ESTRUCTURA Y TECNOLOGÍA DE COMPUTADORES

Departamento de Ingeniería y Tecnología de Computadores

Febrero 2021

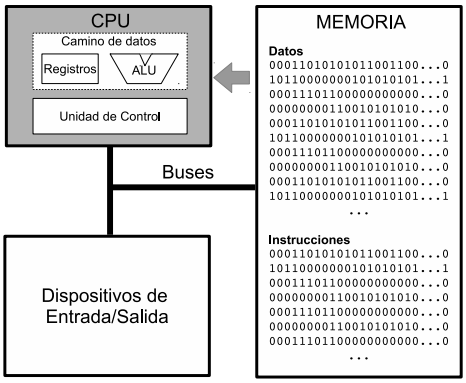
ÍNDICE GENERAL

3. Diseño de un procesador **1**

3.1. Introducción 1
3.2. Repertorio de instrucciones MIPS 2
3.3. Codificación de las instrucciones en MIPS 5
3.3.1. Modos de direccionamiento en MIPS 5
3.3.2. Formatos de instrucción en MIPS 7
3.3.3. Formato de instrucción R 7
3.3.4. Formato de instrucción I 8
3.3.5. Formato de instrucción J 10
3.4. Modelo del tiempo de ejecución de un programa 11
3.5. Descomposición de la ejecución de instrucciones 12
3.6. El lenguaje de transferencia entre registros (RTL) 14
3.7. El camino de datos 16
3.8. Control del camino de datos 27
3.8.1. Señales de control del camino de datos 27
3.8.2. Control de la ALU 27
3.8.3. Implementación cableada de la unidad de control 29
3.9. Metodología para la inclusión de nuevas instrucciones 30
3.9.1. Análisis 32
3.9.2. Diseño 34
3.9.3. Ejemplo: `bltzal` 36
3.10. Ejercicios: cálculo del tiempo de CPU 40
3.11. Solución a ejercicios seleccionados de cálculo del tiempo de CPU de programas sencillos 42
3.12. Ejercicios: introducción de nuevas instrucciones 46
3.13. Solución a ejercicios de introducción de instrucciones 47

CAPÍTULO 3

DISEÑO DE UN PROCESADOR



3.1 INTRODUCCIÓN

El objetivo de este tema es presentar el diseño de un procesador sencillo, capaz de ejecutar programas escritos para el repertorio de instrucciones de MIPS, que está descrito con detalle en el apéndice de este libro.

Por razones didácticas, se abordará la implementación de un número limitado de instrucciones del repertorio de MIPS. Sin embargo, el subconjunto seleccionado de instrucciones incluye instrucciones representativas de cada tipo, de forma que, con los conocimientos adquiridos en este tema, será relativamente sencillo modificar el procesador diseñado para incluir instrucciones adicionales. En concreto, las instrucciones que se implementarán serán:

- Instrucciones aritmético-lógicas: `add`, `sub`, `and`, `or`, `ori` y `slt`.
- Instrucciones de acceso a memoria: `lw` y `sw`.
- Instrucciones de salto condicional: `beq`.
- Instrucciones de salto incondicional: `j`.
- Instrucciones de carga de constantes: `lui`.

Para construir el camino de datos, usaremos unidades funcionales combinacionales y secuenciales que han sido explicadas con anterioridad, tanto en la asignatura de *Fundamentos de Computadores* como en los temas 1 y 2 de este libro: puertas lógicas, multiplexores, sumadores, ALUs, extensores de signo, desplazadores, flip-flops, registros, bancos de registros, memorias, etc.

En la figura 3.1 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.

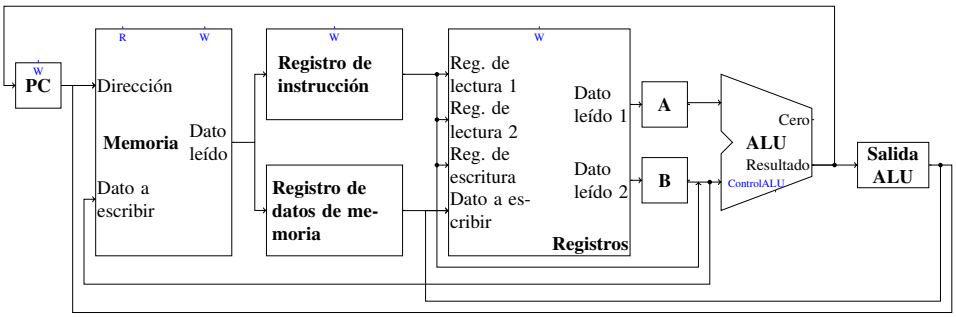


Figura 3.1: Visión de alto nivel de la implementación del procesador.

3.2 REPERTORIO DE INSTRUCCIONES MIPS

El repertorio de instrucciones de un procesador (también conocido como ISA, siglas de *Instruction Set Architecture*) define el conjunto de instrucciones ejecutables por el mismo, y en general, los aspectos del procesador visibles por el programador, como son el número y tipo de los registros, los modos de direccionamiento y el manejo de las excepciones, entre otros.

MIPS es un procesador que implementa un ISA RISC (*Reduced Instruction Set Computer*). Este tipo de repertorios de instrucciones se caracteriza por disponer de un número relativamente reducido de instrucciones, en su mayoría simples, que operan con datos en registros del procesador, y que utilizan un número pequeño de modos de direccionamiento para especificar sus operandos. Las instrucciones en un ISA RISC se codifican empleando el mismo número de bits para todas ellas (instrucciones de tamaño fijo) y a partir de un reducido número de formatos de instrucción. En un repertorio de instrucciones RISC típico, las únicas instrucciones que acceden a memoria son las instrucciones de carga de memoria y almacenamiento en memoria (instrucciones de acceso a memoria), por lo que a este tipo de arquitectura se la conoce también como arquitectura de carga-almacenamiento.

La implementación del procesador que vamos a llevar a cabo va a estar centrada en una versión simplificada del ISA MIPS de 32 bits. En concreto vamos a mostrar la implementación de unas pocas instrucciones características de cada tipo. Eso sí, todas las instrucciones van a ser codificadas utilizando 32 bits, que será también el tamaño de los registros (vamos a disponer de 32 registros de propósito general y algunos resgistros especiales, todos de 32 bits cada uno) y de las direcciones de memoria. Llamaremos «**palabra**» a un grupo de 4 bytes (32 bits), que es, por lo dicho anteriormente, el tamaño de los datos con los que el procesador trabaja normalmente.

En concreto, las instrucciones que vamos a implementar van a estar agrupadas en las siguientes 4 categorías:

- **Instrucciones aritmético-lógicas**: son las instrucciones encargadas de realizar las operaciones aritméticas (sumas, restas, multiplicaciones, divisiones, ...) y lógicas (and, or, not, xor, desplazamientos de bits, ...). En MIPS, estas instrucciones tienen 3 operandos, dos de los cuales actúan como operandos fuentes y el terce-

ro es el destino[1]. El operando destino y el primer operando fuente son siempre registros, y suelen aparecer en primer y segundo lugar respectivamente en la instrucción ensamblador. El segundo operando fuente puede ser un registro (en las instrucciones aritmético-lógicas con operandos en registros) o una constante (en las instrucciones aritmético-lógicas con un operando inmediato).

A pesar de que existe una gran variedad de instrucciones aritmético-lógicas en el repertorio de un procesador RISC, su implementación es bastante similar. En este tema nos vamos a concentrar en 6 de estas instrucciones, 5 de las cuales tienen sus dos operandos fuente en registros, mientras que la restante usa como segundo operando una constante:

```
add $a, $b, $c   # $a = $b + $c
sub $a, $b, $c   # $a = $b - $c
and $a, $b, $c   # $a = $b and $c
or  $a, $b, $c   # $a = $b or $c
slt $a, $b, $c   # $a = si $b < $c entonces 1, en otro caso 0
ori $a, $b, imm  # $a = $b or imm
```

En general, la especificación semántica de estas instrucciones es:

```
$a ← $b func $c
```

Donde *func* representa la operación a realizar (+, -, conjunción, disyunción o comparación $\leq$).

- **Instrucciones de acceso a memoria**: dado que el número de registros del banco de registros de un procesador es limitado (32 en nuestro caso), la mayoría de los datos de un programa estarán en memoria principal. Como ya hemos comentado, en un procesador RISC las instrucciones aritmético-lógicas operan con los datos en registros. Por tanto, es necesario disponer dentro del ISA de instrucciones capaces de traer datos de la memoria a los registros (instrucciones de carga de memoria) y viceversa (instrucciones de almacenamiento en memoria).

  Desde el punto de vista del programador en ensamblador (o del compilador), la memoria está estructurada como una gran tabla unidimiensional y la dirección actúa como índice en esa tabla. En la versión de MIPS que estamos considerando, las direcciones tienen siempre 32 bits y cada una de las 2^{32} direcciones se refieren a un byte de memoria (por lo que se dice que se usa «direccionamiento en bytes»). De esta forma, hay espacio para 2^{32} bytes (4 GiB), o alternativamente, 2^{30} palabras de 4 bytes, almacenadas usando 4 direcciones de memoria consecutivas (una por cada byte de la palabra).

  En nuestra implementación de procesador vamos a considerar dos instrucciones de acceso a memoria: una de carga de palabra y otra de almacenamiento de palabra. La instrucción `lw $a, desp($b)` copia en el registro `$a` el contenido de la palabra de memoria cuya dirección es el resultado de sumar el contenido del registro `$b` (registro base) con el desplazamiento (`desp`) especificado en la instrucción como un número entero con signo de 16 bits. La instrucción `sw $a, desp($b)` realiza la tarea complementaria, esto es, copia el contenido del registro `$a` en 4 bytes de memoria a partir de la posición resultado de sumar el contenido del registro `$b` con el desplazamiento.

[1] Las instrucciones de multiplicación y división no especifican registro destino, sino que está implícito. Los resultados son almacenados en dos registros especiales del procesador denominados *HI* y *LO*.

La especificación semántica de `lw $a, imm($b)` es:

```
$a ← Memoria[$b + imm]
```

Y la de `sw $a, imm($b)` es:

```
Memoria[$b + imm] ← $a
```

- **Instrucciones de salto condicional**: los lenguajes de programación de alto nivel incluyen instrucciones que, dependiendo del valor de determinados datos, hacen que el procesador ejecute un camino del programa u otro. Para poder dar soporte a estas construcciones, el ISA incluye instrucciones de salto condicional que hacen que, en función de la condición especificada, el procesador ejecute las instrucciones situadas a partir de la dirección indicada en la instrucción de salto (cuando la condición se cumple, o lo que es lo mismo, el salto es tomado) o siga por la instrucción siguiente a la de salto (cuando la condición no se cumple, o lo que es lo mismo, el salto no se toma).

  En nuestra implementación del procesador MIPS vamos a incluir una de estas instrucciones de salto condicional. En concreto, la instrucción `beq`, que tiene la siguiente sintaxis:

  ```
  beq $a, $b, etiqueta # Salta a la dirección de memoria denotada
                       # por etiqueta si $a es igual a $b
  ```

  Y su especificación semántica es:

  ```
  $a = $b: PC ← etiqueta
  ```

  En una instrucción de salto, la dirección de memoria de la instrucción destino del salto se expresa en ensamblador normalmente mediante una etiqueta textual que equivale a la dirección absoluta de la instrucción de destino, pero se codifica de forma diferente, como se explica en la siguiente sección.

- **Instrucciones de salto incondicional**: los repertorios de instrucciones incluyen también instrucciones de salto que permiten cambiar el flujo de la ejecución de forma incondicional. Este tipo de instrucciones son conocidas como instrucciones de salto incondicional. La ejecución de un salto incondicional hace que se continúe ejecutando el programa a partir de la dirección de memoria especificada en la instrucción de salto. En el caso del procesador que vamos a diseñar, este va a ser capaz de ejecutar la instrucción `j etiqueta`, en la que la dirección de destino se expresa en ensamblador mediante una etiqueta textual. Su especificación semántica es:

  ```
  PC ← etiqueta
  ```

- **Instrucciones de carga de constantes**: en determinadas ocasiones vamos a necesitar cargar constantes en registros. Si la constante es pequeña, se podría utilizar una instrucción aritmético-lógica (por ejemplo, `ori`) para hacerlo. Sin embargo, para constantes grandes, será necesario hacer la carga en dos pasos: se cargará en primer lugar la parte de la constante que debe ir en la mitad izquierda del registro (los 16 bits más significativos), para en un segundo paso y haciendo uso de una instrucción aritmético-lógica (normalmente `ori`), hacer lo propio con los 16 bits

menos significativos del registro (los motivos por los que ha de hacerse así quedarán claros cuando expliquemos la representación interna de las instrucciones). En el caso concreto del repertorio de instrucciones MIPS, la instrucción usada para el primero de los dos pasos anteriores es `lui $a, cte`, que además pone a 0 los 16 bits menos significativos del registro indicado. Su especificación semántica es («|» representa el operador de concatenación):

```
$a ← (cte | 0x0000 )
```

3.3 CODIFICACIÓN DE LAS INSTRUCCIONES EN MIPS

Antes de pasar a ver la implementación del procesador MIPS capaz de ejecutar el subconjunto de instrucciones descrito en el apartado anterior, es imprescindible conocer la forma en la que se representan internamente las instrucciones. Al fin y al cabo, es la manera en la que el procesador las ve antes de decodificarlas y ejecutarlas.

Al igual que los datos, las instrucciones de un programa se representan dentro del computador como series de ceros y unos. Estas series se organizan en *campos*, cada uno de los cuales servirá para almacenar una información necesaria para la ejecución de la instrucción. Por ejemplo, una instrucción `slt $6, $7, $8`, una vez codificada, dispondrá al menos de un par de campos con un valor determinado para indicar que los registros a comparar son `$7` y `$8`, y no otros, otro campo adicional para indicar que el registro destino (donde escribir el resultado de la comparación) es `$6`, y, por supuesto, algún campo que indique que la operación a realizar es una comparación. A la distribución concreta de estos campos en la secuencia de bits completa para codificar la instrucción es a lo que se denomina *formato* de la instrucción. En MIPS se emplean tres formatos de instrucción para llevar a cabo la codificación de todas las instrucciones. Además, los tres formatos tienen la misma longitud: 32 bits.

Por otro lado, los modos de direccionamiento especifican cómo se codifican dentro de la instrucción los distintos operandos que ésta necesita para su ejecución. Por lo tanto, hemos de analizarlos antes de presentar los formatos de instrucción en MIPS.

3.3.1 Modos de direccionamiento en MIPS

Cinco son los modos de direccionamiento que se utilizan en MIPS para codificar los operandos de las instrucciones:

1. **Modo de direccionamiento registro**: el operando se encuentra en un registro, y en el código de la instrucción se codifica el número del mismo. Puesto que el banco de registros en nuestra implementación de MIPS dispone de 32 registros, se requieren 5 bits ($2^5 = 32$) para codificar un número de registro. Por ejemplo, la instrucción `slt` que describimos con anterioridad, dispondrá de al menos 3 campos de 5 bits cada uno para determinar cada uno de los 3 registros que utiliza.

2. **Modo de direccionamiento base más desplazamiento**: en las instrucciones de acceso a memoria (cargas y almacenamientos), la dirección de memoria a leer (carga) o a escribir (almacenamiento) se especifica en la instrucción utilizando un registro (registro base) y una constante (desplazamiento). En concreto, la dirección de memoria se obtiene sumando al contenido del registro base (que se codificará usando un campo de 5 bits de longitud) la constante incluida en el campo desplazamiento de la propia instrucción (que se codificará usando 16 bits como un entero con signo en complemento a dos).

Por ejemplo, en la instrucción `lw $8, 0x1008($29)` la dirección de memoria a leer sería el resultado de sumar el valor de `$29` (este registro codificado en un campo de 5 bits) y la constante `0x1008` (codificada en un campo de 16 bits).

