Código Secuencial

Código Secuencial

- El código concurrente está destinado para el diseño de circuitos combinacionales.
- Por su parte, el código secuencial puede ser usado indistintamente para el diseño de circuitos combinacionales o circuitos secuenciales.
- En VHDL hay tres tipos de código secuencial: PROCESS, FUNCTION y PROCEDURE (los últimos dos son llamados subprogramas).
- PROCESS esta destinado a la ARCHITECTURE. Mientras que FUCTION y PROCEDURE los están para las bibliotecas.

Código Secuencial

- ✓ Las sentencias secuenciales en VHDL son IF, WAIT, LOOP Y CASE. Estas instrucciones únicamente se pueden declarar dentro del cuerpo de un código secuencial (PROCESS).

Un punto importante cuando se trabaja con código secuencial es entender la diferencias entre una SIGNAL y una VARIABLE.

Propiedades de SIGNAL

- ✓ Una SIGNAL solo puede ser declarada fuera del código secuencial (PROCESS).
- ✓ Una SIGNAL que se usa dentro de un código secuencial no se actualiza inmediatamente: el valor de SIGNAL se actualiza hasta que se concluye el código secuencial (PROCESS).
- ✓ Una asignación de SIGNAL en función de otra señal (por ejemplo el reloj) puede causar la inferencia de un registro.
- ✓ A SIGNAL sólo se le permite una asignación. Es decir, SIGNAL no permite múltiples asignaciones.

Propiedades de VARIABLE

- ✓ Una VARIABLE únicamente puede declararse y usarse dentro de un PROCESS o subprograma.
- ✓ Un variable se actualiza inmediatamente. Por lo tanto, el nuevo valor puede ser usado en la siguiente línea de código).
- ✓ VARIABLE permite múltiples asignaciones.
- ✓ Una asignación de VARIABLE en función de otra señal (por ejemplo el reloj) puede causar la inferencia de un registro.

PROCESS

Es una sección secuencial de código VHDL, que se localiza en el cuerpo de la ARCHITECTURE. En su interior (PROCESS) únicamente se permiten sentencias secuenciales (IF, WAIT, LOOP, CASE). Su sintaxis es la siguiente

```
[label:] PROCESS [(sensitivity_list)] [IS]
        [declarative_part]
        BEGIN
                sequential_statements_part
END PROCESS [label];
```

La parte declarativa de PROCESS puede contener: subprogramas, tipos, constantes, VARIABLE, ALIAS, entre otros.

PROCESS

- La etiqueta (label) es opcional, su propósito es mejorar la legibilidad en códigos largos.
- La lista de sensibilidad (sensitivity_list) es obligatoria y provoca que el PROCESS se ejecute cada vez que uno de los elementos de la lista cambia.

IF (y IF-ELSE)

De las sentencias secuenciales es la más usada. Su sintaxis es

```
[label:] IF conditions THEN
        assignments;
END IF [label];
```

```
[label:] IF conditions THEN
        assignments;
ELSIF conditions THEN
        assignments;
        ...
ELSE
        assignments;
END IF [label];
```

Nota. Por cada IF que se usa debe existir su contraparte END IF. Esto no aplica a ELSIF.

IF (y IF-ELSE)

Ejemplo

```
IF CLK'EVENT and CLK = '1' THEN
   X <= X + 1;
END IF;
```

```
IF (A = '1' AND B ='0') then
   Y <= '0';
ELSE
   Y <= '1';
END IF;
```

```
IF A='1' THEN
   D <= "00";
ELSIF B='1' THEN
   D <= "01";
ELSIF B='1' THEN
   D <= "10";
ELSE B='1' THEN
   D <= "11";
END IF;
```

El Evento de Reloj

Los circuitos secuenciales responden al cambio de la señal de reloj. Es decir, son sensitivos al flanco de subida, la flanco de bajada o ambos flancos. Esto se describe en VHDL de la siguiente manera

