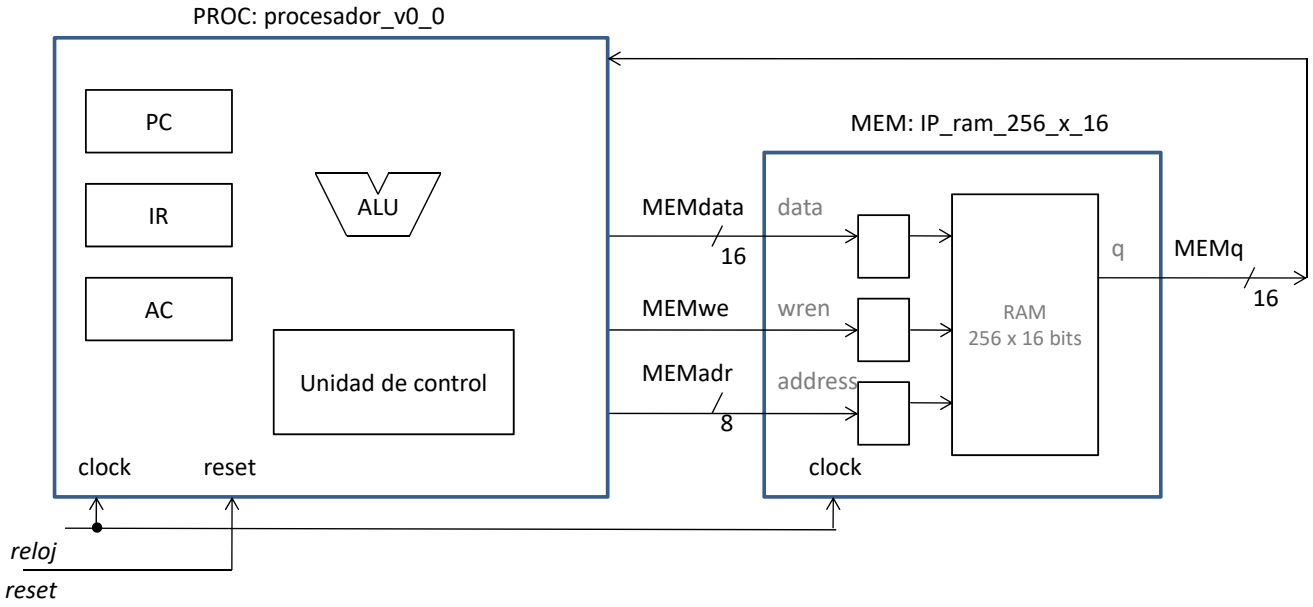
Desarrollo de Hardware Digital
Práctica 1

Diseño de un procesador en VHDL
Análisis y optimización.
Ampliación del repertorio de instrucciones

Computador elemental

my_scomp_v0_0.vhd



Repertorio de instrucciones

CODOP[1]	Instrucción	Descripción
00	ADD address	AC[2] <= AC + M(address)
01	STORE address	M(address) <= AC
02	LOAD address	AC <= M(address)
03	JUMP address	PC[2] <= address
04	SUB address	AC <= AC - M(address)
05	NAND address	AC <= AC NAND M(address)
06	JNEG address	Si AC < 0, entonces PC <= address
07	JPOs address	Si AC > 0, entonces PC <= address
08	JZERO address	Si AC = 0, entonces PC <= address
09	SHL v	Desplazamiento lógico a la izquierda de AC el número de posiciones indicado en el argumento "v"
0A	SHR v	Desplazamiento lógico a la derecha de AC el número de posiciones indicado en el argumento "v"
0B	IN v	Si v(0) = '0', entonces AC(7 downto 0) <= SW[3](7 downto 0)
		Si v(0) = '1' , entonces AC(15 downto 8) <= SW(7 downto 0)
0C	OUT v	Si v(0) = '0', entonces LEDG[3](7 downto 0) <= AC(7 downto 0)
		Si v(0) = '1' , entonces LEDG(7 downto 0) <= AC(15 downto 8)
0D	CALL address	Llamada a subrutina que comienza en la dirección "address"[4]
0E	RET	Retorno de subrutina[4]
0F	STOP	Detiene la ejecución del programa hasta que se actúa[5] sobre una entrada externa (CONT)

Repertorio de instrucciones (versión inicial)

CODOP[1]	Instrucción	Descripción
00	ADD address	AC[2] <= AC + M(address)
01	STORE address	M(address) <= AC
02	LOAD address	AC <= M(address)
03	JUMP address	PC[2] <= address

Redacción de programas

- Redactar en hexadecimal un programa que sume el contenido de tres posiciones consecutivas de memoria, y almacene el resultado en la posición de memoria siguiente.
- Crear con la opción *New → Memory Files → Memory Initialization File* un fichero denominado programa.mif para almacenar el contenido de una memoria de 256 palabras y 16 bits por palabra.
- Almacenar el programa a partir de la dirección 00h, y los datos en las posiciones consecutivas que se hayan asignado.

NOTA: Puede resultar conveniente que tanto el contenido como las direcciones de memoria se muestren en hexadecimal ajustando las opciones View → Memory Radix y View → Address Radix, respectivamente.

Repertorio de instrucciones (versión inicial)

CODOP(1)	Instrucción	Descripción
00	ADD address	AC(2) <= AC + M(add
01	STORE address	M(address) <= AC
02	LOAD address	AC <= M(address)
03	JUMP address	PC(2) <= address

Cada instrucción ocupa 16 bits. En la versión original, los 8 bits más significativos corresponden al código de operación (*CODOP*) y los restantes a una dirección (*address*).

Dirección	Instrucción Nemotécnicos	Descripción	Instrucción máquina (hexadecimal)
00	LOAD H'20	Ac ← M(20)	0220
01	ADD H'21	Ac ← Ac + M(21)	0021
02	ADD H'22	Ac ← Ac + M(22)	0022
03	STORE H'23	M(23) ← Ac	0123
04	JUMP H'04	Salto a H'04	0304

Repertorio de instrucciones (versión inicial)

CODOP[1]	Instrucción	Descripción
00	ADD address	AC[2] <= AC + M(address)
01	STORE address	M(address) <= AC
02	LOAD address	AC <= M(address)
03	JUMP address	PC[2] <= address

```
-- Quartus II generated Memory Initialization File (.mif)

WIDTH=16;
DEPTH=256;

ADDRESS_RADIX=HEX;
DATA_RADIX=HEX;

CONTENT BEGIN
        000  :   0220;
        001  :   0021;
        002  :   0022;
        003  :   0123;
        004  :   0304;
        [005..01F]  :   0000;
        020  :   F000;
        021  :   0500;
        022  :   0034;
        [023..0FF]  :   0000;
END;
```

