

Diseño del microprocesador GUS-16

Jesús Arias Alvarez

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

Universidad de Valladolid

Contents

1	Introducción	2
1.1	¿Por qué GUS-16?	2
2	Unidad Aritmético-Lógica (ALU)	2
3	Primer diseño: CPU Harvard de 1 ciclo por instrucción	8
3.1	Formato de los códigos de operación	9
3.2	Operandos inmediatos	9
3.3	Repertorio de instrucciones	12
4	Segundo diseño: CPU Von Neumann con pipeline	13
4.1	Instrucciones de la CPU con pipeline	16
4.2	Diseño de los bloques funcionales en Verilog	17
4.3	Primeras pruebas	26
5	Tercer diseño: CPU con interrupciones	26
5.1	Registros dobles	28
5.2	Lógica de cambio de modo e instrucción RETI	31
6	Cuarta variante: Constantes simplificadas e interrupciones concatenadas	32
6.1	CPU sin registros RH.	32
6.2	Interrupciones concatenadas	35
7	Quinta variante: Load y Store con desplazamiento inmediato	37
8	Sexta y séptima variantes. Nuevo juego de instrucciones	42
8.1	Versión 7	46
9	Versión V8	46
9.1	Modelo del programador	47
9.2	Estructura interna	48
9.3	Repertorio de instrucciones	50
9.4	Interfaz del core	52

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.

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

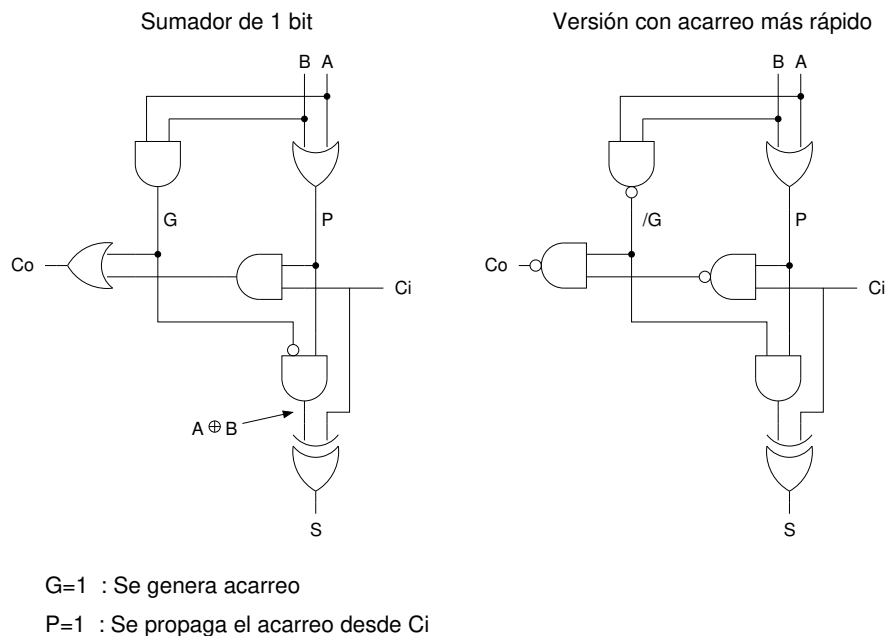


Figure 1: Esquema de un sumador de 1bit

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 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: $A \text{ menos } B = A \text{ más } \text{NOT}(B) \text{ más } 1$. 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

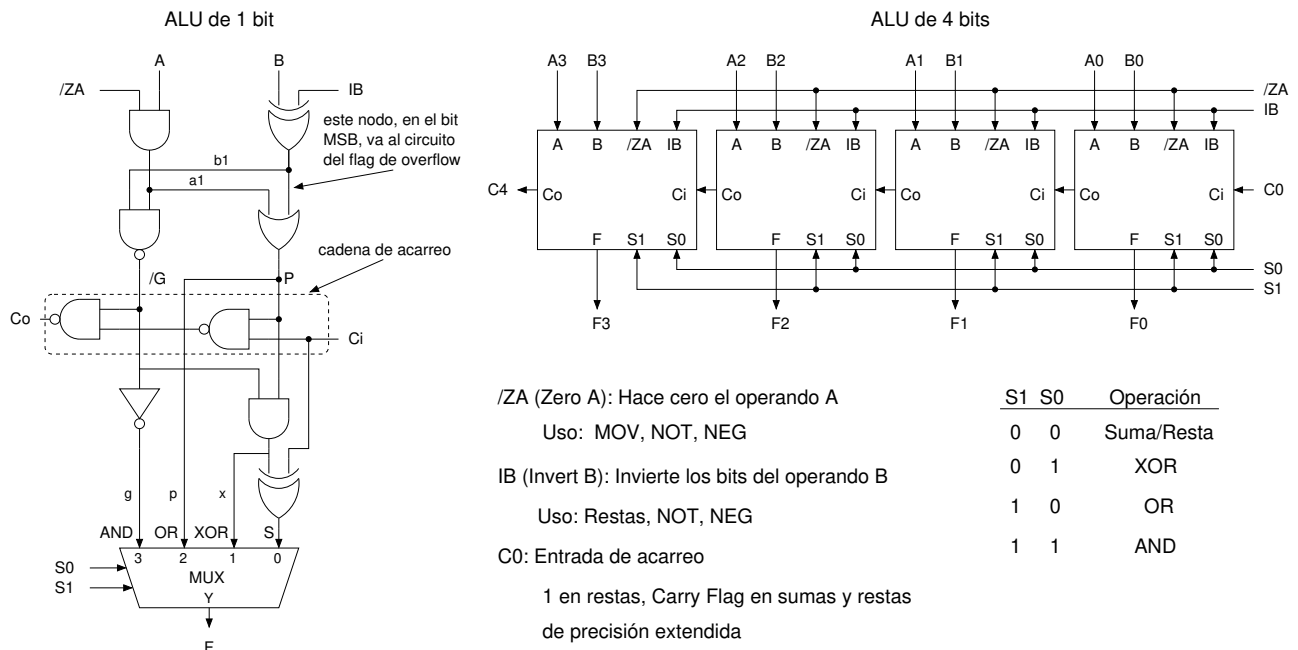


Figure 2: Esquemático de una ALU de 1bit y conexionado en cascada para obtener ALUs del número de bits requerido (ejemplo de 4 bits)

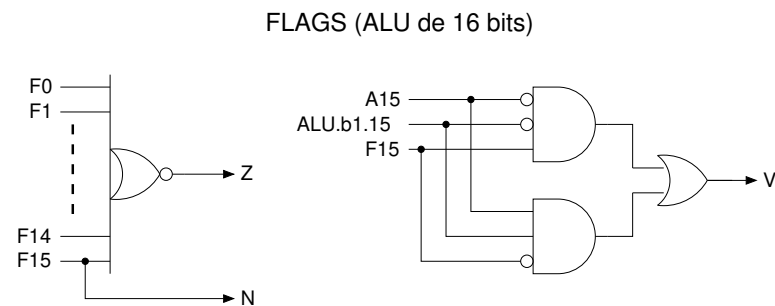
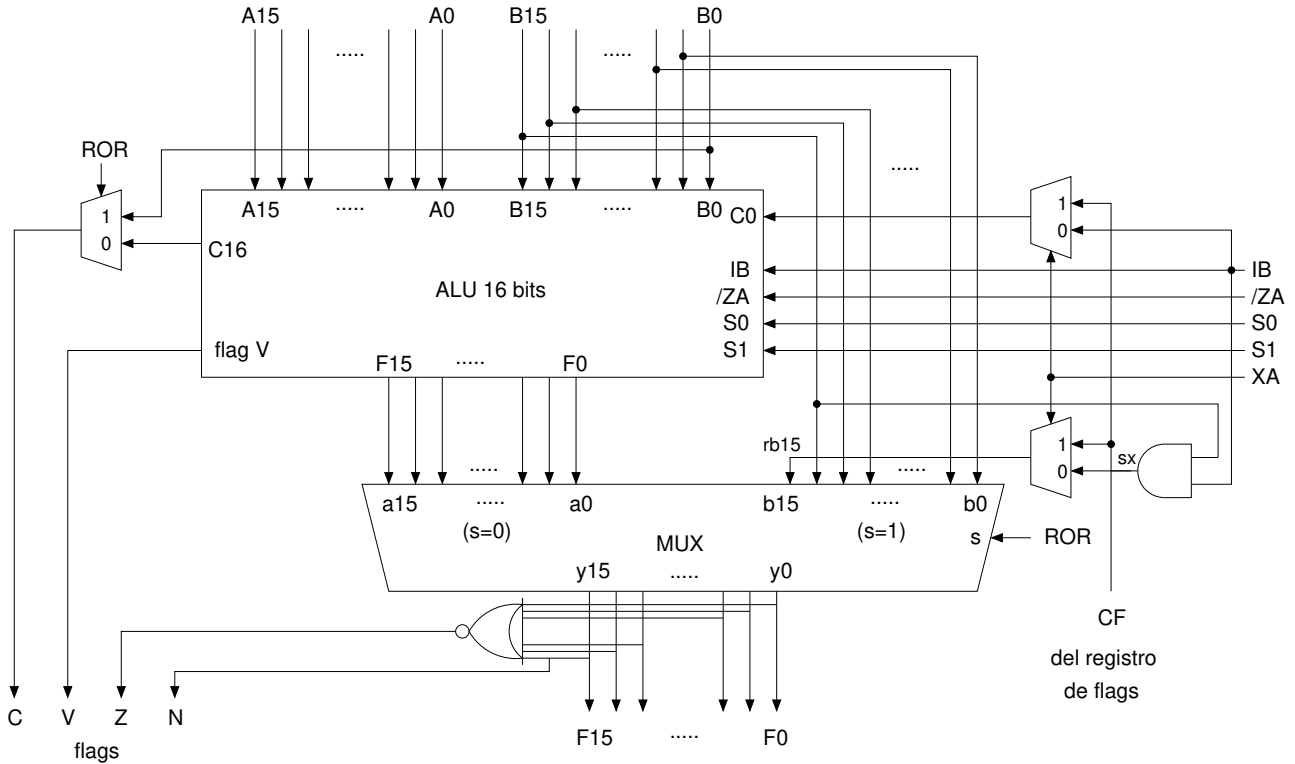


Figure 3: Flags de salida de la ALU

ALU de 16 bits con rotación/desplazamiento a la derecha



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.

indica el signo, y un valor 1 significa que el 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 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 en 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)
    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 extendido (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 explícitamente 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 al,ng,p,g,x,s;
assign al=a&za;
assign b1=b^ib;
assign ng=~(al&b1);
assign g=~ng;
assign p=(al|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;
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

