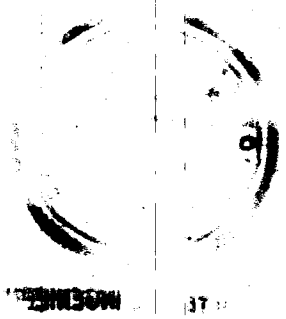
UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

DISEÑO DE MICROPROCESADORES

DR. JESUS SAVAGE

ING. GABRIEL VÁZQUEZ.



DISEÑO DE MICROPROCESADORES

Jesús Savage
Facultad de Ingeniería.
Universidad Nacional Autónoma de México, UNAM.

Gabriel Vázquez
Facultad de Ingeniería.
Universidad Nacional Autónoma de México, UNAM.

PRESENTACIÓN

La Facultad de Ingeniería ha decidido realizar una serie de ediciones provisionales de obras recientemente elaboradas por académicos de la institución, como material de apoyo para sus clases, de manera que puedan ser aprovechadas de inmediato por alumnos y profesores. Tal es el caso de la obra *Diseño de microprocesadores,* elaborada por los profesores Jesús Savage Carmona y Gabriel Vázquez.

Se invita a los estudiantes y profesores a que comuniquen a los autores las observaciones y sugerencias que mejoren el contenido de la obra, con el fin de que se incorporen en una futura edición definitiva.

PREFACIO

Existe una gran variedad de textos que tratan temas referentes al diseño de microprocesadores. Desafortunadamente, para las personas de habla española, la mayoría de esos textos están en otros idiomas, o bien, las traducciones que se hacen de esos textos suelen modificar el significado o la idea que el autor deseaba expresar. Teniendo en cuenta estos puntos, intentamos escribir este libro lo más claro y sencillo posible, y además, en español. No obstante, en varias ocasiones se tuvieron que recurrir a términos en inglés debido a que no se encontró una traducción apropiada al español, o porque la traducción restaba significado al concepto.

La siguiente obra presenta las bases para el diseño de un microprocesador, particularmente, un 'clón' del microprocesador M68HC11 de Motorola®. Adicionalmente, gracias al desarrollo en los últimos años de los dispositivos lógicos programables y de los lenguajes de descripción de hardware de alto nivel, se logró vincular la teoría con la práctica, de manera que el diseño del microprocesador propuesto es implantado en un PLD, empleando uno de los lenguajes de descripción de hardware más utilizados en la actualidad por los países líderes en el área: Verilog HDL.

El antecedente para poder utilizar estas notas es un curso básico de diseño digital en donde se den a conocer las bases de la lógica booleana, la lógica secuencial y del diseño de dispositivos digitales básicos.

Finalmente, los autores desean agradecer a todas aquellas personas que colaboraron en la escritura y corrección del manuscrito, así como a aquellos que nos apoyaron en la recopilación de información para esta obra.

Los Autores.

CONTENIDO

CAPÍTULO 1

INTRODUCCIÓN 1

1.1 ESTRUCTURA BÁSICA DE UNA COMPUTADORA 2

1.2 TIPOS DE COMPUTADORAS 2
- 1.2.1 Computadoras secuenciales 2
- 1.2.2 Computadoras paralelas 3

1.3 SEGMENTACIÓN ENCAUZADA 4

1.4 EL CONTROLADOR DE LA COMPUTADORA 5

1.5 COMPUTADORAS PARALELAS 6
- 1.5.1 Arquitecturas SIMD 6
- 1.5.2 Arquitecturas MIMD 7

CAPÍTULO 2

MÁQUINAS DE ESTADOS Y SU CONSTRUCCIÓN 11

2.1 MÁQUINAS DE ESTADOS 12
- 2.1.1 El algoritmo de la máquina de estados 12
- 2.1.2 Estados y reloj 12

2.2 NOTACIÓN DE LA CARTA ASM 13
- 2.2.1 Representación de estados 13
- 2.2.2 Representación de decisiones 13
- 2.2.3 Representación de salidas 14

2.3 EJEMPLOS DE CARTAS ASM 14

2.4 CONSTRUCCIÓN DE MÁQUINAS DE ESTADOS USANDO LOS MÉTODOS TRADICIONALES 26
- 2.4.1 Circuito secuencial 26
- 2.4.2 Unidad básica de almacenamiento 26
- 2.4.3 Latch tipo D 27
- 2.4.4 Flip-flop tipo D 28
- 2.4.5 Procedimiento para el diseño de circuitos secuenciales 29
- 2.4.6 Ejemplo 30

2.5 DISEÑO DIGITAL USANDO DISPOSITIVOS LÓGICOS PROGRAMABLES 33

CAPÍTULO 3

CONSTRUCCIÓN DE MÁQUINAS DE ESTADOS USANDO MEMORIAS 40

3.1 DIRECCIONAMIENTO POR TRAYECTORIA 41

3.2 DIRECCIONAMIENTO ENTRADA-ESTADO 43

3.3 DIRECCIONAMIENTO IMPLÍCITO 47

CAPÍTULO 4

CONSTRUCCIÓN DE MÁQUINAS DE ESTADOS USANDO SECUENCIADORES 53

4.1 EL SECUENCIADOR BÁSICO 54

4.2 INSTRUCCIONES PARA EL SECUENCIADOR 55
4.2.1 Continúa 55
4.2.2 Salto condicional 56
4.2.3 Salto de transformación 56
4.2.4 Salto condicional usando la dirección de las interrupciones 57

4.3 SECUENCIADORES Y MEMORIAS 58

4.4 IMPLANTACIÓN DE CARTAS ASM USANDO SECUENCIADORES 59

CAPÍTULO 5

COMPONENTES BÁSICOS DE UN PROCESADOR 64

5.1 ESTRUCTURA BÁSICA DE UNA COMPUTADORA 65

5.2 UNIDAD DE CONTROL DE LA COMPUTADORA 66

5.3 UNIDAD DE PROCESOS ARITMÉTICOS 67

5.4 REGISTROS INTERNOS 71
5.4.1 Registros acumuladores 71
5.4.2 El algoritmo de multiplicación 72
5.4.3 Registros contadores de 16 bits 76

5.5 UNIDAD DE CONTROL DE PROGRAMA 77

5.6 REGISTRO DE ESTADOS O BANDERAS 78

5.7 UNIDAD DE CONTROL DE INTERRUPCIONES 80

CAPÍTULO

6 DISEÑO DE UN MICROPROCESADOR DE 8 BITS 83

6.1 ARQUITECTURA DEL MICROPROCESADOR 68HC11 84

6.2 TIPOS DE INSTRUCCIONES
6.2.1 Acceso inmediato 87
6.2.2 Acceso extendido 87
6.2.3 Acceso directo 87
6.2.4 Acceso indexado 88
6.2.5 Acceso relativo 88
6.2.6 Acceso inherente 88
89

6.3 MICROPROGRAMACIÓN
6.3.1 Instrucción INX (acceso inherente) 89
6.3.2 Instrucción INY (acceso inherente) 90
6.3.3 Instrucción XGDX (acceso inherente) 102
6.3.4 Instrucción LDAB (acceso inmediato) 105
6.3.5 Instrucción LDAA (acceso inmediato) 106
6.3.6 Instrucción SUBA (acceso extendido) 107
6.3.7 Instrucción BRA (acceso relativo) 108
6.3.8 Instrucción BEQ (acceso relativo) 110
6.3.9 Instrucción JSR (acceso extendido) 114
6.3.10 Instrucción RTS (acceso inherente) 116
6.3.11 Atención a interrupciones 118
6.3.12 Instrucción RTI (regreso de interrupción) 120
123

CAPÍTULO 7

SEGMENTACIÓN ENCAUZADA (PIPELINE) 128

7.1 INTRODUCCIÓN 129

7.2 LA ARQUITECTURA SEGMENTADA DEL 68HC11
7.2.1 Etapa 1 - Lectura de la instrucción 130
7.2.2 Etapa 2 - Decodificación de la instrucción / Cálculo de la dirección efectiva / Lectura de operandos 131
7.2.3 Etapa 3 - Ejecución / Cálculo de banderas y saltos 134
7.2.4 Etapa 4 - Post-Escritura 138
7.2.5 Representación gráfica de la segmentación encauzada 141
143

7.3 CONJUNTO DE INSTRUCCIONES
7.3.1 Instrucción LDAA (acceso inmediato) 144
7.3.2 Instrucción ABA (acceso inherente) 144
7.3.3 Instrucción ANDB (acceso extendido) 151
7.3.4 Instrucción ASL (acceso indexado) 158
7.3.5 Instrucción STAA (acceso extendido) 164
7.3.6 Instrucción BRA (acceso relativo) 171
7.3.7 Resumen de instrucciones 178
7.3.8 Ejecución de múltiples instrucciones 185
186

7.4 RIESGOS POR DEPENDENCIAS DE DATOS 194

7.5 CONTROL DE RIESGOS POR DEPENDENCIAS DE DATOS 196
7.5.1 Control de riesgos por dependencias de datos por medio de software 197
7.5.2 Control de riesgos por dependencias de datos por medio de detenciones 198
7.5.3 Control de riesgos por dependencias de datos por medio de anticipaciones 205

7.6 RIESGOS POR SALTOS 215
7.6.1 Detenciones 219
7.6.2 Suponer que el salto no es realizado 220

7.7 INTERRUPCIONES 225

APÉNDICE A

USO DE VERILOG HDL EN EL ENTORNO MAX+PLUS II 235

A.1 INICIANDO MAX+PLUS II 236

A.2 USANDO EL EDITOR DE TEXTO DE MAX+PLUS II 238

A.3 USANDO EL EDITOR GRÁFICO DE MAX+PLUS II 244

A.4 EL LENGUAJE DE DESCRIPCIÓN DE HARDWARE VERILOG 247
A.4.1 Módulos 247
A.4.2 Puertos 248
A.4.3 Cometarios 248
A.4.4 Bloques procedurales 248
A.4.5 Eventos 249
A.4.6 Sentencias de control de programa 249
A.4.7 Tipos de datos 252
A.4.8 Formato de números enteros 254
A.4.9 Tipos de operadores 255

A.5 LÓGICA COMBINACIONAL EN VERILOG HDL 260
A.5.1 Asignaciones continuas 260
A.5.2 Construcciones always 260
A.5.3 Construcción de multiplexores 261

A.6 LÓGICA SECUENCIAL EN VERILOG HDL 262
A.6.1 Construcción de registros 262
A.6.2 Construcción de contadores 263
A.6.3 Construcción de latches 264
A.6.4 Construcción de máquinas de estados 264

A.7 PROYECTOS JERÁRQUICOS 266
A.7.1 Cómo usar funciones lógicas de MAX+PLUS II en Verilog 266
A.7.2 Cómo usar instancias de compuertas lógicas en Verilog 267
A.7.3 Cómo usar funciones de usuario en Verilog 268

A.7.4 Cómo usar funciones parametrizadas en Verilog 269
A.7.5 Creación de memorias ROM y RAM en Verilog 270

APÉNDICE B

CONSTRUCCIÓN DE UN MICROPROCESADOR UTILIZANDO VERILOG HDL Y AHDL 271

B.1 NOTA IMPORTANTE 272

B.2 ETAPA 1: CAMINO DE DATOS 272
B.2.1 Registros acumuladores 272
B.2.2 Registro contador de 16 bits 277
B.2.3 Registro de estados o de banderas 282
B.2.4 Unidad de procesos aritméticos 287
B.2.5 Módulo arquitectura_wire_Verilog 300

B.3 ETAPA 2: CONTROL 302
B.3.1 Unidad de control de interrupciones 302
B.3.2 Unidad de control de la computadora 304
B.3.2.1 Registro de instrucción 306
B.3.2.2 Secuenciador 307
B.3.2.3 Lógica de selección 309
B.3.2.4 Memoria de microprograma 313

B.4 ETAPA 3: UN MICROPROCESADOR DE 8 BITS 320
B.4.1 Module_wire_Verilog 320
B.4.2 La memoria externa 322
B.4.3 Microprocesador de 8 bits 324

APÉNDICE C

INTRODUCCIÓN A LOS DISPOSITIVOS LÓGICOS PROGRAMABLES VLSI 326

C.1 INTRODUCCIÓN 327

C.2 EL CPLD MAX7000S DE ALTERA 327

C.3 EL PLD FLEX10K DE ALTERA 330
C.3.1 Embedded array block (EAB) 332
C.3.2 Logic array block (LAB) 333
C.3.3 Logic element (LE) 333
C.3.4 Input/Output element (IOE) 335

C.4 FPGA XC4000 DE XILINX 337

CAPÍTULO I
INTRODUCCIÓN

1.1 ESTRUCTURA GENERAL DE UNA COMPUTADORA

Una computadora digital es una máquina electrónica capaz de realizar cálculos con gran rapidez, obedeciendo instrucciones muy específicas y elementales que reflejan su estructura funcional y organizacional.

Se puede definir conceptualmente a una computadora como una máquina que consta de elementos de entrada, elementos de salida, un procesador central y una memoria.

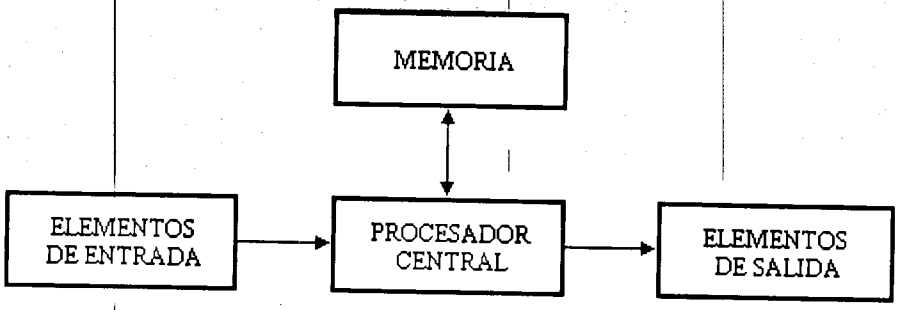


Figura 1.1. Diagrama de bloques de una computadora.

1.2 TIPOS DE COMPUTADORAS

Existen diversos criterios de clasificación para las arquitecturas de computadoras. Uno de ellos es la manera en como ejecutan sus cálculos, el cual origina dos categorías de computadoras:

1. Aquellas que realizan los cálculos de manera secuencial (a la cuál pertenecen la mayoría de las computadoras); y
2. Aquellas que realizan los procesos en paralelo.

1.2.1 COMPUTADORAS SECUENCIALES

Dentro de esta clasificación encontramos a las computadoras SISD (Single-Instruction Stream, Single-Data Stream / Flujo Único de Instrucciones, Flujo Único de Datos).

La arquitectura de von Neumann pertenece a esta clasificación que corresponde a computadoras que tienen un sólo CPU ejecutando una instrucción a la vez, además, en este tipo de computadoras sólo se puede buscar o almacenar un elemento de datos a la vez. En las computadoras de von Neumann los programas y datos se encuentran en una memoria externa.

Para ejecutar una instrucción, la computadora tiene que efectuar las siguiente etapas:

1. Traer el código de la instrucción a ejecutar.

2. Decodificar la instrucción, es decir, saber cuáles son las micro-operaciones que tiene que realizar la computadora para ejecutar dicha instrucción.
3. Traer los operandos en caso de que los requiera la instrucción.
4. Ejecutar la instrucción y guardar el resultado.

Estas etapas se ejecutan de manera secuencial para cada instrucción, por lo tanto, una nueva instrucción no puede comenzar hasta que la anterior termine.

La figura 1.2 muestra la computadora de von Neumann.

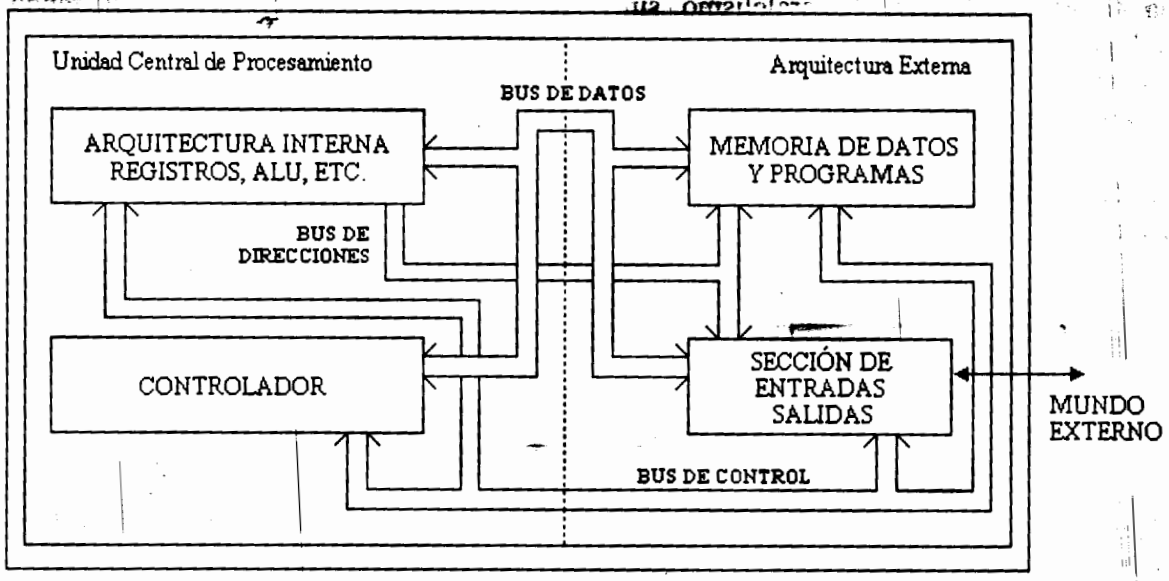


Figura 1.2. Computadora de von Neumann.

1.2.2 COMPUTADORAS PARALELAS

Una computadora que ejecuta procesos en paralelo cuenta con varios de sus componentes internos repetidos, de esta forma puede ejecutar varias instrucciones al mismo tiempo. Dentro de esta clasificación encontramos a las computadoras SIMD (Single-Instruction Stream, Multiple-Data Stream / Flujo Único de Instrucciones, Flujo Múltiple de Datos) y MIMD (Multiple-Instruction Stream, Multiple-Data stream / Flujo Múltiple de Instrucciones, Flujo Múltiple de Datos).

Las arquitecturas SIMD son esenciales en el mundo de las computadoras paralelas, debido a su habilidad para manejar grandes vectores y matrices de datos en tiempos muy cortos. El secreto detrás de este tipo de arquitectura es que cuentan con un varios procesadores ejecutando la misma operación sobre un conjunto de datos. Por ejemplo, cuando una simple instrucción SIMD suma 64 números, el hardware SIMD envía 64 flujos de datos a 64 ALU's (Arithmetic Logic Unit / Unidad Lógico Aritmética) para formar 64 sumas en un sólo ciclo de reloj. Incluso cuando el tamaño del vector es mayor al número de ALU's disponibles, la velocidad, comparada con una computadora secuencial, es inmensa.

Las arquitecturas MIMD, también llamadas máquinas multiprocesadores, tienen más de un procesador funcionando asíncrona e independientemente. Al mismo tiempo, los diferentes procesadores pueden estar ejecutando diferentes instrucciones sobre diferentes conjuntos de datos. Las arquitecturas MIMD se utilizan en aplicaciones de Diseño Asistido por Computadora, Graficación por Computadora, simulaciones en tiempo real, y en general, en aplicaciones que requieran gran poder de cómputo.

1.3 SEGMENTACIÓN ENCAUZADA

Una forma de lograr el paralelismo sin tener componentes repetidos es utilizando una metodología llamada *segmentación encauzada* (en inglés, pipeline). En la segmentación encauzada la computadora está dividida en varias etapas, de manera que cada etapa efectúa operaciones sobre instrucciones diferentes. La figura 1.3 muestra este concepto con cuatro etapas, las cuales están ligadas con registros de acoplo. La primera etapa trae la instrucción Ij; la segunda, decodifica la instrucción Ij-1; la tercera, trae los operandos de la instrucción Ij-2; y por último la cuarta, ejecuta la instrucción Ij-3.

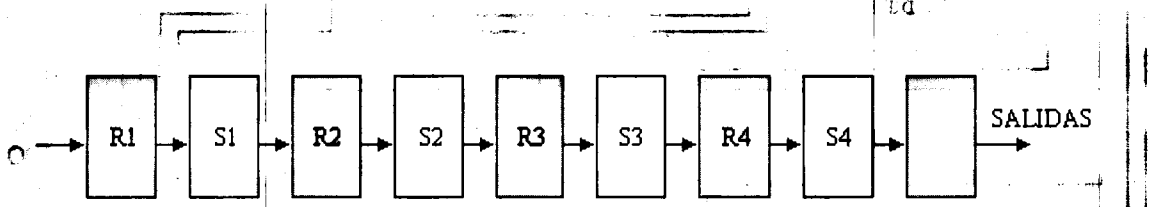


Figura 1.3. Estructura básica de un procesador segmentado.

Un procesador segmentado de k etapas es en un principio k veces más rápido que un procesador sin esta tecnología. La figura 1.4 muestra el diagrama de tiempos / tareas de un procesador segmentado de cuatro etapas. En el diagrama se observa que cada instrucción requiere 4 tiempos en ser ejecutada, sin embargo, una vez que el cauce está completo, el tiempo de salida de cada instrucción con respecto a la anterior es de periodo T.

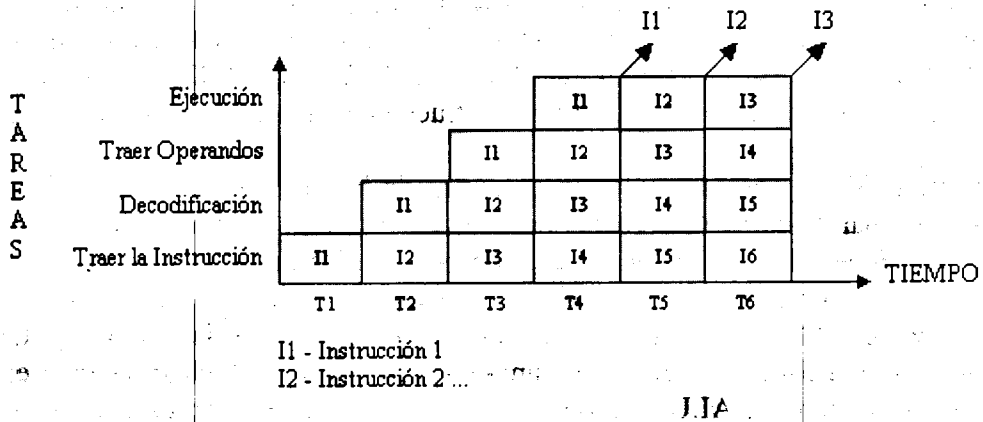


Figura 1.4. Diagrama de Tiempos / Tareas.

En este tipo de arquitecturas es común hacer lecturas y escrituras en la memoria, tanto de instrucciones de programa como de datos, de manera simultánea. Por tal motivo, es necesario que las memorias de datos y programas, con sus respectivos buses, estén separadas. La figura 1.5 muestra un tipo de esta arquitectura conocida como computadora tipo Harvard.

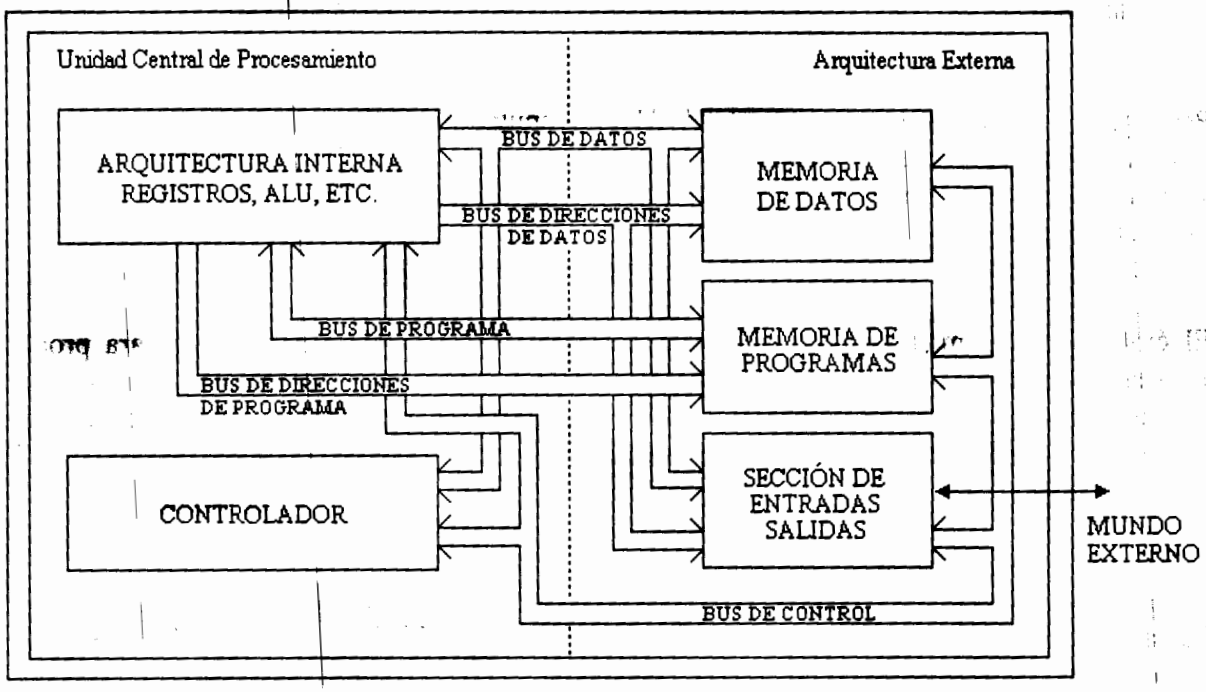


Figura 1.5. Computadora tipo Harvard.

1.4 EL CONTROLADOR DE LA COMPUTADORA

Un elemento fundamental es el controlador de la computadora, que como su nombre lo indica, controla y sincroniza todas las operaciones de la arquitectura interna y externa. Esto es, coordina las actividades de la computadora y determina qué operaciones se deben realizar y en qué orden. Este controlador general, denominado Dispositivo de Máquinas de Estados, se describe en el siguiente capítulo.

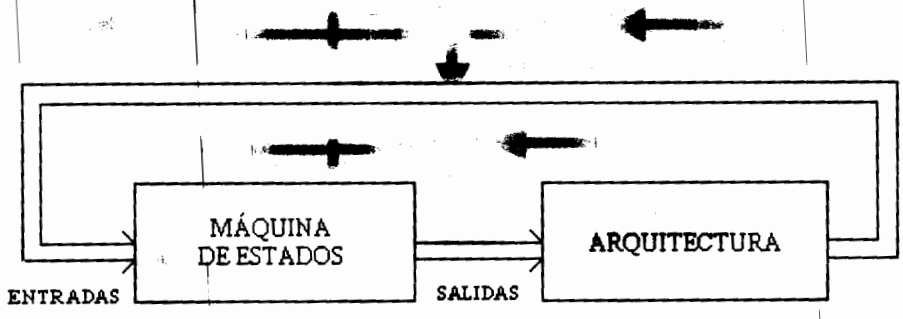


Figura 1.6. Máquina de estados controlando una arquitectura.

1.5 COMPUTADORAS PARALELAS

Como se mencionó anteriormente, dentro de la clasificación de computadoras paralelas se encuentran las arquitecturas SIMD y MIMD. El nombre que reciben estas arquitecturas proviene de la clasificación de computadoras que estableció Flynn, quien utilizó los siguientes dos criterios para crear su taxonomía: 1) el número de instrucciones que se están ejecutando simultáneamente en un momento dado, y 2) el número de datos sobre los que se están ejecutando esas instrucciones.

Pero, ¿qué es una computadora paralela?. Una computadora paralela es un tipo de computadora que posee dos o más procesadores independientes que cooperan y se comunican para solucionar un problema. El programador de este tipo de computadoras debe dividir el problema a resolver en varias partes, de manera que el trabajo total sea distribuido entre los distintos procesadores; además, debe calcular la forma en la que cada parte del trabajo se relaciona con el resto de las partes.

El éxito de las computadoras paralelas sobre las computadoras desarrolladas para propósitos especiales se debe a que las primeras ofrecen rendimientos más altos a precios más bajos. Además, la posibilidad de escalamiento es inherente en las computadoras paralelas, en teoría, estas pueden ser actualizadas cambiando o agregando más procesadores.

1.5.1 ARQUITECTURAS SIMD

Las arquitecturas SIMD (Single Instruction Multiple Data) ejecutan la misma instrucción sobre múltiples datos simultáneamente; de manera semejante a como un sargento ordena a todo el pelotón dar media vuelta, en lugar de hacerlo de soldado en soldado.

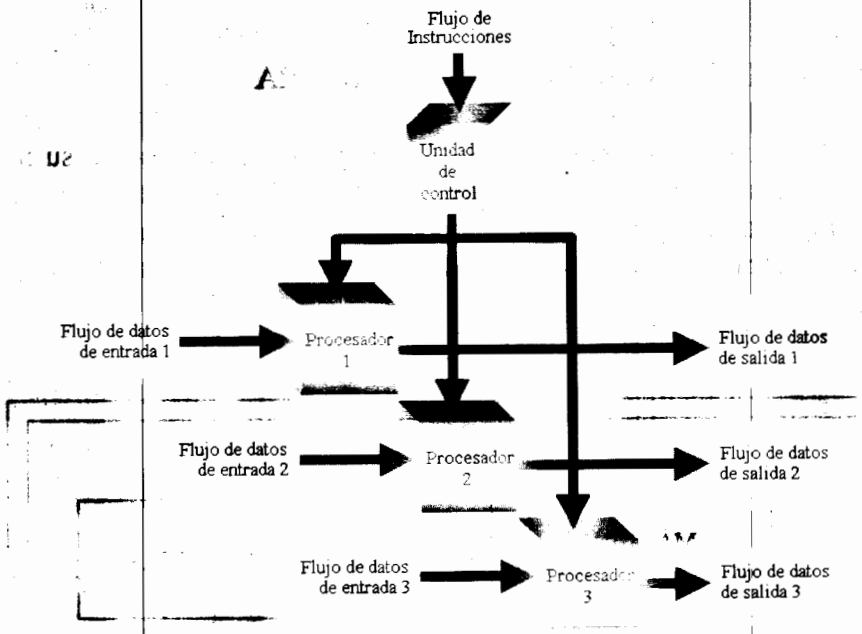


Figura 1.7. Diagrama de bloques de la arquitectura SIMD.

La característica principal de este tipo de arquitecturas es que cuentan con una sola unidad de control, quien es la responsable de interpretar y distribuir la misma instrucción a todos los procesadores. En caso de que no se desee ejecutar la misma instrucción en todos los procesadores, es posible habilitar sólo los procesadores necesarios por medio de una máscara. La figura 1.7 muestra un modelo de arquitectura SIMD de tres procesadores, cada uno operando con su propia memoria local. Todos los procesadores operan bajo el control de una sola secuencia de instrucciones emitida por una unidad de control central. Existen tres flujos de entrada de datos, uno por procesador, los cuales se operan de manera síncrona. En cada paso, todos los procesadores ejecutan la misma instrucción sobre un dato diferente.

La mayoría de las aplicaciones interesantes que utilizan este tipo de computadoras requieren que los procesadores puedan comunicarse entre sí con el fin de intercambiar datos o resultados intermedios. Esta comunicación se logra por alguno de los siguientes dos esquemas: mediante una memoria común (arquitecturas SIMD a memoria compartida) o mediante una red de interconexión (arquitecturas SIMD a memoria distribuida).

En las arquitecturas a memoria compartida todos los procesadores comparten el mismo espacio de memoria; esto significa que el conocimiento de donde se almacenan los datos no es preocupación para el usuario pues hay solamente una memoria para todos los procesadores. En las arquitecturas de memoria distribuida cada procesador tiene su propia memoria asociada. Los procesadores están conectados por medio de una red de interconexión que les permite intercambiar datos entres sus respectivas memorias cuando es necesario. En contraste con las máquinas de memoria compartida, el usuario debe estar enterado de la localización de los datos en las memorias locales y deberá mover o distribuir estos datos explícitamente cuando sea necesario.

Las principales ventajas de estas arquitecturas son las siguientes:

- Funcionan muy bien con vectores de datos.
- La eficiencia es óptima cuando se manejan arreglos de datos en ciclos for.

Entre las desventajas se encuentran las siguientes:

- Las instrucciones de salto y las condicionales no pueden ser ejecutadas en paralelo porque sólo existe una unidad de control de instrucciones.
- El rendimiento decae considerablemente en las sentencias case en un factor de 1/p, donde p es el número de bloques case.
- Durante los ciclos while, los datos en algunos procesadores pueden encontrar la condición de salida del ciclo antes de que ocurra en otros procesadores. En este caso, los procesadores que hayan completado el ciclo deben deshabilitarse hasta que el resto de los procesadores cumplan con la condición de salida.

1.5.2 ARQUITECTURAS MIMD

Esta es quizás la mejor estrategia de diseño orientada a obtener el más alto rendimiento y la mejor relación costo/rendimiento. La idea detrás de las arquitecturas MIMD (Multiple Instruction Multiple

Data) es bastante simple: aumentar la capacidad de trabajo conectando varios procesadores entre sí para obtener un rendimiento global lo más cercano a la suma de rendimientos de cada procesador por separado. La filosofía de trabajo plantea la división de un problema (programa) en varias tareas (procesos) independientes, de manera que a cada procesador se le asigna la resolución de uno de estos procesos.

Generalmente, las arquitecturas de este tipo consisten de varios procesadores autónomos que pueden ejecutar flujos independientes de instrucciones usando datos locales. La característica principal de las arquitecturas MIMD es que son computadoras asíncronas con control descentralizado de hardware, es decir, cada procesador tiene su propia unidad de control ejecutando un programa diferente.

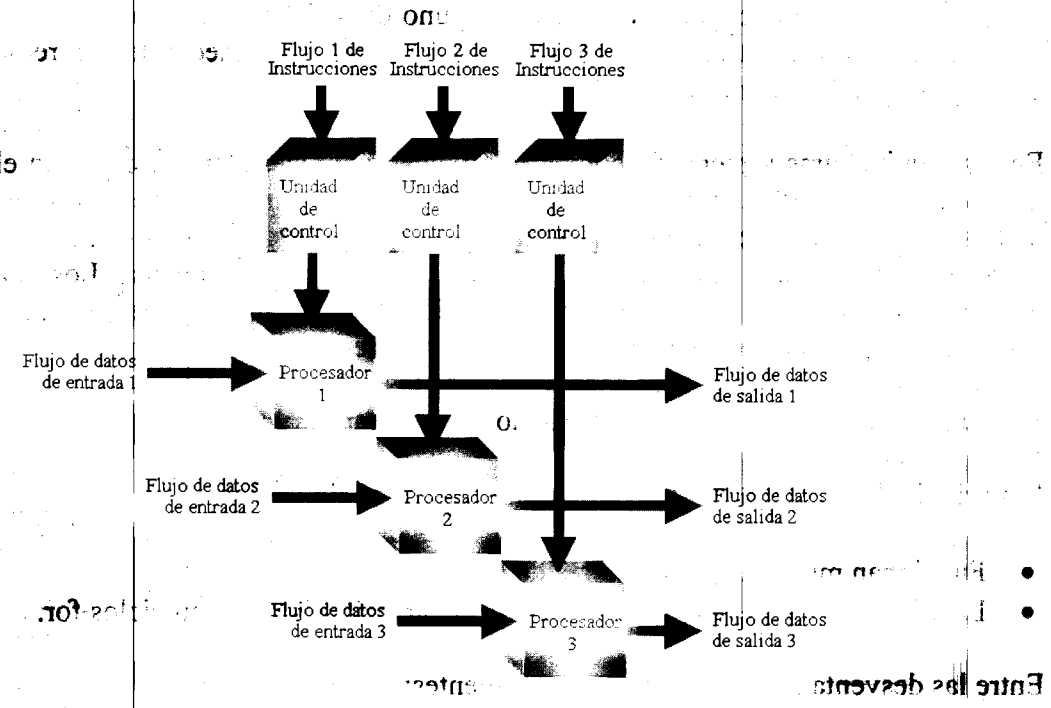


Figura 1.8. Diagrama de bloques de la arquitectura MIMD.

La comunicación entre procesadores se realiza de manera semejante a la arquitecturas SIMD, es decir, mediante una memoria común (arquitecturas MIMD a memoria compartida) o mediante una red de interconexión (arquitecturas MIMD a memoria distribuida).

En las arquitecturas de memoria compartida todos los procesadores se comunican por medio de una memoria común. Por lo tanto, para transmitir un dato desde el procesador Pi al procesador Pj son necesarios dos pasos: 1) el procesador Pi escribe el dato en una dirección de memoria conocida por el procesador Pj; y 2) el procesador Pj lee esa localidad de memoria.

En los sistemas de memoria compartida, permitir múltiples lecturas simultáneas sobre la misma posición de memoria no debe ocasionar ningún problema. Conceptualmente, si cada procesador requiere leer desde una misma posición de memoria copia el contenido de esa posición y lo almacena en su memoria local (memoria caché). Sin embargo, si varios procesadores requieren

escribir simultáneamente diferentes datos sobre la misma posición de memoria, entonces, es necesario establecer mecanismos de sincronización entre los procesadores para el acceso a zonas de memoria compartida.

La otra forma de comunicación entre procesadores es por medio de una red de interconexión. En este modelo, la memoria es dividida entre el conjunto de procesadores para su acceso local, además, cada procesador cuenta con su propia memoria caché. La red de interconexión puede ser de dos tipos: 1) directa, en las que existen enlaces físicos que conectan directamente pares de procesadores, permitiendo enviar o recibir datos en cualquier instante de tiempo; y 2) de múltiples etapas, que tienen una baja cantidad de enlaces entre procesadores, de manera que cuando es necesario comunicar un mensaje entre dos procesadores que no tienen conexión directa, debe encaminarse o enrutarse dicho mensaje por procesadores intermedios entre éstos dos.

En resumen, los MIMD han logrado una posición consolidada en el mercado y han demostrado que para una carga alta de trabajo en tiempo compartido son más eficientes que los SISD. Un programa MIMD no emplea menos tiempo de procesador, pero puede efectuar mayor número de tareas independientes por unidad de tiempo gracias a que distintos programas no comparten el mismo procesador, sino que se ejecutan en procesadores separados y totalmente dedicados.

Por el momento, el único inconveniente es que no existen muchas aplicaciones que corran satisfactoriamente aprovechando las características del paralelismo. Esto no tiene nada que ver con características del hardware, sino más bien con el escaso desarrollo de aplicaciones verdaderamente paralelas aplicables a estas arquitecturas. Finalmente, se muestra un cuadro comparativo entre las arquitecturas SIMD y MIMD en la tabla 1.1.

Arquitecturas SIMD	*Arquitecturas MIMD*
Requiere menos hardware: una unidad de control.	Requiere más hardware, cada procesador tiene su propia unidad de control.
Necesita menos memoria: una sola copia del programa.	Necesita más memoria para cada uno de los programas.
Adecuada para aplicaciones que requieran ejecutar las mismas instrucciones sobre un gran número de datos.	Puede ejecutar tareas distintas al mismo tiempo o emular un procesador SIMD mediante mecanismos de sincronización.
Necesitan menor tiempo para comunicarse con los procesadores vecinos dado que poseen un reloj global.	Para comunicarse es necesario usar mecanismos de sincronización.
Son más costosas porque requieren diseñar un microchip de arquitectura especial.	Son más baratas porque se pueden construir usando computadoras convencionales de propósito general.

Tabla 1.1. Cuadro comparativo entre las arquitecturas SIMD y MIMD.

PROBLEMAS

1. Indique cuáles fueron las principales aportaciones de las siguientes personas en el área de la computación.

   - Charles Babbage
   - Herman Hollerith
   - Alan Mathison Turing
   - Norbert Wiener
   - Claude Elwood Shannon
   - John von Neumann

2. ¿Cuál se considera la primera computadora?

3. ¿Cuál fue la razón fundamental por la que se comenzaron a utilizar los números binarios para representar la información en las computadoras?

4. ¿Qué otra aportación hizo a la sociedad Henry Ford además de haber introducido el concepto de línea de ensamble en la fabricación de automóviles?

5. ¿Qué compañía introdujo el concepto de línea de ensamble en la computación y en qué fechas?

6. Mencione brevemente las características de las arquitecturas SIMD e investigue en qué aplicaciones son utilizadas este tipo de computadoras.

7. Mencione brevemente las características de las arquitecturas MIMD e investigue en qué aplicaciones son utilizadas este tipo de computadoras.

8. Investigue las características de las arquitecturas SISD y MISD. ¿En qué aplicaciones se utilizan o se podrían utilizar este tipo de computadoras?

CAPÍTULO II
MÁQUINAS DE ESTADOS Y SU CONSTRUCCIÓN

2.1 MÁQUINAS DE ESTADOS

El modelo de máquina de estados contiene los elementos necesarios para describir la conducta de un sistema en términos de entradas, salidas y del tiempo.

El siguiente diagrama presenta el modelo general de una Máquina de Estados.

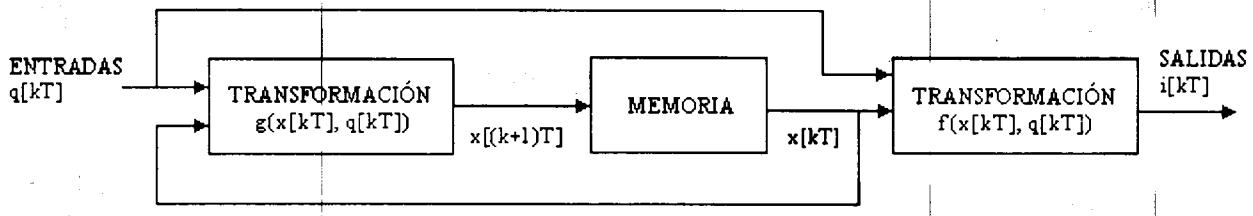


Figura. 2.1. Modelo general de una máquina de estados.

x[kT], representa el estado en el tiempo kT.
x[(k+1)T] = g(x[kT] , q[kT]), representa el siguiente estado.
q[kT], representa las entradas en el tiempo kT.
i[kT] = f(x[kT] , q[kT]), representa las salidas en el tiempo kT.

En donde T es el período de duración de cada estado y k es un contador entero.

2.1.1 EL ALGORITMO DE LA MÁQUINA DE ESTADOS

El algoritmo de la máquina de estados juega un papel muy importante en el diseño de sistemas digitales. Para circuitos síncronos la técnica de la carta ASM (Algorithm State Machine / Algoritmo de la Máquina de Estados) es la mejor notación, por lo tanto, se adoptará para el resto de la obra.

También existen otras técnicas como la de los diagramas de estados que son diagramas muy parecidos a las cartas ASM. Los diagramas de estados muestran gráficamente la secuencia de estados en un sistema y las condiciones de cada estado y de las transiciones entre cada uno de ellos.

Como ejemplo, en la figura 2.2 se describe el comportamiento de un contador binario de 3 bits mediante un diagrama de estados. Este circuito en particular no tiene ninguna entrada aparte de la de reloj, y ninguna otra salida más que las que se toman en cada flip-flop del contador.

2.1.2 ESTADOS Y RELOJ

El algoritmo de la máquina de estados se mueve a través de una secuencia de estados con base en la posición del estado presente y las variables de entrada. Los tiempos del estado están determinados por un reloj maestro.

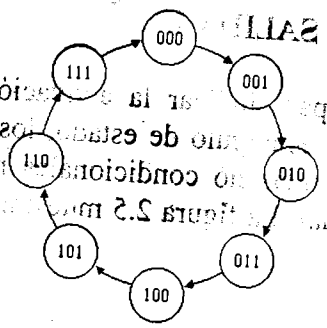


Figura. 2.2. Diagrama de estados para un contador binario de 3 bits.

2.2 NOTACIÓN DE LA CARTA ASM

2.2.1 REPRESENTACIÓN DE ESTADOS

El estado de una máquina de estados es la memoria de la historia pasada, suficiente para determinar las condiciones futuras. En la siguiente figura se muestra la representación del estado. Un estado se representa con un rectángulo y con su nombre simbólico en el extremo superior, encerrado en un círculo.

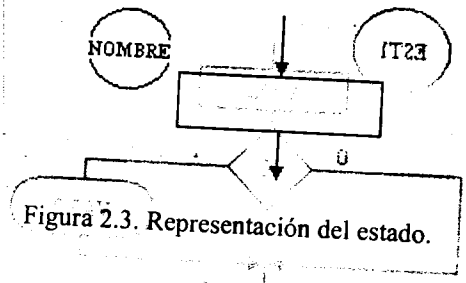


Figura 2.3. Representación del estado.

2.2.2 REPRESENTACIÓN DE DECISIONES

Las decisiones permiten seleccionar el camino que el algoritmo de la máquina de estados debe tomar de acuerdo a la variable o variables de entrada evaluadas. Las decisiones se representan mediante un rombo con el nombre de la variable a probar o una función que evalúe varias variables.

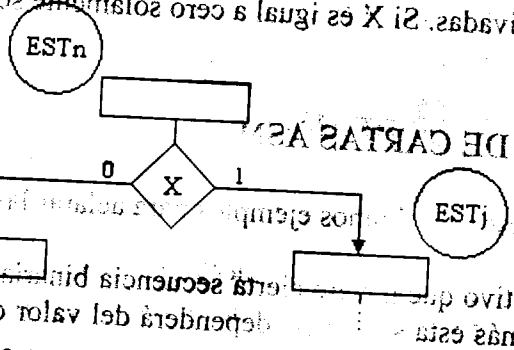


Figura 2.4. Representación de las decisiones.

2.2.3 REPRESENTACIÓN DE SALIDAS

Salidas no condicionales. Sirven para indicar la activación de una variable de salida. Para representarlas, se escriben dentro del rectángulo de estado, los nombres de las variables de salida que se activan en ese estado. Las salidas no condicionales no dependen de las condiciones de entrada, sólo dependen del estado actual. La figura 2.5 muestra la activación de las salidas VAR1 y VAR2 en el estado EST1.

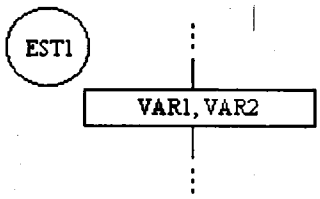


Figura 2.5. Representación de las salidas no condicionales.

Salidas Condicionales. Estas salidas se presentan solamente cuando ciertas condiciones de entrada existen. Se representan con un óvalo y los nombres de las salidas dentro de él.

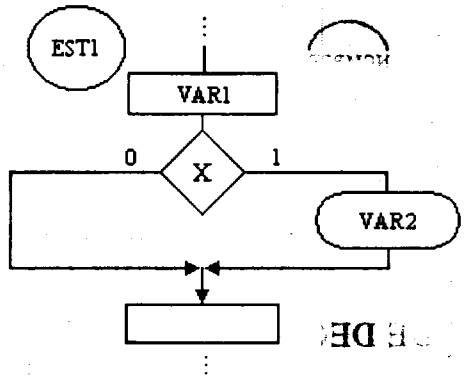


Figura 2.6. Representación de las salidas condicionales.

Para este ejemplo, si en el estado EST1 la variable de entrada X vale uno, entonces las salidas VAR1 y VAR2 serán activadas. Si X es igual a cero solamente se activará VAR1.

2.3 EJEMPLOS DE CARTAS ASM

A continuación se presentan algunos ejemplos para aclarar las ideas antes expuestas.

1) Diseñe un dispositivo que genere cierta secuencia binaria sólo cuando la variable INICIO sea igual a uno. Además esta secuencia dependerá del valor de la entrada X. Si X=0 la secuencia

binaria que se genera es la siguiente: 11, 10, 01, por el contrario, si X=1 la secuencia es: 01, 10, 11. Considere que cada pareja binaria se genera con un ciclo de reloj de diferencia.

Cuando se hace el diseño digital de un sistema es necesario hacer un diagrama de bloques que clarifique cuáles son las señales de entrada, cuáles las señales de salida, quién es el controlador y qué se está controlando.

Para este ejemplo, las señales de entrada son INICIO y X, y las señales de salida son las que representan la secuencia que se quiere generar, nombrémoslas VAR1 y VAR0.

El diagrama de bloques queda de la siguiente manera.

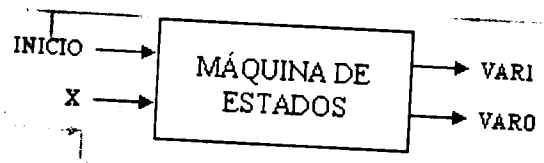


Figura 2.7. Diagrama de bloques para el ejemplo 1.

Y la carta ASM para esta máquina de estados se muestra en la figura 2.8.

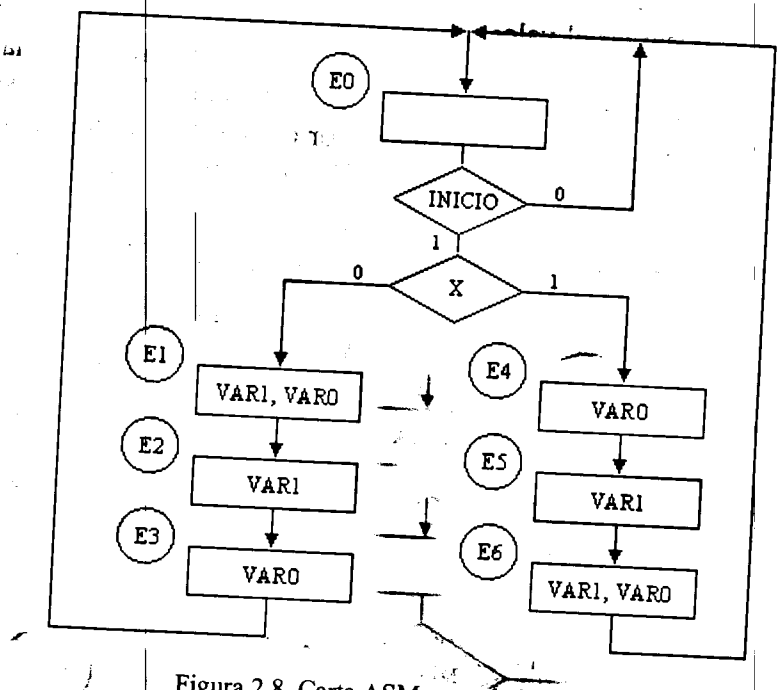


Figura 2.8. Carta ASM para el ejemplo 1.

De la figura 2.8 se puede observar que para activar una señal de salida incondicional, en un estado en particular, es necesario colocar su nombre dentro del rectángulo del estado.

2) Convertir el siguiente código en lenguaje 'C' a una carta ASM.

```
for (x = a; x ≤ b; x = x + c) {
      var1 = 1;  var2 = 0;
}
var1 = 0;
```

El diagrama de bloques de este sistema es el siguiente.

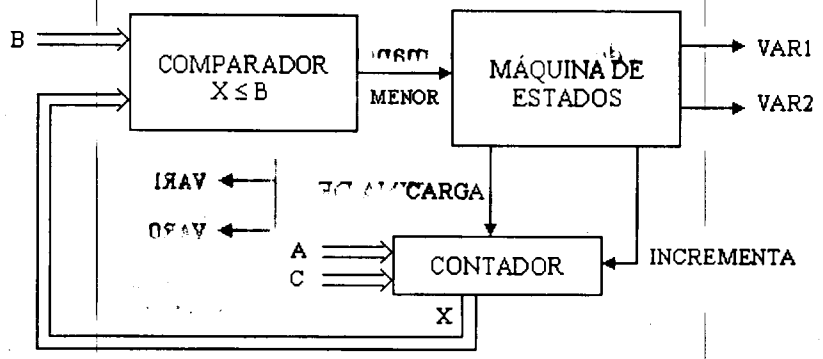


Figura 2.9. Diagrama de bloques para el ejemplo 2.

Se utiliza un contador para cargar el valor inicial de X o incrementar su valor en C unidades. La activación de la señal CARGA inicializará el valor de X con A, mientras que la activación de la señal INCREMENTA incrementará el valor de X en C unidades. También se cuenta con un comparador que evalúa la condición X ≤ B. Si X es menor o igual a B, el resultado es la activación de la señal MENOR, en caso contrario, la señal MENOR permanece en cero.

El algoritmo de la máquina de estados es el siguiente.

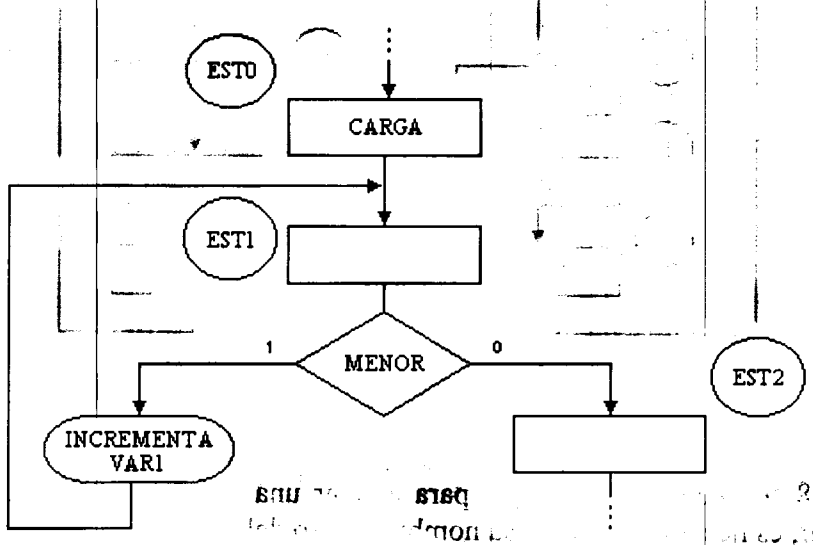


Figura 2.10. Carta ASM para el ejemplo 2.

En el estado EST0 se activa la señal CARGA con el fin de cargar en el contador el valor inicial de X. En el estado EST1 se pregunta por la variable de entrada MENOR, si ésta es igual a cero, la condición $X \leq B$ es falsa. Si MENOR es igual a uno, la condición es verdadera y por tanto, son activadas las señales INCREMENTA y VAR1 como salidas condicionales.

3) Convertir el siguiente código en lenguaje 'C' a una carta ASM.

```
while( x==0 ) {
      var5 = 1;
      var2 = 1;
      if( z==0 ) {
            x = 1;
            var5 = 0;
      }
}
var5 = 0;
var2 = 0;
```

En este ejemplo el valor de la variable X puede ser modificado por la lógica externa o por la máquina de estados. Por ello, para representar a X, utilizaremos un flip-flop cuyo valor será puesto a uno ó a cero dependiendo de las señales internas y externas.

El diagrama de bloques de este sistema es el siguiente.

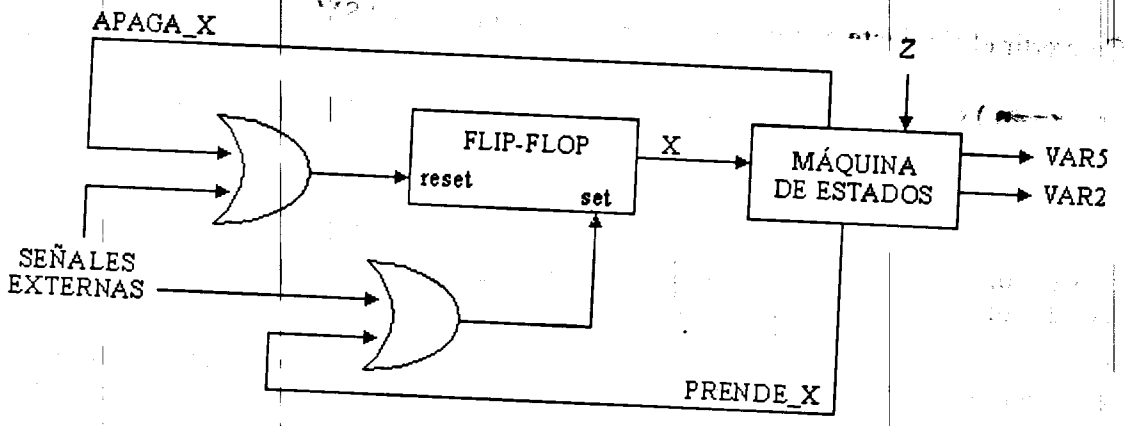


Figura 2.11. Diagrama de bloques para el ejemplo 3.

Y el algoritmo de la máquina de estados queda de la siguiente manera.

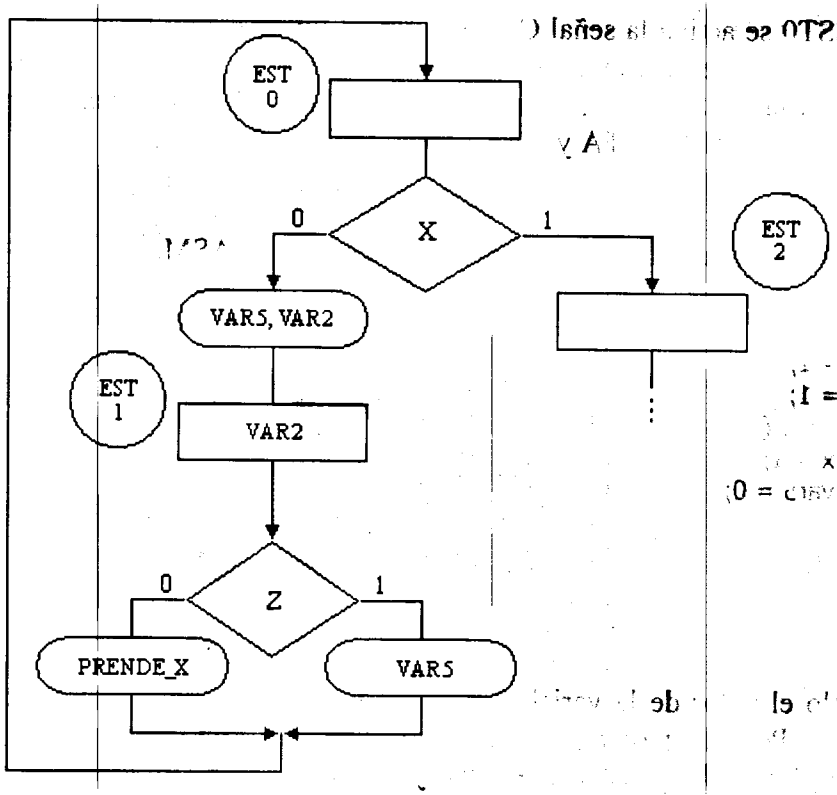
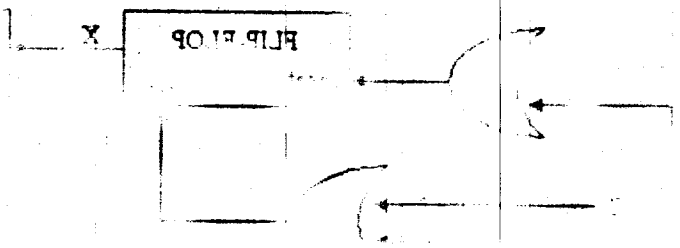


Figura 2.12. Carta ASM para el ejemplo 3.

4) Convertir el siguiente código en lenguaje 'C' a una carta ASM.

```
if ( x==n ) {
      var1 = 1; var2 = 0;
} else {
      var1 = 0; var2 = 1;
}
var1 = 0;
var2 = 0;
```



En este ejemplo las variables de entrada **x** y **n** están definidas como variables de un sólo bit. Para hacer la comparación de las variables **x** y **n** se usa la función lógica XOR, que valdrá cero cuando **x** y **n** sean iguales, y uno, cuando sean diferentes. La tabla XOR se presenta a continuación.

Entradas		Salida
x	n	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Tabla 2.1. Función lógica XOR.

El diagrama de bloques que ejecuta el código anterior en C se presenta enseguida.

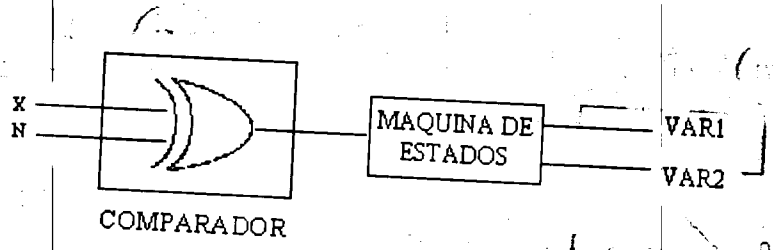


Figura 2.13. Diagrama de bloques para el ejemplo 4.

Y el algoritmo de esta máquina de estados es el siguiente.

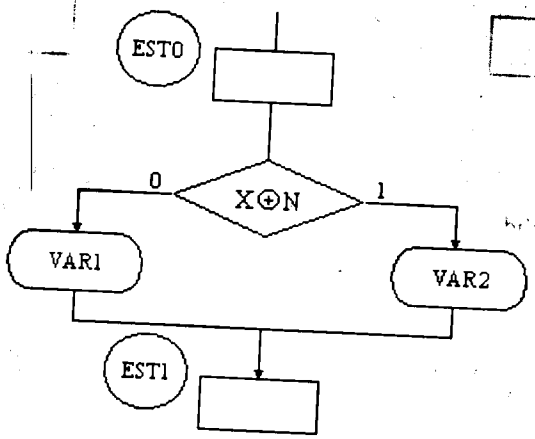


Figura 2.14. Carta ASM para el ejemplo 4.

En el diagrama se observa que si deseamos activar una señal de salida condicional, en un estado en particular, es necesario colocar su nombre dentro de un óvalo.

5) Usando un diagrama de tiempos mostrar la diferencia entre las cartas ASM de la figura 2.15, en donde la variable de salida VAR2 está como salida condicional en la carta ASM1 y como salida incondicional en la carta ASM2.

Si observa detenidamente el diagrama de tiempos de la figura 2.17 observará que la diferencia principal entre las dos cartas ASM es el tiempo cuando se activa la salida VAR2. Cuando está como salida condicional se activa en el estado EST1 junto con VAR1, y cuando está como salida incondicional se activa un flanco positivo después que VAR1.

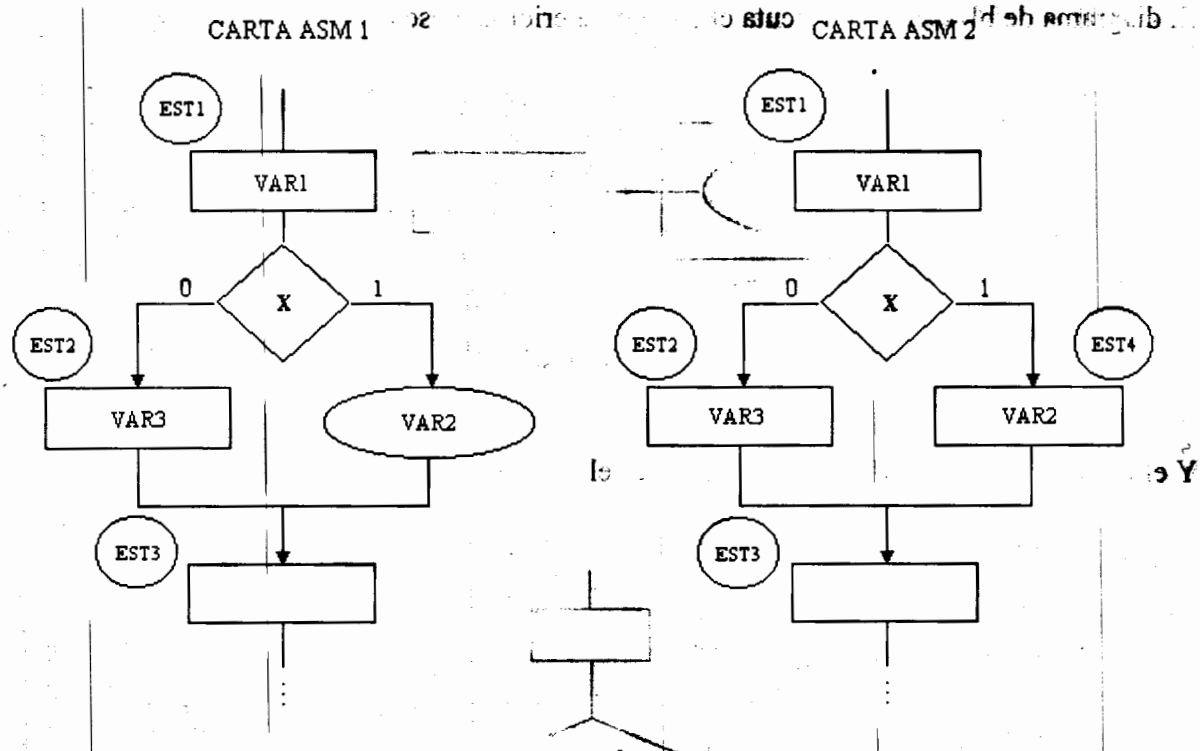


Figura 2.15. La salida VAR2 en la carta ASM1 se presenta como salida condicional, mientras que en la carta ASM2 se presenta como no condicional.

A continuación se muestra el diagrama de tiempos para la carta ASM1 cuando la entrada X es igual que cero. Este diagrama de tiempos es idéntico para la carta ASM2.

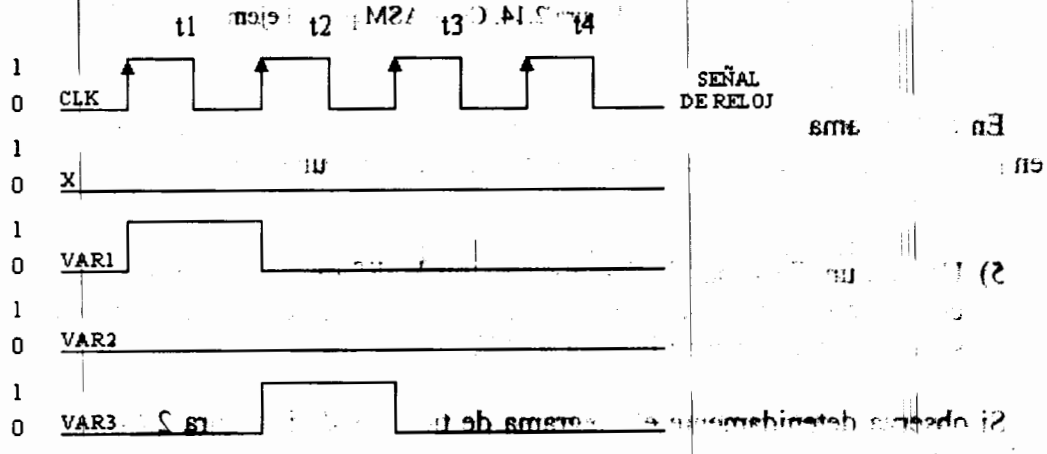


Figura 2.16. Diagrama de tiempos para las cartas ASM1 y ASM2, con x=0.

A continuación se muestran los diagramas de tiempos cuando X es igual a 1.

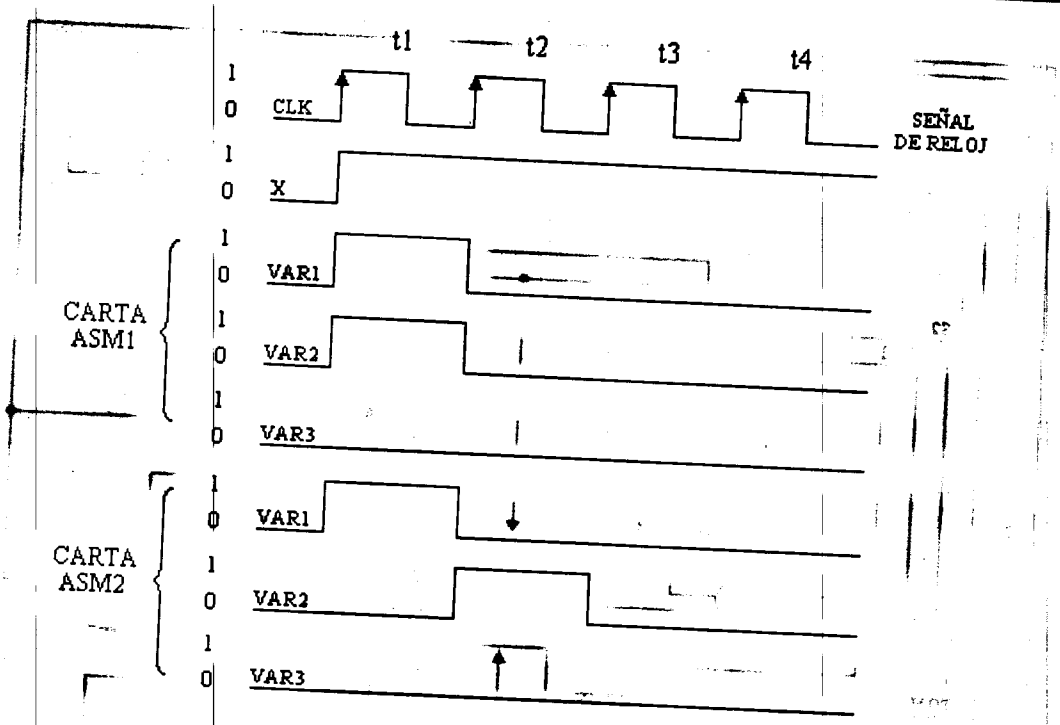


Figura 2.17. Diagrama de tiempos para las cartas ASM1 y ASM2, con x=1.

6) La siguiente figura muestra de manera simple la configuración de N estaciones del metro.

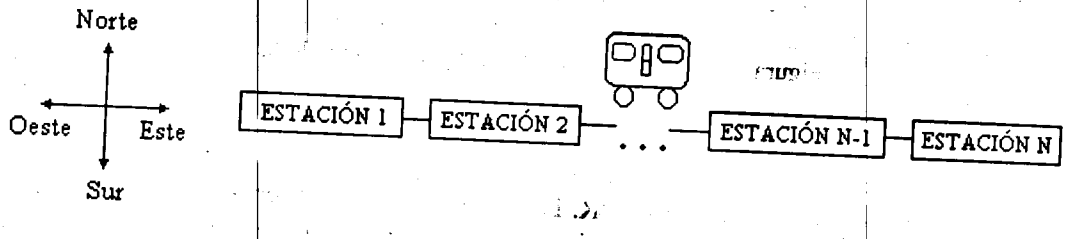


Figura 2.18. Configuración de las N estaciones del Metro.

El objetivo es diseñar un sistema digital, usando máquinas de estados, que mueva al tren de derecha a izquierda sobre la línea. En cada estación hay unos sensores que detectan la entrada de un tren, de manera que cuando arriba a una de ellas, hace una parada de dos minutos.

Además, existe un botón de emergencia en los vagones que hace que el tren se detenga un minuto extra en la estación, si así se requiriera.

A continuación se muestra el diagrama de bloques de este sistema:

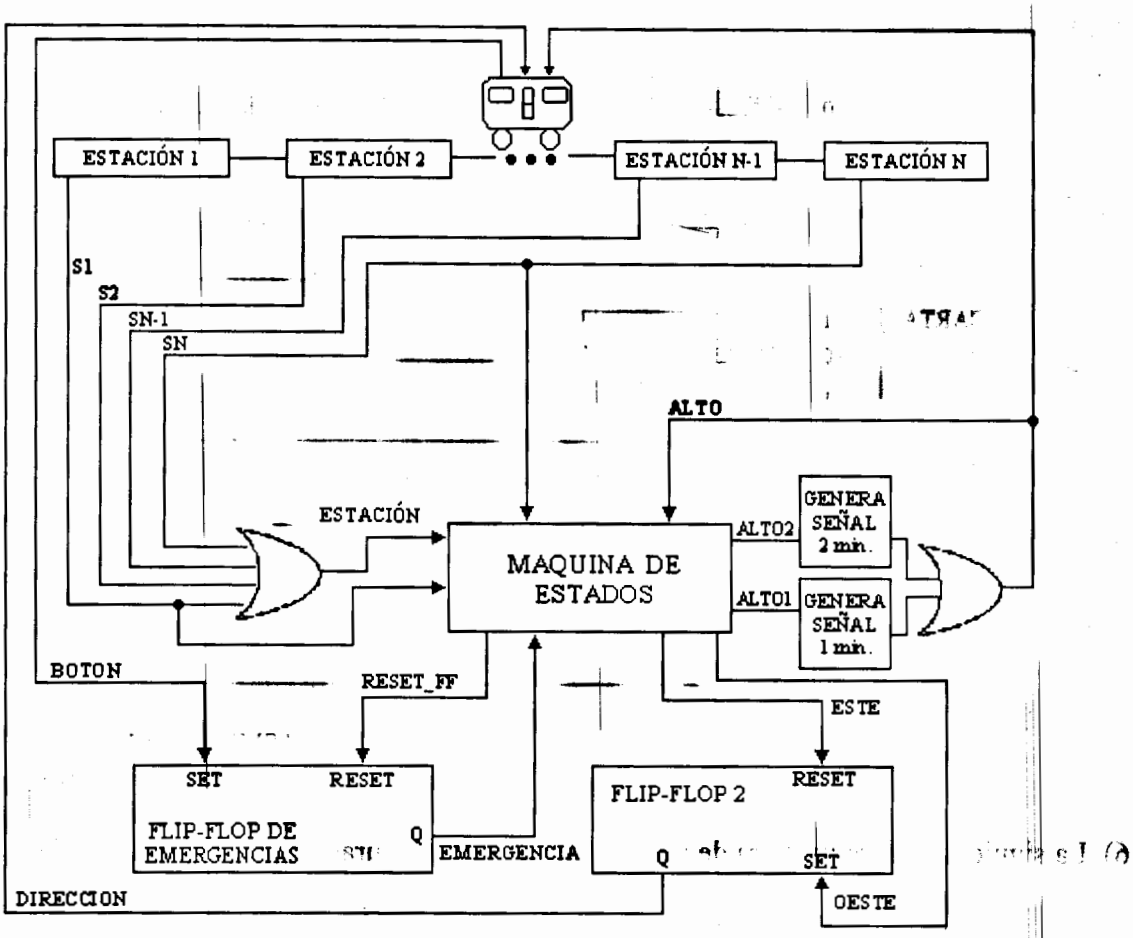


Figura 2.19. Diagrama de bloques para el ejemplo 6.

En el diagrama podemos observar que los sensores que detectan la presencia del tren en la estación están conectados a una compuerta OR. La salida de la compuerta OR, llamada ESTACIÓN, indica si un tren ha entrado en una estación, sin importar a qué estación entró. Sólo interesa saber a qué estación entra si se trata de las estaciones terminales S1 y SN, con el fin de cambiar la dirección de movimiento del tren.

Además, se tiene un módulo que genera una señal de salida de 2 minutos cuando la señal ALTO2 es activada. De manera similar, se tiene otro módulo que genera una señal de 1 minuto cuando ocurre una emergencia, es decir, se activa la señal ALTO1. Ambas señales de espera están conectadas a una compuerta OR. La salida de esta compuerta, llamada ALTO, se encarga de detener al tren durante el tiempo necesario. La salida ALTO también se retroalimenta a la máquina de estados para saber si el tren continúa parado.

También se tienen dos flip-flops, uno indica la dirección de movimiento del tren, y el otro la activación de la señal de emergencia. El flip-flop de emergencias es puesto a uno cuando se oprime el botón de emergencia en el tren, por ello, se conecta la línea del botón de emergencia en el SET del flip-flop. La salida de este flip-flop, denominada EMERGENCIA, entra a la máquina de estados.

El flip-flop de emergencias debe ser puesto a cero nuevamente para permitir otra emergencia, esto se hace por medio de la línea RESET-FF.

Por otra parte, el flip-flop de direcciones le indica al tren la dirección a seguir. Si la salida del flip-flop es igual a cero el tren irá hacia el este, si es uno irá hacia el oeste. El tren estará en movimiento todo el tiempo a menos que la señal de ALTO esté activada.

En la figura 2.20 se muestra la carta ASM de este ejemplo.

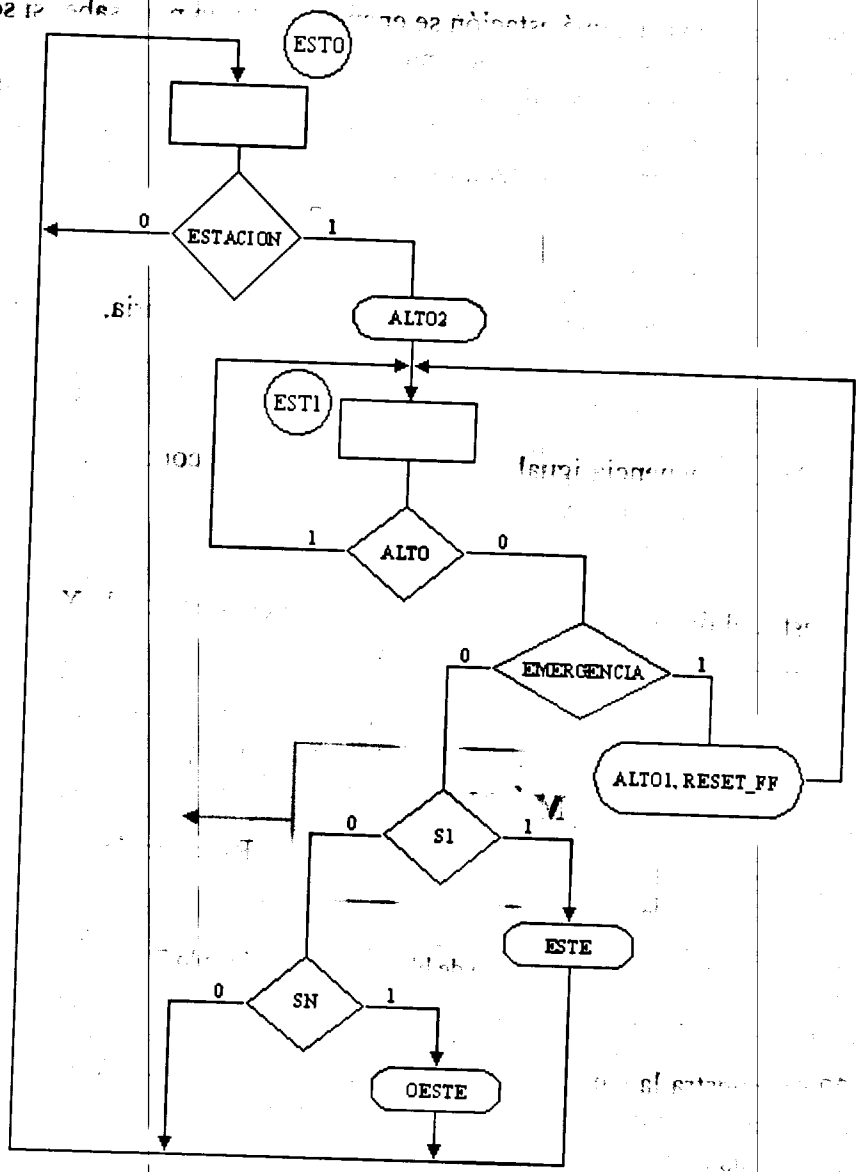


Figura 2.20. Carta ASM para el ejemplo 6.

En el estado EST0 se revisa el valor de la variable ESTACIÓN, si ésta es igual a cero indica que el tren no ha entrado a la estación, es decir, continúa avanzando. Si ESTACIÓN vale uno, entonces se activa la señal ALTO2, la cual genera la señal de 2 minutos que detiene al tren.

En el estado EST1 se revisa la variable ALTO, si ésta vale uno, el algoritmo permanece en ese estado hasta que ALTO valga cero. Cuando ALTO vale cero se revisa la señal de EMERGENCIA. Si el valor de EMERGENCIA es uno, se debe detener el tren por un minuto adicional. Para ello se activa la señal ALTO1 y se limpia el flip-flop de emergencias utilizando la señal de salida RESET_FF, de esta manera nuevas peticiones de emergencia serán permitidas.

Si no hay emergencias, se revisa en qué estación se encuentra el tren para saber si se ha alcanzado una de las estaciones terminales. Si S1 vale uno entonces el tren debe ir al este, por lo tanto, se activa la señal ESTE que pone en cero al flip-flop de dirección. Si SN es igual a uno, entonces se activa la señal OESTE para colocar en uno al flip-flop. Si no se ha alcanzado alguna de las estaciones terminales, el tren continuará avanzando siguiendo la dirección indicada por el flip-flop de direcciones.

7) Diseñe un sistema digital que reconozca la siguiente secuencia binaria.

0 1 0 1 1 1 0 1 1

Cuando haya llegado una secuencia igual, la señal de salida de reconocimiento deberá activarse. Tenga en cuenta que el primer bit de la secuencia es el que se localiza a la derecha, mientras que el último bit de la secuencia es el que se localiza a la izquierda.

La figura 2.21 muestra el diagrama de bloques de este problema en donde X representa cada uno de los bits de la secuencia binaria que hay que reconocer.

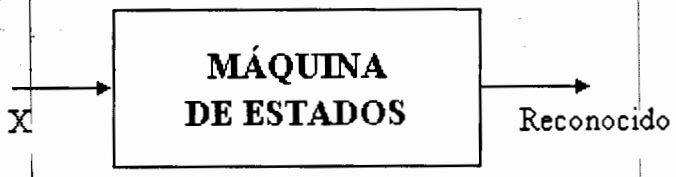


Figura 2.21. Diagrama de bloques para el ejemplo 7.

En la figura 2.22 se muestra la carta ASM de esta máquina de estados. Como puede ver, en cada estado se pregunta por el valor de entrada de la secuencia teniéndose una sincronización entre el tiempo de cada estado y las entradas.

Si en algún punto la secuencia es incorrecta, la máquina de estados regresaría a un estado en donde se tomarían en cuenta los valores de las entradas, que incluyendo el valor actual incorrecto, forman una secuencia válida.

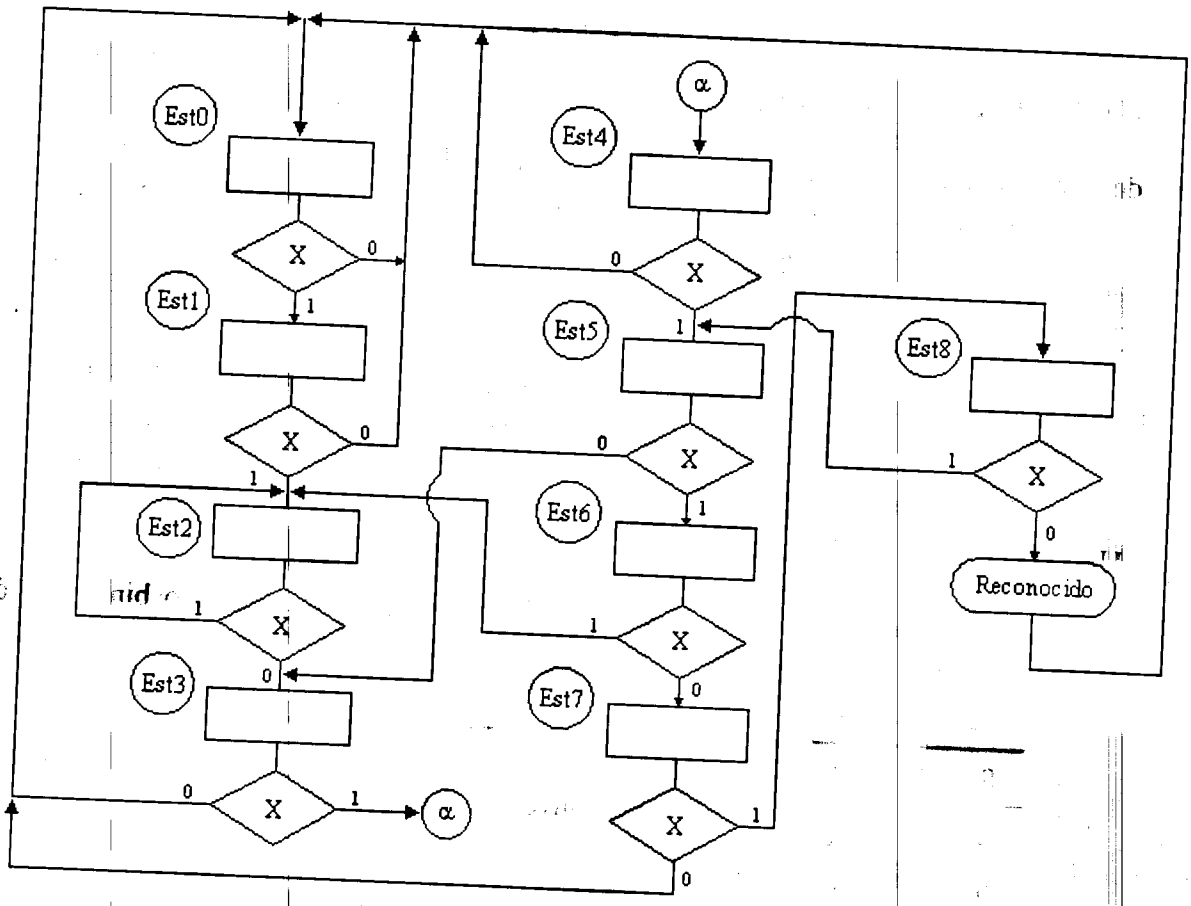


Figura 2.22. Carta ASM para el ejemplo 7.

Uno de los problemas de esta carta ASM es que si se quisiera cambiar la secuencia a reconocer se tendría que cambiar totalmente el algoritmo. Otra forma de resolver este problema sin utilizar cartas ASM se muestra en la figura 2.23, en donde la entrada X se introduce en un registro de corrimiento y se compara contra un registro que contiene el código a reconocer. Como puede ver, esta solución es mejor que el método en donde se utilizan cartas ASM, lo cual indica que no en todos los problemas se debe utilizar esta metodología.

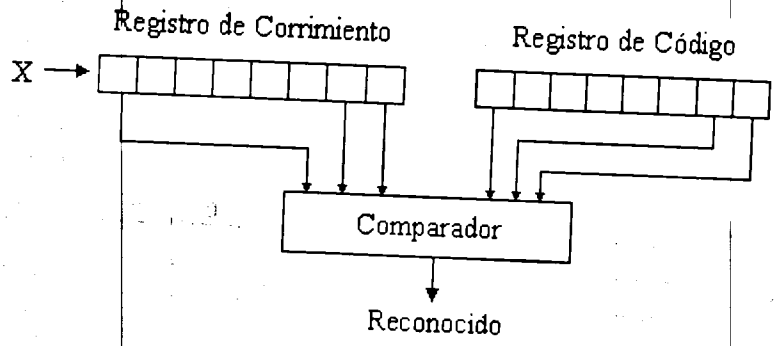


Figura 2.23. Solución al ejemplo 7 sin uso de cartas ASM.

2.4 CONSTRUCCIÓN DE MÁQUINAS DE ESTADOS USANDO LOS MÉTODOS TRADICIONALES

Antes de exponer una técnica de diseño específica, recordemos algunos conceptos de diseño digital como latches, flip-flops y circuitos secuenciales.

2.4.1 CIRCUITO SECUENCIAL

Un circuito secuencial está formado por una etapa de lógica combinacional, y una de etapa de memoria o flip-flops, que se utilizan para representar los estados.

2.4.2 UNIDAD BÁSICA DE ALMACENAMIENTO

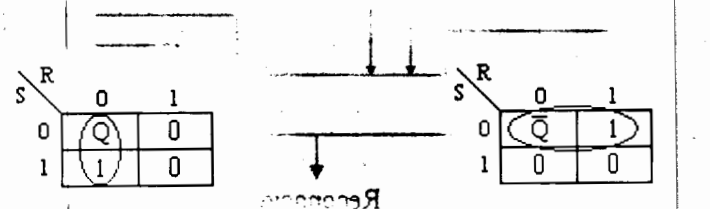
La unidad básica de almacenamiento es un dispositivo que almacena un dato binario, 0 ó 1, dependiendo de los valores de entrada. Este dispositivo presenta el siguiente comportamiento.

S	R	Q^+	$\overline{Q}^+$	Observaciones
0	0	Q	$\overline{Q}$	El dispositivo mantiene el valor que tenía guardado
0	1	0	1	Se almacena un cero
1	0	1	0	Se almacena un uno
1	1	0	0	Condición no válida

Tabla 2.2. Tabla de verdad para la unidad básica de almacenamiento.

Como se puede observar, esta unidad de almacenamiento cuenta con dos líneas de entrada, S (set) y R (reset), las cuales indican la manera en cómo deben operar las salidas Q^+ y Q^+. Las salidas de esta unidad siempre son complementarias una de la otra, es decir, cuando Q está a nivel alto, $\overline{Q}$ está a nivel bajo; y cuando Q está a nivel bajo, $\overline{Q}$ está a nivel alto. Debido a esta razón, la última condición de la tabla es inválida.

A partir de la tabla de verdad y utilizando mapas de Karnaugh es posible encontrar las expresiones lógicas para la unidad básica de almacenamiento.



Y con base en las expresiones lógicas se construye el diagrama lógico.

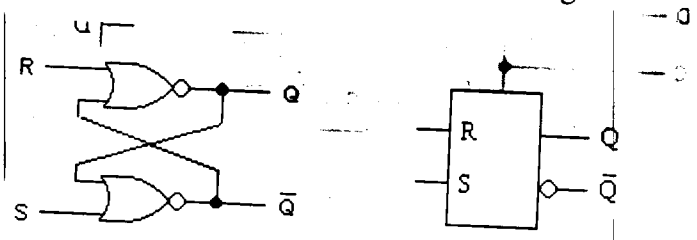


Figura 2.24. Diagrama y símbolo lógico para la unidad básica de almacenamiento.

2.4.3 LATCH TIPO D

El latch tipo D sólo tiene una entrada, además de la entrada de habilitación C, la cual está asociada a un reloj. El comportamiento de este dispositivo se muestra en la siguiente tabla y en el siguiente diagrama de tiempos.

D	C	Q^+	$\bar{Q}^+$	Observaciones
*	0	Q	$\bar{Q}$	El latch no cambia de estado
0	1	0	1	Latch en estado de Reset
1	1	1	0	Latch en estado de Set

Tabla 2.3. Tabla de verdad para el latch tipo D.

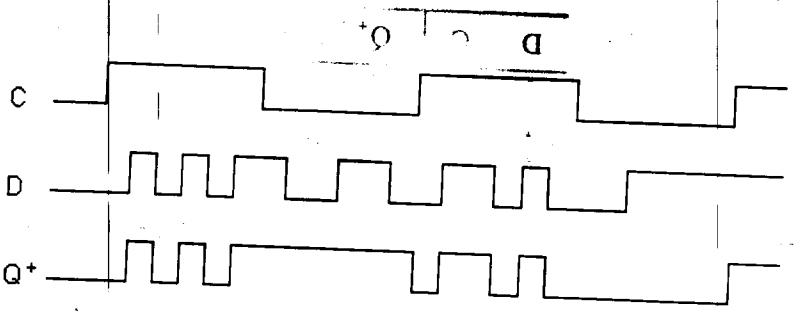


Figura 2.25. Diagrama de tiempos. Comportamiento del latch tipo D.

El diagrama de tiempos muestra de manera gráfica el funcionamiento del latch tipo D. En él se observa que mientras la señal de habilitación C vale uno, la salida Q^+ sigue al valor de la entrada D; y cuando C vale cero, la salida Q^+ mantiene el último valor de D antes de que C cambiara a cero.

El latch también es un dispositivo de almacenamiento de dos estados, por lo tanto, es posible diseñar un latch tipo D a partir de la unidad básica de almacenamiento. La siguiente figura ilustra este procedimiento.

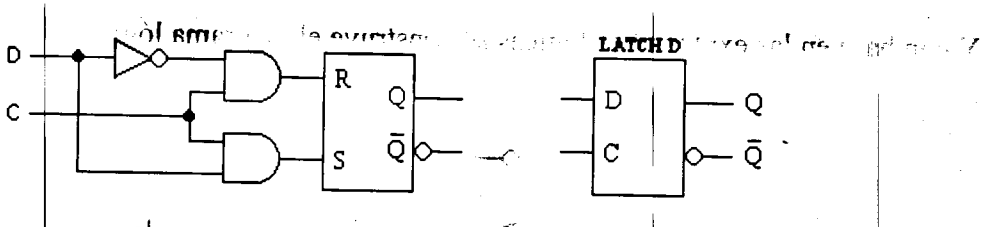


Figura 2.26. Diagrama y símbolo lógico para el latch tipo D.

Flip-flops

Los flip-flops son dispositivos biestables síncronos. En este caso, el término síncrono significa que la salida varía de estado únicamente en un instante específico de una entrada de disparo denominada reloj (clk). Es decir, un flip-flop cambia de estado con el flanco positivo (flanco de subida) o con el flanco negativo (flanco de bajada) del impulso de reloj y es sensible a sus entradas sólo en esta transición de reloj.

Básicamente, los latches son similares a los flip-flops, sin embargo, la diferencia principal entre ambos tipos de dispositivos está en el método empleado para cambiar de estado.

2.4.4 EL FLIP-FLOP TIPO D

Un flip-flop tipo D disparado por flanco negativo presenta el siguiente comportamiento.

D	C	Q^+	$\bar{Q}^+$
0	↓	0	1
1	↓	1	0
*	*	Q	$\bar{Q}$

Tabla 2.4. Tabla de verdad para el flip-flop D disparado por flanco negativo.

Si existe un nivel bajo en la entrada D cuando se aplica el impulso de reloj, el flip-flop se pone en estado de reset y almacena el nivel bajo de la entrada durante el flanco de bajada del reloj. Si cuando se aplica el impulso de reloj la entrada D está a nivel alto, el flip-flop se pone en estado set y almacena el nivel alto de la entrada durante el flanco de bajada del reloj. El flip-flop mantendrá su valor mientras no exista una transición de alto a bajo en la señal de reloj.

La siguiente figura muestra cómo construir un flip-flop tipo D a partir de dos latches tipo D.

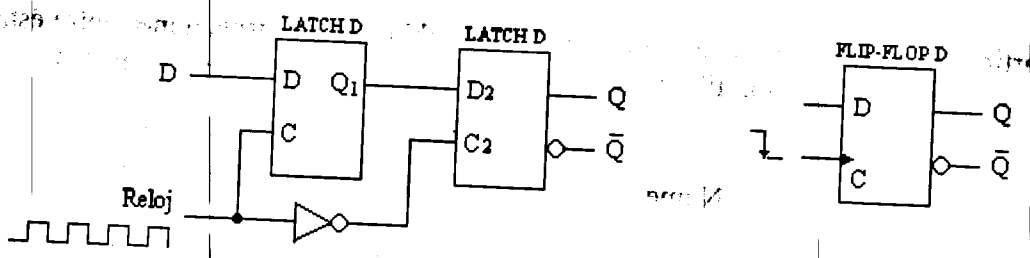


Figura 2.27. Diagrama y símbolo lógico para el flip-flop D.

También se anexa un diagrama de tiempos que muestra de manera gráfica el comportamiento de cada uno de los latches que constituyen al flip-flop tipo D.

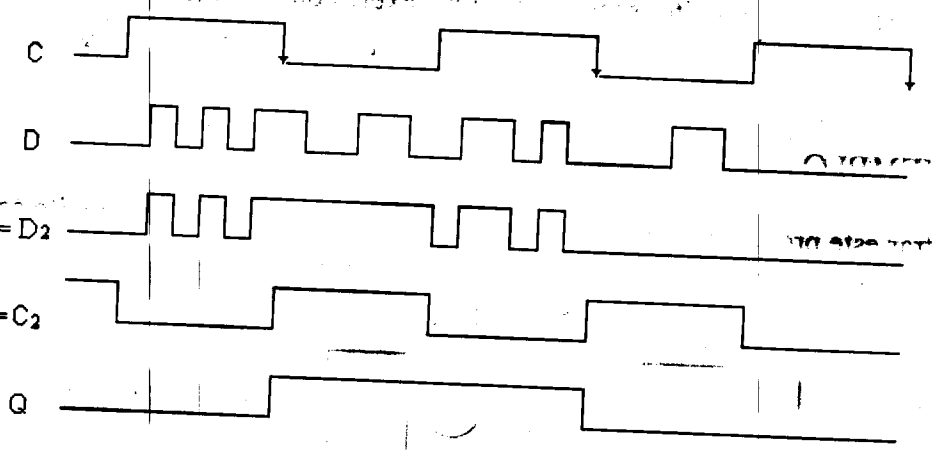


Figura 2.28. Diagrama de tiempos. Comportamiento del flip-flop tipo D.

Del diagrama se puede observar que tener el reloj negado en el segundo latch hace que la entrada D_2 se mantenga fija con el valor de salida del latch 1, el cual tendrá guardado el valor de la entrada D en el momento de la transición del reloj hacia cero.

2.4.5 PROCEDIMIENTO PARA EL DISEÑO DE CIRCUITOS SECUENCIALES

El método tradicional que se utilizará plantea los siguientes pasos para el diseño de un circuito secuencial:

1. Construir un Diagrama de Estados. Un diagrama de estados muestra la progresión de estados por los que el diseño avanza cuando se aplica una señal de reloj. Recuerde que en esta obra, el diagrama de estados corresponde a la carta ASM.

2. A partir del diagrama de estados, desarrollar una tabla con las transiciones entre estados y las salidas para cada estado. El número de flip-flops necesarios para representar todos los estados de la red secuencial está dado por:

$$\text{Número de Flip-flops} = n = \log_2 S$$

donde S es el número total de estados.

3. Transferir los valores para cada entrada D de los flip-flops a un mapa de Karnaugh. A partir de los mapas de Karnaugh podremos determinar las expresiones lógicas para las entradas de los flip-flops.

4. Transferir cada salida a un mapa de Karnaugh y obtener sus expresiones lógicas.

5. Implantación del circuito secuencial. Las expresiones lógicas obtenidas nos permitirán construir el diagrama lógico del circuito.

2.4.6 EJEMPLO

Para ilustrar este procedimiento se desarrollará un ejemplo: Diseñe un circuito secuencial a partir de la siguiente carta ASM.

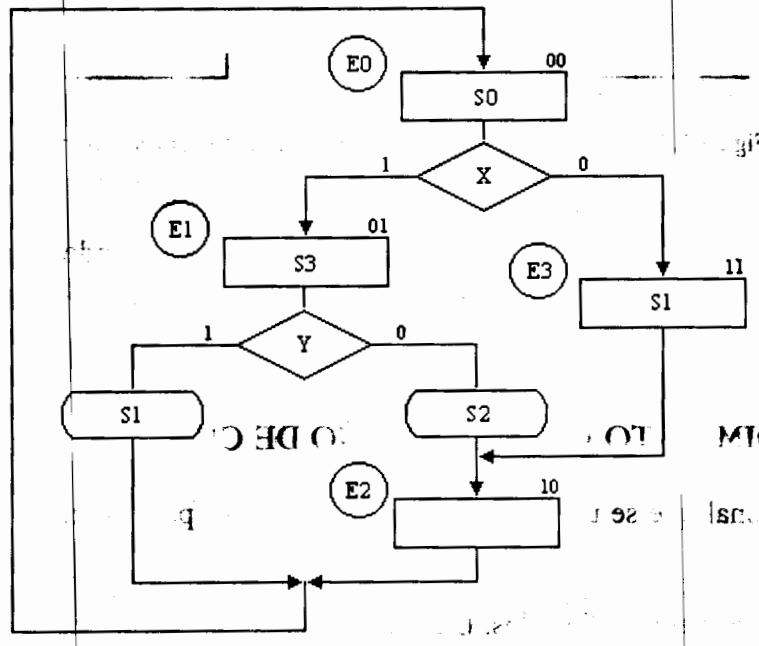


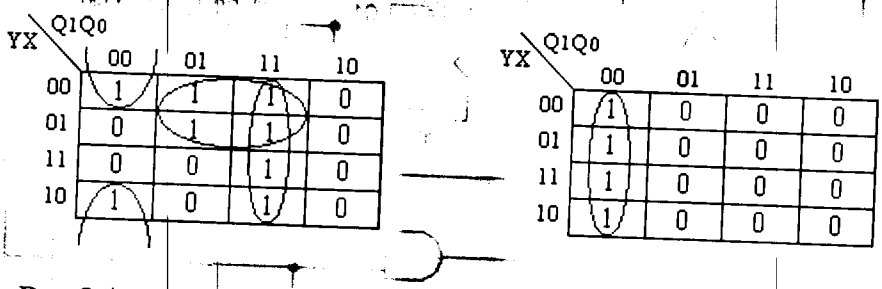
Figura 2.29. Carta ASM.

La tabla de transiciones de estados y de señales de salida es la siguiente.

Estado Presente		*Entradas*		*Estado Siguiente*		*Salidas*			
Q_1	Q_0	Y	X	Q_1^+	Q_0^+	S_3	S_2	S_1	S_0
0	0	0	0	1	1	0	0	0	1
0	0	0	1	0	1	0	0	0	1
0	0	1	0	1	1	0	0	0	1
0	0	1	1	0	1	0	0	0	1
0	1	0	0	1	0	1	1	0	0
0	1	0	1	1	0	1	1	0	0
0	1	1	0	0	0	1	0	1	0
0	1	1	1	0	0	1	0	1	0
1	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0
1	0	1	1	0	0	0	0	0	0
1	1	0	0	1	0	0	0	1	0
1	1	0	1	1	0	0	0	1	0
1	1	1	0	1	0	0	0	1	0
1	1	1	1	1	0	0	0	1	0

Tabla 2.5. Tabla de transiciones de estados y de salidas.

Se emplearon dos flip-flops tipo D para representar las transiciones de los cuatro estados que componen a la carta ASM. Los mapas de Karnaugh y las expresiones lógicas para las entradas de los flip-flops son los siguientes.



$$D_1 = Q_1^+ = Q_1 Q_0 + Q_0 \bar{Y} + \bar{Q}_1 \bar{Q}_0 X$$

$$D_0 = Q_0^+ = Q_1 Q_0$$

El mapa de Karnaugh para la entrada D_1 se construye con los valores del estado Q_1^+, mientras que el mapa para la entrada D_0 se construye con los valores del estado Q_0^+.

De manera similar, se obtienen las expresiones lógicas para las señales de salida.

YX \ Q1Q0	00	01	11	10
00	0	1	0	0
01	0	1	0	0
11	0	1	0	0
10	0	1	0	0

$S_3 = \overline{Q_1}\, Q_0$

YX \ Q1Q0	00	01	11	10
00	0	1	0	0
01	0	1	0	0
11	0	0	0	0
10	0	0	0	0

$S_2 = \overline{Q_1}\, Q_0\, \overline{Y}$

YX \ Q1Q0	00	01	11	10
00	0	0	1	0
01	0	0	1	0
11	0	1	1	0
10	0	1	1	0

$S_1 = Q_1\, Q_0 + Q_0\, Y$

YX \ Q1Q0	00	01	11	10
00	1	0	0	0
01	1	0	0	0
11	1	0	0	0
10	1	0	0	0

$S_0 = \overline{Q_1}\, \overline{Q_0}$

Finalmente, el diagrama lógico de esta máquina de estados es,

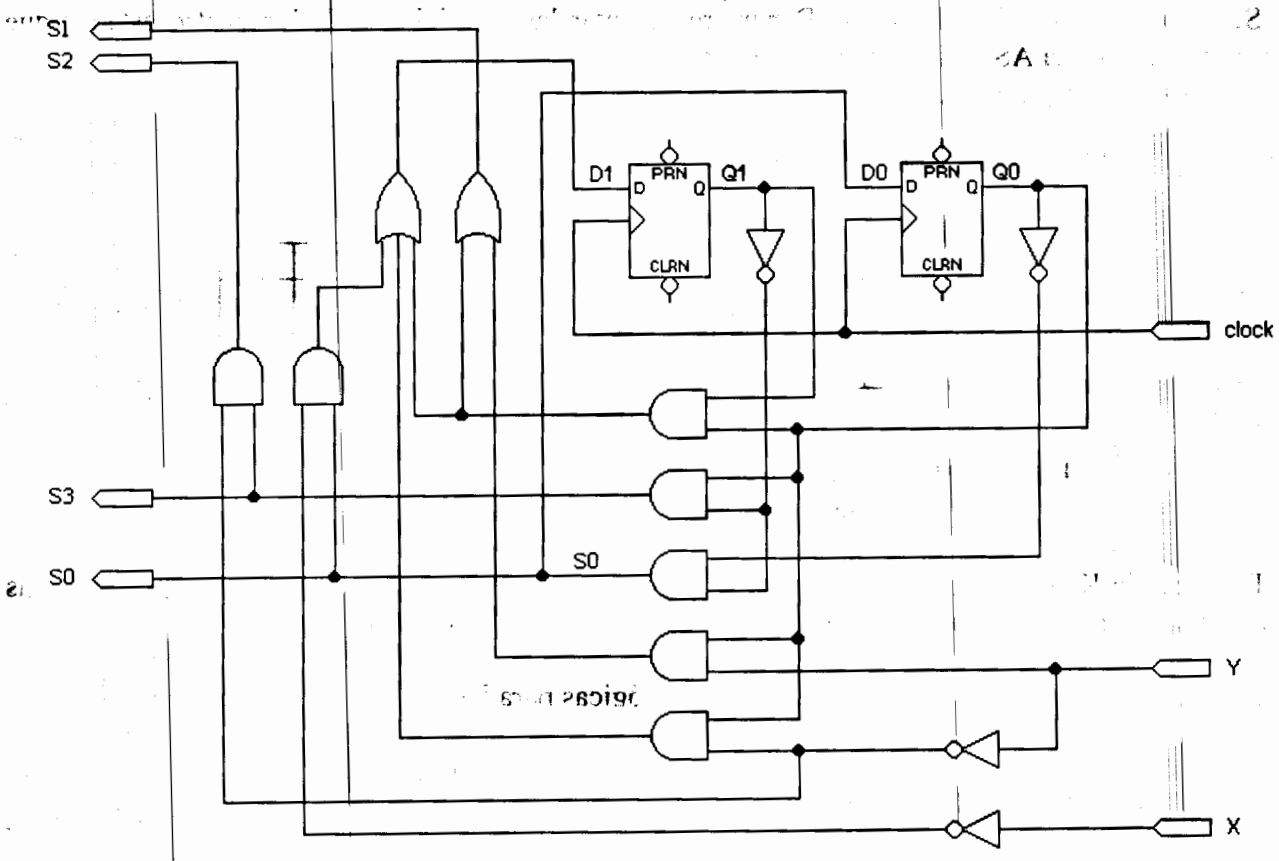


Figura 2.30. Diagrama lógico.

