Implementación de una CPU monociclo con unidad de control microprogramada.

Alexis Rodríguez Casañas.
Alberto Martín Núñez

20.05.2016

Introducción

El presente informe resume y explica la implementación de una CPU monociclo microprogramada y consta de dos secciones. En la primera parte se hace referencia a la CPU de forma genérica en su versión definitiva; se muestran y explican brevemente el repertorio de instrucciones y el camino de datos así como cambios, detalles o dificultades surgidas durante el diseño o la implementación. En la segunda parte se hace referencia al caso concreto de una calculadora tras haber cargado el programa correspondiente en dicha CPU. Se detalla el programa realizado así como la simulación y la prueba real en el hardware. Si se desean conocer detalles a nivel de desarrollo, los códigos fuente están debidamente comentados.

Índice

Sección 1 (CPU)

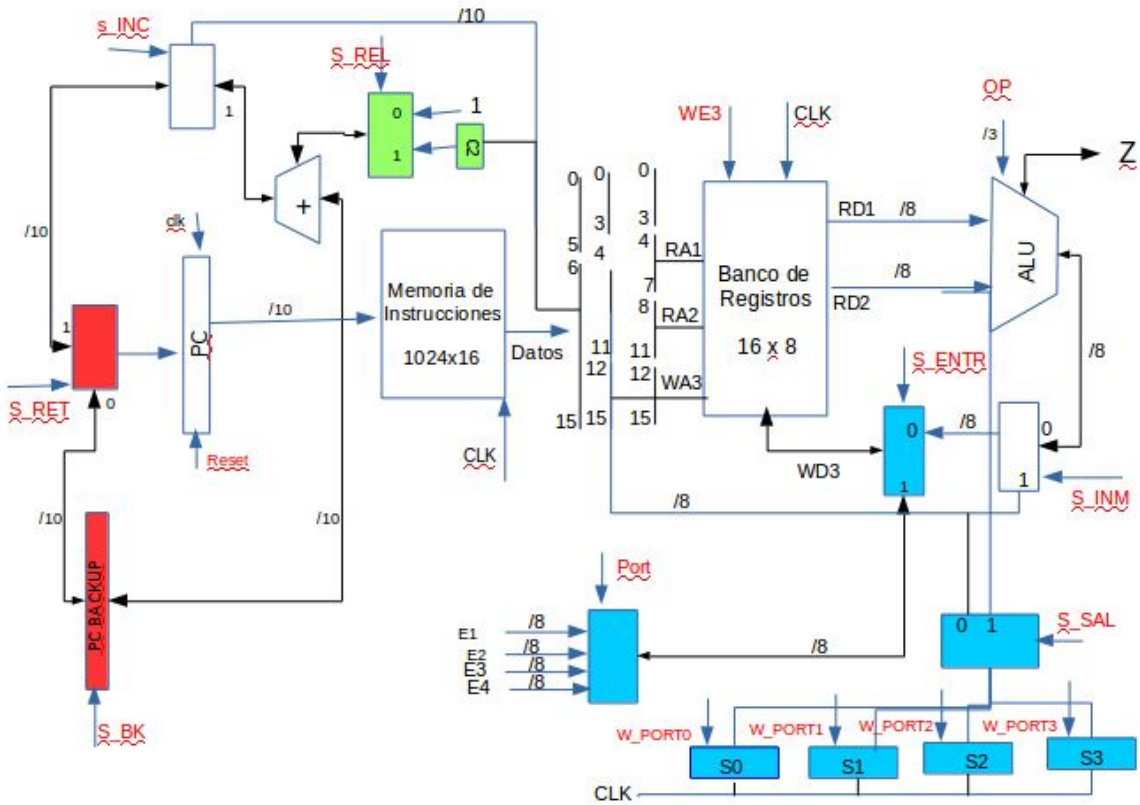
Descripción

Se plantea el desarrollo de una CPU con unidad de control microprogramada. Esta CPU está basada en la arquitectura Harvard (memorias

independientes para datos e instrucciones) y es monociclo. Debe ser capaz de hacer las operaciones de ALU básicas, así como saltos condicionales, relativos, incondicionales y a subrutina. Así mismo, debe tener la capacidad de comunicarse con el “exterior” por lo que también dispone de puertos de entrada y salida con las correspondientes instrucciones para su uso.

