
Proyecto de diseño de un procesador sencillo

Memoria

José Ángel Miranda Calero

Contenido

1. Memoria RAM.	4
1.1. Introducción.	4
1.2. VHDL.....	5
1.3. TESTBENCH.....	6
1.4. Simulación.....	7
1.5. Conclusiones.	8
2. Fichero de Registros.	8
2.1. Introducción.	8
2.2. VHDL.....	9
2.3. TESTBENCH.....	12
2.4. Simulación.....	13
3. Unidad Aritmética.	13
3.1. Introducción.	13
3.1.1. Descripción del componente.	14
3.2. Diseño del componente mediante VHDL.	15
3.3. Creación del TestBench y simulación del componente.....	17
3.3.1. Simulación de las operaciones aritméticas con/sin acarreo.	17
3.3.2. Simulación de las operaciones de desplazamiento con/sin acarreo.....	19
3.3.3. Simulación de las operaciones lógicas.	20
4. STACK POINTER.	21
4.1. Introducción.	21
4.1.1. Descripción del componente.	21
4.2. Diseño del componente mediante VHDL.	23
4.3. Simulación del componente y creación del TestBench.	25
4.4. Simulación y obtención de resultados.	26
5. Registros del procesador.	27
5.1. Introducción.	27
5.2. Contador de programa (PC).	28
5.2.2. PC. Diseño del componente mediante VHDL.....	28

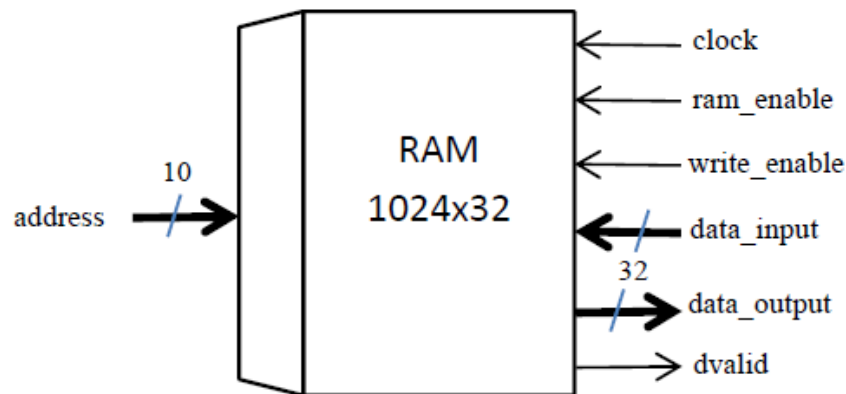
5.2.3. PC. Testbench y simulación.....	29
5.3. Registro de Instrucciones (RI).....	30
5.3.2. RI. Diseño del componente mediante VHDL.....	30
5.3.3. RI. Testbench y simulación.....	30
5.4. Registro de estados (STATUS).....	31
5.4.2. STATUS. Diseño del componente mediante VHDL.....	31
5.4.3. STATUS. Testbench y simulación.....	32
6. Ruta de datos del sistema completo.....	34
6.1. Introducción.....	34
6.1.1. Descripción del componente.....	35
6.2. Diseño del componente mediante VHDL.....	36
7. Unidad de control y Simulación final.....	39
7.1. Manejo de las señales de control.....	41
7.1.2. FETCH1.....	43
7.1.2. FETCH2.....	43
7.1.3. SUM & SUB.....	44
7.1.4. SHR & SHL.....	45
7.1.5. INOT, IAND, IOR, IXOR.....	45
7.1.6. LD.....	45
7.1.7. ST.....	46
7.1.8. JMP.....	47
7.1.9. JSR & RTS.....	47
7.1.10. CLC & SETC.....	48
7.1.11. STOP & NOP.....	48
7.2. Carga en memoria.....	48
7.3. Simulación del sistema.....	50
7.3.1. SUM & SUB Simulación.....	50
7.3.2. ST Simulación.....	51
7.3.2. SHL & SHR Simulación.....	52
7.3.3 Operaciones lógicas. Simulación.....	52
7.3.4. Otras simulaciones.....	53

1. Memoria RAM.

1.1. Introducción.

El objetivo de esta primera parte del proyecto es el diseño y simulación de una memoria RAM de puerto único usando, para ello, un lenguaje de descripción hardware. En este caso, se ha decidido utilizar el lenguaje VHDL usando como herramienta la plataforma ModelSim.

El bloque que describe la memoria consta de las siguientes entradas y salidas:



Señal	Tipo	Anchura	Descripción
clock	entrada	1 bit	Señal de reloj
ram_enable	entrada	1 bit	Habilitación del fichero de registro
write_enable	entrada	1 bit	Habilitación de escritura
data_input	entrada	32 bits	Bus de datos de entrada. El dato se almacena en el registro indicado por "address" cuando "ram_enable=1 y write_enable=1".
data_output	salida	32 bits	Bus de datos de salida. Muestra el dato almacenado en el registro direccionado por "address".
address	entrada	10 bits	Bus de dirección
dvalid	salida	1 bit	Indica que el dato en el bus de datos de salida es válido

1.2. VHDL.

Teniendo en cuenta la descripción de dicha memoria, podemos pasar a definir la entidad que usaremos en nuestro diseño:

```

1  --
2  -- Máster en Microelectrónica: Diseño y Aplicaciones de Sistemas Micro/Nanométricos
3  -- Asignatura: Aplicaciones, sistemas y técnicas para el tratamiento de la información
4  -- Course: Applications, systems and techniques for information processing
5  --
6  -- Create Date:    2014/2015
7  -- Module Name:    RAM_single_port - Behavioral
8  --
9
10 library IEEE;
11 use IEEE.STD_LOGIC_1164.ALL;
12 use IEEE.numeric_std.all;
13 use IEEE.std_logic_unsigned.all;
14
15 entity RAM_single_port is
16     generic (N: integer:=32; -- memory width
17             M: integer:=10); -- address width
18     Port ( clock      : in  STD_LOGIC;
19           ram_enable  : in  STD_LOGIC;
20           write_enable: in  STD_LOGIC;
21           address     : in  STD_LOGIC_VECTOR (M-1 downto 0);
22           data_input  : in  STD_LOGIC_VECTOR (N-1 downto 0);
23           data_output : out STD_LOGIC_VECTOR (N-1 downto 0);
24           dvalid      : out STD_LOGIC);
25 end RAM_single_port;

```

A continuación, desarrollaremos la arquitectura que manejará dicha entidad:

```

26
27 architecture Behavioral of RAM_single_port is
28     type memory_type is array ((2**M)-1 downto 0) of STD_LOGIC_VECTOR (N-1 downto 0);
29     signal memory : memory_type;
30
31     --Register to hold the address
32     signal addr_reg : STD_LOGIC_VECTOR (M-1 downto 0) := (others => '0');
33 begin
34     -- Iniciar aquí la descripción del comportamiento de la memoria RAM
35     process( clock )
36     begin
37         if( clock = '1' ) then
38             if( ram_enable = '1' ) then
39                 if( write_enable = '1' ) then
40                     dvalid <= '0';
41                     memory( conv_integer( address ) ) <= data_input;
42                 else
43                     dvalid <= '1';
44                     addr_reg <= address;
45                 end if;
46             end if;
47         end if;
48     end process;
49
50     data_output <= memory( conv_integer( addr_reg ) );
51
52     -- Fin de la descripción
53 end Behavioral;
54

```

Como podemos observar en el único proceso de la arquitectura, las siguientes condiciones se cumplen:

1. La memoria será accesible siempre y cuando la señal 'ram_enable' esté habilitada.
2. En caso de encontrarse la memoria habilitada y la escritura deshabilitada, en cada flanco de subida de reloj, se mostrará el data direccionado por 'address' en el bus 'data_output' activándose a su vez la señal 'dvalid'.
3. En el caso de activarse la escritura, el data del bus 'data_in' se escribe en la palabra de la memoria direccionada por 'address'.

Cabe destacar que se ha utilizado un array bidimensional (tipo 'memory_type') el cual se usa para el almacenamiento de las distintas palabras dentro de dicha memoria. Además, nos hemos apoyado de funciones propias de la librería 'IEEE.numeric_std', como 'conv_integer(<std_logic_vector>)' para poder acceder o guardar las palabras en la memoria.

Dentro de la arquitectura y fuera del proceso, se ha definido la asignación del bus 'data_output', puesto que en VHDL los puertos de salida de la entidad no se actualizan hasta que acaba el proceso en el que se encuentran, se ha decidido utilizar una señal 'addr_reg', la cual se actualiza en el momento de su cambio, asignando de esta manera de manera conveniente el bus 'data_output' al valor correspondiente en el momento indicado.

El proceso utilizado y la asignación del bus 'data_output' se ejecutan de manera simultánea.

1.3. TESTBENCH.

En el fichero utilizado para realizar la simulación de dicho diseño, cabe destacar el proceso 'stimulus_process' realizado dentro de la arquitectura 'Bench':

```

67 stimulus_process: process
68 begin
69
70     --hold reset state
71     wait for clock_period/2;
72
73     --The RAM is enabled
74     ram_enable <= '1';
75
76     --Two write actions are taken
77     write_enable <= '1';
78     address <= std_logic_vector( to_unsigned( 512, address'length ));
79     data_input <= std_logic_vector( to_unsigned( 1752132705, data_input'length ));
80
81     wait for clock_period;
82
83     address <= std_logic_vector( to_unsigned( 12, address'length ));
84     data_input <= std_logic_vector( to_unsigned( 1634496360, data_input'length ));
85
86     wait for clock_period;
87
88     write_enable <= '0';
89     address <= std_logic_vector( to_unsigned( 512, address'length ));
90
91     wait for clock_period;
92
93     address <= std_logic_vector( to_unsigned( 12, address'length ));
94
95     wait for clock_period/2;
96
97 end process stimulus_process;
98
99 end Bench;

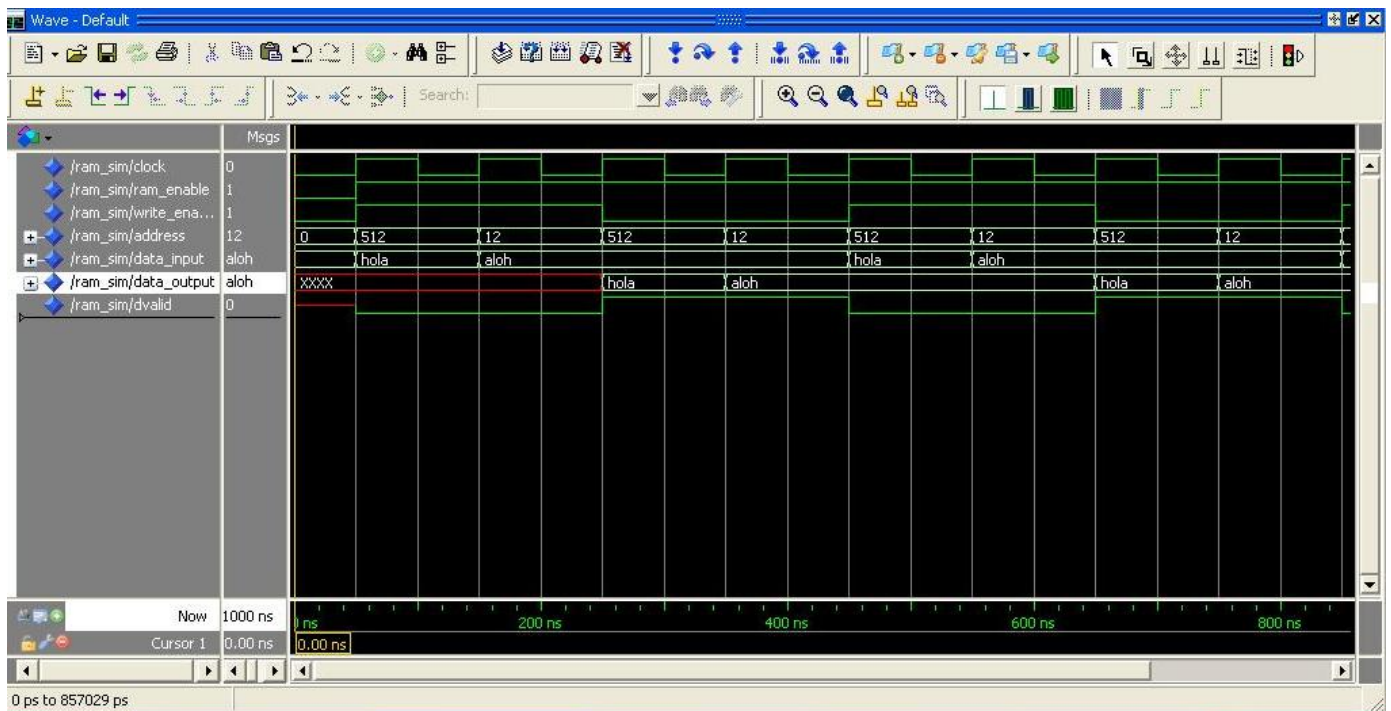
```

Podemos observar como en este proceso se realizan dos escrituras y dos lecturas. Específicamente, durante la primera escritura se escribe en la localización de memoria '512' la palabra 'hola' y durante la segunda escritura se escribe en la localización '12' la palabra 'aloh'. Tras realizar dichas escrituras, se procede a la lectura, donde se leen la primera y segunda escritura respectivamente.

El periodo de reloj ha sido seleccionado arbitrariamente como 100ns.

1.4. Simulación.

En la simulación podemos observar como lo anteriormente descrito se cumple y verifica:



Se observa la escritura en las dos zonas de memoria seleccionadas y su lectura al desactivarse la escritura y coincidiendo con el flanco de subida del reloj.

1.5. Conclusiones.

Es remarcable una posible mejora en dicha memoria, ya que en el momento en el que 'dvalid' tiene el valor '1' y se leen las dos escrituras, cuando vuelve a valer '0', en el bus 'data_output' se observa la última lectura. Esto podría modificarse en el diseño de tal manera que se observasen datos no validos 'XXX' como pasa al principio.

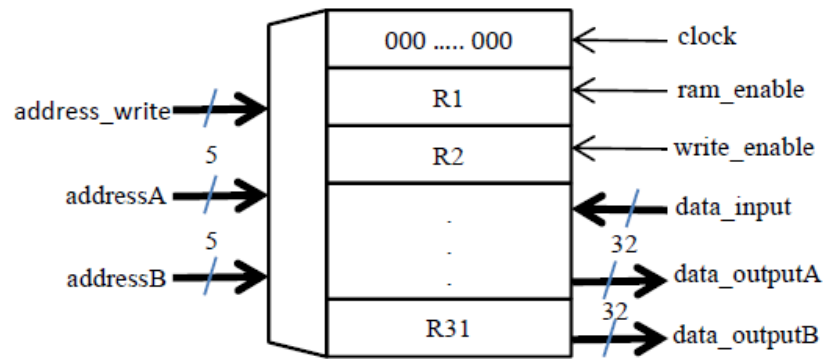
2. Fichero de Registros.

2.1. Introducción.

El objetivo de esta segunda parte del proyecto es el diseño y simulación del fichero de registros que usará el simulador.

Como en el apartado anterior, donde diseñamos la memoria RAM del sistema, en este caso, se ha decidido utilizar de nuevo el lenguaje VHDL usando como herramienta la plataforma ModelSim.

El bloque que describe el fichero consta de las siguientes entradas y salidas:



Señal	Tipo	Anchura	Descripción
clock	entrada	1 bit	Señal de reloj
ram_enable	entrada	1 bit	Habilitación del fichero de registro
write_enable	entrada	1 bit	Habilitación de escritura
data_input	entrada	32 bits	Bus de datos de entrada. El dato se almacena en el registro indicado por "address_write" cuando "ram_enable=1 y write_enable=1".
data_outputA	salida	32 bits	Bus de datos de salida. Muestra el dato almacenado en el registro direccionado por "addressA".
data_outputB	salida	32 bits	Bus de datos de salida. Muestra el dato almacenado en el registro direccionado por "addressB".
address_write	entrada	5 bits	Bus de direcciones para escritura.
addressA	entrada	5 bits	Bus de dirección para lectura por el puerto "addressA".
addressB	entrada	5 bits	Bus de dirección para lectura por el puerto "addressB".

