Глава 4

ЯЗЫКИ ОПИСАНИЯ АППАРАТУРЫ

Цифровая схемотехника и архитектура компьютера, второе издание

Дэвид М. Харрис и Сара Л. Харрис

ELSEVIER

Цифровая схемотехника и архитектура

ЯЗЫКИ ОПИСАНИЯ АППАРАТУРЫ

Эти слайды предназначены для преподавателей, которые читают лекции на основе учебника «Цифровая схемотехника и архитектура компьютера» авторов Дэвида Харриса и Сары Харрис. Бесплатный русский перевод второго издания этого учебника можно загрузить с сайта компании Imagination Technologies:

https://community.imgtec.com/downloads/digital-design-and-computer-architecture-russian-edition-second-edition

Процедура регистрации на сайте компании Imagination Technologies описана на станице:

http://www.silicon-russia.com/2016/08/04/harris-and-harris-2/

Благодарности

ЯЗЫКИ ОПИСАНИЯ АППАРАТУРЫ

Перевод данных слайдов на русский язык был выполнен командой сотрудников университетов и компаний из России, Украины, США в составе:

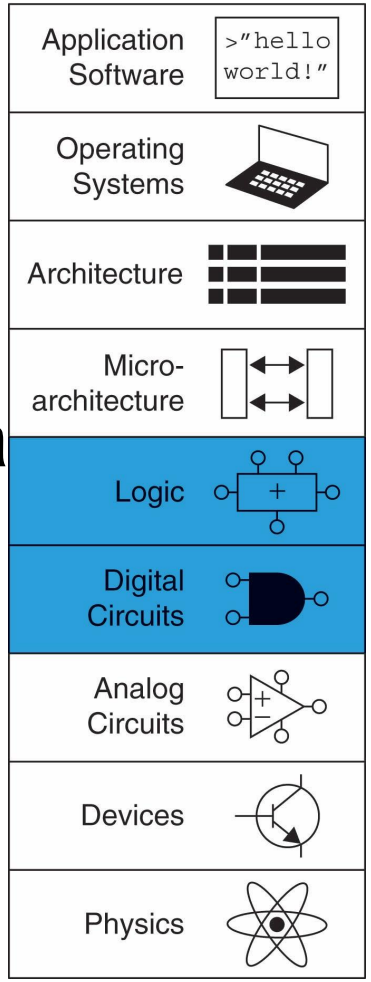
- Александр Барабанов - доцент кафедры компьютерной инженерии факультета радиофизики, электроники и компьютерных систем Киевского национального университета имени Тараса Шевченко, кандидат физ.-мат. наук, Киев, Украина;
- Антон Брюзгин - начальник отдела АО «Вибро-прибор», Санкт-Петербург, Россия.
- Евгений Короткий - доцент кафедры конструирования электронно-вычислительной аппаратуры факультета электроники Национального технического университета Украины «Киевский Политехнический Институт», руководитель открытой лаборатории электроники Lampa, кандидат технических наук, Киев, Украина;
- Евгения Литвинова – заместитель декана факультета компьютерной инженерии и управления, доктор технических наук, профессор кафедры автоматизации проектирования вычислительной техники Харьковского национального университета радиоэлектроники, Харьков, Украина;
- Юрий Панчул - старший инженер по разработке и верификации блоков микропроцессорного ядра в команде MIPS I6400, Imagination Technologies, отделение в Санта-Кларе, Калифорния, США;
- Дмитрий Рожко - инженер-программист АО «Вибро-прибор», магистр Санкт-Петербургского государственного автономного университета аэрокосмического приборостроения (ГУАП), Санкт-Петербург, Россия;
- Владимир Хаханов – декан факультета компьютерной инженерии и управления, проректор по научной работе, доктор технических наук, профессор кафедры автоматизации проектирования вычислительной техники Харьковского национального университета радиоэлектроники, Харьков, Украина;
- Светлана Чумаченко – заведующая кафедрой автоматизации проектирования вычислительной техники Харьковского национального университета радиоэлектроники, доктор технических наук, профессор, Харьков, Украина.

ELSEVIER

Глава 4 : Темы

ЯЗЫКИ ОПИСАНИЯ АППАРАТУРЫ

- **Введение**
- **Комбинационная логика**
- **Структурное моделирование**
- **Последовательностная логика**
- **И снова комбинационная логика**
- **Конечные автоматы**
- **Параметризованные модули**
- **Среда тестирования**



ELSEVIER

Введение

ЯЗЫКИ ОПИСАНИЯ АППАРАТУРЫ

- Языки описания аппаратуры (HDL):
  - Определяют функциональность проектируемого устройства
  - Средства САПР синтезируют оптимизированные схему устройства, состоящую из логических элементов
- Большинство коммерческих проектов построено с использованием языков HDL
- Два лидирующих языка HDL:
  - **SystemVerilog**
    - Разработан в 1984 году компанией Gateway Design Automation
    - Стандарт IEEE standard (1364) – в 1995
    - Расширенный стандарт – в 2005 (IEEE STD 1800-2009)
  - **VHDL 2008**
    - Разработан в 1981 министерством обороны
    - Стандарт IEEE standard (1076) – в 1987
    - Обновлен в 2008 (IEEE STD 1076-2008)

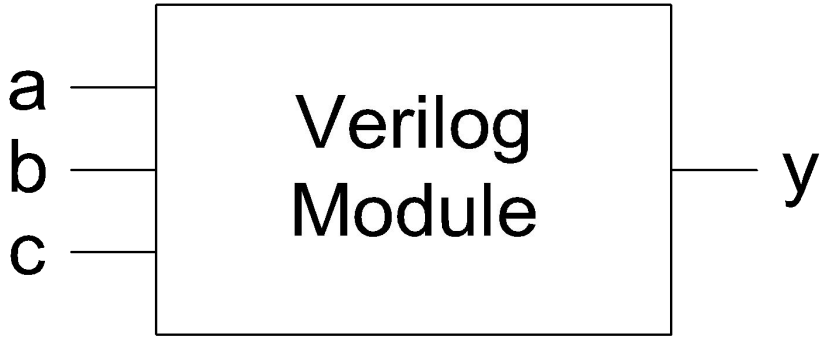
От HDL описания – к логическим

- **Моделирование**
  - Тестовые воздействия подаются на входы
  - Анализ выходов – для проверки корректности работы
  - Миллионы долларов, сэкономленные при отладке в процессе моделирования, – вместо тестирования аппаратуры
- **Синтез**
  - Преобразование HDL кода в список соединений (*netlist)* аппаратного модуля (список элементов и связей между ними)

