Симуляция

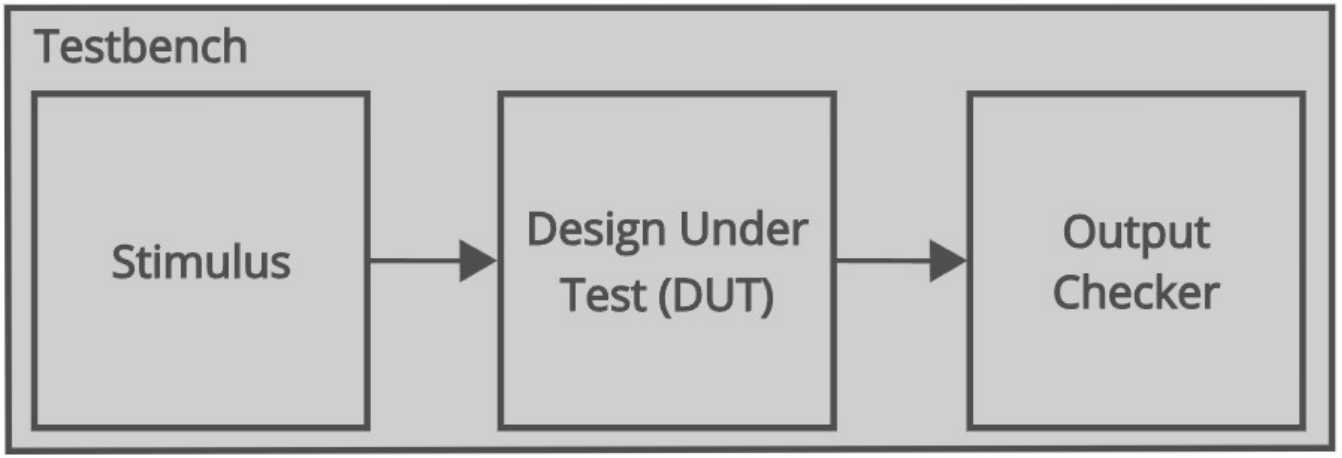
Testbench

```
`timescale 1ns/100ps

module simple_tb();

    reg clk = 1'b0;
    reg [7: 0] in1=8'b0, in2=8'b0;
    wire [7: 0] out1, out2;

    initial forever #1 clk = ~clk;

    initial begin
        @(posedge clk);
        in1 = 8'd10; in2 = 8'd13;
        @(posedge clk);
        in1 = 8'd10; in2 = 8'd13;
        ....
    end

    my_design DUT (.clk(clk), .in1(in1), .in2(in2),
            .out1(out1), .out2(out2));

    initial begin
        $monitor(out1, out2);
        @(posedge clk);
        $monitor(out1, out2);
        @(posedge clk);
        $monitor(out1, out2);
        ...
    end

endmodule
```

Stimulus

DUT

Output checker



Все кроме самого модуля DUT не обязано быть синтезируемым. Поэтому для тестирования мы можем использовать несинтезируемые конструкции.

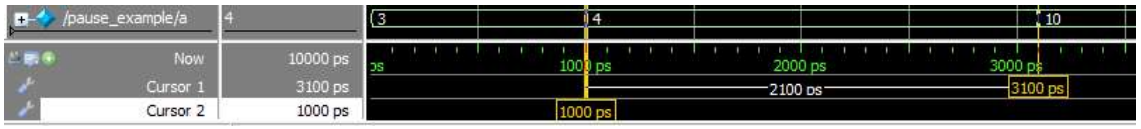
Время и ожидание

```
`timescale 1ns/100ps
```

Размер единицы времени

Точность (происходит округление)

```
int a;
initial begin
    a = 3;
    #1 a = 4;
    #2.1
    a = 10;
end
```



Пауза:

#10.1 – пауза на 10.1 единиц времени

#a – может быть использован с переменной

Ожидание сигнала:

@(posedge clk); - возвращает управление сразу после фронта clk.

```
int a;
initial begin
    a = 3;
    @(posedge clk);
    a = 5;
    @(posedge clk);
    a = 10;
end
```

Initial block

(Procedural blocks)

- Выполнение начинается сразу же после начала симуляции
- Последовательно выполняет инструкции в блоке и завершает его.

```
initial begin
    @(posedge clk);
    in1 = 8'd10; in2 = 8'd13;
    @(posedge clk);
    in1 = 8'd10; in2 = 8'd13;
    ...
end
```

- Так же можно вписать в синтезируемый модуль.
  - В таком случае блок initial будет запущен при симуляции, но никак не повлияет на синтезируемую схему.
  - Иногда может быть удобно для симуляции если требуется установить схему в известное состояние.

```
module clk_div(
    input clk,
    output clk2
);

    reg [11:0] cnt;
    initial cnt = 12'b0;

    assign clk2 = cnt[11];

    always @(posedge clk) begin
        cnt <= cnt + 12'b1;
    end

endmodule
```

Initial block

Внутри блока initial можно использовать операторы ветвления

```
initial begin
    int a;
    if (a==10) begin
        ...
    end
end
```

и циклов (операторы break и continue так же доступны)

```
initial begin
    int a;
    for (a=0; a<10; a=a+1) begin
        // do something
    end

    while (a!= 0) begin
        // do something
    end
end
```

Так же сущевствует оператор бесконечного цикла, которым удобно реализовывать симуляцию clk:

```
reg clk = 1'b0;
initial begin
    forever begin
        #1 clk = ~clk;
    end
end
```

В случае если требуется исполнить всего одну команду, то begin и end можно опустить:

```
initial forever #1 clk = ~clk;
```

Неинициализированные регистры в симуляции

clk_div модуль

```
module clk_div(
    input clk,
    output clk2
);

    reg [11:0] cnt;
    assign clk2 = cnt[11];


    always @(posedge clk) begin
        cnt <= cnt + 12'b1;
    end

endmodule
```

Testbench

```
`timescale 1ns/1ns


module tb_clk_div();

    reg clk = 1'b0;
    initial forever #1 clk=~clk;

    wire clk2;
    clk_div DUT(.clk(clk), .clk2(clk2));

endmodule
```

Т.к. Регистр cnt не был инициализирован, то его состояние X.

Инициализация регистров

Инициализация внутри синтезируемого модуля.

