Práctica III

Introducción a la descripción VHDL del procesador DLX

3.1. Implementación del procesador DLX

La configuración del procesador DLX que vamos a utilizar emplea una configuración Harvard, de tal forma que el programa y los datos son almacenados en memorias diferentes, de igual forma que lo haría un microcontrolador (que son más utilizados en sistemas empotrados que microprocesadores). También será un procesador de 32 bits por lo que tanto los datos como las instrucciones tendrán 32 bits de anchura.

El procesador DLX tendrá la interfaz de entrada/salida mostrada en la figura 3.1, dicha interfaz muestra la conexión con la memoria de datos y la memoria de programa. Las señales son las siguientes:

- Señales de reloj y de reset (clk y clr).
- Conexión con la memoria de datos: bus de lectura (data_in), de escritura (data_out), de direcciones (addr_Dmem) y habilitación de memoria (we_Dmem).
- Conexión con la memoria de programa: bus de lectura (instrucción) y de direcciones (addr_Pmem).

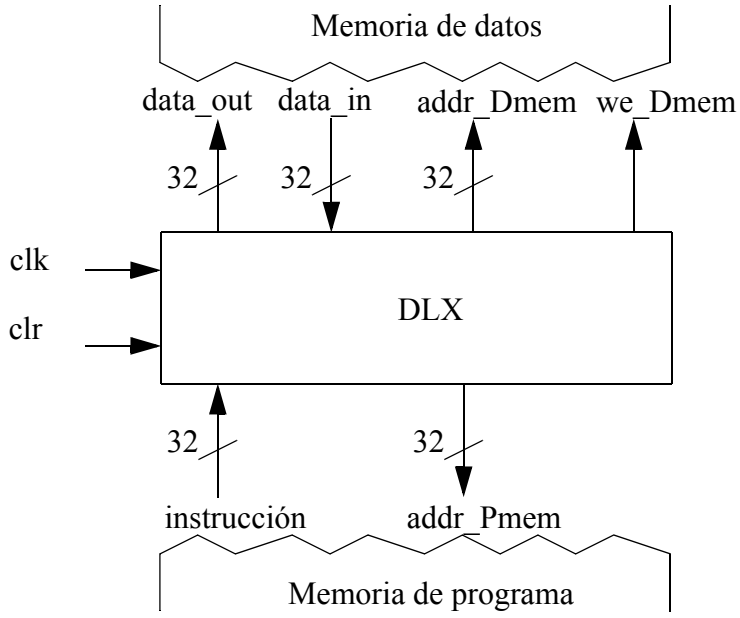

Figura 3.1.- Interfaz del procesador DLX con la memoria de datos y de programa.

El interior del procesador DLX está dividido en cinco etapas de pipeline, como se puede apreciar en la figura 3.2. En cada una de ellas se realizarán las operaciones de las tareas en el ciclo normal de una instrucción, es decir, búsqueda de la instrucción (identificado con el bloque IF), decodificación de la instrucción (identificado con el bloque ID), ejecución de la operación (identificado con el bloque EX), acceso a memoria y almacenamiento del resultado de la operación (identificado con el bloque WB). A continuación vamos a detallar cada una de dichas partes.

La primera etapa es la etapa de búsqueda de la instrucción o IF (Instruction Fetch). En dicha etapa estará el contador de programa (PC), que puede considerarse como primera etapa pipeline. En dicha fase se calcula la dirección de la siguiente operación que puede ser la siguiente posición de la memoria de programa (que va de cuatro en cuatro) o el valor obtenido por la ALU (en el caso de que se trate de instrucciones de salto); y se obtiene la instrucción a ejecutar cuya dirección está almacenada en el contador de programa.

La segunda etapa consiste en la decodificación de la instrucción o ID (Instruction Deodification). En dicha etapa se generarán las señales de control necesarias para la realización de las operaciones que se han de llevar a cabo con la instrucción en curso. Las señales de control se pueden dividir en dos grupos bien diferenciados:

- Señales de control de la ALU (o unidad de ejecución), que indicarán la operación que debe realizar la ALU con la instrucción en curso.
- Señales de selección y habilitación, que indicarán los caminos a seguir por las diferentes señales, así como la habilitación de los diferentes elementos de memoria (banco de registros y unidad de memoria de datos).

En esta etapa se encuentra declarado el banco de registros. Dicho banco es doble para poder disponer de dos puertos de lectura (puertos A y B). Para ello, ambos bancos son replicas (con el mismo dato de entrada y la misma dirección de escritura) con lecturas independientes. La escritura se realiza en el flanco de bajada (a diferencia de las etapas de pipeline) para no aumentar el número de ciclos de una operación.

La tercera etapa consiste en la ejecución de la operación indicada por la instrucción, realizada por el bloque EX (EXecution) o ALU. En dicha etapa se seleccionará las entradas a la ALU, para posteriormente realizar la operación indicada por las señales de control de la ALU.

La cuarta etapa consiste en un acceso a la unidad de memoria. Dicho acceso estará gobernado (habilitado) por el bloque ID, y en caso de habilitación, la dirección de memoria viene dado por el resultado de la ALU, y el dato viene dado por el contenido de uno de los puertos (B) del banco de registros. Al igual que sucede con los bancos de memoria, la escritura en memoria se lleva a cabo en el flanco de bajada para no aumentar el número de ciclos.

Por último, la quinta etapa consiste en el almacenamiento en el banco de registros del dato obtenido por la ALU (en el caso de una operación) o por la unidad de memoria (en el caso de un almacenamiento en registro).

A continuación vamos a ver a modo de ejemplo la secuencia temporal en la ejecución de una instrucción, en concreto de la instrución ori r1, r0, 5. Dicha secuencia se muestra en la figura 3.3. Esta secuencia será desglosada y comentada a conntinuación, según las diferentes etapas de pipeline existente en el procesador DLX.

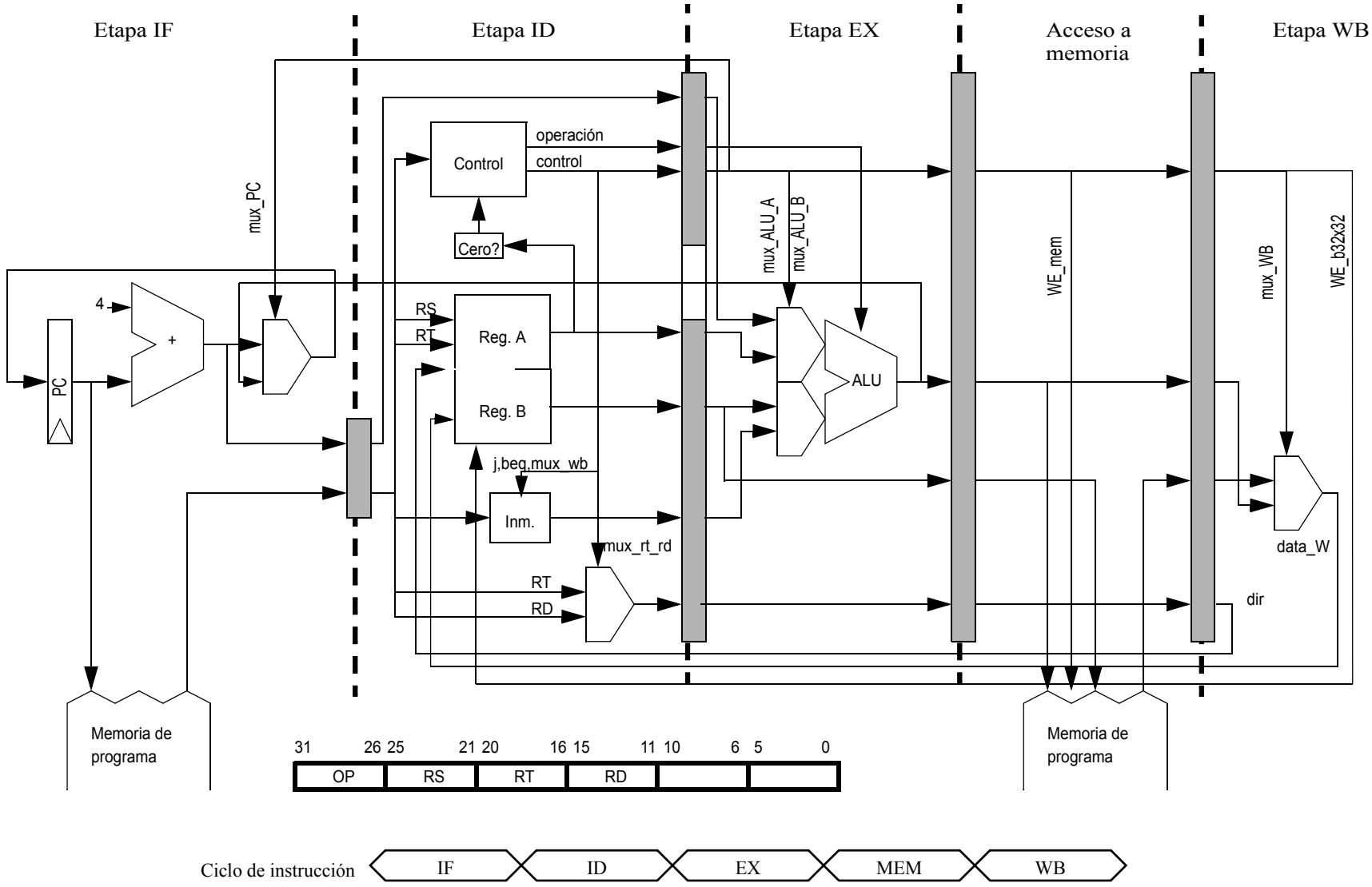

Figura 3.2.- Arquitectura del procesador DLX, junto a su formato de instrucciones genérico.