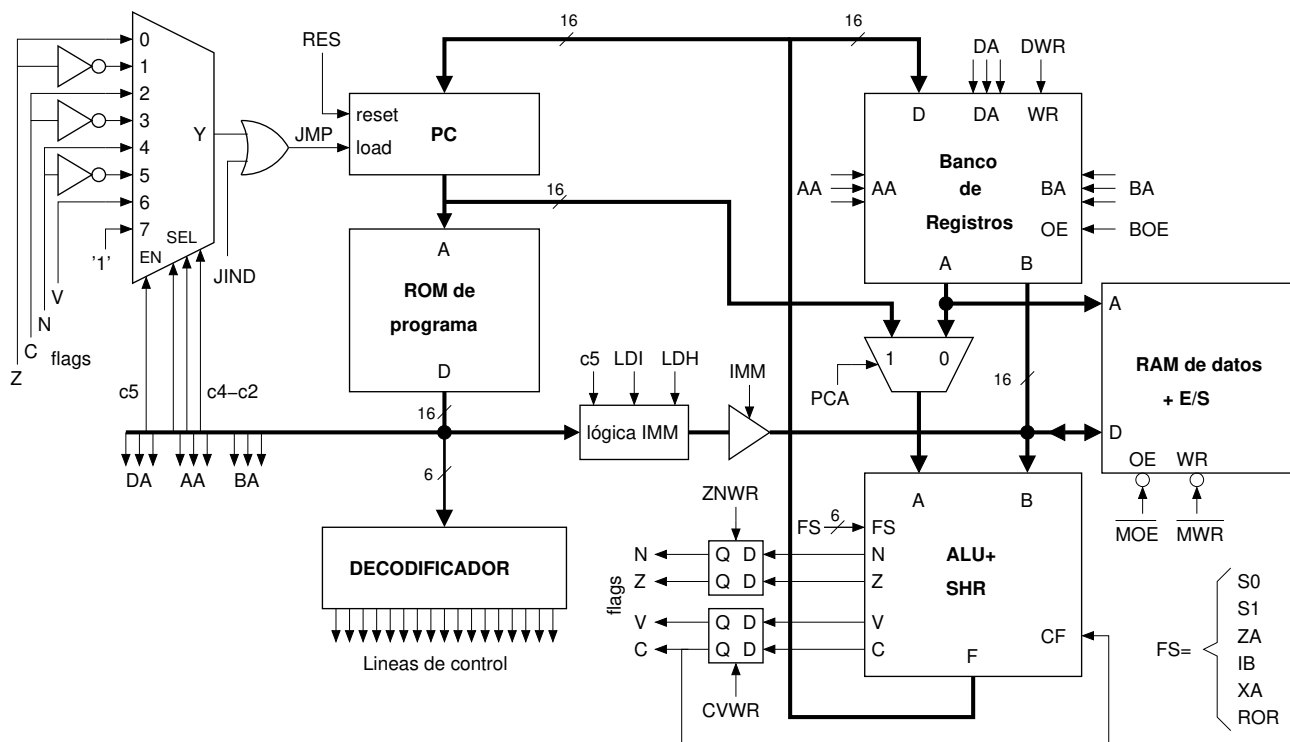


Figure 5: Esquema de la CPU

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 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).

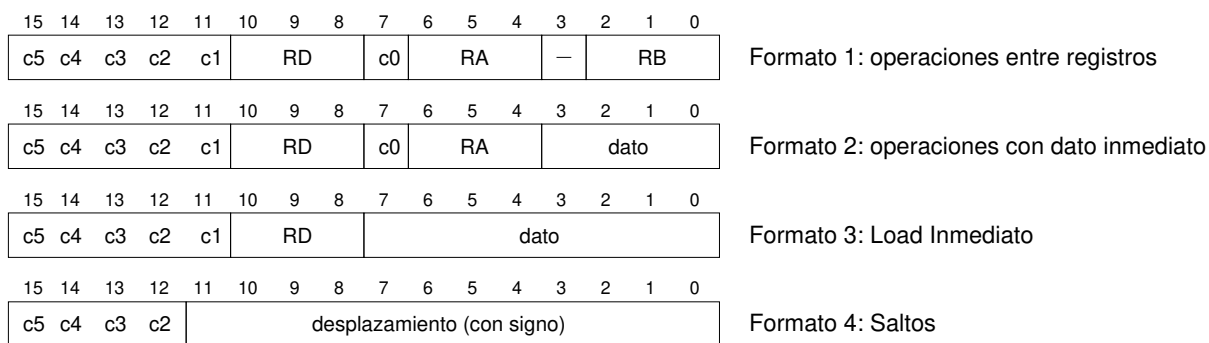


Figure 6: Formatos de los códigos de operación

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.

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
```

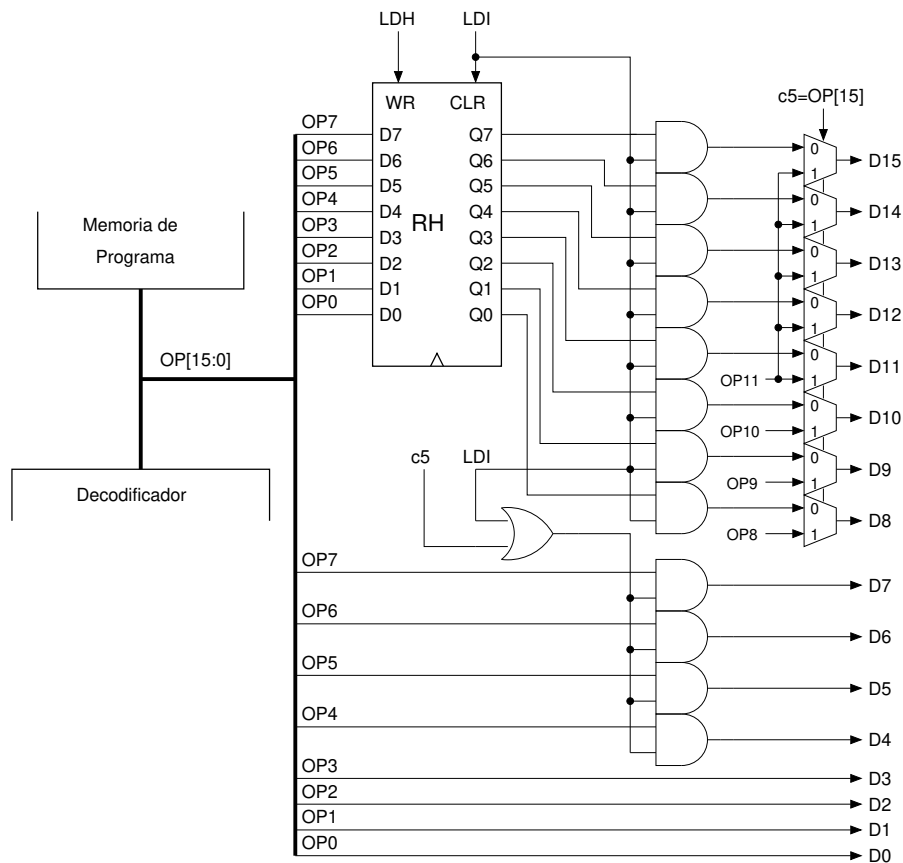


Figure 7: Lógica para la carga de constantes inmediatas.

```
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
```

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	0	1		RD	0		RA	—		RB			ADC	C V N Z	RD=RA+RB+Cflag
0	0	0	0	0	1		RD	1		RA			dato			ADCI	C V N Z	RD=RA+dato+Cflag
0	0	0	0	1	0		RD	0		RA	—		RB			SUB	C V N Z	RD=RA-RB
0	0	0	0	1	0		RD	1		RA			dato			SUBI	C V N Z	RD=RA-dato
0	0	0	0	1	1		RD	0		RA	—		RB			SBC	C V N Z	RD=RA-RB-(~Cflag)
0	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.

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)
```

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

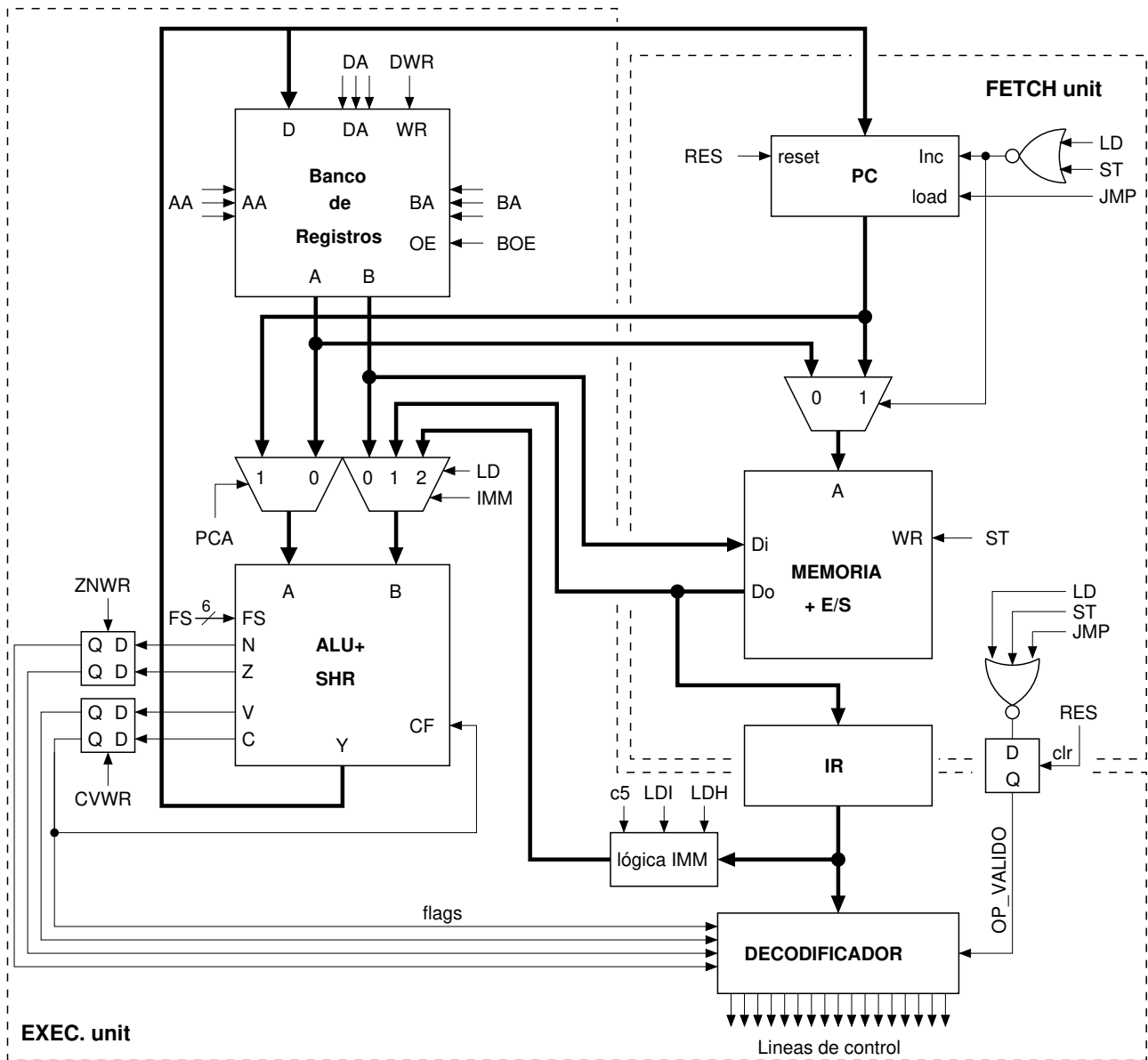


Figure 9: Diagrama de bloques de la CPU mejorada con pipeline en versión para síntesis en FPGA (sin triestados).

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.

