MICROPROCESADOR RISC SINTETIZABLE EN FPGA PARA FINES DOCENTES

J.D. MUÑOZ[1], S. ALEXANDRES[1] Y C. RODRÍGUEZ-MORCILLO[2]
[1]*Departamento de Electrónica y Automática. Escuela Técnica Superior de Ingeniería ICAI. Universidad Pontificia Comillas. España.*
[2]*Instituto de Investigación Tecnológica. Escuela Técnica Superior de Ingeniería ICAI. Universidad Pontificia Comillas. España.*

En esta comunicación se presenta una arquitectura RISC de 16 bits lo suficientemente simple como para poder ser abordada en un curso introductorio de diseño de sistemas digitales, pero lo suficientemente compleja como para poder desarrollar aplicaciones reales con ella. Además es posible implantar esta arquitectura usando una FPGA de bajo coste, como por ejemplo una Flex 10k30 de Altera, lo que facilita su uso en el laboratorio por el alumno.

1. Introducción

Es frecuente encontrar en la literatura arquitecturas diseñadas con fines docentes que se han simplificado tanto que están muy lejos de los diseños actuales. Incluso existen aún diseños basados en acumulador, cuando este tipo de CPUs hace años que están en desuso. Por otro lado, las arquitecturas actuales de 32 bits como IA-32, PowerPC o MIPS son demasiado complejas para ser implantadas en un tiempo razonable en un laboratorio. Además, es deseable que una arquitectura de este tipo no requiera una FPGA con muchos recursos, dado el limitado presupuesto con el que suelen contar las universidades.

Por todo ello, en esta comunicación se presenta una arquitectura simple de 16 bits, pero muy similar al MIPS, por lo que se consigue por un lado el objetivo de tener una arquitectura que necesita pocos recursos *hardware* para implantarse en una FPGA de bajo coste, pero sin que esté alejada de las arquitecturas reales usadas por la industria en la actualidad.

La arquitectura aquí presentada está basada en el "Ridiculously Simple Computer" [1] desarrollado por el profesor Bruce Jacob. No obstante, esta arquitectura, denominada RiSC-16, aunque similar al MIPS [2], difiere un poco en su estructura, lo que incluso dificulta un poco la implantación del camino de datos. La aproximación llevada a cabo en este trabajo ha consistido en rediseñar el juego de instrucciones del MIPS para usar palabras de 16 bits, al igual que en RiSC-16, pero manteniendo la misma estructura usada por el MIPS. Esto permite usar el mismo camino de datos expuesto en [2] sin más que cambiar los anchos de los buses, por lo que el alumno en realidad está aprendiendo a usar una arquitectura real expuesta en uno de los libros de texto clásicos de la materia.

2. La arquitectura ICAI-RiSC-16

En esta sección se presenta la arquitectura diseñada. Tal como se ha expuesto en la introducción, esta arquitectura es una versión de 16 bits del MIPS, con algunos cambios orientados a facilitar su integración en lógica programable. Sus características son:

- Arquitectura Harvard. Para facilitar el diseño, el programa se almacena en una memoria interna de la FPGA configurada como ROM y los datos en otra memoria RAM interna.
- Bus de datos de 16 bits.
- Buses de direcciones de 16 bits.
- El acceso a memoria se realiza siempre en palabras de 16 bits, por lo que las direcciones representan direcciones de palabra, no de byte.
- 8 registros de datos.
- Al igual que todos los procesadores RISC, el registro 0 es la constante 0, por lo que cualquier escritura en este registro se pierde y cualquier lectura devuelve un cero.
- Instrucciones de tres direcciones: dos operandos y un destino.

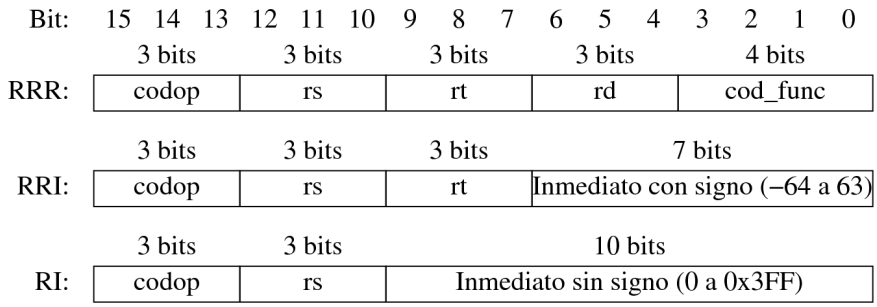

Figura 1. Formatos de instrucción de la arquitectura ICAI-RiSC-16

- Todas las operaciones aritméticas se realizan entre registros (máquina *load/store*).

2.1. Instrucciones y formatos

La arquitectura dispone tan solo de tres formatos de instrucción, los cuales se detallan en la Figura 1. El primer formato, RRR (Registro, Registro, Registro), se usa para codificar las instrucciones aritméticas. En todas ellas el código de operación (`codop`) vale cero y la operación aritmética a realizar se codifica en el campo `cod_func`. Todas estas instrucciones usan dos registros como operandos, especificados en los campos `rs` y `rt`, y un registro destino, especificado en el campo `rd`; a excepción de las instrucciones de desplazamiento, que sólo usan el registro `rs` como fuente y el registro `rd` como destino (el campo `rt` se deja a 0 en este caso). Todas estas instrucciones, así como su codificación, se muestran en la Tabla 1. Conviene destacar que se podrían implantar menos instrucciones aritméticas. En teoría bastaría con la instrucción NAND y la suma para poder realizar cualquier operación aritmética, tal como se expone en [1]. No obstante se complica demasiado la programación, por lo que para hacer la arquitectura más real se ha optado por incluir algunas instrucciones adicionales para facilitar la programación del microprocesador.

