

Universidad de Murcia  
Facultad de Informática

---

---

TÍTULO DE GRADO EN  
INGENIERÍA INFORMÁTICA

# Estructura y Tecnología de Computadores

Tema 4: Diseño de un microprocesador

Apuntes

CURSO 2010 / 11

---

---

VERSIÓN 2.1

Departamento de Ingeniería y Tecnología de Computadores

Área de Arquitectura y Tecnología de Computadores



## Índice general

4.1. Introducción . . . . .	2
4.2. Modelo del tiempo de ejecución de un programa . . . . .	2
4.3. Descomposición de la ejecución de instrucciones . . . . .	3
4.4. El camino de datos . . . . .	5
4.5. Control del camino de datos . . . . .	14
4.5.1. Señales de control del camino de datos . . . . .	14
4.5.2. Control de la ALU . . . . .	14
4.5.3. Implementación cableada de la unidad de control . . . . .	17
4.6. Metodología para la inclusión de nuevas instrucciones . . . . .	19
4.6.1. Análisis . . . . .	19
4.6.2. Diseño . . . . .	20
4.6.3. Ejemplo: bltzal . . . . .	22
<b>E4. Ejercicios</b>	<b>27</b>
E4.1. Calculo del tiempo de ejecución de programas sencillos en la implementación multiciclo . . . . .	27
E4.2. Solución a ejercicios seleccionados de la sección E4.1 . . . . .	33
E4.3. Resultados finales de ejercicios seleccionados de la sección E4.1 . . . . .	39
E4.4. Introducción de nuevas instrucciones en el camino de datos multiciclo . . . . .	40
E4.5. Solución a ejercicios seleccionados de la sección E4.4 . . . . .	42

## 4.1. Introducción

El objetivo de este tema es el diseño de un procesador sencillo capaz de ejecutar parte de las instrucciones del ISA de MIPS que se explican en el tema 3.

Por razones didácticas, las instrucciones cuya implementación abordaremos serán un subconjunto del repertorio completo. Sin embargo, este subconjunto incluirá instrucciones representativas de cada tipo, de forma que, con los conocimientos adquiridos en este tema, el alumno deberá ser capaz de modificar el procesador diseñado para incluir instrucciones adicionales. En concreto, las instrucciones que se implementarán serán:

- Instrucciones aritmético-lógicas: `add`, `sub`, `and`, `or` y `slt`.
- Instrucciones de carga y almacenamiento: `lw` y `sw`.
- Instrucciones de salto condicional: `beq`.
- Instrucciones de salto incondicional: `j`.

Para entender el diseño del procesador que se expone en este tema es imprescindible entender cómo se codifican las instrucciones anteriores. Esto está explicado en detalle en la sección correspondiente del tema 3.

Para construir el camino de datos, usaremos las unidades funcionales combinacionales y secuenciales que estudiamos en los temas 1 y 2 de la asignatura: puertas lógicas, multiplexores, sumadores, ALUs, extensores de signo, desplazadores, flip-flops, registros, bancos de registros, memorias, etc.

En la figura 1 se puede ver un primer esquema simplificado del procesador que implementaremos en este tema. En él se pueden ver las unidades funcionales que lo compondrán y las conexiones principales entre ellas.

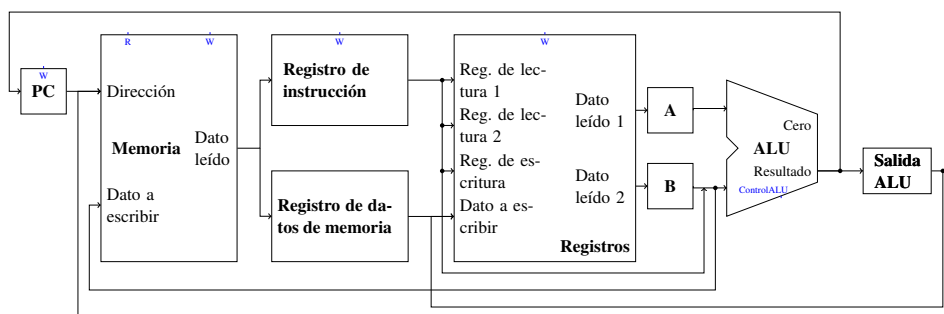


Figura 1: Visión de alto nivel de la implementación del procesador multiciclo.

## 4.2. Modelo del tiempo de ejecución de un programa

Un procesador es, básicamente, un sistema secuencial síncrono que ejecuta instrucciones. Al igual que los sistemas que hemos visto hasta ahora, tendrá unas entradas, un estado, unas salidas y una señal de reloj que dividirá la ejecución en ciclos. Sin embargo, a diferencia de los sistemas secuenciales síncronos estudiados en el tema 2, la cantidad de estados posibles en un procesador es astronómica. Por esta razón, las técnicas que vimos en el tema 2 no son directamente aplicables al diseño de un procesador, por muy simple que éste sea.

Como cabe esperar, hay distintas formas de enfocar el diseño de un procesador. Los diseños que veremos aquí son suficientemente sencillos para poderse estudiar en unas pocas semanas, mientras que los más complejos permiten conseguir un rendimiento más alto.

Para comparar el rendimiento de un diseño de procesador contra otro, usaremos el siguiente modelo del tiempo de ejecución de un programa:

$$T_{ejec} = N_{inst} \times CPI \times T_{ciclo}$$

donde:

$T_{ejec}$ : es el tiempo total que tarda en ejecutarse un programa dado con una entrada fija.

$N_{inst}$ : es el número de instrucciones *dinámicas* que se ejecutan. Se refiere a cuántas veces se ejecuta alguna instrucción, no cuántas instrucciones tiene el código del programa (número de instrucciones *estáticas*). Por ejemplo, una instrucción del programa que fuera parte de un bucle se ejecutaría varias veces, dando lugar a varias instrucciones dinámicas.

$CPI$ : es el número de ciclos que tarda en ejecutarse una instrucción en promedio (Ciclos por Instrucción). El CPI puede ser mayor que, igual a, o menor que 1.

$T_{ciclo}$ : duración del ciclo de reloj (tiempo que transcurre entre dos *flancos activos*); deberá ser lo suficientemente largo para permitir la estabilización de todas las señales de entrada a los elementos secuenciales del circuito.

Como podemos apreciar en la fórmula anterior, el tiempo de ejecución depende de tres factores. El primero de ellos ( $N_{inst}$ ) depende del ISA de la CPU y del compilador que genera las instrucciones de código máquina. En general, un compilador que optimice mejor producirá menor número de instrucciones para un programa<sup>1</sup>. También, un ISA que ofrezca instrucciones más complejas permitirá expresar los programas con menor número de ellas. Nosotros supondremos un ISA fijo (subconjunto del de MIPS que estudiamos en el tema 3).

El CPI y el tiempo de ciclo dependen de la implementación del procesador y están relacionados entre sí. Por ejemplo: es posible realizar una implementación *monociclo* en la que todas las instrucciones tardan exactamente un ciclo en ejecutarse ( $CPI = 1$ ). En este caso, el tiempo de ciclo tendrá que ser suficientemente largo como para realizar todo el trabajo necesario para **la instrucción más larga** (por lo que sobrará tiempo para las instrucciones más cortas y esto reducirá el rendimiento total).

En este tema veremos cómo construir un procesador que divide la ejecución de las instrucciones en pasos y ejecuta cada paso en un ciclo ( $CPI > 1$ ). Así, el tiempo de ciclo puede ser mucho más pequeño ya que sólo es necesario que dé tiempo a ejecutar **el paso más largo**. Cada instrucción o tipo de instrucción diferente tardará un número de ciclos distinto.

También es posible realizar implementaciones con un CPI menor que 1 utilizando técnicas que se verán en asignaturas posteriores de la carrera. El truco para conseguirlo consiste en ejecutar varias instrucciones a la vez, de forma que aunque la ejecución de una instrucción tarde más de un ciclo, el solapamiento de la ejecución hace posible ejecutar  $n$  instrucciones en menos de  $n$  ciclos.

En ocasiones resulta útil considerar directamente el número de ciclos ejecutados ( $N_{ciclos}$ ) en lugar de el número de instrucciones ( $N_{inst}$ ) y su CPI. Teniendo en cuenta que  $N_{ciclos} = N_{inst} \times CPI$ , el tiempo de ejecución se puede expresar mediante la siguiente fórmula:

$$T_{ejec} = N_{inst} \times CPI \times T_{ciclo} = N_{ciclos} \times T_{ciclo}$$

### 4.3. Descomposición de la ejecución de instrucciones

Vamos a descomponer la ejecución de las instrucciones a ejecutar por nuestro procesador en una serie de pasos. Posteriormente, veremos como cada uno de esos pasos se ejecutará en un ciclo. El funcionamiento del procesador será, en esencia, un bucle que va leyendo instrucciones y ejecutándolas paso a paso.

Un objetivo a tener en cuenta a la hora de realizar esta división en pasos/ciclos es tratar de realizar aproximadamente la misma cantidad de trabajo en cada paso. Recordemos que el tiempo de ciclo del procesador

<sup>1</sup>Sin embargo, minimizar el número de instrucciones no es siempre la mejor manera de optimizar el código, ya que hay que tener en cuenta también la duración de las instrucciones elegidas y otros factores aún más importantes como el aprovechamiento de la localidad de los accesos a memoria.

multiciclo estará determinado por la duración del paso más largo. Por tanto, los pasos más cortos tendrán tiempo sobrante en el ciclo durante el que el procesador no estará haciendo nada útil, lo cual reduce la eficiencia del diseño.

Supondremos que las únicas unidades funcionales que introducen un retardo significativo son la memoria, el banco de registros y la ALU. El resto de elementos (multiplexores, puertas lógicas, pequeños registros...) introducen retardos demasiado pequeños en comparación a los de los elementos que hemos mencionado como para tenerlos en cuenta en nuestra aproximación simplificada. El tiempo de ciclo vendrá dado por la latencia de la unidad funcional más lenta de estas tres. Es importante saber que se pueden utilizar varias unidades funcionales en el mismo ciclo, siempre y cuando se utilicen en paralelo y no en serie (es decir, la entrada de una de las unidades funcionales no puede depender de la salida de otra en el mismo ciclo). En ningún caso se podrá utilizar la misma unidad funcional para dos cosas en el mismo ciclo<sup>2</sup>. Por tanto en cada paso se podrá utilizar una vez como máximo cada una de las tres unidades funcionales principales.

Obviamente, el número de pasos y los pasos concretos a dar dependerán de cada instrucción. Sin embargo, veremos que todas las instrucciones comparten ciertas similitudes. Por ejemplo, los dos primeros pasos de la ejecución son comunes a todas ellas:

1. Lectura de la instrucción desde la memoria (memoria) y cálculo de la dirección de la instrucción siguiente (ALU).
2. Decodificación de la instrucción, lectura de operandos (banco de registros) y cálculo de la dirección de destino en caso de que la instrucción fuera un salto condicional (ALU).

Parece lógico que la búsqueda de la instrucción y su decodificación sean pasos comunes a todas las instrucciones. Sin embargo, no todas las instrucciones necesitan leer operandos (por ejemplo, la instrucción `j` ya tiene su único operando incrustado en la propia instrucción) ni todas las instrucciones necesitan calcular la dirección de un salto condicional (de hecho, sólo `beq` lo necesita en nuestro caso). Entonces, ¿por qué incluimos estas acciones en el segundo paso de todas las instrucciones?

Hay una razón fundamental para que los dos primeros pasos sean exactamente iguales para todas las instrucciones: hasta que no acaba el segundo paso no se ha decodificado la instrucción y, por tanto, no se puede tomar ninguna decisión basándose en qué instrucción se está ejecutando. En otras palabras, el procesador no sabe lo que está ejecutando hasta que llega al tercer ciclo.

El resto de acciones realizadas en el segundo paso aparte de la decodificación se realizan de manera *especulativa* (por si acaso resultan útiles luego). La razón para hacerlas durante ese paso es que las unidades funcionales necesarias para realizarlas están desocupadas, por lo que el coste de hacer este trabajo es pequeño<sup>3</sup>. Si no las realizáramos ahora, tendríamos que realizarlas en un ciclo posterior de la ejecución de las instrucciones que efectivamente las necesitaran, posiblemente incrementando el número total de ciclos necesarios para ejecutarlas.

El resto de pasos depende de la instrucción a realizar. En el caso de una instrucción aritmético-lógica (`add`, `sub`, `and`, `or` o `slt`), harán falta dos pasos más:

- 3a. Realización de la operación aritmético-lógica correspondiente (ALU).
- 4a. Escritura del resultado (banco de registros).

En el caso de una instrucción de lectura de memoria (`lw`), harán falta tres pasos más:

- 3b. Cálculo de la dirección de memoria del acceso (ALU).

<sup>2</sup>En realidad, el banco de registros se puede utilizar para leer dos registros y escribir otro. En cierto modo, podríamos considerar cada uno de los dos puertos de lectura y el puerto de escritura como tres unidades funcionales independientes (al menos a efectos de su utilización).

<sup>3</sup>No hay coste en términos de tiempo de ejecución, aunque sí hay un cierto coste en términos de energía.



y desactivando las señales de control apropiadas según la instrucción que se esté ejecutando y el momento actual de la ejecución.

Los elementos secuenciales del camino de datos se controlan mediante la metodología de sincronización por pulsos de reloj vista en el tema 2. Esto implica que las entradas a los bloques lógicos combinacionales serán valores que se calcularon en el ciclo anterior y el valor de salida de un bloque combinacional se escribirá al final del ciclo actual (cuando se alcanza el flanco activo del reloj). En particular, esto permite que la salida de un sistema combinacional actualice alguno de los elementos de estado que se usan para la entrada del mismo.

Dado que todos los bloques secuenciales están conectados a la misma señal de reloj, no incluiremos dicha señal en los esquemas de la implementación para incrementar la claridad de los diagramas.

En la figura 1 se pueden ver los principales componentes del camino de datos:

**Memoria:** supondremos que es una memoria que acepta direcciones de 32 bits y que es capaz de leer o escribir una palabra de 4 bytes en cada ciclo.

Tiene dos señales de control: *ActivarEscritura* (W) y *ActivarLectura* (R). Si ninguna de ellas está activada, la salida del puerto “Dato Leído” es indeterminada. Si *ActivarLectura* está activada, la salida del puerto “Dato Leído” será el valor almacenado en la posición de memoria indicada en el puerto “Dirección” en el último flanco activo. Y si *ActivarEscritura* está activa, se escribe en la dirección de memoria indicada en el puerto “Dirección” el dato indicado en el puerto “Dato a escribir”.

Almacena tanto el código del programa como los datos.

**Banco de registros (Reg):** es un banco de 32 registros que es capaz de leer dos registros y escribir otro en el mismo ciclo. El registro número 0 siempre almacena el valor 0 y se ignoran las escrituras a este registro.

Dispone de una señal de control, *ActivarEscritura* (W), la cual indica si se debe escribir en el registro indicado en el puerto “Reg. de escritura” el valor indicado en el puerto “Dato a escribir”. Los puertos “Dato leído 1” y “Dato leído 2” siempre valen el valor almacenado en los registros indicados por “Reg. de lectura 1” y “Reg de lectura 2”, respectivamente. Es posible hacer que coincidan los dos registros de lectura, e incluso leer y escribir en el mismo registro durante el mismo ciclo.

**ALU:** es una unidad aritmético-lógica capaz de sumar, restar, comprobar si el primer operando es menor que el segundo (*slt*) y hacer un OR o AND bit a bit.

Dispone de una señal de control de 3 bits, *ControlALU*, que permite seleccionar la operación a realizar según se muestra en la tabla 1.

<b>ControlALU</b>	<b>Operación</b>
000	AND
001	OR
010	Suma
110	Resta
111	<i>slt</i>

Tabla 1: Valores posibles de *ControlALU*.

Además del resultado de la operación, la ALU también proporciona una señal que indica si el resultado es igual a cero o no. Esta señal se utilizará a la hora de realizar comparaciones para la instrucción *beq*.

**PC:** registro de 32 bits usado para almacenar el valor actual del contador de programa. Dispone de una señal de control para habilitar o deshabilitar la escritura.

**Registro de instrucción (IR):** registro de 32 bits usado para almacenar la instrucción que se está ejecutando actualmente. Dispone de una señal de control para habilitar o deshabilitar la escritura. Este registro tiene varios puertos de salida, de forma que cada uno de ellos permite leer un subconjunto de los bits de la instrucción, que corresponderá con un campo del formato de instrucción.

**Registro de datos de memoria (MDR):** registro de 32 bits usado para almacenar un valor leído de memoria, de forma que esté disponible para su uso durante el ciclo siguiente al que fue leído.

Este registro es necesario porque la memoria tarda un ciclo entero en leer (o escribir) un valor. Recuerdese que no es posible utilizar dos de las unidades funcionales principales en serie (memoria, banco de registros o ALU), por lo que este registro se debe destinar a utilizar la salida de la memoria como entrada de la ALU o el banco de registros (pero durante el ciclo siguiente a la lectura de memoria).

Este registro no dispone de señal de habilitación de escritura, por lo que se escribe en él todos los ciclos. Es decir, el valor almacenado siempre será el valor que se ha leído de la memoria en el ciclo inmediatamente anterior (si es que se ha leído algún valor).