- 1ª ciclo de ejecución: Búsqueda de la instrucción
  En dicho ciclo, el bloque IF (Instruction Fetch) calcula la dirección de la siguiente instrucción, y lee de la memoria ROM la microinstrucción a ejecutar, es decir:
  00110100000000010000000000000101
  donde podemos observar las diferentes partes del formato de la instrucción:
  - Código de operación: 001101 (ori)
  - Dirección del registro fuente: 00000 (r0)
  - Dirección del registro destino: 00001 (r1)
  - Valor inmediato: 0000000000000101 (5)

- 2º ciclo de ejecución: Decodificar las operaciones a realizar
  En dicho ciclo, el bloque ID (Instruction Decodification) determina la operación que se han de realizar en función del código de operación de la instrucción, así como el establecimiento de las conexiones para poder llevar a cabo dicha operación. Como podemos apreciar de la figura 3.2, existen cuatro bloques bien diferenciados: una unidad de control, en la que se obtienen todas las señales de control necesarias; el banco de registros, el cual tendrá dos puertos (A y B) para su posible conexión como entradas a la ALU; el bloque Inm, que determinará el valor inmediato del formato de instrucciones cuando corresponda; y un multiplexor de direcciones para determinar la dirección del registro de almacenamiento en función del formato de instrucciones. El banco de registros será controlado por la bajada de la señal de reloj para no aumentar el número de ciclos de operación.
  Las operaciones que son necesarias controlar son las siguientes:
  - Se debe calcular el valor inmediato sin signo
  - No se debe almacenar en memoria
  - La entrada B de la ALU debe estar conectada al valor inmediato
  - La entrada A de la ALU debe estar conectada al registro A del banco de registros
  - La ALU estará decodificada para realizar la operación OR de sus entradas
  - El banco de registros debe estar habilitado
  - La entrada del contador de programa debe estar conectada a la dirección de la siguiente instrución
  - La dirección de almacenamiento del banco de registro se obtiene del segundo campo de dirección
  - El dato a almacenar en el banco de registros debe ser la salida de la ALU

- 3º Ciclo de ejecución: Cálculo de la operación
  En dicho ciclo, el bloque EX (Execution), o lo que es lo mismo la ALU, debe realizar las operaciones determinadas por la decodificación de la instrucción, es decir, la operación OR de las entradas.

- 4º ciclo de ejecución: Escritura en memoria
  Cuando se realice una operación de escritura en mmoria, la memoria debe estar habilitada (que no es nuestro caso como pudimos ver en el segundo ciclo. En dicho caso, la salida de la ALU sería la dirección de escritura, y el dato del banco de registros B sería el dato a escribir. Pero como ya hemos comentado, este no es el caso.

- 5º ciclo de ejecución: Almacenamiento en registro
  En este último caso, almacenaremos en el banco de registros el dato de salida, ya sea un dato de memoria o la salida de la ALU. En este caso particular sería la salida de la ALU.

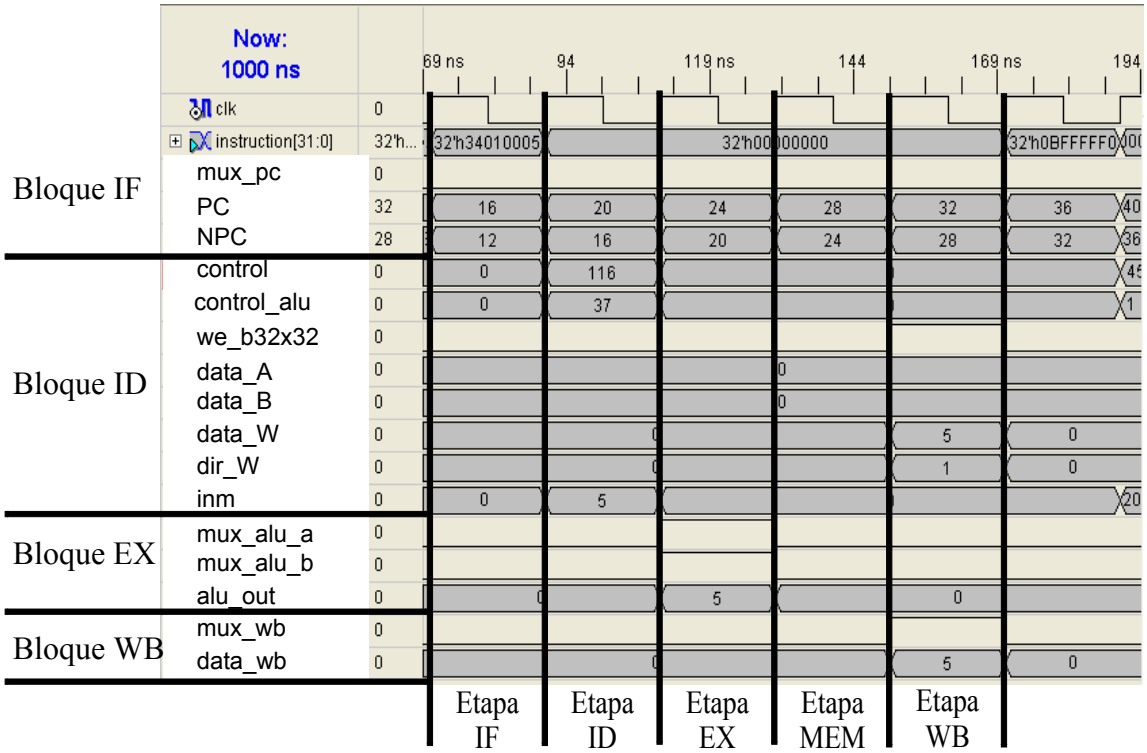

Figura 3.3.- Diferentes fases de la ejecución de la instrucción: ori r1, r0, 5

3.1.1. Implementación del procesador

Como ya hemos comentado, el procesador DLX se encuentra compuesto por cuatro etapas más la memoria de datos: bloque IF, bloque ID, bloque EX, y el bloque WB.

3.1.1.1. Bloque IF

El código VHDL del bloque IF se muestra a continuación.

```
--
-- IF.VHD
--
-- Etapa de busqueda de la instruccion del procesador DLX32p
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity etapa_IF is
port(
    CLK, RESET, INTR: in std_logic;
    EIF1: in std_logic; -- senales de control del multiplexor que alimenta el PC
    EIF6: in std_logic_vector(31 downto 0); -- dirección destino de una instruccion de salto
    SIF1, SIF2: out std_logic_vector(31 downto 0)  -- PC+4 y PC
   );
end etapa_IF;

architecture comportamental of etapa_IF is -- empieza la arquitectura de la etapa de busqueda

signal pc: std_logic_vector(31 downto 0):= (others => '0');
```

```
signal PCW: std_logic;

begin

    PCW <= '1' when (INTR = '0') else '0'; -- Se introduce una burbuja cuando INTR = '1'

    process (CLK,RESET) -- registro del PC
    begin
        if (RESET='0') then
            pc <= (others=>'0');
        elsif (CLK='1' and CLK'event) then
            if (PCW='1') then
                if (EIF1 ='0') then pc <= pc + 4; -- nuevo contador de programa PC
                else  pc <= EIF6; -- direccion destino de la instruccion de salto
                end if;
            end if;
        end if;
    end process;

    SIF1 <= pc + 4; -- PC+4
    SIF2 <= pc; -- PC

end;
```

En este código se lleva a cabo dos operaciones diferentes: la primera etapa de pipeline, consistente en el almacenamiento del contador de programa (el resto de etapas de pipeline es externa al resto de bloques), y el control de la búsqueda de la siguiente instrucción. Dicha búsqueda se realizará sumando al contador de programa cuatro posiciones (ya que una instrucción ocupa 32 bits, es decir, cuatro veces un byte) o bien una señal de entrada que se corresponderá con la dirección de un salto. Como salida, vamos a tener la dirección de la instrucción en curso, y la dirección de la nueva instrucción (PC+4).

El código dispone de un mecanismo de parada de ejecución de tal forma que cuando la señal INTR toma el valor '1', la etapa de pipeline no altera su valor manteniendo toda la información en un estado de espera. Se saldrá de dicho estado cuando la señal INTR vuelva a tomar el valor '0'. Esta situación se extiende al resto de etapas de pipeline, lo cual no aparecerá en el resto del código mostrado porque no se muestran los siguientes registros de pipeline.