2.6 DISEÑO DIGITAL UTILIZANDO DISPOSITIVOS LÓGICOS PROGRAMABLES

Esta sección intenta explicar brevemente las ventajas del diseño utilizando dispositivos lógicos programables (PLDs, por sus siglas en inglés) sobre los métodos de diseño tradicionales. Esto no significa que los métodos tradicionales ya no sirvan, sin embargo, debido al avance tecnológico, los métodos tradicionales se han visto desplazados por nuevos métodos de diseño.

Para entender esta idea se desarrolla el ejemplo de la sección anterior utilizando un lenguaje de programación para PLDs. Para ello, utilizaremos el software MAX+PLUS II de Altera y alguno de los lenguajes de descripción de hardware que soporta.[1] Entre estos lenguajes están AHDL (Altera Hardware Description Language), VHDL (Very High Speed Integrated Circuit Hardware Description Language) y Verilog HDL. En este caso utilizaremos Verilog por su gran difusión en el campo de los dispositivos lógicos programables.

El siguiente programa implanta la carta ASM de la figura 2.29.

```
module Carta_ASM(clk, Y, X, S);

input           clk, Y, X;
output  [3:0]   S;
reg     [1:0]   currentState;
reg     [3:0]   S;

always @ (currentState) begin
    case (currentState)
        2'b00:          S = 4'b0001;
        2'b01:  if (Y)  S = 4'b1010;
                else    S = 4'b1100;
        2'b10:          S = 4'b0000;
        2'b11:          S = 4'b0010;
    endcase
end

always @ (posedge clk) begin
    case (currentState)
        2'b00:  if (X)  currentState = 1;
                else    currentState = 3;
        2'b01:  if (Y)  currentState = 0;
                else    currentState = 2;
        2'b10:          currentState = 0;
        2'b11:          currentState = 2;
    endcase
end

endmodule
```

[1] Consulte el apéndice A para cualquier duda acerca del entorno de MAX+PLUS II y del lenguaje Verilog HDL.

El programa comienza declarando el nombre del módulo y las señales de entrada y salida que lo integran. La señal de reloj es *clk*, las señales *X* e *Y* son las variables de entrada, la señal *S* es un vector de variables de salida (S3-S2-S1-S0), y *currentState* es una variable interna que mantiene el valor del estado actual.

Se emplean dos construcciones always, la primera genera las señales de salida y la segunda calcula las transiciones entre estados.

La primera construcción always tiene como evento de control a la variable *currentState*. Este evento hace que la construcción se ejecute sólo si *currentState* cambia de valor. Cada vez que la construcción se ejecuta, el valor de *currentState* y el de las señales de entrada es revisado, y en base a ellas se asigna un valor al vector de salida *S*.

La segunda construcción always utiliza la señal de reloj como evento de control, esto hace que la construcción sea ejecutada sólo cuando se registra un flanco de subida en la señal de reloj. En esta construcción se genera el valor del estado siguiente en base al valor de la variable *currentState* (el estado actual) y al valor de las entradas.

Las Ecuaciones Lógicas

Cada vez que se compila un proyecto en MAX+PLUS II se genera un archivo con extensión '**.rpt**' con bastante información acerca del proyecto, como por ejemplo, el nombre del dispositivo físico en el cual fue acomodado el diseño, el porcentaje de utilización de este dispositivo, el número de celdas lógicas utilizadas, la asignación de señales de entrada/salida a pines o puertos de entrada/salida en el dispositivo, y quizá la más interesante, las ecuaciones lógicas del proyecto.

Adicionalmente, se crea otro archivo con la información necesaria para programar físicamente el dispositivo lógico programable. Para los PLDs de la familia MAX este archivo tiene extensión '**.pof**' y para los PLDs de la familia FLEX el archivo tiene extensión '**.sof**'.

A continuación se presenta un fragmento del archivo '*Carta_ASM.rpt*' con las ecuaciones lógicas del diseño:

```
** EQUATIONS **

clk     : INPUT;
X       : INPUT;
Y       : INPUT;


-- Node name is ':51' = 'currentState0'
-- Equation name is 'currentState0', location is LC4_A1, type is buried.
currentState0 = DFFE( _EQ001, GLOBAL( clk),  VCC,  VCC,  VCC);
_EQ001 = !currentState0  &  !currentState1;
```

```
-- Node name is ':50' = 'currentState1'
-- Equation name is 'currentState1', location is LC2_A1, type is buried.
currentState1 = DFFE( _EQ002, GLOBAL( clk), VCC, VCC, VCC);
_EQ002 = !currentState0 & !currentState1 & !X # currentState0 & currentState1
       # currentState0 & !Y;

-- Node name is 'S0'
-- Equation name is 'S0', type is output
S0     = _LC7_A1;

-- Node name is 'S1'
-- Equation name is 'S1', type is output
S1     = _LC5_A1;

-- Node name is 'S2'
-- Equation name is 'S2', type is output
S2     = _LC3_A1;

-- Node name is 'S3'
-- Equation name is 'S3', type is output
S3     = _LC1_A1;

-- Node name is ':101'
-- Equation name is '_LC1_A1', type is buried
_LC1_A1  = LCELL( _EQ003);
_EQ003 =  currentState0  &  !currentState1;

-- Node name is ':102'
-- Equation name is '_LC3_A1', type is buried
_LC3_A1  = LCELL( _EQ004);
_EQ004 =  currentState0  &  !currentState1  &  !Y;

-- Node name is ':103'
-- Equation name is '_LC5_A1', type is buried
_LC5_A1  = LCELL( _EQ005);
_EQ005 =  currentState0  &  currentState1  #  currentState0  &  Y;

-- Node name is ':104'
-- Equation name is '_LC7_A1', type is buried
_LC7_A1  = LCELL( _EQ006);
_EQ006 = !currentState0  &  !currentState1;
```

Si compara estas expresiones con las calculadas en la sección anterior notará que son muy parecidas, salvo por algunos detalles como los nombres de algunas variables y la notación para las operaciones lógicas **and** (&), **or** (#) y **not** (!).

Ecuaciones obtenidas manualmente	*Ecuaciones de MAX+PLUS II*
$D_0 = Q_1 Q_0$	_EQ001 = !currentState0 & !currentState1
$D_1 = Q_1 Q_0 + Q_0 \overline{Y} + \overline{Q_1}\,\overline{Q_0}\,X$	_EQ002 = !currentState0 & !currentState1 & !X # currentState0 & currentState1 # currentState0 & !Y
$S_3 = \overline{Q_1} Q_0$	_EQ003 = currentState0 & !currentState1
$S_2 = \overline{Q_1} Q_0 \overline{Y}$	_EQ004 = currentState0 & !currentState1 & !Y
$S_1 = Q_1 Q_0 + Q_0 Y$	_EQ005 = currentState0 & currentState1 # currentState0 & Y
$S_0 = \overline{Q_1}\,\overline{Q_0}$	_EQ006 = !currentState0 & !currentState1

Tabla 2.6. Ecuaciones lógicas.

Por último, se presenta un diagrama de simulación del circuito, es decir, cómo se comporta el circuito a lo largo del tiempo. Note los cambios en las variables *currentState*, *S3*, *S2*, *S1* y *S0*, dependiendo de las señales de entrada y del estado presente.

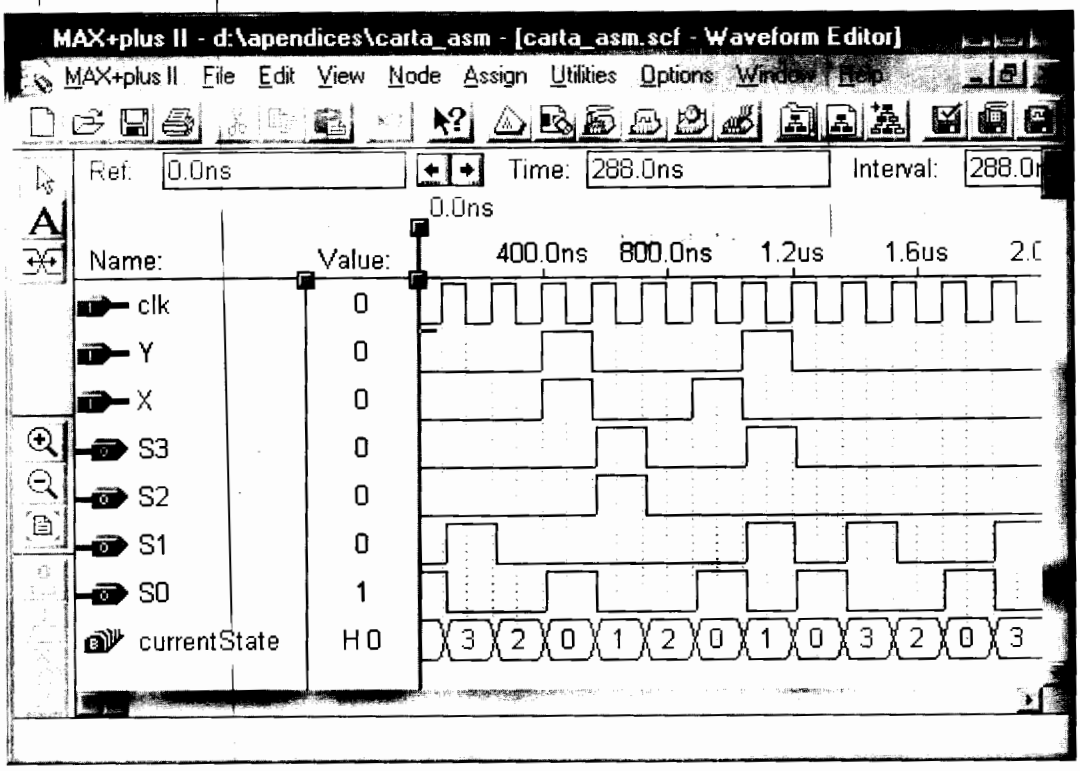


Figura 2.31. Diagrama de simulación.

Del ejemplo anterior podemos concluir lo siguiente:

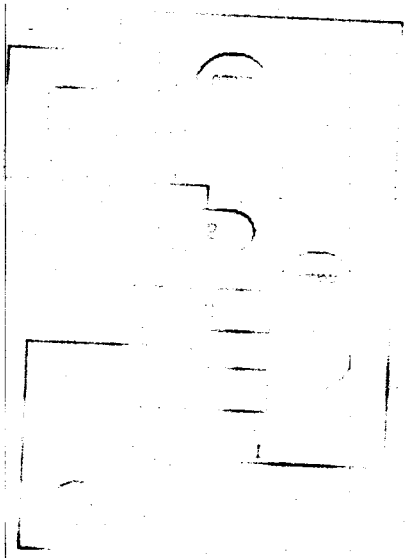
- Los lenguajes de programación de los PLDs permiten diseñar fácilmente cualquier tipo de sistema digital, además de que permiten hacer modificaciones en el diseño de manera rápida.

En cambio, si utilizamos los métodos tradicionales de diseño, una pequeña modificación implicaría recalcular las expresiones lógicas del circuito, y por lo tanto, necesitaríamos dedicar mayor tiempo al diseño.

- Si trabaja con funciones lógicas muy complejas o muy grandes resultará más barato adquirir un PLD en lugar de adquirir los circuitos integrados por separado. Además de que el espacio ocupado por el PLD en su sistema será menor que el espacio ocupado por todos los circuitos individuales.

- Un PLD será de gran utilidad cuando requiera un sistema digital re-configurable, es decir, que la lógica del sistema necesite modificarse regularmente.

- Si requiere diseñar un sistema de alto rendimiento, utilizar un PLD es una buena opción, ya que estos dispositivos tienen tiempos de respuesta muy pequeños comparados con otros circuitos integrados.

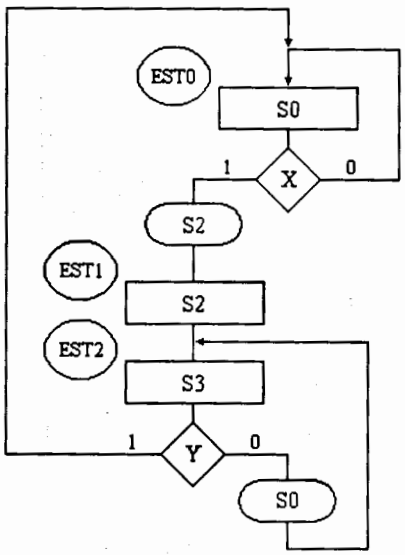


PROBLEMAS

1. Diseñe el algoritmo de una máquina de estados, carta ASM, que genere la siguiente secuencia binaria: 000, 010, 101, 111, 110, 011, 100, 001.

2. Diseñe un sistema digital que reconozca la siguiente secuencia binaria: 1 0 1 1 1 0 1 0. Cuando haya llegado una secuencia igual, la señal de salida de reconocimiento deberá activarse. El primer bit de la secuencia es el que se localiza a la izquierda, mientras que el último bit de la secuencia es el que se localiza a la derecha.

3. Indique el diagrama de tiempos para las variables de salida de la siguiente carta ASM. Suponga que X toma el valor de uno e Y toma el valor de cero.

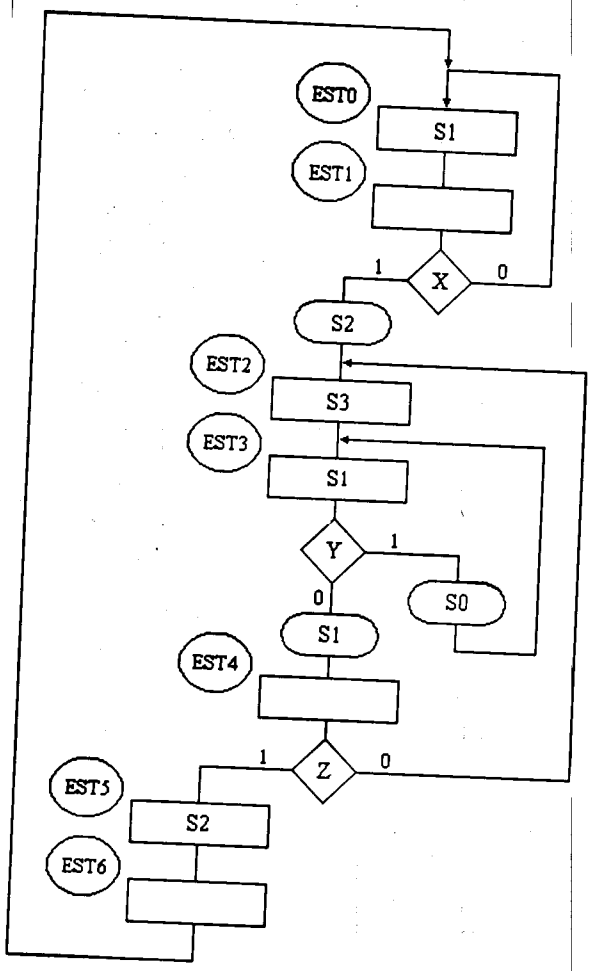


4. Se desea controlar la operación de un elevador para un edificio de ocho pisos. En cada piso, excepto el primero y el último, se cuenta con dos botones que indican la dirección hacia donde se desea ir, es decir, existe un botón para subir y otro para descender. En el primer piso sólo existe el botón para subir y en el último piso sólo existe el botón para bajar. Dentro del elevador se encuentra un tablero con botones que indican el piso al que se desea ir, además, cuenta con botones de abrir y cerrar la puerta y de un botón de emergencia. En los marcos de las puertas se tienen sensores de paso que indican si hay gente entrando o saliendo del ascensor. Diseñe un algoritmo de máquina de estados que controle la operación del elevador, indicando el diagrama de bloques del sistema así como la carta ASM.

5. Modifique el ejemplo de la figura 2.20 para que en lugar de salidas condicionales se tengan solamente salidas no condicionales, ¿qué afecto tiene esto en el desempeño del sistema?.

6. Diseñe una cerradura digital que acepte hasta cinco combinaciones diferentes. Indique el diagrama de bloques del sistema y la carta ASM si se requiriera.

7. Para la siguiente carta ASM encuentre las ecuaciones booleanas de los flip-flops, así como las ecuaciones booleanas de las salidas.



8. Codifique la carta ASM del ejercicio 7 usando el lenguaje de descripción de hardware Verilog HDL.

CAPÍTULO III
CONSTRUCCIÓN DE MÁQUINAS DE ESTADOS USANDO MEMORIAS

En el capítulo anterior se explicó cómo construir máquinas de estados a partir de sus algoritmos, representados con cartas ASM. En cierta forma, el algoritmo era el que definía la configuración física de los componentes (hardware). En este capítulo se explicará cómo instrumentar los algoritmos de máquinas de estados en configuraciones físicas de componentes ya definidas e independientes de los algoritmos a ser ejecutados. En particular, se construirán máquinas de estados utilizando memorias.

3.1 DIRECCIONAMIENTO POR TRAYECTORIA

Este tipo de direccionamiento guarda el estado siguiente y las salidas de cada estado de la carta ASM en una localidad de memoria. La porción de la memoria que indica el estado siguiente es llamada "la liga", mientras que la porción que indica las salidas es llamada "la parte de las salidas".

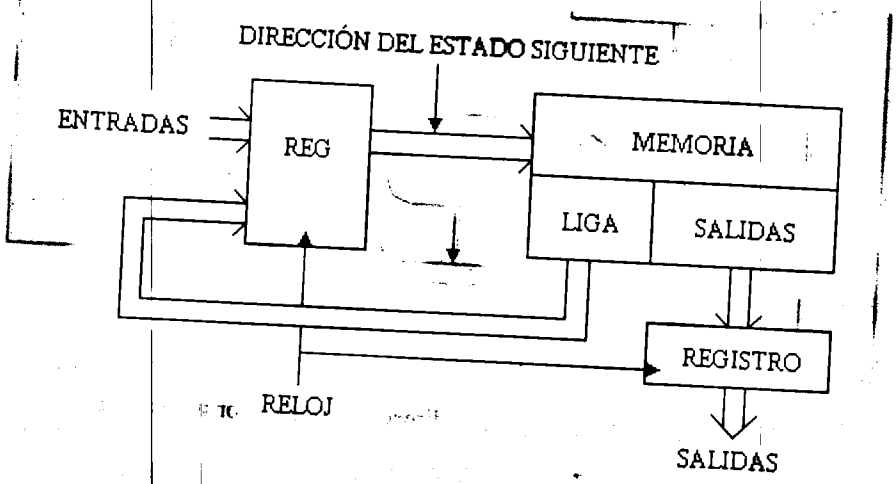


Figura 3.1. Direccionamiento por trayectoria.

Concatenando la liga del estado presente junto con las entradas se forma la dirección de memoria que contiene la dirección del estado siguiente. Esta dirección se guarda en un registro que está conectado a las líneas de dirección de la memoria, como se muestra en la figura 3.1. La señal de reloj conectada a los registros es la que indica la velocidad de la máquina de estados.

La figura 3.2 muestra una carta ASM, y la tabla 3.1 el contenido de la memoria usando el método de direccionamiento por trayectoria. Recuerde que antes de construir la tabla se debe asignar a cada estado de la carta ASM una representación binaria, de esta manera, para representar los cuatro estados que componen a la carta ASM de la figura 3.2 se necesitarán dos bits. Esta asignación binaria aparece en la esquina superior derecha de cada estado.

Para cada estado es necesario considerar todas las posibles combinaciones de las variables de entrada, aún cuando algunas de ellas no se utilicen. Por ejemplo, si en el estado EST0 la variable Q1 es igual a 1, la máquina de estados pasará al estado EST1 independientemente de los valores de las otras variables. Aún así, se deben considerar todas las combinaciones de las otras variables de entrada y colocar en las localidades de memoria correspondientes los valores adecuados.

Cuando una salida se activa, el bit correspondiente de memoria se coloca a 1, en caso contrario se coloca a 0.

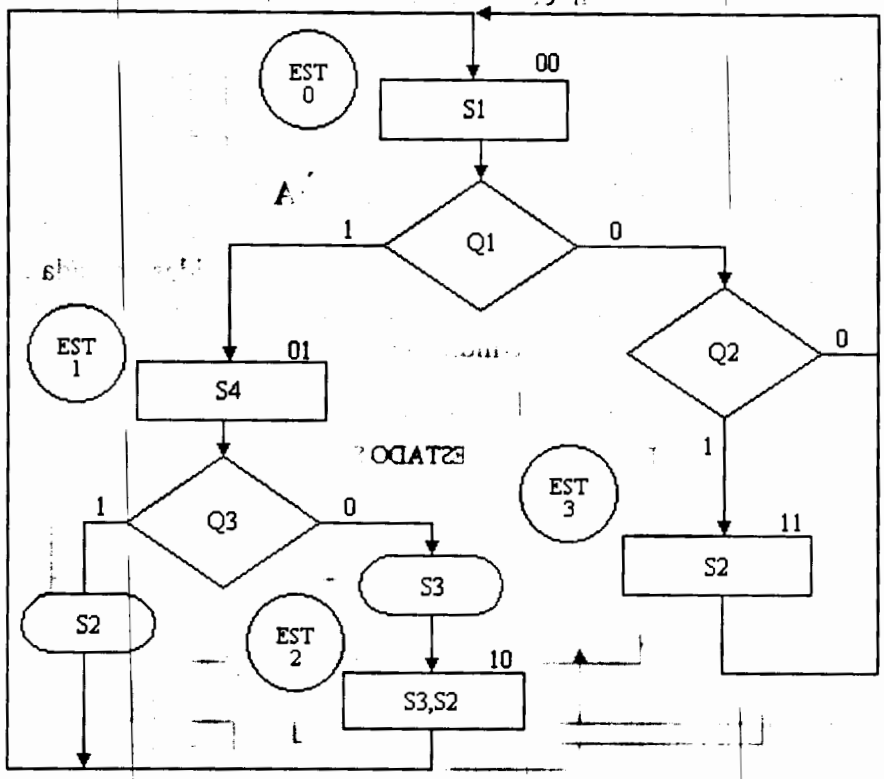


Figura 3.2. Carta ASM para el direccionamiento por trayectoria.

	Dirección de Memoria				Contenido de Memoria					
	Edo. Presente		Entradas			Liga		Salidas		
			Q_1	Q_2	Q_3			S_1	S_2	S_3	S_4
	0	0	0	0	0	0	0	1	0	0	0
	0	0	0	1	0	1	1	1	0	0	0
	0	0	0	1	1	1	1	1	0	0	0
EST0	0	0	0	0	1	0	0	1	0	0	0
	0	0	1	0	0	0	1	1	0	0	0
	0	0	1	0	1	0	1	1	0	0	0
	0	0	1	1	0	0	1	1	0	0	0
	0	0	1	1	1	0	1	1	0	0	0
	0	1	0	0	0	1	0	0	0	1	1
	0	1	0	0	1	0	0	0	1	0	1
	0	1	0	1	0	1	0	0	0	1	1
EST1	0	1	0	1	1	0	0	0	1	0	1
	0	1	1	0	0	1	0	0	0	1	1
	0	1	1	0	1	0	0	0	1	0	1
	0	1	1	1	0	1	0	0	0	1	1
	0	1	1	1	1	0	0	0	1	0	1

EST2	1	0	0	0	0	0	0	1	1	0
	1	0	0	0	1	0	0	1	1	0
	1	0	0	1	0	0	0	1	1	0
	1	0	0	1	1	0	0	1	1	0
	1	0	1	0	0	0	0	1	1	0
	1	0	1	0	1	0	0	1	1	0
	1	0	1	1	0	0	0	1	1	0
	1	0	1	1	1	0	0	1	1	0
EST3	1	1	0	0	0	0	0	1	0	0
	1	1	0	0	1	0	0	1	0	0
	1	1	0	1	0	0	0	1	0	0
	1	1	0	1	1	0	0	1	0	0
	1	1	1	0	0	0	0	1	0	0
	1	1	1	0	1	0	0	1	0	0
	1	1	1	1	0	0	0	1	0	0
	1	1	1	1	1	0	0	1	0	0

Tabla 3.1. Contenido de la memoria.

3.2 DIRECCIONAMIENTO DE ENTRADA-ESTADO

Este tipo de direccionamiento se restringe a cartas ASM con una sola entrada por estado. Una nueva porción de la palabra de memoria contiene una representación binaria de la entrada a probar en cada estado, esta parte es llamada "la parte de prueba". Con esta representación binaria un selector de entrada elige una de las variables de entrada.

La parte de liga tiene dos estados siguientes, escogiéndose uno por el selector de liga, en base a la entrada seleccionada por la parte de prueba. La figura 3.3 muestra el diagrama de bloques de este método.

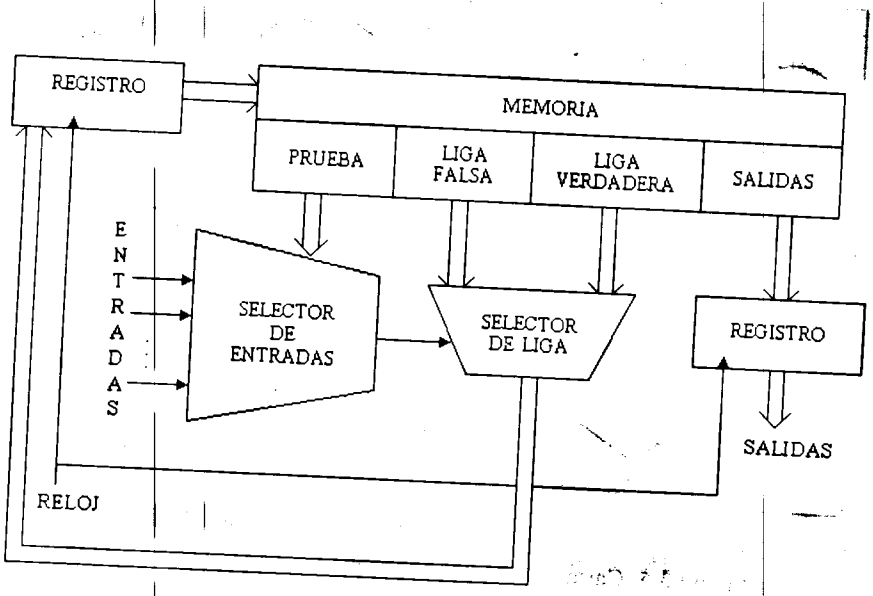


Figura 3.3. Diagrama de bloques para el método entrada-estado.

Si el valor de la entrada seleccionada por el selector de entradas es igual a cero, entonces el selector de liga elegirá la liga falsa, en caso contrario se seleccionará la liga verdadera; este caso se muestra en la figura 3.4.

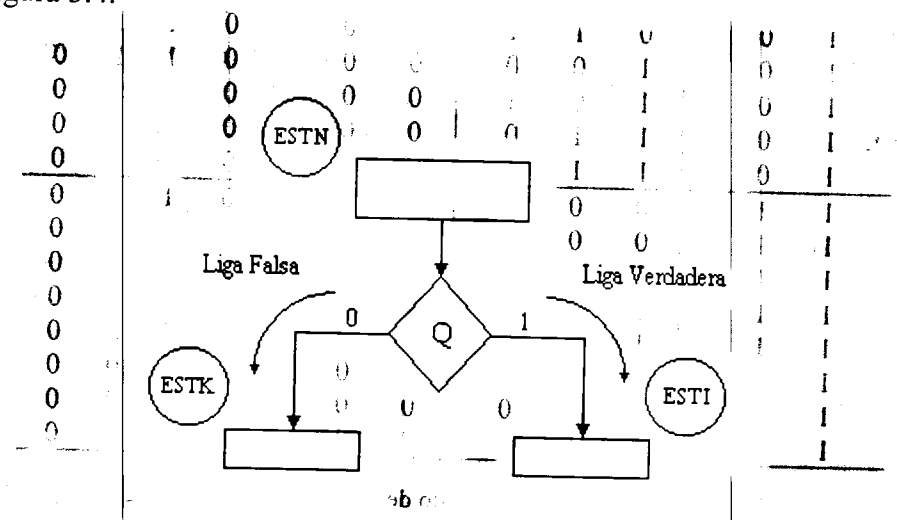


Figura 3.4. La liga falsa es el estado ESTK, mientras que la liga verdadera es el estado ESTI.

La figura 3.5 muestra una carta ASM la cual se construirá utilizando este método.

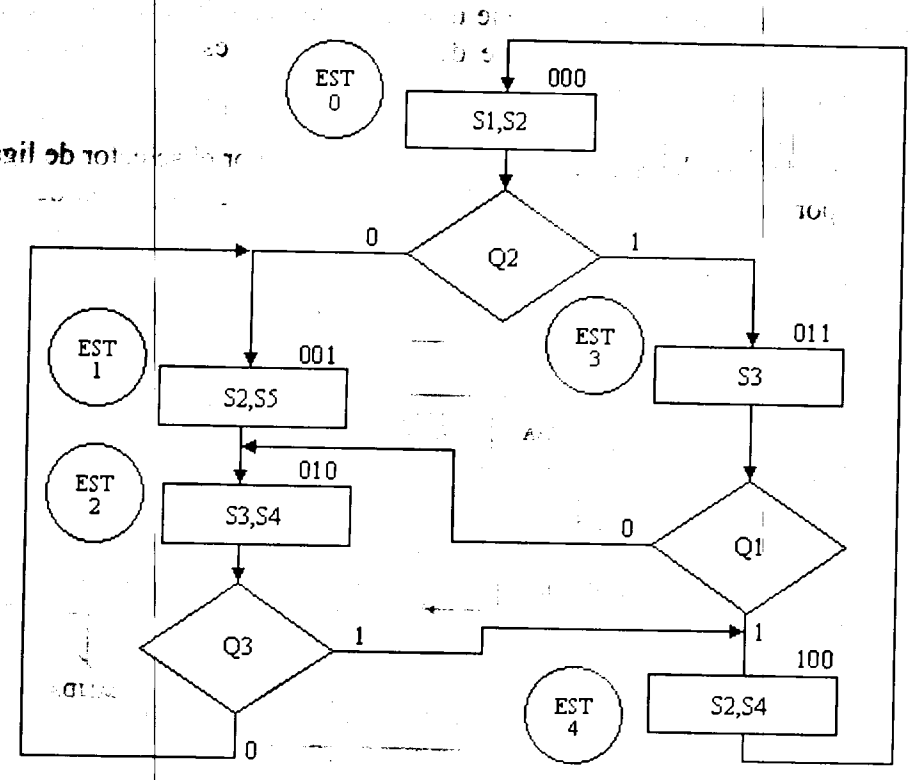


Figura 3.5. Carta ASM para el direccionamiento entrada-estado.

Además de asignar una representación binaria a cada estado, también a cada variable de entrada se le asignará una.

Se utiliza también una variable auxiliar que nos sirve para los estados que no tengan variable de entrada, de manera que cuando en un estado no exista variable de entrada se probará la variable auxiliar, la cual tiene un valor preestablecido de cero ó uno. Así, si quisiéramos forzar a la máquina de estados para que salte a un estado específico se escogería la variable auxiliar Q_x y dependiendo del valor de ésta (0 ó 1), en la liga falsa o verdadera según sea el caso, se tendría la dirección del estado a saltar.

Para el ejemplo de la figura 3.5 se eligió la siguiente representación binaria de las entradas:

Q_x = Variable Auxiliar = 00
Q_1 = 01
Q_2 = 10
Q_3 = 11

A continuación se describe cómo llenar los campos de la memoria para el estado EST0.

En el estado EST0 se selecciona la entrada Q_2, por lo tanto, se coloca en el campo de prueba de la memoria su representación binaria, es decir, un 1 y un 0. Si Q_2 es igual a cero, el estado siguiente es EST1 y su representación binaria, 001, es colocada en el campo de la liga falsa. Si Q_2 es igual a uno, el estado siguiente es EST3 y su representación binaria, 011, es colocada en el campo de la liga verdadera. Con los demás estados se procede de la misma forma.

La tabla 3.2 muestra el contenido de la memoria para el ejemplo anterior.

Dirección de Memoria			**Contenido de la Memoria**												
Edo. Presente			Prueba		Liga Falsa			Liga Verdadera			S1	S2	S3	S4	S5
0	0	0	1	0	0	0	1	0	1	1	1	1	0	0	0
0	0	1	0	0	0	1	0	0	1	0	0	1	0	0	1
0	1	0	1	1	0	0	1	1	0	0	0	0	1	1	0
0	1	1	0	1	0	1	0	1	0	0	0	0	1	0	0
1	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0

Tabla 3.2. Contenido de la memoria.

Para manejar salidas condicionales es necesario modificar el diagrama de bloques del direccionamiento entrada-estado. Esta modificación consiste en tener dos campos de salidas: el campo de salidas falsas y el campo de salidas verdaderas. En el primero, están las salidas que se activan cuando la variable de entrada sensada vale cero, y en el segundo, están las salidas que se activan cuando la variable de entrada sensada vale uno.

La figura 3.6 muestra esta configuración.

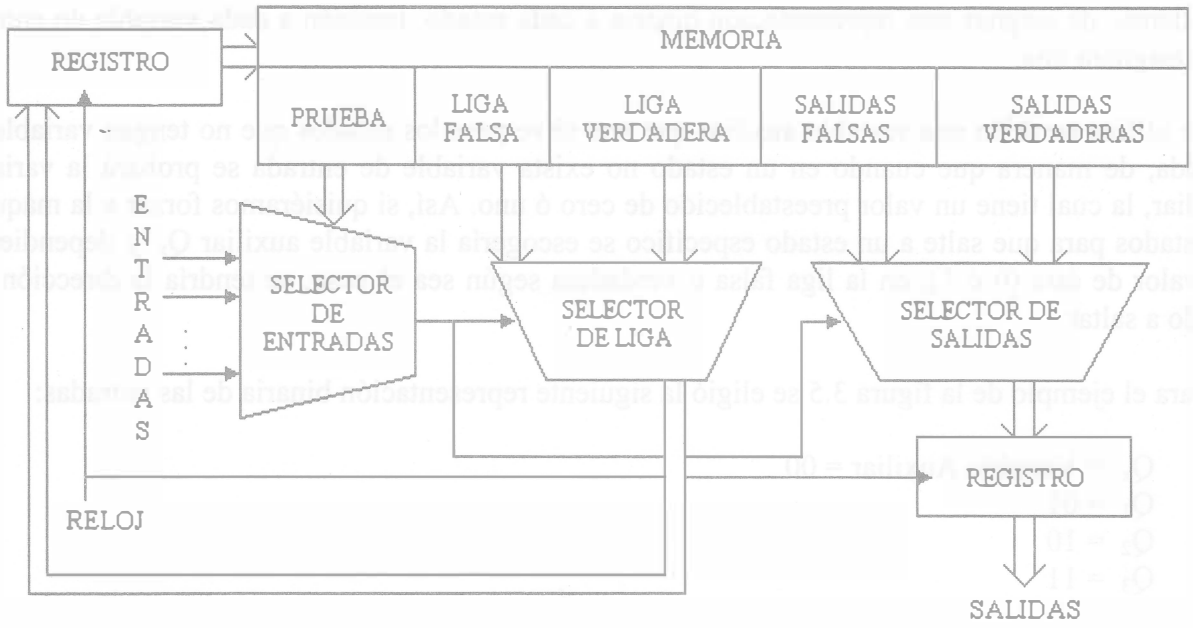
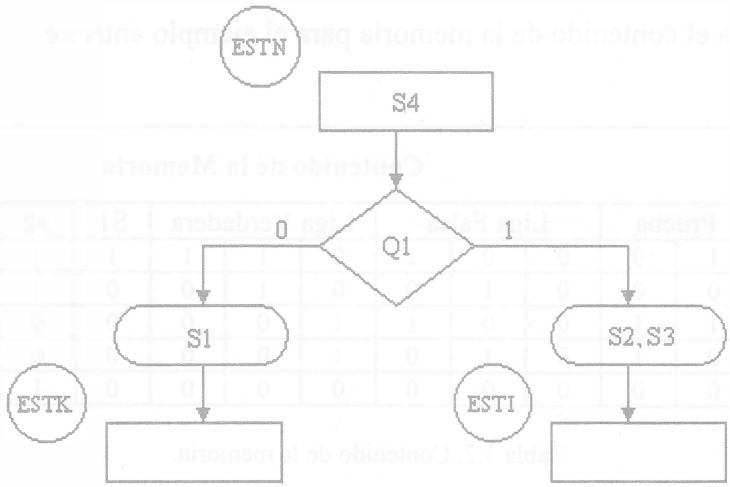


Figura 3.6. Diagrama de bloques del direccionamiento entrada-estado con salidas condicionales.

Como ejemplo se propone la siguiente carta ASM. Además, se muestra el contenido de la memoria para el estado ESTN.



			Salidas Falsas				Salidas Verdaderas			
Prueba	Liga Falsa	Liga Verdadera	S_1	S_2	S_3	S_4	S_1	S_2	S_3	S_4
Código Q1	ESTK	ESTI	1	0	0	1	0	1	1	1

Fig. 3.7. Representación en la memoria del estado ESTN.

3.3 DIRECCIONAMIENTO IMPLÍCITO

Este tipo de direccionamiento utiliza solamente un campo de liga. Una variable de entrada seleccionada por el campo de prueba, y VF, son las que deciden si se utiliza la dirección de liga (se carga el valor de liga en el contador) o no (se incrementa el contador en una unidad). La figura 3.8 muestra el esquema anterior.

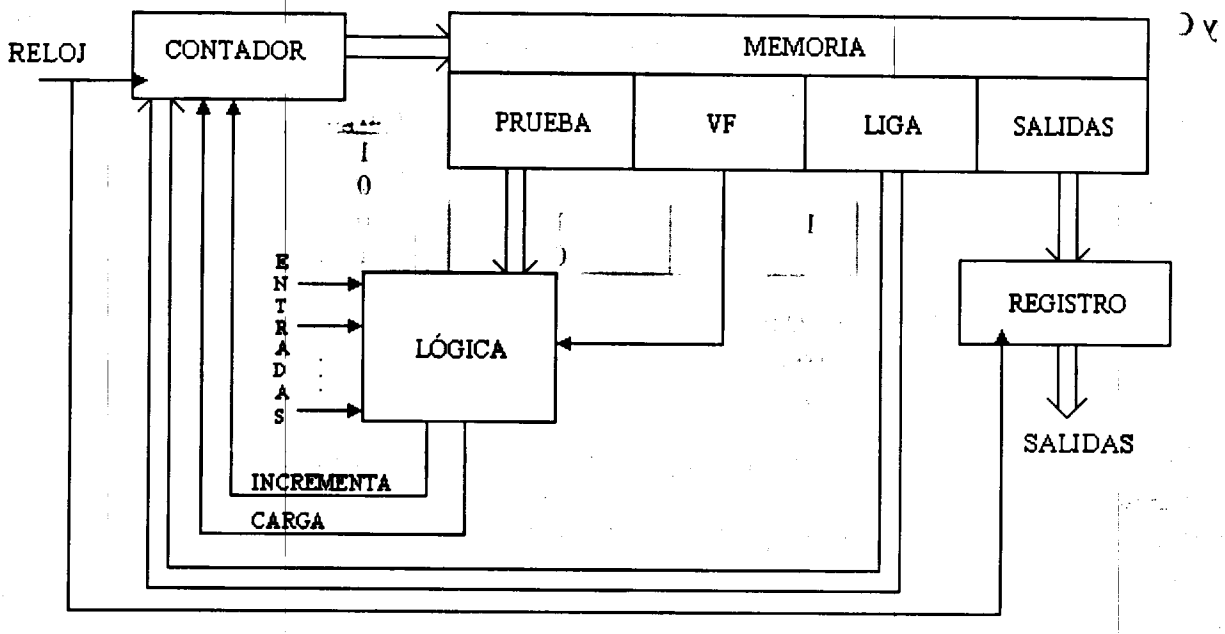


Figura 3.8. Diagrama de bloques del método de direccionamiento implícito.

El campo VF (Verdadero-Falso) sirve para indicarle a la lógica cuánto debe valer la variable de entrada, para así cargar en el contador el valor de la liga y hacer el salto. La figura 3.9 muestra parte de una carta ASM con este esquema.

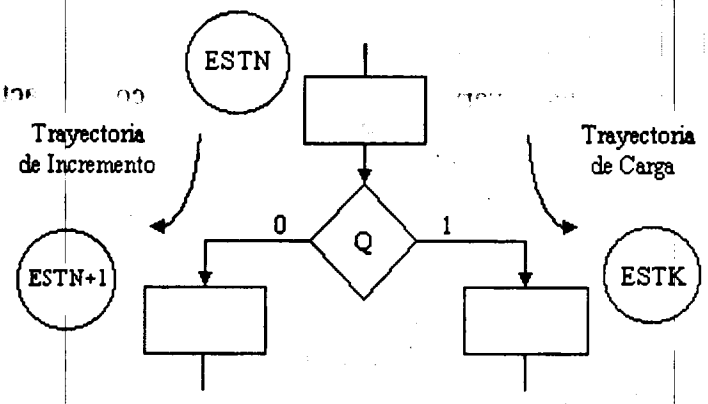


Figura 3.9. Trayectoria de incremento y carga para el estado ESTN.

Supongamos que nos encontramos en el estado ESTN y la variable a sensar es Q, si Q es igual a cero entonces el estado siguiente es la representación del estado presente más uno, por lo tanto, es necesario activar la señal de incremento. Si Q es igual a uno entonces el estado siguiente es el estado ESTK y su representación binaria, contenida en el campo de liga, es cargada en el contador. Para este ejemplo el campo VF es igual a uno ya que cuando Q es igual a uno se requiere hacer una carga en el contador.

La tabla 3.3 muestra la relación de VF y la variable de entrada con las líneas de INCREMENTA y CARGA.

VF	Q	Incrementa	Carga
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

Tabla 3.3. VF y Q se relacionan por medio de una XOR para generar la señal de INCREMENTA y por medio de una XNOR para generar la señal de CARGA.

La figura 3.10 ejemplifica la misma relación mediante un diagrama lógico.

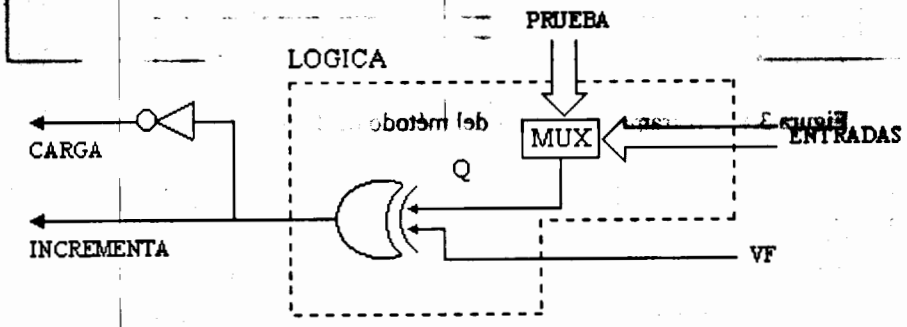


Figura 3.10. Bloque de lógica para el direccionamiento implícito.

El campo de prueba selecciona la variable a sensar que junto con VF activan la señal de carga o de incrementa por medio de una compuerta XOR.

Es necesario tomar precauciones al hacer la asignación binaria de los estados, porque debemos asegurar que por cada variable de entrada sensada existan dos estados siguientes: uno igual al estado presente más uno y el otro igual a cualquier otro valor. Por ejemplo, para la carta ASM de la figura 3.5 es necesario cambiar la asignación binaria de los estados, ya que para pasar del estado EST2 al estado EST1 ó del estado EST2 al estado EST4 se necesitan hacer cargas en ambos casos. La nueva asignación de estados se muestra en la figura 3.11.

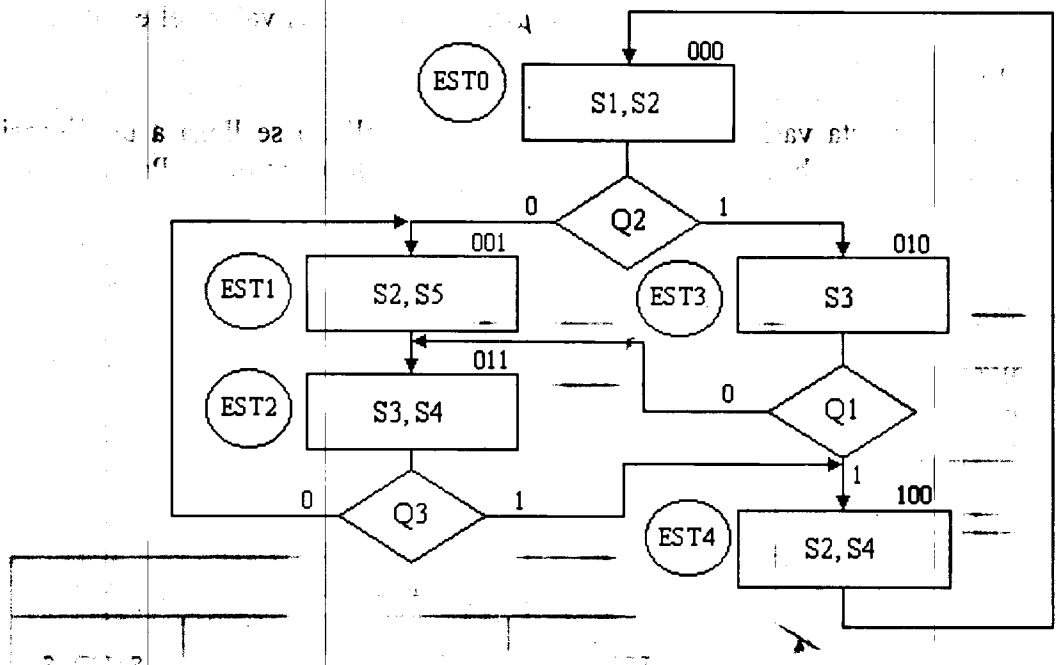


Figura 3.11. Nueva asignación binaria para el direccionamiento implícito.

La asignación binaria de las señales de entrada es idéntica a la asignación binaria propuesta en el ejemplo de direccionamiento entrada-estado. Y el valor asignado a la variable Q_x es igual a uno, es decir, presenta un nivel lógico alto.

La tabla 3.4 muestra el contenido de la memoria para la Figura 3.11.

Dirección de la Memoria	**Contenido de la Memoria**			
Estado Presente	Prueba	VF	Liga	Salidas $S_1S_2S_3S_4S_5$
000	10	1	010	1 1 0 0 0
001	00	1	011	0 1 0 0 1
010	01	1	100	0 0 1 0 0
011	11	0	001	0 0 1 1 0
100	00	1	000	0 1 0 1 0

Tabla 3.4. Contenido de la memoria.

Comparando este método con el método anterior se observa que se obtuvo una ganancia de dos bits en la memoria. En algoritmos de máquina de estados más complejos, donde el número de estados es muy grande, esta ganancia se hace más evidente.

La figura 3.12 muestra una variante del direccionamiento implícito. En lugar de usar el contador de la figura 3.8 se utilizan dos registros: un registro de liga en donde se guarda la dirección del

estado siguiente cuando hay un salto, y un registro de μPC que guarda el valor del estado presente más uno.

Haciendo más compleja esta variante de direccionamiento implícito se llega a un dispositivo llamado secuenciador que es la base fundamental de la Unidad de Control de Procesos de una computadora.

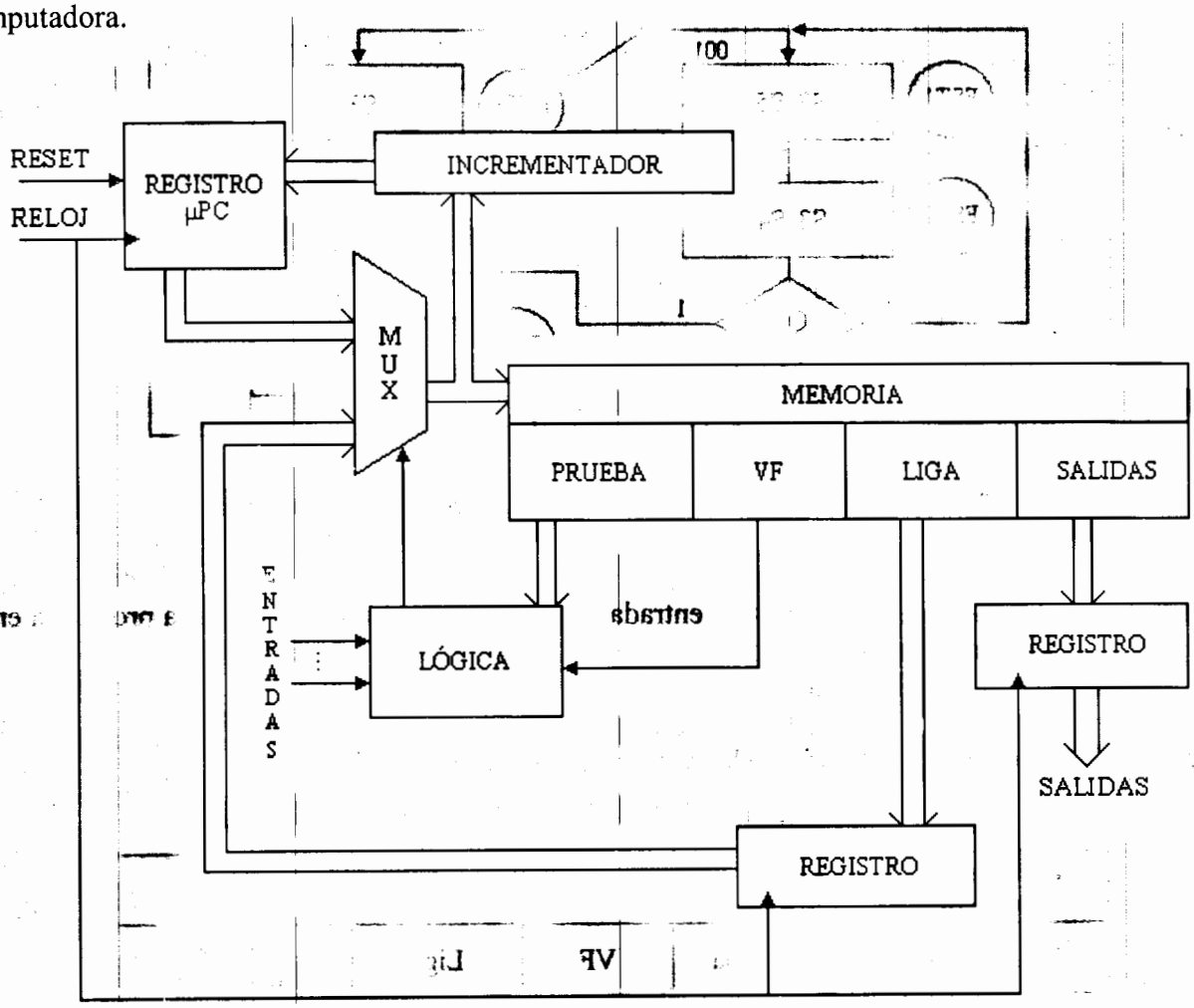


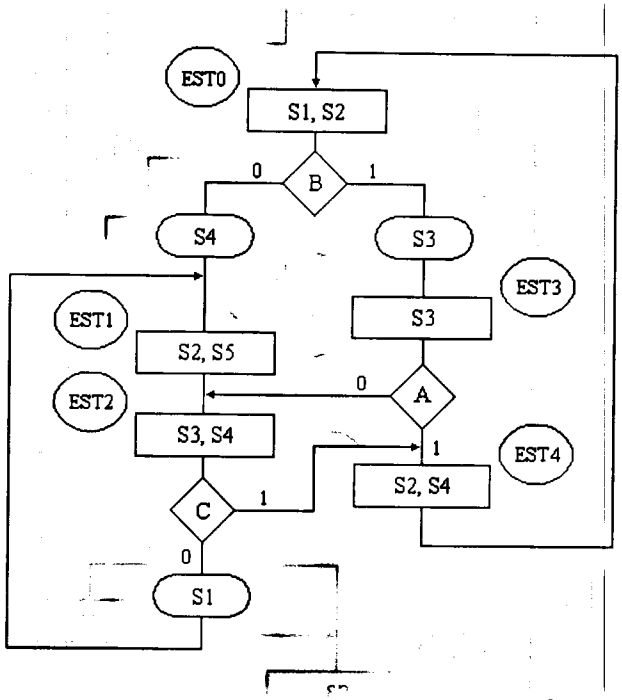
Figura 3.12. Variante del direccionamiento implícito.

PROBLEMAS

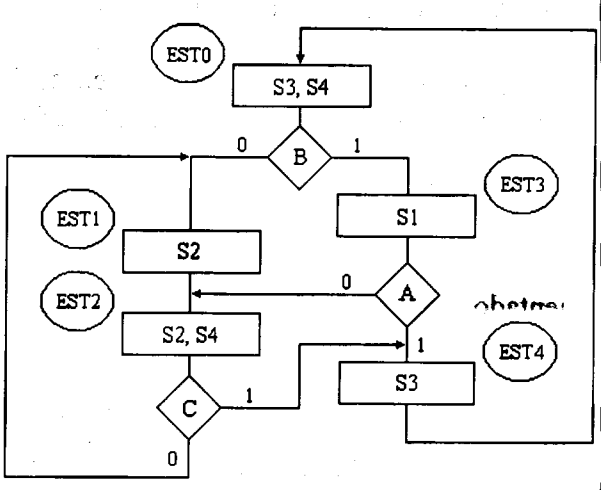
1. Para la siguiente carta ASM:

   a) Encuentre el contenido de la memoria utilizando el direccionamiento por trayectoria.
   b) Encuentre el contenido de la memoria utilizando el direccionamiento entrada-estado.

   No olvide especificar la asignación binaria de los estados y de las entradas.

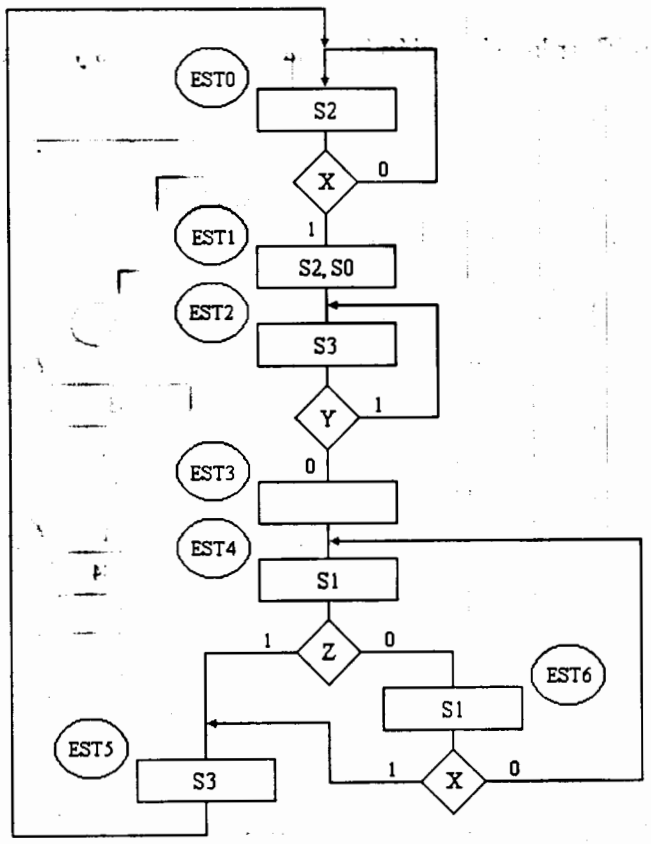


2. Para la siguiente carta ASM encuentre el contenido de la memoria usando el modo de direccionamiento implícito. No olvide especificar la asignación binaria de los estados y de las entradas.



3. Modifique el diagrama de bloques del direccionamiento implícito, mostrado en la figura 3.8, de manera que permita manejar salidas condicionales. Con base en su nuevo diagrama de bloques, obtenga el contenido de la memoria para la carta ASM del problema 1.

4. Encuentre el contenido de la memoria para instrumentar la siguiente carta ASM usando el direccionamiento por trayectoria.



5. Para la carta ASM del problema 4, encuentre el contenido de la memoria usando el direccionamiento entrada-estado. Especifique la asignación binaria de los estados y entradas.

6. Para la carta ASM del problema 4, encuentre el contenido de la memoria usando el direccionamiento implícito. Especifique la asignación binaria de los estados y entradas.

7. Codifique la carta ASM del problema 4 usando el lenguaje de descripción de hardware Verilog HDL.

8. Diseñe el incrementador mostrado en el diagrama de bloques de la variante del direccionamiento implícito (figura 3.12).

CAPÍTULO IV

CONSTRUCCIÓN DE MÁQUINAS DE ESTADOS USANDO SECUENCIADORES

4.1 EL SECUENCIADOR BÁSICO

En el capítulo anterior se vio el diseño de máquinas de estados usando memorias y dispositivos tales como contadores. Para el diseño de los módulos de control de una computadora se requieren máquinas de estados que sean capaces de ejecutar algoritmos más complejos. Haciendo modificaciones y agregando componentes a la variante del direccionamiento implícito se pueden crear máquinas de estados que efectúen cartas ASM con llamadas a subrutinas, estructuras DO WHILE, iteraciones tipo FOR, entre otras. Los dispositivos que son capaces de efectuar este tipo de operaciones son llamados **secuenciadores**.

La figura 4.1 muestra el diagrama de bloques de un secuenciador básico. Como puede observar en el diagrama, la dirección del estado siguiente, dada por el bus Y, puede venir de dos lugares posibles: 1) del registro μPC, ó 2) de la entrada D.

1. El registro de micro-programa (μPC) contiene la dirección del estado presente más uno, es decir, la dirección que se encuentra a la salida del multiplexor es incrementada en una unidad y cargada en este registro en el siguiente ciclo de reloj.

2. En la entrada D se introduce una dirección de salto. Esta dirección puede venir de tres lugares diferentes: del campo de liga, del registro de transformación o del registro de interrupciones.

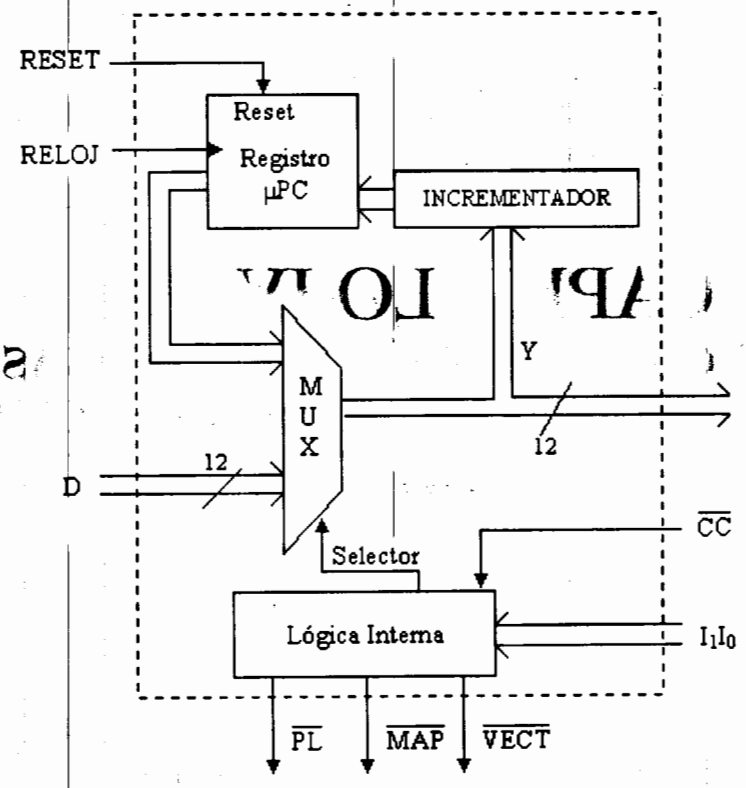


Figura 4.1. Diagrama de bloques del secuenciador básico.

El secuenciador cuenta con una lógica interna que se encarga de generar las señales que controlan al multiplexor. Dependiendo de la instrucción dada por las líneas I_1 e I_0 y de la línea CC, la lógica es capaz de seleccionar entre la salida del registro μPC o la entrada D. Dicha salida direcciona una memoria que contiene el estado siguiente del algoritmo de la máquina de estados.

La lógica interna también genera las líneas PL, MAP y VECT, las cuales seleccionan unos registros cuyas salidas están conectadas a la entrada D del secuenciador. De esta forma la dirección de salto puede venir de tres lugares distintos. Esta característica se utilizará cuando se diseñe la unidad central de procesos (UCP), como se verá más adelante.

La figura 4.2 muestra las señales de entrada y salida del secuenciador.

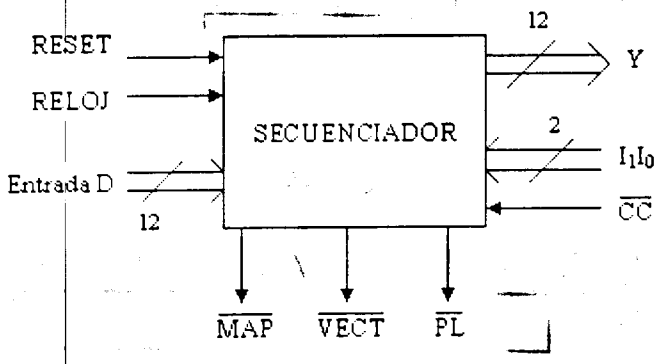


Figura 4.2. Secuenciador básico.

A continuación se muestran las instrucciones que el secuenciador puede ejecutar y su representación en carta ASM.

4.2 INSTRUCCIONES PARA EL SECUENCIADOR

4.2.1 CONTINÚA (C)

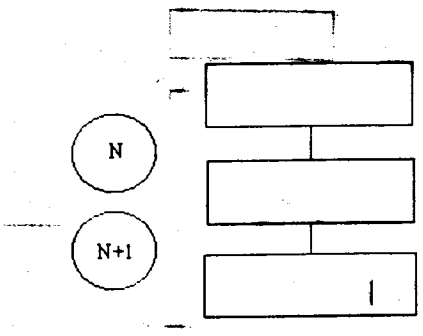


Figura 4.3. Representación en notación ASM de la instrucción continúa.

En la instrucción continúa la dirección del estado siguiente la proporciona el registro μPC.

4.2.2 SALTO CONDICIONAL (SCO)

En esta instrucción se revisa el valor de la línea $\overline{CC}$, si es igual a uno, la dirección del estado siguiente la proporciona el registro μPC; si es igual a cero, la dirección del estado siguiente, contenida en el registro seleccionado por $\overline{PL}$, ingresa a través de la entrada D.

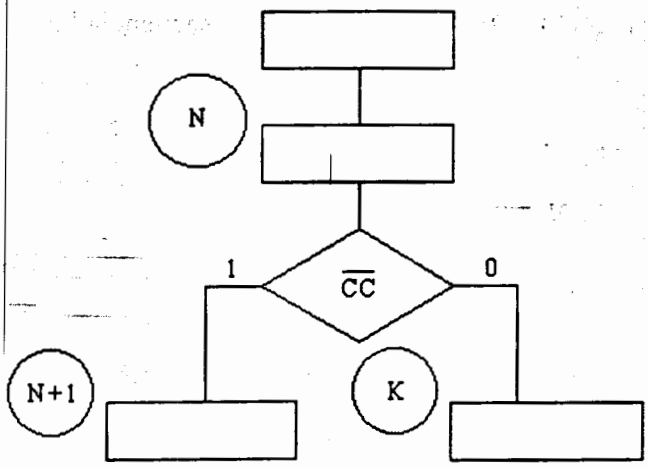


Figura 4.4. Representación en notación ASM de la instrucción SCO.

4.2.3 SALTO DE TRANSFORMACIÓN (ST)

La dirección del estado siguiente se obtiene del registro seleccionado por la línea de $\overline{MAP}$. Este registro también está conectado a la entrada D. Aquí se introduce una nueva notación de carta ASM: un rombo con varias bifurcaciones. La bifurcación que se elija dependerá del contenido del registro seleccionado por $\overline{MAP}$.

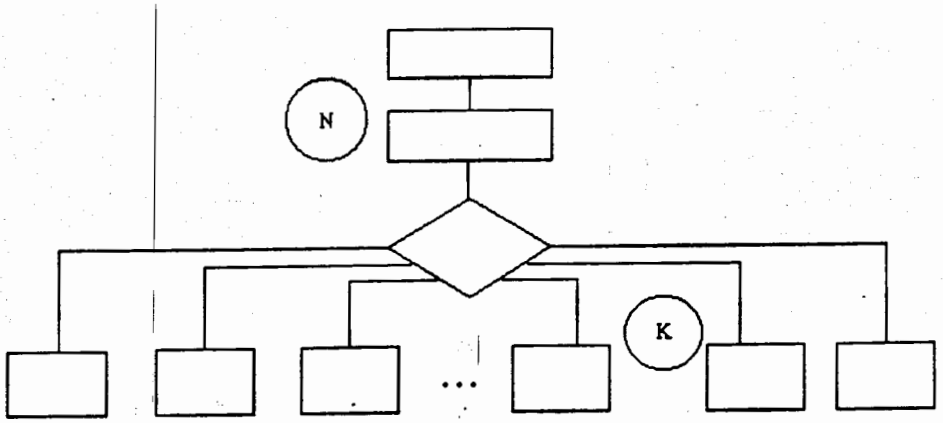


Figura 4.5. Representación en notación ASM de la instrucción ST.

4.2.4 SALTO CONDICIONAL USANDO LA DIRECCIÓN DE LAS INTERRUPCIONES (SCI)

En esta instrucción se revisa el valor de $\overline{CC}$, si es igual a uno, la dirección del estado siguiente proviene del registro μPC; si es igual a cero, la dirección del estado siguiente, contenida en el registro seleccionado por $\overline{VECT}$, ingresa a través de la entrada D.

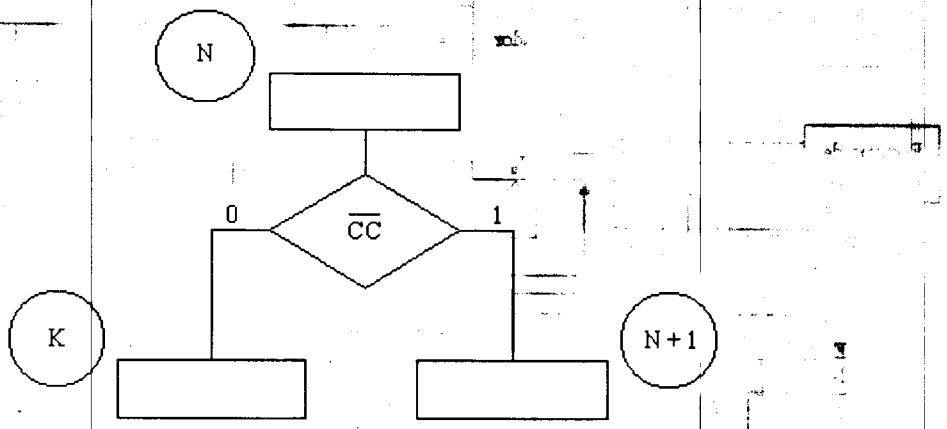


Figura 4.6. Representación en notación ASM de la Instrucción SCI.

La lógica interna del secuenciador se construye a partir de la siguiente tabla.

Entradas			Salidas				
I1	I0	$\overline{CC}$	Selector	$\overline{PL}$	MAP	$\overline{VECT}$	Y
0	0	0	0	1	1	1	μPC
0	0	1	0	1	1	1	μPC
0	1	0	1	0	1	1	Entrada D
0	1	1	0	0	1	1	μPC
1	0	0	1	1	0	1	Entrada D
1	0	1	1	1	0	1	Entrada D
1	1	0	1	1	1	0	Entrada D
1	1	1	0	1	1	0	μPC

Tabla. 4.1. Entradas y salidas de la lógica interna del secuenciador.

4.3 SECUENCIADORES Y MEMORIAS

La figura 4.7 muestra el diagrama de bloques de un secuenciador conectado a una memoria.

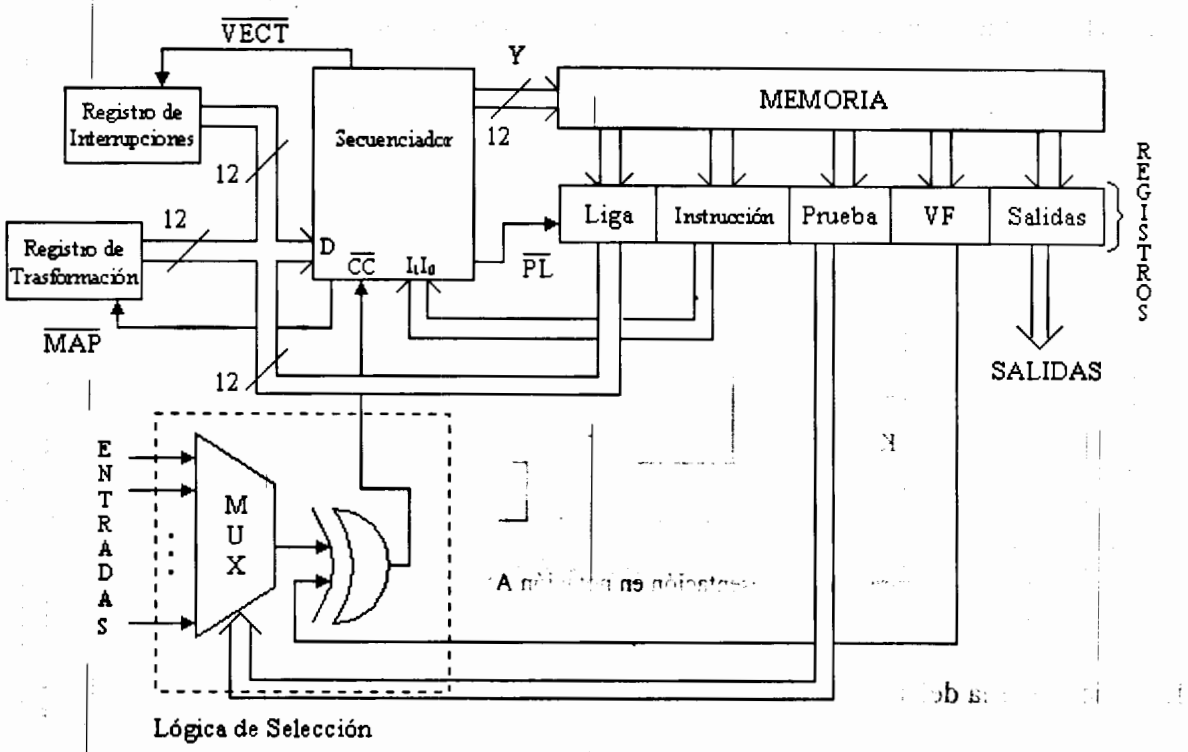


Figura 4.7. Construcción de cartas ASM usando un secuenciador básico y memorias.

Usando este secuenciador, el contenido de la memoria para la carta ASM de la figura 3.11 queda como se muestra en la tabla 4.2.

Dirección de la Memoria			**Contenido de Memoria**												
Estado Presente			Liga			Micro Instrucción		Prueba		VF	Salidas S1	S2	S3	S4	S5
0	0	0	0	1	0	0	1	1	0	1	1	1	0	0	0
0	0	1	0	1	1	0	1	0	0	1	0	1	0	0	1
0	1	0	1	0	0	0	1	0	1	1	0	0	1	0	0
0	1	1	0	0	1	0	1	1	1	0	0	0	1	1	0
1	0	0	0	0	0	0	1	0	0	1	0	1	0	1	0

Tabla 4.2. Contenido de la memoria para la figura 3.11 usando el secuenciador.

4.4 IMPLANTACIÓN DE CARTAS ASM USANDO SECUENCIADORES

En la siguiente figura se presenta una carta ASM en donde se hace uso de todas las instrucciones que este secuenciador básico puede ejecutar.

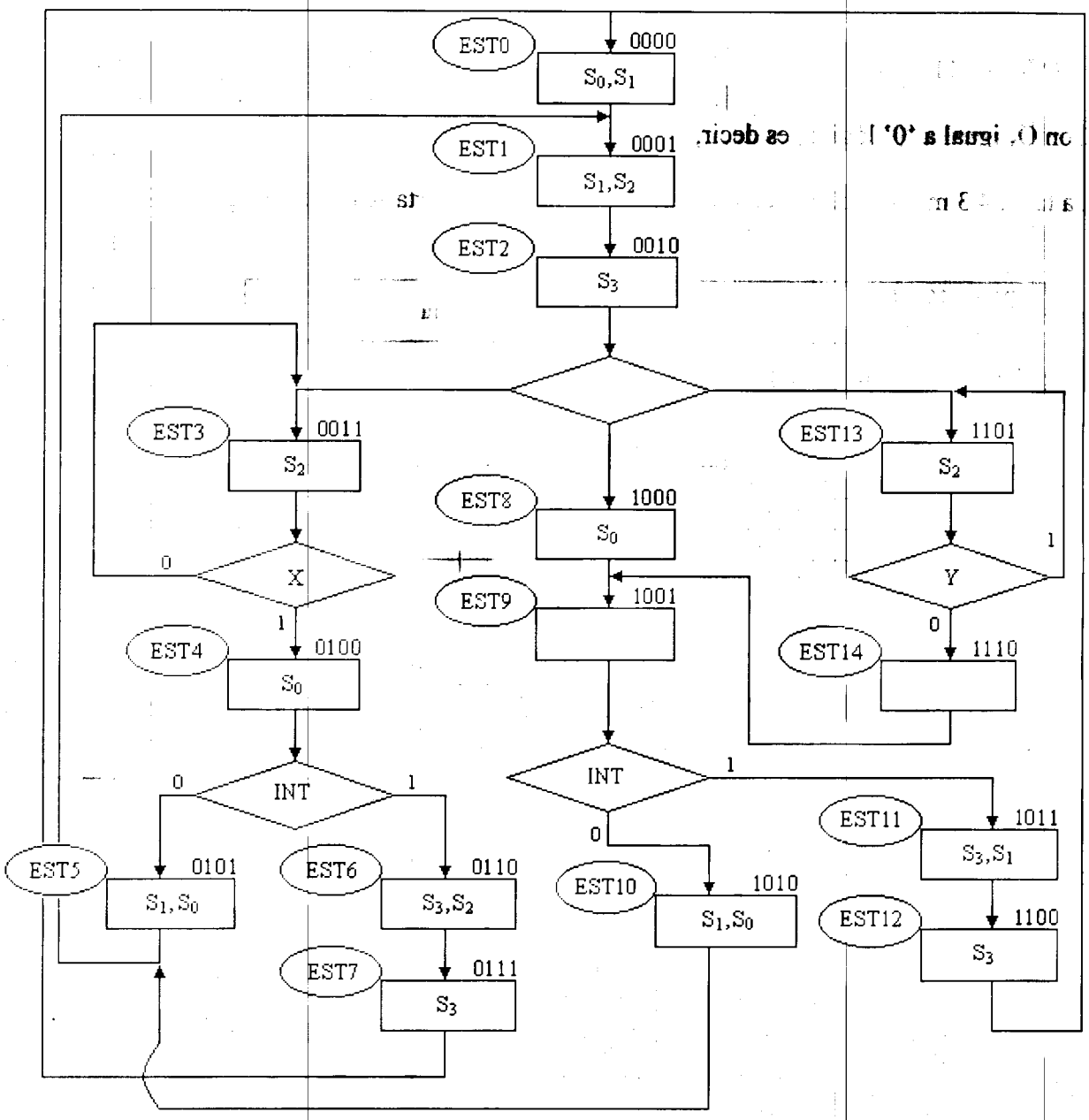


Figura 4.8. Carta ASM.

En el estado EST2 la dirección del estado siguiente está determinada por el contenido del registro de transformación, seleccionado cuando el secuenciador ejecuta la instrucción ST. En el estado

EST4, la dirección del estado siguiente la proporciona el registro de interrupciones o el registro μPC dependiendo del valor de la entrada INT.

La asignación binaria de las variables de entrada para la carta ASM es la siguiente:

Q_x = 00
X = 01
Y = 10
INT = 11

Con Q_x igual a '0' lógico, es decir, Q_x presenta un nivel lógico bajo.

La tabla 4.3 muestra el contenido de la memoria para la carta ASM de la figura 4.8.

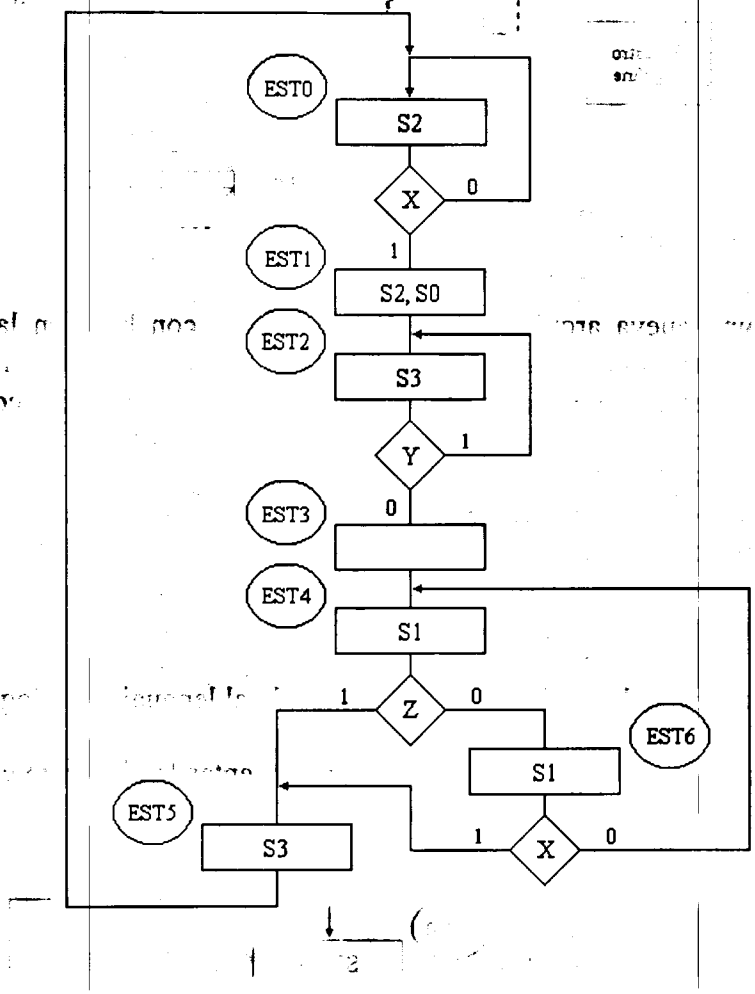
Dirección de la Memoria	**Contenido de la Memoria**					
Estado Presente	Liga	Micro Instrucción	Prueba	VF	$S_0S_1S_2S_3$	Mnemónico de la Instrucción
0000	0000	00	00	0	1 1 0 0	C
0001	0000	00	00	0	0 1 1 0	C
0010	0000	10	00	0	0 0 0 1	ST
0011	0011	01	01	0	0 0 1 0	SCC
0100	0000	11	11	1	1 0 0 0	SCI
0101	0001	01	00	0	1 1 0 0	SCC
0110	0000	00	00	0	0 0 1 1	C
0111	0000	01	00	0	0 0 0 1	SCC
1000	0000	00	00	0	1 0 0 0	C
1001	0000	11	11	1	0 0 0 0	SCI
1010	0001	01	00	0	1 1 0 0	SCC
1011	0000	00	00	0	0 1 0 1	C
1100	0000	01	00	0	0 0 0 1	SCC
1101	1101	01	10	1	0 0 1 0	SCC
1110	1001	01	00	0	0 0 0 0	SCC

Tabla 4.3. Contenido de la memoria para la carta ASM de la figura 4.8.

PROBLEMAS

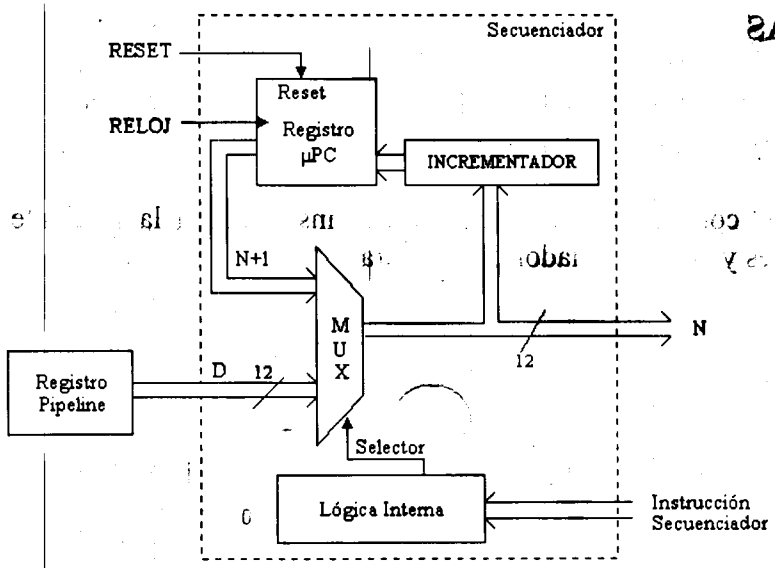
1. Diseñe la lógica interna del secuenciador mostrado en la figura 4.1.

2. Encuentre el contenido de la memoria para instrumentar la siguiente carta ASM. Utilice las instrucciones y el secuenciador de la figura 4.1.



3. En la implantación de máquinas de estados usando memorias se utiliza una variante del direccionamiento implícito denominado secuenciador. La siguiente figura muestra un secuenciador capaz de ejecutar dos instrucciones: continúa y salto. Cuando se ejecuta la instrucción de continúa, el multiplexor selecciona el contenido del registro μPC con la dirección del estado siguiente (N+1, donde N es el estado presente). Cuando se ejecuta la instrucción de salto, el multiplexor selecciona la dirección proveniente del registro Pipeline.

   Nota: La señal 'Instrucción Secuenciador' le indica a la lógica interna qué instrucción estamos ejecutando, de esta manera, la lógica interna puede generar las señales de control adecuadas para el multiplexor.

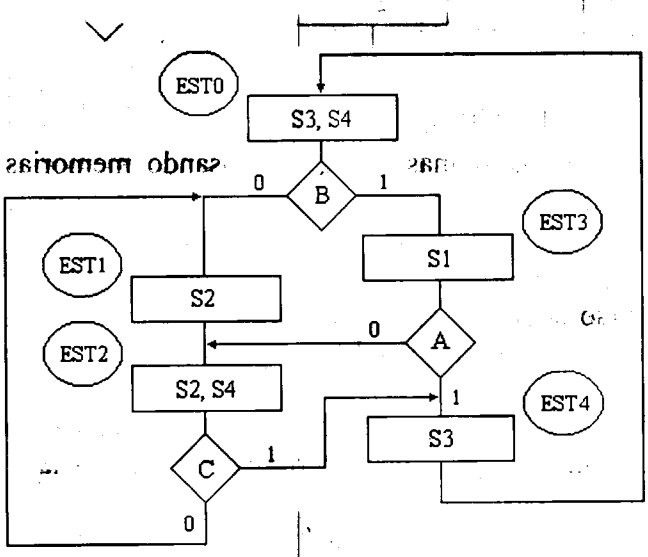


a) Proponga una nueva arquitectura de secuenciador, con base en la figura anterior, que permita ejecutar, adicionalmente, las siguientes dos instrucciones: 1) una instrucción que guarde el contenido del registro µPC y haga un salto a la dirección contenida en el registro Pipeline (análogamente a un llamado a subrutina); y 2) una instrucción que recupere la dirección guardada por la instrucción anterior, la cual servirá como la dirección del estado siguiente (análogamente a un retorno de subrutina). Especifique el funcionamiento de la lógica interna para las dos nuevas instrucciones utilizando una tabla de verdad.

b) ¿Qué dispositivos se tendrían que utilizar para poder hacer este tipo de llamadas de forma anidada, es decir, dentro de una subrutina poder llamar a otra subrutina?

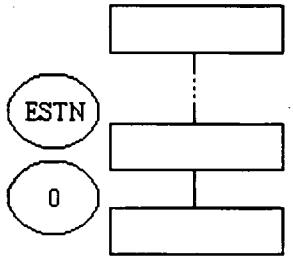
4. Codifique el secuenciador de la figura 4.1 utilizando el lenguaje Verilog HDL.

5. Encuentre el contenido de la memoria para instrumentar la siguiente carta ASM. Utilice las instrucciones y el secuenciador de la figura 4.1.

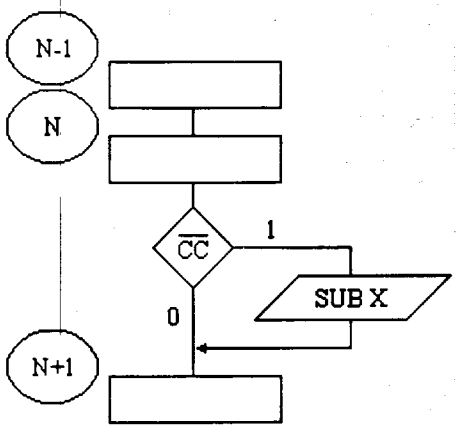


6. Modifique la estructura interna del secuenciador mostrado en la figura 4.1 para que pueda ejecutar, adicionalmente, las siguientes instrucciones.

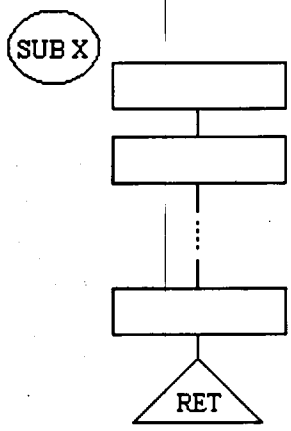
   a) Salto a cero (SC). En esta instrucción se hace un salto del estado N al estado cero, el cual tiene una representación binaria de ceros.



   b) Salto condicional a subrutina (SCS). En esta instrucción, estando en el estado N, se pregunta por la variable de entrada $\overline{CC}$. Si $\overline{CC}$ es igual a uno, la dirección del estado siguiente, procedente del registro de liga, ingresa a través de la entrada D del secuenciador. Esta dirección representa la dirección de inicio de una subrutina. La dirección de regreso de la subrutina, es decir, el estado N+1, debe ser guardado en una pila. Se podrán tener hasta 16 subrutinas anidadas. En caso de que $\overline{CC}$ valga cero, la dirección del estado siguiente estará dada por el μPC.
   c) Regreso de subrutina (RS). La dirección de regreso de la subrutina, el estado N+1, es obtenido de la pila.



Salto condicional a subrutina



Regreso de subrutina

CAPÍTULO V
COMPONENTES BÁSICOS DE UN PROCESADOR

5.1 ESTRUCTURA BÁSICA DE UNA COMPUTADORA

En este capítulo se diseñarán algunos de los componentes que integran una computadora. En la figura 5.1 se muestra el diagrama de bloques general de una computadora.

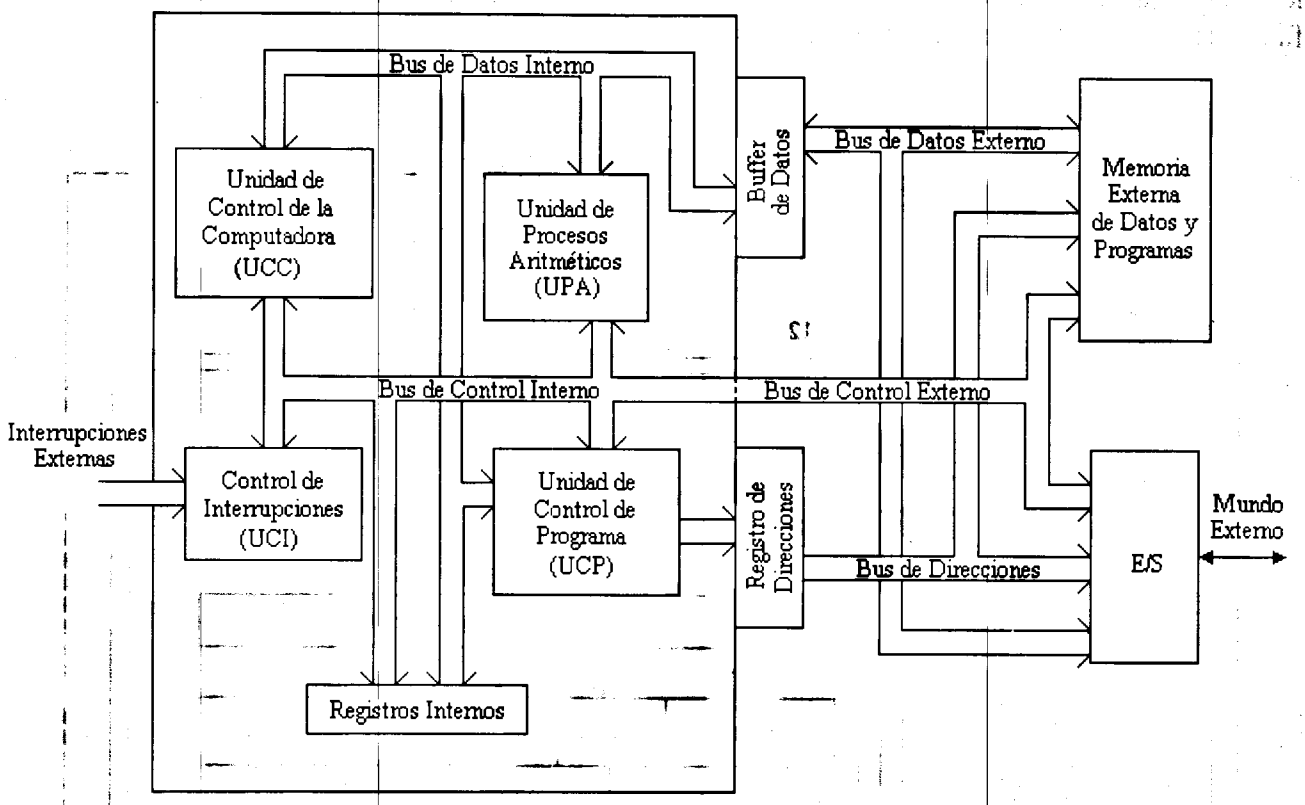


Figura 5.1. Estructura básica de una computadora.

Como se observa en el diagrama de bloques, una computadora está constituida internamente por cinco bloques básicos:

1) *Unidad de Control de la Computadora* (*UCC*). Se encarga de enviar las señales de control a los demás elementos de la computadora.
2) *Unidad de Procesos Aritméticos* (*UPA*). En ella se realizan todas las operaciones lógico aritméticas.
3) *Unidad de Control de Programa* (*UCP*). Calcula la dirección de la siguiente instrucción a ser ejecutada.
4) *Unidad de Registros Internos*. Conjunto de registros capaces de almacenar información.
5) *Unidad de Control de Interrupciones* (*UCI*). Se encarga del manejo de las interrupciones externas.

A continuación se describe cada uno de estos componentes.

5.2 UNIDAD DE CONTROL DE LA COMPUTADORA (UCC)

La UCC se encarga de decodificar las instrucciones en ensamblador y de ejecutar las micro-operaciones necesarias (representadas por medio de cartas ASM) para llevarlas a cabo. Esto es, por cada instrucción en lenguaje ensamblador, serán activadas secuencialmente una serie de señales que le indican a los diferentes componentes de la arquitectura la operación a realizar. La parte fundamental de la UCC es el secuenciador, como se muestra en la figura 5.2.

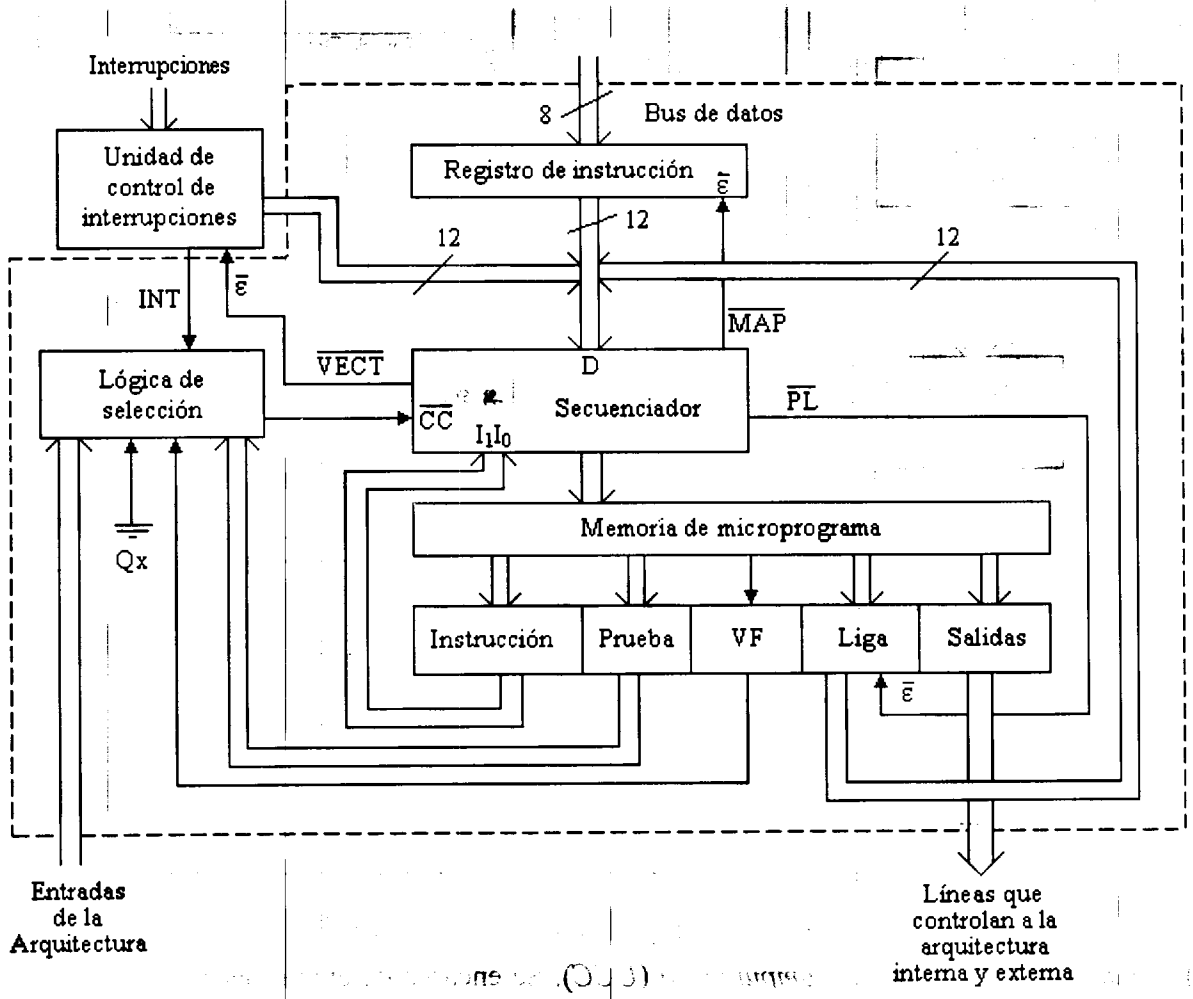


Figura 5.2. Unidad de control de la computadora (UCC).

Como se vio en el capítulo anterior, la dirección de salto en la entrada D del secuenciador puede venir de tres lugares diferentes: del registro de instrucción, de la unidad de control de interrupciones, ó del campo de liga de la memoria de microprograma. La figura 5.2 ilustra esta configuración.

El tamaño de la palabra de las instrucciones codificadas es de 8 bits. Este número es cargado en el registro de instrucciones y extendido de 8 a 12 bits mediante la adición de cuatro ceros a la

derecha. Este nuevo valor es la dirección en la memoria de microprograma en donde comienzan las micro-operaciones que ejecuta esta instrucción.

En el campo de salidas de la memoria de microprograma se tienen las líneas que controlan tanto a la arquitectura interna como a la externa, las cuales se activan según la instrucción a ejecutar.

Por otra parte, las entradas de la arquitectura indican el estado en el que se encuentra tanto la arquitectura interna como la externa, y sirven para que el secuenciador pueda tomar decisiones de acuerdo a ciertos criterios. Estas entradas son seleccionadas en el bloque de lógica de selección por medio del campo de prueba. La línea de INT también se conecta a la lógica de selección para revisar si existe alguna petición de interrupción.

Otra forma alternativa de diseñar la UCC es utilizando los lenguajes de descripción de hardware como Verilog HDL, VHDL ó AHDL. Utilizando alguno de estos lenguajes y el código de la instrucción que se desea ejecutar, es posible describir los pasos requeridos para ejecutar dicha instrucción. En el capítulo 6 se mostrará un ejemplo de cómo hacer esto.

5.3 UNIDAD DE PROCESOS ARITMÉTICOS (UPA)

La unidad de procesos aritméticos (UPA) se encarga de realizar las operaciones lógico aritméticas básicas. Para ello, cuenta con una unidad lógico aritmética que le permite hacer sumas, restas, y operaciones lógicas AND, OR exclusiva, OR exclusiva negada, entre otras. La UPA también cuenta con un registro de corrimiento auxiliar para guardar valores intermedios que posteriormente operará.

La figura 5.3a muestra una UPA de ocho bits basada en una UPA fabricada por AMD de cuatro bits (D2901). Este dispositivo, como en el caso del secuenciador, no existen físicamente en la actualidad, sino como un módulo en software que puede ser integrado en un sistema digital. El apéndice B muestra la realización de éstos módulos usando el lenguaje de descripción de hardware Verilog HDL en el entorno MAX+PLUS II.

Como se observa en la figura 5.3a, las fuentes de la unidad lógico aritmética pueden venir de cinco lugares diferentes: de la entrada A, de la entrada B, del registro de corrimiento auxiliar Q, de la entrada de datos D y el valor de cero.

El resultado de la operación de la ALU puede ser desplazado a la derecha o a la izquierda antes de ser guardado en alguno de los registros de destino. Estos registros de destino son el registro de corrimiento Y_{upa} y el registro de corrimiento Q. Además, observe que la señal DUPA habilita o no la carga de un resultado o de un corrimiento en los registros de destino.

El diagrama de bloques de la UPA se muestra a continuación.

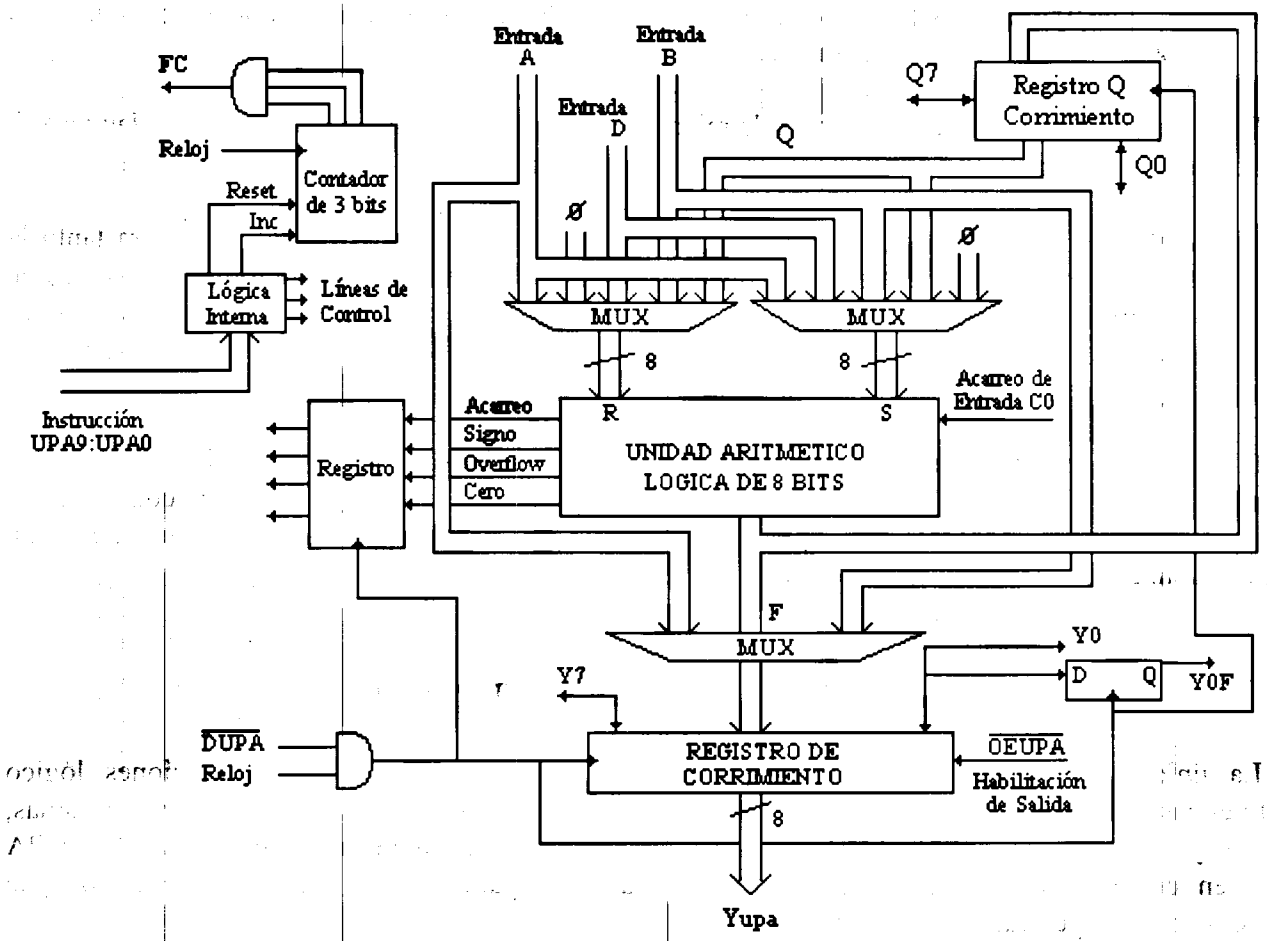


Figura 5.3a. Diagrama de bloques de la unidad de procesos aritméticos (UPA).

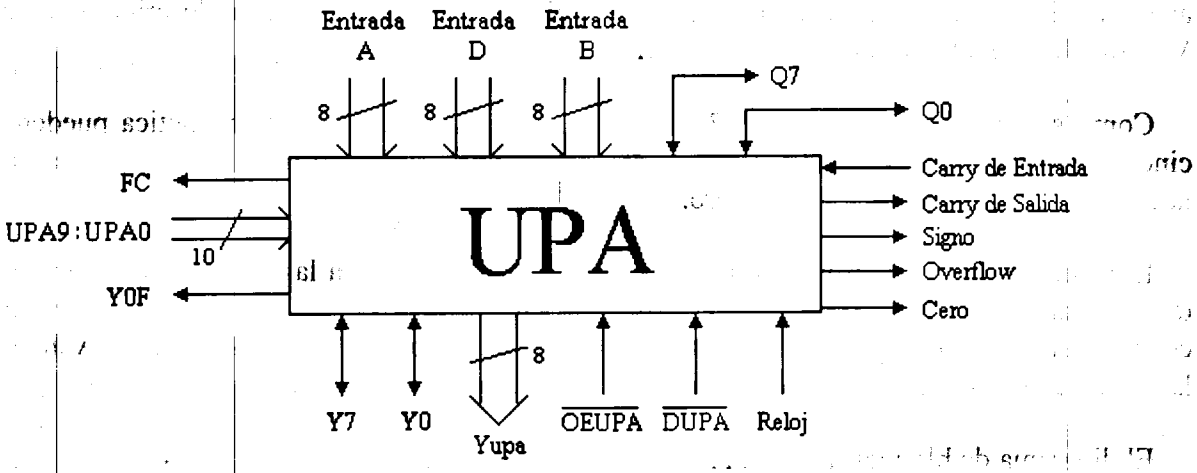


Figura 5.3b. Unidad de procesos aritméticos (UPA).