**Registro A y Registro B:** registros de 32 bits que almacenan los valores leídos del banco de registros, de forma que estén disponibles durante el ciclo siguiente para su uso por parte de la ALU o la memoria. Son necesarios por razones análogas al registro de datos de memoria, y al igual que éste no disponen de señal de habilitación de escritura.

**Salida ALU (ALUOut):** registro de 32 bits que almacena la salida de la ALU en el ciclo anterior para su uso por parte del banco de registros o la memoria. Análogo a los registros A y B y al MDR.

Como vemos, además de los tres componentes principales del camino de datos, es necesario añadir algunos registros auxiliares para almacenar valores temporales. Como regla general, al final de cada ciclo es necesario almacenar en elementos de estado (registros y memoria) los datos que vayan a utilizarse en ciclos posteriores. Los datos a utilizar por las siguientes instrucciones se almacenarán en elementos de estado visibles al programador (PC, banco de registros o memoria), mientras que los datos que se necesiten utilizar en ciclos posteriores de la misma instrucción se almacenarán en registros temporales (A, B, MDR o ALUOut).

Además de los componentes mencionados hasta ahora, nuestro camino de datos necesitará algunos desplazadores y extensores de signo para manipular las constantes de las instrucciones en formato I y J. Por último, también serán necesarios varios multiplexores para permitir diferentes conexiones entre las unidades funcionales según sea necesario para realizar el trabajo correspondiente a cada ciclo.

A continuación analizaremos la ejecución de cada paso de las instrucciones a implementar que mencionamos en la sección 4.3. Para cada paso, veremos cómo se deben conectar las unidades funcionales involucradas y añadiremos aquellos componentes que vayamos necesitando.

- Pasos comunes a todas las instrucciones:

- 1 **Búsqueda de instrucción e incremento del PC:** se envía el PC a memoria para la lectura de la instrucción y se incrementa el contenido del PC en 4.

$$IR = Memoria[PC]$$
$$PC = PC + 4$$

La figura 3 muestra los elementos del camino de datos necesarios para ejecutar esta etapa.

En el registro IR se almacenará la instrucción a ejecutar. Por tanto, es importante recordar cómo se codifican las instrucciones (véase el tema 3). La figura 4 muestra los formatos utilizados por las instrucciones que implementaremos.

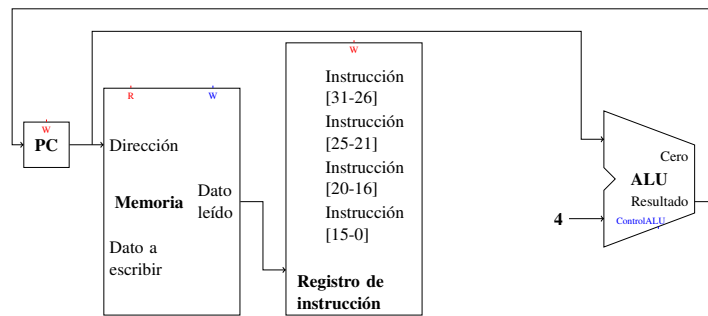


Figura 3: Conexión de los componentes necesarios para el paso 1

Tipo	Formato	Instrucciones																
R	<table border="1" style="width: 100%; text-align: center;"> <tr> <td style="width: 6%;">31</td> <td style="width: 16.5%;">op</td> <td style="width: 8.5%;">rs</td> <td style="width: 8.5%;">rt</td> <td style="width: 8.5%;">rd</td> <td style="width: 8.5%;">shamt</td> <td style="width: 12%;">func</td> <td style="width: 6%;">0</td> </tr> <tr> <td></td> <td>6 bits</td> <td>5 bits</td> <td>5 bits</td> <td>5 bits</td> <td>5 bits</td> <td>6 bits</td> <td></td> </tr> </table>	31	op	rs	rt	rd	shamt	func	0		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits		add, sub, slt, and y or
31	op	rs	rt	rd	shamt	func	0											
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits												
I	<table border="1" style="width: 100%; text-align: center;"> <tr> <td style="width: 6%;">31</td> <td style="width: 16.5%;">op</td> <td style="width: 8.5%;">rs</td> <td style="width: 8.5%;">rt</td> <td style="width: 58%;">imm</td> <td style="width: 6%;">0</td> </tr> <tr> <td></td> <td>6 bits</td> <td>5 bits</td> <td>5 bits</td> <td>16 bits</td> <td></td> </tr> </table>	31	op	rs	rt	imm	0		6 bits	5 bits	5 bits	16 bits		lw, sw y beq				
31	op	rs	rt	imm	0													
	6 bits	5 bits	5 bits	16 bits														
J	<table border="1" style="width: 100%; text-align: center;"> <tr> <td style="width: 6%;">31</td> <td style="width: 16.5%;">op</td> <td style="width: 77%;">j</td> <td style="width: 6%;">0</td> </tr> <tr> <td></td> <td>6 bits</td> <td>26 bits</td> <td></td> </tr> </table>	31	op	j	0		6 bits	26 bits		j								
31	op	j	0															
	6 bits	26 bits																

Figura 4: Formatos de instrucción MIPS.

2 **Decodificación de la instrucción y lectura de los operandos:** tanto en esta etapa como en la anterior todavía no se conoce qué instrucción se está tratando, por lo que sólo se deben realizar operaciones que sean comunes a todas las instrucciones (como leer la instrucción de memoria), u operaciones que puedan ser beneficiosas más adelante para algunas instrucciones y que no tengan efectos perjudiciales para ninguna otra.

Las dos operaciones que no son nunca nocivas y que es interesante realizar especulativamente para *adelantar trabajo* y reducir el número medio de ciclos necesarios para ejecutar una instrucción son:

- La lectura de los registros indicados por los campos **rs** y **rt**. Los valores leídos se almacenan en A y B, y si después no fueran necesarios bastaría con ignorarlos.
- Cálculo de la dirección destino de un salto condicional hipotético utilizando la ALU, almacenándola en el registro ALUOut.

Concretamente, las operaciones a realizar son <sup>4</sup> :

$$\begin{aligned}
 A &= \text{Reg}[\text{IR}[25-21]] \\
 B &= \text{Reg}[\text{IR}[20-16]] \\
 \text{ALUOut} &= \text{PC} + (\text{extender\_signo}(\text{IR}[15-0]) \ll 2)
 \end{aligned}$$

En este ciclo también se accede al campo **op** y al campo **func** de la instrucción en proceso de ejecución (almacenada en el IR) ya que las operaciones a realizar en los ciclos posteriores dependen del contenido de esos campos. La información de estos campos se envía a la unidad de control, que veremos en la sección 4.5.

<sup>4</sup> “X << 2” significa que X se desplaza dos bits a la izquierda (se multiplica por 4).

Obsérvese que, al realizar algunas acciones especulativas, en algunos casos se accederá a campos inexistentes en la instrucción que efectivamente se está ejecutando. En esos casos se utilizarán los bits de la instrucción que ocupen la posición que correspondería a dicho campo. Por ejemplo, si se está ejecutando una instrucción *j*, cuyo formato de instrucción no tiene campo *rs*, en este ciclo se leerá el registro identificado por los bits 25 al 21 de la instrucción, que son parte del campo *j*. Básicamente, se leerá un registro al azar y el valor leído será ignorado.

En realidad, aunque decimos que se lee en este ciclo el valor de los registros especificados por los campos *rs* y *rt*, esta lectura tiene lugar en todos los ciclos posteriores (siempre que el valor de los puertos de entrada “Reg. de lectura 1” y “Reg. de lectura 2” del banco de registros se mantengan estables). En los registros A y B se escriben los valores leídos en todos los ciclos, ya que no disponen de señal de habilitación de escritura.

La figura 5 muestra las conexiones entre los elementos del camino de datos necesarios para ejecutar esta etapa.

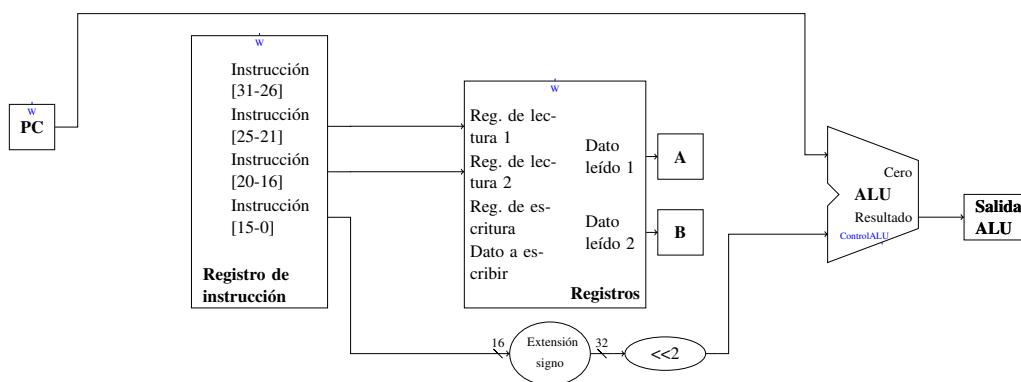


Figura 5: Conexión de los componentes necesarios para el paso 2

Para realizar estas operaciones, ha sido necesario añadir un extensor de signo y un desplazador de 2 bits al camino de datos de la figura 1.

■ Pasos de las instrucciones aritmético-lógicas:

Los pasos específicos a las instrucciones *add*, *sub*, *slt*, *and* y *or* se realizan de la siguiente manera:

3a **Ejecución:** en esta etapa la ALU se utiliza para realizar la operación correspondiente a la instrucción aritmético-lógica. Las cinco instrucciones aritmético-lógicas que estamos implementando se codifican usando el mismo valor para el campo *op*, por lo que la operación concreta a realizar depende del valor del campo *func*. El resultado de la operación se almacena en el registro ALUOut y está disponible al comienzo del siguiente ciclo.

$$ALUOut = A \text{ func } B$$

La figura 6 muestra las conexiones necesarias para este paso.

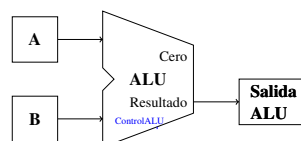


Figura 6: Conexión de los componentes necesarios para el paso 3a

4a **Finalización:** se escribe el resultado calculado por la ALU en el ciclo anterior en el banco de registros.

$$\text{Reg}[\text{Ir}[15-11]] = \text{ALUOut}$$

Las conexiones necesarias se pueden ver en la figura 7.

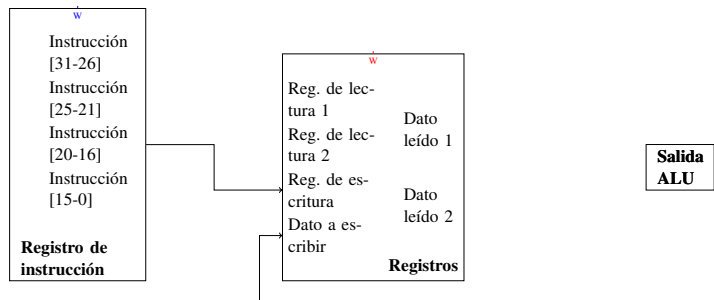


Figura 7: Conexión de los componentes necesarios para el paso 4a

- Pasos de las instrucciones de acceso a memoria:

El siguiente paso es común para las instrucciones `lw` y `sw`:

3b **Cálculo de la dirección:** la ALU se utiliza para calcular la dirección efectiva del acceso a memoria. Esta dirección se almacena en el registro `ALUOut` y se utilizará en el siguiente ciclo para direccionar la memoria.

$$\text{ALUOut} = A + \text{extender\_signo}(\text{IR}[15-0])$$

Se pueden ver las conexiones utilizadas para ejecutar este paso en la figura 8.

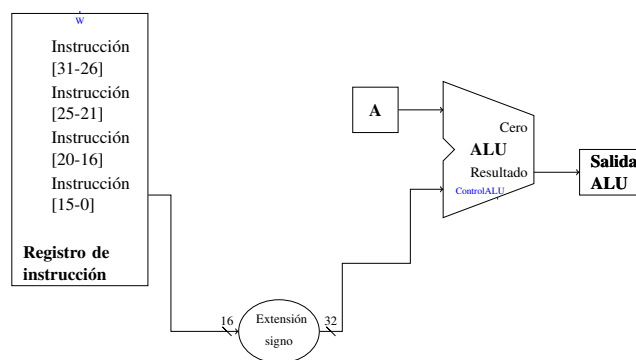


Figura 8: Conexión de los componentes necesarios para el paso 3b

- Pasos de las instrucciones de carga:

Las instrucción de carga, `lw`, necesita dos pasos adicionales:

4b **Lectura de memoria:** se carga en el registro `MDR` el valor leído de memoria.

$$\text{MDR} = \text{Memoria}[\text{ALUOut}]$$

Las conexiones a realizar para este paso se muestran en la figura 9.

5 **Finalización de la lectura:** en esta etapa se completan las instrucciones de carga escribiendo el valor leído en memoria en el banco de registros.

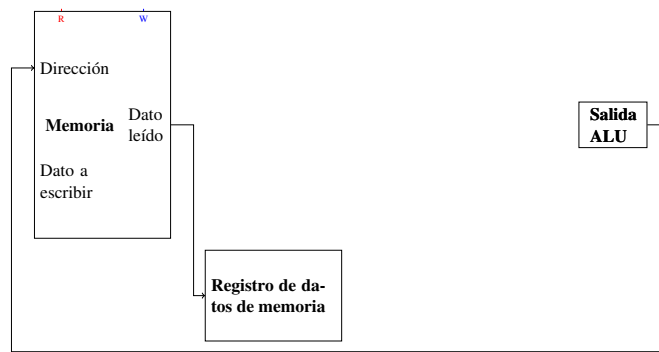


Figura 9: Conexión de los componentes necesarios para el paso 4b

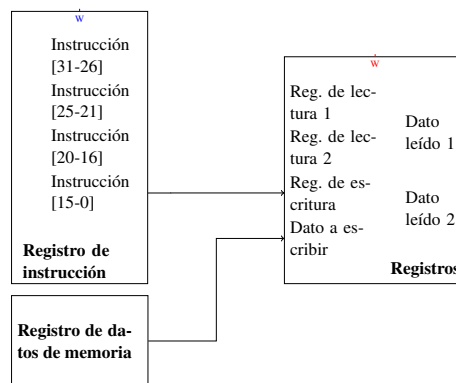


Figura 10: Conexión de los componentes necesarios para el paso 5

$$\text{Reg}[\text{IR}[20-16]] = \text{MDR}$$

La figura 10 muestra las conexiones entre las unidades funcionales implicadas en esta etapa.

- Pasos de las instrucciones de almacenamiento:

Por su parte, la instrucción *sw* requiere el siguiente paso:

- 4c **Escritura en memoria:** el registro B contiene el valor que se ha de almacenar en memoria. En esta etapa, se guarda dicho valor en la memoria.

$$\text{Memoria}[\text{ALUOut}] = \text{B}$$

Las conexiones para este paso se detallan en la figura 11.

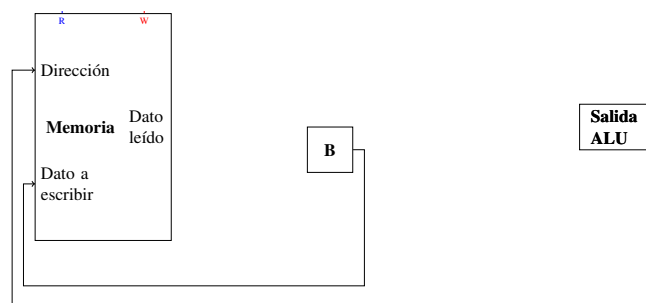


Figura 11: Conexión de los componentes necesarios para el paso 4c

- Pasos de las instrucciones de salto condicional:

La instrucción `beq` realiza el siguiente paso:

- 3c **Comparación y salto:** se comparan los valores almacenados en los registros A y B, para lo cual se realiza una resta con la ALU. Si son iguales (lo cual se detecta mediante la activación de la señal “cero” de la ALU), el PC se actualiza con el contenido del registro ALUOut, que fue calculado en el ciclo anterior. Esta señal se utilizará para controlar la señal de habilitación de escritura del PC.

`si (A == B) entonces PC = ALUOut`

En la figura 12 se muestran los componentes necesarios para esta acción y las conexiones entre ellos.

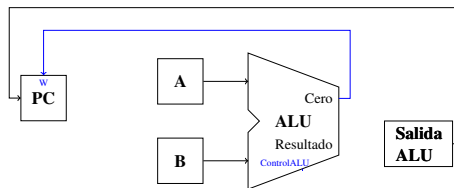


Figura 12: Conexión de los componentes necesarios para el paso 3c

- Pasos de las instrucciones de salto incondicional:

Finalmente, la instrucción `j` tiene un paso propio:

- 3d **Salto:** En este caso no es necesario usar la ALU y el PC se sustituye directamente con la dirección del salto incondicional, que se obtiene concatenando los 4 bits más significativos del registro PC con los 26 bits menos significativos del registro de instrucción desplazados 2 bits a la izquierda.

$$PC = (PC[31-28] \ll 28) \mid (IR[25-0] \ll 2)$$

Para realizar esta operación es necesario añadir un nuevo desplazador. Las conexiones resultantes se pueden ver en la figura 13.

