transmit);
35
36  fifo fifo_input_buffer(
37    .we(we),
38    .re(fifo_re),
39    .data_in(data),
40    .data_out(data_from_fifo_for_transmitter),
41    .empty(fifo_is_empty),
42    .full(full)
43  );
44
45  tansmitter my_transmitter(
46    .start(start),
47    .busy(busy),
48    .data(data_from_fifo_for_transmitter),
49    .tx(transmit_lane)
50  );
51
52  endmodule
```

Листинг 5.1: Описание проектируемого устройства на языке Verilog

5.1 Задание лабораторной работы:

1. Изучить разработку к лабораторной работе.
2. Разработать цифровое устройство, функционирующее согласно следующим принципам:
3. Нажатие кнопки приводит к увеличению текущего значения счётчика на единицу. Одновременно с этим текущее значение счётчика должно быть записано в буфер FIFO. Если в FIFO есть данные, то их выгрузка должна производиться один раз в секунду (одно слово в секунду). Выгруженное значение должно отображаться на семисегментных индикаторах в шестнадцатеричной форме.
4. Провести моделирование работы данного цифрового устройства и продемонстрировать результат.

5. Получить файл конфигурации для ПЛИС учебного стенда и продемонстрировать работу устройства.

5.2 Вопросы к защите лабораторной работы

1. Что такое RAM-память?
2. Изобразите обобщенную структуру RAM-памяти.
3. RAM-память это синхронное или асинхронное устройство?
4. Опишите все входные и выходные сигналы RAM памяти, известные вам.
5. Нарисуйте временную диаграмму записи значения в RAM память.
6. Нарисуйте временную диаграмму чтения значения из RAM памяти.
7. Как функционирует буфер FIFO?
8. Опишите все входные и выходные сигналы FIFO памяти, известные вам.

Лабораторная работа №6 Контроллер PS/2 для клавиатуры

Для успешного применения цифровых устройств в реальности важно обеспечить их взаимодействие с внешним миром. Для этого существуют внешние интерфейсы для обмена информацией с другими устройствами и приборами. Внешние интерфейсы имеют различное функциональное назначение и спецификацию. Например, раньше для подключения клавиатуры и мыши использовался интерфейс PS/2. Один с самых простых способов подключения устройств ввода-вывода.

Давайте изучим протокол и реализацию интерфейса PS/2.

Интерфейс PS/2 сточки зрения соединения представляет собой два провода **Clock** и **Data**. По **Clock** передаются синхроимпульсы, а по **Data** предаются данные. На рисунке пример одной транзакции обмена.

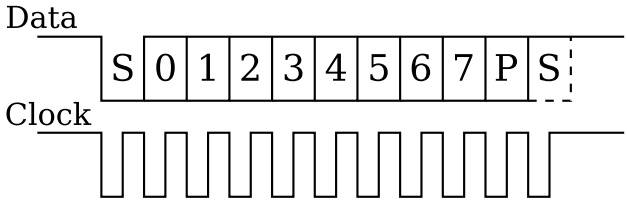


Рис. 6.1: Передача данных по протоколу PS/2

Структура транзакции похожа на UART и состоит из:

1. старт бит – всегда ноль;
2. 8 бит данных;
3. бит четности, равен 1 если количество единиц в данных четно и 0 если нечетно;

4. стоп бит – всегда единица.

Данные на линию выставляются, когда **Clock** равен **1** и считываются, когда **Clock** равен **0**. На практике данные обычно выставляются по положительному фронту и считываются по отрицательному.

Частота сигнала **Clock** примерно 10-16.7кГц. Время от фронта сигнала **Clock** до момента изменения сигнала **Data** не менее 5 микросекунд.

Транзакции разделяются на два вида:

1. От устройства к контроллеру;
2. От контроллера к устройству.

На примере клавиатуры.

1. Клавиатура посылает на контроллер 8 битный код нажатой клавиши;
2. Контроллер посылает на клавиатуру команды управления. Такие как, команды сброса, выключения светодиодов.

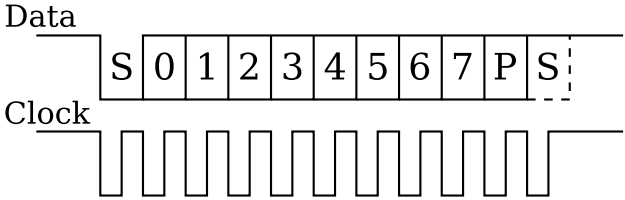


Рис. 6.2: Передача 8 битного пакета данных

При транзакции от устройства (клавиатуры) к контроллеру сигналы на линиях **Clock** и **Data** генерирует устройство. Контроллер выступает в роли приемника считывая данные по отрицательному фронту **Clock**.

При передаче в обратную сторону команд от контроллера к клавиатуре или мыши протокол отличается от описанного выше.

Последовательность обмена другая:

1. контроллер опускает сигнал **Clock** в ноль на время примерно 100 микросекунд;
2. контроллер опускает сигнал **Data** в ноль формируя старт

бит;
3. контроллер отпускает сигнал **Clock** в логическую единицу, клавиатура фиксирует старт бит;
4. далее клавиатура генерирует сигнал **Clock**, а хост контроллер подает передаваемые биты;
5. после того, как контроллер передал все свои биты, включая бит четности и стоп бит, клавиатура посылает последний бит «ноль», который является подтверждением приема.

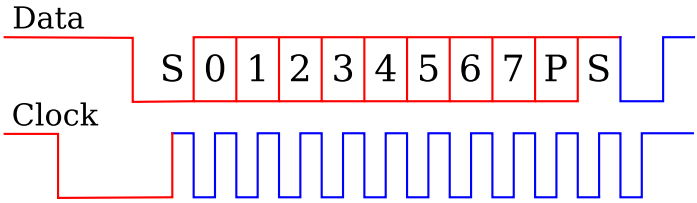


Рис. 6.3: Распределение управления между контроллером и устройством

Поскольку одним сигналом управляют два устройства, то довольно трудно понять, кто в какой момент времени управляет сигналом. По этому, диаграмма нарисована двумя цветами. Красный цвет – сигнал управляется контроллером, а синий – сигнал управляется устройством.

Давайте выясним какие коды же коды передает клавиатура для каждой клавиши.

Ниже приведена таблица кодов для клавиш. Каждой клавише соответствует код генерируемый при нажатии и код генерируемый при деактивации. Коды состоящие из нескольких байтов предаются в нескольких подряд идущих транзакций.

KEY	**MAKE**	**BREAK**	**KEY**	**MAKE**	**BREAK**
A	1C	F0,1C	R ALT	E0,11	E0,F0,11
B	32	F0,32	APPS	E0,2F	E0,F0,2F
C	21	F0,21	ENTER	5A	F0,5A
D	23	F0,23	ESC	76	F0,76
E	24	F0,24	F1	05	F0,05
F	2B	F0,2B	F2	06	F0,06

KEY	MAKE	BREAK	KEY	MAKE	BREAK
G	34	F0,34	F3	04	F0,04
H	33	F0,33	F4	0C	F0,0C
I	43	F0,43	F5	03	F0,03
J	3B	F0,3B	F6	0B	F0,0B
K	42	F0,42	F7	83	F0,83
L	4B	F0,4B	F8	0A	F0,0A
M	3A	F0,3A	F9	01	F0,01
N	31	F0,31	F10	09	F0,09
O	44	F0,44	F11	78	F0,78
P	4D	F0,4D	F12	07	F0,07
Q	15	F0,15	SCROLL	7E	F0,7E
R	2D	F0,2D	[	54	FO,54
S	1B	F0,1B	INSERT	E0,70	E0,F0,70
T	2C	F0,2C	HOME	E0,6C	E0,F0,6C
U	3C	F0,3C	PG UP	E0,7D	E0,F0,7D
V	2A	F0,2A	DELETE	E0,71	E0,F0,71
W	1D	F0,1D	END	E0,69	E0,F0,69
X	22	F0,22	PG DN	E0,7A	E0,F0,7A
Y	35	F0,35	UP	E0,75	E0,F0,75
Z	1A	F0,1A	LEFT	E0,6B	E0,F0,6B
0	45	F0,45	DOWN	E0,72	E0,F0,72
1	16	F0,16	RIGHT	E0,74	E0,F0,74
2	1E	F0,1E	NUM	77	F0,77
3	26	F0,26	KP /	E0,4A	E0,F0,4A
4	25	F0,25	KP *	7C	F0,7C
5	2E	F0,2E	KP -	7B	F0,7B
6	36	F0,36	KP +	79	F0,79
7	3D	F0,3D	KP EN	E0,5A	E0,F0,5A
8	3E	F0,3E	KP .	71	F0,71
9	46	F0,46	KP 0	70	F0,70
‘	0E	F0,0E	KP 1	69	F0,69
-	4E	F0,4E	KP 2	72	F0,72
=	55	F0,55	KP 3	7A	F0,7A
	5D	F0,5D	KP 4	6B	F0,6B
BKSP	66	F0,66	KP 5	73	F0,73
SPACE	29	F0,29	KP 6	74	F0,74

KEY	MAKE	BREAK	KEY	MAKE	BREAK
TAB	0D	F0,0D	KP 7	6C	F0,6C
CAPS	58	F0,58	KP 8	75	F0,75
L SHFT	12	FO,12	KP 9	7D	F0,7D
L CTRL	14	FO,14	]	5B	F0,5B
L GUI	E0,1F	E0,F0,1F	;	4C	F0,4C
L ALT	11	F0,11	’	52	F0,52
R SHFT	59	F0,59	,	41	F0,41
R CTRL	E0,14	E0,F0,14	.	49	F0,49
R GUI	E0,27	E0,F0,27	/	4A	F0,4A

Разработаем простой контроллер для клавиатуры. Контроллер предназначается для работы только в режиме устройство-контроллер.

Для начала определим функционал контроллера и набор сигналов.

Функционал:

1. Считываются данные с линии PS/2 по отрицательному фронту синхросигнала;
2. Проверяется корректность принятых данных. Проверяется стоп бит, бит четности и стартовый бит;
3. Выводит принятые данные на внешнюю шину и формирует сигнал готовности данных.

Набор сигналов:

1. **areset** – асинхронный ресет, активный уровень **1**;
2. **Data** – 8 битная шина данных;
3. **valid_Data** – сигнал готовности данных, равен **1** с момента завершения приема по PS/2 до начала следующей транзакции;
4. **ps2_clk** – тактовая линия PS/2, является внешним сигналом для ПЛИС;
5. **ps2_dat** – сигнальная линия PS/2, является внешним сигналом для ПЛИС.

Примечание. Сигналы **ps2_clk** и **ps2_dat** как внешние сигналы необходимо подключить к соответствующим пинам ПЛИС. Это пины H15(**ps2_clk**) J14(**ps2_dat**), к которым на плате DE1 подключен коннектор PS/2.

