

Комбинационные схемы (КС)

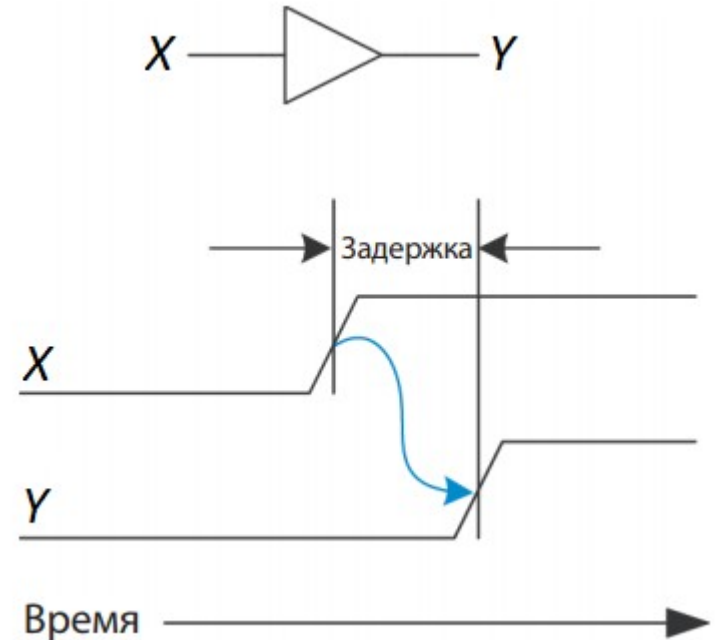
- КС - схема, реализующая логическую функцию и не имеющая обратных связей
- Значение на выходе комбинационной схемы зависит только от значений на входе в текущий момент времени

$$(y_0, y_1, \dots, y_k) = f(x_0, x_1, \dots, x_k), \text{ где}$$

$$y_i \neq f(x_i)$$

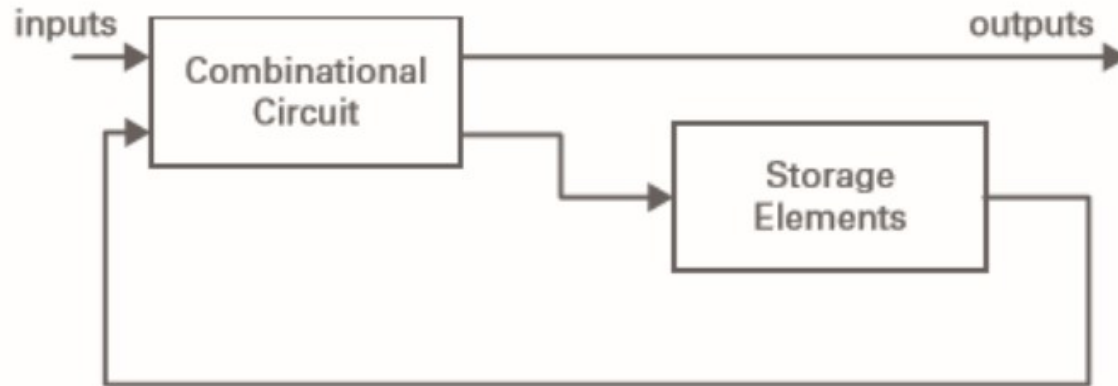
На практике изменение значения выходных сигналов запаздывает по сравнению с моментом изменения входных сигналов

- Задержка распространения (propagation delay) КС – это максимальное время от момента изменения входных сигналов до момента установки всех выходных сигналов
- Чем больше последовательно соединенных элементов содержит КС, тем больше задержка распространения



Последовательностные схемы

- Схема, выходные значения которой зависят не только от текущих, но и предыдущих значений входных сигналов
- Содержит элемент памяти (например, триггер), хранящий предыдущее состояние системы