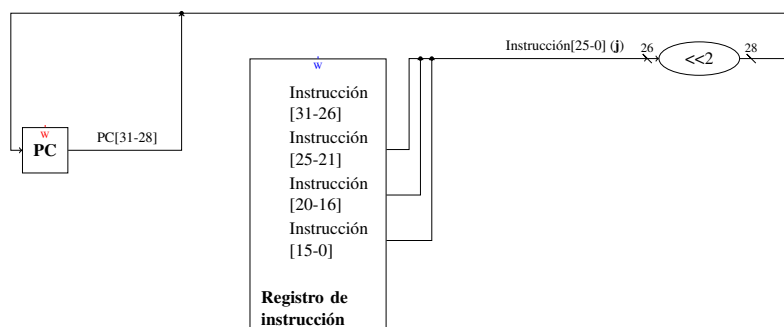


Figura 13: Conexión de los componentes necesarios para el paso 3d

Las figuras 3, 5, 6, 7, 8, 9, 10, 11, 12 y 13 muestran cómo conectar las unidades funcionales entre sí para realizar cada uno de los pasos. En cierto modo, podemos considerar que la figura 1 sería una simplificación del resultado de superponer las figuras anteriores en un solo circuito.

Como se puede ver, las conexiones que necesitamos que existan entre los puertos de las unidades funcionales dependen del ciclo actual y de la instrucción que se esté ejecutando en cada momento. Para acomodar todas

estas necesidades, tenemos que añadir algunos multiplexores en aquellos puertos de entrada que queramos que reciban los datos de varios sitios.

Cada multiplexor requerirá una señal de control para elegir, en cada caso, qué entrada utilizar. En la sección 4.5 veremos cómo se manejan estas señales.

En particular, necesitamos los multiplexores que se detallan a continuación:

- Un multiplexor conectado al puerto “Dirección” de la memoria para seleccionar la dirección de memoria a la que se accede, que puede ser:
  - El valor del registro PC (paso 1).
  - El valor del registro ALUOut (pasos 4b y 4c).
- Un multiplexor conectado al puerto “Reg. de escritura” para seleccionar los bits de la instrucción que se corresponden con el campo que indica el registro a escribir, que pueden ser:
  - Los bits del campo **rd** (paso 4a).
  - Los bits del campo **rt** (paso 5).
- Un multiplexor conectado al puerto “Dato a escribir”, que puede ser:
  - El valor del registro ALUOut (paso 4a).
  - El valor del registro MDR (paso 5).
- Un multiplexor en la primera entrada de la ALU, que puede tomar su valor de:
  - El registro PC (pasos 1 y 2).
  - El registro A (pasos 3a, 3b y 3c).
- Un multiplexor en la segunda entrada de la ALU, que puede tomar su valor de:
  - El registro B (pasos 3a y 3c).
  - La constante cableada 4 (paso 1).
  - El campo **imm** de la instrucción actual después de extenderlo a 32 bits (paso 3b).
  - El campo **imm** de la instrucción actual después de extenderlo a 32 bits y multiplicarlo por 4 (paso 2).
- Un multiplexor a la entrada del registro PC que selecciona de dónde procede la dirección a escribir:
  - Del puerto “Resultado” de la ALU (paso 1).
  - Del registro ALUout (paso 3c).
  - De los 26 bits menos significativos del registro IR, precedidos de los 4 bits más significativos del PC actual y multiplicado el resultado por 4 (paso 3d).

En la figura 14 se puede ver el camino de datos después de añadirle los multiplexores y conexiones que hemos mencionado.

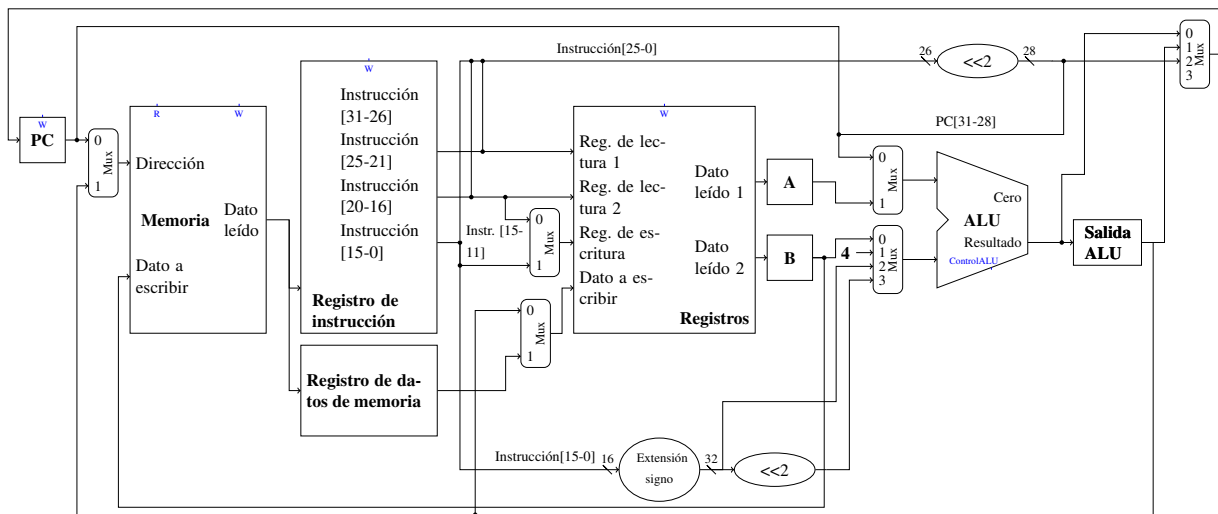


Figura 14: Visión completa de la implementación del camino de datos.

## 4.5. Control del camino de datos

En la figura 14 hemos mostrado el camino de datos completo de nuestro procesador. Para que dicho procesador sea capaz de ejecutar las instrucciones, es necesario que siga el conjunto de pasos que hemos descrito en las secciones 4.3 y 4.4. En la figura 2 se puede ver un resumen de los pasos y las transiciones entre ellos. En cada paso, tendremos que asegurar que el valor de las señales de control de los multiplexores y de todas las unidades funcionales sea el adecuado.

La unidad de control es el componente del procesador que se encarga de activar y desactivar las señales de control para dirigir el funcionamiento del camino de datos. Las señales de control incluyen el control de los multiplexores, las señales de habilitación de escritura de los registros y todas las entradas de control de todas las unidades funcionales.

### 4.5.1. Señales de control del camino de datos

Las señales de control de nuestro camino de datos, y su efecto según su valor en cada momento, serán las que se muestran en la tabla 2.

### 4.5.2. Control de la ALU

Controlar la ALU significa generar sus tres bits de entrada de control para realizar la operación deseada según la tabla 1. Como se puede ver en la sección 4.4, la operación a realizar por la ALU depende casi siempre del paso actual únicamente, excepto en el paso 3a que depende también del valor del campo **func** de la instrucción que se esté ejecutando.

Recordemos que todas las instrucciones aritmético-lógicas que implementa nuestro procesador comparten el mismo código de operación (**op** = 0) en su codificación en código máquina MIPS, y sólo se diferencian en el valor del campo **func** (véase el tema 3 para más detalles).

Para simplificar la construcción de la unidad de control, vamos a crear una unidad de control auxiliar para la ALU. El comportamiento que queremos que tenga esta unidad de control es el siguiente:

- En los pasos 1, 2 y 3b, debe hacer que la ALU realice una suma.
- En el paso 3c, debe hacer que la ALU realice una resta,

Señal	Valor	Efecto
ALUOp	00	La ALU realiza una operación de suma.
	01	La ALU realiza una operación de resta.
	10	El campo función del registro IR determina la operación a realizar por la ALU.
EscrIR	0	Ninguno.
	1	La salida de memoria se escribe en el registro IR.
EscrMem	0	Ninguno.
	1	El valor almacenado en la posición de memoria designada por la dirección se reemplaza por el valor de la entrada de datos.
EscrPC	0	Ninguno.
	1	Se escribe el PC, su origen estará controlado por FuentePC.
EscrPCCond	0	Ninguno.
	1	Se escribe el PC si la señal Cero de la ALU está activa.
EscrReg	0	Ninguno.
	1	El registro se actualiza con el valor a escribir.
FuentePC	00	La salida de la ALU se envía para su escritura en el PC.
	01	El contenido de ALUOut se envía para su escritura en el PC.
	10	El campo <b>j</b> desplazado dos bits a la izquierda y concatenado con los 4 bits más significativos del registro PC se envía para su escritura en el PC.
IoD	0	El PC suministra la dirección para acceder a memoria.
	1	ALUOut proporciona la dirección para acceder a memoria.
LeerMem	0	Ninguno.
	1	El valor contenido en la posición de memoria designada por la dirección se coloca en la salida de lectura.
MemAReg	0	El valor de entrada del banco de registros proviene del registro ALUOut.
	1	El valor de entrada del banco de registros proviene del registro MDR.
RegDest	0	El identificador del registro destino viene determinado por el campo <b>rt</b> .
	1	El identificador del registro destino viene determinado por el campo <b>rd</b> .
SelALUA	0	El primer operando de la ALU es el PC.
	1	El primer operando de la ALU es el registro A.
SelALUB	00	El segundo operando de la ALU es el registro B.
	01	El segundo operando de la ALU es la constante 4.
	10	El segundo operando de la ALU son los 16 bits menos significativos del registro IR, extendidos de signo.
	11	El segundo operando de la ALU son los 16 bits menos significativos del registro IR, extendidos de signo y desplazados 2 bits a la izquierda.

Tabla 2: Señales de control del camino de datos y su significado.

- En el paso 3a, debe realizar una operación que depende de los bits del campo **func** de la instrucción actual. La operación a realizar en cada caso se muestra en la tabla 3.

Instrucción	Campo func	Operación	ControlALU
and	100100	AND bit a bit	000
or	100101	OR bit a bit	001
add	100000	Suma	010
sub	100010	Resta	110
slt	101010	Comparación “menor que”	111

Tabla 3: Operación a realizar por la ALU dependiendo del campo **func**, en el caso de las instrucciones aritmético-lógicas (paso 3a).

Implementaremos esta unidad de control auxiliar como un sistema lógico combinacional que tendrá dos bits de entrada además de los seis bits del campo **func** de la instrucción actual, y tres bits de salida que corresponderán con la señal **ControlALU** definida en la tabla 1. Estos dos bits, a los que llamaremos **ALUOp** serán una más de las señales de control generadas por la unidad de control principal que implementaremos en la sección 4.5.3. El comportamiento que necesitamos que siga la unidad de control de la ALU en función de estos dos bits queda reflejado en la tabla 4.

ALUOp	Acción	ControlALU
00	Suma incondicional	010
01	Resta incondicional	110
10	Según campo <b>func</b>	Ver tabla 3

Tabla 4: Acción de los bits ALUOp.

Para diseñar la correspondiente función combinatoria de 8 variables de entrada y 3 de salida pueden utilizarse las técnicas vistas en el tema 1 de la asignatura. Sin embargo, dada la especialmente simple configuración de nuestra función, resolveremos el problema usando un solo multiplexor y una sencilla combinación de puertas. Dicha solución se muestra en la figura 15. Se puede comprobar de manera trivial que el circuito correspondiente funciona de la manera indicada en las tablas 3 y 4. Obsérvese que, en realidad, solo 4 de los 6 bits del campo **func** tienen influencia en el valor de **ControlALU**.

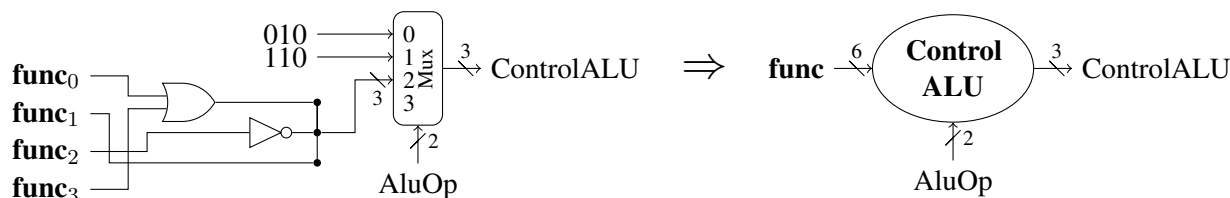


Figura 15: Circuitería de control de la ALU.

### 4.5.3. Implementación cableada de la unidad de control

El valor de las señales de control depende únicamente del paso actual de la ejecución<sup>5</sup>. Por otro lado, la transición de un paso a otro depende de la instrucción que se está ejecutando en cada momento (más concretamente: depende del valor del campo **op** de la última instrucción cargada en el registro de instrucción) y en cada ciclo se debe ejecutar un paso diferente.

Teniendo en cuenta lo anterior, parece evidente que una forma bastante directa de implementar la unidad de control es mediante un sistema secuencial síncrono modelado mediante una máquina de Moore. Las entradas del sistema secuencial serán los bits del campo **op** de la instrucción almacenada en el registro de instrucción, las salidas serán los valores de las señales de control, y tendremos un estado por cada uno de los pasos definidos en la sección 4.4. El autómata que describe el sistema secuencial síncrono a implementar se muestra en la figura 16. Este autómata está basado en el diagrama de la figura 2. El autómata generaría el valor de todas y cada una de las señales de control en cada estado, pero en la figura se muestran únicamente las señales relevantes para la operación que se está realizando en cada estado. Tanto en el autómata de la figura como en cualquier figura similar se supondrá que el valor de las señales de control que habiliten escrituras en registros o en la memoria (por ejemplo, *EschrPCCond*) será 0 cuando no se especifique explícitamente. En el caso de otras señales de control (por ejemplo, las de los multiplexores) que no se especifiquen, se supondrá que su valor es irrelevante para la ejecución correcta de las operaciones.

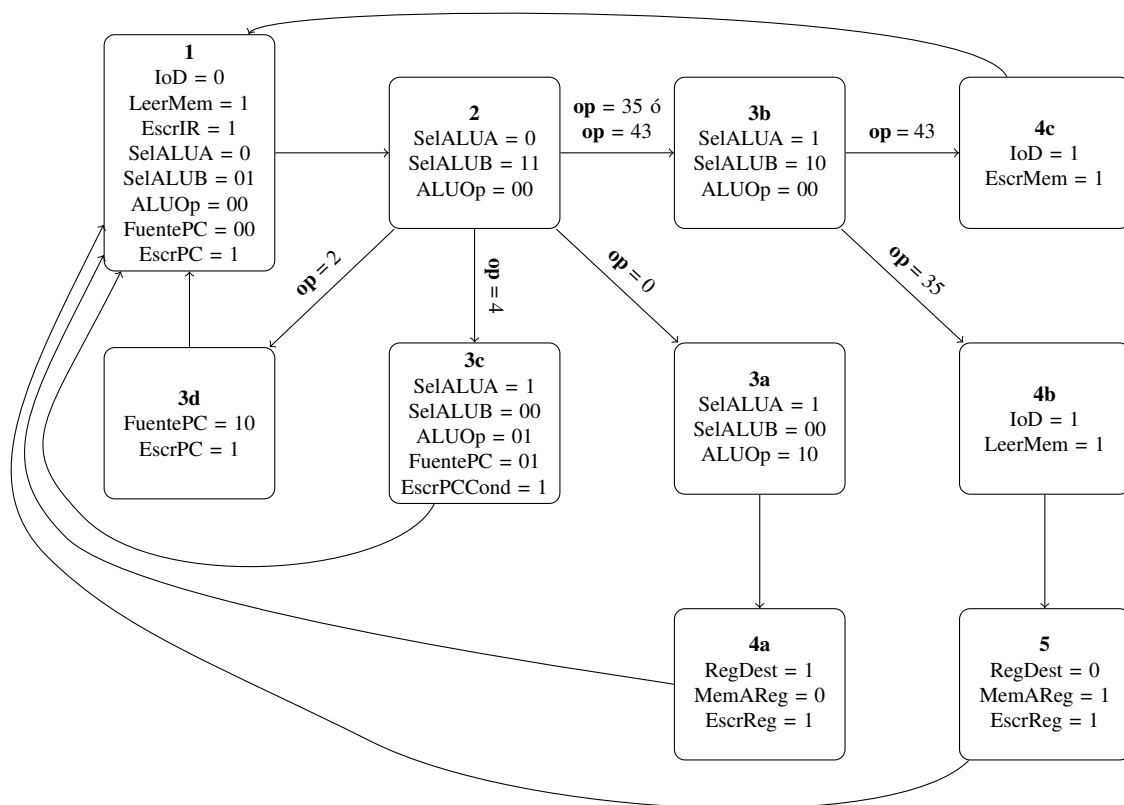


Figura 16: Autómata que describe la unidad de control del microprocesador. Las salidas de cada estado son todas las señales de control de la tabla 2. Aquellas que no se mencionen en algún estado se asume que toman el valor 0.

<sup>5</sup>Estrictamente hablando, también dependen del valor del campo **func** de las instrucciones de tipo R, que determina la operación a realizar por la ALU. Pero podemos ignorar este detalle gracias a que la unidad de control que diseñamos en esta sección no controlará directamente la ALU, sino que controlará a la unidad de control de la ALU que se explica en la sección 4.5.2

Como se ve en la figura 16, los dos primeros estados son comunes a todas las instrucciones, como ya se mencionó antes. Sólo a partir del tercer ciclo se puede transitar a un estado diferente según qué instrucción se esté ejecutando, dado que el valor del campo **op** no está disponible hasta el final del segundo ciclo. En total tenemos un autómata de 10 estados, por lo que necesitaremos un registro de 4 bits para almacenar el estado actual.

