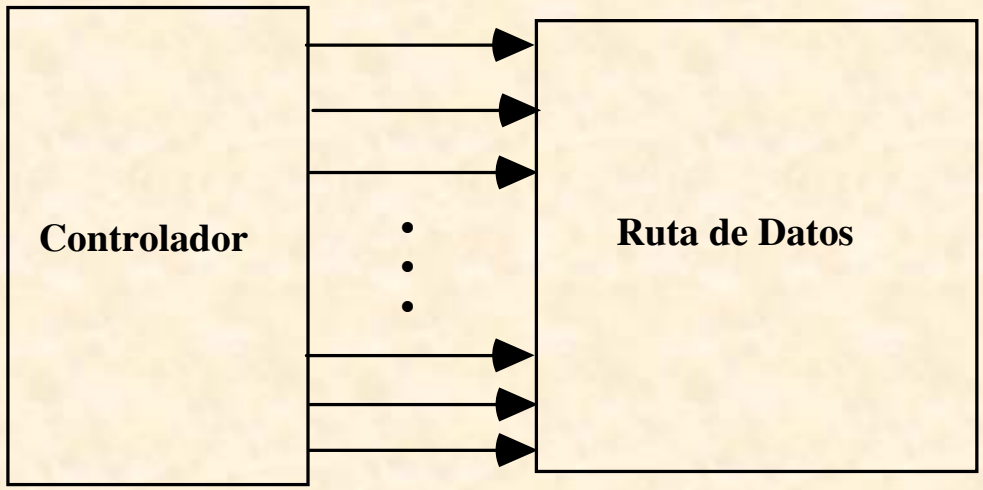
Sesión 5: Diseño Algorítmico de Sistemas Digitales

Diseño de Sistemas Digitales

- Cuando un sistema digital es sencillo se diseña como una red combinacional o secuencial.
- Cuando aumenta su complejidad, el diseño se aborda descomponiendolo en:
  - ***Ruta de datos*** (*datapath*): mantiene y transforma el estado del sistema
  - ***Controlador*** :gobierna las transformaciones en la ruta de datos



Ruta de datos

- La ruta de datos está constituida por:
  - una serie de registros para soportar el estado del sistema
  - una serie de operadores para su transformación
  - una serie de buses de comunicación con una determinada topología.
- Para guiar las transformaciones existen una serie de puntos de control distribuidos por la ruta de datos sobre los que actúa el controlador del sistema.
- Los puntos de control gobiernan:
  - las cargas de los registros
  - las operaciones de transformación
  - la información presente en un determinado bus.

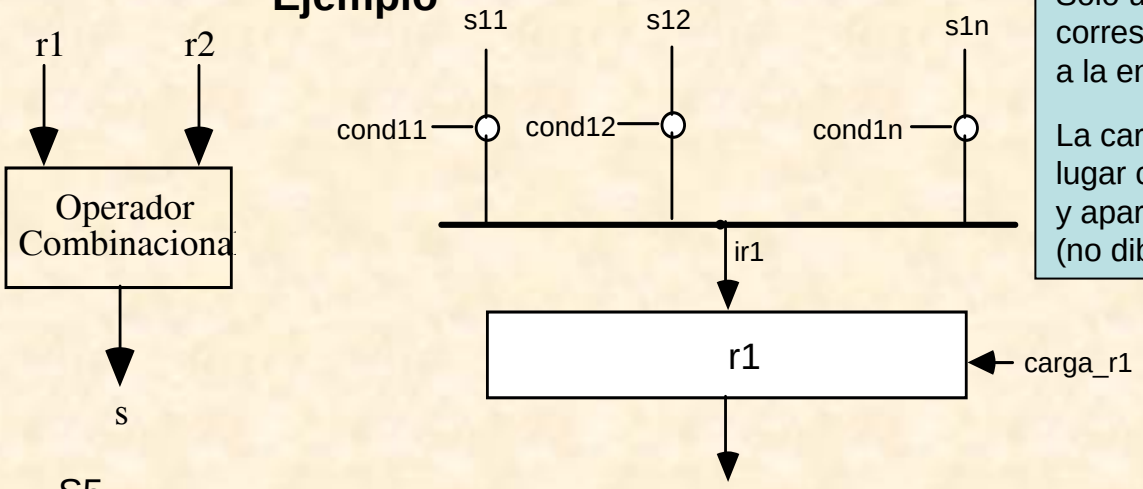
r1 dispone de *n* fuentes *s11, s12,..., s1n* gobernadas por *cond11, cond12,...,con1n.*

La información presente a la entrada *ir1* del registro *r1* (bus fuente de r1) viene determinada por los valores lógicos de las señales de control.

Solo una señal de control deberá valer '1', y su correspondiente fuente será la que se presente a la entrada de *r1*.

La carga de esta información en *r1* tendrá lugar cuando esté activa ('1') la señal *carga_r1* y aparezca unflanco activo por la señal de reloj (no dibujado en la figura)

Ejemplo



Ejemplo: multiplicador binario

Longitud de palabra genérica

Procedimiento de suma-desplazamiento.

En primer lugar declaramos la entidad y definimos una arquitectura de comportamiento.

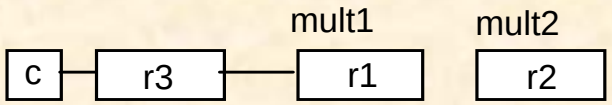
Después construimos una arquitectura algorítmica que, aún siendo de comportamiento, opera con el algoritmo de suma-desplazamiento que utilizará el diseño definitivo.

Algoritmo de suma-desplazamiento: ejemplo (4 bits)

mult1 = 1100 = 12 decimal, **mult2 = 0110 = 6** decimal, resultado **01001000 = 72** decimal.

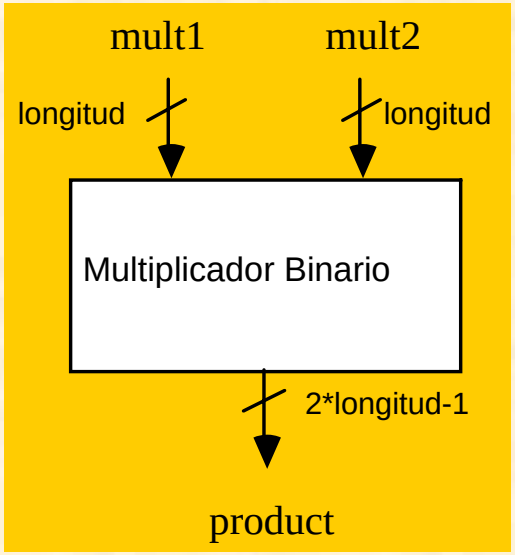
El biestable **c** y los registros **r3** y **r1** tienen capacidad de desplazamiento conjunto a la derecha.

