Tema 4:

Especificación de sistemas digitales usando VHDL

Diseño automático de sistemas

José Manuel Mendías Cuadros
Dpto. Arquitectura de Computadores y Automática
Universidad Complutense de Madrid

Contenidos

- ✓ VHDL "sintetizable"
- ✓ Especificación usando VHDL.
- ✓ Tipos de datos y operadores.
- ✓ Paquetes estándar.
- ✓ Paquetes no estándar.
- ✓ Lógica combinacional.
- ✓ Ejemplos de lógica combinacional.
- ✓ Lógica secuencial.
- ✓ Ejemplos de lógica secuencial.
- ✓ Portabilidad.
- ✓ AMD 7 Series FPGAs: consejos de codificación.
- ✓ Mezclando VHDL.

VHDL "sintetizable"

- VHDL es un lenguaje de modelado, no de especificación
  - *Todo el HW existente puede ser modelado, pero no todo lo que se especifica puede ser sintetizado.*
  - Desde la óptica de las herramientas EDA, las construcciones VHDL se clasifican en:
    - Soportadas: especifican claramente una funcionalidad HW que debe sintetizarse.
    - Ignoradas: pueden encontrarse en el código fuente VHDL, pero no se tienen en cuenta durante el proceso de síntesis.
    - Prohibidas: no pueden aparecer en el código fuente VHDL, si lo hacen se aborta el proceso de síntesis.
  - El subconjunto sintetizable (construcciones soportadas) y su semántica puede variar de herramienta a herramienta.
    - La semántica de simulación es estándar y la respetan todos los simuladores
    - La semántica de síntesis aunque estandarizada, no se respeta.
- VHDL es un lenguaje orientado a HW que permite escribir SW
  - Es necesario cambiar de mentalidad, pensar en software solo trae problemas.
- El uso de VHDL y herramientas EDA, no evita diseñar
  - Antes de codificar, hay que diseñar el sistema hasta un cierto nivel de abstracción.

VHDL "sintetizable"

familia de estándares

- El lenguaje VHDL está estandarizado por el IEEE.
- Desde la óptica de las herramientas EDA son interesantes:
  - 1076 IEEE Standard VHDL Language Reference Manual
    - Define la sintáxis completa del lenguaje y su semántica de simulación.
  - 1164 IEEE Standard Multivalue Logic System for VHDL Model Interoperability
    - Define tipos (atómicos y vectoriales) multivaluados para representar señales digitales.
  - 1076.3 IEEE Standard VHDL Synthesis Packages
    - Define tipos numéricos enteros con y sin signo como vectores de tipo multivaluado.
  - 1076.6 IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis
    - Define un subconjunto sintetizable del lenguaje y su semántica de síntesis.
- Otros estándares, no directamente relacionados con síntesis son:
  - 1076.1 IEEE Standard VHDL Analog and Mixed-Signal Extensions
  - 1076.1.1 IEEE Standard for VHDL Analog and Mixed-Signal Extensions -- Packages for Multiple Energy Domain Support
  - 1076.2 IEEE Standard VHDL Mathematical Packages
  - 1076.4 IEEE Standard VITAL ASIC (Application-Specific Integrated Circuit) Modeling Specification

J.M. Mendías
2024

Especificación usando VHDL

- Pensar siempre en HW síncrono de nivel RT (las herramientas que utilizaremos comienzan desde este nivel de abstracción).
  - No comenzar a codificar hasta tener el diagrama de bloques RTL (módulos combinacionales y registros).
    - El uso de VHDL no evita el diseño manual de la estructura RT, lo que evita es el diseño lógico de los bloques combinacionales/secuenciales y la proyección tecnológica.
  - Cada bloque RT habitualmente deberá codificarse con una sentencia concurrente.
    - Se permiten especificaciones más abstractas, pero para que puedan obtenerse soluciones válidas es necesaria una codificación muy cuidadosa.
  - Nunca mezclar en un mismo diseño diferentes estilos de temporización:
    - En particular, usaremos temporización por flanco de subida de un único reloj
  - El núcleo fundamental de la lógica a sintetizar se especificará mediante expresiones.
    - El resto de construcciones del lenguaje se utilizarán para estructurar el diseño y para especificar su temporización.

- En muchos sentidos *VHDL permite "capturar esquemas" textualmente*.
  - Ya que la estructura RTL especificada se respeta durante la síntesis.

Tipos de datos

- Las herramientas EDA soportan los tipos predefinidos siguientes:
  - `Boolean`, `bit` y `bit_vector`
  - `Character` y `string`
  - `Integer`, `positive` y `natural`
    - No se soportan con precisión infinita, el rango deberá poder inferirse del código o especificarse de forma explícita.
    - Todo objeto se implementa mediante el vector de bits más pequeño capaz de representar el rango especificado.
    - Si este rango incluye números negativos, la representación usada será complemento a 2, en otro caso binaria pura.

- También soportan los tipos siguientes:
  - `std_ulogic`, `std_ulogic_vector`, `std_logic`, `std_logic_vector`
    - Definidos en el paquete estándar `ieee.std_logic_1164`
  - `signed`, `unsigned`
    - Definidos en los paquetes `ieee.numeric_bit` / `ieee.numeric_std` / `std_logic_arith`
    - Representan enteros como vectores de bits codificados en complemento a 2 o en binario puro.

Tipos de datos

- Si un tipo está soportado, también lo están todos los operadores asociados a ellos:
  - Lógicos: **`and`** | **`or`** | **`nand`** | **`nor`** | **`xor`** | **`not`**
  - Relacionales: **`=`** | **`/=`** | **`<`** | **`<=`** | **`>`** | **`>=`**
  - Aritméticos: **`+`** | **`-`** | **`*`** | **`/`** | **`mod`** | **`rem`** | **`abs`**
  - Concatenación: **`&`**

- La definición de tipos por parte del diseñador también se soporta:
  - Enumerados
    - Todo objeto se implementa mediante el vector de bits más pequeño capaz de representar todos los valores posibles del tipo.
    - Por defecto, cada uno de los literales del tipo se representa mediante la codificación en binario puro de su posición dentro de la declaración (aplica a **`boolean`**, **`bit`** y **`character`**).
  - Arrays
    - Se soportan vectores monodimensionales con índice entero de cualquier tipo soportado.
  - Registros
    - Se soportan registros cuyas componentes sean de cualquier tipo soportado.

J.M. Mendías
2024

Paquetes estándar

ieee.std_logic_1164 (i)

- El paquete **std_logic_1164** define 2 tipos atómicos multivaluados (uno resuelto y otro no) y 2 tipos vectoriales asociados:

```
type std_ulogic is ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-' );
type std_ulogic_vector is array ( natural range <> ) of std_ulogic;

subtype std_logic is resolved std_ulogic;
type std_logic_vector is array ( natural range <> ) of std_logic;
```

- También sobrecarga los operadores lógicos (atómicos y vectoriales) para que trabajar sobre los tipos definidos:
  - **and**, **nand**, **or**, **nor**, **xor**, **xnor**, **not**
  - Este paquete NO define ningún operador aritmético ni relacional.

- Los tipos resueltos (**std_logic** y **std_logic_vector**) son el estándar para la especificación del interconexionado entre modelos VHDL
  - Los puertos de un diseño deberán ser siempre de estos tipos.
- Además deben usarse para especificar señales no numéricas.

Paquetes estándar

ieee.std_logic_1164 (ii)

- El significado de cada uno de los 9 valores es:
  - **'0'** y **'1'**: valores lógicos fuertes
  - **'L'** y **'H'**: valores lógicos débiles asimilables a **'0'** y **'1'**

    } su asignación implican conexiones a tierra y alimentación
  - **'Z'**: alta impedancia — su asignación implica un buffer triestado
  - **'-'**: indiferencia (don't care) — su asignación permite simplificar las expresiones en que se usa
  - **'X'**: valor fuerte desconocido
  - **'W'**: valor débil desconocido
  - **'U'**: no inicializado

    } sin uso en síntesis

- La comparación con valores metalógicos no tiene sentido en síntesis:
  - Una comparación de igualdad con un valor metalógico es siempre falsa.
  - Una comparación de desigualdad con un valor metalógico es siempre cierta.

```
z <= '0' when s="00-1" else '1';
```
```
z <= '1';
```

```
z <= '0' when s/="00-1" else '1';
```
```
z <= '0';
```

```
z <= '0' when s="0011" else '-';
```
```
z <= '0';
```

```
with s select z <=
  '0' when "0000",
  '1' when "0001",
  x1  when "0---",
  x2  when others;
```
```
with s select z <=
  '0' when "0000",
  '1' when "0001",
  x2  when others;
```

Paquetes estándar

ieee.numeric_std (i)

- El paquete **numeric_std** define 2 tipos de datos vectoriales que, basados en **std_logic**, son interpretables como valores numéricos codificados.
  - La longitud del vector determina el rango de valores representables.
  - El tipo **unsigned** especifica un número natural representado en binario puro.

```
type unsigned is array (natural range <>) of std_logic;
```

  - El tipo **signed** especifica un número entero representado en complemento a 2

```
type signed is array (natural range <>) of std_logic;
```

  - Estos tipos son compatibles con **std_logic_vector** por casting.
- También define:
  - Funciones de conversión entre los tipos: **integer**, **natural**, **unsigned** y **signed**
  - Operadores sobrecargados aritméticos y relacionales con argumentos mixtos de los tipos **natural/unsigned** o **integer/signed** con resultados **unsigned** o **signed**
  - Operadores sobrecargados lógicos para los tipos: **unsigned** y **signed**
  - Funciones de redimensionado, desplazamiento y rotación para los tipos: **unsigned** y **signed**

J.M. Mendías
2024

Paquetes estándar

ieee.numeric_std (ii)

Perfiles de las funciones aritméticas

```
function "+"(L,R: UNSIGNED) return UNSIGNED;
function "+"(L,R: SIGNED) return SIGNED;
function "+"(L: UNSIGNED; R: NATURAL) return UNSIGNED;
function "+"(L: NATURAL; R: UNSIGNED) return UNSIGNED;
function "+"(L: INTEGER; R: SIGNED) return SIGNED;
function "+"(L: SIGNED; R: INTEGER) return SIGNED;
```

Perfiles de las funciones relacionales

```
function ">"(L,R: UNSIGNED) return BOOLEAN;
function ">"(L,R: SIGNED) return BOOLEAN;
function ">"(L: NATURAL; R: UNSIGNED) return BOOLEAN;
function ">"(L: INTEGER; R: SIGNED) return BOOLEAN;
function ">"(L: UNSIGNED; R: NATURAL) return BOOLEAN;
function ">"(L: SIGNED; R: INTEGER) return BOOLEAN;
```

Perfiles de las funciones de conversión

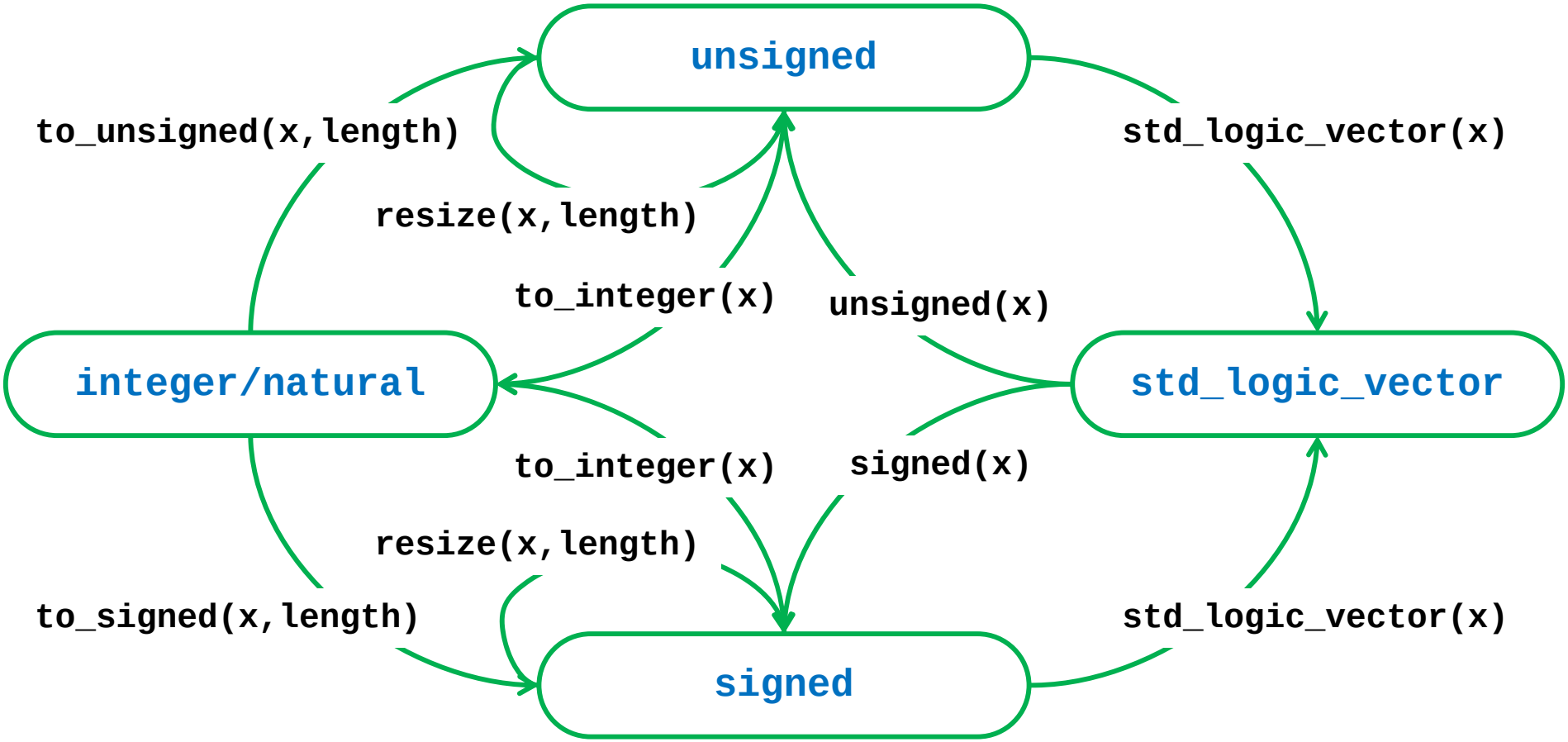
```
function TO_INTEGER(ARG: UNSIGNED) return NATURAL;
function TO_INTEGER(ARG: SIGNED) return INTEGER;
function TO_UNSIGNED(ARG,SIZE: NATURAL) return UNSIGNED;
function TO_SIGNED(ARG: INTEGER; SIZE: NATURAL) return SIGNED;
function TO_UNSIGNED(ARG: STD_LOGIC_VECTOR) return UNSIGNED;
function TO_SIGNED(ARG: STD_LOGIC_VECTOR) return SIGNED;
function TO_STDLOGICVECTOR(ARG: UNSIGNED) return STD_LOGIC_VECTOR;
function TO_STDLOGICVECTOR(ARG: SIGNED) return STD_LOGIC_VECTOR;
```

} funciones alternativas al uso del casting

Paquetes estándar

ieee.numeric_std (iii)

- Los tipos numéricos se usan internamente en los modelos VHDL
  - Para pasar de un tipo a otro se usa casting o funciones de conversión.
  - El uso de casting o funciones de conversión no implica mayor coste hardware.



Paquetes estándar

ieee.numeric_std (iv)

- Todos los operadores binarios admiten argumentos de distinta anchura
  - Cuando ambos argumentos son vectoriales, la anchura del resultado es la anchura del máximo de los argumentos
  - Cuando solo un argumento es vectorial, la anchura del resultado es la anchura de dicho argumento.

- En los operadores relacionales, el uso de argumentos de distintos tipos y distinta anchura puede dar lugar a paradojas:

arg1	op	arg2	unsigned	signed	std_logic_vector
"000"	=	"000"	true	true	true
"00"	=	"000"	true	true	false
"100"	=	"0100"	true	false	false
"000"	<	"000"	false	false	false
"00"	<	"000"	false	false	true
"100"	<	"0100"	false	true	false

Paquetes estándar

ieee.numeric_std (v)

- Aunque los tipos signed/unsigned sean vectores de std_logic, no es necesario que las constantes numéricas sean binarias:
  - Todas las funciones aritméticas y relacionales permiten que uno de sus argumentos sea entero/natural.
  - Para asignar valores explícitos pueden usarse las funciones de conversión.

```
architecture ...;
  ...
  signal a, b, c, d : unsigned(... downto 0);
begin
  ...
  a <= b + 1;
  c <= d when a<100 else to_unsigned(100,c'length);
end;
```

- Además, VHDL ofrece facilidades para expresar valores explícitos:
  - Las cadenas de bit/std_logic pueden expresarse (con y sin separadores) en:
    - Binario: "10100110" / "1010_0110"
    - Hexadecimal: X"10a6" / X"fad2_0023"
  - Los enteros pueden expresarse (con y sin separadores) en:
    - Decimal: 4563 / 4_563
    - Binario: 2#10100110# / 2#1010_0110#
    - Hexadecimal 16#10a6# / 16#fad2_0023#

Paquetes no estándar

std_logic_unsigned / std_logic_signed / std_logic_arith (i)

- Todavía es común encontrar especificaciones que usan paquetes numéricos propietario antiguos:
  - El paquete **`std_logic_unsigned`** define operadores aritméticos y relacionales con argumentos mixtos de tipos: **`integer`**, **`natural`** y **`std_logic_vector`**
    - Interpreta directamente los argumentos de tipo **`std_logic_vector`** como números sin signo representados en binario puro.
    - Se usaban internamente en módulos que manipulen solo datos sin signo.
  - El paquete **`std_logic_signed`** define operadores aritméticos y relacionales con argumentos mixtos de tipos: **`integer`**, **`natural`** y **`std_logic_vector`**
    - Interpreta directamente los argumentos de tipo **`std_logic_vector`** como números con signo representados en complemento a 2.
    - Se usaban internamente en módulos que manipulen solo datos con signo.
  - El paquete **`std_logic_arith`** define
    - 2 tipos vectoriales de **`std_logic`**: **`unsigned`** y **`signed`**
    - Operadores aritméticos y relaciones con argumentos mixtos de los tipos **`integer`**, **`natural`**, **`std_ulogic`**, **`unsigned`** o **`signed`** y con resultados **`std_logic_vector`**, **`unsigned`** o **`signed`**
    - A diferencia de **`numeric_std`**, este paquete define operadores con todas las combinaciones posibles de tipo de argumento y el resultado puede ser **`std_logic_vector`**.

J.M. Mendías
2024

Paquetes no estándar

std_logic_unsigned / std_logic_signed / std_logic_arith (ii)

Perfiles de las funciones aritméticas

```
function "+"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED;
function "+"(L: SIGNED; R: SIGNED) return SIGNED;
function "+"(L: UNSIGNED; R: SIGNED) return SIGNED;
function "+"(L: SIGNED; R: UNSIGNED) return SIGNED;
function "+"(L: UNSIGNED; R: INTEGER) return UNSIGNED;
function "+"(L: INTEGER; R: UNSIGNED) return UNSIGNED;
function "+"(L: SIGNED; R: INTEGER) return SIGNED;
function "+"(L: INTEGER; R: SIGNED) return SIGNED;
function "+"(L: UNSIGNED; R: STD_ULOGIC) return UNSIGNED;
function "+"(L: STD_ULOGIC; R: UNSIGNED) return UNSIGNED;
function "+"(L: SIGNED; R: STD_ULOGIC) return SIGNED;
function "+"(L: STD_ULOGIC; R: SIGNED) return SIGNED;
function "+"(L: UNSIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "+"(L: SIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
function "+"(L: UNSIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
function "+"(L: SIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "+"(L: UNSIGNED; R: INTEGER) return STD_LOGIC_VECTOR;
function "+"(L: INTEGER; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "+"(L: SIGNED; R: INTEGER) return STD_LOGIC_VECTOR;
function "+"(L: INTEGER; R: SIGNED) return STD_LOGIC_VECTOR;
function "+"(L: UNSIGNED; R: STD_ULOGIC) return STD_LOGIC_VECTOR;
function "+"(L: STD_ULOGIC; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "+"(L: SIGNED; R: STD_ULOGIC) return STD_LOGIC_VECTOR;
function "+"(L: STD_ULOGIC; R: SIGNED) return STD_LOGIC_VECTOR;
```

Paquetes standard vs. no

- El uso de un paquete u otro no afecta a la implementación solo afecta a lo compacta que es la especificación:

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

entity adder is
port(
    x :  in std_logic_vector(7 downto 0);
    y :  in std_logic_vector(7 downto 0);
    s : out std_logic_vector(7 downto 0) );
end adder;

library ieee;
use ieee.numeric_std.all;

architecture syn1 of adder is
begin
  s <= std_logic_vector(unsigned(x) + unsigned(y));
end syn1;
```

```
library ieee;
use ieee.std_logic_unsigned.all;

architecture syn2 of adder is
begin
  s <= x + y;
end syn2;
```

```
library ieee;
use ieee.std_logic_arith.all;

architecture syn3 of adder is
begin
  s <= unsigned(x) + unsigned(y);
end syn3;
```

Lógica combinacional

- La lógica combinacional puede ser especificada mediante:
  - Asignaciones concurrentes a señal.
    - En el caso de que sean asignaciones condicionales deben tener rama **`else`**.
  - Procesos sin sentencias **`wait`** tales que:
    - Todas las señales o variables escritas dentro del proceso sean asignadas al menos una vez en toda activación del mismo.
    - No contengan bucles **`while`** o **`loop`** y el rango del índice de los bucles **`for`** sea estático.
    - Aunque la lista de sensibilidad se ignora, para asegurar la coherencia entre simulación y síntesis, esta debe ser completa (formada por todas las señales leídas dentro del proceso).

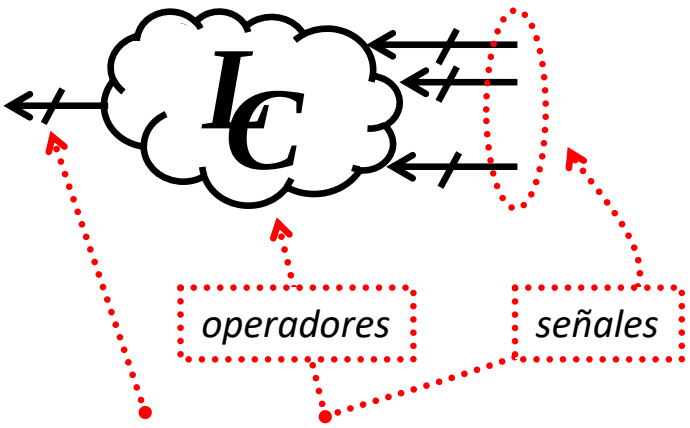
- En cualquiera de los casos, cuando se utilizan señales:
  - Los valores iniciales se ignoran.
  - La definición explícita de retardos (**`after`**) se ignora.
  - La asignación de múltiples elementos de forma de onda está prohibida.

- Para asegurar que la lógica especificada es combinacional:
  - Ninguna señal puede formar parte de la/s expresión/es que la definen.

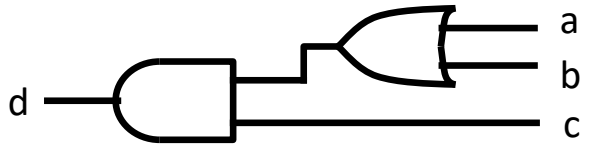
Lógica combinacional

asignaciones concurrentes (i)

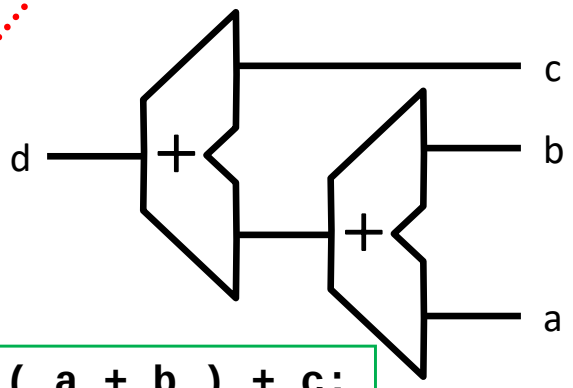
- Toda asignación de señal se implementa como un bloque de lógica combinacional:
  - Con un único puerto de salida (que puede ser vectorial en modelos de nivel RT).
  - Con tantos puertos de entrada como señales diferentes aparezcan en la expresión.
  - Con una funcionalidad especificada por los operadores que forman la expresión.




