CAPÍTULO 4
DISEÑO DE SISTEMAS DIGITALES

DISEÑO DE UN SISTEMA DIGITAL ORDENADOR DE NUMEROS

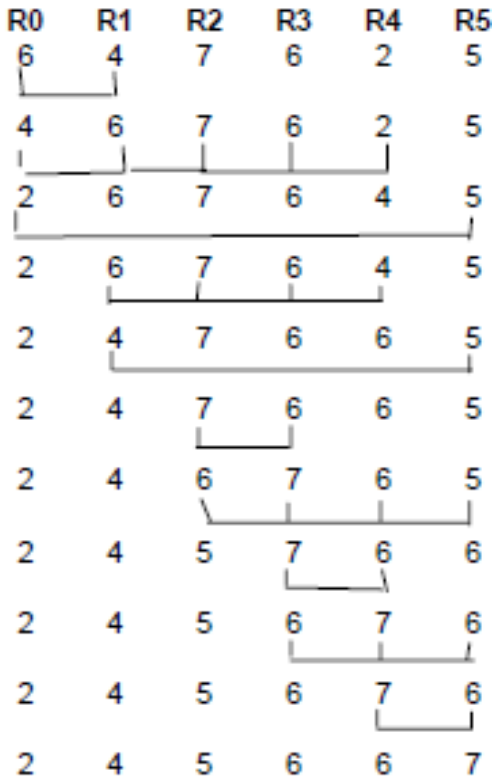
Diseñar un circuito **Ordenador de Números:** Dados **k** números de **n-bits** cada uno almacenados en k registros **Ri (R0,…, Rk-1)** se desea ordenarlos en forma ascendente.

Los datos vienen en la entrada DataIn, cuando WrInit es verdadera, el dato presente en DataIn, se graba en el registro que tenga la dirección Radd.

Para mostrar los datos la señal Rd debe ser verdadera, en ese momento el se muestra en la salida DataOut el dato presente en el registro de dirección Radd.

La idea de solución es comparar cada dato de cada registro con todos los registros siguientes. Si el nuevo dato es menor, se realiza un intercambio de datos, sino queda igual.

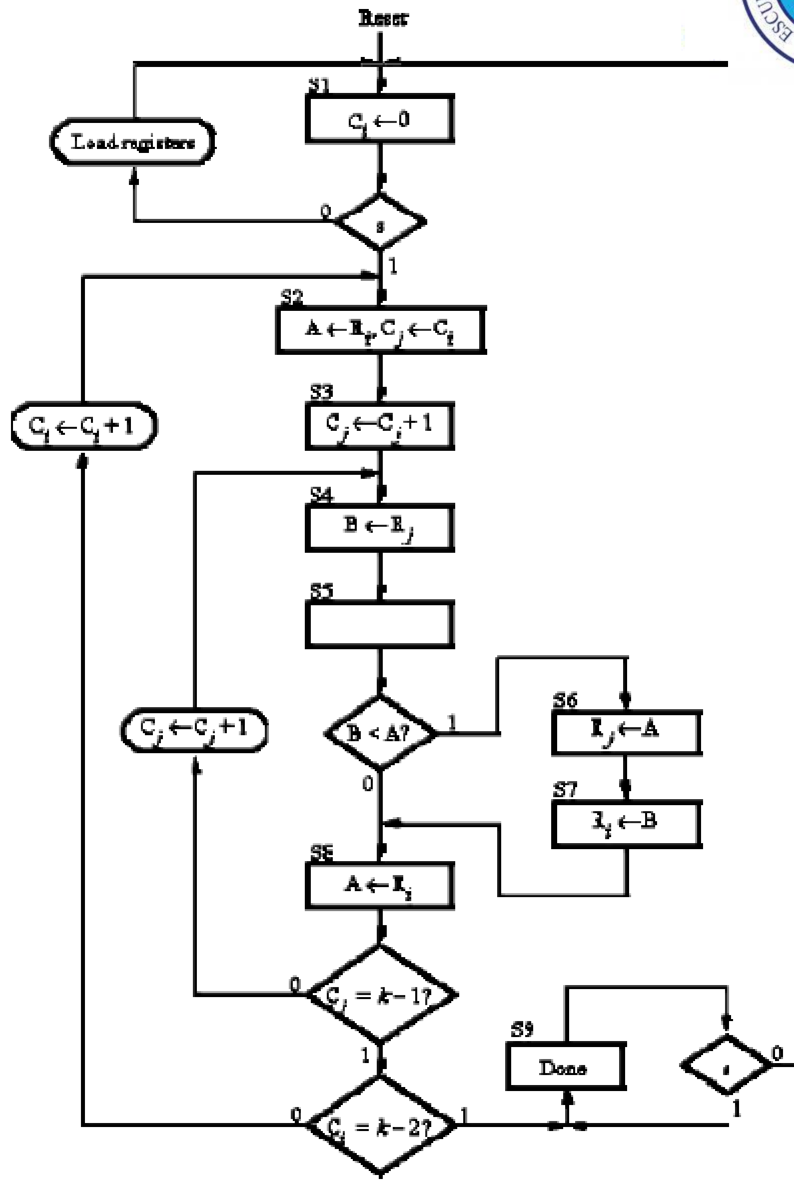
Al terminar de comparar un registro se toma el siguiente, hasta que todos se hayan comparado entre sí .