Важно:

При использовании HDL следует думать об аппаратной реализации HDL кода

Модули SystemVerilog



Два типа модулей:

- **Поведенческий:** описывает что делает модуль
- **Структурный:** определяет модуль как совокупность взаимосвязанных более простых модулей

Поведенческое описание на

SystemVerilog:

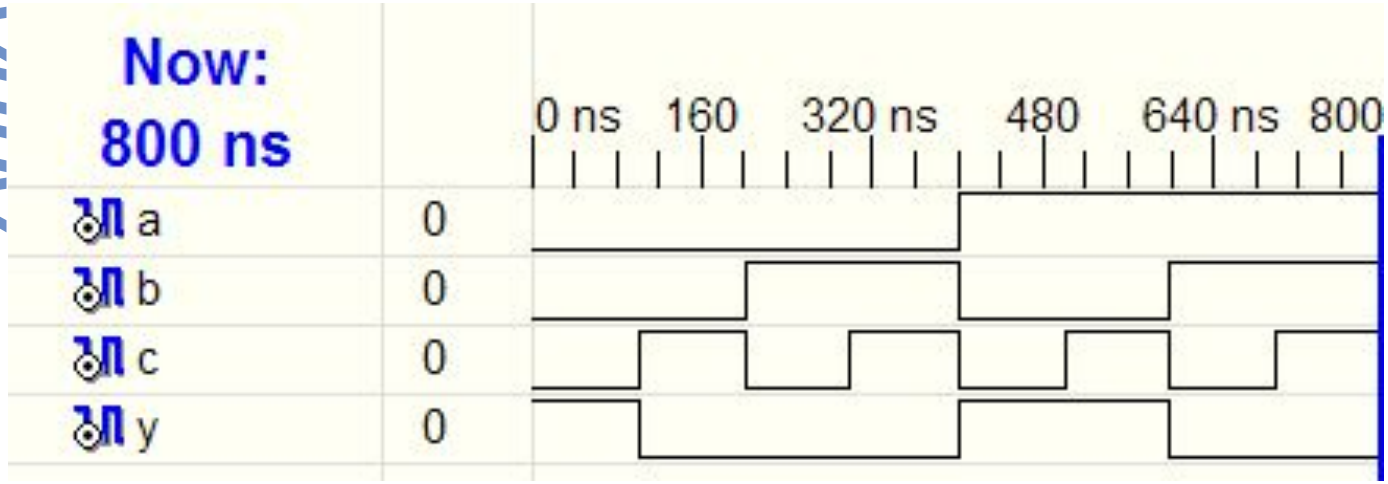
ЯЗЫКИ ОПИСАНИЯ АППАРАТУРЫ

```
module example(input  logic a, b, c,
               output logic y);
  assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b &  c;
endmodule
```

HDL Моделирование

SystemVerilog:

```
module example(input  logic a, b, c,
               output logic y);
  assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b &  c;
endmodule
```

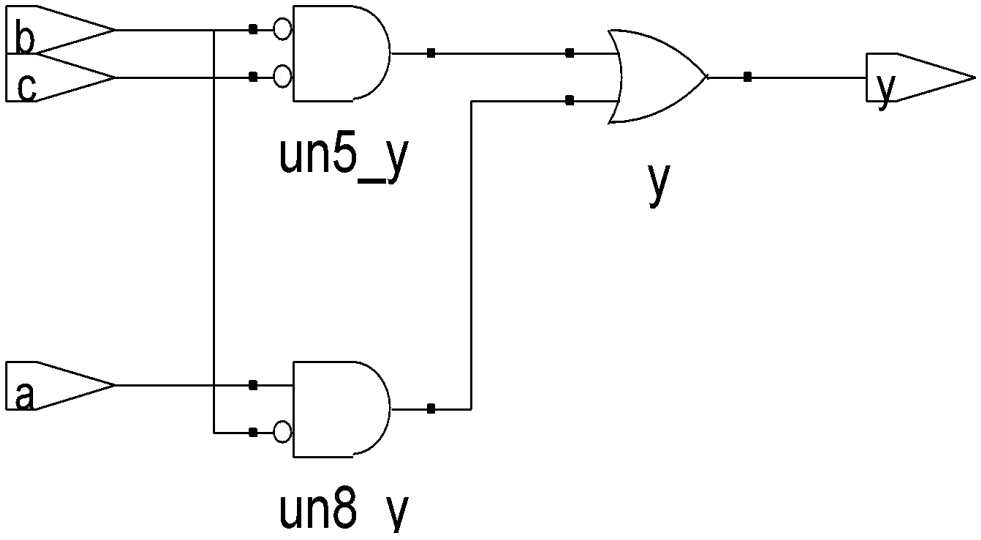


HDL Синтез

SystemVerilog:

```
module example(input  logic a, b, c,
               output logic y);
  assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b &  c;
endmodule
```

Синтез:



Синтаксис SystemVerilog

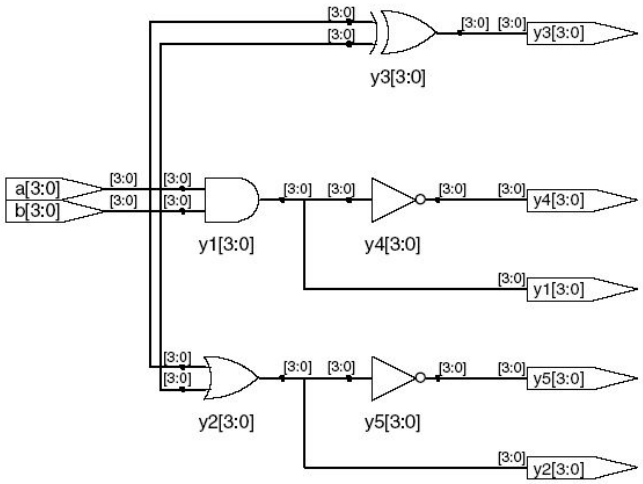
- Чувствительный к регистру символов
  - **Пример:** `reset` и `Reset` не одно и то же.
- Имена не могут начинаться с цифры
  Пример: `2mux` – некорректное имя
- Пробелы игнорируются
- Комментарии:
  - `// однострочный комментарий`
  - `/* многострочный`
    `комментарий */`