```
d <= ( a or b ) and c;
```



Una posible implementación de la sentencia ya que la implementación definitiva siempre la decide la herramienta




```
d <= ( a + b ) + c;
```

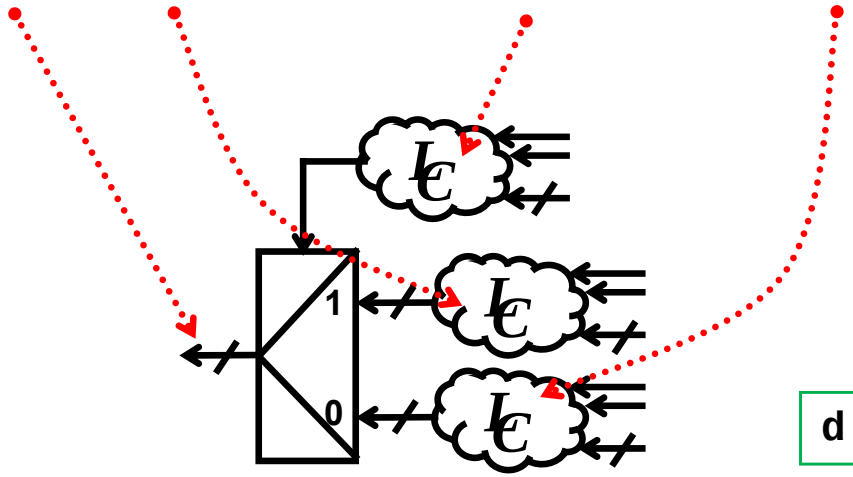
J.M. Mendías
2024

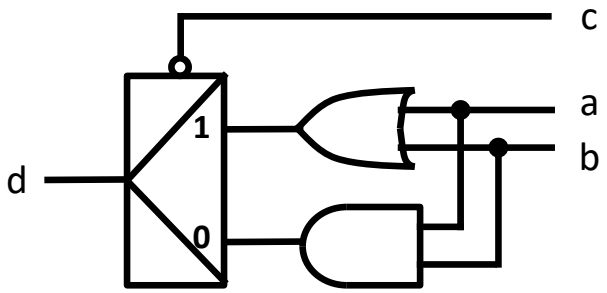
Lógica combinacional

asignaciones concurrentes (ii)

- Toda asignación condicional de señal completa (con rama **`else`**) se implementa como un bloque de lógica combinacional:
  - Con un único puerto de salida (que puede ser vectorial en modelos de nivel RT).
  - Con tantos puertos de entrada como señales diferentes aparezcan en el lado derecho de la asignación (independientemente de la expresión en la que ocurran).
  - Con un comportamiento que se corresponde con el de un multiplexor 2 a 1 cuyas 3 entradas están conectadas a las salidas de 3 bloques combinacionales.
    - La funcionalidad de dichos bloques queda especificada por los operadores que forman cada una de las 3 expresiones de la sentencia.

señal <= expresión **`when`** expresión_booleana **`else`** expresión;





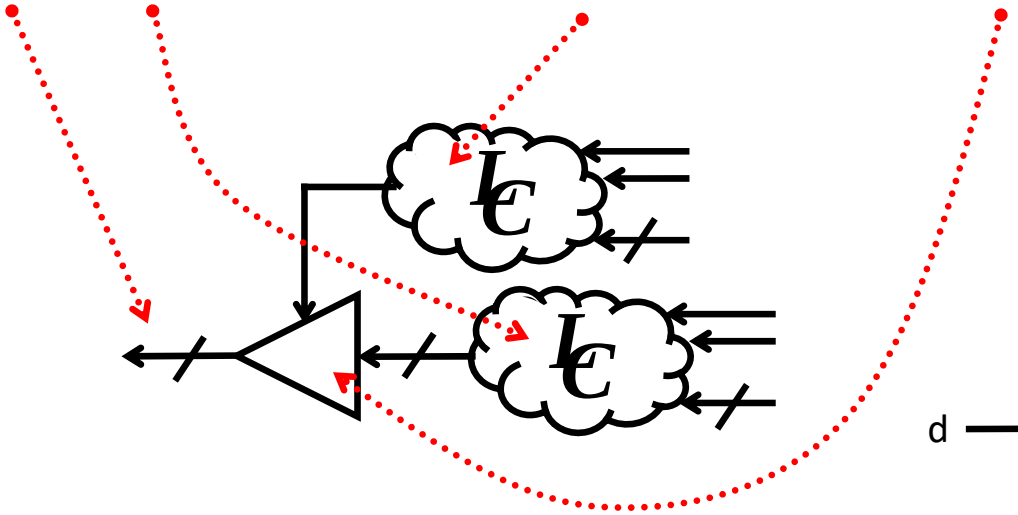

```
d <= (a or b) when c='0' else (a and b);
```

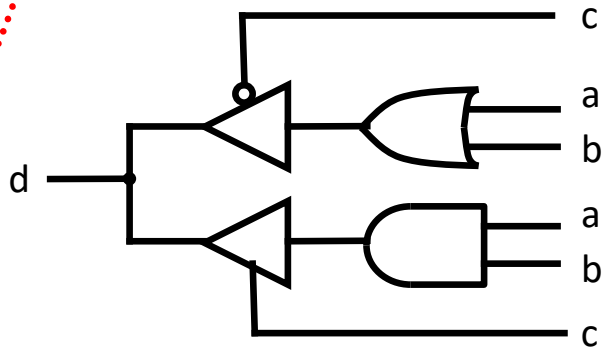
Lógica combinacional

asignaciones concurrentes (iii)

- La asignación condicional del valor explícito 'Z' (uno de los valores del tipo std_logic) especifica la capacidad tri-estado del puerto de salida de la lógica combinacional especificada por el resto de la sentencia.

señal <= expresión **when** expresión_booleana **else** (**others**=>'Z');





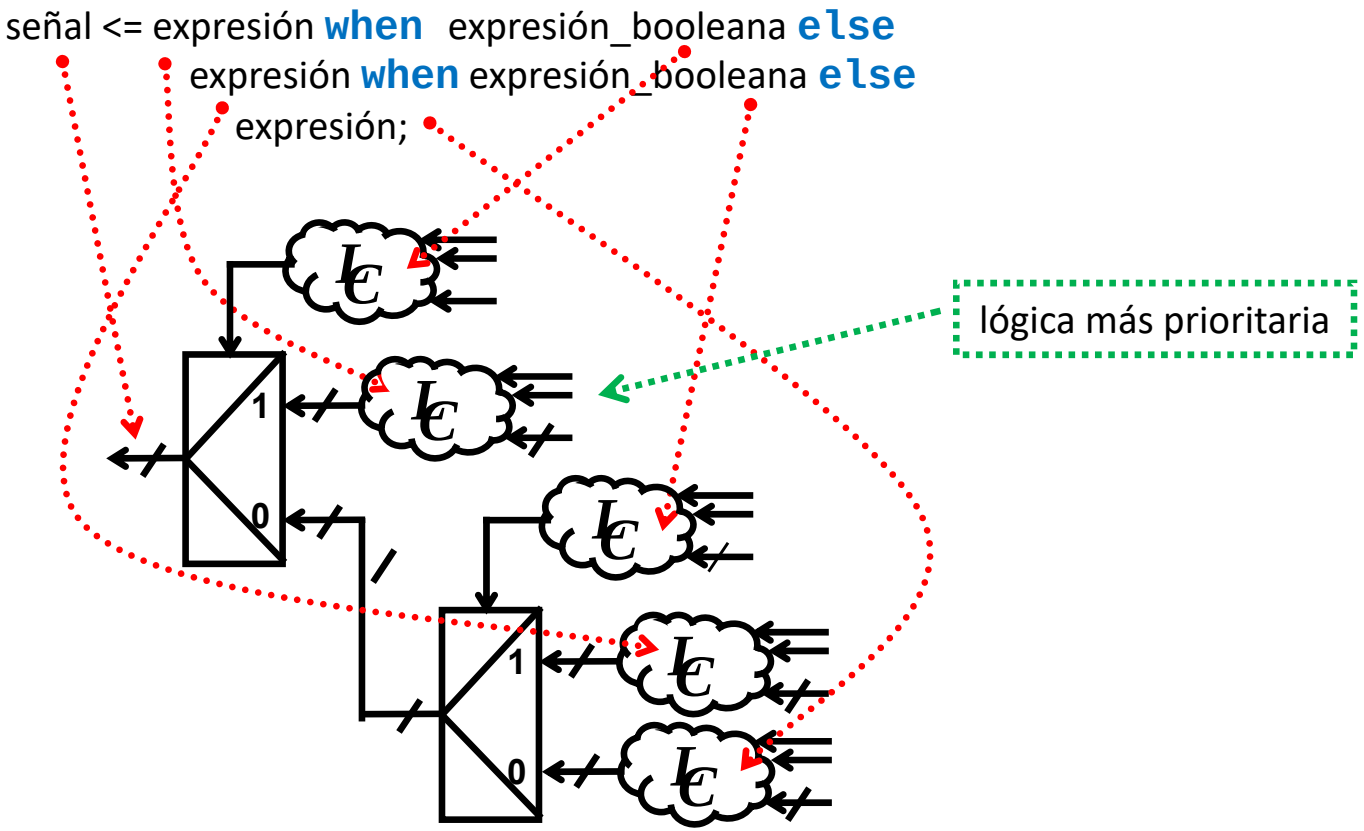

```
d <= (a or b)  when c='0' else 'Z';
d <= (a and b) when c='1' else 'Z';
```

J.M. Mendías
2024

Lógica combinacional

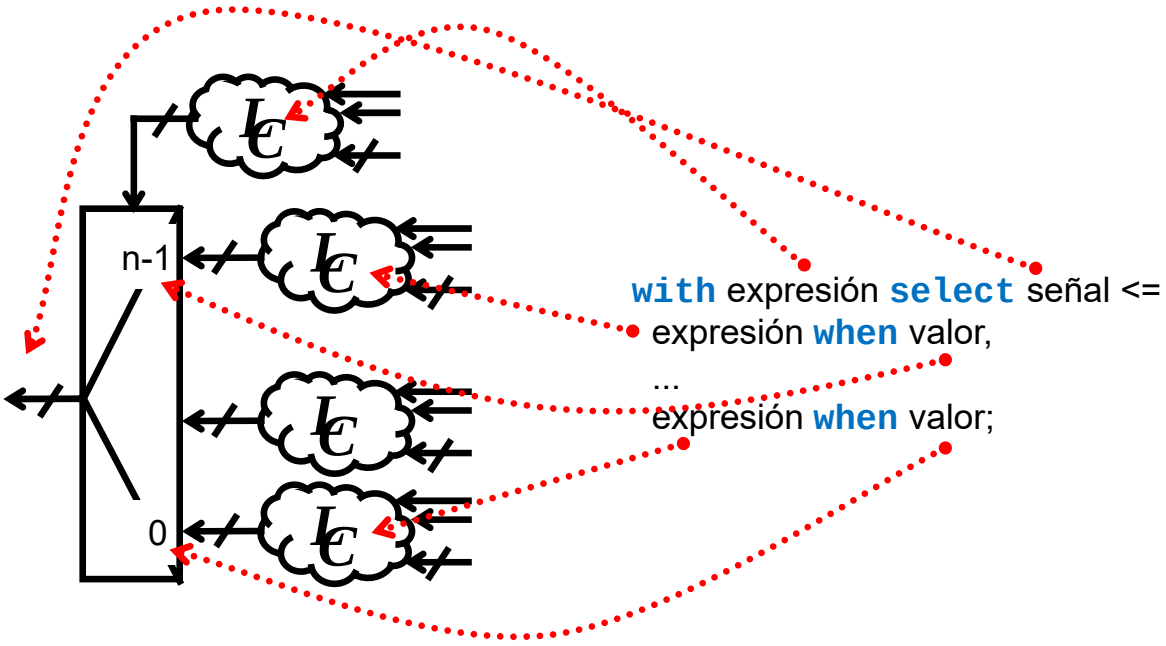
asignaciones concurrentes (iv)

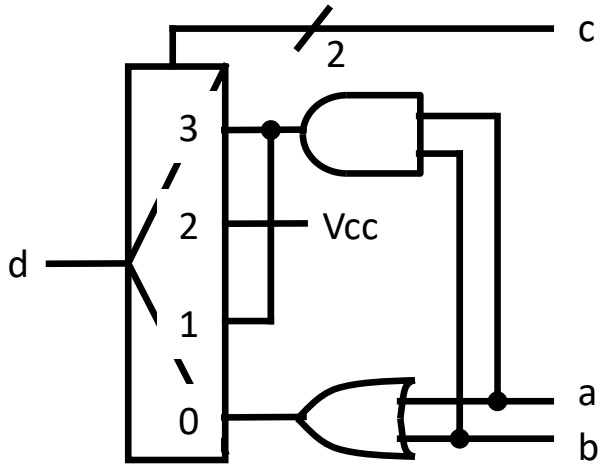
- Una colección de asignaciones condicionales anidadas se implementa como lógica combinacional en cascada que establece prioridades explícitas entre varios cálculos.



Lógica combinacional
asignaciones concurrentes (v)

- Toda asignación selectiva de señal se implementa como un bloque de lógica combinacional:
  - Con un único puerto de salida (que puede ser vectorial en modelos de nivel RT).
  - Con tantos puertos de entrada como señales diferentes aparezcan en el lado derecho de la asignación (independientemente de la expresión en la que ocurran).
  - Con un comportamiento que se corresponde con el de un multiplexor 2^n a 1 cuyas 2^n entradas están conectadas a las salidas de n+1 bloques combinacionales.
    - Donde n el número de bits con los que se codifica el resultado de la expresión seleccionada.
    - La funcionalidad de dichos bloques queda especificada por los operadores que forman cada una de las n+1 expresiones de la sentencia.






```
with c select
  d <= (a or b) when "00",
    (a and b) when "11" | "10",
    '1' when others;
```

Lógica combinacional

procesos equivalentes

- Toda asignación concurrente tiene un proceso equivalente con el mismo comportamiento (y que especifica la misma lógica)

Asignación de señal

```
c <= a and b;
```

```
process (a, b)
begin
  c <= a and b;
end process;
```

Asignación condicional de señal

```
c <= (a and b) when d='1'
     else (a or b);
```

```
process (a, b, d)
begin
  if  d='1' then
    c <= a and b;
  else
    c <= a or b;
  end if;
end process;
```

Asignación selectiva de señal

```
with a+b select
  c <= d when "0000",
       not d when "1111",
       '1' when others;
```

```
process (a, b, d)
begin
  case a+b is
    when "0000" => c <= d;
    when "1111" => c <= not d;
    when others => c <= '1';
  end case;
end process;
```

Lógica combinacional

expresiones (i)

- Las expresiones son el mecanismo fundamental para especificar la funcionalidad combinacional de un circuito.
- Las expresiones VHDL, desde el punto de vista de síntesis, pueden clasificarse en:
  - Expresiones computables: aquellas cuyo resultado puede ser determinado en tiempo de análisis, por tanto tienen un valor estático.
    - No requieren HW que las calcule, se implementan como conexiones a VCC y/o GND.
    - Pueden usarse a discreción para facilitar la legibilidad del código.
  - Expresiones no computables: aquellas cuyo resultado no puede ser determinado en tiempo de análisis, por tanto tienen un valor dinámico.
    - Se implementan mediante HW combinacional que las calcula.
- Son expresiones no computables aquellas que contengan:
  - Puertos.
  - Variables o señales que hayan sido asignadas por una expresión no computable.
  - Variables o señales que hayan sido asignadas (aunque sea por expresiones computables) en función del valor de una condición no computable.

Lógica combinacional

expresiones (ii)

computable, v0 vale '1'

computable, v1 vale '1'

computable, v2 vale '0'

computable, vInt vale 3

computable, vVector vale "1000"

computable, v1 vale '1'

computable, v2 vale '0'

computable, v0 vale '0'

computable, v1 vale '1'

computable, v1 vale '1'

no computable, v2 depende de s

no computable, v1 depende de s

no computable, v2 depende de s

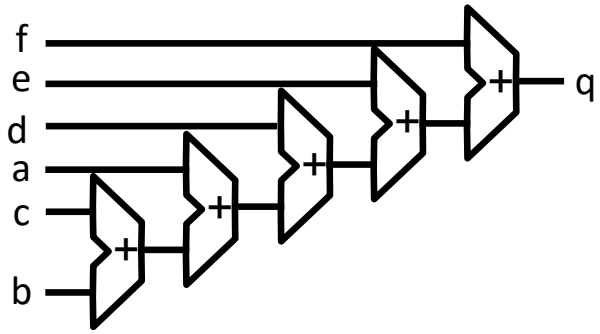
```
architecture ... of ... is
  signal s : std_logic;
  function mux( a, b, c : std_logic ) return std_logic is
  begin
    if (c = '1') then return a; else return b; end if;
  end;
  procedure comp( a : std_logic; b : out std_logic ) is
  begin
    b := not a;
  end;
  ...
begin
  process ( s )
    variable v0, v1, v2 : std_logic;
    variable vInt : integer;
    subtype miVector is std_logic_vector(0 to 3);
    variable vVector : miVector;
  begin
    v0 := '1';
    v1 := v0;
    v2 := not v1;
    for i in 0 to 3 loop vInt := i; end loop;
    vVector := miVector'(v1, v2, '0', '0');
    v1 := mux( v0, v1, v2 );
    comp( v1, v2 );
    v0 := s and '0';
    v1 := mux( s, '1', '0' );
    v1 := mux( '1', '1', s );
    if (s = '1') then v2 := '0'; else v2 := 1; end if;
    v1 := s;
    v2 := v1 or '0';
  end process;
  ...
end ...;
```

Lógica combinacional

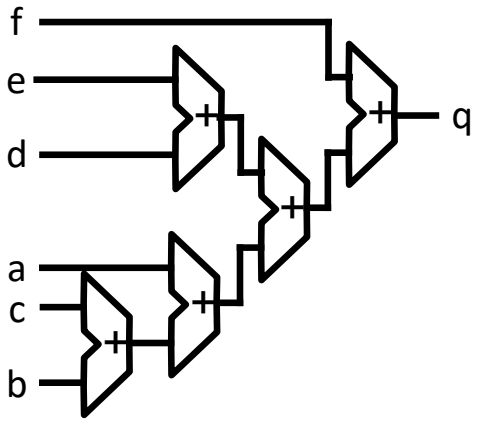
expresiones (iii)

- Las expresiones simples (formadas un único operador) especifican una conducta primitiva sin estructura interna aparente.
- Las expresiones compuestas (formada por varios operadores) o las expresiones en cascada especifican:
  - La propia conducta a diseñar.
  - Una ordenación de los cálculos que se traduce en una estructura inicial sobre la que la herramienta comenzará a optimizar.
  - La estructura inicial es determinante (sobre todo a nivel RT) ya que puede desde facilitar la búsqueda del diseño óptimo hasta ocultarlo.

```
q <= b + c + a + d + e + f;
```




```
q <= (((b+c)+a)+(d+e))+f;
```



J.M. Mendías
2024

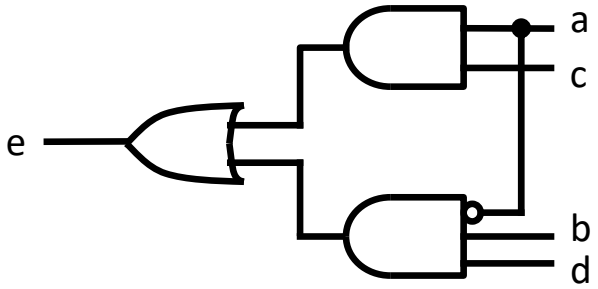
Lógica combinacional

procesos (i)

- Un proceso se implementará como HW combinacional si y solo si
  - No tiene sentencias **wait** y todas las señales o variables escritas dentro del mismo se asignan al menos una vez bajo cualquier condición de ejecución.

```
process (a, b, c, d)
begin
  if a='1' then
    e <= c;
  elsif b='1' then
    e <= d;
  else
    e <= '0';
  end if;
end process;
```

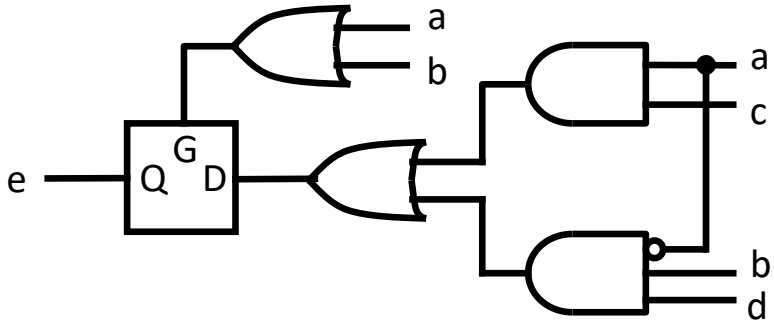
a	b	e
0	0	0
0	1	d
1	0	c
1	1	c



  - Si alguna señal o variable no se asigna bajo alguna condición de ejecución, se implementara HW secuencial para ella.

```
process (a, b, c, d)
begin
  if a='1' then
    e <= c;
  elsif b='1' then
    e <= d;
  end if;
end process;
```

a	b	e
0	0	no cambia
0	1	d
1	0	c
1	1	c



J.M. Mendías
2024

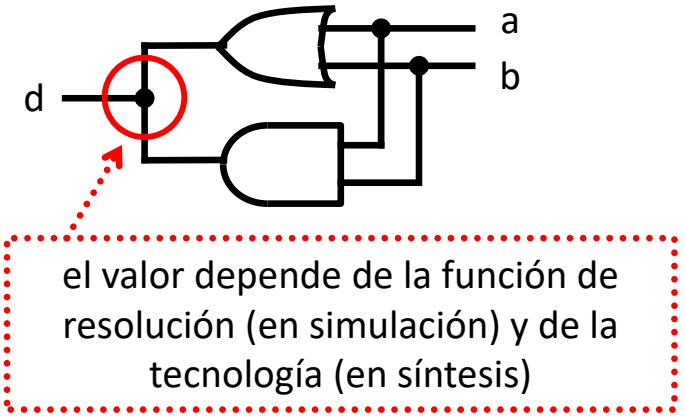
Lógica combinacional

procesos (ii)

- El orden de asignación concurrente de señales no importa.
  - Aunque es posible (solo si el tipo de la señal es resuelto) no es recomendable asignar concurrentemente varias veces una misma señal.

```
architecture ...;
  signal d : ...;
begin
  ...
  d <= a or b;
  d <= a and b;
  ...
end ...;
```

```
architecture ...;
 signal d : ...;
begin
  ...
  d <= a and b;
  d <= a or b;
  ...
end ...;
```



- El orden de asignación secuencial de señales sí importa:
  - Si bajo unas condiciones de ejecución un proceso asigna varias veces una misma señal sólo se diseñará lógica para implementar la última asignación.

```
process (a, b)
begin
  d <= a and b;
  d <= a or b;
end process;
```


```
process (a, b)
begin
  d <= a or b;
  d <= a and b;
end process;
```

J.M. Mendías
2024

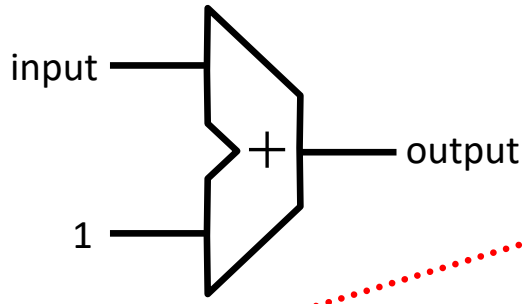
Lógica combinacional

procesos (iii)

- El orden de asignación secuencial de variables puede o no importar:

- En procesos implementados como lógica combinacional, el orden de asignación secuencial de variables no importa.

```
process ( input )
  variable a : ...;
begin
  a := input + 1;
  output <= a;
end process;
```




```
process ( input )
  variable a : ...;
begin
  output <= a;
  a := input + 1;
end;
```

Al ser ambos procesos 'combinacionales', actualizan variables y señales en cualquiera de sus ejecuciones, como en el modelo VHDL las variables conservan valores entre llamadas la implementación debe ser en cualquier caso un incrementador.
Esto provoca inconsistencias entre simulación y síntesis y la herramienta avisa de que se lee la variable antes de escribirla.

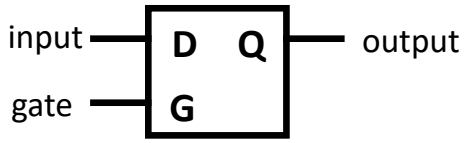
J.M. Mendías
2024

Lógica combinacional

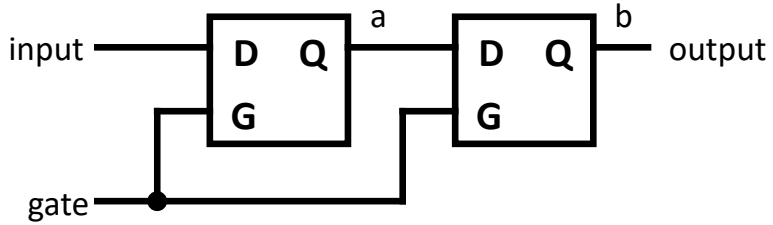
procesos (iv)

- En procesos implementados como lógica secuencial, el orden de asignación secuencial de variables sí importa.
  - Dicho orden determina el número de elementos de memoria que se implementan

```
process (gate, input)
  variable a, b : ...;
begin
  if gate='1' then
    a := input;
    b := a;
    output <= b;
  end if;
end process;
```




```
process (gate, input)
  variable a, b : ...;
begin
  if gate='1' then
    b := a;
    a := input;
    output <= b;
  end if;
end process;
```



- Por ello, se recomienda usar siempre señales para almacenar el estado de un proceso 'secuencial'
  - Las variables se usan para simplificar expresiones en procesos 'combinacionales'

J.M. Mendías
2024

Lógica combinacional

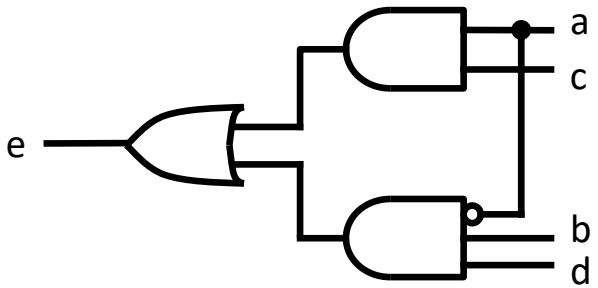
procesos (v)

- Un error típico al implementar procesos es que se genere lógica secuencial cuando el diseñador quiso especificar lógica combinacional
  - Porque dejó alguna sentencias `if` sin rama `else`
  - Porque dejó algún caso sin cubrir en alguna sentencia `case` (o sin `others`).

- Para evitar este error se recomienda asignar valores por defecto a todas las variables o señales al comienzo del proceso.
  - Estos valores por defecto serán sobreescritos por el resto del código.

```
process (a, b, c, d)
begin
  e <= '0';
  if a='1' then
    e <= c;
  elsif b='1' then
    e <= d;
  end if;
end process;
```

a	b	e
0	0	0
0	1	d
1	0	c
1	1	c

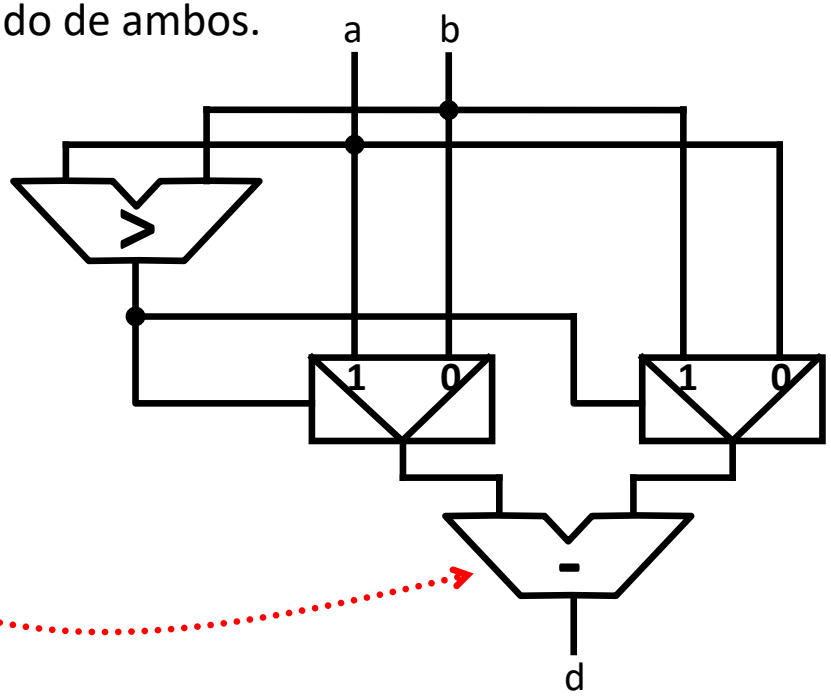


Lógica combinacional

reuso (i)

- Habitualmente, cada ocurrencia de un operador da lugar a un bloque de lógica (recurso) que la implementa.
  - Cuando un único bloque de lógica implementa varias ocurrencias de un operador se dice que dichas ocurrencias comparten el recurso.
- Para que dos operaciones compartan un recurso debe cumplirse que:
  - Ocurran dentro del mismo proceso.
  - Sean mutuamente exclusivos: no exista ninguna posible ejecución en la que se necesite calcular simultáneamente el resultado de ambos.
  - Exista un recurso capaz de ejecutar ambas.

```
process (a, b)
begin
  if a>b then
    d <= a - b;
  else
    d <= b - a;
  end if;
end process;
```



Lógica combinacional

reuso (ii)

- La detección de pares de operaciones mutuamente exclusivas depende de la pericia de la herramienta EDA
  - Típicamente, detectan solo la exclusividad mutua estructural.

```
process (a,b,c,d,e,f,g,h,i,j,k,l,m,n,c1,c2)
begin
  z1 <= a + b;
  if c1 then
    z2 <= c + d;
  else
    z2 <= e + f;
    if c2 then
      z3 <= g + h;
    else
      z3 <= i + j;
    end if;
  end if;
  if not c1 then
    z4 <= k + l;
  else
    z4 <= m + n;
  end if;
end process;
```

Típicamente las herramientas consideran que las condiciones son independientes

mutex	a + b	c + d	e + f	g + h	i + j	k + l	m + n
a + b		no	no	no	no	no	no
c + d	no		si	si	si	no	no
e + f	no	si		no	no	no	no
g + h	no	si	no		si	no	no
i + j	no	si	no	si		no	no
k + l	no	no	no	no	no		si
m + n	no	no	no	no	no	si	

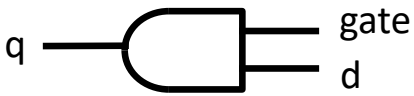
Lógica combinacional

funciones

- Una función (independientemente de su codificación) se implementa siempre como un bloque de lógica combinacional, por ello:
  - Un mismo código puede sintetizarse de manera distinta según se defina o no como función.