3. **Modo de direccionamiento inmediato**: el valor del operando es una constante que aparece codificada en un campo de 16 bits dentro de la instrucción. Por ejemplo, el segundo operando de la instrucción `ori` se codificará usando este modo de direccionamiento. También la instrucción `lui` utiliza como segundo operando una constante de 16 bits para la que se hará uso de este modo de direccionamiento (es obvio, por lo tanto, que puesto que el tamaño de las constantes está limitado a 16 bits, constantes que necesiten un mayor número de bits para su representación deben ser cargadas en los dos pasos mencionados en el apartado 3.2).

4. **Modo de direccionamiento relativo al PC**: se utiliza para especificar la dirección de la instrucción destino de un salto condicional. Ésta se almacena como un desplazamiento con respecto al contenido del registro contador de programa (PC). En concreto, se codifica en un campo de 16 bits como un entero con signo en complemento a dos. Este entero representa el número de instrucciones (palabras) desde la instrucción que está a continuación de la instrucción de salto hasta la instrucción destino del salto. Si el salto es hacia delante será un número positivo, si es hacia atrás, será negativo.

   Por ejemplo, para la secuencia de instrucciones siguiente:

   ```
   etiqueta: add $8, $9, $10
             sub $4, $5, $6
             beq $8, $4, etiqueta
   ```

   una vez codificada la instrucción `beq`, se especificaría un desplazamiento de -3 en un campo de 16 bits.

5. **Modo de direccionamiento pseudodirecto**: la dirección destino de una instrucción de salto incondicional se almacena de forma absoluta utilizando un campo de 26 bits. Obviamente, como las direcciones de memoria son de 32 bits, no es posible almacenarlas enteras en dicho campo. Sólo se almacenan los 26 bits menos significativos a partir del tercero[2]. A la hora de decodificar la instrucción, la dirección se completa a partir de estos 26 bits hasta obtener una dirección de 32 añadiendo al final 2 bits a 0 al final y al principio los 4 bits más significativos de la dirección de la instrucción actual.

   Por ejemplo, para una instrucción `j etiqueta`, donde `etiqueta` representa la dirección de memoria `0x0040 008c`, la instrucción de salto almacenaría en los 26 bits el resultado de eliminar de dicha dirección los 4 bits más significativos y los dos bits menos significativos (que siempre serán 0). De esta forma quedaría la siguiente ristra de 26 bits: $00\,0001\,0000\,0000\,0000\,0010\,0011_{)2}$. Obsérvese que de esta forma sólo es posible codificar en una instrucción `j` direcciones de destino cuyos cuatro bits más significativos coincidan con la propia dirección de salto. En otras palabras, una instrucción `j` sólo puede saltar a instrucciones que se encuentren almacenadas en el mismo trozo de memoria, si dividimos los 4 GiB de espacio de direcciones disponible en 16 trozos de 256 MiB cada uno.

[2]Las direcciones son de instrucciones y éstas estarán almacenadas siempre en direcciones múltiplo de 4 bytes. Por tanto, los dos bits menos significativos de la dirección serán siempre 0.

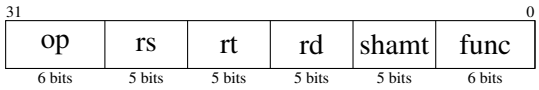
3.3.2 Formatos de instrucción en MIPS

Siguiendo la filosofía RISC, lo ideal a la hora de decidir cómo codificar las instrucciones sería elegir un formato único para todas ellas, ya que de esta forma se simplificaría el hardware necesario para realizar la decodificación. Sin embargo, esta estrategia podría acarrear problemas, tales como obtener instrucciones demasiado largas, con muchos campos sin utilizar (por ejemplo, en MIPS algunas instrucciones necesitan codificar una constante de hasta 16 bits, mientras que otras no lo necesitan y tendrían el correspondiente campo vacío). En MIPS, como solución de compromiso, existen unos pocos formatos de instrucción[3] de 32 bits todos ellos. A la hora de decidir qué formato utilizar para cada instrucción, el factor determinante es qué modos de direccionamiento usa dicha instrucción.

A continuación se pasan a describir los tres formatos de instrucción (formatos R, I y J) disponibles en la arquitectura MIPS, con ejemplos de cómo son utilizados por los diferentes tipos de instrucciones, acomodándose en cada caso al modo de direccionamiento empleado.

3.3.3 Formato de instrucción R

En este formato de instrucción, los operandos fuente y destino se especifican usando el *modo de direccionamiento registro*. El modo concreto de distribuir los 32 bits de la instrucción es el siguiente (el bit 31, de más peso o más significativo, se encuentra en el extremo izquierdo):



El significado de los distintos campos es el siguiente:

op: indica el código de operación. Se utiliza para diferenciar los distintos tipos de instrucciones. Este campo es el único que aparece en todos los formatos de instrucción (no sólo en las de tipo R, sino también las de tipo I y J), puesto que sólo una vez que se lee, se sabe el tipo de instrucción de que se trata, y por tanto, el formato empleado en la codificación de la instrucción.

rs: primer registro fuente de la operación.

rt: segundo registro fuente de la operación.

rd: registro del operando destino, donde se guardará el resultado de la instrucción.

shamt: tamaño del desplazamiento. Sólo se utiliza para almacenar la cantidad de desplazamiento en las instrucciones de desplazamiento o rotación (`sll`, `srl`...), y vale 0 en el resto.

func: sirve para distinguir instrucciones que, por ser muy similares, tienen el mismo código de operación (por ejemplo, todas las aritmético-lógicas que trabajan con

[3] Otra solución posible es la adoptada en los repertorios de instrucciones CISC (*Complex Instruction Set Computer*) x86 (también llamado IA-32) y x86-64 (también conocido como amd64). En estos ISAs, las instrucciones tienen longitud variable y la decodificación es un proceso más complejo. De hecho, implementaciones modernas de estas arquitecturas funcionan traduciendo las instrucciones CISC a un formato interno más regular antes de ejecutarlas (tipo RISC).

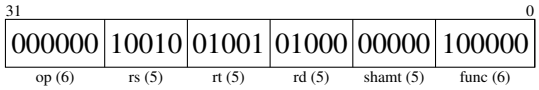
Esta opción, sin embargo, es contraria a la filosofía RISC seguida por el repertorio de instrucciones MIPS, una de cuyas características principales es la uniformidad en el tamaño de sus instrucciones.

tres registros). Le indica a la ALU qué función de las posibles (suma, resta, AND lógico...) debe realizar.

El formato R se utiliza en las instrucciones aritmético-lógicas que operan sobre dos registros fuente para guardar el resultado en un determinado registro destino. Por ejemplo, la instrucción

```
add $8,$18,$9
```

se representaría de la siguiente manera:

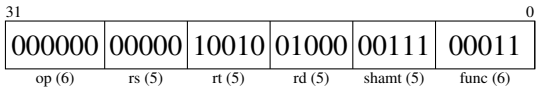


El código de operación **op** indica que se trata de una instrucción aritmético-lógica (**op**=0) con dos registros fuente (**rs**=18, **rt**=9) y un operando destino (**rd**=8). El campo **shamt** no se utiliza, puesto que no es una instrucción de desplazamiento, y se deja a 0. Finalmente, el campo **func** determina el tipo concreto de operación, en este caso una suma (**func**=32). Por tanto, esta instrucción estará almacenada en memoria como la ristra de bits 0000 0010 0100 1001 0100 0000 0010 0000 (0x02494020, en hexadecimal).

De forma similar, la instrucción de desplazamiento[4]

```
sra $8,$18,7
```

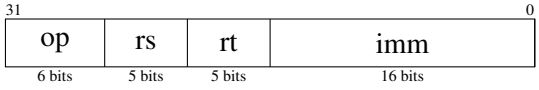
se representaría de la siguiente manera:



En este caso, el campo **func** indica que se trata de un desplazamiento aritmético hacia la derecha, y el campo **shamt** contiene el número de bits a desplazar. El registro a desplazar se encuentra codificado en el campo **rt**, mientras que el campo **rs** no se utiliza y se deja a cero.

3.3.4 Formato de instrucción I

Este formato se caracteriza por tener un campo dedicado a almacenar un valor inmediato de 16 bits. Se utiliza para codificar las instrucciones en las que uno de los operandos utiliza el *direccionamiento inmediato*, el *direccionamiento base más desplazamiento* o el *direccionamiento relativo al contador del programa*. Es decir, será utilizado por instrucciones aritméticas cuando uno de los operandos es un valor constante, instrucciones de acceso a memoria, instrucciones de carga de constantes y saltos condicionales. La distribución de campos es la siguiente:



op: código de operación.

rs: registro fuente.

[4]La instrucción `sra`, que no vamos a incluir en nuestra implementación del procesador MIPS, realiza un desplazamiento a la derecha replicando en los bits más significativos el bit de signo (desplazamiento aritmético). Ver el apéndice A para una descripción detallada de las instrucciones de desplazamiento.

rt: registro fuente/destino dependiendo de la operación.

imm: valor inmediato. Su interpretación depende de la operación.

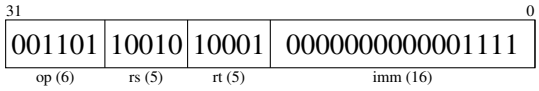
A continuación mostramos como codificar los diferentes tipos de instrucción que utilizan este formato:

Instrucciones aritméticas con un operando inmediato

Estas instrucciones utilizan el campo **imm** para codificar el valor especificado como segundo operando (un entero con signo codificado en complemento a 2 o un entero sin signo, dependiendo de la instrucción concreta). El campo **rs** codifica el otro operando fuente y el campo **rt** codifica el registro de destino. Por ejemplo, la instrucción:

```
ori $17,$18,15
```

se codifica:

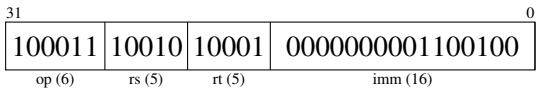


Instrucciones de acceso a memoria

Las instrucciones de acceso a memoria utilizan el campo **imm** para codificar en complemento a dos el desplazamiento que hay que sumarle al contenido del registro base, codificado en **rs**. El campo **rt** codifica el registro fuente o destino, según sea una instrucción de carga o almacenamiento, respectivamente. Por ejemplo, la instrucción:

```
lw $17,100($18)
```

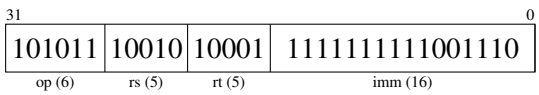
se codifica:



Las instrucciones de almacenamiento en memoria se codificarán de modo análogo, por ejemplo:

```
sw $17,-50($18)
```

se codifica:

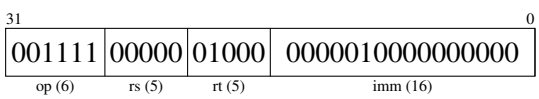


Instrucciones de carga de constantes

La instrucción `lui` utiliza el campo **imm** para codificar en complemento a dos el valor a cargar en los 16 bits más significativo del registro que se codifica en el campo **rt**, que será el registro de destino (observa que no es necesario usar el campo **rs**, cuyo valor se pondrá a 0):

```
lui $8,1024
```

se codifica:



Instrucciones de salto condicional

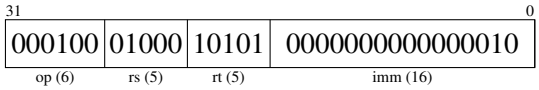
En este caso, el campo **imm** se utiliza para codificar en complemento a dos el número de instrucciones a avanzar, empezando en la instrucción siguiente al salto para llegar a la instrucción de destino (si hubiera que retroceder, se codificaría un número negativo).

Cuando se ejecuta la instrucción, se utiliza el contenido del campo **imm** y el valor del contador de programa (PC) para calcular la dirección de memoria de la instrucción de destino. En MIPS la memoria se direcciona por bytes, pero, como sabemos, cada instrucción ocupa 4 de ellos. Por tanto, una dirección de una instrucción MIPS es siempre múltiplo de 4 (los dos últimos bits siempre valdrán 00). Por esta razón no es necesario codificar en **imm** el número de bytes que hay que sumarle al PC, y se consigue abarcar más instrucciones de destino con el campo inmediato de sólo 16 bits.

Por ejemplo, la instrucción `beq` en la siguiente secuencia:

```
    beq $8,$21,L1
    sub $8,$10,$11
    ori $8,$8,1
L1: ...
```

se codificaría:

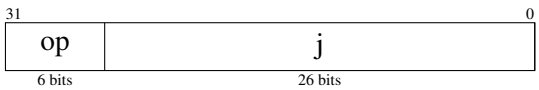


Como se puede observar, la dirección apuntada por la etiqueta se encuentra tres instrucciones después de `beq`. Además, hay que tener en cuenta que el desplazamiento es relativo al valor del contador de programa en el momento de ejecutar la instrucción `beq` (que es PC $+4$, es decir, la dirección de memoria de la instrucción siguiente a `beq`). Por tanto, el valor almacenado en el campo **imm** de la instrucción es $+2$.

3.3.5 Formato de instrucción J

Este último tipo de formato se usa principalmente para las instrucciones de salto incondicional, que usan el *direccionamiento pseudodirecto*. Estas instrucciones especifican casi directamente la dirección destino del salto, al contrario que los saltos condicionales, donde la dirección destino se codifica con respecto a la dirección de la instrucción actual.

Al tener que especificarse directamente la dirección destino, los 16 bits del campo inmediato en el formato I son insuficientes, por lo que se crea un nuevo formato. Es imposible codificar la dirección completa de la instrucción (que tiene 32 bits) de destino junto con el código de operación porque sólo hay 32 bits en cada instrucción. Por tanto, en este formato se reservan todos los bits que no son del código de instrucción (26) para la dirección, y se aprovecha de nuevo el que todas las instrucciones estén almacenadas en direcciones múltiplo de 4. Los campos son los siguientes:



op: especifica el código de operación.

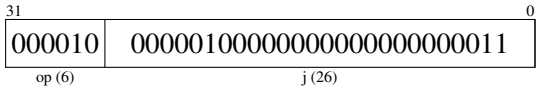
j: almacena los bits del 27 al 2 (inclusive) de la dirección destino.

Como ya se ha comentado, además de los 26 bits almacenados en el campo **j**, son necesarios 6 bits más para conocer la dirección de la instrucción de destino. Los dos bits menos significativos no es necesario codificarlos, porque serán siempre 0 al ser la dirección múltiplo de 4. Los cuatro bits restantes para formar una dirección de 32 bits se toman de los 4 bits más significativos del PC de la instrucción en curso (la de salto).

Supongamos una instrucción

```
j bucle
```

para la que la etiqueta `bucle` se corresponde con la dirección `0x0040000c`. En ese caso, suponiendo que la propia instrucción `j` se encontrase en una dirección que tuviese sus cuatro bits de más peso como ceros (es decir, fuese de la forma `0x0NNNNNNN`), la codificación de la instrucción de salto sería (en binario):



Ten en cuenta que el destino del salto está en `0x0040000c`, que expresado en binario es $00000000010000000000000000001100)_2$. Por tanto, decartando los dos bits menos significativos y expresado en binario en 26 bits, la dirección de salto se codificaría como $00000100000000000000000011)_2$.

3.4 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 1, la cantidad de estados posibles en un procesador es astronómica. Por esta razón, las técnicas que vimos en el tema 1 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 elevado.

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

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

donde:

T_{CPU}: es el tiempo total que tarda la CPU en ejecutar las instrucciones de 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[5]. También, un ISA que ofrezca instrucciones más complejas (un ISA CISC por ejemplo frente a uno RISC) permitirá expresar los programas con menor número de ellas.