3.1.1.2. Bloque ID

El código VHDL del bloque ID se muestra a continuación, incluido todos los módulos utilizados.

```
--
-- IDp.VHD
--
-- Etapa ID de DLX32p: decodificacion y lectura de operandos
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use work.dlx_pack.all;

entity idp is
port(
    clr, clock: in std_logic;
    NextPC: in std_logic_vector(31 downto 0); -- PC+4
    instruccion, Data_W: in std_logic_vector(31 downto 0); -- codigo instruccion en PC, y dato a
escribir en banco de registros
    ALU_control: out std_logic_vector(5 downto 0); -- palabra de control de la ALU
    b_dir_W: in std_logic_vector(4 downto 0); -- direccion del registro de escritura del banco
```

```
    WE_b32x32_ext: in std_logic; -- habilitacion de la operacion de escritura en el banco de registros
    control_word, Data_A, Data_B, inmediato, NextPC_out: out std_logic_vector(31 downto 0)  -- Salida: control, datos de los puertos A y B del banco de registros , inmediato, PC+4
   );
end idp;

architecture comportamental of idp is -- empieza la arquitectura

component control -- unidad de control
port(
    clr: in std_logic;
    COP: in TypeDlxOpcode; -- codigo operacion de la instruccion de 6 bits
    func: in TypeDlxFunc; -- campo "func" de la instruccion de 6 bits
    cond_Z: in std_logic; -- cuando (Z=1) el contenido del registro es cero
    MUX_rt_rd, MUX_PC, WE_b32x32, MUX_ALU_A, MUX_ALU_B, WE_mem, MUX_wb, beq, j: out std_logic; -- senyales de control
    ALU_op: out std_logic_vector(5 downto 0));
end component;

component b32x32  -- banco de 32 registros de 32 bits
port (  DAT_P1W_up : in STD_LOGIC_VECTOR (31 downto 0); -- dato de escritura en el banco de registros
    DIR_P1W_up, DIR_P2R_up, DIR_P3R_up: in STD_LOGIC_VECTOR (4 downto 0); -- P1: puerto de escritura; P2: primer puerto de lectura; P3: segundo puerto de lectura
    WE_up, CLK_up : in STD_LOGIC; -- senyales de sincronismo
    DAT_P2R_up, DAT_P3R_up : out STD_LOGIC_VECTOR (31 downto 0) ); -- datos de lectura de los puertos A y B
end component;

component inm -- calculo del inmediato
port(
    beq, j, mux_wb: in std_logic;
    NPC: in std_logic_vector(31 downto 0); -- 4 bits mas significativos del PC siguiente
    parte_instruccion: in std_logic_vector(25 downto 0);
    inmediato: out std_logic_vector(31 downto 0)  -- Salida inmediato y PA y PB
   );
end component;

signal mux_rt_rd, b32x32_we, control_j, control_beq, Z, control_mux_wb: std_logic:= '0';
signal DataA: std_logic_vector(31 downto 0) := (others => '0');
signal dir_W: std_logic_vector(4 downto 0) := (others => '0');

begin -- empieza la arquitectura

NextPC_out <= NextPC; -- el PC+4 entra y sale
Z <= '1' when (DataA = X"00000000") else '0'; -- Gneracion del flag Zero para las instrucciones de salto condicional

u1: control port map (
  clr => clr,
  COP => instruccion(31 downto 26), -- codigo de operacion de la instruccion
  func => instruccion(5 downto 0), -- codigo de operacion en la ALU
  cond_Z => Z, MUX_rt_rd => mux_rt_rd,
   MUX_PC => control_word(7), WE_b32x32 => b32x32_we, MUX_ALU_A => control_word(5), MUX_ALU_B => control_word(4),
  WE_mem => control_word(3), MUX_wb => control_mux_wb, beq => control_beq ,j => control_j,
  ALU_op => ALU_control -- control de la ALU
);

-- Generacion de la palabra de control, por ahora solo se usan 9 bits de un total de 32
control_word(31 downto 9) <= (others=>'0');
control_word(8) <= mux_rt_rd;
control_word(6) <= b32x32_we;
control_word(2) <= control_mux_wb;
control_word(1) <= control_beq;
control_word(0) <= control_j;

dir_W <= b_dir_W; -- direccion del puerto de escritura del banco de registros que viene desde la etapa WB a traves de la entity
```

```
u2: b32x32 port map (
  DAT_P1W_up => Data_W,  -- dato a escribir por el puerto de escritura
  DIR_P1W_up => dir_W, -- direccion del puerto escritura
  DIR_P2R_up => instruccion(25 downto 21), -- direccion del puerto de lectura rs/rs1
  DIR_P3R_up => instruccion(20 downto 16), -- direccion del puerto de lectura rt/rs2
  WE_up => WE_b32x32_ext, -- habilitacion escritura en banco registros, desde etapa WB
  CLK_up => clock,
  DAT_P2R_up => DataA, -- dato leido por el puerto A del banco registros
  DAT_P3R_up => Data_B -- dato leido por el puerto B del banco registros
);

Data_A <= DataA;

u3: inm port map (
  beq => control_beq,j => control_j, mux_wb => control_mux_wb, -- senyales de control
  NPC => NextPC, -- PC+4
  parte_instruccion => instruccion(25 downto 0), -- entrada del inmediato de 26 bits
  inmediato => inmediato -- Salida del inmediato de 32 bits
);

end;
```

El código IDP está formado por la conexión de tres nuevos bloques: el bloque de control, que generará las señales de control necesarias; el banco de registros; y el bloque de generación del valor inmediato. Dichos bloques se detallan a continuación.

El bloque de control, modelado en el fichero control.vhd, se muestra a continuación. Dicho bloque generará dos tipos de señales de control: señales de control de la ALU, las cuales determinarán la operación que debe realizar la ALU (bloque de ejecución), formado por la señal ALU_op; y señales de selección, que gobernarán los diferentes multiplexores para que el flujo de información vaya por el camino correcto, formado por el resto de señales de salida. Todas estas señales son obtenidas en función de dos campos del formato de instrucción: código de operación (COP), formado por los seis bits más significativos de la instrucción; y código de función (func), formado por los seis bits menos significativos de la instrucción (en el caso de que sea una operación directa con la ALU).

Por ejemplo, la instrucción

slti r3, r2, 273 -- con la microinstrucción ***011010***00010000110000000100010001

en la que se ha remarcado el código de operación (esta instrucción no es una instrucción directa de la ALU, y por lo tanto no tiene código de función), que se corresponde con cOpcode_slti. En este caso, se van a dar las siguientes señales:

- Mux_rt_rd = 0, tenemos una instrucción con un valor inmediato por lo tanto la dirección del registro destino se encuentra en el segundo campo de direcciones del formato de instrucciones
- Mux_PC = 0, no estamos ante una dirección de salto, por lo que la entrada del contador de programa debe ser igual a PC+4
- WE_b32x32 = 1, debemos almacenar en el banco de registros el resultado de la comparación, por lo que debemos habilitar la escritura en el banco de registro
- Mux_ALU_A = 1, la ALU tendrá que realizar la operación con un registro y un valor inmediato, por lo que la entrada A debe ser el puerto A del banco de registro
- Mux_ALU_B =1, para la operación con el inmediato debemos seleccionar el valor inmediato en la entrada B

- WE_mem = 0, no se debe realizar ninguna escritura en memoria, por lo que no se debe habilitar para escritura
- Mux_wb = 1, se debe almacenar la operación realizada por la ALU por lo que el multiplexor deberá seleccionar el canal 1
- BEQ = 0,
- J = 0

```
--
-- control.vhd
--
-- Unidad de control del procesador, la cual se encuentra en la etapa ID del procesador DLX32p
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use work.dlx_pack.all;

entity control  is
port(
    clr: in std_logic;
    COP: in TypeDlxOpcode; -- codigo operacion de la instruccion de 6 bits
    func: in TypeDlxFunc; -- campo "func" de 6 bits de la instruccion
    cond_Z: in std_logic; -- contenido registro cero (Z=1)
    MUX_rt_rd,MUX_PC,WE_b32x32,MUX_ALU_A,MUX_ALU_B,WE_mem,MUX_wb,beq,j:      out
std_logic;
    ALU_op: out std_logic_vector(5 downto 0)); -- palabra de control de la ALU
end control;

architecture estructural of control is

    signal c_out: std_logic_vector(8 downto 0);
    signal COP_int: TypeDlxOpcode;

begin

    COP_int <= cOpcode_undef_06 when (COP=cOpcode_alu and func=cAluFunc_nop)
        else COP; -- se quitan los "nop" de las del tipo R-R para que realmente no hagan nada

    with COP_int select
--c_out=MUX_rt_rd,MUX_PC,WE_b32x32,MUX_ALU_A,MUX_ALU_B,WE_mem,MUX_wb,beq,j;
        c_out <="101100100" when cOpcode_alu, -- operacion ALU
            "001110000" when cOpcode_lw, -- carga desde memoria de datos (lw)
            "000111000" when cOpcode_sw, -- almacenamiento en memoria de datos (sw)
            "001110100" when cOpcode_ori, -- or inmediato
            '0' & cond_Z & "0010010" when cOpcode_beqz,
                -- salto condicional si registro=0 (Z=1,beqz)
            "010010001" when cOpcode_j, -- salto incondicional (j)
            "010100000" when cOpcode_jr, -- salto incondicional con dirección en registro
            "001110100" when cOpcode_addi, -- suma con inmediato
            "001110100" when cOpcode_slti, -- comparación menor con inmediato
            "001110100" when cOpcode_slei, -- comparación menor o igual con inmediato
            "001110100" when cOpcode_seqi, -- comparación igual con inmediato
            "000000000" when others; -- aqui se incluyen los "nop"

    with COP_int select
        ALU_op <= func when cOpcode_alu, -- va directamente a la ALU
            cAluFunc_add when cOpcode_lw, -- sumar direccion e inmediato
            cAluFunc_add when cOpcode_sw, -- sumar direccion e inmediato
            cAluFunc_or when cOpcode_ori, -- or inmediato
            cAluFunc_undef_01 when cOpcode_beqz, -- ALU_out <= PC+4 + Inm
            cAluFunc_undef_01 when cOpcode_j, -- ALU_out<= PC+4 + Inm(25..0)
            cAluFunc_undef_02 when cOpcode_jr, -- ALU_out <= RS
            cAluFunc_add when cOpcode_addi, -- suma con inmediato
            cAluFunc_slt when cOpcode_slti, -- comparación menor con inmediato
            cAluFunc_sle when cOpcode_slei, -- comparación menor o igual con inmediato
            cAluFunc_seq when cOpcode_seqi, -- comparación igual con inmediato
```

```
            cAluFunc_nop when others;
    -- Salidas
    MUX_rt_rd <= c_out(8);
    MUX_PC <= c_out(7) and clr;
    WE_b32x32 <= c_out(6) and clr;
    MUX_ALU_A <= c_out(5) and clr;
    MUX_ALU_B <= c_out(4) and clr;
    WE_mem <= c_out(3) and clr;
    MUX_wb <= c_out(2) and clr;
    beq <= c_out(1) and clr;
    j <= c_out(0) and clr;
end;
```

