Diseño del microprocesador GUS-16

Jesús Arias Alvarez

Dpto. de Electicidad y Electrónica. E.T.S.I. Telecomunicación.

Universidad de Valladolid

Contents

1 Introducción

En este documento se detalla el diseño de un microprocesador simple junto con su repertorio de instrucciones y una colección de periféricos. Se trata de un micro lo suficientemente simple como para poder abordar sin dificultades el diseño de cada uno de sus bloques funcionales, pero a su vez no he querido caer en un exceso de simplificación que convirtiese a nuestro micro en una mera curiosidad académica sin ningún posible uso práctico a causa de sus limitaciones. En particular he creído necesario que esta CPU fuese capaz de implementar un mecanismo de llamadas a subrutinas y he prestado atención al problema de los operandos inmediatos (constantes).

Este diseño sigue la filosofía RISC, en la que las instrucciones son lo suficientemente sencillas como para poder ejecutarse en un único ciclo de reloj, al menos en el diseño inicial que se presenta como una idea general del micro, pero que he abandonado en favor de una segunda variante que incluye el mecanismo del "pipelining" en la que además he hecho un cambio fundamental en la arquitectura que pasa de ser de tipo Harvard, con memorias de programa y de datos separadas, a Von Neumann con una única memoria para programas y datos.

Una variante posterior tiene como objetivo el añadir soporte para interrupciones. Esto se ha conseguido gracias a haber duplicado algunos de los registros como el contador de programa y los flags, que ahora tienen una copia en uso en los programas normales y otra copia distinta en uso durante las interrupciones.

Finalmente, se diseña un sistema en el que el micro anterior se rodea de memoria y periféricos en una FPGA para acabar siendo un microcontrolador práctico, capaz de ejecutar los programas que desarrollemos para él.

Los diseños de las últimas variantes se han escrito en lenguaje Verilog y se han sintetizado con éxito en una FPGA ICE40HX1K de Lattice. Esta es una FPGA económica que tiene la particularidad de disponer de herramientas de desarrollo de dominio público, aunque por otra parte está un tanto limitada en lo tocante a cantidad de celdas lógicas. Sin embargo ha resultado ser suficiente para incluir la totalidad del micro diseñado junto con unos pocos periféricos.

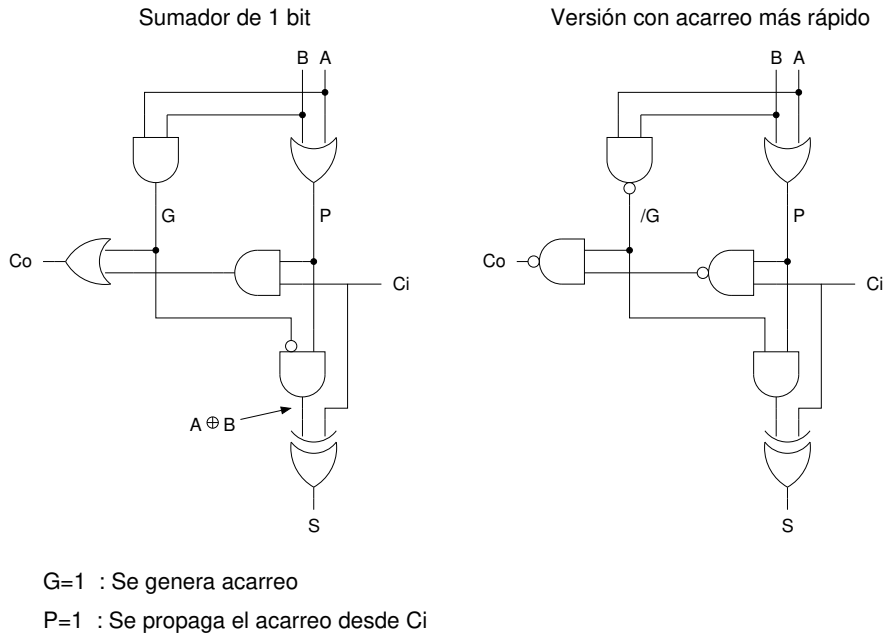


Figure 1: Esquema de un sumador de 1bit

1.1 ¿Por qué GUS-16?

El gusano Caenorhabditis Elegans en su fase adulta está formado por un número constante de células que no es muy distinto del de las celdas lógicas de FPGA necesarias para sintetizar estos diseños. Este animalito no destaca por tener el tamaño de un elefante ni la rapidez de un guepardo, sino por la extremada sencillez de su anatomía, aspectos en los que coincide el diseño de CPU que se describe en este documento. El sufijo '-16' hace referencia al número de bits de sus registros.

2 Unidad Aritmético-Lógica (ALU)

Las CPUs originalmente se diseñaron con el fin de realizar operaciones aritméticas. Es decir: Eran lo que hoy conocemos como calculadoras. Por ello no es extraño que uno de los bloques fundamentales de cualquier CPU sea precisamente la ALU, y ello a pesar de tratarse de un circuito puramente combinacional. Este será por tanto el primer bloque cuyo diseño hay que abordar, antes incluso de plantearnos el diagrama de bloques del resto de la CPU.

La principal operación que ha de realizar una ALU es la de la suma de números binarios. Por ello comenzamos presentado en la figura 1 el esquema de un sumador de un bit en el que basaremos el diseño posterior de la ALU.

En la figura de la izquierda se muestra dicho sumador en el que hacemos especial mención al circuito de generación del acarreo: Se genera (señal G) acarreo en la salida, independientemente de la entrada Ci, si los dos bits de entrada, A y B, son ambos 1. Si sólo uno de los dos bits , A o B, es 1, se generará acarreo si Ci es 1, o dicho de otro modo, el acarreo se propaga (señal P) desde Ci a Co. El circuito de la derecha se ha obtenido sustituyendo el equivalente de Morgan de la puerta OR de la salida Co. Dado que las puertas de tipo NAND son las más básicas de la tecnología, serán también las más rápidas, y esto ayuda a reducir los retardos en la propagación del acarreo.

Observemos además que G es la función AND de A y B, P es la función OR y también está disponible la función OR exclusiva de A y B en otro nodo del circuito del sumador. Estas son precisamente las funciones

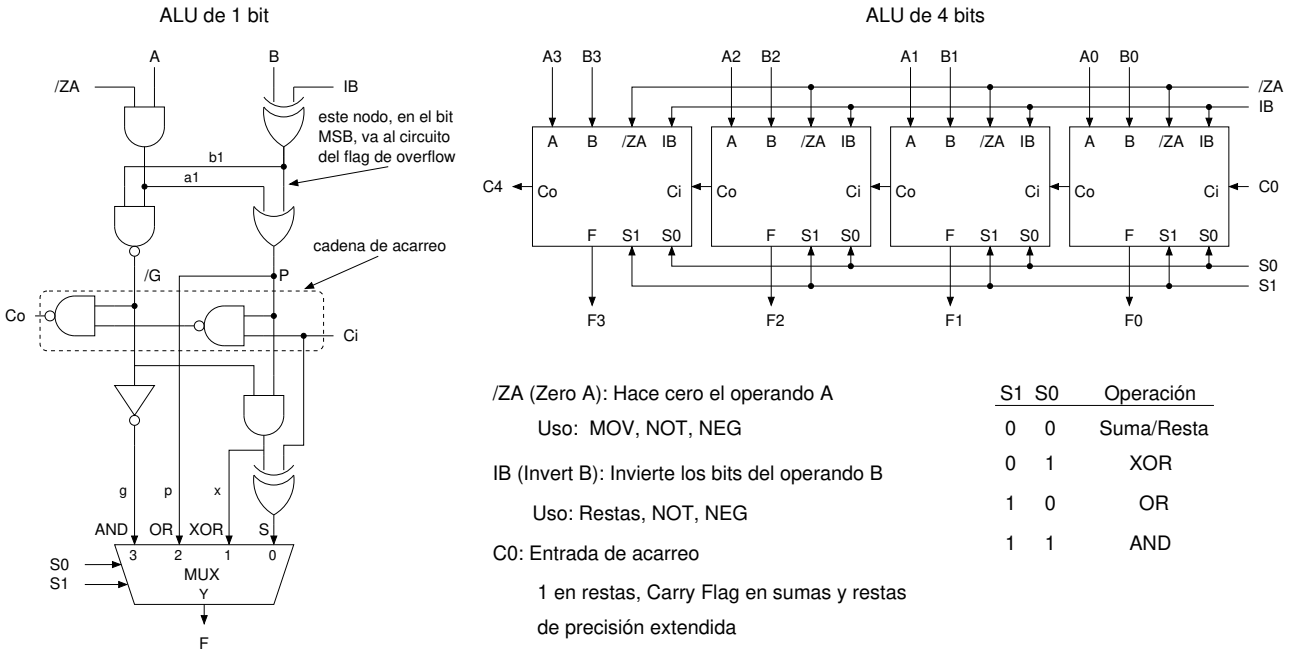


Figure 2: Esquemático de una ALU de 1bit y conexionado en cascada para obtener ALUs del número de bits requerido (ejemplo de 4 bits)

lógicas de la ALU y ya están disponibles en los nodos internos del sumador. La otra función importante que aún nos falta es la de la resta. La resta se obtiene sumando el complemento a dos del operando B, esto es: A menos B = A más NOT(B) más 1. Se requiere, por lo tanto, el poder invertir o no los bits de B. Esto se consigue añadiendo una puerta XOR en la entrada B del sumador. También, en algunas operaciones tenemos sólo un operando, que por mayor versatilidad será el operando B, ya que se podrá invertir. En estos casos podemos forzar que el operando A sea 0 por medio de una puerta AND. De estos razonamientos obtenemos el esquema de la figura 2.

Conectando las salidas de acarreo a las entradas de acarreo de los siguientes bits más significativos podemos obtener una ALU con el ancho de bit deseado. En el esquemático de la derecha de la figura 2 se muestra el conexionado para la obtención de una ALU de 4 bits con acarreo serie. La ALU tiene 5 entradas de control: /ZA, IB, C0, S1 y S0.

La ALU no sólo tiene que generar el valor de salida. A este valor hay que añadir varios bits de tipo indicador, comúnmente llamados “flags”. La salida del acarreo del bit más significativo es uno de estos flags, pero también son muy habituales los flags de cero, Z, que se activa cuando la salida de la ALU vale cero, de negativo, N, que se activa cuando el resultado es un número negativo en la lógica de complemento a dos, y el flag de desbordamiento, V (oVerflow), que se activa cuando en una suma o resta de números con signo hay un cambio de signo inesperado en el resultado. En la lógica de complemento a dos el bit más significativo del dato indica el signo, y un valor 1 significa que del dato es negativo. Por lo tanto el flag de negativo es directamente el bit más significativo del resultado de la ALU. De esta misma consideración se obtiene la lógica del flag V mostrada en la figura 3, pues en una suma hay un desbordamiento si al sumar dos datos positivos obtenemos un resultado negativo, o al sumar dos datos negativos el resultado es positivo.

Por último hay que destacar que la ALU que hemos visto no es capaz de realizar operaciones de desplazamiento o rotación de bits a la derecha. Los desplazamientos a la izquierda sí son posibles ya que equivalen a multiplicar

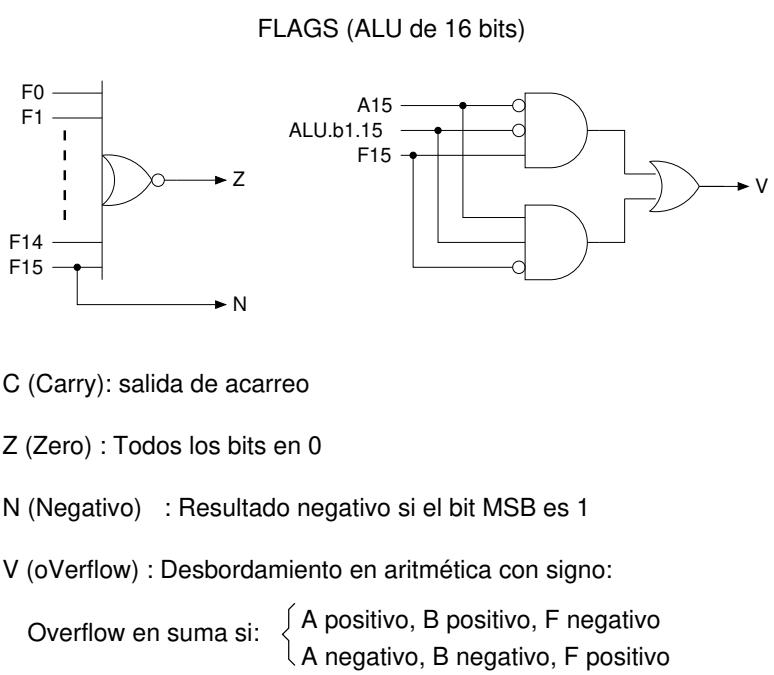
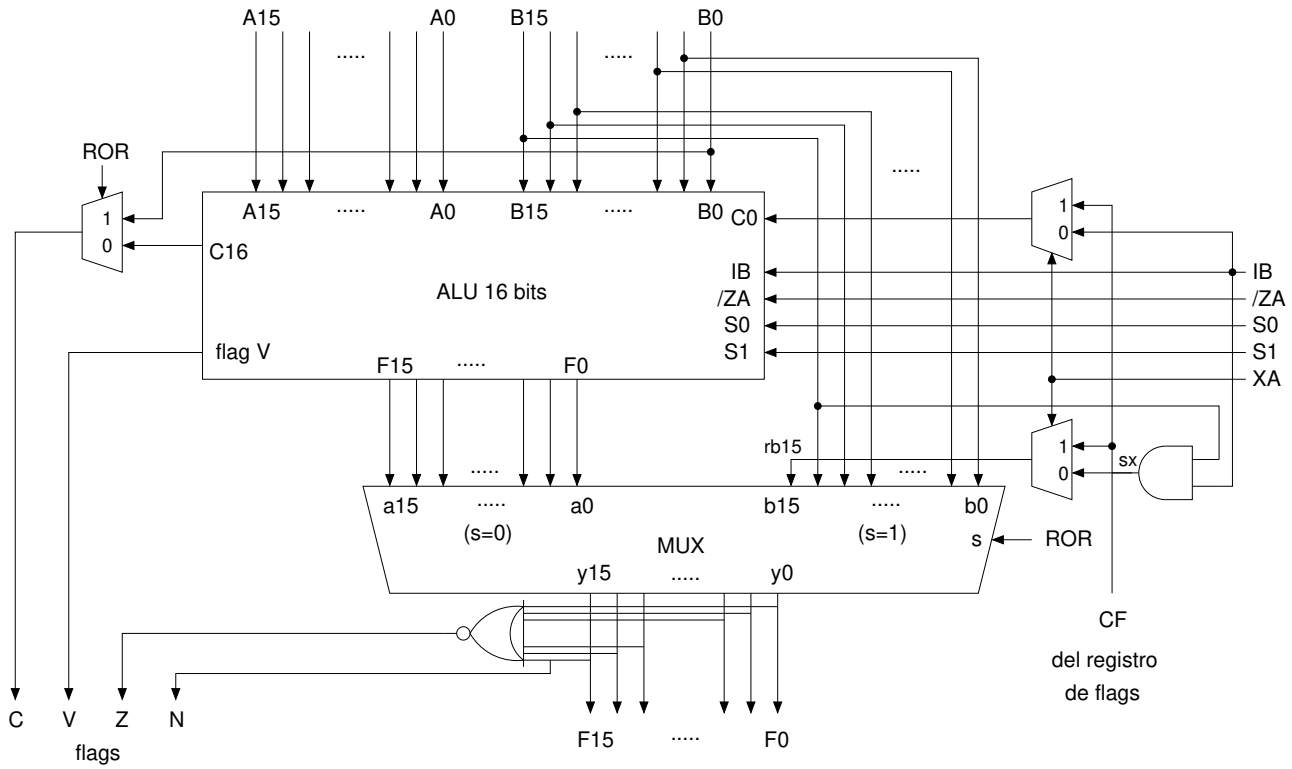


Figure 3: Flags de salida de la ALU

el dato por dos o, a lo que es equivalente, a sumarlo consigo mismo. Los desplazamientos a derechas pueden ser de tipo lógico, cuando se introduce un cero el el bit MSB, o de tipo aritmético, cuando se copia el bit MSB del dato de entrada en el bit MSB del de salida, lo que conserva el signo del dato en la lógica de complemento a dos. También debería ser posible incluir el valor del flag de acarreo tanto en las sumas y restas como en los desplazamientos para poder realizar cálculos con variables de más bits que los que caben en la ALU (aritmética extendida). En el circuito de la figura 4 se han añadido estas funciones en la ALU:

Por último hay que destacar que aunque la representación circuital de la ALU es bastante engorrosa, su descripción en lenguaje Verilog puede resultar mucho más simple pues se puede describir basándonos en su comportamiento en lugar de sus detalles internos. Esto deja para la herramienta de síntesis el trabajo de encontrar el circuito más simple que realiza la función descrita. En el caso de síntesis para FPGAs también de este modo se facilita el uso de la lógica rápida de acarreo de la que disponen estos circuitos. A continuación se incluye el listado Verilog de la ALU descrita:

```
//------------------------------------------------------------------
// ALU
//------------------------------------------------------------------
module ALU (
  output [15:0]y,     // Salida de resultado
  output co,          // Salida de acarreo
  output v,           // Salida de overflow
  output z,           // Salida de cero
  output n,           // Salida de negativo
  input [15:0]a,      // Entrada de operando A
  input [15:0]b,      // Entrada de operando B
  input [1:0]op,      // Operación (0: suma, 1: XOR, 2: OR, 3: AND)
```



XA : Aritmética de precisión extendida (incluye el flag de acarreo)

ROR : Rotación o desplazamiento a derechas, controlado por XA e IB

Figure 4: ALU a la que hemos añadido las funciones de desplazamiento a la derecha.

```
  input cf,          // Entrada de acarreo
  input zab,         // Forzar operando A=0 si 0
  input ib,          // Invertir bits de operando B si 1
  input ror,         // Rotar a derecha si 1
  input xa           // Operaciones con acarreo extendio (ADC,...)
);
wire c0;             // acarreo de entrada
assign c0 = xa ? cf : ib;
// Datos de entrada a sumador
wire [15:0]sa;
wire [15:0]sb;
assign sa= zab ? a  : 16'b0;
assign sb= ib  ? ~b : b;
// Operación ALU interna
reg [15:0]f; // No realmente registros
reg c15;
always@*
  case (op)
     0 : {c15,f} = sa+sb+c0;          // SUMA
     1 : {c15,f} = {1'bx,sa ^ sb};    // XOR
     2 : {c15,f} = {1'bx,sa | sb};    // OR
     3 : {c15,f} = {1'bx,sa & sb};    // AND
  endcase
```

```
// Flag de overflow
assign v = ((~sb[15])&(~sa[15])&f[15]) | (sb[15]&sa[15]&(~f[15]));
// Rotaciones
wire rb15;   // Bit para desplazar a la posición MSB del resultado
assign rb15 = xa ? cf : b[15]&ib;
assign y = ror ? {rb15,b[15:1]} : f;
assign co = ror ? b[0] : c15;
// Flags Z, N
assign z = (y==0);
assign n = y[15];
endmodule
```

Por supuesto, sigue siendo perfectamente posible hacer una descripción de la ALU detallando hasta la última puerta lógica. Es más trabajoso y el resultado final puede ser peor al no identificar la herramienta de síntesis la cadena de acarreo y no dar uso a la lógica rápida que las FPGAs tienen dedicada explicitamente a ello. A título comparativo se ha diseñado la ALU de 1 bit de la figura 2:

```
// ALU de 1 bit
module ALUslice (
    input a,        // entrada A
    input b,        // entrada B
    input ci,       // entrada de acarreo
    input za,       // zero A si 0
    input ib,       // invierte B si 1
    input [1:0]op,  // op: 00: suma, 01: XOR, 10: OR, 11: AND
    output f,       // salida de dato
    output b1,      // copia de la entrada al sumador (para flag overflow)
    output co       // salida de acarreo
);
// nodos internos
wire a1,ng,p,g,x,s;
assign a1=a&za;
assign b1=b^ib;
assign ng=~(a1&b1);
assign g=~ng;
assign p=(a1|b1);
assign co=~((~(ci&p))&ng);
assign x=ng&p;
assign s=x^ci;
// Multiplexor de salida:
assign f=(s&(~op[1])&(~op[0])) | (x&(~op[1])&( op[0])) |
         (p&( op[1])&(~op[0])) | (g&( op[1])&( op[0]));
endmodule
```

Esta ALU de 1 bit se ha instanciado 16 veces en la ALU de la CPU:

```
// Instancias de los bits de la ALU
wire [15:0]f;
```

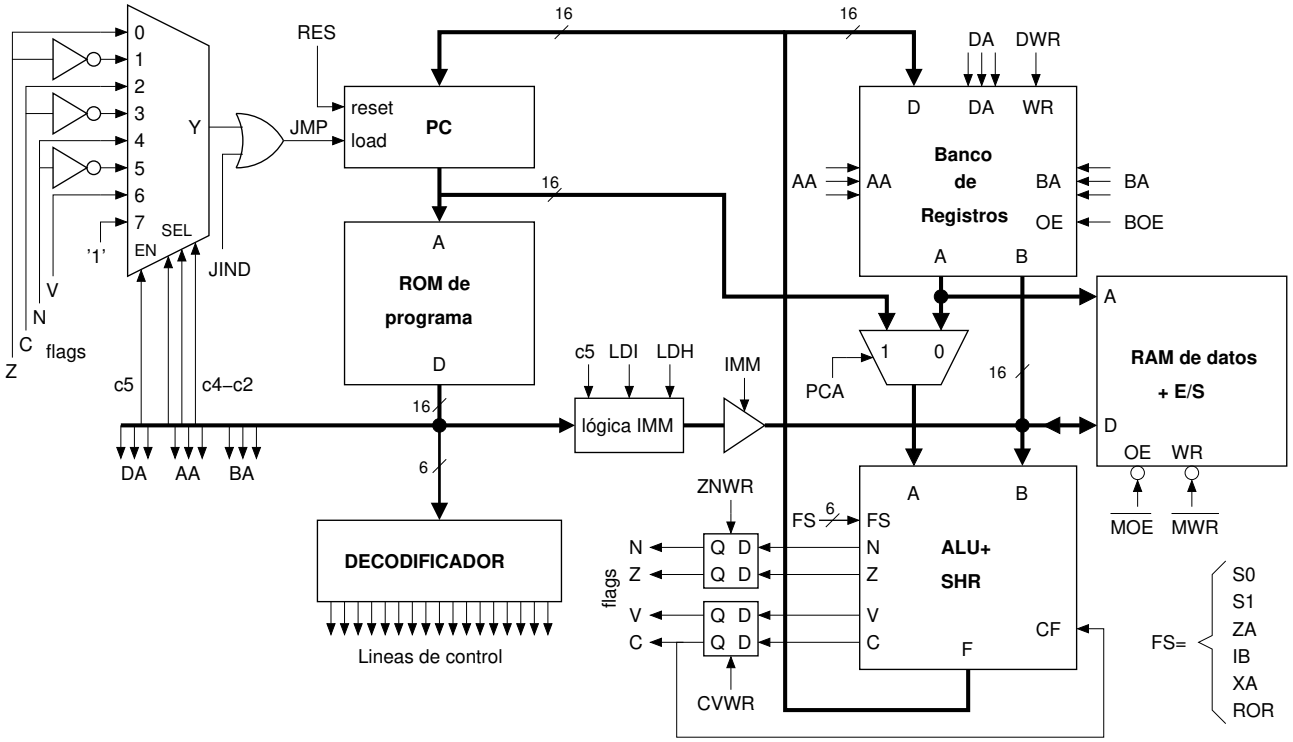


Figure 5: Esquema de la CPU

```
wire [15:1]cy;
ALUslice sl0  (.a(a[0]), .b(b[0]), .ci(c0),    .za(zab),
    .ib(ib),.op(op),.f(f[0]),.co(cy[1]));
ALUslice sl1  (.a(a[1]), .b(b[1]), .ci(cy[1]), .za(zab),
    .ib(ib),.op(op),.f(f[1]),.co(cy[2]));
....
ALUslice sl14 (.a(a[14]),.b(b[14]),.ci(cy[14]),.za(zab),
    .ib(ib),.op(op),.f(f[14]),.co(cy[15]));
ALUslice sl15 (.a(a[15]),.b(b[15]),.ci(cy[15]),.za(zab),
    .ib(ib),.op(op),.f(f[15]),.b1(sb15),.co(c15));
```

La CPU con la ALU detallada a nivel de puertas lógicas se ha sintetizado y ha funcionado correctamente, aunque ha ocupado 11 celdas lógicas más que la versión original y la frecuencia máxima estimada de la CPU ha bajado de 58MHz a 47MHz. Indagando en los listados de la herramienta de síntesis también observamos que ahora tenemos 16 bloques SB_CARRY menos.

Este resultado viene a decirnos que si nuestro diseño va a acabar en una FPGA puede ser contraproducente hacer una descripción del hardware a un nivel demasiado bajo.

3 Primer diseño: CPU Harvard de 1 ciclo por instrucción

En la figura 5 tenemos el esquema de la CPU de ciclo sencillo y arquitectura Harvard propuesta, donde vemos que tenemos memorias separadas para los programas y para los datos. La memoria de programa contiene un máximo de 64K palabras de 16 bits donde cada una de ellas es el código de operación de una instrucción. La memoria de datos tiene un tamaño máximo de 64K palabras de 16 bits, aunque en la práctica podría ser bastante más pequeña dado que habitualmente en los programas se necesita más memoria para el código que para los datos. Además, el acceso a los periféricos también se tiene en este espacio de memoria (memory mapped I/O), de modo que un rango de las direcciones de la memoria de datos estará reservado para los registros de los

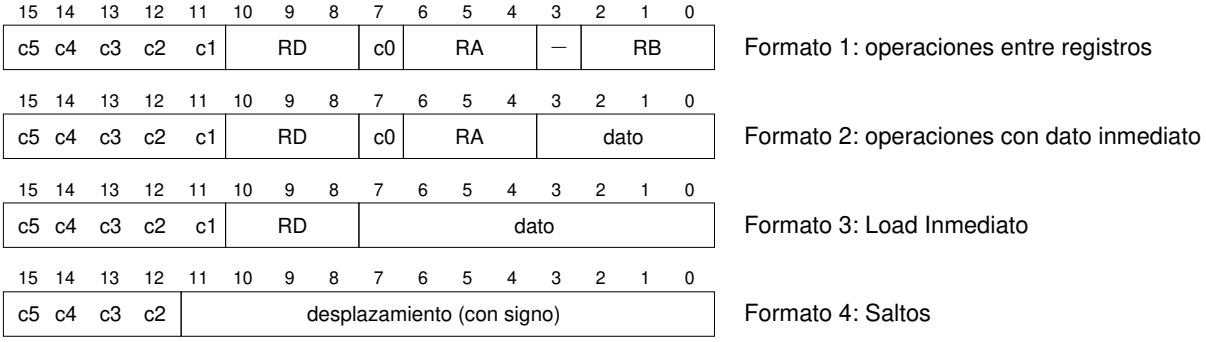


Figure 6: Formatos de los códigos de operación

periféricos.

El "datapath" consta de un banco de 8 registros de 16 bits, la ALU descrita anteriormente, unos registros para almacenar los flags de la ALU, la memoria de datos, y un bloque de lógica destinada a los operandos inmediatos (constantes) de ciertas instrucciones. Los registros del banco pueden leerse en dos buses de salida independientes: bus A y bus B. El registro cuyo contenido se presenta en cada una de las salidas se selecciona mediante las líneas de dirección AA y BA, de 3 bits cada una. La salida del bus B se puede poner en alta impedancia dejando en bajo la señal BOE. Esto es necesario cuando en el bus B se quiere poner un dato procedente de la RAM de datos (instrucción LD) o de la memoria de programa (operandos inmediatos). En estos casos se han de habilitar los "buffer" triestado correspondientes: MOE: memoria de datos, IMM: operando inmediato. También es posible leer el valor del contador de programa en lugar de la salida A del banco de registros mediante la activación de la línea PCA. Esto se hará en las instrucciones de salto relativo y en la instrucción ADPC (sumar PC).

En la ALU la operación a realizar se indica mediante los 6 bits de FS: S0, S1, ZA, IB, XA y ROR, y el resultado se puede escribir en el registro del banco indicado por DA si se activa la señal DWR. También es posible escribir el resultado de la operación en el contador de programa, cosa que haremos en las instrucciones de salto. Los flags de condición se pueden escribir en el registro de flags en grupos de dos: por una parte los flags C y V y por otra los flags Z y N. No todas las instrucciones escriben los flags, y algunas escriben los flags Z y N, pero no los C y V. El valor de los flags se tiene en cuenta en los saltos condicionales.

El registro PC se puede cargar con una dirección de programa de 16 bits si se activa su entrada "load". Esto da lugar a un salto en la ejecución del programa. La dirección que se carga en PC se obtiene en la ALU a partir del contenido del propio PC, al que se suma un desplazamiento inmediato, o a partir de uno de los registros del banco. La lógica de los saltos se ha resuelto con un multiplexor, controlado por los bits más significativos del código de operación, y una puerta OR para los saltos indirectos (instrucción JIND, no condicional).

3.1 Formato de los códigos de operación

Los códigos de operación de las instrucciones pertenecen a uno de los 4 formatos posibles que se muestran en la figura 6. El tipo de instrucción está codificado en los bits "c5-c0", aunque las instrucciones del tipo 3 no utilizan "c0" y las de tipo 4 no utilizan ni "c0" ni "c1". Este último tipo de formato nos permite reservar hasta 12 bits para el desplazamiento de los saltos, con lo que podemos saltar a instrucciones alejadas hasta ± 2048 posiciones de la instrucción actual. El formato 3 sólo se utiliza en la instrucción "load inmediato", LDI, que nos permite cargar constantes de 8 bits en los registros y en LDH que, junto con LDI, permite la carga de constantes de 16 bits. El resto de instrucciones con operando inmediato tienen sólo 4 bits reservados para dicho dato.

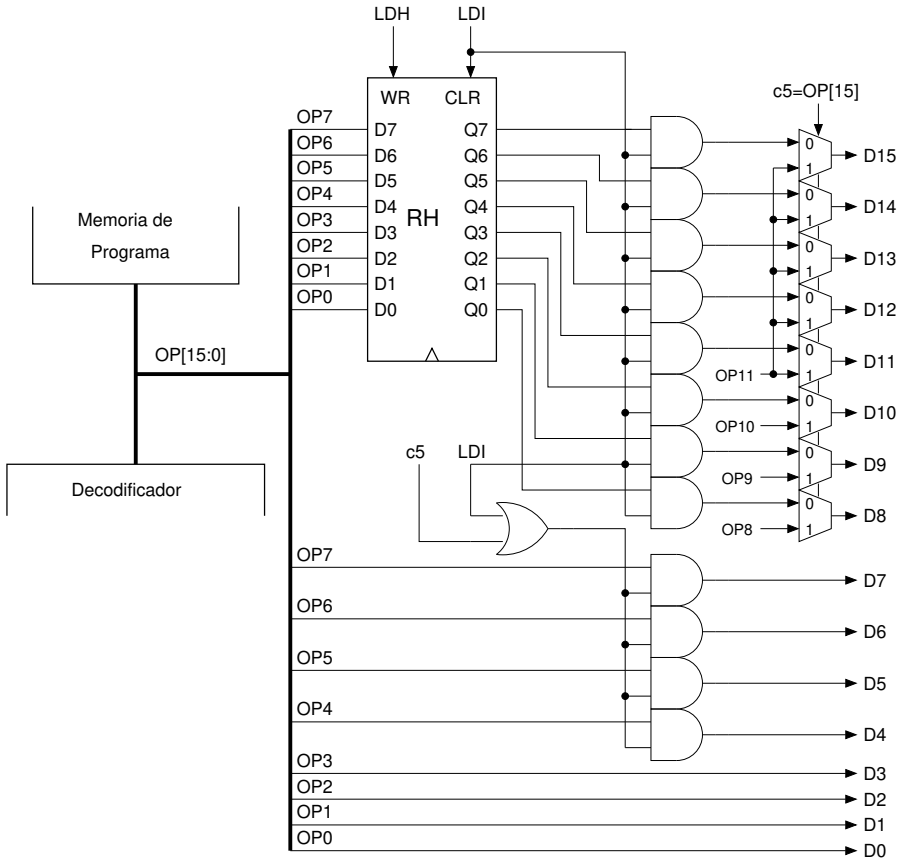


Figure 7: Lógica para la carga de constantes inmediatas.

3.2 Operandos inmediatos

En este tipo de CPU la carga de constantes en los registros podría ser problemática pues no disponemos de suficientes bits en el campo del dato inmediato para representar todas las constantes posibles de 16 bits. Por ello hemos incluido en la lógica de las constantes inmediatas un registro, RH, de 8 bits que almacenará de manera temporal los 8 bits más significativos de dichas constantes (ver figura 7). La instrucción LDH, escribe constantes de 8 bits en RH, y cuando posteriormente se ejecuta la instrucción LDI su contenido aparecerá en los 8 bits más significativos del bus B mientras que el dato inmediato de LDI aparece en los 8 bits menos significativos. Así conseguimos cargar una constante de 16 bits con la ejecución de sólo 2 instrucciones. La instrucción LDI además hace un reset síncrono del registro RH: el registro se carga con cero en el siguiente ciclo de reloj. De esta manera no es necesario cargar RH con cero si las siguientes constantes caben en 8 bits.

Como ejemplo de uso veamos la siguiente secuencia de instrucciones:

```
LDH  0x45    ; RH=0x45
LDI  R0,0x67 ; R0=0x4567, RH=0
LDI  R1,0x12 ; R1=0x0012
```

Por otra parte, las instrucciones de salto tienen constantes de 12 bits con signo. En estos casos hay que extender el signo del desplazamiento hasta 16 bits, lo que se consigue copiando el bit 11 del código de operación en los restantes bits más significativos. En los saltos los 8 bits más significativos provienen directamente del código de operación o de la extensión del signo, pero no de RH, de modo que se han incluido los multiplexores necesarios para seleccionar el dato adecuado en cada caso. El código Verilog equivalente de la lógica de operandos inmediatos sería:

```
//-------------------------------------------------------------
```

```
// Operandos inmediatos
//-------------------------------------------------------------
module IMM(
output [15:0]f,     // Salida de operando inmediato
input [15:0]op,     // Entrada desde reg. de instrucción
input ldi,          // instrucción LDI
input ldh,          // instrucción LDH
input clk
);
reg [7:0]rh;
always @ (posedge clk) rh<= ldi ? 8'h00 : ( ldh ? op[7:0] : rh);
assign f = op[15] ? {op[11],op[11],op[11],op[11],op[11:0]} :
           ( ldi ? {rh,op[7:0]}: {12'h000,op[3:0]});
endmodule
```

3.3 Repertorio de instrucciones

El repertorio de instrucciones se muestra en la figura 8. De todas las combinaciones posibles de los bits "c5-c0" sólo queda un caso sin asignar.

