Descripción de la CPU BN16

Jesús Arias.

Dpto. E. y Electrónica. Univ. de Valladolid

Índice

1. Introducción **1**

2. Arquitectura de la CPU **1**
- 2.1. Registros . . . 2
- 2.2. ALU . . . 3
- 2.3. Lógica de decodificación . . . 3
- 2.4. Pipelining . . . 3

3. Conjunto de instrucciones **4**
- 3.1. Estructuras de programa complejas . . . 4

4. Implementación CMOS **7**
- 4.1. Registros . . . 8
- 4.2. ALU . . . 10
- 4.3. Flags . . . 12
- 4.4. Operandos inmediatos . . . 13
- 4.5. Decodificación de las instrucciones . . . 14

5. Simulación **16**

6. Versión 2: BN16/V2 **16**

7. Comentarios finales **19**

1. Introducción

El procesador BN16 (Bestia número 16 :) es un diseño de una CPU RISC minimalista. Este proyecto comenzó como un ejemplo de diseño digital para la asignatura optativa de Diseño Microelectrónico en Ingeniería Informática hace algunos años. Posteriormente, tras varios años de investigación dedicados al diseño de circuitos integrados analógicos, he decidido quitarle las telarañas a esta CPU, hacer una implementación de la misma en una tecnología CMOS de $0.35\mu m$ y redactar esta documentación. El procesador BN16 tiene un ancho de palabra de 16 bits y no aspira a competir con otras CPUs comerciales o de cualquier tipo. Es sólo un ejemplo de simplicidad en el diseño de este tipo de circuitos.

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

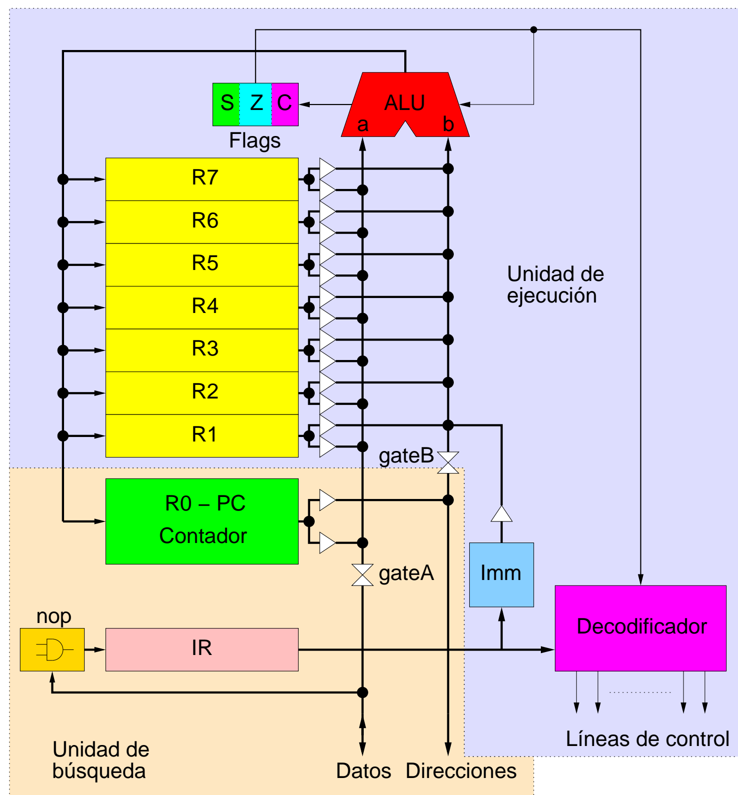


Figura 1: Esquema de la CPU BN16.

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 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 de la CPU BN16. 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. A continuación se describen estos elementos con mayor detalle.

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 ve 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.

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 6 operaciones:

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

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. Con el paso del operando “a” directamente a la salida se da soporte a la instrucción LD (load). 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.

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.

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.

2.4. 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 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.

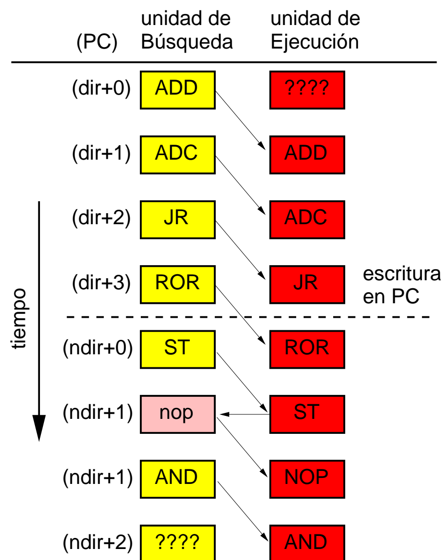


Figura 2: Ejemplo de ejecución de programa en la "pipeline" de 2 etapas del procesador BN16

En la figura 2 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. Es responsabilidad del programador situar una instrucción NOP tras un salto si esto fuese necesario.

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.

3. Conjunto de instrucciones

El procesador tiene 15 instrucciones cuya codificación se muestra en la figura 3. 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, ROR y LD no se utiliza el campo Ra. En la instrucción ST no se utiliza el campo del registro destino, Rd. La instrucción JR lleva implícito el registro destino (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 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.

3.1. Estructuras de programa complejas

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.

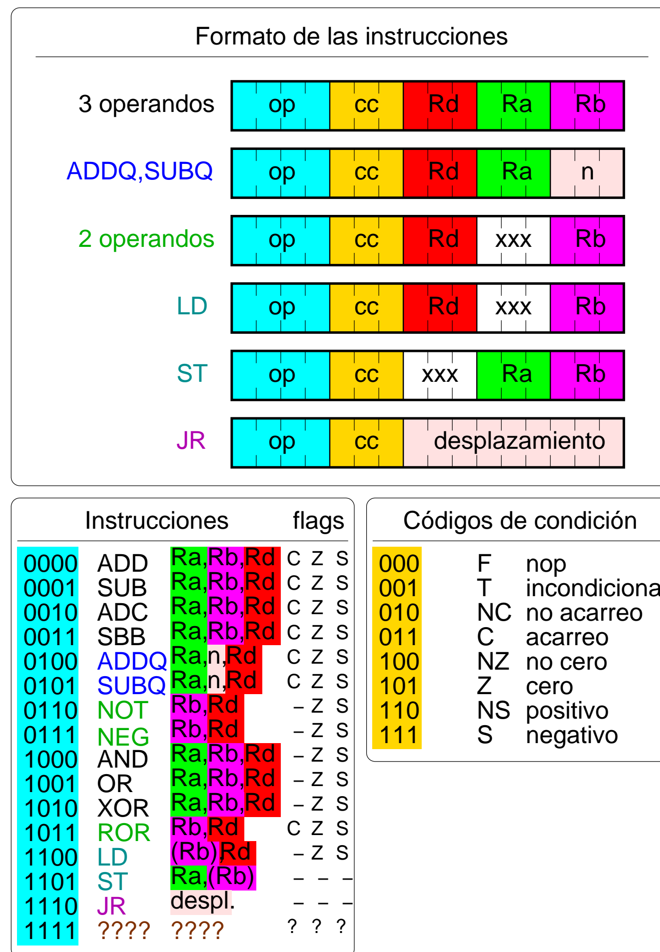


Figura 3: Codificación de las instrucciones en el procesador BN16.

