Descripción de la CPU BN16s1

Jesús Arias.

Dpto. E. y Electrónica. Univ. de Valladolid

Índice

1. Introducción

El procesador BN16s1 es un rediseño de la versión BN16 original que tiene como objetivos el simplificar la ALU a costa de intercambiar los operandos en las instrucciones LD y ST, y por otra parte tener una versión sintetizable en verilog que se pueda programar en una FPGA ICE40HX1K.

Notar que las herramientas de programación antiguas (ensamblador, simulador) no sirven para esta versión de la CPU pues las instrucciones LD y ST tienen una codificación distinta en la que la dirección va en el campo de registro Ra en lugar del Rb.

2. Arquitectura de la CPU

El procesador BN16 es de tipo Von-Newman a pesar de tratarse de un procesador RISC. Esto significa que tiene un único espacio de memoria que comparten instrucciones y datos. La principal motivación para adoptar esta arquitectura fue desde luego la reducción de la complejidad, pero luego se mostró como una decisión acertada: En un procesador RISC basado en un banco de registros los accesos a memoria de datos son poco frecuentes y por lo tanto no merece la pena añadir una memoria específica para los datos. Recordemos que en este diseño se persigue la simplicidad, no las máximas prestaciones posibles. El procesador

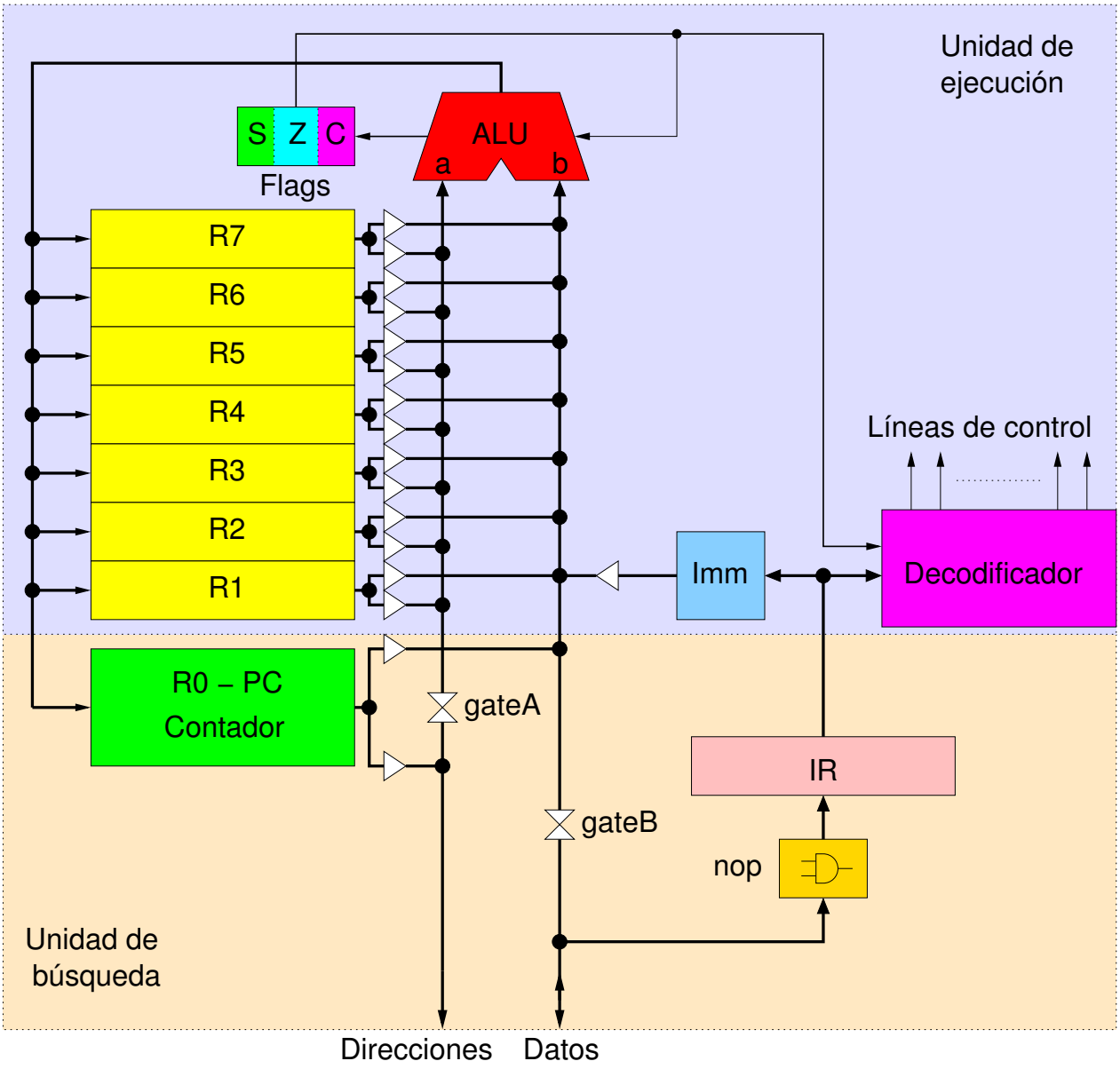

Figura 1: Esquema inicial de la CPU BN16s1.

carece de soporte para interrupciones (esta puede ser la próxima ampliación). El único tipo de dato es el entero de 16 bits y la memoria sólo se puede direccionar usando un registro de 16 bits como puntero.

En la figura 1 se muestra un esquema inicial de la CPU BN16s1. Consta principalmente de un banco de 8 registros de 16 bits en el que el registro R0 es el contador de programa, una ALU de 16 bits, un registro de flags de 3 bits, un registro de instrucción, también de 16 bits, y de la lógica de decodificación de instrucciones. Estos elementos se describirán con detalle más adelante. Antes vamos a presentar su reducido conjunto de instrucciones.

2.1. Conjunto de instrucciones

El procesador tiene 15 instrucciones cuya codificación se muestra en la figura 2. Las instrucciones aritméticas y lógicas típicas tienen dos registros fuente y un registro destino. En las instrucciones ADDQ y SUBQ el registro fuente Rb se ha sustituido por un operando inmediato de 3 bits. En las instrucciones NOT, NEG, y ROR no se utiliza el campo Ra. En la instrucción LD no se usa el campo Rb. En la instrucción ST no se utiliza el campo del registro destino, Rd. La instrucción JR lleva implícitos el registro destino (R0) y el fuente (también R0) y los 9 bits de los campos de registro se utilizan como un operando inmediato con signo que se va a sumar al valor de R0.