Algoritmo de control:

Podemos describir el algoritmo de control del circuito de la siguiente manera:

En el estado inicial **S1**, mientras **Start (s)= 0**, los registros **(R0,…, Rk-1)** deben ser cargados desde la entrada externa. Un contador **Ci** que representa el número del registro **i** se pone en cero. Al activar la entrada **Start** el circuito pasa al estado **S2**. En **S2** el registro **A** es cargado con el contenido del registro **Ri** (inicialmente **R0*).*** También, un contador **Cj** que representa el número del registro **j** se carga con el mismo valor de **i**. Los contadores **Ci** y **Cj** se utilizan para habilitar los registros de los cuales se desea sacar la información para compararla e intercambiarla de ser necesario.



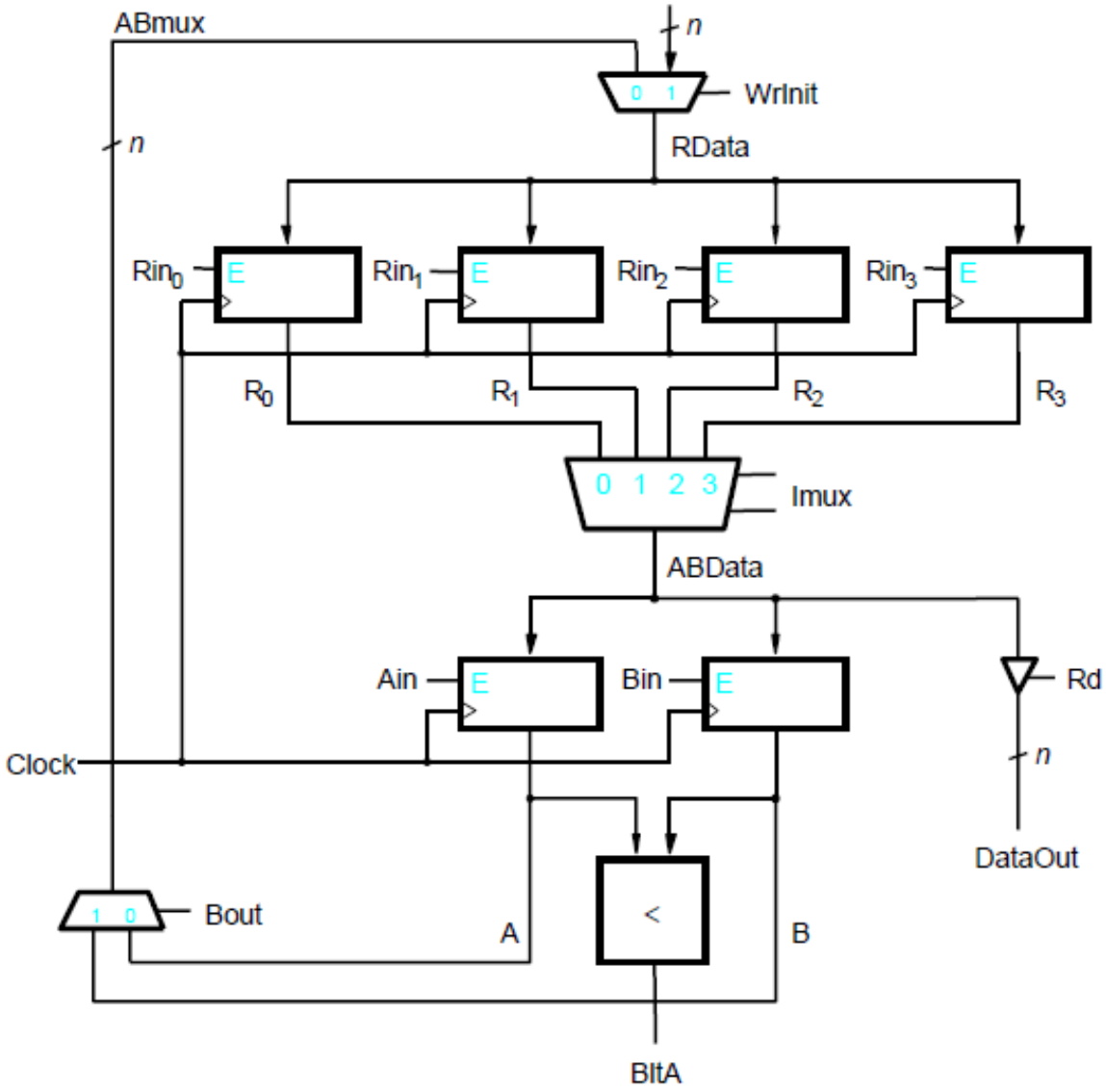
El estado **S3** se utiliza para inicializar el contador **Cj** con el valor **i+1**.
En el estado **S4** se carga el valor del registro **Rj** (donde el valor j es el del contador **Cj**) en el registro **B**. En el estado **S5**, **A** y **B** son comparados, y si **B < A**, el circuito pasa al estado **S6**.