El esquema de la implementación de la unidad de control es el mismo que el de cualquier sistema secuencial síncrono diseñado a partir de una máquina de Moore, y se puede ver en la figura 17.

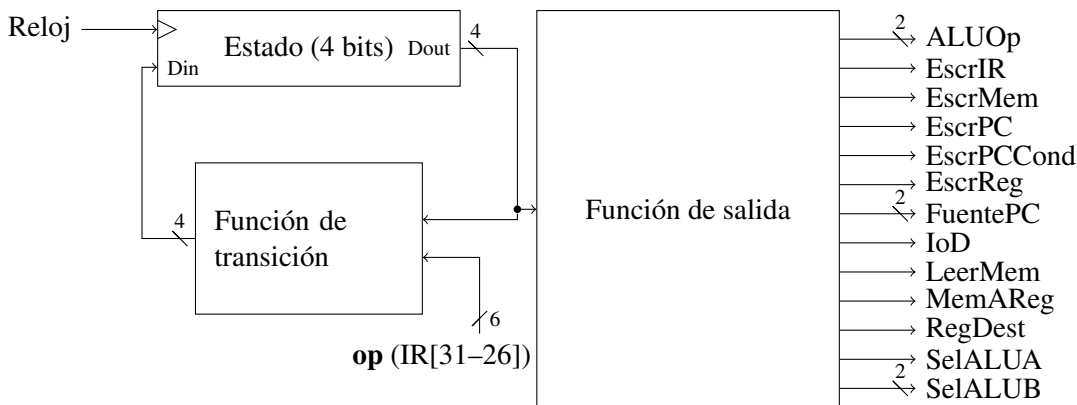


Figura 17: Esquema de implementación de la unidad de control.

Una vez que tenemos diseñada la unidad de control, podemos dibujar un esquema completo de la implementación del procesador como el que se muestra en la figura 18. Obsérvese que se han añadido dos puertas lógicas para implementar la semántica de las señales de control EscrPC y EscrPCCond (con estas puertas, se escribe en el registro PC si la señal EscrPC está activa o si la señal EscrPCCond está activa y además la señal Cero de la ALU está activa).

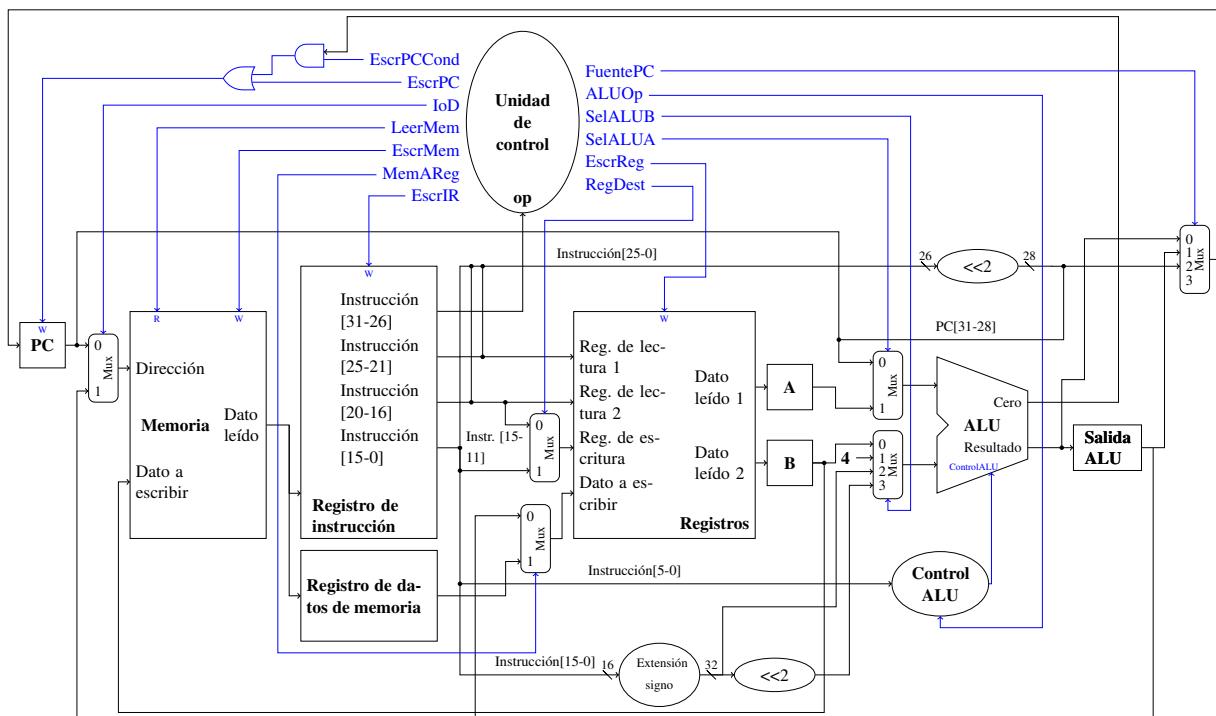


Figura 18: Visión completa de la implementación del procesador, incluyendo la unidad de control.

## 4.6. Metodología para la inclusión de nuevas instrucciones

En esta sección veremos la metodología a seguir a la hora de ampliar la implementación del procesador que hemos presentado en las secciones anteriores para que sea capaz de ejecutar nuevas instrucciones. Para presentar esta metodología, utilizaremos un ejemplo: añadiremos una instrucción `push` al pequeño repertorio de instrucciones presentado en la sección 4.1.

La metodología que seguiremos se compone de dos fases principales: análisis y diseño. Estas fases a su vez se dividen en los siguientes pasos:

### 1. Análisis:

- a) Especificación precisa de la semántica de la instrucción.
- b) Identificación del trabajo a realizar por cada unidad funcional principal.
- c) Establecimiento del orden de precedencia entre las distintas tareas a realizar.

### 2. Diseño:

- a) Definición de la codificación de la instrucción.
- b) División del trabajo en ciclos.
- c) Extensión del camino de datos.
- d) Extensión del control.

A continuación, veremos cada uno de estos pasos en mayor detalle y mostraremos cómo se aplican a la instrucción de ejemplo. A la hora de abordar un problema de este tipo, es conveniente seguir esta serie de pasos en el orden indicado, aunque es posible que haya que volver a considerar las decisiones de un paso anterior una vez que se haya obtenido nueva información en un paso posterior (especialmente durante la fase de diseño).

### 4.6.1. Análisis

El objetivo principal del análisis de la instrucción es identificar qué unidades funcionales se necesitan para ejecutar la instrucción a implementar y repartir el uso de estas unidades en varios ciclos.

#### Especificación semántica precisa

Muy frecuentemente, partiremos de una descripción incompleta del trabajo a realizar por la nueva instrucción. Esta descripción puede presentarse incluso de forma verbal, como por ejemplo:

“La instrucción `push` almacena el valor de un registro en la pila”.

En este paso debemos traducir dicha especificación a una forma simbólica precisa para evitar cualquier tipo de ambigüedad en la definición. Esta especificación debe capturar todos los efectos de la instrucción sobre el estado del computador visible al programador. Es decir, debe reflejar cómo afecta al contenido del banco de registros, la memoria y el contador de programa (no debe incluir detalles sobre los registros internos de la implementación como, por ejemplo, el registro `ALUOut`). Para realizar esta especificación tendremos que hacer una primera descomposición de la instrucción en acciones individuales.

En el caso de la instrucción `push`, como sabemos, el trabajo a realizar para apilar un registro consiste en dos acciones: hacer sitio en la pila (decrementando `$sp`) y copiar el valor del registro a la nueva cima de la pila. Utilizando una notación simbólica expresaríamos el funcionamiento de `push $x` de la siguiente manera:

```

$29 ← $29 - 4
Memoria[$29] = $x

```

En la notación anterior, el uso de  $\$29$  en la segunda línea se refiere al contenido de  $\$29$  después de haber sido actualizado en la primera línea.

### Identificación del trabajo de las unidades funcionales principales

Nuestro camino de datos incluye tres unidades funcionales principales: la memoria, el banco de registros y la ALU. En este paso, identificaremos qué acciones realiza cada una de estas unidades. En el caso de la instrucción `push`:

- Banco de registros:
  - Leer  $\$29$ .
  - Leer  $\$x$ .
  - Escribir  $\$29 - 4$  en  $\$29$ .
- ALU:
  - Calcular  $\$29 - 4$ .
- Memoria:
  - Escribir  $\$x$  en  $\$29 - 4$ .

Podemos observar que al asignar trabajo a las unidades funcionales hemos refinado la división en acciones realizada en el paso anterior. Por ejemplo, la acción “ $\$29 \leftarrow \$29 - 4$ ” se ha dividido en dos acciones: el cálculo de “ $\$29 - 4$ ” realizado en la ALU y la asignación del resultado a  $\$29$ , realizado en el banco de registros.

En este paso también es posible identificar la necesidad de nuevas unidades funcionales si vemos que ninguna de las disponibles puede realizar alguna de las acciones necesarias. No es el caso de la instrucción `push`, cuyas acciones se pueden realizar todas directamente con las unidades funcionales disponibles. Sería necesario añadir, por ejemplo, un multiplicador si necesitáramos realizar una multiplicación en alguno de los pasos.

### Establecimiento del orden de precedencia entre las distintas tareas a realizar

El objetivo de este paso es identificar las relaciones de dependencia entre las acciones asignadas a las unidades funcionales principales. Estas dependencias indican qué acciones tienen que haberse realizado antes de poder empezar a realizar otra y, por tanto, nos limitarán después a la hora de decidir en qué ciclo se realiza cada acción.

En el caso de la instrucción `push`, tenemos las siguientes dependencias:

```

Leer $29  →  Calcular $29 - 4  →  Escribir $29 - 4 en $29
                ↘
Leer $x   →  Escribir $x en $29 - 4

```

#### 4.6.2. Diseño

En la fase de diseño debemos decidir, a partir de los datos obtenidos en el análisis, cómo se realizará la implementación de la instrucción.

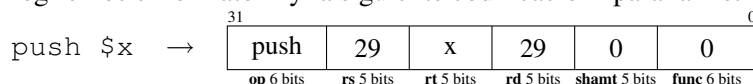
### Definición de la codificación de la instrucción

Se ha de elegir cómo se va a codificar la instrucción. La codificación elegida se debe ajustar a alguno de los tres formatos de instrucción preexistentes (a no ser que sea imposible). Por supuesto, la codificación debe permitir diferenciar la nueva instrucción de las instrucciones ya existentes.

A la hora de elegir el formato a usar conviene fijarse en los operandos de la instrucción. Así, una instrucción que requiera un operando constante requerirá, muy probablemente, que usemos el formato I.

En el caso de la instrucción `push`, el único operando explícito es el registro a apilar. Tiene también dos operandos implícitos: la constante 4 y el registro `$sp` (`$29`).

Elegiremos el formato R y la siguiente codificación<sup>6</sup> para la instrucción “`push $x`”:



En el diagrama anterior “`push`” representa un código de operación cualquiera no utilizado por ninguna instrucción ya implementada. Los campos **shamt** y **func** no nos van a resultar útiles para esta instrucción, por lo que les asignamos el valor 0.

Para decidir la colocación de cada operando, es conveniente tener en cuenta el uso que se va a hacer de ellos después y las conexiones ya disponibles en nuestro camino de datos, de forma que minimicemos el número de cambios a realizar en el camino. En el caso que nos ocupa:

- El registro `$x` va a ser leído únicamente, por lo que podríamos colocarlo tanto en el campo **rs** como el **rt**.
- El registro `$29` va a leerse y escribirse, y además vamos a sumarle 4 al valor leído.

Por tanto, nos conviene que podamos llevarlo al puerto del primer operando de la ALU (para poder aprovechar la constante 4 ya presente como segundo operando). Para esto, lo colocamos en el campo **rs** (obligándonos a colocar el registro `$x` en el campo **rt**).

También nos conviene poderlo llevar al puerto del registro de escritura del banco de registro. Para esto, lo colocamos también en el campo **rd**.

No incluimos el operando 4 en la codificación de la instrucción, ya que dicho operando ya está disponible en el camino de datos de nuestro procesador (entrada 1 del multiplexor controlado por la señal de control `SelALUB`).

### División del trabajo en ciclos

En este paso hemos de decidir en qué ciclo se va a realizar cada una de las acciones identificadas hasta ahora. Esta asignación determinará el número de ciclos que se tardará en ejecutar la instrucción. Se han de respetar tres tipos de restricciones:

- Las dependencias identificadas durante el análisis de la instrucción.
- Las restricciones en el uso de las unidades funcionales:
  - No se puede utilizar la misma unidad dos veces en el mismo ciclo.
  - La entrada de una unidad funcional principal no puede depender de la salida producida por ninguna otra unidad funcional principal.

<sup>6</sup>La solución que se muestra aquí es, probablemente, la más adecuada. Sin embargo, en este caso hay más alternativas que también habrían sido razonables, tales como usar el formato I. Pero las modificaciones al camino de datos en caso de usar el formato I hubieran sido algo más complicadas. La práctica es importante a la hora de elegir la codificación más adecuada.

- Operaciones que ya se realizan independientemente de la instrucción a ejecutar durante los primeros dos ciclos de la ejecución. Frecuentemente se podrá aprovechar el trabajo realizado en estos ciclos, pero ha de tenerse en cuenta que no se pueden realizar acciones distintas durante esos ciclos porque todavía no se ha decodificado la instrucción.

Al realizar la asignación de trabajo a los ciclos tendremos también que indicar el uso que se hará de los registros internos del procesador. También se identificará en esta fase la necesidad de añadir nuevos registros auxiliares, señales de habilitación de escritura para registros ya existentes o nuevas unidades funcionales auxiliares.

En el caso de la instrucción `push`, podemos realizar la siguiente división del trabajo en ciclos:

Ciclo 1	Ciclo 2	Ciclo 3	Ciclo 4
$IR \leftarrow \text{Memoria}[PC]$	$A \leftarrow \$29$	$ALUOut \leftarrow \$29 - 4$	$\$29 \leftarrow ALUOut$
$PC \leftarrow PC + 4$	$B \leftarrow \$x$		$\text{Memoria}[ALUOut] = B$

En general, no es necesario mostrar el primer ciclo en esta tabla ya que es igual para todas las instrucciones.

### Extensión del camino de datos

En este paso se detallan las modificaciones que es necesario realizar en el camino de datos para permitir la realización de las acciones especificadas en el paso anterior.

En el caso de la instrucción `push`, no necesitamos realizar ninguna modificación ya que el camino de datos ya incluye todas las unidades funcionales y conexiones entre unidades necesarias.

### Extensión del control

Finalmente, es necesario modificar el autómeta que define a la unidad de control (figura 16) para que se realicen las acciones adecuadas en cada ciclo.

El autómeta modificado para la instrucción `push` se muestra en la figura 19.

#### 4.6.3. Ejemplo: `bltzal`

Como ejemplo adicional, mostraremos cómo añadir la instrucción `bltzal`, que es una instrucción del ISA de MIPS que realiza un salto condicional enlazado (como `jal`) si el valor de un registro es menor que cero. Al igual que todas las instrucciones de salto condicional, el destino del salto es relativo a la posición actual del programa.

Aplicando la metodología vista anteriormente a esta instrucción, seguiríamos los pasos siguientes:

- Análisis:**

- Especificación precisa de la semántica de la instrucción:**

La semántica de "`bltzal $x, label`" es:

$$\text{si } \$x < 0 \Rightarrow \$ra \leftarrow PC + 4 \\ PC \leftarrow \text{label}$$

- Identificación del trabajo a realizar por cada unidad funcional principal:**

En primer lugar, hay que tener en cuenta que la comprobación de la condición  $\$x < 0$  se puede realizar de varias formas:

- La forma más general de comparar dos valores es utilizar la ALU para restarlos, tal y como hace la instrucción `slt`.
- Dado que en este caso se trata solo de saber si un valor es menor que 0, bastaría con mirar su bit de signo.

