

Circuitos Secuenciales en FPGA

Parte II: Finite State Machine

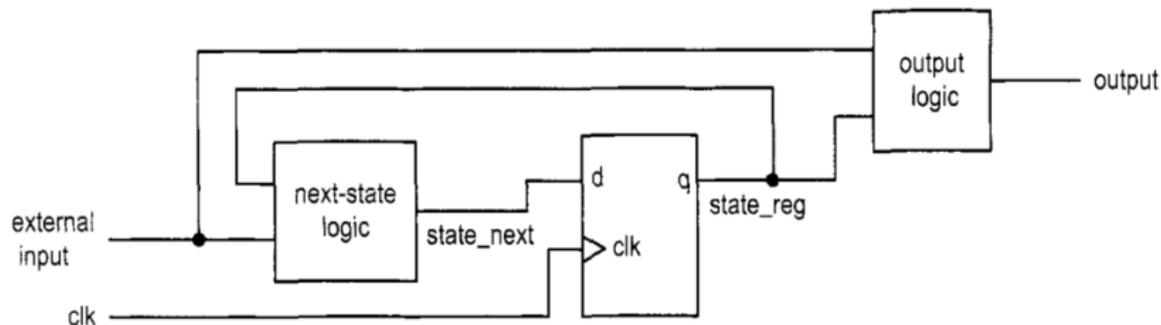
Introducción a la Microfabricación y las FPGA

Instituto Balseiro

9 de Septiembre 2013

- Introducción a los circuitos secuenciales (i.e, con memoria). Tipos de circuitos secuenciales.
- **Metodología de diseño sincrónico**: separar el registro de estado de la lógica combinacional para calcular el próximo estado y los outputs.
- Código para inferir D FF, Registros y Banco de registros para estado.
- Desarrollo de varios ejemplos: Shift Register, Contador Binario Universal, Display multiplexing, Display multiplexing con Hexa.
- **Diseño jerárquico** y reutilización de módulos diseñados anteriormente.

IMPORTANTE: Metodo de Diseño



- Dividir *claramente* el diseño (y por lo tanto el código) en los tres bloques.

0. Definición de Entity

Antes de empezar: identificar las *entradas y salidas* que tiene el circuito, y sus *parámetros*.

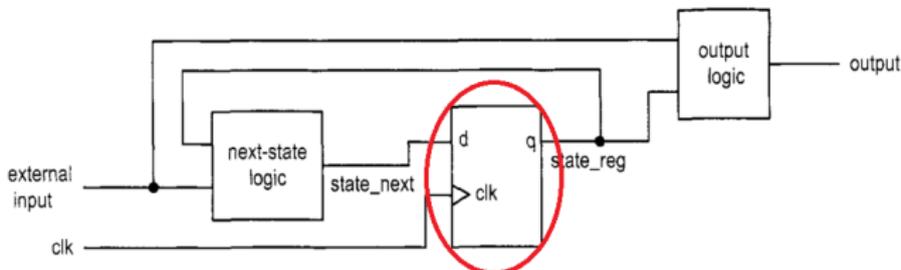
- En diseño síncrono, siempre están como entradas el `clk` y el `reset`.
- Pensar qué tamaños son parametrizables para incluir como genéricos.
- Pensar en las librerías y paquetes a incluir.

syn_clr	load	en	up	q*	Operation
1	-	-	-	00...00	synchronous clear
0	1	-	-	d	parallel load
0	0	1	1	q+1	count up
0	0	1	0	q-1	count down
0	0	0	-	q	pause

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity univ_bin_counter is
    generic(N: integer := 8);
    port(
        clk, reset: in std_logic;
        syn_clr, load, en, up: in std_logic;
        d: in std_logic_vector(N-1 downto 0);
        max_tick, min_tick: out std_logic;
        q: out std_logic_vector(N-1 downto 0)
    );
end univ_bin_counter;
  
```

1. Estado



Primero: identificar el *estado* que tiene el circuito.

- Por cada señal a recordar, definir dos señales `_reg` y `_next`. Ej: `r_reg` y `r_next`.
- El código de esta parte es bien claro:

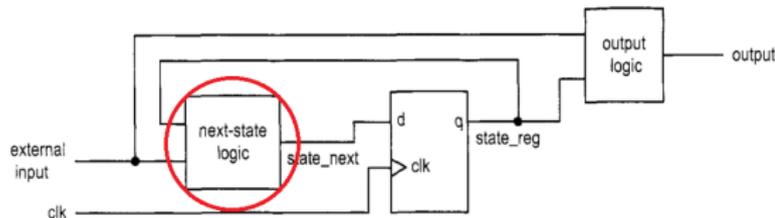
```