```
1  module ps2_keyboard(
2      input             areset,
3      input             ps2_clk,
4      input             ps2_dat,
5      output  reg       valid_data,
6      output     [7:0]  data);
```

Листинг 6.1: Описание входов и выходов контроллера

Основой контроллера будет конечный автомат со следующей последовательностью действий:

1. Состояние покоя;
2. Прием стартового бита и его проверка. Если стартовый бит не равен 0 переходим в состояние покоя;
3. Прием данных;
4. Прием бита четности и стопового бита их проверка.
5. При правильных значениях стопового бита и бита четности формирования сигнала готовности данных.
6. Переход в состояние покоя.

```
1  reg [1:0] state;
2
3  localparam IDLE                  = 2'd0;
4  localparam RECEIVE_DATA          = 2'd1;
5  localparam CHECK_PARITY_STOP_BITS = 2'd2;
6
7
8    always @(negedge ps2_clk or posedge areset)
9    if(areset)
10     state <= IDLE;
11   else
12   begin
13    case (state)
14    IDLE:
15     begin
16         if(!ps2_dat)
17           state = RECEIVE_DATA;
18     end
19    RECEIVE_DATA:
20     begin
```

```
21        if (count_bit == 8)
22          state =
23           CHECK_PARITY_STOP_BITS;
24      end
25     CHECK_PARITY_STOP_BITS:
26      begin
27        state = IDLE;
28      end
29    default:
30      begin
31        state = IDLE;
32      end
33    endcase
34  end
```

Листинг 6.2: Описание конечного автомата контроллера

Конечный автомат имеет три состояния IDLE, RECEIVE_DATA, CHECK_PARITY_STOP_BIT.

IDLE – состояние покоя в котором контроллер ожидает первого отрицательного фронта **ps2_clk**. Переход в состояние RECEIVE_DATA происходит по отрицательному фронту **ps2_clk** если **ps2_dat** равно 0, то есть пришел стартовый бит, иначе остаемся в IDLE.

RECEIVE_DATA – состояние в ходе, которого происходит прием данных и бита четности. Переход в состояние CHECK_PARITY_STOP_BIT происходит при счетчике бит равном 8. Отсчитано 8 бит данных и идет прием бита четности.

Последнее состояние CHECK_PARITY_STOP_BITS длительностью в один период **ps2_clk**. В CHECK_PARITY_STOP_BITS происходит проверка бита паритета и стопового бита.

```
1  reg  [8:0]  shift_reg;
2
3    assign data = shift_reg[7:0];
4
5    always @(negedge ps2_clk or posedge areset) begin
6      if(areset)
7        shift_reg <= 9'b0;
8      else
```

```
 9     if(state == RECEIVE_DATA)
10       shift_reg <=
11         {ps2_dat, shift_reg[8:1]};
12   end
```

Листинг 6.3: Описание сдвигового регистра

Сдвиговый регистр необходим для приема и хранения данных и бита четности. По этому, разрядность регистра равна 9. По завершению транзакции в восьмом бите хранится бит четности с 7-0 бит данные. Данные непрерывным присваиванием выведены на внешнюю шину модуля.

Если обратиться к началу лабораторной работы то стоит заметить что данные передаются по интерфейсу PS/2 начиная с младшего бита. Логично будет использовать схему работы сдвигового регистра, при которой сдвиг происходит вправо. Таким образом, первый принятый бит окажется в 0 ячейке сдвигового регистра по окончании транзакции.

Запись в сдвиговый регистр происходит по отрицательному фронту **ps2_clk** и состоянию конечного автомата RECEIVE_DATA.

```
 1  reg  [3:0]  count_bit;
 2
 3    always @(negedge ps2_clk or posedge areset) begin
 4      if(areset)
 5        count_bit <= 4'b0;
 6     else
 7      if(state == RECEIVE_DATA)
 8        count_bit <= count_bit + 4'b1;
 9      else
10        count_bit <= 4'b0;
11    end
```

Листинг 6.4: Описание счетчика принятых бит

Счетчик принятых бит служит для контроля за процессом приема. Инкрементация счетчика происходит только в состоянии RECEIVE_DATA.

```
1  function parity_calc;
```

```
2   input [7:0] a;
3     parity_calc = ~(a[0] ^ a[1] ^ a[2] ^ a[3] ^
4                     a[4] ^ a[5] ^ a[6] ^ a[7]);
5   endfunction
6
7   always @(negedge ps2_clk or posedge areset) begin
8     if(areset)
9             valid_data <= 1'b0;
10          else
11                if (ps2_dat &&
12                    parity_calc(shift_reg[7:0]) == shift_reg[8
13                    state == CHECK_PARITY_STOP_BITS)
14                valid_data <= 1'b1;
15                else
16                  valid_data <= 1'b0;
17  end
```

Листинг 6.5: Описание вырабатывания сигнала готовности к передаче

Последним нерассмотренным моментом остался вопрос генерации сигнала готовности данных. Как было сказано ранее сигнал готовности генерируется к конце транзакции в случае успешного приема и равен 1 до начала следующей транзакции. То есть пока конечный автомат в состоянии IDLE.

Условием успешного окончания транзакции является стоповый бит равный 1 и бит четности равный рассчитанному значению.

Генерация сигнала готовности происходит в момент приема стопового бита. По этому для его проверки достаточно убедиться что значение на линии **ps2_dat** равно 1.

Для проверки бита четности необходимо рассчитать четность принятых 8 бит данных и сравнить ее с значением бита четности. Для упрощения читаемости кода используется функция расчета честности для 8 разрядного регистра согласно правилу отрицания побитового XOR регистра. Правило расчета бита паритета можно узнать из стандарта на интерфейс PS/2.

6.1 Задание лабораторной работы:

1. Ознакомиться с спецификацией на интерфейс PS/2 и представленной реализацией контроллера клавиатуры.
2. Построить временные диаграммы его работы с помощью САПР Altera Quartus.
3. Подготовиться к выполнению индивидуального задания с использованием предложенного контроллера на лабораторной работе.

6.2 Пример индивидуального задания

Выводить на семисегментные индикаторы только коды клавиш с цифрами.