Las tablas 5.1A, 5.1B y 5.1C muestran la relación existente entre las líneas de control de la UPA (UPA9:UPA0) y las operaciones que ésta puede ejecutar.

En particular, la tabla 5.1A presenta la selección de las fuentes de la ALU, la tabla 5.1B las operaciones que ejecuta la ALU, y la tabla 5.1C los destinos y desplazamientos del resultado obtenido por la ALU.

Líneas de Control				*Fuentes de la ALU y Control del Contador de 3 Bits*			
UPA3	UPA2	UPA1	UPA0	R	S	ResetContador	IncrementaContador
0	0	0	0	A	0	0	0
0	0	0	1	A	B	0	0
0	0	1	0	Q	0	0	0
0	0	1	1	B	0	0	0
0	1	0	0	0	A	0	0
0	1	0	1	D	A	0	0
0	1	1	0	D	Q	0	0
0	1	1	1	D	0	0	0
1	0	0	0	D	B	0	0
1	0	0	1	Q	B	0	0
1	0	1	0	X	X	1	0 *
1	0	1	1	X	X	0	1 *
1	1	0	0	Q	A	0	0

X - Significa "no importa"

Tabla 5.1A. Selección de las fuentes de la ALU y líneas de control para el contador de 3 bits.

Líneas de Control			*Funciones de la ALU*	
UPA6	UPA5	UPA4		
0	0	0	$R + S + Cin$	(Suma)
0	0	1	$S - R - \overline{Cin}$	(Resta)
0	1	0	$R - S - \overline{Cin}$	(Resta)
0	1	1	$R \vee S$	(Or)
1	0	0	$R \wedge S$	(And)
1	0	1	$\overline{R} \wedge S$	(Complemento y And)
1	1	0	$R \oplus S$	(Or exclusivo)
1	1	1	$\overline{R \oplus S}$	(Nor exclusivo)

Tabla 5.1B. Operaciones de la ALU.

* Cuando se ejecuta alguna de estas operaciones, los resultados almacenados en los registros Y_{upa} y Q no deben ser alterados, por lo tanto, es necesaria la activación de la señal $\overline{DUPA}$.

Líneas de Control			*Destinos y Desplazamientos*	
UPA9	UPA8	UPA7	Yupa	Q
0	0	0	F	F
0	0	1	F	-
0	1	0	A	-
0	1	1	B	-
1	0	0	Yupa/2	-
1	0	1	-	Q/2
1	1	0	2Yupa	-
1	1	1	-	2Q

Tabla 5.1C. Destinos y desplazamientos de la UPA.

Por ejemplo, para realizar la operación lógica OR entre las entradas A y B, y colocar el resultado en el registro Y_{upa}, se necesitan activar las siguientes líneas.

1. Las fuentes A y B se seleccionan con UPA3 UPA2 UPA1 UPA0 = 0001
2. La función OR se selecciona con las líneas UPA6 UPA5 UPA4 = 011
3. El destino Y_{upa} se selecciona con las líneas UPA9 UPA8 UPA7 = 000

La siguiente figura muestra la activación de las señales de control de la UPA para efectuar la operación $Y_{upa}=A \vee B$ usando cartas ASM.

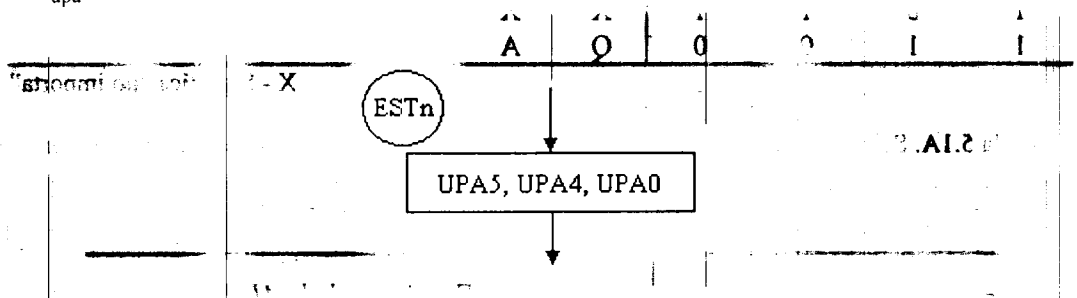


Figura 5.3c. Activación de las líneas de control en una carta ASM.

La UPA también tiene unas líneas de salida que reflejan el resultado de la última operación hecha por la ALU. La línea Z indica si el resultado fue cero; SIGNO indica el valor del bit más significativo; y OVR indica si hubo sobreflujo. También se cuenta con dos líneas de acarreo: uno de entrada y otro de salida.

5.4 REGISTROS INTERNOS

La computadora que diseñaremos requiere una serie de registros de 8 y 16 bits que tanto el usuario como el CPU pueden utilizar. Los registros de 8 bits, denominados acumuladores, sirven únicamente como dispositivos de almacenamiento. Los registros de 16 bits, denominados registros contadores, tienen mayor funcionalidad, pues además de servir como dispositivos de almacenamiento, permiten incrementar o decrementar el dato guardado.

5.4.1 REGISTROS ACUMULADORES

Los registros acumuladores de 8 bits están conectados directamente a las entradas de la UPA, de esta manera se pueden efectuar operaciones lógico aritméticas en forma directa.

La figura 5.4 muestra el diagrama de bloques del acumulador. Como puede observar, el acumulador está formado por un registro "latch" y por tres "transeivers". Los transeivers tienen la función de aislar o conectar el latch a los diferentes buses de datos del acumulador (A, B y C).

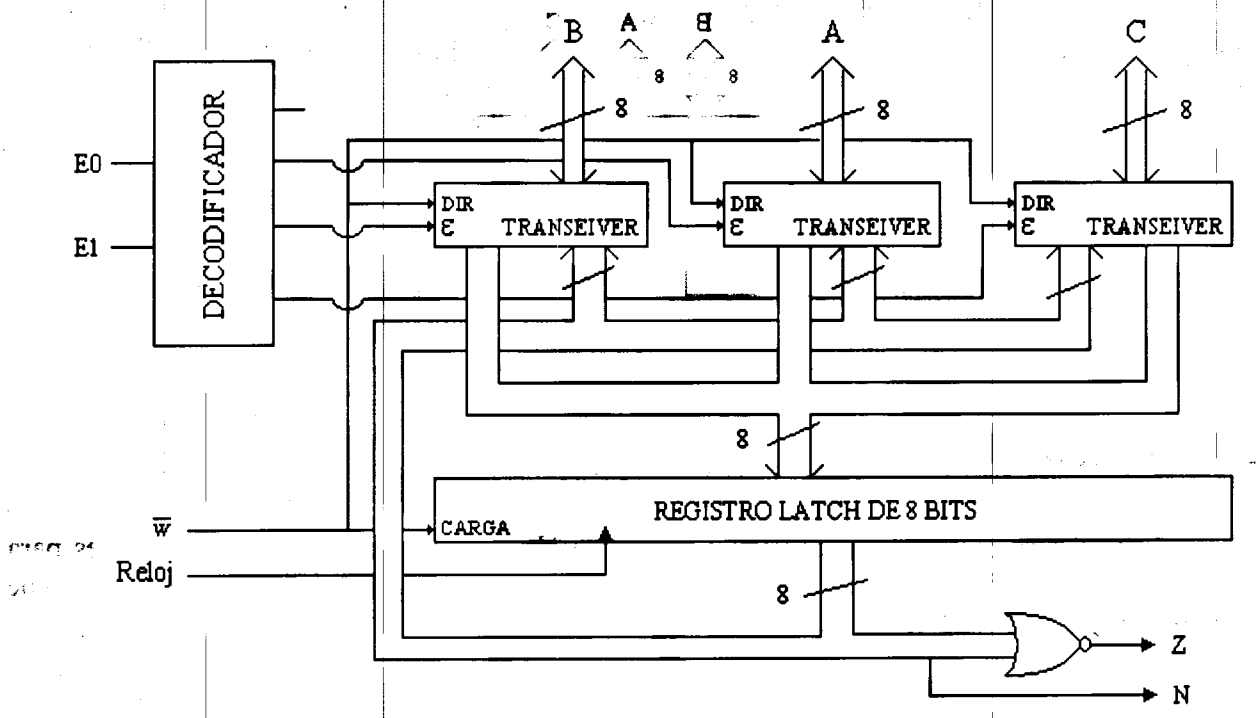


Figura 5.4. Diagrama de bloques del acumulador.

Las líneas de control E1 y E0 permiten seleccionar alguno de los buses de entrada conectados al acumulador. La tabla 5.2 muestra la relación existente entre estas líneas y el bus que seleccionan.

E1	*E0*	*Bus Seleccionado*
0	0	Ninguno
0	1	A
1	0	B
1	1	C

Tabla 5.2. Relación entre las señales de control E1:E0 y los buses que seleccionan.

La línea $\overline{w}$ habilita la carga de datos en el acumulador, de manera que si presenta un nivel lógico bajo (0), escribirá el dato del bus seleccionado en el latch. Si por el contrario, presenta un nivel lógico alto (1), entonces el registro latch estará habilitado sólo para lectura.

También existen dos señales de salida: la bandera de cero (Z) y la bandera de negativo (N). La bandera Z vale uno si el dato en el acumulador es cero, y vale cero en caso contrario. La bandera N refleja el signo del número guardado, dicho signo está dado por el bit más significativo del acumulador.

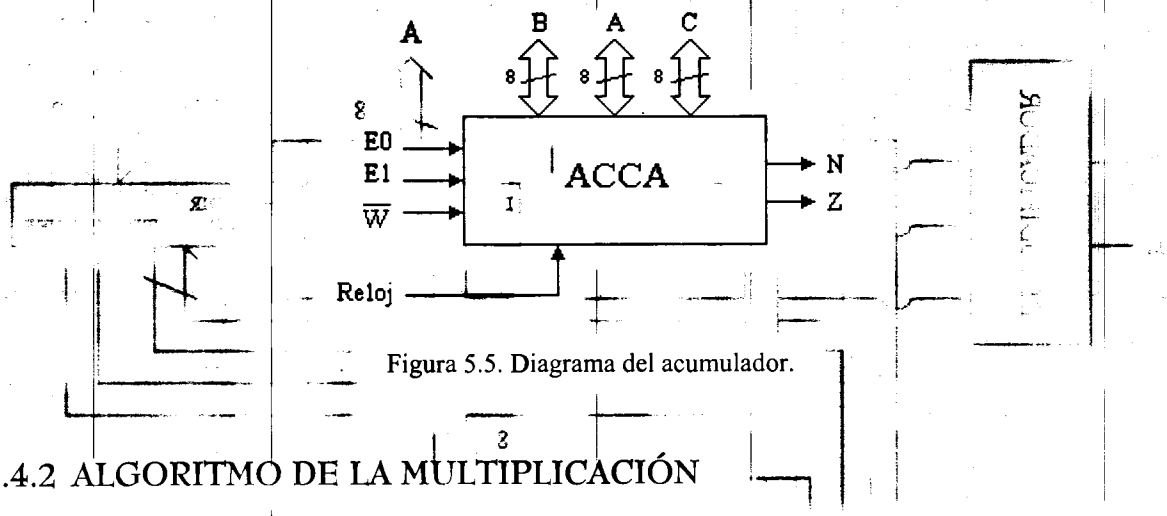


Figura 5.5. Diagrama del acumulador.

5.4.2 ALGORITMO DE LA MULTIPLICACIÓN

La siguiente figura muestra una configuración de la UPA con los registros acumuladores para efectuar la multiplicación de dos números. Tenga en cuenta que el algoritmo de multiplicación debe estar implementado en la máquina de estados.

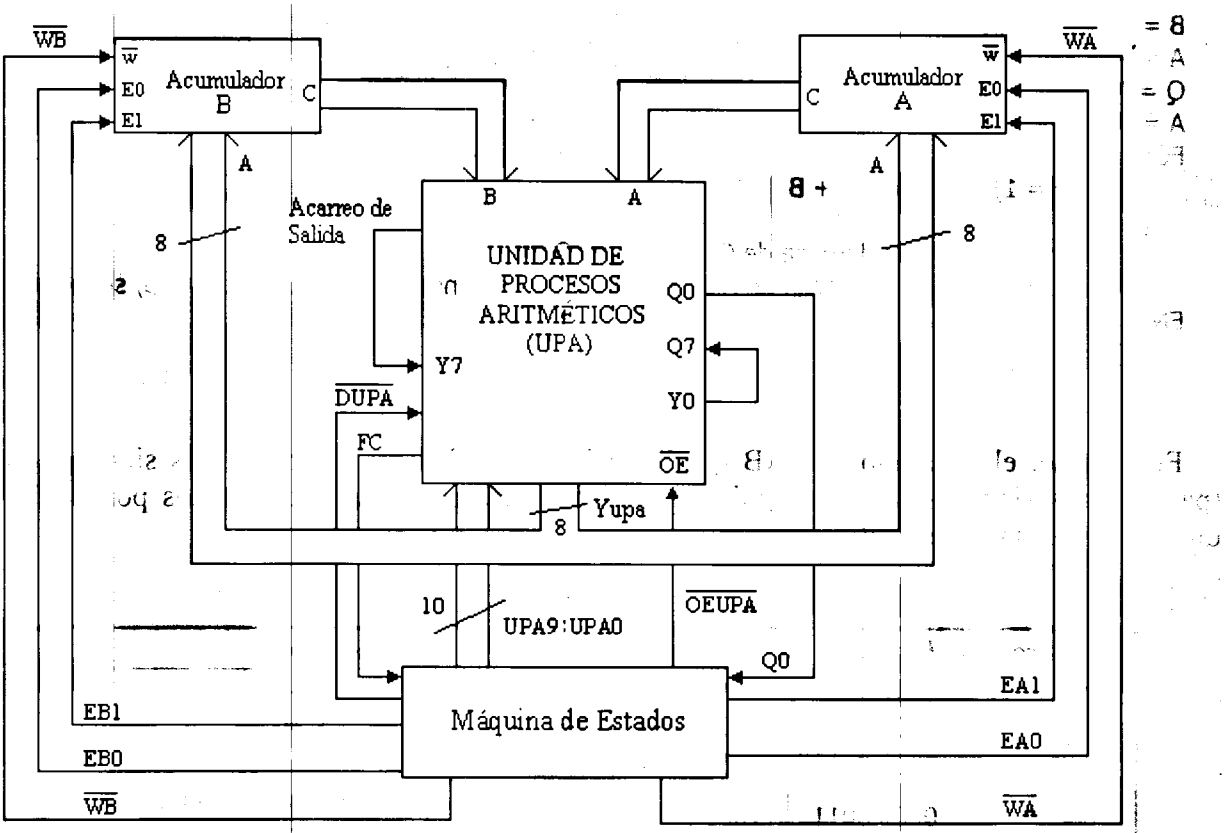


Figura 5.6. Acumuladores A y B conectados a la UPA para efectuar la multiplicación de dos números.

Usando como base este diseño se pueden efectuar multiplicaciones. El siguiente ejemplo ilustra la multiplicación de 2 valores de 8 bits cada uno.

							0	0	0	0	0	1	1	1
						X	0	0	0	0	0	1	0	1
							0	0	0	0	0	1	1	1
						0	0	0	0	0	0	0	0	
					0	0	0	0	0	1	1	1		
				0	0	0	0	0	0	0	0			
			0	0	0	0	0	0	0	0				
		0	0	0	0	0	0	0	0					
	0	0	0	0	0	0	0	0						
0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1

Básicamente, esta multiplicación se puede hacer con sumas y corrimientos. El siguiente algoritmo muestra cómo hacer la multiplicación.

B = Multiplicando
A = Multiplicador
Q = A
A = 0
FOR j = 0:7
IF (Q_0 = 1) THEN A = A + B
Q = Corrimiento a la derecha de Q un bit, con Q_7 = A_0
A = Corrimiento a la derecha de A un bit. Si Q_0 = 1 entonces A_7 = Acarreo de A+B, si no A_7 = 0
END FOR
B = Q

Finalmente, el resultado de AxB queda en los acumuladores A (parte más significativa) y B (parte menos significativa). La siguiente tabla muestra los cálculos efectuados por la UPA para obtener el resultado de AxB.

Registro B	*Registro A*	*Registro Q*	*Acciones*
0000 0111	0000 0101	XXXXXXXX	----------
0000 0111	0000 0000	0000 0101	Q A, A 0
0000 0111	0000 0111	0000 0101	Como Q_0=1 entonces A A+B
0000 0111	0000 0011	1000 0010	Corrimiento de A y Q a la derecha con Q_7=A_0 y A_7=Acarreo de A+B
0000 0111	0000 0011	1000 0010	Como Q_0=0 no hace nada
0000 0111	0000 0001	1100 0001	Corrimiento de A y Q a la derecha con Q_7=A_0 y A_7=0
0000 0111	0000 1000	1100 0001	Como Q_0=1 entonces A B+A
0000 0111	0000 0100	0110 0000	Corrimiento de A y Q a la derecha
0000 0111	0000 0100	0110 0000	Como Q_0=0 no se hace nada
0000 0111	0000 0010	0011 0000	Corrimiento de A y Q a la derecha
0000 0111	0000 0010	0011 0000	Como Q_0=0 no se hace nada
0000 0111	0000 0001	0001 1000	Corrimiento de A y Q a la derecha
0000 0111	0000 0001	0001 1000	Como Q_0=0 no se hace nada
0000 0111	0000 0000	1000 1100	Corrimiento de A y Q a la derecha
0000 0111	0000 0000	1000 1100	Como Q_0=0 no se hace nada
0000 0111	0000 0000	0100 0110	Corrimiento de A y Q a la derecha
0000 0111	0000 0000	0100 0110	Como Q_0=0 no se hace nada
0000 0111	0000 0000	0010 0011	Corrimiento de A y Q a la derecha
0010 0011	0000 0000	0010 0011	B Q

X - Significa "no importa"

Tabla 5.3. Operaciones para efectuar la multiplicación de A por B.

El resultado final queda: AB = 0000 0000 0010 0011.

La figura 5.7 muestra la carta ASM que ejecuta la multiplicación de dos operandos de 8 bits. Observe como las salidas de la carta ASM son las que controlan las funciones de la UPA.

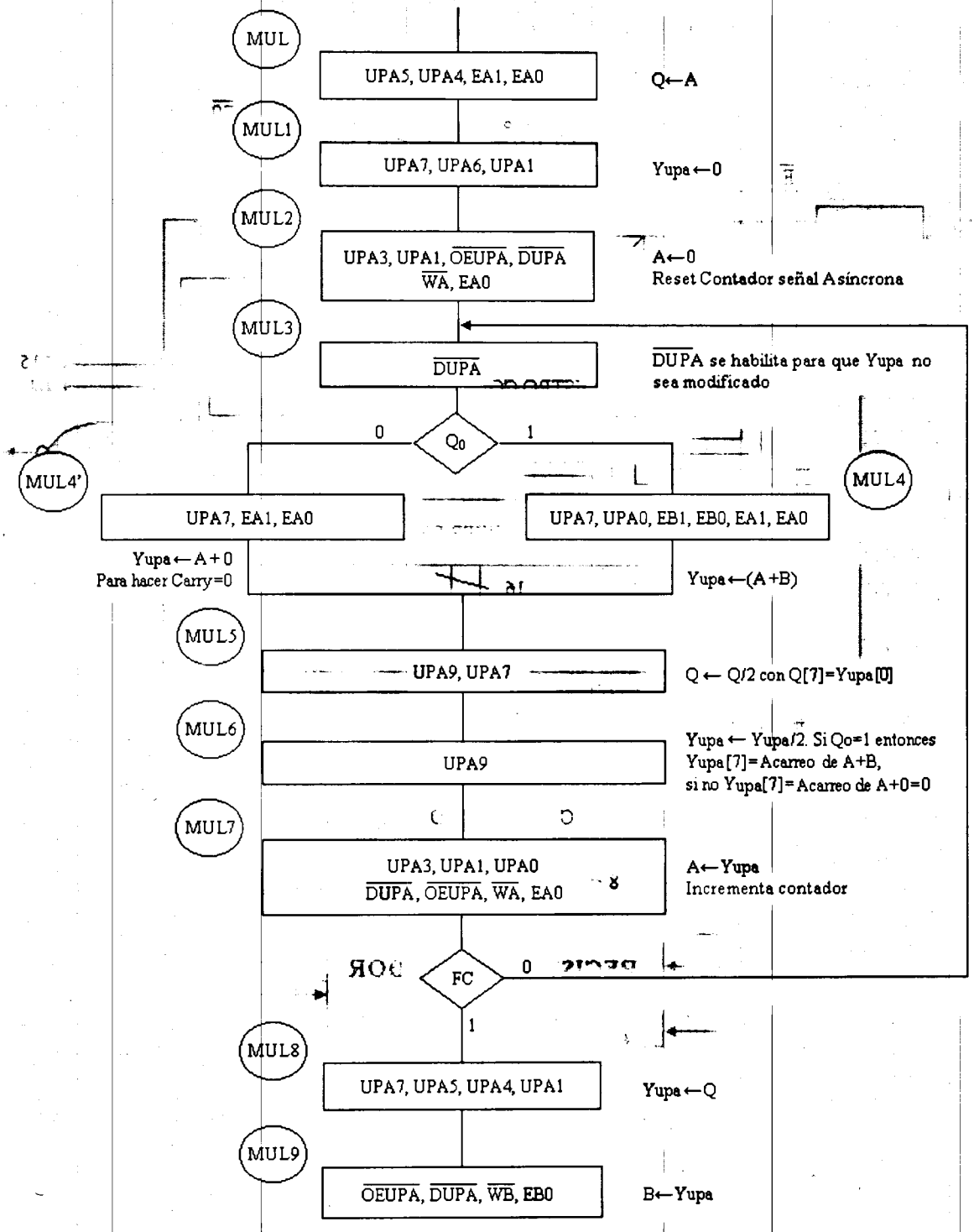


Figura 5.7. Carta ASM para la multiplicación de dos operandos de 8 bits.

5.4.3 REGISTRO CONTADOR DE 16 BITS

La figura 5.8a muestra un registro contador de 16 bits, y tres transeivers que lo aíslan o lo conectan a los buses de la arquitectura.

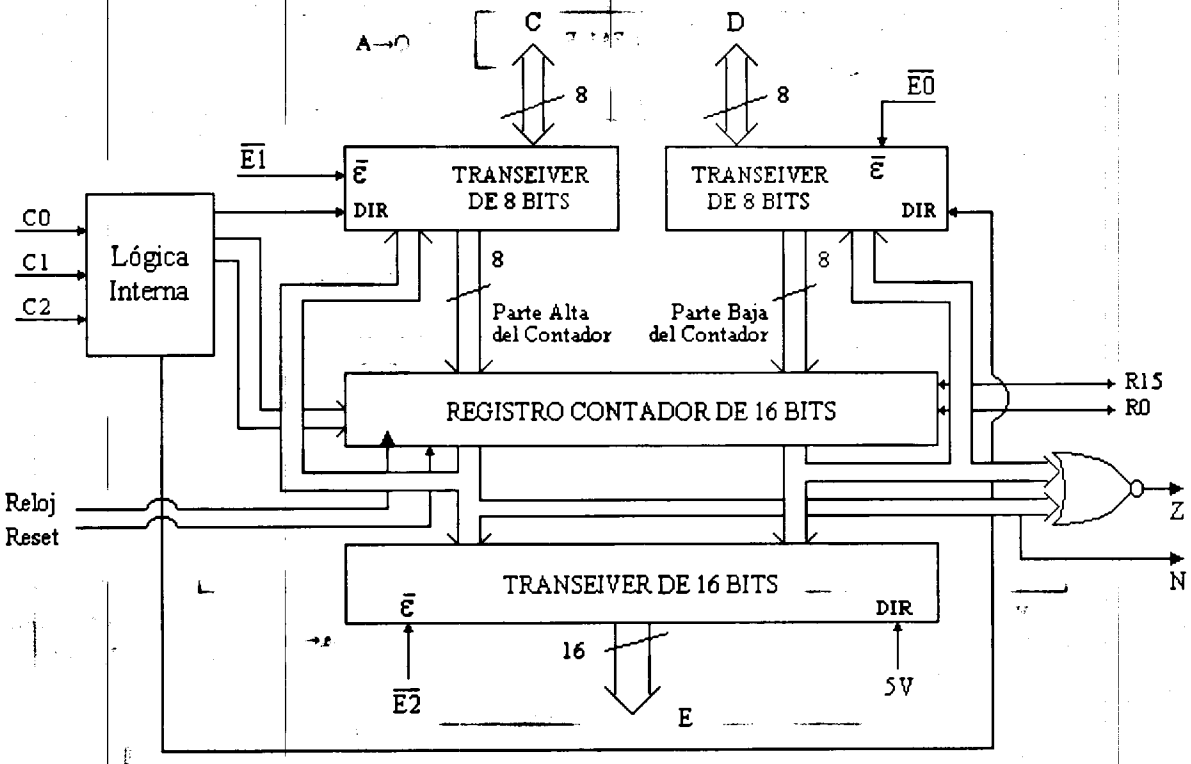


Figura 5.8a. Diagrama de bloques de un registro contador de 16 bits.

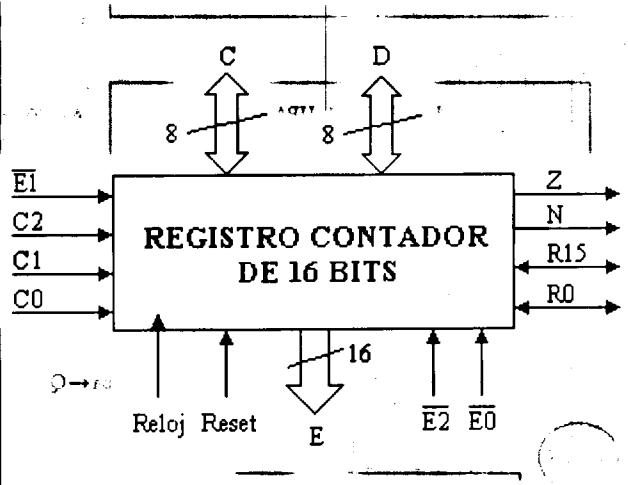


Figura 5.8b. Registro contador de 16 bits.

La tabla 5.4 muestra la relación entre las líneas de control C2, C1 y C0, y las operaciones que realiza el contador.

Líneas de Control			*Función del Contador*
C2	C1	C0	
0	0	0	Mantiene su valor (Lectura)
0	0	1	Incrementa en 1
0	1	0	Decrementa en 1
0	1	1	Carga un valor en la Parte Baja (Escritura)
1	0	0	Carga un valor en la Parte Alta (Escritura)
1	0	1	Carga un valor de 16 bits (Escritura)
1	1	0	Corrimiento a la Izquierda
1	1	1	Corrimiento a la Derecha

Tabla 5.4. Operaciones del registro contador.

Finalmente, las señales $\overline{E2}$, $\overline{E1}$ y $\overline{E0}$ permiten seleccionar los buses de entrada/salida en el registro. Por ejemplo, $\overline{E2}$ selecciona el bus E, $\overline{E1}$ selecciona el bus C y $\overline{E0}$ selecciona el bus D.

5.5 UNIDAD DE CONTROL DE PROGRAMA (UCP)

La UCP se encarga de calcular la dirección de memoria en donde se encuentra el código de la siguiente instrucción a ejecutar. Para esto, cuenta con un registro denominado contador de programa (PC) que contiene la dirección de la siguiente instrucción a ejecutar, y de un registro llamado apuntador de pila (AP) que apunta a una memoria en donde se guardan las direcciones de regreso de las llamadas a subrutinas. La figura 5.9 muestra el diagrama de bloques de la UCP.

Como puede observar, la UCP cuenta con dos registros contadores de 16 bits del mismo tipo que los explicados en el inciso anterior.

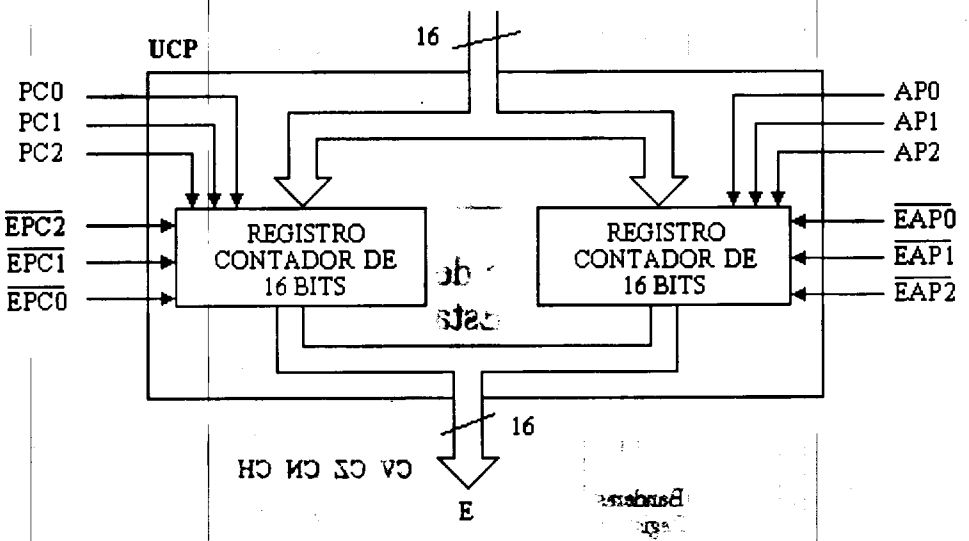


Figura 5.9. Unidad de control de programa (UCP).

5.6 REGISTRO DE ESTADOS O BANDERAS

El registro de banderas contiene los valores de ocho variables que indican el estado de los distintos componentes de la arquitectura. Estos valores pueden venir de alguno de los elementos que integran a la arquitectura, o bien, del bus de datos.

El registro de estados está formado por las siguientes banderas:

- C: Bit de acarreo/borrow.
- V: Bit de sobreflujo.
- Z: Bit de cero. Indica si el resultado de la última operación que se realizó en la UPA, o el valor guardado en alguno de los registros, es igual a cero.
- N: Bit de negativo. Indica el signo del resultado de la UPA, o del valor guardado en alguno de los registros.
- I: Bit de interrupción I. Habilita con un cero, y deshabilita con un uno, las interrupciones conectadas a la línea $\overline{IRQ}$.
- H: Bit de medio acarreo. Acarreo de 4 bits de la UPA. Se utiliza en operaciones donde se usan números con formato BCD.
- X: Bit de interrupción X. Habilita con un cero, y deshabilita con un uno, las interrupciones conectadas a la línea $\overline{XIRQ}$.
- S: Bit de stop. Pone al microprocesador en bajo consumo de energía.

Cada vez que se ejecuta una instrucción en ensamblador, el registro de estados es actualizado con nuevos valores de banderas. Estos valores, dependiendo de la instrucción que se ejecutó, pueden provenir de los registros acumuladores, de los registros de 16 bits, de la UPA, del bus de datos, o de ningún sitio, es decir, no son modificados.

El siguiente diagrama muestra la estructura externa de este registro, también denominado **CCR** (Condition Code Register).

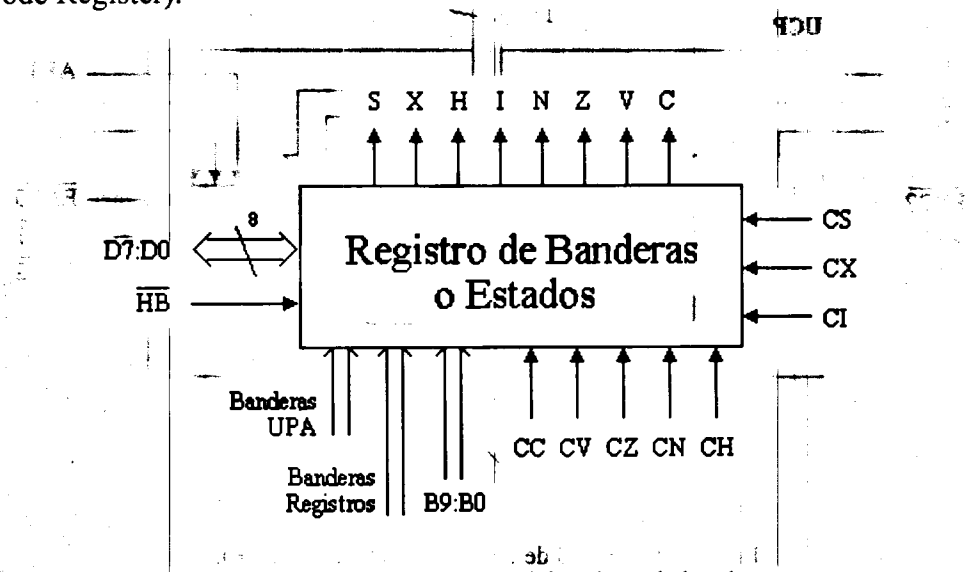


Figura 5.10. Estructura externa del registro de banderas.

La figura 5.11 muestra la estructura interna del registro de banderas.

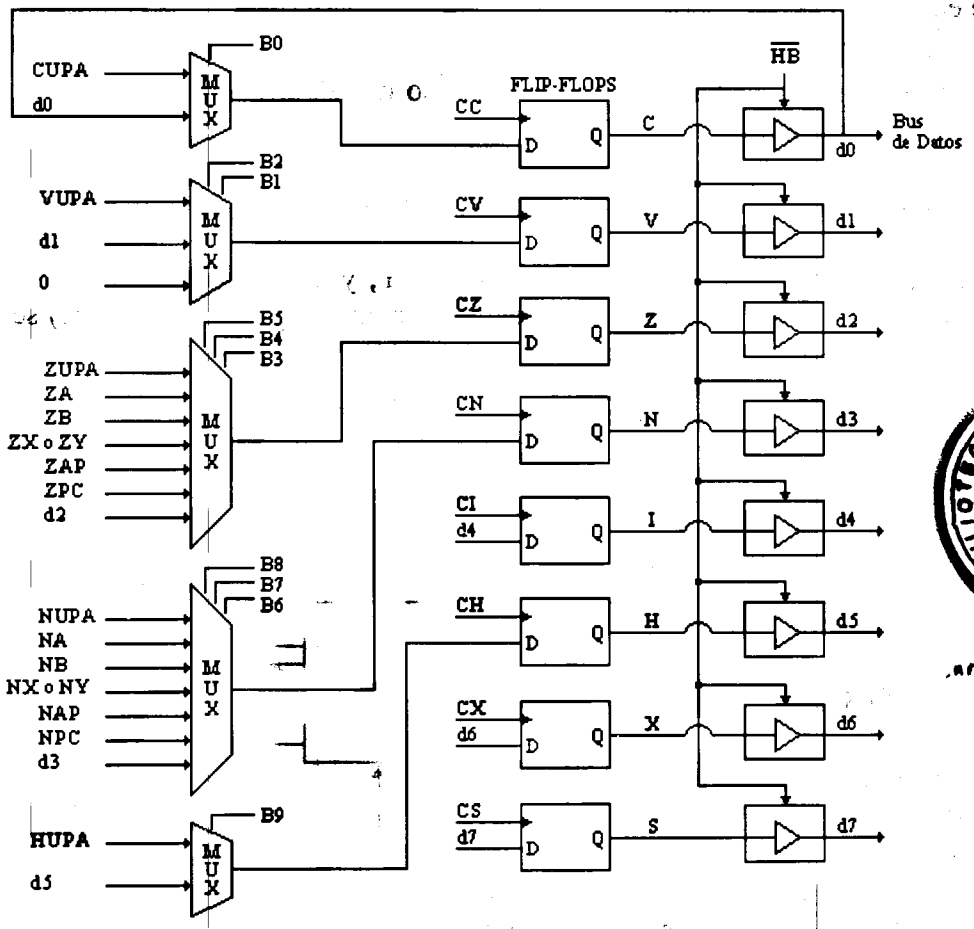


Figura 5.11. Estructura interna del registro de banderas.

Las líneas d7 a d0 conectan al registro de banderas con el bus de datos. Los circuitos tres estados, en conjunto con la señal $\overline{HB}$, aíslan o conectan el registro de banderas al bus de datos interno.

Las líneas CC, CV, CZ, CN, CI, CH, CX y CS controlan los relojes de los flip-flops asociados a las banderas. Las líneas B9 a B0 controlan la selección de los multiplexores, por ejemplo para la bandera de Z, si B5B4B3=000 se selecciona la bandera de Z de la UPA, si B5B4B3=001 la del acumulador A, si B5B4B3=010 la del acumulador B, si B5B4B3=011 la del registro índice X ó Y, si B5B4B3=100 la del apuntador de pila, si B5B4B3=101 la del contador de programa y si B5B4B3=110 la del bus de datos. Para seleccionar el resto de las banderas se procede de manera similar.

5.7 UNIDAD DE CONTROL DE INTERRUPCIONES (UCI)

La UCI se encarga de recibir peticiones de interrupciones externas. Tales peticiones provienen de alguno de los dispositivos conectados a las líneas $\overline{IRQ}$ y $\overline{XIRQ}$. Como respuesta, la UCI envía una dirección de salto al secuenciador indicándole el inicio del algoritmo de máquina de estados que atiende la interrupción pedida.

Antes de atender la interrupción, el algoritmo de la máquina de estados guarda en la pila la dirección de la siguiente instrucción a ser ejecutada, dirección que está contenida en el registro PC. Los valores de los acumuladores, de los registros X e Y, y del registro de estados, también son guardados en la pila. Una vez guardados estos datos, el contador de programa (PC) se carga con la dirección de inicio de la rutina de atención a la interrupción. En el capítulo siguiente se muestran las cartas ASM que atienden las interrupciones.

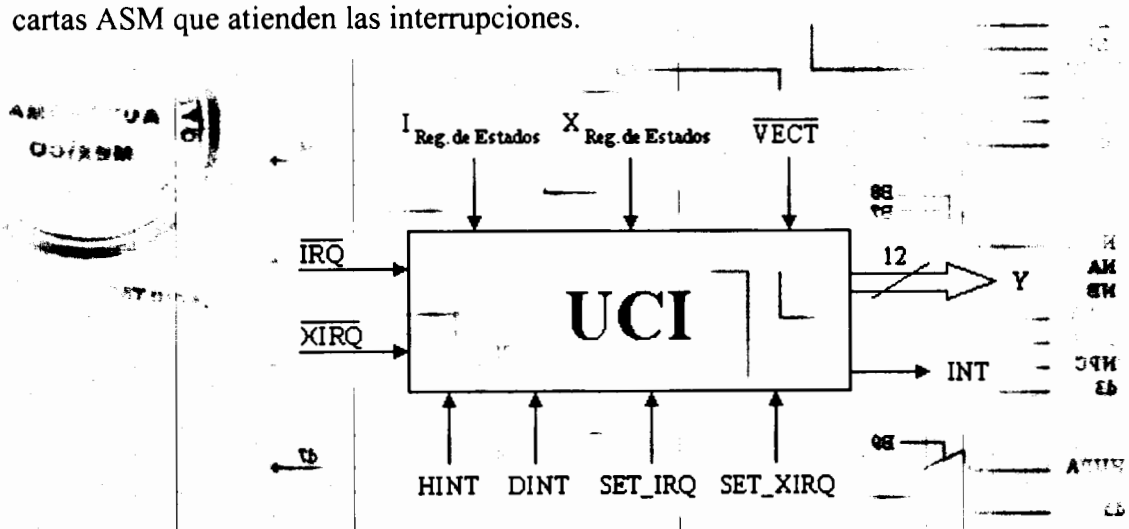


Figura 5.12. Diagramas de bloques externo de la UCI.

En el registro de banderas del 68HC11 se tienen dos bits que activan o desactivan las interrupciones que provienen de los dispositivos conectados a las líneas $\overline{IRQ}$ y $\overline{XIRQ}$. Las desactivaciones se pueden hacer por medio de instrucciones de software, de manera que el usuario tiene el control directo sobre estas líneas. Por otra parte, la interrupción $\overline{XIRQ}$ tiene prioridad sobre la interrupción IRQ, así que si ocurrieran al mismo tiempo, la primera que se atendería sería $\overline{XIRQ}$ y después IRQ.

En la figura 5.13 se muestra el diagrama de bloques de la unidad de control de interrupciones. Las líneas X Reg. de Estados e I Reg. de Estados están conectadas directamente a los bits que habilitan las interrupciones, con un cero se habilitan y con un uno se deshabilitan. Las líneas SET_XIRQ y SET_IRQ deshabilitan las interrupciones cuando hay un reset. Las líneas HINT y DINT sirven para deshabilitar y habilitar la generación de la línea que indica la presencia de una interrupción (INT). Y los registros de direcciones I y X contienen las direcciones a donde tiene que saltar el secuenciador en caso de que ocurriera una interrupción.

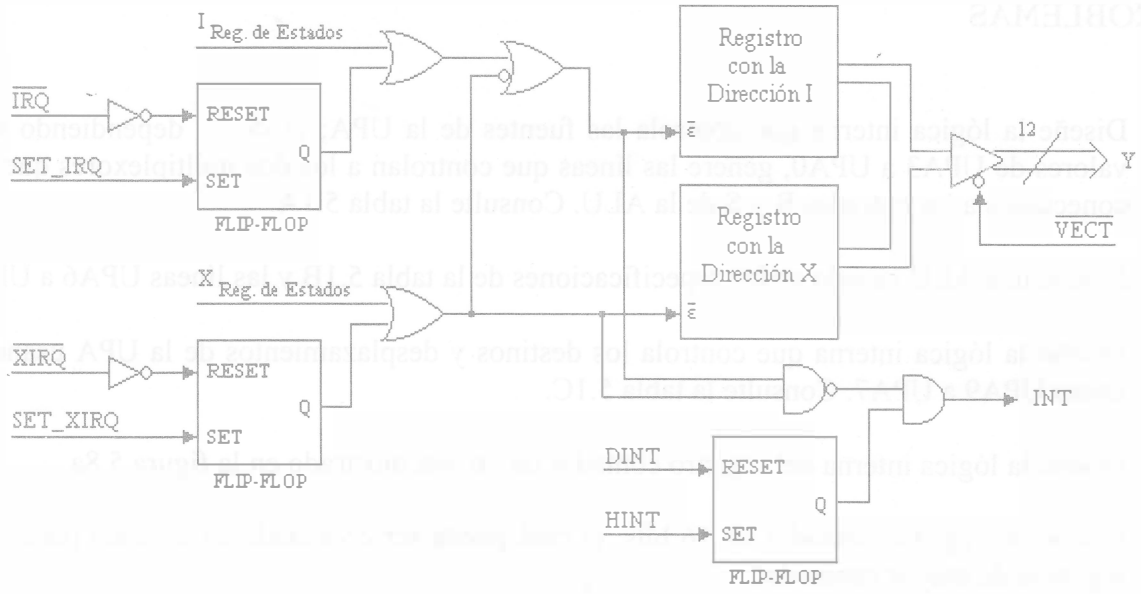


Figura 5.13. Diagrama de bloques de la unidad de control de interrupciones (UCI).

PROBLEMAS

1. Diseñe la lógica interna que controla las fuentes de la UPA; es decir, dependiendo de los valores de UPA3 a UPA0, genere las líneas que controlan a los dos multiplexores que están conectados a las entradas R y S de la ALU. Consulte la tabla 5.1A.

2. Diseñe una ALU basado en las especificaciones de la tabla 5.1B y las líneas UPA6 a UPA4.

3. Diseñe la lógica interna que controla los destinos y desplazamientos de la UPA usando las líneas UPA9 a UPA7. Consulte la tabla 5.1C.

4. Diseñe la lógica interna del registro contador de 16 bits mostrado en la figura 5.8a.

5. Diseñe un registro contador de 16 bits, el cual pueda ser conectado en cascada para formar registros de mayor capacidad.

   Especificaciones del registro contador de 16 bits:

   - Realiza cargas de 8 bits a través de dos buses: el bus A que carga un dato en la parte alta del registro y el bus B que carga un dato en la parte baja.
   - Realiza lecturas de 8 y de 16 bits a través de los buses A y B (de 8 bits cada uno) y del bus C (de 16 bits).
   - Los buses A, B y C utilizan tecnología tres estados.
   - Además, el contador puede realizar incrementos y decrementos.

   a) Dibuje el diagrama de bloques para este registro. Si utiliza bloques de lógica internos, explique cada uno de esos bloques utilizando una tabla de verdad.
   b) Conecte dos registros en cascada y verifique que los incrementos y decrementos estén efectuándose correctamente, para ello, anexe un diagrama de tiempos.

6. Investigue un algoritmo para efectuar la división de dos números enteros positivos, y con base en ese algoritmo responda los siguientes incisos.

   a) Modifique la arquitectura de la figura 5.6, si así lo requiriera, de manera que pueda ser ejecutado su algoritmo de la división.
   b) Diseñe el algoritmo de la máquina de estados, carta ASM, que controle las señales de la UPA y de los registros acumuladores Ay B para efectuar la división.
   c) Pruebe el funcionamiento de su algoritmo ejecutando la siguiente división: 39 entre 8.

7. Construya una unidad de interrupciones que permita ocho niveles de interrupción. Con tres líneas de entrada se indica el número del periférico que está interrumpiendo y con otra línea se indica que hay una nueva interrupción. En el registro de estados se indica a partir de que nivel se aceptarán las interrupciones; por ejemplo, si se escribe un cuatro en el registro de estados, solamente las interrupciones de nivel cuatro y superiores serán atendidas.

CAPÍTULO VI
DISEÑO DE UN MICROPROCESADOR DE 8 BITS

6.1 ARQUITECTURA DEL MICROPROCESADOR 68HC11

En el capítulo anterior fueron diseñados los componentes básicos de una computadora. En este capítulo se muestra cómo hacer la interconexión de esos elementos, y la manera de controlarlos utilizando máquinas de estados.

Si se desea que el microprocesador ejecute un conjunto de instrucciones en lenguaje ensamblador, será necesario codificar cada instrucción en varias operaciones, de manera que sean totalmente entendibles para el microprocesador. La metodología a seguir son las máquinas de estados. Por lo tanto, para cada instrucción en ensamblador existirá un algoritmo de máquina de estados, que activará o desactivará secuencialmente, las líneas de control de la arquitectura.

La figura 6.1 presenta el diagrama general de interconexión de la computadora. Usando como referencia esta figura, los pasos para ejecutar una instrucción en lenguaje ensamblador, residente en memoria externa, son los siguientes.

1) La UCP carga la dirección de la siguiente instrucción en el registro de direcciones, y se habilita la memoria para lectura. El contenido de la dirección seleccionada, con el código de la instrucción, es colocado en el bus de datos externo.
2) El código de la instrucción entra por el buffer de datos y se carga en el registro de instrucción.
3) La UCC decodifica la instrucción, es decir, salta a la dirección de microprograma dada por el código de la instrucción, en donde comienzan las micro-operaciones que serán ejecutadas.
4) Trae los operandos si así lo requiere la instrucción en ensamblador.
5) Si se trata de una operación lógico aritmética, se le indica a la UPA la operación a ejecutar.
6) Guarda el resultado en el lugar indicado por la instrucción en ensamblador y se actualizan las banderas o estados.
7) La UCP prepara la dirección de la siguiente instrucción a ejecutar, pero antes, la UCC revisa si hay interrupciones y efectúa el procedimiento de atención a interrupciones si es necesario.

La tarea de control será ejecutada por la UCC, quien activará las líneas de control de los distintos componentes de la arquitectura, de acuerdo a los algoritmos de máquinas de estados implantados. Recuerde que la activación de las líneas de control de la arquitectura se representan como salidas en un estado de una carta ASM.

A continuación se muestra la arquitectura del 68HC11 con los componentes desarrollados en el capítulo anterior. También se describe la función de las líneas de salida de la memoria de microprograma, líneas que controlan el funcionamiento de la arquitectura.

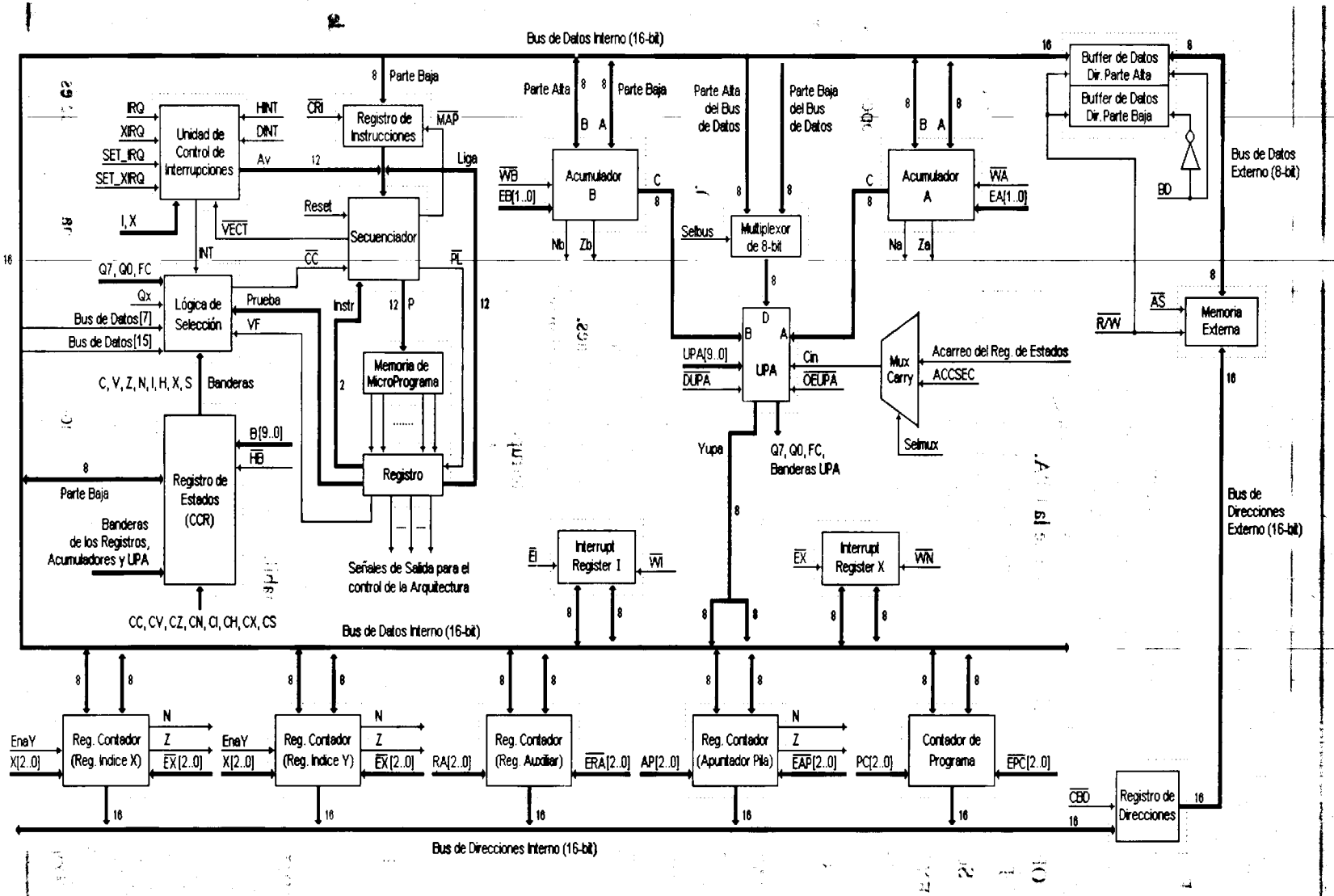


Figura 6.1. Arquitectura del microprocesador 68HC11.

Señales de Control	*Descripción*
$\overline{CRI}$	Carga un dato en el registro de instrucciones.
EB1:EB0	Controlan las operaciones del acumulador B.
$\overline{WB}$	Línea de escritura del acumulador B.
EA1:EA0	Controlan las operaciones del acumulador A.
$\overline{WA}$	Línea de escritura del acumulador A.
Selbus	Selecciona la fuente de datos para la entrada D de la UPA. Si Selbus=0, el dato se toma de la parte baja del bus de datos interno, si no, de la parte alta.
UPA9:UPA0	Líneas de control de la UPA.
$\overline{OEUPA}$	Habilita la salida de la UPA.
$\overline{DUPA}$	Deshabilita el reloj interno de la UPA.
Selmux	Selecciona el carry del registro de estados o del secuenciador. El dato elegido representa el carry de entrada a la UPA.
$\overline{EX2}$: $\overline{EX0}$	Seleccionan los buses del registro índice X ó del registro índice Y.
X2:X0	Controlan las operaciones del registro X ó del registro índice Y.
EnaY	Habilita las operaciones en los registros X e Y. Si EnaY=0, las operaciones en el registro X estarán habilitadas y en el registro Y no lo estarán. Si EnaY=1, las operaciones en el registro Y estarán habilitadas y en el registro X deshabilitadas.
$\overline{ERA2}$: $\overline{ERA0}$	Seleccionan los buses del registro auxiliar RA.
RA2:RA0	Controlan las operaciones del registro auxiliar.
$\overline{EAP2}$: $\overline{EAP0}$	Seleccionan los buses del registro apuntador de pila (AP).
AP2:AP0	Controlan las operaciones del registro AP.
$\overline{EPC2}$: $\overline{EPC0}$	Seleccionan los buses del registro contador de programa (PC).
PC2:PC0	Controlan las operaciones del registro PC.
$\overline{CBD}$	Carga un dato en el registro de direcciones.
$\overline{WX}$	Carga un dato en el registro de interrupciones X.
$\overline{EX}$	Habilita el registro de interrupciones X.
$\overline{WI}$	Carga un dato en el registro de interrupciones I.
$\overline{EI}$	Habilita el registro de interrupciones I.
$\overline{AS}$	Habilita la memoria externa.
$\overline{R/W}$	Señal de lectura/escritura de la memoria externa.
BD	Selecciona el buffer de datos que conecta al bus de datos externo con los buses de datos internos. Con un cero se selecciona la parte baja del buffer de datos, y con un uno la parte alta.
DINT, HINT, SET_IRQ, SET_XIRQ	Líneas que habilitan o deshabilitan la Unidad de Control de interrupciones.
B9:B0, CC, CN, CV, CZ, CI, CH, CX, CS	Líneas que controlan las operaciones en el registro de estados.
HB	Habilita el bus que conecta al registro de estados con el bus de datos interno.
I1:I0	Le indican al secuenciador qué tipo de instrucción ejecutar.
Prueba4:Prueba0	Seleccionan una variable de entrada en la lógica de selección.
VF	Línea de verdadero-falso.
ACCSEC	Acarreo proveniente de la memoria de microprograma. El valor de este acarreo puede ser modificado según nuestras necesidades.

Tabla 6.1. Descripción de las líneas de control para la arquitectura del 68HC11.

6.2 TIPOS DE INSTRUCCIONES DE ENSAMBLADOR DEL 68HC11

Las instrucciones en lenguaje ensamblador que puede ejecutar el microprocesador 68HC11, dependen de la forma en la que se acceden los datos. A continuación se explica brevemente, los seis tipos de acceso que existen.

6.2.1 ACCESO INMEDIATO

Las instrucciones que utilizan el acceso inmediato tienen el siguiente formato: el primer byte de la instrucción corresponde a su código de operación, y el segundo byte al valor de un dato de 8 bits. Un ejemplo es la instrucción ADDA #Dato. Esta instrucción suma al acumulador A el contenido de la localidad de memoria siguiente al código de la instrucción. Note que el dato está precedido por el símbolo #, que se emplea para diferenciar este tipo de acceso de los demás.

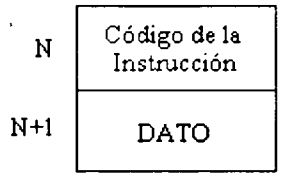


Figura 6.2. Acceso inmediato.

6.2.2 ACCESO EXTENDIDO

Las instrucciones que utilizan el acceso extendido tienen el siguiente formato: el primer byte corresponde al código de operación de la instrucción, y los dos bytes siguientes a una dirección de 16 bits que contiene el valor del operando. Un ejemplo es la instrucción ADDA Dirección_16_Bits. Esta instrucción suma al acumulador A el dato contenido en la localidad de memoria dada por *Dirección_16_Bits*.

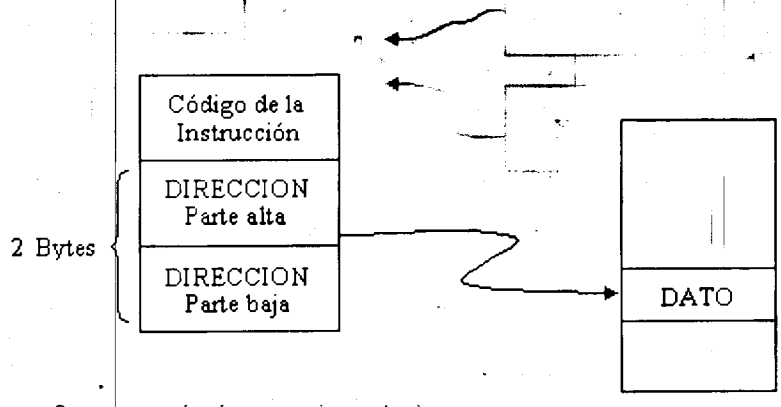


Figura 6.3. Acceso extendido.

6.2.3 ACCESO DIRECTO

Las instrucciones que utilizan el acceso directo tienen el siguiente formato: el primer byte corresponde al código de operación de la instrucción, y el segundo byte a una dirección de 8 bits que contiene el valor del operando. Un ejemplo es la instrucción ADDA Dirección_8_Bits. Esta instrucción suma al acumulador A el dato contenido en la localidad de memoria dada por *Dirección_8_Bits*.

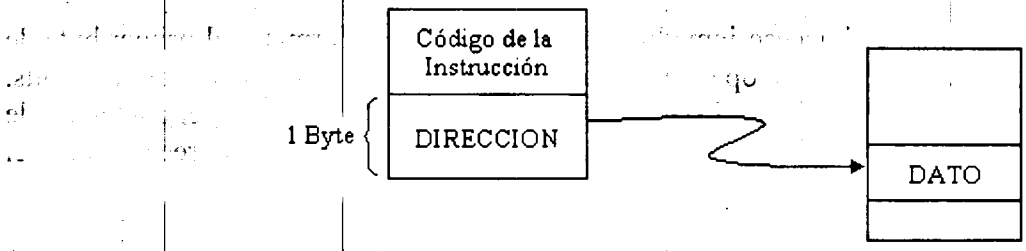


Figura 6.4. Acceso directo.

6.2.4 ACCESO INDEXADO

Las instrucciones que utilizan el acceso indexado tienen el siguiente formato: el primer byte corresponde al código de operación de la instrucción, y el segundo byte a un desplazamiento de 8 bits sin signo, que se emplea para calcular la dirección del operando. La dirección del operando se calcula sumando el valor del desplazamiento más el contenido del registro X, ó el contenido del registro Y. Un ejemplo es la instrucción ADDA Desplazamiento, X. Esta instrucción suma al acumulador A el dato contenido en la dirección (Registro X) + Desplazamiento.

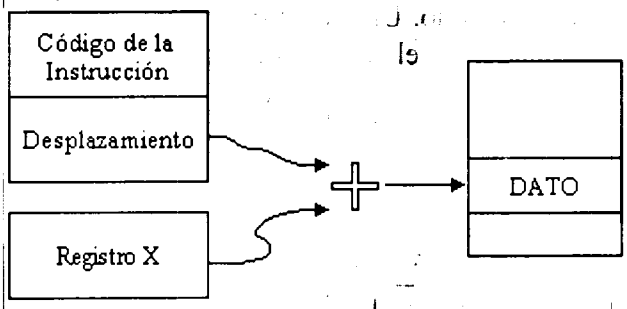


Figura 6.5. Acceso indexado.

6.2.5 ACCESO RELATIVO

Las instrucciones que utilizan el acceso relativo tienen el siguiente formato: el primer byte corresponde al código de operación de la instrucción, y el segundo byte a un desplazamiento de 8 bits con signo, que se emplea para calcular la dirección de la siguiente instrucción a ejecutar. Este

tipo de acceso solo se utiliza en las instrucciones de salto. La dirección de salto se obtiene sumando el contenido del registro PC más el desplazamiento.

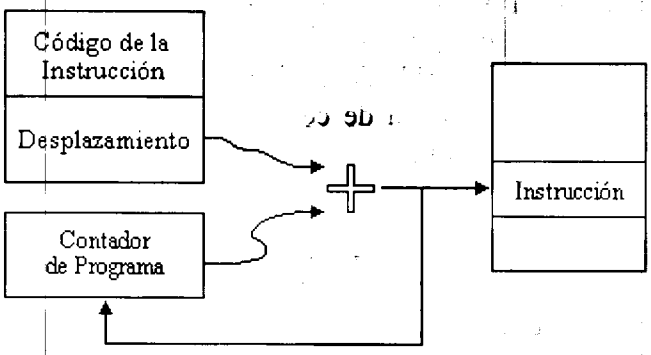


Figura 6.6. Acceso relativo.

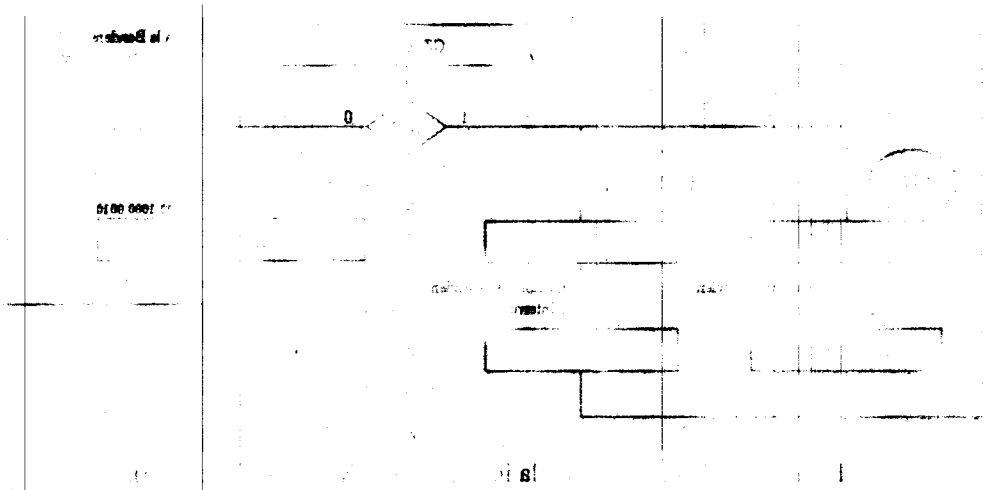
6.2.6 ACCESO INHERENTE

Este acceso no necesita operandos. El código de la instrucción es suficiente para saber el tipo de instrucción y la tarea que debe ejecutar.

Ejemplo: INX. Incrementa el contenido del registro X en una unidad.

6.3 MICROPROGRAMACIÓN

A continuación se muestran algunos ejemplos de algoritmos de máquinas de estados en donde se utilizan los modos de acceso anteriores. Los algoritmos que se presentan hacen uso del modelo de arquitectura presentado en la figura 6.1, de manera que pueden ser ejecutados sobre ella.



6.3.1 INSTRUCCIÓN INX (Acceso Inherente)

Instrucción: INX
Operación: IX ⇐ (IX) + 1
Código[2]: 08
Descripción: Incrementa el contenido del registro X en una unidad.
Banderas: El valor de la bandera de cero (Z) es actualizado tras la ejecución de esta instrucción. Z valdrá uno si el resultado en el registro índice X es cero, y valdrá cero en caso contrario. El estado de las demás banderas no se modifica.

S	X	H	I	N	Z	V	C
—	—	—	—	—	↕	—	—

La siguiente carta ASM representa el algoritmo que ejecuta la instrucción INX.

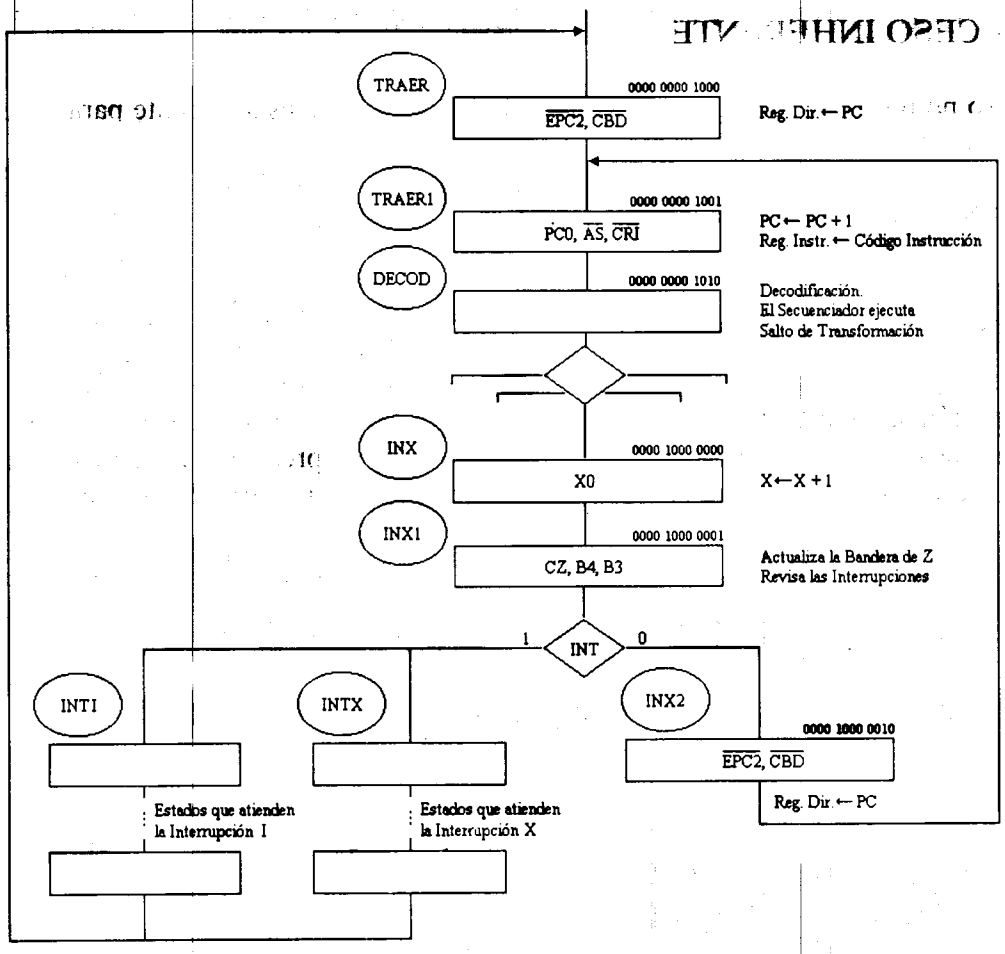


Figura 6.7. Carta ASM para la instrucción INX (acceso inherente).

[2] El código de operación en el 68HC11 es una palabra única de 8 bits para cada instrucción en ensamblador, la cual la identifica del resto de las instrucciones. Estos códigos de operación se presentan en formato hexadecimal.

A continuación se explica la carta ASM de la figura 6.7.

En los primeros estados se ejecuta el ciclo "FETCH", es decir, se trae el código de operación de la siguiente instrucción a ejecutar. El contador de programa, PC, contiene la dirección en memoria en donde se localiza la siguiente instrucción a ejecutar, por lo tanto, se coloca el contenido de PC en el registro de direcciones; éste último está conectado al bus de direcciones de la memoria externa.

En el estado "TRAER" de la carta ASM se efectúan las actividades anteriormente descritas. Primero, se lee el contenido del registro PC, es por ello que las líneas PC2, PC1 y PC0 son puestas a ceros, es decir, se habilita al registro PC para sólo lectura. A continuación se habilita la salida del registro PC para que el dato contenido en él pase hacia el bus de direcciones interno, esto se realiza colocando la línea EPC2 [3] a cero. Finalmente, para cargar la dirección de PC en el registro de direcciones se coloca la señal CBD a cero[4]. La figura 6.8 muestra de manera gráfica lo mencionado con anterioridad.

En el estado "TRAER1" se lee de la memoria externa, el código de operación de la instrucción. La línea R/W puesta a uno habilita la lectura de datos en la memoria externa. En este caso la línea R/W no se escribe dentro del estado, se asume que vale uno puesto que está negada. También se activa la señal AS para habilitar a la memoria antes de efectuar la operación de lectura. El código de la instrucción, leído de memoria, viaja a través del bus de datos externo y continúa su camino hacia la parte baja del bus de datos interno. La señal BD puesta a cero y la señal R/W puesta a uno son suficientes para que el buffer de datos decida entre pasar un dato hacia la parte alta o hacia la parte baja del bus de datos interno. Una vez que el código de la instrucción está presente en el bus de datos, el registro de instrucción podrá guardarlo, para ello se coloca a cero a la señal CRI . De esta manera, en el registro de instrucción queda guardada una copia del código de la instrucción a ejecutar. Por último, la señal PC0=1 incrementa en una unidad el contenido del registro PC para así obtener la dirección de la siguiente palabra en memoria. Todas estas acciones son mostradas en la figura 6.9.

En el estado "DECOD" se decodifica la instrucción en ensamblador, es decir, se le indica al secuenciador a qué dirección saltar para iniciar las microinstrucciones que ejecutan dicha instrucción. En este estado, el secuenciador efectúa un salto de transformación activando la línea de MAP , la cual selecciona el contenido del registro de instrucción como el dato de entrada hacia la entrada D del secuenciador. La figura 6.10 muestra de manera gráfica lo mencionado con anterioridad.

En el registro de instrucción se tiene el código de la instrucción INX, es decir, 08 en formato hexadecimal. Dado que la entrada D del secuenciador es de 12 bits, los 8 bits del registro de instrucción deben extenderse a 12 bits, de manera que se colocan cuatro ceros en la parte menos significativa del contenido del registro de instrucción y así se forma la dirección de inicio de la instrucción INX: 0000 1000 0000.

[3] Véase la tabla de operación del Registro de 16 bits presentada en el Capítulo 5.

[4] Recuerde que en notación de cartas ASM sólo se colocan las salidas activas. Por ejemplo, si la señal de salida está negada, colocar el nombre de la salida en un estado significará que la señal toma el valor de cero, en caso contrario, se asume que la señal vale uno. Por otra parte, si la señal de salida no está negada, colocar el nombre de la salida en un estado significará que la señal toma el valor de uno, si no, se asume que la señal vale cero.

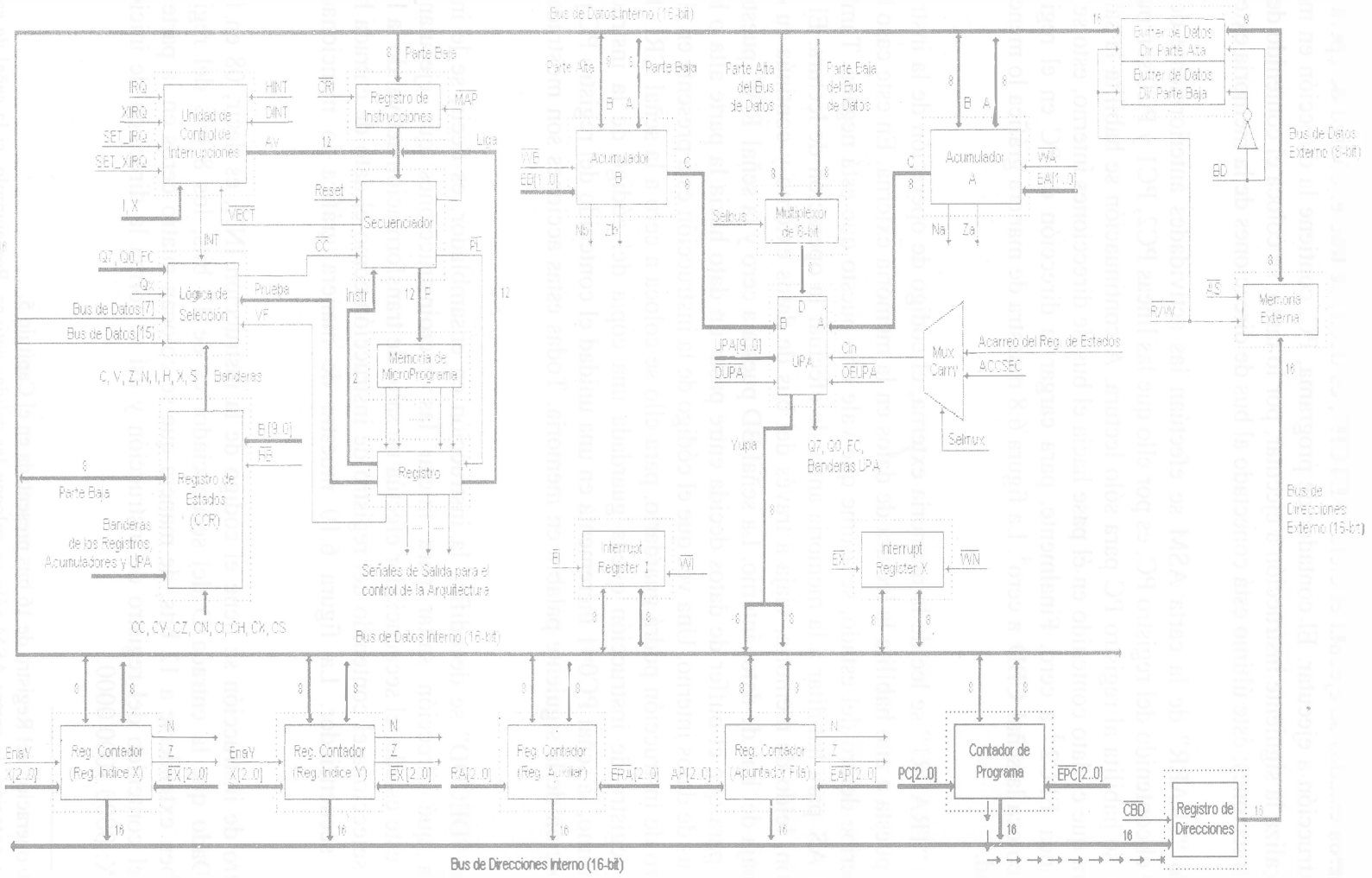


Figura 6.8. Primera fase de ejecución de toda instrucción. Estado TRAER.

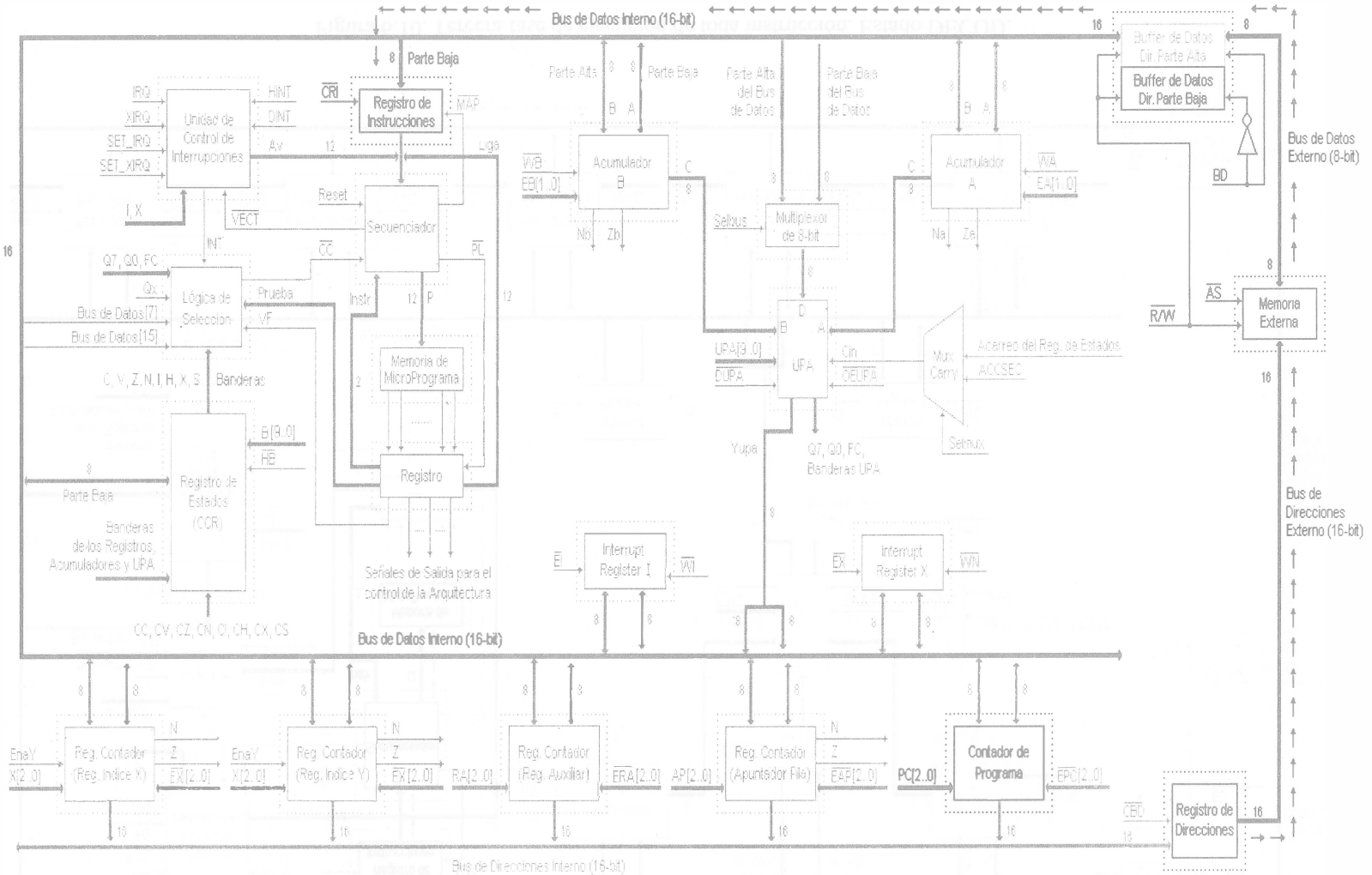


Figura 6.9. Segunda fase de ejecución de toda instrucción. Estado TRAER1.

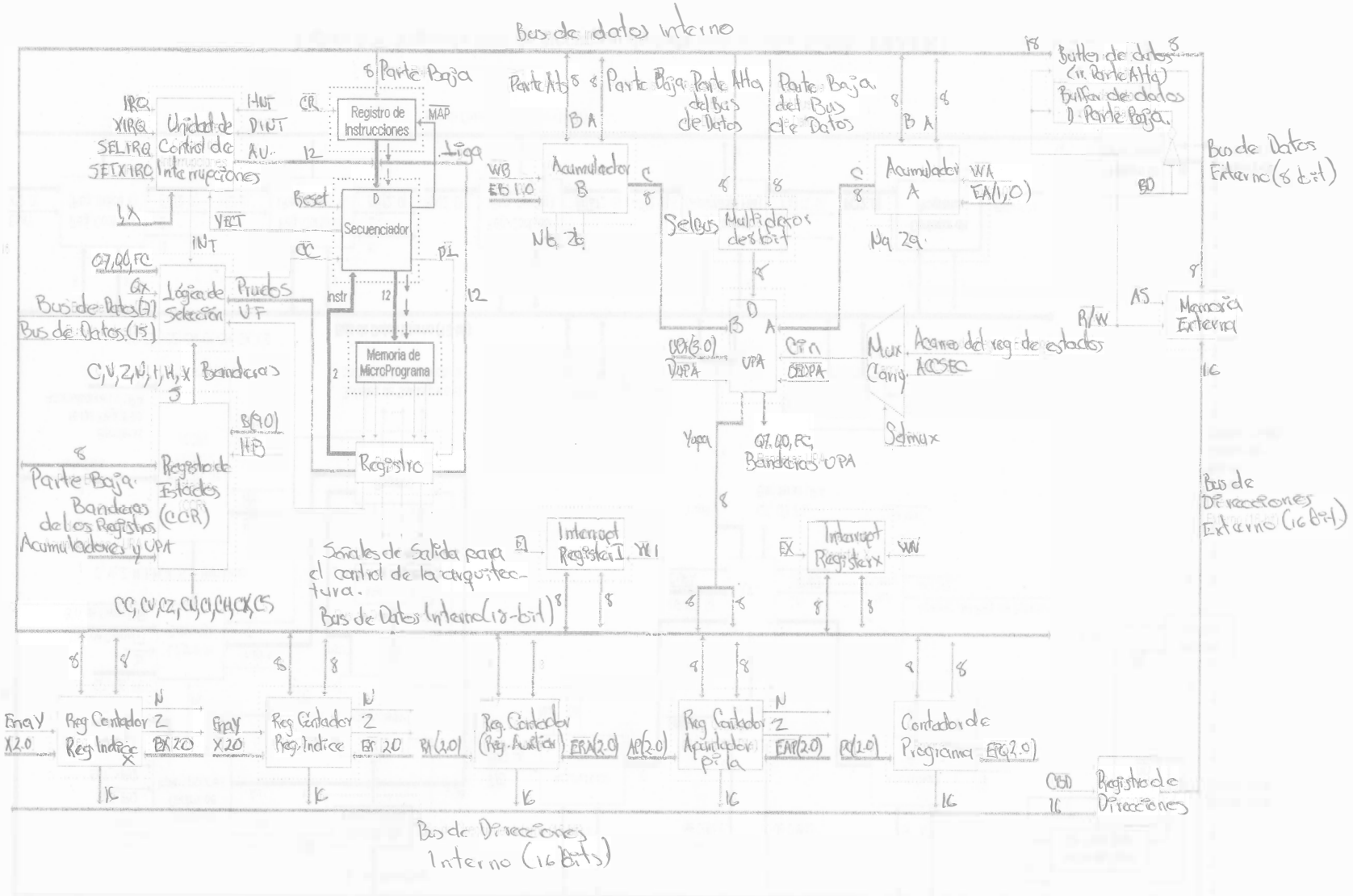


Figura 6.10. Tercera fase de ejecución de toda instrucción. Estado DECOD.

En el estado "INX" comienzan las microinstrucciones necesarias para ejecutar dicha instrucción. En este estado se incrementa el contenido del registro índice X con la activación de la señal X0, es decir, X2X1X0=001. Además, el valor de la señal EnaY debe permanecer en cero, ya que las operaciones que se realizarán serán sobre el registro X. Recuerde que las instrucciones que hacen uso de los registros índices X e Y son idénticas, por ello, ambos registros comparten las mismas líneas de operación y de habilitación. Es por medio de la señal EnaY que se selecciona el registro sobre el que se operará. Más adelante veremos la instrucción INY para ejemplificar el uso de la señal EnaY. El estado "INX" está representado gráficamente en la figura 6.11.

En el estado "INX1" se actualizan los valores de las banderas. Las banderas afectadas se establecen en la especificación de la instrucción. En el caso de la instrucción INX, la bandera de cero (Z) es la única bandera a modificar.

Para actualizar el valor de la bandera de cero en el registro de estados (CCR) debemos habilitar el reloj del flip-flop asociado a esta bandera, es decir, colocar la señal CZ a uno. La habilitación de CZ permitirá cargar un nuevo valor en el registro flip-flop de la bandera Z. La procedencia del nuevo valor de Z se selecciona utilizando las líneas B5, B4 y B3; estas líneas eligen qué bandera de Z cargar en el registro de estados. Por ejemplo, si B5B4B3=011, el nuevo valor de la bandera Z provendrá del registro índice X. La figura 6.12 muestra las actividades realizadas en el estado "INX1".

En el mismo estado "INX1" se revisa si existen interrupciones. Si ocurrió una interrupción, la señal INT tendrá asignado el valor de uno, y se deberá realizar un salto hacia alguno de los estados de atención a la interrupción. El estado INTI será seleccionado si la línea IRQ fue quien generó la señal de interrupción INT, por el contrario, si la línea XIRQ fue quien generó la interrupción, el estado seleccionado será INTX.

La señal INT y la dirección de salto hacia la rutina de atención a la interrupción son generadas por la unidad de control de interrupciones. La señal INT es utilizada por la lógica de selección para generar la señal CC, y la dirección de salto es utilizada por el secuenciador para saber la dirección de inicio de la microrutina de atención a la interrupción. La instrucción que ejecuta el secuenciador es un salto condicional utilizando la dirección de las interrupciones; recuerde que esta instrucción activa la señal VECT para seleccionar la dirección de salto proveniente de la unidad de control de interrupciones.

Si no existe alguna interrupción entonces se debe saltar al estado "TRAER" para comenzar el ciclo "FETCH" de la siguiente instrucción. Debido a las características del secuenciador, la instrucción de salto condicional con interrupciones no permite ejecutar dos saltos, esto es, si no se realiza el salto hacia la dirección de interrupción entonces el estado siguiente está dado por el estado presente más uno. Por lo tanto, es necesario colocar el estado "INX2" y desde ahí realizar el salto hacia el estado "TRAER". Con la finalidad de no desperdiciar el estado "INX2" podemos ejecutar en ese mismo estado las microinstrucciones del estado "TRAER" y realizar un salto condicional al estado "TRAER1". En la figura 6.13 se presentan las actividades ejecutadas en el estado "INX2".

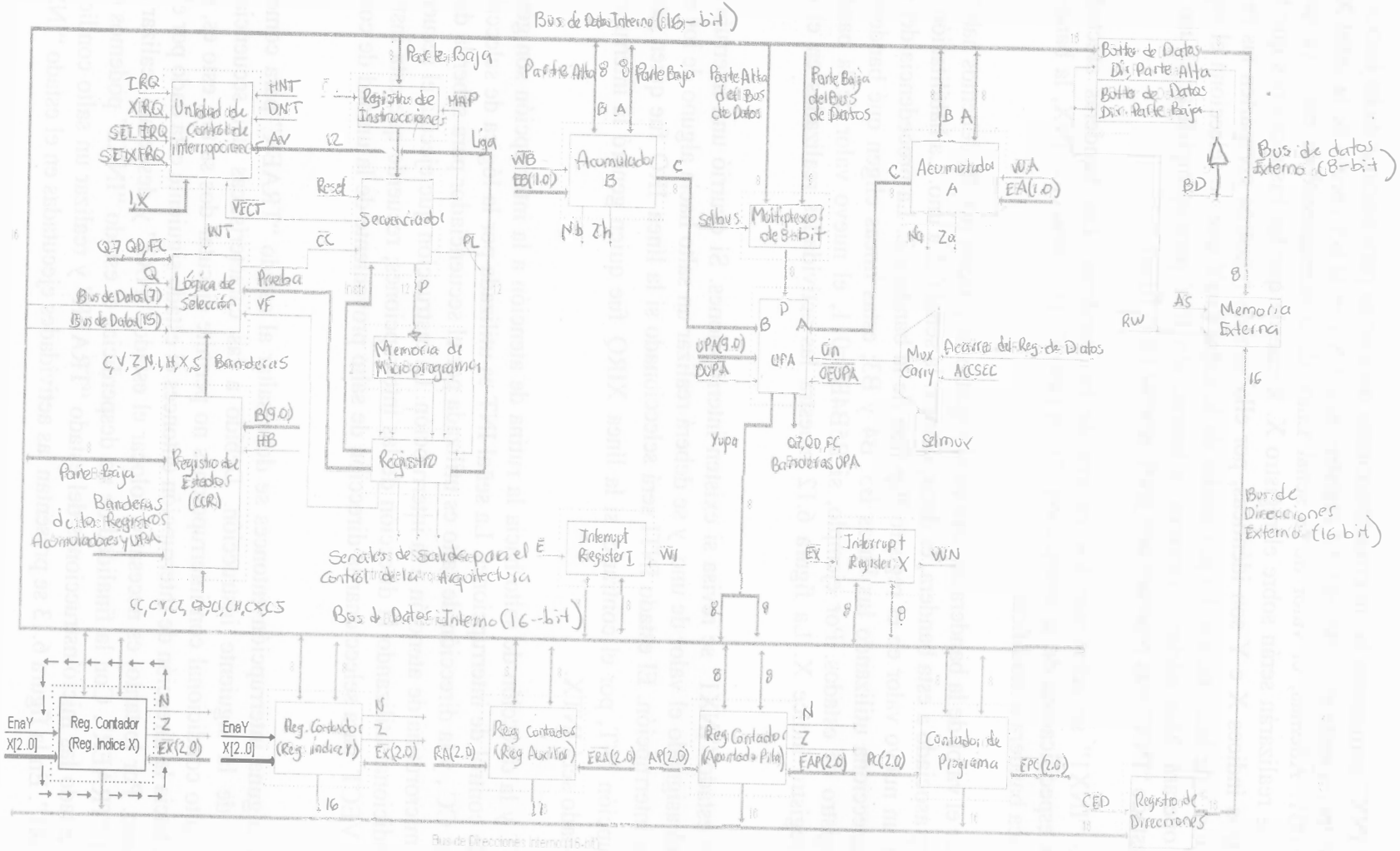


Figura 6.11. Primera fase de ejecución de la instrucción INX. Estado INX.

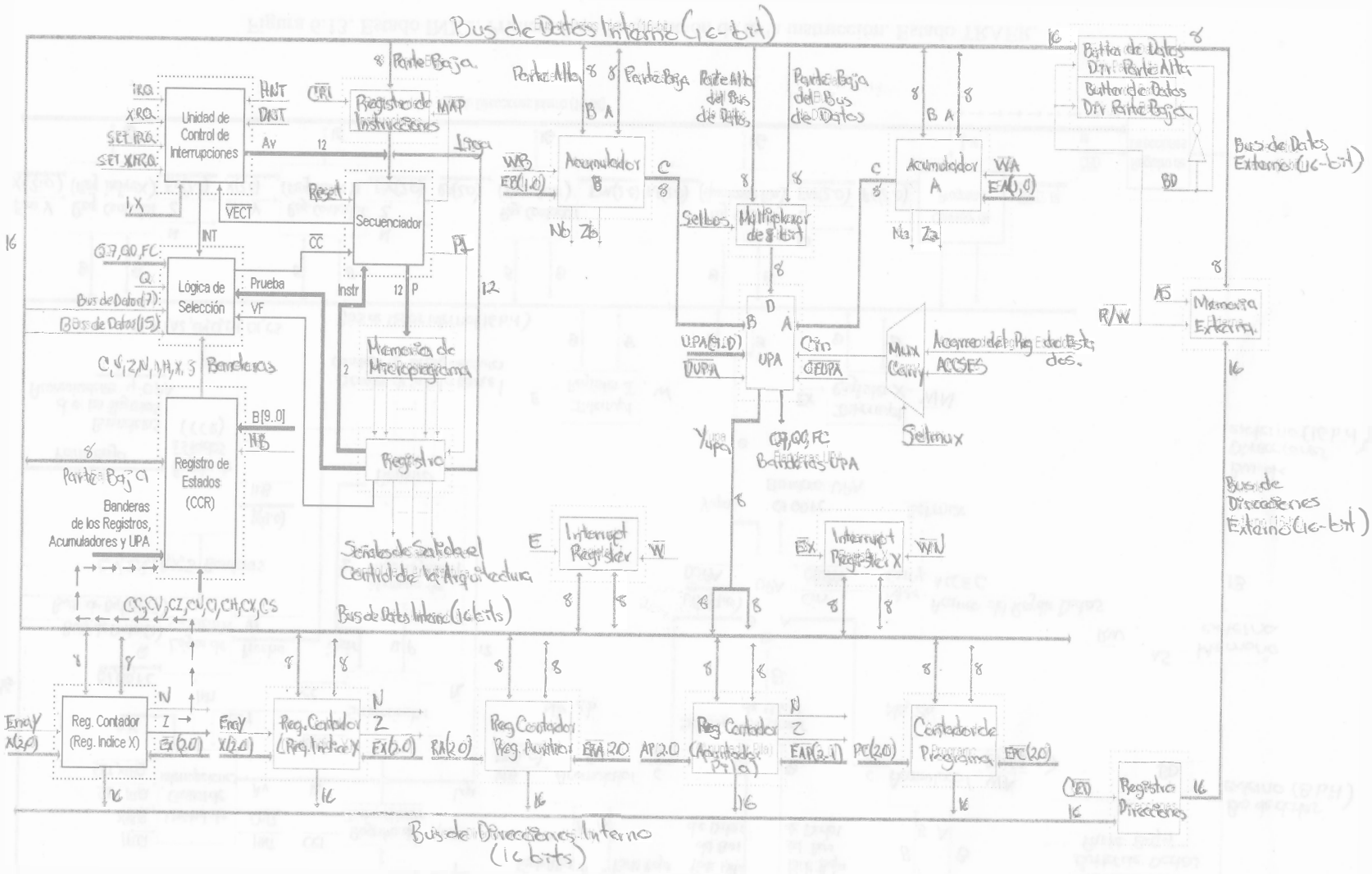


Figura 6.12. Última fase de ejecución de la instrucción INX. Estado INX1.

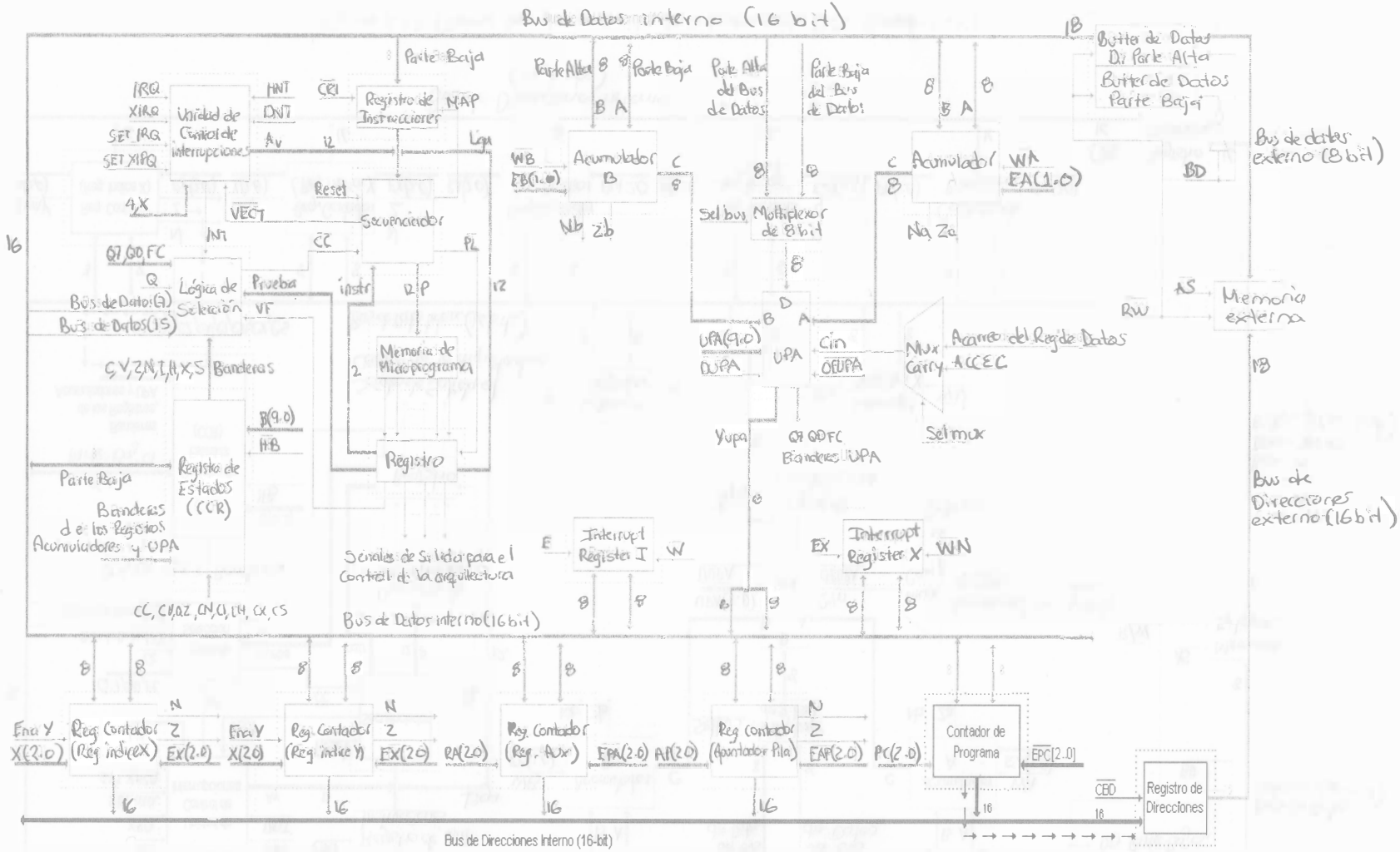


Figura 6.13. Estado INX2. Primera fase de ejecución de toda instrucción. Estado TRAER.

La tabla 6.1 muestra el contenido de la memoria de microprograma para la carta ASM de la figura 6.7. El tamaño de esta memoria es de 4096x100, es decir, 4096 palabras de 100 bits cada una. Por cuestiones de espacio sólo se muestran los bits correspondientes a los siguientes campos: I_1I_0 (la instrucción que ejecuta el secuenciador), Prueba, VF, Liga, y las salidas que intervienen durante la ejecución de las micro-operaciones para la instrucción INX. También por cuestiones de espacio, algunos campos se representan en formato hexadecimal, como Liga y B[9..0]. El resto de los campos se representan en formato binario.

A las variables INT y Q_x les fueron asignados ciertos códigos binarios para poderlas representar dentro del campo de Prueba: INT recibió el código 11111, mientras que Q_x recibió el código 00000. Además, se estableció el valor de Q_x a cero.

Observe que las variables negadas tienen un nivel lógico alto como valor por default. Si estas variables aparecen dentro de un estado en la carta ASM su valor cambia a cero, es decir, se activan. Las variables sin negar tienen un comportamiento inverso, presentan un nivel lógico bajo por default y cuando aparecen en algún estado de las cartas ASM cambian a uno.

DIRECCIÓN	CONTENIDO DE LA MEMORIA															
	Instrucción Secuenciador I1I0	Prueba	VF	Liga	Salidas que controlan la Arquitectura											
					$\overline{EPC}$[2..0]	PC[2..0]	$\overline{CBD}$	$\overline{AS}$	$\overline{R/W}$	$\overline{CRI}$	$\overline{EX}$[2..0]	X[2..0]	EnaY	B[9..0]	CZ	...
0000 0000 1000	00	00000	0	000h	011	000	0	1	1	1	111	000	0	000h	0	...
0000 0000 1001	00	00000	0	000h	111	001	1	0	1	0	111	000	0	000h	0	...
0000 0000 1010	10	00000	0	000h	111	000	1	1	1	1	111	000	0	000h	0	...
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
0000 1000 0000	00	00000	0	000h	111	000	1	1	1	1	111	001	0	000h	0	...
0000 1000 0001	11	11111	1	000h	111	000	1	1	1	1	111	000	0	018h	1	...
0000 1000 0010	01	00000	0	009h	011	000	0	1	1	1	111	000	0	000h	0	...
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...

Tabla 6.1. Contenido de la memoria de microprograma para la instrucción INX.

Una manera alternativa de construir la unidad de control de la computadora (UCC) es usando los lenguajes de descripción de hardware, como por ejemplo Verilog HDL. Entonces, la codificación de la carta ASM para la instrucción INX quedaría de la siguiente manera.

```
/*****************************************************************************
 * Módulo: Unidad de Control de la Computadora (UCC)                         *
 *         Este módulo ejecuta las micro-operaciones para la instrucción INX. Observe *
 *         que sólo se manejaron las señales de control generales y las señales de control *
 *         necesarias para la ejecución de la instrucción INX (en total 34 señales de salida). *
 *         Si desea controlar el resto de la arquitectura deberá anexar las señales de salida *
 *         restantes.                                                        *
 *****************************************************************************/
module UCC(clk, Reset, CCn, D, MAPn, VECTn, SA, SB, SC);

/***********************************************
 * Definición de las Señales de Entrada y de Salida *
 ***********************************************/
input           clk, Reset, CCn;
input    [11:0] D;
output          MAPn, VECTn;
output   [15:0] SA, SB;
output   [1:0]  SC;
reg      [15:0] SA, SB;
reg      [1:0]  SC;
reg      [11:0] EstadoPresente;
reg             MAPn, VECTn;


/************************************
 * Generación de las Señales de Salida *
 ************************************/
always @ (EstadoPresente) begin
    case (EstadoPresente)

        /*************************************************************************
         * Los vectores SA, SB y SC manejan las señales de salida en el siguiente orden. *
         *                                                                       *
         * El vector SA maneja 16 bits pertenecientes a las señales:             *
         *      Prueba[4:0] VF EPCn[2:0] PC[2:0] CBDn Asn R/Wn CRIn              *
         * El vector SB maneja 16 bits pertenecientes a las señales:             *
         *      EXn[2:0] X[2:0] EnaY B[9:1]                                      *
         * El vector SC maneja 2 bits pertenecientes a las señales:              *
         *      B[0] CZ                                                          *
         * Recuerde que la señal de más a la izquierda es la más significativa, mientras que la *
         * la señal de más a la derecha es la menos significativa.               *
         *************************************************************************/

        12'h008: // Salidas correspondientes al Estado TRAER
                 begin
                 SA = 16'b0000000110000111; SB = 16'b1110000000000000; SC = 2'b00;
                 MAPn = 1; VECTn = 1:
                 end
        12'h009: // Salidas correspondientes al Estado TRAER1
                 begin
                 SA = 16'b0000001110011010; SB = 16'b1110000000000000; SC = 2'b00;
                 MAPn = 1; VECTn = 1;
                 end
```

```
            12'h00A: // Salidas correspondientes al Estado DECOD
                     begin
                     SA = 16'b0000001110001111; SB = 16'b1110000000000000; SC = 2'b00;
                     MAPn = 0; VECTn = 1;
                     end
            12'h080: // Salidas correspondientes al Estado INX
                     begin
                     SA = 16'b0000001110001111; SB = 16'b1110000000000000; SC = 2'b00;
                     MAPn = 1; VECTn = 1;
                     end
            12'h081: // Salidas correspondientes al Estado INX1
                     begin
                     SA = 16'b1111111110001111; SB = 16'b1110000000001100; SC = 2'b01;
                     MAPn = 1; VECTn = 0;
                     end
            12'h082: // Salidas correspondientes al Estado INX2 = Estado TRAER
                     begin
                     SA = 16'b0000000110000111; SB = 16'b1110000000000000; SC = 2'b00;
                     MAPn = 1; VECTn = 1;
                     end
            default: // Salidas por default
                     begin
                     SA = 16'b0000001110001111; SB = 16'b1110000000000000; SC = 2'b00;
                     MAPn = 1; VECTn = 1;
                     end
        endcase
    end

    /*******************************************************
     * Instrucciones del Secuenciador y Transiciones entre Estados   *
     *******************************************************/
    always @ (posedge clk) begin
        if (Reset)
            EstadoPresente = 12'h008;
        else
            case (EstadoPresente)
                12'h008: // El secuenciador ejecuta la instrucción continúa
                         EstadoPresente = 12'h009;
                12'h009: // El secuenciador ejecuta la instrucción continúa
                         EstadoPresente = 12'h00A;
                12'h00A: // El secuenciador ejecuta la instrucción salto de transformación
                         EstadoPresente = D;
                12'h080: // El secuenciador ejecuta la instrucción continúa
                         EstadoPresente = 12'h081;
                12'h081: // Se ejecuta la instrucción salto condicional con interrupciones
                         if (CCn)       EstadoPresente = 12'h082;
                         else           EstadoPresente = D;
                12'h082: // El secuenciador ejecuta la instrucción salto condicional
                         EstadoPresente = 12'h008;
            endcase
    end

    endmodule
```

Cada una de las instrucciones en ensamblador podría ser programada de la forma anterior, por lo tanto, surge una interrogante:

¿Qué conviene más, construir la Unidad de Control de la Computadora utilizando el secuenciador y la memoria de microprograma, o bien, construirla utilizando los lenguajes de descripción de hardware y dejar que el compilador de tales lenguajes se encargue de generar las funciones lógicas que la describen?

Para contestar esta pregunta se tiene que tomar en consideración el número de elementos lógicos que se utilizan para cada técnica. Para el caso en donde se utiliza el secuenciador, la memoria de microprograma es la que ocupa mayor espacio. Si el repertorio de instrucciones que ejecuta nuestra computadora es muy grande, quizá resulte conveniente seguir el método del secuenciador, ya que si se utiliza el otro método, las ecuaciones lógicas creadas para generar las salidas que controlan a la arquitectura se vuelven cada vez más complejas. Por consiguiente, el segundo método requiere mayor número de compuertas lógicas a medida que aumenta el número de instrucciones en ensamblador.

Es responsabilidad del diseñador decidir qué método utilizar de acuerdo al espacio disponible en el circuito integrado con el que cuente.