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 ($CPI > 1$). Es decir, vamos a construir un procesador *multiciclo*. Así, el tiempo de ciclo puede ser mucho más pequeño que en un procesador monociclo 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 más avanzadas. 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:

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

Esto es especialmente útil para programas cortos en los que es posible hallar exactamente el número de ciclos necesarios para su ejecución si se cuenta el número de veces que se ejecuta cada instrucción y además se sabe cuántos ciclos tarda en ejecutarse cada una:

$$N_{ciclos} = \sum_{\substack{\text{para cada} \\ \text{instrucción} \\ \text{del código}}} (\text{veces que se ejecuta})(\text{ciclos por ejecución})$$

3.5 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

[5]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.

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 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, etc) introducen retardos demasiado pequeños en comparación a los de los elementos que hemos mencionado como para tenerlos en cuenta en nuestra versión simplificada del procesador. 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 (memoria, banco de registros o ALU), 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[6]. 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[7]. Si no las realizáramos

[6] En realidad, un banco de registros como el visto en el tema 2 se puede utilizar para leer dos registros y escribir otro. En cierto modo, podríamos considerar cada uno de los dos puertos de lectura o escritura como unidades funcionales independientes (al menos a efectos de su utilización).

[7] No hay coste en términos de tiempo de ejecución, aunque sí hay un cierto coste en términos de energía.

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`, `ori` 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).

4b. Lectura del dato de memoria (memoria).

5. Escritura del dato leído en el registro de destino (banco de registros).

En el caso de una instrucción de escritura en memoria (`sw`), harán falta dos pasos además de los que son comunes a todas las instrucciones, el primero de los cuales es común al caso anterior:

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

4c. Escritura del dato en memoria (memoria).

En el caso de un salto condicional, sólo es necesario un paso más:

3c. Comprobación de la condición del salto (ALU) y actualización del contador de programa si procede.

En el caso de un salto incondicional, se necesita un paso adicional también:

3d. Actualización del contador de programa.

Y finalmente, en el caso de una instrucción de carga de datos (`lui`), un tercer paso nos bastará también para completar su ejecución:

3e. Escritura del registro de destino (banco de registros).

En la figura 3.2 se muestra un resumen de los pasos que se han mencionado anteriormente.

3.6 EL LENGUAJE DE TRANSFERENCIA ENTRE REGISTROS (RTL)

Antes de comenzar con el diseño del procesador haremos una breve introducción al lenguaje que utilizaremos para describir el flujo de información entre los diversos elementos (banco de registros, memoria, ALU, registros auxiliares, etc.) que componen nuestro procesador. Así, para cada uno de los pasos en que se han dividido la ejecución de una instrucción, indicaremos el trabajo a realizar utilizando un lenguaje de transferencia entre registros (en inglés, *Register Transfer Language*) en lugar de utilizar un lenguaje natural. Por ejemplo, en lugar de decir que en el primer ciclo hay que realizar la «Lectura de la instrucción desde la memoria», escribiremos:

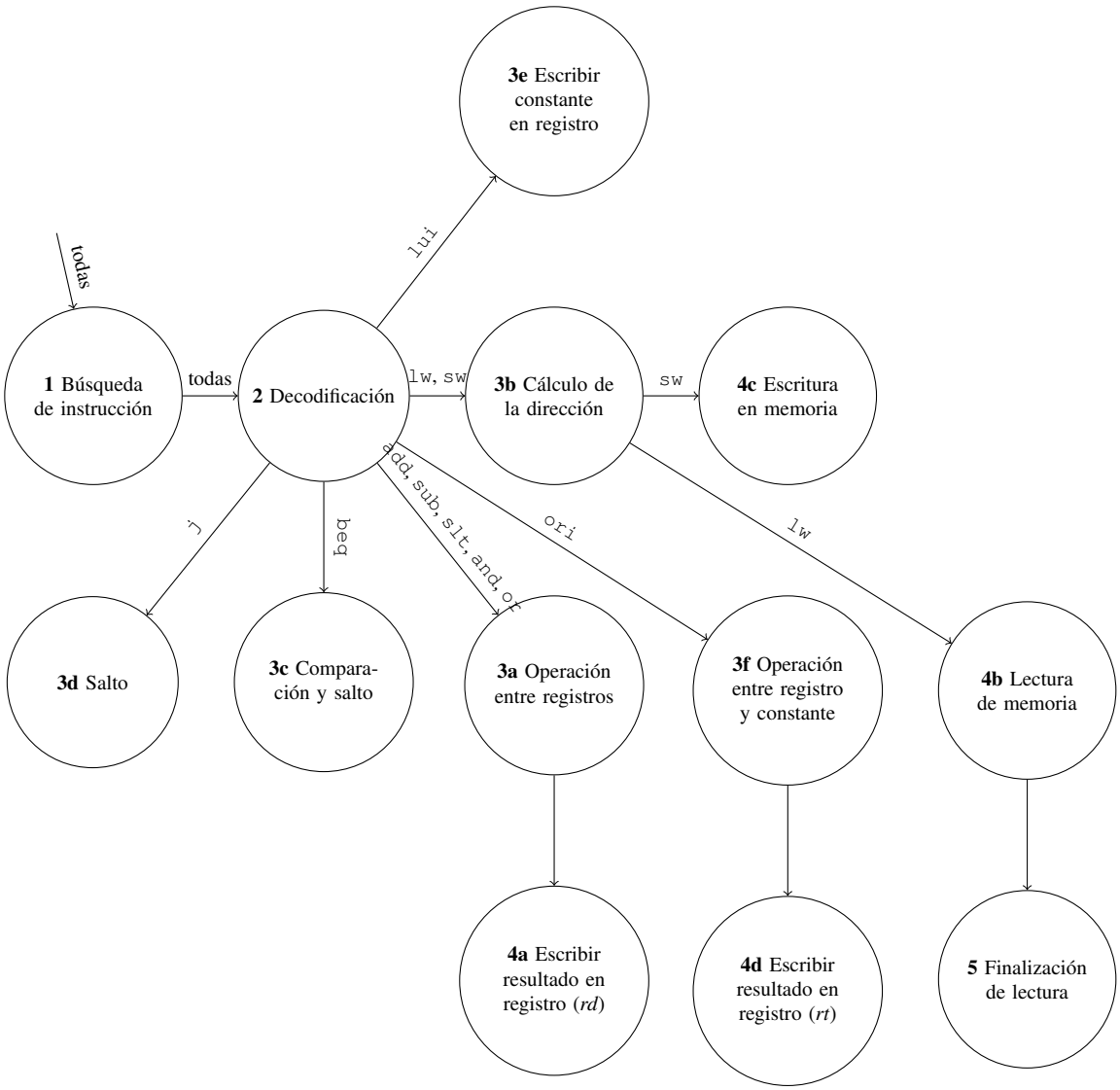


Figura 3.2: Resumen de la división en pasos de las instrucciones.

```
IR ← Memoria[PC]
```

en donde estamos indicando que queremos transferir al registro `IR` el contenido de la palabra almacenada en la posición de la memoria indicada por el registro `PC`. Como veremos en la siguiente sección, tanto el registro `PC` como el `IR` son registros auxiliares que formarán parte de nuestro diseño. Toda sentencia en RTL debe de ejecutarse en un ciclo de reloj.

Frecuentemente, una sentencia RTL se debe ejecutar sólo cuando se cumple determinada condición booleana. Utilizaremos la notación:

```
C: Y ← Z
```

para indicar a la izquierda del símbolo «:» la condición booleana (`C`) que se debe cumplir para realizar la acción. Esta condición booleana puede incluir los operadores booleanos de disyunción, conjunción y negación ($+$, $\cdot$ y $\bar{}$). Alternativamente, se puede utilizar la notación más larga siguiente:

```
si (C) entonces Y ← Z
```

Todas las sentencias RTL se ejecutan en un ciclo, y es frecuente que varias sentencias se ejecuten simultáneamente (en paralelo). Para indicarlo, se separan mediante comas las sentencias que se ejecutan en paralelo. Por ejemplo, la siguiente línea especifica tres sentencias que se ejecutan a la vez:

```
X ← Y, A ← B + C, si (X=2) Z ← B
```

Las sentencias RTL no están limitadas únicamente a la transferencia entre registros. Por ejemplo, los operandos pueden ser un único biestable, un subconjunto del contenido de un registro (sólo unos bits del registro), una posición de memoria o una línea de bus. Para designar una parte específica de un registro, utilizaremos la notación `R[k-m]` para indicar los bits `k` a `m` (ambos incluidos) del registro `R`. De forma similar, para indicar una posición de memoria con dirección `R` utilizaremos `Memoria[R]`, donde `Memoria` representa la memoria principal y `R` la dirección de la palabra en la memoria.

3.7 EL CAMINO DE DATOS

Descompondremos el diseño del procesador en dos partes: el *camino de datos* y la *unidad de control*. El camino de datos es el conjunto de elementos del procesador que realiza el trabajo indicado por las instrucciones, mientras que la unidad de control se encarga de dirigir el trabajo del camino de datos activando 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 1. 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 mejorar la claridad de los diagramas.

En la figura 3.1 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, como el visto en el tema 2. 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, tal y como hemos visto en el tema 2.

Dispone de una señal de control de 3 bits, `ControlALU`, que permite seleccionar la operación a realizar según se muestra en la tabla 3.1.

`ControlALU`	**Operación**
000	AND
001	OR
010	Suma
110	Resta
111	`slt`

Tabla 3.1: 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. Recuérdese 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 3.5. Para ello utilizaremos un lenguaje de transferencia entre registros mostrado en la sección 3.6 que nos permitirá describir de forma concisa como debe de fluir la información entre los registros internos que componen la ruta de datos.

- 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 3.3 muestra los elementos del camino de datos necesarios para ejecutar esta etapa.

  En el registro IR se almacenará la instrucción a ejecutar. La figura 3.4 muestra los formatos utilizados por las instrucciones que implementaremos.

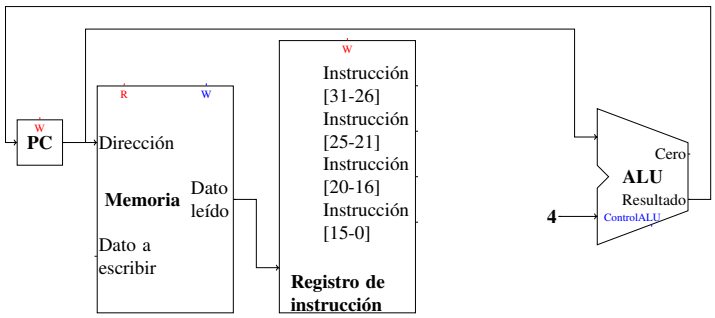


Figura 3.3: Conexión de los componentes necesarios para el paso 1.

Tipo	Formato	Instrucciones
R	31 … 0: op (6 bits) | rs (5 bits) | rt (5 bits) | rd (5 bits) | shamt (5 bits) | func (6 bits)	`add`, `sub`, `slt`, `and` y `or`
I	31 … 0: op (6 bits) | rs (5 bits) | rt (5 bits) | imm (16 bits)	`lw`, `sw`, `beq`, `lui` y `ori`
J	31 … 0: op (6 bits) | j (26 bits)	`j`

Figura 3.4: 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 [8] :

```
A ← Reg[IR[25-21]], B ← Reg[IR[20-16]],
ALUOut ← PC + (extender_signo(IR[15-0]) << 2)
```

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 3.8.

[8] «`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 3.5 muestra las conexiones entre los elementos del camino de datos necesarios para ejecutar esta etapa.

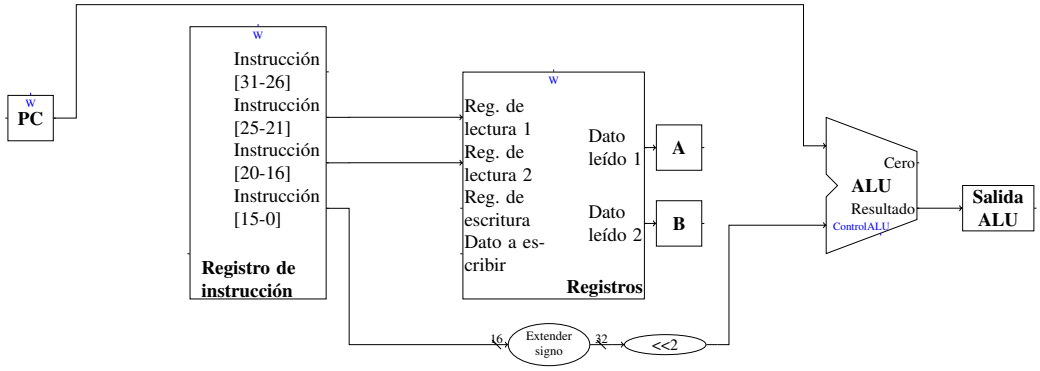


Figura 3.5: Conexión de los componentes necesarios para el paso 2

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 3.1.

- Pasos de las instrucciones aritmético-lógicas con operandos en registros:

  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 func B
  ```

  La figura 3.6 muestra las conexiones necesarias para este paso.

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

  ```
  Reg[IR[15-11]] ← ALUOut
  ```

  Las conexiones necesarias se pueden ver en la figura 3.7.

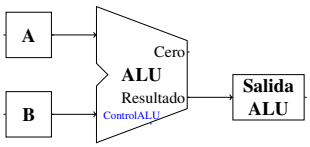


Figura 3.6: Conexión de los componentes necesarios para el paso 3a

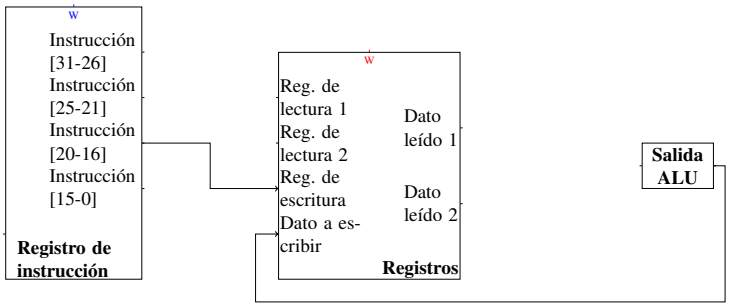


Figura 3.7: Conexión de los componentes necesarios para el paso 4a

- 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.

  ```
  ALUOut ← A + extender_signo(IR[15-0])
  ```

  Se pueden ver las conexiones utilizadas para ejecutar este paso en la figura 3.8.

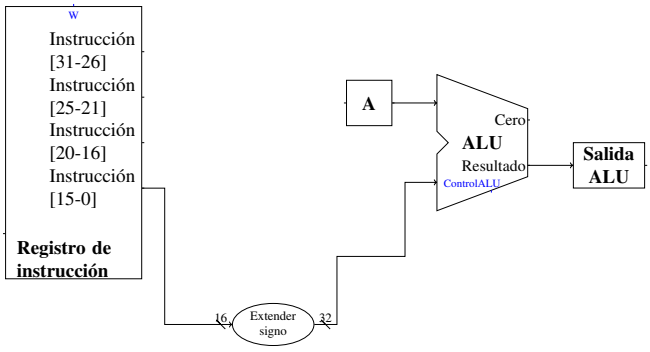


Figura 3.8: Conexión de los componentes necesarios para el paso 3b

- Pasos de las instrucciones de carga de memoria:

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

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

  ```
  MDR ← Memoria[ALUOut]
  ```

  Las conexiones a realizar para este paso se muestran en la figura 3.9.

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

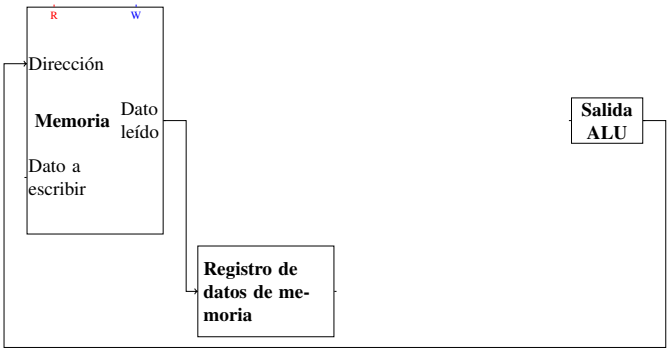


Figura 3.9: Conexión de los componentes necesarios para el paso 4b

```
Reg[IR[20-16]] ← MDR
```

La figura 3.10 muestra las conexiones entre las unidades funcionales implicadas en esta etapa.

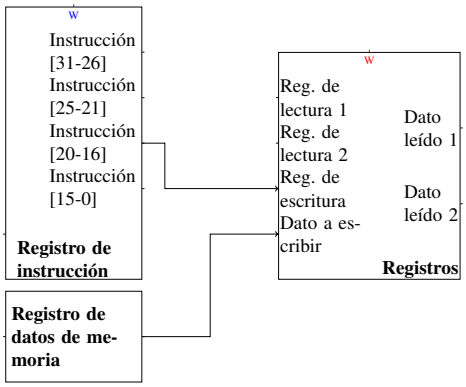


Figura 3.10: Conexión de los componentes necesarios para el paso 5

- Pasos de las instrucciones de almacenamiento en memoria:

  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.

