Práctica IV

Procesador DLX con unidades de entrada/salida

4.1. Introducción

La utilización normal de las unidades de entrada/salida por parte de un procesador es similar a un acceso a memoria de datos exceptuando que el dato leído proviene de la unidad de entrada/salida en lugar de la memoria. Por lo tanto, la conexión a la memoria debe realizarse a través de un controlador que diferencie entre los dos tipos de accesos. La secuencia de operaciones que vamos a considerar son las siguientes:

- El procesador DLX ejecutará una instrucción de lectura de memoria con una de las cuatro últimas posiciones de memoria, que corresponderán a cuatro unidades de entrada/salida.
- Cuando el controlador del bus detecta que es una transacción de entrada/salida, habilitaremos la señal INTR del procesador para instroducir una burbuja a la espera de que la unidad de entrada/salida acabe su operación.
- Cuando la unidad de entrada/salida acabe la operación, coloca el resultado de su operación en el bus de dato de entrada.
- Seguidamente se deshabilita la señal INTR para que el procesador pueda continuar su operación normal.

Luego, el entorno del procesador quedaría según la estructura mostrada en la figura 4.1. En dicha figura podemos apreciar una sola unidad de entrada/salida, pero en el caso de necesitar más, sus conexiones serían las mismas añadiendo las respectias señales *req*, *ack* y *data_int* para cada una de ellas.

En el caso de que las unidades de entrada/salida necesitasen datos de entrada, estos datos se almacenarían en posiciones fijas de memoria desde la cual serían leídas por estas unidades. Es decir,el procesador escribiría en memoria los datos necesarios por la unidad, la cual posteriormente los leería de dichas posiciones de memoria para ser considerados en su posterior operación.

El código VHDL correspondiente a dicho entorno se muestra a continuación, en el cual se han considerado cuatro unidades de entrada/salida identicas.

```
--
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use work.dlx_pack.all;
use work.dlx_prog2.all; -- donde se encuentra el programa en codigo maquina
use std.textio.all;

entity empotrado is
   port (
      clk: in STD_LOGIC;
```