(Для FPGA Intel данный код даже сгенерирует схему асинхронного сброса)
(Для ASIC данный код будет использован только для симуляции если требуется асинхронный сброс, то придется вручную его вписать)

```
module clk_div(
    input clk,
    output clk2
);

    reg [11:0] cnt = 12'b0;
    assign clk2 = cnt[11];


    always @(posedge clk) begin
        cnt <= cnt + 12'b1;
    end

endmodule
```

```
module clk_div(
    input clk,
    output clk2
);

    reg [11:0] cnt;
    initial cnt = 12'b0;
    assign clk2 = cnt[11];


    always @(posedge clk) begin
        cnt <= cnt + 12'b1;
    end

endmodule
```

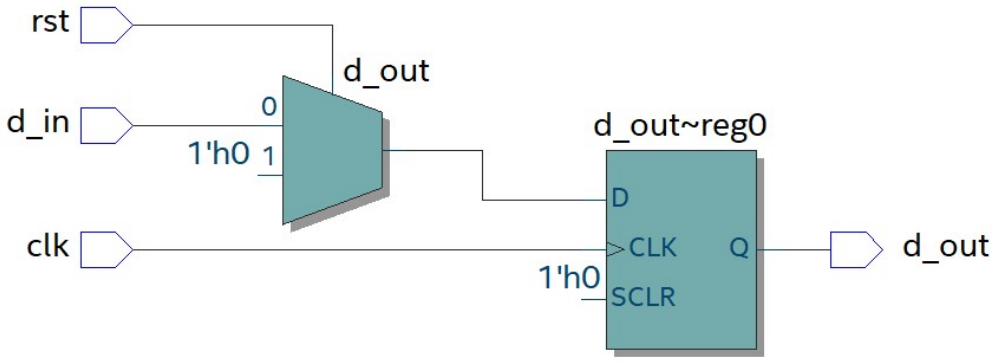
Инициализация внутри testbench.

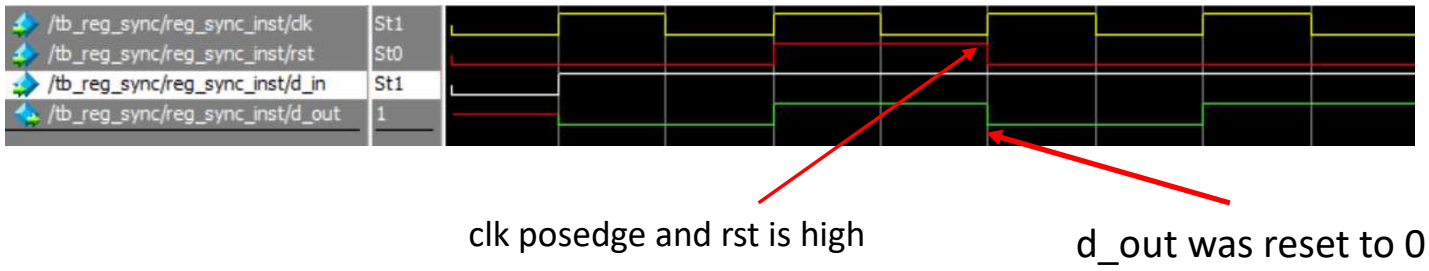
```
`timescale 1ns/1ns


module tb_clk_div();

    reg clk = 1'b0;
    initial forever #1 clk=~clk;

    wire clk2;
    clk_div DUT(.clk(clk), .clk2(clk2));
    initial DUT.cnt = 12'b0;
endmodule
```

Синхронный сброс.

По фронту clk
если сигнал rst активен
происходит сброс в заданное значение.

```
module reg_sync #(
    RST_VAL=1'b0
)(
    input clk,
    input rst,
    input d_in,
    output reg d_out
);

    always @(posedge clk) begin
        if (rst) begin
            d_out <= RST_VAL;
        end
        else begin
            d_out <= d_in;
        end
    end

endmodule
```





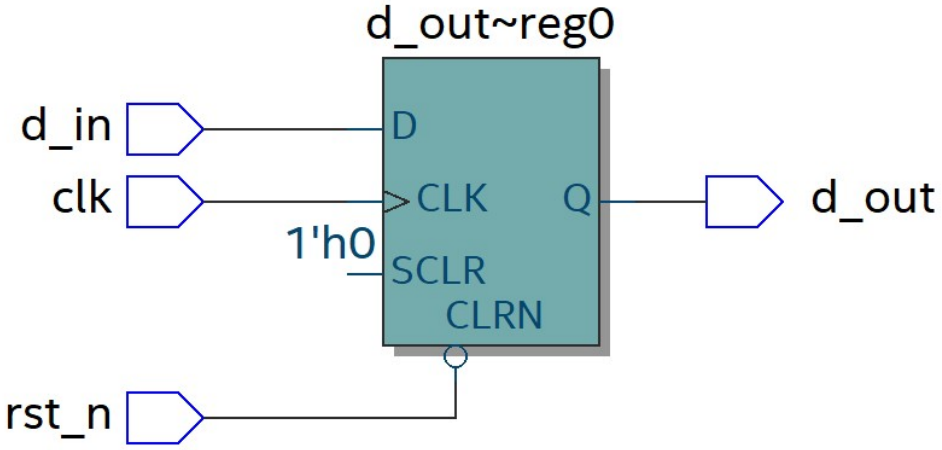
clk posedge and rst is high

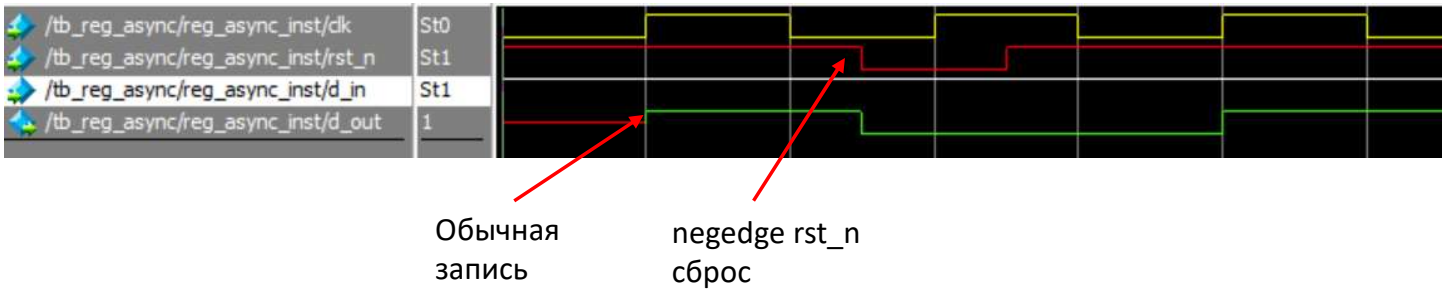
d_out was reset to 0.

Асинхронный сброс

Reset происходит по фронту самого сигнала rst_n.
Сброс происходит независимо от clk.
Чаще всего используется для инициализации при запуске устроства POR (Power On Reset).

```
module reg_async #(
    RST_VAL=1'b0
)(
    input clk,
    input rst_n,
    input d_in,
    output reg d_out
);

    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            d_out <= RST_VAL;
        end
        else begin
            d_out <= d_in;
        end
    end

endmodule
```





Функции

Для повторяющихся вычислений в Verilog так же можно написать функции.
Есть два разных стиля обьявления функций.

```
function integer sum (input integer in_a, in_b);
  begin
    sum = in_a + in_b;
  end
endfunction
```

```
function integer sum;
  input integer in_a;
  input integer in_b;
  begin
    // Return the sum of the two inputs
    sum = in_a + in_b;
  end
endfunction
```

В Verilog есть некоторое количество встроенных математических функций:
$exp, $ceil, $sin, $log10 …..