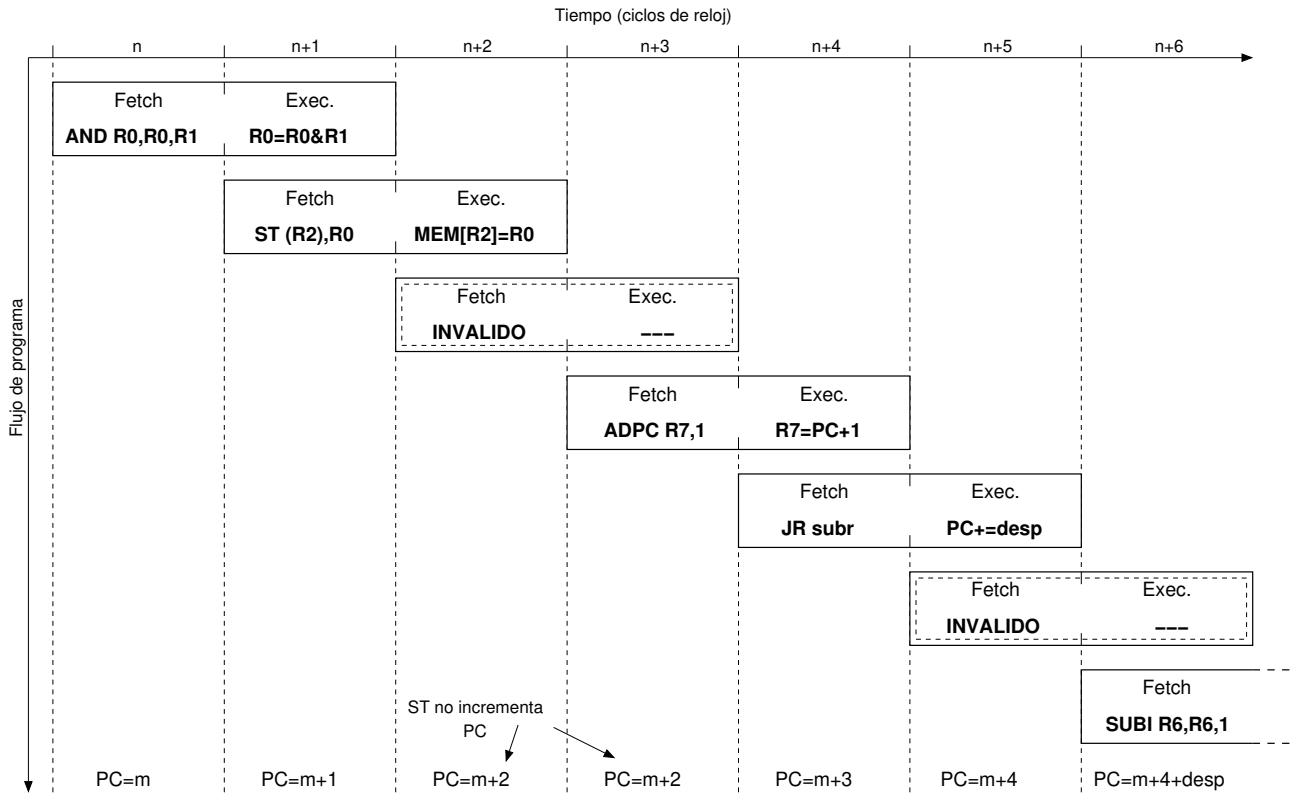


Figure 10: Flujo de ejecución de un segmento de código donde se destacan los dos ciclos empleados por cada instrucción, cómo las fases de ejecución y de búsqueda de código de operación se solapan, y cómo ciertas instrucciones (ST, JR) dan lugar a la carga de códigos inválidos que hacen que sus tiempos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

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 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 wrvc,     // Escribir flags C y V si 1
    input clk
);
reg qc,qv,qz,qn;
always @(posedge clk) begin
    qc <= wrvc ? dc: qc;
    qv <= wrvc ? 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 wrvc,    // 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
    endcase

```

```

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 estar 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
        ; 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

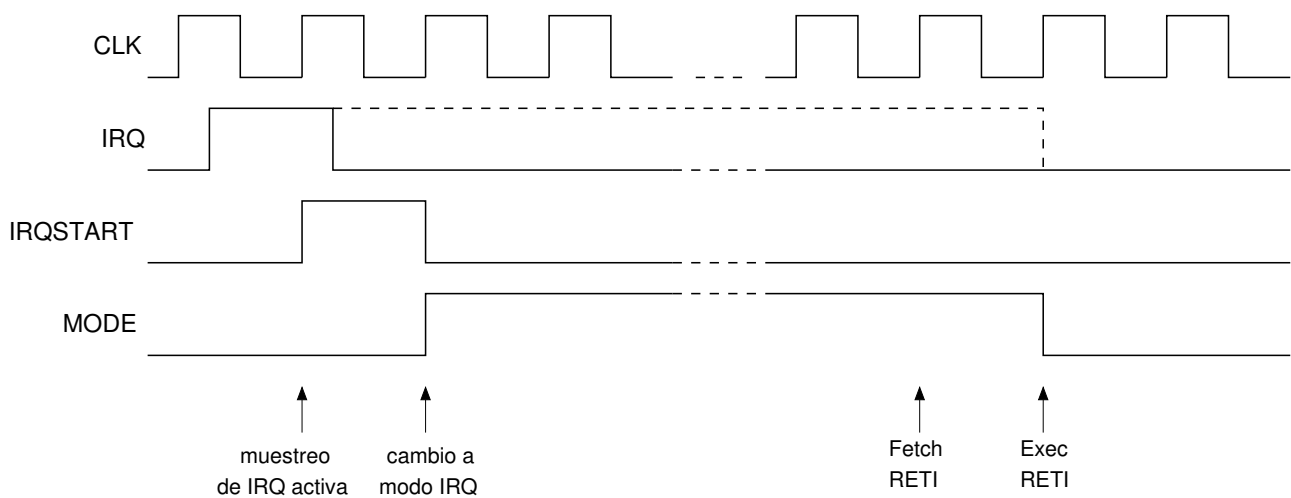


Figure 11: Cronograma de las señales involucradas en la ejecución de una rutina de interrupción.

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.

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.

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

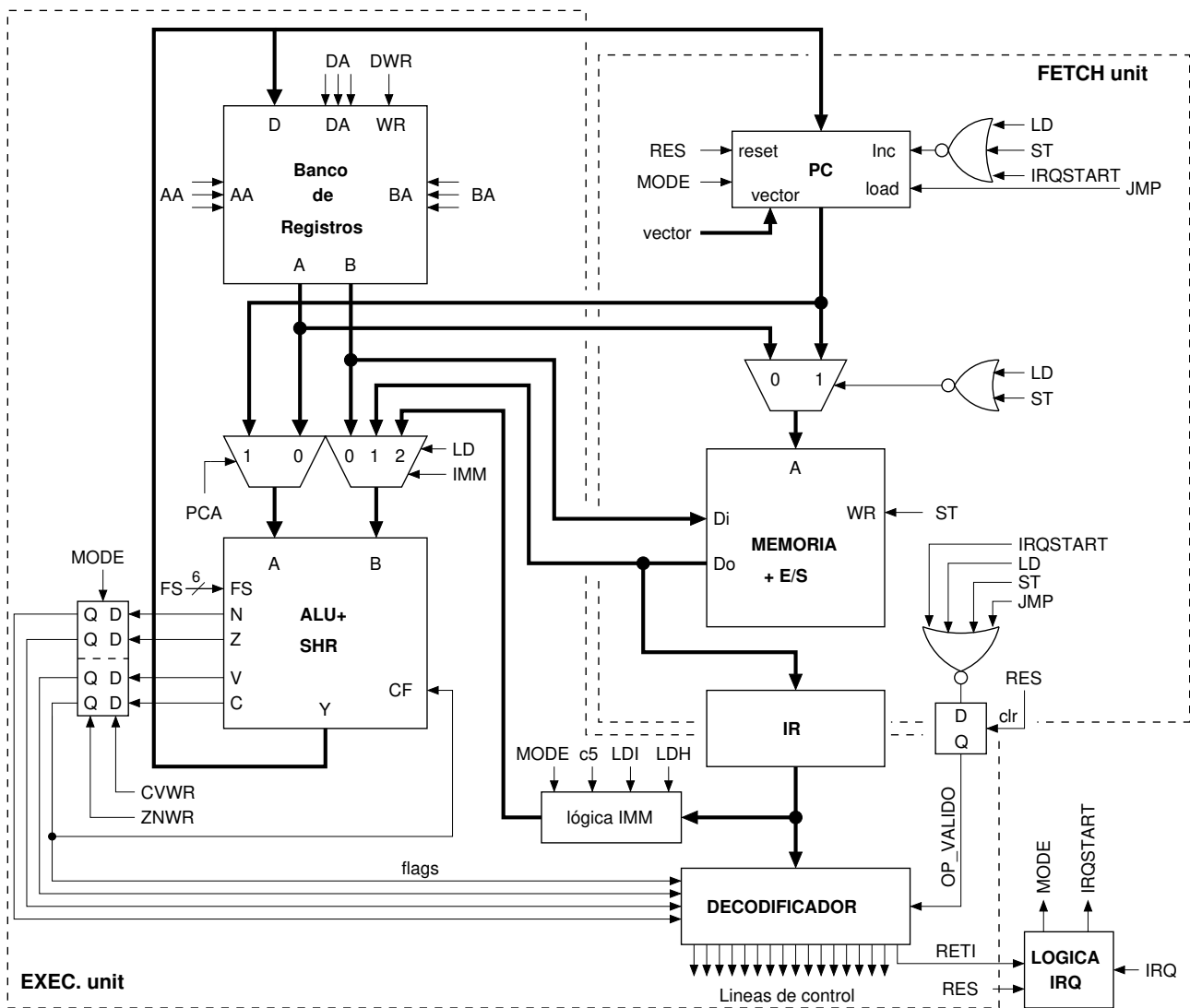


Figure 12: Diagrama de bloques de la CPU con interrupciones.