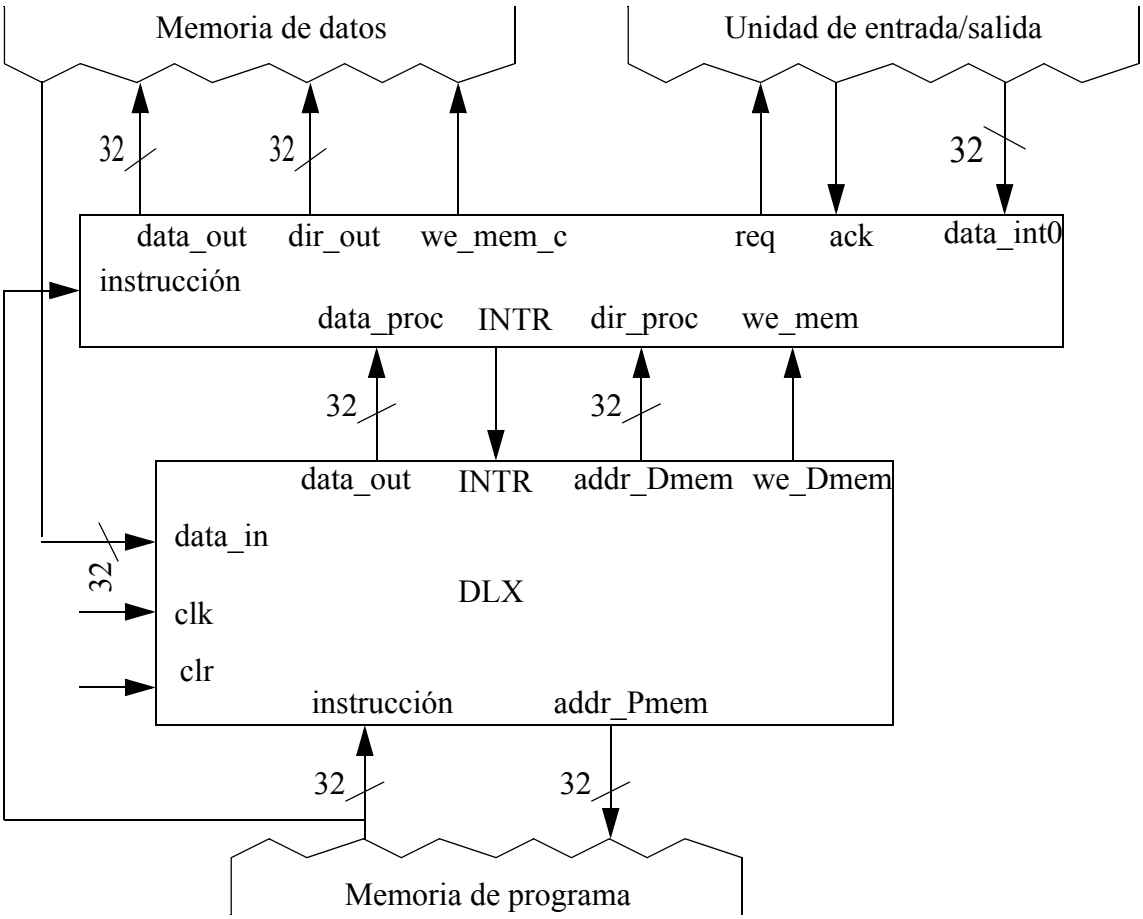


Figura 4.1.- Interfaz del procesador DLX con la memoria de datos y de programa.

```
        clr: in STD_LOGIC;
--          INTR:in std_logic;
        PC: out STD_LOGIC_VECTOR (31 downto 0);
        instruccion: out STD_LOGIC_VECTOR (31 downto 0);
        data_out_Rt: out STD_LOGIC_VECTOR (31 downto 0);
        ALU_out: out STD_LOGIC_VECTOR (31 downto 0);
        BR_A,BR_B,BR_W: out STD_LOGIC_VECTOR (31 downto 0);
        BR_WE: out STD_LOGIC
    );
end empotrado;

architecture main32s_arch of empotrado is

    component DLX32p is -- modulo del procesador DLX32p
        port (
            clk: in STD_LOGIC; -- seal de reloj de todo el sistema computador
            clr: in STD_LOGIC; -- reset del sistema computador
            INTR: in std_logic;
            addr_Pmem: out STD_LOGIC_VECTOR (31 downto 0); -- direccion memoria programa
            instruction: in STD_LOGIC_VECTOR (31 downto 0); -- dato de la memoria programa
            addr_Dmem: out STD_LOGIC_VECTOR (31 downto 0); -- direccion memoria datos
            data_out_Dmem: out STD_LOGIC_VECTOR (31 downto 0); -- dato escrito en memoria
datos
            data_in_Dmem: in STD_LOGIC_VECTOR (31 downto 0); -- dato leido desde memoria
datos
            we_Dmem: out STD_LOGIC; -- habilitacion escritura en memoria de datos
            BR_A,BR_B,BR_W: out STD_LOGIC_VECTOR (31 downto 0); -- datos leidos y escritos
en el banco de registros
```

```
            BR_WE: out STD_LOGIC -- habilitacion de escritura en el banco de registros
        );
    end component ;

    component ROM_32x8 -- memoria de programa
    port(
        ADDR: in ROM_RANGE;
            DATA: out ROM_WORD);
    end component ;

    component MEM -- memoria de datos
    port (
        ADDRESS: in STD_LOGIC_VECTOR (4 downto 0);
        DATA_IN: in STD_LOGIC_VECTOR (31 downto 0);
        DATA_OUT: out STD_LOGIC_VECTOR (31 downto 0);
        WE,CLK : in STD_LOGIC );
    end component ;

    component bus_control
    Port (  clk     : in std_logic;
            -- señales de las unidades de entrada/salida
            ack             : in std_logic_vector(3 downto 0);
            data_int0, data_int1, data_int2, data_int3: in std_logic_vector (31 downto 0);
            -- señales del procesador
            instruccion: in std_logic_vector(31 downto 0);
            we_mem: in std_logic;
            dir_proc: in std_logic_vector (4 downto 0);
            data_proc: in std_logic_vector(31 downto 0);
            -- señales de salida
            INTR, we_mem_c: out std_logic;
            req             : out std_logic_vector(3 downto 0);
            dir_out: out std_logic_vector(4 downto 0);
            data_out: out std_logic_vector(31 downto 0));
    end component;

    component io1
    Port ( clk : in  STD_LOGIC;
         req : in  STD_LOGIC;
         ack : out  STD_LOGIC;
         data : out  STD_LOGIC_VECTOR (31 downto 0));
    end component;

    -- VARIABLES GLOBALES
    signal addr_Pmem,ins,addr_Dmem,data_in,
        data_out: std_logic_vector(31 downto 0) := (others => '0');
    signal addr: signed(31 downto 0):= (others => '0');
    signal addr1: ROM_RANGE;
    signal we, n_clk: std_logic := '0';

    signal intr_int: std_logic;

    signal data_int0, data_int1, data_int2,
        data_int3, data_out1: std_logic_vector (31 downto 0);
    signal addr_D1mem: std_logic_vector (4 downto 0);
    signal req, ack: std_logic_vector (3 downto 0);
    signal we_mem_c: std_logic;

begin -- empieza a describirse la arquitectura

    n_clk <= not(clk); -- la seal de reloj se INVIERTE
--  INTR <= intr_int;

    u1: DLX32p port map (
            clk => clk,
            clr => clr,
            intr => intr_int,
