Лабораторная работа №4
Исследование последовательностных логических устройств

1. Цель работы:

Изучить на примере исследования последовательностных логических схем возможности языка Verilog: задание времени моделирования и временных задержек, а также операторы присвоения, операторы задания постоянного выполнения последовательности команд и другие операторы языка Verilog.

2. Теоретические сведения

2.1. Краткие теоретические сведения по языку Verilog (продолжение)

При работе с языками описания аппаратуры необходимо всегда помнить одну особенность. Программа, написанная на этом языке (в том числе – на Verilog) выполняется **параллельно**! В этом главное отличие языка Verilog от таких процедурных языков программирования как C и Pascal. Понять это довольно просто. Ведь с помощью языков HDL описывается схема электрическая принципиальная (т.е. – собственно устройство, в котором электрические сигналы всегда распространяются **одновременно)**.
Для этого в синтаксисе языка Verilog предусмотрены следующие операторы, выполняющиеся параллельно:

- **`assign;`**
- **`always.`**

Таким образом, если в программе описания модуля используется несколько операторов `assign` и `always`, все они будут выполняться параллельно (одновременно).

2.2. Понятия моделирования и синтеза

Работу логического устройства, описанного с помощью языков HDL, можно проверить двумя способами: либо на персональном компьютере (**моделирование**), либо непосредственной реализацией в микросхеме (например – ПЛИС) (**синтез**). Процесс синтеза намного сложнее моделирования, поскольку предполагает, помимо компиляции кода, еще и синтез логической схемы (т.е. реализацию устройства с помощью доступных компонентов ПЛИС и трассировку проводников в выбранной микросхеме). Поэтому результат применения некоторых конструкций языка Verilog при моделировании и синтезе **отличается**! Не все операторы языка Verilog являются **синтезируемыми!**
Обычно для моделирования используются специальные тестовые программы – тестбенчи (**testbench**), которые предназначены для проверки программы описания модулей (их верификации). Обычно, для выполнения моделирования используется специализированное ПО, например – **ModelSim**.

2.3. Модельное время

Итак, можно выделить три факта. Во-первых, моделирование логического устройства можно выполнять на ПК. Во-вторых, многие операторы языка Verilog выполняются параллельно. В-третьих, ПК выполняет инструкции строго последовательно.
Тогда, как с помощью компьютера выполнить моделирование событий, которые выполняются одновременно?
В начале необходимо разобраться с понятиями **реального** и **модельного** времени. Они, естественно, отличаются друг от друга.
В *реальном времени* описываются все изменения, происходящие в моделируемом устройстве. В *модельном времени* описываются все изменения, происходящие в моделирующей программе. Такое разделение позволяет выполнить на компьютере вначале последовательно все параллельные инструкции, а затем предоставить результат в таком виде, как будто бы они выполнялись одновременно.
Для задания модельного времени в языке Verilog служит конструкция **`'timescale param1/param2. param1`** задает единицу модельного времени (по умолчанию – 1 нс), а **`param2`** – точность модельного времени. Например, **`'timescale 1ns/10ps`** означает, что

единица модельно времени 1 наносекунда. Шаг в 10 единиц модельного времени составит 10 нс. Значение 10 пикосекунд (второй параметр директивы `timescale`) указывает на точность, с которой будут выполняться вычисления всех изменений за один шаг моделирования. Естественно, что при вычислении задержек в устройстве, время будет округляться с точностью до 10 пикосекунд. С помощью этой конструкции можно задавать точность моделирования.

2.4. Временные задержки

Различают временные задержки, связанные с моделированием и синтезом.

Временные задержки при моделировании задаются пользователем с помощью оператора **#time** (параметр **`time`** определяет время задержки в единицах модельного времени). Обычно, такие задержки имитируют конечное время распространения сигналов в реально работающих устройствах. Например, чтобы организовать задержку в 5 наносекунд необходимо записать следующий код: **`#5;`**

Временную задержку можно вводить перед оператором присваивания значения сигналу, моделируя время распространения сигнала в реальной схеме:

```
assign #10 c = a^b;
```

Данная конструкция описывает элемент «исключающее ИЛИ» (XOR) с задержкой распространения равной 10 (10 единиц модельного времени – первого параметра директивы **`timescale**, который по умолчанию равен 1 нс). При этом, все задержки в непрерывных присвоениях являются инерциальными. Это значит, что изменения сигнала A на время меньшее 10 нс не приведет к изменениям сигнала C. Для того, чтобы произошло изменение сигнала C требуется, чтобы сигнал A был зафиксирован в новом состоянии на время более 10 нс.