Camino de datos

A continuación se detalla el camino de datos final, que se ha ido enriqueciendo conforme a las sucesivas entregas.
El primer requisito que necesitó de añadir hardware adicional fue la comunicación con el exterior mediante puertos. En azul, se detalla el hardware implementado. Seguidamente se implementó la funcionalidad de salto relativo, para lo que fue añadido el hardware coloreado en verde. Finalmente, en rojo se muestran los componentes necesarios para el último requisito; la subrutinas. Las señales salientes, en rojo, van conectadas a la CPU.
Un problema surgido durante la implementación fue la necesidad de retrasar el valor Z de la operación de ALU, lo cual se solucionó conectando un registro a dicha salida. A continuación surgió otro problema y era que en este registro se escribían datos no válidos si se ejecutaban dos saltos seguidos. Esto se solucionó utilizando un registro con enable, conectando dicho enable al bit de operación de ALU. Con esto, el registro Z conservaría la última salida **válida**.



Repertorio de instrucciones

Las instrucciones son de 16 bits. Se han utilizado 4 bits para el OPCODE de las operaciones de ALU y 6 para el resto (a excepción de la operación cargar un inmediato, que también utiliza un OPCODE de 4 bits). A continuación se detalla el diseño de dichas operaciones.

Los registros utilizan 4 bits, los inmediatos 8 bits y las direcciones de memoria son de 10 bits. El registro $0 al igual que en MIPS está reservado y siempre contiene el valor 0.

INSTRUCCIÓN											OPCODE					
											OPCODE					
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Asignación	REG DEST				0				REG A				0	0	0	0
Ca1 de A	REG DEST				0				REG A				0	0	0	1
A + B	REG DEST				REG B				REG A				0	0	1	0
A - B	REG DEST				REG B				REG A				0	0	1	1
A & B	REG DEST				REG B				REG A				0	1	0	0
A | B	REG DEST				REG B				REG A				0	1	0	1
Ca2 de A	REG DEST				0				REG A				0	1	1	0
Ca2 de B	REG DEST				REG B				0				0	1	1	1
Cargar inmediato	REG DEST				DATO								1	0	0	0
Salto incondicional	DIRECCIÓN										0	0	1	0	0	1
Salto condicional (si 0)	DIRECCIÓN										0	0	1	0	1	0
Salto condicional (si no 0)	DIRECCIÓN										0	0	1	0	1	1
Salto relativo	S	DIRECCIÓN									0	1	1	0	0	1
Salto a subrutina	DIRECCIÓN										0	1	1	0	1	0
Retorno de subrutina	0										0	1	1	0	1	1
Leer de puerto	REG DEST				0				PORT		0	0	1	1	0	0
Escribir en puerto	0				REG ORIG				PORT		0	0	1	1	0	1
Esc. inmediato en puerto	DATO								PORT		0	0	1	1	1	0

ATR: Traductor de codigo en formato assembler.

Dado que la escritura de código en binario resulta laboriosa, lenta y se presta a comenter muchos errores (más aún cuando utilizamos saltos) se ha programado una pequeña herramienta para facilitar la escritura del mismo con la cual esta codificación es invisible para el programador. En la presente entrega se adjunta un manual que trata exclusivamente sobre dicha herramienta y sobre el repertorio de instrucciones en formato ensamblador que el programador puede usar.