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
```

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.

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

```
XOR R7,R7,R7  ; Pone a cero R7
```

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
```

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
   ...
```

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 crece hacia abajo.

Introducción de datos (R7) en la pila:

```
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 se sigue el proceso inverso:

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

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.

```
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
```

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
```

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

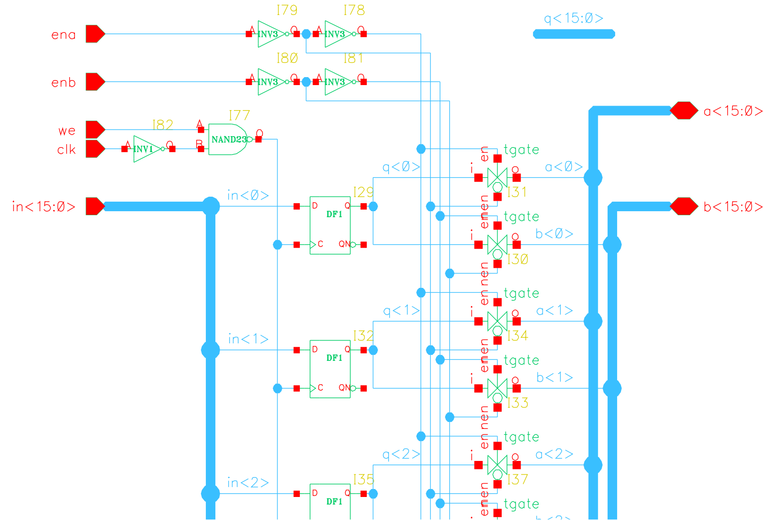
```
; Programa principal
    ADDQ PC,2,R2  ; Cálculo de la dirección de retorno
    JR RUTINA1    ; Salto a la subrutina
    NOP           ; Se ejecuta por estar en el pipeline
    ...           ; Aquí continúa la ejecución al retornar
; Subrutina de primer nivel
RUTINA1:
    SUBQ R1,1,R1  ; Guardamos 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 R6
RUTINA2:
    ...
    ADDQ R2,0,PC  ; Retornamos (además ponemos C=0)
    NOP           ; Se ejecuta por estar en el pipeline
```

4. Implementación CMOS

El procesador se ha diseñado usando el editor de esquemáticos de Cadence en una tecnología CMOS de $0.35\mu m$. El recuento de transistores ronda los 6700, lo que sitúa al BN16 en un nivel de complejidad intermedio entre el de un 6502 y un Z80. El diseño ha sido jerárquico: partiendo de una librería de elementos digitales básicos como las puertas lógicas, puertas de transmisión y flip-flops, se han implementado cada uno de los bloques funcionales de la CPU. A continuación se presentan los aspectos más destacados de estos circuitos.

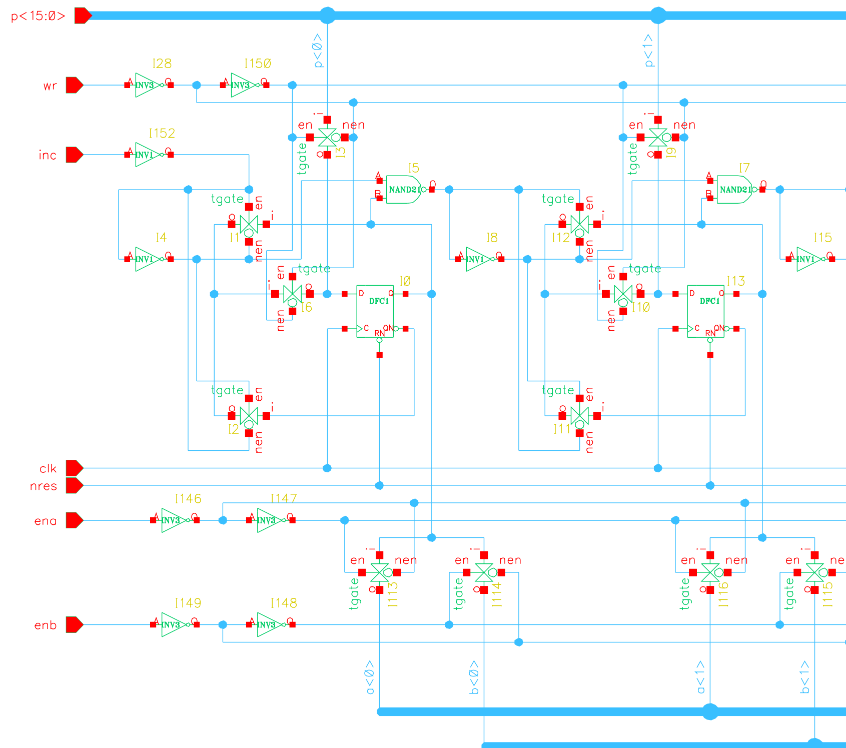
4.1. Registros

Registros R1 a R7



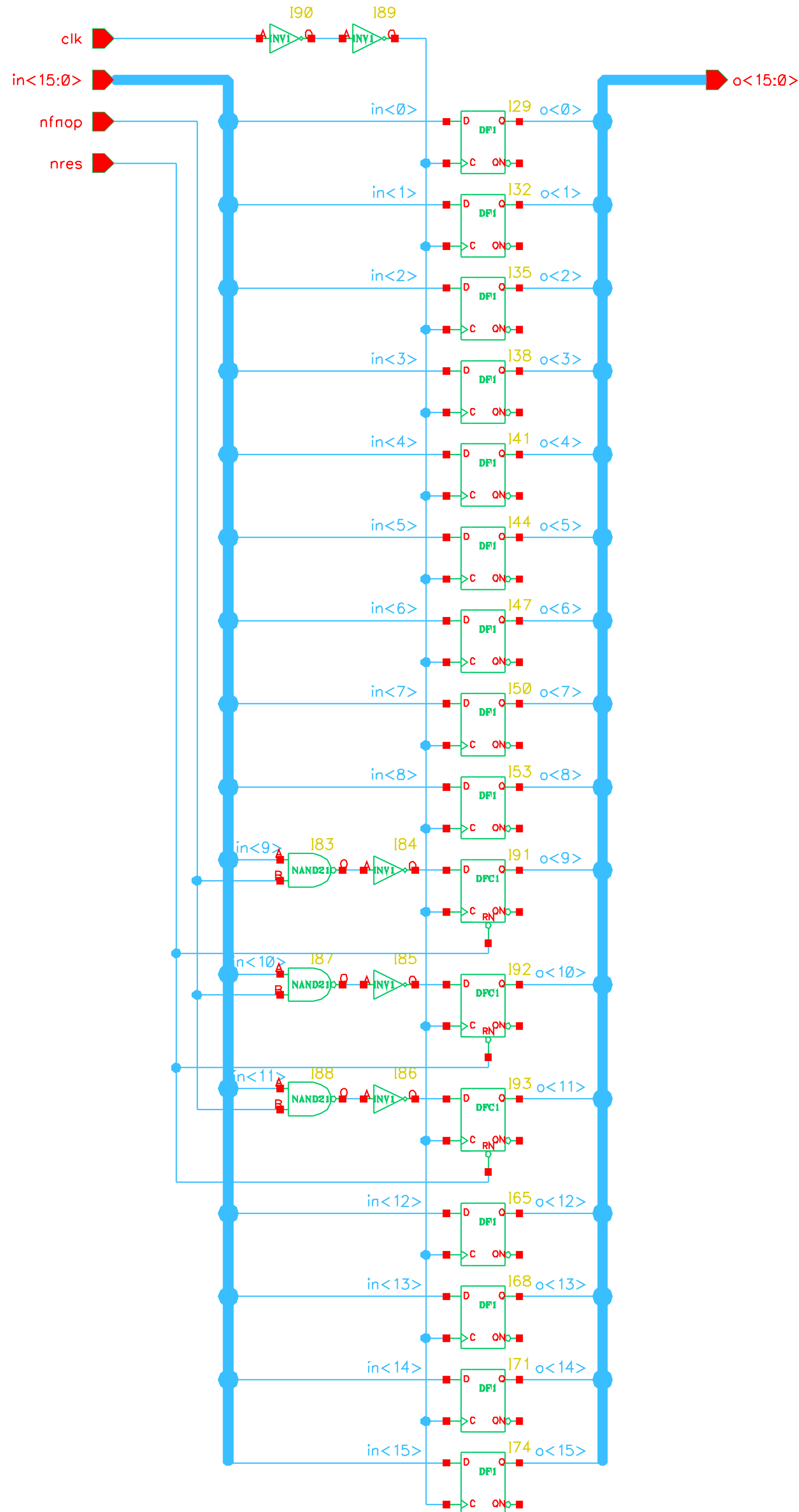
Los registros constan de 16 flip-flops con 2 salidas triestado. La escritura se realiza mediante un pulso en la entrada de reloj de estos flip-flops. Sólo se muestran los bits de menor peso de uno de los registros.

Registro PC



El registro contador de programa es más sofisticado. Es un contador síncrono con carga en paralelo síncrona y reset asíncrono que se basa en flip-flops de tipo T (flip-flop D más multiplexor) y puertas AND. Alternativamente se puede ver este bloque como un registro normal realimentado a través de un circuito incrementador construido con semisumadores y acarreo serie. La entrada de control *inc* hace que el PC se incremente cada ciclo de reloj cuando está en alto o mantenga su valor en caso contrario. La entrada de reset inicializa el PC con el valor cero lo que es crucial para que la CPU tenga un arranque previsible. En la figura se muestran tan sólo los dos bits menos significativos del registro PC.

Registro IR



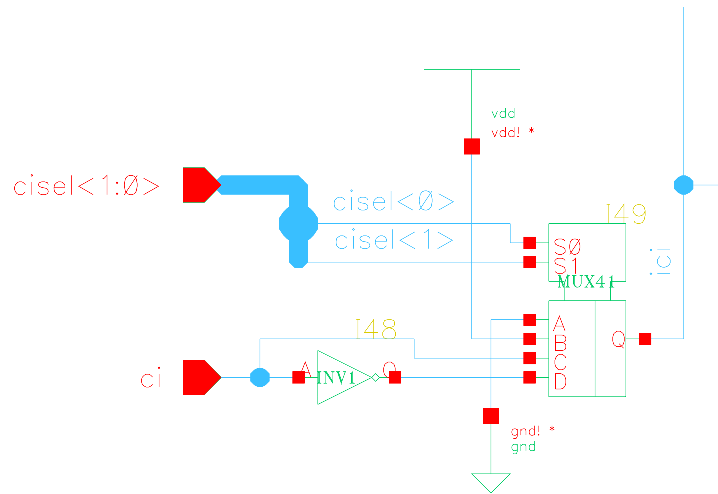
El registro IR incluye tres puertas AND para hacer 0 los 3 bits del código de condición cuando el contenido del bus de datos no es el código de una instrucción. Haciendo esto se consigue cargar una instrucción NOP en IR. Estos 3 bits también se hacen cero mediante una entrada de reset asíncrono para que al arrancar la CPU comience ejecutando una instrucción NOP.

