Circuitos Secuenciales en FPGA
Parte I: Secuenciales Regulares

Introducción a la Microfabricación y las FPGA

Instituto Balseiro

11 de Agosto 2014

La clase pasada

- Introducción a VHDL.
  - VHDL para síntesis: nuestro primer programa a nivel de Gates.
  - VHDL para simulación: nuestro primer testbench.
- Herramientas:
  - *ISE*: Crear proyecto, fuente, ucf (user constraint file) y todo el proceso de síntesis hasta generar el archivo de programa (bitstream).
  - *ISim*: Usando el ISE, crear un testbench y simularlo con el ISE Simulator (ISim).
  - *Digilent Adept*: Configurar la FPGA.
- Circuitos combinacionales a nivel RTL. Nuevos tipos de datos, nuevos operadores, ruteo de señales.
- Empezamos con Procesos.

Hoy veremos

- Lo que quedó de **Circuitos combinacionales**. Ejercicios.
- **Circuitos secuenciales**
  - Circuitos con memoria: la salida es función de la entrada y del estado interno del circuito.
  - Hoy veremos el primer tipo: circuitos secuenciales regulares.

Procesos

- Procesos: es una sentencia concurrente, pero dentro tiene una cantidad de sentencias secuenciales que describen su comportamiento.
- Sirven para simplificar la tarea de diseño. Pero hay que tener cuidado: nunca hay que perder de vista que aún estas "sentencias secuenciales" están sintetizando hardware concurrente.
- Siempre pensar en qué se está sintetizando.

```
process(sensitivity_list)
begin
   sequential statement;
   sequential statement;
   . . .
end process;
```

Error: inferencia de memoria

- El standard VHDL especifica que toda señal *mantiene su valor anterior* si no es asignada.
- En el proceso de síntesis esto infiere un latch.
- Reglas para prevenir la inferencia de memoria no intencionada:
  - Todas las señales de input tienen que estar en la lista de sensitividad.
  - Incluir todos los else-branch en los ifs.
  - Asignar un valor a cada señal en cada branch.

Error: inferencia de memoria

```
process(a)
begin
    if (a > b) then
        gt <= '1';
    elsif (a = b) then
        eq <= '1';
    end if;
end process;
```

Tiene todos los errores

- Señal `b` no está en la lista de sensibilidad.
- `eq` no asignada en primer branch.
- `gt` no asignada en segundo branch.
- El branch `else` no está.

Error: inferencia de memoria corregido

```
process(a,b)
begin
    if (a > b) then
        gt <= '1';
        eq <= '0';
    elsif (a = b) then
        gt <= '0';
        eq <= '1';
    else
        gt <= '0';
        eq <= '0';
    end if;
end process;
```

```
process(a,b)
begin
    gt <= '0';
    eq <= '0';
    if (a > b) then
        gt <= '1';
    elsif (a = b) then
        eq <= '1';
    end if;
end process;
```

Constantes

Para limites de arreglos y expresiones. Sintaxis:

```
constant const_name : data_type := value_expression;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity add_w_carry is
   port(
      a, b: in std_logic_vector(3 downto 0);
      cout: out std_logic;
      sum: out std_logic_vector(3 downto 0)
   );
end add_w_carry;

architecture hard_arch of add_w_carry is
   signal a_ext, b_ext, sum_ext: unsigned(4 downto 0);
begin
   a_ext <= unsigned('0' & a);
   b_ext <= unsigned('0' & b);
   sum_ext <= a_ext + b_ext;
   sum <= std_logic_vector(sum_ext(3 downto 0));
   cout <= sum_ext(4);
end hard_arch;
```

Sumador con constante

```
architecture const_arch of add_w_carry is
    constant N: integer := 4;
    signal a_ext, b_ext, sum_ext: unsigned(N downto 0);
begin
    a_ext <= unsigned('0' & a);
    b_ext <= unsigned('0' & b);
    sum_ext <= a_ext + b_ext;
    sum <= std_logic_vector(sum_ext(N-1 downto 0));
    cout <= sum_ext(N);
end const_arch;
```

Genéricos

Con las constantes, es mas fácil de leer, pero si quiero instanciar un sumador de 5 bits, el de 4 bits no me sirve.
Los `generic` son parámetros de los módulos: me permiten instanciar módulos con la misma estructura, pero distintos tamaños.

```
entity entity_name is
    generic(
        generic_name: data_type := default_values;
        generic_name: data_type := default_values;
        . . .
        generic_name: data_type := default_values
    )
    port(
        port_name: mode data_type;
        . . .
    );
end entity_name;
```

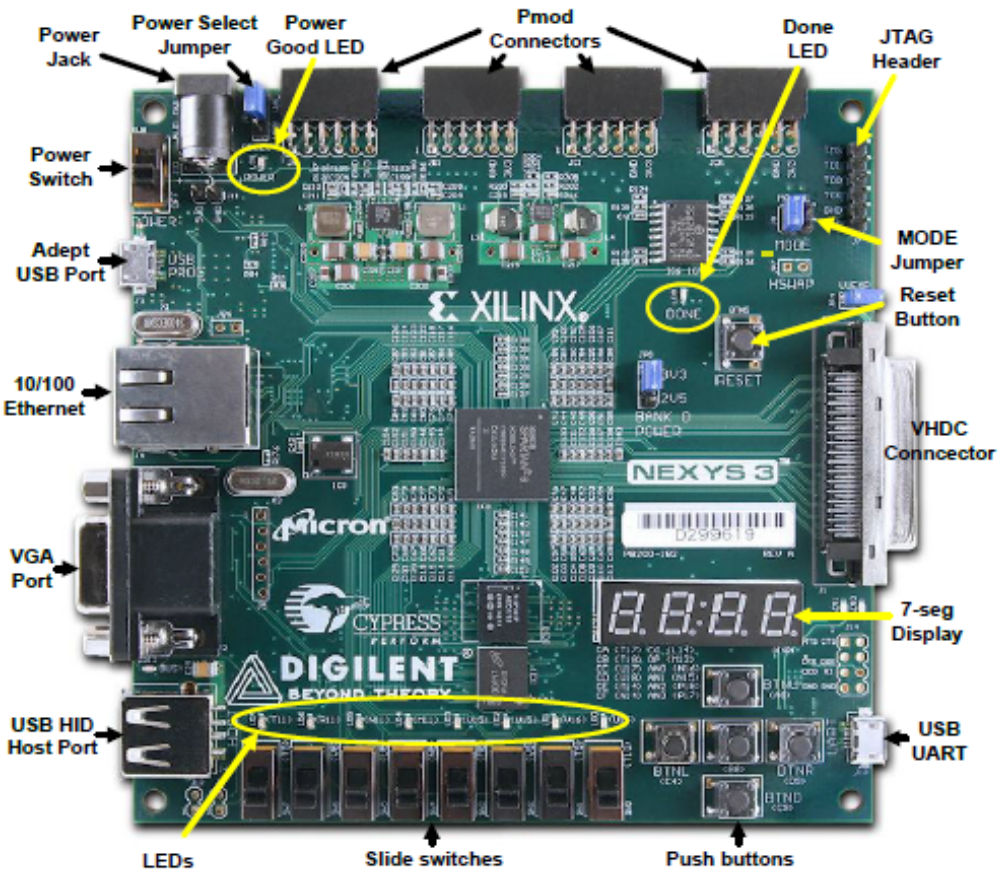
Sumador con Generic

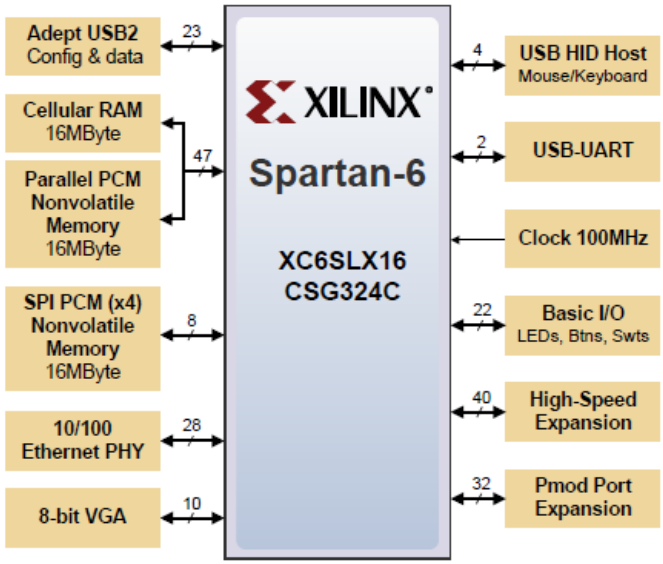
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity gen_add_w_carry is
   generic(N: integer:=4);
   port(
      a, b: in std_logic_vector(N-1 downto 0);
      cout: out std_logic;
      sum: out std_logic_vector(N-1 downto 0)
   );
end gen_add_w_carry;

architecture arch of gen_add_w_carry is
   signal a_ext, b_ext, sum_ext: unsigned(N downto 0);
begin
   a_ext <= unsigned('0' & a);
   b_ext <= unsigned('0' & b);
   sum_ext <= a_ext + b_ext;
   sum <= std_logic_vector(sum_ext(N-1 downto 0));
   cout <= sum_ext(N);
end arch;
```

Sumador con Generic instanciado