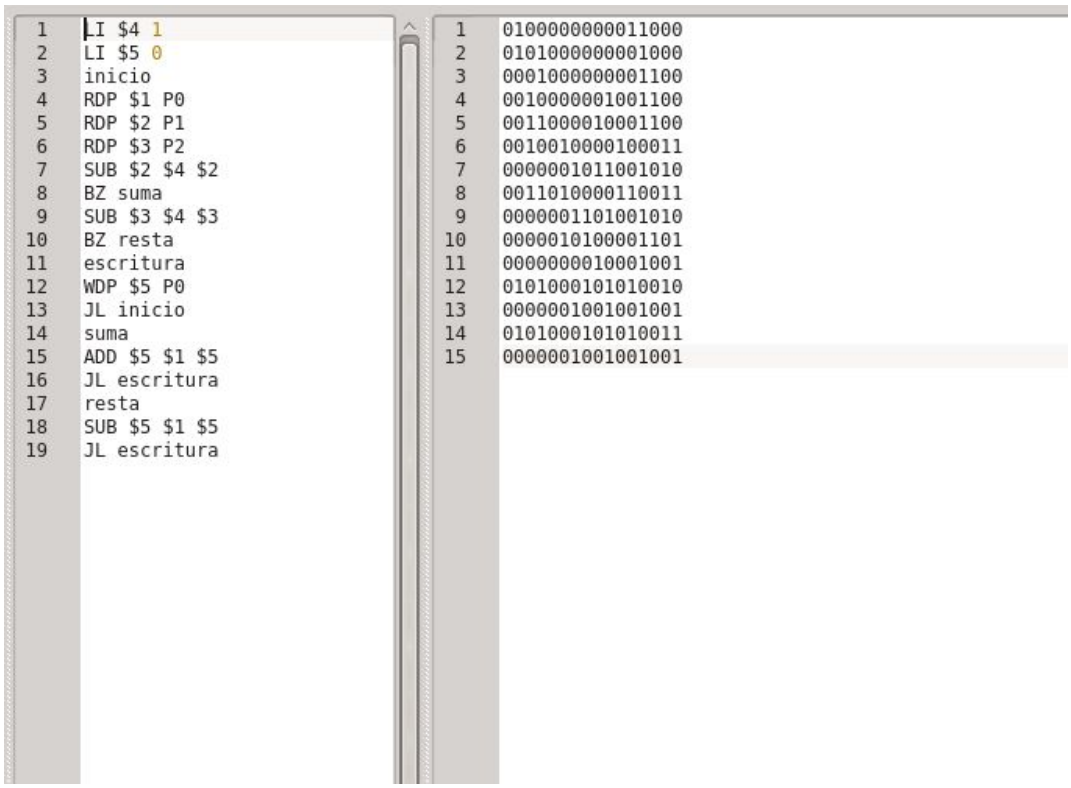
Sección 2 (Calculadora)

Descripción

Se plantea, partiendo de la CPU realizada, la implementación de una calculadora. Dicha calculadora es de un solo registro, es decir, las operaciones son del tipo ACUMULADOR = ACUMULADOR +/- OPERANDO. Para ello, se ha utilizado el mapeo que se muestra a continuación y se han codificado los displays de 7 segmentos para controlar el número introducido con mayor facilidad.

Programa implementado

Se ha implementado un sencillo programa haciendo uso de la herramienta de traducción. Dicho programa, con su respectiva codificación para la CPU es el que se adjunta a continuación. Se recomienda consultar el manual del traductor para comprender las instrucciones.

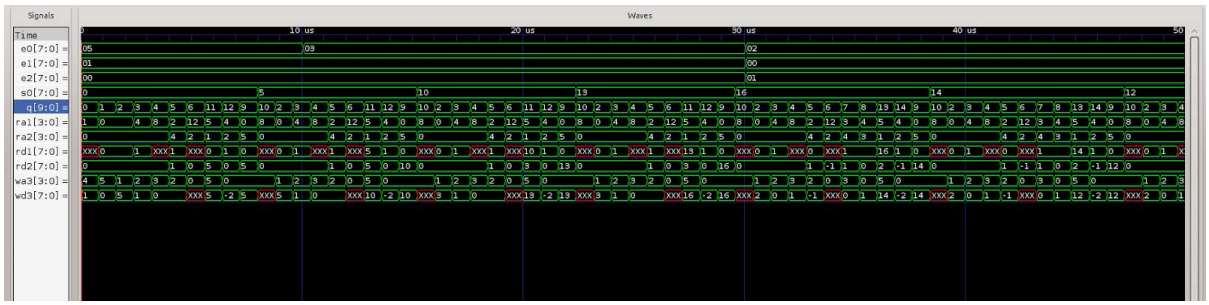


```
1  LI $4 1
2  LI $5 0
3  inicio
4  RDP $1 P0
5  RDP $2 P1
6  RDP $3 P2
7  SUB $2 $4 $2
8  BZ suma
9  SUB $3 $4 $3
10 BZ resta
11 escritura
12 WDP $5 P0
13 JL inicio
14 suma
15 ADD $5 $1 $5
16 JL escritura
17 resta
18 SUB $5 $1 $5
19 JL escritura
```

```
1  0100000000011000
2  0101000000001000
3  0001000000001100
4  0010000001001100
5  0011000010001100
6  0010010000100011
7  0000001011001010
8  0011010000110011
9  0000001101001010
10 0000010100001101
11 0000000010001001
12 0101000101010010
13 0000001001001001
14 0101000101010011
15 0000001001001001
```

Simulación genérica en GTKWave

Una vez está todo preparado, se ha procedido a una primera simulación en GTKWave, como se muestra en la siguiente imagen.