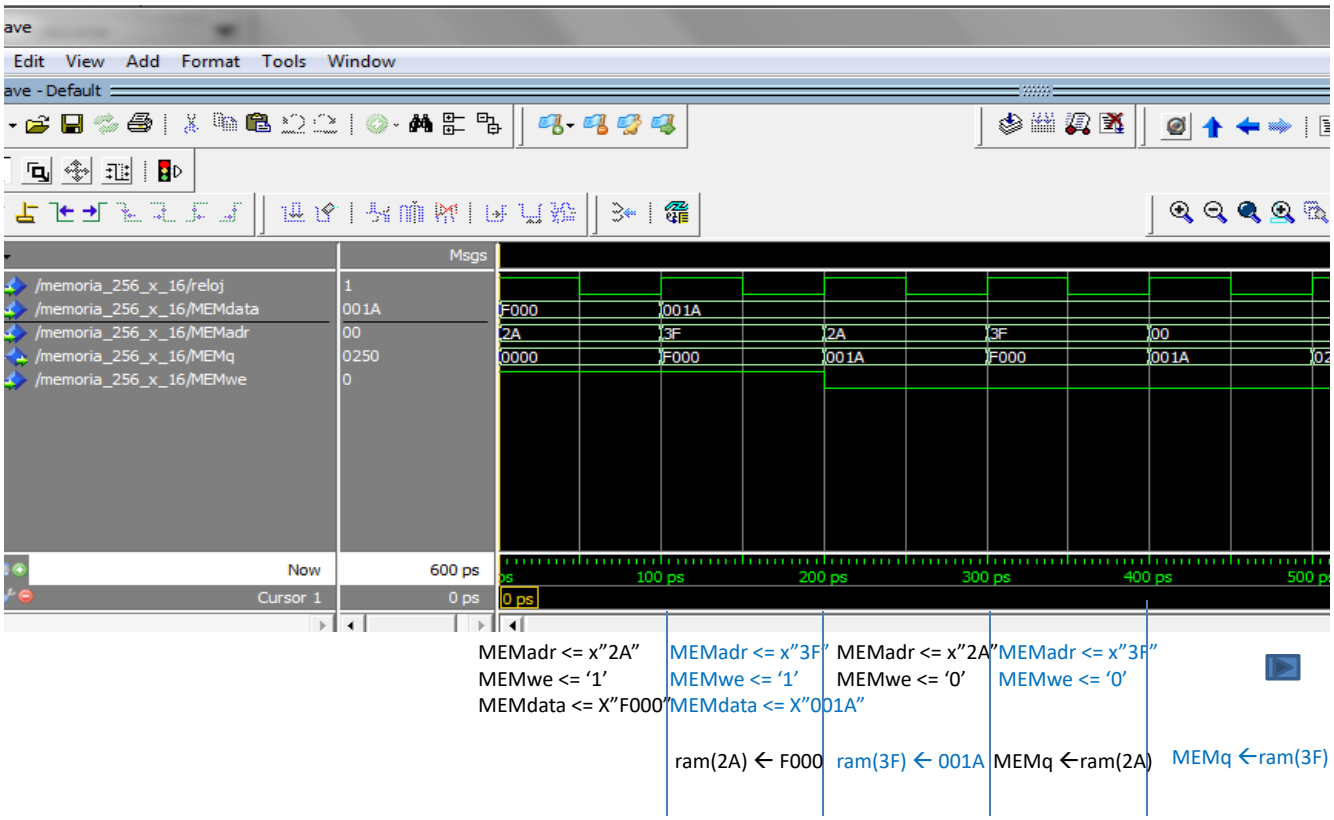
Generación del módulo de memoria

- Emplearemos el asistente para generar un *core* IP (denominado `IP_ram_256_x_16`) correspondiente a una memoria RAM de 1 sólo puerto con 256 palabras y 16 bits por palabra.
- Las entradas de datos, direcciones y control de escritura deben estar REGISTRADAS.
- Las salidas de datos de la memoria sean NO REGISTRADAS.
- Por tanto, en un ciclo se ordena la operación (lectura/escritura), se indica la dirección y el dato (si es escritura), y después del flanco (en el ciclo siguiente) se efectúa la operación en el bloque RAM.

Instanciación del módulo de memoria

- Editar el fichero `my_scomp_v0_0.vhd` para declarar el componente de la memoria, e instanciarlo conectando sus puertos según se indica en la Figura 1.

Memoria RAM (256 x 16 bits)



Verificación del procesador (versión 0.0)

- Crear un proyecto para la descripción `my_scomp_v0_0.vhd`
- Analizar la descripción VHDL del componente `procesador_v0_0` y dibujar el diagrama de estados de la unidad de control del procesador en su versión inicial.

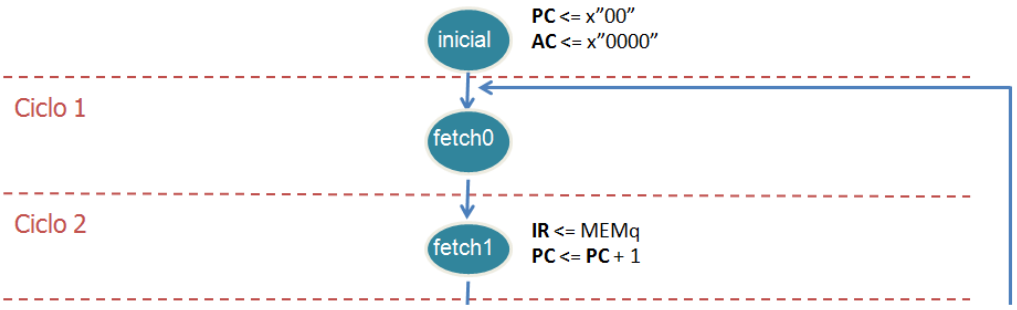
El procesador (versión 0.0)

```
-- Descripción de una procesador que ejecuta cuatro instrucciones.
-- Basado en ejemplo de Hamblen, J.O., Hall T.S., Furman, M.D.:
-- Rapid Prototyping of Digital Systems : SOPC Edition, Springer 2008.
-- (Capítulo 9)

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY procesador_v0_0 IS
PORT( clock : IN STD_LOGIC;
      reset : IN STD_LOGIC;
      AC_out : out std_logic_vector(15 downto 0);
      IR_out : out std_logic_vector(15 downto 0);
      PC_out : out std_logic_vector(7 downto 0);
      MEMq : in std_logic_vector(15 downto 0);
      MEMdata: out std_logic_vector(15 downto 0);
      MEMwe : out std_logic;
      MEMadr : out std_logic_vector(7 downto 0)
      );
END procesador_v0_0;

ARCHITECTURE rtl OF procesador_v0_0 IS
   TYPE STATE_TYPE IS ( reset_pc, fetch0, fetch1, decode, add0, add1, load0, load
                        store0, store1, jump);
   SIGNAL state: STATE_TYPE;
   SIGNAL IR, AC: STD_LOGIC_VECTOR(15 DOWNTO 0 );
   SIGNAL PC : STD_LOGIC_VECTOR( 7 DOWNTO 0 );
```

El procesador (versión 0.0)

```
    BEGIN


    -- Asignaciones a puertos de salida
    --
    AC_out <= AC;
    IR_out <= IR;
    PC_out <= PC;


FSMD: PROCESS ( CLOCK, RESET, state, PC, AC, IR )

 BEGIN

 -- Asignaciones a REGISTROS en datapath y MAQUINA DE ESTADOS de la unidad de control
IF reset = '1' THEN
    state <= reset_pc;
    ELSIF clock'EVENT AND clock = '1' THEN
     CASE state IS
        WHEN reset_pc =>
            PC <= "00000000";
            AC <= "0000000000000000";
            state <= fetch0;
        WHEN fetch0 =>
            state <= fetch1;
        WHEN fetch1 =>
            IR <= MEMq;
            PC <= PC + 1;
            state <= decode;
```

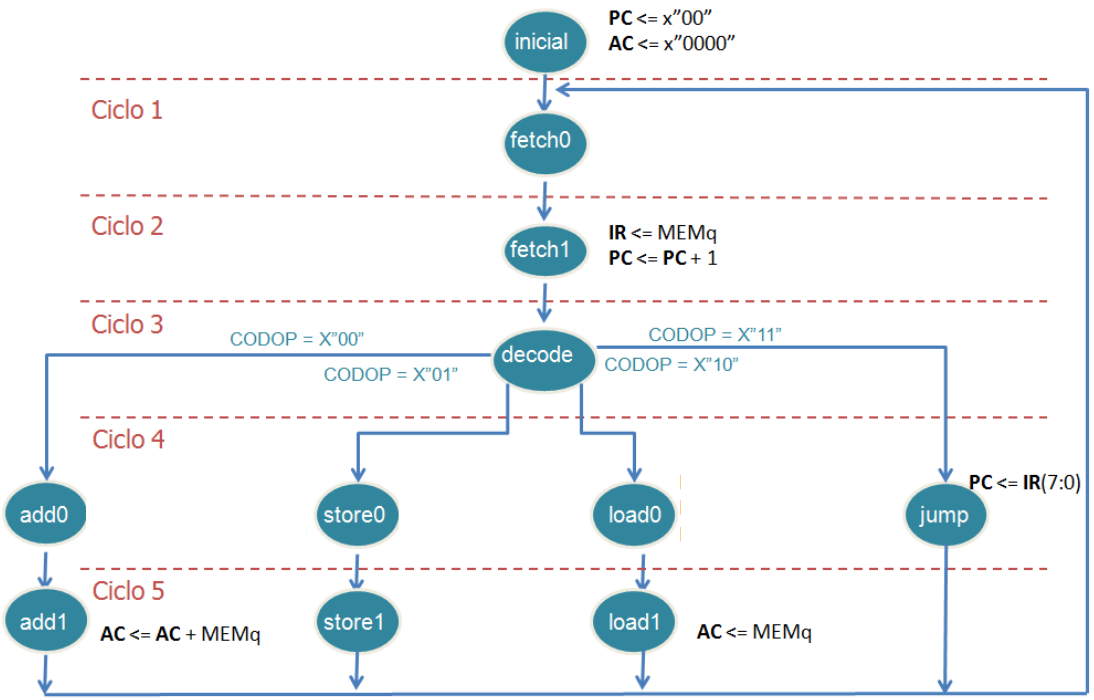
El procesador (versión 0.0)



El procesador (versión 0.0)

```
      WHEN decode =>
         CASE IR( 15 DOWNTO 8 ) IS
            WHEN "00000000" =>
               state <= add0;
            WHEN "00000001" =>
               state <= store0;
            WHEN "00000010" =>
               state <= load0;
            WHEN "00000011" =>
               state <= jump;
            WHEN OTHERS =>
               state <= fetch0;
         END CASE;
      WHEN add0 =>
         state <= add1;
      WHEN add1 =>
         AC <= AC + MEMq;
         state <= fetch0;
      WHEN store0 =>
         state <= store1;
      WHEN store1 =>
         state <= fetch0;
      WHEN load0 =>
         state <= load1;
      WHEN load1 =>
         AC <= MEMq;
         state <= fetch0;
      WHEN jump =>
         PC <= IR( 7 DOWNTO 0 );
         state <= fetch0;
      WHEN OTHERS =>
         state <= fetch0;
   END CASE;
```

El procesador (versión 0.0)



El procesador (versión 0.0)