Функции предназначены только для вычислений и не могут содержать операторы задержки или ожидания такие как #1 и @(posedge clk)!!

Task

В отличие от функций, task может ‘возвращать’ несколько значений и использовать конструкции затрачивающие время симуляции (#1 и @(posedge clk)). Но Task может быть использован только для симуляции.

```
// style 1
task sum_on_clk(input clk, input [3: 0] a, b, output [3: 0] c);
begin
   @(posedge clk);
   c = a+b;
end
endtask

// style 2
task sum_on_clk;
    input [3: 0] a, b;
    output [3: 0] c;
    begin
        @(posedge clk);
        c = a+b;
    end
endtask
```

```
// call
initial begin
    reg clk;
    reg [3: 0] x, y, z;
    sum_on_clk(clk, x, y, z);
end
```

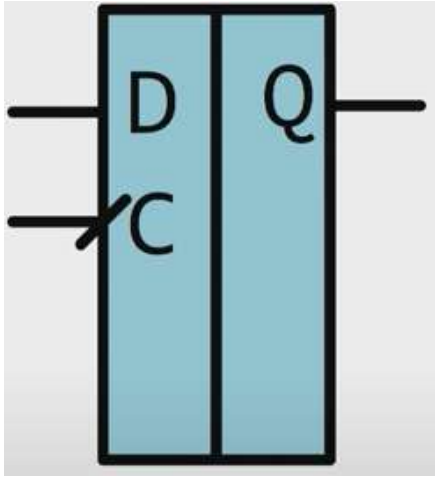
Sytem Tasks

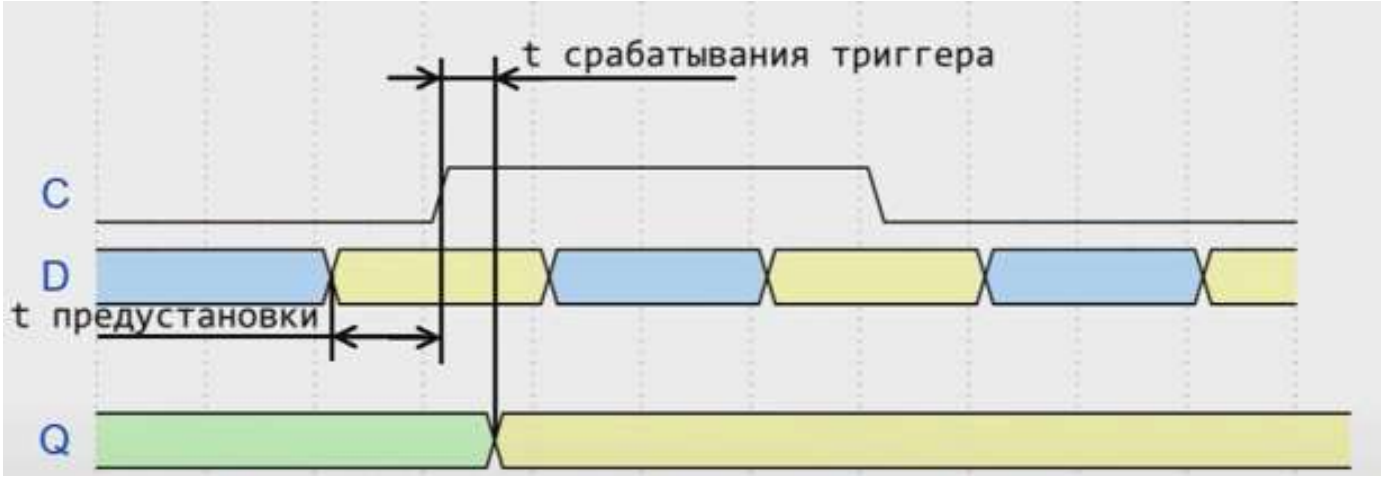
Полный список если потребуется всегда можно посмотреть в документе стандарта (Например Verilog 2005). Там же можно посмотреть их аргументы (т.к. Порой найти информацию по непопулярным таскам бывает проблематично)

- Display tasks
  - $display
  - $monitor
- File IO
  - $fopen
  - $fread
  - $fwrite
- Simulation Control
  - $stop
  - $finish
- Simulation time
  - $time
  - $realtime
- File IO
  - $fopen
  - $fread
  - $fwrite
- Probabilistic distribution
  - $random
  - $dist_normal

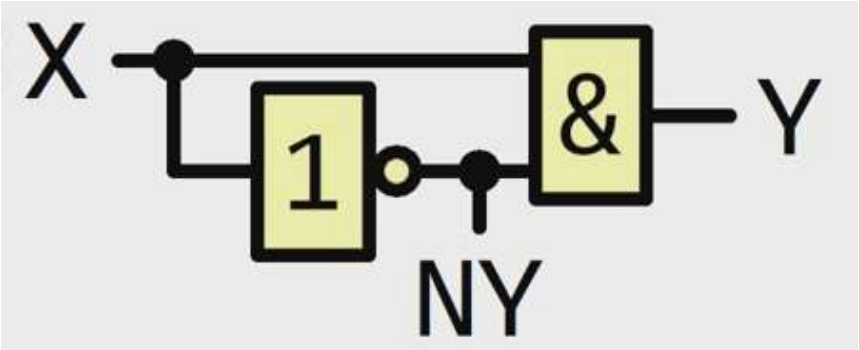
Основы синхронной логики

Особенности работы D триггера



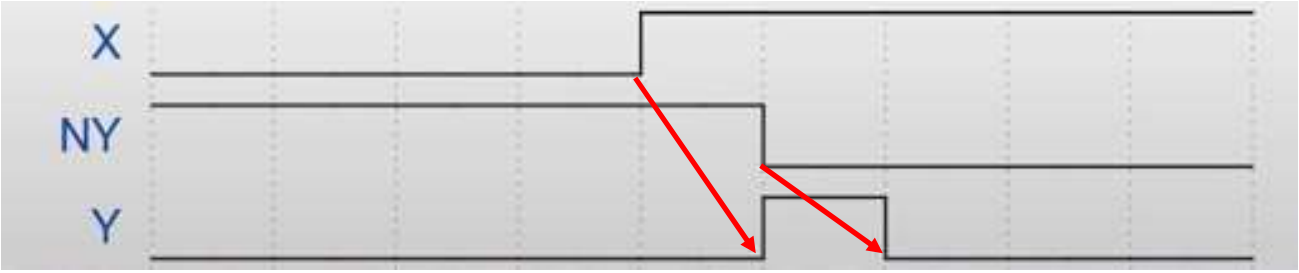


Комбинационная логика
(Hazard)