Последовательностные схемы

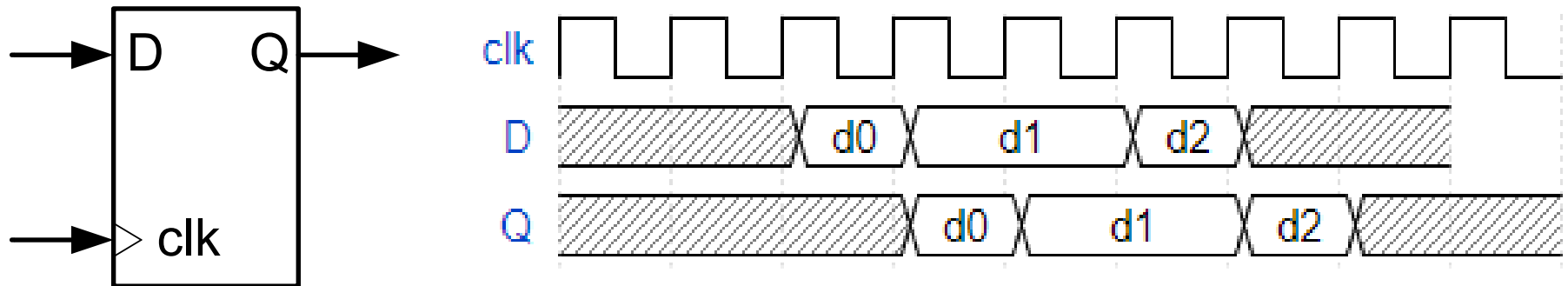
- Синхронные – состояние и выходные сигналы изменяются по сигналу синхронизации
- Асинхронные – состояние и выходные сигналы изменяются при изменении входных сигналов в любой момент времени

Схема является синхронной, если:

- Каждый элемент схемы является либо регистром, либо КС
- Как минимум один элемент схемы является регистром
- Все изменения сигналов схемы синхронизированы с одним тактовым сигналом
- В каждом циклическом пути присутствует регистр

Триггер

- Простейшая синхронная схема с двумя состояниями {0, 1}
- Сигнал со входа D передается на выход Q по переднему фронту тактового сигнала
- Упорядоченная последовательность триггеров с общим тактовым сигналом называется регистром



Описание триггера без дополнительных сигналов

```
always @(posedge clk) begin  
    q <= d;  
end
```

```
always_ff @(posedge clk) begin  
    q <= d;  
end
```

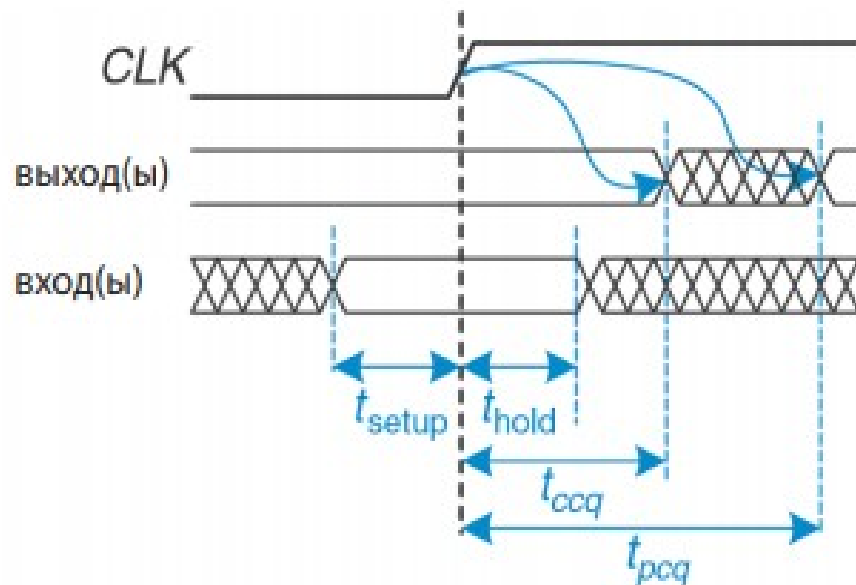
Временные характеристики триггера

Для корректной работы триггера необходимо, чтобы до и после фронта тактового сигнала данные на его входах были стабильными в течение определенного промежутка времени, определяемого конструкцией триггера