Todas las instrucciones tienen asociado un campo de código de condición de 3 bits. De las 8 posibles combinaciones las dos

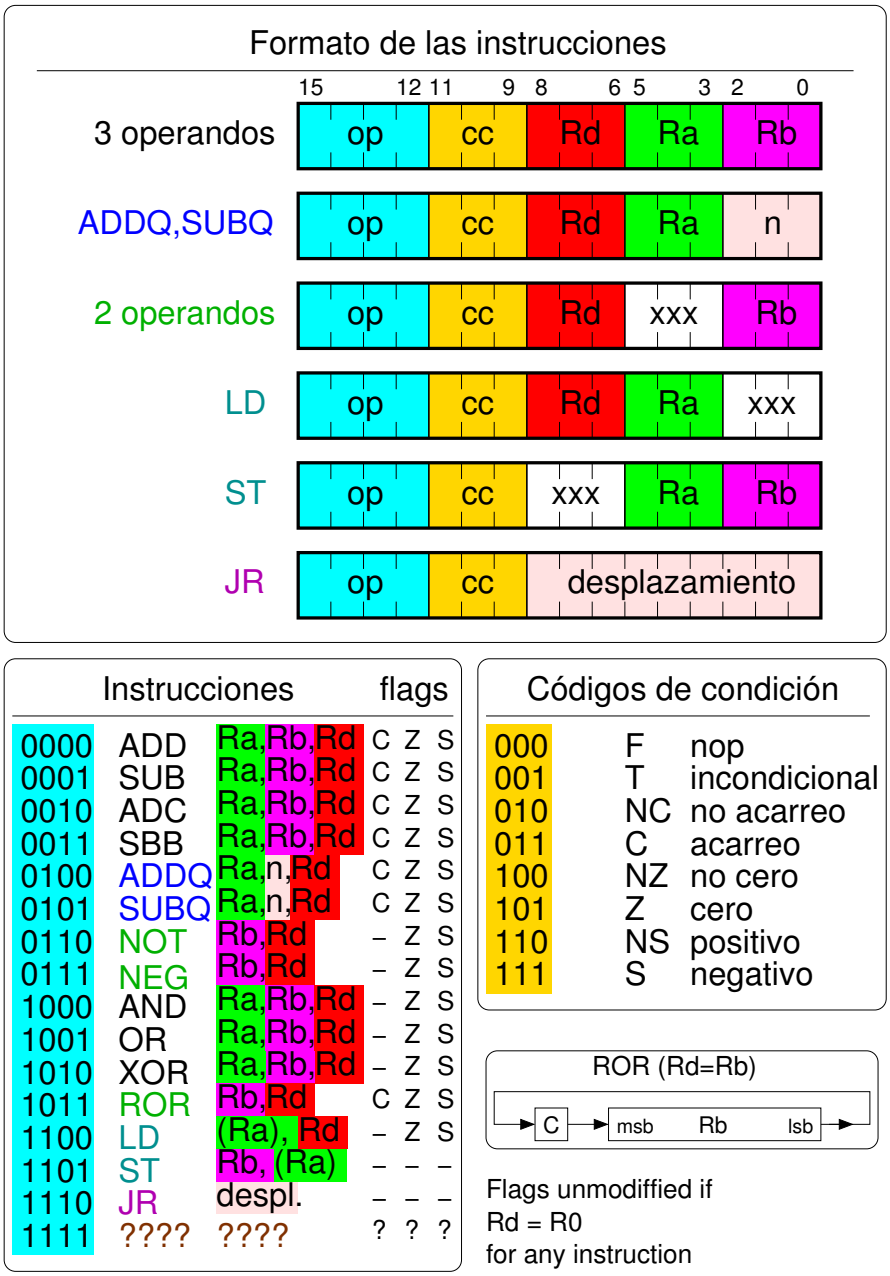

Figura 2: Codificación de las instrucciones en el procesador BN16s1.

primeras no dependen del valor de los flags. Si este campo es 000 la instrucción es NOP sea cual sea su código de operación. Si el campo de código de condición es 001 la instrucción se ejecutará siempre (instrucción no condicional). El resto de combinaciones del campo de código de condición da lugar a la ejecución condicional de la instrucción.

El código 1111 (0xf) aún no se ha asignado a ninguna instrucción. En la implementación actual esta instrucción puede provocar el bloqueo de la CPU lo que equivaldría a una instrucción HALT, si bien este ha sido un resultado no buscado durante el diseño.

Hay que destacar que en esta revisión del diseño del procesador BN16 se ha optado por mantener los flags sin modificar en los saltos. Esto es: en cualquier instrucción que tenga como destino el registro R0, no sólo en JR. Por ejemplo, la instrucción:

LD (Rx),R0

Es un salto pues modifica R0 y por consiguiente no altera ningún flag, mientras que

LD (Rx),R1

Es una instrucción LD corriente que activa el flag de cero (Z) si el registro R1 se carga con cero y el flag de signo (S) si el valor cargado en R1 es negativo.