```
 1  module ps2_keyboard(
 2      input            areset,
 3      input      ps2_clk,
 4      input      ps2_dat,
 5      output  reg    valid_data,
 6      output     [7:0} data);
 7
 8
 9  reg  [3:0]  count_bit;
10  reg  [8:0]  shift_reg;
11
12    assign data = shift_reg[7:0];
13
14    function parity_calc;
15    input [7:0] a;
16      parity_calc = ~(a[0] ^ a[1] ^ a[2] ^ a[3] ^
17          a[4] ^ a[5] ^ a[6] ^ a[7]);
18    endfunction
19
20  reg [1:0] state;
21
22  localparam IDLE                    = 2'd0;
23  localparam RECEIVE_DATA            = 2'd1;
```

```
24 localparam CHECK_PARITY_STOP_BITS = 2'd2;
25
26
27   always @(negedge ps2_clk or posedge areset)
28   if(areset)
29     state <= IDLE;
30   else
31   begin
32    case (state)
33    IDLE:
34     begin
35       if(!ps2_dat)
36         state = RECEIVE_DATA;
37     end
38    RECEIVE_DATA:
39     begin
40       if (count_bit == 8)
41         state =
42           CHECK_PARITY_STOP_BITS;
43     end
44    CHECK_PARITY_STOP_BITS:
45     begin
46       state = IDLE;
47     end
48   default:
49     begin
50       state = IDLE;
51     end
52   endcase
53 end
54
55   always @(negedge ps2_clk or posedge areset) begin
56     if(areset)
57       valid_data <= 1'b0;
58     else
59     if (ps2_dat &&
60         parity_calc(shift_reg[7:0]) ==
61       shift_reg[8] &&
62         state == CHECK_PARITY_STOP_BITS)
63           valid_data <= 1'b1;
```