```
function foo( d, gate : bit ) return bit is
  variable q: bit;
begin
  if gate='1' then
    q := d;
  end if;
  return q;
end;
...
q <= foo( d, gate );
...
```

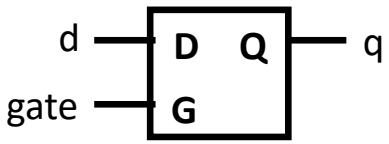
d	gate	q
0	0	0
0	1	0
1	0	0
1	1	1



Las variables de funciones son dinámicas, por lo que no retienen valores entre llamadas

Cuando una variable se crea, se inicializa por defecto al primer valor de su declaración de tipo, en el caso de bit, este valor es '0' y se conservará hasta que se asigne otro

```
process ( gate, d )
begin
  if gate='1' then
    q <= d;
  end if;
end process;
```

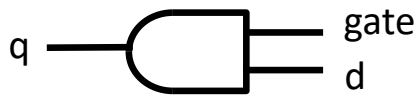


Lógica combinacional

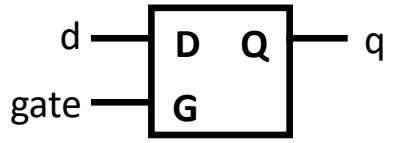
procedimientos

- Un procedimiento (según su codificación) se implementa como un bloque de lógica combinacional o un bloque de lógica secuencial.

```
procedure foo( signal d, gate : in bit, signal q : out bit ) is
  variable z: bit;
begin
  if gate='1' then
    z := d;
  end if;
  q <= z;
end;
...
foo( d, gate, q );
...
```




```
procedure foo( signal d, gate : in bit, signal q : out bit ) is
begin
  if gate='1' then
    q <= d;
  end if;
end;
...
foo( d, gate, q );
...
```



- Por ello, se recomienda encapsular 'hardware' mediante componentes.
  - Los subprogramas se reservan para encapsulamiento de funciones 'software'

J.M. Mendías
2024

Lógica combinacional

ROM

- Es común especificar lógica combinacional en forma de ROM
  - Lo que no implica que necesariamente se implemente a nivel físico como una ROM, ya que esto depende de la tecnología.

```
architecture ...;
  ...
  type romType is array (0 to ...) of std_logic_vector(... downto 0);
  constant rom : romType :=
    (  0 => "...",
       1 => "...",
       ... );
begin
  ...
  data <= rom( to_integer( unsigned( address ) ) );
  ...
end;
```

```
process (address)
begin
  case address is
    when "00...00" => data <= "...";
    when "00...01" => data <= "...";
    ...
    when others    => data <= "...";
  end case;
end process;
```

J.M. Mendías
2024

Ejemplos

Multiplexor vectorial 2 a 1

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

entity multiplexer is
  port(
    x0  : in  std_logic_vector(7 downto 0);
    x1  : in  std_logic_vector(7 downto 0);
    sel : in  std_logic;
    y   : out std_logic_vector(7 downto 0) );
end multiplexer;

architecture syn1 of multiplexer is
begin
  y <= x1 when sel='1' else x0;
end syn;
```

```
architecture syn2 of multiplexer is
begin
  process (sel, x0, x1)
  begin
    if sel='1' then
      y <= x1;
    else
      y <= x0;
    end if;
  end process;
end syn2;
```

```
architecture syn3 of multiplexer is
  signal aux : std_logic_vector(7 downto 0);
begin
  aux <= (others=>sel);
  y <= (x1 and aux) or (x0 and not aux);
end syn3;
```

J.M. Mendías
2024

Ejemplos

Multiplexor vectorial 2 a 1 genérico

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

entity multiplexer is
  generic( n : integer := 8 );
  port(
    x0  : in  std_logic_vector(n-1 downto 0);
    x1  : in  std_logic_vector(n-1 downto 0);
    sel : in  std_logic;
    y   : out std_logic_vector(n-1 downto 0) );
end multiplexer;

architecture syn of multiplexer is
begin
  process (sel, x0, x1)
  begin
    if sel='1' then
      y <= x1;
    else
      y <= x0;
    end if;
  end process;
end syn;
```

Ejemplos

Multiplexor vectorial 4 a 1 genérico

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

entity multiplexer is
  generic( n : integer := 8 );
  port(
    x0, x1, x2, x3  : in  std_logic_vector(n-1 downto 0);
    sel             : in  std_logic_vector(1 downto 0);
    y               : out std_logic_vector(n-1 downto 0) );
end multiplexer;

architecture syn of multiplexer is
begin
  process (sel, x0, x1, x2, x3)
  begin
    case sel is
      when "00"   => y <= x0;
      when "01"   => y <= x1;
      when "10"   => y <= x2;
      when others => y <= x3;
    end case;
  end process;
end syn;
```

importante el others para cubrir los valores metalógicos

Ejemplos

codificador de prioridad 8 a 3 (i)

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

entity priorityEncoder is
  port(
    x  : in  std_logic_vector(7 downto 0);
    y  : out std_logic_vector(2 downto 0);
    gs : out std_logic );
end priorityEncoder;

architecture syn1 of priorityEncoder is
begin
  process (x)
  begin
    if    x(7)='1' then y <= "111"; gs <= '1';
    elsif x(6)='1' then y <= "110"; gs <= '1';
    elsif x(5)='1' then y <= "101"; gs <= '1';
    elsif x(4)='1' then y <= "100"; gs <= '1';
    elsif x(3)='1' then y <= "011"; gs <= '1';
    elsif x(2)='1' then y <= "010"; gs <= '1';
    elsif x(1)='1' then y <= "001"; gs <= '1';
    elsif x(0)='1' then y <= "000"; gs <= '1';
    else y <= "000"; gs <= '0';
    end if;
  end process;
end syn1;
```

el proceso se activa cuando hay un evento en cualquiera de las componentes de x

x(7) se chequea en primer lugar es el más prioritario

la anidación de if especifica la prioridad de las entradas

la respuesta cuando no hay entradas activadas se especifica en la última rama else

Ejemplos

codificador de prioridad 8 a 3 (ii)

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

entity priorityEncoder is
  port(
    x  : in  std_logic_vector(7 downto 0);
    y  : out std_logic_vector(2 downto 0);
    gs : out std_logic );
end priorityEncoder;

architecture syn2 of priorityEncoder is
begin
  process (x)
  begin
    y <= "000"; gs <= '0';
    if x(0)='1' then y <= "000"; gs <= '1'; end if;
    if x(1)='1' then y <= "001"; gs <= '1'; end if;
    if x(2)='1' then y <= "010"; gs <= '1'; end if;
    if x(3)='1' then y <= "011"; gs <= '1'; end if;
    if x(4)='1' then y <= "100"; gs <= '1'; end if;
    if x(5)='1' then y <= "101"; gs <= '1'; end if;
    if x(6)='1' then y <= "110"; gs <= '1'; end if;
    if x(7)='1' then y <= "111"; gs <= '1'; end if;
  end process;
end syn2;
```

valores 'por defecto' de las salidas cuando no hay entradas activadas

la prioridad de las entradas se especifica sobreescribiendo selectivamente el valor asignado por anteriores sentencias

x(7) se chequea en último lugar es el más prioritario

Ejemplos

codificador de prioridad genérico

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

entity priorityEncoder is
  generic( n : integer := 3 );
  port(
    x  : in  std_logic_vector(2**n-1 downto 0);
    y  : out std_logic_vector(n-1 downto 0);
    gs : out std_logic );
end priorityEncoder;

architecture syn of priorityEncoder is
begin
  process (x)
  begin
    y <= (others=>'0'); gs <= '0';
    for i in x'reverse_range loop
      if x(i)='1' then
        y <= std_logic_vector(to_unsigned( i, n ));
        gs <= '1';
      end if;
    end loop;
  end process;
end syn;
```

la prioridad de las entradas se especifica sobreescribiendo selectivamente el valor asignado por anteriores sentencias

el rango de i es computable y el bucle puede desenrrollarse

expresión computable

Ejemplos

decodificador 3 a 8 (i)

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

entity decoder is
  port(
    x  : in  std_logic_vector(2 downto 0);
    en : in  std_logic;
    y  : out std_logic_vector(7 downto 0) );
end decoder;

architecture syn1 of decoder is
begin
  process (x, en)
  begin
    y <= "00000000";
    if en='1' then
      case x is
        when "000"  => y(0) <= '1';
        when "001"  => y(1) <= '1';
        when "010"  => y(2) <= '1';
        when "011"  => y(3) <= '1';
        when "100"  => y(4) <= '1';
        when "101"  => y(5) <= '1';
        when "110"  => y(6) <= '1';
        when others => y(7) <= '1';
      end case;
    end if;
  end process;
end syn1;
```

la selección se anida dentro de la habilitación
define un reuso de los cálculos intermedios
e implica una estructura en cascada

Ejemplos

decodificador genérico


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

entity decoder is
  generic( n : integer := 3 );
  port(
    x  : in  std_logic_vector(n-1 downto 0);
    en : in  std_logic;
    y  : out std_logic_vector(2**n-1 downto 0) );
end decoder;

architecture syn of decoder is
begin
  process (x, en)
  begin
    y <= (others=>'0');
    if en='1' then
      y( to_integer(unsigned(x)) ) <= '1';
    end if;
  end process;
end syn;
```

las conversiones de tipo no requieren HW ya que sólo indican cómo debe interpretarse las señales

Ejemplos

decodificador 3 a 8 (ii)

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

entity decoder is
  port(
    x  : in  std_logic_vector(2 downto 0);
    en : in  std_logic;
    y  : out std_logic_vector(7 downto 0) );
end decoder;

architecture syn2 of decoder is
begin
  y(0) <= '1' when (en='1' and x="000") else '0';
  y(1) <= '1' when (en='1' and x="001") else '0';
  y(2) <= '1' when (en='1' and x="010") else '0';
  y(3) <= '1' when (en='1' and x="011") else '0';
  y(4) <= '1' when (en='1' and x="100") else '0';
  y(5) <= '1' when (en='1' and x="101") else '0';
  y(6) <= '1' when (en='1' and x="110") else '0';
  y(7) <= '1' when (en='1' and x="111") else '0';
end syn2;
```

cada salida se asigna por separado

no se comparte la expresión de habilitación
luego implica una estructura paralela

Ejemplos

decodificador genérico

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

entity decoder is
  generic( n : integer := 3 );
  port(
    x  : in  std_logic_vector(n-1 downto 0);
    en : in  std_logic;
    y  : out std_logic_vector(2**n-1 downto 0) );
end decoder;

architecture syn of decoder is
begin
  for i in y'range generate
  begin
    y(i) <= '1' when (en='1' and x=unsigned(x)) else '0';
  end generate;
end syn;
```

Ejemplos

sumador genérico (i)

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

entity adder is
  generic( n : integer := 8 );
  port(
    x : in  std_logic_vector(n-1 downto 0);
    y : in  std_logic_vector(n-1 downto 0);
    s : out std_logic_vector(n-1 downto 0) );
end adder;

library ieee;
use ieee.numeric_std.all;

architecture syn1 of adder is
begin
  s <= std_logic_vector(unsigned(x) + unsigned(y));
end syn1;
```

```
library ieee;
use ieee.std_logic_unsigned.all;

architecture syn2 of adder is
begin
  s <= x + y;
end syn2;
```

Según el paquete usado la expresión puede ser más o menos compacta. El HW implementado es el mismo

Ejemplos

sumador genérico (ii)

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

entity adder is
  generic( n : integer := 8 );
  port(
    x    : in  std_logic_vector(n-1 downto 0);
    y    : in  std_logic_vector(n-1 downto 0);
    cin  : in  std_logic;
    s    : out std_logic_vector(n-1 downto 0);
    cout : out std_logic );
end adder;

architecture syn of adder is
  signal cinAux : unsigned(0 downto 0);
  signal temp   : unsigned(n downto 0);
begin
  cinAux <= (0=>cin);
  temp <= to_unsigned( to_integer(unsigned(x)) + to_integer(unsigned(y))
    + to_integer(cinAux), n+1);
  s <= std_logic_vector(temp(n-1 downto 0));
  cout <= std_logic(temp(n));
end syn;
```

para obtener el carry out es necesario aumentar en 1 el número de bits

Lógica secuencial

- La lógica secuencial puede especificarse mediante:
  - Procesos sin sentencias **`wait`** tales que:
    - Existan señales o variables escritas por el proceso que no se asignen bajo alguna condición de ejecución del mismo.
    - Serán expresiones de flanco o nivel referidas a una señal de reloj las que regulen las condiciones de asignación.
    - Pueden especificar lógica secuencial con temporización (por flanco o nivel) e inicialización (síncrona o asíncrona) de cualquier tipo.
    - Aunque la lista de sensibilidad se ignora, para asegurar la coherencia entre simulación y síntesis, esta debe ser parcial (formada por aquellas señales que disparan cambios de estado: reloj, reset, clear, etc ...).
  - Procesos con sentencias **`wait`**
    - Debe tener la forma de **`wait until`** referida a una única señal de reloj.
    - Típicamente la sentencia **`wait`** debe ser la primera o la última del proceso.
    - Solo puede especificarse lógica secuencial con temporización por flanco y reset síncrono.
    - El standard VHDL define más casos de uso pero no los soporta Xilixn ISE.
  - Asignaciones concurrentes condicionales incompletas (sin rama **`else`**)

J.M. Mendías
2024

Lógica secuencial

ejemplos elementales (i)

```
process
begin
  wait until rising_edge(clk);
  q <= d;
end process;
```

Flip-flop D

```
process (clk)
begin
  if rising_edge(clk) then
    q <= d;
  end if;
end process;
```

Latch D

```
process (gate, d)
begin
  if gate='1' then
    q <= d;
  end if;
end process;
```

Flip-flop D con reset asíncrono

```
process (rst, clk)
begin
  if rst='1' then
    q <= '0';
  elsif rising_edge(clk) then
    q <= d;
  end if;
end process;
```

Latch D con reset asíncrono

```
process (rst, gate, d)
begin
  if rst='1' then
    q <= '0';
  elsif gate='1' then
    q <= d;
  end if;
end process;
```

Flip-flop D con reset síncrono

```
process
begin
  wait until rising_edge(clk);
  if rst='1' then
    q <= '0';
  else
    q <= d;
  end if;
end process;
```

```
process (clk)
begin
  if rising_edge(clk) then
    if rst='1' then
      q <= '0';
    else
      q <= d;
    end if;
  end if;
end process;
```

Lógica secuencial

ejemplos elementales (ii)

- Existen expresiones de flanco de subida equivalentes
  - Dispositivos a flanco de bajada tienen expresiones análogas con polaridad invertida.

```
process
begin
  wait until rising_edge(clk);
  q <= d;
end process;
```

```
process (clk)
begin
  if rising_edge(clk) then
    q <= d;
  end if;
end process;
```

```
process
begin
  wait until clk'event and clk='1';
  q <= d;
end process;
```

```
process (clk)
begin
  if clk'event and clk='1' then
    q <= d;
  end if;
end process;
```

```
process
begin
  wait until not clk'stable and clk='1';
  q <= d;
end process;
```

```
process (clk)
begin
  if not clk'stable and clk='1' then
    q <= d;
  end if;
end process;
```

```
process
begin
  wait until clk='1';
  q <= d;
end process;
```

Flip-flop D

Lógica secuencial

ejemplos elementales (iii)

- Las asignaciones concurrentes condicionales permiten también especificar lógica secuencial
  - Debido a la equivalencia entre procesos y asignaciones concurrentes

Flip-flop D

```
q <= d when rising_edge(clk);
```

```
process (clk, d)
begin
  if rising_edge(clk) then
    q <= d;
  end if;
end process;
```

Latch D

```
q <= d when gate='1';
```

```
with clk select
  z <= d when '1',
       z when others;
```

Flip-flop D con reset asíncrono

```
q <= '0' when rst='1' else
      d  when rising_edge(clk);
```

Latch D con reset asíncrono

```
q <= '0' when rst='1' else
      d  when gate='1';
```

Flip-flop D con reset síncrono

```
q <= d  when rising_edge(clk) else
    '0' when rst='1';
```

- No obstante, se recomienda el uso de procesos.

Lógica secuencial

expresiones de flanco (i)

- Una sentencia **`if`** que tenga por condición una especificación de flanco no puede tener rama **`else`**.
  - De hecho, en sentencias **`if-then-elsif`** la especificación de flanco sólo podrá ser la condición del último **`if`** (que no podrá tener rama **`else`**).

```
process (clk)
begin
  if rising_edge(clk) then
    q <= ...;
  else
    ...
  end if;
end process;
```

✗

```
process (rst, clk)
begin
  if rst='1' then
    q <= ...;
  elsif rising_edge(clk) then
    q <= ...;
  else
    ...
  end if;
end process;
```

✗

  - En caso contrario la acción especificada debería realizarse en todo momento menos en el preciso instante en el que el reloj cambia, cosa sin sentido en hardware.

J.M. Mendías
2024

Lógica secuencial

expresiones de flanco (ii)

- Una especificación de flanco no debe utilizarse como operando.
  - Aunque su presencia en VHDL pueda evaluarse, un evento HW es el punto de división entre dos valores estables diferentes; por tanto, al no ser un valor en sí mismo, no puede utilizarse como argumento en un cálculo.

```
process (rst, clk)
begin
  if rising_edge(clk) or rst='1' then
    q <= ...;
  end if;
end process;
```

✘

```
process (clk)
begin
  if not rising_edge(clk) then
    q <= ...;
  end if;
end process;
```

✘

  - En algunos casos se permite, pero no se recomienda:

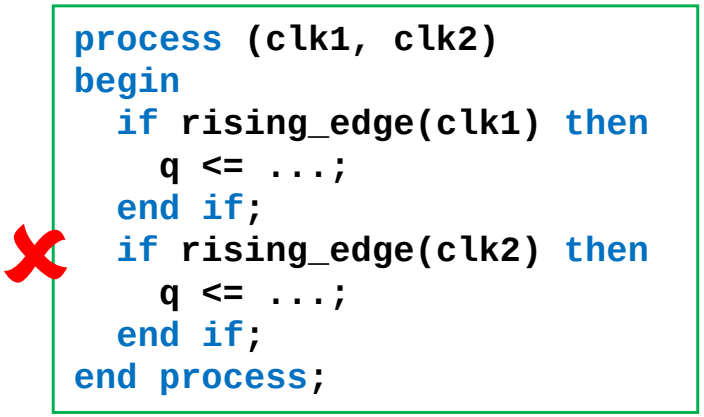
```
process( rst, clk )
begin
  if rising_edge(clk) and enable='1' then
    q <= ...;
  end if;
end process;
```

✓

Lógica secuencial

expresiones de flanco (ii)

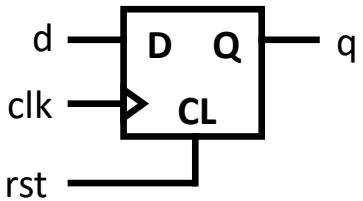
- Una señal puede estar afectada solo por una especificación de flanco.
  - En caso contrario se estaría especificando HW secuencial sensible a varios relojes.



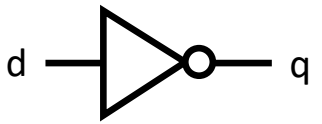
```
process (clk1, clk2)
begin
  if rising_edge(clk1) then
    q <= ...;
  end if;
  if rising_edge(clk2) then
    q <= ...;
  end if;
end process;
```

- Aún cuando una señal esté afectada por una especificación por flanco, el orden de asignación secuencial de la misma sigue importando.

```
process (rst, clk)
begin
  if rising_edge(clk) then
    q <= d;
  end if;
  if rst='1' then
    q <= '0';
  end if;
end process;
```



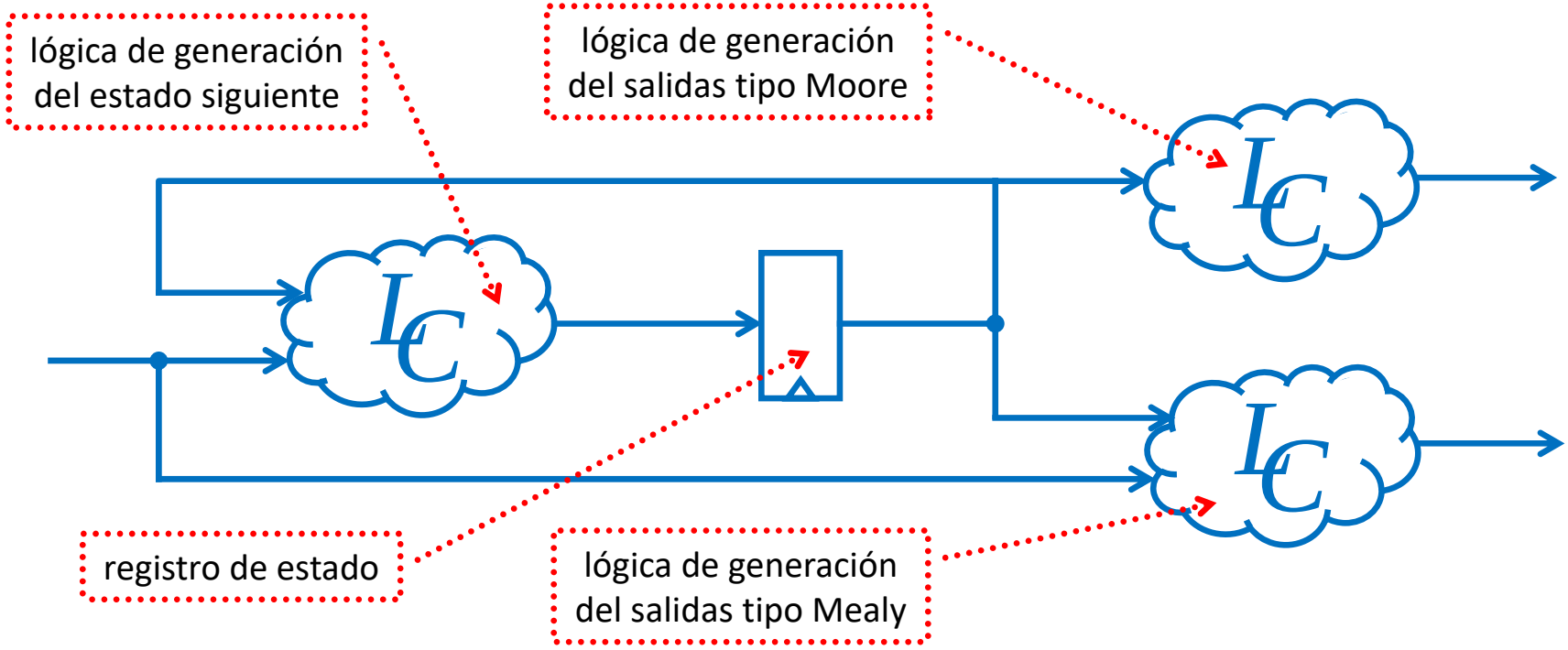

```
process (rst, clk)
begin
  if rising_edge(clk) then
    q <= d;
  end if;
  q <= not( d );
end process;
```



Lógica secuencial

FSM (i)

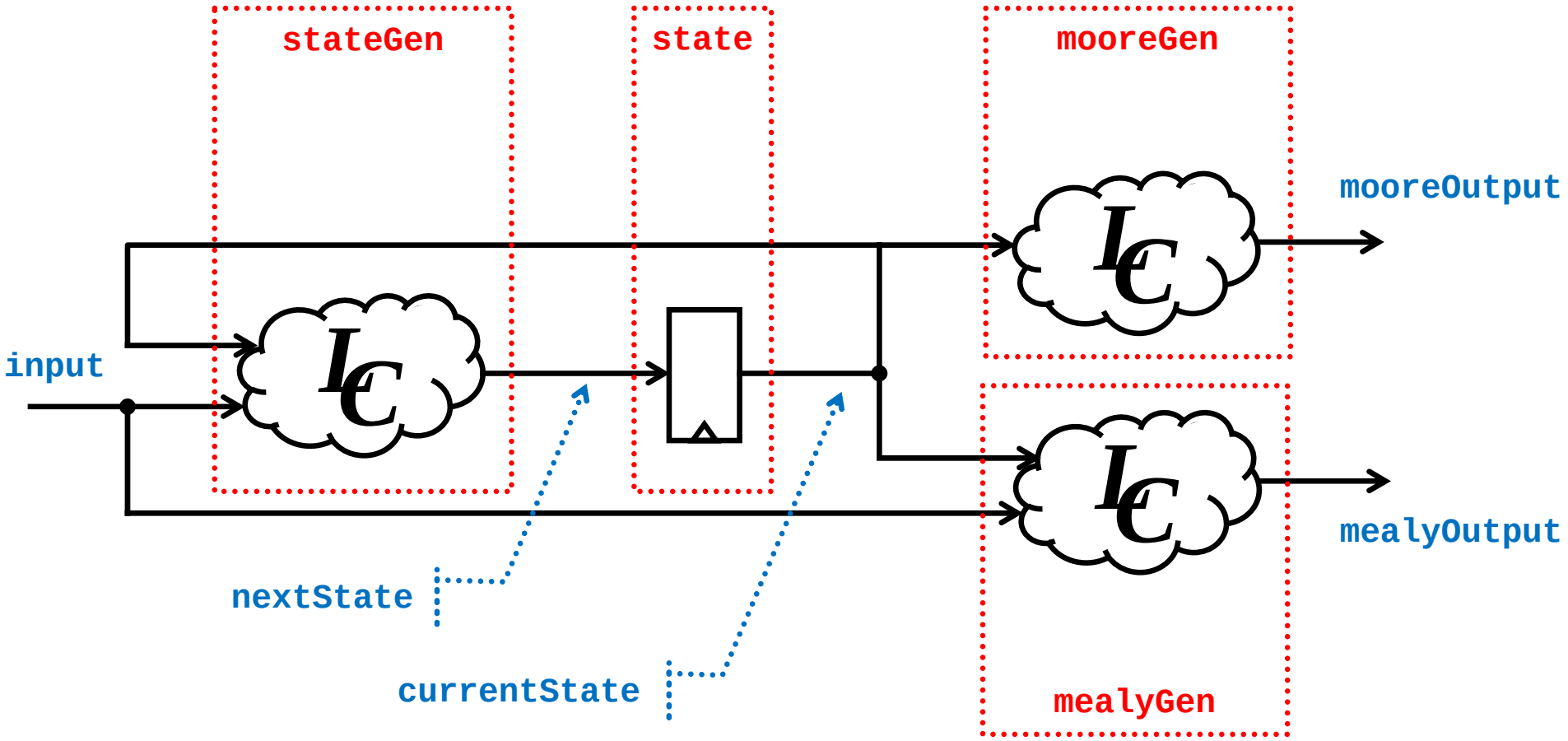
- Todo sistema secuencial puede describirse usando 4 elementos:
  - Un registro de estado.
  - Una red combinacional que calcula el estado siguiente.
  - Una red combinacional que calcula las salidas Moore (dependientes solo del estado).
  - Una red combinacional que calcula las salidas Mealy (dependientes del estado y de la entrada.



Lógica secuencial

FSM (ii)

- Primera alternativa
  - Usar un proceso para especificar cada elemento.



Lógica secuencial

FSM (iii)

asignar valores por defecto asegura lógica combinacional

```
stateGen:
process (currentState, input)
begin
  nextState <= currentState;
  case currentState is
    when ... =>
      if (input ...) then
        nextState <= ...;
      elsif (input ...) then
        ...
      else
        ...
      end if;
    ...
  end case;
end process;
```