Además de estas instrucciones aritméticas, que trabajan sólo con registros, existe una instrucción adicional que permite sumar una constante a un registro y almacenar el resultado en otro registro. Dicha instrucción, denominada `addi`, se muestra en la Tabla 2. Como se puede apreciar, el formato usado para codificar esta instrucción es el RRI (Registro, Registro, Inmediato).

Si se usa el registro cero (constante cero) como operando (`rs`), la instrucción `addi` también sirve para cargar una constante en un registro, solo que dicha constante está limitada a 7 bits (-64 a

Ensamblador	Formato					Descripción
`add rd, rs, rt`	0	rs	rt	rd	0	rd <= rs + rt
`sub rd, rs, rt`	0	rs	rt	rd	1	rd <= rs – rt
`nand rd, rs, rt`	0	rs	rt	rd	2	rd <= rs NAND rt
`sll rd, rs`	0	rs	0	rd	3	rd <= rs desplazado 1 bit a la izquierda
`sra rd, rs`	0	rs	0	rd	4	rd <= rs desplazado aritméticamente 1 bit a la derecha
`srl rd, rs`	0	rs	0	rd	5	rd <= rs desplazado lógicamente 1 bit a la derecha
`sltu rd, rs, rt`	0	rs	rt	rd	6	rd <= 1 si rs < rt, 0 si no. Comparación sin signo

Tabla 1. Instrucciones aritméticas de la arquitectura ICAI-RiSC-16

63). Si se desea cargar una constante mayor, ha de hacerse en dos pasos: en primer lugar se cargan los 10 bits más significativos de la constante y a continuación se usa la instrucción `addi` para sumar los 6 bits menos significativos de la constante. Para cargar los bits más significativos existe la instrucción

Ensamblador	Formato				Descripción
`addi rt, rs, Inm`	1	rs	rt	Inmediato (-64 a 63)	rt <= rs + Inm

Tabla 2. Instrucción `addi` de la arquitectura ICAI-RiSC-16

Ensamblador	Formato			Descripción
`lui rs, Inm`	3	rs	Inmediato (0 a 0x3FF)	rs(15..6) <= Inm; rs(5..0) = 0

Tabla 3. Instrucción `lui` de la arquitectura ICAI-RiSC-16

`lui` (*Load Upper Immediate*), la cual se describe en la Tabla 3. Para codificar esta instrucción se usa el formato RI (Registro, Inmediato).

Como se ha mencionado antes, la arquitectura ICAI-RiSC-16 es una arquitectura *load-store*. Esto significa que todas las operaciones aritméticas se realizan sobre registros y que los únicos accesos a memoria se realizan mediante dos instrucciones, una para cargar un dato en un registro, denominada `lw`, y otra para guardar un registro en la memoria, denominada `sw`. En ambas instrucciones se usa un direccionamiento relativo a registro base, es decir, la dirección se calcula sumando un desplazamiento, especificado en el campo inmediato de la instrucción, a un registro que contiene la dirección base. Esto permite acceder fácilmente a vectores y a estructuras de datos. La codificación de ambas instrucciones se muestra en la Tabla 4, donde se puede apreciar que ambas instrucciones usan el formato RRI.

La arquitectura ICAI-RiSC-16 sólo dispone de una instrucción de salto condicional denominada `beq` (*Branch on EQual*). Esta instrucción compara dos registros y salta si ambos son iguales. La dirección de salto se calcula como PC + 1 + Inm, en donde PC es la dirección en la que está situada la instrucción `beq`. Obviamente el ensamblador se encarga de calcular el desplazamiento correcto a partir de la etiqueta de salto especificada en la instrucción. Nótese también que esta instrucción puede ser usada para realizar un salto incondicional comparando un registro con él mismo. La codificación de la instrucción se muestra en la Tabla 5.

Los saltos a subrutina se realizan mediante la instrucción `jalr` (*Jump And Link Register*). Esta instrucción salta a la dirección almacenada en el registro `rs` y almacena la dirección de retorno (PC +1) en el registro `rt`. La codificación de la instrucción se muestra en la Tabla 5. Otra utilidad de esta instrucción es la de poder realizar un salto absoluto a cualquier parte de la memoria, cargando previamente la dirección de destino del salto en un registro y usando el registro r0 para descartar la dirección de retorno.

2.2. Convención sobre el uso de registros

Ensamblador	Formato				Descripción
`sw rt, rs, Inm`	4	rs	rt	Inmediato (-64 a 63)	Memoria[rs + Inm] <= rt
`lw rt, rs, Inm`	5	rs	rt	Inmediato (-64 a 63)	rt <= Memoria[rs + Inm]

Tabla 4. Instrucciones de carga y almacenamiento de la arquitectura ICAI-RiSC-16

Ensamblador	Formato				Descripción
`beq rs, rt, Etiq`	6	rs	rt	Inmediato (-64 a 63)	Si rs = rt => PC<=PC+1+Inm.
`jalr rs, rt`	7	rs	rt	No usado. Debe estar a 0	PC = rs; rt <= PC + 1