Flanco de subida

```
PROCESS(CLK)
BEGIN
   IF CLK'EVENT and CLK = '1' THEN
      -- assigments;
   END IF;
END PROCESS
```

Flanco de bajada

```
PROCESS(CLK)
   BEGIN
   IF CLK'EVENT and CLK = '0' THEN
      -- assigments;
   END IF;
END PROCESS
```

Ambos flancos

```
PROCESS(CLK)
   BEGIN
   IF (CLK'EVENT and CLK = '1') OR (CLK'EVENT and CLK = '0') THEN
      -- assigments;
   END IF;
END PROCESS
```

Reinicio

Por el general los circuitos secuenciales tienen un mecanismo de reinicio, el cual puede ser síncrono o asíncrono con la señal de reloj. En VHDL ambos mecanismos se describen de la siguiente manera (DOUT es la salida del circuito)

Síncrono

```
PROCESS(CLK)
   BEGIN
   IF CLK'EVENT and CLK = '0' THEN
      IF RESET = '1' THEN
         DOUT <= '0';
      ElSIF
         -- DOUT assigments;
         ...
      END IF;
   END IF;
END PROCESS
```

Asíncrono

```
PROCESS(CLK, RESET)
BEGIN
   IF RESET = '1' THEN
      DOUT <= '0';
   ELSIF CLK'EVENT and CLK = '1' THEN
      -- DOUT assigments;
   END IF;
END PROCESS
```

Aspectos Importantes de PROCESS

- ✓ Si el circuito secuencial no tiene elementos asíncronos, entonces la lista de sensibilidad únicamente contiene la señal de reloj. Es decir, PROCESS sólo se ejecuta cuando hay un cambio en la señal de reloj.

- ✓ Si el circuito secuencial tiene elementos asíncronos (una o varias entradas), entonces la lista de sensibilidad debe contener dichos elementos más la señal de reloj. Esto permite que el PROCESS se ejecute cuando hay un cambio de las señales asíncronas, así como de la señal de reloj.

- ✓ PROCESS puede usarse para describir circuitos combinacionales. En este caso la lista de sensibilidad debe contener todas las señales de entrada del circuito.

Recomendaciones

Los circuitos combinacionales descritos con PROCESS deben evitar la inferencia de registros. Para ello, las siguientes recomendaciones deben observarse:

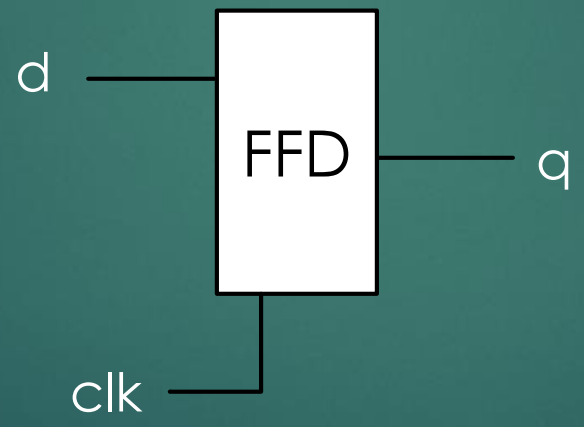
- ✓ La tabla de verdad el circuito debe ser especificada completamente.
- ✓ Todas la señales de entrada del circuito deben aparecer en la lista de sensibilidad de PROCESS.

No cumplir el primer punto causará que el sintetizador envíe un alerta indicando que ciertas señales son usadas dentro de PROCESS pero no están en la lista de sensibilidad.

No tomar en cuenta la segunda recomendación causa que el sintetizador infiera LATCHES para mantener los estados previos del sistema.