Descripción de las señales:
e0: Operando.
e1: Operación suma.
e2: Operación resta.
S0: Salida (resultado).
q0: Contador de programa.
Resto de señales: Banco de registros (véase el camino de datos).

Se puede observar como efectivamente, como dicta el código visto anteriormente, se escribe en la salida el resultado de efectuar la operación correspondiente en la instrucción 9, que es la de escritura en puerto (nótese que el PC apunta a la **siguiente** instrucción). Véase como se acumula el resultado aunque varíe el operando o la operación.

Simulación funcional en ModelSim

Una vez hecha una primera aproximación, se procede a la simulación funcional con ModelSim. GTKWave es una herramienta genérica, pero con el IDE Quartus y el simulador ModelSim ya estamos especificando un hardware concreto, por lo que es una simulación más realista y exigente. Efectivamente ya fue necesario cambiar algunas partes del código que eran demasiado genéricas o implícitas. Tras la correcta compilación se procedió a la simulación, que como se muestra, también se comportó de acuerdo a lo esperado.

Prueba en FPGA

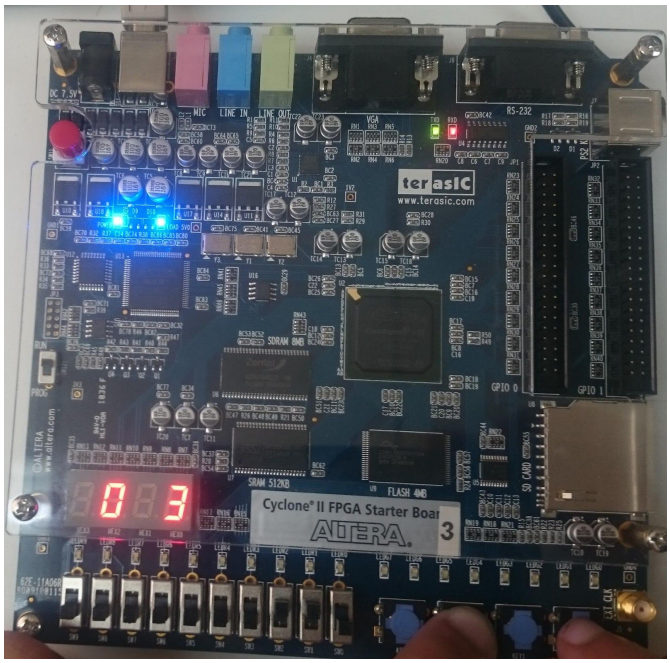
Mapeo de los puertos

Antes de proceder, se han procedido a mapear los puertos como se muestra a continuación.

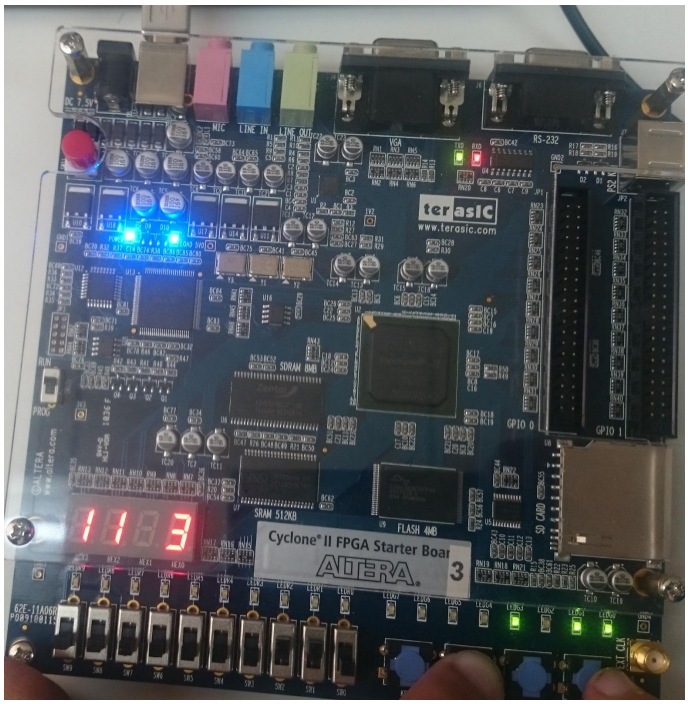
1: Clock, 2: Suma, 3: Resta, 4: Reset, 5: Operando (4 bits).
También se han utilizado los displays de 7 segmentos proporcionados, utilizando los dos de la izquierda para mostrar el contador de programa y los dos de la derecha para mostrar el operando introducido.