2.2. VHDL.

Teniendo en cuenta la descripción de dicho fichero de registros, podemos pasar a definir la entidad que usaremos en nuestro diseño. Hay que tener en cuenta que, debido a la similitud con la memoria RAM, en cuanto a puertos de entrada/salida, el diseño del fichero de registros se asemeja bastante al de dicha memoria.

```

1
2
3  -- Máster en Microelectrónica: Diseño y Aplicaciones de Sistemas Micro/Nanométricos
4  -- Asignatura: Aplicaciones, sistemas y técnicas para el tratamiento de la información
5  -- Course: Applications, systems and techniques for information processing
6
7  -- Create Date:    2014/2015
8  -- Module Name:    file_register - Behavioral
9
10
11  library IEEE;
12  use IEEE.STD_LOGIC_1164.ALL;
13  use IEEE.STD_LOGIC_ARITH.ALL;
14  use IEEE.STD_LOGIC_UNSIGNED.ALL;
15
16  entity file_register is
17  Port ( clock      : in  STD_LOGIC;
18        ram_enable  : in  STD_LOGIC;
19        write_enable : in  STD_LOGIC;
20        address_write: in  STD_LOGIC_VECTOR (4 downto 0);
21        addressA     : in  STD_LOGIC_VECTOR (4 downto 0);
22        addressB     : in  STD_LOGIC_VECTOR (4 downto 0);
23        data_input   : in  STD_LOGIC_VECTOR (31 downto 0);
24        data_outputA : out STD_LOGIC_VECTOR (31 downto 0);
25        data_outputB : out STD_LOGIC_VECTOR (31 downto 0));
26  end file_register;
27

```

A continuación, desarrollaremos la arquitectura que manejará dicha entidad:

```

28  architecture Behavioral of file_register is
29  -- Definir aquí señales, tipos, funciones, etc
30  type file_type is array (31 downto 0) of STD_LOGIC_VECTOR (31 downto 0);
31  signal file_reg: file_type := ((others=> (others=>'0')));
32
33  --Registro para guardar las direcciones de lectura:
34  signal addr_regA : STD_LOGIC_VECTOR (4 downto 0) := (others => '0');
35  signal addr_regB : STD_LOGIC_VECTOR (4 downto 0) := (others => '0');
36  -- Fin de definiciones
37  begin
38  -- Iniciar aquí la descripción
39  process( clock )
40  begin
41      if( clock = '1' ) then
42
43          if( ram_enable = '1' ) then
44
45              if( write_enable = '1' ) then
46
47                  if( address_write /= "00000" ) then
48
49                      file_reg( conv_integer( address_write )) <= data_input;
50
51                  end if;
52
53              else
54
55                  addr_regA <= addressA;
56                  addr_regB <= addressB;
57
58              end if;
59
60          end if;

```

```

61
62         end if;
63
64     end process;
65
66     data_outputA <= file_reg( conv_integer( addr_regA ));
67     data_outputB <= file_reg( conv_integer( addr_regB ));
68
69     -- Fin de la descripción
70
71 end Behavioral;
72

```

Como podemos observar en el único proceso de la arquitectura, las siguientes condiciones se cumplen:

1. El fichero de registros será accesible siempre y cuando la señal 'ram_enable' esté habilitada.
2. En caso de encontrarse el fichero habilitado y la escritura deshabilitada, en cada flanco de subida de reloj, se mostrará el data guardado en el registro referenciado por 'addressA' y 'addressB' en sus respectivos buses.
3. En el caso de activarse la escritura, el data del bus 'data_in' se escribe en el registro referenciado por 'address_write'.
4. No se permite la escritura en el primer registro del fichero y este siempre tiene valor cero (se inicializa a cero al declarar la señal 'file_reg' de tipo 'file_type').

Cabe destacar que se ha utilizado un array bidimensional (tipo "file_type ") el cual se usa para el almacenamiento de las distintas palabras dentro de cada uno de los registros. Además, nos hemos apoyado de funciones propias de la librería 'IEEE.numeric_std', como 'conv_integer(<std_logic_vector>)' para poder acceder o guardar las palabras en los registros.

Dentro de la arquitectura y fuera del proceso, se ha definido la asignación del bus 'data_outputA' y 'data_outputB', puesto que en VHDL los puertos de salida de la entidad no se actualizan hasta que acaba el proceso en el que se encuentran, se ha decidido utilizar dos señales 'addr_regA' y 'addr_regB', las cuales se actualizan en el momento de su cambio, asignando de esta manera de manera conveniente los buses de lectura valor correspondiente en el momento indicado.

El proceso utilizado y la asignación de los buses de salida para A y B se ejecutan de manera simultánea.

2.3. TESTBENCH.

En el fichero utilizado para realizar la simulación de dicho diseño, cabe destacar el proceso 'stimulus_process' realizado dentro de la arquitectura 'Bench'.

```

72  --Stimulus process
73  stimulus_process: process
74  begin
75
76      --hold reset state
77      wait for clock_period/2;
78
79      --The RAM is enabled
80      ram_enable <= '1';
81
82      --Two write actions are taken
83      write_enable <= '1';
84      address_write <= std_logic_vector( to_unsigned( 1, address_write'length ));
85      data_input <= std_logic_vector( to_unsigned( 1752132705, data_input'length ));
86
87      wait for clock_period;
88
89      address_write <= std_logic_vector( to_unsigned( 10, address_write'length ));
90      data_input <= std_logic_vector( to_unsigned( 1634496360, data_input'length ));
91
92      wait for clock_period;
93
94      write_enable <= '0';
95      addressA <= std_logic_vector( to_unsigned( 1, addressA'length ));
96      addressB <= std_logic_vector( to_unsigned( 10, addressB'length ));
97      address_write <= std_logic_vector( to_unsigned( 2, address_write'length ));
98      data_input <= std_logic_vector( to_unsigned( 1634496360, data_input'length ));
99
100     wait for clock_period;
101
102     write_enable <= '1';
103
104     address_write <= std_logic_vector( to_unsigned( 0, address_write'length ));
105     data_input <= std_logic_vector( to_unsigned( 1634496360, data_input'length ));
106
107     wait for clock_period;
108
109     write enable <= '0';
110
111     addressA <= std_logic_vector( to_unsigned( 0, addressA'length ));
112     addressB <= std_logic_vector( to_unsigned( 2, addressB'length ));
113
114     wait for clock_period/2;
115     end process stimulus_process;

```

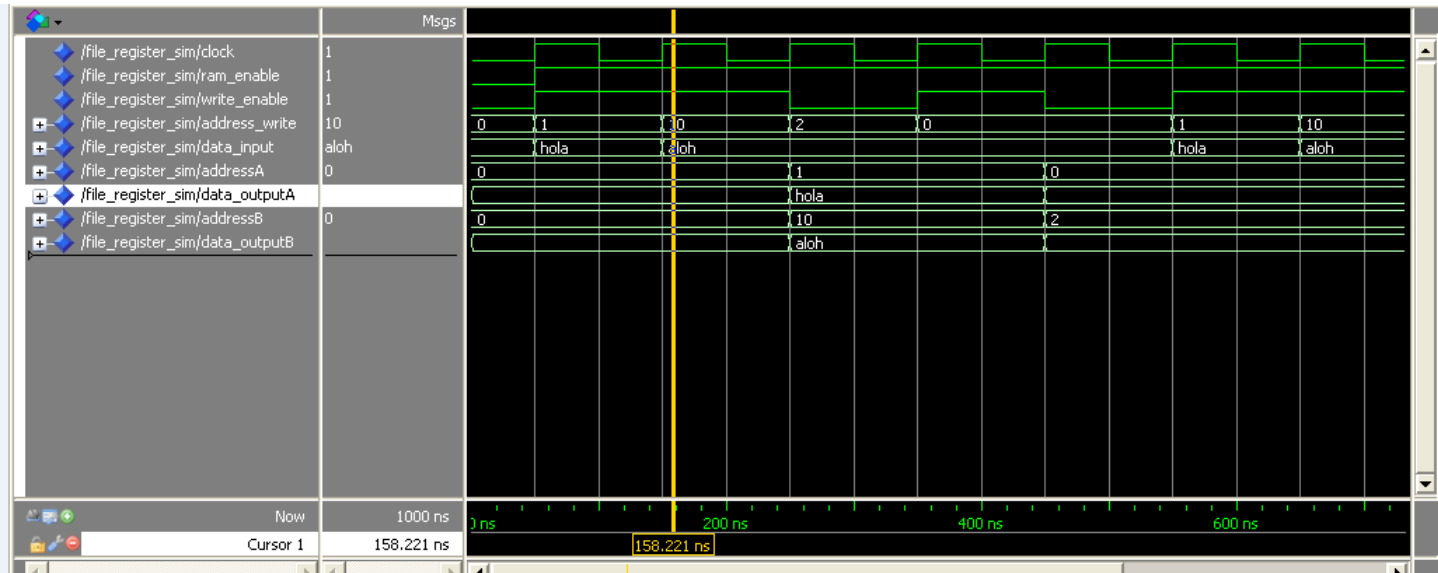
Podemos observar como en este proceso se realizan dos escrituras y dos lecturas. Específicamente, durante la primera escritura se escribe en el registro numero uno la palabra 'hola' y durante la segunda escritura se escribe en el registro número diez la palabra 'aloh'. Tras realizar dichas escrituras, se procede a la lectura, donde se leen la primera y segunda escritura desde los buses A y B respectivamente.

También se lleva a cabo la escritura en el registro número cero y en el número dos, ambas sin poder realizarse debido a la restricción de escritura en el registro número cero y a la des habilitación de la señal 'write_enable' al escribir en el registro número dos.

El periodo de reloj ha sido seleccionado arbitrariamente como 100ns.

2.4. Simulación.

En la simulación podemos observar como lo anteriormente descrito se cumple y verifica:



Se observa la escritura en los registros uno y diez y su lectura, al desactivarse la escritura y coincidiendo con el flanco de subida del reloj, en los buses A y B. También nos percatamos de la escritura en los registros número dos y cero y su posterior lectura verificando el comportamiento descrito por las especificaciones del sistema.

3. Unidad Aritmética.

3.1. Introducción.

En esta parte del proyecto, se llevará a cabo el diseño y simulación de la ALU (Arithmetic Logic Unit) del procesador. Dicha ALU es un circuito digital combinacional que lleva a cabo operaciones aritméticas, lógicas y desplazamientos; en este caso en concreto, la ALU podrá realizar hasta un máximo de ocho operaciones. Ésta es un elemento fundamental e imprescindible para el correcto funcionamiento del futuro procesador, ya que dicha unidad será la encargada de llevar a cabo todas las operaciones en las que el procesador esté implicado.

Una vez más se ha decidido realizar la implementación usando VHDL y ModelSim debido a la familiaridad adquirida con el lenguaje y dicho software tras la realización de los anteriores entregables.

3.1.1. Descripción del componente.

Como ya se ha comentado, la ALU podrá realizar hasta ocho operaciones dependiendo de las señales de control recibidas y los dos operandos de entrada, generando de esta manera un resultado en la salida de datos.

En la operaciones aritméticas y de desplazamiento, el componente recibe un acarreo de entrada y genera uno de salida; además de esto, en las operaciones aritméticas también se genera la salida de estado "Cero", la cual indica que el dato de salida de la ALU es cero. En cuanto a las operaciones de desplazamiento, se realizan usando el acarreo de entrada como parte del dato de entrada y genera un bit en el acarreo de salida.

Llegados a este punto podemos definir la tabla de operaciones de la ALU:

Tabla. Operaciones de la ALU		
Control	Operación	Descripción
000	$F_{out} = R_{in} + S_{in} + C_{in}$	Suma
001	$F_{out} = R_{in} - S_{in} - C_{in}$	Resta
010	$\{C_{out}, F_{out}\} = \{R_{in}(30:0), C_{in}\}$	Rotación a la derecha
011	$\{F_{out}, C_{out}\} = \{C_{in}, R_{in}(31:1)\}$	Rotación a la izquierda
100	$F_{out} = \text{not } R_{in}$	Not
101	$F_{out} = R_{in} \text{ and } S_{in}$	And
110	$F_{out} = R_{in} \text{ or } S_{in}$	Or
111	$F_{out} = R_{in} \text{ xor } S_{in}$	Xor

Así, las entradas y salidas de nuestro componente se describen y especifican a continuación:

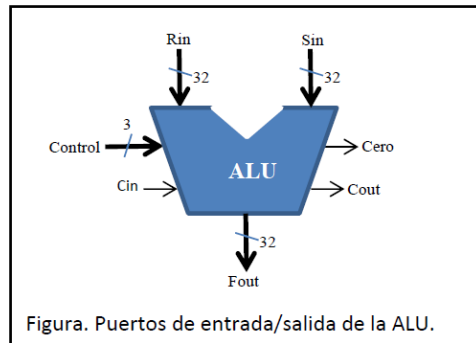
1. Entradas.

- 1.1. Control - 3 bits. Especifica el tipo de operación a realizar por el componente.
- 1.2. Cin - 1 bit. Acarreo de entrada.
- 1.3. Rin - 32 bits. Bus de datos de entrada.
- 1.4. Sin - 32 bits. Bus de datos de entrada.

2. Salidas.

- 2.1. Cero - 1 bit. Señal encargada de indicar el momento en el que la salida de datos en operaciones aritméticas es cero.
- 2.2. Cout - 1 bit. Acarreo de salida.

2.3. Fout - 32 bits. Bus de datos de salida.



3.2. Diseño del componente mediante VHDL.

En este punto, el código será comentado paso por paso, desde la entidad usada hasta los procesos implementados en el diseño.

En primer lugar, la entidad que se ha codificado es la siguiente:

```

14 entity alu is
15   port (Rin, Sin : In std_logic_vector (31 downto 0); -- Entrada de datos
16         Cin      : In std_logic;                -- Entrada de acarreo
17         Fout     : Out std_logic_vector (31 downto 0); -- Salida de datos
18         Cout     : Out std_logic;                -- Salida de acarreo
19         Cero     : Out std_logic;                -- Salida de resultado igual a cero
20         Control  : In std_logic_vector (2 downto 0)); -- Entradas de control de operacion
21 end alu;
22

```

Figura 2. Entidad del componente.

Se pueden distinguir las distintas entradas y salidas ya descritas en el anterior apartado.

Para llevar a cabo la funcionalidad del componente nos hemos servido única y exclusivamente del uso de un proceso y una variable interna en dicho proceso. Esta variable ha sido utilizada para guardar el valor temporal de las operaciones aritméticas realizadas dentro del proceso (suma o resta), asignando dicha variable al bus de datos de salida y a la señal de acarreo de salida como se explica detalladamente más adelante. Esta manera de realizar dicho proceso se ha llevado a cabo de esa forma debido al retardo de actualización de las señales dentro de un proceso en VHDL (hasta que el proceso no acaba no se actualiza el valor de dicha señal, en cambio el valor de una variable se actualiza al instante de operar con ella).

```

23 architecture funcional of alu is
24   -- Definir aqui señales, tipos, funciones, etc
25   -- Fin de definiciones
26 begin
27   -- Iniciar aqui la descripción
28   process( Control,Rin,Sin,Cin ) is
29     variable temp: std_logic_vector(32 downto 0) := (others => '0');
30     begin

```

Figura 3. Variable utilizada durante el proceso del

componente.

A continuación se muestra el código correspondiente a las dos operaciones aritméticas realizadas:

```

31 case Control is
32
33     when "000" => -- Suma
34         temp := ('0' & Rin) + Sin + Cin;
35         Fout <= temp(31 downto 0);
36         Cout <= temp(32);
37         if( temp = 0 ) then
38             Zero <= '1';
39         else
40             Zero <= '0';
41         end if;
42
43     when "001" => -- Resta
44         temp := ('0' & Rin) - Sin - Cin;
45         Fout <= temp(31 downto 0);
46         Cout <= temp(32);
47         if( temp = 0 ) then
48             Zero <= '1';
49         else
50             Zero <= '0';
51         end if;

```

Figura 4. Operación de suma y resta.

A la hora de describir el código de la figura 4, el cual pertenece a las operaciones de suma y resta, cabe destacar el uso de la variable 'temp' como registro temporal del valor de la operación y como esta variable posee 33, mientras que los demás buses del sistema solo tienen 32 bits, lo cual permite almacenar en el último de ellos el bit de acarreo de salida y asignarlo de ese modo a la señal 'Cout'. Remarcar también que para poder operar con vectores de 32 bits asignando su operación a un vector de 33 bits, como se realiza con estas operaciones aritméticas con la variable 'temp', es necesario realizar cierto "añadido a mano" para que al menos uno de los vectores de 32 bits se convierta en uno de 33 bits ('0' & Rin).

En la siguiente figura se muestra las operaciones referidas a la rotación o desplazamiento hacia la derecha y hacia la izquierda.

```

53 when "010" => -- Rotación a la Derecha
54     Fout <= Cin & Rin(31 downto 1);
55     Cout <= Rin(0);
56
57 when "011" => -- Rotación a la Izquierda
58     Fout <= Cin & Rin(30 downto 0);
59     Cout <= Rin(31);

```

Figura 5. Operaciones de desplazamiento.

Por último podemos analizar el trozo de código correspondiente a las operaciones lógicas (NOT, AND, OR y XOR):


```

61      when "100"=> -- NOT
62          Fout <= not Rin;
63
64      when "101"=> -- AND
65          Fout <= Rin and Sin;
66
67      when "110"=> -- OR
68          Fout <= Rin or Sin;
69
70      when "111"=> -- XOR
71          Fout <= Rin xor Sin;
72
73      when others => --nothing
74          Fout <= (others => '0');
75
76      end case;
77      end process;
78      -- Fin de la descripción
79      end funcional;

```

Figura 6. Operaciones lógicas y fin del proceso.

Al finalizar el 'switch' basado en la señal de control 'Control' se debe especificar 'when others' (otras posibles elecciones que no estén contempladas dentro de los casos anteriores), para lo cual se ha decidido darle un valor de cero al bus de salida de datos.

3.3. Creación del TestBench y simulación del componente.

En este apartado se analizará el proceso de estímulos realizado para chequear y testear la funcionalidad de nuestro componente. No se explicará en ningún apartado de este documento el diseño del TestBench, ni la creación de la UUT, etc. ya que esto se supone por conocido.

Las funcionalidades a probar son las siguientes:

1. Correcto funcionamiento de las funciones aritméticas de la ALU.
2. Correcto funcionamiento de las funciones de desplazamiento de la ALU.
3. Correcto funcionamiento de las funciones lógicas de la ALU.
4. Correcto funcionamiento de los acarreo, tanto de entrada como de salida.
5. Correcto funcionamiento de la señal 'Cero'.

3.3.1. Simulación de las operaciones aritméticas con/sin acarreo.

En primer lugar se realiza el test para comprobar el correcto funcionamiento de la operación suma y se aprovecha para chequear también la funcionalidad de los acarreo, por lo que tendríamos cubiertos el primer y cuarto punto.

```

53  --Stimulus process
54  stimulus_process: process
55  begin
56      --SUMA
57      -- con Cin y con Cout
58      Control<="000";
59      Rin<=x"80000000";
60      Sin<=x"80000000";
61      Cin<='1';
62
63      wait for 10 ns;
64
65      -- sin Cin y sin Cout
66      Rin<=std_logic_vector( to_unsigned( 30, Rin'length ));
67      Sin<=std_logic_vector( to_unsigned( 30, Rin'length ));
68      Cin<='0';
69
70      wait for 10 ns;
71
72      -- con Cin y sin Cout
73      Rin<=std_logic_vector( to_unsigned( 30, Rin'length ));
74      Sin<=std_logic_vector( to_unsigned( 30, Rin'length ));
75      Cin<='1';
76
77      wait for 10 ns;
78
79      -- sin Cin y con Cout
80      Rin<=x"80000000";
81      Sin<=x"80000000";
82      Cin<='0';
83
84      wait for 10 ns;

```

Figura 7. Test ALU suma y acarreo.