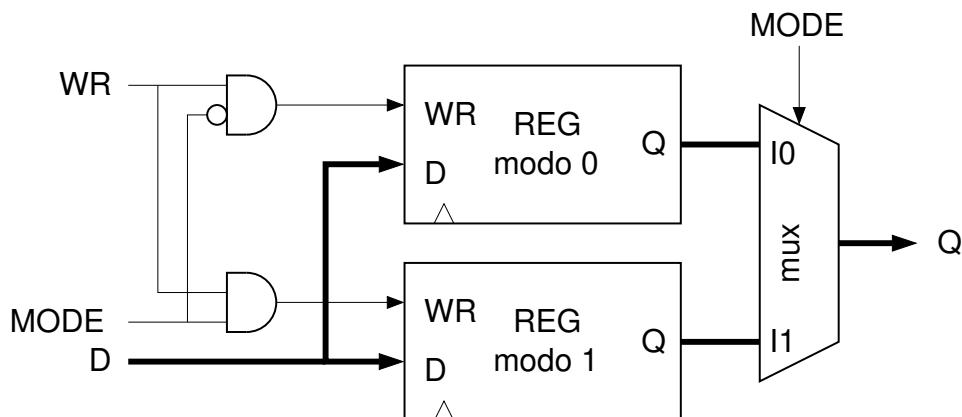


Figure 13: Diagrama simplificado de los registros dobles Flags, y RH

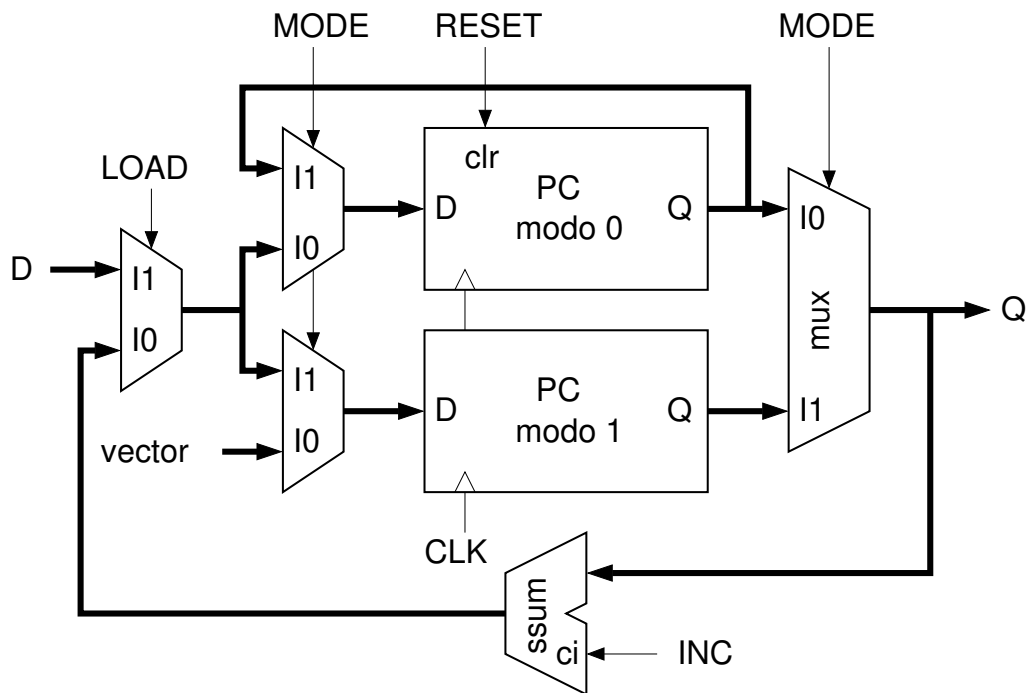


Figure 14: Esquema del registro PC doble.

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;
// registro PC modo IRQ
always @(posedge clk )
q1<= mode ? li : vector;
```

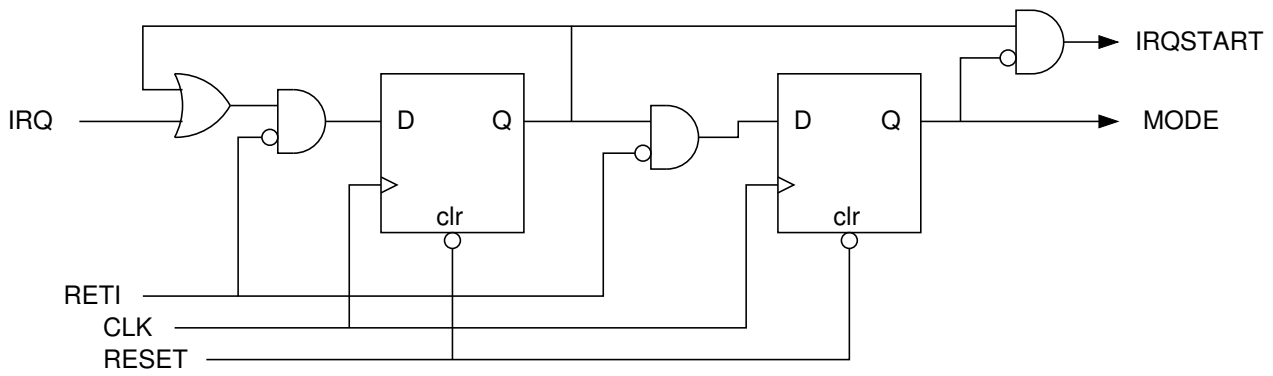


Figure 15: Lógica de control de interrupciones.

```
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 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:

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)

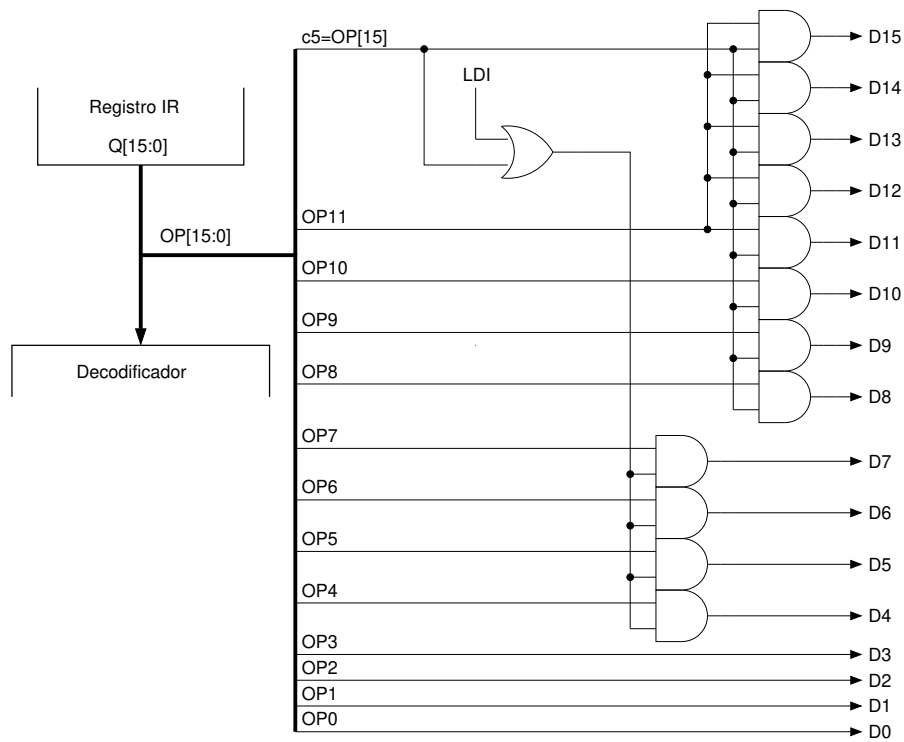


Figure 17: Lógica para la carga de constantes inmediatas en la CPU v4.