- Inicialmente cargamos **c** con **0**, **r3** con **0000**, **r1** con **mult1** y **r2** con **mult2**.
- Analizamos el bit menos significativo de **r1**:
  - si vale **1** sumamos al contenido **r3** el contenido de **r2**
  - en caso contrario sumamos **0000** (el arrastre lo cargamos en **c**).
- Desplazamos el conjunto **c-r3-r1** una posición a la derecha (se pierde el bit más a la derecha).
- Este proceso se repite **4** veces (el número de bits de los operandos).
- El resultado (doble palabra, **8** bits) será el contenido conjunto de **r3-r1**



c	r3	r1	r2	operación	paso de control
0	0 0 0 0	0 0 0 0	0 0 0 0	valores iniciales	
0	0 0 0 0	1 1 0 **0**	0 1 1 0	carga externa	
0	0 0 0 0	1 1 0 0	0 1 1 0	0000 + 0000	1
0	0 0 0 0	0 1 1 **0**	0 1 1 0	desplaza	2
0	0 0 0 0	0 1 1 0	0 1 1 0	0000 + 0000	3
0	0 0 0 0	0 0 1 **1**	0 1 1 0	desplaza	4
0	0 1 1 0	0 0 1 1	0 1 1 0	0000 + 0110	5
0	0 0 1 1	0 0 0 **1**	0 1 1 0	desplaza	6
0	1 0 0 1	0 0 0 1	0 1 1 0	0011 + 0110	7
0	0 1 0 0	1 0 0 0	0 1 1 0	desplaza	8

Declaración de entidad




```
USE WORK.utilidad.ALL;
ENTITY multiplicador IS
  GENERIC(longitud : integer := 4);
  PORT(mult1, mult2 : IN bit_vector(longitud-1 DOWNTO 0);
       product : OUT bit_vector(2*longitud-1 DOWNTO 0));
END multiplicador;
```

- Consta de una clausula genérica para la longitud de las entradas de operandos
- Una clausula para puertos: dos de entrada y uno de salida de tipo *bit_vector*.
- La declaración va precedida de una sentencia *use* para hacer visible en todas las arquitecturas que definamos para esta entidad las funciones del paquete *utilidad* que definiremos a continuación

Paquete auxiliar

```
PACKAGE utilidad IS
FUNCTION bin_ent (v : bit_vector; l : integer) RETURN integer;
FUNCTION ent_bin (e, l : integer) RETURN bit_vector;
FUNCTION sum_bin (op1, op2 : bit_vector; l : integer) RETURN bit_vector;
END utilidad;
```

```
PACKAGE BODY utilidad IS
-- Función de conversión de binario a entero
FUNCTION bin_ent  (v : bit_vector; l : integer) RETURN integer IS
  VARIABLE int_var : integer := 0;
BEGIN
  FOR i IN 0 TO l-1 LOOP
    IF ( v(i) = '1') THEN int_var := int_var + (2**i);
    END IF;
  END LOOP;
  RETURN int_var;
END bin_ent;
-- Función de conversión de entero a binario
FUNCTION ent_bin  (e, l : integer) RETURN bit_vector IS
  VARIABLE int_var : bit_vector(l-1 DOWNTO 0);
  VARIABLE temp1 : integer := 0;
  VARIABLE temp2 : integer := 0;
BEGIN
  temp1 := e;
  FOR i IN l-1 DOWNTO 0 LOOP
    temp2 := temp1/(2**i);
    temp1 := temp1 mod (2**i);
    IF ( temp2 = 1 ) THEN int_var(i) := '1';
                    ELSE int_var(i) := '0';
    END IF;
  END LOOP;
  RETURN int_var;
END ent_bin;
```

Define las tres funciones auxiliares siguientes:

bin_ent : transforma un vector binario en entero;

ent_bin**:** transforma un entero en vector binario

sum_bin**:** producir la suma binaria

```
-- Función de suma binaria
FUNCTION sum_bin (op1, op2 : bit_vector; l : integer) RETURN
bit_vector IS
  VARIABLE s : bit_vector(l DOWNTO 0);
BEGIN
  s := ent_bin((bin_ent(op1,l) + bin_ent(op2,l)),l+1);
  RETURN s;
END sum_bin;
END utilidad;
```

Arquitectura de comportamiento funcional

- Esta primera arquitectura es una arquitectura de comportamiento funcional puro en la que las entradas binarias se convierten a enteros, se realiza la multiplicación de enteros '*' y se vuelve a convertir el resultado a binario.

- Se utilizan dos de las tres funciones definidas en el paquete (*utilidad*).

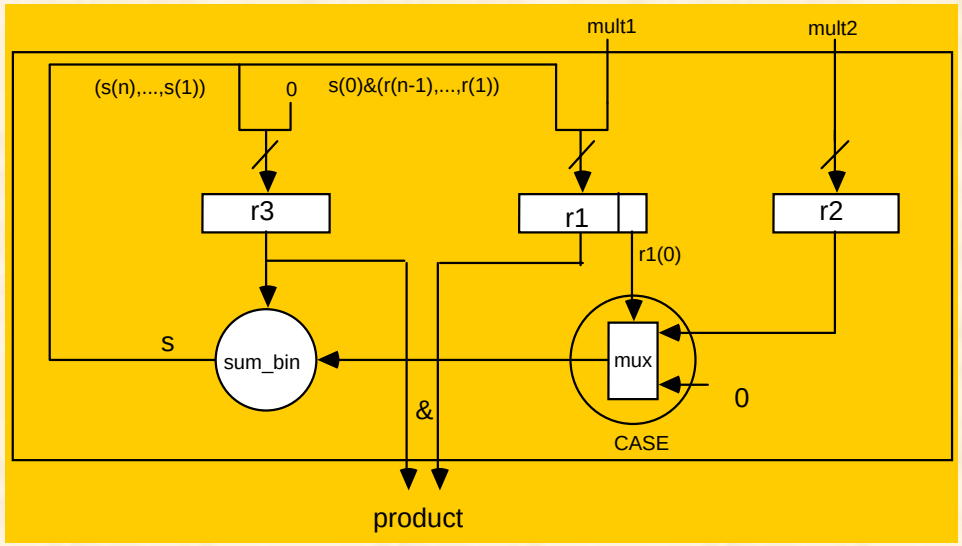
```
ARCHITECTURE comporta OF multiplicador IS
BEGIN
  PROCESS(mult1, mult2)
    VARIABLE p : integer;
  BEGIN
    p := bin_ent(mult1, longitud)*bin_ent(mult2, longitud);
    product <= ent_bin(p,2*longitud);
  END PROCESS;
END comporta;
```

- Con una arquitectura de este tipo se valida la especificación del dispositivo a diseñar y sirve de referencia para comprobar el correcto funcionamiento de las siguientes arquitecturas que irán aproximándose en su estructura a la del dispositivo real.

- Además, esta arquitectura supone para el simulador poca carga de computación, por lo que se podrá utilizar para instanciar componentes de su tipo en una arquitectura estructural compleja.