4.2. ALU

Bit de la ALU

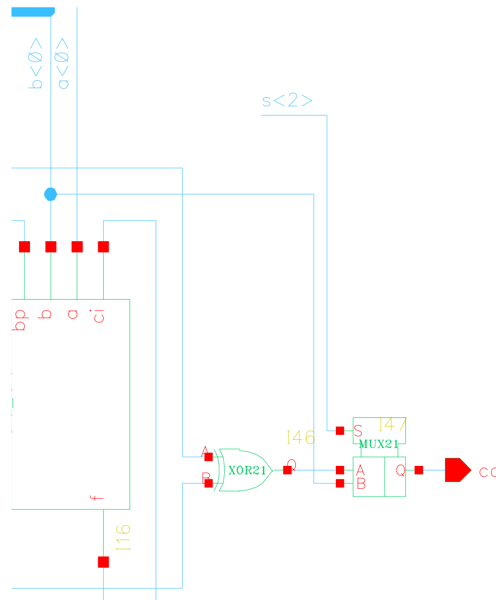
La ALU utiliza acarreo serie. Cada uno de los 16 bits de la ALU siguen este esquema. La entrada *invb* permite invertir los bits del operando B mientras que la entrada *ea* hace cero el operando A cuando *ea* vale cero. La salida *f* se puede elegir entre 6 posibles mediante puertas de transmisión CMOS.

Entradas de acarreo



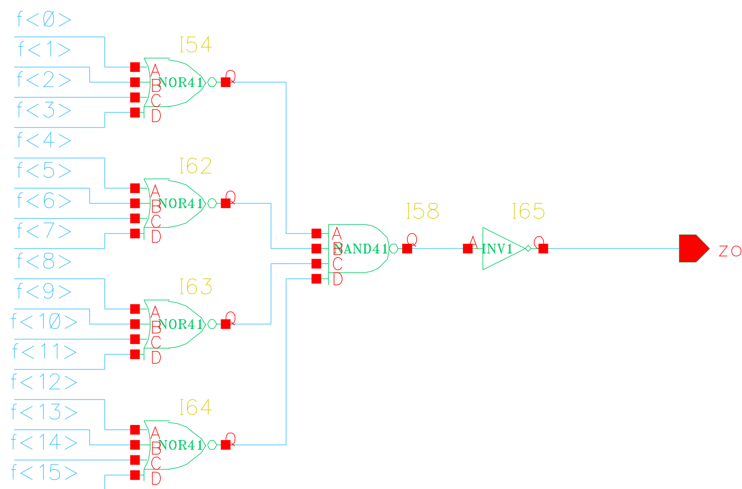
El acarreo de entrada a la ALU se puede elegir como cero, uno, el flag de acarreo o el complemento de dicho flag. Las instrucciones de resta necesitan un acarreo que es el complemento del usado en las sumas pues en las restas el flag C hace la función de *"débito"* (ingles: *Borrow*). Este es el tipo de comportamiento que se tiene en CPUs como el Z80 o el 8086, mientras que en el 6502 o el PowerPC no se invierte el acarreo.

Salida de acarreo



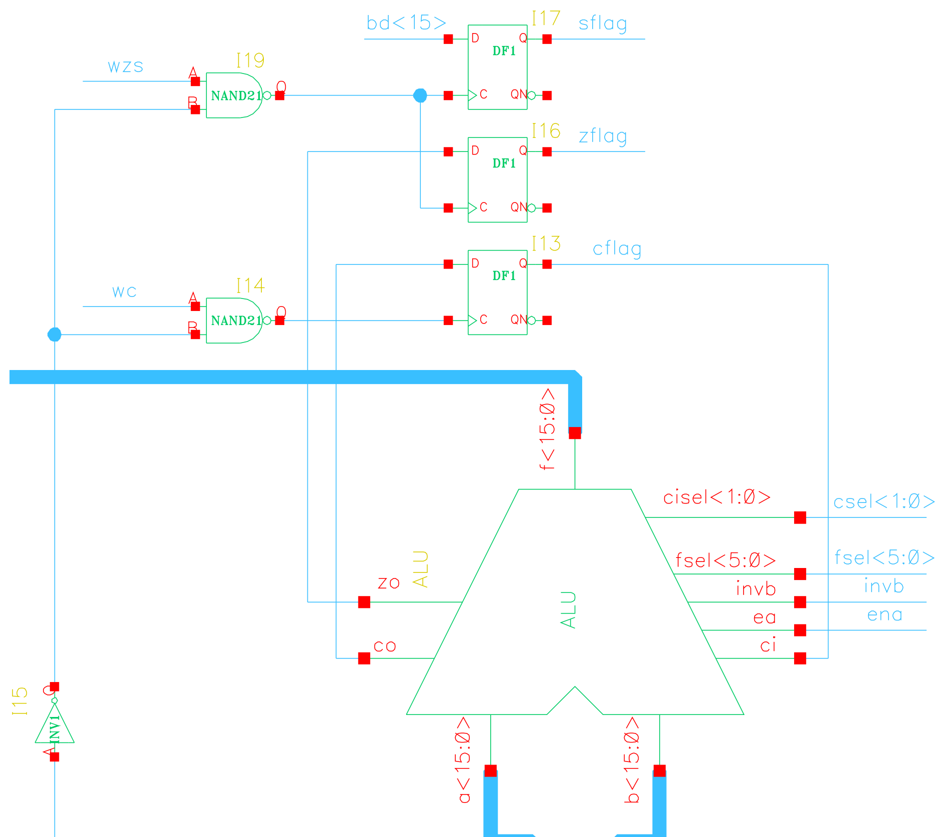
La salida de acarreo se ha de complementar en las instrucciones de resta (*Borrow*) y en la instrucción ROR el acarreo proviene directamente de b<0>.

Flag de cero



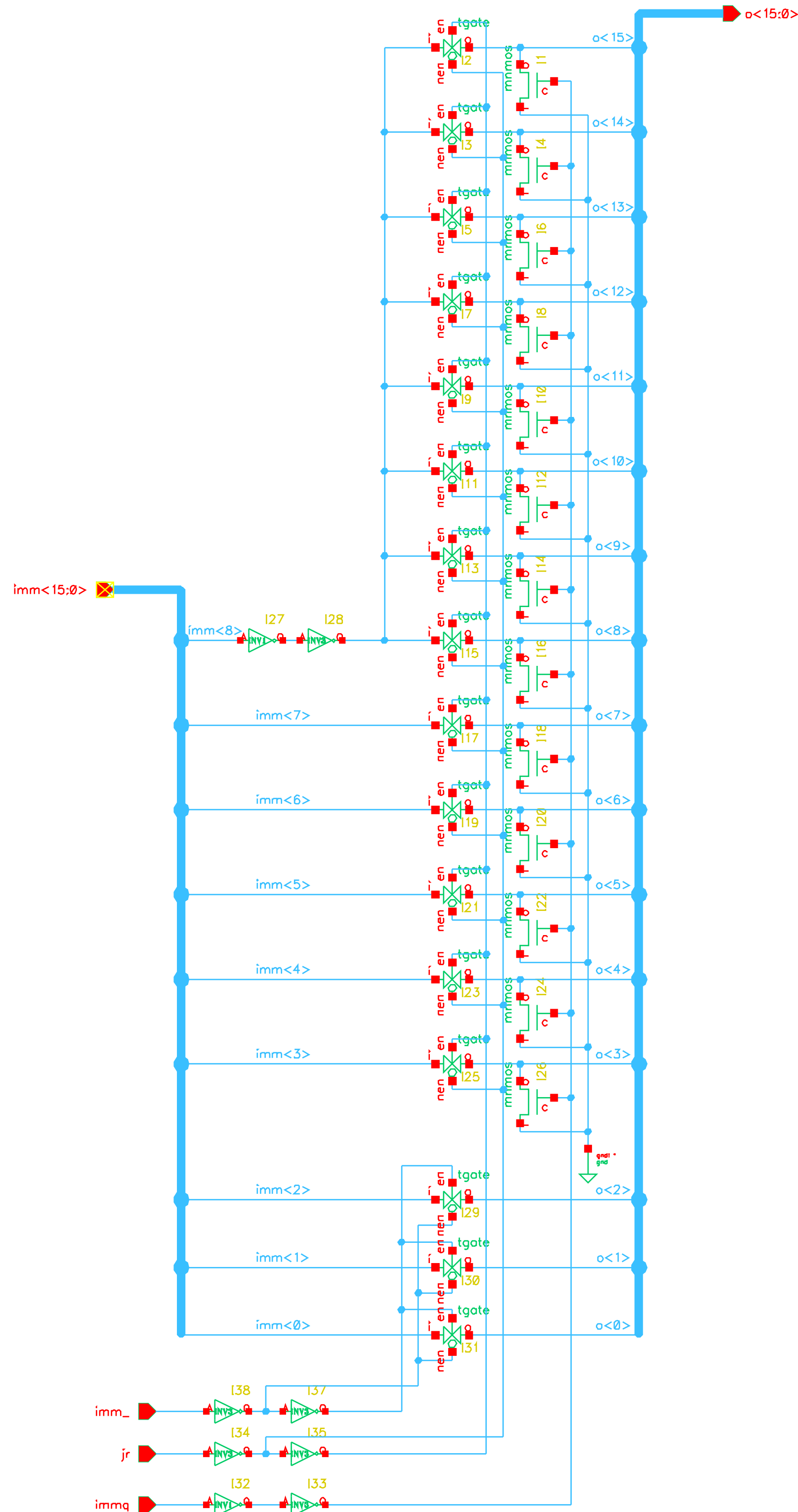
El flag de cero se activa cuando los 16 bits de salida de la ALU valen 0. La función NOR correspondiente se ha descompuesto en 4 bloques de 4 bits al no disponer en la librería de puertas de más entradas.

4.3. Flags



Cada flag se almacena en un FF. Las líneas de control de escritura en los flags se generan en la unidad de decodificación. Los flags de cero (Z) y signo (S) tienen una señal de escritura común.

