Secuenciales II

Diseño de Sistemas con FPGA

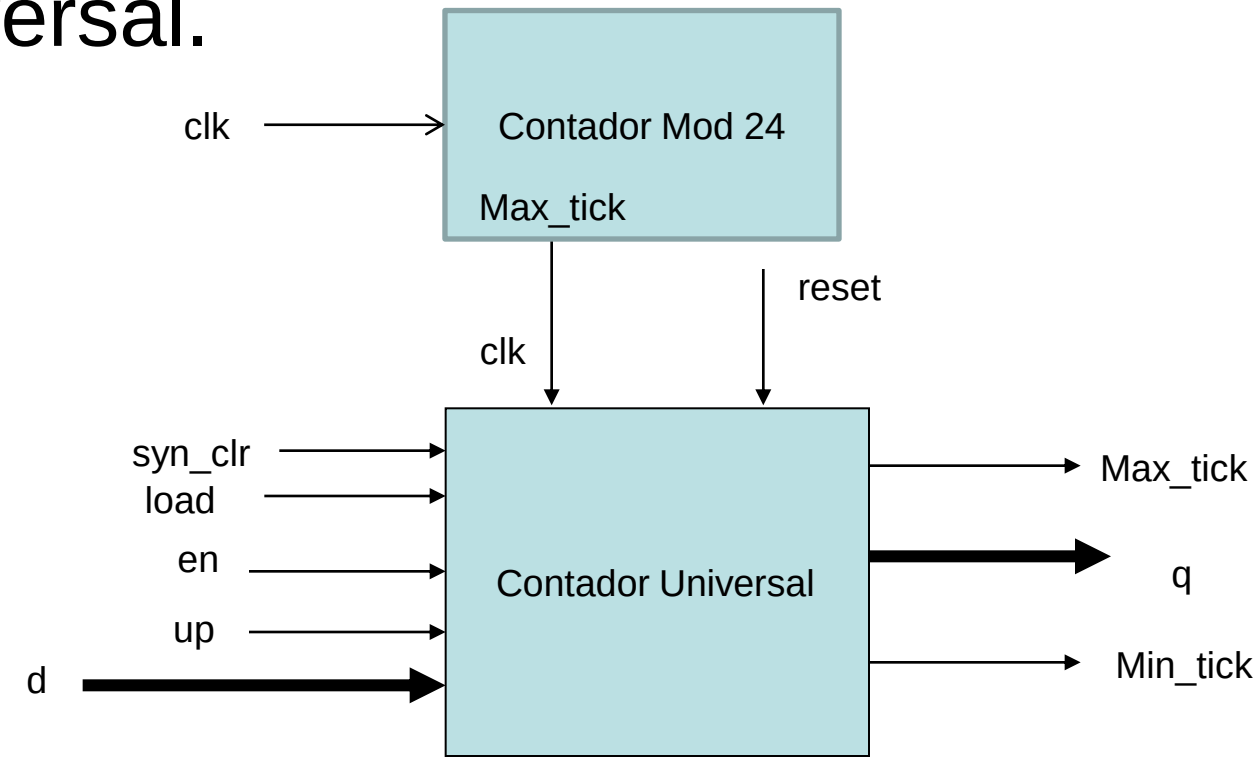
Patricia Borensztejn

En la clase pasada…

- Hicimos un contador universal
- Algunos de ustedes se dieron cuenta que la frecuencia de conteo, no podía ser la frecuencia del reloj externo de 50 MHz, porque eso es muy rápido para verlo en los leds o en los siete segmentos
- Algunos de ustedes entonces utilizaron los pushbottoms como reloj…
- Otros usaron un contador modulo 24 o 25 para usar la salida max_tick o min_tick como entrada del reloj del contador

En la clase pasada…

- Otros usaron un contador módulo 24 o 25 para usar la salida max_tick o min_tick como entrada del reloj del contador universal.



Módulo clk

- Podemos hacer un módulo que nos sirva para utilizarlo cada vez que deseemos bajar la frecuencia del reloj.
- Teniendo en cuanta la siguiente ecuación:

$$Escala = \frac{25000000}{Frecuencia}$$

Donde la frecuencia es la deseada, y la constante es la cantidad de ciclos por segundo (dividido por dos) , el módulo clk quedaría….