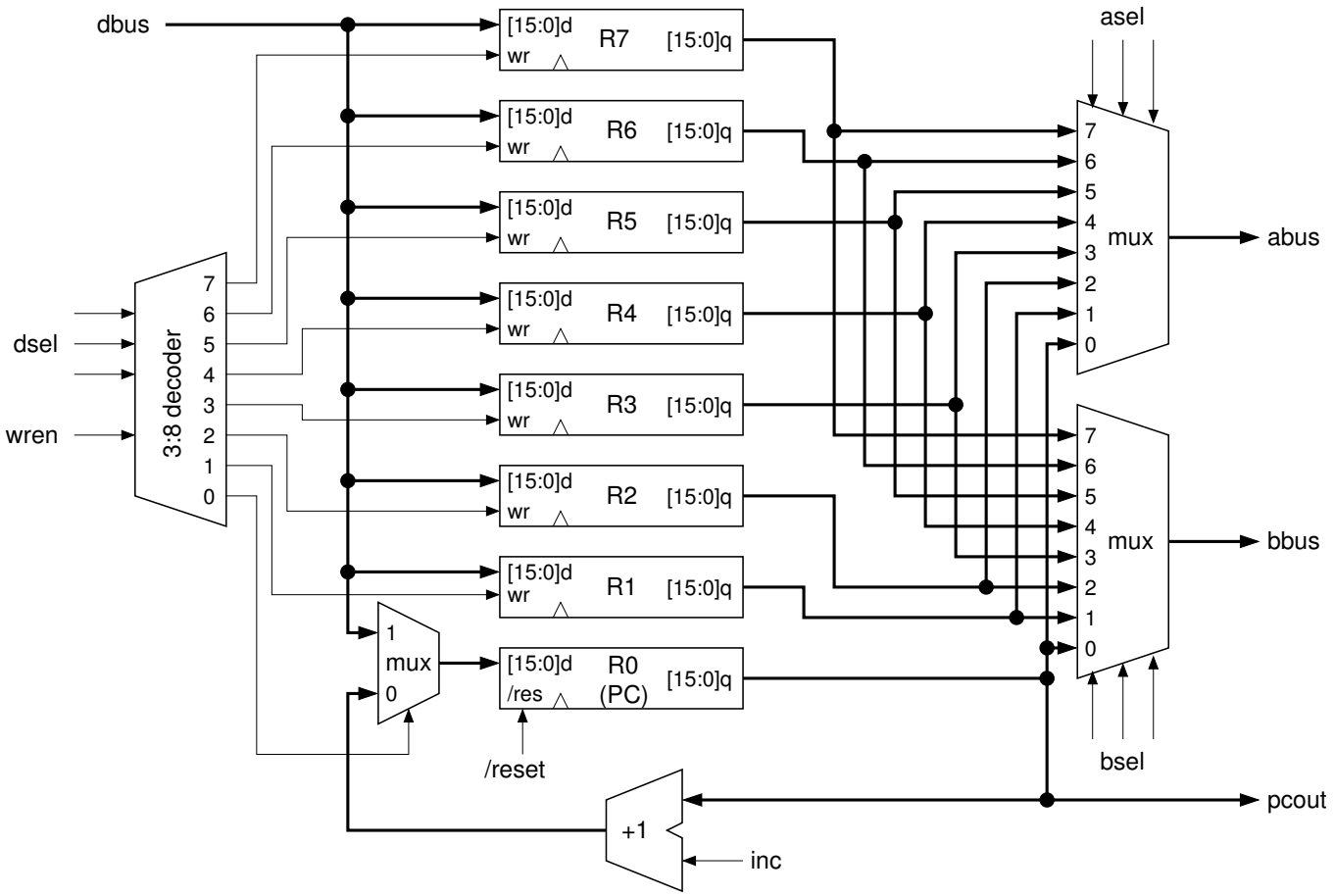

Figura 3: Esquema del banco de registros

2.2. Bloques funcionales

2.2.1. Registros

Los registros del banco de registros, R0 a R7, tienen dos salidas que pueden ponerse en alta impedancia. De este modo el contenido de un registro puede rutarse a la entrada “a” o “b” de la ALU según convenga. Los registros constan simplemente de 16 flip-flops conectados en paralelo con una señal de escritura común controlada por la lógica de decodificación y el reloj. El registro R0 es especial ya que se trata de un contador. R0 es el contador de programa y contiene la dirección de la siguiente instrucción que se va a ejecutar. Este registro se incrementa siempre que su contenido se usa para direccionar la memoria. Por lo demás, R0 es equivalente a cualquier otro registro desde el punto de vista de los programas y puede participar en las operaciones de la ALU como operando fuente o como destino, lo que confiere a esta CPU una funcionalidad mucho mayor que la que pudiera parecer a primera vista.

El registro IR contiene el código de operación de la instrucción actual. Cuando no es posible acceder a la memoria para leer un código de operación el registro IR se carga de forma automática con una instrucción NOP. Esto ocurre cuando se ejecutan las instrucciones “Load” y “Store”.

Hay tres registros de Flags: Acarreo, C, cero, Z, y signo, S. El valor de los flags se obtiene de las operaciones de la ALU. No todas las instrucciones alteran el valor de todos los flags. El valor de los flags interviene en la decodificación de las instrucciones: Si el valor del flag referenciado no coincide con el del campo de código de condición de la instrucción ésta no se ejecutará. Todas las instrucciones son por lo tanto de ejecución condicional, y eso incluye las que cambian el valor de R0 (saltos). Una modificación reciente del procesador BN16 impide que se alteren los flags cuando el registro destino de la instrucción es R0, independientemente de la instrucción ejecutada. El contenido de los flags no se puede transferir directamente desde o hacia otro registro.

El diagrama de bloques inicial del banco de registros ha tenido que ser modificado para adaptarlo a la filosofía de diseño de las FPGA en la que no se permiten las salidas triestado. La misma funcionalidad se consigue utilizando multiplexores, tal como

se muestra en la figura 3, donde las señales de control dsel y wren activan la escritura en el registro correspondiente del banco, mientras que las señales asel y bsel seleccionan qué registro se ruta a cada uno de los respectivos buses. El registro R0 sigue siendo especial pues se trata de un contador que se escribe en todos los ciclos de reloj, bien con un dato explícito procedente del bus D, o con su propio valor, incrementado o no dependiendo de la señal de control inc. El bloque incrementador se trata de un sumador pero simplificado, ya que basta con sumar 1 o 0, lo que reduce drásticamente su complejidad interna (Hay que señalar que la combinación registro+incrementador está implícita en cualquier contador).

El registro R0 también es distinto del resto por poseer una entrada de reset asíncrono que se va a utilizar para llevar el PC a un valor 0 cuando se active, de modo que la ejecución comenzará en la dirección de memoria 0. La entrada de reset asíncrono también se tiene en el registro IR, de modo que tras el reset la CPU ejecutará un NOP antes de cargar el primer código de operación de la memoria.

A continuación se detalla la implementación de estos bloques mediante Verilog, comenzando por los propios registros:

```
// Registro simple con habilitación de escritura para instanciar en el banco
module REG16 (output [15:0]q, input [15:0]d, input wr, input clk);
reg [15:0]q;
always @(posedge clk) q<= wr ? d : q;
endmodule
// Registro simple con reset para instanciar en el banco
module REG16R (output [15:0]q, input [15:0]d, input resb, input clk);
reg [15:0]q;
always @(posedge clk or negedge resb )
    if (!resb) q<=16'h0000;
    else q<=d;
endmodule
```

Y ahora el banco completo:

```
module REGBANK (
output [15:0]a,   // Salida para bus A (ALU, dir memoria)
output [15:0]b,   // Salida para bus B (ALU, datos memoria)
output [15:0]pco, // Salida del contador de programa (R0)
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 inc,        // Incrementar PC si 1
input clk,        // Reloj
input resb        // Reset de PC
);
// Decodificador 3:8
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]);
// Registro R0 (Contador de Programa)
wire [15:0]pci;
assign pci = wr[0] ? d : (pco+inc) ;
```

```
REG16R r0(.q(pco),.d(pci),.resb(resb),.clk(clk));
// Registros R1-R7
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;
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 <= pco;
    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 <= pco;
     1 : b <= q1;
     2 : b <= q2;
     3 : b <= q3;
     4 : b <= q4;
     5 : b <= q5;
     6 : b <= q6;
     7 : b <= q7;
  endcase
endmodule
```

El registro de instrucción tiene como aspecto destacado el poder hacer cero los 3 bits correspondientes al código de condición de la instrucción cargada, lo que la convierte en un NOP. Esto se hará durante la fase de ejecución de las instrucciones LD y ST pues en esos ciclos no se puede leer un código de operación desde la memoria. La descripción de este registro es:

```
//-------------------------------------------------
// Registro de instrucción
//-------------------------------------------------
module IR (
output [15:0]q, // Salida (hacia operandos immediatos y decodificación)
input [15:0]d,  // Entrada (desde bus de datos)
input fnop,     // fetch NOP si 1 (para instrucciones LD y ST)
```

```
input clk,
input resb
);
reg [15:0]q=0;
always @(posedge clk or negedge resb)
    if (!resb) q<=16'h0000;
    else       q<= {d[15:12],(fnop ? 3'b000 : d[11:9]), d[8:0]};
endmodule
```

Por último hay que mencionar el registro de Flags que contiene el flag de acarreo, C, proveniente de la ALU, y los flags de cero, Z, y signo, S, que se computan a partir del dato presente en el bus D (salida de la ALU). El flag Z se pone en 1 cuando todos los bits del bus valen cero, y el flag de signo es directamente una copia del bit más significativo del bus.