architecture arch of univ_bin_counter is
    signal r_reg: unsigned(N-1 downto 0);
    signal r_next: unsigned(N-1 downto 0);
begin
    -- register
    process (clk, reset)
    begin
        if (reset='1') then
            r_reg <= (others=>'0');
        elsif (clk'event and clk='1') then
            r_reg <= r_next;
        end if;
    end process;

```

2. Lógica del próximo estado

Segundo: Escribir la lógica del próximo estado. ¿Qué valor van a tomar mis registros en el próximo reloj?



- Calcular el *próximo valor* (i.e, r_next) en función del *valor actual* r_reg y las señales de entrada.
- El código puede ser con sentencias concurrentes o dentro de un process.
- Si está en un process, la lista de sensibilidad incluye a los `***_reg` y a las entradas.

syn_clr	load	en	up	q*	Operation
1	-	-	-	00...00	synchronous clear
0	1	-	-	d	parallel load
0	0	1	1	q+1	count up
0	0	1	0	q-1	count down
0	0	0	-	q	pause

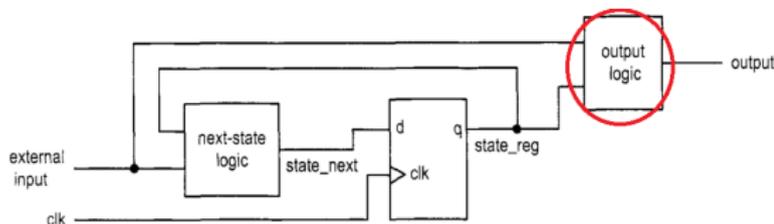
— *next-state logic*

```
r_next <= (others=>'0') when syn_clr='1' else
          unsigned(d)   when load='1' else
          r_reg + 1     when en='1' and up='1' else
          r_reg - 1     when en='1' and up='0' else
          r_reg;
```

3. Lógica de Salida

Tercero: Escribir la lógica de salida.

- Calcular las salidas en *este reloj* (i.e, q) en función del *valor actual* r_reg y las señales de entrada.
- El código puede ser con sentencias concurrentes o dentro de un process.
- Si está en un process, la lista de sensibilidad incluye a los `***_reg` y a las entradas.



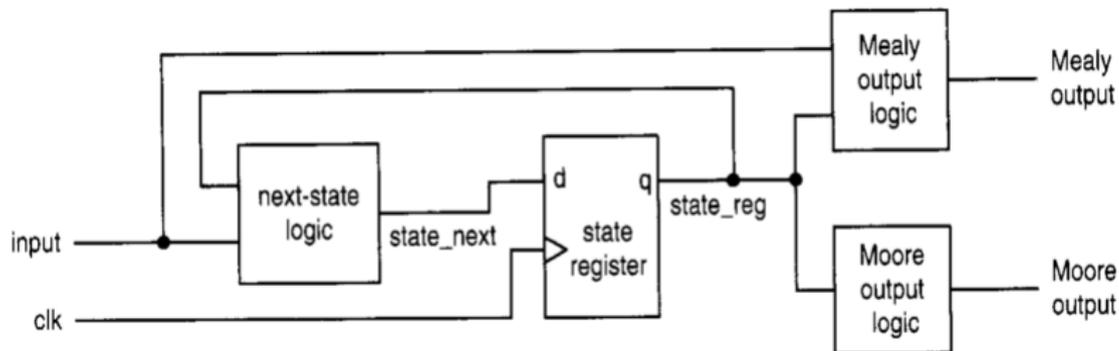
-- *output logic*