Tabla 5. Instrucciones de salto de la arquitectura ICAI-RiSC-16

Registro	Uso
r0	Constante cero
r1	Devolución de resultados / registro temporal
r2	Argumento 1
r3	Argumento 2
r4	Temporal
r5	Variables
r6	Dirección de retorno
r7	Puntero de Pila

Tabla 6. Convención sobre el uso de registros

Aunque desde el punto de vista del *hardware* todos los registros son iguales (salvo el cero), es conveniente adoptar una convención sobre su uso para que las funciones sean interoperables. En la Tabla 6 se muestra la convención adoptada. Como se puede apreciar, siguiendo la filosofía RISC, no existe un registro especial para el puntero al tope de la pila, sino que se reserva el registro r7 para ello. Además la instrucción de salto a subrutina no guarda la dirección de retorno en la pila, sino en un registro. Esto tiene dos ventajas: por un lado se simplifica el *hardware*, pues la misma instrucción `jalr` puede ser usada para saltar a subrutina y para retornar de ella (saltando al registro r6 y usando el registro r0 para descartar la dirección de retorno). Por otro lado, si una función no llama a otra función, se ahorra tiempo al no ser necesario acceder a memoria, tanto al llamar a la función como al retornar. Por último, como se ha dicho en el apartado 2, sólo hay dos instrucciones para acceder a memoria (máquina *load/store*), por lo que si `jalr` accediese a memoria se estaría violando este principio (además de complicar el *hardware*).

3. Organización del ICAI-RiSC-16

La arquitectura presentada en la sección anterior admite varias organizaciones, desde una sencilla organización uniciclo, la cual tiene como ventaja principal la sencillez de su unidad de control; hasta una organización con *pipeline*. No obstante, en este trabajo se usa una organización tradicional multiciclo, la cual se considera que es la más apropiada para ser desarrollada por un alumno en el laboratorio de un curso introductorio. El alumno parte de una ruta de datos y una máquina de estados de control que se exponen en clase y su labor es implantarla en una FPGA. En esta sección se presenta brevemente la ruta de datos y el circuito de control usado.

3.1 Ruta de datos

En la Figura 2 se muestra la ruta de datos usada para implantar la arquitectura. Como se puede observar, los elementos de la ruta de datos son:

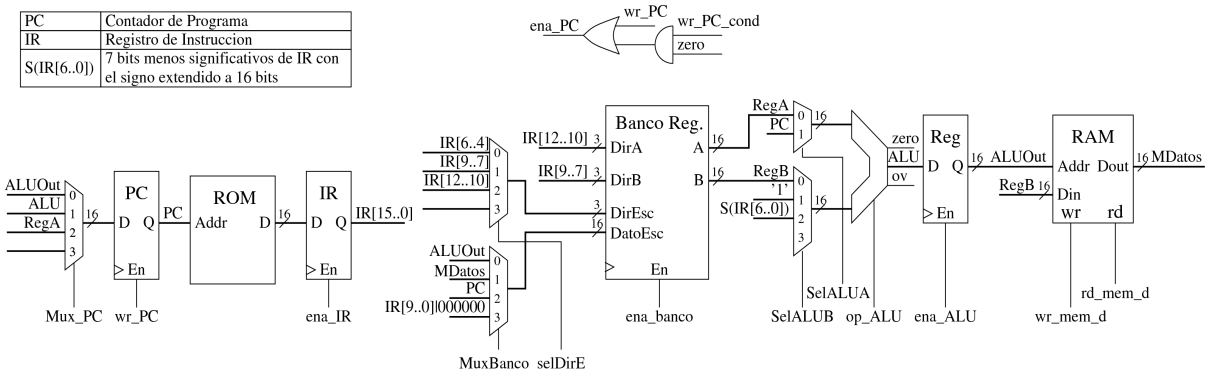

Figura 2: Diagrama de bloques de la ruta de datos del ICAI-RiSC-16

- Un registro (PC) para almacenar el contador de programa.
- Una memoria ROM para almacenar el programa.
- Un registro (IR) para almacenar la instrucción.
- Un banco de 8 registros de propósito general.
- Una ALU de 16 bits.
- Un registro para almacenar temporalmente el resultado de la ALU.
- Una memoria RAM para almacenar los datos del programa.
- Una serie de multiplexores para encaminar la información entre los distintos elementos del camino de datos en función de la instrucción que se esté ejecutando en cada momento.

Nótese que todos los registros de la ruta de datos se sincronizan con el flanco de subida del reloj.

3.2 Etapas de ejecución de las instrucciones

La ejecución de una instrucción se divide en una serie de etapas, cada una con una duración de un ciclo de reloj. Aunque no todas las instrucciones se ejecutan de la misma forma, todas ellas son bastante similares. Para empezar, las dos primeras etapas son idénticas para todas las instrucciones. El resto de etapas son distintas, aunque como se verá a continuación, son bastante similares. En primer lugar se estudian las dos etapas comunes a todas las instrucciones y a continuación, se expondrán las siguientes etapas para cada una de las instrucciones.

Para cada una de las etapas se muestra una figura con las unidades activas en la etapa. Para representar el estado de las señales en el diagrama se ha usado un código de colores. Así, las señales en color negro están inactivas, las señales de color verde están "transportando" datos y las señales en color rojo son señales de control que están activas en esta etapa. Si la señal de control es de un solo bit, el color rojo indica que está activa (y el negro que está inactiva). Si por el contrario la señal de control es de varios bits, entre paréntesis se indica su valor.

