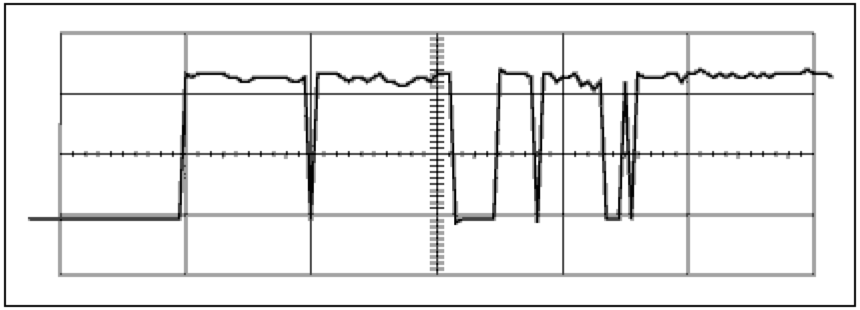
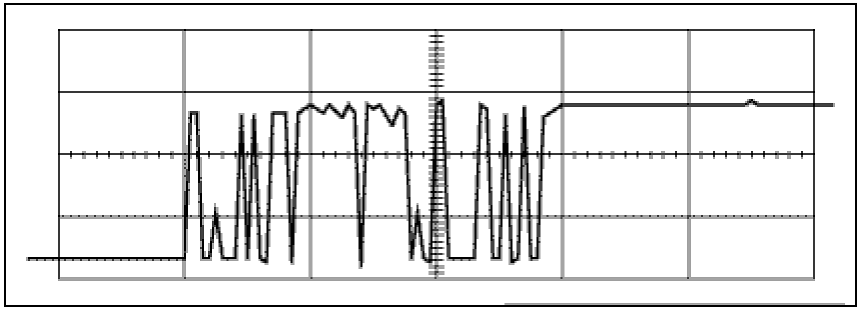
INTERFACING WITH INPUT SIGNALS

I) Mechanical Switch Bouncing

- Most mechanical switches "bounce" rapidly while transitioning between an open and closed state
- These bounces can produce:
    1) runt pulses that do not reach a full "0" or "1" state, and/or
    2) rapid bouncing between "0" and "1" states



A rising-edge switch bounce for a small pushbutton switch with an approximate 5 ms bounce interval and 10 transitions



A rising-edge switch bounce for a 5A contact relay with an approximate 5.5 ms bounce interval and 20 full-amplitude transitions

[https://www.maximintegrated.com/en/app-notes/index.mvp/id/287]

I) Input De-Bouncing

- De-bouncing solutions are typically best designed with circuits such as:
  1. A low pass filter such as a resistor-capacitor (RC) filter
     - The RC product can not be too small (allows bounces through) or too large (long rise/fall time and slow response)
  2. A double-throw switch with an attached SR latch
     - An excellent solution if the switch type and latch are available
  3. Some type of sampling or gating function that is tailored to the bouncing characteristics
- The DE-10 Lite board contains debouncing circuits for all SW switches and KEY buttons

II) Edge Detection

- In many problems, a circuit needs to take an action on only a particular edge of a signal—for example, on only the rising edge of a signal
- For example, if a signal will be asserted many cycles but it is desired to count the event only once, the rising or falling *edge* can be used to trigger the event rather than the *level* of the signal
- Example: the key of a keyboard may be sampled at a very high rate but only one character should be processed each time a key is pressed
- Example: "mouse up" event