А что же будет при синтезе? Если использовать задержки такого вида при синтезе, то ничего не произойдет! Данный оператор является **несинтезируемым,** т.е. он будет игнорироваться. Для формирования задержек в реальном устройстве используются другие методы.

2.5. Операторы языка Verilog

Оператор `assign`

Используется для непрерывного присвоения сигнала переменной типа **`wire`**.

Синтаксис:

```
assign var = expression;
```

Действие оператора: при изменении выражения `expression` (например, изменилось значение переменной, входящей в `expression`), вычисляется новое значение выражения и результат присваивается переменной `var`. В левой части выражения переменная может быть только типа `wire`, а в правой части – возможна комбинация переменных `wire, reg, integer`. Все операторы **`assign`** в модуле выполняются параллельно.

Оператор `always`

Один из основных, эффективных операторов языка Verilog. Позволяет задать постоянное выполнение последовательности команд. Указанная последовательность может выполняться либо циклически (в бесконечном цикле), либо только после появления определенного события.

***Синтаксис*:**

```
1  always @( …events… )
2  begin
3  …
4  // последовательность операторов (выполняется последовательно, один за другим)
5  …
6  end
```

Описание: Данный оператор применяется тогда, когда возникает необходимость последовательного выполнения команд. Для этого, после ключевого слова **always**, следует последовательность операторов, заключенных в блок **begin/end**. Все команды, расположенные внутри такого блока, выполняются последовательно, а сами операторы **always** – параллельно. Когда блок **begin/end** отсутствует, то действие оператора **always** распространяется только на один следующий за ним оператор. Если в операторе **always** отсутствует конструкция **@(…events…)**, то набор команд, расположенных между **begin** и **end**, выполняется в бесконечном цикле.
Для указания условия, необходимого для запуска блока **always**, используется конструкция **@(…events…)**. В этом случае, внутри круглых скобок указывается список условий (событий) инициирующих выполнение оператора **always**. Условием может быть появление положительного или отрицательного фронта, изменение значения сигнала и пр. Положительный фронт указывается ключевым словом **posedge**, отрицательный фронт – **negedge**. После ключевых слов **posedge** и **negedge** следует имя сигнала. Если необходимым условием является любое изменение сигнала – в скобках просто указывается имя контролируемого сигнала.

Примеры:

```
1  always @(posedge CLK)    // условие запуска оператора always – положительный
2                           // фронт сигнала CLK;
3  always @(negedge CLK)    // условие запуска оператора always – отрицательный
4                           // фронт сигнала CLK;
5  always @( CLK )          // условие запуска оператора always – любое
6                           // изменение уровня сигнала CLK;
```

Если необходимо указать несколько условий, приводящих к запуску оператора **always**, внутри скобок их объединяют операторами **or** или **and**. Либо записывают через запятую (эквивалент оператора **or**). Например, следующие операторы описывают условия запуска RS триггера:

```
1  always @(posedge CLK or R or S)   // обе эти записи приводят
2
3  always @(posedge CLK,R,S)         // к одинаковому результату
```

Можно использовать неявное указание событий запуска. В этом случае внутри скобок ставится звездочка **(*)**. В данном случае срабатывание оператора **always** произойдет при любом изменение значений переменных, стоящих в правой части оператора присваивания, изменение условий и выражений в операторах **if** и **case** (когда они применяются в операторе **always**).
Пример:

```
1  always @(*)   // тоже самое, что использовать always @( a or b or c or d or f )
2
3  y = (a&b)|(c&d) | myfunction(f);
```

Важное замечание! Внутри оператора **always** нельзя использовать переменные типа `wire` (т.е. все переменные, которым присваивается значение внутри данного оператора, должны быть типа `reg`). Иначе, это приведет к ошибкам компиляции. Внутри оператора **always** невозможен вызов другого модуля. Все операторы **always** выполняются параллельно.

Оператор `initial`

Данный оператор используется для задания последовательности команд, которые необходимо выполнить один раз при запуске проекта на моделирование. Обратите внимание, оператор **`initial`** не поддерживается при синтезе, а используется только при моделировании.