```
q <= std_logic_vector(r_reg);
max_tick <= '1' when r_reg=(2**N-1) else '0';
min_tick <= '1' when r_reg=0 else '0';
```

Máquina de estados, máquina de finitos estados.

- Modelar un sistema que pasa por varios estados internos. La transición entre los estados depende del estado interno y de los inputs.
- No presentan un patrón regular de transición.
- *Son muy útiles para implementar el control de algoritmos complejos.*

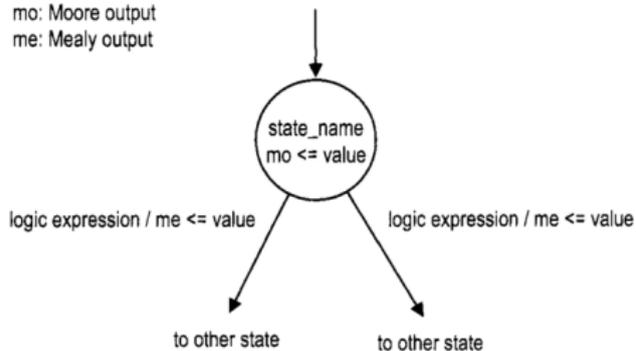
FSM: Diseño síncrono



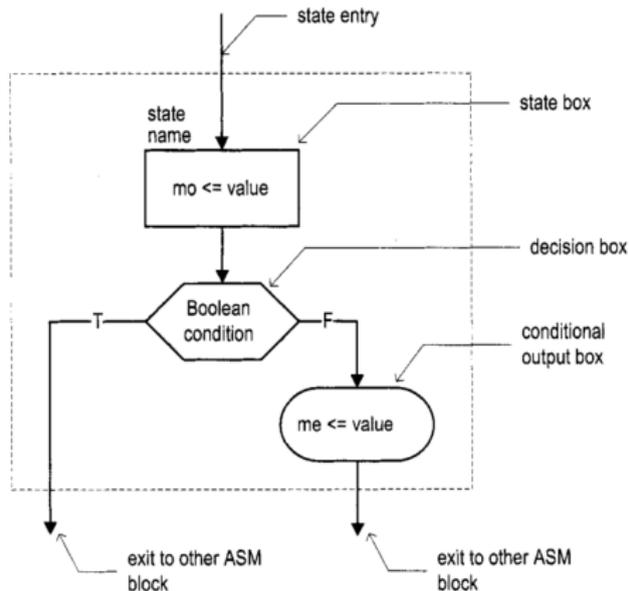
Lo que cambia es la lógica del próximo estado (de regular a FSM).

Representación

mo: Moore output
me: Mealy output

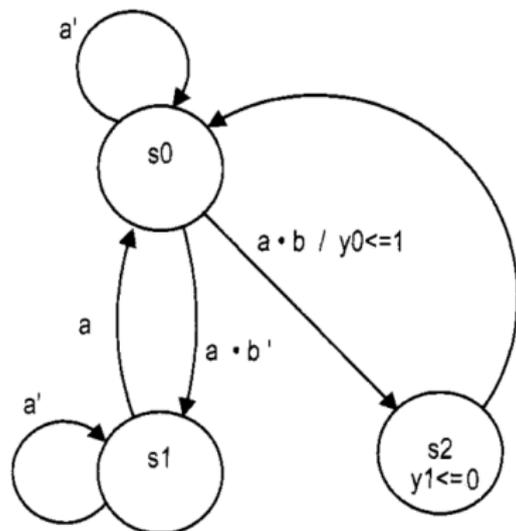


(a) Node

Finite State Machine*Algorithmic State Machine Chart*

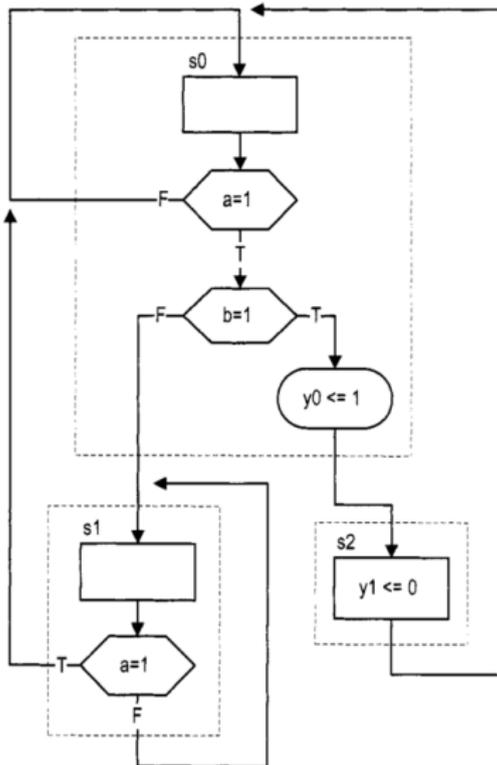
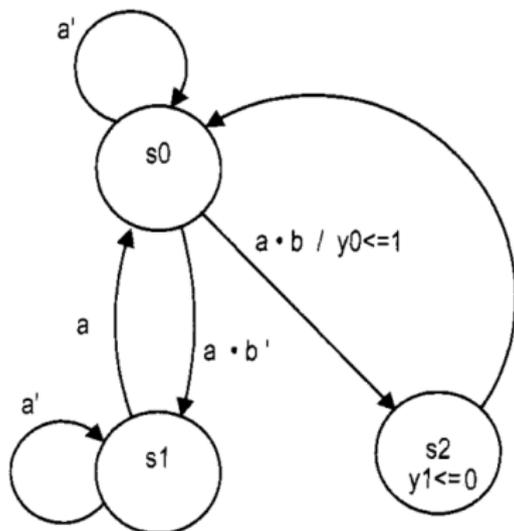
Ejemplo

Entradas: clk, reset, a,b. Salidas: y0, y1. Por defecto: $y_0 \leq '0'$, $y_1 \leq '1'$



Ejemplo

Entradas: clk, reset, a,b. Salidas: y0, y1. Por defecto: $y0 \leq '0'$, $y1 \leq '1'$



0. Declaración de Entidad

Antes de empezar: identificar las *entradas y salidas* que tiene el circuito, y sus *parámetros*.

- En diseño síncrono, siempre están como entradas el `clk` y el `reset`.
- Pensar qué tamaños son parametrizables para incluir como genéricos.
- Pensar en las librerías y paquetes a incluir.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity ejemplo is
    port (
        clk, reset : in std logic;
        a,b: in std logic;
        y0, y1: out std logic
    );
end ejemplo;
```

1. Estado

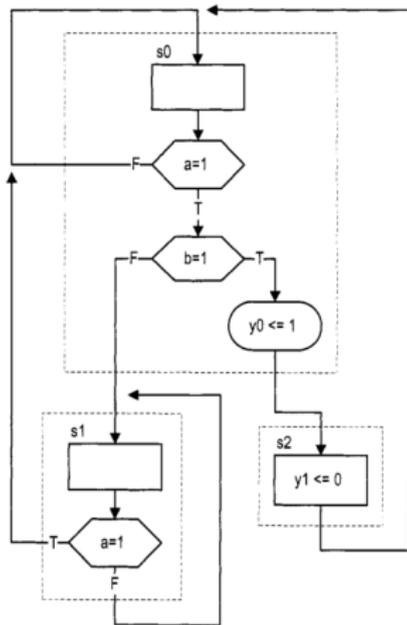
Primero: aislar el *registro de estado*, separándolos de la lógica del próximo estado y de la lógica de la salida.