Синтез структурных моделей - иерархия


```
module and3(input  logic a, b, c,
            output logic y);
  assign y = a & b & c;
endmodule


module inv(input  logic a,
           output logic y);
  assign y = ~a;
endmodule


module nand3(input  logic a, b, c
             output logic y);
  logic n1;                      // внутренний сигнал

  and3 andgate(a, b, c, n1);  // экземпляр and3
  inv  inverter(n1, y);       // экземпляр inverter
endmodule
```

Поразрядные операторы


```
module gates(input  logic [3:0]  a, b,
             output logic [3:0] y1, y2, y3, y4, y5);
   /* Five different two-input logic
      gates acting on 4 bit busses */
   assign y1 = a & b;     // AND
   assign y2 = a | b;     // OR
   assign y3 = a ^ b;     // XOR
   assign y4 = ~(a & b); // NAND
   assign y5 = ~(a | b); // NOR
endmodule
```

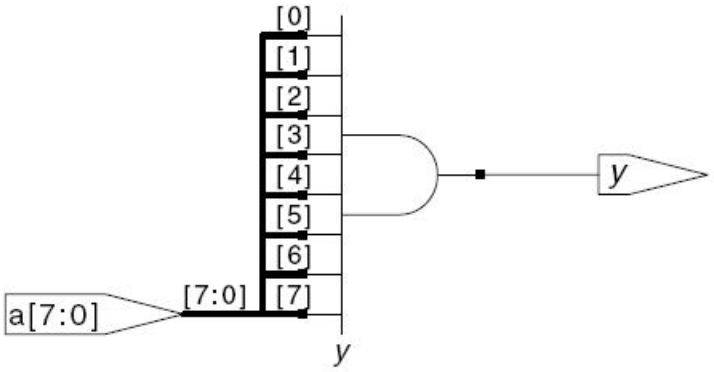


`//` комментарий в одной строке

`/*...*/` комментарий в нескольких строках

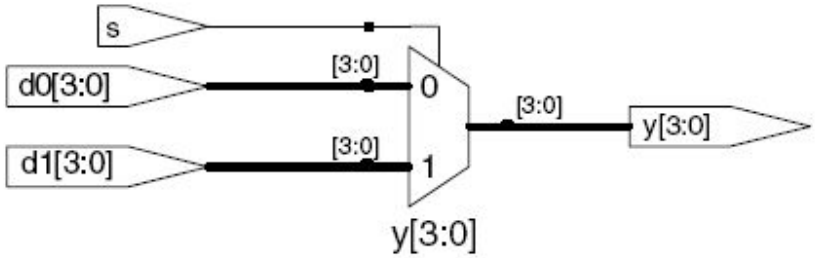
Операторы сокращения


```
module and8(input  logic [7:0] a,
            output logic       y);
   assign y = &a;
   // &a is much easier to write than
   // assign y = a[7] & a[6] & a[5] & a[4] &
   //            a[3] & a[2] & a[1] & a[0];
endmodule
```



Условное присваивание

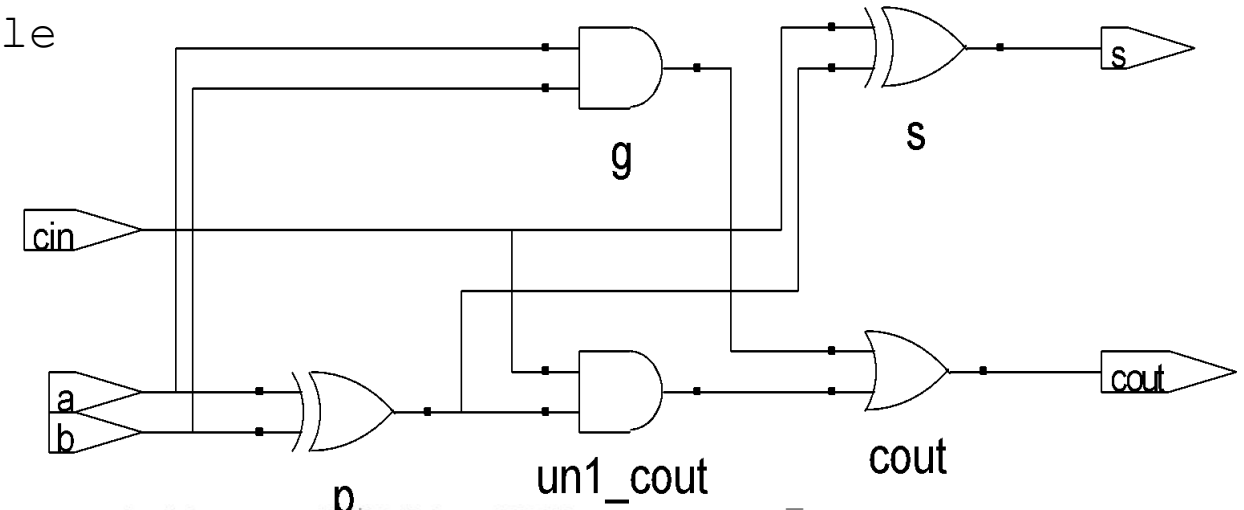

```
module mux2(input  logic [3:0] d0, d1,
            input  logic       s,
            output logic [3:0] y);
   assign y = s ? d1 : d0;
endmodule
```



`?:` также называется тернарным оператором потому, что он имеет 3 входа: `s`, `d1` и `d0`.

Внутренние сигналы


```
module fulladder(input  logic a, b, cin,
                 output logic s, cout);
  logic p, g;    // internal nodes

  assign p = a ^ b;
  assign g = a & b;

  assign s = p ^ cin;
  assign cout = g | (p & cin);
endmodule
```



Приоритет операций

Порядок операций

Высший

~	Отрицание
*, /, %	Умножение, деление, остаток
+, -	Сложение, вычитание
<<, >>	Сдвиг
<<<, >>>	Арифметический сдвиг
<, <=, >, >=	Сравнение (больше-меньше)
==, !=	Сравнение на равенство
&, ~&	И, И-НЕ
^, ~^	Исключающее ИЛИ, исключающее ИЛИ-НЕ
|, ~|	ИЛИ, ИЛИ-НЕ
?:	Тернарный оператор

Низший

Формы представления чисел

Формат: N'B value

N = количество разрядов, **B** = основание

N'B не является обязательным, но рекомендуется (по умолчанию используется десятичная система)