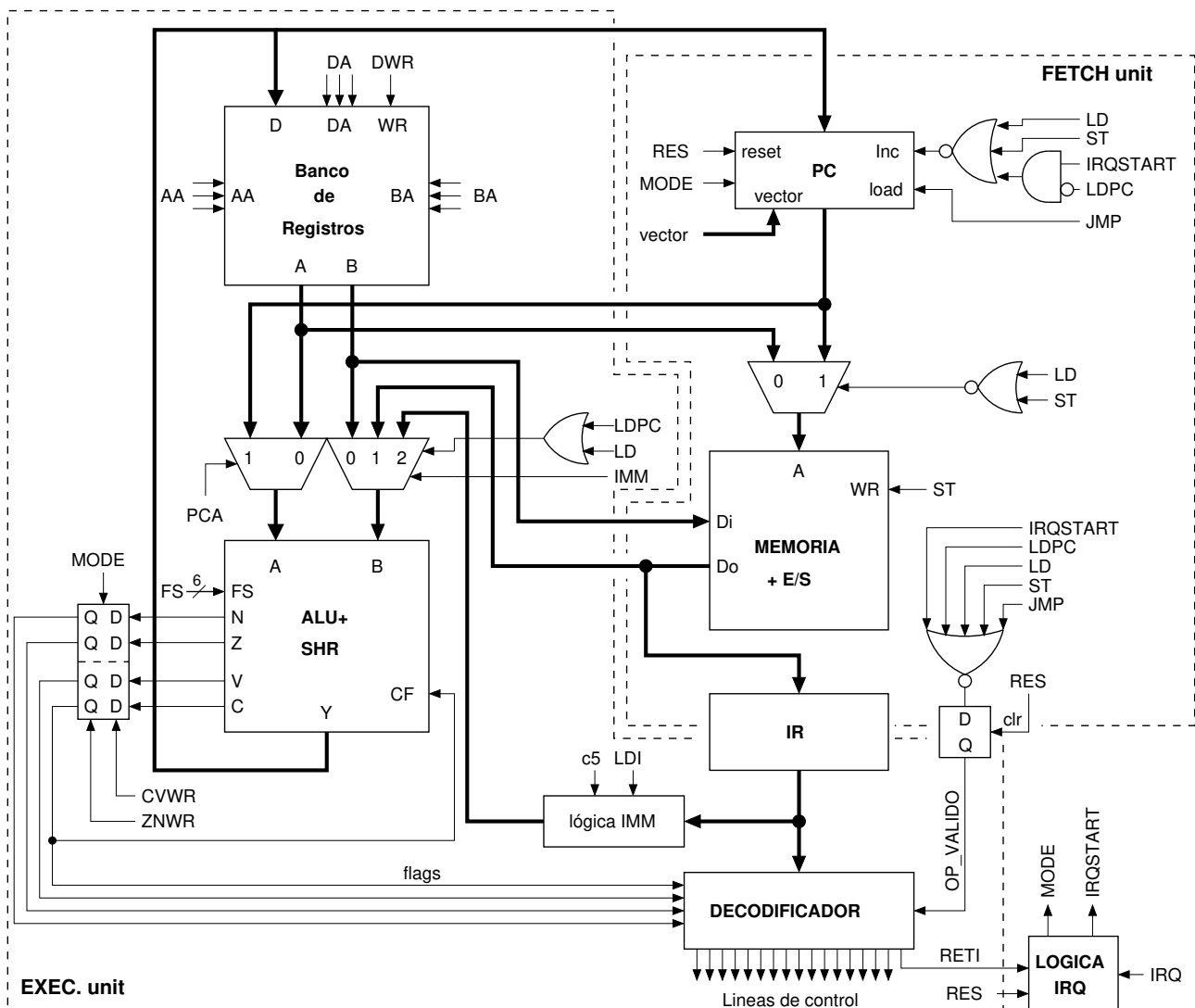


Figure 18: Diagrama de bloques de la CPU v4.

```

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:

CPU v3			CPU v4	
LDH >dato	; (MSB)		LDPC Rx	
LDI Rx,<dato	; (LSB)		word dato	

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.

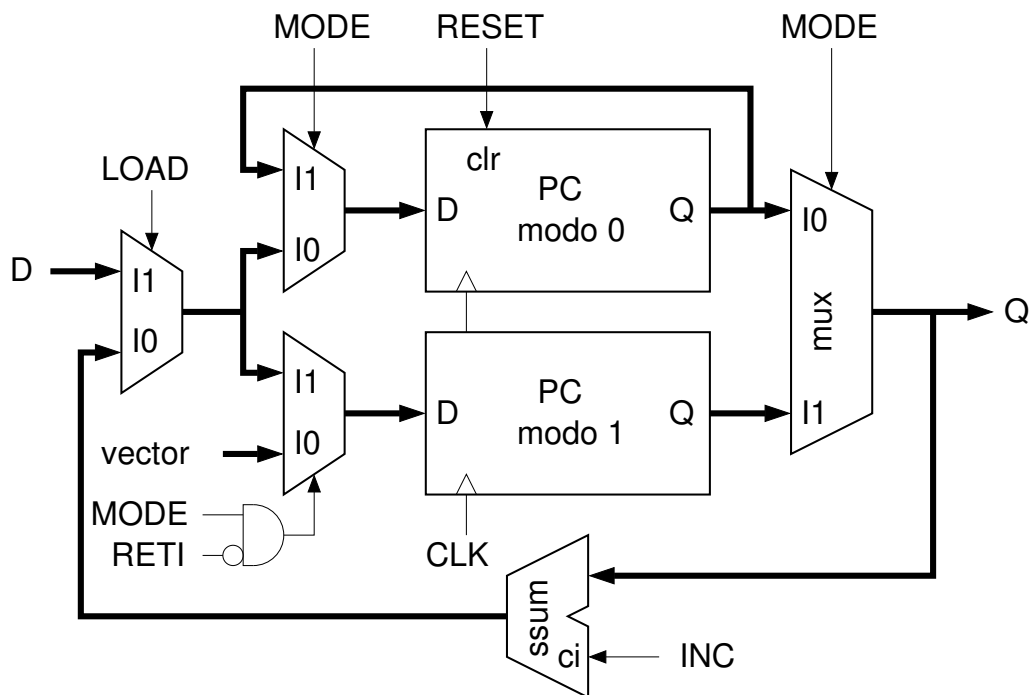


Figure 19: Esquema del registro PC doble modificado para la concatenación de interrupciones.

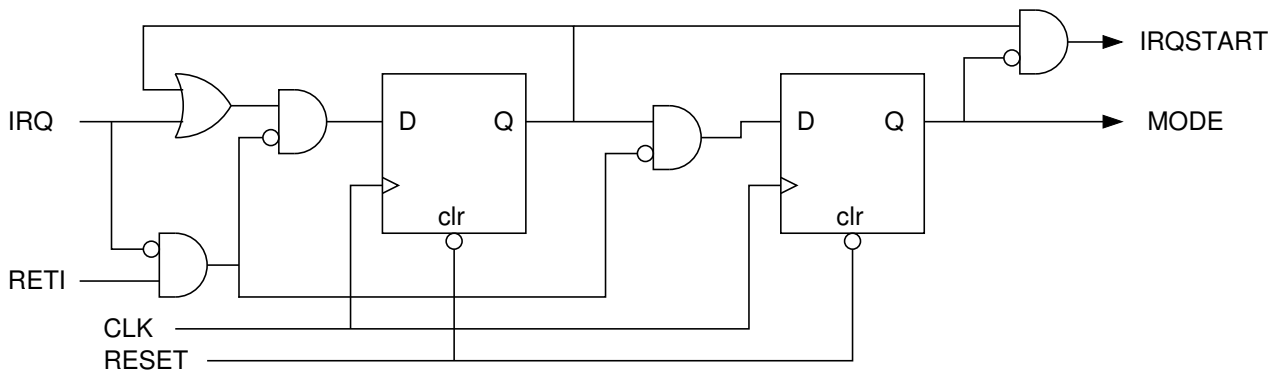


Figure 20: Lógica de control de interrupciones modificada para dar soporte a las interrupciones concatenadas.

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 reestructuración 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.

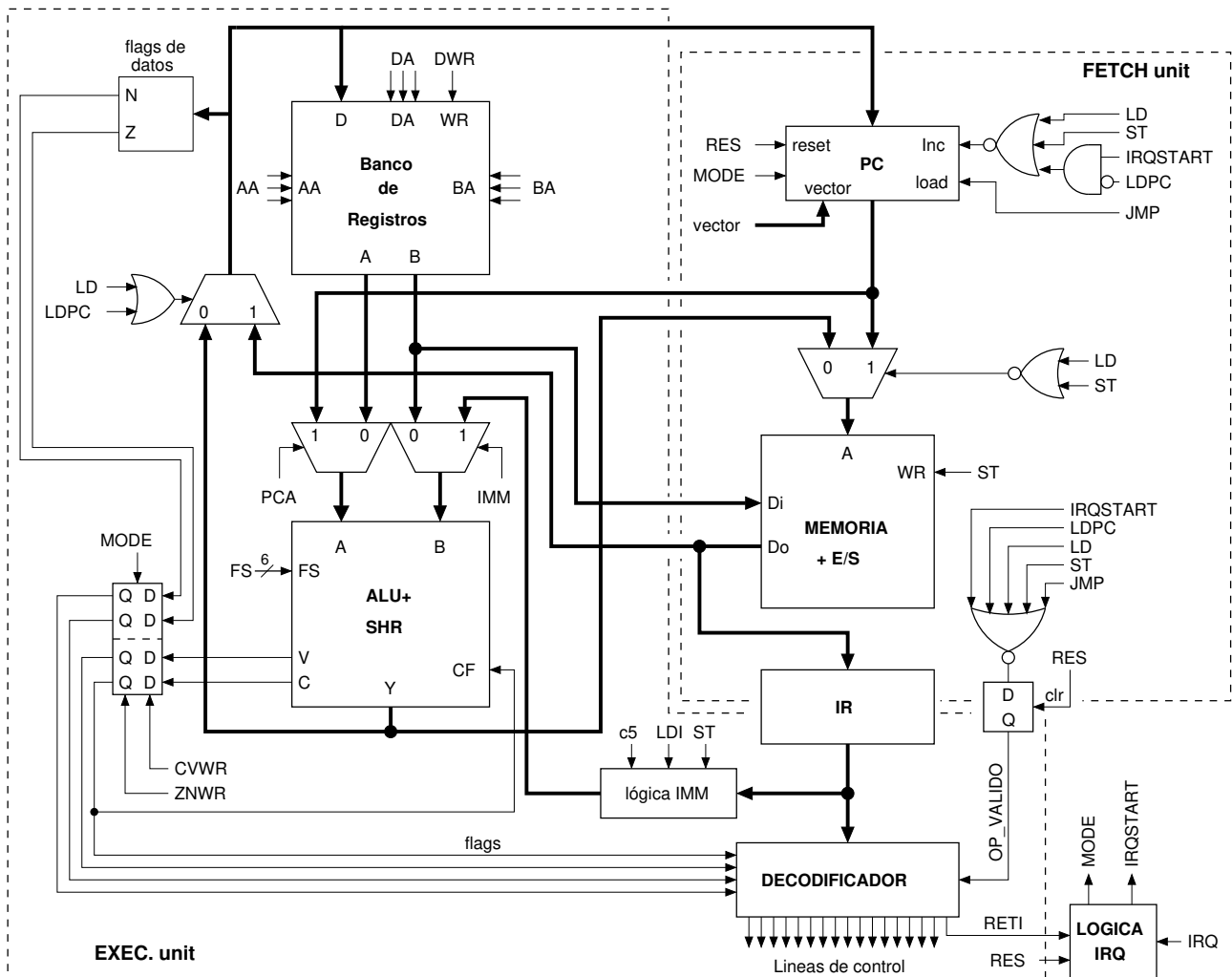


Figure 22: Diagrama de bloques de la CPU v5.

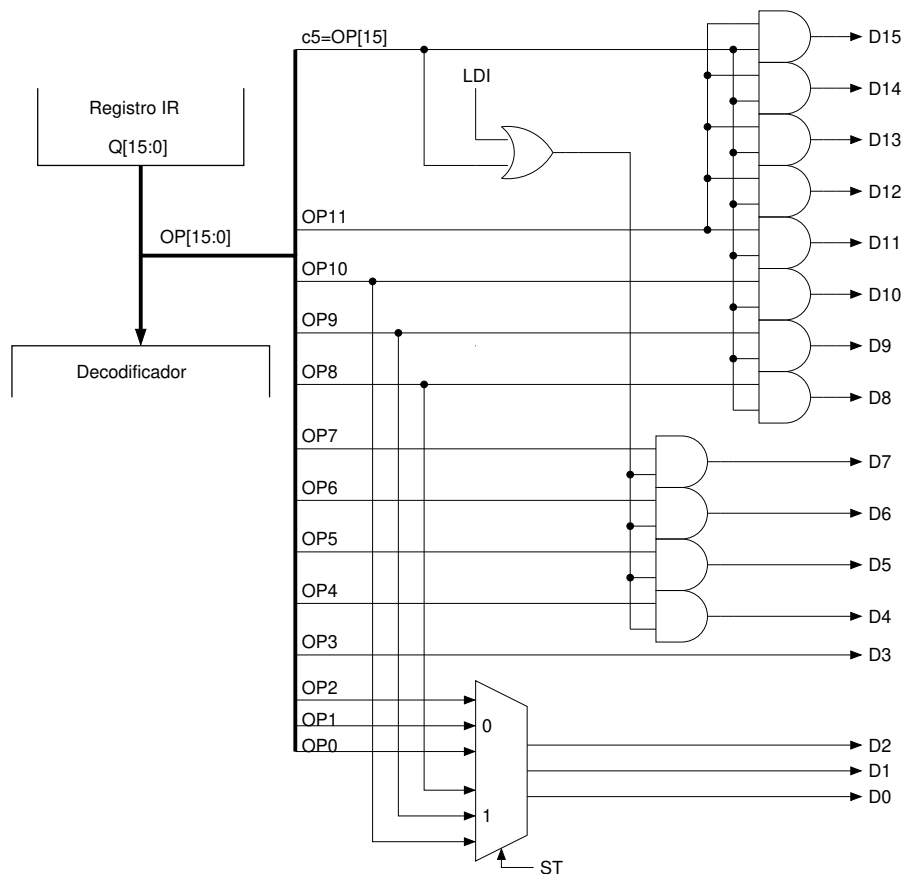


Figure 23: Lógica para la carga de constantes inmediatas en la CPU v5.

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 procesador RISC-V ;)

8 Sexta y séptima variantes. 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:

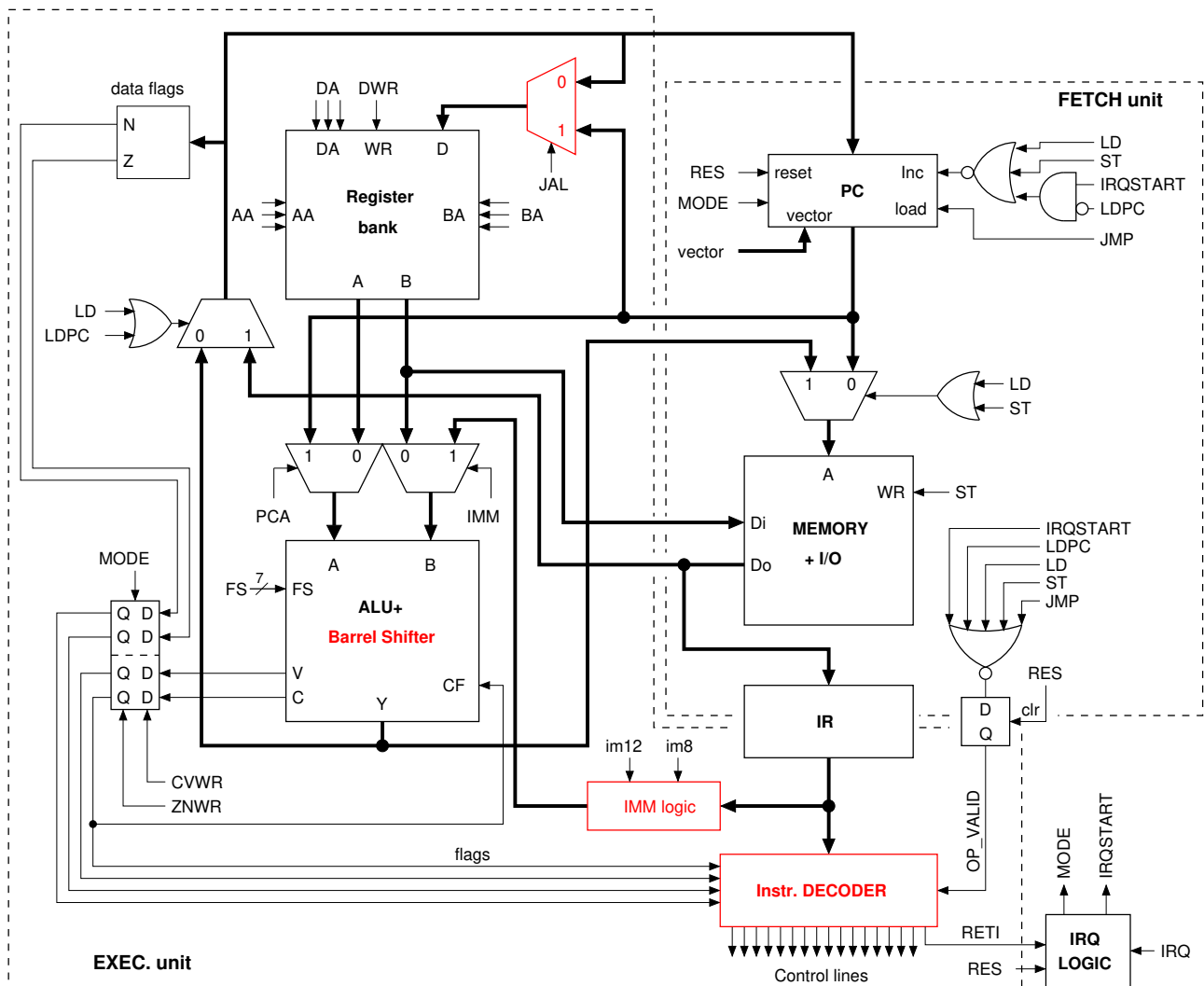
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	
oph RD RA RB opl	Format 1: 3 regs
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	
op RD,RA ulit8	Format 2: Literal operand
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	
oph RD opl1 RB opl2	Format 3: 2 regs
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	
oph RD opl ulit4h RB ulit4l	Format 4: RORI
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	
op RD RA udisp5	Format 5: Load
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	
op RB RA udisp5	Format 6: Store
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	
op sdisp12	Format 7: Jumps

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.

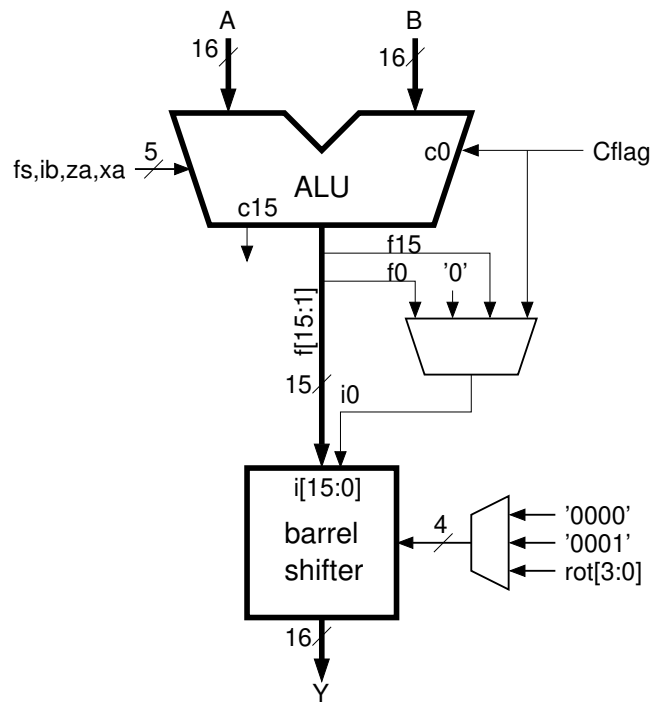
La tabla completa con las instrucciones es:

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	0	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	

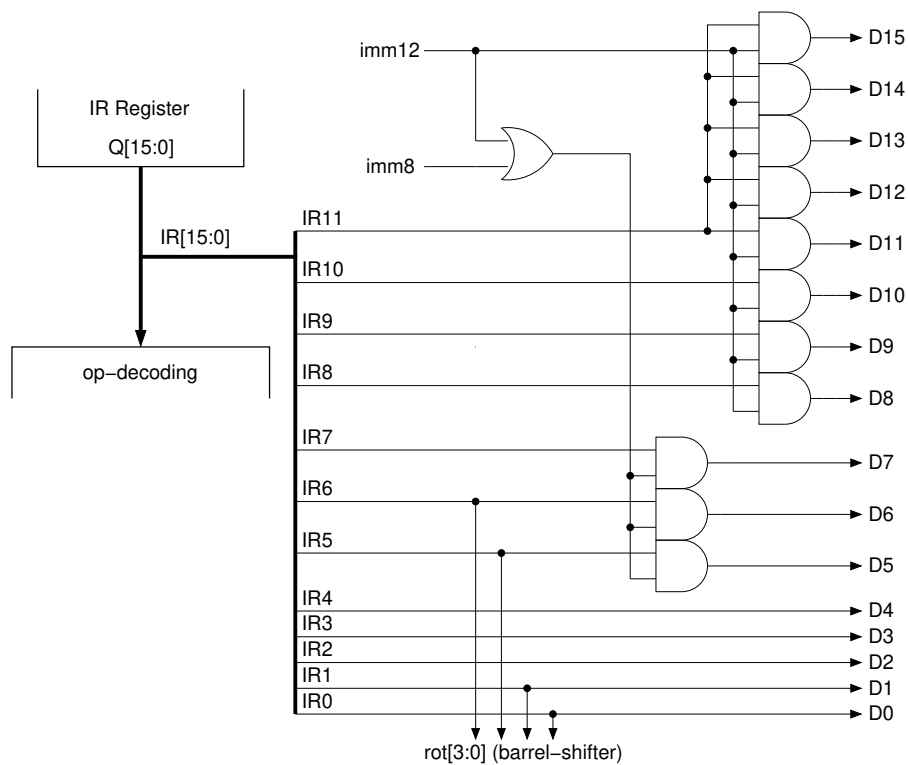
El diagrama de bloques del nuevo procesador se muestra en la siguiente figura, en la que los bloques nuevos o modificados se señalan en color rojo. El bloque con más cambios es sin duda el decodificador de instrucciones, seguido por la ALU. El bloque de operandos inmediatos ha cambiado poco.



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 elige 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.

8.1 Versión 7

Esta versión supone un único cambio en el juego de instrucciones. La instrucción BIC se sustituye por:

ROR Rdest, Rsource, Rnbits

Esto ha supuesto añadir una cuarta entrada al multiplexor de las rotaciones del “barrel-shifter”. Como anécdota señalar que esta variante del core se hizo con vistas a una aplicación para controlador de bus JTAG, donde en una subrutina se necesitaba hacer un desplazamiento de un número variable de bits.

9 Versión V8

Esta ha sido hasta la fecha la mayor revisión del core GUS16, propiciada por Carlos Venegas para poder portar a este core un compilador de lenguaje C / C++. El principal objetivo ha sido el disponer de un tipo de datos de 8 bits, lo que ha supuesto un importante cambio en el direccionamiento de la memoria. Este es un resumen de las nuevas características:

- Direcciones de bytes, no de palabras de 16 bits. Esto ha reducido a la mitad el espacio de memoria direccionable, quedando en 32768 palabras (o 65536 bytes), lo que no creemos que sea una gran limitación en la práctica, pero que además supone:
 - El bus de direcciones no tiene un bit #0.
 - Hay dos nuevas señales para seleccionar la parte alta o baja de los buses de datos: BHE y BLE.
 - El registro PC tiene su bit #0 siempre en 0, o lo que es lo mismo: siempre apunta a direcciones pares.
 - Nuevas instrucciones para lectura de bytes: LDB y LDBS. Estas instrucciones pueden apuntar a direcciones impares, y LDBS además extiende el signo del byte leído.
 - Nueva instrucción para escritura de bytes: STB. También puede apuntar a direcciones impares.
 - Arquitectura Little-Endian: El byte menos significativo de un dato de 16 bits está en una dirección par.
- Repertorio de instrucciones ampliado. Incluye todas las instrucciones de la versión V7, aunque con distinta codificación, además otras nuevas.
- Desplazamientos de 5 bits con signo para las instrucciones de los tipos Load y Store.

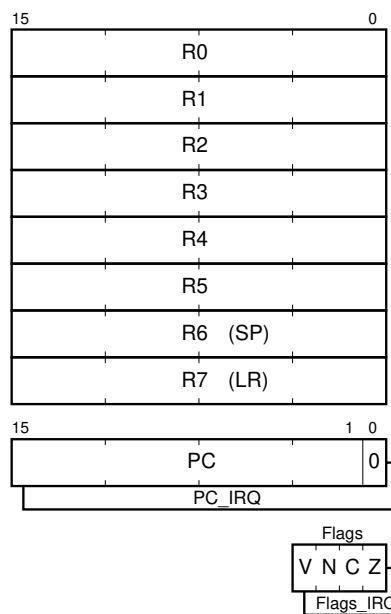


Figure 24: Registros del procesador GUS16 V8

9.1 Modelo del programador

En la figura 24 se muestran los registros que están visibles para el programador. Estos son:

- R0 a R7. Registros de propósito general. Aunque en las primeras versiones del core todos ellos eran equivalentes en la versión V8 el registro R7 es especial pues es el registro en el que se guarda la dirección de retorno en las llamadas a subrutinas (link register, LR). El resto de los registros son realmente equivalentes, pero por convención el registro R6 se usará como puntero de pila en el software (stack pointer, SP).
- PC. Program counter. Hay dos de estos registros: uno está en uso durante la ejecución normal de los programas y otro lo sustituye cuando se salta a una rutina de interrupción. La instrucción RETI, que sólo deberá usarse al final de estas rutinas, recupera el PC del modo normal con lo que continúa la ejecución del programa interrumpido. Esos registros realmente son de 15 bits ya que su bit LSB está siempre fijo en cero, con lo que sólo pueden apuntar a direcciones pares de la memoria.