Módulo clock

```
// Basys Board and Spartan-3E Starter Board
// Crystal Clock Oscillator  clkosc.v
// c 2008 Embedded Design using Programmable Gate Arrays  Dennis Silage

module clock(input CCLK, input [31:0] clkscale, output reg clk);
// CCLK crystal clock oscillator 50 MHz

reg [31:0] clkq = 0;                              // clock register, initial value of 0

always@(posedge CCLK)
   begin
      clkq=clkq+1;                    // increment clock register
       if (clkq>=clkscale)            // clock scaling
          begin
               clk=~clk;              // output clock
               clkq=0;                // reset clock register
          end
   end

endmodule
```

Módulo TestClk

```
module testclk(
    input cclk,
    output ld0,
    output ld1,
    output ld2
    );

reloj M0(.cclk(cclk),.escala(25000000),.clk(ld0)); 1Hz
reloj M1(.cclk(cclk),.escala(12500000),.clk(ld1));  2Hz
reloj M2(.cclk(cclk),.escala(6250000),.clk(ld2));    4Hz
```

Modelado de Sistemas Secuenciales

- Según la función del siguiente estado se dividen en:
  - Circuitos secuenciales regulares: el estado siguiente se construye con un incrementador o un shifter
  - FSM (Finite State Machines): el estado siguiente no exhibe un pattern repetitivo, sino mas bien random.
  - FSMD (FSM with Data Path): combina los dos tipos anteriores de sistemas secuenciales, es decir, está compuesto por dos partes:
    - FSM: llamado “control path” es el encargado de examinar las entradas externas y de generar las señales para el funcionamiento de los
    - Circuitos secuenciales regulares: llamado “data path”.

FSM: Máquina de Estado Finita

- Se usa para modelar un sistema que transita por un número finito de estados internos.
- A diferencia de los sistemas secuenciales regulares, las transiciones de estado no exhiben un patrón repetitivo, y por lo tanto no pueden ser construidos con componentes standard sino que debe ser lógica ad-hoc. Es decir, la diferencia fundamental reside en la lógica de siguiente estado.

FSM

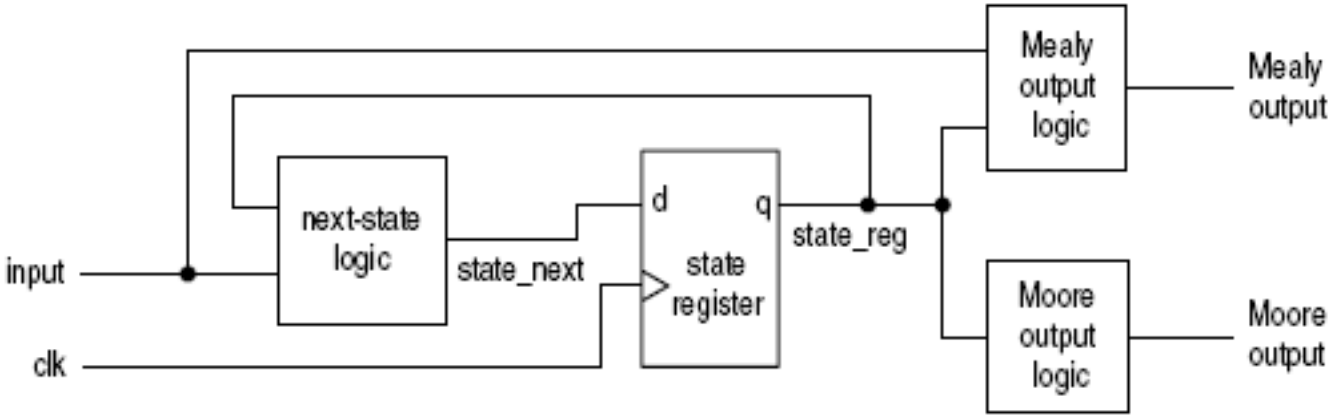
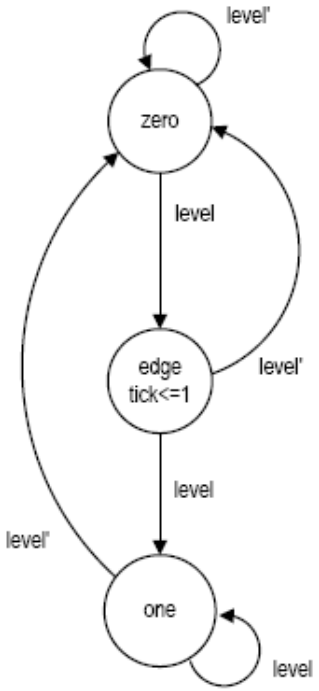