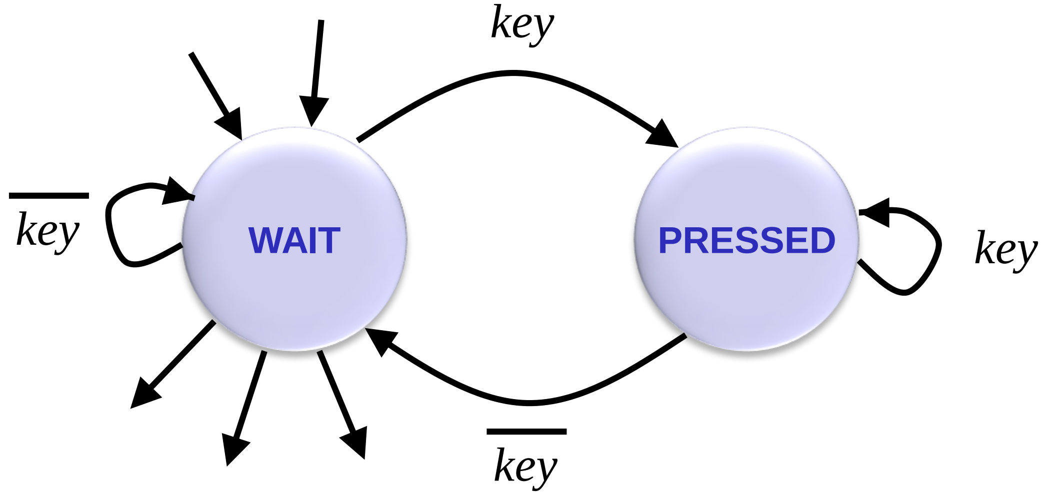
II) Edge Detection Method 1

- Edge detection solutions are probably best implemented with digital logic

1. Extra states can be added to the state machine which processes the input signal (or a dedicated state machine can be made)
   – The general idea is to transition to a dedicated state on the first edge of the input (e.g., rising edge), stay there while the input is at that level, and then return to the original state on the second edge of the input (e.g., falling edge)

II) Edge Detection Method 1

- For example, design a keyboard controller that increments a counter only once when a key is pressed even though it may be pressed for 1000s of clock cycles
- *key* = 1, key is pressed
- *key* = 0, key is not pressed



II) Edge Detection Method 1

- In this example code, **`count`** is incremented on exactly the same cycle as when **`state`** changes to the PRESSED state
- Of course this is actually done by setting **`count_c`** and **`state_c`** the previous cycle

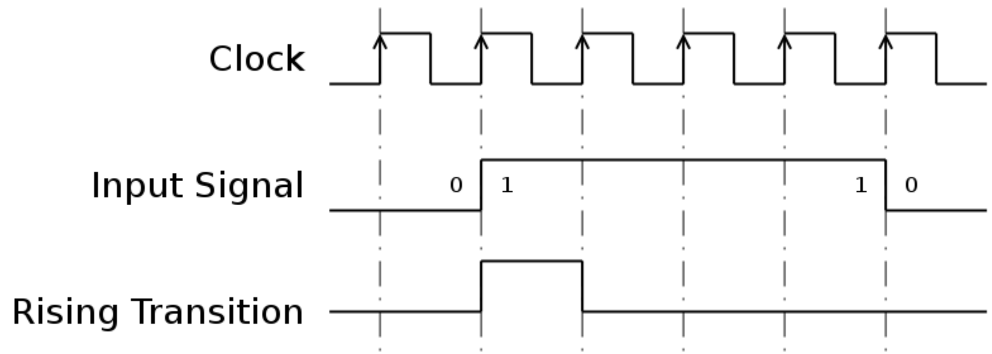
```
always @(*) begin
  // defaults
  state_c = state;
  count_c = count;   // default do not add

  case(state)
    WAIT: begin
      if (key == 1'b1) begin
        state_c = PRESSED;
        count_c = count + 8'h01;   // Add +1 here!
      end
    end

    PRESSED: begin
      // Do nothing special when key==1
      if (key == 1'b0) begin
         state_c = WAIT;
      end
    end
  endcase
end   // always
```

II) Edge Detection Method 2

2. A circuit can look at both the current and previous value of a signal and output a single-cycle pulse on the desired edge(s). Designs can be made with either Mealy or Moore style outputs as shown on later slides.

Detecting Signal Transitions

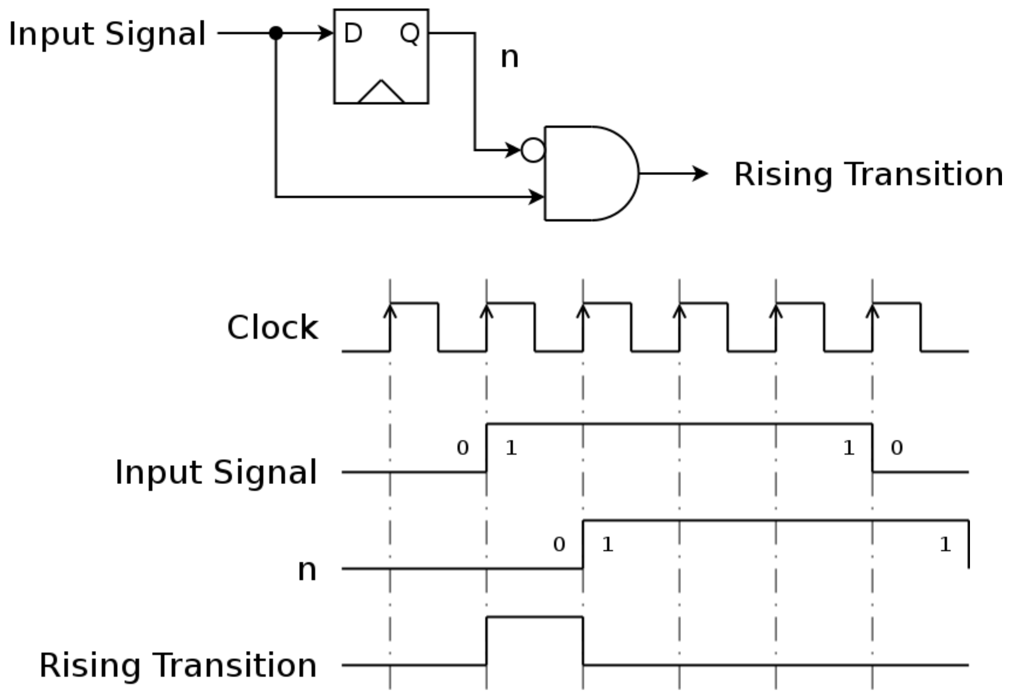
- The goal is to design a state machine/circuit that is sensitive to only a *change* in an input signal (e.g., change from 0 to 1)
- It can be awkward to design an FSM for signal transitions