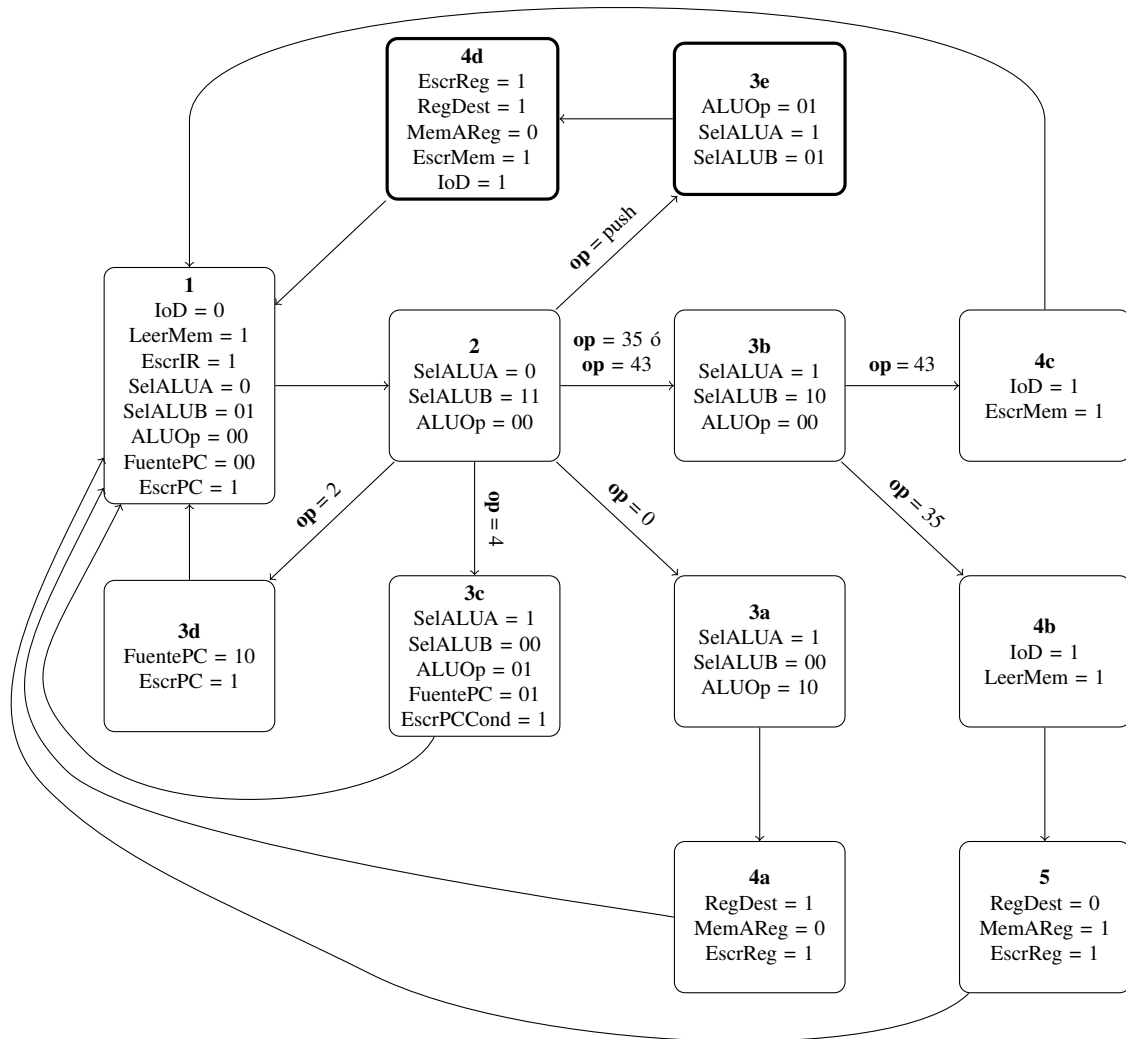


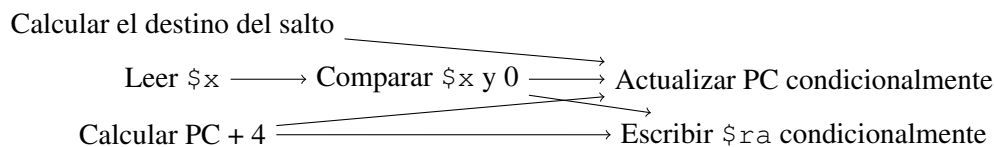
Figura 19: Autómata de control modificado para incluir la instrucción push. Los estados añadidos han sido resaltados.

Cualquiera de las dos formas anteriores valdría. Utilizaremos la segunda por ser más sencilla y para evitar tener que usar la ALU para la comparación. De esta forma, el uso que se hará de las unidades funcionales para esta instrucción será:

- Banco de registros:
  - Leer  $\$x$ .
  - Escribir  $PC + 4$  en  $\$ra$  ( $\$31$ ) condicionalmente.
- ALU:
  - Calcular  $PC + 4$  (se calcula siempre en el ciclo 1).
  - Calcular la dirección de destino del salto (se calcula siempre en el ciclo 2).
- Memoria: No se accede a memoria.

Además, se escribe en el PC si el resultado de la comparación es que  $\$x$  es menor que 0.

3. **Establecimiento del orden de precedencia entre las distintas tareas a realizar:**



■ **Diseño:**

1. **Definición de la codificación de la instrucción:** Debido a que la instrucción tiene dos operandos en registros (se lee  $\$x$  y se escribe  $\$ra$ ) y otro operando inmediato (la etiqueta de destino), debemos usar el formato I. Usaremos la siguiente distribución de campos<sup>7</sup>:



La etiqueta se codificará de forma análoga a la instrucción `beq`.

2. **División del trabajo en ciclos:**

Ciclo 1	Ciclo 2	Ciclo 3
IR ← Memoria[PC] PC ← PC + 4	A ← $\$x$ ALUOut ← $PC + (\text{imm} \ll 2)$	si $A[31] = 1 \Rightarrow PC \leftarrow \text{ALUOut}$ si $A[31] = 1 \Rightarrow \$31 \leftarrow PC$

3. **Extensión del camino de datos:**

De las acciones especificadas en la tabla anterior, no podemos realizar las del tercer ciclo sin modificar el camino de datos de la figura 18. Las modificaciones son las siguientes:

- Para “si  $A[31] = 1 \Rightarrow PC \leftarrow \text{ALUOut}$ ”:  
La salida del registro ALUOut ya está conectada con el PC, pero tenemos que lograr que la señal de habilitación de escritura se active sólo cuando  $A[31]$  sea 1. Para ello, seguiremos un esquema análogo al de la señal `EscrPCCond`:  
Añadiremos una puerta AND conectada a la puerta OR que controla la señal de habilitación de escritura del PC. Esta puerta AND tendrá 2 entradas: una conectada al bit 31 del registro A y la otra conectada una nueva señal de control que llamaremos `EscrPCSignA`. Cuando la señal `EscrPCSignA` esté activa, se escribirá en el PC si el bit de signo de A está activo.

<sup>7</sup>Esta no es la codificación de la instrucción `bltzal` en el ISA real de MIPS.

- Para “si  $A[31] = 1 \Rightarrow PC \leftarrow PC$ ”:

Tenemos que conectar la salida del registro PC a la entrada de datos del banco de registros. Para ello, ampliaremos el multiplexor controlado por la señal de control MemAReg, la cual pasa a tener dos bits en lugar de uno. Cuando MemAReg valga 10, se escribirá en el banco de registros el valor del registro PC (si se escribe algo). El significado del resto de valores posibles de MemAReg que se indican en la tabla 2 no cambia (simplemente se añade un cero a la izquierda a los valores).

Además, necesitamos que la escritura en el banco de registros sea condicional. Actualmente, la escritura está controlada por la señal EscrReg, que causa una escritura incondicional. Añadiremos una puerta OR conectada a la señal de habilitación de escritura del banco de registros. A esta puerta OR se le conectará, por un lado, la señal de control EscrReg (para que el significado de ésta no cambie) y, por otro lado, una puerta AND adicional. A esta puerta AND se le conectará el bit 31 del registro A y una nueva señal de control que llamaremos EscrRegSignA. Cuando la señal EscrRegSignA esté activa, se escribirá en el banco de registros si el bit de signo de A está activo.

Las modificaciones realizadas se pueden ver en la figura 20.

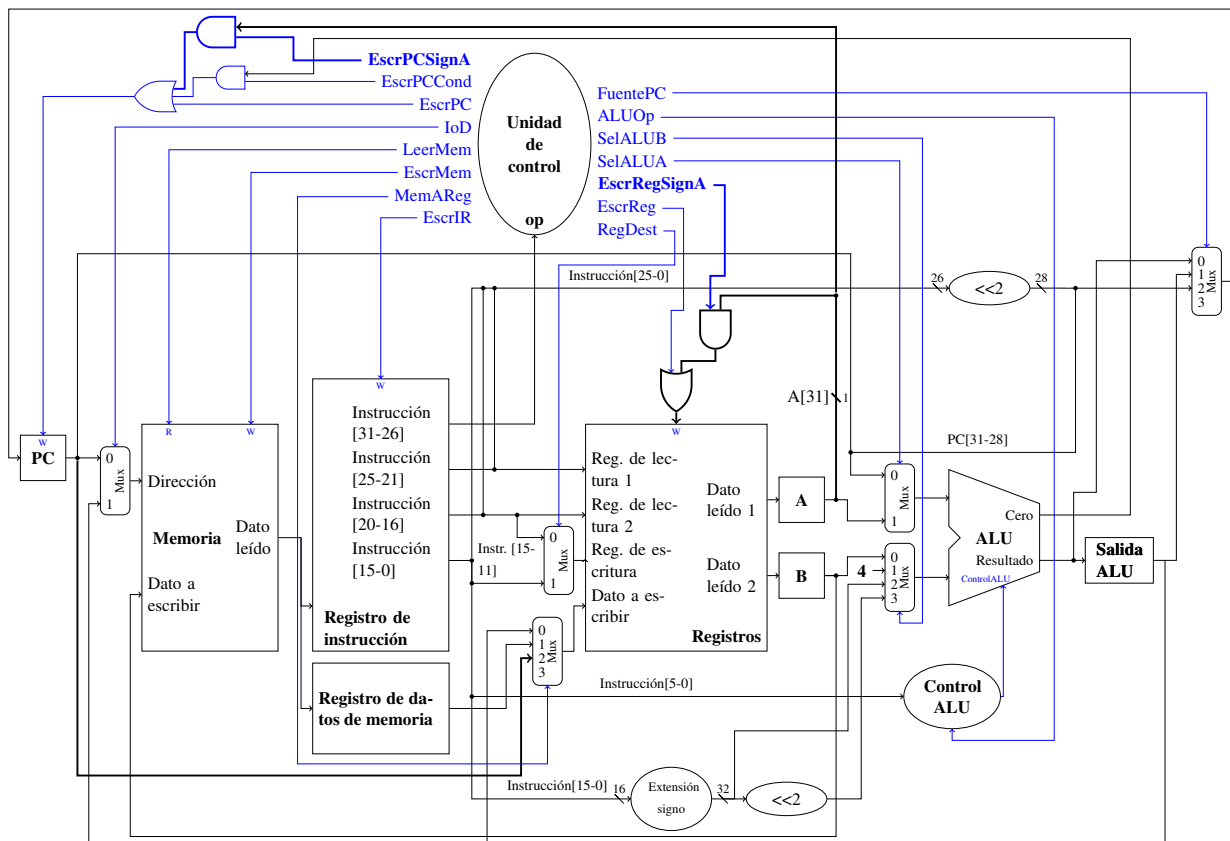


Figura 20: Visión completa de la implementación del procesador con las modificaciones requeridas por la instrucción bltzal.

#### 4. Extensión del control:

Habrá que añadir un nuevo estado al autómata de control de la figura 16. El resultado se muestra en la figura 21.

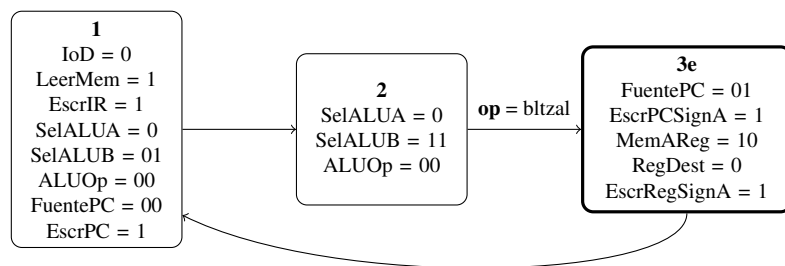


Figura 21: Modificaciones al autómata de control para incluir la instrucción `bltzal`. Por brevedad, se muestran solo los estados involucrados en la ejecución de la instrucción.

## Ejercicios

### E4.1. Cálculo del tiempo de ejecución de programas sencillos en la implementación multiciclo

- Supongamos que los tiempos de operación para las principales unidades funcionales de la implementación multiciclo estudiada en clase son:

- Unidad de memoria: 8 ns.
- ALU: 8 ns.
- Banco de registros (lectura o escritura): 4 ns.

Queremos comparar la implementación anterior con una implementación monociclo cuyo tiempo de ciclo es de 32 ns. Para ello, utilizamos un programa de prueba, *pr\_test*, en el que el 16 % de las instrucciones son saltos (condicionales e incondicionales). Discuta cuál de las dos implementaciones (monociclo o multiciclo) es más rápida en función del porcentaje de aparición de instrucciones `lw` en el programa *pr\_test*.

- Consideremos la siguiente idea: vamos a modificar el ISA de MIPS de tal manera que eliminemos la posibilidad de indicar un desplazamiento en los accesos a memoria. Es decir, todas las instrucciones de acceso a memoria con desplazamientos distintos de cero pasarían a ser pseudoinstrucciones que se implementarían usando dos instrucciones. Por ejemplo:

```
lw $t0, 0x16($t1)  →  addi $at, $t1, 0x16
                    lw $t0, $at
```

De esta forma, el tiempo de ejecución de las instrucciones de acceso a memoria se reduce ya que ahora no es necesario utilizar la ALU para determinar la dirección efectiva. Sin embargo, aquellos accesos a memoria que originalmente tuvieran desplazamientos distintos de cero tienen un coste mayor al tenerse que implementar mediante dos instrucciones en el procesador modificado.

Suponiendo que en el tiempo de ejecución de las instrucciones sólo tenemos en cuenta las unidades funcionales más importantes y que sus latencias son:

- Unidad de memoria: 2 ns.
- ALU: 2 ns.
- Banco de registros (lectura o escritura): 1 ns.

Y suponiendo que la frecuencia de ejecución de las instrucciones es:

- Cargas: 24 %.
- Almacenamientos: 12 %.
- Tipo R: 44 %.
- Saltos condicionales: 18 %.
- Saltos incondicionales: 2 %.

Calcule cuál es el mayor porcentaje de instrucciones de acceso a memoria con desplazamiento no nulo que podemos tolerar sin que se pierda rendimiento respecto a una implementación del ISA MIPS original.

## 3. Dado el siguiente fragmento de código MIPS:

```

        li    $8, 0x10010000
        li    $9, 0x1001FFFC
        move  $10, $zero
loop:   sll   $11, $10, 2
        add  $11, $8, $11
        lw   $11, 0($11)
        sw   $11, 0($9)
        beq  $11, $zero, fin
        addi $10, $10, 1
        addi $9, $9, -4
        j    loop
fin:    ...

```

Y suponiendo que la memoria contiene los siguientes valores en las direcciones desde 0x10010000 hasta 0x10020000:

Dirección	Contenido (hexadecimal)			
	+0	+1	+2	+3
0x10010000	02	00	00	00
0x10010004	04	00	00	00
0x10010008	06	00	00	00
0x1001000C	08	00	00	00
0x10010010	0A	00	00	00
0x10010014	0C	00	00	00
0x10010018	0E	00	00	00
0x1001001C	02	00	00	00
0x10010020	01	00	00	00
0x10010024	00	00	00	00
...resto...	00	00	00	00

Calcule el tiempo que tardará en ejecutarse en un procesador multiciclo como el visto en clase suponiendo que las latencias de sus unidades funcionales son:

- Unidad de memoria: 30 ns.
- ALU: 12 ns.
- Banco de registros (lectura o escritura): 20 ns.

A la hora de resolver el ejercicio, para las instrucciones que no estuvieran presentes en el modelo original del procesador visto en clase suponga que tardan 4 ciclos en ejecutarse si son aritmético-lógicas o de desplazamiento. Además, tenga en cuenta que algunas instrucciones de ensamblador pueden ser realmente pseudoinstrucciones que se implementen con una o más instrucciones reales distintas. Por último, suponga que la implementación MIPS utilizada para ejecutar el programa es *little-endian*.

## 4. Dado el siguiente fragmento de código MIPS:

```

        li    $8, 0x10018008

```

```

        move    $9, $zero
loop:   lb     $10, 1($8)
        beq    $10, $0, fin
        blt    $10, $0, neg
        add    $9, $9, $10
        move   $10, $zero
neg:    sub    $9, $9, $10
        addi   $8, $8, 4
        j     loop
fin:    sll   $9, $9, 1
        ...

```

Y suponiendo que la memoria contiene los siguientes valores en las direcciones desde 0x10018004 hasta 0x1001803c:

Dirección	Contenido (hexadecimal)			
	+0	+1	+2	+3
0x10018004	00	00	01	00
0x10018008	00	05	F0	FF
0x1001800C	00	F6	0F	F0
0x10018010	00	02	00	00
0x10018014	00	F7	00	30
0x10018018	00	FF	8A	3F
0x1001801C	03	03	03	03
0x10018020	0A	07	06	FF
0x10018024	F4	F8	F0	00
0x10018028	00	00	00	00
...resto...	00	00	00	00

- Indique el valor que se obtiene en el registro \$9 después de ejecutar dicho código.
- Calcule el tiempo que tardará en ejecutarse en un procesador multiciclo como el visto en clase cuyas unidades funcionales tienen las siguientes latencias:
  - Unidad de memoria: 20 ns.
  - ALU: 10 ns.
  - Banco de registros (lectura o escritura): 15 ns.