            addr_Pmem => addr_Pmem, -- direccion memoria programa
            instruction => ins, -- se introduce la instruccion en la memoria programa
            addr_Dmem => addr_Dmem, -- se envia la direccion a la memoria de datos
            data_out_Dmem => data_out, -- se envia el dato a la memoria datos
```

```
            data_in_Dmem => data_in, -- se recibe el dato leido desde memoria datos
            we_Dmem => we, -- habilitacion escritura de la memoria de datos
            BR_A => BR_A,
            BR_B => BR_B,
            BR_W => BR_W, -- datos de las salidas A y B y entrada W del banco de registros
            BR_WE => BR_WE -- senyal habilitacion del banco de registros

    );

    instruccion <= ins;
    PC <= addr_Pmem;
    addr <= MVL_TO_SIGNED('0'&'0'& addr_Pmem(31 downto 2)); -- transformamos direccion
para acceder a "dlx_prog2.vhd"
    addr1 <= conv_integer(addr);
    ALU_out <= addr_Dmem;
    data_out_Rt <= data_out;

    u2: ROM_32x8 port map (
        ADDR => addr1, -- por donde se recibe la direccion de la instruccion
        DATA => ins -- por donde se obtiene la instruccion
    );

    u4: bus_control PORT MAP(
        clk => clk,
        ack => ack,
        data_int0 => data_int0,
        data_int1 => data_int1,
        data_int2 => data_int2,
        data_int3 => data_int3,
        instruccion => ins,
        we_mem => we,
        dir_proc => addr_Dmem(6 downto 2),
        data_proc => data_out,
        INTR => intr_int,
        we_mem_c => we_mem_c,
        req => req,
        dir_out => addr_D1mem,
        data_out => data_out1
    );

    u3: MEM port map ( -- La memoria de datos dispone de 32 entradas de 4 bytes cada una de
ellas. Es decir, cada entrada abarca 4 direcciones consecutivas de memoria
        ADDRESS => addr_D1mem, -- por donde se recibe la direccion del dato en la memoria de
datos. El dato es de 4 bytes. Por eso se empieza a direccionar desde el bit 2
        DATA_IN => data_out1, -- por donde se recibe el dato a escribir desde el procesador
        DATA_OUT => data_in, -- por donde se envia el dato a leer por el procesador
        WE => we_mem_c ,CLK => n_clk -- senyales de sincronismo
    );

    u5: io1 port map (
        clk => clk,
        req => req(0),
        data => data_int0,
        ack => ack(0)
    );

    u6: io1 port map (
        clk => clk,
        req => req(1),
        data => data_int1,
        ack => ack(1)
    );

    u7: io1 port map (
        clk => clk,
        req => req(2),
        data => data_int2,
        ack => ack(2)
    );
```

```
    u8: io1 port map (
        clk => clk,
        req => req(3),
        data => data_int3,
        ack => ack(3)
    );

end main32s_arch;
```

4.2. Ejemplo

Para mostrar la utilización del interfaz con el procesador DLX vamos a utilizar como ejemplo el algoritmo de ultiplicación de sumas sucesivas cuando los multiplicandos son introducidos mediantes sendas unidades entrada/salidas. El algoritmo que vamos a imlpementar tendrán los siguientes pasos:

- Leer de la unidad de entrada/salida 1 el primer dato de entrada. Si dicho dato es “0”, volover al incio de la operación.
- Leer de la unidad de entrada/salida 2 el segundo dato de entrada. Si dicho dato es “0”, volver al incio de la operación.
- Una vez que se han almacenado en registros los dos multiplicandos, se obtiene el mayor para que las iteraciones se reduzcan al mínimo.
- Se designa un acumulador, que irá acumulando el operando mayor.
- Se designa un contador que irá contando cada una de las iteraciones.
- Mientras que el contador no alcance al operando menor, se sigue acumulando y contando.
- Cuando el contador alcance al operando menor, se dará por finalizada la operación de multiplicar