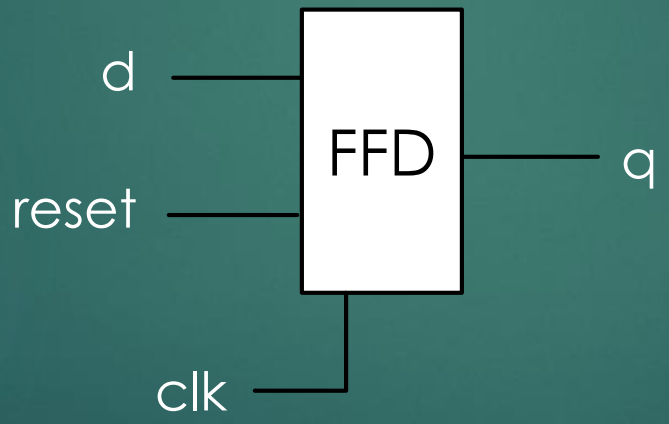
Ejercicios

1. Implemente un flip-flop tipo D, sensitivo al flanco de subida del reloj (ver figura).



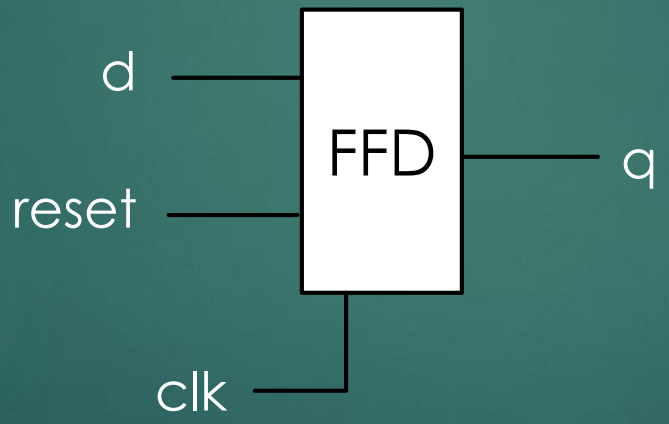
Ejercicios

2. Implemente un flip-flop tipo D, sensitivo al flanco de subida del reloj y con reinicio síncrono (ver figura).



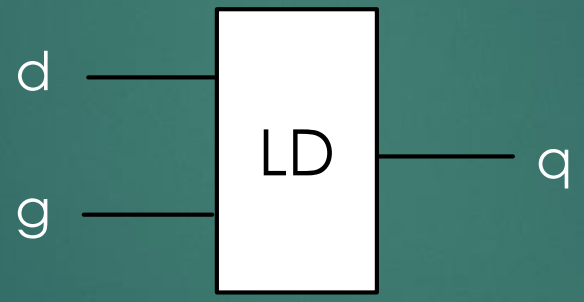
Ejercicios

3. Implemente un flip-flop tipo D, sensitivo al flanco de bajada del reloj y con reinicio asíncrono (ver figura).



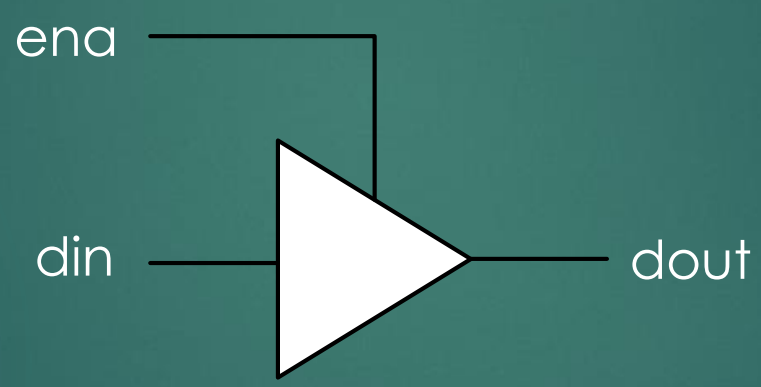
Ejercicios

4. Implemente un LATCH (ver figura).



Ejercicios

5. Implemente un buffer de tercer estado (ver figura).



LOOP

Cómo su nombre lo dice, LOOP es usado cuando una pieza de código debe repartirse varias veces. Hay cinco casos que envuelven la sentencia LOOP:

- ✓ Unconditional LOOP:

```
[label:] LOOP
      sequential_statements
END LOOP [label];
```

Ejemplo

```
LOOP
   IF clk = '1' THEN
      count := count + 1;
   END IF;
END LOOP;
```

LOOP

- LOOP con FOR

```
[label:] FOR identifier IN range LOOP
      sequential_statements
END LOOP [label];
```

Ejemplo

```
FOR i IN 0 TO 5 LOOP
   x(i) <= a(i) AND b(5-i);
   y(0, i) <= c(i);
END LOOP;
```

LOOP

- ✓ LOOP con WHILE:

```
[label:] WHILE condition LOOP
      sequential_statements
END LOOP [label];
```

Ejemplo

```
WHILE (i<10) LOOP
    IF clk'EVENT AND clk='1' THEN
    ...
END LOOP;
```

LOOP

- LOOP con EXIT

```
[loop_label:] [FOR identifier IN range] LOOP
      ...
      [exit_label:] EXIT [loop_label] [WHEN condition];
      ...
END LOOP [loop_label];
```

Ejemplo

```
FOR i IN data'RANGE LOOP
   CASE data(i) IS
      WHEN '0' => count:=count+1;
      WHEN OTHERS => EXIT;
   END CASE;
END LOOP;
```

LOOP

- ✓ LOOP con NEXT:

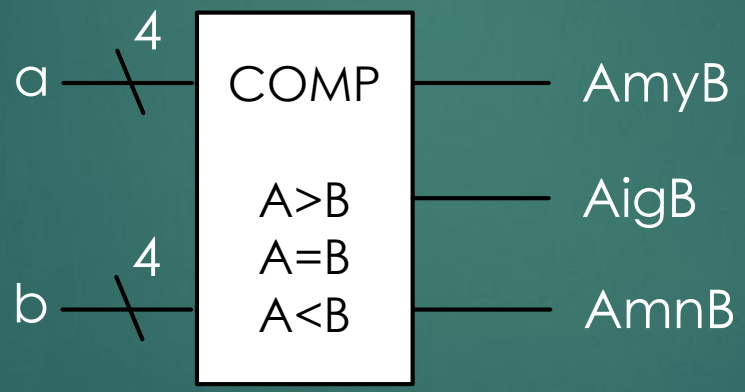
```
[loop_label:] [FOR identifier IN range] LOOP
      ...
      [next_label:] NEXT [loop_label] [WHEN condition];
      ...
END LOOP [loop_label];
```