- Flags. Son cuatro bits cuyo valor depende de los resultados de las instrucciones ejecutadas anteriormente y que intervienen principalmente en la ejecución o no de los saltos condicionales. Al igual que ocurría con el PC, tenemos dos registros de flags: uno para el modo de ejecución normal y otro para las rutinas de interrupción. Y también en este caso la instrucción RETI recupera los flags del modo normal con los mismos valores que tenían antes de producirse la interrupción. Los flags son:
 - Z, Cero. Se pone en uno si el resultado de una operación es cero. Nótese que las instrucciones de tipo Load modifican este flag.
 - C, Acarreo. Se pone en uno cuando hay una llevada en el bit #15 en las operaciones de suma, o cuando NO hay llevada en el bit #15 en las operaciones de resta o comparaciones. También puede cargarse con el bit #0 del operando fuente en los desplazamientos de un bit a la derecha. Su valor, aparte de afectar a los saltos condicionales, también interviene en un buen número de instrucciones tales como las de suma y resta con acarreo, y en la rotación RORC.
 - N, Negativo. Es una copia del bit #15 del resultado de la operación. Si está en 1 indica que el dato es negativo en la aritmética de “complemento a dos”. Nótese que las instrucciones de tipo Load modifican este flag.

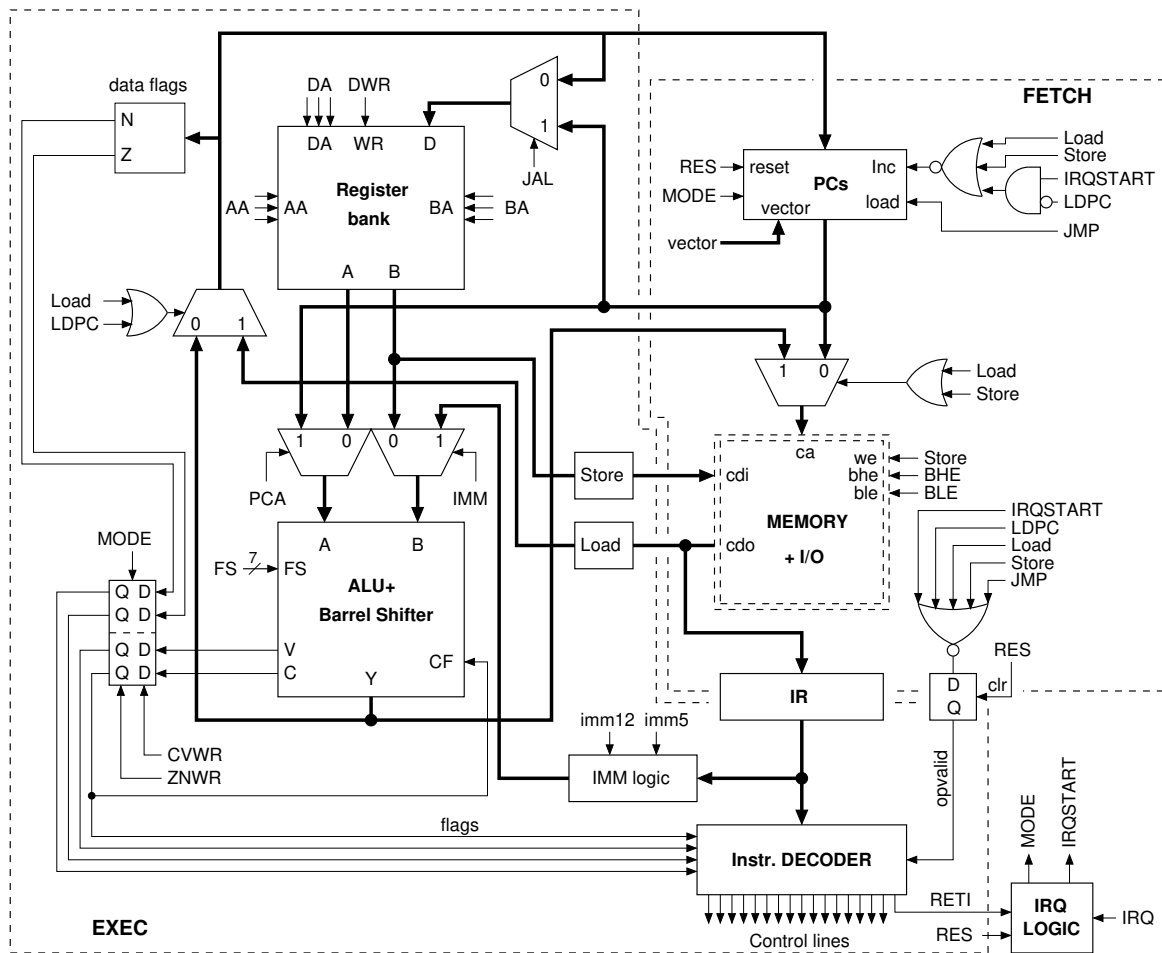


Figure 25: Diagrama de bloque del procesador GUS16 V8

- V. Overflow. Se pone en uno cuando hay un cambio inesperado en el signo del resultado en operaciones aritméticas, como cuando al sumar dos datos positivos se obtiene un resultado negativo. Este flag realmente sólo tiene sentido cuando los datos son variables con signo en la aritmética de “complemento a dos”.

9.2 Estructura interna

El diagrama de bloque del procesador GUS16 V8 se muestra en la figura 25, y si lo comparamos con el del de la versión V6 podemos comprobar que los únicos bloques nuevos son los etiquetados como 'Load' y 'Store' lo cual no significa que no haya otros cambios importantes dentro de otros bloques (principalmente en 'IMM logic', 'PCs', e 'Instr. Decoder')

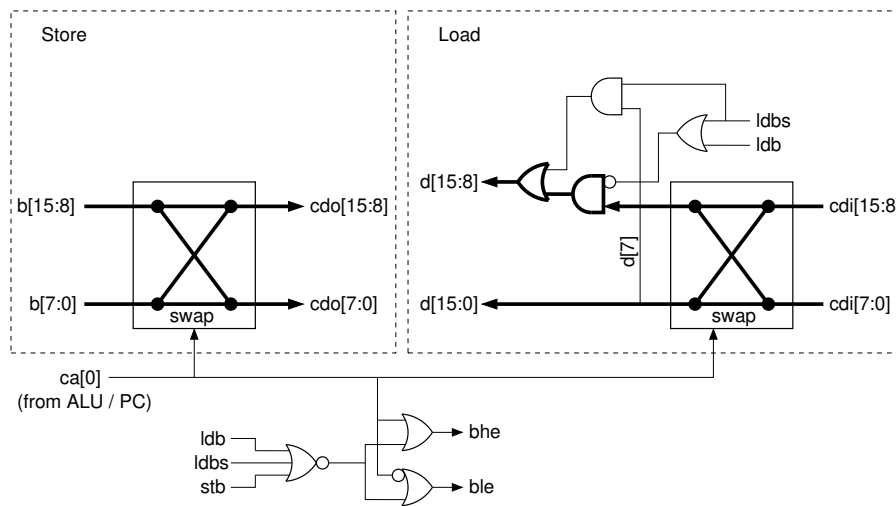
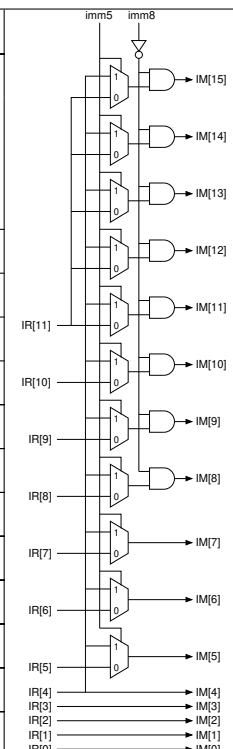


Figure 26: Lógica de los bloques Load y Store

Los bloques Load y Store tienen como cometido el traspasar datos de tamaño byte entre el banco de registros y la memoria (los datos de 16 bits pasan directos). En la figura 26 se muestra la estructura interna de estos bloques, donde podemos comprobar que en el caso de las escrituras (Store) se trata de un simple circuito que intercambia los bytes alto y bajos del dato cuando la dirección es impar, mientras que en las lecturas hay un poco más de lógica encargada de hacer que los 8 bits altos del resultado se hagan todos cero en el caso de la instrucción LDB, o una copia del bit de signo del byte (bit #7) si se ejecuta LDBS

El bloque 'IMM logic' se modificó notablemente respecto de la versión v6 pues ahora los desplazamientos a sumar a la dirección del registro base de las instrucciones Load y Store tienen signo. Los operandos literales serán por tanto datos de 5 bits con signo para las instrucciones Load / Store, de 8 bits sin signo para todas las instrucciones con operando inmediato, y de 12 bits con signo para los saltos relativos. Si bien en este último caso el bit #0 es irrelevante dado que en el PC este bit siempre es cero (y de hecho se ha reutilizado como parte del código de operación de las instrucciones de salto). Estos detalles están resumidos en la siguiente tabla:

bit #	LD/ST	Imm.	Jumps
15	IR[4]	0	IR[11]
14			
13			
12			
11			IR[11]
10			IR[10]
9			IR[9]
8			IR[8]
7	IR[4]	IR[7]	IR[7]
6		IR[6]	IR[6]
5		IR[5]	IR[5]
4	IR[4]	IR[4]	IR[4]
3	IR[3]	IR[3]	IR[3]
2	IR[2]	IR[2]	IR[2]
1	IR[1]	IR[1]	IR[1]
0	IR[0]	IR[0]	x



En resumen: los 5 bits bajos del registro IR pasan directos a la salida, mientras que los bits 5 a 7 pueden tener dos valores distintos, y los 8 bits restantes tienen tres valores posibles, aunque uno de ellos es cero.

9.3 Repertorio de instrucciones

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		ROR		– – N Z		RD = (RB>>RA) (RB<<16–RA)		
0 0 0 1 0					– – –		RA		RB		0 0		TST		– – N Z		RA & RB		
0 0 0 1 0					– – –		RA		RB		0 1		CMP		C V N Z		RA – RB		
0 0 0 1 1																			
0 0 1 0 0					RD		ulit8						ADDI		C V N Z		RD= RD + ulit8		
0 0 1 0 1					RD		ulit8						SUBI		C V N Z		RD= RD – ulit8		
0 0 1 1 0					RD		ulit8						ADCI		C V N Z		RD= RD + ulit8 +Cflag		
0 0 1 1 1					RD		ulit8						SBCI		C V N Z		RD= RD – ulit8 – ~Cflag		
0 1 0 0 0					RD		ulit8						ANDI		– – N Z		RD= RD & ulit8		
0 1 0 0 1					RD		ulit8						ORI		– – N Z		RD= RD ulit8		
0 1 0 1 0					RD		ulit8						XORI		– – N Z		RD= RD ^ ulit8		
0 1 0 1 1					RD		ulit8						CMPI		C V N Z		RD – ulit8		
0 1 1 0 0					RD		ulit8						TSTI		– – N Z		RD & ulit8		
0 1 1 0 1					RD		ulit8						LDI		– – – –		RD= ulit8		
0 1 1 1 0																			
0 1 1 1 1					RD		0	ulit4h		RB		ulit4l		RORI		– – N Z		RD = (RB>>uli4) (RB<<16–ulit4)	
0 1 1 1 1					RD		1	0 0		RB		0 0		RORC		C ? N Z		RD = {Cflag,RB>>1}, Cflag=RB0	
0 1 1 1 1					RD		1	0 0		RB		0 1		SHR		C ? N Z		RD = RB>>1, Cflag=RB0	
0 1 1 1 1					RD		1	0 0		RB		1 0		SHRA		C ? N Z		RD = RB>>1 (signed) , Cflag=RB0	
0 1 1 1 1					RD		1	0 1		RB		0 0		NOT		– – N Z		RD = ~RB	
0 1 1 1 1					RD		1	0 1		RB		0 1		NEG		C V N Z		RD = –RB	
0 1 1 1 1					RD		1	0 1		RB		1 0		MOV		– – – –		RD = RB	
0 1 1 1 1					RD		1	1 1		– 0 0		0 0		LDPC class	LI	– – – –		RD = Mem(PC++)	
0 1 1 1 1					– – –		1	1 1		– 0 1		0 0			JMP	– – – –		PC = Mem(PC)	
0 1 1 1 1					– – –		1	1 1		– 1 0		0 0			JMWL	– – – –		PC = Mem(PC), Rlink=PC	
0 1 1 1 1					– – –		1	1 1		0 0 0		0 1		NOP *		– – – –		–	
0 1 1 1 1					– – –		1	1 1		– 0 1		0 1		WFI		– – – –		wait for interrupt	