Un posible código en ensamblador de dicho algoritmo se muestra a continuación:

```
; Multiplicación r3 = r1 x r2
        nop
        nop
inicio:nop
        lw r10, 30(r0)
        nop
        nop
        nop
        beqz r10, inicio
        nop
        nop
        lw r11, 31(r0)
        nop
        nop
        nop
        beqz r11, inicio
        nop
        nop
        ori r7, r0, 0
; Obtener el menor operando
        or r4, r0, r10
        or r5, r0, r11
```

```
        slt r6, r4, r5
        nop
        nop
        nop
        beqz r6, mult
        nop
        nop
        or r4, r0, r11
        or r5, r0, r10
; Sumas sucesivas
mult:add r3, r3, r4
        addi r7, r7, 1
        nop
        nop
        nop
        slt r6, r7, r5
        nop
        nop
        nop
        beqz r6, inicio
        nop
        nop
        j mult
        nop
        nop
fin:    nop
```

4.2.1. Modelo de la unidad de entrada/salida

A modo de prueba vamos a implementar una sola unidad de entrada/salida. Dicha unidad va a alternar un valor diferente de cero con cero, de tal forma que el comportamiento se puede describir por el diagrama de estados mostrado en la figura 4.2. Cuando estando en los estados de Inicio o Inicio1 llega una petición (una lectura de la posición de memoria asignada a dicha unidad, que activará a señal *req*), se pasará al estado de devolver un valor (ya sea cero o diferente de cero).

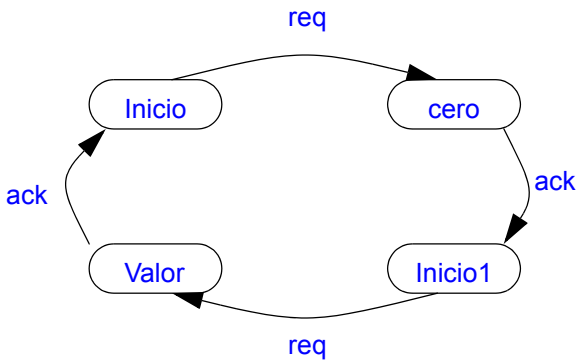


Figura 4.2.- Comportamiento de la unidad de entrada/salida

Un posible código VHDL de dicho comportamiento podría ser el siguiente:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
```

```
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity io1 is
    Port ( clk : in  STD_LOGIC;
           req : in  STD_LOGIC;
           ack : out  STD_LOGIC;
           data : out  STD_LOGIC_VECTOR (31 downto 0));
end io1;

architecture Behavioral of io1 is
    type estado is (inicio, cero, inicio1, valor);
    signal ep, pe: estado;
    signal data_tmp: std_logic_vector (31 downto 0) := (others => '0');
begin
    p1:process (clk, ep)
    begin
        case ep is
            when inicio =>
                ack <= '0';
                data <= (others => '0');
                if (req = '1') then pe <= cero; end if;
            when cero =>
                ack <= '1';
                data <= (others => '0');
                if (req = '0') then pe <= inicio1; end if;
            when inicio1 =>
                ack <= '0';
                data <= (others => '0');
                if (req = '1') then pe <= valor; end if;
            when valor =>
                ack <= '1';
                data <= data_tmp;
                if (req = '0') then
                    pe <= inicio;
                    data_tmp <= data_tmp+5;
                end if;
        end case;
        if (clk = '0' and clk'event) then ep <= pe; end if;
    end process;
end Behavioral;
```

Para la obtención ded icho código se han realizado las siguientes consideraciones:

- El incremento del valor se ha realizado en el cambio de estado desde valor a inicio, para que dicho incremento se lleve a cabo una sola vez en cada ciclo.
- Todos los cambios de estado se llevarán a cabo en el flanco de bajada de la señal de reloj.

4.2.2. Control del bus

El controlador del bus que hemos implementado sigue un protocolo handshake de cuatro fases, es decir, sensible al nivel de las señales de petición (req) y reconocimiento (ack). Un posible código VHDL de este controlador se muestra a continuación