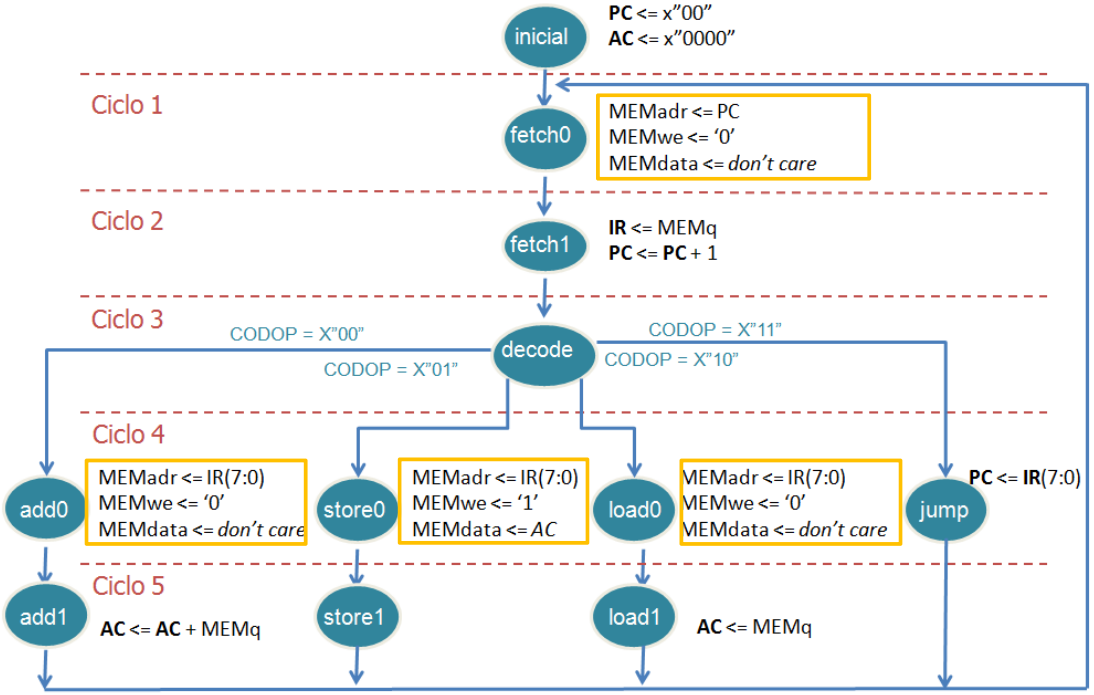
```
-- Asignaciones a BUSES de entrada a MEMORIA (Direcciones, Datos y control de escritura)

   CASE state IS
      WHEN fetch0 =>
         MEMadr <= PC;
         MEMwe <= '0';
         MEMdata <= (others =>'-');
      WHEN add0 | load0 =>
         MEMadr <= IR(7 downto 0);
         MEMwe <= '0';
         MEMdata <= (others =>'-');
      WHEN store0 =>
         MEMadr <= IR(7 downto 0);
         MEMwe <= '1';
         MEMdata <= AC;
      WHEN others =>
         MEMadr <= IR(7 downto 0);
         MEMwe <= '0';
         MEMdata <= (others =>'-');
   end case;

END PROCESS;

END rtl;
```

El procesador (versión 0.0)

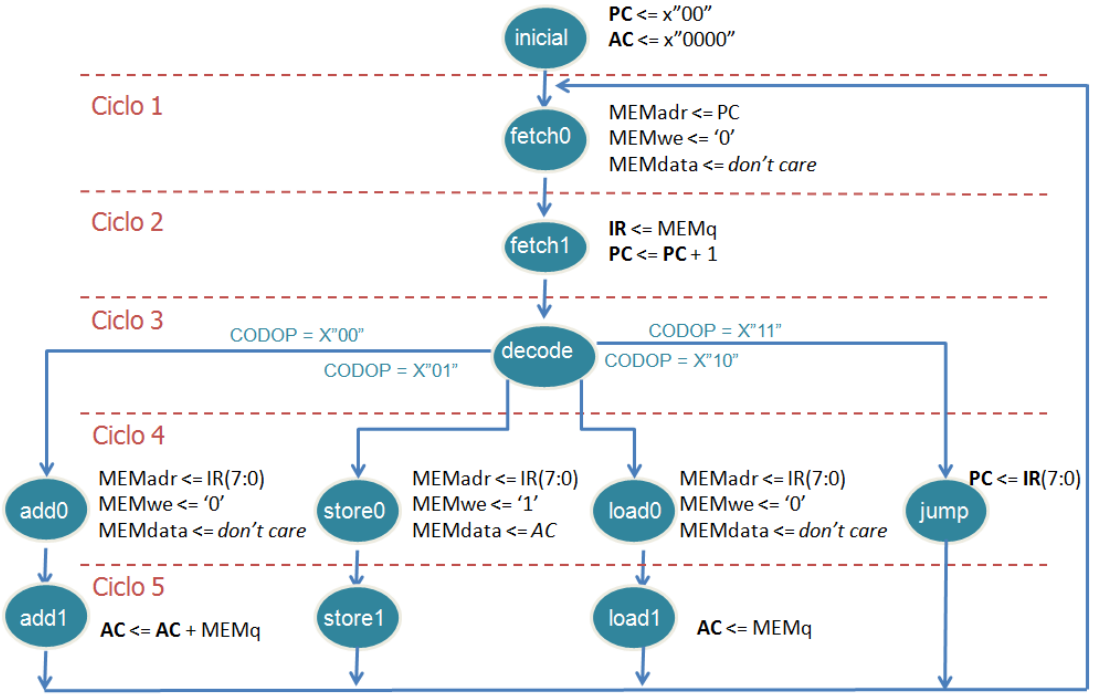
```
-- Asignaciones a BUSES de entrada a MEMORIA (Direcciones, Datos y control de escritura)

   CASE state IS
      WHEN fetch0 =>
         MEMadr <= PC;
         MEMwe <= '0';
         MEMdata <= (others =>'-');
      WHEN add0 | load0 =>
         MEMadr <= IR(7 downto 0);
         MEMwe <= '0';
         MEMdata <= (others =>'-');
      WHEN store0 =>
         MEMadr <= IR(7 downto 0);
         MEMwe <= '1';
         MEMdata <= AC;
      WHEN others =>
         MEMadr <= IR(7 downto 0);
         MEMwe <= '0';
         MEMdata <= (others =>'-');
   end case;

END PROCESS;

END rtl;
```

Por razones de claridad, no se representan estas asignaciones por defecto en el diagrama de la siguiente transparencia

El procesador (versión 0.0)



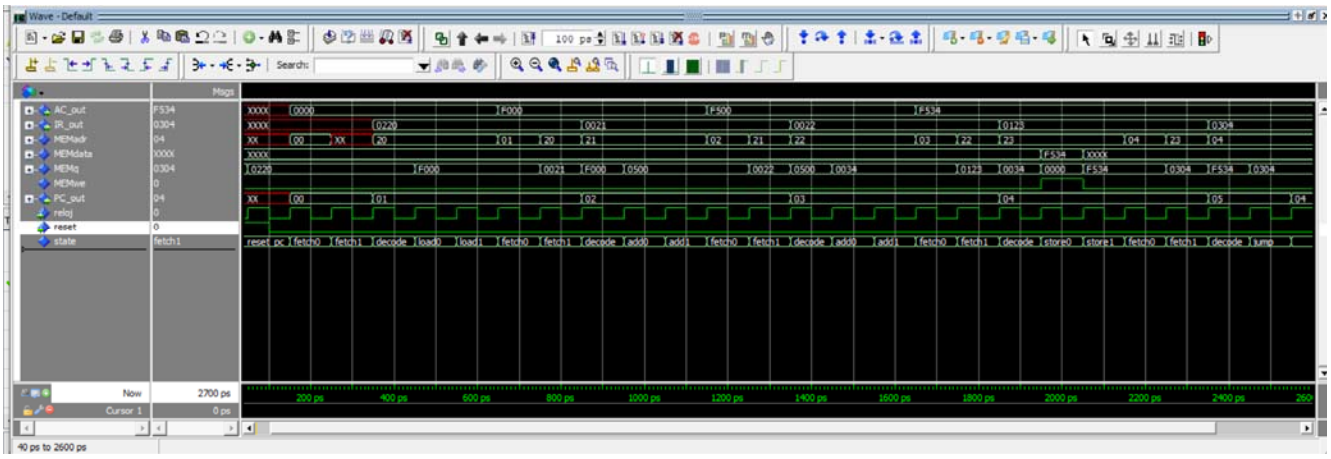
El procesador (versión 0.0)



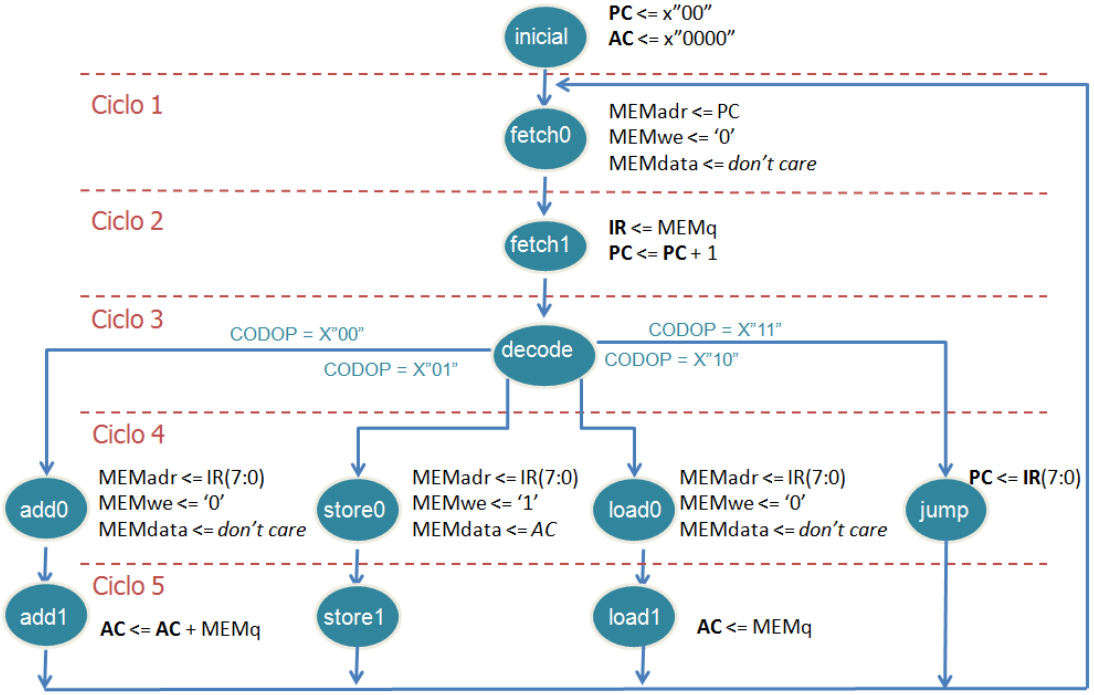
Verificación del procesador (versión 0.0)