```
64      else
65        valid_data <= 1'b0;
66    end
67
68    always @(negedge ps2_clk or posedge areset) begin
69      if(areset)
70        shift_reg <= 9'b0;
71      else
72      if(state == RECEIVE_DATA)
73            shift_reg <=
74        {ps2_dat, shift_reg[8:1]};
75    end
76
77    always @(negedge ps2_clk or posedge areset) begin
78      if(areset)
79        count_bit <= 4'b0;
80     else
81      if(state == RECEIVE_DATA)
82        count_bit <= count_bit + 4'b1;
83      else
84        count_bit <= 4'b0;
85    end
86
87  endmodule
```

Листинг 6.6: Пример реализации контроллера PS/2

6.3 Варианты индивидуальных заданий

1. Выводить на семисегментные индикаторы только коды клавиш "W "A "S "D"и "пробел".

2. Выводить на семисегментные индикаторы только коды клавиш "1 "3 "5 "7 "9".

3. Выводить на семисегментные индикаторы только коды клавиш, находящихся на num-pad.

4. Выводить на семисегментные индикаторы только коды клавиш "Q "W "E "R "T "Y".

5. Выводить на семисегментные индикаторы только коды клавиш "I "D "Q "D".

6. Выводить на семисегментные индикаторы только коды клавиш со стрелками.

7. Выводить на семисегментные индикаторы только коды клавиш "I "D "K "F "A".

8. Выводить на семисегментные индикаторы только коды клавиш F1 – F12.

9. Выводить на семисегментные индикаторы только коды клавиш "Z "X "C "V "B "N".

10. Выводить на семисегментные индикаторы только коды клавиш "L "J "S "P "Q K.

Лабораторная работа №7 FLASH память

7.1 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.

7.2 Принципы работы FLASH-памяти

В качестве элемента хранения информации FLASH-память

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

На Рис.7.1 схематично изображена структура такого транзистора.

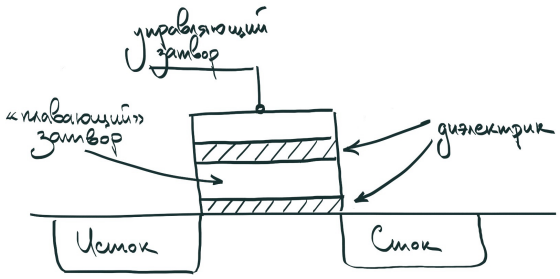


Рис. 7.1: Транзистор с плавающим затвором

Как вы видите, он содержит два затвора: управляющий и плавающий. Плавающий затвор — это полупроводник, 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 про-

текать через него при меньшем напряжении «управляющий затвор—исток».

На Рисунке 7.2 изображены вольт-амперные характеристики транзистора с плавающем затвором в двух разных состояниях: когда в плавающем затворе накоплен отрицательный заряд и когда плавающий затвор не имеет заряда.

Для хранения информации используют следующий принцип: 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».

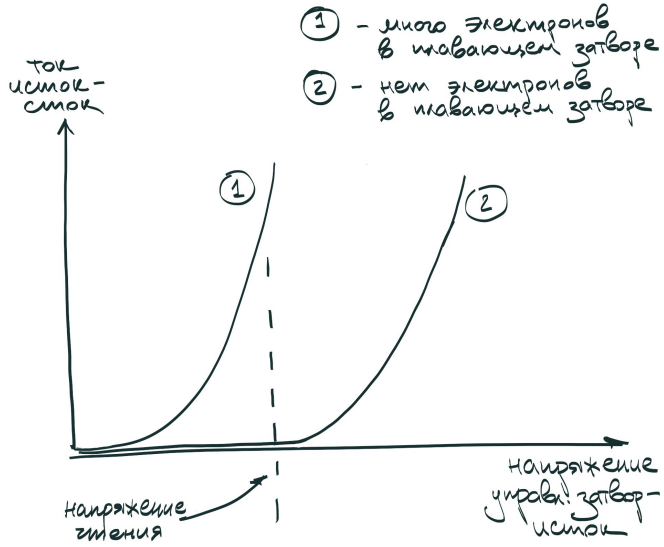


Рис. 7.2: Чтение значение из транзистора с плавающим затвором

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.7.3).

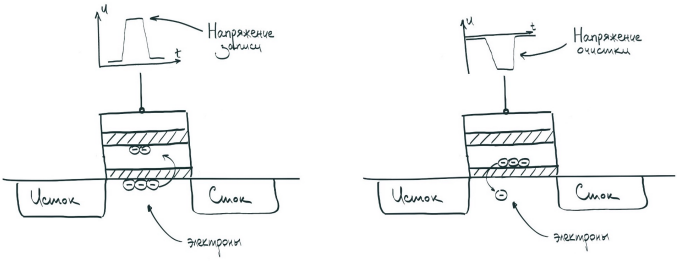


Рис. 7.3: Изменение состояния плавающего затвора

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.

7.3 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.

7.4 Структура FLASH-памяти

На Рисунке 7.4 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.

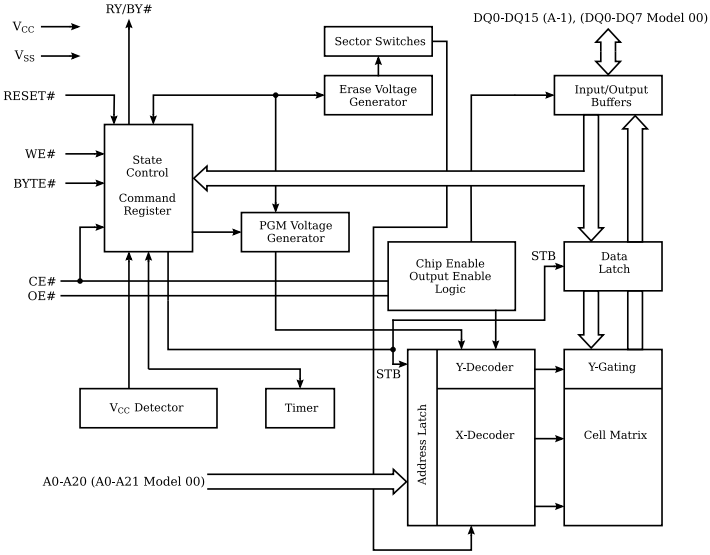


Рис. 7.4: Структура flash-памяти S29AL032D

7.5 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.

7.5.1 Проектирование контроллера S29AL032D

Как мы уже знаем, контроллер предназначен для обмена информацией с внешними цифровыми устройствами. Он должен 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 информацию о самой микросхеме и операциях записи и чтения, описанные в datasheet S29AL032D.

7.5.2 Операция чтения

Важной особенностью микросхемы S29AL032D является использование двунаправленной шины данных. Это сделано для минимизации разводки печатной платы при её использовании. В дальнейшем нам придется разработать механизм,