- Para los estados: usar el tipo de datos enumerate de VHDL.
- El sintetizador codifica estos estados como números binarios.
- Definir dos señales `state_reg` y `state_next` (¡igual que antes!)
- El código del registro tb es igual que antes.

```
architecture verbose of ejemplo is
    type eg state type is (s0, s1, s2);
    signal state_reg, state_next : eg state type;
begin
    -- state register
    process (clk, reset)
    begin
        if (reset = '1') then
            state_reg <= s0;
        elsif (clk'event and clk = '1') then
            state_reg <= state_next;
        end if;
    end process;
end;
```

2. Lógica del próximo estado

Definir un process con un case que siga *exactamente* el diagrama FSM o ASM. Lista de sensibilidad: state_reg y todas las entradas.

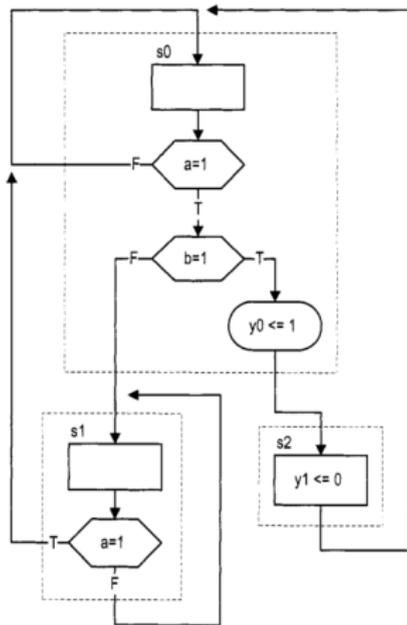


```

-- next state logic
process (state reg, a, b)
begin
  case state reg is
    when s0 =>
      if a = '1' then
        if b='1' then
          state next <= s2;
        else
          state next <= s1;
        end if;
      else
        state next <= s0;
      end if;
    when s1 =>
      if a = '1' then
        state next <= s0;
      else
        state next <= s1;
      end if;
    when s2 =>
      state next <= s0;
  end case;
end process;
  
```

3. Lógica de Salida

Definir dos process, uno para salidas Moore y otro para Mealy, con un case que siga *exactamente* el diagrama FSM o ASM.



```

-- Moore output
process(state reg)
begin
  case state reg is
    when s0 | s1 =>
      y1 <= '1';
    when s2 =>
      y1 <= '0';
  end case;
end process;

```

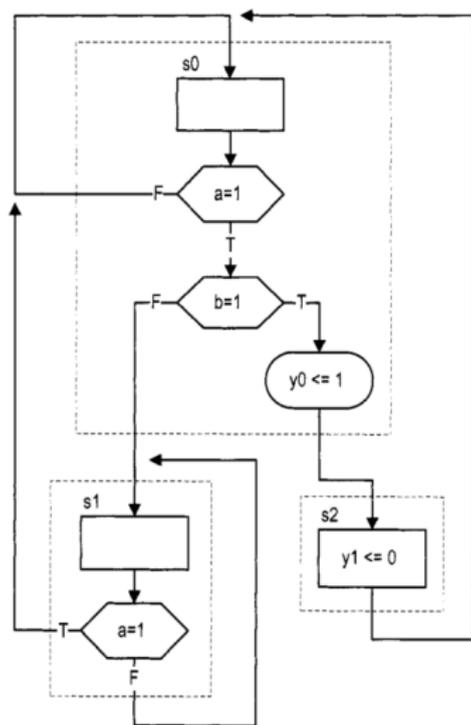
```

-- Mealy output
process(state reg, a, b)
begin
  case state reg is
    when s0 =>
      if a = '1' and b = '1' then
        y0 <= '1';
      else
        y0 <= '0';
      end if;
    when s1 | s2 =>
      y0 <= '0';
  end case;
end process;

```

Valores por defecto

En la lógica del próximo estado:

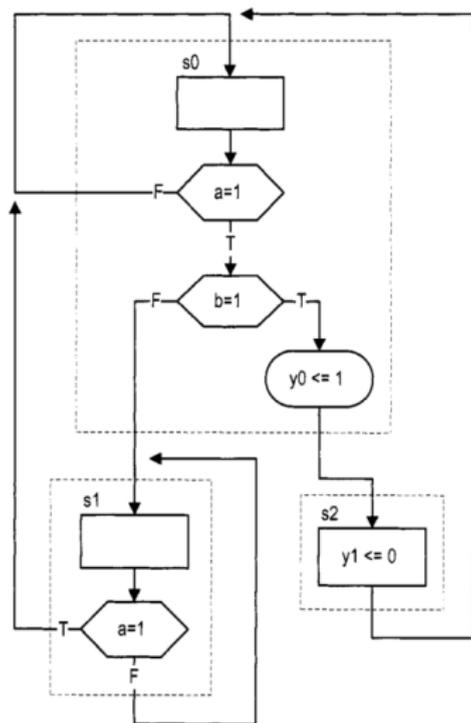


```

-- next state logic
process(state reg, a, b)
begin
  -- valor defecto: me quedo en
  -- el mismo estado!
  state next <= state reg;
  case state reg is
    when s0 =>
      if a = '1' then
        if b='1' then
          state next <= s2;
        else
          state next <= s1;
        end if;
      --else
      -- state next <= s0;
    end if;
    when s1 =>
      if a = '1' then
        state next <= s0;
      --else
      -- state next <= s1;
    end if;
    when s2 =>
      state next <= s0;
  end case;
end process;
  