A la hora de resolver el ejercicio, para las instrucciones que no estuvieran presentes en el modelo original del procesador visto en clase suponga que tardan 4 ciclos en ejecutarse si son aritmético-lógicas o de desplazamiento, 5 ciclos si son de lectura de memoria y 3 ciclos si son saltos. Además, tenga en cuenta que algunas instrucciones de ensamblador pueden ser realmente pseudoinstrucciones que se implementen con una o más instrucciones reales distintas. Por último, suponga que la implementación MIPS utilizada para ejecutar el programa es *little-endian*.

- El siguiente fragmento de código MIPS recibe en \$a0 la dirección de un array de enteros y en \$a1 el número de elementos de dicho array, que se garantiza que es siempre mayor o igual a 2:

```

sw     $0, 0($a0)
li     $t0, 1
sw     $t0, 4($a0)

```

```

        addi   $t0, $a0, 8
        sll   $t1, $a1, 2
        add   $t1, $a0, $t1
bucle:  bge   $t0, $t1, fin
        lw    $t2, -8($t0)
        lw    $t3, -4($t0)
        add   $t2, $t2, $t3
        sw    $t2, 0($t0)
        addi  $t0, $t0, 4
        j     bucle
fin:

```

- Explique qué hace el programa.
- Calcule, en función del valor inicial del registro `$a1`, el tiempo que tardará en ejecutarse en un procesador multiciclo como el visto en clase cuyas unidades funcionales tienen las siguientes latencias:
  - Unidad de memoria: 17 ns.
  - ALU: 10 ns.
  - Banco de registros (lectura o escritura): 8 ns.

A la hora de resolver el ejercicio, para las instrucciones que no estuvieran presentes en el modelo original del procesador visto en clase suponga que tardan 4 ciclos en ejecutarse si son aritmético-lógicas o de desplazamiento, 5 ciclos si son de lectura de memoria y 3 ciclos si son saltos. Además, tenga en cuenta que algunas instrucciones de ensamblador pueden ser realmente pseudoinstrucciones que se implementen con una o más instrucciones reales distintas.

#### 6. Dado el siguiente fragmento de código MIPS:

```

        li     $t3, 10
        li     $v0, 0
loop:   lb     $t2, 0($a0)
        beq   $t2, $zero, fin
        addi  $t2, $t2, -48
        blt  $t2, $zero, err
        bge  $t2, $t3, err
        mult $v0, $t3
        mflo $v0
        add  $v0, $v0, $t2
        addi $a0, $a0, 1
        j   loop
err:    li     $v0, -1
fin:

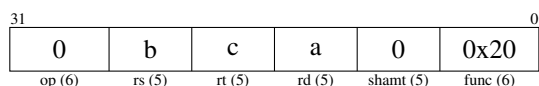
```

- Sabiendo que el registro `$a0` contiene inicialmente un puntero a un array de caracteres y que 48 (0x30 en hexadecimal) es el código ASCII del carácter '0', explique qué hace el código anterior.
- Considerando el volcado de memoria que se muestra a continuación, calcule el tiempo de ejecución de este programa en el procesador multiciclo estudiado en clase si inicialmente `$a0` contiene el valor 0x10018000.

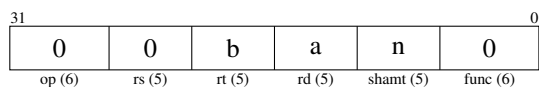
Dirección	Contenido (hexadecimal)			
	+0	+1	+2	+3
0x10018000	31	32	33	34
0x10018004	00	00	01	00
0x10018008	00	05	F0	FF
0x1001800C	00	F6	0F	F0
...resto...	00	00	00	00

A la hora de resolver el ejercicio, para las instrucciones que no estuvieran presentes en el modelo original del procesador visto en clase suponga que tardan 4 ciclos en ejecutarse si son aritméticas o de desplazamiento, 5 ciclos si son de lectura de memoria y 3 ciclos si son saltos. Además, tenga en cuenta que algunas instrucciones de ensamblador pueden ser realmente pseudoinstrucciones que se implementen con una o más instrucciones reales distintas. Por último, suponga que la implementación MIPS utilizada para ejecutar el programa es *little-endian*.

7. Como sabemos, la codificación de una instrucción `add $a, $b, $c` en MIPS es la siguiente:



Por otro lado, la codificación de una instrucción `sll $a, $b, n` es:



Un diseñador se da cuenta de que el campo **shamt** (utilizado para codificar el número de bits de desplazamiento en las instrucciones `sll`, `srl` y `sra`) no está siendo aprovechado para las instrucciones aritméticas como `add`. Al mismo tiempo, se da cuenta de que en muchos programas aparecen con mucha frecuencia secuencias de instrucciones como la siguiente:

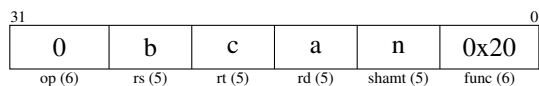
```
sll $t0, $s1, 2 # $s1: índice de un elemento de un array
add $t0, $s0, $t0 # $s0: la dirección de inicio del array
```

Si `$s0` contiene la dirección de inicio de un array cuyos elementos ocupan 4 bytes cada uno, la secuencia anterior calcularía en `$t0` la dirección de memoria del elemento cuyo índice está almacenado en `$s1`.

Al diseñador se le ocurre que se podrían combinar las dos instrucciones anteriores en una nueva que decide llamar `slladd`. Esta instrucción tiene la siguiente forma:

```
slladd $a, $b, $c, n # almacena en $a el resultado de
                    # $b + ($c << n) = $b + $c * 2 ^ n
```

Y se codificaría exactamente igual que la instrucción `add` excepto que el campo **shamt** tendría un valor distinto de 0:



De esta forma, el fragmento de código de dos instrucciones anterior sería equivalente a la siguiente instrucción:

```
slladd $t0, $s0, $s1, 2
```

Por último, el diseñador modifica el compilador para que haga uso de la nueva instrucción siempre que sea posible. El compilador mejorado es capaz de eliminar el 60 % de las instrucciones `sll` presentes en los programas originales, combinándolas con una instrucción `add` posterior.

Para comprobar que todo funciona, realiza pruebas con un conjunto de benchmarks que presenta la siguiente mezcla de instrucciones:

add	25 %
sll	5 %
Otras aritmético-lógicas	20 %
Escritura en memoria	10 %
Lectura de memoria	20 %
Saltos condicionales	15 %
Saltos incondicionales	5 %

Sabiendo que el procesador original (antes de añadirle la instrucción `slladd`) con un reloj a 1GHz ejecuta el conjunto de benchmarks en 12.30 segundos, calcule el tiempo de ejecución en el procesador mejorado bajo los siguientes supuestos:

- Suponiendo que la instrucción `slladd` tarda 5 ciclos en ejecutarse, mientras que los `add` normales (con `shamt` igual a 0) tardan 4 ciclos.
- Suponiendo que tanto `slladd` como los `add` normales tardan 4 ciclos, pero el tiempo de ciclo se incrementa un 3 % para poder realizar el desplazamiento.
- Suponiendo que tanto `slladd` como los `add` normales tardan 5 ciclos, pero el tiempo de ciclo se mantiene igual. El resto de instrucciones aritmético-lógicas sigue tardando 4 ciclos.

## E4.2. Solución a ejercicios seleccionados de la sección E4.1

4. Dado el siguiente fragmento de código MIPS:

```

        li    $8, 0x10018008
        move  $9, $zero
loop:   lb    $10, 1($8)
        beq  $10, $0, fin
        blt  $10, $0, neg
        add  $9, $9, $10
        move $10, $zero
neg:    sub  $9, $9, $10
        addi $8, $8, 4
        j    loop
fin:    sll  $9, $9, 1
        ...

```

Y suponiendo que la memoria contiene los siguientes valores en las direcciones desde 0x10018004 hasta 0x1001803c:

Dirección	Contenido (hexadecimal)			
	+0	+1	+2	+3
0x10018004	00	00	01	00
0x10018008	00	05	F0	FF
0x1001800C	00	F6	0F	F0
0x10018010	00	02	00	00
0x10018014	00	F7	00	30
0x10018018	00	FF	8A	3F
0x1001801C	03	03	03	03
0x10018020	0A	07	06	FF
0x10018024	F4	F8	F0	00
0x10018028	00	00	00	00
...resto...	00	00	00	00

- a) Indique el valor que se obtiene en el registro \$9 después de ejecutar dicho código.
- b) Calcule el tiempo que tardará en ejecutarse en un procesador multiciclo como el visto en clase cuyas unidades funcionales tienen las siguientes latencias:
- Unidad de memoria: 20 ns.
  - ALU: 10 ns.
  - Banco de registros (lectura o escritura): 15 ns.

A la hora de resolver el ejercicio, para las instrucciones que no estuvieran presentes en el modelo original del procesador visto en clase suponga que tardan 4 ciclos en ejecutarse si son aritméticas o de desplazamiento, 5 ciclos si son de lectura de memoria y 3 ciclos si son saltos. Además, tenga en cuenta que algunas instrucciones de ensamblador pueden ser realmente pseudoinstrucciones que se implementen con una o más instrucciones reales distintas. Por último, suponga que la implementación MIPS utilizada para ejecutar el programa es *little-endian*.

**Solución:**

- a) El código tiene un bucle en el que se van leyendo los bytes apuntados por \$8 + 1 (línea 3). El registro \$8 se inicializa a 0x10018008 (línea 1) y se incrementa en 4 en cada iteración (línea 9). El bucle termina cuando el elemento leído es un 0 (línea 4). La instrucción lb extiende el signo del byte leído de memoria, por lo que los valores leídos son 5, -10, 2, -9, -1, 3, 7, -8 y 0.

Para cada valor leído, se comprueba si es negativo (línea 5). Si no es negativo, el valor se suma al registro \$9 (línea 6), y si es negativo se resta (línea 8). La línea 7 evita que la línea 8 tenga ningún efecto en el caso de los valores positivos. Por tanto, teniendo en cuenta que \$9 se inicializa a 0 antes del bucle (línea 2), al final del bucle \$9 contendrá la suma de los valores absolutos de los valores leídos. Finalmente, la línea 11 multiplica el valor de \$9 por 2.

Con los datos de este problema, el valor final de \$9 será 90.

- b) El tiempo de ejecución del programa vendrá dado por la expresión:

$$T_{ejec} = N_{inst} \times CPI \times T_{ciclo} = N_{ciclos} \times T_{ciclo}$$

El tiempo de ciclo ( $T_{ciclo}$ ) está determinado por la unidad funcional más lenta, que en este caso es la memoria que tarda 30 ns.

Para hallar el número total de ciclos necesarios para la ejecución del programa, tendremos que seguir la ejecución del mismo paso a paso y contar cuántas veces se ejecuta cada instrucción.

Habrá que tener en cuenta que la primera instrucción (li) es en realidad una pseudoinstrucción que se traduce por dos instrucciones de código máquina (un lui y un ori), que las instrucciones de las líneas 2 y 7 (move) son también pseudoinstrucciones que se traducen con add, y que la instrucción de la línea 5 es otra pseudoinstrucción que se traduce con un slt y un bne.

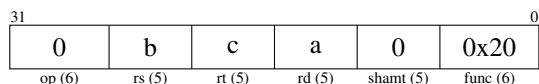
Por tanto:

Instrucción	Ejecuciones	Ciclos por ejecución	Ciclos totales
lui \$8, 0x1001	1	4	4
ori \$8, 0x8008	1	4	4
add \$9, \$0, \$0	1	4	4
loop: lb \$10, 1(\$8)	9	5	45
beq \$10, \$0, fin	9	3	27
slt \$at, \$10, \$0	8	4	32
bne \$at, \$0, neg	8	3	24
add \$9, \$9, \$10	4	4	16
add \$10, \$0, \$0	4	4	16
neg: sub \$9, \$9, \$10	8	4	32
addi \$8, \$8, 4	8	4	32
j loop	8	3	24
fin: sll \$9, \$9, 1	1	4	4
			<b>Total: 264</b>

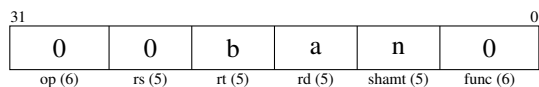
Por tanto:

$$T_{ejec} = N_{ciclos} \times T_{ciclo} = 264 \times 20 \text{ ns} = 5,280 \mu\text{s}$$

7. Como sabemos, la codificación de una instrucción add \$a, \$b, \$c en MIPS es la siguiente:



Por otro lado, la codificación de una instrucción `sll $a, $b, n` es:



Un diseñador se da cuenta de que el campo **shamt** (utilizado para codificar el número de bits de desplazamiento en las instrucciones `sll`, `srl` y `sra`) no está siendo aprovechado para las instrucciones aritméticas como `add`. Al mismo tiempo, se da cuenta de que en muchos programas aparecen con mucha frecuencia secuencias de instrucciones como la siguiente:

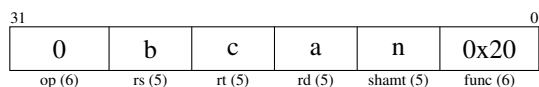
```
sll $t0, $s1, 2 # $s1: índice de un elemento de un array
add $t0, $s0, $t0 # $s0: la dirección de inicio del array
```

Si `$s0` contiene la dirección de inicio de un array cuyos elementos ocupan 4 bytes cada uno, la secuencia anterior calcularía en `$t0` la dirección de memoria del elemento cuyo índice está almacenado en `$s1`.

Al diseñador se le ocurre que se podrían combinar las dos instrucciones anteriores en una nueva que decide llamar `slladd`. Esta instrucción tiene la siguiente forma:

```
slladd $a, $b, $c, n # almacena en $a el resultado de
                    # $b + ($c << n) = $b + $c * 2 ^ n
```

Y se codificaría exactamente igual que la instrucción `add` excepto que el campo **shamt** tendría un valor distinto de 0:



De esta forma, el fragmento de código de dos instrucciones anterior sería equivalente a la siguiente instrucción:

```
slladd $t0, $s0, $s1, 2
```

Por último, el diseñador modifica el compilador para que haga uso de la nueva instrucción siempre que sea posible. El compilador mejorado es capaz de eliminar el 60% de las instrucciones `sll` presentes en los programas originales, combinándolas con una instrucción `add` posterior.

Para comprobar que todo funciona, realiza pruebas con un conjunto de benchmarks que presenta la siguiente mezcla de instrucciones:

add	25 %
sll	5 %
Otras aritmético-lógicas	20 %
Escritura en memoria	10 %
Lectura de memoria	20 %
Saltos condicionales	15 %
Saltos incondicionales	5 %

Sabiendo que el procesador original (antes de añadirle la instrucción `slladd`) con un reloj a 1GHz ejecuta el conjunto de benchmarks en 12.30 segundos, calcule el tiempo de ejecución en el procesador mejorado bajo los siguientes supuestos:

- a) Suponiendo que la instrucción `slladd` tarda 5 ciclos en ejecutarse, mientras que los `add` normales (con `shamt` igual a 0) tardan 4 ciclos.
- b) Suponiendo que tanto `slladd` como los `add` normales tardan 4 ciclos, pero el tiempo de ciclo se incrementa un 3 % para poder realizar el desplazamiento.
- c) Suponiendo que tanto `slladd` como los `add` normales tardan 5 ciclos, pero el tiempo de ciclo se mantiene igual. El resto de instrucciones aritmético-lógicas sigue tardando 4 ciclos.

**Solución:**

De los datos del enunciado podemos calcular el CPI y el número de instrucciones total de los benchmarks utilizados cuando se ejecutan en el procesador original. Para calcular el CPI, es suficiente con sumar los productos de la frecuencia de cada tipo de instrucción y su duración en ciclos:

$$\begin{aligned}
 CPI_{original} &= f_{add} \times Ciclos_{add} \\
 &+ f_{sll} \times Ciclos_{sll} \\
 &+ f_{otrasAL} \times Ciclos_{otrasAL} \\
 &+ f_{sw} \times Ciclos_{sw} \\
 &+ f_{lw} \times Ciclos_{lw} \\
 &+ f_{branch} \times Ciclos_{branch} \\
 &+ f_j \times Ciclos_j \\
 &= 0,25 \times 4 + 0,05 \times 4 + 0,20 \times 4 + 0,10 \times 4 + 0,20 \times 5 + 0,15 \times 3 + 0,05 \times 3 \\
 &= 4,0
 \end{aligned}$$

El número total de instrucciones se puede despejar de la siguiente ecuación:

$$T_{original} = NI_{original} \times CPI_{original} \times TC_{original}$$

En la que sabemos que  $T_{original} = 12,30$  s y  $TC_{original} = \frac{1}{1\text{GHz}} = 1$  ns. Por tanto:

$$NI_{original} = \frac{T_{original}}{CPI_{original} \times TC_{original}} = \frac{12,30}{4,0 \times 1 \times 10^{-9}} = 3,075 \times 10^9 \text{ instrucciones}$$

Al utilizar el nuevo compilador, el número de instrucciones y la frecuencia relativa de las mismas cambia. En particular, el 60 % del 5 % de instrucciones totales originales son `sll` que desaparecen (es decir el 3 % de las instrucciones totales), y el mismo número de instrucciones `add` se transforman en `slladd`. Por tanto, la nueva distribución de instrucciones será:

add	$\frac{25-3}{100-3} \% = 22,6804 \%$
sll	$\frac{5-3}{100-3} \% = 2,0619 \%$
slladd	$\frac{3}{100-3} \% = 3,0928 \%$
Otras aritmético-lógicas	$\frac{20}{100-3} \% = 20,6186 \%$
Escritura en memoria	$\frac{10}{100-3} \% = 10,3093 \%$
Lectura de memoria	$\frac{20}{100-3} \% = 20,6186 \%$
Saltos condicionales	$\frac{15}{100-3} \% = 15,4639 \%$
Saltos incondicionales	$\frac{5}{100-3} \% = 5,1546 \%$

El número de instrucciones disminuye un 3 % en total, por lo que:

$$NI_{modificado} = \frac{100 - 3}{100} \times NI_{original} = 0,97 \times 3,075 \times 10^9 = 2,98275 \times 10^9 \text{ instrucciones}$$

- a) Suponiendo que la instrucción `slladd` tarda 5 ciclos en ejecutarse, mientras que los `add` normales (con `shamt` igual a 0) tardan 4 ciclos.