```
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
```

```
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity bus_control is
    Port ( clk  : in std_logic;
                -- señales de las unidades de entrada/salida
                ack            : in std_logic_vector(3 downto 0);
                data_int0, data_int1, data_int2, data_int3: in std_logic_vector (31 downto 0);
                -- señales del procesador
                instruccion: in std_logic_vector (31 downto 0);
                we_mem: in std_logic;
                dir_proc: in std_logic_vector (4 downto 0);
                data_proc: in std_logic_vector (31 downto 0);
                -- señales de salida
                INTR, we_mem_c: out std_logic;
                req            : out std_logic_vector (3 downto 0);
                dir_out: out std_logic_vector (4 downto 0);
                data_out: out std_logic_vector (31 downto 0)
    );
end bus_control;

architecture Behavioral of bus_control is
    type estado is (inicio, interrupcion);
    signal ep, pe: estado;
    signal cop: std_logic_vector(5 downto 0);
    signal dir, dir1: std_logic_vector(4 downto 0);
begin
    cop <= instruccion(31 downto 26);
    dir <= instruccion(6 downto 2);
    p1: process (clk, ep)
    begin
        case ep is
        when inicio =>
            INTR <= '0';
            req <= "0000";
            we_mem_c <= we_mem;
            dir_out <= dir_proc;
            dir1 <= dir;
            data_out <= data_proc;
            if (cop = "100011" and dir > "10011") then pe <= interrupcion;
            end if;
        when interrupcion =>
            INTR <= '1';
            we_mem_c <= '1';
            dir_out <= dir1;
            case dir1 is
                when "11100" => data_out <= data_int0; req <= "0001";
                when "11101" => data_out <= data_int1; req <= "0010";
                when "11110" => data_out <= data_int2; req <= "0100";
                when "11111" => data_out <= data_int3; req <= "1000";
                when others => req <= "0000";
            end case;
            if (ack /= "0000") then pe <= inicio; end if;
        end case;
        ep <= pe;
    end process;
end Behavioral;
```

Es dcir, implementa una máquina de estados con dos estados, inicio e interrupción, cuyas acciones serán las siguientes:

- El estado de inicio es aquel en el que no se atiende a ninguna interrupción ya que no se ha solicitado la intervención de ninguna unidad de entrada/salida. Luego, la habilitación de la memoria de dato y las señales de dirección y de dato se conectan directamente a las señales provenientes del procesador; no se hace ninguna petición a las

unidades de entrada/salida (req <= “0000”); y no se activa la señal de interrupción (INTR) del procesador para detener sus operaciones. De este estado se saldrá cuando exista una instrucción de lectura de la memoria de dato en las cuatro últimas posiciones de memoria, es decir, el campo COP de la instrucción sea “100011” (que corresopnde a la instrucción LW) y el campo de dirección sea mayor que “1011XX” (que corresponde a las cuatro últimas posiciones de memoria, las dos inespecificaciones no se tienen en cuent a la razón de ser es que cada dirección apunta a un byte y no a una palabra).