```

Valores por defecto - cont

En la lógica de salida:



```

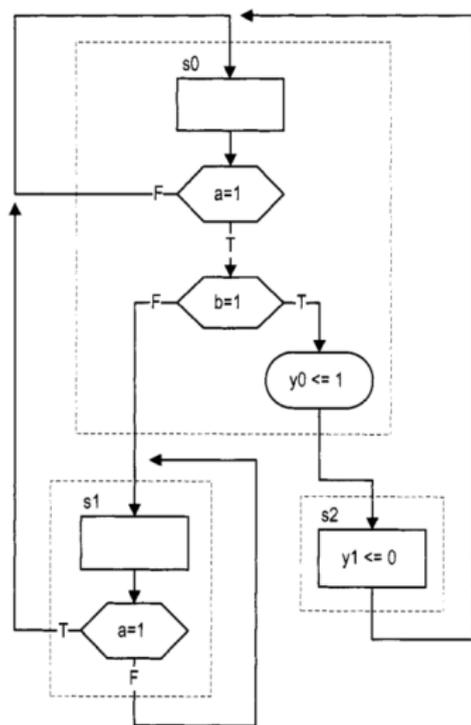
-- Moore output
process(state reg)
begin
  y1 <= '1';
  case state reg is
    when s0 | s1 =>
      y1 <= '1';
    when s2 =>
      y1 <= '0';
  end case;
end process;
  
```

```

-- Mealy output
process(state reg, a, b)
begin
  y0 <= '0';
  case state reg is
    when s0 =>
      if a = '1' and b = '1' then
        y0 <= '1';
      --else
      -- y0 <= '0';
      end if;
    when s1 | s2 =>
      y0 <= '0';
  end case;
end process;
  
```

Lógica de próximo estado y salida juntas

Todo junto con valores default



```

-- next state/output logic
process(state reg, a, b)
begin
  -- valores por defecto
  state next <= state reg;
  y0 <= '0';
  y1 <= '1';
  case state reg is
    when s0 =>
      if a = '1' then
        if b='1' then
          state next <= s2;
          y0 <= '1';
        else
          state next <= s1;
        end if;
      end if;
    when s1 =>
      if a = '1' then
        state next <= s0;
      end if;
    when s2 =>
      state next <= s0;
      y1 <= '0';
    end case;
end process;
  
```

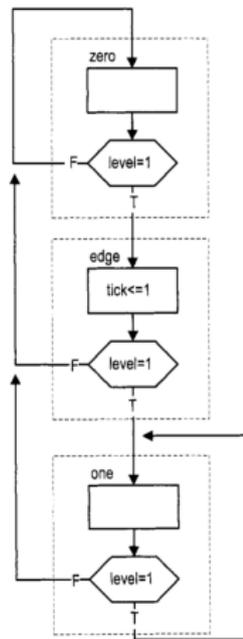
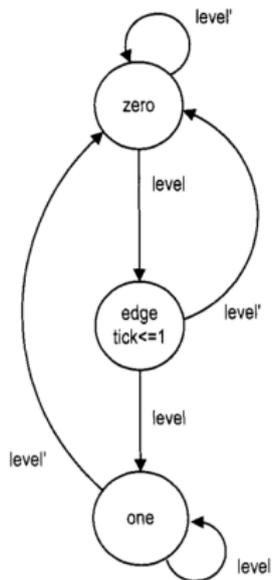
Detector de Rising Edge

Un circuito que cuando detecta que la señal de entrada cambia de '0' a '1', genera un pulso de un ciclo de reloj en la salida. ¡Primero DISEÑO!

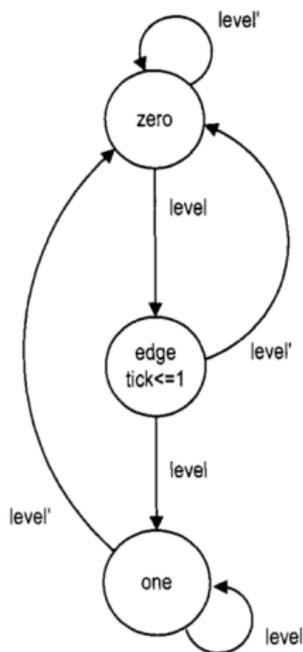
Detector de Rising Edge

Un circuito que cuando detecta que la señal de entrada cambia de '0' a '1', genera un pulso de un ciclo de reloj en la salida. ¡Primero DISEÑO!

Entradas: `clk`, `reset`, `level`. Salida: `tick`. Por defecto: `tick <= '0'`.



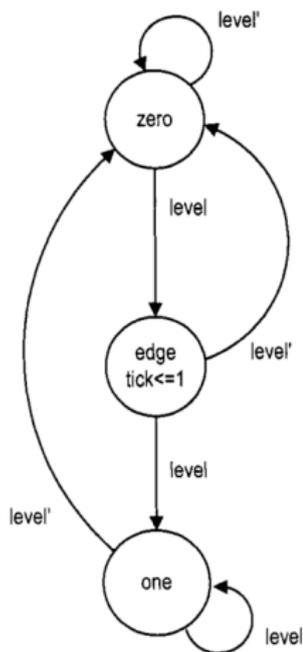
0. Declaración de Entidad



```
library IEEE;
use IEEE.STD LOGIC 1164.ALL;

entity edge detector is
    port(
        clk, reset : in std logic;
        level : in std logic;
        tick : out std logic
    );
end edge detector;
```