```
mealyGen:
process (currentState, input)
begin
  mealyOutput <= ...;
  case currentState is
    when ... =>
      if (input ...) then
        mealyOutput <= ...;
      elsif (input ...) then
        ...
      else
        ...
      end if;
    ...
  end case;
end process;
```

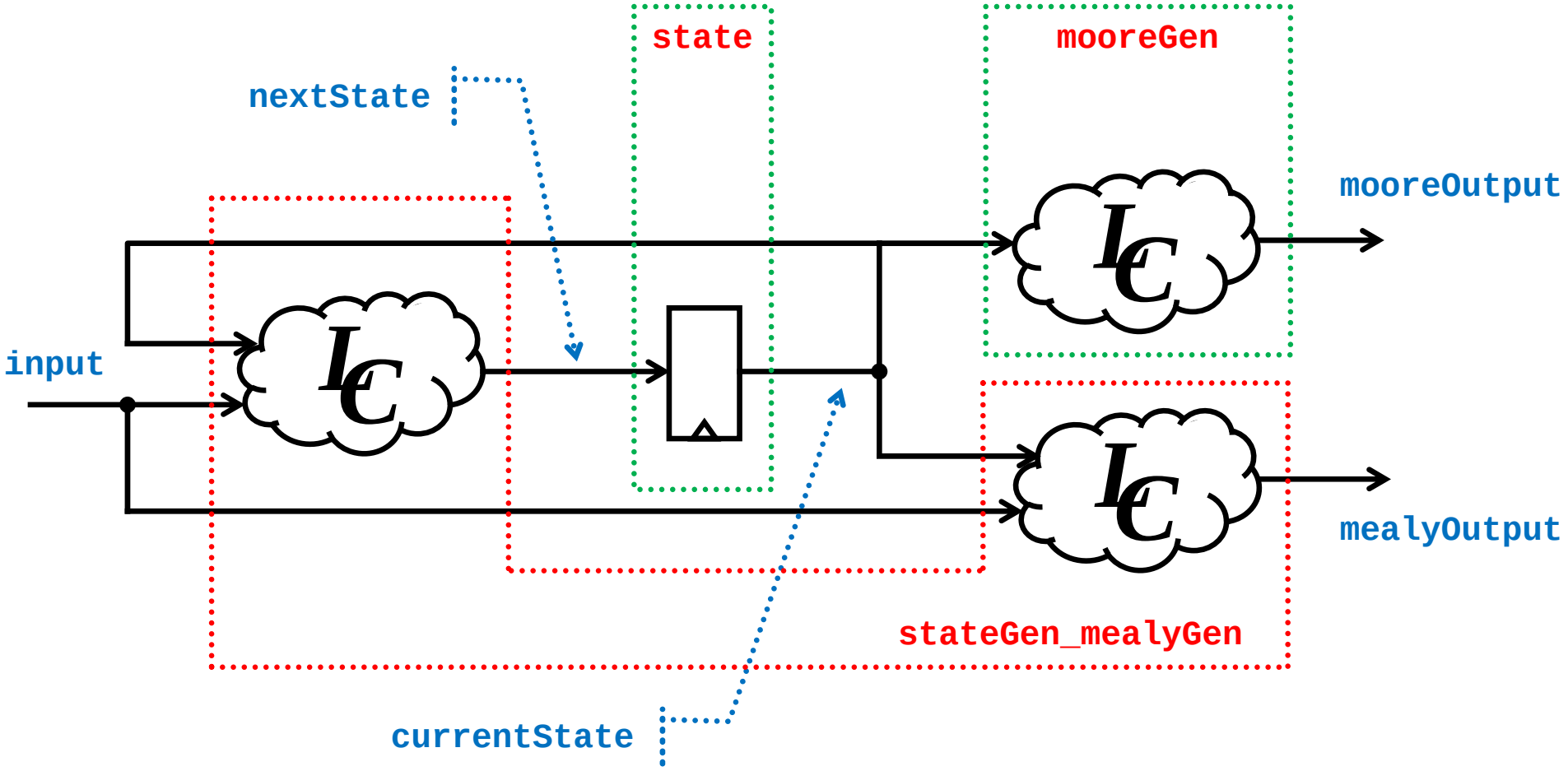
```
state:
process (rst, clk)
begin
  if rst='1' then
    currentState <= ...;
  elsif risign_edge(clk) then
    currentState <= nextState;
  end if;
end process;
```

```
mooreGen:
process (currentState)
begin
  mooreOutput <= ...;
  case currentState is
    when ... =>
      mooreOutput <= ...;
    ...
  end case;
end process;
```

Lógica secuencial

FSM (iv)

- Segunda alternativa
  - Agrupar en un mismo proceso el cálculo del estado siguiente y el cálculo de las salidas tipo Mealy dado que sus estructuras VHDL son equivalentes.



J.M. Mendías
2024

Lógica secuencial

FSM (v)

```
stateGen_mealyGen:
process (currentState, input)
begin
  nextState <= currentState;
  mealyOutput <= ...;
  case currentState is
    when ... =>
      if (input ...) then
        nextState <= ...;
        mealyOutput <= ...;
      elsif (input ...) then
        ...
      else
        ...
      end if;
    ...
  end case;
end process;
```

```
state:
process (rst, clk)
begin
  if rst='1' then
    currentState <= ...;
  elsif risign_edge(clk) then
    currentState <= nextState;
  end if;
end process;
```

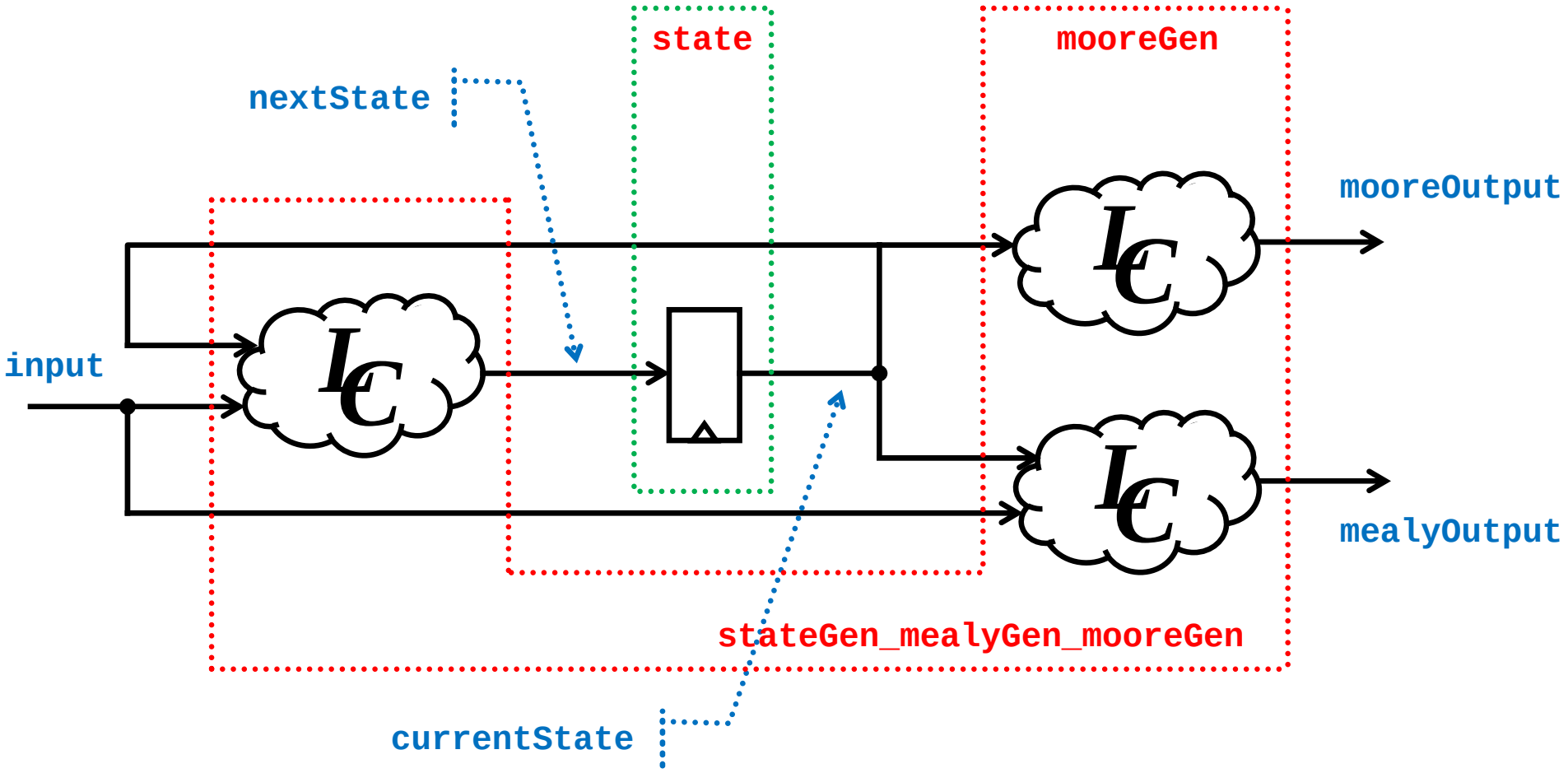
```
mooreGen:
process (currentState)
begin
  mooreOutput <= ...;
  case currentState is
    when ... =>
      mooreOutput <= ...;
    ...
  end case;
end process;
```

J.M. Mendías
2024

Lógica secuencial

FSM (vi)

- Tercera alternativa:
  - Agrupar en un único proceso toda la lógica combinacional.



J.M. Mendías
2024

Lógica secuencial

FSM (vii)

```
stateGen_mealyGen_mooreGen:
process (currentState, input)
begin
  nextState <= currentState;
  mealyOutput <= ...;
  mooreOutput <= ...;
  case currentState is
    when ... =>
      mooreOutput <= ...
      if (input ...) then
        nextState <= ...;
        mealyOutput <= ...;
      elsif (input ...) then
        ...
      else
        ...
      end if;
    ...
  end case;
end process;
```

```
state:
process (rst, clk)
begin
  if rst='1' then
    currentState <= ...;
  elsif risign_edge(clk) then
    currentState <= nextState;
  end if;
end process;
```

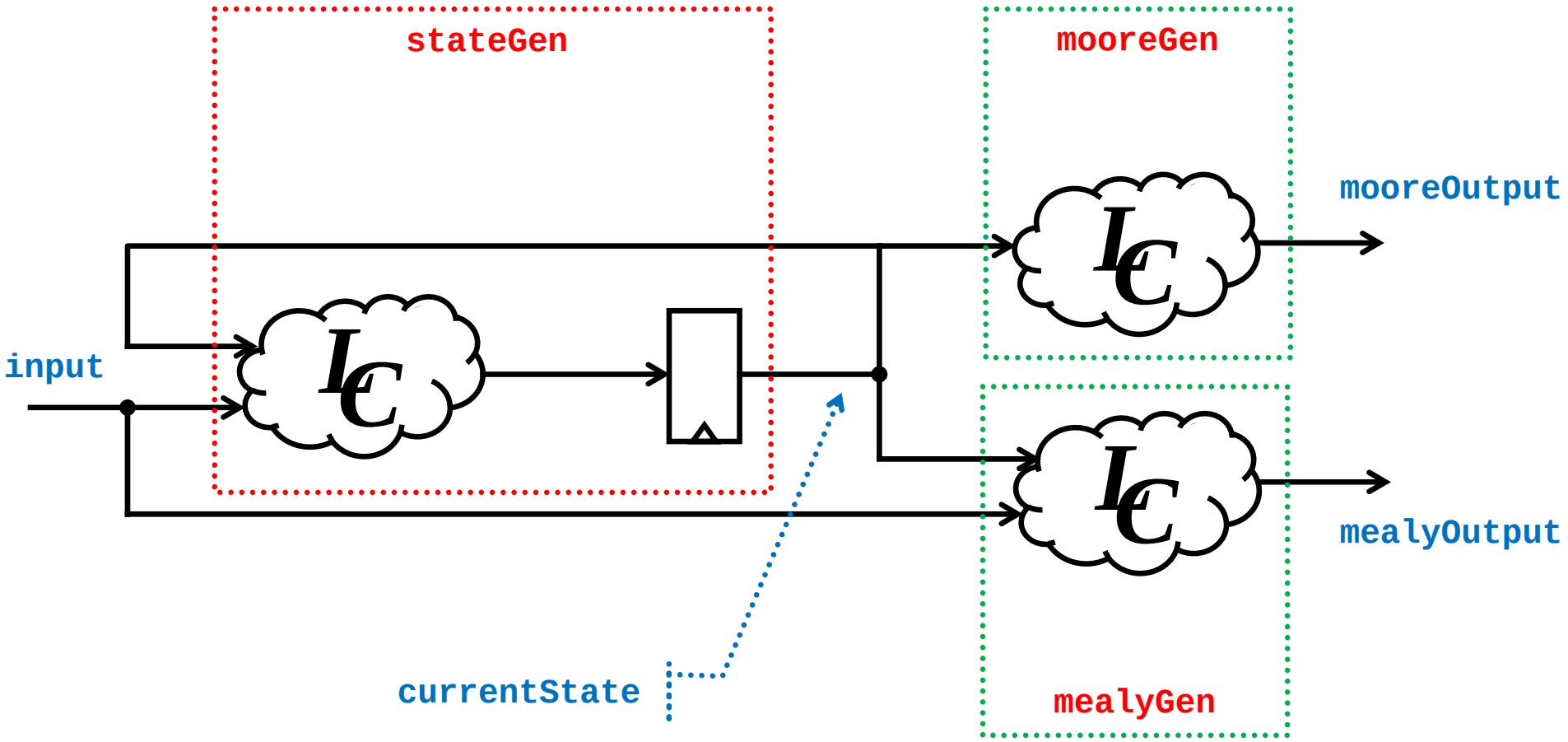
las salidas tipo Moore sólo dependen del estado

el estado siguiente y las salidas tipo Mealy dependen del estado y de las entradas

Lógica secuencial

FSM (viii)

- Cuarta alternativa:
  - Hacer el cálculo del estado siguiente local al proceso que almacena el estado, dado que el cambio de estado solo se hace efectivo tras eventos del reloj.



Lógica secuencial

FSM (ix)

```
stateGen_state:
process (rst, clk)
begin
  if rst='1' then
    currentState <= ...;
  elsif risign_edge(clk) then
    case currentState is
      when ... =>
        if (input ...) then
          currentState <= ...;
        elsif (input ...) then
          ...
        else
          ...
        end if;
      ...
    end case;
  end if;
end process;
```

ahora no es necesaria la asignación de un valor por defecto a currentState. Si no se asigna conserva su valor

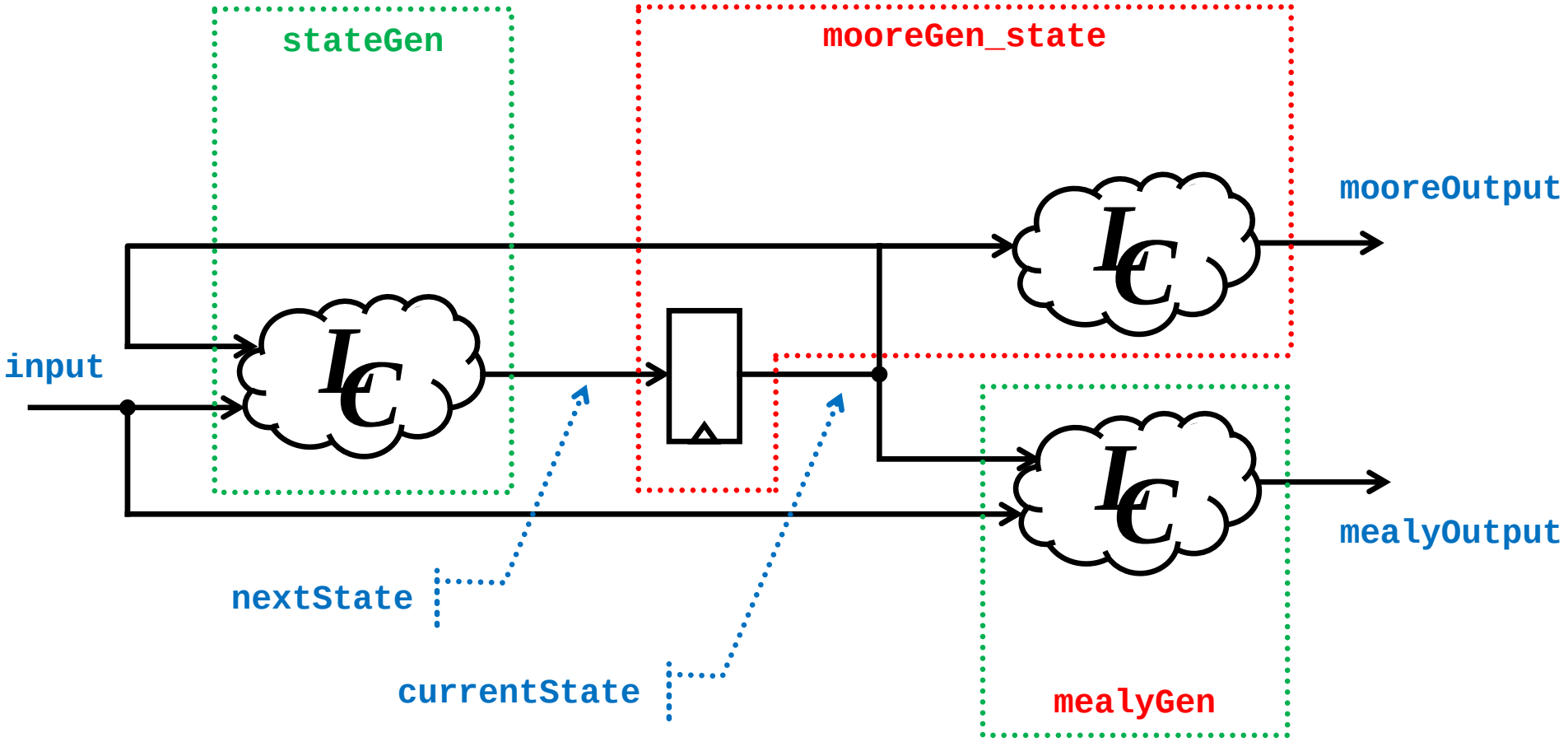
```
mealyGen:
process (currentState, input)
begin
  mealyOutput <= ...;
  case currentState is
    when ... =>
      if (input ...) then
        mealyOutput <= ...;
      elsif (input ...) then
        ...
      else
        ...
      end if;
    ...
  end case;
end process;
```

```
mooreGen:
process (currentState)
begin
  mooreOutput <= ...;
  case currentState is
    when ... =>
      mooreOutput <= ...;
    ...
  end case;
end process;
```

Lógica secuencial

FSM (x)

- Quinta alternativa:
  - Agrupar el proceso que almacena el estado con el que calcula las salidas tipo Moore.



Lógica secuencial

FSM (xi)

```
mooreGen_state:
process (rst, clk, currentState)
begin
  mooreOutput <= ...;
  case currentState is
    when ... =>
      mooreOutput <= ...
    ...
  end case;
  if rst='1' then
    currentState <= ...;
  elsif rising_edge(clk) then
    currentState <= nextState;
  end if;
end process;
```

```
stateGen:
process (currentState, input)
begin
  nextState <= currentState;
  case currentState is
    when ... =>
      if (input ...) then
        nextState <= ...;
      elsif (input ...) then
        ...
      else
        ...
      end if;
    ...
  end case;
end process;
```

```
mealyGen:
process (currentState, input)
begin
  mealyOutput <= ...;
  case currentState is
    when ... =>
      if (input ...) then
        mealyOutput <= ...;
      elsif (input ...) then
        ...
      else
        ...
      end if;
    ...
  end case;
end process;
```

necesaria para que simule correctamente

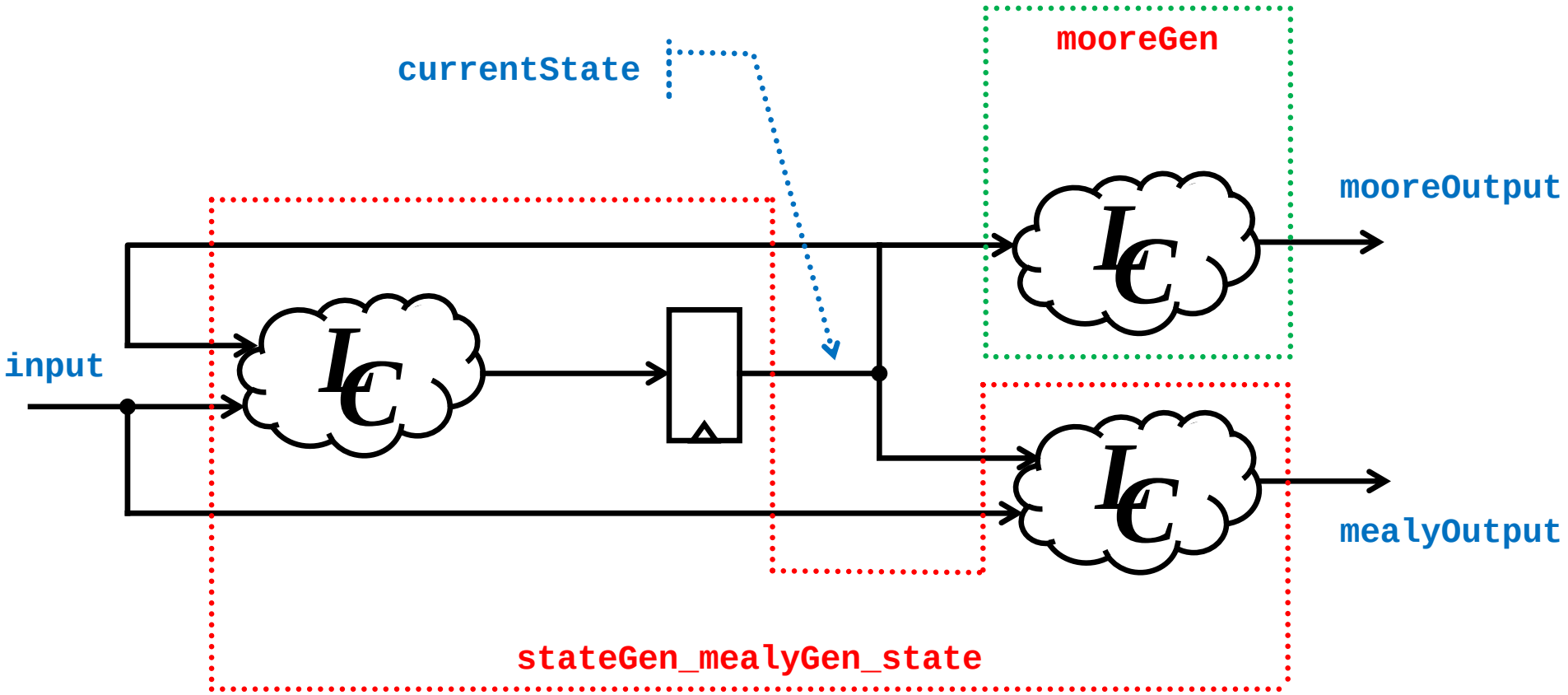
el cálculo de la salida se realiza incondicionalmente, si se anidara dentro del if, se interpretaría que dichas salidas hay también que almacenarlas (aunque sólo cambien tras el cambio de estado)

J.M. Mendías
2024

Lógica secuencial

FSM (xii)

- Sexta alternativa



J.M. Mendías
2024

Lógica secuencial

FSM (xiii)

```
stateGen_MealyGen_state:
process (rst, clk, currentState, input)
begin
  mealyOutput <= ...;
  case currentState is
    when ... =>
      if (input ...) then
        mealyOutput <= ...;
      elsif (input ...) then
        ...
      else
        ...
      end if;
    ...
  end case;
  if rst='1' then
    currentState <= ...;
  elsif risign_edge(clk) then
    case currentState is
      when ... =>
        if (input ...) then
          currentState <= ...;
        elsif (input ...) then
          ...
        else
          ...
        end if;
      ...
    end case;
  end if;
end process;
```

```
mooreGen:
process (currentState)
begin
  mooreOutput <= ...;
  case currentState is
    when ... =>
      mooreOutput <= ...;
  end case;
end process;
```

necesaria para que simule correctamente

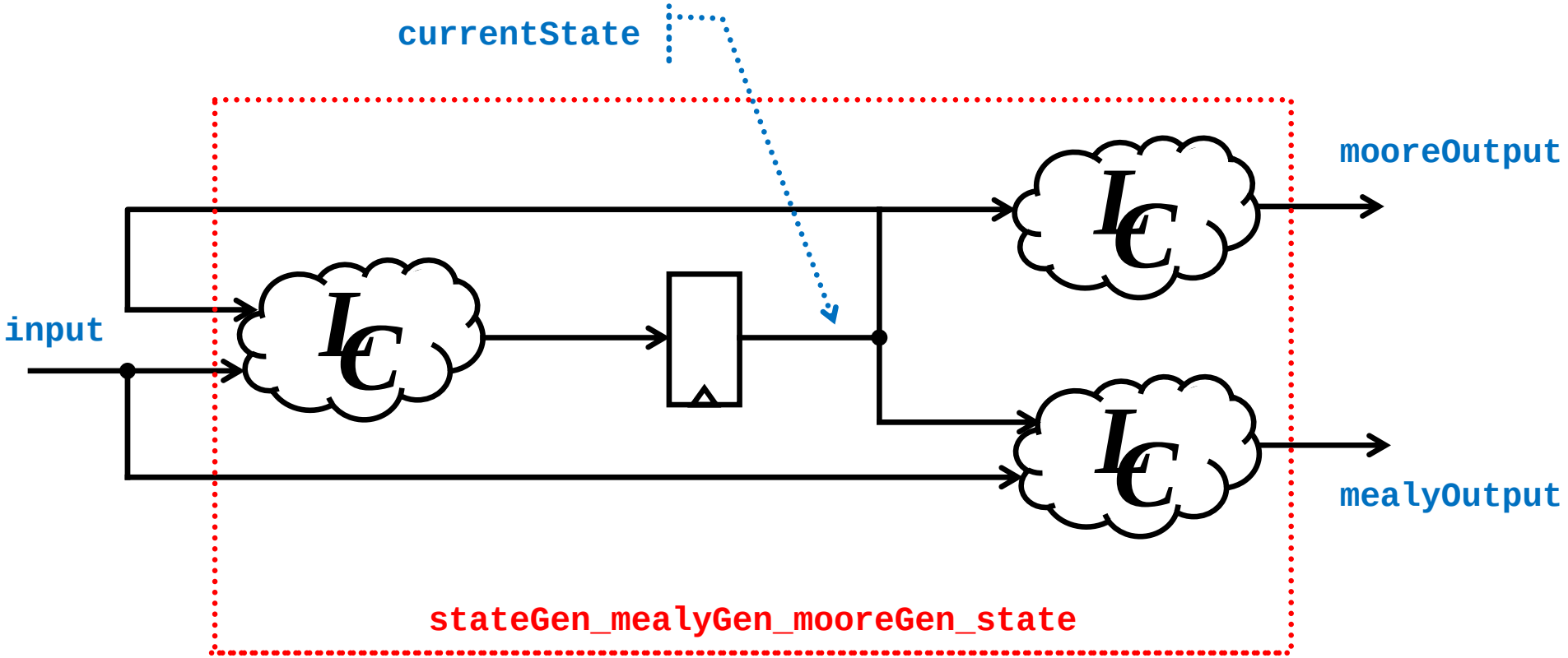
necesaria para que simule correctamente

el cálculo de la salida se realiza incondicionalmente

Lógica secuencial

FSM (xiv)

- Séptima alternativa:
  - Especificar todo usando un único proceso



Lógica secuencial

FSM (xv)

```
stateGen_mealyGen_mooreState_state:
process (rst, clk, currentState, input )
begin
  mealyOutput <= ...;
  mooreOutput <= ...;
  case currentState is
    when ... =>
      mooreOutput <= ...;
      if (input ...) then
        mealyOutput <= ...;
      elsif (input ...) then
        ...
      else
        ...
      end if;
    ...
  end case;
  if rst='1' then
    currentState <= ...;
  elsif risign_edge(clk) then
    case currentState is
      when ... =>
        if (input ...) then
          currentState <= ...;
        elsif (input ...) then
          ...
        else
          ...
        end if;
      ...
    end case;
  end if;
end process;
```

el cálculo de la salida se realiza incondicionalmente

Aunque resulte poco intuitivo, este es el método fiable de especificación de HW a nivel RT más abstracto, existen otros pero son difíciles de controlar

J.M. Mendías
2024

Lógica secuencial

FSM (xvi)