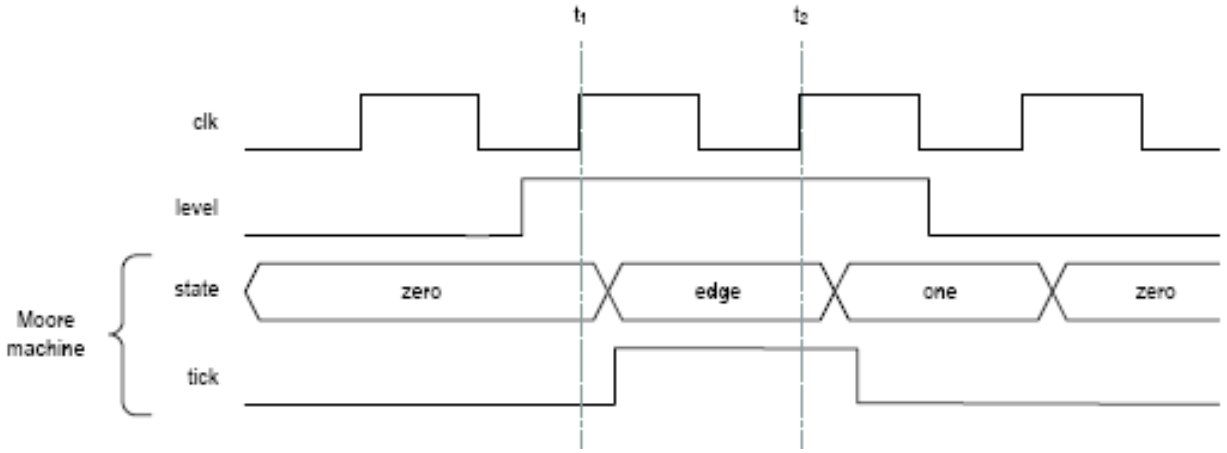
Figure 5.1 Block diagram of a synchronous FSM.

- Se representa mediante Diagramas de Estado (State Diagram) o bien mediante ASM (Algorithmic State Machine Chart).
- En cualquier caso, la implementación es como en el caso de los circuitos secuenciales regulares: se separa el registro de estado y se realiza la logica combinatoria para la función del estado siguiente y la salida.

FSM: Ejemplo
Detector de flanco de subida de una señal



(a) State diagram



FSM: Ejemplo

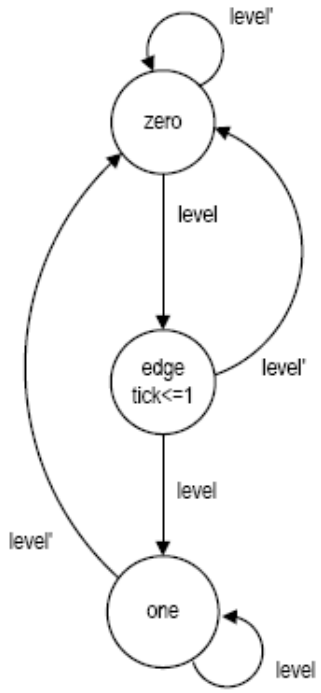
```
module edge_detect_moore
  (
   input wire clk, reset,
   input wire level,
   output reg tick
  );

  // symbolic state declaration
  localparam [1:0]
     zero = 2'b00,
     edg = 2'b01,
     one = 2'b10;

  // signal declaration
  reg [1:0] state_reg, state_next;

  // state register
   always @(posedge clk, posedge reset)
      if (reset)
         state_reg <= zero;
      else
         state_reg <= state_next;
```

```
  // next-state logic and output logic
    always @*
    begin
       state_next = state_reg;  // default state: the same
       tick = 1'b0;             // default output: 0
       case (state_reg)
          zero:
             if (level)
                state_next = edg;
          edg:
             begin
                tick = 1'b1;
                if (level)
                   state_next = one;
                else
                   state_next = zero;
              end
          one:
             if (~level)
                state_next = zero;
          default: state_next = zero;
       endcase
    end
endmodule
```

