Práctica 5 – MiniMIPS: un microprocesador MIPS simple

El objetivo de esta práctica es afianzar el conocimiento adquirido en la descripción estructural, comportamental y de máquinas de estados, través del diseño de un subconjunto de 8 bits del microprocesador MIPS de Patterson & Hennesy.

1. Arquitectura MIPS

La arquitectura MIPS32 es una arquitectura RISC de 32 bits simple. El subconjunto seleccionado de esta arquitectura utiliza **instrucciones codificadas con 32 bits** pero solo cuenta con **ocho registros de propósito general de 8 bits** denominados \$0-\$7. También utilizaremos un **contador de programa (PC) de 8 bits**. El registro \$0 siempre contiene el número 0 de manera invariable. El código y los datos se almacenarán, de manera conjunta, en una **memoria de 256 posiciones** y un **ancho de palabra de 8 bits**. La **escritura de la memoria** se realiza de manera **síncrona**, pero **la lectura es asíncrona**. **Las tres últimas posiciones** de memoria (253–255) **se encuentran mapeadas con los puertos de entrada/salida** del microprocesador. De esta manera, solo es necesario escribir un dato en estas posiciones para que se visualice en los **leds** o **displays de 7 segmentos** disponibles en la placa de prototipado. Las instrucciones disponibles son ADD, SUB, AND, OR, SLT, ADDI, BEQ, J, LB y SB.

Tabla 1. Conjunto de instrucciones MIPS (subconjunto soportado)

Instrucción	Función		Codificación	op	funct
add \$1, \$2, \$3	suma:	\$1 ← \$2 + \$3	R	000000	100000
sub \$1, \$2, \$3	resta:	\$1 ← \$2 – \$3	R	000000	100010
and \$1, \$2, \$3	and bit a bit:	\$1 ← \$2 and \$3	R	000000	100100
or \$1, \$2, \$3	or bit a bit:	\$1 ← \$2 or \$3	R	000000	100101
slt \$1, \$2, \$3	set si menor que:	\$1 ← 1 si \$2 < \$3 \$1 ← 0 en otro caso	R	000000	101010
addi \$1, \$2, inm	suma inmediato:	\$1 ← \$2 + inm*	I	001000	n/a
beq \$1, \$2, inm	salto si igual:	PC ← PC + inm*	I	000100	n/a
j destino	salto:	PC ← destino	J	000010	n/a
lb \$1, inm(\$2)	cargar byte:	\$1 ← mem[\$2 + inm]	I	100000	n/a
sb \$1, inm(\$2)	almacenar byte:	mem[\$2 + inm] ← \$1	I	101000	n/a

* Técnicamente, las direcciones MIPS indican bytes. Las instrucciones requieren una palabra de cuatro bytes y deben comenzar en direcciones que sean múltiplos de cuatro. Para utilizar de manera más efectiva los bits de las instrucciones de 32 bits de la arquitectura MIPS, las constantes de salto se especifican en palabras y deben multiplicarse por cuatro (desplazar 2 bits a la izquierda) para convertirlas en bytes.

El funcionamiento y codificación de cada instrucción aparece en la Tabla 1. Cada instrucción se codifica utilizando uno de los siguientes tres patrones: R, I y J. Las instrucciones de tipo R (basadas en *registros*) se utilizan para cálculos aritméticos, y especifican dos registros fuente y otro destino. Las instrucciones de tipo I se utilizan cuando se necesitan una constante de 16 bits (conocida como *valor inmediato*) y dos registros. Las instrucciones de tipo J (*saltos*) dedican casi toda la palabra de la instrucción al destino del salto de 26 bits. El formato de cada codificación se define en la Figura 1. Los seis bits más significativos de todos los formatos son el código de operación (op). Todas las instrucciones de tipo R comparten un op = 000000 y utilizan seis bits adicionales (funct) para diferenciar las operaciones.

Figura 1. Formatos de codificación de las instrucciones

Formato	Ejemplo	Codificación					
		6	5	5	5	5	6
R	add $rd $ra $rb	0	ra	rb	rd	0	funct
		6	5	5	16		
I	beq $a $b inm	op	ra	rb	inm		
		6	26				
J	j dest	op	dest				

Podemos escribir programas para el procesador MIPS en *lenguaje ensamblador*, donde cada línea del programa contiene una instrucción como ADD o BEQ. Sin embargo, el hardware MIPS finalmente debe leer el programa como una serie de números de 32 bits denominado *lenguaje máquina*. Un *ensamblador* automatiza el tedioso proceso de traducir de lenguaje ensamblador a lenguaje máquina utilizando la codificación definida en la Tabla 1 y la Figura 1. Escribir programas no triviales en lenguaje ensamblador es también tedioso, por lo que los programadores normalmente utilizan *lenguajes de alto nivel* como C. Un *compilador* traduce el programa del *código fuente* en lenguaje de alto nivel al *código objeto* en lenguaje máquina apropiado.

