UNIVERSIDAD CARLOS III DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

DEPARTAMENTO DE TECNOLOGÍA ELECTRÓNICA

PROYECTO FIN DE CARRERA

DISEÑO Y SÍNTESIS EN VHDL DEL NÚCLEO DEL MICROCONTROLADOR 68HC05

INGENIERÍA TÉCNICA INDUSTRIAL
ELECTRÓNICA INDUSTRIAL

AUTORA: Mª BELÉN FERNÁNDEZ SAAVEDRA
DIRECTOR: RAÚL SÁNCHEZ REILLO

JUNIO 2004

A mis padres, a mi hermana
y mi abuela que está en el cielo.

AGRADECIMIENTOS

A mis padres Milagros y Martín porque gracias a su preocupación por sus hijas y a su esfuerzo ha sido posible que las dos estudiáramos una carrera. Gracias por educarnos, por darnos todo lo que hemos necesitado, por vuestro apoyo incondicional, por estar siempre a nuestro lado y sobre todo por vuestro cariño. Un beso muy fuerte.

A mi hermana Rocío que la quiero un montón y que siempre ha estado animándome con la carrera y más aún en época de exámenes. Roci, gracias por tus consejos y te deseo mucha suerte con tu tesis.

A mi abuela Martina y a mis tíos Cali, “el Tito” y Joaquín por su cariño y por animarme a trabajar y a seguir siempre hacia adelante.

A mi tutor Raúl por la dedicación que me ha prestado, por atender mis dudas, por sus enseñanzas y sugerencias. A Mario por su valiosa ayuda y al resto de profesores del Departamento por responder a todas mis preguntas.

A todos mis amigos por los buenos momentos que hemos pasado juntos. Sobre todo a mis compañeros de Universidad que son con los que más he compartido experiencias a lo largo de la carrera y que han hecho más entretenidas y agradables muchas de las clases, prácticas y exámenes. Os deseo lo mejor en el futuro.

A mis compañeros de laboratorio con los que más tiempo he pasado mientras realizaba este proyecto y que también me han prestado su ayuda cuando la he necesitado. Mucha suerte en vuestros proyectos.

Y también, aunque ya no esté entre nosotros, a mi abuela Milagros por hacerme compañía mientras estudiaba muchas de las asignaturas de esta carrera, por su sonrisa, sus bromas, y sobre todo por su cariño, por sus cuidados, por sus consejos, por estar siempre orgullosa de sus nietas y por su manera de vivir. Gracias yaya, que lo de abuela siempre dijiste que te hacía más vieja.

¡¡ MUCHAS GRACIAS A TODOS !!

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

CAPÍTULO 1

INTRODUCCIÓN Y OBJETIVOS DEL PROYECTO

1. INTRODUCCIÓN Y OBJETIVOS DEL PROYECTO

Hoy en día el uso de los microcontroladores está muy extendido. Estos dispositivos debido a su bajo consumo, alta fiabilidad, reducido tamaño, bajo coste y gran versatilidad, se encuentran en la mayoría de los aparatos que empleamos a diario. Los podemos encontrar controlando el funcionamiento de nuestros teléfonos, electrodomésticos, tarjetas inteligentes, automóviles, robots, juguetes, etc.

Por esta razón el mercado nos ofrece una amplia gama de estos productos variando sus posibilidades, es decir, con diferentes encapsulados, pines de entrada-salida, diversos tipos y tamaños de memoria, diferentes juegos de instrucciones, modos de direccionamiento, etc.

Pero actualmente, a pesar de las diferentes opciones que ofrece la industria, a la hora de utilizar un microcontrolador hay que emplear el dispositivo completo, es decir, el núcleo junto con todos los periféricos que lo acompañan, al formar parte todos ellos de un mismo circuito integrado. Esto presenta dos inconvenientes:

- El primero de ellos ocurre cuando para determinadas aplicaciones en las que se usan microcontroladores no se utilizan todos los recursos de los que estos disponen al no ser necesarios, desperdiciándose así parte de ellos al estar todos integrados dentro de un mismo chip.
- El otro inconveniente surgue en sentido contrario, es decir, cuando para el diseño de un circuito es necesario utilizar más recursos que los que un componente por sí solo nos proporciona y hay que añadir otros dispositivos ocupando mayor espacio y teniendo que realizar conexiones entre sí que pueden no ser tan fiables como si estuvieran todos integrados dentro del mismo chip.

Dadas las posibilidades de la industria, las necesidades dentro del campo del diseño de circuitos y para simplificar las tareas en la creación de nuevos circuitos, en este proyecto se ha diseñado y sintetizado el núcleo del microcontrolador de 8 bits 68HC05 cuyo fabricante es Motorola.

La razón de diseñar el núcleo del microcontrolador es que este es un elemento imprescindible, ya que éste es común para todos los componentes de la familia, de manera que, con sólo añadirle los periféricos necesarios, podría ser utilizado para una determinada aplicación.

El modelo que se va a diseñar en este proyecto ha de realizar el mismo juego de instrucciones y tener las mismas posibilidades de comunicación con el resto de los periféricos, que el dispositivo fabricado por Motorola.

Además de esto, este modelo ha de ser sintetizable, de manera que el núcleo se pueda implementar y probar en una placa de desarrollo.

Para llevar a cabo esta tarea se ha utilizado el lenguaje de descripción *hardware* VHDL (*VHSIC Hardware Description Language*). Este lenguaje hace posible la descripción del *hardware* a diferentes niveles de abstracción, pudiendo adaptarse a distintos propósitos y utilizarse en diferentes fases de diseño. Esto permitirá la división del núcleo en componentes más simples y la posibilidad de verificación de cada componente por separado. Así como también hará posible que una vez que se disponga del núcleo, se le podrán añadir los dispositivos necesarios para realizar cualquier aplicación que se desee en futuras aplicaciones.

El VHDL es también un lenguaje estándar que facilita la adaptación de los diseños a diferentes tecnologías. De esta forma el núcleo diseñado será independiente de la tecnología empleada y podrá ser implementado en la tecnología actual de cada momento.

El dispositivo elegido en la implementación del diseño es una FPGA (*Field Programmable Gate Array*). Este es un dispositivo programable con una alta escala de integración que va a permitir poder probar diseños amplios, como es el caso del núcleo del microcontrolador. La programación de una FPGA puede llevarse a cabo por el propio usuario utilizando una herramienta *software* adecuada, en un tiempo muy corto, de manera que pueden realizarse cambios fácilmente, si el diseño no se adecua a las especificaciones impuestas en un principio.

En la memoria del presente proyecto se va a explicar el desarrollo de todo el proyecto. Para ello esta se ha dividido en los siguientes apartados:

- Descripción general del microcontrolador 68HC05. Este apartado se encuentra en el capítulo 2 de la memoria. En él se pretende realizar una pequeña introducción al microcontrolador 68HC05. Para ello se explican sus características más importantes, los periféricos que puede llegar a tener y las diferentes versiones de dispositivos que componen la familia con las particularidades de cada uno.
- Descripción del núcleo del microcontrolador. Este es el capítulo 3 de la memoria. En él se realiza una descripción más específica de cómo es el núcleo del microcontrolador, explicando sus modos de direccionamiento, las interrupciones que puede llegar a tener, así como se entra ya en detalle de las distintas partes que lo componen y la forma en que se ha diseñado cada una de ellas.
- Juego de instrucciones del microcontrolador. Este apartado compone el capítulo 4 de la memoria y en él se enumera el conjunto de instrucciones del microcontrolador explicando concretamente cómo se ha realizado cada una de ellas.
- Realización de pruebas y resultados obtenidos. Esta parte ocupa el capítulo 5 de la memoria y en él se relatan las diferentes pruebas realizadas para la verificación del correcto funcionamiento del diseño y de los resultados obtenidos al implantarlo en una FPGA. Así como cúal ha sido la FPGA y la placa de desarrollo utilizadas para ello.
- Conclusiones, líneas futuras y anexos. Estas componen los últimos capítulos de la memoria y en ellos se dan a conocer las conclusiones obtenidas en el desarrollo del proyecto, los posibles trabajos futuros relacionados con el mismo y el código en VHDL utilizado para el diseño del núcleo.

CAPÍTULO 2

VISIÓN GENERAL DEL MICROCONTROLADOR

2. VISIÓN GENERAL DEL MICROCONTROLADOR

Este capítulo trata de ser una introducción al microcontrolador 68HC05 objeto de este proyecto y dar una visión generalizada del mismo. Esta va a ser muy importante ya que a la hora de diseñar el núcleo es necesario conocer, con qué otros elementos (periféricos) puede llegar a interactuar, los recursos necesarios de los que debe disponer, y qué posibilidades debe alcanzar para ser capaz de controlar y atender todas las peticiones que le sean solicitadas.

Para ello, por un lado se va a describir cómo es un microcontrolador 68HC05 y todos los elementos que puede llegar a tener. Dentro de esta explicación se va a hacer más hincapié en todos los elementos que formando parte del dispositivo no constituyen el núcleo en sí, ya que al ser el diseño de este el objeto del proyecto, se tratará en profundidad en los próximos capítulos.

Por el otro lado, se va describir la situación actual de la familia de microcontroladores 68HC05, así como las diferentes versiones que hoy en día existen en el mercado, resaltando las características más significativas de cada una de ellas, sus principales aplicaciones y posibles usos dentro de algunas áreas específicas de la industria como son la automoción, la informática, y las telecomunicaciones, así como de la industria en general.

2.1. DESCRIPCIÓN GENERAL

La descripción general del microcontrolador 68HC05 consta de los siguientes apartados:

1. El primero de todos es una enumeración sus características principales, tanto las del *hardware* como las de programación, para proporcionar una idea global del dispositivo y de su capacidad.
2. En este segundo apartado se realiza una breve descripción de la arquitectura que presenta este microcontrolador.
3. Por último se detallan todos los periféricos que puede llegar a tener una determinada versión del microcontrolador y los tipos de memoria de los que dispone.

2.1.1. CARACTERÍSTICAS GENERALES DE LA FAMILIA 68HC05

El 68HC05 es un microcontrolador de propósito general fabricado por Motorota con una CPU de 8 bits y con las siguientes características:

- Características HARDWARE:
  - Longitud de palabra de 8 bits.
  - Tecnología de fabricación HCMOS.
  - Alimentación de 3 a 6 V.
  - Oscilador interno.
  - Pines bidireccionales de entrada/salida.
  - Velocidad del bus interno de hasta 4 MHz con una alimentación de 5 V y de hasta 2 MHz si la alimentación es de 3 V.
  - Memoria RAM, ROM ó PROM de varios tipos, con la posibilidad de elegir su configuración mediante software.
  - Periféricos adicionales dependiendo de la versión de la cuál se trate:
    - Temporizador automático de 16 bits.
    - Temporizador multifunción de 15 etapas.
    - Watchdog.
    - Interfaz para las comunicaciones serie (SCI).
    - Interfaz serie para periféricos (SPI).
    - Convertidor analógico/digital.
    - Generador PWM.

- Características de PROGRAMACIÓN:
  - Amplio juego de instrucciones de fácil manejo.
  - Posibilidad de manipulación de forma aislada de bits.
  - Realización de multiplicaciones sencillas de 8 bits.
  - 8 modos de direccionamiento con la posibilidad de direccionamiento indexado de 8 y 16 bits para el acceso a tablas.
  - Entradas/salidas mapeadas en memoria.
  - Dos modos de espera de bajo consumo de energía (Wait y Stop).
  - Amplio conjunto de interrupciones programables.

A continuación, en la Figura 2.1, se muestra un diagrama de los bloques de uno de los miembros de esta familia de microcontroladores, el 68HC705C8 a modo de ejemplo. En esta figura se puede observar los distintos periféricos que componen este dispositivo, así como su núcleo. Al ser un componente concreto de la familia no dispone ciertos módulos otros si poseen pero proporciona una idea general de cómo son estos microcontroladores.

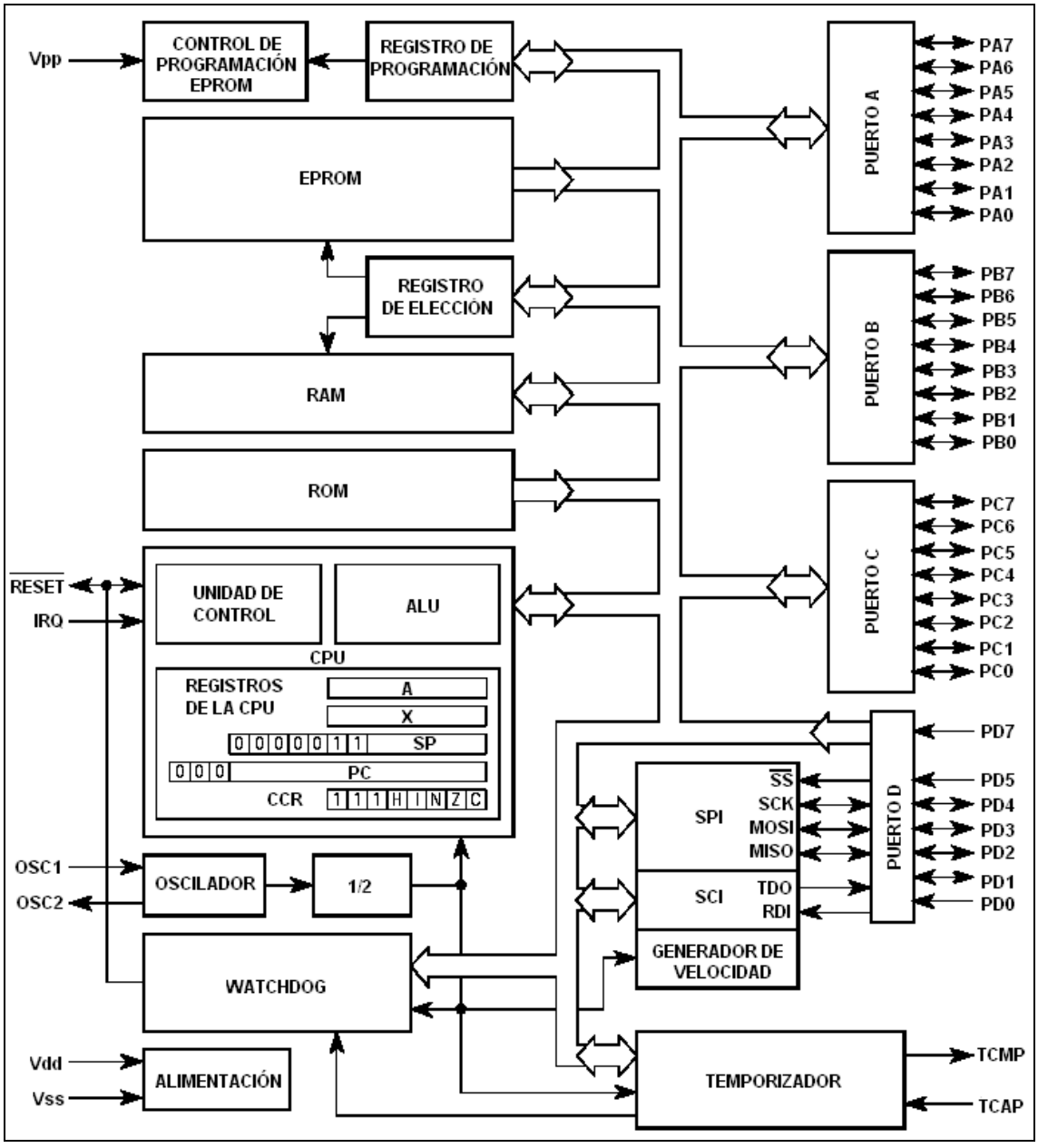


Figura 2.1 Diagrama de bloques del microcontrolador.

Una vez enumeradas las características principales, se va a describir cómo es la arquitectura de esta familia de microcontroladores, cuáles son los periféricos internos que los forman y qué tipos de memoria pueden tener, es decir, todo lo relacionado con las características *hardware* de los periféricos del dispositivo. En cuanto a las características *hardware* del núcleo y todas las características de programación, se tratarán más detalladamente en el capítulo siguiente, ya que forman la parte más importante a tener en cuenta a la hora de realizar el diseño del núcleo de este microcontrolador.

2.1.2. ARQUITECTURA DEL MICROCONTROLADOR

El 68HC05 es un microcontrolador de 8 bits con una organización específica denominada arquitectura Von Neumann. Esta arquitectura, como se muestra en la Figura 2.2, consta de una unidad de control (CPU) interconectada con una única memoria. En la memoria, se almacenan las instrucciones de programa y los datos. La conexión se realiza por medio de un bus de datos, un bus de direcciones y un bus de control. El bus de direcciones identifica la posición de memoria a la cuál se quiere acceder, mientras que el bus de datos es el que transfiere el dato de la CPU a la memoria o viceversa. El bus de control es el encargado de dar las órdenes necesarias para que se produzca la transferencia de información entre la memoria y la CPU. Estas órdenes provienen de la CPU, de su unidad de control, y van dirigidas a la memoria.

El número de líneas del bus de direcciones depende del tamaño de la memoria, es decir, de cuantas posiciones de memoria son accesibles. Por el contrario, el número de líneas del bus de datos tiene que ver con la longitud de palabra de los datos que maneja el microcontrolador. En este caso serán 8 líneas.

El bus de control no es un bus propiamente dicho, sino simplemente es un conjunto de señales aunque normalmente se le denomina como bus. En este caso este bus está compuesto por dos señales: una señal que habilita la memoria, *output enable* (oe), y otra que indica el sentido de los datos, es decir, si la CPU va a leer o escribir en la memoria, *read/write* ($r/\overline{w}$).

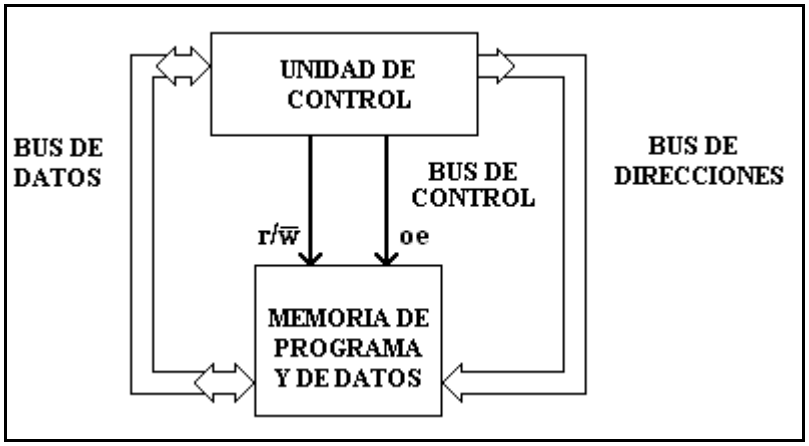


Figura 2.2 Arquitectura de Von Neumann.

2.1.3. PERIFÉRICOS QUE COMPONEN EL MICROCONTROLADOR

Para resolver problemas del mundo real, un microcontrolador debe tener además de una CPU potente y suficiente capacidad de memoria, una serie de recursos hardware que le permitan acceder a la información del mundo exterior. A la vez que la CPU recoge la información y procesa los datos, también debe poder efectuar cambios en el mundo exterior. Éstos dispositivos hardware, llamados periféricos, son la ventana al exterior del microcontrolador.

Los periféricos internos hacen que se extiendan las capacidades de los microcontroladores, proporcionando a los dispositivos una mayor versatilidad. Esto hace al microcontrolador un dispositivo mucho más competitivo. También permiten que se reduzca la carga de proceso en la CPU al liberarle de funciones que son realizadas por ellos, y tan sólo reclamando su atención, cuando es necesario, por medio de las interrupciones.

Los dispositivos periféricos que podemos encontrar en las diferentes versiones del microcontrolador 68HC05 son los siguientes:

- Oscilador o reloj.
- Puertos paralelos.
- Puertos serie.
- Temporizadores.
- Convertidor analógico/digital.
- Generador de PWM (*Pulse Widht Modulation*).

2.1.3.1. OSCILADOR

Todos los procesos implicados en la ejecución de un programa deben llevarse a cabo de forma ordenada y sincronizada. Por ello, es indispensable en el dispositivo la presencia de un oscilador o reloj que proporcione la señal de referencia necesaria para que efectúe su labor de control y manipulación secuencial con la cadencia adecuada.

Estos microcontroladores presentan dos entradas de reloj a las que se les puede conectar un cristal, un circuito RC, o bien una señal externa de reloj. Cualquiera de ellas sirven para realizar el control del oscilador interno del dispositivo que simplemente divide la frecuencia de la señal de entrada que reciben entre dos.

2.1.3.2. PINES DEL MICROCONTROLADOR

Los pines del microcontrolador son los principales encargados del intercambio de información con el mundo exterior. Son los responsables de que, tanto el microcontrolador reciba la información necesaria, así como, de transmitírsela al resto de componentes que estén conectados a él. Existen dos tipos de pines en el microcontrolador: los pines bidireccionales de entrada/salida que se les conoce normalmente como puertos paralelo y los pines que, o bien son utilizados como entradas ,o bien como salidas.

Los pines que forman parte de los puertos se utilizan principalmente para el intercambio de datos entre el dispositivo y el exterior y de ellos se hablará cuando se traten los puertos paralelo. La finalidad del resto de los pines tiene que ver con la obtención de otro tipo de fuentes del exterior que no son necesariamente datos, pero que son imprescindibles para el correcto funcionamiento de todos los periféricos que forman parte del microcontrolador. Estos pines son los siguientes:

- Los que le proporcionan la alimentación necesaria para su funcionamiento. Tanto el pin V_{DD} el cual proporciona la tensión nominal, como el pin V_{SS}, la masa del dispositivo.
- La entrada de RESET para la reinicialización del componente.
- Los que permiten la entrada de interrupciones de dispositivos externos como por ejemplo el pin $\overline{\text{IRQ}}$ (*Maskable Interrupt Request*).
- Los pines que permiten la conexión de un cristal o una entrada externa de reloj necesarios para la generación del reloj interno.

- Los que permiten una comunicación serie del componente con el exterior.
- Los pines relacionados con el funcionamiento del temporizador programable.
- Y demás tipos que también dependen de las necesidades que tenga el componente de la familia en cuestión, como puede ser si posee convertidor analógico/digital, salida PWM, memoria PROM que necesita ser programada, étc.

En la mayoría de los dispositivos de la familia no se disponen de los pines necesarios para todas las entradas y salidas que el microcontrolador precisa, por lo que algunos de estos pines que se acaban de nombrar, son comunes a los de los puertos paralelo bidireccionales. Este es el caso del llamado puerto D, en el que aun siendo un puerto paralelo, muchos de sus pines se utilizan para realizar la comunicación serie del microcontrolador.

2.1.3.3. PUERTOS PARALELOS

Como ya se ha comentado anteriormente, el HC05 tiene diversos pines bidireccionales de entrada/salida. Al conjunto de varios de estos pines se le denomina puerto. Dependiendo del miembro de la familia que se trate tendrá un determinado número de puertos. Estos puertos son los que permiten al microcontrolador disponer tanto de entradas, como de salidas en paralelo. Para poder utilizar un determinado pin del puerto como entrada o como salida hay que configurarlo previamente.

La mayoría de los componentes disponen de 4 puertos denominados A, B, C y D. Suelen disponer de 8 pines cada uno ya que este es el tamaño de palabra que maneja esta familia de microcontroladores con la excepción del puerto D que es un puerto especial cuyos pines tienen funciones comunes con otros periféricos como ya se ha comentado en el apartado anterior.

2.1.3.4. PUERTOS SERIE

Algunos miembros de la familia 68HC05 contienen periféricos que permiten a la unidad de control enviar bits consecutivamente a otros dispositivos externos. Usando

una estructura de comunicación en serie, en lugar de un formato en paralelo, se requieren menos pines de E/S para realizar la función de la transferencia de información.

Existen varios tipos de puertos serie básicos en la familia:

- SCI (*Serial Communications Interface*): Este puerto de comunicaciones serie, es un puerto transmisor/receptor asíncrono universal (UART), es decir, se comunica de forma asíncrona con otros dispositivos. Este tipo de puerto serie utiliza la interfaz hardware más simple, es decir, solamente se requieren dos pines para la transmisión de datos bidireccional. Los datos se transmiten desde el microcontrolador por un pin y se reciben por otro. Cada conjunto de datos transmitidos o recibidos por el SCI está formado por un bit de inicio (start), varios bits de datos y un bit de parada (stop). El bit de inicio (start) y el bit de parada (stop), se usan para sincronizar los dos dispositivos que se están comunicando. Este tipo de interfaz serie se usa a menudo cuando un microcontrolador se debe comunicar a distancias moderadas. Para aumentar estas distancias se pueden utilizar convertidores de nivel de tensión (tipo MAX232) conectados a los pines de transmisión y de recepción. El SCI se puede utilizar para comunicarse con ordenadores personales o con otros sistemas con microcontrolador con el estándar RS232 u otras formas de codificación de la información.

  Esta interfaz dispone de las siguientes funcionalidades:

  - Funcionamiento en modo full-duplex sin restricción.
  - Doble búfer para recibir y transmitir.
  - Longitud del dato enviado programable a 8 o 9 bits, autorizando así la emisión del bit de paridad si es necesario.
  - Velocidad de transmisión programable mediante software, sobre 32 valores normalizados.
  - Indicador de error de recepción análogo al permitido por las UARTs: error de formato, error de desbordamiento, error debido al ruido.
  - Modo de activación automático durante la recepción de señales válidas.

- Funcionamiento basado en interrupciones, existiendo diferentes condiciones de generación de interrupciones.

- SPI (*Serial Peripheral Interface*): Ésta es una interfaz serie para la comunicación del microcontrolador principalmente con periféricos externos que se encuentran a poca distancia. El SPI permite una comunicación síncrona con otros dispositivos, transfiriendo los datos de forma bidireccional. Para establecer su comunicación requiere tres pines del microcontrolador: además de dos pines, uno para transmitir y otro para recibir, el tercer pin es el encargado de proporcionar la señal de sincronización también denominada reloj. Este tipo de interfaz serie, normalmente se usa para comunicar dispositivos periféricos en la misma placa de circuito impreso a alta velocidad. Estos suelen ser periféricos tales como un simple registro de desplazamiento, memoria EEPROM, un sistema LCD o un conversor A/D.

  Las características de esta interfaz son:

  - Comunicación full-duplex, sincronía mediante 3 líneas. Aunque en la práctica es semi-duplex debido al protocolo de comunicación que utiliza el SPI.
  - Operación en modo maestro o esclavo.
  - Indicador de interrupción al finalizar la transmisión.
  - Frecuencia máxima del maestro de 1 MHz y de 2 MHz el esclavo.

- SIOP (*Serial Input/Output Port*): Esta interfaz es un puerto serie de entrada/salida síncrono. Es muy parecido al SPI, pero está destinado, principalmente, a aplicaciones que no tienen más de dos circuitos para intercambiar datos: un maestro y un esclavo. También dispone al igual que el SPI de 3 líneas para realizar la comunicación y permite decidir cuál de los dispositivos actúa como maestro.

2.1.3.5. TEMPORIZADORES

Aunque existe una amplia variedad de temporizadores en la familia 68HC05, sus funciones básicas están relacionadas con la medida o generación de eventos basados en el tiempo. Los temporizadores normalmente miden tiempo relativo al reloj interno del microcontrolador, aunque algunos pueden tener una entrada de reloj externa.

Los temporizadores que se pueden encontrar en los diferentes miembros de la familia son:

- TEMPORIZADOR DE 16 BITS: Las características de este temporizador son las siguientes:
    - Contador autónomo de 16 bits con preescalado.
    - Indicador de desbordamiento para aumentar el rango de conteo.
    - Registro de 16 bits de comparación a la salida.
    - Registro de 16 bits de captura de entrada.
    - 3 fuentes de interrupción.

  Este temporizador está basado en un contador automático de 16 bits precedido por un preescalado que divide el reloj interno. El tiempo es representado por la cuenta de este contador y puede ser leído en cualquier momento, además de proporcionar una base de tiempos para sus dos modos de funcionamiento: modo captura y modo comparación.

  El modo captura utiliza el registro de 16 bits del temporizador como entrada. Cuando la entrada correspondiente del microcontrolador recibe una transición válida, el valor que contiene en ese instaste el contador de 16 bits se transfiere al registro de captura. Además se activa un indicador de desbordamiento y se puede generar una interrupción. Este modo se puede usar para la medida del periodo de una señal de entrada, para la medida del ancho de un pulso de entrada o como una interrupción adicional.

  En el modo comparación se utiliza el registro de comparación del temporizador como salida. Este registro se carga con el valor a comparar y cuando el contador lo alcanza, una determinada salida del microcontrolador toma el valor de un bit interno que previamente ha sido programado. También se activa en este caso un indicador que puede generar una

interrupción. Este modo se utiliza en aplicaciones que requieran una interrupción periódica, una señal de salida de frecuencia variable, una señal de salida modulada en ancho de pulso o para obtener un pulso con un determinado ancho a la salida.

- MTF (*Multi Function Timer*): se encuentra en algunas versiones del 68HC05 en lugar del temporizador anterior o además de este. Este temporizador proporciona una referencia de tiempo con la capacidad de poder programar interrupciones en tiempo real. Dispone de un registro legible de 8 bits (TCR), donde se almacena la cuenta que se va realizando, que no puede ser alterado por la CPU. Cuando este registro se desborda, puede provocar una interrupción si ésta ha sido previamente habilitada. En unión con este registro, existe otro de 7 bits que se activa con el desbordamiento del anterior, y que también puede provocar otra interrupción si se desea.

- WATCHDOG: Este temporizador, también llamado COP (*Computer Operating Properly*), sirve como su nombre indica para proteger contra fallos de software. En las versiones del 68HC05 que no tienen temporizador multifunción suele ser independiente, mientras que en las que si lo tienen formar parte de él.

  El funcionamiento de este temporizador consiste en que el programador debe reactivarlo regularmente antes de que se consuma un determinado tiempo para evitar que provoque un reset del microcontrolador. Esto se realiza mediante instrucciones del programa, de manera que, si la secuencia del programa no está siendo ejecutada correctamente, llegará un momento en el que se produzca el reset.

  El COP tiene la ventaja de que el periodo de reactivación puede ser modificado, y depende, tanto de la frecuencia del reloj, como del valor que se le dé a dos bits de su registro de control.

2.1.3.6. CONVERTIDOR ANALÓGICO DIGITAL

Todas las señales que existen en el mundo real no son directamente compatibles con los pines de E/S del microcontrolador. Algunos miembros de la familia 68HC05 incluyen un convertidor analógico/digital (A/D). Este convertidor realiza un muestreo de señales analógicas externas y produce periódicamente los valores digitales correspondientes. Los usos típicos suelen ser la medida de entradas analógicas como el voltaje de una batería, la temperatura, la presión, o el nivel de un líquido.

Este convertidor es un modelo de 8 bits por aproximaciones lineales sucesivas. Las entradas al convertidor son comunes con un puerto paralelo al igual que la referencia de tensión que utiliza. Posee un indicador que señala cuándo se ha alcanzado el final de la conversión, que puede ser utilizado como fuente de una interrupción. Este convertidor también se compone de un multiplexor analógico que permite variar el número de canales a convertir, ampliando así sus posibilidades.

2.1.3.7. CONVERTIDOR DIGITAL ANALÓGICO O PUERTO PWM (*Pulse Widht Modulation*)

Ciertas versiones del 68HC05 disponen de un puerto que se puede calificar de convertidor digital/analógico elemental o, más exactamente, de puerto PWM (*Pulse Widht Modulation*) o modulador de la duración de impulsos. Estos módulos funcionan en conjunción con el temporizador, utilizando su registro de recuento. Posee dos registros de 8 bits asociados con él, los cuáles a su vez están asociados con dos salidas del microcontrolador.

La principal aplicación de este modulador es generar una señal periódica cuyo ciclo de trabajo es programable. Este puede seleccionarse por pasos de 1/256 entre 0 y 1 al ser los registros de 8 bits, es decir, si el registro tiene por ejemplo el dato $00, la señal de salida será toda a nivel bajo (ciclo de trabajo nulo), y si tiene el valor $80, la salida tendrá un ciclo de trabajo del 50%. La frecuencia de funcionamiento de este modulador puede variar de 122 Hz a 1953 Hz, para un microcontrolador que trabaja con un cristal de cuarzo a 4 MHz.

2.1.4. CAPACIDAD DE MEMORIA DEL MICROCONTROLADOR

Los microcontroladores utilizan varios tipos de información que requiere almacenarla en diferentes clases de memoria. Las instrucciones que controlan el funcionamiento de los microcontroladores se guardan en una memoria no-volátil para que el sistema no tenga que ser reprogramado cada vez que pierdan la alimentación. Por el contrario, para trabajar con las variables y los resultados intermedios, es necesario que sean guardados en una memoria que pueda escribirse rápida y fácilmente durante el funcionamiento del sistema. No es importante conservar este tipo de información cuando no hay alimentación, por lo que puede usarse una memoria volátil.

Dentro de los diferentes tipos de memoria, en los componentes de la familia 68HC05 podemos encontrar los siguientes:

- Memoria RAM (*Random Access Memory*): es una memoria volátil que puede ser leída o escrita por la CPU. Aunque cuando apareció esta memoria su ventaja más importante era que disponía de acceso aleatorio, es decir, que se puede llegar a cualquiera de sus posiciones sin tener en cuenta el orden, hoy en día lo que prima es la facilidad y la rapidez que le proporciona a la CPU en los procesos de lectura y escritura. Por sus características se usa para el almacenamiento temporal de datos y el cálculo intermedio de los resultados durante las operaciones.
- Memoria ROM (*Read Only Memory*): Es una memoria de sólo lectura. En este tipo de memoria es imposible cambiar su contenido después que haya sido programada en fábrica, por lo tanto, no volátil. Se utiliza principalmente para guardar las instrucciones del programa y los datos cuyo valor es permanente durante toda la ejecución del programa como es el caso de las constantes.
- Memoria PROM (*Programmable Read Only Memory*): Es muy similar a la ROM con la diferencia de que puede ser programada después de fabricar el circuito integrado. Dentro de este tipo de memoria programable podemos distinguir:
    - Memoria OTP (*One Time Programable*): Esta memoria se diferencia de las demás en que en vez de venir programada de fábrica se puede programar eléctricamente, pero sólo una vez. Es similar a la

EPROM pero con un encapsulado opaco. Se utiliza también para almacenar las instrucciones de programa y datos invariantes.

- Memoria EPROM (*Erasable Programmable Read Only Memory*): El proceso de borrado se realiza mediante luz ultravioleta a través de una ventana en la parte superior del dispositivo. Esta memoria por lo general sólo se utiliza para almacenar las instrucciones de programa y datos de contenido invariable.
- Memoria EEPROM (*Electrically Erasable Programmable Read Only Memory*): este tipo de PROM tiene la particularidad de que se puede borrar eléctricamente mediante algunos comandos del microcontrolador. Esta memoria normalmente se utiliza como complemento de la memoria EPROM para guardar los parámetros de configuración de una determinada aplicación.

Dependiendo del miembro de la familia que se trate, el tamaño de cada tipo de memoria variará, teniendo en cuenta de que determinados tipos de memoria no se encuentran en todos los dispositivos. Esto ocurre en el caso de las memorias PROM, en las cuáles, dependiendo del componente del que se trate tendrá memoria EPROM, OTPROM ó EEPROM. Además, existe un registro mapeado en la memoria que permite la elección de la configuración de la memoria para determinadas zonas con la programación del valor de dos de sus bits, dando a elegir entre memoria RAM o PROM.

Un ejemplo del mapa de memoria de un miembro de la familia de estos microcontroladores se muestra en la Figura 2.3. En este caso es el mapa de memoria de la versión 68HC705C8 que posee 8 Kbytes de memoria en total distribuidos entre memoria RAM, ROM y PROM. La memoria ROM en este caso ocupa 240 bytes, mientras que el resto se distribuye entre la memoria RAM (que puede llegar a tener de 176 a 304 bytes) y la memoria PROM (de 7600 a 7744 bytes)

En la Figura 2.3 se pueden observar todos estos tipos de memoria y las zonas de memoria en las que es posible decidir la clase de memoria que se adecua más a las necesidades de cada aplicación. La decisión del tamaño de estas dos clases contempla

varias posibilidades dependiendo del valor que tomen los bits RAM0 y RAM1 que se encuentran situados dentro de un registro en la posición $1FDF.

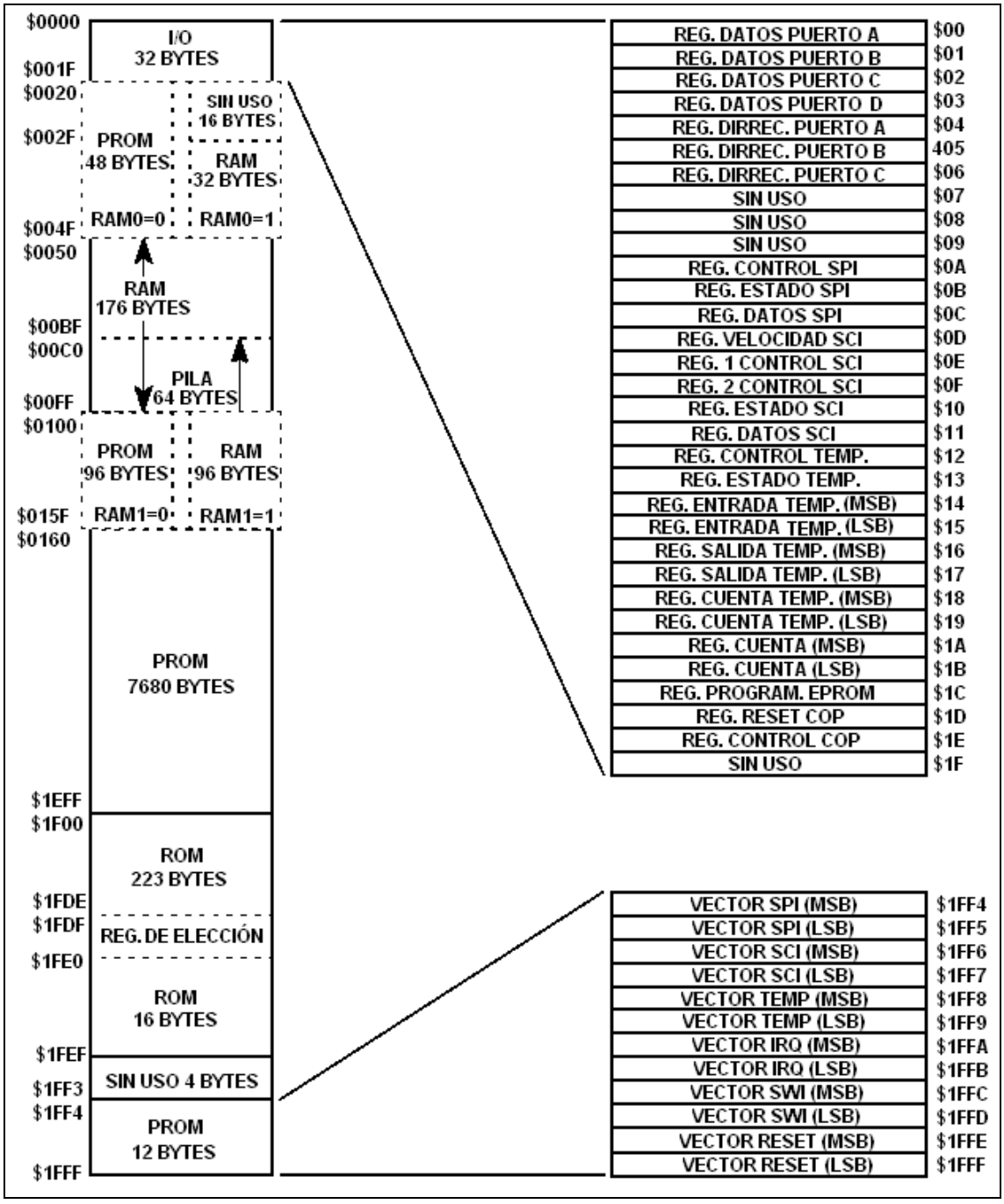


Figura 2.3 Típico mapa de memoria.

2.2. LA FAMILIA 68HC05

El microcontrolador 68HC05 es un dispositivo fabricado por Motorola que entra dentro de la familia de microcontroladores de 8 bits.

Como antecesor de esta familia, se encuentra la familia 6805 que utilizaba la tecnología HMOS y se caracteriza por un consumo de energía relativamente grande y una alimentación única de 5 voltios.

Una tecnología más moderna es la formada por la familia 68HC05, es decir, la tecnología HCMOS, lo cual permite a estos microcontroladores que sean mucho más rápidos, consuman poca energía, sean totalmente estáticos y puedan alimentarse con una tensión comprendida entre 3 y 6 voltios. Dentro de esta familia existen diversas versiones de circuitos disponibles, las cuáles se diferencian por las letras o números que componen la referencia.

- Atendiendo al tipo de memoria de programa en lugar de la ROM:
    - 68HC705. Este grupo tiene memoria del tipo EPROM.
    - 68HC805. Este en cambio tiene memoria EEPROM.
- Atendiendo a otras características como son el número de puertos E/S, puertos serie, temporizadores, conversores A/D, étc, podemos encontrar una amplia variedad de dispositivos los cuáles se distinguen por letras añadidas al final de la referencia. De todos ellos, los más importantes se encuentran enumerados en la Tabla 2.1 junto con todas estas características que los distinguen entre sí. En realidad existen muchísimos miembros más pertenecientes a la familia 68HC05 que los que se adjuntan en esta tabla, pero debido a esta amplia variedad, la tabla sólo contempla los que hoy en día ofrece el fabricante Motorota en su página web “http:// www.motorola.com”.

  Todos estos dispositivos, a su vez, se agrupan en conjuntos los cuáles su fabricante, Motorola, también les da el nombre de familias. Por esta razón se ha realizado una tabla (la Tabla 2.2) donde se detallan las características principales de cada una de ellas y sus aplicaciones más comunes en cada una de las diferentes áreas de la industria como son la automoción, las telecomunicaciones, y la informática. Las aplicaciones que no se puedan clasificar dentro de ninguna de las áreas anteriores se nombran como aplicaciones de la industria en general.

Tabla 2.1 Características de las diferentes versiones del 68HC05.

REFERENCIA	RAM (Byte)	ROM (Byte)	EEPROM (Byte)	EPROM (Byte)	OTPROM (Byte)	INTERFAZ SERIE	TEMPORIZADORES	CONVERSOR A/D		ALIMENTACIÓN (Typ), (V)	PWM		I/O PINES
							CANALES	CANALES	BITS		CANALES	BITS	
68HC05B6	176	5936	256	---	---	SCI	4	8	8	3.3, 5	2	8	32
68HC05B8	176	7424	256	---	---	SCI	4	8	8	3.3, 5	2	8	32
68HC05B16	352	15360	256	---	---	SCI	4	8	8	3.3, 5	2	8	32
68HC705B16	352	---	256	15360	15360	SCI	4	8	8	3.3, 5	2	8	32
68HC05B32	528	32768	256	---	---	SCI	4	8	8	3.3, 5	2	8	32
68HC705B32	528	---	256	32768	---	SCI	4	8	8	3.3, 5	2	8	32
68HC05C8A	176	7744	---	7744, 8192	---	SCI, SPI	2	---	---	3.3, 5	---	---	31
68HC705C8A	304	---	---	8192	8092	SCI, SPI	2	---	---	3.3, 5	---	---	31
68HC05C9A	352	15936	---	---	---	SCI, SPI	2	---	---	3.3, 5	---	---	31
68HC705C9A	352	---	---	15932	15932	SCI, SPI	2	---	---	3.3, 5	---	---	31
68HC705F32	920	---	256	32768	---	SCI, SPI	8	8	8	2.7, 5	3	8	69, 80
68HC05J1A	64	1240	---	---	---	---	---	---	---	3.3, 5	---	---	14
68HC705J1A	64	---	---	---	1240	---	---	---	---	3.3, 5	---	---	14
68HC05J5A	128	2560	---	---	---	---	1	---	---	2.2, 5	---	---	14
68HC705J5A	128	---	---	2560	2560	---	1	---	---	2.2, 5	---	---	14
68HC05JB3	144	2560	---	---	---	USB 1.0	2	---	---	5	---	---	19
68HC705JB3	144	---	---	2560	---	USB 1.0	2	---	---	5	---	---	19
68HC05JJ6	224	6160	---	---	---	SIOP	2	4	12	3, 5	---	---	14
68HC705JJ7	224	---	---	6160	---	SIOP	2	4	12	3, 5	---	---	14
68HC05JP6	224	6160	---	---	---	SIOP	2	4	12	3, 5	---	---	22
68HC705JP7	224	---	---	6160	---	SIOP	2	4	12	3, 3.3, 5	---	---	22
68HC05K3	64	928	16	---	---	---	---	---	---	3, 5	---	---	10
68HC05L16	512	16400	---	---	---	SSPI	2	---	---	2.7, 3.3, 5	---	---	39
68HC705L16	512	---	---	16400	---	SSPI	2	---	---	2.7, 3.3, 5	---	---	39
68HC05L25	176	6160	---	---	---	SPI	---	2	8	3.3, 5	---	---	20

68HC05LJ5	64	1296	---	---	---	---	---	---	---	3.3, 5	---	---	14
68HC05P18A	192	8064	128	---	---	SIOP	2	4	8	5	---	---	21
68HC05P4A	176	4160	---	---	---	SIOP	2	---	---	3.3, 5	---	---	21
68HC05P6	176	4672	---	---	---	SIOP	2	4	8	3.3, 5	---	---	21
68HC705P6A	176	---	---	4672	---	SIOP	2	4	8	3.3, 5	---	---	21
68HC05PV8A	192	7936	7936	---	---	---	4	6	8	5	1	8	20
68HC805PV8	192	---	7936	---	---	---	4	6	8	5	1	8	20
68HC705SR3	192	---	---	3840	---	---	---	4	8	3, 5	---	---	32
68HC05SU3A	192	3840	---	---	---	---	---	---	---	5	---	---	32
68HC705X4	176	4096	---	4096	---	MCAN	2	---	---	5	---	---	16
68HC05X16	352	15102	256	---	---	MCAN, SCI	4	8	8	5	2	8	32
68HC05X32	528	31232	256	---	---	MCAN, SCI	4	8	8	5	2	8	32
68HC705X32	528	---	256	31232	---	MCAN, SCI	4	8	8	5	2	8	32

Tabla 2.2 Aplicaciones de las diferentes familias del 68HC05.

FAMILIA	CARACTERÍSTICAS PRINCIPALES	AUTOMOCIÓN	TELECOMUNICACIÓN	INFORMÁTICA	INDUSTRIA
B	-Memoria EEPROM -Abundantes puertos E/S -Convertidor A/D -Moduladores PWM -Puerto Serie SCI	-Control de la posición de los asientos. -Sistemas de audio con memoria de emisoras.	En equipos portátiles: -para el almacenamiento de números de teléfono, -y en la medida de la cantidad de batería disponible.	-------	-Sistemas de adquisición de datos. -Contadores Lógicos Programables (PLC).
C	-Amplia variedad de tipos de memoria. -Puerto Serie SCI y SPI. -Contador de 16 bits. -Watchdog.	Control de los sistemas de: -navegación, -encendido, -y entretenimiento.	-Teléfonos y contestadores automáticos.	-Control de teclados y de monitores.	-Sistemas de control de procesos donde se requieren múltiples líneas de E/S y salidas a dispositivos LED. -Reproductores de CD´s. -Mandos a distancia.
F	-Generador multifrecuencia.	-------	Dispositivos que permiten: -el sintonizado automático, -el almacenamiento de números, -y el control de displays.	-------	-------

J	-Desde 16 hasta 20 pines de E/S. -Baratos. -CPU potente. -Temporizador multifunción de 15 etapas. -Contador de 8 bits.	-Alarmas de coches. -Elevalunas. -Apertura automática de puertas. -Airbags.	-------	-Ratones inalámbricos.	-Detectores de humos. -Sistemas de seguridad. -Termostatos. -Lavadoras. -Mandos a distancia. -Sistemas de encendido de hornos.
JB	-Interfaz USB.	-------	-------	-Ratones.	-------
JJ y JP	-Puerto Serie SIOP. -Convertidor A/D con una resolución de hasta 12 bits.	-------	-------	-------	Se utilizan sobre todo en este campo.
K	- Buena relación calidad-precio.	-------	-------	-------	Control de diversos productos como: -lavadoras, -hornos, -y mandos a distancia.
L	-Pequeño tamaño. -Bajo consumo. -Temporizador de 16 bits. -Watchdog. -Puertos para displays LCD.	-------	Equipos portátiles como: -móviles, -y agendas electrónicas.	-------	Equipos que requieren una salida audible y necesitan displays: -alarmas, -termostatos, -y reproductores de CD’s.

P	-28 pines de E/S. -Puerto Serie SIOP. -Convertidor A/D.	-------	-------	--------	Muy útil en el manejo de displays y periféricos.
PV	-Mempria EEPROM. -Convertidor A/D. -Generadores de PWM.	Se utiliza sobre todo en este campo en una amplia gama de funciones.	-------	-------	-------
X	-Interfaz Serie CAN (*Controller Area Network*) que le permite comunicarse fácilmente con otros módulos electrónicos.	Es muy utlizada en este campo.	-------	-------	-Automatización de fábricas. -Aplicaciones donde es necesario el uso de sensores.

CAPÍTULO 3

DESCRIPCIÓN DE LA UNIDAD CENTRAL DE PROCESO

3. DESCRIPCIÓN DE LA UNIDAD CENTRAL DE PROCESO

El núcleo del microcontrolador es lo que comúnmente se entiende como la unidad central de proceso o CPU (*Central Process Unit*). Las funciones que debe realizar son las de coordinar, controlar y ejecutar todas las instrucciones del programa que tiene almacenado en la memoria. Para ello, extrae las instrucciones de la memoria, las analiza, y emite las órdenes pertinentes para desencadenar su ejecución a la vez que controla a todas las unidades implicadas en este proceso.

En el presente capítulo se ofrece la descripción detallada de todo el *hardware* que compone el núcleo del microcontrolador además de la metodología utilizada en el diseño del mismo. Se van a explicar los registros que contiene la CPU, los registros de acceso a memoria, el registro de instrucción, la unidad aritmético lógica, los buses (de control, de direcciones y de datos) y la unidad de control.

Para realizar esta descripción se requiere conocer, en un principio, los diferentes modos de direccionamiento que utiliza este dispositivo, así como las interrupciones que puede recibir de los periféricos conectados a él o del exterior, ya que se hacen necesarias a la hora de plantear la implementación de las diferentes partes del núcleo y de explicar la arquitectura de los registros que lo componen. Por lo tanto, antes de comenzar a describir el *hardware* del núcleo, lo primero que se va a comentar son estas dos características de programación de que dispone este microcontrolador.

Una vez hecho esto, se comienza a describir los diferentes módulos del *hardware* que forman el núcleo, pero diferenciando las dos partes importantes a la hora de realizar el diseño: por un lado la ruta de datos y por el otro la unidad de control. Dentro de la ruta de datos se describen los diferentes registros que componen el núcleo del microcontrolador, la unidad aritmético lógica y los buses de datos, de direcciones y de control. En el apartado de la unidad de control se describe un módulo que hace las funciones de decodificador de las instrucciones además de controlar todos los registros de la ruta de datos.

Por último hay que destacar que en este capítulo a la vez de realizar la descripción de los elementos que forman parte del núcleo en cuanto a su arquitectura y a sus funciones, se entra ya en detalle y se explica la metodología de diseño e implementación en el lenguaje VHDL que se ha realizado para cada uno de ellos. Esta

descripción incluye una figura donde se pueden observar sus pines de E/S además de la enumeración y explicación de cada una de las señales utilizadas en su implementación.

3.1. MODOS DE DIRECCIONAMIENTO

La potencia de un microcontrolador reside, en gran medida, en la capacidad de su acceso a memoria. Esta capacidad viene definida por los modos de direccionamiento. Estos especifican la manera, por la cuál, una instrucción obtiene el operando requerido para su ejecución. Debido a los diferentes modos de direccionamiento, una instrucción puede acceder al operando que necesita según una de las distintas maneras que estos le indican.

En el caso del microcontrolador 68HC05 se utilizan 6 modos de direccionamiento: inherente, inmediato, directo, extendido, indexado (sin *offset*, con *offset* de 8 bits y con *offset* de 16 bits) y relativo. Estos permiten al juego de instrucciones del microcontrolador expandir sus 62 operaciones básicas a 210 códigos de operación distintos.

Cada uno de los diferentes modos de direccionamiento se explican más detenidamente en los apartados que vienen a continuación además de que, una vez finalizados estos, se adjunta una tabla general (Tabla 3.1), donde se pueden observar todo el conjunto de instrucciones por orden alfabético y cuáles son los modos de direccionamiento que se pueden utilizar en cada una de ellas. Cada modo de direccionamiento viene indicado por el código de operación que se utiliza en la correspondiente instrucción.

3.1.1. DIRECCIONAMIENTO INHERENTE

En este modo de direccionamiento todo la información requerida para las operaciones es conocida, como su propio nombre indica, inherentemente por la CPU y, no necesita de ningún operando que se encuentre en la memoria o en el programa. En el caso de que la instrucción requiera alguno, estos operandos son los propios registros de la CPU como el acumulador o el registro de indexado y vienen determinados según el código de operación de la instrucción.

3.1.2. DIRECCIONAMIENTO INMEDIATO

En este modo de direccionamiento el operando está almacenado en el byte siguiente al código de instrucción. Este modo se utiliza para almacenar un valor o una constante la cuál es conocida en el momento de escribir el programa y no cambia durante la ejecución del mismo.

Las instrucciones con este modo de direccionamiento ocupan dos bytes, uno para el código de instrucción y otro para el operando.

3.1.3. DIRECCIONAMIENTO DIRECTO

Este modo de direccionamiento, la **dirección** del operando se encuentra almacenada en el byte siguiente al código de operación. Debido a que el bus de direcciones maneja datos cuya longitud es mayor de 8 bits, es necesario completar los restantes bits, ya que al leer un dato de la memoria sólo se pueden obtener los 8 bits menos significativos. Por ello, el valor del byte más significativo de la dirección donde se va a buscar el operando de la instrucción que se encuentra en ese momento en ejecución, es siempre $00. De esta forma solamente el byte menos significativo es necesario para realizar la instrucción y por ello este modo de direccionamiento permite direccionar exclusivamente los 256 bytes más bajos de la memoria. Esta zona de la memoria incluye los registros de E/S y la memoria RAM. Este modo resulta muy eficiente en el tiempo de acceso a la memoria que invierte la CPU.

Las instrucciones que manejan este modo de direccionamiento son de dos bytes, el primero contiene el código de operación y el segundo la parte baja de la dirección donde se encuentra el operando.

3.1.4. DIRECCIONAMIENTO EXTENDIDO

En este modo de direccionamiento, la dirección del operando se encuentra almacenada en los dos bytes siguientes al código de instrucción. Con este modo de direccionamiento es posible referenciar cualquier posición de memoria dentro del espacio de memoria del microcontrolador incluyendo la memoria RAM, ROM, PROM y las E/S que se encuentran mapeadas en ella.

Este modo utiliza tres bytes, el primero es el código de operación, el siguiente, el byte más significativo de la dirección del operando y el último, es el menos significativo.

3.1.5. DIRECCIONAMIENTO INDEXADO

En este modo de direccionamiento la dirección efectiva es variable dependiendo sobre todo de dos factores: del contenido que en ese momento posea el registro de indexado (registro de 8 bits que forma parte de la CPU) y del *offset* que presentan los bytes siguientes al código de operación de la instrucción que se encuentra en ejecución.

En los microcontroladores de la familia 68HC05, este modo de direccionamiento puede ser de varios tipos: sin *offset*, con un *offset* de 8 bits, o con uno de 16 bits. Un buen código ensamblador utiliza el direccionamiento indexado que menos número de bytes requiere para expresar el *offset*.

- SIN OFFSET: en este modo de direccionamiento la dirección efectiva la proporciona el byte que almacena en ese momento el registro de indexado, por lo tanto, sólo puede acceder a las 256 primeras posiciones de la memoria. La instrucciones que lo utilizan ocupan un solo byte.
- OFFSET DE 8 BITS: en este caso la dirección efectiva se obtiene mediante la suma del contenido del byte siguiente al código de instrucción más el valor que en ese momento contenga el registro de indexado. Este modo de direccionamiento se utiliza normalmente para seleccionar un elemento de una tabla. Para poder usarlo, la tabla debe estar contenida en las primeras 256 posiciones de la memoria y podrá extenderse hasta la posición número 511. Este direccionamiento se puede utilizar tanto para leer memoria RAM, ROM, y registros de E/S mapeados en memoria, al ser estos los que podemos encontrar dentro de las posiciones antes nombradas.

  La instrucción que utiliza este modo de direccionamiento tiene que ser de dos bytes, el primero contiene el código de instrucción mientras que el segundo es el *offset* deseado, que se representa mediante 8 bits y es un número entero sin signo.
- OFFSET DE 16 BITS: en este caso la dirección efectiva se forma mediante la suma del contenido que presenta el registro de indexado durante la

ejecución de la instrucción actual, más el *offset* de 16 bits que forman los dos bytes siguientes al código de operación. Por lo tanto, la instrucción está formada por tres bytes: el primero es el código de operación, el siguiente contiene el byte más significativo del *offset* y el último, es el byte menos significativo.

Este modo de direccionamiento al igual que el anterior, se suele utilizar para acceder a los datos de una tabla, pero con la particularidad de que al tener un *offset* de 16 bits, la tabla puede estar contenida en cualquier zona de la memoria.

3.1.6. DIRECCIONAMIENTO RELATIVO

Este modo de direccionamiento sólo se utiliza en el microcontrolador 68HC05 para las bifurcaciones condicionales. Estas instrucciones y otras versiones de las mismas como son las que tratan a su vez la manipulación de bits, se componen de dos bytes, el primero es el código de operación y el siguiente es el desplazamiento (*offset*) relativo respecto al valor del contador de programa. El salto puede producirse en cualquier dirección, por lo que el byte de desplazamiento se representa mediante un número de 8 bits en complemento a dos. Esto proporciona un rango de salto de entre –128 y +127 posiciones de memoria con respecto a la dirección de la instrucción siguiente a la instrucción condicional.

El funcionamiento es el siguiente, si la condición es verdadera, el valor que tenga en ese momento el contador de programa, se suma al contenido del byte que sigue al código de instrucción para crear la dirección efectiva. En esta operación hay que tener en cuenta que el valor del segundo sumando tiene signo y que la operación puede ser una suma o una resta dependiendo del mismo. En el caso de que la condición sea falsa, el control pasa a la instrucción que se encuentra inmediatamente después de la instrucción condicional.

A la hora de realizar la programación en ensamblador, la forma que tiene el programador de decir a que parte del programa desea saltar, es o bien dar el valor de una dirección absoluta, o bien poner una etiqueta (la cuál se refiere a una dirección

absoluta), y es el ensamblador el que se encarga de calcular cuál es el *desplazamiento* necesario para llegar hasta esa etiqueta.

A continuación, en la Tabla 3.1, se pueden observar las instrucciones que utilizan este modo de direccionamiento junto con todas las demás señalando sus respectivos modos de direccionamiento.

Tabla 3.1 Instrucciones y sus modos de direccionamiento.

INSTRUCCIÓN	INHERENTE	INMEDIATO	DIRECTO	EXTENDIDO	INDEXADO			RELATIVO
					SIN OFFSET	OFFSET 8 BITS	OFFSET 16 BITS	
ADC	-	A9	B9	C9	F9	E9	D9	-
ADD	-	AB	BB	CB	FB	EB	DB	-
AND	-	A4	B4	C4	F4	E4	D4	-
ASL	48 (A) 58 (X)	-	38	-	78	68	-	-
ASR	47 (A) 57 (X)	-	37	-	77	67	-	-
BCC	-	-	-	-	-	-	-	24
BCLR n	-	-	11 13 15 17 19 1B 1D 1F	-	-	-	-	-
BCS	-	-	-	-	-	-	-	25
BEQ	-	-	-	-	-	-	-	27
BHCC	-	-	-	-	-	-	-	28
BHCS	-	-	-	-	-	-	-	29
BHI	-	-	-	-	-	-	-	22
BHS	-	-	-	-	-	-	-	24
BIH	-	-	-	-	-	-	-	2F
BIL	-	-	-	-	-	-	-	2E
BIT	-	A5	B5	C5	F5	E5	D5	-
BLO	-	-	-	-	-	-	-	25
BLS	-	-	-	-	-	-	-	23
BMC	-	-	-	-	-	-	-	2C
BMI	-	-	-	-	-	-	-	2B

BMS	-	-	-	-	-	-	-	2D
BNE	-	-	-	-	-	-	-	26
BPL	-	-	-	-	-	-	-	2A
BRA	-	-	-	-	-	-	-	20
BRCLR n	-	-	-	-	-	-	-	01 03 05 07 09 0B 0D 0F
BRN	-	-	-	-	-	-	-	21
BRSET n	-	-	-	-	-	-	-	00 02 04 06 08 0A 0C 0E
BSET n	-	-	10 12 14 16 18 1A 1C 1E	-	-	-	-	-
BSR	-	-	-	-	-	-	-	AD
CLC	98	-	-	-	-	-	-	-
CLI	9A	-	-	-	-	-	-	-
CLR	4F (A) 5F (X)	-	3F	-	7F	6F	-	-
CMP	-	A1	B1	C1	F1	E1	D1	-
COM	43 (A) 53 (X)	-	33	-	73	63	-	-
CPX	-	A3	B3	C3	F3	E3	D3	-
DEC	4A (A) 5A (X)	-	3A	-	7A	6A	-	-
EOR	-	A8	B8	C8	F8	E8	D8	-
INC	4C (A) 5C (X)	-	3C	-	7C	6C	-	-

JMP	-	-	BC	CC	FC	EC	DC	-
JSR	-	-	BD	CD	FD	ED	DD	-
LDA	-	A6	B6	C6	F6	E6	D6	-
LDX	-	AE	BE	CE	FE	EE	DE	-
LSL	48 (A) 58 (X)	-	38	-	78	68	-	-
LSR	44 (A) 54 (X)	-	34	-	74	64	-	-
MUL	42	-	-	-	-	-	-	-
NEG	40 (A) 50 (X)	-	30	-	70	60	-	-
NOP	9D	-	-	-	-	-	-	-
ORA	-	AA	BA	CA	FA	EA	DA	-
ROL	49 (A) 59 (X)	-	39	-	79	69	-	-
ROR	46 (A) 56 (X)	-	36	-	76	66	-	-
RSP	9C	-	-	-	-	-	-	-
RTI	80	-	-	-	-	-	-	-
RTS	81	-	-	-	-	-	-	-
SBC	-	A2	B2	C2	F2	E2	D2	-
SEC	99	-	-	-	-	-	-	-
SEI	9B	-	-	-	-	-	-	-
STA	-	-	B7	C7	F7	E7	D7	-
STOP	8E	-	-	-	-	-	-	-
STX	-	-	BF	CF	FF	EF	DF	-
SUB	-	A0	B0	C0	F0	E0	D0	-
SWI	83	-	-	-	-	-	-	-
TAX	97	-	-	-	-	-	-	-
TST	4D (A) 5D (X)	-	3D	-	7D	6D	-	-
TXA	9F	-	-	-	-	-	-	-
WAIT	8F	-	-	-	-	-	-	-

3.2. INTERRUPCIONES DEL MICROCONTROLADOR

Las interrupciones son sucesos asíncronos externos al núcleo del microcontrolador que hacen que este interrumpa la ejecución normal del flujo de programa y dedique todos sus recursos hacia ellos. Estas suelen venir provocadas por cualquiera de las siguientes fuentes:

- La primera de ellas son los periféricos internos del microcontrolador. La mayoría de ellos no están dotados de inteligencia propia y requieren ayuda por parte de la CPU. Para impedir que esta dedique una atención constante hacia ellos, los periféricos la solicitan a menudo en forma de interrupciones.
- La segunda fuente son los sucesos externos al microcontrolador. Muchos procesos requieren a menudo que su funcionamiento normal sea interrumpido si algún evento externo necesita ser atendido.
- La tercera de las fuentes de interrupción son los eventos internos de la CPU como por ejemplo si tiene lugar alguna instrucción ilegal.
- La última fuente es el programador. Dentro de las instrucciones del programa se puede solicitar que en un momento determinado se atienda cierto proceso. Para ello se provoca una interrupción mediante el *software*.

Dentro de todas las posibles fuentes de interrupción que pueden existir entre los miembros de la familia 68HC05, sólo se van a considerar las siguientes en el diseño y la implementación del núcleo:

INTERRUPCIONES HARDWARE: pueden tener lugar hasta 6 fuentes de interrupción *hardware* de tipo enmascarable. Estas fuentes son las siguientes:

- la interrupción externa ($\overline{\text{IRQ}}$),
- la interrupción del puerto serie síncrono (SPI),
- la del puerto serie asíncrono (SCI),
- la interrupción del temporizador,
- la interrupción del conversor A/D y
- la interrupción del generador PWM.

Al ser interrupciones de tipo enmascarables, sólo son atendidas si previamente ha sido deshabilitada la máscara de interrupción (bit I del registro de estado).

Dentro de las interrupciones *hardware* también se va a incluir el *reset*. Esta interrupción es de tipo no enmascarable y por ello, es atendida en todo momento.

Todas las interrupciones *hardware*, a excepción del $\overline{\text{IRQ}}$ y del *reset*, son generadas por los periféricos que acompañan al núcleo pero que no han sido implementados en este proyecto. Estos poseen varios *flags* que pueden causar una interrupción y que generalmente, están localizados en registros de estado de sólo lectura. Los bits que los permiten actuar se encuentran en los registros de control asociados a ellos pero nunca en los mismos registros. Si el bit que habilita la interrupción está al valor lógico '0', se bloquea la interrupción y no se permite que esta ocurra. Pero esto, es independiente del *flag* que señala que ha tenido lugar una interrupción, que por el contrario, si se activa.

En este proyecto la manera de tener en cuenta todas las interrupciones generadas por los periféricos, que se puedan añadir al núcleo en posteriores trabajos, ha sido considerándolas como entradas al diseño al igual que en el caso de la interrupción IRQ y del *reset*.

INTERRUPCIONES SOFTWARE: Esta interrupción se trata de la interrupción SWI y se realiza mediante una instrucción dentro del programa. Es no enmascarable y, por lo tanto, se atiende en todo momento.

Cuando una de las interrupciones tiene lugar (exceptuando el *reset* que se explicará más adelante) y se encuentra habilitada, el procedimiento normal de atención a la interrupción se suspende hasta el final de la ejecución de la instrucción actual. Es entonces, cuando comienza la secuencia general de atención a una interrupción que sigue los siguientes pasos:

1. Se almacena el valor de los registros de la CPU en la pila. En esta tarea se tiene que seguir un orden establecido de manera que sea el contrario al de la instrucción que realiza el retorno de interrupción. Este orden se muestra en la Figura 3.1.
2. El bit I del registro de estado se pone al valor lógico '1' para que otra interrupción no tenga lugar mientras la actual está siendo atendida.