```
signal a4, b4, sum4: unsigned(3 downto 0);
signal a8, b8, sum8: unsigned(7 downto 0 );
signal a16, b16, sum16: unsigned(15 downto 0 );
signal c4, c8, c16: std_logic;
....
--instancio sumador de 8 bits
adder-8-unit: work.gen_add_w_carry(arch)
generic map ( N = >8)
port map(a=>a8, b=>b8, cout=>c8, sum=>sum8));
--instancio sumador de 16 bits
adder-16-unit: work.gen_add_w_carry(arch)
generic map (N = > 16)
port map(a=>a16, b=>b16, cout=>c 6 , sum=>sum16));
--instancio sumador de 16 bits
--no hago generic map: se usa el default (4)
adder-4-unit: work.gen_add_w_carry(arch)
port map(a=>a4, b=>b4, cout=>c4, sum=>sum4));
```

Nexys3

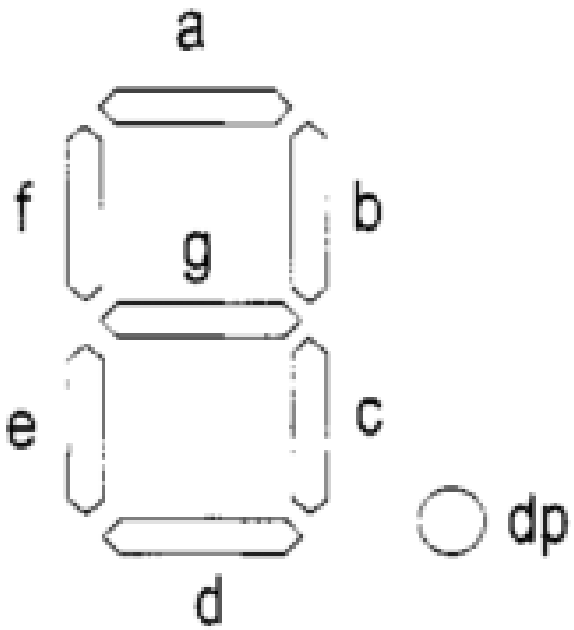

Power Jack
Power Select Jumper
Power Good LED
Pmod Connectors
Done LED
JTAG Header
Power Switch
Adept USB Port
MODE Jumper
Reset Button
10/100 Ethernet
VHDC Conncector
VGA Port
7-seg Display
USB HID Host Port
USB UART
LEDs
Slide switches
Push buttons
XILINX
NEXYS 3
DIGILENT


Adept USB2 Config & data
23
Cellular RAM 16MByte
Parallel PCM Nonvolatile Memory 16MByte
47
SPI PCM (x4) Nonvolatile Memory 16MByte
8
10/100 Ethernet PHY
28
8-bit VGA
10
XILINX
Spartan-6
XC6SLX16 CSG324C
4
USB HID Host Mouse/Keyboard
2
USB-UART
Clock 100MHz
22
Basic I/O LEDs, Btns, Swts
40
High-Speed Expansion
32
Pmod Port Expansion

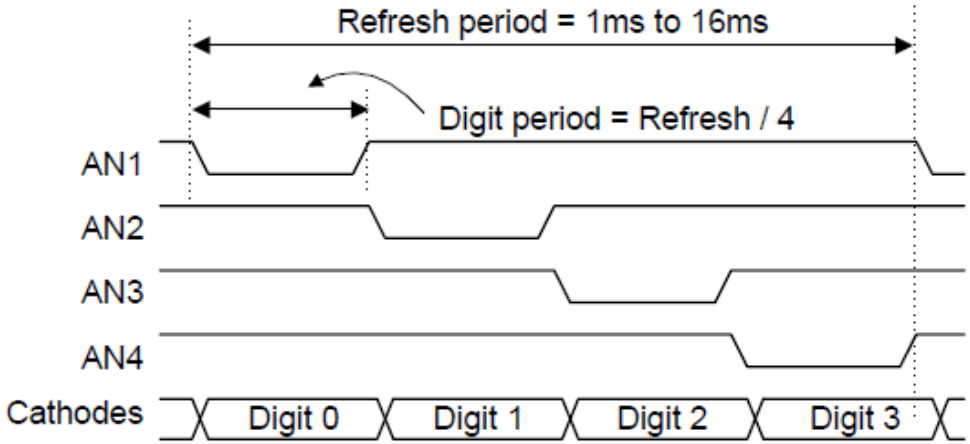
Ejercicios

- Realizar un módulo que convierta números binarios al display de 7 segmentos.
- Realizar un sumador de signo y magnitud de 4 bits (1 bit signo, 3 bits magnitud). Mostrar signo en led, magnitud en 7seg an0.
- Implementar una pequeña calculadora que realice suma, resta y multiplicación de numeros de 4 bits con signo y magnitud. Mapear la selección de la función a los push buttons, la entrada de los operandos a los switches, y mostrar el signo del numero resultado en un led y su magnitud en el 7seg.

7 segmentos



- 15 números (0 a F).
- Cada barrita está prendida si tiene un 0.
- Los 4 7seg están multiplexados: selección an0, an1, an2, an3.

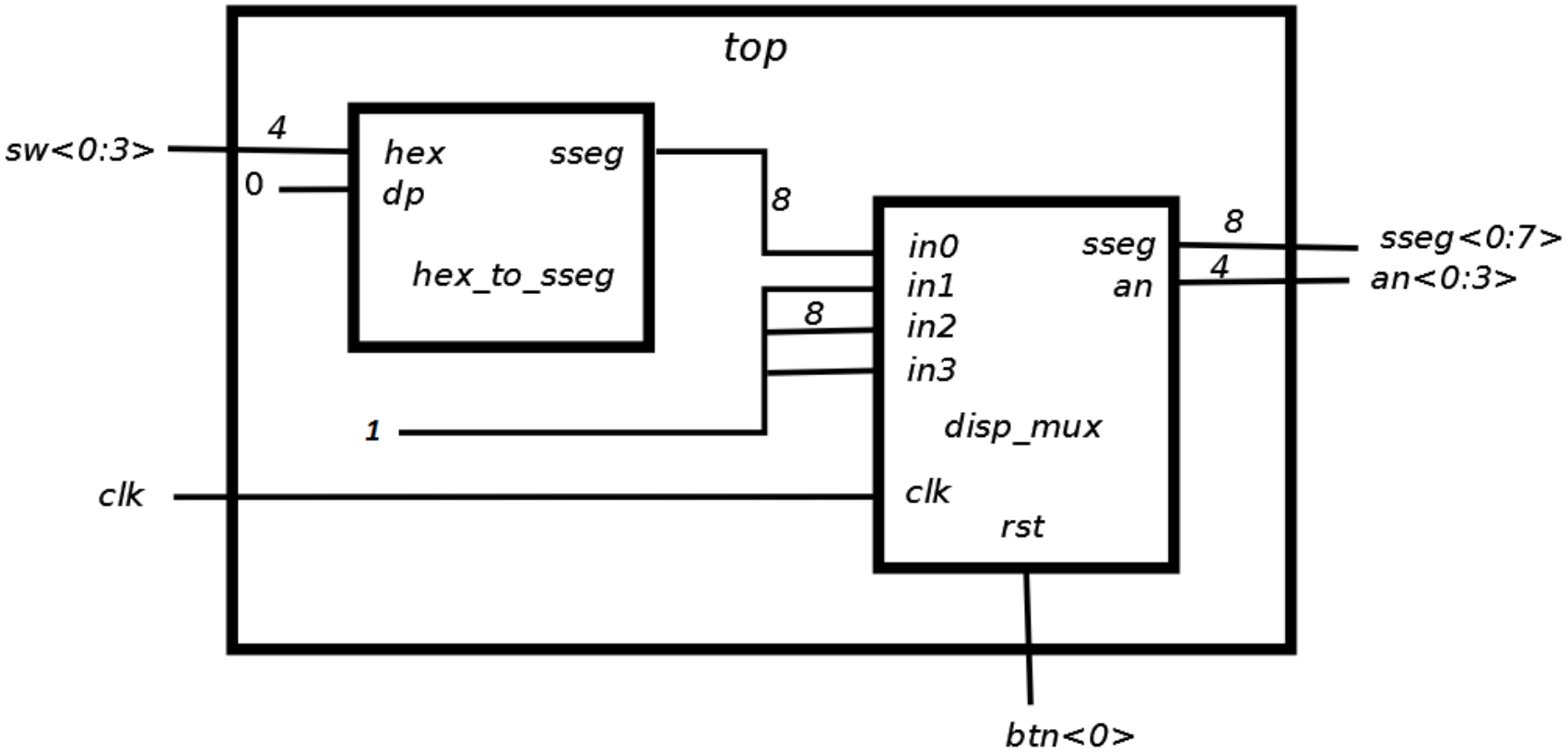


Ejercicios

Para todos los ejercicios:

1. Realizar el diseño. Dibujar modulos, conexiones, ruteo.
2. Codificar en VHDL los módulos.
3. Codificar un testbench cubriendo casos representativos y simular el diseño.
4. Mapear I/O del diseño, implementarlo y configurar la FPGA.

Diseño


top
sw<0:3>
4
0
hex
dp
sseg
hex_to_sseg
8
1
8
in0
in1
in2
in3
sseg
an
disp_mux
clk
rst
8
4
sseg<0:7>
an<0:3>
clk
btn<0>

A codificar

A codificar...

En resumen

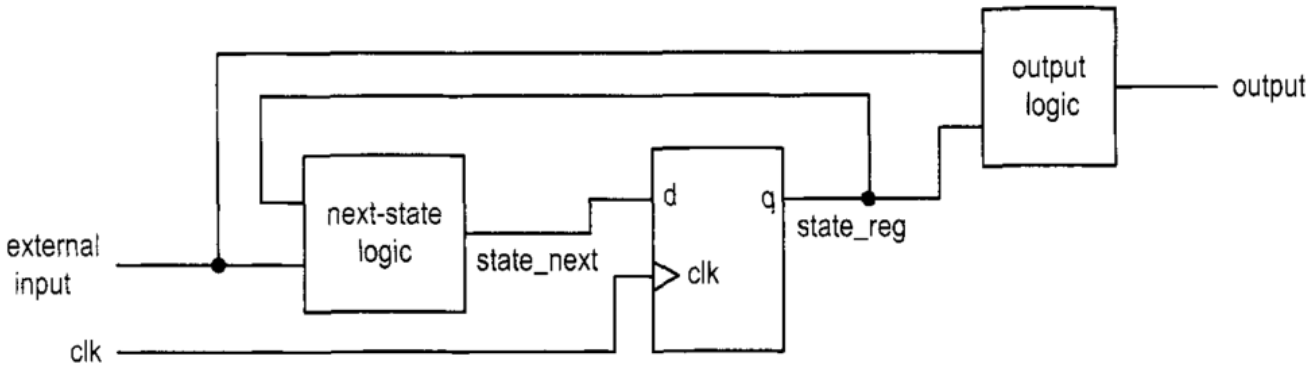
Diseño a nivel de Register Transfer inicial, sólo combinacional.

- Continuando con sentencias concurrentes:
  - Tipos de datos. Operaciones aritméticas y relacionales. Overloading y casting.
  - Ruteo concurrente: conditional signal assig (*when* - ruteo con prioridad) y selected signal assig (*select* - gran multiplexor).
- Agregamos procesos: sentencias secuenciales!
  - Ruteo dentro de procesos: *if* y *case*
  - Reglas para que el proceso infiera un circuito combinacional (i.e., sin memoria).
- Constantes y Genéricos
- Ejemplos usando la Nexys3.

Circuitos secuenciales

- Circuitos con memoria.
- La salida es función de la entrada y del estado interno del circuito.
- Como el estado depende de la secuencia de eventos que hayan sucedido anteriormente, se conocen como circuitos secuenciales.
- Hoy veremos el primer tipo: circuitos secuenciales regulares.

Diseño Síncrono



- *Registro de Estado*: Conjunto de FF controlados por un único reloj.
- *Lógica del Próximo Estado*: circuito combinacional que usa el input y el estado interno para determinar el próximo estado (i.e, el prox valor del registro).
- *Lógica de Salida*: lógica combinacional que genera las salidas.

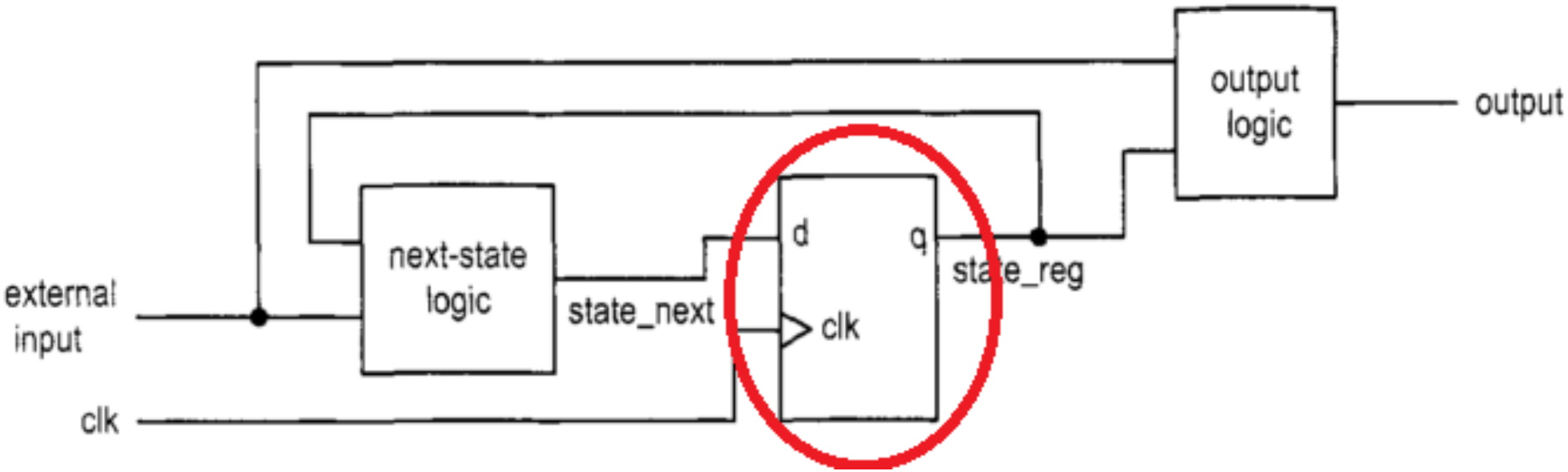
Tipos de circuitos secuenciales

Según qué tan compleja es la lógica del próximo estado.

- *Regulares*: Las transiciones de estados muestran un patrón regular, como un contador. La lógica del próximo estado se construye principalmente con un componente combinacional de RTL "pre-diseñado" como un sumador.
- *Finite State Machine*: Las transiciones entre estados no exhiben un patrón regular. La lógica del próximo estado se construyen con FSMs.
- *Finite State Machine con Datapath*: 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".

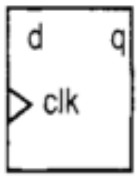
  Se usa para implementar un algoritmo con la metodología RTL, que describe la operación del sistema como una secuencia de transferencia y manipulación de datos entre registros.

Registro de Estado



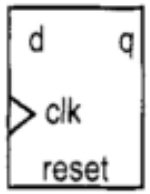
- D Flip Flop.
- Registros
- Banco de Registros.

D Flip Flop



clk	q*
0	q
1	q
↓	d

(a) D FF



reset	clk	q*
1	-	0
0	0	q
0	1	q
0	↓	d

(b) D FF with asynchronous reset

reset	clk	en	q*
1	-	-	0
0	0	-	q
0	1	-	q
0	↓	0	q
0	↓	1	d

(c) D FF with synchronous enable

Código para D FF sin reset