- Despite being highly tempting, we can not use the signal itself as the clock of an edge-triggered flip-flop—this would lead to poor timing and unreliable circuits. This would also break a fundamental rule discussed in the Clock section
- The key idea is to look for the point in time when the value from the previous clock cycle is a 0 and the value from the current cycle is a 1
    - This implies we need to save the old value (in a flip-flop)

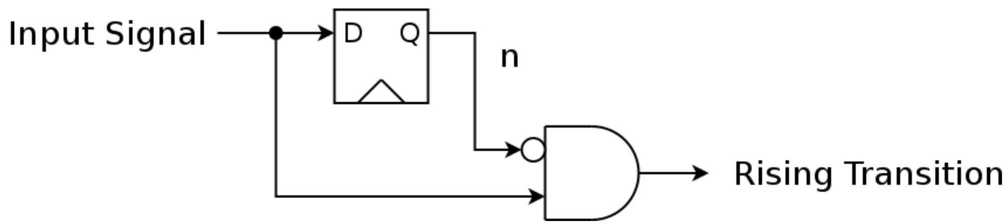
[M. Hildebrand]

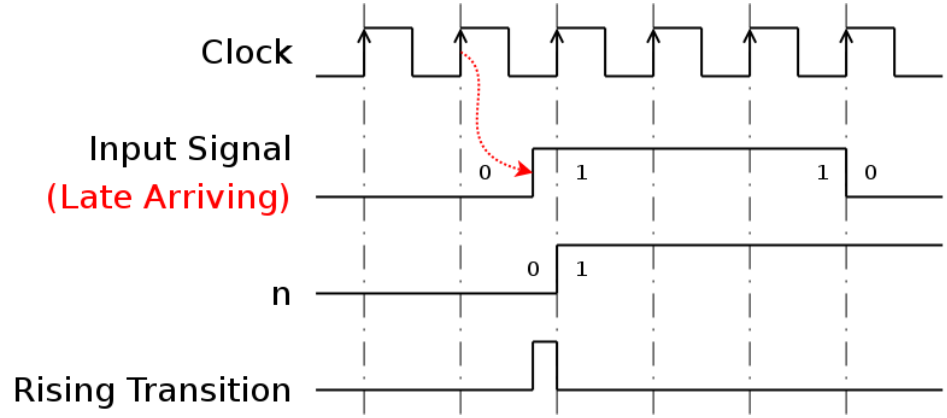
Edge Detection Circuit Solution 1: Mealy, Early Input Arrival



[M. Hildebrand]