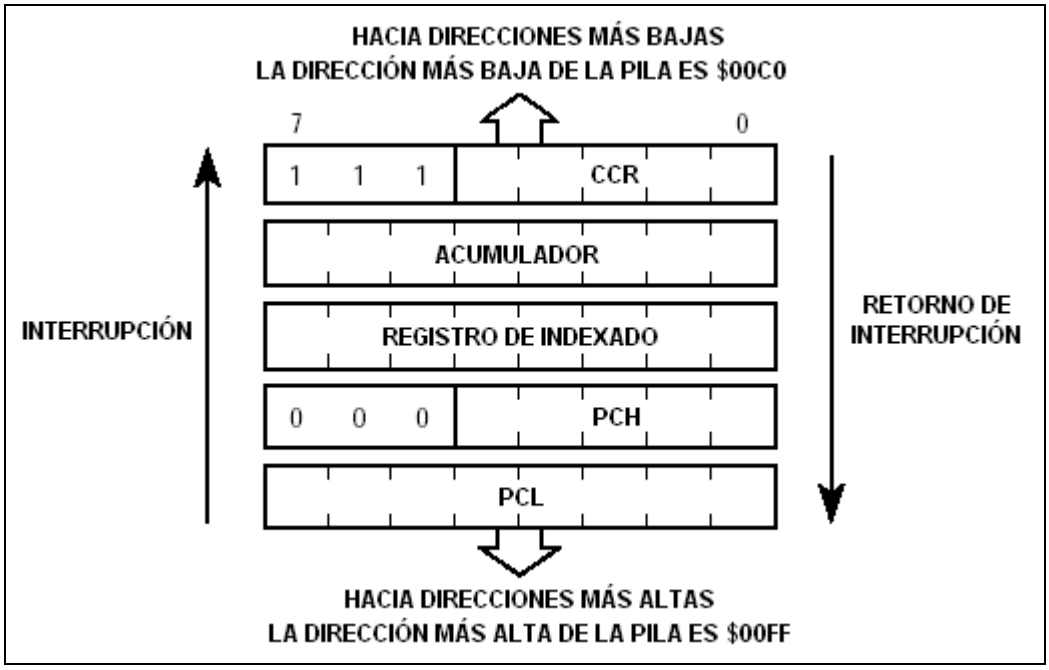


Figura 3.1 Orden para almacenar y extraer los registros de la pila.

3. Dependiendo de la interrupción que se haya producido, la CPU accede a su respectivo vector de interrupción, y la dirección que contenga este, es cargada en el contador de programa. Esta dirección es el punto donde comienza la subrutina de atención a la interrupción.
4. Una vez completada esta subrutina, la instrucción RTI (*Return from Interrupt*), que es normalmente la última instrucción de la subrutina de interrupción, hace que los registros recuperen de la pila el valor que tenían antes de que se produjera la interrupción, y que se vuelva al proceso normal de ejecución del programa continuando con la instrucción siguiente que se hubiera ejecutado en el caso de que no hubiera saltado la interrupción.

Si la interrupción resulta ser un *reset* el proceso es algo distinto. No es necesario que se termine de ejecutar la instrucción que tenia lugar en ese momento para que sea atendido y los pasos a seguir son :

1. Se inicializa el puntero de pila al valor $00FF.
2. El bit I del registro de estado se pone a ‘1’ impidiendo que las interrupciones enmascarables puedan tener lugar.

3. Se carga en el contador de programa el vector de interrupción de *reset*. Este vector se encuentra en las posiciones de memoria $1FFE - $1FFF y contiene la dirección de la primera instrucción del programa principal.

Para el caso de la interrupción *software* este proceso forma parte de la ejecución de la instrucción SWI que se explicará más adelante dentro del juego de instrucciones del microcontrolador.

A continuación se hará una descripción de cada una de las interrupciones pueden ocurrir en este microcontrolador comenzando por la interrupción *software* y a continuación las *hardware*.

3.2.1. SWI (INTERRUPCIÓN SOFTWARE)

Esta interrupción es una instrucción ejecutable dentro del programa que se encuentra almacenado en la memoria del microcontrolador. El procedimiento de atención es similar al de las interrupciones *hardware*. El SWI se ejecuta sin importar el estado de la máscara de la interrupción, el bit I del registro de estado, al ser una interrupción de tipo no enmascarable. La dirección del vector de interrupción, donde comienza la rutina de atención a esta interrupción en concreto, se encuentra en la posiciones de memoria $1FFC y $1FFD.

3.2.2. INTERRUPCIÓN EXTERNA

La interrupción externa recibe este nombre al ser provocada por una señal que recibe el microcontrolador del exterior mediante el pin $\overline{\text{IRQ}}$ dedicado exclusivamente para ello. Esta interrupción tiene lugar si la máscara de interrupción está habilitada y el señal $\overline{\text{IRQ}}$ se encuentra activada. Cuando esto ocurre, se sigue la secuencia normal de atención a la interrupción. La dirección del vector de interrupción se encuentra en las posiciones de memoria $1FFA y $1FFB.

3.2.3. INTERRUPCIÓN DEL TEMPORIZADOR

Esta interrupción puede ser producida por varios de los *flags* que existen dentro de los temporizadores, siempre y cuando estén activos y habilitados a la vez que la máscara de interrupción. Estos *flags* son los 3 bits más significativos del registro de estado del temporizador TSR (*Timer Status Register*). El vector de atención a la interrupción se encuentra en la misma posición de memoria para todos ellos, $1FF8 y $1FF9.

3.2.4. INTERRUPCIÓN DEL SCI

Esta interrupción ocurre cuando uno de los bits que indican una interrupción en el registro de estado del puerto serie asíncrono (SCI) se encuentra activo y habilitado, a la vez que está habilitada la máscara de interrupción. Los bits que permiten la interrupción del SCI se encuentran en el registro de control del mismo. El vector de atención la interrupción del SCI se encuentra en las posiciones de memoria $1FF6 y $1FF7.

3.2.5. INTERRUPCIÓN DEL SPI

Esta interrupción tiene lugar, al igual que las anteriores, cuando alguno de los *flags* de interrupción del registro de estado del puerto serie síncrono (SPI) está activo y habilitado, además de que la máscara de interrupción se encuentre a su vez activa. El vector de atención a esta interrupción se encuentra en las posiciones de memoria $1FF4 y $1FF5.

3.2.6. INTERRUPCIÓN DEL CONVERTIDOR A/D Y DEL GENERADOR PWM

Estas dos interrupciones no se dan en la mayoría de los componentes de la familia de este microcontrolador, aunque si que se han tenido en cuenta a la hora de diseñar e implementar el tratamiento de las interrupciones como muestra de otras interrupciones que sí pueden llegar a tener lugar dependiendo del miembro de la familia que se esté utilizando. La zona dentro de la memoria donde se localizaría su vector de

interrupción sería desde la posición $1FF0 hasta la posición $1FF3. Esta zona se compone de 4 bytes que normalmente no se utilizan.

3.3. RUTA DE DATOS

Se entiende por ruta de datos como el camino que pueden llegar a tomar los datos en el proceso de lectura y ejecución de cada una de las instrucciones que forman parte del programa. La estructura que sigue es la de un sistema secuencial y está constituida por:

- Un conjunto de registros de almacenamiento de datos (datos de entrada, intermedios, y resultados), que son: los registros de la CPU, el registro de instrucción, los registros de acceso a memoria y los registros auxiliares.
- Un sistema de computación, que realiza las operaciones pertinentes sobre los datos, como es la unidad aritmético lógica o ALU.
- Varias líneas de interconexión entre ellos y a su vez con la memoria como son los buses de datos, de direcciones y de control.

3.3.1. REGISTROS DE LA CPU

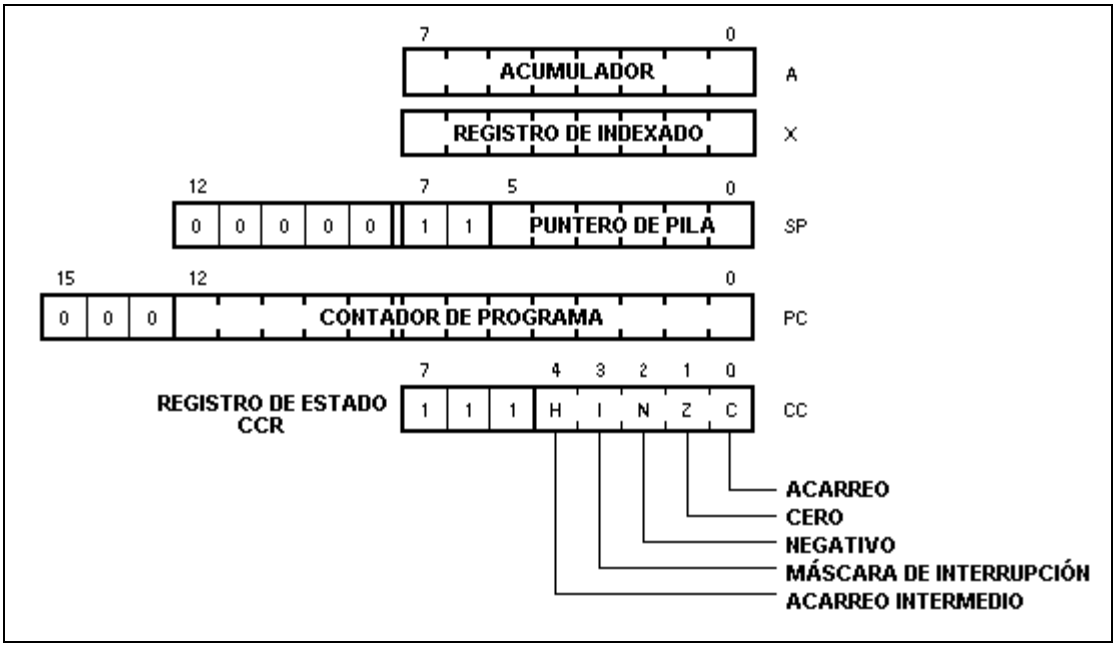


Figura 3.2 Registros de la CPU.

La CPU contiene 5 registros los cuáles se muestran en la Figura 3.2. Estos registros se encuentran dentro del microcontrolador pero sin formar parte de su mapa de memoria. Esta característica permite un acceso mucho más rápido, por parte de la CPU, en comparación con los registros que se encuentran mapeados en la memoria. Una descripción más detallada de la arquitectura, de sus funciones, y de cómo se ha implementado cada registro en el lenguaje VHDL es la se ofrece en los apartados que vienen a continuación.

3.3.1.1. ACUMULADOR

El acumulador es un registro de uso general de 8 bits como se observa en la Figura 3.3, que se utiliza para almacenar los operandos, los resultados de las operaciones que realiza la ALU y los datos que están siendo manipulados. También se usa para operaciones que no son aritméticas como las operaciones lógicas, las de transferencia entre registros, etc.

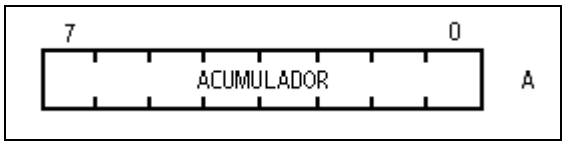


Figura 3.3 Acumulador.

IMPLEMENTACIÓN

El acumulador está compuesto por 8 biestables síncronos con una señal asíncrona de *reset* que inicializa el registro a cero. Este registro se actualiza con el dato que tenga en su entrada, en cada flanco de subida de la señal de reloj si, el estado de una señal de carga proveniente de la unidad de control lo permite.

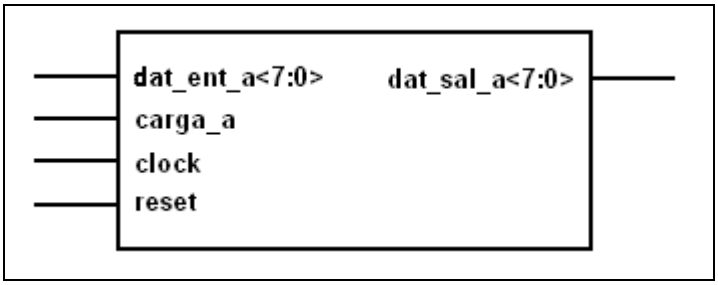


Figura 3.4 Pines de E/S del acumulador.

Los pines del componente que se pueden observar en la Figura 3.4 y sus funciones son las siguientes:

PINES DE ENTRADA:

- **dat_ent_a <7:0>** : dato de entrada al registro acumulador.
- **carga_a** : permite la carga del registro.
- **clock** : señal de reloj del registro al ser un componente síncrono.
- **reset** : señal asíncrona que hace que el registro se cargue a un valor inicial.

PINES DE SALIDA:

- **dat_sal_a<7:0>** : dato de salida del acumulador.

El código fuente donde se encuentra la implementación del registro acumulador se puede observar en el Capítulo 9 Anexo 9.2 dentro del fichero **acumulador.vhd**, y el resultado de la síntesis del mismo se muestra en la Figura 3.5.

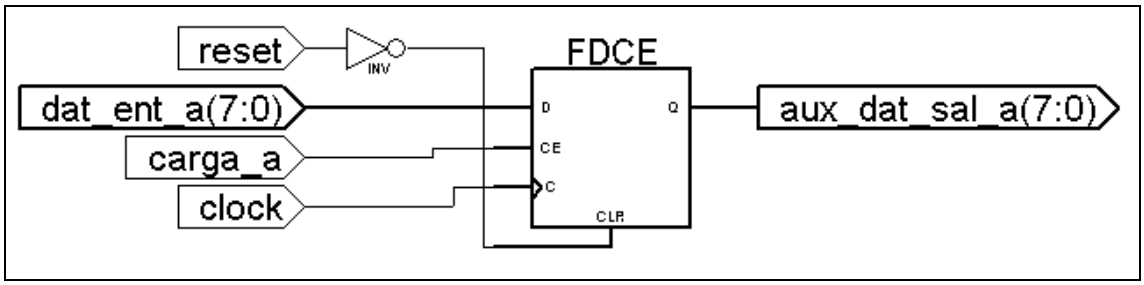


Figura 3.5 Resultado de la síntesis del acumulador.

3.3.1.2. REGISTRO DE INDEXADO

El registro de indexado que se muestra en la Figura 3.6, es un registro de 8 bits que se utiliza principalmente en los modos de direccionamiento indexado o como un acumulador auxiliar. Este registro puede ser cargado directamente por la ALU o desde la memoria y permite realizar diversas operaciones sobre su contenido como incrementarlo, decrementarlo, realizar su complemento, borrarlo, etc, al igual que el acumulador.

En las instrucciones de indexación, el registro de indexado también llamado registro X, proporciona un valor de 8 bits que sumado con los que le proporciona la instrucción crean la dirección efectiva. La longitud de los datos que hay que sumar al registro de indexado puede ser de 0,1 ó 2 bits.

Figura 3.6 Registro de Indexado.

IMPLEMENTACIÓN

Este registro se compone de 8 biestables síncronos con una señal asíncrona de *reset* que inicializa el registro a '0'. La carga del registro con el dato de entrada sólo tiene lugar si lo ordena la unidad de control mediante una señal de carga y si la señal de reloj se encuentra en un flanco de subida.

Los pines de este componente se pueden observar en la Figura 3.7 y sus funciones son las siguientes:

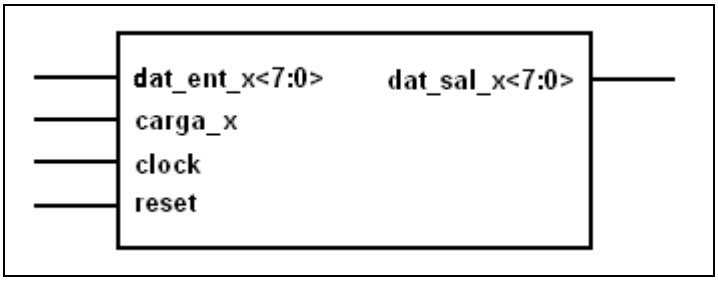


Figura 3.7 Pines de E/S del registro de indexado.

PINES DE ENTRADA:

- **dat_ent_x <7:0>** : dato de entrada al registro de indexado.
- **carga_x** : permite la carga de este registro.
- **clock** : señal de reloj del registro al ser un componente síncrono.
- **reset** : señal asíncrona que hace que el registro se cargue a un valor inicial.

PINES DE SALIDA:

- **dat_sal_x<7:0>** : dato de salida del registro de indexado.

El código fuente que contiene la implementación de este registro se encuentra en el Capítulo 9 Anexo 9.2 dentro del fichero **x.vhd**, y el resultado de la síntesis se puede observar en la Figura 3.8.

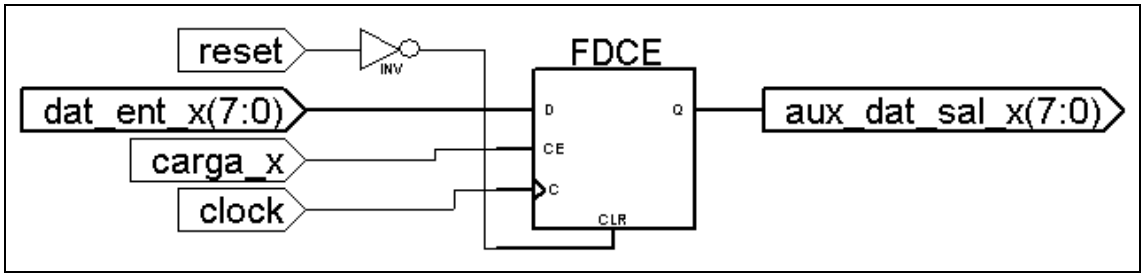


Figura 3.8 Resultado de la síntesis del registro de indexado.

3.3.1.3. REGISTRO DE ESTADO

El registro de estado contiene 5 *flags* que reflejan los resultados de determinadas operaciones de la CPU. Los 5 *flags* son el acarreo intermedio (H), la máscara de las interrupciones (I), el *flag* de un resultado negativo (N), el de cero (Z), y por último el acarreo general (C). Este registro se representa en la Figura 3.9 donde también se pueden observar la posición dentro del mismo de los diferentes *flags* antes mencionados.

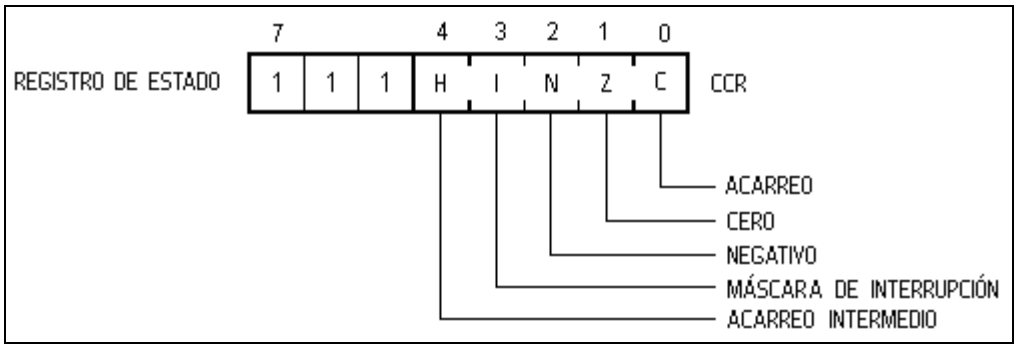


Figura 3.9 Registro de Estado.

- Bit de acarreo intermedio (H): es un *flag* que se utiliza en las operaciones aritméticas para el código binario-decimal y afecta únicamente a las instrucciones de suma sin y con acarreo. El bit H se activa cuando ocurre un acarreo entre los bits 3 y 4 del resultado de estas operaciones.
- Bit máscara de las interrupciones (I): este bit tiene la posibilidad de neutralizar o habilitar todas las fuentes de interrupción enmascarables. Si se borra este bit, se permiten todas las interrupciones y, si se activa, se inhiben. Como ya se ha comentado, cuando una interrupción ocurre el bit I es puesto a '1' automáticamente después que los registros son almacenados en la pila, pero antes de que se lea el vector de interrupción. Si ocurre una interrupción externa mientras que el bit I está activo, la interrupción se almacena y se procesa después que se borre el bit. Por lo tanto, las interrupciones externas que recibe el microcontrolador mediante el pin $\overline{\text{IRQ}}$, no se pierden a pesar de que el bit I se encuentre activo.

  Después de que una interrupción ha sido atendida, la instrucción de retorno de interrupción (RTI), hace que los registros recuperen los valores que tenían antes de que esta se produjera. Normalmente, el bit I se vuelve a poner a cero después de que se ejecute un retorno de interrupción ya que se recupera el valor que tenía el registro de estado antes de que ocurriera la interrupción y para que esta tuviera lugar, el bit debería estar habilitado. Después de un *reset*, este bit es puesto al valor lógico '1' y sólo puede ser borrado mediante la instrucción CLI (*Clear Interrupt Mask Bit*).
- Bit negativo (N): este bit se activa al valor lógico '1' cuando el resultado de una operación aritmética, lógica o de manipulación de datos es negativo, es decir, el bit 7, el más significativo del resultado, se encuentra a '1'.
- Bit Cero (Z): este bit se encuentra activado al nivel lógico '1' cuando es cero el resultado de la última operación aritmética, lógica o de manipulación de datos realizada por la ALU.
- Bit de acarreo (C) : se utiliza para indicar si hay o no acarreo en el resultado de una suma o de una resta. Las instrucciones de rotación y de desplazamientos operan a través del bit de acarreo para facilitar múltiples operaciones de cambio de palabras. Este bit se utiliza también en determinadas instrucciones de salto y condicionales.

IMPLEMENTACIÓN

A pesar de ser un registro de 8 bits, como los 3 primeros bits no se utilizan, la forma de diseñarlo ha sido crear un registro de sólo 5 biestables síncronos con señal de *reset* asíncrona para inicializarlos. Este registro posee una señal de carga para cada bit de forma independiente ya que no en todas las operaciones que realiza el microcontrolador se actualizan todos los bits del registro de estado. La salida es equivalente a la entrada cada vez que se produzca un flanco de subida en la señal de reloj.

Los pines del registro de estado se pueden muestran en la Figura 3.10 y sus funciones son las siguientes:

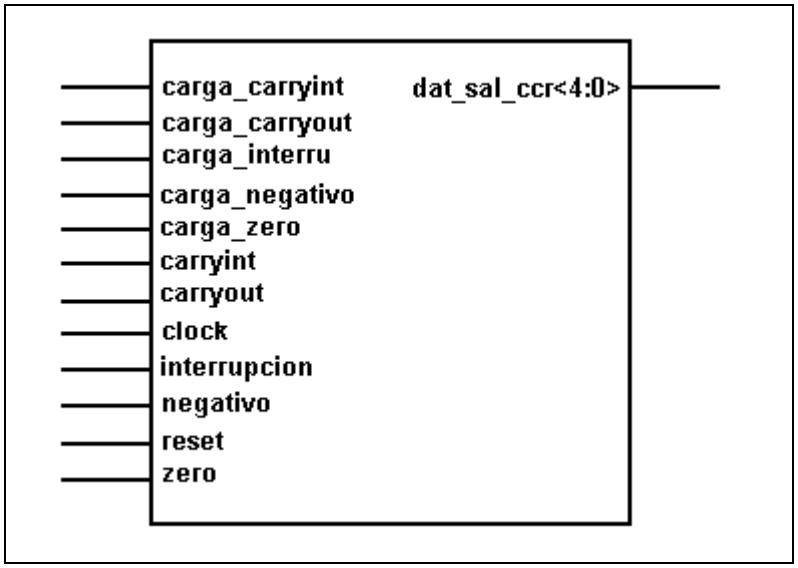


Figura 3.10 Pines de E/S del registro de estado.

PINES DE ENTRADA:

- **carga_carryint** : permite la carga del bit H.
- **carga_carryout** : permite la carga del bit C.
- **carga_interru** : permite la carga del bit I.
- **carga_negativo** : permite la carga del bit N.
- **carga_ zero** : permite la carga del bit Z.
- **carryint** : dato de entrada al bit H.
- **carryout** : dato de entrada al bit C.
- **clock** : señal de reloj del registro al ser un componente síncrono.
- **interrupcion** : dato de entrada al bit I.

- **negativo** : dato de entrada al bit N.
- **reset** : señal asíncrona que hace que el registro se cargue a un valor inicial.
- **zero** : dato de entrada al bit Z.

PINES DE SALIDA:

- **dat_sal_ccr<7:0>** : dato de salida del registro de estado.

El código fuente de este registro se encuentra en el Capítulo 9 Anexo 9.2, y más concretamente en el fichero **ccr.vhd**. El resultado de la síntesis del mismo se puede observar en la Figura 3.11.

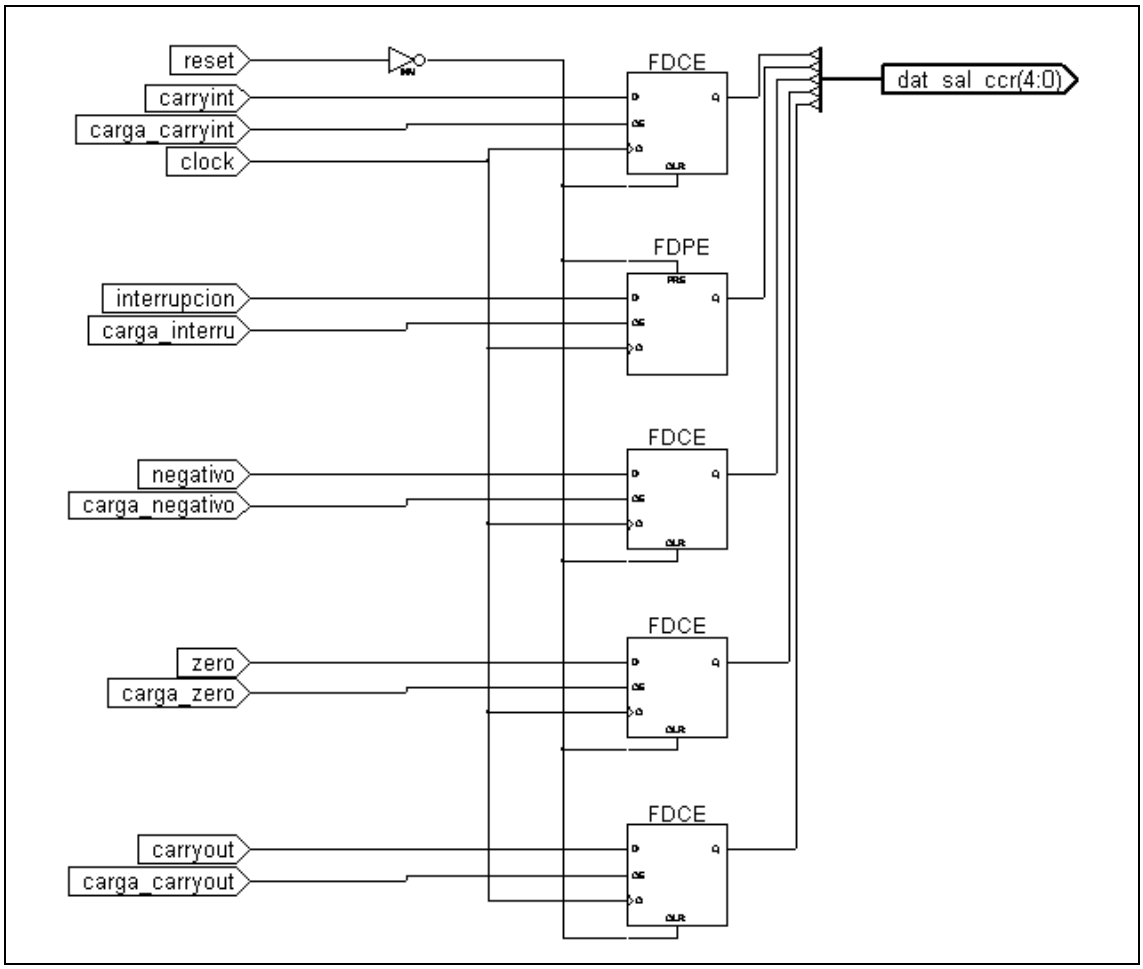


Figura 3.11 Resultado de la síntesis del registro de estado.

3.3.1.4. CONTADOR DE PROGRAMA

El contador de programa es un registro de 15 bits que se utiliza para almacenar la dirección de la próxima instrucción que va a ser ejecutada por la CPU. Normalmente este contador se incrementa al tiempo que la instrucción de la que actualmente guarda su dirección es procesada. Este registro se representa en la Figura 3.12.

Figura 3.12 Contador de Programa.

Las instrucciones de salto, condicionales y el tratamiento de las interrupciones hacen que algunas veces se cargue en este registro una dirección de memoria diferente a la que le correspondería si se siguiese con la secuencia habitual del programa.

IMPLEMENTACIÓN

Es un registro de 15 bits pero para el caso del miembro de la familia que se va a diseñar, sólo se utilizan 13 bits, al disponer el dispositivo de una memoria con un tamaño de 8 Kbytes. Es por esto que los 3 bits más significativos del registro toman siempre el valor '0'. Luego el registro, para este caso, se compone de 13 biestables síncronos con una señal de *reset* asíncrona que le inicializa a la dirección donde se encuentra el vector de interrupción de *reset*, es decir, al valor $1FFE como ya se ha comentado en el apartado de las interrupciones y concretamente en el proceso de *reset.* Con cada flanco de subida de la señal de reloj los biestables pueden incrementarse, cargarse a un determinado valor, o bien, mantener el dato que tenían actualmente, todo ello gobernado por señales que proporciona la unidad de control.

Este registro es algo más complejo que los anteriores ya que dispone en sí mismo de la opción de poder incrementarse. Esto se debe a que después de realizar cada instrucción de programa necesita poder acceder a la siguiente posición de memoria para leer la instrucción que va a continuación o los operandos necesarios para realizar la presente instrucción.

Aparte de esta opción también hay que tener en cuenta que el microcontrolador dispone de la instrucción de salto incondicional además del direccionamiento relativo, de manera que este registro debe de tener la posibilidad de poder ser cargado a un determinado valor (el que indiquen los operandos de la instrucción) o bien de poder sumar a su contenido actual un valor tanto positivo como negativo (también indicado en los operandos) para realizar el direccionamiento relativo.

Para que el registro se pueda incrementar así como realizar el salto condicional tiene instanciado un sumador-restador de 13 bits. Este sumador-restador es el resultado de la unión de 13 sumadores totales de un bit. Si se desea que el contador de programa simplemente se incremente los sumandos son: el valor actual del registro, más la unidad. Si por el contrario se quiere realizar el direccionamiento relativo, los sumandos son: el valor actual del registro, y el *offset* relativo contenido en los operandos de la instrucción. Al estar en este tipo de direccionamiento, este dato viene dado en complemento a dos, de manera que, la decisión de sumar o restar la proporciona su bit más significativo, siendo éste el bit que marca el signo del segundo sumando. Si simplemente se quiere realizar un salto incondicional, no se utiliza para nada el sumador-restador que contiene el registro y sólo con cargarlo a un determinado valor que le llega como entrada ya es suficiente.

En vez de utilizar el sumador de 13 bits que se ha comentado en el párrafo anterior, se podría haber utilizando la ALU, pero eso supondría un aumento en el número de ciclos de reloj utilizados para ejecutar cada instrucción y puesto que este registro está sometido a que su valor cambie continuamente se ha preferido que sea autónomo en cuanto a que él mismo puede realizar estas operaciones sin necesidad de utilizar la ALU.

Como para la implementación de este circuito se han necesitado entidades de menor nivel como son los circuitos sumadores se va a explicar desde la entidad de menor nivel hasta llegar a la de mayor nivel, el contador de programa.

En un principio se ha implementado un sumador total de un bit. Este sumador contiene dos operadores lógicos que calculan el resultado de la suma y el acarreo de salida a partir de los operandos A, B y el acarreo de entrada. Los pines de E/S que presenta el circuito sumador total de un bit se muestran en la Figura 3.13 y son los siguientes:

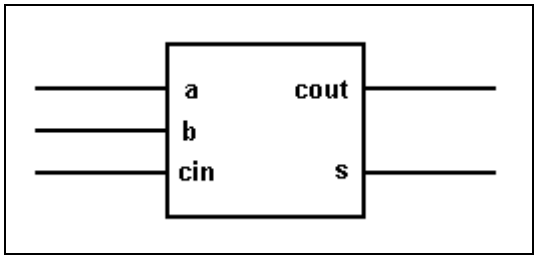


Figura 3.13 Pines de E/S de un sumador total de 1 bit.

PINES DE ENTRADA:

- **a** : dato de entrada. Es uno de los operandos y se le suele denominar operando A.
- **b** : dato de entrada. Este es el otro operando, es decir, el operando B.
- **cin** : es el acarreo de entrada.

PINES DE SALIDA:

- **cout** : acarreo de salida.
- **s** : dato de salida. Es el resultado de la suma.

El código del sumador total de un bit se puede observar en el fichero **sum_total.vhd** que se adjunta en el Capítulo 9 Anexo 9.2, y el resultado de la síntesis se muestra en la Figura 3.14.

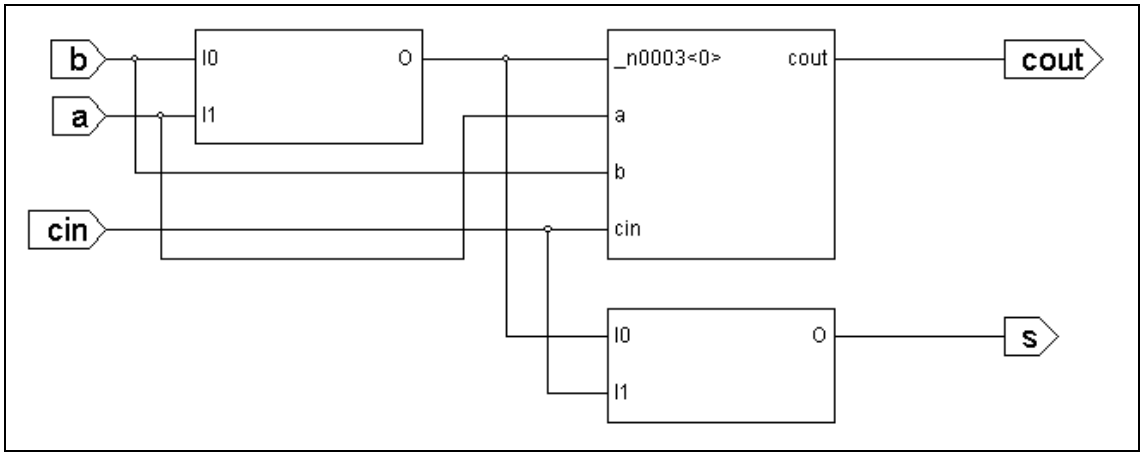


Figura 3.14 Resultado de la síntesis del sumador total de 1 bit.

Una vez que se ha implementado el sumador de 1 bit, necesitamos extenderlo a 13 bits siendo esta la longitud de datos que maneja el contador de programa. Para ello basta con instanciar en una entidad de mayor nivel el sumador de un bit 13 veces y asignarle las señales correspondientes a cada uno de sus pines.

Los pines de E/S que presenta el circuito sumador total de 13 bits son los que se pueden observar en la Figura 3.15 y llevan asignadas las señales siguientes:

PINES DE ENTRADA:

- **a<12:0>** : dato de entrada. Es el operando A.
- **b<12:0>** : dato de entrada. El operando B.
- **cin** : es el acarreo de entrada.

PINES DE SALIDA:

- **s<12:0>** : dato de salida. Es el resultado de la suma.
- **cout** : acarreo de salida.

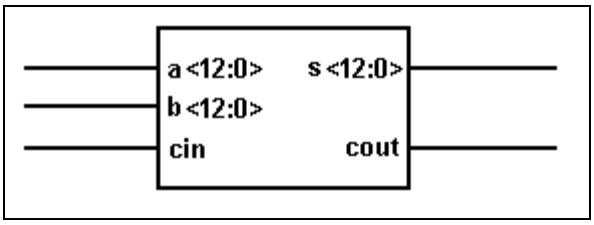


Figura 3.15 Pines de E/S de un sumador total de 13 bits.

El código fuente de este componente se adjunta en el Capítulo 9 Anexo 9.2 dentro del fichero **sum_total_13bits.vhd,** y el resultado de la síntesis en la Figura 3.16.

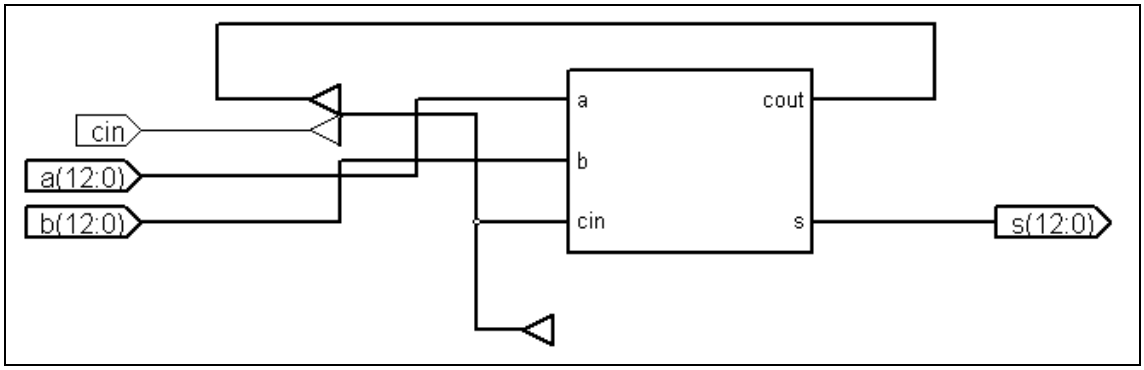


Figura 3.16 Resultado de la síntesis del sumador total de 13 bits.

Una vez que ya se ha diseñado el sumador total de 13 bits que lleva incorporado el contador de programa, es necesario que este sumador tenga la posibilidad de restar para las instrucciones con direccionamiento relativo como ya se comentó anteriormente. Por eso a partir del sumador total de 13 bits es necesario realizar un sumador-restador. Esto se realiza instanciando en una módulo de mayor jerarquía el sumador total de 13 bits y asignando a sus pines diferentes señales dependiendo si se quiere sumar o restar.

Los pines de E/S que presenta el circuito sumador-restador del contador de programa se muestran en la Figura 3.17 y son los siguientes:

PINES DE ENTRADA:

- **a<12:0>** : dato de entrada. Es el operando A.
- **b<12:0>** : dato de entrada. El operando B.
- **cin** : es el acarreo de entrada.
- **s_r** : esta señal indica si la operación a realizar es la suma o la resta.

PINES DE SALIDA:

- **s<12:0>** : dato de salida. Es el resultado de la suma.
- **cout** : acarreo de salida.

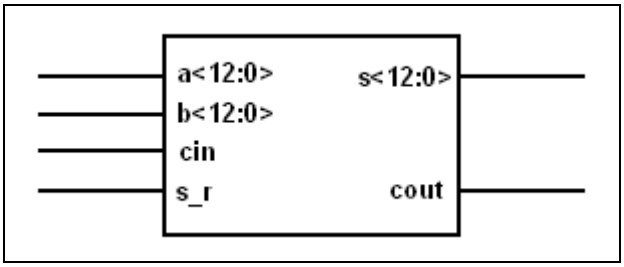


Figura 3.17 Pines de E/S del sumador-restador del contador de programa.

El resultado de la síntesis de este componente se muestra en la Figura 3.18 y el código fuente mediante el que ha sido implementado en el fichero **sum_res_pc.vhd** en el Capítulo 9 Anexo 9.2.

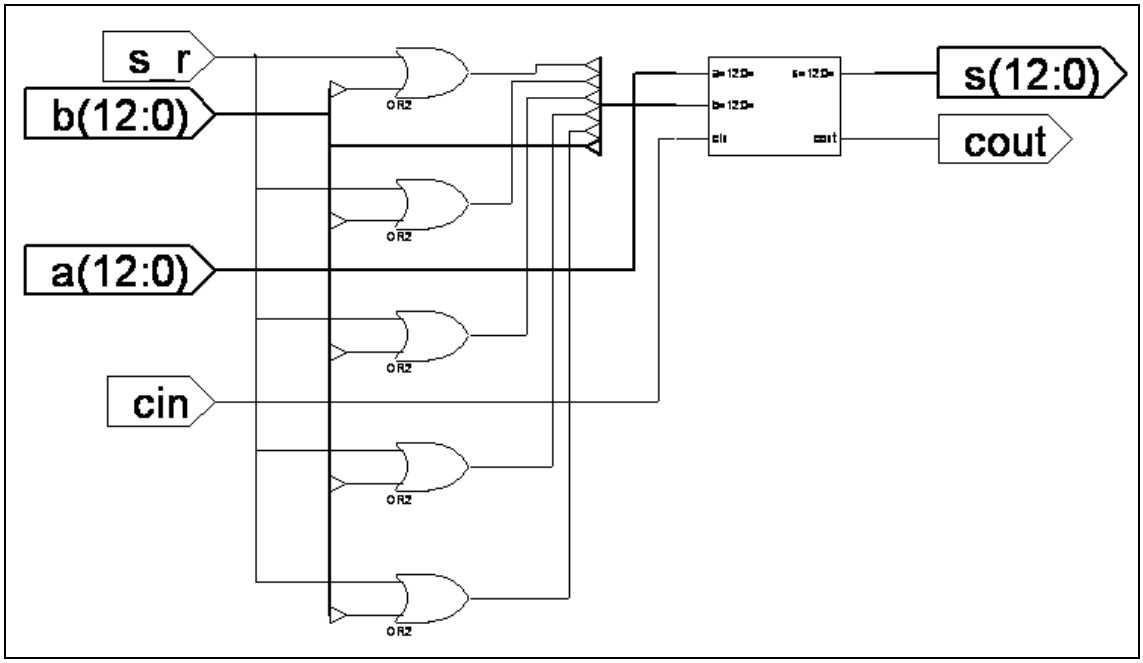


Figura 3.18 Resultado de la síntesis del circuito sumador-restador del contador de programa.

Una vez que ya se ha completado el circuito operador que lleva incorporado el contador de programa sólo es necesario instanciarlo y asignarle sus correspondientes señales junto con el resto de componentes que forman el contador de programa como son los 13 biestables síncronos ya descritos.

Los pines del contador de programa que se pueden ver en la Figura 3.19 y sus funciones son las siguientes:

PINES DE ENTRADA:

- **dat_ent_pc <12:0>** : dato de entrada al contador de programa.
- **carga_pc** : permite la carga del contador con el dato de entrada.
- **clock** : señal de reloj del registro al ser un componente síncrono.
- **inc_pc** : permite el incremento en una unidad del dato almacenado en el registro.
- **relativo** : señal que le indica si se tiene un direccionamiento relativo y, si la cantidad a incrementar o decrementar el dato del contador de programa es el dato de entrada al registro en vez de la unidad.
- **reset** : señal asíncrona que hace que el registro se cargue a un valor inicial.

PINES DE SALIDA:

- **dat_sal_pc<12:0>** : dato de salida del contador de programa.

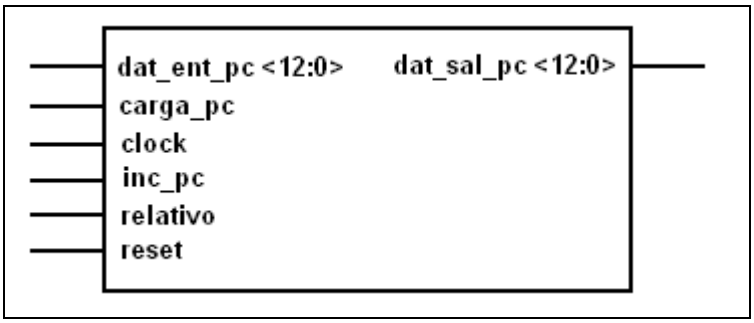


Figura 3.19 Pines de E/S del contador de programa.

El código fuente donde se ha realizado la implementación de este componente se encuentra en el fichero **pc.vhd** dentro del Capítulo 9 Anexo 9.2 del presente documento y el resultado de la síntesis del mismo se puede observar en la Figura 3.20.

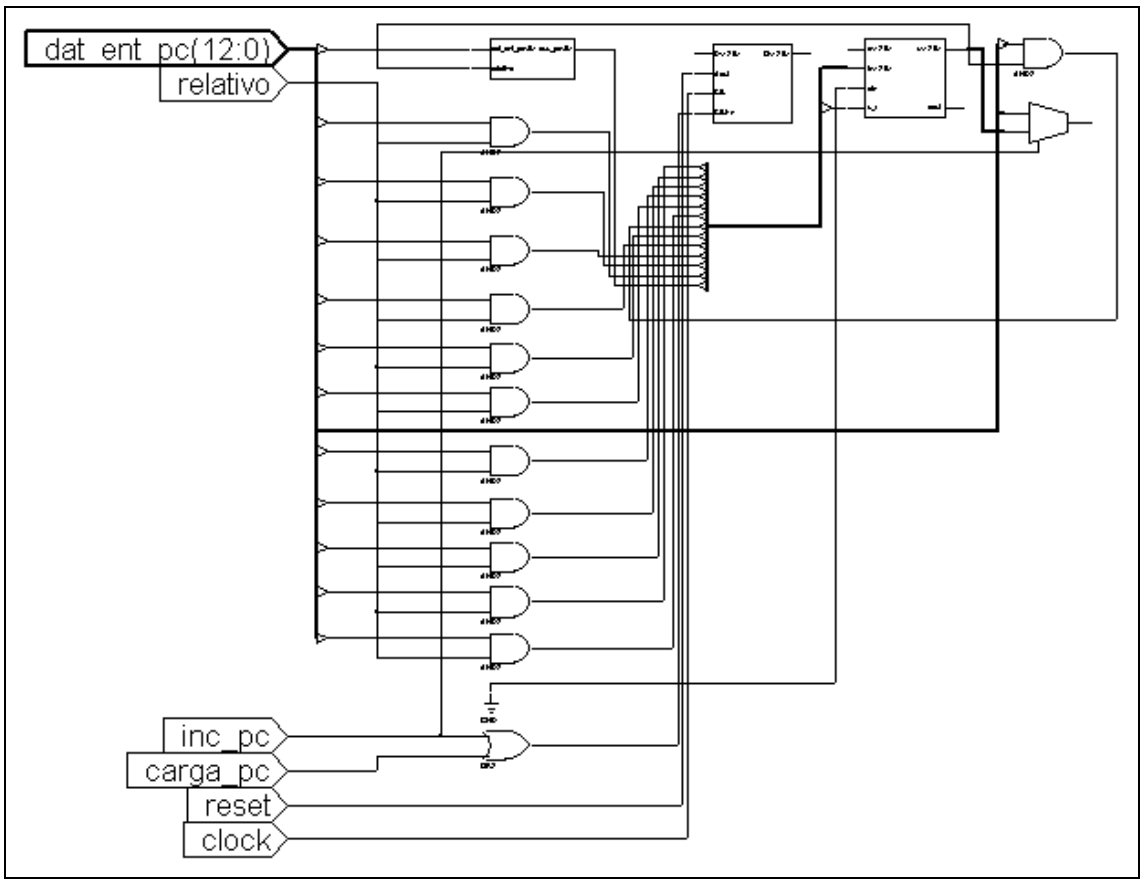


Figura 3.20 Resultado de la síntesis del contador de programa.

3.3.1.5. PUNTERO DE PILA

El puntero de pila es un registro también de 13 bits que viene representado en la Figura 3.21. Este registro se encarga de almacenar la dirección de la siguiente posición que hay libre en la zona de la memoria denominada pila. En el proceso de *reset* este registro es inicializado al valor $00FF. Este valor es decrementado cada vez que se deposita un dato en la pila y por el contrario es incrementado si se extrae un dato de la misma.

Este registro tiene los siete bits más significativos puestos permanentemente al valor “0000011”, que añadidos al valor de los bits menos significativos, producen una dirección con un rango desde $00FF hasta $00C0, es decir, que las subrutinas o interrupciones tienen disponibles 64 posiciones de memoria para guardar los datos de los diferentes registros. Si se exceden estas 64 posiciones, el puntero de pila comienza de nuevo a apuntar al valor inicial $00FF, y se pierde la información almacenada previamente. Una subrutina ocupa dos posiciones de la pila, mientras que una interrupción ocupa cinco de estas posiciones.

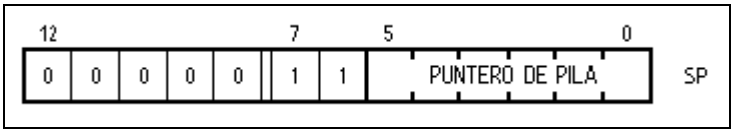


Figura 3.21 Puntero de Pila.

IMPLEMENTACIÓN

Este registro simplemente se encarga de proporcionar la dirección de la parte de la memoria donde se carga el dato que se requiere almacenar durante una interrupción o un salto a subrutina, con lo cuál, simplemente debe disponer de la opción de poder incrementarse o decrementarse. Esto se realiza a partir de un circuito sumador.

Este registro a pesar de ser un registro de 13 bits, se ha diseñado como un registro de 8 biestables síncronos. La razón de esto se debe a que los 5 bits más significativos del registro siempre se encuentran al valor lógico ‘0’, con lo cuál, sólo ha hecho falta asignarles este valor. Por esta razón se debería haber implementado sólo los 6 bits menos significativos ya que como se puede observar en la Figura 3.21 los bits 7 y 6 se encuentran a ‘1’ todo el tiempo. La razón de incluir estos dos bits se debe a que a la hora de instanciar un sumador para poder incrementar o decrementar el valor del

puntero, se ha utilizado el sumador de 8 bits que previamente ya se había diseñado para utilizarlo en otros elementos del microcontrolador.

Los 8 biestables síncronos disponen de una señal asíncrona, el *reset*, que en este caso los inicializa al valor $FF para así cumplir las especificaciones de que el puntero apunte al primer valor de la pila. Estos biestables almacenan el valor actual al que debe se debe apuntar en ese momento y sólo se actualizan con la salida del sumador del que dispone este registro cuando una señal que proviene de la unidad de control lo ordena. El sumador descuenta o añade una unidad al valor actual del registro si las señales correspondientes también gobernadas por la unidad de control lo disponen (*push, pull* respectivamente). Como el circuito que lleva este registro es únicamente un sumador la manera de decrementar es haciendo que el segundo sumando sea el complemento a dos de la unidad.

El circuito sumador necesario para el puntero de pila debe ser de 8 bits ya que esta es la longitud de los datos que maneja. Este sumador se realiza instanciando 8 veces un sumador total de 1 bit (comentado en el apartado del contador de programa) y asignándole las señales correspondientes. Los pines de E/S que presenta este circuito se pueden observar en la Figura 3.22 y son los siguientes:

PINES DE ENTRADA:

- **a<7:0>** : dato de entrada. Es el operando A.
- **b<7:0>** : dato de entrada. El operando B.
- **cin** : es el acarreo de entrada.

PINES DE SALIDA:

- **s<7:0>** : dato de salida. Es el resultado de la suma.
- **cout** : acarreo de salida.

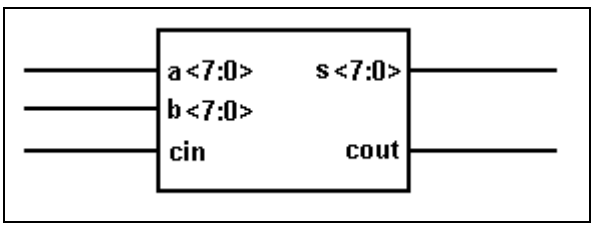


Figura 3.22 Pines de E/S del sumador total de 8 bits.

El código fuente donde se encuentra la implementación de este componente está en el Capítulo 9 Anexo 9.2 dentro del fichero **sum_total_8bits.vhd,** y el resultado de la síntesis se puede observar en la Figura 3.23.

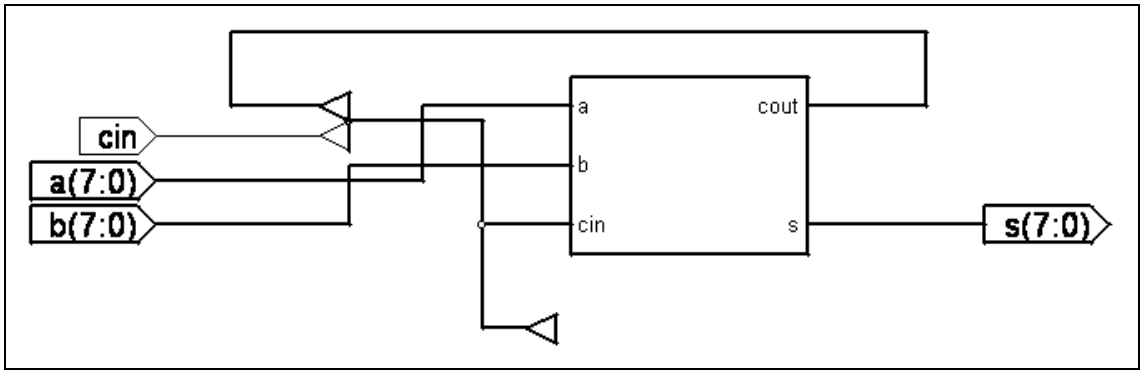


Figura 3.23 Resultado de la síntesis del sumador total de 8 bits.

Una vez que ya se ha implementado el sumador de 8 bits ya se puede instanciar este dentro de la entidad superior, es decir, el puntero de pila. Los pines de este componente se pueden observar en la Figura 3.24 y son las siguientes:

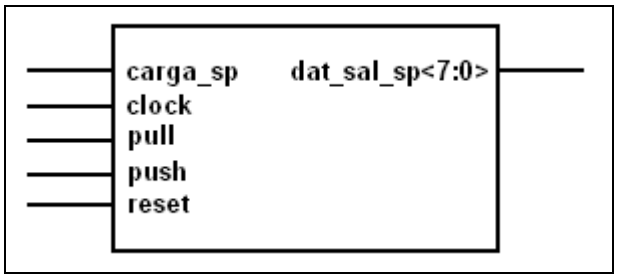


Figura 3.24 Pines de E/S del puntero de pila.

PINES DE ENTRADA:

- **carga_sp** : permite la carga del puntero de pila.
- **clock** : señal de reloj del registro al ser un componente síncrono.
- **pull** : señal que indica si hay que incrementar en una unidad el contenido del puntero de pila. Se activará cuando se desee extraer un dato de la pila.
- **push** : señal que indica si por el contrario hay que decrementar en una unidad el contenido del puntero de pila y que se activará cuando se quiera almacenar un dato en la pila.

- **reset** : señal asíncrona que hace que el registro se cargue a un valor inicial.

PINES DE SALIDA:

- **dat_sal_sp<7:0>** : dato de salida del puntero de pila.

Los resultados de la síntesis del puntero de pila se muestran en la Figura 3.25 y el código fuente mediante el cuál ha sido implementado en el Capítulo 9 Anexo 9.2 dentro del fichero **sp.vhd.**

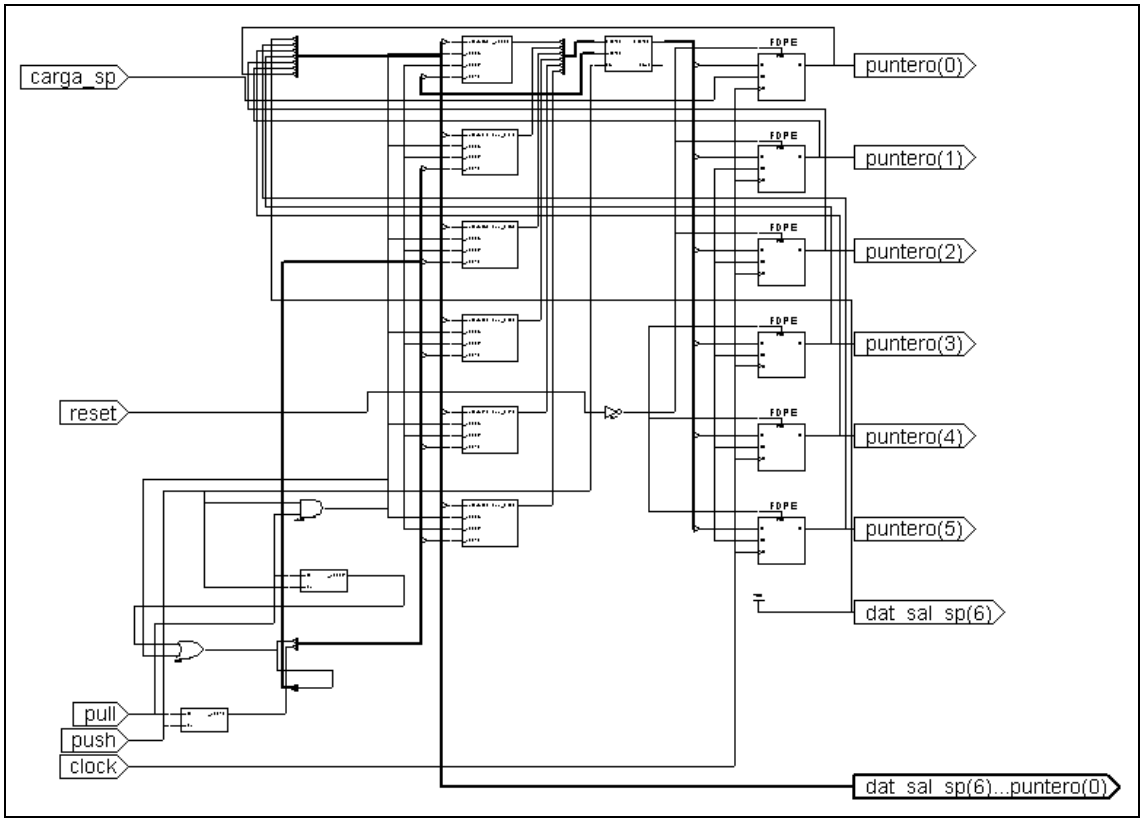


Figura 3.25 Resultado de la síntesis del puntero de pila.

3.3.2. REGISTROS DE ACCESO A MEMORIA

Estos registros representan un nexo de unión entre lo que es el núcleo del microcontrolador y la memoria o cualquier otro dispositivo periférico que pudiera añadirse al núcleo en futuras aplicaciones. Su papel es muy importante por ser los que

proporcionan la manera de comunicarse al núcleo, pero es imprescindible dentro de este proyecto a la hora de realizar las pruebas necesarias para validar el funcionamiento del mismo. Estos registros son los siguientes:

3.3.2.1. MAR

Este registro llamado MAR (*Memory Addres Register*) que se representa en la Figura 3.26, es el encargado de almacenar el valor de la dirección de memoria a la cuál desea acceder la CPU. Es el intermediario entre el bus de direcciones y la memoria. Este es un registro unidireccional ya que es la CPU la que envía la dirección de memoria a la cuál se quiere tener acceso.

Figura 3.26 Registro MAR.

IMPLEMENTACIÓN

Se trata de un conjunto de 13 biestables síncronos que simplemente cargan el valor de la entrada que reciben en el registro cuando la unidad de control lo permite. Todos los biestables tiene una señal asíncrona de *reset* que en este caso inicializa el registro a la dirección de la memoria donde se encuentra el vector de *reset*, es decir, al valor $1FFE.

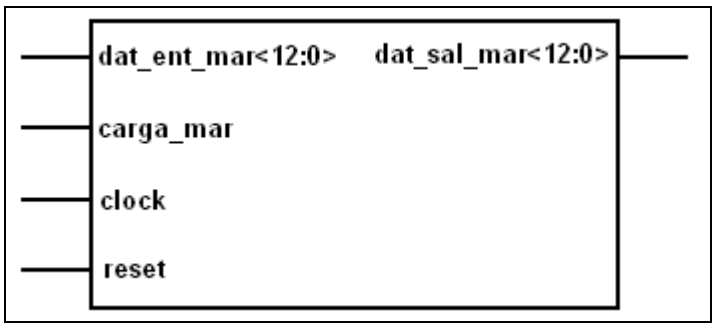


Figura 3.27 Pines de E/S del registro MAR.

Los pines del registro MAR que se pueden observar en la Figura 3.27 y sus funciones son las siguientes:

PINES DE ENTRADA:

- **dat_ent_mar<12:0>** : dato de entrada al registro MAR.
- **carga_mar** : permite la carga del registro.
- **clock** : señal de reloj del registro al ser un componente síncrono.
- **reset** : señal asíncrona que hace que el registro se cargue a un valor inicial.

PINES DE SALIDA:

- **dat_sal_mar<12:0>** : dato de salida del registro MAR.

El código fuente donde se puede ver la implementación del registro MAR se encuentra en el fichero **mar.vhd** del Capítulo 9 Anexo 9.2. Los resultados de la síntesis de este código se muestran en la Figura 3.28.

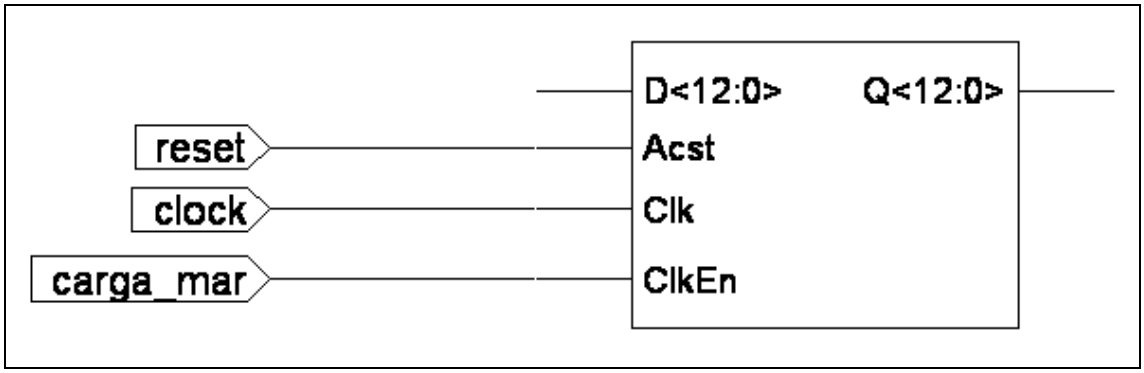


Figura 3.28 Resultado de la síntesis del registro MAR.

3.3.2.2. MBR

El registro MBR (*Memory Buffer Register*), que se representa en la Figura 3.29, es muy similar al anterior, pero en este caso, es el encargado de comunicar la memoria con el bus de datos. La función de este registro es la de enviar los datos que se encuentran almacenados dentro de la posición de memoria a la que ha accedido la CPU al bus de datos y viceversa, es decir, almacenar en la memoria los datos que la CPU envía, por medio del bus de datos, a una determinada posición de la memoria.

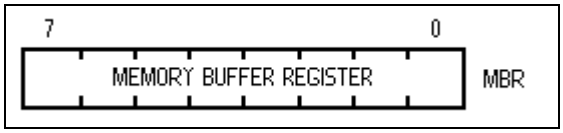


Figura 3.29 Registro MBR.

IMPLEMENTACIÓN

El registro está compuesto por 16 biestables síncronos con una señal de *reset* asíncrona, que inicializa a todos ellos a cero. Con cada flanco de subida de la señal de reloj, la entrada del registro es asignada a una señal auxiliar teniendo en cuenta el sentido que haya sido seleccionado en ese momento. Tiene 16 biestables ya que 8 guardan los datos que le llegan de la memoria y otros 8 los que se van a escribir en ella.

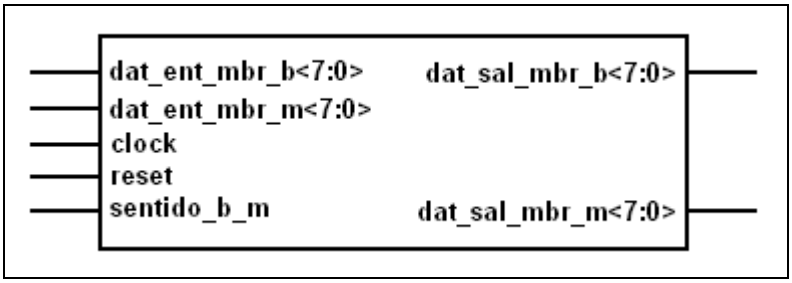


Figura 3.30 Pines de E/S del registro MBR.

Los pines del registro MBR se muestran en la Figura 3.30 y sus funciones son las siguientes:

PINES DE ENTRADA:

- **dat_ent_mbr_b<7:0>** : dato de entrada al registro desde el bus de datos.
- **dat_ent_mbr_m<7:0>** : dato de entrada al registro desde la memoria.
- **clock** : señal de reloj del registro al ser un componente síncrono.
- **reset** : señal asíncrona que hace que el registro se cargue a un valor inicial.
- **sentido_b_m** : señal que indica el sentido de los datos, si van desde la memoria al bus de datos, como ocurre cuando se realiza una lectura de un dato de la memoria o, al contrario, del bus de datos a la memoria, que tiene lugar cuando se quiere escribir en la memoria.

PINES DE SALIDA:

- **dat_sal_mbr_b<7:0>** : dato de salida del registro al bus de datos.
- **dat_sal_mbr_m<7:0>** dato de salida del registro a la memoria.

El código fuente de esta registro se detalla en el Capítulo 9 Anexo 9.2 dentro del fichero **mbr.vhd,** y los resultados de la síntesis del mismo se muestran en la Figura 3.31**.**

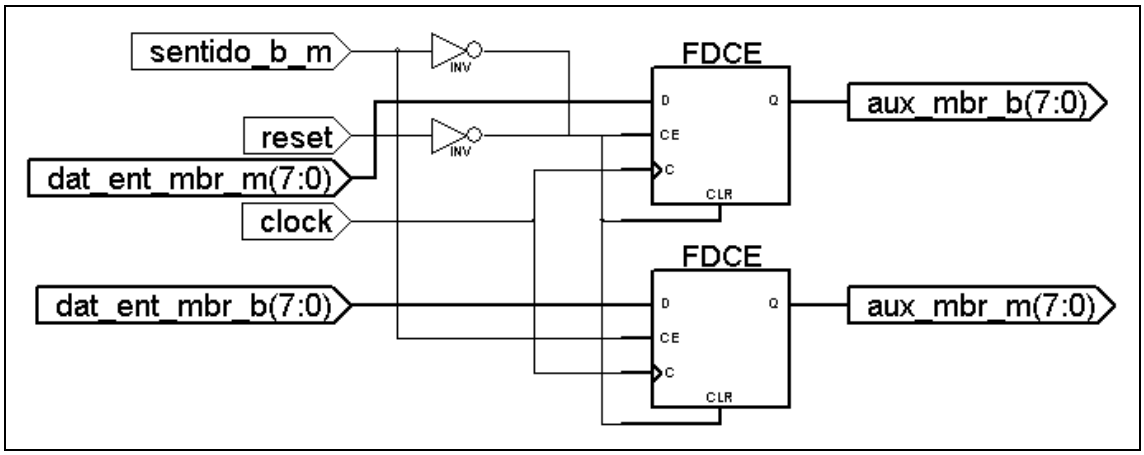


Figura 3.31 Resultado de la síntesis del registro MBR.

3.3.3. REGISTRO DE INSTRUCCIÓN

El registro de instrucción es el encargado de almacenar la instrucción que el microcontrolador está ejecutando en ese momento. Este registro, que se representa en la Figura 3.32, es imprescindible dentro del núcleo, ya la unidad de control se basa en su contenido a la hora de tener que enviar las órdenes necesarias al resto de los registros y activar las señales correspondientes para que se ejecute la instrucción almacenada en él.

Figura 3.32 Registro de Instrucción.