```
library ieee;
use ieee.std_logic_1164.all;
entity d_ff is
    port(
        clk: in std_logic;
        d: in std_logic;
        q: out std_logic
    );
end d_ff;

architecture arch of d_ff is
begin
    process(clk)
    begin
        if (clk'event and clk='1') then
            q <= d;
        end if;
    end process;
end arch;
```

clk	q*
0	q
1	q
↓	d

- Aparece señal del clk.
- No todas las señales del proceso estan en la lista de sensitividad (d no está).
- *atributos* de las señales.

Atributos en las señales

S'delayed(T)	A signal that takes on the same values as S but is delayed by time T.
S'stable(T)	A Boolean signal that is true if there has been no event on S in the time interval T up to the current time, otherwise false.
S'quiet(T)	A Boolean signal that is true if there has been no transaction on S in the time interval T up to the current time, otherwise false.
S'transaction	A signal of type bit that changes value from '0' to '1' or vice versa each time there is a transaction on S.
S'event	True if there is an event on S in the current simulation cycle, false otherwise.
S'active	True if there is a transaction on S in the current simulation cycle, false otherwise.
S'last_event	The time interval since the last event on S.
S'last_active	The time interval since the last transaction on S.
S'last_value	The value of S just before the last event on S.

Rising edge

```
architecture arch of d_ff is
begin
    process(clk)
    begin
        if (rising_edge(clk)) then
            q <= d;
        end if;
    end process;
end arch;
```

- Función definida en `ieee.std_logic_1164`

Código para D FF con reset asíncrono

```
library ieee;
use ieee.std_logic_1164.all;
entity d_ff_reset is
    port(
        clk, reset: in std_logic;
        d: in std_logic;
        q: out std_logic
    );
end d_ff_reset;

architecture arch of d_ff_reset is
begin
    process(clk,reset)
    begin
        if (reset='1') then
            q <='0';
        elsif (clk'event and clk='1') then
            q <= d;
        end if;
    end process;
end arch;
```

reset	clk	q*
1	-	0
0	0	q
0	1	q
0	↓	d

- Usar reset asíncrono va en contra del diseño síncrono.
- Se usa para inicialización: todos los FF deben tener el mismo reset.
- El reset está en la lista de sensibilidad y se chequea antes que el clk, tiene prioridad.

Código para D FF con enable síncrono

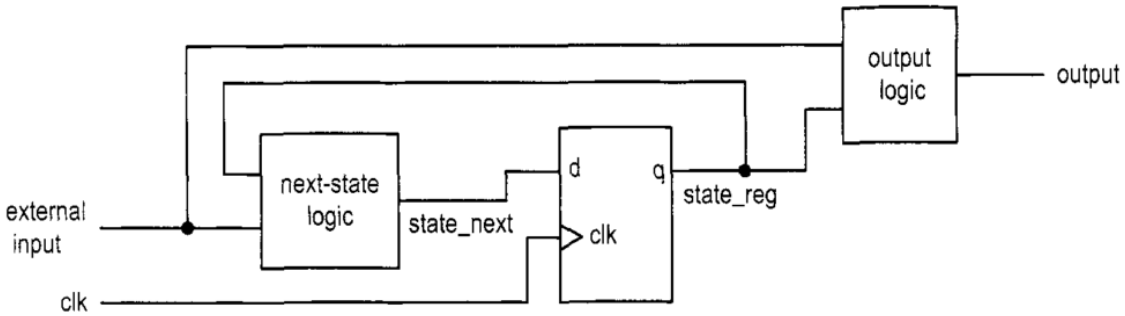
```
library ieee;
use ieee.std_logic_1164.all;
entity d_ff_en is
    port(
        clk, reset: in std_logic;
        en: in std_logic;
        d: in std_logic;
        q: out std_logic
    );
end d_ff_en;