Arquitectura algorítmica

- Esta segunda arquitectura, aunque consta como la anterior de un único proceso, su algoritmo refleja ya la estructura interna y el funcionamiento de suma-desplazamiento del multiplicador.
- Se utilizan tres variables vectoriales (*array*) *r1*, *r2*, y *r3* para los tres registros básicos del multiplicador.
- La sentencia de asignación de variable opera como mecanismo de carga de los registros
- Los valores individuales (indexados) asignados a las variables *r1* y *r2* implementan el desplaz. derecho.
- La suma binaria se realiza con la función *sum_bi* y el multiplexor con una sentencia *case*.



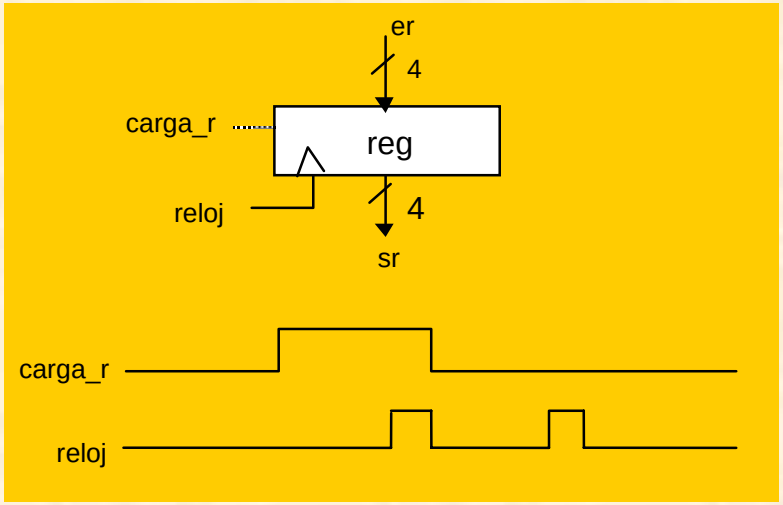

```
USE WORK.utilidad.ALL;
ARCHITECTURE algoritmo OF multiplicador IS
BEGIN
PROCESS(mult1, mult2)
  VARIABLE r1, r2, r3 : bit_vector(longitud-1 DOWNTO 0);
  VARIABLE s : bit_vector(longitud DOWNTO 0);
  CONSTANT cero : bit_vector(longitud-1 DOWNTO 0)
                                        :=(OTHERS =>'0');
BEGIN
  r1 := mult1;
  r2 := mult2;
  r3 := (OTHERS => '0');
  FOR i IN r1'REVERSE_RANGE LOOP
    CASE r1(0) IS
      WHEN '1' => s := sum_bin(r3, r2, longitud);
      WHEN '0' => s := sum_bin(r3, cero, longitud);
    END CASE;
    r3 := s(s'LEFT DOWNTO 1);
    r1 := s(0)&r1(r1'LEFT DOWNTO 1);
  END LOOP;
  product <= r3&r1;
END PROCESS;
END algoritmo;
```

Arquitectura estructural

Registro

Ruta de datos: componentes

- Realizan la operación correspondiente a la línea de control con valor *1* en el instante que aparece un flanco positivo en la señal de reloj.
- *r1* utiliza una única línea de control: *carga_r*.
- Diagrama de bloques y evolución de las señales de control correspondientes a la carga del registro con el valor de su entrada paralela *er* a la llegada del primer flanco positivo del *reloj*:



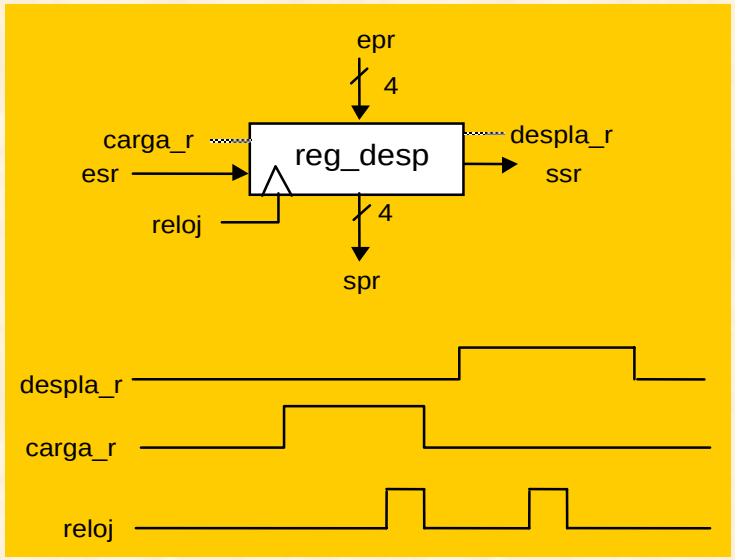

```
USE STD.TEXTIO.ALL;
ENTITY reg IS
   GENERIC (longitud : natural := 8;
                  nombre : STRING := "<r2>=");
   PORT (  er : IN bit_vector(longitud-1 DOWNTO 0);
         reloj, carga_r : IN bit;
         sr : OUT bit_vector(longitud-1 DOWNTO 0));
END reg;

ARCHITECTURE comporta OF reg IS
BEGIN
   PROCESS
      VARIABLE r : bit_vector(longitud-1 DOWNTO 0);
      VARIABLE l : LINE;
   BEGIN
      WAIT UNTIL (reloj = '1') AND (reloj'EVENT);
      IF carga_r = '1' THEN r := er; END IF;
      sr <= r;
      --
        WRITE(l, nombre);
        WRITE(l,r);
        WRITELINE(OUTPUT,l);
      --
   END PROCESS;
END comporta;
```

Ruta de datos: componentes

Registro de desplazamiento

- Los registros de desplazamiento utilizan dos señales de control, *carga_r*, idéntica al registro anterior, y *despla_r*, que produce el desplazamiento del contenido del registro una unidad a la derecha cuando aparece un flanco positivo en la señal de reloj.

- Diagrama de bloques y evolución de las señales de control correspondientes a una carga paralela seguida de un desplazamiento:

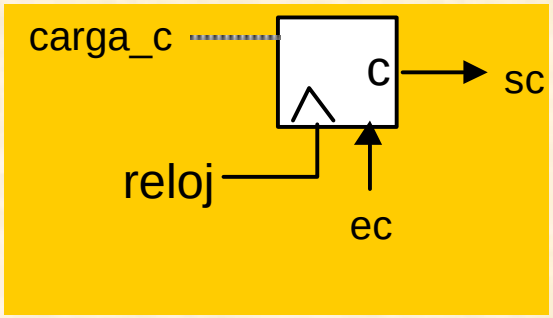



```
USE STD.TEXTIO.ALL;
ENTITY reg_desp IS
   GENERIC (longitud : natural := 8; nombre : STRING := "<r1>=");
   PORT (  epr : IN bit_vector(longitud-1 DOWNTO 0);
         reloj, carga_r, despla_r : IN bit;
                                    esr : IN bit;
                                    ssr : OUT bit;
         spr : OUT bit_vector(longitud-1 DOWNTO 0));
END reg_desp;
ARCHITECTURE comporta OF reg_desp IS
BEGIN
   PROCESS
      VARIABLE r : bit_vector(longitud-1 DOWNTO 0);
      VARIABLE l : LINE;
   BEGIN
      WAIT UNTIL (reloj = '1') AND (reloj'EVENT);
      IF carga_r = '1' THEN r := epr; END IF;
                  IF despla_r = '1' THEN r := esr&r(longitud-1 DOWNTO 1);
END IF;
      spr <= r;
                  ssr <= r(0);
      --
        WRITE(l, nombre);
        WRITE(l,r);
        WRITELINE(OUTPUT,l);
      --
   END PROCESS;
END comporta;
```