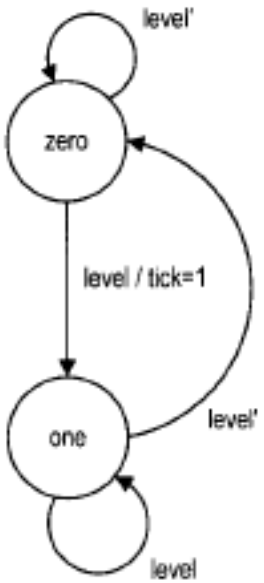
Máquinas de Estado

- Máquina de Moore es aquella FSM donde las salidas son solo función del estado
- Máquina de Mealy es aquella FSM donde las salidas son función del estado y de la entrada externa.
- Ambas máquinas solo difieren en la función de salida.
- Una FSM compleja puede contener los dos tipos de salidas.

Máquina de Moore y de Mealy



(a) State diagram



Máquina de Moore y de Mealy

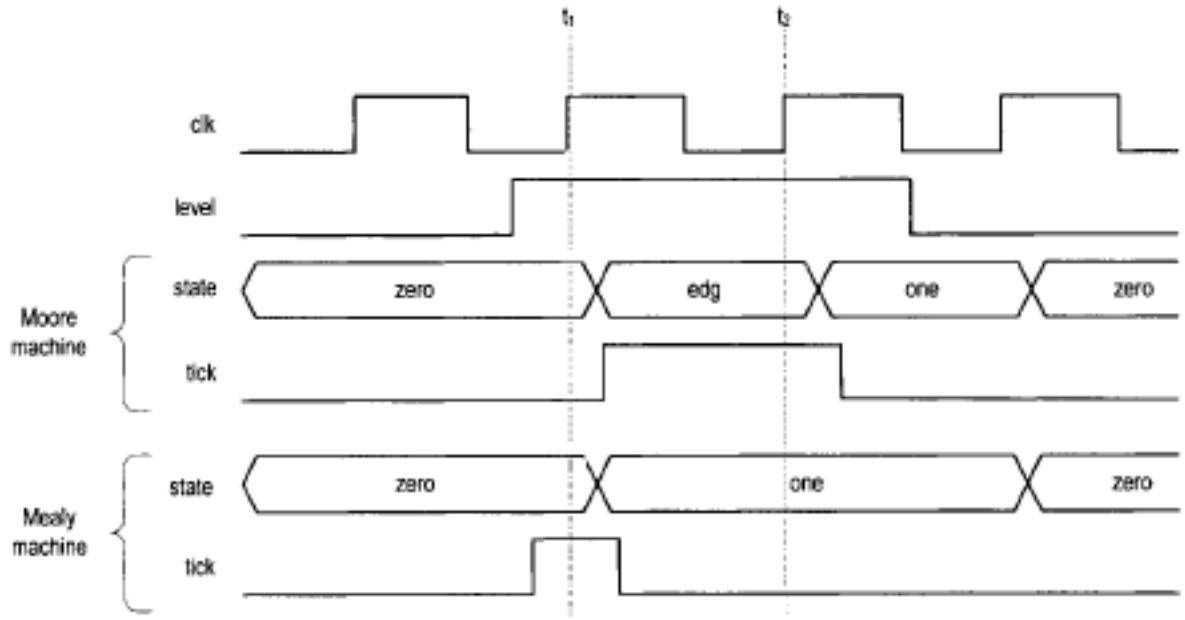


Figure 5.5 Timing diagram of two edge detectors.

Código para Mealy

```
// Listing 5.4
module edge_detect_mealy
   (
    input wire  clk, reset,
    input wire  level,
    output reg tick
   );

   // symbolic state declaration
   localparam zero = 1'b0,
              one = 1'b1;

   // signal declaration
   reg state_reg, state_next;

   // state register
    always @(posedge clk, posedge reset)
       if (reset)
          state_reg <= zero;
       else
          state_reg <= state_next;
```

```
 // next-state logic and output logic
   always @*
   begin
      state_next = state_reg;  // default state: the same
      tick = 1'b0;             // default output: 0
      case (state_reg)
         zero:
            if (level)
               begin
                  tick = 1'b1;
                  state_next = one;
               end
         one:
            if (~level)
               state_next = zero;
         default: state_next = zero;
      endcase
   end

endmodule
```