6.3.2 INSTRUCCIÓN INY (Acceso Inherente)

Instrucción: INY
Operación: IY ⇐ (IY) + 1
Código[5]: 18 08
Descripción: Incrementa el contenido del registro Y en una unidad.
Banderas: Z=1 si el resultado del incremento es cero, Z=0 en caso contrario.

S	X	H	I	N	Z	V	C
—	—	—	—	—	↕	—	—

Formato: El primer byte del código de la instrucción corresponde al valor hexadecimal 0x18; este byte sirve para indicar que la próxima instrucción a ejecutar utiliza el registro índice Y. El segundo byte del código es el identificador de la instrucción a ejecutar, en este caso es un incremento.

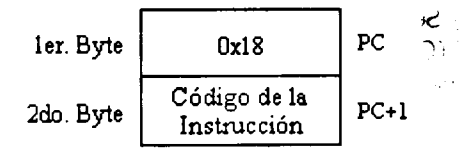


Figura 6.14. Formato de la instrucción INY.

[5] El código de operación de las instrucciones que utilizan el registro Y está precedido por el valor 0x18. Este valor le informa a la arquitectura que la siguiente operación a efectuar es sobre el registro Y.

La carta ASM de la figura 6.15 representa el algoritmo que ejecuta la instrucción INY.

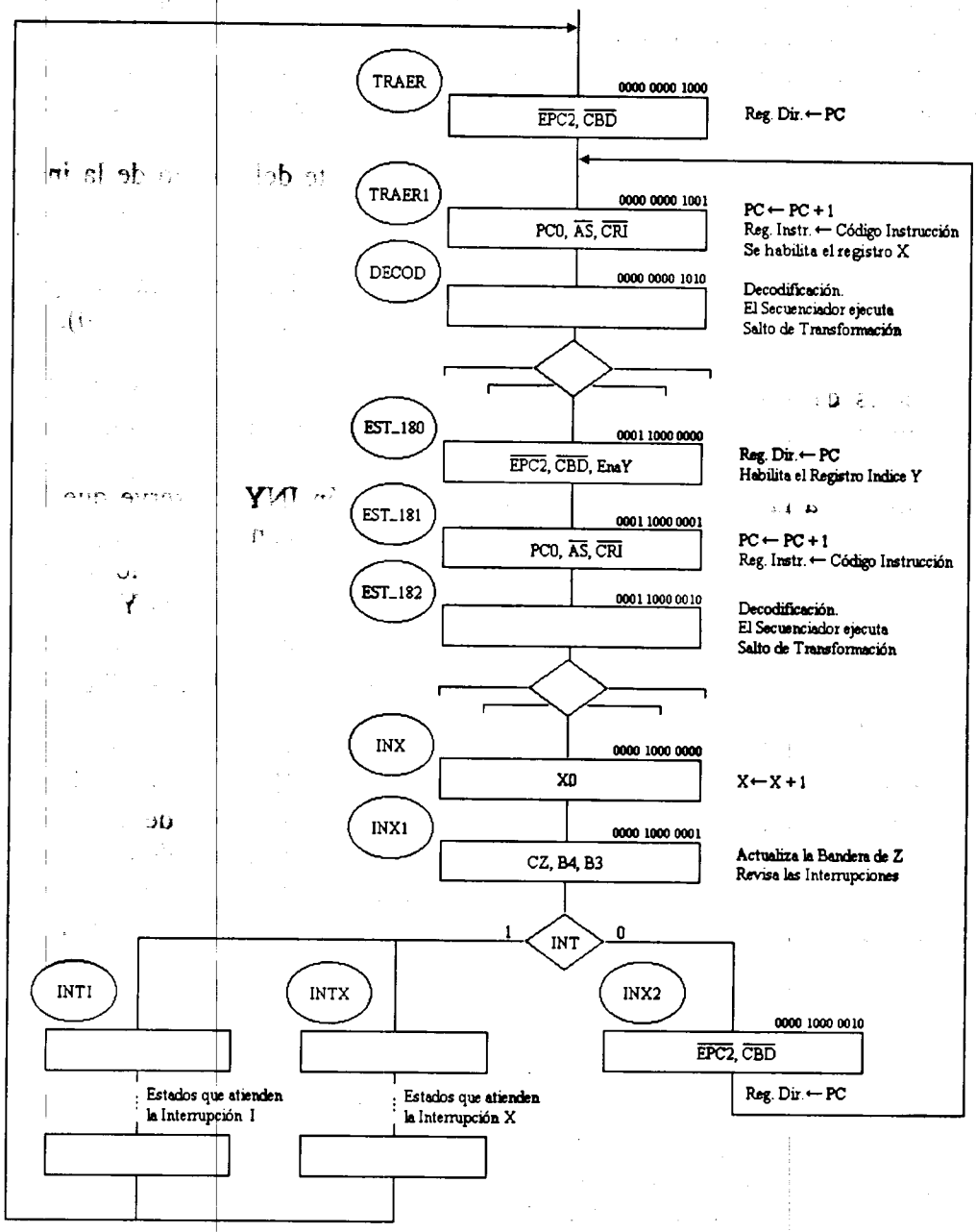


Figura 6.15. Carta ASM para la instrucción INY (acceso inherente).

Los estados "TRAER", "TRAER1" y "DECOD" son los mismos para todas las instrucciones, en ellos se trae de memoria el código de operación de la próxima instrucción a ejecutar y se decodifica dicha instrucción. Para esta instrucción en particular, en el estado "DECOD", se decodifica el primer byte de la instrucción INY (0x18) dando como resultado que el secuenciador realice un salto a la dirección 0001 1000 0000 (0x180) en la memoria de microprograma.

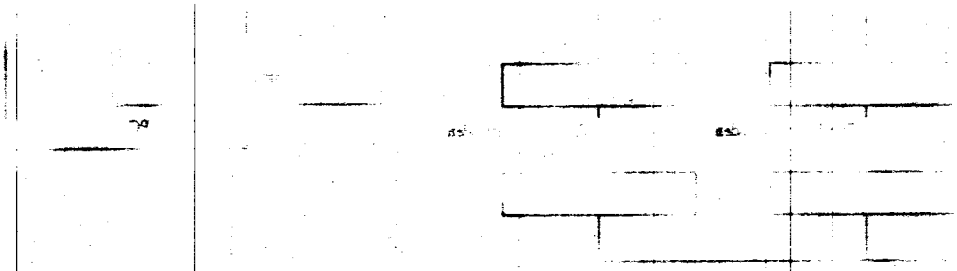
En el estado "EST_180" se habilita el registro índice Y, y se comienza un ciclo fetch para traer el segundo byte del código de operación de la instrucción. En este estado, la señal EnaY es la encargada de habilitar al registro Y, el cual permanecerá habilitado hasta que la ejecución de una nueva instrucción comience; durante este tiempo, el registro X estará deshabilitado. En este estado también son habilitadas las señales $\overline{EPC2}$ y $\overline{CBD}$ con el fin de cargar el contenido del registro PC (con la dirección en memoria del segundo byte de la instrucción) en el registro de direcciones.

En el estado "EST_181" se lee de memoria el segundo byte del código de la instrucción y se carga en el registro de instrucción. También es incrementado en una unidad el contenido del registro PC. Por otra parte, en el estado "EST_182" se decodifica el segundo byte del código de la instrucción, este valor le indica al secuenciador la dirección de inicio de las micro-operaciones para la instrucción INY; dicha dirección es la misma que para la instrucción INX (0x080).

En los siguientes estados de la instrucción INY se incrementa el contenido del registro, se actualizan banderas y se revisan las interrupciones, tal y como se realiza para la instrucción INX.

Ahora comparemos a la instrucción INX con la instrucción INY. Observe que los códigos de operación para las instrucciones sobre el registro X no cuentan con un precódigo que especifique explícitamente que las operaciones a ejecutar son sobre el registro X; por lo tanto, es necesario activar las operaciones sobre el registro X, y desactivarlas sobre el registro Y, antes de comenzar una nueva instrucción. De esta manera, si una instrucción utiliza al registro X, las operaciones sobre él estarán activadas; y si una instrucción utiliza al registro Y, entonces, el precódigo se encargará de activar al registro Y.

Como se mencionó con anterioridad, la activación del registro X y la desactivación del registro Y será al inicio de cada instrucción, específicamente en el estado 'TRAER1' del ciclo fetch, el cual es ejecutado por todas las instrucciones antes de empezar la ejecución de las micro-operaciones propias de la instrucción. Note que no hay conflictos para las instrucciones que utilizan el registro Y porque el precódigo de estas instrucciones será suficiente para habilitar al registro índice Y y desactivar al registro X.



6.3.3 INSTRUCCIÓN XGDX (Acceso Inherente)

Instrucción: XGDX
Operación: (IX) ⇔ (ACCD)
Código: 8F
Descripción: El contenido del acumulador D es transferido al registro índice X, y el contenido del registro índice X es transferido al doble acumulador D.
Banderas: Ninguna bandera es afectada.

S	X	H	I	N	Z	V	C
—	—	—	—	—	—	—	—

A continuación se presenta la carta ASM que ejecuta esta instrucción.

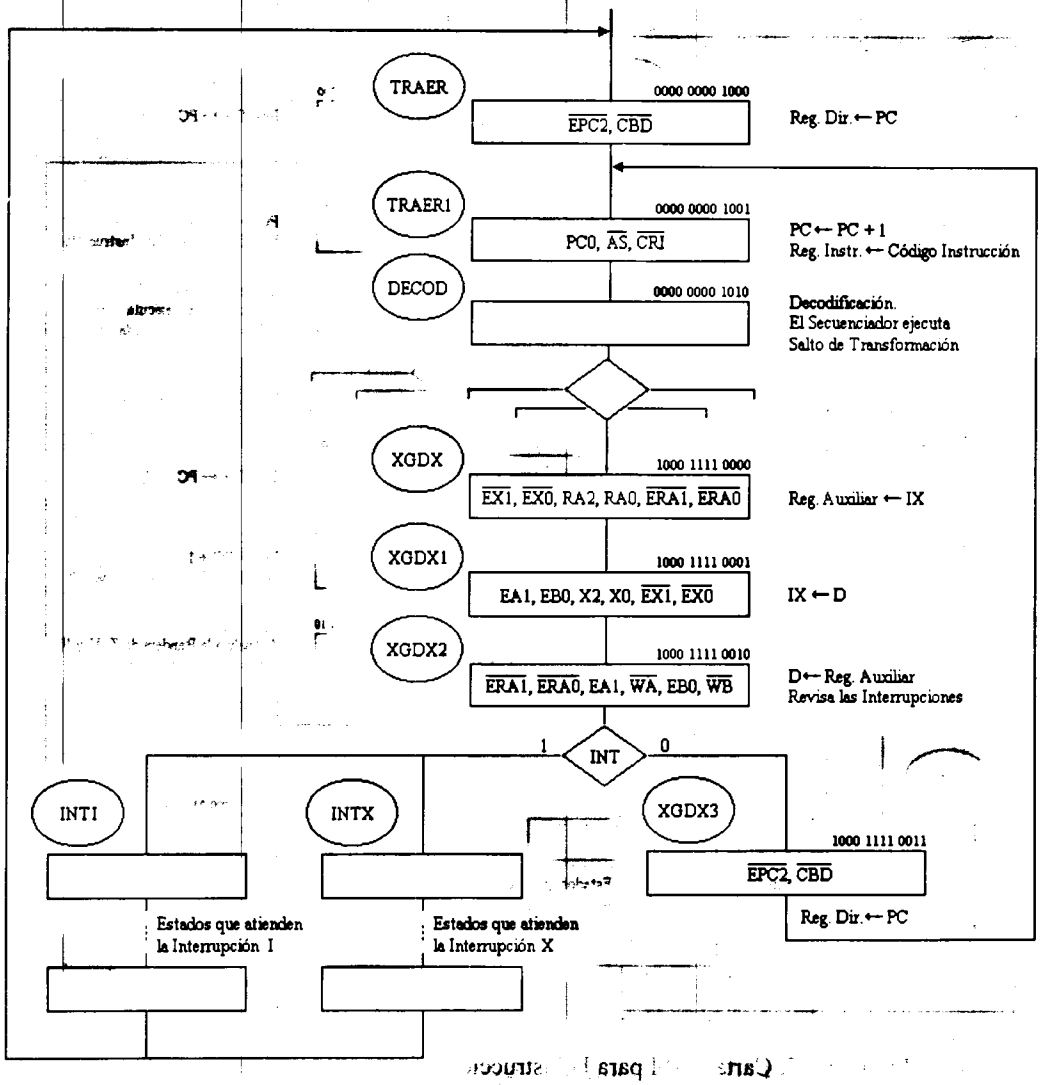


Figura 6.16. Carta ASM para la instrucción XGDX (acceso inherente).

6.3.4 INSTRUCCIÓN LDAB (Acceso Inmediato)

Instrucción: LDAB #DATO
Operación: ACCB ⇐ (Memoria)
Código: C6
Descripción: Carga en el acumulador B, un dato inmediato de 8 bits contenido en memoria.
Banderas: N=1 si el MSB[6] del resultado está encendido, N=0 en caso contrario.
Z=1 si el resultado en el registro es cero, Z=0 en caso contrario.
V se coloca a cero.

S	X	H	I	N	Z	V	C
—	—	—	—	↕	↕	0	—

A continuación se presenta la carta ASM que ejecuta esta instrucción.

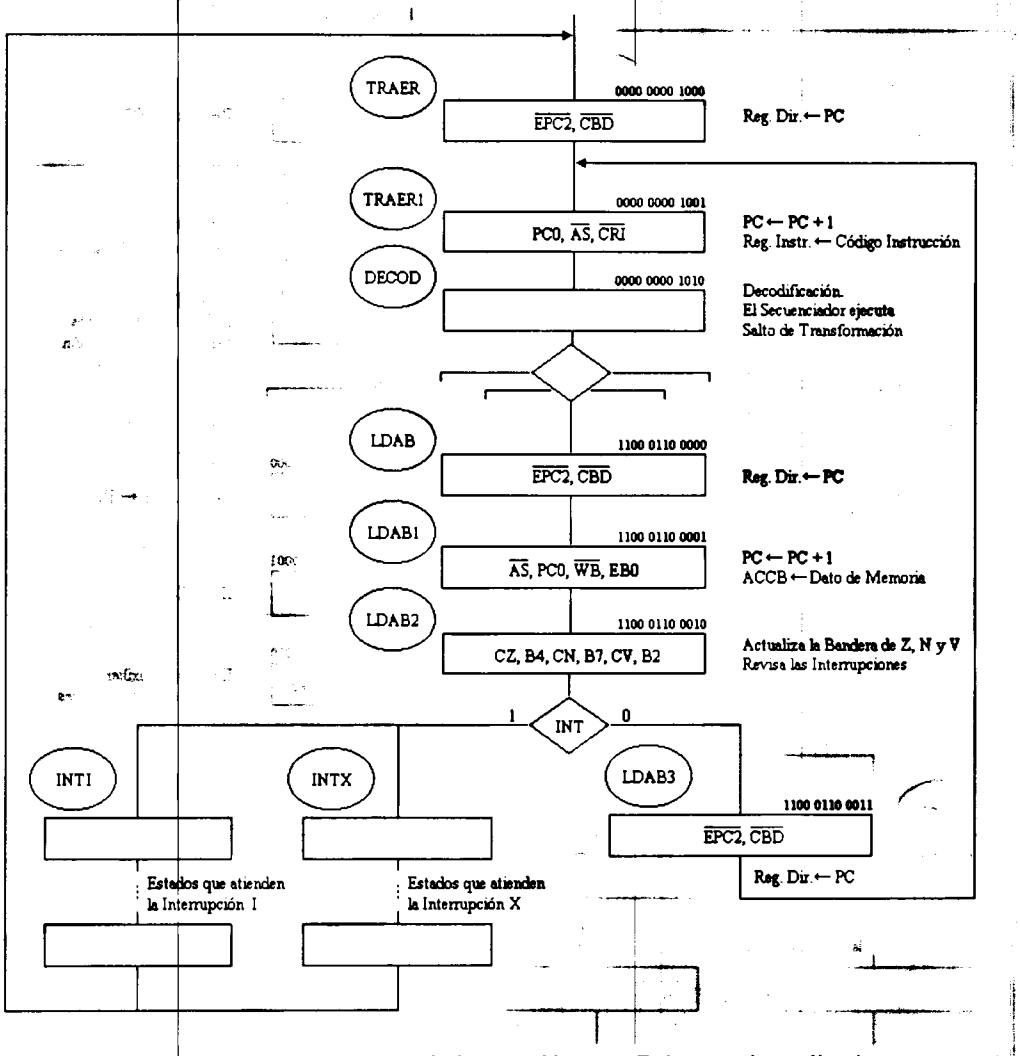


Figura 6.17. Carta ASM para la instrucción LDAB (acceso inmediato).

[6] MSB = Bit más significativo.

6.3.5 INSTRUCCIÓN LDAA (Acceso Inmediato)

Instrucción: LDAA #DATO
Operación: ACCA ⇐ (Memoria)
Código: 86
Descripción: Carga en el acumulador A, un dato inmediato de 8 bits contenido en memoria.
Banderas: N=1 si el MSB del resultado está encendido, N=0 en caso contrario.
Z=1 si el resultado en el registro es cero, Z=0 en caso contrario.
V se coloca a cero.

S	X	H	I	N	Z	V	C
—	—	—	—	↕	↕	0	—

A continuación se presenta la carta ASM que ejecuta esta instrucción.

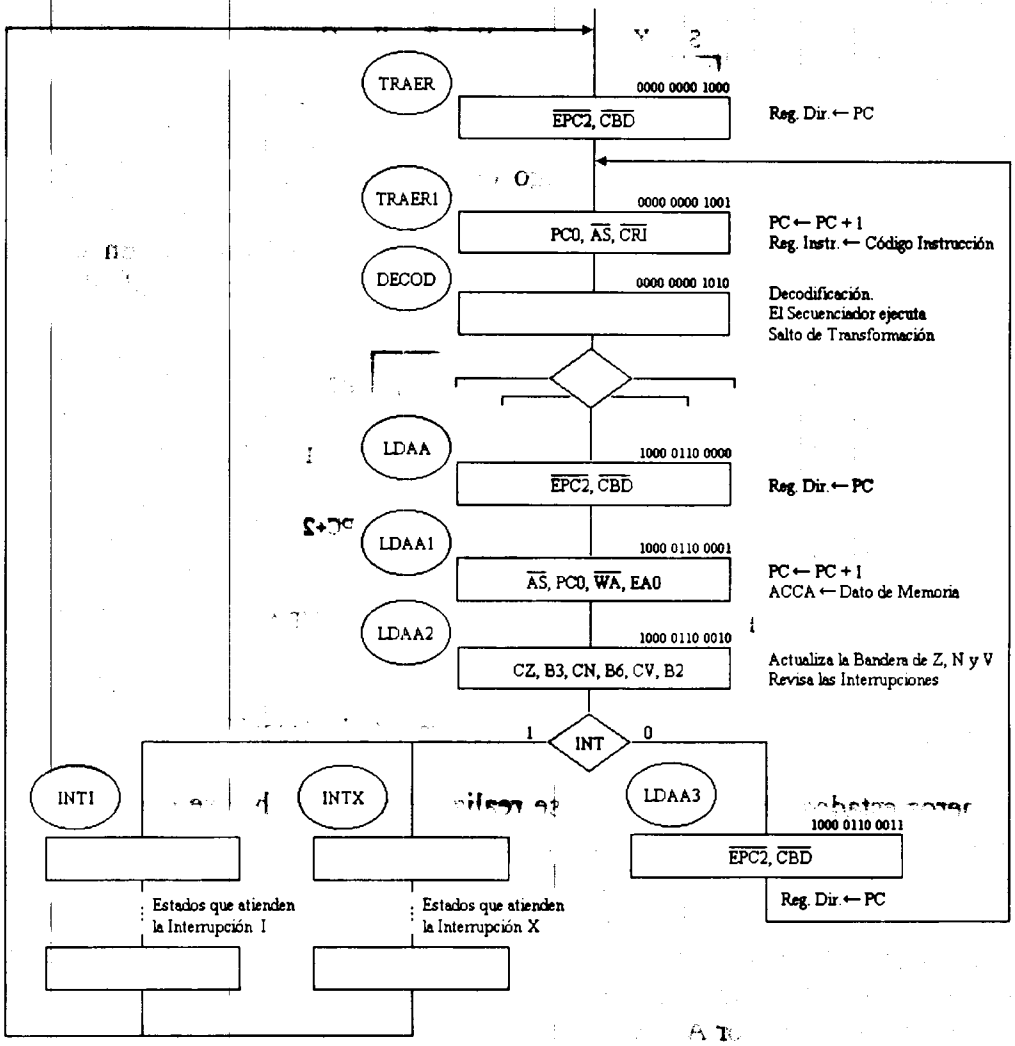


Figura 6.18. Carta ASM para la instrucción LDAA (acceso inmediato).

6.3.6 INSTRUCCIÓN SUBA (Acceso Extendido)

Instrucción: SUBA Dirección_16_Bits
Operación: ACCA ⇐ (ACCA) – (Memoria)
Código: B0
Descripción: Resta al contenido del acumulador A el contenido de la Memoria. El resultado es almacenado nuevamente en el acumulador A.

Banderas: N=1 si el MSB del resultado de la resta es igual a uno, N=0 en caso contrario.
Z=1 si el resultado de la resta es cero, Z=0 en caso contrario.
$V = ACCA7 \bullet \overline{MEM7} \bullet \overline{RES7} + \overline{ACCA7} \bullet MEM7 \bullet RES7$
$C = \overline{ACCA7} \bullet MEM7 + MEM7 \bullet RES7 + RES7 \bullet \overline{ACCA7}$

donde, ACCA7 = Bit más significativo del dato contenido en el acumulador A.
MEM7 = Bit más significativo del dato contenido en memoria.
RES7 = Bit más significativo del resultado de la resta.

S	X	H	I	N	Z	V	C
–	–	–	–	↕	↕	↕	↕

Formato: El primer byte del código de operación corresponde al valor hexadecimal 0xB0; el segundo byte es la parte alta de una localidad de memoria de 16 bits; y el tercer byte es la parte baja de dicha localidad En esta localidad de memoria se encuentra el dato que será restado al contenido del acumulador A.

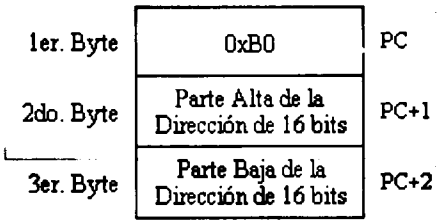


Figura 6.19. Formato de la instrucción SUBA.

En la figura 6.20 se presenta la carta ASM que ejecuta esta instrucción.

En los primeros estados de la carta ASM se realiza el ciclo fetch y se decodifica la instrucción SUBA. En los estados "SUBA" a "SUBA4" se obtiene la dirección en memoria del dato a restar, y se lee el dato contenido en dicha dirección.

En el estado "SUBA5" se efectúa la resta de los operandos, por lo tanto, se le indica a la unidad de procesos aritméticos (UPA) que reste al dato de la entrada A el dato de la entrada D, es decir, que reste al contenido del acumulador A el dato de la memoria. Recuerde que la operación de resta en la UPA se define de la siguiente manera: $S - R - \overline{Cin}$; por lo tanto, es necesario colocar el valor de Cin a cero para que el resultado de la resta no sea afectado por un valor de $\overline{Cin}$ indeseado. Para ello, se

activa la salida ACCSEC, y mediante la activación de señal Selmux se selecciona a ACCSEC como el acarreo de entrada a la UPA.

En el último estado, "SUBA6", se actualizan los valores de las banderas de negativo (N), cero (Z), sobreflujo (V) y acarreo (C) de acuerdo a los valores calculados por la UPA, y se verifica si existe alguna interrupción.

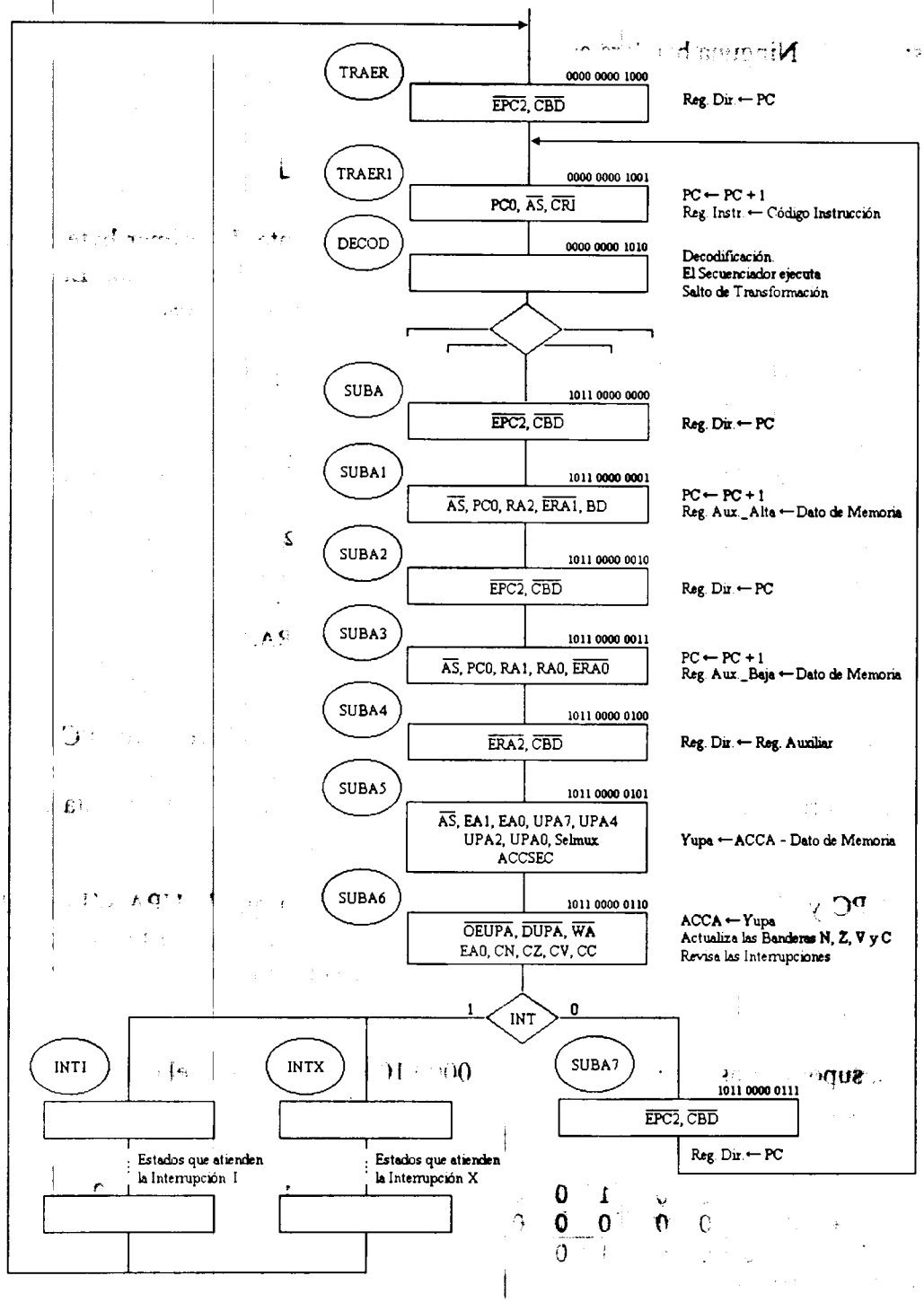


Figura 6.20. Carta ASM para la instrucción SUBA (acceso extendido).

6.3.7 INSTRUCCIÓN BRA (Acceso Relativo)

Instrucción: BRA Desplazamiento
Operación: PC ⇐ (PC) + 2 + Desplazamiento
Código: 20
Descripción: Salto incondicional a la dirección: PC + 2 + Desplazamiento. Donde el Desplazamiento es un número en complemento a dos.

Banderas: Ninguna bandera es afectada.

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Formato: El formato de esta instrucción es el siguiente. El primer byte corresponde al código de operación de la instrucción, es decir, 0x20. El segundo byte corresponde al desplazamiento en complemento a dos.

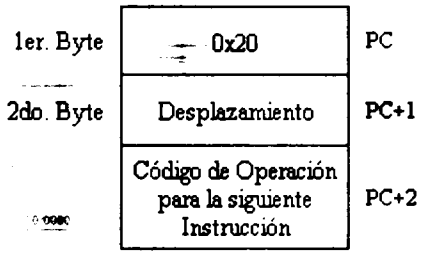


Figura 6.21. Formato de la instrucción BRA.

La dirección de salto se obtiene sumando al contenido del registro PC el valor del desplazamiento. Antes de calcular esta dirección de salto es necesario que PC apunte a la dirección de la siguiente instrucción en memoria, es decir, PC+2. Una vez que PC apunta a la dirección correcta podremos sumarle el desplazamiento.

La suma de PC y el desplazamiento se hace en dos pasos, ya que la UPA sólo puede efectuar operaciones de 8 bits y PC es de 16 bits. En el primer paso se suma a la parte baja del PC el valor del desplazamiento, y en el segundo paso, el valor del acarreo resultante de la primera operación se suma o se resta a la parte alta del PC, según el signo del desplazamiento.

Por ejemplo, supongamos que PC+2 es igual a 0000 1000 1111 1100 y el desplazamiento es igual a 0000 0100 (un desplazamiento positivo), por lo tanto, el nuevo valor de PC será el siguiente.

```
  0 0 0 0  1 0 0 0  1 1 1 1  1 1 0 0
+ 0 0 0 0  0 0 0 0  0 0 0 0  0 1 0 0
-------------------------------------
  0 0 0 0  1 0 0 1  0 0 0 0  0 0 0 0
```

Esta suma se puede hacer en dos partes:

$PC_{Baja} \leftarrow PC_{Baja} + Desplazamiento$
$PC_{Alta} \leftarrow PC_{Alta} + Acarreo\ (de\ la\ suma\ anterior)$

Ahora suponga un desplazamiento negativo. La suma es la siguiente.

$$\begin{array}{cccccccccccccccc} & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ + & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ \hline 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \end{array}$$

Esta suma también se puede hacer en dos partes:

$PC_{Baja} \leftarrow PC_{Baja} + Desplazamiento$
$PC_{Alta} \leftarrow PC_{Alta} + (-1) + Acarreo\ (de\ la\ suma\ anterior)$

Si el acarreo es cero:

$$PC_{Alta} \leftarrow PC_{Alta} + (-1) + 0 = PC_{Alta} - \overline{Acarreo}$$

Si el acarreo es uno:

$$PC_{Alta} \leftarrow PC_{Alta} + (-1) + 1 = PC_{Alta} + 0 = PC_{Alta} - \overline{Acarreo}$$

Entonces, para ambos casos:

$$PC_{Alta} \leftarrow PC_{Alta} - \overline{Acarreo}$$

En la figura 6.22 se presenta la carta ASM que ejecuta esta instrucción.

Los primeros tres estados de la carta ASM sirven para traer el código de operación de la instrucción y para decodificarla.

En el estado "BRA" guardamos una copia del registro de estados en la parte baja del registro auxiliar, con el fin de no alterar las banderas cuando se realice la suma de PC más el desplazamiento. La activación de las señales $\overline{HB}$, $\overline{ERA0}$, RA1 y RA0 permiten guardar el contenido del registro de estados en la parte baja del registro auxiliar. En este mismo estado se guarda en el registro de direcciones la dirección en memoria del desplazamiento, la cual está almacenada en PC. Esto se realiza mediante la activación de las señales $\overline{EPC2}$ y $\overline{CBD}$.

En el estado "BRA1" se guarda el desplazamiento leído de memoria en el registro Q de la UPA, y se incrementa en una unidad el contenido de PC. En este momento PC apunta a la dirección en memoria de la siguiente instrucción a ejecutar, así que ya se tiene el valor correcto de PC para calcular el salto. Recuerde que en el estado "TRAER1" se incrementó por primera vez el PC.

También observe que la única forma de guardar el valor del desplazamiento en el registro Q, sin alterar el contenido de los acumuladores ni el valor del desplazamiento, es a través de la entrada D de la UPA y aplicando la función lógica OR entre el desplazamiento y el valor de cero.

En el estado "BRA2" se suma a la parte baja del PC el valor del desplazamiento y el resultado se guarda en el registro Y_{upa} de la UPA (el contenido del registro Q no se modifica). Note que el acarreo de entrada a la UPA para esta operación es ACCSEC=0, el cual se selecciona por medio de la línea Selmux.

En el estado "BRA3" se guarda el resultado de la suma en la parte baja de PC y se almacena en el registro de estados el valor de la bandera de acarreo obtenido en la suma. Tenga presente que en este estado la señal $\overline{DUPA}$ es habilitada para no modificar el resultado calculado en el estado anterior. Por otra parte, la señal $\overline{OEUPA}$ habilita el bus de salida de la UPA para que el resultado pueda ser cargado en la parte baja de PC. En este estado también se pregunta por el valor del bit más significativo del desplazamiento, ya que este valor determina si se suma ó se resta, el valor del acarreo a la parte alta de PC.

Si el bit más significativo del desplazamiento es cero entonces se transita al estado "BRA4". En este estado se suma a la parte alta de PC el valor del acarreo del registro de estados. Recuerde que este acarreo se obtuvo cuando fueron sumados la parte baja de PC y el desplazamiento. Por el contrario, si el bit más significativo del desplazamiento es uno, entonces, se transita al estado "BRA8". En "BRA8" se resta a la parte alta de PC el valor negado del acarreo guardado en el registro de estados.

En el estado "BRA5" se guarda el resultado de la segunda suma ó de la resta en la parte alta de PC, para ello se activan las señales PC2, $\overline{OEUPA}$, $\overline{DUPA}$ y $\overline{EPC1}$. En el estado "BRA6" la activación de las señales $\overline{ERA0}$, CC y B0, permiten restaurar el valor original de la bandera de acarreo en el registro de estados. También en este estado se revisa la existencia de interrupciones, si existe alguna interrupción, entonces, se salta al estado de atención a la interrupción, si no, se salta al estado "BRA7".

En "BRA7" se ejecuta la primera instrucción del ciclo Fetch y se salta al estado "TRAER1".

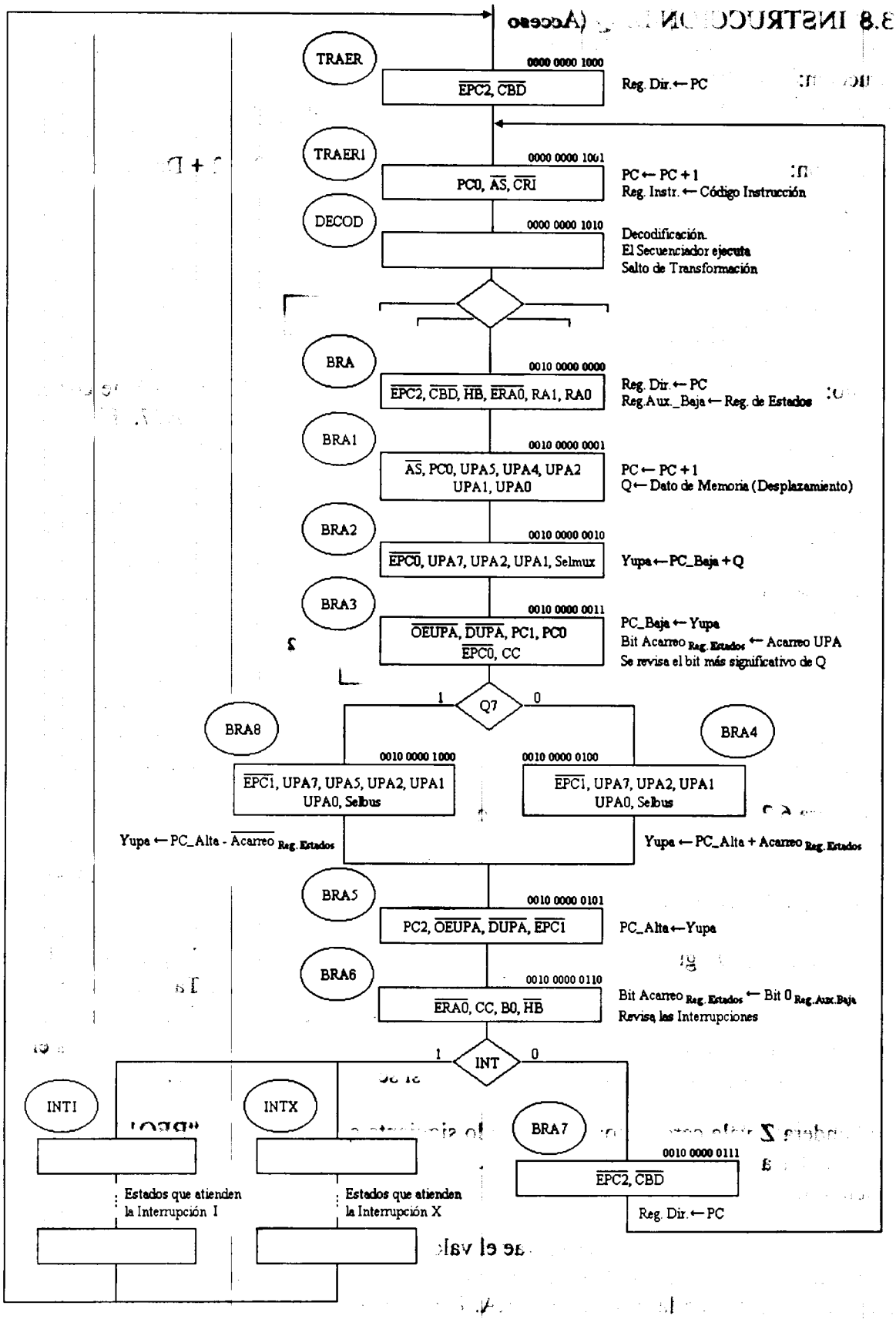


Figura 6.22. Carta ASM para la instrucción BRA (acceso relativo).

6.3.8 INSTRUCCIÓN BEQ (Acceso Relativo)

Instrucción: BEQ Desplazamiento
Operación: Si Z=1, PC ⇐ (PC) + 2 + Desplazamiento
Código: 27
Descripción: Salto condicional. Si Z=1 entonces PC ⇐ (PC) + 2 + Desplazamiento, si Z=0 entonces PC ⇐ (PC) + 2.
Banderas: Ninguna bandera es afectada.

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Formato: El formato de esta instrucción es el siguiente. El primer byte corresponde al código de operación de la instrucción, es decir, 0x27. El segundo byte corresponde al desplazamiento en complemento a dos.

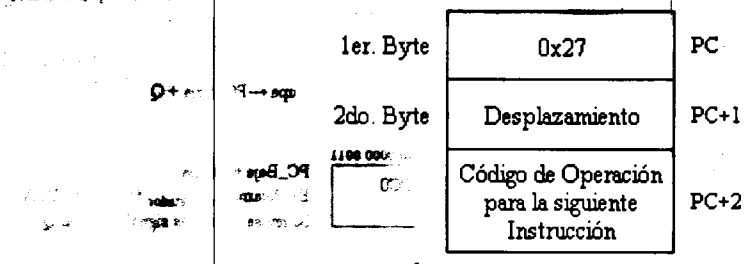


Figura 6.23. Formato de la instrucción BEQ.

En la figura 6.24 se presenta la carta ASM que ejecuta esta instrucción.

Los primeros tres estados de la carta ASM sirven para traer el código de operación de la instrucción y para decodificarla.

En el estado "BEQ" guardamos una copia del registro de estados en la parte baja del registro auxiliar, para ello, son activadas las señales $\overline{HB}$, $\overline{ERA0}$, RA1 y RA0. También se guarda en el registro de direcciones la dirección en memoria del desplazamiento, la cual está almacenada en PC. Esto se realiza mediante la activación de las señales $\overline{EPC2}$ y $\overline{CBD}$. Por último, se revisa el valor de la bandera de cero (Z), ya que de este valor depende si se efectúa o no el salto.

Si la bandera Z vale cero, entonces el estado siguiente es "BEQ1". En "BEQ1" se incrementa el PC para apuntar a la dirección en memoria de la siguiente instrucción, y se revisa la existencia de interrupciones.

Si la bandera Z vale uno, entonces se trae el valor del desplazamiento, y se calcula la dirección de salto sumando al PC el valor de dicho desplazamiento, es decir, se comienzan a ejecutar las mismas micro-operaciones que en la instrucción BRA. Por lo tanto, se podría saltar directamente al estado de inicio de la instrucción BRA; sin embargo, para aprovechar eficientemente el uso de los estados,

en el estado "BEQ", además de preguntar por la bandera Z, también se ejecutan las micro-operaciones del estado "BRA", de manera que si Z es igual a uno, entonces se salta al estado "BRA1". Recuerde que menos estados en las cartas ASM se traduce en computadoras más rápidas.

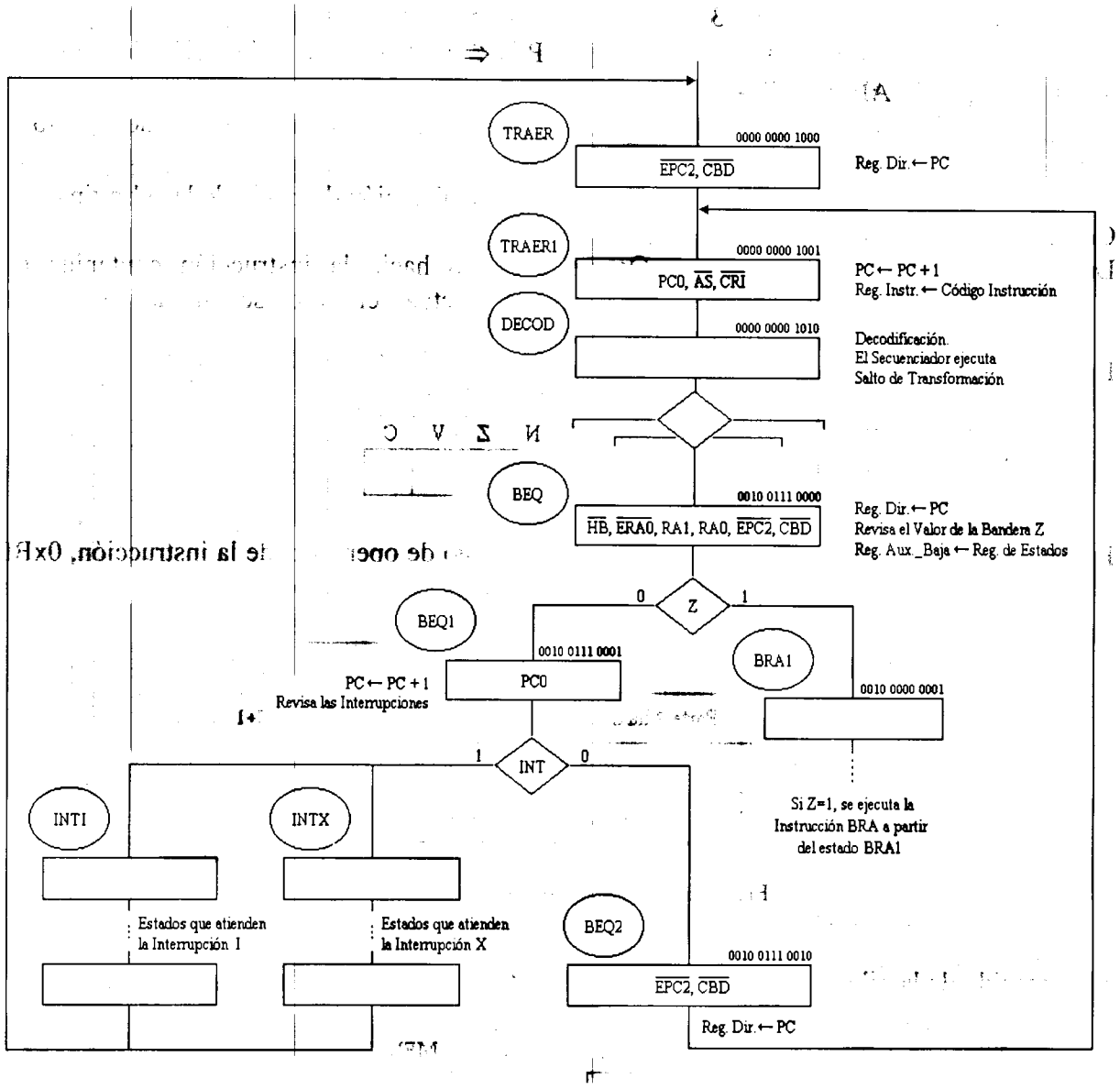
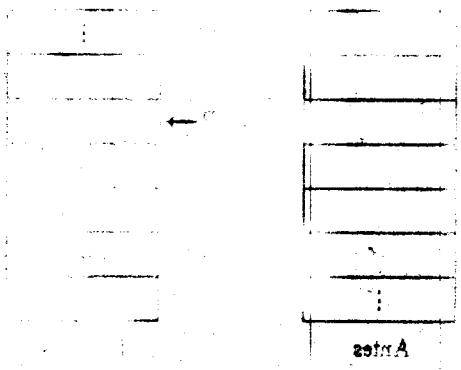


Figura 6.24. Carta ASM para la instrucción BEQ (acceso relativo).



6.3.9 INSTRUCCIÓN JSR (Acceso Extendido)

Instrucción: JSR Dirección_Subrutina_16bits

Operación:

PC ⇐ (PC) + 3	
⇓ (PC $_{Baja}$)	Pila ⇐ Parte baja de la dirección de regreso
AP ⇐ (AP) - 1	
⇓ (PC $_{Alta}$)	Pila ⇐ Parte alta de la dirección de regreso
AP ⇐ (AP) - 1	
PC ⇐ Dirección	PC ⇐ Dirección de inicio de la subrutina

Código: BD

Descripción: Salto a subrutina. Ocurre un salto hacia la instrucción contenida en la dirección de 16 bits. Antes de efectuar el salto, se guarda en la pila, la dirección de regreso de la subrutina.

Banderas: Ninguna bandera es afectada.

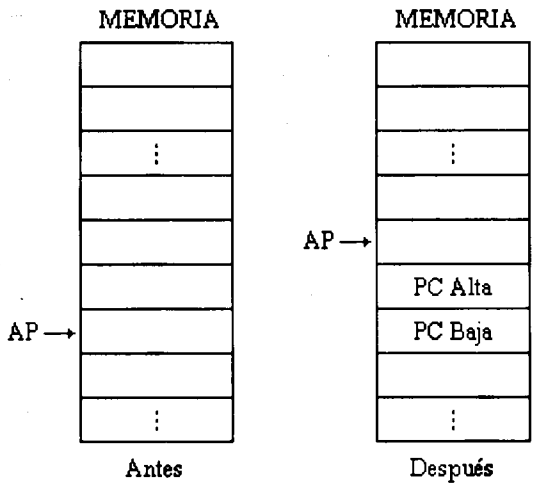
S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Formato: El primer byte corresponde al código de operación de la instrucción, 0xBD, y los dos bytes siguientes a la dirección de 16 bits donde comienza la subrutina.

1er. Byte	0xBD	PC
2do. Byte	Parte Alta de la Dirección de la Subrutina	PC+1
3er. Byte	Parte Baja de la Dirección de la Subrutina	PC+2
	Código de Operación para la siguiente Instrucción	PC+3

Figura 6.25. Formato de la instrucción JSR.

El contenido de la pila después de ejecutar está instrucción queda de la siguiente manera.



A continuación se presenta la carta ASM que ejecuta esta instrucción.

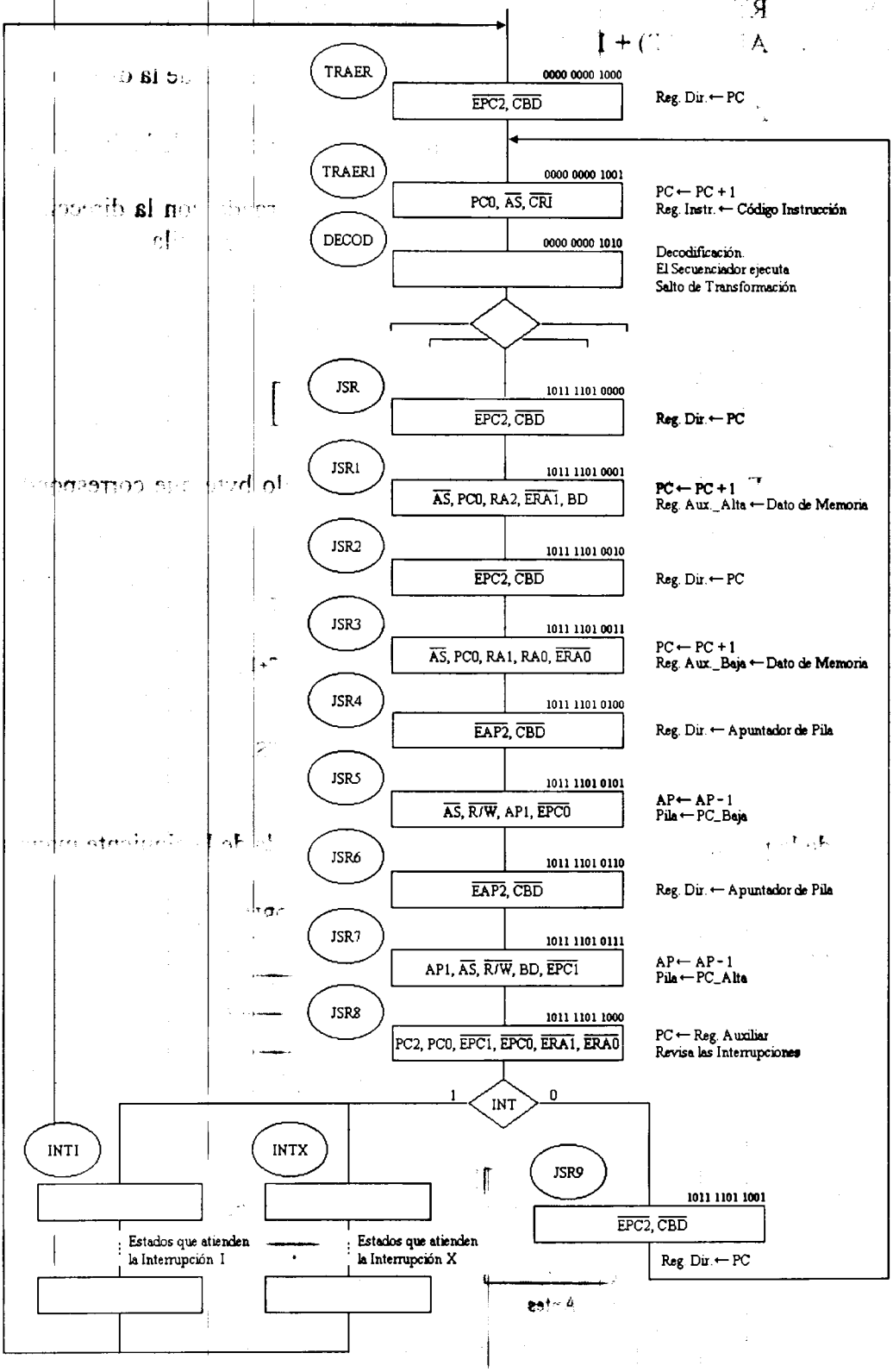


Figura 6.26. Carta ASM para la instrucción JSR (acceso extendido).

6.3.10 INSTRUCCIÓN RTS (Acceso Inherente)

Instrucción: RTS

Operación: AP ⇐ (AP) + 1
⇑ (PC Alta) — Recupera la parte alta de la dirección de regreso
AP ⇐ (AP) + 1
⇑ (PC Baja) — Recupera la parte baja de la dirección de regreso

Código: 39

Descripción: Regreso de Subrutina. El registro PC es cargado con la dirección de regreso de la subrutina, la cual se encuentra guardada en la pila.

Banderas: Ninguna bandera es afectada.

S	X	H	I	N	Z	V	C
—	—	—	—	—	—	—	—

Formato: Esta instrucción está compuesta por un sólo byte que corresponde al código de operación de la instrucción, 0x39.

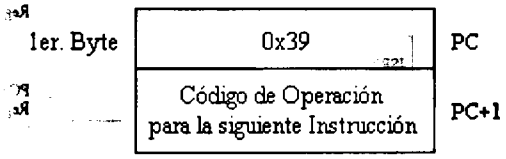
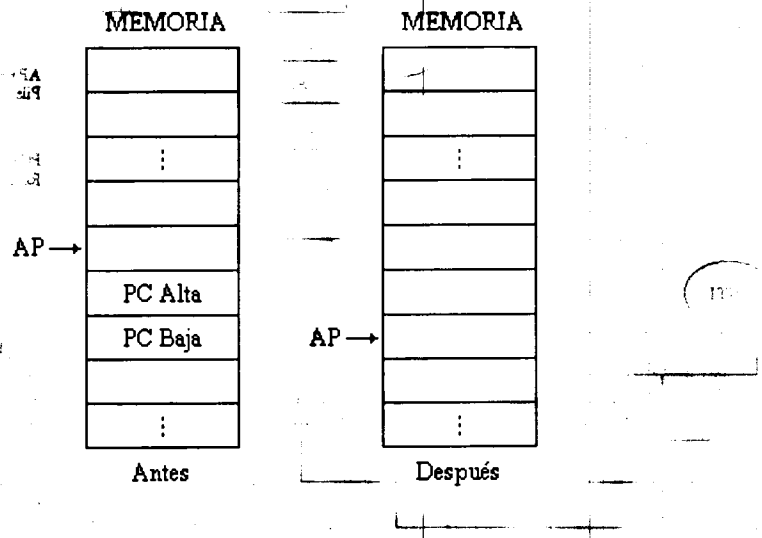


Figura 6.27. Formato de la instrucción RTS.

El contenido de la pila después de ejecutar está instrucción queda de la siguiente manera.



A continuación se presenta la carta ASM que ejecuta esta instrucción.

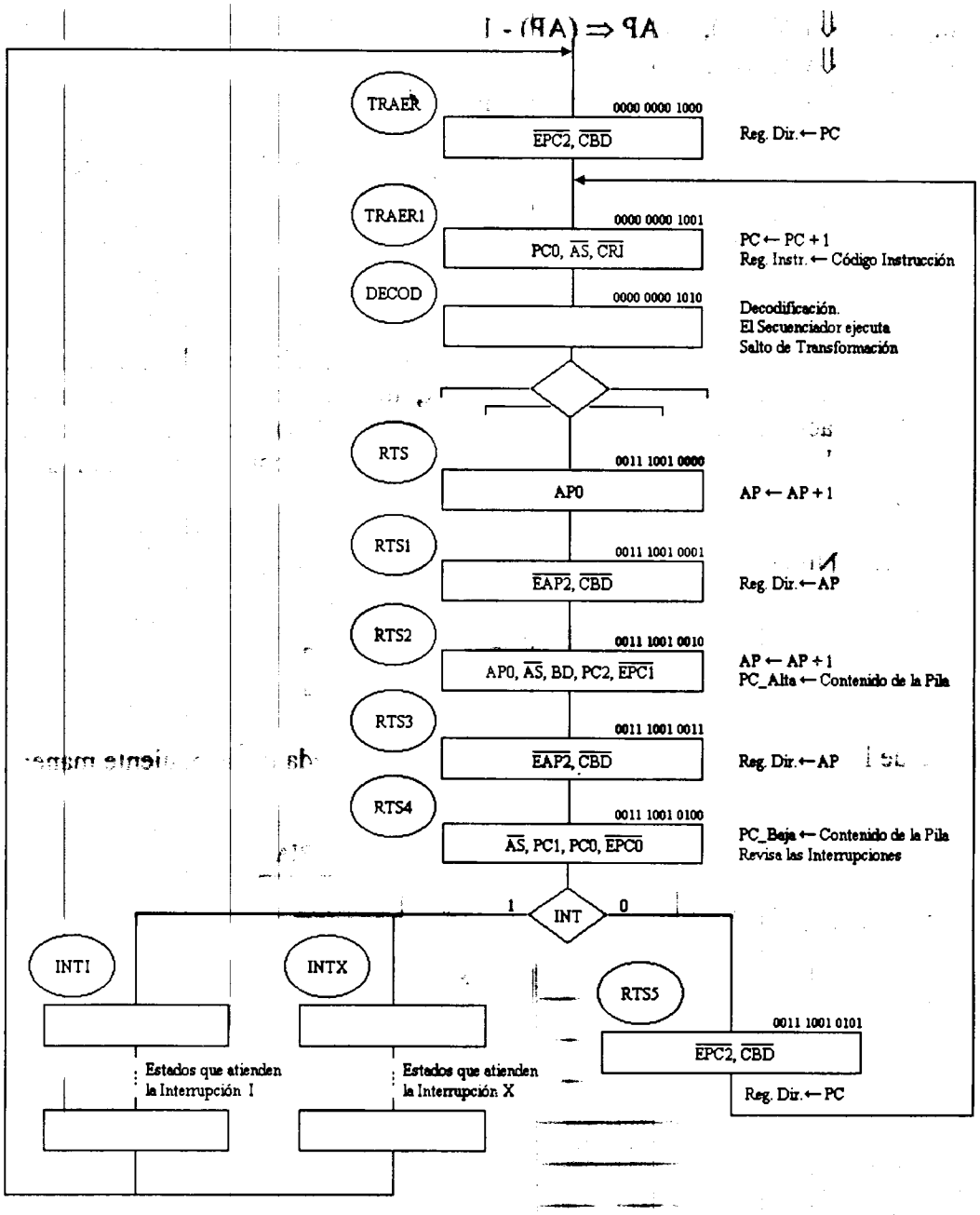


Figura 6.28. Carta ASM para la instrucción RTS (acceso inherente).

6.3.11 ATENCIÓN A INTERRUPCIONES

Operación:

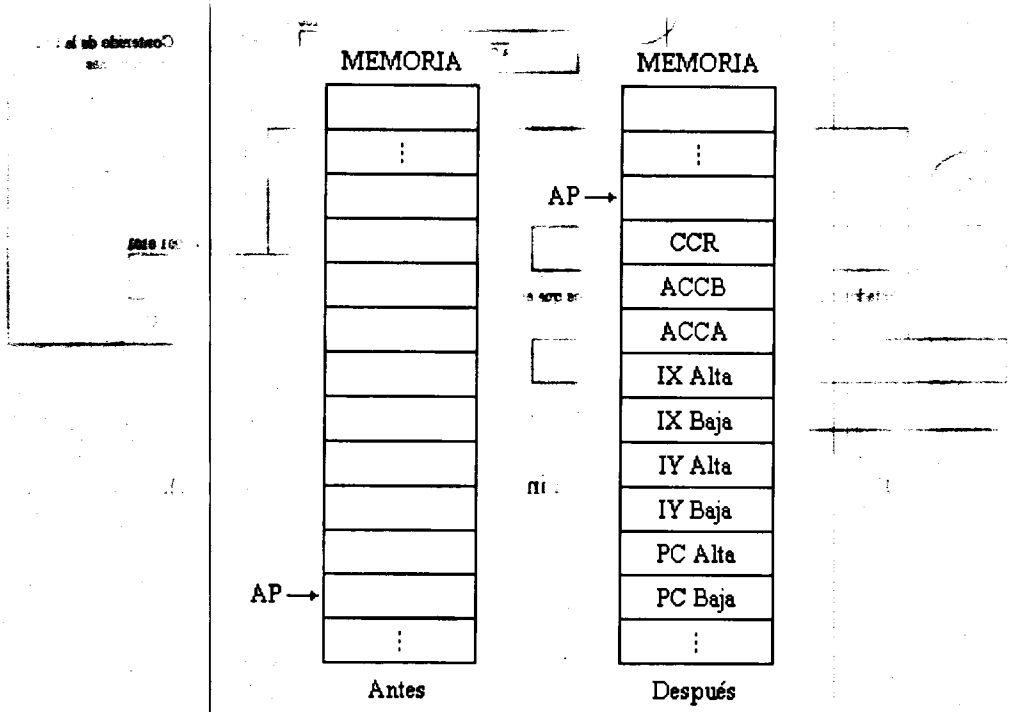
⇓ (PC Baja), AP ⇐ (AP) - 1
⇓ (PC Alta), AP ⇐ (AP) - 1
⇓ (IY Baja), AP ⇐ (AP) - 1
⇓ (IY Alta), AP ⇐ (AP) - 1
⇓ (IX Baja), AP ⇐ (AP) - 1
⇓ (IX Alta), AP ⇐ (AP) - 1
⇓ (ACCA), AP ⇐ (AP) - 1
⇓ (ACCB), AP ⇐ (AP) - 1
⇓ (CCR), AP ⇐ (AP) - 1

Descripción: Antes de atender una petición de interrupción se debe guardar el contenido del registro PC, del registro IX, del registro IY, del acumulador A, del acumulador B y del registro de estados (CCR). Una vez guardado el status de la arquitectura, entonces cargamos el PC con la dirección de inicio del manejador que atiende a la interrupción.

Banderas: Ninguna bandera es afectada.

S	X	H	I	N	Z	V	C
—	—	—	—	—	—	—	—

El contenido de la pila después de ejecutar está instrucción queda de la siguiente manera.



A continuación se presenta la carta ASM para las interrupciones.

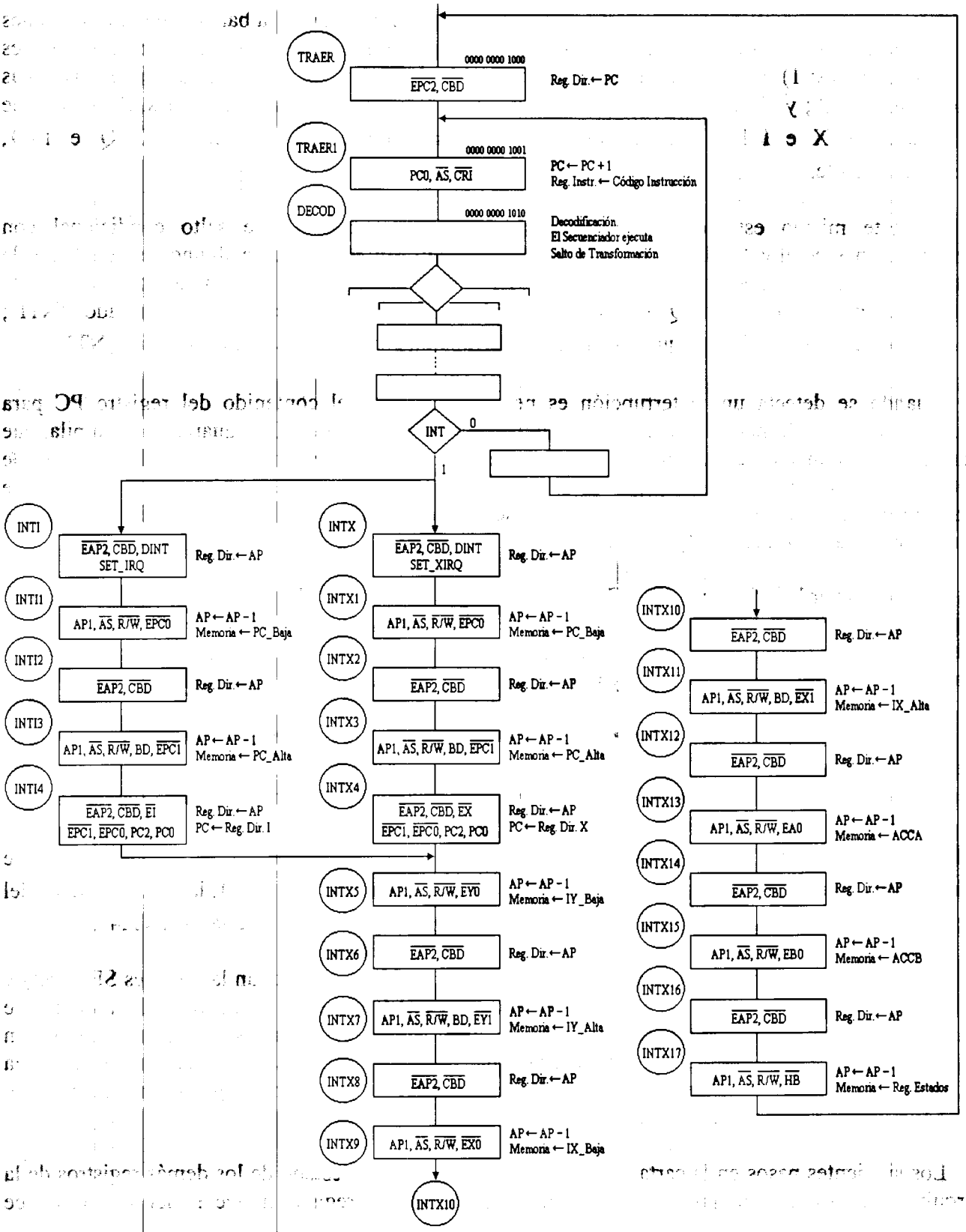


Figura 6.29. Carta ASM para atender interrupciones.

En el penúltimo estado de cada instrucción en ensamblador se revisa la bandera de interrupciones (INT). Esta variable valdrá uno si ha ocurrido una interrupción, es decir, si se cumplen las siguientes dos condiciones: 1) hubo una transición de un nivel lógico alto a un nivel lógico bajo en las líneas $\overline{IRQ}$ y/o $\overline{XIRQ}$; y 2) en el registro de banderas están habilitadas las interrupciones. Recuerde que las banderas X e I habilitan las interrupciones provenientes de las líneas $\overline{XIRQ}$ e $\overline{IRQ}$, respectivamente.

En este mismo estado el secuenciador ejecuta la instrucción de salto condicional con interrupciones. Si el valor de INT es uno entonces se realizará el salto hacia alguno de los estados de atención a la interrupción, cuya dirección será proporcionada por la Unidad de Control de Interrupciones. Si la línea $\overline{IRQ}$ fue quien generó la interrupción entonces se salta al estado "INTI"; por el contrario, si fue $\overline{XIRQ}$ quien generó la interrupción entonces se saltará al estado "INTX".

Cuando se detecta una interrupción es necesario guardar el contenido del registro PC para poderlo restaurar después de atender a la interrupción. Este valor de PC es guardado en la pila, que es una zona de memoria que utiliza el microprocesador para almacenar datos temporales. El valor de PC almacenado en la pila corresponde a la dirección de la siguiente instrucción a ejecutar, de manera que una vez atendida la interrupción, se podrá continuar con la ejecución del programa a partir de dicha instrucción.

Una vez guardada la copia de PC, el PC se carga con el valor de la dirección de inicio de un manejador o driver. Los manejadores o drivers son segmentos de código que atienden las peticiones de los dispositivos que generaron la interrupción. Las direcciones para estos drivers se obtienen de los registros de interrupciones DIRI y DIRX, según la línea que generó la interrupción.

Por ejemplo, suponga que la línea $\overline{XIRQ}$ fue quien generó la interrupción, por lo tanto, primero se guarda el contenido de PC en la pila y enseguida se carga el PC con la dirección de inicio del manejador para la interrupción $\overline{XIRQ}$; esta dirección se obtiene del registro de interrupción DIRX. Los pasos anteriores se ejecutan del estado "INTX" al estado "INTX4". Ahora suponga que la línea $\overline{IRQ}$ fue quien generó la interrupción, por lo tanto, se guarda el contenido de PC en la pila y se carga el PC con la dirección de inicio del manejador para la interrupción $\overline{IRQ}$, la cual se obtiene del registro de interrupción DIRI. Estos pasos se ejecutan del estado "INTI" al estado "INTI4".

Es importante mencionar que en los estados "INTI" e "INTX" se activan las señales SET_IRQ y SET_XIRQ, respectivamente. Estas señales colocan en estado de set a los flip-flops de la unidad de control de interrupciones, lo que permite detectar otras interrupciones para su posterior atención. En estos mismos estados también es activada la señal DINT, la cual deshabilita la generación de la señal INT. De esta manera, es posible detectar nuevas peticiones de interrupción, las cuales serán atendidas hasta que la petición en curso sea terminada[7].

Los siguientes pasos en la carta ASM consisten en guardar el estado de los demás registros de la arquitectura, es decir, son guardados en la pila el contenido del registro índice Y, del registro índice

[7] Consulte la Figura 5.12 para mayor información sobre la estructura interna de la Unidad de Control de Interrupciones.

X, del acumulador A, del acumulador B y del registro de banderas. Este procedimiento se realiza del estado "INTX5" al estado "INTX17".

Finalmente, se comienza un nuevo ciclo fetch utilizando la dirección de inicio del driver, es decir, se trae la primera instrucción que atiende al dispositivo que generó la interrupción.

6.3.12 REGRESO DE INTERRUPCIÓN

Instrucción: RTI

Operación:

AP ⇐ (AP) + 1, ⇑ (CCR)
AP ⇐ (AP) + 1, ⇑ (ACCB)
AP ⇐ (AP) + 1, ⇑ (ACCA)
AP ⇐ (AP) + 1, ⇑ (IX Alta)
AP ⇐ (AP) + 1, ⇑ (IX Baja)
AP ⇐ (AP) + 1, ⇑ (IY Alta)
AP ⇐ (AP) + 1, ⇑ (IY Baja)
AP ⇐ (AP) + 1, ⇑ (PC Alta)
AP ⇐ (AP) + 1, ⇑ (PC Baja)

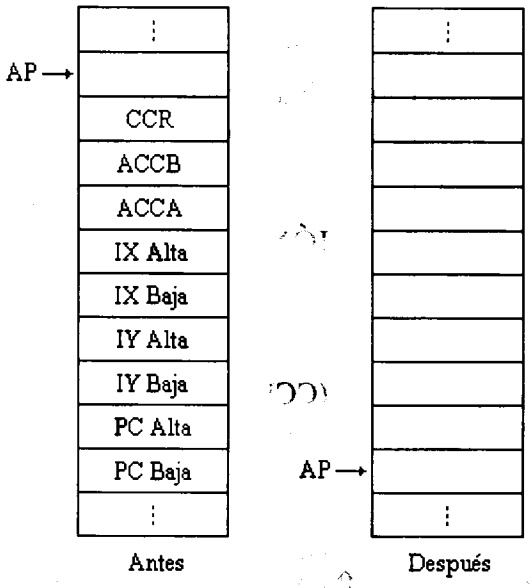
Código: 3B

Descripción: Regreso de Interrupción. Es restaurado el contenido del registro de estados, del acumulador B, del acumulador A, del registro índice X, del registro índice Y, y del contador de programa.

Banderas: Los bits de estado son modificados de acuerdo al valor almacenado en la pila, a excepción de la bandera X, la cual no puede cambiar de cero a uno. Sólo está permitido que cambie de uno a cero, o que mantenga su valor.

S	X	H	I	N	Z	V	C
↕	↓	↕	↕	↕	↕	↕	↕

El contenido de la pila después de ejecutar está instrucción queda de la siguiente manera.



En la figura 6.30 se presenta la carta ASM que ejecuta esta instrucción.

Como es sabido, en los primeros dos estados se obtiene el código de operación de la instrucción, y en el tercero es decodificado dicho código.

En el estado "RTI" se incrementa el apuntador de pila (AP), de manera que apunte a la dirección en memoria en donde se encuentra el contenido del registro de estados. En el siguiente estado, "RTI1", la dirección guardada en AP se carga en el registro de direcciones. En el estado "RTI2", se lee de memoria el dato a restaurar y se guarda en el registro de banderas. En este mismo estado también se incrementa el apuntador de pila para continuar restaurando el resto de los registros.

Del estado "RTI3" al estado "RTI14" es restaurado el contenido del acumulador B, del acumulador A, del registro índice X y del registro índice Y; y del estado "RTI15" al estado "RTI18" se restaura el contenido de PC para continuar con la ejecución del programa. Recuerde que antes de atender a la interrupción se guardó el valor de PC con la dirección de la siguiente instrucción a ejecutar.

Cuando se trataron las interrupciones[8] se mencionó que durante la atención a una interrupción es desactivada la generación de la línea INT, de esta manera, sólo se registran las nuevas peticiones de interrupción, las cuales serán atendidas hasta que la interrupción actual ha concluido; es decir, hasta que se ejecute una instrucción RTI. Por lo tanto, en el estado "RTI17" también se activa la señal HINT, para que nuevamente se pueda generar la línea INT y así atender a alguna interrupción que se haya solicitado mientras se atendía la interrupción actual.

[8] Para mayor información consulte la figura 6.29 y el apartado de atención a interrupciones.

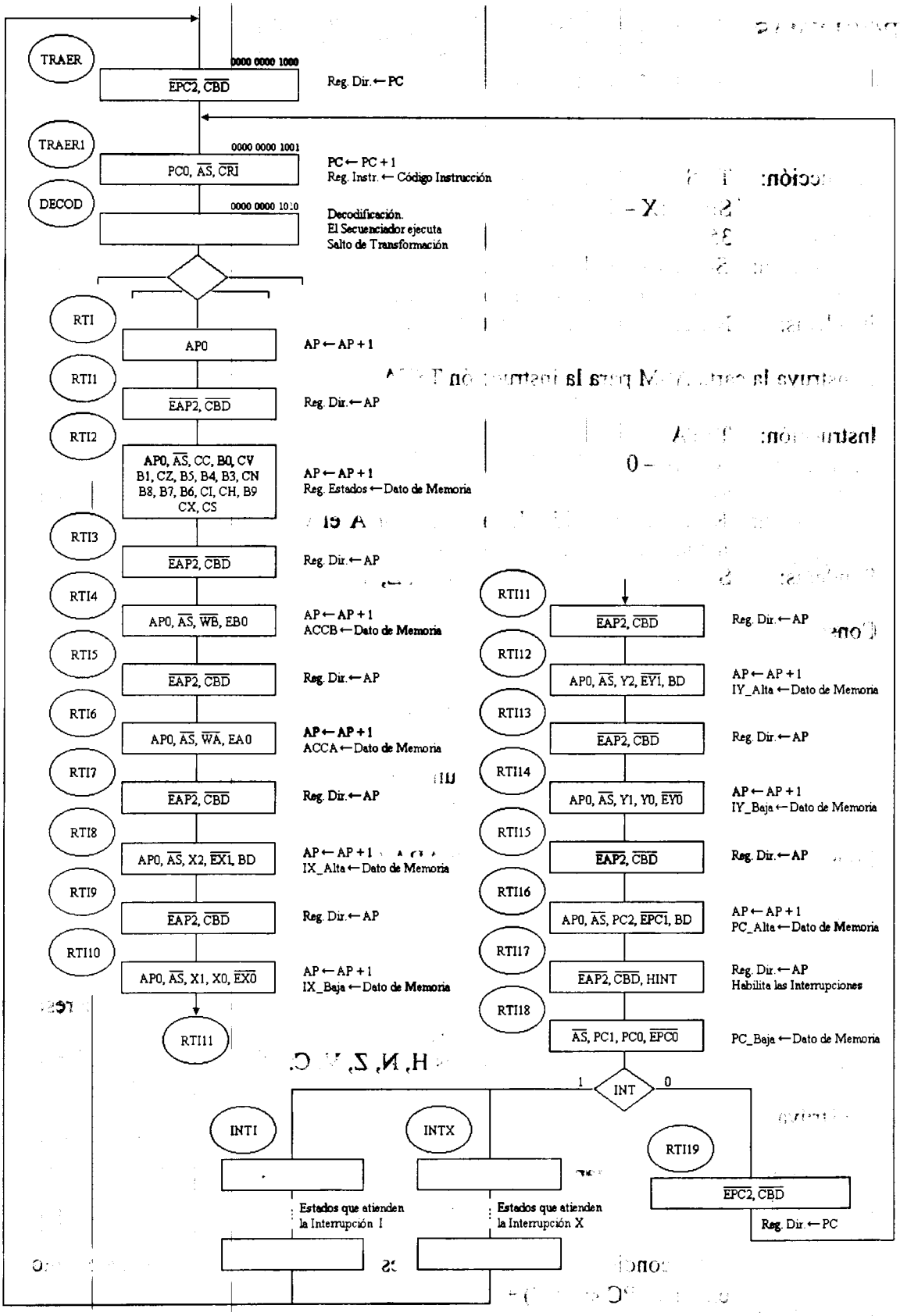


Figura 6.30. Carta ASM para la instrucción **RTI** (acceso inherente).

PROBLEMAS

1. Construya la carta ASM para la instrucción TXS (acceso inherente).

   Instrucción: TXS
   Operación: SP ⇐ IX – 1
   Código: 35
   Descripción: Se carga en el registro apuntador de pila el contenido del registro índice IX menos uno. El contenido de IX no se modifica.
   Banderas: Ninguna bandera es afectada.

2. Construya la carta ASM para la instrucción TSTA (acceso inherente).

   Instrucción: TSTA
   Operación: ACCA – 0
   Código: 4D
   Descripción: Resta al contenido del acumulador A el valor de cero. Esta instrucción sólo actualizada banderas en el registro de estados.
   Banderas: Son actualizadas las banderas N, Z, V=0 y C=0.

3. Construya la carta ASM para la instrucción LDAA (acceso directo).

   Instrucción: LDAA Dirección_8Bits
   Operación: ACCA ⇐ (Memoria)
   Código: 96
   Descripción: Carga en el acumulador A, un dato inmediato de 8 bits contenido en memoria.
   Banderas: Son actualizadas las banderas N, Z, V=0.

4. Construya la carta ASM para la instrucción ABA (acceso inherente).

   Instrucción: ABA
   Operación: ACCA ⇐ ACCA + ACCB
   Código: 1B
   Descripción: Suma los contenidos de los registros acumuladores A y B. El resultado es guardado en el acumulador A.
   Banderas: Son actualizadas las banderas H, N, Z, V, C.

5. Construya la carta ASM para la instrucción BNE (acceso relativo).

   Instrucción: BNE Desplazamiento
   Operación: Si Z=0, PC ⇐ (PC) + 2 + Desplazamiento
   Código: 26
   Descripción: Salto condicional. Si Z=0 entonces PC ⇐ (PC) + 2 + Desplazamiento, si Z=1 entonces PC ⇐ (PC) + 2.
   Banderas: Ninguna bandera es afectada.

6. Construya la carta ASM para la instrucción PSHX (acceso inherente).

Instrucción: PSHX
Operación: $\Downarrow IX_{BAJA}$, $SP \Leftarrow SP - 1$
$\Downarrow IX_{ALTA}$, $SP \Leftarrow SP - 1$
Código: 3C
Descripción: El contenido del registro índice IX es guardado en la pila. El apuntador de pila (SP) se decrementa cada vez que guardamos un byte en la pila.
Banderas: Ninguna bandera es afectada.

7. Construya la carta ASM para la instrucción ABX (acceso inherente).

Instrucción: ABX
Operación: $IX \Leftarrow IX + ACCB$
Código: 3A
Descripción: Suma el contenido del acumulador B (un dato de 8 bits sin signo) al contenido del registro índice IX. El resultado es guardado en el registro IX.
Banderas: Ninguna bandera es afectada.

8. Construya la carta ASM para la instrucción COM (acceso extendido).

Instrucción: COM Dirección_16Bits
Operación: Memoria $\Leftarrow \overline{\text{Memoria}}$ = 0xFF – Memoria
Código: 73
Descripción: Reemplaza el dato contenido en memoria por su complemento a uno.
Banderas: Son actualizadas las banderas N, Z, V=0 y C=1. Si desea calcular el complemento a uno de un número sin afectar la bandera de acarreo, entonces, ejecute la operación OR-exclusiva entre el número deseado y el valor 0xFF.

9. Construya la carta ASM para la instrucción BCLR (acceso directo).

Instrucción: BCLR Dirección_8Bits Máscara
Operación: Memoria $\Leftarrow$ (Memoria) • Máscara
Código: 15
Descripción: Primero se lee un dato de la localidad de memoria Dirección_8Bits. Los bits de ese dato se limpian según el valor de la máscara de 8 bits, y el resultado obtenido es guardado nuevamente en Dirección_8Bits.
Banderas: Son actualizadas las banderas N, Z y V=0.

CAPÍTULO VII

SEGMENTACIÓN ENCAUZADA (PIPELINE)

7.1 INTRODUCCIÓN

El "pipeline" ó segmentación encauzada, como se conoce en español, es una técnica utilizada en el diseño e implantación de microprocesadores en la cual múltiples instrucciones pueden ejecutarse simultáneamente. La técnica de la segmentación encauzada no reduce el tiempo que tarda una instrucción en ejecutarse, sólo incrementa el número de instrucciones que se ejecutan simultáneamente; es decir, mejora el rendimiento incrementando la productividad de las instrucciones en lugar del tiempo de ejecución de las instrucciones individuales.

Es fácil entender el concepto de segmentación encauzada si pensamos en una línea de ensamble, en donde cada etapa de la línea completa una parte del trabajo total. Al igual que en la línea de ensamble, el trabajo que se realiza para cada instrucción se descompone en partes más pequeñas; cada una de estas partes, denominadas etapas ó segmentos, necesitan una fracción del tiempo total para completar la instrucción.

Entre dos etapas de la línea de ensamble se coloca un registro de acoplo, también denominado registro de segmentación, que se encarga de guardar los datos y las señales de control necesarias para etapas posteriores. Por ejemplo, el registro de acoplo situado entre las etapas 2 y 3 de la figura 7.1 guarda los datos y las señales de control generadas en la etapa 2 para su uso posterior en la etapa 3. Gracias a los registros de acoplo es posible manejar múltiples instrucciones al mismo tiempo.

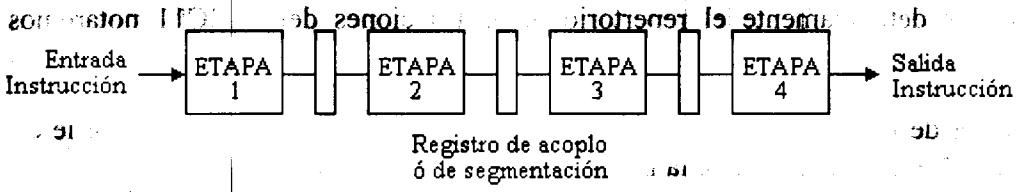


Figura 7.1. Etapas o segmentos.

De la figura 7.2 se observa que la ejecución de una instrucción consta de cuatro etapas o módulos: 1) traer la instrucción, 2) decodificarla, 3) traer operandos, y 4) ejecutarla. Si cada uno de estos módulos es independiente de los demás, entonces, en el tiempo '*n*' se empezaría a traer la instrucción I_n; al mismo tiempo se estaría decodificando la instrucción I_{n-1}, trayendo los operandos de la instrucción I_{n-2} y terminando de ejecutar la instrucción I_{n-3}. Esto significa que una vez que el cauce está lleno, idealmente, en cada ciclo de reloj se estaría ejecutando una instrucción.

Un aspecto de suma importancia en la segmentación encauzada es la duración del ciclo de reloj. Este reloj es el encargado de sincronizar todas las etapas de la segmentación, por lo tanto, debe ser lo suficientemente grande para acomodar las operaciones de la etapa más lenta.

Idealmente, la mejora de velocidad debido a la segmentación encauzada es igual al número de etapas, esto es, una arquitectura segmentada de cuatro etapas es cuatro veces más rápida que una arquitectura sin esta tecnología. Sin embargo, esta velocidad no se alcanza en la realidad ya que en algunas ocasiones el cauce tiene que ser llenado de nuevo, como por ejemplo, cuando ocurre un salto; en este caso, la secuencia de instrucciones que estaba dentro del cauce tiene que ser eliminada

para comenzar a ejecutar nuevas instrucciones a partir de la dirección de salto. Adicionalmente, debido a que las etapas están equilibradas imperfectamente, el tiempo por instrucción en una arquitectura segmentada no tiene el valor mínimo posible y la mejora en velocidad será menor que el número de etapas.

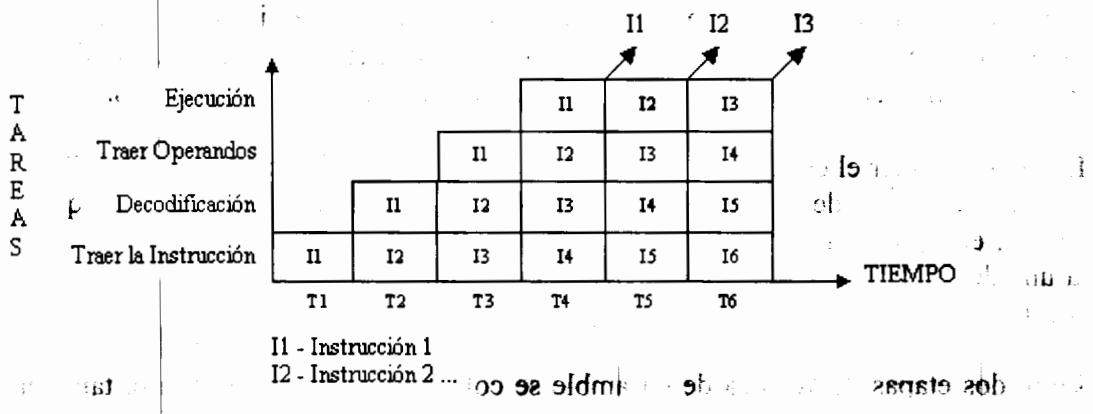


Figura 7.2. Diagrama de Tiempos / Tareas.

7.2 LA ARQUITECTURA SEGMENTADA DEL 68HC11

Si revisamos detenidamente el repertorio de instrucciones del 68HC11 notaremos que cada instrucción ejecuta una serie de pasos. En general, los pasos a seguir son los siguientes.

1. Traer de la memoria la instrucción que se desea ejecutar (a este paso se le conoce como ciclo fetch ó búsqueda de la instrucción)
2. Decodificación de la instrucción
3. Si la instrucción requiere leer un operando de la memoria, entonces se calcula la dirección efectiva de ese operando y se lee el dato de la memoria
4. Si lo requiere la instrucción, se leen de los registros internos del microprocesador los operandos necesarios
5. Ejecución, es decir, se realiza una operación en la unidad de procesos aritméticos con los operandos leídos anteriormente
6. Se guardan los resultados de la operación y se actualiza el registro de banderas

Observe que estos pasos son similares a los ejecutados en las cartas ASM para las instrucciones vistas en el capítulo VI. La arquitectura segmentada del 68HC11 también ejecutará estos mismos pasos, pero agrupados en las siguientes cuatro etapas.

1. Etapa IF (traer la instrucción / instruction fetch). La instrucción a ejecutar es leída de la memoria de instrucciones
2. Etapa ID (decodificación / instruction decode). Se decodifica la instrucción y se traen los operandos necesarios por la instrucción (tanto de memoria como de registros internos)
3. Etapa EX (ejecución / execution). Se procesan los operandos en la UPA (unidad de procesos aritméticos)
4. Etapa WB (post-escritura / write back). Se guardan resultados

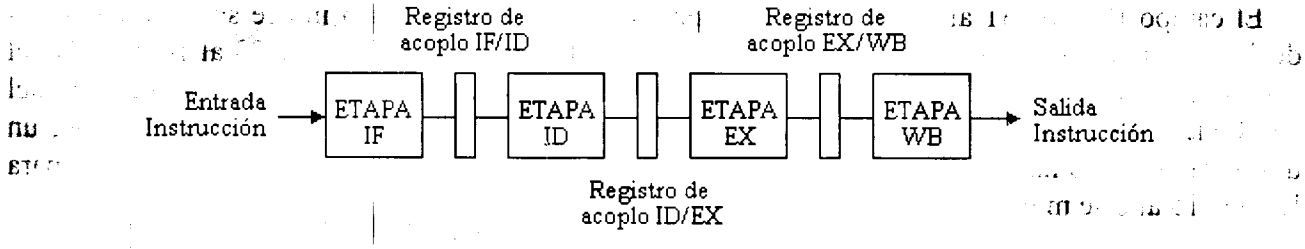


Figura 7.3. Etapas propuestas para la arquitectura segmentada del 68HC11.

A continuación se describen las tareas que se ejecutan en cada etapa y se presenta un diagrama de bloques con el hardware necesario para su implantación. Hay que tener presente que esta arquitectura no soporta todo el conjunto de instrucciones del 68HC11; algunas instrucciones no son soportadas y otras requieren redefinir el formato de la instrucción, ó agregar hardware adicional, para su posible implantación.

7.2.1 ETAPA 1 - LECTURA DE LA INSTRUCCIÓN

El primer paso que realiza todo microprocesador es leer de memoria la siguiente instrucción a ejecutar. Recordemos que en la arquitectura secuencial del 68HC11, descrita en el capítulo VI, el primer paso que se realiza para cada instrucción es su ciclo fetch, es decir, traer de memoria la instrucción a ejecutar. Enseguida, si la instrucción lo requería, también eran leídos de memoria los datos y/o las direcciones en memoria de los datos. Este proceso necesitaba acceder a memoria cierto número de veces, según la instrucción que se tratase.

Para una arquitectura segmentada, acceder tantas veces a la memoria complica el hardware y retrasa el comienzo de la siguiente instrucción a ejecutar. No olvide que la segmentación encauzada recomienda que el flujo de datos sea siempre hacia adelante, es decir, que se avance hacia etapas posteriores en el cauce; sin embargo, habrá etapas, como la de post-escritura, en la que se necesitará retroceder en el cauce.

Una manera de evitar los accesos a memoria repetidamente es leyendo en una sola pasada toda la información que la instrucción vaya a necesitar, esto es, leer el código de operación de la instrucción, leer los datos inmediatos y leer las direcciones de memoria. Para ello, el tamaño de la instrucción para el 68HC11 segmentado ha sido extendido a 32 bits, los cuales contendrán toda la información necesaria según el modo de direccionamiento del que se trate. Además, la memoria externa será separada en memoria de instrucciones o programa y en memoria de datos.

Como resultado del aumento en el tamaño de la instrucción, el formato de ésta se ha modificado de la siguiente manera.

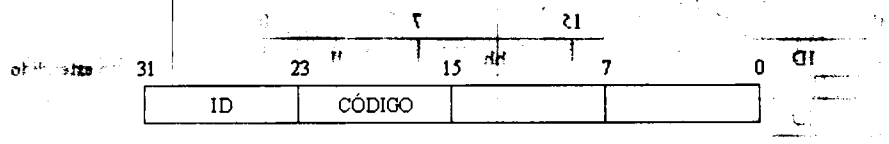


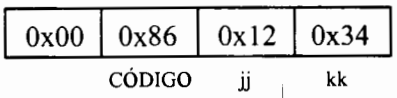
Figura 7.4. Formato general para las instrucciones del 68HC11 segmentado.

El campo ID, bits 31 al 24, sirven para especificar sobre qué registro índice se va a operar, es decir, se trata del registro índice IX ó del registro IY. El campo CÓDIGO, bits 23 al 16, guardan el código de operación de la instrucción, tal y como lo establece el conjunto de instrucciones del 68HC11. Y los bits 15 al 0 pueden almacenar el valor de un dato inmediato, una dirección, un desplazamiento, ó nada, según el modo de direccionamiento que se trate. Los formatos válidos para los bits 15 al 0 se muestran en la siguiente tabla.

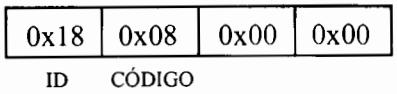
Bits 15-8	*Bits 7-0*	*Descripción*
hh	ll	Dirección de 16 bits
00	dd	Dirección de 8 bits
jj	kk	Dato inmediato de 16 bits
00	ii	Dato inmediato de 8 bits
00	ff	Desplazamiento de 8 bits sin signo
nn	mm	Máscara de 16 bits
uu	vv	Desplazamiento de 16 bits con signo
00	rr	Desplazamiento de 8 bits con signo

Tabla 7.1. Contenido para los 16 bits menos significativos del formato de instrucciones.

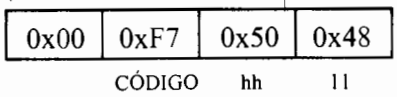
A continuación se analizan algunos ejemplos para dejar en claro el formato de instrucciones.



El campo código, 0x86, nos indica que la instrucción a ejecutar es *ldaa* con modo de direccionamiento inmediato. El dato inmediato, 0x1234, se encuentra guardado en los 16 bits menos significativos del formato de la instrucción.



El campo código, 0x08, junto con el campo ID, 0x18, nos indican que la instrucción a ejecutar es *iny*. Como *iny* utiliza el modo de direccionamiento inherente, el resto de los campos no son relevantes.



El campo código, 0xF7, nos indica que la instrucción a ejecutar es *stab* con modo de direccionamiento extendido. La dirección extendida, 0x5048, se encuentra guardada en los 16 bits menos significativos del formato de la instrucción.

La siguiente figura muestra los formatos de instrucción genéricos para los modos de direccionamiento directo y extendido.

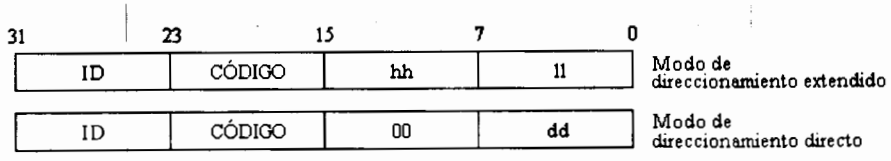


Figura 7.5. Formatos para los modos de direccionamiento directo y extendido.

Note que en ambos casos se están ocupando los 32 bits del formato de la instrucción, por lo tanto, resulta poco efectivo mantener ambos formatos, ya que con el modo extendido es posible manejar el modo directo. Algo similar ocurre con el formato de los desplazamientos con signo, y con el formato de los datos inmediatos de 8 y 16 bits. Aún con el conocimiento de que hay formatos repetidos ó innecesarios, éstos serán mantenidos con el fin de adaptar la nueva arquitectura al conjunto de instrucciones que ya habíamos manejado en el capítulo VI.

Finalmente, el hardware para la etapa de la lectura de la instrucción queda de la siguiente manera.

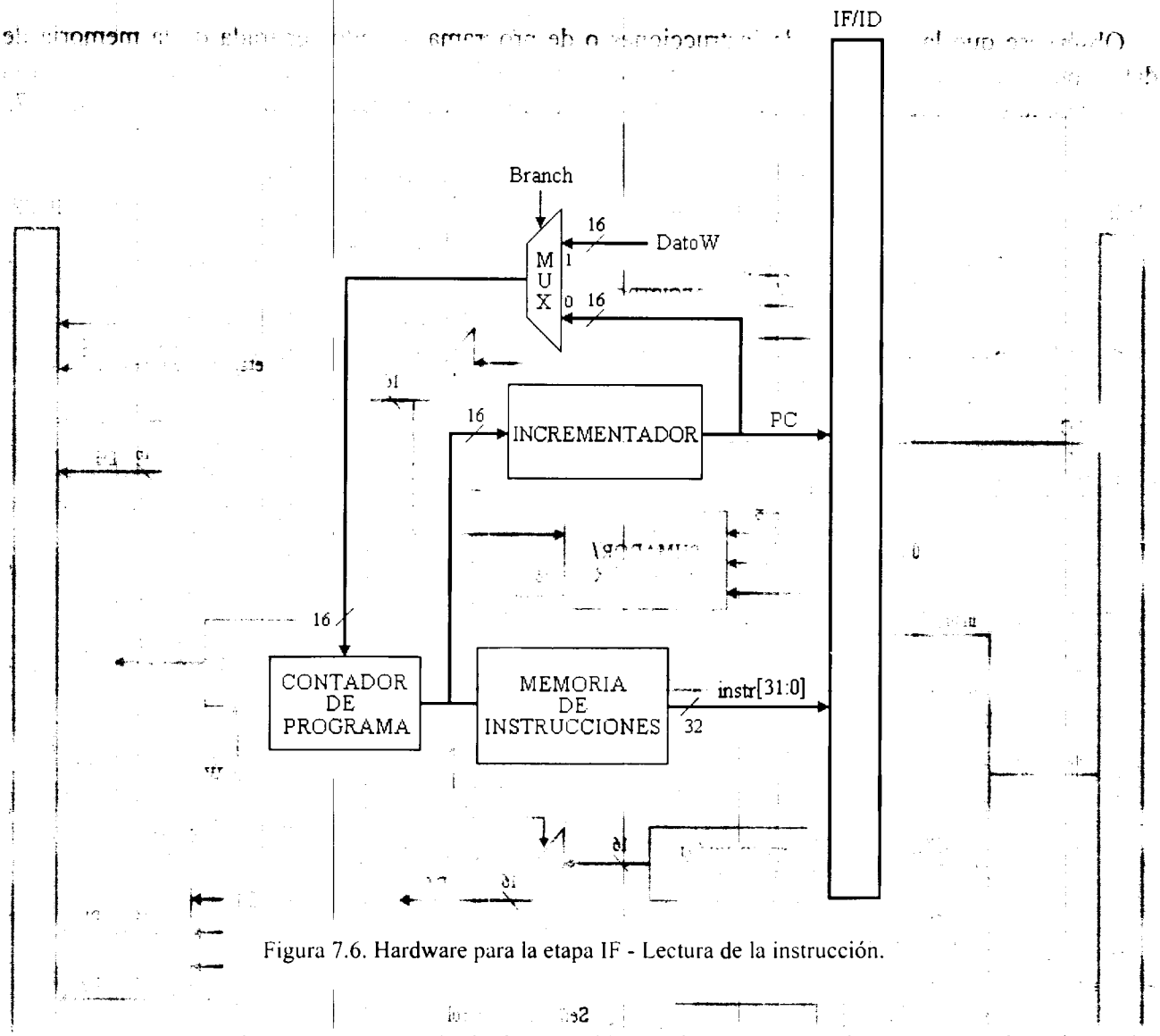


Figura 7.6. Hardware para la etapa IF - Lectura de la instrucción.

En esta etapa se lee de la memoria la instrucción a ejecutar y se almacena en el registro de segmentación IF/ID. La dirección de esta instrucción está dada por el contador de programa (PC); dicha dirección se incrementa y se vuelve a cargar en el PC para ser leída en el siguiente ciclo de reloj. La dirección incrementada también se guarda en el registro de segmentación IF/ID, pues es posible que la necesite otra instrucción posteriormente, por ejemplo, la instrucción *bra*.

7.2.2 ETAPA 2 - DECODIFICACIÓN DE LA INSTRUCCIÓN / CÁLCULO DE LA DIRECCIÓN EFECTIVA / LECTURA DE OPERANDOS

Durante esta etapa, la instrucción leída en la etapa anterior es decodificada. Una vez que se conoce la instrucción a ejecutar, el módulo de control genera las señales de control que gobernarán el hardware de esta etapa y de etapas posteriores. Por ejemplo, las señales de control que se emplean en la segunda etapa se encargan de informarle a los distintos componentes de esta etapa qué operaciones realizar. Estas operaciones consisten en lectura de operandos de registros, cálculo de direcciones, lectura de operandos de la memoria de datos, principalmente.

Obsérvese que la memoria de instrucciones o de programa ha sido separada de la memoria de datos, pues de estar unidas, la continúa lectura de operandos y escritura de resultados demoraría la lectura de la siguiente instrucción a ejecutar. El hardware para esta etapa se muestra en la figura 7.7.

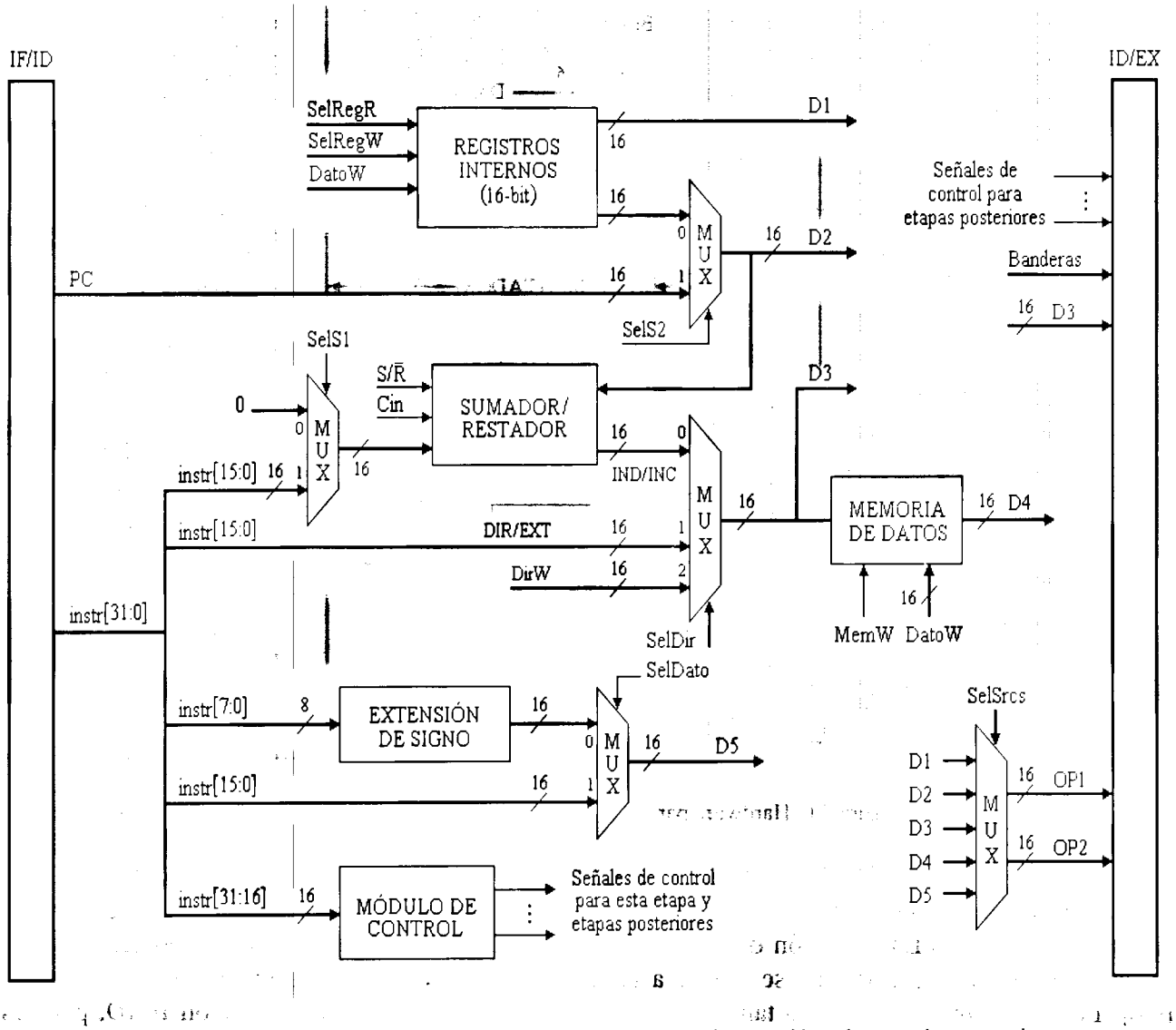


Figura 7.7. Hardware para la etapa ID - Decodificación de la instrucción y lectura de operandos.

El módulo de control de la figura 7.7 es el encargado de generar todas las señales de control. Algunas de estas señales serán utilizadas durante esta etapa y otras serán guardadas en los registros de segmentación para su utilización en etapas posteriores. La generación de las señales de control es posible gracias a la información que le brindan al módulo las líneas *instr*[31:16], pues estas líneas transportan el código de operación de la instrucción con la información suficiente para saber de qué instrucción se trata y su modo de direccionamiento.

Por otra parte, en el módulo de registros internos se encuentran los acumuladores A y B, y los registros internos IX, IY, SP y AUX. Todos estos registros son de 16 bits de tamaño. El siguiente diagrama muestra la disposición de todos ellos dentro del módulo.

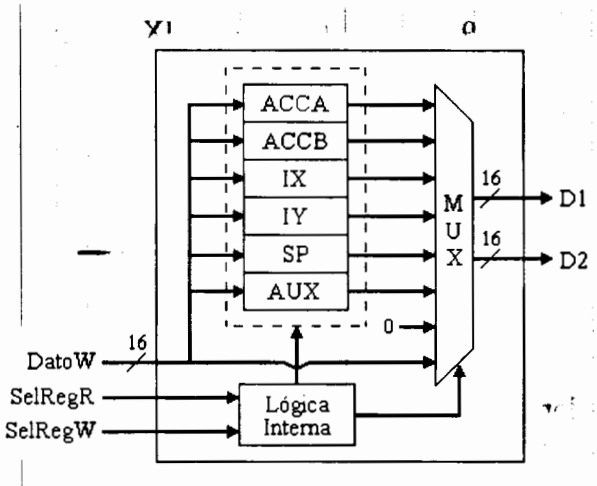


Figura 7.8. Módulo de registros internos.

La línea de control *SelRegR* le indica al módulo qué registros deseamos leer; la línea *SelRegW* le indica en qué registro se desea escribir el valor de *DatoW*; y las salidas *D1* y *D2* transportan los valores leídos de dichos registros. Posteriormente se describirá el funcionamiento de la lógica interna de este módulo, la cual nos permitirá adelantar datos, y evitar lecturas y escrituras al mismo tiempo en el mismo registro. A continuación se muestran las tablas para la escritura y lectura de los registros internos.