Ejemplo

```
FOR i IN 0 TO 15 LOOP
   NEXT WHEN i=skip;
   ...
END LOOP;
```

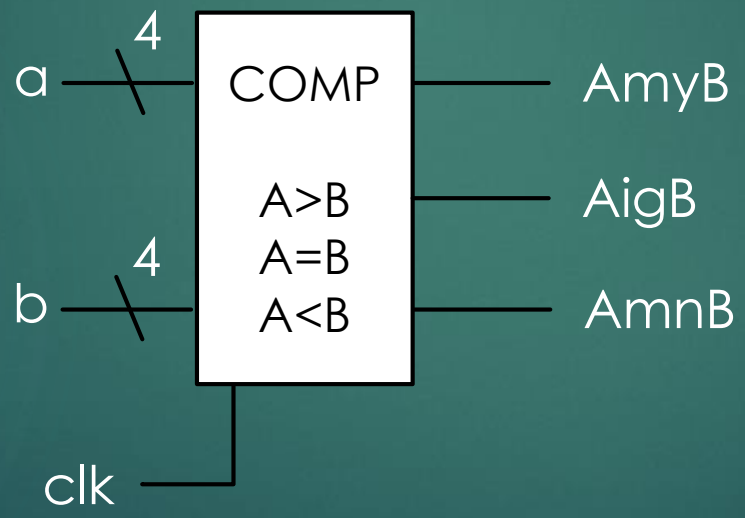
Ejercicios

6. Implemente un comparador combinacional (ver figura).



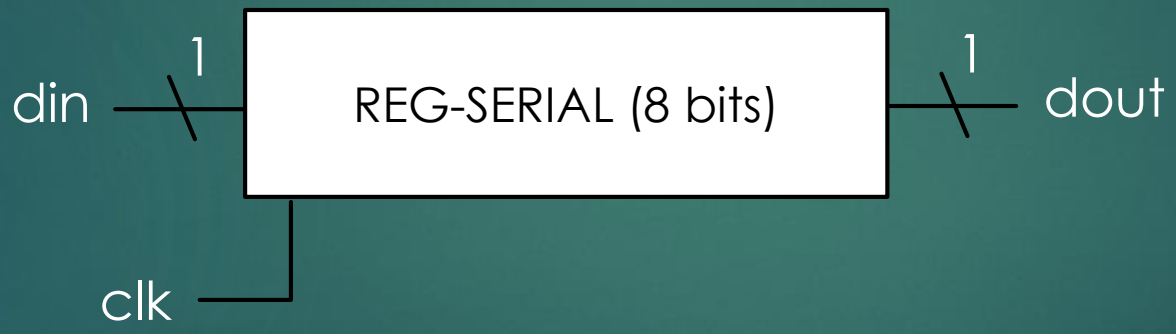
Ejercicios

7. Implemente un comparador secuencial (ver figura) .



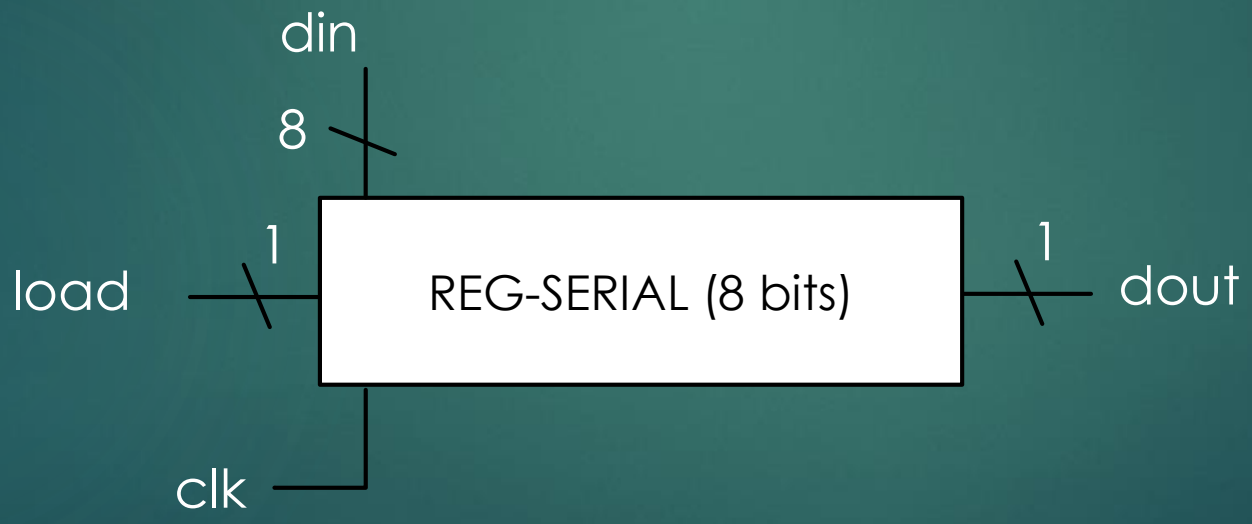
Ejercicios

8. Implemente un registro de 8 bits con corrimiento serial a la derecha (ver figura) .



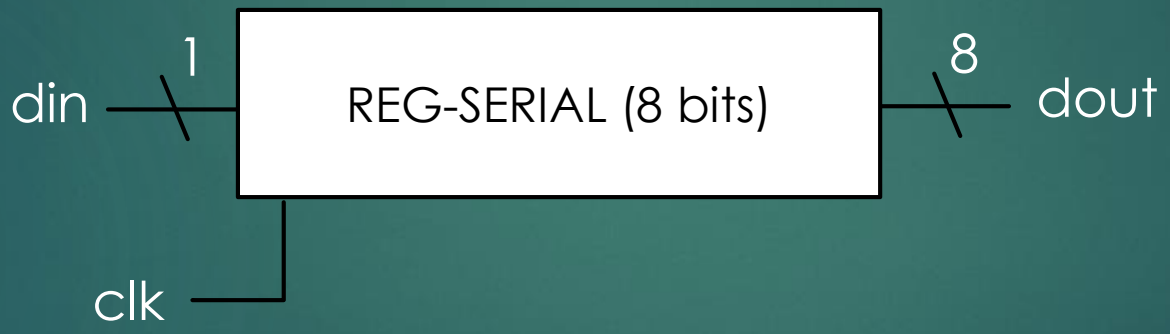
Ejercicios

9. Implemente un registro de 8 bits con corrimiento serial a la derecha y carga en paralelo (ver figura).



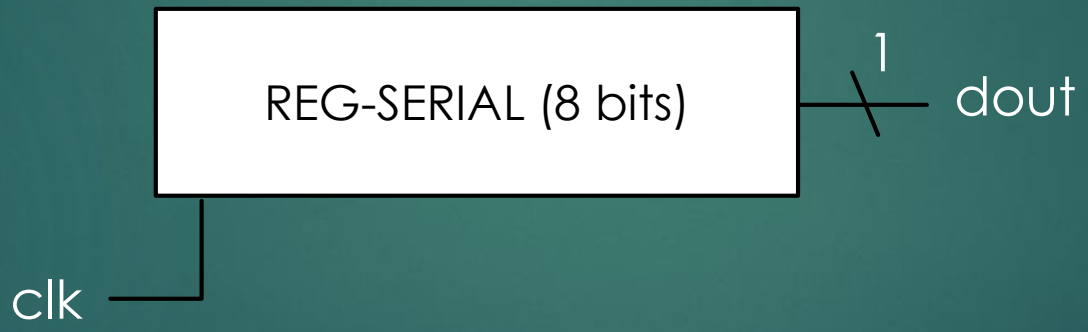
Ejercicios

10. Implemente un registro de 8 bits con corrimiento serial a la derecha y salida en paralelo (ver figura).