Código Moore vs Mealy

```
 // next-state logic
always @*
   begin
      state_next = state_reg;  // default state: the
same
   case (state_reg)
         zero:
            if (level)
               state_next = edg;
         edg:
             if (level)
                  state_next = one;
               else
                  state_next = zero;
         one:
            if (~level)
               state_next = zero;
         default: state_next = zero;
      endcase
   end
//output logic
assign tick= (state_reg==edg) ? 1'b1 : 1'b0;
endmodule
```

```
 // next-state logic and output logic
   always @*
   begin
      state_next = state_reg;  // default state: the same
      tick = 1'b0;             // default output: 0
      case (state_reg)
         zero:
            if (level)
               begin
                  tick = 1'b1;
                  state_next = one;
               end
         one:
            if (~level)
               state_next = zero;
         default: state_next = zero;
      endcase
   end

endmodule
```

Sutiles Diferencias

- …. Entre ambos modelos
- Mealy tiene menos estados
- El modelo de Mealy tiene disponible la salida un ciclo antes que Moore
- El problema es que los defasajes de las entradas pasan a las salidas…cosa que no seria un problema si las salidas del sistema son entradas a otro sistema secuencial síncrono que muestrea la señal con el mismo clk….
- Sin embargo…. Usamos Moore.

Otro Ejemplo: Circuito Antirrebote

- Como el switch es un dispositivo mecánico, al ser este presionado, puede rebotar mas de una vez antes de quedar establecida la señal. El tiempo de establecimiento puede ser aproximadamente de 20 ms.
- El propósito del circuito antirrebote es el de filtrar los rebotes.

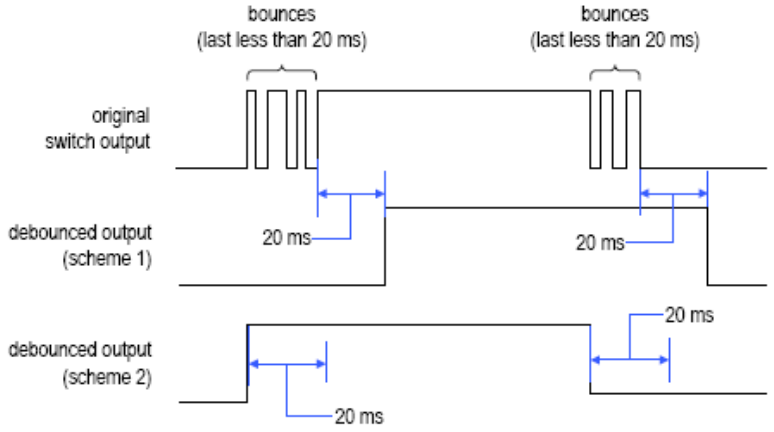
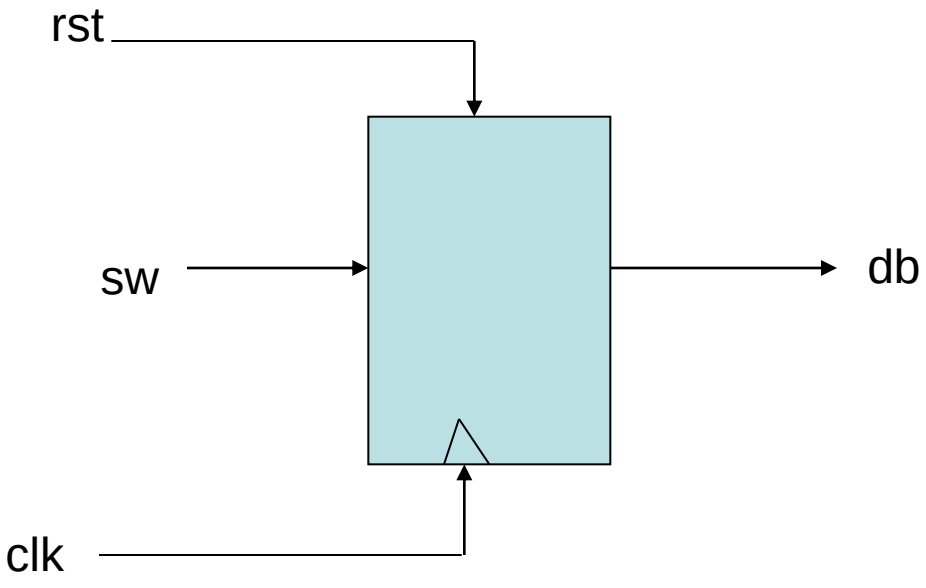


Figure 5.8 Original and debounced waveforms.