IMPLEMENTACIÓN

Este registro está compuesto por 8 biestables síncronos con una señal de *reset* como señal asíncrona. Cuando se produce un *reset*, los 8 biestables son inicializados al valor $90. Este valor es un código de operación que no corresponde a ninguna instrucción y que se ha asignado por defecto para que la unidad de control sepa que tiene que ejecutar la rutina de *reset* y pueda tomar las decisiones adecuadas utilizando procedimientos similares a los que se utilizan para el resto de las instrucciones del programa. La carga de este registro es gobernada por la unidad de control que activa una señal que hace que con cada flanco de subida de la señal de reloj, la entrada al registro se cargue en él.

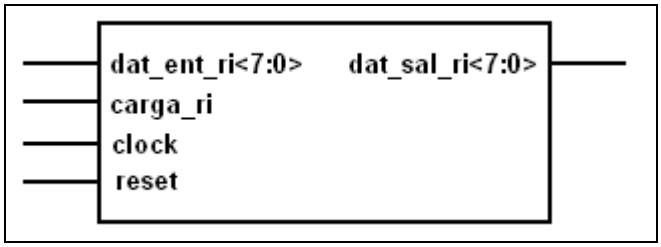


Figura 3.33 Pines de E/S del registro de instrucción.

Los pines del componente que se pueden observar en la Figura 3.33 y las señales asignadas a cada uno de ellos son las siguientes:

PINES DE ENTRADA:

- **dat_ent_ri<7:0>** : dato de entrada al registro.
- **carga_ri** : permite la carga del registro.
- **clock** : señal de reloj del registro al ser un componente síncrono.
- **reset** : señal asíncrona que hace que el registro se cargue a un valor inicial.

PINES DE SALIDA:

- **dat_sal_ri<7:0>** : dato de salida del registro.

Los resultados de la síntesis de este registro se pueden ver en la Figura 3.34 y el código fuente mediante el que ha sido implementado se encuentra en el fichero **ri.vhd** del Capítulo 9 Anexo 9.2.

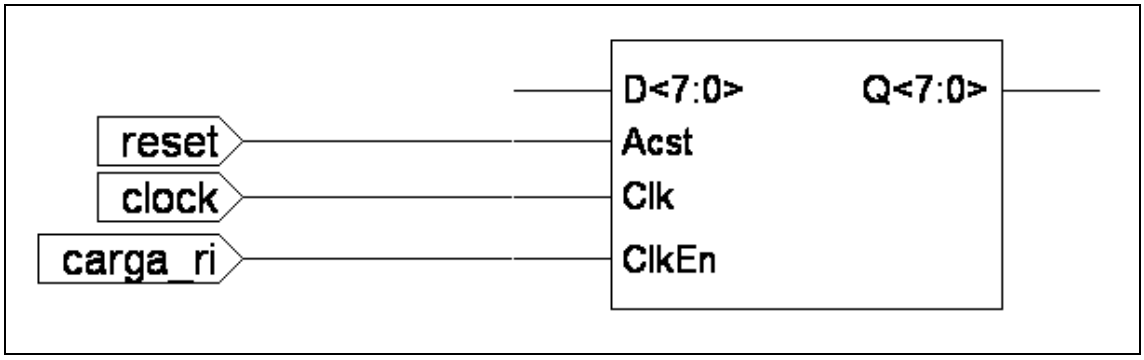


Figura 3.34 Resultado de la síntesis del registro de instrucción.

3.3.4. REGISTROS AUXILIARES

Para la implementación de ciertas operaciones del microcontrolador han sido necesarios tres registros auxiliares: el registro reg_aux, el registro reg_sal_alu y el registro reg_dir. Estos tres registros tienen como función principal almacenar datos que en cierto momento son calculados o extraídos de la memoria, pero que no se pueden designar a su destino hasta pasado cierto número de ciclos de reloj debido a que el registro destino debe contener otro tipo de datos en ese momento. Estos tres registros son:

3.3.4.1. REG_AUX

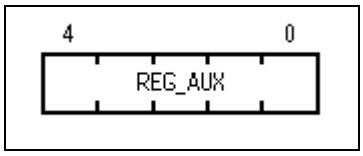


Figura 3.35 Registro auxiliar reg_aux.

Este registro llamado reg_aux se encarga de almacenar el byte más significativo en los modo de direccionamiento extendido e indexado con *offset* de 16 bits. Es necesario ya que en estos dos tipos de direccionamiento primero se obtiene el byte más significativo y después el menos significativo a la hora de formar la dirección efectiva. Hasta que este último no haya sido leído de la memoria, se precisa almacenar el byte más significativo en un registro para, más tarde, poder conformar la dirección indicada.

Como la dirección está formada por 13 bits, sólo se va a tener que almacenar los 5 bits menos significativos del byte alto, es por ello que este es un registro de 5 bits. Este registro se muestra en la Figura 3.35.

IMPLEMENTACIÓN

Este es un registro síncrono compuesto por 5 biestables y con una señal asíncrona de *reset* que lo inicializa al valor $00. Con cada flanco de subida de la señal de reloj y si la señal de carga se encuentra activada se almacena en el registro el dato que hay en su entrada.

Los pines de E/S a este registro auxiliar son los que se muestran en la Figura 3.36 y se enumeran a continuación:

PINES DE ENTRADA:

- **dat_ent_reg_aux<4:0>** : dato de entrada al registro reg_aux.
- **clock** : señal de reloj del registro al ser un componente síncrono.
- **ext_ind16** : señal que indica la carga del registro.
- **reset** : señal asíncrona que hace que el registro se cargue a un valor inicial.

PINES DE SALIDA:

- **dat_sal_reg_aux<4:0>** : dato de salida del registro auxiliar reg_aux.

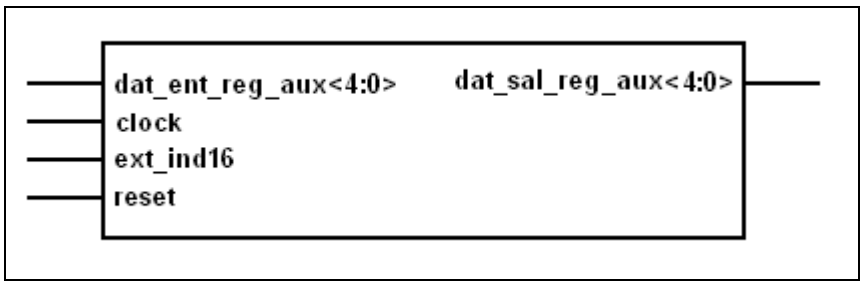


Figura 3.36 Pines de E/S del registro auxiliar reg_aux.

El código fuente de este registro se encuentra en el fichero **reg_aux.vhd** del Capítulo 9 Anexo 9.2, y el resultado de la síntesis del mismo se muestra en la Figura 3.37.

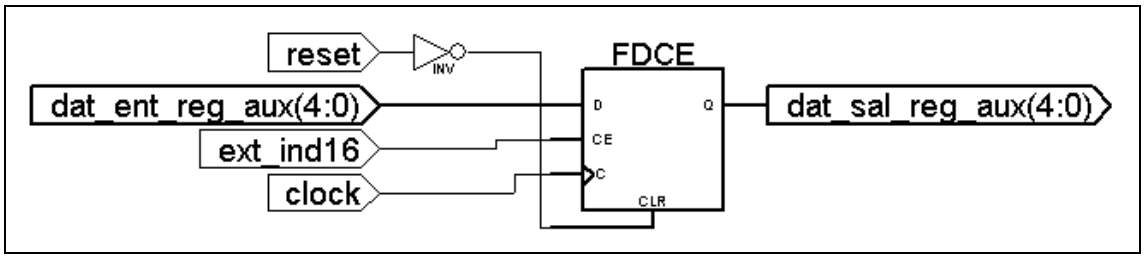


Figura 3.37 Resultado de la síntesis del registro auxiliar reg_aux.

3.3.4.2. REG_SAL_ALU

Este registro llamado reg_sal_alu que se representa en la Figura 3.38, es un registro de 8 bits, encargado de almacenar el resultado de la ALU para determinadas operaciones en las que hay que esperar un ciclo de reloj para poder almacenar este resultado en el registro destino. Un ejemplo de esto es el caso de la operación INCA. En esta operación se proporciona a la ALU el dato del acumulador para que lo incremente, pero al ser la ALU un componente combinacional en el mismo ciclo obtenemos el resultado que hay que volver a almacenar en el acumulador. Pero en ese momento, el dato del acumulador es el que todavía no ha sido incrementado y si se intenta guardar el nuevo dato se produce un error, así que hay que esperar al siguiente ciclo de reloj para poder almacenar el nuevo dato en el acumulador.

Este registro también dispone de la capacidad de almacenar el bit de acarreo por la misma razón dada anteriormente. Tanto en el direccionamiento indexado con *offset* de 8 como de 16 bits es necesario conocer si a la hora de realizar la suma del valor del registro de indexado más el byte de *offset* (en el caso de que el direccionamiento tenga un *offset* de 16 bits se suma al byte menos significativo) se ha producido un acarreo en la misma ya que, en el caso de que se haya producido, hay que tenerlo en cuenta a la hora de formar la dirección efectiva y sumarles una unidad a los 5 bits restantes que completan esta dirección.

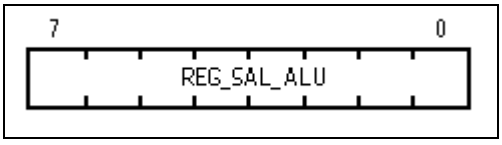


Figura 3.38 Registro auxiliar reg_sal_alu.

IMPLEMENTACIÓN

Este es un registro síncrono compuesto por 9 biestables (8 para el resultado de la ALU y 1 para el bit de acarreo) y con una señal asíncrona de *reset* que lo inicializa al valor $00. Con cada flanco de subida de la señal de reloj se almacena en el registro los datos que hay en sus entrada. Este registro a su salida se encuentra conectado al bus de datos y al bus de direcciones. Los pines de E/S a este registro auxiliar son los que se muestran en la Figura 3.39 y se enumeran a continuación:

PINES DE ENTRADA:

- **dat_ent_reg<7:0>** : dato de entrada al registro reg_sal_alu.
- **clock** : señal de reloj del registro al ser un componente síncrono.
- **dat_ent_carry** : dato de entrada al registro que refleja el resultado del bit C después de ser modificado por la ALU.
- **reset** : señal asíncrona que hace que el registro se cargue a un valor inicial.

PINES DE SALIDA:

- **dat_sal_reg<7:0>** : dato de salida del registro.
- **dat_sal_carry<7:0>** dato de salida del resultado del bit C.

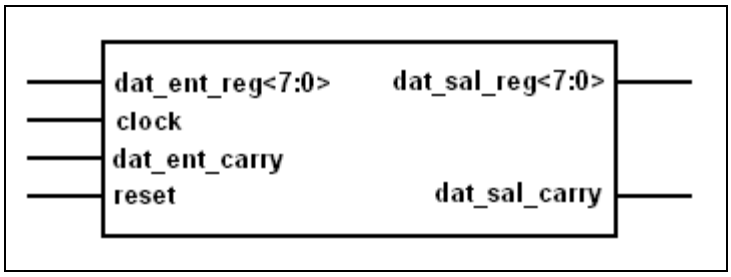


Figura 3.39 Pines de E/S del registro auxiliar reg_sal_alu.

El código fuente de este registro se encuentra en el fichero **reg_sal_alu.vhd** del Capítulo 9 Anexo 9.2, y el resultado de la síntesis del mismo se muestra en la Figura 3.40.

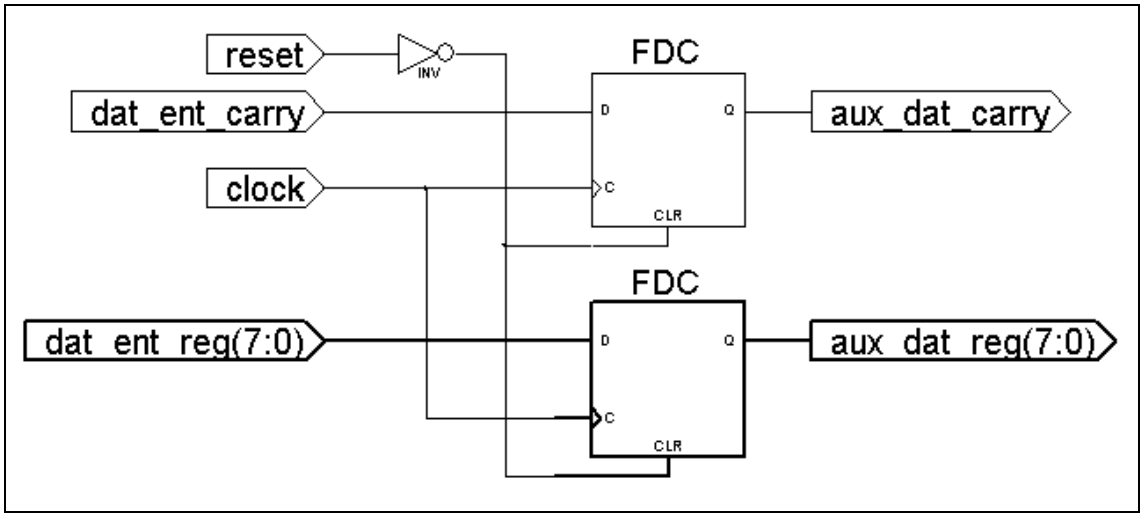


Figura 3.40 Resultado de la síntesis del registro auxiliar reg_sal_alu.

3.3.4.3. REG_DIR

Este registro llamado reg_dir se muestra en la Figura 3.41 y es el que se encarga de almacenar la dirección de la memoria cuando se produce una operación de salto a subrutina. En este tipo de operaciones los operandos que siguen al código de instrucción indican la dirección o el desplazamiento relativo al valor del contador de programa donde se encuentra la subrutina y, por lo tanto, donde debe continuar la secuencia de programa. Estas instrucciones almacenan en la pila el valor del contador de programa donde se debería seguir la secuencia de programa después de una retorno de subrutina. El problema está en que primero se debe calcular la dirección efectiva para que el contador de programa se incremente y se sitúe en el valor de la dirección de memoria la instrucción siguiente, siendo esta la que hay que almacenar en la pila. Pero entonces, hasta que no se haya guardado el valor de esta dirección en la pila, no se puede almacenar en el contador de programa la dirección efectiva del salto a subrutina previamente obtenida. Luego es necesario un registro que almacene este valor hasta que se le pueda asignar al contador de programa. Esta es la misión del registro reg_dir.

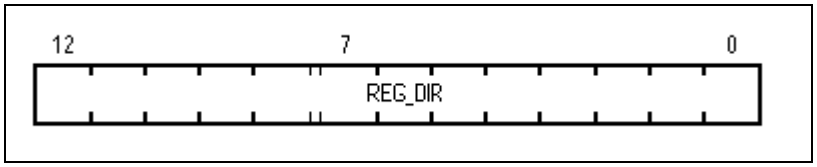


Figura 3.41 Registro auxiliar reg_dir.

IMPLEMENTACIÓN

Este registro se compone de 13 biestables síncronos con una señal asíncrona de *reset* que inicializa su valor a $00. Cuando en la señal de reloj se produce un flanco de subida y la señal que indica la carga del registro se encuentra activada, se permite que el dato de entrada sea almacenado en el registro.

Los pines de E/S de este registro se muestran en la Figura 3.42 y tienen asignados las siguientes señales que se enumeran a continuación:

PINES DE ENTRADA:

- **dat_ent_reg_dir<12:0>** : dato de entrada al registro reg_dir.
- **carga_reg_dir** : permite la carga de este registro.
- **clock** : señal de reloj del registro al ser un componente síncrono.
- **reset** : señal asíncrona que hace que el registro se cargue a un valor inicial.

PINES DE SALIDA:

- **dat_sal_reg<12:0>** : dato de salida del registro.

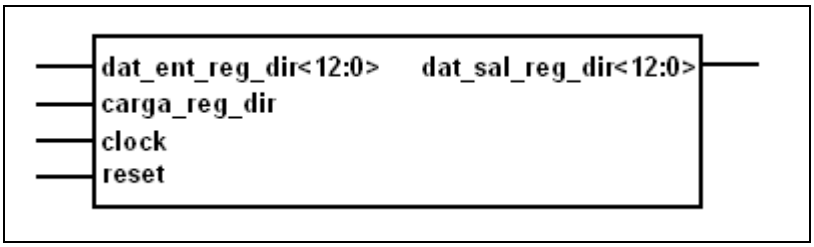


Figura 3.42 Pines de E/S del registro auxiliar reg_dir.

El código fuente de este registro se encuentra en el fichero **reg_dir.vhd** del Capítulo 9 Anexo 9.2, y el resultado de la síntesis del mismo se puede ver en la Figura 3.43.

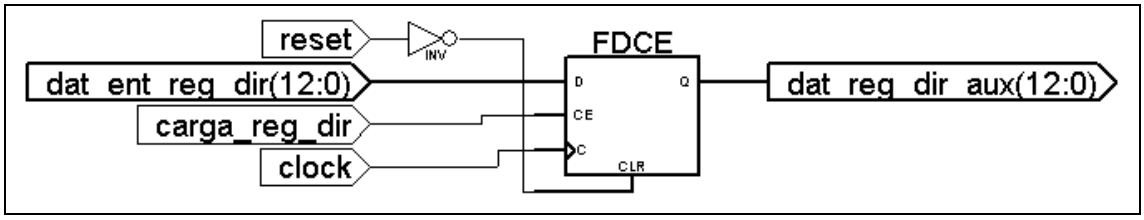


Figura 3.43 Resultado se la síntesis del registro auxiliar reg_dir.

3.3.5. UNIDAD ARITMÉTICO LÓGICA (ALU)

La unidad aritmético lógica, más conocida por las siglas ALU, es un circuito combinacional encargado de realizar todos los cálculos que el juego de instrucciones del microcontrolador exige a la CPU para que estas sean ejecutadas correctamente.

Las operaciones que debe efectuar la ALU para esta familia de microcontroladores son las siguientes:

- Lógicas: AND, OR, NOT, OR, XOR.
- Aritméticas: la adicción, la sustracción, el producto, el incremento, y el decremento.
- Operaciones con bits: Poner un determinado bit al valor lógico ‘0’ o ‘1’.
- De desplazamiento de bits:
    - Desplazamiento aritmético: si se duplica el bit de signo y se pierde el último bit.
    - Desplazamiento lógico: cuando entra un ‘0’ o un ‘1’ y se pierde el último o el primer bit.
    - Rotación: si el último bit pasa a ser el primero o viceversa. Normalmente estas operaciones se realizan teniendo en cuenta el bit C del registro de estado.

Para realizar todas estas operaciones, la unidad aritmético lógica está formada por una serie de operadores, tales como sumadores-restadores, además de comparadores y otra serie de circuitos lógicos. Las entradas a cada uno de estos circuitos son dos datos de 8 bits, más ciertas señales de control que le indican qué tipo operación ha de realizar así como, si esta requiere el bit de acarreo, los bits sobre los cuáles se opera, etc. Toda esta información es generada por el circuito de control y viene dada en función de la instrucción en curso y el estado en el que se encuentre la CPU. Una vez que ha operado la ALU almacena el resultado en un registro auxiliar del cuál se tomará el mismo cuando la unidad de control lo indique.

La ALU, a su vez, también se encarga de actualizar el registro de estado cada vez que una operación es realizada. De modo que además de disponer de los circuitos anteriormente comentados, también debe poseer la lógica necesaria para que una vez que haya obtenido un determinado resultado, le envíe las señales correspondientes al registro de estado para que este modifique sus valores si alguno de ellos ha cambiado.

Estas señales determinarán si el resultado ha sido positivo, negativo o cero, si lleva acarreo o no tanto si se trata del intermedio como del general.

IMPLEMENTACIÓN

Para llegar a tener un módulo que realice todas las tareas de las que debe hacerse cargo la ALU, es necesario ir por partes.

La manera de diseñar la parte de la ALU que realiza las operaciones de suma y resta ha sido a partir de un sumador total de un bit. Una vez diseñado este, se han instanciado y unido varios de estos sumadores en un componente de una jerarquía mayor hasta completar un sumador total de 8 bits de manera que el tamaño de los datos sea el que va a utilizar esta unidad. Como es necesario realizar también la operación de sustracción, el siguiente paso es convertir al sumador total de 8 bits en un sumador-restador. Una vez que se dispone de este bloque, ya se puede instanciar éste dentro del módulo ALU, de manera que así se resuelve el modo de actuar de unas cuantas operaciones como son la adicción, la sustracción, el incremento y el decremento, donde, con sólo variar los datos de entrada a la ALU y con alguna señal que proporcione la unidad de control que le indique el tipo de operación, se pueden resolver muchos de los cálculos que conlleva algunas de las instrucciones del microcontrolador.

La razón de diseñar los sumadores y restadores a partir de sumadores de un bit, para más tarde diseñar el de 8 bits y así sucesivamente hasta llegar al sumado-restador de 8 bits en vez de hacerlo simplemente con el signo de sumar o restar en el código fuente donde sea necesario, es para no utilizar más sumadores de los estrictamente necesarios y ocupar el menor área a la hora de sintetizar.

Como el microcontrolador dispone de las operaciones de suma con o sin acarreo al igual que la resta, también la ALU se ha diseñado de manera que todas las opciones sean posibles con las correspondientes órdenes dadas por señales de la unidad de control. Para ello ha sido necesario que el sumador-restador tenga a continuación otro más sencillo por si se desea realizar una resta con acarreo, ya que en esta operación se utiliza el acarreo para poder realizar el complemento a dos del segundo sumando y es necesario colocar este otro sumador en serie para que además reste el acarreo.

Por otro lado hay que tener en cuenta que el sumador es un circuito combinacional que siempre va a estar sumando lo que haya en sus entradas por lo que hay que llevar un control de lo que le llega. Para ello se dispone de una señal que

solamente deja pasar el dato de entrada si la unidad de control lo requiere, si no, hace que el dato de entrada sea el valor $00 . De esta manera se evitan errores en la suma.

Para poder realizar el producto se ha utilizado simplemente el operador que está contenido en una de las librerías de las que dispone el lenguaje VHDL. En el caso de este microcontrolador ha sido necesario añadir lógica adicional debido a que la salida sólo es de 8 bits y el resultado de una multiplicación de dos bytes requiere un mayor número de bits. De esta forma se puede obtener a la salida el resultado deseado por medio de una señal que indica si queremos obtener los bits más o menos significativos del resultado.

Para las operaciones de desplazamiento de bits sólo ha sido necesario implementar la rotación tanto hacia la derecha como hacia la izquierda. Estas se han diseñado simplemente por medio de la asignación de señales. Para efectuar un tipo de desplazamiento de bits u otro, se ha hecho de tal forma que sea la unidad de control la que decida qué tipo de bits participan dependiendo de la instrucción a ejecutar.

Para las instrucciones lógicas se han utilizado los operadores lógicos de los que dispone ya el lenguaje VHDL en sus librerías, como son: AND, OR, NOT y XOR. Estos también se utilizan en las operaciones de manipulación de bits. Cuando se requiere dar a un bit un determinado valor lógico, la ALU aplica una máscara al operando correspondiente por medio de estos operadores. La unidad de control es la que le proporciona el dato del bit sobre el que se quiere operar.

Este microcontrolador también dispone de instrucciones que cargan un dato directamente desde la memoria a alguno de los registros de la CPU como el acumulador o el registro de indexado. Esto no seria posible sino se hace de forma que el dato pase por la ALU ya que estos registros se han diseñado de manera que no pueden ser cargados directamente desde el bus de datos. Para ello la ALU debe pasar el dato a través de ella sin que sea modificado. Esto se realiza asignando la señal de entrada directamente a la salida. Esto es lo que se llama que la ALU esté en modo transparente.

Otra de las operaciones de las que tiene que disponer la ALU es la operación de *set* y de *reset*. La primera de ellas se trata simplemente de que el bit C del registro de estado se ponga a ‘1’, mientras que la segunda lo que hace es borrar todos los bits del registro de estado incluido, es decir, que valgan todos ‘0’.

Una vez que se la ALU realiza la correspondiente operación, guarda los resultados en señales auxiliares, que luego hay que asignar a la salida dependiendo de la

misma. Si es una operación aritmética la salida vendrá del sumador excepto en el caso del producto, pero si es otro tipo de operación la salida será otra señal auxiliar.

Por otra parte se ha comentado que la ALU es la encargada de obtener los nuevos valores del registro de estado en cada operación. Para ello se utiliza la lógica necesaria de manera que mediante operadores lógicos con los bits de la salida del resultado de la ALU en ese momento, se calculen los correspondientes valores del bit N, el bit Z, el bit H y el bit C. Para este último una de las formas de calcularlo si se trata de operaciones aritméticas, es que la salida en vez de 8 bits sea de 9, de manera que si se desborda el resultado esto se refleje en el noveno bit.

Las diferentes entidades de menor jerarquía que forman el módulo de la ALU como son el sumador total de 1 bit y el sumador total de 8 bits ya se han comentado anteriormente. Una vez que se dispone del sumador total de 8 bits, es necesario que este sea capaz de sumar, restar, incrementar y decrementar. Para ello se ha diseñado un bloque de manera que convierte el sumador total de 8 bits en un sumador-restador. En este bloque se ha instanciado el sumador de 8 bits y se le han asignado las señales correspondientes a sus entradas para que actúe de manera correcta en cada caso.

Los pines de E/S que componen el sumador-restador son los que se muestran en la Figura 3.44 y son los que se detallan a continuación:

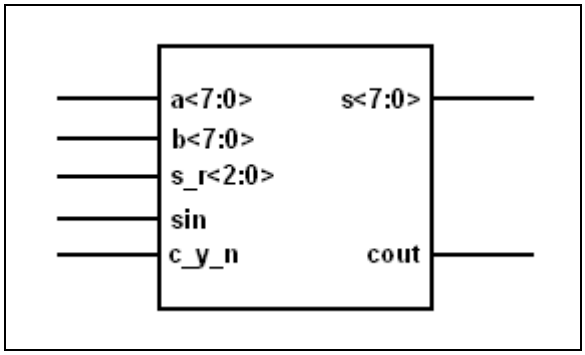


Figura 3.44 Pines de E/S del sumador-restador de la ALU.

PINES DE ENTRADA:

- **a<7:0>** : dato de entrada. Es el operando A.
- **b<7:0>** : dato de entrada. El operando B.
- **cin** : es el acarreo de entrada.
- **s_r** : esta señal indica si la operación a realizar es la suma o la resta.

PINES DE SALIDA:

- **s<7:0>** : dato de salida. Es el resultado de la suma.
- **cout** : acarreo de salida.

El código fuente del sumador-restador que contiene la ALU se puede observar en el fichero **sum_res.vhd** en el Capítulo 9 Anexo 9.2, y el resultado de sintetizar este código se muestra en la Figura 3.45.

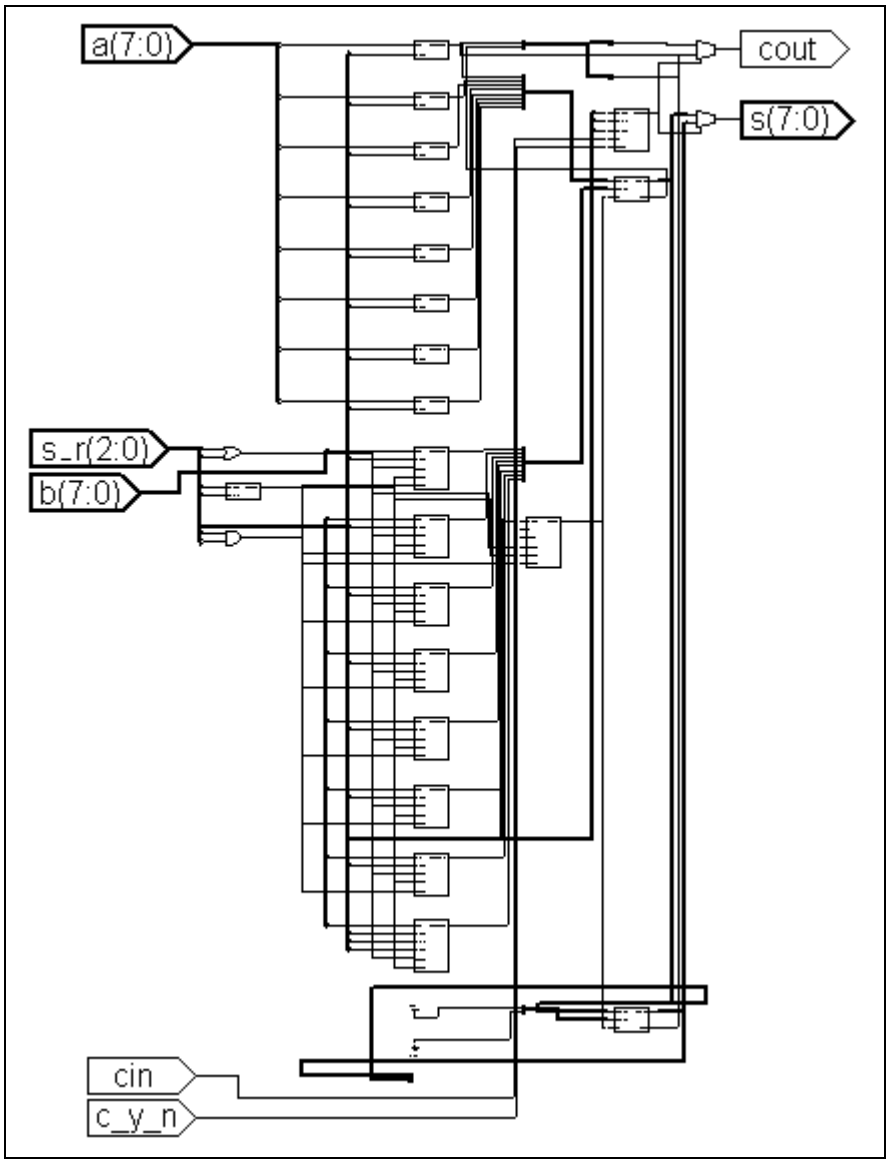


Figura 3.45 Resultado de la síntesis del circuito sumador-restador de la ALU.

Una vez que ya se dispone de este circuito sólo es necesario instanciarlo dentro de la entidad superior ALU y asignarle las señales necesarias a los pines de E/S. Esto se realiza junto con la implementación de los demás posibilidades que debe tener la ALU de las que ya se ha hablado. Los pines del componente de la ALU que se pueden observar en la Figura 3.46 y son los siguientes:

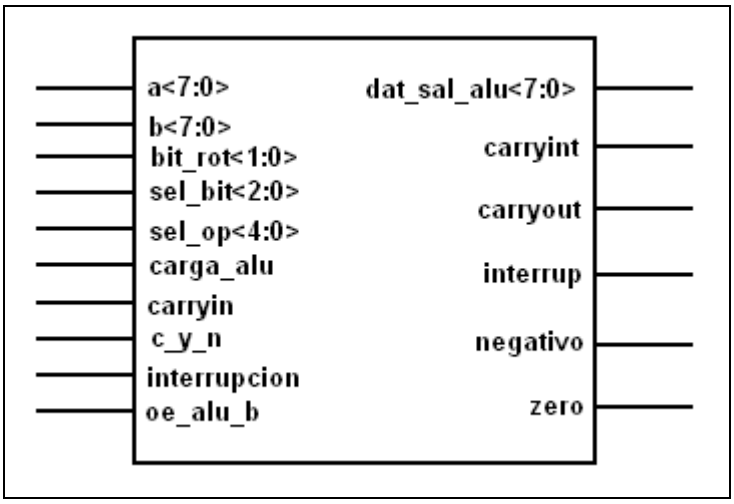


Figura 3.46 Pines de la ALU.

PINES DE ENTRADA:

- **a<7:0>** : dato de entrada a la ALU.
- **b<7:0>** : dato de entrada a la ALU. Esta señal y la anterior son los dos operandos que de la ALU.
- **bit_rot<1:0>** : esta señal se utiliza en la operaciones de desplazamiento y rotación de bits e indica el valor que se adjudicará en cada caso al bit más o menos significativo si la rotación o el desplazamiento se realiza hacia la derecha o izquierda respectivamente.
- **sel_bit <2:0>** : esta señal se utiliza en las operaciones de manipulación de bits e indica cúal de los 8 bits del dato es sobre el que se va a operar.
- **carga_alu** : permite la carga del registro.
- **carryin** : acarreo de entrada.
- **c_y_n** : esta señal se utiliza en las operaciones de suma y resta y es la que indica si estas van a ser con o sin acarreo.
- **interrupcion** : dato de entrada que muestra el valor del bit I del registro de estado.

- **oe_alu_b** : esta señal permite la carga del operando B. Esto se debe a que los sumadores que tiene instanciados la ALU para realizar determinadas operaciones son combinacionales y suman cualquier dato que se encuentre en sus entradas continuamente. Al operando B le llegan los datos del bus de datos que muchas veces se encuentra a alta impedancia. Si se realiza la suma siendo uno se los operandos una alta impedancia esto causa un error, es por ello por lo que se necesita un control de los datos que le llegan al operando B.

PINES DE SALIDA:

- **dat_sal_alu<7:0>** : dato de salida de la ALU.
- **carryint** : salida de la ALU que modifica el bit H del registro de estado.
- **carryout** : salida de la ALU que modifica el bit C del registro de estado.
- **interrup** : salida de la ALU que modifica el bit I del registro de estado.
- **negativo** : salida de la ALU que modifica el bit N del registro de estado.
- **zero** : salida de la ALU que modifica el bit Z del registro de estado.

El código fuente en el que se ha implementado la ALU se encuentra en el fichero **alu_sum_res.vhd** del Capítulo 9 Anexo 9.2.

3.3.6. BUSES DE COMUNICACIÓN ENTRE REGISTROS

Un bus es el lugar físico por el que transita la información en el microcontrolador. Es un conjunto de hilos conductores donde, en cada uno de ellos, se hace presente la información de un bit, (definiéndose el bit como la unidad elemental de información). Los buses son los encargados de las conexiones entre las distintas unidades funcionales del microcontrolador. Dentro de este microcontrolador hay tres tipos de buses: el bus de direcciones, el de datos y el bus de control.

3.3.6.1. BUS DE DIRECCIONES

El bus de direcciones es el que unifica todos los registros que tenga que ver con la dirección de la memoria a la que la CPU quiere tener acceso y el que transporta el

valor de esta dirección hasta el registro MAR. El número de líneas de este bus viene dado por el tamaño que tenga la memoria del microcontrolador. Como ya se vio antes este tamaño cambia dependiendo del miembro de la familia que se trate ya que la cantidad de memoria de unos componentes a otros varía considerablemente. En la realización del presente proyecto se ha diseñado un núcleo para un dispositivo que tenga 8 Kbytes de memoria en total, por lo que esto supone que el bus de direcciones disponga de 13 líneas. Esto concuerda a su vez con el tamaño del contador de programa, del registro MAR y del registro auxiliar reg_dir.

IMPLEMENTACIÓN

La manera mediante la que se ha creado el bus de direcciones ha sido declarar una señal denominada bus_direccion del tamaño de los datos que se van a enviar por ella, es decir de 13 bits, y se le han ido asignando las diferentes señales que van conectadas a él. Las señales de entrada al bus, es decir las señales de salida de los diferentes registros son asignadas al bus mediante un multiplexor con prioridad controlado mediante señales de *output enable* gobernadas por la unidad de control. Por el contrario las señales de salida del bus, es decir las señales de entrada de los diferentes registros, son asignadas en todo momento y es la unidad de control la que decide la carga de cada uno de ellos. Si no se le asigna ningún dato al bus este debe permanecer en alta impedancia.

Las señales conectadas al bus de direcciones son las que se detallan a continuación:

SEÑALES DE ENTRADA AL BUS:

- aux_sal_pc: dato de salida del contador de programa.
- aux_sal_reg_dir: dato de salida del registro auxiliar reg_dir. SE utiliza en las operaciones de salto a subrutina para guardar el valor de la dirección efectiva.
- aux_sal_reg_sal_alu: dato de salida del registro auxiliar reg_sal_alu. Es necesaria para cierto tipos de direccionamiento como el directo, extendido, indexado y el relativo.
- aux_sal_sp: dato de salida del puntero de pila. Se utiliza en las operaciones en las que se almacena un dato en la pila.

- aux_dat_reg_aux: dato de salida del registro auxiliar reg_aux. Se utiliza para formar la dirección efectiva cuando se trata del direccionamiento extendido e indexado con *offset* de 16 bits.
- Diferentes valores constantes como son los vectores de interrupción y el vector de *reset*. Se utilizan en los procesos de atención a las interrupciones.

SEÑALES DE SALIDA DEL BUS:

- dat_ent_pc: dato de entrada al contador de programa.
- dat_ent_mar: dato de entrada al registro MAR.
- dat_ent_reg_dir: dato de entrada al registro auxiliar reg_dir.
- aux_alu_a: dato de entrada a la ALU. Es necesaria en las intrucciones donde se requiere guardar el valor del contador de programa en la pila.

3.3.6.2. BUS DE DATOS

Es el encargado de transportar la información de los datos entre los diferentes registros que componen el núcleo. El número de líneas de que dispone esta directamente relacionado con la longitud de palabra que maneja el microcontrolador, es decir, el bus de datos del 68HC05 es de 8 líneas.

IMPLEMENTACIÓN

La implementación del bus de datos es similar a la comentada en el caso del bus de direcciones. En este caso se declara una señal llamada bus_datos del tamaño del bus, es decir de 8 bits y se le asignan las señales correspondientes. Las salidas de registros conectadas al bus van multiplexadas y las entradas de registros conectadas a él van asignadas directamente siendo de nuevo la unidad de control la que se encarga de que no exista colisión entre los datos. En el caso de que ningún dato es asignado al bus este permanece en alta impedancia.

Las señales conectadas al bus de datos son las que se enumeran a continuación:

SEÑALES DE ENTRADA AL BUS:

- aux_sal_reg_sal_alu: dato de salida del registro auxiliar reg_sal_alu.
- aux_sal_mbr: dato de salida del registro MBR.

SEÑALES DE SALIDA DEL BUS:

- dat_ent_a: dato de entrada al acumulador.
- dat_ent_mbr_b: dato de entrada al registro MBR.
- dat_ent_x: dato de entrada al registro de indexado.
- dat_ent_reg_aux: dato de entrada al registro auxiliar reg_aux.
- aux_alu_a: dato de entrada al operando A de la ALU.
- aux_alu_b: dato de entrada al operando B de la ALU.
- aux_carryout: dato de entrada al bit C del registro de estado.
- aux_carryint: dato de entrada al bit H del registro de estado.
- aux_zero: dato de entrada al bit Z del registro de estado.
- aux_negativo: dato de entrada al bit N del registro de estado.
- aux_interrup: dato de entrada al bit I del registro de estado.
- aux_ent_ri: dato de entrada al registro de instrucción.

3.3.6.3. BUS DE CONTROL

El bus de control es un bus algo diferente a los dos anteriores. Normalmente se denomina bus de control al conjunto de todas las señales que envía la unidad de control a los diferentes registros para la ejecución ordenada de todas las operaciones que debe realizar el núcleo del microcontrolador. Las señales que componen este bus coinciden con las salidas de la unidad de control por lo que se comentarán en la explicación del correspondiente apartado.

3.4. UNIDAD DE CONTROL

La unidad de control es la encargada de coordinar a todos los registros de la ruta de datos de manera que se realicen todos los procesos necesarios de forma ordenada para ejecutar cada una de las instrucciones del programa almacenado en la memoria del microcontrolador. Dependiendo de la instrucción actual los procedimientos serán muy diferentes. Es por ello que la unidad de control se compone de un decodificador de instrucciones cuyas salidas son las señales de control de todos los elementos que conforman la ruta de datos.

3.4.1. DECODIFICADOR DE INSTRUCCIONES

Este módulo es el que forma la unidad de control. Recibe la instrucción almacenada en el registro de instrucción y según su valor, activa las señales de salida correspondientes siguiendo un determinado orden. De esta forma el microcontrolador ejecuta todas las instrucciones que se le indican en el programa almacenado en su memoria.

El decodificador de instrucciones se basa en una máquina de estados donde en cada estado se cargan los datos en unos registros y se extraen de otros, activando y desactivando las señales correspondientes.

IMPLEMENTACIÓN

La forma de implementar este módulo como ya se ha mencionado antes ha sido mediante una máquina de estados. Se trata de un registro síncrono que dispone de una señal "actual", que con cada flanco de subida de la señal de reloj se actualiza con el valor de un nuevo estado almacenado en otra señal llamada "siguiente". Junto con este registro se han creado otros de forma similar que almacenan variables internas del módulo para tomar decisiones sobre la secuencia de estados. Todos ellos llevan una señal asíncrona de *reset* que inicializa la máquina al primero de los estados y el resto de los registros a los valores iniciales deseados.

Los pines del componente que se pueden observar en la Figura 3.47 y sus funciones son las siguientes:

PINES DE ENTRADA:

- **ccr<4:0>** : señal que indica los valores de los bits del registro de estado.
- **dat_instr<7:0>** : es la instrucción que se está ejecutando en ese momento y que contiene el registro de instrucción.
- **clock** : señal de reloj del registro al ser un componente síncrono.
- **interrupcion** : señal que indica si ha tenido lugar alguna interrupción *hardware*.
- **IRQ** : es la entrada de la interrupción externa. Es necesaria para tomar decisiones dentro del decodificador.
- **reset** : señal asíncrona que hace que el registro se cargue a un valor inicial.

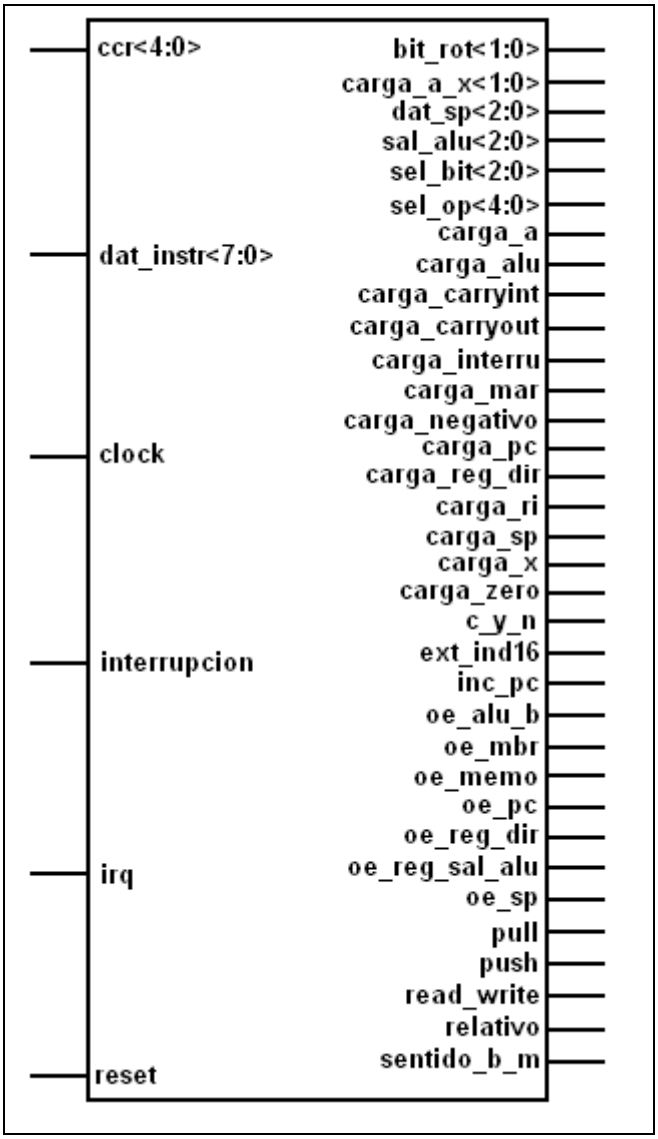


Figura 3.47 Pines de E/S del decodificador de instrucciones.

PINES DE SALIDA:

- **bit_rot<1:0>** : señal que indica a la ALU el valor que debe tomar el bit más o menos significativo en las instrucciones que realizan desplazamientos o rotaciones. Si se realizan hacia la derecha sería el más significativo mientras que si es hacia la izquierda sería el menos significativo.
- **carga_a_x<1:0>** : esta señal determina el dato que se debe cargar en el operando A de la ALU. Este puede ser: o bien el dato almacenado en el acumulador, o en el registro de indexado, o en el registro de estado, o en

el bus de datos, o en el de direcciones, o bien el valor $00 dependiendo de la instrucción que se esté ejecutando.

- **dat_sp<2:0>** : esta señal indica el dato a cargar en el operando A de la ALU en el caso de que se esté ejecutando un salto o un retorno a una subrutina y hay que almacenar datos en la pila.
- **sal_alu<2:0>** : esta señal permite que en el bus de direcciones se carguen diferentes datos para formar la dirección de cada modo de direccionamiento.
- **sel_bit<2:0>** : señal que muestra el bit sobre el cuál la ALU debe operar.
- **sel_op<4:0>** : esta señal indica la operación a realizar por la ALU.
- **carga_a** : permite la carga del acumulador.
- **carga_alu** : permite la carga de la ALU.
- **carga_carryint** : permite la carga del bit H del registro de estado.
- **carga_carryout** : permite la carga del bit C del registro de estado.
- **carga_interru** : permite la carga del bit I del registro de estado.
- **carga_mar** : permite la carga del registro MAR.
- **carga_negativo** : permite la carga del bit N del registro de estado.
- **carga_pc** : permite la carga del contador de programa.
- **carga_reg_dir** : permite la carga del registro auxiliar “reg_dir”.
- **carga_ri** : permite la carga del registro de instrucción.
- **carga_sp** : permite la carga del puntero de pila.
- **carga_x** : permite la carga del registro de indexado.
- **carga_zero** : permite la carga del bit Z del registro de estado.
- **c_y_n** : señal que indica a la ALU si la operación debe utilizar o no el bit de acarreo.
- **ext_ind16** : esta señal permite que se guarde en el registro “reg_aux” los 5 bits menos significativos del dato que se encuentra en el bus de datos en ese momento. Se utiliza en los modos de direccionamiento extendido e indexado con *offset* de 16 bits para formar más adelante la dirección efectiva.
- **inc_pc** : señal que indica al contador de pila que se incremente o bien una unidad o que sume o reste el *offset* relativo si la instrucción a ejecutar tiene este tipo de direccionamiento.

- **oe_alu_b** : *output enable* del operando B de la ALU.
- **oe_mbr** : *output enable* del registro MBR.
- **oe_memo** : *enable* de la memoria con la que intercambia información el núcleo del microcontrolador.
- **oe_pc** : *output enable* del contador de programa.
- **oe_reg_dir** : *output enable* del registro auxiliar "reg_dir".
- **oe_reg_sal_alu** : *output enable* del registro auxiliar "reg_sal_alu".
- **oe_sp** : *output enable* del puntero de pila.
- **pull :** permite que el puntero de pila se incremente en una unidad.
- **push** : permite que el puntero de pila se decremente en una unidad.
- **read_write** : indica a la memoria conectada al núcleo del microcontrolador si desea leer o escribir.
- **relativo** : indica al puntero de pila si debe incrementar el dato en una unidad o por el contrario sumar o restar el dato de entrada. Se utiliza si el direccionamiento es relativo.
- **sentido_b_m** : indica el sentido de los datos en el registro MBR, si el dato va del bus de datos a la memoria o viceversa. Esto depende de si se pretende escribir en ella o leer de ella respectivamente.

El código fuente de este componente viene descrito en el fichero **dec_instr.vhd** en el Capítulo 9 Anexo 9.2.

MÁQUINA DE ESTADOS DEL DECODIFICADOR DE INSTRUCCIONES

La máquina de estados está formada por una serie de estados cuya secuencia depende del código de operación de cada instrucción. A continuación de forma general se van a detallar todos los estados que forman parte de la misma describiendo las señales que se activan en cada uno de ellos.

Una vez comentados todos se incluyen unas figuras en las que se pueden observar las secuencias más importantes que tienen lugar en la máquina de estados como pueden ser: el ciclo de *fetch* de las instrucciones, las secuencias de los diferentes modos de direccionamiento, el proceso de *reset* y para el proceso de salto a una rutina de atención a interrupción.

ESTADOS QUE COMPONEN LA MÁQUINA:

- **reposo**: este estado únicamente tiene lugar si lo que se está ejecutando es un proceso de *reset*. Se encarga de activar la señal interna de la máquina de estados que desencadena un direccionamiento extendido de manera que el núcleo lea los dos bytes que se encuentran dentro del vector de *reset* para formar la dirección de memoria donde se encuentra el programa principal.
- **lectura_dato**: este estado se encarga de realizar la lectura de los datos de la memoria tanto si se trata de un operando como de una instrucción. Este estado a su vez mantiene los valores de las señales auxiliares internas de la máquina de estados.
- **mbr**: este estado se encarga principalmente de tomar el dato que le llega de la memoria y traspasarlo al bus de datos. Para ello marca el sentido que los datos deben tomar dentro del registro MBR. También mantiene los valores de las señales auxiliares internas.
- **dato_a_bus**: este estado se encarga de extraer los datos del registro MBR para que pasen al bus de datos. Es el momento en el que se dispone de ellos en el bus y dependiendo del valor de las señales auxiliares internas de este módulo o de los cuatro bits menos significativos del código de operación se realizarán diferentes acciones. Las señales internas indican, o bien el tipo de direccionamiento que se está realizando o, si el dato se trata de una instrucción.
- **decod_dir**: en este estado dependiendo del modo de direccionamiento que viene dado por los códigos de operación se siguen diferentes secuencias y acciones. Estos van a ser:

  INHERENTE: si se trata de un direccionamiento inherente, la instrucción no tiene ningún operando y la ejecución de cada instrucción es particular de cada una, por lo que las respectivas acciones se comentarán en el capítulo siguiente donde se tratan una a una todas las instrucciones.

  INMEDIATO: en el caso de tratarse de un direccionamiento inmediato el operando se encuentra en la siguiente dirección que indica el contador de programa por lo que hay que realizar una nueva lectura de la memoria. Para ello se activan las señales necesarias que hagan que este valor salga del contador de programa al bus de direcciones y se cargue en el registro MAR.

  DIRECTO: en este caso se activa una señal interna que indica que se está realizando un direccionamiento directo y se le pasa a la memoria la siguiente dirección para

leer el operando de la instrucción. Para ello se activa la salida del contador de programa y se carga el dato en el registro MAR.

EXTENDIDO: en este caso se activa una señal interna que indica que se está realizando un direccionamiento extendido y se le pasa a la memoria la siguiente dirección para leer los operandos de la instrucción activando las mismas señales que en el direccionamiento directo.

INDEXADO SIN OFFSET: cuando tenemos este tipo de direccionamiento en el estado decod_dir se toma el dato del registro de indexado y se le pasa al registro reg_sal_alu haciendo que la ALU se encuentre en modo transparente.

INDEXADO CON OFFSET DE 8 BITS: en este caso se activa una señal interna que indica que se está realizando un direccionamiento indexado y se le pasa a la memoria la siguiente dirección para leer el operando de la instrucción de la misma manera que en los direccionamientos directo y extendido.

INDEXADO CON OFFSET DE 16 BITS: en este caso se activa una señal interna que indica que se está realizando un direccionamiento indexado. También se le pasa a la memoria la siguiente dirección para leer los operandos de la instrucción de la misma forma que en el direccionamiento directo y extendido.

RELATIVO: en este tipo de direccionamiento si la condición es falsa, la secuencia de estados tiene que leer la siguiente instrucción del programa almacenado en memoria, por lo que se activa la señal interna que indica que el siguiente dato de la memoria es una instrucción y se lee una nueva dirección de la misma activando las señales de salida del contador de programa y de carga del registro MAR. En el caso de que la condición sea verdadera, el dato que actualmente se encuentra en el bus de datos se pasa por la ALU en modo transparente y, se almacena en el registro reg_sal_alu ya que este es el *offset* del direccionamiento relativo que más tarde se sumará o restará al valor del contador de programa.

- **direccionamiento**: en este estado se permite la salida del dato almacenado en el registro reg_sal_alu y se le pasa al bus de direcciones añadiendo los bits necesarios hasta completar los 13 bits de la dirección. Una vez que se obtiene la dirección efectiva en el bus de direcciones se lee el siguiente dato de la memoria. Para ello es necesario activar la señal de carga del registro MAR.

- **dato_pc**: en este estado se activan las señales pertinentes para que el contador de programa en un direccionamiento relativo calcule la nueva dirección donde continúa el programa.
- **dato_a_reg**: este estado almacena un dato en un registro. Para ello activa la señal que extrae el dato del registro "reg_sal_alu" y las señales de carga o bien del registro acumulador, o bien del registro de indexado dependiendo de la instrucción. También en este estado hay que activar: la señal que extrae un dato del contador de programa, la que carga el registro MAR con ese dato, y la que indica que el dato de viene a continuación es una nueva instrucción para continuar con la ejecución del programa almacenado en memoria.
- **dato_a_memo**: este estado tiene lugar si la instrucción que se está ejecutando tiene que escribir en la memoria. Para ello activa la señal que hace que el dato del registro reg_sal_alu este disponible en el bus de datos. En el caso de que se esté realizando el salto a una subrutina de interrupción, se conserva el valor de las señales internas que indican el dato que en cada momento hay que almacenar en la pila y se descuenta una unidad del puntero de pila.
- **escribir_dato**: en este estado al igual que el anterior sólo tiene lugar cuando se va a escribir en la memoria. En él se habilita la memoria y se le indica que se va a realizar una operación de escritura en ella dando el valor correspondiente a la señal "read_write" ('1' para leer y '0' para escribir). En el caso de que se encuentre el microcontrolador realizando un salto a una subrutina de interrupción, además de guardar el dato del registro en la pila, hace que se almacene en el registro de salida de la ALU el dato del siguiente registro a guardar poniendo la ALU en modo transparente y seleccionando la salida del mismo.
- **pila**: este estado es a partir del que comienza la secuencia de estados para la ejecución de salto a una subrutina de interrupción. Activa la señal correspondiente para cargar el primer dato a almacenar en la pila en el registro de salida de la ALU, poniendo esta en modo transparente. Como este dato es el byte menos significativo del contador de programa también tiene que permitir la salida de este registro al bus de direcciones y seleccionar este como entrada de la ALU.
- **lect_pila_pc**: este estado tiene lugar dentro de la secuencia de estados de un retorno de subrutina de interrupción. En él se indica al núcleo que lo siguiente que está almacenado en la pila es el dato donde debe continuar la secuencia de programa.

Este estado activa: una señal interna para que la lectura de la pila se realice siguiendo un direccionamiento extendido, la señal que extrae al bus de direcciones la dirección que indica el puntero de pila y, por último, la que carga este dato en el registro MAR.

- **lect_pila**: este estado inicia la secuencia de estados de un retorno de subrutina de interrupción cargando la primera dirección a la que apunta el puntero de pila en el registro MAR. Para ello activa la señal de salida del puntero de pila y la señal de carga del registro MAR.
- **carga_registros**: este estado pertenece también a la secuencia de estados de una subrutina de interrupción y se encarga de permitir la carga de los datos que se extraen de la pila en los diferentes registros. También carga el dato de salida del puntero de pila en el registro MAR para leer el siguiente dato almacenado en la pila.
- **cargar_sp**: este estado también pertenece a la secuencia de un retorno de subrutina de interrupción y lo que hace es incrementar el puntero de pila para obtener los datos almacenados en ella.

Una vez explicados todos los estados se van a mostrar y comentar las secuencias de estados más importantes que se pueden tener lugar:

CICLO DE FETCH: este ciclo se encarga de leer de la memoria una instrucción y almacenarla en el registro de instrucción. La secuencia de estados que sigue se muestra en la Figura 3.48 y se da cada vez que se va a ejecutar la siguiente instrucción del programa que tiene almacenado en memoria el microcontrolador.

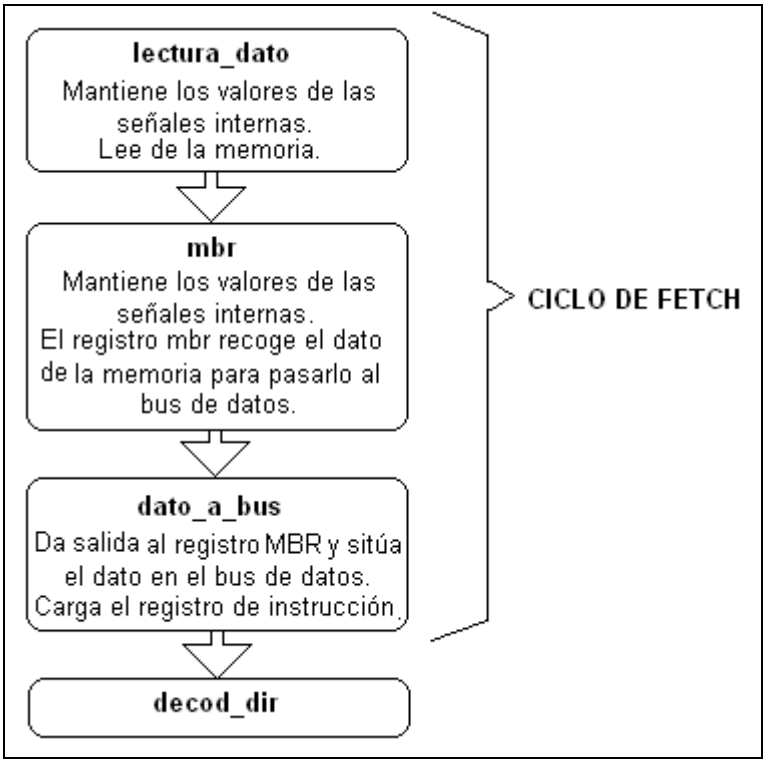


Figura 3.48 Secuencia de estados del ciclo de *fetch*.

MODOS DE DIRECCIONAMIENTO. Cada modo de direccionamiento depende del valor de los diferentes códigos de operación de las instrucciones. Estos códigos vienen dados sistema hexadecimal con lo que se componen por los números del 0 al 9 y por las letras de la A a la F. Como la instrucción es un valor de 8 bits el código de operación se compone por dos valores hexadecimales:

- El primero de los valores expresa el modo de direccionamiento principalmente. En la Tabla 3.2 se puede observar los valores del código de operación que Motorola ha asignado a cada modo de direccionamiento.
- El segundo número o letra indica concretamente la instrucción a realizar.

Tabla 3.2 Modos de direccionamiento con sus respectivos códigos de operación.

MODO DE DIRECCIONAMIENTO	PRIMER VALOR DEL CÓDIGO DE OPERACIÓN
INHERENTE	4, 5, 8 y 9
INMEDIATO	A
DIRECTO	1, 3 y B

EXTENDIDO	C
INDEXADO SIN OFFSET	7 y F
INDEXADO CON OFFSET DE 8 BITS	6 y E
INDEXADO CON OFFSET DE 16 BITS	D
RELATIVO	0 y 2

Una vez que ha concluido el ciclo de *fetch*, se pasa al estado "decod_dir" donde dependiendo del valor de los primeros cuatro bits del código de operación se inicia la secuencia de estados correspondiente a cada uno. Todas estas secuencias se muestran en las siguientes figuras:

- direccionamiento inmediato en la Figura 3.49.
- direccionamiento indexado en la Figura 3.50.
- direccionamiento directo en la Figura 3.51.
- direccionamiento extendido en la Figura 3.52.
- direccionamiento indexado con *offset* de 8 bits en la Figura 3.53.
- direccionamiento indexado con *offset* de 16 bits en la Figura 3.54.
- direccionamiento relativo en la Figura 3.55.

PROCESO DE RESET. Esta secuencia comienza en el momento en que se activa la señal de *reset* y sigue los estados que se le indican en la Figura 3.56.

SALTO A UNA INTERRUPCIÓN. Esta secuencia de estados de muestra en la Figura 3.57 y realiza todo el proceso comentado en el apartado dónde se explican los pasos para atender a una interrupción.

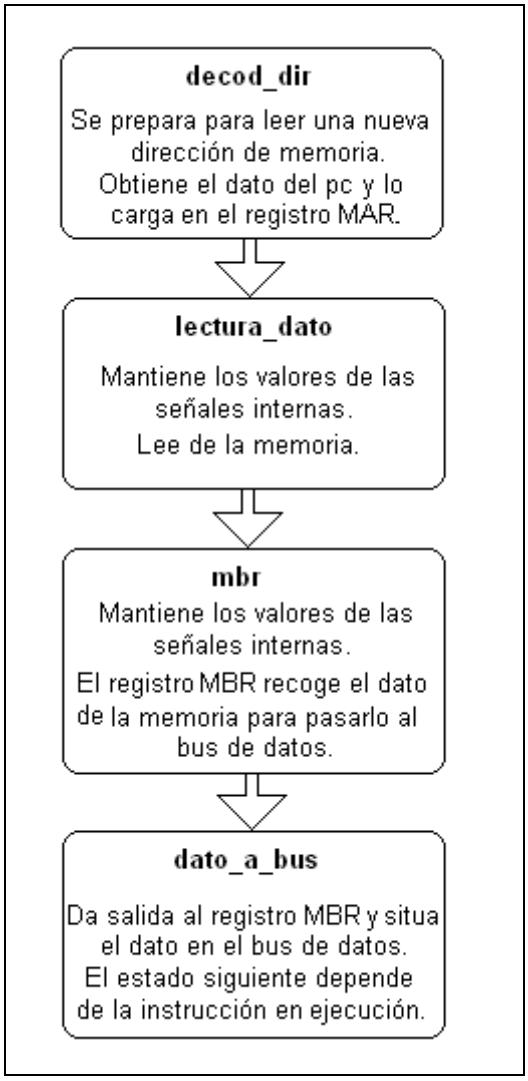


Figura 3.49 Secuencia de estados del direccionamiento inmediato.

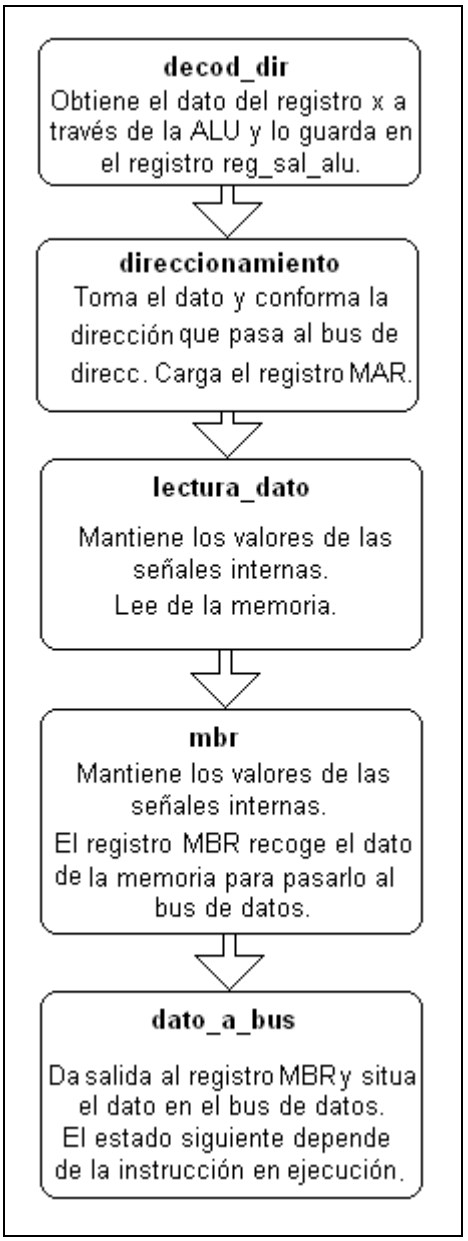


Figura 3.50 Secuencia de estados del direccionamiento indexado.

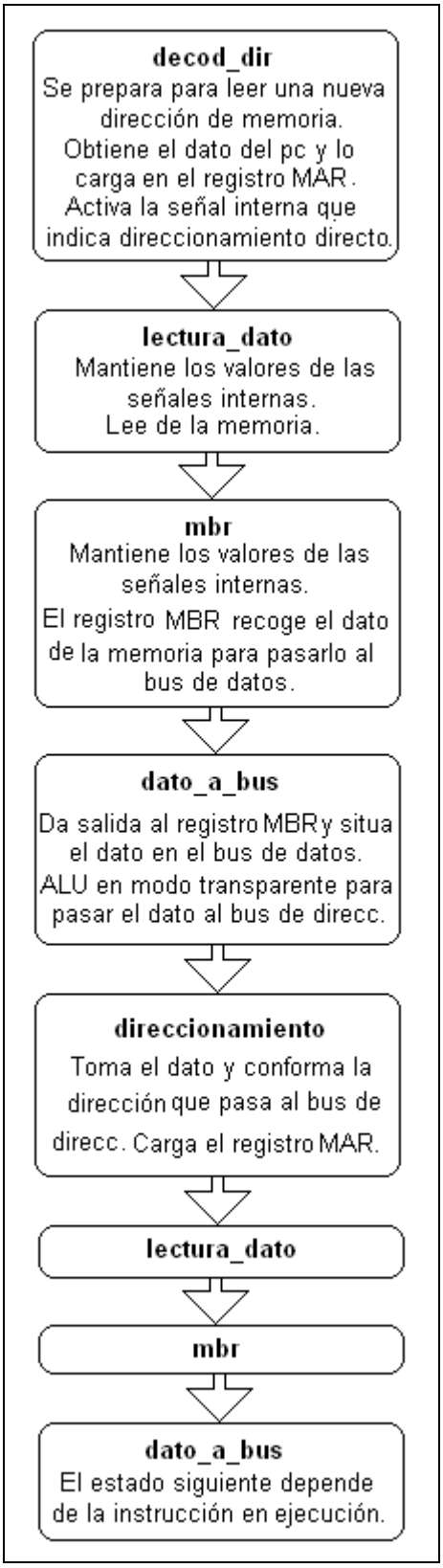


Figura 3.51 Secuencia de estados del direccionamiento directo.

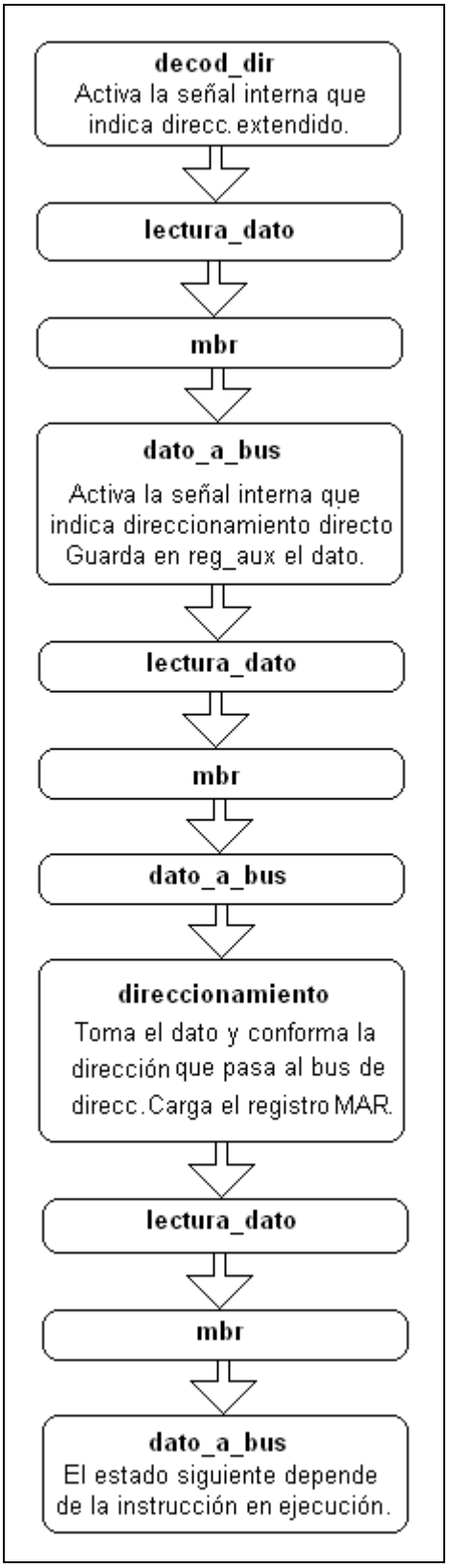


Figura 3.52 Secuencia de estados del direccionamiento extendido.