3.3 Etapa 1: Carga de la instrucción

En esta etapa, mostrada en la Figura 3, se carga de memoria ROM la instrucción a la que apunta el PC y se almacena en el registro IR, pues se necesita para el resto de etapas. Además se usa la ALU para incrementar el PC en 1 y así apuntar a la siguiente instrucción.

3.4 Etapa 2: Decodificación y cálculo de la dirección de salto

En esta etapa, mostrada en la Figura 4, mientras la unidad de control decodifica la instrucción, se usa la ALU para calcular la dirección de destino del salto, pues los datos necesarios para ello ya están disponibles. Esto permite ahorrar un ciclo en el caso de que la instrucción sea efectivamente un salto condicional. Si al decodificar la instrucción, ésta es de otro tipo, simplemente se descarta el valor calculado por la ALU.

Las siguientes etapas dependen del tipo de instrucción decodificada en esta etapa.

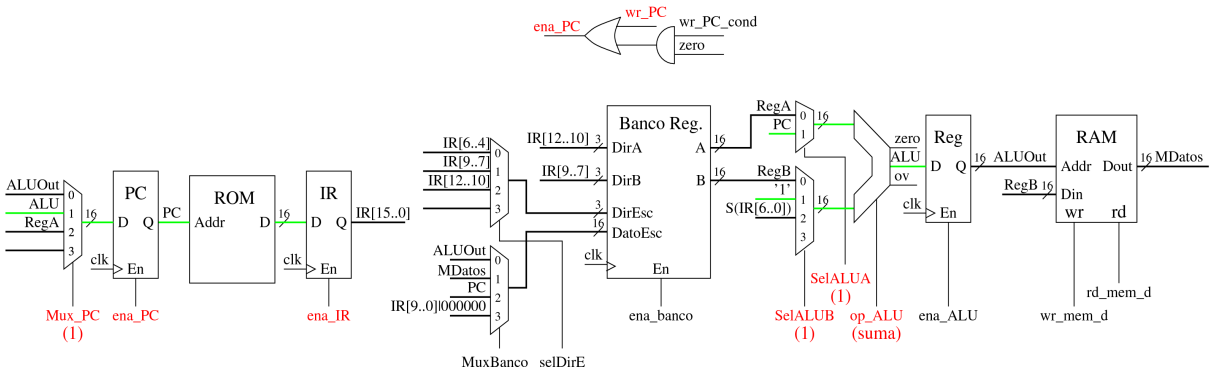

Figura 3. Etapa 1: carga de la instrucción

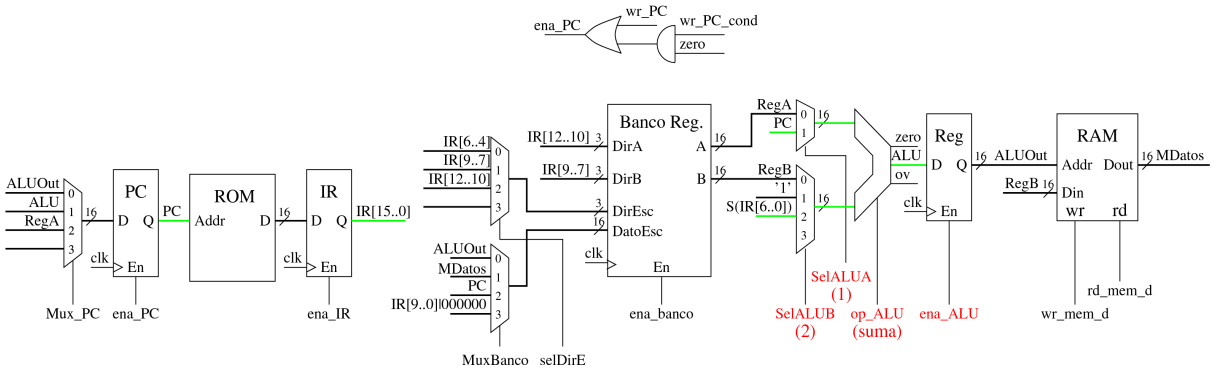

Figura 4. Etapa 2: decodificación y cálculo de la dirección de salto

3.5 Etapa 3 de `beq`: salto condicional

La tercera etapa de la instrucción `beq`, mostrada en la Figura 5, se comparan los dos registros especificados en la instrucción y si estos son iguales se cargará en el PC la dirección de destino del salto, que se almacenó en la etapa anterior en el registro ALUOut. En la figura se muestran en color azul las señales que dependen de la comparación de los dos registros. Esta instrucción finaliza en esta etapa.