SelRegW	*Registro que se escribe*
0	Ninguno
1	ACCA
2	IX
3	IY
4	ACCB
5	AUX
6	SP

Tabla 7.2. Selección de los registros para escritura.

SelRegR	*Registro seleccionado para lectura*	
	D1	D2
0	0	0
1	ACCA	ACCB
2	ACCB	IX
3	ACCB	IY
4	ACCA	0
5	ACCB	0
6	ACCA	IX
7	ACCA	IY
8	AUX	0
9	0	IX
A	0	IY
B	0	SP
C	ACCA	SP
D	ACCB	SP
E	IX	SP
F	IY	SP

Tabla 7.3. Selección de los registros para lectura.

El módulo sumador/restador de esta etapa se utiliza para calcular incrementos, decrementos, y sobre todo, para el cálculo de la dirección efectiva en instrucciones con modo de direccionamiento indexado. Más adelante, se verá un ejemplo de este tipo de instrucción.

Las señales de control para el módulo sumador/restador son las siguientes: $S/\overline{R}$, permite seleccionar la operación a ejecutar, una suma (si vale uno) y una resta (si vale cero); y *Cin*, que es el acarreo de entrada al sumador. Note que este módulo obtiene sus operandos de dos multiplexores. El primer multiplexor utiliza la señal de control *SelS1*; si *SelS1* vale uno, entonces se elige el bus *instr*[15:0], pero si vale cero, entonces se elige el valor de cero. El segundo multiplexor utiliza la señal de control *SelS2*; si *SelS2* vale uno, entonces se elige el valor de *PC*, y si vale cero, se elige el contenido de alguno de los registros internos (*D2*).

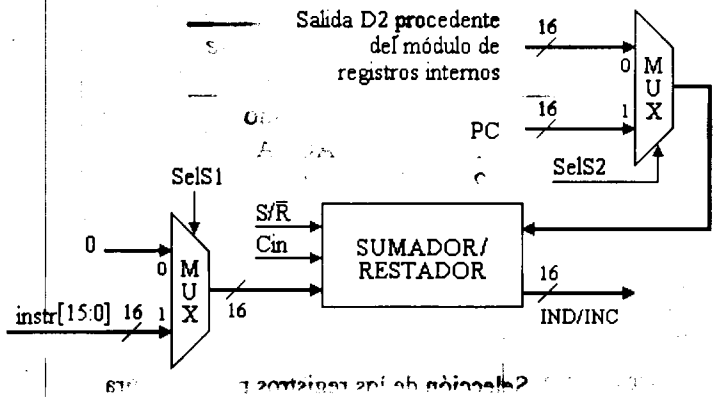


Figura 7.9. Módulo sumador / restador y sus señales de control.

En esta etapa también se encuentra la memoria de datos la cual es totalmente independiente de la memoria de instrucciones, de ella leeremos operandos, o bien, almacenaremos los resultados de nuestras operaciones, según sea el caso.

La única señal de control que se utiliza en la memoria es *MemW*, la cual indica si se realiza una operación de escritura o de lectura en ella. Si *MemW* vale uno la operación a efectuar será escritura, y si vale cero la operación será lectura. La dirección del dato a leer o escribir en memoria proviene de un multiplexor, esta dirección se selecciona con la línea de control *SelDir*. Cuando *SelDir*=0 la dirección que se elige proviene del módulo sumador/restador; si *SelDir*=1 proviene del bus *instr*[15:0]; y si *SelDir*=2 la dirección que se toma será *DirW*.

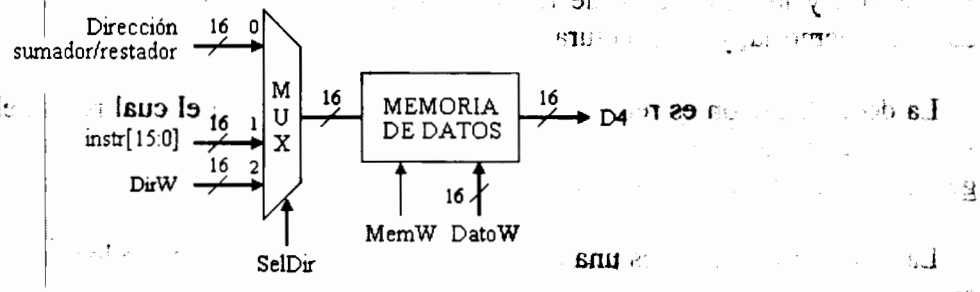


Figura 7.10. Memoria de datos y sus señales de control.

Por otra parte, las instrucciones de salto o de acceso relativo utilizan un desplazamiento de 8 bits con signo para calcular una dirección de salto. El módulo encargado de calcular esta dirección de salto es la UPA (unidad de procesos aritméticos) ubicada en la siguiente etapa. Sin embargo, como se verá más adelante, la UPA sólo ejecuta operaciones de 16 bits, por lo que el desplazamiento de 8 bits con signo necesitará ser extendido a 16 bits. El módulo encargado de esta tarea es el módulo de extensión de signo, quien tomará los 8 bits del desplazamiento con signo y los extenderá a 16 bits, repitiendo en los 8 bits más significativos el bit de signo del desplazamiento original, es decir,

Desplazamiento original ⇒ Desplazamiento extendido

$$\underline{0}1100011 \Rightarrow 0000000\underline{0}01100011$$
$$\underline{1}0111100 \Rightarrow 1111111\underline{1}10111100$$

Finalmente, existe un multiplexor que selecciona los operandos para la siguiente etapa. Este multiplexor elige estos operandos de los datos presentes en los buses *D1*, *D2*, *D3*, *D4* y *D5*, según la instrucción que se trate. La selección de estos operandos se muestra en la siguiente tabla.

SelSrcs	*Operandos seleccionados*	
	OP1	OP2
0	0	0
1	D1	D2
2	D1	D4
3	D1	D5

4	D4	D3
5	D2	D5
6	D2	D4

Tabla 7.4. Selección de los operandos para la etapa de ejecución.

Los operandos que se seleccionaron utilizando el multiplexor anterior, la dirección efectiva y las señales de control para las etapas posteriores son guardados temporalmente en el registro de segmentación ID/EX.

En resumen, esta segunda etapa se encarga de realizar tres tareas: 1) la decodificación de la instrucción y la generación de las señales de control; 2) el cálculo de direcciones efectivas para datos en memoria; y 3) la lectura de operandos.

La decodificación es realizada por el módulo de control el cual revisa el código de operación de la instrucción (los dos bytes más significativos del formato de la instrucción), y con base en él, genera las señales de control necesarias para la etapa actual y para las etapas posteriores.

La dirección efectiva es una dirección en memoria de donde se lee un dato, o bien, una dirección en memoria en donde se guarda un dato. Para algunos modos de direccionamiento esta dirección no es inmediata. Por ejemplo, el modo de direccionamiento indexado, calcula la dirección efectiva sumando al contenido de un registro base un desplazamiento; en cambio, los modos de direccionamiento directo y extendido proporcionan la dirección efectiva de forma inmediata.

La lectura de operandos consiste en obtener los datos que se operarán en la siguiente etapa. Los operandos pueden provenir de los registros internos, de la memoria de datos, o bien, pueden estar contenidos en el mismo formato de la instrucción. Para obtener el contenido de algunos de los registros basta con indicar al módulo de registros internos qué registros se desean leer. Para obtener el operando de la memoria de datos es necesario contar con la dirección efectiva donde se encuentra ese dato. Y si el operando está contenido en el formato de la instrucción, como es el caso del direccionamiento inmediato, la línea de control *SelDato*, de uno de los multiplexores, permitirá su selección.

7.2.3 ETAPA 3 - EJECUCIÓN / CÁLCULO DE BANDERAS Y SALTOS

Esta etapa ejecuta tres tareas: 1) opera los operandos obtenidos en la etapa de decodificación (etapa 2); 2) actualiza el registro de estados o banderas; y 3) calcula la condición de salto.

El hardware para esta etapa se muestra en la figura 7.11.

La unidad de procesos aritméticos (UPA) se encarga de obtener el resultado entre los operandos según la operación establecida en *SelOp*. *SelOp* es una señal de control generada en la etapa anterior, pero que es empleada hasta esta etapa. En la UPA se realizan las operaciones lógicas, las operaciones aritméticas y los corrimientos, tal y como se muestra en la tabla 7.5.

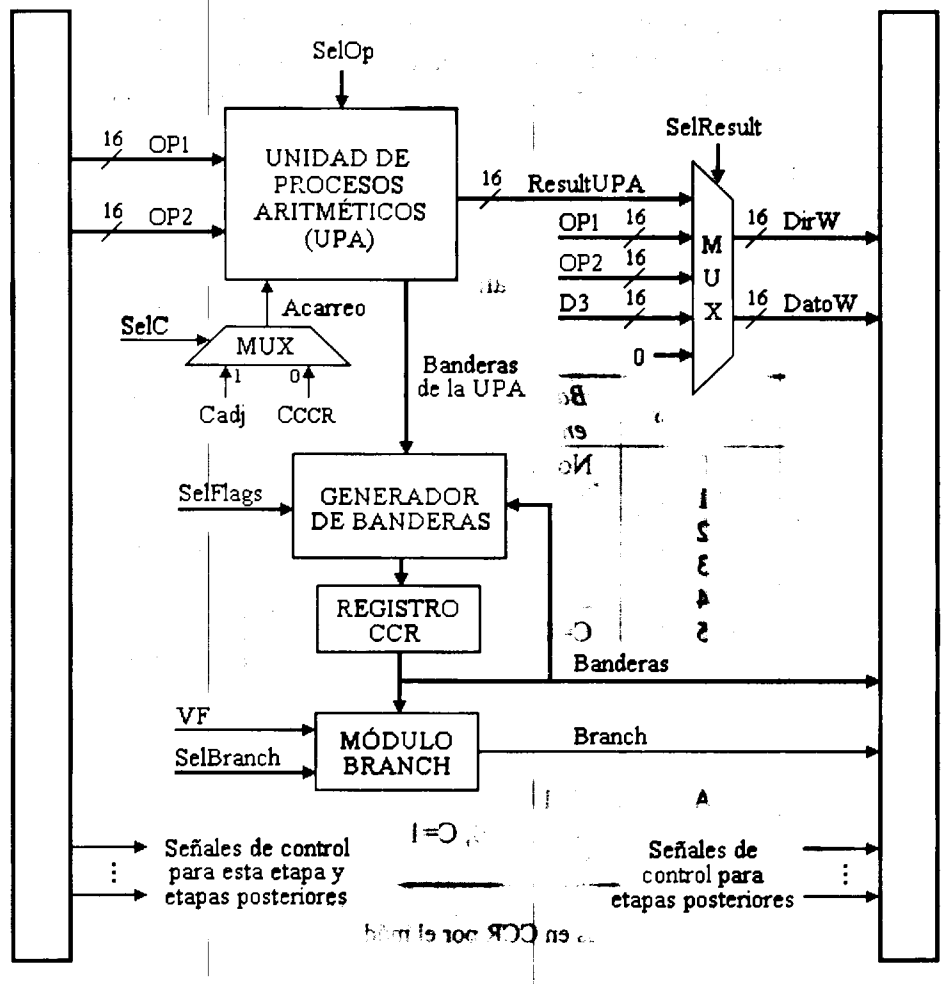


Figura 7.11. Hardware para la etapa EX - Ejecución.

SelOp	*Operación ejecutada*
0	Ninguna
1	OP1 + OP2 + Acarreo
2	OP1 – OP2 – $\overline{\text{Acarreo}}$
3	OP1 and OP2
4	OP1 or OP2
5	OP1 xor OP2
6	Corrimiento a la izquierda de OP1 con $B_0 = 0$
7	Corrimiento a la derecha de OP1 con $B_{15} = B_{15}$
8	OP2 – OP1 – $\overline{\text{Acarreo}}$
9	Corrimiento a la derecha de $\overline{\text{OP1}}$ con $\overline{B_{15}} = 0$
A	Rotación a la izquierda de OP1 con $B_0 = C_{CCR}$
B	Rotación a la derecha de OP1 con $B_{15} = C_{CCR}$

Tabla 7.5. Operaciones de la UPA.

El acarreo de entrada al módulo UPA es seleccionado mediante la señal de control *SelC*. Si *SelC* vale cero, el acarreo elegido proviene del registro de estados (C_{CCR}); si *SelC* vale uno, entonces el acarreo elegido es C_{adj}, el cual es generado por el módulo de control de la etapa 2 y es establecido a un cierto valor según la instrucción que se trate.

Las banderas que se modificaron tras la operación ejecutada en la UPA son guardadas en el registro de estados (CCR, Condition Code Register) por el módulo generador de banderas. La tabla 7.6 muestra la relación entre la señal de control *SelFlags* y las banderas que son actualizadas en el registro de estados por el módulo generador de banderas.

SelFlags	*Banderas que son actualizadas en el CCR*
0	No se modifica el CCR
1	N, Z, V=0
2	N, Z, V, C, H
3	N, Z, V, C
4	Z
5	C=0
6	I=0
7	V=0
8	C=1
9	I=1
A	V=1
B	N, Z, V=0, C=1
C	N, Z, V

Tabla 7.6. Banderas afectadas en CCR por el módulo generador de banderas.

En caso de ejecutar una instrucción de salto, la UPA calculará la dirección a donde probablemente se deba saltar, y el módulo Branch evaluará la condición de salto para determinar si en verdad se ejecuta el salto o no. Recuerde que las señales de control, *VF* y *SelBranch*, empleadas en el módulo Branch, fueron generadas en la etapa anterior.

El módulo Branch genera una señal de salida denominada **'Branch'**. Esta señal, generada a partir de la condición de salto y del valor de *VF*, nos permite saber si el salto se realiza o no. Para ello, la condición de salto debe ser evaluada; si el resultado de esta evaluación es igual al valor de *VF*, entonces la señal **'Branch'** toma el valor de uno, si no, **'Branch'** toma el valor de cero. La tabla 7.7 muestra la relación entre la señal de control *SelBranch* y la condición de salto que se evalúa.

SelBranch	*Condición a evaluar*
0	Se compara con cero
1	C
2	Z
3	$N \oplus V$
4	$Z + (N \oplus V)$

5	C + Z
6	N
7	V

Tabla 7.7. Condiciones de salto para el módulo Branch.

Por último, son guardados en el registro de segmentación EX/WB el resultado de la UPA, la dirección efectiva obtenida en la etapa 2, algunos valores de banderas, y las señales de control necesarias para la última etapa. A continuación se presenta la relación de la señal *SelResult* con las fuentes seleccionadas hacia el registro de segmentación EX/WB.

SelResult	*Fuentes seleccionadas*	
	DatoW	DirW
0	0	0
1	ResultUPA	D3
3	OP1	D3

Tabla 7.8. Fuentes seleccionadas por la señal de control SelResult.

7.2.4 ETAPA 4 – POST-ESCRITURA

En las arquitecturas segmentadas las instrucciones y los datos se desplazan generalmente de izquierda a derecha a través de las etapas, sin embargo, hay dos excepciones que se presentan en la etapa de post-escritura:

1. Cuando se guarda el resultado de la UPA en los registros o en memoria, haciendo que se retroceda a la etapa 2.
2. Cuando se selecciona el nuevo valor de PC en la etapa 1, valor que puede ser el PC incrementado, o bien, la dirección de salto calculada en la etapa 3.

En resumen, la etapa de post-escritura se encarga de actualizar los resultados obtenidos en etapas anteriores. Recuerde que las señales de control utilizadas en esta última etapa fueron generadas en la etapa 2 y se propagaron a través de los registros de segmentación hasta la etapa requerida.

La arquitectura completa del 68HC11 segmentado se muestra en la figura 7.12.

De la figura 7.12 notará que la señal de control *SelDir* se utiliza en dos etapas de la arquitectura, en la etapa 2 y en la etapa 4. Durante la etapa 2, la señal *SelDir* seleccionará la dirección de memoria de donde se leerá un dato. Esta dirección puede venir de dos buses, del bus IND/INC (si la instrucción utiliza acceso indexado), o del bus DIR/EXT (si la instrucción utiliza acceso directo o extendido). En cambio, en la etapa 4, la señal *SelDir* seleccionará la dirección en memoria en donde se guardará algún resultado, si es que la instrucción así lo especifica. Esta dirección provendrá exclusivamente del bus DirW.

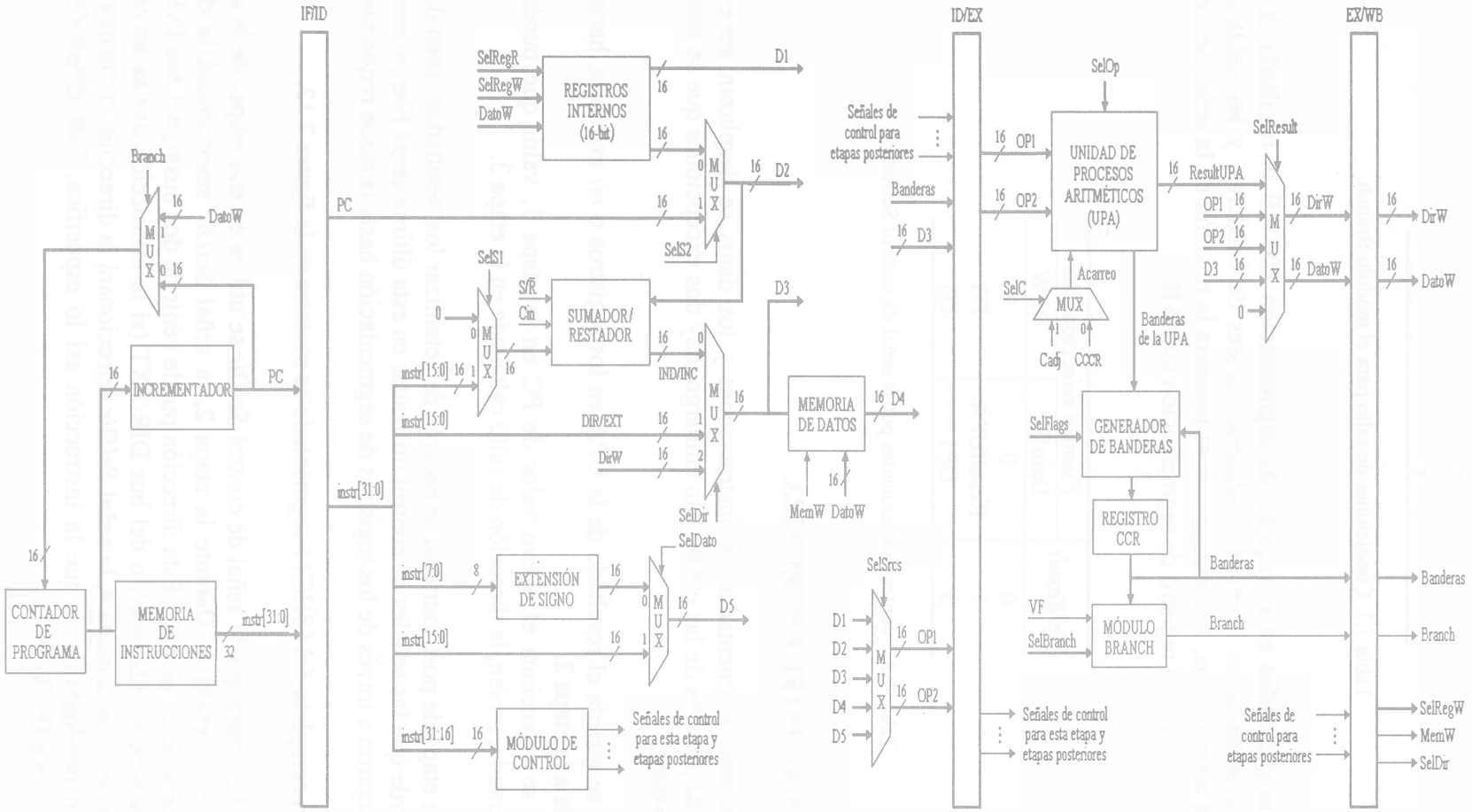


Figura 7.12. Arquitectura del 68HC11 utilizando la técnica de la segmentación encauzada.

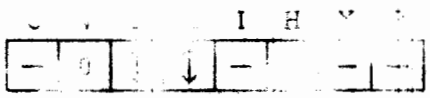
La unidad de control ubicada en la etapa 2 es la encargada de generar los valores de la señal *SelDir* de acuerdo a la instrucción que se ejecuta. De esta manera, serán generados dos valores para la señal *SelDir*, el primer valor se utilizará durante la etapa 2, y el segundo valor será guardado en el registro de segmentación ID/EX junto con las señales de control para las etapas siguientes, y será utilizado hasta la etapa 4.

7.2.5 REPRESENTACIÓN GRÁFICA DE LA SEGMENTACIÓN ENCAUZADA

La segmentación encauzada puede ser difícil de comprender, ya que múltiples instrucciones se ejecutan simultáneamente en un sólo camino de datos en cada ciclo de reloj. Para ayudar a comprenderlo hay dos estilos básicos de diagramas: los diagramas de múltiples ciclos de reloj y los diagramas de un sólo ciclo de reloj.

En los diagramas de múltiples ciclos de reloj el tiempo avanza de izquierda a derecha en el diagrama, y las instrucciones avanzan de arriba hacia abajo. Utilizamos estos diagramas para dar visiones generales de las situaciones en la segmentación.

Los diagramas de un sólo ciclo de reloj muestran el estado del camino de datos durante un sólo ciclo de reloj. Generalmente, todas las instrucciones de la segmentación encauzada se identifican por rótulos debajo de sus respectivas etapas. Estos diagramas se utilizan para mostrar detalladamente lo que ocurre en la segmentación durante cada ciclo de reloj.



7.3 CONJUNTO DE INSTRUCCIONES

En esta sección se analizará el comportamiento de algunas de las instrucciones del 68HC11 en cada una de las etapas de la segmentación. Para ilustrar con detalle el progreso de estas instrucciones utilizaremos los diagramas de un sólo ciclo de reloj.

7.3.1 INSTRUCCIÓN LDAA (Acceso Inmediato)

Instrucción: LDAA #Dato_16Bits
Operación: ACCA ⇐ (Memoria)
Código: 0086
Descripción: Carga en el registro ACCA un dato inmediato de 16 bits contenido en memoria.
Banderas: N=1 si el MSB[9] del resultado está encendido, N=0 en caso contrario.
Z=1 si el resultado en el registro es cero, Z=0 en caso contrario.
V se coloca a cero.

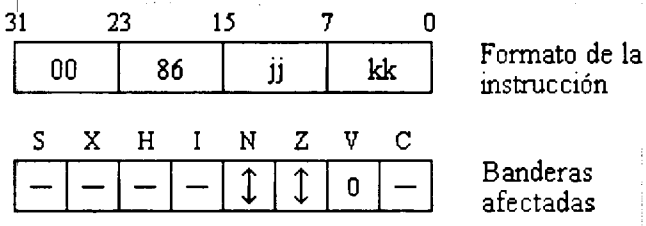


Figura 7.13. Formato de la instrucción LDAA y banderas que afecta.

El comportamiento de la instrucción *ldaa* es el siguiente.

Etapa 1. Lectura de la instrucción

En esta etapa se lee de la memoria la instrucción a ejecutar, es decir, la instrucción *ldaa*. La dirección de esta instrucción, dada por el contador de programa, se incrementa y se vuelve a cargar en el contador de programa. Este nuevo valor corresponde a la dirección en memoria de la próxima instrucción a ejecutar. La dirección incrementada y la instrucción se guardan en el registro de segmentación IF/ID para su uso posterior en la etapa 2 (véase la figura 7.14).

Etapa 2. Decodificación / Cálculo de la dirección efectiva / Lectura de operandos

El modo de direccionamiento para esta instrucción es inmediato, es decir, el mismo formato de la instrucción, guardado en el registro de segmentación IF/ID, contiene el dato a cargar en el registro

[9] MSB = Bit más significativo.

ACCA; por lo tanto, no es necesario calcular ninguna dirección efectiva. Sin embargo, observe que la carga del dato inmediato en el registro ACCA no puede hacerse en la etapa 2, ya que *DatoW* que es el bus de datos con los resultados obtenidos proviene de la etapa 4. Así que deberemos esperar hasta la etapa 4 para hacer la escritura del dato inmediato en el registro ACCA.

Las señales de control para esta etapa y las etapas posteriores se muestran en la siguiente tabla.

Etapa en la que se utiliza la señal	*Señales de control*	*Valor de la señal*
Etapa 2	SelRegR	0
	SelS1	0
	S/$\overline{R}$	1
	Cin	0
	SelS2	0
	SelDato	1
	SelScrs	3
	SelDir	0
Etapa 3	SelOp	4
	SelResult	1
	SelC	1
	Cadj	0
	SelFlags	1
	SelBranch	0
	VF	1
Etapa 4	SelRegW	1
	MemW	0
	SelDir	0

Tabla 7.9. Señales de control para la instrucción LDAA.

Por el momento sólo analizaremos las señales correspondientes a la etapa 2, el resto de ellas serán analizadas en su respectiva etapa. La señal SelRegR=0 le informa al módulo de registros internos qué registros leer, en este caso, se leen valores de cero. Por otra parte, la señal S/$\overline{R}$ =1 le indica al módulo sumador/restador que realice la suma entre los operandos seleccionados con SelS2=0, SelS1=0, y el acarreo de entrada Cin=0. La señal SelS2=0 elige el segundo dato que genera el módulo de registros y la señal SelS1=0 elige el valor de cero.

La señal SelDato=1 selecciona el dato inmediato proveniente del formato de la instrucción y lo asigna al bus D5. Con SelScrs=3 se guarda el dato inmediato y el dato contenido en el bus D1 en el registro de segmentación ID/EX. Finalmente, la señal SelDir=0 selecciona el resultado obtenido por el módulo sumador/restador; este valor también se guarda en el registro de segmentación. Observe que para esta instrucción la dirección efectiva contenida en el bus D3 no se utiliza. Las señales de control para las etapas 3 y 4 también son guardadas en ID/EX (véase la figura 7.15).

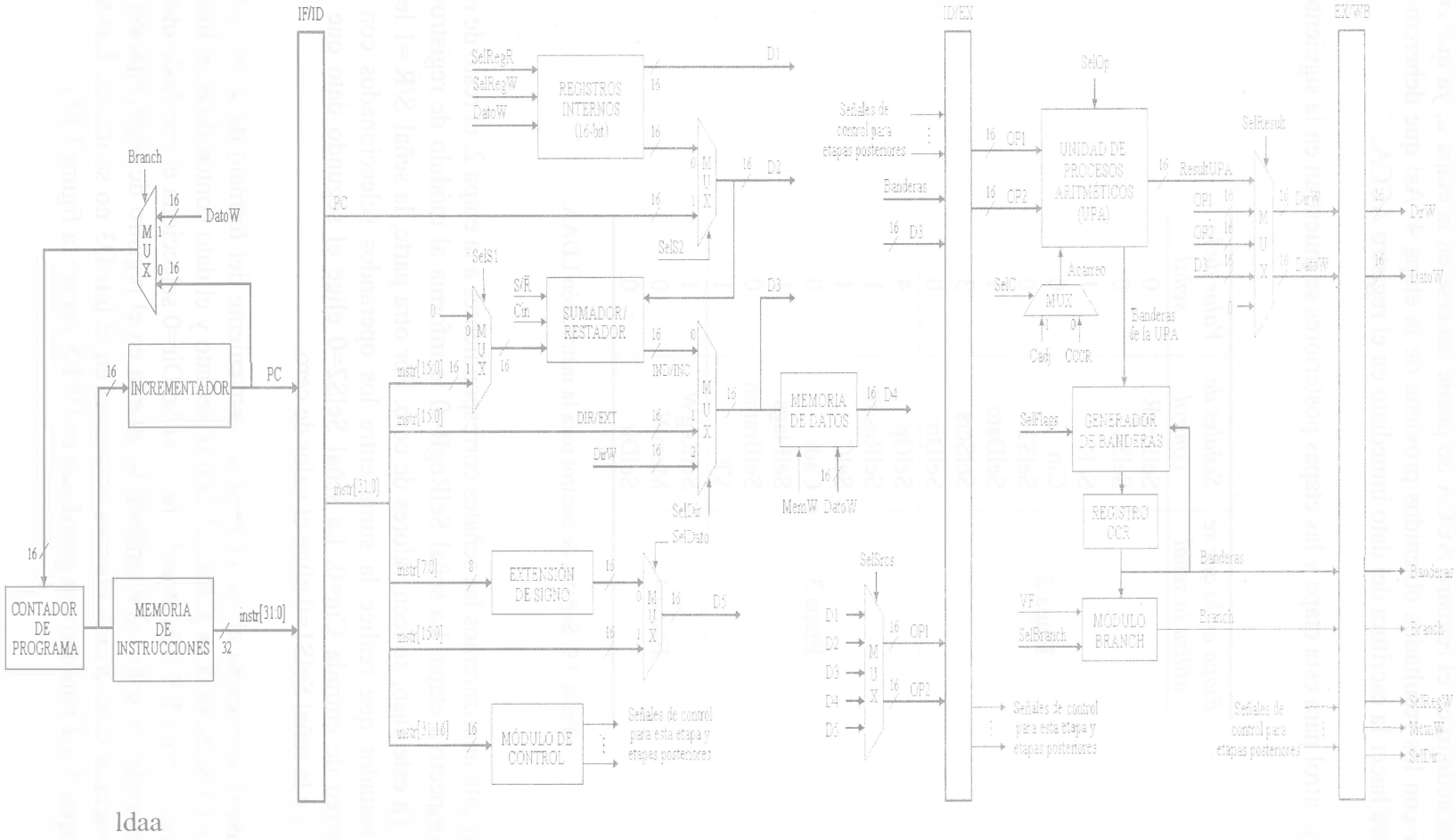
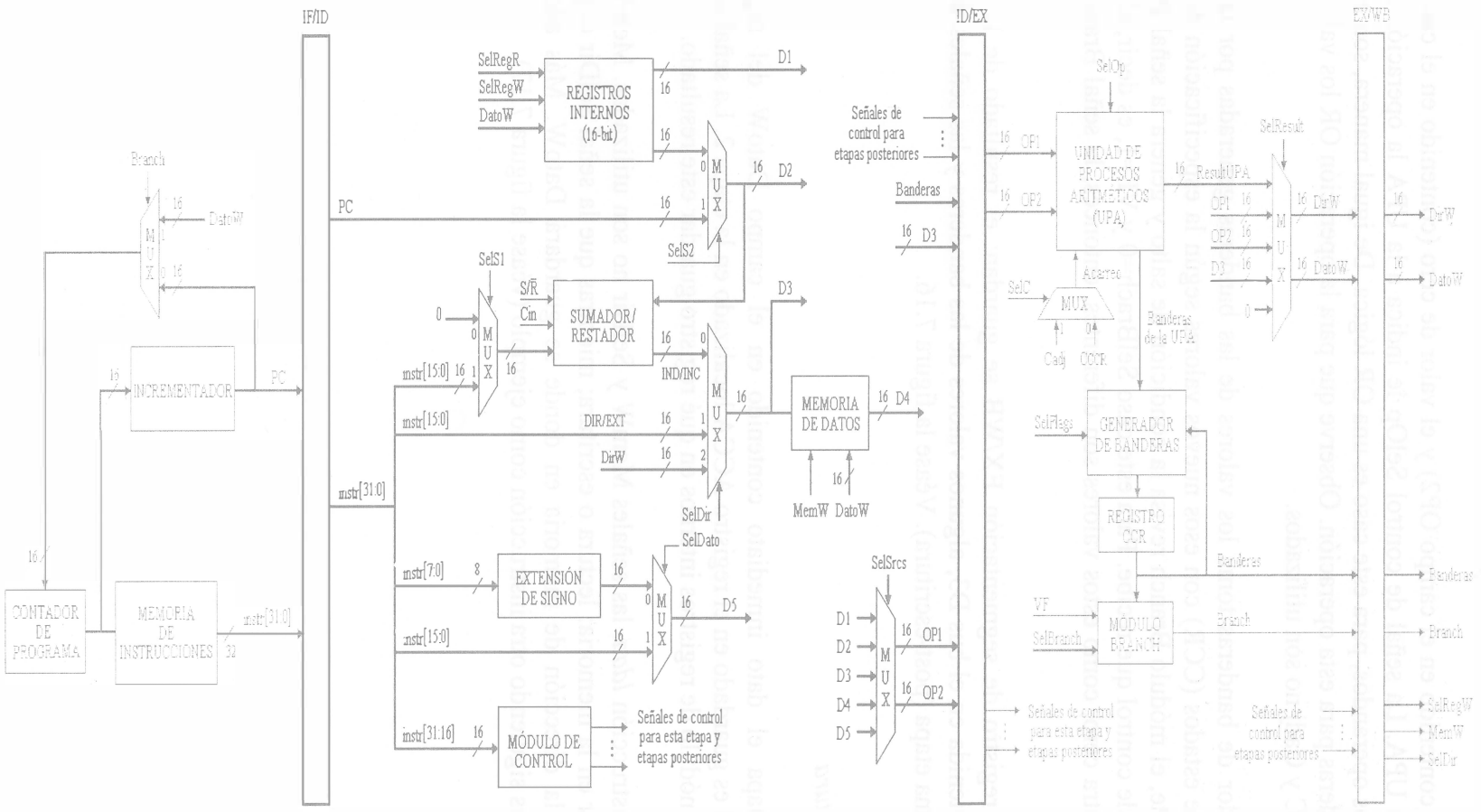


Figura 7.14. En esta primera etapa, correspondiente al primer ciclo de reloj, la instrucción LDAA es leída de la memoria de instrucciones.



ldaa

Figura 7.15. En el segundo ciclo de reloj la instrucción LDAA se encuentra en la etapa de decodificación. En esta etapa se extrae un dato inmediato almacenado en el formato de la instrucción, y se generan las señales de control para las etapas siguientes.

Etapa 3. Ejecución / Cálculo de banderas y saltos

El dato inmediato (contenido en el campo OP2) y el valor de cero (contenido en el campo OP1) son procesados por la UPA. La señal de control SelOp le indica a la UPA la operación que debe ejecutar entre estos dos operandos, para este caso es una OR lógica. De igual manera, son generados los valores de las banderas para esta operación. Observe que para la operación OR los valores de las señales de control SelC y Cadj no son utilizados.

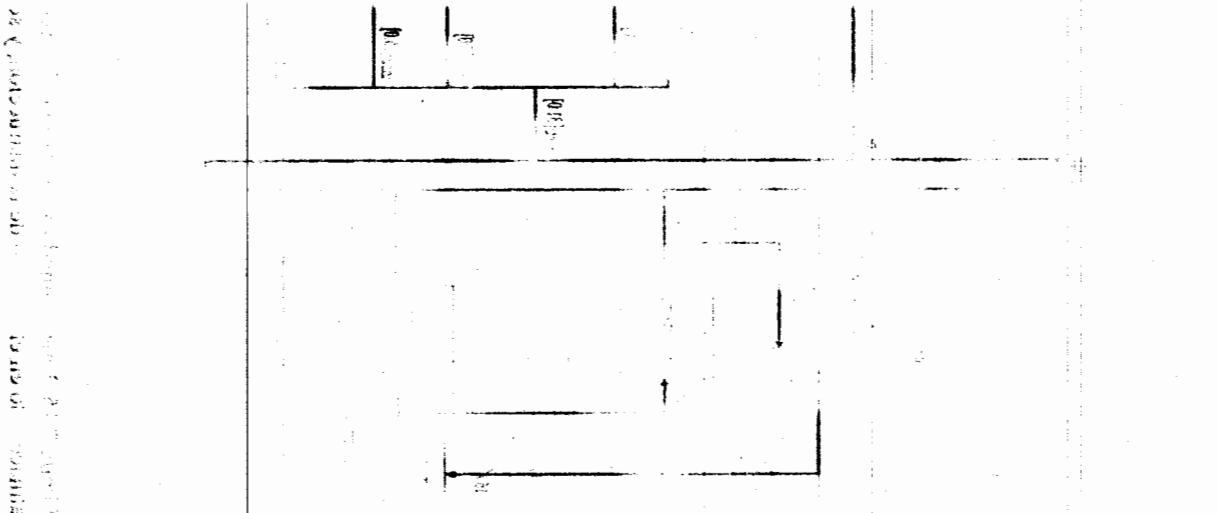
El módulo generador de banderas toma los valores de las banderas generadas por la UPA, y actualiza el registro de estados (CCR) con esos nuevos valores según la especificación de la señal SelFlags. Por otra parte, el módulo Branch revisa la condición de salto y genera la señal Branch de acuerdo a las señales de control que recibe. Para este caso SelBrach=0 y VF=1, es decir, el valor de VF es comparado contra cero, como estos valores son diferentes, entonces, la señal Branch que se genera vale cero.

Por último, en el registro de segmentación **EX/WB** se guardan: el resultado de la UPA, la dirección efectiva contenida en el bus D3, algunos valores de las banderas y las señales de control necesarias para la última etapa (post-escritura). Véase la figura 7.16.

Etapa 4. Post-escritura

En esta última etapa el dato inmediato contenido en el campo DatoW del registro de segmentación EX/WB es guardado en el registro ACCA localizado en la etapa 2. La señal de control SelRegW le indica al módulo de registros internos en qué registro guardar este resultado.

Note que para la instrucción *ldaa*, las señales MemW y SelDir no son utilizadas. MemW indica qué operación efectuar en la memoria: lectura o escritura; mientras que la señal SelDir elige el bus de donde provendrá la dirección de memoria en donde se guardará DatoW. Más adelante se analizarán estas señales siguiendo otra instrucción como ejemplo (véase la figura 7.17).



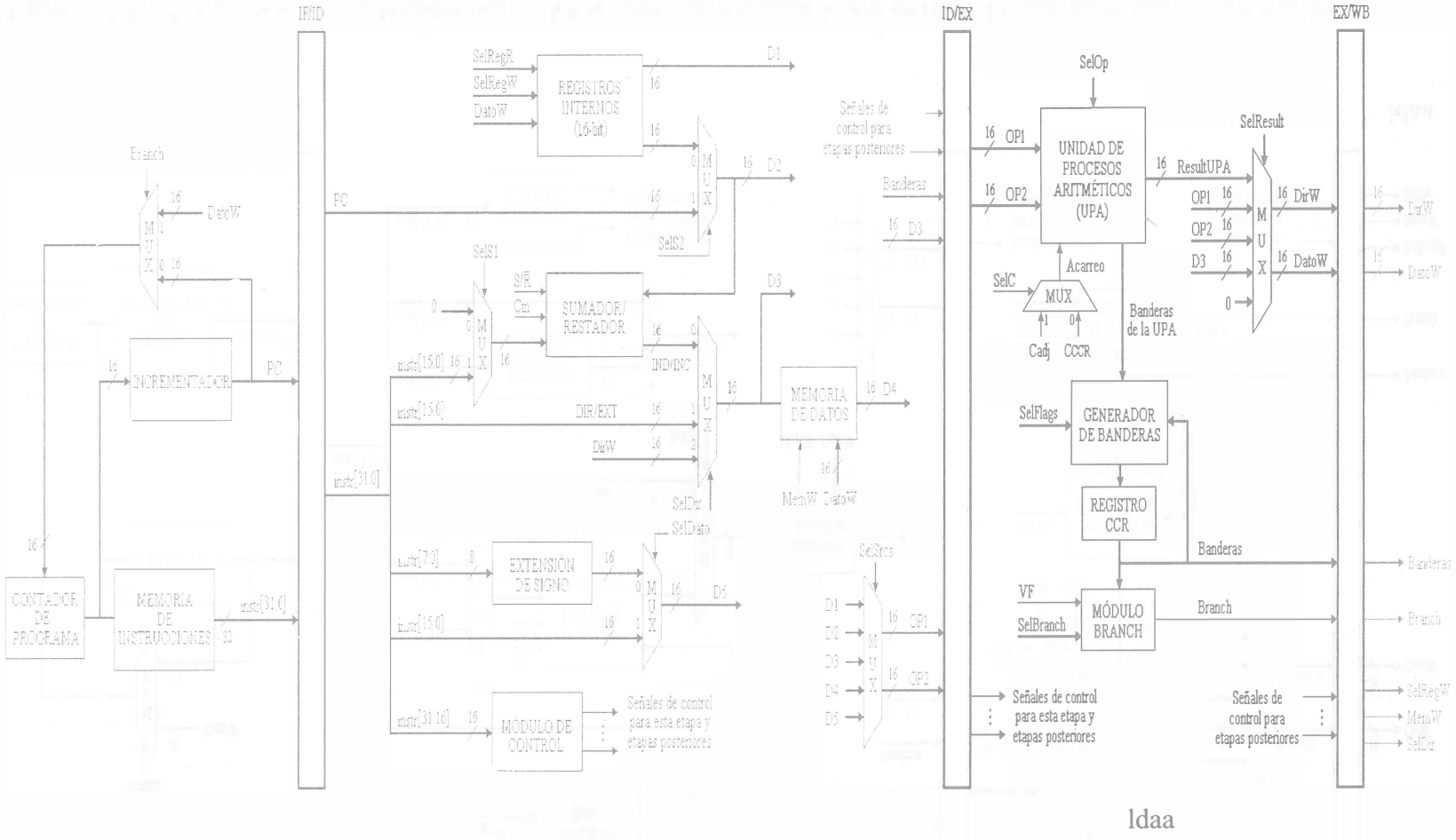


Figura 7.16. En el tercer ciclo de reloj la instrucción LDAA se encuentra en la etapa de ejecución. Durante esta etapa los operandos seleccionados en la etapa anterior son operados en la UPA, se calculan los valores de las banderas afectadas por esta instrucción y se evalúa la condición de salto.

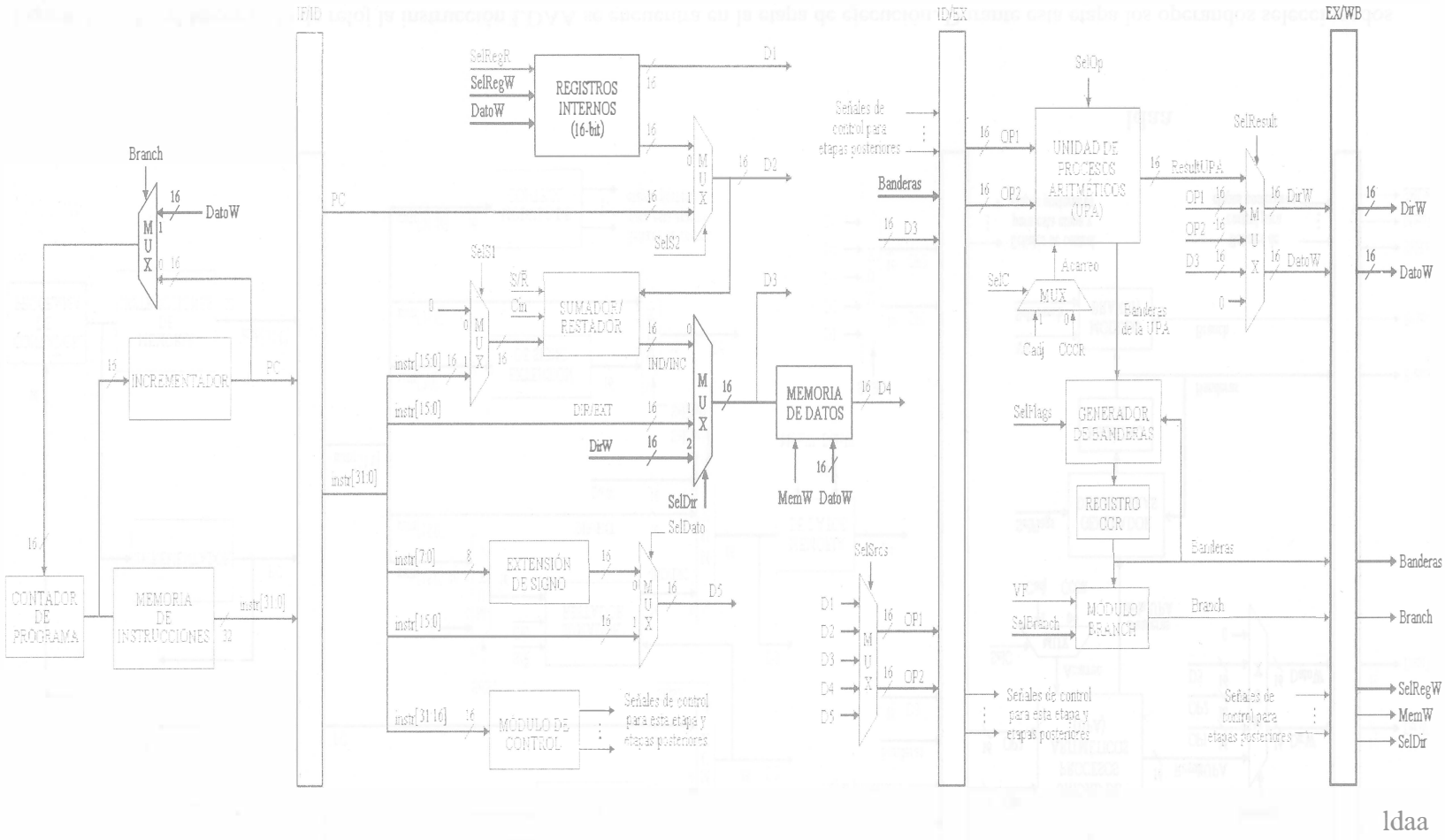


Figura 7.17. En el cuarto ciclo de reloj la instrucción LDAA se encuentra en la etapa de post-escritura. En esta última etapa, el dato inmediato, contenido en el bus DatoW, es guardado en el registro ACCA.

7.3.2 INSTRUCCIÓN ABA (Acceso Inherente)

Instrucción: ABA
Operación: ACCA ⇐ (ACCA) + (ACCB)
Código: 001B
Descripción: Suma el contenido del registro ACCA al contenido del registro ACCB, y el resultado lo guarda nuevamente en ACCA.

Banderas:
$H = ACCA7 \bullet ACCB7 + ACCB7 \bullet \overline{RES7} + \overline{RES7} \bullet ACCA7$
N=1 si el MSB del resultado está encendido, N=0 en caso contrario.
Z=1 si el resultado en el registro es cero, Z=0 en caso contrario.
$V = ACCA15 \bullet ACCB15 \bullet \overline{RES15} + \overline{ACCA15} \bullet \overline{ACCB15} \bullet RES15$
$C = ACCA15 \bullet ACCB15 + ACCB15 \bullet \overline{RES15} + \overline{RES15} \bullet ACCA15$

donde,
ACCA15 = Bit más significativo del dato contenido en el registro ACCA
ACCB15 = Bit más significativo del dato contenido en el registro ACCB
RES15 = Bit más significativo del resultado de la suma

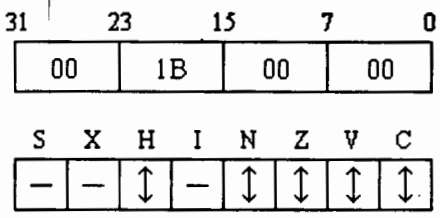


Figura 7.18. Formato de la instrucción ABA y banderas que afecta.

El comportamiento de la instrucción *aba* es el siguiente.

Etapa 1. Lectura de la instrucción

En esta etapa se lee de la memoria la instrucción a ejecutar, es decir, la instrucción *aba*. La dirección de esta instrucción, dada por el contador de programa, se incrementa y se vuelve a cargar en el contador de programa. Este nuevo valor corresponde a la dirección en memoria de la próxima instrucción a ejecutar. La dirección incrementada y la instrucción se guardan en el registro de segmentación IF/ID para su uso posterior en la etapa 2 (ver figura 7.19).

Etapa 2. Decodificación / Cálculo de la dirección efectiva / Lectura de operandos

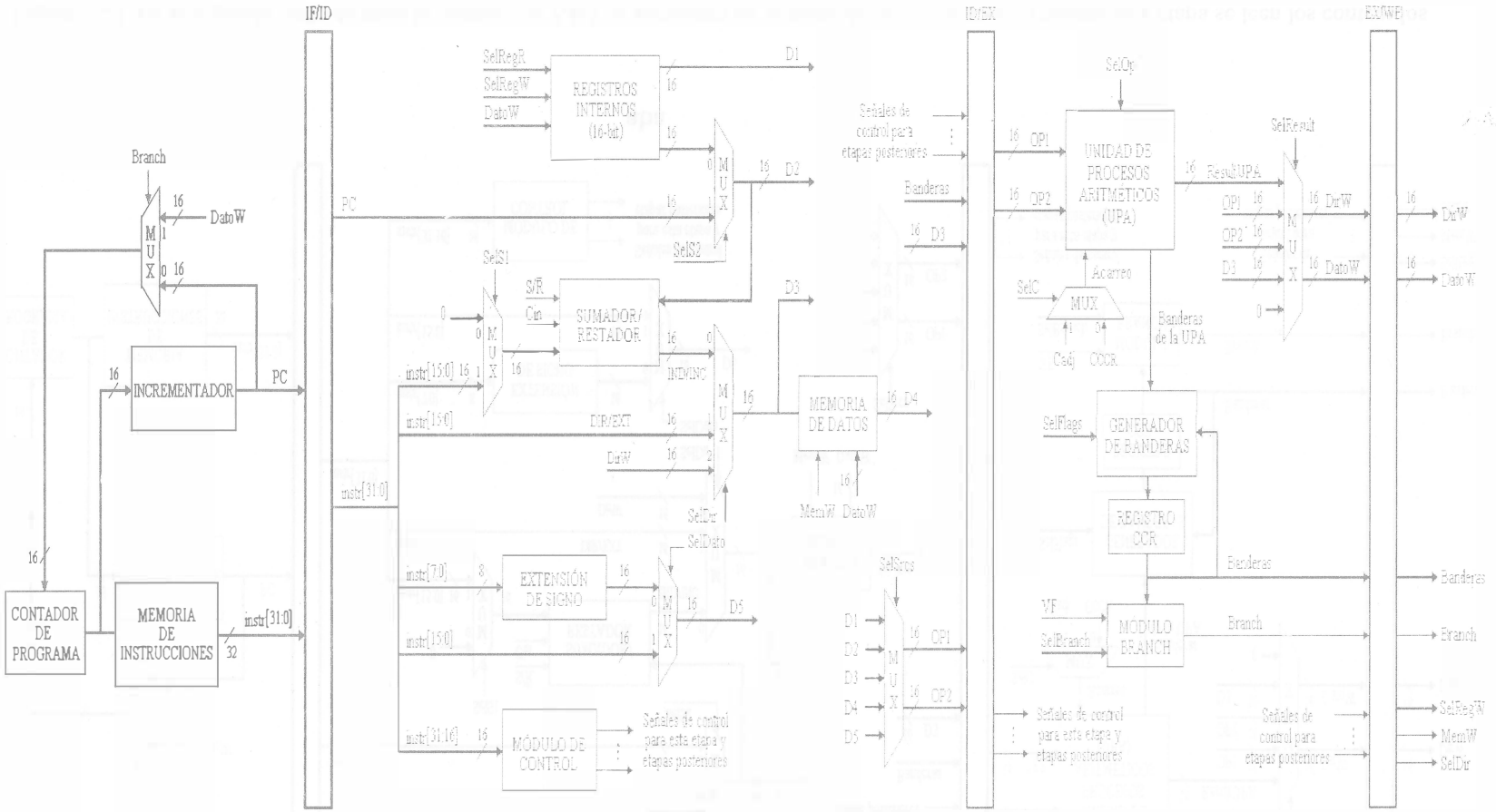
El modo de direccionamiento para esta instrucción es inherente, es decir, el código de la instrucción es suficiente para saber el tipo de instrucción y la tarea que debe ejecutar. Las señales de control para esta etapa y las etapas posteriores se muestran en la siguiente tabla.

Etapa en la que se utiliza la señal	*Señales de control*	*Valor de la señal*
Etapa 2	SelRegR	1
	SelS1	0
	S/$\overline{R}$	1
	Cin	0
	SelS2	0
	SelDato	1
	SelScrs	1
	SelDir	0
Etapa 3	SelOp	1
	SelResult	1
	SelC	1
	Cadj	0
	SelFlags	2
	SelBranch	0
	VF	1
Etapa 4	SelRegW	1
	MemW	0
	SelDir	0

Tabla 7.10. Señales de control para la instrucción ABA.

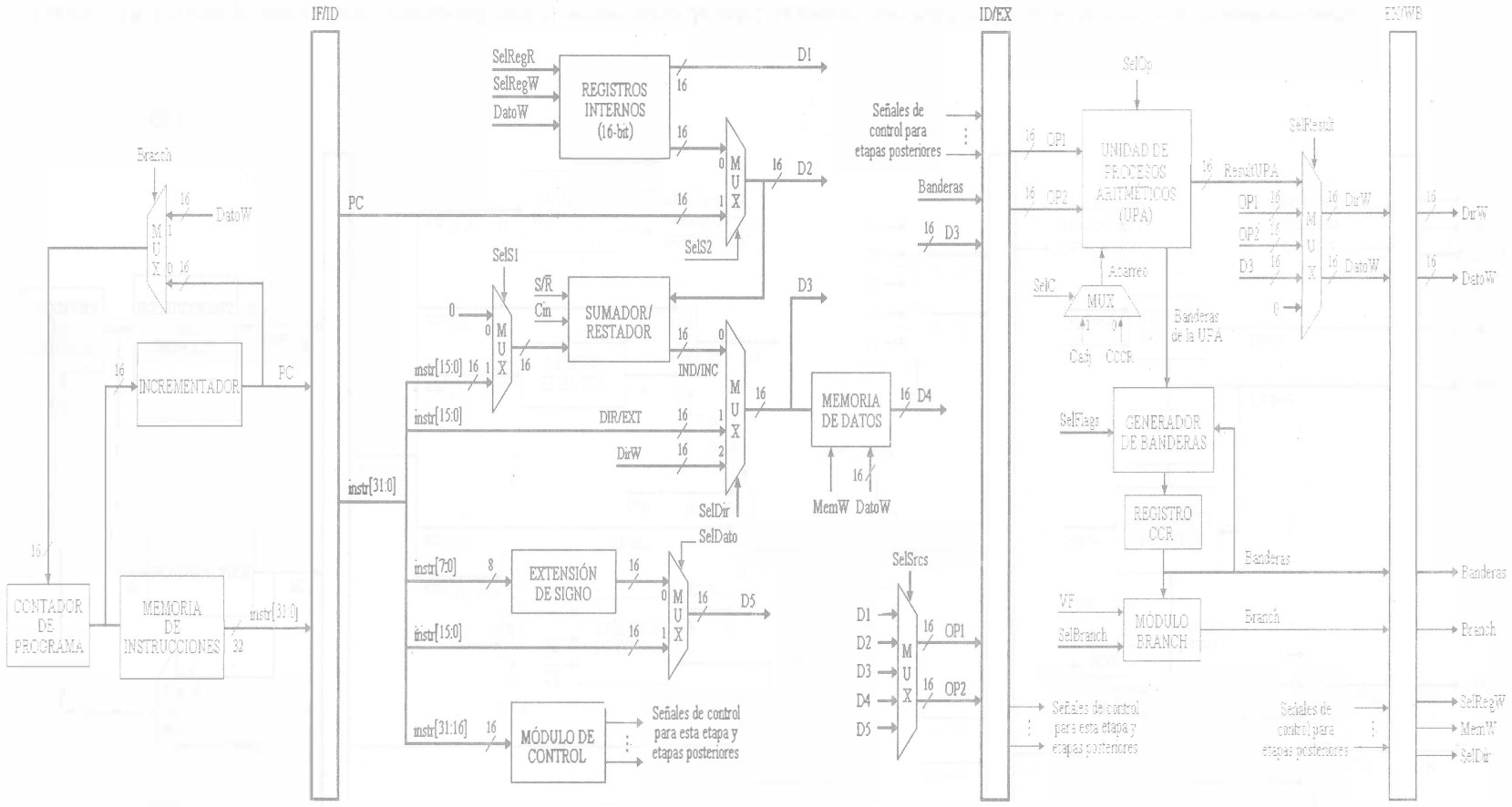
La señal SelRegR=1 permite leer el contenido de los registros ACCA y ACCB. Con SelS2=0 seleccionamos la segunda fuente del módulo de registros, de manera que, el contenido de ACCB está disponible en el bus D2 y el contenido de ACCA en el bus D1.

Para esta instrucción no es necesario calcular la dirección efectiva de los operandos en memoria, ya que éstos son leídos de los registros internos; por lo tanto, los datos seleccionados por medio de las señales SelDir y SelDato son ignorados por esta instrucción. Los datos que verdaderamente utilizaremos son los operandos de los buses D1 y D2, los cuales son seleccionados por medio de la señal SelSrcs=1, y guardados en el registro de segmentación ID/EX. Las señales de control para las etapas 2 y 3 también son guardadas en el registro de segmentación, así como la dirección efectiva del bus D3, que como se mencionó anteriormente, no se utiliza para esta instrucción (ver figura 7.20).



aba

Figura 7.19. En esta primera etapa, correspondiente al primer ciclo de reloj, la instrucción ABA es leída de la memoria de instrucciones.



aba

Figura 7.20. En el segundo ciclo de reloj la instrucción ABA se encuentra en ıa etapa de decodificación. Durante esta etapa se leen los contenıdos de los registros ACCA y ACCB del módulo de registros internos, y se generan las señales de control para las etapas siguientes.

Etapa 3. Ejecución / Cálculo de banderas y saltos

En esta etapa, los operandos contenidos en los campos OP1 y OP2 del registro de segmentación ID/EX son procesados por la UPA. La señal de control SelOp=1 hace que la UPA calcule la suma entre OP1, OP2 y un acarreo de entrada, el cual es seleccionado por medio de la señal SelC=1. Recuerde que la operación de adición en la UPA se define de la siguiente manera: OP1 + OP2 + Acarreo, por lo tanto, es necesario obligar a que el valor del acarreo sea cero, pues con ello se garantiza que el resultado de la suma no es afectado por un acarreo indeseado.

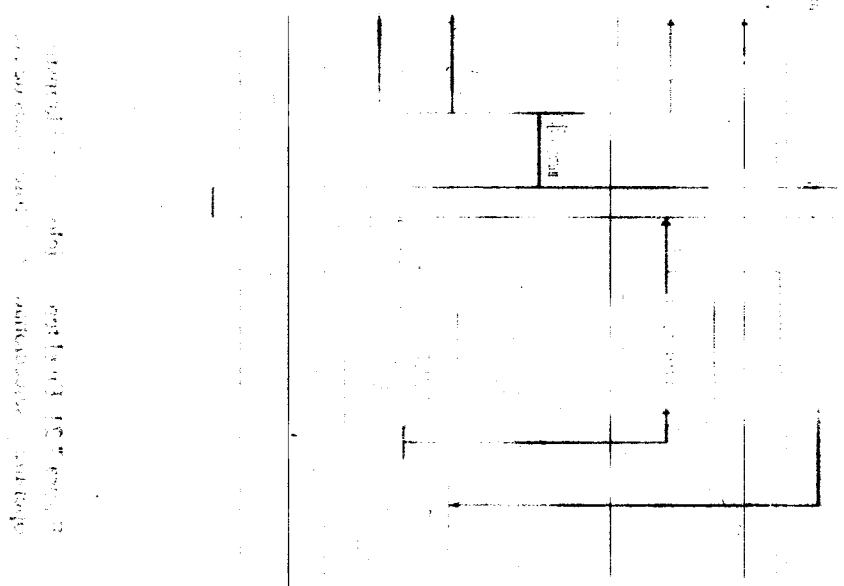
La UPA también calculará los valores de las banderas de acuerdo con la especificación de la instrucción. Estos valores de banderas son leídos por el módulo generador de banderas, el cual con base en la señal SelFlags, actualizará las banderas en el registro de estados (registro CCR).

El módulo Branch revisa la condición de salto dada por la señal SelBranch; si al evaluar la condición ésta resulta ser igual al valor de la señal VF, entonces, la señal Branch se activa, de lo contrario, Branch vale cero. Para este caso, Branch vale cero pues la condición de salto vale cero y VF vale uno, visto de otra manera, no se realiza el salto, lo cual suena lógico ya que la instrucción *aba* no es una instrucción de salto.

Finalmente, la señal SelResult selecciona el resultado de la UPA y la dirección efectiva guardada en el campo D3 del registro de segmentación ID/EX, y las guarda en el registro de segmentación EX/WB. En este registro también se guardan algunas banderas y las señales de control necesarias para la última etapa (ver figura 7.21).

Etapa 4. Post-escritura

Sólo falta guardar el resultado obtenido en la etapa anterior en el registro ACCA, para esto, la señal SelRegW es colocada a 1. Las señales SelDir y MemW valen cero ya que no deseamos escribir nada en memoria (ver figura 7.22).



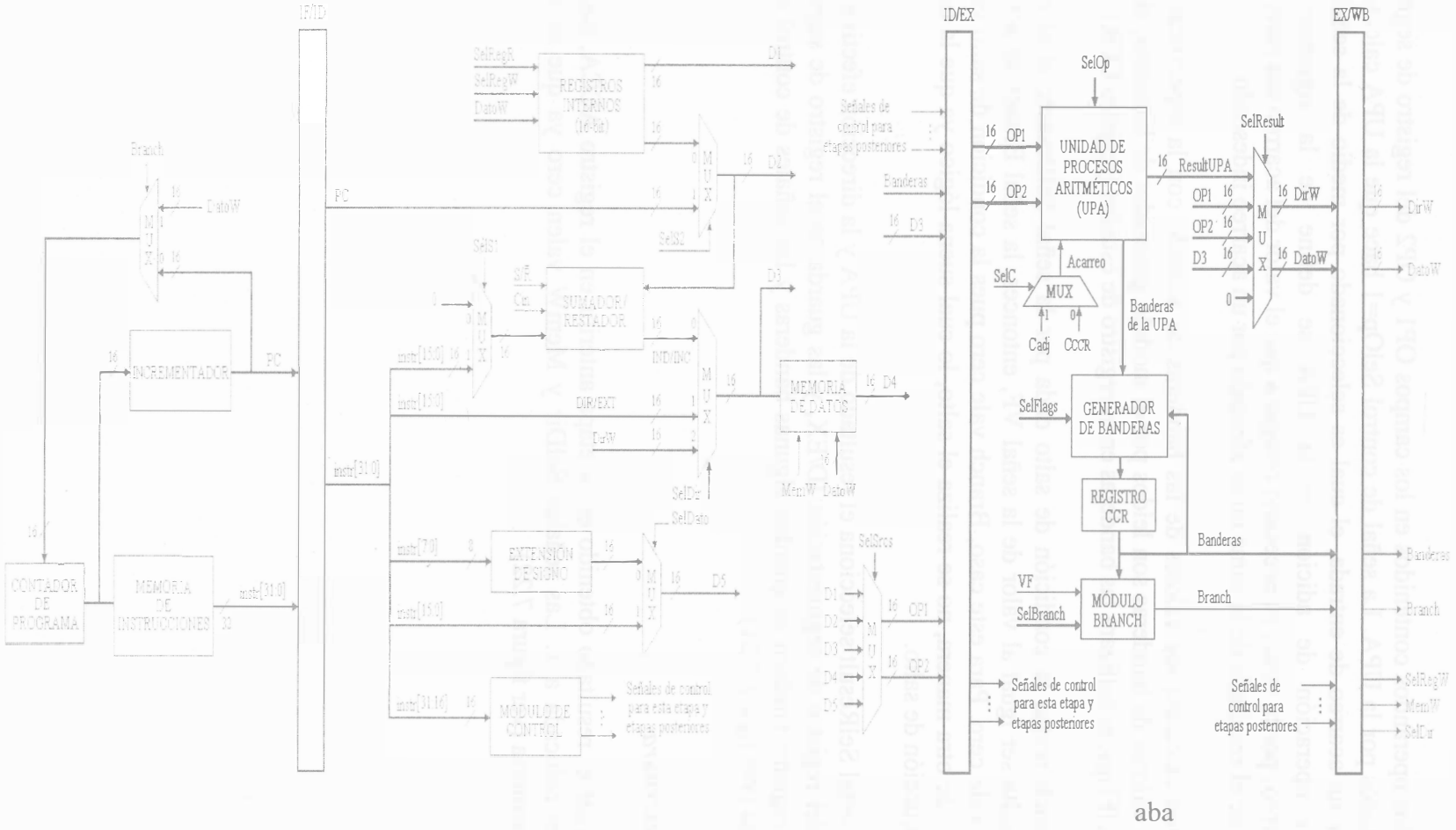


Figura 7.21. En el tercer ciclo de reloj la instrucción ABA se encuentra en la etapa de ejecución. Durante esta etapa se calcula la suma entre los operandos seleccionados (los contenidos de ACCA y ACCB), se calculan los valores de las banderas afectadas y se evalúa la condición de salto.

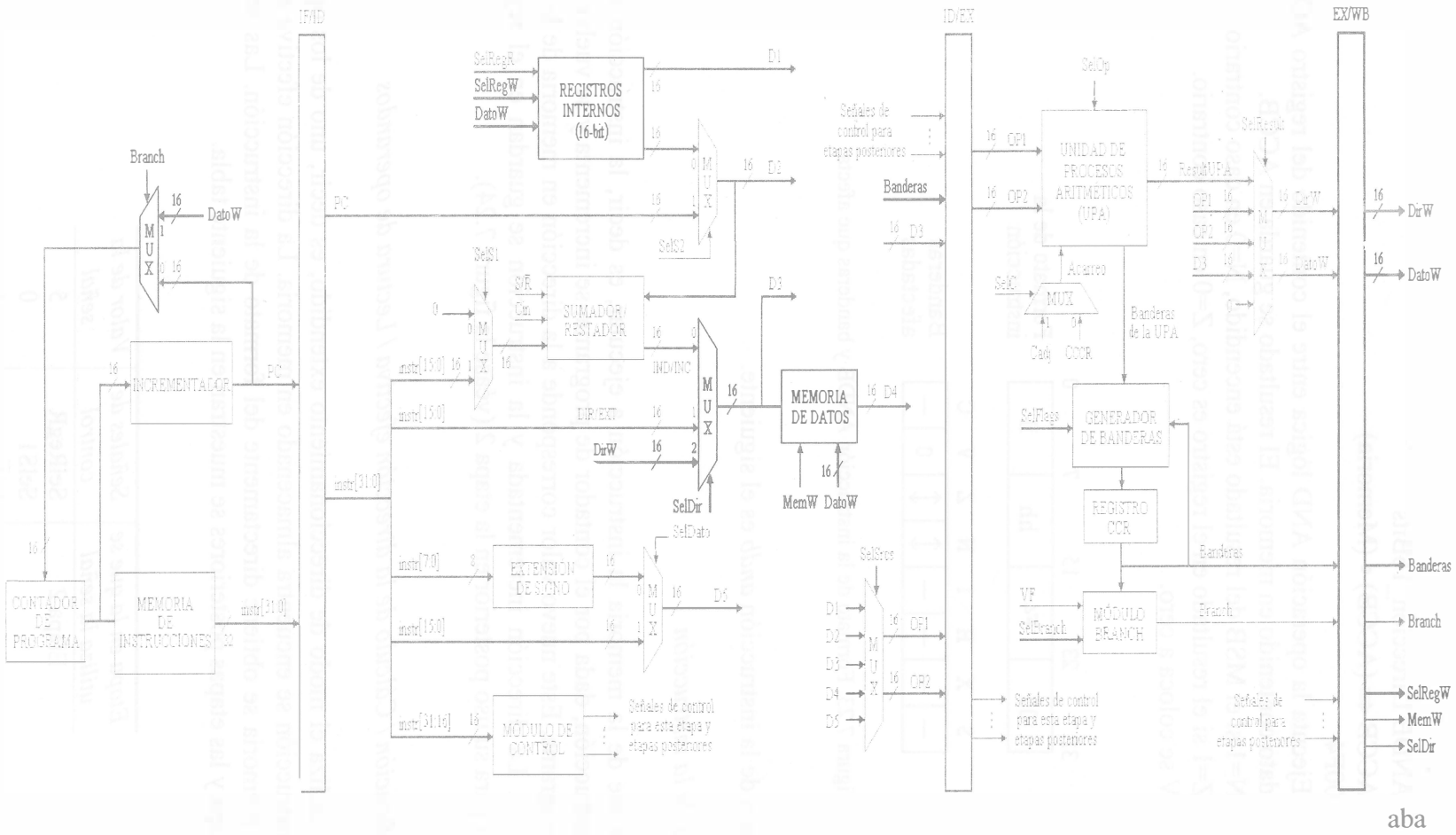


Figura 7.22. En el cuarto ciclo de reloj la instrucción ABA se encuentra en la etapa de post-escritura. En esta última etapa, el resultado de la suma, contenido en el bus DatoW, es guardado en el registro ACCA.

7.3.3 INSTRUCCIÓN ANDB (Acceso Extendido)

Instrucción: ANDB Dirección_16Bits
Operación: ACCB ⇐ (ACCB) • (Memoria)
Código: 00F4
Descripción: Ejecuta la operación AND lógica entre el contenido del registro ACCB y un dato contenido en memoria. El resultado se guarda en ACCB.
Banderas: N=1 si el MSB del resultado está encendido, N=0 en caso contrario.
Z=1 si el resultado en el registro es cero, Z=0 en caso contrario.
V se coloca a cero.

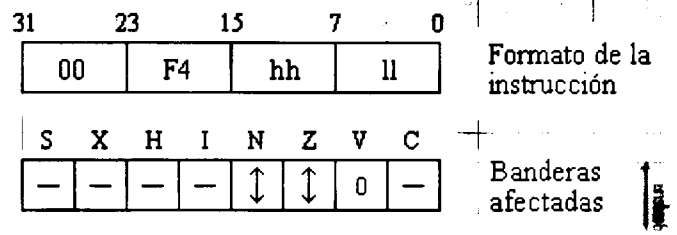


Figura 7.23. Formato de la instrucción ANDB y banderas que afecta.

El comportamiento de la instrucción *andb* es el siguiente.

Etapa 1. Lectura de la instrucción

En esta etapa se lee de la memoria la instrucción a ejecutar, es decir, la instrucción *andb*. La dirección de esta instrucción, dada por el contador de programa, se incrementa y se vuelve a cargar en el contador de programa. Este nuevo valor corresponde a la dirección en memoria de la próxima instrucción a ejecutar. La dirección incrementada y la instrucción se guardan en el registro de segmentación IF/ID para su uso posterior en la etapa 2 (véase la figura 7.24).

Etapa 2. Decodificación / Cálculo de la dirección efectiva / Lectura de operandos

Esta instrucción utiliza el modo de direccionamiento extendido, es decir, uno de los operandos necesarios por la instrucción se encuentra almacenado en memoria. La dirección efectiva en donde se aloja el dato en memoria se obtiene directamente del formato de la instrucción. Las señales de control para esta etapa y las etapas posteriores se muestran en la siguiente tabla.

Etapa en la que se utiliza la señal	*Señales de control*	*Valor de la señal*
Etapa 2	SelRegR	5
	SelS1	0
	S/$\overline{R}$	1
	Cin	0

	SelS2	0
	SelDato	1
	SelScrs	2
	SelDir	1
Etapa 3	SelOp	3
	SelResult	1
	SelC	1
	Cadj	0
	SelFlags	1
	SelBranch	0
	VF	1
Etapa 4	SelRegW	4
	MemW	0
	SelDir	0

Tabla 7.11. Señales de control para la instrucción ANDB.

El primer operando se lee del registro ACCB por medio de la señal SelRegR=5, y el segundo operando se lee de la memoria de datos. La dirección efectiva para el segundo operando se extrae del formato de la instrucción, esta dirección se pasa a la memoria mediante SelDir=1, y el dato contenido en esa dirección es leído y colocado en el bus D4. A continuación, con SelScrs=2 se escriben los datos del bus D1 (con el contenido del registro ACCB) y del bus D4 (con el contenido de memoria) en el registro de segmentación ID/EX. También guardamos en el registro ID/EX las señales de control para las etapas 3 y 4, así como la dirección efectiva contenida en el bus D3 (véase la figura 7.25).

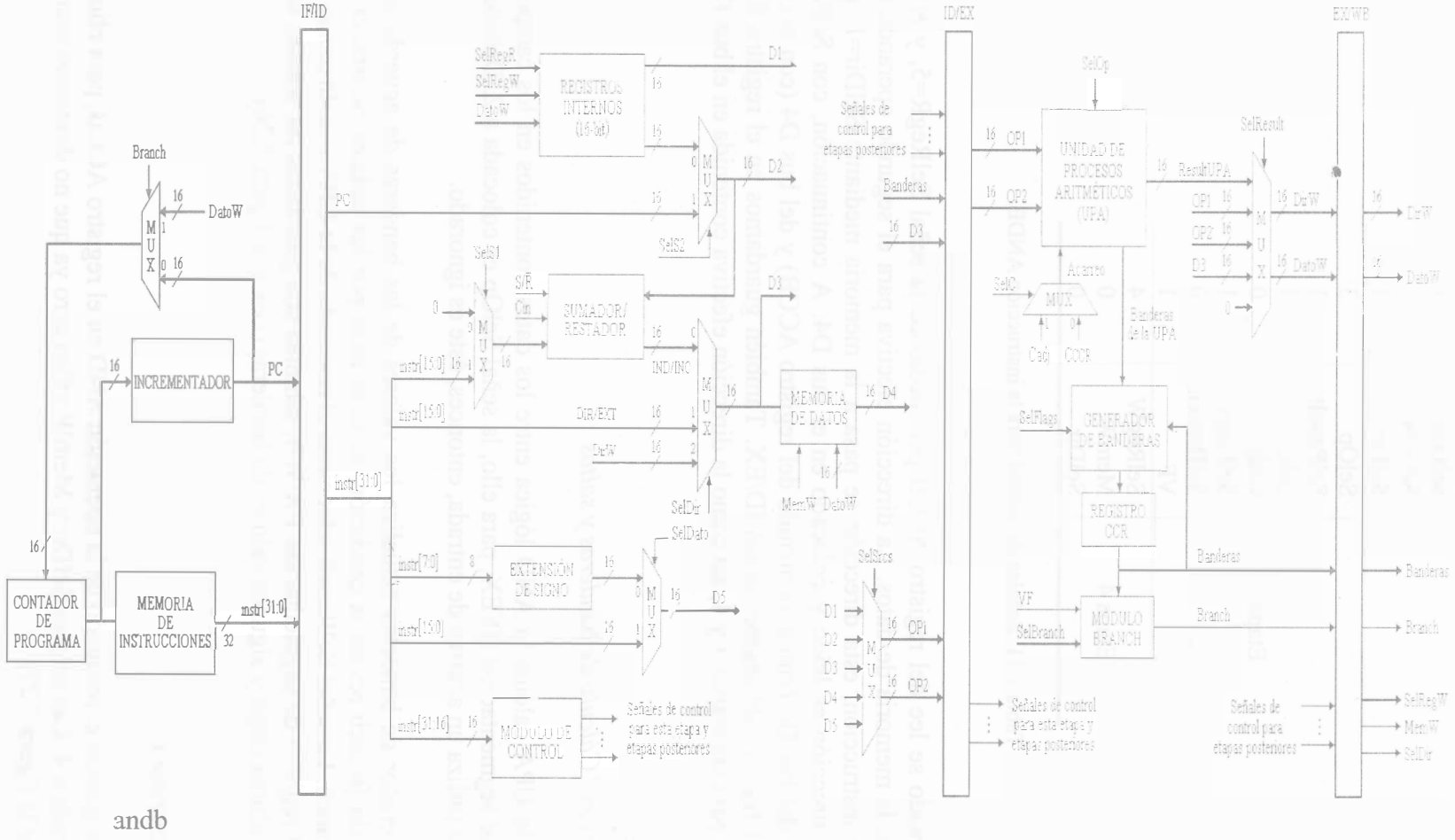
Etapa 3. Ejecución / Cálculo de banderas y saltos

En esta etapa, la UPA calcula la AND lógica entre los datos contenidos en los campos OP1 y OP2 del registro de segmentación ID/EX, para ello, la señal SelOp es colocada a 2. Debido a que la operación AND no utiliza un acarreo de entrada, entonces, éste es ignorado.

El módulo generador de banderas actualiza los valores de las banderas de acuerdo a la señal SelFlags. El módulo Branch revisa la condición de salto dada por las señales SelBranch y VF, y genera la señal Branch. La señal SelResult selecciona el resultado de la UPA y la dirección efectiva, y los guarda en el registro de segmentación EX/WB; también son guardadas las señales de control necesarias para la última etapa y algunos valores de banderas (véase la figura 7.26).

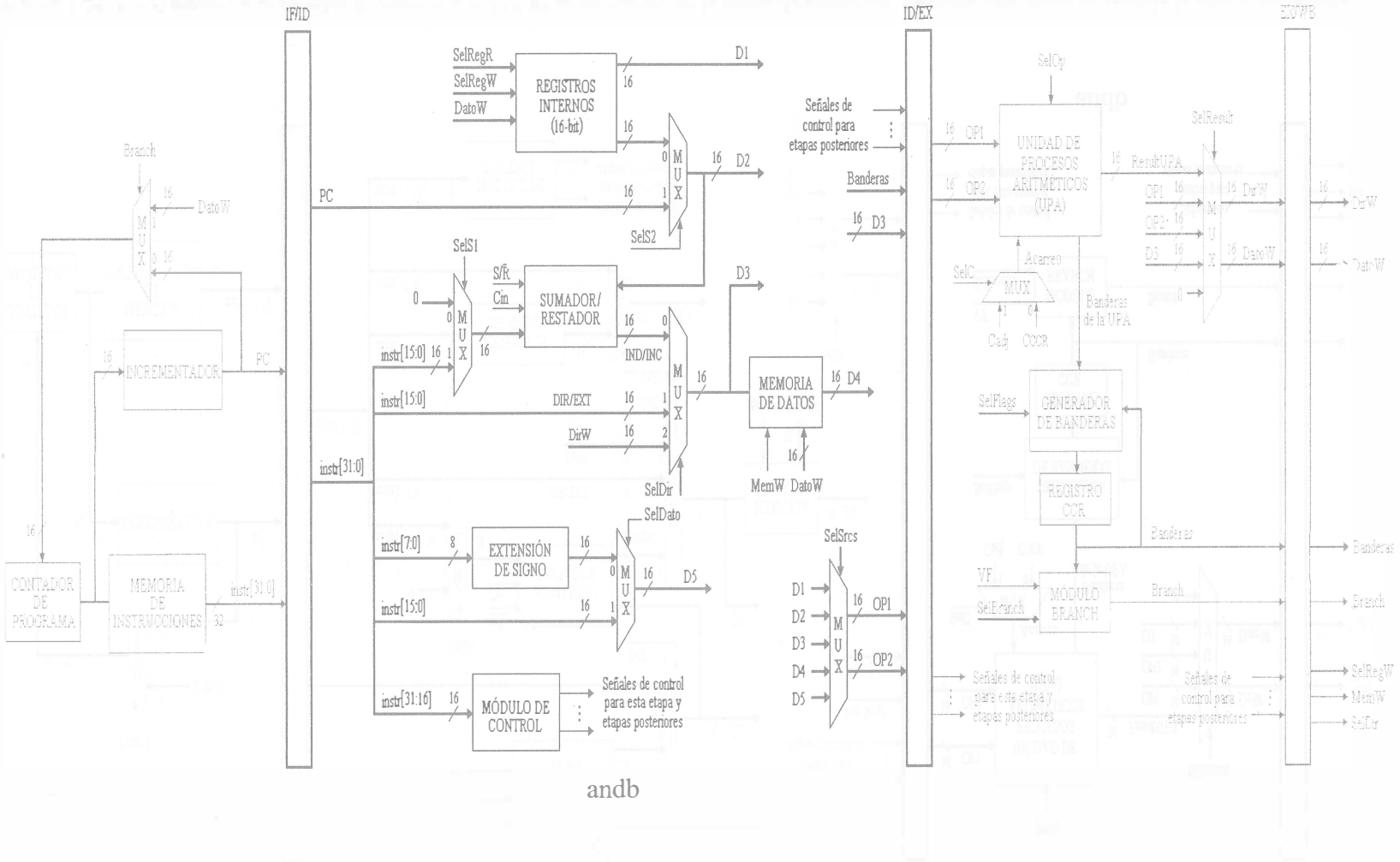
Etapa 4. Post-escritura

En esta etapa se guarda el resultado de la operación AND en el registro ACCB, para ello, la señal SelRegW es colocada a 4. Las señales SelDir y MemW valen cero ya que no deseamos escribir nada en memoria (véase la figura 7.27).



andb

Figura 7.24. En esta primera etapa, correspondiente al primer ciclo de reloj, la instrucción ANDB es leída de la memoria de instrucciones.



andb

Figura 7.25. En el segundo ciclo de reloj la instrucción ANDB se encuentra en la etapa de decodificación. Durante esta etapa se obtienen los operandos que necesita la instrucción, el contenido del registro ACCB y un dato contenido en memoria, y se generan las señales de control para las etapas siguientes.

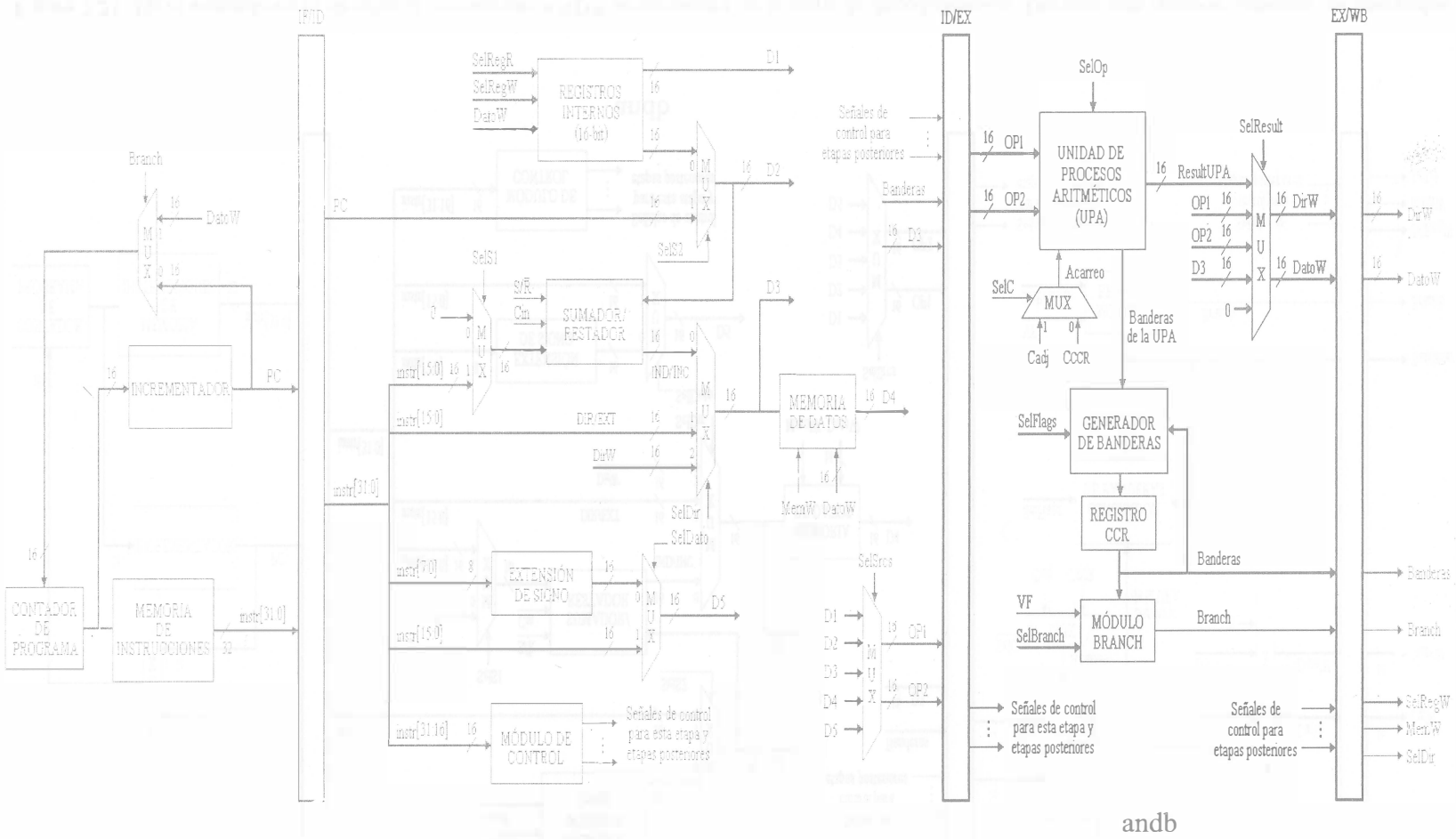


Figura 7.26. En el tercer ciclo de reloj la instrucción ANDB se encuentra en la etapa de ejecución. Durante esta etapa se calcula la operación lógica AND entre los operandos seleccionados (el registro ACCB y el dato de memoria), se actualizan los valores de las banderas afectadas y se evalúa la condición de salto.

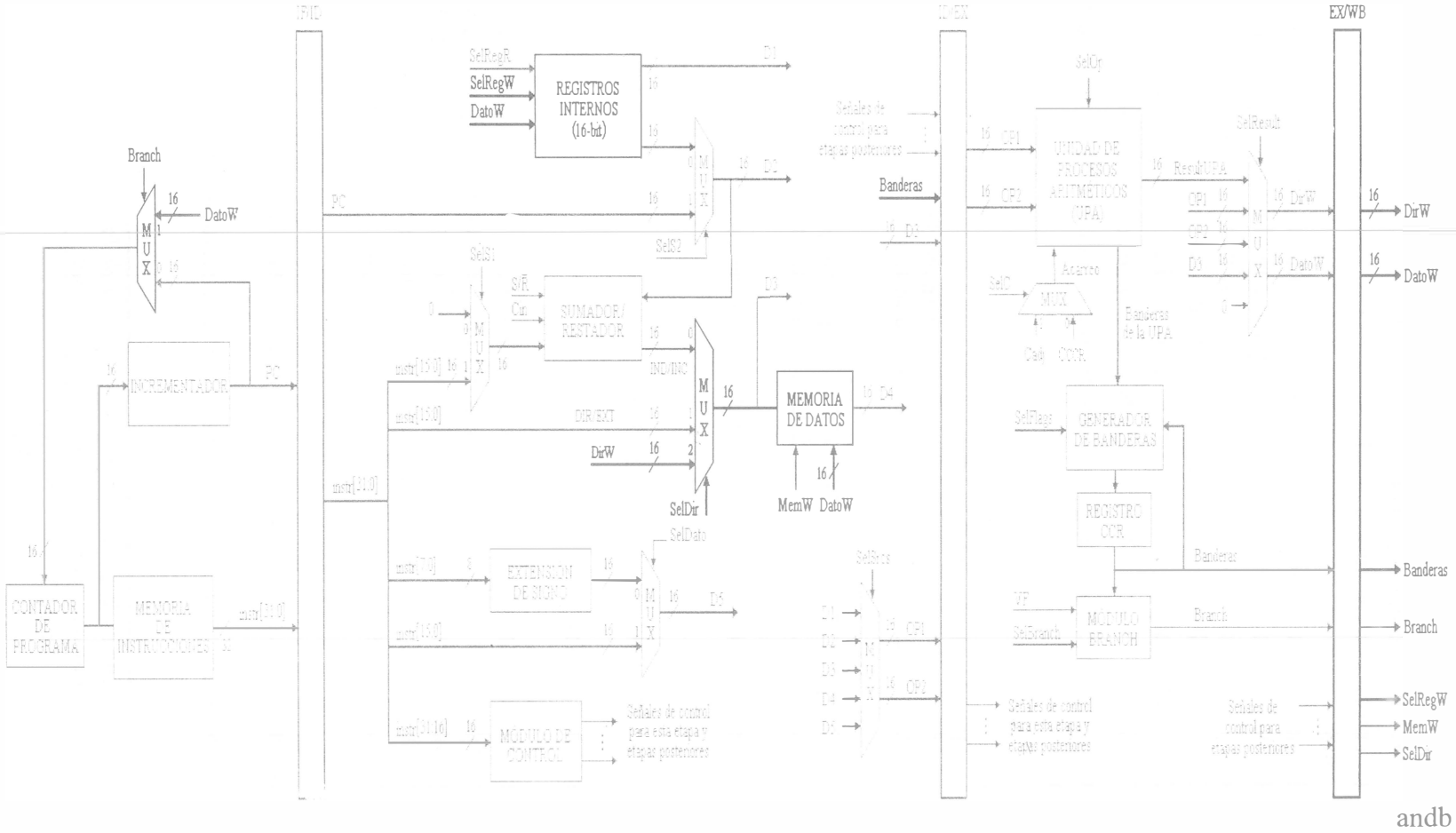


Figura 7.27. En el cuarto ciclo de reloj la instrucción ANDB se encuentra en la etapa de post-escritura. En esta última etapa, el resultado de la operación lógica, contenido en el bus DatoW, es guardado en el registro ACCB.

7.3.4 INSTRUCCIÓN ASL (Acceso Indexado)

Instrucción: ASL Desplazamiento_8Bits_sin_signo, Y
Operación: (Memoria) ⇐ (Memoria) << 1
Código: 1868
Descripción: Realiza un corrimiento hacia la izquierda del dato contenido en memoria. El bit de acarreo del registro CCR es cargado con el bit más significativo del dato de memoria, mientras que el bit 0 del dato de memoria es cargado con cero.
Banderas: N=1 si el MSB del resultado está encendido, N=0 en caso contrario.
Z=1 si el resultado en el registro es cero, Z=0 en caso contrario.
$V = N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ para N y C después del corrimiento.
C=1 si el MSB del dato de memoria (antes del corrimiento) está encendido, C=0 en caso contrario.

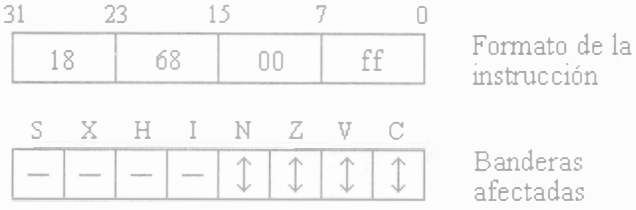


Figura 7.28. Formato de la instrucción ASL y banderas que afecta.

El comportamiento de la instrucción *asl* es el siguiente.

Etapa 1. Lectura de la instrucción

En esta etapa se lee de la memoria la instrucción a ejecutar, es decir, la instrucción *asl*. La dirección de esta instrucción, dada por el contador de programa, se incrementa y se vuelve a cargar en el contador de programa. Este nuevo valor corresponde a la dirección en memoria de la próxima instrucción a ejecutar. La dirección incrementada y la instrucción se guardan en el registro de segmentación IF/ID para su uso posterior en la etapa 2 (véase la figura 7.30).

Etapa 2. Decodificación / Cálculo de la dirección efectiva / Lectura de operandos

Las señales de control para las etapas 2, 3 y 4 se muestran en la siguiente tabla.

Etapa en la que se utiliza la señal	*Señales de control*	*Valor de la señal*
Etapa 2	SelRegR	A
	SelS1	1
	S/R	1
	Cin	0

	SelS2	0
	SelDato	1
	SelScrs	4
	SelDir	0
Etapa 3	SelOp	6
	SelResult	1
	SelC	1
	Cadj	0
	SelFlags	3
	SelBranch	0
	VF	1
Etapa 4	SelRegW	0
	MemW	1
	SelDir	2

Tabla 7.12. Señales de control para la instrucción ASL.

Esta instrucción utiliza el modo de direccionamiento indexado, por lo tanto, la dirección efectiva del operando en memoria se calcula sumando al contenido del registro índice IY el desplazamiento de 8 bits sin signo.

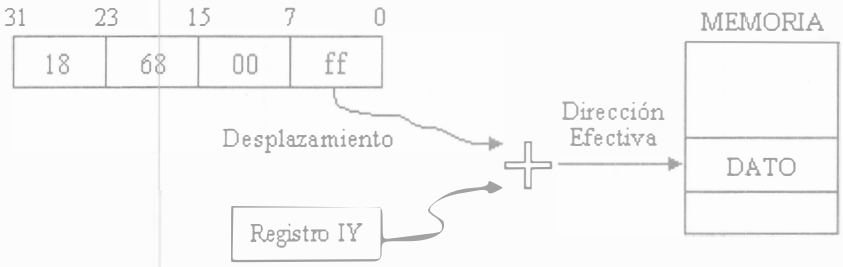


Figura 7.29. Cálculo de la dirección efectiva para el modo de direccionamiento indexado.

La señal SelRegR=A lee el contenido del registro índice IY del módulo de registros internos. La señal SelS1=1 selecciona el desplazamiento de 8 bits sin signo proveniente del formato de la instrucción, y el módulo sumador/restador suma al desplazamiento el dato seleccionado con SelS2=0, es decir, con el contenido del registro IY. De esta manera, el módulo sumador/restador calcula la dirección efectiva del dato en memoria. A continuación, la señal SelDir=0 pasa esta dirección a la memoria, y el dato contenido en dicha localidad es leído y colocado en el bus D4. En esta ocasión la señal SelDato vale uno, pero el dato que selecciona es ignorado.

Finalmente, la señal SelSrcs=4 toma los contenidos de los buses D4 y D3, y los guarda en los campos OP1 y OP2 del registro de segmentación ID/EX, respectivamente. Recuerde que las señales de control para las etapas siguientes y la dirección efectiva también son guardadas en este registro. Note que para esta instrucción sí es necesario guardar la dirección efectiva, pues de no hacerlo, no se sabrá en donde almacenar el resultado del corrimiento que se obtendrá en la siguiente etapa (véase la figura 7.31).

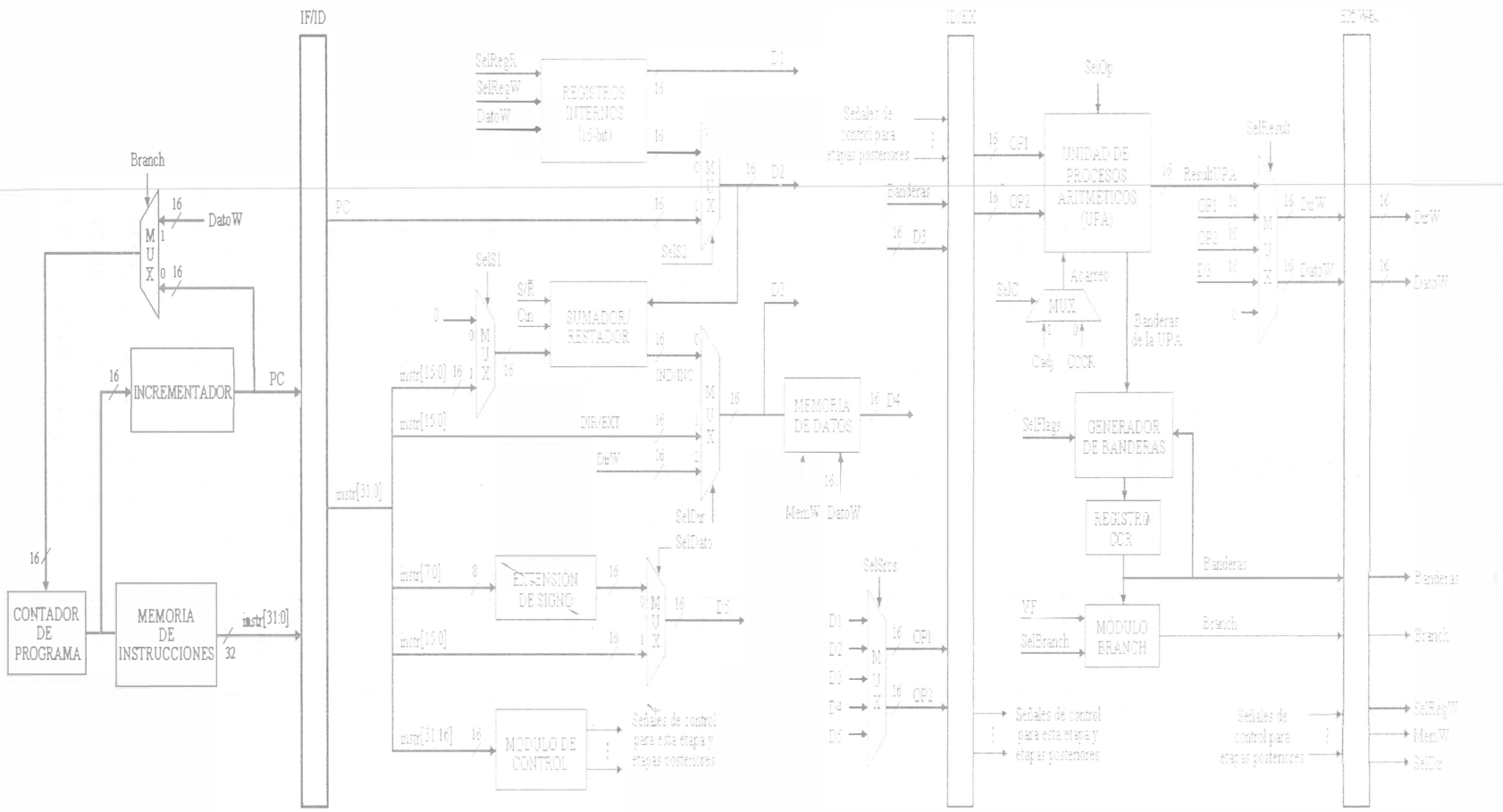
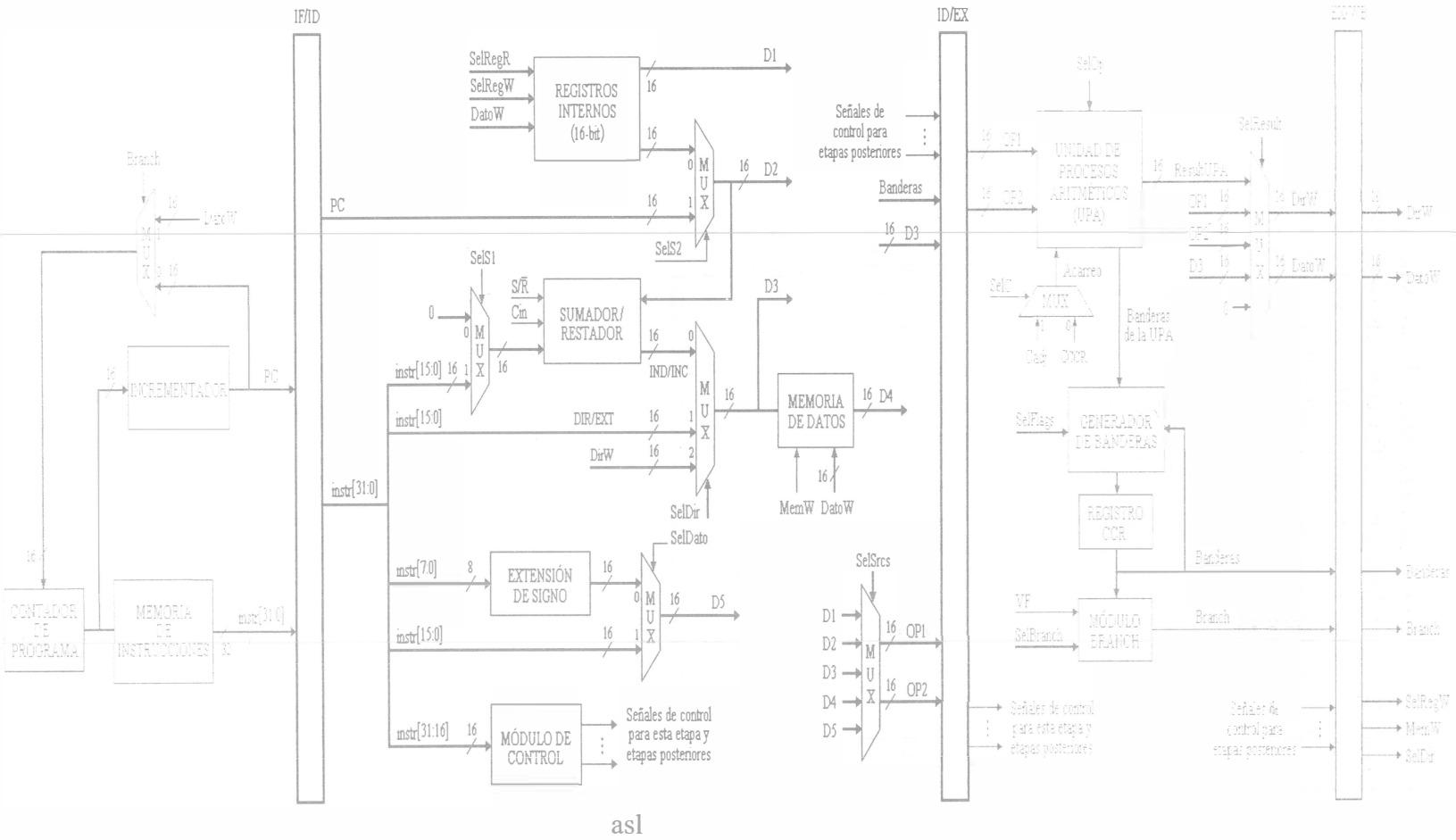


Figura 7.30. En esta primera etapa, correspondiente al primer ciclo de reloj, la instrucción ASL es leída de la memoria de instrucciones.



asl

Figura 7.31. En el segundo ciclo de reloj la instrucción ASL se encuentra en la etapa de decodificación. Durante esta etapa se calcula la direccion efectiva de un operando en memoria, se lee dicho operando y se generan las señales de control para las etapas siguientes.

Etapa 3. Ejecución / Cálculo de banderas y saltos

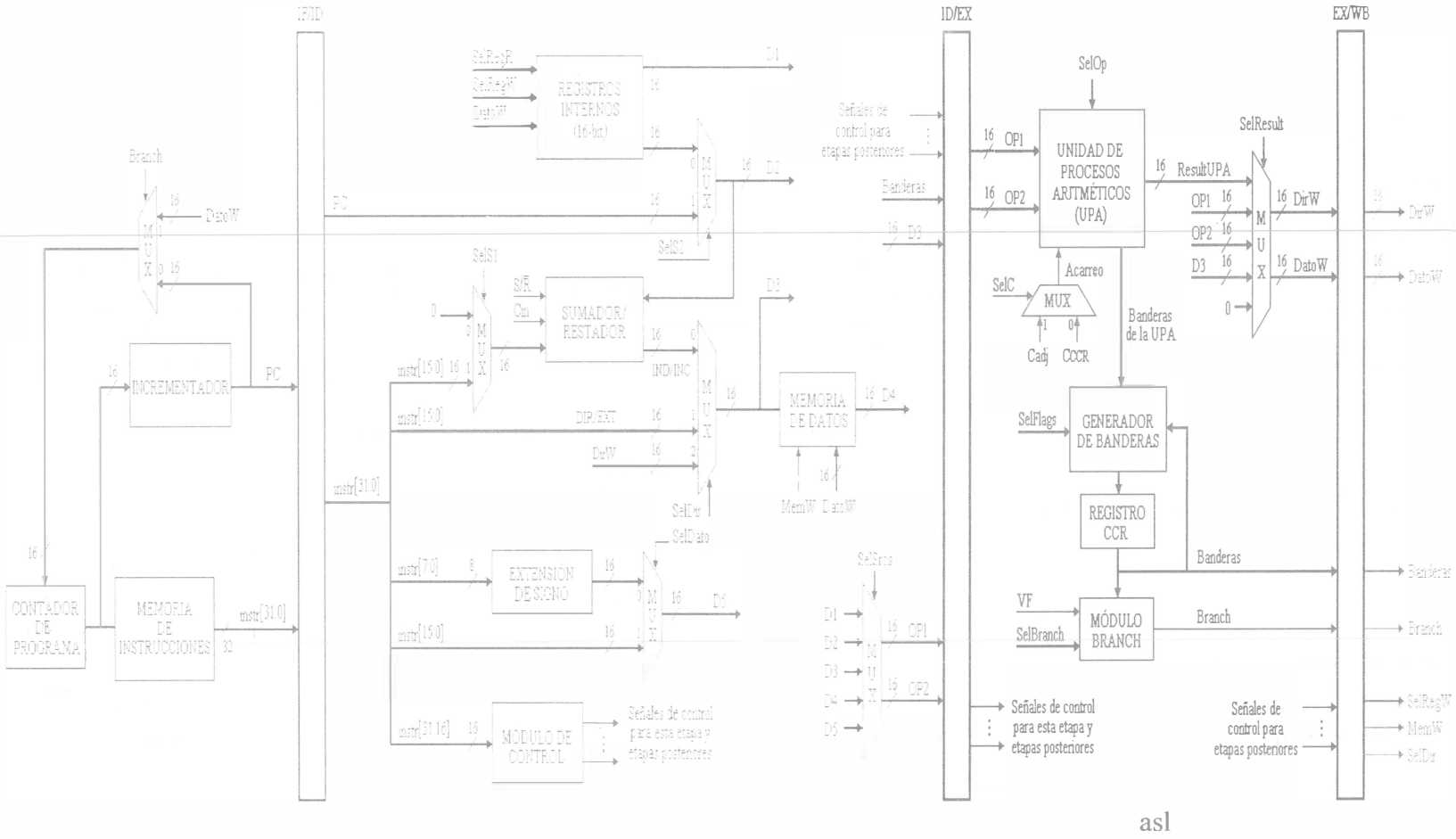
La operación de corrimiento a la izquierda y el cálculo de las banderas, tal y como lo establece la especificación de la instrucción, se realizan en la UPA con SelOp=6. El resultado de esta operación junto con la dirección efectiva (contenida en el campo D3 del registro ID/EX) son seleccionadas por medio de la señal SelResult=1, y guardadas en el registro de segmentación EX/WB en el flanco de subida del reloj. Las señales de control para la última etapa y algunas banderas también son guardadas en el registro EX/WB.