```
stateGen_mealyGen_mooreState_state:
process (rst, clk, input )
  variable currentState : ...;
begin
  mealyOutput <= ...;
  mooreOutput <= ...;
  case currentState is
    when ... =>
      mooreOutput <= ...;
      if (input ...) then
        mealyOutput <= ...;
      elsif (input ...) then
        ...
      else
        ...
      end if;
    ...
  end case;
  if rst='1' then
    currentState := ...;
  elsif risign_edge(clk) then
    case currentState is
      when ... =>
        if (input ...) then
          currentState := ...;
        elsif (input ...) then
          ...
        else
          ...
        end if;
      ...
    end case;
  end if;
end process;
```

Adicionalmente, dado que el estado actual es local al proceso, esta señal se puede sustituir por variable

En este caso se estrictamente necesario respetar el orden de los bloques, de manera que la asignación de currentState sea posterior a su lectura

Lógica secuencial

FSM (xvii)

```
stateGen_mealyGen_mooreState_state:
process (rst, clk, input )
  variable currentState : ...;
begin
  mealyOutput <= ...;
  mooreOutput <= ...;
  case currentState is
    when ... =>
      mooreOutput <= ...;
      if (input ...) then
        mealyOutput <= ...;
      elsif (input ...) then
        ...
      else
        ...
      end if;
    ...
  end case;
  if rst='1' then
    currentState := ...;
  elsif risign_edge(clk) then
    case currentState is
      when ... =>
        if (input ...) then
          currentState := ...;
        elsif (input ...) then
          ...
        else
          ...
        end if;
      ...
    end case;
  end if;
end process;
```

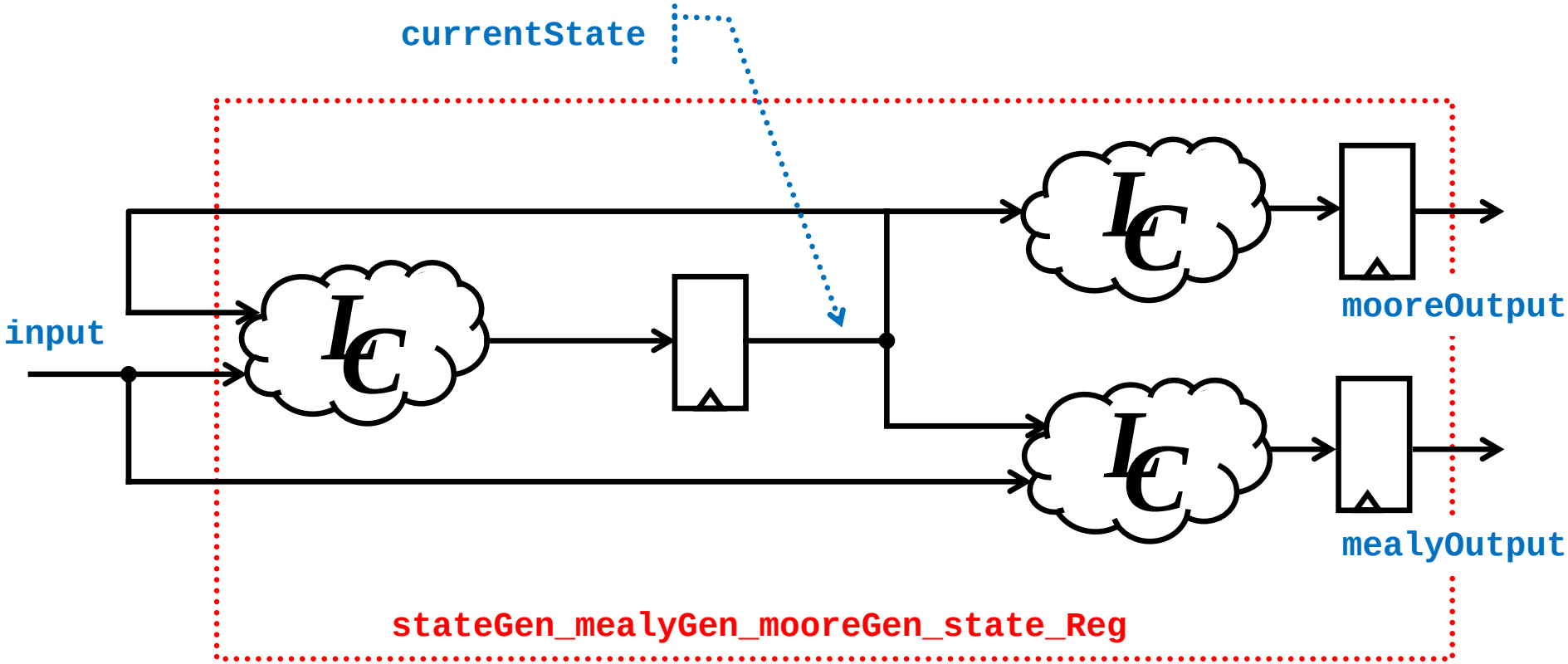
IMPORTANTE: no son equivalentes

```
stateGen_mealyGen_mooreState_state_Reg:
process (rst, clk, input )
  variable currentState : ...;
begin
if rst='1' then
    currentState := ...;
  elsif risign_edge(clk) then
    case currentState is
      when ... =>
        mooreOutput <= ...;
        if (input ...) then
           mealyOutput <= ...;
           currentState := ...;
        elsif (input ...) then
          ...
        else
          ...
        end if;
      ...
    end case;
  end if;
end process;
```

Lógica secuencial

FSM (xviii)

- Octava alternativa:
  - FSM con salidas registradas.



Lógica secuencial

RAM (i)

- Es común especificar lógica secuencial en forma de RAM
  - Lo que no implica que necesariamente se implemente a nivel físico como una RAM, ya que esto depende de la tecnología.
  - De hecho, cuando la tecnología objetivo dispone de RAM físicas, lo más práctico es instanciarlas como componentes.

RAM con lectura y escritura asíncrona (puertos de datos separados)

```
architecture ...;
  ...
  type ramType is array (0 to ...) of std_logic_vector(... downto 0);
  signal ram : ramType;
begin
  ...
  process (we, dataIn, address)
  begin
    if we='1' then
      ram( to_integer( unsigned( address ) ) ) <= dataIn;
    end if;
  end process;
  dataOut <= ram( to_integer( unsigned( address ) ) ) ;
  ...
end;
```

J.M. Mendías
2024

Lógica secuencial

RAM (ii)

RAM con lectura y escritura asíncrona (puerto de datos único)

```
process (we, data)
begin
  if we='1' then
    ram( to_integer( unsigned( address ) ) ) <= data;
  end if;
end process;
data <= ram( to_integer( unsigned( address ) ) ) when re='1' else (others => 'Z');
```

RAM con lectura asíncrona y escritura síncrona (puerto de datos separado)

```
process (clk)
begin
  if rising_edge(clk) then
    if we='1' then
      ram( to_integer( unsigned( address ) ) ) <= dataIn;
    end if;
  end if;
end process;
dataOut <= ram( to_integer( unsigned( address ) ) );
```

RAM con lectura y escritura síncrona (puerto de datos separado)

```
process (clk)
begin
  if rising_edge(clk) then
    if we='1' then
      ram( to_integer( unsigned( address ) ) ) <= dataIn;
    end if;
    dataOut <= ram( to_integer( unsigned( address ) ) );
  end if;
end process;
```

la RAM dispone de un registro a su salida que retrasa un ciclo la lectura.

Ejemplos

registro genérico

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

entity reg is
  generic( n : integer := 8 );
  port(
    rst, clk, ld : in  std_logic;
    din          : in  std_logic_vector(n-1 downto 0);
    dout         : out std_logic_vector(n-1 downto 0) );
end reg;

architecture syn1 of reg is
  signal cs, ns : std_logic_vector(n-1 downto 0);
begin
  process (rst, clk)
  begin
    if rst='1' then
      cs <= (others=>'0');
    elsif rising_edge(clk) then
      cs <= ns;
    end if;
  end process;
  ns <= din when ld='1'
        else cs;
  dout <= cs;
end syn1;
```

```
architecture syn2 of reg is
begin
  process (rst, clk)
  begin
    if rst='1' then
      dout <= (others=>'0');
    elsif rising_edge(clk) then
      if ld='1' then
        dout <= din;
      end if;
    end if;
  end process;
end syn2;
```

Ejemplos

registro genérico con salida en alta impedancia

```
library ieee; use ieee.std_logic_1164.all;
entity triStateReg is
  generic( n : integer := 8 );
  port(
    rst, clk, ld, en : in  std_logic;
    din              : in  std_logic_vector( n-1 downto 0 );
    dout             : out std_logic_vector( n-1 downto 0 ) );
end triStateReg;
architecture syn1 of triStateReg is
  signal cs, ns : std_logic_vector(n-1 downto 0);
begin
  process (rst, clk)
  begin
    if rst='1' then
      cs <= (others=>'0');
    elsif rising_edge(clk) then
      cs <= ns;
    end if;
  end process;
  ns <= din when ld='1'
        else cs;
  dout <= cs when en='1'
          else (others=>'Z');
end syn1;
```

la señal cs se necesita para distinguir el estado que puede inicializarse y cargarse de modo independiente a la salida que puede desabilitarse

```
architecture syn2 of triStateReg is
  signal cs : std_logic_vector(n-1 downto 0);
begin
  process (rst, clk, en, cs)
  begin
    dout <= (others=>'Z');
    if en='1' then
      dout <= cs;
    end if;
    if rst='1' then
      cs <= (others=>'0');
    elsif rising_edge(clk) then
      if ld='1' then
        cs <= din;
      end if;
    end if;
  end process;
end syn2;
```

Ejemplos

registro de desplazamiento genérico

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

entity shiftReg is
  generic( n : integer := 8 );
  port(
    rst, clk, sht :  in std_logic;
    din           :  in std_logic;
    dout          : out std_logic_vector(n-1 downto 0) );
end shiftReg;

architecture syn1 of shiftReg is
  signal cs, ns : std_logic_vector(n-1 downto 0);
begin
  process (rst, clk)
  begin
    if rst='1' then
      cs <= (others=>'0');
    elsif rising_edge(clk) then
      cs <= ns;
    end if;
  end process;
  process (cs, sht, din)
  begin
    ns <= cs;
    if sht='1' then
      for i in ns'high downto ns'low+1
      loop
        ns(i) <= cs(i-1);
      end loop;
      ns(0) <= din;
    end if;
  end process;
  dout <= cs;
end syn1;
```

```vhdl
architecture syn2 of shiftReg is
  signal cs : std_logic_vector(n-1 downto 0);
begin
  process (rst, clk)
  begin
    if rst='1' then
      cs <= (others=>'0');
    elsif rising_edge(clk) then
      if sht='1' then
        for i in cs'high downto cs'low+1
        loop
          cs(i) <= cs(i-1);
        end loop;
        cs(0) <= din;
      end if;
    end if;
  end process;
  dout <= cs;
end syn2;
```

J.M. Mendias
2024

Ejemplos

contador ascendente genérico (i)

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

entity counter is
  generic( n : integer := 8 );
  port(
    rst, clk, ld, ce :  in std_logic;
    din              :  in std_logic_vector(n-1 downto 0);
    tc               : out std_logic;
    dout             : out std_logic_vector(n-1 downto 0));
end counter;

architecture syn1 of counter is
  signal cs, ns : unsigned(n-1 downto 0);
begin
  process (rst, clk)
  begin
    if rst='1' then cs <= (others=>'0');
    elsif rising_edge(clk) then cs <= ns;
    end if;
  end process;
  process (ld, ce, din)
  begin
    if ld='1' then ns <= unsigned(din);
    elsif ce='1' then ns <= cs + 1;
    else ns <= cs;
    end if;
  end process;
  dout <= std_logic_vector(cs);
  tc <= '1' when ce='1' and cs=2**n-1 else '0';
end syn1;
```

J.M. Mendías
2024

Ejemplos

contador ascendente genérico (ii)

```vhdl
architecture syn2 of counter is
  signal cs : unsigned(n-1 downto 0);
begin
  process (rst, clk, ce, cs)
  begin
    dout <= std_logic_vector(cs);
    if ce='1' and cs=2**n-1 then
      tc <= '1';
    else
      tc <= '0';
    end if;
    if rst='1' then
      cs <= (others=>'0');
    elsif rising_edge(clk) then
      if ld='1' then
        cs <= unsigned(din);
      elsif ce='1' then
        cs <= cs + 1;
      end if;
    end if;
  end process;
end syn2;
```

```vhdl
architecture syn3 of counter is
  signal cs : integer;
begin
  process (rst, clk, ce, cs)
  begin
    dout <= std_logic_vector(to_unsigned(cs,n));
    if ce='1' and cs=2**n-1 then
      tc <= '1';
    else
      tc <= '0';
    end if;
    if rst='1' then
      cs <= 0;
    elsif rising_edge(clk) then
      if ld='1' then
        cs <= to_integer(unsigned(din));
      elsif ce='1' then
        cs <= cs + 1;
      end if;
    end if;
  end process;
end syn3;
```

Ejemplos

contador ascendente modulo-máximo genérico

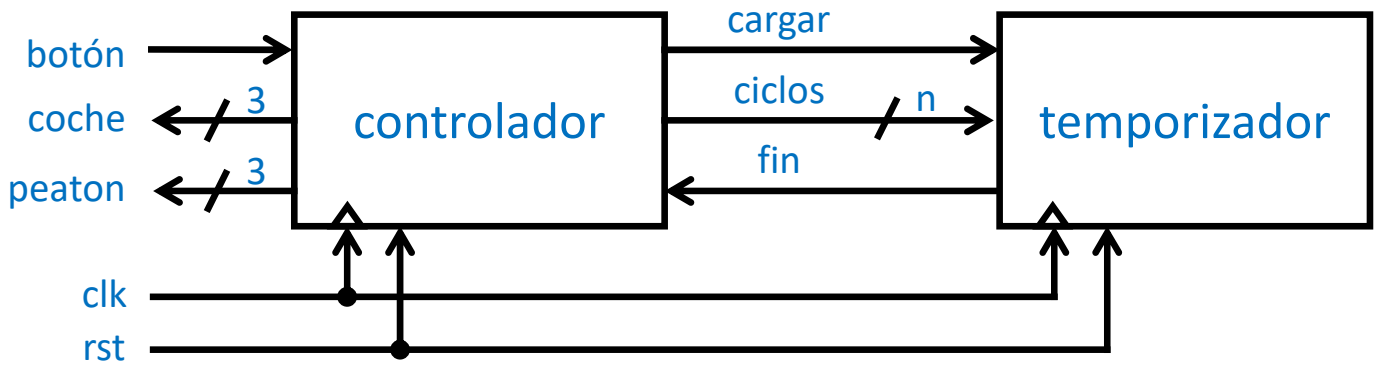
```
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity counter is
  generic(
    n   : integer := 4; max : integer := 10 );
  port(
    rst, clk, ce :  in std_logic;
    dout         : out std_logic_vector(n-1 downto 0));
end counter;

architecture syn1 of counter is
  signal cs : unsigned(n-1 downto 0);
begin
  process (rst, clk, ce, cs)
  begin
    dout <= std_logic_vector(cs);
    if rst='1' then
      cs <= (others=>'0');
    elsif rising_edge(clk) then
      if ce='1' then
        if cs=max then
          cs <= (others=>'0');
        else
          cs <= cs + 1;
        end if;
      end if;
    end if;
  end process;
end syn1;
```

```
architecture syn2 of counter is
  signal cs : unsigned(n-1 downto 0);
begin
  process (rst, clk, ce, cs)
  begin
    dout <= std_logic_vector(cs);
    if rst='1' then
      cs <= (others=>'0');
    elsif rising_edge(clk) then
      if ce='1' then
        cs <= (cs + 1) mod max;
      end if;
    end if;
  end process;
end syn2;
```

Ejemplos

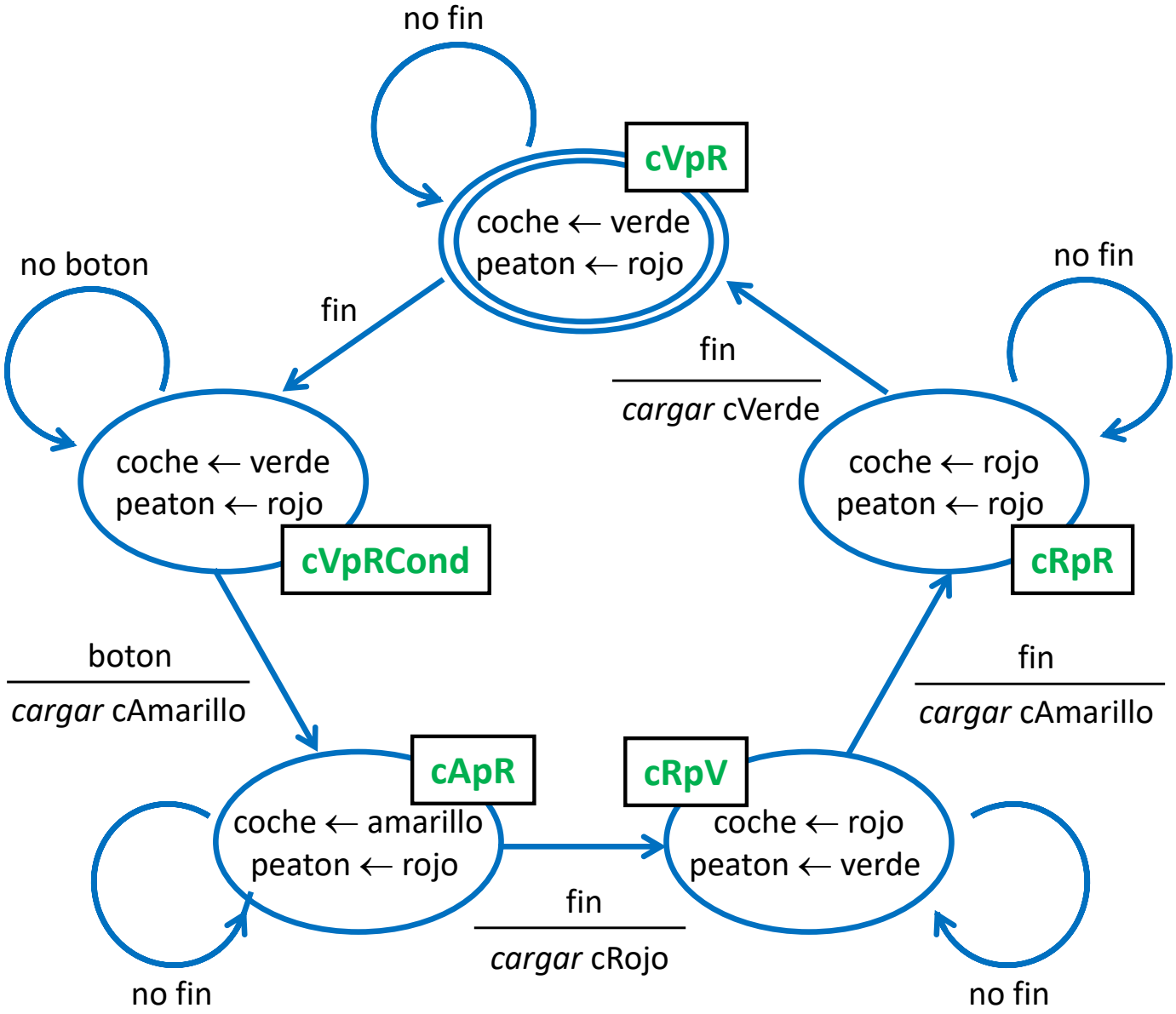
FSM temporizadas

- Se desea diseñar un sistema digital que controle un semáforo con botón peatonal para solicitar el paso:
  - El semáforo de coches estará verde como mínimo un periodo verde y continuará en verde hasta que no se pulse el botón.
  - Si se pulsa el botón, sólo si el semáforo de coches ha estado en verde durante un periodo verde completo, el semáforo de coches pasará a amarillo durante un periodo amarillo, tras el cual se pondrá en rojo. Entonces el semáforo de peatones pasará a verde.
  - El semáforo de coches permanecerá en rojo un periodo rojo, tras el cual el semáforo de peatones pasará a rojo. El semáforo de coches pasará a verde transcurrido un periodo amarillo desde que el de peatones cambió.



Ejemplos

FSM temporizadas: controlador



Ejemplos

FSM temporizadas: codificación con 7 sentencias

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

entity controlSemaforo is
  generic( cRojo, cAmarillo, cVerde : natural );
  port(
    rst, clk : in std_logic;
    boton : in std_logic;
    coche, peaton : out std_logic_vector(2 downto 0)
  );
end controlSemaforo;

architecture syn of controlSemaforo is

  constant sRojo     : std_logic_vector(2 downto 0) := "100";
  constant sAmarillo : std_logic_vector(2 downto 0) := "010";
  constant sVerde    : std_logic_vector(2 downto 0) := "001";

  signal cargar, fin : std_logic;
  signal csT, nsT, ciclos : natural;

  type estados_t is ( cVpR, cVpRCond, cApR, cRpV, cRpR );
  signal csC, nsC : estados_t;

begin
  ...
end syn;
```

J.M. Mendías
2024

Ejemplos

FSM temporizadas: temporizador VHDL

```
stateTemporizador:
process (rst, clk)
begin
  if rst = '1' then
    csT <= cVerde;
  elsif rising_edge(clk) then
    csT <= nsT;
  end if;
end process;
```

```
mooreGenTemporizador:
process (csT)
begin
  if csT=0 then
    fin <= '1';
  else
    fin <= '0';
  end if;
end process;
```

```
stateGenTemporizador:
process (csT, fin, cargar, ciclos)
begin
  if cargar='1' then
    nsT <= ciclos;
  elsif fin='1' then
    nsT <= csT;
  else
    nsT <= csT - 1;
  end if;
end process;
```

J.M. Mendías
2024

Ejemplos

FSM temporizadas: controlador VHDL (i)

```
stateControlador:
process (rst, clk)
begin
  if rst='1' then
    csC <= cVpR;
  elsif rising_edge(clk) then
    csC <= nsC;
  end if;
end process;
```

```
stateGenControlador:
process (csC, boton, fin)
begin
  nsC <= csC;
  case csC is
    when cVpR =>
      if fin='1' then
        nsC <= cVpRCond;
      end if;
    when cVpRCond =>
      if boton='1' then
        nsC <= cApR;
      end if;
    when cApR =>
      if fin='1' then
        nsC <= cRpV;
      end if;
    when cRpV =>
      if fin='1' then
        nsC <= cRpR;
      end if;
    when cRpR =>
      if fin = '1' then
        nsC <= cVpR;
      end if;
  end case;
end process;
```

Ejemplos

FSM temporizadas: controlador VHDL (ii)

```
mooreGenControlador:
process (csC)
begin
  case csC is
    when cVpR =>
      coche  <= sVerde;
      peaton <= sRojo;
    when cVpRCond =>
      coche  <= sVerde;
      peaton <= sRojo;
    when cApR =>
      coche  <= sAmarillo;
      peaton <= sRojo;
    when cRpV =>
      coche  <= sRojo;
      peaton <= sVerde;
    when cRpR =>
      coche  <= sRojo;
      peaton <= sRojo;
  end case;
end process;
```

```
mealyGenControlador:
process (csC, boton, fin)
begin
  cargar <= '0';
  ciclos <= cVerde;
  case csC is
    when cVpR =>
      null;
    when cVpRCond =>
      if boton='1' then
        cargar <= '1';
        ciclos <= cAmarillo;
      end if;
    when cApR =>
      if fin='1' then
        cargar <= '1';
        ciclos <= cRojo;
      end if;
    when cRpV =>
      if fin='1' then
        cargar <= '1';
        ciclos <= cAmarillo;
      end if;
    when cRpR =>
      if fin='1' then
        cargar <= '1';
        ciclos <= cVerde;
      end if;
  end case;
end process;
```

Ejemplos

FSM temporizadas: codificación con 2 sentencias

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

entity controlSemaforo is
 generic( cRojo, cAmarillo, cVerde : natural );
  port(
    rst, clk : in std_logic;
    boton : in std_logic;
    coche, peaton : out std_logic_vector(2 downto 0)
  );
end controlSemaforo;

architecture syn of controlSemaforo is

  constant sRojo     : std_logic_vector(2 downto 0) := "100";
  constant sAmarillo : std_logic_vector(2 downto 0) := "010";
  constant sVerde    : std_logic_vector(2 downto 0) := "001";

  signal cargar, fin : std_logic;
  signal numCiclos, ciclos : natural;

  type estados_t is ( pVsR, pVsRCond, pAsR, pRsV, pRsA );
  signal estado : estados_t;

begin
  ...
end syn;
```

Ejemplos

FSM temporizadas: temporizador VHDL unificado


```
temporizador:
process (rst, clk)
begin
  if numCiclos=0 then
    fin <= '1';
  else
    fin <= '0';
  end if;
  if rst='1' then
    numCiclos <= cVerde;
  elsif rising_edge(clk) then
    if cargar='1' then
      numCiclos <= ciclos;
    elsif fin='0' then
      numCiclos <= numCiclos - 1;
    end if;
  end if;
end process;
```

Ejemplos

FSM temporizadas: controlador VHDL unificado

```
controlador:
process (rst, clk, estado, boton, fin)
begin
  cargar <= '0'; ciclos <= cVerde;
  case estado is
    when cVpR =>
      coche <= sVerde; peaton <= sRojo;
    when cVpRCond =>
      coche <= sVerde; peaton <= sRojo;
      if boton='1' then
        cargar <= '1'; ciclos <= cAmarillo;
      end if;
    when cApR =>
      coche <= sAmarillo; peaton <= sRojo;
      if fin='1' then
        cargar <= '1'; ciclos <= cRojo';
      end if;
    when cRpV =>
      coche <= sRojo; peaton <= sVerde;
      if fin='1' then
        cargar <= '1'; ciclos <= cAmarillo;
      end if;
    when cRpR =>
      coche <= sRojo; peaton <= sRojo;
      if fin='1' then
        cargar <= '1'; ciclos <= cVerde;
      end if;
  end case;
  ...
```

```
  ...
  if rst = '1' then
    estado <= cVpR;
  elsif rising_edge(clk) then
    case estado is
      when cVpR =>
        if fin='1' then
          estado <= cVpRCond;
        end if;
      when cVpRCond =>
        if boton='1' then
          estado <= cApR;
        end if;
      when cApR =>
        if fin='1' then
          estado <= cRpV;
        end if;
      when cRpV =>
        if fin='1' then
          estado <= cRpR;
        end if;
      when cRpR =>
        if fin='1' then
          estado <= cVpR;
        end if;
    end case;
  end if;
end process;
```

Ejemplos

FSMT: codificación con 1 sentencia

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

entity controlSemaforo is
 generic( cRojo, cAmarillo, cVerde : natural );
  port(
    rst, clk : in std_logic;
    boton : in std_logic;
    coche, peaton : out std_logic_vector(2 downto 0)
  );
end controlSemaforo;

architecture syn of controlSemaforo is

  constant sRojo     : std_logic_vector(2 downto 0) := "100";
  constant sAmarillo : std_logic_vector(2 downto 0) := "010";
  constant sVerde    : std_logic_vector(2 downto 0) := "001";

  signal numCiclos : natural;

  type estados_t is ( pVsR, pVsRCond, pAsR, pRsV, pRsA );
  signal estado : estados_t;

begin
  ...
end syn;
```

Ejemplos

FSMT: temporizador + controlador unificados (i)