- Crear un proyecto para la descripción `my_scomp_v0_0.vhd`
- Analizar la descripción VHDL del componente `procesador_v0_0` y dibujar el diagrama de estados de la unidad de control del procesador en su versión inicial.
- Realizar la síntesis RT-lógica *(Processing → Start → Analysis y Synthesis, Ctrl-K)* y visualizar la netlist resultante de la síntesis RT inicial *(Tools → Netlist Viewers → RTL Viewer)*.
- Realizar con Modelsim-Altera una simulación funcional del diseño `my_scomp_v0_0.vhd`.

Simulación funcional (my_scomp_v0_0)



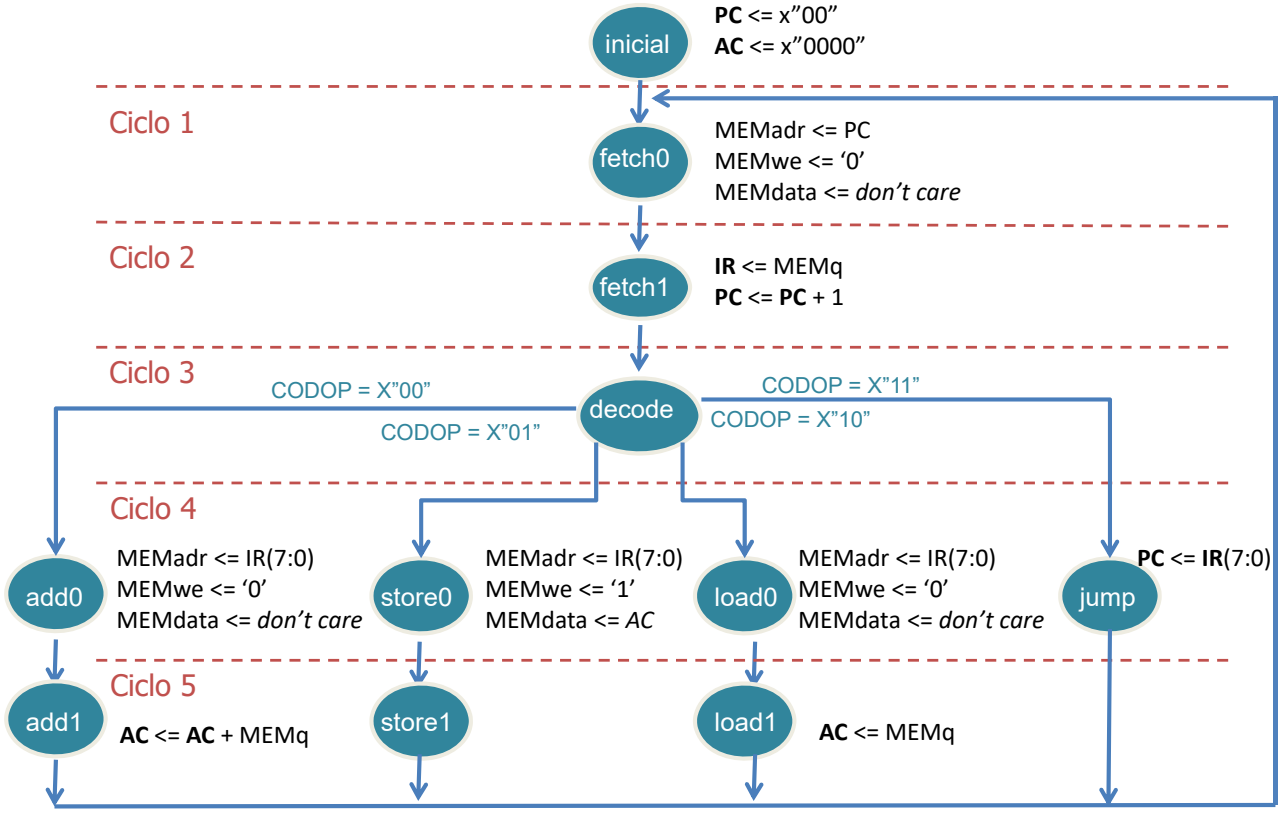
El procesador (versión 0.0)