Los estados **S6** y **S7** se utilizan para intercambiar los contenidos de los registros **Ri** y **Rj** solo si B fue menor que A.
En el estado **S8,** el contenido de **Ri** se carga en **A**. Aunque este paso es necesario solo cuando **B < A**, el flujo de control es más simple si esta operación se realiza en ambos casos, tanto cuando **B < A,** como cuando **A < B** (no afecta en nada a esta otra condición).

Si **Cj ≠ k-1** (que representa el ultimo registro), el circuito regresa de **S8** a **S4** y repite este lazo interno hasta que **Cj = k-1**.

Si **Cj = k-1** y **Ci ≠ k-2** (que representa el penúltimo registro), entonces el circuito habilita el siguiente registro incrementando el contador **Ci** y repite el lazo externo regresando al estado **S2**.

Procesador de Datos.



Hay varias maneras de implementar el **Procesador de Datos**.
Para simplificar el diseño asumimos que **k = 4**. Por lo tanto necesitamos 4 registros. Inicialmente estos registros pueden ser cargados con datos usando la entrada **DataIn**,activando la entrada de control externa **WrInit** e indicando la dirección del registro al cual se desea cargar el dato por medio de entrada externa **RAdd**.

Los datos desde las salidas de estos registros pueden ser cargados en registros **A** y **B** usando bloques **Mux 4 a 1** (salida **ABData**).
Las salidas de **A** y **B** se conectan a un comparador y, también, a través de un bloque de **Mux 2 a 1** pueden de nuevo ser cargados a los registros.
Los señales de habilitación de los registros **Rin0**,…, **Rink-1** son controlados por medio de un **Decodificador de 2 a 4**. Si la entrada de control **Int =1**, el decodificador es manejado por uno de los contadores **Ci** o **Cj**. Si **Int=0**, entonces el decodificador es controlado por la entrada externa **RAdd**.
Los señales **zi** y **zj** se hacen igual a **1** cuando **Ci=k-2** y **Cj=k-1**, respectivamente.
El buffer de tres-estados controlado por la señal externa de control **Rd,** pasa el contenido de los registros en la salida **DataOut**.