```
fsmt:
process (rst, clk, estado)
begin
  case estado is
    when cVpR =>
      coche  <= sVerde;
      peaton <= sRojo;
    when cVpRCond =>
      coche  <= sVerde;
      peaton <= sRojo;
    when cApR =>
      coche  <= sAmarillo;
      peaton <= sRojo;
    when cRpV =>
      coche  <= sRojo;
      peaton <= sVerde;
    when pRpR =>
      coche  <= sRojo;
      peaton <= sRojo;
  end case;
  ...
```

solo cambia de estado si el temporizador a finalizado la cuenta de ciclos

programa el temporizador (el número de ciclos puede ser distinto en cada estado)

```
  ...
  if rst = '1' then
    estado    <= cVpR;
    numCiclos <= cVerde;
  elsif rising_edge(clk) then
    if numCiclos /= 0 then
      numCiclos <= numCiclos - 1;
    else
      case estado is
        when cVpR =>
          estado <= cVpRCond;
        when cVpRCond =>
          if boton='1' then
            estado    <= cApR;
            numCiclos <= cAmarillo;
          end if;
        when cApR =>
          estado    <= cRpV;
          numCiclos <= cRojo;
        when cRpV =>
          estado    <= cRpR;
          numCiclos <= cAmarillo;
        when cRpR =>
          estado    <= cVpR;
          numCiclos <= cVerde;
      end case;
    end if;
  end if;
end process;
```

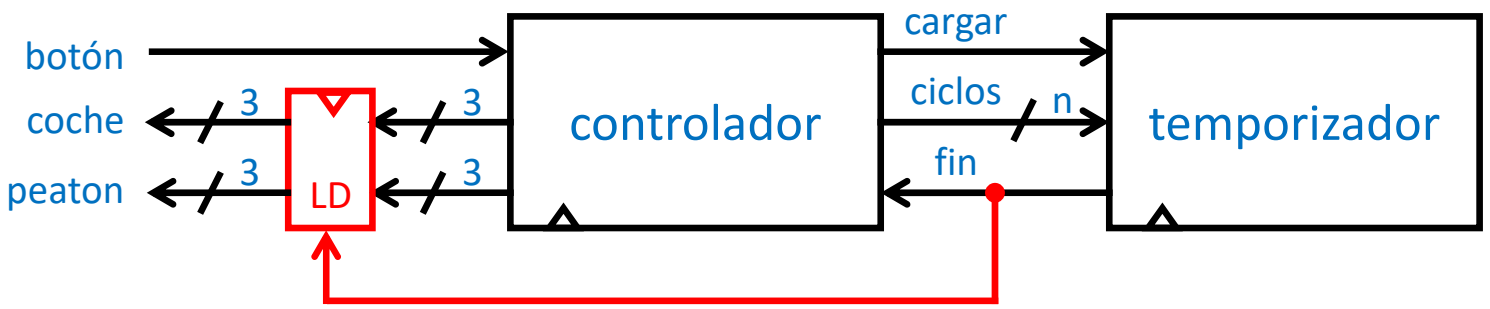
Ejemplos

FSMT: temporizador + controlador unificados (ii)

```
fsmt:
process (rst, clk, estado)
begin
if rst = '1' then
    estado    <= cVpR;
    numCiclos <= cVerde;
  elsif rising_edge(clk) then
    if numCiclos /= 0 then
      numCiclos <= numCiclos - 1;
    else
      case estado is
        when cVpR =>
          coche  <= sVerde;
          peaton <= sRojo;
          estado <= cVpRCond;
        when cVpRCond =>
          coche  <= sVerde;
          peaton <= sRojo;
          if boton='1' then
            estado    <= cApR;
            numCiclos <= cAmarillo;
          end if;
...
```

```
...
        when cApR =>
          coche     <= sAmarillo;
          peaton    <= sRojo;
          estado    <= cRpV;
          numCiclos <= cRojo;
        when cRpV =>
          coche     <= sRojo;
          peaton    <= sVerde;
          estado    <= cRpR;
          numCiclos <= cAmarillo;
        when cRpR =>
          coche     <= sRojo;
          peaton    <= sRojo;
          estado    <= cVpR;
          numCiclos <= cVerde;
      end case;
    end if;
  end if;
end process;
```

INCORRECTO ??



Ejemplos

FSMT: codificación con 1 sentencia y variables


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

entity controlSemaforo is
 generic( cRojo, cAmarillo, cVerde : natural );
  port(
    rst, clk : in std_logic;
    boton : in std_logic;
    coche, peaton : out std_logic_vector(2 downto 0)
  );
end controlSemaforo;

architecture syn of controlSemaforo is

  constant sRojo     : std_logic_vector(2 downto 0) := "100";
  constant sAmarillo : std_logic_vector(2 downto 0) := "010";
  constant sVerde    : std_logic_vector(2 downto 0) := "001";

begin
  ...
end syn;
```

Ejemplos

FSMT: temporizador + controlador unificados y variables

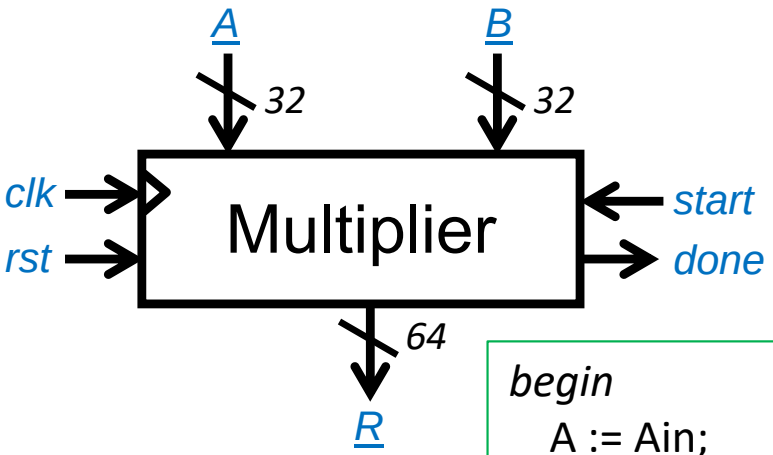
```
fsmt:
process (rst, clk)
  type estados_t is
   ( cVpR, cVpRCond, cApR,cRpV, cRpR );
  variable estado : estados_t;
  variable numCiclos : natural;
begin
  case estado is
    when cVpR =>
      coche  <= sVerde;
      peaton <= sRojo;
    when cVpRCond =>
      coche  <= sVerde;
      peaton <= sRojo;
    when cApR =>
      coche  <= sAmarillo;
      peaton <= sRojo;
    when cRpV =>
      coche  <= sRojo;
      peaton <= sVerde;
    when cRpR =>
      coche  <= sRojo;
      peaton <= sRojo;
  end case;
  ...
```

```
  ...
  if rst = '1' then
    estado    := cVpR;
    numCiclos := cVerde;
  elsif rising_edge(clk) then
    if numCiclos /= 0 then
      numCiclos := numCiclos - 1;
    else
      case estado is
        when cVpR =>
          estado := cVpRCond;
        when cVpRCond =>
          if boton='1' then
            estado    := cApR;
            numCiclos := cAmarillo;
          end if;
        when cApR =>
          estado    := cRpV;
          numCiclos := cRojo;
        when cRpV =>
          estado    := cRpR;
          numCiclos := cAmarillo;
        when cRpR =>
          estado    := cVpR;
          numCiclos := cVerde;
      end case;
    end if;
  end if;
end process;
```

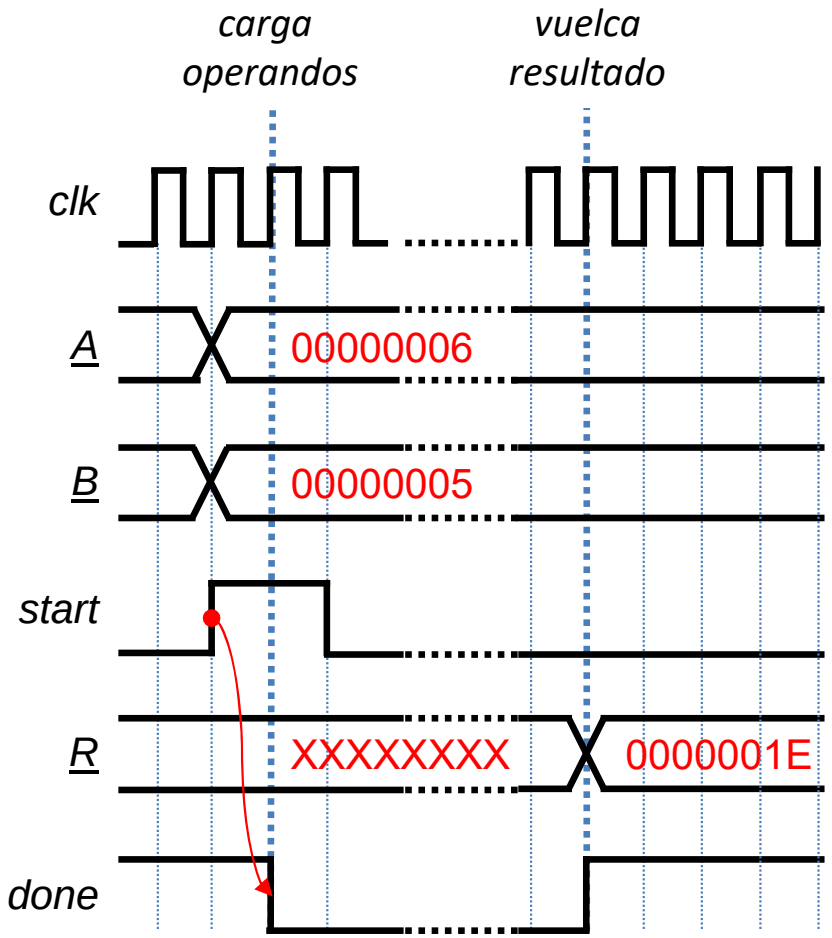
Ejemplos

FSMD: FSM + ruta de datos

- Se desea diseñar un multiplicador sin signo por el algoritmo de suma y desplazamiento.

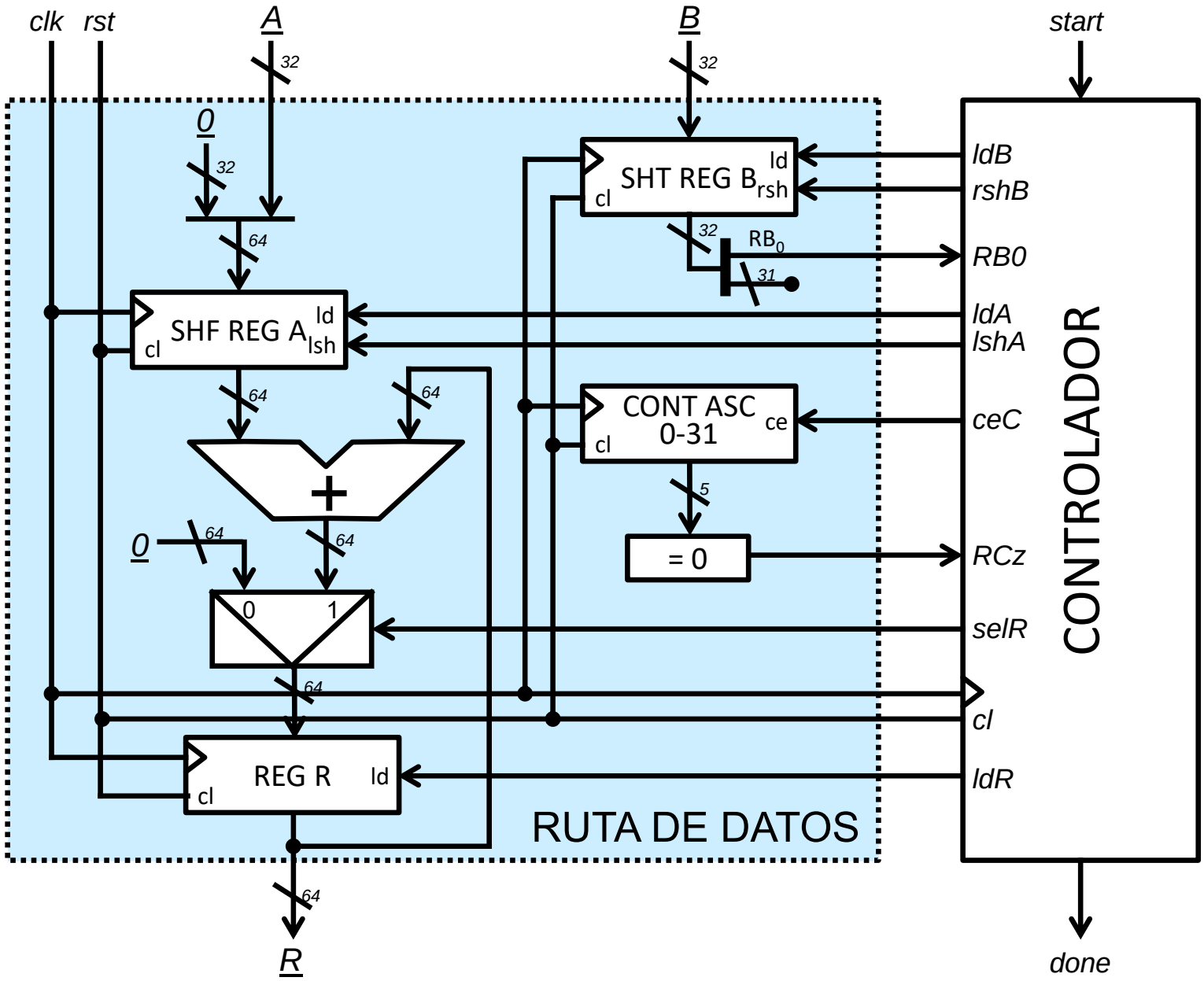



```
begin
   A := Ain;
   B := Bin;
   R := 0;
   for C:=0 to 31 do begin
      if B0=1 then R := R+A;
        A := A << 1;
   B := B >> 1;
   end for;
   Rout := R;
end;
```



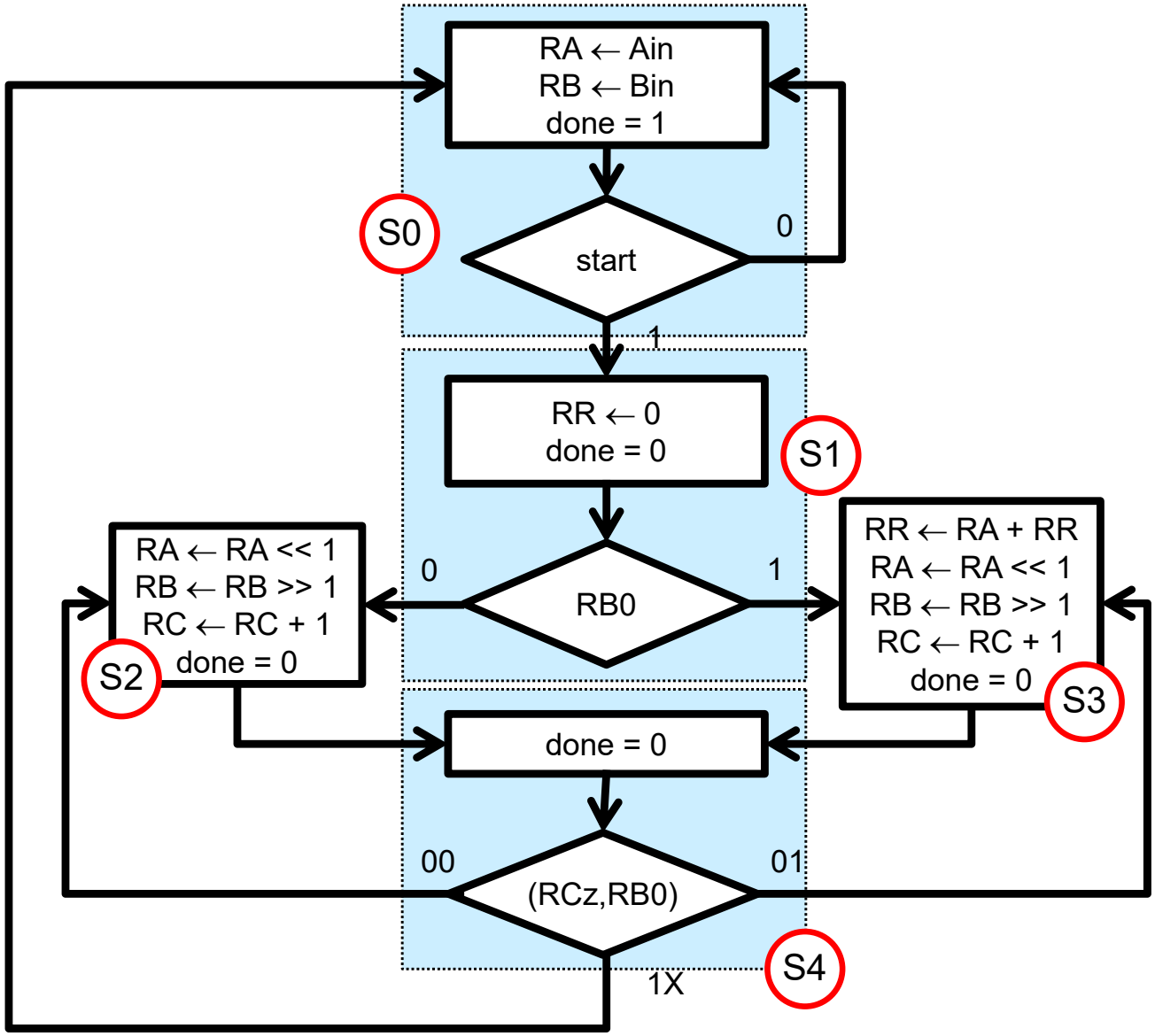
Ejemplos

FSMD: estructura



Ejemplos

FSMD: controlador




```
begin
   A := Ain;
   B := Bin;
   R := 0;
   for C:=0 to 31 do begin
      if B0=1 then R := R+A;
        A := A << 1;
   B := B >> 1;
   end for;
   Rout := R;
end;
```

Ejemplos

FSMD: codificación con datapath explícito (i)

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

entity multiplier is
  port
  (
    rst   :  in std_logic;
    clk   :  in std_logic;
    start :  in std_logic;
    done  : out std_logic;
    a     :  in std_logic_vector(31 downto 0);
    b     :  in std_logic_vector(31 downto 0);
    r     : out std_logic_vector(63 downto 0)
  );
end multiplier;

architecture syn of multiplier is

  signal ra, ra_next : unsigned(63 downto 0);
  signal rb, rb_next : unsigned(31 downto 0);
  signal rr, rb_next : unsigned(63 downto 0);
  signal rc, rc_next : unsigned(4 downto 0);

  type states_t is ( s0, s1, s2, s3, s4 );
  signal state : states_t;

begin
  ...
end syn;
```

estado actual y siguiente de los registros de la ruta de datos

estado actual del controlador

Ejemplos

FSMD: codificación con datapath explícito (ii)

Controlador

```
process (rst, clk)
begin
  if rst='1' then
    state <= s0;
  elsif rising_edge(clk) then
    case state is
      when s0 =>
        if start='1' then
          state <= s1;
        end if;
      when s1 =>
        if rb(0)='1' then
          state <= s3;
        else
          state <= s2;
        end if;
      when s2 | s3 =>
        state <= s4;
...
```

```
...

      when s4 =>
        if rc=0 then
          state <= s0;
        else
          if rb(0)='1' then
            state <= s3;
          else
            state <= s2;
          end if;
        end if;
    end case;
  end if;
end process;
```

Ejemplos

FSMD: codificación con datapath explícito (iii)

Ruta de datos (registros)

```
process (rst, clk)
begin
  if rst='1' then
    ra <= (others => '0');
    rb <= (others => '0');
    rr <= (others => '0');
    rc <= (others => '0');
  elsif rising_edge(clk) then
    ra <= ra_next;
    rb <= rb_next;
    rr <= rr_next;
    rc <= rc_next;
  end if;
end process;
```

Ruta de datos (elementos combinacionales)

```
done <= '1' when state=s0 else '0';

r <= std_logic_vector(rr);

with state select
  ra_next <=
    resize( unsigned(a), 64) when s0,
    ra(62 downto 0) & '0' when s2 | s3,
    ra when others;

with state select
  rb_next <=
    unsigned(b) when s0,
    '0' & rb(31 downto 1) when s2 | s3,
    rb when others;

with state select
  rr_next <=
    (others => '0') when s1,
    ra + rr when s3,
    rr when others;

with state select
  rc_next <=
    rc + 1 when s2 | s3,
    rc when others;
```

el reuso de recursos es explícito
y no depende de la herramienta EDA

Ejemplos

FSMD: codificación con datapath implícito (i)

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

entity multiplier is
  port
  (
    rst   :  in std_logic;
    clk   :  in std_logic;
    start :  in std_logic;
    done  : out std_logic;
    a     :  in std_logic_vector(31 downto 0);
    b     :  in std_logic_vector(31 downto 0);
    r     : out std_logic_vector(63 downto 0)
  );
end multiplier;

architecture syn of multiplier is

  signal ra : unsigned(63 downto 0);
  signal rb : unsigned(31 downto 0);
  signal rr : unsigned(63 downto 0);
  signal rc : unsigned(4 downto 0);

  type states_t is ( s0, s1, s2, s3, s4 );
  signal state : states_t;

begin
  ...
end syn;
```

estado actual y de los registros de la ruta de datos

estado actual del controlador

J.M. Mendías 2024

Ejemplos

FSMD: codificación con datapath implícito (ii)

```
done <= '1' when state=s0 else '0';

r <= std_logic_vector(rr);

process (rst, clk, cs)
begin
  if rst='1' then
    ra <= (others => '0');
    rb <= (others => '0');
    rr <= (others => '0');
    rc <= (others => '0');
    state <= s0;
  elsif rising_edge(clk) then
    case state is
      when s0 =>
        ra <= resize( unsigned(a), 64);
        rb <= unsigned(b);
        if start='1' then
          state <= s1;
        end if;
      when s1 =>
        rr <= (others => '0');
        if rb(0)='1' then
          state <= s3;
        else
          state <= s2;
        end if;
...
```

```
...
      when s2 =>
        ra <= ra(62 downto 0) & '0';
        rb <= '0' & rb(31 downto 1);
        rc <= rc + 1;
        state <= s4;
      when s3 =>
        rr <= ra + rr;
        ra <= ra(62 downto 0) & '0';
        rb <= '0' & rb(31 downto 1);
        rc <= rc + 1;
        state <= s4;
      when s4 =>
        if rc=0 then
          state <= s0;
        else
          if rb(0)='1' then
            state <= s3;
          else
            state <= s2;
          end if;
        end if;
    end case;
  end if;
end process;
```

el reuso de recursos es implícito y depende de la herramienta EDA

Ejemplos

FSMD: codificación VHDL con datapath implícito y variables

```
process (rst, clk)

  variable ra : unsigned(63 downto 0);
  variable rb : unsigned(31 downto 0);
  variable rr : unsigned(63 downto 0);
  variable rc : unsigned(4 downto 0);

  type states_t is ( s0, s1, s2, s3, s4 );
  variable state : states_t;

begin
  r <= std_logic_vector(rr);
  if state=s0 then
    done <= '1';
  else
    done <= '1';
  end if;
  if rst='1' then
    ra := (others => '0');
    rb := (others => '0');
    rr := (others => '0');
    rc := (others => '0');
    state := s0;
  elsif rising_edge(clk) then
    case state is
      when s0 =>
        ra := resize( unsigned(a), 64);
        rb := unsigned(b);
        if start='1' then
          state := s1;
        end if;
...
```

```
...
      when s1 =>
        rr := (others => '0');
        if rb(0)='1' then
          state := s3;
        else
          state := s2;
        end if;
      when s2 =>
        ra := ra(62 downto 0) & '0';
        rb := '0' & rb(31 downto 1);
        rc := rc + 1;
        state := s4;
      when s3 =>
        rr := ra + rr;
        ra := ra(62 downto 0) & '0';
        rb := '0' & rb(31 downto 1);
        rc := rc + 1;
        state := s4;
      when s4 =>
        if rc=0 then
          state := s0;
        else
          if rb(0)='1' then
            state := s3;
          else
            state := s2;
          end if;
        end if;
    end case;
  end if;
end process;
```

Portabilidad

- En teoría el código VHDL es portable.
  - Un mismo código VHDL sintetizado por diferentes herramientas EDA sobre diferentes tecnologías dará lugar a diseños con comportamiento equivalente.
- En la práctica, para obtener diseños eficientes, es necesario codificar teniendo en mente la tecnología objetivo y la herramienta usada:
  - Un mismo circuito puede requerir diferente especificación VHDL según se diseñe para proyectarse sobre una FPGA o sobre ASIC de celdas estándar.
  - Incluso en caso de proyectarse sobre una misma tecnología, por ejemplo FPGA, el código el VHDL para ser eficaz debe ser diferente de un fabricante y otro, e incluso de una familia a otra de un mismo fabricante.
- Para una síntesis eficiente el diseñador debe:
  - Conocer los recursos HW disponibles en la tecnología objetivo.
  - Conocer como los infiere la herramienta de síntesis desde el código VHDL.
  - Adaptar su estilo de codificación según lo anterior.

AMD 7 Series FPGAs

Recursos funcionales (i)

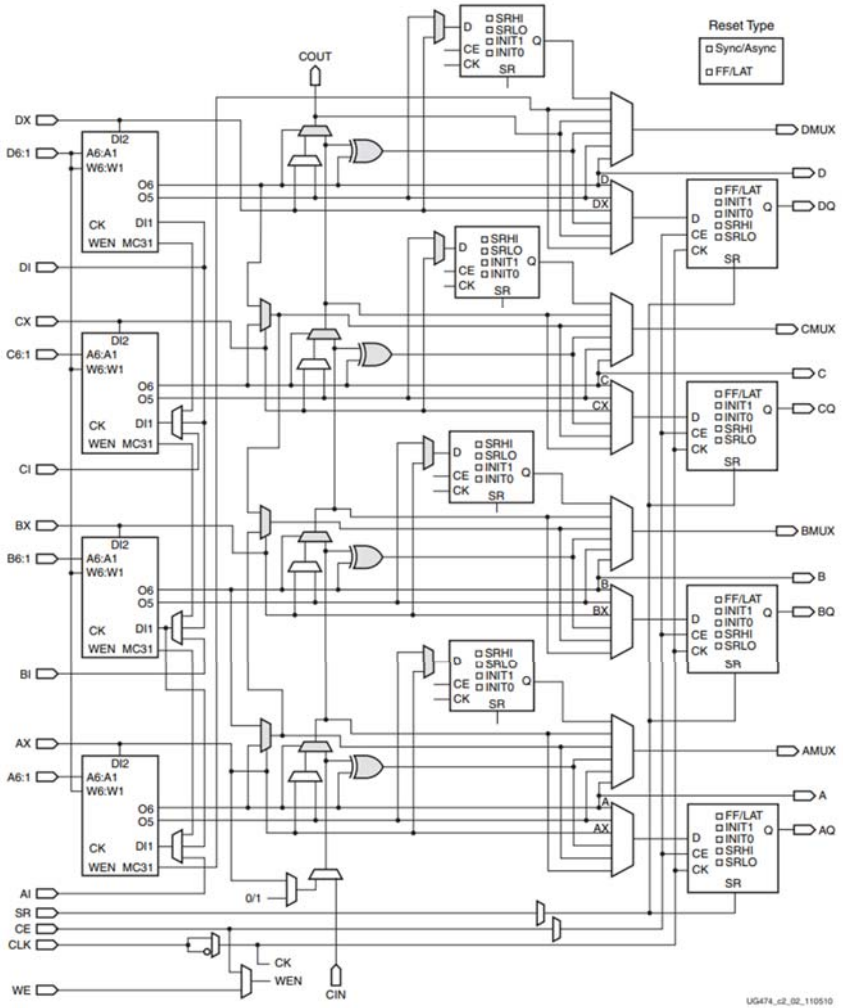
- Los principales recursos funcionales predifundidos que dispone la serie 7 sobre los que proyectar lógica son:
  - LUT: bloque configurable de lógica combinacional.
  - LUTRAM: LUT cuya memoria de configuración es usada como RAM síncrona.
  - LUTROM: LUT cuya memoria de configuración es usada como ROM.
  - SLR: LUT cuya memoria de configuración es usada como un registro de desplazamiento.
  - FFD/LD: biestable D de tipo flip-flop/latch.
  - BRAM: bloque de memoria RAM síncrona de doble puerto.
  - BROM: BRAM usada como ROM.
  - DSP48E1: bloque aritmético segmentado configurable (incluye registros, ALU y MULT).