Число	Кол-во разрядов	Основание	Десятичный эквивалент	Число в памяти
3'b101	3	Двоичное	5	101
'b11	Не определено	Двоичное	3	00…0011
8'b11	8	Двоичное	3	00000011
8'b1010_1011	8	Двоичное	171	10101011
3'd6	3	Десятичное	6	110
6'o42	6	Восьмеричное	34	100010
8'hAB	8	Шестнадцатеричное	171	10101011
42	Не определено	Десятичное	42	00…0101010

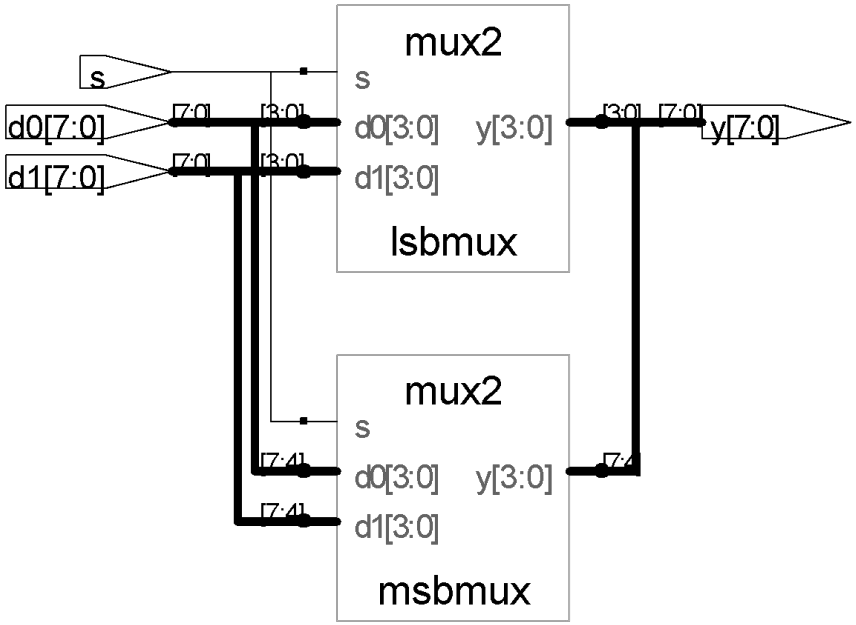
Работа с битами: Пример 1


```
assign y = {a[2:1], {3{b[0]}}, a[0], 6'b100_010};

// если y - 12-битовый сигнал, оператор выше сформирует:
y = a[2] a[1] b[0] b[0] b[0] a[0] 1 0 0 0 1 0

// подчеркивание (_) используется только для
// удобочитаемости. SystemVerilog его игнорирует.
```

Работа с битами: Пример 2

SystemVerilog:

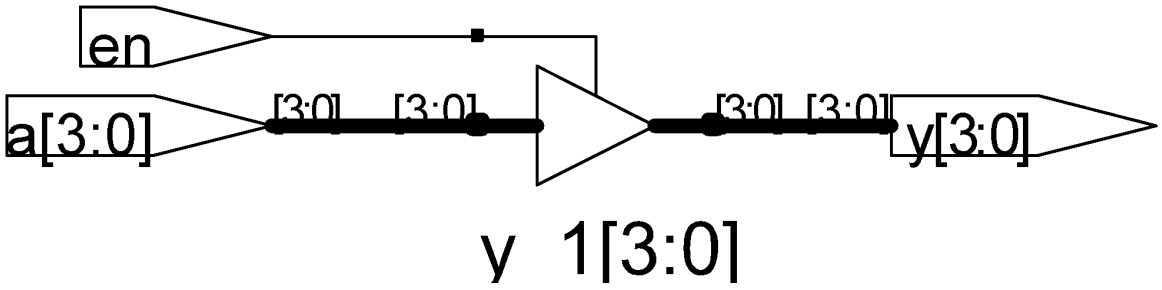
```
module mux2_8(input  logic [7:0] d0, d1,
              input  logic       s,
              output logic [7:0] y);

  mux2 lsbmux(d0[3:0], d1[3:0], s, y[3:0]);
  mux2 msbmux(d0[7:4], d1[7:4], s, y[7:4]);
endmodule
```



Z: Неподключенное (высокоимпедансное) состояние

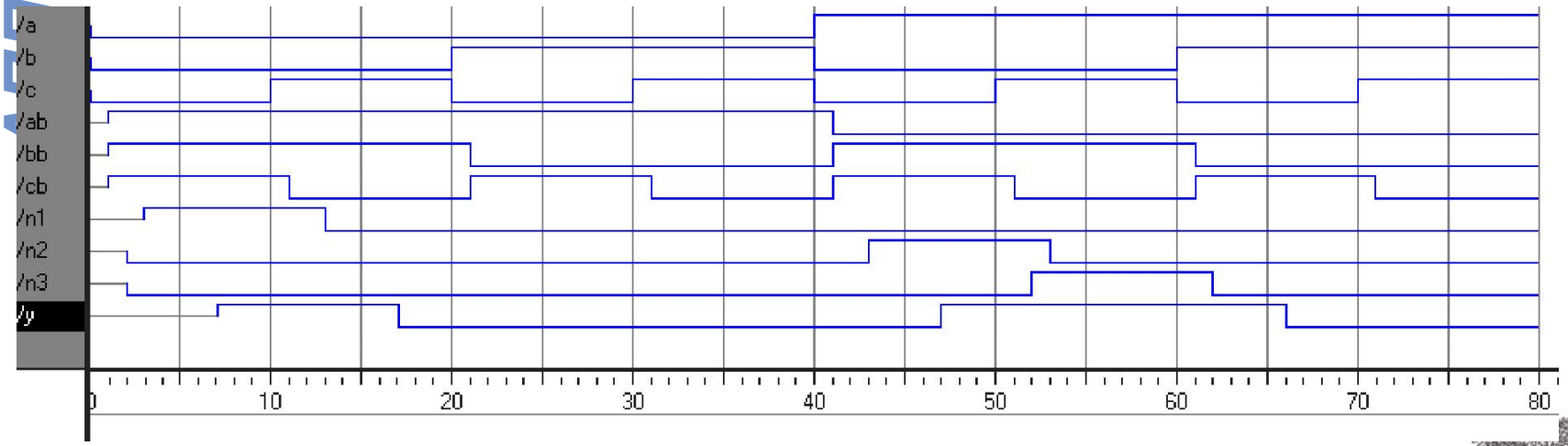
SystemVerilog:

```
module tristate(input  logic [3:0] a,
                input  logic       en,
                output logic [3:0] y);
   assign y = en ? a : 4'bz;
endmodule
```