Ejecución en la FPGA

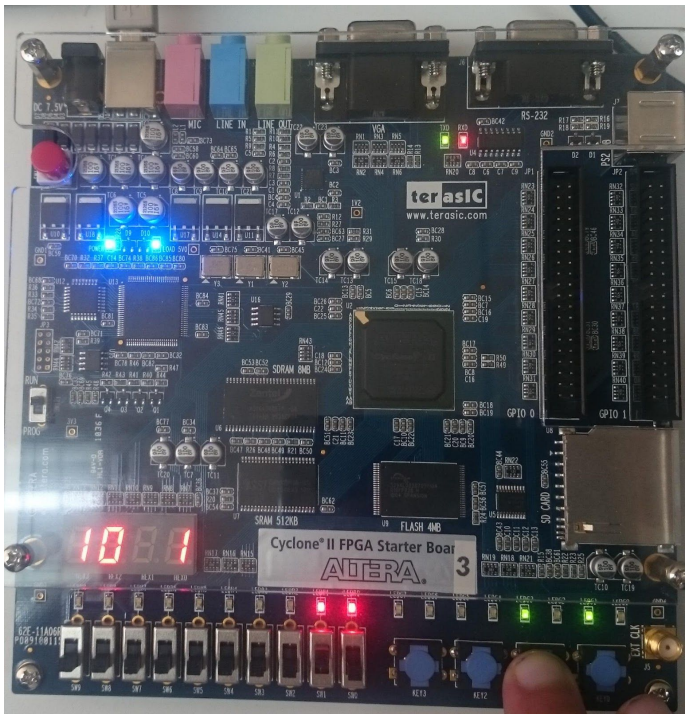
Una vez está todo listo, se puede proceder a la implementación en la FPGA. Para ello se ha implementado el módulo **fpga.v** que hará de capa superior, encapsulando todo lo demás. Este módulo conectará los displays y dará un formato correcto al operando y las operaciones para garantizar su correcto funcionamiento, ya que con los botones leemos 1 bit y los puertos de entrada son de 8 bits. Se muestra a continuación una traza del resultado obtenido. Se puede comprobar el program counter con el programa visto al principio de esta sección.



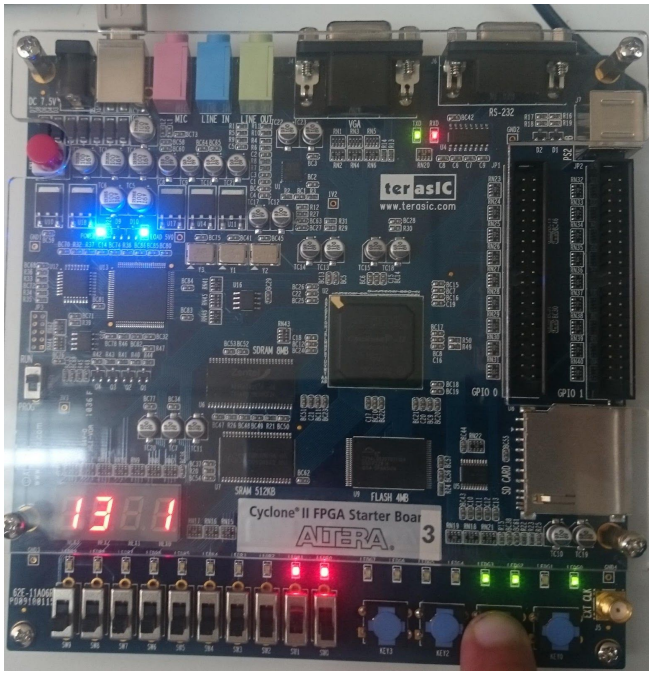
Comienza la ejecución con un 0 en el acumulador. Introducimos un 3 como operando y elegimos la operación de suma. Recuérdese que el display de la izquierda contiene el program counter y no el acumulador.