- Cada 4 LUT + 8 FFD/LD forman un SLICE, y los hay de 2 tipos:
  - SLICEM: si las LUT pueden usarse como LUTRAM/SRL.
  - SLICEL: si las LUT no pueden usarse como LUTRAM/SRL.

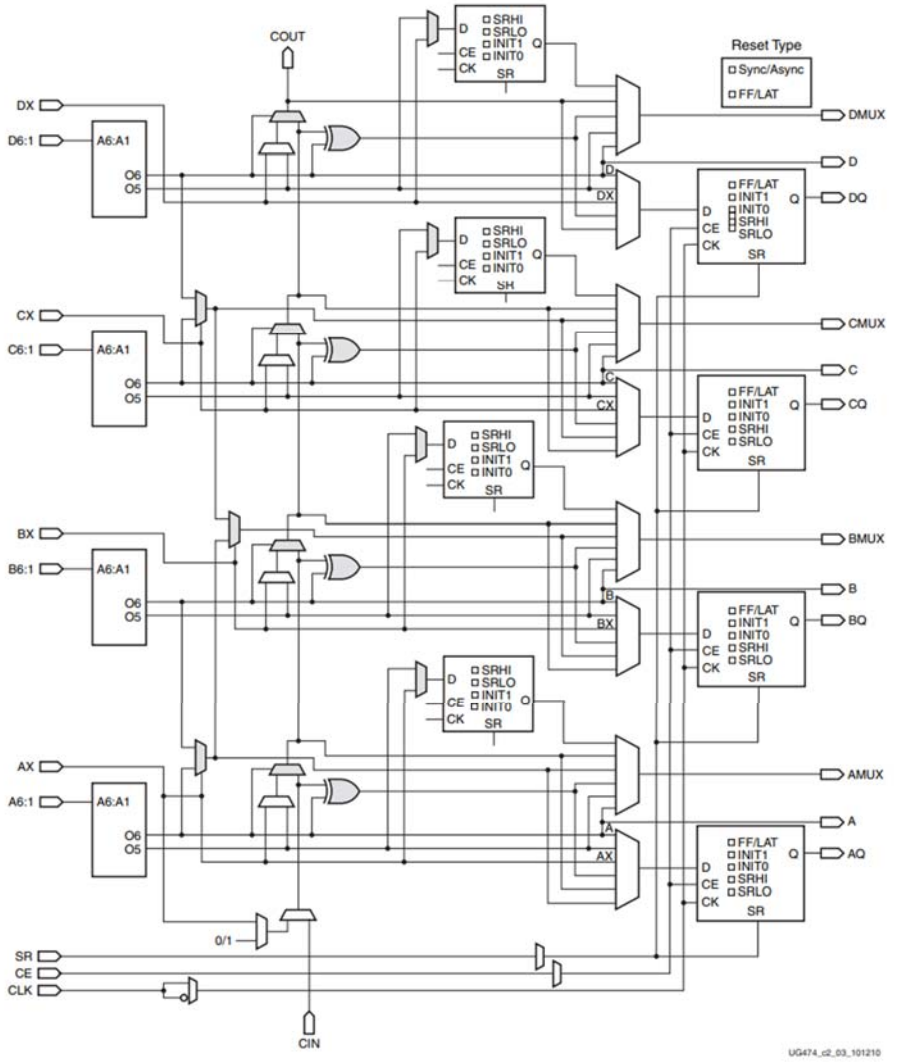
J.M. Mendías
2024

AMD 7 Series FPGAs

Recursos funcionales (ii)



SLICEM (lógica + RAM)

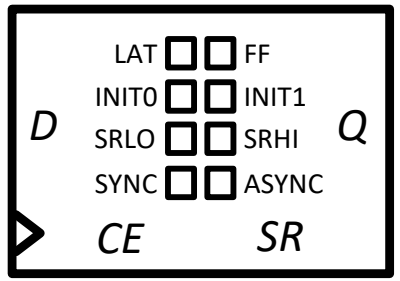


SLICEL (solo lógica)

AMD 7 Series FPGAs

FFD/LD (i)

- Todos los biestables de la FPGA son de tipo D y tienen 5 puertos:
  - CLK: entrada de reloj
  - CE: entrada de capacitación del reloj (activa en alta)
    - Cuando la lógica del diseño la desactiva, inhibe el reloj y el biestable no cambia de estado.
  - SR: entrada de reset local (activo en alta)
    - Cuando la lógica del diseño la activa, el biestable se inicializa.
  - D: entrada de datos
  - Q: salida de datos



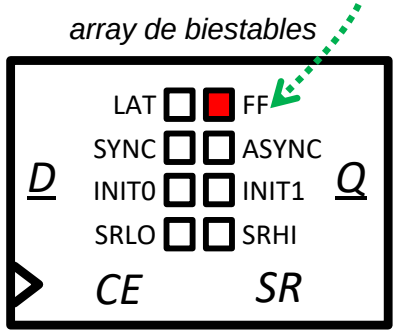
- Adicionalmente en cada biestable puede configurarse:
  - El tipo de comportamiento síncrono que tendrá: como latch o como flip-flop.
  - El valor inicial que tendrá cuando finalice la configuración y el diseño arranque.
  - El valor al que se inicializará cada vez que se active SR.
  - Si la inicialización por activación de SR será síncrona o asíncrona.
  - Esta configuración es fija para cada diseño.

- Es importante conocer cómo Vivado, a partir del código VHDL, infiere la configuración de los biestables y la lógica que estimula cada puerto.

AMD 7 Series FPGAs

diseños con reset asíncrono

```
process (rst, clk)
  variable cs : unsigned(n-1 downto 0);
begin
  if rst='1' then
    cs := (others => '0');
  elsif rising_edge(clk) then
    if clear='1' then
      cs := (others => '0');
    elsif ce='1' then
      cs := (cs + 1) mod MAXVAL then
    end if;
  end if;
end process;
```

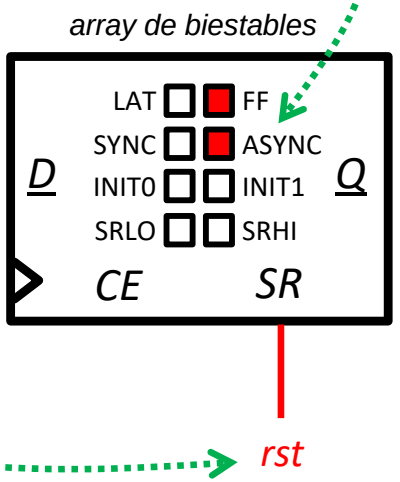


J.M. Mendías
2024

AMD 7 Series FPGAs

diseños con reset asíncrono

```
process (rst, clk)
  variable cs : unsigned(n-1 downto 0);
begin
  if rst='1' then
    cs := (others => '0');
  elsif rising_edge(clk) then
    if clear='1' then
      cs := (others => '0');
    elsif ce='1' then
      cs := (cs + 1) mod MAXVAL then
    end if;
  end if;
end process;
```



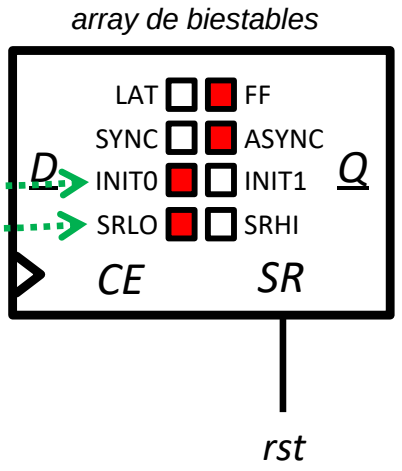
AMD 7 Series FPGAs

diseños con reset asíncrono

```
process (rst, clk)
  variable cs : unsigned(n-1 downto 0);
begin
  if rst='1' then
    cs := (others => '0');
  elsif rising_edge(clk) then
    if clear='1' then
      cs := (others => '0');
    elsif ce='1' then
      cs := (cs + 1) mod MAXVAL then
    end if;
  end if;
end process;
```

Al no indicarse el valor inicial de **cs**, *el valor del biestable tras start-up se infiere de la expresión de reset*

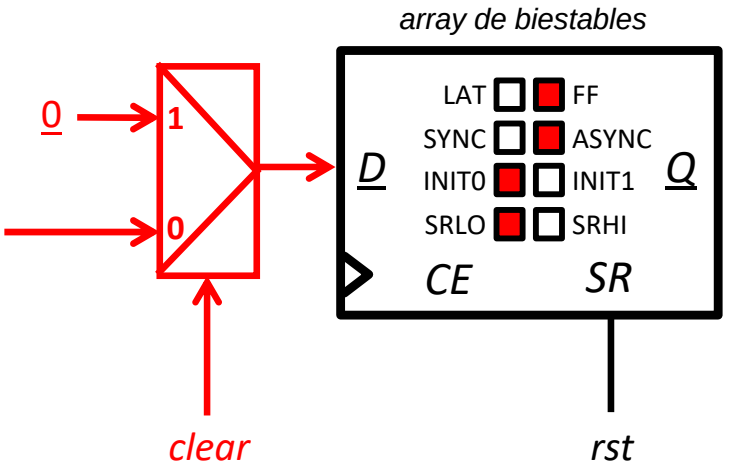
Tanto si se activa el reset global (GSR) como si se activa el reset local (rst) el biestable irá a 0 asíncronamente



AMD 7 Series FPGAs

diseños con reset asíncrono

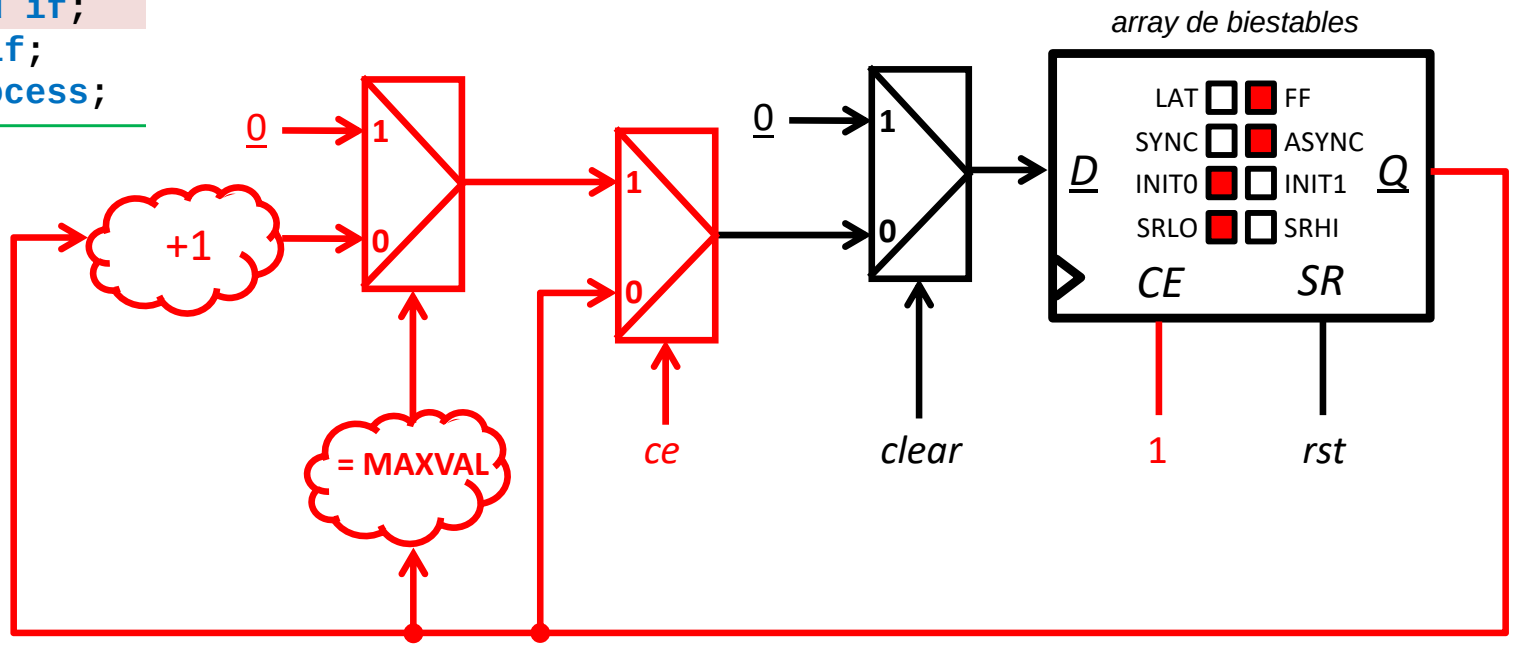
```
process (rst, clk)
  variable cs : unsigned(n-1 downto 0);
begin
  if rst='1' then
    cs := (others => '0');
  elsif rising_edge(clk) then
    if clear='1' then
      cs := (others => '0');
    elsif ce='1' then
      cs := (cs + 1) mod MAXVAL then
    end if;
  end if;
end process;
```



AMD 7 Series FPGAs

diseños con reset asíncrono

```
process (rst, clk)
  variable cs : unsigned(n-1 downto 0);
begin
  if rst='1' then
    cs := (others => '0');
  elsif rising_edge(clk) then
    if clear='1' then
      cs := (others => '0');
    elsif ce='1' then
      cs := (cs + 1) mod MAXVAL then
    end if;
  end if;
end process;
```

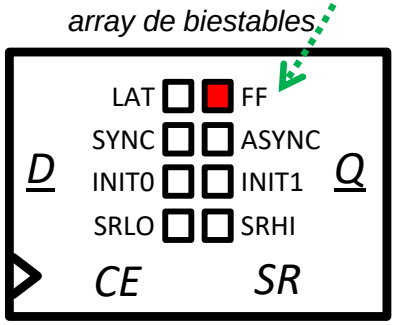


J.M. Mendías
2024

AMD 7 Series FPGAs

diseños con reset síncrono (i)

```
process (clk)
  variable cs : unsigned(n-1 downto 0);
begin
  if rising_edge(clk) then
    if rst='1' then
      cs := (others => '0');
    elsif clear='1' then
      cs := (others => '0');
    elsif ce='1' then
      cs := (cs + 1) mod MAXVAL then
    end if;
  end if;
end process;
```

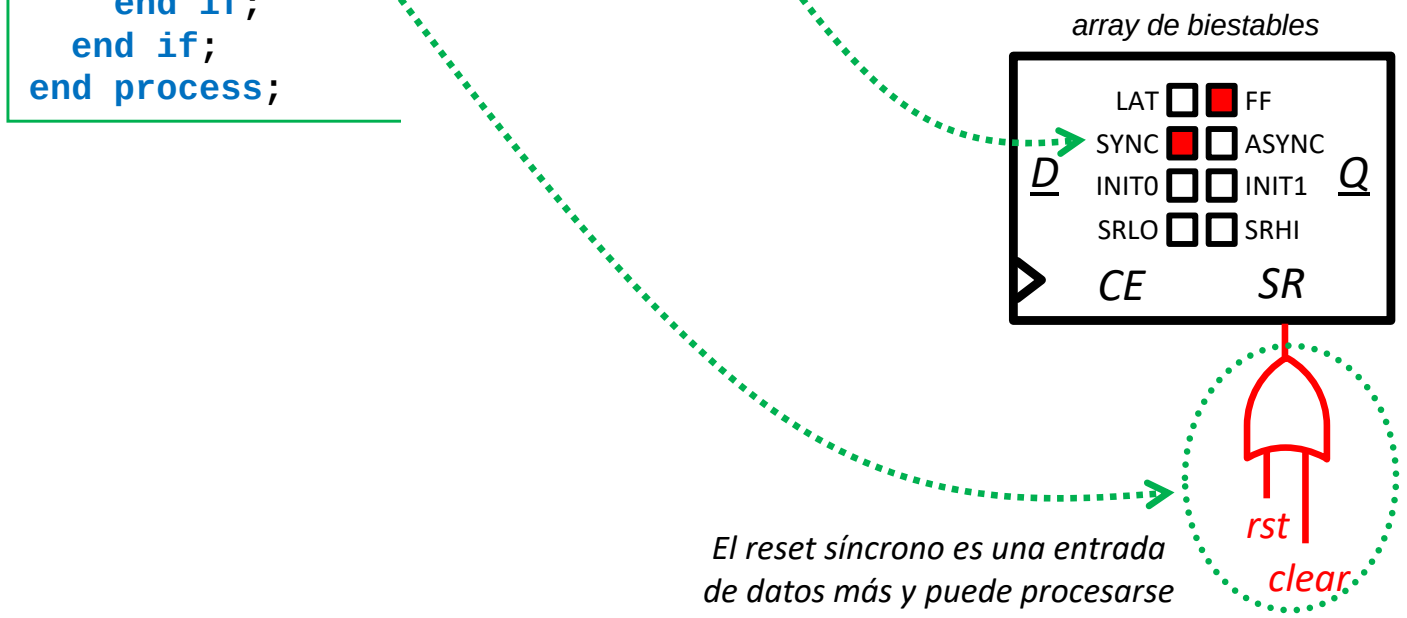


J.M. Mendías
2024

AMD 7 Series FPGAs

diseños con reset síncrono (i)

```
process (clk)
  variable cs : unsigned(n-1 downto 0);
begin
  if rising_edge(clk) then
    if rst='1' then
      cs := (others => '0');
    elsif clear='1' then
      cs := (others => '0');
    elsif ce='1' then
      cs := (cs + 1) mod MAXVAL then
    end if;
  end if;
end process;
```

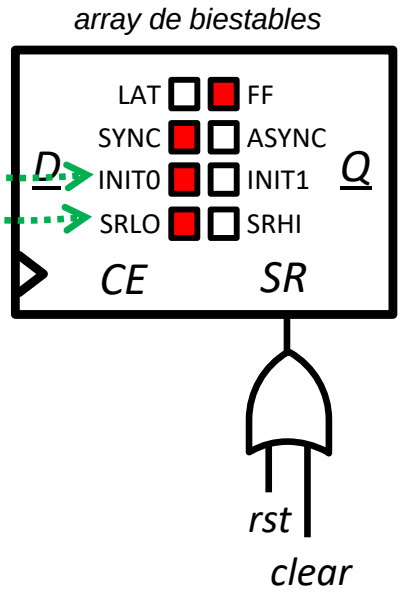


AMD 7 Series FPGAs

diseños con reset síncrono (i)

```
process (clk)
  variable cs : unsigned(n-1 downto 0);
begin
  if rising_edge(clk) then
    if rst='1' then
      cs := (others => '0');
    elsif clear='1' then
      cs := (others => '0');
    elsif ce='1' then
      cs := (cs + 1) mod MAXVAL then
    end if;
  end if;
end process;
```

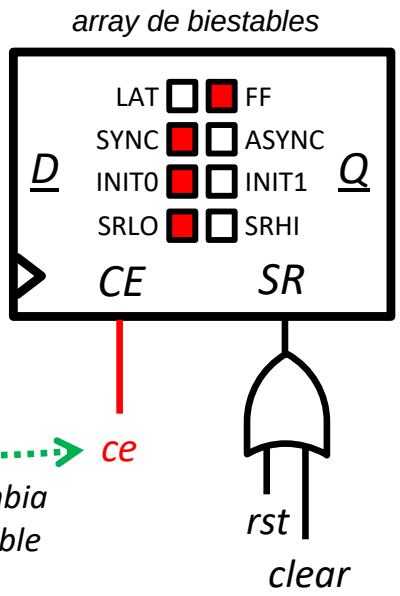
Al no indicarse el valor inicial de **cs**, *el valor del biestable tras start-up se infiere de la expresión de reset*



AMD 7 Series FPGAs

diseños con reset síncrono (i)

```
process (clk)
  variable cs : unsigned(n-1 downto 0);
begin
  if rising_edge(clk) then
    if rst='1' then
      cs := (others => '0');
    elsif clear='1' then
      cs := (others => '0');
    elsif ce='1' then
      cs := (cs + 1) mod MAXVAL then
    end if;
  end if;
end process;
```



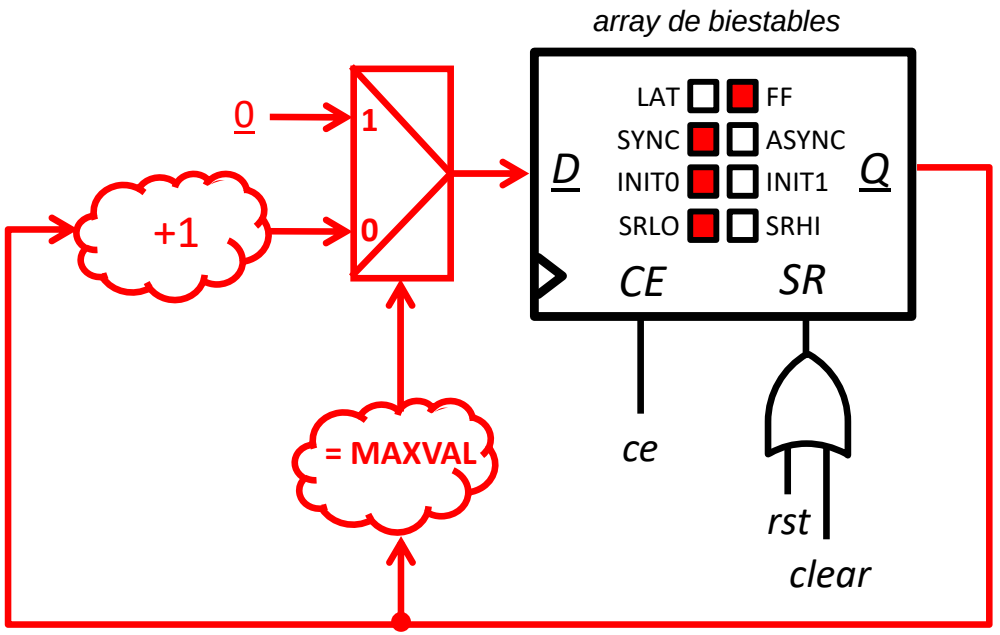
Si **ce** *no se activa, el estado no cambia*
luego se infiere el uso de clock-enable

J.M. Mendías
2024

AMD 7 Series FPGAs

diseños con reset síncrono (i)

```
process (clk)
  variable cs : unsigned(n-1 downto 0);
begin
  if rising_edge(clk) then
    if rst='1' then
      cs := (others => '0');
    elsif clear='1' then
      cs := (others => '0');
    elsif ce='1' then
      cs := (cs + 1) mod MAXVAL then
    end if;
  end if;
end process;
```

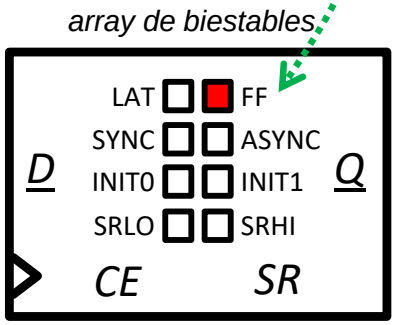


J.M. Mendías
2024

AMD 7 Series FPGAs

diseños con reset síncrono (ii)

```
process (clk)
  variable cs : unsigned(n-1 downto 0) := (others => '1');
begin
  if rising_edge(clk) then
    if clear='1' then
      cs := (others => '0');
    elsif ce='1' then
      cs := (cs + 1) mod MAXVAL then
    end if;
  end if;
end process;
```

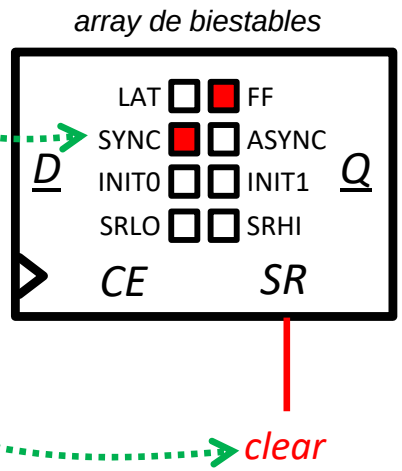


J.M. Mendías
2024

AMD 7 Series FPGAs

diseños con reset síncrono (ii)

```
process (clk)
  variable cs : unsigned(n-1 downto 0) := (others => '1');
begin
  if rising_edge(clk) then
    if clear='1' then
      cs := (others => '0');
    elsif ce='1' then
      cs := (cs + 1) mod MAXVAL then
    end if;
  end if;
end process;
```

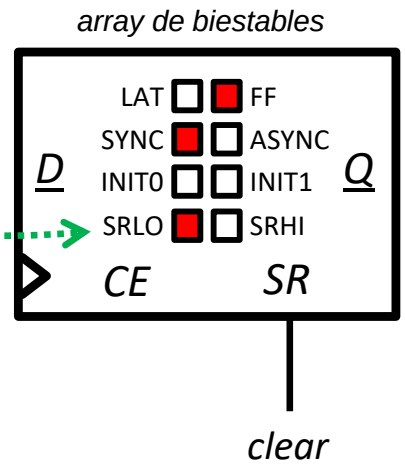


J.M. Mendías
2024

AMD 7 Series FPGAs

diseños con reset síncrono (ii)

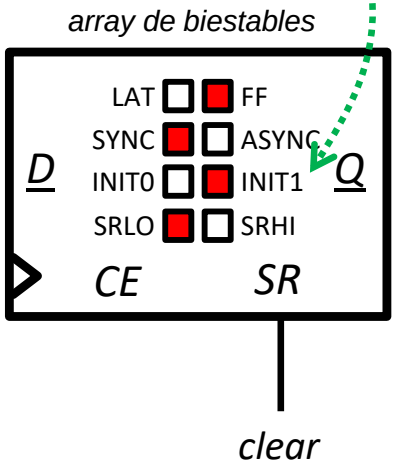
```
process (clk)
  variable cs : unsigned(n-1 downto 0) := (others => '1');
begin
  if rising_edge(clk) then
    if clear='1' then
      cs := (others => '0');
    elsif ce='1' then
      cs := (cs + 1) mod MAXVAL then
    end if;
  end if;
end process;
```



AMD 7 Series FPGAs

diseños con reset síncrono (ii)

```
process (clk)
  variable cs : unsigned(n-1 downto 0) := (others => '1');
begin
  if rising_edge(clk) then
    if clear='1' then
      cs := (others => '0');
    elsif ce='1' then
      cs := (cs + 1) mod MAXVAL then
    end if;
  end if;
end process;
```