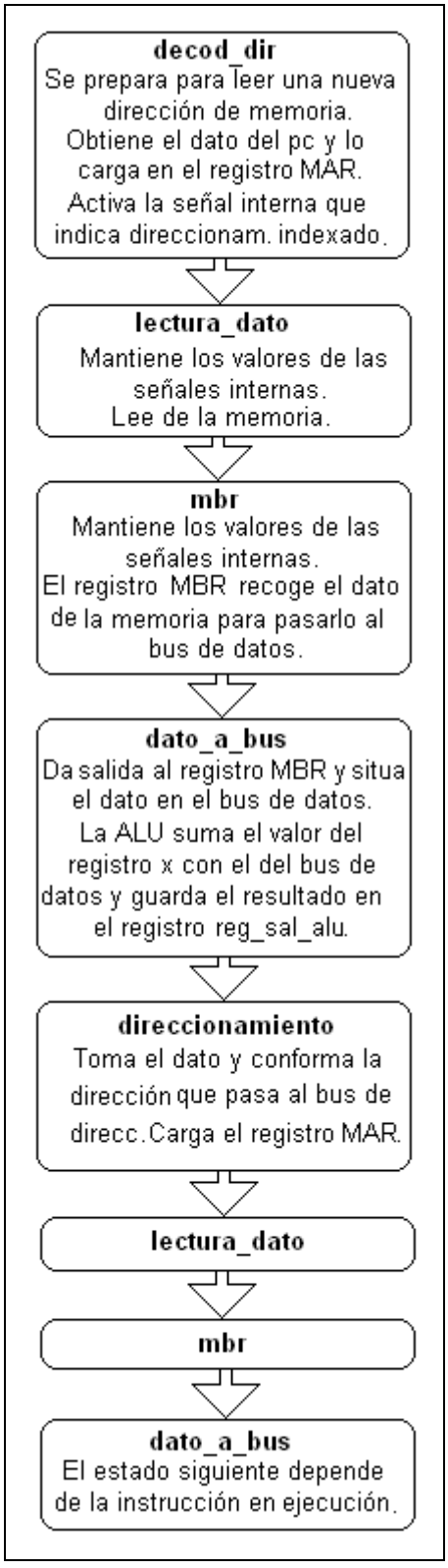


Figura 3.53 Secuencia de estados del direccionamiento indexado con *offset* de 8 bits.

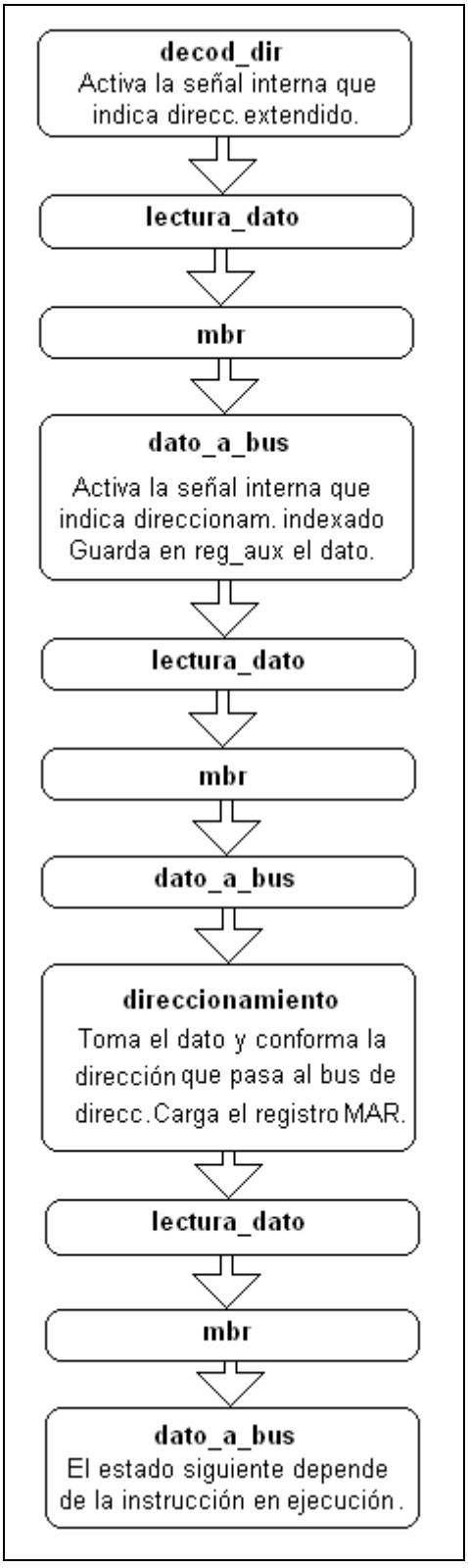


Figura 3.54 Secuencia de estados del direccionamiento indexado con *offset* de 16 bits.

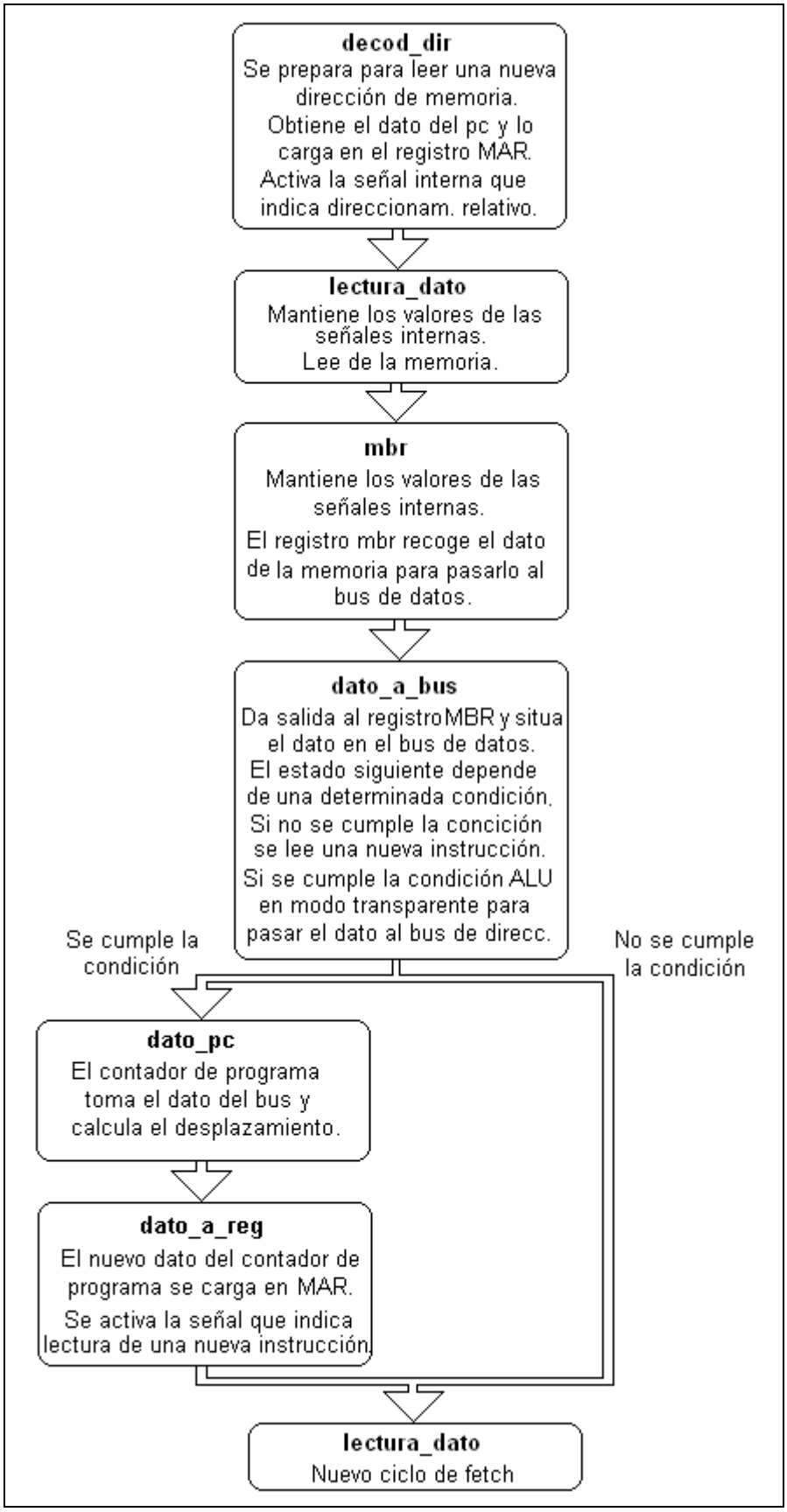


Figura 3.55 Secuencia de estados del direccionamiento relativo.

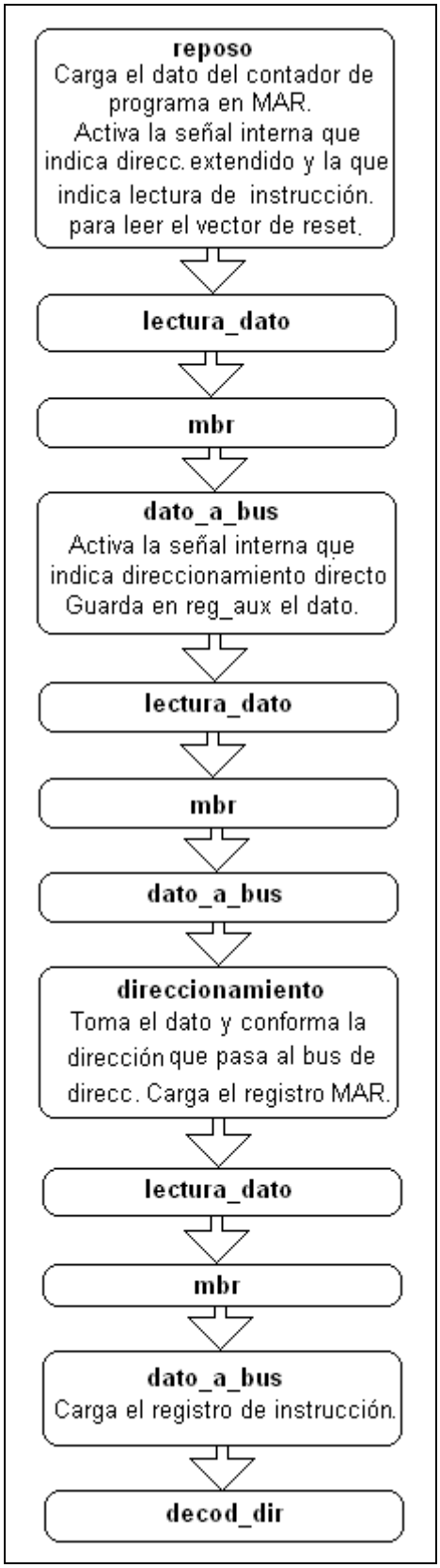


Figura 3.56 Secuencia de estados del proceso de *reset*.

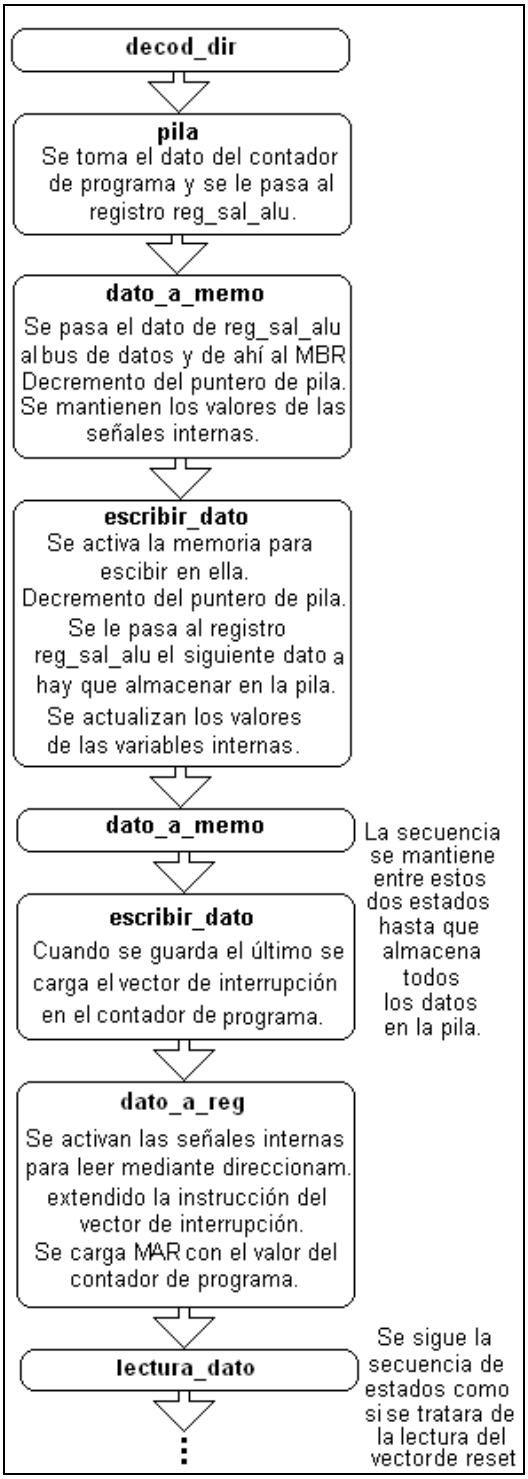


Figura 3.57 Secuencia de estados de un salto a interrupción.

3.5. ARQUITECTURA GENERAL DEL NÚCLEO DEL 68HC05

Una vez que se ha realizado la ruta de datos con todos los registros que la componen y la unidad de control, es necesario unir las dos partes para poder formar lo que es el núcleo del microcontrolador.

Para ello, se han instanciado todos los registros, la ALU, el decodificador de instrucciones que forma la unidad de control y se han declarado las señales que van a formar parte del bus de direcciones y de datos. Una vez hecho esto, se han unido las entradas y salidas de todos los registros con las señales que les corresponden.

Hay que tener en cuenta que hay componentes que tienen entradas a las que le tienen que llegar varias señales por lo que se han creado multiplexores que gestionen las mismas. Estos van a ser comentados a continuación así como la gestión de interrupciones, de los buses de datos y de direcciones, y el bloqueo del reloj bloqueado en las instrucciones de bajo consumo.

- MULTIPLEXOR DE LA ENTRADA AL OPERANDO A DE LA ALU: este decide el dato sobre el que va a operar la ALU. Este puede ser el que contiene el acumulador, el registro de indexado, el bus de datos, el registro de estado o el del contador de programa. El control de este multiplexor se hace desde el decodificador de instrucciones.
- MULTIPLEXOR DE LA ENTRADA AL OPERANDO B DE LA ALU: es muy similar al multiplexor del operando A, nada más que al operando B sólo le llegan datos del acumulador, del registro de indexado y del bus de datos.
- MULTIPLEXOR DE LA ENTRADA AL REGISTRO DE INSTRUCCIÓN: este multiplexor decide si el dato que se carga en el registro de instrucción es el dato del bus de datos o un valor constante. Esto depende de la instrucción que se esté ejecutando en ese momento y de si se ha producido o no una interrupción.

  En el caso de que haya tenido lugar una interrupción, el valor constante que se carga es $82. Este valor es un código de operación libre, es decir, no pertenece a ninguna instrucción y es el que se ha elegido para que la unidad de control ejecute la atención a la interrupción como si se tratase de otra instrucción.
- MULTIPLEXOR DE LA ENTRADA AL REGISTRO DE ESTADO: este multiplexor selecciona la entrada a este registro entre las señales provenientes de la ALU y el bus de datos. Todo depende si el microcontrolador está ejecutando una

instrucción cualquiera o la instrucción de retorno de una interrupción (cuyo código de operación es $80) respectivamente.

- GESTIÓN DEL BUS DE DIRECCIONES: es un multiplexor con prioridad que dependiendo de las órdenes de la unidad de control deja pasar un único dato al bus de direcciones. Estos datos pueden venir de cualquiera de las señales de salida de los diferentes registros que se encuentran conectados a él y que ya han sido comentadas en el apartado del bus de direcciones.
- GESTIÓN DEL BUS DE DATOS: la gestión del bus de datos es idéntica a la del bus de direcciones y los registros cuyas salidas van a parar al bus de datos son lo que se han comentado previamente en el apartado del bus de datos.
- RESET DEL PUNTERO DE PILA: es un multiplexor que se utiliza en la instrucción que produce un *reset* del puntero de pila (cuyo código de operación es $9C). Lo que hace es que a parte del *reset* general del microcontrolador, crea una señal de *reset* especial para el puntero de pila que se activa tanto si en el registro de instrucción esté almacenado el código de operación de esta instrucción, como si se produce un *reset* en todo el microcontrolador.
- RELOJ DEL MICROCONTROLADOR: también se dispone de una lógica combinacional que hace que cuando el núcleo se encuentre en modo bajo consumo, es decir ,ejecutando las operaciones WAIT y STOP, se pare la señal de reloj que sincroniza todos los registros (WAIT) o, además de esta, el oscilador del microcontrolador (STOP).
- CONTROL DEL RELOJ DEL MICROCONTROLADOR: es un registro que dependiendo de: si se está ejecutando una instrucción de bajo consumo, se encuentra habilitada la máscara de interrupción y, se ha producido una interrupción, activa o no las señales que controlan la señal de reloj que va a todos los registros del núcleo, o el oscilador del microcontrolador al completo.
- REGISTRO DE INTERRUPCIONES: es un registro que da valor a unas señales auxilares que indican si se ha producido o no una interrupción y el tipo de interrupción se ha producido respetando el orden de prioridad entre ellas.
- INTERRUPCIÓN: es una señal que resulta de realizar lógica combinacional entre todas las interrupciones e indica si se ha producido cualquiera de ellas.

El resultado final es un componente que realiza las funciones del núcleo del microcontrolador 68HC05. Su esquema de bloques general se puede ver en la Figura 3.59.

Los pines del componente que forma el núcleo del microcontrolador se muestran en la Figura 3.58 y son los siguientes:

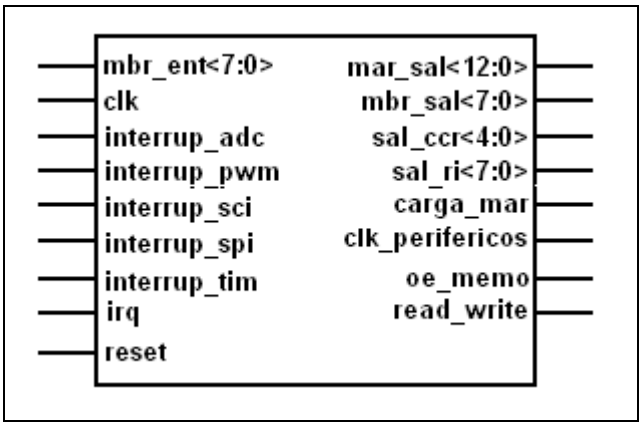


Figura 3.58 Pines de E/S del núcleo del microcontrolador.

PINES DE ENTRADA:

- **mbr_ent<7:0>** : dato de entrada al registro MBR. Es el dato que le llega al núcleo cuando este realiza una lectura de la memoria.
- **clk** : señal de reloj del núcleo del microcontrolador.
- **interrup_adc** : señal de interrupción del convertidor analógico-digital.
- **interrup_pwm** : señal de interrupción del generador PWM.
- **interrup_sci** : señal de interrupción del puerto serie asíncrono.
- **interrup_spi** : señal de interrupción del puerto serie síncrono.
- **interrup_tim** : señal de interrupción del temporizador.
- **irq** : señal de la interrupción externa IRQ.
- **reset** : señal asíncrona que hace que se inicialicen todos los registros del núcleo.

PINES DE SALIDA:

- **mar_sal<12:0>** : dato de salida del registro MAR. Es la dirección de memoria que se va a leer o en la que se va a escribir.

- **mbr_sal<7:0>** : dato de salida del registro MBR. Es el dato que se va a escribir en la memoria.
- **sal_ccr<4:0>** : esta señal muestra el valor del registro de estado. Se toma como salida para las pruebas de funcionamiento del núcleo que se comentarán en posteriores capítulos, pero no es necesaria.
- **sal_ri<7:0>** : esta señal muestra el valor del registro de instrucción. La razón de que sea una salida del núcleo es la misma que la de la señal anterior, pero tampoco es necesaria.
- **carga_mar** : señal de carga del registro mar. Esta señal se toma como salida por la misma razón que las dos anteriores.
- **clk_perifericos** : señal de reloj que va a cada uno de los periféricos del microcontrolador.
- **oe_memo** : *enable* de la memoria. Permite leer o escribir en ella.
- **read_write** : señal que indica si se desea leer o escribir de la memoria.

El código fuente de este utilizado para el diseño de este componente se adjunta en el Capítulo 9 Anexo 9.2 dentro del fichero **unicon.vhd**.

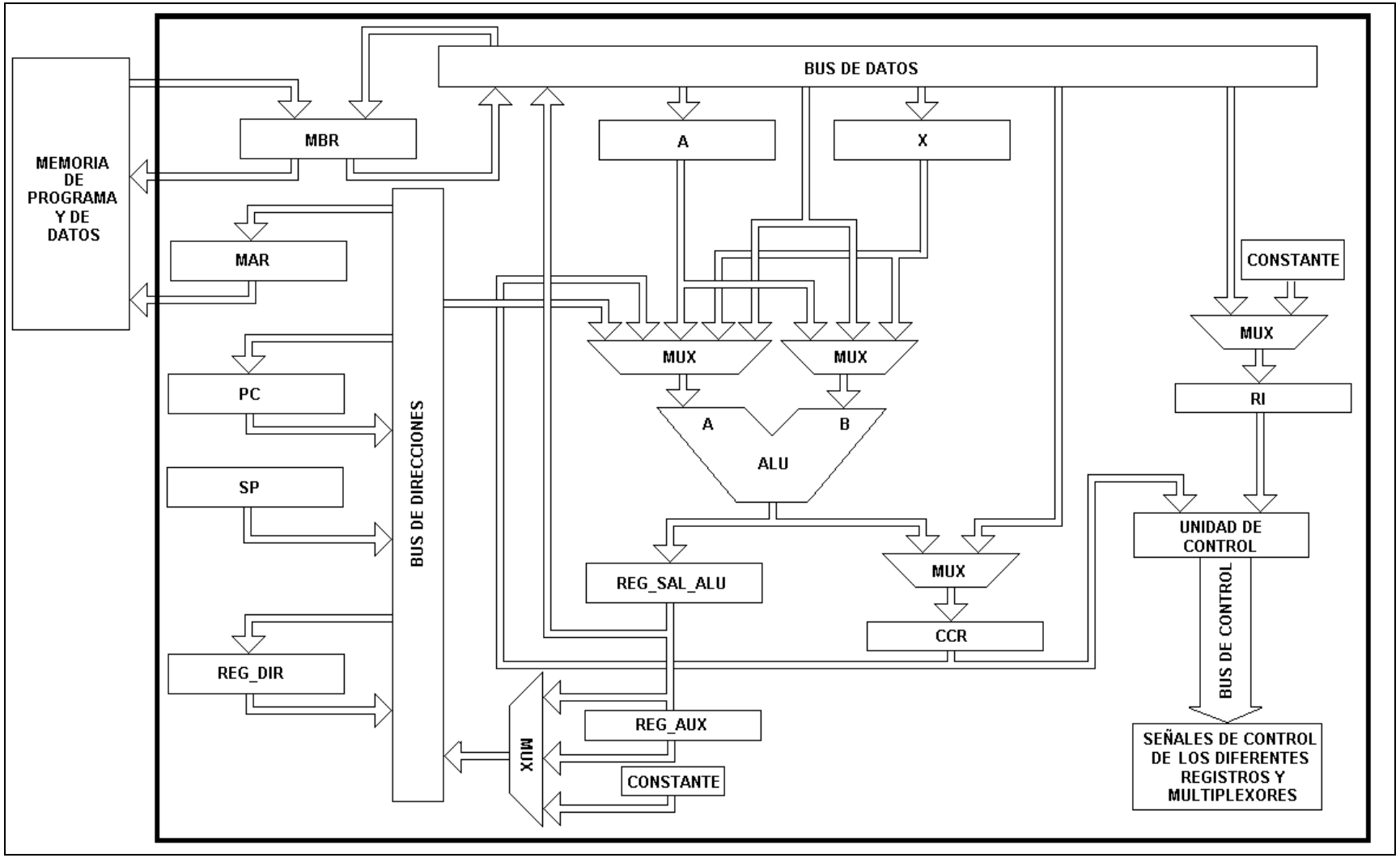


Figura 3.59 Diagrama de bloques del núcleo del 68HC05.

CAPÍTULO 4

DESCRIPCIÓN DEL JUEGO DE INSTRUCCIONES

4. DESCRIPCIÓN DEL JUEGO DE INSTRUCCIONES

En este capítulo se va a describir el juego completo de instrucciones que presenta este microcontrolador. Para ello se ha realizado una división en grupos de las mismas atendiendo a su función. Estos grupos son los siguientes:

- **Instrucciones de transferencia de datos**: el conjunto de estas instrucciones permite el movimiento de datos entre registros de la CPU y entre registros y memoria.
- **Instrucciones aritméticas**: este grupo se encarga de realizar las operaciones aritméticas básicas como la suma y la resta con y sin acarreo, la multiplicación sin signo, el incremento, el decremento, etc.
- **Instrucciones lógicas:** estas instrucciones efectúan como su propio nombre indica operaciones lógicas como el AND, OR, NOT, XOR, etc.
- **Instrucciones de comparación:** estas instrucciones comparan los datos que almacenan determinados registros con la memoria o entre sí, o bien comprueban su signo, o si el dato es nulo, etc.
- **Instrucciones de desplazamiento y rotación de bits:** estas instrucciones se encargan del desplazamiento y rotación de los datos contenidos tanto en registros de la CPU como de determinadas posiciones de memoria, bien hacia la izquierda o hacia la derecha. Las rotaciones de los mismos se realizan junto con el bit C del registro de estado, etc.
- **Instrucciones de manipulación de bits:** estas instrucciones cambian el valor de determinados bits. Puede ser o bien de un registro, o de una posición de memoria, o del registro de estado como el bit C (bit de acarreo) o bien el bit I (máscara de las interrupciones).
- **Instrucciones de salto y condicionales:** estas instrucciones producen un salto dentro de flujo normal de ejecución de un programa atendiendo en ciertos casos a que se cumplan determinadas condiciones.
- **Instrucciones de control:** estas instrucciones son las que se encargan del manejo del flujo de programa. Se trata de la interrupción *software*, de los retornos de subrutinas, y de las operaciones que hacen que el microcontrolador esté en bajo consumo como son el WAIT y el STOP.

Dentro de cada grupo se describe en profundidad en qué consiste cada una de las instrucciones pertenecientes al mismo. Una vez hecho esto, se proporciona una tabla donde se pueden observar los diferentes modos de direccionamiento que se pueden utilizar en esa instrucción junto con sus respectivos códigos mnemónicos, códigos de operación y los ciclos de reloj que ocupa la ejecución de cada uno de ellos tanto en el microcontrolador fabricado por Motorola como en el que en el presente proyecto se ha implementado. También se explicará cuáles son los bits del registro de estado que cada instrucción es capaz de modificar y de qué modo lo hace.

Por último, se detalla la secuencia de estados seguida en su ejecución. Como muchas de ellas coinciden en la secuencia de estados, se explicará en cada una de ellas cuáles son las variaciones principales que dan lugar a la ejecución de cada instrucción en concreto.

4.1. TRANSFERENCIA DE DATOS

4.1.1. LDA – Carga el acumulador desde la memoria

Esta instrucción consiste en que un dato que se encuentra guardado en una determinada posición de la memoria se carga en el acumulador.

ACCA ← (M)

Según el modo de direccionamiento se obtiene la posición o el dato directamente a guardar en el acumulador. En este caso se disponen de 6 modos de direccionamiento. Todos ellos, además de sus respectivos códigos mnemónicos, códigos de operación y ciclos de reloj que tarda cada uno en ejecutarse, se muestran en la Tabla 4.1.

La instrucción LDA sólo tiene la posibilidad de modificar dos bits del registro de estado, el bit N y el bit Z. El bit N tomará el valor ‘1’ o ’0’ dependiendo de si el dato que se carga en ese momento en el acumulador tiene su bit más significativo a ‘1’ o ‘0’. El bit Z se activará a ‘1’ sólo si el dato es cero, si no tendrá el valor ‘0’.

La secuencia de estados que se sigue para llevar a cabo la instrucción es la que se puede ver en la Figura 4.1. Esta secuencia también se ejecuta en otro tipo de operaciones que se comentarán cuando se describan cada una de ellas.

Tabla 4.1 Instrucción LDA.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
LDA (opr)	INM	A6	2	7
LDA (opr)	DIR	B6	3	11
LDA (opr)	EXT	C6	4	14
LDA,X	IX	F6	3	8
LDA (opr),X	IX8	E6	4	11
LDA (opr),X	IX16	D6	5	14

4.1.2. LDX – Carga el registro de indexado desde la memoria

Esta instrucción es prácticamente igual a la anterior. La única diferencia reside en que en vez de cargar el dato en el acumulador, se carga en el registro de indexado.

X ← (M)

La posición de memoria donde se encuentra el dato o el dato mismo, viene dada también según uno de los 6 modos de direccionamiento disponibles para esta instrucción y que, junto con más información sobre la instrucción, se pueden observar en la Tabla 4.2.

Por último al igual que la anterior, sólo tiene la posibilidad de modificar dos bits del registro de estado, el bit N y el bit Z y lo hará también de la misma forma que la instrucción LDA.

Tabla 4.2 Instrucción LDX.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
LDX (opr)	INM	AE	2	7
LDX (opr)	DIR	BE	3	11
LDX (opr)	EXT	CE	4	14
LDX,X	IX	FE	3	8
LDX (opr),X	IX8	EE	4	11
LDX (opr),X	IX16	DE	5	14

La secuencia de estados que se sigue para llevar a cabo la instrucción es la misma que la anterior y se muestra en la Figura 4.1. La única diferencia es que en el último estado "dato_a_bus", dependiendo de la instrucción de que se trate se activa, bien la señal de carga del registro acumulador, o bien la del registro de indexado.

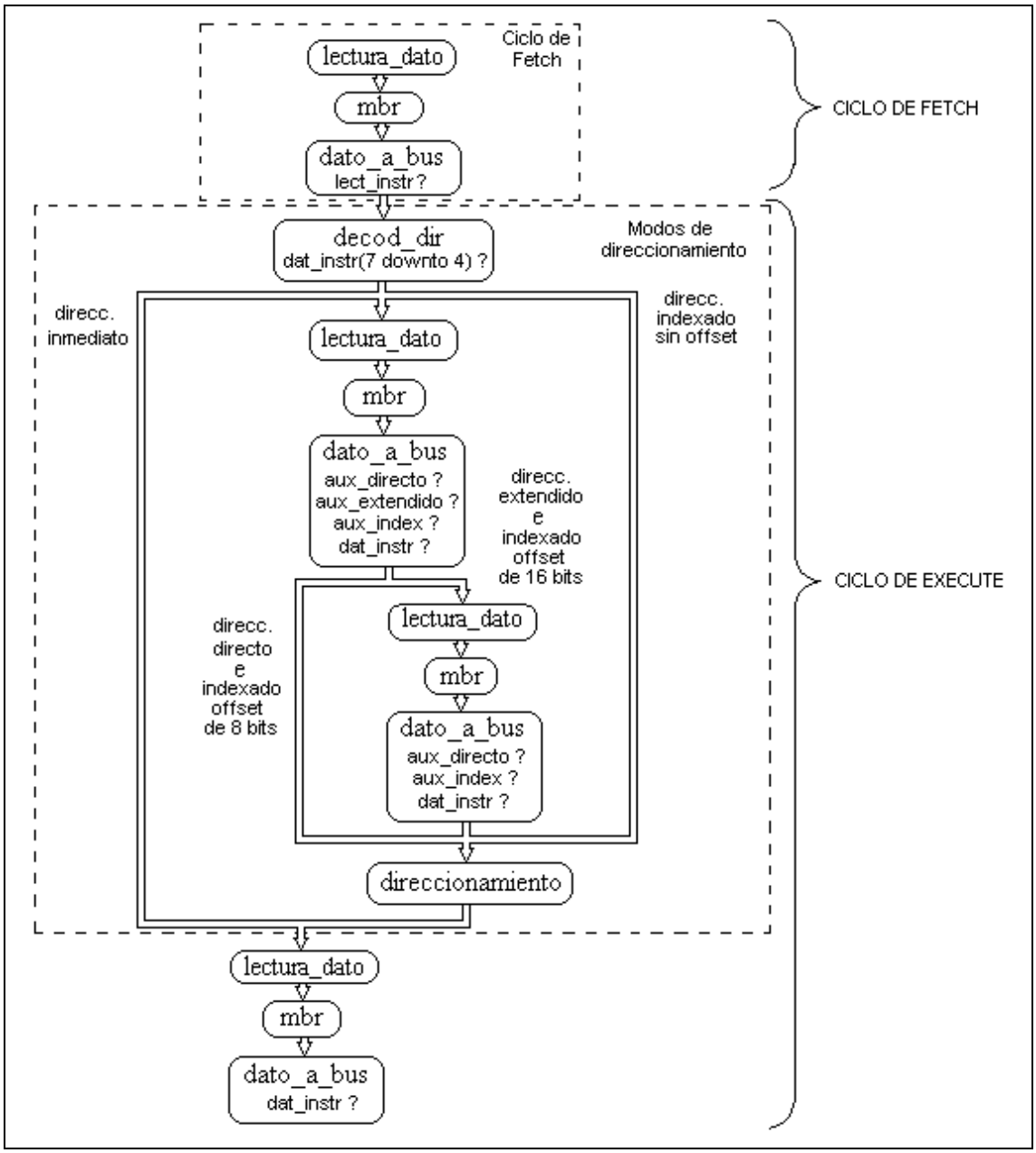


Figura 4.1 Secuencia de estados de las instrucciones LDA y LDX.

4.1.3. STA – Guarda el acumulador en la memoria

Esta instrucción se encarga de tomar el dato almacenado en el registro acumulador y guardarlo en una determinada posición de la memoria.

M ← (ACCA)

El dato del acumulador permanece inalterado. La posición de memoria viene indicada en los operandos de la instrucción conforme a uno de los 5 modos de direccionamiento que puede tener y que se detallan en la Tabla 4.3.

Tabla 4.3 Instrucción STA.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorota	Ciclos Implementación
STA (opr)	DIR	B7	4	10
STA (opr)	EXT	C7	5	13
STA,X	IX	F7	4	7
STA (opr),X	IX8	E7	5	10
STA (opr),X	IX16	D7	6	13

Esta instrucción puede modificar dos bits del registro de estado, el bit N y el bit Z, de la misma forma que las dos instrucciones anteriores.

La secuencia de estados que se sigue para llevar a cabo la instrucción es la que se muestra en la Figura 4.2.

4.1.4. STX – Guarda el registro de indexado en la memoria

Esta instrucción es muy similar a la anterior con la diferencia de que el dato que se guarda en memoria es el que contiene el registro de indexado.

M ← (X)

También posee 5 modos de direccionamiento y los bits del registro de estado que se pueden modificar son el bit N y el bit Z, de nuevo de la misma forma que las anteriores. Todos los modos de direccionamiento, sus códigos y los ciclos de ejecución de cada uno están contemplados en la Tabla 4.4.

Tabla 4.4 Instrucción STX.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorota	Ciclos Implementación
STX (opr)	DIR	BF	4	10
STX (opr)	EXT	CF	5	13
STX,X	IX	FF	4	7
STX (opr),X	IX8	EF	5	10
STX (opr),X	IX16	DF	6	13

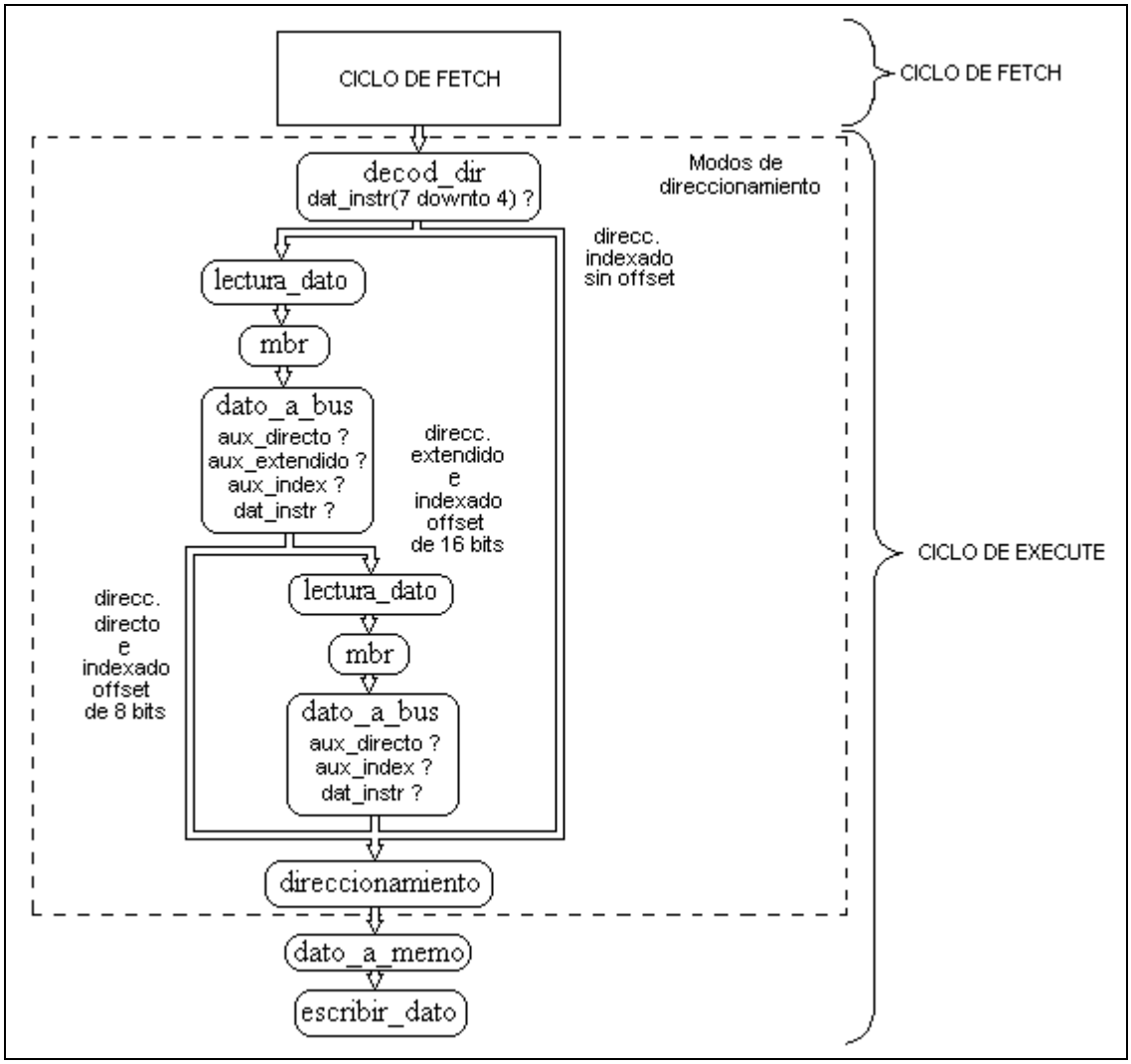


Figura 4.2 Secuencia de estados de las instrucciones STA y STX.

La secuencia de estados es la misma que la instrucción STA, y solamente difiere en las acciones que se llevan a cabo en el estado “direccionamiento”. En este estado se

carga en la ALU el dato que más tarde va a ser escrito en la memoria y es aquí cuando se decide si es el dato del acumulador para la instrucción STA o si es el del registro de indexado, en caso de que se esté ejecutando una instrucción STX. Esto se realiza por medio de la activación de la señal "carga_a_x". Esta secuencia se muestra en la Figura 4.2.

4.1.5. TAX – Transfiere el acumulador al registro de indexado

Esta instrucción permite que el contenido del registro acumulador se cargue en el registro de indexado. El dato almacenado en el acumulador no cambia.

X ← (ACCA)

Esta instrucción tiene un único modo de direccionamiento, el modo inherente, y ninguno de los bits del registro de estado se ve afectado por ella. La información relacionada con esta instrucción se puede observar en la Tabla 4.5.

Tabla 4.5 Instrucción TAX.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
TAX	INH	97	2	5

La secuencia de estados de la instrucción TAX se puede ver en la Figura 4.3. En este caso el ciclo de *fetch* no ha sido detallado al ser idéntico al de la Figura 4.1.

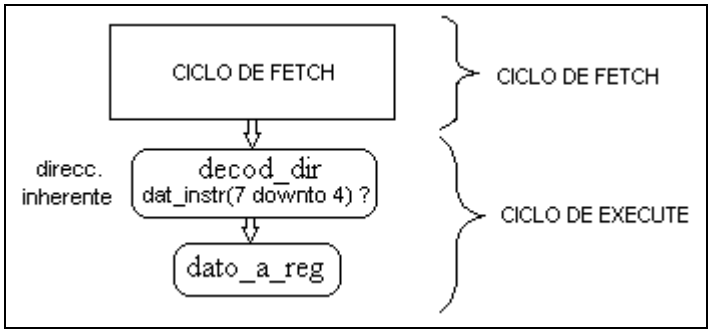


Figura 4.3 Secuencia de estados de las instrucciones TAX y TXA.

4.1.6. TXA – Transfiere el registro de indexado al acumulador

Esta instrucción es la inversa a la anterior. En este caso el contenido del registro de indexado se carga en el registro acumulador y como antes el dato almacenado en el registro de donde se obtiene no varía.

ACCA ← (X)

Esta operación tiene también un único modo de direccionamiento, el modo inherente, y tampoco se ven afectados los bits del registro de estado. La Tabla 4.6 muestra toda la información sobre esta instrucción.

Tabla 4.6 Instrucción TXA.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
TXA	INH	9F	2	5

La secuencia de estados es similar a la de la instrucción TAX y se muestra en la Figura 4.3. La única variación que distingue a una de la otra es que en el estado "decod_dir" se activa la señal "carga_a_x" que permite que la ALU tome el dato del registro adecuado y lo pase al bus de datos. En el siguiente estado se activa la señal que carga el dato en el registro correspondiente dependiendo de la instrucción que se está ejecutando en ese momento.

4.2. INSTRUCCIONES ARITMÉTICAS

4.2.1. ADC – Suma con acarreo

Esta instrucción consiste en sumar el valor del bit C del registro de estado a la suma de un dato que se encuentra guardado en una determinada posición de la memoria más el dato almacenado en ese instante en el acumulador. El resultado de la operación se lleva al acumulador.

ACCA ← (ACCA) + (M) + (C)

El sumando almacenado en memoria o su posición dentro de la misma vendrán dados según el modo de direccionamiento. En este caso se dispone de 6 modos de

direccionamiento. Todos ellos al igual que el código de operación de la instrucción, su mnemónico y los ciclos de reloj que tarda en su ejecución se pueden ver en la Tabla 4.7.

Tabla 4.7 Instrucción ADC.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
ADC (opr)	INM	A9	2	8
ADC (opr)	DIR	B9	3	12
ADC (opr)	EXT	C9	4	15
ADC,X	IX	F9	3	9
ADC (opr),X	IX8	E9	4	12
ADC (opr),X	IX16	D9	5	15

Esta instrucción puede modificar todos los bits del registro de estado exceptuando la máscara de interrupción (bit I) de la siguiente forma:

- El bit N tomará el valor del bit más significativo del resultado.
- El bit Z solamente se activará si el resultado es un valor negativo.
- El bit C toma el valor '1' solamente si existe acarreo en la suma.
- Y, por último, el bit H o acarreo intermedio, que sólo se activa si tiene lugar un acarreo entre los bits 3 y 4.

La secuencia de estados que sigue la instrucción ADC se muestra en la Figura 4.4. Esta secuencia se va a seguir en diversas instrucciones. La diferencia entre la ejecución de las diferentes instrucciones se da en el estado "dato_a_bus". En este estado se le ordena a la ALU por medio de la señal "sel_op" realizar una u otra operación dependiendo del valor de los últimos 4 bits de la instrucción que en ese instante se está ejecutando. Para este caso esta operación es la suma. Además de esto, para poder sumar con el bit C se debe tener activada en este estado la señal "c_y_n".

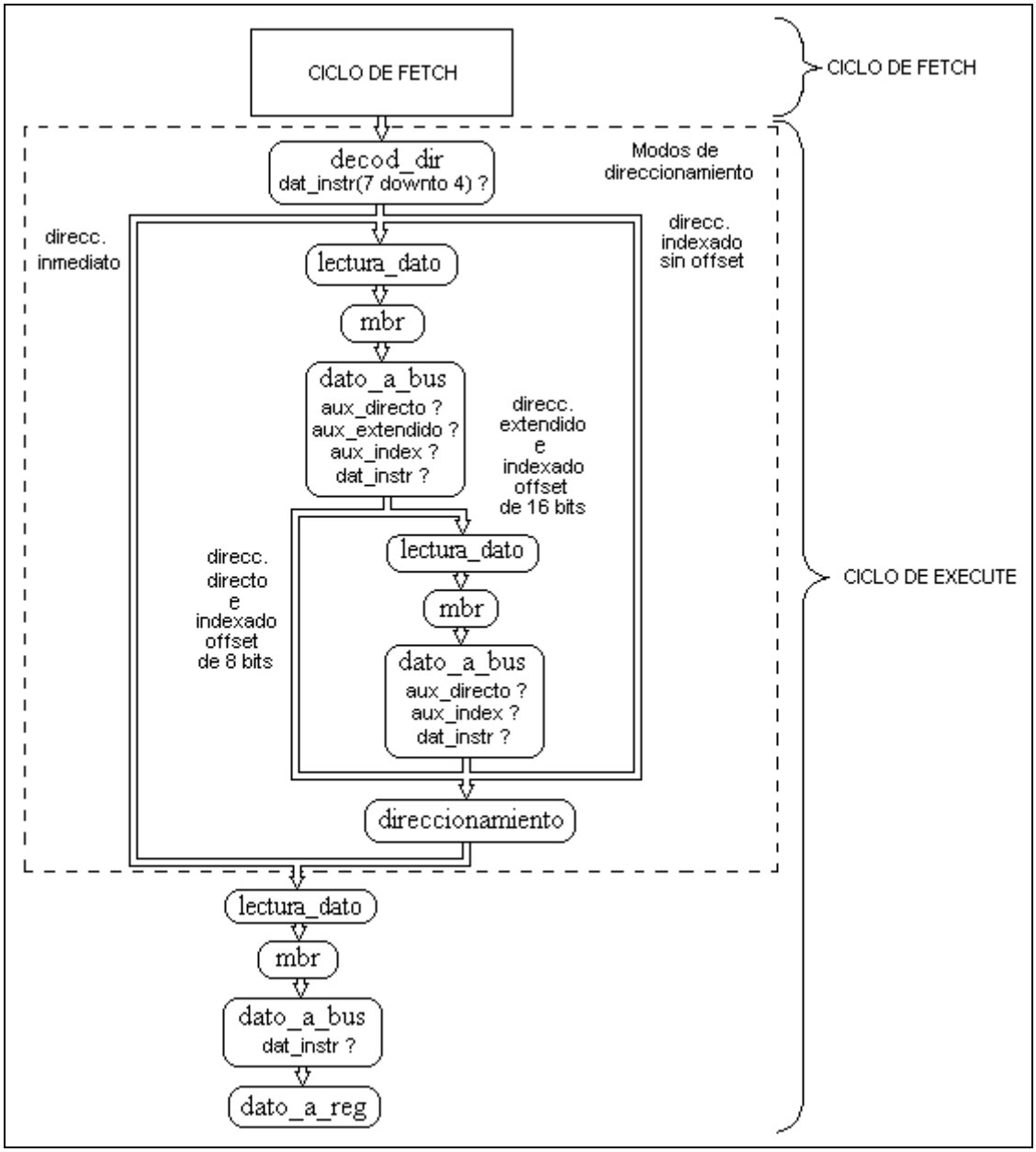


Figura 4.4 Secuencia de estados de la instrucción ADC.

4.2.2. ADD – suma sin acarreo

Esta instrucción es idéntica a la anterior pero sumando única y exclusivamente el acumulador más un dato almacenado en memoria sin tener en cuenta el valor del bit C. El resultado se guarda de nuevo en el registro acumulador.

ACCA ← (ACCA) + (M)

Los modos de direccionamiento, su código de operación y toda la información sobre la misma se puede observar en la Tabla 4.8. Esta instrucción puede cambiar el

valor de los mismos bits del registro de estado y de la misma forma que la instrucción ADC.

Tabla 4.8 Instrucción ADD.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
ADD (opr)	INM	AB	2	8
ADD (opr)	DIR	BB	3	12
ADD (opr)	EXT	CB	4	15
ADD,X	IX	FB	3	9
ADD (opr),X	IX8	EB	4	12
ADD (opr),X	IX16	DB	5	15

La secuencia de estados que sigue esta instrucción resulta ser la misma que la que se muestra en la Figura 4.4. La diferencia en este caso, además de que en el estado "dato_a_bus" se selecciona que la ALU realice la operación de sumar, es que en este caso no es necesaria la activación de la señal "c_y_n" ya que la suma es sin acarreo.

4.2.3. DEC – Decrementa

Esta instrucción decrementa en una unidad el contenido o bien del registro acumulador, o bien del registro de indexado, o bien de una determinada posición de memoria.

ACCA ← (ACCA) - $01

X ← (X) - $01

M← (M) - $01

Las diferentes posibilidades que tiene esta instrucción y sus modos de direccionamiento se pueden observar en la Tabla 4.9.

Esta instrucción puede modificar dos bits del registro de estado, el bit N y el bit Z y lo hace de la misma forma que ya se ha explicado para otras operaciones.

La secuencia de estados que sigue esta instrucción se muestra en la Figura 4.5. El ciclo de *fetch* no ha sido detallado por ser el mismo que se puede encontrar en la Figura 4.1.

Tabla 4.9 Instrucción DEC.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
DECA	INH	4A	3	5
DECX	INH	5A	3	5
DEC (opr)	DIR	3A	5	13
DEC,X	IX	7A	5	10
DEC (opr),X	IX8	6A	6	13

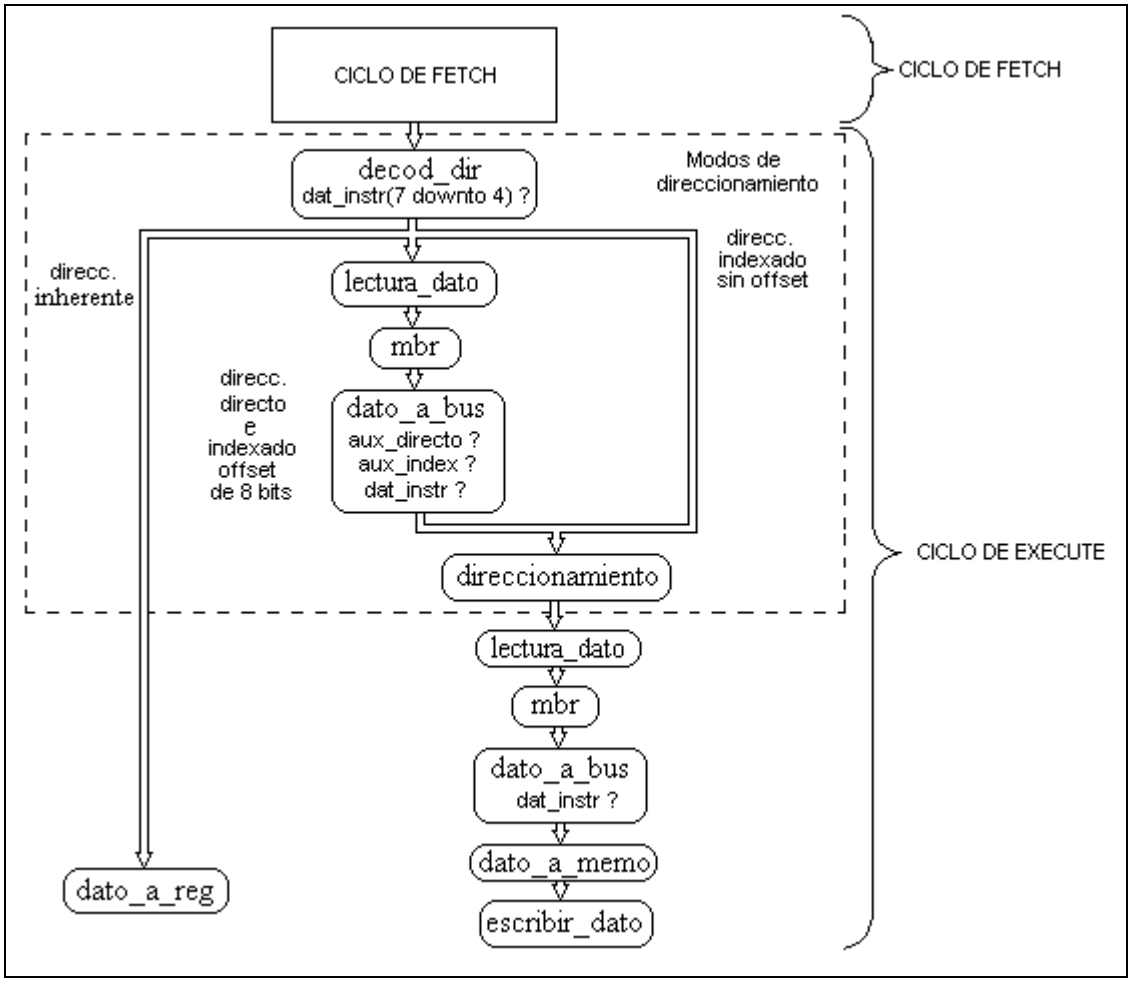


Figura 4.5 Secuencia de estados de las instrucciones DEC e INC.

4.2.4. INC – Incrementa

Esta instrucción incrementa en una unidad el contenido, bien del registro acumulador, o bien del registro de indexado, o bien de una determinada posición de memoria.

ACCA ← (ACCA) + $01

X ← (X) + $01

M← (M) + $01

Las diferentes posibilidades que tiene la instrucción INC y sus modos de direccionamiento se pueden observar en la Tabla 4.10. Esta puede modificar dos bits del registro de estado, el bit N y el bit Z, y lo hace de la misma forma que ya se ha explicado para otras operaciones.

Tabla 4.10 Instrucción INC.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
INCA	INH	4C	3	5
INCX	INH	5C	3	5
INC (opr)	DIR	3C	5	13
INC,X	IX	7C	5	10
INC (opr),X	IX8	6C	6	13

La secuencia de estados que sigue esta instrucción es la misma que la de la instrucción DEC y se muestra en la Figura 4.5. La única diferencia entre una operación y otra es que, o bien en el estado “decod_dir” si se trata de direccionamiento inherente, o bien en el estado “dato_a_bus” si es cualquiera de los demás direccionamientos, se le ordena a la ALU por medio de la señal “sel_op” que realice un tipo de operación u otra. En este caso la operación que se lleva a cabo es la de incrementar.

4.2.5. MUL – Multiplicación sin signo

Esta instrucción multiplica los 8 bits del dato almacenado en el registro de indexado por los 8 bits del dato almacenado en el acumulador. El resultado es guardado

en estos mismos registros, los 8 bits más significativos se guardan en el registro de indexado y los menos significativos en el acumulador.

X : A ← X * A

Esta instrucción solamente dispone del modo de direccionamiento inherente. Su código mnemónico y demás información se muestran en la tabla Tabla 4.11.

Tabla 4.11 Instrucción MUL.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
MUL	INH	42	11	6

Los bits del registro de estado que se ven afectados por esta instrucción son el bit H y el bit C, los dos acarreos. Esta instrucción hace que ambos se pongan a '0'.

La secuencia de estados que sigue la instrucción MUL se muestra a continuación en la Figura 4.6.

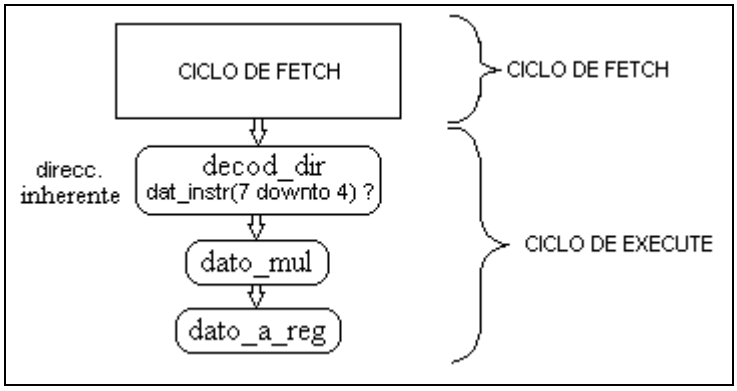


Figura 4.6 Secuencia de estados de la instrucción MUL.

4.2.6. SBC – Resta con acarreo

Esta instrucción realiza la resta o sustracción del dato que contenga el acumulador en ese instante, menos un dato almacenado en memoria, menos el valor actual del bit C del registro de estado. El resultado se guarda de nuevo en el registro acumulador.

ACCA ← (ACCA) – (M) – (C)

El dato de la memoria o su posición vienen dados por los diferentes modos de direccionamiento que se pueden observar en la Tabla 4.12.

Esta instrucción puede cambiar tres bits del registro de estado de la siguiente manera:

- El bit N toma el valor del bit más significativo del resultado.
- El bit Z, de la misma forma que siempre, se activa al valor lógico '1' si el resultado de la operación es nulo.
- El bit C se activa si el valor absoluto del dato de la memoria es mayor que el valor absoluto del que se encuentra almacenado en el registro acumulador.

Tabla 4.12 Instrucción SBC.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
SBC (opr)	INM	A2	2	8
SBC (opr)	DIR	B2	3	12
SBC (opr)	EXT	C2	4	15
SBC,X	IX	F2	3	9
SBC (opr),X	IX8	E2	4	12
SBC (opr),X	IX16	D2	5	15

La secuencia de estados que sigue esta instrucción también resulta la misma que la que se muestra en la Figura 4.4. La diferencia, en este caso, es que se le ordena a la ALU por medio de la señal "sel_op" que realice la operación de la sustracción en el estado "dato_a_bus" y que esta operación sea con acarreo. Esto último se realiza mediante la activación de la señal "c_y_n".

4.2.7. SUB – Sustracción

Esta instrucción es muy parecida a la anterior con la salvedad de que en este caso se realiza la resta del dato guardado en el acumulador menos un dato almacenado en memoria sin tener en cuenta el bit C. El resultado se guarda de nuevo en el registro acumulador.

ACCA ← (ACCA) – (M)

El dato de la memoria o su posición vienen dados por los diferentes modos de direccionamiento que se pueden observar en la Tabla 4.13. Esta instrucción puede cambiar los mismos bits del registro de estado que la instrucción SBC y de la misma forma.

Tabla 4.13 Instrucción SUB.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
SUB (opr)	INM	A0	2	8
SUB (opr)	DIR	B0	3	12
SUB (opr)	EXT	C0	4	15
SUB,X	IX	F0	3	9
SUB (opr),X	IX8	E0	4	12
SUB (opr),X	IX16	D0	5	15

La secuencia de estados que sigue la instrucción SUB también resulta la misma que la que se muestra en la Tabla 4.13. Ahora bien, para la ejecución en concreto de esta instrucción en el estado “dato_a_bus” se le ordena a la ALU por medio de la señal “sel_op” que realice la operación de la sustracción, mientras que la señal “c_y_n” debe estar desactivada al no tener que utilizar el bit C para la resta.

4.3. INSTRUCCIONES LÓGICAS

4.3.1. AND – AND Lógico

Esta instrucción realiza la operación lógica AND entre un dato que se encuentra guardado en una determinada posición de la memoria y el dato almacenado en ese instante en el acumulador. El resultado se guarda en el registro acumulador.

$$\mathbf{ACCA \leftarrow (ACCA) \cdot (M)}$$

El operando almacenado en memoria o su posición dentro de la misma vendrá dado según uno de los diferentes modos de direccionamiento que puede tener esta instrucción y que se muestran en la Tabla 4.14 junto con otra información sobre la instrucción.

Tabla 4.14 Instrucción AND.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
AND (opr)	INM	A4	2	8
AND (opr)	DIR	B4	3	12
AND (opr)	EXT	C4	4	15
AND,X	IX	F4	3	9
AND (opr),X	IX8	E4	4	12
AND (opr),X	IX16	D4	5	15

Esta operación modifica solamente los bits N y Z del registro de estado de la misma forma que ya se ha comentado en el resto de operaciones anteriores.

La secuencia de estados que sigue es también la misma que la que se muestra en la Figura 4.4. Para la ejecución de esta instrucción se le ordena a la ALU por medio de la señal “sel_op” que realice la operación AND en el estado “dato_a_bus”.

4.3.2. COM – Complemento

Esta instrucción se encarga de realizar el complemento a uno del dato guardado en el acumulador, en el registro de indexado o en una determinada posición de la memoria.

$$\mathbf{ACCA \leftarrow (\overline{ACCA}) = \$FF - (ACCA)}$$

$$\mathbf{X \leftarrow (\overline{X}) = \$FF - (X)}$$

$$\mathbf{M \leftarrow (\overline{M}) = \$FF - (M)}$$

Esta operación dispone de varios modos de direccionamiento que junto con sus códigos de operación, mnemónicos y ciclos de reloj se muestran en la Tabla 4.15.

Los bits del registro de estado que se ven afectados por esta instrucción son el bit Z y el bit N de la misma forma que en el resto de instrucciones anteriores. También se modifica el valor del bit C que siempre toma el valor ‘1’.

Tabla 4.15 Instrucción COM.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
COMA	INH	43	3	5
COMX	INH	53	3	5
COM (opr)	DIR	33	5	13
COM,X	IX	73	5	10
COM (opr),X	IX8	63	6	13

La secuencia de estados que sigue esta instrucción es la misma que las de las instrucciones DEC e INC y se muestra en la Figura 4.5. Para que se ejecuten las diferentes instrucciones, bien en el estado “decod_dir” si se trata de direccionamiento inherente, o bien en el estado “dato_a_bus” si es cualquiera de los demás direccionamientos, se le ordena a la ALU por medio de la señal “sel_op” que realice un tipo de operación u otra. Para la instrucción COM hay que realizar la operación lógica NOT.

4.3.3. EOR – OR Exclusiva

Esta instrucción realiza la operación lógica XOR entre un dato que se encuentra guardado en una determinada posición de la memoria y el dato almacenado en ese instante en el acumulador. El resultado se guarda en el registro acumulador.

$$\mathbf{ACCA \leftarrow (ACCA) \oplus (M)}$$

La información de la instrucción EOR se muestra en la Tabla 4.16. Esta solamente modifica los bits N y Z del registro de estado de la misma forma que ya se ha comentado en el resto de operaciones anteriores.

Tabla 4.16 Instrucción EOR.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
EOR (opr)	INM	A8	2	8
EOR (opr)	DIR	B8	3	12

EOR (opr)	EXT	C8	4	15
EOR,X	IX	F8	3	9
EOR (opr),X	IX8	E8	4	12
EOR (opr),X	IX16	D8	5	15

La secuencia de estados que sigue esta instrucción es también la misma que la que se muestra en la Figura 4.4. La diferencia para este caso, es que en el estado "dato_a_bus" la operación que realiza la ALU es la XOR.

4.3.4. NEG – Negación

La instrucción NEG realiza el complemento a dos del dato almacenado, bien en el acumulador, o bien en el registro de indexado, o en una posición de memoria dada por el modo de direccionamiento.

$$\mathbf{ACCA \leftarrow -(ACCA) = \$00 - (ACCA)}$$

$$\mathbf{X \leftarrow -(X) = \$00 - (X)}$$

$$\mathbf{M \leftarrow -(M) = \$00 - (M)}$$

Los diferentes modos de direccionamiento, sus códigos mnemónicos de operación y demás datos sobre esta instrucción se muestran en la Tabla 4.17.

Los bits del registro de estado que pueden ser alterados por esta instrucción son el bit Z, el bit N y el bit C. Los dos primeros cambian de la misma forma que ya se ha explicado anteriormente. El bit C se pone a '1' en todos los casos menos cuando el dato sobre el que se va a realizar el complemento a dos es $00, que se mantiene a '0'.

Tabla 4.17 Instrucción NEG.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
NEGA	INH	40	3	5
NEGX	INH	50	3	5
NEG (opr)	DIR	30	5	13
NEG,X	IX	70	5	10
NEG (opr),X	IX8	60	6	13

La secuencia de estados que sigue esta instrucción es la misma que las de las instrucciones DEC e INC y se muestra en la Figura 4.5. La diferencia reside en que, bien en el estado "decod_dir" si se trata de direccionamiento inherente, o bien en el estado "dato_a_bus" si es cualquiera de los demás direccionamientos, se le ordena a la ALU por medio de la señal "sel_op" que realice un tipo de operación u otra. En este caso para realizar el complemento a dos se selecciona la operación de restar. Se resta el dato sobre el que se opera al valor $00. Si el dato sobre el que se realiza es $80 este va a permanecer inalterado.

4.3.5. ORA – OR Inclusivo

Esta instrucción realiza la operación lógica OR entre un dato que se encuentra guardado en una determinada posición de la memoria y el dato almacenado en ese instante en el acumulador. El resultado se guarda en el registro acumulador.

ACCA ← (ACCA) + (M)

El operando almacenado en memoria o su posición dentro de la misma vendrá dado según uno de los diferentes modos de direccionamiento que puede tener esta instrucción y que se muestran en la Tabla 4.18 junto con más información sobre la instrucción.

Esta instrucción modifica solamente los bits N y Z del registro de estado de la misma forma que ya se ha comentado en el resto de operaciones anteriores.

Tabla 4.18 Instrucción ORA.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
ORA (opr)	INM	AA	2	8
ORA (opr)	DIR	BA	3	12
ORA (opr)	EXT	CA	4	15
ORA,X	IX	FA	3	9
ORA (opr),X	IX8	EA	4	12
ORA (opr),X	IX16	DA	5	15

La secuencia de estados que sigue esta instrucción es también la misma que la que se muestra en la Figura 4.4. La diferencia en la ejecución de una instrucción u otra se encuentra de nuevo en el estado “dato_a_bus”. Para esta instrucción la operación que realiza la ALU es la operación lógica OR.