позволяющий использовать одну шину и для операции чтения и для операции записи.

Чтение данных из микросхемы S29AL032D не требует никакой дополнительной подготовки. Временная диаграмма чтения приведена в пункте 17.2 datasheet и представлена на Рисунке7.5.

Времена, указанные на диаграмме, приведены в Таблице 7.1.

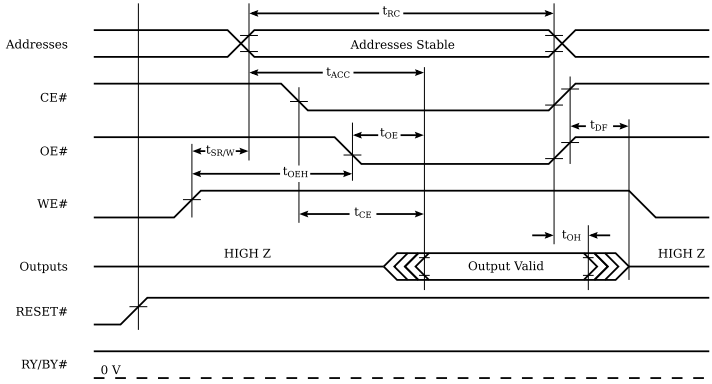


Рис. 7.5: Временная диаграмма чтения S29AL032D

Обозн.	Описание	Min	Max
t_{RC}	Продолжительность цикла чтения	70 нс	—
t_{ACC}	Задержка Адрес — Данные	—	70 нс
t_{CE}	От активного CE до формирования выхода	—	70 нс
t_{OE}	От активного OE до формирования выхода	—	30 нс
t_{DF}	От снятия CE до Z-состояния выхода	—	16 нс
$t_{SR/W}$	Задержка между операциями чтения и записи	20 нс	—
t_{OEH}	Время активного сигнала OE	10 нс	—
t_{OH}	Время удержания данных	0 нс	—

Таблица 7.1: Временные характеристики операции чтения S29AL032D в режиме 70 нс.

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 7.1 к тактовому сигналу частоты 50 МГц, которым тактируется устройство. При этом учтём, что некоторые задержки могут быть равны нулю.

Полученная временная диаграмма показана на Рисунке 7.6

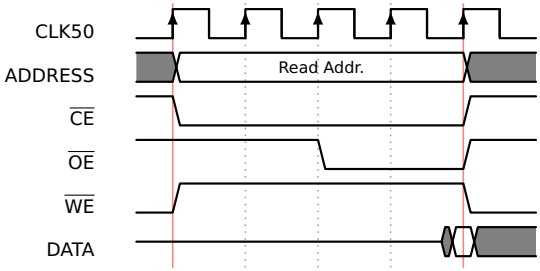


Рис. 7.6: Временная диаграмма операции чтения S29AL032D, привязанная к тактовому сигналу, частотой 50 МГц.

7.5.3 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 завершающей части одной операции записи приведена в пункте 17.4 datasheet и представлена на Рисунке 7.7, а её временные характеристики приведены в Табл. 7.2.

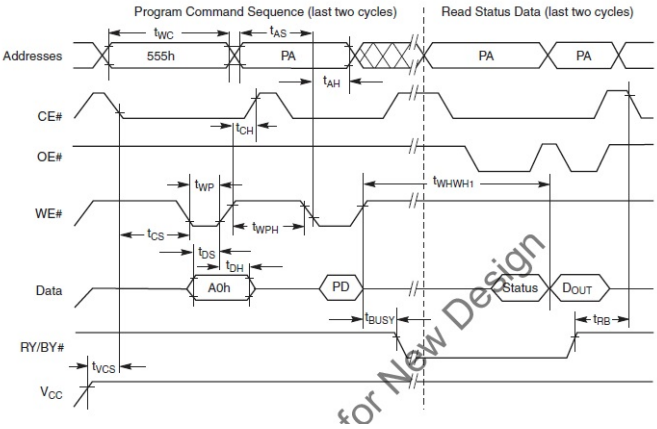


Рис. 7.7: Временная диаграмма фрагмента операции записи S29AL032D

Обозн.	Описание	Min	Max
t_{WC}	Продолжительность цикла записи	70 нс	—
t_{AS}	Время установки адреса	0 нс	—
t_{AH}	Время удерживания адреса	45	—
t_{DS}	Время установки данных	35 нс	—
t_{DH}	Время удержания данных	0 нс	—
t_{CS}	Время установки сигнала CE	0 нс	—
t_{CH}	Время удержания сигнала CE	0 нс	—
t_{WP}	Ширина импульса записи	35 нс	—
t_{WPH}	Интервал между записью	30 нс	—
t_{WHWH1}	Время программирования байта	≈ 9 мкс	
t_{BUSY}	От активного CE до установка выхода	35 нс	70 нс
t_{RB}	От активного OE до установки выхода	—	30 нс

Таблица 7.2: Временные характеристики операции записи S29AL032D в режиме 70 нс.

Аналогично операции чтения, привяжем форму и времена временной диаграммы записи к тактовому сигналу, частотой 50 МГц. Полученная диаграмма, представлена на Рис. 7.8

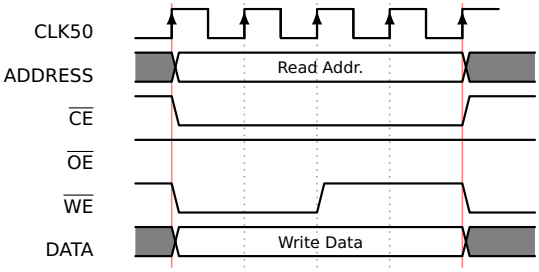


Рис. 7.8: Временная диаграмма одного цикла операции записи S29AL032D, привязанная к тактовому сигналу, частотой 50 МГц.

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.

7.5.4 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 — приблизительно 0.7 сек.

Полная очистка микросхемы — приблизительно 45 сек.

7.5.5 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. 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 7.3

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».

Операция	DQ7	DQ6	DQ5	DQ4	DQ3	DQ2
Запись	∼DQ7	меняет значение	0	таймаут	—	не меняет значение
Очистка	0	меняет значение	0	таймаут	1	меняет значение

Таблица 7.3: Значение разрядов статуса.

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.

7.5.6 Проектирование контроллера Flash (продолжение)

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

Контроллер должен обеспечивать чтение, запись и очистку 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.