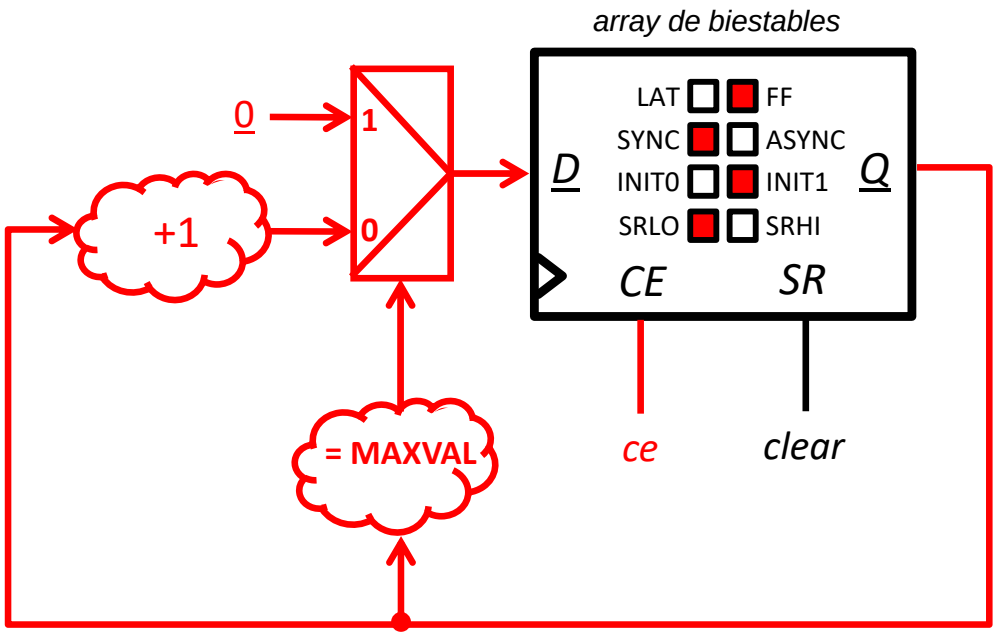


J.M. Mendías
2024

AMD 7 Series FPGAs

diseños con reset síncrono (ii)

```
process (clk)
  variable cs : unsigned(n-1 downto 0) := (others => '1');
begin
  if rising_edge(clk) then
    if clear='1' then
      cs := (others => '0');
    elsif ce='1' then
      cs := (cs + 1) mod MAXVAL then
    end if;
  end if;
end process;
```



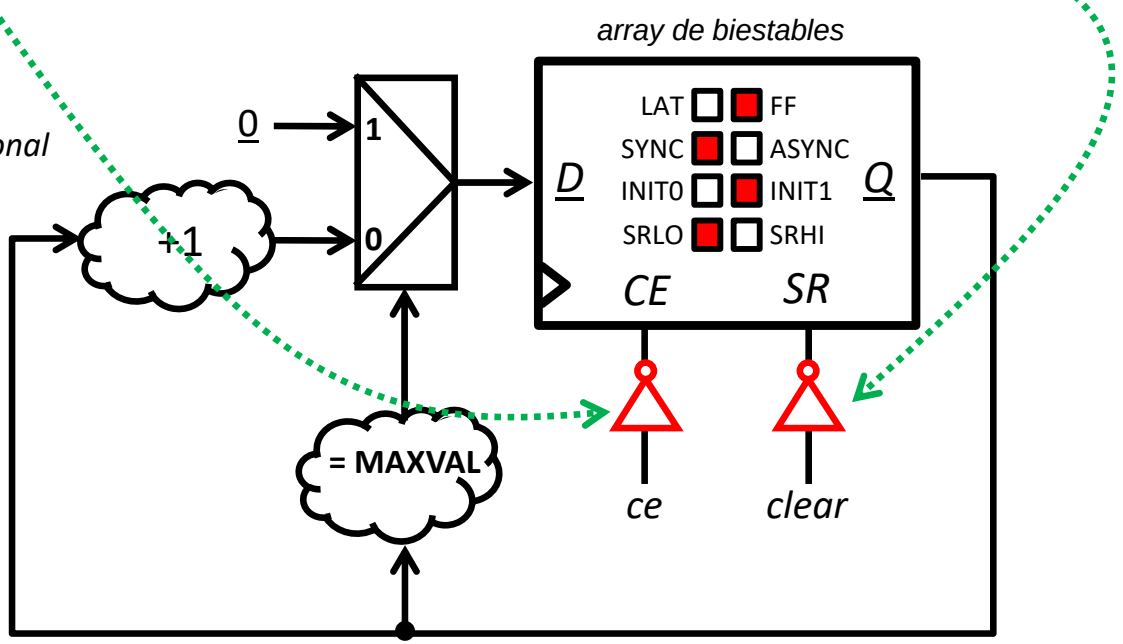
J.M. Mendías
2024

AMD 7 Series FPGAs

diseños con reset síncrono (iii)

```
process (clk)
  variable cs : unsigned(n-1 downto 0) := (others => '1');
begin
  if rising_edge(clk) then
    if clear='0' then
      cs := (others => '0');
    elsif ce='0' then
      cs := (cs + 1) mod MAXVAL then
    end if;
  end if;
end process;
```

El uso de lógica inversa, obliga a insertar lógica adicional

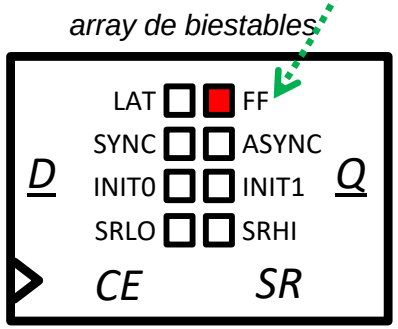


J.M. Mendías
2024

AMD 7 Series FPGAs

diseños sin reset (i)

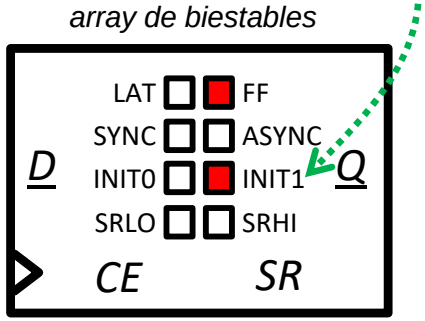
```
process (clk)
  variable cs : unsigned(n-1 downto 0) := (others => '1');
begin
  if rising_edge(clk) then
    if ce='1' then
      cs := (cs + 1) mod MAXVAL then
    elsif clear='1' then
      cs := (others => '0');
    end if;
  end if;
end process;
```



AMD 7 Series FPGAs

diseños sin reset (i)

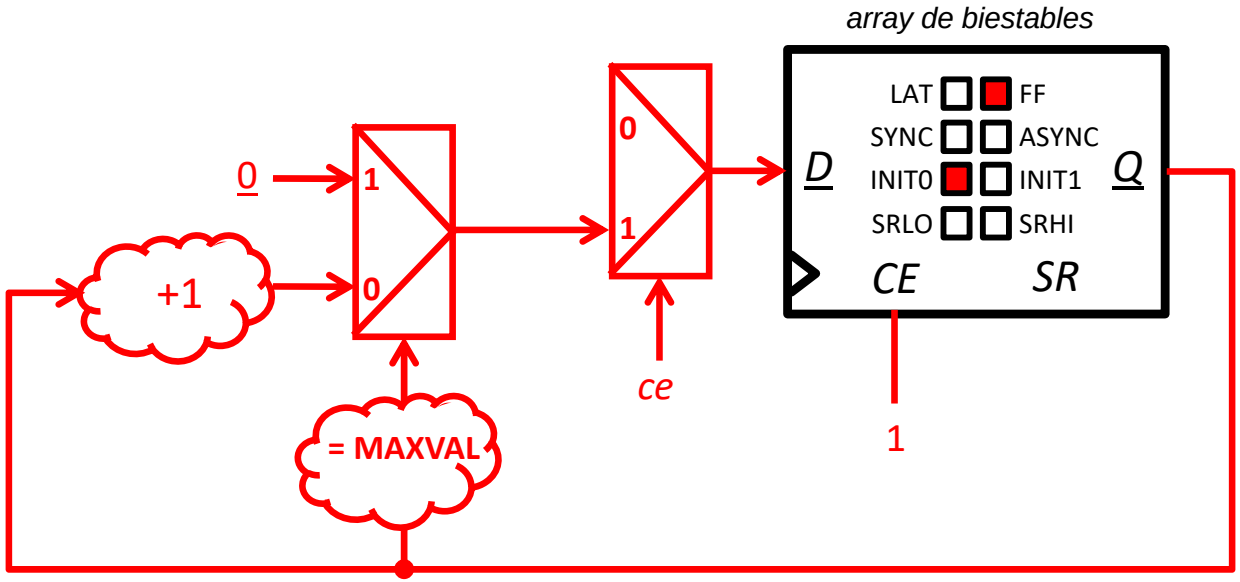
```
process (clk)
  variable cs : unsigned(n-1 downto 0) := (others => '1');
begin
  if rising_edge(clk) then
    if ce='1' then
      cs := (cs + 1) mod MAXVAL then
    elsif clear='1' then
      cs := (others => '0');
    end if;
  end if;
end process;
```



AMD 7 Series FPGAs

diseños sin reset (i)

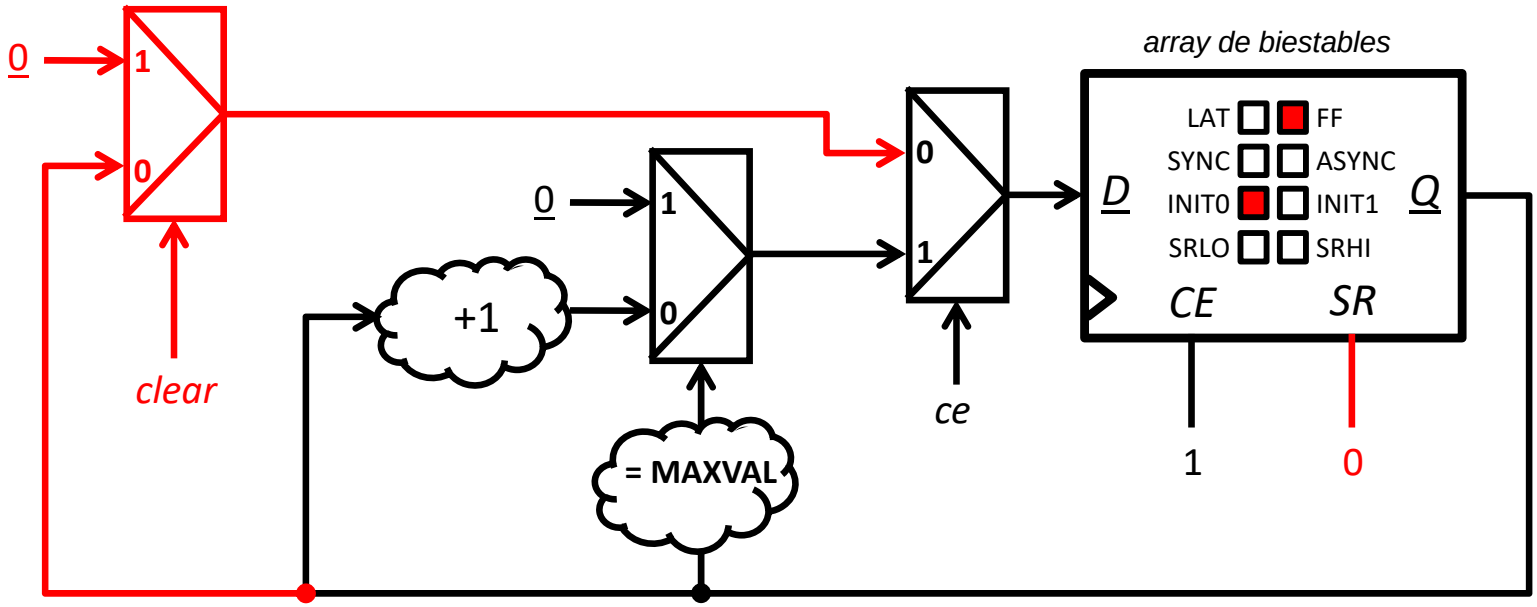
```
process (clk)
  variable cs : unsigned(n-1 downto 0) := (others => '1');
begin
  if rising_edge(clk) then
    if ce='1' then
      cs := (cs + 1) mod MAXVAL then
    elsif clear='1' then
      cs := (others => '0');
    end if;
  end if;
end process;
```



AMD 7 Series FPGAs

diseños sin reset (i)

```
process (clk)
  variable cs : unsigned(n-1 downto 0) := (others => '1');
begin
  if rising_edge(clk) then
    if ce='1' then
      cs := (cs + 1) mod MAXVAL then
    elsif clear='1' then
      cs := (others => '0');
    end if;
  end if;
end process;
```

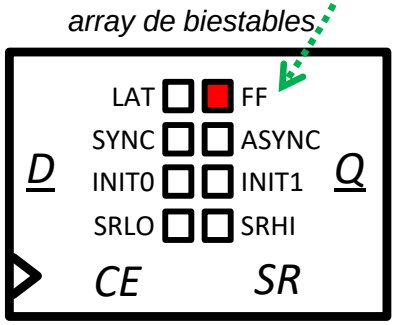


J.M. Mendías
2024

AMD 7 Series FPGAs

diseños sin reset (ii)

```
process (clk)
  variable cs : unsigned(n-1 downto 0) := (others => '0');
begin
  if rising_edge(clk) then
    if ce='1' then
      cs := (cs + 1) mod MAXVAL then
    end if;
  end if;
end process;
```

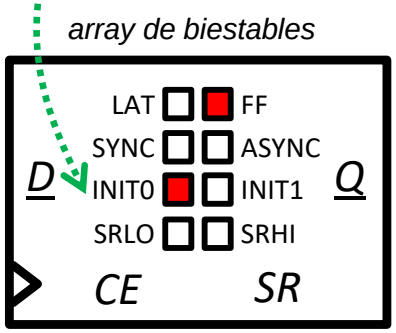


J.M. Mendías
2024

AMD 7 Series FPGAs

diseños sin reset (ii)

```
process (clk)
  variable cs : unsigned(n-1 downto 0) := (others => '0');
begin
  if rising_edge(clk) then
    if ce='1' then
      cs := (cs + 1) mod MAXVAL then
    end if;
  end if;
end process;
```

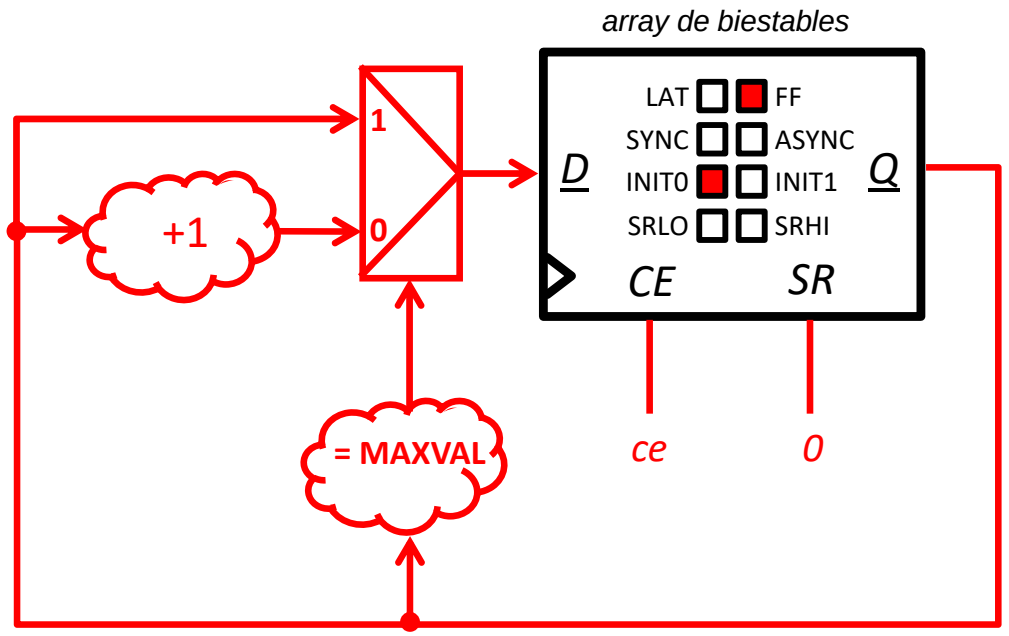


J.M. Mendías
2024

AMD 7 Series FPGAs

diseños sin reset (ii)

```
process (clk)
  variable cs : unsigned(n-1 downto 0) := (others => '0');
begin
  if rising_edge(clk) then
    if ce='1' then
      cs := (cs + 1) mod MAXVAL then
    end if;
  end if;
end process;
```



AMD 7 Series FPGAs

Consejos de codificación (i)

- Limitar la especificación de lógica secuencial con set/rst explícito:
  - Impide su proyección sobre los recursos físicos que carecen de esta capacidad: SRL, LUTRAM, BRAM o registros del DSP48E1 (solo disponen de rst).
  - Reservar el uso de set/rst para lógica secuencial que estrictamente lo requiera (que podrá ser solo proyectada sobre FFD).
- En su lugar, usar preferentemente set/rst implícito:
  - Mediante la declaración de las señales secuenciales con valor inicial.
  - Se infiere el uso de la señal de set/reset global (GSR) que se activa tras la configuración de la FPGA e inicializa todos los biestables.

```
signal q : std_logic;
...
process (clk)
begin
  if rising_edge(clk) then
    if rst='1' then
      q <= '0';
    else
      q <= d;
    end if;
  end if;
end process;
```

✗

```
signal q : std_logic := '0';
...
process (clk)
begin
  if rising_edge(clk) then
    q <= d;
  end if;
end process;
```

✓

AMD 7 Series FPGAs

Consejos de codificación (ii)

- Cuando se requiera un lógica secuencial con inicialización explícita, se recomienda evitar señales de set/rst asíncrono:
  - Algunos recursos físicos no tienen esta capacidad (DSP48E1, BRAM) y otros (LUTRAM y FFD) serían configurados de forma subóptima.

- Además con independencia de que sean síncronas o asíncronas, evitar:
  - La posibilidad dual de set/rst: ninguno de los recursos físicos soportan nativamente esta posibilidad sin añadir lógica y señales adicionales.
  - Que la activación rst/set dependan de lógica operacional.
  - Que sea dinámico el valor asignado a la señal secuencial.

```
process (rst, clk)
begin
  if rst='1' then
    q <= '0';
  elsif rising_edge(clk) then
    q <= d;
  end if;
end process;
```

```
process (rst, set, clk)
begin
  if rst='1' then
    q <= '0';
  elsif set='1' then
    q <= '1';
  elsif rising_edge(clk) then
    q <= d;
  end if;
end process;
```

```
process (rst, ld, clk)
begin
  if rising_edge(clk) then
    if rst='1' and ld='0' then
      q <= init_signal;
    else
      q <= d;
    end if;
  end if;
end process;
```

AMD 7 Series FPGAs

Consejos de codificación (iii)

- Todas las primitivas secuenciales de la familia 7 se controlan con 3 señales de control:
  - SR (set/rst), CK (clock) y CE (clock enable) que además son compartidas por todos los elementos de memoria de un mismo SLICE.
  - CE y SR son nativamente activas a lógica positiva, siendo SR prioritaria.

- Se denomina conjunto de control a una cierta combinación de valores concretos de dichas señales de control.
  - Solo lógica secuencial con idéntico conjunto de control pueden proyectarse sobre un mismo SLICE.

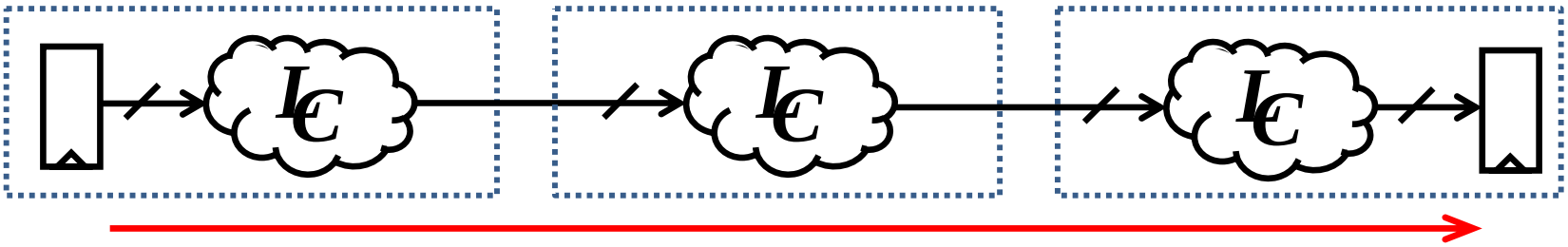
- Para maximizar el aprovechamiento de los slices y minimizar la creación de lógica adicional, debe intentarse especificar lógica secuencial que:
  - Use el menor número de conjuntos de control distintos.
  - Use siempre lógica positiva para las señales de control.
  - Chequee set/rst en primer lugar a otras señales de control.

J.M. Mendías 2024

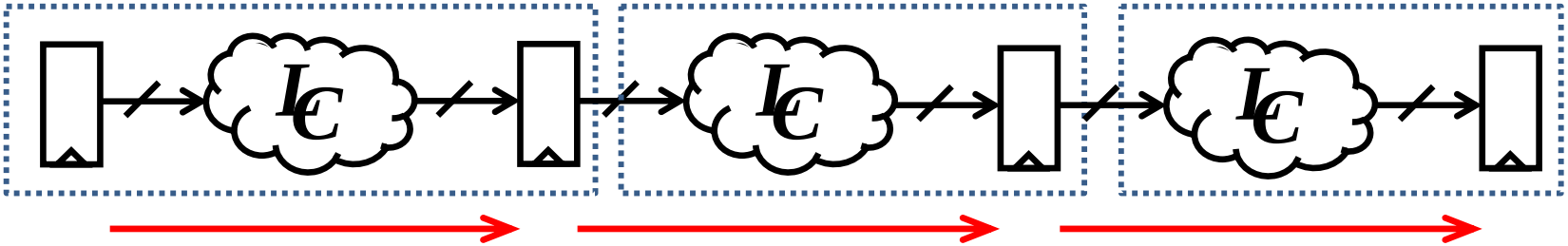
AMD 7 Series FPGAs

Consejos de codificación (iv)

- Las FPGAs disponen de miles de FFs que si no se usan se desperdician:
  - Cada slice incluyen LUTs combinacionales en serie con FFs.
- Es buena técnica registrar las salidas (o entradas) de los circuitos:
  - Sincronizan las salidas evitando propagar transitorios (reduciendo el consumo).
  - Si, además, utilizan alguna señal de carga evitan propagar valores irrelevantes (ídem).
  - Reducen los caminos críticos.



$$t_{clk} \geq t_{critico} = (t_d^{lc_1} + t_d^{lc_2} + t_d^{lc_3})$$



$$t_{clk} \geq t_{critico} = max(t_d^{lc_1}, t_d^{lc_2}, t_d^{lc_3})$$

AMD 7 Series FPGAs

Consejos de codificación (v)

- Tener cuenta algunos números "mágicos" de la familia 7 para evitar la infrautilización de los recursos físicos.
  - Cada LUT tiene 6 entradas:
    - FC de hasta 6 variables se proyectan en 1 nivel LUT.
    - FC de 36 variables (20 a efectos prácticos) se proyectan en 2 niveles de LUT...
    - En cada SLICEL/M puede proyectarse hasta un sumador de 3 operandos de 4 bits.
  - Cada LUT puede configurarse como un MUX 4 a 1:
    - En cada SLICEL/M puede proyectarse hasta un MUX 16 a 1.
  - Cada LUTRAM tiene 64b de capacidad:
    - En cada SLICEM puede proyectarse una RAM de hasta 256b de capacidad.
  - Cada SLR tiene 32b de profundidad:
    - En cada SLICEM puede proyectarse una SLR de hasta 128b de profundidad.
  - Cada BRAM tiene 36Kb de capacidad máxima.
  - En cada DSP48E1 puede proyectarse hasta:
    - Una multiplicación con signo de 18x25b.
    - Una suma/resta con signo de 48b.
    - Un contador de 48b.

Mezclando VHDL

directivas: atributos y pragmas

- La mayor parte de las herramientas EDA pueden ser parcialmente controladas desde el propio código VHDL:
  - Caracterización del entorno de funcionamiento del circuito.
  - Ligaduras u opciones del proceso de síntesis.
  - Control sobre el modo en que se interpretan las construcciones VHDL.

- Habitualmente existe 2 maneras:
  - Mediante atributos VHDL

```
attribute OPT_MODE : string;
attribute OPT_MODE of cronometro : entity is "area"
```

  - Mediante pragmas, comentarios VHDL que tienen un significado especial para la herramienta de síntesis

```
-- pragma translate_off
          ...
-- pragma translate_on
```

Mezclando VHDL

componentes prediseñados (i)

- No todo el diseño debe ser especificado en VHDL:
  - Las herramientas EDA permiten mezclar código con otros mecanismos de especificación
    - Esquemáticos, diagramas de estados, módulos prediseñados, uso de otros lenguajes.
  - Desde el punto de vista VHDL:
    - Estos comportamientos se encapsulan en bibliotecas.
    - Se instancian como componentes.
- Ventajas:
  - Se puede ahorrar tiempo y esfuerzo (discutible, excepto en caso de módulos prediseñados)
- Problemas:
  - Pérdida de portabilidad: VHDL "simulable" es estándar, VHDL "sintetizable" más o menos, los restantes mecanismos de especificación son dependientes de herramienta.
  - Dependencia tecnológica: muchos módulos prediseñados pueden sólo ser aplicables para ciertas tecnologías objetivo.
  - Necesidad de co-simulación: se necesitan la interacción de varios simuladores cada uno especializado en una representación
    - Existen generadores de modelos simulables VHDL.

Mezclando VHDL

componentes prediseñados (ii)

- Existen diferentes tipos de componentes prediseñados:
  - Soft-macros: especificaciones (esquemáticos o descripciones HDL) que se mezclan y sintetizan con el resto de los componentes del sistema.
    - No se puede garantizar su rendimiento.
  - Hard-macros: bloques presintetizados (típicamente netlist) que incluyen datos relativos a emplazamiento y rutado.
    - Su rendimiento puede garantizarse.
  - Hardwired-macros: bloques prefabricados y predifundidos sobre silicio
    - Su rendimiento está completamente caracterizado.
- Los componentes prediseñados pueden
  - Estar almacenados en bibliotecas de módulos o crearse por un generador de módulos.
  - Tener pinout, funcionalidad y rendimiento fijo o parametrizable.
  - La parametrización puede ser realizada explícitamente por el diseñador o implícitamente por la herramientas EDA según las ligaduras de diseño.
  - Su uso puede ser gratuito o puede requerir el pago de licencias (según el uso).
  - Pueden tener diversos grados de complejidad:
    - Primitivos: celdas elementales (AND, FF, ...) proyectables directamente sobre el HW.
    - IP-cores: bloques de alta complejidad diseñados por compañías independientes.

Mezclando VHDL

componentes prediseñados (iii)

- Los componentes prediseñados pueden usarse y parametrizarse por:
  - Instanciación directa.
  - Inferencia a partir de un operador, una función o un fragmento de código que responda a una cierta estructura.

```
library UNIMACRO;
use UNIMACRO.vcomponents.all;

architecture ...;
  ...
begin
  ...
  multiplier1 : MULT_MACRO
    generic map (
      DEVICE => "7SERIES", LATENCY => 0, WIDTH_A => 18, WIDTH_B => 18 )
    port map (
      a => leftOp1, b => rightOp1, p => product1,
      ce => '0', clk => '0', rst => '0'
    );

  multiplier2:
  product2 <= leftOp2 * rigthOp2;
  ...
end ...;
```

en ambos casos se usará el multiplicador 18x25 del recurso predifundido DSP48E1 de la FPGA

Acerca de *Creative Commons*

- Licencia CC (Creative Commons)
  - Ofrece algunos derechos a terceras personas bajo ciertas condiciones. Este documento tiene establecidas las siguientes:

Reconocimiento (*Attribution*):
En cualquier explotación de la obra autorizada por la licencia hará falta reconocer la autoría.

No comercial (*Non commercial*):
La explotación de la obra queda limitada a usos no comerciales.

Compartir igual (*Share alike*):
La explotación autorizada incluye la creación de obras derivadas siempre que mantengan la misma licencia al ser divulgadas.

Más información: https://creativecommons.org/licenses/by-nc-sa/4.0/