El flag de acarreo tiene una señal de escritura separada del resto pues son menos las instrucciones que lo modifican respecto de los flags Z y S, que tienen una señal de escritura común:

```
//---------------------------
// FLAGS
//---------------------------
module FLAGS (
output qc,     // Salida de flag C
output qz,     // Salida de flag Z
output qs,     // Salida de flag S
input [15:0]f, // Entrada de resultado de ALU
input ci,      // Entrada de flag C
input wrzs,    // Escribir flags Z y S si 1
input wrc,     // Escribir flag C si 1
input clk
);
reg qc,qz,qs;
// Flag Z
wire zi;
assign zi=~(f[15]|f[14]|f[13]|f[12]|f[11]|f[10]|f[9]|f[8]|
            f[7]|f[6]|f[5]|f[4]|f[3]|f[2]|f[1]|f[0] );
always @(posedge clk) begin
    if (wrc) qc<=ci;
    if (wrzs) begin
        qz<=zi;
        qs<=f[15];
    end
end
endmodule
```

2.2.2. ALU

La unidad aritmética y lógica tiene dos entradas de datos, “a” y “b”, provenientes principalmente del banco de registros. El operando “a” se puede hacer cero y el operando “b” puede invertir sus bits de acuerdo con las respectivas líneas de control provenientes de la lógica de decodificación. Con los operandos “a” y “b” ya preprocesados se pueden realizar 5 operaciones:

Operación de la ALU
“a” más “b”
“a” AND “b”
“a” OR “b”
“a” XOR “b”
rotación a la derecha de “b”

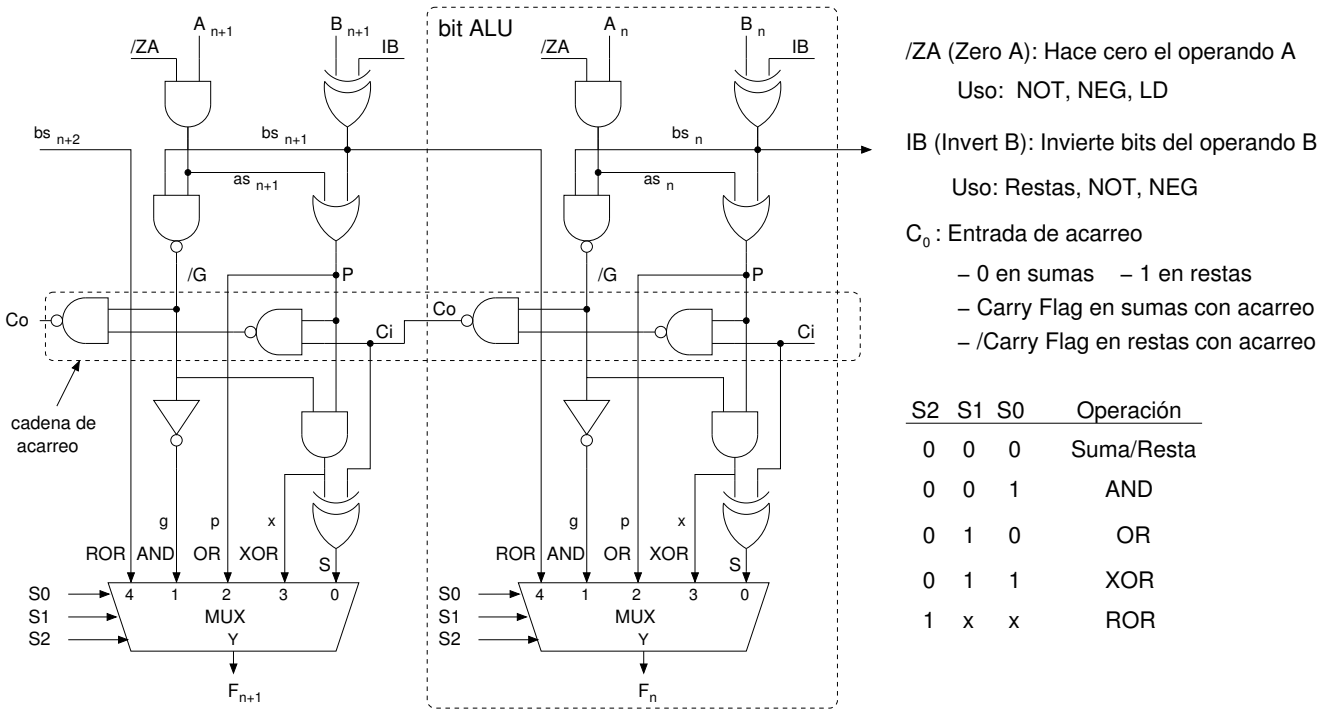

Figura 4: Esquema de un bit de la ALU y si interconexión con el bit adyacente

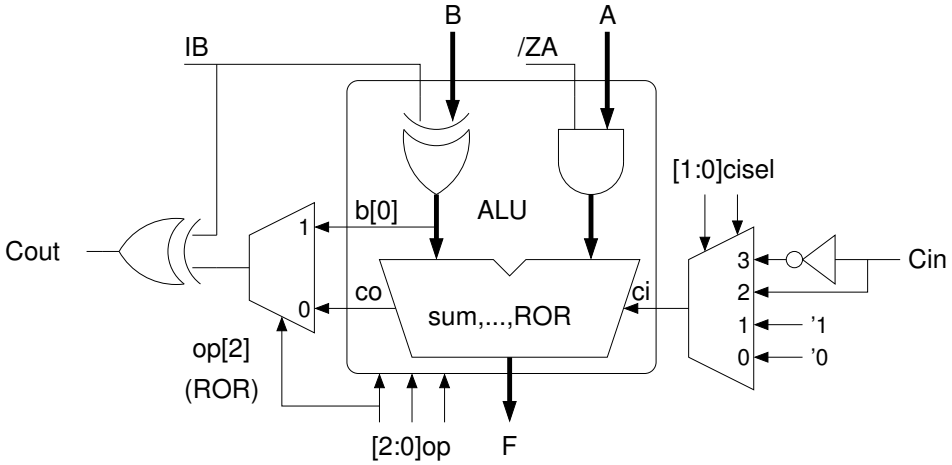

Figura 5: Lógica de acarreo en la ALU

La ALU tiene una entrada de acarreo que puede hacerse 0 o 1 o puede provenir del flag de acarreo directamente o tras un complemento. El control de la entrada de acarreo, junto con la inversión de los bits de “b”, permite la realización de restas y de operaciones de más de 16 bits. La rotación a la derecha de “b” se emplea en la instrucción ROR. El operando “a” se hace cero en las instrucciones NOT y NEG.