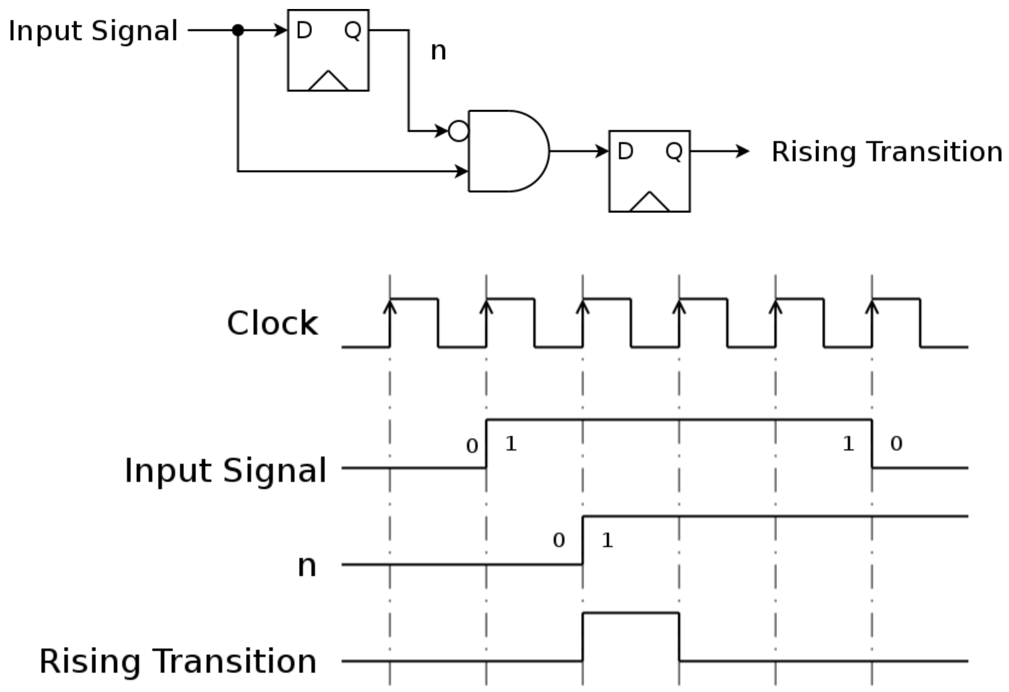
Edge Detection Circuit Solution 1: Mealy, Late Input Arrival





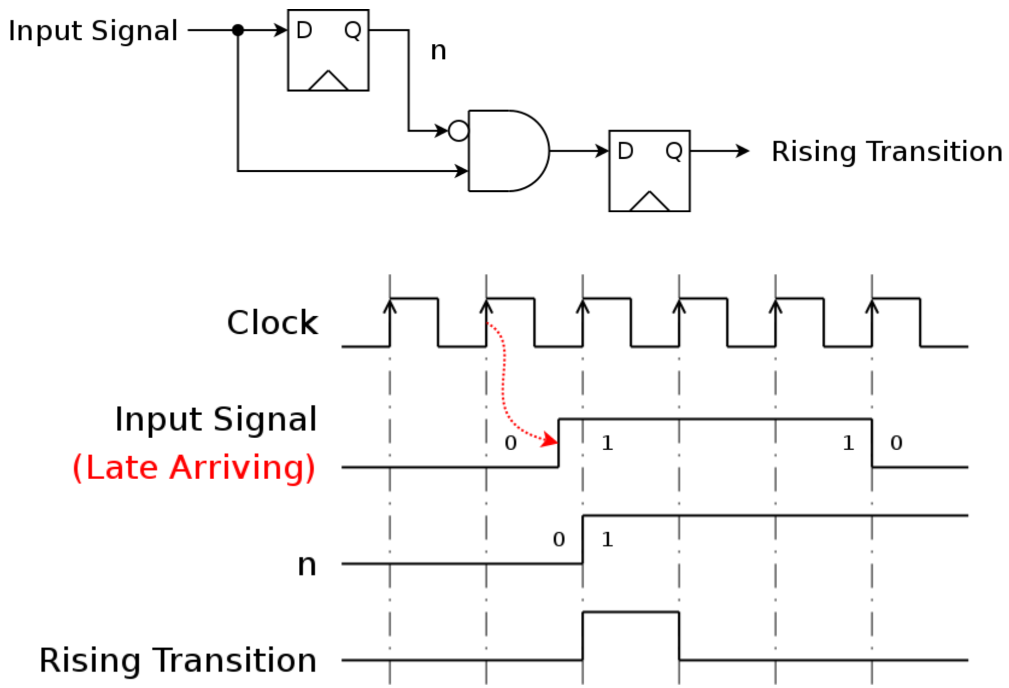
[M. Hildebrand]

If needed, Rising Transition can be registered (Moore)



[M. Hildebrand]

Circuit operates the same with late arriving inputs (Moore)



[M. Hildebrand]

Example Verilog module for edge detection with registered output (Moore)

```
module edge_detection (
   input  clock,
   input  input_signal,
   output rising_transition
   );

// declarations
reg  n;
reg  rising_transition;
wire rising_transition_c;

// logic to detect 0 in previous cycle and 1 in current cycle
assign rising_transition_c = ~n & input_signal;

// flip-flop instantiations
always @(posedge clock) begin
    n                 <= #1 input_signal;
    rising_transition <= #1 rising_transition_c;
end

endmodule
```


[M. Hildebrand]