	Msgs								
/alu_sim/Control	000	000							
/alu_sim/Rin	80000000	(80000000		0000001E			80000000		
/alu_sim/Sin	80000000	(80000000		0000001E			80000000		
/alu_sim/Cin	0								
/alu_sim/Fout	00000000	(00000001		0000003C		0000003D		00000000	
/alu_sim/Cout	1								
/alu_sim/Cero	1								

En las formas de onda mostradas se observan los cuatro test realizados para comprobar el **correcto funcionamiento de la operación suma y de los acarreo**; se observa además que, **en el último test, la señal 'Cero' se activa al detectar que la salida del bus de datos es cero**, por lo que el comportamiento de esta señal queda de la misma manera verificado.

Siguiendo con las pruebas, se pasa a la **evaluación de la resta**. Se considera que, tras haber probado la funcionalidad de los acarreo de entrada y salida con la operación suma, no es necesario verificar la simulación de los acarreo en la resta debido a la similitud del código.

```

85
86  --resta
87  Control<="001";
88  Rin<=std_logic_vector( to_unsigned( 31, Rin'length ));
89  Sin<=std_logic_vector( to_unsigned( 30, Rin'length ));
90  Cin<='1';
91
92  wait for 10 ns;

```

Figura 8. Test ALU resta.

Msgs									
+ /alu_sim/Control	000	000					001		010
+ /alu_sim/Rin	80000000	0000001E			80000000		0000001F		00000000
+ /alu_sim/Sin	80000000	0000001E			80000000		0000001E		
+ /alu_sim/Cin	0								
+ /alu_sim/Fout	00000000	0000003C		0000003D		00000000			00000000
+ /alu_sim/Cout	1								
+ /alu_sim/Cero	1								

El recuadro amarillo en la simulación indica la zona en la que se lleva a cabo la resta, en este caso específico vemos como ' $C_{in} = 1$ '; de manera que:

$$F_{out} = x^{''0000001F''} - x^{''0000001E''} - 1 = x^{''00000000''}$$

3.3.2. Simulación de las operaciones de desplazamiento con/sin acarreos.








Se verifica la funcionalidad ahora de las operaciones de desplazamiento, para este caso se realizan dos test para cada operación, con y sin acarreo de salida.

```

94      -- Rotacion a la Drch
95      -- Rotacion a la Drch con Cout
96      Control<="010";
97      Rin<=x"C0000011";
98      Cin<='0';
99      wait for 10 ns;
100
101      -- Rotacion a la Drch sin Cout
102      Rin<=x"C0000000";
103      Cin<='0';
104      wait for 10 ns;
105
106      --Rotacion a la Izq
107      -- Rotacion a la Izq con Cout
108      Control<="011";
109      Rin<=x"C0000000";
110      Cin<='0';
111      wait for 10 ns;
112
113      -- Rotacion a la Izq sin Cout
114      Rin<=x"0000000C";
115      Cin<='1';
116      wait for 10 ns;

```

Figura 9. Test ALU desplazamiento Izq, Drch sin/con acarreo.

		Msgs									
	/alu_sim/Control	000	000	001	010		011				
	/alu_sim/Rin	80000000	80000000	0000001F	C0000011	C0000000		0000000C			
	/alu_sim/Sin	80000000	80000000	0000001E							
	/alu_sim/Cin	0									
	/alu_sim/Fout	00000000	00000000		60000008	60000000	40000000	8000000C		FF	
	/alu_sim/Cout	1									
	/alu_sim/Cero	1									

Se puede observar como en el primer caso (recuadro rojo), se lleva a cabo la verificación del desplazamiento hacia la derecha con acarreo de salida; mientras que en el segundo caso del mismo test no existe acarreo de salida. Para el primer caso:

$$Rin = C0000011_{hex} = 1100 \dots 0001 \ 0001_{bin}$$

→ desplazado hacia la derecha sin Cin →

$$Rin = 0110 \dots 0000 \ 1000_{bin} = 60000008_{hex}$$

En el segundo caso (recuadro amarillo), se llevan a cabo dos desplazamientos hacia la izquierda, en el primero se identifica acarreo de salida, mientras que en el segundo se puede apreciar el efecto de la inclusión del acarreo de entrada.

3.3.3. Simulación de las operaciones lógicas.

Por último, y no menos importante, se verifica el correcto comportamiento de las operaciones lógicas NOT, AND, OR y XOR. Para esto se ha elaborado el siguiente test en el archivo de TestBench:

```

53      when "010" => -- Rotación a la Derecha
54          Fout <= Cin & Rin(31 downto 1);
55          Cout <= Rin(0);
56
57      when "011" => -- Rotación a la Izquierda
58          Fout <= Cin & Rin(30 downto 0);
59          Cout <= Rin(31);
60
61      when "100" => -- NOT
62          Fout <= not Rin;
63
64      when "101" => -- AND
65          Fout <= Rin and Sin;
66
67      when "110" => -- OR
68          Fout <= Rin or Sin;
69
70      when "111" => -- XOR
71          Fout <= Rin xor Sin;
72
73      when others => --nothing
74          Fout <= (others => '0');
75
76      end case;
77      end process;
78      -- Fin de la descripción
79  end funcional;

```

Figura 10. Test ALU operaciones lógicas.

	Msgs								
/alu_sim/Control	000	011	100	101	110	111			
/alu_sim/Rin	80000000	0000000C	00000008	0000000C	00000008				
/alu_sim/Sin	80000000	0000001E		00000000	00000004	00000002			
/alu_sim/Cin	0								
/alu_sim/Fout	00000000	8000000C	FFFFFFF7	00000000	0000000C	0000000A			
/alu_sim/Cout	1								
/alu_sim/Cero	1								

De este modo tenemos, en orden de mayor a menor según la señal de control, que:

1. Operación NOT:

$$Rin = 00000008_{hex} \rightarrow Fout = \sim Rin = FFFFFFF7_{hex}$$

2. Operación AND:

$$Fout = Rin \text{ AND } Sin = 0000000C_{hex} \text{ AND } 00000000_{hex} = 00000000_{hex}$$

3. Operación OR:

$$Fout = Rin \text{ OR } Sin = 00000008_{hex} \text{ OR } 00000004_{hex} = 0000000C_{hex}$$

4. Operación XOR:

$$Fout = Rin \text{ XOR } Sin = 00000008_{hex} \text{ XOR } 00000002_{hex} = 0000000A_{hex}$$

4. STACK POINTER.

4.1. Introducción.

En esta parte del proyecto, se llevará a cabo el diseño y simulación del puntero de pila del procesador. Dicho puntero de pila, entendido como un registro de hardware, apunta a más reciente localización de la pila. Éste es un elemento fundamental e imprescindible para el correcto funcionamiento del futuro procesador, ya que dicho puntero de pila o "Stack Pointer (SP)" almacena la próxima dirección de la pila que deberá guardar la dirección de retorno de la subrutina.

Una vez más se ha decidido realizar la implementación usando VHDL y ModelSim debido a la familiaridad adquirida con el lenguaje y dicho software tras la realización de los anteriores entregables.

4.1.1. Descripción del componente.

Como ya se ha comentado el puntero de pila es un registro (referenciado como "registro SP" en los sucesivos puntos del documento) que almacena direcciones de memoria de la pila; dicha pila se encuentra en la memoria externa y ocupa el rango de direcciones proporcionado por los siguientes parámetros (expresiones numéricas en hexadecimal):

$$STACK_BASE_ADDRESS = X"00001100"$$

$$STACK_HIGH_ADDRESS = X"000013FF"$$

Llegados a este punto cabe destacar el error detectado en dichos parámetros presentes en el archivo "configuracion.vhd", donde se observa que el valor del primero (BASE) es mayor que el valor del segundo (HIGH), lo cual obviamente es una simple errata, por lo que se ha decidido asignar un valor de manera arbitraria a STACK_HIGH_ADDRESS.

El registro SP será inicializado con el valor BASE y en cada salto a subrutina el registro será incrementado en una unidad hasta alcanzar el máximo valor posible (STACK_HIGH_ADDRESS), donde se activará la señal "full" la cual inhabilita la posibilidad de incremento. Por el contrario, con cada instrucción de retorno de subrutina, dicho registro será disminuido en una unidad hasta llegar a el mínimo valor posible (STACK_BASE_ADDRESS), donde se activará la señal "empty" la cual inhabilita la posibilidad de disminución.

Este componente será diseñado como si de un contador up/down que permite la carga de datos se tratase. Así, las entradas y salidas de nuestro componente se describen y especifican a continuación:

3. Entradas.

- 3.1. CLK - 1 bit. Reloj del sistema.
- 3.2. Reset - 1 bit. Señal asíncrona de re-inicialización o puesta a cero (A pesar de que en el documento no se especifique que esta señal de entrada sea síncrona o asíncrona, se ha decidido tomar la segunda aproximación puesto que una señal de puesta a cero se deduce que, debido a su comportamiento y a lo que desencadena, no tiene porque depender de la señal de reloj del componente).
- 3.3. Up - 1 bit. Genera la cuenta ascendente del registro ($SP = SP + 1$).
- 3.4. Down - 1 bit. Genera la cuenta descendente del registro ($SP = SP - 1$).
- 3.5. Load - 1 bit. Señal que determina la carga de datos desde el bus "Din" hacia el registro SP.
- 3.6. Din - 32 bits. Bus de datos de entrada al registro.

4. Salidas.

- 4.1. Full - 1 bit. Señal encargada de indicar el momento en el que la pila está llena.
- 4.2. Empty - 1 bit. Indicador de pila vacía.
- 4.3. Dout - 32 bits. Bus de datos de salida del registro.

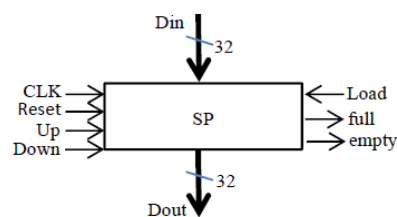


Figura 1. Puertos de entrada/salida del puntero de la pila.

4.2. Diseño del componente mediante VHDL.

En este punto, el código será comentado paso por paso, desde la entidad usada hasta los procesos implementados en el diseño.

En primer lugar, la entidad que se ha codificado es la siguiente:

```

9
10 library IEEE;
11 use IEEE.std_logic_1164.all;
12 use IEEE.std_logic_arith.all;
13 use IEEE.std_logic_unsigned.all;
14
15 use WORK.configuracion.all;
16
17 entity SP is
18   port (CLK : in STD_LOGIC;
19         Reset : in STD_LOGIC;
20         Up : in STD_LOGIC;
21         Down : in STD_LOGIC;
22         Load : in STD_LOGIC;
23         Din : in STD_LOGIC_VECTOR (31 downto 0);
24         Dout : out STD_LOGIC_VECTOR (31 downto 0);
25         full : out STD_LOGIC;
26         empty : out STD_LOGIC);
27 end SP;
28

```

Figura 2. Entidad del componente.

Se pueden distinguir las distintas entradas y salidas ya descritas en el anterior apartado.

Para llevar a cabo la funcionalidad del componente nos hemos servido única y exclusivamente del uso de un proceso y una variable interna en dicho proceso. Esta variable ha sido utilizada para guardar el valor del bus de datos de entrada y, de esta manera, tras realizar las operaciones pertinentes (incrementar o disminuir el valor del registro SP), asignar dicho valor al bus de datos de salida. Esta manera de realizar dicho proceso se ha llevado a cabo de esa forma debido al retardo de actualización de las señales dentro de un proceso en VHDL (hasta que el proceso no acaba no se actualiza el valor de dicha señal, en cambio el valor de una variable se actualiza al instante de operar con ella).

```

29 architecture funcional of SP is
30   -- Definir aquí señales, tipos, funciones, etc
31   -- Fin de definiciones
32 begin
33   -- Iniciar aquí la descripción
34   SP: process ( CLK )
35     variable stackPointer: std_logic_vector(31 downto 0) := STACK_BASE_ADDRESS;
36     begin

```

Figura 3. Variable utilizada durante el proceso SP.

La variable "stackPointer" guarda e indica el valor del registro SP, es sobre esta variable sobre la que se aplican las operaciones de incremento o disminución del valor del registro.

```

32 begin
33   -- Iniciar aquí la descripción
34   SP: process ( CLK )
35     variable stackPointer: std_logic_vector(31 downto 0) := STACK_BASE_ADDRESS;
36     begin
37       if( rising_edge(CLK) and Reset = '0' ) then
38
39         if( Load = '1' ) then
40           -- Operación de carga del SP
41           if( Din >= STACK_BASE_ADDRESS and Din <= STACK_HIGH_ADDRESS ) then
42             stackPointer := Din;
43             Dout <= stackPointer;
44             if( Din = STACK_BASE_ADDRESS ) then
45               full <= '0';
46               empty <= '1';
47             elsif( Din = STACK_HIGH_ADDRESS ) then
48               full <= '1';
49               empty <= '0';
50             else
51               full <= '0';
52               empty <= '0';
53             end if;
54           end if;
55         end if;
56       end if;
57     end if;
58   end process;

```

Figura 4. Operación de carga del registro SP.

A la hora de describir el código de la figura 4, el cual pertenece a la operación de carga desde el bus de datos "Din" hacia el registro SP, cabe destacar el primer filtro aplicado al bus de datos, chequeando que los datos que entran están dentro de los límites establecidos. Y en el caso de que los datos de entrada se encuentren en alguno de los límites de direcciones indicados por `STACK_BASE_ADDRESS` y `STACK_HIGH_ADDRESS`, se asignan a cero o uno las señales pertinentes, `full` y `empty`. También destacar la señal "CLK" incluida dentro de la lista de sensibilidad del proceso.

A continuación, se muestra el código correspondiente a ambas operaciones (incremento y decremento):

```

59   -- Operación de incremento del SP.
60   if( Up = '1' and Down = '0' ) then
61     if( stackPointer = STACK_HIGH_ADDRESS ) then
62       full <= '1';
63     else
64       stackPointer := stackPointer + 1;
65       Dout <= stackPointer;
66       full <= '0';
67       empty <= '0';
68     end if;
69   end if;
70
71   -- Operación de decremento del SP.
72   if( Down = '1' and Up = '0' ) then
73     if( stackPointer = STACK_BASE_ADDRESS ) then
74       empty <= '1';
75     else
76       stackPointer := stackPointer - 1;
77       Dout <= stackPointer;
78       empty <= '0';
79       full <= '0';
80     end if;
81   end if;

```

Figura 5. Operación de incremento y decremento.

Por último podemos analizar el trozo de código correspondiente al reset o puesta a cero del componente, donde el valor del registro SP se actualiza al valor de inicialización:

```

83      elsif( Reset = '1' ) then
84
85          -- El SP es re-inicializado.
86          stackPointer := STACK_BASE_ADDRESS;
87          full <= '0';
88          empty <= '1';
89          Dout <= stackPointer;
90
91      end if;
92  end process;
93
94  -- Fin de la descripción
95  end funcional;

```

Figura 6. Operación de puesta a cero.

4.3. Simulación del componente y creación del TestBench.

En este apartado se analizará el proceso de estímulos realizado para chequear y testear la funcionalidad de nuestro componente. No se explicará en ningún apartado de este documento el diseño del TestBench, ni la creación de la UUT, etc. ya que esto se supone por conocido.

En primer lugar se observa el test realizado para la operación de incremento, decremento y reset:

```

72  --Stimulus process
73  stimulus_process: process
74  begin
75      --wait for clock_period/2;
76      Load <= '1';
77      Din <= x"00001100";
78      wait for clock_period;
79      Load <= '0';
80
81      -- * Operacion de Incremento del SP *
82      -- *****
83      Up <= '1';
84      Down <= '0';
85      wait for 4*clock_period;
86
87      -- *****
88      -- * Operacion de Decremento del SP *
89      -- *****
90      Up <= '0';
91      Down <= '1';
92      wait for 2*clock_period;
93
94      -- *****
95      -- * Operacion de Reset del SP *
96      -- *****
97      Reset <= '1';
98      wait for clock_period;
99      Reset <= '0';
100

```

Figura 7. Test UP, Down y Reset.

Tras realizar estos tres diferentes test, donde se observa la gran parte de la funcionalidad del componente, se realizan otros dos test para observar que el componente cumple las excepciones descritas como no poder incrementar el registro del SP una vez ha llegado a su valor máximo o no poder disminuir el valor de dicho registro una vez llegado a su valor mínimo.

```

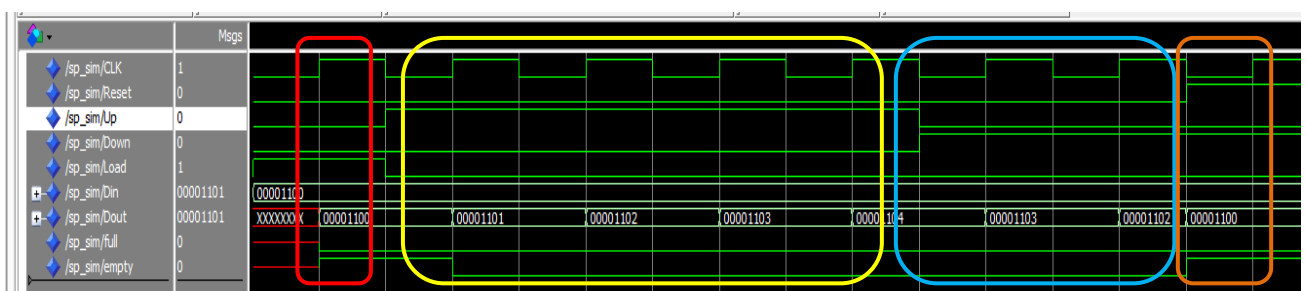
101  -- *****
102  -- *   Maxima dirección del SP   *
103  -- *****
104  Up   <= '0';
105  Down <= '0';
106  Load <= '1';
107  Din  <= x"000013FE";
108  wait for clock_period;
109  Load <= '0';
110  Up   <= '1';
111  Down <= '0';
112  wait for 2*clock_period;
113
114  -- *****
115  -- *   Minima dirección del SP   *
116  -- *****
117  Up   <= '0';
118  Down <= '0';
119  Load <= '1';
120  Din  <= x"00001101";
121  wait for clock_period;
122  Load <= '0';
123  Up   <= '0';
124  Down <= '1';
125  wait for 2*clock_period;
126
127  end process stimulus_process;
128
129  end Bench;

```

Figura 8. Test Máxima y Mínima dirección.