Circuito Antirrebote

- La idea es utilizar un timer que genere una señal cada 10 ms, y una FSM. La FSM usa esta información para saber si la entrada está o no estabilizada.
- Si la entrada sw está activa por mas de 20 ms, entonces ha habido un evento.



Circuito Antirrebote: Diagrama de Estados (FSM)

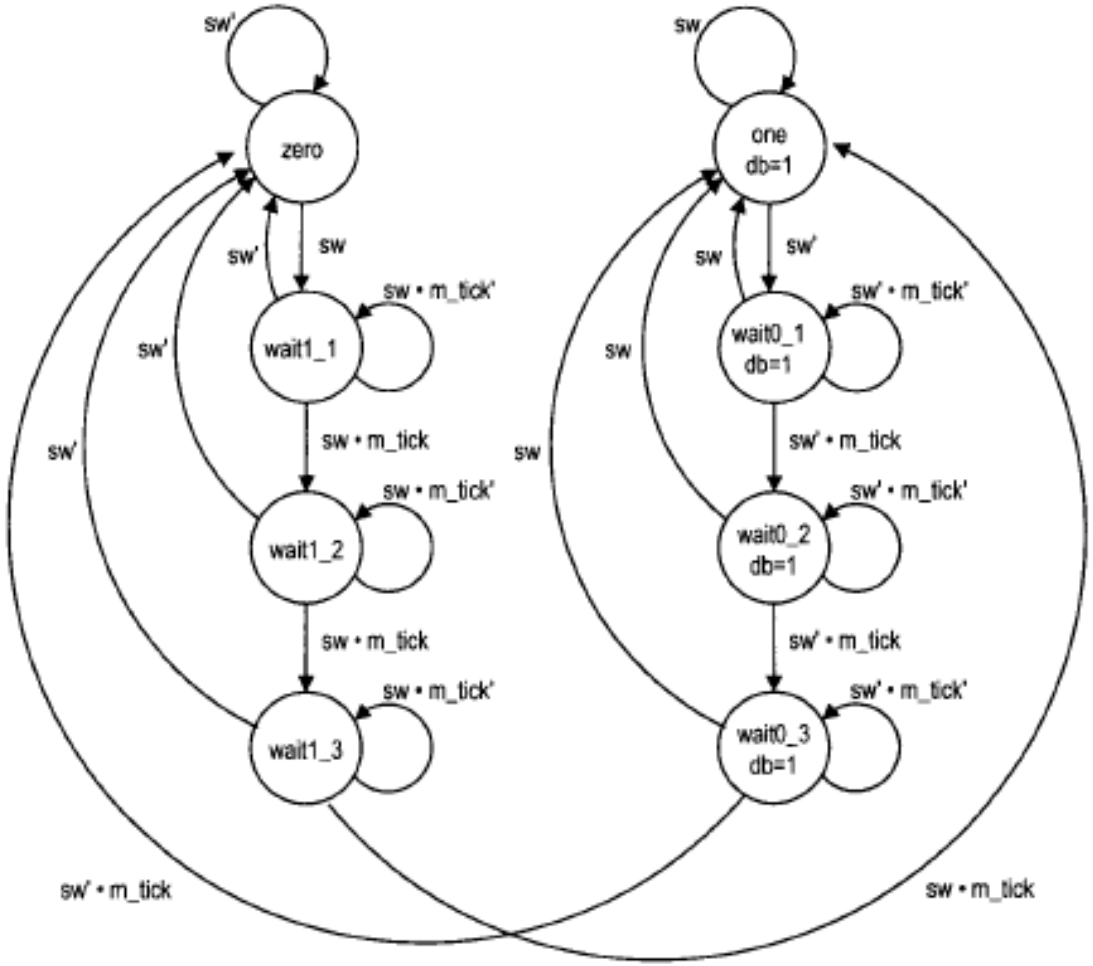


Figure 5.9 State diagram of a debouncing circuit.