A continuación mostramos el código referente al banco de registros del procesador DLX. Se ha optado por la solución de tener dos bancos de registros replicados para disponer de dos puertos de lectura. Debido a la réplica de los dos bancos, únicamente se dispone de un puerto de escritura que estará conectado a los dos bancos. Los registros tendrán una lectura asíncrona y una escritura síncrona.

```
--
-- b32x32.vhd
--
-- Banco de 3 puertos de 32 registros de 32 bits
--
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity b32x32 is
-- P1: puerto de escritura; P2: primer puerto de lectura; P3: segundo puerto de lectura
port (  DAT_P1W_up : in STD_LOGIC_VECTOR (31 downto 0);
    DIR_P1W_up, DIR_P2R_up, DIR_P3R_up: in STD_LOGIC_VECTOR (4 downto 0);
    WE_up, CLK_up : in STD_LOGIC;
    DAT_P2R_up, DAT_P3R_up : out STD_LOGIC_VECTOR (31 downto 0));
end b32x32;

architecture behav of b32x32 is

    component BANCOREG -- memoria de 1 puerto de lectura y 1 puerto de escritura
    -- P1: puerto de escritura; P2: puerto de lectura
    port (  DAT_P1W_int : in STD_LOGIC_VECTOR (31 downto 0);
        DIR_P1W_int,DIR_P2R_int: in STD_LOGIC_VECTOR (4 downto 0);
        WE_int,CLK_int : in STD_LOGIC;
        DAT_P2R_int,DAT_P1R_int : out STD_LOGIC_VECTOR (31 downto 0) );
    end component;

    signal NULO1,NULO2: STD_LOGIC_VECTOR(31 DOWNTO 0);
    signal WE_int: STD_LOGIC;

begin -- EL BANCO DE REGISTROS REALMENTE ESTA DUPLICADO: u1 y u2

    WE_int <= '0' when (DIR_P1W_up = "00000") else WE_up; -- se detecta la seleccion del
registro R0 para deshabilitar escrituras al R0

    u1:BANCOREG port map( -- primera replica
        DAT_P1W_int => DAT_P1W_up,
        DIR_P1W_int => DIR_P1W_up, DIR_P2R_int => DIR_P2R_up,
        WE_int => WE_int, CLK_int => CLK_up,
        DAT_P2R_int => DAT_P2R_up, DAT_P1R_int => NULO1
    );

    u2:BANCOREG port map( -- segunda replica
        DAT_P1W_int => DAT_P1W_up,
        DIR_P1W_int => DIR_P1W_up, DIR_P2R_int => DIR_P3R_up,
        WE_int => WE_int, CLK_int => CLK_up,
        DAT_P2R_int => DAT_P3R_up, DAT_P1R_int => NULO2
    );

end behav;
```

El banco de registros estará formado por elementos de memoria de 16 palabras de ocho bits. Por lo tanto, un banco estará compuesto por ocho de estos elementos, de los que un bloque de cuatro formará un almacenamiento de 16 palabras completas (de 32 bits).

```
--
-- BANCOREG.vhd
--
-- Banco de 32 registros de 32 bits con 1 puerto de escritura y 1 puerto de lectura
--
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity BANCOREG is
-- P1: puerto de escritura; P2: primer puerto de lectura
port (  DAT_P1W_int : in STD_LOGIC_VECTOR (31 downto 0);
    DIR_P1W_int,DIR_P2R_int: in STD_LOGIC_VECTOR (4 downto 0);
    WE_int,CLK_int : in STD_LOGIC;
    DAT_P2R_int,DAT_P1R_int : out STD_LOGIC_VECTOR (31 downto 0) );
end BANCOREG;

architecture behav of BANCOREG is

component mem16x8d
-- P1: puerto de escritura; P2: primer puerto de lectura
port ( DAT_P1W : in STD_LOGIC_VECTOR (7 downto 0);
    DIR_P1W,DIR_P2R: in STD_LOGIC_VECTOR (3 downto 0);
    WE,CLK : in STD_LOGIC;
    DAT_P2R,DAT_P1R : out STD_LOGIC_VECTOR (7 downto 0) );
end component;

signal DAT_P2Ri,DAT_P2R1o,DAT_P1Ri,DAT_P1R1o : STD_LOGIC_VECTOR (31 downto 0);
signal WE_0, WE_1: std_logic;

begin -- empieza la arquitectura

-- Se utilizan 8 bancos de de memoria de doble puerto de 16 datos x 8 bits/dato (RAM16x8D)
-- La palabra de 32 bits de salida se selecciona con DIR_P2R(4), bien de los 4 primeros bloques
(u1..u4) o de los segundos (u5..u8).
-- La escritura viene seleccionada por DIR_P1W(4)

WE_0 <= not(DIR_P1W_int(4)) and WE_int;
WE_1 <= DIR_P1W_int(4) and WE_int;
DAT_P2R_int <= DAT_P2R1o when (DIR_P2R_int(4)='1') else DAT_P2Ri ;
DAT_P1R_int <= DAT_P1R1o when (DIR_P1W_int(4)='1') else DAT_P1Ri ;

u1:mem16x8d
port map(
DAT_P1W=>DAT_P1W_int(7 downto 0),
DIR_P1W=>DIR_P1W_int(3 downto 0),
DIR_P2R=>DIR_P2R_int(3 downto 0),
WE=>WE_0, CLK=>CLK_int,
DAT_P2R=>DAT_P2Ri(7 downto 0),
DAT_P1R=>DAT_P1Ri(7 downto 0)
);

u2:mem16x8d
port map(
DAT_P1W=>DAT_P1W_int(15 downto 8),
DIR_P1W=>DIR_P1W_int(3 downto 0),
DIR_P2R=>DIR_P2R_int(3 downto 0),
WE=>WE_0, CLK=>CLK_int,
DAT_P2R=>DAT_P2Ri(15 downto 8),
DAT_P1R=>DAT_P1Ri(15 downto 8)
);

u3:mem16x8d
port map(
DAT_P1W=>DAT_P1W_int(23 downto 16),
DIR_P1W=>DIR_P1W_int(3 downto 0),
```

```
DIR_P2R=>DIR_P2R_int(3 downto 0),
WE=>WE_0, CLK=>CLK_int,
DAT_P2R=>DAT_P2Ri(23 downto 16),
DAT_P1R=>DAT_P1Ri(23 downto 16)
);

u4:mem16x8d
port map(
DAT_P1W=>DAT_P1W_int(31 downto 24),
DIR_P1W=>DIR_P1W_int(3 downto 0),
DIR_P2R=>DIR_P2R_int(3 downto 0),
WE=>WE_0, CLK=>CLK_int,
DAT_P2R=>DAT_P2Ri(31 downto 24),
DAT_P1R=>DAT_P1Ri(31 downto 24)
);

u5:mem16x8d
port map(
DAT_P1W=>DAT_P1W_int(7 downto 0),
DIR_P1W=>DIR_P1W_int(3 downto 0),
DIR_P2R=>DIR_P2R_int(3 downto 0),
WE=>WE_1, CLK=>CLK_int,
DAT_P2R=>DAT_P2R1o(7 downto 0),
DAT_P1R=>DAT_P1R1o(7 downto 0)
);

u6:mem16x8d
port map(
DAT_P1W=>DAT_P1W_int(15 downto 8),
DIR_P1W=>DIR_P1W_int(3 downto 0),
DIR_P2R=>DIR_P2R_int(3 downto 0),
WE=>WE_1, CLK=>CLK_int,
DAT_P2R=>DAT_P2R1o(15 downto 8),
DAT_P1R=>DAT_P1R1o(15 downto 8)
);

u7:mem16x8d
port map(
DAT_P1W=>DAT_P1W_int(23 downto 16),
DIR_P1W=>DIR_P1W_int(3 downto 0),
DIR_P2R=>DIR_P2R_int(3 downto 0),
WE=>WE_1, CLK=>CLK_int,
DAT_P2R=>DAT_P2R1o(23 downto 16),
DAT_P1R=>DAT_P1R1o(23 downto 16)
);

u8:mem16x8d
port map(
DAT_P1W=>DAT_P1W_int(31 downto 24),
DIR_P1W=>DIR_P1W_int(3 downto 0),
DIR_P2R=>DIR_P2R_int(3 downto 0),
WE=>WE_1, CLK=>CLK_int,
DAT_P2R=>DAT_P2R1o(31 downto 24),
DAT_P1R=>DAT_P1R1o(31 downto 24)
);

end behav;



--
-- mem16x8d.vhd
--
-- Banco de 16 registros de 8 bits
--
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
```

```
entity mem16x8d is
-- P1: puerto de escritura; P2: primer puerto de lectura
port ( DAT_P1W : in STD_LOGIC_VECTOR (7 downto 0);
    DIR_P1W,DIR_P2R: in STD_LOGIC_VECTOR (3 downto 0);
    WE,CLK : in STD_LOGIC;
    DAT_P2R,DAT_P1R : out STD_LOGIC_VECTOR (7 downto 0) );
end mem16x8d;

architecture behav of mem16x8d is

    type ram_type is array (0 to 15) of
        std_logic_vector(7 downto 0);
    signal tmp_ram: ram_type := (others => X"00");

begin

    -- Write Functional Section
    process(clk)
    begin
        if (clk'event and clk='1') then
            if we='1' then
                tmp_ram(conv_integer(dir_p1w)) <= dat_p1w;
            end if;
        end if;
    end process;
    dat_p1r <= tmp_ram(conv_integer(dir_p1w));
    dat_p2r <= tmp_ram(conv_integer(dir_p2r));

end behav;
```