4.4. Simulación y obtención de resultados.

A la hora de plasmar los resultados obtenidos, la mejor manera es usando los "waveforms" que ModelSim genera a la hora de simular el diseño implementado.

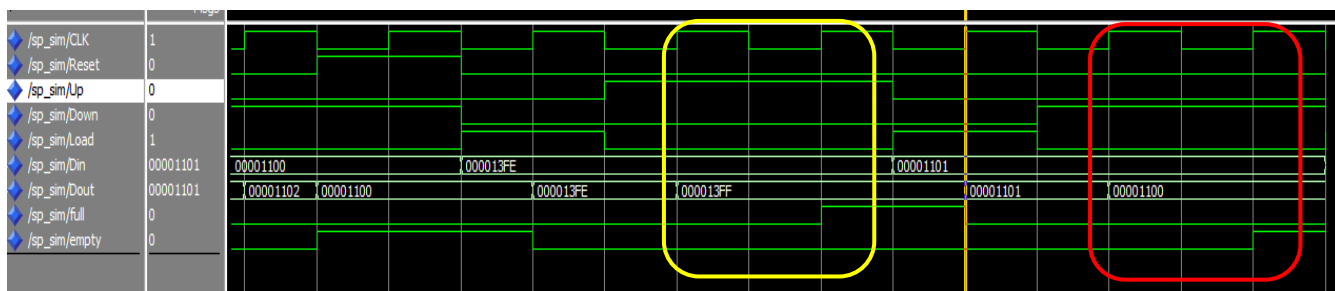


El primer recuadro rojo corresponde con la primera operación llevada a cabo en el TestBench, esta es **la carga inicial de valor en el registro SP poniendo la señal Load a '1' y asignando valor al bus de datos de entrada (En este caso le hemos asignado el valor mínimo por defecto X"00001100")**. Se observa también que al detectar el valor mínimo especificado anteriormente por STACK_BASE_ADDRESS, la señal **empty** se activa.

Durante el recuadro amarillo se puede observar como **la operación de incremento** del puntero de la pila se lleva a cabo correctamente durante los cuatro ciclos de reloj especificados en el Testbench, estando activa la señal **Up** y aumentando el valor de salida **Dout** por cada ciclo de reloj. Seguidamente se aprecia, siendo enmarcado por el recuadro azul, **la operación de decremento** al activarse la señal **Down** y disminuir el valor del bus de datos de salida **Dout**.

La última parte de la simulación, englobada por el recuadro naranja, corresponde a una **secuencia de reset** o puesta a cero del componente; en esta parte vemos que al activarse la señal **Reset** del componente e independientemente de en qué proceso estuviese involucrado, **el registro se inicializa al valor de STACK_BASE_ADDRESS o X"00001100"**.

Tras realizar estas tres diferentes pruebas y ver su correcta funcionalidad, podemos pasar a la simulación y verificación de resultados de los dos test restantes, chequeando las excepciones especificadas en el apartado anterior (no poder incrementar el registro del SP una vez ha llegado a su valor máximo o no poder disminuir el valor de dicho registro una vez llegado a su valor mínimo).



En esta segunda captura de pantalla de la simulación se observa, en el primer recuadro de color amarillo como, tras haberse cargado el registro con el valor X"000013FE" y haber pasado dos ciclos de reloj, **el valor del bus de datos de salida o del registro SP no aumenta más sino que se queda en el máximo: STACK_HIGH_ADDRESS (definido ahora a X"000013FF")**. Por otro lado, en la zona indicada por el recuadro rojo, es totalmente al contrario, ya que al registro se le carga con el valor X"00001101" y se **produce su decremento hasta que llega al mínimo permitido a partir de donde ya no puede disminuir mas**.

5. Registros del procesador.

5.1. Introducción.

En esta parte del proyecto, se llevará a cabo el diseño y simulación de tres registros fundamentales que maneja el procesador. Estos son, el contador de programa, el registro de instrucciones y el registro de estados. El contador de programa es el encargado de almacenar la dirección de memoria que contiene la siguiente instrucción que será ejecutada por el procesador. El registro de instrucciones almacena la instrucción actual que se está ejecutando. Y el registro de estados es una colección de biestables

independientes los cuales manejan determinador estados de ciertas unidades fundamentales del procesador. Una vez más se ha decidido realizar la implementación usando VHDL y ModelSim debido a la familiaridad adquirida con el lenguaje y dicho software tras la realización de los anteriores entregables.

5.2. Contador de programa (PC).

Este registro se inicializa con el valor MEM_BASE_ADDRESS que apunta a la dirección de inicio del programa que se ejecuta; y se incrementa en cada ciclo de búsqueda de la instrucción. Cabe destacar que se ha decidido implementar una característica adicional al componente. Al incrementar dicho contador puede darse el caso de que se llegue al máximo permitido, esto es MEM_HIGH_ADDRESS, lo cual dará lugar a un 'reset' automático del registro, volviendo al valor de inicio del programa. Las entradas y salidas del componente son las siguientes:

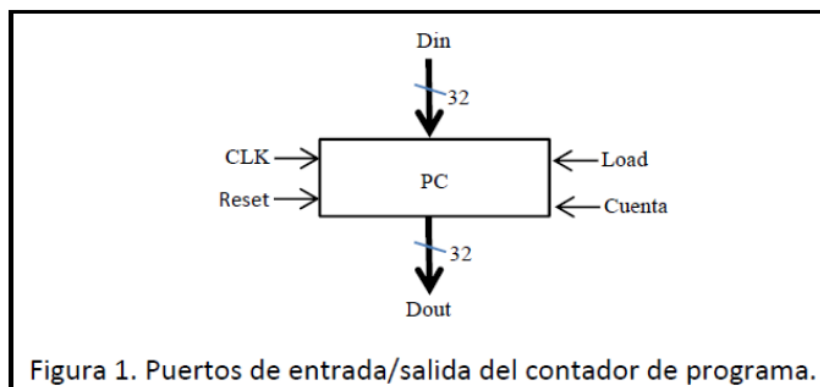


Figura 1. Puertos de entrada/salida del contador de programa.

5.2.2. PC. Diseño del componente mediante VHDL.

En primer lugar, dentro del único proceso implementado, podemos ver las diferentes acciones a llevar a cabo si la señal de 'Reset' ha sido activada o no:

```

PC: process ( CLK )
    variable ProgramCounter: std_logic_vector(31 downto 0) := MEM_
    begin
        if( rising_edge(CLK) and Reset = '0' ) then

            -- Operación de carga del ProgramCounter
            if( Load = '1' ) then
                elsif( Cuenta = '1' ) then
            elsif( Reset = '1' ) then

                -- El ProgramCounter es re-inicializado.
                ProgramCounter := MEM_BASE_ADDRESS;
                Dout <= ProgramCounter;

            end if;
        end process;
    
```

Figura 2. Proceso PC.

Se puede observar como la señal 'Reset' da lugar a la re-inicialización del registro. En el caso de que se desee cargar el contador con una determinada dirección en las instrucciones de salto: Figura 3. Carga del PC.

```

-- Operación de carga del ProgramCounter
if( Load = '1' ) then

    if( Din >= MEM_BASE_ADDRESS and Din <= MEM_HIGH_ADDRESS ) then
        ProgramCounter := Din;
        Dout <= ProgramCounter;
    end if;

```

Figura 3. Carga del PC.

Se chequea que la entrada esté entre los límites permitidos y se carga el registro con dicho valor. Para el incremento del contador se usa la señal 'Cuenta':

```

elseif( Cuenta = '1' ) then
    ProgramCounter := ProgramCounter + 1;

    -- En el caso en el que el contador de programa
    -- llegue al máximo de direcciones posible, se
    -- resetea automáticamente ( característica implementada
    -- adicionalmente por el usuario, no especificada en el
    -- documento ).
    if ( ProgramCounter > MEM_HIGH_ADDRESS ) then
        ProgramCounter := MEM_BASE_ADDRESS;
    end if;

    Dout <= ProgramCounter;
end if;

```

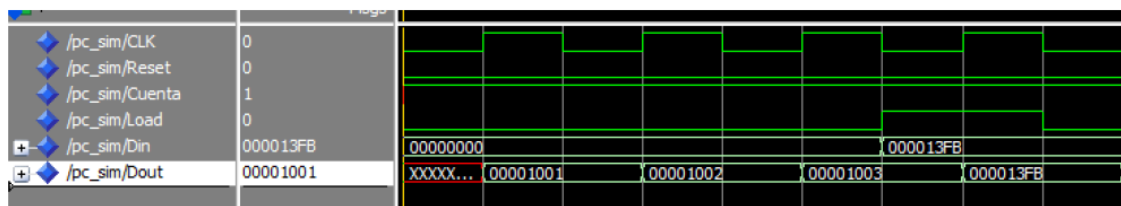
Figura 4 . Incremento del PC.

En la figura 4 se observa cómo se incrementa el contador de programa y se añade la característica adicional ya comentada, detectando de esta manera el límite superior permitido y re-inicializando en el caso de llegar a dicho límite.

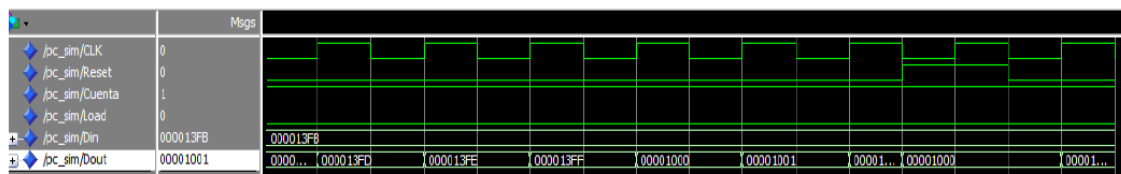
5.2.3. PC. Testbench y simulación.

En el testbench implementado se destaca principalmente el proceso de estímulos:

En este proceso se lleva a cabo tres diferentes verificaciones del sistema; en primer lugar, el incremento del PC durante tres ciclos de reloj, en segundo lugar la carga del registro activando la señal 'Load' y en tercero y último lugar la activación de la señal reset. Estas tres partes demostrarán y verificarán la funcionalidad del diseño:



En esta primera simulación se observa el funcionamiento correcto del contador incrementándose por cada ciclo de reloj y la carga de valor al activarse la señal 'Load'.



Y en esta segunda simulación se puede distinguir como al llegar al máximo permitido, el registro se re-inicializa (característica adicional añadida) y **como al activarse la señal 'Reset' el registro toma el valor inicial.**

5.3. Registro de Instrucciones (RI).

Este registro es más simple que el anterior; simplemente guarda la dirección de memoria de la instrucción que se está ejecutando en el momento. El dato de entrada se almacena en el registro al activarse la señal 'Load'. Estas son las entradas y salidas de dicho registro:

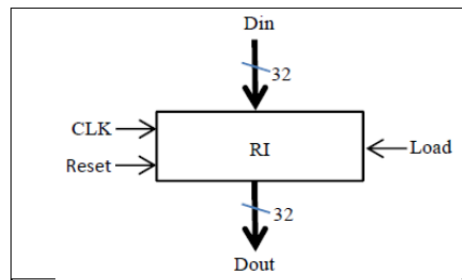


Figura 6. Entradas/salidas del RI.

5.3.2. RI. Diseño del componente mediante VHDL.

El diseño del RI es mucho más simple que el anterior registro; se puede mostrar en una única imagen:

```

FC: process ( CLK )
begin
    if( rising_edge(CLK) and Reset = '0' ) then

        -- Operación de carga del RI
        if( Load = '1' ) then

            if( Din >= MEM_BASE_ADDRESS and Din <= MEM_HIGH_ADDRESS ) then
                Dout <= Din;
            end if;
        end if;

        elsif( Reset = '1' ) then

            -- El RI es re-inicializado.
            Dout <= MEM_BASE_ADDRESS;

        end if;
    end process;
-- Fin de la descripción
end funcional;

```

Figura 7. Proceso del RI.

Se observa en la figura 7 como se trata la carga del registro y su re-inicialización. En este caso no hacen falta variables intermedias puesto que no existe operación alguna con el valor del registro.

5.3.3. RI. Testbench y simulación.

El testbench utilizado es el que aparece en la figura 8:

```

--Stimulus process
stimulus_process: process
begin
    -- Carga del registro
    Load <= '1';
    Din <= x"000013FB";
    wait for clock_period;
    Load <= '0';

    wait for clock_period*2;

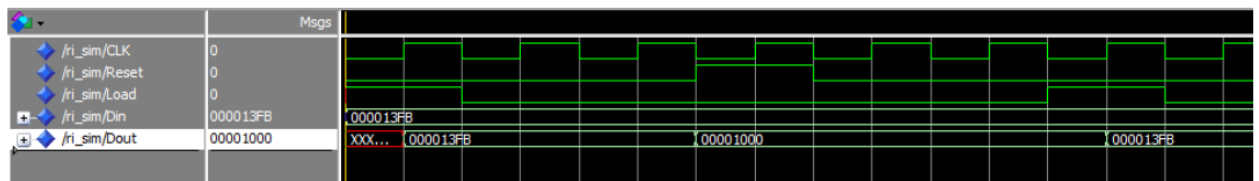
    -- Carga del registro y reset
    Reset <= '1';
    wait for clock_period;
    Reset <= '0';

    wait for clock_period*2;
end process stimulus_process;

```

Figura 8. Stimulus process Testbench.

Se distinguen las dos diferentes pruebas: la carga del registro y la activación de la señal reset:



5.4. Registro de estados (STATUS).

Este registro de estados está formado de cuatro biestables independientes que solo comparten la señal de reloj y reset. El esquemático de entradas y salidas sería el siguiente:

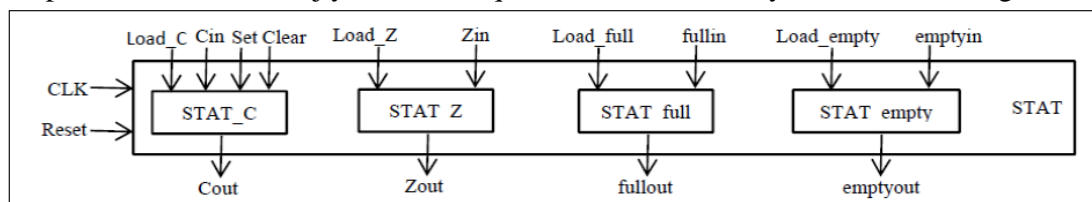


Figura 9. Entradas/Salidas registro STATUS.

5.4.2. STATUS. Diseño del componente mediante VHDL.

Para cada biestable crearemos un proceso independiente:

```

STAT_C: process ( CLK )
begin
    if( rising_edge(CLK) and Reset = '0' ) then
        -- Operación de escritura del biestable STAT_C
        if( STAT_C_load = '1' ) then
            if( STAT_C_Set = '1' and STAT_C_Clear = '0' ) then
                STAT_C_out <= '1';
            elsif( STAT_C_Clear = '1' and STAT_C_Set = '0' ) then
                STAT_C_out <= '0';
            else
                STAT_C_out <= STAT_C_in;
            end if;
        end if;
    elsif( Reset = '1' ) then
        STAT_C_out <= '0';
    end if;
end process;

STAT_Z: process ( CLK )
begin
    if( rising_edge(CLK) and Reset = '0' ) then
        -- Operación de escritura del biestable STAT_Z
        if( STAT_Z_load = '1' ) then
            STAT_Z_out <= STAT_Z_in;
        end if;
    elsif( Reset = '1' ) then
        STAT_Z_out <= '0';
    end if;
end process;

STAT_FULL: process ( CLK )
begin
    if( rising_edge(CLK) and Reset = '0' ) then
        -- Operación de escritura del biestable STAT_FULL
        if( STAT_SPF_load = '1' ) then
            STAT_SPF_out <= STAT_SPF_in;
        end if;
    elsif( Reset = '1' ) then
        STAT_SPF_out <= '0';
    end if;
end process;

STAT_EMPTY: process ( CLK )
begin
    if( rising_edge(CLK) and Reset = '0' ) then
        -- Operación de escritura del biestable STAT_FULL
        if( STAT_SPE_load = '1' ) then
            STAT_SPE_out <= STAT_SPE_in;
        end if;
    elsif( Reset = '1' ) then
        STAT_SPE_out <= '0';
    end if;
end process;

```

Figura 10. Biestables C, Z, SPF y SPE.

Cabe destacar que los cuatro procesos se podrían haber englobado en uno solo, pero por claridad y organización se ha decidido realizar un proceso para cada biestable. En cada proceso se trata la operación de escritura para cada biestable siempre y cuando la señal de escritura particular de cada uno esté activada y teniendo en cuenta otras posibles señales como 'set' y 'clear' en el caso del primer biestable.

5.4.3. STATUS. Testbench y simulación.

El testbench implementado es sencillo, llevando a cabo verificaciones de las distintas señales de los biestables y del reset general del registro.


```

--Stimulus process
stimulus_process: process
begin
    --STAT_C
    -- Operación de escritura del biestable
    STAT_C_in <= '1';
    STAT_C_load <= '1';
    wait for clock_period;

    -- Operación de 'clear' del biestable
    STAT_C_Clear <= '1';
    wait for clock_period;
    STAT_C_Clear <= '0';

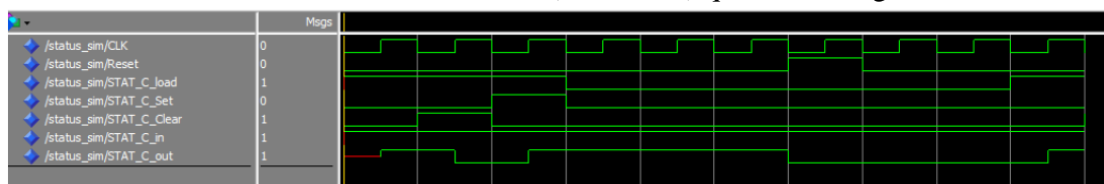
    -- Operación de 'set' del biestable
    STAT_C_Set <= '1';
    wait for clock_period;
    STAT_C_Set <= '0';

    STAT_C_load <= '0';

```

Figura 11. Testbench STAT_C.