En este caso, el CPI del nuevo procesador será:

$$\begin{aligned} CPI_{modificado} &= f_{add} \times Ciclos_{add} \\ &+ f_{sll} \times Ciclos_{sll} \\ &+ f_{slladd} \times Ciclos_{slladd} \\ &+ f_{otrasAL} \times Ciclos_{otrasAL} \\ &+ f_{sw} \times Ciclos_{sw} \\ &+ f_{lw} \times Ciclos_{lw} \\ &+ f_{branch} \times Ciclos_{branch} \\ &+ f_j \times Ciclos_j \\ &= 0,226804 \times 4 + 0,020619 \times 4 + 0,030928 \times 5 + 0,206186 \times 4 \\ &+ 0,103093 \times 4 + 0,206186 \times 5 + 0,154639 \times 3 + 0,051546 \times 3 \\ &= 4,0309 \end{aligned}$$

$$Y TC_{modificado} = TC_{original} = 1 \times 10^{-9} \text{ s.}$$

Y por tanto:

$$\begin{aligned} T_{modificado} &= NI_{modificado} \times CPI_{modificado} \times TC_{modificado} \\ &= 2,98275 \times 10^9 \times 4,0309 \times 1 \times 10^{-9} = 12,0232 \text{ s} \end{aligned}$$

- b) Suponiendo que tanto `slladd` como los `add` normales tardan 4 ciclos, pero el tiempo de ciclo se incrementa un 3 % para poder realizar el desplazamiento.

En este caso, el CPI del nuevo procesador será:

$$\begin{aligned} CPI_{modificado} &= f_{add} \times Ciclos_{add} \\ &+ f_{sll} \times Ciclos_{sll} \\ &+ f_{slladd} \times Ciclos_{slladd} \\ &+ f_{otrasAL} \times Ciclos_{otrasAL} \\ &+ f_{sw} \times Ciclos_{sw} \\ &+ f_{lw} \times Ciclos_{lw} \\ &+ f_{branch} \times Ciclos_{branch} \\ &+ f_j \times Ciclos_j \\ &= 0,226804 \times 4 + 0,020619 \times 4 + 0,030928 \times 4 + 0,206186 \times 4 \\ &+ 0,103093 \times 4 + 0,206186 \times 5 + 0,154639 \times 3 + 0,051546 \times 3 \\ &= 4,0000 \end{aligned}$$

$$Y TC_{modificado} = 1,03 \times TC_{original} = 1,03 \times 10^{-9} \text{ s.}$$

Y por tanto:

$$\begin{aligned} T_{modificado} &= NI_{modificado} \times CPI_{modificado} \times TC_{modificado} \\ &= 2,98275 \times 10^9 \times 4,0000 \times 1,03 \times 10^{-9} = 12,2889 \text{ s} \end{aligned}$$

- c) Suponiendo que tanto `slladd` como los `add` normales tardan 5 ciclos, pero el tiempo de ciclo se mantiene igual. El resto de instrucciones aritmético-lógicas sigue tardando 4 ciclos.

En este caso, el CPI del nuevo procesador será:

$$\begin{aligned} CPI_{modificado} &= f_{add} \times Ciclos_{add} \\ &+ f_{sll} \times Ciclos_{sll} \\ &+ f_{slladd} \times Ciclos_{slladd} \\ &+ f_{otrasAL} \times Ciclos_{otrasAL} \\ &+ f_{sw} \times Ciclos_{sw} \\ &+ f_{lw} \times Ciclos_{lw} \\ &+ f_{branch} \times Ciclos_{branch} \\ &+ f_j \times Ciclos_j \\ &= 0,226804 \times 5 + 0,020619 \times 4 + 0,030928 \times 5 + 0,206186 \times 4 \\ &+ 0,103093 \times 4 + 0,206186 \times 5 + 0,154639 \times 3 + 0,051546 \times 3 \\ &= 4,2577 \end{aligned}$$

$$Y TC_{modificado} = TC_{original} = 1 \times 10^{-9} \text{ s.}$$

Y por tanto:

$$\begin{aligned} T_{modificado} &= NI_{modificado} \times CPI_{modificado} \times TC_{modificado} \\ &= 2,98275 \times 10^9 \times 4,2577 \times 1 \times 10^{-9} = 12,6997 \text{ s} \end{aligned}$$

### E4.3. Resultados finales de ejercicios seleccionados de la sección E4.1

En esta sección se muestra el resultado final de algunos ejercicios para permitir la comprobación por parte del alumno de la corrección de su respuesta. No se muestra el desarrollo del problema, por lo que estas respuestas serían insuficientes para considerar el ejercicio como correcto en un examen o control.

1. Ambas implementaciones obtendrán el mismo rendimiento si el porcentaje de instrucciones  $l_w$  es igual a 16%. En caso de ser mayor, el procesador monociclo será más rápido, y en caso de ser menor, el multiciclo será más rápido.
2. El procesador modificado será más rápido siempre que el porcentaje de instrucciones  $l_w$  cuyo desplazamiento sea distinto de 0 sea menor del 25%. Obsérvese que no es necesario utilizar todos los datos dados en el enunciado.
3.  $9,57 \mu s$ .
5.
  - a) Rellena el array con los primeros  $\$a1$  elementos de la secuencia de Fibonacci (0, 1, 1, 2, 3, 5, 8...).
  - b)  $(544 \times \$a1 + 561) ns$ .
6.
  - a) Interpreta el array de caracteres apuntado por  $\$a0$  como una cadena acabada en 0 (como en C) que representa un entero no negativo, calculando el valor representado y dejándolo en  $\$v0$ . Si la cadena contiene algún carácter que no sea un dígito decimal,  $\$v0$  valdrá -1 al final del fragmento.
  - b) 196 ciclos.

## E4.4. Introducción de nuevas instrucciones en el camino de datos multiciclo

Utilizando la metodología descrita en los apuntes para la inclusión de nuevas instrucciones en el esquema de implementación multiciclo, realice las fases de análisis y diseño de las siguientes instrucciones<sup>1</sup>:

1. `bzr $a, $b`

Salta a la dirección contenida en el registro `$b` si `$a` es igual a cero.

2. `abs $a, $b`

Almacena en el registro `$a` el valor absoluto del valor contenido en el registro `$b`.

3. `min $a, $b, $c`

Escribir en el registro `$a` el mínimo de los valores almacenados en los registros `$b` y `$c`.

4. `jal label`

Salta a la dirección de `label` y guarda la dirección de retorno (dirección de la instrucción siguiente al `jal`) en el registro `$ra` (`$31`).

5. `call label`

Salta a la dirección de `label` y guarda la dirección de retorno en la pila (igual que `jal` pero guardando la dirección de retorno en la pila en lugar de en `$ra`).

6. `subm $a, imm($b)`

Resta de `$a` el contenido de la palabra de memoria almacenada en la dirección `$b + imm`, guardando el resultado de dicha resta en `$a`.

7. `test&incr $a,$b`

Si el contenido del registro `$b` es menor que cero, carga la palabra cuya dirección de memoria es el contenido de `$a`, incrementa su valor en 1 y almacena el resultado de dicha suma en `$a`.

8. `cmp&swap $a,$b,imm`

Lee el valor de la posición de memoria apuntada por `$b` y si el contenido de esa posición es igual a `imm`, intercambia los contenidos del registro `$a` y la posición de memoria.

9. `lweq $a, ($b), $c`

Escribe en el registro `$a` el contenido de la memoria cuya dirección está almacenada en el registro `$b` si el contenido del registro `$c` es igual al valor original del registro `$a`. Obsérvese que la dirección de memoria no admite desplazamiento y que el registro `$a` actúa tanto como registro de escritura como de lectura.

10. `beqri $a, $b, $c`

Si los contenidos de los registros `$a` y `$b` coinciden, salta a la dirección de memoria contenida en la palabra de memoria apuntada por `$c`.

11. `bnzi $a, $b, $c`

Si el contenido del registro `$a` es distinto de cero, salta a la dirección de memoria contenida en la dirección de memoria resultado de la suma de los valores almacenados en los registros `$b` y `$c`.

<sup>1</sup>`$a`, `$b` y `$c` se refieren a cualquiera de los 32 registros de MIPS, `label` se refiere a cualquier etiqueta e `imm` se refiere a cualquier constante con signo de 16 bits.

12. `pop $a`

Realiza la operación contraria a `push`: almacena el elemento que se encuentra en la cima de la pila (apuntado por `$sp`) en `$x` e incrementa `$sp` en 4.

13. `pushm ($a)`

Apila el valor contenido en la dirección de memoria contenida en el registro `$a`.

14. `bge $a, ($b), label`

Salta a la dirección de `label` si el valor contenido en el registro `$a` es mayor o igual que el valor contenido en la dirección de memoria contenida en el registro `$b`.

15. `inc&cpy $a, $b`

Incrementa en 4 la dirección contenida en el registro `$a` (el cual actualiza), y si el valor almacenado en el registro `$b` es distinto de 0 lo copia a la nueva dirección de memoria calculada.

16. `avg $a, $b, $c`

Calcula la media aritmética del contenido de los registros `$b` y `$c`, almacenando el resultado en el registro `$a`.

17. `vcall $a, imm`

Esta instrucción permitiría implementar eficientemente algunas llamadas a métodos virtuales de ciertos lenguajes de programación como Java, C++ o C#.

El registro `$a` contendrá un puntero a un objeto<sup>2</sup> cuya primera palabra será un puntero a un array de punteros a procedimientos. La instrucción transferirá el control al procedimiento número `imm` de dicha tabla, almacenando la dirección de retorno en el registro `$ra` (`$31`).

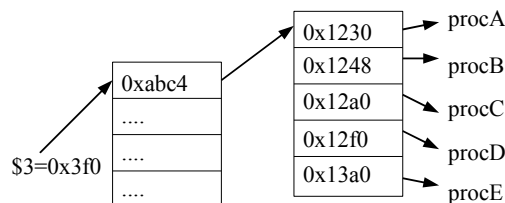


Figura E4.1: Ejemplo de las estructuras en memoria usadas por la instrucción `vcall`.

Por ejemplo: dada la figura E4.1 que representa un objeto (que comienza en la dirección `0x3f0`) apuntado por el registro 3 y su array de punteros a procedimientos (que comienza en la dirección `0xabc4`), la instrucción `vcall $3, 2` llamaría al procedimiento `procC` (que comienza en la dirección `0x12a0`), almacenando en el registro `$31` la dirección de la instrucción siguiente a `vcall`.

18. `swc $a, $b, $c`

Escribe el valor almacenado en el registro `$c` en todas las palabras de memoria desde la apuntada por la dirección almacenada en el registro `$a` a la apuntada por el registro `$b`.

<sup>2</sup>Para nuestros propósitos, un objeto es lo mismo que cualquier estructura de datos en memoria.

### E4.5. Solución a ejercicios seleccionados de la sección E4.4

6. `subm $a, imm($b)`

Resta de \$a el contenido de la palabra de memoria almacenada en la dirección \$b + imm, guardando el resultado de dicha resta en \$a.

**Solución:**

■ **Análisis:**

a) **Especificación precisa de la semántica de la instrucción:**

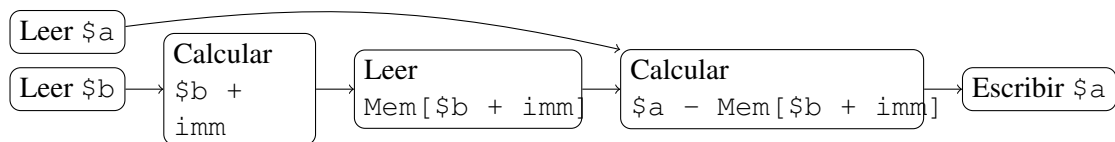
La semántica de “`subm $a, imm($b)`” es:

$$\$a = \$a - \text{Mem}[\$b + \text{imm}]$$

b) **Identificación del trabajo a realizar por cada unidad funcional principal:**

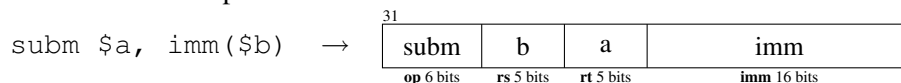
- Banco de registros:
  - Leer \$a.
  - Leer \$b.
  - Escribir \$a.
- ALU:
  - Calcular \$b + imm.
  - Calcular \$a - Mem[\$b + imm].
- Memoria:
  - Leer Mem[\$b + imm].

c) **Establecimiento del orden de precedencia entre las distintas tareas a realizar:**



■ **Diseño:**

a) **Definición de la codificación de la instrucción:** Debido a que la instrucción tiene dos operandos en registros y otro operando inmediato, debemos usar el formato I. Usaremos la siguiente distribución de campos:



La codificación es análoga a la instrucción `lw`. Conviene colocar el registro `a` en el campo `rt` para poder escribir en él fácilmente, y conviene colocar el registro `b` en `rs` para realizar el cálculo de la dirección efectiva de la misma forma que `lw`.

b) **División del trabajo en ciclos:**

Ciclo 1	Ciclo 2	Ciclo 3
IR ← Memoria[PC] PC ← PC + 4	A ← \$b B ← \$a	ALUOut ← A + extender_signo(IR[15-0])
Ciclo 4	Ciclo 5	Ciclo 6
MDR ← Memoria[ALUOut]	ALUOut ← B - MDR	\$a ← ALUOut

c) **Extensión del camino de datos:**

De las acciones especificadas en la tabla anterior, no podemos realizar las acciones del ciclo 5 con el camino de datos del procesador sin modificar. Para hacerlo posible tendremos que extenderlo de la siguiente manera:

- Añadir una nueva entrada al multiplexor controlado por la señal de control SelALUA que esté conectada al registro B (en el que tenemos el contenido de \$a). La señal SelALUA pasará a ser una señal de dos bits en lugar de uno, y la combinación “10” seleccionará la nueva entrada.
- Añadir una nueva entrada al multiplexor controlado por la señal de control SelALUB que esté conectada al registro de datos de memoria (MDR). La señal de control pasará a tener tres bits en lugar de dos, y la combinación “100” seleccionará la nueva entrada.

d) **Extensión del control:**

Habrà que añadir 2 nuevos estados al autómata de control original y añadir una nueva transición entre el estado 2 y el estado 3b. El resultado se muestra en la figura E4.2.

Obsérvese que se reutilizan los estados 3b (que calculan la dirección efectiva del acceso a memoria para lw, sw y, ahora, subm) y 4b (que realiza la lectura de memoria).