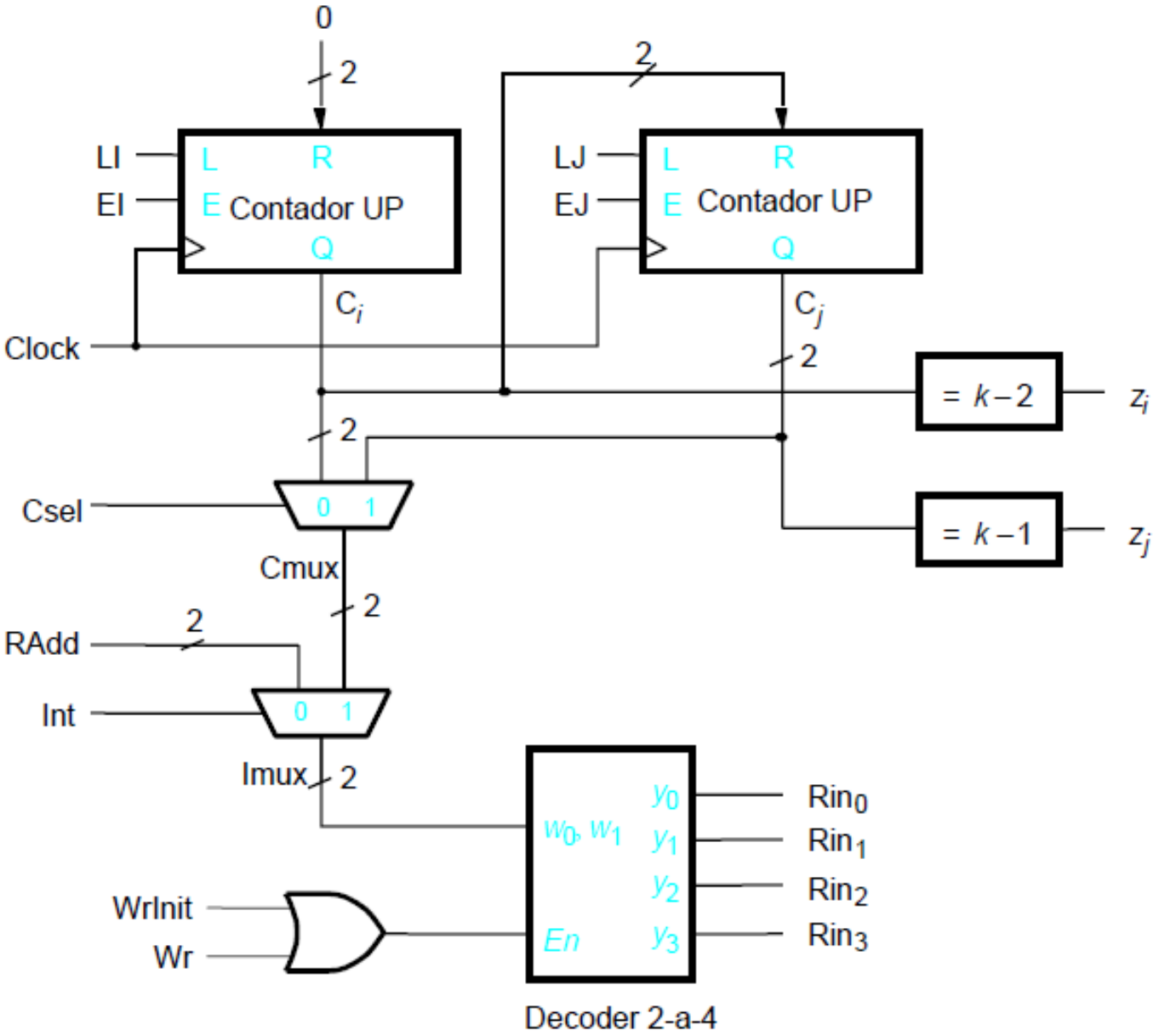