El listado 1 muestra un simple programa en C que calcula el *n*-ésimo número de Fibonacci f_n definido recursivamente por $n > 0$ como $f_n = f_{n-1} + f_{n-2}$, $f_{-1} = -1$, $f_0 = 1$. La variable **n** indica el número de Fibonacci a calcular.

Listado 1. Programa de Fibonacci en C

```
int fib(void)
{
        int n = 8;                      /* calcula el n-ésimo número de Fibonacci */
        int f1 = 1, f2 = –1;            /* últimos dos números de Fibonacci */

        while (n != 0) {            /* cuenta hasta n = 0 */
                f1 = f1 + f2;
                f2 = f1 – f2;
                n = n – 1;
        }
        return f1;
}
```

Listado 2. Programa de Fibonacci en lenguaje ensamblador

```
# fib asm
# Registros usados: $3: n $4: f1 $5: f2
# el contenido de los registros $3, $4 y $5 se enviará a los puertos de salida

fib:    addi $3, $0, 8          # inicializar n = 8
        sb $3, 255($0)          # almacenar n en la posición 255
        addi $4, $0, 1          # inicializar f1 = 1
        sb $4, 254($0)          # almacenar f1 en la posición 254
        addi $5, $0, –1         # inicializar f2 = –1
        sb $5, 253($0)          # almacenar f2 en la posición 253
bucle:  beq $3, $0, fin         # Fin del bucle si n = 0
        add $4, $4, $5          # f1 = f1 + f2
        sb $4, 254($0)          # almacenar f1 en la posición 254
        sub $5, $4, $5          # f2 = f1 – f2
        sb $5, 253($0)          # almacenar f2 en la posición 253
        addi $3, $3, –1         # n = n –1
        sb $3, 255($0)          # almacenar n en la posición 255
        j bucle                 # repetir hasta terminar
fin:    sb $4, 254($0)          # almacenar el resultado en la dirección 254
```

Listado 3. Programa de Fibonacci en código máquina

Instrucción	Codificación binaria				Codificación hexadecimal
addi $3, $0, 8	001000	00000	00011	0000000000001000	20030008
sb $3, 255($0)	101000	00000	00011	0000000011111111	A00300FF
addi $4, $0, 1	001000	00000	00100	0000000000000001	20040001
sb $4, 254($0)	101000	00000	00100	0000000011111110	A00400FE
addi $5, $0, –1	001000	00000	00101	1111111111111111	2005FFFF
sb $5, 253($0)	101000	00000	00101	0000000011111101	A00500FD
beq $3, $0, fin	000100	00011	00000	0000000000010000	1060001D
add $4, $4, $5	000000	00100	00101	00100 00000 100000	00852020
sb $4, 254($0)	101000	00000	00100	0000000011111110	A00400FE
sub $5, $4, $5	000000	00100	00101	00101 00000 100010	00852822
sb $5, 253($0)	101000	00000	00101	0000000011111101	A00500FD
addi $3, $3, –1	001000	00011	00011	1111111111111111	2063FFFF
sb $3, 255($0)	101000	00000	00011	0000000011111111	A00300FF
j bucle	000010		00000000000000000000000110		08000006
sb $4, 254($0)	101000	00000	00100	0000000011111110	A00400FE

2. Microarquitectura multiciclo MIPS

Implementaremos la microarquitectura MIPS multiciclo que aparece en el capítulo 5 del Patterson & Hennesy modificada para procesar 8 bits de datos. Esta microarquitectura se muestra en la Figura 2. Los rectángulos representan registros o memoria. Los rectángulos redondeados representan multiplexores. Los óvalos representan lógica de control. Las líneas finas indican señales individuales, mientras las líneas gruesas indican buses. La lógica de control y las señales se muestran en azul, mientras que la ruta de datos aparece en negro. Las señales de control generalmente alimentan las señales de selección de los multiplexores y de habilitación de los registros para indicar a la ruta de datos como ejecutar una instrucción.