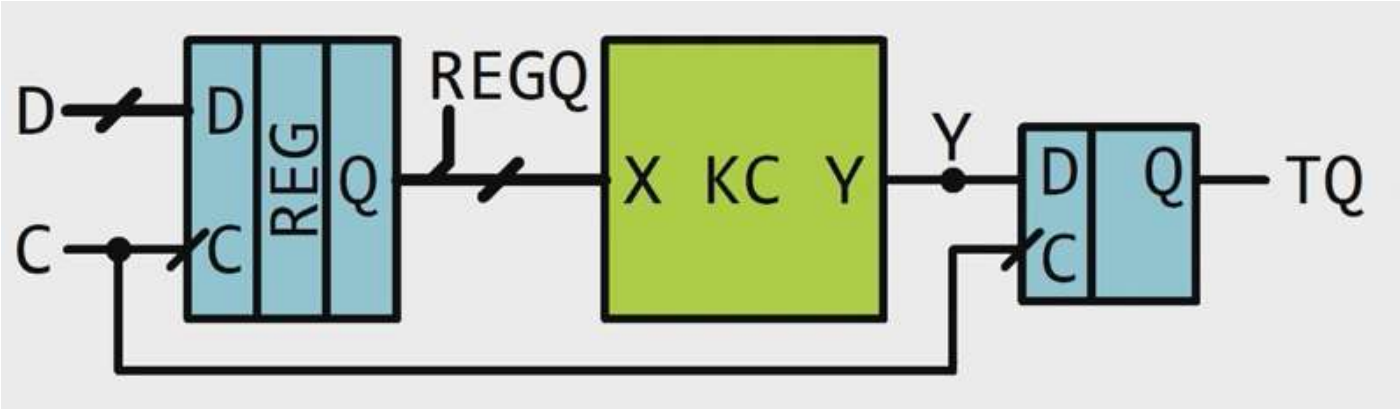


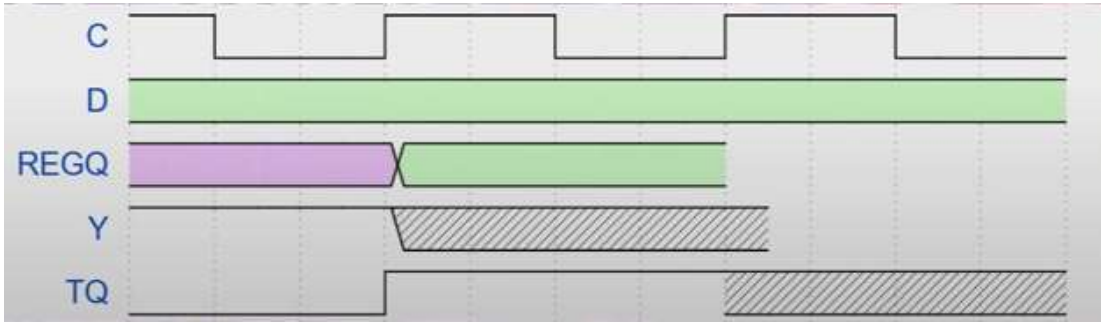
Из за того что задержка распространения сигнала по различным путям разная, то выход комбинационной схемы некоторое время может переключаться.

Время дребезга определяется длиннейшим путем.

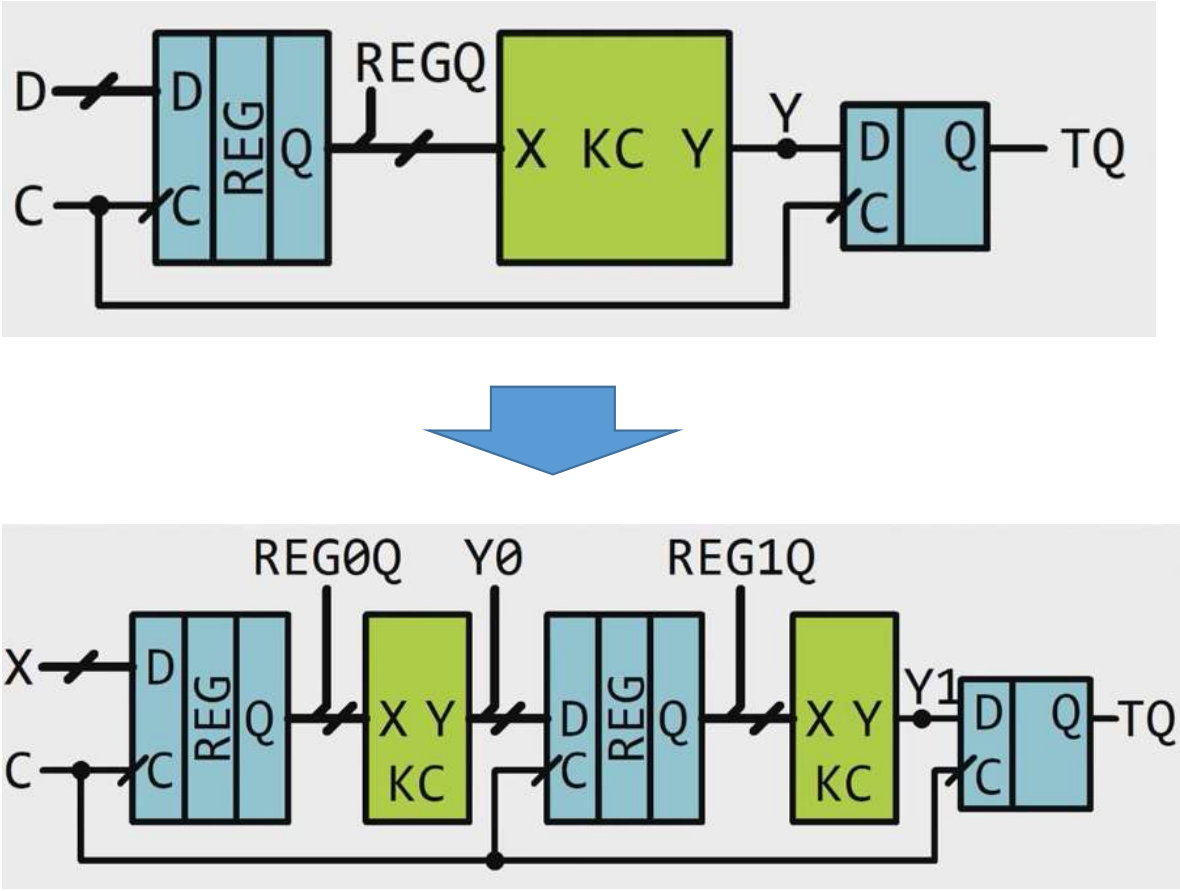


Комбинационная логика
(Hazard)