0 1 1 1 1					– – –		1	1 1		– 1 0		0 1		BRK		– – – –		breakpoint	
0 1 1 1 1					– – –		1	1 1		RB		1 0		JIND		– – – –		PC = RB	
0 1 1 1 1					– – –		1	1 1		– – –		1 1		RETI		– – – –		return from interrupt	
1 0 0 0 0					RD		RA		sdisp5						LD		– – N Z		RD= Mem(RA+sdisp5) (16)
1 0 0 0 1					RD		RA		sdisp5						LDB		– – N Z		RD= Mem(RA+sdisp5) (u8)
1 0 0 1 0																			
1 0 0 1 1					RD		RA		sdisp5						LDBS		– – N Z		RD= Mem(RA+sdisp5) (s8)
1 0 1 0 0					RB		RA		sdisp5						ST		– – – –		Mem(RA+sdisp5)=RB (16)
1 0 1 0 1					RB		RA		sdisp5						STB		– – – –		Mem(RA+sdisp5)=RB (8)
1 0 1 1				sdisp11								0		JR		– – – –		PC = PC+sdisp11	
1 0 1 1				sdisp11								1		JAL		– – – –		PC = PC+sdisp11, Rlink=PC	
1 1 0 0				sdisp11								0		JZ		– – – –		PC = PC+sdisp11 if Zflag	
1 1 0 0				sdisp11								1		JNZ		– – – –		PC = PC+sdisp11 if ~Zflag	
1 1 0 1				sdisp11								0		JC		– – – –		PC = PC+sdisp11 if Cflag	
1 1 0 1				sdisp11								1		JNC		– – – –		PC = PC+sdisp11 if ~Cflag	
1 1 1 0				sdisp11								0		JMI		– – – –		PC = PC+sdisp11 if Nflag	
1 1 1 0				sdisp11								1		JPL		– – – –		PC = PC+sdisp11 if ~Nflag	
1 1 1 1				sdisp11								0		JV		– – – –		PC = PC+sdisp11 if Vflag	
1 1 1 1				sdisp11								1		JNV		– – – –		PC = PC+sdisp11 if ~Vflag	

En la tabla anterior se presentan el repertorio de instrucciones del procesador GUS16 V8, con las nuevas instrucciones marcadas en rojo. Hay que destacar que las instrucciones de la clase 'LDPC' tienen un opcode de 32 bits efectivos ya que todas ellas usan el PC como puntero de datos y tienen que ir seguidas de un dato de 16 bits. las nuevas instrucciones son:

- TST Ra, Rb : Test. Realiza la operación AND entre los dos registros y actualiza los flags Z y N, pero se descarta el resultado.
- CMP Ra, Rb : Comparison. Resta Ra - Rb y actualiza todos los flags, también C y V, pero se descarta el resultado.
- TSTI Rd, nn : Test Immediate. Realiza la operación AND entre el registro y el dato literal de 8 bits. Se actualizan Z y N, pero se descarta el resultado.
- MOV Rd, Rb : Move. Mueve datos entre registros sin afectar a los flags.
- LI Rd, imm16 : Load 16-bit Immediate (antes llamada LDPC). Debe ir seguida de un dato de 16 bits.
- JMP imm16 : Jump. Salto a dirección absoluta. Debe ir seguido de una palabra de 16 bits con la dirección destino del salto.
- JMPL imm16 : Jump & Link. Llamada a subrutina en dirección absoluta. Debe ir seguido de una palabra de 16 bits con la dirección de la subrutina. Además de ejecutar el salto, al valor actual de PC se le suma 2 y se escribe en el registro R7 (LR). De esta manera LR queda apuntando a la dirección de la siguiente instrucción (al igual que ocurre con JAL).
- NOP : No se modifica ningún registro ni flag. En realidad todos los códigos de operación no asignados se ejecutan como NOP, si bien el que se muestra en la tabla seguiría siendo NOP aunque se añadan más instrucciones en revisiones futuras del procesador.
- WFI : Wait for Interrupt. Detiene el procesador hasta que se reciba una petición de interrupción. El consumo de potencia del procesador también se reduce notablemente cuando está detenido.
- BRK : Breakpoint. Usado en emuladores para detener la ejecución. El core trata esta instrucción como un NOP.
- LDB Rd,(Ra+sdisp) : Load byte. Lee 8 bits de la memoria y los deja en la parte baja de Rd. Los 8 bits altos de Rd se hacen cero.
- LDBS Rd,(Ra+sdisp) : Load byte signed. Lee 8 bits de la memoria y los deja en la parte baja de Rd. Los 8 bits altos de Rd son una copia del bit #7 del byte, de modo que el dato de 16 bits que queda en Rd tiene el mismo signo que el byte que estaba en la memoria (extensión de signo).
- STB (Ra+sdisp),Rb : Store byte. Escribe los 8 bits bajos de Rb en la memoria.
- JNV : Jump relative if not overflow. Este salto condicional faltaba en las versiones anteriores del core.

En una segunda revisión se han añadido unas pocas instrucciones más con la intención de facilitar el acceso a los registros PC y Flags del modo normal (usuario) y con ello permitir la conmutación de tareas entre otras posibilidades. Estas instrucciones son:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	mnemonic	flags	operation
0	1	1	1	1		RD		1	1	0	—	—	—	0	0	RDPC	— — — —	RD = PC_normal
0	1	1	1	1		RD		1	1	0	—	—	—	0	1	RDFL	— — — —	RD = Flags_normal (VNCZ)
0	1	1	1	1	—	—	—	1	1	0		RB		1	0	WRPC	— — — —	NOP / PC_normal = RB
0	1	1	1	1	—	—	—	1	1	0		RB		1	1	WRFL	RB[3:0]	Flags_normal = RB
0	1	1	1	1		ulit3		1	1	1	—	1	1	0	1	SWI	— — — —	software interrupt

- **RDPC Rd** : Read PC. El valor del PC del modo normal se copia en Rd. Destacar que el PC del modo interrupción no se puede leer con esta instrucción.
- **RDFL Rd** : Read Flags. El registro de flags del modo normal, junto con el valor actual del modo (0: normal, 1: interrupción), se copia en los bits LSB del registro Rd de acuerdo con la siguiente tabla. Los flags del modo interrupción no se pueden leer con esta instrucción.

Rd[15:5]	Rd[4]	Rd[3]	Rd[2]	Rd[1]	Rd[0]
0	Mode	V	N	C	Z

- **WRPC Rb** : Write PC. En modo normal esta instrucción es un NOP, pero en el modo interrupción el valor de Rb se escribe en el registro PC del modo normal, de modo que se cambia la dirección de retorno de interrupción.
- **WRFL Rb** : Write Flags. Los 4 bits LSB del registro Rb se escriben en los flags del modo normal, tanto en el modo normal como en el de interrupción. El orden de los bits es:

V	N	C	Z
Rb[3]	Rb[2]	Rb[1]	Rb[0]

- **SWI nn** : Software Interrupt. Cambia al modo interrupción y salta a la rutina de interrupción asociada al vector indicado en su literal de 3 bits, “nn” (salta a VECTORBASE + nn * 4).

9.4 Interfaz del core