Из состояния бездействия возможны три различных перехода: операция чтения, операция записи и операция очистки. После окончания этих операций автомат снова возвращается в состояние бездействия.

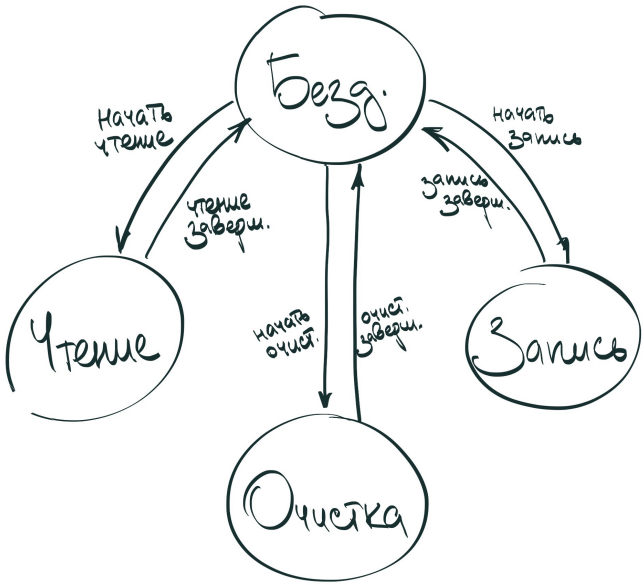


Рис. 7.9: Управляющий конечный автомат контроллера S29AL032D

Прежде всего, нас интересуют сложные операции «запись» и «очистка». Выделим их основные этапы.

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

Для контроля статуса операции нам нужно считать данные из адреса, по которому производится запись и проанализировать их.

Аналогичные рассуждения справедливы и для операции очистки. Отразим все эти замечания в состояниях конечного автомата (см. Рисунок 7.10).

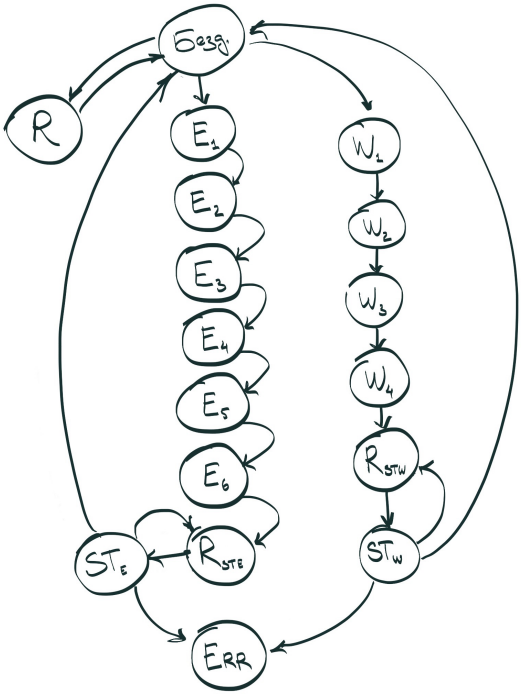


Рис. 7.10: Доработанный управляющий конечный автомат контроллера S29AL032D

Раньше мы не разделяли эти состояния $W_1 ... W_4, ST_w$ и всё вместе называли «запись». Но, постепенно уточняя детали, мы разбили сложную операцию на более простые этапы.

Первое, что бросается в глаза в получившемся графе — это многократное повторение операций записи в состояниях $W_1 ... W_4$ и $E_1 ... E_6$. Повторение операции чтения менее заметно: она происходит в состояниях R и ST_W, ST_E (так как используется для проверки статуса).

Также можно постараться выделить анализ статуса ST_W и ST_E в состояние, общее для веток записи и очистки.

Тогда структура конечного автомата приобретает вид, показанный на Рисунке 7.11

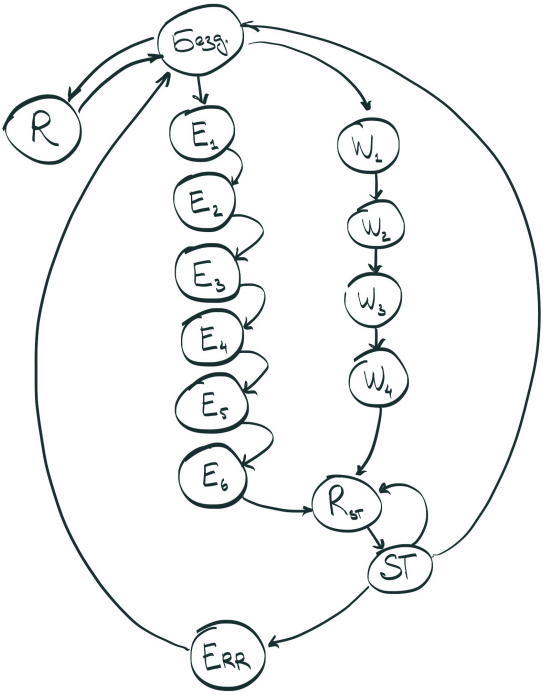


Рис. 7.11: Конечный автомат контроллера S29AL032D с общим состоянием проверки статуса

7.5.7 Реализация операций чтения и записи

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.

Для того, чтобы провести чтение, необходимо развернуть временную диаграмму, привязанную к тактовому сигналу, которую мы получили на Рисунке 7.6. Аналогичная временная диаграмма для записи была представлена на Рисунке 7.8.

Как мы уже обсуждали, схема которую можно использовать для разделения событий во времени — это конечный ав-

томат. Например, чтобы реализовать операцию чтения, разобьём временную диаграмму чтения на этапы, и поставим каждому этапу в соответствие уникальное состояние, как представлено на Рисунке 7.12

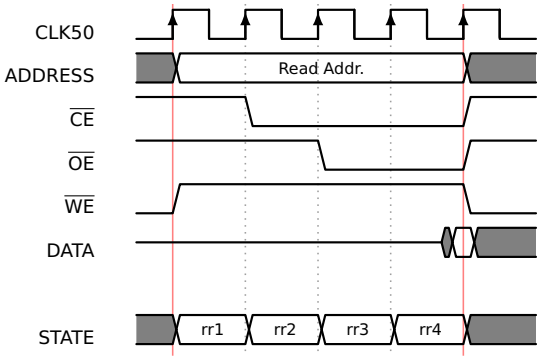


Рис. 7.12: Соответствие состояний этапам операции чтения S29AL032D

Мы могли бы добавить эти состояния в конечный автомат, который мы уже начали проектировать, но тогда нам пришлось бы каждое из состояний ST, RD, $W_1 ... W_4$ и $E_1 ... E_6$, разбить на несколько состояний. Это привело бы к ненужному усложнению структуры конечного автомата и аппаратной избыточности.

Вместо этого мы можем сделать отдельные небольшие блоки, которые будут выполнять эти операции и, таким образом, разделить конченые автоматы автоматы.

Нам надо будет только предусмотреть, что для использования этих блоков надо предусмотреть механизмы запуска операции и сигнализации о окончании работы блока.

Сигналом запуска для таких блоков будет признак того, что основной автомат находится в состоянии «чтение» или «запись», соответственно, а сигнал окончания работы будет вырабатываться в состоянии «завершено».

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

7.5.8 Реализация контроллера микросхемы S29AL032D на языке Verilog

ИНТЕРФЕЙС

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. Но для данных, по аналогии с RAM-памятью потребуются два отдельных входа — для записи и для чтения.