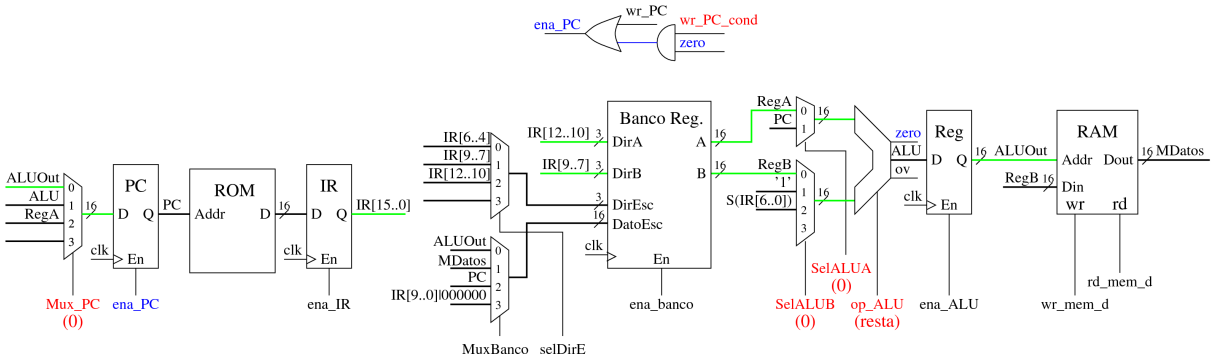

Figura 5. Etapa 3 de beq: salto condicional

3.6 Etapa 3 de las instrucciones aritméticas

En la tercera etapa de las instrucciones aritméticas, se realiza en la ALU la operación especificada en la instrucción mediante el campo `cod_func`. Para ello la ALU ha de haberse diseñado de acuerdo con la codificación del campo `cod_func`, mostrado en la Tabla 1. El funcionamiento de la etapa se ilustra en la Figura 6.

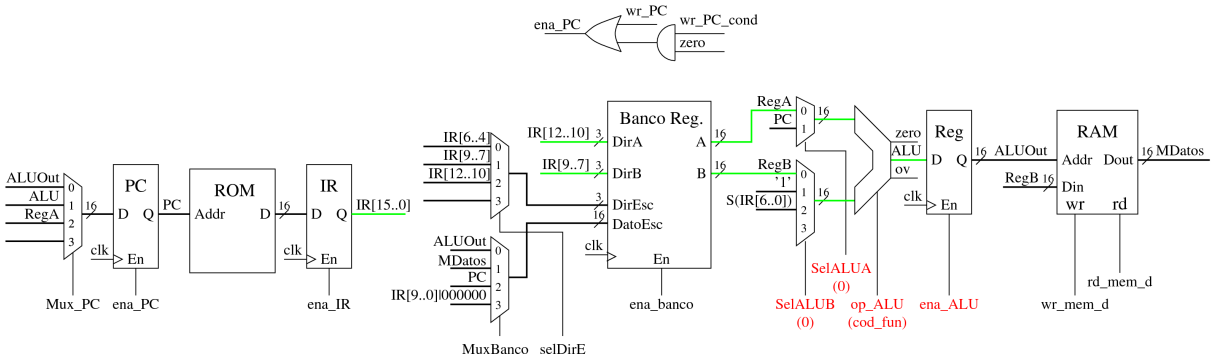

Figura 6. Etapa 3 de las instrucciones aritméticas

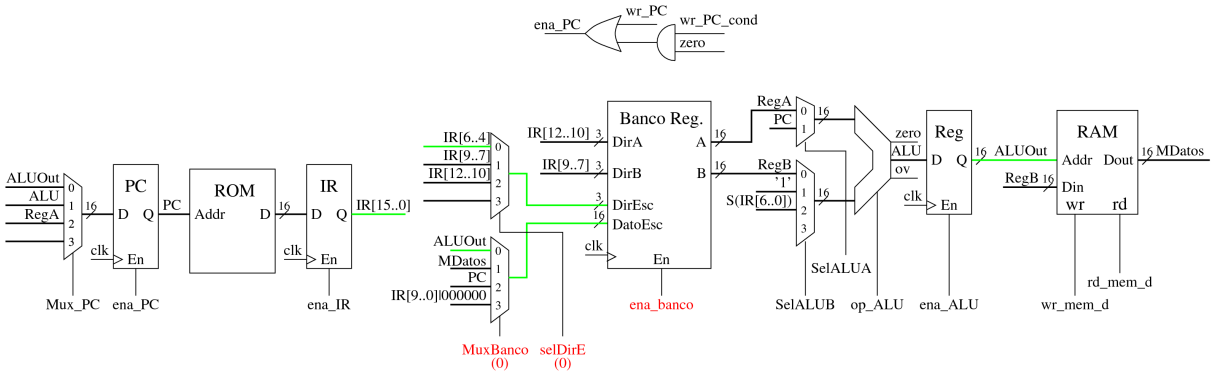

Figura 7. Etapa 4 de las instrucciones aritméticas

3.7 Etapa 4 de las instrucciones aritméticas

En esta última etapa, el resultado de la operación de la ALU calculado en la etapa anterior y almacenado en el registro temporal ALUOut, se guarda en el banco de registros. La Figura 7 ilustra el funcionamiento de la etapa.

3.8 Etapa 3 de las instrucciones de carga y almacenamiento

En la tercera etapa, estas instrucciones calculan la dirección de memoria del dato a leer o a escribir en memoria. Para ello se suma el contenido del registro base, especificado en el campo rs al campo inmediato de la instrucción con el signo extendido. Las unidades del camino de datos activas en esta etapa se muestran en la Figura 8.