Por otra parte, el módulo generador de banderas junto con la señal de control SelFlags, actualizan las banderas calculadas por la UPA en el registro CCR. Y el módulo Branch junto con las señales SelBranch y VF, determinan si se ejecuta un salto o no (véase la figura 7.32).

Etapa 4. Post-escritura

En esta última etapa se guarda el resultado de la UPA en memoria. La localidad de memoria en donde se almacenará el resultado es la misma de donde fue leído el dato, es decir, en la dirección dada por IY + desplazamiento, la cual está guardada en el campo DirW del registro de segmentación EX/WB.

Para guardar un dato en memoria se necesita activar la señal MemW, la cual nos permite realizar una operación de escritura en ella. También es necesario colocar la señal SelDir a 2, pues así se seleccionará el bus DirW con la dirección de la localidad de memoria en donde se guardará el resultado (DatoW). Observe que para esta instrucción no se actualiza ningún registro interno, por eso, SelRegW es colocado a cero (véase la figura 7.33).



asl

Figura 7.32. En el tercer ciclo de reloj la instrucción ASL se encuentra en la etapa de ejecución. Durante esta etapa se calcula el corrimiento del dato leído de memoria en la etapa anterior, se actualizan los valores de las banderas afectadas y se evalúa la condición de salto.

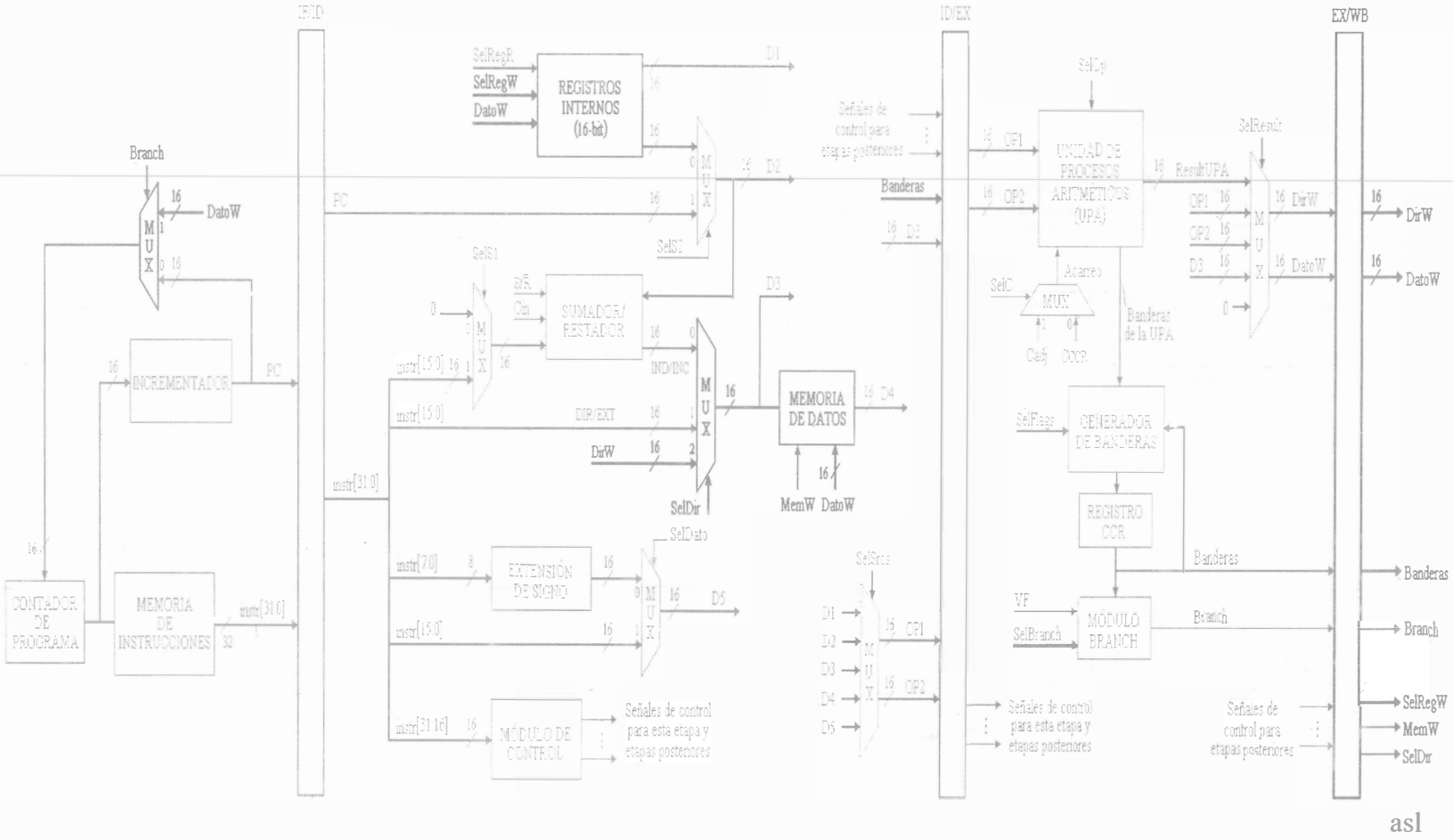


Figura 7.33. En el cuarto ciclo de reloj la instrucción ASL se encuentra en la etapa de post-escritura. En esta última etapa, el resultado del corrimiento, calculado en la etapa anterior y contenido en el bus DatoW, es guardado en la localidad de memoria dada por DirW.

7.3.5 INSTRUCCIÓN STAA (Acceso Extendido)

Instrucción: STAA Dirección_16Bits
Operación: (Memoria) ⇐ (ACCA)
Código: 00B7
Descripción: Almacena el contenido del registro ACCA en memoria. El contenido de ACCA permanece sin cambios.
Banderas: N=1 si el MSB del resultado está encendido, N=0 en caso contrario.
Z=1 si el resultado en el registro es cero, Z=0 en caso contrario.
V se coloca a cero.

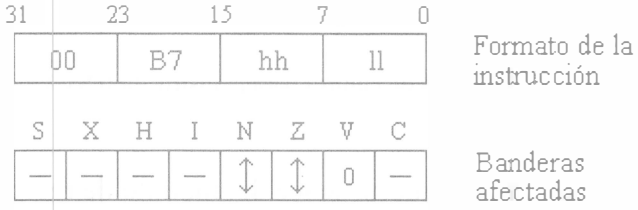


Figura 7.34. Formato de la instrucción STAA y banderas que afecta.

El comportamiento de la instrucción *staa* es el siguiente.

Etapa 1. Lectura de la instrucción

En esta etapa se lee de la memoria la instrucción a ejecutar, es decir, la instrucción *staa*. La dirección de esta instrucción, dada por el contador de programa, se incrementa y se vuelve a cargar en el contador de programa. Este nuevo valor corresponde a la dirección en memoria de la próxima instrucción a ejecutar. La dirección incrementada y la instrucción se guardan en el registro de segmentación IF/ID para su uso posterior en la etapa 2 (véase la figura 7.35).

Etapa 2. Decodificación / Cálculo de la dirección efectiva / Lectura de operandos

En esta instrucción el modo de direccionamiento es extendido. La dirección de la localidad de memoria, en donde se guardará el contenido del registro ACCA, se obtiene directamente del formato de la instrucción. Las señales de control para esta etapa y las etapas posteriores se muestran en la siguiente tabla.

Etapa en la que se utiliza la señal	*Señales de control*	*Valor de la señal*
Etapa 2	SelRegR	4
	SelS1	1
	S/R	1
	Cin	0

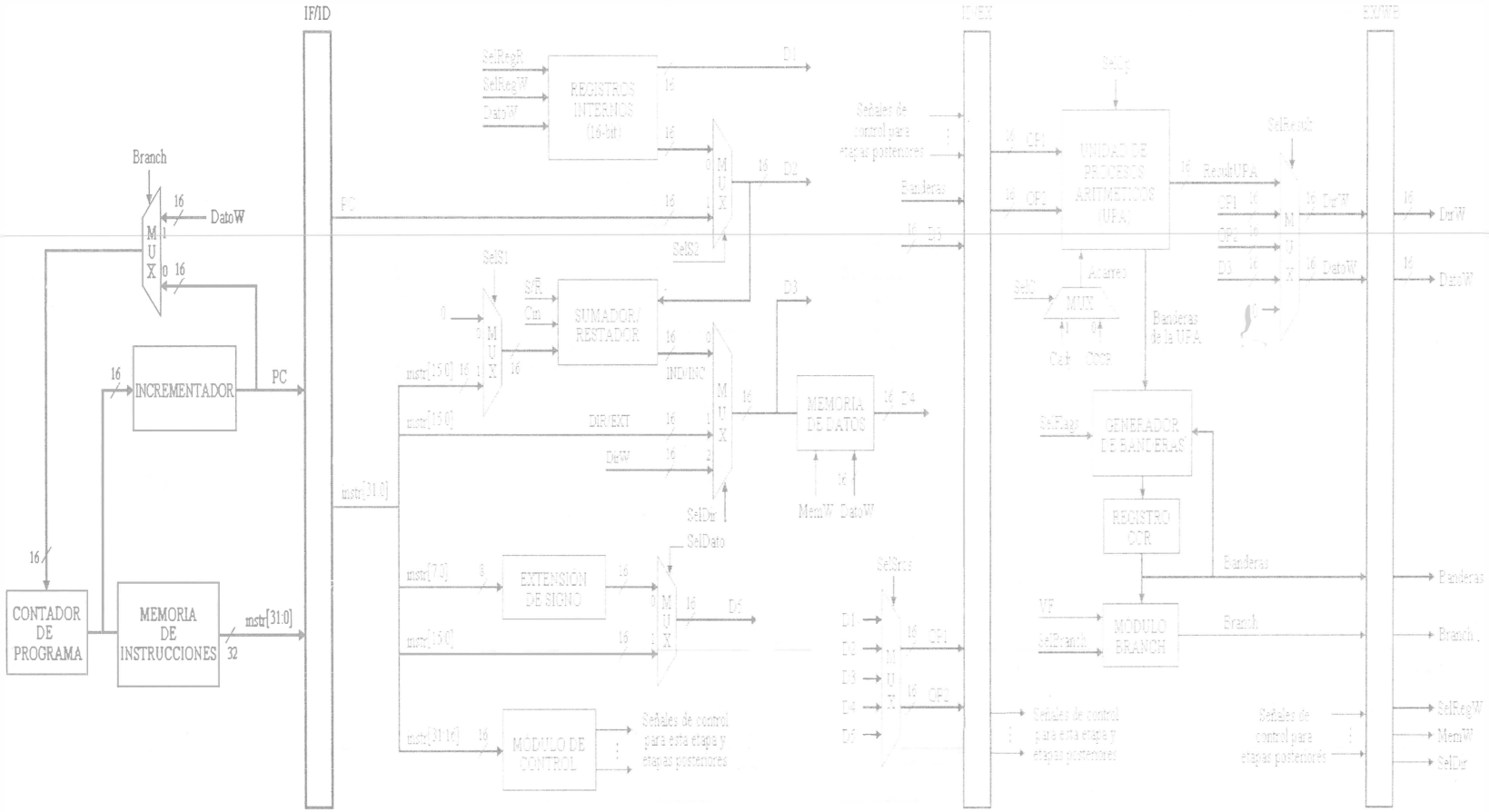
	SelS2	0
	SelDato	1
	SelScrs	1
	SelDir	0
Etapa 3	SelOp	4
	SelResult	1
	SelC	1
	Cadj	0
	SelFlags	1
	SelBranch	0
	VF	1
Etapa 4	SelRegW	0
	MemW	1
	SelDir	2

Tabla 7.13. Señales de control para la instrucción STAA.

El dato que deseamos guardar en la memoria se lee del módulo de registros internos por medio de la señal SelRegR=4. La dirección efectiva en donde se guardará el contenido del registro ACCA se pasa por el módulo sumador/restador, quien suma al dato seleccionado con SelS1=1 el dato seleccionado con SelS2=0, es decir, se suma a la dirección de 16 bits un cero. El resultado final, la misma dirección de 16 bits, es seleccionada por medio de la señal SelDir=0.

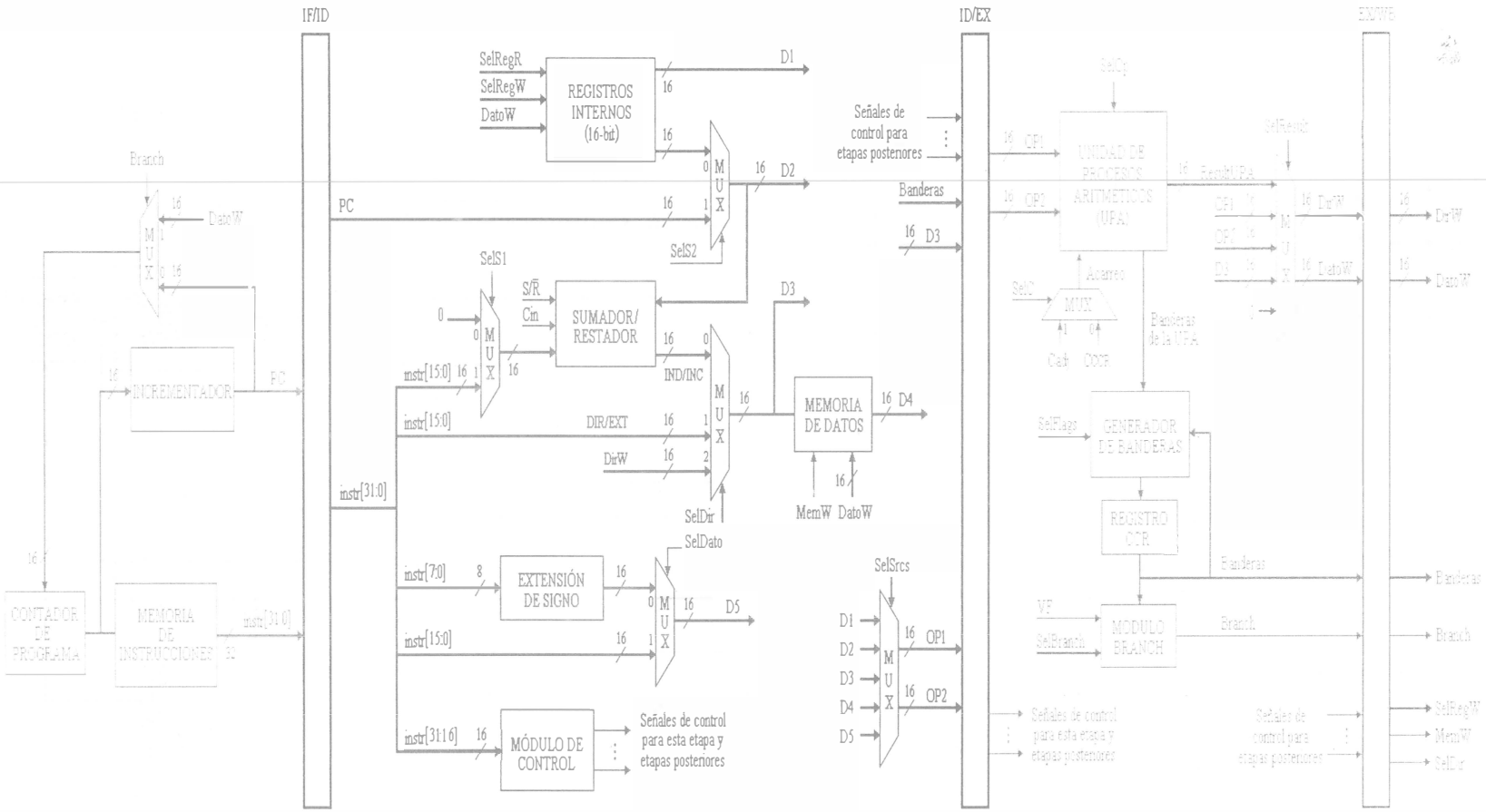
De esta manera, en el bus D1 se tendrá el contenido del registro ACCA, en el bus D2 un cero, y en el bus D3 la dirección efectiva de 16 bits. Todos estos datos son guardados en el registro de segmentación ID/EX mediante la señal SelSrcs=1. Recuerde que también se guardan en el registro de segmentación ID/EX las señales de control para las etapas siguientes.

Note que para la instrucción *staa* el resultado seleccionado por la señal SelDato carece de importancia (véase la figura 7.36).



staa

Figura 7.35. En esta primera etapa, correspondiente al primer ciclo de reloj, la instrucción STAA es leída de la memoria de instrucciones.



staa

Figura 7.36. En el segundo ciclo de reloj la instrucción STAA se encuentra en la etapa de decodificación. Durante esta etapa se lee el contenido del registro ACCA, se extrae la dirección efectiva en donde se almacenará dicho contenido, y se generan las señales de control para las etapas siguientes.

Etapa 3. Ejecución / Cálculo de banderas y saltos

El dato contenido en el campo OP1 del registro de segmentación ID/EX corresponde al contenido del registro ACCA, y el dato contenido en el campo OP2 corresponde al valor de cero. La señal SelOp=4 opera los contenidos de estos dos campos utilizando la operación lógica OR, de manera que el dato en OP1 no es modificado. A simple vista esta operación puede parecer innecesaria, sin embargo, sí tiene una razón de ser, pues calcula los valores de las banderas especificados por la instrucción.

Las banderas afectadas son actualizadas en el registro CCR según la señal SelFlags; y como esta instrucción no es una instrucción de salto, entonces, colocamos las señales SelBranch a cero y VF a uno para que el módulo Branch evite generar una señal de salto.

La señal SelResult=1 selecciona el resultado de la operación lógica OR y lo guarda en el registro de segmentación EX/WB junto con la dirección efectiva (contenida en el campo D3 del registro de segmentación ID/EX). También son guardadas en el registro EX/WB algunas banderas y las señales de control necesarias para la última etapa (véase la figura 7.37).

Etapa 4. Post-escritura

En esta última etapa se guarda el contenido del registro ACCA en memoria. El contenido de dicho registro está almacenado en el campo DatoW del registro de segmentación EX/WB, y la localidad de memoria en donde es almacenado el dato también está guardada en el registro EX/WB, pero en el campo DirW.

Finalmente, para guardar el dato en memoria se activa la señal MemW, la cual permite realizar una operación de escritura en la memoria. También es necesario colocar la señal SelDir a 2, pues con ello se seleccionará el bus DirW con la dirección de la localidad de memoria en donde se guardará el contenido del registro ACCA. Observe que para esta instrucción no se actualiza ningún registro interno, por eso la señal SelRegW se coloca a cero (véase la figura 7.38).

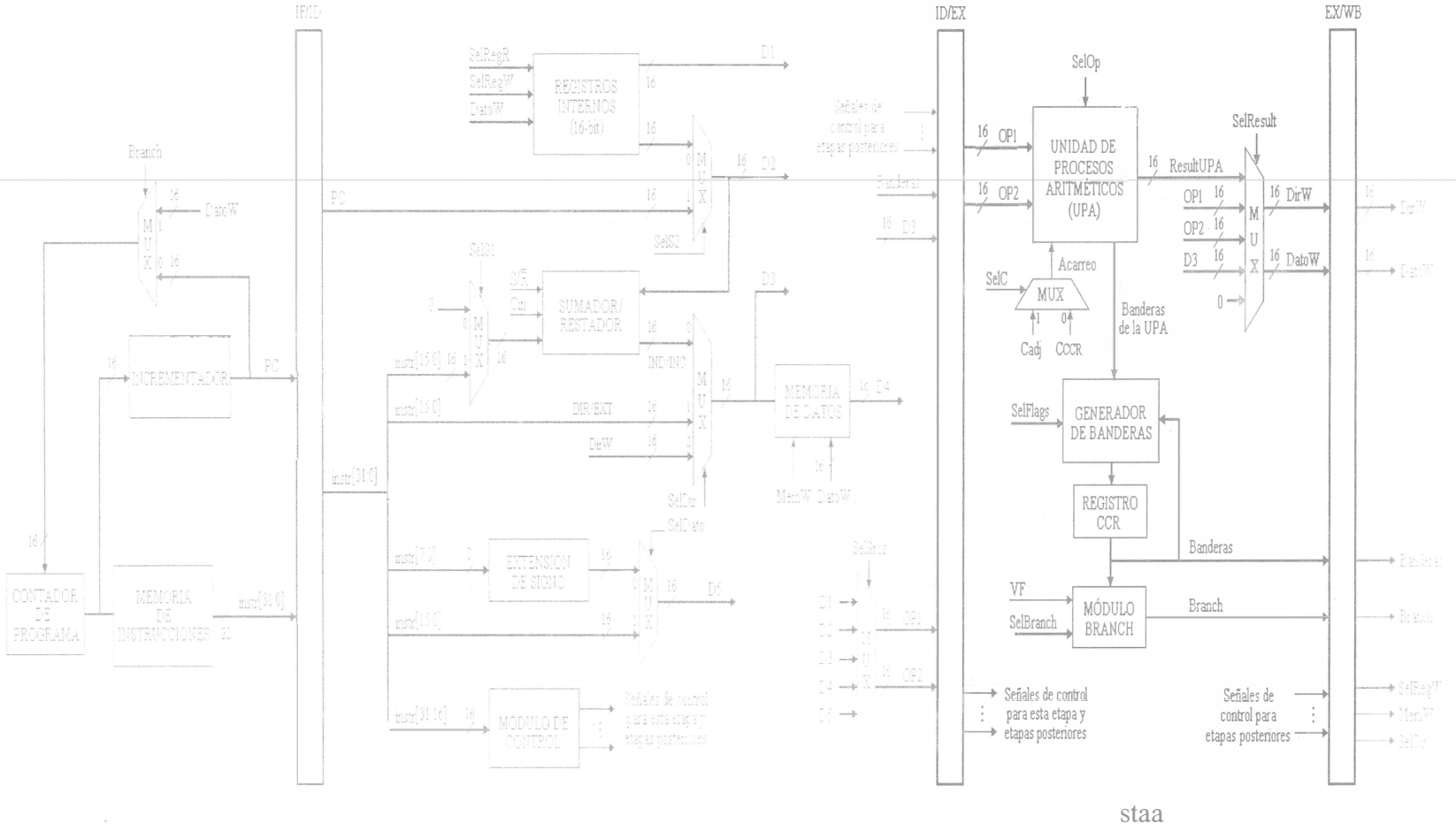


Figura 7.37. En el tercer ciclo de reloj la instrucción STAA se encuentra en la etapa de ejecución. Durante esta etapa se actualizan los valores de las banderas afectadas y se evalúa la condición de salto.

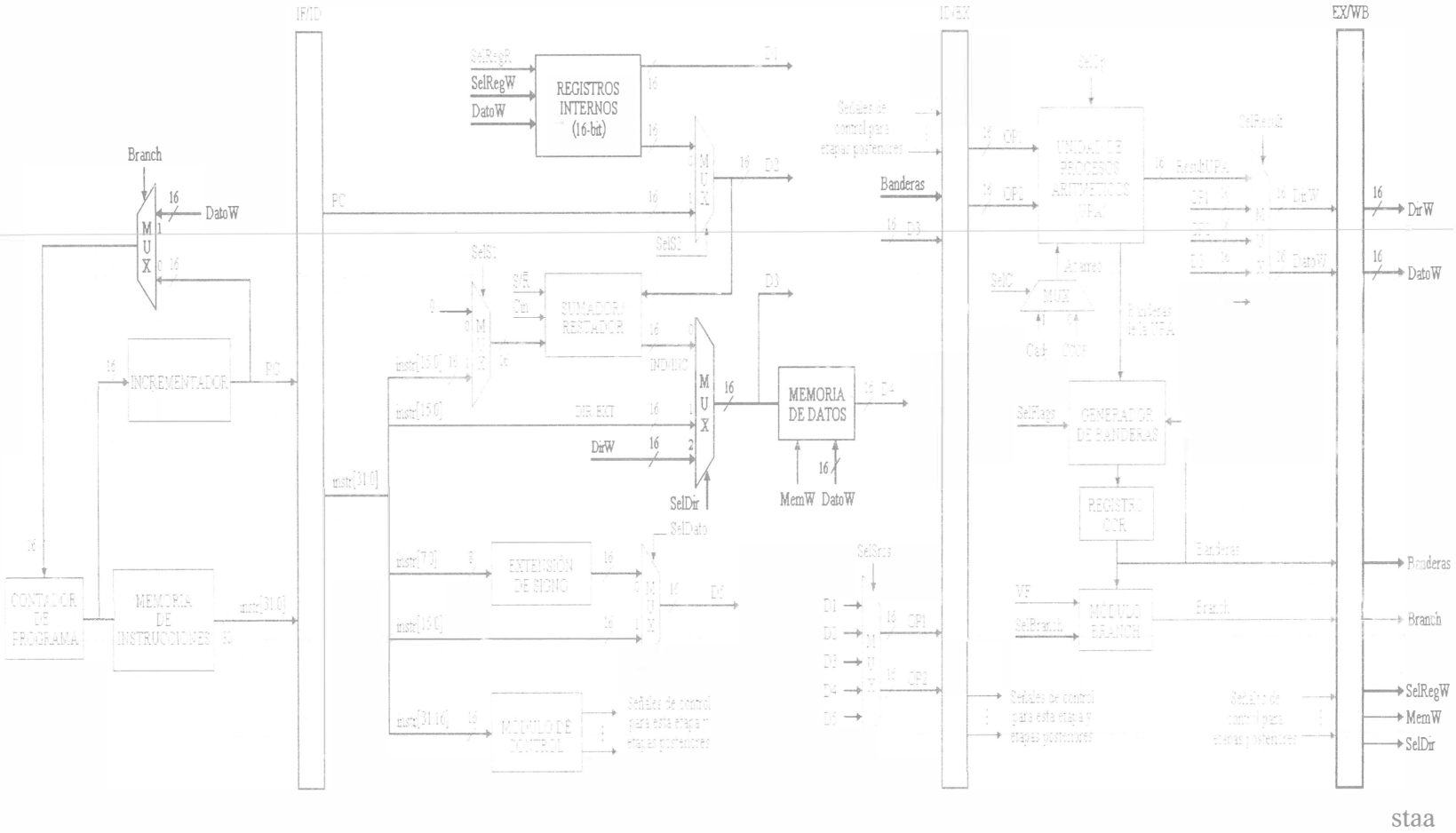


Figura 7.38. En el cuarto ciclo de reloj la instrucción STAA se encuentra en la etapa de post-escritura. En esta última etapa, el contenido del registro ACCA presente en el bus DatoW, es guardado en la localidad de memoria dada por DirW.

7.3.6 INSTRUCCIÓN BRA (Acceso Relativo)

Instrucción: BRA Desplazamiento
Operación: PC ⇐ (PC) + 1 + Desplazamiento
Código: 0020
Descripción: Salto incondicional a la dirección: PC + 1 + Desplazamiento; donde el desplazamiento es un número de 8 bits en complemento a dos.
Banderas: Ninguna bandera es afectada.

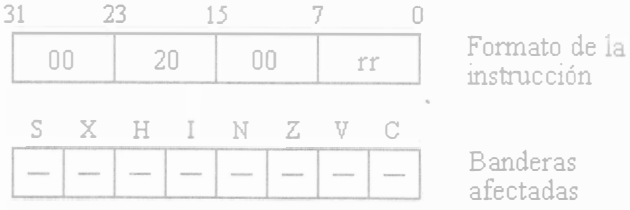


Figura 7.39. Formato de la instrucción BRA y banderas que afecta.

El comportamiento de la instrucción *bra* es el siguiente.

Etapa 1. Lectura de la instrucción

En esta etapa se lee de la memoria la instrucción a ejecutar, es decir, la instrucción *bra*. La dirección de esta instrucción, dada por el contador de programa, se incrementa y se vuelve a cargar en el contador de programa. Este nuevo valor corresponde a la dirección en memoria de la próxima instrucción a ejecutar. La dirección incrementada y la instrucción se guardan en el registro de segmentación IF/ID para su uso posterior en la etapa 2 (véase la figura 7.40).

Etapa 2. Decodificación / Cálculo de la dirección efectiva / Lectura de operandos

La dirección de salto se obtiene sumando al contenido del registro PC el valor del desplazamiento. Sin embargo, antes de calcular esta dirección de salto es necesario que PC apunte a la dirección de la siguiente instrucción en memoria, es decir, a PC + 1. Una vez que PC apunta a la dirección correcta podremos sumarle el desplazamiento. Quizá ahora resulta más claro el por qué de guardar en el registro de segmentación IF/ID el PC incrementado. Las señales de control para esta etapa y las etapas posteriores se muestran en la siguiente tabla.

Etapa en la que se utiliza la señal	*Señales de control*	*Valor de la señal*
Etapa 2	SelRegR	0
	SelS1	0
	S/$\overline{R}$	1
	Cin	0

	SelS2	1
	SelDato	0
	SelScrs	5
	SelDir	0
Etapa 3	SelOp	1
	SelResult	1
	SelC	1
	Cadj	0
	SelFlags	0
	SelBranch	0
	VF	0
Etapa 4	SelRegW	0
	MemW	0
	SelDir	0

Tabla 7.14. Señales de control para la instrucción BRA.

Como se mencionó anteriormente, para calcular la dirección de salto se necesitan sumar el valor del PC incrementado y el valor del desplazamiento. Dicha suma será ejecutada por la unidad de procesos aritméticos hasta la siguiente etapa, por lo tanto, la etapa de decodificación deberá enviar a la UPA los operandos adecuados.

El PC incrementado se obtiene del campo PC del registro IF/ID y el desplazamiento se extrae directamente del formato de la instrucción. El PC incrementado es seleccionado con la señal SelS2=1 y asignado al bus D2, mientras que el desplazamiento es seleccionado con SelDato=0 y asignado al bus D5. Antes de que el desplazamiento sea asignado al bus D5, es necesaria su extensión de 8 bits a 16 bits (recuerde que la UPA sólo efectúa operaciones de 16 bits), para ello, el módulo de extensión de signo repite en los 8 bits más significativos del nuevo desplazamiento el bit de signo del desplazamiento original.

El PC incrementado y el desplazamiento extendido son seleccionados con SelScrs=5 y guardados en el registro de segmentación ID/EX junto con las señales de control para las etapas posteriores. Note que para esta instrucción no se leen datos de registros internos ni de la memoria de datos (véase la figura 7.41).

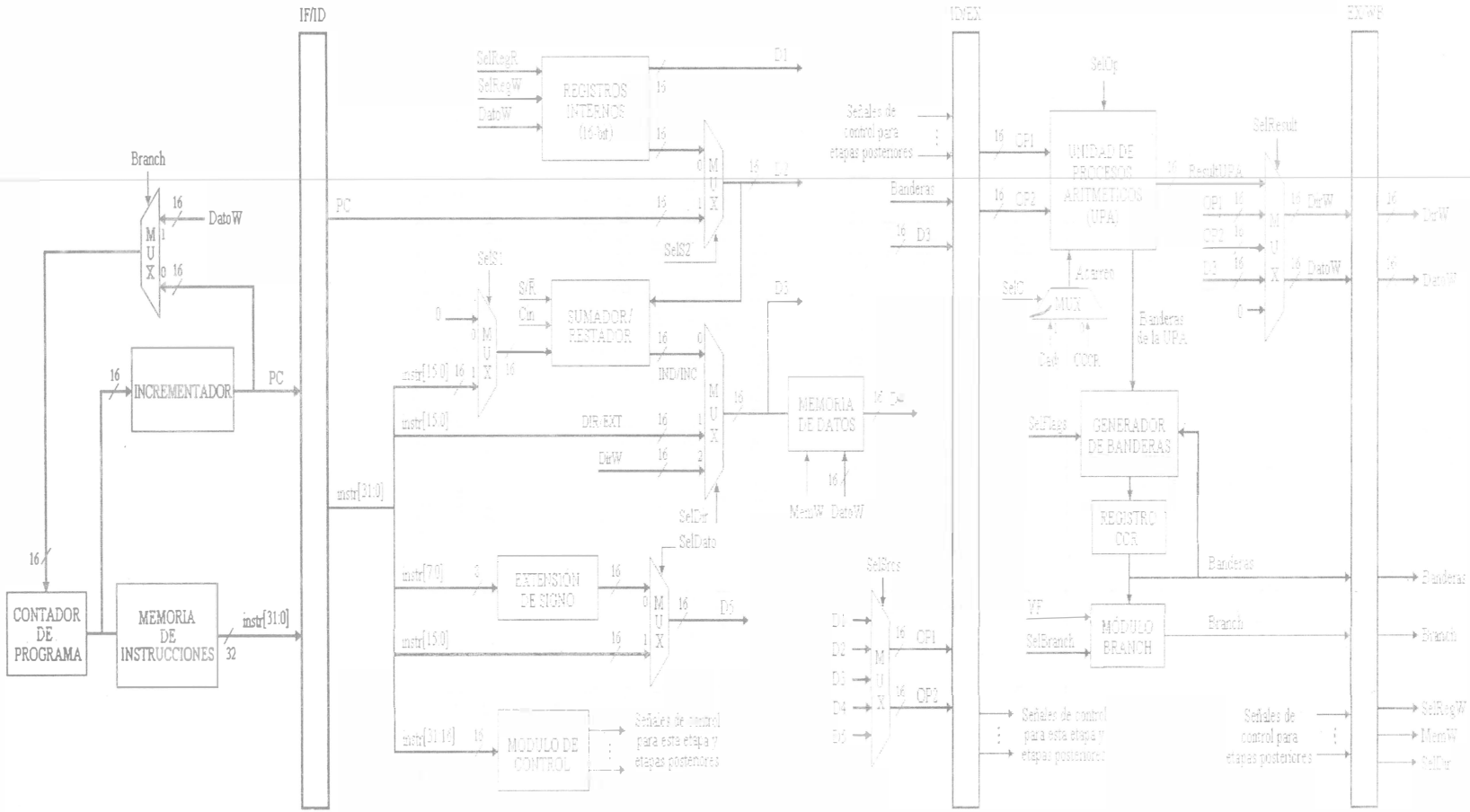
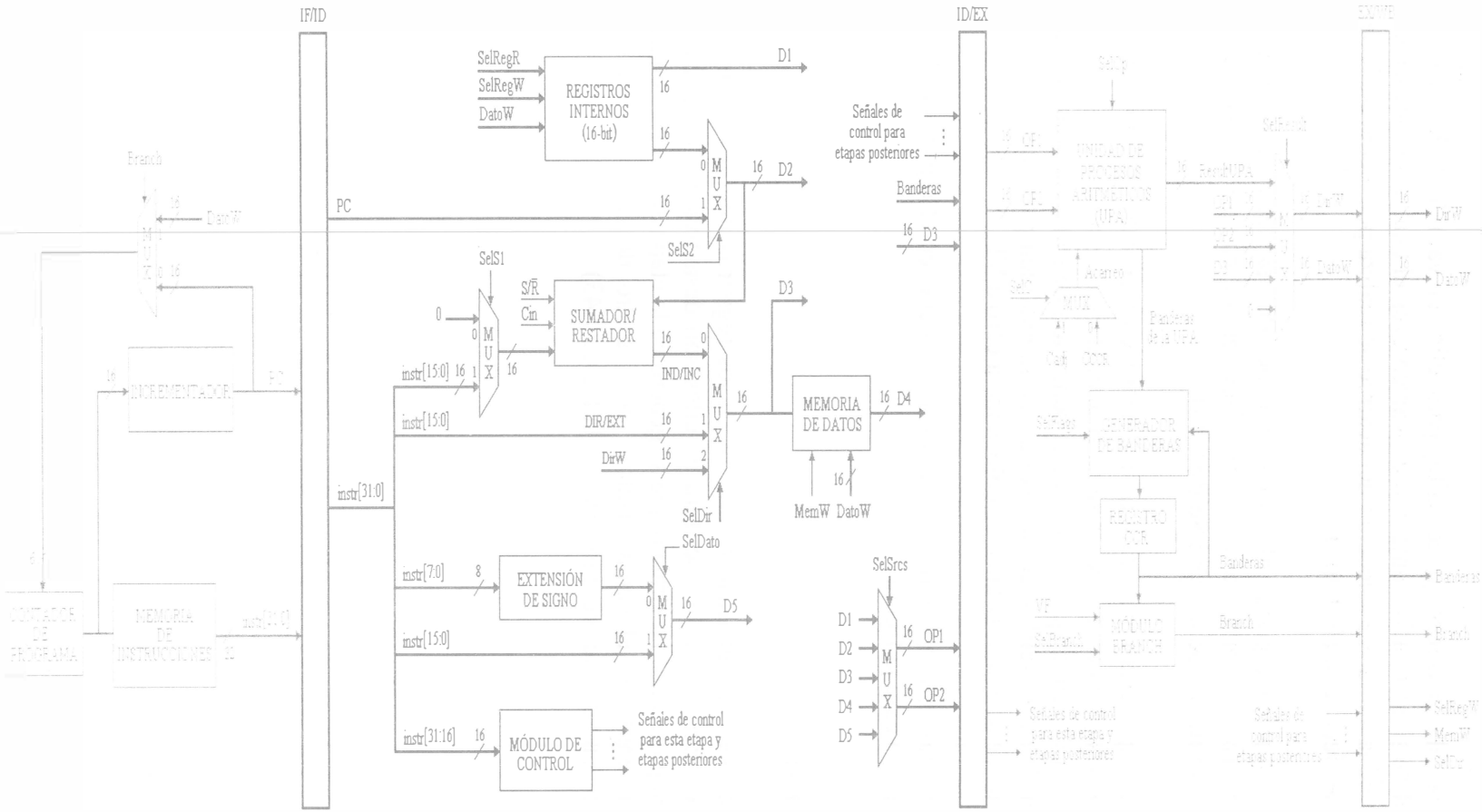


Figura 7.40. En esta primera etapa, correspondiente al primer ciclo de reloj, la instrucción BRA es leída de la memoria de instrucciones.



bra

Figura 7.41. En el segundo ciclo de reloj la instrucción BRA se encuentra en la etapa de decodificación. Durante esta etapa se extiende el desplazamiento de 8 bits con signo a 16 bits, se obtiene el valor del PC incrementado y se generan las señales de control para las etapas siguientes.

Etapa 3. Ejecución / Cálculo de banderas y saltos

En esta etapa se calcula la dirección de salto, para ello, el valor del PC incrementado y el desplazamiento extendido son sumados junto con el acarreo Cadj, el cual se coloca a cero para evitar modificar el resultado de la suma. La selección del acarreo se realiza por medio de la señal SelC=1, mientras que la suma se ejecuta con SelOp=1.

El resultado de la suma corresponde al nuevo valor del PC, es decir, la dirección a donde el programa debe saltar, así que este valor es guardado en el campo DatoW del registro de segmentación EX/WB por medio de la señal SelResult=1.

Por otra parte, se sabe que esta instrucción es una instrucción de salto incondicional, es decir, se debe ejecutar el salto, o visto de otra forma, la señal Branch debe colocarse a uno. Para obligar a que Branch valga uno hay que asegurar que el valor de la condición de salto sea igual al valor de VF. Si se utiliza la condición de salto SelBranch=0, esto es, se intenta comparar contra el valor de cero, entonces, la única forma de activar a Branch es colocando a cero a VF.

Finalmente, la instrucción *bra* no modifica valores de banderas en el registro CCR, por lo tanto, SelFlags se coloca a cero (véase la figura 7.42).

Etapa 4. Post-escritura

Las instrucciones de salto no guardan resultados en los registros internos ni en la memoria de datos, por lo tanto, las operaciones de escritura en ellos son desactivadas. La única señal que se utiliza en esta cuarta etapa es la señal Branch que se generó en la etapa anterior, la cual selecciona la dirección de salto contenida en el campo DatoW del registro de segmentación EX/WB, para guardarla en el registro contador de programa (véase la figura 7.43).

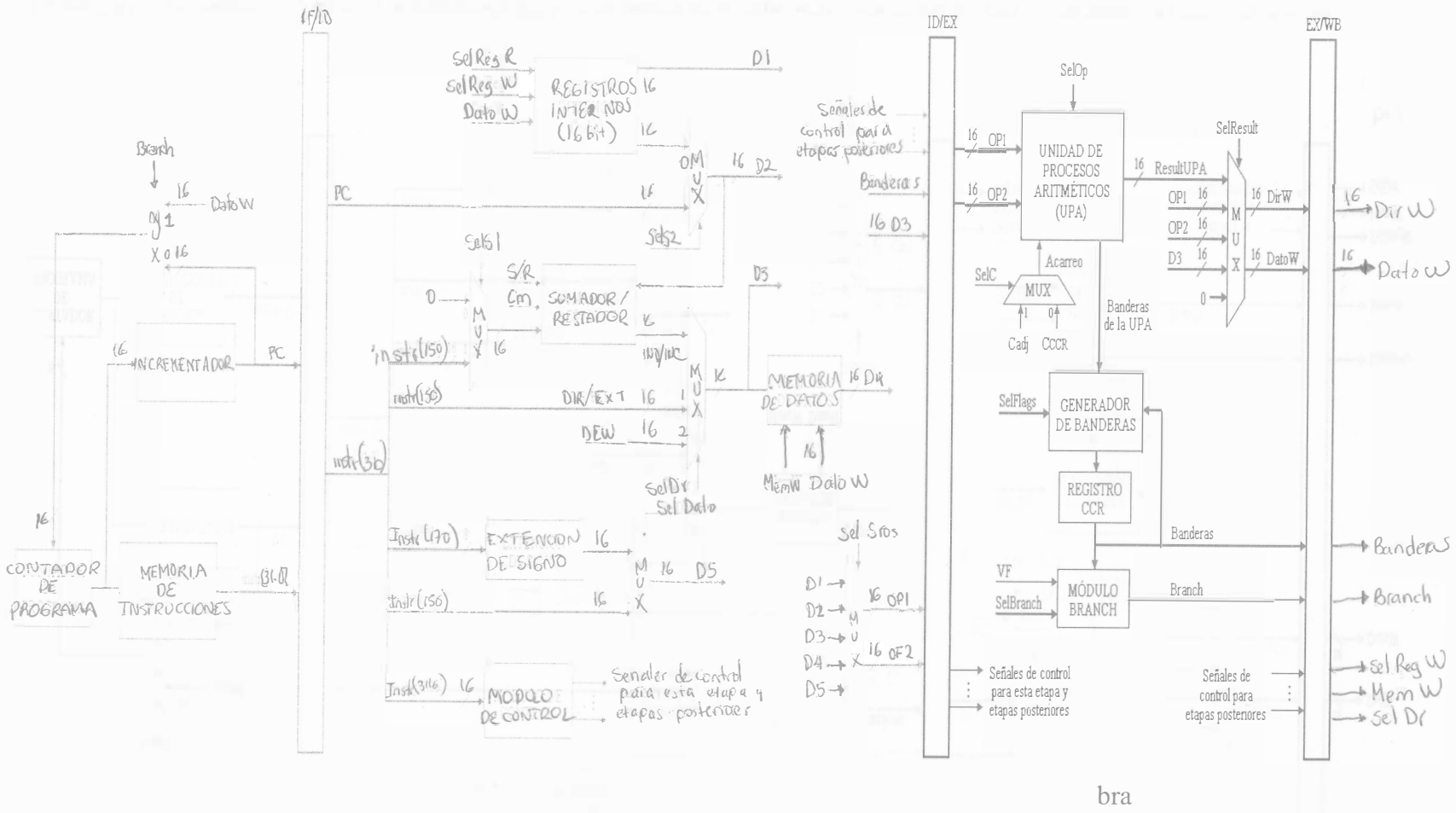


Figura 7.42. En el tercer ciclo de reloj la instrucción BRA se encuentra en la etapa de ejecución. Durante esta etapa la UPA calcula la dirección de salto y el módulo Branch evalúa la condición de salto.

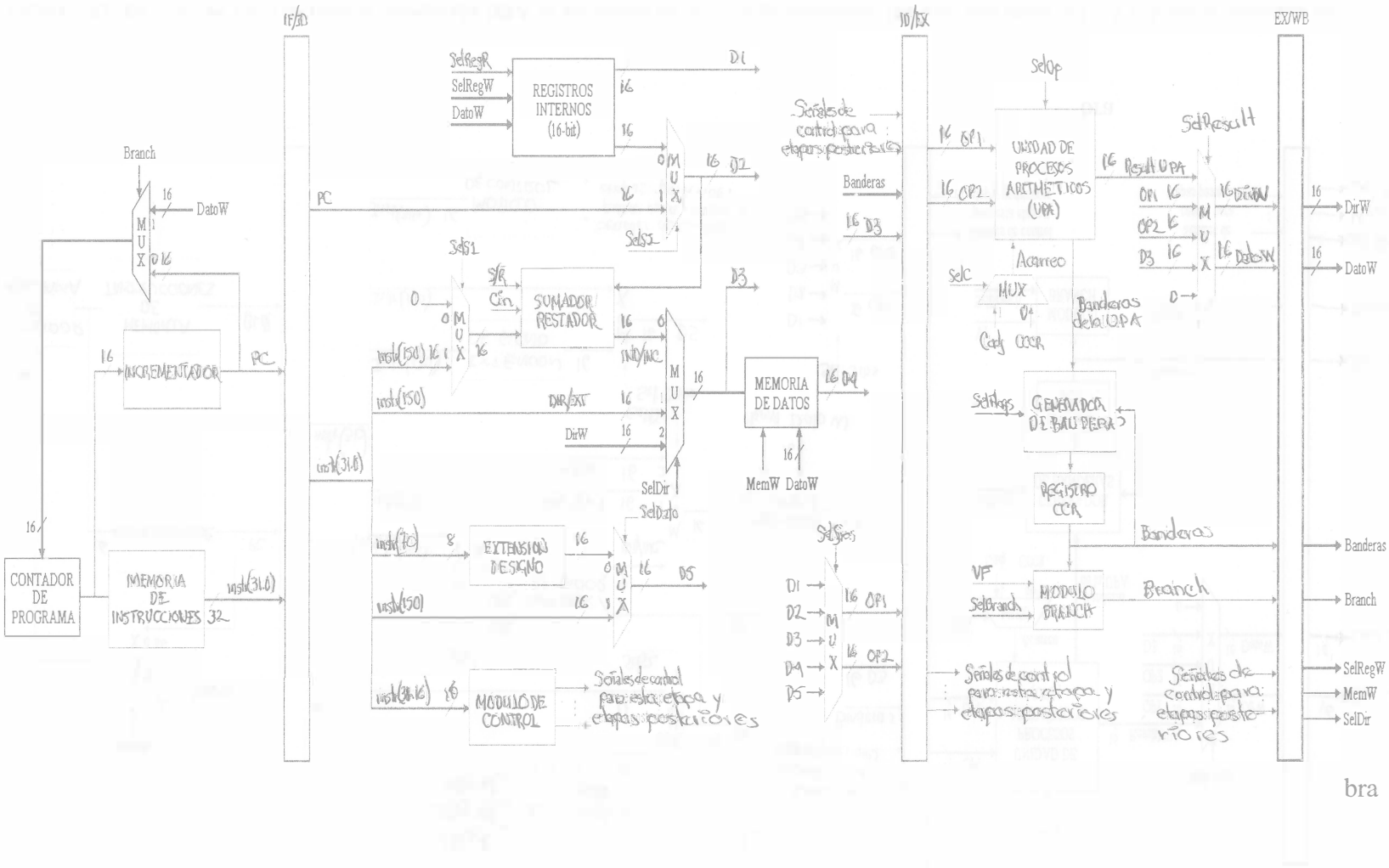


Figura 7.43. En el cuarto ciclo de reloj la instrucción BRA se encuentra en la etapa de post-escritura. En esta última etapa, la dirección de salto, contenida en el bus DatoW, es guardada en el registro contador de programa.

7.3.7 RESUMEN DE INSTRUCCIONES

A continuación se presenta una tabla con las señales de control para las instrucciones vistas anteriormente y para otras instrucciones que tienen soporte en esta arquitectura segmentada.

Instrucciones		*Señales de Control*																		
		Etapa 2								Etapa 3							Etapa 4		
Mnemónico	Instr[31:16]	SelRegR	SelS1	S/$\overline{R}$	Cin	SelS2	SelDato	SelScrs	SelDir	SelOp	SelResult	SelC	Cadj	SelFlags	SelBranch	VF	SelRegW	MemW	SelDir
aba (INH)	001B	1	0	1	0	0	1	1	0	1	1	1	0	2	0	1	1	0	0
aby (INH)	183A	3	0	1	0	0	1	1	0	1	1	1	0	0	0	1	3	0	0
adcb (DIR)	00D9	5	0	1	0	0	1	2	1	1	1	0	0	2	0	1	4	0	0
anda (EXT)	00B4	4	0	1	0	0	1	2	1	3	1	1	0	1	0	1	1	0	0
andb (EXT)	00F4	5	0	1	0	0	1	2	1	3	1	1	0	1	0	1	4	0	0
andb (IND,X)	00E4	2	1	1	0	0	1	2	0	3	1	1	0	1	0	1	4	0	0
asl (IND,Y)	1868	A	1	1	0	0	1	4	0	6	1	1	0	3	0	1	0	1	2
asr (IND,X)	0067	9	1	1	0	0	1	4	0	7	1	1	0	3	0	1	0	1	2
asrb (INH)	0057	5	0	1	0	0	1	1	0	7	1	1	0	3	0	1	4	0	0
bcc (REL)	0024	0	0	1	0	1	0	5	0	1	1	1	0	0	1	0	0	0	0
bita (IMM)	0085	4	0	1	0	0	1	3	0	3	0	1	0	1	0	1	0	0	0
bra (REL)	0020	0	0	1	0	1	0	5	0	1	1	1	0	0	0	0	0	0	0
cba (INH)	0011	1	0	1	0	0	1	1	0	2	0	1	1	3	0	1	0	0	0
clr (EXT)	007F	0	0	1	0	0	1	2	1	3	1	1	0	3	0	1	0	1	2
cmpb (DIR)	00D1	5	0	1	0	0	1	2	1	2	0	1	1	3	0	1	0	0	0
com (IND,X)	0063	9	1	1	0	0	1	2	0	2	1	1	0	B	0	1	0	1	2
comb (INH)	0053	5	0	1	0	0	1	1	0	8	1	1	0	B	0	1	4	0	0
cpy (IND,Y)	18AC	A	1	1	0	0	1	6	0	2	0	1	1	3	0	1	0	0	0
dec (EXT)	007A	0	0	1	0	0	1	2	1	8	1	1	0	C	0	1	0	1	2
des (INH)	0034	B	0	1	0	0	1	1	0	8	1	1	0	0	0	1	6	0	0
incb (INH)	005C	5	0	1	0	0	1	1	0	1	1	1	1	C	0	1	4	0	0
ldaa (IMM)	0086	0	0	1	0	0	1	3	0	4	1	1	0	1	0	1	1	0	0
ldy (IND,Y)	18EE	A	1	1	0	0	1	2	0	4	1	1	0	1	0	1	3	0	0
neg (EXT)	0070	0	0	1	0	0	1	2	1	2	1	1	1	3	0	1	0	1	2
nop (INH)	0001	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
ror (IND,Y)	1866	A	1	1	0	0	1	4	0	B	1	0	0	3	0	1	0	1	2
sbcb (IMM)	00C2	5	0	1	0	0	1	3	0	2	1	0	0	3	0	1	4	0	0
staa (EXT)	00B7	4	1	1	0	0	1	1	0	4	1	1	0	1	0	1	0	1	2
stx (IND,X)	00EF	9	1	1	0	0	1	1	0	4	1	1	0	1	0	1	0	1	2
tst (IND,Y)	186D	A	1	1	0	0	1	2	0	8	0	1	1	3	0	1	0	0	0
tsx (INH)	0030	B	0	1	0	0	1	1	0	1	1	1	1	0	0	1	2	0	0
tys (INH)	1835	A	0	1	0	0	1	1	0	8	1	1	0	0	0	1	6	0	0

Tabla 7.15. Señales de control para algunas instrucciones del 68HC11.

7.3.8 EJECUCIÓN DE MÚLTIPLES INSTRUCCIONES

Hasta el momento sólo hemos visto la ejecución de instrucciones de manera individual, por lo tanto, no hemos observado el potencial de las arquitecturas segmentadas.

El siguiente ejemplo intenta mostrar este potencial mediante la ejecución simultánea de cuatro instrucciones: ldaa, ldab, inx y staa. Tome en cuenta las siguientes condiciones iniciales: la dirección en memoria de la primera instrucción a ejecutar es 0x0400, el contenido del registro IX vale 0x2232, y la señal Branch vale cero para el primer ciclo de reloj. Además, considere que las instrucciones anteriormente ejecutadas en el cauce no almacenan resultados en registros ni en memoria.

```
0x0400    ldaa   #0080
0x0401    ldab   #4000
0x0402    inx
0x0403    staa   1000
```

El ejemplo es muy sencillo, la primera instrucción carga en el registro ACCA un dato inmediato de 16 bits en formato hexadecimal; la segunda instrucción carga en el registro ACCB otro dato inmediato; la tercera instrucción incrementa en una unidad el contenido del registro IX; y la última instrucción guarda en la dirección de memoria 0x1000 el contenido del registro ACCA. La secuencia de ejecución de estas cuatro instrucciones se presenta en el siguiente diagrama de múltiples ciclos de reloj (véase la figura 7.44).

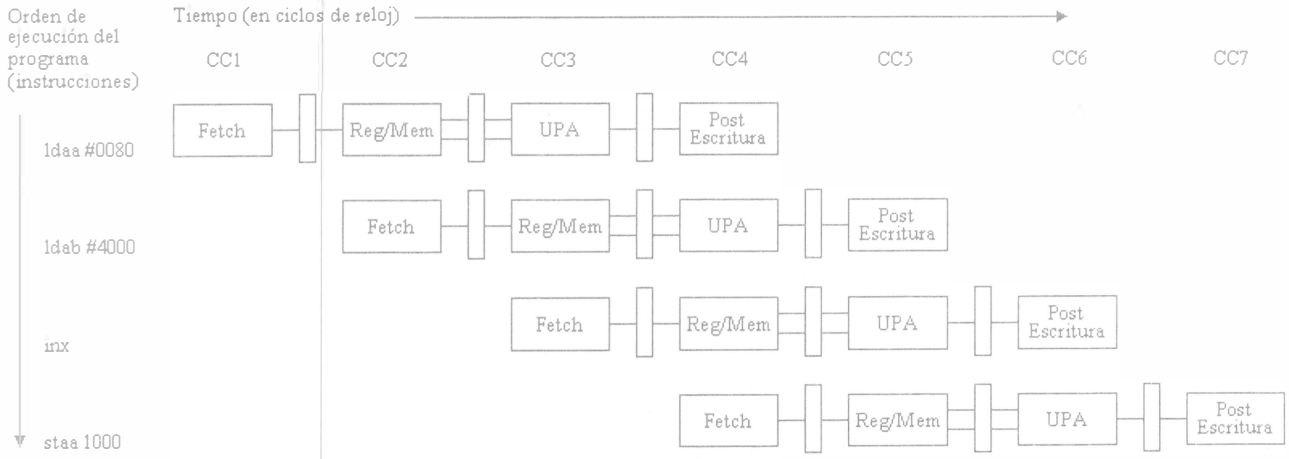


Figura 7.44. Diagrama de segmentación de múltiples ciclos de reloj para las cuatro instrucciones del ejemplo.

También se anexan los diagramas de un sólo ciclo de reloj para observar en detalle lo que ocurre en cada etapa de la segmentación (véanse las figuras 7.45 a la 7.51). En estos últimos diagramas se resaltan los componentes que intervienen en el ciclo de reloj descrito.

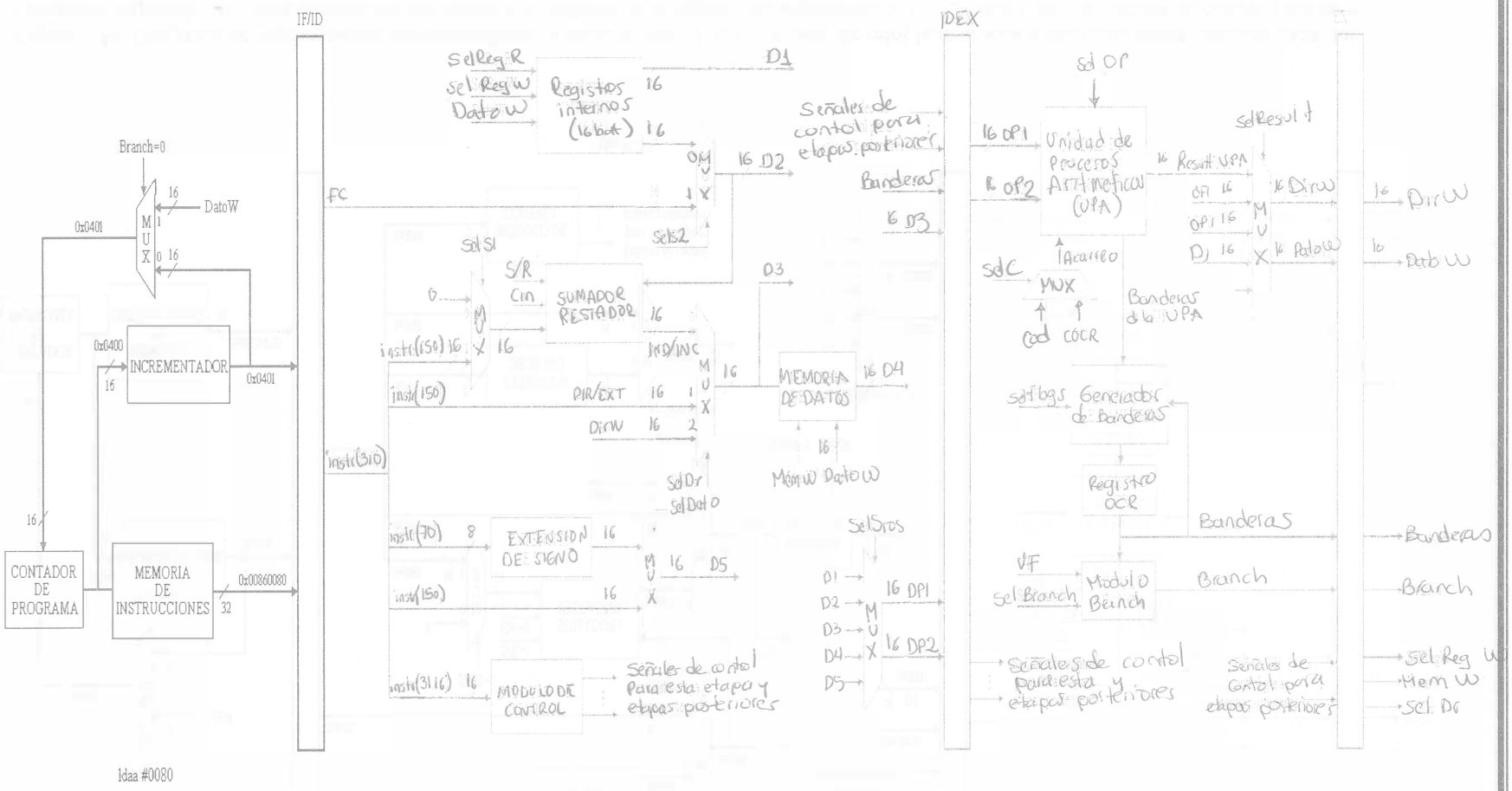


Figura 7.45. Diagrama de segmentación correspondiente al ciclo de reloj 1. En este ciclo de reloj se lee la instrucción *ldaa* de la dirección 0x0400 de la memoria de instrucciones.

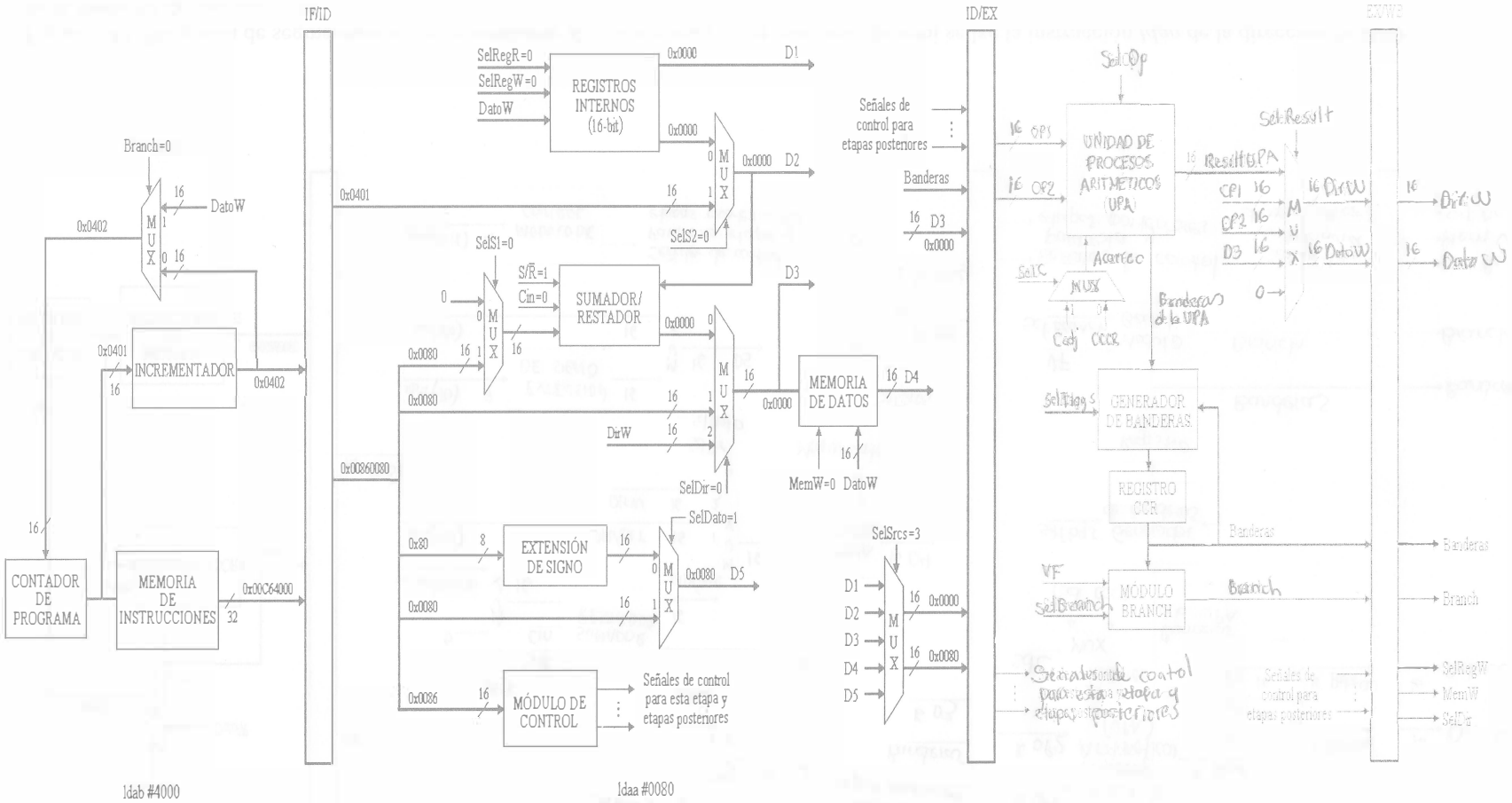


Figura 7.46. Diagrama de segmentación correspondiente al ciclo de reloj 2. En este ciclo de reloj la instrucción *ldaa* está siendo decodificada; los operandos requeridos por esta instrucción son leídos y guardados en el registro de segmentación ID/EX junto con las señales de control para las etapas posteriores. En el mismo ciclo de reloj, en la etapa 1, se lee de la memoria la instrucción *ldab*.

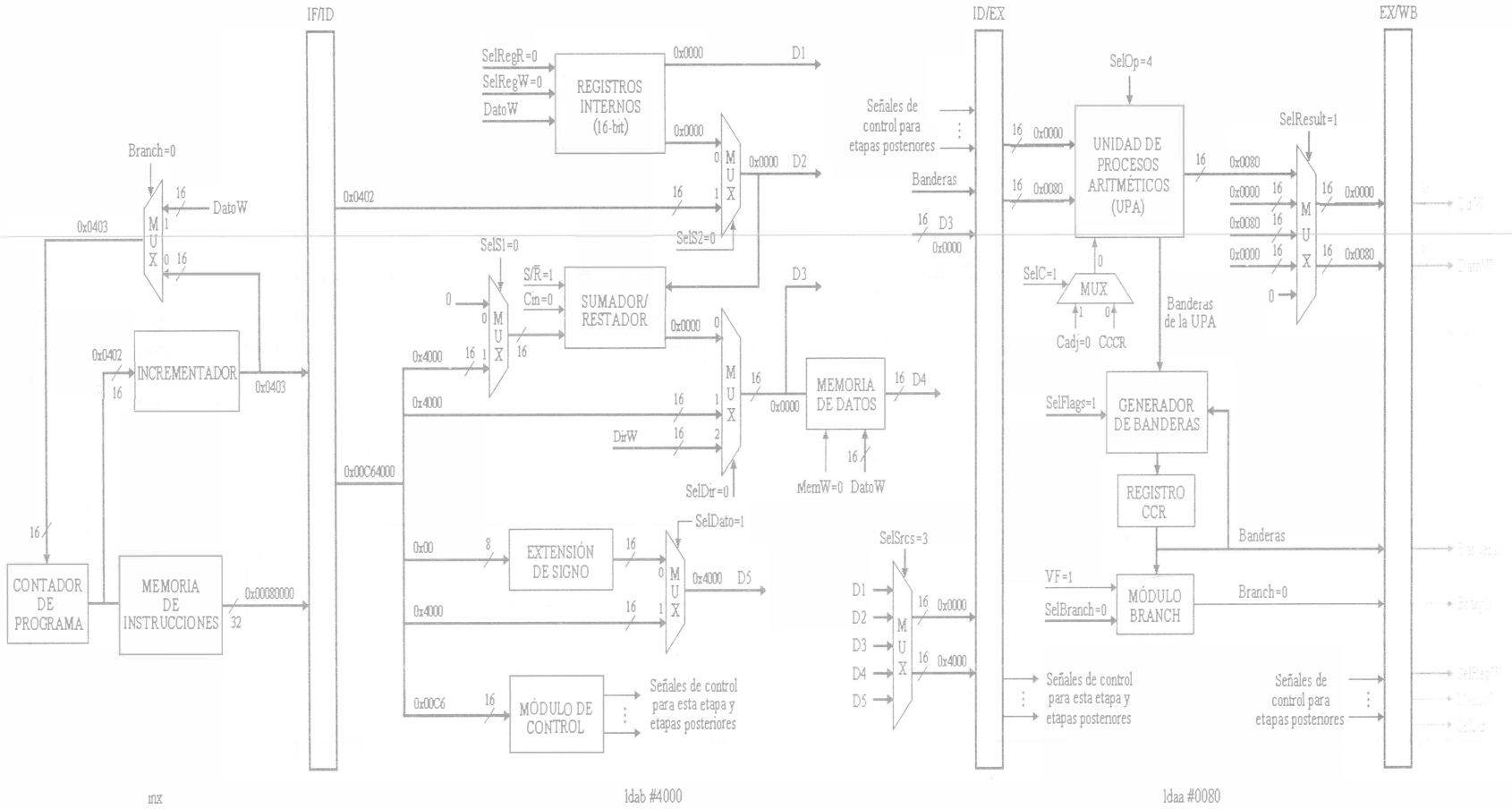


Figura 7.47. Diagrama de segmentación correspondiente al ciclo de reloj 3. La instrucción *ldaa* ahora se encuentra en la etapa 3; en esta etapa, la UPA calcula los resultados con base en los operandos y en las señales de control que le fueron proporcionados por la etapa anterior. En la etapa 2, la instrucción *ldab* es decodificada y son obtenidos los operandos que esta instrucción necesitará en las etapas siguientes. Finalmente, en la etapa 1, se lee la instrucción *inx* de la memoria de instrucciones.

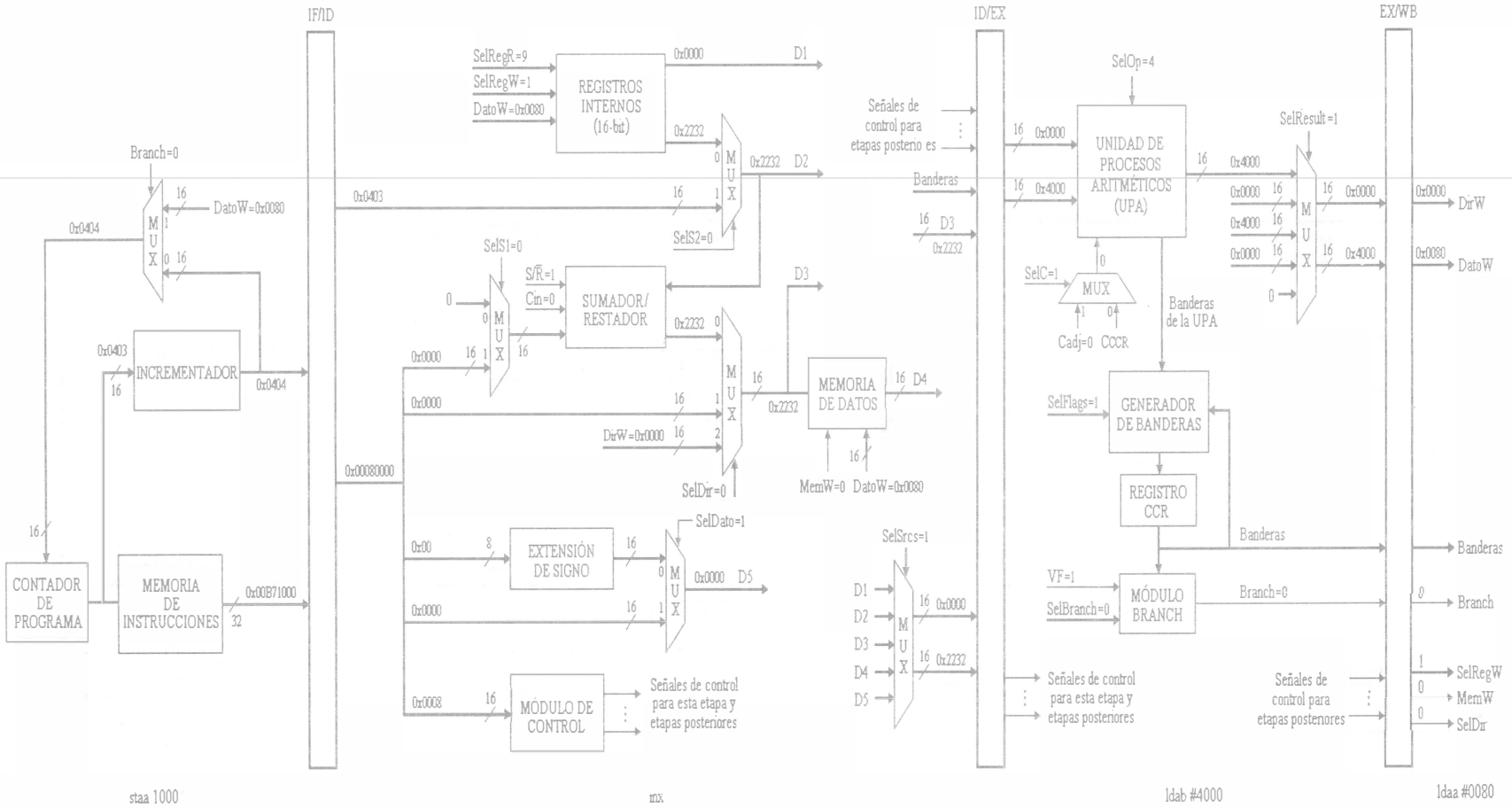


Figura 7.48. Diagrama de segmentación correspondiente al ciclo de reloj 4. En este ciclo de reloj el cauce está a toda su capacidad ejecutando una instrucción diferente en cada una de sus etapas. En la etapa de post-escritura, la instrucción *ldaa* guarda el dato inmediato 0x0080 en el registro ACCA; en la etapa de ejecución, se calculan resultados para la instrucción *ldab*; en la etapa de decodificación, se traen los operandos y las señales de control para la instrucción *inx*; y en la etapa de lectura de instrucción, se lee de la memoria de instrucciones la instrucción *staa*.

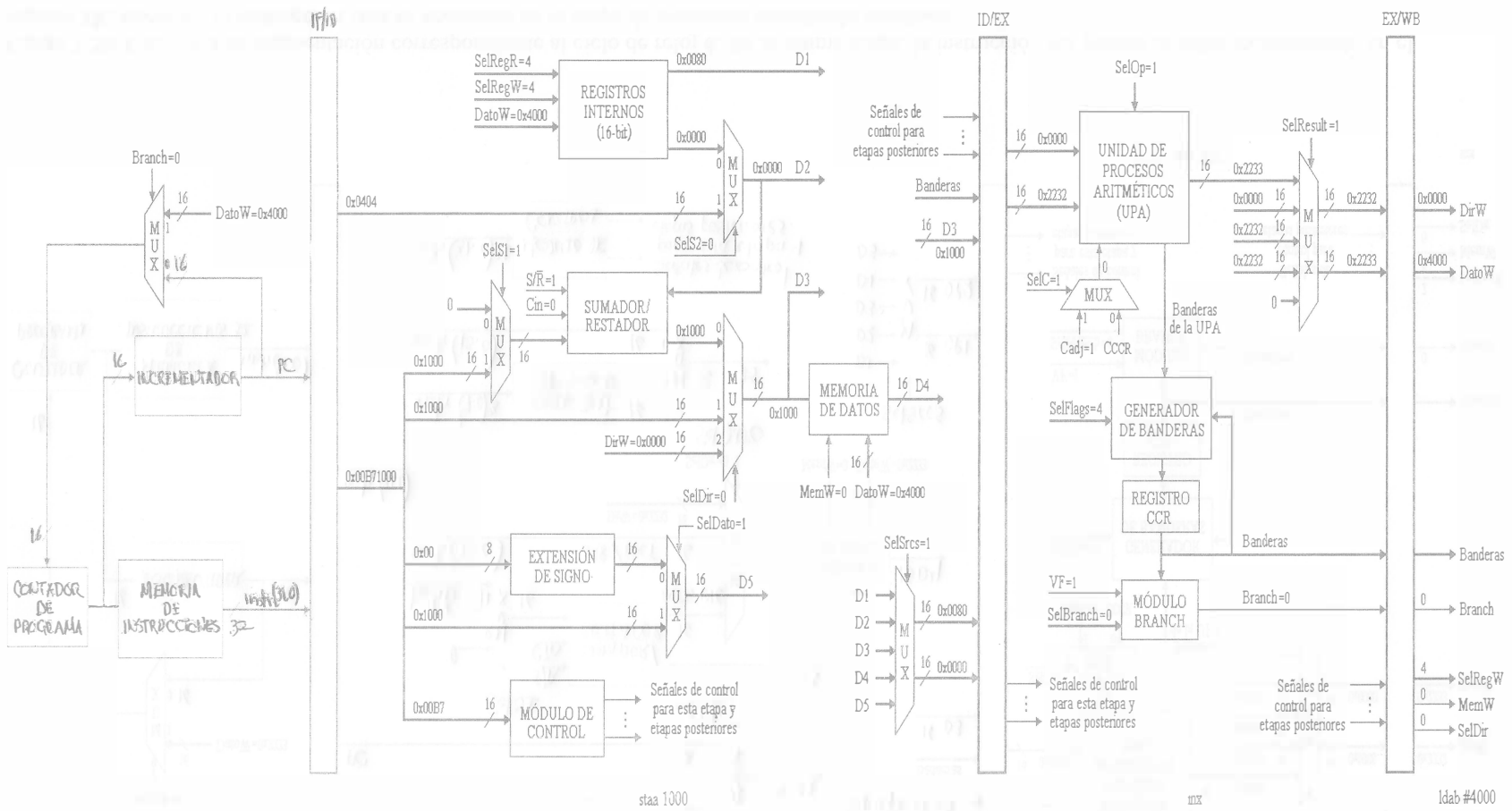


Figura 7.49. Diagrama de segmentación correspondiente al ciclo de reloj 5. En la etapa de post-escritura, la instrucción *ldab* guarda el dato inmediato 0x4000 en el registro ACCB; en la etapa de ejecución, la instrucción *inx* calcula el incremento; y en la etapa de decodificación, se traen los operandos y las señales de control para la instrucción *staa*. Observe que de tener más instrucciones, en la etapa 1, se estaría leyendo de memoria la siguiente instrucción a ejecutar.

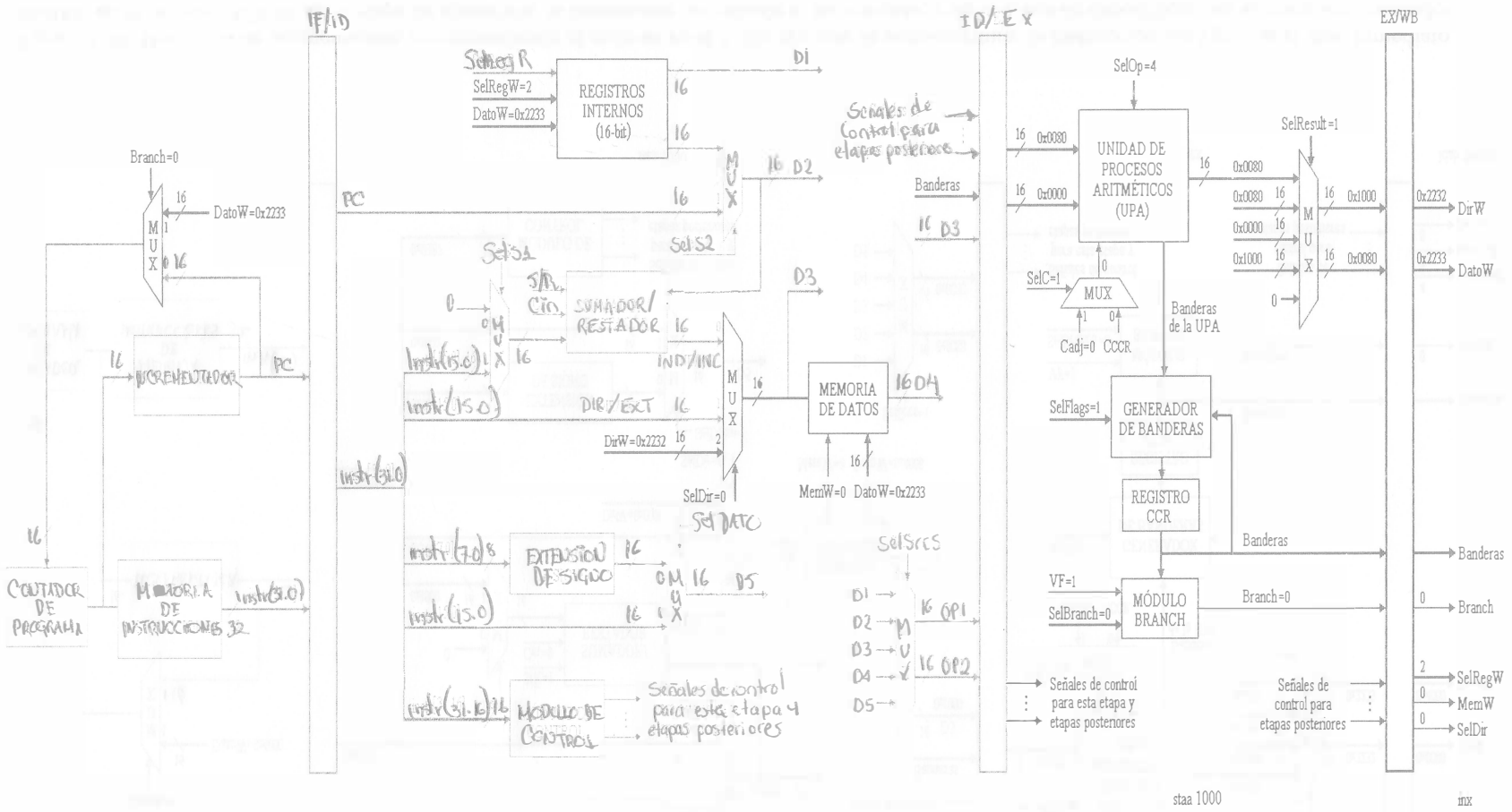


Figura 7.50. Diagrama de segmentación correspondiente al ciclo de reloj 6. En la última etapa, la instrucción *inx* guarda el valor incrementado en el registro IX; mientras, la instrucción *staa* se encuentra en la etapa de ejecución calculando resultados.

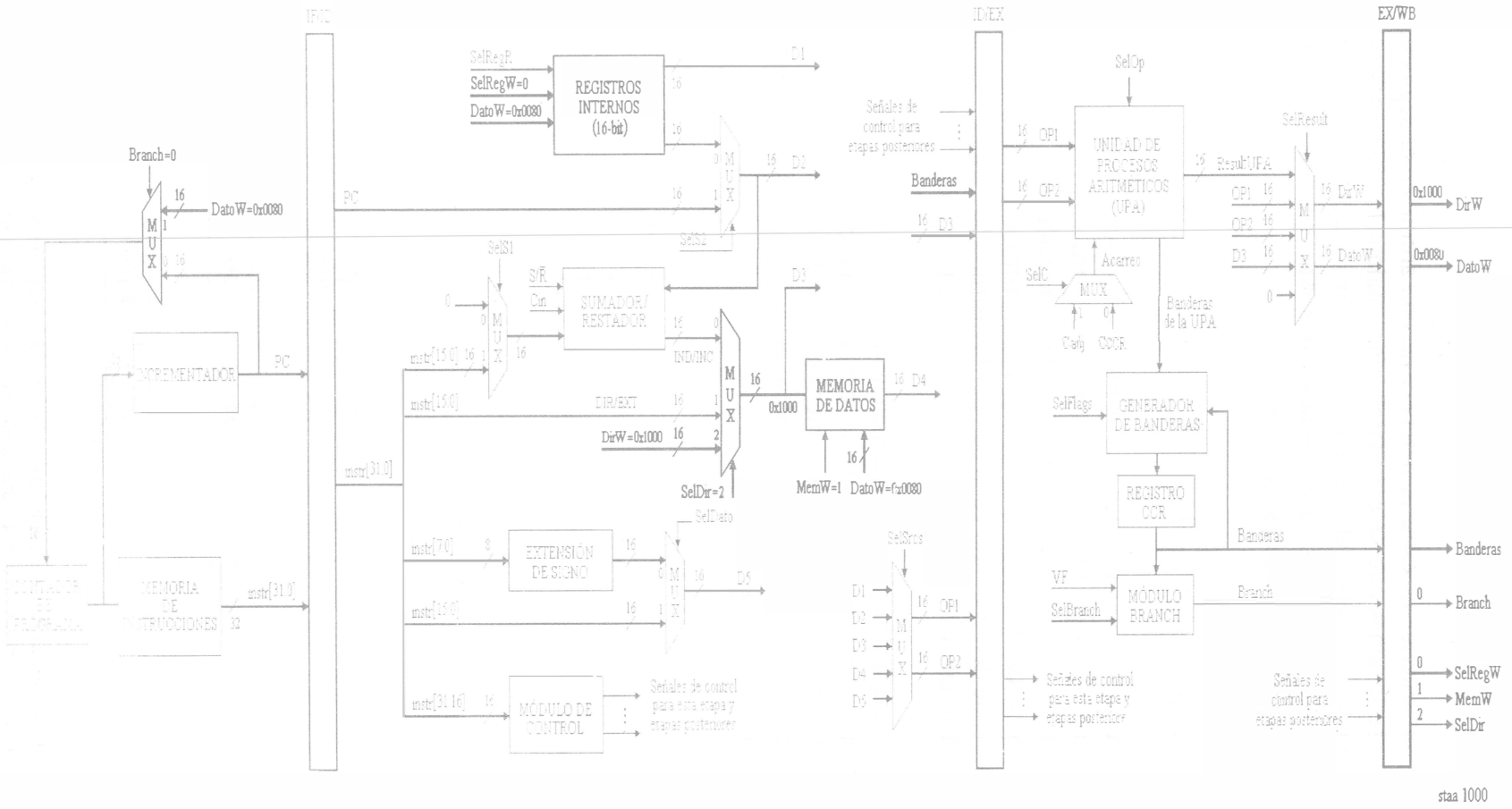


Figura 7.51. Diagrama de segmentación correspondiente al ciclo de reloj 7. En este último ciclo de reloj la instrucción *staa* se encuentra en la etapa de post-escritura guardando, en la dirección 0x1000 de la memoria de datos, el valor leído del registro ACCA.

7.4 RIESGOS POR DEPENDENCIAS DE DATOS

El ejemplo de la sección anterior muestra la potencia de una arquitectura segmentada para un caso ideal; es decir, las instrucciones que se ejecutaron eran totalmente independientes una de la otra, en otras palabras, ninguna de ellas utilizaba los resultados calculados por una instrucción anterior. Ahora es el momento de dejar los casos ideales y observar qué ocurre con los programas reales, por ejemplo, examine los siguientes dos ejemplos.

Riesgos por dependencias de datos: Ejemplo 1

```
aba          ; El resultado obtenido por esta instrucción es escrito en el registro ACCA
anda #6011   ; Uno de los operandos utilizados por esta instrucción, el valor de ACCA,
             ; depende del resultado guardado por la instrucción aba
oraa #FF00   ; Uno de los operandos utilizados por esta instrucción, el valor de ACCA,
             ; depende del resultado guardado por la instrucción anda
staa 1000    ; El dato guardado por staa depende del resultado escrito por oraa
```

Observe que las últimas tres instrucciones son dependientes de los resultados calculados por las instrucciones anteriores, entonces, ¿cómo afecta la dependencia de datos a la ejecución de instrucciones en el cauce?. Para responder esta interrogante nos apoyaremos en el siguiente diagrama de múltiples ciclos de reloj, que muestra cómo se ejecuta la secuencia de instrucciones del ejemplo 1 sobre el modelo de arquitectura de la figura 7.12.

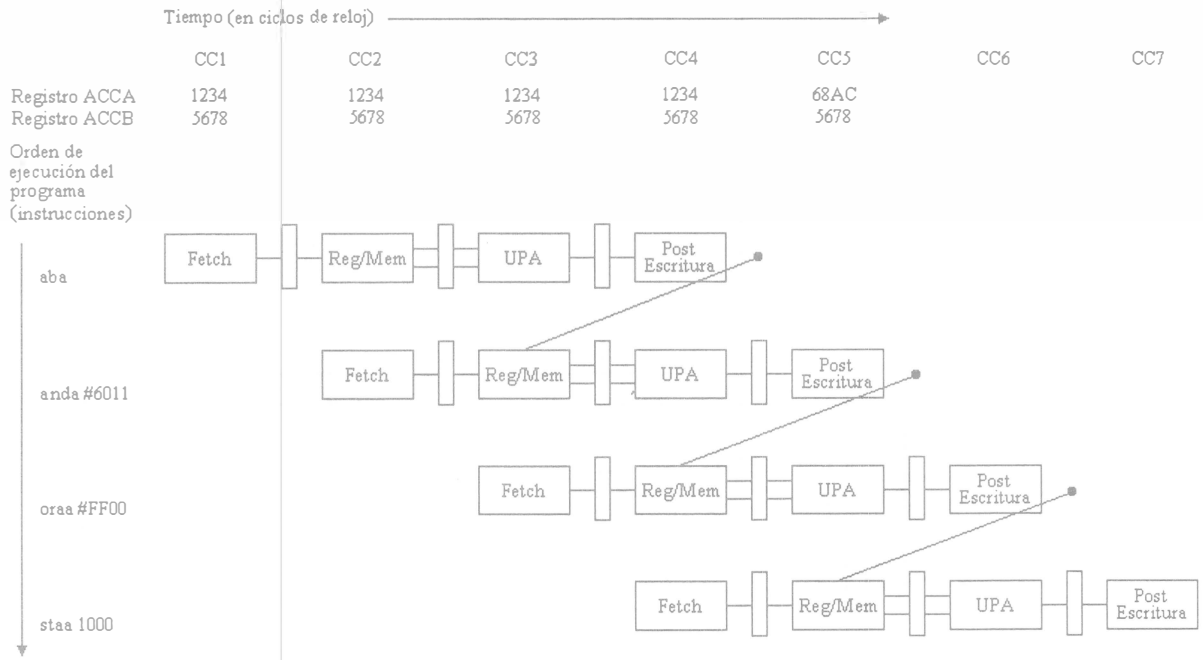


Figura 7.52. Diagrama de múltiples ciclos de reloj para el ejemplo 1. Las líneas trazadas entre las etapas muestran los riesgos por dependencias de datos; estas líneas se trazan desde caminos de datos superiores hacia inferiores, esto es, desde instrucciones anteriores hacia posteriores.

Antes de comenzar a ejecutar la instrucción *aba*, suponga que los registros ACCA y ACCB contienen los valores hexadecimales 0x1234 y 0x5678, respectivamente. En la figura 7.52 los valores de estos registros se muestran en la parte superior del diagrama.

Ahora observe detenidamente el diagrama. La instrucción *aba* requiere de cuatro ciclos de reloj para tener listo el resultado de la suma; este resultado es escrito en el registro ACCA durante el ciclo de reloj CC4, y hasta el ciclo de reloj CC5 podrá ser leído por otra instrucción.

Por otra parte, la instrucción *anda* necesita el resultado de la instrucción anterior para calcular su propio resultado. Observe que cuando la instrucción *anda* lee el contenido del registro ACCA durante el ciclo de reloj CC3, ACCA tiene un valor que no corresponde al resultado de la instrucción *aba*, por lo tanto, la instrucción *anda* ejecutará la operación AND sobre operandos incorrectos. Para que la instrucción *anda* lea el operando adecuado debe esperar hasta el ciclo de reloj CC5, pues es cuando el registro ACCA tiene el resultado correcto. A estas situaciones en donde se leen datos que serán escritos más tarde se les denominan riesgos por dependencias de datos o conflictos por dependencias de datos (data hazards), y son la razón por la complican el diseño de arquitecturas segmentadas de alto rendimiento.

Algo similar ocurre para la instrucción *oraa*, necesita del resultado escrito por la instrucción *anda* para operar correctamente; este resultado se escribe durante el ciclo de reloj CC5 y hasta el ciclo de reloj CC6 estará disponible para lectura. De esta manera, cuando la instrucción *oraa* lee el contenido del registro ACCA en el ciclo de reloj CC4, éste aún conserva el valor 0x1234, el cual, en ese mismo ciclo de reloj, está siendo actualizado con el resultado de la instrucción *aba*. Nuevamente se tiene una situación de riesgo por dependencia de datos.

Finalmente, la instrucción *staa* guarda en memoria el resultado obtenido por la instrucción *oraa*. Un nuevo riesgo por dependencia de datos se hace presente, ya que la instrucción *staa* lee, durante el ciclo de reloj CC5, el contenido del registro ACCA con el resultado de la instrucción *aba* y no con el resultado de la instrucción *oraa*, el cual será escrito hasta el ciclo de reloj CC7.

Note que al final de la secuencia de instrucciones la única instrucción que se calculó correctamente fue *aba* y las demás, debido a las dependencias de datos, guardaron y obtuvieron resultados incorrectos.

Riesgos por dependencias de datos: Ejemplo 2

```
ldab #1234      ; Carga en el registro ACCB el dato inmediato 0x1234
ldaa #5678      ; Carga en el registro ACCA el dato inmediato 0x5678
aba             ; Los operandos que utiliza esta instrucción dependen de los valores escritos
                ; por las instrucciones ldab y ldaa
abx             ; Uno de los operandos, el valor de ACCB, depende de la instrucción ldab
```

A continuación se analiza esta secuencia de instrucciones utilizando el diagrama de múltiples ciclos de reloj mostrado en la figura 7.53.

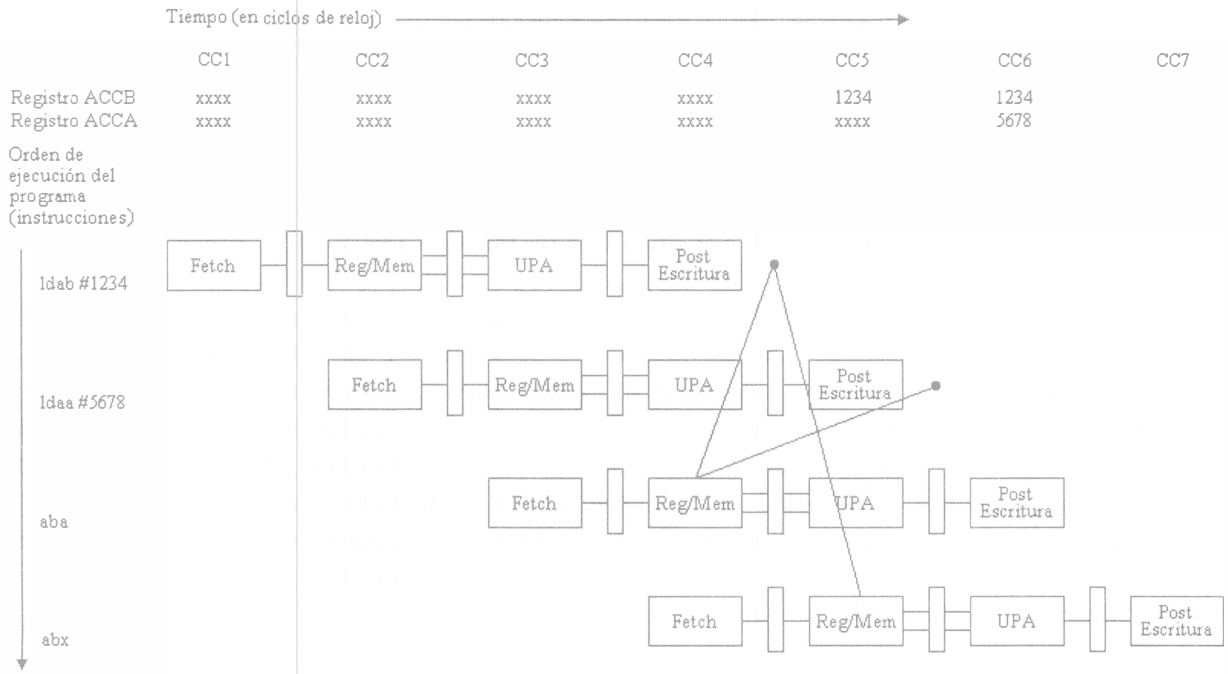


Figura 7.53. Diagrama de múltiples ciclos de reloj para el ejemplo 2. Las líneas trazadas entre las etapas muestran las dependencias de datos, sin embargo, sólo las que retroceden en el tiempo representan riesgos.

La instrucción *ldab* carga en el registro ACCB el valor hexadecimal 0x1234; la carga es ejecutada durante el ciclo de reloj CC4, pero hasta el ciclo de reloj CC5 el nuevo valor estará disponible para lectura. De manera similar, la instrucción *ldaa* carga en el registro ACCA el valor hexadecimal 0x5678, el cual estará disponible para lectura en el ciclo de reloj CC6. Note que la instrucción *ldaa* no utiliza ningún resultado calculado por la instrucción *ldab*, por lo tanto, decimos que estas instrucciones son independientes.

Por otra parte, la instrucción *aba* opera sobre los valores contenidos en los registros ACCB y ACCA, los cuales son modificados por las instrucciones *ldab* y *ldaa*; es decir, la instrucción *aba* es dependiente de las instrucciones *ldab* y *ldaa*. Observe que cuando la instrucción *aba* lee sus operandos durante el ciclo de reloj CC4, éstos aún no han sido escritos por las instrucciones anteriores; por lo tanto, los operandos leídos en este tiempo no corresponden a los valores adecuados, en otras palabras, se tiene una situación de riesgo por dependencia de datos.

Por último, la instrucción *abx* también depende de la instrucción *ldab*, sin embargo, cuando la instrucción *abx* está en busca de sus operandos en el ciclo de reloj CC5, en particular del contenido del registro ACCB, éste ya está disponible para lectura.

7.5 Control de riesgos por dependencias de datos

Existen varios esquemas que nos permiten resolver los riesgos por dependencias de datos. Algunos de ellos son implantados en software y otros en hardware, sin embargo, para nosotros, los esquemas en hardware serán los que tengan mayor importancia.

7.5.1 CONTROL DE RIESGOS POR DEPENDENCIAS DE DATOS POR MEDIO DE SOFTWARE

Este método consiste en dar reglas al compilador de manera que no genere secuencias de instrucciones como las secuencias de los ejemplos de la sección anterior. Por ejemplo, para la secuencia del ejemplo 1, el compilador podría insertar dos instrucciones independientes entre las instrucciones *aba* y *anda*, haciendo que desaparezca el riesgo. Algo similar se haría para eliminar el riesgo entre las instrucciones *anda* y *oraa*, y las instrucciones *oraa* y *staa*.

Cuando no se puedan encontrar instrucciones independientes, el compilador podría insertar instrucciones *nop* garantizando así la independencia de datos entre instrucciones. La abreviatura *nop* significa no operación, porque esta instrucción no lee ningún registro, no modifica ningún dato y no escribe ningún resultado.

A continuación se presentan las secuencias de instrucciones para los ejemplos 1 y 2 de la sección 7.4 utilizando este esquema de control de riesgos por dependencias de datos.

Control de riesgos por medio de software: Ejemplo 1

```
aba          ; Suma ACCA y ACCB, el resultado se guarda en ACCA
nop          ; La dependencia de datos entre las instrucciones aba y anda se elimina
nop          ; mediante la inserción de estas dos instrucciones nop
anda #6011   ; AND lógica entre el valor 0x6011 y el resultado de aba
nop          ; La dependencia de datos entre las instrucciones anda y oraa se elimina
nop          ; mediante la inserción de estas dos instrucciones nop
oraa #FF00   ; OR lógica entre el valor 0xFF00 y el resultado de anda
nop          ; La dependencia de datos entre las instrucciones oraa y staa se elimina
nop          ; mediante la inserción de estas dos instrucciones nop
staa 1000    ; Guarda el resultado obtenido por la instrucción oraa
```

Control de riesgos por medio de software: Ejemplo 2

```
ldab #1234   ; Carga en ACCB el valor 0x1234
ldaa #5678   ; Carga en ACCA el valor 0x5678
nop          ; La dependencia de datos entre las instrucciones ldab, ldaa y aba se elimina
nop          ; mediante la inserción de estas dos instrucciones nop
aba          ; Suma ACCA y ACCB, el resultado se guarda en ACCA
abx          ; Suma IX y ACCB, el resultado se guarda en IX
```

Aunque las nuevas secuencias de instrucciones funcionan adecuadamente en la arquitectura segmentada de la figura 7.12, estas instrucciones *nop* ocupan ciclos de reloj que no realizan trabajo útil. Idealmente, el compilador encontrará instrucciones para ayudar al cálculo en lugar de insertar instrucciones inactivas.

7.5.2 CONTROL DE RIESGOS POR DEPENDENCIAS DE DATOS POR MEDIO DE DETENCIONES

El esquema más sencillo para resolver los riesgos por dependencias de datos en hardware es detener las instrucciones en el cauce hasta que se resuelva el riesgo. Este tipo de detenciones se conocen con el sobrenombre de burbujas (bubbles); con esta estrategia, primero se detecta un riesgo por dependencia de datos, y después se detienen las instrucciones en el cauce (se insertan burbujas) hasta que se resuelve el riesgo.

Si realiza una inspección más estricta de la arquitectura segmentada mostrada en la figura 7.12, notará que un riesgo por dependencia de datos se presenta cuando una instrucción trata de leer en su etapa 2 el mismo registro que una instrucción anterior intenta escribir en su etapa 4. De esta manera, se tienen dos condiciones que permiten determinar si existen riesgos por dependencia de datos o no; estas condiciones son las siguientes.

1. ID/EX.SelRegW := Etapa2.SelRegR
2. EX/WB.SelRegW := Etapa2.SelRegR

La notación anterior nos permite describir con mayor precisión en qué parte de la arquitectura se presenta el riesgo por dependencia de datos. Por ejemplo, la primera condición revisa el valor de la señal de escritura que guarda una instrucción anterior en el registro de segmentación ID/EX, si este valor corresponde a alguno de los registros que intenta leer una instrucción posterior en su etapa 2, entonces se generará un riesgo. Recuerde que la etapa 4 de la segmentación se dedica a la escritura de registros, por lo tanto, las señales de control necesarias para hacer la escritura son guardadas temporalmente en los registros de segmentación ID/EX y EX/WB.

La segunda condición revisa el valor de la señal de escritura guardada por una instrucción anterior en el registro de segmentación EX/WB, si este valor corresponde a alguno de los registros que intenta leer una instrucción posterior en su etapa 2, entonces se generará un riesgo.

Para dejar en claro estas dos condiciones se revisarán nuevamente los ejemplos de la sección 7.4. También es conveniente que revise las figuras 7.52 y 7.53 para cualquier duda sobre las etapas en las que se presentan y detectan los riesgos.

Control de riesgos por medio de detenciones: Ejemplo 1

En este ejemplo existen tres riesgos por dependencias de datos. El primer riesgo se presenta entre las instrucciones *aba* y *anda*, y corresponde a la condición ID/EX.SelRegW := Etapa2.SelRegR. Esto significa, que durante la etapa de decodificación de la instrucción *anda* (ciclo de reloj CC3) se está intentando leer el registro ACCA, el cual aún no ha sido actualizado por la instrucción *aba*, ya que la señal de control encargada de actualizar este registro, SelRegW, se encuentra almacenada en el registro de segmentación ID/EX.

El segundo riesgo se presenta entre las instrucciones *anda* y *oraa*, y al igual que el riesgo anterior, corresponde a la condición ID/EX.SelRegW := Etapa2.SelRegR. Esto quiere decir, que la instrucción *oraa* también está intentando leer un registro (durante el ciclo de reloj CC4) que aún no

ha sido actualizado por una instrucción anterior, ya que la señal de control encargada de actualizar dicho registro, SelRegW, aún se encuentra almacenada en el registro de segmentación ID/EX.

Finalmente, el tercer riesgo se presenta entre las instrucciones *oraa* y *staa*, y también corresponde a la condición ID/EX.SelRegW := Etapa2.SelRegR; es decir, la instrucción *staa* está tratando de leer un registro (en el ciclo de reloj CC5) que aún no ha sido actualizado por la instrucción *oraa*.

Control de riesgos por medio de detenciones: Ejemplo 2

En este ejemplo existen dos riesgos por dependencias de datos. El primer riesgo se presenta entre las instrucciones *ldab* y *aba*, y corresponde a la condición EX/WB.SelRegW := Etapa2.SelRegR. Esto significa, que durante la etapa de decodificación de la instrucción *aba* (ciclo de reloj CC4) se está intentando leer el registro ACCB, el cual no ha sido actualizado por la instrucción *ldab*, ya que la señal de control encargada de actualizar este registro, SelRegW, se encuentra almacenada en el registro de segmentación EX/WB.

El segundo riesgo se presenta entre las instrucciones *ldaa* y *aba*, y corresponde a la condición ID/EX.SelRegW := Etapa2.SelRegR. Esto significa, que durante la etapa de decodificación de la instrucción *aba* (ciclo de reloj CC4) se está intentando leer el registro ACCA, el cual no ha sido actualizado por la instrucción *ldaa*, ya que la señal de control encargada de actualizar este registro, SelRegW, se encuentra almacenada en el registro de segmentación ID/EX.

Por último, observe que entre las instrucciones *ldab* y *abx* no existe riesgo, ya que cuando *abx* lee el contenido del registro ACCB durante el ciclo de reloj CC5, éste ya está disponible.

Cómo implantar las detenciones en hardware

Ya que se sabe en dónde y cuándo se presentan los riesgos por dependencias de datos, entonces se pueden detectar y eliminar. La detección de un riesgo, consiste en comparar las señales de control que se estudiaron arriba; mientras que su eliminación, consiste en detener a la instrucción dependiente hasta que se termine de ejecutar la instrucción causante de la dependencia.