- Время до фронта тактового импульса – время предустановки (setup time)
- Время после фронта тактового импульса – время удержания (hold time)

Временные характеристики триггера

- t_{ccq} (clock-to-Q contamination delay) – наименьшая задержка схемы
- t_{pcq} (clock-to-Q propagation delay) – наибольшая задержка схемы
- Период тактовых импульсов должен быть достаточно большим, чтобы переходные процессы всех сигналов успели завершиться



Сигнал сброса

- Триггер может содержать сигнал сброса rst
- При подаче сигнала сброса, триггер обнуляет данные на выходе
- Сброс может быть синхронным или асинхронным
- Использование синхронного сброса предпочтительнее

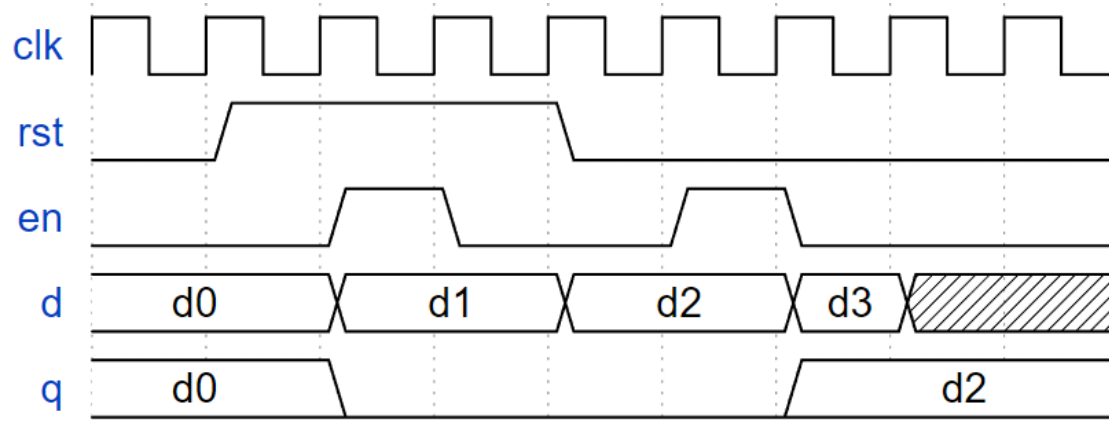
Сигнал разрешения

- Обычно обозначается en
- При отсутствии сигнала разрешения, триггер игнорирует входные данные по приходу тактового сигнала

Описание триггера с синхронным сбросом

```
always_ff @(posedge clk) begin
    if(rst)
        q <= 0;
    else
        q <= d;
end
```

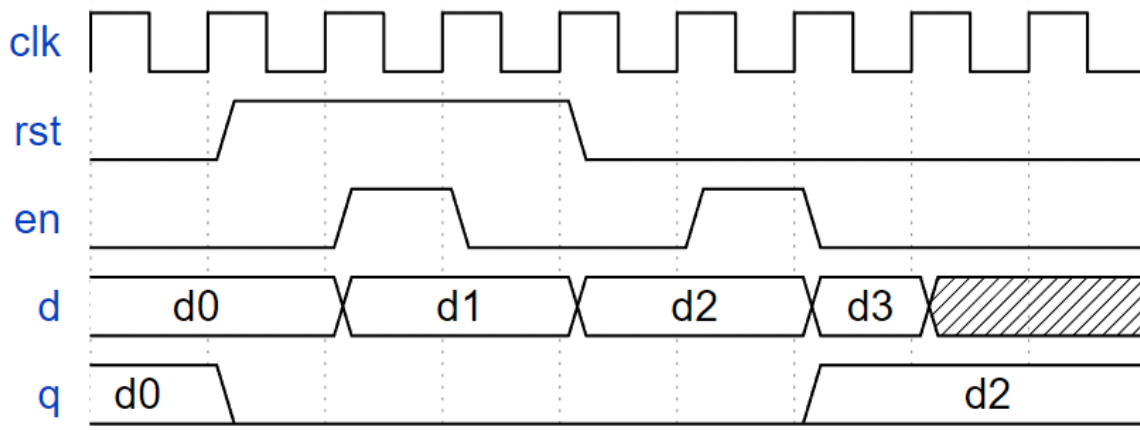
```
always_ff @(posedge clk) begin
    if(rst)
        q <= 0;
    else if(en)
        q <= d;
end
```