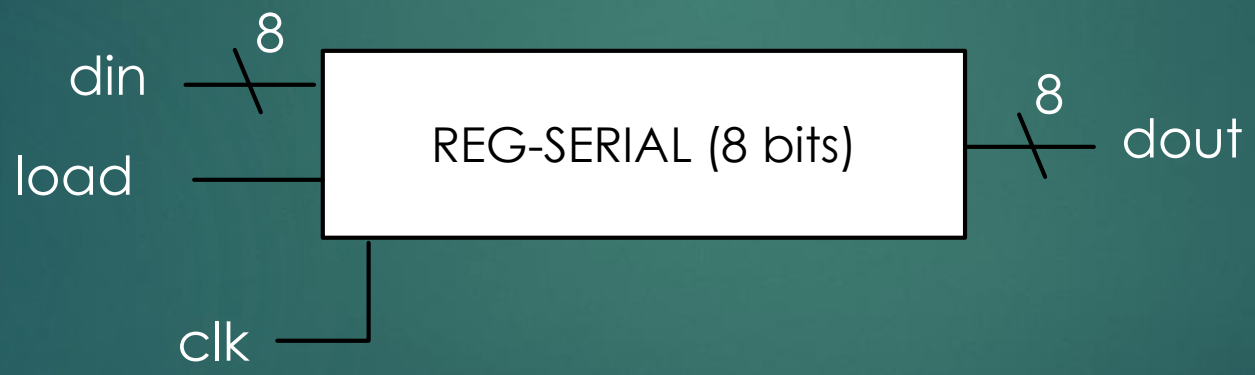
Ejercicios

11. Implemente un registro de 8 bits con rotación a la derecha (ver figura). El valor del registro es “11101010” .



Ejercicios

12. Implemente un registro de 8 bits con entrada y salida en paralelo.



CASE

La sintaxis de CASE es la siguiente

```
[label:] CASE expression IS
      WHEN value => assignments;
      WHEN value => assignments;
      ...
END CASE;
```

Al igual que SELECT, la sentencia CASE permite el uso múltiples valores que pueden agruparse con " | " (que significa "o") o TO (para indicar un rango. Por ejemplo

```
WHEN value1 | value2 |... --value1 or value2 or ...
WHEN value1 TO value2      --range (for enumerated types only)