Para detener las instrucciones en el cauce se necesita conseguir el mismo efecto que produce la ejecución de la instrucción *nop*; para ello, se agrega una unidad de detenciones, la cual detecta los riesgos y genera, durante la etapa 2, las señales de control propias de la instrucción *nop*; en términos de diseñadores de hardware, se dice que se inserta una burbuja. Sin embargo, si una instrucción en la etapa 2 es detenida, entonces la instrucción de la etapa 1 también debe ser detenida; de lo contrario, se pierde la instrucción buscada en la etapa 1. Para que esto no ocurra, el contenido del registro PC (contador de programa) y el contenido del registro de segmentación IF/ID no deben modificarse; de esta manera, la etapa 1 traerá la siguiente instrucción a ejecutar utilizando el mismo valor de PC, y la etapa 2 continuará leyendo la misma instrucción del registro de segmentación IF/ID.

La figura 7.54 muestra la nueva arquitectura segmentada utilizando el esquema de detenciones para reducir los riesgos por dependencias de datos.

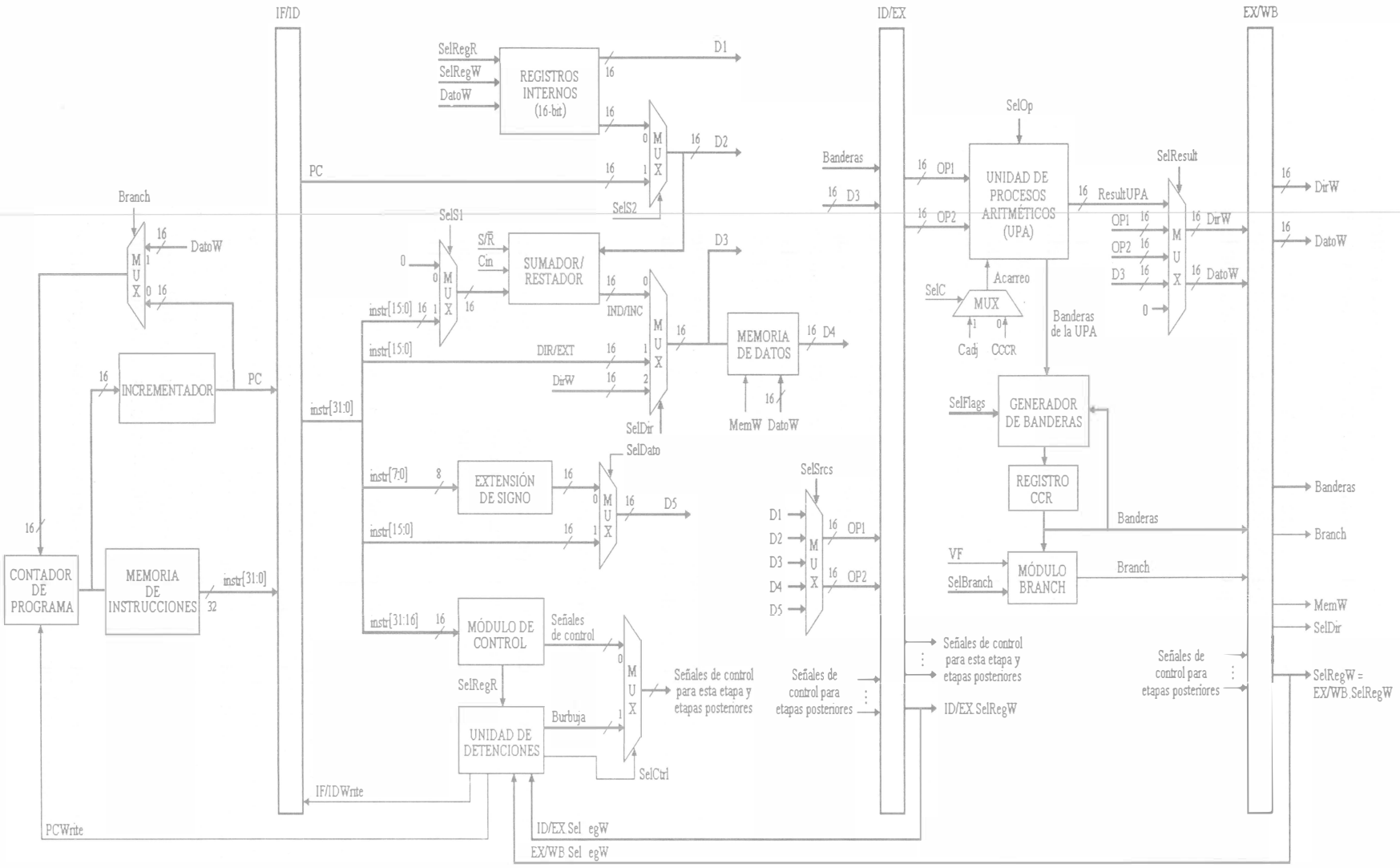


Figura 7.54. Arquitectura segmentada que utiliza el esquema de detenciones para reducir los riesgos por dependencias de datos.

El funcionamiento de la unidad de detenciones es muy simple. Primero compara el valor de la señal SelRegR, proveniente de la unidad de control de la etapa 2, contra los valores de las señales SelRegW, guardadas en los registros de segmentación ID/EX y EX/WB. Si se encuentra alguna correspondencia entre estas señales, significa que se está tratando de leer un registro que no ha sido actualizado, en otras palabras, se detecta un riesgo por dependencia de datos.

Una vez registrado el riesgo, la unidad de detenciones debe generar una burbuja; es decir, debe asignar ciertos valores a las señales de control de manera que se obtenga el mismo efecto de una instrucción *nop*. La asignación de estos valores se realiza por medio de la señal SelCtrl, la cual seleccionará las señales de control del módulo de detenciones en lugar de las señales de control del módulo de control. Recuerde que algunas señales de control son utilizadas en la etapa 2 y otras son guardadas en el registro de segmentación ID/EX para etapas posteriores. Adicionalmente, se cuenta con las señales PCWrite e IF/IDWrite que habilitan las operaciones de escritura en el registro contador de programa y en el registro de segmentación IF/ID, respectivamente.

La siguiente tabla muestra las condiciones para detectar los riesgos por dependencias de datos, así como las señales de salida, generadas por la unidad de detenciones, que permiten eliminarlos.

Condiciones de entrada		*Señales de salida*		
SelRegR	ID/EX.SelRegW ó EX/WB.SelRegW	PCWrite	IF/IDWrite	SelCtrl
1	1, 4	0	0	1
2	2, 4	0	0	1
3	3, 4	0	0	1
4	1	0	0	1
5	4	0	0	1
6	1, 2	0	0	1
7	1, 3	0	0	1
8	5	0	0	1
9	2	0	0	1
A	3	0	0	1
B	6	0	0	1
C	1, 6	0	0	1
D	4, 6	0	0	1
E	2, 6	0	0	1
F	3, 6	0	0	1
Cualquier otra combinación no presente en esta tabla		1	1	0

Tabla 7.16. Condiciones para detectar los riesgos por dependencias de datos y señales de salida generadas por la unidad de detenciones para eliminar estos riesgos.

La figura 7.55 muestra un diagrama de múltiples ciclos de reloj usando el esquema de detenciones para resolver los riesgos por dependencias de datos del ejemplo 2. Este ejemplo fue elegido porque en él se presentan las dos condiciones de riesgo por dependencia de datos.

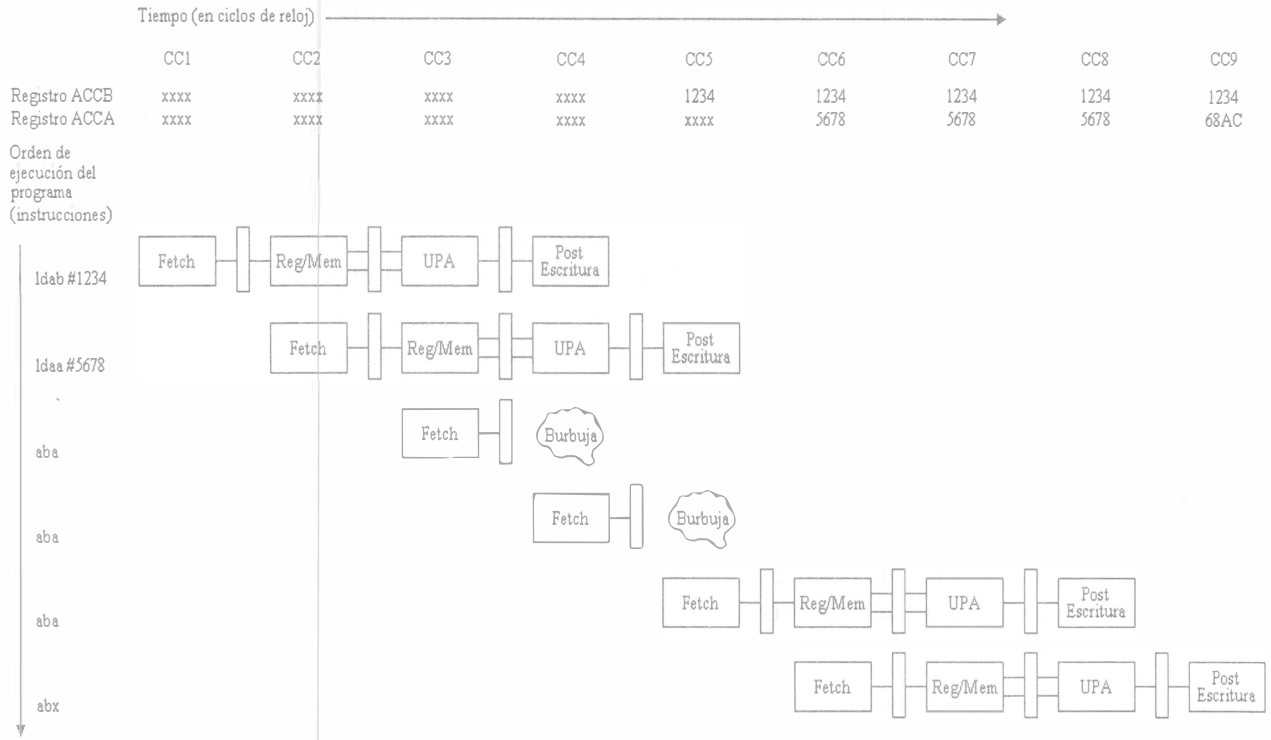


Figura 7.55. Diagrama de múltiples ciclos de reloj para el ejemplo 2; en él se utiliza el esquema de detenciones para resolver los riesgos por dependencias de datos.

Cuando la instrucción *aba* intenta leer los registros ACCA y ACCB durante el ciclo de reloj CC4, la unidad de detenciones detecta la presencia de un riesgo e inserta una burbuja. La detección del riesgo se logra gracias a la señal de lectura SelRegR de la instrucción *aba* y a las señales de escritura SelRegW de las instrucciones *ldab* y *ldaa*, las cuales se encuentran almacenadas temporalmente en los registros de segmentación EX/WB e ID/EX, respectivamente. Observe que en este instante, la unidad de detenciones detecta las dos condiciones de riesgo para la misma instrucción: la primera condición, ID/EX.SelRegW := Etapa2.SelRegR, se presenta entre las instrucciones *aba* y *ldaa*; mientras que la segunda condición, EX/WB.SelRegW := Etapa2.SelRegR, se presenta entre las instrucciones *aba* y *ldab*.

Una vez detectado el riesgo, la unidad de detenciones activa la señal SelCtrl para seleccionar las señales de control de la burbuja. No olvide que las señales de control para la burbuja son generadas por la unidad de detenciones y producen el mismo efecto que la instrucción *nop*. Por otra parte, las señales PCWrite e IF/IDWrite son colocadas a cero para deshabilitar las operaciones de escritura en los registros PC e IF/ID, de esta manera, en el siguiente ciclo de reloj, se intentará decodificar nuevamente la instrucción que originó el riesgo, es decir, *aba*.

En el ciclo de reloj CC5 el registro ACCB ya fue actualizado. En este mismo ciclo de reloj, la instrucción *ldaa* está por escribir el registro ACCA, la burbuja insertada en el ciclo anterior está en la etapa de ejecución, y la instrucción *aba* se intenta decodificar por segunda ocasión. Debido a que la instrucción *ldaa* no ha terminado su ejecución, la unidad de detenciones vuelve a encontrar un riesgo por dependencia de datos, ya que la señal SelRegR de la instrucción *aba* coincide con la señal

de escritura SelRegW de la instrucción *ldaa*, la cual está guardada en el registro de segmentación EX/WB. En consecuencia, la unidad de detenciones inserta otra burbuja en la etapa 2 y deshabilita las operaciones de escritura en los registros PC e IF/ID.

En el ciclo de reloj CC6, la burbuja insertada en el ciclo CC4 está en la etapa de post-escritura, la burbuja insertada en el ciclo CC5 está en la etapa de ejecución, y la instrucción *aba* se intenta decodificar por tercera vez. En esta ocasión, la unidad de detenciones no detecta ningún riesgo entre la instrucción *aba* y las burbujas de las etapas 3 y 4; por lo tanto, el riesgo ha sido eliminado y la ejecución de instrucciones en el cauce continúa de manera normal hasta que un nuevo riesgo es detectado. No olvide que las burbujas no leen operandos, no calculan resultados y no escriben en registros; es decir, son independientes de cualquier instrucción.

Detenciones debido a accesos múltiples a memoria

Suponga que en el ciclo de reloj X una instrucción en la etapa 4 intenta guardar un resultado en la memoria de datos, y en ese mismo instante, una instrucción en la etapa 2 intenta leer un dato de la misma memoria. Un nuevo tipo de riesgo se presenta en la arquitectura segmentada, ya que la memoria de datos ubicada en la etapa 2, sólo puede leer ó escribir datos en un instante dado, pero nunca los dos. Para poder detectar y eliminar este nuevo riesgo, se extenderá la unidad de detenciones de la figura 7.54. El nuevo modelo de arquitectura, que incorpora detenciones debido a accesos múltiples a memoria, y detenciones por dependencias de datos, se muestra en la figura 7.56.

Para detectar un riesgo debido a accesos múltiples a memoria se utilizan las señales de control SelSrcs y EX/WB.MemW; estas dos señales son suficientes para determinar si en la memoria se está intentando leer y escribir simultáneamente. Por ejemplo, si la señal SelSrcs selecciona al bus D4, significa que la memoria será accedida para lectura, ya que el bus D4 es el bus por donde se leen datos de la memoria; por otra parte, si la señal MemW, proveniente del registro de segmentación EX/WB, está encendida, significa que un dato será escrito en memoria. Cuando las dos condiciones anteriores se presentan, entonces se genera un riesgo por accesos múltiples a memoria.

Una vez detectado el riesgo, el siguiente paso es eliminarlo utilizando la técnica de las detenciones. Esta técnica consiste en detener la instrucción que intenta leer un dato de memoria, y permitir que la instrucción que intenta escribir en ella termine de ejecutar su tarea; para ello, se debe generar una burbuja en la etapa 2 tal y como se realiza para los riesgos por dependencias de datos, y se deben deshabilitar las escrituras en los registros PC e IF/ID para no perder la instrucción que se trataba de ejecutar al momento de detectar el riesgo. Si observa cuidadosamente la figura 7.54 notará que la generación de la burbuja en la etapa 2 ya fue implantada cuando se resolvieron los riesgos por dependencias de datos; por lo tanto, lo único que hace falta, es permitir que la instrucción que va a escribir en memoria termine de ejecutar su trabajo. Para esto, ha sido agregada una señal de control cuyo nombre es SelD, y es generada por la unidad de detenciones tras detectar un riesgo por accesos múltiples a memoria. La señal SelD presenta el siguiente comportamiento: si SelD=0, entonces se selecciona la señal SelDirCtrl, que es la señal SelDir que genera el módulo de control para la etapa 2; y si SelD=1, entonces se selecciona la señal SelDir proveniente del registro de segmentación EX/WB, dando prioridad a la escritura en memoria.

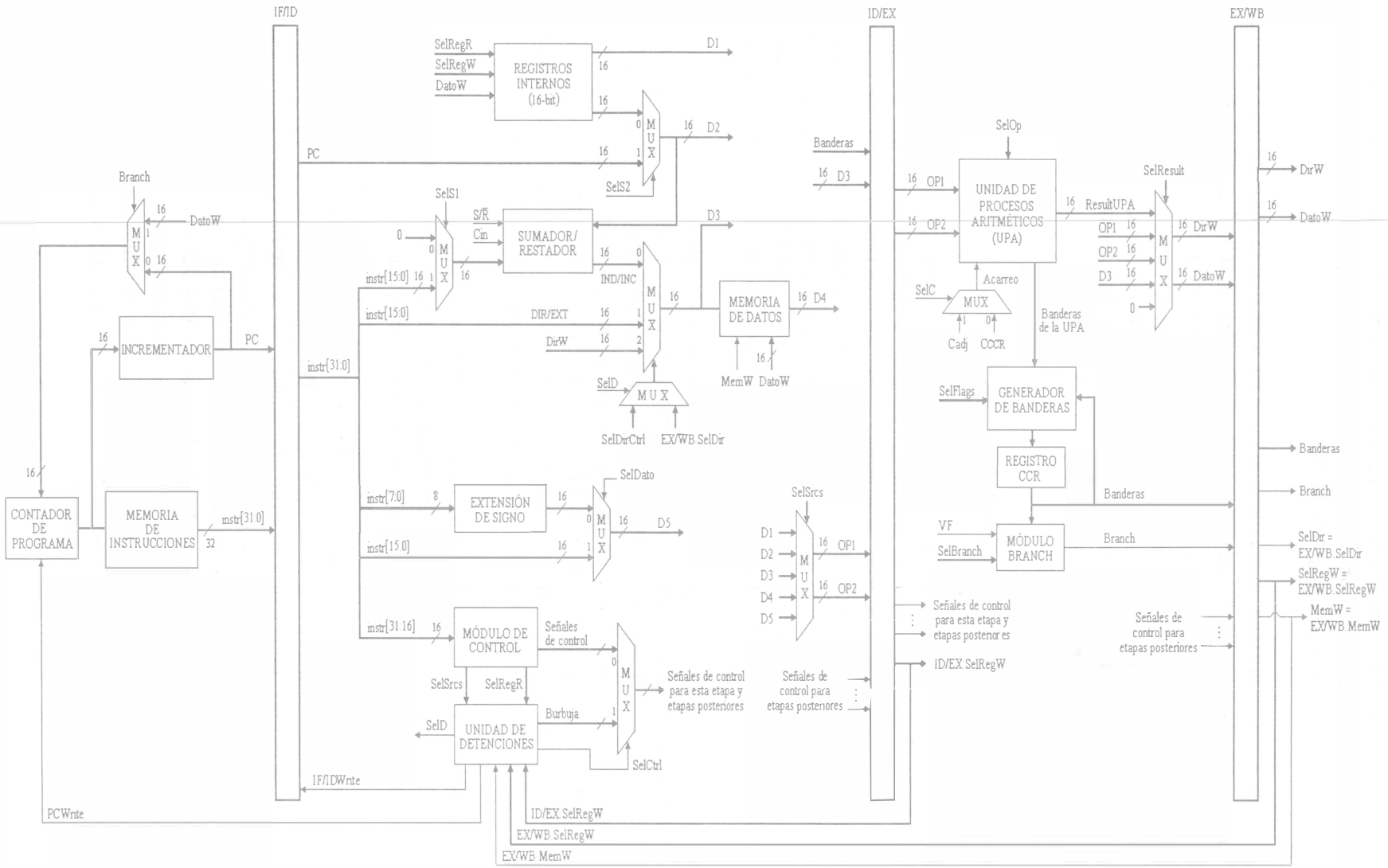


Figura 7.56. Arquitectura segmentada que utiliza el esquema de detenciones para reducir los riesgos por dependencias de datos y los accesos múltiples a memoria.

La siguiente tabla muestra las condiciones para detectar los riesgos por accesos múltiples a memoria, así como las señales de salida, generadas por la unidad de detenciones, que permiten eliminar los riesgos de este tipo.

Condiciones de entrada		*Señales de salida*			
SelSrcs	EX/WB.MemW	PCWrite	IF/IDWrite	SelD	SelCtrl
2	1	0	0	1	1
4	1	0	0	1	1
6	1	0	0	1	1
2	0	1	1	0	0
4	0	1	1	0	0
6	0	1	1	0	0

Tabla 7.17. Condiciones para detectar los riesgos por accesos múltiples a memoria y señales de salida generadas por la unidad de detenciones para eliminar estos riesgos.

Tenga en cuenta que tanto la tabla 7.16 como la tabla 7.17 están implantadas dentro de la unidad de detenciones, la cual dará prioridad a los riesgos por accesos múltiples a memoria antes que a los riesgos por dependencias de datos.[10]

Finalmente, las figuras 7.57 a 7.60 muestran de manera detallada los eventos que ocurren en la arquitectura para el programa de la figura 7.55; estas figuras utilizan el esquema de detenciones tratado en la figura 7.56. Considere como condición inicial que la dirección en memoria de la primera instrucción a ejecutar es 0x0400.

7.5.3 CONTROL DE RIESGOS POR DEPENDENCIAS DE DATOS POR MEDIO DE ANTICIPACIONES

Detener las instrucciones en el cauce garantiza la ejecución de instrucciones dependientes muy próximas de manera correcta, sin embargo, el costo de las detenciones afecta negativamente al rendimiento.

En el ejemplo 1 de la sección 7.4 se observó que la instrucción *anda* necesitaba del resultado de la instrucción *aba* para operar correctamente, ya que dicho resultado era uno de los operandos de la instrucción *anda*. Si analiza detenidamente la figura 7.52 notará que el resultado de la instrucción *aba* se utiliza hasta la etapa de ejecución de la instrucción *anda* (ciclo de reloj CC4); en este momento, el resultado de *aba* no ha sido escrito en el registro ACCA, sin embargo, ya está disponible en el campo DatoW del registro de segmentación EX/WB. Lo mismo ocurre cuando la instrucción *oraa* intenta calcular su resultado en el ciclo de reloj CC5; el resultado de la instrucción *anda* no ha sido guardado en el registro ACCA, pero ya está disponible en el registro de segmentación EX/WB.

[10] Para poder implantar las tablas 7.16 y 7.17 en la unidad de detenciones, el número de salidas para ambos casos debe ser el mismo; por lo tanto, la tabla 7.16 también debe anexar la señal de salida SelD, la cual tomará el valor de cero para cualquier caso de riesgo por dependencia de datos.

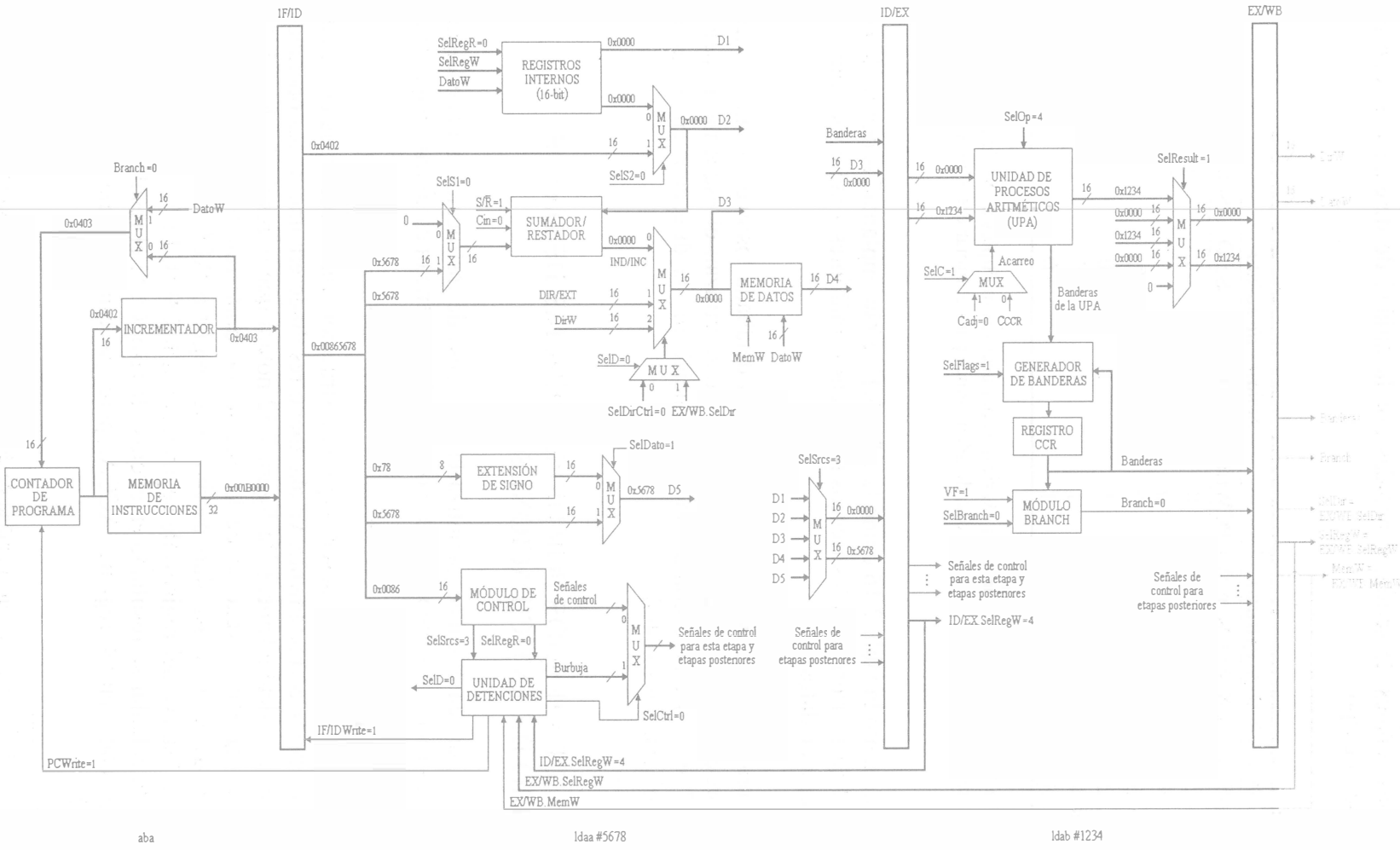


Figura 7.57. Diagrama de segmentación correspondiente al ciclo de reloj 3. La instrucción *ldab* se encuentra en la etapa 3, la instrucción *ldaa* en la etapa 2 y la instrucción *aba* en la etapa 1; hasta el momento no se han presentado riesgos por dependencias de datos o por accesos múltiples a memoria.

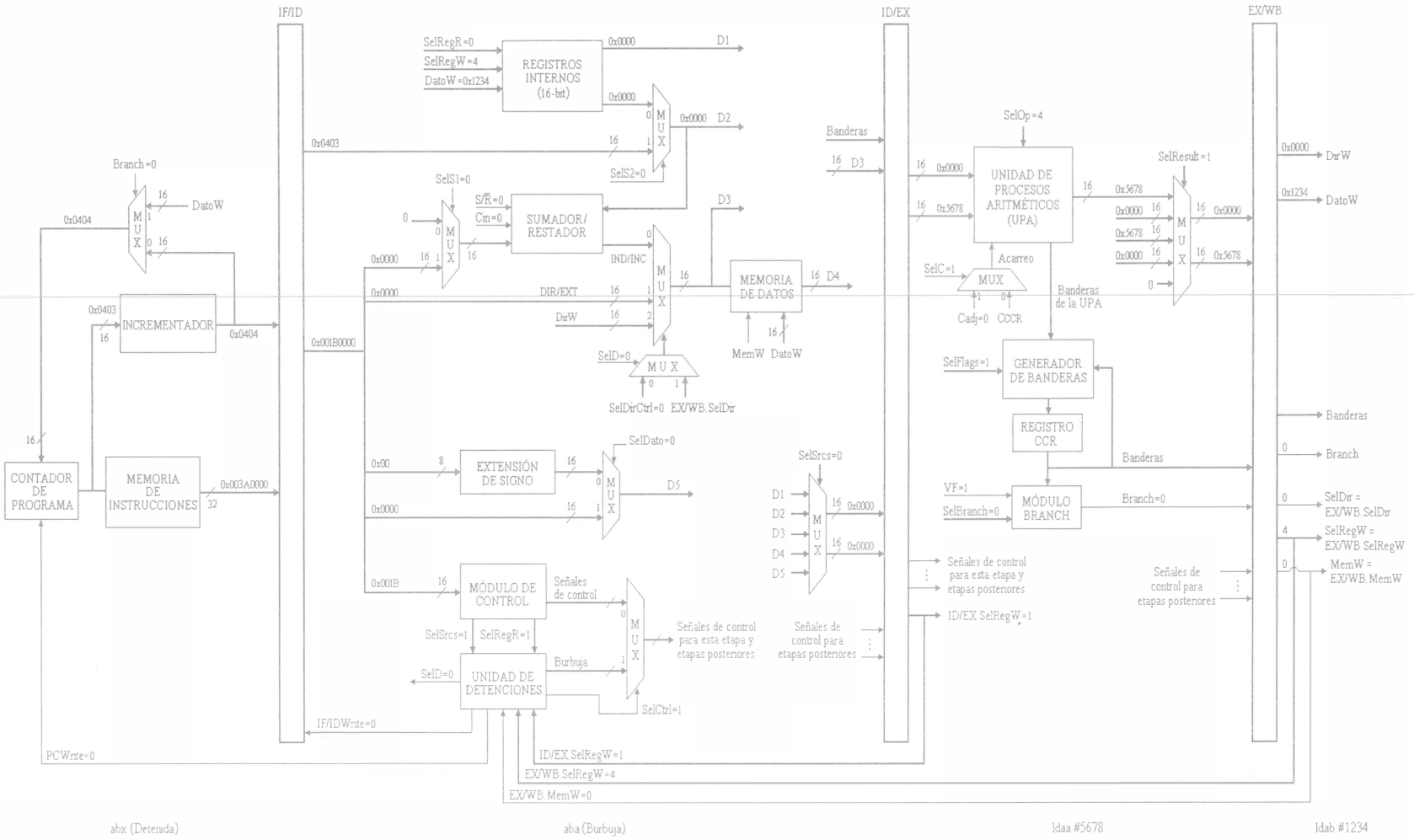


Figura 7.58. Diagrama de segmentación correspondiente al ciclo de reloj 4. La instrucción *ldab* se encuentra en la etapa de post-escritura intentado escribir un resultado en el registro ACCB; mientras, la instrucción *ldaa* está en la etapa 3, la instrucción *aba* en la etapa 2 y la instrucción *abx* en la etapa 1. En este instante, la unidad de detenciones detecta un riesgo por dependencia de datos, ya que la instrucción de la etapa 2 intenta leer un registro que está siendo escrito en este mismo momento por la instrucción *ldab*. También es detectado un riesgo por dependencia de datos entre la instrucción *aba* y la instrucción *ldaa*, pues se intenta leer el registro ACCA que aún no ha sido actualizado. Por lo tanto, la unidad de detenciones inserta una burbuja en la etapa 2 y detiene las operaciones de escritura en los registros PC e IF/ID.

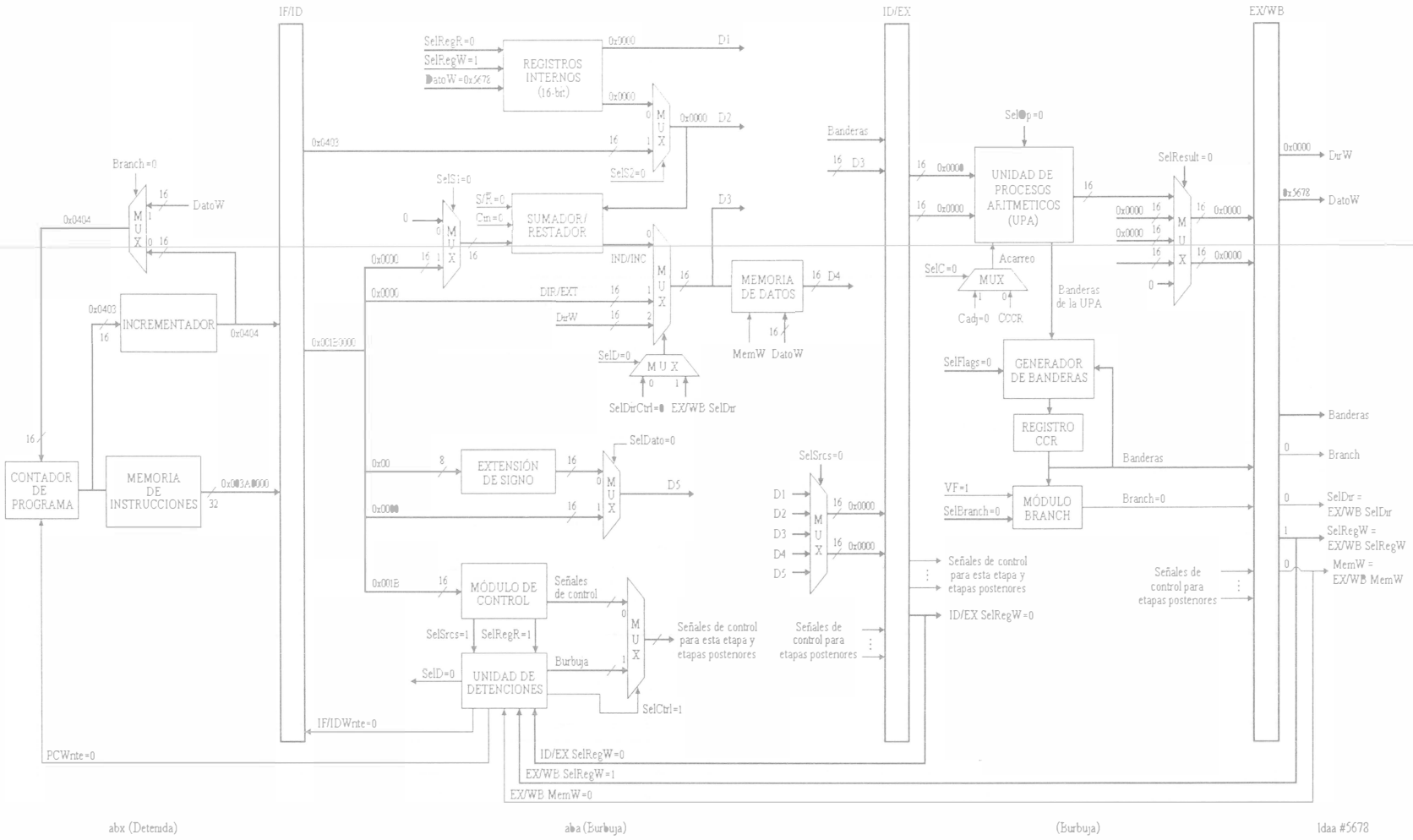


Figura 7.59. Diagrama de segmentación correspondiente al ciclo de reloj 5. La instrucción *ldab* ha terminado de ejecutarse; la instrucción *ldaa* se encuentra escribiendo un resultado en el registro ACCA; y las instrucciones *aba* y *abx* aún continúan detenidas en las etapas 2 y 1, respectivamente. Observe que en la etapa 3 está la burbuja que se insertó en el ciclo de reloj anterior; las operaciones que se ejecutan en esta etapa corresponden a las de una instrucción *nop*. Por otra parte, la unidad de detenciones vuelve a detectar un riesgo por dependencia de datos entre las instrucciones *ldaa* y *aba*, ya que se intenta leer un registro que está siendo escrito en este mismo instante. Por lo tanto, nuevamente es insertada una burbuja en la etapa 2 y son detenidas las operaciones de escritura en los registros PC e IF/ID.

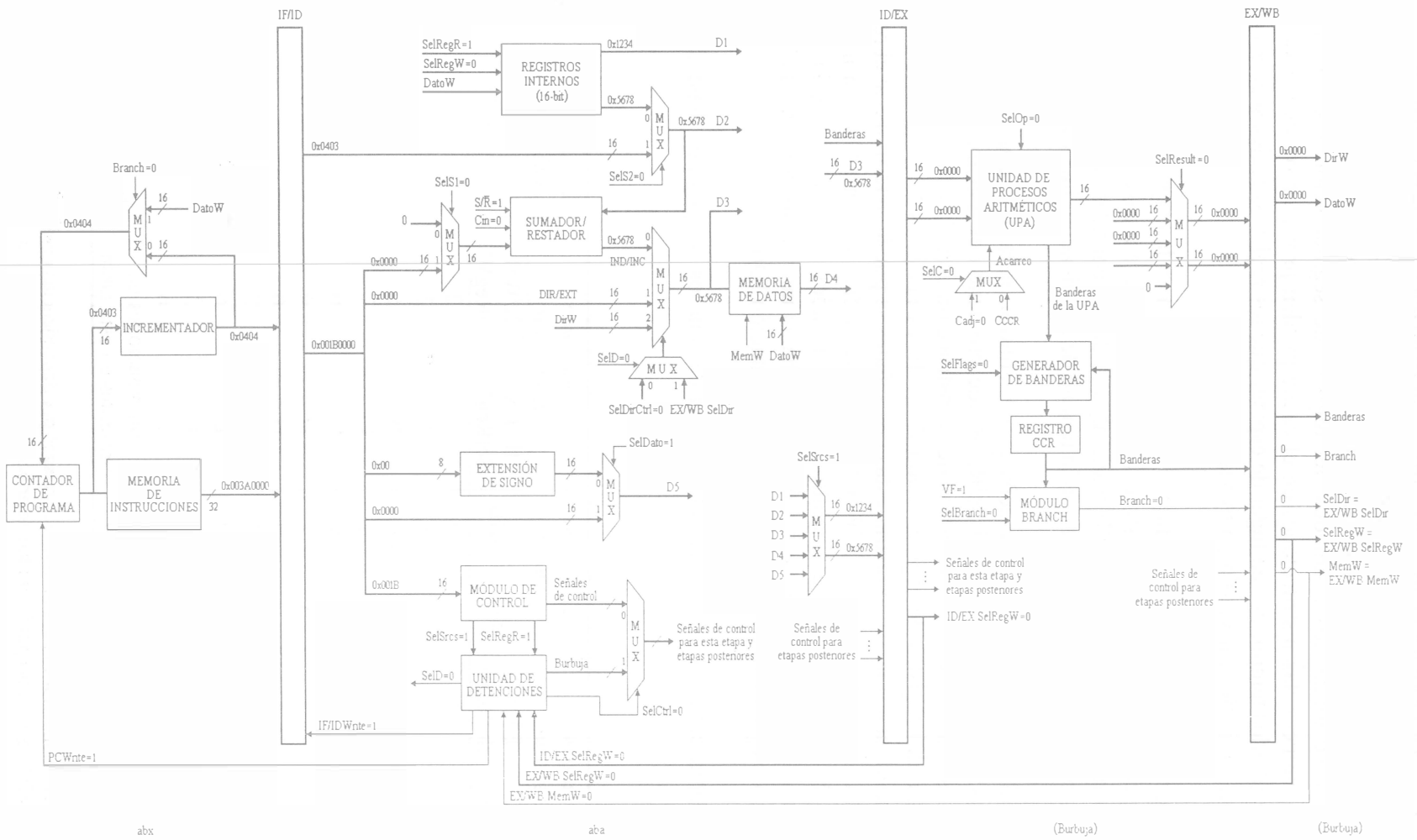


Figura 7.60. Diagrama de segmentación correspondiente al ciclo de reloj 6. La instrucción *ldaa* ya terminó de ejecutarse; y en las etapas 3 y 4 continúa la ejecución de las burbujas anteriormente insertadas. En este momento, la unidad de detenciones no detecta ningún riesgo, por lo que las instrucciones *aba* y *abx* continúan ejecutándose de manera normal.

La disponibilidad del dato requerido en el registro de segmentación EX/WB sugiere un atajo que puede reducir las pérdidas de tiempo debido a las detenciones, pues es posible leer los resultados del registro de segmentación en lugar de esperar a que la etapa 4 escriba los resultados en los registros reales. Por lo tanto, si se toman los resultados del registro de segmentación EX/WB hacia las entradas de la UPA, entonces, las instrucciones en el cauce pueden proceder sin detenciones. Esta técnica, que utiliza resultados temporales en lugar de esperar que los registros reales sean escritos, se denomina anticipación (en inglés forwarding o bypassing).

Para implantar este método, se utiliza una unidad de anticipación en la etapa de ejecución, la cual detecta las dependencias de datos entre instrucciones y anticipa, en caso de dependencia, el resultado del registro de segmentación EX/WB hacia alguna de las entradas de la UPA. La figura 7.61 muestra la nueva arquitectura segmentada con la unidad de anticipación incorporada.

La unidad de anticipación necesita dos señales de control para detectar los riesgos por dependencias de datos: la primera es la señal SelRegR que proviene del registro de segmentación ID/EX, es decir, es la señal de lectura de registros de la instrucción actual; y la segunda es la señal SelRegW que proviene del registro de segmentación EX/WB, es decir, es la señal de escritura de registros de la instrucción anterior. Al comparar estas dos señales se puede determinar si el resultado guardado en el registro de segmentación EX/WB corresponde al dato requerido por la instrucción actual en la etapa 3. Si es así, existen dos multiplexores colocados a la entrada de la UPA que adelantan el resultado del registro de segmentación EX/WB hacia una de las entradas de la UPA, o bien, eligen los operandos guardados en el registro de segmentación ID/EX.

La tabla 7.18 muestra cómo la unidad de anticipación detecta los riesgos por dependencias de datos y los elimina.

Condiciones de entrada		*Señales de salida*		*Condiciones de entrada*		*Señales de salida*	
ID/EX SelRegR	EX/WB SelRegW	SelA	SelB	ID/EX SelRegR	EX/WB SelRegW	SelA	SelB
1	1	1	0	9	2	0	1
1	4	0	1	A	3	0	1
2	4	1	0	B	6	0	1
2	2	0	1	C	1	1	0
3	4	1	0	C	6	0	1
3	3	0	1	D	4	1	0
4	1	1	0	D	6	0	1
5	4	1	0	E	2	1	0
6	1	1	0	E	6	0	1
6	2	0	1	F	3	1	0
7	1	1	0	F	6	0	1
7	3	0	1	Combinaciones no presentes en esta tabla		0	0
8	5	1	0				

Tabla 7.18. Condiciones para detectar los riesgos por dependencias de datos y señales de salida generadas por la unidad de anticipación para eliminar estos riesgos.

A continuación se analizan algunas de las combinaciones mostradas en la tabla 7.18.

Suponga que la señal SelRegR del registro de segmentación ID/EX vale uno, al igual que la señal SelRegW del registro de segmentación EX/WB. Gracias a la señal SelRegR, la unidad de anticipación sabe qué registros leyó la instrucción actual en la etapa de decodificación; para este caso, los registros leídos fueron ACCA y ACCB, cuyos contenidos están almacenados en los campos OP1 y OP2, respectivamente, del registro de segmentación ID/EX. Y gracias a la señal SelRegW, se puede determinar si alguno de los registros leídos en la etapa anterior aún no había sido actualizado; para este caso, el registro ACCA aún no había sido actualizado. Esto significa que el dato guardado en el campo OP1 del registro de segmentación ID/EX no tiene el valor correcto de ACCA, sin embargo, como se estudió anteriormente, el resultado necesario está almacenado en el registro de segmentación EX/WB.

Tras esta situación, la unidad de anticipación, por medio de la señal SelA=1, adelanta dicho resultado hacia la primera entrada de la UPA; de esta manera, el valor de ACCA, guardado en el registro de segmentación ID/EX, es reemplazado por el resultado guardado en el registro de segmentación EX/WB. Note que el segundo operando de la UPA, el contenido del registro ACCB, sí tiene el valor correcto, por lo tanto, es seleccionado directamente del registro de segmentación ID/EX por medio de la señal SelB=0.

Ahora suponga que SelRegR es igual a uno y SelRegW es igual a cuatro. Gracias a la señal SelRegR, la unidad de anticipación sabe que los operandos guardados en el registro de segmentación ID/EX corresponden a los contenidos de los registros ACCA y ACCB; y gracias a la señal SelRegW, sabe que el contenido leído de ACCB aún no había sido actualizado por la instrucción anterior. Por lo tanto, el dato guardado en el campo OP2 del registro de segmentación ID/EX no es el valor correcto de ACCB, pero el resultado almacenado temporalmente en el campo DatoW del registro de segmentación EX/WB, sí lo es.

En este caso, la unidad de anticipación, por medio de la señal SelB=1, adelanta el resultado guardado en el registro de segmentación EX/WB hacia la segunda entrada de la UPA; de esta manera, el valor leído de ACCB es reemplazado por el resultado correcto. Note que el primer operando de la UPA, el contenido del registro ACCA, sí tiene el valor correcto, por lo tanto, es seleccionado directamente del registro de segmentación ID/EX por medio de la señal SelA=0.

El módulo de registros internos

Note que la unidad de anticipación sólo detecta una condición de riesgo por dependencia de datos. Otra condición de riesgo no es posible porque suponemos que el módulo de registros internos suministra el resultado correcto si una instrucción en la etapa 2 intenta leer el mismo registro que otra instrucción en su etapa 4 intenta escribir, entonces, se puede decir que el módulo de registros internos es otra forma de anticipación. De esta manera, la unidad de anticipación y el módulo de registros internos se encargarán de resolver las dos condiciones de riesgos por dependencias de datos que se estudiaron en la sección 7.5.2.

La tabla 7.19 muestra el funcionamiento de la lógica interna del módulo de registros.

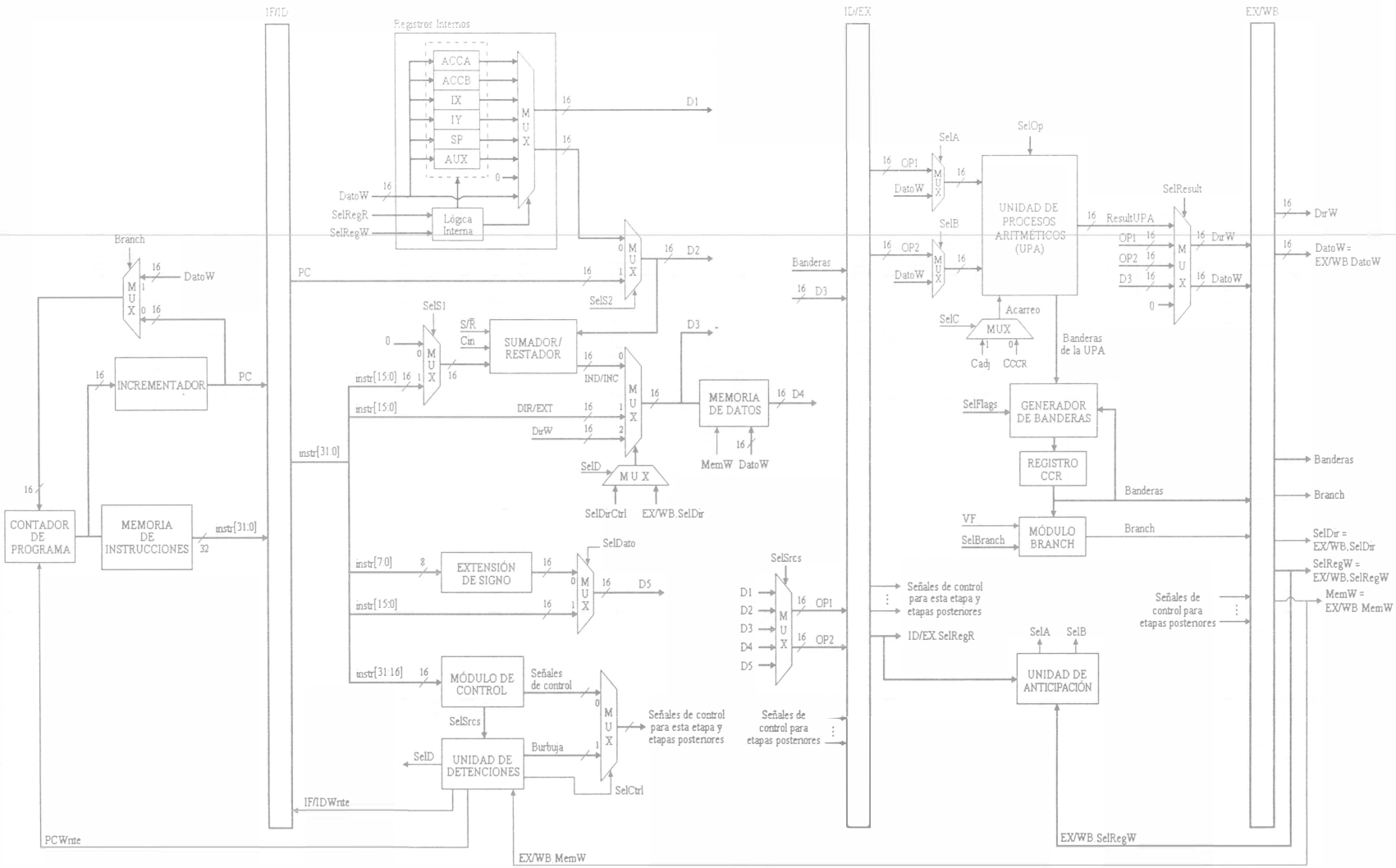


Figura 7.61. Esta arquitectura segmentada emplea el esquema de anticipaciones para eliminar los riesgos por dependencias de datos y el esquema de detenciones para resolver los accesos múltiples a memoria.

Condiciones de entrada		*Señales de salida*		*Condiciones de entrada*		*Señales de salida*	
SelRegR	EX/WB SelRegW	D1	D2	SelRegR	EX/WB SelRegW	D1	D2
1	1	DatoW	ACCB	9	2	0	DatoW
1	4	ACCA	DatoW	A	3	0	DatoW
2	4	DatoW	IX	B	6	0	DatoW
2	2	ACCB	DatoW	C	1	DatoW	SP
3	4	DatoW	IY	C	6	ACCA	DatoW
3	3	ACCB	DatoW	D	4	DatoW	SP
4	1	DatoW	0	D	6	ACCB	DatoW
5	4	DatoW	0	E	2	DatoW	SP
6	1	DatoW	IX	E	6	IX	DatoW
6	2	ACCA	DatoW	F	3	DatoW	SP
7	1	DatoW	IY	F	6	IY	DatoW
7	3	ACCA	DatoW	Combinaciones no		Consultar tablas	
8	5	DatoW	0	presentes en la tabla		7.2 y 7.3	

Tabla 7.19. Lógica interna del módulo de registros. Al detectar un riesgo por dependencia de datos se adelanta el contenido del bus DatoW hacia alguna de las salidas del módulo.

Si una instrucción intenta leer el mismo registro que otra instrucción intenta escribir, entonces, la lógica interna del módulo de registros adelanta el resultado, contenido en el campo DatoW del registro de segmentación EX/WB, hacia alguna de las salidas del módulo, D1 ó D2.

Por ejemplo, suponga que las señales SelRegR y SelRegW valen uno. La señal SelRegR indica que se intentan leer los contenidos de los registros ACCA y ACCB; mientras que la señal SelRegW indica que se intenta escribir el resultado del bus DatoW en el registro ACCA. Dada esta condición, la lógica interna del módulo de registros sabe que el contenido del registro ACCA aún no ha sido actualizado con el resultado del bus DatoW. Por lo tanto, en lugar de leer el contenido del registro ACCA, la lógica interna adelanta el contenido del bus DatoW hacia la primera salida del módulo de registros; es decir, hacia D1, que es por donde se lee el contenido de ACCA según la tabla 7.2. Observe que la segunda salida del módulo de registros, D2, se asigna con el contenido del registro ACCB, ya que éste contiene un valor actualizado. Posteriormente, durante el flanco de subida del reloj, el registro ACCA será actualizado con el valor DatoW.

Algo similar ocurre para la siguiente combinación, SelRegR=1 y SelRegW=4. En este caso, se intentan leer los contenidos de los registros ACCA y ACCB, y se intenta escribir el contenido del bus DatoW en el registro ACCB. Dada esta condición, la lógica interna sabe que ACCB no ha sido actualizado con el resultado de DatoW; por lo tanto, el contenido del bus DatoW es adelantado hacia la salida D2, que es por donde se lee el contenido del registro ACCB según la tabla 7.2. Note que la primera salida del módulo de registros, D1, se asigna con el contenido del registro ACCA, ya que éste contiene un valor actualizado. Finalmente, no olvide que en el flanco de subida del reloj el registro ACCB será actualizado con el valor del bus DatoW.

Para el resto de las combinaciones en la tabla se sigue un razonamiento similar; y en caso de que no se presente un riesgo, el módulo de registros funcionará de acuerdo a las tablas 7.2 y 7.3.

Detenciones debido a accesos múltiples a memoria

Gracias al esquema de anticipaciones, los riesgos por dependencias de datos son resueltos sin retrasos de tiempo. Desafortunadamente, los riesgos debidos a accesos múltiples a memoria no pueden ser anticipados de la misma forma, ya que la memoria de datos de la etapa 2 sólo puede leer ó escribir datos en un instante dado. Por lo tanto, se incorpora una unidad de detenciones en la arquitectura segmentada de la figura 7.61 para resolver los riesgos por accesos múltiples a memoria; recuerde que el funcionamiento de esta unidad de detenciones se explicó en la tabla 7.17.

Control de riesgos por medio de anticipaciones: Ejemplo 2

Nuevamente recurrimos al ejemplo 2 de la sección 7.4 porque muestra los dos tipos de anticipación que se estudiaron:

1. anticipación vía el registro de segmentación EX/WB hacia la UPA; y
2. anticipación vía el registro de segmentación EX/WB hacia el módulo de registros internos

La figura 7.62 presenta el diagrama de múltiples ciclos de reloj para este ejemplo; además, las figuras 7.63 a 7.65 muestran detalladamente los eventos que ocurren en cada etapa de la arquitectura durante los ciclos de reloj CC3 a CC5 de la figura 7.62.

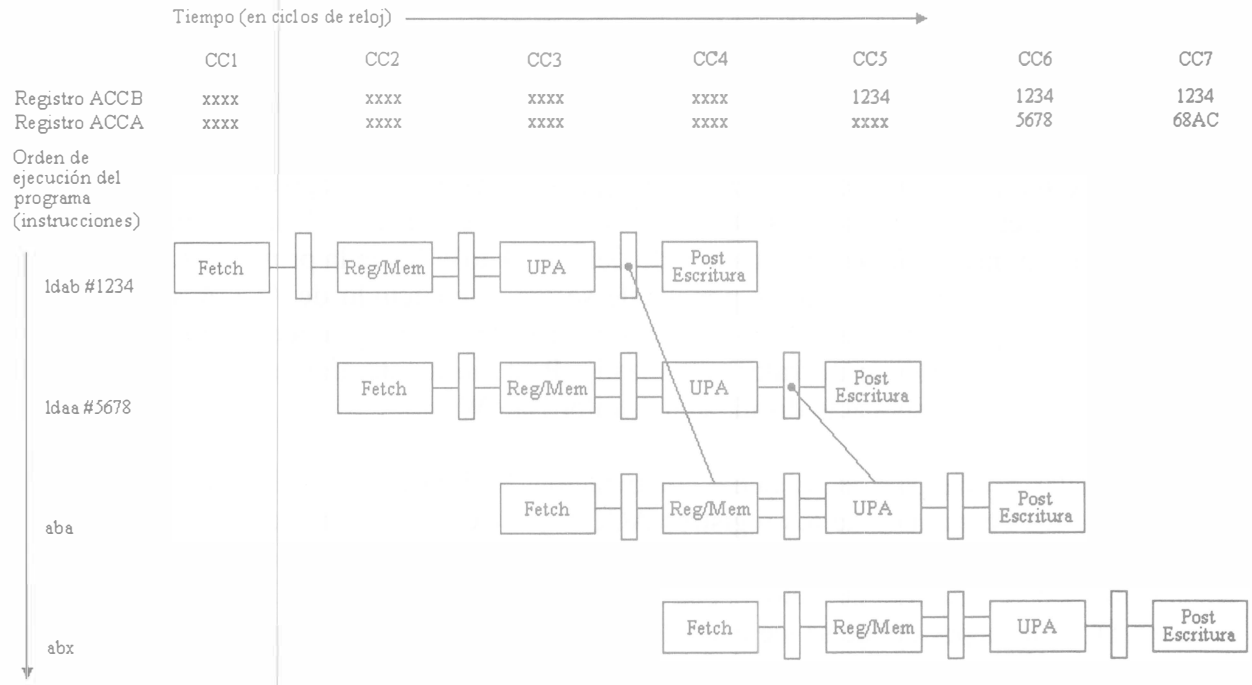


Figura 7.62. Diagrama de múltiples ciclos de reloj para el ejemplo 2. Se utiliza el esquema de anticipación para resolver los riesgos por dependencia de datos.

Observe cuidadosamente las instrucciones en el ciclo de reloj CC4 de la figura 7.62. La instrucción *ldab* se encuentra en la etapa de post-escritura intentando escribir un dato en el registro ACCB; la instrucción *ldaa* se encuentra en la etapa de ejecución; la instrucción *aba* en la etapa de lectura de operandos; y la instrucción *abx* está siendo leída de la memoria de instrucciones.

Note que la instrucción *aba* lee sus operandos en este mismo ciclo de reloj; es decir, lee los contenidos de los registros ACCA y ACCB, los cuales no han sido actualizados con los valores de las instrucciones de carga anteriores. En este momento, la lógica interna del módulo de registros detecta que se intenta leer y escribir el registro ACCB, por lo tanto, el dato que se va a escribir es adelantado del registro de segmentación EX/WB hacia alguna de las salidas del módulo de registros. De esta manera, uno de los operandos empleados por la instrucción *aba*, el contenido del registro ACCB, ya tiene el valor correcto, pero el operando correspondiente al contenido del registro ACCA aún no lo tiene (véase la figura 7.64).

En el siguiente ciclo de reloj, CC5, la instrucción *ldaa* intenta escribir un dato en el registro ACCA, la instrucción *aba* se encuentra en la etapa de ejecución, y la instrucción *abx* en la etapa de lectura de operandos. En este mismo instante, la unidad de anticipación, colocada en la etapa de ejecución, compara los valores de las señales SelRegR y SelRegW de las instrucciones *aba* y *ldaa*, respectivamente. Gracias a estas señales la unidad de anticipación sabe que el contenido del registro ACCA, leído en la etapa anterior, no corresponde al valor actualizado, ya que en el registro de segmentación EX/WB se tiene el resultado y la señal de escritura para este registro. Por lo tanto, el resultado del registro de segmentación EX/WB es anticipado hacia una de las entradas de la UPA, reemplazando el valor de ACCA leído en la etapa anterior. Finalmente la instrucción *aba* tiene los operandos correctos sobre los que operará la UPA (véase la figura 7.65).

7.6 Riesgos por saltos

Hasta el momento hemos limitado nuestro interés a riesgos que involucran operaciones aritméticas y transferencias de datos (tanto a memoria como a registros), pero existe otro tipo de riesgo que se suele presentar debido a los saltos, los cuales son cambios en el flujo de control del programa. Por ejemplo, suponga que tiene la siguiente secuencia de instrucciones:

```
0x0400    bmi 07
0x0401    anda #0013
0x0402    orab #FFFF
0x0403    adda #0010
.....     .....
0x0408    ldaa #1000
0x0409    ldab #8080
```

La primera instrucción, *bmi*, es una instrucción de salto condicional. Esta instrucción revisa el valor de la bandera de negativo del registro de estados, si su valor es igual a uno, entonces se realiza un salto hacia la instrucción ubicada en la localidad de memoria 0x0408, si no, la siguiente instrucción a ejecutar es la de la localidad 0x0401.

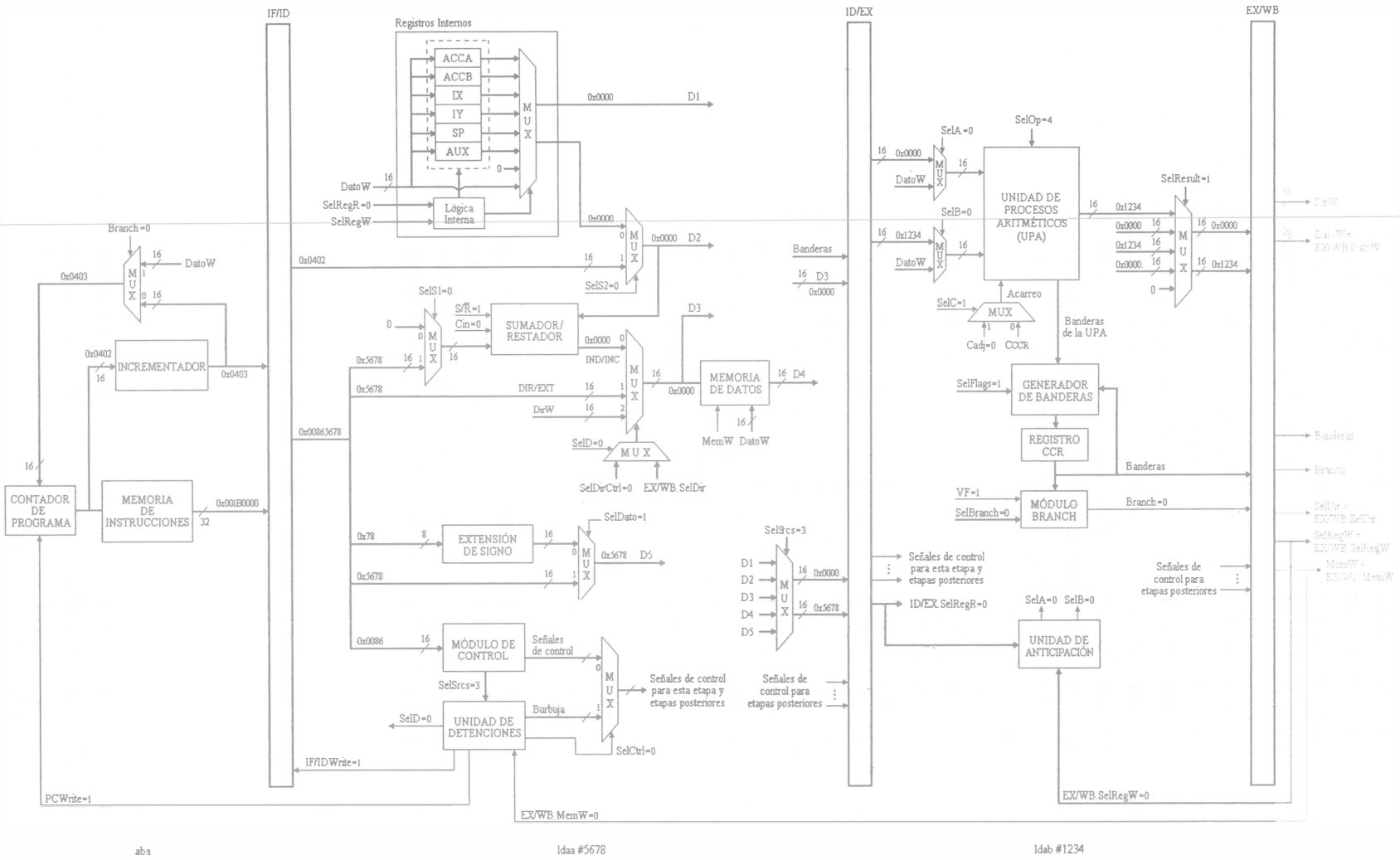


Figura 7.63. Diagrama de segmentación correspondiente al ciclo de reloj 3. Hasta el momento, la ejecución de las instrucciones en el cauce ha transcurrido sin contratiempos. La instrucción *ldab* se encuentra en la etapa de ejecución (etapa 3), la instrucción *ldaa* en la etapa de lectura de operandos (etapa 2) y la instrucción *aba* en la etapa de traer una instrucción de memoria (etapa 1)

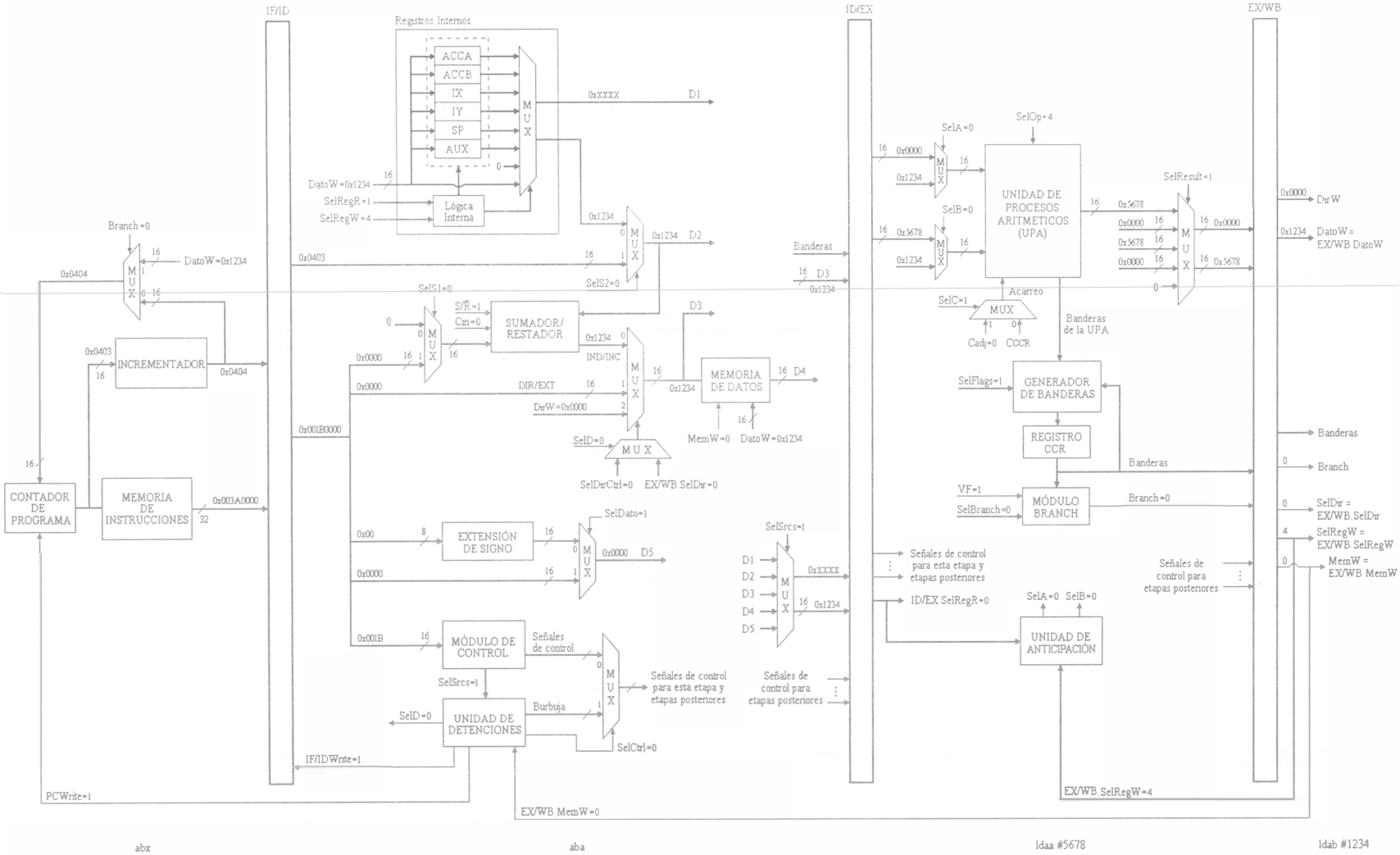


Figura 7.64. Diagrama de segmentación correspondiente al ciclo de reloj 4. La instrucción *ldab* intenta escribir un resultado en el registro ACCB, al mismo tiempo, la instrucción *aba* intenta leer los contenidos de los registros ACCA y ACCB; por lo tanto, el módulo de registros internos adelanta el resultado de la instrucción *ldab* del registro de segmentación EX/WB hacia el bus de salida D2 del módulo de registros. Observe que sólo es adelantado el dato del registro ACCB, por lo que el dato leído del registro ACCA no corresponde al valor de carga de la instrucción *ldaa*. Por otro lado, en el mismo ciclo de reloj, las instrucciones *ldaa* y *abx* están siendo ejecutadas sin contratiempos en las etapas 3 y 1, respectivamente.

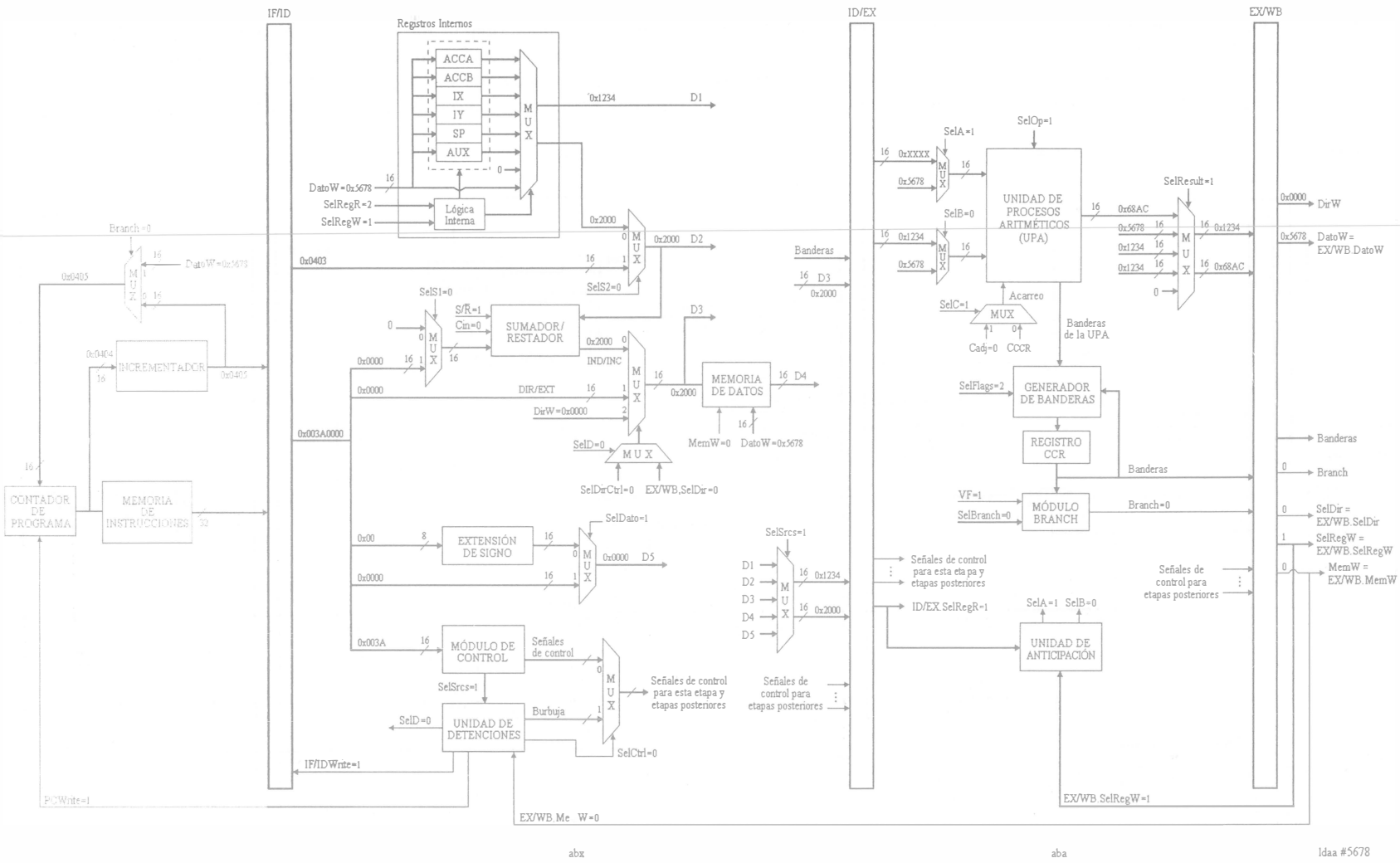


Figura 7.65. Diagrama de segmentación correspondiente al ciclo de reloj 5. La instrucción *ldaa* intenta escribir un resultado en el registro ACCA, en este instante, la unidad de anticipación se da cuenta de que el dato correspondiente al contenido del registro ACCA, que se leyó en la etapa anterior, no es el correcto; por lo tanto, la unidad de anticipación adelanta el resultado del registro de segmentación EX/WB hacia la primera entrada de la UPA, reemplazando el dato no actualizado por el valor correcto. La ejecución del resto de las instrucciones continuará sin problemas hasta que un nuevo riesgo sea detectado.

Recuerde que los saltos son resueltos hasta la última etapa de la segmentación (consulte la sección 7.3.6 para mayor información); en esta última etapa, con base en el valor de la señal Branch, se puede determinar si se realiza el salto o no. Como el salto es resuelto hasta la última etapa de la segmentación, significa que las etapas 3, 2 y 1 deben estar ejecutando las instrucciones *anda*, *oraa* y *adda*, respectivamente. Cuando el salto no es efectuado, la secuencia de ejecución continúa sin contratiempos ejecutando las instrucciones *anda*, *oraa* y *adda*. Pero, si se realiza el salto, el flujo de control del programa debe ser transferido a la localidad de memoria 0x0408; por lo tanto, las instrucciones que se ejecutaban en las etapas anteriores deben ser descartadas del cauce. Entonces surgen algunas preguntas: ¿cómo descartar esas instrucciones del cauce si el salto se realiza? ó ¿cómo evitar su ejecución para no tener problemas?. Las respuestas a estas preguntas se encontrarán al estudiar los siguientes dos métodos que nos permiten resolver los riesgos por saltos:

1. Detenciones
2. Suponer que el salto no es realizado

7.6.1 DETENCIONES

Este método plantea que la solución para eliminar los riesgos por saltos es detenerse hasta que el salto es resuelto, es decir, detener las instrucciones posteriores al salto hasta que se concluya la instrucción de salto. Observe que este método es similar al método de detenciones que se utilizó para resolver los riesgos por dependencias de datos, en el cual se detenían las instrucciones posteriores hasta resolver el riesgo. Recuerde que la desventaja de este método es la penalización de varios ciclos de reloj que se presenta cuando el salto no se realiza. Esta penalización se muestra en el diagrama de múltiples ciclos de reloj de la figura 7.66.

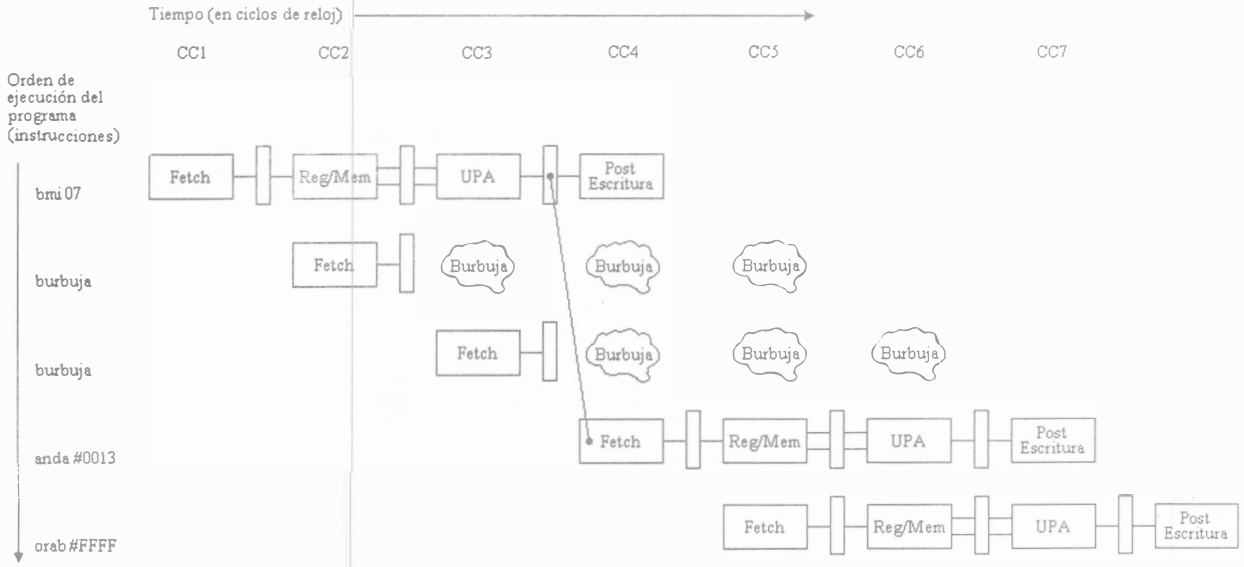


Figura 7.66. Diagrama de múltiples ciclos de reloj en donde se utiliza el esquema de detenciones para resolver los riesgos por saltos. Este esquema espera a que la instrucción de salto termine de ejecutarse para después continuar con la ejecución de las demás instrucciones; note que en este caso el salto no se realiza.

7.6.2 SUPONER QUE EL SALTO NO ES REALIZADO

Una mejora al esquema de detenciones es suponer que el salto no se realiza, por lo tanto, se continúa avanzando en la ejecución del flujo secuencial de instrucciones. La figura 7.67 muestra esta idea utilizando el ejemplo planteado al inicio de esta sección.

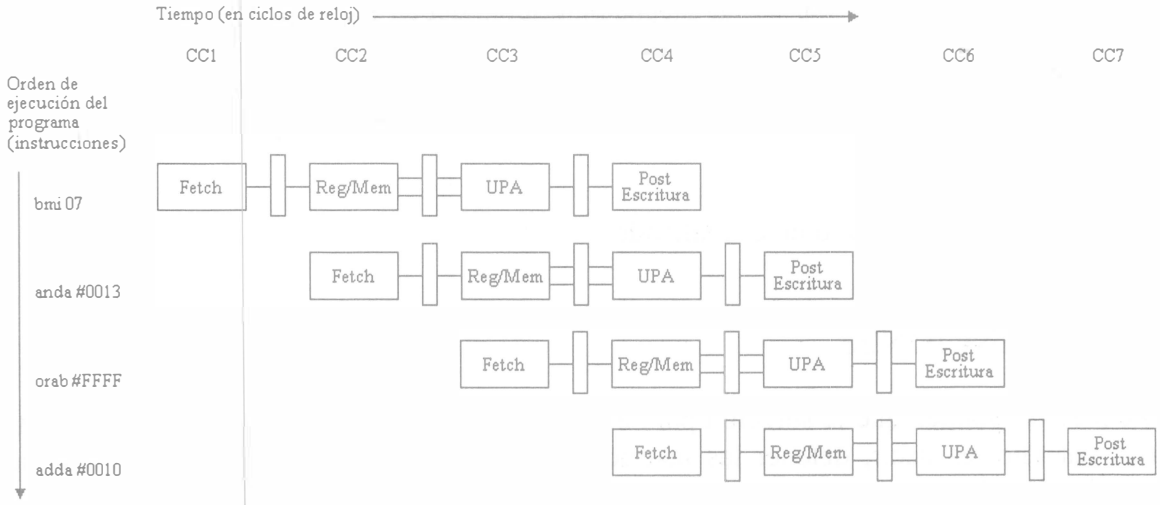


Figura 7.67. En este segundo esquema, si el salto no se realiza, no se desperdicia tiempo esperando a que el cauce vuelva a llenarse con las instrucciones siguientes.

En caso de que se realice el salto, las instrucciones posteriores a la instrucción de salto, las cuales están siendo ejecutadas, son descartadas (se limpia el cauce); y la ejecución de instrucciones reinicia a partir de la dirección destino del salto. Esta idea se expresa en la figura 7.68.

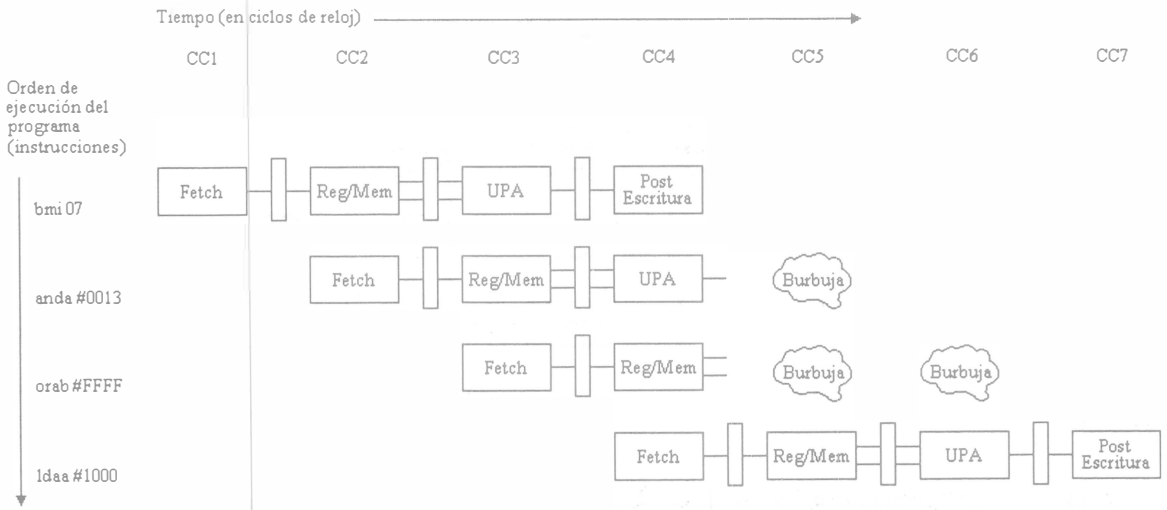


Figura 7.68. Si el salto se realiza, las instrucciones posteriores al salto que están siendo ejecutadas (*anda* y *oraa*) son descartadas, y el flujo de control de programa se transfiere a la dirección destino del salto (*ldaa*).

El método que se utiliza para descartar las instrucciones (limpiar el cauce) es muy parecido al método de generación de burbujas. Estas burbujas son insertadas en las últimas etapas de la segmentación, es decir, los contenidos de los registros de segmentación ID/EX y EX/WB son reemplazados con nuevas señales de control, obteniendo el mismo efecto de la instrucción *nop*.

Nuevamente la unidad encargada de detectar los saltos y descartar las instrucciones cuando se realiza el salto es la unidad de detenciones. Tres nuevas señales de control son agregadas a esta unidad: Branch, que indica cuándo realizar el salto; isBranch, que indica si la instrucción que se está ejecutando en la etapa 4 corresponde a una instrucción de salto; y EXFlush, que coloca a ceros el contenido del registro de segmentación EX/WB. Adicionalmente, la unidad de detenciones incorpora las señales de control necesarias para detectar y eliminar los riesgos debidos a accesos múltiples a memoria.

El funcionamiento de la nueva unidad de detenciones se resume en la tabla 7.20; y el diagrama de la figura 7.69 muestra la nueva arquitectura segmentada del 68HC11 utilizando el esquema planteado en la sección 7.6.2 para resolver los riesgos por saltos.

Condiciones de entrada				*Señales de salida*				
SelSrcs	EX/WB MemW	EX/WB Branch	EX/WB isBranch	PCWrite	IF/IDWrite	SelD	SelCtrl	EXFlush
2, 4, 6	1	0	0	0	0	1	1	0
No Importa	0	1	1	1	1	0	1	1
Combinaciones no presentes en la tabla				1	1	0	0	0

Tabla 7.20. Condiciones para detectar los riesgos por accesos múltiples a memoria y los riesgos por saltos, y señales de salida generadas por la unidad de detenciones para eliminar estos riesgos.

Finalmente, las figuras 7.70 y 7.71 muestran detalladamente los eventos que ocurren en la arquitectura durante los ciclos de reloj CC4 y CC5 de la figura 7.68.

En el ciclo de reloj CC4, la instrucción *bmi* se encuentra en la etapa de post-escritura intentando ejecutar un salto a la dirección 0x0408; en consecuencia, la instrucción contenida en dicha dirección, ldaa #1000, es leída de la memoria de instrucciones de la etapa 1. Como el salto sí es realizado, entonces las instrucciones *anda* y *oraa* de las etapas 3 y 2 respectivamente, deben ser descartadas por la unidad de detenciones. Esta unidad detecta el riesgo por salto y genera las señales de control necesarias para limpiar los registros de segmentación ID/EX y EX/WB durante el flanco de subida del reloj (véase la figura 7.70).

En el ciclo de reloj CC5, las instrucciones *anda* y *oraa* ya fueron eliminadas del cauce, en su lugar son insertadas dos burbujas en las etapas 4 y 3. Observe que el resultado de limpiar los registros de segmentación ID/EX y EX/WB es la generación de estas dos burbujas. Por otra parte, la secuencia de ejecución continúa a partir de la dirección de salto, por ello, ahora la instrucción *ldaa* se encuentra en la etapa 2 y la instrucción *ldab* en la etapa 1 (véase la figura 7.71).

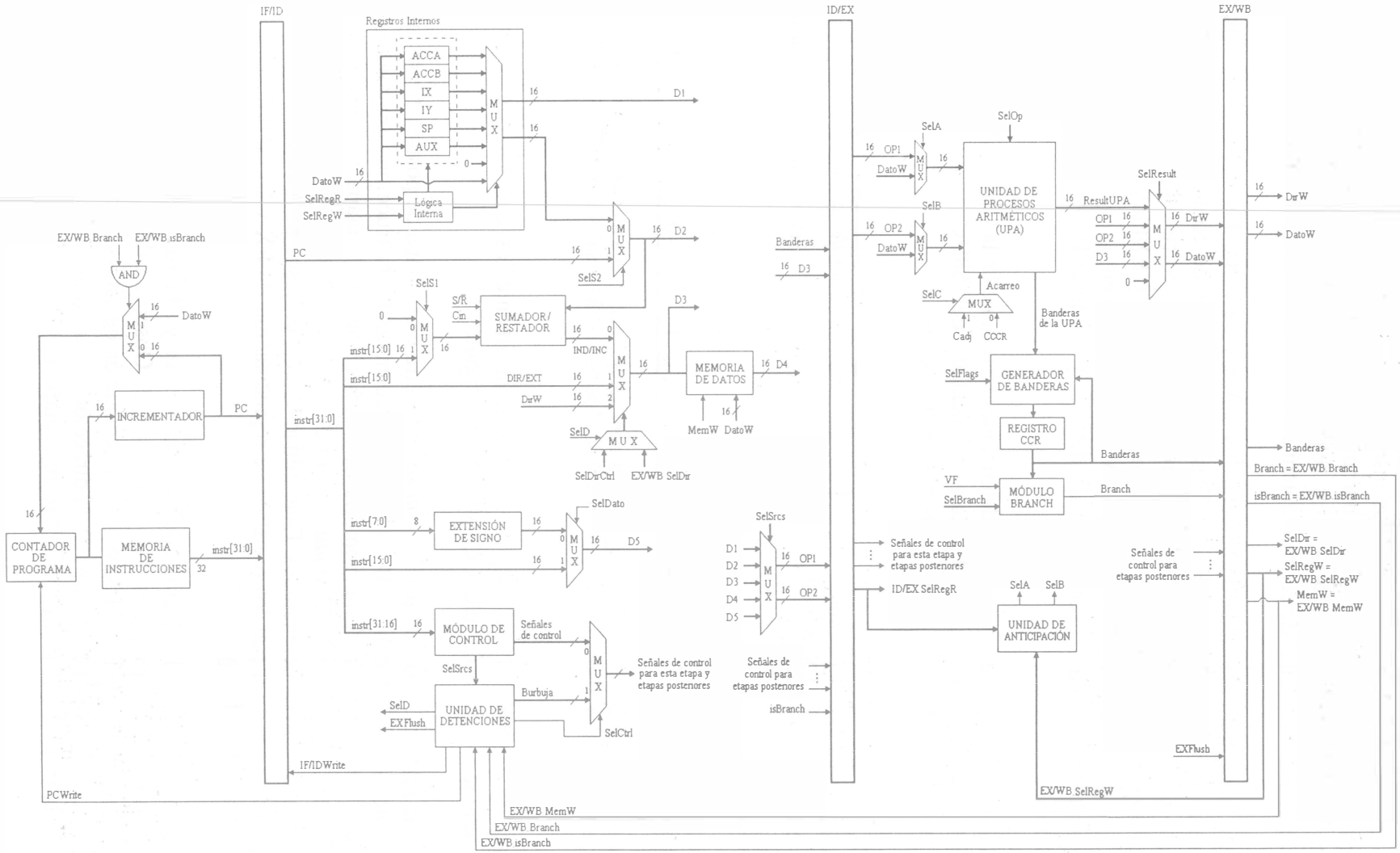


Figura 7.69. Esta arquitectura segmentada utiliza el esquema de anticipaciones para eliminar los riesgos por dependencias de datos, el esquema de detenciones para resolver los accesos múltiples a memoria, y el esquema de saltos de la sección 7.6.2 para resolver los riesgos por saltos.

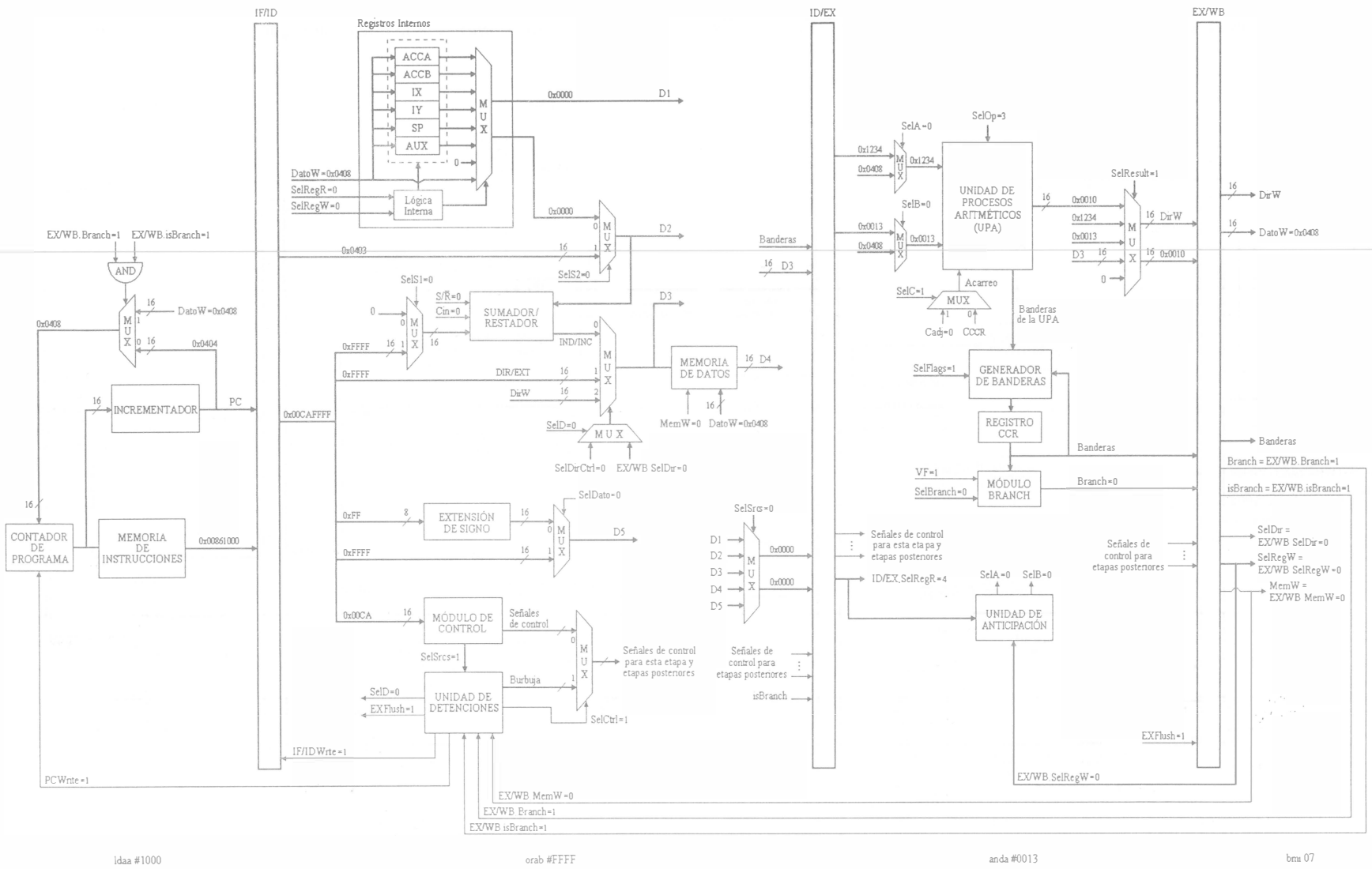


Figura 7.70. Ciclo de reloj 4. La instrucción *bmi* ejecuta un salto a la dirección 0x0408, leyendo de memoria a la instrucción *ldaa*. Como sí se realiza el salto, la unidad de detenciones detecta el riesgo y prepara las señales de control necesarias para descartar a las instrucciones *anda* y *oraa*.

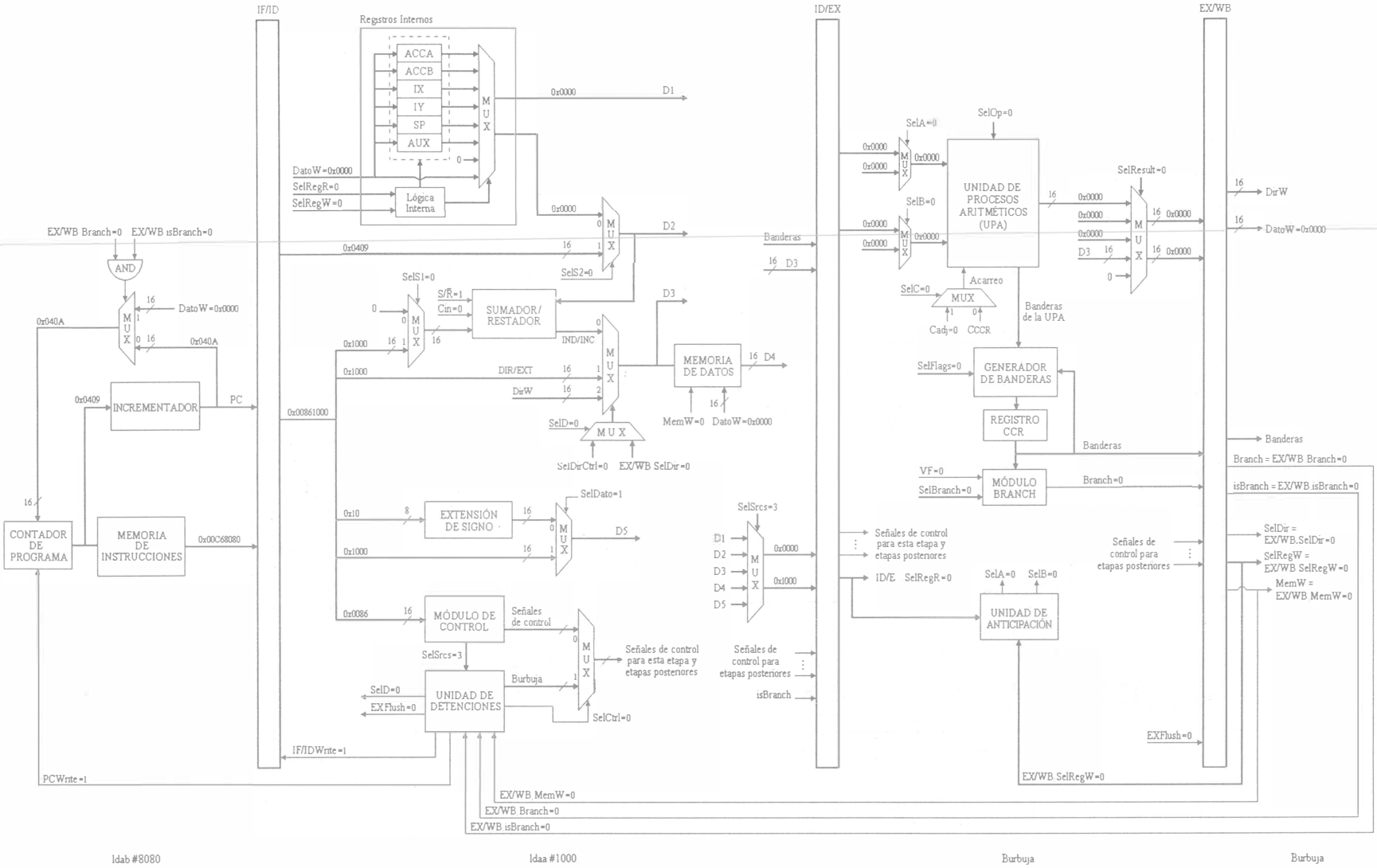


Figura 7.71. Ciclo de reloj 5. Las instrucciones *anda* y *oraa* ya fueron eliminadas del cauce, en su lugar fueron insertadas dos burbujas en las etapas 4 y 3. La secuencia de ejecución de instrucciones continúa a partir de la dirección de salto.

7.7 INTERRUPCIONES

Como se ha visto a lo largo de este capítulo, el control es el aspecto más complicado en el diseño de un procesador, pero la parte más ardua del control es la implantación de las interrupciones. Una interrupción es un evento que proviene del exterior del procesador y provoca un cambio inesperado en el flujo de control del programa. Las interrupciones son utilizadas por los dispositivos de entrada/salida para comunicarse con el procesador, y de manera similar a los saltos, cambian el flujo normal de ejecución de las instrucciones.

Inicialmente, las interrupciones se crearon para manejar ciertos eventos inesperados, por ejemplo, las peticiones de servicio provenientes de los dispositivos de entrada/salida. Actualmente, este mecanismo se ha extendido para manejar también los eventos que se generan internamente en el procesador, como por ejemplo, desbordamientos aritméticos, instrucciones indefinidas, entre otras. Los diseñadores de hardware utilizan el término "excepción" para referirse a estos tipos de interrupciones internas.

Para nuestro caso particular, el único tipo de interrupciones que se atenderán serán las solicitadas por los dispositivos de entrada/salida. Para su implantación, se han colocado en la arquitectura dos nuevos módulos: el módulo PCTrap y la unidad de interrupciones. La figura 7.72 incorpora el hardware necesario para el manejo de este tipo de interrupciones.

El módulo PCTrap está compuesto por un registro de 16 bits, el cual almacena la dirección de regreso de la interrupción. Recuerde que antes de cambiar el flujo de control del programa hacia la rutina de atención a la interrupción, el procesador debe guardar la dirección de la próxima instrucción a ejecutar, con el fin de regresar a ejecutar esa instrucción una vez que la interrupción haya sido atendida. De esta manera, la señal de control SavePC guarda en PCTrap una copia de la dirección de regreso, la cual es obtenida del campo PC del registro de segmentación IF/ID.

La unidad de interrupciones se encarga de informarle al procesador cuándo un dispositivo de entrada/salida necesita atención. Si un dispositivo externo requiere de los servicios del procesador, basta que active la línea $\overline{IRQ}$ ó $\overline{XIRQ}$; en caso de que ambas líneas estén activadas, la interrupción $\overline{XIRQ}$ tendrá prioridad sobre la interrupción $\overline{IRQ}$. Si el procesador puede atender a la interrupción, la unidad de interrupciones proporciona una dirección de salto a la localidad de memoria en donde se encuentra la rutina de atención a la interrupción. Un aspecto de gran importancia que permite simplificar el diseño de la unidad de interrupciones es que el procesador no maneja interrupciones anidadas, es decir, mientras se esté atendiendo una interrupción no será posible aceptar otra.

Las condiciones de entrada para la unidad de interrupciones están dadas por las líneas EnaINT, isBranch, Branch, $\overline{IRQ}$ y $\overline{XIRQ}$, las cuales se describen a continuación.

- Señal de entrada EnaINT. Cuando la unidad de interrupciones acepta atender una petición de servicio, ésta es deshabilitada con el fin de que no puedan ser atendidas otras interrupciones hasta que la interrupción en curso termine su ejecución. La señal EnaINT, proveniente del registro de segmentación EX/WB, permite habilitar nuevamente a la unidad de interrupciones una vez que termina de atender la interrupción en curso, es decir, cuando se

ejecuta una instrucción de regreso de interrupción (RTI). La señal EnaINT es generada por la unidad de control, y es activada sólo cuando se ejecuta una instrucción de regreso de interrupción (RTI).

- Señal de entrada isBranch. Esta señal le informa a la unidad de interrupciones que la instrucción que se está ejecutando en la última etapa de la segmentación corresponde a una instrucción de salto. Es importante conocer el estado de esta señal porque en algunos casos no está permitido ejecutar una interrupción si un salto está en progreso. Recuerde que la señal isBranch proviene del registro de segmentación EX/WB; dicha señal es generada por la unidad de control y es activada sólo cuando la instrucción que se decodifica es una instrucción de salto.

- Señal de entrada Branch. Si la instrucción que se ejecuta en la última etapa de la segmentación es una instrucción de salto, entonces, la señal Branch le informa a la unidad de interrupciones si en verdad se realiza el salto o no. La señal Branch también proviene del registro de segmentación EX/WB, y es generada por el módulo Branch ubicado en la etapa de ejecución. Una interrupción no es atendida por la unidad de interrupciones si se presenta una condición de salto al mismo tiempo, es decir, no es posible atender una interrupción si el procesador está ejecutando una instrucción salto en la última etapa de la segmentación. Esto significa que la ejecución de un salto tendrá mayor prioridad que la atención a una interrupción externa. Una vez terminado el salto, la unidad de interrupciones podrá atender al dispositivo causante de la interrupción.

- Señal de entrada IRQ. La señal IRQ es activada cuando el dispositivo externo conectado a esta línea requiere de la atención del procesador.

- Señal de entrada XIRQ. La señal XIRQ es activada cuando el dispositivo externo conectado a esta línea requiere de la atención del procesador. La interrupción $\overline{\text{XIRQ}}$ tiene mayor prioridad que la interrupción IRQ.

Una vez validada la condición de entrada, la unidad de interrupciones contesta a través de las líneas SelINT y SavePC, las cuales se describen a continuación.

- Señal de salida SelINT. La señal SelINT selecciona la procedencia de una dirección de salto. Si SelINT=1, se selecciona la dirección DirINT que corresponde a la dirección de inicio de la rutina de atención a la interrupción. En cambio, si SelINT=0, la dirección que se selecciona proviene del incrementador, o bien, del bus DatoW. La dirección de inicio de la rutina de interrupción, DirINT, es proporcionada por la unidad de interrupciones.

- Señal de salida SavePC. Una vez que la unidad de interrupciones decide atender a un dispositivo externo, es necesario guardar la dirección en memoria de la siguiente instrucción a ejecutar, para que una vez atendida la interrupción, el procesador continúe ejecutando el programa a partir de esa instrucción. La señal SavePC habilita al registro PCTrap para guardar la dirección de regreso de la interrupción.

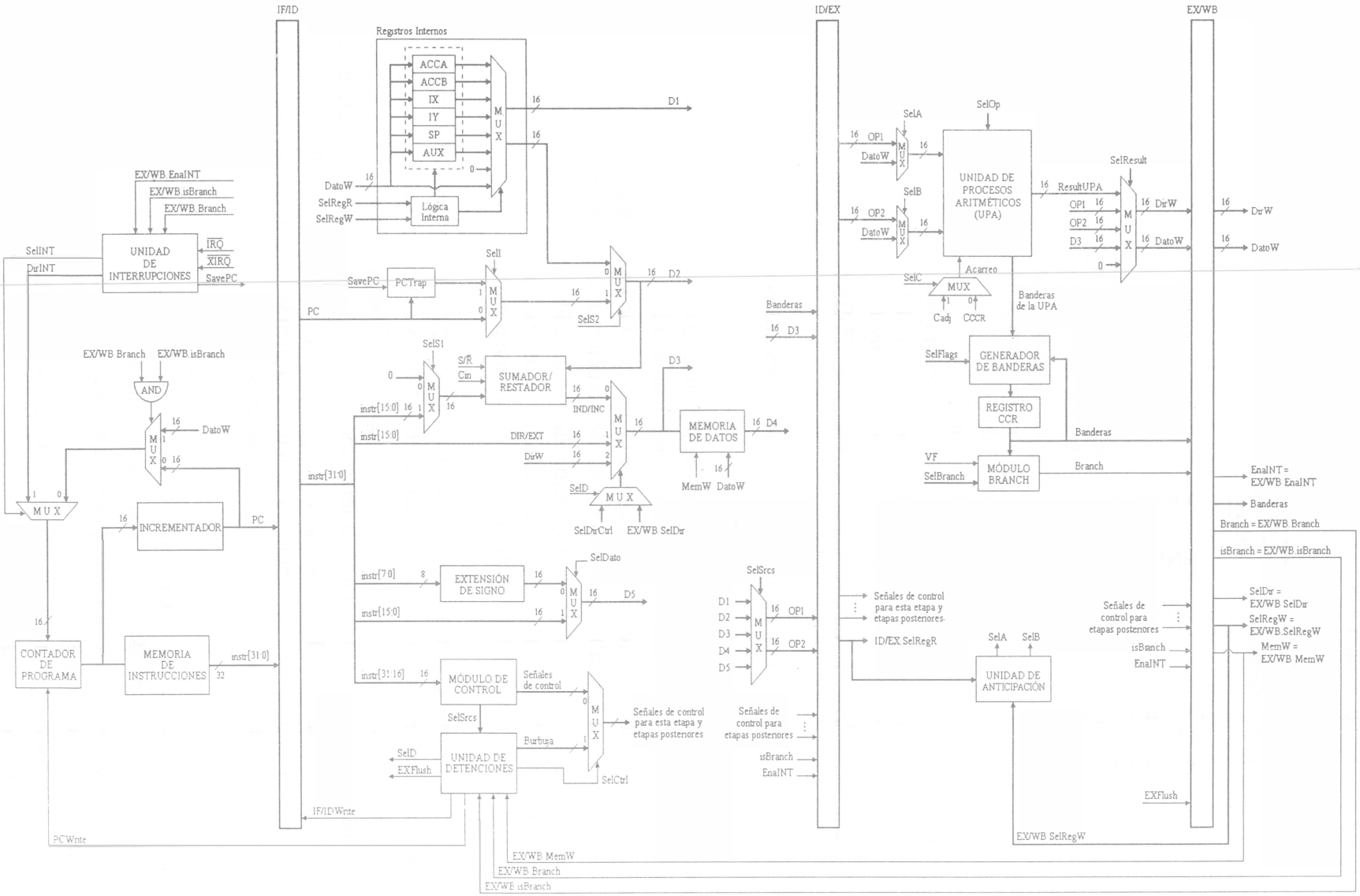


Figura 7.72. Esta arquitectura utiliza el esquema de anticipaciones para eliminar los riesgos por dependencias de datos, el esquema de detenciones para resolver los accesos múltiples a memoria, y el esquema de saltos de la sección 7.6.2 para resolver los riesgos por saltos. Adicionalmente, incluye una unidad de interrupciones para atender peticiones de servicio de dispositivos externos.