En este caso la simulación de dicho biestable (STAT_C) queda de la siguiente manera:



Se observa cómo se carga en primera instancia el dato observado en C_in gracias a la activación de la señal C_load; seguidamente se activa la señal C_Clear que vuelve a poner a cero el biestable y en el siguiente flanco de subida se activa C_Set que finalmente pone a uno dicho biestable. Al producirse el 'Reset' se inicializa el biestable observándose un cero.

Para el biestable STAT_Z:

```

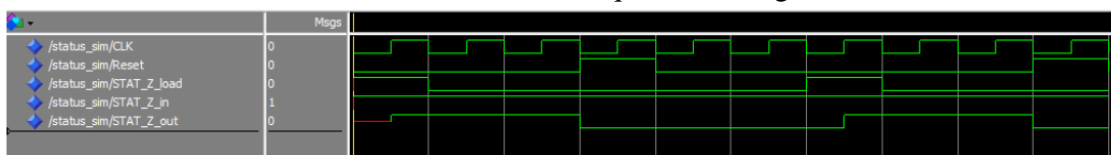
--STAT_Z
-- Operación de escritura del biestable
STAT_Z_in <= '1';
STAT_Z_load <= '1';
wait for clock_period;

STAT_Z_load <= '0';

```

Figura 12. Testbench STAT_Z.

En este caso la simulación del biestable STAT_Z queda de la siguiente manera:



Se pueden sacar las mismas conclusiones que en el biestable anterior. Finalmente para los biestables del stackpointer: STAT_F y STAT_E.

```

--STAT_SPF
-- Operación de escritura del biestable
STAT_SPF_in <= '1';
STAT_SPF_load <= '1';
wait for clock_period;

STAT_SPF_load <= '0';

--STAT_SPE
-- Operación de escritura del biestable
STAT_SPE_in <= '1';
STAT_SPE_load <= '1';
wait for clock_period;

STAT_SPE_load <= '0';

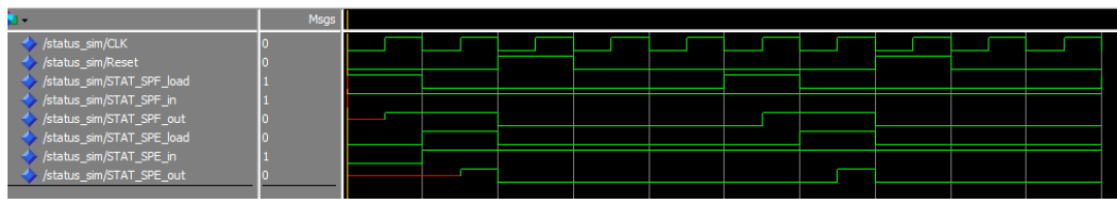
-- Reset de los biestables
Reset <= '1';
wait for clock_period;
Reset <= '0';

wait for clock_period*2;

```

Figura 13. Testbench STAT_F & E.

Cabe destacar que la operación de 'Reset' que se muestra en la figura 13 se ha aplicado para los cuatro biestables por igual dentro del mismo proceso stimulus (como se puede apreciar en las imágenes de simulación).



6. Ruta de datos del sistema completo.

6.1. Introducción.

En esta parte del proyecto, se llevará a cabo el diseño de la ruta de datos del procesador o inter-conexionado entre los distintos bloques del sistema, apoyándonos en multiplexores realizaremos el conexionado entre el registro de instrucciones, el puntero de pila, el contador de programa, el fichero de registros, la unidad aritmético lógica y el registro de estados. Las distintas señales de control que puedan gobernar los bloques descritos serán generadas por la unidad de control, la cual se implementará en el siguiente entregable.

Una vez más se ha decidido realizar la implementación usando VHDL y ModelSim debido a la familiaridad adquirida con el lenguaje y dicho software tras la realización de los anteriores entregables.

6.1.1. Descripción del componente.

Como se ha comentado en el punto anterior, las señales de control que gobiernan los distintos componentes o bloques de nuestro sistema serán implementadas en siguiente entregable, siendo este el diseño de la unidad de control.

La siguiente imagen muestra la conexión entre los distintos componentes:

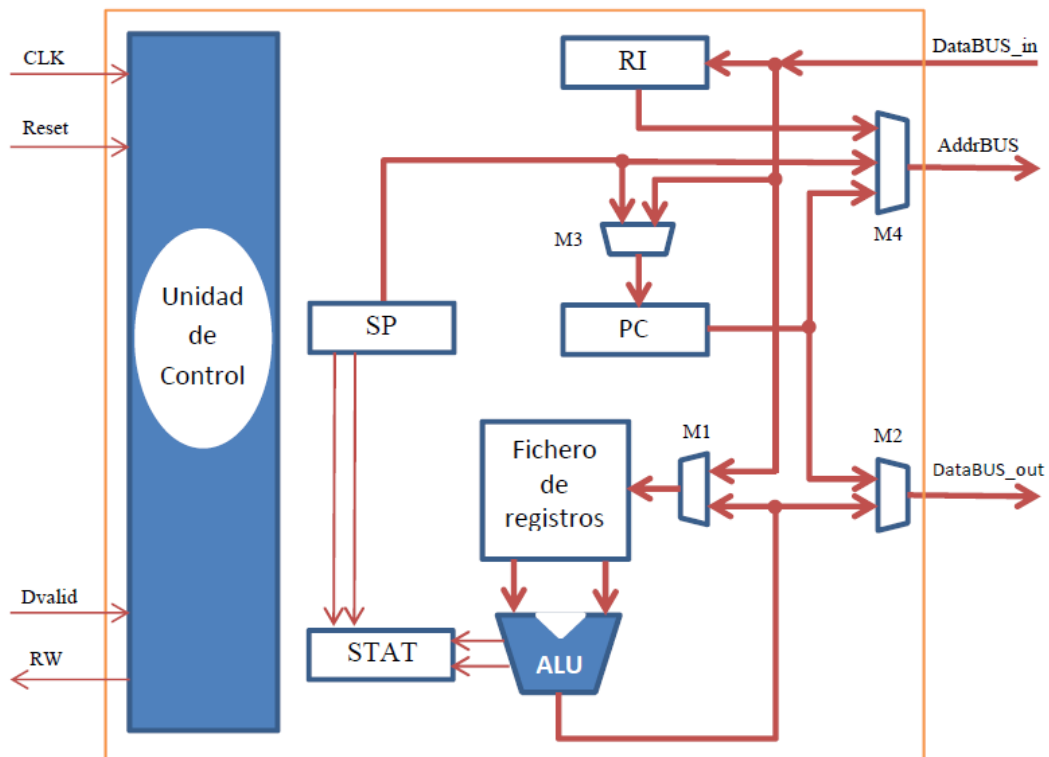


Figura 1. Esquema de bloques del procesador PSM

Nos ayudaremos de los proyectos realizados en entregables anteriores, realizando los componentes de estos y su respectivo "port map" con las señales correspondientes. Usaremos también señales temporales para guardar ciertas señales como las salidas de los multiplexores etc. Además de los componentes de entregables anteriores (puntero de pila, fichero de registros, etc.), diseñaremos dos nuevos componentes, un mux2to1 y un mux3to1 (este último es en realidad un mux4to1 con una de las entradas desconectadas).

Las entradas y salidas del sistema son las especificadas en la siguiente tabla:

Tabla 1. Señales de entrada y salida del procesador PSM

Señal	Tipo	Anchura	Descripción
CLK	entrada	1 bit	Señal de reloj global del sistema
Reset	entrada	1 bit	Reset o puesta a cero global del sistema
Dvalid	entrada	1 bit	Indicador de que el dato en el bus de dato es válido
WR_enable	salida	1 bit	Control de escritura
DATABUS_in	entrada	32 bits	Bus de datos de entrada
DATABUS_out	salida	32 bits	Bus de datos de salida
ADDRBUS	salida	32 bits	Bus de direcciones

También existen señales de control responsables del control de los multiplexores:

Tabla 2. Señales de control de los multiplexores

Señal	Tipo	Anchura	Descripción
control_Reg_Din	M1	1 bit	Control del bus de entrada del fichero de registros (Reg_Din). Selecciona entre el bus de datos (DataBUS_in) y la salida de la ALU.
control_DATABUS_out	M2	1 bit	Control del bus de salida (DATABUS_out). Selecciona entre una de los buses de salida del fichero de registros (Reg_Dout_A) y la salida del contador de programa (PC_Dout).
control_PC_Din	M3	1 bit	Control del bus de entrada del contador de programa (PC_Din). Selecciona entre el bus de datos de entrada (DataBUS_in) y el campo dir del registro de instrucciones.
control_ADDRBUS	M4	2 bits	Control del bus de direcciones (ADDRBUS). (Reg_Din). Selecciona entre la salida del contador de programa (PC_Dout), la salida del puntero de la pila (SP_Dout) y el campo dir del registro de instrucciones.

6.2. Diseño del componente mediante VHDL.

En este punto, el código será comentado paso por paso, desde la entidad usada hasta los procesos implementados en el diseño.

En primer lugar, la entidad que se ha codificado es la siguiente:

```

entity PSM is
  port (CLK      : in  STD_LOGIC;
        Reset    : in  STD_LOGIC;
        WR_enable : out STD_LOGIC;
        dvalid    : in  STD_LOGIC;
        -- buses externos
        DATABUS_in : in  STD_LOGIC_VECTOR (31 downto 0);
        DATABUS_out : out STD_LOGIC_VECTOR (31 downto 0);
        ADDRBUS    : out STD_LOGIC_VECTOR (31 downto 0));
end PSM;
```

Figura 2. Entidad del componente.

Se pueden distinguir las distintas entradas y salidas ya descritas en el anterior apartado.

Para llevar a cabo el conexionado entre los distintos componentes nos hemos servido de varias señales temporales y los multiplexores ya descritos:

```

--component for mux2to1
component mux2to1 is
  port ( A_IN      : in std_logic_vector( 31 downto 0 );
        B_IN      : in std_logic_vector( 31 downto 0 );
        CONT_SIG   : in std_logic;
        OUT_SIG    : out std_logic_vector( 31 downto 0 ) );
end component;

--component for mux3to1
component mux3to1 is
  port ( A_IN      : in std_logic_vector( 31 downto 0 );
        B_IN      : in std_logic_vector( 31 downto 0 );
        C_IN      : in std_logic_vector( 31 downto 0 );
        CONT_SIG   : in std_logic_vector(1 downto 0);
        OUT_SIG    : out std_logic_vector( 31 downto 0 ) );
end component;

```

Figura 3. Componentes mux2to1 y mux3to1.

Los multiplexores usados siguen la implementación descrita a continuación:

<pre> Architecture behavioral of mux3to1 is begin Process(CONT_SIG,A_IN,B_IN,C_IN) variable temp:std_logic_vector(31 downto 0); Begin case CONT_SIG is when "00" => temp:=A_IN; when "01" => temp:=B_IN; when "10" => temp:=C_IN; when Others => temp:=temp; end case; OUT_SIG<=temp; end Process; end behavioral; </pre>	<pre> --Architecture of the multiplexer architecture RTL of mux2to1 is begin --DISP_MUX process DISP_MUX: process begin wait on A_IN,B_IN,CONT_SIG; if CONT_SIG = '1' then OUT_SIG <= A_IN; else OUT_SIG <= B_IN; end if; end process; end RTL; </pre>
--	--

Figura 4. Funcionalidad mux3to1 y mux2to1.

Dicha arquitectura es bastante sencilla y fácil de comprender.

Las señales temporales de entrada y/o salida para los distintos multiplexores y componentes son:

```

--señales de control
signal control_reg_din:      std_logic := '0';
signal control_DATABUS_out:  std_logic := '0';
signal control_PC_Din:      std_logic := '0';
signal control_ADDRBUS:     std_logic_vector(1 downto 0) := (others => '0');

--señales temporales de entrada
signal temp_in_fileReg:     std_logic_vector( 31 downto 0 ) := (others => '0'); --Entrada Fichero registros
signal temp_in_pc:         std_logic_vector( 31 downto 0 ) := (others => '0'); --Entrada PC

--señales temporales de salida
signal temp_out_alu:        std_logic_vector( 31 downto 0 ) := (others => '0'); -- Salida ALU
signal temp_out_alu_cout:   std_logic := '0'; -- Salida ALU acarreo
signal temp_out_alu_cero:   std_logic := '0'; -- Salida ALU indicador de cero
signal temp_out_pc:         std_logic_vector( 31 downto 0 ) := (others => '0'); -- Salida PC
signal temp_out_sp:         std_logic_vector( 31 downto 0 ) := (others => '0'); -- Salida out SP
signal temp_out_sp_empty:   std_logic := '0'; -- Salida empty SP
signal temp_out_sp_full:    std_logic := '0'; -- Salida full SP
signal temp_out_ri:         std_logic_vector( 31 downto 0 ) := (others => '0'); -- Salida RI
signal temp_out_fileReg_A:  std_logic_vector( 31 downto 0 ) := (others => '0'); -- Salida A Fichero de registros
signal temp_out_fileReg_B:  std_logic_vector( 31 downto 0 ) := (others => '0'); -- Salida B Fichero de registros

```

Figura 4. Señales temporales utilizadas.

También se puede observar la declaración de las señales de control para los multiplexores. Estas señales temporales servirán en el sistema como si de los cables se trataran, conectando unos componentes con otros y dando un sentido ordenado y lógico a la asignación posterior de señales dentro de la arquitectura.

Una vez declarado los distintos componentes a usar y las distintas señales de conexionado, pasamos a realizar el mapeo de las señales dentro de la arquitectura señalada:

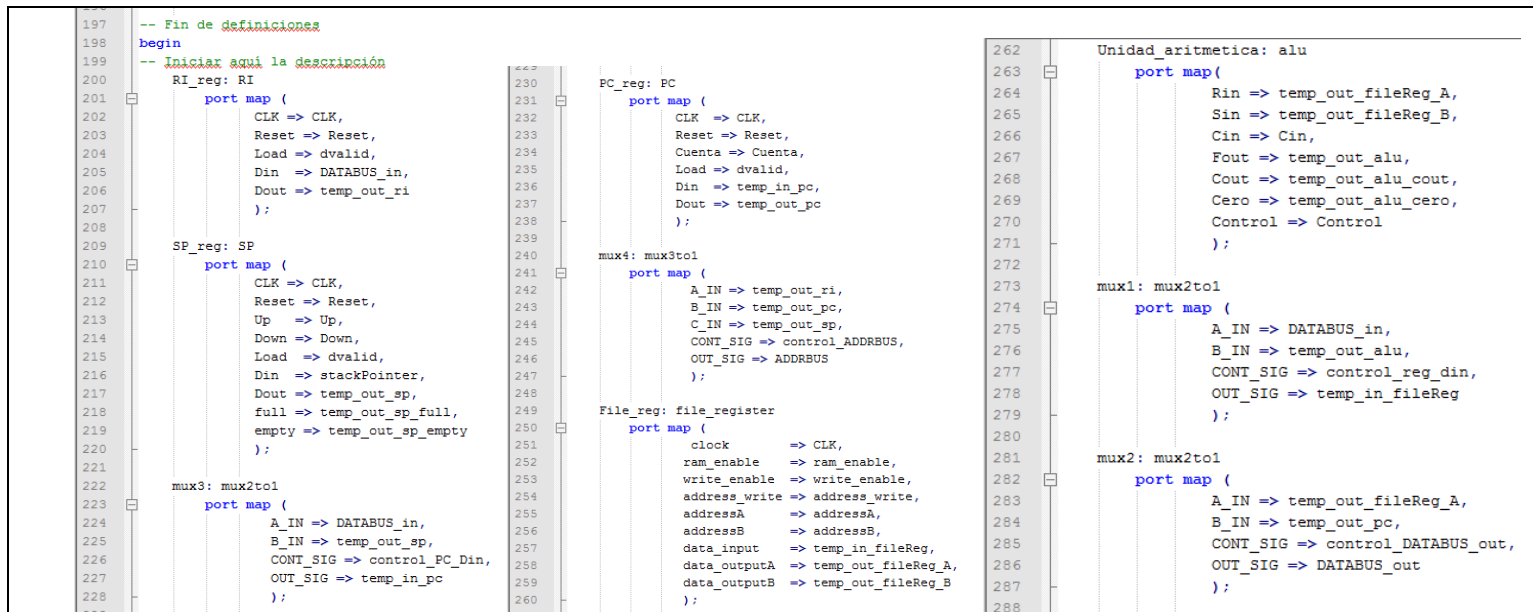


Figura 5. Signal Mapping.

Estos son algunos de los 'port mapping' realizados, cabe destacar que las señales de control específicas de ciertos componentes se han decidido declarar momentáneamente como señales temporales para no provocar errores de compilación (recordar que estas señales de control son generadas por la unidad de control la cual será implementada en el siguiente entregable).