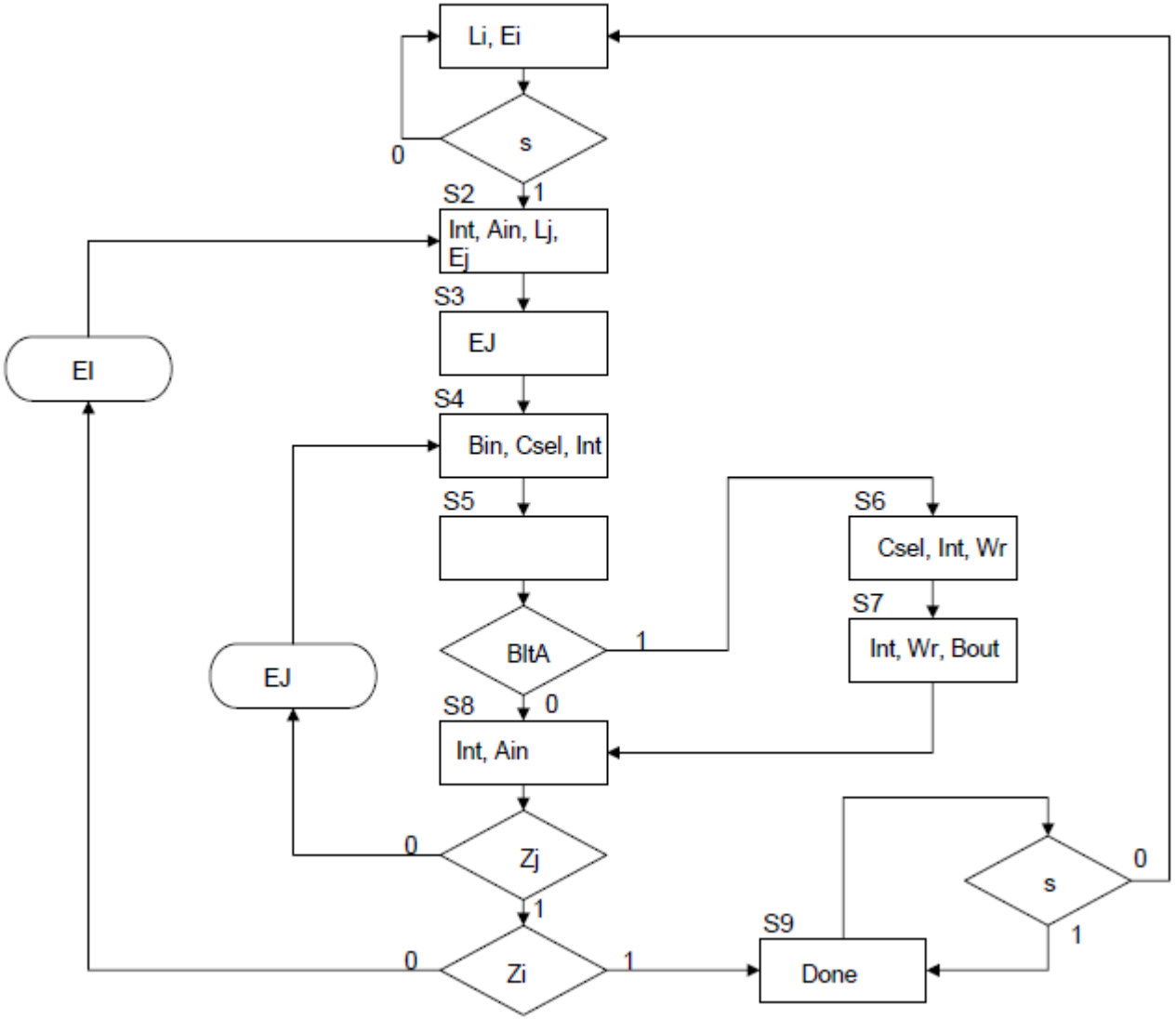