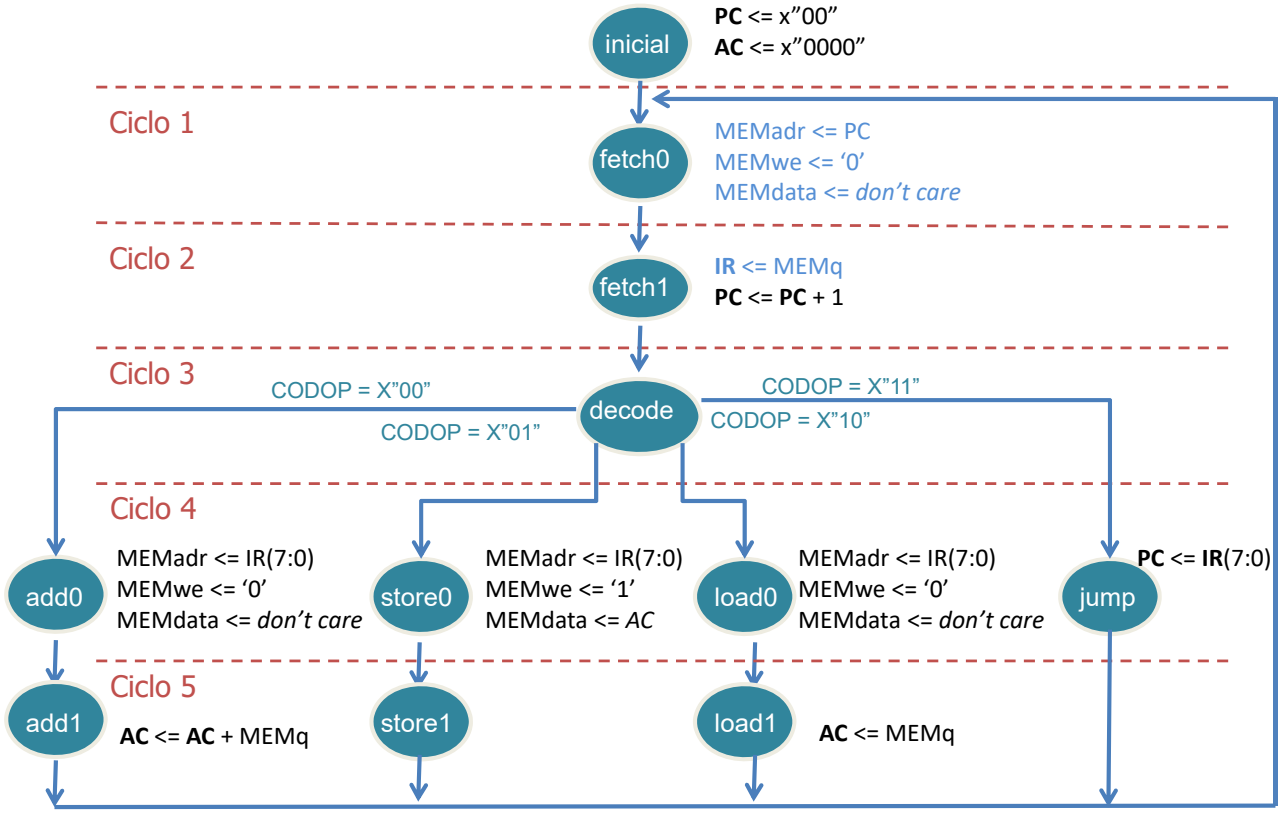


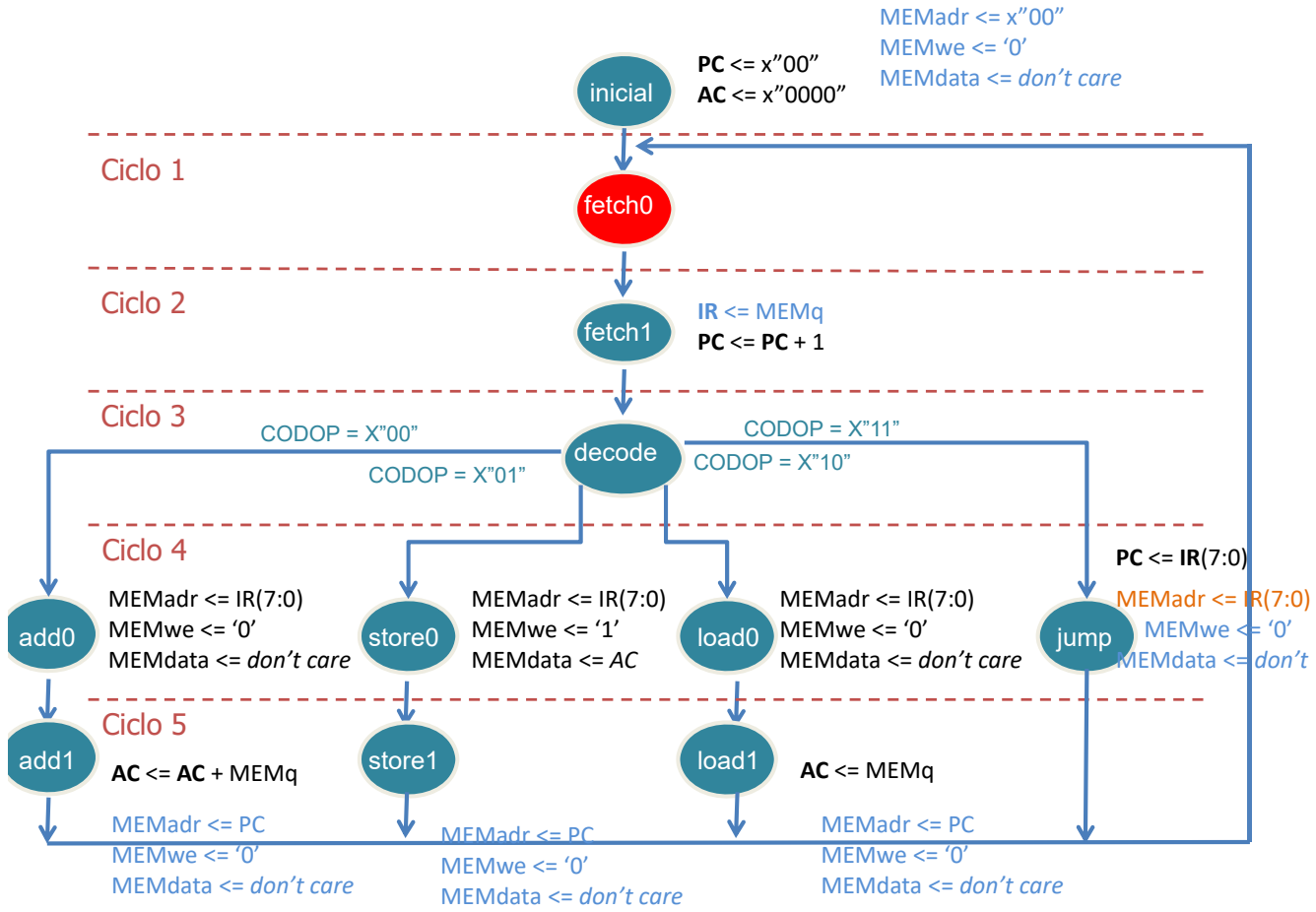
Optimización

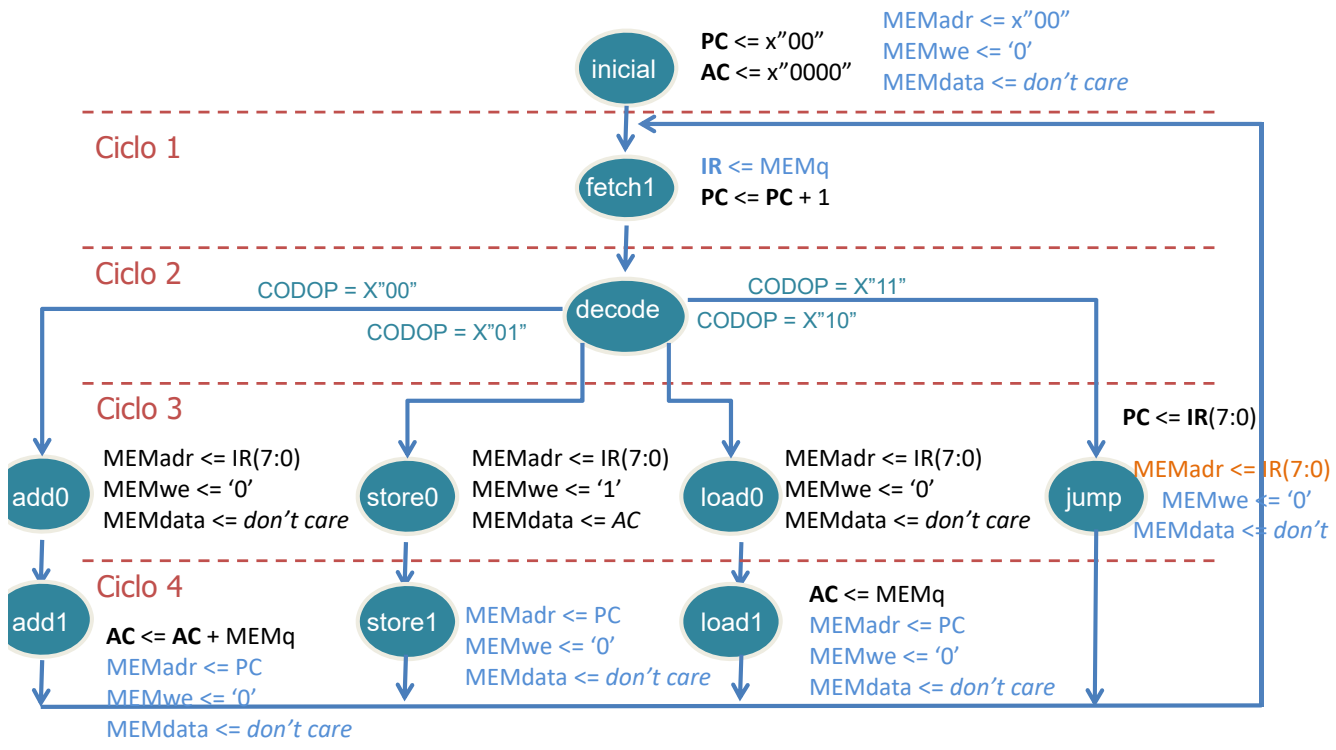
- **Reducción del número de ciclos por instrucción**

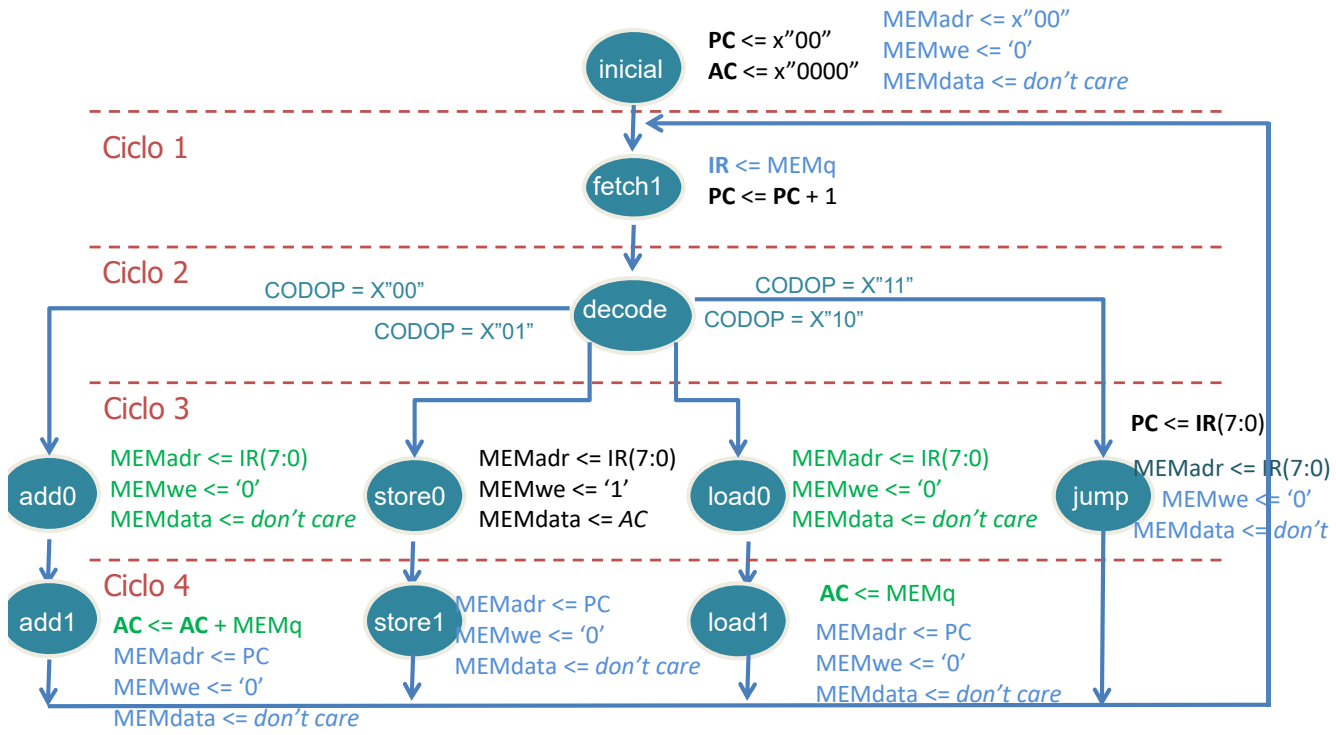
Eliminando en lo posible estados que únicamente se utilizan para ordenar operaciones con la memoria y reutilizando los estados en el ciclo anterior para asignar valor a los buses de entrada a memoria.