El bloque inmediato generará el valor constante de las instrucciones del tipo R-I. En función de estas instrucciones, el valor inmediato debe ser sin signo, con signo. También podemos necesitar que el valor inmediato debe ser sumado a la dirección de la nueva instrucción, es decir, pc+4+inmediato.

```
--
-- INM.VHD
--
-- Etapa de generacion del valor inmediato de 32 bits a partir del codigo de instruccion para llevarlo
a la ALU del DLX32p
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
use work.dlx_pack.all;

entity inm is
port(
    beq,j,mux_wb:    in std_logic; -- indica que tipo de inmediato necesitan las distintas instrucciones
    NPC:     in std_logic_vector(31 downto 0); -- PC+4
    parte_instruccion: in std_logic_vector(25 downto 0); -- 26 bits menos significativos del codigo de
instruccion
    inmediato:    out std_logic_vector(31 downto 0)  -- Salida del valor inmediato de 32 bits
   );
end inm;

architecture comportamental of inm is

    signal extension1,extension2,extension3,extension4: std_logic_vector(31 downto 0);
    signal PCW: std_logic_vector(2 downto 0);
    signal extension,cero16: std_logic_vector(15 downto 0);
    signal ext: std_logic_vector(5 downto 0);

begin -- empieza la arquitectura

    PCW <= beq & j & mux_wb;
    extension <= (others=>parte_instruccion(15)); -- generamos parte CON extension signo 16 bits
    ext <= (others=>parte_instruccion(25)); -- extendemos el signo al inmediato 6 bits
```

```
    cero16 <= (others=>'0'); -- generamos parte SIN extension signo 16 bits

    process(parte_instruccion,NPC,extension,ext) -- genera la direccion de salto en la instruccion
"beqz" y "j"
        variable temp1,temp2 : signed(31 downto 0);
    begin
        --beqz: extension1 <= (PC + 4) + inmediato-16 bits-extendido en signo
        temp1 := MVL_TO_SIGNED(extension & parte_instruccion(15 downto 0)); -- convertimos
inm-32 bits a entero con signo
        temp2 := temp1 + MVL_TO_SIGNED(NPC); -- sumamos
        extension1 <= CONV_STD_LOGIC_VECTOR(temp2,32); -- convertimos a std_logic
        --j: extension2 <= (PC + 4) + inmediato-26 bits-extendido en signo
        temp1 := MVL_TO_SIGNED(ext & parte_instruccion); -- convertimos a entero con signo
        temp2 := temp1 + MVL_TO_SIGNED(NPC); -- sumamos
        extension2 <= CONV_STD_LOGIC_VECTOR(temp2,32); -- convertimos a std_logic
    end process;

    extension3 <= extension & parte_instruccion(15 downto 0); -- inmediato de 32 bits con
extension de signo para operaciones aritmeticas con inmediato
    extension4 <= cero16 & parte_instruccion(15 downto 0); -- inmediato de 32 bits sin extension de
signo para operaciones logicas con inmediato

    with PCW select -- seleccionamos finalmente el tipo de inmediato de salida
        inmediato <= extension1 when "100", -- beq
            extension2 when "010", -- j
            extension4 when "001", -- OR inmediato
            extension3 when others; -- demas instrucciones
end;
```

3.1.1.3. Bloque EX

El bloque EX es el encargado de llevar a cabo la ejecución de las operaciones relativas a la instrucción en curso. En dicho bloque seleccionaremos las entradas a la ALU, y la ALU misma. Las instrucciones que han sido implementadas en el procesador se pueden ver en este bloque así como en el bloque de control. En dichos bloques se hacen referencia a una serie de nombres que facilitan la comprensión de la instrucción a la cual hacen alusión; la combinación binaria que está enlazada con dichos nombres se encuentra en el paquete dlx_pack.vhd, al cual se referencia en el código.

```
--
-- EX.VHD
--
-- Etapa de ejecucion de la instruccion de DLX32P
--
library IEEE;
use IEEE.std_logic_1164.all;
use ieee.std_logic_arith.all;
use work.dlx_pack.all;

entity ex is
    port (
        MUX_A_control,MUX_B_control: in STD_LOGIC; -- bit de control de los multiplexores de
entrada a la ALU: MUX_ALU_A y MUX_ALU_B
        ALU_control: in STD_LOGIC_VECTOR (5 downto 0); -- palabra de control de la ALU
        NextPC,Data_A,Data_B,Inm: in STD_LOGIC_VECTOR (31 downto 0); -- NextPC=PC+4,
Data_A,Data_B= datos desde banco de registros, Inm= valor inmediato de 32 bits
        ALU_out: out STD_LOGIC_VECTOR (31 downto 0) -- ALUout= salida de la ALU
    );
end ex;

architecture ex_arch of ex is
begin -- empieza la arquitectura
```

```
    process(ALU_control,Data_A,Data_B,NextPC,Inm,MUX_A_control,MUX_B_control)

        variable index, v2: integer;
        variable MUX_A_out, MUX_B_out, v5, iALU_out: SIGNED(31 DOWNTO 0); -- v1, v3, v4
        variable COUNT,dummy1,dummy2: unsigned(31 downto 0);

    begin

        case MUX_A_control is -- multiplexor para la entrada A de la ALU
            when '1' => MUX_A_out := MVL_TO_SIGNED(Data_A); -- dato desde banco de registros
            when others => MUX_A_out := MVL_TO_SIGNED(NextPC); -- PC+4
        end case;

        case MUX_B_control is -- multiplexor para la entrada B de la ALU
            when '0' => MUX_B_out := MVL_TO_SIGNED(Data_B); -- dato desde banco de registros
            when others => MUX_B_out := MVL_TO_SIGNED(Inm); -- valor inmediato de 32 bits
        end case;

        case ALU_control is
            when cAluFunc_add => iALU_out := MUX_A_out + MUX_B_out; -- operacion de
instruccion ADD
            when cAluFunc_undef_01 => iALU_out := MUX_B_out; -- desde bloque inmediato sale la
direccion de salto INCONDICIONAL
            when cAluFunc_undef_02 => iALU_out := MUX_A_out; -- ALU = registro A
            when cAluFunc_sub => iALU_out := MUX_A_out - MUX_B_out; -- operacion de
instruccion SUB
            when cAluFunc_and => -- operacion de AND logica
                for index in 31 downto 0 loop
                    v5(index) := MUX_A_out(index) and MUX_B_out(index);
                end loop;
                iALU_out := v5;
            when cAluFunc_or => -- operacion de OR logica
                for index in 31 downto 0 loop
                    v5(index) := MUX_A_out(index) or MUX_B_out(index);
                end loop;
                iALU_out := v5;
            when cAluFunc_xor =>
                for index in 31 downto 0 loop
                    v5(index) := MUX_A_out(index) xor MUX_B_out(index);
                end loop;
                iALU_out := v5;
            when cAluFunc_sra => -- operacion de desplazamiento aritmetico a la derecha
                COUNT := CONV_UNSIGNED(MUX_B_out,32);
                iALU_out := SHR(MUX_A_out,COUNT); -- SHR funcion XILINX, desplazamiento
aritmetico, replica signo
            when cAluFunc_srl => -- operación de desplazamiento lógico a la derecha
                COUNT := CONV_UNSIGNED(MUX_B_out,32);
                dummy1 := CONV_UNSIGNED(MUX_A_out,32);
                dummy2 := SHR(dummy1,COUNT); -- SHR funcion XILINX
                iALU_out := CONV_SIGNED(dummy2,32);
            when cAluFunc_sll => -- operacion de desplazamiento logico a la izquierda
                COUNT := CONV_UNSIGNED(MUX_B_out,32);
                dummy1 := CONV_UNSIGNED(MUX_A_out,32);
                dummy2 := SHL(dummy1,COUNT); -- SHL funcion XILINX
                iALU_out := CONV_SIGNED(dummy2,32);
            when cAluFunc_slt => -- operacion de comparacion menor
                if (MUX_A_out < MUX_B_out) then
                    v5(0) := '1';
                else v5(0) := '0';
                end if;
                v5(31 downto 1) := (others => '0');
                iALU_out := v5;
            when cAluFunc_seq => -- operacion de comparacion igual
                if (MUX_A_out = MUX_B_out) then
                    v5(0) := '1';
                else v5(0) := '0';
                end if;
                v5(31 downto 1) := (others => '0');
                iALU_out := v5;
```

```
            when cAluFunc_sle => -- operacion de comparacion menor o igual
                if (MUX_A_out <= MUX_B_out) then
                    v5(0) := '1';
                else v5(0) := '0';
                end if;
                v5(31 downto 1) := (others => '0');
                iALU_out := v5;
            when others =>  iALU_out := CONV_SIGNED(0,32);
        end case;

        ALU_out <= CONV_STD_LOGIC_VECTOR(iALU_out,32); -- salida de la ALU

    end process;

end ex_arch;
```

3.1.1.4. Bloque WB

El bloque de retroescritura únicamente identifica el valor que se debe almacenar en el banco de registros: el dato proveniente de la ALU o de la memoria de datos.

```
--
-- WB.VHD
--
-- Etapa WB del procesador DLX32P
--
library IEEE;
use IEEE.std_logic_1164.all;

entity wb is
    port (
        data_mem: in STD_LOGIC_VECTOR (31 downto 0);
        data_ALU: in STD_LOGIC_VECTOR (31 downto 0);
        mux_wb:   in STD_LOGIC;
        data_WB:  out STD_LOGIC_VECTOR (31 downto 0)
    );
end wb;

architecture wb_arch of wb is
begin -- empieza la arquitectura
data_WB <= data_mem when (mux_wb='0') else data_ALU; -- multiplexor que selecciona el dato
de la ALU o de la memoria de datos
end wb_arch;
```

3.2. Conjunto de instrucciones implementado

El procesador que vamos a utilizar se trata de un procesador DLX simplificado en el sentido de que no contará con todas las instrucciones típicas de dicho procesador. Las instrucciones que se encuentran implementadas se muestran en la tabla 3.1.