***Синтаксис*:**

```
1  initial
2  begin
3        …
4        // операторы (выполняются последовательно)
5        …
6  end
```

***Описание*:** операторы, расположенные между конструкцией **`begin`** и **`end`** выполняются только один раз при запуске программы на моделирование. Позволяют задать исходное состояние устройства.

Оператор конкатенации `{}`

Фигурные скобки (оператор конкатенации) используются для объединения (склеивания или слияния) нескольких различных сигналов в один. Разрядность полученного сигнала равна сумме разрядностей всех объединяемых сигналов.

***Синтаксис*:**

```
1  {arg1, arg2, … , argN}
```

Описание: переменные различной разрядности **`arg1…argN`** объединяются в один сигнал.

Оператор проверки условия ?:

Оператор проверки условия (вопросительный знак с двоеточием для разделения действий) на языке Verilog работает также как в С.

***Синтаксис*:**

```
1  assign var1 = (condition) ? var2:var3;
```

Описание: Вначале проверяется условие ***condition***, если оно истинно – выполняется первая команда, если ложно – команда, следующая за разделительным двоеточием. Пример реализации простейшего мультиплексора 2-в-1 с помощью условного оператора:

```
1  assign Y = (SEL) ? A:B;
```

Оператор `parameter`. Объявление символического имени.

Этот оператор используется тогда, когда необходимо назначить константам символические имена.

Синтаксис:

```
1  parameter [1:0] state_A =0, state_B = 1, state_C = 2, state_D = 3;
```

Оператор выбора `case`.

Применяется для выбора одного из нескольких вариантов.

Синтаксис:

```
case( expr )
    case_item1:
            begin
                ...
            end
    case_item2:
            begin
                ...
            end
    default:
            begin
                ...
            end
endcase
```

Описание: Оператор выбора `case` проверяет выражение `expr` и, в зависимости от его значения, выполняется определенная команда. Данный оператор гарантирует исполнение одной ветви. В случае если ни одно из перечисленных значений не совпадает с текущим, выполняется ветвь **`default`**. Пример использования оператора выбора – реализация двоично-десятичного дешифратора (аналог м\с К155ИД3):

```
case( rega )
        4'd0: result = 10'b0111111111;
        4'd1: result = 10'b1011111111;
        4'd2: result = 10'b1101111111;
        4'd3: result = 10'b1110111111;
        4'd4: result = 10'b1111011111;
        4'd5: result = 10'b1111101111;
        4'd6: result = 10'b1111110111;
        4'd7: result = 10'b1111111011;
        4'd8: result = 10'b1111111101;
        4'd9: result = 10'b1111111110;
        default: result = 10'bx;
endcase
```

Оператор проверки условия `if…else`

Синтаксис:

```
if (<expression>)
     <statement1>
else
    <statement2>
```

Описание: оператор проверки условия проверяет истинность выражения `expression` (любое выражение языка Verilog). Если условие истинно – выполняется команда *statement1* (оператор или группа операторов между **`begin`** и **`end`**). Если условие ложно – выполняется команда *statement2,* расположенная в ветви **`else`**. Ветвь **`else`** может отсутствовать, но если имеются вложенные условные операторы **`if`** (как в примере ниже), то **`else`** относится к ближайшему оператору **if**. Для выполнения нескольких команд при совпадении условия следует пользоваться конструкцией **`begin`** и **`end`**. Выражение `expression` считается истинным, если его значение не равно 0 и не является неопределенным (x или z). Следует помнить, что также, как и в языке C, операция сравнения записывается как == (два подряд знака =), в отличие от операции присваивания = (один знак). Операции сравнения при неопределенных операндах возвращает неопределенное значение (x). Выражение

expression не обязательно должно быть выражением типа `boolean`, а является любым выражением, которое может быть приведено к типу `integer`. Здесь прослеживается аналогия с языком С, единственное отличие состоит в том, что тип `integer` в языке Verilog, в отличие от типа `int` языка С, может принимать неопределенные значения (x или z). В случае, когда выражение принимает эти значения, выполняется ветвь **`else`**. Пример использования условного оператора – реализация синхронного одноразрядного регистра-мультиплексора с сигналом разрешения выхода OE:

```
1  always @(negedge Clk)
2     begin
3          if (OE)
4               out = data;
5          if (LE)
6               data = inA;
7          else
8               data = inB;
9     end
```

Оператор цикла `for`

Данный оператор применяется для организации циклического повторения команды или группы команд при известном числе итераций.