```

module CPU_GUSV8(
output [15:1]ca,      // Core Address
output [15:0]cdo,     // Core Data Output
output we,           // Write Enable (store)
output bhe,          // Bus High Enable
output ble,          // Bus Low Enable
input [15:0]cdi,     // Core Data Input
input clk,           // Clock
input cen,           // Clock enable
input reset,         // async. Reset (active high)
input irq,           // Interrupt Request
input [2:0]ivector,  // Interrupt vector
output reg mode,     // CPU mode 0: normal 1: Interrupt
output fetch,        // Valid opcode fetch if 1
output dload,        // Data load if 1
input brk            // Break interrupt if 1
);

```

- **'ca'**: Core Address . Falta ca[0] porque la memoria externa tiene 16 bits de ancho mientras que las direcciones son de byte.
- **'cdo' / 'cdi'**: Buses de datos (salida y entrada)
- **'we'**: Write enable. Activa en alto durante la ejecución de instrucciones 'Store'

- **'bhe' / 'ble'**: Byte high enable / byte low enable. Cuando alguna de estas señales está en alto y se está haciendo una escritura (we también alto) la mitad correspondiente de 'cdo' (cdo[15:8] / cdo[7:0]) se ha de escribir en la memoria. Estas señales también seleccionan el byte a leer, pero en este caso no son estrictamente necesarias ya que la mitad de 'cdi' no seleccionada se va a descartar de todos modos. Durante las lecturas y escrituras de datos de 16 bits 'bhe' y 'ble' están en alto de forma simultánea.
- **'clk'**: Core clock. Activo en los flancos de subida.
- **'cen'**: Clock enable. Entrada síncrona que inhibe el reloj si está en bajo cuando 'clk' sube (estado de espera).
- **'reset'**: Reset. Señal asíncrona que reinicia el core si se pone en alto (hace PC=0x0000, mode=normal, e invalida el último opcode que se haya cargado en IR)
- **'irq'**: Interrupt request. Petición de interrupción, activa en alto. Debe mantenerse durante al menos dos ciclos de reloj hasta que el core cambie al modo interrupción y comience a ejecutar la rutina de interrupción. Esta señal se ignora durante la ejecución de la rutina de interrupción salvo por su última instrucción, RETI, que encadenará otra interrupción si 'irq' está todavía en alto o retornará al modo normal en caso contrario.
- **'ivector'**: Entrada de 3 bits que identifica la fuente de la interrupción. Cuando 'irq' se activa el core cambia a modo interrupción y salta a la dirección 'VECTORBASE' + 'ivector' \times 4. 'VECTORBASE' es un parámetro del core que por defecto vale 0x0000. Notar que en este caso una interrupción con 'ivector'=0 salta a la misma dirección que un reset hardware.
- **'fetch'**: Salida que indica la lectura de un opcode válido, activa en alto. Esta señal se desactiva durante el ciclo de ejecución de las instrucciones de los tipos load, store, y saltos. Su principal uso es la validación del bus de direcciones en la implementación de breakpoints hardware.
- **'dload'**: Salida que se activa durante la ejecución de instrucciones del tipo 'Load'. Su principal uso está en la implementación de breakpoints hardware.
- **'brk'**: Breakpoint. Entrada que fuerza una IRQ #7 si se pone activa en alto. Está pensada para la implementación de breakpoints hardware y tiene algunas diferencias respecto de una interrupción 'irq' con 'ivector'=7:
 - El core pasa al modo interrupción en el siguiente ciclo de reloj mientras que para 'irq' tarda dos ciclos.
 - La instrucción actual (apuntada por PC) no se ejecuta, mientras que con 'irq' la instrucción actual completa su ejecución.
 - 'brk' se inhibe (no se muestrea) durante un ciclo después que el core sale del modo interrupción. Esto permite que se ejecute la instrucción que había quedado interrumpida tras salir de la rutina de interrupción.
 - 'brk' tiene implícito el vector #7 y su prioridad es la más alta de todas.