NOTA: *Aunque el dibujo no lo muestre, todos los registros, la memoria (lectura asíncrona, pero escritura síncrona) y el controlador principal* ***comparten la misma señal de reloj****, y todos los registros y el controlador* ***disponen de señal de reset*** *para su inicialización.*

La ejecución de instrucciones generalmente fluye de izquierda a derecha. El contador de programa (PC) especifica la dirección de la instrucción. La instrucción se carga en el registro de instrucción de 32 bits (IR), byte a byte, durante cuatro ciclos desde la memoria. El campo *op* (bits 31:26 de la instrucción) se envía al controlador, que secuencia la ruta de datos a través de las operaciones correctas para ejecutar la instrucción. Por ejemplo, en una instrucción ADD, los dos registros fuentes se leen desde el *banco de registros* a los registros temporales *a* y *b*. El resultado es capturado por el registro *aluout*. En el tercer ciclo, el resultado se escribe en el registro destino apropiado en el banco de registros.

El controlador es una máquina de estados finita (FSM) que gestiona la selección de los multiplexores y la habilitación de los registros para secuenciar la ruta de datos. El diagrama de estados de la FSM se muestra en la Figura 3. **Cada estado muestra únicamente aquellas señales "activas", mientras que el resto de señales del procesador deben permanecer inactivas ('0' o "0…0")**. Como se ha indicado previamente, los primeros cuatro estados traen la instrucción desde memoria. La FSM determina, en base a *op*, como ejecutar la instrucción en particular.

Observar que el controlador produce una salida de dos bits *aluop*. La unidad *alucontrol* utiliza lógica combinacional para calcular la señal de 3 bits *alucontrol* a partir de *aluop* y *funct*, como indica la Tabla 2. *alucontrol* controla los multiplexores de la ALU para seleccionar el cálculo apropiado.

Tabla 2. Especificación de ALUControl

aluop	funct	alucontrol	Significado
00	X	010	ADD
01	X	110	SUB
10	100000	010	ADD
10	100010	110	SUB
10	100100	000	AND
10	100101	001	OR
10	101010	111	SLT
11	X	X	No definido