architecture arch of d_ff_en is
begin
    process(clk,reset)
    begin
        if (reset='1') then
            q <='0';
        elsif (clk'event and clk='1') then
            if (en='1') then
                q <= d;
            end if;
        end if;
    end process;
end arch;
```

reset	clk	en	q*
1	-	-	0
0	0	-	q
0	1	-	q
0	↓	0	q
0	↓	1	d

- Mantener sincronismo entre un sistema mas rápido y uno mas lento.

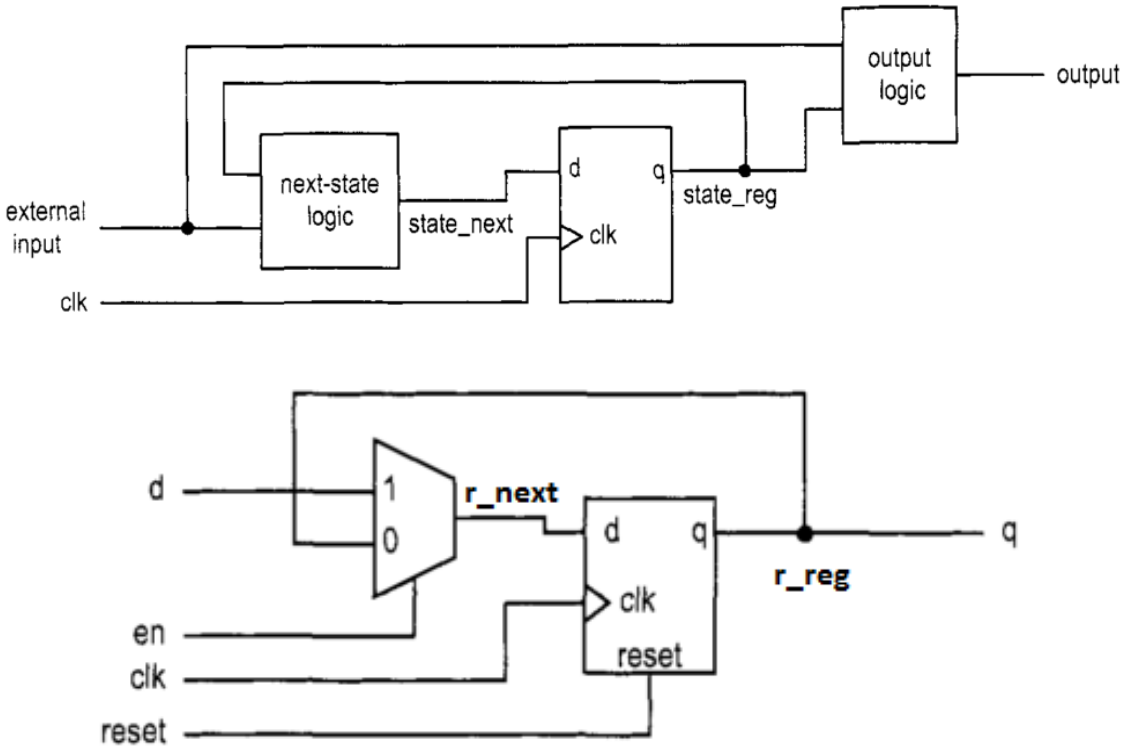
D FF con enable: diseño síncrono

- Como la señal de enable es síncrona, este circuito se puede construir de un D FF normal y una lógica de próximo estado sencilla

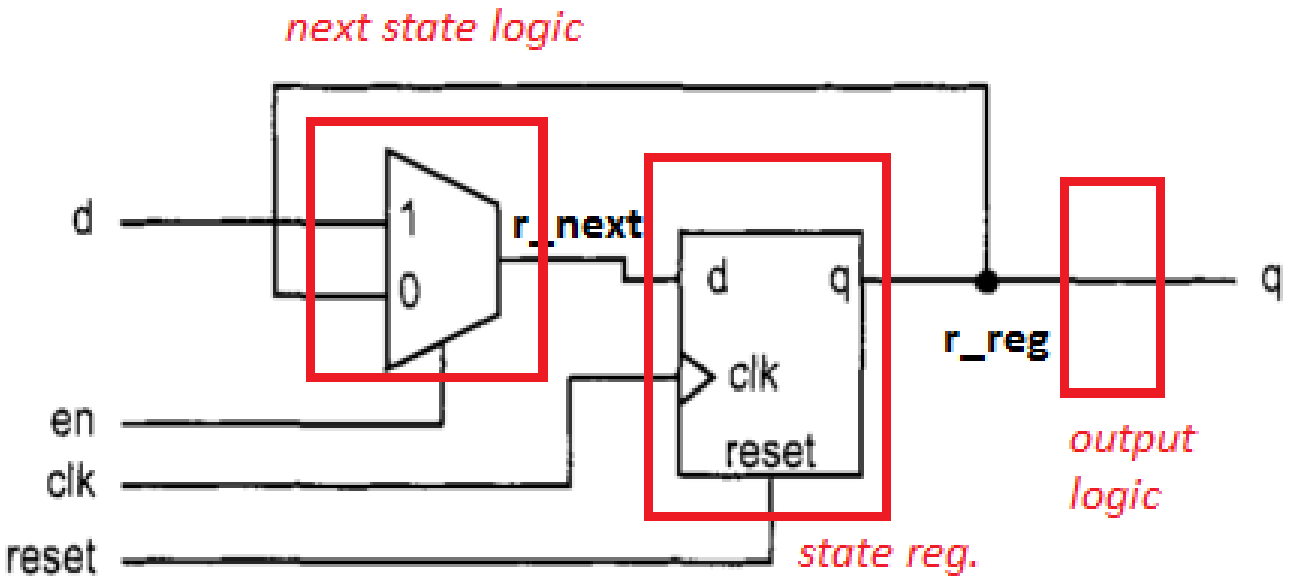


D FF con enable: diseño síncrono

- Como la señal de enable es síncrona, este circuito se puede construir de un D FF normal y una lógica de próximo estado sencilla



Diseño síncrono: ordenando el código



Código para D FF con diseño síncrono

```
architecture dis_sinc_arch of d_ff_en is
   signal r_reg, r_next: std_logic;
begin
   -- D FF : Estado
   process(clk,reset)
   begin
      if (reset='1') then
         r_reg <='0';
      elsif (clk'event and clk='1') then
         r_reg <= r_next;
      end if;
   end process;

   -- lógica del prox estado
   r_next <= d when en ='1' else
             r_reg;

   -- lógica de salida
   q <= r_reg;
end dis_sinc_arch;
```

Registro

```
library ieee;
use ieee.std_logic_1164.all;

entity reg_reset is
    port(
        clk, reset: in std_logic;
        d: in std_logic_vector(7 downto 0);
        q: out std_logic_vector(7 downto 0)
    );
end reg_reset;

architecture arch of reg_reset is
begin
    process(clk,reset)
    begin
        if (reset='1') then
            q <=(others=>'0');
        elsif (clk'event and clk='1') then
            q <= d;
        end if;
    end process;
end arch;
```

Banco de registros - entity

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity reg_file is
    generic(
        B: integer:=8; -- número de bits
        W: integer:=2  -- número de bits de dirección
    );
    port(
        clk, reset: in std_logic;
        wr_en: in std_logic;
        w_addr, r_addr: in std_logic_vector (W-1 downto 0);
        w_data: in std_logic_vector (B-1 downto 0);
        r_data: out std_logic_vector (B-1 downto 0)
    );
end reg_file;
```

Banco de registros - Arquitectura

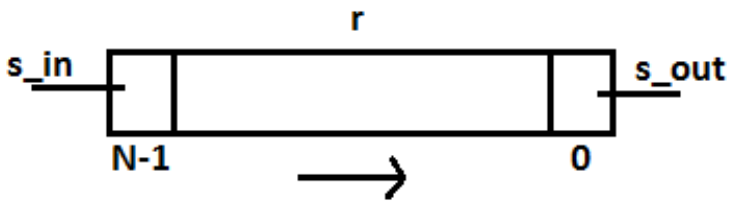
```
architecture arch of reg_file is
   type reg_file_type is array (2**W-1 downto 0) of
        std_logic_vector(B-1 downto 0);
   signal array_reg: reg_file_type;
begin
   process(clk,reset)
   begin
      if (reset='1') then
         array_reg <= (others=>(others=>'0'));
      elsif (clk'event and clk='1') then
         if wr_en='1' then
            array_reg(to_integer(unsigned(w_addr))) <= w_data;
         end if;
      end if;
   end process;

   -- puerto de lectura
   r_data <= array_reg(to_integer(unsigned(r_addr)));

end arch;
```

Banco de registros - Arquitectura

- ¿Qué sintetiza este código? Muchos Slice Flip Flops y Luts como multiplexores de salida!
- La Nexys3 tiene elementos de hardware mejor preparados para hacer de bancos de registros/memoria:
  - Block RAM.
  - Distributed RAM - LUTs usadas como RAM
- ... Pero hay que saber qué pattern de código escribir para que el XST entienda y sintetice uno de estos bancos... Lo veremos mas adelante...