  ```
  Memoria[ALUOut] ← B
  ```

  Las conexiones pare este paso se detallan en la figura 3.11.

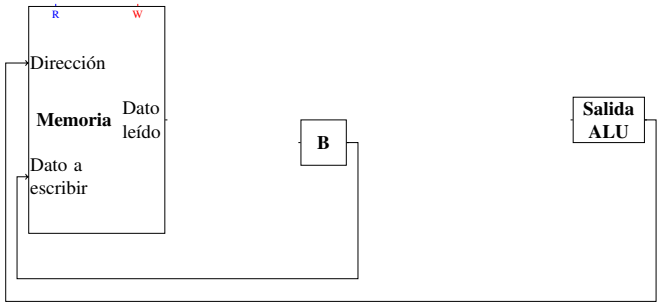


Figura 3.11: Conexión de los componentes necesarios para el paso 4c

- 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.

  ```
  A=B: PC ← ALUOut
  ```

  En la figura 3.12 se muestran los componentes necesarios para esta acción y las conexiones entre ellos.

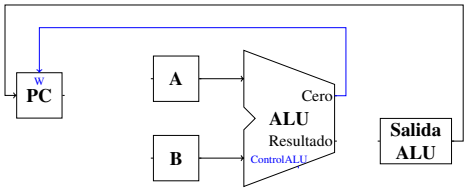


Figura 3.12: Conexión de los componentes necesarios para el paso 3c

Es importante aclarar que la escritura de un registro se hace al final del ciclo. Por tanto, el valor del registro "ALUOut", que se ha calculado en el paso 2, se escribe en el registro PC, al mismo tiempo que se modifica "ALUOut" con el resultado de la operación realizada por la ALU.

- Pasos de las instrucciones de salto incondicional:

  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] << 28) | (IR[25-0] << 2)
  ```

  Para realizar esta operación es necesario añadir un nuevo desplazador. Las conexiones resultantes se pueden ver en la figura 3.13.

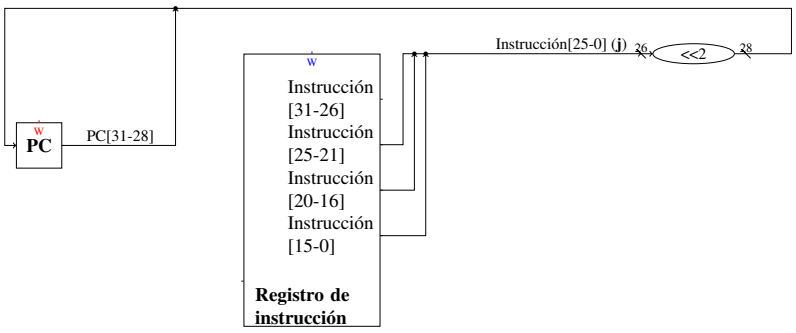


Figura 3.13: Conexión de los componentes necesarios para el paso 3d

- Pasos de las instrucciones de carga de constantes:

  La instrucción `lui` tiene un paso propio:

  3e **Escritura de constante**: En este caso, hay que escribir la constante codificada en los 16 bits menos significativos del formato I, desplazada 16 bits a la izquierda, en el registro correspondiente del banco de registros.

  ```
  Reg[IR[20-16]] ← (IR[15-0] << 16)
  ```

  Para esta paso se requiere añadir un desplazador de 16 bits a la izquierda al camino de datos. Las conexiones resultantes se pueden ver en la figura 3.14.

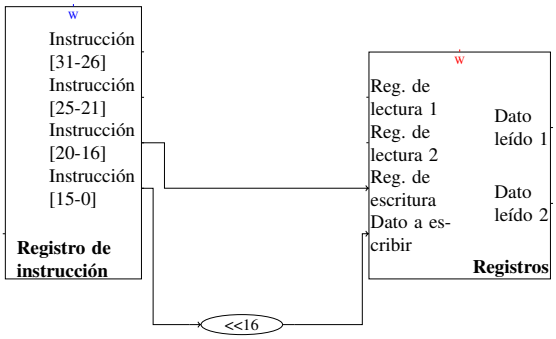


Figura 3.14: Conexión de los componentes necesarios para el paso 3e

- Pasos de las instrucciones aritmético-lógicas con un operando inmediato:

  Finalmente, los pasos específicos a la instrucción `ori` (muy similares a los descritos para las instrucciones aritmético-lógicas con operandos en registros) serían los siguientes:

  3f **Ejecución**: en esta etapa la ALU se utiliza para realizar la operación *or* requerida por la instrucción. Dado que el inmediato está codificado usando solo 16 bits y la ALU opera con datos de 32 bits, debemos concatenar 16 bits «0» a la izquierda el inmediato. El resultado de la operación *or* se almacena en el registro ALUOut y está disponible al comienzo del siguiente ciclo.

  ```
  ALUOut ← A or extender_cero(IR[15-0])
  ```

  Para este paso hemos de añadir un extensor de cero al camino de datos. La figura 3.15 muestra las conexiones necesarias para este paso.

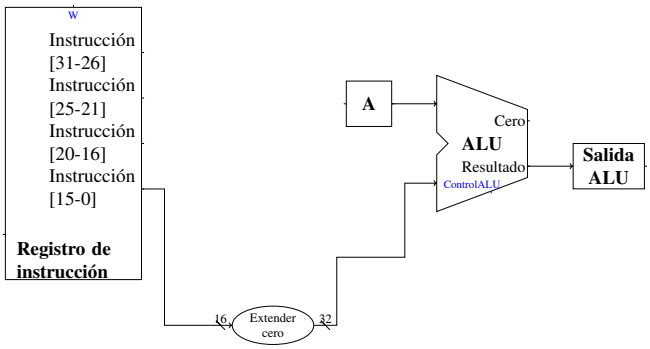


Figura 3.15: Conexión de los componentes necesarios para el paso 3f

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

```
Reg[IR[20-16]] ← ALUOut
```

Las conexiones necesarias se pueden ver en la figura 3.16.

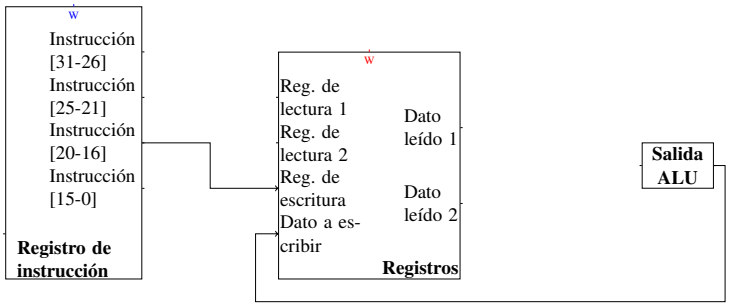


Figura 3.16: Conexión de los componentes necesarios para el paso 4d

La descripción completa en RTL del trabajo a realizar para la ejecución de cualquiera de las instrucciones implementadas sería:

```
                T1: IR ← Memoria[PC], PC ← PC + 4
                T2: A ← Reg[IR[25-21]], B ← Reg[IR[20-16]],
                    ALUOut ← PC + (extender_signo(IR[15-0]) << 2)
          T3 && (op=0): ALUOut ← A func B
T3 && (op=35 || op=43): ALUOut ← A + extender_signo(IR[15-0])
  T3 && (op=4) && (A=B): PC ← ALUOut
          T3 && (op=2): PC ← (PC[31-28] << 28) | (IR[25-0] << 2)
          T3 && (op=15): Reg[IR[20-16]] ← (IR[15-0] << 16
          T3 && (op=13): ALUOut ← A or extender_zero(IR[15-0])
          T4 && (op=0): Reg[IR[15-11]] ← ALUOut
        T4 && (op=35): MDR ← Memoria[ALUOut]
        T4 && (op=43): Memoria[ALUOut] ← B
        T4 && (op=13): Reg[IR[20-16]] ← ALUOut
        T5 && (op=35): Reg[IR[20-16]] ← MDR
```

que se obtiene de incluir todas las sentencias RTL indicadas en los distintos pasos, indicando como condición de ejecución que nos encontremos en el paso concreto (`T1, ..., T5`) y que se trate de la instrucción adecuada (en función del código de operación, `op` , guardado en `IR[31-26]`[9]).

Las figuras 3.3, 3.5, 3.6, 3.7, 3.8, 3.9, 3.10, 3.11, 3.12, 3.13, 3.14, 3.15, 3.16 muestran cómo conectar las unidades funcionales entre sí para realizar cada uno de los pasos. En cierto modo, podemos considerar que la figura 3.1 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 3.8 veremos cómo se manejan estas señales.

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

[9] op=0 para las instrucciones aritmético-lógicas, op=2 para la instrucción `j` y 4 para la `beq`, op=13 para la instrucción `ori` y op=15 para la instrucción `lui`. Finalmente, `lw` tiene como código de operación el 35 y `sw` el 43.

- 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» del banco de registros 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» del banco de registros, que puede ser:
  - El valor del registro ALUOut (paso 4a).
  - El valor del registro MDR (paso 5).
  - El campo **imm** de la instrucción actual después de desplazarlo 16 bits a la izquierda (paso 3e).
- 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).
  - El campo **imm** de la instrucción actual después de extenderlo con ceros a 32 bits (paso 3f).
- 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 3.17 se puede ver el camino de datos después de añadirle los multiplexores y conexiones que hemos mencionado.

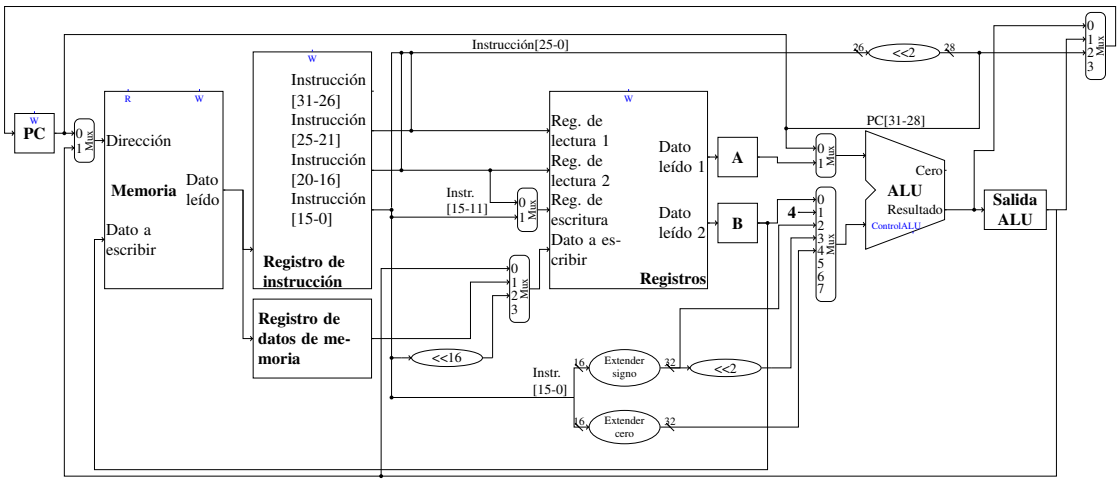


Figura 3.17: Visión completa de la implementación del camino de datos.

3.8 CONTROL DEL CAMINO DE DATOS

La figura 3.17 muestra 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 3.5 y 3.7. En la figura 3.2 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.

3.8.1 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 3.2.

3.8.2 Control de la ALU

Controlar la ALU significa generar sus tres bits de entrada de control para indicar la operación a realizar según la tabla 3.1. Como se puede ver en la sección 3.7, 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**.

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	La ALU realiza una operación OR bit a bit.
	11	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 indicado en la entrada «Registro de escritura» del banco de registros 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 **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.
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.
FuenteDato	00	El valor de entrada del banco de registros proviene del registro ALUOut.
	01	El valor de entrada del banco de registros proviene del registro MDR.
	10	El valor de entrada del banco de registros proviene del desplazador de 16 bits a la izquierda.
RegDest	0	El identificador del registro destino viene determinado por el campo **rt**.
	1	El identificador del registro destino viene determinado por el campo **rd**.
SelALUA	0	El primer operando de la ALU es el PC.
	1	El primer operando de la ALU es el registro A.
SelALUB	000	El segundo operando de la ALU es el registro B.
	001	El segundo operando de la ALU es la constante 4.
	010	El segundo operando de la ALU son los 16 bits menos significativos del registro IR, extendidos de signo.
	011	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.
	010	El segundo operando de la ALU es el resultado de extender los 16 bits menos significativos del registro IR con 16 bits cero por la izquierda.

Tabla 3.2: Señales de control del camino de datos y su significado.

- En el paso 3f, debe hacer que la ALU realice un OR bit a bit,
- 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 3.3.

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 3.3: 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 3.1. 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 3.8.3. El comportamiento que necesitamos que siga la unidad de control de la ALU en función de estos dos bits queda reflejado en la tabla 3.4.

ALUOp	**Acción**	**ControlALU**
00	Suma incondicional	010
01	Resta incondicional	110
10	OR incondicional	001
11	Según campo `func`	Ver tabla 3.3

Tabla 3.4: 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 3.18. Se puede comprobar de manera trivial que el circuito correspondiente funciona de la manera indicada en las tablas 3.3 y 3.4. Obsérvese que, en realidad, sólo 4 de los 6 bits del campo **func** tienen influencia en el valor de `ControlALU` (ya que los dos dos más sinificativos son iguales). Esta unidad de control de la ALU será añadida al camino de datos de la figura 3.17.

3.8.3 Implementación cableada de la unidad de control

El valor de las señales de control depende únicamente del paso actual de la ejecución[10]. Por otro lado, la transición de un paso a otro depende de la instrucción que se

[10]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 3.8.2

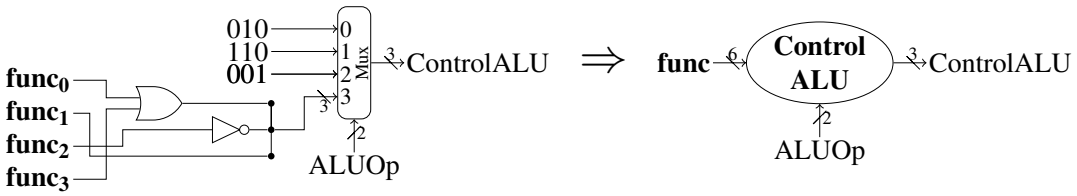


Figura 3.18: Circuitería de control de la ALU.

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 3.7. El autómata que describe el sistema secuencial síncrono a implementar se muestra en la figura 3.19. Este autómata está basado en el diagrama de la figura 3.2. 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, EscrPCCond) 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.

Como se ve en la figura 3.19, 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 13 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 3.20.