***Синтаксис*:**

```
1  for(_index =__low_range;__index <__high_range; __index =__index + __step)
2  begin: (name)
3        …
4        // some code (группа циклически выполняемых команд)
5        …
6  end
```

Описание: принцип работы оператора цикла **`for`** напоминает работу оператора цикла языка С. Единственное отличие состоит в том, что при увеличении переменной `__index` возникает соблазн воспользоваться оператором +=, которого нет в языке Verilog. В данном случае используется конструкция `__index = __index + __step`. Для досрочного выхода из цикла (блоки команд должны быть именованы – уникальное имя после ключевого слова **`begin:`**) используется оператор **`disable`**. Для сравнение с языком С: действие оператора **`disable`** совпадает с действием оператора **`break`** языка С.

***Пример*:**

```
1  initial
2       begin : break
3             for(i = 0; i <n; i =i+1)
4             always @CLK if (a == 0)
5             disable break;
6       end
```

Оператор цикла `while`

Синтаксис:

```
1  while (condition)
2  begin
3       …
4         //some code (исполняемый фрагмент программы)
5       …
6  end
```

Описание: оператор цикла **`while`** работает так же, как в других языках программирования. Если проверяемое условие *(condition)* истинно, то выполняется последовательность команд, расположенных между операторами **`begin`** и **`end`**. Если условие ложно – оператор цикла не выполняется. Он используется только в процедурных блоках, например – в секции **`initial`**. Оператор цикла **`while`** не синтезируемый (применяется только для моделирования).

Оператор цикла `repeat`

Данный оператор позволяет организовать цикл с конечным (фиксированным) количеством повторений.

Синтаксис:

```
1  repeat(N)
2  begin
3       // some code (исполняемый фрагмент программы)
4  end
```

Описание: последовательность команд, расположенных между операторами **`begin`** и **`end`** будет выполнена N раз. Этот оператор цикла используется только в процедурных блоках, например – в **`initial`** или **`always`**. Оператор цикла **`repeat`** не синтезируемый (применяется только для моделирования).

Пример применения данного оператора – генерация убывающей последовательности от 127 до 0:

```
1  integer count;
2  initial
3  begin
4       count = 128;
        repeat (count)
        begin
           count = count - 1;
        end
   end
```

Оператор бесконечного цикла forever

Для реализации бесконечного цикла в языке Verilog (аналогичного оператору **`for(;;)`** других языков программирования) используется оператор **`forever`**.

Синтаксис:

```
1  forever
2  begin
3       // some code (исполняемый фрагмент программы)
4  end
```

Описание: фрагмент программы, расположенный между операторами **`begin`** и **`end`** будет повторяться в бесконечном цикле. Поэтому, необходимо предусмотреть условия завершения цикла **`forever`**. Это можно сделать, например, с помощью системной функции **`$finish`**. Данный оператор используется только в процедурных блоках. Оператор бесконечного цикла **`forever`** не синтезируемый (применяется только для моделирования).

Пример – формирование тактового сигнала с периодом повторения, равным 10 шагам модельного времени на интервале 2000 шагов модельного времени:

```
1   reg clock;
2   initial
3   begin
4      clock = 1'b0;
5      forever #5 clock = ~clock; // the clock flips every 5 time units.
6   end
7   initial #2000 $finish;
```

Арифметические и логические операторы

***Арифметические операторы*:** + , – , * , / , %;

Логические операторы (для сгруппированных значений, используются с одной переменной):

- **!** – логическое отрицание;
- **&&** – логическое “И”;
- **||** – логическое “ИЛИ”;

Логические операторы побитовые (используются с несколькими переменными):

- **~** – побитовое отрицание;
- **&** – побитовое “И”;
- **|** – побитовое “ИЛИ”;
- **^** – побитовое “ИСКЛЮЧАЮЩЕЕ ИЛИ”;

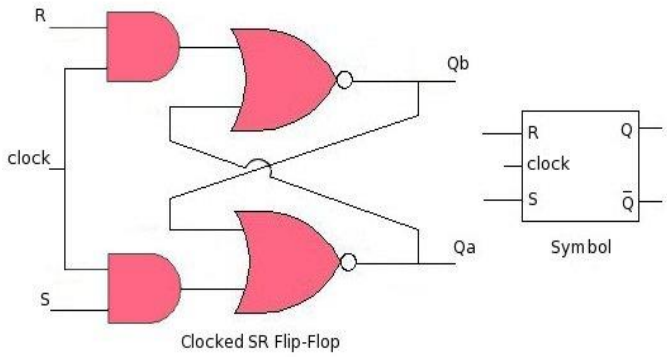
***Операторы сравнения*:**

- **==** – логическое равенство;
- **!=** – логическое неравенство;
- **===** – побитовое равно (полное совпадение);
- **!==** – побитовое неравно (полное несовпадение);
- **<=** – логическое сравнение «меньше – или – равно»;
- **>=** – логическое сравнение «больше – или – равно»;
- **>** – логическое сравнение «больше»;
- **<** – логическое сравнение «меньше».