1. Estado



```

architecture moore of edge detector is
  type state type is (zero, edge, one);
  signal state req, state next : state type;
begin

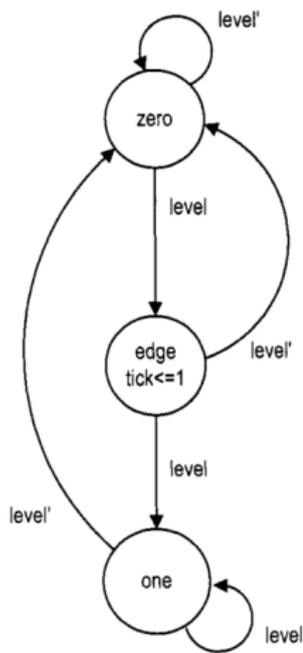
```

```

  -- registro de estado
  process(clk, reset)
  begin
    if (reset = '1') then
      state req <= zero;
    elsif (clk'event and clk = '1') then
      state req <= state next;
    end if;
  end process;

```

2. Lógica del próximo estado

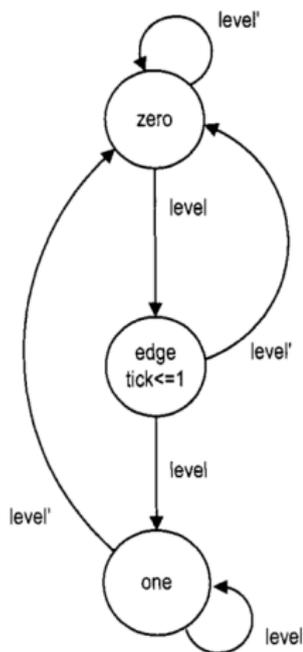


```

-- next state logic
process(state req, level)
begin
  state next <= state req; -- default
  case state req is
    when zero =>
      if level = '1' then
        state next <= edge;
      end if;
    when edge =>
      if level = '1' then
        state next <= one;
      else
        state next <= zero;
      end if;
    when one =>
      if level = '0' then
        state next <= zero;
      end if;
  end case;
end process;

```

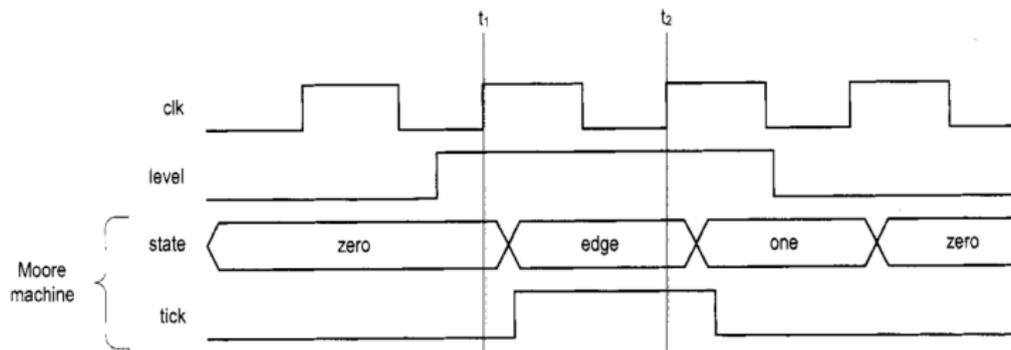
3. Lógica de Salida



```

-- output logic: moore
process(state req)
begin
  case state req is
    when edge =>
      tick <= '1';
    -- default
    when others =>
      tick <= '0';
  end case;
end process;

```



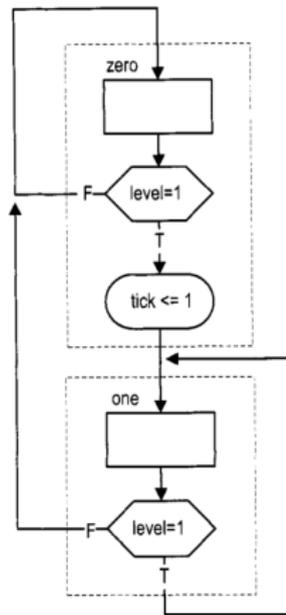
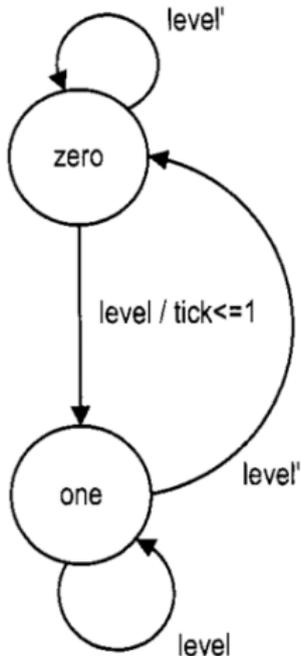
Detector de Rising Edge

Un circuito que cuando detecta que la señal de entrada cambia de '0' a '1', genera un pulso de un ciclo de reloj en la salida. ¡Primero DISEÑO!

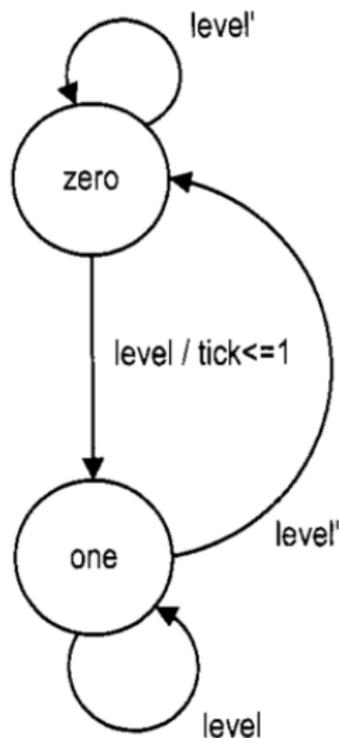
Detector de Rising Edge

Un circuito que cuando detecta que la señal de entrada cambia de '0' a '1', genera un pulso de un ciclo de reloj en la salida. ¡Primero DISEÑO!