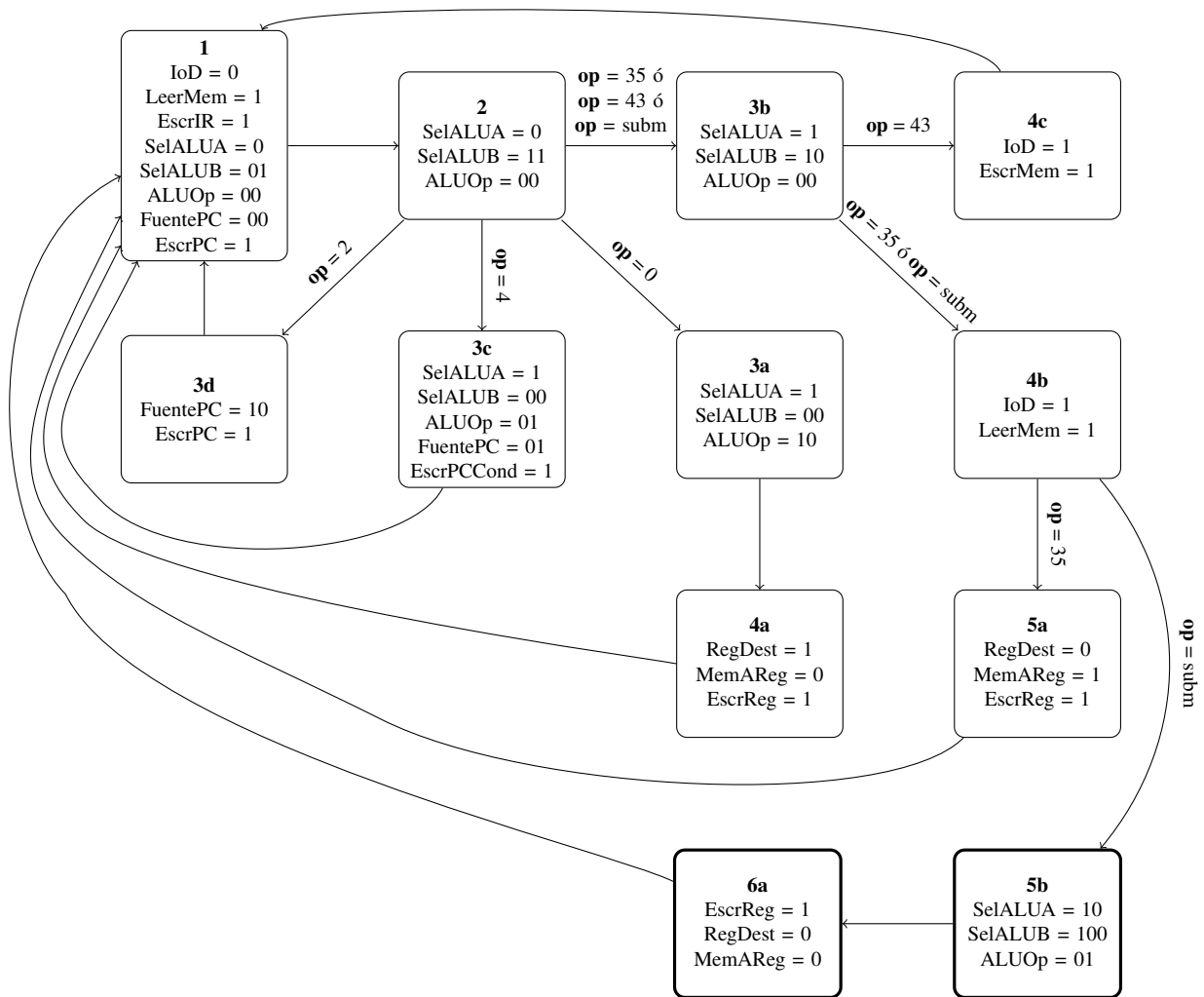


Figura E4.2: Modificaciones al autómata de control para incluir la instrucción subm.

10. beqri \$a, \$b, \$c

Si los contenidos de los registros \$a y \$b coinciden, salta a la dirección de memoria contenida en la palabra de memoria apuntada por \$c.

**Solución:**

■ **Análisis:**

a) **Especificación precisa de la semántica de la instrucción:**

La semántica de “beqri \$a, \$b, \$c” es:

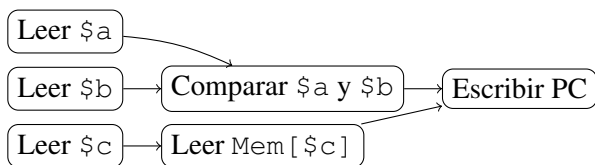
$$\text{si } \$a = \$b \Rightarrow PC \leftarrow \text{Mem}[\$c]$$

b) **Identificación del trabajo a realizar por cada unidad funcional principal:**

- Banco de registros:
  - Leer \$a.
  - Leer \$b.
  - Leer \$c.
- ALU:
  - Comparar \$a y \$b (restándolos).
- Memoria:
  - Leer Mem[\$c].

Además, se escribe en el registro PC el valor leído de memoria si el resultado de la ALU es cero.

c) **Establecimiento del orden de precedencia entre las distintas tareas a realizar:**



■ **Diseño:**

a) **Definición de la codificación de la instrucción:** Debido a que la instrucción tiene tres operandos en registros, debemos usar el formato R. Usaremos la siguiente distribución de campos:



Es importante colocar el registro c en el campo **rt** (o, alternativamente, en el **rs**) para poder leerlo en el ciclo 2 sin tener que modificar el comportamiento del procesador en dicho ciclo.

b) **División del trabajo en ciclos:**

Ciclo 1	Ciclo 2	Ciclo 3	Ciclo 4
IR ← Memoria[PC]	A ← \$a	A ← \$a	si A = B entonces PC ← MDR
PC ← PC + 4	B ← \$c	B ← \$b	
		MDR ← Memoria[B]	

Nótese que la comparación de A y B y la escritura condicional en PC se puede realizar en el mismo ciclo, de forma análoga a como se hace en el ciclo 3 de la instrucción beq. La escritura condicional se implementará usando la señal *cero* de la ALU y la señal de control *EscrPCCond*.

c) **Extensión del camino de datos:**

No podemos realizar algunas de las acciones especificadas en la tabla anterior con el camino de datos visto en clase. Por tanto, tendremos que extenderlo de la siguiente manera:

- Para  $B \leftarrow \$b$ : Añadir un multiplexor en la entrada «Reg. de lectura 2». Tendrá dos entradas y estará controlado por una nueva señal *RegOri2* con los siguientes significados:
  - *RegOri2* = 0: El registro que se lee y se almacena en B viene dado por el campo **rt** (como en el camino no modificado).
  - *RegOri2* = 1: El registro que se lee y se almacena en B viene dado por el campo **rd**. *RegOri2* tomará el valor 0 en los estados preexistentes del autómata de control.
- Para  $MDR \leftarrow Memoria[B]$ : Añadir una entrada nueva al multiplexor controlado por la señal de control *IoD* que esté conectada a la salida del registro B. La señal *IoD* pasa a tener 2 bits de ancho y su significado será:
  - *IoD* = 00: El PC suministra la dirección para acceder a memoria.
  - *IoD* = 01: ALUOut proporciona la dirección para acceder a memoria.
  - *IoD* = 10: El registro B proporciona la dirección para acceder a memoria.
- Para si  $A = B$  entonces  $PC \leftarrow MDR$ : Conectar la salida del registro MDR con la entrada libre disponible en el multiplexor controlado por la señal *FuentePC*. El significado de esta señal según su valor será ahora:
  - *FuentePC* = 00: La salida de la ALU se envía para su escritura en el PC.
  - *FuentePC* = 01: El contenido de ALUOut se envía para su escritura en el PC.
  - *FuentePC* = 10: El campo **j** desplazado dos bits a la izquierda y concatenado con los 4 bits más significativos del registro PC se envía para su escritura en el PC.
  - *FuentePC* = 11: El contenido de MDR se envía para su escritura en el PC.

d) **Extensión del control:**

Habrà que añadir 2 nuevos estados al autómata de control original. El resultado se muestra en la figura E4.3.

13. `pushm ($a)`

Apila el valor contenido en la dirección de memoria contenida en el registro \$a.

**Solución:**■ **Análisis:**a) **Especificación precisa de la semántica de la instrucción:**

La semántica de “`pushm ($a)`” es:

$$\begin{aligned} \$29 &\leftarrow \$29 - 4 \\ Mem[\$29] &\leftarrow Mem[\$a] \end{aligned}$$

(suponiendo que ambas operaciones se realizan secuencialmente)

b) **Identificación del trabajo a realizar por cada unidad funcional principal:**

- Banco de registros:
  - Leer \$a.
  - Leer \$29.
  - Escribir \$29.
- ALU:
  - Calcular  $\$29 - 4$ .
- Memoria:

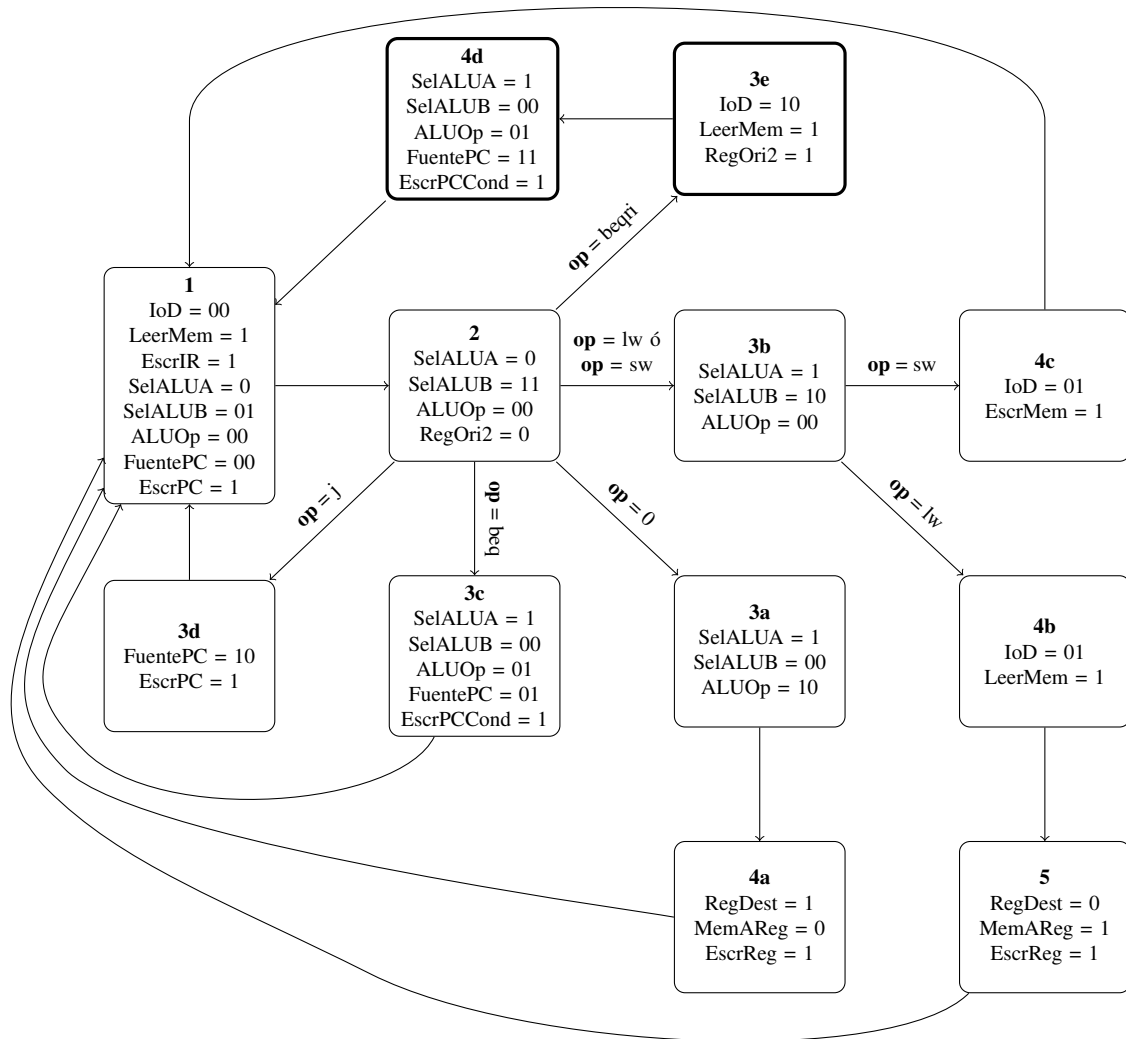
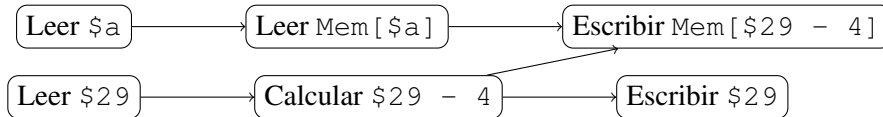


Figura E4.3: Autómata de control modificado para incluir la instrucción `beqri`. Los estados añadidos han sido resaltados.

- o Leer Mem [ \$a ].
- o Escribir Mem [ \$29 ].

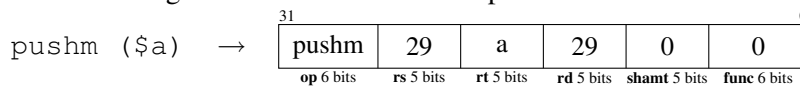
c) **Establecimiento del orden de precedencia entre las distintas tareas a realizar:**



■ **Diseño:**

a) **Definición de la codificación de la instrucción:** Aunque la instrucción tiene un operando en memoria, éste se especifica sin desplazamiento. Por tanto, podemos utilizar el formato R para codificarla. Utilizando este formato, podemos usar los campos **rs** y **rt** para codificar el registro 29 en el cual hay que leer y escribir.

Usaremos la siguiente distribución de campos:



b) **División del trabajo en ciclos:**

Ciclo 1	Ciclo 2	Ciclo 3	Ciclo 4
IR ← Memoria[PC]	A ← \$29	ALUOut ← A - 4	Memoria[ALUOut] ← MDR
PC ← PC + 4	B ← \$a	MDR ← Memoria[B]	\$29 ← ALUOut

c) **Extensión del camino de datos:**

Para realizar algunas de las acciones especificadas en la tabla anterior necesitamos modificar el camino de datos:

**MDR ← Memoria[B]:** Añadir una nueva entrada al multiplexor controlado por la señal de control IoD que esté conectada al registro B (en el que tenemos el contenido de \$a). La señal IoD pasará a ser una señal de dos bits en lugar de uno, y la combinación “10” seleccionará la nueva entrada.

**Memoria[ALUOut] ← MDR:** Añadir un multiplexor a la entrada “Dato a escribir” de la memoria que estará controlado por una nueva señal que llamaremos MemSrc. Esta señal podrá tomar el valor 0 para que el dato que se escriba en la memoria sea el contenido en el registro B o 1 para que el dato a escribir sea el contenido en el registro de datos de memoria (MDR). La nueva señal tomaría el valor 0 en los estados preexistentes del autómata de control.

d) **Extensión del control:**

Habrá que añadir 2 nuevos estados al autómata de control original. El resultado se muestra en la figura E4.4.

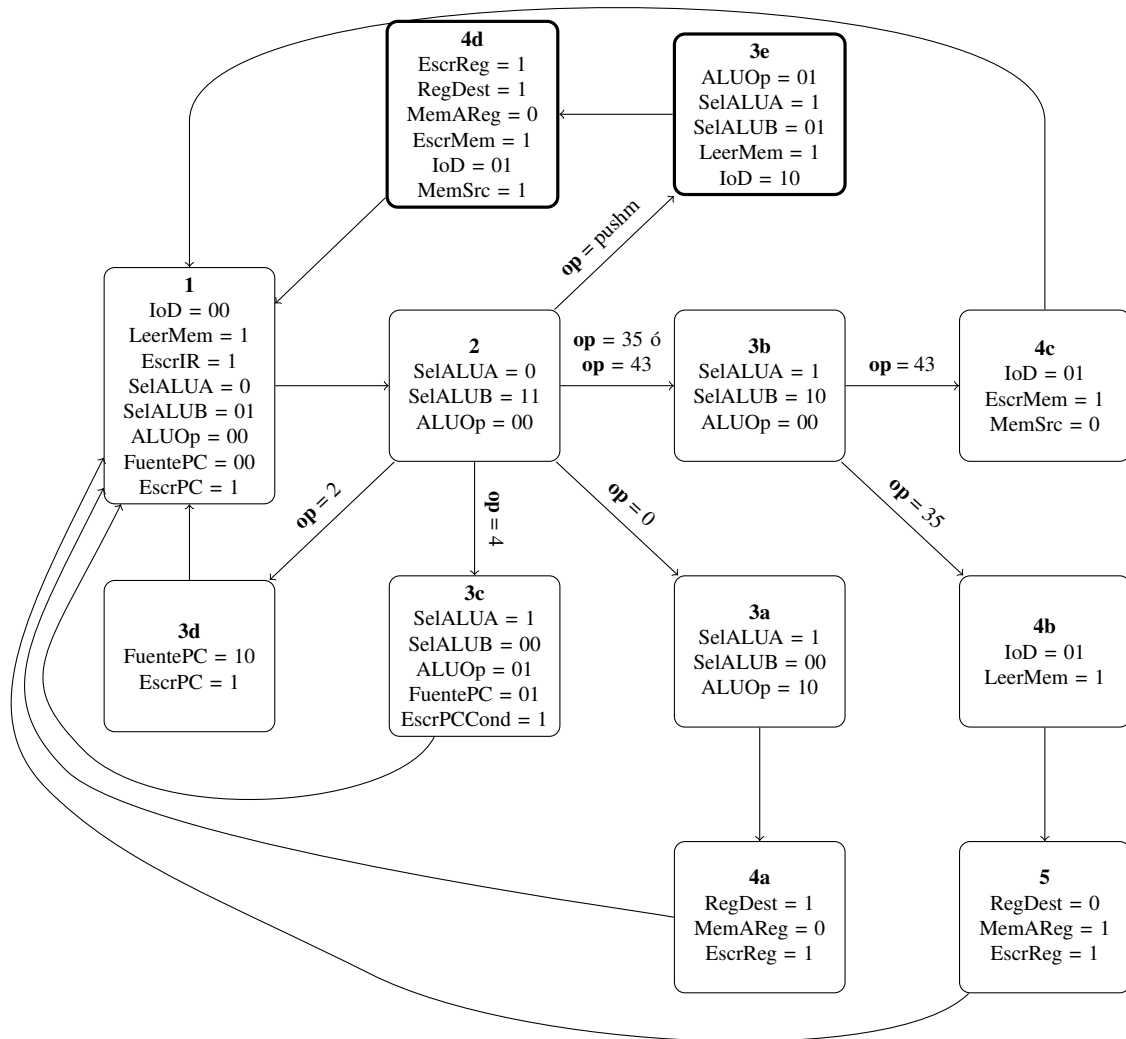


Figura E4.4: Modificaciones al autómata de control para incluir la instrucción pushm.