Описание триггера с асинхронным сбросом

```
always_ff @(posedge clk or posedge rst) begin
    if(rst)
        d <= q;
    else
        d <= q;
end
```

```
always_ff @(posedge clk or posedge rst) begin
    if(rst)
        d <= q;
    else if(en)
        d <= q;
end
```



- Нельзя делать присвоения в один триггер в двух разных `always` блоках
- Реакция триггера на сигнал `rst` всегда описывается в начале `always` блока
- В списке чувствительности `always` блока для асинхронного триггера присутствует `rst`
- Рекомендуется без необходимости не делать триггеры с возможностью сброса

Присваивания в SV

- Программа SV – совокупность процедурных блоков `always` и непрерывных присваиваний
- Их выполнение параллельно
- Порядок их исполнения не зависит от очередности, в которой они написаны
- Внутри процедурного блока инструкции выполняются последовательно

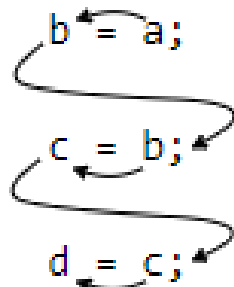
```
module my_module(  
    input logic clk,  
    input logic in,  
    input logic out  
);  
    // комбинационная логика  
    always_comb begin  
        // ...smth  
    end  
  
    // триггер  
    always_ff @(posedge clk) begin  
        // ...smth  
    end  
  
    // комбинационная логика  
    assign out = ~in; // непрерывное присваивание  
  
endmodule
```

Блокирующие и неблокирующие присваивания

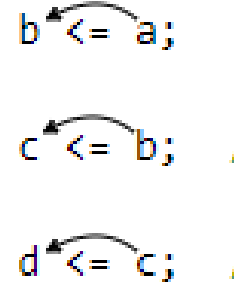
- Присваивания в процедурных блоках делятся на блокирующие (=) и неблокирующие (<=)
- Блокирующее присваивание блокирует исполнение дальнейших выражений до завершения вычисления в правой части и присваивания вычисленного результата левой
- Неблокирующее присваивание производит вычисление правой части и откладывает присваивание вычисленного значения, позволяя выполняться остальным выражениям до завершения присваивания левой части

Блокирующие и неблокирующие присваивания

```
logic a, b;  
logic c, d;  
always_ff @(posedge clk) begin  
    b ← a;  
    c ← b; // обновленное значение b  
    d ← c; // обновленное значение c  
end
```



```
logic a, b;  
logic c, d;  
always_ff @(posedge clk) begin  
    b ←<= a;  
    c ←<= b; // старое значение b  
    d ←<= c; // старое значение c  
end
```



Правила хорошего тона

- При описании последовательной логики используется неблокирующее присваивание
- При описании комбинационной логики используется блокирующее присваивание

Конструкция generate

- Позволяет выборочно включать/исключать блоки кода (условная генерация) или создавать несколько экземпляров блока кода (итерационная генерация)
- Может быть использована совместно с циклом for или операторами if, case
- Не может быть использована внутри процедурных блоков

Итерационная генерация

```
module example_for_generate (
    input logic [31:0] in,
    output logic [31:0] out
);
    genvar i;
    generate
        for (i = 0; i < 4; i = i + 1) begin: set_of_my_modules
            my_module exp(
                .in(in[i*8+7:i*8]),
                .out(out[i*8+7:i*8])
            );
        end
    endgenerate
endmodule
```

```
module example_for_generate_copy (
    input logic [31:0] in,
    output logic [31:0] out
);
    my_module exp_0(
        .in(in[7:0]),
        .out(out[7:0])
    );
    my_module exp_1(
        .in(in[15:8]),
        .out(out[15:8])
    );
    my_module exp_2(
        .in(in[23:16]),
        .out(out[23:16])
    );
    my_module exp_3(
        .in(in[31:24]),
        .out(out[31:24])
    );
endmodule
```