3. Порядок выполнения работы

Особенностью последовательностных логических устройств является зависимость выходного сигнала не только от действующих в настоящий момент входных логических сигналов, но и от тех, которые действовали в предыдущий момент. Для выполнения этих условий, последовательностное устройство должно обладать памятью. Функцию запоминания значений логических переменных в цифровых схемах выполняют **триггеры**. Таким образом, триггер является неотъемлемой частью любого последовательностного устройства.

Опишем на языке Verilog RS-триггер, схема которого представлена на рисунке:

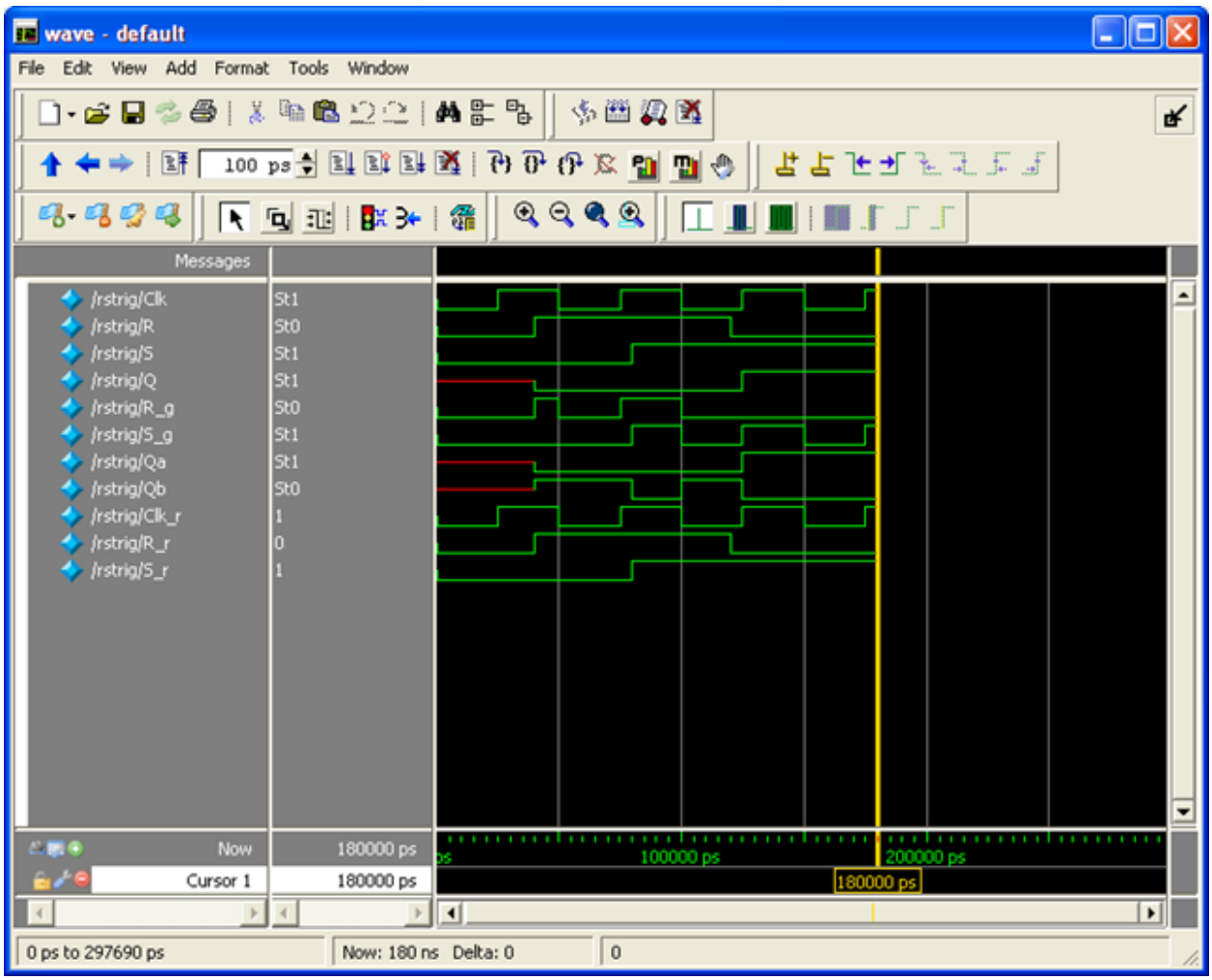


Для этого:

1. Создайте новый проект в среде ModelSim.
2. Создайте новый исходный файл с описанием RS-триггера (исходный текст программы приводится ниже):

```
1   `timescale 1 ns/ 10 ps
2   module rstrig (Clk, R, S, Q);
3   input Clk, R, S;
4   output Q;
5   wire R_g, S_g, Qa, Qb;
6   reg Clk_r, R_r, S_r;
7   parameter period = 50;
8   initial
9   begin
10       Clk_r = 1'b0;
11       forever #(period /2) Clk_r = ~Clk_r;
12  end
13  initial
14  begin
15       R_r = 1'b0;
16       #40 R_r = 1'b1;
17       #80 R_r = 1'b0;
18       #130 R_r = 1'b1;
19  end
20  initial
21  begin
22       S_r = 1'b0;
23       #80 S_r = 1'b1;
24  end
25  initial
26         #180 $finish;
27  and(R_g, R, Clk);
28  and(S_g, S, Clk);
29  nor(Qa, R_g, Qb);
30  nor(Qb, S_g, Qa);
31  assign Q = Qa;
32  assign Clk = Clk_r;
33  assign R = R_r;
34  assign S = S_r;
35  endmodule
```

3. Скомпилируйте программу. После успешной компиляции перейдите в режим моделирования.
4. Откройте графическое окно и добавьте в него проверяемые сигналы. Запустите проект на моделирование. Ответьте **НЕТ** на вопрос, хотите ли Вы закончить работу с симулятором. Проверьте полученные результаты:



Обратите внимание – в данной работе не используется файл с описанием тестовых векторов. Все изменения входных сигналов задаются с помощью оператора **`initial`**. Само описание триггера выполнено не на поведенческом уровне, а на структурном. Описаны элементы **`and`** и **`nor`** с подключаемыми к ним сигналами.

5. Выйдите из режима моделирования.

4. Задания

4.1. Задание 1. Реализуйте последовательностное устройство и проверьте его работу в среде ModelSim в соответствии с заданным вариантом:

Вариант 1

8-разрядный синхронный параллельный регистр со входами сброса и установки.

Вариант 2

16-разрядный сдвиговый регистр с параллельной загрузкой. Направление сдвига задается логическим уровнем на соответствующем входе.

Вариант 3

Преобразователь последовательного кода интерфейса RS-232 в параллельный код.

Вариант 4

Кольцевой 16-разрядный регистр сдвига. Входные данные – 8-разрядные. Направление и величина сдвига задаются внешними сигналами.

Вариант 5

Блок памяти типа FIFO – восемь 8-разрядных слов.

Вариант 6

Блок ОЗУ – шестнадцать 8-разрядных слов.

4.2. Задание 2. Реализуйте счетчик и проверьте его работу в среде ModelSim в соответствии с заданным вариантом:

Вариант 1

Двоично-десятичный счетчик с параллельной загрузкой.

Вариант 2

Двоичный счетчик по модулю 16 с возможностью реверсивного счета. Направление счета задается логическим уровнем на соответствующем входе.

Вариант 3

4-разрядный счетчик Джонсона. Последовательность изменения состояний данного счетчика следующая: 0h – 1h – 3h – 7h – Fh – Eh – Ch – 8h -0h .

Вариант 4

3-разрядный счетчик-формирователь кода Грея. Последовательность состояний данного счетчика следующая: 0 – 1 – 3 – 2 – 6 – 7 – 5 – 4 – 0.

Вариант 5

3-х канальный генератор импульсных последовательностей с изменяемой фазой.

Вариант 6

Генератор ШИМ-сигнала. Скважность генерируемого сигнала задается входным 4-разрядным кодом.

5. Подготовить и представить отчет о выполненной лабораторной работе в электронном виде.