Ruta de datos: componentes

Biestable

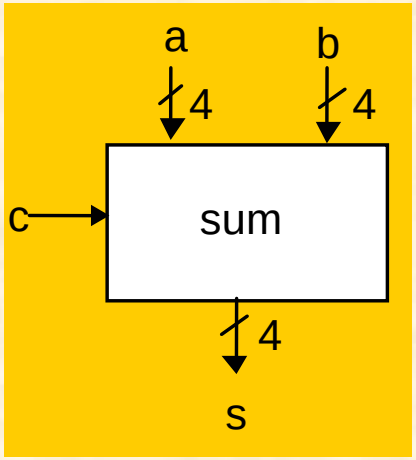
Diagrama de bloques del biestable, cuyo comportamientoes como el del registro *r1* con la diferencia de la longitud del dato, que en este caso es 1




```
USE STD.TEXTIO.ALL;
ENTITY ff IS
   GENERIC (nombre : STRING := "<c>");
   PORT (  ec : IN bit;
           reloj, carga_r : IN bit;
           sc : OUT bit);
END ff;
ARCHITECTURE comporta OF ff IS
BEGIN
   PROCESS
      VARIABLE r : bit;
      VARIABLE l : LINE;
   BEGIN
      WAIT UNTIL (reloj = '1') AND (reloj'EVENT);
      IF carga_r = '1' THEN r := ec; END IF;
      sc <= r;
      --
        WRITE(l, nombre);
        WRITE(l,r);
        WRITELINE(OUTPUT,l);
      --
   END PROCESS;
END comporta;
```

Ruta de datos: componentes

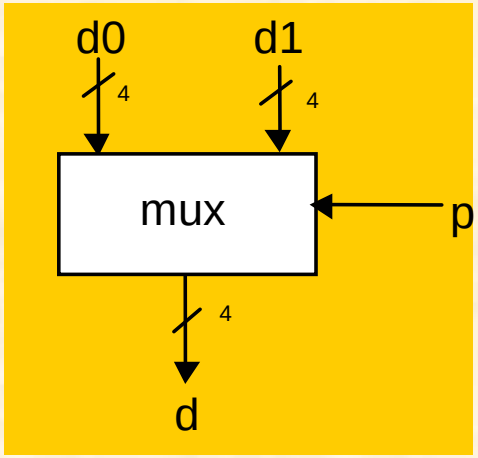
Sumador




```
USE WORK.utilidad.ALL;
ENTITY sum IS
   GENERIC (longitud : natural := 8);
   PORT(   a, b : IN bit_vector(longitud-1 DOWNTO 0);
           s : OUT bit_vector(longitud-1 DOWNTO 0);
           c : OUT bit);
END sum;
ARCHITECTURE comporta OF sum IS
BEGIN
   PROCESS(a,b)
   VARIABLE as : bit_vector(8 DOWNTO 0);
   BEGIN
      as := sum_bin(a, b, longitud);
      c <= as(longitud);
              s <= as(longitud-1 DOWNTO 0);
   END PROCESS;
END comporta;
```

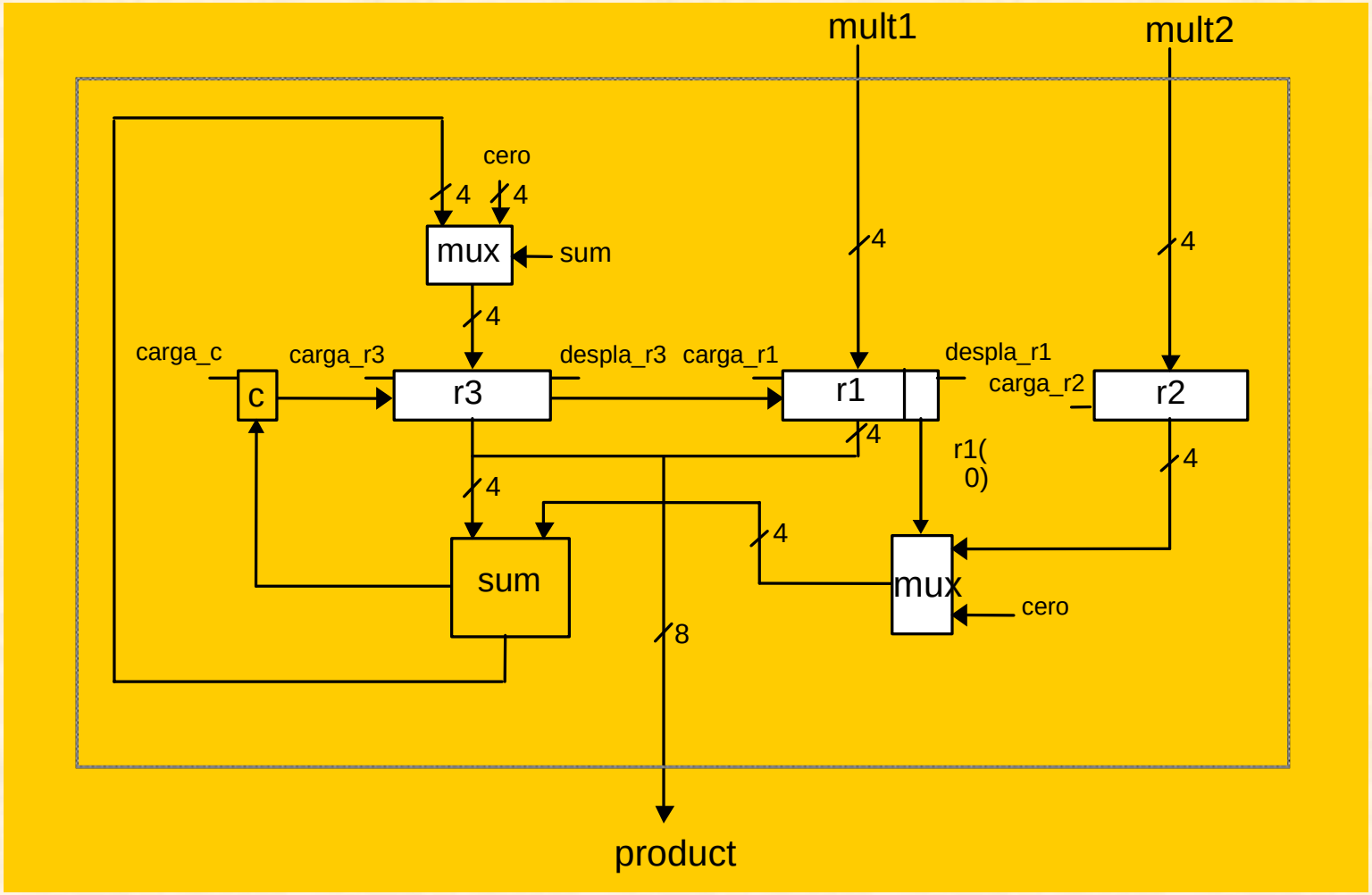
Ruta de datos: componentes

Multiplexor




```
ENTITY mux IS
   GENERIC(longitud : natural :=8);
   PORT (d0, d1 : IN bit_vector(longitud-1 DOWNTO 0);
         z : OUT bit_vector(longitud-1 DOWNTO 0);
         sel : IN bit);
END mux;
ARCHITECTURE comportamiento OF mux IS
BEGIN
  z <= d0 WHEN sel = '0' ELSE d1;
END comportamiento;
```