4.4. INSTRUCCIONES DE COMPARACIÓN

4.4.1. BIT – Bit de prueba de la memoria con el acumulador

Esta instrucción realiza la comparación lógica AND entre el contenido del acumulador y de una determinada posición de la memoria. El dato almacenado en el acumulador y en la memoria van a permanecer inalterados, lo único que va a variar es el contenido del registro de estado.

(ACCA) · (M)

Esta operación dispone de 6 modos de direccionamiento que, junto con sus códigos de operación y al igual que en las instrucciones anteriores, se pueden ver en la Tabla 4.19.

Esta instrucción modifica solamente los bits N y Z del registro de estado de la misma forma que ya se ha comentado en el resto de operaciones anteriores.

Tabla 4.19 Instrucción BIT.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
BIT (opr)	INM	A5	2	7
BIT (opr)	DIR	B5	3	11
BIT (opr)	EXT	C5	4	14
BIT,X	IX	F5	3	8
BIT (opr),X	IX8	E5	4	11
BIT (opr),X	IX16	D5	5	14

La secuencia de estados que sigue esta instrucción es también la misma que la que se muestra en la Figura 4.1. La diferencia se encuentra de nuevo en la operación que

realiza la ALU en el estado “dato_a_bus”. Para la instrucción BIT esta operación es la operación lógica AND.

4.4.2. CMP – Compara el acumulador con memoria

Esta instrucción compara el contenido del dato que se encuentra en el acumulador con el de una determinada dirección de memoria. Para realizar la comparación ejecuta una resta entre los dos, de manera que no varíe el contenido de ninguno de los registros sino que solamente se ven afectados algunos de los bits del registro de estado.

(ACCA) – (M)

Esta operación tiene 6 modos de direccionamiento los cuales se pueden ver en la Tabla 4.20 además de sus códigos mnemónicos, de operación y ciclos de reloj que tarda en ejecutarse cada uno de ellos.

Tabla 4.20 Instrucción CMP.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
CMP (opr)	INM	A1	2	7
CMP(opr)	DIR	B1	3	11
CMP (opr)	EXT	C1	4	14
CMP,X	IX	F1	3	8
CMP (opr),X	IX8	E1	4	11
CMP (opr),X	IX16	D1	5	14

Los bits del registro de estado que modifica esta instrucción son: el bit Z y el bit N de la misma forma que en el resto de operaciones, y el bit C que toma el valor ‘1’ en el caso de que el valor absoluto del dato almacenado en la memoria sea mayor que el del acumulador. En caso contrario toma el valor ’0’.

La secuencia de estados que sigue esta instrucción es también la misma que la que se muestra en la Figura 4.1. En este caso la diferencia es que en el estado “dato_a_bus” la operación a realizar por la ALU es la operación de la resta además de

que se necesita que uno de los sumandos sea el acumulador, para lo cual se activarán las señales que sean necesarias.

4.4.3. CPX – Compara el registro de indexado con la memoria

Esta instrucción es muy similar a la anterior con la diferencia que ahora se compara el contenido del dato almacenado en el registro de indexado con el de una determinada dirección de memoria. No varía el contenido de ninguno de los registros y solamente se ven afectados algunos de los bits del registro de estado.

(X) – (M)

Esta operación al igual que la anterior tiene también 6 modos de direccionamiento los cuales se pueden ver en la Tabla 4.21.

Tabla 4.21 Instrucción CPX.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
CPX (opr)	INM	A3	2	7
CPX(opr)	DIR	B3	3	11
CPX (opr)	EXT	C3	4	14
CPX,X	IX	F3	3	8
CPX (opr),X	IX8	E3	4	11
CPX (opr),X	IX16	D3	5	14

Los bits del registro de estado que modifica esta instrucción son los mismos y de la misma forma que la instrucción CMP con la particularidad de que, en este caso, el bit C se pone a ‘1’ si el valor absoluto del dato almacenado en la memoria es mayor que el del registro de indexado.

La secuencia de estados que sigue esta instrucción es la misma que la de la instrucción anterior y se muestra en la Figura 4.1. En el estado “dato_a_bus” la operación a realizar por la ALU es la resta, además de que hay que activar las señales necesarias en anteriores estados para que en este caso uno de los sumandos sea el registro de indexado.

4.4.4. TST – Prueba de cero o negativo

Esta instrucción se encarga de activar los correspondientes bits del registro de estado dependiendo si el dato almacenado en el acumulador, o en el registro de indexado, o en una determinada posición de memoria es cero o negativo.

(ACCA) – $00

(X) – $00

(M) – $00

Los modos de direccionamiento de esta instrucción y demás información se pueden encontrar en la Tabla 4.22 que se muestra a continuación.

Tabla 4.22 Instrucción TST.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
TSTA	INH	4D	3	4
TSTX	INH	5D	3	4
TST (opr)	DIR	3D	5	11
TST,X	IX	7D	5	8
TST (opr),X	IX8	6D	6	11

Los bits que modifica esta instrucción del registro de estado son el bit Z y el bit N de la misma forma que en el resto de las instrucciones anteriores.

La secuencia de estados que sigue esta instrucción se muestra en la Figura 4.7. En esta instrucción como el dato que maneja no cambia no hace falta volver a almacenarlo en el registro o la dirección de la memoria de la que se obtiene.

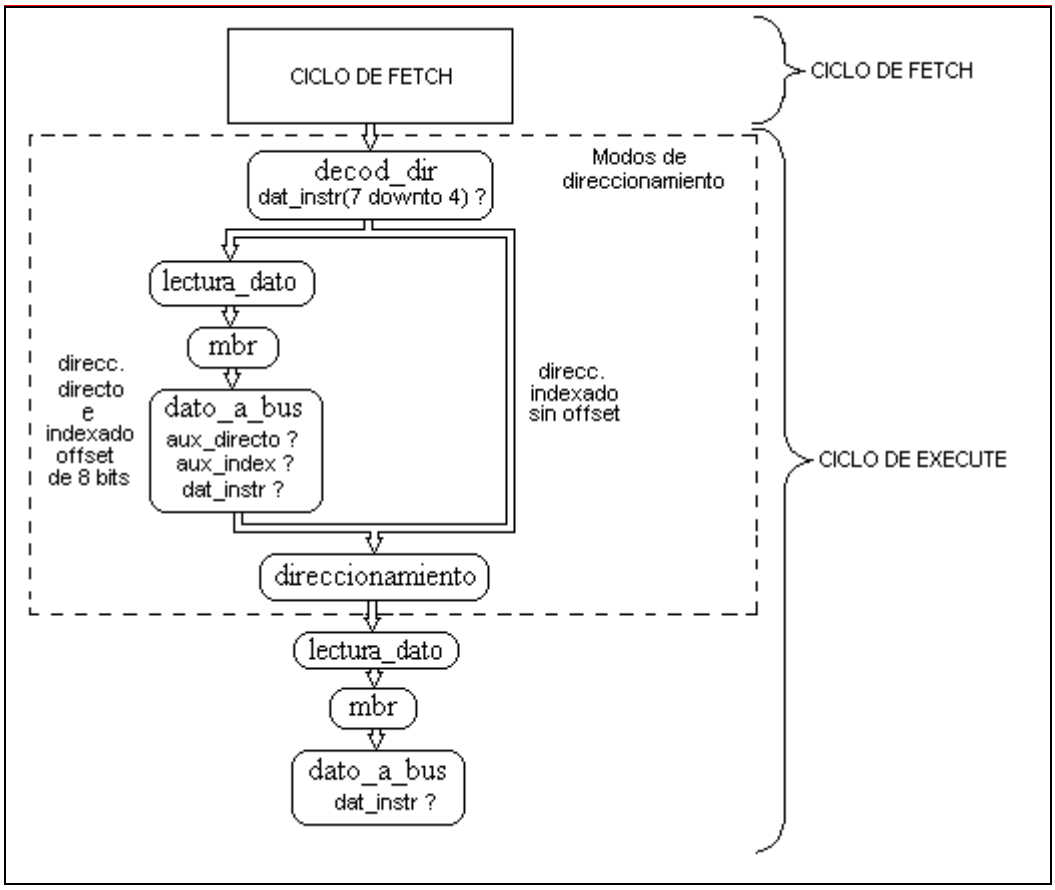
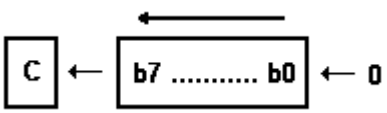


Figura 4.7 Secuencia de estados de la instrucción TST.

4.5. INSTRUCCIONES DE DESPLAZAMIENTO Y ROTACIÓN DE BITS

4.5.1. ASL – Desplazamiento aritmético a la izquierda

Esta instrucción desplaza todos los bits o del acumulador, o del registro de indexado, o de una determinada posición de la memoria hacia la izquierda. En el bit menos significativo se escribe un '0' y el bit más significativo se pasa al bit C del registro de estado.



En la Tabla 4.23 se pueden observar todos los modos de direccionamiento de esta instrucción además de sus respectivos códigos mnemónicos, de operación y los ciclos de reloj que tarda en ejecutarse.

Tabla 4.23 Instrucción ASL.

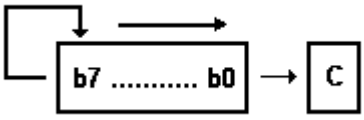
Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
ASLA	INH	48	3	5
ASLX	INH	58	3	5
ASL (opr)	DIR	38	5	13
ASL,X	IX	78	5	10
ASL (opr),X	IX8	68	6	13

Los bits del registro de estado que puede modificar esta operación son el bit Z, el bit N, y el bit C. El bit Z y el bit N varían de la misma forma que en anteriores instrucciones mientras que el bit C se pone a ‘1’ si el bit más significativo del contenido del registro que se va a desplazar antes de realizar la operación está a ‘1’, y a ‘0’ si no lo está.

La secuencia de estados que sigue la instrucción ASL es la misma que las de las instrucciones DEC e INC y se muestra en la Figura 4.5. En este caso en los estados “decod_dir” y “dato_a_bus” se le ordena a la ALU que realice la operación de rotar hacia la izquierda, y mediante la señal “bit_rot” se elige el valor que debe tomar el bit menos significativo. Para el caso de esta instrucción el valor es ‘0’.

4.5.2. ASR – Desplazamiento aritmético a la derecha

Esta instrucción consiste en desplazar los bits o del acumulador, o del registro de indexado, o de una determinada posición de memoria un lugar hacia la derecha. El bit más significativo del dato a rotar se mantiene constante mientras que el bit menos significativo se carga en el bit C del registro de estado.



Toda la información acerca de los modos de direccionamiento, sus códigos de operación y mnemónicos y los ciclos de reloj que dura esta instrucción se detallan en la Tabla 4.24.

Tabla 4.24 Instrucción ASR.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
ASRA	INH	47	3	5
ASRX	INH	57	3	5
ASR (opr)	DIR	37	5	13
ASR,X	IX	77	5	10
ASR (opr),X	IX8	67	6	13

Los bits del registro de estado que modifica esta instrucción son los mismos que los de la instrucción anterior. El bit N y el bit Z de la misma forma que siempre mientras que el bit C toma el valor del bit menos significativo del dato que se está rotando.

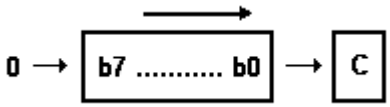
La secuencia de estados es también la misma que se sigue en las instrucciones DEC e INC y que se puede ver en la Figura 4.5. En este caso la diferencia se encuentra en que en el estado “decod_dir” y en “dato_a_bus” se activa la operación de la ALU de rotar hacia la derecha y la señal “bit_rot”, que elige el valor que debe tomar el bit más significativo del resultado de la rotación. Para esta instrucción este valor es el del bit más significativo del dato a rotar.

4.5.3. LSL – Desplazamiento lógico a la izquierda

Esta instrucción es idéntica a la instrucción ASL comentada al principio de este grupo de instrucciones. Por esta razón no se explicará de nuevo y toda la información relacionada con ella se puede ver en la explicación de la instrucción ASL.

4.5.4. LSR – Desplazamiento lógico a la derecha

Esta instrucción consiste en desplazar los bits o del acumulador, o del registro de indexado, o de una determinada posición de memoria un lugar hacia la derecha. El bit más significativo del dato a rotar toma el valor '0' mientras que el bit menos significativo se carga en el bit C del registro de estado.



La información de esta instrucción se muestra en la Tabla 4.25. Los bit del registro de estado que modifica son los bits N, Z y C. Los dos primero de la misma forma que siempre mientras que el bit C toma el valor del bit menos significativo del dato a rotar.

Tabla 4.25 Instrucción LSR.

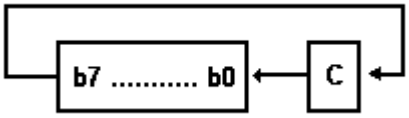
Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
LSRA	INH	44	3	5
LSRX	INH	54	3	5
LSR (opr)	DIR	34	5	13
LSR,X	IX	74	5	10
LSR (opr),X	IX8	64	6	13

La secuencia de estados es también la misma que se sigue en las instrucciones DEC e INC y que se puede ver en la Figura 4.5. En este caso la diferencia se encuentra en que en el estado "decod_dir" y en "dato_a_bus" se activa la operación de la ALU de rotar hacia la derecha. Por medio de la señal "bit_rot" se elige el valor que debe tomar el bit más significativo del resultado de la rotación. En este caso este valor será '0'.

4.5.5. ROL – Rotación a la izquierda a través del bit C

Esta instrucción consiste en rotar los bits o del acumulador, o del registro de indexado, o de una determinada posición de memoria un lugar hacia la derecha a través del bit C del registro de estado. El valor menos significativo del resultado de la rotación

toma el valor del bit C, mientras que este bit toma el valor más significativo del dato a rotar.



Toda la información acerca de la instrucción ROL se en detalla en la Tabla 4.26. Los bits del registro de estado que modifica esta instrucción son los mismos que los de la instrucción anterior. El bit N y el bit Z de la misma forma que siempre y el bit C como ya se ha explicado.

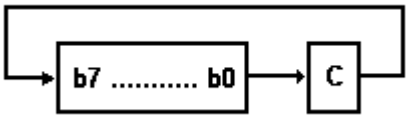
Tabla 4.26 Instrucción ROL.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
ROLA	INH	49	3	5
ROLX	INH	59	3	5
ROL (opr)	DIR	39	5	13
ROL,X	IX	79	5	10
ROL (opr),X	IX8	69	6	13

La secuencia de estados es también la que se puede ver en la Figura 4.5. En este caso la diferencia se encuentra en que en el estado “decod_dir” y en “dato_a_bus” se activa la operación de la ALU de rotar hacia la izquierda y la señal de “bit_rot” elige el valor que debe tomar el bit menos significativo del resultado de la rotación que, en este caso, es el bit C.

4.5.6. ROR – Rotación a la derecha a través del bit C

Esta instrucción consiste, al contrario que la anterior, en rotar los bits o del acumulador, o del registro de indexado, o de una determinada posición de memoria un lugar hacia la izquierda a través del bit C del registro de estado. El bit C toma en este caso el valor del bit menos significativo del dato a rotar mientras que el bit más significativo del resultado será el valor del bit C.



La información de esta instrucción se muestra en la Tabla 4.27. Los bit del registro de estado que modifica son los bits N, Z y C. Los dos primeros igual que en instrucciones anteriores, mientras que el bit C toma el valor del bit menos significativo del dato a rotar.

Tabla 4.27 Instrucción ROR.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
RORA	INH	46	3	5
RORX	INH	56	3	5
ROR (opr)	DIR	36	5	13
ROR,X	IX	76	5	10
ROR (opr),X	IX8	66	6	13

La secuencia de estados es también la misma que se sigue en las instrucciones DEC e INC y que se puede ver en la Figura 4.5. En este caso la diferencia se encuentra en que en el estado “decod_dir” y en “dato_a_bus” se activa la operación de la ALU de rotar hacia la derecha y se elige el valor de la señal de “bit_rot” para que el bit más significativo del resultado de la rotación tome el valor del bit C.

4.6. INSTRUCCIONES DE MANIPULACION DE BITS

4.6.1. BCRL n – Pone a cero un bit de la memoria

Esta instrucción pone a cero un bit de la memoria dejando inalterados el resto. El bit sobre el que opera le viene indicado por el código de operación.

Mn ← 0

Las posiciones de la memoria sobre las que puede actuar son sólo de la $0000 a la $00FF al tener esta instrucción como único modo de direccionamiento el directo.

Toda la información referente a esta instrucción se encuentra contemplada en la Tabla 4.28.

Tabla 4.28 Instrucción BCLR.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
BCLR 0,(opr)	DIR	11	5	13
BCLR 1,(opr)	DIR	13	5	13
BCLR 2,(opr)	DIR	15	5	13
BCLR 3,(opr)	DIR	17	5	13
BCLR 4,(opr)	DIR	19	5	13
BCLR 5,(opr)	DIR	1B	5	13
BCLR 6,(opr)	DIR	1D	5	13
BCLR 7,(opr)	DIR	1F	5	13

Por otro lado esta instrucción no afecta a ninguno de los bits del registro de estado, permaneciendo estos inalterados durante su ejecución.

La secuencia de estados que se desarrolla para la ejecución la instrucción BCLR se muestra en la Figura 4.8 y va a ser la misma que la de la instrucción contraria a esta, la BSET. La diferencia entre una y otra se encuentra en el estado “dato_a_bus”. En este estado se elige mediante la señal “sel_bit” sobre que bit se opera, y mediante la señal “sel_op” la operación que se va a realizar. Para esta instrucción se trata de la operación “bit_reset” que pone a cero el correspondiente bit.

4.6.2. BSET n – Pone a uno un bit de la memoria

Esta instrucción es la contraria de la anterior y consiste en poner a uno un determinado bit de la memoria indicado mediante el código de operación. Las posiciones de memoria a las que tiene acceso son las mismas que la instrucción anterior.

Mn ← 1

Esta instrucción tampoco modifica ningún bit del registro de estado y todo lo referente a ella se muestra en la Tabla 4.29.

Tabla 4.29 Instrucción BSET.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
BSET 0,(opr)	DIR	10	5	13
BSET 1,(opr)	DIR	12	5	13
BSET 2,(opr)	DIR	14	5	13
BSET 3,(opr)	DIR	16	5	13
BSET 4,(opr)	DIR	18	5	13
BSET 5,(opr)	DIR	1A	5	13
BSET 6,(opr)	DIR	1C	5	13
BSET 7,(opr)	DIR	1E	5	13

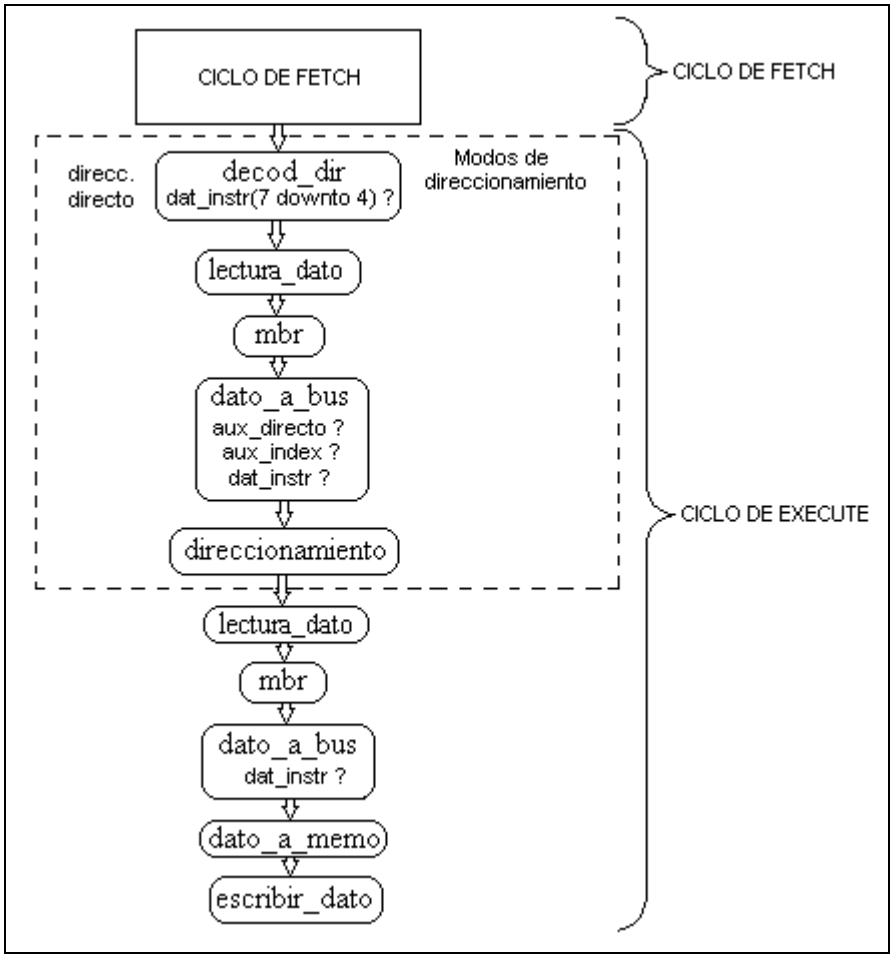


Figura 4.8 Secuencia de estados de la instrucción BCLR y BSET.

La secuencia de estados es la misma que la de la instrucción anterior y se puede ver en la Figura 4.8. La diferencia se encuentra en que en esta instrucción la operación que se le pide a la ALU que realice en el estado "dato_a_bus" es la "bit_set", de manera que el bit seleccionado se ponga a uno.

4.6.3. CLC – Pone a cero el bit C

Esta instrucción se encarga de poner a cero el bit C o bit de acarreo perteneciente al registro de estado.

$$\mathbf{C \leftarrow 0}$$

Solamente posee un modo de direccionamiento, el inherente y el único bit del registro de estado que se ve afectado por ella es, claro está, el bit C. La información de esta instrucción se muestra en la Tabla 4.30.

La secuencia de estados para esta instrucción es muy sencilla y se puede contemplar en la Figura 4.9. La operación principal se realiza en el estado "decod_dir" donde se activa la señal que carga el nuevo valor en el bit C del registro de estado y donde se le ordena a la ALU realizar un *reset* de este bit.

Tabla 4.30 Instrucción CLC.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
CLC	INH	98	2	4

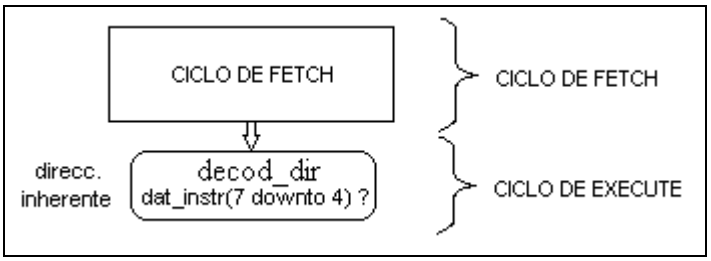


Figura 4.9 Secuencia de estados de las instrucciones CLC y CLI.

4.6.4. CLI – Pone a cero el bit I

Esta instrucción consiste es poner a uno la máscara de interrupción o bit I del registro de estado.

I ← 0

En la Tabla 4.31 se encuentran los detalles sobre la instrucción y se puede ver como esta es muy parecida a la instrucción anterior CLC.

Tabla 4.31 Instrucción CLI.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
CLI	INH	9A	2	4

El bit que se ve afectado en el registro de estado es únicamente el bit I y la secuencia de estados que sigue es la misma que la operación anterior activando en el estado “decod_dir” la señal que ordena la carga del bit I. Esta secuencia se muestra en la Figura 4.9.

4.6.5. CLR – Pone a cero un registro

Esta instrucción pone a cero todos los bits correspondientes bien al acumulador, o bien al registro de indexado, o bien a una determinada posición de la memoria.

ACCA ← $00

X ← $00

M ← $00

Esta instrucción sólo afecta a los bits N y Z del registro de estado. Estos bits se ven modificados de la misma forma que para el resto de las instrucciones que los alteran, pero en este caso toman un valor fijo ya que el resultado de la operación siempre es el mismo $00. Es por esto que al ejecutar esta instrucción el bit N toma siempre el valor ‘0’ mientras que el bit Z el valor ‘1’.

En la Tabla 4.32 se contemplan los modos de direccionamiento que posee esta instrucción junto con otros datos sobre la misma.

Tabla 4.32 Instrucción CLR.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
CLRA	INH	4F	3	5
CLRX	INH	5F	3	5
CLR (opr)	DIR	3F	5	13
CLR,X	IX	7F	5	10
CLR (opr),X	IX8	6F	6	13

La secuencia de estados que tiene lugar en la ejecución de esta instrucción es la que se muestra en la Figura 4.5. En este caso tanto en el estado "decod_dir" como en el estado "dato_a_bus", dependiendo de si se tiene un direccionamiento inherente o no, se activa la señal de la ALU "sel_op" de manera que se realice la operación de *reset* para borrar todos los bits del dato que se está manipulando.

4.6.6. SEC – Pone a uno el bit C

Esta instrucción es la contraria a la instrucción CLC y consiste en poner el bit C o bit de acarreo del registro de estado a '1'.

$$\mathbf{C \leftarrow 1}$$

La información sobre ella se puede ver en la Tabla 4.33. El único bit que modifica es el bit C del registro de estado.

Tabla 4.33 Instrucción SEC.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
SEC	INH	99	2	4

La secuencia de estados es la misma que la de la instrucción CLC y se puede observar en la Figura 4.9. Para que se ejecute esta instrucción en concreto en el estado "dato_a_bus" se activa la carga del bit C, mientras que la operación que va a realizar la ALU es la operación *set* para que este bit tome el valor uno.

4.6.7. SEI – Pone a uno el bit I

Esta instrucción consiste es poner a uno la máscara de interrupción o bit I del registro de estado. Es la instrucción contraria a la instrucción CLI.

I ← 1

Esta instrucción sólo modifica el bit I del registro de estado y todos los datos sobre ella se encuentran en la Tabla 4.34.

Tabla 4.34 Instrucción SEI.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
SEI	INH	9B	2	4

La secuencia de estados que sigue es de nuevo la que se muestra en la Figura 4.9. En este caso en el estado “decod_dir” se activa la carga del bit I y se selecciona la operación *set* de la ALU para que ponga el bit a ‘1’.

4.7. INSTRUCCIONES DE SALTO Y CONDICIONALES

4.7.1. BCC – Salto condicional si el bit C está a 0

Esta instrucción provoca un salto siguiendo un direccionamiento relativo si el bit C del registro de estado se encuentra a ‘0’.

PC ← (PC) + $0002 + Rel **Si (C) = 0**

Para realizar el salto el contador de programa se carga con el valor de la dirección de la siguiente instrucción de programa, una vez que ha leído los operandos de la presente instrucción, más el valor del *offset* relativo.

Toda la información acerca de esta instrucción viene contemplada en la Tabla 4.35.

Tabla 4.35 Instrucción BCC.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
BCC	REL	24	3	9

La instrucción BCC no modifica ninguno del los bit del registro de estados. Su secuencia de estados se puede observar en la Figura 4.10 y va a ser la misma que para la mayoría de las instrucciones de salto condicional. La diferencia entre una y otra se encuentra en el estado "dato_a_bus" donde dependiendo del código de operación se consulta una condición u otra. Tratándose de esta instrucción la condición que se tiene en cuenta es el valor del bit C.

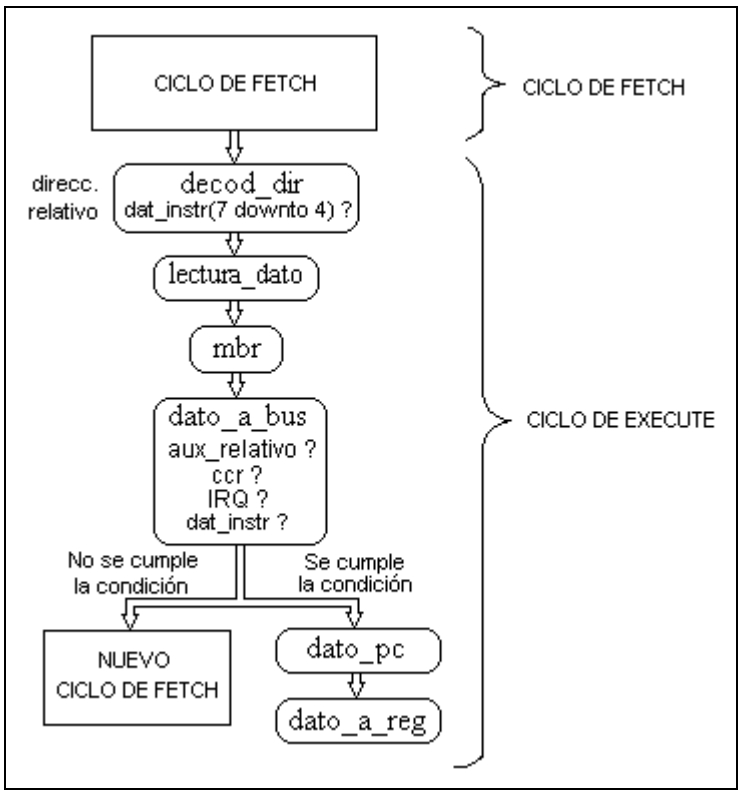


Figura 4.10 Secuencia de estados de las instrucciones condicionales.

4.7.2. BCS – Salto condicional si el bit C está a 1

Esta instrucción salta si el valor del bit C está a '1'. La condición es justo la contraria a la instrucción BCC.

PC ← (PC) + $0002 + Rel Si (C) = 1

Esta instrucción tampoco modifica ningún bit del registro de estado y la información relacionada con ella se puede observar en la Tabla 4.36.

Tabla 4.36 Instrucción BCS.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
BCS	REL	25	3	9

La secuencia de estados es también la que se ve en la Figura 4.10. En este caso la condición que se consulta en el estado "dato_a_bus" tiene que ver con el valor del bit C del registro de estado.

4.7.3. BEQ – Salto condicional si es igual

Esta instrucción salta si el valor del bit Z del registro de estado es '1'. Por lo general suele venir después de comparar dos datos de manera que, si son iguales, el bit Z estará activado y se podrá realizar el salto, mientras que si no lo son se continúa con la siguiente instrucción de programa.

PC ← (PC) + $0002 + Rel **Si (Z) = 1**

Esta instrucción tampoco modifica los valores del registro de estado y todo lo referente a ella se encuentra en la Tabla 4.37.

Tabla 4.37 Instrucción BEQ.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
BEQ	REL	27	3	9

La secuencia de estados que sigue es también la que se muestra en la Figura 4.10. En este caso en la condición de salto impuesta en el estado "dato_a_bus" entra en juego el valor del bit Z. Por lo demás el resto de la ejecución se desarrolla de la misma forma que las anteriores.

4.7.4. BHCC – Salto condicional si el bit H está a 0

Esta instrucción salta si el valor del bit H o acarreo intermedio es '0'. Este se activa si se produce un acarreo entre los bits 3 y 4 como ya se ha comentado anteriormente.

PC ← (PC) + $0002 + Rel Si (H) = 0

Esta no cambia ninguno de los bits del registro de estado. Todos los datos sobre la misma se detallan en la Tabla 4.38.

Tabla 4.38 Instrucción BHCC.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
BHCC	REL	28	3	9

La secuencia de estados es de nuevo la que se puede ver en la Figura 4.10. En este caso la condición del estado "dato_a_bus" según la cual se salta o no, tiene que ver con el valor del bit H en ese momento.

4.7.5. BHCS – Salto condicional si el bit H está a 1

Esta instrucción es la contraria a la anterior, es decir, ahora el salto se produce si el valor del bit H del registro de estado es uno.

PC ← (PC) + $0002 + Rel Si (H) = 1

Los datos sobre su código de operación, mnemónico y ciclos que tarda en ejecutarse se muestran en la Tabla 4.39.

Tabla 4.39 Instrucción BHCS.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
BHCS	REL	29	3	9

Esta instrucción tampoco altera el valor del registro de estado, consultando el valor del bit H en el estado "dato_a_bus" dentro de la secuencia de estados de ejecución de la misma. Esta secuencia es idéntica a las de las anteriores instrucciones de salto condicional y se muestra en la Figura 4.10.

4.7.6. BHI – Salto condicional si es mayor

Esta instrucción salta si el resultado de la operación lógica OR de los bit C y Z del registro de estado es cero.

PC ← (PC) + $0002 + Rel **Si (C) + (Z) = 0**

La instrucción BHI normalmente se suele realizar después de alguna instrucción en la que se comparen datos, de manera que el valor del registro de estado refleje el resultado de la comparación. La operación lógica OR de los bits C y Z refleja si uno de los datos es mayor que el otro sin considerar el signo, es decir, en valor absoluto. La información acerca de la instrucción viene dada en la Tabla 4.40.

Tabla 4.40 Instrucción BHI.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
BHI	REL	22	3	9

Esta instrucción tampoco cambia el valor del registro de estado. Su secuencia de estados es la misma que la de las anteriores instrucciones de salto condicional y se puede ver en la Figura 4.10. En este caso la condición a contemplar en el estado "dato_a_bus" es la operación lógica OR comentada anteriormente.

4.7.7. BHS – Salto condicional si es mayor o igual

Esta instrucción es idéntica a la instrucción BCC, por lo que la información referente a ella se encuentra comentada en el apartado donde ya se ha explicado la instrucción BCC.

4.7.8. BIH – Salto condicional si el pin IRQ está a 1

Esta instrucción salta si el valor de la señal que recibe el microcontrolador mediante el pin dedicado a la interrupción externa IRQ es '1'.

PC ← (PC) + $0002 + Rel **Si IRQ = 1**

La instrucción BIH no altera el valor de ninguno de los bits del registro de estado y toda la información sobre la misma se puede ver en la Tabla 4.41.

Tabla 4.41 Instrucción BIH.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
BIH	REL	2F	3	9

De nuevo la secuencia de estados es la misma que las de las instrucciones anteriores y se puede observar en la Figura 4.10. En esta ocasión en la condición de salto se consulta el estado de la señal IRQ que es la que refleja el estado del pin de la interrupción externa.

4.7.9. BIL – Salto condicional si el pin IRQ está a 0

Esta instrucción es la contraria a la instrucción BIH. En este caso el salto se produce si el valor de la señal que recibe el microcontrolador por el pin IRQ es '0'.

PC ← (PC) + $0002 + Rel Si IRQ = 0

Esta instrucción de salto condicional al igual que las anteriores no modifica el valor de los bits del registro de estado. Los datos relacionados con la misma se muestran en la Tabla 4.42.

Tabla 4.42 Instrucción BIL.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
BIL	REL	2E	3	9

La secuencia de estados que sigue esta instrucción en su ejecución se puede observar en la Figura 4.10. En este caso en la condición de salto que se impone en el estado "dato_a_bus" es el valor de la señal IRQ.

4.7.10. BLO – Salto condicional si es menor

Esta instrucción es idéntica a la instrucción BCS comentada anteriormente. Por esta razón no se explicará de nuevo y toda la información relacionada con ella se puede ver en el apartado donde se detalla la instrucción BCS.

4.7.11. BLS – Salto condicional si es menor o igual

Esta instrucción salta si el resultado de la operación lógica OR de los bits Z y C del registro de estado es '1'. Es la instrucción contraria a la BHI.

PC ← (PC) + $0002 + Rel **Si (C) + (Z) = 1**

Normalmente esta instrucción se realiza después de ejecutar una operación que compare el valor de dos datos. El resultado de los datos queda reflejado en los valores que toma el registro de estado de manera que, si se realiza la operación lógica OR entre el bit Z y el bit C y el resultado es la unidad, quiere decir que el primero de los datos es menor o igual que el otro sin considerar el signo de ambos.

Esta instrucción no cambia el valor del registro de estado y la información relacionada con ella se puede contemplar en la Tabla 4.43.

Tabla 4.43 Instrucción BLS.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
BLS	REL	23	3	9

La secuencia de estados que sigue esta instrucción es también la que se presenta en la Figura 4.10. La diferencia reside en que la condición a consultar para esta instrucción es la operación lógica OR que se ha explicado anteriormente.

4.7.12. BMC – Salto condicional si el bit I está a 0

Esta instrucción sólo salta si la máscara de interrupción o bit I del registro de estado se encuentra a '0'. Es la instrucción contraria a la instrucción BMS que se explicará en los apartados siguientes.

PC ← (PC) + $0002 + Rel **Si (I) = 0**

La instrucción BMC tampoco modifica el registro de estado. Los detalles sobre sus código mnemónico, de operación y ciclos que tarda en ejecutarse se pueden observar en la

Tabla 4.44.

Tabla 4.44 Instrucción BMC.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
BMC	REL	2C	3	9

La secuencia de estados que sigue esta instrucción es la que se muestra en la Figura 4.10. Para que se ejecute concretamente esta instrucción, dentro de las condiciones que se consultan para su respectivo código de operación en el estado "dato_a_bus", hay que tener en cuenta el valor del bit I del registro de estado.

4.7.13. BMI – Salto condicional si es menor

Esta instrucción salta si el valor del bit N del registro de estado es '1'. Es la instrucción contraria a la instrucción BPL que se comentará más adelante.

PC ← (PC) + $0002 + Rel Si (N) = 1

La instrucción BMI generalmente se utiliza después de una instrucción que sirva para comparar datos, de manera que el registro de estado refleja los resultados de la comparación. Mediante la consulta del bit N del registro de estado se puede saber si el primero de los datos a comparar es menor que el otro.

Esta instrucción tampoco cambia ninguno de los valores del registro de estado. La información relacionada con ella se muestra en la Tabla 4.45.

Tabla 4.45 Instrucción BMI.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
BMI	REL	2B	3	9

En la secuencia de estados que se sigue para la ejecución de esta instrucción, la condición a consultar es el valor del bit N. Esta secuencia se puede ver en la Figura 4.10.

4.7.14. BMS – Salto condicional si el bit I está a 1

En esta instrucción se produce el salto si el valor de la máscara de interrupción o bit I del registro de estado es la unidad. Es la contraria a la instrucción BMC.

PC ← (PC) + $0002 + Rel Si (I) = 1

Esta instrucción al igual que el resto de instrucciones de salto condicional hasta ahora comentadas no modifica el valor de ninguno de los bits del registro de estado. Toda la información en relación con ella se encuentra en la Tabla 4.46.

Tabla 4.46 Instrucción BMS.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
BMS	REL	2D	3	9

La secuencia de estados de esta instrucción es la que siguen las anteriores instrucciones de salto condicional y se puede observar en la Figura 4.10. En este caso dentro del estado “dato_a_bus” la condición impuesta para saltar o no es el valor de la máscara de interrupción.

4.7.15. BNE – Salto condicional si no es igual

Esta instrucción salta cuando el valor del bit Z o bit cero del registro de estado se encuentra a ‘0’. Es la instrucción contraria a la instrucción BEQ.

PC ← (PC) + $0002 + Rel Si (Z) = 0

Esta instrucción suele tener lugar después de haber ejecutado alguna de las instrucciones que sirven para comparar datos, de tal modo que el bit Z tras estas instrucciones es el que refleja si los datos son iguales o no.

Esta instrucción no varía el valor de ninguno de los bits del registro de estado y los datos que tienen relación con ella se muestran en la Tabla 4.47.

Esta instrucción sigue la secuencia de estados que se puede ver en la Figura 4.10. Esta secuencia es la misma que se sigue en todas las instrucciones de salto condicional vistas hasta ahora. La diferencia entre ellas se encuentra en la condición que se impone en el estado “dato_a_bus” para saltar o no. Esta condición en este caso es el valor del bit Z.

Tabla 4.47 Instrucción BNE.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
BNE	REL	26	3	9

4.7.16. BPL – Salto condicional si es positivo

Esta instrucción salta si el valor del bit N del registro de estado es '0'. Es la instrucción contraria a la instrucción BMI.

PC ← (PC) + $0002 + Rel Si (N) = 0

Esta instrucción tampoco modifica el valor del registro de estado. En la Tabla 4.48 se encuentra reflejada toda la información acerca de la instrucción.

Tabla 4.48 Instrucción BPL.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
BPL	REL	2A	3	9

La secuencia de estados es de nuevo la que se puede observar en la Figura 4.10. En este caso la condición a consultar para saltar o no en el estado "dato_a_bus" es el valor del bit N.

4.7.17. BRA – Salto incondicional

Esta instrucción es un poco diferente a las anteriores que se han explicado, ya que no es necesaria ninguna condición para realizar el salto o no, sino que este siempre se va a llevar a cabo.

PC ← (PC) + $0002 + Rel

Esta instrucción no cambia el valor de los bits del registro de estado. La información sobre la misma se puede observar en la Tabla 4.49.

Tabla 4.49 Instrucción BRA.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
BRA	REL	20	3	9

La secuencia de estados que se sigue para la ejecución de la instrucción es la misma que la que se puede ver en la Figura 4.10. En este caso no hay bifurcación después del estado “dato_a_bus” ya que siempre se salta.

4.7.18. BRCLR n – Salto condicional si el bit n es 0

Esta instrucción salta si un determinado bit de una posición de memoria se encuentra a ‘0’.

PC ← (PC) + $0003 + Rel **Si el bit n de M = 0**

Las posiciones de memoria que pueden ser consultadas para realizar el salto o no tienen que estar desde la $0000 a la $00FF ya que estas vienen dadas siguiendo un direccionamiento directo.

Esta instrucción va a modificar el bit C del registro de estado tomando el valor del bit de la posición de memoria que se consulta. Toda la información acerca de la misma se puede ver en la Tabla 4.50.

Tabla 4.50 Instrucción BRCLR.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
BRCLR 0,(opr)	DIR	01	5	15
BRCLR 1,(opr)	DIR	03	5	15
BRCLR 2,(opr)	DIR	05	5	15
BRCLR 3,(opr)	DIR	07	5	15
BRCLR 4,(opr)	DIR	09	5	15
BRCLR 5,(opr)	DIR	0B	5	15
BRCLR 6,(opr)	DIR	0D	5	15
BRCLR 7,(opr)	DIR	0F	5	15

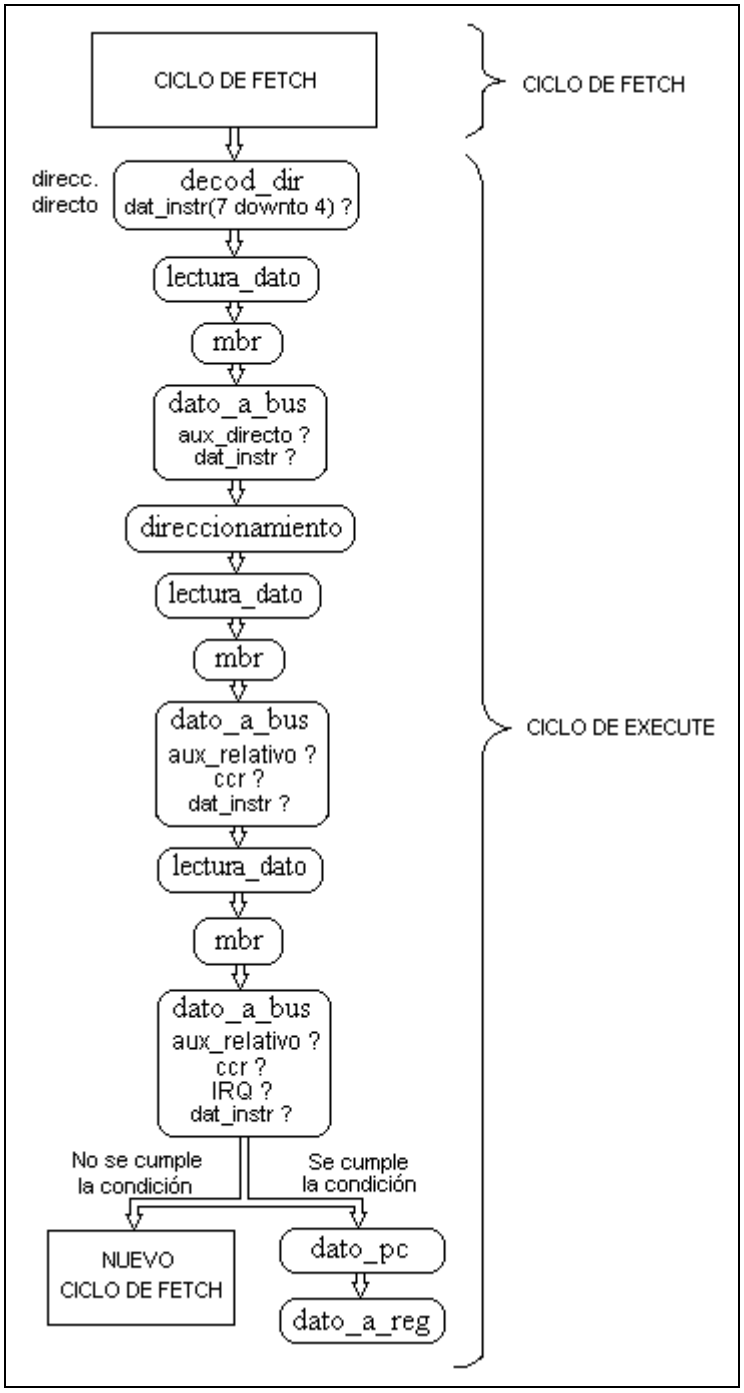


Figura 4.11 Secuencia de estados de la instrucción BRCLR y BRSET.

La secuencia de estados que sigue esta instrucción varía respecto a las de las anteriores instrucciones de salto condicional. La nueva secuencia de estados se muestra en la Figura 4.11. En ella se puede ver como hay que realizar una lectura de memoria mediante direccionamiento directo para obtener el dato de memoria donde se encuentra el bit cuyo estado se va a consultar. Una vez que se tiene el dato de la posición de memoria se lleva a la ALU. Este toma como bit C el valor del bit seleccionado además

de que se carga este valor en el registro de estado. Una vez hecho esto se pasa a leer el byte de *offset* y es ahora, en el estado "dato_a_bus", cuando se decide si se realiza el salto o no dependiendo del valor del bit C. Si tiene lugar el salto se suma el dato de *offset* al valor del contador de programa en ese instante.

4.7.19. BRN – Nunca salta

Esta instrucción nunca salta, siempre continúa con el programa leyendo la siguiente dirección que contiene el contador de programa sin sumarle ningún *offset*. Es la instrucción contraria a la instrucción BRA.

PC ← (PC) + $0002

Esta instrucción no va a cambiar el valor del registro de estado. La información relacionada con ella está reflejada en la Tabla 4.51.

Tabla 4.51 Instrucción BRN.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
BRN	REL	21	3	4

Para la instrucción BRN la secuencia de estados también es la misma que se muestra en la Figura 4.10. En este caso lo que ocurre es que en el estado "dato_a_bus" no se consulta ninguna condición y, a partir de él siempre se sigue la opción de leer una nueva instrucción de programa.

4.7.20. BRSET n – Salto condicional si el bit n es 1

Esta instrucción salta si el valor de uno de los bits de una determinada posición de memoria es '1'. Esta instrucción es contraria a la instrucción BRCLR.

PC ← (PC) + $0003 + Rel **Si el bit n de M = 1**

Esta instrucción cambia únicamente el valor del bit C del registro de estado. Este bit toma el valor del bit que impone la condición de salto. Toda los datos sobre esta instrucción se encuentran en la Tabla 4.52.

Tabla 4.52 Instrucción BRSET.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
BRSET 0,(opr)	DIR	00	5	15
BRSET 1,(opr)	DIR	02	5	15
BRSET 2,(opr)	DIR	04	5	15
BRSET 3,(opr)	DIR	06	5	15
BRSET 4,(opr)	DIR	08	5	15
BRSET 5,(opr)	DIR	0A	5	15
BRSET 6,(opr)	DIR	0C	5	15
BRSET 7,(opr)	DIR	0E	5	15

La secuencia de estados de la instrucción BSET es la que se muestra en la Figura 4.11. Esta secuencia es similar a la de la instrucción BRCLR con la diferencia de que en este caso se salta si el valor del bit C (que posee el valor de correspondiente bit de la memoria) se encuentra a '1'.

4.7.21. BSR – Salto a subrutina

Esta instrucción salta a una subrutina siguiendo un direccionamiento relativo. Para ello lee el byte de *offset*, guarda los valores del contador de programa en la pila y, por último, realiza el salto a subrutina sumando el *offset* relativo al valor que tenga en ese momento el contador de programa.

PC ← (PC) + $0002

↓ (PCL); SP ← (SP) - $0001

↓ (PCL); SP ← (SP) - $0001

PC ← (PC) + Rel

En la instrucción BSR es necesario guardar el valor del contador de programa en la pila para que el microcontrolador sepa la dirección donde debe continuar el programa cuando se realice el retorno de subrutina.

Esta instrucción no modifica el valor de ninguno de los bits del registro de estado. Toda la información acerca de la misma se encuentra en la Tabla 4.53.

La secuencia de estados que tiene lugar para poder ejecutar esta instrucción se puede ver en la Figura 4.12. Esta secuencia sigue los pasos ya comentados en la explicación de la instrucción.

Tabla 4.53 Instrucción BSR.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
BSR	REL	AD	3	14

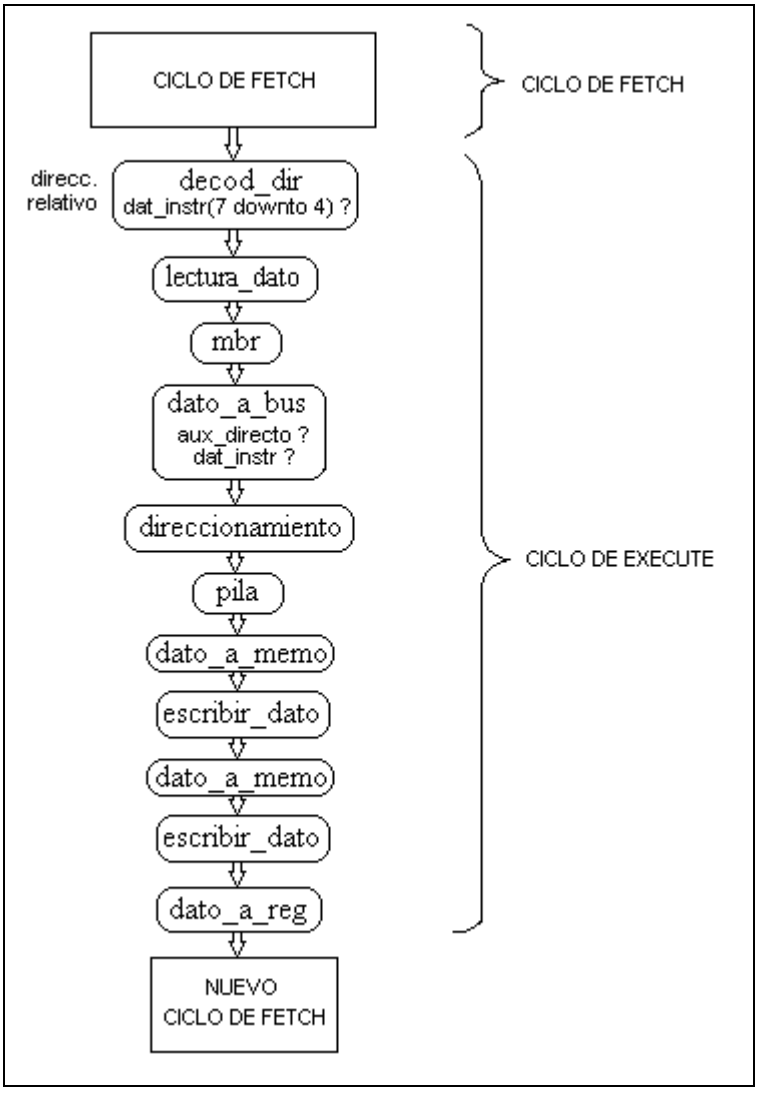


Figura 4.12 Secuencia de estados de la instrucción BSR.

4.7.22. JMP – Salto

Esta instrucción realiza un salto a la dirección que le viene dada por los operandos que acompañan al código de operación de la instrucción.

PC ← Dirección efectiva

Los modos de direccionamiento por los cuales se calcula la dirección efectiva a donde se quiere saltar junto con otros datos acerca de la instrucción se pueden ver en la Tabla 4.54.

Tabla 4.54 Instrucción JMP.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorota	Ciclos Implementación
JMP(opr)	DIR	BC	2	8
JMP (opr)	EXT	CC	3	11
JMP,X	IX	FC	4	5
JMP (opr),X	IX8	EC	3	8
JMP (opr),X	IX16	DC	2	11

Esta instrucción no altera el valor de ninguno de los bits del registro de estado. La secuencia de estados que sigue la instrucción JMP viene ilustrada mediante la Figura 4.13.

4.7.23. JSR – Salto a subrutina

Esta instrucción salta a una subrutina cuya dirección efectiva viene dada por el modo de direccionamiento y los operandos que siguen a la instrucción.

PC ← (PC) + n (n=1,2 o 3)

↓ (PCL); SP ← (SP) - $0001

↓ (PCL); SP ← (SP) - $0001

PC ← (PC) + Dirección efectiva

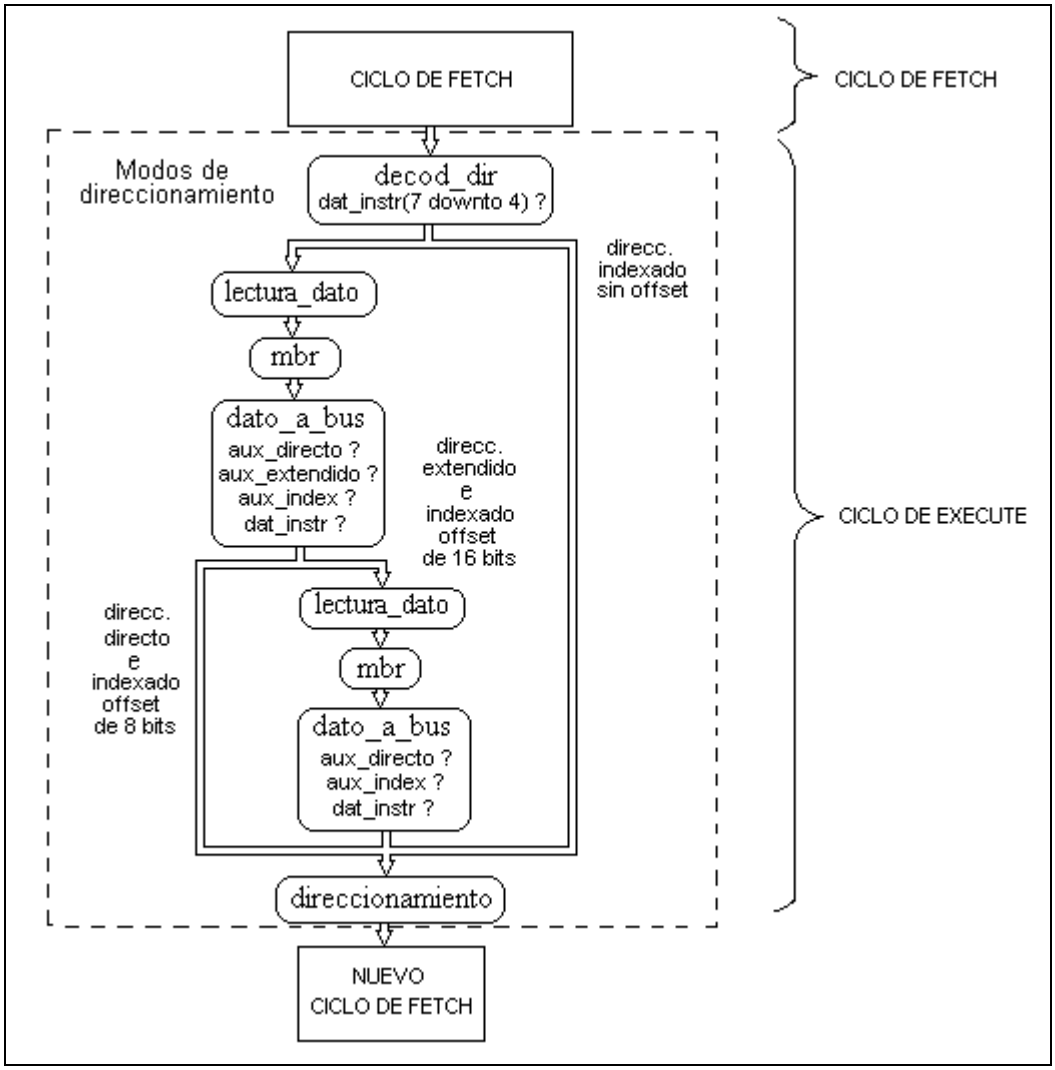


Figura 4.13 Secuencia de estados de la instrucción JMP.

En la instrucción JSR primero se obtiene la dirección donde se encuentra la subrutina. A continuación, se almacena el valor del contador de programa en la pila. Esto va a permitir volver al programa a la dirección correcta al terminar de ejecutar la subrutina. Por último, se carga la dirección donde está situada la subrutina en el contador de programa.

Tabla 4.55 Instrucción JSR.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorota	Ciclos Implementación
JSR(opr)	DIR	BD	5	13
JSR (opr)	EXT	CD	6	16
JSR,X	IX	FD	7	10

JSR (opr),X	IX8	ED	6	13
JSR (opr),X	IX16	DD	5	16

Esta instrucción no modifica el valor del registro de estado. Toda la información relacionada con ella se encuentra en la Tabla 4.55.

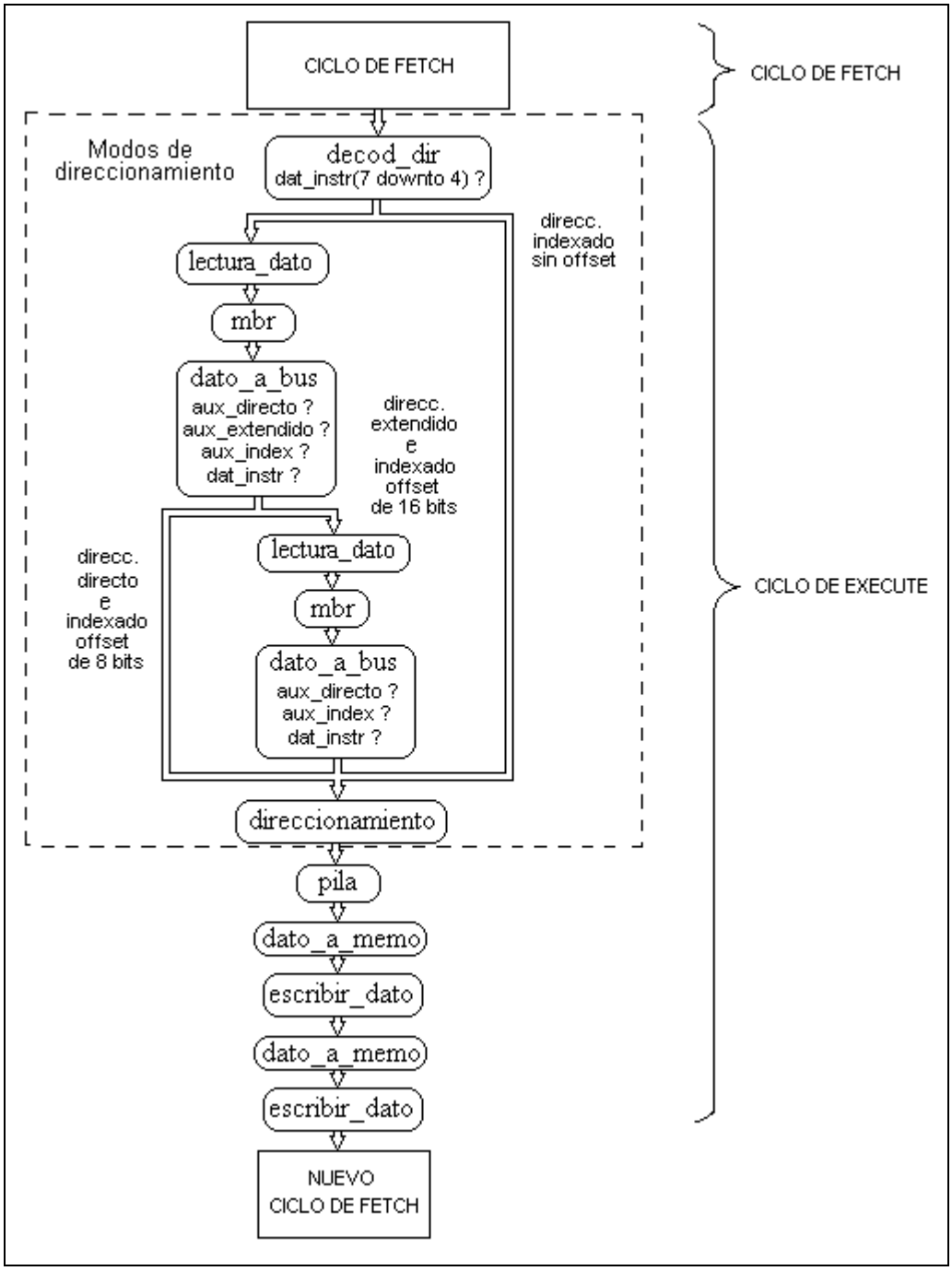


Figura 4.14 Secuencia de estados de la instrucción JSR.

La secuencia de estados que tiene lugar en el decodificador de instrucciones para la ejecución de la instrucción se muestra en la Figura 4.14.

4.8. INSTRUCCIONES DE CONTROL

4.8.1. NOP – No operación

Esta instrucción no realiza ninguna operación sino que, simplemente incrementa el contador de programa para leer la siguiente instrucción.

Ningún valor de los registros ni de la memoria es alterado. La información sobre ella se puede ver en la Tabla 4.56.

Tabla 4.56 Instrucción NOP.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
NOP	INH	9D	2	4

La secuencia de estados de esta instrucción es la que se muestra en la Figura 4.9. La diferencia está en que para la ejecución de la instrucción NOP no se altera ninguno de los bits del registro de estado como pasa en otras instrucciones como CLC, CLI, SEC y SEI.

4.8.2. RSP – Reset del puntero de pila

Esta instrucción produce un *reset* del puntero de pila, es decir, se carga el valor $00FF en el registro puntero de pila.

SP ← $00FF

Esta instrucción no altera el valor del registro de estado. En la Tabla 4.57 se puede ver la información referente a la instrucción.

La secuencia es la misma que la de la instrucción NOP y que las instrucciones CLC, CLI, SEC y SEI y se puede observar en la Figura 4.9. En este caso la máquina de estados sólo se encarga de leer la instrucción y de cargarla en el registro de instrucción, ya que el *reset* del puntero de pila se ha diseñado de manera que se produzca cuando

ocurra un *reset* general del microcontrolador y cuando el código de operación de esta instrucción se encuentre en el registro de instrucción.

Tabla 4.57 Instrucción RSP.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
RSP	INH	9C	2	4

4.8.3. RTI – Retorno de interrupción

Esta instrucción se encarga de devolver los valores que tenían todos los registros antes de que ocurriera una interrupción. A su vez continúa con la siguiente instrucción que hubiera tenido lugar en el caso de que no hubiera sucedido una interrupción.

SP ← (SP) + $0001; ↑ CCR;

SP ← (SP) + $0001; ↑ ACCA;

SP ← (SP) + $0001; ↑ X;

SP ← (SP) + $0001; ↑ PCH;

SP ← (SP) + $0001; ↑ PCL;

La instrucción RTI suele ser la última instrucción de la rutina de interrupción. Se encarga de leer los valores guardados en la pila antes de pasar a la ejecutar la rutina de interrupción y cargarlos en su correspondiente registro. El orden que se sigue para guardar cada uno de los datos es el que se muestra en las líneas anteriores. En la Tabla 4.58 se muestra la información relacionada con esta instrucción.

En esta instrucción se modifican todos los valores del registro de estado ya que este, al igual que los otros registros, recupera los valores que tenía antes de que saltara la interrupción.

Tabla 4.58 Instrucción RTI.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
RTI	INH	80	9	25

La secuencia de estados que se sigue en esta instrucción es la que se puede ver en la Figura 4.15.

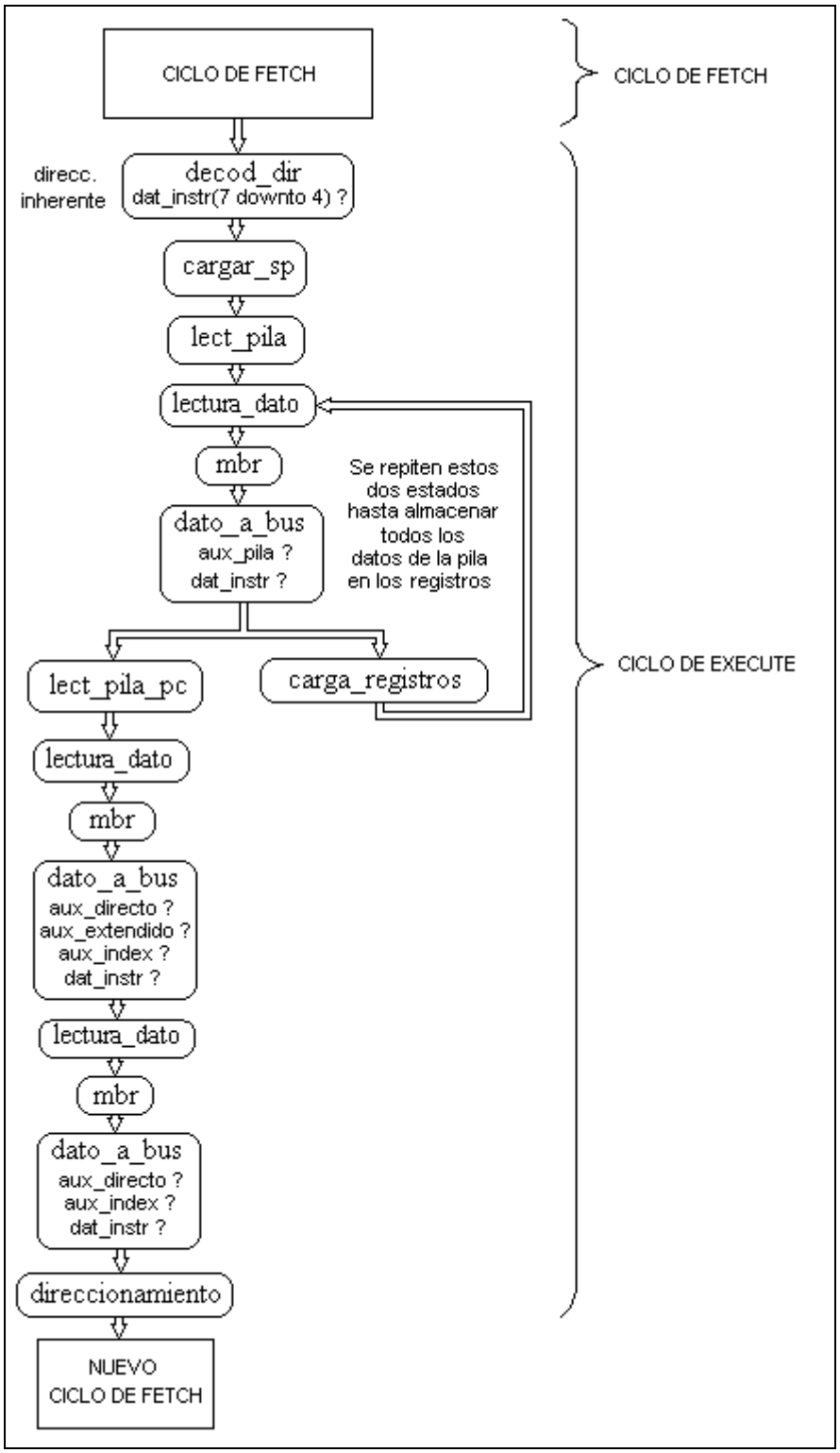


Figura 4.15 Secuencia de estados de la instrucción RTI.