A continuación vamos a dar algunos detalles de las instrucciones. En principio las operaciones se realizarán entre los valores almacenados en el banco de registros internos del procesador; no obstante también existen operaciones entre el valor de un registro y un valor fijo incluido en la instrucción (denominado valor inmediato), dichas instrucciones tienen la letra "I" como última letra en su denominación (por ejemplo, ADD será la suma entre dos registros y ADDI será la suma de un registro con un valor inmediato). Las siguientes intrucciones realizarán las siguientes operaciones:

- add rd, rs1, rs2 --> rd = rs1 + rs2

Instr.	Formato	Ejemplo	Instr.	Formato	Ejemplo
Instrucciones aritmético-lógicas					
ADD	add rd, rs1, rs2	add r4, r1, r2	ADDI	addi rd, rs1, valor	addi r4, r1, 20
SUB	sub rd, rs1, rs2	sub r4, r1, r2			
OR	or rd, rs1, rs2	or r4, r1, r2	ORI	ori rd, rs1, valor	ori r4, r3, 10
AND	and rd, rs1, rs2	and r4, r1, r2	XOR	xor rd, rs1, rs2	xor r4, r3, r2
SRA	sra rd, rs1, rs2	sra r4, r1, r2			
SRL	srl rd, rs1, rs2	srl r4, r1, r2	SLL	sll rd, rs1, rs2	sll r4, r2, r10
Instrucciones de comparación					
SLT	slt rd, rs1, rs2	slt r4, r1, r2	SLTI	slti rd, rs1, valor	slti r4, r2, 5
SEQ	seq rd, rs1, rs2	seq r4, r1, r2	SEQI	seqi rd, rs1, valor	seqi r4, r2, 10
SLE	sle rd, rs1, rs2	sle r4, r1, r2	SLEI	slei rd, rs1, valor	slei r4, r5, 40
Instrucciones de almacenamiento					
LW	lw rd, offset(rs)	lw r4, 3(r1)	SW	sw offset(rs1), rs2	sw -1(r3), r5
Instrucciones de salto					
J	j etiqueta	j inicio	JR	jr rs	jr r3
BEQZ	beqz rs, etiqueta	beqz r1, inicio			

Tabla 3.1. Conjunto de instrucciones del DLX modelado

- addi rd, rs1, valor --> rd = rs1 + valor
- sub rd, rs1, rs2 --> rd = rs1 - rs2
- or rd, rs1, rs2 --> rd = rs1 or rs2
- ori rd, rs1, valor --> rd = rs1 or valor
- and rd, rs1, rs2 --> rd = rs1 and rs2
- xor rd, rs1, rs2 --> rd = rs1 xor rs2
- sra rd, rs1, rs2 --> rd = $rs1(31)^{rs2}$ | rs1(31..rs2)
  Se mantiene el bit más significativo de rs1 (31) tantos bits como el valor almacenado en rs2, y el resto se mantiene con los bits más significativos de rs1 hasta llegar a los 32 bits. En el caso de que rs2 valga 10, y el valor de rs1 sea 10111111110000000000000000000000, entonces el valor de salida será 11111111111011111111000000000000
- srl rd, rs1, rs2 --> rd = 0^{rs2} | rs1(31..rs2)
  Igual que sra pero desplazando un ‘0’ en lugar del bit más significativo de rs1
- sll rd, rs1, rs2 --> rd = rs1(31-rs2..0) | 0^{rs2}

- slt rd, rs1, rs2 --> rd = “1” si rs1 < rs2, else rd = “0”
- slti rd, rs1, valor --> rd = “1” si rs1 < valor, else rd = “0”
- seq rd, rs1, rs2 --> rd = “1” si rs1 = rs2, else rd = “0”
- seqi rd, rs1, valor --> rd = “1” si rs1 = valor, else rd = “0”
- sle rd, rs1, rs2 --> rd = “1” si rs1 <= rs2, else rd = “0”
- slei rd, rs1, valor --> rd = “1” si rs1 <= valor, else rd = “0”
- lw rd, offset(rs1) --> rd = memoria(offset+rs1)
- sw offset(rs1), rs2 --> memoria(offset+rs1) = rs2
- j etiqueta --> PC = PC + 4 + (etiqueta(25)^{6} | etiqueta(25..0))
  Si PC es igual a 40 y etiqueta se corresponde con la posición -20, el valor final será 24. El último bit de la etiqueta es el signo por lo que en caso de valor ‘1’ (ser negativo), la suma se convierte en resta.
- jr rs1 --> PC = rs1
- beqz rs1, etiqueta --> si rs1 = 0 => PC = PC + 4 + (etiqueta(15)^{16}|etiqueta(15..0)), else PC = PC + 4

Todas estas instrucciones se encuentran identificada por cada uno de los formatos de instrucciones mostrados en la figura 3.4.

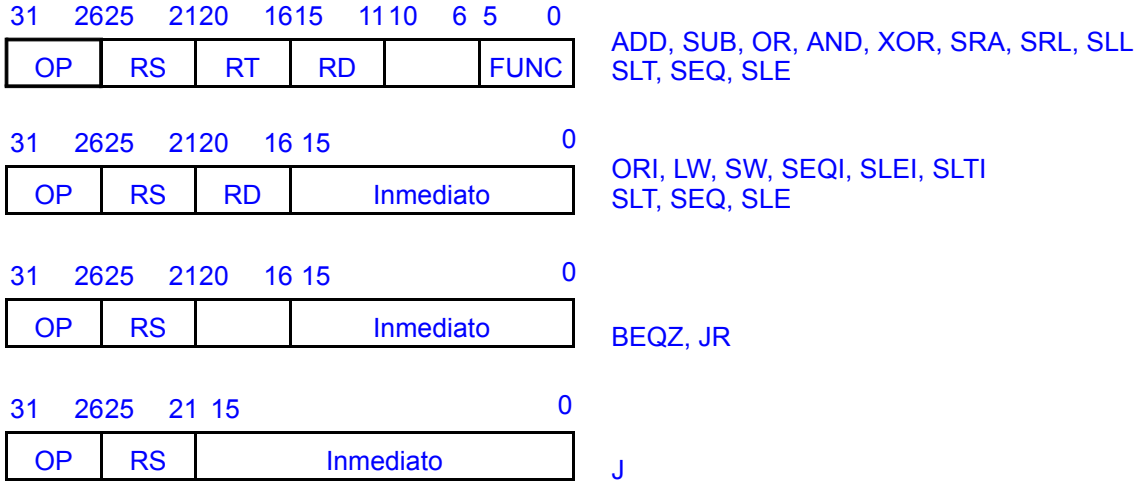

Figura 3.4.- Formato de instrucciones, junto a las instrucciones implementadas que lo utilizan.

Dicha diferencia de formato es la que conlleva que sean necesarios varios controles para indicar los diferentes campos de registro destino (por ejemplo) en las instrucciones.

3.3. Memoria de programa

El procesador DLX implementado va a sguir una arquitectura Harvard, por lo que diferenciaremos entre memoria de programa y memoria de datos. Cabe destacar que ambas memorias son de 32 bits, pero el procesador DLX puede acceder a nivel de byte (aunque en esta

versión dichas instrucciones no están implementadas); por lo tanto, dos palabras consecutivas se encuentran diferenciadas por cuatro combinaciones de dirección, es decir, si PC es la dirección de la instrucción en curso, la dirección de la siguiente instrucción será PC+4. Dicha apreciación también sirve para la memoria de datos. La memoria de programa es modelada en VHDL de la siguiente forma:

```
--
-- MEM_PROG.VHD
--
-- Memoria de programa que utiliza una estructura ROM
--
use work.DLX_pack.all;
use work.DLX_prog2.all; -- donde se encuentra el programa a ejecutar

entity ROM_32x8 is
  port(ADDR: in ROM_RANGE;
       DATA: out ROM_WORD);
end ROM_32x8;

architecture comportamental of ROM_32x8 is
begin
  DATA <= ROM(ADDR);  -- lectura desde la ROM
end comportamental;
```

Dicha memoria se corresponde con una memoria de lectura asíncrona, de tal forma que el bus de datos (la señal DATA) siempre está conectada al elemento de la matriz de información determinada por la señal de dirección. Dicha matriz es indicada en un nuevo paquete denominado dlx_prog2, y guardado en el fichero dlx_prog2.vhd. Un ejemplo de este programa se muestra a continuación.

```
--
-- DLX_PROG2.vhd
--
-- Programa para el DLX32p
-- Se ha insertado nop para evitar las dependencias. Esta es la diferencia con DLX_PROG.vhd
-- que es el que se ha utilizado para DLX32s monociclo.
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
use work.DLX_pack.all;

package DLX_prog2 is
  type ROM_TABLE is array (0 to 14) of ROM_WORD;

  constant ROM: ROM_TABLE := ROM_TABLE'(

      ROM_WORD'("00000000000000000000000000000000"),
      ROM_WORD'("00000000000000000000000000000000"),
      ROM_WORD'("00110100000000100000000000010100"),
      ROM_WORD'("00000000000000000000000000000000"),
      ROM_WORD'("00000000000000000000000000000000"),
      ROM_WORD'("10001100000000110000000011111111"),
      ROM_WORD'("00000000000000000000000000000000"),
      ROM_WORD'("00000000000000000000000000000000"),
      ROM_WORD'("00000000000000000000000000000000"),
      ROM_WORD'("00000000000000000000000000000000"),
      ROM_WORD'("00000000000000000000000000000000"),
      ROM_WORD'("00000000000000000000000000000000"),
      ROM_WORD'("00000000000000000000000000000000"),
      ROM_WORD'("00000000000000000000000000000000"),
      ROM_WORD'("00000000000000000000000000000000")
      );
```

```
end DLX_prog2;
```

En dicho modelo podemos apreciar la matriz de información como una matriz constante del número de palabras del programa por 32 (bits de palabra). Cada palabra de la matriz anterior se corresponde con una microinstrucción del programa.

No obstante, no debermos escribir el programa del procesador utilizando microinstrucciones, sino que utilizaremos un compilador en el que únicamente deberemos introducir el código en ensamblador.

3.3.1. Compilador de DLX

El compilador de DLX utilizado está desarrollado en Java, y necesita como mínimo la versión 1.5 para poder ser ejecutado. La ventana del compilador se muestra en la figura 3.5. En dicha aplicación podemos apreciar cuatro ventanas diferentes:

- *Código original*, en la cual se introducirá el código ensamblador del programa directamente, o donde se cargará el fichero de entrada.
- *Código original limpio*, que será el mismo que el código original pero con la eliminación de todos los comentarios.
- *Código optimizado*, en esta versión del compilador no existe ningún tipo de optimización por lo que representará de nuevo el código original limpio.
- *Código máquina*, en la cual se representarán las microinstrucciones relativas al código optimizado.