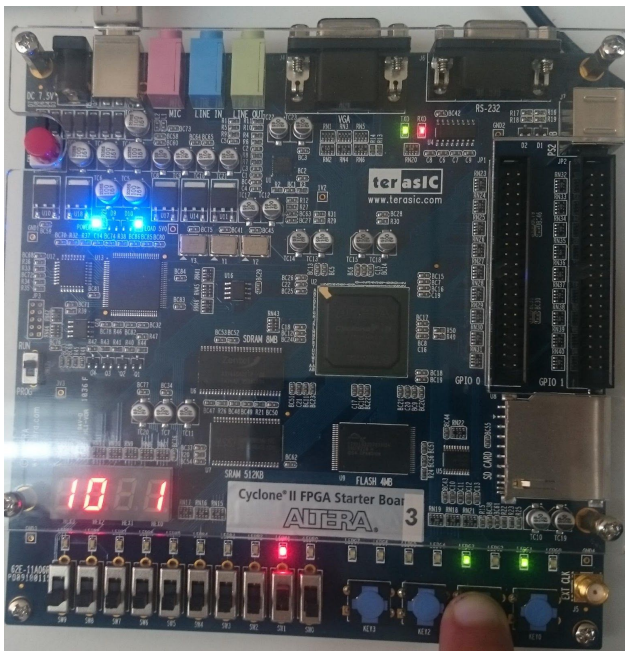
Como queremos sumar, el program counter ha saltado a la línea de la subrutina suma.



Al igual que en la simulación, el resultado se ha escrito cuando el program counter apunta a la instrucción número 10. El acumulador contenía un 0 y tras aplicar la suma del operando, ahora contiene un 3, reflejado en la salida. Se ha introducido ahora otro operando y se ha seleccionado hacer la operación de resta.



El programa vuelve al principio, pero al contrario que en el caso anterior el program counter apunta a la instrucción 13, que es la línea perteneciente a la subrutina suma.



Nuevamente se escribe el resultado cuando el program counter apunta a la instrucción 10. El acumulador contenía un 3 y tras la operación resta con el nuevo operando (1), el acumulador ahora contiene un 2, tal como se refleja en la salida.

Prueba definitiva: Calculadora con reloj integrado

Una vez todo funciona correctamente, se debe proceder a conectar el reloj de la CPU al reloj interno de la FPGA para que funcione de forma autónoma. Cambiado esto en el mapeo de pines, fue necesario modificar el programa para que se quedase retenido mientras el usuario no soltase el botón de operación, ya que al estar hablando de Mhz, por muy rápido que se pulse una operación ésta se ejecutará como mínimo miles de veces y desbordará el acumulador.
A continuación se presenta el programa con su equivalente a código máquina una vez pasado por la herramienta de traducción.

```
**Calculadora
**AlexisR.C. y Alberto M.N.
LI $4 1
LI $5 0
inicio
RDP $1 P0
RDP $2 P1
RDP $3 P2
SUB $2 $4 $2
BZ suma
SUB $3 $4 $3
BZ resta
escritura
WDP $5 P0
JL inicio
suma
ADD $5 $1 $5
JL clearSuma
resta
SUB $5 $1 $5
JL clearResta
*
***Control de pulsacion
clearSuma
RDP $2 P1
SUB $2 $4 $2
BNZ escritura
JL clearSuma
clearResta
RDP $3 P2
SUB $3 $4 $3
BNZ escritura
JL clearResta
```

```
0100000000011000
0101000000001000
0001000000001100
0010000001001100
0011000010001100
0010010000100011
0000001011001010
0011010000110011
0000001101001010
0000010100001101
0000000010001001
0101000101010010
0000001111001001
0101000101010011
0000010011001001
0010000001001100
0010010000100011
0000001001001011
0000001111001001
0011000010001100
0011010000110011
0000001001001011
0000010011001001
```