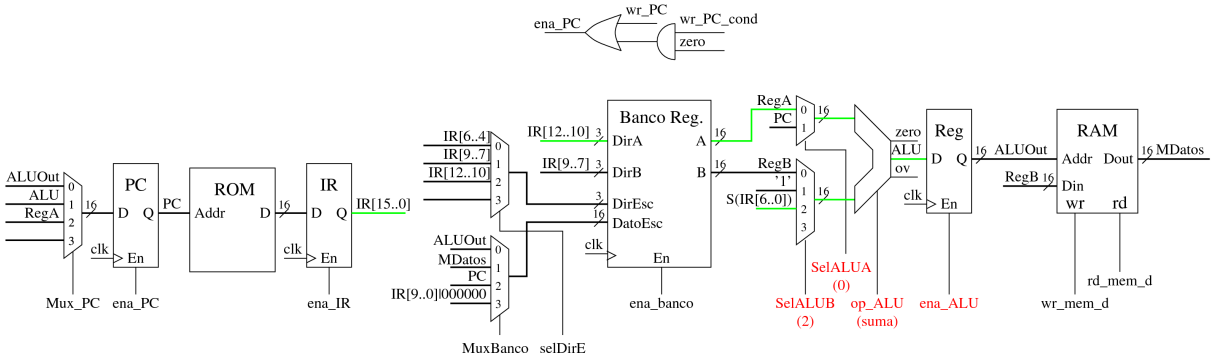

Figura 8. Etapa 3 de las instrucciones lw/sw

3.9 Etapa 4 de la instrucción de carga

En esta etapa se carga de memoria la palabra almacenada en la dirección calculada en la etapa anterior, la cual está almacenada en el registro temporal ALUOut. El dato leído de memoria se almacena en el registro indicado en el campo rt, tal como se ilustra en la Figura 9.

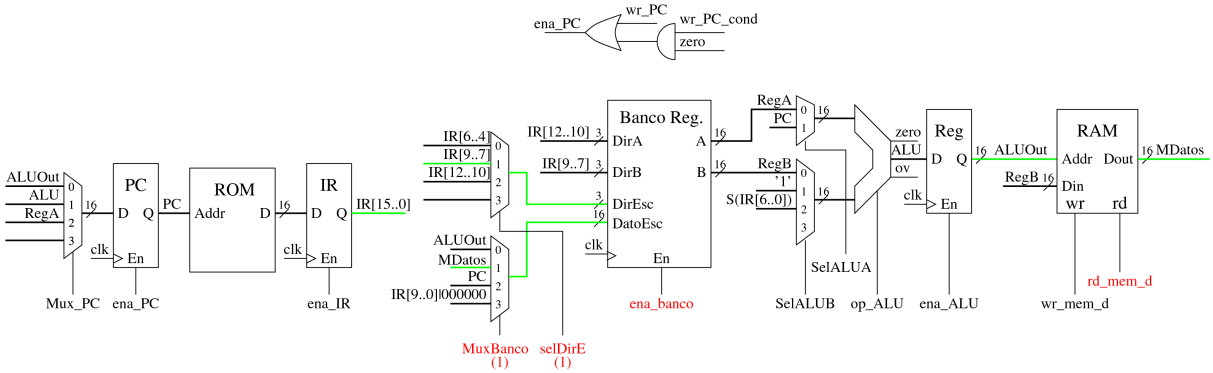

Figura 9. Etapa 4 de la instrucción lw

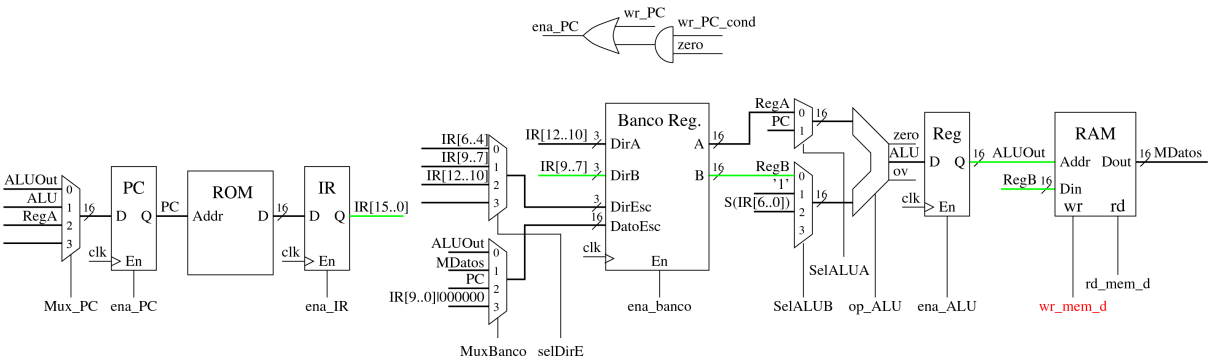

Figura 10. Etapa 4 de la instrucción sw

3.10 Etapa 4 de la instrucción de almacenamiento

En esta etapa. Mostrada en la Figura 10, se guarda el contenido del registro indicado en el campo rt en la posición de memoria cuya dirección se ha calculado en la etapa 3 y que está almacenada en el registro ALUOut.

3.11 Resto de instrucciones

Por motivos de espacio, no se ha incluido en la comunicación el funcionamiento del resto de instrucciones, `addi`, `lui` y `jalr`, aunque es fácil deducirlo a partir de las instrucciones básicas expuestas. Tampoco se especifican estas instrucciones en la documentación de prácticas aportada al alumno, dejándolas como ejercicio para que profundice en la arquitectura.