El repertorio de instrucciones puede parecer demasiado escaso en una primera vista. Por ello pasamos a comentar algunos trucos de programación que nos permitirán realizar ciertas operaciones que no están incluidas entre las instrucciones listadas.

No hay instrucciones para mover datos entre registros. Sin embargo, esta operación se puede realizar sin problema mediante instrucciones aritméticas o lógicas, como en los siguientes ejemplos:

```
ADDI R1,R0,0    ; R1=R0, Cflag=0, Zflag=(R0==0), Nflag=R0.15
SUBI R1,R0,0    ; R1=R0, Cflag=1, Zflag=(R0==0), Nflag=R0.15
ORI  R1,R0,0    ; R1=R0,          Zflag=(R0==0), Nflag=R0.15
OR   R1,R0,R0   ; R1=R0,          Zflag=(R0==0), Nflag=R0.15
AND  R1,R0,R0   ; R1=R0,          Zflag=(R0==0), Nflag=R0.15
```

Un posible inconveniente de estas instrucciones es que alteran el contenido de los flags, aunque en ocasiones podríamos explotar este efecto en nuestro beneficio.

El contenido de dos registros se puede intercambiar sin modificar ningún otro registro mediante una secuencia de tres instrucciones XOR:

```
XOR R1,R1,R0    ; Intercambio de R1 y R0
XOR R0,R1,R0
XOR R1,R1,R0
```

Tampoco tenemos instrucciones específicas para incrementar o decrementar registros, pero estas son innecesarias al disponer de instrucciones de suma y resta con datos inmediatos: basta que el dato inmediato sea 1.

No hay instrucciones de desplazamiento ni de rotación a izquierdas dado que se puede conseguir el mismo resultado sumando el contenido de un registro consigo mismo:

```
ADD R1,R0,R0    ; R1 = R0*2 = R0<<1 -> R1=SHL(R0)
ADC R1,R0,R0    ; R1 = R0*2+Cflag -> R1=ROL(R0)
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	mnemónico	flags				operación
0	0	0	0	0	RD			0	RA			–	RB			ADD	C	V	N	Z	RD= RA+RB
0	0	0	0	0	RD			1	RA			dato				ADDI	C	V	N	Z	RD= RA+dato
0	0	0	0	1	RD			0	RA			–	RB			ADC	C	V	N	Z	RD=RA+RB+Cflag
0	0	0	0	1	RD			1	RA			dato				ADCI	C	V	N	Z	RD=RA+dato+Cflag
0	0	0	1	0	RD			0	RA			–	RB			SUB	C	V	N	Z	RD=RA–RB
0	0	0	1	0	RD			1	RA			dato				SUBI	C	V	N	Z	RD=RA–dato
0	0	0	1	1	RD			0	RA			–	RB			SBC	C	V	N	Z	RD=RA–RB–(~Cflag)
0	0	0	1	1	RD			1	RA			dato				SBCI	C	V	N	Z	RD=RA–dato–(~Cflag)
0	0	1	0	0	–	–	–	0	RA			–	RB			CMP	C	V	N	Z	RA–RB
0	0	1	0	0	–	–	–	1	RA			dato				CMPI	C	V	N	Z	RA–dato
0	0	1	0	1	RD			0	RA			–	RB			AND	–	–	N	Z	RD=RA&RB
0	0	1	0	1	RD			1	RA			dato				ANDI	–	–	N	Z	RD=RA&dato
0	0	1	1	0	–	–	–	0	RA			–	RB			TST	–	–	N	Z	RA&RB
0	0	1	1	0	–	–	–	1	RA			dato				TSTI	–	–	N	Z	RA&dato
0	0	1	1	1	RD			0	RA			–	RB			OR	–	–	N	Z	RD=RA|RB
0	0	1	1	1	RD			1	RA			dato				ORI	–	–	N	Z	RD=RA|dato
0	1	0	0	0	RD			0	RA			–	RB			XOR	–	–	N	Z	RD=RA^RB
0	1	0	0	0	RD			1	RA			dato				XORI	–	–	N	Z	RD=RA^dato
0	1	0	0	1	RD			0	–	–	–	–	RB			NOT	–	–	N	Z	RD=~Rb
0	1	0	0	1	RD			1	–	–	–	–	RB			NEG	–	–	N	Z	RD=–RB
0	1	0	1	0	RD			0	–	–	–	–	RB			SHR	C	?	N	Z	RD=RB/2, Cflag=RB.0
0	1	0	1	0	RD			1	–	–	–	–	RB			SHRA	C	?	N	Z	RD=RB/2, Cflag=RB.0 (con signo)
0	1	0	1	1	RD			0	–	–	–	–	RB			ROR	C	?	N	Z	RD=(RB>>1)|(Cflag<<15), Cflag=RB.0
0	1	0	1	1				1								ILEG					
0	1	1	0	0	RD			0	RA			–	–	–	–	LD	–	–	N	Z	RD=Mem(RA)
0	1	1	0	0	–	–	–	1	RA			–	RB			ST	–	–	–	–	Mem(RA)=RB
0	1	1	0	1	RD			0	–	–	–	dato				ADPC	–	–	–	–	RD=PC+dato
0	1	1	0	1	–	–	–	1	–	–	–	–	RB			JIND	–	–	–	–	PC=RB
0	1	1	1	0	–	–	–	dato								LDH	–	–	–	–	RH (temporal)=dato
0	1	1	1	1	RD			dato								LDI	–	–	–	–	RD=(RH<<8)|dato, RH=0
1	0	0	0	desplazamiento con signo												JZ	–	–	–	–	salto si Zflag=1
1	0	0	1	desplazamiento con signo												JNZ	–	–	–	–	salto si Zflag=0
1	0	1	0	desplazamiento con signo												JC	–	–	–	–	salto si Cflag=1
1	0	1	1	desplazamiento con signo												JNC	–	–	–	–	salto si Cflag=0
1	1	0	0	desplazamiento con signo												JMI	–	–	–	–	salto si Nflag=1 (negativo)
1	1	0	1	desplazamiento con signo												JPL	–	–	–	–	salto si Nflag=0 (positivo)
1	1	1	0	desplazamiento con signo												JV	–	–	–	–	salto si Vflag=1 (overflow)
1	1	1	1	desplazamiento con signo												JR	–	–	–	–	salto incondicional

Figure 8: Tabla con los códigos de operación de las instrucciones y lista de flags afectados.

Para hacer una rotación a la izquierda sin incluir el acarreo (D0 pasa a valer lo que antes era D15 en lugar del acarreo) podemos ejecutar la siguiente secuencia de dos instrucciones:

```
ADD R6,R0,R0    ; Cflag=R0.15, R6 se ignora
ADC R1,R0,R0    ; R1 = ROLnc(R0), Cflag=R0.15
```

Para hacer una rotación similar a derechas también recurrimos a una secuencia de dos instrucciones, en este caso ROR:

```
ROR R6,R0       ; Cflag=R0.0, R6 se ignora
ROR R1,R0       ; R1 = RORnc(R0), Cflag=R0.0
```

No hay registro puntero de pila ni instrucciones relacionadas. Sin embargo resulta fácil emular las funciones de una pila por programa. En el siguiente ejemplo el registro R7 hace las funciones de puntero de pila:

```
SUBI R7,R7,1    ; PUSH R0       LD   R0,(R7)    ; POP R0
ST   (R7),R0    ;               ADDI R7,R7,1    ;
```

Las llamadas a subrutinas son un poco más complicadas de realizar. El punto crucial es la capacidad de retornar al programa principal al finalizar la subrutina. Esto se puede conseguir gracias a la instrucción de salto indirecto. En el siguiente ejemplo utilizaremos el registro R6 para almacenar la dirección de retorno, y calcularemos dicha dirección antes de saltar a la subrutina partiendo del valor del PC mediante la instrucción ADPC, que nos permite sumar un dato inmediato al valor del PC y almacenar el resultado en un registro:

```
; Llamada a subrutina
PC   -> ADPC R6,2    ; R6=PC+2 (dirección de retorno)
PC+1 -> JR   ruti1
PC+2 -> ...          ; dirección de retorno
```

La subrutina se ejecuta sin alterar el valor de R6 y retorna con la instrucción JIND:

```
ruti1: ....      ; código de la subrutina
       JIND  R6  ; retorno
```

Resulta obvio que una subrutina no podría llamar a otra subrutina dado que ello supondría perder el valor de la dirección de retorno. Por ello, en estos casos, lo que hacemos es guardar la dirección de retorno en una pila:

```
ruti1: SUBI R7,R7,1 ; Push R6
       ST   (R7),R6
       ...
       ADPC R6,2    ; Llamada a otra subrutina
       JR   ruti2
       ...
       LD   R6,(R7) ; Pop R6
       ADDI R7,R7,1
       JIND R6       ; retorno
```

Todos los saltos condicionales son relativos, y también la instrucción de salto incondicional, JR. Para saltar a una posición absoluta de la memoria de programa habrá que cargar dicha dirección en un registro y saltar luego con JIND:

```
LDH  0x10     ; reg temporal=0x10
LDI  R6,0x43  ; R6=0x1043 (dirección de salto)
JIND R6
```

4 Segundo diseño: CPU Von Neumann con pipeline

La CPU de un ciclo que acabamos de describir tiene un par de inconvenientes destacables: Baja frecuencia de reloj y dificultad con el manejo de constantes. El primer problema se debe a la suma de todos los retardos de los diversos bloques que componen la CPU, de modo que el periodo mínimo del reloj será:

$$T_{CLK,min} = t_{pd,PC} + t_{pd,ROM} + t_{pd,DECOD} + t_{pd,REG} + t_{pd,ALU} + t_{setup,REG}$$

En el caso de la instrucción LD también hay que incluir el retardo de propagación de la RAM de datos, aunque en este caso el retardo de la ALU va a ser menor. Los retardos mayores van a ser los de la memoria ROM y los de la ALU, especialmente en las operaciones aritméticas en las que el acarreo se propaga en serie. Si suponemos unos retardos de 2ns para la puerta más simple, y 50ns para la ROM, podemos estimar estos tiempos como: $t_{pd,PC} = 2ns$, $t_{pd,DECOD} = 10ns$ (5 niveles de puertas), $t_{pd,ALU} = 58ns$ (16*3ns en acarreo + 5 niveles de puertas) y $t_{setup,REG} = 2ns$, lo que nos da un periodo de reloj mínimo de 132ns o una frecuencia de reloj máxima de 7.57MHz, que es un tanto baja ya que la ROM por si sola permitiría 20 millones de lecturas por segundo.

Para mejorar la velocidad de ejecución podemos recurrir a la técnica del pipeline, que consiste en dividir el proceso de ejecución en pasos simples y realizar todos los pasos en paralelo, si bien cada uno de ellos correspondería a una instrucción distinta.

El problema del manejo de constantes queda patente con un programa tan simple como el típico "Hello World". En este caso el texto en ASCII se tiene que almacenar en la memoria de programa como instrucciones LDI, lo que hace muy complicado manejar estas constantes como tablas de datos, además de suponer un uso muy poco eficiente de la memoria de programa mientras que por otra parte la memoria de datos está infrautilizada. Propondremos por lo tanto un cambio a una arquitectura de tipo Von Neumann con un único espacio de direcciones, tanto para instrucciones como para datos. Esto supondrá una penalización en los tiempos de ejecución de las instrucciones LD y ST, que pasarán a ser de dos ciclos, pero afortunadamente estas instrucciones no son demasiado frecuentes en el código.

En la figura 9 se muestra el diagrama de bloques de la nueva CPU propuesta, en la que casi todos los elementos son los mismos de la CPU original, incluyendo la lógica de decodificación de los saltos que, por brevedad, ahora está incluida dentro del bloque DECODIFICADOR. En este nuevo diseño también hemos tenido en cuenta las limitaciones de las FPGAs en las que pretendemos sintetizar la CPU. En particular hemos tenido que eliminar todas las salidas triestado y por ello hemos sustituido los buses triestado por multiplexores.

Sin embargo, la principal diferencia con el diseño original es la presencia de un registro adicional, IR, entre la memoria y el decodificador. Este registro es el que hace posible el pipeline ya que, con la excepción de las instrucciones LD y ST, se pueden leer códigos de operación de la memoria a IR a la vez que se decodifican y ejecutan los que estaban almacenados del ciclo de reloj anterior. Esto va a permitir reducir el periodo de reloj pues ahora se pueden solapar algunos retardos en lugar de sumarse. El análisis de los retardos nos va a dar dos periodos mínimos de reloj, uno para la unidad de búsqueda (Fetch) y otro para la de ejecución (Exec), de los que deberemos quedarnos con el más grande:

$$T_{CLK,FETCH} = t_{pd,PC} + t_{pd,MEM} + t_{setup,IR} = 54ns$$

$$T_{CLK,EXEC} = t_{pd,IR} + t_{pd,DECOD} + t_{pd,REG} + t_{pd,ALU} + t_{setup,REG} = 74ns$$

El periodo mínimo de reloj queda por lo tanto en 74ns y la frecuencia de reloj máxima en 13.5MHz. vemos por lo tanto que la estrategia del pipeline nos permite casi duplicar la frecuencia de reloj respecto del diseño

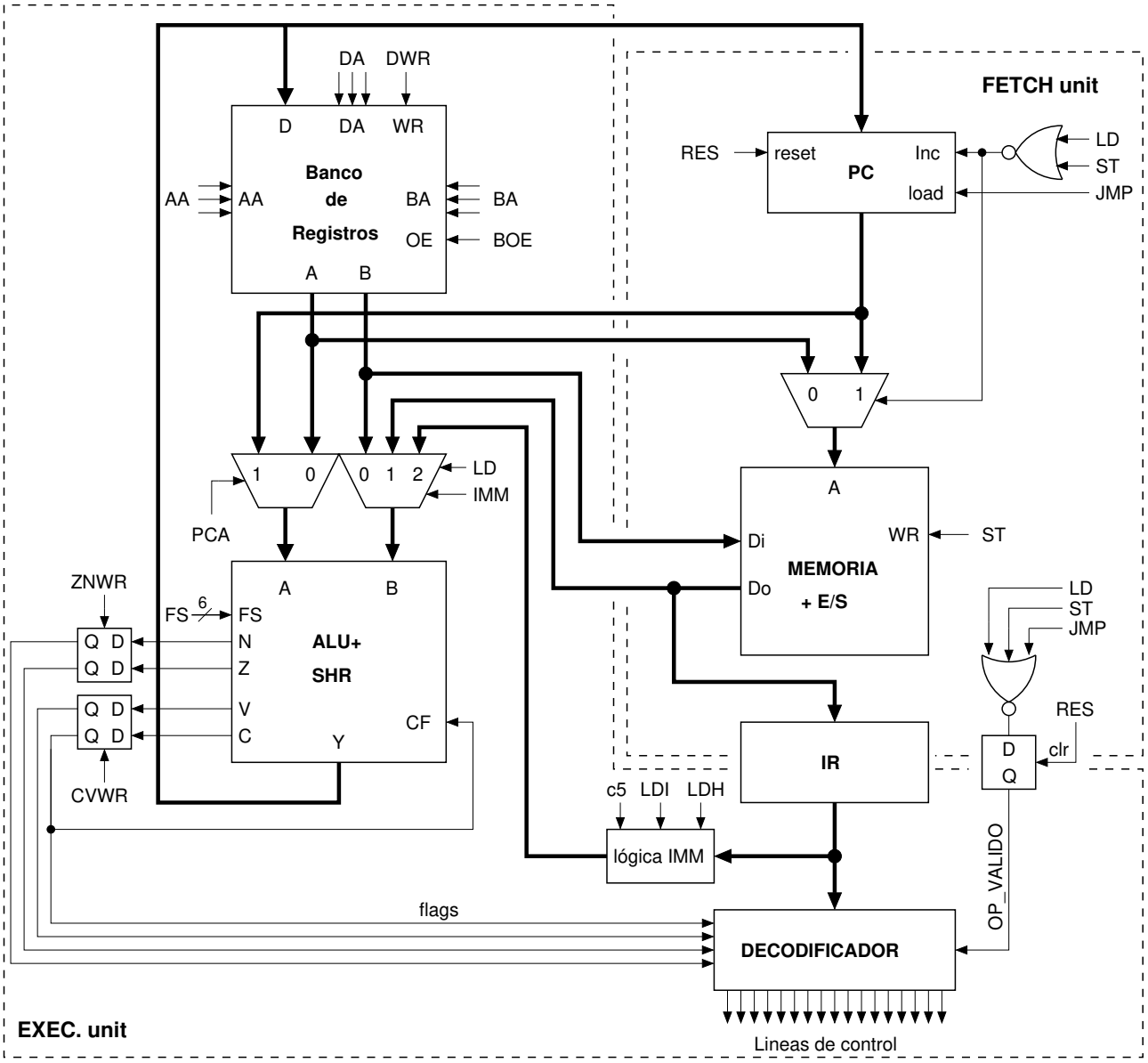


Figure 9: Diagrama de bloques de la CPU mejorada con pipeline en versión para síntesis en FPGA (sin triestados).

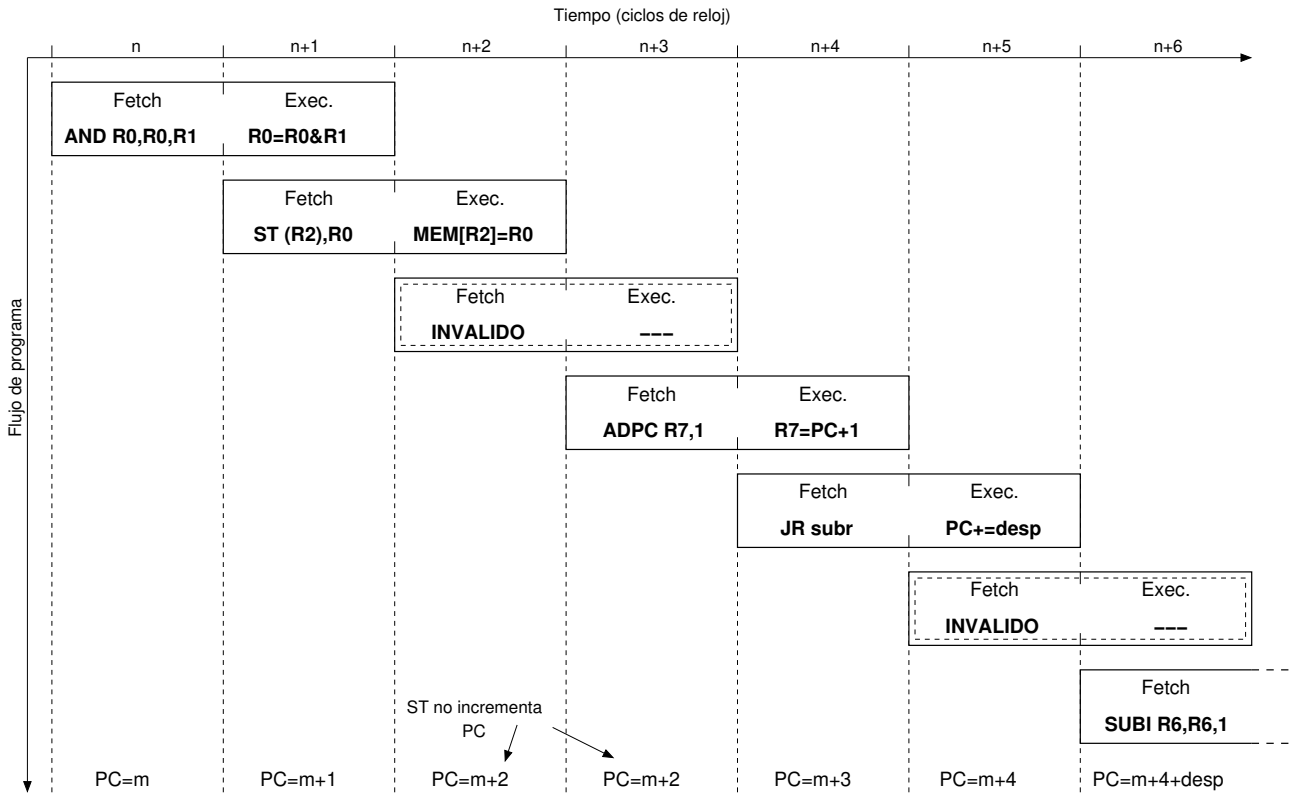


Figure 10: Flujo de ejecución de un segmento de código donde se destacan los dos ciclos empleados por cada instrucción, cómo las fases de ejecución y de búsqueda de código de operación se solapan, y cómo ciertas instrucciones (ST, JR) dan lugar a la carga de códigos inválidos que hacen que sus tiempos efectivos de ejecución sean de dos ciclos en lugar de uno.

original, y eso va a compensar con creces el mayor tiempo de ejecución de algunas instrucciones, mientras que el aumento en la complejidad de la CPU se reduce a poco más que la inclusión del registro IR.

El registro IR se puede considerar constituido por 17 bits: 16 bits que almacenen el código de operación leído de la memoria más un bit adicional que indica si los 16 bits anteriores son válidos o no. Este bit forma parte de la decodificación de las instrucciones y cuando su valor es cero se inhiben todas las señales de escritura que salen del decodificador (DWR, ST, JMP, NZWR, y CVWR) de modo que la instrucción ejecutada equivale a un NOP. También se inhibe LD para evitar quedarse bloqueados con LD activa. El bit de código de operación válido se desactiva en los siguientes casos:

- Ejecución de LD o ST, ya que lo que se escribe en IR no es un código de operación sino un dato transferido desde o hacia la memoria.
- Ejecución de instrucciones de salto: Cuando se ejecuta la instrucción de salto el PC está apuntando a la siguiente posición de la memoria y se lee un código de operación incorrecto pues se debería cargar el de la posición destino del salto.
- Reset, ya que el contenido de IR puede ser cualquiera y aún no se ha leído el primer código de operación.

Obsérvese que la carga y ejecución de un código de operación inválido equivale al vaciamiento del pipeline y a la pérdida de un ciclo de reloj sin que la unidad de ejecución realice ninguna operación útil.

Durante la ejecución de las instrucciones LD y ST también se inhibe el incremento del contador de programa para evitar saltarse una instrucción. Todo esto se traduce en que las instrucciones de salto, junto con las de LD y ST, tardan en ejecutarse dos ciclos efectivos de reloj mientras que el resto se ejecuta en un sólo ciclo. Esto queda de manifiesto en el ejemplo de la figura 10. Los saltos condicionales tardan en ejecutarse un ciclo de reloj cuando no se cumple la condición y dos ciclos de reloj cuando se cumple y se salta.

4.1 Instrucciones de la CPU con pipeline

El repertorio de instrucciones de esta variante de CPU es el mismo de la CPU original pero sin embargo los programas escritos para una CPU no corren en la otra. Ello se debe al hecho de que el contador de programa apuntaba a la instrucción que se estaba ejecutando en el diseño original mientras que en la CPU modificada el PC va una posición de memoria por delante. Este comportamiento afecta a las instrucciones de salto relativo y ADPC pues el valor del PC interviene en el cálculo de la dirección de destino. La instrucción JIND, en cambio, no se ve afectada. Por ello, la forma de llamar a una subrutina en la CPU nueva será:

```
ADPC  R6,1
JR    subrutina
; dirección de retorno
```

La instrucción ADPC ahora suma 1 al PC en lugar de 2 dado que el PC ya está apuntando una posición por delante cuando se ejecuta. Obsérvese además que el desplazamiento del código de operación del salto, JR, también es distinto pero de ese detalle se encargará el programa ensamblador, al que deberemos indicar el tipo de CPU para el que se va a generar el código. El retorno de subrutina se haría de la forma habitual con "JIND R6".

En cuanto al rendimiento de la nueva CPU, la simulación de algunos programas nos permite estimar que ésta emplea alrededor de un 25% más de ciclos de reloj que la CPU original para ejecutar el mismo código. Esto se debe en mayor medida a las instrucciones de salto que a las LD y ST. Sin embargo, si tenemos en cuenta que el reloj va a ser más rápido, el tiempo de ejecución se reduce al 70% de la CPU original.

La nueva CPU es de tipo Von Neumann y ello nos va a permitir la lectura de constantes de la misma memoria que las instrucciones. Hay que notar que los siguientes ejemplos de código no funcionan correctamente con la CPU original pues la memoria de instrucciones y la de datos eran distintas. Como primer ejemplo se incluye el código de una subrutina que accede a una tabla de constantes que puede estar ubicada en cualquier parte de la memoria ya que su dirección base se obtiene del contador de programa:

```
                ; R0: índice del dato en la tabla
ADPC  R1,3      ; R1=dirección de la base de la tabla
ADD   R1,R0,R1  ; R1=dirección del elemento de la tabla
LD    R0,(R1)   ; Lectura del dato
JIND  R6        ; Retorno de subrutina
word 0x1234     ; Base de la tabla (índice #0)
word 0x5678     ; (índice #1)
...
```

También se puede utilizar este mismo mecanismo para implementar una bifurcación múltiple con el destino del salto seleccionado mediante R0:

```
ADPC  R1,3      ; R1=dirección base de la tabla de saltos
ADD   R1,R0,R1  ; R1=dirección del elemento de la tabla
LD    R0,(R1)   ; R0=dirección destino del salto
JIND  R0        ; Bifurcación múltiple
word destino_0  ; Base de la tabla de saltos (índice #0)
word destino_1  ; (índice #1)
...
```

4.2 Diseño de los bloques funcionales en Verilog

Esta variante de la CPU se ha diseñado completamente en lenguaje Verilog y se ha sintetizado en una FPGA ICE40HX1K de Lattice. Junto con un periférico de tipo UART ha ocupado 766 celdas lógicas de las 1280 disponibles y ha sido capaz de ejecutar algunos programas de prueba con una frecuencia de reloj de 36MHz. Pensamos que el principal factor que nos limita la velocidad es el uso de la memoria RAM síncrona interna de la FPGA pues nos obliga a dividir el ciclo de reloj en dos semiperiodos dejando la mitad de tiempo para los accesos a la RAM. En concreto, la RAM debe tener un dato válido en los flancos de bajada del reloj, mientras que el resto de los registros, PC incluido, cambian en los flancos de subida. El diseño podría modificarse para aprovechar el carácter síncrono de la RAM interna pero en ese caso no sería fácil añadir memoria externa asíncrona.

A continuación detallamos la descripción Verilog de los bloques de la CPU aún no comentados, comenzando por el banco de registros y el contador de programa. Nótese que se asigna a los registros un valor inicial de 0, lo que no tiene coste en una implementación en FPGA pues todos los flip-flops tienen un estado inicial válido, pero ayuda enormemente a evitar estados desconocidos en las simulaciones.

```
//-------------------------------------------------------------------
// Banco de Registros
//-------------------------------------------------------------------
module REGBANK (
output [15:0]a,     // Salida para bus A (ALU, dir memoria)
output [15:0]b,     // Salida para bus B (ALU, datos memoria)
input [15:0]d,      // Entrada de datos al banco
input [2:0]asel,    // Selección de registro para lectura a bus A
input [2:0]bsel,    // Selección de registro para lectura a bus B
input [2:0]dsel,    // Selección de registro para escritura
input wren,         // Habilitación de escritura si 1
input clk
);
wire [7:0]wr;
assign wr[0]=wren & (~dsel[2]) & (~dsel[1]) & (~dsel[0]);
assign wr[1]=wren & (~dsel[2]) & (~dsel[1]) & ( dsel[0]);
assign wr[2]=wren & (~dsel[2]) & ( dsel[1]) & (~dsel[0]);
assign wr[3]=wren & (~dsel[2]) & ( dsel[1]) & ( dsel[0]);
assign wr[4]=wren & ( dsel[2]) & (~dsel[1]) & (~dsel[0]);
assign wr[5]=wren & ( dsel[2]) & (~dsel[1]) & ( dsel[0]);
assign wr[6]=wren & ( dsel[2]) & ( dsel[1]) & (~dsel[0]);
assign wr[7]=wren & ( dsel[2]) & ( dsel[1]) & ( dsel[0]);
// Registros R0-R7
wire [15:0]q0;
wire [15:0]q1;
wire [15:0]q2;
wire [15:0]q3;
wire [15:0]q4;
wire [15:0]q5;
wire [15:0]q6;
wire [15:0]q7;
```

```
// Instancias de registros
REG16 r0(.q(q0),.d(d),.wr(wr[0]),.clk(clk));
REG16 r1(.q(q1),.d(d),.wr(wr[1]),.clk(clk));
REG16 r2(.q(q2),.d(d),.wr(wr[2]),.clk(clk));
REG16 r3(.q(q3),.d(d),.wr(wr[3]),.clk(clk));
REG16 r4(.q(q4),.d(d),.wr(wr[4]),.clk(clk));
REG16 r5(.q(q5),.d(d),.wr(wr[5]),.clk(clk));
REG16 r6(.q(q6),.d(d),.wr(wr[6]),.clk(clk));
REG16 r7(.q(q7),.d(d),.wr(wr[7]),.clk(clk));
// Multiplexores para buses A y B
reg [15:0]a; // No son registros
reg [15:0]b;
always@*
  case (asel)
     0 : a <= q0;
     1 : a <= q1;
     2 : a <= q2;
     3 : a <= q3;
     4 : a <= q4;
     5 : a <= q5;
     6 : a <= q6;
     7 : a <= q7;
  endcase
always@*
  case (bsel)
     0 : b <= q0;
     1 : b <= q1;
     2 : b <= q2;
     3 : b <= q3;
     4 : b <= q4;
     5 : b <= q5;
     6 : b <= q6;
     7 : b <= q7;
  endcase
endmodule
// Registro simple para instanciar en el banco
module REG16 (output [15:0]q, input [15:0]d, input wr, input clk);
reg [15:0]q=0;
always @(posedge clk) q<= wr ? d : q;
endmodule
// Registro Contador de programa
module REGPC (output [15:0]q, input [15:0]d, input resb,
              input load, input inc, input clk);
reg [15:0]q=0;
always @(posedge clk or negedge resb )
if (!resb) q<=16'h0000;
```

```
else q<= load ? d : (inc ? q+1 : q);
endmodule
```

De modo similar se han descrito los registros de flags y el registro de instrucción IR:

```
//-------------------------------------------------------
// Registro de instrucción
//-------------------------------------------------------
module IR (
  output [15:0]q,   // Salida (hacia bloque IMM y decodificación)
  output opval,     // Instrucción válida si 1
  input [15:0]d,    // Entrada (desde bus de datos)
  input ld,         // 1 si instrucción LD
  input st,         // 1 si instrucción LD
  input jmp,        // 1 si instrucciones de salto
  input clk,
  input resb
);
reg [15:0]q;
reg opval=0;
always @(posedge clk) q<= d;
always @(posedge clk or negedge resb)
if (!resb)  opval<=1'b0;
else  opval<= ~(ld | st | jmp);
endmodule
//-------------------------------------------------------
// FLAGS
//-------------------------------------------------------
module FLAGS (
  output qc,    // Salida de flag C
  output qv,    // Salida de flag V
  output qz,    // Salida de flag Z
  output qn,    // Salida de flag N
  input dc,     // Entrada de flag C
  input dv,     // Entrada de flag V
  input dz,     // Entrada de flag Z
  input dn,     // Entrada de flag N
  input wrzn,   // Escribir flags Z y S si 1
  input wrcv,   // Escribir flags C y V si 1
  input clk
);
reg qc,qv,qz,qn;
always @(posedge clk) begin
qc <= wrcv ? dc: qc;
qv <= wrcv ? dv: qv;
qz <= wrzn ? dz: qz;
qn <= wrzn ? dn: qn;
```

```
end
endmodule
```

Aunque sin duda el bloque más complejo es el decodificador que genera las señales de control a partir del código de operación de la instrucción. Se trata de un circuito puramente combinacional en el que todo el repertorio de instrucciones de la CPU está contenido:

```
//-------------------------------------------------------------
// Decodificación de instrucciones
//-------------------------------------------------------------
module DECODING(
  output [2:0]aa,  // Selección de registro para lectura a bus A
  output [2:0]ba,  // Selección de registro para lectura a bus B
  output [2:0]da,  // Selección de registro para escritura
  output dwr,      // Habilitación de escritura en registro
  output jmp,      // Instrucción de salto (escribir PC)
  output ld,       // Instrucción LD (lectura de memoria)
  output st,       // Instrucción ST (escritura en memoria)
  output ldi,      // Instrucción LDI (escritura en memoria)
  output ldh,      // Instrucción LDH (escritura en memoria)
  output imm,      // Operandos inmediatos a bus B si 1
  output pca,      // PC -> ALU_A si 1
  output wrcv,     // Escribir en flags C y V
  output wrzn,     // Escribir en flags Z y N
  output [1:0]fs,  // Operación de la ALU
  output zab,      // Forzar entrada A a 0 en la ALU si 0
  output ib,       // Invertir bits de entrada B de la ALU si 1
  output xa,       // Instrucciones extendidas (con acarreo)
  output ror,      // Rotaciones/desplazamientos a derechas
  input [15:0]op,  // Código de operación desde registro de instrucción
  input cf,        // Entrada de Flag de Acarreo
  input vf,        // Entrada de Flag de Overflow
  input zf,        // Entrada de Flag de Zero
  input nf,        // Entrada de Flag de Signo
  input opval      // Entrada de instrucción válida
);
// multiplexor de 8 a 1 para saltos
reg jr;    // No es un registro
always@*
  case (op[14:12])
     0 : jr <=  zf;
     1 : jr <= ~zf;
     2 : jr <=  cf;
     3 : jr <= ~cf;
     4 : jr <=  nf;
     5 : jr <= ~nf;
     6 : jr <=  vf;
     7 : jr <= 1'b1;
  endcase
wire jind;
assign jind = opval & (~op[15]) & op[14] & op[13] & (~op[12]) & op[11] & op[7];
assign jmp = (opval & op[15] & jr) | jind;
```