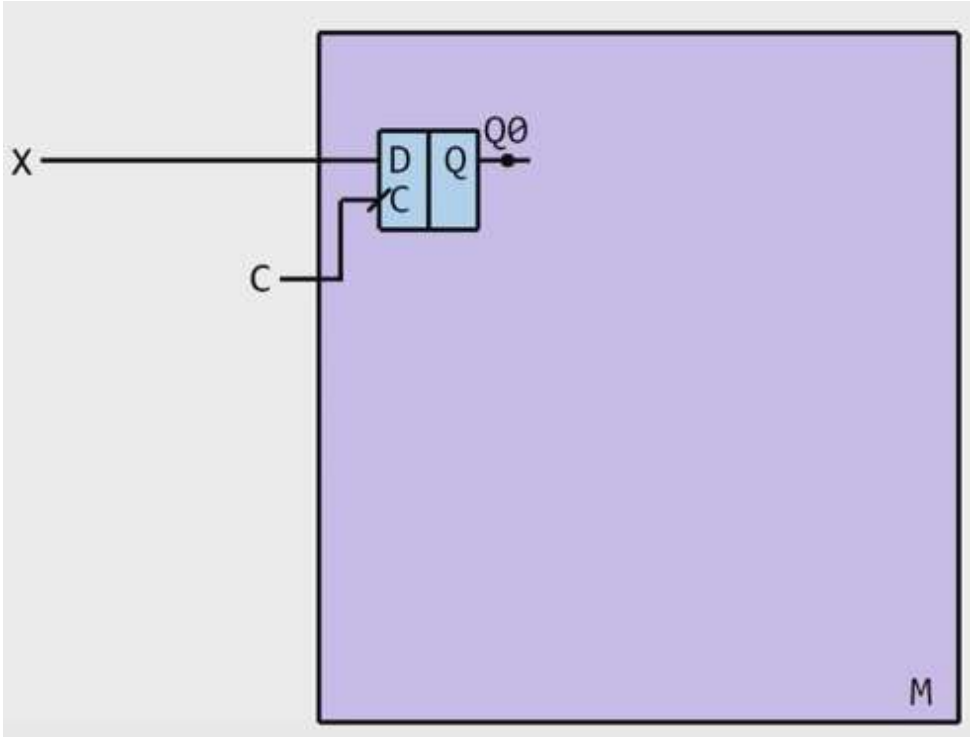


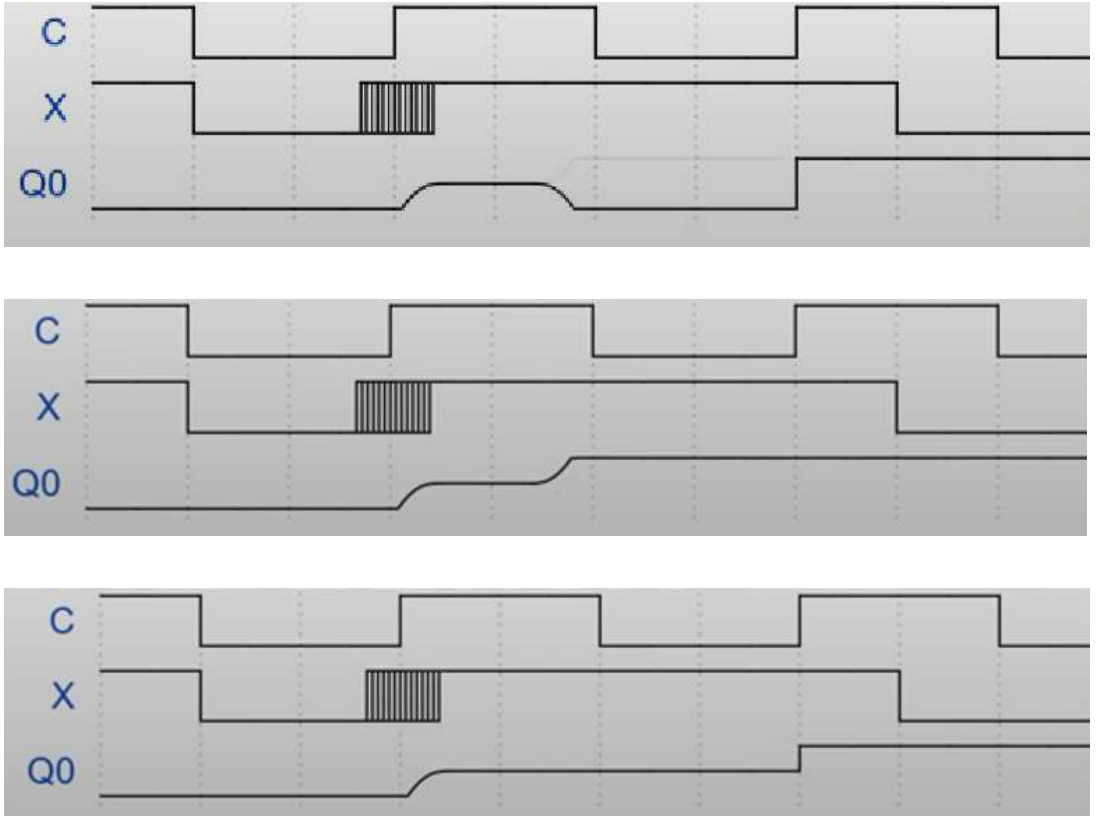


Способ решения проблемы



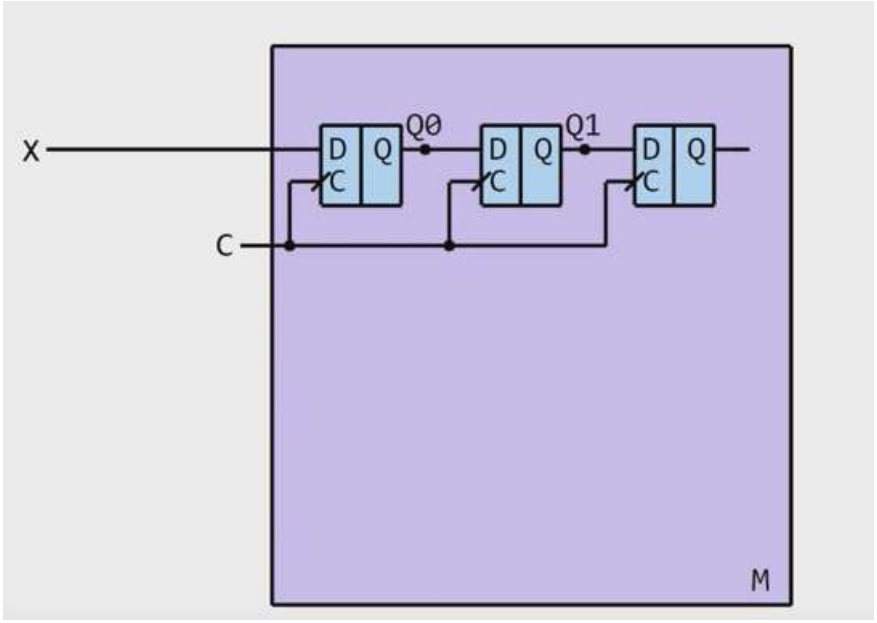
Синхронизация внешних сигналов
(Метастабильность)