Итерационная генерация

- Переменная цикла объявляется с ключевым словом `genvar`
- Цикл объявляется внутри блока генерации, а не в процедурном блоке `always`

Условная генерация

- На основе условия создается только один экземпляр модуля

```
module example_if_generate#(  
    parameter GEN = 1  
) (  
    input  logic [31:0] in,  
    output logic [31:0] out  
);  
    generate  
        if (GEN) begin: gen_my_module_1  
            my_module_1 exp(  
                .in(in),  
                .out(out)  
            );  
        end else begin: gen_my_module_2  
            my_module_2 exp(  
                .in(in),  
                .out(out)  
            );  
        end  
    endgenerate  
endmodule
```

Знаковая арифметика

- Знаковые числа представляются в дополнительном коде
- Признак отрицательного числа – это единица в старшем разряде
- Это позволяет производить сложение и вычитание при помощи одной схемы

7	4'b0111
6	4'b0110
5	4'b0101
4	4'b0100
3	4'b0011
2	4'b0010
1	4'b0001
0	4'b0000
-1	4'b1111
-2	4'b1110
-3	4'b1101
-4	4'b1100
-5	4'b1011
-6	4'b1010
-7	4'b1001
-8	4'b1000

- По умолчанию переменные являются беззнаковыми
- Для объявления знаковой переменной используется `signed`
- При знаковых арифметических операциях нужно следить за размерностью результата и знаком операндов
- Для перевода переменных используются ключевые слова `signed'()` / `unsigned'()` или системные функции `$signed()` / `$unsigned()`

Пример 1

- Если хотя бы один из операндов не объявлен как знаковый, будет произведено беззнаковое сложение

```
logic signed[4:0]  a;
logic unsigned[4:0] b;
logic signed[5:0]  c;

a = -4'd3; // 5'b11101
a = unsigned'(-4'd3); // 5'b01101
a = $unsigned(-4'd3); // 5'b01101
a = signed'(-4'd3); // 5'b11101
b = 1;
c = a + b; // 6'b011110 // 1e
c = $signed(a) + $signed(b); // 6'b111110 // 3e
```

Пример 2

- Если знаковой 4-х битной переменной присвоить число больше 7, то она будет восприниматься отрицательной
- Если оба операнда знаковые, то будет произведено знаковое суммирование
- Объявление переменной для записи результата, не влияет на то, какое будет производится суммирование

```
logic signed[3:0]  a;  
logic signed[3:0]  b;  
logic  [4:0]  c;
```

```
a = 8; // 4'b1000
```

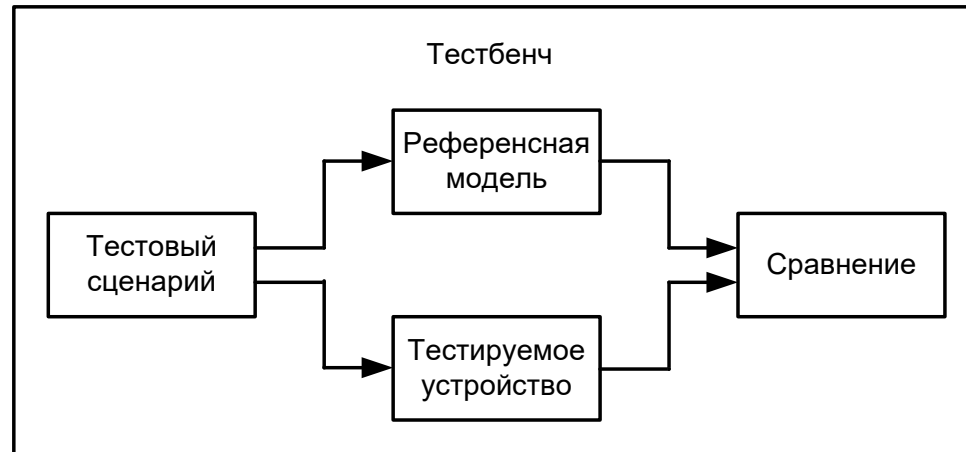
```
b = 1; // 4'b0001
```

```
c = a + b; // 5'b11001 // -7
```

```
c = $signed(a) + $signed(b); // 5'b11001 // -7
```