En esta revisión se ha sustituido la ALU original por una versión simplificada. Si bien la ALU es un bloque combinacional que se sintetizará mediante una descripción funcional de hardware en Verilog, el tipo de circuito que aspiramos obtener como resultado de la síntesis es el de la figura 4. Observar que ya no es posible obtener en la salida F una copia de A, pero esto ya no es necesario pues la instrucción LD ahora pone los datos en la entrada B de la ALU. Asimismo, la lógica es más simple al recurrir a al estrategia de acarreo generado (G) y propagado (P), lo que reduce el tamaño de las puertas lógicas a sólo dos entradas.

La lógica del acarreo se muestra en la figura 5, donde podemos observar que la entrada de acarreo al sumador de la ALU se puede elegir entre cuatro posibles, que incluyen el valor 0 para las instrucciones de suma, 1 para las instrucciones de resta, el nivel del flag de acarreo para la instrucción ADC, y el complemento del flag de acarreo en la instrucción SBB. Observar también que el acarreo de salida se invierte en las restas (señal IB en alto), de modo que el flag de acarreo se hará 1 cuando tengamos una llevada en una resta, o lo que es lo mismo: el flag hace una función de “borrow” en las restas, al igual que ocurre en los

procesadores de Intel o Zilog, pero en contraposición en lo que ocurre en micros como los ARM o 6502 en los que las llevadas dejan el flag de acarreo en cero. Finalmente hay que destacar que en la instrucción ROR la salida de acarreo se obtiene del bit menos significativo del operando B, y no de la cadena de acarreo del sumador.

A continuación se lista el código Verilog de la ALU descrita:

```
module ALU (
  output [15:0]f,   // Salida de resultado
  output co,        // Salida de acarreo
  input [15:0]a,    // Entrada de operando A
  input [15:0]b,    // Entrada de operando B
  input ci,         // Entrada de acarreo
  input [2:0]op,    // Operación (0: suma, 1: AND, 2: OR, 3: XOR, 4: ROR)
  input zab,        // Forzar operando A=0 si 0
  input ib,         // Invertir bits de operando B si 1
  input [1:0]cisel  // Selección de acarreo de entrada: (0:L, 1:H, 2:ci, 3:~ci)
);
wire [15:0]sa;
wire [15:0]sb;
reg [15:0]f;     // No realmente registros
reg sco;
// Multiplexor de acarreo de entrada
reg sci;
always @*
  case (cisel)
  0 : sci = 0;
  1 : sci = 1;
  2 : sci = ci;
  3 : sci = ~ci;
  endcase
// Datos intermedios
assign sa= zab ? a  : 16'b0;
assign sb= ib  ? ~b : b;
// Operación
always @*
  case (op)
    0 : {sco,f} = sa+sb+sci;            // SUMA
    1 : {sco,f} = {1'bx,sa & sb};       // AND
    2 : {sco,f} = {1'bx,sa | sb};       // OR
    3 : {sco,f} = {1'bx,sa ^ sb};       // XOR
    4 : {sco,f} = {sb[0],ci,sb[15:1]};  // ROR
    default : {sco,f} ={1'bx,16'bxxxxxxxxxxxxxxxx};
endcase
assign co = ib ? ~sco : sco;
endmodule
```

2.2.3. Lógica de decodificación

La lógica de decodificación es un circuito puramente combinacional que activa las líneas de control pertinentes dependiendo del código de operación almacenado en el registro IR y del valor de los flags. Todas las instrucciones son condicionales de modo que si los flags no tienen el valor adecuado la instrucción se convierte en NOP. Esto se consigue al inhibir las señales de escritura en los registros (flags inclusive) y de escritura en memoria. Internamente se genera una señal auxiliar, EXE, con un valor 1 si la instrucción tiene permiso para ejecutarse. Esta señal se obtiene mediante un multiplexor:

```
reg exe; // Ejecutar / No ejecutar (no es un FF)
```

```
// multiplexor de 8 a 1 para exe
always@*
  case (op[11:9])
    0 : exe <= 0;
    1 : exe <= 1;
    2 : exe <= ~cf;
    3 : exe <= cf;
    4 : exe <= ~zf;
    5 : exe <= zf;
    6 : exe <= ~sf;
    7 : exe <= sf;
endcase
```

También se generan algunas señales dependientes de ciertos códigos de operación:

```
// Ciertos OP-codes y líneas de control
assign ld = exe & op[15] & op[14] &(~op[13]) &(~op[12]);
assign st = exe & op[15] & op[14] &(~op[13]) & op[12];
assign jr = exe & op[15] & op[14] & op[13] &(~op[12]);
assign imm = jr | ( (~op[15]) & op[14] &(~op[13]));
assign wren = exe & (~st);
assign incpc = (~(ld |st)) | (~(sela[2]|sela[1]|sela[0]));
```

En concreto, las señales ld, st, y jr se activan con con sus correspondientes códigos de operación. La señal imm indica un operando inmediato (instrucciones ADDQ, SUBQ, y JR), wren indica una escritura en el banco de registros, e incpc un incremento del contador de programa.

La selección de los registros se hace directamente con sus respectivos campos del código de operación, pero hay una excepción con la instrucción JR. En este caso el registro fuente y destino es R0 independientemente del valor de los bits de los campos de registro:

```
// Selección de registros
assign sela = jr ? 3'b000 : op[5:3];
assign selb = op[2:0];
assign seld = jr ? 3'b000 : op[8:6];
```