Experimenting with SW and KEY Inputs on the DE10-Lite Board

- An experiment was performed on the DE10-Lite board counting the number of 1) edges, and 2) cycles the signal is high for both KEY buttons and SW switches using a 50 MHz clock

	KEY Button	**SW Switch**
Level count	8,432,414 (0.17 seconds)	20,753,489 (0.42 seconds)
Edge count	**1**	**1**

[Tim Andreas, 2018/05/18]

Verilog Code

```
module edge_detection (
   input clock,
   input input_signal,
   output reg rising_transition
   );

reg  n;
wire rising_transition_c;

// logic to detect 0 in previous cycle and 1 in current cycle
assign rising_transition_c = ~n & input_signal;

always @(posedge clock) begin
   n <= #1 input_signal;
   rising_transition <= #1 rising_transition_c;
end
endmodule
```

```
module level_count (
   input clock,
   input input_signal,
   output reg [31:0] cycles_high
   );

reg [31:0] cycles_high_c;

//Every clock cycle, check if the input signal is high.
//Increment the level counter if high, hold if low.
always @(*) begin
   if (input_signal)
      cycles_high_c = cycles_high + 1'b1;
   else
      cycles_high_c = cycles_high;
end

//Instantiate flip-flops
always @(posedge clock) begin
   cycles_high <= cycles_high_c;
end
endmodule
```

[Tim Andreas, 2018/05/18]

```
//  This code is generated by Terasic System Builder
module top (
   //////////// CLOCK //////////
   input              ADC_CLK_10,
   input              MAX10_CLK1_50,
   input              MAX10_CLK2_50,

   //////////// SEG7 //////////
   output   [7:0]     HEX0,
   output   [7:0]     HEX1,
   output   [7:0]     HEX2,
   output   [7:0]     HEX3,
   output   [7:0]     HEX4,
   output   [7:0]     HEX5,

   //////////// KEY //////////
   input    [1:0]     KEY,

   //////////// LED //////////
   output   [9:0]     LEDR,

   //////////// SW //////////
   input    [9:0]     SW
);

//----- reg and wire declarations

// alias for clock signal
wire clk = MAX10_CLK1_50;
wire reset = ~KEY[0];

// input/output registers
reg [1:0] KEY_post;
reg [9:0] SW_post;

wire [31:0] SW_level_count;
wire [31:0] KEY_level_count;

reg [7:0] SW_edge_count;
reg [7:0] KEY_edge_count;

reg [23:0] hex_input;

//----- Main
hex hex0(.in(hex_input[3:0]),    .hex(HEX0));
hex hex1(.in(hex_input[7:4]),    .hex(HEX1));
hex hex2(.in(hex_input[11:8]),   .hex(HEX2));
hex hex3(.in(hex_input[15:12]),  .hex(HEX3));
hex hex4(.in(hex_input[19:16]),  .hex(HEX4));
hex hex5(.in(hex_input[23:20]),  .hex(HEX5));

level_count (.clock(clk), .input_signal(SW_post[9]),    .cycles_high(SW_level_count));
level_count (.clock(clk), .input_signal(~KEY_post[1]), .cycles_high(KEY_level_count));

edge_detection (.clock(clk), .input_signal(SW_post[9]),    .rising_transition(SW_edge));
edge_detection (.clock(clk), .input_signal(~KEY_post[1]), .rising_transition(KEY_edge));

always @(*) begin
   case (SW[2:0])
      3'b000: hex_input = {16'h0000, SW_edge_count};
      3'b001: hex_input = {16'h0000, KEY_edge_count};
      3'b010: hex_input = SW_level_count[23:0];
      3'b011: hex_input = KEY_level_count[23:0];
      3'b100: hex_input = {16'h0000, SW_edge_count};
      3'b101: hex_input = {16'h0000, KEY_edge_count};
      3'b110: hex_input = {16'h0000, SW_level_count[31:24]};
      3'b111: hex_input = {16'h0000, KEY_level_count[31:24]};
   endcase
end

always @(posedge clk) begin
   //register inputs for better timing
   KEY_post <= KEY;
   SW_post <= SW;

   KEY_edge_count <= KEY_edge_count + KEY_edge;
   SW_edge_count <= SW_edge_count + SW_edge;
end
endmodule
```

Best Solution

- In some cases there can be a race condition in the way the synthesis tool forms the circuit
- Two solutions were found:
  1. registering the inputs as they arrive from the SW or KEY
  2. registering the output of the edge detector circuit
- Registering the output was observed to always avoid the issue but the race condition is not guaranteed to be avoided
- The best solution is to register the inputs as soon as they arrive