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 3.21. Obsérvese que se han añadido dos puestas 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).

3.9 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 3.1.

La metodología que seguiremos se compone de dos fases principales: análisis y di-

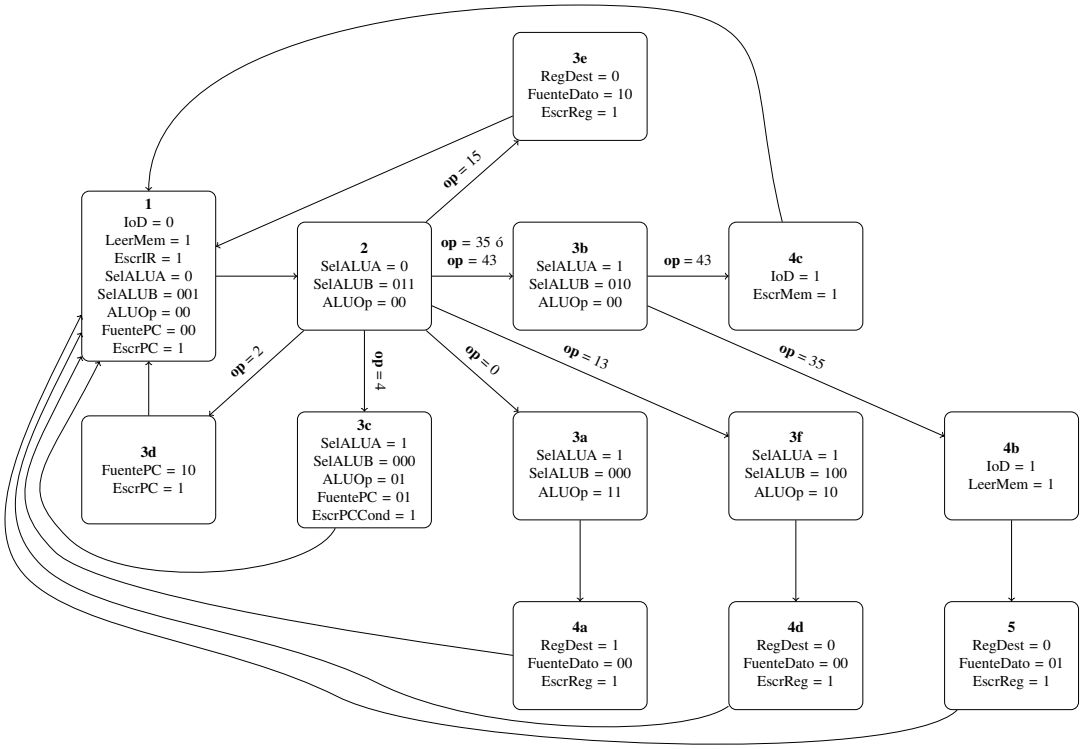


Figura 3.19: Autómata que describe la unidad de control del microprocesador. Las salidas de cada estado son todas las señales de control de la tabla 3.2. Aquellas que no se mencionen en algún estado se asume que toman el valor 0.

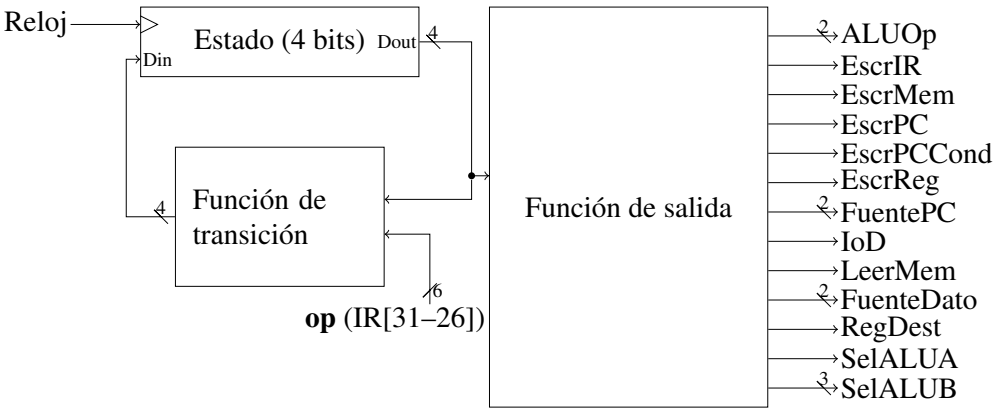


Figura 3.20: Esquema de implementación de la unidad de control.

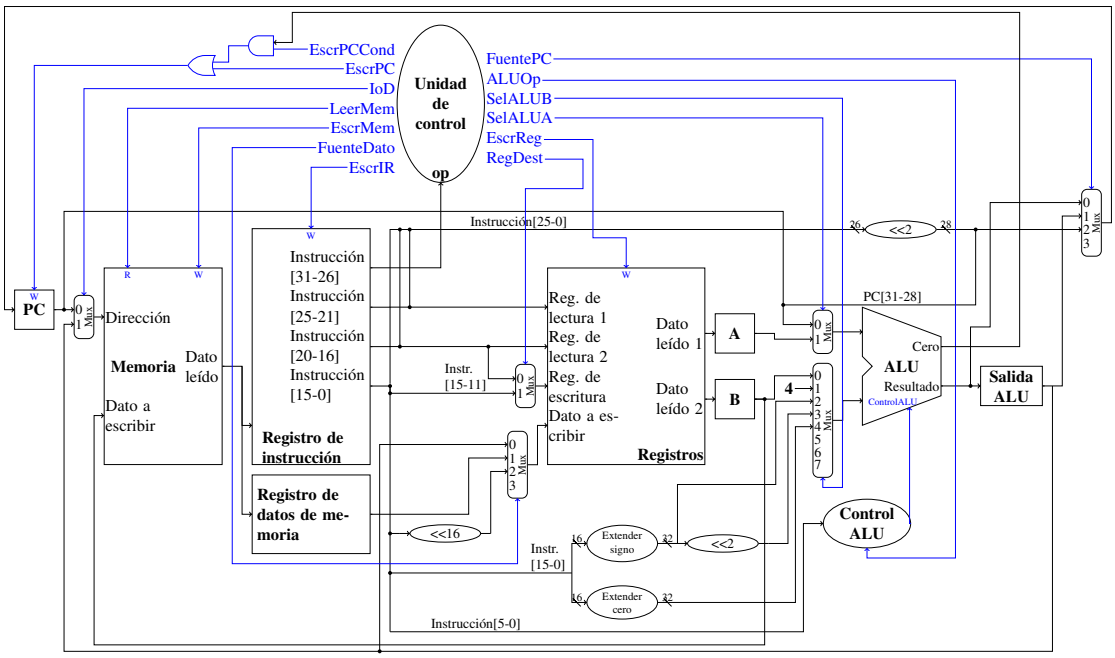


Figura 3.21: Visión completa de la implementación del procesador, incluyendo la unidad de control.

seño. Estas fases a su vez de 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).

3.9.1 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 (utilizando una notación RTL) 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). En la sección 3.2 se muestra la especificación semántica de las instrucciones ya implementadas en el procesador. 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 (es decir, debe realizarse en un ciclo posterior).

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`.

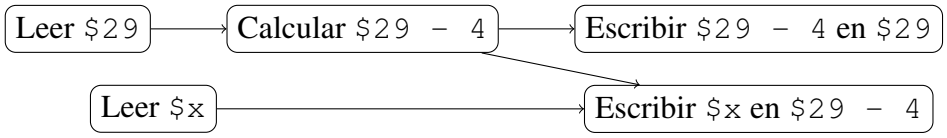
Podemos observar que al asignar trabajo a las unidades funcionales hemos refinado la división en acciones realizada en el paso anterior. Por ejemplo, la acción «`$29 ← $29 - 4`» se ha dividido en dos acciones: el cálculo de «`$29 - 4`» realizado en la ALU y la asignación del resultado a `$29`, realizado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



3.9.2 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

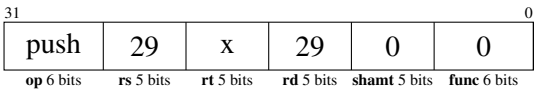
Se ha de elegir cómo se va a codificar la instrucción. La codificación elegida se debe ajustar a alguno de los tres formatos de instrucción preexistentes (a no ser que sea imposible). Por supuesto, la codificación debe permitir diferenciar la nueva instrucción de las instrucciones ya existentes.

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 ya que se necesita leer dos registros y escribir uno, por otro lado la constante 4 ya entra como parámetro en la entrada B de la ALU[11]. La codificación para la instrucción «`push $x`» quedaría:

`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.

[11] 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.

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.
- 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 ← Memoria[PC] PC ← PC + 4	A ← Reg[**rs**] B ← Reg[**rt**]	ALUOut ← A − 4	Reg[**rd**] ← ALUOut 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 autómata que define a la unidad de control (figura 3.19) indicando para los nuevos estados el valor de las señales de control para que se realicen las acciones adecuadas en cada ciclo.

El autómata modificado para la instrucción `push` se muestra en la figura 3.22.

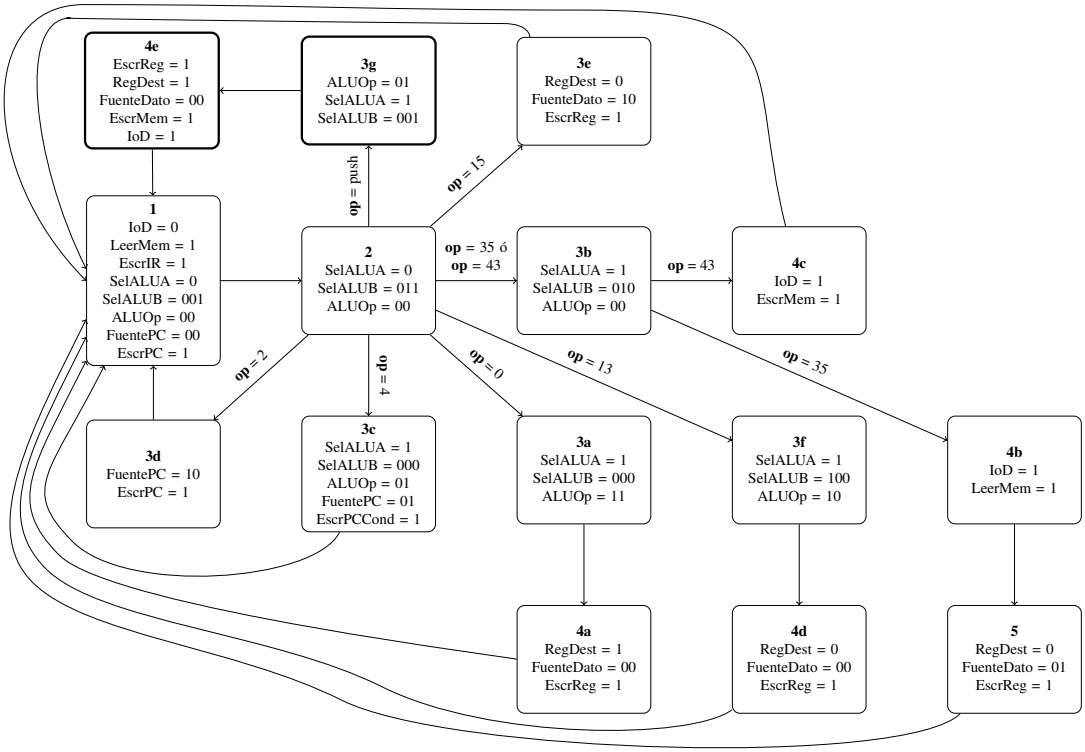


Figura 3.22: Autómata de control modificado para incluir la instrucción `push`. Los estados añadidos han sido resaltados.

3.9.3 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. Al ser un salto condicional enlazado, el valor PC+4 (que se calcula en el paso 1) debe guardarse en el registro `$ra`.

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

- **Análisis**:

  1. **Especificación precisa de la semántica de la instrucción**:
     La semántica de «`bltzal $x, label`» es:

     `($x < 0): $ra ← PC + 4, PC ← label`

  2. **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:

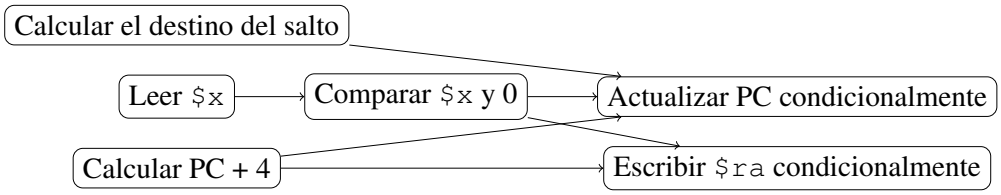
     - La forma más general de comparar dos valores es utilizar la ALU para restarlos, tal y como hace la instrucción `slt`.
     - Dado que en este caso se trata solo de saber si un valor es menor que 0, bastaría con mirar su bit de signo.

     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**:

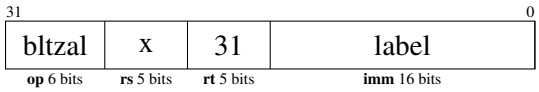


- **Diseño**:

  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[12]:

[12] Esta no es la codificación de la instrucción `bltzal` en el ISA real de MIPS.

`bltzal $x, label` →



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]	A ← Reg[**rs**]	A[31]=1: PC ← ALUOut
PC ← PC + 4	ALUOut ← PC + (**imm** << 2)	A[31]=1: Reg[**rt**] ← 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 3.21. Las modificaciones son las siguientes:

   - Para «A[31]=1: PC ← 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.
   - Para «A[31]=1: Reg[**rt**] ← PC»:
     Tenemos que conectar la salida del registro PC a la entrada de datos del banco de registros. Para ello, usaremos el multiplexor controlado por la señal de control FuenteDato. Cuando FuenteDato valga 11, se escribirá en el banco de registros el valor del registro PC (si se ecribe algo). El significado del resto de valores posibles de FuenteDato que se indican en la tabla 3.2 no cambia.
     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 3.23.

4. **Extensión del control**:

   Habrá que añadir un nuevo estado al autómata de control de la figura 3.19. El resultado se muestra en la figura 3.24.

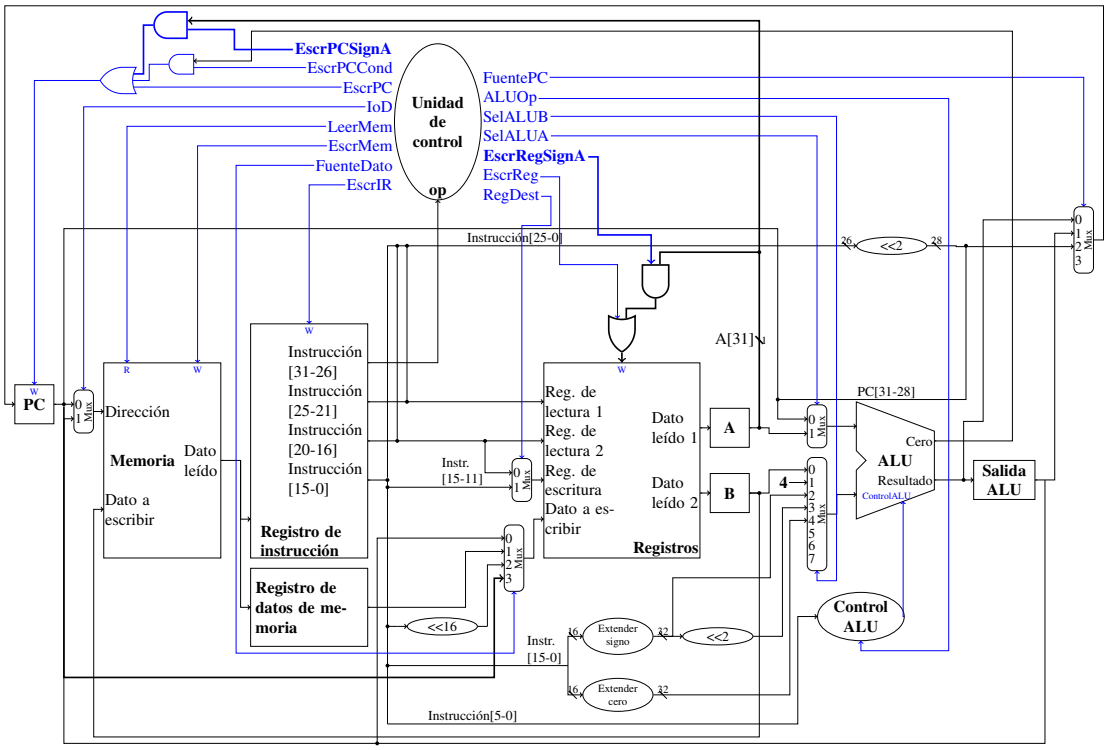


Figura 3.23: Visión completa de la implementación del procesador con las modificaciones requeridas por la instrucción `bltzal`.