0
2
LI
EI
L
R
E Contador UP
Q
2
LJ
EJ
L
R
E Contador UP
Q
Ci
Cj
Clock
2
= k – 2
zi
2
Csel
0 1
= k – 1
zj
Cmux
2
RAdd
2
Int
0 1
Imux
2
w0, w1
y0
y1
y2
y3
En
Rin0
Rin1
Rin2
Rin3
WrInit
Wr
Decoder 2-a-4

Diagrama ASM del Controlador.



En el **Diagrama ASM** no se indica la operación que se debe hacer en cada estado, si señal especifica que debe generar en este estado el circuito **Controlador**.

En el estado S1 enceramos el contador Ci (Ei y Li), mientras que s=0. Si s=1 se pasa al estado S2 donde se carga el contador Cj (Ej y Lj) con el valor actual de Ci, se habilita el registro A (Ain) y para la dirección de los registros se escoge las salidas del contador Ci (Int=1 y Csel=0), de esta manera el dato Ri se almacena en A.

En el estado S3 se habilita el contador Cj para que se incremente y así se lo pueda comparar con el dato anterior. En el estado S4, se habilita el registro B (Bin) y como dirección se escoge el valor de Cj (Int=1 y Csel=1), de esta manera el dato Rj se almacena en B.

En S5 se pregunta si B es menor que A. De ser así se pasa a S6 y S7 donde se intercambia el dato entre los registros A y B. Para esto el dato que está en A se graba en el registro Rj (Csel=1, Int=1, Bout=0, Wr=1) y luego el dato en B se graba en Ri (Csel=0, Int=1, Bout=1, Wr=1).
En S8 el dato del registro Ri se graba en A (Csel=0, Int=1, Ain=1). Aquí también se pregunta si el número en Cj=k-1 (último registro). Si es falso, el contador Cj se incrementa. Si es verdadero se pregunta si el número en Ci=k-2 (penúltimo registro). Si no es así, Ci se incrementa. Si es cierto el proceso de finalización en S9 arranca.

En este diseño utilizamos para el **Procesador de Datos,** los componentes que están empaquetados y guardados en la librería **work**.

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

--circuito controlador del ordenador de numeros
entity ordenador_de_numeros is
    generic (n: integer := 4);
    port    (   Clock, Resetn         : in std_logic;
                Start, WrInit, Rd     : in std_logic;
                DataIn                : in std_logic_vector (n-1 downto 0);
                RAdd                  : in integer range 0 to 3;
                DataOut               : buffer std_logic_vector (n-1 downto 0);
                Done                  : buffer std_logic);
end ordenador_de_numeros;

architecture comportamiento of ordenador_de_numeros is
    type estado is (S1, S2, S3, S4, S5, S6, S7, S8, S9);
    signal y: estado;
    signal Ci, Cj: integer range 0 to 3;
    signal Rin: std_logic_vector (3 downto 0);
    type RegArray is array (3 downto 0) of std_logic_vector (n-1 downto 0);
    signal R: RegArray;
    signal RData, ABMux: std_logic_vector (n-1 downto 0);
    signal Int, Csel, Wr, BltA: std_logic;
    signal CMux, IMux : integer range 0 to 3;
    signal Ain, Bin, Aout, Bout: std_logic;
    signal Li, Ei, Lj, Ej, zi, zj: std_logic;
    signal zero: integer range 3 downto 0; -- dato para cargar Ci=0
    signal A, B, ABData: std_logic_vector (n-1 downto 0);
```

VHDL posee una gran versatilidad para la definición de nuevos tipos de datos (enteros, enumerados, arreglos, etc).

type < nombre> ***is*** <descripción**>;**

Por ejemplo,
type semana ***is*** (L, M, X, J, V, S, D***);***
type memoria ***is*** array (0 ***to*** 31) ***of*** bit_vector (4 ***downto*** 0***);***

En vez de definir las señales de datos con registros separados (**R0**, … , **Rk-1),** se puede especificar al conjunto de registros como un arreglo ***(array***), en forma similar a las memorias RAM.

Este método permite referirse a cada uno de los cuatro registros como **R(i),** dentro de la declaración ***for generate*** que utilizaremos más tarde para instanciar cada registro.

El arreglo de registros es definido en dos pasos:
Primero se define un tipo ***(type***) que llamamos **RegArray**.
type RegArray ***is array*** **(3** ***downto*** **0**) ***of*** **std_logic_vector (n-1** ***downto*****0);**
Luego, la señal **R** es definida como tipo **RegArray**.

A continuación, se describen los cambios de estados del circuito **Controlador** usando la declaración ***process***.

```
begin
    -- Transiciones de la MSS
    MSS_Transiciones: process (Clock, Resetn)
    begin
        if Resetn ='0' then y <= S1;
        elsif (Clock'event and Clock ='1') then
            case y is
                when S1 => if Start ='0' then y <=S1; else y <= S2; end if;
                when S2 =>  y <= S3;
                when S3 =>  y <= S4;
                when S4 =>  y <= S5;
                when S5 =>  if  BltA ='1' then y <= S6; else y <= S8; end if;
                when S6 => y <= S7;
                when S7 => y <= S8;
                when S8 => if zj ='0' then y <= S4;
                    elsif zi ='0' then y <= S2; else y <= S9; end if;
                when S9 => if Start ='1' then y <= S9; else y <= S1; end if;
            end case;
        end if;
    end process;
```

Las salidas del **Controlador** se generan de manera un tanto diferente a lo hecho anteriormente.

La salida Int es igual a cero (**Int = 0)** solo en el estado inicial **S1,** con el fin de poder direccionar los registros externamente y cargarlos con los valores de **DataIn** o para ver los datos ordenados. Por lo tanto, la generamos utilizando la declaración concurrente condicional.