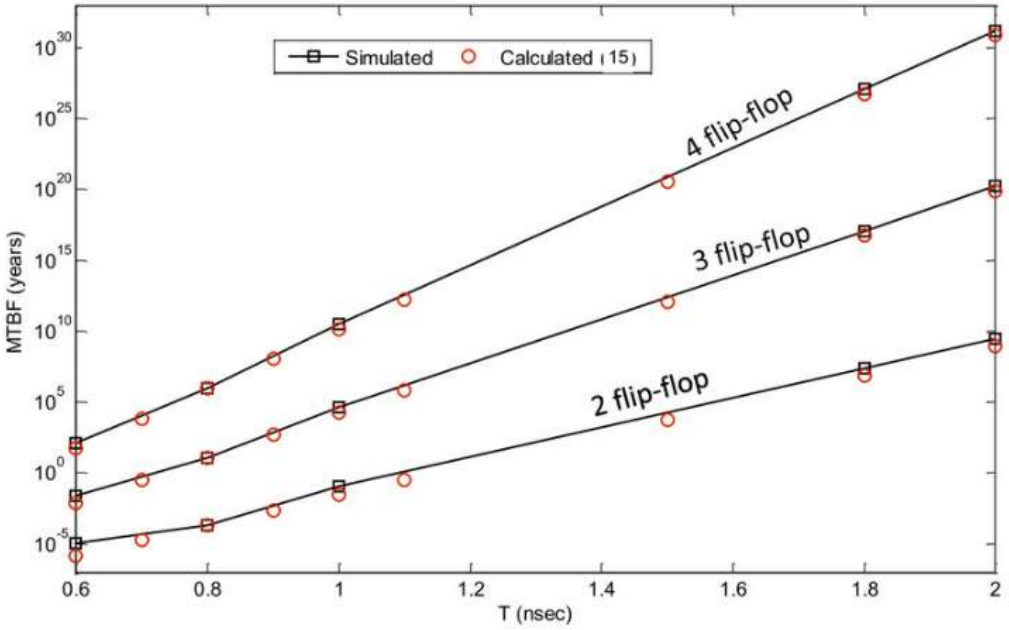




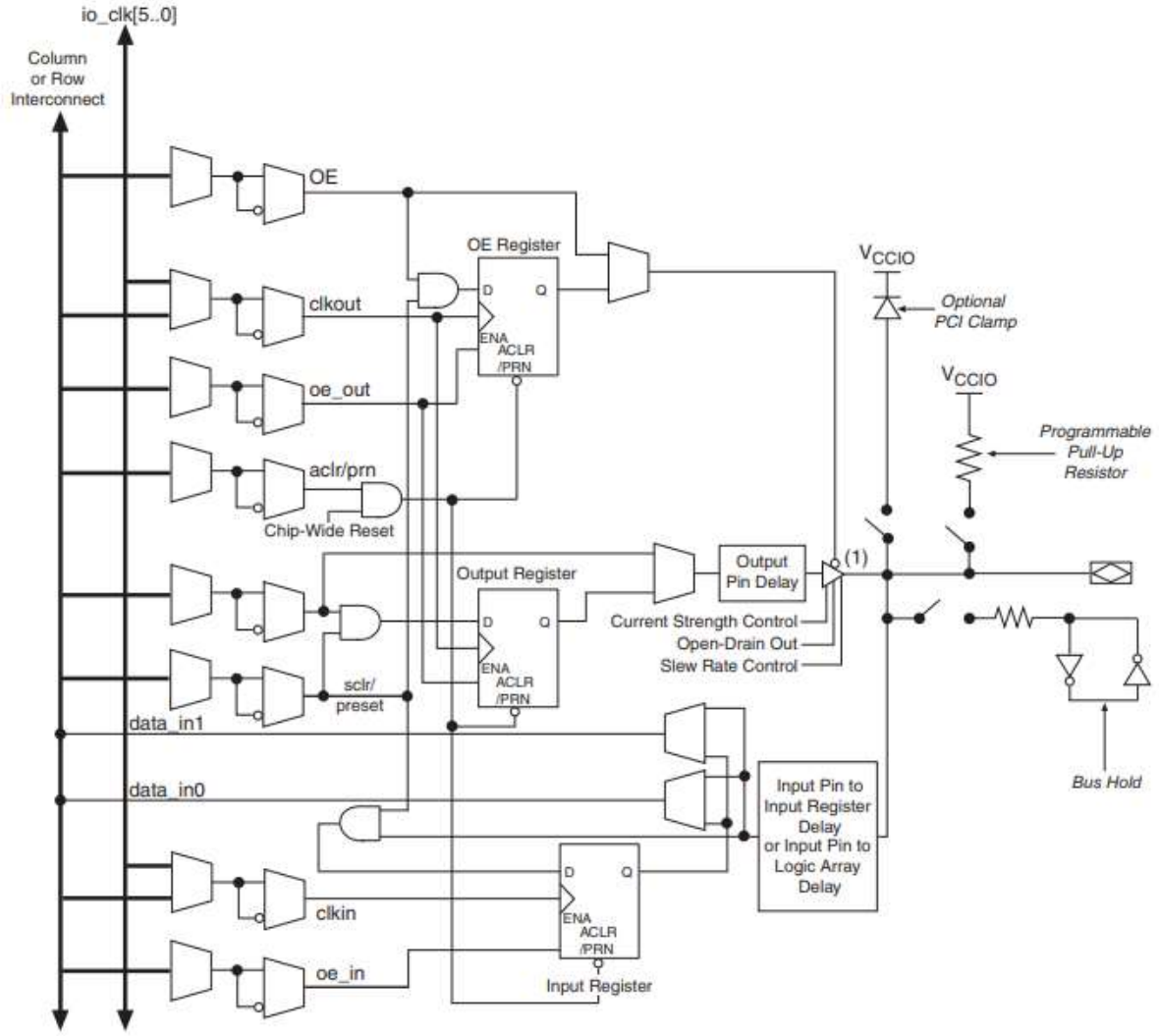
Решение проблемы метастабильности

Для уменьшения вероятности попадания в метастабильное состояние можно увеличить цепочку триггеров