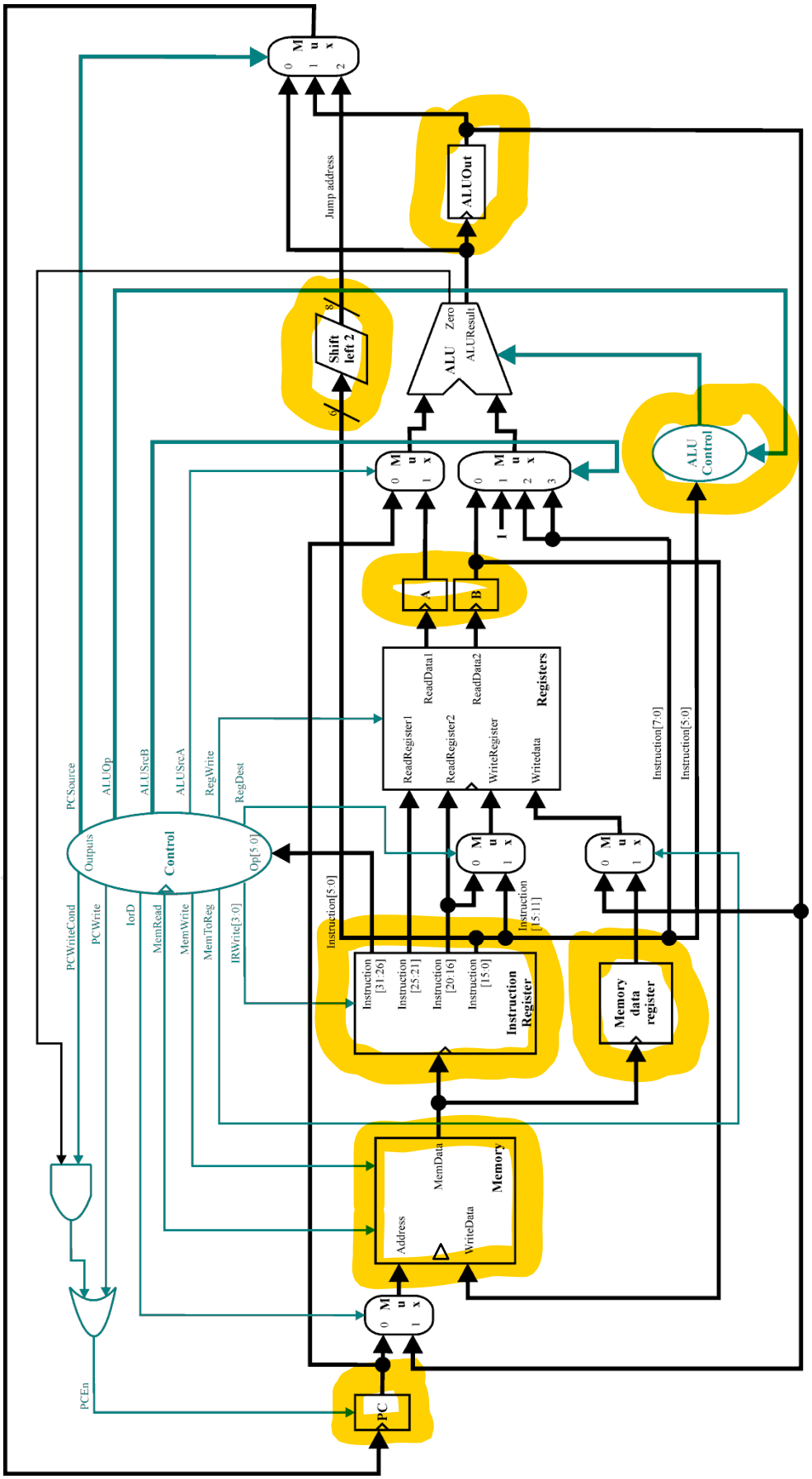

Figura 2. Microarquitectura MIPS multiciclo. Los componentes secuenciales aparecen identificados por medio del símbolo correspondiente a una señal de reloj (>).