```
-- salidas generadas por la MSS
    Int <='0' when y = S1 else '1';
    Done <='1' when y = S9 else '0';
```

Para las demás salidas utilizamos declaración ***process*** y luego ***case***.

```
MSS_salidas: process (y, zi, zj)
begin
    Li <='0'; Ei <='0'; Lj <='0'; Ej <='0'; Csel <='0';
    Wr <='0'; Ain <='0'; Bin <='0'; Aout <= '0'; Bout <='0';
    case y is
        when S1 => Li <='1'; Ei <='1';
        when S2 => Ain <='1'; Lj <='1'; Ej <='1';
        when S3 => Ej <='1';
        when S4 => Bin <='1'; Csel <='1';
        when S5 => -- ninguna salida
        when S6 => Csel <='1'; Wr <='1';
        when S7 => Wr <='1'; Bout <='1';
        when S8 => Ain <='1';
            if zj ='0' then Ej <='1'; else Ej <='0';
                if zi ='0' then Ei <='1'; else Ei <='0'; end if;
            end if;
        when S9 => -- salida Done ya esta asignada
    end case;
end process;
```

Luego se debe describir cada uno de los componentes del **Procesador de Datos**.

0110000101110011011000010110111001111010011000011

```
-- Procesador de datos

    registros_datos: for i in 0 to 3 generate
    registros_datos: registro_sost generic map (n => n)
        port map (Resetn, Clock, RData, Rin(i), R(i));
    end generate;

    registro_A: registro_sost generic map (n => n)
        port map (Resetn, Clock, ABData, Ain, A);

    registro_B: registro_sost generic map (n => n)
        port map (Resetn, Clock, ABData, Bin, B);
```

La declaración ***for generate*** proporciona una manera muy conveniente de crear instanciones múltiples, típicamente declaraciones de descripción de componentes.
También se puede tener la declaración ***if generate,*** que se utiliza para selectivamente ejecutar un conjunto de declaraciones cuando las condiciones especificadas se hacen presentes.

```
BltA <='1' when B < A else '0';

ABMux <= A when Bout <='0' else B;

RData <= ABMux when WrInit ='0' else DataIn;
```

Con estas declaraciones condicionales instanciamos el comparador y los dos **Mux de 2-a-1**.

```
zero <= 0;
C_i: contador_up1 generic map (modulo => 4)
                  port map (Resetn, Clock, Ei, Li, zero, Ci);
C_j: contador_up1 generic map (modulo => 4)
                  port map (Resetn, Clock, Ej, Lj, Ci, Cj);
CMux <= Ci when Csel ='0' else Cj;
IMux <= CMux when Int ='1' else RAdd;
```

En el estado **S1** se debe cargar el contador **Ci** con **0**. Pero es ilegal asociar un valor constante **'0'** o **'1'** con una entrada ***(port***) del componente.

Por lo tanto, en la declaración ***architecture*** creamos la señal **zero:**
signal **zero: integer *range* 3 *downto* 0**.
Y asignamos el valor de **'0'** a esta señal.

También describimos los **Mux de 2-a-1 (Imux** y **Cmux**) que se usaron para seleccionar la dirección del registro a utilizar en cada paso.

Para el **Mux de 4-a-1** que selecciona el registro del dato a cargar en el registro **A** o en el registro **B,** utilizamos la declaración ***with – select* -when**.

```
with IMux select
    ABData <= R(0) when 0,
              R(1) when 1,
              R(2) when 2,
              R(3) when others;
```

El **Decodificador de 2-a-4** es instanciado con las declaraciones secuenciales ***process - if - case***.