Por último se incluye una ROM/PLA con las señales activas por cada código de operación, que incluyen dos señales de escritura en los flags, una para el flag C, y otra para los flags Z y S, que se validan con la señal EXE:

```
reg [2:0]aluop;
reg aluzab, aluib;
reg [1:0] cisel;
reg twrc, twrzs;
always@*
 case (op[15:12])
  0: {aluop,aluzab,aluib,cisel,twrc,twrzs}<={3'b000, 1'b1 ,1'b0 ,2'b00, 1'b1, 1'b1}; // ADD
  1: {aluop,aluzab,aluib,cisel,twrc,twrzs}<={3'b000, 1'b1 ,1'b1 ,2'b01, 1'b1, 1'b1}; // SUB
  2: {aluop,aluzab,aluib,cisel,twrc,twrzs}<={3'b000, 1'b1, 1'b0, 2'b10, 1'b1, 1'b1}; // ADC
  3: {aluop,aluzab,aluib,cisel,twrc,twrzs}<={3'b000, 1'b1, 1'b1, 2'b11, 1'b1, 1'b1}; // SBB
  4: {aluop,aluzab,aluib,cisel,twrc,twrzs}<={3'b000, 1'b1, 1'b0, 2'b00, 1'b1, 1'b1}; // ADDQ
  5: {aluop,aluzab,aluib,cisel,twrc,twrzs}<={3'b000, 1'b1, 1'b1, 2'b01, 1'b1, 1'b1}; // SUBQ
  6: {aluop,aluzab,aluib,cisel,twrc,twrzs}<={3'b010, 1'b0, 1'b1, 2'bxx, 1'b1, 1'b1}; // NOT
  7: {aluop,aluzab,aluib,cisel,twrc,twrzs}<={3'b000, 1'b0, 1'b1, 2'b01, 1'b1, 1'b1}; // NEG
  8: {aluop,aluzab,aluib,cisel,twrc,twrzs}<={3'b001, 1'b1, 1'b0, 2'bxx, 1'b0, 1'b1}; // AND
  9: {aluop,aluzab,aluib,cisel,twrc,twrzs}<={3'b010, 1'b1, 1'b0, 2'bxx, 1'b0, 1'b1}; // OR
```

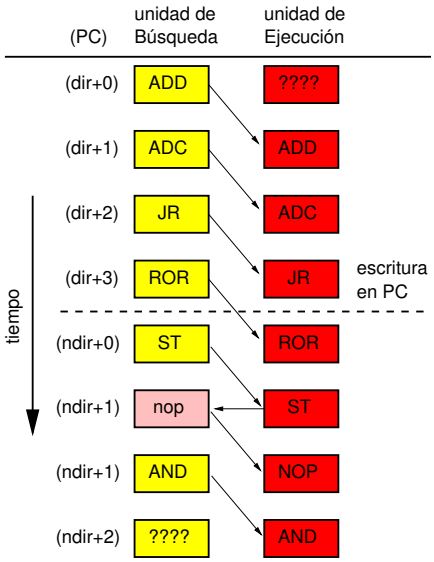

Figura 6: Ejemplo de ejecución de programa en la “pipeline” de 2 etapas del procesador BN16

```
 10: {aluop,aluzab,aluib,cisel,twrc,twrzs}<={3'b011, 1'b1, 1'b0, 2'bxx, 1'b0, 1'b1}; // XOR
 11: {aluop,aluzab,aluib,cisel,twrc,twrzs}<={3'b100, 1'b1, 1'b0, 2'b10, 1'b1, 1'b1}; // ROR
 12: {aluop,aluzab,aluib,cisel,twrc,twrzs}<={3'b010, 1'b0, 1'b0, 2'bxx, 1'b0, 1'b1}; // LD
 13: {aluop,aluzab,aluib,cisel,twrc,twrzs}<={3'bxxx, 1'bx, 1'bx, 2'bxx, 1'b0, 1'b0}; // ST
 14: {aluop,aluzab,aluib,cisel,twrc,twrzs}<={3'b000, 1'b1, 1'b0, 2'b00, 1'b0, 1'b0}; // JR
 default: {aluop,aluzab,aluib,cisel,twrc,twrzs} <= {3'bxxx,1'bx,1'bx,2'bxx,1'bx,1'bx};
endcase
// Escritura en flags
assign wrc  = exe & twrc  & (seld[2]|seld[1]|seld[0]);
assign wrzs = exe & twrzs & (seld[2]|seld[1]|seld[0]);
```

En algunas instrucciones (ADDQ, SUBQ y JR) hay datos inmediatos incluidos en el código de operación de la instrucción. En estos casos se extiende el número de bits del dato a 16 bits y se conecta a la entrada “b” de la ALU. La operación de extensión se realiza en el bloque IMM de la figura 1. En las instrucciones ADDQ y SUBQ los 13 bits más significativos se ponen a 0, mientras que en la instrucción JR se copia el noveno bit en todos los restantes bits más significativos (extensión de signo). De este modo los saltos relativos pueden tener desplazamientos tanto positivos como negativos.

```
//-------------------------------------------------------------------------------
//Operandos inmediatos
//-------------------------------------------------------------------------------
module IMM(
output [15:0]f, // Salida de operando inmediato
input [15:0]op, // Entrada desde reg. de instrucción
input jr        // Tipo de dato (0: 3bit + zero fill, 1: 9bit + extensión signo)
);
assign f= jr ? {op[8],op[8],op[8],op[8],op[8],op[8],op[8], op[8:0]} // extensión de signo
             : {13'b0000000000000,op[2:0]};                         // Zero fill
endmodule
```

2.3. Pipelining

El procesador BN16 está dividido en dos unidades que operan de forma independiente en la mayoría de los casos. Estas son la unidad de búsqueda (fetch) y la unidad de ejecución (execute). La unidad de búsqueda incluye el contador de programa, el registro IR y la memoria externa. La unidad de ejecución incluye la lógica de decodificación, la ALU y el banco de registros. Estas dos unidades funcionan siguiendo la estrategia de “pipelining”: En el mismo ciclo de reloj la unidad de ejecución realiza