Arquitectura estructural de la ruta de datos (esquema)



Arquitectura estructural de la ruta de datos (código)

```
USE WORK.tipos_control_st.ALL;

ENTITY ruta_mul IS
   GENERIC (longitud : natural := 8);
   PORT (mult1,mult2 :IN bit_vector(longitud-1 DOWNTO 0);
         control : IN bus_control;
         reloj : IN bit;
         product : OUT bit_vector(2*longitud-1 DOWNTO 0));
END ruta_mul;

ARCHITECTURE estructura OF ruta_mul IS

COMPONENT mux
GENERIC(longitud : natural :=8);
PORT (d0, d1 : IN bit_vector(longitud-1 DOWNTO 0);
     z : OUT bit_vector(longitud-1 DOWNTO 0);
     sel : IN bit);
END COMPONENT;

COMPONENT reg
   GENERIC (longitud : natural := 8;
                nombre : STRING := "<r2>=");
   PORT (  er : IN bit_vector(longitud-1 DOWNTO 0);
         reloj, carga_r : IN bit;
         sr : OUT bit_vector(longitud-1 DOWNTO 0));
END COMPONENT;

COMPONENT reg_desp
   GENERIC (longitud : natural := 8;
                nombre : STRING := "<r1>=");
   PORT (  epr : IN bit_vector(longitud-1 DOWNTO 0);
         reloj, carga_r, despla_r : IN bit;
                                    esr : IN bit;
                                    ssr : OUT bit;
         spr : OUT bit_vector(longitud-1 DOWNTO 0));
END COMPONENT;
```

```
COMPONENT ff
   GENERIC (nombre : STRING := "<c>=");
   PORT (  ec : IN bit;
         reloj, carga_r : IN bit;
         sc : OUT bit);
END COMPONENT;
COMPONENT sum
   GENERIC (longitud : natural := 8);
   PORT(   a, b : IN bit_vector(longitud-1 DOWNTO 0);
         s : OUT bit_vector(longitud-1 DOWNTO 0);
         c : OUT bit);
END COMPONENT;
SIGNAL s1, s2, s3, s4, s5, s6, s7,
          s8 : bit_vector(longitud-1 DOWNTO 0);
SIGNAL b1, b2, b3, b4 : bit;
SIGNAL cero : bit_vector(longitud-1 DOWNTO 0);

BEGIN
 multiplexor1 : mux   GENERIC MAP(longitud)
                      PORT MAP(cero, s1, s7, control(suma));
 multiplexor2 : mux   GENERIC MAP(longitud)
                      PORT MAP(cero, s2, s5, b3);
 registro1: reg_desp  GENERIC MAP(longitud, "<r1>=")
                      PORT MAP (mult1, reloj,
                                control(carga_r1),
                                control(despla_r1), b2, b3, s8);
 registro2: reg       GENERIC MAP(longitud, "<r2>=")
                      PORT MAP (mult2, reloj, control(carga_r2), s2);
 registro3: reg_desp  GENERIC MAP(longitud, "<r3>=")
                      PORT MAP (s7, reloj, control(carga_r3),
                                control(despla_r3), b1, b2, s6);
 biestable: ff        GENERIC MAP("<c>=")
                      PORT MAP(b4, reloj, control(carga_c), b1);
 sumador : sum        GENERIC MAP(longitud)
                      PORT MAP(s5, s6, s1, b4);
   product <= s6&s8;
END estructura;
```

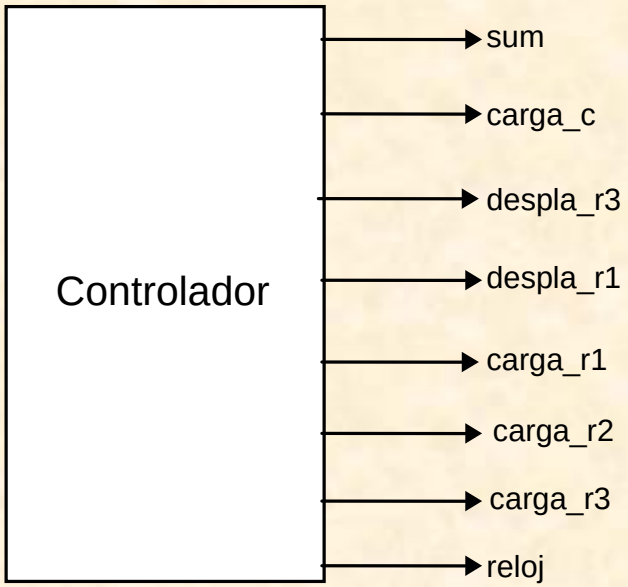
Paquete de tipos de control

```
PACKAGE tipos_control_st IS
  TYPE senales_control IS (suma,carga_r1,carga_r2,carga_r3, carga_c,despla_r1,despla_r3);
  TYPE vector_senales_control IS ARRAY (natural RANGE <>) OF senales_control;
  TYPE bus_control IS ARRAY (senales_control) OF bit;
  FUNCTION proyec (ent : vector_senales_control) RETURN bus_control;
  FUNCTION proyec (ent : senales_control) RETURN bus_control;
  FUNCTION proyec RETURN bus_control;
END tipos_control_st;

PACKAGE BODY tipos_control_st IS
  FUNCTION proyec(ent : vector_senales_control) RETURN bus_control IS
    VARIABLE res : bus_control := (OTHERS => '0');
  BEGIN
    FOR i IN ent'RANGE LOOP res(ent(i)) := '1'; END LOOP;
    RETURN res;
  END proyec;
FUNCTION proyec(ent : senales_control) RETURN bus_control IS
  VARIABLE res : bus_control := (OTHERS => '0');
BEGIN
  res(ent) := '1';
  RETURN res;
END proyec;
FUNCTION proyec RETURN bus_control IS
  VARIABLE res : bus_control := (OTHERS => '0');
BEGIN
  RETURN res;
END proyec;
END tipos_control_st;
```

El modelo estructural utiliza el siguiente paquete de tipos de control (*tipos_control_st*) análogo al utilizado en el modelo de flujo de datos pero contemplando las señales de control de la ruta de datos estructural