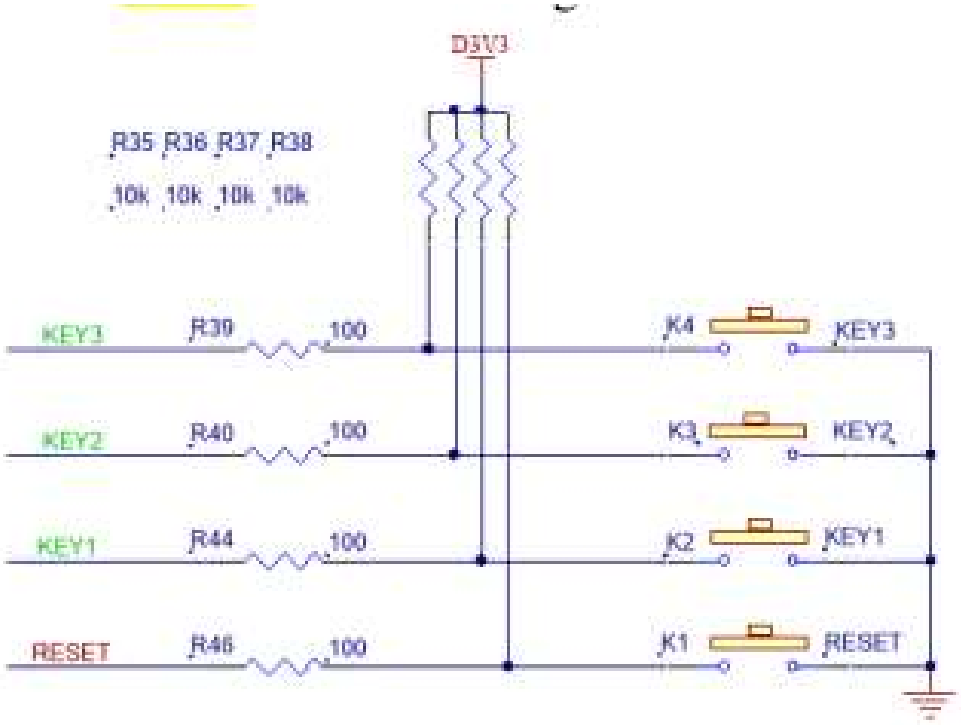
IO порты


io_clk[5..0]
Column or Row Interconnect
OE
OE Register
clkout
oe_out
aclr/prn
Chip-Wide Reset
Output Register
sclr/ preset
data_in1
data_in0
clkin
oe_in
Input Register
D
Q
ENA
ACLR /PRN
Output Pin Delay
Current Strength Control
Open-Drain Out
Slew Rate Control
(1)
VCCIO
Optional PCI Clamp
Programmable Pull-Up Resistor
Bus Hold
Input Pin to Input Register Delay or Input Pin to Logic Array Delay

Кнопка подклюлчение

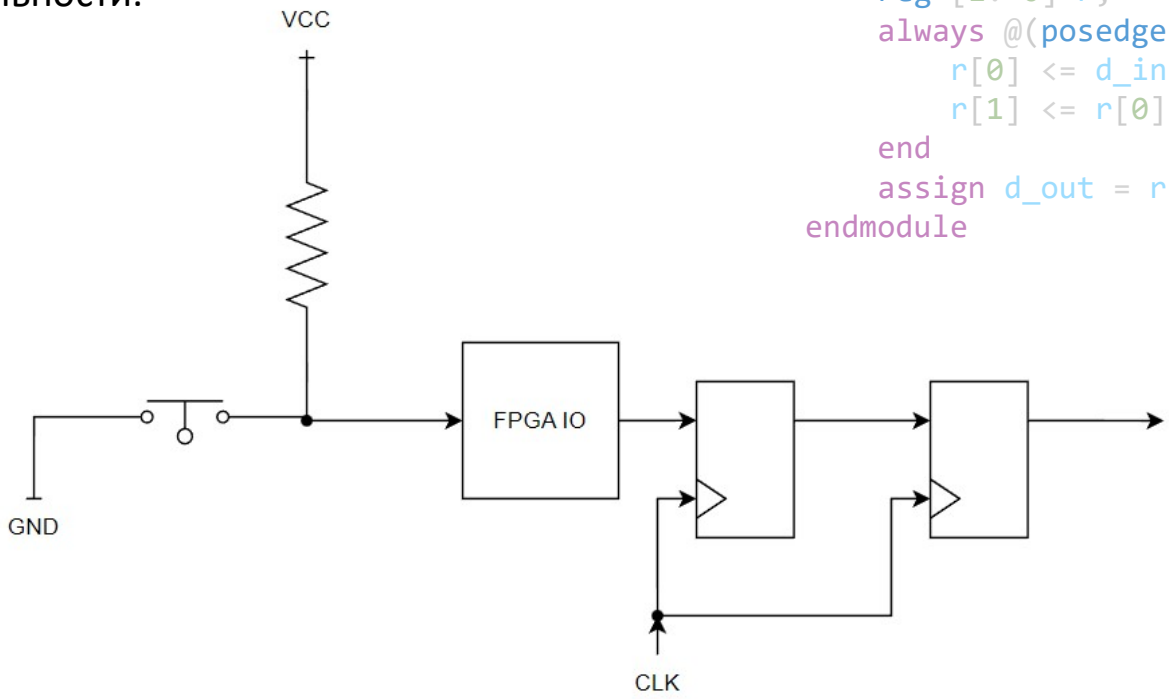
В отпущенном состоянии – 1.
В нажатом – 0.

Возможно подключение наоборот через подтягивающий к земле резистор.



Кнопка асинхронный вход.

Сигнал с кнопки асинхронный, поэтому его следует в первую очередь синхронизировать чтобы избежать метастабильности.

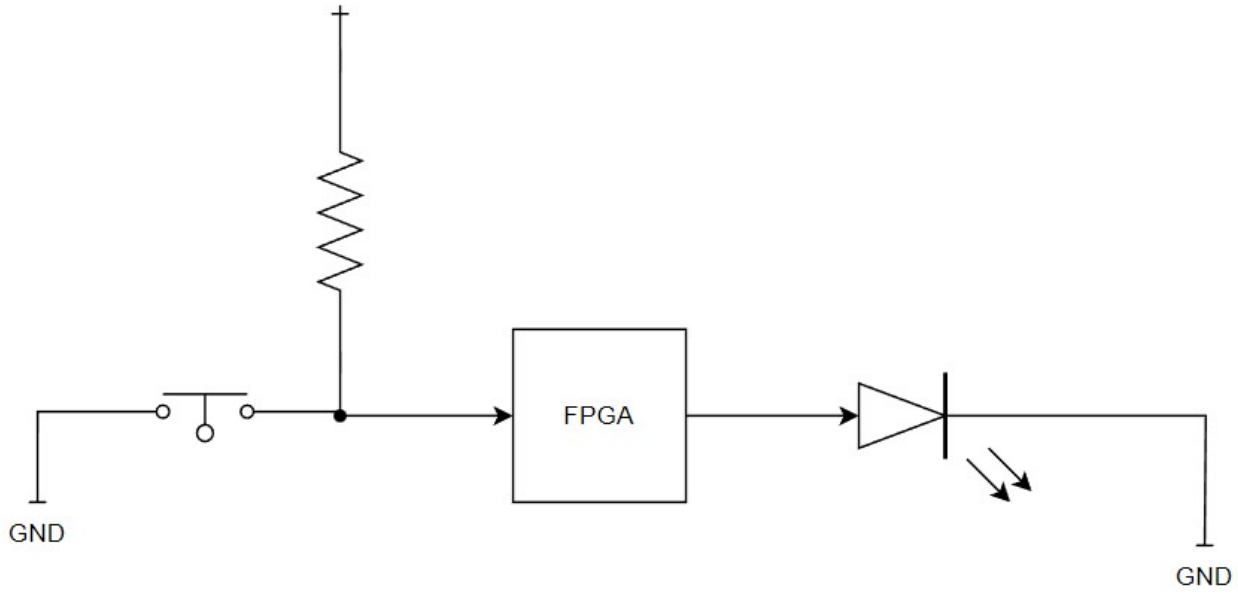



```
module sync(
    input clk,
    input d_in,
    output d_out
);
    reg [1: 0] r;
    always @(posedge clk) begin
        r[0] <= d_in;
        r[1] <= r[0];
    end
    assign d_out = r[1];
endmodule
```