4.8.4. RTS – Retorno de subrutina

Esta instrucción se encarga de recuperar el valor del contador de programa después de un salto a subrutina.

SP ← (SP) + $0001; ↑ PCH;

SP ← (SP) + $0001; ↑ PCL;

Por lo general la instrucción RTS va situada al final de la subrutina. Su función es la leer de la pila los datos del contador de programa y almacenarlos en él. De esta forma el programa continuará leyendo de la memoria la dirección que va después de la que almacena la instrucción de salto a subrutina.

La instrucción RTS no modifica ninguno de los bits del registro de estado. La información sobre esta instrucción se encuentra en la Tabla 4.59.

Tabla 4.59 Instrucción RTS.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
RTS	INH	81	6	12

La secuencia de estados de la instrucción RTS se muestra en la Figura 4.16. Es similar a la última parte de la instrucción SWI, ya que la recuperación de los datos del contador de programa de la pila se realiza de la misma forma.

4.8.5. STOP – Para el oscilador

Esta instrucción habilita la interrupción externa $\overline{\text{IRQ}}$ y para el oscilador del microcontrolador. Es una de las instrucciones de bajo consumo que posee el microcontrolador. Una vez que se ejecuta se reduce el consumo de potencia, eliminando así toda disipación de potencia dinámica.

Si se produce un *reset* del microcontrolador o se activa la interrupción externa el oscilador vuelve de nuevo a activarse, y se realiza la correspondiente rutina de *reset* o de atención a la interrupción.

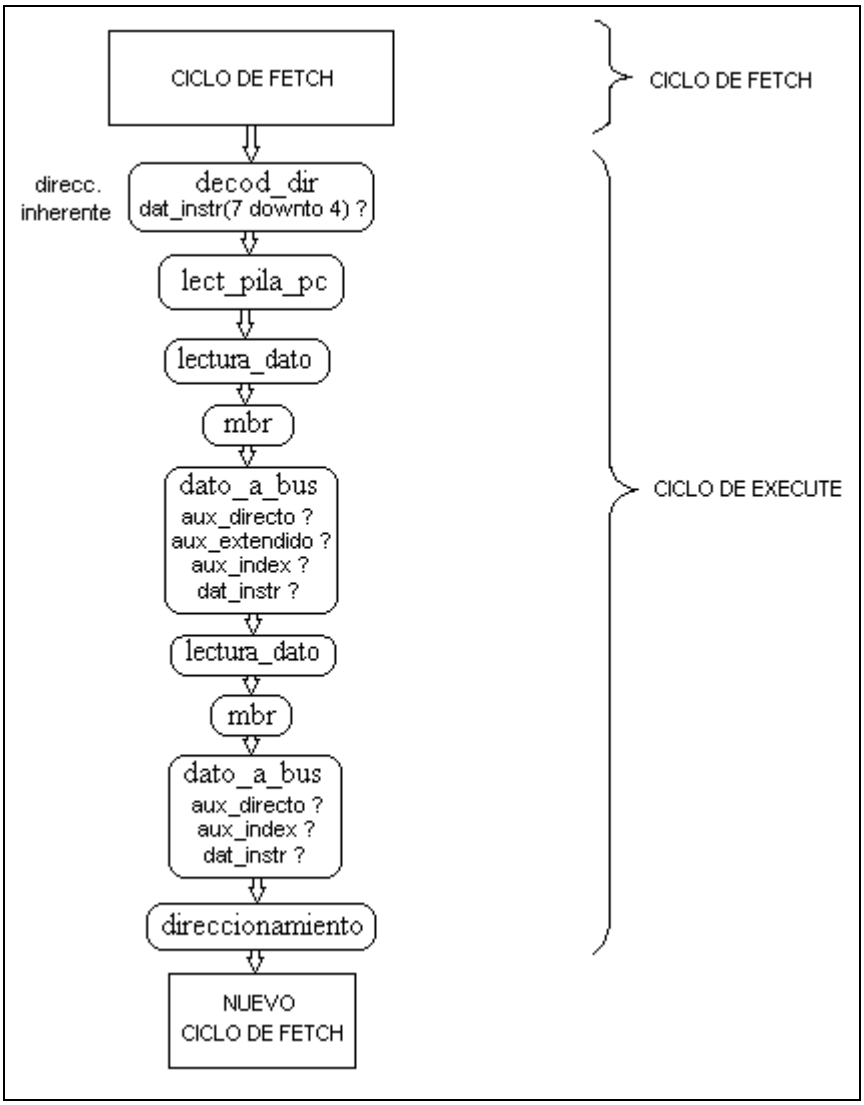


Figura 4.16 Secuencia de estados de la instrucción RTS.

Esta instrucción modifica el bit I o máscara de interrupción de manera este toma siempre el valor '0'. De esta manera se habilita la interrupción externa. Toda la información relacionada con la instrucción se puede ver en la Tabla 4.60.

Tabla 4.60 Instrucción STOP.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
STOP	INH	8E	2	5

La secuencia de estados para que tenga lugar esta instrucción se muestra en la Figura 4.17. Las acciones que realiza son la de leer la instrucción de la memoria y almacenarla en el registro de instrucción además de dar al bit I el valor '0'. De esta forma prepara al microcontrolador para poder recibir, además de un *reset*, la interrupción externa.

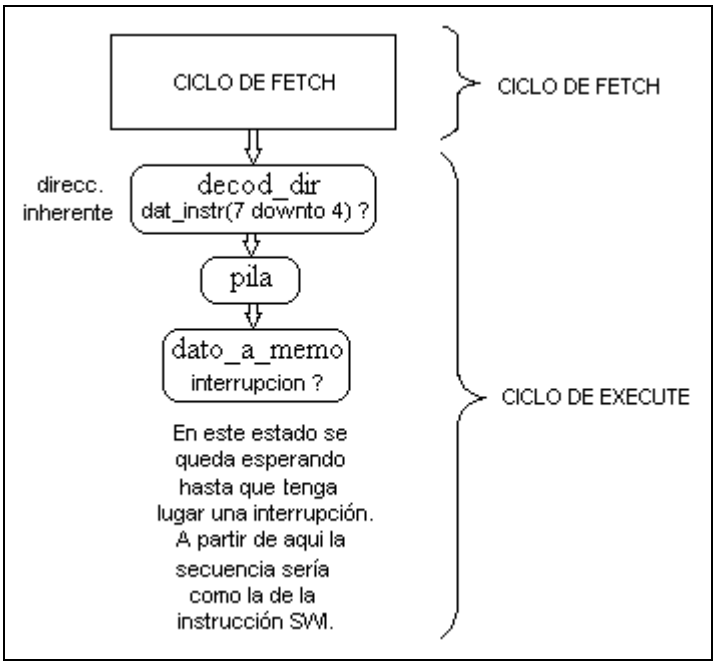


Figura 4.17 Secuencia de estados de las instrucciones WAIT y STOP.

4.8.6. SWI – Interrupción por software

Esta instrucción se trata de la interrupción *software* SWI. Como ya se ha comentado anteriormente, es una interrupción no enmascarable y que ocurre siempre que se encuentre dentro del programa almacenado en memoria.

PC ← (PC) + $0001

↓ (PCL); SP ← (SP) - $0001

↓ (PCL); SP ← (SP) - $0001

↓ (X); SP ← (SP) - $0001

↓ (ACCA); SP ← (SP) - $0001

↓ (CCR); SP ← (SP) - $0001

I ← 1

PCH ← ($1FFE)

PCL ← ($1FFF)

Los pasos a seguir cuando tiene lugar esta instrucción son los mismos que se siguen a la hora de atender a cualquier interrupción y que ya se han explicado cuando se detallaron todas las interrupciones. No obstante de forma simplificada, estos pasos son:

1. Se guardan todos los valores de los diferentes registros en la pila.
2. Se pone el bit I a '1' para que no tenga lugar otra interrupción.
3. Se lee el vector de interrupción para localizar la dirección donde se encuentra la subrutina de interrupción.

El único bit que altera su valor es el bit I que durante la ejecución de la instrucción SWI siempre toma el valor lógico '1'. En la Tabla 4.61 se pueden observar los datos relacionados con ella.

Tabla 4.61 Instrucción SWI.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
SWI	INH	83	10	23

La secuencia de estados que tiene lugar para ejecutar esta instrucción es la que se muestra en la Figura 4.18. Una vez que se almacenan todos los valores de los diferentes registros en la pila, se pasa a leer los vectores de interrupción donde se encuentra la dirección de la subrutina de interrupción mediante un direccionamiento extendido.

4.8.7. WAIT – Para la CPU

Es otra de las instrucciones de bajo consumo que junto con la instrucción de STOP posee el microcontrolador. Esta instrucción se encarga de parar el reloj de la CPU y habilitar la máscara de interrupción.

Una vez que el microcontrolador ejecuta la instrucción WAIT se queda esperando hasta que cualquiera de las interrupciones tiene lugar. A partir de entonces atiende a la interrupción de la misma forma que siempre.

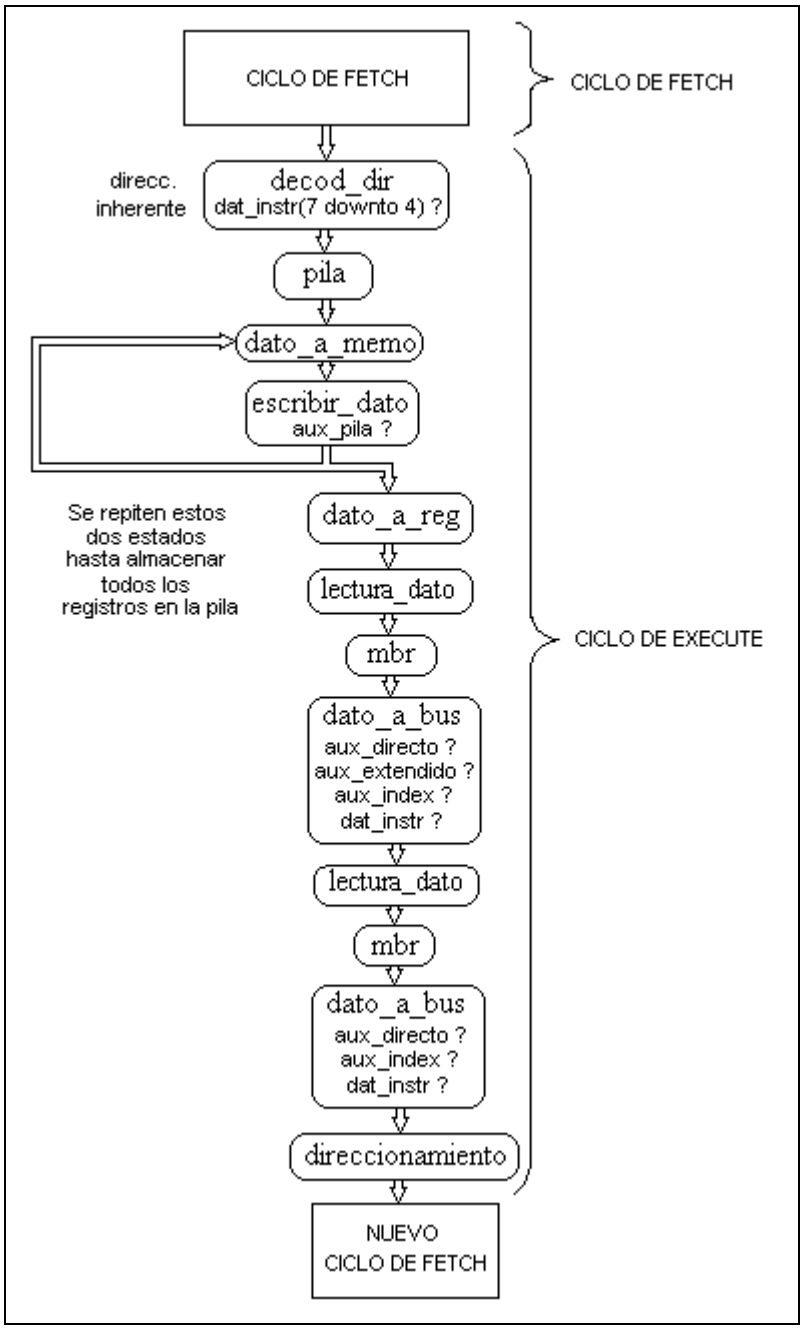


Figura 4.18 Secuencia de estados de la interrupción *software* SWI.

El único bit del registro de estado que modifica esta instrucción es la máscara de interrupción haciendo que tome el valor '0'. Esta es la manera que puedan tener lugar cualquiera de las interrupciones al habilitar su máscara.

Todos los datos que guardan relación con la instrucción se pueden ver en la Tabla 4.62.

Tabla 4.62 Instrucción WAIT.

Mnemónico	Modo de direccionamiento	Código de operación	Ciclos Motorola	Ciclos Implementación
WAIT	INH	8F	2	5

La secuencia de estados que se lleva a cabo en la ejecución de esta instrucción se puede observar en la Figura 4.17. Esta es la misma secuencia que para la instrucción STOP. Las diferencias entre una instrucción y otra se dan una vez que la máquina de estados ha cargado el código de operación de la instrucción y se han habilitado las interrupciones. Es entonces cuando la CPU consulta el valor del registro de instrucción y dependiendo de él tiene en cuenta las interrupciones que afectan a cada instrucción. Una vez que alguna de ellas tiene lugar y en el caso de que esté permitida, se comienza con la rutina de atención a la correspondiente interrupción.

CAPÍTULO 5

REALIZACIÓN DE PRUEBAS Y RESULTADOS OBTENIDOS

5. REALIZACIÓN DE PRUEBAS Y RESULTADOS OBTENIDOS

Todo diseño de cualquier circuito microelectrónico debe pasar por las siguientes fases:

1. Realización del diseño. Como la misma palabra lo dice se trata de elaborar el diseño del circuito electrónico. Para ello, en este proyecto se ha utilizado el lenguaje VHDL (*VHSIC Hardware Description Language*). Este es un lenguaje estándar utilizado para realizar descripción *hardware* en el más alto nivel de abstracción para dispositivos programables, como es el caso del presente proyecto en el que el dispositivo empleado para la implementación del diseño es una FPGA (*Field Programmable Gate Array*).
2. Síntesis del diseño. Es el proceso por el cuál se transforma la descripción en lenguaje VHDL de la etapa anterior, a un conjunto de puertas lógicas, en base a una librería seleccionada. El resultado es un archivo de "netlist" o archivo de conexiones. El proceso de síntesis por lo general depende de la tecnología y se suele realizar con las herramientas de *software* que proporciona el fabricante de la FPGA donde posteriormente se va implementar el diseño.

   Una vez concluido este proceso se recomienda realizar una simulación funcional *pre-layout* del mismo. Esta simulación sirve tanto para corregir errores de sintaxis en el código, como para comprobar el buen funcionamiento del diseño, es decir, que las salidas del diseño respondan a los estímulos de entrada de la manera deseada.
3. Implementación del diseño. En esta etapa se realiza una sínteis física, es decir, se traduce el diseño de puertas lógicas tomando el archivo creado en la fase anterior a elementos particulares del dispositivo dónde se va a programar. Se divide a su vez en tres etapas:
   a. Optimización. Esta etapa consiste en la mejora de las expresiones lógicas en función de los recursos disponibles en la FPGA y de determinadas restricciones introducidas por el usuario, como pueden ser restricciones de área o de tiempo.
   b. Mapeado. Se trata de la asignación de los elementos lógicos del diseño identificados en los procesos de síntesis y optimización, a elementos

físicos específicos que actualmente implementan las funciones lógicas en la FPGA elegida para introducir el diseño.

c. *Place & Route*. Se sitúan los elementos dentro de la FPGA y se realizan las conexiones necesarias entre ellos.

Después de pasar por esta fase de diseño se puede realizar una simulación funcional *post-layout* en la que ya se tienen en cuenta los retardos introducidos por los elementos físicos de la FPGA.

4. Programación de la placa. El diseño ya está listo para grabarlo en la FPGA y así poder comprobar ya de manera física su correcto funcionamiento.

Todas estas fases que componen el flujo de diseño se pueden observar en la Figura 5.1.

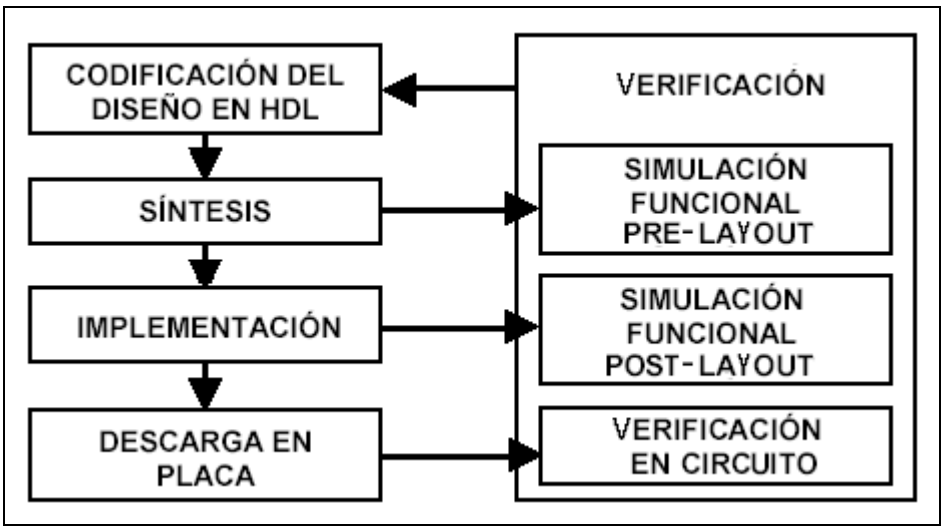


Figura 5.1 Flujo de diseño de un circuito electrónico.

En este capítulo se van a explicar cuáles han sido las diferentes fases del diseño de este proyecto resaltando las principales pruebas que se han realizado para comprobar el correcto funcionamiento del circuito. También se explicarán los resultados obtenidos después de realizar la síntesis y optimización del mismo. Para ello, el capítulo se ha dividido en varios apartados.

En el primero de ellos se va a comentar como se ha realizado la simulación funcional *pre-layout* del circuito. Esta simulación se realiza una vez que se tiene el diseño escrito en código VHDL y antes de la síntesis. En ella se pueden ver los valores que toman cada una de las señales de salida e internas del microcontrolador según se

van ejecutando las diferentes instrucciones y van cambiando los valores de las señales de entrada.

En el segundo de los apartados se explica la segunda prueba que se ha realizado en el circuito, que ha sido ya su prueba en una placa de desarrollo. Esto ha tenido lugar una vez se ha sintetizado e implementado el diseño y habiendo generado antes el fichero de programación de la FPGA. En este apartado se comentará la placa utilizada y cómo se han llevado a cabo las pruebas necesarias del diseño completo.

En el último de los apartados se detallarán los resultados de la síntesis e implementación del diseño teniendo en cuenta dos de las principales restricciones que se le suelen imponer a todo circuito electrónico, es decir, cuál es el área que ocupa y cuál es la frecuencia máxima que puede llegar a alcanzar.

5.1. SIMULACIÓN DEL NÚCLEO DEL MICROCONTROLADOR

La simulación funcional *pre-layout* es la primera prueba que se ha realizado para comprobar el correcto funcionamiento del núcleo una vez introducido el diseño en lenguaje VHDL. En esta simulación se pueden comprobar aparte de los errores sintácticos en al escritura del código, los errores a la hora de ver si el circuito reacciona como el diseño lo exige ante ciertos estímulos de entrada. Para ello se ha empleado la herramienta Synopsys®. Mediante la utilización de la misma se han realizado diversas simulaciones hasta comprobar el funcionamiento completo de todas las instrucciones disponibles en el núcleo del 68HC05, así como los procesos que se desarrollan tanto si tiene lugar una interrupción como un *reset*.

La manera de poder comprobar el buen funcionamiento del núcleo del microcontrolador es mediante la unión de este con otro dispositivo que actúe de memoria. Esta memoria va a ser imprescindible ya que en ella es donde va a estar almacenado el programa a ejecutar cuyas instrucciones son las que se van a probar.

Para hacer más fácil la prueba de las diferentes instrucciones estas se programan en ensamblador en fichero con extensión ".s", escribiendo en él sus correspondientes códigos mnemónicos. Este fichero se compila y ensambla con la herramienta de *software* ASHC5ASM proporcionada por Motorota para programar estos microcontroladores, de manera que se obtiene el código máquina. Este código máquina contiene los códigos de operación de cada instrucción y de los datos necesarios para

ejecutar cada una de ellas y será el que deba contener la memoria con al que interacciona el microcontrolador. Todo esto se obtiene en un fichero con extensión ".S19".

La forma de realizar el módulo memoria ha sido crear una matriz de las dimensiones de la memoria, es decir, una matriz con 8192 posiciones de 8 bits cada una. Esta matriz se rellena con los datos del fichero ".S19" de manera que el componente de memoria diseñado en lenguaje VHDL lee los datos del fichero y se los va asignando a cada una de las 8192 posiciones. A parte de esta matriz, este componente lleva el código necesario de manera que se pueda leer y escribir en ella como si de una memoria RAM se tratará, según se lo indiquen las señales que recibe desde el núcleo del microcontrolador. La operación de almacenar el programa en la memoria se realiza en el momento en que el microcontrolador empieza a funcionar.

Una vez que se tiene la memoria es necesario unirla con el núcleo. Para ello se han instanciado ambas en un módulo de mayor jerarquía en el cuál se han unido ambos componentes.

Para poder realizar la simulación del diseño es necesario crear un *testbench* (banco de pruebas) donde se asignan valores en cada instante de tiempo a todas las entradas y salidas del diseño. A partir de este fichero la herramienta Synopsys® genera la simulación funcional dando valores a todas las salidas de manera que respondan a los estímulos de entrada según se ha diseñado.

Estas simulaciones *pre_layout* no son capaces de detectar posibles errores de sincronismo entre registros debido a que no contemplan los retardos entre los diferentes componentes de la FPGA utilizados en el diseño al realizarse estos procesos de manera estándar. Para poder detectar estos errores es necesario sintetizar el diseño con la una tecnología concreta. Es por ello que después de comprobar la simulación funcional *pre-layout* con Synopsys® se pasa a utilizar la herramienta ISE de Xilinx® propia del fabricante de la FPGA donde se va a introducir el diseño. Con esta herramienta es con la ya se han realizado todos los pasos posteriores que componen el flujo del diseño hasta llegar a programarlo dentro de la FPGA. Todos estos pasos se comentarán en los siguientes apartados.

En la Figura 5.2 se puede ver el resultado de una de estas simulaciones. En ella se muestra el cronograma de la ejecución de la instrucción LDA con un

direccionamiento inmediato. El dato que se almacena en el acumulador es el dato $C7. En la señal “actual” se pueden observar los estados por los que va pasando la máquina de estado del decodificador de instrucciones. Una vez que se llega al estado “dato_a_bus”, en la señal de entrada al acumulador se encuentra el dato a almacenar en él y, la señal de carga es activada de manera que el dato queda almacenado en él. Por otro lado se puede ver como la señal “dat_sal_ccr” que contempla el valor del registro de estado muestra como el bit N modifica su valor. Esto es debido a que dentro de esta instrucción este bit se activa si el dato que se guarda en el acumulador tiene el bit más significativo a ‘1’, cosa que ocurre en este momento. En las demás señales se pueden observar los valores que van tomando en cada instante para, en un primer momento, lograr almacenar el código de operación de la instrucción en el registro de instrucción para luego, más tarde, leer el operando de la instrucción. En este caso este operando es directamente el dato a almacenar en el acumulador al ser un direccionamiento inmediato.

5.2. PRUEBAS EN PLACA DEL NÚCLEO DEL MICROCONTROLADOR

Una vez comprobado el diseño en simulación se pasa a realizar la síntesis e implementación del mismo. Todo este proceso lo realiza automáticamente la herramienta ISE de Xilinx® indicándole ciertos parámetros a cerca de las restricciones y especificaciones que se deseen. Una vez concluidas todas estas fases, la herramienta genera un fichero con el que más tarde se programa la FPGA además de otros muchos donde se pueden observar todos los datos acerca de las fases de la síntesis y de la implementación.

La última fase del diseño es programar la FPGA con el fichero de salida. Para que esto tenga lugar es necesario que existan otros elementos interactuando con ella. Es por ello que la FPGA utilizada en este proyecto forma parte de una placa de desarrollo. Todos la información sobre la FPGA, su placa de desarrollo y los elementos que la componen se detalla a continuación.

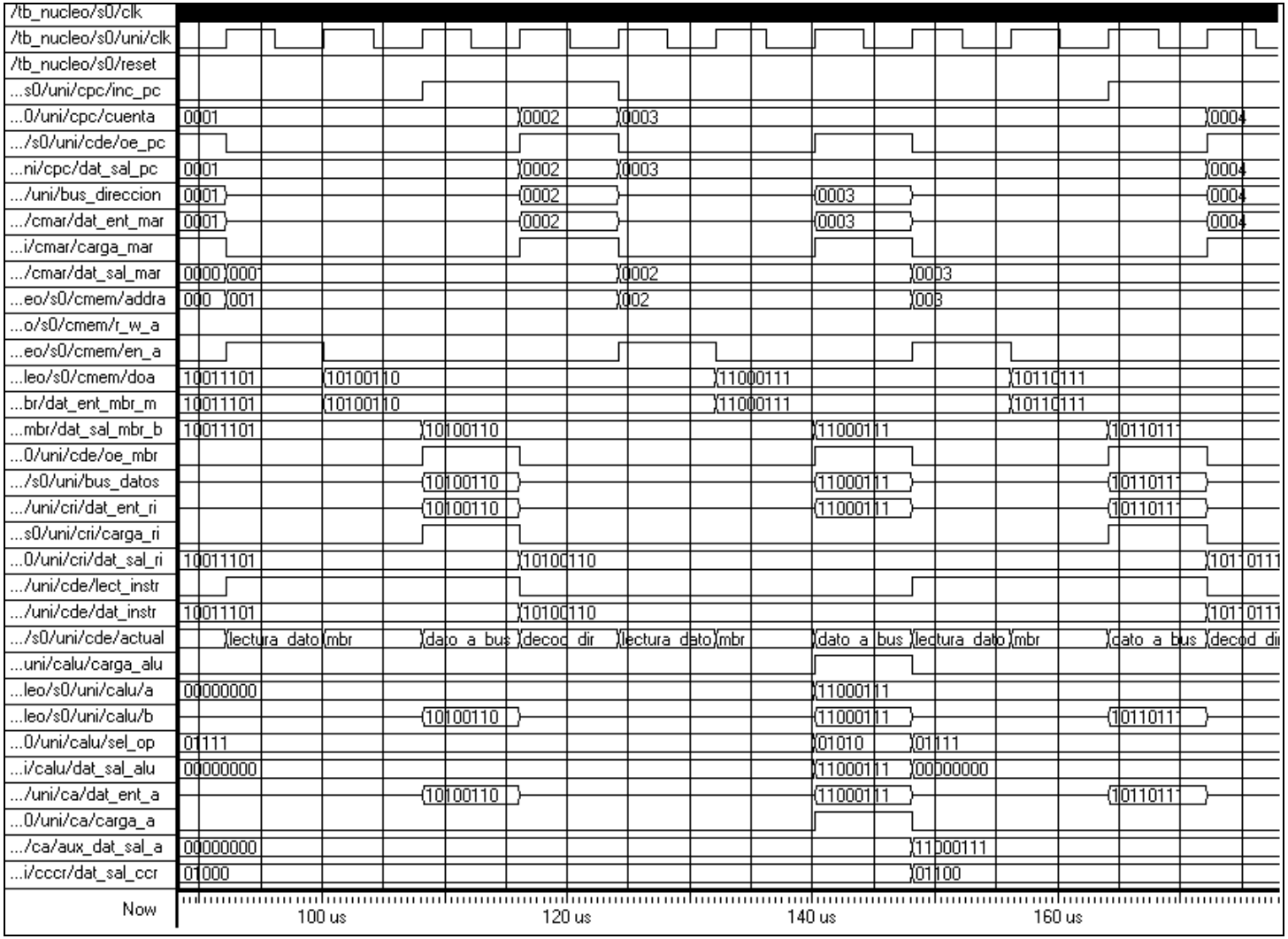


Figura 5.2 Simulación del núcleo.

La placa de desarrollo empleada en este proyecto es el modelo Spartan-IIE de Xilinx®. Esta placa está compuesta además de por una FPGA, por otros componentes como por ejemplo: la alimentación, el reloj, la interfaz JTAG para comunicarse con el ordenador y poder grabar en ella el circuito que se desea probar, elementos que muestren salidas, etc. Una fotografía de la placa utilizada se muestra en la Figura 5.3.

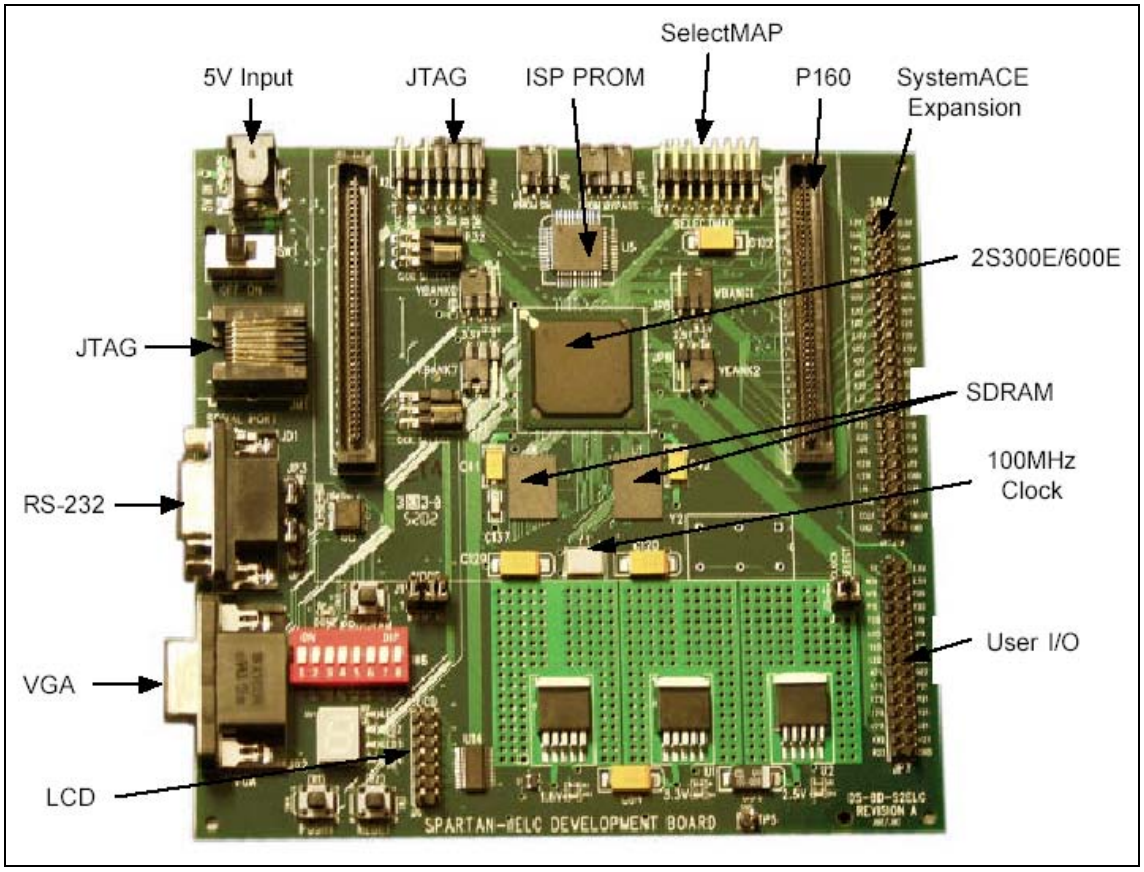


Figura 5.3 Placa de desarrollo utilizada en las pruebas del núcleo.

Aunque la placa Spartan-IIE dispone de varios componentes como ya se ha comentado antes, sólo se van a explicar aquellos que se han utilizado en la verificación del diseño objeto de este proyecto. Estos han sido los siguientes:

- **FPGA**: esta placa utiliza una FPGA que pertenece a la familia de Xilinx® Spartan-IIE. En concreto el modelo es XC2S300E –6FG4 56C compuesto por 300.000 puertas equivalentes.
- **Display**: esta placa dispone de un display 7 segmentos en ánodo común. Los diodos LED del display se encienden cuando la señal correspondiente a cada uno de ellos se encuentra a nivel bajo.

- **Diodos Led**: la placa también cuenta con 4 diodos LED. Todos ellos se activan si la señal asignada a ellos se encuentra a nivel bajo.
- **Interruptores**: dentro de la placa se encuentran también 8 interruptores. Estos pueden ser activados a nivel bajo o alto.
- **Pulsadores**: la placa también dispone de dos pulsadores. Estos generan una señal a nivel bajo en el momento que son activados.
- **Reloj**: la placa Spartan-IIE dispone de un oscilador con una frecuencia fija de 100Mhz.

Una vez conocido el entorno del cuál se dispone para realizar la verificación del núcleo, se van a explicar los recursos empleados para poder ajustar su diseño a la placa de desarrollo de manera que se puedan emplear los interruptores y pulsadores como entradas y los diodos LED junto con el display para ver sus salidas.

Por otro lado, al igual que ocurría en la simulación, para poder probar el núcleo también es necesario añadirle una memoria con la que pueda interactuar, de manera que, además de que lea de ella para que le suministre el programa a ejecutar, debe también poder almacenar en ella datos si alguna de las instrucciones del programa se lo indican.

En este caso se ha utilizado un componente de memoria suministrado por Xilinx®. Este componente es el elemento RAMB4_S8_S8 y consiste en una memoria de doble puerto. Con cada puerto se accede a las mismas posiciones de memoria por separado de manera que así se permite que el núcleo puede comunicarse con ella mediante uno de sus puertos (puerto A), mientras que por el otro (puerto B), se pueden leer las posiciones de la memoria que se quieran y mostrarlas por medio del display y los diodos LED de la placa para de esta forma comprobar si estas guardan el dato correcto.

Este componente tiene una dimensión fija de manera que si los datos son de 8 bits como es el caso, sólo dispone de 512 posiciones de memoria. Para direccionar 512 posiciones sólo son necesarios 9 bits mientras que el núcleo ha sido diseñado para interactuar con memorias de 8192 posiciones, es decir, que la dirección que suministra tiene una longitud de 13 bits. Por ello y para adecuar otras señales para que el núcleo lea y escriba en el puerto A, además de las que se utilizan para leer desde el puerto B y así poder mostrar lo que el núcleo ha almacenado en la memoria, ha sido necesario ajustar

las señales de nuestro diseño para que se adapten al modo de funcionamiento de este componente.

A continuación se va a explicar la adaptación de todo el diseño para realizar la verificación del mismo:

1. Se ha instanciado en un componente de jerarquía mayor la memoria de doble puerto RAMB4_S8_S8 junto con el componente unidad.vhd que es el que contiene el diseño completo del núcleo. A esta entidad de jerarquía superior es a la que se le ha dado el nombre de nucleo.vhd aunque además de este contiene el componente que actúa como memoria. La implementación de este componente se puede ver en el fichero **nucleo.vhd** en el Capítulo 9 Anexo 9.2.
2. Dentro del fichero nucleo.vhd también se ha instanciado un contador. Este se encarga de reducir la frecuencia del reloj ya que el diseño, como se comentará más adelante, no puede funcionar a tanta frecuencia como la que posee el reloj de la placa. Además de esto, en la verificación hay que reducir mucho esta frecuencia para poder apreciar los valores que va tomando cada señal en los diferentes instantes de tiempo.
3. Junto con todo lo anterior en el fichero nucleo.vhd también se diseña la lógica necesaria para adaptar todas las señales. Se niegan las salidas que van conectadas tanto al display como a los diodos LED ya que estos se activan a nivel bajo.
4. También hay que adaptar las conexiones entre el núcleo y la memoria. Para ello ha sido necesario crear las señales correspondientes para poder leer desde su segundo puerto, negar la entrada de $r/\overline{w}$ para que coincida con la del componente y asignar las correspondientes entradas y salidas del núcleo a las entradas y salidas del primero de los puertos o puerto A.

Al final de todo se tiene un componente que es el que va a ser programado en la FPGA y cuyas entradas y salidas correspondientes se le asignan a los siguientes elementos de la placa:

- **Reloj**: se asigna a la entrada **clk** que a su vez es la entrada del contador que reduce la frecuencia del mismo. La salida de este contador es ya la entrada

del reloj del resto de componentes del diseño como son el núcleo y la memoria.

- **Pulsador R1**: se asigna a la entrada **reset** y actúa de *reset* general tanto del núcleo como del contador.
- **Pulsador R2**: se asigna a la entrada **en_b**. Esta entrada se encarga de habilitar el segundo puerto de la memoria o puerto B para poder leer de él.
- **Interruptor 1**: entrada de la interrupción externa **IRQ**.
- **Interruptor 2**: entrada de la interrupción del temporizador **interrup_TIM**.
- **Interruptor 3**: entrada de la interrupción del puerto serie asíncrono SCI **interrup_SCI**.
- **Interruptor 4**: entrada de la interrupción del puerto serie síncrono SPI **interrup_SPI**.
- **Interruptor 5**: entrada de la interrupción del convertidor analógico-digital **interrup_ADC**.
- **Interruptor 6**: entrada de la interrupción de generador de PWM **interrup_PWM**.
- **Interruptor 7 y 8**: entradas a la señal **eleccion**. Estas dos entradas son las señales de control de un multiplexor de manera que por las mismas salidas se puedan ver diferentes señales.
- **Display y diodo LED 1**: a cada uno de los segmentos del display se le asigna uno de los bits de la señal de salida **salida_led**. Como la señal de salida es de 8 bits y el display sólo dispone de 7 segmentos el octavo bit se le asigna a uno de los diodos LED. Mediante estas salidas se puede ver el valor de diferentes señales según se haya elegido una de ellas mediante los interruptores que van a parar a la señal “eleccion”. Estas señales son las siguientes: la dirección de memoria que se esté leyendo por el puerto B, o el valor de los bits del registro de estado, o la dirección que contiene el registro MAR o el valor del registro de instrucción. Con ellas se puede ir viendo la ejecución de cada una de las instrucciones y los resultados que se van obteniendo.
- **Diodos LED 2 y 3**: estos diodos han sido asignados a las señales **sal_ena** y **sal_rwa** respectivamente. De esta forma se puede comprobar cuando la memoria es habilitada y cuando se realiza una lectura o una escritura.

- **Diodo LED 4**: este diodo ha sido asignado a la salida **clk_perifericos** y muestra el reloj asignado a todos los demás periféricos que puede llegar a tener el microcontrolador. Si se ejecuta la instrucción de STOP, se puede ver como esta señal se para debido a que el núcleo, durante la ejecución de esta instrucción, para el oscilador.

Una vez que se tiene el componente que une el núcleo con la memoria y con todos los elementos necesarios y se han asignado los pines de los diferentes elementos de la placa a las entradas y salidas de este componente, mediante la herramienta de *software* y como se ha comentado anteriormente, se efectúa todo el flujo de diseño y se genera el fichero con el que se programa la FPGA. Cuando se ha realizado con éxito la programación de esta, el núcleo del microcontrolador empieza a funcionar ejecutando todas las instrucciones que contiene el componente de memoria. Activando o no los diferentes interrupciones por medio de los interruptores así como el pulsador de *reset* de la placa, se puede comprobar como el núcleo actúa ante cada una de las entradas. También a través de los interruptores que controlan a la señal “elección” se pueden ir observando las diferentes salidas tanto en los diodos como en el display a medida que transcurre la ejecución del programa de la memoria.

De esta forma ha sido como se ha comprobado el correcto funcionamiento del diseño ya en un componente real como es la FPGA.

A continución, en la Figura 5.4 se muestra una parte del diseño en VHDL en dónde se instancia la memoria de doble puerto. En ella se pueden ver los códigos de operación y los operandos de las instrucciones que componen un pequeño programaa modo de ejemplo mediante el que se ha realizado la verificación de la instrucción de bajo consumo STOP. Tanto los códigos como los operandos de las instrucciones que forman el programa se asignan a las correspondientes posiciones de memoria.

También en la figura se muestra la asignación de señales de entrada y salida de este componente a los dos puertos: al puerto A para que interactúe con el núcleo, mientras que al puerto B, las señales para que se puedan leer las diferentes posiciones de memoria como se ha explicado anteriormente.

El programa que almacena el componente de la figura ejecuta las siguientes instrucciones:

PROGRAMA PRINCIPAL

- NOP
- LDX #$96 :se almacena un dato en el registro de indexado.
- LDA #$30 :se almacena un dato en el acumulador.
- STOP :instrucción de STOP. Se habilitan las interrupciones y se para el oscilador. El núcleo se queda esperando a que tenga lugar alguna de las interrupciones o un *reset*.
- NOP
- NOP
- STA $2F :se almacena el dato del acumulador en la dirección $2F de la memoria. Si ahora mediante el puerto B se lee esta dirección se observa si después de la instrucción RTI se han recuperado los datos de los registros que tenián cuando tuvo lugar una interrupción.
- STX $2F :se realiza igual para el registro de indexado. El valor del registro de estado se puede ver en el display y los diodos LED mediante los interruptores de la placa. El valor del contador de programa es correcto si se continúa la ejecución del programa por la instrucción que seguía al STOP, es decir, por la instrucción NOP.

RUTINA DE INTERRUPCIÓN

- LDA #$1X :se almacena un dato en el acumulador. Dependiendo de la interrupción varía el valor de los cuatro bits menos significativos. De esta forma se comprueba si se ha iniciado la rutina de interrupción que corresponde a la interrupción que ha tenido lugar y si es correcta la prioridad en caso de que se activen varias a la vez.
- STA $2F :se almacena el dato del acumulador en la dirección $2F de la memoria. Si ahora mediante el puerto B se lee esta dirección se observa si el núcleo ha ejecutado la instrucción LDA correctamente.
- RTI :Retorno de interrupción. Los registros recuperan los datos que tenían antes de que tuviera lugar la interrupción y se contínua con la instrucción siguiente al NOP.

```
RAMB4_S8_S8_INSTANCE_NAME : RAMB4_S8_S8
generic map (
  --         1F        1A        15          0F        0A        05        00
INIT_00 => X"9191919191919191919191919191919191919D2FBF2FB79D9D8F30A696AE9D",--00X y 01X
INIT_01 => X"9191919191919191919191919191919191919191919191919191919191919191",--02X y 03X
INIT_02 => X"9191919191919191919191919191818191919191919191919191802FB714A6",--04X y 05X
INIT_03 => X"9191919191919191919191919191919191919191919191919191802FB716A6",--06X y 07X
INIT_04 => X"9191919191919191919191919191919191919191919191919191802FB718A6",--08X y 09X
INIT_05 => X"9191919191919191919191919191919191919191919191919191802FB71AA6",--0AX y 0BX
INIT_06 => X"9191919191919191919191919191919191919191919191919191802FB71CA6",--0CX y 0DX
INIT_07 => X"9191919191919191919191919191919191919191919191919191802FB71EA6",--0EX y 0FX
INIT_08 => X"9191919191919191919191919191919191919191919191919191919191919181",--10X y 11X
INIT_09 => X"3091919191919191919191919191818191919191919191919191919191915691",--12X y 13X
INIT_0A => X"9191919191919191919191919191919191919191919191919191919191919191",--14X y 15X
INIT_0B => X"9191919191919191919191919191919191919191919191919191919191919191",--16X y 17X
INIT_0C => X"9191919191919191919191919191919191919191919191919191919191919191",--18X y 19X
INIT_0D => X"9191919191919191919191919191919191919191919191919191919191919191",--1AX y 1BX
INIT_0E => X"9191919191919191919191919191919191919191919191919191919191919191",--1CX y 1DX
INIT_0F => X"0000200040006000800A000C000E000919191919191919191919191919191919")--1EX y 1FX

port map (DOA => aux_mbr_ent,
          DOB => aux_salidas_led,
          ADDRA => aux_mar_sal(8 DOWNTO 0),
          ADDRB => dir_pc,
          CLKA => aux_clock_10,
          CLKB => aux_clock_10,
          DIA => aux_mbr_sal,
          DIB => dato_pb,
          ENA => aux_oe_memo,
          ENB => en_b,
          RSTA => saux_read_write ,
          RSTB => lectura,
          WEA => saux_read_write,
          WEB => lectura);
```

Figura 5.4 Instanciación de la memoria de doble puerto.

5.3. RESULTADOS DE LA SÍNTESIS E IMPLEMENTACIÓN

Para poder programar la FPGA con el circuito diseñado, es necesario antes realizar la síntesis e implementación del mismo. En la síntesis la herramienta interpreta el código VHDL del programa utilizado en el diseño e identifica los diferentes elementos. En el siguiente paso, la implementación es donde la herramienta traduce todos ellos a elementos de la FPGA, teniendo en cuenta las restricciones impuestas en la optimización.

Antes de pasar a comentar los resultados obtenidos, decir que la síntesis y optimización del diseño se ha realizado de dos formas: una considerando sólo el diseño del núcleo y la otra teniendo en cuenta el componente de memoria y el contador utilizados para la verificación del mismo. Para ambas, las restricciones impuestas son mínima área y máxima velocidad. Los resultados obtenidos se muestran en la Tabla 5.1.

Tabla 5.1 Resultados de área y velocidad después de la implementación del diseño.

RESTRICCIONES	DISEÑO	BIESTABLES	LUTs	PINES E/S	PUERTAS EQUIVALENTES	PERIODO MÍNIMO	FRECUENCIA MÁXIMA
ÁREA	NÚCLEO	111	1111	53	8.626	62,075ns	16,110Mhz
	NÚCLEO + MEMORIA + CONTADOR	162	1.156	21	25.514	60,225ns	16,604MHz
VELOCIDAD	NÚCLEO	161	1.369	53	10.482	44,373ns	22,536MHz
	NÚCLEO + MEMORIA + CONTADOR	209	1.416	21	27.462	51,600ns	19,380MHz

El diseño mejor sería aquel que para la menor área proporciona una mayor velocidad, pero si se analizan los resultados obtenidos, se puede ver como para las restricciones de área se sacrifica la velocidad y viceversa. De esta forma es el diseñador el que, dependiendo de la función para la que se va a destinar el circuito y del presupuesto del que disponga, decide si se prefiere el diseño con un área más reducida o el que alcanza una mayor velocidad.

De los resultados obtenidos se puede ver cómo en el caso del diseño del núcleo exclusivamente si se optimiza en área la ocupación de biestables es del 1% y LUTs es del 18% mientras que si la optimización se hace con la restricción de la velocidad la ocupación de biestables es del 2% y de LUTs del 22%. Si se analiza el núcleo junto con la memoria y el contador ocurre algo parecido.

Es por ello que en este caso es recomendable realizar la optimización en velocidad ya que se aprecia una mayor diferencia entre las velocidades para ambas restricciones tanto si se comtempla el núcleo sólo o con los demás elementos, mientras que, si se estudian los datos de ocupación de la FPGA, la diferencia entre optimizar para área o para velocidad produce una variación que no resulta ser mayor del 1% en biestables y del 4% en las LUTs utilizadas.

CAPÍTULO 6
CONCLUSIONES

6. CONCLUSIONES

El objetivo de este proyecto era realizar el diseño e implementación del núcleo del microcontrolador 68HC05 fabricado por Motorola para una FPGA. Este objetivo se ha llevado a cabo realizando un diseño en lenguaje VHDL con las siguientes características:

- El diseño es capaz de realizar todo el juego de instrucciones que realiza el núcleo 68HC05 fabricado por Motorola. Este objetivo se ha llevado a cabo hasta comprobar realmente en una FPGA el correcto funcionamiento de todas las intrucciones del núcleo así como su modo de proceder ante las diferentes interrupciones, el *reset* del microcontrolador, y los modos de bajo consumo.
- Es un diseño sintetizable y por lo tanto es posible su implemetación en un dispositivo programable. En este caso se ha implementado en una FPGA como ya se ha comentado anteriormente. Con ello también se ha podido comprobar que se consigue una velocidad mucho mayor que la del dispositivo fabricado por Motorola.
- Es portable a las diferentes tecnologías por estar descrito en lenguaje VHDL a pesar de que se ha realizado para la tecnología de Xilinx.
- Permite la adhesión de otros periféricos en futuras aplicaciones. En este caso para su verificación se le ha añadido la memoria con el programa a ejecutar pero se ha podido comprobar como responde correctamente a las entradas de interrupción que le llegan simulando así las interrupciones de periféricos que se pudieran agregar.

CAPÍTULO 7

LÍNEAS FUTURAS

7. LÍNEAS FUTURAS

Una vez concluido el diseño del núcleo del microcontrolador 68HC05, es posible desarrollar determinados trabajos futuros a partir del mismo bien para mejorarlo, o bien para ampliarlo. Algunos de estos trabajos podrían ser los siguientes:

- Diseño e implementación de los periféricos que acompañan al núcleo. Este trabajo sería el complementario al diseño del núcleo y puede ir ya encaminado hacía una determinada aplicación. Es ese caso no sería necesario implementar todos los periféricos de que dispone el microcontrolador fabricado por Motorola sino sólo los que fueran precisos para atender las necesidades de la aplicación.
- Ajuste de los ciclos de reloj que tardan en ejecutarse las diferentes instrucciones a los ciclos que indica Motorola en su juego de instrucciones. Esta tarea no ha sido desarrollada en el presente proyecto pero si se ha dejado indicado los ciclos de reloj que tarda cada instrucción con vistas a un futuro trabajo en esta línea.
- Diseñar e implementar el núcleo de otras versiones más avanzadas de este microcontrolador como pueden ser el 68HC08 o el 68HC11. La versión 68HC08 tiene muchas de sus instrucciones idénticas a las del 68HC05 mientras que el 68HC11 posee los mismos modos de direccionamiento con alguna pequeña variación en el direccionamiento indexado.

  Las diferencias más amplias se encuentran en cuanto a un mayor número y tamaño de los registros que posee la CPU, mayor número también de instrucciones disponibles para ambos y en el caso del 68HC08 muchos más modos de direccionamiento.

CAPÍTULO 8

BIBLIOGRAFÍA

8. BIBLIOGRAFÍA

MICROCONTROLADOR 68HC05

[1] Chistrian Tavernier. “Microcontroladores de 4 y 8 bits”. Ed.Paraninfo,1997.

[2] Motorola. “M68HC05 Aplications guide”. Revisión 4. Marzo, 2002.

[3] Página web de Motorola, sección microcontroladores: http://e-www.motorola.com/webapp/sps/site/homepage.jsp?nodeId=03t3ZG

LENGUAJE VHDL

[4] “IEEE Standard VHDL Language Reference Manual”. IEEE Sid. 1076-1987.

[5] Douglas E. Ott, Thomas J. Wilderotter. “A designer´s guide to VHDL synthesis”. Boston. Kluwer Academic, 1997.

[6] Página web: http://www.vhdl-online.de

FPGA-PLACA DE DESARROLLO

[7] Página web de Xilinx, sección de productos y servicios, Spartan-IIE FPGAs: http://www.xilinx.com/xlnx/xil_prodcat_landingpage.jsp?title=Spartan-IIE

[8] Página web de Xilinx, sección soportes, apartado de documentación: http://www.xilinx.com/support/library.htm

CAPÍTULO 9
ANEXOS

9. ANEXOS

Dentro de los anexos se encuentra el código fuente en lenguaje VHDL de todos los módulos que se han realizado en el diseño del núcleo del microcontrolador 68HC05 junto con el del banco de pruebas utilizado en la simulación del mismo así como el orden de compilación que se debe seguir en el diseño.

9.1. ANEXO I: LISTADO DE FICHEROS EN ORDEN DE COMPILACIÓN

En el lenguaje VHDL es necesario seguir un orden para compilar los ficheros que componen un diseño de manera que antes de compilar una entidad en la que se instancia un componente de menor jerarquía este debe de haber sido compilado anteriormente. Es por ello que a continuación se da un listado de orden de compilación que se debe seguir para el diseño de este proyecto.

El orden de la compilación de los ficheros debe ser:

1. acumulador.vhd
2. x.vhd
3. ccr.vhd
4. sum_total.vhd
5. sum_total_13bits.vhd
6. sum_res_pc.vhd
7. pc.vhd
8. sum_total_8bits.vhd
9. sp.vhd
10. mar.vhd
11. mbr.vhd
12. ri.vhd
13. reg_aux.vhd
14. reg_sal_alu.vhd
15. reg_dir.vhd.
16. sum_res.vhd
17. alu_sum_res.vhd
18. dec_instr.vhd
19. unicon.vhd
20. contador.vhd
21. nucleo.vhd
22. tb_nucleo.vhd

9.2. ANEXO II: CÓDIGO FUENTE

--Fichero **acumulador.vhd**--

```
--ACUMULADOR.
--REGISTRO DE 8 BITS DE LA CPU.
--Está formado por 8 biestables implementado en un solo proceso
--El registro se carga con el valor de la entrada cuando el flanco de reloj es de subida y la señal de carga
--está activada.
--El valor del registro es asignado a la señal de salida del registro en todo momento.


LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY acumulador IS
  PORT(  clock              :IN     STD_LOGIC;
         reset              :IN     STD_LOGIC;
         dat_ent_a          :IN     STD_LOGIC_VECTOR(7 DOWNTO 0);
         carga_a            :IN     STD_LOGIC;
         dat_sal_a          :OUT    STD_LOGIC_VECTOR(7 DOWNTO 0));
END acumulador;

ARCHITECTURE acum OF acumulador IS
 SIGNAL aux_dat_sal_a       :STD_LOGIC_VECTOR(7 DOWNTO 0);
BEGIN
         PROCESS(reset,clock)
         BEGIN
                  IF reset='0' THEN
                           aux_dat_sal_a <= (OTHERS => '0');
                  ELSIF clock'EVENT AND clock='1' THEN
                           IF carga_a='1' THEN
                                    aux_dat_sal_a <= dat_ent_a;
                           END IF;
                  END IF;
         END PROCESS;

         dat_sal_a <= aux_dat_sal_a;

END acum;
```

--Fichero **x.vhd**--

```
--REGISTRO DE INDEXADO.
--REGISTRO DE 8 BITS DE LA CPU.
--Está formado por 8 biestables implementado en un sólo proceso.
--El registro se carga con el valor de la entrada cuando el flanco de reloj es de subida y la señal de carga
--está activada.
--El valor del registro es asignado a la señal de salida del registro en todo momento.


LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY x IS
 PORT(   clock              :IN     STD_LOGIC;
         reset              :IN     STD_LOGIC;
         dat_ent_x          :IN     STD_LOGIC_VECTOR(7 DOWNTO 0);
         carga_x            :IN     STD_LOGIC;
         dat_sal_x          :OUT    STD_LOGIC_VECTOR(7 DOWNTO 0));
END x;

ARCHITECTURE reg_index OF x IS
  SIGNAL aux_dat_sal_x      :STD_LOGIC_VECTOR(7 DOWNTO 0);
BEGIN
         PROCESS(clock,reset)
         BEGIN
                  IF reset='0' THEN
                           aux_dat_sal_x  <=  (OTHERS => '0');
                  ELSIF clock'EVENT AND clock='1' THEN
                           IF carga_x='1' THEN
                                    aux_dat_sal_x  <=  dat_ent_x;
                           END IF;
                  END IF;
         END PROCESS;

         dat_sal_x <= aux_dat_sal_x;

END reg_index;
```

--Fichero **ccr.vhd**--

```
--REGISTRO DE ESTADO.
--REGISTRO DE 5 BITS DE LA CPU.
--Este registro está implementado en un solo proceso. Se componen de 5 biestables síncronos.
--Cada uno de ellos puede cargarse con el valor de su entrada independientemente de los demás
--si su señal de carga está activada y el flanco de la señal de reloj es de subida.
--Si se produce un reset este registro toma el valor “01000” de manera que quedan deshabilitadas
--todas las interrupciones enmascarables.

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY ccr IS
  PORT(   clock              :IN     STD_LOGIC;
          reset              :IN     STD_LOGIC;
          carga_carryout     :IN     STD_LOGIC;
          carga_zero         :IN     STD_LOGIC;
          carga_negativo     :IN     STD_LOGIC;
          carga_carryint     :IN     STD_LOGIC;
          carga_interru      :IN     STD_LOGIC;
          carryout           :IN     STD_LOGIC;
          zero               :IN     STD_LOGIC;
          negativo           :IN     STD_LOGIC;
          interrupcion       :IN     STD_LOGIC;
          carryint           :IN     STD_LOGIC;
          dat_sal_ccr        :OUT    STD_LOGIC_VECTOR(4 DOWNTO 0));
END ccr;

ARCHITECTURE status OF ccr IS
BEGIN
          PROCESS(clock,reset)
          BEGIN
                    IF reset='0' THEN
                              dat_sal_ccr  <= "01000";
                    ELSIF clock'EVENT AND clock='1' THEN
                              IF carga_carryout='1' THEN
                                        dat_sal_ccr(0) <= carryout;
                              END IF;
                              IF carga_zero='1' THEN
                                        dat_sal_ccr(1) <= zero;
                              END IF;
                              IF carga_negativo='1' THEN
                                        dat_sal_ccr(2) <= negativo;
                              END IF;
                              IF carga_interru = '1' THEN
                                        dat_sal_ccr(3) <= interrupcion;
                              END IF;
                              IF carga_carryint = '1' THEN
                                        dat_sal_ccr(4) <= carryint;
                              END IF;
                    END IF;
          END PROCESS;

END status;
```

--Fichero **sum_total.vhd**--

```
--SUMADOR TOTAL DE 1 BIT.
--Este sumador es la entidad de menor jerarquía a partir de la cuál se van a generar los sumadores
--necesarios dentro del diseño.
--Consiste en la realización de forma concurrente de las dos puertas lógicas que componen un
--sumador total.


LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY sum_total IS
  PORT(   a          :IN     STD_LOGIC;
          b          :IN     STD_LOGIC;
          cin        :IN     STD_LOGIC;
          s          :OUT    STD_LOGIC;
          cout       :OUT    STD_LOGIC);
END sum_total;

ARCHITECTURE suma OF sum_total IS
BEGIN
          s          <= (a XOR b XOR cin) ;
          cout       <= ((a AND b) OR ((a XOR b) AND cin));
END suma;
```

--Fichero **sum_total_13_bits.vhd**--

```
--SUMADOR TOTAL DE 13 BITS.
--Este sumador forma parte del sumador que contiene el contador de programa.
--Es por eso que el tamaño de los datos que debe sumar sea de 13 bits.
--La manera de implementarlo ha sido instanciando el sumador de 1 bit 13 veces en una entidad de
--mayor nivel como es ésta y haciendo las conexiones oportunas entre ellos.

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY sum_total_13bits IS
 PORT(  a       :IN     STD_LOGIC_VECTOR(12 DOWNTO 0);
        b       :IN     STD_LOGIC_VECTOR(12 DOWNTO 0);
        cin     :IN     STD_LOGIC;
        s       :OUT    STD_LOGIC_VECTOR(12 DOWNTO 0);
        cout    :OUT    STD_LOGIC);
END sum_total_13bits;

ARCHITECTURE suma_13 OF sum_total_13bits IS

        --Se declara el componente que se va a utilizar en el diseño.
        COMPONENT sum_total
          PORT(a        :IN     STD_LOGIC;
               b        :IN     STD_LOGIC;
               cin      :IN     STD_LOGIC;
               s        :OUT     STD_LOGIC;
               cout     :OUT     STD_LOGIC);
        END COMPONENT;
        SIGNAL c        :STD_LOGIC_VECTOR(11 DOWNTO 0);

BEGIN
        --Se instancia el componente uniendo sus entradas y salidas con las señales que le
        --correspondan.
        sum0: sum_total PORT MAP (  a       => a(0),
                                    b       => b(0),
                                    cin     => cin,
                                    s       => s(0),
                                    cout    => c(0));
        sum1: sum_total PORT MAP (  a       => a(1),
                                    b       => b(1),
                                    cin     => c(0),
                                    s       => s(1),
                                    cout    => c(1));
        sum2: sum_total PORT MAP (  a       => a(2),
                                    b       => b(2),
                                    cin     => c(1),
                                    s       => s(2),
                                    cout    => c(2));
        sum3: sum_total PORT MAP (  a       => a(3),
                                    b       => b(3),
                                    cin     => c(2),
                                    s       => s(3),
                                    cout    => c(3));
        sum4: sum_total PORT MAP (  a       => a(4),
                                    b       => b(4),
                                    cin     => c(3),
                                    s       => s(4),
                                    cout    => c(4));
        sum5: sum_total PORT MAP (  a       => a(5),
                                    b       => b(5),
                                    cin     => c(4),
                                    s       => s(5),
                                    cout    => c(5));
        sum6: sum_total PORT MAP (  a       => a(6),
                                    b       => b(6),
                                    cin     => c(5),
                                    s       => s(6),
                                    cout    => c(6));
        sum7: sum_total PORT MAP (  a       => a(7),
                                    b       => b(7),
                                    cin     => c(6),
                                    s       => s(7),
                                    cout    => c(7));
        sum8: sum_total PORT MAP (  a       => a(8),
                                    b       => b(8),
                                    cin     => c(7),
                                    s       => s(8),
                                    cout    => c(8));
        sum9: sum_total PORT MAP (  a       => a(9),
                                    b       => b(9),
                                    cin     => c(8),
                                    s       => s(9),
                                    cout    => c(9));
        sum10:sum_total PORT MAP (  a       => a(10),
                                    b       => b(10),
                                    cin     => c(9),
                                    s       => s(10),
                                    cout    => c(10));
```

```
        sum11:sum_total PORT MAP (  a       =>  a(11),
                                    b       =>  b(11),
                                    cin     =>  c(10),
                                    s       =>  s(11),
                                    cout    =>  c(11));
         sum12:sum_total PORT MAP ( a       =>  a(12),
                                    b       =>  b(12),
                                    cin     =>  c(11),
                                    s       =>  s(12),
                                    cout    =>  cout);

END suma_13;
```

--Fichero **sum_res_pc.vhd**--

```
--SUMADOR-RESTADOR DE 13 BITS DEL CONTADOR DE PROGRAMA.
--Una vez que se tiene el sumador de 13 bits es necesario que también pueda restar valores.
--Esto se debe a que el contador de programa tiene que poder sumar y restar valores si el
--direccionamiento es relativo.
--En el diseño del sumador restador de 13 bits se instancia el sumador de 13 bits en una entidad de
--mayor jerarquía y se dan valores a sus entradas dependiendo si se quiere una suma o una resta.
--Esto se realiza dentro de un proceso asignando valores a las señales auxiliares que van a parar a las
--entradas del sumador.

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY sum_res_pc IS
 PORT(  a       :IN     STD_LOGIC_VECTOR(12 DOWNTO 0);
        b       :IN     STD_LOGIC_VECTOR(12 DOWNTO 0);
        s_r     :IN     STD_LOGIC;
        cin     :IN     STD_LOGIC;
        s       :OUT    STD_LOGIC_VECTOR(12 DOWNTO 0);
        cout    :OUT    STD_LOGIC);
END sum_res_pc;

ARCHITECTURE suma_resta_pc OF sum_res_pc IS
        --Se declara el componente que se va a utilizar en el diseño.
        COMPONENT sum_total_13bits
          PORT(  a      :IN     STD_LOGIC_VECTOR(12 DOWNTO 0);
                 b      :IN     STD_LOGIC_VECTOR(12 DOWNTO 0);
                 cin    :IN     STD_LOGIC;
                 s      :OUT    STD_LOGIC_VECTOR(12 DOWNTO 0);
                 cout   :OUT    STD_LOGIC);
        END COMPONENT;

        --Señales auxiliaries que se utilizan dentro del proceso.
        SIGNAL aux_a    :STD_LOGIC_VECTOR(12 DOWNTO 0);
        SIGNAL aux_b    :STD_LOGIC_VECTOR(12 DOWNTO 0);
        SIGNAL aux_c    :STD_LOGIC;
BEGIN
        PROCESS(a,b,s_r,cin,aux_a,aux_b,aux_c)
        BEGIN
        --Se asignan valores a las señales auxiliaries dependiendo si se va a realizar una suma o una
        --resta. Esto lo indica la entrada  "s_r" de la entidad.
                IF s_r='0' THEN
                        aux_b  <=  b;
                ELSE
                        aux_b(7  DOWNTO 0)          <= b(7 DOWNTO 0);
                        aux_b(12 DOWNTO 8)          <= (OTHERS => '1');
                END IF;
                aux_c   <=  cin;
                aux_a   <= a;
        END PROCESS;

        --Instanciación del sumador de 13 bits.
        su_re_pc: sum_total_13bits PORT MAP (   a       =>  aux_a,
                                                b       =>  aux_b,
                                                cin     =>  aux_c,
                                                s       =>  s,
                                                cout    =>  cout);
END suma_resta_pc;
```

--Fichero **pc.vhd**--

```
--REGISTRO DE 13 BITS DE LA CPU.
--CONTADOR DE PROGRAMA.
--Lleva la cuenta de la siguiente dirección de memoria donde se va a buscar la siguiente instrucción de
--programa a ejecutar.
--Este registro está compuesto por un registro de 13 biestables síncronos junto con el sumador-restador
--de 13 bits creado anteriormente.
--El registro almacena o bien el valor de la salida del aumador-restador o bien el valor que tiene en su
--entrada cuando las señales "inc_pc" y "carga_pc" se encuentran activadas respectivamente, además
--de que la señal de reloj se encuentre en un flanco de subida.
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

ENTITY pc IS
 PORT(  clock              :IN    STD_LOGIC;
        reset              :IN    STD_LOGIC;
        inc_pc             :IN    STD_LOGIC;
        relativo           :IN    STD_LOGIC;
        carga_pc           :IN    STD_LOGIC;
        dat_ent_pc         :IN    STD_LOGIC_VECTOR(12 DOWNTO 0);
        dat_sal_pc         :OUT   STD_LOGIC_VECTOR(12 DOWNTO 0));
 END pc;

ARCHITECTURE contador OF pc IS

        -- Declaración de las señales auxiliaries.
        SIGNAL cuenta              :STD_LOGIC_VECTOR(12 DOWNTO 0);
        --Señal que lleva la cuenta.
        SIGNAL aux_cuenta          :STD_LOGIC_VECTOR(12 DOWNTO 0);
        --Señal de salida del sumador-restador de 13bits.
        SIGNAL aux_pc              :STD_LOGIC_VECTOR(12 DOWNTO 0);
        --Sumando B del sumador-restador.
        SIGNAL aux_cout            :STD_LOGIC;
        --Acarreo de salida del sumador –restador.
        SIGNAL su_re               :STD_LOGIC;
        --Señal que indica si se tiene que realizar una suma o una resta.
        CONSTANT aux_cin           :STD_LOGIC:='0';
        -- El acarreo de entrada del sumador restador de 13 bits no se utiliza. Por eso su entrada se
        --define como constante.

        --Declaración del  sumador-restador de 13 bits que se va a utilizar.
        COMPONENT sum_res_pc
          PORT(  a          :IN      STD_LOGIC_VECTOR(12 DOWNTO 0);
                 b          :IN      STD_LOGIC_VECTOR(12 DOWNTO 0);
                 s_r        :IN      STD_LOGIC;
                 cin        :IN      STD_LOGIC;
                 s          :OUT     STD_LOGIC_VECTOR(12 DOWNTO 0);
                 cout       :OUT     STD_LOGIC);
        END COMPONENT;
BEGIN
        PROCESS(reset,clock)
        BEGIN
                IF reset='0' THEN
                        cuenta    <=  "1111111111110";
                ELSIF clock'EVENT AND clock='1' THEN
                        IF inc_pc='1' THEN
                                cuenta   <=  aux_cuenta;
                        ELSIF carga_pc='1' THEN
                                cuenta   <=  dat_ent_pc;
                        END IF;
                END IF;
        END PROCESS;

        su_re_pc: sum_res_pc PORT MAP (         a          => cuenta,
                                                b          => aux_pc,
                                                s_r        => su_re,
                                                cin        => aux_cin,
                                                s          => aux_cuenta,
                                                cout       => aux_cout);

        --El operando B del sumador-restador va a ser: o el dato de entrada al contador de programa,
        --o la unidad, dependiendo si es un direccionamiento relativo o no.
        aux_pc     <=          dat_ent_pc          WHEN relativo='1' ELSE "0000000000001";

        --Dependiendo del valor del bit más significativo del operando B se suma o se resta si el
        --direccionamiento es relativo.
        su_re      <=          dat_ent_pc(7)       WHEN relativo='1' ELSE '0';
        dat_sal_pc <= cuenta;

  END contador;
```