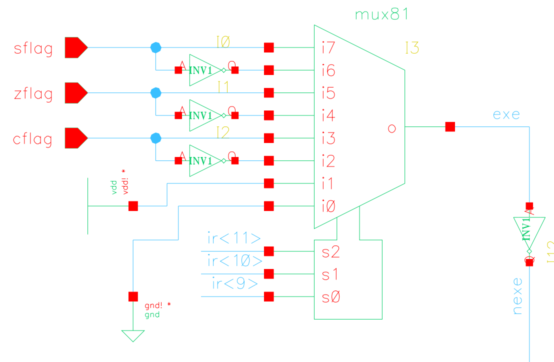
4.4. Operandos inmediatos



Las instrucciones que incluyen operandos inmediatos (ADDQ, SUBQ y JR) obtienen estos a partir del contenido del registro IR. El circuito de la figura genera el valor apropiado para los bits más significativos de la constante. En el caso de la instrucción JR se extiende el signo a la constante de 9 bits. En las instrucciones ADDQ y SUBQ se hacen cero los 13 bits más significativos de la constante.

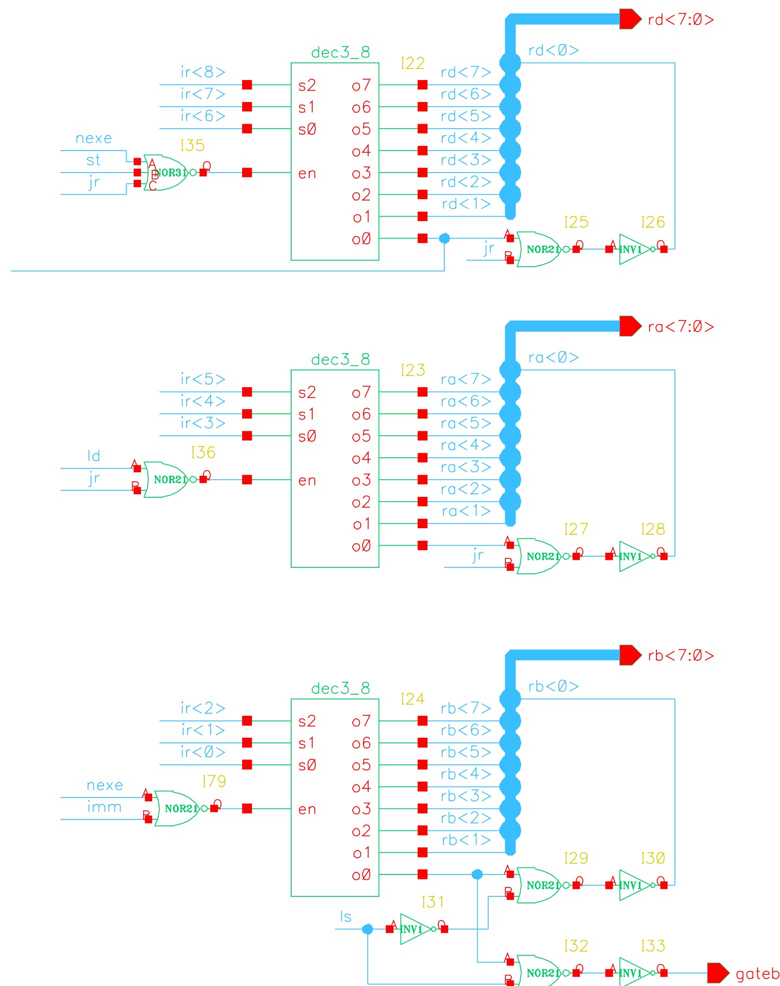
4.5. Decodificación de las instrucciones

Ejecución condicional



La ejecución de la instrucción contenida en IR depende de su campo de código de condición y del valor de los flags. Mediante el multiplexor de la figura se obtiene una señal lógica de *ejecutar / no ejecutar* que sirve de habilitación para el resto de la lógica de decodificación.

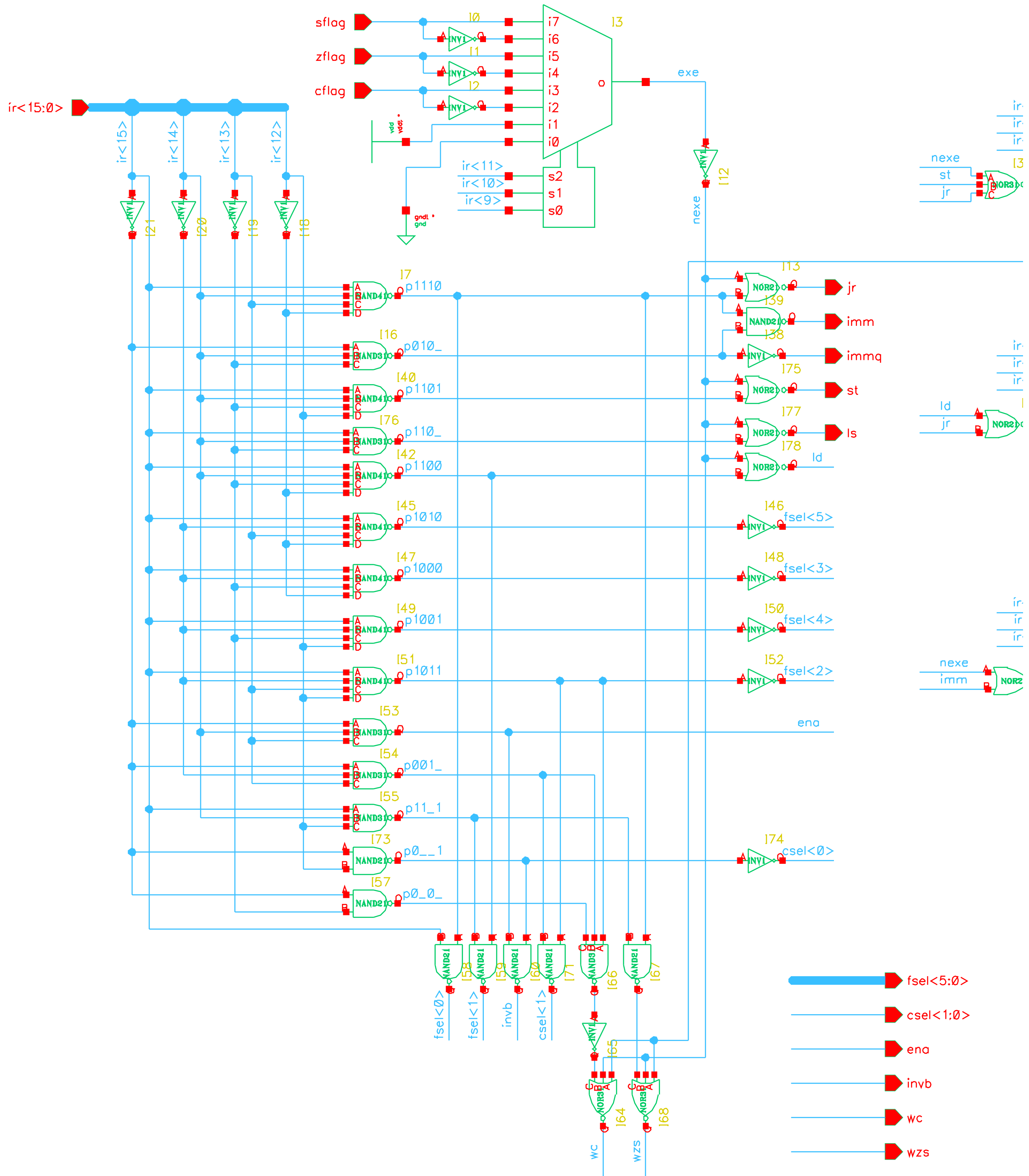
Selección de registros



Las señales de selección de registro se obtienen de los campos de registro de las instrucciones mediante tres decodificadores binarios de 3 a 8 líneas. El registro R0 es especial y se ha añadido la lógica necesaria para obtener sus señales de control. Además, las instrucciones con operandos inmediatos no deben seleccionar el registro Rb, la instrucción LD

no debe seleccionar el registro Ra y la instrucción ST no debe seleccionar un registro destino. Se ha incluido la lógica necesaria para habilitar los decodificadores en estos casos particulares.

Decodificación de operando



En la figura se muestra el resto de la lógica de decodificación de instrucciones. A partir de los 4 bits del código de operación se generan todas las señales necesarias para el control de la ALU, escritura en los flags y en memoria, control de las puertas *gateA* y *gateB*, etc.