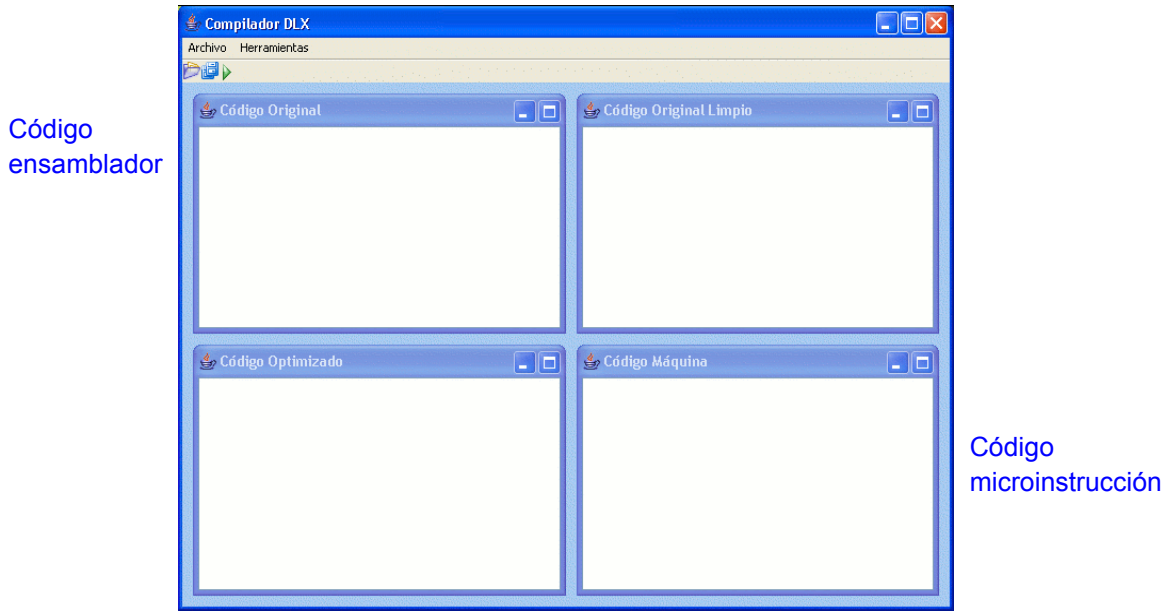

Figura 3.5.- Ventana del compilador de DLX.

A pesar de que el modelo del procesador DLX no implementa todas las instrucciones, todas ellas sí están incluidas en el compilador; por lo tanto hay que tener cuidado con evitar el uso de instrucciones no implementadas en el modelo VHDL, ya que el modelo las tratará como NOP.

En primer lugar debemos cargar el fichero de síntaxis, en el cual se encuentra la relación existente entre el nemotécnico y el código de microinstrucciones. Para ello, en el menú de *Herramientas* seleccionamos el submenú *Opciones*. En la ventana que se abre, indicamos el camino al fichero de sintaxis que está en el mismo directorio del compilador y es denominado *sintaxis.txt*.

Una vez que hemos escrito el código en ensamblador (o hemos abierto un fichero de texto con dicho código, que aparecerá en la ventana *Código original*), pulsamos el botón *Play* (compilar) y en la ventana del *Código máquina* aparecerán las microinstrucciones relativas al código original.

Después de tener visible las microinstrucciones, del menú Archivo seleccionamos el submenú *Guardar Código Máquina como*, para después seleccionar el fichero en el que queremos grabar el paquete del programa especificado en VHDL. Para que dicho paquete sea considerado por el modelo de DLX se debe almacenar con el nombre dlx_prog2.vhd, en el directorio del modelo del DLX y el fichero antiguo será sobreescrito. Esto último es la razón por la cual suele ser aconsejable guardarlo en otro lugar y/o con otro nombre para evitar su pérdida.

3.4. Ejemplo de simulación

A modo de ejemplo, vamos a considerar un caso de simulación. El entorno en el que se va incluir el procesador DLX no incluye ningún elemento de entrada-salida; por lo tanto, sólo estarán presentes el procesador DLX, la memoria de programas y la memoria de datos, como vimos en la figura 3.1.

El programa que vamos a ejecutar es el mostrado en la figura 3.6. En este programa podemos distinguir diferentes tipo de operaciones, las cuales vamos a describir paso a paso.

- Las no operaciones (NOP) son incluidas para que las operaciones de las diferentes instrucciones no se interfieran entre sí, es decir, su misión es la de retrasar las operaciones de determinadas instrucciones. Por ello, no las consideraremos en el resto de la discusión.
- Las primeras instrucciones ORI (operación or con un inmediato) se utilizan para cargar en un registro (destino) el valor inmediato incluido en la instrucción ya que el registro de entrada tiene almacenado un “0”. Para forzar el almacenamiento se suele utilizar como registro fuente el registro R0, que en el DLX siempre tiene almacenado un “0”.
  Por lo que cargaremos en los registros R1 y R2 los valores “2” y “3” respectivamente.
- Seguidamente sumaremos (instrucción ADD) los contenidos de dos registros, cuyo resultado se almacenará en otro registro.
  En este caso se sumará el contenido de los registros R1 y R2 y lo almacena en R3; por lo tanto, R3 tendrá el valor “5” almacenado.
- La siguiente instrucción es SW, es decir, un almacenamiento en memoria del contenido de un registro fuente. La dirección de memoria en la que se almacenará será la suma del contenido de un registro y un desplazamiento. Por lo tanto, si queremos identificar la dirección de memoria por el offset, el registro fuente debe tener almacenado un “0”, como puede ser le R0.
  Luego almacenaremos la posición de memoria utilizada ser de 4 (2 + 2, de R1) byte, y

```
--
-- DLX_PROG2.vhd
--
-- Programa para el DLX32p
-- Se ha insertado nop para evitar las dependencias. Esta es
la diferencia con DLX_PROG.vhd
-- que es el que se ha utilizado para DLX32s monociclo.
--                                                          Código ensamblador
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
use work.DLX_pack.all;

package DLX_prog2 is
  type ROM_TABLE is array (0 to 29) of ROM_WORD;

  constant ROM: ROM_TABLE := ROM_TABLE'(

     ROM_WORD'("00000000000000000000000000000000"),      nop
     ROM_WORD'("00000000000000000000000000000000"),      nop
     ROM_WORD'("00110100010000010000000000000010"),      ori r1, r2, 2
     ROM_WORD'("00110100011000100000000000000011"),      ori r2, r3, 3
     ROM_WORD'("00000000000000000000000000000000"),      nop
     ROM_WORD'("00000000000000000000000000000000"),      nop
     ROM_WORD'("00000000000000000000000000000000"),      nop
     ROM_WORD'("00000000010001000011000000100000"),      add r3, r1, r2
     ROM_WORD'("00000000000000000000000000000000"),      nop
     ROM_WORD'("00000000000000000000000000000000"),      nop
     ROM_WORD'("00000000000000000000000000000000"),      nop
     ROM_WORD'("10101100001000110000000000000010"),      sw 2(r1), r3
     ROM_WORD'("00000000000000000000000000000000"),      nop
     ROM_WORD'("00000000000000000000000000000000"),      nop
     ROM_WORD'("00000000000000000000000000000000"),      nop
     ROM_WORD'("10001100001010000000000000000010"),      salto:lw r8, 2(r1)
     ROM_WORD'("00000000110001001001000000100010"),      sub r9, r3, r2
     ROM_WORD'("00000000010001001010000000100100"),      sigue:and r10, r1, r2
     ROM_WORD'("00000000010001001011000000100101"),      or r11, r1, r2
     ROM_WORD'("00000000000000000000000000000000"),      nop
     ROM_WORD'("00000000000000000000000000000000"),      nop
     ROM_WORD'("00000001011000010110000000000111"),      sra r12, r11, r1
     ROM_WORD'("00000000000000000000000000000000"),      nop
     ROM_WORD'("00000000000000000000000000000000"),      nop
     ROM_WORD'("00010001100000001111111111011000"),      beqz r12, salto
     ROM_WORD'("00000000000000000000000000000000"),      nop
     ROM_WORD'("00000000000000000000000000000000"),      nop
     ROM_WORD'("00001011111111111111111111010100"),      j sigue
     ROM_WORD'("00000000000000000000000000000000"),      nop
     ROM_WORD'("00000000000000000000000000000000")       nop
     );

end DLX_prog2;
```

Figura 3.6.- Fichero del programa almacenado en la memoria de programas en el ejemplo.

como la palabra son 4 byte, corresponderá a la primera palabra. El valor que se debe almacenar será el contenido del registro R3, es decir, "5".

- Después, almacenaremos en un registro el contenido de una posición de memoria, instrucción LW.
  Se almacenará en el registro R8 el contenido de la posición de memoria anterior.
- Después se realizará la diferencia entre los valores de dos registros (instrucción SUB), la cual es almacenado en un registro.
  Luego, en el registro R9 se almacenará la diferencia entre los valores de los registros

R3 y R2, es decir, “2”.

- Seguidamente se llevará a cabo la operación AND bit a bit de dos registros (instrucción AND).
  Por lo tanto, el valor del registro destino, R10, será la operación AND de R1 y R2, es decir, el valor “2”.
- La siguiente operación a realizar es OR, por lo que se llevará a cabo la operación OR bit a bit del contenido de dos registros.
  Así, el valor del registro R11 será la operación OR de los registros R1 y R2, es decir, el valor “3”.
- Después se ejecutará la instrucción SRA, por lo que se llevará a cabo el desplazamiento aritmético hacia la derecha. Dicha instrucción añade tantas veces como indique el valor del segundo registro fuente el bit más significativo del primer registro fuente, es decir, el bit de signo; y los bits menos significativos del primer registro fuente desaparecen.
  Luego, en el registro R12 se almacenaría “00000000000000000000000000000000”, ya que los dos (valor de R1) bits menos significativos del registro R3, es decir, “11”, desaparecen y se añaden como bits más significativos dos veces (valor de R1) el bit más significativo de R3, es decir, ‘0’.
- Seguidamente se realizará un salto condicional con la instrucción BEQZ. Dicho salto se llevará a cabo siempre y cuando el registro fuente tenga almacenado un “0”. El salto se realizará a la dirección determinada por la etiqueta.
  Por lo tanto, se realizará el salto si el contenido del R12 es “0”
- Si no se ha realizado el salto anterior, con la siguiente instrucción J se llevará a cabo un salto incondicional hacia la dirección relativa a la etiqueta.