--Fichero **sum_total_8bits.vhd**--

```
--SUMADOR TOTAL DE 8 BITS.
--Este sumador forma parte del sumador que contiene tanto el puntero de pila como la ALU.
--Es por eso que el tamaño de los datos que debe sumar sea de 8 bits.
--La manera de implementarlo ha sido instanciando el sumador de 1 bit 8 veces en una entidad de
--mayor nivel como es ésta y haciendo las conexiones oportunas entre ellos.

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY sum_total_8bits IS
 PORT(  a          :IN      STD_LOGIC_VECTOR(7 DOWNTO 0);
        b          :IN      STD_LOGIC_VECTOR(7 DOWNTO 0);
        cin        :IN      STD_LOGIC;
        s          :OUT     STD_LOGIC_VECTOR(7 DOWNTO 0);
        cout       :OUT     STD_LOGIC);
END sum_total_8bits;
```

```
ARCHITECTURE suma_8 OF sum_total_8bits IS

        --Se declara el sumador de 1 bit.
        COMPONENT sum_total
          PORT(   a       :IN     STD_LOGIC;
                  b       :IN     STD_LOGIC;
                  cin     :IN     STD_LOGIC;
                  s       :OUT    STD_LOGIC;
                  cout    :OUT    STD_LOGIC);
        END COMPONENT;

        --Se declaran las señales auxiliares.
        SIGNAL c        :STD_LOGIC_VECTOR(6 DOWNTO 0);
BEGIN
        --Se instancian los 8 sumadores y se unen entre sí.
        sum0: sum_total PORT MAP (  a       => a(0),
                                    b       => b(0),
                                    cin     => cin,
                                    s       => s(0),
                                    cout    => c(0));
         sum1: sum_total PORT MAP ( a       => a(1),
                                    b       => b(1),
                                    cin     => c(0),
                                    s       => s(1),
                                    cout    => c(1));
        sum2: sum_total PORT MAP (  a       => a(2),
                                    b       => b(2),
                                    cin     => c(1),
                                    s       => s(2),
                                    cout    => c(2));
        sum3: sum_total PORT MAP (  a       => a(3),
                                    b       => b(3),
                                    cin     => c(2),
                                    s       => s(3),
                                    cout    => c(3));
        sum4: sum_total PORT MAP (  a       => a(4),
                                    b       => b(4),
                                    cin     => c(3),
                                    s       => s(4),
                                    cout    => c(4));
        sum5: sum_total PORT MAP (  a       => a(5),
                                    b       => b(5),
                                    cin     => c(4),
                                    s       => s(5),
                                    cout    => c(5));
        sum6: sum_total PORT MAP (  a       => a(6),
                                    b       => b(6),
                                    cin     => c(5),
                                    s       => s(6),
                                    cout    => c(6));
         sum7: sum_total PORT MAP ( a       => a(7),
                                    b       => b(7),
                                    cin     => c(6),
                                    s       => s(7),
                                    cout    => cout);

END suma_8;
```

--Fichero **sp.vhd**--

```
--PUNTERO DE PILA.
--REGISTRO DE 8 BITS DE LA CPU.
--Este registro mantiene el valor de la dirección de memoria dónde se va a guardar el siguiente
--elemento que se quiera almacenar en la pila.
--Es un registro de 8 bits compuesto por 8 biestables síncronos junto con un sumador. El sumador se
--utiliza para poder incrementar o decrementar el valor del puntero.
--El registro se actualiza con la salida del sumador cuando la señal de reloj se encuentra en un flanco
--de subida y la señal de carga está activada.
--La forma de implementarlo es mediante dos procesos secuenciales junto con la instanciación del
--sumador de manera concurrente. En uno se diseñan los registros mientras que en el otro se dan
--valores a las entradas y salidas del sumador.


LIBRARY IEEE;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

ENTITY sp IS
  PORT(  clock          :IN     STD_LOGIC;
         reset          :IN     STD_LOGIC;
         carga_sp       :IN     STD_LOGIC;
         push           :IN     STD_LOGIC;
         pull           :IN     STD_LOGIC;
         dat_sal_sp     :OUT    STD_LOGIC_VECTOR(7 DOWNTO 0));
END sp;
```

```
ARCHITECTURE puntero OF sp IS

        --Se declaran las señales auxiliaries.
        SIGNAL  puntero     :STD_LOGIC_VECTOR(7 DOWNTO 0);
        --Señal que indica el valor actual del puntero de pila.
        SIGNAL  puntero_a   :STD_LOGIC_VECTOR(7 DOWNTO 0);
        -- Señal que indica la salida del sumador de 8 bits.
        SIGNAL  aux_a       :STD_LOGIC_VECTOR(7 DOWNTO 0);
        SIGNAL  aux_b       :STD_LOGIC_VECTOR(7 DOWNTO 0);
        --Operandos de entrada al sumador de 8 bits.
        SIGNAL  aux_c       :STD_LOGIC;
        --Acarreo de entrada al sumador de 8 bits.
        SIGNAL  aux_cout            :STD_LOGIC;
        --Acarreo de salida del sumador.

        --Se declara el sumador de 8 bits que se utiliza en el diseño.
        COMPONENT sum_total_8bits
        PORT(   a           :IN     STD_LOGIC_VECTOR(7 DOWNTO 0);
                b           :IN     STD_LOGIC_VECTOR(7 DOWNTO 0);
                cin         :IN     STD_LOGIC;
                s           :OUT    STD_LOGIC_VECTOR(7 DOWNTO 0);
                cout        :OUT    STD_LOGIC);
        END COMPONENT;
BEGIN
        --En este proceso donde se implementa el registro.
        PROCESS(reset,clock)
        BEGIN
                IF reset='0' THEN
                        puntero             <=  (OTHERS =>'1');
                ELSIF clock'EVENT AND clock='1' THEN
                        IF carga_sp='1' THEN
                                puntero(5 DOWNTO 0) <=  puntero_a(5 DOWNTO 0);
                                puntero(7 DOWNTO 6) <=  "11";
                        END IF;
                END IF;
        END PROCESS;

        --En este proceso de dan valores al sumador de manera que si la señal de entrada  “push”
        --está activa hay que decrementar el puntero. Para ello el operando B junto con el acarreo de
        --entrada realizan el complemento a 2 de la unidad.
        --Si por el contrario la señal activa es “pull” hay que incrementar el puntero. El operando B
        --es la unidad mientras que el acarreo de entrada es nulo.
        PROCESS(push,puntero,pull)
        BEGIN
                IF push ='1' THEN
                        aux_a(5 DOWNTO 0) <= puntero(5 DOWNTO 0);
                        aux_a(7 DOWNTO 6) <= "11";
                        aux_b       <= "11111110";
                        aux_c       <= '1';
                ELSIF pull='1' THEN
                        aux_a(5 DOWNTO 0) <= puntero(5 DOWNTO 0);
                        aux_a(7 DOWNTO 6) <= "11";
                        aux_b       <= "00000001";
                        aux_c       <= '0';
                ELSE
                        aux_a       <= (OTHERS =>'0');
                        aux_b       <= (OTHERS =>'0');
                        aux_c       <= '0';
                END IF;
        END PROCESS;

        --Se instancia el sumador de 8 bits.
        sum_t_8: sum_total_8bits PORT MAP (     a       =>  aux_a,
                                                b       =>  aux_b,
                                                cin     =>  aux_c,
                                                s       =>  puntero_a,
                                                cout    =>  aux_cout);

        --La salida del sumador va a ser la señal que guarda el valor del registro en todo momento.
        dat_sal_sp <= puntero;

END puntero;
```

--Fichero **mar.vhd**--

```
--MAR.
--Registro que guarda la dirección a la que va a cceder la memoria.
--Es un registro de 13 bits compuesto por 13 biestables síncronos. La salida del  registro se actualiza
--con el  valor de la entrada cuando se activa la señal de carga y tiene lugar un flanco de subida en la
--señal de reloj.
--La implementación de este registro se realiza en un sólo proceso.

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY mar IS
 PORT(  clock               :IN     STD_LOGIC;
        reset               :IN     STD_LOGIC;
        dat_ent_mar         :IN     STD_LOGIC_VECTOR(12 DOWNTO 0);
        carga_mar           :IN     STD_LOGIC;
```

```
        dat_sal_mar         :OUT  STD_LOGIC_VECTOR(12 DOWNTO 0));
END mar;


ARCHITECTURE addres OF mar IS
BEGIN
        PROCESS(reset,clock)
        BEGIN
                IF reset='0' THEN
                        dat_sal_mar  <=  "1111111111110";
                ELSIF clock'EVENT AND clock='1' THEN
                        IF carga_mar='1' THEN
                                dat_sal_mar  <=  dat_ent_mar;
                        END IF;
                END IF;
        END PROCESS;
END addres;
```

--Fichero **mbr.vhd**--

```
--MBR.
--Registro que guarda el dato que se obtiene de la memoria o que se va a guardar en ella. Esto depende
--de si se trata de una lectura o una escritura.
--Es un registro de 8 bits compuesto por 16 biestables síncronos (8 para el dato que le llega desde la
--memoria y 8 para el dato que envía a la memoria.
--El registro se ha implementado en un solo proceso.
--Dependiendo del valor de la señal que decide el sentido “sentido_b_m” y de si la señal de reloj se
--encuentra en un flanco de subida, se carga una señal u otra con el dato de la entrada contraria, es
--decir, la señal de memoria se carga con la entrada del bus y la señal del bus se carga con la entrada
--que llega de la memoria. De esta forma se intercambian los datos entre el bus y la memoria.
--Cada salida lleva asignado el valor del dato almacenado en su señal intermedia.

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY mbr IS
 PORT(  clock           :IN     STD_LOGIC;
        reset           :IN     STD_LOGIC;
        dat_ent_mbr_b   :IN     STD_LOGIC_VECTOR(7 DOWNTO 0);
        dat_sal_mbr_b   :OUT    STD_LOGIC_VECTOR(7 DOWNTO 0);
        dat_ent_mbr_m   :IN     STD_LOGIC_VECTOR(7 DOWNTO 0);
        dat_sal_mbr_m   :OUT    STD_LOGIC_VECTOR(7 DOWNTO 0);
        sentido_b_m     :IN     STD_LOGIC);
END mbr;

ARCHITECTURE dat OF mbr IS
        SIGNAL aux_mbr_b        :STD_LOGIC_VECTOR(7 DOWNTO 0);
        --Señal que almacena el dato que le llega de la memoria para pasarlo al bus de datos.
        SIGNAL aux_mbr_m        :STD_LOGIC_VECTOR(7 DOWNTO 0);
        --Señal que guarda el dato del bus de datos y se lo pasa a la memoria.
BEGIN
        PROCESS(reset,clock)
        BEGIN
                IF reset='0' THEN
                        aux_mbr_b       <=  (OTHERS => '0');
                        aux_mbr_m       <=  (OTHERS => '0');
                ELSIF clock'EVENT AND clock='1' THEN
                        --Dependiendo del sentido los datos llevan una dirección.
                        IF sentido_b_m='1' THEN
                                --El dato del bus se le pasa a la memoria.
                                aux_mbr_m       <=  dat_ent_mbr_b;
                        ELSE
                                --El dato de la memoria se pasa al bus de datos.
                                aux_mbr_b       <=  dat_ent_mbr_m;
                        END IF;
                END IF;
        END PROCESS;

        --Se asignan las señales auxiliares a las salidas.
        dat_sal_mbr_b   <= aux_mbr_b;
        dat_sal_mbr_m   <= aux_mbr_m;

END dat;
```

--Fichero **ri.vhd**--

```
--REGISTRO DE INSTRUCCIÓN.
--Registro que guarda la instrucción que se está ejecutando en ese momento.
--Es un registro de 8 bits compuesto por 8 biestables síncronos. Se ha implementado en un sólo proceso
--de manera que en cada flanco de subida de la señal de reloj y si la señal de carga está activada el
--registro almacena el dato que tiene en su entrada.
--La señal de salida del registro muestra el valor del dato almacenado en todo momento.

LIBRARY ieee;
USE ieee.std_logic_1164.all;
```

```
ENTITY ri IS
  PORT(   clock              :IN     STD_LOGIC;
          reset              :IN     STD_LOGIC;
          dat_ent_ri         :IN     STD_LOGIC_VECTOR(7 DOWNTO 0);
          carga_ri           :IN     STD_LOGIC;
          dat_sal_ri         :OUT    STD_LOGIC_VECTOR(7 DOWNTO 0));
END ri;

ARCHITECTURE reg_instr OF ri IS
BEGIN
        PROCESS(clock,reset)
        BEGIN
                IF reset='0' THEN
                        --En un reset este registro toma el valor $90 por defecto. Este es un
                        --código de operación libre que se ha tomado así de manera que el
                        --proceso de reset del microcontrolador se ejecute al igual que el resto
                        --de las instrucciones, es decir, consultando el valor de este registro y
                        --tomando decisiones respecto a él.
                        dat_sal_ri  <=  "10010000";
                ELSIF clock'EVENT AND clock='1' THEN
                        IF carga_ri='1' THEN
                                dat_sal_ri <=  dat_ent_ri;
                        END IF;
                END IF;
        END PROCESS;
END reg_instr;
```

--Fichero **reg_aux.vhd**--

```
--REGISTRO AUXILIAR DE 8 BITS.
--Se encarga de almacenar el byte más significativo en los modos de direccionamiento extendido e
--indexado con offset de 16 bits.
--Se compone de un resgistro de 5 biestables síncronos que se actualizan con la señal de entrada si la
--señal de carga está activa y se produce un flanco de subida en la señal de reloj.

LIBRARY IEEE;
USE ieee.std_logic_1164.all;

ENTITY reg_aux IS
  PORT(   clock              :IN     STD_LOGIC;
          reset              :IN     STD_LOGIC;
          ext_ind16          :IN     STD_LOGIC;
          dat_ent_reg_aux    :IN     STD_LOGIC_VECTOR(4 DOWNTO 0);
          dat_sal_reg_aux    :OUT    STD_LOGIC_VECTOR(4 DOWNTO 0));
END reg_aux;

ARCHITECTURE ar_reg_aux OF reg_aux IS
BEGIN
        PROCESS(clock,reset)
        BEGIN
                IF reset='0' THEN
                        dat_sal_reg_aux   <=  (OTHERS => '0');
                ELSIF clock'EVENT AND clock='1' THEN
                        IF ext_ind16='1' THEN
                                dat_sal_reg_aux  <=  dat_ent_reg_aux(4 DOWNTO 0);
                        END IF;
                END IF;
        END PROCESS;
END ar_reg_aux;
```

--Fichero **reg_sal_alu.vhd**--

```
--REGISTRO AUXILIAR DE 8 BITS + EL ACARREO.
--Este registro se encarga de almacenar los datos a la salida de la ALU.
--Es un registro compuesto por 9 biestables síncronos (8 del dato y uno para bit de el acarreo).
--Con cada flanco positivo de la señal de reloj los registros se cargan con el valor de su entrada.
--La salida del registro muestra este valor en todo momento.

LIBRARY IEEE;
USE ieee.std_logic_1164.all;

ENTITY reg_sal_alu IS
  PORT(   clock              :IN     STD_LOGIC;
          reset              :IN     STD_LOGIC;
          dat_ent_reg        :IN     STD_LOGIC_VECTOR(7 DOWNTO 0);
          dat_ent_carry      :IN     STD_LOGIC;
          dat_sal_carry      :OUT    STD_LOGIC;
          dat_sal_reg        :OUT    STD_LOGIC_VECTOR(7 DOWNTO 0));
END reg_sal_alu;

ARCHITECTURE ar_reg_sal_alu OF reg_sal_alu IS
        SIGNAL aux_dat_reg          :STD_LOGIC_VECTOR(7 DOWNTO 0);
        SIGNAL aux_dat_carry        :STD_LOGIC;
BEGIN
```

```
        PROCESS(clock,reset)
        BEGIN
                IF reset='0' THEN
                        aux_dat_reg         <= (OTHERS=>'0');
                        aux_dat_carry       <= '0';
                ELSIF clock'EVENT AND clock='1' THEN
                        aux_dat_reg         <= dat_ent_reg;
                        aux_dat_carry       <= dat_ent_carry;
                END IF;
        END PROCESS;

        dat_sal_reg         <= aux_dat_reg;
        dat_sal_carry       <= aux_dat_carry;

END ar_reg_sal_alu;
```

--Fichero **reg_dir.vhd**--

```
--REGISTRO AUXILIAR DE 13 BITS.
--Este registro se encarga de almacenar la dirección en las instrucciones que realizan un salto a
--subrutina.
--Es un registro de 13 bits compuesto por 13 biestables síncronos.  En cada flanco de subida de la señal
--de reloj y si la señal de carga está activada, se almacena en el registro el dato de entrada.
--La salida del registro muestra su valor en todo momento.

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

ENTITY reg_dir IS
 PORT(  clock           :IN    STD_LOGIC;
        reset           :IN    STD_LOGIC;
        carga_reg_dir   :IN    STD_LOGIC;
        dat_ent_reg_dir :IN    STD_LOGIC_VECTOR(12 DOWNTO 0);
        dat_sal_reg_dir :OUT STD_LOGIC_VECTOR(12 DOWNTO 0));
 END reg_dir;

ARCHITECTURE ar_reg_dir OF reg_dir IS
   SIGNAL dat_reg_dir_aux :STD_LOGIC_VECTOR(12 DOWNTO 0);
BEGIN
        PROCESS(clock,reset)
        BEGIN
                IF reset='0' THEN
                        dat_reg_dir_aux   <= (OTHERS=>'0');
                ELSIF clock'EVENT AND clock='1' THEN
                        IF carga_reg_dir='1' THEN
                                dat_reg_dir_aux   <= dat_ent_reg_dir;
                        END IF;
                END IF;
        END PROCESS;

        dat_sal_reg_dir <= dat_reg_dir_aux;

END ar_reg_dir;
```

--Fichero **sum_res.vhd**--

```
--SUMADOR-RESTADOR DE 8 BITS DE LA ALU.
--A partir del sumador de 8 bits realizado anteriormente es necesario formar un sumador-restador de 8
--bits para las operaciones de suma y resta de la ALU.
--En el diseño del sumador-restador de 8 bits se instancia el sumador de 8 bits en una entidad de mayor
--jerarquía y se dan valores a sus entradas dependiendo si se quiere una suma o una resta.
--Esto se realiza dentro de un proceso asignando valores a las señales auxiliares que van a parar a las
--entradas del sumador.
--Este sumador-restador tiene que tener en cuenta la operación de resta con acarreo. En este caso deben
--realizar dos restas, por lo que van a ser necesarios dos sumadores.

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY sum_res IS
 PORT(  a       :IN     STD_LOGIC_VECTOR(7 DOWNTO 0);
        b       :IN     STD_LOGIC_VECTOR(7 DOWNTO 0);
        s_r     :IN     STD_LOGIC_VECTOR(2 DOWNTO 0);
        c_y_n   :IN     STD_LOGIC;
        cin     :IN     STD_LOGIC;
        s       :OUT    STD_LOGIC_VECTOR(7 DOWNTO 0);
        cout    :OUT    STD_LOGIC);
END sum_res;

ARCHITECTURE suma_resta OF sum_res IS
        --Se declara el componente que se va a utilizar.
        COMPONENT sum_total_8bits
          PORT(  a      :IN     STD_LOGIC_VECTOR(7 DOWNTO 0);
```

```
        b       :IN     STD_LOGIC_VECTOR(7 DOWNTO 0);
        cin     :IN     STD_LOGIC;
        s       :OUT    STD_LOGIC_VECTOR(7 DOWNTO 0);
        cout    :OUT    STD_LOGIC);
END COMPONENT;

--Se declaran las señales auxiliares.
SIGNAL  aux_a           :STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL  aux_b           :STD_LOGIC_VECTOR(7 DOWNTO 0);
--Señales de entrada a los operandos.
SIGNAL  aux_c           :STD_LOGIC;
--Señal del acarreo de entrada.
SIGNAL  aux_s1          :STD_LOGIC_VECTOR(7 DOWNTO 0);
--Señal de salida del primer sumador.
SIGNAL  aux_cout1       :STD_LOGIC;
--Acarreo de salida del primer sumador.
SIGNAL  aux_s2          :STD_LOGIC_VECTOR(7 DOWNTO 0);
--Señal de salida del segundo sumador.
SIGNAL  aux_cout2       :STD_LOGIC;
--Acarreo de salida del segundo sumador.
SIGNAL  condicion       :STD_LOGIC;
--Señal que en el caso de la resta indica si ésta es con o sin acarreo.
CONSTANT  cte           :STD_LOGIC_VECTOR(7 DOWNTO 0):="11111110";
--Valor constante. Es el inverso de la unidad.
BEGIN

PROCESS(a,b,s_r,c_y_n,cin,aux_a,aux_b,aux_c)
BEGIN
        CASE s_r IS
                WHEN "000"  =>  --Suma
                                aux_a   <=  a;
                                aux_b   <=  b;
                                IF c_y_n ='0' THEN
                                        aux_c   <= '0';
                                ELSE
                                        aux_c   <= cin;
                                END IF;
                WHEN "001"  =>  --Resta
                                aux_a   <= a;
                                aux_b   <= NOT(b);
                                aux_c   <= '1';
                WHEN "010"  =>  --Incremento
                                aux_a   <= a;
                                aux_b   <= "00000001";
                                aux_c   <= '0';
                WHEN "011"  =>  --Decremento
                                aux_a   <= a;
                                aux_b   <= cte;
                                aux_c   <= '1';
                WHEN OTHERS =>  --Nop
                                aux_a   <= (OTHERS => '0');
                                aux_b   <= (OTHERS => '0');
                                aux_c   <= '0';
        END CASE;
END PROCESS;

su_re1: sum_total_8bits PORT MAP (      a       =>  aux_a,
                                        b       =>  aux_b,
                                        cin     =>  aux_c,
                                        s       =>  aux_s1,
                                        cout    =>  aux_cout1);
su_re2: sum_total_8bits PORT MAP (      a       =>  aux_s1,
                                        b       =>  cte,
                                        cin     =>  aux_c,
                                        s       =>  aux_s2,
                                        cout    =>  aux_cout2);
 --Señal que indica si es necesario utilizar el segundo sumador al tener una operación de resta
--con acarreo.
condicion  <= (c_y_n AND cin)  WHEN s_r="001"     ELSE '0';

--Las salidas se toman del primero o del segundo sumador.
s       <= aux_s2       WHEN condicion='1' ELSE aux_s1;
cout    <= aux_cout2    WHEN condicion='1' ELSE aux_cout1;

END suma_resta;
```

--Fichero **alu_sum_res.vhd**--

```
--UNIDAD ARITMÉTICO-LÓGICA ALU.
--Se encarga de realizar todas las operaciones del núcleo y de modificar los bits del registro de estado.
--La ALU es un componente asíncrono y está formada por varios procesos.


LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

ENTITY alu_sum_res IS
 PORT(  carga_alu       :IN     STD_LOGIC;
        a               :IN     STD_LOGIC_VECTOR(7 DOWNTO 0);
        b               :IN     STD_LOGIC_VECTOR(7 DOWNTO 0);
```

```
        carryin         :IN     STD_LOGIC;
        c_y_n           :IN     STD_LOGIC;
        sel_op          :IN     STD_LOGIC_VECTOR(4 DOWNTO 0);
        sel_bit         :IN     STD_LOGIC_VECTOR(2 DOWNTO 0);
        bit_rot         :IN     STD_LOGIC_VECTOR(1 DOWNTO 0);
        oe_alu_b        :IN     STD_LOGIC;
        interrupcion    :IN     STD_LOGIC;
        dat_sal_alu     :OUT    STD_LOGIC_VECTOR(7 DOWNTO 0);
        carryout        :OUT    STD_LOGIC;
        zero            :OUT    STD_LOGIC;
        negativo        :OUT    STD_LOGIC;
        carryint        :OUT    STD_LOGIC;
        interrup        :OUT    STD_LOGIC);
END alu_sum_res;

ARCHITECTURE primera_sum_res OF alu_sum_res IS

        --Se declaran las señales auxiliaries.
        SIGNAL bitsel           :STD_LOGIC_VECTOR(7 DOWNTO 0);
        --Señal que indica el bit sobre el que se va a operar
        SIGNAL auxiliar         :STD_LOGIC_VECTOR(8 DOWNTO 0);
        --Señal que almacena el resultado + acarreo
        SIGNAL bit_aux          :STD_LOGIC;
        --Señal que indica el bit que acompaña a la rotacion.
        SIGNAL sumador          :STD_LOGIC_VECTOR (2 DOWNTO 0);
        --Señal que le indica al sumador si se quiere sumar, restar, incrementar o decrementar.
        SIGNAL aux_b            :STD_LOGIC_VECTOR(7 DOWNTO 0);
        --Señal que almacena el operando B.
        SIGNAL aux_dat_sal_alu  :STD_LOGIC_VECTOR(8 DOWNTO 0);
        --Señal que guarda el dato de salida de la ALU.
        SIGNAL producto         :STD_LOGIC_VECTOR(15 DOWNTO 0);
        --Señal que almacena el resultado de la multiplicación.
        SIGNAL aux_zero         :STD_LOGIC;
        --Señal que almacena el bit Z.
        SIGNAL aux_carryint     :STD_LOGIC;
        --Señal que almacena el acarreo intermedio.
        SIGNAL aux_carryout     :STD_LOGIC;
        --Señal que almacena el acarreo de salida.

        --Se declara el componente que se va a utilizar .
        COMPONENT sum_res
          PORT(  a      :IN     STD_LOGIC_VECTOR(7 DOWNTO 0);
                 b      :IN     STD_LOGIC_VECTOR(7 DOWNTO 0);
                 s_r    :IN     STD_LOGIC_VECTOR(2 DOWNTO 0);
                 c_y_n  :IN     STD_LOGIC;
                 cin    :IN     STD_LOGIC;
                 s      :OUT    STD_LOGIC_VECTOR(7 DOWNTO 0);
                 cout   :OUT    STD_LOGIC);
        END COMPONENT;
BEGIN
        --En este proceso se indica el bit sobre el que se va a operar.
        PROCESS(sel_bit)
        BEGIN
                CASE sel_bit IS
                        WHEN "000"      => bitsel <= "00000001";
                        WHEN "001"      => bitsel <= "00000010";
                        WHEN "010"      => bitsel <= "00000100";
                        WHEN "011"      => bitsel <= "00001000";
                        WHEN "100"      => bitsel <= "00010000";
                        WHEN "101"      => bitsel <= "00100000";
                        WHEN "110"      => bitsel <= "01000000";
                        WHEN OTHERS     => bitsel <= "10000000";
                END CASE;
        END PROCESS;

        --En este proceso se indica el bit que se va a uitlizar en las rotaciones y desplazamientos.
        PROCESS(bit_rot,carryin,a)
        BEGIN
                CASE bit_rot IS
                        WHEN "00"       => bit_aux <= '0';
                        WHEN "01"       => bit_aux <= carryin;
                        WHEN OTHERS     => bit_aux <= a(7);
                END CASE;
        END PROCESS;

        --En este proceso se realizan las diferentes operaciones.
        PROCESS(sel_op,carryin,a,b,bit_aux,bitsel,carga_alu,interrupcion,sel_bit,auxiliar)
        BEGIN
                producto   <= (OTHERS => '0');
                interrup   <=  interrupcion;
                sumador    <= "111";
                IF carga_alu = '1' THEN
                        CASE sel_op IS
                                WHEN "00000"    =>      --suma
                                        aux_dat_sal_alu  <= auxiliar;
                                        sumador          <= "000";
                                WHEN "00001"    =>      --resta
                                        aux_dat_sal_alu  <= auxiliar;
                                        sumador          <= "001";
                                WHEN "00010"    =>      --and
                                        aux_dat_sal_alu(7 downto 0) <= a AND b;
                                        aux_dat_sal_alu(8)          <= '0';
```

```
WHEN "00011"          =>        --not
            aux_dat_sal_alu(7 downto 0) <= NOT(a);
            aux_dat_sal_alu(8)          <= '1';
WHEN "00100"          =>        --xor
            aux_dat_sal_alu(7 downto 0) <= a XOR b;
            aux_dat_sal_alu(8)          <= '0';
WHEN "00101"          =>        --or
            aux_dat_sal_alu(7 downto 0) <= a OR b;
            aux_dat_sal_alu(8)          <= '0';
WHEN "00110"          =>        --rotación_derecha
            aux_dat_sal_alu(7)          <= bit_aux;
            aux_dat_sal_alu(6 downto 0) <= a(7 downto 1);
            aux_dat    _sal_alu(8)      <= a(0);
 WHEN "00111"          =>        --rotación_izquierda
            aux_dat_sal_alu(8 downto 1) <= a;
            aux_dat_sal_alu(0)          <= bit_aux;
WHEN "01000"          =>        --bit_set
            aux_dat_sal_alu(7 downto 0) <= a OR bitsel;
            aux_dat_sal_alu(8)  <= a(conv_integer(sel_bit));
WHEN "01001"          =>        --bit_reset
       aux_dat_sal_alu(7 downto 0)<= a AND (NOT(bitsel));
            aux_dat_sal_alu(8)  <= a(conv_integer(sel_bit));
WHEN "01010"          =>        --transparencia
            aux_dat_sal_alu(7 downto 0) <= a;
            aux_dat_sal_alu(8)          <= '0';
WHEN "01011"          =>        --set
            aux_dat_sal_alu(7 downto 0) <= (others => '0');
            aux_dat_sal_alu(8)          <= '1';
            interrup                    <= '1';
WHEN "01100"          =>        --reset
            aux_dat_sal_alu             <= (others => '0');
            interrup                    <= '0';
WHEN "01101"          =>        --incremento
            aux_dat_sal_alu             <= auxiliar;
            sumador                     <= "010";
WHEN "01110"          =>        --decremento
            aux_dat_sal_alu             <= auxiliar;
            sumador                     <= "011";
WHEN "10000"          =>        --multiplicación
            producto                    <= a * b;
            sumador                     <= "100";
            aux_dat_sal_alu             <= (others => '0');
WHEN "10001"          =>        --multiplicación_2
            producto                    <= a * b;
            sumador                     <= "101";
            aux_dat_sal_alu             <= (others => '0');
                        WHEN OTHERS       =>
                                    aux_dat_sal_alu             <= (others => '0');
                END CASE;
        ELSE
                aux_dat_sal_alu <= (others => '0');
        END IF;
END PROCESS;

--Se instancia el componente sumador-restador.
Su_re0:sum_res PORT MAP (   a       =>  a,
                            b       =>  aux_b,
                            s_r     =>  sumador,
                            c_y_n   =>  c_y_n,
                            cin     =>  carryin,
                            s       =>  auxiliar(7 DOWNTO 0),
                            cout    =>  auxiliar(8));

--Se asignan valores a las salidas que modifican el registro de estado dependiendo de la
--operación.
carryout    <= '0' WHEN (sumador="100" OR sumador="101") ELSE aux_carryout;
carryint    <= '0' WHEN (sumador="100" OR sumador="101") ELSE aux_carryint;
negativo    <= aux_dat_sal_alu(7);
zero        <= NOT(aux_zero);

--En este proceso se dan valores a la señal auxiliar del acarreo de salida dependiendo de la
--operación indicada por el valor de la señal “sumador” ya que en el caso de la resta este
--toma otro valor.
PROCESS(sumador,aux_dat_sal_alu,a,aux_b)
BEGIN
        IF sumador/="001" THEN
                aux_carryout    <= aux_dat_sal_alu(8);
        ELSE
                aux_carryout <= (((NOT(a(7))) AND aux_b(7)) OR (aux_b(7) AND
                                aux_dat_sal_alu(7)) OR (aux_dat_sal_alu(7) AND
                                (NOT(a(7)))));
        END IF;
END PROCESS;

--Se asignan los valores correspondientes a las señales auxiliares que representan los valores
--del bit Z y del acarreo intermedio del registro de estado.
aux_zero <= (aux_dat_sal_alu(7) OR aux_dat_sal_alu(6) OR aux_dat_sal_alu(5) OR
            aux_dat_sal_alu(4) OR aux_dat_sal_alu(3) OR aux_dat_sal_alu(2) OR
            aux_dat_sal_alu(1) OR aux_dat_sal_alu(0));
aux_carryint  <= (((a(3) AND aux_b(3)) OR (aux_b(3) AND (NOT(aux_dat_sal_alu(3))))
                OR (a(3) AND (NOT(aux_dat_sal_alu(3))))));
```

```
	--En este proceso se asignan valores a la salida dependiendo de la operación.
	PROCESS(aux_dat_sal_alu,producto,sumador)
	BEGIN
		IF sumador="100" THEN
			dat_sal_alu		<= producto(7 DOWNTO 0);
		ELSIF sumador="101" THEN
			dat_sal_alu		<= producto(15 DOWNTO 8);
		ELSE
			dat_sal_alu		<= aux_dat_sal_alu(7 DOWNTO 0);
		END IF;
	END PROCESS;



	--Sólo se permite la carga del operando B del sumador-restador en determinados momentos
	--que vienen indicados por la unidad de control mediante la señal “oe_alu_b”.
	aux_b	<= b WHEN oe_alu_b='1' ELSE (OTHERS => '0');

END primera_sum_res;

-------------------------------------------------------------------------------------------------------------------------------


--Fichero dec_instr.vhd--

--DECODIFICADOR DE INSTRUCCIONES DE LA UNIDAD DE CONTROL.
--Está formado por una máquina de estados implementada en dos procesos.
--En cada estado de la máquina se activan determinadas señales de salida dependiendo de los registros
--que haya que cargar en ese estado y de las acciones a realizar en el mismo.
--Sus entradas son la instrucción a ejecutar, los valores de los bits del registro de estado, las
--interrupciones, además claro está del reloj y la señal de reset.

LIBRARY IEEE;
USE ieee.std_logic_1164.all;

ENTITY dec_instr IS
  PORT(	clock		:IN	STD_LOGIC;
	reset		:IN	STD_LOGIC;
	dat_instr	:IN	STD_LOGIC_VECTOR(7 DOWNTO 0);
	ccr		:IN	STD_LOGIC_VECTOR(4 DOWNTO 0);
	IRQ		:IN	STD_LOGIC;
	interrupcion	:IN	STD_LOGIC;
	carga_alu	:OUT	STD_LOGIC;
	carga_mar	:OUT	STD_LOGIC;
	oe_mbr		:OUT	STD_LOGIC;
	sentido_b_m	:OUT	STD_LOGIC;
	oe_alu_b	:OUT	STD_LOGIC;
	carga_ri	:OUT	STD_LOGIC;
	oe_pc		:OUT	STD_LOGIC;
	inc_pc		:OUT	STD_LOGIC;
	carga_pc	:OUT	STD_LOGIC;
	oe_memo		:OUT	STD_LOGIC;
	read_write	:OUT	STD_LOGIC;
	sel_op		:OUT	STD_LOGIC_VECTOR(4 DOWNTO 0);
	sel_bit		:OUT	STD_LOGIC_VECTOR(2 DOWNTO 0);
	bit_rot		:OUT	STD_LOGIC_VECTOR(1 DOWNTO 0);
	carga_a_x	:OUT	STD_LOGIC_VECTOR(1 DOWNTO 0);
	sal_alu		:OUT	STD_LOGIC_VECTOR(2 DOWNTO 0);
	carga_a		:OUT	STD_LOGIC;
	carga_x		:OUT	STD_LOGIC;
	ext_ind16	:OUT	STD_LOGIC;
	c_y_n		:OUT	STD_LOGIC;
	carga_carryout	:OUT	STD_LOGIC;
	carga_zero	:OUT	STD_LOGIC;
	carga_negativo	:OUT	STD_LOGIC;
	carga_interru	:OUT	STD_LOGIC;
	carga_carryint	:OUT	STD_LOGIC;
	relativo	:OUT	STD_LOGIC;
	carga_sp	:OUT	STD_LOGIC;
	oe_sp		:OUT	STD_LOGIC;
	push		:OUT	STD_LOGIC;
	pull		:OUT	STD_LOGIC;
	dat_sp		:OUT	STD_LOGIC_VECTOR(2 DOWNTO 0);
	carga_reg_dir	:OUT	STD_LOGIC;
	oe_reg_dir	:OUT	STD_LOGIC;
	oe_reg_sal_alu	:OUT	STD_LOGIC);
END dec_instr;

ARCHITECTURE unidad OF dec_instr IS

	--Se declaran las señales auxiliares.
	TYPE estado IS (reposo,lectura_dato,mbr,decod_dir,direccionamiento,dato_a_bus,
		escribir_dato,dato_a_memo,dato_a_reg,dato_mul,dato_pc,pila,lect_pila_pc,
		lect_pila,carga_registros,cargar_sp);
	--Estados que forman parte de la máquina de estados.
	SIGNAL actual,siguiente		:estado;
	--Señales que indican el estado en el que se encuentra la máquina y el próximo estado
	--respectivamente.
	SIGNAL aux_directo,aux_directo_siguiente	:STD_LOGIC;
	--Señales que indican si el direccionamiento es directo.
	SIGNAL aux_extendido,aux_extendido_siguiente	:STD_LOGIC;
	--Señales que indican si el direccionamiento es extendido.
```

```
        SIGNAL aux_index,aux_index_siguiente        :STD_LOGIC;
        --Señales que indican si el direccionamiento es indexado.
        SIGNAL aux_relativo ,aux_relativo_siguiente :STD_LOGIC;
        --Señales que indican si el direccionamiento es relativo.
        SIGNAL lect_instr,lect_instr_siguiente      :STD_LOGIC;
        --Señales que indican si el siguiente dato de la memoria es una instrucción.
        SIGNAL aux_pila,aux_pila_siguiente          :STD_LOGIC_VECTOR(2 downto 0);
        --Señales que indican el dato a almacenar en la pila.
BEGIN
        PROCESS(reset,clock)
        BEGIN
                IF reset='0' THEN
                        --Se activa la señal que indica direcc. extendido para leer los dos
                        --bytes que forman el vector de reset de la memoria.
                        --Se lleva a la máquina al  primero de los estados.
                        actual          <= reposo;
                        aux_directo     <= '0';
                        aux_extendido   <= '1';
                        aux_index       <= '0';
                        lect_instr      <= '0';
                        aux_pila        <= "000";
                ELSIF clock'EVENT AND clock='1' THEN
                        --En cada flanco de subida de la señal de reloj se pasa al próximo
                        --estado. Las señales auxiliares se actualizan.
                        actual          <= siguiente;
                        aux_directo     <= aux_directo_siguiente;
                        aux_extendido   <= aux_extendido_siguiente;
                        aux_index       <= aux_index_siguiente;
                        aux_relativo    <= aux_relativo_siguiente;
                        lect_instr      <= lect_instr_siguiente;
                        aux_pila        <= aux_pila_siguiente;
                END IF;
        END PROCESS;

        PROCESS(actual,dat_instr,aux_directo,aux_extendido,aux_relativo,lect_instr,
                        aux_index,ccr,IRQ,aux_pila,interrupcion)
        BEGIN
                --Todas las señales toman un valor por defecto para que si en algún estado no se
                --les asigna ninguno no se cree un latch.
                oe_pc                   <= '0';
                inc_pc                  <= '0';
                carga_pc                <= '0';
                relativo                <= '0';
                carga_mar               <= '0';
                read_write              <= '1';
                oe_memo                 <= '0';
                sentido_b_m             <= '1';
                oe_mbr                  <= '0';
                carga_ri                <= '0';
                carga_alu               <= '0';
                sel_op                  <= "01111";  --op nop
                sel_bit                 <= (OTHERS => '0');
                bit_rot                 <= (OTHERS => '0');
                c_y_n                   <= '0';
                oe_alu_b                <= '0';
                carga_reg_dir           <= '0';
                oe_reg_dir              <= '0';
                carga_a                 <= '0';
                carga_carryout          <= '0';
                carga_zero              <= '0';
                carga_negativo          <= '0';
                carga_interru           <= '0';
                carga_carryint          <= '0';
                carga_x                 <= '0';
                carga_sp                <= '0';
                oe_sp                   <= '0';
                push                    <= '0';
                pull                    <= '0';
                dat_sp                  <= "111";
                aux_directo_siguiente   <= '0';
                aux_extendido_siguiente <= '0';
                aux_index_siguiente     <= '0';
                lect_instr_siguiente    <= '0';
                aux_relativo_siguiente  <= '0';
                sal_alu                 <= "111";
                carga_a_x               <= "11";
                ext_ind16               <= '0';
                aux_pila_siguiente      <= "000";
                oe_reg_sal_alu          <= '0';

                CASE actual IS
                        WHEN reposo     =>
                                --En este estado se carga el registro mar con la dirección
                                --del dato a leer.Para ello:
                                oe_pc                   <= '1';
                                -- el dato del contador de programa  pasa al bus de direcc.
                                inc_pc                  <= '1';
                                --se incrementa el contador para que una próxima lectura.
                                aux_extendido_siguiente <= '1';
                                --direccionamiento extendido para leer el vertor de reset.
                                carga_mar               <= '1';
                                --el registro MAR almacena el dato del bus de direcciones.
```

```
            siguiente                   <= lectura_dato;
WHEN lectura_dato =>
            --En este estado se obtiene el dato de la memoria y se pasa
            --al registro MBR. Para ello:
            oe_memo             <= '1';
            --se habilita la memoria para poder leer de ella.
            --Se mantienen los valores de las señales auxiliares.
            IF aux_extendido='1' THEN
                --En el caso de la instrucción RTI o RTS. Hay que
                --incrementar el puntero de pila en este estado para
                --leer el siguiente dato de la pila.
                IF (dat_instr="10000000" OR dat_instr="10000001") THEN
                    pull  <= '1';
                END IF;
                aux_extendido_siguiente <= '1';
            ELSIF aux_directo='1' THEN
                aux_directo_siguiente   <= '1';
            ELSIF aux_index='1' THEN
                aux_index_siguiente     <= '1';
            ELSIF aux_relativo='1' THEN
                aux_relativo_siguiente  <= '1';
            ELSIF aux_pila="001" THEN
                aux_pila_siguiente      <= "001";
            ELSIF aux_pila="010" THEN
                aux_pila_siguiente      <= "010";
            ELSIF lect_instr='1' THEN
                lect_instr_siguiente    <= '1';
            END IF;
            siguiente   <= mbr;
WHEN mbr  =>
            --En este estado se toma el dato de la memoria y se pasa al
            --bus de datos. Para ello:
            sentido_b_m         <= '0';
            --se elige el sentido del MBR para que vaya de la memoria
            --al bus.
            --Se mantienen los valores de las señales auxiliares.
            IF aux_extendido='1' THEN
                aux_extendido_siguiente  <= '1';
                --Si se trata de la intrucc. RTI o RTS se carga el
                --puntero de pila.
                IF (dat_instr="10000001" OR dat_instr="10000000") THEN
                    carga_sp            <= '1';
                    pull                <= '1';
                END IF;
            ELSIF aux_directo='1' THEN
                aux_directo_siguiente   <= '1';
            ELSIF aux_index='1' THEN
                aux_index_siguiente     <= '1';
            ELSIF aux_relativo='1' THEN
                aux_relativo_siguiente  <= '1';
            ELSIF aux_pila="001" THEN
                pull                    <= '1';
                aux_pila_siguiente      <= "001";
            ELSIF aux_pila="010" THEN
                pull                    <= '1';
                aux_pila_siguiente      <= "010";
            ELSIF lect_instr='1' THEN
                lect_instr_siguiente    <= '1';
            ELSIF (dat_instr="10000000" OR dat_instr="10000001") THEN
                pull                    <= '1';
            END IF;
            siguiente   <= dato_a_bus;
WHEN dato_a_bus     =>
            -- En este estado el dato del MBR llega al bus.
            --Dependiendo de: el modo de direccionamiento, de si el
            --dato es una intrucción, o es un operando se actúa de
            --diferente forma.
            oe_mbr              <= '1';
            --se habilita la salida del registro MBR al bus.
            IF aux_extendido='1' THEN
                --Si el direccionesmiento es extendido  o indexado con
                --offset de 16 bits:
                ext_ind16       <= '1';
                --el dato es el byte alto de la dirección y se guarda en
                --el registro reg_aux para más adelante poder formarla.
                carga_mar       <= '1';
                --se carga la siguiente dirección del dato a leer. Este
                --será el byte bajo de la dirección a formar.
                inc_pc          <= '1';
                --se incrementa el contador de programa.
                siguiente       <= lectura_dato;
                --Se activan las señales auxiliares dependiendo del
                --modo de direccionamiento (ext. o index16).
                IF dat_instr(7 DOWNTO 4)="1101" THEN
                    aux_index_siguiente  <= '1';
                ELSE
                    aux_directo_siguiente<= '1';
                END IF;
                --EL dato que se carga en MAR es: o el del puntero de
                --pila (RTI o RTS), o el del contador de programa.
                IF (dat_instr="10000001" OR dat_instr="10000000") THEN
                    oe_sp       <= '1';
```

```
        ELSE
                oe_pc       <= '1';
        END IF;
ELSIF aux_directo='1' THEN
        --para el direccionamiento directo:
        carga_alu           <= '1';
        sel_op              <= "01010";  --op transp
        carga_a_x           <= "10";
        --se pasa el dato del bus al registro reg_sal_alu con la
        --ALU en modo transparente.
        siguiente           <= direccionamiento;
ELSIF aux_index='1' THEN
        --para el direcc. indexado con offset de 8 bits:
        carga_alu           <= '1';
        carga_a_x           <= "01";
        oe_alu_b            <= '1';
        sel_op              <= "00000";
       --se toma el dato del bus y se le suma al dato del
       --registro de indexado.
        siguiente           <= direccionamiento;
ELSIF aux_relativo='1' THEN
        --para el direccionamiento relativo. Dependiendo de la
        --instrucción se consulta cierta condicción. Si ésta es
        --verdadera se salta guardando el dato en reg_sal_alu.
        --Si es falsa no se salta y se carga MAR con la
        --dirección de la siguiente instrucción.
        CASE dat_instr(3 DOWNTO 0) IS
                WHEN "0000" =>
                        IF ((dat_instr(7 downto 4)="0000" and ccr(0)='1')
                             or dat_instr(7 downto 4)="0010") THEN
                                carga_alu  <= '1';
                                carga_a_x <= "10";
                                sel_op      <= "01010";
                                siguiente   <= dato_pc;
                        ELSE
                                carga_mar <= '1';
                                oe_pc       <= '1';
                                lect_instr_siguiente <= '1';
                                siguiente <= lectura_dato;
                        END IF;
                WHEN "0001" =>
                        IF ((dat_instr(7 downto 4)="0000" and ccr(0)='1')
                             or dat_instr(7 downto 4)="0010") THEN
                                carga_mar <= '1';
                                oe_pc       <= '1';
                                lect_instr_siguiente <= '1';
                                siguiente   <= lectura_dato;
                        ELSE
                                carga_alu  <= '1';
                                carga_a_x <= "10";
                                sel_op      <= "01010";
                                siguiente   <= dato_pc;
                        END IF;
                WHEN "0010" =>
                        IF ((dat_instr(7 downto 4)="0000" and ccr(0)='1')
                             or (dat_instr(7 downto 4)="0010" and (ccr(0)
                             or ccr(1))='0')) THEN
                                carga_alu  <= '1';
                                carga_a_x <= "10";
                                sel_op      <= "01010";
                                siguiente   <= dato_pc;
                         ELSE
                                carga_mar <= '1';
                                oe_pc       <= '1';
                                lect_instr_siguiente <= '1';
                                siguiente   <= lectura_dato;
                        END IF;
                WHEN "0011" =>
                        IF ((dat_instr(7 downto 4)="0000" and ccr(0)='0')
                             or (dat_instr(7 downto 4)="0010" and (ccr(0)
                             or ccr(1))='1')) THEN
                                carga_alu  <= '1';
                                carga_a_x <= "10";
                                sel_op      <= "01010";
                                siguiente   <= dato_pc;
                        ELSE
                                carga_mar <= '1';
                                oe_pc       <= '1';
                                lect_instr_siguiente <= '1';
                                siguiente   <= lectura_dato;
                        END IF;
                WHEN "0100" =>
                        IF ((dat_instr(7 downto 4)="0000" and ccr(0)='1')
                             or (dat_instr(7 downto 4)="0010" and
                             ccr(0)='0')) THEN
                                carga_alu  <= '1';
                                carga_a_x <= "10";
                                sel_op      <= "01010";
                                siguiente   <= dato_pc;
                        ELSE
                                carga_mar <= '1';
                                oe_pc       <= '1';
```

```
            lect_instr_siguiente <= '1';
            siguiente   <= lectura_dato;
      END IF;
WHEN "0101" =>
      IF ((dat_instr(7 downto 4)="0000" and ccr(0)='0')
            or (dat_instr(7 downto 4)="0010" and
            ccr(0)='1')) THEN
            carga_alu  <= '1';
            carga_a_x <= "10";
            sel_op       <= "01010";
            siguiente   <= dato_pc;
      ELSE
            carga_mar <= '1';
            oe_pc       <= '1';
            lect_instr_siguiente <= '1';
            siguiente   <= lectura_dato;
      END IF;
WHEN "0110" =>
      IF ((dat_instr(7 downto 4)="0000" and ccr(0)='1')
            or (dat_instr(7 downto 4)="0010" and
            ccr(1)='0')) THEN
            carga_alu  <= '1';
            carga_a_x <= "10";
            sel_op       <= "01010";
            siguiente   <= dato_pc;
      ELSE
            carga_mar <= '1';
            oe_pc       <= '1';
            lect_instr_siguiente <= '1';
            siguiente   <= lectura_dato;
      END IF;
WHEN "0111" =>
      IF ((dat_instr(7 downto 4)="0000" and ccr(0)='0')
            or (dat_instr(7 downto 4)="0010" and
            ccr(1)='1')) THEN
            carga_alu  <= '1';
            carga_a_x <= "10";
            sel_op       <= "01010";
            siguiente   <= dato_pc;
      ELSE
            carga_mar <= '1';
            oe_pc       <= '1';
            lect_instr_siguiente <= '1';
            siguiente   <= lectura_dato;
      END IF;
WHEN "1000" =>
      IF ((dat_instr(7 downto 4)="0000" and ccr(0)='1')
            or (dat_instr(7 downto 4)="0010" and
            ccr(4)='0')) THEN
            carga_alu  <= '1';
            carga_a_x <= "10";
            sel_op       <= "01010";
            siguiente   <= dato_pc;
      ELSE
            carga_mar <= '1';
            oe_pc       <= '1';
            lect_instr_siguiente <= '1';
            siguiente   <= lectura_dato;
      END IF;
WHEN "1001" =>
      IF ((dat_instr(7 downto 4)="0000" and ccr(0)='0')
            or (dat_instr(7 downto 4)="0010" and
            ccr(4)='1')) THEN
            carga_alu  <= '1';
            carga_a_x <= "10";
            sel_op       <= "01010";
            siguiente   <= dato_pc;
      ELSE
            carga_mar <= '1';
            oe_pc       <= '1';
            lect_instr_siguiente <= '1';
            siguiente   <= lectura_dato;
      END IF;
WHEN "1010" =>
      IF ((dat_instr(7 downto 4)="0000" and ccr(0)='1')
            or (dat_instr(7 downto 4)="0010" and
            ccr(2)='0')) THEN
            carga_alu  <= '1';
            carga_a_x <= "10";
            sel_op       <= "01010";
            siguiente   <= dato_pc;
      ELSE
            carga_mar          <= '1';
            oe_pc       <= '1';
            lect_instr_siguiente <= '1';
            siguiente <= lectura_dato;
      END IF;
WHEN "1011" =>
      IF ((dat_instr(7 downto 4)="0000" and ccr(0)='0')
            or (dat_instr(7 downto 4)="0010" and
            ccr(2)='1')) THEN
```

```
                carga_alu  <= '1';
                carga_a_x <= "10";
                sel_op      <= "01010";
                siguiente   <= dato_pc;
            ELSE
                carga_mar <= '1';
                oe_pc       <= '1';
                lect_instr_siguiente <= '1';
                siguiente   <= lectura_dato;
            END IF;
        WHEN "1100" =>
            IF ((dat_instr(7 downto 4)="0000" and ccr(0)='1')
                or (dat_instr(7 downto 4)="0010" and
                ccr(3)='0')) THEN
                carga_alu  <= '1';
                carga_a_x <= "10";
                sel_op      <= "01010";
                siguiente   <= dato_pc;
            ELSE
                carga_mar <= '1';
                oe_pc       <= '1';
                lect_instr_siguiente <= '1';
                siguiente   <= lectura_dato;
            END IF;
        WHEN "1101" =>
            IF ((dat_instr(7 downto 4)="0000" and ccr(0)='0')
                or (dat_instr(7 downto 4)="0010" and
                ccr(3)='1')) THEN
                carga_alu  <= '1';
                carga_a_x <= "10";
                sel_op      <= "01010";
                siguiente   <= dato_pc;
            ELSE
                carga_mar <= '1';
                oe_pc       <= '1';
                lect_instr_siguiente <= '1';
                siguiente   <= lectura_dato;
            END IF;
        WHEN "1110" =>
            IF ((dat_instr(7 downto 4)="0000" and ccr(0)='1')
                or (dat_instr(7 downto 4)="0010" and
                IRQ='0')) THEN
                carga_alu  <= '1';
                carga_a_x <= "10";
                sel_op      <= "01010";
                siguiente   <= dato_pc;
            ELSE
                carga_mar <= '1';
                oe_pc       <= '1';
                lect_instr_siguiente <= '1';
                siguiente   <= lectura_dato;
            END IF;
        WHEN "1111" =>
            IF ((dat_instr(7 downto 4)="0000" and ccr(0)='0')
                or (dat_instr(7 downto 4)="0010" and
                IRQ='1')) THEN
                carga_alu  <= '1';
                carga_a_x <= "10";
                sel_op      <= "01010";
                siguiente   <= dato_pc;
            ELSE
                carga_mar <= '1';
                oe_pc       <= '1';
                lect_instr_siguiente <= '1';
                siguiente   <= lectura_dato;
            END IF;
        WHEN OTHERS =>
    END CASE;
ELSIF lect_instr='1' THEN
    --si el dato es una instrucción:
    carga_ri    <= '1';
    --se carga el registro de instrucción con el dato del bus.
    IF interrupcion='0' THEN
        --Si no ocurre una interrupción hay que
        --incrementar el contador de programa.
        inc_pc      <= '1';
    END IF;
    siguiente       <= decod_dir;
ELSIF dat_instr="10101101" THEN
    --si es la instrucción BSR este dato es el
    --desplazamiento relativo.
    carga_alu       <= '1';
    carga_a_x       <= "10";
    sel_op          <= "01010";
    siguiente       <= direccionamiento;
ELSIF dat_instr="10000000" THEN
    --si es la instrucción RTI el dato se pasa a reg_sal_alu
    --con la ALU en modo transparente y se incrementa sp.
    carga_alu       <= '1';
    carga_a_x       <= "10";
    sel_op          <= "01010";
    carga_sp        <= '1';
```

```
        pull            <= '1';
        IF aux_pila="001" THEN
            --se van devolviendo los datos a los diferentes
            --registros.
            aux_pila_siguiente  <= "001";
            siguiente           <= carga_registros;
        ELSIF aux_pila="010" THEN
            --se devuelve el valor al contador de programa.
            aux_pila_siguiente  <= "010";
            siguiente           <= lect_pila_pc;
        ELSE
            siguiente           <= carga_registros;
        END IF;
    ELSIF (dat_instr(7 downto 4)="0001" or
                dat_instr(7 downto 4)="0000") THEN
        --si es la instrucción BRCLR-BRSET o BCLR-BSET:
        carga_alu       <= '1';
        carga_a_x       <= "10";
        IF dat_instr(7 DOWNTO 4)="0001" THEN
            --Si es BCLR-BSET. Se pasa a guardar el dato
            --en la memoria.
            siguiente           <= dato_a_memo;
        ELSE
            --Si es BRCLR-BRSET. Se actualiza el bit C
            --del ccr y se lee el offset del direcc. relativo.
            aux_relativo_siguiente<= '1';
            carga_carryout      <= '1';
            carga_mar           <= '1';
            oe_pc               <= '1';
            inc_pc              <= '1';
            siguiente           <= lectura_dato;
        END IF;
        CASE dat_instr(3 DOWNTO 0) IS
            --Dependiendo de la intrucción se selecciona el
            --bit del dato y la operación a realizar.
            WHEN "0000" =>  sel_op  <= "01000";
                            sel_bit <= "000";
            WHEN "0001" =>  sel_op  <= "01001";
                            sel_bit <= "000";
            WHEN "0010" =>  sel_op  <= "01000";
                            sel_bit <= "001";
            WHEN "0011" =>  sel_op  <= "01001";
                            sel_bit <= "001";
            WHEN "0100" =>  sel_op  <= "01000";
                            sel_bit <= "010";
            WHEN "0101" =>  sel_op  <= "01001";
                            sel_bit <= "010";
            WHEN "0110" =>  sel_op  <= "01000";
                            sel_bit <= "011";
            WHEN "0111" =>  sel_op  <= "01001";
                            sel_bit <= "011";
            WHEN "1000" =>  sel_op  <= "01000";
                            sel_bit <= "100";
            WHEN "1001" =>  sel_op  <= "01001";
                            sel_bit <= "100";
            WHEN "1010" =>  sel_op  <= "01000";
                            sel_bit <= "101";
            WHEN "1011" =>  sel_op  <= "01001";
                            sel_bit <= "101";
            WHEN "1100" =>  sel_op  <= "01000";
                            sel_bit <= "110";
            WHEN "1101" =>  sel_op  <= "01001";
                            sel_bit <= "110";
            WHEN "1110" =>  sel_op  <= "01000";
                            sel_bit <= "111";
            WHEN "1111" =>  sel_op  <= "01001";
                            sel_bit <= "111";
            WHEN OTHERS =>
        END CASE;
    ELSE
        --si no es ningún modo de direcc. ni ninguna instrucc.:
        carga_alu       <= '1';
        carga_zero      <= '1';
        carga_negativo  <= '1';
        IF dat_instr(7)='1' THEN
            --si el código de operación comienza por el
            --valor A,B,C,D,E, o F
            CASE dat_instr(3 DOWNTO 0) IS
                --Dependiendo de la instrucción se realiza
                --una operación distinta:
                WHEN "0000"     => --SUB
                    carga_a_x <= "00";
                    sel_op    <= "00001";
                    oe_alu_b  <= '1';
                    carga_carryout <= '1';
                    siguiente  <= dato_a_reg;
                WHEN "0010"     =>  --SBC
                    carga_a_x <= "00";
                    sel_op    <= "00001";
                    oe_alu_b  <= '1';
                    carga_carryout<= '1';
                    c_y_n     <= '1';
```

```
        siguiente   <= dato_a_reg;
WHEN "0001"|"0011" => --CMP y CPX
        IF dat_instr(3 downto 0) ="0001" THEN
                carga_a_x <= "00"; --CMP
        ELSE
                carga_a_x <= "01"; --CPX
        END IF;
        oe_alu_b   <= '1';
        sel_op      <= "00001";
        carga_carryout <= '1';
        lect_instr_siguiente <= '1';
        carga_mar <= '1';
        oe_pc       <= '1';
        siguiente <= lectura_dato;
WHEN "0100"        => --AND
        carga_a_x <= "00";
        sel_op      <= "00010";
        oe_alu_b   <= '1';
        siguiente   <= dato_a_reg;
WHEN "0101"       => --BIT
        carga_a_x <= "00";
        oe_alu_b   <= '1';
        sel_op      <= "00010";
        lect_instr_siguiente <= '1';
        carga_mar <= '1';
        oe_pc       <= '1';
        siguiente    <= lectura_dato;
WHEN "0110"|"1110" => --LDA o LDX
        IF dat_instr(3 downto 0)="0110" THEN
                carga_a   <= '1'; --LDA
        ELSE
                carga_x   <= '1'; --LDX
        END IF;
        carga_a_x <= "10";
        sel_op      <= "01010";
        lect_instr_siguiente <= '1';
        carga_mar <= '1';
        oe_pc       <= '1';
        siguiente    <= lectura_dato;
WHEN "1000"        => --EOR
        carga_a_x <= "00";
        sel_op      <= "00100";
        oe_alu_b   <= '1';
        siguiente   <= dato_a_reg;
WHEN "1001"        => --ADC
        carga_a_x <= "00";
                sel_op      <= "00000";
                oe_alu_b   <= '1';
                carga_carryint          <= '1';
                carga_carryout          <= '1';
                c_y_n       <= '1';
                siguiente   <= dato_a_reg;

        WHEN "1010"        => --ORA
                carga_a_x <= "00";
                sel_op      <= "00101";
                oe_alu_b   <= '1';
                siguiente   <= dato_a_reg;
        WHEN "1011"        => --ADD
                carga_a_x <= "00";
                sel_op      <= "00000";
                oe_alu_b   <= '1';
                carga_carryint          <= '1';
                carga_carryout          <= '1';
                siguiente   <= dato_a_reg;
        WHEN OTHERS =>
    END CASE;
ELSE
    -- Si el código de operación comienza por el
    --valor 3,6, o 7:
    carga_a_x <= "10";
    CASE dat_instr(3 DOWNTO 0) IS
        --Dependiendo de la intrucción:
        WHEN "0000"      => --NEG
                sel_op      <= "00001";
                oe_alu_b   <= '1';
                carga_carryout <= '1';
                siguiente   <= dato_a_memo;
        WHEN "0011"      => --COM
                sel_op      <= "00011";
                carga_carryout <= '1';
                siguiente   <= dato_a_memo;
        WHEN "0100"      => --LSR
                sel_op      <= "00110";
                carga_carryout <= '1';
                siguiente   <= dato_a_memo;
        WHEN "0110"      => --ROR
                sel_op      <= "00110";
                carga_carryout <= '1';
                bit_rot     <= "01";
                siguiente   <= dato_a_memo;
        WHEN "0111"      => --ASR
```

```
                    sel_op      <= "00110";
                    bit_rot     <= "11";
                    carga_carryout <= '1';
                    siguiente   <= dato_a_memo;
                WHEN "1000"     => ----ASL o LSL
                    sel_op      <= "00111";
                    carga_carryout <= '1';
                    siguiente   <= dato_a_memo;
                WHEN "1001"     => --ROL
                    sel_op      <= "00111";
                    carga_carryout <= '1';
                    bit_rot     <= "01";
                    siguiente   <= dato_a_memo;
                WHEN "1010"     => --DEC
                    sel_op      <= "01110";
                    siguiente   <= dato_a_memo;
                WHEN "1100"     => --INC
                    sel_op      <= "01101";
                    siguiente   <= dato_a_memo;
                WHEN "1101"     => --TST
                    sel_op      <= "01010";
                    lect_instr_siguiente <= '1';
                    carga_mar <= '1';
                    oe_pc       <= '1';
                    siguiente   <= lectura_dato;
                WHEN "1111"     => --CLR
                    sel_op      <= "01100";
                    siguiente   <= dato_a_memo;
                WHEN OTHERS     =>
            END CASE;
        END IF;
    END IF;
WHEN decod_dir  =>
    --En este estado se descodifica la instrucción según los
    --diferentes modos de direccionamiento.
    CASE dat_instr(7 DOWNTO 4) IS
        --Si es direccionamiento inherente:
        WHEN "1001" | "0100" | "0101" | "1010" | "0010" |
             "1000" =>
            --Dependiendo de la instr. con direcc. inherente
            IF (dat_instr(7 downto 4)="1010" or dat_instr(7
                downto 4)="0010") THEN
                    --Se lee el siguiente dato de la mem.
                    oe_pc       <= '1';
                    carga_mar <= '1';
                    siguiente   <= lectura_dato;
                    inc_pc      <= '1';
                IF dat_instr(7 downto 4)="0010" THEN
                    aux_relativo_siguiente <= '1';
                END IF;
            ELSIF dat_instr="10010111" THEN
                --TAX. Se carga el reg. de indexado.
                carga_alu           <= '1';
                carga_a_x           <= "00";
                sel_op              <= "01010";
                siguiente           <= dato_a_reg;
            ELSIF dat_instr="10011111" THEN
                --TXA. Se carga el acumulador.
                carga_alu           <= '1';
                carga_a_x           <= "01";
                sel_op              <= "01010";
                siguiente           <= dato_a_reg;
            ELSIF (dat_instr="10000001"  OR
                        dat_instr="10000000") THEN
                --RTI o RTS. Se incrementa el sp.
                pull                <= '1';
                siguiente           <= cargar_sp;
            ELSIF (dat_instr="10000011" OR
                        dat_instr="10000010") THEN
                --Si es una interrupción.
                siguiente           <= pila;
            ELSIF (dat_instr="10001110" OR
                        dat_instr="10001111") THEN
                --WAIT o STOP.
                carga_alu           <= '1';
                sel_op              <= "01100";
                carga_interru       <= '1';
                siguiente           <= pila;
            ELSIF dat_instr(7 DOWNTO 4)="1001" THEN
                --SEC, SEI, CLC o CLI:
                oe_pc               <= '1';
                lect_instr_siguiente <= '1';
                carga_mar           <= '1';
                siguiente   <= lectura_dato;
                CASE dat_instr(3 DOWNTO 0) IS
                    WHEN "1000" => --CLC
                        carga_alu  <= '1';
                        carga_carryout <= '1';
                        sel_op      <= "01100";
                    WHEN "1001" => --SEC
                        carga_alu  <= '1';
                        carga_carryout <= '1';
```

```
                sel_op      <= "01011";
            WHEN "1010" => --CLI
                carga_alu  <= '1';
                carga_interru <= '1';
                sel_op      <= "01100";
            WHEN "1011" => --SEI
                carga_alu  <= '1';
                carga_interru <= '1';
                sel_op      <= "01011";
            WHEN OTHERS =>
        END CASE;
ELSIF (dat_instr(7 downto 4)="0100" or
       dat_instr(7 downto 4)="0101") THEN
        --si el código de operación comienza
        --por el valor 4 o 5:
        IF dat_instr(7 downto 4)="0100" THEN
                carga_a_x <= "00";
        ELSE
                carga_a_x <= "01";
        END IF;
        carga_alu  <= '1';
        carga_zero<= '1';
        carga_negativo <= '1';
        CASE dat_instr(3 DOWNTO 0) IS
            WHEN "0000"    => --NEG
                sel_op      <= "00001";
                oe_alu_b   <= '1';
                carga_carryout <= '1';
                siguiente   <= dato_a_reg;
            WHEN "0010"    => --MUL
                sel_op      <= "10000";
                carga_carryout <= '1';
                carga_carryint <= '1';
                carga_zero<= '0';
                carga_negativo <= '0';
                siguiente   <= dato_mul;
            WHEN "0011"    => --COM
                sel_op      <= "00011";
                carga_carryout <= '1';
                siguiente   <= dato_a_reg;
            WHEN "0100"    => --LSR
                sel_op      <= "00110";
                carga_carryout <= '1';
                siguiente   <= dato_a_reg;
            WHEN "0110"    => --ROR
                sel_op      <= "00110";
                carga_carryout <= '1';
                bit_rot     <= "01";
                siguiente   <= dato_a_reg;
            WHEN "0111"    => --ASR
                sel_op      <= "00110";
                bit_rot     <= "11";
                carga_carryout <= '1';
                siguiente   <= dato_a_reg;
            WHEN "1000"    => --ASL-LSL
                sel_op      <= "00111";
                carga_carryout <= '1';
                siguiente   <= dato_a_reg;
            WHEN "1001"    => --ROL
                sel_op      <= "00111";
                carga_carryout <= '1';
                bit_rot     <= "01";
                siguiente   <= dato_a_reg;
            WHEN "1010"    => --DEC
                sel_op      <= "01110";
                siguiente   <= dato_a_reg;
            WHEN "1100"    => --INC
                sel_op      <= "01101";
                siguiente   <= dato_a_reg;
            WHEN "1101"    => --TST
                sel_op      <= "01010";
                lect_instr_siguiente <= '1';
                carga_mar <= '1';
                oe_pc       <= '1';
                siguiente <= lectura_dato;
            WHEN "1111"     => --CLR
                sel_op      <= "01100";
                siguiente   <= dato_a_reg;
            WHEN OTHERS  =>
        END CASE;
    END IF;
WHEN "1011" | "0011" | "0001" | "1100" | "0000"    =>
    --Si el direccionamiento es directo se lee el
    --siguiente dato de la memoria.
    oe_pc                   <= '1';
    carga_mar               <= '1';
    siguiente               <= lectura_dato;
    inc_pc                  <= '1';
    IF dat_instr(7 DOWNTO 4)="1100" THEN
            aux_extendido_siguiente  <= '1';
    ELSE
            aux_directo_siguiente      <= '1';
```

```
                END IF;
            WHEN "1111" | "0111"                    =>
                --Si el direcc. es indexado sin offset se toma el
                --dato del reg. de indexado y se pasa al bus de
                --direcciones.
                siguiente   <=  direccionamiento;
                carga_alu       <= '1';
                sel_op          <= "01010";
                carga_a_x       <= "01";
            WHEN "1110" | "0110" | "1101"           =>
                --Si el direcc. es indexado con offset de 8 bits se
                --lee el siguiente dato de la memoria.
                oe_pc               <= '1';
                inc_pc              <= '1';
                carga_mar           <= '1';
                siguiente           <= lectura_dato;
                IF dat_instr(7 DOWNTO 4)="1101" THEN
                        aux_extendido_siguiente  <= '1';
                ELSE
                        aux_index_siguiente     <= '1';
                END IF;
            WHEN OTHERS    =>
        END CASE;
WHEN direccionamiento  =>
        --En este estado se forma la dirección de memoria según el
        --modo de direccionamiento.
        oe_reg_sal_alu      <= '1';
        IF (dat_instr="10010000" OR dat_instr="10000001" OR
            dat_instr="10000000") THEN
                --si se trata de un reset o de la instr. RTI o RTS
                --se carga en el contador de programa.
                carga_mar           <= '1';
                sal_alu             <= "001";
                carga_pc            <= '1';
                lect_instr_siguiente    <= '1';
                siguiente           <= lectura_dato;
        ELSE
                CASE dat_instr(7 DOWNTO 4) IS
                    WHEN "1011"|"0001"|"0011"|"1111"|"0111"|
                         "0000"|"1010"|"1000" =>
                        --directo o index. sin offset.
                        sal_alu     <= "000";
                    WHEN "1100"             => --extendido.
                        sal_alu     <= "001";
                    WHEN "1101"|"0110"   => --index. 16 bits.
                        sal_alu     <= "011";
                    WHEN "1110"             => --index. 8 bits.
                        sal_alu     <= "010";
                    WHEN OTHERS =>
                END CASE;
                IF (dat_instr(7)='1' and (dat_instr(3 downto 0)="0111"
                    or dat_instr(3 downto 0)="1111") and
                    dat_instr/="10001111") THEN
                        --STA o STX. El dato es la direc. de
                        --memoria donde se escribe.
                        --Se carga en MAR.
                        IF dat_instr(3 downto 0)="0111" THEN
                                carga_a_x    <= "00";
                        ELSE
                                carga_a_x    <= "01";
                        END IF;
                        sel_op              <= "01010";
                        carga_mar           <= '1';
                        carga_alu           <= '1';
                        carga_zero          <= '1';
                        carga_negativo      <= '1';
                        siguiente           <= dato_a_memo;
                ELSIF ((dat_instr(7)='1' and dat_instr(3 downto 0)=
                        "1100") or dat_instr="10000011" or
                        dat_instr="10000010" or dat_instr="10001110"
                        or dat_instr="10001111") THEN
                        --JMP, interrupción, WAIT o STOP.
                        --Se carga el contador de programa.
                        carga_pc            <= '1';
                        carga_mar           <= '1';
                        lect_instr_siguiente    <= '1';
                        siguiente           <= lectura_dato;
                ELSIF (dat_instr(7)='1' and dat_instr(3 downto 0)
                        ="1101") THEN
                        --BSR o JSR. Se guarda la dirección
                        --en reg_dir. Mientras se almacena el
                        --valor del pc en la pila.
                        carga_reg_dir       <= '1';
                        siguiente           <= pila;
                ELSE
                        --en el resto de los casos se carga la
                        --dirección en MAR.
                        carga_mar           <= '1';
                        siguiente           <= lectura_dato;
                END IF;
END IF;
```

```
WHEN dato_a_memo  =>
        --En este estado se le pasa al MBR el dato a escribir en
        --memoria. Si se van a escribir datos en la pila se
        --decrementa el puntero de pila y se modifica la señal
        --auxiliar que indica el dato que se está guardando.
        oe_reg_sal_alu <= '1';
        siguiente <= escribir_dato;
        IF ((dat_instr(7)='1' and dat_instr(3 downto 0)="1101") or
            dat_instr="10000011" or dat_instr="10000010" or
            dat_instr="10001110" or dat_instr="10001111") THEN
                --BSR, JSR, WAIT, STOP o interrupciones.
                --Se decrementa puntero de pila.
                oe_sp           <= '1';
                carga_mar       <= '1';
                push            <= '1';
        END IF;
        IF (aux_pila="001" and dat_instr(7)='1' and
            dat_instr(3 downto 0)="1101") THEN
                aux_pila_siguiente <= "001";
        ELSIF aux_pila="001" THEN
                aux_pila_siguiente <= "010";
        ELSIF aux_pila="010" THEN
                aux_pila_siguiente <= "011";
        ELSIF aux_pila="011" THEN
                aux_pila_siguiente <= "100";
        ELSIF aux_pila="100" THEN
                aux_pila_siguiente <= "101";
        END IF;
WHEN escribir_dato =>
        --En este estado se habilita la memoria para escribir en ella.
        --En el caso de ciertas instrucciones también se prepara el
        --siguiente dato a almacenar en la pila. Si ya se han
        --almacenado todos se comienza con la lectura del vector
        --de interrupción o de la dirección donde debe continuar el
        --programa.
        oe_memo         <= '1';
        read_write      <= '0';
        --Se habilita la memoria en modo escritura.
        IF ((dat_instr(7)='1' and dat_instr(3 downto 0)="1101") or
            dat_instr="10000011" or dat_instr="10000010" or
            dat_instr="10001110" or dat_instr="10001111") THEN
                push    <= '1';
                carga_sp <= '1';
                --Se prepara la siguiente dirección de la pila y el
                --dato a almacenar en ella.
                IF aux_pila="010" THEN
                        carga_a_x       <= "10";
                        dat_sp          <= "011";
                        carga_alu       <= '1';
                        sel_op          <= "01010";
                        aux_pila_siguiente <= "010";
                        siguiente       <= dato_a_memo;
                ELSIF aux_pila="011" THEN
                        carga_a_x       <= "10";
                        dat_sp          <= "010";
                        carga_alu       <= '1';
                        sel_op          <= "01010";
                        aux_pila_siguiente <= "011";
                        siguiente       <= dato_a_memo;
                ELSIF aux_pila="100" THEN
                        carga_a_x       <= "10";
                        dat_sp          <= "100";
                        carga_alu       <= '1';
                        sel_op          <= "01010";
                        aux_pila_siguiente <= "100";
                        siguiente       <= dato_a_memo;
                ELSIF (aux_pila="001" or aux_pila="101") THEN
                        --Si es el ultimo dato a guardar en la
                        --pila :
                        carga_pc        <= '1';
                        IF dat_instr="10101101" THEN
                                --BSR
                                carga_pc   <='1';
                                oe_reg_dir <='1';
                                relativo   <= '1';
                                inc_pc     <= '1';
                                siguiente  <= dato_a_reg;
                        ELSIF (dat_instr="10000011" or
                               dat_instr="10000010" or
                               dat_instr="10001110" or
                               dat_instr="10001111") THEN
                                --Interrup.,WAIT y STOP
                                carga_alu  <= '1';
                                sel_op     <= "01011";
                                carga_interru <= '1';
                                IF dat_instr="10000011" THEN
                                sal_alu    <= "100";
                                ELSE
                                sal_alu    <= "101";
                                END IF;
                                siguiente  <= dato_a_reg;
```

```
                ELSE
                        carga_mar <= '1';
                        oe_reg_dir <='1';
                        lect_instr_siguiente <= '1';
                        siguiente <= lectura_dato;
                END IF;
        ELSE
                oe_pc       <= '1';
                carga_alu   <= '1';
                carga_a_x   <= "10";
                dat_sp      <= "001";
                sel_op      <= "01010";
                aux_pila_siguiente <= "001";
                siguiente   <= dato_a_memo;
        END IF;
ELSE
        --Si es una escritura en la memoria sin que sea
        --un dato a almacenar en la pila.
        oe_pc               <='1';
        carga_mar           <= '1';
        lect_instr_siguiente <= '1';
        siguiente           <= lectura_dato;
END IF;
WHEN dato_a_reg =>
        --En este estado se almacenan los datos en los registros.
        IF (dat_instr(7 downto 4)/="0010" or
                        dat_instr(7 downto 4)/="0000") THEN
                --Se extrae el dato del registro reg_sal_alu al
                --bus de direcciones o de datos.
                oe_reg_sal_alu      <= '1';
                --Dependiendo de la instrucción va a ser el
                --acumulador o el registro de indexado.
                IF (dat_instr(7 downto 4)="0101" or
                                dat_instr="10010111") THEN
                        carga_x <= '1';
                ELSE
                        IF dat_instr="01000010" THEN
                                carga_x <= '1';
                        ELSE
                                carga_a <= '1';
                        END IF;
                END IF;
        END IF;
        --Se carga el contador de programa.
        carga_mar           <= '1';
        oe_pc               <= '1';
        IF (dat_instr="10000011" OR dat_instr="10000010" OR
            dat_instr="10001110" OR dat_instr="10001111") THEN
                --En este caso para leer el vector de interrup.
                lect_instr_siguiente    <= '0';
                carga_a                 <= '0';
                inc_pc                  <= '1';
                aux_extendido_siguiente <= '1';
        ELSE
                --En este caso para leer la siguiente instrucción.
                lect_instr_siguiente    <= '1';
        END IF;
        siguiente           <= lectura_dato;
WHEN dato_mul =>
        --En este estado se guarda el bits más significativo del
        --resultado de una multiplicación en el acumulador además
        --de preparar el menos significativo para guardarlo en el
        --registro de indexado.
        oe_reg_sal_alu      <= '1';
        carga_alu           <= '1';
        carga_a_x           <= "00";
        sel_op              <= "10001";
        carga_a             <= '1';
        siguiente           <= dato_a_reg;
WHEN dato_pc =>
        --En este estado el pc toma el dato de offset en un
        --direccionamiento relativo y lo suma al valor que tiene.
        oe_reg_sal_alu      <= '1';
        sal_alu             <= "000";
        carga_pc            <='1';
        relativo            <= '1';
        inc_pc              <= '1';
        siguiente           <= dato_a_reg;
WHEN pila =>
        --En este estado se prepara el primer dato a almacenar en la
        --pila.
        oe_pc               <= '1';
        carga_alu           <= '1';
        carga_a_x           <= "10";
        sel_op              <= "01010";
        dat_sp              <= "000";
        siguiente           <= dato_a_memo;
WHEN lect_pila_pc =>
        --En este estado se comienza la lectura del dato de la pila
        --para recuperar la dirección del contador de programa.
        oe_sp               <= '1';
        carga_mar           <= '1';
```

```
                siguiente               <= lectura_dato;
                inc_pc                  <= '1';
                aux_extendido_siguiente <= '1';
                IF dat_instr="10000000" THEN
                        carga_x         <= '1';
                        oe_reg_sal_alu  <= '1';
                END IF;
        WHEN lect_pila =>
                --En este estado se toma el dato del sp para leer de la pila.
                oe_sp                   <= '1';
                carga_mar               <= '1';
                siguiente               <= lectura_dato;
                inc_pc                  <= '1';
        WHEN carga_registros =>
                --En este estado los diferentes registros recuperan el dato
                --de la pila.
                oe_reg_sal_alu          <= '1';
                oe_sp                   <= '1';
                carga_mar               <= '1';
                siguiente               <= lectura_dato;
                IF aux_pila="000" THEN
                        carga_negativo  <= '1';
                        carga_zero      <= '1';
                        carga_interru   <= '1';
                        carga_carryout  <= '1';
                        carga_carryint  <= '1';
                        aux_pila_siguiente <= "001";
                ELSIF aux_pila="001" THEN
                        carga_a         <= '1';
                        aux_pila_siguiente <= "010";
                END IF;
        WHEN cargar_sp =>
                --En este estado se incrementa el sp para recuperar de la
                --pila el dato del contador de programa.
                pull <= '1';
                carga_sp  <= '1';
                IF dat_instr="10000001" THEN
                        siguiente       <= lect_pila_pc;
                ELSE
                        siguiente       <= lect_pila;
                END IF;
        WHEN OTHERS     =>
    END CASE;
  END PROCESS;
END unidad;
```