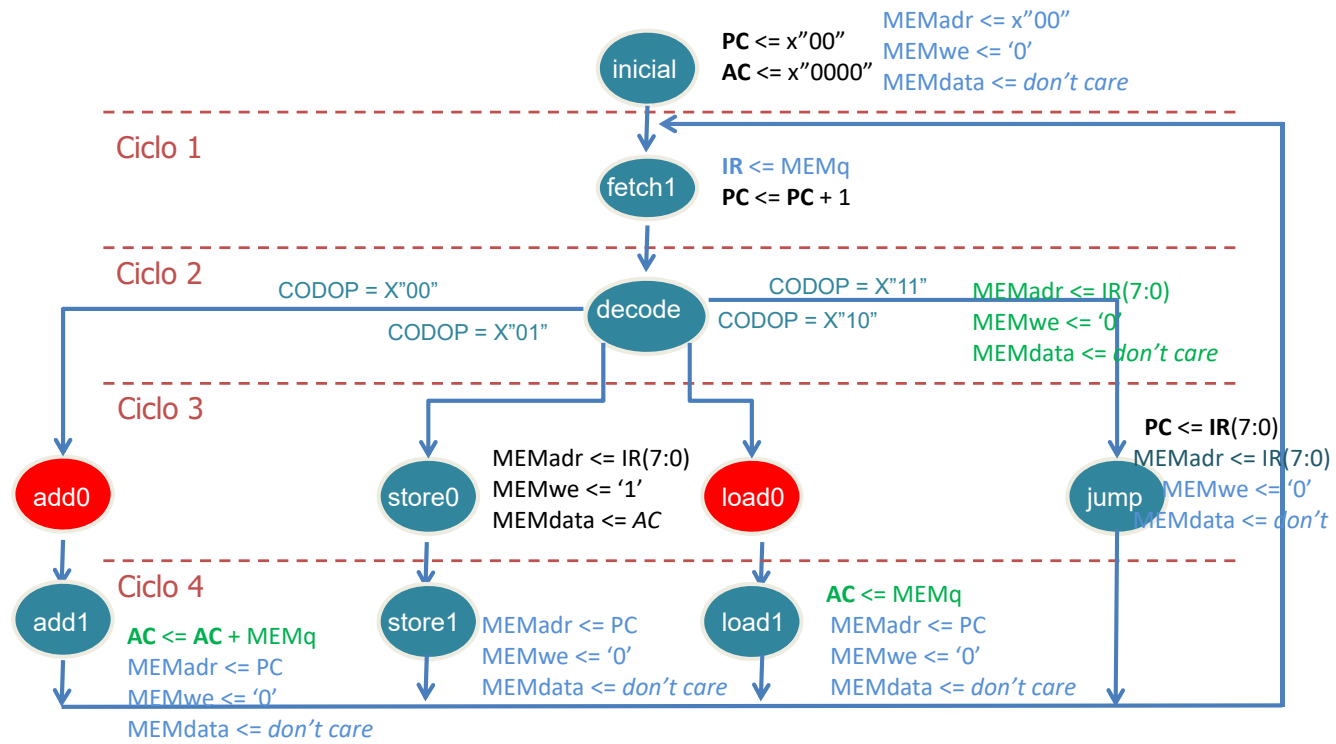

inicial
PC <= x"00"
AC <= x"0000"
Ciclo 1
fetch0
MEMadr <= PC
MEMwe <= '0'
MEMdata <= don't care
Ciclo 2
fetch1
IR <= MEMq
PC <= PC + 1
Ciclo 3
decode
CODOP = X"00"
CODOP = X"01"
CODOP = X"11"
CODOP = X"10"
Ciclo 4
add0
MEMadr <= IR(7:0)
MEMwe <= '0'
MEMdata <= don't care
store0
MEMadr <= IR(7:0)
MEMwe <= '1'
MEMdata <= AC
load0
MEMadr <= IR(7:0)
MEMwe <= '0'
MEMdata <= don't care
jump
PC <= IR(7:0)
Ciclo 5
add1
AC <= AC + MEMq
store1
load1
AC <= MEMq


inicial
PC <= x"00"
AC <= x"0000"
Ciclo 1
fetch0
MEMadr <= PC
MEMwe <= '0'
MEMdata <= don't care
Ciclo 2
fetch1
IR <= MEMq
PC <= PC + 1
Ciclo 3
decode
CODOP = X"00"
CODOP = X"01"
CODOP = X"11"
CODOP = X"10"
Ciclo 4
add0
MEMadr <= IR(7:0)
MEMwe <= '0'
MEMdata <= don't care
store0
MEMadr <= IR(7:0)
MEMwe <= '1'
MEMdata <= AC
load0
MEMadr <= IR(7:0)
MEMwe <= '0'
MEMdata <= don't care
jump
PC <= IR(7:0)
Ciclo 5
add1
AC <= AC + MEMq
store1
load1
AC <= MEMq

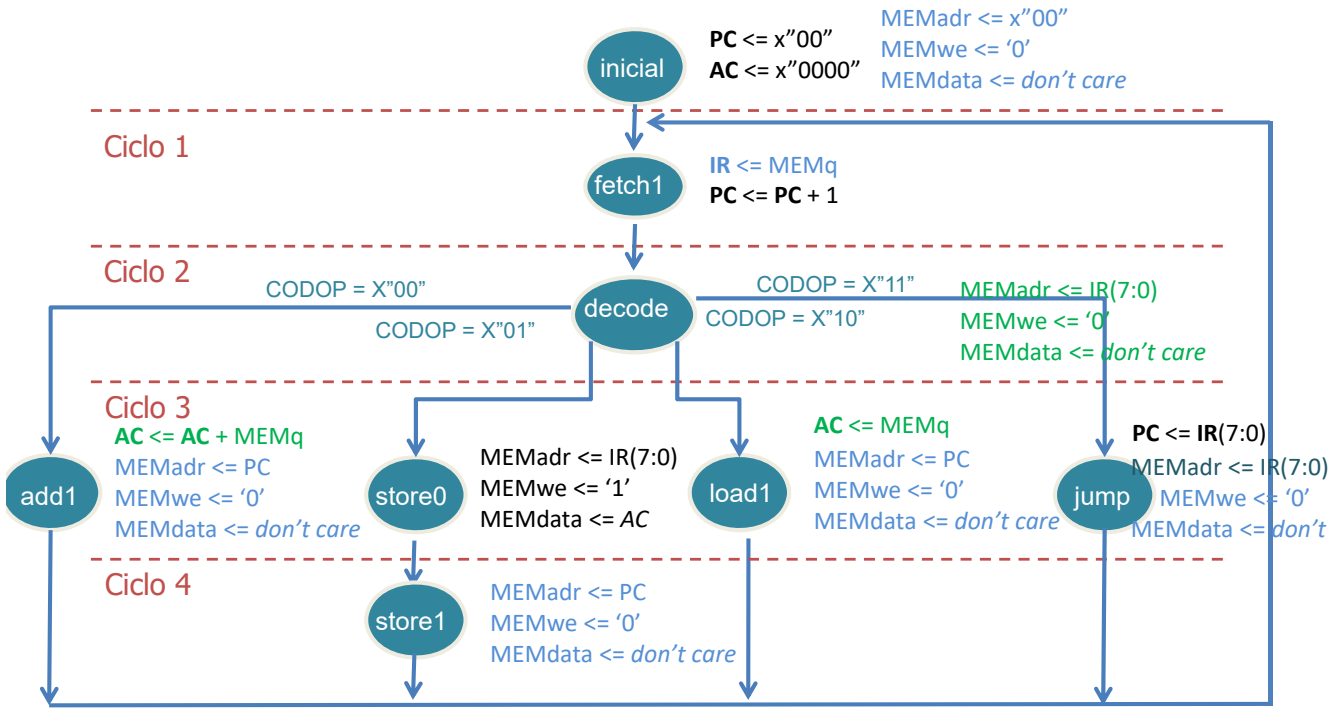

inicial
PC <= x"00"
AC <= x"0000"
MEMadr <= x"00"
MEMwe <= '0'
MEMdata <= don't care
Ciclo 1
fetch0
Ciclo 2
fetch1
IR <= MEMq
PC <= PC + 1
Ciclo 3
decode
CODOP = X"00"
CODOP = X"01"
CODOP = X"11"
CODOP = X"10"
Ciclo 4
add0
MEMadr <= IR(7:0)
MEMwe <= '0'
MEMdata <= don't care
store0
MEMadr <= IR(7:0)
MEMwe <= '1'
MEMdata <= AC
load0
MEMadr <= IR(7:0)
MEMwe <= '0'
MEMdata <= don't care
jump
PC <= IR(7:0)
MEMadr <= IR(7:0)
MEMwe <= '0'
MEMdata <= don't
Ciclo 5
add1
AC <= AC + MEMq
store1
load1
AC <= MEMq
MEMadr <= PC
MEMwe <= '0'
MEMdata <= don't care
MEMadr <= PC
MEMwe <= '0'
MEMdata <= don't care
MEMadr <= PC
MEMwe <= '0'
MEMdata <= don't care