Задержки


```
module example(input  logic a, b, c,
               output logic y);
  logic ab, bb, cb, n1, n2, n3;
  assign #1 {ab, bb, cb} = ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```



Задержки


```
module example(input  logic a, b, c,
               output logic y);
  logic ab, bb, cb, n1, n2, n3;
  assign #1 {ab, bb, cb} =
                  ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```

Последовательностная

- SystemVerilog использует **идиомы** для описания защелок, триггеров и конечных автоматов
- Произвольные стили HDL кодирования могут моделироваться правильно, но результат синтеза может не соответствовать ни результат моделирования, ни желаниям разработчика

Оператор Always

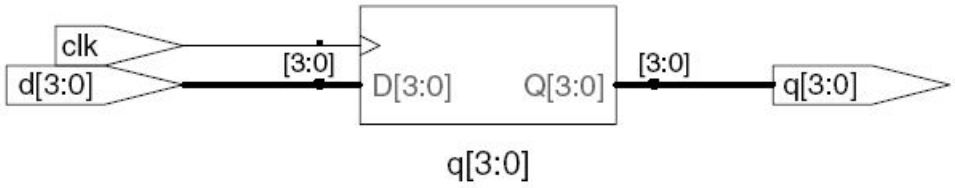
Общая структура:

```
always @(sensitivity list)
  statement;
```

Всякий раз, когда происходит событие из списка `sensitivity list`, выполняется оператор `statement`

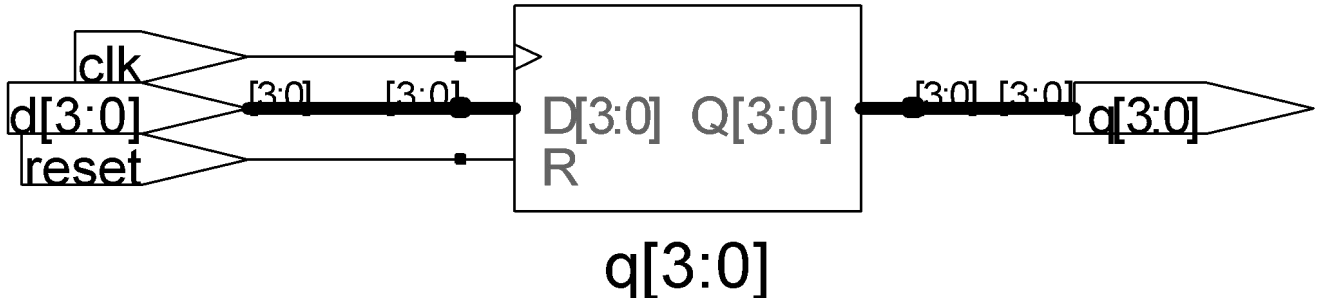
D триггер


```
module flop(input  logic       clk,
            input  logic [3:0] d,
            output logic [3:0] q);

  always_ff @(posedge clk)
    q <= d;                  //произносится "q получает d"

endmodule
```



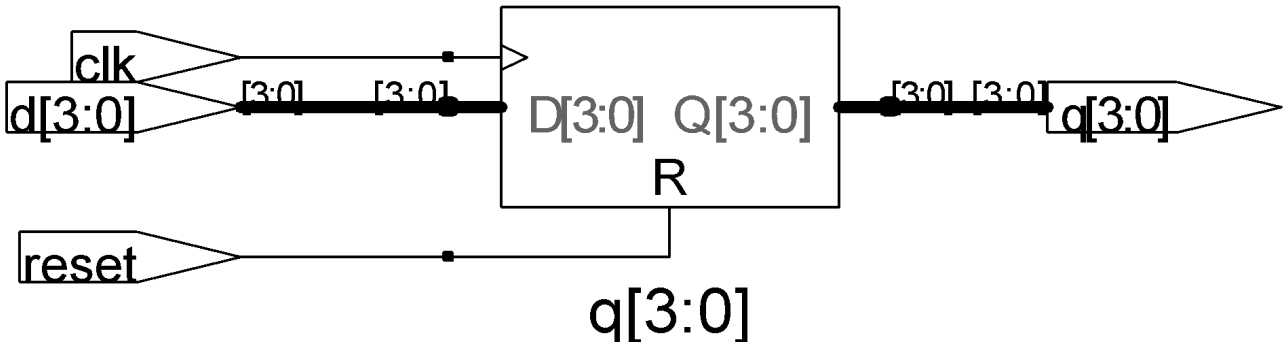
D триггер со сбросом


```
module flopr(input  logic        clk,
             input  logic        reset,
             input  logic [3:0]  d,
             output logic [3:0]  q);

  // синхронный сброс
  always_ff @(posedge clk)
    if (reset) q <= 4'b0;
    else       q <= d;

endmodule
```



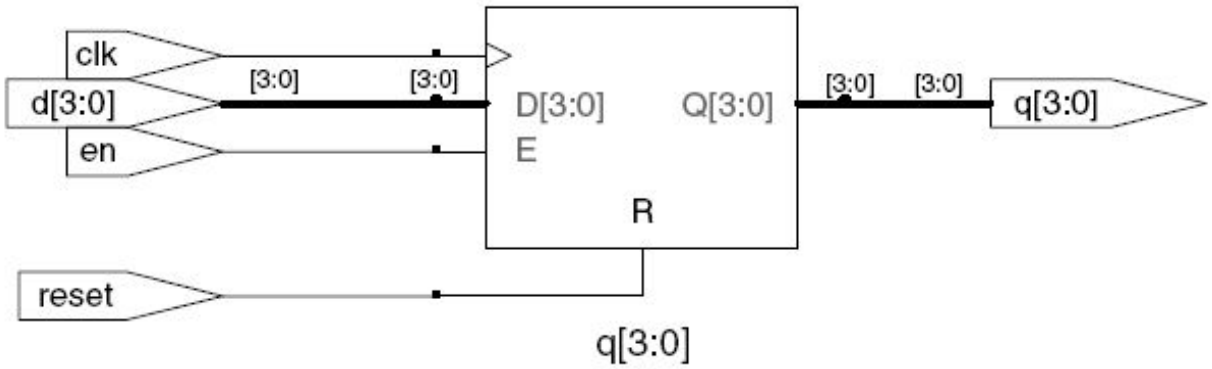
D триггер со сбросом


```
module flopr(input  logic        clk,
             input  logic        reset,
             input  logic [3:0]  d,
             output logic [3:0]  q);

  // асинхронный сброс
  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 4'b0;
    else       q <= d;

endmodule
```