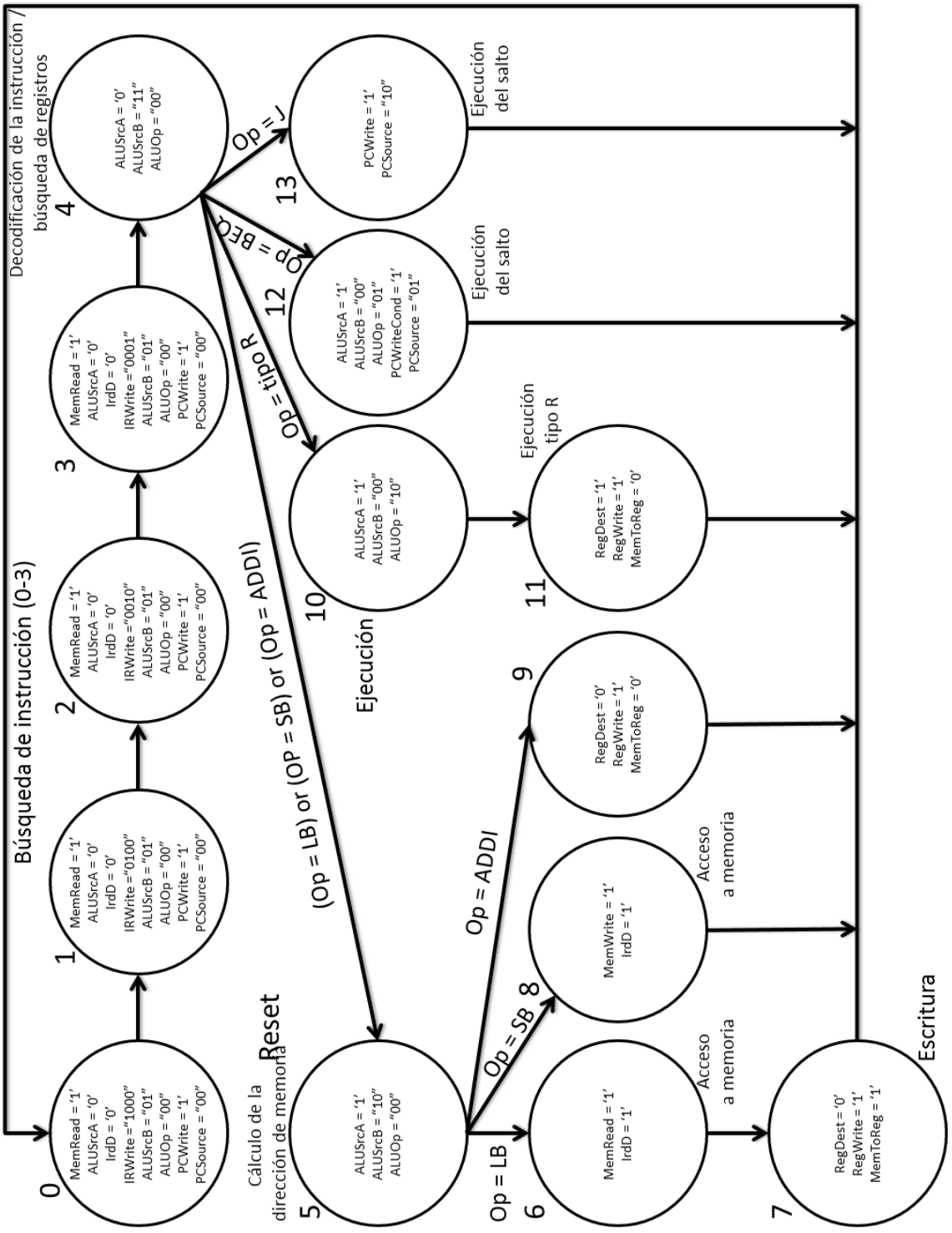

Figura 3. FSM de control del MIPS multiciclo
(el valor por defecto de todas las señales que no están incluidas en cada estado es '0' o "0…0").

Ejemplo de cómo ejecutar una instrucción SUB a partir de las Figuras 2 y 3.

El primer paso es obtener la instrucción de 32 bits. Esto lleva cuatro ciclos debido a que la instrucción se obtiene de una memoria con una interfaz de 8 bits. En cada ciclo, obtenemos un byte de la dirección de memoria indicada por el contador de programa, entonces la unidad aritmético-lógica incrementa el contador de programa en uno para apuntar al siguiente byte.

La búsqueda la realizan los estados 0-3 de la FSM de la Figura 3. Comencemos por el estado 0. El contador de programa (PC) contiene la dirección del primer byte de la instrucción. El controlador debe seleccionar *iord = 0* para que el multiplexor envíe esta dirección a la memoria. *memread* también debe activarse para que la memoria sitúe el byte en el bus *memdata*. Finalmente, *irwrite3* debe activarse para habilitar la escritura de *memdata* en el byte más significativo del registro de instrucción (IR).

Mientras tanto, necesitamos incrementar el contador de programa. Podemos hacerlo con la ALU indicando el PC como una entrada, '1' como la otra entrada, y ADD como operación. Para seleccionar el PC como primera entrada, *alusrca = 0*. Para seleccionar '1' como la otra entrada, *alusrcb = 01*. Para realizar la suma, *aluop = 00*, según la Tabla 2. Para escribir el resultado en el contador de programa al final del ciclo, *pcsource = 00* y *pcen = 1* (se consigue con *pcwrite = 1*).

Todas estas señales de control se indican en el estado 0 de la Figura 3. La habilitación del resto de registro se asume que es 0 si no se indica específicamente, y la señal de selección del resto de multiplexores es "no importa". Los siguientes tres estados son idénticos, excepto que escriben los bytes 2, 1 y 0 en el IR, respectivamente.

El siguiente paso es leer los registros fuente en el estado 4. Los dos registros fuente se indican en los bits 25:21 y 20:16 del IR. El banco de registros, que contiene estos registros, vuelca esta información en los registros *A* y *B*. Ninguna señal de control es necesaria para SUB (aunque el estado 4 realiza un cálculo de dirección de salto por si la instrucción es BEQ).

El siguiente paso es realizar la resta. De acuerdo al campo *op* (bits 31:26 de IR), la FSM salta al estado 10 porque SUB es una instrucción de tipo R. Los dos registros fuentes se seleccionan como entradas de la ALU poniendo *alusrca = 1* y *alusrcb = 00*. Indicando *aluop = 00*, el decodificador de control de la ALU selecciona la señal *alucontrol = 110*, resta. Otras instrucciones de tipo R se ejecutan exactamente igual, pero el decodificador recibe otro código *funct* (bits 5:0 de IR) y, por lo tanto, genera una señal *alucontrol* diferente. El resultado se almacena en el registro *ALUOut*.