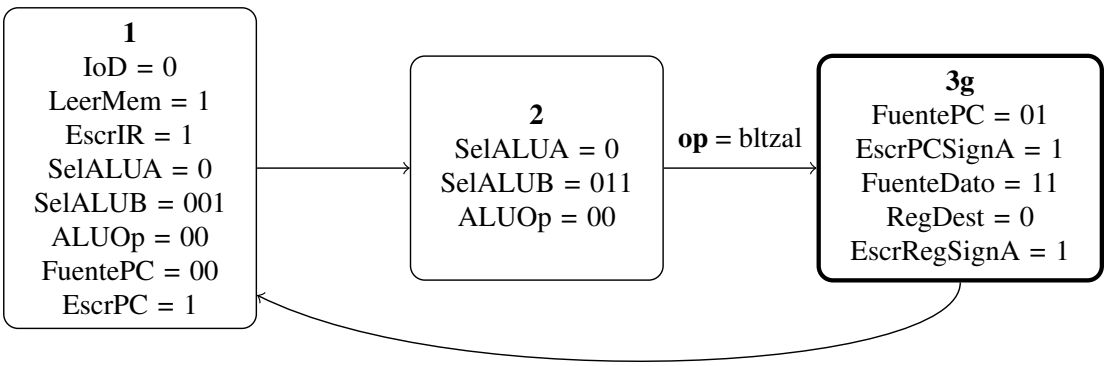


Figura 3.24: Modificaciones al autómata de control para incluir la instrucción `bltzal`. Por brevedad, se muestran solo los estados involucrados en la ejecución de la instrucción.

```
1         .data
2  array: .word 8, 7, 6, 5, 4, 3
3
4         .text
5   lab0: la    $t0, array
6   lab1: lw    $t2, 0($t0)
7         sll   $t2, $t2, 8
8         sw    $t0, 4($t2)
9         beq   $t2, $a0, lab2
10        addi  $t0, $t0, 4
11        j     lab1
12  lab2: ...
```

Figura 3.25: Código para el ejercicio 2.

3.10 EJERCICIOS: CÁLCULO DEL TIEMPO DE CPU DE PROGRAMAS SENCILLOS EN LA IMPLEMENTACIÓN MULTICICLO

Para la realización de algunos de los ejercicios de esta sección es aconsejable haber leído el apéndice sobre programación en ensamblador.

1. (*) Para el procesador multiciclo con una frecuencia de 400 MHz, calcule el tiempo de ejecución en segundos de un programa con la siguiente composición de intrucciones:

Tipo de instrucción	Cantidad de instrucciones
Artimético lógicas	3 500 000
Lecturas de memoria	2 800 000
Escrituras en memoria	2 000 000
Saltos condicionales	1 400 000
Saltos incondicionales	600 000
Cargas de datos	100 000

2. (*) Calcule el tiempo de ejecución en microsegundos del código de la figura 3.25 desde la etiqueta `lab0` a la `lab2`, si se ejecuta en una versión del procesador multiciclo que funciona a una frecuencia de 50 MHz. El valor inicial de `$a0` es 1536. Suponga que la pseudoinstrucción `la` se ensambla como dos instrucciones: una instrucción `lui` seguida de una instrucción `ori`.

3. (*) Dado el trozo de código MIPS de la figura 3.26, en el que se sabe que la etiqueta `array` se corresponde con la dirección `0x10010000`, y la etiqueta `main` con la dirección `0x00400000`:

   a) Detalle exactamente qué es lo que hace la rutina `main`, así como el contenido final de los registros `$s0`, `$t0`, `$t1` y `$t2` al finalizar la ejecución de la misma.

```
1         .data
2  array: .word 11,13,14,15,17
3         .word 19,21,22,24,27
4         .text
5   main: li    $s0, 0x12345678
6         la    $t0, array
7         li    $t1, 10
8         sll   $t1, $t1, 2
9         add   $t1, $t1, $t0
10        move  $t2, $t0
11   buc: lw    $t3, 0($t2)
12        andi  $t3, $t3, 1
13        beq   $t3, $0, zer
14        addi  $s0, $s0, 1
15   zer: addi  $t2, $t2, 4
16        beq   $t2, $t1, fin
17        j     buc
18   fin: jr    $ra
```

Figura 3.26: Código para el ejercicio 3.

b) Calcule cuánto tiempo tardará en ejecutarse todo el trozo de código mostrado (desde su comienzo en la línea 5 hasta que se ejecuta la instrucción en la línea 18) en un procesador multiciclo como el visto en clase, suponiendo que las latencias de sus unidades funcionales principales son:

- Unidad de memoria: 15 ns.
- ALU: 10 ns.
- Banco de registros (lectura o escritura): 12 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 del tipo cargas inmediatas, aritmético-lógicas o de desplazamiento, 3 si son de salto condicional o incondicional, 4 si son de almacenamiento en memoria, y 5 si son de carga de memoria. Además, tenga también en cuenta que algunas instrucciones de ensamblador pueden ser realmente pseudoinstrucciones que se implementen con una o más instrucciones reales distintas. En concreto, la primera instrucción (`li`) es en realidad una pseudoinstrucción que se traduce con dos instrucciones de código máquina (un `lui` y un `ori`), la instrucción `la` es una pseudoinstrucción que se traduce a un sólo `lui`, y el resto de pseudoinstrucciones `li` (línea 7) y `move` (línea 10) se traducen, respectivamente, a sendas instrucciones aritmético-lógicas `addi` y `add`.

c) Sabiendo que los códigos de operación de las instrucciones `lui`, `beq` y `j` son, respectivamente, 15, 4 y 2 (en decimal), especifique el código máquina (en hexadecimal, en la forma 0xYYYYYYYY) de las instrucciones de las líneas 6, 13 y 17, justificando en cada caso la respuesta. Tenga también en cuenta para la codificación que `$t0`=`$8`, `$t1`=`$9`, `$t2`=`$10`, y así sucesivamente.

```
1         li    $8, 0x10010000
2         li    $9, 0x1001FFF0
3         move  $10, $0
4   loop: sll   $11, $10, 2
5         add   $11, $8, $11
6         lw    $12, 0($11)
7         lw    $13, 4($11)
8         sw    $12, 0($9)
9         beq   $12, $13, fin
10        addi  $10, $10, 1
11        addi  $9, $9, -4
12        j           loop
13  fin:  ...
```

Dirección	Contenido
0x10010000	0x00000002
0x10010004	0x00000004
0x10010008	0x00000006
0x1001000C	0x00000008
0x10010010	0x0000000A
0x10010014	0x0000000C
0x10010018	0x0000000C
0x1001001C	0x00000002
0x10010020	0x00000001
0x10010024	0x00000000
...resto...	0x00000000

Figura 3.27: Código y datos para el ejercicio 4

4. Dado el fragmento de código MIPS mostrado a la izquierda de la figura 3.27 y suponiendo que la memoria contiene los valores mostrados a la derecha en las direcciones desde `0x10010000` hasta `0x10020000`:

   a) Explique brevemente qué hace el programa.

   b) Calcule el tiempo que tardará en ejecutarse en el procesador multiciclo suponiendo que las latencias de sus unidades funcionales principales son:

   - Unidad de memoria: 20 ns.
   - ALU: 5 ns.
   - Banco de registros (lectura o escritura): 10 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. En concreto, las dos primeras instrucciones (`li`) son en realidad pseudoinstrucciones que se traducen cada una por otras dos[13]: un `lui` y un `ori` y la instrucción `move` también es una pseudoinstrucción que se traduce por un `addi` o un `ori`.

5. El programa de la figura 3.28 se ejecuta en el procesador multiciclo visto en clase. Se sabe que tarda 13.2 microsegundos en ejecutarse desde la etiqueta `inicio` a la `fin` (sin incluir esta última). Calcule la frecuencia del procesador utilizado. Suponga que la pseudoinstrucción `la` se ensambla como dos instrucciones: una instrucción `lui` seguida de una instrucción `ori`.

3.11 SOLUCIÓN A EJERCICIOS SELECCIONADOS DE CÁLCULO DEL TIEMPO DE CPU DE PROGRAMAS SENCILLOS

[13] El primer `li` se podría traducir utilizando sólo la instrucción `lui`, ya que los 16 bits inferiores son 0.

```
1          .data
2  array:  .word 0x1004, 0x2004, 0
3
4          .text
5  inicio: la    $t0, array
6          li    $t1, 0x11
7          sll   $t1, $t1, 8
8  bucle:  lw    $t2, 0($t0)
9          beqz  $t2, fin
10         add   $t3, $t2, $t1
11         sw    $0,  0($t3)
12         addi  $t0, $t0, 4
13         j     bucle
14 fin:    ...
```

Figura 3.28: Código para el ejercicio 5.

1. El tiempo de ejecución de un programa concreto en un procesador concreto viene dado por la expresión:

$$T = NITCCPI$$

El número de instrucciones (NI) será de $3\,500\,000 + 2\,800\,000 + 2\,000\,000 + 1\,400\,000 + 700\,000 = 10\,400\,000$.

El tiempo de ciclo $TC = \frac{1}{400\text{ MHz}} = 0.0025$ µs. El CPI dependerá de la mezcla de instrucciones, teniendo en cuenta la duración en ciclos de cada una:

Tipo de instrucción	**Cantidad**	**Fracción sobre el total**	**Duración**
Artimético lógicas	3 500 000	$\frac{3\,500\,000}{10\,400\,000} = 0.3365$	4
Lecturas de memoria	2 800 000	$\frac{2\,800\,000}{10\,400\,000} = 0.2692$	5
Escrituras en memoria	2 000 000	$\frac{2\,000\,000}{10\,400\,000} = 0.1923$	4
Saltos condicionales	1 400 000	$\frac{1\,400\,000}{10\,400\,000} = 0.1346$	3
Saltos incondicionales	600 000	$\frac{600\,000}{10\,400\,000} = 0.0577$	3
Cargas de datos	100 000	$\frac{100\,000}{10\,400\,000} = 0.0096$	3

Con lo que el CPI es:

$$CPI_{\text{multi}} = 0.33654 + 0.26925 + 0.19234 + 0.13463 + 0.05773 + 0.00963 = 4.0669$$

Por tanto, el tiempo de ejecución es:

$$T_{\text{multi}} = 10\,400\,000 0.0025\text{ µs} 4.0669 = 0.1057\text{ s}$$

2. El programa lee las palabras de la memoria a partir de la dirección `array` y las multiplica por 256 (mediante un desplazamiento a la izquierda en la línea 8). Después de realizar la multiplicación, realiza un almacenamiento en la dirección resultante. El proceso termina cuando la dirección resultante es igual a `$a0`.

   Con los datos de entrada del problema, se leerán los tres primeros elementos del array (ya que 6256 = 1536), a partir de lo cual es fácil deducir cuántas veces se ejecutará cada instrucción.

   Por tanto, podemos calcular el tiempo de ejecución del programa a partir del número de ciclos que tarda en ejecutarse cada instrucción (ten en cuenta que la pseudoinstrucción `la` se sustituye por `lui` y `ori`):

Línea	Instrucción		Ejecuciones	Ciclos por ejecución	Ciclos totales
5	`lui`	`$t0, array`	1	3	3
	`ori`	`$t0, array`	1	4	4
6	`lw`	`$t2, 0($t0)`	3	5	15
7	`sll`	`$t2, $t2, 8`	3	4	12
8	`sw`	`$t0, 4($t2)`	3	4	12
9	`beq`	`$t2, $a0, lab2`	3	3	9
10	`addi`	`$t0, $t0, 4`	2	4	8
11	`j`	`lab1`	2	3	6
					Total: **69**

   Por tanto, el programa requeriría un total de 69 ciclos para ejecutarse. Como la frecuencia es de 50 MHz, el tiempo de ciclo es $TC = \frac{1}{50\ \text{MHz}} = 0.02\ \mu\text{s}$. Por tanto, el programa tardará en ejecutarse 690.02 µs = 1.38 µs.

3. a) El código de la subrutina `main` tiene una inicialización que va desde la línea 5 a la 10 (ambas inclusive) y a continuación un bucle (líneas 11 a 17) del cual se sale cuando la condición que se comprueba en la línea 16 se cumple (es decir, cuando `$t2`, que inicialmente vale 0x10010000 —apuntando al comienzo del array— y se va incrementando en cada paso en 4 unidades, llega a valer lo mismo que `$t1`, que vale 0x10010000 + 4 × 10 = 0x10010028, y que marca por tanto el final del array). Dado que en cada paso `$t2` se va incrementando de 4 en 4, en principio todas las instrucciones del bucle se ejecutan exactamente 10 veces, excepto la instrucción en la línea 14, que se ejecuta sólo para los valores impares del array (dado que éstos son los que tienen activo su último bit, provocando que no se cumpla la condición del salto de la línea 13), así como la instrucción `j` de la línea 17, que se ejecuta sólo 9 veces. Dado que 7 son los números impares del array, la instrucción de la línea 14 se ejecuta sólo 7 veces.