Código Circuito Antirrebote

```
module db_fsm
   (
    input wire clk, reset,
    input wire sw,
    output reg db
   );

   // symbolic state declaration
   localparam  [2:0]
               zero    = 3'b000,
               wait1_1 = 3'b001,
               wait1_2 = 3'b010,
               wait1_3 = 3'b011,
               one     = 3'b100,
               wait0_1 = 3'b101,
               wait0_2 = 3'b110,
               wait0_3 = 3'b111;

   // number of counter bits (2^N * 20ns = 10ms tick)
   localparam N =19;

   // signal declaration
   reg [N-1:0] q_reg;
   wire [N-1:0] q_next;
   wire m_tick;
   reg [2:0] state_reg, state_next;
```

```
// body

   //=============================================
   // counter to generate 10 ms tick
   //=============================================
   always @(posedge clk)
      q_reg <= q_next;
   // next-state logic
   assign q_next = q_reg + 1;
   // output tick
   assign m_tick = (q_reg==0) ? 1'b1 : 1'b0;

   //=============================================
   // debouncing FSM
   //=============================================
   // state register
    always @(posedge clk, posedge reset)
      if (reset)
         state_reg <= zero;
      else
         state_reg <= state_next;
```

Código Circuito Antirrebote

```
// next-state logic and output logic
   always @*
   begin
      state_next = state_reg;  // default state: the same
      db = 1'b0;               // default output: 0
      case (state_reg)
         zero:
            if (sw)
               state_next = wait1_1;
         wait1_1:
            if (~sw)
               state_next = zero;
            else
               if (m_tick)
                  state_next = wait1_2;
         wait1_2:
            if (~sw)
               state_next = zero;
            else
               if (m_tick)
                  state_next = wait1_3;
         wait1_3:
            if (~sw)
               state_next = zero;
            else
               if (m_tick)
                  state_next = one;
```

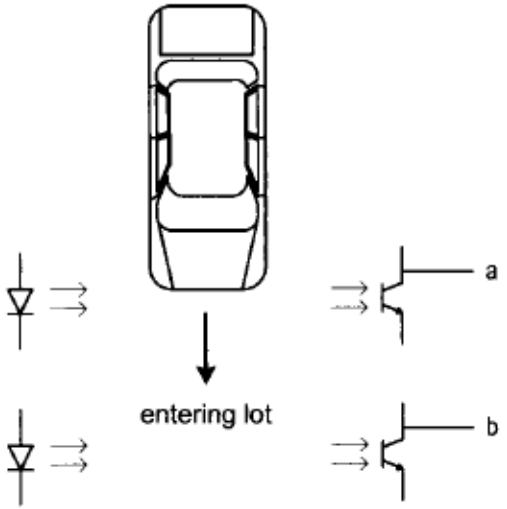
```
one:
            begin
              db = 1'b1;
              if (~sw)
                 state_next = wait0_1;
            end
         wait0_1:
            begin
               db = 1'b1;
               if (sw)
                  state_next = one;
               else
                 if (m_tick)
                    state_next = wait0_2;
            end
         wait0_2:
            begin
               db = 1'b1;
               if (sw)
                  state_next = one;
               else
                 if (m_tick)
                    state_next = wait0_3;
            end
```

```
wait0_3:
            begin
               db = 1'b1;
               if (sw)
                  state_next = one;
               else
                 if (m_tick)
                    state_next = zero;
            end
          default: state_next = zero;
      endcase
   end

endmodule
```

Algo a considerar: los ticks actúan como señales combinacionales!
Fijarse que solo estará activa durante un ciclo de reloj.

Aplicación

- Se desea diseñar una aplicación para determinar el número de autos que han utilizado un parquímetro.
- Cada parquímetro cuenta con dos pares de sensores de luz (emisor-receptor) como se indica en la figura.
- El parquímetro tiene una única entrada y salida.



Aplicación

- Monitoreando los dos sensores (a y b) se puede determinar si:
  - Un auto está entrando
    - Inicialmente los dos sensores están no bloqueados (a=b=0)
    - Se bloquea el sensor a (ab=10)
    - Se bloquean los dos sensores (ab=11)
    - El sensor a se desbloquea (ab=01)
    - Los dos sensores estan desbloqueados (ab=00)
  - Un auto está saliendo
  - Un hombre está pasando
- La aplicación debe incrementar un contador por cada auto que ha ocupado el parquímetro en el día.
- La aplicación debe activar dos señales de salida: enter y exit.
- Los sensores a y b deben simularse, con el comportamiento adecuado.