--Fichero **unidad.vhd**--

```
--NÚCLEO DEL MICROCONTROLADOR 68HC05.
--En este componente se han unido la ruta de datos con la unidad de control (decodificador de
--instrucciones).
--Consta de varios procesos además de la instanciación de todos los componentes para unirlos entre sí.
--Los procesos se encargan de la gestión del bus de direcciones, del bus de datos, de los multiplexores
--que llevan ciertas entradas que reciben más de un dato, de las interrupciones y de las señales que
--paran el reloj en los modos de bajo consumo.


LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_signed.all;

ENTITY unicon IS
 PORT(  clk                     :IN     STD_LOGIC;
        reset                   :IN     STD_LOGIC;
        interrup_SPI            :IN     STD_LOGIC;
        interrup_SCI            :IN     STD_LOGIC;
        interrup_ADC            :IN     STD_LOGIC;
        interrup_TIM            :IN     STD_LOGIC;
        interrup_PWM            :IN     STD_LOGIC;
        IRQ                     :IN     STD_LOGIC;
        mbr_ent                 :IN     STD_LOGIC_VECTOR(7 DOWNTO 0);
        mbr_sal                 :OUT    STD_LOGIC_VECTOR(7 DOWNTO 0);
        clk_perifericos         :OUT    STD_LOGIC;
        mar_sal                 :OUT    STD_LOGIC_VECTOR(12 DOWNTO 0);
        read_write              :OUT    STD_LOGIC;
        oe_memo                 :OUT    STD_LOGIC;
        sal_ri                  :OUT    STD_LOGIC_VECTOR(7 DOWNTO 0);
        carga_mar               :OUT    STD_LOGIC;
        sal_ccr                 :OUT    STD_LOGIC_VECTOR(4 DOWNTO 0));
END unicon;

ARCHITECTURE unidad_con OF unicon IS

        --Se instancian todos los components que se van a utilizar.
        COMPONENT acumulador
          PORT(  clock          :IN     STD_LOGIC;
                 reset          :IN     STD_LOGIC;
                 dat_ent_a      :IN     STD_LOGIC_VECTOR(7 DOWNTO 0);
                 carga_a        :IN     STD_LOGIC;
                 dat_sal_a      :OUT    STD_LOGIC_VECTOR(7 DOWNTO 0));
        END COMPONENT;
```

```
COMPONENT alu_sum_res
 PORT(   carga_alu          :IN     STD_LOGIC;
         a                  :IN     STD_LOGIC_VECTOR(7 DOWNTO 0);
         b                  :IN     STD_LOGIC_VECTOR(7 DOWNTO 0);
         carryin            :IN     STD_LOGIC;
         c_y_n              :IN     STD_LOGIC;
         sel_op             :IN     STD_LOGIC_VECTOR(4 DOWNTO 0);
         sel_bit            :IN     STD_LOGIC_VECTOR(2 DOWNTO 0);
         bit_rot            :IN     STD_LOGIC_VECTOR(1 DOWNTO 0);
         oe_alu_b           :IN     STD_LOGIC;
         interrupcion       :IN     STD_LOGIC;
         dat_sal_alu        :OUT    STD_LOGIC_VECTOR(7 DOWNTO 0);
         carryout           :OUT    STD_LOGIC;
         zero               :OUT    STD_LOGIC;
         negativo           :OUT    STD_LOGIC;
         carryint           :OUT    STD_LOGIC;
         interrup           :OUT    STD_LOGIC);
END COMPONENT;
COMPONENT ccr
 PORT(   clock              :IN     STD_LOGIC;
         reset              :IN     STD_LOGIC;
         carga_carryout     :IN     STD_LOGIC;
         carga_zero         :IN     STD_LOGIC;
         carga_negativo     :IN     STD_LOGIC;
         carga_carryint     :IN     STD_LOGIC;
         carga_interru      :IN     STD_LOGIC;
         carryout           :IN     STD_LOGIC;
         zero               :IN     STD_LOGIC;
         negativo           :IN     STD_LOGIC;
         interrupcion       :IN     STD_LOGIC;
         carryint           :IN     STD_LOGIC;
         dat_sal_ccr        :OUT    STD_LOGIC_VECTOR(4 DOWNTO 0));
END COMPONENT;
COMPONENT  dec_instr
 PORT(   clock              :IN     STD_LOGIC;
         reset              :IN     STD_LOGIC;
         dat_instr          :IN     STD_LOGIC_VECTOR(7 DOWNTO 0);
         ccr                :IN     STD_LOGIC_VECTOR(4 DOWNTO 0);
         IRQ                :IN     STD_LOGIC;
         interrupcion       :IN     STD_LOGIC;
         carga_alu          :OUT    STD_LOGIC;
         carga_mar          :OUT    STD_LOGIC;
         oe_mbr             :OUT    STD_LOGIC;
         sentido_b_m        :OUT    STD_LOGIC;
         oe_alu_b           :OUT    STD_LOGIC;
         carga_ri           :OUT    STD_LOGIC;
         oe_pc              :OUT    STD_LOGIC;
         inc_pc             :OUT    STD_LOGIC;
         carga_pc           :OUT    STD_LOGIC;
         oe_memo            :OUT    STD_LOGIC;
         read_write         :OUT    STD_LOGIC;
         sel_op             :OUT    STD_LOGIC_VECTOR(4 DOWNTO 0);
         sel_bit            :OUT    STD_LOGIC_VECTOR(2 DOWNTO 0);
         bit_rot            :OUT    STD_LOGIC_VECTOR(1 DOWNTO 0);
         carga_a_x          :OUT    STD_LOGIC_VECTOR(1 DOWNTO 0);
         sal_alu            :OUT    STD_LOGIC_VECTOR(2 DOWNTO 0);
         carga_a            :OUT    STD_LOGIC;
         carga_x            :OUT    STD_LOGIC;
         ext_ind16          :OUT    STD_LOGIC;
         c_y_n              :OUT    STD_LOGIC;
         carga_carryout     :OUT    STD_LOGIC;
         carga_zero         :OUT    STD_LOGIC;
         carga_negativo     :OUT    STD_LOGIC;
         carga_interru      :OUT    STD_LOGIC;
         carga_carryint     :OUT    STD_LOGIC;
         relativo           :OUT    STD_LOGIC;
         carga_sp           :OUT    STD_LOGIC;
         oe_sp              :OUT    STD_LOGIC;
         push               :OUT    STD_LOGIC;
         pull               :OUT    STD_LOGIC;
         dat_sp             :OUT    STD_LOGIC_VECTOR(2 DOWNTO 0);
         carga_reg_dir      :OUT    STD_LOGIC;
         oe_reg_dir         :OUT    STD_LOGIC;
         oe_reg_sal_alu     :OUT    STD_LOGIC);
END COMPONENT;
COMPONENT  mar
 PORT(   clock              :IN     STD_LOGIC;
         reset              :IN     STD_LOGIC;
         dat_ent_mar        :IN     STD_LOGIC_VECTOR(12 DOWNTO 0);
         carga_mar          :IN     STD_LOGIC;
         dat_sal_mar        :OUT    STD_LOGIC_VECTOR(12 DOWNTO 0));
END COMPONENT;
COMPONENT  mbr
 PORT(   clock              :IN     STD_LOGIC;
         reset              :IN     STD_LOGIC;
         dat_ent_mbr_b      :IN     STD_LOGIC_VECTOR(7 DOWNTO 0);
         dat_sal_mbr_b      :OUT    STD_LOGIC_VECTOR(7 DOWNTO 0);
         dat_ent_mbr_m      :IN     STD_LOGIC_VECTOR(7 DOWNTO 0);
         dat_sal_mbr_m      :OUT    STD_LOGIC_VECTOR(7 DOWNTO 0);
         sentido_b_m        :IN     STD_LOGIC);
END COMPONENT;
```

```
COMPONENT  pc
 PORT(   clock              :IN     STD_LOGIC;
         reset              :IN     STD_LOGIC;
         inc_pc             :IN     STD_LOGIC;
         relativo           :IN     STD_LOGIC;
         carga_pc           :IN     STD_LOGIC;
         dat_ent_pc         :IN     STD_LOGIC_VECTOR(12 DOWNTO 0);
         dat_sal_pc         :OUT    STD_LOGIC_VECTOR(12 DOWNTO 0));
END COMPONENT;
COMPONENT ri
 PORT(   clock              :IN     STD_LOGIC;
         reset              :IN     STD_LOGIC;
         dat_ent_ri         :IN     STD_LOGIC_VECTOR(7 DOWNTO 0);
         carga_ri           :IN     STD_LOGIC;
         dat_sal_ri         :OUT    STD_LOGIC_VECTOR(7 DOWNTO 0));
END COMPONENT;
COMPONENT  x
 PORT(   clock              :IN     STD_LOGIC;
         reset              :IN     STD_LOGIC;
         dat_ent_x          :IN     STD_LOGIC_VECTOR(7 DOWNTO 0);
         carga_x            :IN     STD_LOGIC;
         dat_sal_x          :OUT    STD_LOGIC_VECTOR(7 DOWNTO 0));
END COMPONENT;
COMPONENT  sp
 PORT(   clock              :IN     STD_LOGIC;
         reset              :IN     STD_LOGIC;
         carga_sp           :IN     STD_LOGIC;
         push               :IN     STD_LOGIC;
         pull               :IN     STD_LOGIC;
         dat_sal_sp         :OUT    STD_LOGIC_VECTOR(7 DOWNTO 0));
END COMPONENT;
COMPONENT  reg_dir
 PORT(   clock              :IN     STD_LOGIC;
         reset              :IN     STD_LOGIC;
         carga_reg_dir      :IN     STD_LOGIC;
         dat_ent_reg_dir    :IN     STD_LOGIC_VECTOR(12 DOWNTO 0);
         dat_sal_reg_dir    :OUT    STD_LOGIC_VECTOR(12 DOWNTO 0));
END COMPONENT;
COMPONENT reg_aux
 PORT(   clock              :IN     STD_LOGIC;
         reset              :IN     STD_LOGIC;
         ext_ind16          :IN     STD_LOGIC;
         dat_ent_reg_aux    :IN     STD_LOGIC_VECTOR(4 DOWNTO 0);
         dat_sal_reg_aux    :OUT    STD_LOGIC_VECTOR(4 DOWNTO 0));
END COMPONENT;
```

```
COMPONENT reg_sal_alu
 PORT(   clock              :IN     STD_LOGIC;
         reset              :IN     STD_LOGIC;
         dat_ent_reg        :IN     STD_LOGIC_VECTOR(7 DOWNTO 0);
         dat_ent_carry      :IN     STD_LOGIC;
         dat_sal_carry      :OUT    STD_LOGIC;
         dat_sal_reg        :OUT    STD_LOGIC_VECTOR(7 DOWNTO 0));
END COMPONENT;

--Se declaran las señales auxiliaries que se utilizan:
SIGNAL  bus_datos           :STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL  bus_direccion       :STD_LOGIC_VECTOR(12 DOWNTO 0);
SIGNAL  reg_status          :STD_LOGIC_VECTOR(4 DOWNTO 0);
SIGNAL  aux_carga_alu       :STD_LOGIC;
SIGNAL  aux_carga_a         :STD_LOGIC;
SIGNAL  aux_carga_ri        :STD_LOGIC;
SIGNAL  aux_carga_mar       :STD_LOGIC;
SIGNAL  aux_oe_mbr          :STD_LOGIC;
SIGNAL  aux_sentido         :STD_LOGIC;
SIGNAL  aux_oe_pc           :STD_LOGIC;
SIGNAL  aux_inc_pc          :STD_LOGIC;
SIGNAL  aux_carga_pc        :STD_LOGIC;
SIGNAL  aux_sel_op          :STD_LOGIC_VECTOR(4 DOWNTO 0);
SIGNAL  aux_sel_bit         :STD_LOGIC_VECTOR(2 DOWNTO 0);
SIGNAL  aux_bit_rot         :STD_LOGIC_VECTOR(1 DOWNTO 0);
SIGNAL  aux_carryout        :STD_LOGIC;
SIGNAL  aux_zero            :STD_LOGIC;
SIGNAL  aux_negativo        :STD_LOGIC;
SIGNAL  aux_carryint        :STD_LOGIC;
SIGNAL  aux_interrup        :STD_LOGIC;
SIGNAL  aux_sal_alu         :STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL  aux_sal_a           :STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL  aux_sal_x           :STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL  aux_ent_dec         :STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL  aux_alu_a           :STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL  aux_alu_b           :STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL  aux_carga_a_x       :STD_LOGIC_VECTOR(1 DOWNTO 0);
SIGNAL  salida_alu          :STD_LOGIC_VECTOR(2 DOWNTO 0);
SIGNAL  aux_carga_x         :STD_LOGIC;
SIGNAL  aux_ext_ind16       :STD_LOGIC;
SIGNAL  aux_c_y_n           :STD_LOGIC;
SIGNAL  aux_oe_alu_b        :STD_LOGIC;
SIGNAL  aux_carga_cout      :STD_LOGIC;
SIGNAL  aux_carga_zero      :STD_LOGIC;
SIGNAL  aux_carga_negativo  :STD_LOGIC;
SIGNAL  aux_carga_interru   :STD_LOGIC;
```

```
        SIGNAL  aux_carga_cint          :STD_LOGIC;
        SIGNAL  aux_relativo            :STD_LOGIC;
        SIGNAL  aux_carga_sp            :STD_LOGIC;
        SIGNAL  aux_oe_sp               :STD_LOGIC;
        SIGNAL  aux_sal_sp              :STD_LOGIC_VECTOR(7 DOWNTO 0);
        SIGNAL  aux_push                :STD_LOGIC;
        SIGNAL  aux_pull                :STD_LOGIC;
        SIGNAL  reset_sp                :STD_LOGIC;
        SIGNAL  aux_dat_sp              :STD_LOGIC_VECTOR(2 DOWNTO 0);
        SIGNAL  aux_cout                :STD_LOGIC;
        SIGNAL  aux_z                   :STD_LOGIC;
        SIGNAL  aux_n                   :STD_LOGIC;
        SIGNAL  aux_cint                :STD_LOGIC;
        SIGNAL  aux_interr              :STD_LOGIC;
        SIGNAL  aux_stop_cpu            :STD_LOGIC;
        SIGNAL  aux_stop_osc            :STD_LOGIC;
        SIGNAL  sclk                    :STD_LOGIC;
        SIGNAL  aux_ent_ri              :STD_LOGIC_VECTOR(7 DOWNTO 0);
        SIGNAL  aux_INT_IRQ             :STD_LOGIC;
        SIGNAL  aux_INT_TIM             :STD_LOGIC;
        SIGNAL  aux_INT_SCI             :STD_LOGIC;
        SIGNAL  aux_INT_SPI             :STD_LOGIC;
        SIGNAL  aux_INT_ADC             :STD_LOGIC;
        SIGNAL  aux_INT_PWM             :STD_LOGIC;
        SIGNAL  aux_carga_reg_dir       :STD_LOGIC;
        SIGNAL  aux_oe_reg_dir          :STD_LOGIC;
        SIGNAL  interrupcion            :STD_LOGIC;
        SIGNAL  aux_dat_reg_aux         :STD_LOGIC_VECTOR(4 DOWNTO 0);
        SIGNAL  aux_oe_reg              :STD_LOGIC;
        SIGNAL  aux_sal_reg_sal_alu     :STD_LOGIC_VECTOR(7 DOWNTO 0);
        SIGNAL  aux_sal_mbr             :STD_LOGIC_VECTOR(7 DOWNTO 0);
        SIGNAL  aux_carry               :STD_LOGIC;
        SIGNAL  aux_sal_pc              :STD_LOGIC_VECTOR(12 DOWNTO 0);
        SIGNAL  aux_sal_reg_dir         :STD_LOGIC_VECTOR(12 DOWNTO 0);
BEGIN
        --Instanciación de los diferentes componentes:
        ca: acumulador    PORT MAP (  clock           =>  sclk,
                                      reset           =>  reset,
                                      dat_ent_a       =>  bus_datos,
                                      carga_a         =>  aux_carga_a,
                                      dat_sal_a       =>  aux_sal_a);
        calu: alu_sum_res PORT MAP (carga_alu         =>  aux_carga_alu,
                                      a               =>  aux_alu_a,
                                      b               =>  aux_alu_b,
                                      carryin         =>  reg_status(0),
                                      c_y_n           =>  aux_c_y_n,
                                      sel_op          =>  aux_sel_op,
                                      sel_bit         =>  aux_sel_bit,
                                      bit_rot         =>  aux_bit_rot,
                                      oe_alu_b        =>  aux_oe_alu_b,
                                      interrupcion    =>  reg_status(3),
                                      dat_sal_alu     =>  aux_sal_alu,
                                      carryout        =>  aux_cout,
                                      zero            =>  aux_z,
                                      negativo        =>  aux_n,
                                      carryint        =>  aux_cint,
                                      interrup        =>  aux_interr);
        cccr: ccr         PORT MAP (  clock           =>  sclk,
                                      reset           =>  reset,
                                      carga_carryout  =>  aux_carga_cout,
                                      carga_zero      =>  aux_carga_zero,
                                      carga_negativo  =>  aux_carga_negativo,
                                      carga_interru   =>  aux_carga_interru,
                                      carga_carryint  =>  aux_carga_cint,
                                      carryout        =>  aux_carryout,
                                      zero            =>  aux_zero,
                                      negativo        =>  aux_negativo,
                                      interrupcion    =>  aux_interrup,
                                      carryint        =>  aux_carryint,
                                      dat_sal_ccr     =>  reg_status);
        cde: dec_instr    PORT MAP (  clock           =>  sclk,
                                      reset           =>  reset,
                                      dat_instr       =>  aux_ent_dec,
                                      ccr             =>  reg_status,
                                      IRQ             =>  IRQ,
                                      interrupcion    =>  interrupcion,
                                      carga_alu       =>  aux_carga_alu,
                                      carga_mar       =>  aux_carga_mar,
                                      oe_mbr          =>  aux_oe_mbr,
                                      sentido_b_m     =>  aux_sentido,
                                      oe_alu_b        =>  aux_oe_alu_b,
                                      carga_ri        =>  aux_carga_ri,
                                      oe_pc           =>  aux_oe_pc,
                                      inc_pc          =>  aux_inc_pc,
                                      carga_pc        =>  aux_carga_pc,
                                      oe_memo         =>  oe_memo,
                                      read_write      =>  read_write,
                                      sel_op          =>  aux_sel_op,
                                      sel_bit         =>  aux_sel_bit,
                                      bit_rot         =>  aux_bit_rot,
                                      carga_a_x       =>  aux_carga_a_x,
                                      sal_alu         =>  salida_alu,
```

```
                                carga_a          =>  aux_carga_a,
                                carga_x          =>  aux_carga_x,
                                ext_ind16        =>  aux_ext_ind16,
                                c_y_n            =>  aux_c_y_n,
                                carga_carryout   =>  aux_carga_cout,
                                carga_zero       =>  aux_carga_zero,
                                carga_negativo   =>  aux_carga_negativo,
                                carga_interru    =>  aux_carga_interru,
                                carga_carryint   =>  aux_carga_cint,
                                relativo         =>  aux_relativo,
                                carga_sp         =>  aux_carga_sp,
                                oe_sp            =>  aux_oe_sp,
                                push             =>  aux_push,
                                pull             =>  aux_pull,
                                dat_sp           =>  aux_dat_sp,
                                carga_reg_dir    =>  aux_carga_reg_dir,
                                oe_reg_dir       =>  aux_oe_reg_dir,
                                oe_reg_sal_alu   =>  aux_oe_reg);
cmar: mar       PORT MAP (      clock            =>  sclk,
                                reset            =>  reset,
                                dat_ent_mar      =>  bus_direccion,
                                carga_mar        =>  aux_carga_mar,
                                dat_sal_mar      =>  mar_sal);
cmbr: mbr       PORT MAP (      clock            =>  sclk,
                                reset            =>  reset,
                                dat_ent_mbr_b    =>  bus_datos,
                                dat_sal_mbr_b    =>  aux_sal_mbr,
                                dat_ent_mbr_m    =>  mbr_ent,
                                dat_sal_mbr_m    =>  mbr_sal,
                                sentido_b_m      =>  aux_sentido);
cpc: pc         PORT MAP (      clock            =>  sclk,
                                reset            =>  reset,
                                inc_pc           =>  aux_inc_pc,
                                relativo         =>  aux_relativo,
                                carga_pc         =>  aux_carga_pc,
                                dat_ent_pc       =>  bus_direccion,
                                dat_sal_pc       =>  aux_sal_pc);
cri: ri         PORT MAP (      clock            =>  sclk,
                                reset            =>  reset,
                                dat_ent_ri       =>  aux_ent_ri,
                                carga_ri         =>  aux_carga_ri,
                                dat_sal_ri       =>  aux_ent_dec);
cx: x           PORT MAP (      clock            =>  sclk,
                                reset            =>  reset,
                                dat_ent_x        =>  bus_datos,
                                carga_x          =>  aux_carga_x,
                                dat_sal_x        =>  aux_sal_x);
csp: sp         PORT MAP (      clock            =>  sclk,
                                reset            =>  reset_sp,
                                carga_sp         =>  aux_carga_sp,
                                push             =>  aux_push,
                                pull             =>  aux_pull,
                                dat_sal_sp       =>  aux_sal_sp);
creg_dir: reg_dir PORT MAP (    clock            =>  sclk,
                                reset            =>  reset,
                                carga_reg_dir    =>  aux_carga_reg_dir,
                                dat_ent_reg_dir  =>  bus_direccion,
                                dat_sal_reg_dir  =>  aux_sal_reg_dir);
creg_aux: reg_aux PORT MAP (clock                =>  sclk,
                                reset            =>  reset,
                                ext_ind16        =>  aux_ext_ind16,
                                dat_ent_reg_aux  =>  bus_datos(4 DOWNTO 0),
                                dat_sal_reg_aux  =>  aux_dat_reg_aux);
creg_alu: reg_sal_alu PORT MAP (clock            =>  sclk,
                                reset            =>  reset,
                                dat_ent_reg      =>  aux_sal_alu,
                                dat_ent_carry    =>  aux_cout,
                                dat_sal_carry    =>  aux_carry,
                                dat_sal_reg      =>  aux_sal_reg_sal_alu);

--En este proceso se gestiona la entrada al operando A de la ALU.
PROCESS(aux_carga_a_x,aux_sal_a,aux_sal_x,bus_datos,aux_ent_dec,aux_sel_op,
        aux_sal_alu,bus_direccion,aux_dat_sp,aux_sal_sp,reg_status)
BEGIN
        CASE aux_carga_a_x IS
            WHEN "00"   => --Operando A= acumulador.
                IF (aux_ent_dec(7)='0' and aux_ent_dec(3 DOWNTO 0)="0000"
                    AND aux_sel_op="00001") THEN
                        --Instrucción NEG.
                        aux_alu_a   <=  (OTHERS=>'0');
                ELSE
                        --Para el resto de instrucciones.
                        aux_alu_a   <=  aux_sal_a;
                END IF;
            WHEN "01"   => --Operando A= registro de indexado.
                IF (aux_ent_dec(7)='0' AND aux_ent_dec(3 DOWNTO 0)="0000"
                    AND aux_sel_op="00001") THEN
                        --Instrucción NEG.
                        aux_alu_a   <=  (OTHERS=>'0');
                ELSE
                        aux_alu_a   <=  aux_sal_x;
                END IF;
```

```
                WHEN "10"   => --Operando A= bus de datos memos en determinados casos:
                        IF (aux_ent_dec(7)='0' AND aux_ent_dec(3 DOWNTO 0)="0000"
                            AND aux_sel_op="00001") THEN
                                --Instrucción NEG.
                                aux_alu_a   <=  (OTHERS=>'0');
                        ELSIF (aux_ent_dec(3 downto 0)="1101" or aux_ent_dec(3 downto 0)="0011"
                                or aux_ent_dec="10001110" or aux_ent_dec="10001111" or
                                aux_ent_dec="10000010") THEN
                                --Instrucciones que almacenan datos en la pila.
                                CASE aux_dat_sp IS
                                    WHEN "000"   =>
                                        aux_alu_a<=  bus_direccion(7 DOWNTO 0);
                                    WHEN "001"   =>
                                        aux_alu_a(4 downto 0)<=  bus_direccion(12 downto 8);
                                        aux_alu_a(7 DOWNTO 5)<=  "000";
                                    WHEN "010"   =>
                                        aux_alu_a               <=  aux_sal_a;
                                    WHEN "011"   =>
                                        aux_alu_a               <=  aux_sal_x;
                                    WHEN "100"   =>
                                        aux_alu_a(4 DOWNTO 0)     <=  reg_status;
                                        aux_alu_a(7 DOWNTO 5)      <=  "111";
                                    WHEN OTHERS  =>
                                        aux_alu_a           <=  bus_datos;
                                END CASE;
                        ELSE
                                aux_alu_a   <=  bus_datos;
                        END IF;
                WHEN OTHERS =>
                        aux_alu_a   <=  aux_sal_a;
        END CASE;
END PROCESS;

--Gestión del bus de direcciones.
PROCESS(salida_alu,aux_dat_reg_aux,aux_sal_reg_sal_alu,aux_carry,aux_oe_pc,
        aux_oe_reg_dir,aux_oe_sp,aux_sal_sp,aux_sal_pc,aux_sal_reg_dir,aux_int_IRQ,
        aux_int_TIM,aux_int_SCI,aux_int_SPI,aux_int_ADC,aux_int_PWM)
BEGIN
        --Dependiendo del registro que habilite su salida le llega un dato:
        IF aux_oe_pc='1' THEN
                --Recibe el dato del contador de programa.
                bus_direccion <= aux_sal_pc;
        ELSIF aux_oe_reg_dir='1' THEN
                --Recibe el dato del registro auxiliar reg_dir.
                bus_direccion <= aux_sal_reg_dir;
ELSIF aux_oe_sp='1' THEN
        --Recibe el dato del puntero de pila.
        bus_direccion(7 DOWNTO 0)    <= aux_sal_sp;
        bus_direccion(12 DOWNTO 8)  <= "00000";
ELSE
        --Conforma la dirección según el modo de direccionamiento:
        CASE salida_alu IS
            WHEN "000"  => --direccionamiento directo e indexado sin offset.
                bus_direccion(7 DOWNTO 0)    <= aux_sal_reg_sal_alu;
                bus_direccion(12 DOWNTO 8)  <= "00000";
            WHEN "001"  => --direccionamiento extendido.
                bus_direccion(7 DOWNTO 0)    <= aux_sal_reg_sal_alu;
                bus_direccion(12 DOWNTO 8)  <= aux_dat_reg_aux;
            WHEN "010"  => --direccionamiento indexado con offset de 8 bits.
                bus_direccion(7 DOWNTO 0)    <= aux_sal_reg_sal_alu;
                IF aux_carry='1' THEN
                        bus_direccion(12 DOWNTO 8)<= "00001";
                ELSE
                        bus_direccion(12 DOWNTO 8)<= "00000";
                END IF;
            WHEN "011"  => --direccionamiento index. con offset de 16 bits.
                bus_direccion(7 DOWNTO 0)    <= aux_sal_reg_sal_alu;
                IF aux_carry='1' THEN
                        bus_direccion(12 downto 8)<= aux_dat_reg_aux +'1';
                ELSE
                        bus_direccion(12 DOWNTO 8)<= aux_dat_reg_aux;
                END IF;
            WHEN "100"  => --vector de interrupción SWI.
                bus_direccion           <= "1111111111100";
            WHEN "101"  =>
                IF aux_int_IRQ ='1' THEN
                        --vector de interrupción IRQ.
                        bus_direccion    <= "1111111111010";
                ELSIF aux_int_TIM ='1' THEN
                        --vector de interrupción del temporizador.
                        bus_direccion    <= "1111111111000";
                ELSIF aux_int_SCI ='1' THEN
                        --vector de interrupción del puesto serie SCI.
                        bus_direccion    <= "1111111110110";
                ELSIF aux_int_SPI ='1' THEN
                         --vector de interrupción del puerto serie SPI.
                        bus_direccion    <= "1111111110100";
                ELSIF aux_int_ADC ='1' THEN
                        --vector de interrupción del convertidor A/D.
                        bus_direccion    <= "1111111110010";
```

```
                        ELSIF aux_int_PWM ='1' THEN
                            --vector de interrupción del generador de PWM.
                            bus_direccion   <= "1111111110000";
                        ELSE
                            bus_direccion   <= (OTHERS => 'Z');
                        END IF;
                    WHEN OTHERS =>
                        bus_direccion       <= (OTHERS => 'Z');
                END CASE;
        END IF;
END PROCESS;

--Gestión del bus de datos.
PROCESS(aux_oe_mbr,aux_sal_mbr,aux_oe_reg,aux_sal_reg_sal_alu)
BEGIN
        --Dependiendo de la señal que habilita la salida de los registros le llegará el dato
        --de:
        IF aux_oe_reg='1' THEN
                --el registro auxiliar que almacena los resultados de la ALU.
                bus_datos <= aux_sal_reg_sal_alu;
        ELSIF aux_oe_mbr='1' THEN
                --el registro MBR.
                bus_datos <= aux_sal_mbr;
        ELSE
                bus_datos <= (OTHERS => 'Z');
        END IF;
END PROCESS;

--Gestión del operando B de la ALU.
PROCESS(bus_datos,aux_alu_b,aux_ent_dec,aux_sal_a,aux_sal_x,aux_sel_op)
BEGIN
        IF (aux_ent_dec(3 DOWNTO 0)="0000" AND aux_sel_op="00001") THEN
                --Instrucción  NEG.
                CASE aux_ent_dec(7 DOWNTO 4) IS
                    WHEN "0100"             => aux_alu_b <= aux_sal_a;
                    WHEN "0101"             => aux_alu_b <= aux_sal_x;
                    WHEN OTHERS             => aux_alu_b <= bus_datos;
                END CASE;
        ELSIF  aux_ent_dec(7 DOWNTO 0)="01000010" THEN
                --Instrucción MUL.
                aux_alu_b <= aux_sal_x;
        ELSE
                --Dato del bus de datos.
                aux_alu_b <= bus_datos;
        END IF;
END PROCESS;
```

```
--Gestión del reset del puntero de pila.
PROCESS(aux_ent_dec,reset_sp,reset)
BEGIN
        --El reset del puntero de pila tiene que darse si se produce un reset de todo el
        --microcontrolador o si se ejecuta la instrucción RSP.
        IF (aux_ent_dec="10011100" OR reset='0') THEN
                reset_sp  <= '0';
        ELSE
                reset_sp  <= '1';
        END IF;
END PROCESS;

--Gestión de las entradas al registro de estado.
PROCESS(bus_datos,aux_carryout,aux_zero,aux_negativo,aux_interrup,aux_carryint,
                aux_ent_dec,aux_cout,aux_z,aux_n,aux_interr,aux_cint)
BEGIN
        IF aux_ent_dec="10000000" THEN
                --Si se trata de la instrucción RTI el registro recupera sus valores de la
                --pila mediante una lectura de memoria y los datos le llegan por el bus
                --de datos.
                aux_carryout        <= bus_datos(0);
                aux_zero            <= bus_datos(1);
                aux_negativo        <= bus_datos(2);
                aux_interrup        <= bus_datos(3);
                aux_carryint        <= bus_datos(4);
        ELSE
                --Si se trata del resto de instrucciones los valores son los que calcula la
                --ALU.
                aux_carryout        <= aux_cout;
                aux_zero            <= aux_z;
                aux_negativo        <= aux_n;
                aux_interrup        <= aux_interr;
                aux_carryint        <= aux_cint;
        END IF;
END PROCESS;

--Gestión de las interrupciones.
--Cuando se produce una interrupción se active una señal si esta puede tener lugar. Está
--señal se desacitva si la interrupción ha sido atendida.
PROCESS(reset,clk)
BEGIN
        IF reset='0' THEN
                aux_INT_IRQ         <= '0';
                aux_INT_TIM         <= '0';
                aux_INT_SCI         <= '0';
                aux_INT_SPI         <= '0';
```

```
                aux_INT_ADC         <= '0';
                aux_INT_PWM         <= '0';
        ELSIF clk'EVENT AND clk='1' THEN
                IF IRQ='0' THEN
                        aux_INT_IRQ         <= '1';
                ELSIF (interrup_TIM='0' AND reg_status(3)='0' AND
                        aux_ent_dec/="10001110" AND interrupcion='0') THEN
                        aux_INT_TIM         <= '1';
                ELSIF (interrup_SCI='0' AND reg_status(3)='0' AND
                        aux_ent_dec/="10001110" AND interrupcion='0') THEN
                        aux_INT_SCI         <= '1';
                ELSIF (interrup_SPI='0' AND reg_status(3)='0' AND
                        aux_ent_dec/="10001110" AND interrupcion='0') THEN
                        aux_INT_SPI         <= '1';
                ELSIF (interrup_ADC='0' AND reg_status(3)='0' AND
                        aux_ent_dec/="10001110" AND interrupcion='0') THEN
                         aux_INT_ADC        <= '1';
                ELSIF (interrup_PWM='0' AND reg_status(3)='0' AND
                        aux_ent_dec/="10001110" AND interrupcion='0') THEN
                        aux_INT_PWM         <= '1';
                ELSIF bus_direccion="1111111111011" THEN
                        aux_INT_IRQ         <= '0';
                ELSIF bus_direccion="1111111111001" THEN
                        aux_INT_TIM         <= '0';
                ELSIF bus_direccion="1111111110111" THEN
                        aux_INT_SCI         <= '0';
                ELSIF bus_direccion="1111111110101" THEN
                        aux_INT_SPI         <= '0';
                ELSIF bus_direccion="1111111110011" THEN
                        aux_INT_ADC         <= '0';
                ELSIF bus_direccion="1111111110001" THEN
                        aux_INT_PWM         <= '0';
                END IF;
        END IF;
END PROCESS;

--Lógica que indica cuando se produce una interrupción.
interrupcion <= ((aux_INT_IRQ AND (NOT(reg_status(3)))) OR aux_INT_TIM OR
                aux_INT_SCI OR aux_INT_SPI OR aux_INT_ADC OR
                aux_INT_PWM);

--Gestión de la entrada al registro de instrucción.
PROCESS(aux_carga_ri,interrupcion,bus_datos,aux_ent_dec)
BEGIN
        IF (aux_carga_ri='1' AND interrupcion='1' AND aux_ent_dec/="10000011"
        AND aux_ent_dec/="10001110" AND aux_ent_dec/="10001111" AND
        aux_ent_dec/="10000010")THEN
                --Si se ha producido una interrupción el registro de instrucción se
                --carga con el valor $82. Este valor es un código de operación no
                --utilizado que hace que el proceso de atención a interrupción se
                --ejecute como si se tratara de una instrucción más.
                aux_ent_ri <= "10000010";
        ELSE
                --Sino con el dato del bus de datos que es la instrucción que se ha
                --leído de la memoria.
                aux_ent_ri          <=  bus_datos;
        END IF;
END PROCESS;

--Gestión de las señales que paran la CPU y el oscilador en los modos de bajo consumo
--(instrucciones WAIT y STOP).
PROCESS(reset,clk)
BEGIN
        IF reset='0' THEN
                aux_stop_osc        <= '0';
                aux_stop_cpu        <= '0';
        ELSIF clk'EVENT AND clk='1' THEN
                IF (aux_ent_dec="10001111" AND reg_status(3)='0' AND
                    interrupcion='0') THEN
                        aux_stop_cpu        <= '1';
                        aux_stop_osc        <= '0';
                ELSIF (aux_ent_dec="10001110" AND reg_status(3)='0' AND
                        interrupcion='0') THEN
                        aux_stop_osc        <= '1';
                        aux_stop_cpu        <= '1';
                ELSE
                        aux_stop_osc        <= '0';
                        aux_stop_cpu        <= '0';
                END IF;
        END IF;
END PROCESS;

--Lógica combinacional que hace que se pare el reloj del núcleo y el reloj de los periféricos.
sclk                <= clk AND (NOT(aux_stop_cpu));
clk_perifericos     <= clk AND (NOT(aux_stop_osc));

--Asignación de señales para la verificación del núcleo. Estas no serían necesarias.
sal_ri              <= aux_ent_dec;
carga_mar           <= aux_carga_mar;
sal_ccr             <= reg_status;

END unidad_con;
```

--Fichero **contador.vhd**--

```
--CONTADOR.
--Este componente sólo se utiliza en la verificación del núcleo para reducir la frecuenca del reloj que
--proporciona la placa de desarrollo.
--La frecuencia para la prueba en placa se ha reducido bastante para poder ir comprobando paso a paso
--el valor de los diferentes registros.
--La señal de salida de este contador va a ser la señal de reloj que le llega al diseño del núcleo.

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY contador IS
  PORT(   clock      :IN       STD_LOGIC;
          reset      :IN       STD_LOGIC;
          clock_10   :OUT      STD_LOGIC);
END contador;

ARCHITECTURE cuenta OF contador IS
   SIGNAL aux_cuenta   :INTEGER RANGE 0 TO 9999999;
   SIGNAL aux_clock    :STD_LOGIC;
BEGIN

  PROCESS(reset,clock)
  BEGIN
    IF reset='0' THEN
          aux_cuenta <= 0;
          aux_clock  <= '0';
    ELSIF clock'EVENT AND clock='1' THEN
          IF aux_cuenta/=9999999 THEN
                --Si no se ha llegado al final de la cuenta se sigue contando.
                aux_cuenta <= aux_cuenta + 1;
          ELSE
                --Si se ha llegado al final de la cuenta se reinicia el contador.
                --Se invierte el valor de la señal de salida para formar la señal de reloj.
                aux_clock   <= NOT(aux_clock);
                aux_cuenta  <= 0;
          END IF;
          clock_10   <= aux_clock;
    END IF;
  END PROCESS;

END cuenta;
```

--Fichero **nucleo.vhd**--

```
--DISEÑO COMPLETO PARA LA VERIFICACIÓN.
--Este componente engloba a todos los componentes que forman el diseño entero del núcleo junto los
--que se utilizan para su verificación como el contador que reduce la frecuencia y un componente
--proporcionado por Xilinx que actúa de memoria y es donde se encuentra el programa a ejecutar.

LIBRARY ieee;
USE ieee.std_logic_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY nucleo IS
  PORT(   clk                :IN       STD_LOGIC;
          reset              :IN       STD_LOGIC;
          en_b               :IN       STD_LOGIC;
          interrup_SPI       :IN       STD_LOGIC;
          interrup_SCI       :IN       STD_LOGIC;
          interrup_ADC       :IN       STD_LOGIC;
          interrup_TIM       :IN       STD_LOGIC;
          interrup_PWM       :IN       STD_LOGIC;
          IRQ                :IN       STD_LOGIC;
          clk_perifericos    :OUT      STD_LOGIC;
          salidas_led        :OUT      STD_LOGIC_VECTOR(7 DOWNTO 0);
          eleccion           :IN       STD_LOGIC_VECTOR(1 DOWNTO 0);
          sal_ena            :OUT      STD_LOGIC;
          sal_rwa            :OUT      STD_LOGIC);
END nucleo;

ARCHITECTURE ar_nucleo OF nucleo IS

          --Se declaran los componentes que se van a utilizar.
          --Componente proporcionado por Xilinx. Es una memoria de doble puerto.
          COMPONENT RAMB4_S8_S8
                generic (   INIT_00   : bit_vector;
                            INIT_01   : bit_vector;
                            INIT_02   : bit_vector;
                            INIT_03   : bit_vector;
                            INIT_04   : bit_vector;
                            INIT_05   : bit_vector;
                            INIT_06   : bit_vector;
                            INIT_07   : bit_vector;
                            INIT_08   : bit_vector;
                            INIT_09   : bit_vector;
                            INIT_0A   : bit_vector;
                            INIT_0B   : bit_vector;
                            INIT_0C   : bit_vector;
```

```
                INIT_0D  : bit_vector;
                INIT_0E  : bit_vector;
                INIT_0F  : bit_vector );
        port (  DOA      : out    STD_LOGIC_VECTOR (7 downto 0);
                DOB      : out    STD_LOGIC_VECTOR (7 downto 0);
                ADDRA    : in     STD_LOGIC_VECTOR (8 downto 0);
                ADDRB    : in     STD_LOGIC_VECTOR (8 downto 0);
                CLKA     : in     STD_ULOGIC;
                CLKB     : in     STD_ULOGIC;
                DIA      : in     STD_LOGIC_VECTOR (7 downto 0);
                DIB      : in     STD_LOGIC_VECTOR (7 downto 0);
                ENA      : in     STD_ULOGIC;
                ENB      : in     STD_ULOGIC;
                RSTA     : in     STD_ULOGIC;
                RSTB     : in     STD_ULOGIC;
                WEA      : in     STD_ULOGIC;
                WEB      : in     STD_ULOGIC);
END COMPONENT;

--Núcleo del microcontrolador.
COMPONENT  unicon
  PORT(  clk              :IN     STD_LOGIC;
         reset            :IN     STD_LOGIC;
         interrup_SPI     :IN     STD_LOGIC;
         interrup_SCI     :IN     STD_LOGIC;
         interrup_ADC     :IN     STD_LOGIC;
         interrup_TIM     :IN     STD_LOGIC;
         interrup_PWM     :IN     STD_LOGIC;
         IRQ              :IN     STD_LOGIC;
         mbr_ent          :IN     STD_LOGIC_VECTOR(7 DOWNTO 0);
         mbr_sal          :OUT    STD_LOGIC_VECTOR(7 DOWNTO 0);
         clk_perifericos  :OUT    STD_LOGIC;
         mar_sal          :OUT    STD_LOGIC_VECTOR(12 DOWNTO 0);
         read_write       :OUT    STD_LOGIC;
         oe_memo          :OUT    STD_LOGIC;
         sal_ri           :OUT    STD_LOGIC_VECTOR(7 DOWNTO 0);
         carga_mar        :OUT    STD_LOGIC;
         sal_ccr          :OUT    STD_LOGIC_VECTOR(4 DOWNTO 0));
END COMPONENT;

--Contador.
COMPONENT contador
  PORT(  clock            :IN     STD_LOGIC;
         reset            :IN     STD_LOGIC;
         clock_10         :OUT    STD_LOGIC);
END COMPONENT;
```

```
        --Se declaran las señales auxiliares que se van a utilizar.
        SIGNAL aux_clock_10          :STD_LOGIC;
        SIGNAL aux_mar_sal           :STD_LOGIC_VECTOR(12 DOWNTO 0);
        SIGNAL aux_read_write        :STD_LOGIC;
        SIGNAL saux_read_write       :STD_LOGIC;
        SIGNAL aux_oe_memo           :STD_LOGIC;
        SIGNAL aux_salidas_led       :STD_LOGIC_VECTOR(7 DOWNTO 0);
        SIGNAL aux_mbr_ent           :STD_LOGIC_VECTOR(7 DOWNTO 0);
        SIGNAL aux_mbr_sal           :STD_LOGIC_VECTOR(7 DOWNTO 0);
        CONSTANT dato_pb             :STD_LOGIC_VECTOR(7 DOWNTO 0):="00000000";
        CONSTANT lectura             :STD_LOGIC:='0';
        CONSTANT dir_pc              :STD_LOGIC_VECTOR(8 DOWNTO 0):="000101111";
        CONSTANT pb_oe_memo          :STD_LOGIC:='1';
        SIGNAL aux_sal_ri            :STD_LOGIC_VECTOR(7 DOWNTO 0);
        SIGNAL aux_carga_mar         :STD_LOGIC;
        SIGNAL aux_sal_ccr           :STD_LOGIC_VECTOR(4 DOWNTO 0);
        SIGNAL neg_aux_sal_ri        :STD_LOGIC_VECTOR(7 DOWNTO 0);
        SIGNAL neg_aux_salidas_led   :STD_LOGIC_VECTOR(7 DOWNTO 0);
        SIGNAL neg_aux_sal_mar       :STD_LOGIC_VECTOR(8 DOWNTO 0);
        SIGNAL neg_aux_sal_ccr       :STD_LOGIC_VECTOR(4 DOWNTO 0);
BEGIN
        --Se invierte el valor de la señal de lectura/escitura en memoria ya que el componente de
        --memoria de Xilinx funciona al contrario de como se había diseñado.
        saux_read_write      <= NOT(aux_read_write);

        --Se invierte el valor de todas las señales que van al display y los diodos LED de la placa ya
        --que estos se activan a nivel bajo.
        neg_aux_sal_ri       <= NOT(aux_sal_ri);
        neg_aux_sal_mar      <= NOT(aux_mar_sal(8 DOWNTO 0));
        neg_aux_salidas_led  <= NOT(aux_salidas_led);
        neg_aux_sal_ccr      <= NOT(aux_sal_ccr);

        --Este proceso crea un multiplexor de manera que el display y alguno de los diodos puedan
        --mostrar varias señales de salida.
        PROCESS(neg_aux_sal_ccr,eleccion,neg_aux_sal_ri,neg_aux_sal_mar,neg_aux_salidas_led)
        BEGIN
                CASE eleccion IS
                    WHEN "00"       =>  salidas_led(4 DOWNTO 0) <= neg_aux_sal_ccr;
                                        salidas_led(7 DOWNTO 5) <= "111";
                    WHEN "01"       =>  salidas_led <= neg_aux_sal_ri;
                    WHEN "10"       =>  salidas_led <= neg_aux_sal_mar(7 DOWNTO 0);
                    WHEN OTHERS =>  salidas_led <= neg_aux_salidas_led;
                END CASE;
        END PROCESS;
```

```
--Se instancian todos los componentes.
--En la memoria se escriben los códigos de operación del programa a ejecutar. A las
--posiciones vacías se les ha dado el valor $91 que no equivale a ningún código de
--operación.
RAMB4_S8_S8_INSTANCE_NAME : RAMB4_S8_S8
generic map (
INIT_00 =>
X"9191919191919191919191919191919191919D2FBF2FB79D9D8F30A696AE9D",
INIT_01 =>
X"91919191919191919191919191FFB7FFB691919191919191919130BD2FB712A69C9A",
INIT_02 =>
X"9191919191919191919191919191919181919191919191919191919191802FB714A6",
INIT_03 =>
X"9191919191919191919191919191919191919191919191919191919191802FB716A6",
INIT_04 =>
X"9191919191919191919191919191919191919191919191919191919191802FB718A6",
INIT_05 =>
X"9191919191919191919191919191919191919191919191919191919191802FB71AA6",
INIT_06 =>
X"9191919191919191919191919191919191919191919191919191919191802FB71CA6",
INIT_07 =>
X"9191919191919191919191919191919191919191919191919191919191802FB71EA6",
INIT_08 =>
X"9191919191919191919191919191919191919191919191919191919191919191181",
INIT_09 =>
X"3091919191919191919191919191918191919191919191919191919191915691",
INIT_0A =>
X"9191919191919191919191919191919191919191919191919191919191919191191",
INIT_0B =>
X"9191919191919191919191919191919191919191919191919191919191919191191",
INIT_0C =>
X"9191919191919191919191919191919191919191919191919191919191919191191",
INIT_0D =>
X"9191919191919191919191919191919191919191919191919191919191919191191",
INIT_0E =>
X"9191919191919191919191919191919191919191919191919191919191919191191",
INIT_0F =>
X"0000200040006000800 0A000C000E000919191919191919191919191919191191")

port map ( DOA          => aux_mbr_ent,
           DOB          => aux_salidas_led,
           ADDRA        => aux_mar_sal(8 DOWNTO 0),
           ADDRB        => dir_pc,
           CLKA         => aux_clock_10,
           CLKB         => aux_clock_10,
           DIA          => aux_mbr_sal,
           DIB          => dato_pb,
           ENA          => aux_oe_memo,
           ENB          => en_b,
           RSTA         => saux_read_write ,
           RSTB         => lectura,
           WEA          => saux_read_write,
           WEB          => lectura);

uni: unicon     PORT MAP (  clk             =>  aux_clock_10,
                            reset           =>  reset,
                            interrup_SPI    =>  interrup_SPI,
                            interrup_SCI    =>  interrup_SCI,
                            interrup_ADC    =>  interrup_ADC,
                            interrup_TIM    =>  interrup_TIM,
                            interrup_PWM    =>  interrup_PWM,
                            IRQ             =>  IRQ,
                            mbr_ent         =>  aux_mbr_ent,
                            mbr_sal         =>  aux_mbr_sal,
                            clk_perifericos =>  clk_perifericos,
                            mar_sal         =>  aux_mar_sal,
                            read_write      =>  aux_read_write,
                            oe_memo         =>  aux_oe_memo,
                            sal_ri          =>  aux_sal_ri,
                            carga_mar       =>  aux_carga_mar,
                            sal_ccr         =>  aux_sal_ccr);
ccont: contador   PORT MAP (  clock         =>  clk,
                            reset           =>  reset,
                            clock_10        =>  aux_clock_10);

--Señales de salida que se desea verificar su valor. Se invierten su valor debido a que los
--diodos se activan a nivel bajo.
sal_ena     <= NOT(aux_oe_memo);
sal_rwa   <= NOT(aux_carga_mar);

END ar_nucleo;
```

--Fichero **tb_nucleo.vhd**--

```
--BANCO DE PRUEBAS DE LA UNIDAD DE CONTROL.
--Este componente es necesario para poder realizar la simulación del diseño. En el se dan valores a las
--señales de entrada en el tiempo para en la simulación poder observar los valores que tomen  las
--señales de salida en cada momento.

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY tb_nucleo IS
END tb_nucleo;

ARCHITECTURE banco_nucleo OF tb_nucleo IS

        COMPONENT nucleo
          PORT(   clk                 :IN     STD_LOGIC;
                  reset               :IN     STD_LOGIC;
                  en_b                :IN     STD_LOGIC;
                  interrup_SPI        :IN     STD_LOGIC;
                  interrup_SCI        :IN     STD_LOGIC;
                  interrup_ADC        :IN     STD_LOGIC;
                  interrup_TIM        :IN     STD_LOGIC;
                  interrup_PWM        :IN     STD_LOGIC;
                  IRQ                 :IN     STD_LOGIC;
                  eleccion            :IN     STD_LOGIC_VECTOR(1 DOWNTO 0);
                  clk_perifericos     :OUT    STD_LOGIC;
                  salidas_led         :OUT    STD_LOGIC_VECTOR(7 DOWNTO 0);
                  sal_ena             :OUT    STD_LOGIC;
                  sal_rwa             :OUT    STD_LOGIC);
        END COMPONENT;
        SIGNAL sclk                             : STD_LOGIC:='0';
        SIGNAL sreset                           : STD_LOGIC:='0';
        SIGNAL sen_b                            : STD_LOGIC;
        SIGNAL sIRQ                             : STD_LOGIC;
        SIGNAL sTIM                             : STD_LOGIC;
        SIGNAL sSPI                             : STD_LOGIC;
        SIGNAL sSCI                             : STD_LOGIC;
        SIGNAL sADC                             : STD_LOGIC;
        SIGNAL sPWM                             : STD_LOGIC;
        SIGNAL sclk_perifericos         : STD_LOGIC;
        SIGNAL ssalidas_led             : STD_LOGIC_VECTOR(7 DOWNTO 0);
        SIGNAL ssal_ena                 : STD_LOGIC;
        SIGNAL ssal_rwa                 : STD_LOGIC;
        SIGNAL seleccion                : STD_LOGIC_VECTOR(1 DOWNTO 0);
BEGIN
        S0:nucleo PORT MAP(     clk             => sclk,
                                reset           => sreset,
                                en_b            => sen_b,
                                IRQ             => sIRQ,
                                interrup_TIM    => sTIM,
                                interrup_SPI    => sSPI,
                                interrup_SCI    => sSCI,
                                interrup_ADC    => sADC,
                                interrup_PWM    => sPWM,
                                eleccion        => seleccion,
                                clk_perifericos => sclk_perifericos,
                                salidas_led     => ssalidas_led,
                                sal_ena         => ssal_ena,
                                sal_rwa         => ssal_rwa);

        --Se dan valores a la señales de entrada.
        sclk            <= NOT(sclk) AFTER 20 NS;
        sreset          <= '0','1' AFTER 150 NS;
        sen_b           <= '1';
        seleccion       <= "11";
        sIRQ            <= '1','0' AFTER 250000 NS,'1' AFTER 260000 NS;
        sTIM            <= '1';
        sSPI            <= '1';
        sSCI            <= '1';
        sADC            <= '1';
        sPWM            <= '1';

END banco_nucleo;
```