Entradas: clk , $reset$, $level$. Salida: $tick$. Por defecto: $tick \leq '0'$.



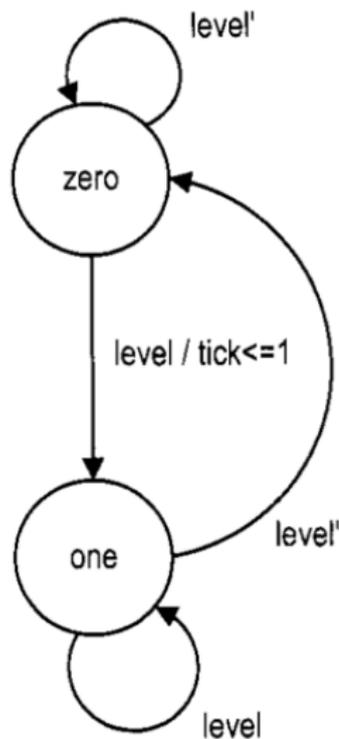
0. Declaración de Entidad



```
library IEEE;
use IEEE.STD LOGIC 1164.ALL;

entity edge detector is
  port(
    clk, reset : in std logic;
    level : in std logic;
    tick : out std logic
  );
end edge detector;
```

1. Estado



```

architecture mealy of edge detector is
  type state type is (zero, one);
  signal state reg, state next : state type;
begin

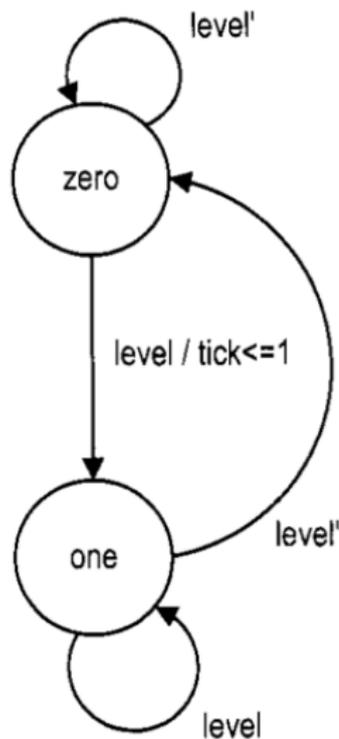
```

```

  -- registro de estado
  process(clk, reset)
  begin
    if (reset = '1') then
      state reg <= zero;
    elsif (clk'event and clk = '1') then
      state reg <= state next;
    end if;
  end process;

```

2. Lógica del próximo estado

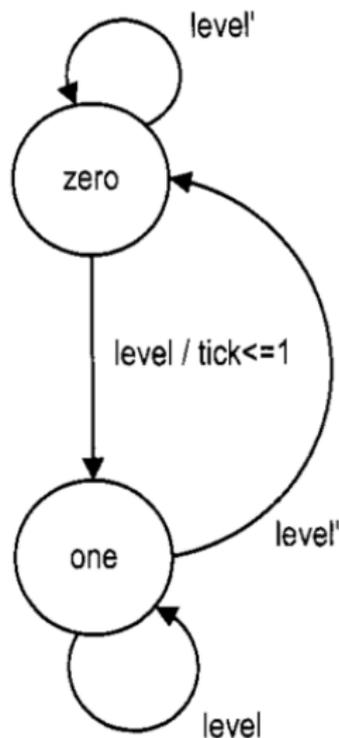


```

-- next state logic
process(state req, level)
begin
  state next <= state req; -- default
  case state req is
    when zero =>
      if level = '1' then
        state next <= one;
      end if;
    when one =>
      if level = '0' then
        state next <= zero;
      end if;
  end case;
end process;