Finalmente, el resultado debe escribirse en el banco de registros en el estado 11. Los datos vienen del registro *ALUOut* por medio de *memtoreg = 0*. El registro destino se especifica en los bits 15:11 de la instrucción, por lo que *regdst = 1*. *regwrite* debe activarse para realizar la escritura. El control de la FSM vuelve al estado 0 para buscar la siguiente instrucción.

3. Memoria implementada mediante un array

El objetivo de la práctica es implementar el subconjunto de la arquitectura MIPS presentada anteriormente.

Se deberá implementar una entidad, correspondiente al *top level* de la jerarquía, que incluirá todos los componentes previamente especificados y su interconexión (descripción estructural). Los multiplexores y puertas lógicas no es necesario especificarlos en ficheros diferentes, ya que pueden definirse directamente por medio de una descripción de flujo de datos. El código deberá estar convenientemente comentado.

La memoria, además de las entradas/salidas mostradas en la Figura 2, debe disponer de 3 puertos de salida (*port0*, *port1* y *port2*) de 8 bits. Estos puertos mapean directamente la memoria del microcontrolador con sus puertos de salida, con lo que el contenido de las posiciones de memoria 253–255 se visualizará directamente en los displays de la placa de prototipado. Para ello, además de incluir estos puertos en el componente que implementa la memoria se deberá hacer uso, en el nivel más alto de la jerarquía, del **gestor de displays 7 segmentos**, el **conversor de hexadecimal a 7 segmentos** y el **divisor de reloj** diseñados en la prácticas anteriores. Esto permitirá visualizar la evolución de la ejecución del algoritmo y verificar su correcto funcionamiento.

Adicionalmente, para permitir visualizar los datos en los displays, será necesario reducir la frecuencia de funcionamiento del microprocesador (el reloj de la placa funciona a 100MHz) por medio de un **divisor de reloj** (reducir a 100 Hz → 100 ciclos por segundo o similar). De otro modo, el algoritmo se ejecutará tan rápido que no podrán visualizarse los resultados intermedios.

Simular primeramente su correcto comportamiento y, posteriormente, comprobar su implementación en la placa de desarrollo.

NOTA: El programa que debe almacenar la memoria se suministra en el fichero **Fibonacci8.txt**. Este contenido se introducirá en el diseño de la memoria del microprocesador.

4. Memoria implementada mediante BlockRAM

Si se comprueba el resultado de la implementación, se podrá observar que la memoria descrita se ha implementado por medio de un conjunto enorme de biestables, lo que puede resultar ineficiente. Esto es debido a que la descripción de la memoria no se adapta a las características de los bloques de memoria (BRAM) disponibles en el interior de la FPGA. En concreto, la operación de lectura debería ser también síncrona, para permitir la utilización de estos bloques.

Para utilizar BRAM en la descripción del sistema, es necesario crear un nuevo componente de tipo IP Core tal y como se explica en la Práctica 3 (Diseño e implementación de un controlador de VGA).

NOTA: El programa que debe almacenar la memoria se suministra en el fichero **Fibonacci8.coe**. Este contenido se introducirá en el diseño del bloque de memoria.

La integración de este componente en el diseño presenta un pequeño problema, ya que la salida BRAM se encuentra "*registrada*", es decir, dispone de un registro en la salida que almacena y proporciona el valor esperado de la lectura de las posiciones de memoria accedidas. Así, **el resultado** de la operación **de lectura se obtiene en el siguiente ciclo** de reloj, y todas las señales de control se encuentran desfasadas.

Las opciones que pueden considerarse para solucionar este problema son, entre otras:

1. Introducir un **nuevo divisor de frecuencia** que genere una señal de habilitación para el bloque de memoria. Esta señal de habilitación funcionará al **doble de frecuencia que** la proporcionada al **resto del sistema**. Con ello, se obtendrá el resultado esperado en la salida de la memoria en el ciclo de reloj adecuado con respecto al sistema global.
2. Introducir **nuevos estados en el controlador para acceder a la memoria en lectura**. En el primero simplemente se activará MemRead, para disponer del resultado en el siguiente ciclo de reloj. En el siguiente estado, se utilizarán las mismas señales que se muestran en los estados descritos en la Figura 3, excepto que MemRead permanecerá inactiva (ya se recuperó el valor en el ciclo anterior).

Se pide realizar la especificación utilizando BRAM, simular su correcto comportamiento, y verificar su implementación en la placa de prototipado al igual que se realizó en el caso de utilizar un *array* para especificar la memoria.