```
// Selección de registros
assign aa = op[6:4];
assign ba = op[2:0];
assign da = op[10:8];
// Ciertos OP-codes y líneas de control
assign ld  = opval & (~op[15]) & op[14] & op[13] & (~op[12]) & (~op[11]) & (~op[7]);
assign st  = opval & (~op[15]) & op[14] & op[13] & (~op[12]) & (~op[11]) & op[7];
assign ldh = opval & (~op[15]) & op[14] & op[13] &   op[12]  & (~op[11]);
assign ldi = opval & (~op[15]) & op[14] & op[13] &   op[12]  &   op[11];
reg [1:0]fs;
reg zab, ib, xa, ror, imm, wd, wc, wz, pca;
always@*
 casex ({op[15:11],op[7]})
 6'b000000:{fs,zab,ib,xa,ror,imm,wd,wc,wz,pca}<=
           //  fs  zab   ib   xa  ror   imm   wd   wc   wz  pca
           {2'b00,1'b1,1'b0,1'b0,1'b0, 1'b0,1'b1,1'b1,1'b1,1'b0}; // ADD
 6'b000001:{fs,zab,ib,xa,ror,imm,wd,wc,wz,pca}<=
           //  fs  zab   ib   xa  ror   imm   wd   wc   wz  pca
           {2'b00,1'b1,1'b0,1'b0,1'b0, 1'b1,1'b1,1'b1,1'b1,1'b0}; // ADDI
 6'b000010:{fs,zab,ib,xa,ror,imm,wd,wc,wz,pca}<=
           //  fs  zab   ib   xa  ror   imm   wd   wc   wz  pca
           {2'b00,1'b1,1'b0,1'b1,1'b0, 1'b0,1'b1,1'b1,1'b1,1'b0}; // ADC
 6'b000011:{fs,zab,ib,xa,ror,imm,wd,wc,wz,pca}<=
           //  fs  zab   ib   xa  ror   imm   wd   wc   wz  pca
           {2'b00,1'b1,1'b0,1'b1,1'b0, 1'b1,1'b1,1'b1,1'b1,1'b0}; // ADCI
 6'b000100:{fs,zab,ib,xa,ror,imm,wd,wc,wz,pca}<=
           //  fs  zab   ib   xa  ror   imm   wd   wc   wz  pca
           {2'b00,1'b1,1'b1,1'b0,1'b0, 1'b0,1'b1,1'b1,1'b1,1'b0}; // SUB
 6'b000101:{fs,zab,ib,xa,ror,imm,wd,wc,wz,pca}<=
           //  fs  zab   ib   xa  ror   imm   wd   wc   wz  pca
           {2'b00,1'b1,1'b1,1'b0,1'b0, 1'b1,1'b1,1'b1,1'b1,1'b0}; // SUBI
 6'b000110:{fs,zab,ib,xa,ror,imm,wd,wc,wz,pca}<=
           //  fs  zab   ib   xa  ror   imm   wd   wc   wz  pca
           {2'b00,1'b1,1'b1,1'b1,1'b0, 1'b0,1'b1,1'b1,1'b1,1'b0}; // SBC
 6'b000111:{fs,zab,ib,xa,ror,imm,wd,wc,wz,pca}<=
           //  fs  zab   ib   xa  ror   imm   wd   wc   wz  pca
           {2'b00,1'b1,1'b1,1'b1,1'b0, 1'b1,1'b1,1'b1,1'b1,1'b0}; // SBCI
 6'b001000:{fs,zab,ib,xa,ror,imm,wd,wc,wz,pca}<=
           //  fs  zab   ib   xa  ror   imm   wd   wc   wz  pca
           {2'b00,1'b1,1'b1,1'b0,1'b0, 1'b0,1'b0,1'b1,1'b1,1'b0}; // CMP
 6'b001001:{fs,zab,ib,xa,ror,imm,wd,wc,wz,pca}<=
           //  fs  zab   ib   xa  ror   imm   wd   wc   wz  pca
           {2'b00,1'b1,1'b1,1'b0,1'b0, 1'b1,1'b0,1'b1,1'b1,1'b0}; // CMPI
 6'b001010:{fs,zab,ib,xa,ror,imm,wd,wc,wz,pca}<=
           //  fs  zab   ib   xa  ror   imm   wd   wc   wz  pca
           {2'b11,1'b1,1'b0,1'bx,1'b0, 1'b0,1'b1,1'b0,1'b1,1'b0}; // AND
 6'b001011:{fs,zab,ib,xa,ror,imm,wd,wc,wz,pca}<=
           //  fs  zab   ib   xa  ror   imm   wd   wc   wz  pca
           {2'b11,1'b1,1'b0,1'bx,1'b0, 1'b1,1'b1,1'b0,1'b1,1'b0}; // ANDI
 6'b001100:{fs,zab,ib,xa,ror,imm,wd,wc,wz,pca}<=
           //  fs  zab   ib   xa  ror   imm   wd   wc   wz  pca
           {2'b11,1'b1,1'b0,1'bx,1'b0, 1'b0,1'b0,1'b0,1'b1,1'b0}; // TST
```

```
6'b001101:{fs,zab,ib,xa,ror,imm,wd,wc,wz,pca}<=
          //  fs  zab   ib   xa  ror   imm   wd   wc   wz  pca
          {2'b11,1'b1,1'b0,1'bx,1'b0, 1'b1,1'b0,1'b0,1'b1,1'b0}; // TSTI
6'b001110:{fs,zab,ib,xa,ror,imm,wd,wc,wz,pca}<=
          //  fs  zab   ib   xa  ror   imm   wd   wc   wz  pca
          {2'b10,1'b1,1'b0,1'bx,1'b0, 1'b0,1'b1,1'b0,1'b1,1'b0}; // OR
6'b001111:{fs,zab,ib,xa,ror,imm,wd,wc,wz,pca}<=
          //  fs  zab   ib   xa  ror   imm   wd   wc   wz  pca
          {2'b10,1'b1,1'b0,1'bx,1'b0, 1'b1,1'b1,1'b0,1'b1,1'b0}; // ORI
6'b010000:{fs,zab,ib,xa,ror,imm,wd,wc,wz,pca}<=
          //  fs  zab   ib   xa  ror   imm   wd   wc   wz  pca
          {2'b01,1'b1,1'b0,1'bx,1'b0, 1'b0,1'b1,1'b0,1'b1,1'b0}; // XOR
6'b010001:{fs,zab,ib,xa,ror,imm,wd,wc,wz,pca}<=
          //  fs  zab   ib   xa  ror   imm   wd   wc   wz  pca
          {2'b01,1'b1,1'b0,1'bx,1'b0, 1'b1,1'b1,1'b0,1'b1,1'b0}; // XORI
6'b010010:{fs,zab,ib,xa,ror,imm,wd,wc,wz,pca}<=
          //  fs  zab   ib   xa  ror   imm   wd   wc   wz  pca
          {2'b10,1'b0,1'b1,1'bx,1'b0, 1'b0,1'b1,1'b0,1'b1,1'bx}; // NOT
6'b010011:{fs,zab,ib,xa,ror,imm,wd,wc,wz,pca}<=
          //  fs  zab   ib   xa  ror   imm   wd   wc   wz  pca
          {2'b00,1'b0,1'b1,1'b0,1'b0, 1'b0,1'b1,1'b0,1'b1,1'bx}; // NEG
6'b010100:{fs,zab,ib,xa,ror,imm,wd,wc,wz,pca}<=
          //  fs  zab   ib   xa  ror   imm   wd   wc   wz  pca
          {2'bxx,1'bx,1'b0,1'b0,1'b1, 1'b0,1'b1,1'b1,1'b1,1'b0}; // SHR
6'b010101:{fs,zab,ib,xa,ror,imm,wd,wc,wz,pca}<=
          //  fs  zab   ib   xa  ror   imm   wd   wc   wz  pca
          {2'bxx,1'bx,1'b1,1'b0,1'b1, 1'b0,1'b1,1'b1,1'b1,1'b0}; // SHRA
6'b010110:{fs,zab,ib,xa,ror,imm,wd,wc,wz,pca}<=
          //  fs  zab   ib   xa  ror   imm   wd   wc   wz  pca
          {2'bxx,1'bx,1'b1,1'b1,1'b1, 1'b0,1'b1,1'b1,1'b1,1'b0}; // ROR
6'b011000:{fs,zab,ib,xa,ror,imm,wd,wc,wz,pca}<=
          //  fs  zab   ib   xa  ror   imm   wd   wc   wz  pca
          {2'b10,1'b0,1'b0,1'bx,1'b0, 1'b0,1'b1,1'b0,1'b1,1'bx}; // LD
6'b011001:{fs,zab,ib,xa,ror,imm,wd,wc,wz,pca}<=
          //  fs  zab   ib   xa  ror   imm   wd   wc   wz  pca
          {2'bxx,1'bx,1'bx,1'bx,1'bx, 1'bx,1'b0,1'b0,1'b0,1'b0}; // ST
6'b011010:{fs,zab,ib,xa,ror,imm,wd,wc,wz,pca}<=
          //  fs  zab   ib   xa  ror   imm   wd   wc   wz  pca
          {2'b00,1'b1,1'b0,1'b0,1'b0, 1'b1,1'b1,1'b0,1'b0,1'b1}; // ADPC
6'b011011:{fs,zab,ib,xa,ror,imm,wd,wc,wz,pca}<=
          //  fs  zab   ib   xa  ror   imm   wd   wc   wz  pca
          {2'b10,1'b0,1'b0,1'bx,1'b0, 1'b0,1'b0,1'b0,1'b0,1'bx}; // JIND
6'b01110?:{fs,zab,ib,xa,ror,imm,wd,wc,wz,pca}<=
          //  fs  zab   ib   xa  ror   imm   wd   wc   wz  pca
          {2'bxx,1'bx,1'bx,1'bx,1'bx, 1'bx,1'b0,1'b0,1'b0,1'bx}; // LDH
6'b01111?:{fs,zab,ib,xa,ror,imm,wd,wc,wz,pca}<=
          //  fs  zab   ib   xa  ror   imm   wd   wc   wz  pca
          {2'b10,1'b0,1'b0,1'bx,1'b0, 1'b1,1'b1,1'b0,1'b0,1'bx}; // LDI
6'b1?????:{fs,zab,ib,xa,ror,imm,wd,wc,wz,pca}<=
          //  fs  zab   ib   xa  ror   imm   wd   wc   wz  pca
          {2'b00,1'b1,1'b0,1'b0,1'b0, 1'b1,1'b0,1'b0,1'b0,1'b1}; // JRx
 default: {fs,zab,ib,xa,ror,imm,wd,wc,wz,pca}<=
```

```
            //  fs  zab   ib   xa  ror   imm   wd   wc   wz  pca
            {2'bxx,1'bx,1'bx,1'bx,1'bx, 1'bx,1'bx,1'bx,1'bx,1'bx};
 endcase
// Escrituras en registros
assign dwr  = opval & wd;  // banco de registros
assign wrcv = opval & wc;  // Flags C y V
assign wrzn = opval & wz;  // Flags Z y N
endmodule
```

Instanciando e interconectándolos los bloques anteriores tenemos finalmente nuestra descripción de la CPU en lenguaje Verilog.

```
//-------------------------------------------------------------
// CPU
//-------------------------------------------------------------
module CPU_1CVN(
  output [15:0]a,   // Bus de direcciones
  output [15:0]do,  // Bus de datos de salida
  output rw,        // Escritura si 0, Lectura si 1
  input  [15:0]di,  // Bus de datos de entrada
  input  clk,       // Entrada de Reloj
  input  resetb     // Entrada de Reset, activa en bajo
);
wire [15:0]rega;
wire [15:0]regb;
wire [15:0]busd;
wire [2:0]aa;
wire [2:0]ba;
wire [2:0]da;
wire dwr;
REGBANK rbnk ( .a(rega),.b(regb),.d(busd),.asel(aa),.bsel(ba),
               .dsel(da),.wren(dwr),.clk(clk));
assign do = regb;
wire [15:0]regpc;
wire pcinc,ld,st,jmp,imm;
assign pcinc = ~(ld | st);
REGPC pc ( .q(regpc), .d(busd), .resb(resetb),
           .load(jmp), .inc(pcinc), .clk(clk) );
assign a = (ld | st) ? rega: regpc;
wire [15:0]alua;
wire [15:0]alub;
wire cd,vd,zd,nd;
wire [1:0]aluop;
wire zab,ib,xa,ror,pca;
assign alua = pca ? regpc : rega;
assign alub = ld ? di : (imm ? busimm : regb);
ALU alu ( .y(busd),.co(cd),.v(vd),.z(zd),.n(nd),.a(alua),.b(alub),
   .op(aluop),.cf(cf),.zab(zab),.ib(ib),.ror(ror),.xa(xa) );
```

```
wire cf,vf,zf,nf;
wire wrcv,wrzn;
FLAGS flagr ( .qc(cf),.qv(vf),.qz(zf),.qn(nf), .dc(cd),.dv(vd),
              .dz(zd),.dn(nd),.wrzn(wrzn), .wrcv(wrcv), .clk(clk) );
wire [15:0]busop;
wire opval;
IR ir( .q(busop), .opval(opval), .d(di),
    .ld(ld), .st(st), .jmp(jmp), .clk(clk), .resb(resetb) );
wire [15:0]busimm;
wire ldi, ldh;
IMM immdat( .f(busimm), .op(busop), .ldi(ldi), .ldh(ldh), .clk(clk) );
DECODING decoder(.aa(aa),.ba(ba),.da(da),.dwr(dwr),
 .jmp(jmp),.ld(ld),.st(st),.ldh(ldh),.ldi(ldi),.pca(pca),
 .imm(imm),.wrcv(wrcv),.wrzn(wrzn),
                 .fs(aluop),.zab(zab),.ib(ib),.xa(xa),.ror(ror),
 .op(busop),.cf(cf),.vf(vf),.zf(zf),.nf(nf),.opval(opval) );
assign rw = ~st;
endmodule
```

El último problema que tenemos que afrontar es que la CPU por sí sola no sirve de nada. Necesitamos instanciarla en un sistema que además incluya una memoria y al menos un periférico que nos comunique con el exterior. Estos componentes también sería interesante tenerlos sintetizados dentro de la FPGA, lo que hacemos en el siguiente código en el que además incluimos una descripción parametrizable para la memoria RAM (la UART se describirá al final de este documento). El contenido inicial de la memoria se puede especificar y queda almacenado en los datos de configuración de la FPGA, con lo que no necesitamos incluir en nuestro sistema ninguna memoria ROM.

```
//-------------------------------------------------------------------
// Sistema con CPU, RAM y UART
//-------------------------------------------------------------------
module SYSTEM (output txd, input rxd, input clk, input resb);
//-- Instanciamos
wire [15:0]addr;
wire [15:0]cpu_di;
wire [15:0]cpu_do;
wire rw;
CPU_1CVN cpu( .a(addr),.do(cpu_do),.rw(rw),.di(cpu_di),.clk(clk),
              .resetb(resb));
wire [15:0]ram_do;
wire ramwr;
genram #(
    .INITFILE("out.hex"),
    .AW(12),
    .DW(16)
    ) ram0 (.clk(clk),.addr(addr[11:0]),.rw(ramwr),.data_in(cpu_do),
            .data_out(ram_do));
```

```
wire [7:0]uartdo;
UART #(.DIVISOR(208)) uart1(.txd(txd), .do(uartdo), .rxd(rxd),
            .di(cpu_do[7:0]),.rs(addr[0]), .cs(csuart), .rw(rw),
            .clk(clk));
//-- Decodificación de direcciones:
//-- RAM = 0x0000 a 0x0FFF
//-- UART_DATA = 0xFFF0
//-- UART_STAT = 0xFFF1
wire csram, csuart;
assign csram = (~addr[15]) & (~addr[14]) & (~addr[13]) & (~addr[12]);
assign csuart = addr[15] & addr[14] & addr[13] & addr[12]
  & addr[11] & addr[10] & addr[ 9] & addr[ 8]
  & addr[ 7] & addr[ 6] & addr[ 5] & addr[ 4];
assign ramwr = ~((~rw) & csram);
assign uwr = (~rw) & csuart & (~addr[0]);
// Multiplexor de datos de entrada a CPU
assign cpu_di = csuart ? {8'h00,uartdo} : (csram ? ram_do : 16'hxxxx );
endmodule
//-----------------------------------------------------------------------
//-- Memoria RAM genérica
//-----------------------------------------------------------------------
module genram (
    input clk,                      //-- Señal de reloj global
    input wire [AW-1: 0] addr,      //-- Direcciones
    input wire rw,                  //-- Modo lectura (1) o escritura (0)
    input wire [DW-1: 0] data_in,   //-- Dato de entrada
    output reg [DW-1: 0] data_out   //-- Dato a escribir
);
parameter INITFILE = "out.hex";     // Datos iniciales por defecto
parameter AW = 8;    // Anchura de bus de direcciones por defecto
parameter DW = 16;   // Anchura de buses de datos por defecto
  //-- Memoria
  reg [DW-1: 0] ram [0: (1<<AW)-1];
  //-- Lectura de la memoria (en flanco de bajada)
  always @(negedge clk) begin
    if (rw == 1)
    data_out <= ram[addr];
  end
  //-- Escritura en la memoria
  always @(posedge clk) begin
    if (rw == 0)
     ram[addr] <= data_in;
  end
//-- Cargar en la memoria el fichero ROMFILE
//-- Los valores deben estan dados en hexadecimal
initial begin
```

```
    $readmemh(INITFILE, ram);
end
endmodule
```

4.3 Primeras pruebas

Como ejemplo de funcionamiento mostramos los datos transmitidos desde la UART. El código de la CPU está calculando los números primos hasta el 32749 usando el método de la criba de Eratóstenes:

```
2 3 5 7 11 13 17 19 23 29
31 37 41 43 47 53 59 61 67 71
73 79 83 89 97 101 103 107 109 113
127 131 137 139 149 151 157 163 167 173
179 181 191 193 197 199 211 223 227 229
233 239 241 251 257 263 269 271 277 281
283 293 307 311 313 317 331 337 347 349
353 359 367 373 379 383 389 397 401 409
419 421 431 433 439 443 449 457 461 463
467 479 487 491 499 503 509 521 523 541
547 557 563 569 571 577 587 593 599 601
...
```

5 Tercer diseño: CPU con interrupciones

Ninguna de las dos CPUs anteriores era capaz de dar soporte a interrupciones. Por ello esta parecía la siguiente mejora a implementar en nuestro diseño. Una interrupción es un evento externo a la CPU, generado por algún periférico, que detiene la ejecución del programa que en ese momento estuviese corriendo en el procesador para saltar de forma automática a otra sección de código que atiende al evento, y al finalizar dicho código se retorna a la ejecución del programa que había sido interrumpido. Todo el proceso ha de resultar invisible para el programa que se interrumpe, si exceptuamos el mayor tiempo de ejecución, y ello nos obliga a:

1. Guardar el estado del procesador antes de ejecutar el código de la rutina de interrupción. Este estado incluye el valor de todos los registros cuyo contenido se vaya a modificar en la rutina de interrupción. Y no sólo hablamos de los registros de propósito general, no debemos olvidarnos del propio contador de programa, y los flags de estado.

2. Recuperar el valor de todos los registros modificados antes de retornar al programa interrumpido.

3. Incluir una nueva instrucción para retornar de la interrupción al programa interrumpido. Esta será la última instrucción que se ejecute en la rutina de interrupción.

Los registros de propósito general se pueden guardar en la memoria por programa. Una buena idea sería recurrir a una pila de datos. Si, por ejemplo, usamos como puntero de pila el registro R7, y queremos guardar y recuperar el contenido de R0 y R1, el código sería:

```
IRQ:    SUBI    R7,R7,1     ; Decrementamos puntero de pila
        ST      (R7),R0     ; Guardamos R0
        SUBI    R7,R7,1     ; Y repetimos para R1
        ST      (R7),R1
```

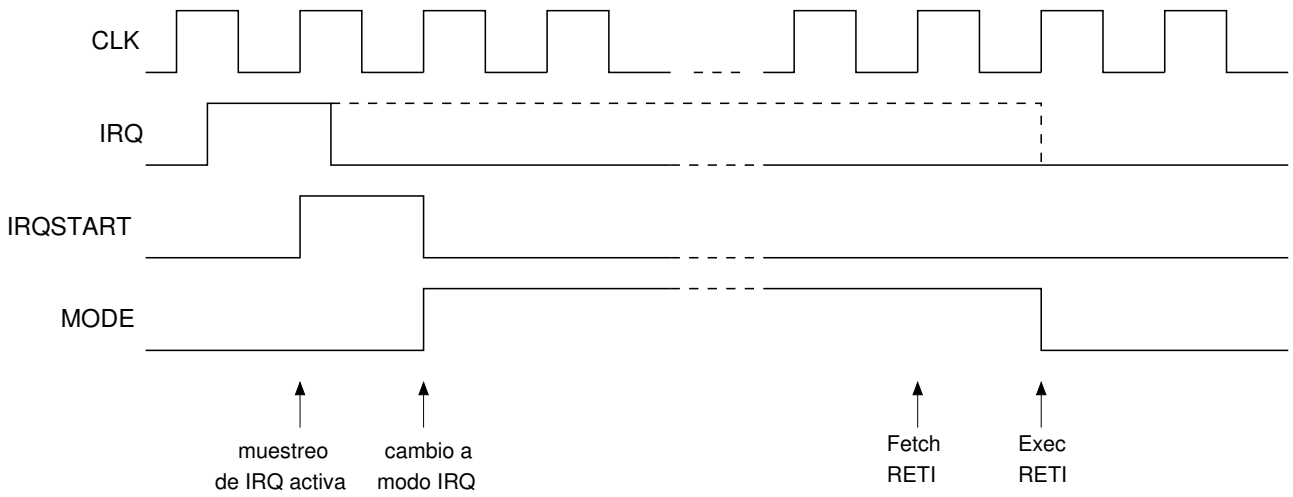


Figure 11: Cronograma de las señales involucradas en la ejecución de una rutina de interrupción.

```
; Aquí va el código útil de la rutina de interrupción
.....
LD      R1,(R7)     ; Recuperamos R1
ADDI    R7,R7,1     ; Incrementamos puntero de pila
LD      R0,(R7)     ; Y repetimos para R0
ADDI    R7,R7,1
RETI                ; Retornamos de la interrupción
```

El resto de los registros modificados se tienen que grabar de forma automática al producirse la interrupción y recuperar su contenido al ejecutar la instrucción RETI. En particular se trata de los registros contador de programa, que contiene la dirección a la que tendrá que retornar la ejecución tras la interrupción, el registro de flags, que con toda seguridad se va a modificar con el código de la rutina de interrupción, y no debemos olvidarnos de RH, que también se va a modificar si en la rutina de interrupción tenemos alguna instrucción LDH o LDI. Estos registros no pueden grabarse en la memoria por programa, por lo que hemos de añadir un almacenamiento para guardar sus valores en el momento de producirse la interrupción.

La solución por la que hemos optado ha sido la de añadir unos registros alternativos para PC, flags, y RH, de modo que en el programa principal se usarán los registros principales y en la rutina de interrupción los alternativos. Dado que así la interrupción no modifica los registros principales no hay necesidad de copiarlos y recuperarlos, nos basta con cambiar de usar unos a otros.

Esta solución tiene como inconveniente el no permitir el anidamiento de interrupciones. Es decir, si ya estamos ejecutando una rutina de interrupción esta no se puede volver a interrumpir. Esto no pensamos que sea un gran inconveniente en la práctica ya que las rutinas de interrupción han de ser breves. La ventaja es, sin embargo, una latencia de interrupción muy corta.

La implementación hardware de este mecanismo pasa por duplicar los registros PC, Flags, y RH, y seleccionar el registro adecuado mediante multiplexores controlados por una señal de 'MODO' que valdrá 0 cuando se ejecuta el programa principal y 1 cuando se ejecuta la rutina de interrupción.

En el cambio de modo hay que ser cautelosos a causa del pipelining. Si se cambia la señal de modo antes de que termine de ejecutarse la instrucción actual ésta podría modificar los registros equivocados. Para evitar esto el cambio de la señal de modo debe retrasarse un ciclo, de modo que se termine de ejecutar la instrucción actual a la vez que se invalida el contenido de IR, lo que nos garantiza que en el ciclo en el que se cambia MODO no se va a ejecutar ninguna instrucción. Este ciclo de reloj previo al cambio de modo se señaliza poniendo en alto la línea 'IRQSTART', tal como se muestra en el cronograma de la llamada a la rutina de interrupción de la figura 11.

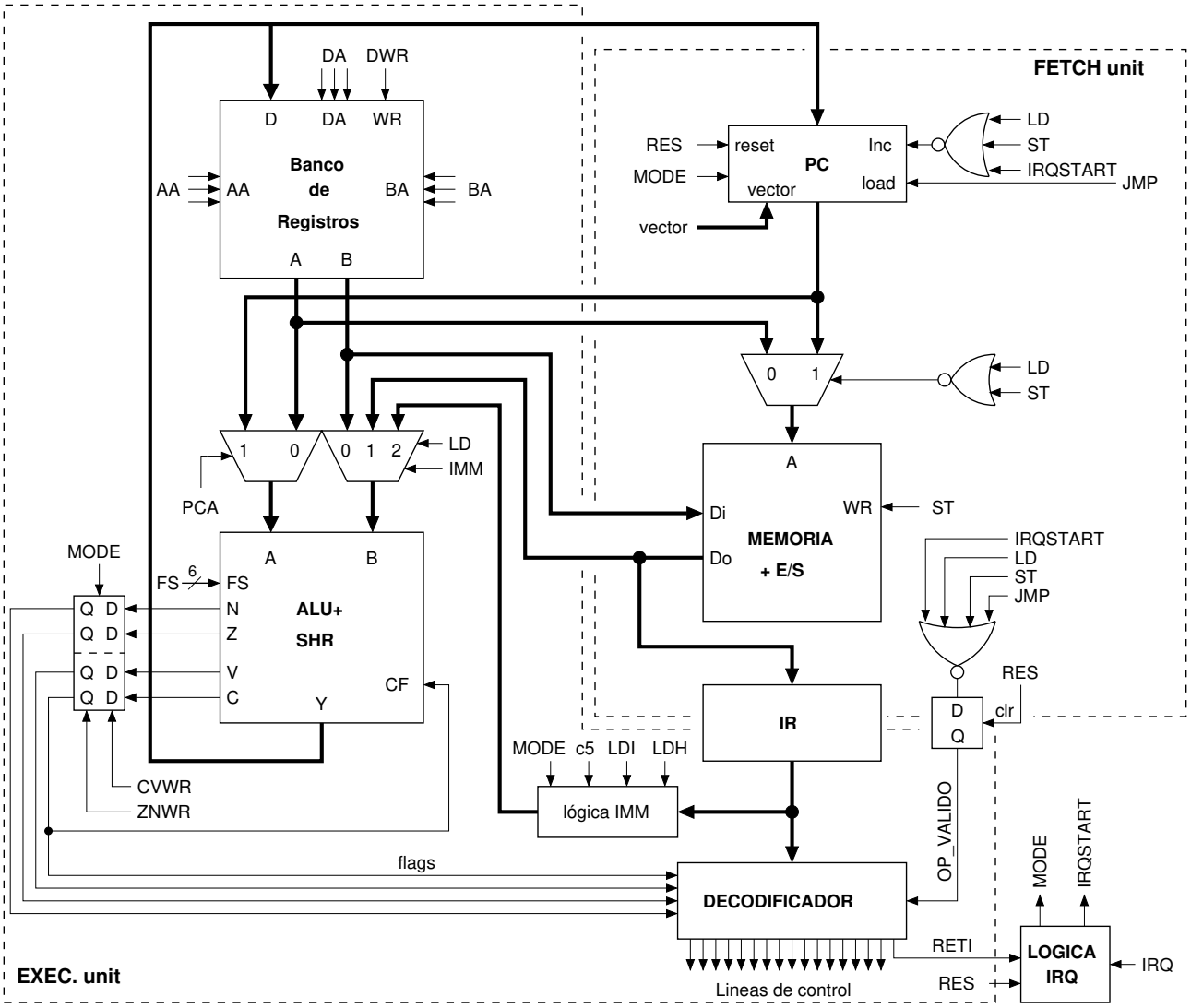


Figure 12: Diagrama de bloques de la CPU con interrupciones.

El mismo problema se plantea al retornar al programa principal, salvo que aquí sabemos qué instrucción concreta se va a ejecutar, que no es otra sino RETI. También en este caso se ha de invalidar el contenido de IR, pero analizando el repertorio de instrucciones de la figura 8 hemos tenido la suerte de disponer de una instrucción de salto con bits irrelevantes en su código de operación. En particular se trata de JIND, en la que ahora si ponemos el bit 3 de su código de operación en 1 pasará a ser RETI. RETI se ejecuta como una instrucción de salto, y de hecho va a modificar el valor del PC alternativo, pero una vez finalizada la rutina de interrupción eso no importa, lo realmente importante es que al tratarse de un salto va a invalidar el contenido de IR. Nótese que en el programa principal RETI es indistinguible de JIND R0, mientras que en la rutina de interrupción su principal cometido es el de volver a poner la señal de MODO en 0.

Con estas consideraciones podemos presentar el diagrama de bloques para la CPU con interrupciones mostrado en la figura 12. Aquí hay que mencionar además que se ha inhibido el incremento del PC durante el ciclo IRQSTART para evitar saltarnos una instrucción sin ejecutar cuando se produce una interrupción, pues recordemos que en ese ciclo el código de operación leído de la memoria se ha invalidado. Por lo demás hay que destacar que los bloques PC, Flags, y lógica de Inmediatos (registro RH) ahora cuentan con una entrada de MODO que selecciona uno de los dos registros posibles de cada tipo. Y finalmente también incluimos un bloque de lógica de interrupción que será el encargado de generar las temporizaciones del cronograma de la figura 11.

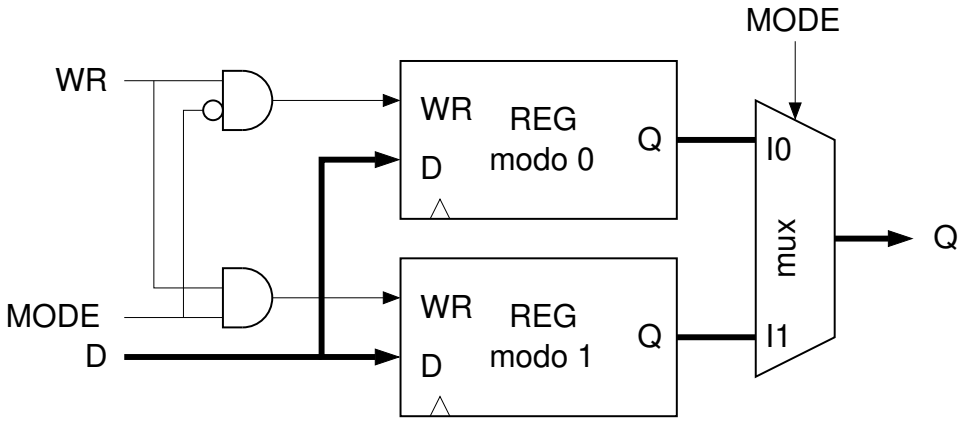


Figure 13: Diagrama simplificado de los registros dobles Flags, y RH

5.1 Registros dobles

En la figura 13 se da un esquemático para los registros dobles en el que se muestra que dependiendo del valor de la señal MODE las lecturas y las escrituras se dirigen hacia uno de los dos registros posibles, quedando el otro inalterado. Este es el diagrama seguido en el rediseño de los registros de los Flags y RH. El contador de programa es distinto ya que debe poder incrementarse y esa función de incremento sólo se efectúa sobre uno de los dos registros. Por ello pensamos que implementar este registro como dos contadores seleccionables no es la forma más óptima de hacerlo. En su lugar hemos recurrido al diagrama de la figura 14, dónde la función de incremento se tiene en un semisumador que suma el bit INC al valor de la salida del registro. Por otra parte vemos que cuando MODE vale 0 el registro PC del modo 1 se escribe con una constante, 'vector', que es precisamente la dirección de la memoria en la que comienza la rutina de interrupción, de modo que cuando se conmute al modo de interrupción ya tenemos el PC precargado con la dirección a la que se va a saltar. Vector puede ser una constante con la dirección a la que se saltará al producirse la interrupción, o podría ser una variable de 16 bits procedente de una lógica de interrupciones vectorizadas, lo que nos permitiría saltar a una dirección distinta por cada fuente de interrupción.

Una modificación adicional ha consistido en añadir una señal de reset para el registro RH del modo IRQ. Esto nos puede ahorrar una instrucción 'LDH 0' al comienzo de las rutinas de interrupción.

A continuación se incluye el listado del código Verilog del nuevo registro PC:

```
//
// Registro Contador de programa
//
module REGPC (output [15:0]q, input [15:0]d,
              input [15:0]vector, input resb, input load,
              input inc, input mode, input clk);
reg [15:0]q0=0;
reg [15:0]q1;
wire [15:0]li;
wire [15:0]inco;
assign inco = q + inc;          // incrementador
assign li = load ? d : inco;    // mux load / inc
// registro PC modo normal
always @(posedge clk or negedge resb )
if (!resb) q0<=16'h0000;
else q0<= mode ? q0 : li;
```

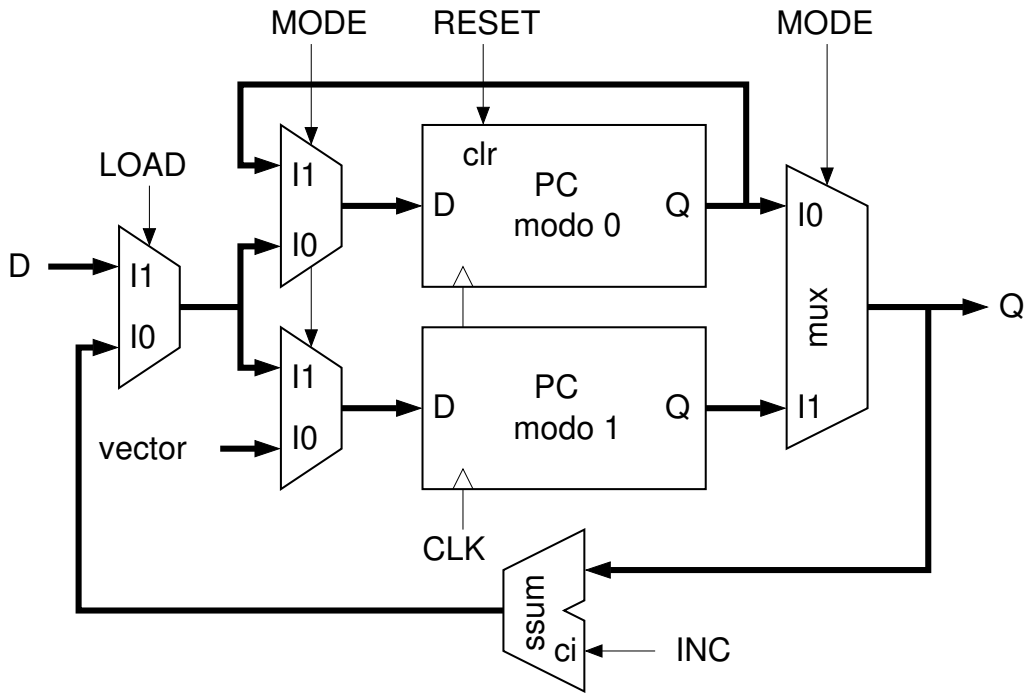


Figure 14: Esquema del registro PC doble.

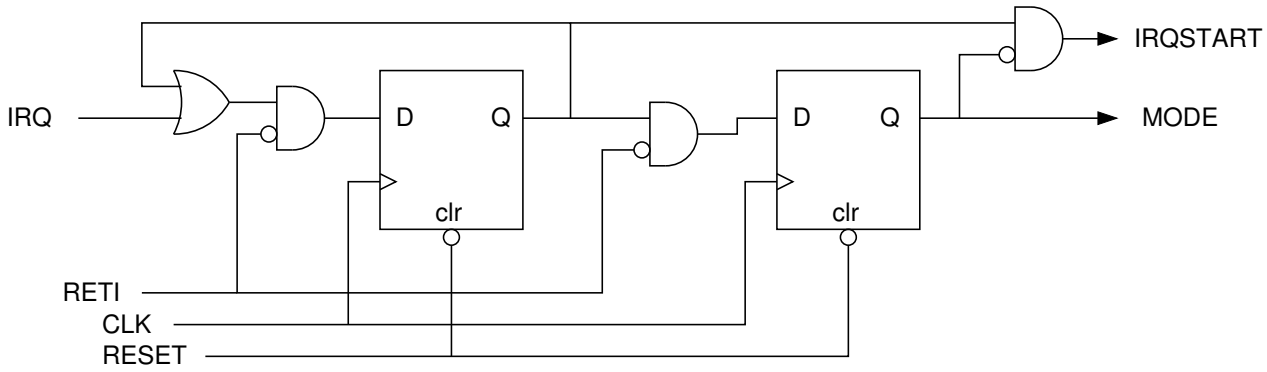


Figure 15: Lógica de control de interrupciones.