3.12 Circuito de control

El circuito de control de la CPU, se encarga de generar las señales de control de la ruta de datos para cada una de las fases de ejecución de las instrucciones. Aunque existen varias técnicas para realizar este tipo de circuitos, en esta CPU tan simple lo más apropiado es usar una máquina de estados. El diagrama de estados de dicha máquina se muestra en la Figura 11. Como se puede apreciar, en cuanto se desactiva la señal de reset, la máquina de estados empieza a ejecutar instrucciones. Así, el primer estado es el de carga de la instrucción de memoria (FETCH) y el siguiente el de decodificación (DECOD). El estado siguiente se elige en función del código de operación de la instrucción. Así, por ejemplo, si el código de operación es 0 (instrucción aritmética) se pasará al estado ARIT3 para ejecutar la tercera fase de la instrucción y luego al estado ARIT4 para ejecutar la última fase. Como se puede observar, desde el último estado de todas las instrucciones se pasa siempre de nuevo al estado de carga (FETCH), para pasar así a ejecutar la siguiente instrucción.

Nótese que al desactivar la señal de reset, la máquina de estados cargará la primera instrucción del programa. Esta instrucción se cargará desde la dirección a la que apunte el PC, por lo que el valor inicial de este registro definirá dónde empieza el programa. Por ejemplo, si el valor inicial vale 0, la primera instrucción del programa ha de estar al principio de la memoria ROM de programa.

4. Implantación de la arquitectura ICAI-RiSC-16

El procesador descrito en esta comunicación ha sido implantado por los autores en una plataforma de desarrollo basada en la FPGA Flex10k30 de la firma Altera, ocupando un 36% del chip (629 bloques lógicos). Esto demuestra los pocos recursos que son necesarios para implantar la arquitectura descrita, por lo que puede usarse prácticamente con cualquier plataforma de lógica programable. El circuito ha sido implantado usando el programa Quartus II de Altera. La mayoría de los bloques se han especificado en VHDL. No obstante, elementos como las memorias han sido diseñados usando los asistentes proporcionados por la herramienta. A pesar de ello el diseño es fácilmente migrable a otras arquitecturas, sin más que modificar estos bloques de memoria.

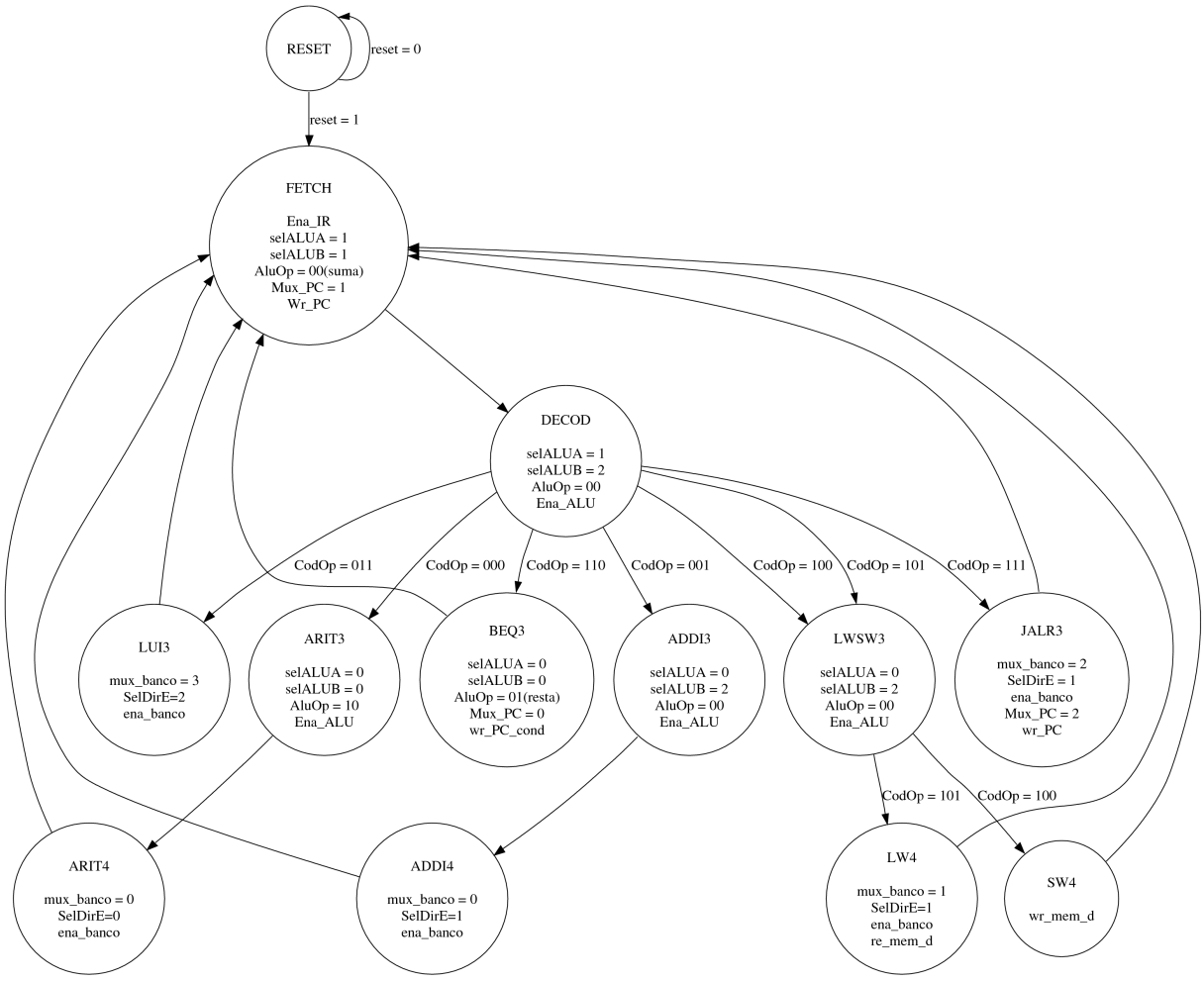

Figura 11: Diagrama de estados del circuito de control

5. Desarrollo del laboratorio

Esta arquitectura se ha usado para la docencia de la asignatura "Sistemas Electrónicos Digitales" de 1º de la titulación de segundo ciclo "Ingeniero en Automática y Electrónica Industrial". En esta asignatura se afianzan los conocimientos de electrónica digital adquiridos por el alumno en un curso introductorio realizado en el primer ciclo. En la asignatura se hace especial énfasis en la implantación con lógica programable y en la descripción de circuitos usando VHDL.