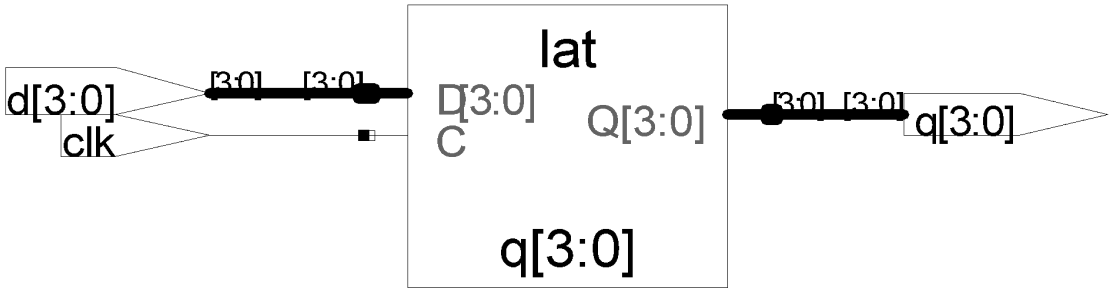
D триггер с сигналом


```
module flopren(input  logic        clk,
               input  logic        reset,
               input  logic        en,
               input  logic [3:0]  d,
               output logic [3:0]  q);

  // асинхронный сброс и разрешение
  always_ff @(posedge clk, posedge reset)
    if      (reset) q <= 4'b0;
    else if (en)    q <= d;

endmodule
```



Защелки


```
module latch(input  logic        clk,
             input  logic [3:0] d,
             output logic [3:0] q);

  always_latch
    if (clk) q <= d;

endmodule
```



Внимание: Мы не используем защелки в нашел курсе.
Но вы можете написать код, который непреднамеренно реализует защелку.
Проверьте синтезированный аппаратный модуль – если он имеет защелку в нем, то вы, вероятно, совершили ошибку.

Другие поведенческие

- Операторы, которые должны быть расположены внутри оператора `always` :
  - `if/else`
  - `case, casez`

Комбинационная логика с always


```
//комбинационная логика с использованием оператора always
module gates(input  logic [3:0] a, b,
             output logic [3:0] y1, y2, y3, y4, y5);
  always_comb          // need begin/end because there is
    begin              // more than one statement in always
      y1 = a & b;      // AND
      y2 = a | b;      // OR
      y3 = a ^ b;      // XOR
      y4 = ~(a & b);   // NAND
      y5 = ~(a | b);   // NOR
    end
endmodule
```

Этот аппаратный модуль может быть описан с помощью оператора непрерывного присваивания `assign` **с меньшим количеством строк кода, так что в этом случае лучше использовать операции непрерывного присваивания.**

Комбинационная логика с


```
module sevenseg(input  logic [3:0] data,
                output logic [6:0] segments);
  always_comb
    case (data)
      //                      abc_defg
      0: segments =       7'b111_1110;
      1: segments =       7'b011_0000;
      2: segments =       7'b110_1101;
      3: segments =       7'b111_1001;
      4: segments =       7'b011_0011;
      5: segments =       7'b101_1011;
      6: segments =       7'b101_1111;
      7: segments =       7'b111_0000;
      8: segments =       7'b111_1111;
      9: segments =       7'b111_0011;
      default: segments = 7'b000_0000; // необходимо
    endcase
endmodule
```

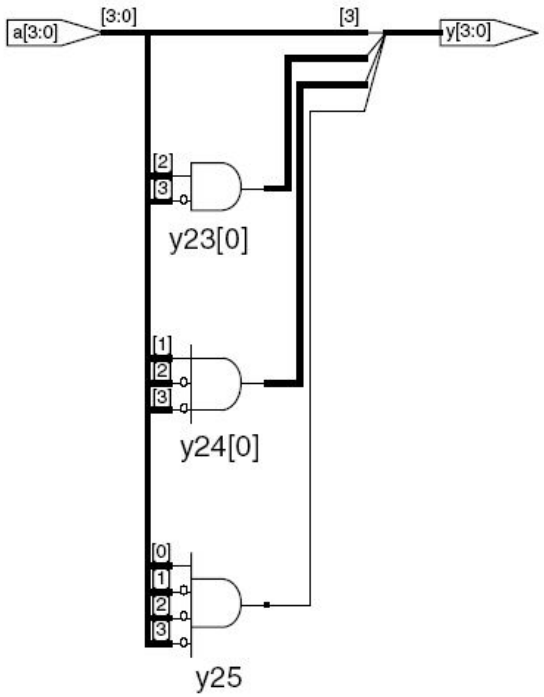
Комбинационная логика с

- Оператор `Case` реализует комбинационную логику, **только если** в его ветвях перечислены все возможные входные комбинации
- Помните об использовании **`default`** (выбор по умолчанию)

Комбинационная логика с

ЯЗЫКИ ОПИСАНИЯ АППАРАТУРЫ

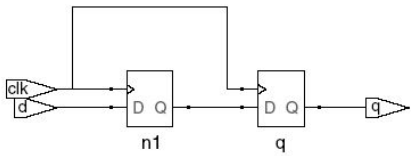
```
module priority_casez(input  logic [3:0] a,
                      output logic [3:0] y);

  always_comb
    casez(a)
      4'b1???: y = 4'b1000;  // ? = don't care
      4'b01??: y = 4'b0100;
      4'b001?: y = 4'b0010;
      4'b0001: y = 4'b0001;
      default: y = 4'b0000;
   endcase
endmodule
```



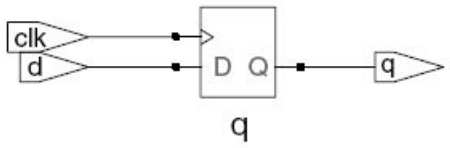
Блокирующие и неблокирующие присваивания