      Así pues, la rutina completa va recorriendo el array de enteros de 4 bytes, desde su comienzo en la dirección `0x10010000` hasta su final en la dirección `0x10010024` (la `0x10010028` sería la primera posición libre a continuación). Y para cada valor leído, si este es impar, incrementa el contador que hay en `$s0`. Puesto que el valor inicial de `$s0` es `0x12345678`, al terminar la rutina los valores de `$t0`, `$t1`, `$t2` y `$s0` serán, respectivamente, `0x10010000`, `0x10010028`, `0x10010028` y `0x1234567f`.

b) En cuanto al tiempo de ejecución del programa, éste vendrá dado por la expresión:

$$T_{\text{ejecución}} = N_{\text{ciclos}} T_{\text{ciclo}}$$

El tiempo de ciclo (T_{ciclo}) está determinado por la unidad funcional más lenta, que en este caso es la memoria que tarda 15 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. Teniendo en cuenta todo lo comentado en el apartado a), nos quedará la siguiente tabla de ejecución para cada instrucción:

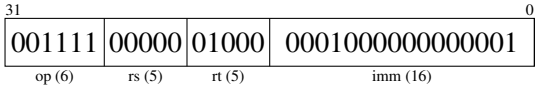
Dirección	Instrucción			Ejecuciones	Ciclos por ejecución	Ciclos totales
0x00400000	`main:`	`lui`	`$s0,0x1234`	1	3	3
0x00400004		`ori`	`$s0,$s0,0x5678`	1	4	4
0x00400008		`lui`	`$t0,0x1001`	1	3	3
0x0040000c		`addi`	`$t1,$0,10`	1	4	4
0x00400010		`sll`	`$t1,$t1,2`	1	4	4
0x00400014		`add`	`$t1,$t1,$t0`	1	4	4
0x00400018		`add`	`$t2,$t0,$0`	1	4	4
0x0040001c	`buc:`	`lw`	`$t3,0($t2)`	10	5	50
0x00400020		`andi`	`$t3,$t3,1`	10	4	40
0x00400024		`beq`	`$t3,$0,zer`	10	3	30
0x00400028		`addi`	`$s0,$s0,1`	7	4	28
0x0040002c	`zer:`	`addi`	`$t2,$t2,4`	10	4	40
0x00400030		`beq`	`$t2,$t1,fin`	10	3	30
0x00400034		`j`	`buc`	9	3	27
0x00400038	`fin:`	`jr`	`$ra`	1	3	3
					Total:	**274**

Por tanto:

$$T_{\text{ejecución}} = N_{\text{ciclos}} T_{\text{ciclo}} = 27615\,\text{ns} = 4110\,\text{ns} = 4.11\,\mu\text{s}$$

c)
- Instrucción `la`:
  Puesto que la etiqueta array se corresponde a la dirección `0x10010000`, y el código de operación de la instrucción `lui` es $15 = 001111)_2$, tendremos:

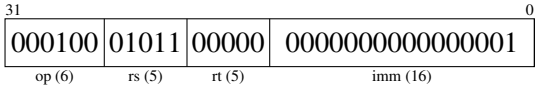
  `la $t0, array` ⇒ `lui $8, 0x1001`

  = `0x3c081001`

- Instrucción `beq`:
  Puesto que el destino del salto está justo una instrucción por delante de la instrucción siguiente al `beq`, y el código de operación de `beq` es $4 = 000100)_2$, tendremos:

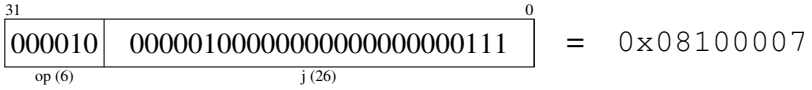
  `beq $t3,$0,zer` ⇒ `beq $11,$0,0x0001`

  = `0x11600001`

- Instrucción `j`:
  Puesto que el destino del salto está en `0x0040001c`, y por otro lado `0x0040001c` descartando los dos bits menos significativos y expresado en binario en 26 bits, es exactamente $00000100000000000000000111)_2$, y además el código de operación de la instrucción `j` es $2 = 000010)_2$, tendremos:

  `j buc` $\Rightarrow$ `j 0x0040001c`



$= \texttt{0x08100007}$

3.12 EJERCICIOS: INTRODUCCIÓN DE NUEVAS INSTRUCCIONES EN EL CAMINO DE DATOS MULTICICLO

Utilizando la metodología descrita para la inclusión de nuevas instrucciones en el esquema de implementación multiciclo, realice las fases de análisis y diseño de las siguientes instrucciones[14]:

1. (*) **`xchg $x, ($y):`** Intercambia el contenido de la palabra de memoria cuya dirección está en el registro `$y` y el contenido del registro `$x`.

2. (*) **`bnezm ($x), label:`** Salta a `label` si el contenido de la dirección de memoria almacenada en `$x` no es cero.

3. (*) **`maxm $x, $y, ($z):`** Almacena en el registro `$x` el máximo entre el valor almacenado en el registro `$y` y el contenido en la posición de memoria cuya dirección se encuentra en el registro `$z`.

4. (*) **`abs $x,$y:`** Calcula el valor absoluto del registro `$y`, y lo almacena en `$x`.

5. (*) **`jalr $a:`** Salta a la dirección contenida en el registro `$a` y guarda la dirección de retorno (dirección de la instrucción siguiente al `jalr`) en el registro `$ra` (`$31`).

6. **`bltm $a, ($b), label:`** Salta a la dirección de `label` si el valor contenido en el registro `$a` es menor que el valor contenido en la dirección de memoria contenida en el registro `$b`.

7. **`add3 $a, ($b), Inm:`** Realiza la suma del contenido del registro `$a`, la palabra de memoria cuya dirección se encuentra en `$b` y el inmediato `Inm`. El resultado de la suma se guarda en `$a`.

8. **`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`.

9. **`beqm $a, ($b), label`**: Salta a la dirección `label` si el valor contenido en el registro `$a` igual que el valor contenido en la dirección de memoria contenida en el registro `$b`.

[14] `$a`, `$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.

10. **`lwadd $a, ($b), $c`:** Lee el contenido de la dirección de memoria almacenada en el registro `$b` y le suma el contenido del registro `$c`. El resultado de la suma se almacena en el registro `$a`. Es decir, la instrucción se comporta como `add`, con la salvedad de que el primer operando está almacenado en memoria.

3.13 SOLUCIÓN A EJERCICIOS SELECCIONADOS DE INTRODUCCIÓN DE NUEVAS INSTRUCCIONES

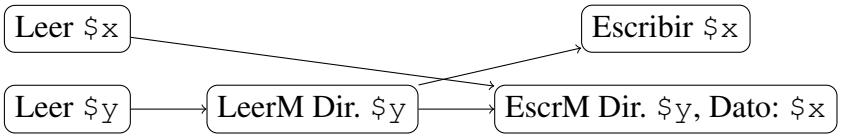
1. La aplicación de la metodología para la inclusión de nuevas instrucciones aplicada al caso de la instrucción `xchg $x, ($y)` sería:

   - **Análisis**:

     a) La semántica de «`xchg $x, ($y)`» puede especificarse de la siguiente forma:
     `$x` ← `Memoria[$y]` , y a la misma vez, `Memoria[$y]` ← `$x`

     b) Para poder ejecutar la instrucción `xchg`, se tendrían que usar las siguientes unidades funcionales:
     - Banco de registros:
       - Leer `$x`, Leer `$y`, Escribir `$x`.
     - ALU:
       - –
     - Memoria:
       - LeerM Dir. `$y`, EscrM Dir. `$y`, Dato: `$x`.

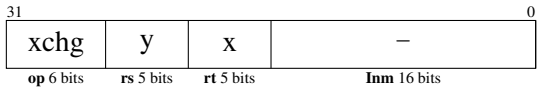
     c) El orden de dichas tareas y las dependencias serían las siguientes:

   - **Diseño**:

     a) Podríamos usar tanto un formato R como I para la instrucción. En este caso usaremos el formato I:

     `xchg $x, ($y)`

     b) El número de ciclos mínimos para ejecutar la instrucción es de 4. Las lecturas de `$x` y `$y` se hacen en el segundo ciclo. En el tercer ciclo se leerá el contenido de la dirección de memoria almacenada en `$y`. Por último, en el cuarto ciclo, el valor leído de memoria (y que se encuentra en el registro intermedio `$RDM`) se escribirá en el banco de registro, concretamente en el registro `$x`, y a la vez, el valor del registro `$x`, que está almacenado en el registro intermedio `B`, se escribirá en la dirección de memoria que hay en `$y` (almacenada en el registro intermedio `$A`). Los ciclos, y lo que se haría durante cada uno de ellos, serían los siguientes:

Ciclo 1	**Ciclo 2**	**Ciclo 3**	**Ciclo 4**
IR ← Memoria[PC]	A ← Reg[**rs**]	RDM ← `Memoria[A]`	`Memoria[A]` ← `B`
PC ← PC + 4	B ← Reg[**rt**]		Reg[**rt**] ← RDM

c) El único cambio que se requeriría en el camino de datos para implementar la instrucción sería la ampliación del multiplexor que controla la señal `IoD` para poder leer de memoria el contenido del registro intermedio `A`. Para esto haremos que el multiplexor que controla la señal `IoD` pase a tener 4 entradas y conectaremos a la entrada 2 la salida (32 bits) del registro intermedio `A`.

d) Habrá que añadir dos nuevos estados al autómata de control original. Además, en los estados 1, 4b y 4c habrá que actualizar la señal de control `IoD`, que como se ha explicado es ahora de 2 bits (se muestra el cambio en el estado 1). El resultado se muestra en la figura 3.29.

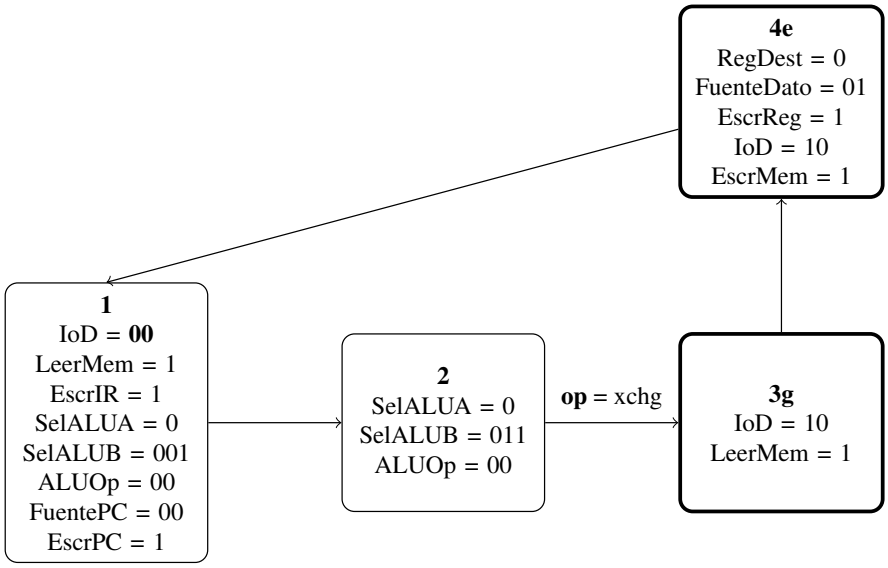


Figura 3.29: Modificaciones en el autómata de control para incluir la instrucción `xchg`. Los estados añadidos han sido resaltados.

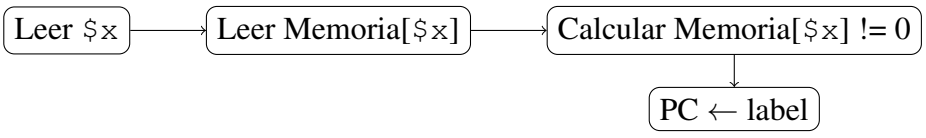
2. La aplicación de la metodología para la inclusión de nuevas instrucciones aplicada al caso de la instrucción `bnezm ($x), label` sería:

   - **Análisis**:

     a) La semántica de «`bnezm ($x), label`» es:
     `Memoria[$x] != 0: PC ← label`

     b) Para poder ejecutar la instrucción `bnezm`, se tendrían que usar las siguientes unidades funcionales:
     - Banco de registros:
       - Leer `$x`.
     - ALU:
       - Calcular `Memoria[$x]` != 0.
     - Memoria:
       - Leer `Memoria[$x]`.

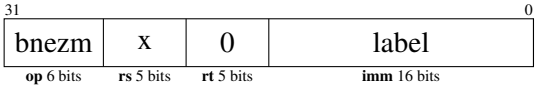
     c) El orden de dichas tareas y las dependencias serían las siguientes:



- **Diseño**:

  a) El formato más adecuado para la instrucción sería el I, ya necesitamos codificar la etiqueta y al menos un registro (`$x`). Además como en el formato tenemos un campo sin utilizar para especificar un registro, podemos hacer uso del registro `$zero` que almacena siempre el valor 0 para calcular Memoria[`$x`] != 0.

  `bnezm ($x), label`



  b) El número de ciclos mínimos para ejecutar la instrucción es 4. Las lecturas de `$x` y `$y` se hacen en el segundo ciclo. Además se calcula la dirección destino del salto. Esto en realidad lo hacen todas las instrucciones. En el tercer ciclo se accederá a memoria a la dirección de `$x`. Además, repetimos todas las operaciones realizadas en el ciclo 2, ya que en el cuarto ciclo necesitaremos tanto el valor de B (cero) como el cálculo de la dirección destino. En el cuarto ciclo se calculará `Memoria[$x] != 0`. Para ello tenemos varias opciones. Por ejemplo, podemos restar `$x`, guardado en RDM, y `$zero`, guardado en B. Si el resultado es distinto de cero, esto es, si la salida `cero` de la ALU no se activa, hay que actualizar el valor del PC con la dirección destino del salto calculada en el ciclo 3.
  Los ciclos, y lo que se haría durante cada uno de ellos, serían los siguientes:

Ciclo 1	**Ciclo 2**	**Ciclo 3**	**Ciclo 4**
IR ← Memoria[PC]	A ← Reg[**rs**]	RDM ← Memoria[A]	$\overline{(RDM-B)_{cero}}$: PC ← SalidaALU
PC ← PC + 4	B ← Reg[**rt**]	B ← Reg[**rt**]	
	SalidaALU ← PC + (imm << 2)	SalidaALU ← PC + (imm << 2)	

  c) El camino de datos hay que modificarlo para los ciclos 3 y 4.
  En el ciclo 3, necesitamos que la dirección que se lee de memoria venga del registro A (actualmente solo tiene la entrada de SalidaALU y PC). Por tanto, el multiplexor deberá ser de 3 entradas, la tercera viniendo del registro A.
  En el ciclo 4, en primer lugar, tenemos que calcular la resta de RDM y B. Para ello tenemos que llevar RDM a la entrada A de la ALU y, por tanto, extender el multiplexor SelALUA. En segundo lugar, el PC se debería modificar dependiendo de la salida cero de la ALU. Aunque ya tenemos una puerta AND con la entrada de la salida cero (necesaria para `beq`), ahora por el contrario necesitamos que PC se actualice si la salida no está activa. Haría falta otra puerta AND conectada a la salida cero negada y a una señal de control llamada por ejemplo `EscrPCCondNoCero`. La salida de esta puerta se conectaría a la puerta OR que controla la escritura del PC.

  d) Habrá que añadir dos nuevos estados al autómata de control original. Además, en los estados 1, 4b y 4c habrá que actualizar la señal de control `IoD`, que como se ha explicado es ahora de 2 bits, y en los estados 1, 2, 3a, 3b, 3c y 3f habrá que hacer lo propio con la señal `SelALUA`

(se muestra los cambios en los estados 1 y 2). El resultado se muestra en la figura 3.30.

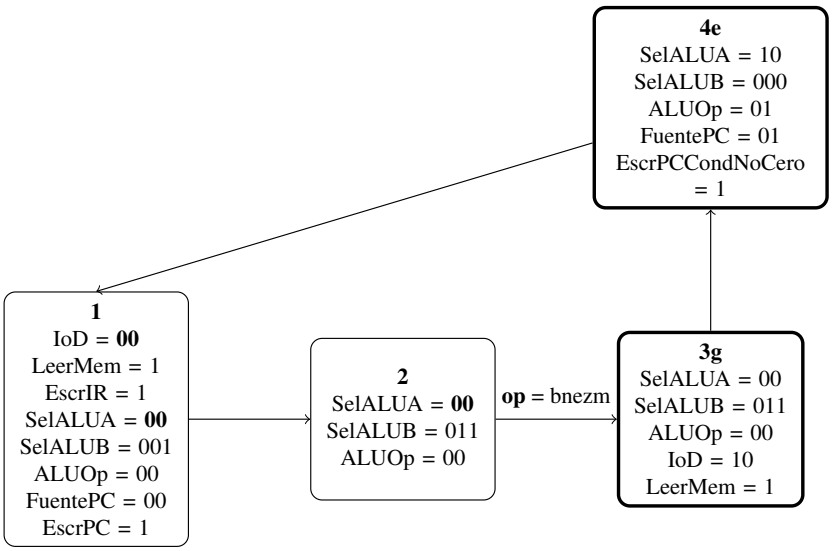


Figura 3.30: Modificaciones en el autómata de control para incluir la instrucción `bnezm`. Los estados añadidos han sido resaltados.