inicial
PC <= x"00"
AC <= x"0000"
MEMadr <= x"00"
MEMwe <= '0'
MEMdata <= don't care
Ciclo 1
fetch1
IR <= MEMq
PC <= PC + 1
Ciclo 2
decode
CODOP = X"00"
CODOP = X"01"
CODOP = X"11"
CODOP = X"10"
Ciclo 3
add0
MEMadr <= IR(7:0)
MEMwe <= '0'
MEMdata <= don't care
store0
MEMadr <= IR(7:0)
MEMwe <= '1'
MEMdata <= AC
load0
MEMadr <= IR(7:0)
MEMwe <= '0'
MEMdata <= don't care
jump
PC <= IR(7:0)
MEMadr <= IR(7:0)
MEMwe <= '0'
MEMdata <= don't
Ciclo 4
add1
AC <= AC + MEMq
MEMadr <= PC
MEMwe <= '0'
MEMdata <= don't care
store1
MEMadr <= PC
MEMwe <= '0'
MEMdata <= don't care
load1
AC <= MEMq
MEMadr <= PC
MEMwe <= '0'
MEMdata <= don't care


inicial
PC <= x"00"
AC <= x"0000"
MEMadr <= x"00"
MEMwe <= '0'
MEMdata <= don't care
Ciclo 1
fetch1
IR <= MEMq
PC <= PC + 1
Ciclo 2
decode
CODOP = X"00"
CODOP = X"01"
CODOP = X"11"
CODOP = X"10"
Ciclo 3
add0
MEMadr <= IR(7:0)
MEMwe <= '0'
MEMdata <= don't care
store0
MEMadr <= IR(7:0)
MEMwe <= '1'
MEMdata <= AC
load0
MEMadr <= IR(7:0)
MEMwe <= '0'
MEMdata <= don't care
jump
PC <= IR(7:0)
MEMadr <= IR(7:0)
MEMwe <= '0'
MEMdata <= don't
Ciclo 4
add1
AC <= AC + MEMq
MEMadr <= PC
MEMwe <= '0'
MEMdata <= don't care
store1
MEMadr <= PC
MEMwe <= '0'
MEMdata <= don't care
load1
AC <= MEMq
MEMadr <= PC
MEMwe <= '0'
MEMdata <= don't care


inicial
PC <= x"00"
AC <= x"0000"
MEMadr <= x"00"
MEMwe <= '0'
MEMdata <= don't care
Ciclo 1
fetch1
IR <= MEMq
PC <= PC + 1
Ciclo 2
decode
CODOP = X"00"
CODOP = X"01"
CODOP = X"11"
CODOP = X"10"
MEMadr <= IR(7:0)
MEMwe <= '0'
MEMdata <= don't care
Ciclo 3
add0
store0
MEMadr <= IR(7:0)
MEMwe <= '1'
MEMdata <= AC
load0
jump
PC <= IR(7:0)
MEMadr <= IR(7:0)
MEMwe <= '0'
MEMdata <= don't
Ciclo 4
add1
AC <= AC + MEMq
MEMadr <= PC
MEMwe <= '0'
MEMdata <= don't care
store1
MEMadr <= PC
MEMwe <= '0'
MEMdata <= don't care
load1
AC <= MEMq
MEMadr <= PC
MEMwe <= '0'
MEMdata <= don't care

El procesador (versión 1_0)



Optimización

- **Reducción del número de ciclos por instrucción**

  Eliminando en lo posible estados que únicamente se utilizan para ordenar operaciones con la memoria y reutilizando los estados en el ciclo anterior para asignar valor a los buses de entrada a memoria.

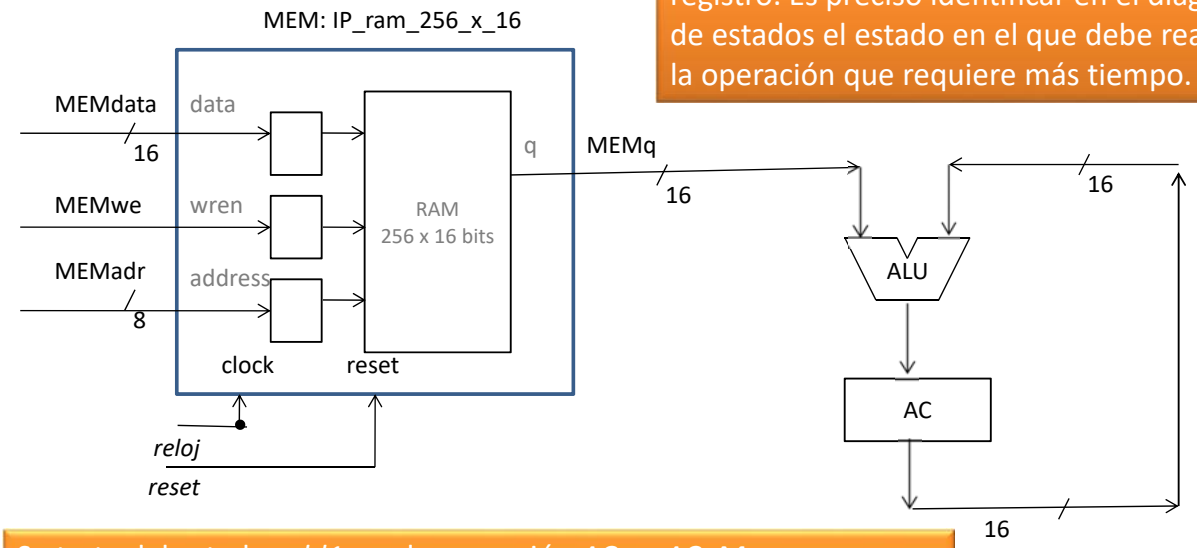
- **Aumento de la frecuencia máxima estimada**

  Segmentando el camino más largo de registro a registro.

Computador elemental

my_scomp_v0_0.vhd

La frecuencia máxima de funcionamiento depende del camino más largo de registro a registro. Es preciso identificar en el diagrama de estados el estado en el que debe realizarse la operación que requiere más tiempo.

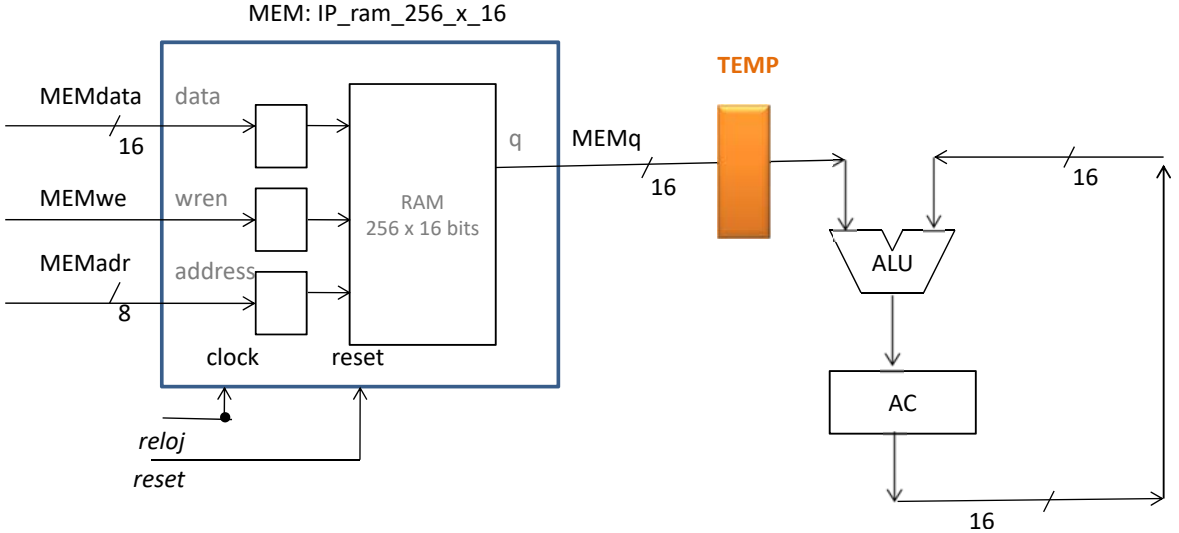


Se trata del estado *add1* con la operación *AC <= AC+Memq*, que implica leer de memoria y realizar la suma en el mismo ciclo.