```

CASE

Nota. CASE requiere que todos los casos sean cubiertos.

El rol principal de CASE es permitir la construcción de circuitos combinacionales dentro de código secuencial (PROCESS). CASE es la contraparte de SELECT.

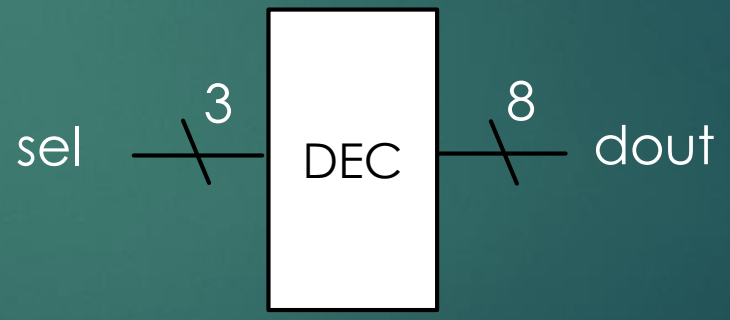
Ejemplo

```
CASE control IS
   WHEN "000"           => x <= a ;  y <= b;
   WHEN "000" | "111"   => x <= b ;  y <= '0';
   WHEN OTHERS          => x <= '0'; y <= '1';
END CASE;
```

Ejercicios

13. Implemente un decodificador que cumpla con la siguiente tabla de verdad.

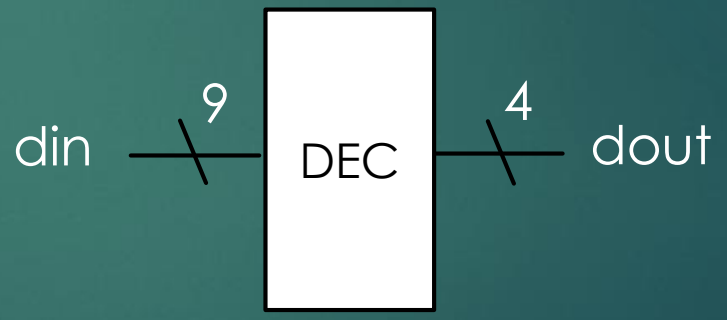
SEL	DOUT
000	00000001
001	00000010
010	00000100
011	00001000
100	00010000
101	00100000
110	01000000
111	10000000