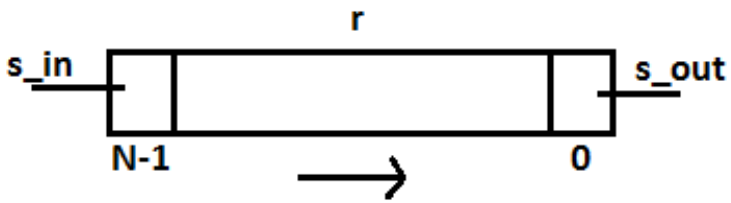
Free running shift register

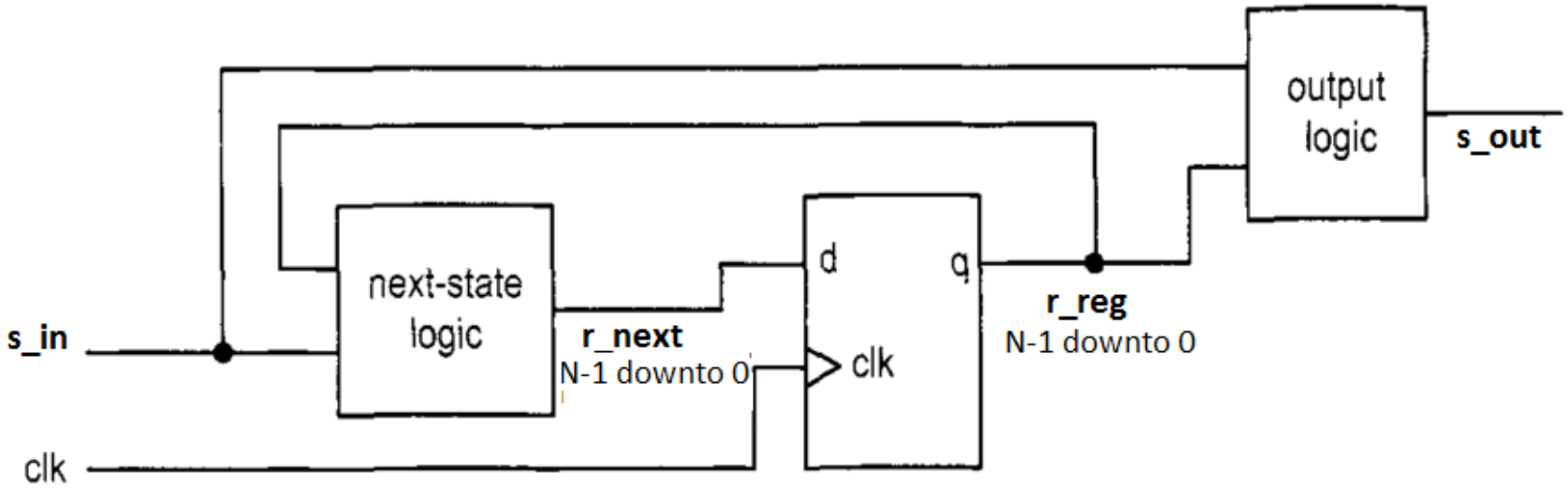
```
library ieee;
use ieee.std_logic_1164.all;

entity free_run_shift_reg is
    generic(N: integer := 8);
    port(
        clk, reset: in std_logic;
        s_in: in std_logic;
        s_out: out std_logic
    );
end free_run_shift_reg;
```



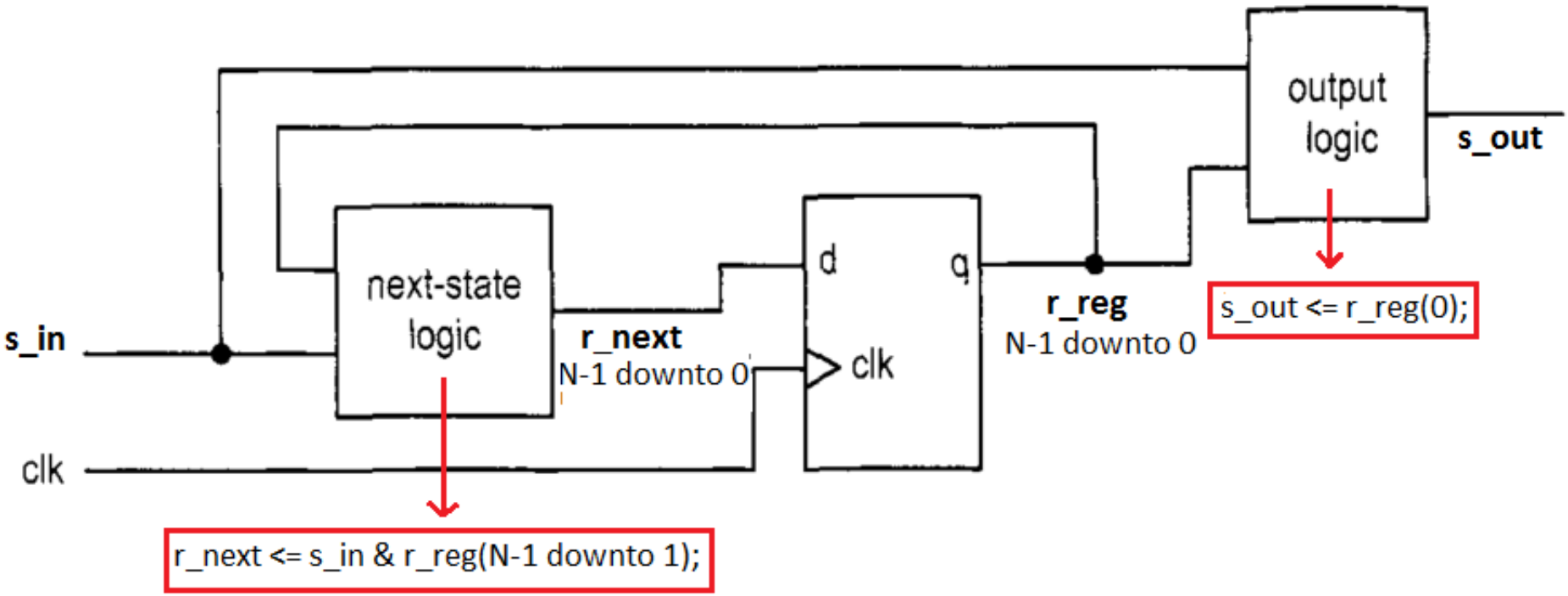
Free running shift register

```
library ieee;
use ieee.std_logic_1164.all;

entity free_run_shift_reg is
   generic(N: integer := 8);
   port(
      clk, reset: in std_logic;
      s_in: in std_logic;
      s_out: out std_logic
   );
end free_run_shift_reg;
```





Free running shift register - con codigo



Free running shift register: código

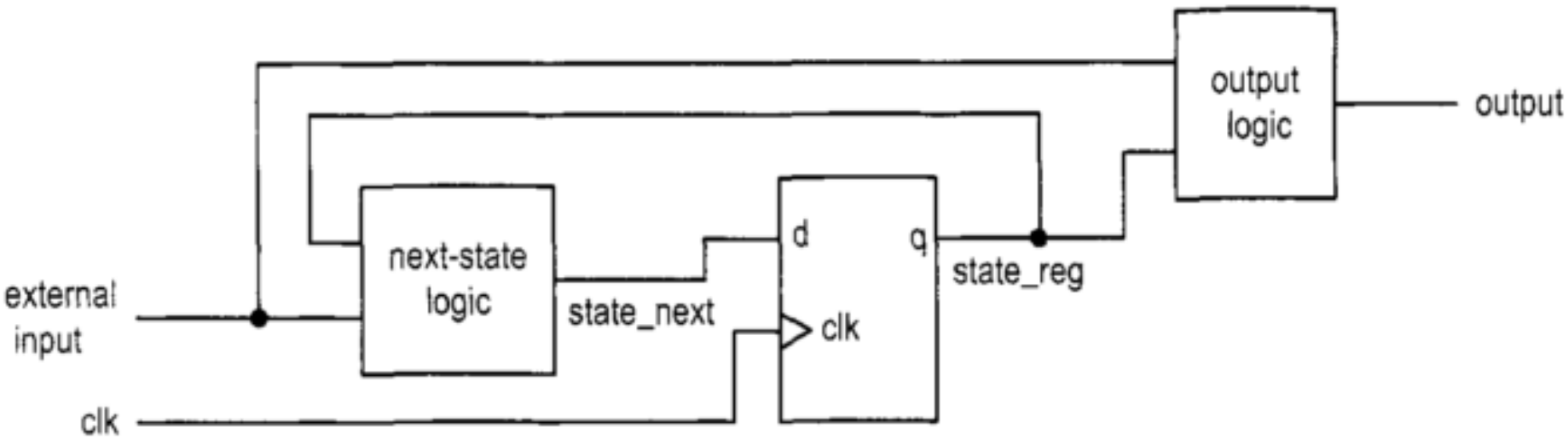
```
architecture arch of free_run_shift_reg is
    signal r_reg: std_logic_vector(N-1 downto 0);
    signal r_next: std_logic_vector(N-1 downto 0);
begin
    -- registro
    process(clk,reset)
    begin
        if (reset='1') then
            r_reg <= (others=>'0');
        elsif (clk'event and clk='1')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

    -- lógica de prox estado (shift a derecha 1 bit)
    r_next <= s_in & r_reg(N-1 downto 1);

    -- lógica de salida (bit 0 del registro)
    s_out <= r_reg(0);
end arch;
```