Controlador

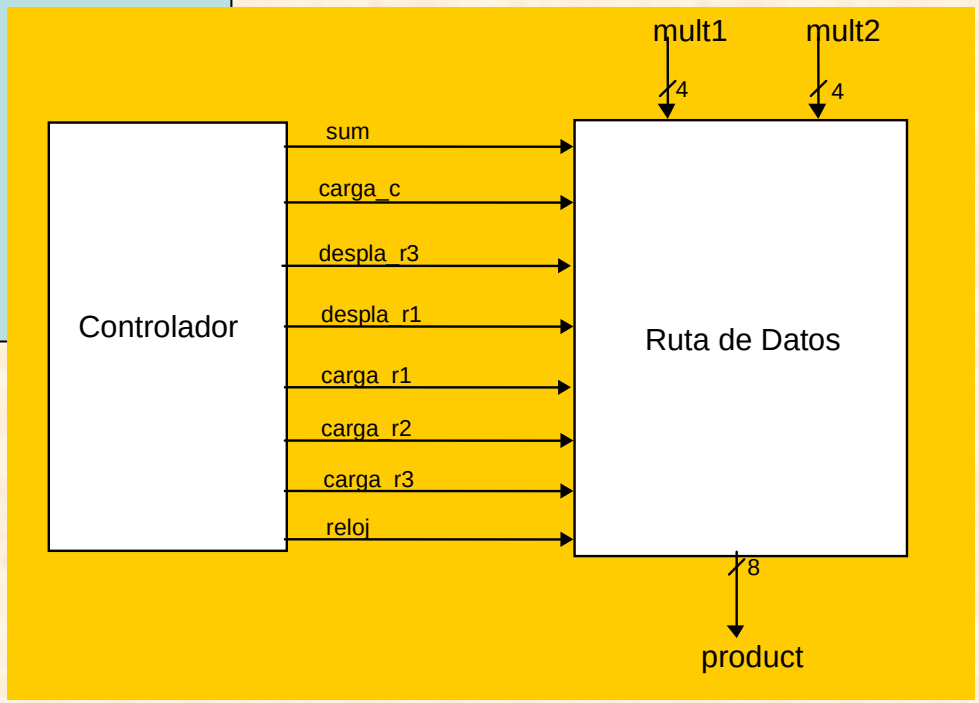
Utilizando el paquete *tipos_control_st* podemos diseñar un controlador de comportamiento en el que quedan explícitas las señales que intervienen en los sucesivos pasos de control




```
USE WORK.tipos_control_st.ALL;
ENTITY controlador_st IS
  GENERIC(longitud : natural := 8);
  PORT(control : OUT bus_control; reloj : INOUT bit; reset : IN bit);
END controlador_st;
ARCHITECTURE mef OF controlador_st IS
BEGIN
  PROCESS(reloj,reset)
    TYPE estados IS (carga, suma, despla);
    VARIABLE estado : estados;
    VARIABLE contador : NATURAL;
  BEGIN
    IF reset = '1'
                    THEN
                     control <= "0000000";
                     estado := carga;
                     contador := longitud + 1;
                    ELSE
                     IF reloj = '1' AND reloj'EVENT THEN
                      IF contador > 1
                        THEN
                         CASE estado IS
          WHEN carga   =>        control <= proyec(carga_r1 & carga_r2);
                                 estado := suma;
          WHEN suma    =>        control <= proyec(carga_r3 & carga_c & suma);
                                 estado := despla;
          WHEN despla  =>        control <= proyec(despla_r1 & despla_r3 & suma);
                                 estado := suma;
                                 contador := contador-1;
        END CASE;
       ELSE
                      control <= proyec;
       END IF;
      END IF;
    END IF;
  END PROCESS;
  reloj <= NOT reloj AFTER 5 ns;
END mef;
```

Conexión estructural Controlador-Ruta de datos

```
USE WORK.tipos_control_st.ALL;
ARCHITECTURE estructural OF multiplicador IS
COMPONENT ruta_mul
   GENERIC (longitud : natural := 8);
   PORT (mult1,mult2 :IN bit_vector(longitud-1 DOWNTO 0);
                      control : IN bus_control;
                      reloj : IN bit;
                      product : OUT bit_vector(2*longitud-1 DOWNTO 0));
END COMPONENT;
COMPONENT controlador_st
  GENERIC(longitud : natural := 8);
  PORT(control : OUT bus_control; reloj : INOUT bit; reset : IN bit);
END COMPONENT;
 SIGNAL reloj : bit;
 SIGNAL control : bus_control;
BEGIN
 rd : ruta_mul
   GENERIC MAP (8)
   PORT MAP (mult1, mult2, control, reloj, product);
 ct : controlador_st
   GENERIC MAP (8)
   PORT MAP (control, reloj, reset);
END estructural;
```



Prueba del modelo

Valores: *mult1 = 00001010*, *mult2 = 00000011*, introducidos en el instante de tiempo *0*.
El pulso de *reset* se da a los 5 ns. y dura 10 ns.
La evolución de todas las señales del modelo, desde el instante 0 hasta que se estabiliza el resultado correcto en la señal de salida *product* es la siguiente:

ns	delta	mult1	mult2	product	reset	reloj	control
0	+0	00000000	00000000	0000000000000000	0	0	0000000
0	+1	00001010	00000011	0000000000000000	0	0	0000000
5	+0	00001010	00000011	0000000000000000	1	1	0000000
10	+0	00001010	00000011	0000000000000000	1	0	0000000
15	+0	00001010	00000011	0000000000000000	0	1	0000000
15	+1	00001010	00000011	0000000000000000	0	1	0110000
20	+0	00001010	00000011	0000000000000000	0	0	0110000
25	+0	00001010	00000011	0000000000000000	0	1	0110000
25	+1	00001010	00000011	0000000000000000	0	1	1001100
25	+2	00001010	00000011	0000000000001010	0	1	1001100
30	+0	00001010	00000011	0000000000001010	0	0	1001100
35	+0	00001010	00000011	0000000000001010	0	1	1001100
35	+1	00001010	00000011	0000000000001010	0	1	1000011
40	+0	00001010	00000011	0000000000001010	0	0	1000011
45	+0	00001010	00000011	0000000000001010	0	1	1000011
45	+1	00001010	00000011	0000000000001010	0	1	1001100
45	+2	00001010	00000011	0000000000000101	0	1	1001100
50	+0	00001010	00000011	0000000000000101	0	0	1001100
55	+0	00001010	00000011	0000000000000101	0	1	1001100
55	+1	00001010	00000011	0000000000000101	0	1	1000011
55	+2	00001010	00000011	0000001100000101	0	1	1000011
60	+0	00001010	00000011	0000001100000101	0	0	1000011
65	+0	00001010	00000011	0000001100000101	0	1	1000011
65	+1	00001010	00000011	0000001100000101	0	1	1001100
65	+2	00001010	00000011	0000000110000010	0	1	1001100
70	+0	00001010	00000011	0000000110000010	0	0	1001100
75	+0	00001010	00000011	0000000110000010	0	1	1001100
75	+1	00001010	00000011	0000000110000010	0	1	1000011
80	+0	00001010	00000011	0000000110000010	0	0	1000011
85	+0	00001010	00000011	0000000110000010	0	1	1000011
85	+1	00001010	00000011	0000000110000010	0	1	1001100
85	+2	00001010	00000011	0000000011000001	0	1	1001100
90	+0	00001010	00000011	0000000011000001	0	0	1001100
95	+0	00001010	00000011	0000000011000001	0	1	1001100
95	+1	00001010	00000011	0000000011000001	0	1	1000011
95	+2	00001010	00000011	0000001111000001	0	1	1000011
100	+0	00001010	00000011	0000001111000001	0	0	1000011
105	+0	00001010	00000011	0000001111000001	0	1	1000011
105	+1	00001010	00000011	0000001111000001	0	1	1001100
105	+2	00001010	00000011	0000000111100000	0	1	1001100
110	+0	00001010	00000011	0000000111100000	0	0	1001100
115	+0	00001010	00000011	0000000111100000	0	1	1001100
115	+1	00001010	00000011	0000000111100000	0	1	1000011
120	+0	00001010	00000011	0000000111100000	0	0	1000011
125	+0	00001010	00000011	0000000111100000	0	1	1000011
125	+1	00001010	00000011	0000000111100000	0	1	1001100
125	+2	00001010	00000011	0000000011110000	0	1	1001100
130	+0	00001010	00000011	0000000011110000	0	0	1001100
135	+0	00001010	00000011	0000000011110000	0	1	1001100
135	+1	00001010	00000011	0000000011110000	0	1	1000011
140	+0	00001010	00000011	0000000011110000	0	0	1000011
145	+0	00001010	00000011	0000000011110000	0	1	1000011
145	+1	00001010	00000011	0000000011110000	0	1	1001100
145	+2	00001010	00000011	0000000001111000	0	1	1001100
150	+0	00001010	00000011	0000000001111000	0	0	1001100
155	+0	00001010	00000011	0000000001111000	0	1	1001100
155	+1	00001010	00000011	0000000001111000	0	1	1000011
160	+0	00001010	00000011	0000000001111000	0	0	1000011
165	+0	00001010	00000011	0000000001111000	0	1	1000011
165	+1	00001010	00000011	0000000001111000	0	1	1001100
165	+2	00001010	00000011	0000000000111100	0	1	1001100
170	+0	00001010	00000011	0000000000111100	0	0	1001100
175	+0	00001010	00000011	0000000000111100	0	1	1001100
175	+1	00001010	00000011	0000000000111100	0	1	1000011
180	+0	00001010	00000011	0000000000111100	0	0	1000011
185	+0	00001010	00000011	0000000000111100	0	1	1000011
185	+1	00001010	00000011	0000000000111100	0	1	0000000
185	+2	00001010	00000011	0000000000011110	0	1	0000000

Practica 5 : Diseño Algorítmico con VHDL

Objetivos:

Diseño de sistemas digitales de tamaño mediano que requieren un planteamiento algorítmico (ruta de datos + control) utilizando dos estilos de descripción VHDL: comportamiento y estructural.

Práctica a realizar:

Diseñar un divisor binario utilizando los dos niveles de descripción en VHDL:

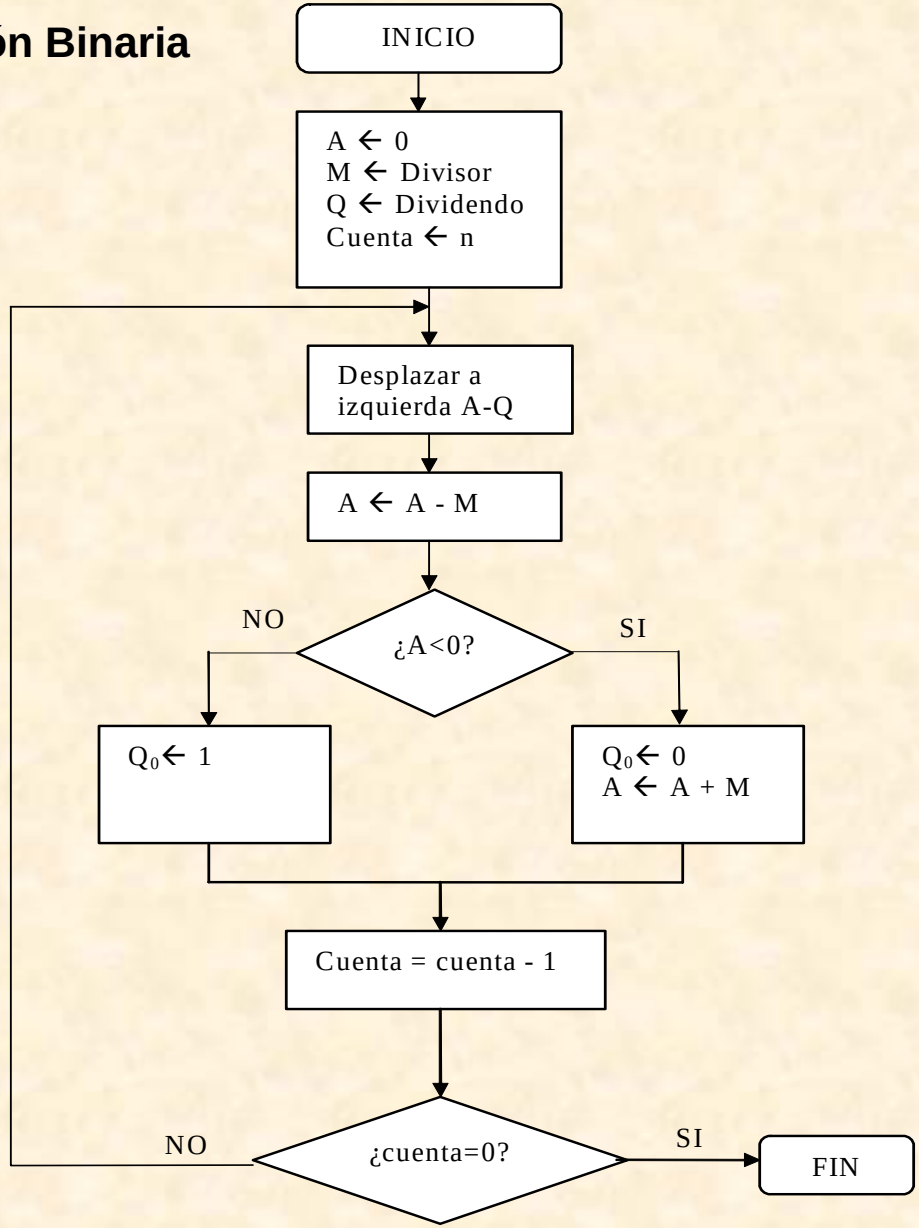
1. Comportamiento
2. Estructural

Resultados a entregar:

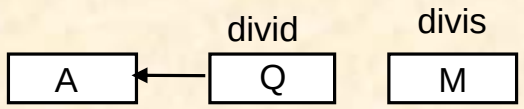
Documentación del diseño incluyendo:

1. Especificación completa y precisa del divisor.
2. Listado VHDL comentado de los programas correspondientes a cada nivel de descripción.
3. Tests de entrada/salida que muestren el correcto funcionamiento del divisor en cada nivel.