Тестбенч

- Модуль, задающий тестовое окружение для тестируемого модуля т.е. отвечает за генерацию входных сигналов для тестируемого модуля, захват его выходных данных и сравнение с ожидаемыми данными
- Используются только для моделирования, поэтому можно использовать несинтезируемое подмножество SV



Директива timescale

- 'timescale <N1> / <N2>
- Определяет единицу времени и точность для следующих за ней модулей
- N1 – значение времени моделирования и задержек
- N2 – степень точности N1, т.е. как округляются значения задержки перед использованием в моделировании
- N1 и N2 могут указываться в ns или ps и принимать значения 1, 10, 100

Симуляция

- Цель симуляции – вычислить значения всех сигналов на определенном интервале времени
- Вычисления осуществляются в дискретные моменты времени изменения сигналов
- Процессы (процедурные блоки `always`, `initial`, `assign` и т.п.) работают конкурентно (параллельно)
- Если исполнение процесса началось, оно не прервется, пока симулятор не дойдет до конца, либо до оператора ожидания
- Операторы ожидания: `@`, `#`, `wait()`
- Внутри процедурных блоков инструкции выполняются сверху вниз
- Если внутри блока есть оператор ожидания, его выполнение приостанавливается до момента выполнения условия, указанного для оператора
- После приостановки выполнение передается другому процессу

Процедурные блоки `always` и `initial`

- Блок `always` выполняется на каждом временном шаге или по событию
- Для выполнения по событию указывается `@(sensitivity_list)`

```
localparam int unsigned CLK_PERIOD = 4;  
logic clk = 0;  
always #(CLK_PERIOD / 2) clk = ~clk;
```

Процедурные блоки `always` и `initial`

- Блок `initial` выполняется один раз в начале моделирования
- Тестбенч может содержать несколько блоков

```
initial localparam int unsigned CLK_PERIOD = 4;
        logic clk;
        initial begin
            clk = 0;
            forever begin
                #(CLK_PERIOD / 2);
                clk = ~clk;
            end
        end
end
```

Задачи (task) и функции (function)

- Это подпрограммы, которые можно использовать многократно
- Переменные, объявленные внутри task (function), имеют область видимости задачи (функции), внутри которой они объявлены
- Могут быть объявлены с ключевым словом automatic

Функции

- Выполняются немедленно и не могут содержать операторы управления временем (`#`, `@`, `wait`)
- Можно использовать только блокирующие присваивания
- Внутри функции нельзя вызвать `task`
- Аргументы могут иметь “направления” `input`, `output`, `inout` и `ref`
- Может возвращать только одно значение “классическим” способом и несколько значений через `output` аргументы

ФУНКЦИИ

```
// First function declaration style - inline arguments  
function <return_type> <name> (input <arguments>);  
    // Declaration of local variables  
    begin  
        //function code  
    end  
endfunction
```

```
// Second function declaration style - arguments in body  
function <return_type> <name>;  
    input <arguments>;  
    // Declaration of local variables  
    begin  
        //function code  
    end  
endfunction
```

Задачи

- Могут содержать операторы управления временем
- Можно использовать как блокирующее, так и неблокирующее присваивание
- Может возвращать любое количество данных через output аргументы (не имеет возвращаемого значения в классическом понимании)
- Изнутри задачи можно вызвать другую задачу или функцию

Задачи

```
// First task declaration style - inline arguments  
task <name> (<arguments>);  
    begin  
        // code which implements the task  
    end  
endtask
```

```
// Second task declaration style - arguments in body  
task <name>;  
    <arguments>;  
    begin  
        // code which implements the task  
    end  
endtask
```

Литература

- Дэвид М. Хэррис, Сара Л. Хэррис:
“Цифровая схемотехника и архитектура
компьютера”
- IEEE Std 1800-2017 Standard for
SystemVerilog - Unified Hardware Design,
Specification, and Verification Language