Contador Binario Universal

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;
```

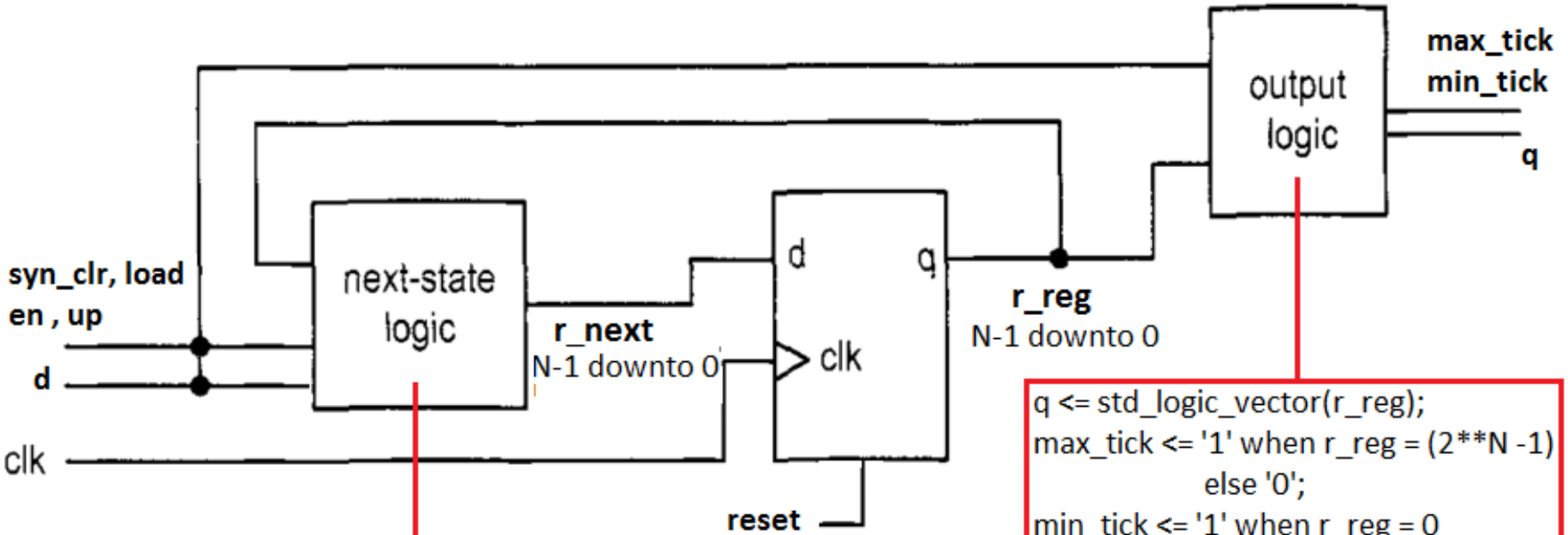
Contador Binario Universal: diseño sincrónico



Contador Binario Universal: a la máquina!

A la máquina

Contador Binario Universal



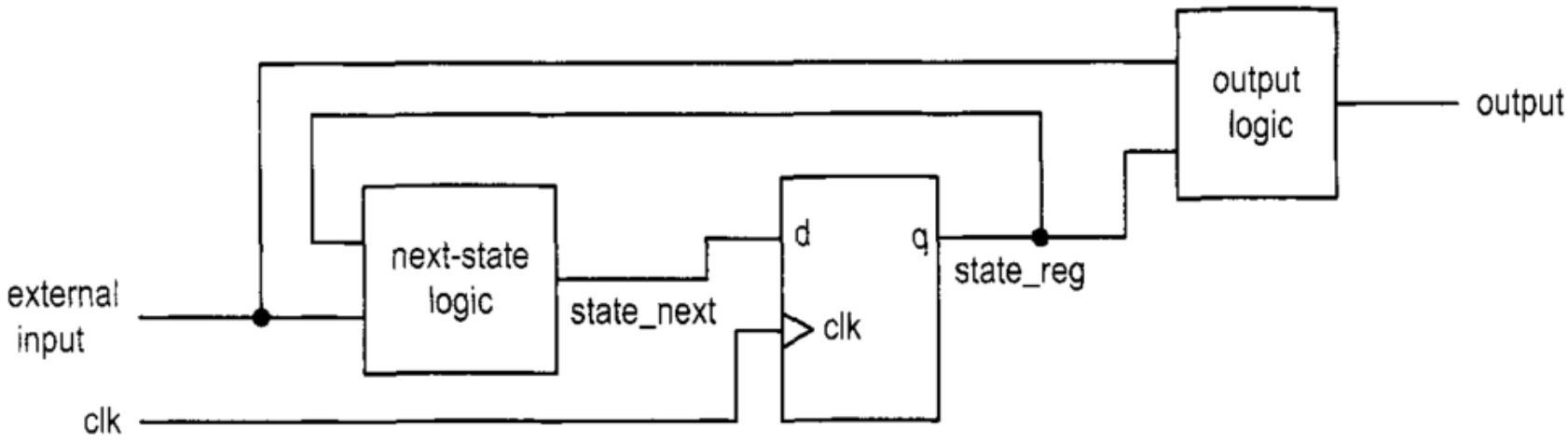

```
r_next <= (others => '0') when syn_clr = '1' else
          unsigned(d)     when load = '1' else
          r_reg + 1       when en = '1' and up = '1' else
          r_reg - 1       when en = '1' and up = '0' else
          r_reg;
```

```
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';
```

Contador Binario Universal: código

```
architecture arch of univ_bin_counter is
   signal r_reg: unsigned(N-1 downto 0);
   signal r_next: unsigned(N-1 downto 0);
begin
   -- Estado: registro
   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;

   -- lógica del prox estado
   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;

   -- lógica de salida
   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';
end arch;
```

IMPORTANTE: Metodo de Diseño



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

0. Definición de Entity

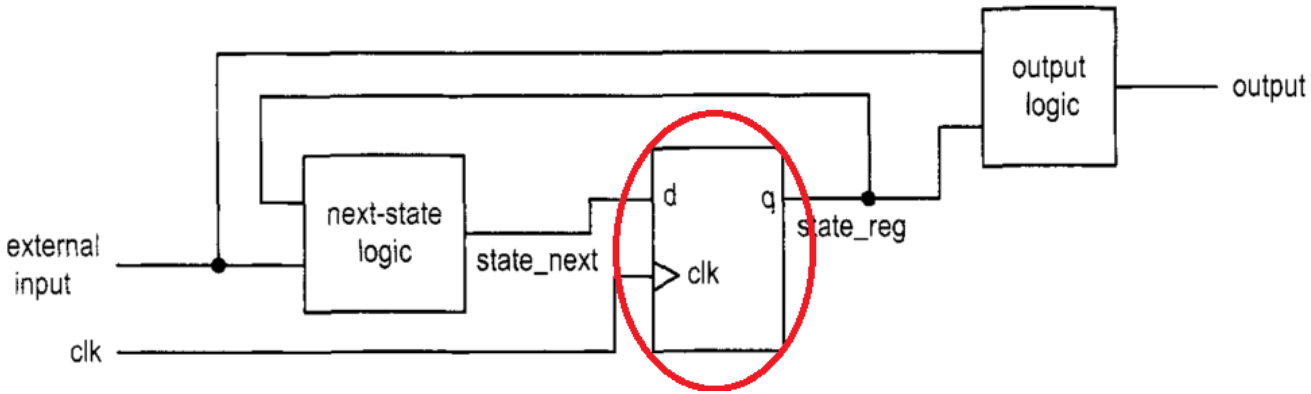
Antes de empezar: identificar las *entradas y salidas* que tiene el circuito, y sus *parámetros*.

- En diseño sincrónico, 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.

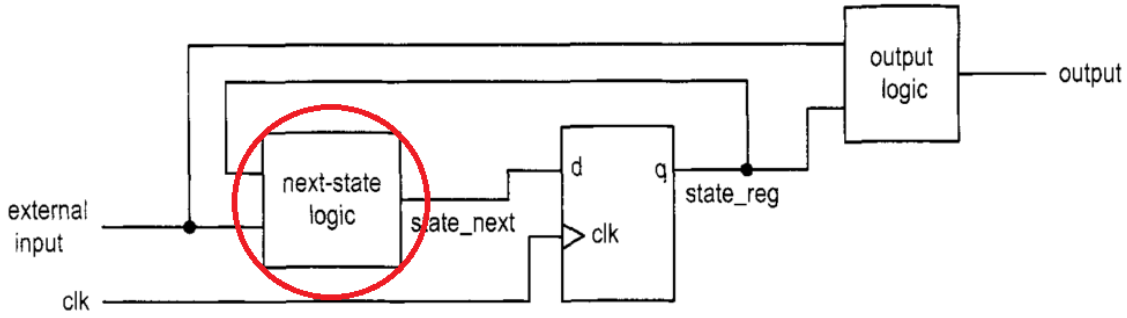
- Por cada señal a recordar, definir dos señales `_reg` y `_next`. Ej: `r_reg` y `r_next`.
- El código de esta parte es bien claro:

```
architecture arch of univ_bin_counter is
   signal r_reg: unsigned(N-1 downto 0);
   signal r_next: unsigned(N-1 downto 0);
begin
   -- Estado: registro
   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

Seugndo: 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.