Algoritmo de División Binaria



Ejemplo de División Binaria



		Q_0		operación	paso control
	0 0 0 0	0 0 0 0	0 0 0 0	valores iniciales	
	0 0 0 0	**0 1 1 1**	**0 0 1 1**	carga externa	
	0 0 0 0	1 1 1 0	0 0 1 1	←	1
<	1 1 0 1	1 1 1 **0**	0 0 1 1	A - M	2
	0 0 0 0	1 1 1 0	0 0 1 1	A + M	3
	0 0 0 1	1 1 0 0	0 0 1 1	←	4
<	1 1 1 0	1 1 0 **0**	0 0 1 1	A - M	5
	0 0 0 1	1 1 0 0	0 0 1 1	A + M	6
	0 0 1 1	1 0 0 0	0 0 1 1	←	7
=	0 0 0 0	1 0 0 **1**	0 0 1 1	A - M	8
	0 0 0 1	0 0 1 0	0 0 1 1	←	9
<	1 1 1 0	0 0 1 **0**	0 0 1 1	A - M	10
	0 0 0 1	**0 0 1 0**	0 0 1 1	A + M	11
	resto	cociente			

Dividendo: $7 = 0\,1\,1\,1_{(2}$

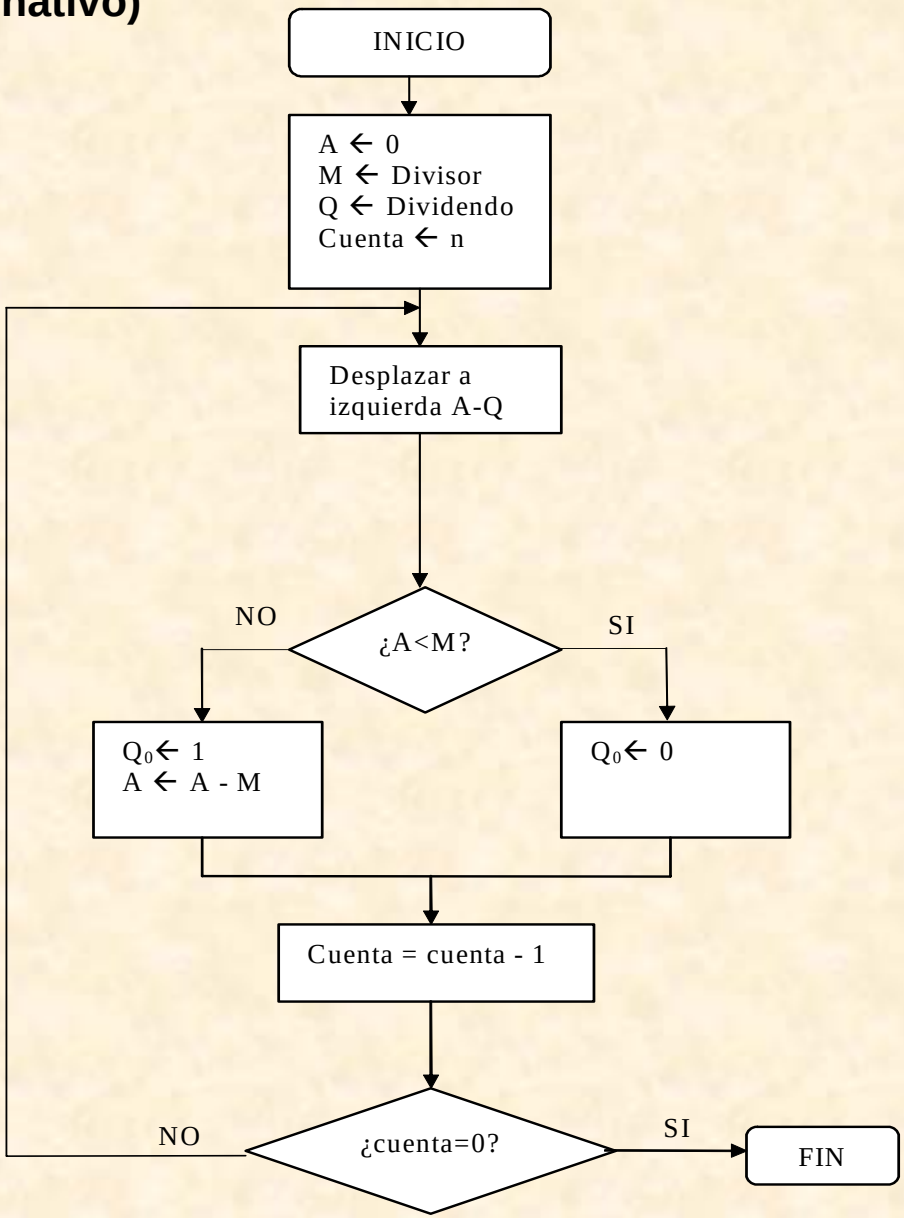
Divisor: $3 = 0\,0\,1\,1_{(2}$

Cociente: $2 = 0\,0\,1\,0_{(2}$

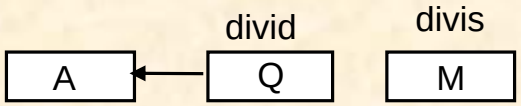
Resto $1 = 0\,0\,0\,1_{(2}$

Algoritmo de División Binaria (alternativo)

- En lugar de restar sistemáticamente A – M y restaurar la resta sumando A + M cuando el resultado es negativo (A < 0), podemos comparar A con M y restar sólo cuando no se cumple que A < M
- El resto del algoritmo permanece igual
- En este caso en lugar de un sumador/restador binario como era necesario en el algoritmo anterior, utilizaremos un comparador y un restador



Ejemplo de División Binaria



Dividendo:	7 = 0 1 1 1$_{(2}$
Divisor:	3 = 0 0 1 1$_{(2}$
Cociente:	2 = 0 0 1 0$_{(2}$
Resto	1 = 0 0 0 1$_{(2}$

		Q0		operación	paso control
	0 0 0 0	0 0 0 0	0 0 0 0	valores iniciales	
	0 0 0 0	**0 1 1 1**	**0 0 1 1**	carga externa	
	0 0 0 0	1 1 1 0	0 0 1 1	←	1
si	0 0 0 0	1 1 1 **0**	0 0 1 1	A < M	2
	0 0 0 1	1 1 0 0	0 0 1 1	←	3
si	0 0 0 1	1 1 0 **0**	0 0 1 1	A < M	4
	0 0 1 1	1 0 0 0	0 0 1 1	←	5
no	0 0 1 1	1 0 0 **1**	0 0 1 1	A < M	6
	0 0 0 0	1 0 0 1	0 0 1 1	A ← A - M	7
	0 0 0 1	0 0 1 0	0 0 1 1	←	8
si	**0 0 0 1**	**0 0 1 0**	0 0 1 1	A < M	9