De gran importancia es también observar que las señales de los multiplexores estén correctamente asignadas:

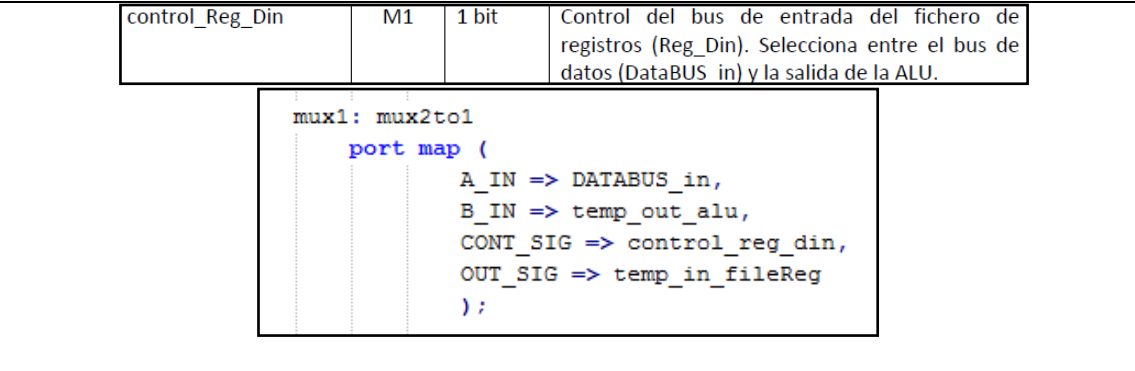


Figura 6. Correcto mapping de las señales.

Y de esta manera se puede chequear para los demás multiplexores.

Resaltar también que aunque en la figura 1 parezca que una de las entradas del M1 es igual que otra del M2, esto no es así; es una pequeña errata que se resuelve guiándose por la tabla de señales de control y no por la figura, puesto que en la figura parece que la salida de la ALU se conecta a dicha entrada pero es lógico pensar que un bloque asíncrono como la unidad aritmético lógica no se conectará jamás a un bus con retrasos como el DATABus_out.

7. Unidad de control y Simulación final.

Finalmente, tras realizar el conexionado y llevar a cabo el mapeado de los diferentes componentes del sistema, obtenemos el siguiente bloque de componentes:

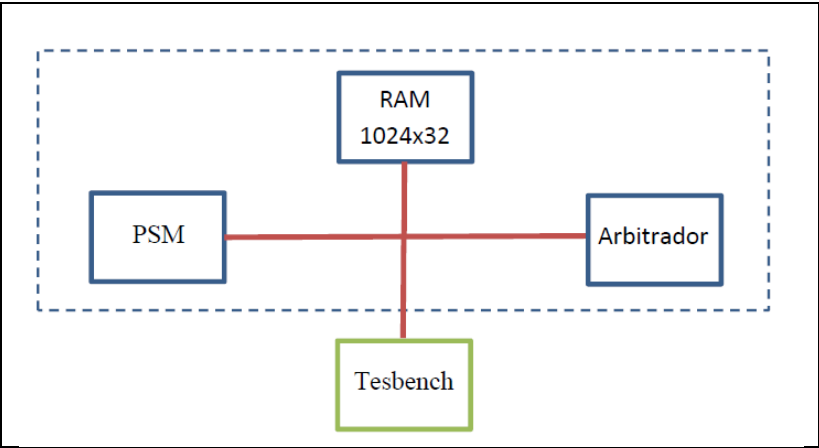


Figura 1. Diagrama de bloques conexionado.

Es crucial destacar el juego de instrucciones, pues será lo que nos proporcione la funcionalidad requerida y deseada a nuestro procesador:

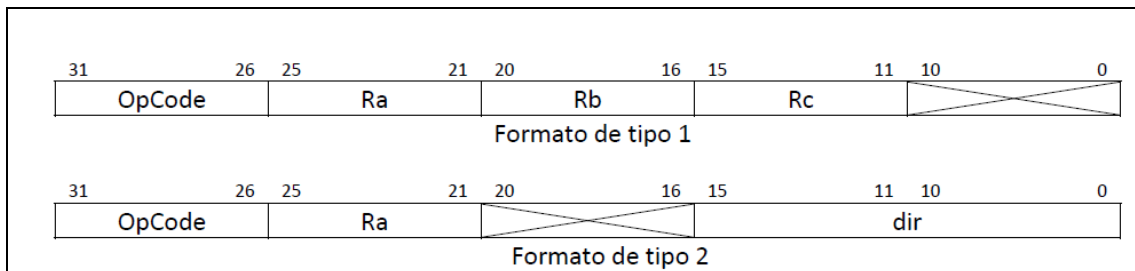


Figura 2. Formato del juego de instrucciones.

El formato de tipo 1 corresponde a instrucciones que requieren 3 operandos. Son instrucciones que hacen referencia a operaciones con la ALU. En este caso los bits 25 al 21 indican el registro destino, los dos campos siguientes (bits 20 a 16 y bits 15 a 11) indican la dirección de los registros que contienen los datos de entrada.

El formato de tipo 2 corresponde a instrucciones que requieren una referencia a un registro interno y la dirección que contiene el dato. Dicha dirección es de 16 bits (bits 15 a 0).

El juego de instrucciones puede agruparse en 4 conjuntos:

- Instrucciones aritmético-lógica-desplazamiento.
- Instrucciones de transferencias de datos (entre registros internos y memoria externa)
- Instrucciones de salto o control de flujo
- Instrucciones de control

La siguiente tabla muestra el conjunto de instrucciones. En dicha tabla se indica el mnemónico, el código de operación de la instrucción, la función que realiza y una breve descripción.

Tabla 9. Juego de instrucciones del procesador PSM

Mnemónico	Código	Tipo	Operación	Descripción
SUM Ra Rb Rc;	000000	Aritmética	$Ra \leftarrow Rb + Rc$	Suma
SUB Ra Rb Rc;	000001	Aritmética	$Ra \leftarrow Rb - Rc$	Resta
SHR Ra Rb;	000010	Desplazam.	$Ra \leftarrow \{Rb \gg 1, STAT_C\}$	Desplazamiento a la derecha
SHL Ra Rb;	000011	Desplazam.	$Ra \leftarrow \{STAT_C, Rb \ll 1\}$	Desplazamiento a la izquierda
NOT Ra Rb;	000100	Lógica	$Ra \leftarrow \text{not } Rb$	Not
AND Ra Rb Rc;	000101	Lógica	$Ra \leftarrow Rb \text{ and } Rc$	And
OR Ra Rb Rc;	000110	Lógica	$Ra \leftarrow Rb \text{ or } Rc$	Or
XOR Ra Rb Rc;	000111	Lógica	$Ra \leftarrow Rb \text{ xor } Rc$	Xor
LD Ra dir;	001000	Transferenc.	$Ra \leftarrow \text{Mem}(\text{dir})$	Carga registro desde memoria
ST dir Rb;	001001	Transferenc.	$\text{Mem}(\text{dir}) \leftarrow Rb$	Almacenamiento en memoria
JMP dir;	001010	Salto	$PC \leftarrow \text{dir}$	Salto incondicional
JPC dir;	001011	Salto	$\text{If } (C) \text{ } PC \leftarrow \text{dir}$	Salto condicional
JSR dir;	001100	Salto	$S \leftarrow SP + 1$ $\text{Mem}(SP) \leftarrow PC$ $PC \leftarrow \text{dir}$	Salto a subrutina
RST;	001101	Salto	$PC \leftarrow \text{Mem}(SP)$ $SP \leftarrow SP - 1$	Retorno de subrutina
CLC;	001110	Control	$STAT_C \leftarrow 0$	Puesta a cero del acarreo

7.1. Manejo de las señales de control.

En este apartado se explicará paso por paso el uso de las diferentes señales de control en la unidad de control para llevar a cabo todas y cada una de las instrucciones posibles.

Centrándonos primero en el archivo 'PSM.vhd', ya explicado en el sexto apartado de esta memoria; se puede observar la inclusión de un nuevo componente, la unidad de control:

```
--component for UC (Unit Control)
component Control is
  port (CLK          : in  STD_LOGIC;
        Reset        : in  STD_LOGIC;
        -- control externo
        WR_enable     : out  STD_LOGIC;
        dvalid        : in  STD_LOGIC;
        -- control de buses
        control_DATABUS_out: out std_logic; -- Control de bus externo de datos de salida
        control_ADDRBUS   : out std_logic_vector(1 downto 0); -- Control de bus de externo de direccion
        control_PC_Din    : out std_logic; -- control del bus de entrada del PC
        control_Reg_Din   : out std_logic; -- control del bus de entrada del fichero de registros
        -- control del contador de programa
        PC_cuenta: out STD_LOGIC;
        PC_load  : out STD_LOGIC;
        -- control del fichero de registros de proposito general
        Reg_enable: out STD_LOGIC;
        Reg_write  : out STD_LOGIC;
        Reg_address_WR: out std_logic_vector (4 downto 0);
        Reg_addr_A: out std_logic_vector (4 downto 0);
        Reg_addr_B: out std_logic_vector (4 downto 0);
        -- control de la ALU
        ALU_control: out std_logic_vector (2 downto 0); -- control de operacion de la ALU
        -- control del registro de instrucciones (RI: registro)
        RI_load: out STD_LOGIC;
        RI_Dout: in STD_LOGIC_VECTOR (31 downto 0);
        -- control del stack pointer (USP: SP)
        SP_up  : out STD_LOGIC;
        SP_down: out STD_LOGIC;
        SP_load: out STD_LOGIC;
        -- control del registro de estado
        STAT_C : in std_logic;
        STAT_C_load: out std_logic;
```

Figura 3. Componente unidad de control archivo PSM.vhd.

Las conexiones de las señales de control pertenecientes a los multiplexores del sistema vienen definidas por la siguiente tabla:

Tabla 2. Señales de control de los multiplexores			
Señal	Tipo	Anchura	Descripción
control_Reg_Din	M1	1 bit	Control del bus de entrada del fichero de registros (Reg_Din). Selecciona entre el bus de datos (DataBUS_in) y la salida de la ALU.
control_DATABUS_out	M2	1 bit	Control del bus de salida (DATABUS_out). Selecciona entre una de los buses de salida del fichero de registros (Reg_Dout_A) y la salida del contador de programa (PC_Dout).
control_PC_Din	M3	1 bit	Control del bus de entrada del contador de programa (PC_Din). Selecciona entre el bus de datos de entrada (DataBUS_in) y el campo dir del registro de instrucciones.
control_ADDRBUS	M4	2 bits	Control del bus de direcciones (ADDRBUS). (Reg_Din). Selecciona entre la salida del contador de programa (PC_Dout), la salida del puntero de la pila (SP_Dout) y el campo dir del registro de instrucciones.

Las demás señales de control van conectadas a los respectivos componentes indicados en los comentarios: "control del contador de programa" PC_Cuenta, PC_Load, etc.

Cabe destacar cierta conexión que no se tuvo en cuenta en el sexto apartado de esta memoria, esto es lo que se especifica en la tabla anterior como campo 'dir' del registro de instrucciones, este campo, como vemos en el formato de instrucciones, son los 15 primeros bits de la instrucción en ejecución. De esta manera, en el PC se almacenarán las direcciones de las instrucciones del programa ejecutado (teniendo en cuenta el inicio de memoria: MEM_BASE_ADDRESS):

```
dir <= (x"0000" & temp_out_ri(15 downto 0)) + MEM_BASE_ADDRESS;
```

Una vez comentado brevemente los cambios efectuados en el archivo PSM.vhd; pasamos a comentar las modificaciones realizadas en el archivo Control.vhd donde se da valor a todas y cada una de las señales de control del sistema.

En primer lugar la primera modificación se observa al haber cambiado el primer proceso en cuanto a la detección del flanco de reloj, puesto que por defecto era de bajada; y ya que **todos los diseños realizados anteriormente han sido con detección de flanco de subida**, el de la unidad de control también debe ser de esta manera para evitar problemas de simulación.

La **máquina de estados que define las transiciones entre unas instrucciones u otras** se define dentro del fichero Control.vhd, como ejemplo de máquina de estados del procesador podemos mostrar el siguiente esquemático:

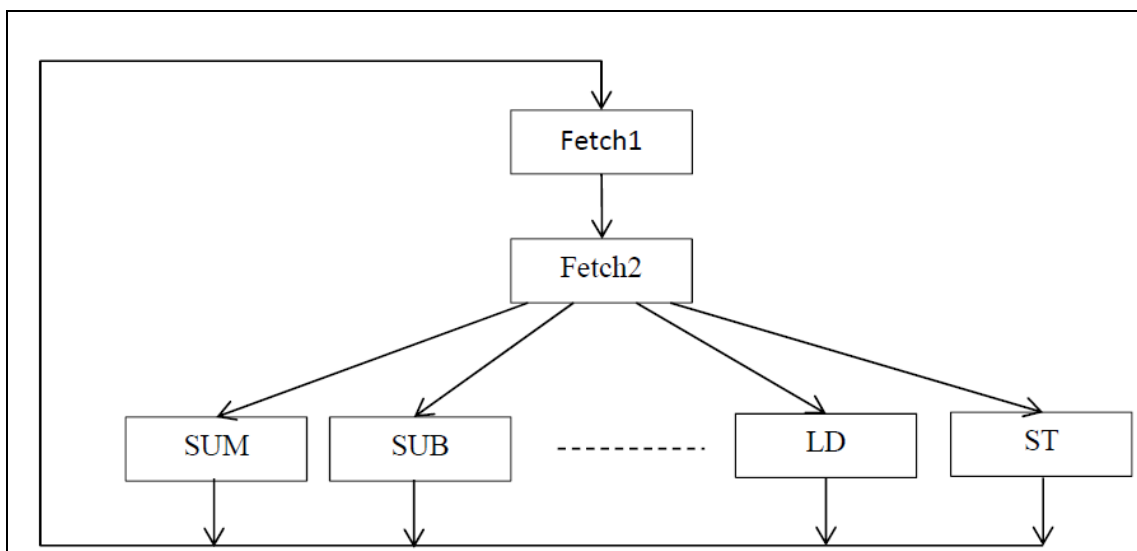


Figura 4. Máquina de estados procesador. Ejemplo ilustrativo.

A continuación pasaremos a comentar uno a uno los diferentes estados de dicha máquina, cabe destacar que previo a la iniciación de la máquina de estados, durante el reset, los registros pertinentes (SP,PC,etc) son cargados con sus respectivos valores por defecto.

7.1.2. FETCH1.

Primer ciclo de búsqueda de la instrucción.

Tras haberse almacenado el programa en memoria satisfactoriamente, se carga en el registro de instrucciones la primera instrucción y se incrementa el contador de programa a uno. Al incrementar aquí el contador de programa se hace innecesario incrementarlo en cada una de las sucesivas instrucciones; de tal manera que solo necesitamos incrementarlo durante el ciclo de búsqueda de instrucción.

$$PC_{Cuenta} = 1; \quad RI_{load} = 1$$

El resto de señales permanecen a cero.

```

case estado_actual is
when FETCH1 => -- ciclo 1 de busqueda de instrucción
-- control externo
WR_enable <= '0';
-- control de buses
control_DATABUS_out <= '0';
control_ADDRBUS <= "000";
control_PC_Din <= '0';
control_Reg_Din <= '0';
-- control del contador de programa
PC_cuenta <= '1'; PC_load <= '0';
-- control del fichero de registros de proposito general
Reg_enable <= '0';
Reg_write <= '0';
Reg_address_WR <= (others=>'0');
Reg_addr_A <= (others=>'0');
Reg_addr_B <= (others=>'0');
-- control de la ALU
ALU_control <= "000"; -- control de operacion de la ALU
-- control del registro de instrucciones (RI: registro)
RI_load <= '1';
-- control del stack pointer (USP: SP)
SP_up <= '0'; SP_down <= '0'; SP_load <= '0';
-- control del registro de estado
STAT_C_load <= '0'; STAT_Z_load <= '0';
STAT_C_Set <= '0'; STAT_C_Clear <= '0';
STAT_SPF_load <= '0'; STAT_SPE_load <= '0';

proximo_estado <= FETCH2;

```

Figura 5. Fetch1.

7.1.2. FETCH2.

Segundo ciclo de búsqueda de instrucción.

En este ciclo se decodifica la instrucción identificada en el ciclo anterior, para ello se usa del bit 30 al bit 26 de la instrucción almacenada en el registro de instrucciones; seleccionando de esta manera el tipo de instrucción que realizará el procesador. Cabe destacar en este ciclo la habilitación del fichero de registros y la asignación de la fuente A y B, siendo estas las mismas que se han indicado en la figura 2 donde se ilustra el

formato de instrucciones (Source A es Rb y Source C es Rc). El motivo de esta habilitación y asignación prematura de fuentes, se debe al simple hecho de tener cargado en el fichero de registros los datos pertinentes a instrucciones algorítmicas (suma, resta, etc), puesto que de no hacer esto observaríamos como la ALU, cuyas entradas están conectadas a las salidas del fichero de registros y su salida a la entrada del mismo fichero de registros, no es capaz de localizar dichas entradas ya que no estarían cargadas aún en el fichero y debiera de pasar un ciclo de reloj más para que estas se cargasen en el fichero.

```

when FETCH2 => -- ciclo 2 de búsqueda de instrucción
-- control externo
WR_enable <= '0';
-- control de buses
control_DATABUS_out <= '0';
control_ADDRBUS <= "00";
control_PC_Din <= '0';
control_Reg_Din <= '0';
-- control del contador de programa
PC_cuenta <= '0'; PC_load <= '0';
-- control del fichero de registros de proposito general
Reg_enable <= '1';
Reg_write <= '0';
Reg_address_WR <= (others=>'0');
Reg_addr_A <= SourceA;
Reg_addr_B <= SourceB;
-- control de la ALU
ALU_control <= "000"; -- control de operacion de la ALU
-- control del registro de instrucciones (RI: registro)
RI_load <= '0';
-- control del stack pointer (USP: SP)
SP_up <= '0'; SP_down <= '0'; SP_load <= '0';
-- control del registro de estado
STAT_C_load <= '0'; STAT_Z_load <= '0';
STAT_C_Set <= '0'; STAT_C_Clear <= '0';
STAT_SPF_load <= '0'; STAT_SPE_load <= '0';

case OpCode is
when "00000" => proximo_estado <= SUM;
when "00001" => proximo_estado <= SUB;
when "00010" => proximo_estado <= SHR;

```

Figura 6. Fetch2.