5. Simulación

El procesador ha sido simulado a nivel funcional con un emulador escrito en lenguaje C. Mediante este emulador se han ejecutado programas sencillos que han servido para aclarar el funcionamiento de los diversos bloques funcionales de la CPU. Una vez diseñada la CPU a nivel de esquemáticos en Cadence se han generado a partir de dichos esquemáticos descripciones funcionales en lenguaje Verilog. A los netslist en Verilog se han añadido un bloque de memoria y un módulo principal en el que se interconectan CPU y memoria. La colección de módulos en Verilog se ha simulado con *cver* (Pragmatic Software) y los resultados se han visualizado con *gtkwave*. En la figura se muestra una captura de la CPU ejecutando el código de un programa de prueba.

Un detalle que el simulador de Verilog no es capaz de reproducir correctamente es la inicialización de los registros mediante instrucciones XOR Rx,Rx,Rx o SUB Rx,Rx,Rx. Cualquiera de estas instrucciones da como resultado un registro con el valor cero. Sin embargo, el valor inicial de los registros es desconocido ($XXXX) y para el simulador la operación XOR de X con X da el resultado X, con lo que el registro en cuestión sigue con un valor desconocido. Esto no ocurre en la realidad ya que los dos operandos de entrada de la función XOR son idénticos. Una forma de solventar este problema sin recurrir a un código demasiado estúpido consiste en utilizar R0 como registro fuente para la instrucción XOR. Dado que el valor del PC sí es conocido el resultado será 0:

```
init:
    XOR PC,PC,R1
    XOR PC,PC,R2
    ...
```

6. Versión 2: BN16/V2

La versión 2 del procesador BN16 difiere de la original en que se ha incluido soporte para interrupciones. Eso ha supuesto añadir dos registros, $R0_{INTERR}$ y $FLAGS_{INTERR}$ (Figura 4), que sustituyen al contador de programa y flags

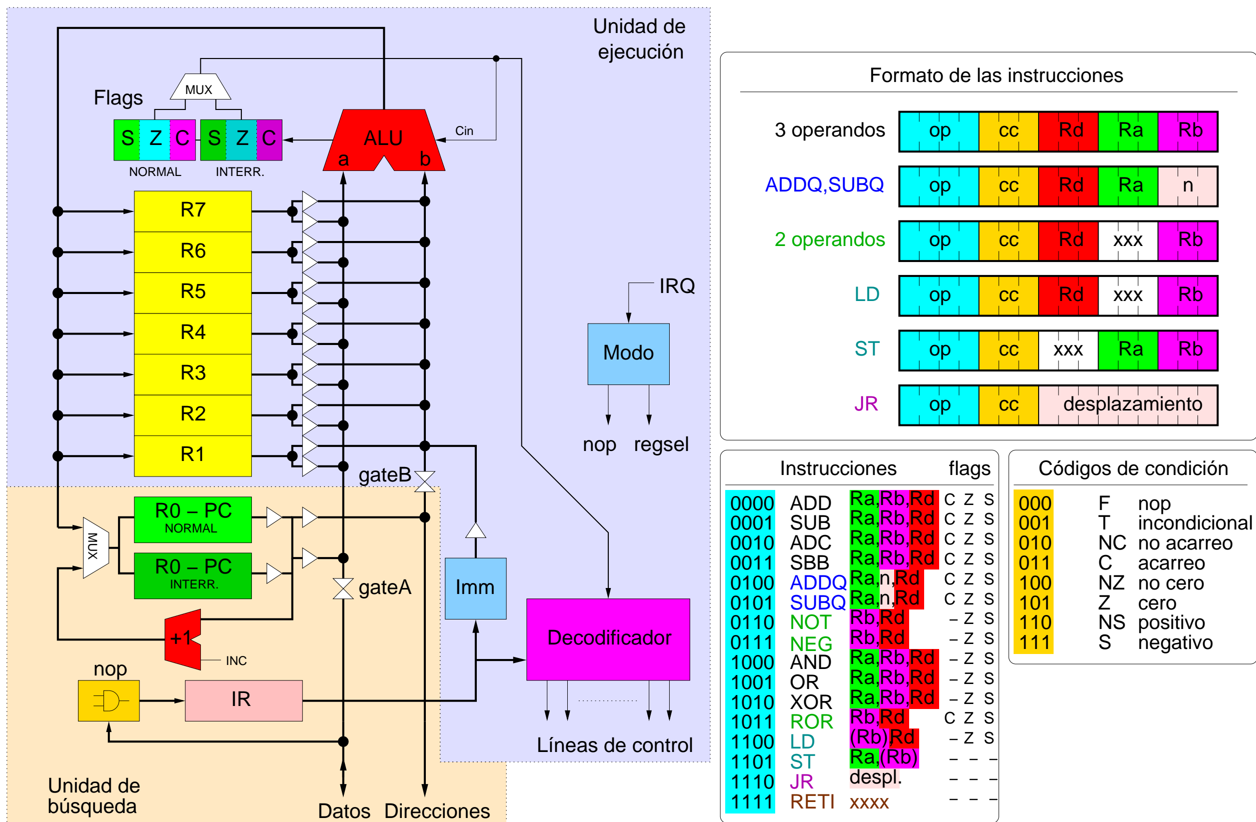


Figura 4: Diagrama de bloques de la CPU BN16/V2 y su correspondiente juego de instrucciones.

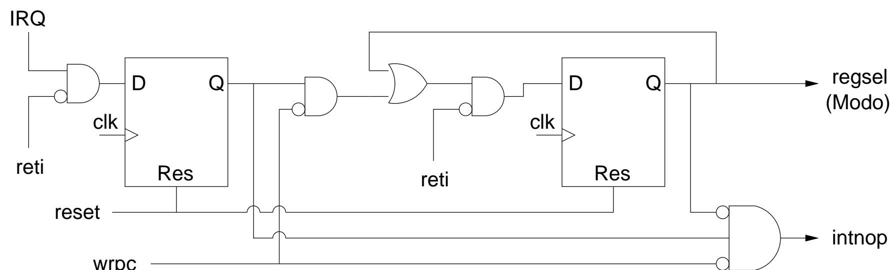


Figura 5: Detalle de la lógica de detección de interrupción (bloque *Modo*).

normales durante la ejecución de las rutinas de interrupción. Además el código de operación 1111 se ha asignado a la instrucción RETI (retorno de interrupción) con lo que ahora los 16 códigos de operación se corresponden con instrucciones válidas. La CPU puede funcionar en dos modos distintos: el modo normal y el modo interrupción. Se pasa al modo de interrupción cuando se recibe en la CPU una petición de interrupción y se retorna al modo normal al ejecutar la instrucción RETI. En el modo normal el contador de programa de interrupción permanece resetado con el valor 2, de modo que tras recibir una petición de interrupción comienza la ejecución de la rutina de interrupción en la dirección 2 (la dirección 0 es la de RESET). La rutina de interrupción finaliza con la instrucción RETI seguida de un NOP. Este NOP se necesita puesto la la instrucción que sigue a RETI ya se ha cargado en IR cuando se ejecuta RETI y su ejecución podría alterar los registros $R0_{NORMAL}$ y/o $FLAGS_{NORMAL}$.

La detección de la interrupción se hace de acuerdo con el esquema de la figura 5. En el primer ciclo de reloj tras la puesta a uno de IRQ se fuerza una instrucción NOP en IR a través de la línea de control *intnop*. Esto se hace antes de conmutar a los registros del modo de interrupción para que la instrucción actual pueda terminar de ejecutarse en los registros del modo normal. Además se inhibe el incremento del contador de programa de modo que cuando se retorne al modo normal la instrucción que fue descartada volverá a cargarse en IR y se ejecutará. Si la instrucción que se está ejecutando utiliza R0 como puntero (LD o ST) sí que se debe permitir su incremento. En el segundo ciclo de reloj tras la puesta a uno de IRQ *regsel* pasa a uno y se seleccionan los registros $R0_{INTERR}$ y $FLAGS_{INTERR}$ (modo de interrupción). La ejecución

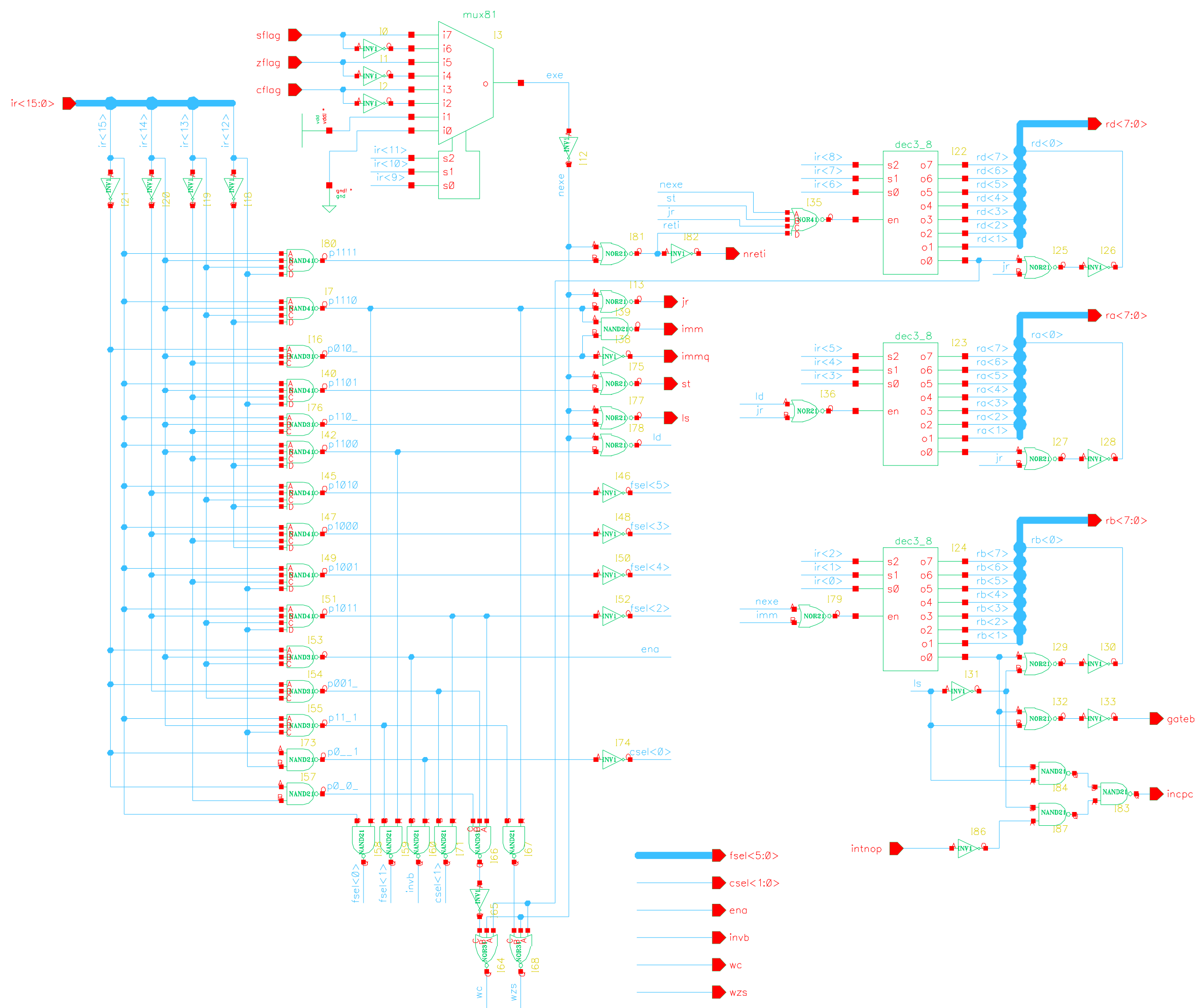


Figura 6: Lógica de decodificación de instrucciones para la CPU BN16/v2.