```

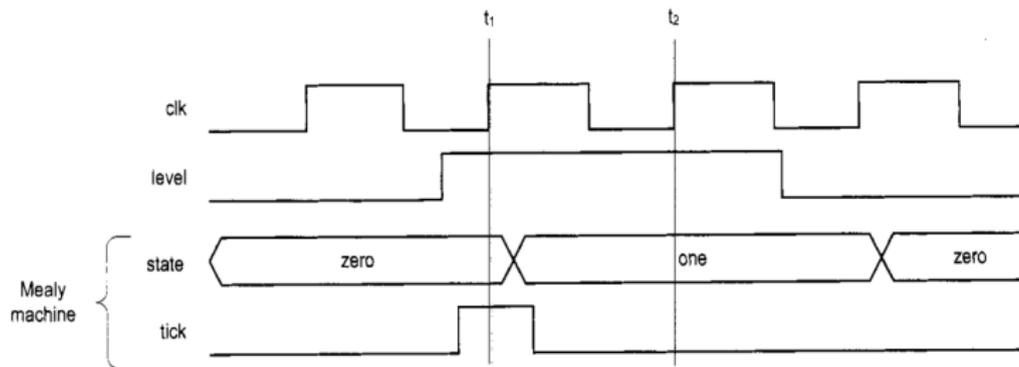
3. Lógica de Salida

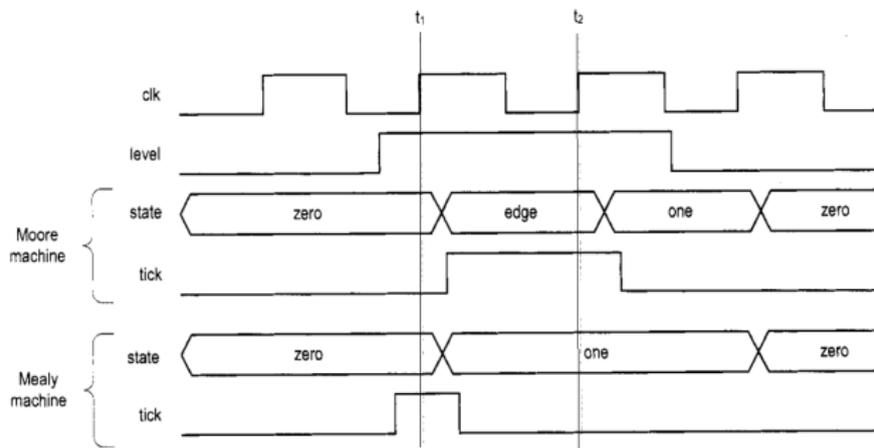


```

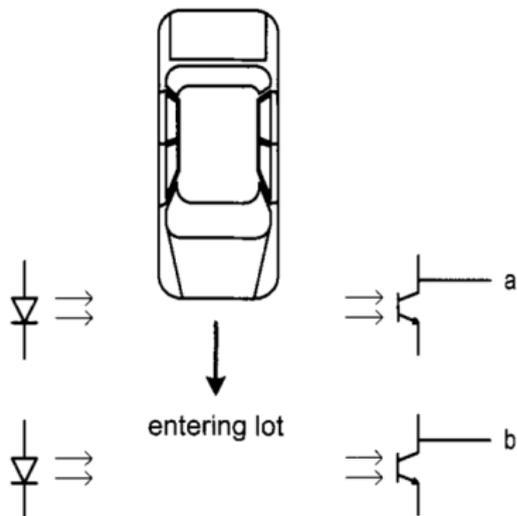
-- output logic: mealy
process(state req, level)
begin
  tick <= '0'; -- default
  case state req is
    when zero =>
      if level = '1' then
        tick <= '1';
      end if;
    -- default
    when others =>
      tick <= '0';
  end case;
end process;

```

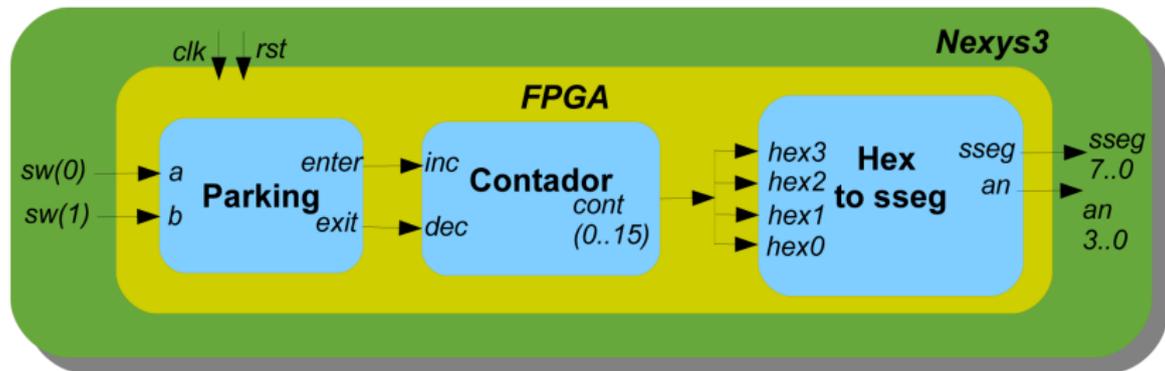




- Ambos producen un tick corto.
- Mealy: menos estados y responde mas rápido, pero responde a glitches en la entrada y no está definido el ancho del pulso. SI está definido que alrededor del rising edge del clk, va a estar estable en 1.
- Moore: mas estados y responde 1 clk mas tarde. Asegura tick = 1 durante un clk.
- Depende de aplicación. Fundamental especificar el funcionamiento del módulo.



- Diseñar FSM/ASM con dos entradas (a,b) y dos salidas (enter, exit). Las salidas se ponen en 1 un clk después de la entrada y salida de un auto respectivamente.
- Derivar el código HDL para la FSM.
- Diseñar un contador con dos señales de control inc y dec que incrementa y decrementa el contador respectivamente. Derivar el código HDL.
- Combinar el contador, la FSM y el código para led-multiplexing. Usar los switches para representar el funcionamiento de los sensores (a y b); y mostrar el contador de autos en el 7-seg.



- Combinacional:
 - Dígito hexadecimal a 7-segmentos.
 - Sumador de signo y magnitud.
- Secuenciales Regulares:
 - Multiplexación en tiempo para mostrar 4 dígitos hexadecimales en el 7-segmentos.
 - Pulse Width Modulator (Ej 4.7.1 del Chino)
- FSM:
 - Estacionamiento completo.