Простейшее устройство с кнопкой

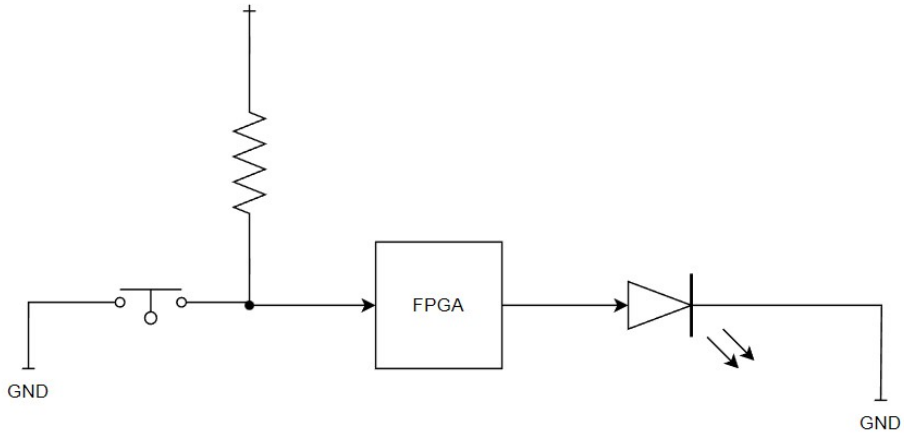
```
module button_connect(
    input clk,
    input btn,
    output led
);

    wire btn_sync;
    sync sync_inst(
        .clk(clk),
        .d_in(btn),
        .d_out(btn_sync));

    assign led = btn_sync;


endmodule
```

При нажатии кнопки диод загорается.



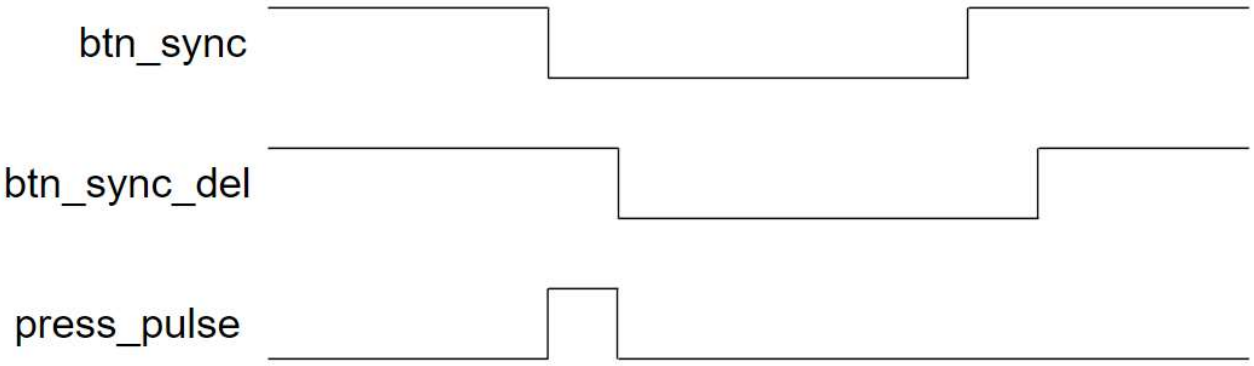
Детектирование нажатия

Что если мы хотим менять состояние кнопки нажатием.
Первое нажатие – диод горит.
Второе нажатие – диод гаснет.



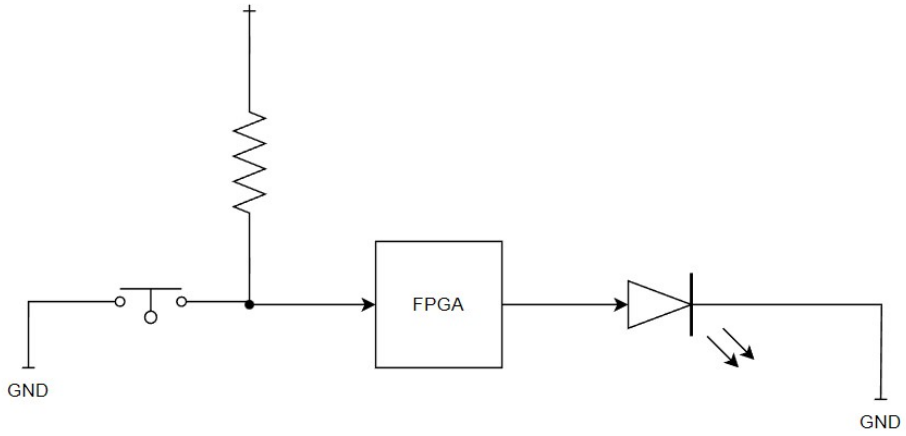
Нам потребуется как то определить событие нажатия

```
reg btn_sync_del;
always @(posedge clk) btn_sync_del <= btn_sync;

wire press_pulse = (btn_sync_del)&(~btn_sync);
```



Переключение диода по нажатию

```
module button_state(input clk, btn, output led);

    wire btn_sync;
    sync sync_inst(.clk(clk), .d_in(btn), .d_out(btn_sync));

    reg btn_sync_del;
    always @(posedge clk) btn_sync_del <= btn_sync;
    wire press_pulse = (btn_sync_del)&(~btn_sync);

    reg state = 1'b0;
    always @(posedge clk) begin
        if (press_pulse) state <= ~state;
    end

    assign led = state;

endmodule
```



Дребезг контактов

Кнопка замыкается не единомоментно а сдребезгом. Поэтому вместо одного нажатия может быть сгенерировано множественное нажатие. Есть разные способы борьбы с дребезгом.
Простейший способ - подождать ~10 мсек после первого срабатывания.