```
 1  module flash_controller
 2  ( //module interface
 3    input        clk,
 4    input  [7:0] data_for_flash,
 5    output [7:0] data_from_flash,
 6    input [21:0] address,
 7    input        we,
 8    input        oe,
 9    input        erase,
10    output       error,
11    output       ready,
12
13    //flash interface
14    inout  [7:0]  flash_data,
15    output [21:0] flash_address,
16    output        flash_nwe,
17    output        flash_nrst,
18    output        flash_nce,
19    output        flash_noe );
20  endmodule
```

Листинг 7.1: Описание интерфейса контроллера S29AL032D

ОСНОВНОЙ УПРАВЛЯЮЩИЙ КОНЕЧНЫЙ АВТОМАТ

Теперь опишем основной управляющий конечный автомат. Необходимо помнить, что данные, которые мы записываем во flash-память нужно сохранить, чтобы они оставались неизменными все время выполнения операции. Поэтому сразу дополним описание регистром для хранения этих данных.

Сразу учтём, что эти же данные можно будет использовать для проверки статуса. Поэтому для операции очистки сектора, запишем в регистр 8'hFF.

```
 1  reg [3:0] main_state;
 2
 3  localparam idle = 4'd0;
 4  localparam r    = 4'd1;
 5  localparam w1   = 4'd2;
 6  localparam w2   = 4'd3;
 7  localparam w3   = 4'd4;
 8  localparam w4   = 4'd5;
 9  localparam e1   = 4'd6;
10  localparam e2   = 4'd7;
11  localparam e3   = 4'd8;
12  localparam e4   = 4'd9;
13  localparam e5   = 4'd10;
14  localparam e6   = 4'd11;
15  localparam rs   = 4'd12;
16  localparam st   = 4'd13;
17  localparam err  = 4'd14;
18
19  wire r_done, w_done;
20  wire wire status_ok;
21  wire error;
22
23  always @(posedge clk or posedge rst) begin
24    if (rst) main_state <= idle;
25    else begin
26      case (main_state)
27        idle: if (we)         main_state <= w1;
28              else if (oe)    main_state <= r;
29              else if (erase) main_state <= e1;
```