```
// registro PC modo IRQ
always @(posedge clk )
q1<= mode ? li : vector;
assign q = mode ? q1 : q0;      // mux de salida
endmodule
```

5.2 Lógica de cambio de modo e instrucción RETI

El bloque de lógica de interrupción se trata de un circuito secuencial bastante simple, como el que se detalla en la figura 15. Nótese que una vez que se muestrea IRQ activa este estado queda memorizado hasta la ejecución de RETI, que IRQSTART sólo está en alto durante un ciclo, y que recurrimos a un reset asíncrono para comenzar la ejecución con MODE en 0.

El listado Verilog de este bloque es el siguiente:

```
//
// Lógica de petición de interrupciones
//
module IRQLOGIC (
```

```
  input    irq,          // Petición de interrupción
  input    reti,         // final de interrupción
  output   irqstart,     // pulso de comienzo de IRQ
  output   mode,         // modo del micro: normal o IRQ
  input    resb,         // reset, activo en bajo
  input    clk
);
reg q0=0, mode=0;
always @(posedge clk or negedge resb )
  begin
    if (!resb) begin
        q0<=1'b0;
        mode<=1'b0;
    end
    else begin
        q0 <= (~reti) & (q0 | irq);
        mode <= (~reti) & q0;
    end
  end
assign irqstart = (~mode) & q0;
endmodule
```

En lo tocante al decodificador la instrucción RETI activa las mismas señales que JIND puesto que no lo hemos modificado para nada. Aunque sí se ha añadido una lógica de tipo AND para detectar el código de operación 1'b01101xxx1xxx1xxx, dando una señal RETI activa cuando el registro IR contiene este código de operación a la vez que está activo OP_VALIDO. Su código Verilog es:

```
assign reti= opval & (~op[15]) & op[14] & op[13] & (~op[12])
             & op[11] & op[7] & op[3];
```

En el ensamblador el mnemónico RETI genera el código de operación 0x6888, que equivale a JIND R0, de modo que el contenido de R0 se copia en el PC del modo 1 cuando se retorna de la interrupción (JIND Rx genera los códigos 0x6880 a 0x6887). Sin embargo, en el siguiente ciclo el PC del modo 1 va a volver a escribirse con 'vector', con lo que su valor va a ser correcto antes de que se produzca otra interrupción y se conmute al modo 1.

6 Cuarta variante: Constantes simplificadas e interrupciones concatenadas

6.1 CPU sin registros RH.

Reflexionando acerca de la carga de constantes de 16 bits observé que la instrucción LDH era en realidad un mecanismo retorcido, heredado del diseño de CPU Harvard de un ciclo, que se podría obviar en las versiones Von Neumann ahorrando con ello un poco de lógica. La idea aquí es usar el propio contador de programa como puntero a la constante que queremos cargar. Así la instrucción LDH pasa a ser LDPC Rd (ver figura 16), donde el dato leído desde la memoria se copiará en Rd además de en el registro IR (en el que quedará marcado como código de operación no válido). El resultado final es que la instrucción LDPC ocupa dos palabras efectivas, pues ha de ir seguida de un dato de 16bits, y tarda dos ciclos en ejecutarse. Esto es lo mismo que teníamos con las secuencias de instrucciones LDH más LDI, por lo que no se esperan mejoras en el rendimiento del

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	mnemónico	flags				operación
0	0	0	0	0	RD			0	RA			–	RB			ADD	C	V	N	Z	RD= RA+RB
0	0	0	0	0	RD			1	RA			dato				ADDI	C	V	N	Z	RD= RA+dato
0	0	0	0	1	RD			0	RA			–	RB			ADC	C	V	N	Z	RD=RA+RB+Cflag
0	0	0	0	1	RD			1	RA			dato				ADCI	C	V	N	Z	RD=RA+dato+Cflag
0	0	0	1	0	RD			0	RA			–	RB			SUB	C	V	N	Z	RD=RA−RB
0	0	0	1	0	RD			1	RA			dato				SUBI	C	V	N	Z	RD=RA−dato
0	0	0	1	1	RD			0	RA			–	RB			SBC	C	V	N	Z	RD=RA−RB−(~Cflag)
0	0	0	1	1	RD			1	RA			dato				SBCI	C	V	N	Z	RD=RA−dato−(~Cflag)
0	0	1	0	0	–	–	–	0	RA			–	RB			CMP	C	V	N	Z	RA−RB
0	0	1	0	0	–	–	–	1	RA			dato				CMPI	C	V	N	Z	RA−dato
0	0	1	0	1	RD			0	RA			–	RB			AND	–	–	N	Z	RD=RA&RB
0	0	1	0	1	RD			1	RA			dato				ANDI	–	–	N	Z	RD=RA&dato
0	0	1	1	0	–	–	–	0	RA			–	RB			TST	–	–	N	Z	RA&RB
0	0	1	1	0	–	–	–	1	RA			dato				TSTI	–	–	N	Z	RA&dato
0	0	1	1	1	RD			0	RA			–	RB			OR	–	–	N	Z	RD=RA|RB
0	0	1	1	1	RD			1	RA			dato				ORI	–	–	N	Z	RD=RA|dato
0	1	0	0	0	RD			0	RA			–	RB			XOR	–	–	N	Z	RD=RA^RB
0	1	0	0	0	RD			1	RA			dato				XORI	–	–	N	Z	RD=RA^dato
0	1	0	0	1	RD			0	–	–	–	–	RB			NOT	–	–	N	Z	RD=~Rb
0	1	0	0	1	RD			1	–	–	–	–	RB			NEG	–	–	N	Z	RD=−RB
0	1	0	1	0	RD			0	–	–	–	–	RB			SHR	C	?	N	Z	RD=RB/2, Cflag=RB.0
0	1	0	1	0	RD			1	–	–	–	–	RB			SHRA	C	?	N	Z	RD=RB/2, Cflag=RB.0 (con signo)
0	1	0	1	1	RD			0	–	–	–	–	RB			ROR	C	?	N	Z	RD=(RB>>1)|(Cflag<<15), Cflag=RB.0
0	1	0	1	1				1								ILEG					
0	1	1	0	0	RD			0	RA			–	–	–	–	LD	–	–	N	Z	RD=Mem[RA]
0	1	1	0	0	–	–	–	1	RA			–	RB			ST	–	–	–	–	Mem[RA]=RB
0	1	1	0	1	RD			0	–	–	–	dato				ADPC	–	–	–	–	RD=PC+dato
0	1	1	0	1	–	–	–	1	–	–	–	0	RB			JIND	–	–	–	–	PC=RB
0	1	1	0	1	–	–	–	1	–	–	–	1	–	–	–	RETI	–	–	–	–	Retorna de Interrupcion
0	1	1	1	0	RD			–	–	–	–	–	–	–	–	LDPC	–	–	–	–	RD=Mem[PC++]
0	1	1	1	1	RD			dato								LDI	–	–	–	–	RD=dato (8 bits)
1	0	0	0	desplazamiento con signo												JZ	–	–	–	–	salto si Zflag=1
1	0	0	1	desplazamiento con signo												JNZ	–	–	–	–	salto si Zflag=0
1	0	1	0	desplazamiento con signo												JC	–	–	–	–	salto si Cflag=1
1	0	1	1	desplazamiento con signo												JNC	–	–	–	–	salto si Cflag=0
1	1	0	0	desplazamiento con signo												JMI	–	–	–	–	salto si Nflag=1 (negativo)
1	1	0	1	desplazamiento con signo												JPL	–	–	–	–	salto si Nflag=0 (positivo)
1	1	1	0	desplazamiento con signo												JV	–	–	–	–	salto si Vflag=1 (overflow)
1	1	1	1	desplazamiento con signo												JR	–	–	–	–	salto incondicional

Figure 16: Tabla con los códigos de operación de las instrucciones y lista de flags afectados para la CPU de la nueva revisión (CPU v4)

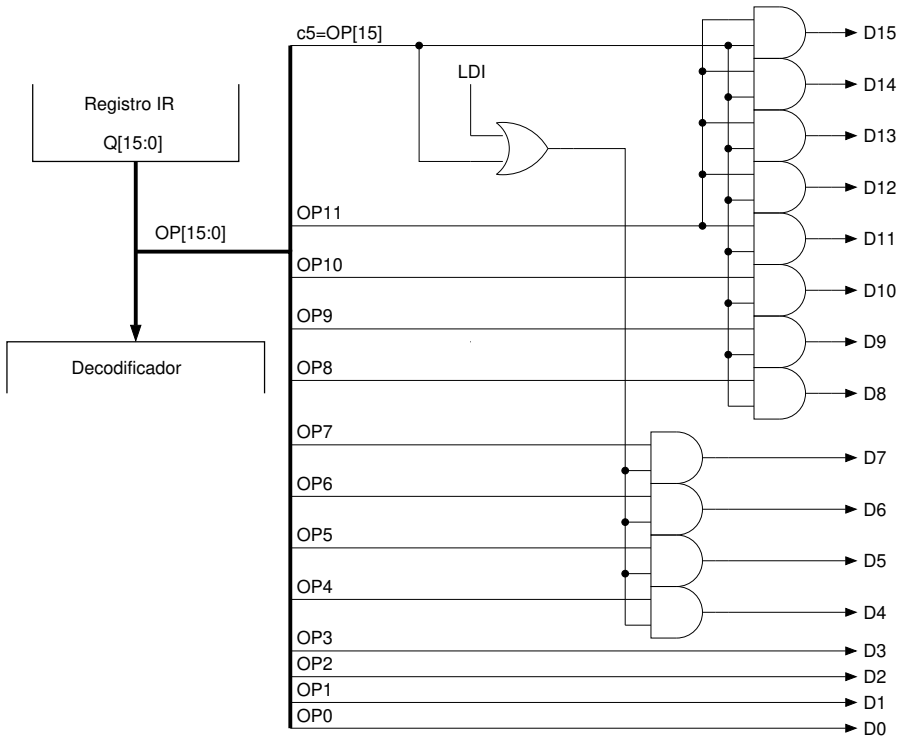


Figure 17: Lógica para la carga de constantes inmediatas en la CPU v4.

software. Sin embargo ahora la lógica de los operandos inmediatos se simplifica mucho pues ya no se necesitan los registros RH, tal como podemos comprobar en la figura 17.

El código Verilog de este bloque también ha quedado mucho más simple, ya que ahora es un circuito puramente combinacional:

```
//-------------------------------------------------------------------------
// Operandos inmediatos
//-------------------------------------------------------------------------
module IMM(
output [15:0]f,         // Salida de operando inmediato
input [15:0]op,         // Entrada desde reg. de instrucción
input ldi,              // instrucción LDI
);
// Salida de operando inmediato
assign f = op[15] ? {op[11],op[11],op[11],op[11],op[11:0]} :
           ( ldi ? {8'h00,op[7:0]}: {12'h000,op[3:0]});
endmodule
```

En la figura 18 se muestra el nuevo diagrama de bloques donde podemos ver que los cambios, aparte de la nueva lógica simplificada para los operandos inmediatos, afectan a sólo dos bloques. Uno de ellos es la lógica del operando válido, que ahora incluye la instrucción LDPC como invalidante del contenido de IR:

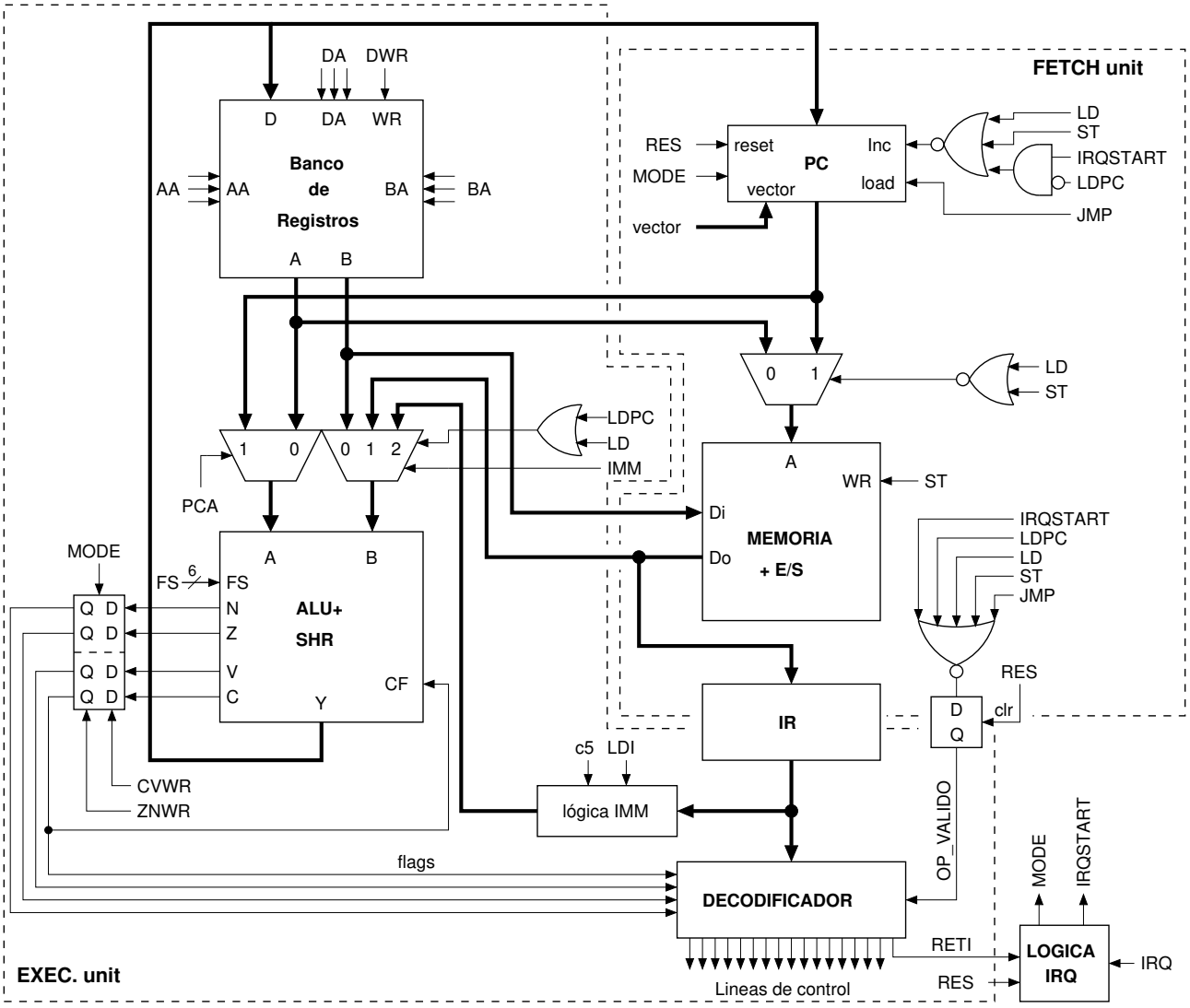


Figure 18: Diagrama de bloques de la CPU v4.

```
assign opvali=~(ld | ldpc | st | jmp | irqstart);
IR ir( .q(busop), .opval(opval), .d(di), .opvali(opvali),
       .clk(clk), .resb(resetb) );
```

También vemos en la figura 18 que ha habido que modificar ligeramente la lógica de selección del operando B de la ALU, lo que se traduce en el siguiente cambio en una línea del código Verilog:

```
assign alub = (ld | ldpc) ? di : (imm ? busimm : regb);
```

Lo que no se muestra en la figura 18 son los cambios en el bloque decodificador que ahora va a generar casi las mismas señales de control que la instrucción LD cuando el código de operación sea el correspondiente a la instrucción LDPC (anteriormente era el código de LDH). Esto es, va a escribir en el banco de registros lo que esté presente en la entrada B de la ALU. Sin embargo, a diferencia de lo que ocurría con LD, que modificaba los flags Z y N dependiendo del dato cargado, LDPC no modifica ningún flag.

Otra diferencia entre LDPC y LD es que la primera instrucción deja incrementarse al contador de programa, de modo que la constante que sigue a la instrucción se salta en el flujo de programa. En cambio LD no incrementa el contador de programa cuando se ejecuta, precisamente para no saltarse la siguiente instrucción. Este comportamiento ha resultado ser problemático en las interrupciones, ya que si una interrupción comienza justo cuando la instrucción LDPC está en su fase de ejecución el PC no se incrementa, provocando que al retorno de la interrupción la constante de LDPC se interprete como un código de operación válido. Para evitar este problema ahora se deja el PC sin incrementar sólo si la interrupción no coincide con la ejecución de una instrucción LDPC:

```
assign pcinc = ~(ld|st|(irqstart&(~ldpc)));     // Incrementa PC
```

En resumen: esta modificación está orientada a simplificar el hardware aunque no mejora el rendimiento del software, si bien tampoco lo empeora. Hay que destacar que esto ha sido posible gracias a que en esta revisión del diseño se tiene una única memoria, tanto para programa como para datos. La síntesis de la nueva CPU ha resultado en una ocupación de la FPGA en la que se tienen 16 flip-flops menos, tal como se esperaba al eliminar los dos registros RH, y 11 celdas lógicas menos.

En cuanto a la incompatibilidad del software de la nueva CPU podemos indicar que es un problema poco preocupante por dos motivos: El primero es la pura escasez de programas para estas CPUs, y el segundo el hecho de que los códigos de una CPU se pueden reconvertir a los de la otra de forma automática gracias a la siguiente equivalencia:

6.2 Interrupciones concatenadas

En el diseño anterior si la línea IRQ sigue estando activa cuando se retorna de una rutina de interrupción su valor se va a ignorar durante dos ciclos de reloj. Esto es: se ejecuta un ciclo del programa principal antes de volver a pasar al modo interrupción (el segundo ciclo se invalida y se ejecuta como un NOP al tener la señal IRQSTART activa). Sería mejor comprobar el valor de IRQ en el momento de ejecutar RETI y evitar retornar al modo normal si una interrupción sigue pendiente, lo que reduciría en dos ciclos la latencia de la interrupción. La mejora de rendimiento que se podría conseguir con esta modificación no es que sea muy sustancial, pues las rutinas de interrupción suelen durar mucho más de dos ciclos, pero como el hardware adicional que supone ha resultado ser casi insignificante, tan sólo dos puertas AND de dos entradas, me he decidido a implementarla.

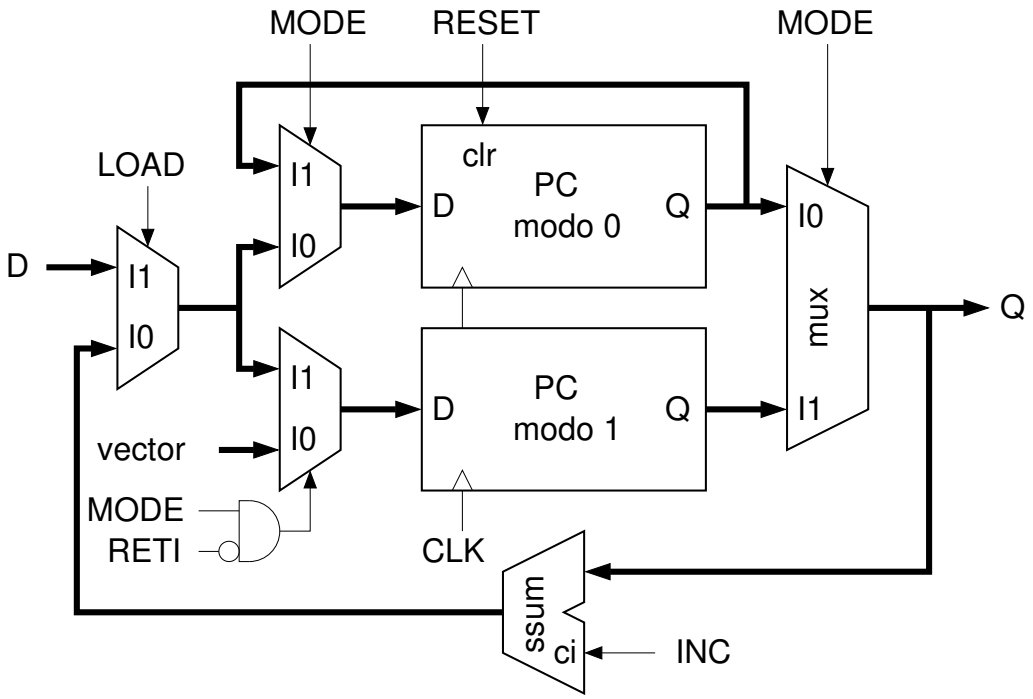


Figure 19: Esquema del registro PC doble modificado para la concatenación de interrupciones.

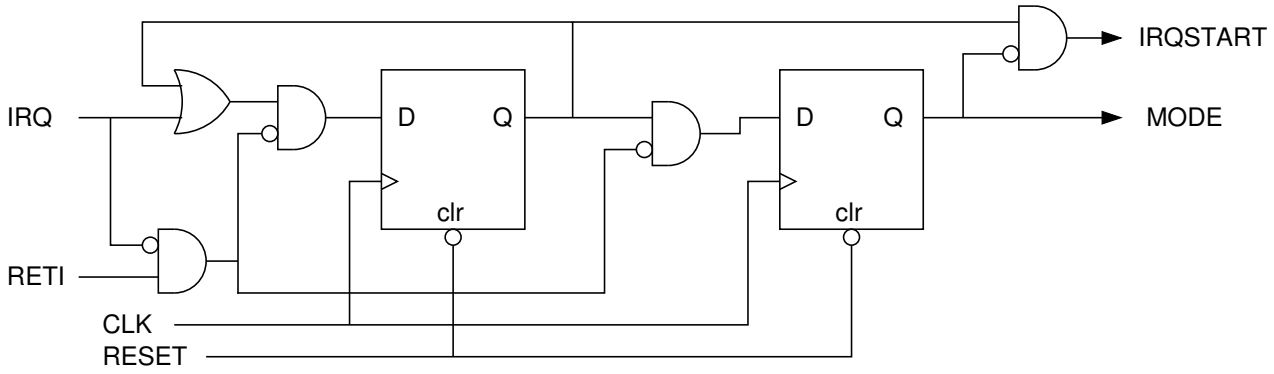


Figure 20: Lógica de control de interrupciones modificada para dar soporte a las interrupciones concatenadas.

Tan sólo ha sido necesario modificar dos bloques. Uno de ellos es el del registro PC doble, que ahora incorpora una entrada RETI adicional. Esta señal procede del decodificador y se activa cuando se ejecuta la instrucción RETI. Su efecto en el registro es el de cargar el PC del modo IRQ con el valor del vector de interrupción cuando RETI está activa, de modo que la ejecución puede seguir con la siguiente rutina de interrupción pendiente. Recordemos que RETI se ejecutaba como una instrucción JIND que escribía un valor incorrecto en el PC del modo IRQ, aunque eso no nos importaba pues al pasar al modo normal el PC del modo IRQ se volvía a cargar con su correspondiente vector de interrupción. Ahora no se va a conmutar al modo normal cuando se encadenan interrupciones y es necesario que RETI cargue el vector correcto en el registro PC del modo 1. Esto se ha conseguido haciendo que la selección del multiplexor de entrada al PC del modo 1 proceda de la AND de las señales MODE y el complemento de RETI, tal como podemos ver en la figura 19.

El segundo bloque que ha requerido una modificación ha sido el de la lógica de interrupciones (ver figura 20). En este caso se ha añadido una puerta AND en la entrada RETI de modo que sólo se resetean los flip-flops y se vuelve al modo normal cuando se ejecuta RETI sin tener IRQ activa. Si IRQ vale 1 cuando se ejecuta RETI no va a haber ningún cambio en la señal MODO, con lo que seguiremos estando en el modo de interrupción.

Con tan sólo dos puertas AND conseguimos tener interrupciones concatenadas, lo que reduce ligeramente la latencia de interrupción en sistemas con varias fuentes de interrupción activas. Sin embargo ahora hemos de ser más cuidadosos en nuestro código y asegurarnos de borrar siempre las peticiones de interrupción de los

periféricos en sus correspondientes rutinas de interrupción antes de retornar con RETI, pues de lo contrario se van a concatenar estas interrupciones de forma indefinida y la ejecución no va a retornar nunca al programa principal.

7 Quinta variante: Load y Store con desplazamiento inmediato

Esta modificación de la CPU surge después de analizar el juego de instrucciones del procesador RISC-V en el que la gestión de la pila se hace de un modo muy similar al de la CPU GUS16. La principal diferencia es la posibilidad de sumar al registro puntero un desplazamiento codificado como constante en los códigos de operación de las instrucciones Load y Store, algo que se echa de menos en la CPU GUS16. Sin embargo, si analizamos los códigos de operación de las instrucciones LD y ST (ver figuras 16 y 21) vemos que ambas tienen 4 bits sin usar y que se podrían emplear para codificar un desplazamiento de 4 bits, lo que nos permitiría acceder directamente a posiciones de memoria ubicadas hasta 15 palabras por encima de la dirección del puntero. Esto no parece gran cosa, pero sin embargo puede resultar muy útil en los programas ya que nos evitaría tener que estar continuamente cambiando el valor de los punteros, especialmente en el caso del puntero de pila. El mayor inconveniente es que los bits no usados en la instrucción ST no coinciden con la ubicación habitual de las constantes inmediatas (los de la instrucción LD sí coinciden), aunque esto se puede solventar fácilmente mediante un multiplexor para los 3 bits descolocados.

En las versiones viejas del ensamblador los bits no utilizados de las instrucciones LD y ST estaban codificados como ceros, de modo que en la CPU nueva esas instrucciones tendrán unos desplazamientos de cero palabras, con lo que se van a ejecutar exactamente igual que en la versión antigua. Por lo tanto no se va a generar ninguna incompatibilidad con el software existente, si bien sería conveniente reescribir los programas antiguos para aprovechar las posibilidades de las nuevas instrucciones LD y ST.

Lo cierto es que estas nuevas instrucciones han supuesto una reestructuración importante de los componentes de la CPU, en particular en la parte de la unidad de ejecución, la unidad de fetch no ha cambiado (ver figuras 18 y 22). Los bloques internos siguen siendo los mismos, pero su interconexión ahora es diferente. Hay que destacar los siguientes cambios:

- La dirección de la memoria durante la ejecución de las instrucciones LD y ST se obtiene de la salida de la ALU en lugar del bus A del banco de registros. Esto es necesario para realizar la suma del desplazamiento a la dirección base que se presenta en el bus A.
- En las instrucciones LD (y también en LDPC) el dato procedente de la memoria se lleva directamente al bus D del banco de registros para su escritura sin pasar por la ALU. Anteriormente la ALU dejaba pasar a su través el dato sin modificar, pero ahora la ALU se está usando para calcular la dirección.
- El decodificador se ha modificado para activar la señal IMM en las instrucciones LD y ST, al igual que para seleccionar una suma en la función de la ALU.
- La instrucción LD modifica los flags Z y N, pero como el dato cargado ahora no pasa por la ALU ha habido que mover la lógica de estos flags fuera de la ALU para poder probar el dato presente en el bus D.
- Finalmente, también ha sido necesario modificar el bloque de lógica IMM para que cuando se ejecute la instrucción ST se obtenga el desplazamiento de los bits que le corresponden en el código de operación en lugar de los habituales (figura 23)

En resumen: la inclusión de los desplazamientos en las instrucciones LD y ST ha supuesto una restructuración del procesador, pero el único bloque adicional es el multiplexor de 6 a 3 líneas de la figura 23, lo que supone realmente muy poca lógica adicional. Por otra parte he de reconocer que me preocupaba el retardo

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	mnemónico	flags				operación
0	0	0	0	0	RD			0	RA			–	RB			ADD	C	V	N	Z	RD= RA+RB
0	0	0	0	0	RD			1	RA			dato				ADDI	C	V	N	Z	RD= RA+dato
0	0	0	0	1	RD			0	RA			–	RB			ADC	C	V	N	Z	RD=RA+RB+Cflag
0	0	0	0	1	RD			1	RA			dato				ADCI	C	V	N	Z	RD=RA+dato+Cflag
0	0	0	1	0	RD			0	RA			–	RB			SUB	C	V	N	Z	RD=RA−RB
0	0	0	1	0	RD			1	RA			dato				SUBI	C	V	N	Z	RD=RA−dato
0	0	0	1	1	RD			0	RA			–	RB			SBC	C	V	N	Z	RD=RA−RB−(~Cflag)
0	0	0	1	1	RD			1	RA			dato				SBCI	C	V	N	Z	RD=RA−dato−(~Cflag)
0	0	1	0	0	–	–	–	0	RA			–	RB			CMP	C	V	N	Z	RA−RB
0	0	1	0	0	–	–	–	1	RA			dato				CMPI	C	V	N	Z	RA−dato
0	0	1	0	1	RD			0	RA			–	RB			AND	–	–	N	Z	RD=RA&RB
0	0	1	0	1	RD			1	RA			dato				ANDI	–	–	N	Z	RD=RA&dato
0	0	1	1	0	–	–	–	0	RA			–	RB			TST	–	–	N	Z	RA&RB
0	0	1	1	0	–	–	–	1	RA			dato				TSTI	–	–	N	Z	RA&dato
0	0	1	1	1	RD			0	RA			–	RB			OR	–	–	N	Z	RD=RA|RB
0	0	1	1	1	RD			1	RA			dato				ORI	–	–	N	Z	RD=RA|dato
0	1	0	0	0	RD			0	RA			–	RB			XOR	–	–	N	Z	RD=RA^RB
0	1	0	0	0	RD			1	RA			dato				XORI	–	–	N	Z	RD=RA^dato
0	1	0	0	1	RD			0	–	–	–	–	RB			NOT	–	–	N	Z	RD=~Rb
0	1	0	0	1	RD			1	–	–	–	–	RB			NEG	–	–	N	Z	RD=−RB
0	1	0	1	0	RD			0	–	–	–	–	RB			SHR	C	?	N	Z	RD=RB/2, Cflag=RB.0
0	1	0	1	0	RD			1	–	–	–	–	RB			SHRA	C	?	N	Z	RD=RB/2, Cflag=RB.0 (con signo)
0	1	0	1	1	RD			0	–	–	–	–	RB			ROR	C	?	N	Z	RD=(RB>>1)|(Cflag<<15), Cflag=RB.0
0	1	0	1	1				1								ILEG					
0	1	1	0	0	RD			0	RA			d[3:0]				LD	–	–	N	Z	RD=Mem[RA+d]
0	1	1	0	0	d[2:0]			1	RA			d[3]	RB			ST	–	–	–	–	Mem[RA+d]=RB
0	1	1	0	1	RD			0	–	–	–	dato				ADPC	–	–	–	–	RD=PC+dato
0	1	1	0	1	–	–	–	1	–	–	–	0	RB			JIND	–	–	–	–	PC=RB
0	1	1	0	1	–	–	–	1	–	–	–	1	–	–	–	RETI	–	–	–	–	Retorna de Interrupcion
0	1	1	1	0	RD			–	–	–	–	–	–	–	–	LDPC	–	–	–	–	RD=Mem[PC++]
0	1	1	1	1	RD			dato								LDI	–	–	–	–	RD=dato (8 bits)
1	0	0	0	desplazamiento con signo												JZ	–	–	–	–	salto si Zflag=1
1	0	0	1	desplazamiento con signo												JNZ	–	–	–	–	salto si Zflag=0
1	0	1	0	desplazamiento con signo												JC	–	–	–	–	salto si Cflag=1
1	0	1	1	desplazamiento con signo												JNC	–	–	–	–	salto si Cflag=0
1	1	0	0	desplazamiento con signo												JMI	–	–	–	–	salto si Nflag=1 (negativo)
1	1	0	1	desplazamiento con signo												JPL	–	–	–	–	salto si Nflag=0 (positivo)
1	1	1	0	desplazamiento con signo												JV	–	–	–	–	salto si Vflag=1 (overflow)
1	1	1	1	desplazamiento con signo												JR	–	–	–	–	salto incondicional

Figure 21: Tabla con los códigos de operación de las instrucciones para la CPU V5 en la que se han resaltado las nuevas instrucciones LD y ST.

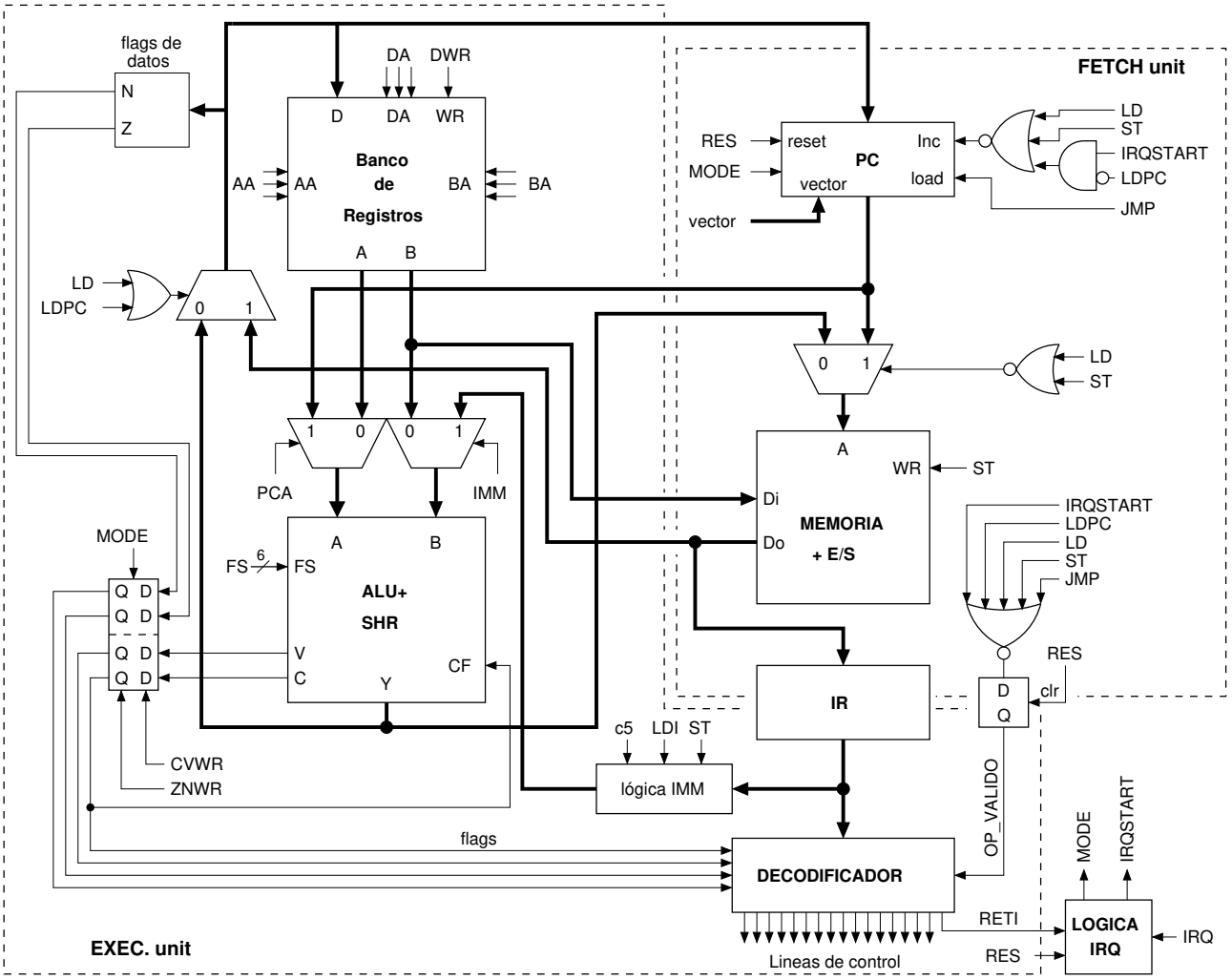


Figure 22: Diagrama de bloques de la CPU v5.

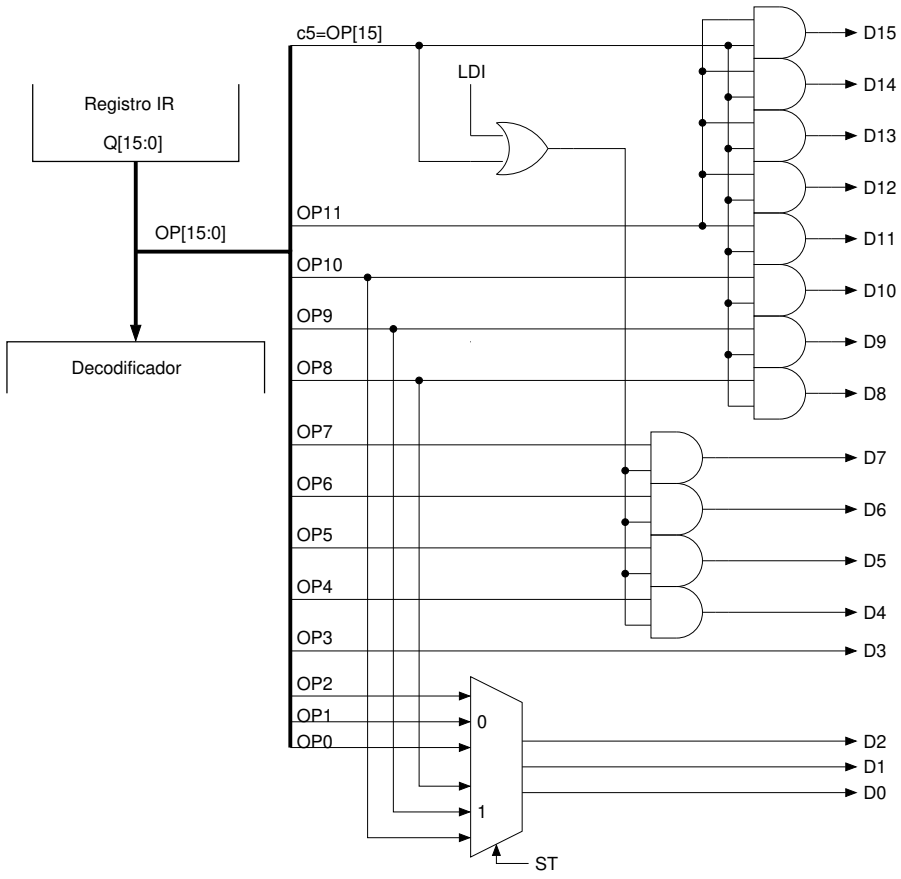


Figure 23: Lógica para la carga de constantes inmediatas en la CPU v5.

de propagación que introduce la ALU en la generación de las direcciones de memoria, pero tras una síntesis en FPGA he podido comprobar que la frecuencia máxima de reloj no ha empeorado nada, lo que me lleva a pensar que el acceso a la memoria no era el retardo crítico del procesador.

La primera prueba de la nueva CPU ha consistido en verificar que las aplicaciones existentes siguen corriendo igual que en la versión antigua. Luego ha venido el trabajo realmente pesado que ha consistido en la actualización del ensamblador para la inclusión de la nueva sintaxis de las instrucciones LD y ST, y también la actualización de la documentación antes de que los detalles queden en el olvido. Con las nuevas instrucciones ahora se pueden optimizar los accesos a memoria, como por ejemplo cuando se guardan datos en la pila:

```
Antes:
IRQ1:  subi r7,r7,1 ; save regs
       st   (r7),r0
       subi r7,r7,1
       st   (r7),r1
       subi r7,r7,1
       st   (r7),r2
       ...
       ld   r2,(r7); restore regs
       addi r7,r7,1
       ld   r1,(r7)
       addi r7,r7,1
       ld   r0,(r7)
       addi r7,r7,1
```

```
Después:
IRQ1:  subi r7,r7,3 ; save regs
       st   (r7+2),r0
       st   (r7+1),r1
       st   (r7),r2
       ...

       ld   r2,(r7); restore regs
       ld   r1,(r7+1)
       ld   r0,(r7+2)
       addi r7,r7,3
       reti
```

```
        reti
```

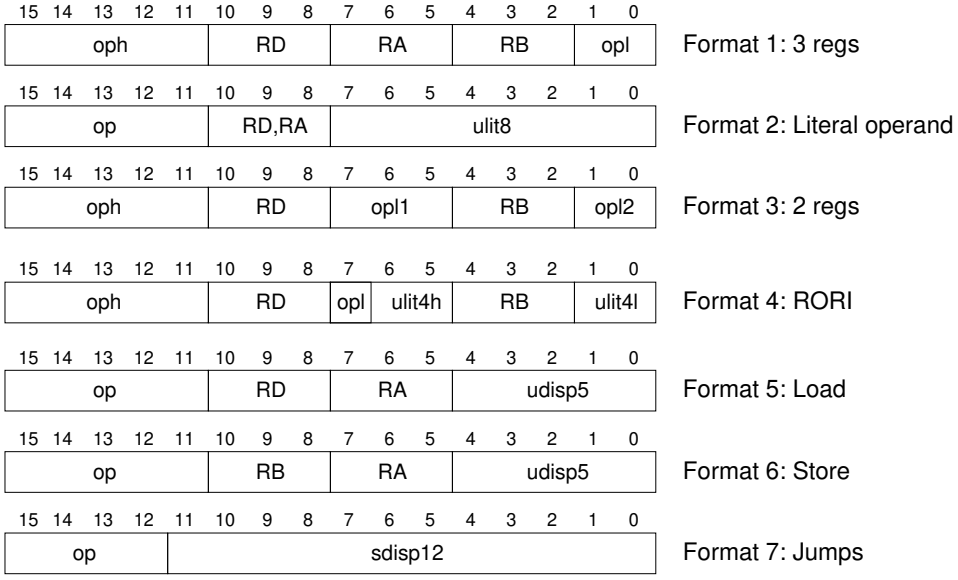
En este ejemplo de código podemos comprobar cómo a pesar de disponer de desplazamientos sólo positivos, y limitados a un valor máximo de 15, nuestro código puede mejorar notablemente gracias a las nuevas instrucciones LD y ST (además de parecerse sospechosamente al código del pocesador RISC-V ;)

8 Sexta variante. Nuevo juego de instrucciones

Unos cuantos miles de líneas de código fuente en el ensamblador de este micro has sido suficientes para reconocer que serían convenientes algunos cambios en el procesador. La versión V6 incluye un juego de instrucciones completamente nuevo. En concreto, hay que destacar:

- Llamada a subrutinas con una única instrucción, Jump And Link (JAL). Esta instrucción graba el valor del PC en R6 a la vez que calcula la dirección del salto. (El registro de enlace es un parámetro en el archivo fuente del procesador. Por defecto se usa R6, pero podría ser cualquier otro registro)
- Operandos inmediatos de 8 bit en lugar de 4. Esto se ha conseguido a costa de hacer que el registro fuente, RA, y el destino, RD, sean el mismo, y de eliminar algunas instrucciones.
- Rotaciones de número de bits arbitrario, RORI. Esta nueva instrucción ha sido posible gracias a la inclusión de un "barrel-shifter" en la salida de la ALU. Lo cierto es que inicialmente pensaba incluir tan solo una instrucción SWAP para intercambiar los bytes de los registros, pues era algo que realmente se necesitaba para manejar datos de 8 bit. Una instrucción SWAP requeriría un multiplexor de 32 entradas a 16, pero junto con el ya existente para las instrucciones del tipo SHR equivaldrían a medio "barrel shifter", así que me he decidido a incluir el desplazador completo junto con una instrucción, RORI, que permita explotar sus posibilidades.
- En el nuevo repertorio de instrucciones he tenido que eliminar algunas de las existentes, bien porque ahora serían redundantes, o porque simplemente no quedaban suficientes combinaciones de bits para incluir todas las instrucciones posibles. Ahora no tenemos:
  - ADPC rd,n. Su principal uso era para la llamada a subrutinas y ahora tenemos JAL. Si simplemente quisiéramos copiar el valor del PC a un registro podemos seguir usando JAL: " JAL .+1" salta a la siguiente instrucción copiando el valor del PC a R6.
  - CMP. Se puede sustituir por SUB cuidando de dejar el resultado en un registro sin uso. La instrucción CMPI se conserva puesto que se usa muy frecuentemente.
  - TST, TSTI. Se pueden sustituir por AND o ANDI, si bien ANDI modifica el registro que queremos probar.
  - ROR. Rotación de un bit incluyendo acarreo. Ahora se llama RORC para distinguirla de RORI que no incluye el acarreo.

El formato de los nuevos códigos de operación es el siguiente:

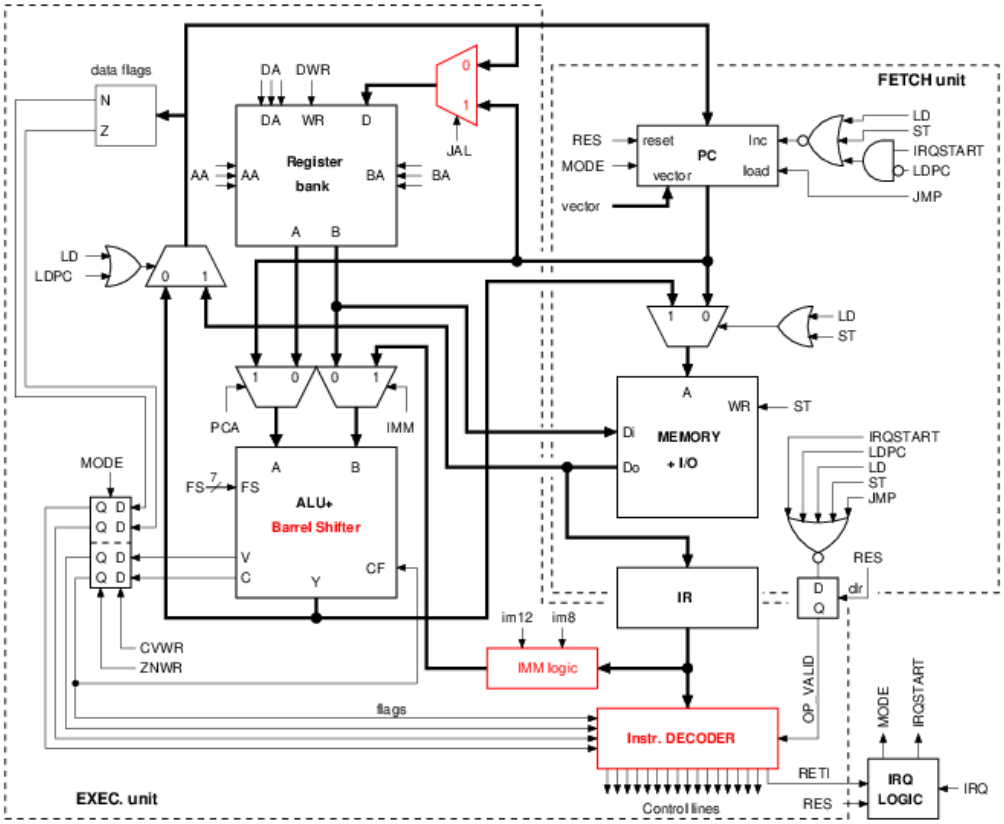


Un cambio respecto de la codificación anterior es el hecho de no tener siempre la selección de los registros en los mismos bits. Así, en las instrucciones con operando inmediato (formato 2) la selección del registro RA se hace con los mismos bits de RD, y en la instrucción ST el registro RB usa los bits de RD. También tenemos el caso de JAL, que lleva implícito el registro R6 en la selección de RD. Todo esto nos obliga a usar multiplexores para la selección de los valores adecuados a cada registro dependiendo del código de operación de la instrucción.

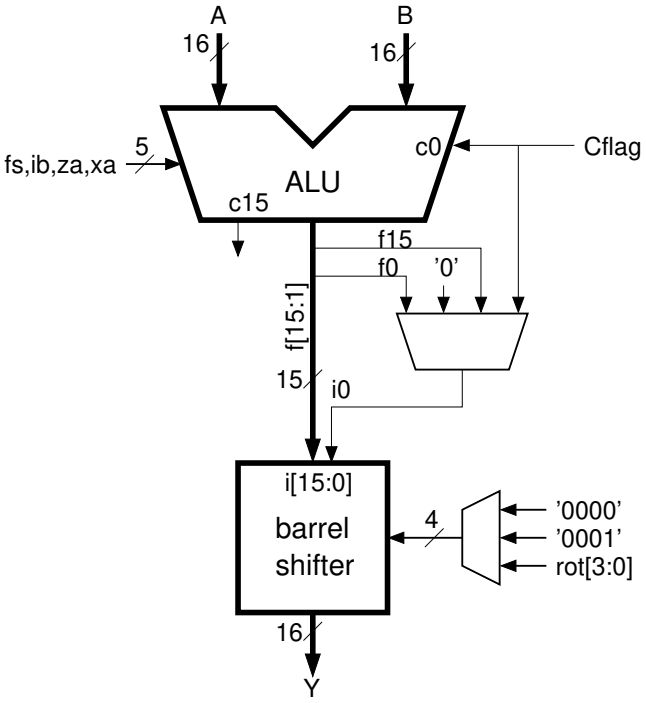
La tabla completa con las instrucciones es:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	mnemonic	flags				operation
0	0	0	0	0	RD			RA			RB			0	0	ADD	C	V	N	Z	RD= RA + RB
0	0	0	0	0	RD			RA			RB			0	1	SUB	C	V	N	Z	RD= RA – RB
0	0	0	0	0	RD			RA			RB			1	0	ADC	C	V	N	Z	RD= RA + RB+Cflag
0	0	0	0	0	RD			RA			RB			1	1	SBC	C	V	N	Z	RD= RA– RB –~cflag
0	0	0	0	1	RD			RA			RB			0	0	AND	–	–	N	Z	RD= RA & RB
0	0	0	0	1	RD			RA			RB			0	1	OR	–	–	N	Z	RD= RA | RB
0	0	0	0	1	RD			RA			RB			1	0	XOR	–	–	N	Z	RD= RA ^ RB
0	0	0	0	1	RD			RA			RB			1	1	BIC	–	–	N	Z	RD= RA & (~RB)
0	0	0	1	0	RD			ulit8								ADDI	C	V	N	Z	RD= RD + ulit8
0	0	0	1	1	RD			ulit8								SUBI	C	V	N	Z	RD= RD – ulit8
0	0	1	0	0	RD			ulit8								ADCI	C	V	N	Z	RD= RD + ulit8 +Cflag
0	0	1	0	1	RD			ulit8								SBCI	C	V	N	Z	RD= RD – ulit8 –~Cflag
0	0	1	1	0	RD			ulit8								ANDI	–	–	N	Z	RD= RD & ulit8
0	0	1	1	1	RD			ulit8								ORI	–	–	N	Z	RD= RD | ulit8
0	1	0	0	0	RD			ulit8								XORI	–	–	N	Z	RD= RD ^ ulit8
0	1	0	0	1	RD			ulit8								CMPI	C	V	N	Z	RD – ulit8
0	1	0	1	0	RD			ulit8								LDI	–	–	–	–	RD= ulit8
0	1	0	1	1	RD			0	ulit4h			RB			ulit4l	RORI	–	–	N	Z	RD = (RB>>uli4) | (RB<<16–ulit4)
0	1	0	1	1	RD			1	0	0	RB			0	0	RORC	C	?	N	Z	RD = {Cflag,RB>>1}, Cflag=RB0
0	1	0	1	1	RD			1	0	0	RB			0	1	SHR	C	?	N	Z	RD = RB>>1, Cflag=RB0
0	1	0	1	1	RD			1	0	0	RB			1	0	SHRA	C	?	N	Z	RD = RB>>1 (signed) , Cflag=RB0
0	1	0	1	1	RD			1	0	1	RB			0	0	NOT	–	–	N	Z	RD = ~RB
0	1	0	1	1	RD			1	0	1	RB			0	1	NEG	C	V	N	Z	RD = –RB
0	1	0	1	1	RD			1	1	1	–	–	–	0	0	LDPC	–	–	–	–	RD = Mem(PC++)
0	1	0	1	1	–	–	–	1	1	1	RB			1	0	JIND	–	–	–	–	PC = RB
0	1	0	1	1	–	–	–	1	1	1	–	–	–	1	1	RETI	–	–	–	–	return from interrupt
0	1	1	0	0	RD			RA			udisp5					LD	–	–	N	Z	RD= Mem(RA+udisp5)
0	1	1	0	1	RB			RA			udisp5					ST	–	–	–	–	Mem(RA+udisp5)=RB
0	1	1	1	sdisp12												JAL	–	–	–	–	PC = PC+sdisp12, Rlink=PC
1	0	0	0	sdisp12												JZ	–	–	–	–	PC = PC+sdisp12 if Zflag
1	0	0	1	sdisp12												JNZ	–	–	–	–	PC = PC+sdisp12 if ~Zflag
1	0	1	1	sdisp12												JC	–	–	–	–	PC = PC+sdisp12 if Cflag
1	0	1	1	sdisp12												JNC	–	–	–	–	PC = PC+sdisp12 if ~Cflag
1	1	0	0	sdisp12												JMI	–	–	–	–	PC = PC+sdisp12 if Nflag
1	1	0	1	sdisp12												JPL	–	–	–	–	PC = PC+sdisp12 if ~Nflag
1	1	1	0	sdisp12												JV	–	–	–	–	PC = PC+sdisp12 if Vflag
1	1	1	1	sdisp12												JR	–	–	–	–	PC = PC+sdisp12

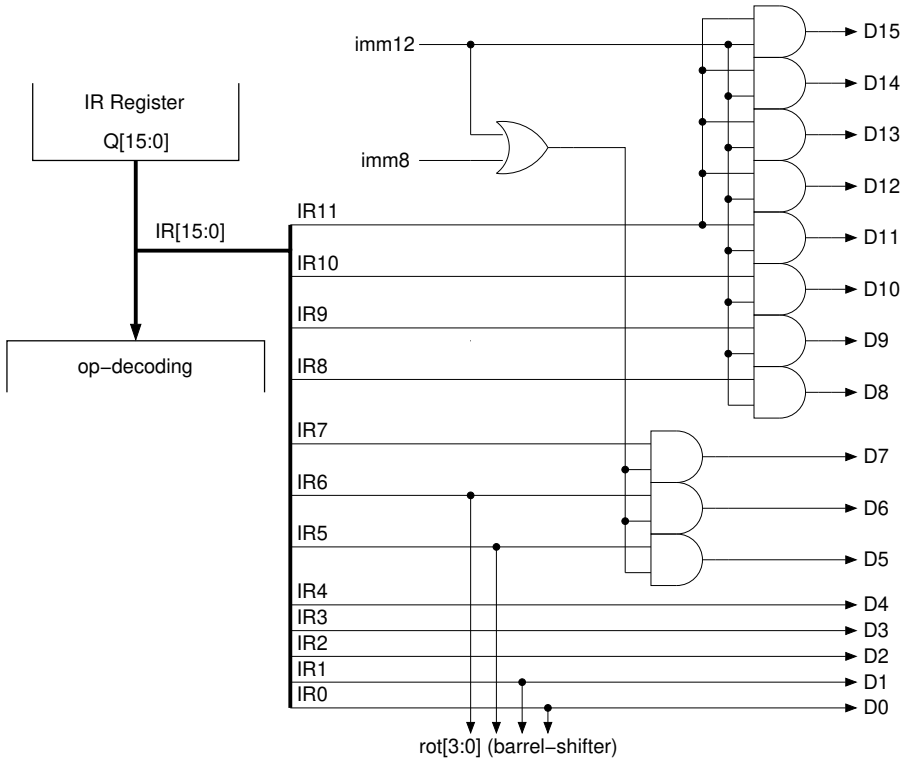
El diagrama de bloques del nuevo procesador se muestra en la siguiente figura, en la que los bloques nuevos o modificados se señalan en color rojo. El bloque con más cambios es sin duda el decodificador de instrucciones, seguido por la ALU. El bloque de operandos inmediatos ha cambiado poco.



La ALU ahora va seguida de un bloque combinacional capaz de rotar hacia la derecha el número de bits que se indiquen en sus 4 entradas de control. Este circuito, conocido habitualmente como "barrel-shifter" se construye internamente como 4 multiplexores en serie, cada uno de ellos de 32 entradas y 16 salidas. El primero elige los datos sin rotar, o rotados 8 bits hacia la derecha. El segundo elije los datos sin rotar o rotados 4 bits hacia la derecha, y así hasta el último que rota sólo un bit. El rotador se encarga de ejecutar la instrucción RORI, pero también las instrucciones SHR, SHRA, y RORC. Estas últimas son en realidad desplazamientos en los que el bit LSB del dato de entrada al rotador ha de elegirse entre un valor de 0 para SHR, el bit de signo para SHRA, o el flag de acarreo para RORC. El resto de las instrucciones, incluyendo RORI, necesitan que este bit sea simplemente el bit LSB de la salida de la ALU. Asimismo, el número de bits a rotar se fuerza como uno en las instrucciones de desplazamiento, proviene de los operandos inmediatos para RORI, o es cero para el resto de las instrucciones.



El bloque de los operandos inmediatos se muestra en la siguiente figura, en la que vemos que es muy similar al de la versión V5. Su función se limita a rellenar los bits MSB de los operandos inmediatos con cero o con el bit de signo en el caso de las instrucciones de salto.



Tras estas modificaciones estos son los resultados de la síntesis en una FPGA del tipo Lattice ICE40Hx:

	celdas lógicas
GUS16-V6	760
GUS16-V5	673
diferencia	87

El rotador supone 64 celdas lógicas a mayores, aunque sustituye a un multiplexor de 16 celdas. Pero también se ha añadido otro de estos multiplexores para dar soporte a la instrucción JAL, de modo que las

nuevas celdas lógicas son en realidad las 64 del "barrel-shifter" más 23 celdas lógicas repartidas entre el resto de la lógica añadida y las modificaciones del bloque decodificador.

En cuanto a la mejora en el software, una primera estimación tras la conversión de un programa algo complejo ("Floppyton", ~2400 líneas en ensamblador) es de un 12% de reducción en el número de instrucciones respecto del mismo código con el juego de instrucciones de la versión V5. Una buena parte de esta reducción proviene de la instrucción JAL, seguida de RORI, que simplifica notablemente las operaciones aritméticas y de manejo de bytes ("RORI Rd,Rb,8" equivale a intercambiar los bytes alto y bajo de Rb), y también los nuevos operandos inmediatos de 8 bits que ayudan a eliminar un buen número de instrucciones LDI que anteriormente se necesitaban si las constantes eran mayores que 15.

9 Más que CPU: Microcontrolador

La CPU descrita anteriormente ocupa aproximadamente la mitad de las celdas lógicas de una FPGA del tipo Lattice ICE40HX1K, que no destaca precisamente por ser una FPGA muy grande ni sofisticada. Para poder probar el correcto funcionamiento de la CPU se necesitan más componentes que afortunadamente aún pueden ser sintetizados dentro de la FPGA. En concreto, necesitaremos una memoria y un conjunto mínimo de periféricos.

La FPGA mencionada incluye un total de 64Kbit de memoria RAM que la CPU no usa para nada y que nos pueden proporcionar hasta 4Kx16 palabras de memoria interna. Además el contenido inicial de esta memoria se puede almacenar con el resto de los datos de configuración de la FPGA, lo que nos permite prescindir de una memoria ROM de arranque para nuestra CPU.

En cuanto a los periféricos, desde un principio se pensó en incluir una UART simple. Más adelante, cuando la CPU incluía el soporte para interrupciones, también se incluyó un temporizador. Finalmente, para concluir el diseño de lo que ya sería un microcontrolador, se incluyeron también un puerto de entrada y salida de propósito general, y un puerto SPI con el objeto de aprovechar la propia memoria flash de configuración de la FPGA como almacenamiento adicional de datos.

Basándonos en las consideraciones anteriores nos hemos propuesto diseñar un microcontrolador como el de la figura 24. Aquí hemos añadido además una interfaz para memoria externa para poder expandir el espacio de direcciones hasta los 64K posibles, aunque seguimos incluyendo los 4K de la memoria interna como nuestra memoria primaria. Observemos que la memoria externa tiene un bus de datos bidireccional, por lo cual esta requiere el uso de las celdas triestado de los pines de la FPGA (SB_IO).

En primer lugar hemos de pensar cómo ubicar todos los componentes de este sistema en el espacio de direcciones de memoria del que disponemos. Finalmente me he decidido por un mapa de memoria como el mostrado en la figura 25, en el que las primeras 32 direcciones están ubicadas en la memoria interna y se usarán para albergar el código de reset y de los vectores de interrupción (4 direcciones/vector). A continuación tenemos otras 32 direcciones que redirigen las lecturas y escrituras hacia los periféricos. Desde la dirección 64 (0x40) hasta la 4095 (0x0FFF) se vuelve a seleccionar la RAM interna, y para direcciones mayores de 4095 se selecciona la memoria externa.

El bloque de periféricos ha parecido conveniente ubicarlo por debajo de la dirección 0x100 para poder cargar la dirección de los registros de los periféricos con instrucciones 'LDI' simples en lugar de tener que usar secuencias 'LDPC, word'. Esto nos resta un poco de la memoria interna (32 posiciones), pero puede que realmente nos permita ahorrar memoria en cuanto los programas sean un poco complejos gracias a las instrucciones 'LDPC' que ahora son innecesarias.

Tengo que mencionar que la decodificación de las direcciones no es del todo completa pues, por ejemplo, una escritura en la dirección 0x22 además de escribir un dato para transmitir en la UART también lo escribe en la memoria interna y también en la externa. Del mismo modo una lectura de la misma dirección lee también

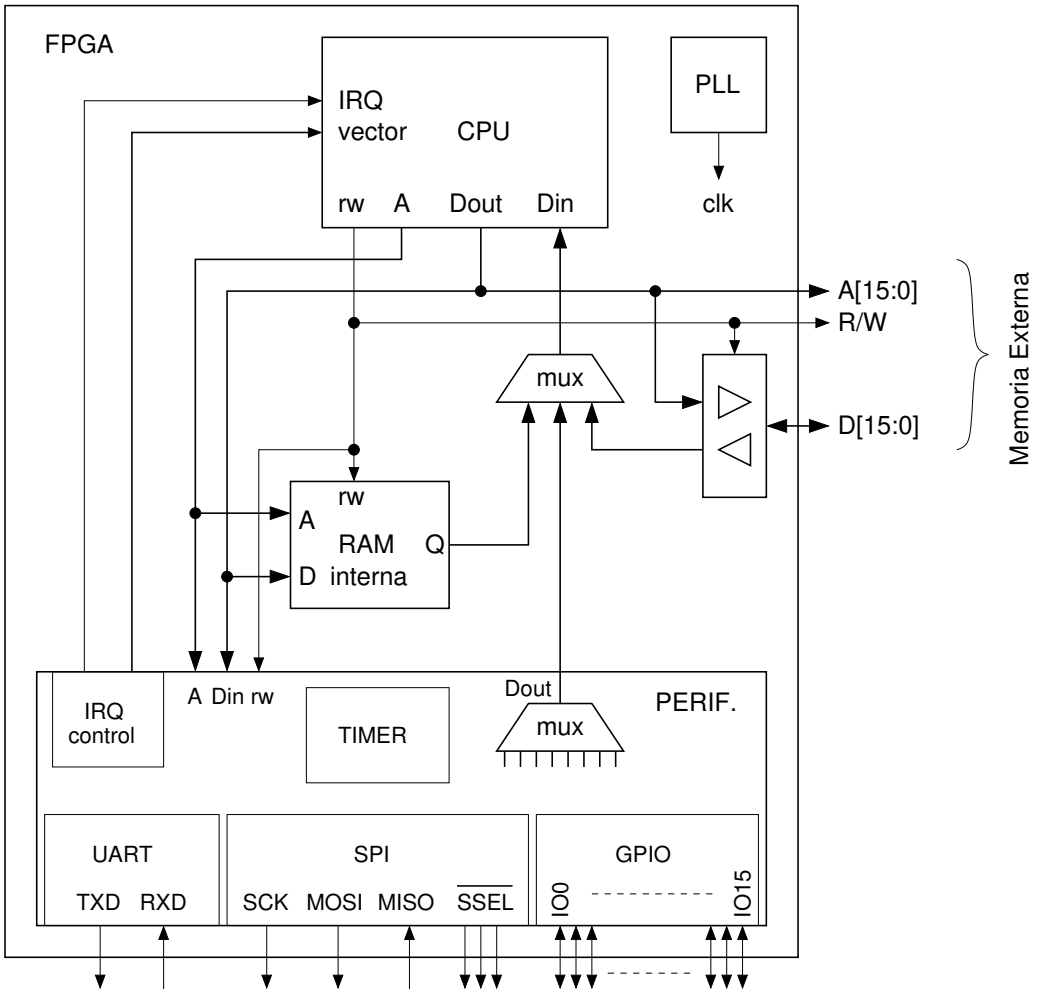


Figure 24: Diagrama simplificado del diseño del microcontrolador propuesto.

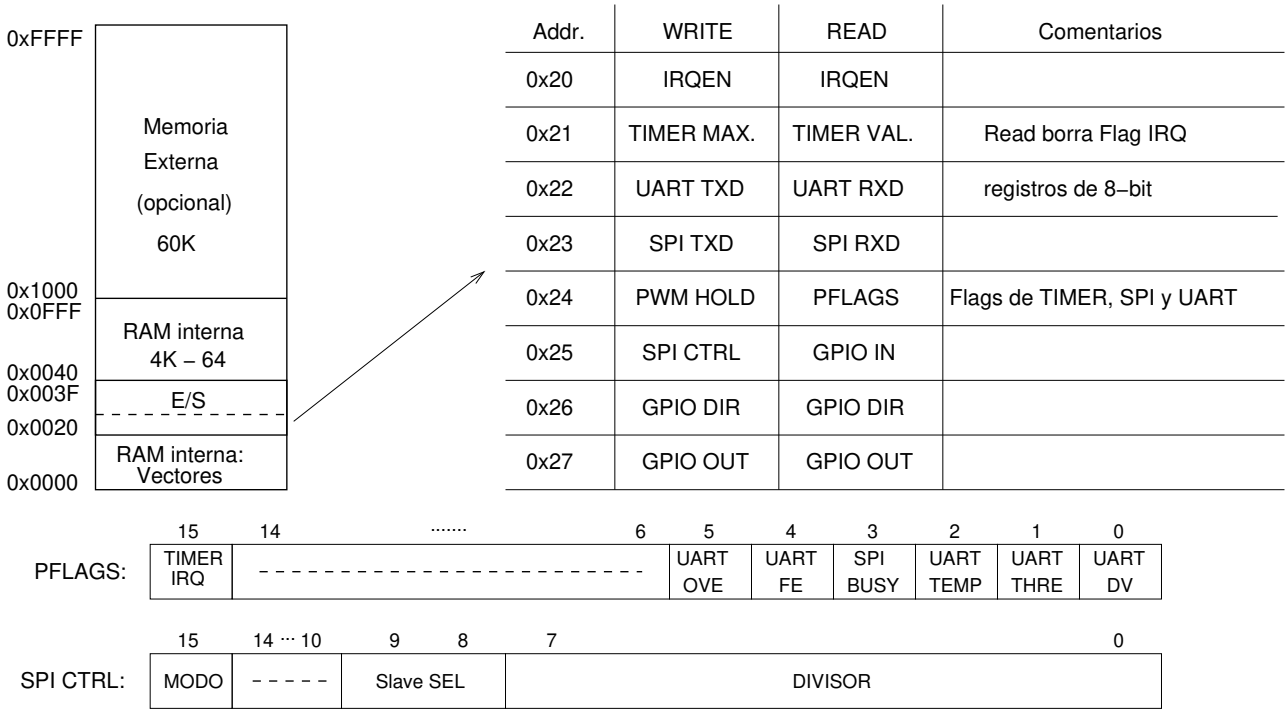


Figure 25: Mapa de la memoria, del bloque de periféricos, y detalle de los registros de flags y control.

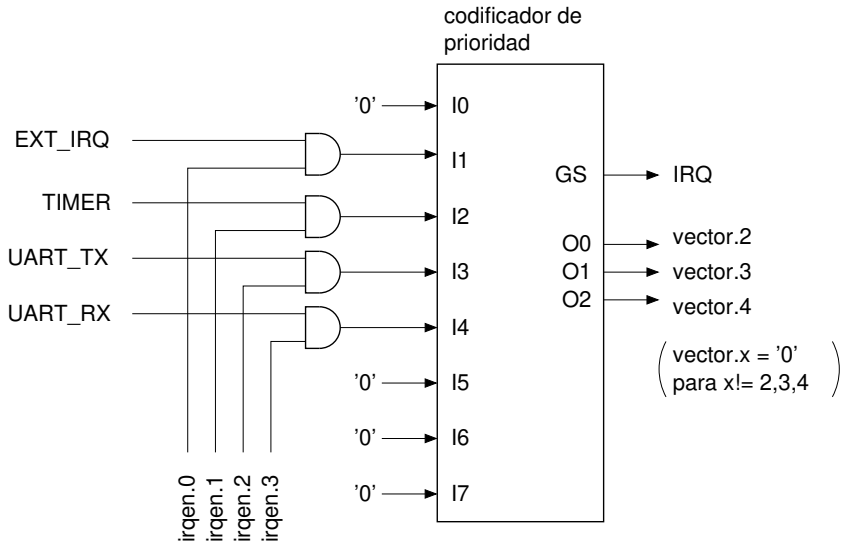


Figure 26: Lógica de vectorización de interrupciones

datos de todas las memorias, pero el dato que finalmente llega a la CPU es el procedente del receptor de la UART.

El mapa de registros de los periféricos se limita a 8 direcciones de las 32 posibles, y la misma dirección se puede usar con registros distintos si uno de ellos es de sólo lectura y otro de sólo escritura. Los flags de estado de todos los periféricos se han agrupado en un único registro, PFLAGS, de sólo lectura, cuyo contenido también se muestra en la figura 25.

A continuación vamos a describir los bloques de la sección de periféricos.

9.1 Controlador de interrupciones

El disponer de varios periféricos y fuentes de interrupción hace que sea conveniente el incluir un mecanismo de interrupciones vectorizadas. La lógica necesaria no ha sido mucha y se muestra en la figura 26, en la que observamos que disponemos de cuatro señales de petición de interrupción, que se pueden enmascarar de forma individual desde un registro IRQEN, y que se convierten en una señal de petición de interrupción global, IRQ, y un número de vector, gracias a un codificador de prioridad. Vemos que no se usa la entrada I0 del codificador ya que el vector que resultaría en tal caso sería el número cero y coincidiría con la dirección de RESET del micro. Con este conexionado la interrupción externa saltará a la dirección 0x0004, la del temporizador a la dirección 0x0008, la interrupción de listo para transmitir en la UART saltará a la dirección 0x000C, y la de dato recibido en la UART a la dirección 0x0010, siendo esta última la interrupción más prioritaria. El código Verilog correspondiente está repartido entre el módulo CPU y el del sistema, y sería:

```
// Interrupciones
wire irq;
wire [15:0]vector;
// Peticiones de interrupción. Enmascaradas por registro IRQEN
assign irqs[6:4]=3'b000;  // No implementadas
assign irqs[3] = dv       & irqen[3]; // UART RX
assign irqs[2] = thre     & irqen[2]; // UART TX av.
assign irqs[1] = timirq   & irqen[1]; // Timer
assign irqs[0] = (~eintb) & irqen[0]; // Pin externo
// Petición de interrupción global
assign irq = irqs[0]|irqs[1]|irqs[2]|irqs[3]|irqs[4]|irqs[5]|irqs[6];
```

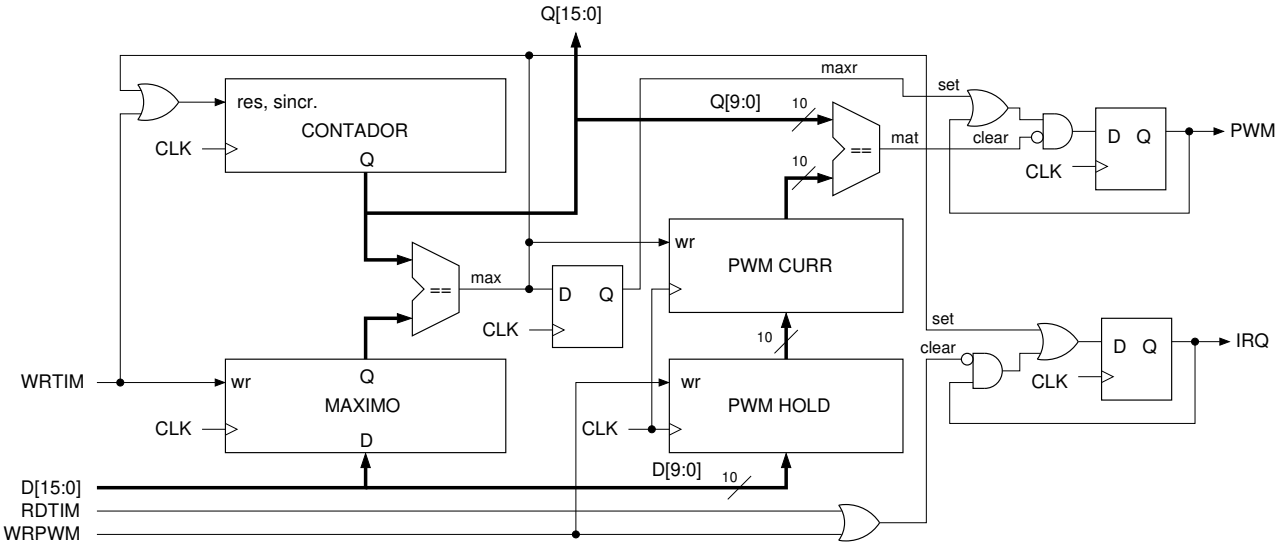


Figure 27: Diagrama del temporizador con la función adicional de modulación PWM

```
// Codificador de prioridad / vector
wire [2:0]nvector;
assign nvector = (irqs[6]?3'h7:(irqs[5]?3'h6:(irqs[4]?3'h5:(irqs[3]?3'h4:
                 (irqs[2]?3'h3:(irqs[1]?3'h2:(irqs[0]?3'h1:3'bxxx) ))))));
assign vector = {11'h000,nvector,2'b00};
```

El registro IRQEN se puede leer además de escribir y después de un reset tiene todos sus 4 bits en cero.

9.2 Temporizador y PWM

El esquema del temporizador implementado es el que se muestra en la figura 27, donde podemos considerar este esquemático como la fusión de dos circuitos: por una parte un temporizador de intervalos programables, y por otra un modulador PWM. El objeto del primer circuito, además de llevar la cuenta de los ciclos de reloj, es el de generar peticiones de interrupción periódicas a intervalos de tiempo programables. Para ello tenemos un contador de 16 bits que se va incrementando hasta que su valor coincide con el almacenado en un registro de máximo. Cuando esto ocurre, en el siguiente ciclo de reloj el contador se reinicia con un valor de cero y se activa la petición de interrupción.

Notemos que cuando se activa WRTIM se carga el dato presente el el bus de entrada al registro de máximo a la vez que hace cero el contador. Las lecturas, en cambio, nos devuelven el valor actual del contador y tienen el efecto adicional de borrar el flag de interrupción. Este flag se podrá consultar por programa en el registro PFLAGS (bit 15).

La parte de modulación de anchura de pulsos incluye un par de registros adicionales, HOLD y CURR, que en este caso son de sólo 10 bits, y que almacenan el valor de la anchura del pulso. El valor del registro CURR se compara con los 10 bits menos significativos del contador y cuando hay una coincidencia la salida PWM se pone en nivel bajo. Por otra parte, la señal PWM se pone en alto cuando el contador vale cero. En lugar de usar un comparador adicional para el valor cero he creído que sería más sencillo retardar un ciclo la señal de máximo del temporizador, lo que se hace con un simple flip-flop. Así la señal "maxr" se va a activar justo cuando el contador vale cero. Esta señal pone PWM en alto justo cuando se copia el contenido del registro HOLD en el registro CURR. HOLD puede escribirse en cualquier momento pero su valor se consulta sólo durante el ciclo en el que el contador pasa a valer cero. Así evitamos la generación de pulsos incorrectos en la salida si cambiamos el nivel de comparación durante el ciclo de la onda PWM.

Observemos que las señales "maxr" y "mat" pueden activarse simultáneamente si el registro CURR vale 0.

En este caso la señal “mat” es más prioritaria y la salida PWM se mantiene siempre en cero, (En la salida IRQ la señal “max” es la más prioritaria y pone IRQ en 1). También podemos observar que las escrituras en HOLD tienen el efecto adicional de borrar la petición de interrupción del temporizador.

El código Verilog del temporizador es el siguiente:

```
// Temporizador con salida PWM
// pero para PWM sólo se usan 10 bits
module TIMERPWM(
  input clk,      // Reloj principal
  input [15:0]d,  // Datos desde la CPU
  input wrtim,    // strobe de escritura en timer
  input wrpwm,    // strobe de escritura en PWM (borra IRQ)
  input rdtim,    // strobe de lectura (borra IRQ)
  output [15:0]q, // salida hacia CPU
  output irq,     // Salida de petición de interrupción
  output pwm      // Salida PWM
);
reg [15:0]timerval=0;    // Contador
reg [15:0]timermax=0;    // Valor máximo
reg [9:0] hold=0;        // registro temporal de valor
reg [9:0] curr=0;        // registro de valor del ciclo PWM actual
reg pwm=0;               // Salida PWM
reg irq=0;               // Salida de petición de interrupción
wire max;                // Final de cuenta
reg maxr;                // Final de cuenta retrasado 1 ciclo
wire mat;                // matching entre cont y curr
assign max = (timerval==timermax);
assign mat = (timerval[9:0]==curr);
always @ (posedge clk)
begin
    maxr <= max;
    timermax <= wrtim ? d : timermax;
    timerval <= (wrtim|max) ? 0 : timerval+1;
    hold <= wrpwm ? d[9:0] : hold;
    curr <= max ? hold : curr;
    irq  <= max ? 1 : ((wrpwm|rdtim) ? 0 : irq);
    pwm  <= mat ? 0 : (maxr ? 1 : pwm);
end
assign q = timerval;
endmodule
```

Para la generación de ondas PWM debemos tener en cuenta:

- El registro de máximo no debería tener un valor mayor de 1023. Este registro define el periodo de la onda PWM y si escribimos en él un valor mayor todavía seguimos generando una onda PWM pero la anchura del pulso nunca puede llegar a ser del 100% del periodo.
- Si queremos que la señal PWM esté continuamente en alto hay que hacer que el registro HOLD contenga

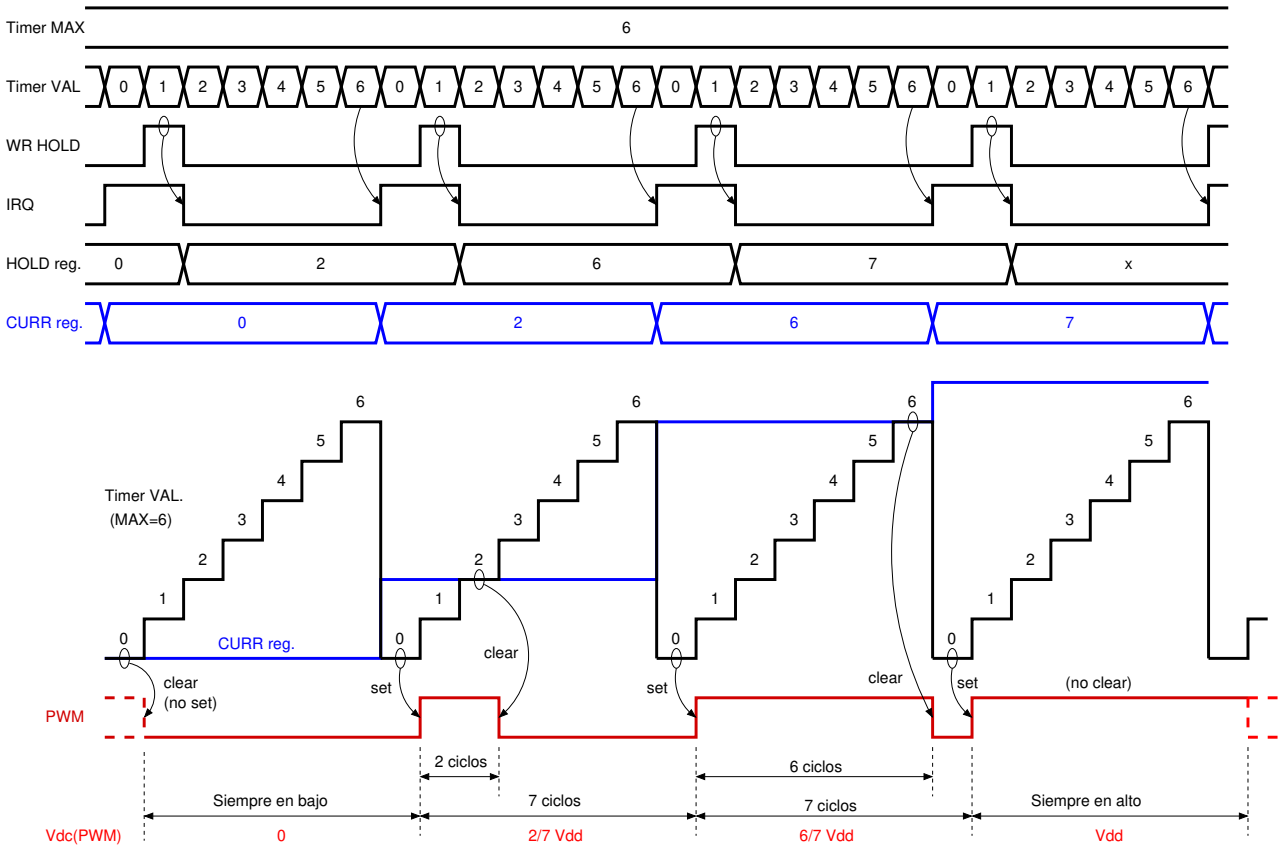


Figure 28: Ejemplo de generación de onda PWM de 3 bits de resolución.

un valor mayor que el registro de máximo, con lo que la señal "mat" nunca se activa y la salida se queda en alto. Si el contenido de estos dos registros es el mismo la salida PWM genera pulsos con un ciclo de reloj en bajo y MAX ciclos en alto. Por ejemplo, haciendo MAX=1022 podemos generar ondas con un nivel PWM desde 0/1023 (siempre en bajo) hasta 1023/1023 (siempre en alto).

- Si utilizamos interrupciones disponemos de todo un ciclo de la onda PWM para escribir el nuevo valor del la anchura de pulso en el registro HOLD. Al hacerlo borramos además la petición de interrupción, con lo que no necesitamos código adicional para el borrado del flag.
- El valor que escribimos en HOLD no va a afectar a la salida hasta que pase un ciclo adicional de la onda PWM.

Estas consideraciones se reflejan en el ejemplo de cronograma de la figura 28. Aquí el registro de máximo se ha escrito con el valor 6, de modo que el contador repite su secuencia cada 7 ciclos. En el registro de HOLD del nivel PWM podemos escribir los valores desde el cero, que equivale a una salida siempre en bajo, hasta el 7, que equivale a una salida siempre en alto. Los valores de HOLD mayores que 7 tienen el mismo efecto que el 7, lo que nos da un total de 8 niveles de salida posibles.

9.2.1 PWM con signo

Una modificación adicional ha consistido en añadir signo a la modulación PWM. Pensemos por ejemplo en el control de un motor de DC en el que además de la potencia queremos controlar el sentido de giro. Para ello usaremos un puente H como conmutador de potencia y aplicaremos la onda PWM en una de sus entradas o la contraria dependiendo del sentido de giro deseado. La entrada en la que no hay señal PWM se mantiene siempre a cero. Esto implica disponer de un bit adicional con el que representaremos el signo del nivel PWM que aplicamos al motor.

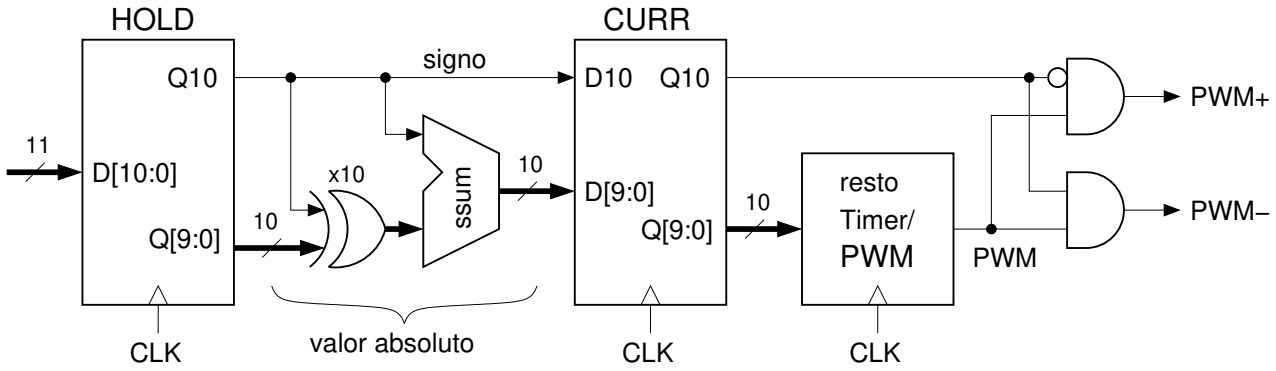


Figure 29: Modificación para incluir signo en la señal PWM

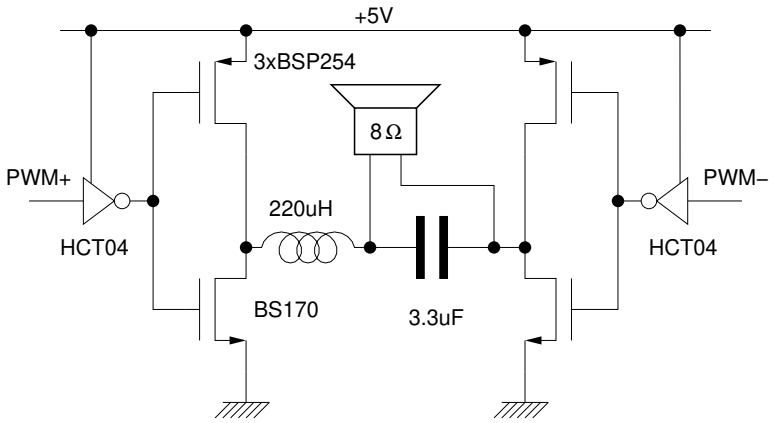


Figure 30: Esquema del amplificador de audio de clase D con puente H controlado por la salida PWM con signo

En la figura 29 mostramos la modificación circuital que hemos llevado a cabo para implementar el signo en el modulador PWM. Por lo pronto ahora tenemos dos salidas con las que controlar el puente H. Los pulsos PWM se rutan a una u otra dependiendo del bit de signo. Los registros HOLD y CURR son ahora de 11 bits en lugar de 10 para incluir el signo del dato. Sin embargo el nivel PWM tiene que ser proporcional al valor absoluto del dato que se escribe en HOLD. Para calcular este valor recurrimos a un circuito combinacional compuesto de un semisumador de 10 bits y 10 puertas XOR que hemos colocado entre los registros HOLD, en el que el dato es de 11 bits con signo, y CURR, en el que el dato es de 10 bits sin signo más un bit adicional con el signo.

Observemos que aunque la modulación PWM sigue siendo de 10 bits (1024 ciclos por pulso), ahora tenemos el doble de niveles de salida posibles al considerar el signo, con lo que hemos añadido un bit de resolución al modulador.

Aunque antes he mencionado la posibilidad de controlar la potencia y dirección de giro de un motor DC la aplicación que en realidad tenía en mente para la salida PWM era la generación de audio. En la figura 30 se muestra un amplificador conmutado de clase D construido alrededor de cuatro transistores MOSFET. El integrado 74HCT04 se usa aquí como convertidor de niveles digitales pues la salida de la FPGA es de sólo 3.3V. El altavoz podría conectarse directamente entre las dos salidas del puente, aunque en este ejemplo he preferido usar un filtro LC para eliminar la portadora PWM de la señal que llega al altavoz. Este filtro además tiene el efecto beneficioso de reducir el consumo de corriente.

Con este amplificador conectado a las salidas PWM, y utilizando la memoria flash SPI de configuración de la FPGA como almacenamiento para una grabación digital de audio, se ha podido reproducir la grabación en el altavoz con una calidad sorprendentemente buena. Y ello sin emplear ningún componente analógico.

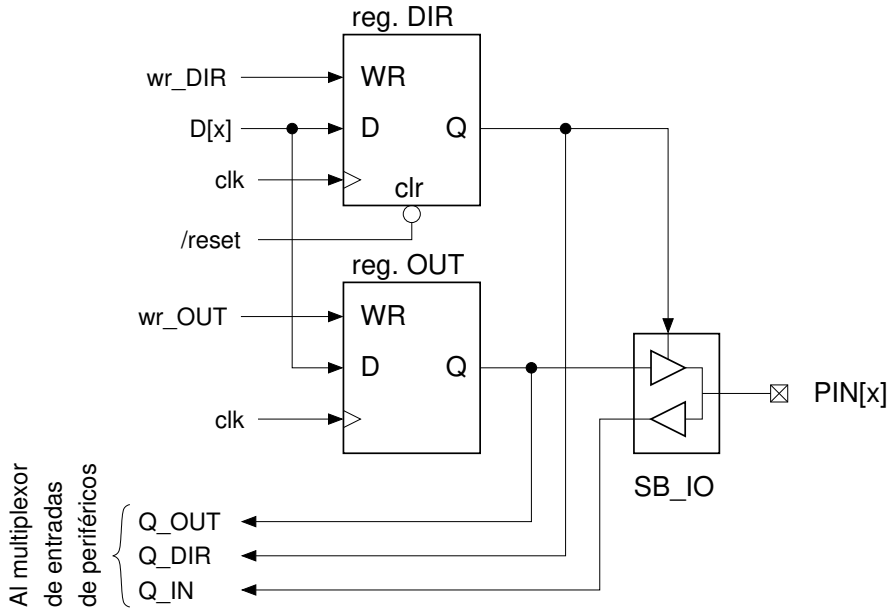


Figure 31: Diagrama de un pin de las entradas y salidas de propósito general (GPIO)

9.3 GPIO

Otro periférico obligado es un puerto de entrada y salida de propósito general similar al que podemos encontrar en la mayoría de microcontroladores comerciales. Tal como vemos en la figura 31, consta de un registro de salida y otro de direcciones, ambos de lectura y escritura. El estado del pin se lee en una dirección distinta de la del registro de salida, a la manera de los puertos de los microcontroladores AVR, lo que nos evita potenciales problemas con las secuencias read-modify-write sobre los pines de GPIO.

El registro de dirección tiene todos los bits en cero después de un reset, con lo que los pines del puerto GPIO comienzan programados como entradas.

Para implementar este periférico necesitamos unos bloque especiales de la FPGA (SB_IO) que incorporan los triestados en los pines. Estos mismos bloques serán también necesarios para las señales del bus de datos externo del microcontrolador. El código Veriolog del puerto es:

```
// GPIO
reg [15:0]gpo;
reg [15:0]gpdir=0;
wire [15:0]gpi;
// registro GPOUT
always @(posedge clk) gpo<=iowe7 ? cpu_do : gpo;
// registro GPDIR
always @(posedge clk or negedge resb )
    if (!resb) gpdir<=16'h0000;
    else gpdir<=iowe6 ? cpu_do : gpdir;
```

Y el código Verilog de un pin bidireccional SB_IO para la FPGA es el siguiente (Estos bloques pueden ser distintos en otras FPGAs):

```
SB_IO #( .PIN_TYPE(6'b 1010_01), .PULLUP(1'b 0) )
io_block_instance0 (
  .PACKAGE_PIN(   gpio[0]),
  .OUTPUT_ENABLE(gpdir[0]),
```

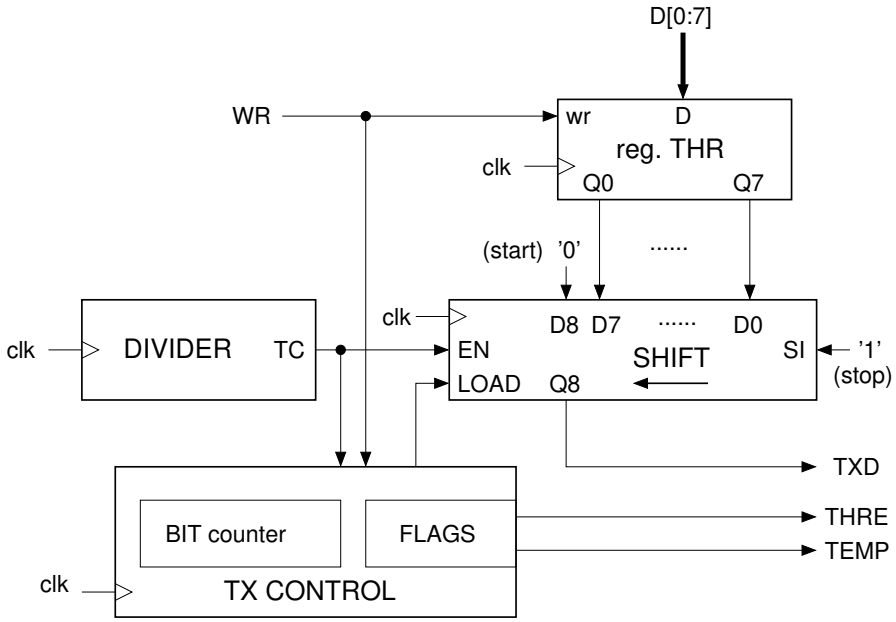


Figure 32: Diagrama simplificado del transmisor de la UART

```
    .D_OUT_0(         gpo[0]),
    .D_IN_0(          gpi[0])
);
```

9.4 UART

La UART son en realidad dos periféricos independientes: un transmisor y un receptor. Comenzamos viendo el transmisor cuyo esquemático se muestra en la figura 32.

El transmisor consta fundamentalmente de 4 bloques. El primero de ellos es simplemente un divisor para el reloj que genera un pulso cada cierto número de ciclos de reloj. Se trata de un contador de módulo fijo, aunque seleccionable en el momento de la síntesis en la FPGA, que se decrementa hasta llegar a cero para en el siguiente ciclo cargarse con el valor del divisor. Esto nos dará una velocidad de transmisión de $Baud = f_{CLK}/(DIVIDER+1)$.

El siguiente bloque es un registro de almacenamiento temporal, THR, en el que se puede escribir un dato de 8 bits para transmitir cuando todavía se está transmitiendo el anterior. El flag THRE se pone en '0' cuando se escribe en este registro y se vuelve a poner en '1' cuando su contenido se transfiere al registro de desplazamiento. Este flag se puede consultar en el registro PFLAGS (bit 1), y también puede generar una petición de interrupción.

El registro de desplazamiento es de 9 bits para incorporar en primer lugar el bit de START de las comunicaciones asíncronas. Además, por su entrada serie se insertan unos, que acabarán siendo los bits de STOP después de terminar de transmitir los datos.

El bloque de control gestiona la cuenta de los bits transmitidos, la carga del registro de desplazamiento, y los flags de estado, que incluyen el flag TEMP además del THRE. El flag TEMP (bit 2 de PFLAGS) se activa cuando tanto el registro THR como el de desplazamiento están vacíos, lo que significa que se ha transmitido hasta el bit de stop del último dato. El único formato de datos soportado por este transmisor es el de 8 bits de datos, sin paridad, y con un sólo bit de stop.

A continuación sigue el listado del código verilog del transmisor de la UART:

```
//-------------------------------------------
// Core de UART
```

```
// - Doble función de transmisor y receptor
//-----------------------------------------
module UART_CORE(
  output txd,    // Salida TX
  output tend,   // Flag TX completa
  output thre,   // Flag Buffer TX vacío
  input [7:0]d,  // Datos TX
  input wr,      // Escritura en TX
  output [7:0]q, // Datos RX
  output dv,     // Flag dato RX válido
  output fe,     // Flag Framing Error
  output ove,    // Flag Overrun
  input rxd,     // Entrada RX
  input rd,      // Lectura RX (borra DV)
  input clk
);
parameter DIVISOR= 16;
localparam NDIV = $clog2(DIVISOR);
//////////////////////////////////////
// Transmisor
reg [7:0]thr;    // Buffer TX
reg thre=1;      // Estado THR 1: vacío, 0: con dato
reg [8:0]shtx;   // Reg. desplazamiento de 9 bits
reg [3:0]cntbit; // Contador de bits transmitidos
reg rdy=1;       // Estado reg. despl. (1==idle)
// Divisor de TX
reg [NDIV-1:0] divtx=0;
wire clko;       // pulsos de 1 ciclo de salida
assign clko = (divtx==0);
always @ (posedge clk)
    divtx <= (wr&rdy) ? 0 : (clko ? DIVISOR-1: divtx-1);
always @(posedge clk)
begin
    if (wr) begin  // Escritura en buffer THR
`ifdef SIMULATION
        $write ("%c",d&255); $fflush ( );
`endif
        thr<=d;
        thre<=1'b0;
    end
    if (clko) begin
        if(rdy&(~thre)) begin      // Carga de reg. desp
            rdy<=1'b0;
            thre<=1'b1;
            shtx<={thr[7:0],1'b0}; // Incluido bit de START
            cntbit<=4'b0000;
```

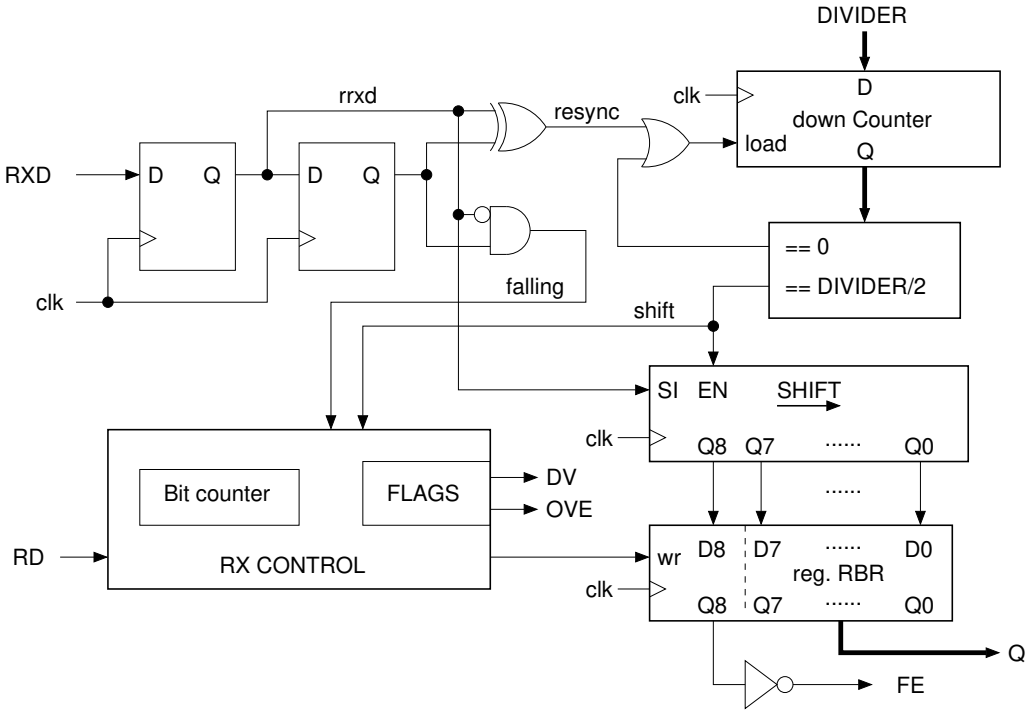


Figure 33: Diagrama simplificado del receptor de la UART

```
        end
        if(~rdy) begin              // Desplazamiento de bits
            shtx<={1'b1,shtx[8:1]};
            cntbit<=cntbit+1;
            if (cntbit[3]&cntbit[1]) rdy<=1'b1; // 9 bits: terminado
        end
    end
end
assign txd = shtx[0];
assign tend = thre&rdy;
```

El diagrama del receptor de la UART se muestra en la figura 33. En este caso tenemos la complicación adicional de sincronizar nuestro reloj de desplazamiento de bits con los datos recibidos. Para ello usamos un detector de cambios en RXD, formado por un par de flip-flops y una puerta XOR, que genera un pulso en *resync* cada vez que RXD cambia de estado. Este pulso reinicia el contador del divisor, lo que provocará que el final de la cuenta coincida aproximadamente con el final del tiempo de bit. Sin embargo RXD debe muestrearse en la mitad del tiempo de bit, y para ello el divisor genera un pulso, *shift*, cuando se llega a la mitad de su valor inicial. Este pulso desplaza los datos en el registro de desplazamiento que es de 9 bits aunque cada carácter dura 10 pulsos de *shift*. De este modo en el registro de desplazamiento el bit de comienzo del dato, que no tiene interés, se pierde pero quedan almacenados los 8 bits del dato junto con un bit de stop.

El bloque de control se encarga de la detección del comienzo de la trama, del contaje de los bits, de transferir los datos recibidos al registro buffer, RBR, y de activar los flags oportunos. Estos flags incluyen DV (dato válido, en bit 0 de PFLAGS), que se pone en 1 en cuanto un dato se copia a RBR y que se borra mediante un pulso de lectura en RD, y OVE (overrun, bit 5 de PFLAGS), que se activa cuando se transfiere un dato a RBR estando DV en '1', y que supone la pérdida de un dato por sobrescritura. El último flag, FE (bit 4 de PFLAGS), es el propio bit de stop invertido, de modo que un valor '1' indica un error de trama recibida. Todos estos flags se pueden consultar en el registro PFLAGS y además el flag DV puede usarse como petición de interrupción.

El código Verilog del receptor de la UART es el siguiente:

```
////////////////////////////////////
// Receptor
/// Sincronismo de reloj
reg [1:0]rrxd=2'b11; // RXD registrada dos veces
wire resinc;         // activa si cambio en RXD (resincroniza divisor)
wire falling;        // activa si flanco de bajada en RXD (para start)
always @(posedge clk) #1 rrxd<={rrxd[0],rxd};
assign resinc = rrxd[0]^rrxd[1];
assign falling = (~rrxd[0])&rrxd[1];
/// Divisor
// Genera un pulso en mitad de la cuenta (centro de bit)
// se reinicia con resinc
reg [NDIV-1:0] divrx=0;
wire shift; // Pulso de 1 ciclo de salida
wire clki0; // recarga de contador
assign shift = (divrx==(DIVISOR/2+1));
assign clki0= (divrx==0);
always @ (posedge clk) divrx <= (resinc|clki0) ? DIVISOR-1: divrx-1;
reg dv=0;               // Dato válido si 1
reg ove=0;              // Overrun
reg [8:0]shrx;          // Reg. desplazamiento (9 bits para stop)
reg [7:0]rbr;           // Buffer RX
reg stopb;              // Bit de stop recibido
reg [3:0]cbrx=4'b1111;  // Contador de bits / estado (1111== idle)
wire rxst;              // Guardar shr en buffer si flanco subida
assign rxst=(cbrx==4'b1111);
reg rxst0;              // Para detectar flanco
always @(posedge clk)
begin
    rxst0<=rxst;
    if (rxst & falling) cbrx<=4'h9;  // START: 9 bits a recibir
    if (shift & (~rxst)) begin       // Desplazando y contando bits
        shrx<= #1 {rrxd[0],shrx[8:1]};
        cbrx<=cbrx-1;
    end
    if (rxst & (~rxst0)) begin   // Final de cuenta
        {stopb,rbr}<=shrx;       // Guardando dato y bit STOP
        dv<=1;                   // Dato válido
        ove<=dv;                 // Overrun si ya hay dato válido
    end
    if (rd) begin   // Lectura: Borra flags
        dv<=0;
        ove<=0;
    end
end
```

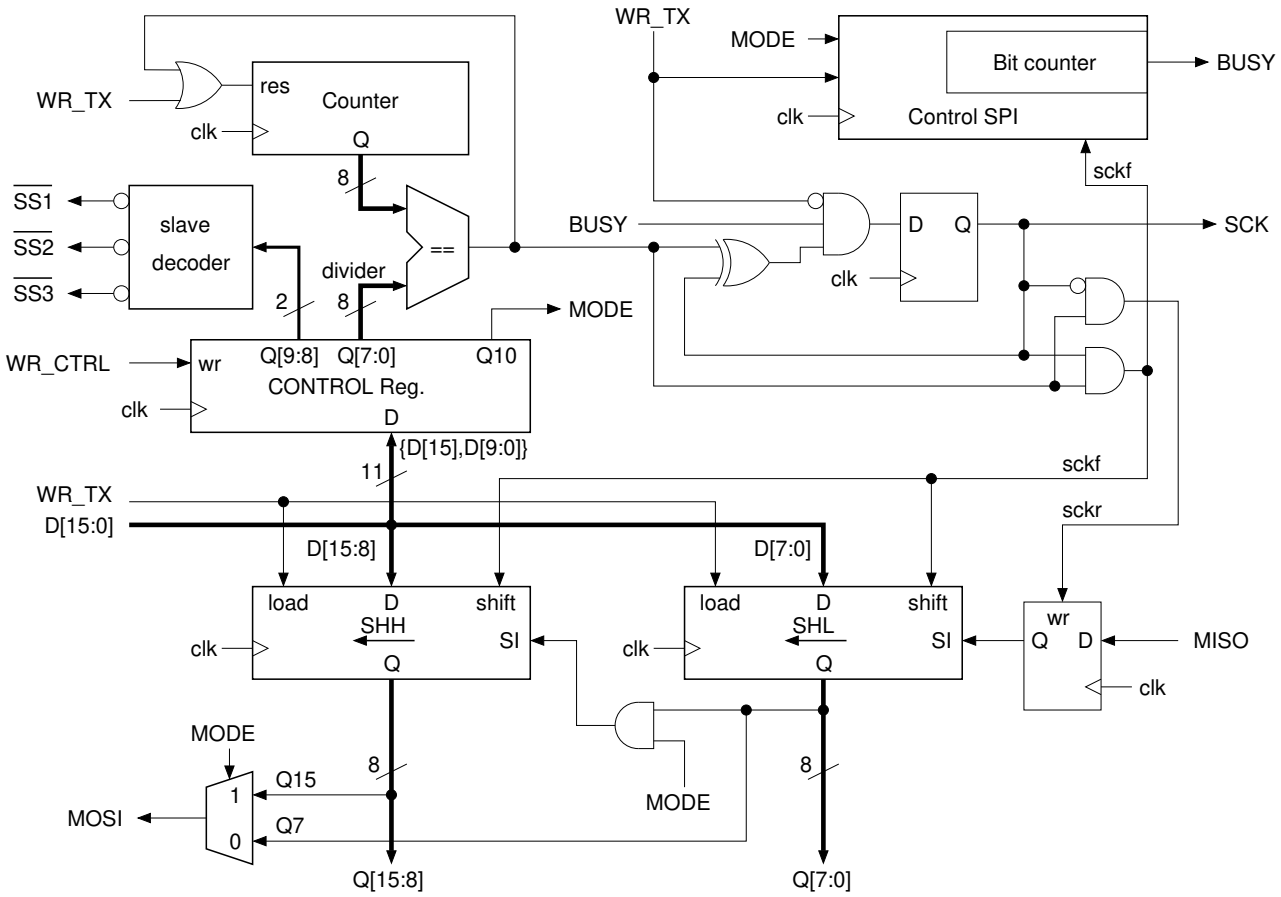


Figure 34: Diagrama simplificado del periférico SPI

```
assign fe=~stopb;   // el Flag FE es el bit de STOP invertido
assign q = rbr;
endmodule
```

Observemos que tanto el transmisor como el receptor de la UART funcionan con velocidades fijas. Una vez seleccionado el valor del DIVISOR durante la síntesis de la FPGA, éste no puede cambiarse. Sin embargo no sería muy difícil modificar la UART para que el valor de DIVISOR estuviese almacenado en un registro adicional. La modificación no es por lo tanto muy difícil de llevar a la práctica, aunque de momento queda pospuesta hasta ver si el resto de los periféricos caben en la FPGA.

Los registros THR y RBR tienen un ancho de 8 bits, con lo que los bits 8 al 15 de los buses de datos se ignoran en la escritura y se leen con todos los bits en 0 en las lecturas.

9.5 SPI

El controlador SPI se ha diseñado para poder transferir datos de 8 o 16 bits. Para ello se incluye un bit de control, MODO, que selecciona la longitud de los datos como 8 bits cuando vale 0 y 16 bits cuando vale 1. Además se puede seleccionar la frecuencia del reloj mediante un divisor de 8 bits, lo que nos permitirá generar en SCK frecuencias entre $f_{CLK}/2$ y $f_{CLK}/512$, de acuerdo con las necesidades de los periféricos SPI (por ejemplo las tarjetas SD necesitan un reloj de menos de 400kHz durante la fase de inicialización, y algunos ADCs para pantallas táctiles resistivas pueden requerir un reloj de no más de 125kHz).

En la figura 34 se muestra el diagrama del controlador SPI. En primer lugar hay que destacar el divisor, que genera un pulso cada (*divider*+1) ciclos de reloj. Este pulso hace conmutar el flip-flop de SCK, generando en esta señal una onda cuadrada de frecuencia $f_{SCK} = f_{CLK}/[2*(divider+1)]$. Además se generan pulsos independientes en los flancos de subida, *sck*r, y de bajada, *sckf*, de SCK que se van a utilizar para desplazar los

bits de datos y para el contaje de dichos bits. Fijémonos que SCK se fuerza a un valor 0 cuando BUSY está inactiva, y también cuando se realiza una escritura en el transmisor. Esto nos garantiza que comenzamos la transmisión con SCK en bajo, y la terminamos también con SCK en bajo.

El registro de desplazamiento es el mismo para los datos transmitidos y recibidos y está dividido en dos mitades de 8 bits. Si MODE vale '1' se utilizan los 16 bits del registro, pero si vale '0' se desplazan hacia MOSI sólo los 8 bits menos significativos y los 8 bits más significativos del registro de desplazamiento acaban siendo todos '0'. Los datos serie de entrada no provienen directamente de MISO, sino de una copia de dicha señal registrada en los flancos de subida de SCK, de acuerdo con el cronograma del modo 0 del bus SPI.

El bloque de control es básicamente un contador que se incrementa con cada flanco de bajada de SCK y mantiene activa la señal BUSY mientras la cuenta es menor que 8 para MODE='0', o menor que 16 para MODE='1'. Este contador se pone en 0 con las escrituras en el transmisor y se detiene cuando llega al final de la cuenta y desactiva BUSY. El flag BUSY se puede consultar en el registro PFLAGS (bit 3).

Por último hay que destacar la inclusión de un par de bits de selección de esclavo en el registro de control. Habitualmente la selección de los dispositivos esclavos en el bus SPI se realiza mediante pines de GPIO, y aquí también seguiría siendo factible, pero los bits del registro de control tienen las ventajas de ser siempre de salida y estar decodificados: La combinación de bits '00' no selecciona ningún esclavo, la '01' selecciona /SS1, la '10' selecciona /SS2, y '11' selecciona /SS3.

El código Verilog del periférico SPI es el siguiente:

```
module SPI(
  input clk,       // Reloj principal
  input [15:0]d,   // Datos desde la CPU
  input wr0,       // strobe de escritura en datosTX
  input wr1,       // strobe de escritura en REG_control
  output [15:0]q, // Datos hacia la CPU
  output busy,     // Flag de ocupado
  output SCK,      // Señales del bus SPI
  output MOSI,
  input  MISO,
  output [3:1]SSB // Selecciones de esclavo
);
reg [7:0]cntdiv=0;    // Contador del divisor de reloj
reg [10:0]controlr=0; // Registro de control: divisor, modo, ss
wire modo;            // bit de control: 0 = 8 bits , 1 = 16 bits
// reg de control
assign modo = controlr[10];
wire rstdiv;
assign rstdiv=(cntdiv==controlr[7:0]);
always @(posedge clk) begin
    cntdiv<=(wr0|rstdiv) ? 8'h00 : cntdiv+1;
    controlr <= wr1 ? {d[15],d[9:0]} : controlr;
end
assign SSB[1] = ~((~controlr[9])&( controlr[8]));
assign SSB[2] = ~(( controlr[9])&(~controlr[8]));
assign SSB[3] = ~(( controlr[9])&( controlr[8]));
// Toggle FF para SCK
reg SCK=0;          // Reloj SPI
```

```
wire sckr, sckf;    // strobes de flanco de subida y de bajada
always @(posedge clk) SCK<=(~wr0)&busy&(SCK^rstdiv);
assign sckr = rstdiv & (~SCK);
assign sckf = rstdiv & ( SCK);
// Contador de bits
reg [4:0]cntbit=5'b11111;  // Tiene que contar hasta 16
wire bittc;                // Final de cuenta de contador de bits
assign busy = ~(cntbit[4]|(cntbit[3]&(~modo))); // flag de BUSY
always @(posedge clk)
  cntbit= wr0 ? 0 :(busy ? (sckf ? (cntbit+1) : cntbit) : 5'b11111);
// reg de desplazamiento
reg [7:0]shh=0; // byte alto
reg [7:0]shl=0; // byte bajo
reg misoreg;    // MISO retardado medio ciclo
always @(posedge clk) begin
    shh <= wr0 ? d[15:8] :(sckf ? {shh[6:0],(shl[7]&modo)} : shh);
    shl <= wr0 ? d[7:0]  :(sckf ? {shl[6:0],misoreg} : shl);
    misoreg <= sckr ? MISO : misoreg;
end
assign q = {shh,shl};
assign MOSI = modo ? shh[7] : shl[7];
endmodule
```

9.6 Conectando todo

Para completar el diseño de nuestro microcontrolador tan sólo nos resta interconectar todos los bloques de acuerdo con lo mostrado en la figura 24 y añadir la lógica de selección apropiada para obtener el mapa de memoria de la figura 25. Esto se ha hecho mediante el siguiente código Verilog, en el que previamente se han instanciado los módulos de la CPU, RAM, y periféricos:

```
//-- Decodificación de direcciones
wire csiram,csio;
assign csram = (~addr[15]) & (~addr[14]) & (~addr[13]) & (~addr[12]);
assign csio =  (~addr[15]) & (~addr[14]) & (~addr[13]) & (~addr[12])&
               (~addr[11]) & (~addr[10]) & (~addr[9])  & (~addr[8])&
               (~addr[7])  & (~addr[6])  & ( addr[5])  ;
assign ramwr = ~((~rw) & csram);
//
// Multiplexado de datos de entrada a CPU
reg [15:0]ioin; // Datos desde periféricos (no es un registro)
assign cpu_di = csio ? ioin : (csram ? ram_do : exdin );
//
// Bus externo
assign exrw = rw | clk;  // escritura activa si reloj en bajo
assign exdo = cpu_do;
assign exaddr=addr;
//-------------
// Periféricos
//-------------
```

```
// Escrituras en registros (strobes)
wire iowe0,iowe1,iowe2,iowe3,iowe5,iowe6,iowe7;
assign iowe0=(~rw)&(csio)&(~addr[4])&(~addr[3])&(~addr[2])&(~addr[1])&(~addr[0]);
assign iowe1=(~rw)&(csio)&(~addr[4])&(~addr[3])&(~addr[2])&(~addr[1])&( addr[0]);
assign iowe2=(~rw)&(csio)&(~addr[4])&(~addr[3])&(~addr[2])&( addr[1])&(~addr[0]);
assign iowe3=(~rw)&(csio)&(~addr[4])&(~addr[3])&(~addr[2])&( addr[1])&( addr[0]);
assign iowe5=(~rw)&(csio)&(~addr[4])&(~addr[3])&( addr[2])&(~addr[1])&( addr[0]);
assign iowe6=(~rw)&(csio)&(~addr[4])&(~addr[3])&( addr[2])&( addr[1])&(~addr[0]);
assign iowe7=(~rw)&(csio)&(~addr[4])&(~addr[3])&( addr[2])&( addr[1])&( addr[0]);
//
// Lecturas de registros (stobes)
wire iore1,iore2;
assign iore1=( rw)&(csio)&(~addr[4])&(~addr[3])&(~addr[2])&(~addr[1])&( addr[0]);
assign iore2=( rw)&(csio)&(~addr[4])&(~addr[3])&(~addr[2])&( addr[1])&(~addr[0]);
//
// Multiplexor de entradas de I/O
always@*
  case (addr[2:0])
     0 : ioin <=  {12'h000,irqen}; // Máscara de IRQ
     1 : ioin <=  timerval;        // Timer
     2 : ioin <=  {8'h00,uart_do}; // UART rx
     3 : ioin <=  spi_do;          // SPI RX
     4 : ioin <=  {timirq,9'h000,ove,fe,spibusy,tend,thre,dv}; // PFLAGS
     5 : ioin <=  gpi;
     6 : ioin <=  gpdir;
     7 : ioin <=  gpo;
     default : ioin <= 16'hxxxx;
  endcase
```

Con esto completamos un módulo que incluye todo el microcontrolador, pero para poder sintetizar este diseño en una FPGA aún faltan algunos detalles específicos de estos dispositivos. Ya hemos mencionado los módulos SB_IO para los pines bidireccionales, y aún nos falta un módulo PLL para generar una señal de reloj de una frecuencia adecuada. Estos componentes son específicos de cada FPGA.

Y por último nos queda mapear cada una de las señales del microcontrolador a los pines externos de la FPGA, algo que también es dependiente del modelo concreto de FPGA y hasta de las herramientas de desarrollo empleadas.

El microcontrolador diseñado ocupa un 92% de las celdas lógicas de una FPGA ICE40HX1K. Para poder hacer un análisis comparativo del espacio ocupado por sus diferentes bloques se ha realizado una síntesis de bloques individuales con los siguientes resultados:

Bloque	Celdas lógicas (LC)	LC/1280	Flip-flops
Microcontrolador completo	1174	91.7%	394
CPU	745	58.2%	188
TIMER-PWM	111	8.7%	57
UART	84	6.6%	58
SPI	64	5%	42
resto	170	13.3%	49

En la tabla anterior también he añadido una columna con el número de celdas lógicas que contienen flip-flops. Realmente este dato lo he usado para confirmar que la síntesis ha sido tal como esperaba y que no se han

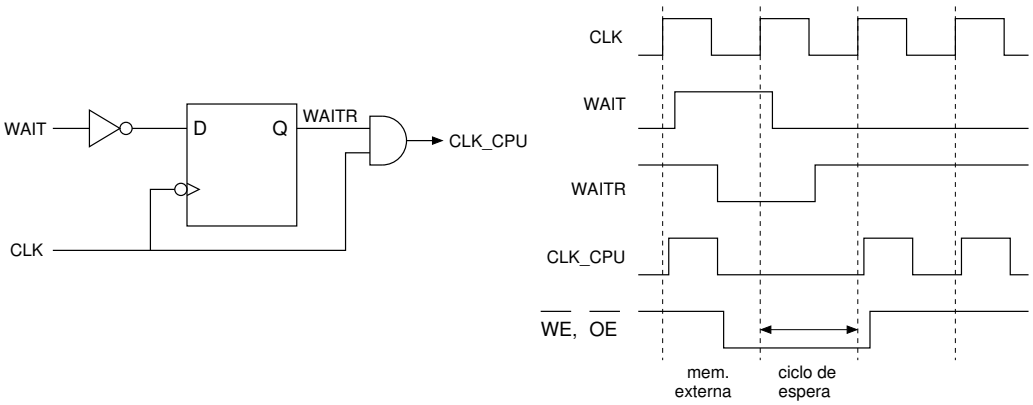


Figure 35: Entrada de espera para la CPU y cronograma asociado.

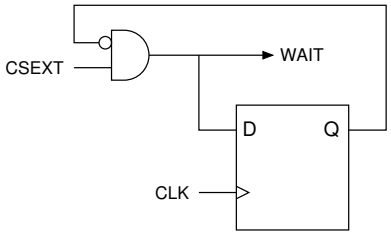


Figure 36: Lógica para añadir un estado de espera a los accesos a la memoria externa.

sintetizado más flip-flops de la cuenta a causa de construcciones "case" que acabasen siendo secuenciales en lugar de combinacionales. En la fila "resto" he incluido las celdas del microcontrolador que no están contadas en los cuatro bloques anteriores y que deberían corresponder entre otras cosas al periférico GPIO, al codificador de prioridad de las interrupciones, y a la "glue-logic" que interconecta todos los bloques.

Con todo esto, acabamos teniendo un microcontrolador dentro de una FPGA que es capaz de ejecutar los programas que carguemos en su memoria. Pero claro, esta es una CPU completamente distinta de todas las que hay en el mercado con lo que tampoco disponemos de ninguna herramienta de desarrollo que podamos usar para crear nuestros programas. Lo único que podemos hacer es partir de la tabla de instrucciones de la figura 16 y convertir a mano cada instrucción a su código máquina equivalente, lo que no es precisamente agradable. Por ello el siguiente proyecto a abordar sería el desarrollo de un ensamblador cruzado para al menos poder usar el lenguaje ensamblador. Y también sería conveniente escribir un simulador de alto nivel para poder depurar nuestros códigos. Los detalles de estas aplicaciones escapan al contenido de este documento, aunque el ensamblador, en particular, se ha utilizado para generar el código máquina de los ejemplos de programa que se muestran más adelante.

10 Memoria externa, estados de espera, estado "SLEEP"

La implementación para ICESTICK carecía de memoria externa pues esa placa no tiene disponibles los pines suficientes para conectar a la FPGA una memoria SRAM. Finalmente la memoria externa se ha añadido en una placa nueva, ICECREAM, diseñada a propósito para incluir una memoria SRAM de 64K x 16 bits. La memoria SRAM supuestamente tenía un tiempo de acceso de 10ns, con lo que podría funcionar sin problemas a la misma frecuencia que la CPU, pero desafortunadamente el modelo de SRAM que se soldó en el prototipo no era el que se esperaba sino una versión de la misma SRAM de 5V en lugar de 3,3V. Al estar alimentada con

2/3 de su tensión nominal es de esperar que la memoria sea más lenta de lo previsto, si es que llega a funcionar. Efectivamente, la CPU es capaz de leer y escribir en la memoria externa con un reloj de no más de 25MHz mientras que con la memoria interna la frecuencia de reloj puede superar los 40MHz.

Dado que la memoria es capaz de funcionar con un reloj de frecuencia más baja no me me he planteado desoldar el chip y sustituirlo por el correcto. Aunque tampoco me agrada la idea de bajar la frecuencia de reloj a la mitad cuando tenemos accesible la memoria externa. La solución ideal pasaría por seguir funcionando con la frecuencia de reloj máxima pero introduciendo un estado de espera cuando accedemos a la memoria externa. La CPU debería tener una entrada adicional, WAIT, que cuando se active detenga la ejecución del código, y que estará controlada desde la "glue logic" del microcontrolador (archivo system.v). El problema que se nos presenta es que durante todo el diseño de la CPU para nada me había planteado la posibilidad de detener la ejecución y además el microcontrolador ya ocupa el 90% de las celdas lógicas de la FPGA, con lo que una implementación de la espera como un rediseño de máquina de estados síncrona podría agotar los escasos recursos de la FPGA.

La solución ha pasado por dejar el módulo de la CPU tal como estaba y en su lugar añadir una pequeña lógica para "robar" pulsos a la señal de reloj cuando la entrada WAIT está activa. En la figura 35 se muestra el esquemático de esta lógica, en el que hay que destacar que los cambios de la señal WAIT se han sincronizado con los flancos de bajada de CLK para evitar la generación de pulsos espúreos en la salida. De este modo aunque WAIT sea una señal asíncrona siempre se van a quitar un número entero de pulsos de reloj de la señal de salida y no se generan glitches.

Tan sólo nos falta activar la señal WAIT cuando se accede a la memoria externa. Pero WAIT también se debe poder desactivar o de lo contrario la CPU se quedará bloqueada de forma indefinida en cuanto haya un acceso a la memoria externa. La lógica que he empleado es la de la figura 36, donde vemos que WAIT se activa si la selección de la memoria externa, CSEXT, está activa pero no si WAIT ya estaba activada en el ciclo anterior.

Con sólo esta poca lógica se han podido implementar los estados de espera en los accesos a la memoria externa. Ahora los tiempos de ejecución de los programas van a depender de la memoria que se esté utilizando. A título de ejemplo se muestran los tiempos de ejecución de un programa concreto (conjunto de Mandelbrot, 800×600 pixels, aritmética de 16bits, reloj de 36MHz) :

Tiempo total (segundos)	Condiciones
257	Sólo memoria interna
268 (+4.3%)	Pila en memoria externa
480 (+87%)	Código y Pila en memoria externa

Notar que en en la pila no sólo estaban las direcciones de retorno de las subrutinas, sino también unas pocas variables locales que no cabían en los registros. Cuando todo el código se ejecuta desde la memoria externa el tiempo tampoco llega a ser exactamente el doble ya que este programa genera un volumen grande de datos que se transmiten por la UART, y en las zonas del conjunto de Mandelbrot donde el cálculo es rápido hay que esperar a que la UART termine de transmitir los datos que tiene pendientes, retardos que no dependen de WAIT.

La entrada WAIT de la CPU puede tener otros usos interesantes. Por ejemplo se puede activar al escribir un determinado dato en un registro de un periférico, lo que equivale a "dormir" la CPU. WAIT se desactivaría al tener una petición de interrupción, lo que "despertará" la CPU. Mientras la CPU está dormida su reloj está siempre en nivel bajo, con lo que en su interior no hay ninguna actividad y ello va a reducir el consumo de corriente. Sin embargo los periféricos siguen teniendo su señal de reloj normal y pueden seguir generando peticiones de interrupción. En nuestro diseño la CPU se duerme cuando se escribe en la dirección 0x20 (registro IRQEN) un dato con el bit 7 en 1, tal como se muestra en el siguiente ejemplo:

```
IRQEN=      0x20 ; Reg Habilitación IRQ / Sleep
```

```
EUARTDV=    8    ; Habilita IRQ UART RX
SLEEP=      0x80 ; A dormir si 1
    ldi    r1,IRQEN
    ldi    r0,(EUARTDV|SLEEP) ; A dormir hasta IRQ de UART RX
    st     (r1),r0
```

Este código para el reloj de la CPU hasta que en la UART se recibe un dato y se genera la petición de interrupción correspondiente. El consumo de corriente en el core de la FPGA baja de 10.8mA durante la ejecución normal a 2.4mA cuando la CPU está dormida. Hay que tener en cuenta que los periféricos siguen teniendo su señal de reloj, lo cual implica un consumo de corriente, al igual que el hecho de seguir teniendo el PLL del multiplicador de reloj activo. A pesar de ello el ahorro de corriente con la CPU dormida es muy notable, con lo que sería una buena práctica de programación el dormir la CPU siempre que no haya nada que hacer y usar las interrupciones para despertarla.

11 Ejemplos de programa

Los siguientes ejemplos se escribieron para la variante de procesador GUS-16 V4, que tiene el juego de instrucciones original, incluyendo la instrucción LDPC en lugar de LDH, pero no las LD / ST con desplazamientos.

11.1 Hello world y un poco más

A continuación listamos algunos ejemplos de programa para nuestra CPU. El primero de ellos es un código que ejecuta varias tareas simples: Por una parte el programa principal imprime el mensaje “Hola Mundo” en la UART, y por otra una interrupción periódica del temporizador cambia el estado de los LED de la placa ICESTICK que se han asignado a los pines de GPIO y programado como salidas. Otra interrupción captura los datos recibidos desde la UART y los almacena en una variable que se consulta desde el programa principal.

11.1.1 Convención de uso de registros

Antes de escribir el código de este programa de ejemplo hay que establecer algunos criterios acerca del uso de los registros, lo que ayudará a la hora de asignarles uso y facilitará que podamos reutilizar partes del código sin necesidad de cambios. En particular hemos establecido las siguientes convenciones:

- El registro R7 va a ser el puntero de pila.
- El registro R6 va a contener la dirección de retorno en las subrutinas.
- Los registros R0, R1, y R2, se usan para pasar parámetros a las subrutinas y para devolver resultados. Su contenido no es necesario conservarlo en las llamadas a subrutinas.
- Los registros R3, R4, y R5, se usan para almacenar variables locales. Las llamadas a subrutinas no alteran el valor de estos registros.

11.1.2 Código fuente

El ensamblador que hemos desarrollado nos permite asignar valores a símbolos, y precisamente nuestro programa de ejemplo comienza definiendo una serie de tales símbolos, todos ellos relacionados con los periféricos:

```
;----------------------------------------
;----------- REGISTROS de E/S -----------
```

```
IRQEN=    0x20
TIMER=    0x21
UARTDAT=  0x22
SPIDAT=   0x23
PFLAGS=   0x24
SPICTRL=  0x25
GPIN=     0x25
GPDIR=    0x26
GPOUT=    0x27
;--------- Habilitación de interrupciones --------
EINTEX=    1   ; Habilita IRQ externa
ETIMER=    2   ; Habilita IRQ Timer
EUARTTHRE= 4   ; Habilita IRQ UART TX
EUARTDV=   8   ; Habilita IRQ UART RX
;--------- bits de PFLAGS -----------
UARTDV=   1      ; Dato válido en UART RX
UARTTHRE= 2      ; Listo para transmitir en UART TX
UARTTEND= 4      ; Transmisión completa en UART TX
SPIBUSY=  8      ; SPI ocupado transfiriendo datos
UARTFE=   0x10   ; Error de trama en UART RX
UARTOVE=  0x20   ; Error de overrun en UART RX
TIMERIRQ= 0x8000 ; Flag de IRQ del temporizador
```

A continuación sigue el código propiamente dicho, comenzando por los vectores de reset e interrupciones que están ubicados en la zona más baja de la memoria:

```
;--------------------------------------
;----------- VECTORES -----------------
;--------------------------------------
        org 0    ; RESET
        ldpc     r6
        word     inicio
        jind     r6
        org 4    ; Interrupción externa
        jr       IRQ0
        org 8    ; Interrupción del TIMER
        jr       IRQ1
        org 0xc  ; Interrupción de UART TX
        jr       IRQ2
        org 0x10 ; Interrupción de UART RX
        jr       IRQ3
```

Las posiciones de memoria comprendidas entre la dirección 32 y la 63 estás reservadas para los registros de los periféricos y por ello hemos de comenzar el resto del código en la dirección 64:

```
;--------------------------------------
;------------- CODIGO -----------------
;--------------------------------------
```

```
        org     0x40
```

A partir de este momento podemos escribir el código en cualquier orden, pero en nuestro ejemplo hemos comenzado por las rutinas de interrupción. Esto nos permite usar saltos cercanos (intrucción JR con desplazamiento máximo de ±2048 posiciones de memoria) para las rutinas de interrupción, mientras que para el vector de reset usamos un salto absoluto (Lo que en este ejemplo no sería necesario pues todo el código del programa ocupa mucho menos de 2048 palabras en la memoria)

```
;-------------------------------------
; Interrupción de TIMER
;-------------------------------------
IRQ1:   subi    r7,r7,1     ; R0, R1 -> pila
        st      (r7),r0
        subi    r7,r7,1
        st      (r7),r1
        ldi     r1,TIMER    ; La lectura borra la IRQ del timer
        ld      r0,(r1)
        ldpc    r1          ; Contador que se muestra en LEDs GPIO
        word    LEDcnt      ; LEDcnt++
        ld      r0,(r1)
        addi    r0,r0,1
        st      (r1),r0
        ldi     r1,GPOUT   ; GPOUT=LEDcnt>>11
        adc     r0,r0,r0   ; 5 bits MSB a LSB: R0 = ROL({cy,R0},6)
        adc     r0,r0,r0
        adc     r0,r0,r0
        adc     r0,r0,r0
        adc     r0,r0,r0
        adc     r0,r0,r0
        st      (r1),r0
        ld      r1,(r7)     ; R1, R0 <- pila
        addi    r7,r7,1
        ld      r0,(r7)
        addi    r7,r7,1
IRQ0:
IRQ2:   reti                ; Y pasamos a modo normal
;-------------------------------------
; Interrupción de UART RX
;-------------------------------------
IRQ3:   subi    r7,r7,1 ; R0, R1 -> pila
        st      (r7),r0
        subi    r7,r7,1
        st      (r7),r1
        ldi     r1,UARTDAT
        ld      r0,(r1)
        ldpc    r1
        word    inchar
```

```
        st      (r1),r0
        ld      r1,(r7) ; R1, r0 <- pila
        addi    r7,r7,1
        ld      r0,(r7)
        addi    r7,r7,1
        reti
```

A continuación van el código del programa principal y las subrutinas utilizadas:

```
;-------------------------------------------------
;       INICIO
;-------------------------------------------------
inicio:
        ldpc    r7          ; Puntero de Pila al final de RAM
        word    4096
        ; GPIO: LEDs como salida
        ldi     r1,GPDIR
        ldi     r0,0x1F
        st      (r1),r0
        ; Interrupcion en timer
        ldi     r1,TIMER    ; 4096 interrupciones / segundo
        ldpc    r0
        word    (8789-1)
        st      (r1),r0
        ld      r0,(r1)     ; Borramos IRQ
        ldi     r1,IRQEN    ; Habilita IRQ Timer y UART RX
        ldi     r0,(ETIMER|EUARTDV)
        st      (r1),r0
mbuc:   ; Bucle principal del programa
        ; Imprimimos mensaje "Hello World"
        ldpc    r0          ; Imprime cadena de caracteres TXT
        word    txtle       ; empaquetada tipo Little-Endian
        adpc    r6,1        ; Llamada a subrutina
        jr      putsle
        ldpc    r1          ; Espera por caracter desde UART
        word    inchar
mb2:    ld      r0,(r1)
        jz      mb2
        ldi     r6,0        ; Borramos caracter
        st      (r1),r6
        adpc    r6,1        ; eco
        jr      putch
        jr      mbuc
;--------------------------------------
;------------  SUBRUTINAS  -------------
;--------------------------------------
;-------------------------------------------------
```

```
; Envía cadena de caracteres empaquetados al terminal
; parámetros:
;    R0: puntero a la cadena de caracteres
;    R6: dirección de retorno
; retorna:
;    R0-R2: modificados
putsle: subi    r7,r7,1
        st      (r7),r6
        or      r2,r0,r0    ; R2=R0
putsle1:
        ld      r0,(r2)
        ldi     r1,0xff     ; byte LSB
        and     r0,r0,r1
        jz      putsle3     ; final de cadena
        adpc    r6,1
        jr      putch
        ld      r0,(r2)     ; byte MSB
        shr     r0,r0
        shr     r0,r0
        shr     r0,r0
        shr     r0,r0
        shr     r0,r0
        shr     r0,r0
        shr     r0,r0
        shr     r0,r0
        jz      putsle3     ; final de cadena
        adpc    r6,1
        jr      putch
        addi    r2,r2,1
        jr      putsle1
putsle3:
        ld      r6,(r7)
        addi    r7,r7,1
        jind    r6
;-----------------------------------------------
; Envía caracter a la UART
; parámetros:
;    R0: dato a enviar
;    R6: dirección de retorno
; retorna:
;    R1: modificado
putch:  ldi     r1,PFLAGS    ; Esperamos flag THRE en 1
        ld      r1,(r1)
        tsti    r1,UARTTHRE
        jz      putch
        ldi     r1,UARTDAT
```

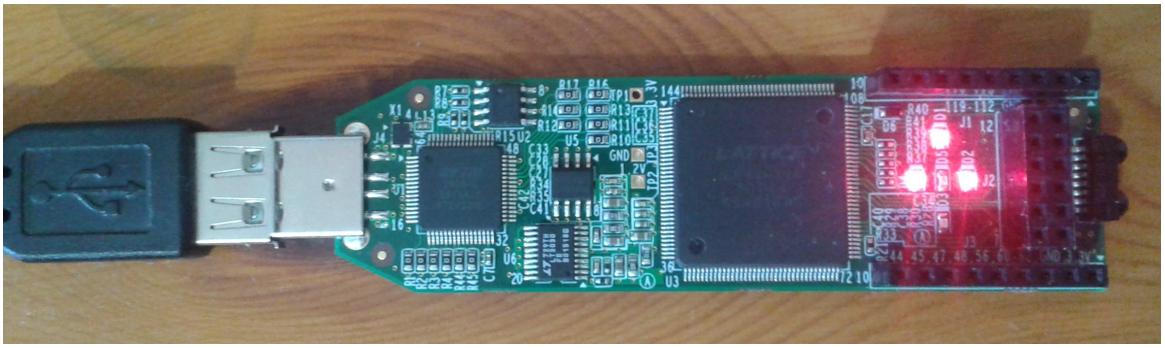


Figure 37: Fotografía de la placa ICESTICK con contaje binario en los LEDs.