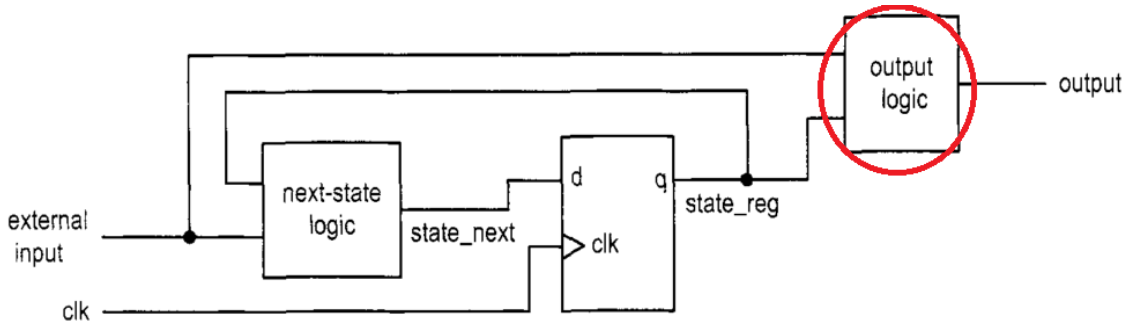
syn_clr	load	en	up	q*	**Operation**
1	–	–	–	00 · · · 00	synchronous clear
0	1	–	–	d	parallel load
0	0	1	1	q+1	count up
0	0	1	0	q−1	count down
0	0	0	–	q	pause

```
-- lógica del prox estado
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.

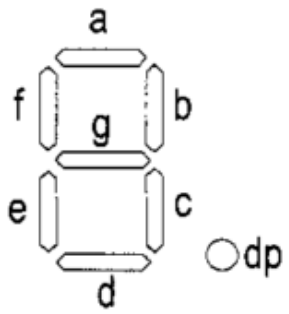
- Calcular las salidas en *este reloj* (i.e, q) en función del *valor actual* `r_reg` y las señales de entrada.
- El código puede ser con sentencias concurrentes o dentro de un process.
- Si está en un process, la lista de sensibilidad incluye a los `***_reg` y a las entradas.

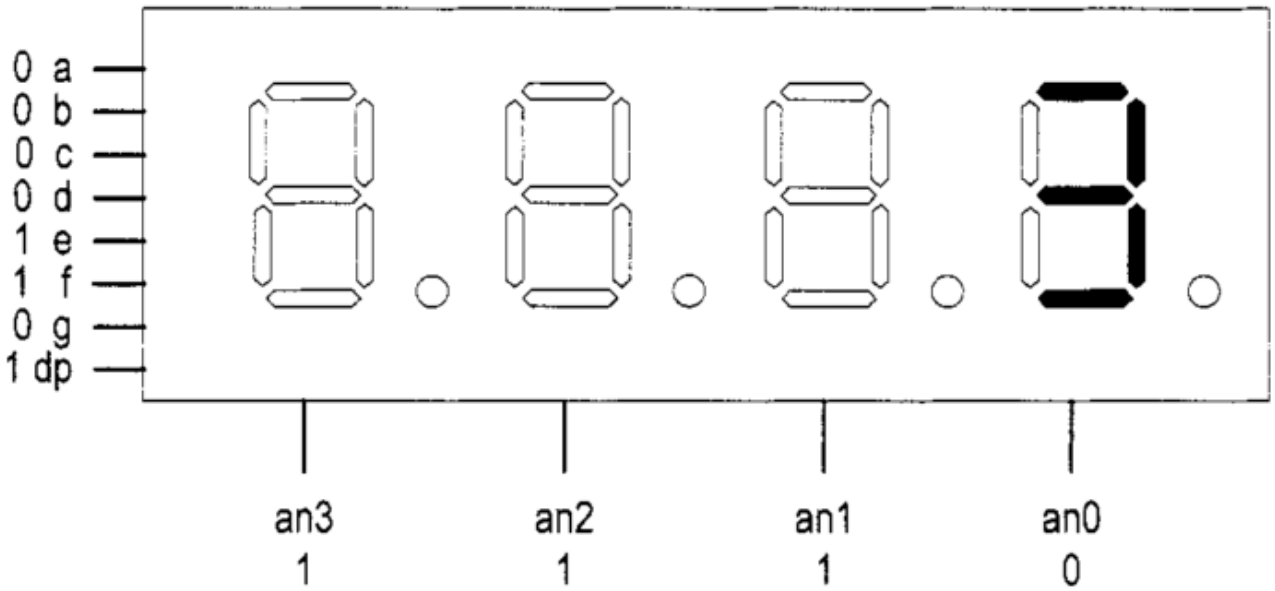