comienza en la dirección de memoria 2. Los valores de los registros $R0_{NORMAL}$ y $FLAGS_{NORMAL}$ permanecen inalterados hasta que se retorna al modo normal tras ejecutar RETI. La petición de interrupción no se atiende si la instrucción que va a ser interrumpida modifica el contador de programa (salto). Esto se hace para que se pueda mantener la funcionalidad de los saltos retardados en los que la siguiente instrucción al salto se ejecuta. De no inhibir las interrupciones durante los saltos la instrucción que sigue al salto podría quedar sin ejecutarse si se ha producido una interrupción. En el esquema de la figura 5 la línea *wrpc* indica cuándo se va a modificar el registro PC y se usa para inhibir las interrupciones en esos ciclos de reloj. La ejecución de la instrucción RETI hace que los dos flip-flops se carguen con el valor 0 pasando al modo normal durante al menos dos ciclos de reloj. Así se garantiza que se ejecutará como poco una instrucción en modo normal entre dos interrupciones.

Se han hecho algunos otros cambios en la estructura de la CPU para evitar repetir lógica innecesaria. Ahora los registros R0 son simples grupos de flip-flops de tipo D en lugar de contadores. El contaje se consigue con un circuito incrementador basado en semisumadores que puede usarse con cualquiera de los registros R0. Este circuito incrementador en realidad ya estaba de forma implícita en el contador de programa de la CPU original, así que no supone un aumento de la complejidad del procesador. También ha sido necesario modificar la lógica de decodificación (figura 6) que ahora debe reconocer el código de operación 1111 y generar la señal *reti* correspondiente. También se ha incluido en el bloque de decodificación la lógica para generar la señal *incpc*, que indica cuándo se ha de incrementar el PC.

Con las últimas modificaciones el número de transistores de la CPU ha aumentado a 7677.

7. Comentarios finales

Durante la fase de diseño del procesador BN16 aún no conocía la arquitectura ARM, y sin embargo, no cabe duda que el BN16 tiene muchos parecidos con el ARM. No deja de sorprenderme que el mismo condicionante de la simplicidad, pero no a cualquier precio, haya dado lugar a arquitecturas tan parecidas. Los procesadores ARM son de 32 bits y deben considerarse ya como CPUs serias mientras que el BN16 con sus 16 bits no pasa de ser una curiosidad académica. Por supuesto, todo se puede mejorar. A continuación se enumera una lista de posibles mejoras del procesador BN16:

- Implementación mediante FPGA. Los esquemáticos pueden que fuese necesario rehacerlos para adaptarlos a las herramientas de programación de las FPGAs. En particular, las puertas de transmisión son unos dispositivos que no encajan bien con los compiladores para FPGAs. Habría que sustituirlas por multiplexores.

- ALU con acarreo adelantado. En el diseño actual la ALU tiene acarreo serie y esto da lugar a retardos muy largos que limitan la máxima frecuencia de reloj a unos 30MHz para la tecnología de $0.35\mu m$.

- Una etapa más de pipeline: Búsqueda $\rightarrow$ DECODIFICACIÓN $\rightarrow$ Ejecución. De esta manera no se sumarían los retardos de la unidad de decodificación y de la ALU.

- Diseño de una librería de periféricos para añadir a la CPU y obtener microcontroladores “a la carta”.