- El estado de interrupción es aquel en el que la unidad de entrada/salida realiza su operación, y por lo tanto, la operación del procesador estará detenida (según la estrategia seguida). En dicho estado se realizarán las siguientes operaciones: la señal de interrupción (INR) del procesdaor se activará deteniendo su operación; se habilitará la escritura en la memoria de datos; la señal de dirección de la memoria se conectará a la dirección del campo del formato de instrucción; se activará la petición de la unidad de entrada/salida asignada a dicha dirección; el bus de datos de la memoria se conectará al bus de datos de la unidad de entrada/salida asignada. De dicho estado se saldrá cuando la unidad de entrada salida reconozca que su operación ha finalizado, es decir, su señal ack se activará.

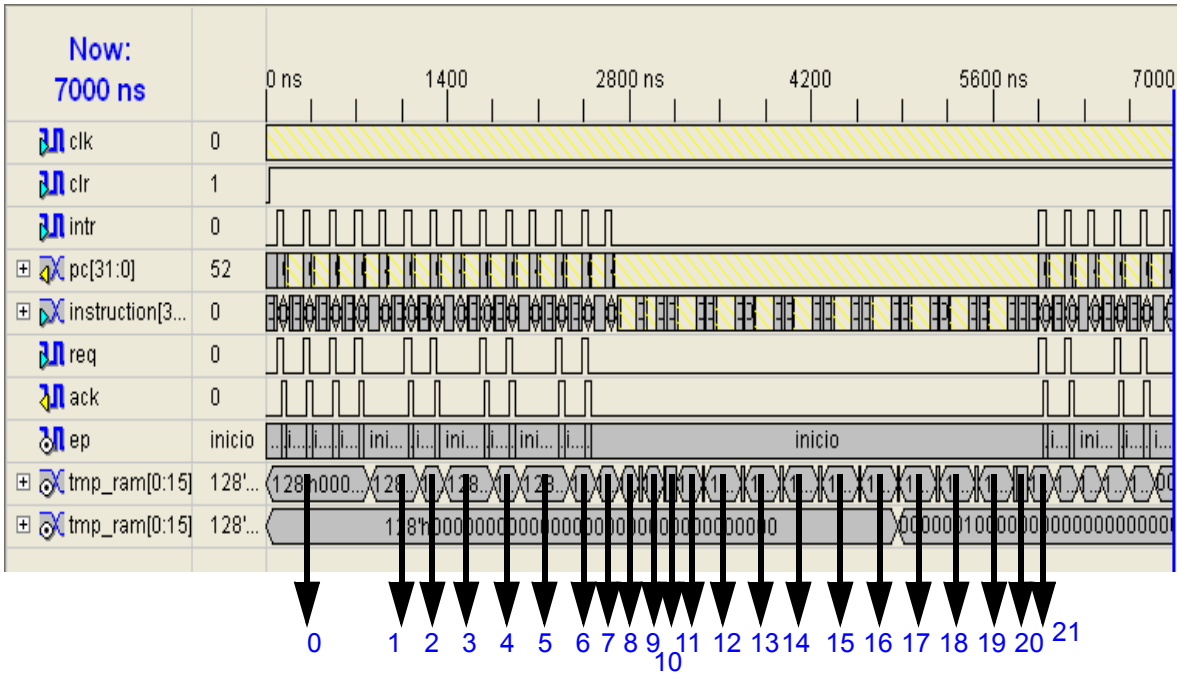
4.2.3. Resultados de simulación

Una vez cargado el programa en ensablador anterior en la memoria de programa y realizada la simulación, obtenemos las formas de onda mostradas en la figura 4.3. En dicha figura mostramos las siguientes señales: clk, clr, INTR, PC, instrución, req, ack, ep (de la unidad de entrada/salida 3), y los cuatro dígitos hexadecmales menos significativos de los dieciséis primeros bits. Para verificar la operación del sistema, observamos los contenidos de los registros que son utilizados en el código ensamblador, es decir, R10, R11, R4, R5, R3, R7 y R6. Debido al zoom de la figura no se pueden apreciar los contenidos de estos registros en la forma de onda, por lo que los hemos sacado a una tabla mostrada en la misma figura de una serie de instantes temporales.

Podemos apreciar que las dos unidades de entrada/salida no producen un valor diferente de cero hasta el instante 8, en el cual cesan las interrupciones. En el instante siguiente vemos como los multiplicandos pasan a los registros R4 y R5, que contendrán al multiplicando mayor y menor respectivamente. En el instante 10 empieza la acumulación en el registro R3 y la cuenta en el registro R7 (para saber cuando acaban las iteraciones). Las acumulaciones acaban en el instante 21, momento en el cual se detecta que el contador (R7) es igual al multiplicando menor, colocando un “0” en el registro R6. A partir de ahí se inicia una nueva multiplicación.

4.3. Ejemplo 2

Utilizando el mismo entorno del ejemplo anterior, obtener los resultados de la simulación de un programa que devuelva el cociente y el resto de una división, en la cual, el dividendo y el divisor serán introducidos por unidades de entrada/salida.



	R10	R11	R4	R5	R3	R7	R6
0	0	0	0	0	0	0	0
1	10	0	0	0	0	0	0
2	0	0	0	0	0	0	0
3	20	0	0	0	0	0	0
4	0	0	0	0	0	0	0
5	30	0	0	0	0	0	0
6	0	0	0	0	0	0	0
7	40	0	0	0	0	0	0
8	40	10	0	0	0	0	0
9	40	10	40	10	0	0	0
10	40	10	40	10	40	1	0
11	40	10	40	10	40	1	1
12	40	10	40	10	80	2	1
13	40	10	40	10	120	3	1
14	40	10	40	10	160	4	1
15	40	10	40	10	200	5	1
16	40	10	40	10	240	6	1
17	40	10	40	10	280	7	1
18	40	10	40	10	320	8	1
19	40	10	40	10	360	9	1
20	40	10	40	10	400	10	1
21	40	10	40	10	400	10	0

Figura 4.3.- Formas de onda del multiplicador con datos provenientes de unidaddes de entrada/salida