```
30
31        w1: if (w_done) main_state <= w2;
32        w2: if (w_done) main_state <= w3;
33        w3: if (w_done) main_state <= w4;
34        w4: if (w_done) main_state <= st;
35
36        e1: if (w_done) main_state <= e2;
37        e2: if (w_done) main_state <= e3;
38        e3: if (w_done) main_state <= e4;
39        e4: if (w_done) main_state <= e5;
40        e5: if (w_done) main_state <= e6;
41        e6: if (w_done) main_state <= st;
42
43        r:  if (r_done) main_state <= idle;
44        rs: if (r_done) main_state <= st;
45        st: begin
46              if (status_ok)  main_state <= idle;
47              else if (error) main_state <= err;
48              else            main_state <= rs;
49            end
50
51        err: main_state <= idle;
52      endcase
53    end
54  end
55
56  reg [7:0] w_data;
57  always @(posedge clk or posedge rst) begin
58    if (rst) w_data <= 8'h00;
59    else begin
60      if ( main_stat == idle) begin
61        if ( we )    w_data <= data_for_flash;
62        if ( erase ) w_data <= 8'hFF;
63      end
64    end
65  end
```

Листинг 7.2: Описание управляющего конечного автомата контроллера S29AL032D

КОНЕЧНЫЙ АВТОМАТ ЧТЕНИЯ

Опишем автомат чтения и наладим связь между автоматами. Для этого опишем сигналы, которые понадобятся для управления каждым из этих автоматов.

```
reg [2:0] read_state;
localparam rr1     = 3'd0;
localparam rr2     = 3'd1;
localparam rr3     = 3'd2;
localparam rr4     = 3'd3;

always @(posedge clk or posedge rst) begin
  if (rst) read_state <= r_idle;
  else begin
    case (read_state) begin
      rr1: if (start_rea) state <= rr2;
      rr2: state <= rr3;
      rr3: state <= rr4;
      rr4: state <= rr1;
    endcase
  end
end

assign r_done   = (read_state == rr4);
wire read_nce   = (read_state == rr1);
wire read_noe   = ((read_state == rr1) |
                   (read_state == rr2));
wire read_nwe   = 1'b1;
wire start_read = (main_state == r) |
                  (main_state == rs);

reg [7:0] r_data;
always @(posedge clk or posedge rst) begin
  if (rst)         r_data <= 8'h00;
  else if (r_done) r_data <= flash_data;
end
```

Листинг 7.3: Описание конечного автомата чтения контроллера S29AL032D

ПРОВЕРКА СТАТУСА ОПЕРАЦИИ

Мы заранее позаботились о том, чтобы проверка статуса не требовала от нас больших усилий: мы защелкиваем данные при начале операции записи и очистки. Теперь нам достаточно сравнить полученное значение с эталонным:

```
1 wire   timeout   = data_from_flash[4];
2 assign status_ok = ( r_data == w_data )
3 assign error     = ~status_ok && timeout;
```

Листинг 7.4: Проверка статуса операции в контроллере S29AL032D

УПРАВЛЕНИЕ МИКРОСХЕМОЙ S29AL032D

Опишем поведение двунаправленной шины данных и сигналов управления для S29AL032D. В состояниях чтения, шиной данный должна управлять микросхема, в состояниях записи — контроллер:

```
1 wire writing = ((main_state == w1) ||
2                 (main_state == w2) ||
3                 (main_state == w3) ||
4                 (main_state == w4));
5
6 assign flash_data = writing? w_data : 8'hZZ;
7 assign flash_noe  = writing? write_noe : read_noe;
8 assign flash_nce  = (main_state == idle);
9 assign flash_nwe  = writing? write_nwe : read_nwe;
```

Листинг 7.5: Управление микросхемой S29AL032D

7.6 Задание лабораторной работы

1. Разработать конечный автомат записи;
2. Разработать механизм передачи различных данных в состояниях $W_1, ..., W_4$ и $E_1, ..., E_6$;
3. Выполнить полную интеграцию контроллера;
4. Создать проект, для демонстрации возможностей контроллера, удовлетворяющий требованиям, представленным ниже.

7.6.1 Требования к демонстрационному проекту

РЕЖИМЫ РАБОТЫ

С помощью $SW[1], SW[2]$ задается режим — запись, чтение или очистка.

СБРОС

Сброс устройства происходит по нажатию $BNT[0]$

РЕЖИМ ЗАПИСИ

В режиме записи с помощью кнопок $BTN[3], BTN[2]$ задается адрес записи, затем ввод подтверждается нажатием $BTN[1]$. Здесь и далее — во время ввода, вводимое значение отображается на семисегментных индикаторах.

После подтверждения ввода адреса записи, с помощью кнопок $BTN[3], BTN[2]$ вводится записываемое значение и подтверждается нажатием $BTN[1]$.

После подтверждения ввода значения происходит запись в микросхему S29AL032D. В случае успешной записи подсвечивается $LEDG[0]$ и демо-проект возвращается к вводу адреса записи, а в случае ошибки подсвечивается $LEDR[0]$.

РЕЖИМ ЧТЕНИЯ

В режиме чтения с помощью кнопок $BTN[3], BTN[2]$ задается адрес чтения, затем ввод подтверждается нажатием $BTN[1]$.

После подтверждения ввода адреса чтения, происходит чтение значения из микросхемы S29AL032D. Считанное значение выводится на семисегментные индикаторы. После нажатия $BTN[1]$, устройство переходит в выбранный режим.

РЕЖИМ ОЧИСТКИ

В режиме очистки, с помощью кнопок $BTN[3], BTN[2]$ задается номер очищаемого сектора. Послед подтверждения ввода с помощью кнопки $BTN[1]$, производится выбранная операция.

После окончания операции в случае успешной очистки подсвечивается $LEDG[0]$, а в случае ошибки $LEDR[0]$.