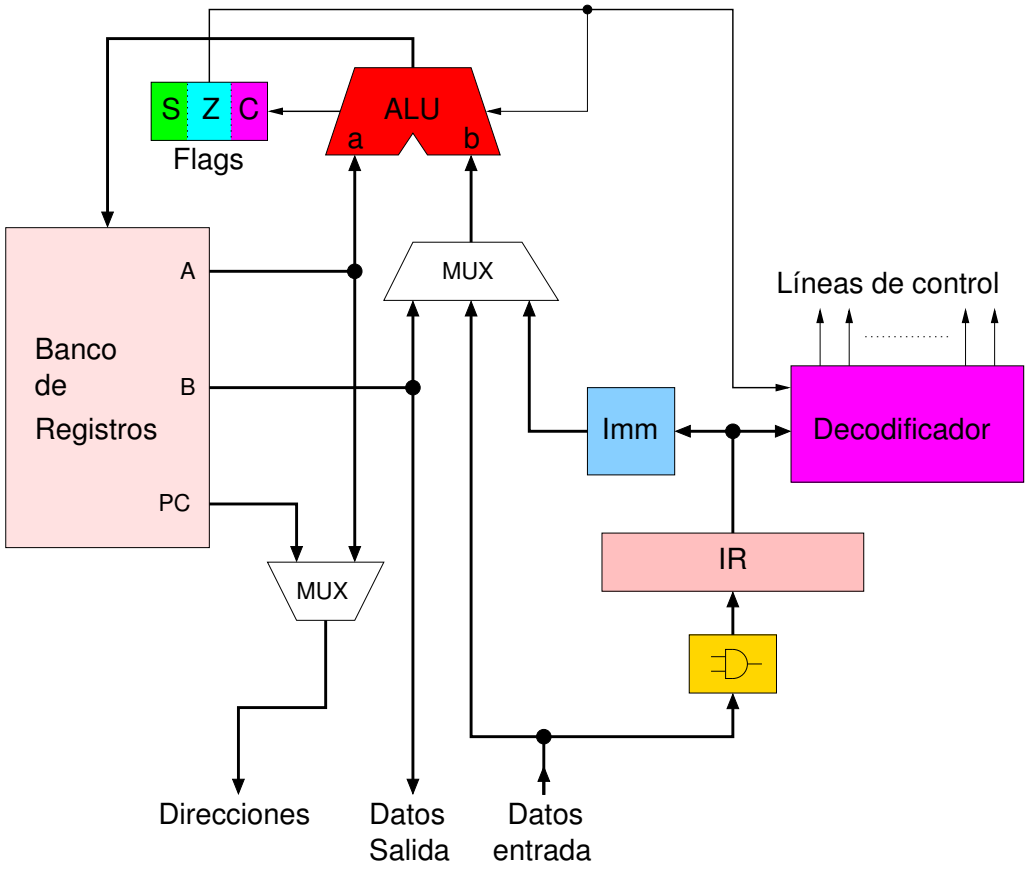

Figura 7: Diagrama de bloques de la implementación FPGA

las operaciones relacionadas con la instrucción actual, y simultáneamente la unidad de búsqueda lee de la memoria el código de operación de la siguiente instrucción. Las unidades de búsqueda y ejecución no son completamente independientes: El PC puede ser uno de los operandos de la ALU y las instrucciones LD y ST necesitan un acceso a la memoria independiente del PC. En la figura 1 las puertas de transmisión gateA y gateB mantienen las unidades aisladas salvo cuando es preciso su interconexión.

En la figura 6 se muestra un ejemplo de ejecución de una sección de código que ilustra algunas de las particularidades del "pipelining" del procesador BN16. Al principio vemos como cada instrucción tarda en ejecutarse 2 ciclos de reloj, aunque se termina una instrucción cada ciclo gracias al "pipelining".

Un caso interesante se tiene cuando una instrucción altera el valor del PC (salto). En este ejemplo, cuando la instrucción JR se ejecuta ya se ha cargado la siguiente instrucción (ROR) en IR y este código no se descarta. Por lo tanto todas las instrucciones que sigan a un salto se van a ejecutar y es responsabilidad del programador situar una instrucción NOP tras un salto si esto fuese necesario (Hay una excepción a esta regla cuando el salto se realiza mediante "LD (Rx),PC").

El otro caso especial es el de la ejecución de las instrucciones LD y ST. Estas instrucciones suponen un acceso a memoria y por lo tanto no se puede leer un código de operación simultáneamente. Cuando se ejecuta una instrucción LD o ST se carga una instrucción NOP en el registro IR y el registro PC no se incrementa (salvo que se use como puntero en la instrucción). Las instrucciones LD y ST tienen por lo tanto un tiempo de ejecución efectivo de 2 ciclos de reloj.

2.4. F PeGAs

Esta revisión del diseño se ha llevado a cabo con la intención de sintetizar el procesador en una FPGA, y ello nos obliga a tener en cuenta las limitaciones impuestas por este tipo de lógica. En general, las FPGAs están reñidas con las puertas triestado y los buses bidireccionales, por no hablar de las puertas de transmisión que son poco menos que anatema. Siendo justos hay que admitir que los triestados sí que se soportan, pero normalmente sólo en los pines que comunican la FPGA con el exterior. Esto nos obliga a rediseñar nuestro diagrama de bloques usando multiplexores en lugar de triestados y dos buses de datos, uno de entrada y otro de salida, tal como se muestra en la figura 7. Los dos buses de datos no son un problema para la interconexión con los bloques de memoria interna de la FPGA, que asimismo tienen dos buses. Y para la interconexión con memorias externas se

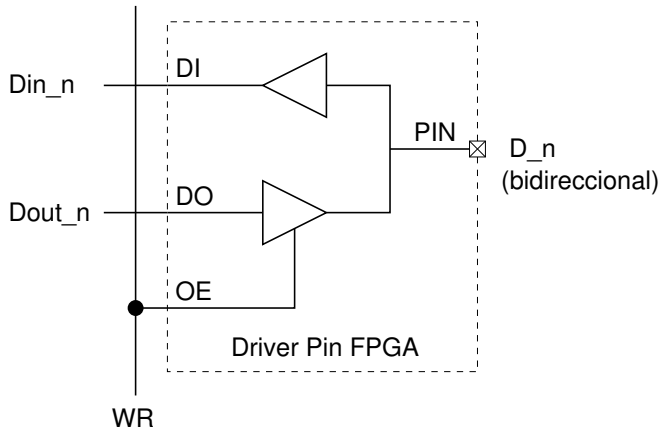

Figura 8: Cómo obtener un bus de datos externo bidireccional

puede hacer uso de los drivers triestado de los pines de la FPGA para reconducir los dos buses de datos internos a un solo bus de datos bidireccional externo, tal como se muestra en la figura 8.

3. Sugerencias para programación

El conjunto de instrucciones del procesador BN16 puede parecer demasiado reducido como para resultar práctico. Veremos sin embargo que estas instrucciones adecuadamente combinadas cubren las necesidades habituales de los programas.

3.1. Rotaciones y desplazamientos

La instrucción ROR realiza una rotación a derechas a través del acarreo. Si se desea realizar un desplazamiento en lugar de una rotación habrá que poner a cero antes el flag de acarreo. Esto puede conseguirse sumando 0 a un registro:

```
ADDQ R7,0,R7  ; Borra el acarreo
ROR  R7,R7    ; Desplazamiento a derechas
```

Si se desea dividir por 2 conservando el signo hay que copiar el bit 15 en el acarreo antes de ejecutar la instrucción ROR:

```
ADD  R7,R7,R6 ; Bit 15 de R7 en el acarreo. R6 no se usa
ROR  R7,R7    ; División por 2 con signo
```

Los desplazamientos a la izquierda equivalen a multiplicar el dato por 2, o a lo que es lo mismo: a sumarlo consigo mismo:

```
ADD R7,R7,R7  ; Se desplaza R7 un bit a la izquierda
```

Si la suma incluye el acarreo tendremos una rotación:

```
ADC R7,R7,R7  ; Se rota R7 un bit a la izquierda
```

3.2. Carga de constantes

La forma más conveniente de cargar una constante en un registro es utilizando el registro PC como puntero:

```
LD (PC),R7    ; Sigue el dato a cargar en R7
word 1234     ; dato (1234)
...
```

De esta forma el dato de la posición de memoria que sigue a la instrucción LD se carga en el registro R1. El contador de programa no se incrementa cuando se ejecutan las instrucciones LD o ST, salvo cuando se usa el propio PC como puntero, cosa que ocurre en este caso. Por lo tanto la posición de memoria ocupada por la constante se salta durante la ejecución del programa.