Ejercicios

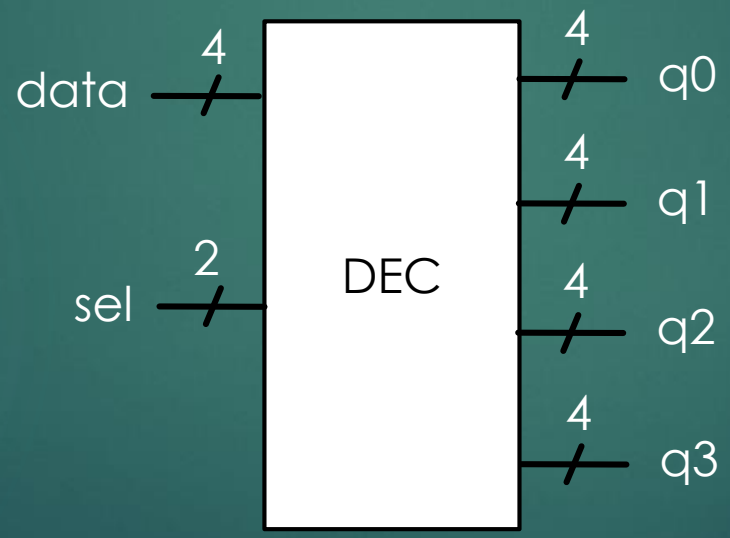
14. Implemente un codificador que cumpla con la siguiente tabla de verdad.

DIN	DOUT
000000000	0000
000000001	0001
000000010	0010
000000100	0011
000001000	0100
000010000	0101
000100000	0110
001000000	0111
010000000	1000
100000000	1001



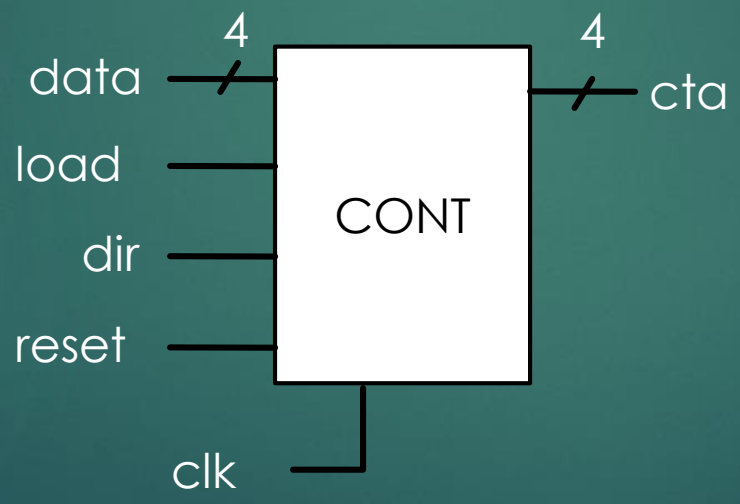
Ejercicios

15. Implemente el demultiplexor que se muestra en la figura



Ejercicios

16. Implemente un contador de 4 bits con cuenta ascendente o descendente, carga en paralelo y reinicio síncrono, (ver figura).



WAIT

Es otra sentencia secuencial. Está disponible en tres formas, de las cuales dos son sintetizables y una es para simulación:

- ✓ WAIT UNTIL. Causa que PROCEES o subprograma se ejecute hasta que la condición se cumpla. Su sintaxis es

```
[label:] WAIT UNTIL condition;
```

Ejemplo

```
PROCESS
BEGIN
WAIT UNTIL (clk'EVENT AND clk='1');
IF (clr='1') THEN
q <= '0';
ELSE
q <= d;
END IF;
END PROCESS;
```

Nota. PROCESS no lleva lista de de sensibilidad en este caso.

WAIT

- WAIT ON. Causa que PROCEES o subprograma se ejecute cuando cambie un elemento que monitorea WAIT ON. Su sintaxis es

```
[label:] WAIT ON sensitivity_list;
```

Ejemplo

```
PROCESS
BEGIN
   IF (clk'EVENT AND clk='1') THEN
      IF (clr='1') THEN
         q <= '0';
      ELSE
         q <= d;
      END IF;
   END IF;
   WAIT ON clk;
END PROCESS;
```

WAIT

- WAIT FOR. Esta sentencia es usada para simulación y describe tiempo de espera. Su sintaxis es

```
[label:] WAIT FOR time_expression;
```

Ejemplo

```
WAIT FOR 40ns;
clk <= NOT clk;
```