- <= неблокирующее присваивание
- Выполняется одновременно с другими
- = блокирующее присваивание
- Выполняется в порядке, описанном в файле

```
// Хороший синхронизатор,
   использующий
// неблокирующее присваивание
module syncgood(input  logic clk,
                input  logic d,
                output logic q);
  logic n1;
  always_ff @(posedge clk)
    begin
      n1 <= d;  // nonblocking
      q  <= n1; // nonblocking
    end
endmodule
```




```
// Плохой синхронизатор,
   использующий
// блокирующее присваивание
module syncbad(input logic  clk,
               input  logic d,
               output logic q);
  logic n1;
  always_ff @(posedge clk)
    begin
      n1 = d;  // blocking
      q  = n1; // blocking
    end
endmodule
```



Правила присваивание сигналов

- **Синхронная последовательная логика:** использует **`always_ff @(posedge clk)`** и неблокирующее присваивание (**<=**)

```
always_ff @ (posedge clk)
  q <= d; // nonblocking
```

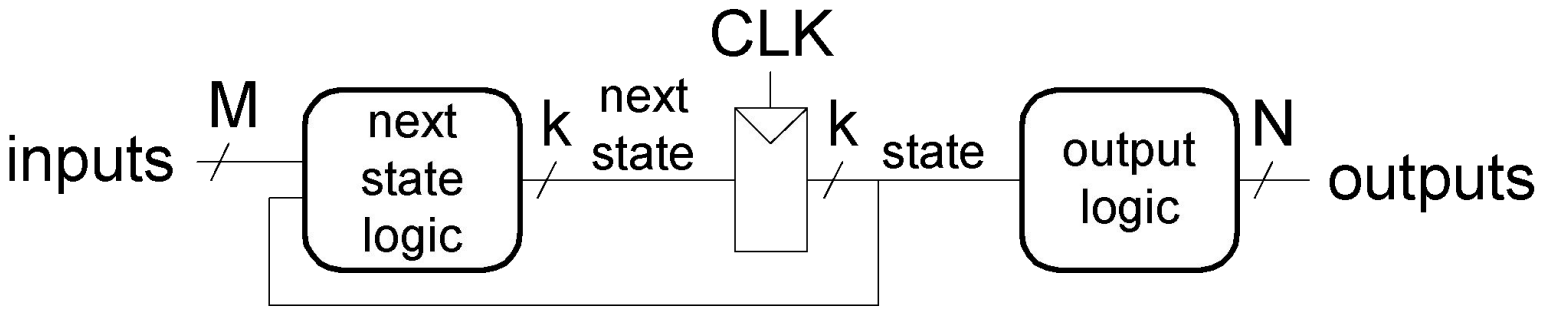
- **Простая комбинационная логика:** использует непрерывное присваивание (**`assign`**…)

```
assign y = a & b;
```

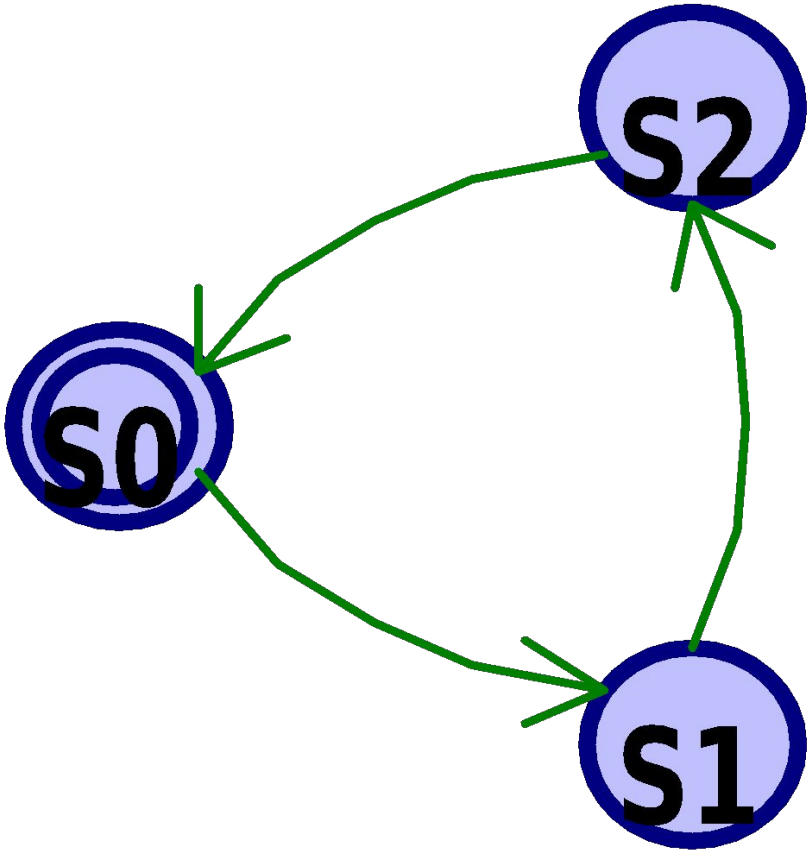
- **Более сложная комбинационная логика:** использует **`always_comb`** и блокирующее присваивание (**=**)
- Сигнал изменяется **только одним** оператором `always` или оператором непрерывного присваивания (попытка изменить сигнал несколькими операторами `always` или `assign` без использования отключенного состояния приведет к конфликту и ошибке синтеза).

Конечный автомат

- **Три блока:**
  - Логика следующего состояния
  - Регистр состояний
  - Логика выходов



Пример конечного автомата: Делитель на 3



Двойной круг определяет состояние сброса

Конечный автомат на


```
module divideby3FSM (input  logic clk,
                     input  logic reset,
                     output logic q);
   typedef enum logic [1:0] {S0, S1, S2} statetype;
   statetype [1:0] state, nextstate;

   // регистр состояний
   always_ff @ (posedge clk, posedge reset)
      if (reset) state <= S0;
      else       state <= nextstate;

   // логика следующего состояния
   always_comb
      case (state)
         S0:      nextstate = S1;
         S1:      nextstate = S2;
         S2:      nextstate = S0;
         default: nextstate = S0;
      endcase

   // логика выходных сигналов
   assign q = (state == S0);
endmodule
```

Параметризированные

2:1 мультиплексор:

```
module mux2
  #(parameter width = 8)  // name and default value
   (input  logic [width-1:0] d0, d1,
    input  logic             s,
    output logic [width-1:0] y);
  assign y = s ? d1 : d0;
endmodule
```

Пример с 8-битной шиной (используется по умолчанию):

```
  mux2 mux1(d0, d1, s, out);
```