```
st      (r1),r0      ; envía dato
jind    r6           ; y retornamos
```

La memoria además de los códigos de las instrucciones también contiene las constantes y variables del programa. En nuestro ejemplo tenemos como constante la cadena de caracteres "¡¡¡Hola Mundo!!!" que está empaquetada como dos caracteres por cada posición de memoria gracias a la directiva "asczle", donde "le" indica que el primer carácter de cada pareja va en la parte menos significativa del dato (little-endian). Como variables tenemos el valor que se muestra en los LEDs, " LEDcnt", y el caracter que se ha recibido desde la UART mediante una rutina de interrupción, "inchar".

```
;---------------------------------------
;------------- CONSTANTES --------------
;---------------------------------------
txtle:  asczle "\n\n¡¡¡Hola Mundo!!!\n"
;---------------------------------------
;-------------- VARIABLES --------------
;---------------------------------------
LEDcnt: word 0
inchar: word 0
```

11.1.3 Resultados

El resultado de la ejecución de este código es una serie de mensajes "¡¡¡Hola Mundo!!!" que se repiten cada vez que pulsamos una tecla, 'A' en este ejemplo, en un terminal serie con una velocidad de 115200 bits por segundo.

```
¡¡¡Hola Mundo!!!
A
¡¡¡Hola Mundo!!!
A
¡¡¡Hola Mundo!!!
```

Además, los LEDs de la placa ToolStick muestran un contaje binario lo suficientemente lento como para poder ser observado a simple vista:

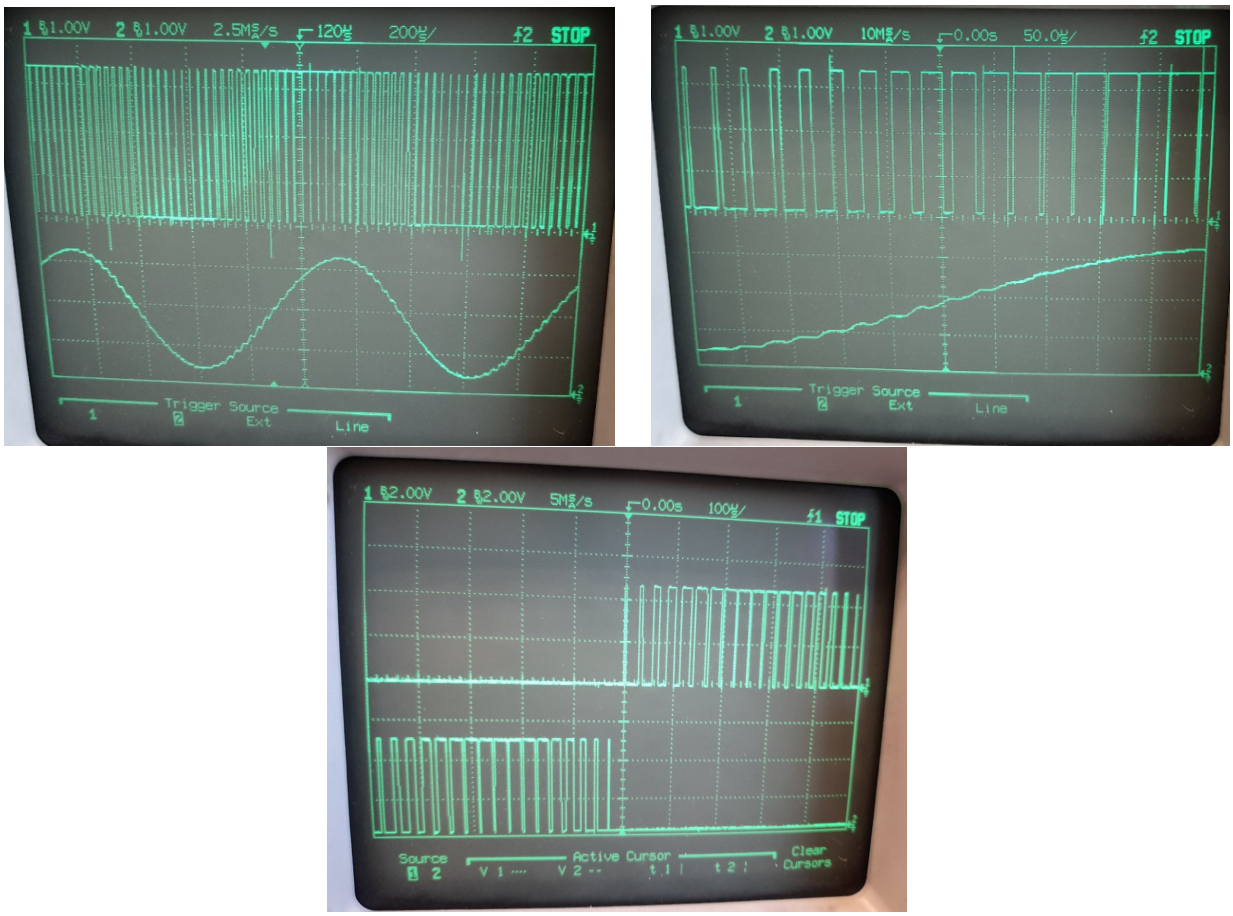


Figure 38: Arriba, iquierda: Forma de onda en la salida PWM de la FPGA y la misma señal tras un filtrado pasa-bajos. Arriba, derecha: detalle de las mismas señales. Abajo: Ondas PWM para una sinusoide con signo

11.2 Generador de señal PWM

El siguiente ejemplo es un programa que genera una sinusoide a través de la salida PWM usando una rutina de interrupción. En este programa sólo vamos a generar niveles de señal PWM positivos, con lo que los pulsos se obtendrán exclusivamente en la salida PWM+, lo que nos facilita su filtrado y visualización en el osciloscopio.

Para comenzar programamos el máximo del contador en 1022, con lo que tendremos 1023 ciclos por cada pulso de la onda PWM, que será de una frecuencia:

$$f_S = f_{CPU}/(MAX+1) = 35190.6\text{Hz}$$

Con cada interrupción calculamos un nuevo valor que escribimos en el registro de HOLD del modulador PWM. El cálculo simula un resonador sin pérdidas con la siguiente formula de recurrencia

$$y_i = 2\cos(2\pi f/f_S)y_{i-1} - y_{i-2}$$

Donde f es la frecuencia de la señal generada, e y_j son las muestras de la secuencia de salida. Para que el algoritmo funcione como se espera los valores iniciales de y_{i-1} e y_{i-2} deben estar próximos al valor máximo de la sinusoide. Pero no debemos olvidar que estos valores tienen signo, y que su rango va desde -32768 hasta +32767. En en ejemplo comenzamos con un valor +30000.

Las muestras generadas son de 16 bits con signo, pero lo que queremos son datos de 10 bits sin signo. Para adaptar los datos primero sumamos un offset de 32768, lo que equivale a representar la misma sinusoide con datos de 16 bits pero sin signo, y después nos quedamos con los 10 bits más significativos, que acabamos copiando al registro HOLD del modulador PWM.

La ejecución del programa que se lista al final de esta subsección genera una onda rectangular de anchura de pulsos variable que no se asemeja mucho a una sinusoide. Sin embargo, si hacemos un filtrado pasa-bajos de dicha señal la sinusoide se muestra de forma clara en la pantalla del osciloscopio (ver figura 38). El filtro eran dos simples redes RC conectadas en cascada con una frecuencia de corte de unos 5kHz ($R_1 = 3.3k\Omega$, $C_1 = 10nF$, $R_2 = 10k\Omega$, $C_2 = 3.3nF$)

La misma sinusoide podía generarse usando la modulación PWM con signo. En este caso los semiciclos positivos generan una secuencia de pulsos en la salida PWM+ y los negativos en PWM-. Las muestras "y" tienen signo, pero son de 16 bits. Tan sólo hemos de dividirlas por 32 para que estén dentro del rango del modulador PWM, pero la división ha de conservar el signo. Por ello usamos la instrucción SHRA:

```
; R2 = y = 2*y1 -y1/32 -y2
shra    r0,r2
shra    r0,r0
shra    r0,r0
shra    r0,r0
shra    r0,r0   ; R0 = y/32 (desde -1024 hasta +1023)
ldi     r1,PWM
st      (r1),r0 ; nuevo valor PWM
```

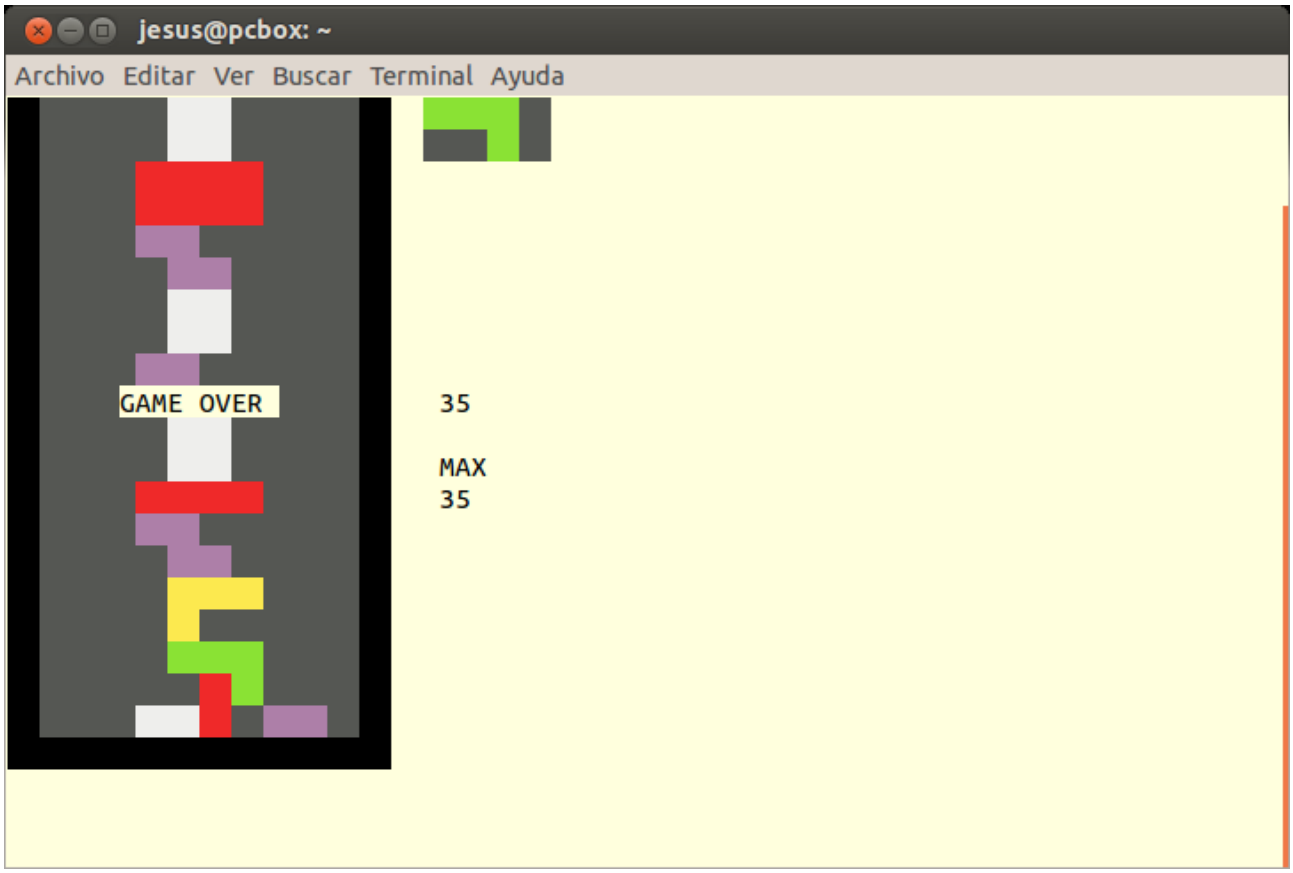
En la figura 38 también se muestran las ondas de salida en PWM+ y PWM- para la sinusoide con signo. Observemos que se trata de una salida diferencial y que hay que restar la onda PWM- de la PWM+ para obtener la tensión de salida final.

A continuación sigue el listado del generador de sinusoide sin signo:

```
IRQEN=      0x20
TIMER=      0x21
PWM=        0x24
ETIMER=     2
;-------- vectores ------------
        org     0           ; RESET
        jr      inicio
        org     8           ; IRQ1: TIMER
        jr      IRQ1
;-------- Variables estáticas ----------
        org     0x40
siny1:  word    30000   ; valor inicial del seno (muestra -1)
siny2:  word    30000   ; valor inicial del seno (muestra -2)

;------------------ IRQ TIMER -------------------------------
; calcula y=2*cos(2*pi*f/fs)*y1 - y2
; con 2*cos(2*pi*f/fs) = (2-1/32) esto nos da
; fs = 36MHz/1023 = 35191Hz, y
; f=fs*arccos((2-1/32)/2)/(2*pi) = 991.388 Hz

IRQ1:   subi    r7,r7,1     ; registros modificados a la pila
        st      (r7),r0
        subi    r7,r7,1
        st      (r7),r1
        subi    r7,r7,1
        st      (r7),r2

timl1:  ldi     r1,siny1    ; R2 = y1
        ld      r2,(r1)
        shra    r0,r2       ; R0 = y1/32 (con signo)
        shra    r0,r0
        shra    r0,r0
        shra    r0,r0
        shra    r0,r0
        add     r2,r2,r2    ; R0 = 2*y1 -y1/32
        sub     r2,r2,r0
        ldi     r1,siny2
        ld      r0,(r1)
        sub     r2,r2,r0    ; R2 = y = 2*y1 -y1/32 -y2
        ldpc    r0
        word    0x8000
        add     r0,r0,r2    ; R0 = y sin signo (de 0 a 65535)
        shr     r0,r0
        shr     r0,r0
        shr     r0,r0
        shr     r0,r0
        shr     r0,r0
        shr     r0,r0       ; R0 = (y+32768)>>6 (de 0 a 1023)
        ldi     r1,PWM
        st      (r1),r0     ; nuevo valor PWM y borrado de IRQ

        ldi     r1,siny1    ; y2 = y1
        ld      r0,(r1)
        ldi     r1,siny2
        st      (r1),r0
        ldi     r1,siny1    ; y1 = y
        st      (r1),r2

timl2:  ld      r2,(r7)     ; recuperamos registros de la pila
        addi    r7,r7,1
        ld      r1,(r7)
        addi    r7,r7,1
        ld      r0,(r7)
        addi    r7,r7,1
        reti

;-------------------------------------------------------------------
;       INICIO
;-------------------------------------------------------------------
inicio:
        ldpc    r7          ; Puntero de Pila al final de RAM
        word    4096

        ldi     r1,TIMER    ; MAX=1022
        ldpc    r0
        word    1022
        st      (r1),r0
        ldi     r1,IRQEN    ; Habilita IRQ del Timer
        ldi     r0,ETIMER
        st      (r1),r0

mbuc:   jr      .           ; Bucle sin fin. El procesamiento se hace en IRQ1
```

11.3 Videojuego

La siguiente aplicación es una versión para terminal ANSI del conocido videojuego TETRIS. El código fuente es demasiado largo para incluirlo en este documento, pero no podíamos dejar de incluir una captura del terminal serie:



Aunque no incluya todo el código fuente al menos quería mencionar que dicho código se ha generado "compilando" pacientemente a mano las fuentes en lenguaje C del videojuego, tal como se muestra en la función que se incluye como ejemplo. Esta función rota $\pm 90^{\circ}$ la pieza que que está cayendo. "pieza" es una variable global de 4x4 palabras que se encuentra por debajo de la dirección 0x100, con lo que su dirección se puede cargar con una simple instrucción LDI. De hecho todas las variables estáticas de pequeño tamaño se encuentran en el área de memoria que va desde la dirección 0x40 hasta la 0xff. Las tablas de datos grandes, en cambio, están ubicadas después del código. Las variables locales se han asignado a alguno de los registros, preferiblemente R3, R4, o R5. En este caso concreto no se llama a otras subrutinas y con ello el registro R6 también se ha usado para una variable local. Como aspecto destacable de este código tenemos el hecho de almacenar una tabla de datos local, "tmp[4][4]" en el espacio de la pila.

```
;void rota(int giro)
rota:   ; R0: 0 horario,  !=0 antihorario
        subi    r7,r7,1     ; prólogo
        st      (r7),r6
        subi    r7,r7,1
        st      (r7),r5
        ;   int i,j; R6=i, R5=j
        ;   uchar tmp[4][4];
        ldi     r1,16
        sub     r7,r7,r1    ; espacio para tabla en la pila
;   if (giro!=0) {      //Giro antihorario de 90º
        or      r0,r0,r0
        jz      rotah
;       for (i=0;i<4;i++)
;           for (j=0;j<4;j++) tmp[3-j][i]=pieza[i][j];
        ldi     r1,pieza
        ldi     r6,0
rota1:  ldi     r5,0
rota2:  ldi     r2,3
        sub     r2,r2,r5    ; 3-j
        add     r2,r2,r2
        add     r2,r2,r2
        add     r2,r2,r6    ; [3-j]*4 + i
        add     r2,r7,r2    ; &tmp[3-j][i]
        ld      r0,(r1)
        addi    r1,r1,1
        st      (r2),r0
        addi    r5,r5,1
        cmpi    r5,4
        jnz     rota2
        addi    r6,r6,1
        cmpi    r6,4
        jnz     rota1
        jr      rota4
;   } else {            //Giro horario de 90º
;       for (i=0;i<4;i++)
;           for (j=0;j<4;j++) tmp[j][3-i]=pieza[i][j];
rotah:  ldi     r1,pieza
        ldi     r6,0
rota5:  ldi     r5,0
rota6:  add     r2,r5,r5
        add     r2,r2,r2    ; j*4
        addi    r2,r2,3
        sub     r2,r2,r6    ; j*4 +3 -i
        add     r2,r2,r7    ; R2 = &tmp[j][3-i]
        ld      r0,(r1)
        addi    r1,r1,1
        st      (r2),r0
        addi    r5,r5,1
        cmpi    r5,4
        jnz     rota6
        addi    r6,r6,1
        cmpi    r6,4
        jnz     rota5
;   }
rota4:  ;       for (i=0;i<4;i++)
        ;           for (j=0;j<4;j++) pieza[i][j]=tmp[i][j];
        ldi     r1,pieza
        or      r2,r7,r7
        ldi     r6,16
rota3:  ld      r0,(r2)
        st      (r1),r0
        addi    r1,r1,1
        addi    r2,r2,1
        subi    r6,r6,1
        jnz     rota3

        ldi     r1,16       ; reajusta la pila
        add     r7,r7,r1
        ld      r5,(r7)     ; epílogo
        addi    r7,r7,1
        ld      r6,(r7)
        addi    r7,r7,1
        jind    r6
```

12 Apéndice I. Organización de las fuentes, síntesis, simulación

Hay dos conjuntos de archivos fuente: uno de ellos es el destinado a la síntesis en FPGA mientras que el otro se ha usado para la simulación del diseño. La diferencia entre ellos se limita al módulo TOP, que en el primer caso se tiene en el archivo "main.v" mientras que en el segundo se encuentra en "tb.v". A continuación se describen las funciones de cada fuente:

12.1 Archivos para la síntesis en FPGA

- **main.v**: Tiene la función de interfaz entre el diseño del microcontrolador y los detalles físicos de la FPGA. Durante la fase de "place & route" este archivo está directamente relacionado con el de condicionantes físicos "**pines.pcf**", en el que se indica en qué pin concreto de la FPGA va cada señal del diseño. Incluye los módulos:
  - pll, del archivo "**pll.v**", generado mediante el comando "`icepll -i 12 -o 36 -m -f pll.v`". Este bloque multiplica por 3 la frecuencia del reloj para obtener internamente un reloj de 36MHz.
  - SB_IO, bloque para salidas triestados utilizado en los pines de GPIO y en el bus de datos de la memoria externa.
  - SYSTEM, del archivo "**system.v**". En este archivo se detalla la interconexión de los distintos bloques que componen el microcontrolador y se incluye la "glue-logic" necesaria para implementar el mapa de memoria deseado. Incluye los módulos:
    - CPU_1CV2, del archivo "**cpu.v**". Este módulo contiene el diseño de la CPU GUS16.
    - genram, bloque de memoria RAM parametrizable para la memoria interna del micro.
    - UART_CORE, del archivo "**uart.v**", con el periférico UART.
    - TIMERPWM, del archivo "**timer.v**", que detalla el diseño del temporizador con la función PWM.
    - SPI, del archivo "**spi.v**", con el diseño del periférico SPI.
    - Otros periféricos más simples, como GPIO, o IRQEN, se han codificado directamente en el módulo SYSTEM.

La síntesis se controla desde el archivo "**Makefile**". Se invoca con los comandos **"make sint**" o "**make burn**", y consta de los siguientes pasos:

- Síntesis lógica mediante el programa "yosys":

```
yosys -p "synth_ice40 -relut" -o main.json main.v cpu.v system.v uart.v
                          spi.v pll.v timer.v
```

Este programa básicamente convierte nuestro diseño en Verilog en un conjunto de bloques de FPGA interconectados. El resultado queda en el archivo "main.json", que a pesar de ser un archivo de texto es bastante críptico.

- Place & route mediante el programa "nextpnr":

```
nextpnr-ice40 --hx1k --pcf pines.pcf --json main.json --asc main.txt
                          --package tq144
```

En este paso el programa “nextpnr” asigna a cada bloque una posición concreta dentro de la FPGA después de analizar una serie de posibilidades y de quedarse con la más óptima. Opcionalmente se puede añadir al comando la condición “–freq=<frecuencia de reloj en MHz>” para forzarle a descartar las posibles asignaciones que resulten en una frecuencia máxima de reloj inferior a la indicada. Sin embargo esta opción también implica que “nextpnr” puede no encontrar ninguna solución satisfactoria, así que hay que usarla con cuidado.

El resultado es un mapa de bits de configuración para la FPGA en el archivo “main.txt”. Este es un archivo de texto que no nos sirve aún para programarlo en la FPGA.

- Generación del mapa de bits de configuración con el programa “icepack”:

```
icepack main.txt main.bin
```

Con esto generamos finalmente el archivo binario “main.bin” que contiene una imagen exacta de los datos que se han de programar en la RAM de configuración de la FPGA, bien directamente o desde la memoria Flash serie.

El último paso es la programación de este archivo en la FPGA o en la Flash, pero la aplicación a usar depende de la placa FPGA y de la interfaz que esta tenga con el PC. Así tenemos los comando:

- Placa ICESTICK:
  - `iceram main.bin`: Para cargar la imagen directamente en la FPGA.
  - `iceprog main.bin`: Para programar la imagen en la Flash serie no volátil.
- Placa ICECREAM:
  - `iceload -c main.bin`: Para cargar la imagen directamente en la FPGA.
  - `iceload -p main.bin`: Para programar la imagen en la Flash serie no volátil.

12.2 Archivos para simulación

- **tb.v**: “test-bench”. Contiene una instancia de SYSTEM junto con un bloque “initial” en el que se generan los estímulos necesarios para la simulación, que en su versión más simple han de incluir al menos las señales del reloj y de reset. El resto de módulos son los mismos que se han utilizado para la síntesis:
  - **system.v**
    - **cpu.v**
    - **uart.v**
    - **timer.v**
    - **spi.v**

La simulación también se gestiona desde el archivo “Makefile” y para invocarla basta con el comando “**make**” o “**make sim**”. En ambos casos se ejecuta el programa “**iverilog**”:

```
iverilog tb.v cpu.v system.v uart.v spi.v timer.v -DSIMULATION
```

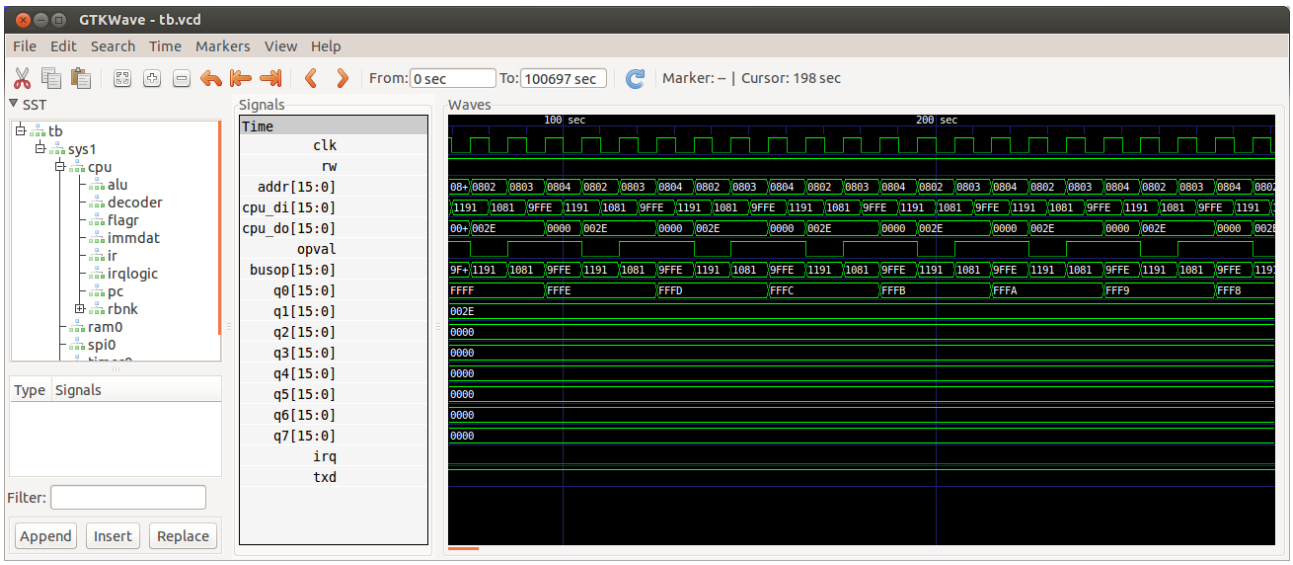
Vemos que en la simulación no intervienen los archivos “main.v” ni “pll.v” y que definimos el símbolo SIMULATION, de modo que en nuestros fuentes podemos incluir código condicional que se tenga en cuenta para la simulación pero no durante la síntesis (en concreto se tiene en “uart.v”). Este comando genera un archivo ejecutable “tb.out” que debemos ejecutar para realizar la simulación:

```
./tb.out
```

El resultado es un archivo “tb.vcd” con las formas de onda de las señales grabadas durante la simulación que podremos visualizar con el programa “**gtkwave**”:

```
gtkwave tb.vcd tb.gtkw
```

“tb.gtkw” es un archivo en el que podemos guardar la configuración de gtkwave para no tener que hacerlo cada vez que ejecutemos esta aplicación. Lo que veremos en la ventana de “gtkwave” sería algo como lo siguiente:



13 Apéndice II. Documentación del programa ensamblador

El formato genérico de una línea de código fuente es:

```
etiqueta: mnemónico/directiva operando,operando... ;(comentarios)
etiqueta= expresión ; (comentarios)
```

La etiqueta puede estar o no estar presente. El nombre de la etiqueta no puede empezar con un dígito numérico y debe estar terminada bien por dos puntos ’:’ o por igual ’=’. En el primer caso a la etiqueta se le asigna la dirección de la instrucción actual, mientras que en el segundo se asigna el resultado de evaluar la expresión que sigue.

Los operandos de las instrucciones pueden ser registros (’R0’ a ’R7’) o expresiones evaluables para los casos de operandos literales. En el caso de los saltos relativos a estas expresiones se les resta de forma automática la dirección de la instrucción para obtener un desplazamiento con signo. En las instrucciones con más de un operando el primero es el registro destino y el último podría ser un dato inmediato.

Ejemplos:

```
ORG 0x120
label1: LDI R1,<const16 ; label1 pasa a valer 0x120
label2= 0x314 ; label2 pasa a valer 0x314
```

Obsérvese que el segundo operando de la instrucción LDI del ejemplo es una expresión. En particular es el byte menos significativo (<) del valor de la etiqueta ’const16’ que debe estar definida en alguna otra parte del programa.

A la hora de evaluar etiquetas, mnemonicos, registros, y expresiones no se consideran diferencias entre mayúsculas y minúsculas. Esto sólo es relevante para las constantes y cadenas en ASCII. Así por ejemplo 'A' vale 0x41 mientras que 'a' vale 0x61.

13.1 Directivas

Estas son las directivas soportadas:

- `ORG <expresión>` Hace que el contador de direcciones pase a tener el valor indicado.
- `WORD <expresión>` Genera directamente una palabra en el código con el valor indicado
- `ASCZBE "texto entre comillas dobles"` Empaqueta los caracteres ASCII del texto en una serie de palabras, dos bytes por palabra, con el primer byte del texto en la parte más significativa de la primera palabra generada. La cadena de texto se termina con uno o dos bytes en cero, hasta completar un número par de bytes.
- `ASCZLE "texto entre comillas dobles"` Empaqueta los caracteres ASCII del texto en una serie de palabras, dos bytes por palabra, con el primer byte del texto en la parte menos significativa de la primera palabra generada. La cadena de texto se termina con uno o dos bytes en cero, hasta completar un número par de bytes.

Las cadenas de texto pueden incluir caracteres de escape para generar caracteres especiales. Estos caracteres comienzan por la barra de escape '\' y son:

- `\r` Retorno de carro (ASCII 0x0D)
- `\n` Nueva línea (ASCII 0x0A)
- `\t` Tabulador (ASCII 0x09)
- `\b` Backspace (ASCII 0x08)
- `\"` Comillas dobles (ASCII 0x22)
- `\'` Comillas simples (ASCII 0x27)
- `\\` Barra de escape (ASCII 0x5C)
- `\e` Escape (ASCII 0x1B)

13.2 Expresiones

Las expresiones incluyen constantes numéricas, variables predefinidas, etiquetas, y sus posibles combinaciones mediante operaciones aritméticas y/o lógicas. A continuación describimos estos componentes:

- Constantes numéricas:

  `512` Constante expresada en base decimal

  `0x200` La misma constante en base hexadecimal

  `'A'` valor del código ASCII de la letra A mayúscula.

- Variables predefinidas:

  `.` `(punto)` Es el valor actual del contador de direcciones.

  Ejemplo de uso: "`JR . ; bucle infinito`"

  `CPUMODEL` Tipo de CPU para el que se genera el código. 0: CPU Harvard, 1: CPU con pipeline, 2: CPU con pipeline y LDPC en lugar de LDH.

- Operadores unarios:

  Son el primer caracter de la expresión:

  `-valor` La expresión "valor" se cambia de signo mediante el cálculo de su complemento a dos.

  `~valor` Los bits de la expresión "valor" se invierten (complemento a uno)

  `<valor` Los 8 bits menos significativos de la expresión "valor". Equivale a (valor&0xff)

  `>valor` Los 8 bits más significativos de la expresión valor. Equivale a (valor>>8)

- Operadores binarios:

  `a+b` La suma de las expresiones a y b

  `a-b` La resta de a y b

  `a*b` El producto de a por b

  `a/b` El cociente entero de la división de a entre b

  `a%b` El resto (módulo) de la división de a entre b

  `a&b` La AND lógica de los bits de a y de b

  `a|b` La OR lógica de los bits de a y de b

  `a^b` La función O-exclusiva (XOR) de los bits de a y de b

  `a>>b` a desplazado b bits a la derecha. Se introducen ceros en los bits MSB.

  `a<<b` a desplazado b bits a la izquierda. Se introducen ceros en los bits LSB.

- Paréntesis:

  Permiten especificar el orden de las operaciones en expresiones complejas. A continuacón se muestran algunos ejemplos:

```
indice= (etiq2-etiq1)/2
pos= base+(resvd+indice)*2
divider= (FCLK/16+BAUD/2)/BAUD-1
TSTI R0,(1<<RX_AVIL)
ANDI R0,(~((1<<ENAB)|(1<<PWON)))&0xf
org (.+0x000f)&0xfff0 ; alinea a múltiplo de 16
```

Las expresiones se evalúan usando aritmética entera de 32bits, lo que suele evitar los desbordamientos en los resultados intermedios, pero no se comprueba tal eventualidad.

13.3 Fichero de salida

El fichero de salida contiene una secuencia de segmentos de memoria listados como valores hexadecimales de 16 bits. Cada segmento comienza con una marca de dirección del tipo "`@abcd`" a la que sigue un listado de los datos del segmento.

Ejemplo con dos segmentos, uno de una única palabra en la dirección 0x0000 y otro de 5 palabras en la dirección 0x0200:

```
@0000
F200
@0200
6F02
F002
FFFE
0081
6887
```

Este tipo de ficheros se pueden leer en lenguaje Verilog mediante:

```
$readmemh
```

13.4 Opciones de línea de comando

La línea de comando del ensamblador cruzado para la CPU de un ciclo es:

```
./ucasm [opciones] sourcefile.asm
```

Hay dos posibles opciones para indicar en la línea de comandos:

`-o outputfile.hex` Indica el nombre del fichero de salida. Por defecto es "out.hex"

`-l listfile.lst` Indica el nombre del fichero de listado. Por defecto es "out.lst"