Todo este comportamiento se muestra en la figura 3.7. En el caso de la primera instrucción ORI, se requieren cinco ciclos de ejecución para completarla: en el primer ciclo se obtendrán los datos de entrada, es decir, la lectura del registro fuente y el cálculo del valor inmediato; en el segundo ciclo, la ALU ejecutará la operación OR; el tercer ciclo será un ciclo de espera ya que se destina al acceso a memoria de datos, la cual no es necesaria en la instrucción en curso; el cuarto ciclo será destinado a la preparación de los datos que se han de almacenar en el registro destino, cuyo almacenamiento será llevado a cabo en la bajada del reloj. En el caso de almacenamiento en memoria, instrucción SW, se utilziarán tres ciclos de operación para completarla: en el primer ciclo, se obtendrá el valor inmediato correspondiente al desplazamiento de la dirección de memoria, el valor del registro en el que se encuentra la segunda componente de la dirección de memoria, y el valor del registros que vamos a almacenar en memoria; en el segundo ciclo, se calculará la dirección de memoria como la suma del desplazamiento y el valor del registro correspondiente; y en el tercer ciclo se almacenará en la memoria de dato en la bajada de la señal de reloj. En el caso de la instrucción de salto, como la instrucción BEQZ, se necesitarán dos ciclos para su ejecución: en el primer ciclo se realizará el chequeo de si el registro fuente toma el valor “0”, se obtendrá el valor inmediato de la posición de la siguiente instrucción en caso de que se produzca el salto; en el segundo ciclo, se calculará la posición de memoria como NPC(=PC+4)+valor, la cual pasará a la etapa de búsqueda de instrucción.

Por lo tanto, según lo anterior podemos afirmar que cualquier instrucción instrucción que requiera un almacenamiento en registro necesitará cuatro ciclos de operación para completarla, un almacenamiento en memoria requiere de tres ciclos y un salto requiere de dos ciclos. Luego,

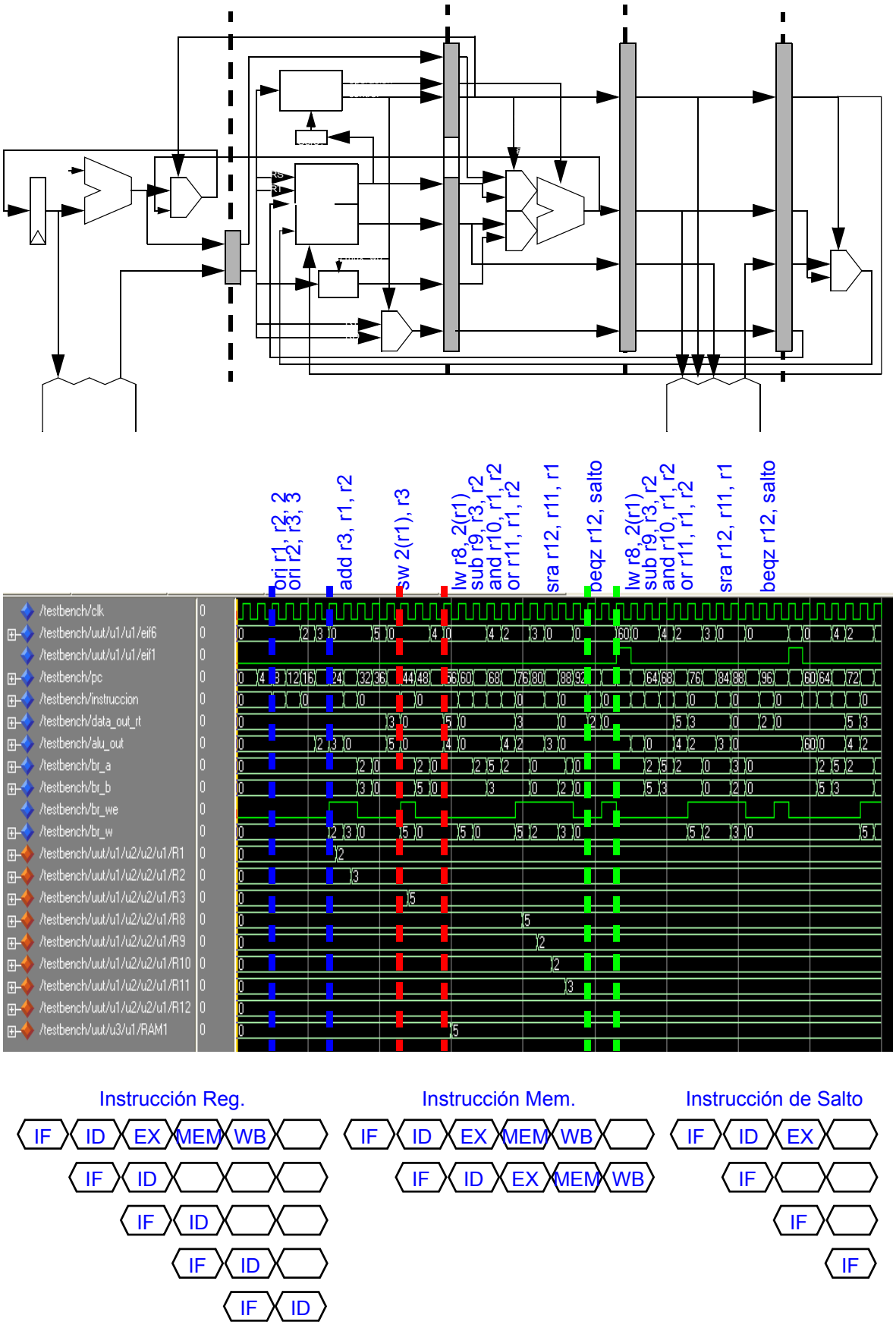

Figura 3.7.- Formas de onda del código ensamblador ejemplo, junto al esquema del procesador DLX implementado.

podemos indicar una serie de directrices para realizar el programa en ensamblador:

- La utilización de un registro debe estar separada en cuatro ciclos de operación desde la instrucción en la que fue actualizado, es decir, la fase WB de la instrucción que actualiza el contenido del registro debe ser anterior a la etapa ID de la instrucción que utilizará el contenido de dicho registro, como se muestra en la figura 3.7.
- La utilización de una palabra de memoria se puede realizar en el siguiente ciclo de operación, ya que su utilización sería en la fase WB que siempre está después de la fase MEM, como se muestra en la figura 3.7.
- Una instrucción de salto debe estar separada tres ciclos de operación de la siguiente instrucción, la etapa EX de la instrucción del salto y la etapa IF de la siguiente instrucción operativa, como se muestra en la figura 3.7. Por lo tanto, después de cada instrucción de saltosiempre debe haber dos no operaciones.
- Todas las operaciones con el mismo número de ciclos de operaciones pueden realizarse consecutivamente (teniendo en cuenta la primera indicación) sin necesidad de añadir ciclos de no operación (nop).

Las formas de onda de la figura 3.7 no pertenecen al software de XILINX, ya que éste no desglosa apropiadamente las matrices de señales. Las formas de onda pertenecientes al software de XILINX se muestran en la figura 3.8. En ella las cuatro primeras señales etiquetadas como tmp_ram[0:15] se corresponde con los dieciséis primeros registros, de talforma que los dos primeros dígitos de cada una de ellas corresponden al registro R0, las dos siguientes al R1 y así sucesivamente. En cambio, las cuatro últimas señales son las dieciséis posiciones más bajas de la memoria dispuestas de la misma forma que los registros. Luego, a los 600 ns (por ejemplo) el valor almacenado en los diferentes registros y posiciones de memoria son los siguientes:

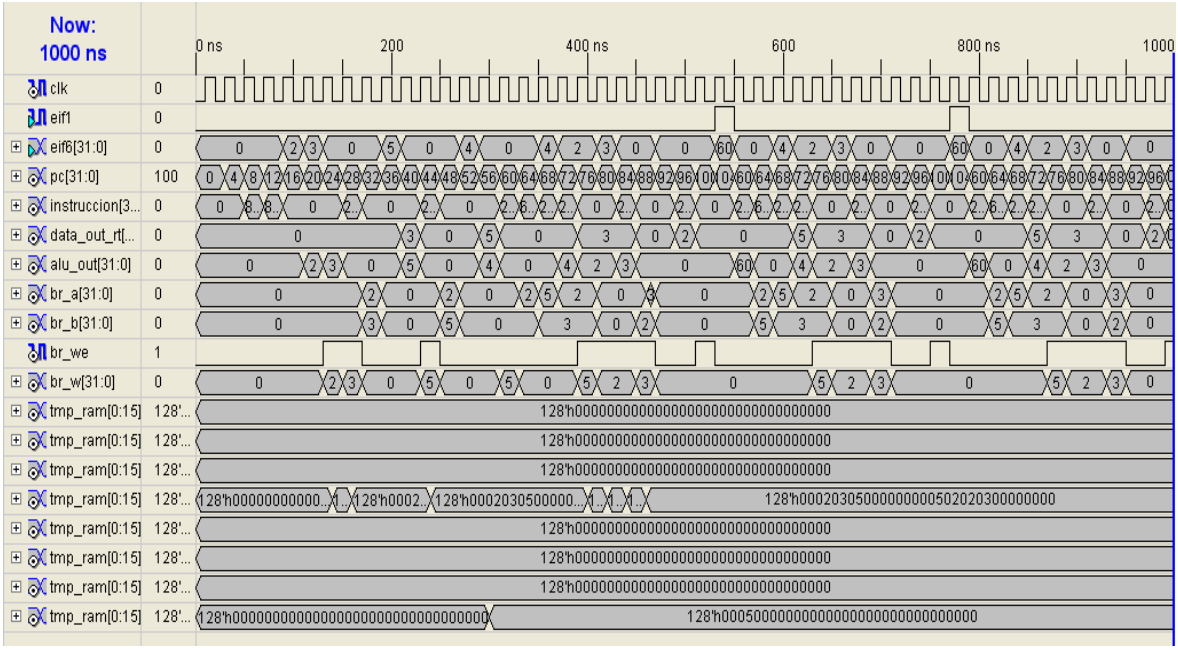

Figura 3.8.- Formas de onda de la simulación de la figura 3.7 en el software de XILINX.

- R1 = 2

- R2 = 3
- R3 = 5
- R8 = 5
- R9 = 2
- R10 = 2
- R11 = 3
- M(1) = 5