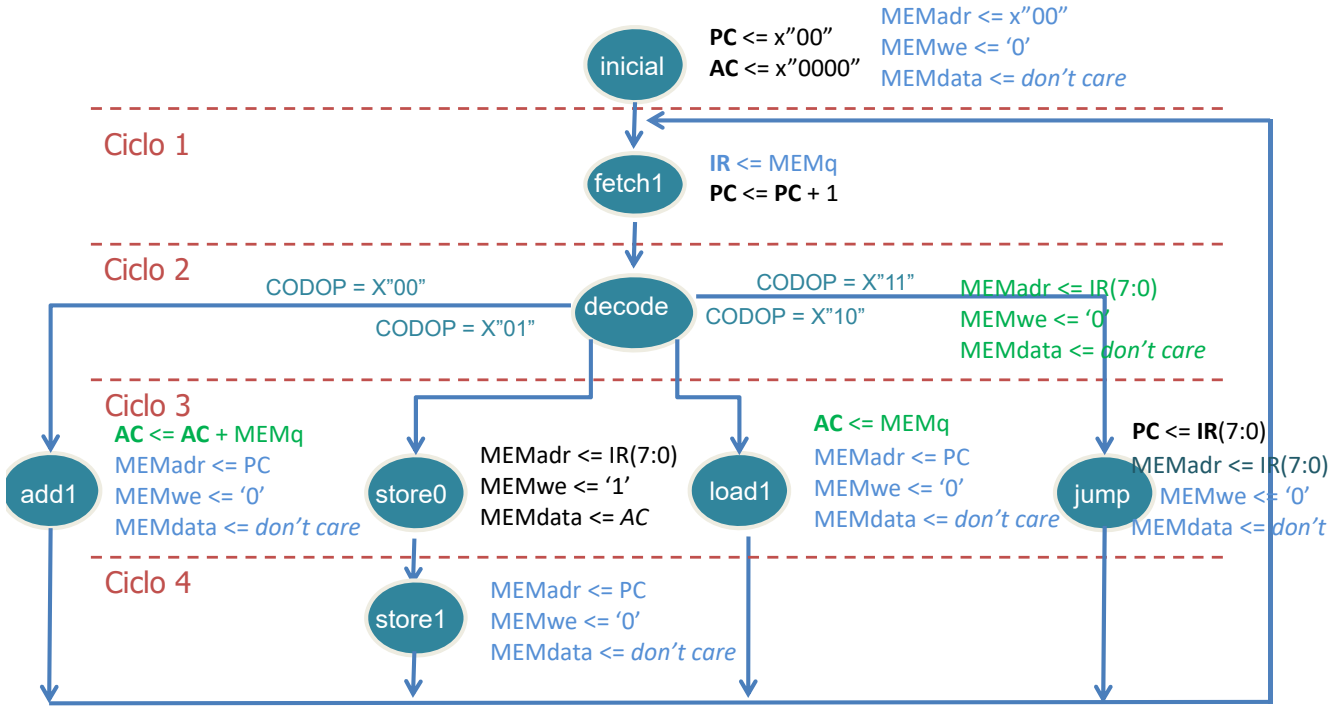
Computador elemental

my_scomp_v0_0.vhd

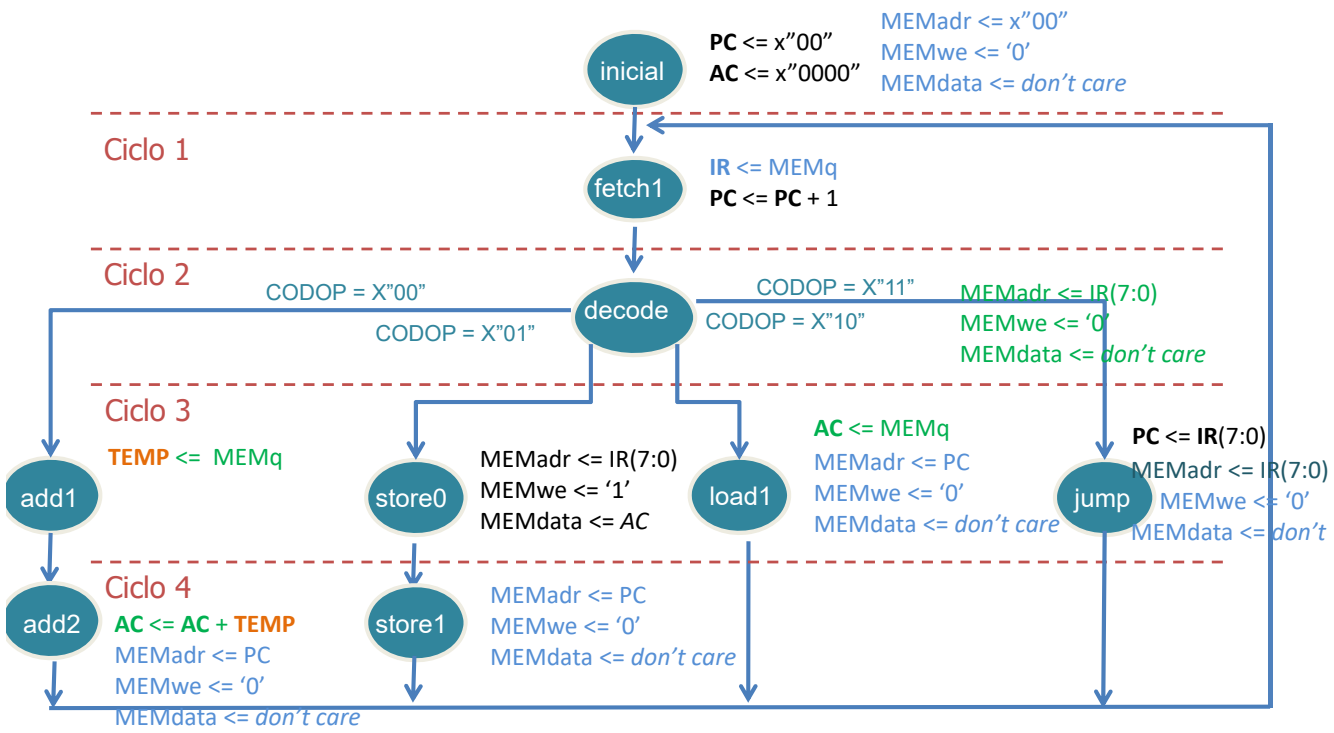
Se puede reducir el periodo mínimo de reloj para todos los estados intercalando un registro para dividir la operación en dos estados: en el primero se lee de memoria, en el segundo se realiza la suma. Se incrementa el número de ciclos de ejecución de la instrucción de suma, pero se disminuye el periodo mínimo de la señal de reloj.



El procesador (versión 1_0)



El procesador (versión 1_1)



Recomendaciones sobre ampliación del repertorio de instrucciones (I)

- *La ampliación debe realizarse a partir de la versión optimizada del procesador*
- *Comenzar la ampliación del repertorio añadiendo a la descripción original las instrucciones SUB y NAND, revisando el código VHDL del procesador:*
  - *Revisar la declaración del tipo enumerado STATE_TYPE ;*
  - *Incluir el CODOP (consignado en la Tabla 1) de la instrucción en selección del estado siguiente al estado de decodificación de la instrucción.*
  - *Añadir las asignaciones a registros ligados a cada estado de ejecución de la instrucción en la estructura CASE dentro de la sentencia condicional que identifica el flanco de reloj.*
  - *Añadir las asignaciones a los buses de memoria en la estructura CASE que genera lógica combinacional. Es preciso asignar valor a todas las señales (MEMadr, MEMwe, MEMdata) para cada estado.*

Recomendaciones sobre ampliación del repertorio de instrucciones (II)

- *Redactar un programa de prueba que utilice las instrucciones que se van añadiendo al repertorio y verificar mediante simulación su correcto funcionamiento.*
- *En las instrucciones de salto condicional (JNEG, JPOS y JZERO), hay que evaluar si la condición es cierta tanto en la parte de asignaciones a registros, como en la parte de asignaciones a buses de memoria.*
- *Para comprobar si el contenido del registro AC es negativo, se puede evaluar si es cierta la condición AC(15)='1' comprobando el bit de signo.*

Recomendaciones sobre ampliación del repertorio de instrucciones (III)

- *En la parte de asignaciones a buses de memoria, para la instrucción JNEG podría hacerse lo siguiente:*

```
when jneg =>
if AC(15)='1' then
   MEMadr <=         ;
   MEMwe <= '0';
   MEMdata <= (others =>'-');
else
   MEMadr <=         ;
   MEMwe <= '0';
   MEMdata <= (others =>'-');
end if;
```

Faltaría indicar qué valor se asigna al bus de direcciones de la memoria, en función de que deba realizarse el salto o no, teniendo en cuenta que en el estado siguiente (de captación de instrucción) debe tener en el bus de datos una instrucción que deberá leerse de la dirección de salto o de la dirección almacenada en el contador de programa.

Recomendaciones sobre ampliación del repertorio de instrucciones (IV)

- *En el paquete STD_LOGIC_ARITH hay definidos operadores aritméticos y relacionales sobre tipos de datos para números con signo y sin signo. El paquete STD_LOGIC_UNSIGNED permite que los objetos declarados como STD_LOGIC_VECTOR se consideren como números sin signo, y directamente se pueden chequear las siguientes condiciones para las instrucciones JPOS y JZERO:*

```
if (AC > 0 and AC(15)='0')...
if AC = 0 ...
```

- *Al redactar los programas de prueba, es preciso considerar tanto casos en los que la condición sea cierta, como casos en los que la condición sea falsa.*

Recomendaciones sobre ampliación del repertorio de instrucciones (V)

- *Para implementar las instrucción de desplazamiento, se pueden utilizar las siguientes funciones:*

```
function SHL(ARG: UNSIGNED; COUNT: UNSIGNED) return UNSIGNED;
function SHR(ARG: UNSIGNED; COUNT: UNSIGNED) return UNSIGNED;
```

donde el primer argumento es el vector a desplazar, y el segundo el número de posiciones (que se indica en la parte baja de la instrucción)

- *Al añadir las instrucciones IN y OUT es necesario declarar puertos de 8 bits (ports) de entrada y de salida accesibles desde exterior del computador.*

Referencias

[1] Hamblen, J.O., Hall T.S., Furman, M.D.: Rapid Prototyping of Digital Systems : SOPC Edition, Springer 2008 (*Recurso electrónico*)

[2] Guía de usuario de las memorias embebidas en FPGAs de Altera. https://www.altera.com/en_US/pdfs/literature/an/an207.pdf