```
-- lógica de salida
q <= std_logic_vector(r_reg);
max_tick <= '1' when r_reg=(2**N-1) else '0';
min_tick <= '1' when r_reg=0 else '0';
```

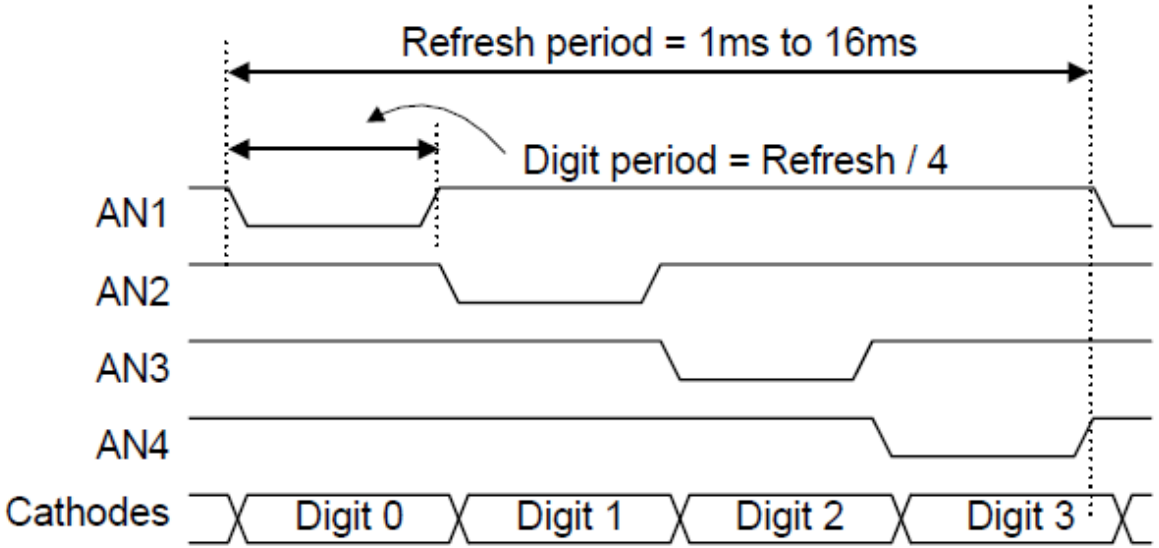
Testbench

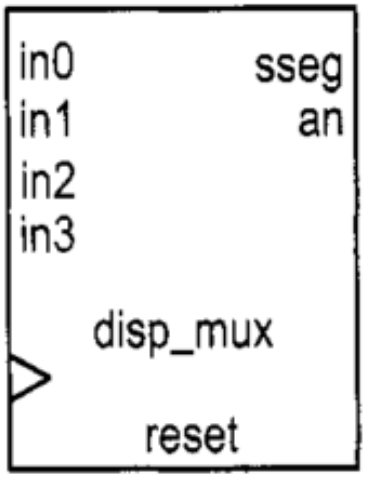
Display multiplexing



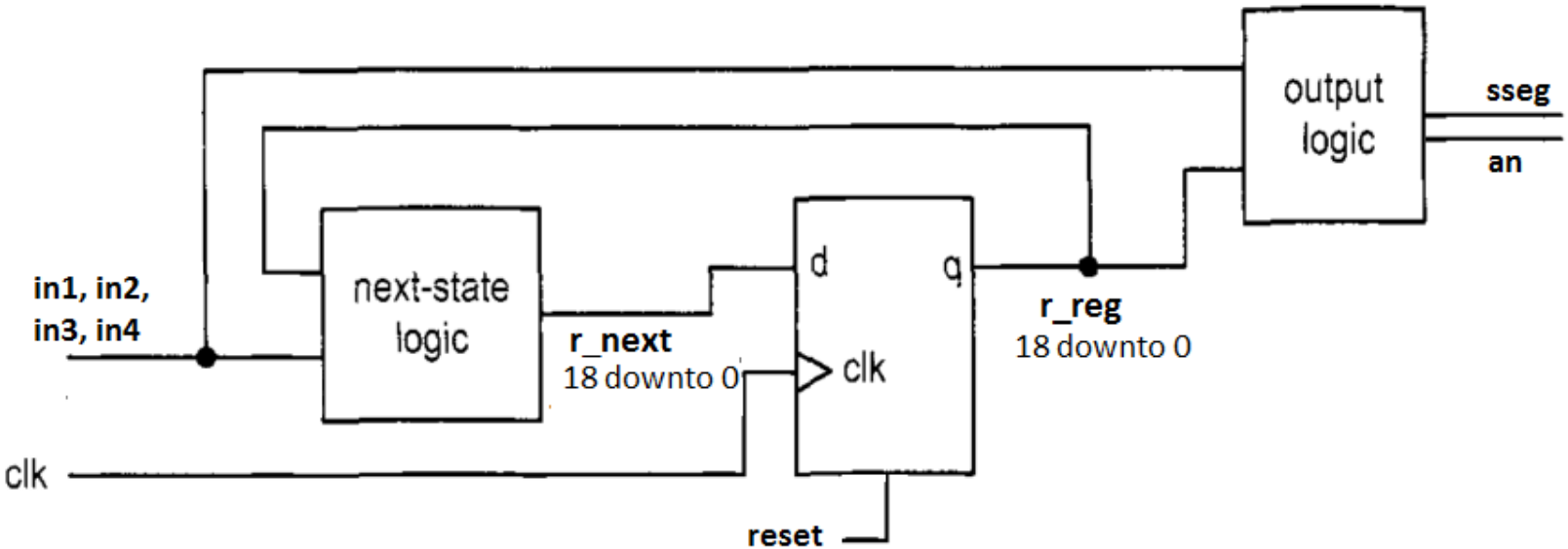


Display multiplexing: tiempo





Display Multiplexing Diagrama



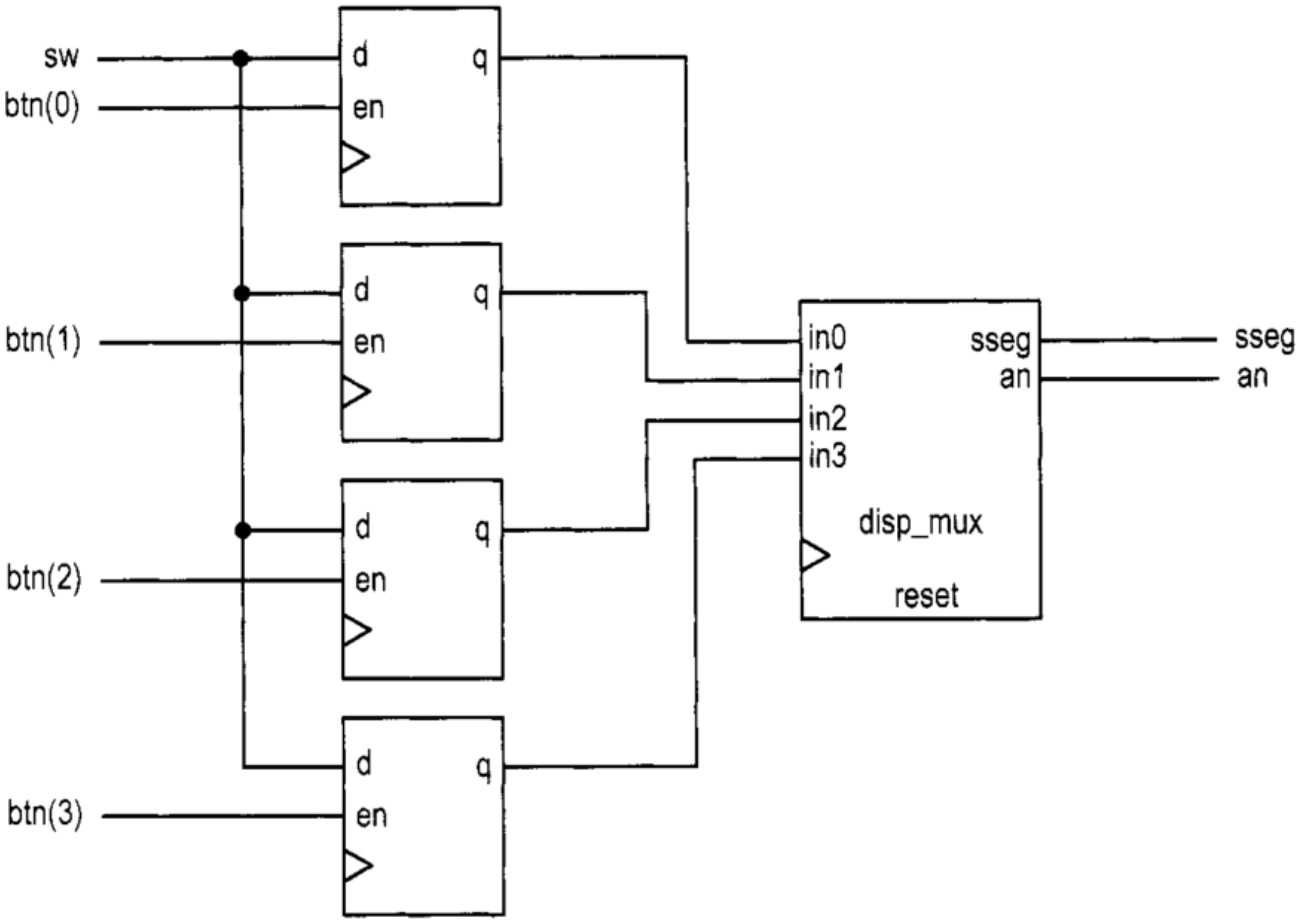
Display multiplexing entity code

```
entity disp_mux is
    port(
        clk, reset: in std_logic;
        in3, in2, in1, in0: in std_logic_vector(7 downto 0);
        an: out std_logic_vector(3 downto 0);
        sseg: out std_logic_vector(7 downto 0)
    );
end disp_mux ;
```

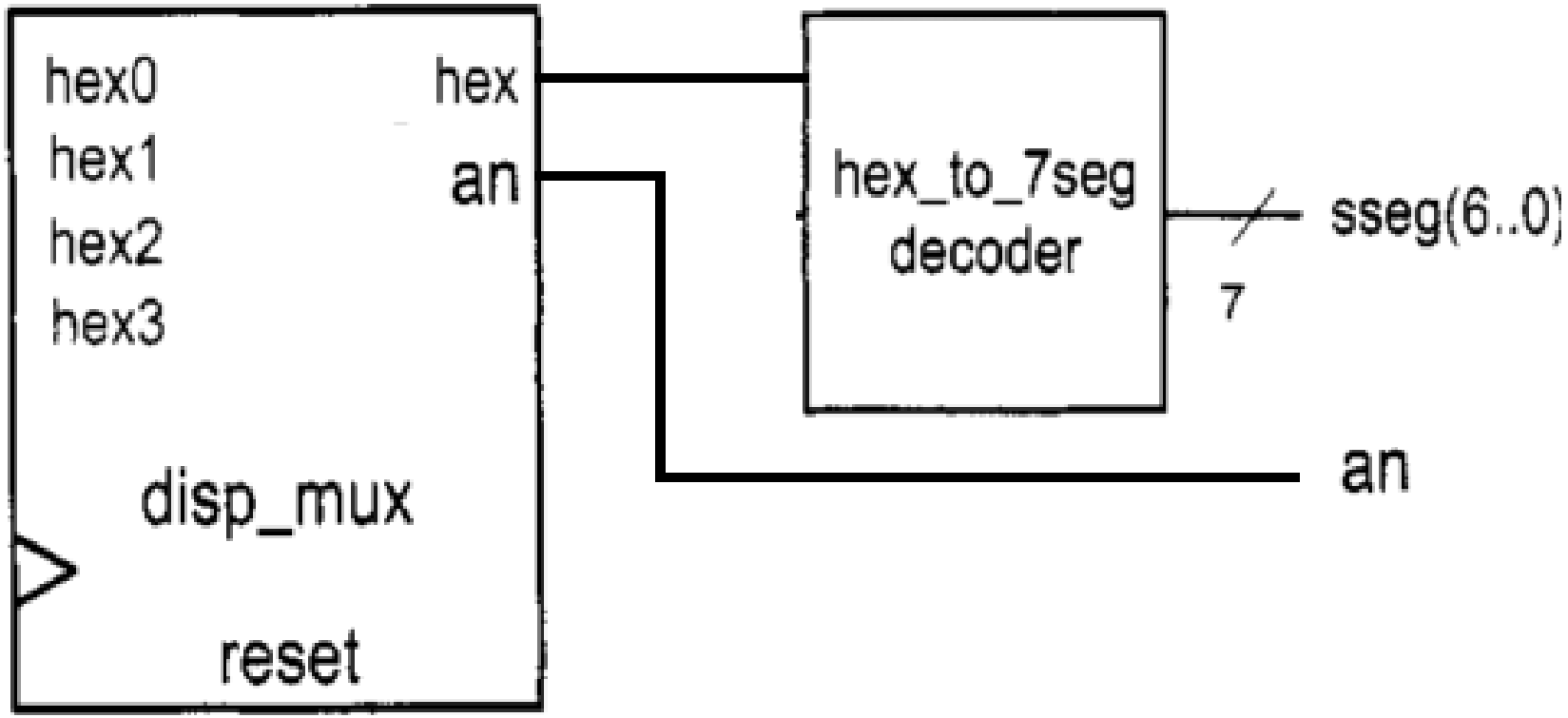
Display multiplexing: a codificar!

A codificar!

Display multiplexing: testing



Display multiplexing tomando inputs hexadecimales



En resumen

- 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 pre-diseñados.