La asignatura consta de un laboratorio en el que el alumno pone en práctica los conocimientos adquiridos en la teoría, usando para ello una herramienta CAD (Quartus II) y una placa de desarrollo basada en una FPGA de la familia Flex 10k de Altera. El laboratorio consta de las siguientes prácticas [3]:

1. Introducción al sistema de desarrollo. En esta práctica el alumno diseña la ALU de microprocesador ICAI-RiSC-16. Este diseño sencillo le permite familiarizarse con la herramienta CAD.
2. Comunicación Serie. En esta práctica el alumno diseña una UART básica para comunicar la placa con un PC. Lo que se pretende en esta práctica es que el alumno utilice técnicas de diseño jerárquico para abordar un diseño complejo. Además esta práctica no requiere grandes conocimientos previos, por lo que puede realizarse mientras que en la teoría se expone la arquitectura ICAI-RiSC-16.
3. Arquitectura ICAI-RiSC-16. En esta práctica el alumno diseña la arquitectura expuesta en esta comunicación. El trabajo se divide en dos partes: en primer lugar se diseña el camino de datos

y en segundo lugar el circuito de control. Al alumno se le proporciona un programa ensamblador básico que le permite generar programas de prueba.

5.2. Resultados obtenidos

Al ser este el primer año en usar la arquitectura, se ha sido demasiado optimista en cuanto a la estimación del tiempo necesario para la realización de la práctica. De los 8 grupos de alumnos que han cursado el laboratorio, sólo tres de ellos tuvieron tiempo suficiente para finalizar la práctica en cuatro sesiones de 2 horas. No obstante, el resto de alumnos fueron capaces de implantar tanto la ruta de datos como el control, aunque no tuvieron tiempo suficiente para realizar la depuración y puesta en marcha del circuito.

A pesar de todo ello, la experiencia ha sido positiva por las siguientes razones:

- El concepto de microprocesador ha sido "desmitificado". Al ser capaz de construir un microprocesador, aunque sea básico, el alumno comprende que una CPU no es algo "ultrasofisticado", sino sólo un circuito complejo.
- El alumno ha comprendido conceptos claves de arquitectura de ordenadores: cómo se codifica un programa en lenguaje máquina, cómo se lee una instrucción, se decodifica y se ejecuta; qué es un banco de registros y cómo se implanta, etc.
- El diseño de un circuito de la complejidad de un microprocesador le obliga al alumno a dividir su implantación en ruta de datos + circuito de control, lo cual no es tan estrictamente necesario cuando se diseñan circuitos más simples.
- En este diseño complejo es necesario usar todos los bloques básicos estudiados en electrónica digital: multiplexores, registros, sumadores, etc. Por tanto, es una buena práctica final de laboratorio que sintetiza lo aprendido en éste y en los cursos anteriores.

6. Conclusiones

En esta comunicación se ha presentado una arquitectura de un procesador RISC que es lo suficientemente sencilla como para ser usada con fines docentes, pero lo suficientemente compleja como para que pueda usarse en aplicaciones reales.

En la comunicación se ha expuesto en detalle la arquitectura propuesta, así como su implantación mediante una organización multiciclo. La exposición realizada es un resumen del material proporcionado al alumno para la realización de la práctica, la cual se encuentra al completo en la página web del laboratorio [3].

Se ha mostrado también que la arquitectura diseñada requiere muy pocos recursos, ocupando tan solo 629 bloques lógicos de una FPGA de la familia Flex 10k.

La arquitectura descrita ha sido usada para la docencia de la asignatura "Sistemas Electrónicos Digitales" de 1º de IAEI. La arquitectura ha sido expuesta en la clase teórica para que el alumno la implante en el laboratorio usando una organización multiciclo. Al ser este el primer año en el que se ha usado esta práctica, el tiempo asignado no ha sido suficiente para la mayoría de los alumnos, habiendo finalizado la práctica sólo el 37,5 % de los alumnos. No obstante, los objetivos docentes perseguidos si se han obtenido, ya que todos los alumnos han aprendido el funcionamiento de una CPU, así como el proceso de diseño de un circuito complejo, dividiendo su implantación en ruta de datos + circuito de control.

Referencias

[1] Jacob, B. *The RiSC-16 Architecture.* Disponible Online en: http://www.eng.umd.edu/~blj/RiSC/ (última visita 30/04/08)

[2] D. A. Patterson y J. L. Hennessy. *Computer Organization and Design. The Hardware/Software Interface.* 3ª Edición. Editorial Morgan Kaufmann/Elsevier. San Francisco (2005).

[3] J. D. Muñoz Frías. *Página web del Laboratorio de Sistemas Electrónicos Digitales*. Disponible Online en: http://www.dea.icai.upcomillas.es/daniel/asignaturas/SisEleDig_1_IAEI/ (última visita 30/04/08)