Nótese que la instrucción LD no debe ser condicional, pues en ese caso si no se cumple la condición se va a intentar ejecutar la constante como un código de operación válido.

Para cargar un registro con cero basta con la instrucción XOR o SUB:

```
XOR R7,R7,R7  ; Pone a cero R7
SUB R7,R7,R7  ; Pone a cero R7 y también el flag C
```

Si la constante difiere en un valor menor que 7 del contenido conocido de un registro pueden usarse ADDQ o SUBQ:

```
; R7 se ha cargado previamente con el valor 3
ADDQ R7,5,R6  ; R6=R7+5=8
```

3.3. Saltos

La instrucción JR permite realizar un salto a una posición de programa que no diste más de 256 posiciones de la actual. Esta es una instrucción muy conveniente para ejecutar la mayoría de los saltos, pero puede que no sea siempre suficiente. Recordemos que cualquier instrucción que tenga como destino el registro R0 va a ser un salto. Así, si queremos realizar un salto a una posición absoluta de la memoria lo haríamos de la siguiente manera:

```
LD (PC),PC    ; El PC se carga con el dato de la siguiente dir. de mem.
word 1024     ; Dirección a la que se salta (1024)
```

El siguiente es un ejemplo de bifurcación múltiple mediante una tabla de saltos:

```
; R7 tiene la entrada de la tabla a usar
ADDQ R7,1,R7    ; PC apunta una posición antes de la tabla
ADD  PC,R7,R7   ; Sumamos offset
LD   (R7),PC    ; cargamos PC con lo que leemos de la tabla
word CASE0_addr
word CASE1_addr
word CASE2_addr
...
```

3.4. Pilas

Aunque no se disponga de un registro puntero de pila resulta bastante fácil realizar su función mediante programa. En los siguientes ejemplos el registro R1 se usa como puntero de pila. La pila es de tipo “Full-descending”. Esto es: crece hacia abajo y el puntero de pila, R1, apunta a un dato válido, no a una posición de memoria libre.

Introducción de datos (R7) en la pila (PUSH):

```
SUBQ R1,1,R1  ; Primero decrementamos el puntero de pila
ST R7,(R1)    ; y luego se guarda el dato
```

Para recuperar los datos de la pila (POP) se sigue el proceso inverso:

```
LD (R1),R7    ; Recuperamos el dato
ADDQ R1,1,R1  ; e incrementamos el puntero de pila
```

3.5. Subrutinas

La llamada a una subrutina supone en primer lugar guardar la dirección de retorno y en segundo lugar realizar un salto. En el siguiente ejemplo se utiliza el registro R2 para guardar la dirección de retorno, de manera similar a la función del registro LR en procesadores ARM:

```
ADDQ PC,2,R2  ; Cálculo de la dirección de retorno
JR RUTINA     ; (pc)   Salto a la subrutina
NOP           ; (pc+1) Se ejecuta por estar en el pipeline
...           ; (pc+2) Aquí continúa la ejecución al retornar
```

Destacar que la instrucción NOP podría sustituirse por otra instrucción útil y se ejecutaría antes del código de la subrutina.

Si la rutina se encuentra en una dirección de memoria distante emplearemos LD en lugar de JR:

```
ADDQ PC,2,R2  ; Cálculo de la dirección de retorno
LD (PC),PC    ; (pc)   Salto a la subrutina
WORD RUTINA   ; (pc+1)
...           ; (pc+2) Aquí continúa la ejecución al retornar
```

La subrutina no debe alterar el valor de R2 que contiene la dirección de retorno. Si ello fuese necesario, como por ejemplo para llamar a otra subrutina, se deberá guardar el valor de R2 en una pila y luego recuperarlo antes de retornar de la subrutina. La subrutina concluye por ejemplo con:

```
OR R2,R2,PC   ; Se carga el PC con la dirección de retorno
NOP           ; Se ejecuta por estar en el pipeline
```

De nuevo, la instrucción NOP podría sustituirse por una instrucción útil que se ejecutaría antes de retornar.

El siguiente es un ejemplo de subrutinas anidadas. R1 se usa como puntero de pila y R2 como almacenamiento de la dirección de retorno:

```
; Programa principal
    ADDQ PC,2,R2  ; Cálculo de la dirección de retorno
    JR RUTINA1    ; Salto a la subrutina
    NOP           ; Se ejecuta por estar en el pipeline
    ...           ; Aquí continúa la ejecución al retornar
; Subrutina de primer nivel
RUTINA1:
    SUBQ R1,1,R1  ; Guardamos R2 en la pila
    ST R2,(R1)
    ...
    ADDQ PC,2,R2  ; Cálculo de la dirección de retorno
    JR RUTINA2    ; Salto a la subrutina
    NOP           ; Se ejecuta por estar en el pipeline
    ...
    LD (R1),R2    ; Recuperamos R2 de la pila
    OR R2,R2,PC   ; Retornamos
    ADDQ R1,1,R1  ; Se ejecuta por estar en el pipeline
; Subrutina de segundo nivel. No llama otras subrutinas ni cambia R2
RUTINA2:
    ...
    ADDQ R2,0,PC  ; Retornamos (además ponemos C=0)
    NOP           ; Se ejecuta por estar en el pipeline
```

3.6. Overflow

El procesador BN16 no tiene un flag de overflow, lo que supone una limitación a la hora de comprobar el rango de los resultados aritméticos con signo. De todos modos sabemos que un overflow se produce cuando al sumar dos datos con el mismo signo obtenemos un resultado de signo contrario:

```
; Sumamos R6 + R7 -> R5
ADD R7,R6,R5    ; suma
NOT R6,R4
XOR R7,R4,R4    ; R7 y R6 mismo signo => MSB=1
XOR R7,R5,R3    ; resultado signo contrario => MSB=1
AND R4,R3,R4    ; Si el flag S está activo hubo un overflow
```

En las restas cambia el signo del sustraendo:

```
; Restamos R7 - R6 -> R5
SUB R7,R6,R5    ; resta
XOR R7,R6,R4    ; R6 distinto signo que R7 => MSB=1
XOR R7,R5,R3    ; resultado signo contrario => MSB=1
AND R4,R3,R4    ; Si el flag S está activo hubo un overflow
```

Como vemos la comprobación del overflow aritmético es posible aunque costoso. Afortunadamente esta no es una condición que necesite comprobarse con mucha frecuencia.