Пример с 12-битной шиной :

```
    mux2 #(12) lowmux(d0, d1, s, out);
```

Среда тестирования (Testbenches)

- HDL модуль, который тестирует другой модуль: тестируемое устройство (DUT)
- **Не** предназначена для синтеза
- Типы:
  - Простая
  - С самопроверкой
  - С самопроверкой и тестовыми векторами

Пример среды тестирования

- Написать System Verilog код для аппаратной реализации следующей функции:

$$y = \overline{bc}\ \overline{+}\ \overline{ab}$$

- Имя модуля `sillyfunction`

Пример среды тестирования

- Написать System Verilog код для аппаратной реализации следующей функции:

$$y = \bar{b}\bar{c} + a\bar{b}$$

```
module sillyfunction(input  logic a, b, c,
                     output logic y);
  assign y = ~b & ~c | a & ~b;
endmodule
```

Простая среда тестирования


```
module testbench1();
  logic a, b, c;
  logic y;
  // экземпляр проверяемого устройства
  sillyfunction dut(a, b, c, y);
  // последовательно формируются значения
  // сигналов на входах
initial begin
    a = 0; b = 0; c = 0; #10;
    c = 1; #10;
    b = 1; c = 0; #10;
    c = 1; #10;
    a = 1; b = 0; c = 0; #10;
    c = 1; #10;
    b = 1; c = 0; #10;
    c = 1; #10;
  end
endmodule
```

Среда тестирования с самопроверкой


```
module testbench2();
  logic  a, b, c;
  logic y;
  sillyfunction dut(a, b, c, y); // экземпляр dut
  initial begin // последовательно формируются значения сигналов
// на входах и анализирует результат тестирования на выходах
     a = 0; b = 0; c = 0; #10;
    if (y !== 1) $display("000 failed.");
    c = 1; #10;
    if (y !== 0) $display("001 failed.");
    b = 1; c = 0; #10;
    if (y !== 0) $display("010 failed.");
    c = 1; #10;
    if (y !== 0) $display("011 failed.");
    a = 1; b = 0; c = 0; #10;
    if (y !== 1) $display("100 failed.");
    c = 1; #10;
    if (y !== 1) $display("101 failed.");
    b = 1; c = 0; #10;
    if (y !== 0) $display("110 failed.");
    c = 1; #10;
    if (y !== 0) $display("111 failed.");
  end
endmodule
```

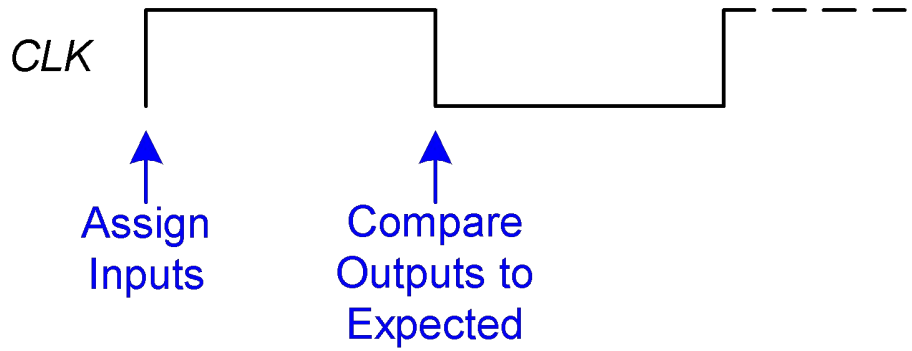
Среда тестирования с тестовыми

- Файл тестовых векторов: входные сигналы и ожидаемые состояния выходов
- Среда тестирования:

1. Формирование тактового сигнала для изменения входов, считывание выходных сигналов
2. Считывание тестовых векторов из файла в буферный массив для последующей подачи их на входы
3. Присвоение значении входным сигналам, определение ожидаемых значений выходных сигналов
4. Сравнение реальных значений выходных сигналов

Среда тестирования с тестовыми векторами

- Среда тестирования, тактовый сигнал:
  - Изменение входных сигналов по переднему фронту тактового сигнала
  - Сравнение состояний выходов с ожидаемыми значениями по заднему фронту тактового сигнала



- Тактовый сигнал среды тестирования также используется для синхронизации последовательностных схем

Файл тестовых векторов

- Файл: `example.tv`
- Содержит вектора abc_yexpected

```
000_1
001_0
010_0
011_0
100_1
101_1
110_0
111_0
```

1. Генерация тактового


```
module testbench3();
  logic        clk, reset;
  logic        a, b, c, yexpected;
  logic        y;
  logic [31:0] vectornum, errors;    // bookkeeping variables
  logic [3:0]  testvectors[10000:0]; // array of testvectors

  // создание экземпляра тестируемого устройства
  sillyfunction dut(a, b, c, y);

  // генерация тактового сигнала
  always     // no sensitivity list, so it always executes
    begin
      clk = 1; #5; clk = 0; #5;
    end
```

2. Считывание тестовых векторов в массив


```
// при запуске теста загрузка векторов и генерация сигнала сброса
initial
  begin
    $readmemb("example.tv", testvectors);
    vectornum = 0; errors = 0;
    reset = 1; #27; reset = 0;
  end

// Примечание: $readmemh считывает файл тестовых векторов,
// записанных в шестнадцатеричном представлении
```

3. Назначение входов & ожидаемые состояния выходов


```
// подача тестовых векторов по переднему фронту
// синхросигнала
always @(posedge clk)
  begin
    #1; {a, b, c, yexpected} = testvectors[vectornum];
  end
```

4. Сравнение выходных сигналов с ожидаемыми


```
// проверка результатов по заднему фронту синхросигнала
   always @(negedge clk)
    if (~reset) begin // skip during reset
      if (y !== yexpected) begin
        $display("Error: inputs = %b", {a, b, c});
        $display("  outputs = %b (%b expected)",y,yexpected);
        errors = errors + 1;
      end

// Примечание: для вывода на печать в шестнадцатеричном коде
// (hexadecimal), используйте %h. Например,
//       $display("Error: inputs = %h", {a, b, c});
```

4. Сравнение выходных сигналов с ожидаемыми


```
// инкремент индекса массива и считывание очередного
// тестового вектора
      vectornum = vectornum + 1;
      if (testvectors[vectornum] === 4'bx) begin
          $display("%d tests completed with %d errors",
                vectornum, errors);
        $finish;
      end
    end
endmodule


// === and !== can compare values that are 1, 0, x, or z.
```