7.1.3. SUM & SUB.

A la hora de realizar dichas operaciones aritméticas, la ALU se ve implicada en el proceso, por lo que la entrada de control de la ALU deberá de ser la correspondiente (para suma ALU_Control es '000' y para resta es '001'). Además de esto, se debe de guardar el dato manipulado por la ALU en el fichero de registros, por eso el mux1 deberá de estar multiplexado a la entrada que proviene de la salida de la ALU y no a el bus de datos de entrada. A la hora de escribir el nuevo dato manipulado debemos de dar acceso de escritura a el fichero de registros y indicar el registro en el cual queremos guardar dicho dato (Reg_enable & Reg_write = '1'; Reg_address_WR = 'Destination'). Por último, los registros de estado, tanto el indicador de cero como el indicador de acarreo deben de ser cargados con las salidas obtenidas por la ALU pertenecientes a dichas indicaciones del sistema.

```

when SUM => -- SUM Ra,Rb,Rc: Ra<-Rb+Rc
-- control externo
WR_enable <= '0';
-- control de buses
control_DATABUS_out <= '0';
control_ADDRBUS <= "00";
control_PC_Din <= '0';
control_Reg_Din <= '1';
-- control del contador de programa
PC_cuenta <= '0'; PC_load <= '0';
-- control del fichero de registros de proposito general
Reg_enable <= '1';
Reg_write <= '1';
Reg_address_WR <= Destination;
Reg_addr_A <= (others=>'0');
Reg_addr_B <= (others=>'0');
-- control de la ALU
ALU_control <= "000"; -- control de operacion de la ALU
-- control del registro de instrucciones (RI: registro)
RI_load <= '0';
-- control del stack pointer (USP: SP)
SP_up <= '0'; SP_down <= '0'; SP_load <= '0';
-- control del registro de estado
STAT_C_load <= '1'; STAT_Z_load <= '1';
STAT_C_Set <= '0'; STAT_C_Clear <= '0';
STAT_SPF_load <= '0'; STAT_SPE_load <= '0';

proximo_estado <= FETCH1;

```

Figura 7. SUM.

7.1.4. SHR & SHL.

Ambas operaciones siguen el mismo procedimiento que las operaciones de suma y resta, las dos diferencias principales residen en la señal de control de la ALU ('011' para SHL y '010' para SHR) y la carga del registro de estado Z (indicador de cero), la cual no es necesaria en este caso, puesto que dicho registro se utiliza únicamente para indicar si el resultado obtenido por la ALU es cero.

7.1.5. INOT, IAND, IOR, IXOR.

Para las operaciones lógicas seguiremos el mismo procedimiento que en la clausula anterior para SHR y SHL, diferenciándose únicamente en la señal de control de la ALU, la cual será la correspondiente a cada operación, como ya se describió en el apartado respectivo de esta memoria.

7.1.6. LD.

Esta operación produce la carga de un registro en el fichero de registros desde memoria. Como se observa en el archivo Control.vhd, esta instrucción se divide en dos ciclos, LD1 y LD2. Para cargar un registro desde memoria, se debe primero leer la memoria en la dirección que se desee y seguidamente escribir, con el dato obtenido en dicha dirección, el registro pertinente:

when LD1	=> -- Ld Ra,dir: Ra<-Mem(dir) -- control externo WR_enable <= '0'; -- control de buses control_DATABUS_out <= '0'; control_ADDRBUS <= "10"; control_PC_Din <= '0'; control_Reg_Din <= '0'; -- control del contador de programa PC_cuenta <= '0'; PC_load <= '0'; -- control del fichero de registros de proposito general Reg_enable <= '1'; Reg_write <= '1'; Reg_address_WR <= Destination; Reg_addr_A <= (others=>'0'); Reg_addr_B <= (others=>'0'); -- control de la ALU ALU_control <= "000"; -- control de operacion de la ALU -- control del registro de instrucciones (RI: registro) RI_load <= '0'; -- control del stack pointer (USP: SP) SP_up <= '0'; SP_down <= '0'; SP_load <= '0'; -- control del registro de estado STAT_C_load <= '0'; STAT_Z_load <= '0'; STAT_C_Set <= '0'; STAT_C_Clear <= '0'; STAT_SPF_load <= '0'; STAT_SPE_load <= '0'; proximo_estado <= LD2;	when LD2	=> -- Ld Ra,dir: Ra<-Mem(dir) -- control externo WR_enable <= '0'; -- control de buses control_DATABUS_out <= '0'; control_ADDRBUS <= "00"; control_PC_Din <= '0'; control_Reg_Din <= '0'; -- control del contador de programa PC_cuenta <= '0'; PC_load <= '0'; -- control del fichero de registros de proposito general Reg_enable <= '1'; Reg_write <= '1'; Reg_address_WR <= Destination; Reg_addr_A <= (others=>'0'); Reg_addr_B <= (others=>'0'); -- control de la ALU ALU_control <= "000"; -- control de operacion de la ALU -- control del registro de instrucciones (RI: registro) RI_load <= '0'; -- control del stack pointer (USP: SP) SP_up <= '0'; SP_down <= '0'; SP_load <= '0'; -- control del registro de estado STAT_C_load <= '0'; STAT_Z_load <= '0'; STAT_C_Set <= '0'; STAT_C_Clear <= '0'; STAT_SPF_load <= '0'; STAT_SPE_load <= '0'; if dvalid='1' then proximo_estado <= FEICH1; else proximo_estado <= LD2; end if;
----------	--	----------	--

Figura 8. LD.

Como se puede observar en la figura 8, en el primer ciclo, la señal de control 'control_ADDRBUS' se establece a '10', dejando la salida del mux4 a la entrada dir (15 a 0 bits del RI). Aunque en el primer ciclo aparezca ya la escritura del fichero de registros en el destino indicado por la instrucción, esta escritura no es llevada a cabo realmente hasta el segundo ciclo, donde se escribe en el registro indicado el dato observado en la memoria alojado en la dirección designada.

7.1.7. ST.

Esta instrucción también se realiza en dos ciclos de reloj y realiza la operación inversa que la anterior instrucción, puesto que graba en memoria el dato almacenado en un determinado registro del fichero de registros.

En este caso, la señal de control de escritura de la memoria RAM es habilitada y la salida del fichero de registros que va hacia el mux2 es establecida como la fuente del dato que irá dirigido a memoria:

```

when ST1 => -- ST Ra,dir: Mem(dir)<-Ra
-- control externo
WR_enable <= '1';
-- control de buses
control_DATABUS_out <= '0';
control_ADDRBUS <= "10";
control_PC_Din <= '0';
control_Reg_Din <= '0';
-- control del contador de programa
PC_cuenta <= '0'; PC_load <= '0';
-- control del fichero de registros de proposito general
Reg_enable <= '1';
Reg_write <= '0';
Reg_address_WR <= (others=>'0');
Reg_addr_A <= SourceA;
Reg_addr_B <= (others=>'0');
-- control de la ALU
ALU_control <= "000"; -- control de operacion de la ALU
-- control del registro de instrucciones (RI: registro)
RI_load <= '0';
-- control del stack pointer (USP: SP)
SP_up <= '0'; SP_down <= '0'; SP_load <= '0';
-- control del registro de estado
STAT_C_load <= '0'; STAT_Z_load <= '0';
STAT_C_Set <= '0'; STAT_C_Clear <= '0';
STAT_SPF_load <= '0'; STAT_SPE_load <= '0';

proximo_estado <= ST2;

```

Figura 9. ST.

7.1.8. JMP.

Esta instrucción se encarga de realizar el salto a la dirección indicada en el campo 'dir' de la instrucción. De esta manera, para llevar a cabo el salto de dirección, lo único que se debe hacer es cargar el contador de programa (PC) con el valor de dir (para ello el mux3 deja pasar la señal 'dir'). Por lo que, sabiendo que el PC adquirirá un valor nuevo, lo único que debemos hacer es habilitar la señal de carga del PC.

7.1.9. JSR & RTS.

La instrucción JSR se encarga de realizar el salto a una determinada subrutina y guardar, en el contador de programa, la dirección de comienzo de la subrutina. Para ello se tendrá que guardar primero la dirección actual del PC en la pila (almacenada en la memoria externa), sirviendo esta como dirección de retorno. Además el puntero de pila SP se aumentará al realizar el salto a subrutina. En el archivo Control.vhd podemos diferenciar tres ciclos consecutivos de esta instrucción: JSR1, JSR2 y JSR3.

En el primer ciclo, JSR1, se incrementa el puntero de pila SP. Durante el segundo ciclo, JSR2, la señal de control del mux4 es establecida de tal manera que sea la señal de salida la que corresponda al ADDRBUS; y la señal de control del mux2 se establece a '1' para que se obtenga como salida DATABUS_out el registro PC. Con esto se consigue que en la dirección que apunta el puntero de pila se guarde el valor de comienzo de la subrutina o el valor del PC antes de saltar a la subrutina. Finalmente, para que en el PC resulte el valor 'dir' correspondiente a la nueva subrutina, el mux3 debe de establecer su salida (entrada del PC) como la entrada proveniente del campo dir del registro RI.

La instrucción RTS lleva el proceso inverso que la anterior instrucción comentada, de tal manera que se debe de recuperar la dirección almacenada en la pila (memoria externa) y decrementar el puntero de pila SP. Como se puede observar en el fichero Control.vhd, de la misma manera que con el salto a subrutina, esta instrucción se lleva a cabo en tres ciclos consecutivos; en el primer ciclo el mux4 establece la salida correspondiente a la entrada del SP, esto se hace para localizar el valor almacenado para el retorno de rutina; en el segundo ciclo, tras haber localizado en memoria el valor almacenado del PC, el mux3 establece como salida su entrada proveniente del bus de datos de entrada; y por último, en el tercer ciclo, el puntero de pila es decrementado, volviendo al valor original previo a el salto a subrutina.

7.1.10. CLC & SETC.

Estas dos instrucciones son bastante simple, pues definen por un lado (CLC) la carga a cero del bit de estado de acarreo (STAT_C_load = '1' & STAT_C_Clear = '1') y por otro lado (SETC) la carga a uno del bit de estado de acarreo (STAT_C_load = '1' & STAT_C_Set = '1'). Ambas se realizan en un solo y único ciclo de reloj.

7.1.11. STOP & NOP.

La instrucción STOP para de manera indefinida el procesador. El procesador se quedará en este estado hasta que se aplique un reset global del sistema.

La instrucción NOP (No OPeración) provoca que el procesador no haga nada durante un ciclo de reloj. A continuación se ejecutará la siguiente instrucción. Ambas se realizan en un solo y único ciclo de reloj.

7.2. Carga en memoria.

El programa ejecutado para testear el funcionamiento del procesador es el siguiente:

--Archivo programa.txt:

```
ST 54 R3;
LD R1 52;
LD R2 53;
SUM R3 R1 R2;
ST 54 R3;
SUB R4 R2 R1;
ST 55 R4;
SHR R5 R1;
SHL R6 R1;
SETC;
SHL R7 R1;
NOT R8 R1;
AND R9 R1 R2;
OR R10 R1 R2;
XOR R11 R1 R2;
ST 56 R5;
ST 57 R6;
ST 58 R7;
ST 59 R8;
ST 60 R9;
ST 61 R10;
ST 62 R11;
```

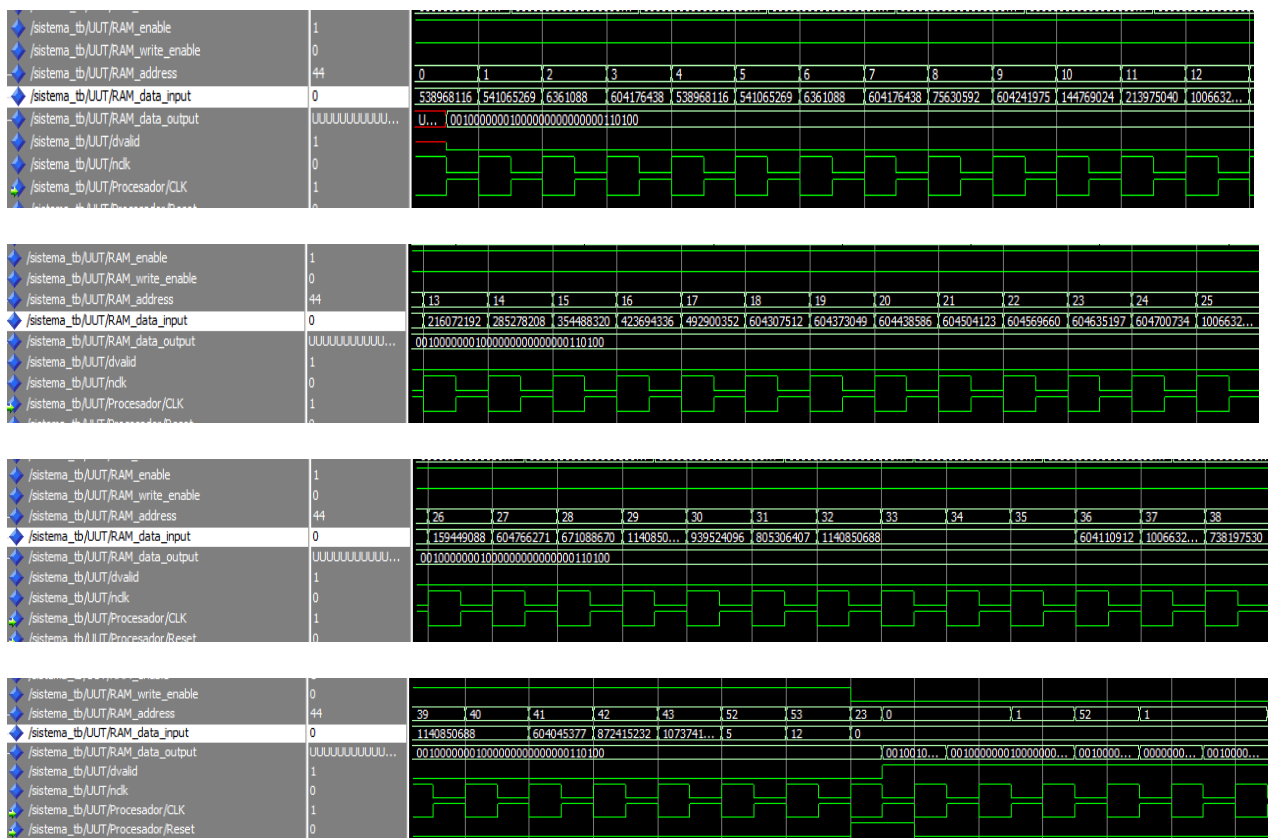


```

SETC;
SHR R12 R1;
ST 63 R12;
JMP 30;
NOP;
CLC;
JSR 39;
NOP;
NOP;
NOP;
NOP;
ST 64 R2;
SETC;
JPC 26;
NOP;
NOP;
ST 65 R1;
RTS;
STOP;
DATO 52 5
DATO 53 12

```

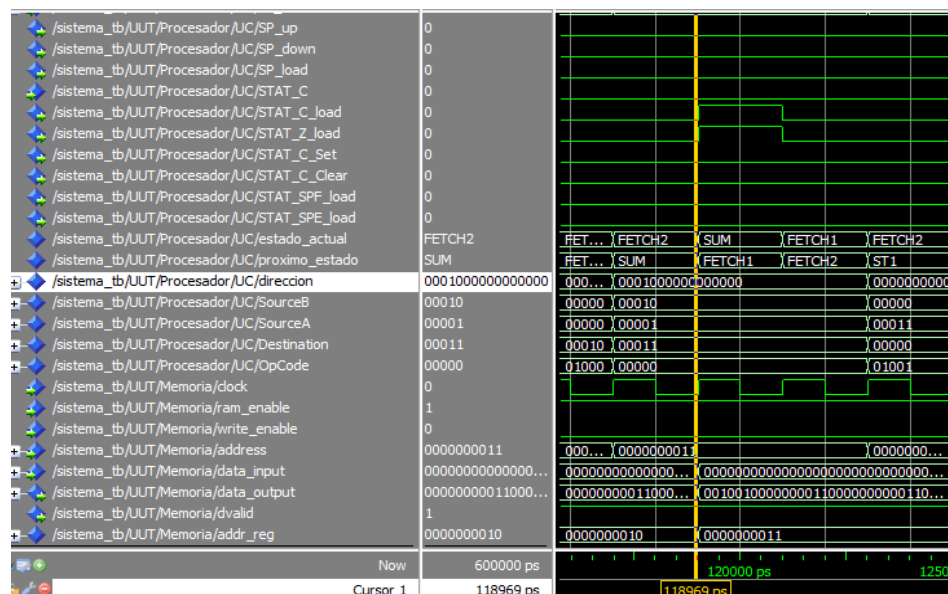
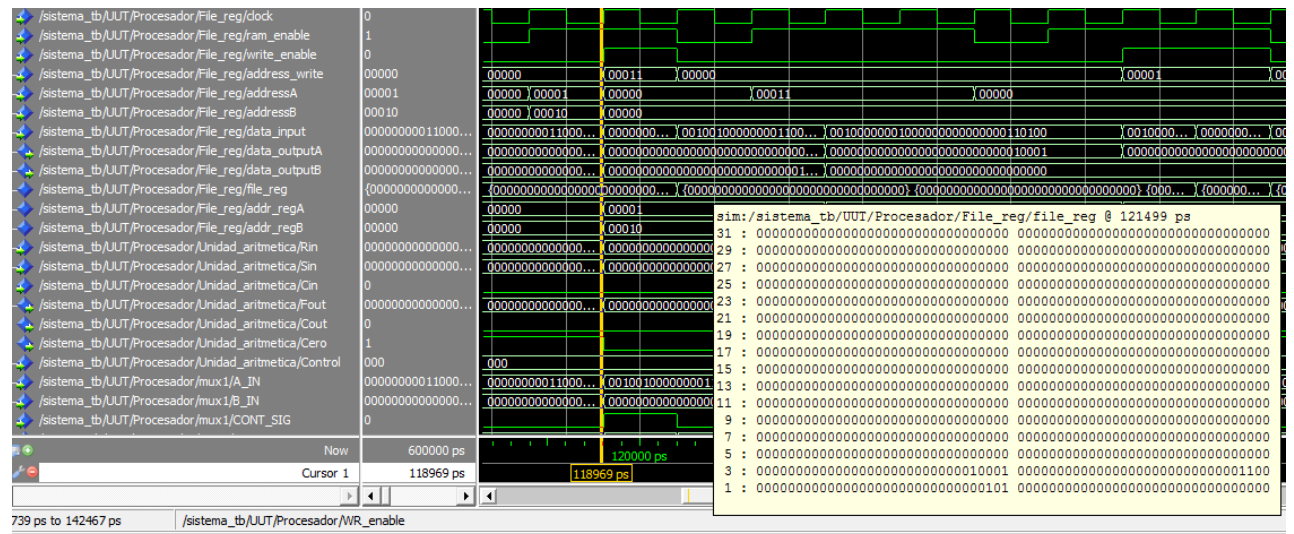
Contando con un total de 45 instrucciones, siendo dos de ellas el almacenamiento de los datos 5 y 12 en las direcciones 52 y 53. La carga de las instrucciones en la memoria se realiza satisfactoriamente:



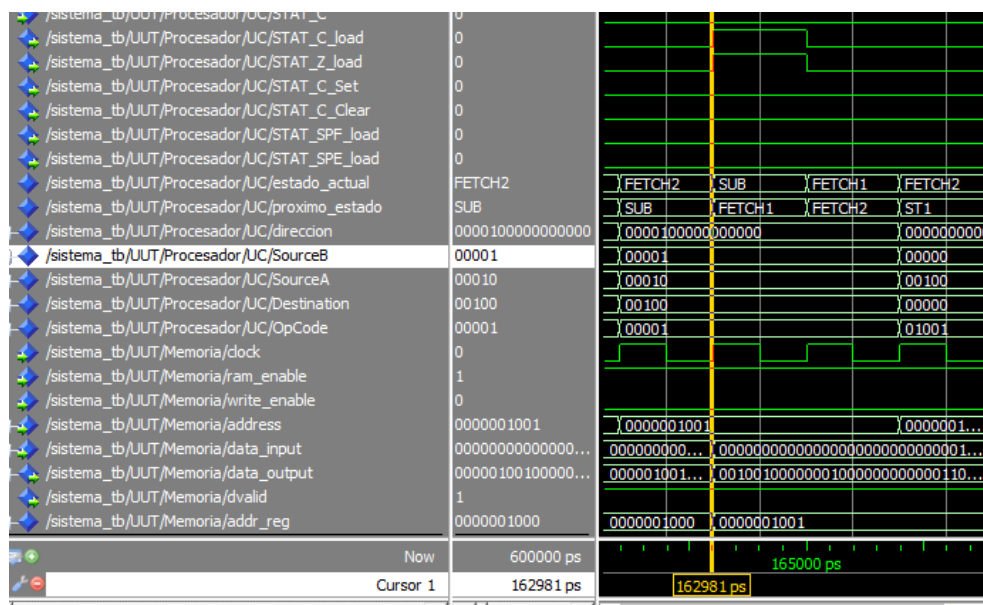
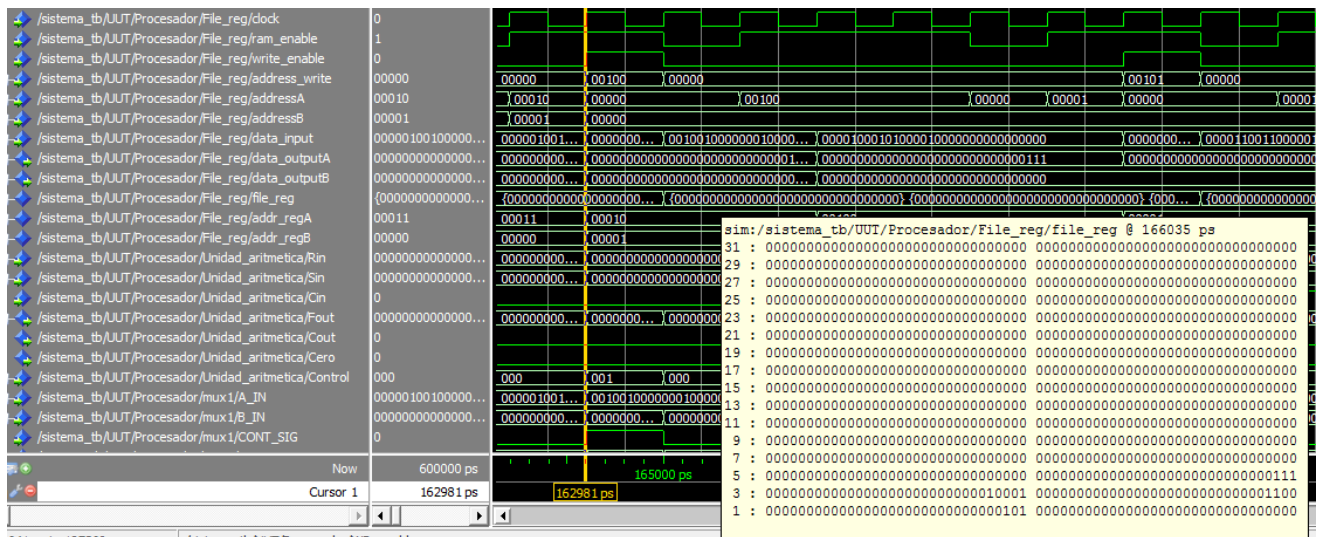
Este es un extracto del pantallazo con la carga de las instrucciones en la memoria, hasta llegar al registro 43 (ultima instruccion STOP), en dicho momento se pasa a los registros 52 y 53 donde se guardan los datos 5 y 12. Se observa que al llegar al final de la escritura, se lleva a cabo el reset del procesador así como la señal de 'write_enable' de la memoria de deshabilita.

7.3. Simulación del sistema.

7.3.1. SUM & SUB Simulación.

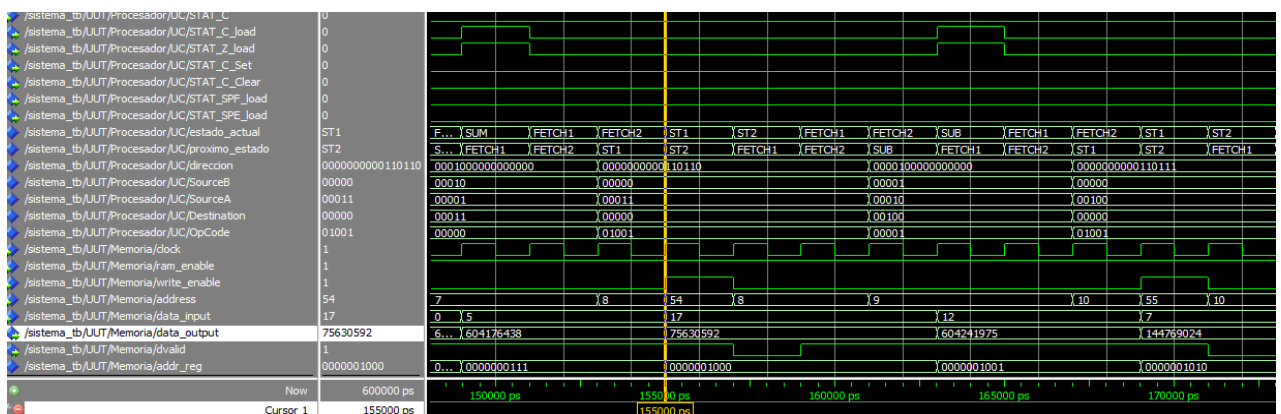


En las figuras de arriba se observa el momento en el que se ejecuta la instrucción SUM y el valor correspondiente a la operación de sumar 5 y 12, lo cual resulta en 17 y almacenado en el tercer registro del fichero de registros.



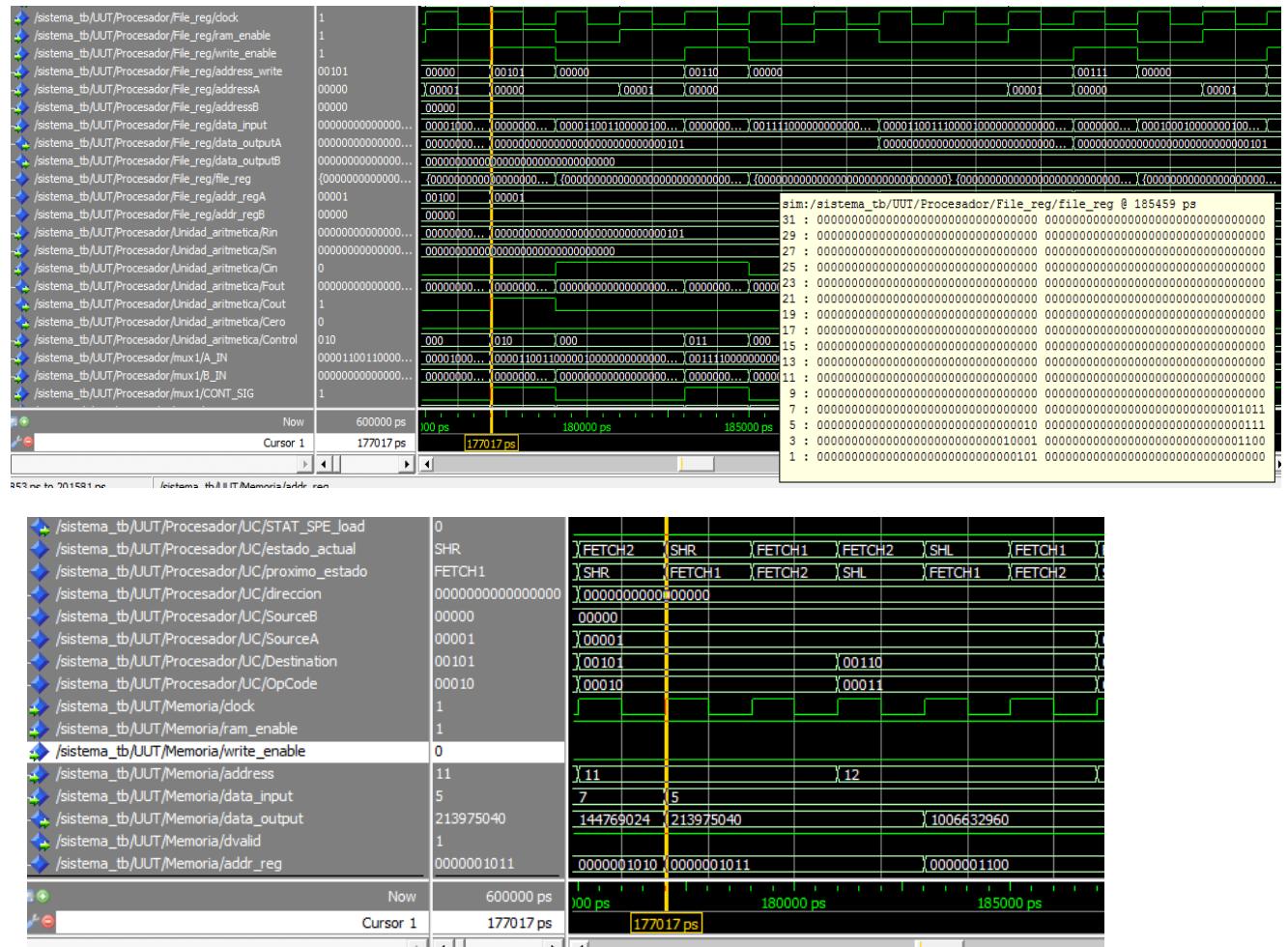
En las figuras de arriba se observa el momento en el que se ejecuta la instrucción SUB y el valor correspondiente a la operación de sumar 5 y 12, lo cual resulta en 7 y almacenado en el cuarto registro del fichero de registros.

7.3.2. ST Simulación.



Se puede observar como el resultado de las operaciones de la suma y de la resta se guardan en memoria en los registros indicados por el programa de instrucciones, 17 en el 54 y 7 en el 55.

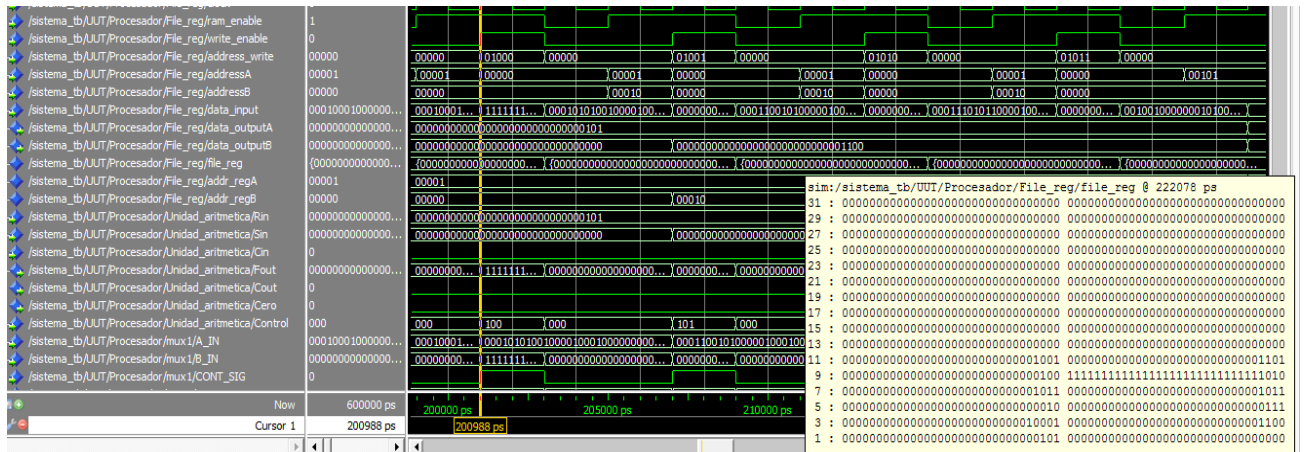
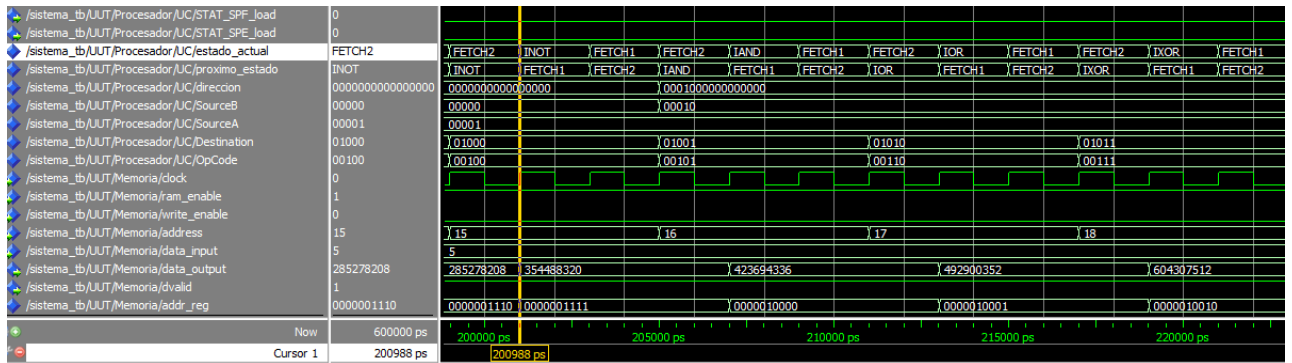
7.3.2. SHL & SHR Simulación.



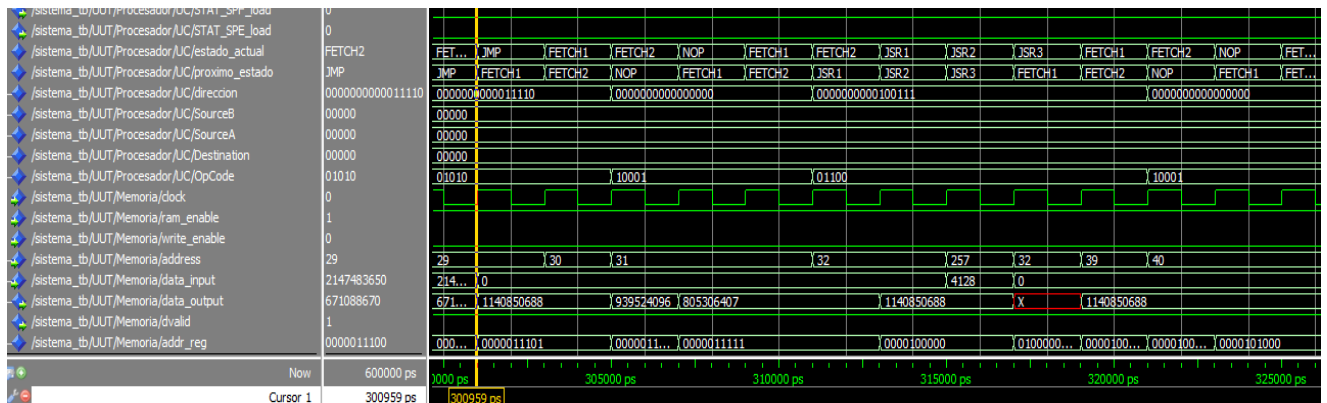
Se observa como en la instrucciones SHR y SHL los registros 5 y 6 se ven afectados respectivamente. En R5, R1 es desplazado hacia la derecha; y en R6, R1 es desplazado hacia la izquierda con acarreo de entrada habilitado.

7.3.3 Operaciones lógicas. Simulación.

Para este tipo de operaciones nos sirve con chequear el fichero de registros y ver que todas las operaciones se han realizado correctamente:



7.3.4. Otras simulaciones:



En este caso podemos observar las simulaciones y salidas obtenidas durante el chequeado de otras instrucciones como la de JMP y JSR.

A la hora de simular la instrucción de retorno de subrutina, se ha producido un fallo de simulación para el cual no se ha logrado encontrar solución; la address de la memoria no se establece correctamente al valor guardado en la pila anteriormente.