```
Rin_decodificador: process (WrInit, Wr, IMux)
begin
    if (WrInit or Wr) ='1' then
        case IMux is
            when 0 => Rin <="0001";
            when 1 => Rin <="0010";
            when 2 => Rin <="0100";
            when others => Rin <="1000";
        end case;
    else Rin <="0000";
    end if;
end process;
```


0110100100110010100110000101101110

Solo faltan los circuitos para generar las salidas **zi** y **zj** que son las salidas de los comparadores.

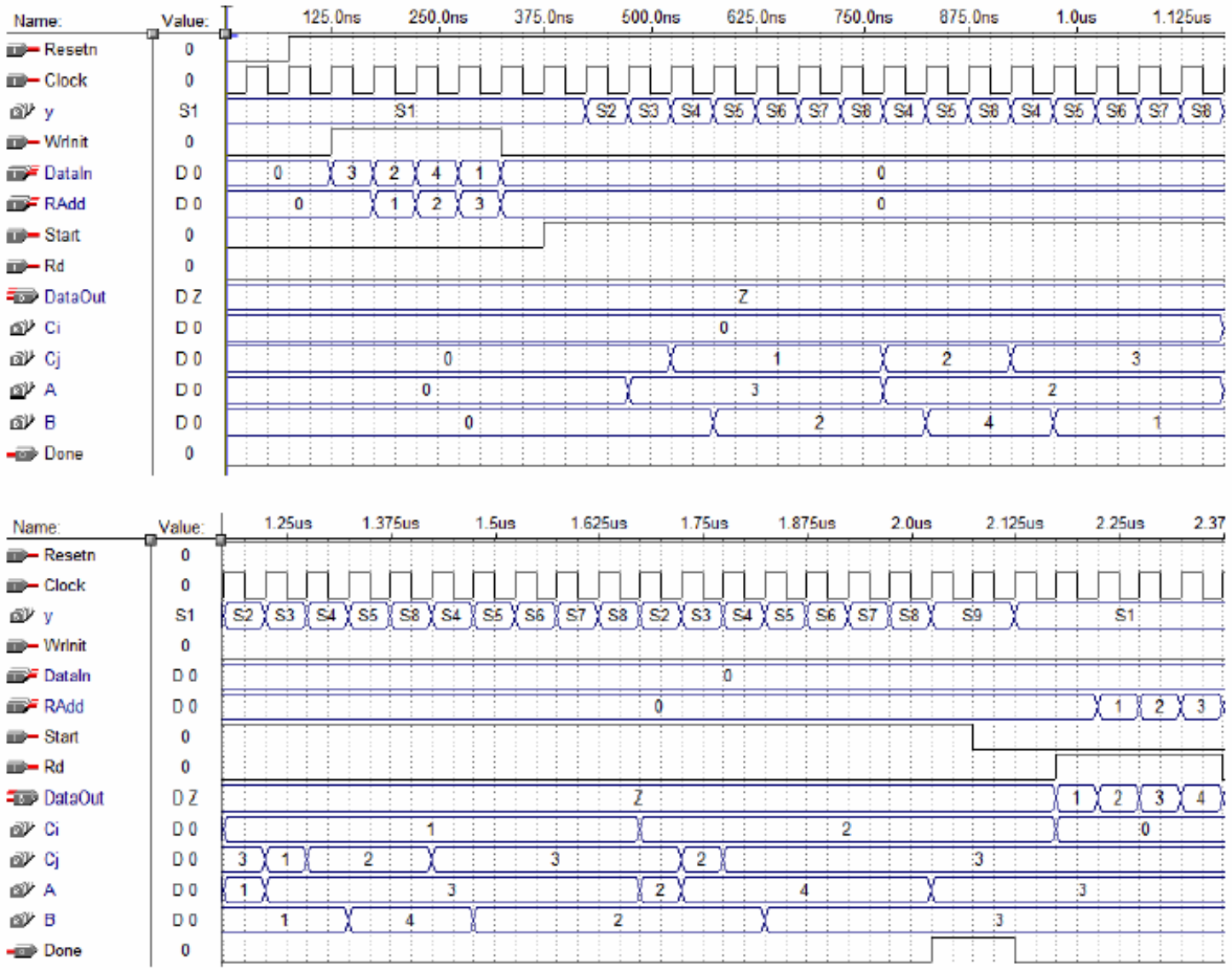
```
zi <='1' when Ci = 2 else '0';
zj <='1' when Cj = 3 else '0';
```

También falta describir los componentes “buffer” de tres estados.

```
        DataOut <= (others => 'Z') when Rd ='0' else ABData;
end comportamiento;
```

Tanto las señales **zi** y **zj** como la señal de salida **DataOut** se genera n utilizando declaraciones concurrentes condicionales.

Diagramas de Tiempo.



01100001011100110110000101101110011110100110000 1