3. La aplicación de la metodología para la inclusión de nuevas instrucciones aplicada al caso de la instrucción `maxm $x, $y, ($z)` sería:

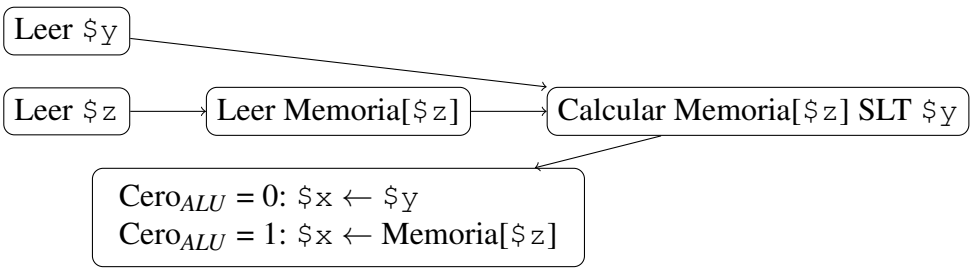
   - **Análisis**:

     a) La semántica de «`maxm $x, $y, ($z)`» puede especificarse de la siguiente forma:
     `$y` $\leq$ `Memoria[$z] ? $x` $\leftarrow$ `Memoria[$z] : $x` $\leftarrow$ `$y`

     b) Para poder ejecutar la instrucción `maxm`, se tendrían que usar las siguientes unidades funcionales:

     - Banco de registros:
       - Leer `$y`, Leer `$z`, Escribir `$x`.
     - ALU:
       - Calcular `Memoria[$z] SLT $y`.
     - Memoria:
       - Leer `Memoria[$z]`.

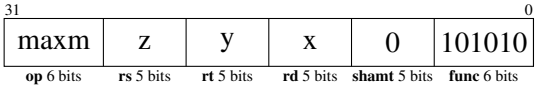
     c) El orden de dichas tareas y las dependencias serían las siguientes:



   - **Diseño**:

a) El formato más adecuado para la instrucción sería el R, ya que necesitamos codificar tres registros (`$x`, `$y` y `$z`). Dado que el registro `$x` es a escribir, lo pondremos en el campo `rd`. En `rs` pondremos `$z` y en `rt`, `$y`. Aprovechamos también el campo de función para introducir el código correspondiente a `SLT`.

`maxm $x, $y, ($z)`



b) El número de ciclos mínimos para ejecutar la instrucción es 5. Las lecturas de `$y` y `$z` se hacen en el segundo ciclo, y se debe repetir la de `$y` en el tercer y cuarto ciclos, ya que su valor se utiliza también durante los ciclos 4 y 5. En el tercer ciclo se accederá a memoria a la dirección de `$z`. En el cuarto ciclo se calculará `Memoria[$z] SLT $y`, y en el último ciclo escribiremos en `$x` el valor más grande en función de la salida `Cero` de la ALU. Observa que para asegurar que no cambia el valor de la salida `Cero` de la ALU, tendremos que mantener los valores de las señales de control de la misma durante el ciclo 5. De igual forma, para no perder el valor leído de memoria, tendremos que repetir la lectura en el ciclo 4.
Los ciclos, y lo que se haría durante cada uno de ellos, serían los siguientes:

Ciclo 1	**Ciclo 2**	**Ciclo 3**	**Ciclo 4**	**Ciclo 5**
IR ← Memoria[PC] PC ← PC + 4	A ← Reg[**rs**] B ← Reg[**rt**] SalidaALU ← PC + (imm << 2)	RDM ← Memoria[A] B ← Reg[**rt**]	RDM ← Memoria[A] RDM SLT B B ← Reg[**rt**]	RDM SLT B Cero = 0 : Reg[**rd**] ← B Cero = 1 : Reg[**rd**] ← RDM

c) El camino de datos tendremos que realizar las siguientes modificaciones:

- Para poder hacer `RDM ← Memoria[A]` en el ciclo 3 tendremos que ampliar el multiplexor que controla la señal de control `IoD`, de forma que podamos leer la dirección de memoria almacenada en el registro intermedio `A` (`IoD=10`).
- Para poder hacer `RDM SLT B` en el ciclo 4 tendremos que ampliar el multiplexor que controla la señal de control `SelALUA`, de forma que podamos operar a través de la primera entrada de la ALU con el valor almacenado en el registro intermedio `RDM` (`SelALUA=10`).
- Para poder elegir el valor a escribir en el banco de registros en el último paso, añadimos un multiplexor 2 a 1 controlado por la salida `Cero` de la ALU, de forma que en su entrada 0 conectamos la salida del registro intermedio `B`, y en su entrada 1 la del registro intermedio `RDM`. Usamos el multiplexor controlado por la señal `FuenteDato`, de forma que a su entrada 3 hasta ahora no utilizada (`FuenteDato=11`) conectamos la salida del multiplexor 2 a 1 añadido.

d) Habrá que añadir tres nuevos estados al autómata de control original. Además, en los estados 1, 4b y 4c habrá que actualizar la señal de control `IoD`, que como se ha explicado es ahora de 2 bits, y en los estados

1, 2, 3a, 3b, 3c y 3f habrá que hacer lo propio con la señal `SelALUA` (se muestra los cambios en los estados 1 y 2). El resultado se muestra en la figura 3.31.

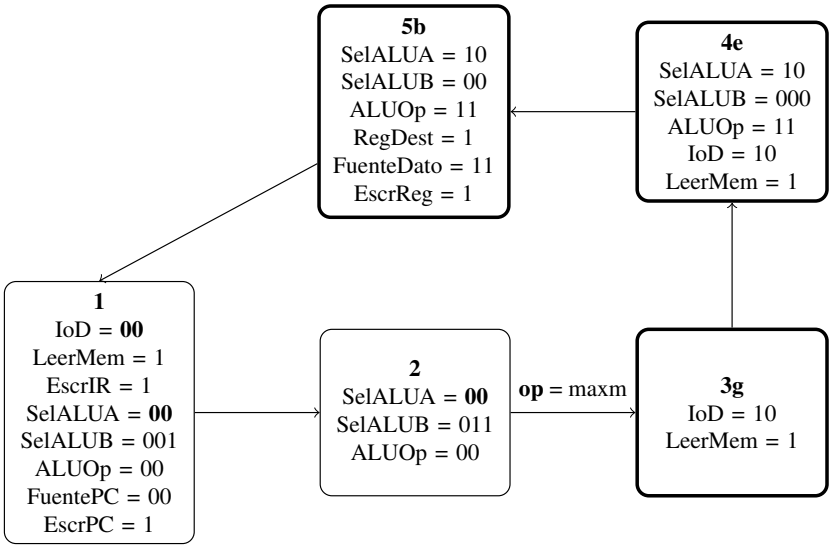


Figura 3.31: Modificaciones en el autómata de control para incluir la instrucción `maxm`. Los estados añadidos han sido resaltados.

4. La aplicación de la metodología para la inclusión de nuevas instrucciones aplicada al caso de la instrucción `abs $x,$y` sería:

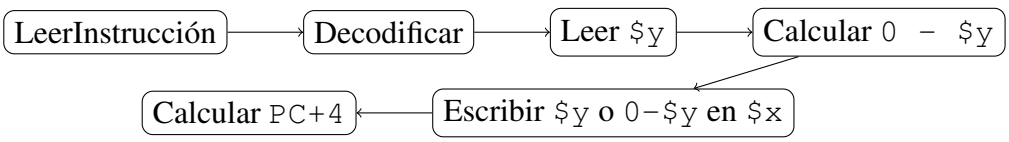
   - **Análisis**:

     a) La semántica de «`abs $x, $y`» es:
     `$y >= 0 : $x ← $y`
     `$y < 0 : $x ← 0 - $y`

     b) Para poder ejecutar la instrucción `abs`, se tendrían que usar las siguientes unidades funcionales:

     - Banco de registros:
       - Leer `$y`.
       - Escribir `$x`.
     - ALU:
       - Calcular `0 - $y`.
       - Calcular `PC + 4`.
     - Memoria:
       - LeerInstrucción.

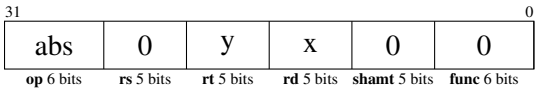
     Además, el orden de dichas tareas sería el siguiente:



   - **Diseño**:

a) El formato más adecuado para la instrucción sería el R, ya que además de los registros `$x` y `$y`, vamos a hacer uso del registro `$zero` que almacena el valor 0, y lo usaremos para calcular 0 - `$y`

`abs $x, $y` →



b) El número de ciclos mínimos para ejecutar la instrucción son 4. La lectura de `$y` se hace en el segundo ciclo, en el tercer ciclo se calculará `0-$y`, y en el cuarto se modifica el registro `$x` dependiendo del valor inicial del registro `$y`. Si el bit de signo (bit 31) del registro `$y` es 0, se guardará `$y` en `$x`. Sin embargo, si el bit de signo de `$y` es 1 (indicando que `$y` es negativo), en `$x` se guardará 0 - `$y`.
Los ciclos serían los siguientes:

Ciclo 1	**Ciclo 2**	**Ciclo 3**	**Ciclo 4**
IR ← Memoria[PC]	A ← Reg[**rs**]	SalidaALU ← A − B	B[31] = 0 : Reg[**rd**] ← B
PC ← PC + 4	B ← Reg[**rt**]		B[31] = 1 : Reg[**rd**] ← SalidaALU

c) El camino de datos sólo se tendría que modificar para el ciclo 4, ya que en el banco de registros habrá que escribir **B** o **SalidaALU** dependiendo del bit de signo. Para que el ciclo 4 se ejecute correctamente hay que añadir un nuevo multiplexor para seleccionar entre el registro `B` o `SalidaALU`. La línea de control de este nuevo multiplexor será `B[31]`. La primera entrada del multiplexor será `B` (elegida cuando `B[31]=0`), y la segunda será `SalidaALU` (elegida cuando `B[31]=1`).
Además conectaremos a la entrada 3 (hasta ahora no usada) del multiplexor controlado por `FuenteDato` la salida del nuevo multiplexor.

d) Habrá que añadir dos nuevos estados al autómata de control original. El resultado se muestra en la figura 3.32.

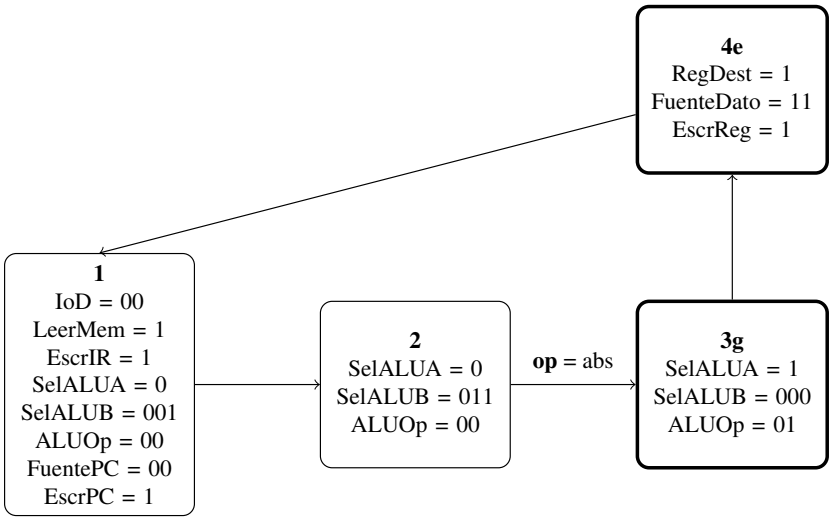


Figura 3.32: Modificaciones en el autómata de control para incluir la instrucción `abs`. Los estados añadidos han sido resaltados.

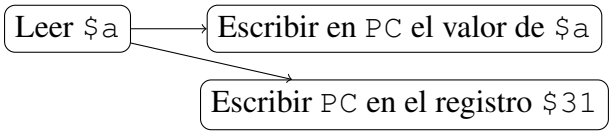
5. La aplicación de la metodología para la inclusión de nuevas instrucciones aplicada al caso de la instrucción `jalr $a` sería:

   - **Análisis**:

     a) La semántica de «`jalr $a`» es:
     `$31 ← PC + 4`
     `PC ← $a`

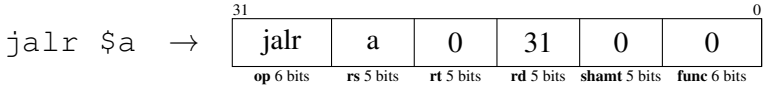
     b) Para poder ejecutar la instrucción `jalr`, se tendrán que usar las siguientes unidades funcionales:
     - Banco de registros:
       - Leer `$a`.
       - Escribir `$31`.
     - ALU: No hace nada (el cálculo de `PC + 4` ya se realiza).
     - Memoria: No hace nada.

     c) Respecto a la dependencia de tareas, la modificación de registro `PC` se hace en el paso 3, después de haber leído `$a`, y en el mismo paso el contenido que hubiese en `PC` se escribirá en el registro `$31`.



   - **Diseño**:

     a) Respecto a su codificación, esta instrucción necesita un operando en registro (`$a`). Además, también utiliza el registro `$31` (`$ra`) implícitamente, y podemos aprovechar algún campo de la instrucción codificada para indexarlo. Por tanto, dado que solo se necesitan dos registros se podría utilizar tanto el forma I como el R. Elegiremos el formato R que es el que utiliza MIPS para codificar esta instrucción en la realidad.



     b) El número de ciclos mínimos para ejecutar la instrucción son 3. Tras haber leído en el paso 2 el registro `$a`, en el paso 3 ya se puede modificar el PC y a la vez el valor de PC (al que se le ha sumado 4 ya en el primer ciclo) se puede escribir en `$31`.

Ciclo 1	**Ciclo 2**	**Ciclo 3**
IR ← Memoria[PC]	A ← Reg[**rs**]	PC ← A
PC ← PC + 4		Reg[**rd**] ← PC

     c) Para realizar algunos ciclos indicados en la sección anterior, anterior necesitamos modificar el camino de datos ya que no podemos realizar las acciones del ciclo 3. Para hacerlo posible el camino de datos lo tendremos que extender de la siguiente manera:

     Para `PC ← A`: Añadir una nueva entrada al multiplexor controlado por la señal de control FuentePC que esté conectada al registro A (en el que tenemos el contenido de `$a`). Dicho multiplexor tenía anteriormente 3 entradas conectadas, por lo que podemos añadir una

más sin aumentar el tamaño de la señal de control FuentePC. La combinación «11» seleccionará la nueva entrada.

Para `Reg[rd]` ← `PC`: Añadir una nueva entrada al multiplexor controlado por la señal de control FuenteDato que esté conectada al contador de programa (PC). Al igual que antes, dicho multiplexor tenía anteriormente 3 entradas conectadas, por lo que podemos añadir una más sin aumentar el tamaño de la señal de control FuenteDato.

d) Respecto al camino de control, habrá que añadir un nuevo estado al autómata de control original. El resultado se muestra en la figura 3.33.

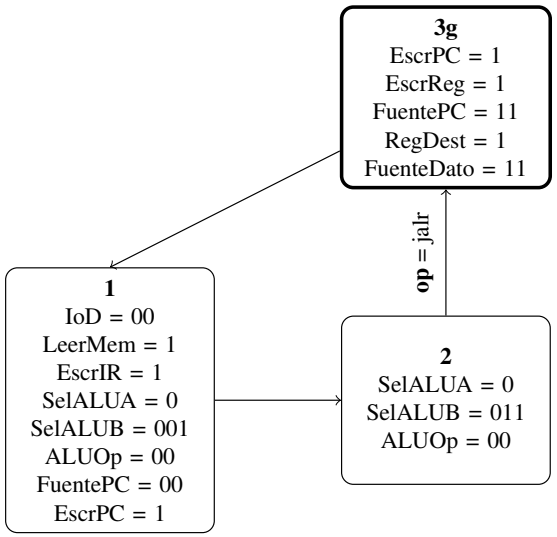


Figura 3.33: Modificaciones en el autómata de control para incluir la instrucción `jalr`. Los estados añadidos han sido resaltados.