La tabla 7.21 muestra la lógica interna de la unidad de interrupciones.

Condiciones de entrada							*Señales de salida*		*Comentario*
EX/WB EnaINT	EX/WB Branch	EX/WB isBranch	$\overline{IRQ}$	$\overline{XIRQ}$			SelINT	SavePC	
0	x	0	x	0			1	1	Atiende interrupción [A.1]
0	x	0	0	1			1	1	Atiende interrupción [A.2]
0	0	1	x	0			1	1	Atiende interrupción [B.1]
0	0	1	0	1			1	1	Atiende interrupción [B.2]
0	1	1	x	0			0	0	Salto [C.1]
0	1	1	0	1			0	0	Salto [C.1]
0	x	x	1	1			0	0	No hay interrupción
1	0	x	x	x			0	0	Condición inválida
1	1	0	x	x			0	0	Condición inválida
1	1	1	x	x			0	0	Regreso de interrupción [D.1]

El valor lógico 'x' significa no importa

Tabla 7.21. Funcionamiento de la unidad de interrupciones.

A.1. Si la unidad de interrupciones está habilitada para atender peticiones de servicio, entonces, ésta da prioridad en el servicio al dispositivo conectado a la línea $\overline{XIRQ}$. Observe que con SavePC=1 se guarda la dirección de regreso de la interrupción en el registro PCTrap, y con SelINT=1 se realiza un salto hacia la rutina de atención a la interrupción $\overline{XIRQ}$. La dirección de inicio de esta rutina es proporcionada por la unidad de interrupciones a través del bus DirINT de 16 bits. Una vez que la unidad de interrupciones comienza a atender una interrupción, ésta no puede atender otra hasta que la primera haya finalizado.

A.2. Si la unidad de interrupciones está habilitada para atender peticiones de servicio, entonces, ésta atiende al dispositivo conectado a la línea $\overline{IRQ}$. Observe que con SavePC=1 se guarda la dirección de regreso de la interrupción en el registro PCTrap, y con SelINT=1 se realiza un salto hacia la rutina de atención a la interrupción $\overline{IRQ}$. La dirección de inicio de esta rutina es proporcionada por la unidad de interrupciones a través del bus DirINT de 16 bits. Una vez que la unidad de interrupciones comienza a atender una interrupción, ésta no puede atender otra hasta que la primera haya finalizado.

B.1. En esta condición ocurren dos eventos al mismo tiempo: 1) existe una petición de interrupción, y 2) se está ejecutando una instrucción de salto. Como la instrucción de salto no realiza el salto, la unidad de interrupciones puede atender la interrupción $\overline{XIRQ}$ siempre y cuando no haya otra interrupción en ejecución. Observe que con SavePC=1 se guarda la dirección de regreso de la interrupción en el registro PCTrap, y con SelINT=1 se realiza un salto hacia la rutina de atención a la interrupción $\overline{XIRQ}$. La dirección de inicio de esta rutina es proporcionada por la unidad de interrupciones a través del bus DirINT de

16 bits. Una vez que la unidad de interrupciones comienza a atender una interrupción, ésta no puede atender otra hasta que la primera haya finalizado.

B.2. En esta condición ocurren dos eventos al mismo tiempo: 1) existe una petición de interrupción, y 2) se está ejecutando una instrucción de salto. Como la instrucción de salto no realiza el salto, la unidad de interrupciones puede atender la interrupción $\overline{IRQ}$ siempre y cuando no haya otra interrupción en ejecución. Observe que con SavePC=1 se guarda la dirección de regreso de la interrupción en el registro PCTrap, y con SelINT=1 se realiza un salto hacia la rutina de atención a la interrupción $\overline{IRQ}$. La dirección de inicio de esta rutina es proporcionada por la unidad de interrupciones a través del bus DirINT de 16 bits. Una vez que la unidad de interrupciones comienza a atender una interrupción, ésta no puede atender otra hasta que la primera haya finalizado.

C.1. En esta condición ocurren dos eventos al mismo tiempo: 1) existe una petición de interrupción, y 2) se está ejecutando un salto. Debido a que la instrucción de salto sí realiza el salto, entonces, la unidad de interrupciones no puede atender la interrupción, ya que de hacerlo, se guardaría una dirección errónea de regreso de la interrupción. Por lo tanto, para evitar este tipo de inconvenientes, la unidad de interrupciones le otorga mayor prioridad a los saltos que a las interrupciones.

D.1. La interrupción está terminando su ejecución. Cuando se presenta esta condición, la instrucción de regreso de interrupción, RTI, ejecuta un salto hacia la dirección de regreso de la interrupción. En este mismo instante, la unidad de interrupciones vuelve a habilitarse para permitir la atención de nuevas interrupciones, sin embargo, note que no se puede comenzar la atención de una nueva interrupción en este mismo ciclo de reloj, ya que el salto que ejecuta la instrucción RTI tiene mayor prioridad que la atención a una nueva interrupción.

Además del módulo PCTrap y de la unidad de interrupciones, fue anexado un multiplexor a la salida del módulo PCTrap. Este multiplexor utiliza la señal de control SelI, la cual es generada por el módulo de control y permite seleccionar la dirección guardada en el registro PCTrap, o bien, la dirección guardada en el campo PC del registro de segmentación IF/ID. La señal SelI será colocada a uno cuando se desee recuperar la dirección de regreso de la interrupción, es decir, durante la ejecución de una instrucción RTI.

Atención a la interrupción

Considere el siguiente ejemplo.

```
0x0400    ldaa #1234
0x0401    ldab #5678
0x0402    aba
0x0403    abx
0x0404    oraa #1000
....      ....   ....
```

Como es sabido, antes de atender una petición de interrupción se deben guardar los contenidos de los registros IY, IX, ACCA y ACCB, así como la dirección de regreso de la interrupción. La dirección de regreso de la interrupción se guarda en el registro PCTrap, mientras que los contenidos de los registros se guardan en el área de la pila de la memoria de datos. Una vez salvado el estado de la arquitectura se comenzarán a ejecutar las instrucciones propias de la interrupción. La rutina de atención a la interrupción tiene la siguiente apariencia:

```
Rutina_INT:

0xFD00      push_IY
0xFD01      push_IX
0xFD02      push_ACCA
0xFD03      push_ACCB
            ....    ....
0xFEFC      pull_ACCB
0xFEFD      pull_ACCA
0xFEFE      pull_IX
0xFEFF      pull_IY
0xFF00      rti
0xFF01      instrucción_x1
0xFF02      instrucción_x2
            ....    ....
```

Como puede observar, al inicio de la rutina de atención a la interrupción hay varias instrucciones *push*, las cuales guardan en la pila los contenidos de los registros del procesador. Después de las instrucciones *push* es colocado el código que atiende a la interrupción; y finalmente, antes de ejecutar la instrucción de regreso de interrupción, son restaurados los registros guardados en la pila por medio de las instrucciones *pull*.

La figura 7.73 muestra los eventos que acontecen cuando se atiende una llamada a interrupción.

El comienzo de la interrupción ocurre en el ciclo de reloj CC4. En este mismo instante, la instrucción *ldaa* está en ejecución en la última etapa de la segmentación sin ocasionar conflictos con la atención a las interrupciones; por lo tanto, la unidad de interrupciones puede atender la interrupción solicitada. En consecuencia, el módulo PCTrap guarda una copia de la dirección de regreso de la interrupción, la cual está almacenada temporalmente en el campo PC del registro de segmentación IF/ID. Observe que para nuestro ejemplo, la dirección de regreso de la interrupción, 0x0403, corresponde a la instrucción *abx*. Esto significa que antes de comenzar a ejecutar las instrucciones de la rutina de interrupción, las instrucciones *ldaa*, *ldab* y *aba* terminarán su ejecución.

En el mismo ciclo de reloj, en la etapa 1, el registro PC es cargado con la dirección de inicio de la rutina de interrupción, 0xFD00. Dicha dirección es generada por la unidad de interrupciones y seleccionada por medio de la señal SelINT=1; por lo tanto, también en el ciclo de reloj CC4, se comienzan a traer las primeras instrucciones de la rutina de interrupción. De acuerdo a la rutina de interrupción que se estudió con anterioridad, las primeras instrucciones que se ejecutarían serían las instrucciones *push*.

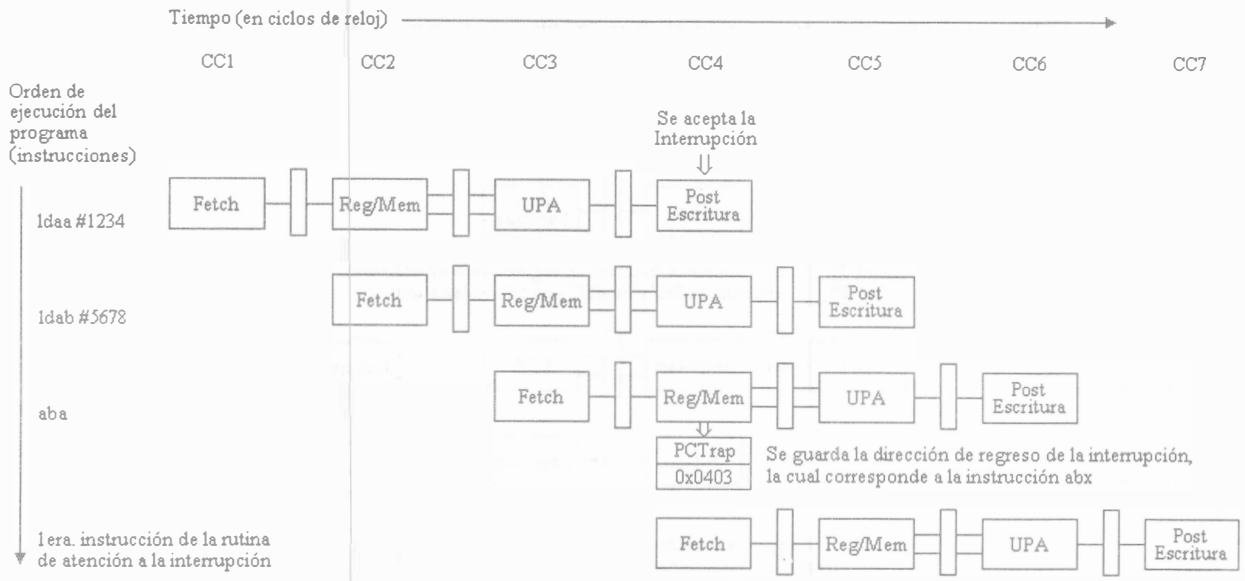


Figura 7.73. Atención a la interrupción.

Regreso de interrupción

Una vez atendido el dispositivo causante de la interrupción y restaurados los registros de la arquitectura, es ejecutada la instrucción de regreso de interrupción, *rti*.

Cuando la instrucción *rti* es decodificada, la unidad de control genera dos señales muy importantes: SelI que selecciona la dirección de regreso de la interrupción, y EnaINT que habilitará nuevamente a la unidad de interrupciones para la atención de nuevas interrupciones. No olvide que la unidad de interrupciones es deshabilitada cada vez que se inicia la atención a una interrupción, de esta manera, es imposible aceptar nuevas interrupciones si hay una interrupción en proceso.

La figura 7.74 muestra los eventos que acontecen cuando se ejecuta una instrucción de regreso de interrupción. Observe que el regreso de la interrupción ocurre hasta que la instrucción *rti* está en la última etapa de la segmentación, es decir, hasta el ciclo de reloj CC4. En ese momento, la unidad de interrupciones vuelve a habilitarse, y mientras, la arquitectura ejecuta un salto incondicional hacia la dirección de regreso de la interrupción. De manera que en el ciclo de reloj CC4, el PC se carga con la dirección 0x0403 para comenzar la búsqueda en memoria de la instrucción *abx*.

Suponiendo que en el ciclo de reloj CC4 ocurriera otra interrupción, ésta no podría ser atendida, ya que en el mismo instante, la instrucción *rti* estaría ejecutando un salto. No está de sobra recordar que la unidad de interrupciones le otorga mayor prioridad a los saltos que a las interrupciones. Por último, las instrucciones que se ejecutaban después de la instrucción *rti*, *instrucción_x1* e *instrucción_x2*, son descartadas de la segmentación gracias a la unidad de detenciones y al salto incondicional que ejecuta la instrucción *rti*.

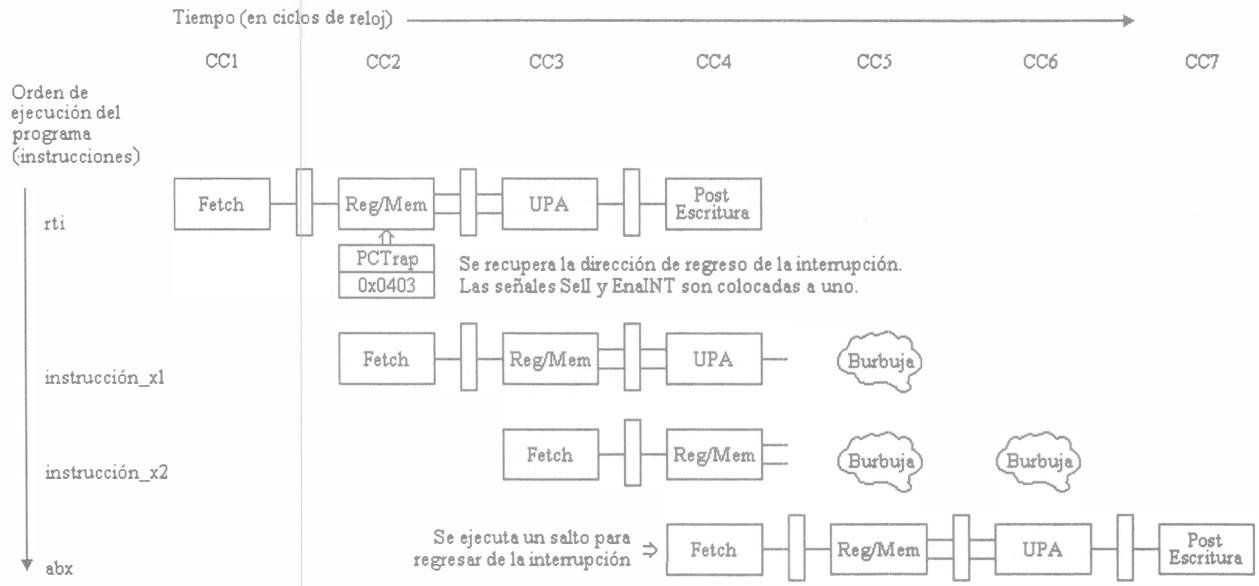


Figura 7.74. Regreso de interrupción.

Como se habrá dado cuenta, el manejo y control de las interrupciones en la segmentación no son tareas triviales, pues hay muchas condiciones que se deben considerar para poder realizar las decisiones correctas. Aún así, el manejo de las interrupciones externas son mucho más flexibles de implantar en hardware que las interrupciones internas, las cuales requieren de rigurosos métodos de control para saber exactamente qué instrucción causa la excepción.

PROBLEMAS

1. Utilice la arquitectura de la figura 7.12 para mostrar el comportamiento de las siguientes instrucciones en cada una de las etapas del "pipeline". Emplee diagramas de un sólo ciclo de reloj para cada una de las etapas y explique los eventos que ocurren en cada una de ellas.

a.	0x00	0x89	0xjj	0xkk	Instrucción adca (acceso inmediato)
b.	0x00	0x08	0x00	0x00	Instrucción inx (acceso inherente)
c.	0x00	0x2B	0x00	0xrr	Instrucción bmi (acceso relativo)
d.	0x00	0x7E	0xhh	0xll	Instrucción jmp (acceso extendido)

2. Construya una tabla con las señales de control para las siguientes instrucciones.

Instrucciones		*Señales de Control*		
		Etapa 2	Etapa 3	Etapa 4
abx (INH)	003A			
adca (IMM)	0089			
asla (INH)	0048			
bmi (REL)	002B			
clc (INH)	000C			
inx (INH)	0008			
jmp (EXT)	007E			
rol (EXT)	0079			
staa (EXT)	00B7			

3. Construya una tabla con las señales de control para las siguientes instrucciones.

Instrucciones		*Señales de Control*		
		Etapa 2	Etapa 3	Etapa 4
bcc (REL)	0024			
deca (INH)	004A			
eora (IMM)	0088			
inc (EXT)	007C			
iny (INH)	1808			
lslb (INH)	0058			
oraa (DIR)	009A			
rola (INH)	0049			
tab (INH)	0016			

4. Demuestre que las instrucciones *psha* y *pula* no pueden implantarse de manera directa en la arquitectura segmentada de la figura 7.12. Utilice diagramas de un sólo ciclo de reloj para mostrar sus conclusiones, y explique qué ocurre en cada uno ellos.

   Recuerde que la instrucción *psha* inserta el contenido del registro ACCA en la pila, mientras que la instrucción *pula* extrae un dato de la pila y lo guarda en el registro ACCA. Suponga que el área de memoria de la pila se encuentra en la parte alta de la memoria de datos, la cual es direccionada por medio del registro SP (stack pointer, apuntador de pila).

5. Conteste las siguientes preguntas con base en las conclusiones del problema anterior.

   a) ¿Las instrucciones *pshb* y *pulb* puede implantarse de manera directa?, ¿ocurre lo mismo para cualquier instrucción push ó pull que se intentara implantar?
   b) Sugiera una alternativa para poder implantar cualquier instrucción push y pull en la arquitectura segmentada de la figura 7.12. Demuestre que su alternativa funciona.

6. Utilice el diagrama de la figura 7.12 para ejecutar la siguiente secuencia de instrucciones. Incluya diagramas de un sólo ciclo de reloj para mostrar los eventos que ocurren en cada etapa de la segmentación, y coloque etiquetas sobre los buses de datos y señales de control indicando el valor que toman en ese instante.

   ```
   0x0400    staa 1000
   0x0401    ldaa #0080
   ```

7. Utilice el diagrama de la figura 7.61 para ejecutar la siguiente secuencia de instrucciones. Incluya diagramas de un sólo ciclo de reloj para mostrar los eventos que ocurren en cada etapa de la segmentación, y coloque etiquetas sobre los buses de datos y señales de control indicando el valor que toman en ese instante.

   ```
   0x0400    ldaa #0080
   0x0401    staa 1000
   ```

APÉNDICE A

USO DE VERILOG HDL EN EL ENTORNO MAX+PLUS II

A.1 INICIANDO MAX+PLUS II

MAX+PLUS II (Multiple Array Matrix Programmable Logic User System) es un ambiente gráfico de gran utilidad en el diseño, análisis y simulación de sistemas digitales. Si usted ya es un usuario experimentado en el manejo de MAX+PLUS II quizá desee pasar directamente al apartado de Verilog HDL. Si no es así, esta sección le mostrará cómo compilar y simular sus diseños mediante un ejemplo.

Una vez instalado el software de MAX+PLUS II, inicie la aplicación y espere a que aparezca la siguiente pantalla.

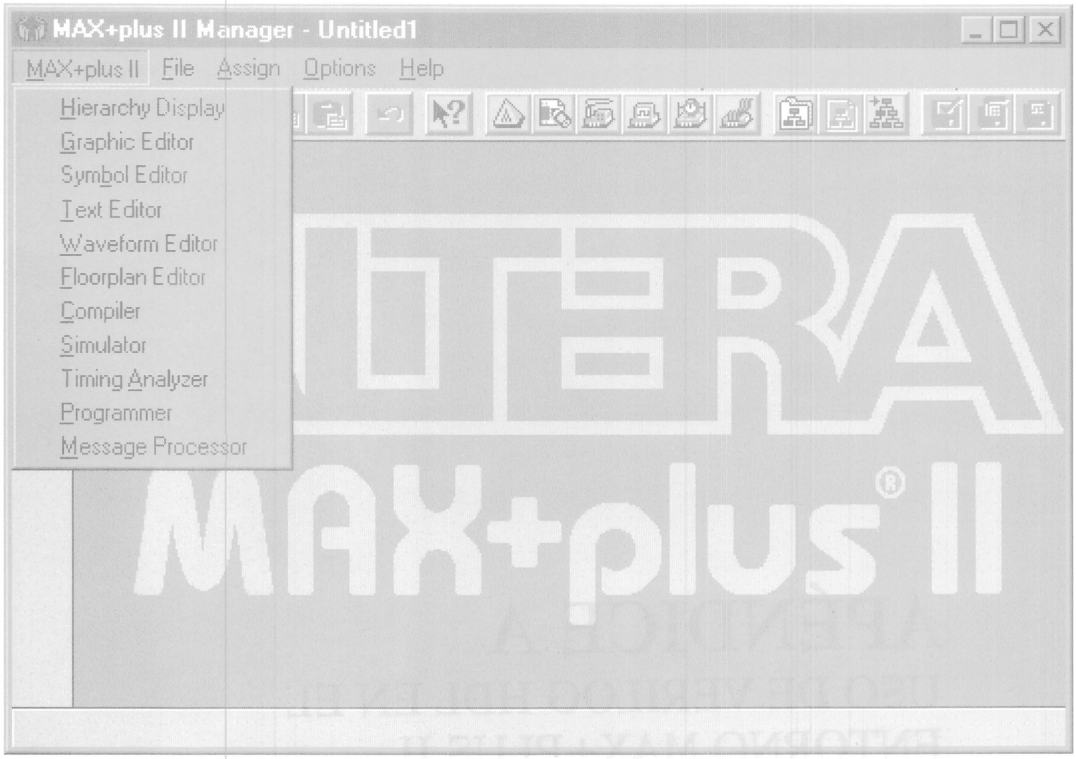


Figura A.1. El manager de MAX+PLUS II.

Éste es el Manager de MAX+PLUS II que consta de once aplicaciones para el diseño de sistemas digitales. Estas once aplicaciones se listan en el menú '*MAX+PLUS II*' de la barra de herramientas y son las siguientes.

- *Hierarchy Display*. Muestra la jerarquía actual del proyecto. Esta jerarquía se presenta como una estructura de árbol con ramificaciones que representan los distintos subdiseños.
- *Graphic Editor*. Permite crear un diseño visual en un ambiente 'lo que ves es lo que obtienes' (WYSIWYG). Para ello, existen bloques predefinidos de Altera que ejecutan determinadas funciones lógicas, o bien, es posible crear nuevos bloques con funciones definidas por el usuario.

- *Symbol Editor*. Permite editar los símbolos de los módulos existentes y también crear nuevos símbolos.
- *Text Editor*. Permite crear y editar diseños de texto basados en los lenguajes de descripción de hardware AHDL, VHDL y Verilog HDL.
- *Waveform Editor*. Permite observar el comportamiento de nuestro diseño a lo largo del tiempo. En otras palabras, presenta los resultados de la simulación.
- *Floorplan Editor*. Permite asignar manualmente los puertos de entrada y de salida del diseño lógico, a los pines de entrada y salida del dispositivo físico.
- *Compiler*. Procesa los proyectos creados y genera los archivos necesarios para la programación y simulación del circuito.
- *Simulator*. Permite revisar la operación lógica y la respuesta en el tiempo del circuito lógico.
- *Timing Analyzer*. Analiza el rendimiento del circuito lógico después de haber sido sintetizado por el compilador.
- *Programmer*. Permite programar los dispositivos físicos de Altera.
- *Message Processor*. Muestra mensajes de error, advertencias e información sobre el estado actual del proyecto.

A.2 USANDO EL EDITOR DE TEXTO DE MAX+PLUS II

Elija la opción '*Text Editor*' del menú '*MAX+PLUS II*'. Escriba el siguiente programa dentro de la ventana del Editor de Texto y guárdelo con el nombre 'Example_1.v'. Es importante que el nombre del archivo sea el mismo que el nombre del módulo, de lo contrario, el compilador marcará errores. Por otra parte, la extensión .v indica que es un diseño de Verilog.

```
module Example_1 (a, b, e, c, d);
    input   a, b, e;
    output  c, d;
    assign  c = a & b;
    assign  d = e;
endmodule
```

Su diseño deberá parecerse al de la siguiente figura.

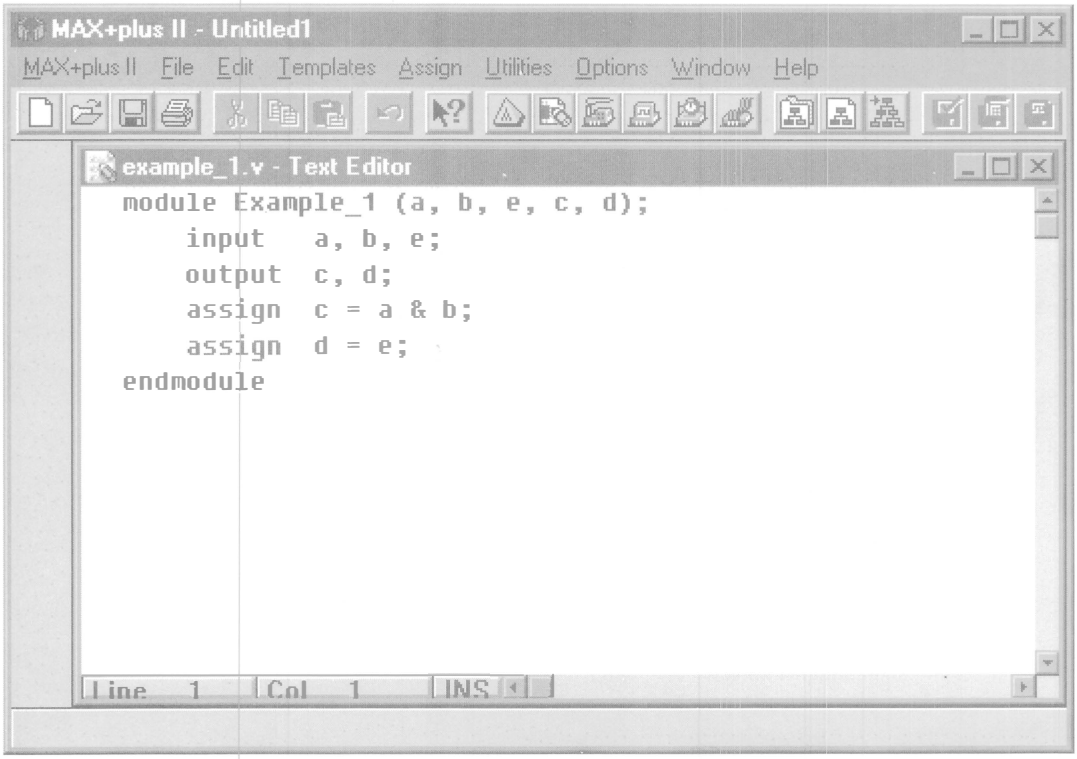


Figura A.2. Editor de texto.

Por el momento no se preocupe por la sintaxis del programa, sólo necesita saber que la señal 'c' recibe el resultado de la operación lógica 'a AND b', y que se asigna a la señal 'd' el valor de la señal 'e'. Más adelante volveremos a ver este ejemplo y se explicará a detalle su funcionamiento.

Antes de compilar el programa necesita especificar el nombre del proyecto, esto es, el nombre del archivo con mayor jerarquía. En este caso, el único archivo es 'Example_1.v' y por lo tanto, es el archivo con mayor jerarquía.

Para establecer el nombre del proyecto, diríjase al menú '*File*' de la barra de herramientas, abra el menú '*Project*' y elija '*Set Project to Current File*'. Es importante que especifique el nombre del archivo correcto porque de no hacerlo puede omitir archivos o agregar otros que no correspondan al proyecto.

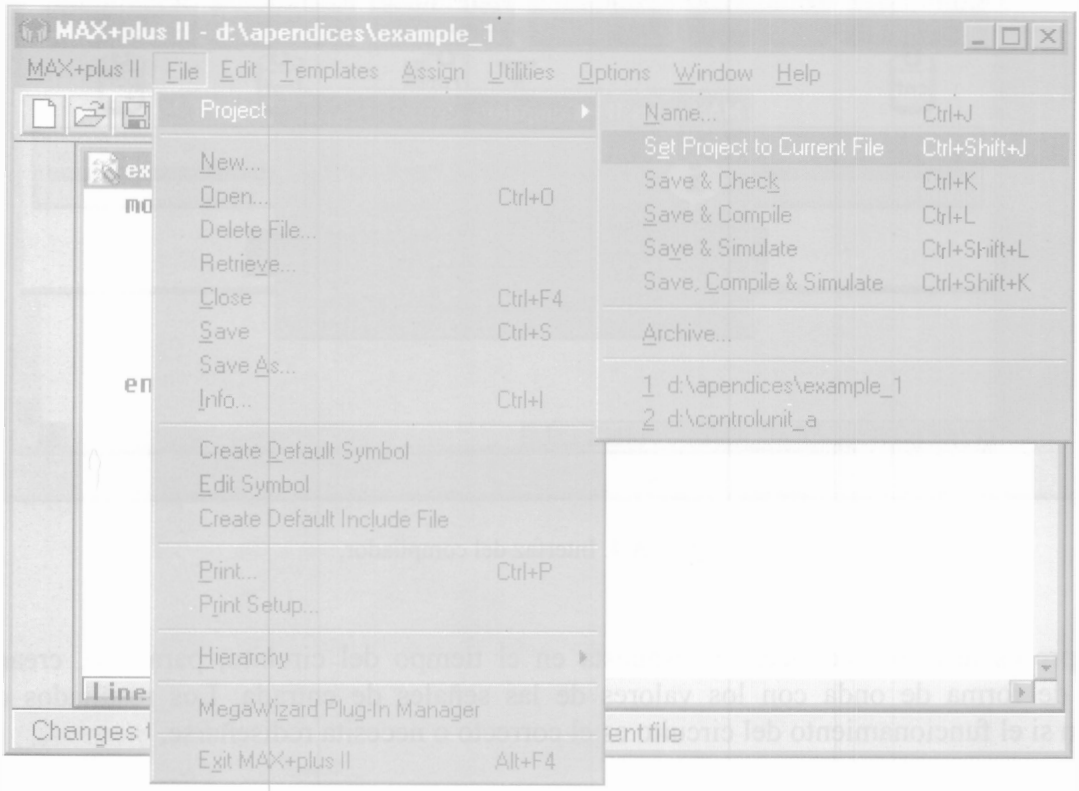


Figura A.3. Especifique el nombre del proyecto.

Ahora compile el proyecto. Vaya al menú '*File*' de la barra de herramientas, abra el menú '*Project*' y elija la opción '*Save & Compile*'. En esta ocasión aparecerá la interfaz de compilador mostrando el status de la compilación. Si no existen errores de sintaxis, el diseño será procesado satisfactoriamente.

El compilador de MAX+PLUS II procesará el proyecto y lo asignará a un dispositivo físico de la familia MAX o FLEX de Altera. Si desea cambiar este dispositivo, abra el menú '*Assign*' de la barra de herramientas y seleccione la opción '*Device*'. Una ventana aparecerá con los dispositivos lógicos disponibles, elija uno según sus necesidades.[11]

[11] Todos los ejemplos de este libro fueron compilados para el dispositivo EPF10K20RC240 de la familia FLEX.

Vuelva a compilar el proyecto para actualizar cualquier cambio realizado.

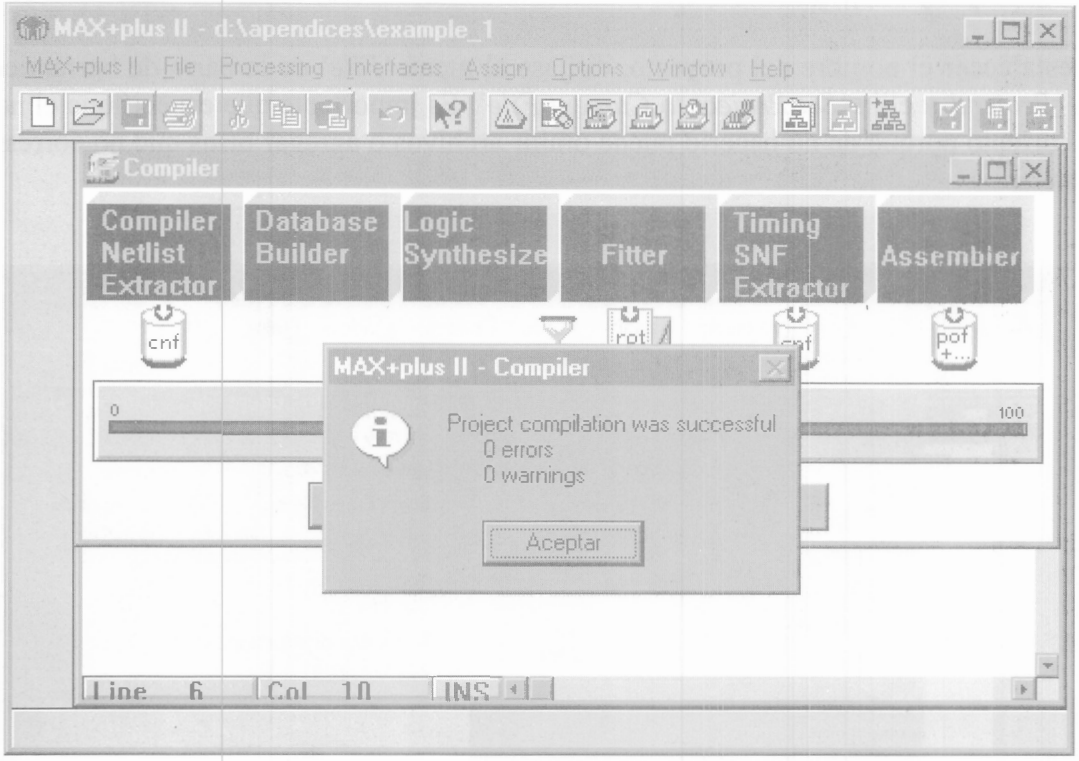


Figura A.4. Interfaz del compilador.

Siempre es deseable conocer la respuesta en el tiempo del circuito, para ello, crearemos un archivo de forma de onda con los valores de las señales de entrada. Los resultados obtenidos indicarán si el funcionamiento del circuito es el correcto o necesita rediseñarse.

Para crear un archivo de forma de onda seleccione la opción '*Waveform Editor*' del menú '*MAX+PLUS II*'. Aparecerá una ventana en donde deberá colocar los nombres de las señales de entrada y salida que desea monitorear.

Si le es incómodo estar recordando los nombres de las señales, utilice la opción '*Enter Nodes from SNF*' ubicada en el menú '*Node*' de la barra de herramientas. Esta opción abrirá un cuadro de diálogo con la lista de todas las señales utilizadas en el proyecto.

Dentro del cuadro de diálogo podrá seleccionar qué señales listar: entradas, salidas, nodos de grupo (señales compuestas por más de una línea) o combinaciones de las anteriores. Para actualizar la lista oprima el botón '*List*' ubicado en el extremo superior derecho del cuadro de diálogo. Las señales disponibles aparecerán dentro del recuadro '*Available Nodes & Groups*', marque las señales que desee monitorear y transfiéralas al recuadro '*Selected Nodes & Groups*'.

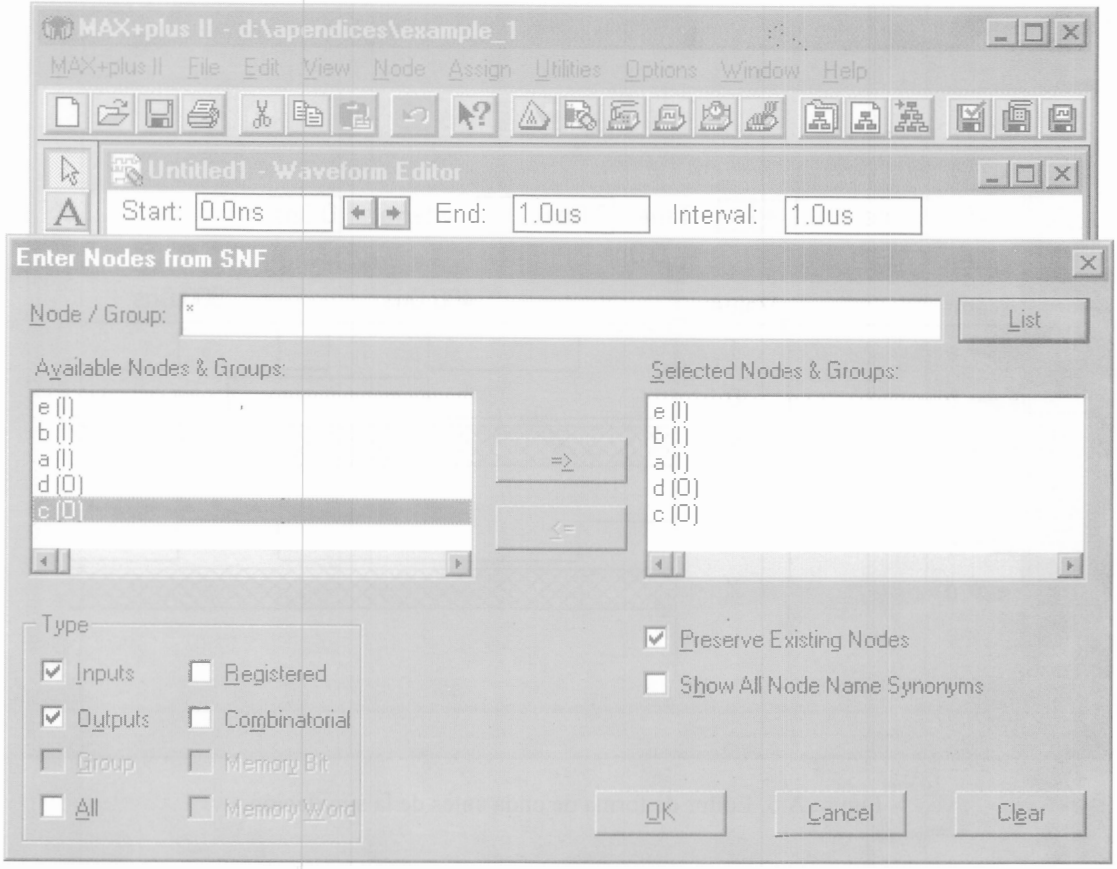


Figura A.5. Selección de las señales de entrada y salida en el editor de forma de onda.

Dé clic en el botón '*OK*'. Las señales seleccionadas aparecerán en el editor de forma de onda.

Ahora dé valores a las señales de entrada. Para ello, seleccione un rango de tiempo arrastrando el puntero de mouse y asigne un valor a ese rango. Los valores son asignados mediante los botones de la barra de herramientas de la izquierda, de manera que podemos tener valores de cero (nivel lógico bajo), de uno (nivel lógico alto), no importa (nivel lógico indefinido) y alta impedancia (Z).

Antes de la simulación las señales de salida presentan niveles lógicos indefinidos. Una vez simulado el circuito, estas señales cambian dependiendo de los valores de las entradas y la lógica interna del circuito.

Termine de asignar valores arbitrarios a las señales de entrada, guarde el archivo con el nombre '*Example_1.scf*' y seleccione la opción '*Save & Simulate*' ubicada en el menú '*Project*' del menú '*File*'. Esta opción abrirá la ventana del Simulador, quien calculará los valores de las salidas en base a los datos de entrada.

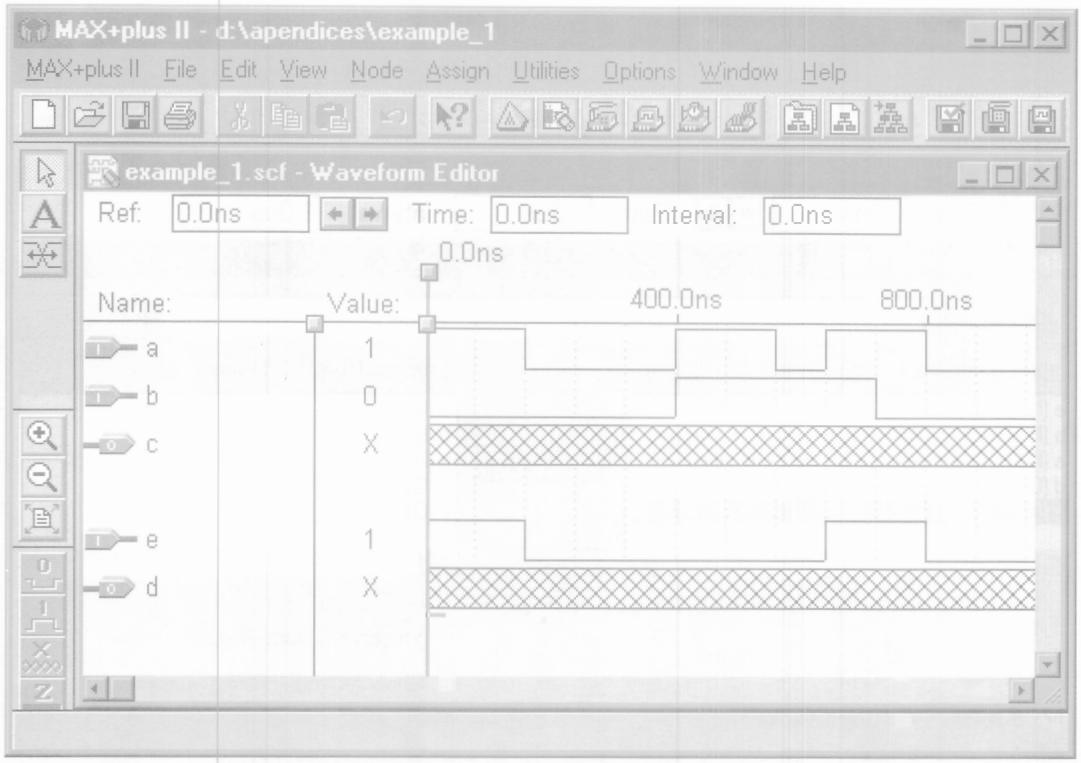


Figura A.6. Editor de forma de onda antes de la simulación.

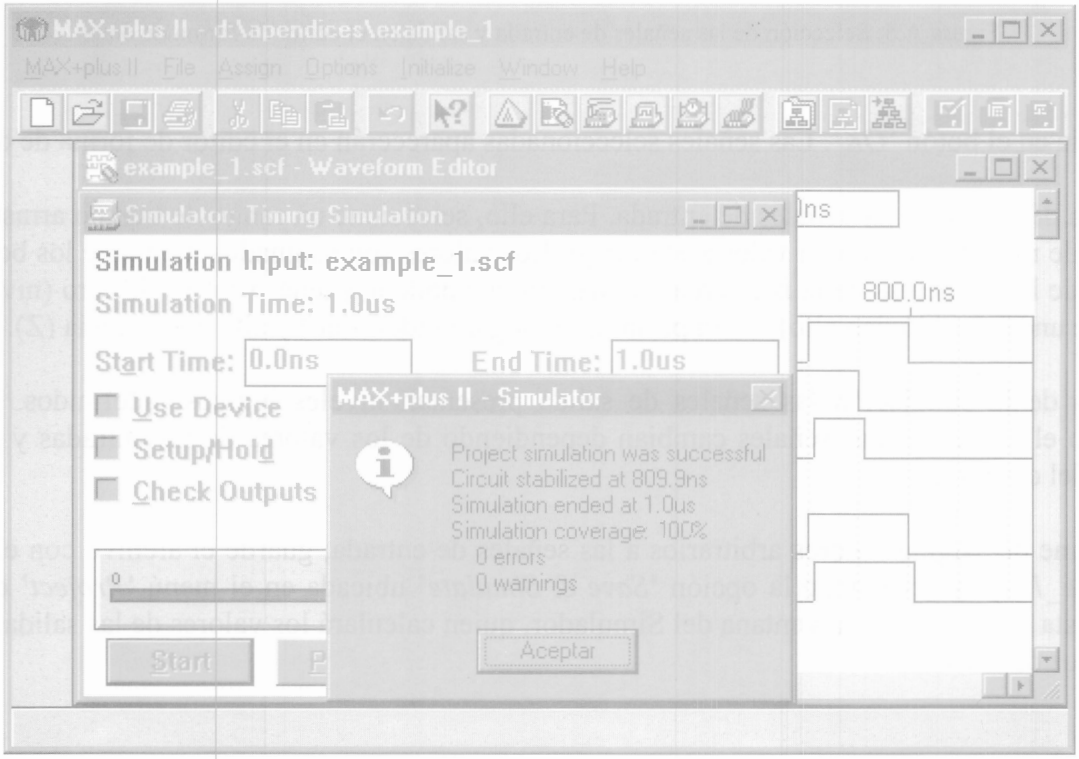


Figura A.7. Fin de la simulación.

La siguiente figura presenta los resultados de la simulación. La señal 'c' muestra el resultado de la operación AND entre los valores de las entradas 'a' y 'b', dicha operación vale '1' si ambas señales de entrada valen '1' y vale '0' para cualquier otra combinación de valores de las entradas. Por otra parte, la señal 'd' recibe exactamente el valor de la señal de entrada 'e'.

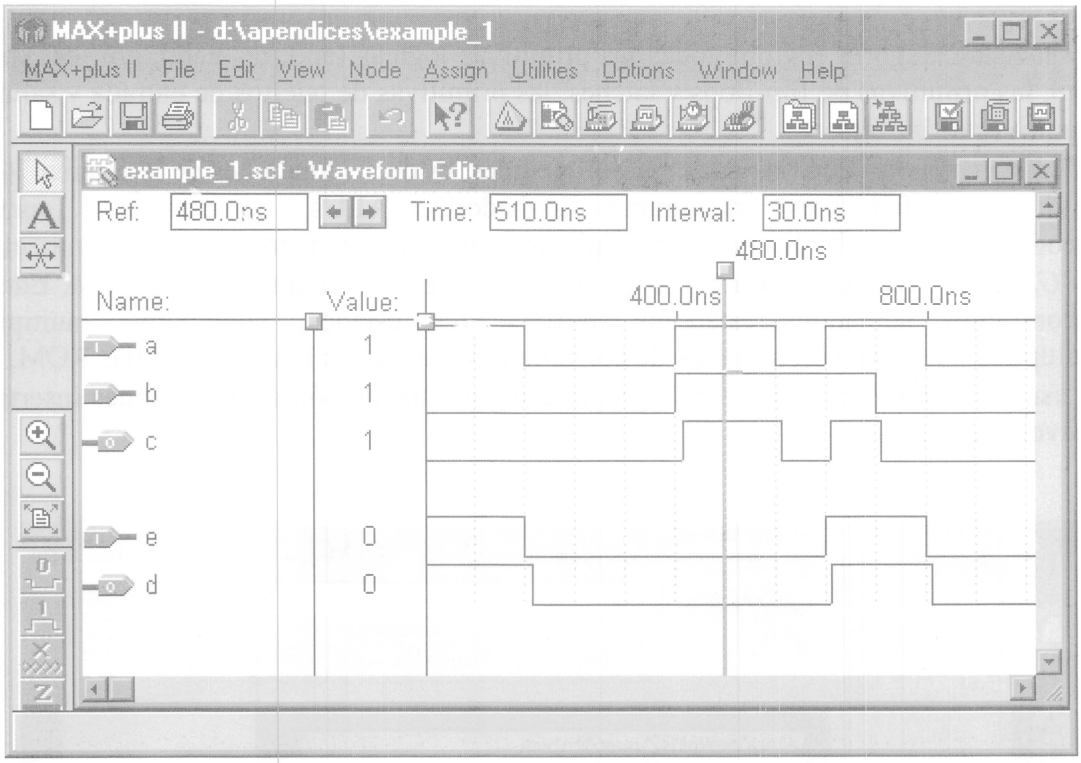


Figura A.8. Editor de forma de onda después de la simulación.

Si observa cuidadosamente el archivo de simulación notará que el resultado presenta un pequeño retardo. Esto es totalmente normal y no afecta en nada al desempeño del circuito, ya que el retardo es insignificante (10.4 ns). Sin embargo, estos retardos sí son de gran importancia en circuitos con tiempos de respuesta muy pequeños, pues retrasan las operaciones de todo el circuito.

En ocasiones requerirá que el tiempo total de simulación sea mayor a 1µs. Si es así, diríjase al menú '*File*' de la barra de herramientas y seleccione la opción '*End Time*'. Ahora cambie el tiempo final de simulación en el cuadro de diálogo que aparece.

Si requiere cambiar el tamaño de la cuadrícula, diríjase al menú '*Options*' de la barra de herramientas y elija '*Grid Size*'. Un nuevo cuadro de diálogo aparecerá con el tamaño actual de la cuadrícula, modifíquelo de acuerdo a sus necesidades.

A.3 USANDO EL EDITOR GRÁFICO DE MAX+PLUS II

Seleccione la opción '*Graphic Editor*' del menú '*MAX+PLUS II*'. A continuación elija del menú '*Symbol*' la opción '*Enter Symbol*'. Entonces aparecerá una ventana con los nombres de los símbolos disponibles clasificados en varias categorías.

- *Prim*. Esta categoría engloba todas las primitivas lógicas (and, or, nand, nor, or, xnor, xor, etc.), las primitivas de flip-flops y latches, las primitivas de puertos de entrada y salida, y las primitivas de buffers.
- *MF*. Esta categoría engloba a las macrofunciones, es decir, funciones de la familia 74LSxxx. Entre estas funciones encontramos sumadores, ALUs, comparadores, contadores, decodificadores, multiplicadores, multiplexores, registros, registros de corrimiento, etc.
- *MEGA_LPM*. Esta categoría encierra todas las funciones parametrizadas LPM. Entre estas encontramos sumadores/restadores, buffers tres estados, contadores, multiplexores, multiplicadores, latches, registros de corrimientos, memorias RAM, memorias ROM, etc.
- El usuario también puede crear su propia categoría de símbolos para después insertarlos en nuevos proyectos.

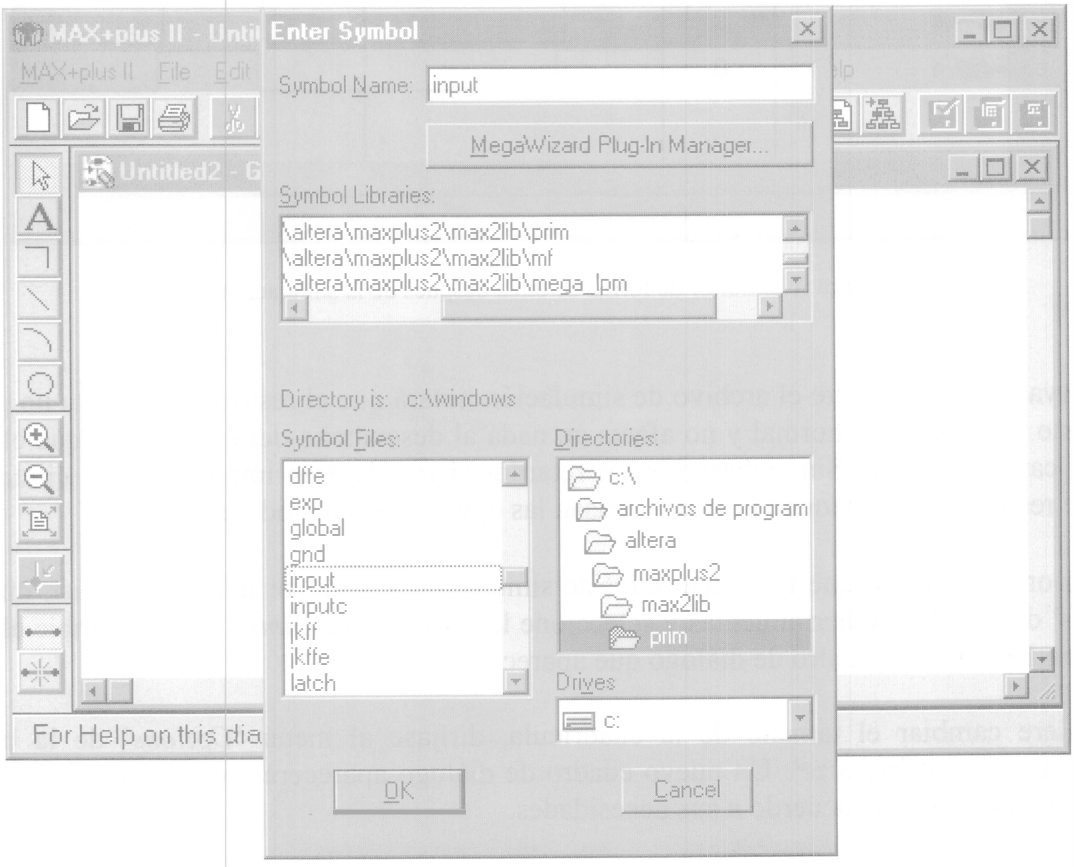


Figura A.9. Inserción de símbolos en el editor gráfico.

Como ejemplo ilustrativo desarrollaremos el mismo programa presentado en el editor de texto.

En primer lugar abra la ventana de '*Enter Symbol*' y seleccione de la categoría de primitivas una compuerta AND de dos entradas (and2). Vuelva a abrir la ventana y elija tres primitivas de puertos de entrada (input) y dos de puertos de salida (output).

A continuación conecte todos los elementos como se muestra en la figura A.10. Conectar dichos elementos es muy sencillo: coloque el puntero del mouse sobre alguno de los pines, si se trata de un pin válido el puntero del mouse cambiará a una cruz. Enseguida presione el botón izquierdo del mouse y sin soltarlo arrastre el puntero hasta el pin que desee conectar. Suelte el botón y la conexión entre los pines quedará realizada.

También renombre los pines de entrada y de salida. Para ello, seleccione el campo etiquetado con el nombre '*PIN_NAME*', dé doble clic sobre este campo y cambie cada etiqueta por el nombre sugerido.

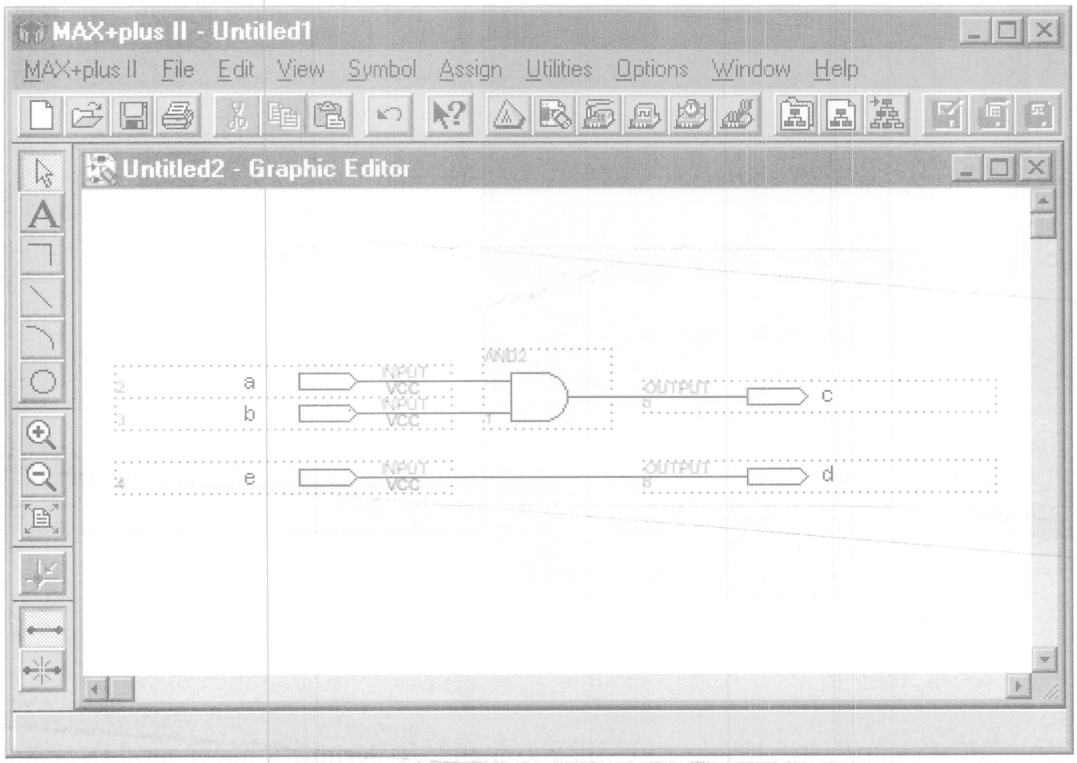


Figura A.10. Circuito equivalente utilizando el editor gráfico.

Guarde el archivo con el nombre 'Example_1b.gdf'' y compílelo. Recuerde que antes de compilar un nuevo proyecto debe especificar su nombre mediante la opción '*Set Project to Current File*' ubicada en el menú '*Project*' del menú '*File*'. Enseguida cree el archivo de forma de onda mostrado en la figura A.6 y simúlelo.

Ahora compare la salida de este circuito con la salida del ejemplo hecho en Verilog; notará que ambas simulaciones son idénticas, por lo tanto, los archivos son equivalentes.

Por último, construiremos el símbolo de nuestro módulo con el fin de poderlo agregar en otros proyectos gráficos. Así que abra el menú '*File*' de la barra de herramientas y seleccione la opción '*Create Default Symbol*'. Espere unos segundos y tendrá un archivo con extensión .sym.

Siga el mismo procedimiento para crear el símbolo del archivo '*Example_1.v*'. En este caso, aparecerá la interfaz del compilador para mostrar el avance en la creación del símbolo y un cuadro de diálogo para indicar la terminación.

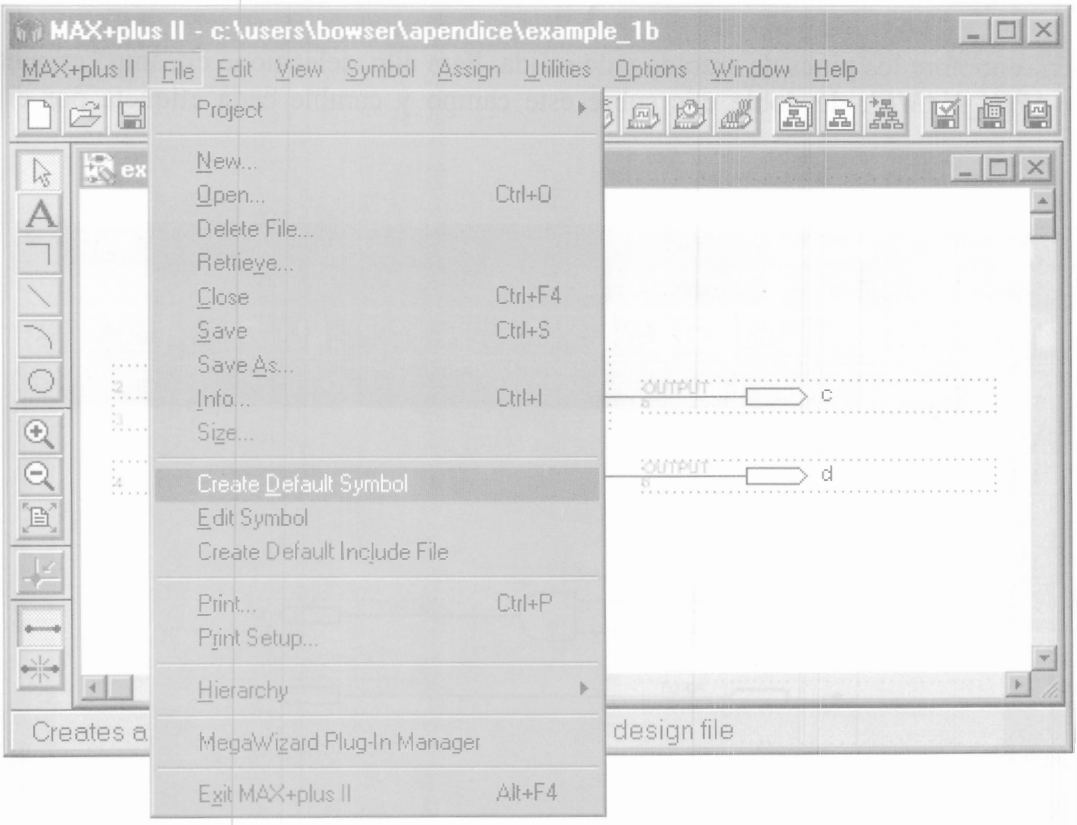


Figura A.11. Creación del símbolo.

La opción '*Edit Symbol*', localizada en el mismo menú, le permitirá hacer modificaciones sobre el símbolo, incluyendo cambios en la disposición de los pines de entrada y salida, cambios en el tipo de fuente de las letras o en el tamaño de las mismas.

También se tiene la opción de crear el archivo Include tanto de un archivo de texto como de uno gráfico. Estos archivos Include tienen la misma función que los archivos de símbolos, sólo que los archivos Include son para insertarse en diseños de texto y no en diseños gráficos.

El archivo Include es creado utilizando la opción "*Create Default Include File*' del menú '*File*'. Si este archivo es creado desde el editor de texto, la interfaz del compilador aparecerá mostrando el estado de la creación del archivo. Si es creado con el editor gráfico, sólo es necesario esperar unos segundos y el archivo con extensión .inc será creado.

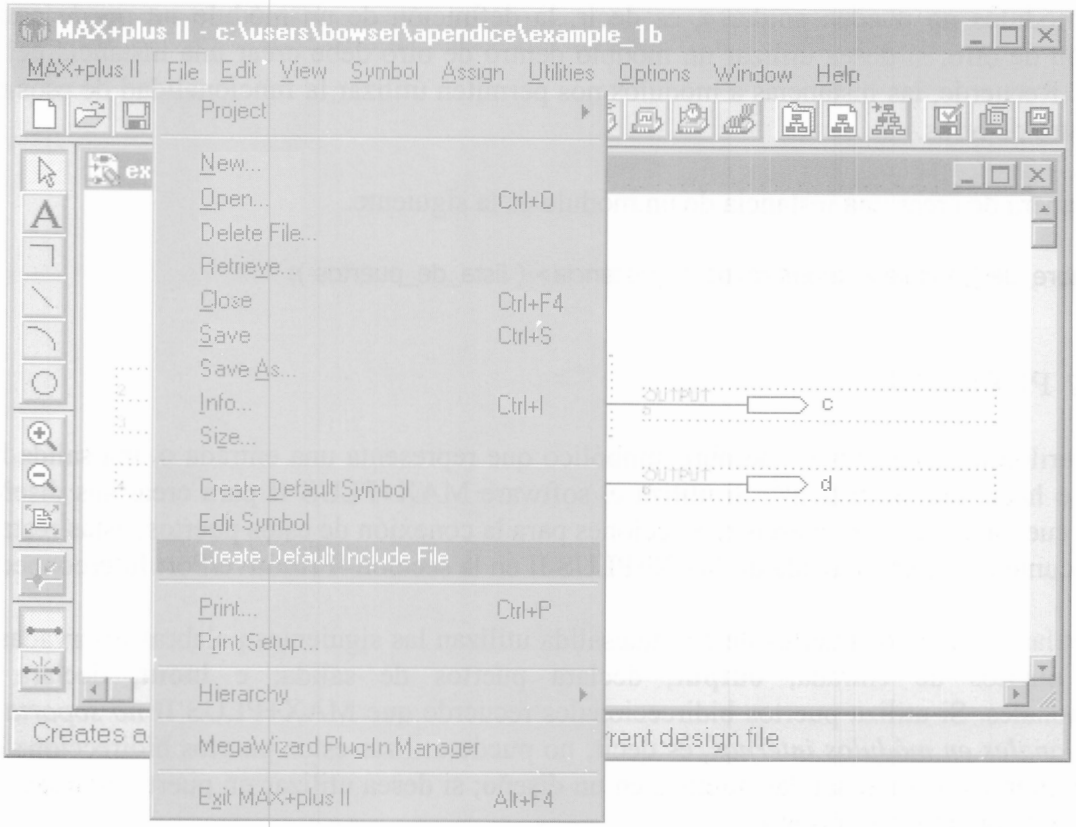


Figura A.12. Creación del archivo Include.

A.4 EL LENGUAJE DE DESCRIPCIÓN DE HARDWARE VERILOG

Verilog fue desarrollado en 1984 por Gateway Design Automation. Su aparición fue bien recibida por todos los diseñadores de circuitos integrados y de sistemas digitales, de esta manera, en poco tiempo se convirtió en uno de los lenguajes más populares. Quizá su gran aceptación se debió a que es fácil de aprender, pues los programadores encuentran mucha similitud con otros lenguajes como C ó Pascal.

A.4.1 MÓDULOS

Los módulos son la unidad básica de un diseño en Verilog. Un diseño en Verilog consiste de uno o varios módulos organizados jerárquicamente que contienen información sobre el diseño, o bien, contienen instancias de otros módulos.

La sintaxis para la declaración de un módulo es la siguiente.

```
module <nombre_del_módulo> ( lista_de_puertos ) ;
    // Declaración de la lista de puertos
    // Código del módulo
endmodule
```

Los módulos no pueden anidarse, es decir, la definición de un módulo no puede contener la definición de otro; si desea utilizar un módulo dentro de otro debe crear una instancia del módulo deseado. Recuerde, las instancias a módulos nos permiten utilizar la funcionalidad de esos módulos pero dentro de otros.

La manera de crear una instancia de un módulo es la siguiente.

<nombre_del_módulo> <nombre_de_la_instancia> (lista_de_puertos);

A.4.2 PUERTOS

En Verilog un puerto es un nombre simbólico que representa una entrada o una salida hacia un módulo o hacia una primitiva[12]. Si utiliza el software MAX+PLUS II para crear sus diseños debe tener en cuenta que existen ciertas restricciones para la conexión de estos puertos; estas restricciones están documentadas en la ayuda de MAX+PLUS II en la sección 'Primitive/Port Interconnections'.

La declaración de los puertos de entrada/salida utilizan las siguientes palabras reservadas: **input**, declara puertos de entrada; **output**, declara puertos de salida; e **inout**, declara puertos bidireccionales. Si utiliza puertos bidireccionales recuerde que MAX+PLUS II *no soporta puertos bidireccionales en módulos internos*, es decir, no puede utilizar estos puertos bidireccionales como entradas internas ó como salidas internas en un diseño; si desea utilizar un puerto bidireccional éste debe ser externo a todo el diseño.

A.4.3 COMENTARIOS

Los comentarios son utilizados para introducir descripciones adicionales, explicaciones y cualquier otra información que no es parte del código pero que es muy útil para entenderlo. Los comentarios son ignorados durante la compilación del diseño.

Hay dos tipos de comentarios: los de una sola línea y los de múltiples líneas. Los comentarios de una sola línea comienzan con los caracteres // y terminan con el carácter de fin de línea. En cambio, los comentarios de múltiples líneas comienzan con la secuencia /* y terminan con */; todo lo que esté entre estos delimitadores será un comentario.

A.4.4 BLOQUES PROCEDURALES

Los bloques procedurales (initial y always) son las construcciones básicas para el modelado de comportamientos en Verilog. Los bloques procedurales representan flujos de tareas, y pueden ser secuenciales o concurrentes. Los bloques procedurales no pueden anidarse.

[12] Una primitiva es un bloque funcional básico que se utiliza en MAX+PLUS II para el diseño de circuitos. Todas las primitivas soportadas por MAX+PLUS II se encuentran en el directorio \maxplus2\max2lib\prim que se creó tras la instalación del software e incluyen: buffers, flip-flops, latches, compuertas lógicas, entre otros.

Bloque initial

Su especificación consiste de la palabra reservada **initial** seguida de una sentencia o un bloque de sentencias que será ejecutado cuando el bloque se active. Recuerde que las palabras reservadas **begin** y **end** sirven para agrupar bloques de sentencias.

```
initial begin
    // Sentencia o bloque de sentencias
end;
```

El bloque *initial* es activado al inicio de la simulación y sólo se ejecuta una vez; regularmente se utiliza para inicializar algunas variables de nuestro diseño. Si el usuario especifica más de un bloque *initial*, éstos serán ejecutados al comienzo de la simulación de manera concurrente.

Bloque always

Su especificación consiste de la palabra reservada **always** seguida de una sentencia o de un bloque de sentencias que será ejecutado cuando el bloque se active. Un bloque *always* es activado al inicio de la simulación y continuará ejecutándose durante toda la simulación. Normalmente los bloques *always* son manejados por un evento de control que especifica cuándo debe ejecutarse el bloque. Estos eventos pueden ser los flancos del reloj, cambios en los valores de las variables, entre otros. Más adelante se explica cómo construir lógica combinacional y secuencial mediante el uso de los bloques *always*.

A.4.5 EVENTOS

Los eventos permiten controlar la ejecución de una sentencia o bloque de sentencias. En su forma más simple, un evento es un cambio de valor en una net o en un registro. Su especificación es utilizando el símbolo @ seguido por el nombre de la net o del registro.

```
@ (nombre_de_la_net_o_del_registro)
```

Las sentencias escritas después del evento de control serán ejecutadas sólo cuando el evento o los eventos de control sean detectados. Los tipos más comunes de eventos son el flanco de subida y el flanco de bajada de una señal. Para detectar el flanco de subida de una señal utilizamos la palabra reservada **posedge** y para registrar el flanco de bajada utilizamos la palabra **negedge**.

A.4.6 SENTENCIAS DE CONTROL DE PROGRAMA

En cierto sentido, las sentencias de control de programa son la esencia de cualquier lenguaje porque gobiernan el flujo de ejecución de un programa. Las sentencias de control de programa de Verilog podemos englobarlas en dos categorías. La primera está formada por las instrucciones condicionales *if* y case; y la segunda por las sentencias de control de bucles *while*, *for*, entre otras.

Sentencia If

La sintaxis general de la sentencia if es,

```
if ( expresión_condicional )[13]
    // Bloque de sentencias 1
else
    // Bloque de sentencias 2
```

La cláusula *else* es opcional. Si *expresión_condicional* evalúa a verdadero (valores diferentes de cero), entonces el bloque de sentencias 1 es ejecutado; en caso contrario, el bloque de sentencias 2 será ejecutado. Recuerde que sólo se ejecuta el código asociado al *if* o el código asociado al *else*, pero nunca los dos. Las sentencias *if* sí pueden anidarse.

Sentencia Case

La sentencia *case* es muy parecida a la sentencia *if*. La forma general de la sentencia *case* es,

```
case ( expresión )
    case_item_1:   // Bloque de sentencias 1
    case_item_2:   // Bloque de sentencias 2
    ...            ...
    case_item_n:   // Bloque de sentencias n
    default:       // Bloque de sentencias default
endcase
```

La sentencia *case* va comparando sucesivamente una *expresión* con una lista de constantes enteras. Cuando se encuentra una coincidencia, se ejecuta el bloque de sentencias asociado a esa constante. Las constantes no necesitan estar en ningún orden especial. Si no se encuentra alguna coincidencia, entonces se ejecuta el bloque de sentencias asociado a la etiqueta *default*. La etiqueta *default* es opcional y, si no está presente, no se hará nada en caso de que no se produzca alguna coincidencia.

Bucle While

La forma general del bucle *while* es,

```
while ( expresión_condicional )
    // Bloque de sentencias
```

El bucle *while* se ejecutará mientras *expresión_condicional* sea verdadera. Esta condición es evaluada cada vez que el bucle inicia, si la condición continúa siendo verdadera entonces se ejecutará el bloque de sentencias asociado, si no, el control del programa pasará a la siguiente línea fuera del bucle.

[13] Una expresión combina operandos y operadores para producir nuevos valores, los cuales pueden ser asignados a nets o a registros, o bien, pueden determinar una condición.

Bucle For

La forma general del bucle *for* es,

```
for ( asignación_inicial; expresión; actualiza_contador );
begin
    // Bloque de sentencias
end
```

La ejecución de un bucle *for* se ejecuta en tres pasos: 1) se inicializa el contador del bucle como lo especifica *asignación_inicial*; 2) la condición de salida dada por *expresión* es evaluada; si es cero, desconocido o alta impedancia, el bucle finaliza; de lo contrario, el bucle continúa ejecutando su bloque de sentencias asociado; 3) el contador del bucle es modificado de acuerdo a lo indicado por la expresión *actualiza_contador*; los pasos 2 y 3 son repetidos hasta que *expresión* sea falsa.

Bucle Repeat

La forma general del bucle *repeat* es,

```
repeat ( expresión )
    // Bloque de sentencias
```

El bucle *repeat* se utiliza para ejecutar un bloque de sentencias determinado número de veces. Este número es especificado entre paréntesis después de la palabra reservada **repeat** y puede ser una constante o una variable, pero debe ser un número entero. Este tipo de bucle es muy útil en aquellos casos en los que se conoce por adelantado el número de veces que se desea ejecutar, por ejemplo, al inicializar vectores y memorias.

Bucle Forever

La forma general del bucle *forever* es,

```
forever
begin
    // Bloque de sentencias
end
```

Este es el bucle más simple que maneja Verilog. Su especificación comienza con la palabra reservada **forever** seguida de la sentencias o bloque de sentencias que se desean repetir. Recuerde que si el bloque de sentencias tiene más de una sentencia, entonces deberán escribirse entre las palabras **begin** y **end** para delimitar el bloque. El bucle *forever* repetirá el bloque de sentencias continuamente hasta finalizar la simulación; este bucle tiene la misma funcionalidad que el bloque procedural *always*.

A.4.7 TIPOS DE DATOS

Nets

Una *net* representa un cable transportando un señal que viaja entre diferentes componentes lógicos de un diseño. Las *nets*, también llamadas redes de conexión, actualizan continuamente sus salidas con respecto a los cambios registrados en sus entradas. Por ejemplo en la figura A.13, la net *out_b* está conectada a *in_a* mediante una compuerta not (inversor); cualquier cambio en el valor de *in_a* provocará la actualización de *out_b*. Note que los nuevos valores de *out_b* no necesitan ser asignados explícitamente.


```
reg in_a;
wire out_b;
not comp(out_b, in_a);
```

Figura A.13. Ejemplo de una net.

En Verilog, las nets son declaradas mediante las siguientes palabras reservadas: **wire**, **supply0**, **supply1**, entre otras. Su tamaño por default es de 1 bit.

Regs (registros)

La diferencia fundamental entre las *nets* y los *regs* es que los registros tienen que ser asignados con valores de manera explícita. El valor almacenado en un *reg* se mantendrá hasta que un nuevo valor sea asignado. Los registros se declaran con la palabra reservada **reg**, y su tamaño es de un bit.

Integers

Los *integers* permiten almacenar números enteros negativos y positivos. Su declaración es mediante la palabra reservada **integer** y su tamaño depende de la máquina host, pero al menos son de 32 bits. Observe que los enteros son muy parecidos a los registros, la diferencia entre ellos radica en que los registros son tratados como valores sin signo, en cambio, los integers son tratados como valores con signo. Por lo tanto, tenga mucho cuidado al hacer operaciones aritméticas entre ellos, pues aunque los registros pueden almacenar números negativos en complemento a dos, durante la aritmética serán tratados como números sin signo.

Reales

Los números reales son aquellos que tienen parte entera y parte decimal. Cuando declare este tipo de números tenga en cuenta que debe escribir al menos un dígito después del punto decimal, de lo contrario, estará incorrecta la sintaxis. Los números reales se declaran con la palabra reservada **real**;

su tamaño depende de la máquina host, pero al menos son de 64 bits. Recuerde que un número real puede ser convertido en entero, para esto, el número real es redondeado al entero más cercano. A continuación se presentan algunos ejemplos de números reales válidos.

```
real num1, num2, num3;          // declaración de tres variables del tipo real
num1 = 3.141596259;
num2 = 1.3;                     // un número real en formato decimal
num3 = 1.3e27;                  // un número real en notación científica
```

Time

Las variables del tipo *time* son registros de 64 bits capaces de almacenar valores sin signo. Se declaran utilizando la palabra reservada **time**.

Vectores

Tanto las nets como los regs son variables de un bit, por lo tanto, si se desea usar una variable de mayor tamaño se debe declarar como un vector. Los vectores pueden accederse totalmente o parcialmente, es decir, podemos asignar valores a todo el vector o a ciertos elementos de él. La declaración de un vector es la siguiente:

```
<Tipo_de_dato> [Tamaño_del_vector] <Nombre_de_la_variable>
```

donde: Tamaño_del_vector es un rango de valores que representa el número de elementos que integran al vector. Normalmente, el rango del vector se declara colocando el valor del índice del bit más significativo a la izquierda y el índice del bit menos significativo a la derecha, esto es,

```
[Tamaño_del_vector] = [Bit_más_significativo : Bit_menos_significativo]
```

Y la forma de referenciar a cada bit del vector es,

```
<Nombre_de_la_variable> [Referencia_a_los_bits]
```

Observe el siguiente ejemplo.

```
reg  [3:0]  salida;          // salida es un registro de 4 bits, el índice del bit más significativo
                             // es 3 y el índice del bit menos significativo es 0
wire  [31:0]  dato;          // dato es una net de 32 bits

salida = 4'b0101;            // asignación total del vector salida
dato[3:0] = salida;          // asignación parcial del vector dato
```

Arreglos

La declaración de un arreglo (array) de elementos es la siguiente.

<Tipo_de_dato> [Tamaño_del_vector] <Nombre_de_la_variable> [Tamaño_del_arreglo]

La referencia a los elementos del arreglo se hace de la siguiente manera.

<Nombre_de_la_variable> [Referencia_al_arreglo] [Referencia_a_los_bits]

Observe el ejemplo.

```
integer dato [7:0];                 // dato es un arreglo de 8 elementos del tipo integer
reg [15:0] memoria [1023:0];        // memoria es un arreglo de 1024 elementos del tipo reg
                                    // de 16 bits cada uno
memoria[489] = 16'hABCD;            // asignación total del elemento 489 de la variable memoria
memoria[489][7:0];                  // referencia parcial a los 8 bits menos significativos del
                                    // elemento 489 de la variable memoria
```

A.4.8 FORMATO DE NÚMEROS ENTEROS

En Verilog, los números enteros pueden representarse en binario, decimal, hexadecimal u octal. El formato general para su representación es el siguiente.

<tamaño>'<base><número>

donde: *Tamaño* especifica el número de bits utilizados para representar al número. Por ejemplo, para representar un dígito binario se requiere un bit de tamaño, por el contrario, para representar un dígito hexadecimal se requieren cuatro bits. Si no se especifica un valor para el parámetro tamaño entonces se asignará un tamaño por default el cual es dependiente de la máquina, pero al menos de 32 bits.

Base indica la base numérica en la que está representado el número. Si el número se presenta en binario, base vale **b** ó **B**; si está en decimal, **d** ó **D**; si está en hexadecimal, **h** ó **H**; y si está en octal, **o** u **O**. Si no se especifica un valor para base el default es decimal.

Número es el número escrito en la base indicada. Para la base binaria los dígitos permitidos son {0,1}, para la base octal {0,1,2,3,4,5,6,7}, para la base decimal {0,1,2,3,4,5,6,7,8,9} y para la base hexadecimal {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F}.

Los valores X y Z

Además existen dos valores especiales, **x** y **z**, que también se utilizan para representar números. El valor *x* representa un valor desconocido y *z* un valor de alta impedancia. Una *x* declara cuatro bits desconocidos en hexadecimal, tres en octal y uno en binario. Una *z* declara valores de alta impedancia de manera similar a *x*. Observe los siguientes ejemplos.

```
8'b10100010     // número de 8 bits en binario
8'hA2           // número de 8 bits en hexadecimal
```

```
4'b10x0        // número de 4 bits en binario con el 2do. bit menos significativo desconocido
8'h4z          // número de 8 bits en hexadecimal con los cuatro bits menos significativos
               // en alta impedancia
```

El valor de *z* es muy útil para implantar circuitos tres estados. Recuerde que un circuito tres estados es aquel cuya salida puede presentar un nivel lógico alto, un nivel lógico bajo o una alta impedancia (en este último caso, la salida se aísla del resto del circuito). Estos circuitos son empleados en arquitecturas donde las salidas de diversos componentes están conectadas a un mismo bus, de esta manera, el circuito tres estados se asegura de conectar la salida adecuada al bus y de aislar al resto de las salidas para evitar múltiples escrituras en el bus.

Números negativos

Verilog también permite declarar números negativos, para ello, debemos anteponer al campo *tamaño* un signo de menos. Es muy importante que el signo de menos aparezca al inicio de todo el número, ya que si aparece entre el tamaño y la base ó entre base y el número, entonces el número será inválido.

```
-8'd5          // complemento a dos de 5, guardado en 8 bits
8'd-5          // sintaxis inválida
```

A.4.9 TIPOS DE OPERADORES

Operadores aritméticos

Símbolos: *, /, +, -, %
Los operadores aritméticos son multiplicación, división, suma, sustracción y módulo.

Sean a, b y c tres vectores del tipo reg de 4 bits cada uno, donde a = 4'b1100 y b = 4'b0011 entonces,

```
Multiplicación:   c = a * b   = 4'b1000 (los cuatro bits menos significativos de 36)
División:         c = a / b   = 4'b0100
Adición:          c = a + b   = 4'b1111
Sustracción:      c = a – b   = 4'b1001
Módulo:           c = a % b   = 4'b0000
```

Operadores lógicos

Símbolos: !, &&, | |
Los operadores lógicos son AND lógico, OR lógico y NOT lógico. Todos los operadores lógicos evalúan a verdadero (1), falso (0) o desconocido (x). Un operando es verdadero si es diferente de cero, y falso si es cero. Los valores de alta impedancia y desconocido evalúan a falso. Un operando puede ser una variable o una expresión la cual será evaluada a verdadero o falso.

Sean a = 4'b0010, b = 4'b0000 y c = 4'bxxxx entonces,

```
AND lógico:     a && b   evalúa a falso
OR lógico:      a | | b  evalúa a verdadero
OR lógico:      a | | c  evalúa a verdadero
NOT lógico:     ! c      evalúa a desconocido
```

Operadores de relación

Símbolos: >, <, >=, <=

Los operadores de relación son mayor que, menor que, mayor o igual a, y menor o igual a. Los valores verdadero y falso se definen de la misma manera que en los operadores lógicos. En este caso si algún operando es desconocido entonces toda la expresión se evalúa a desconocido.

Sean a = 4'h2, b = 4'h5 y c = 4'hx entonces,

```
Menor que:      a < b    evalúa a verdadero
Mayor que:      a > b    evalúa a falso
Menor o igual a: c <= a  evalúa a desconocido
```

Operadores de igualdad

Símbolos: ==, !=, ===, !==

Los operadores de igualdad son igualdad lógica (==) y desigualdad lógica (!=). Estos operadores comparan sus operandos bit a bit. Si los bits más significativos de alguno de los operandos presentan valores de desconocido o de alta impedancia (x ó z), entonces el resultado de la evaluación será desconocido (x). Para comparar igualdad y desigualdad entre valores de *x* y *z* deberá utilizar los operadores === para igualdad y !== para desigualdad.

Sean a = 4'h4, b = 4'h7 y c = 4'bxx10 entonces,

```
Igualdad:       a == b   evalúa a falso
Desigualdad:    a != b   evalúa a desconocido
```

Operadores bitwise

Símbolos: ~, &, |, ^, (~^, ^~)

Los operadores bitwise son negación, AND, OR , XOR y XNOR. Estos operadores ejecutan operaciones bit a bit sobre sus operandos, si alguno de los operandos es más pequeño en tamaño respecto al otro entonces es extendido hacia la izquierda con los ceros necesarios.

Sean a = 4'b1100, b = 4'b0011 y c = 4'b0101 entonces,

```
Negación:       ~a       evalúa a 4'b0011
AND:            a & c    evalúa a 4'b0100
OR:             a | b    evalúa a 4'b1111
XOR:            b ^ c    evalúa a 4'b0110
XNOR:           a ~^ c   evalúa a 4'b0110
```

Operadores de Reducción

Símbolos: &, ~&, |, ~|, ^, (~^, ^~)

Los operadores de reducción son AND, NAND, OR, NOR, XOR y XNOR. Estos operadores toman un sólo operando y ejecutan la operación descrita entre los bits del operando, el resultado queda almacenado en un bit.

Sean a = 4'b1111 y b = 4'b0101 entonces,

AND:	& a	evalúa a 1 (se ejecuta la operación 1 & 1 & 1 & 1)
OR:	| b	evalúa a 1 (se ejecuta la operación 0 | 1 | 0 | 1)
XOR:	^ a	evalúa a 0 (se ejecuta la operación 1 ^ 1 ^ 1 ^ 1)

Operadores de corrimiento

Símbolos: <<, >>

Los operadores de corrimiento son corrimiento a la izquierda y corrimiento a la derecha. El operador de corrimiento toma como parámetros un vector y un entero que le indica el número de corrimientos deseado. Los bits vacíos provocados por los corrimientos son llenados con ceros.

Sea a = 4'b1010 entonces,

Izquierda:	a << 1	corrimiento a la izquierda de un bit = 4'b0100
Derecha:	a >> 2	corrimiento a la derecha de dos bits = 4'b0010

Operadores de concatenación

Símbolos: { , }

Permite juntar nets, registros y constantes en una sola variable.

Sea a = 2'b01 y b = 6'b101001 entonces,

Concatenación:	{ a, 2'b11 }	produce un número de 4 bits {01,11} = 4'b0111
Concatenación:	{ b[5:3], a }	produce un número de 5 bits {101,01}= 5'b10101

Operador condicional

Este operador es una abreviatura de la sentencia if-else; en él la expresión condicional es evaluada, si esta expresión resulta verdadera entonces se procede a evaluar a expresión_verdadera, si no, se evaluará a expresión_falsa. La sintaxis para este operador es la siguiente.

```
expresión_condicional ? expresión_verdadera : expresión_falsa;
```

Precedencia de operadores

Si ningún paréntesis es utilizado para separar los operandos de una expresión, entonces, Verilog utiliza las siguientes reglas de precedencia para sus operadores. Generalmente es buena idea utilizar paréntesis para hacer las expresiones entendibles. A continuación se presenta la lista de los operadores y su precedencia.

Operador	*Precedencia*
+ - ! ~ (unarios)	La mayor
* / %	
+ - (binarios)	
<< >>	
< <= > >=	
== != === !==	
& ~&	
^ ^~	
| ~|	
&&	
||	
?: (condicional)	La menor

Tabla A.1. Precedencia de operadores.

Tipo	*Símbolo*	*Descripción*	*Número de operandos*
Aritméticos	*	Multiplicación	2
	/	División	2
	+	Suma	2
	-	Sustracción	2
	%	Módulo	2
Lógicos	!	NOT lógico	1
	&&	AND lógico	2
	||	OR lógico	2
Relación	>	Mayor que	2
	<	Menor que	2
	>=	Mayor o igual a	2
	<=	Menor o igual a	2
Igualdad	==	Igualdad lógico	2
	!=	Desigualdad lógico	2
	===	Igualdad case	2
	!==	Desigualdad case	2
Bitwise	~	Negación bitwise	1
	&	AND bitwise	2
	|	OR bitwise	2
	^	XOR bitwise	2
	~^ ó ^~	XNOR bitwise	2
Reducción	&	AND reducción	1
	~&	NAND reducción	1

	|	OR reducción	1
	~|	NOR reducción	1
	^	XOR reducción	1
	^~ ó ~^	XNOR reducción	1
Corrimiento	<<	Corrimiento a la izquierda	2
	>>	Corrimiento a la derecha	2
Concatenación	{ , }	Concatenación	Varios
Condicional	?:	Condicional	3

Tabla A.2. Operadores en Verilog.

Palabras Reservadas en Verilog

Las palabras reservadas son identificadores prefinidos que se utilizan para formar las construcciones de un lenguaje. Además, las palabras reservadas no pueden ser empleadas como identificadores de usuario. El conjunto de palabras reservadas que incorpora Verilog consiste de los siguientes 102 identificadores definidos en minúsculas.

Palabras Reservadas				
always	endprimitive	medium	real	time
and	endspecify	module	realtime	tran
assign	endtable	nand	reg	tranif0
begin	endtask	negedge	release	tranif1
buf	event	nmos	repeat	tri
bufif0	for	nor	rnmos	tri0
bufif1	force	not	rpmos	tri1
case	forever	notif0	rtran	triand
casex	fork	notif1	rtranif0	trior
casez	function	or	rtranif1	trireg
cmos	highz0	output	scalared	vectored
deassign	highz1	parameter	small	wait
default	if	pmos	specify	wand
defparam	ifnone	posedge	specparam	weak0
disable	initial	primitive	strong0	weak1
edge	inout	pull0	strong1	while
else	input	pull1	supply0	wire
end	integer	pullup	supply1	wor
endcase	join	pulldown	table	xnor
endmodule	large	rcmos	task	xor
endfunction	macromodule			

Tabla A.3. Palabras reservadas en Verilog.

Sea cuidadoso al utilizar las palabras reservadas de la lista anterior ya que para MAX+PLUS II no todas las palabras reservadas de Verilog son soportadas.

A.5 LÓGICA COMBINACIONAL EN VERILOG HDL

La lógica es combinacional si las salidas en el tiempo *T* son función sólo de las entradas en ese mismo tiempo. Ejemplos de funciones lógicas combinacionales son los decodificadores, los multiplexores, los sumadores, entre otros. En Verilog también es posible describir comportamientos combinacionales puros mediante el uso de *Asignaciones Continuas* y de *Construcciones Always*. Dichos comportamientos son independientes del flanco de subida o de bajada de un reloj.

A.5.1 ASIGNACIONES CONTINUAS

Cuando se utiliza una asignación continua, el compilador de MAX+PLUS II crea las compuertas lógicas necesarias para poder evaluar la expresión planteada. Las asignaciones continuas serán reevaluadas cada vez que el valor de las entradas cambie. El siguiente ejemplo emplea asignaciones continuas para crear una compuerta **and** de dos entradas, y para conectar dos nodos. Tenga en mente que estas declaraciones se ejecutan concurrentemente.

```
module Example_1 (a, b, e, c, d);
    input   a, b, e;
    output  c, d;
    assign c = a & b;
    assign d = e;
endmodule
```

En este programa, el valor de la operación *a&b* es asignado a *c* y el valor de la señal *e* es asignado a *d*. La siguiente figura muestra el diagrama lógico equivalente del programa anterior.

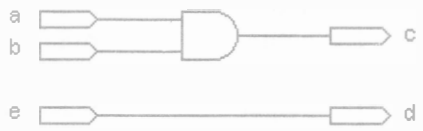


Figura A.14. Diagrama lógico.

A.5.2 CONSTRUCCIONES ALWAYS

Las construcciones always también permiten crear lógica combinacional si los eventos de control que manejan son no sensibles a los flancos de subida o de bajada de un reloj.

Si un evento de control ocurre, el código de la construcción asociado a ese evento se ejecuta y son calculadas las salidas conforme a las asignaciones procedurales establecidas. Estas asignaciones procedurales pueden estar compuestas de condicionales if, sentencia case, bucles for, entre otras.

En el siguiente ejemplo se utiliza una construcción always para crear un dispositivo que registra el número de bits '1' en un arreglo de entrada *d*.

```
module Example_2 (d, q);
    input   [2:0] d;
    output  [1:0] q;
    integer num_bits;

    always @ (d) begin: block
            integer i;
            num_bits = 0;
            for(i = 0; i < 3; i = i+1)
                    if( d[i]==1 )    num_bits = num_bits + 1;
    end
    assign q = num_bits;
endmodule
```

Para este ejemplo, el arreglo de entrada *d* es el evento de control, de manera que cuando *d* cambia de valor las asignaciones en la construcción always se ejecutan.

El bucle for inicializa a la variable *i* con cero. Si *i* es menor a 3, el bloque de sentencias del for se ejecuta. Dentro del for, la condicional if prueba si el i-ésimo elemento del arreglo de entrada es 1, si es así, incrementa en una unidad a *num_bits*, de lo contrario, el valor de *num_bits* no se altera. Finalmente, el valor de *i* se incrementa en una unidad y el proceso vuelve a ejecutarse si la condición del for continúa siendo válida.

El módulo termina asignando el valor del entero *num_bits* al arreglo de salida *q*. La variable *num_bits* se mantiene en cero si ningún elemento de *d* fue '1'.

A.5.3 CONSTRUCCIÓN DE MULTIPLEXORES

Un multiplexor es un dispositivo que permite dirigir la información digital procedente de diversas fuentes hacia una única línea de salida. La selección de la fuente dependerá de la secuencia de entrada especificada.

El siguiente ejemplo muestra una variante de este tipo de circuito, ya que maneja buses de entrada y de salida en lugar de líneas sencillas.

```
module busmux_8bit (selbus, low, high, Q);
    input           selbus;
    input   [7:0]   low, high;
    output  [7:0]   Q;

    assign Q = (selbus==0) ? low : high;
endmodule
```

Este ejemplo emplea el operador ternario **?** que sirve como sustituto de la sentencia *if-else*. Éste operador primero evalúa la expresión *selbus==0*. Si es verdadera, entonces evalúa la expresión *low*. El valor obtenido se convierte en el valor de la expresión. Si es falso, entonces evalúa la expresión *high* y su valor se convierte en el valor de la expresión. Cualquiera que fuese el valor obtenido será asignado a Q.

A.6 LÓGICA SECUENCIAL EN VERILOG HDL

La lógica es secuencial si las salidas en el tiempo *T* son función de las entradas en ese tiempo y en tiempos anteriores. Algunos ejemplos de lógica secuencial son las máquinas de estados, los contadores, los registros de corrimiento, los controladores, entre otros. La lógica secuencial es implantada en Verilog mediante construcciones always. El compilador interpreta estas construcciones y crea la circuitería lógica necesaria que será controlada por el reloj de la construcción o por otras señales de control.

A.6.1 CONSTRUCCIÓN DE REGISTROS

La creación de registros en Verilog se hace mediante inferencias a registros, es decir, utilizando variables del tipo 'reg'. Este tipo de variable permite crear registros con señal de reloj y cualquier combinación de señales clear, preset, load y enable síncronas.

También se requiere de una construcción always sensible a los flancos de subida o de bajada de un reloj. La sensibilidad de la construcción always estará dada por el evento de control definido después de la palabra reservada **always**.

- El reloj de una construcción always es la señal de control de la construcción. Se utilizan las palabras reservadas posedge y negedge para indicar flanco de subida o flanco de bajada, respectivamente. Dicha señal no puede utilizarse en el resto de la construcción. Además, sólo se permite una señal de reloj por construcción always.

- Las variables del tipo 'reg' son declaradas empleando la palabra reservada **reg**. Durante la síntesis lógica el compilador insertará automáticamente una o varias instancias de flip-flops y los conectará como se haya definido en las asignaciones procedurales.

También es posible construir registros explícitamente si se utilizan instancias de módulos. Sin embargo, las inferencias a registros son independientes de la arquitectura a diferencia de las instancias de módulos.

El siguiente ejemplo crea un registro de 4 bits con señales de clear, preset y load síncronas.

```
module Example_3(d, clk, clear, preset, load, q);
    input  [3:0]  d;
    input         clk, clear, preset, load;
    output [3:0]  q;
    reg    [3:0]  q;

    always @ (posedge clk) begin
            if (clear)       q = 4'b0000;
            else if (preset) q = 4'b1111;
            else if (load)   q = d;
    end
endmodule
```

La construcción always define como evento de control a la señal *clk*. Esta señal sirve también como reloj para los flip-flops ya que no se utiliza en el resto de la construcción always.

El proceso espera hasta que se presenta un flanco de subida en el reloj (posedge *clk*). A continuación prueba el valor de la señal *clear*. Si *clear* vale 1, se coloca a ceros el contenido de los flip-flops, si no, se prueba el valor de *preset*. Si *preset* vale 1, se coloca a unos el contenido de los flip-flops, si no, se prueba el valor de la señal *load*. Finalmente, si *load* vale 1, el contenido del bus *d* se carga en los flip-flops; si ninguna de las condiciones resulta verdadera, entonces, el contenido de los flip-flops no se altera.

A.6.2 CONSTRUCCIÓN DE CONTADORES

En Verilog los contadores se construyen a partir de condicionales if y de construcciones always. Las condicionales if especifican la lógica para sumar o restar cierto valor al contenido del registro, mientras que la construcción always implanta la lógica secuencial que contabiliza los pulsos de un reloj.

El siguiente ejemplo ilustra la creación de un registro contador de 4 bits con señales de *clear*, *load* y *up_down*.

```
module Example_4(d, clk, clear, load, up_down, q);
    input   [3:0]   d;
    input           clk, clear, load, up_down;
    output  [3:0]   q;
    reg     [3:0]   q;
    integer direction;

    always @ (posedge clk) begin
            if (up_down)    direction = 1;
            else            direction = -1;
            if (clear)      q = 0;
            else if (load)  q = d;
            else            q = q + direction;
    end
endmodule
```

En este ejemplo la construcción always es sensible sólo a los cambios en el reloj. El resto de la señales son síncronas, pero no son señales de control de la construcción.

En primer lugar se prueba el valor de la señal *up_down*. Si *up_down* vale 1, entonces se coloca la variable *direction* a 1, es decir, se establece una cuenta ascendente. En caso contrario, *direction* se coloca a –1, es decir, una cuenta descendente.

A continuación se revisa el valor de la señal *clear*. Si *clear* vale 1, los flip-flops se cargan con ceros, si no, se revisa el valor de la señal *load*. Si *load* vale 1, el dato presente en el bus *d* es cargado en los flip-flops. En caso de que *clear* y *load* valgan cero, el contenido del registro será incrementado o decrementado dependiendo del valor de *direction*.

A.6.3 CONSTRUCCIÓN DE LATCHES

Un latch puede construirse a partir de una condicional if y de una construcción always no sensible a los flancos del reloj. Por ejemplo:

```
module Example_5(d, enable, q);
    input       d, enable;
    output      q;
    reg         q;

    always @ (enable or d)
            if (enable) q = d;
endmodule
```

Este procedimiento se ejecuta cada vez que la señal *enable* o la señal *d* cambian de valor. El cuerpo del procedimiento consta de una condicional if que prueba el valor de la señal *enable*. Si *enable* vale 1, el valor de *d* es asignado a *q*, de lo contrario, el circuito mantiene su estado previo.

A.6.4 CONSTRUCCIÓN DE MÁQUINAS DE ESTADOS

Para implantar una máquina de estados utilizaremos dos construcciones always. La primera se encargará de generar las salidas de acuerdo al estado actual. Y la segunda calculará el estado siguiente en base a las entradas y al estado presente. En otras palabras, la segunda construcción always calculará las transiciones entre estados.

Como ejemplo construiremos la siguiente máquina de estados.

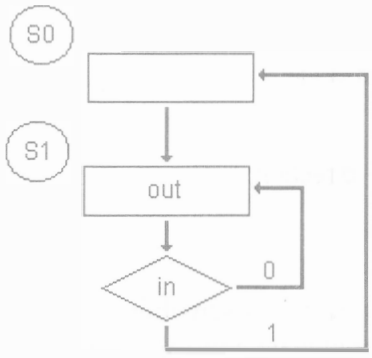


Figura A.15. Máquina de estados.

El código fuente que la implanta es el siguiente:

```
module Example_6(clk, in, reset, out);
    input       clk, in, reset;
    output      out;
    reg         out, state;
    parameter   s0 = 0, s1 = 1;
```

```
    always @ (state) begin
            case (state)
                    s0:     out = 0;
                    s1:     out = 1;
            endcase
    end

    always @ (posedge clk or posedge reset) begin
            if (reset)      state = s0;
            else
                    case (state)
                            s0:     state = s1;
                            s1:     if (in)   state = s0;
                                    else      state = s1;
                    endcase
    end
endmodule
```

El programa utiliza una variable del tipo reg, denominada *state*, para guardar el estado actual de la máquina de estados y dos parámetros *s0=0* y *s1=1* que corresponden a los nombres de los estados con su respectiva asignación binaria.[14] Además, se emplean dos construcciones always: una para generar las salidas y otra para calcular el estado siguiente.

La primera construcción always está pendiente de algún cambio en la variable *state*. Cada vez que este cambio se presenta la salida *out* se actualiza con el valor correcto.

La segunda construcción always tiene como evento de control a las señales *clk* y *reset*, de manera que el procedimiento se ejecuta cuando se registra un flanco de subida en algunas de ellas. Dentro de este procedimiento se revisa el valor de la señal *reset*. Si éste vale 1, entonces la máquina de estados regresa a su estado inicial, de lo contrario, se calcula el estado siguiente en base a las entradas presentes y al estado actual.

La siguiente figura muestra un diagrama de la máquina de estados utilizando los métodos tradicionales para su obtención.

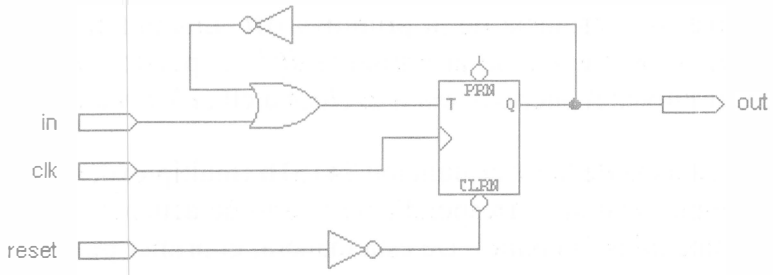


Figura A.16. Implantación de la máquina de estados utilizando métodos tradicionales.

[14] Para este ejemplo, los parámetros *s0* y *s1* definen constantes que serán utilizadas a lo largo del módulo. Sin embargo, los parámetros también permiten crear módulos parametrizados como los que estudiaremos más adelante.

A.7 PROYECTOS JERÁRQUICOS

Los archivos de diseño de Verilog pueden combinarse en un proyecto jerárquico con otros archivos de diseño de Verilog, archivos de diseño VHDL, TDFs, GDFs, archivos de entrada EDIF y archivos esquemáticos OrCAD.

A.7.1 CÓMO EMPLEAR FUNCIONES LÓGICAS DE MAX+PLUS II DENTRO DE UN PROYECTO DE VERILOG

En Verilog se pueden utilizar instancias de módulos para insertar funciones lógicas de MAX+PLUS II. Entre estas funciones encontramos a las primitivas, las megafunciones y las macrofunciones.

Cuando se procesa la instancia del módulo, el compilador agrega el archivo **Include** adecuado y automáticamente conecta los puertos de la función lógica con los puertos de nuestro módulo.

- Verilog no soporta directamente nombres de funciones o de puertos que comiencen con números. Si se desea emplear un nombre de este tipo debe escribirse el prefijo '\', por ejemplo, para usar la macrofunción 7480 debemos declararla de la siguiente manera: \7480. Durante el procesamiento el software MAX+PLUS II removerá el prefijo '\' de cualquier función o puerto declarado.

El siguiente ejemplo muestra cómo utilizar las instancias de módulos.

```
module Example_7 ( data, clock, clearn, presetn, a, b, c, gn, d, q_out, y, wn );
    input          data, clock, clearn, presetn, a, b, c, gn;
    input  [7:0]   d;
    output         q_out, y, wn;

    dff        dff1  ( .d (data), .q (q_out), .clk (clock), .clrn (clearn), .prn (presetn) );
    \74151b    mux   ( c, b, a, d, gn, y, wn );
endmodule
```

En primer lugar, se crea una instancia de la primitiva **dff**. Los puertos de nuestro módulo son conectados explícitamente a los puertos de la primitiva *dff1*: el puerto *data* de nuestro módulo se asigna al puerto d de la primitiva, el puerto *q_out* a q, *clock* a clk, *clearn* a clrn y *presetn* a prn.

También se crea una instancia de la macrofunción **74151b** (multiplexor). Los parámetros pasados a la instancia *mux* son asignados a su correspondiente puerto de acuerdo al prototipo de la función (ubicado en el directorio \maxplus2\maxinc). De esta manera, el primer puerto en la función 74151b es conectado a *c*, el segundo puerto a *b*, y así sucesivamente.

A continuación se presenta el circuito equivalente utilizando el editor gráfico de MAX+PLUS II.

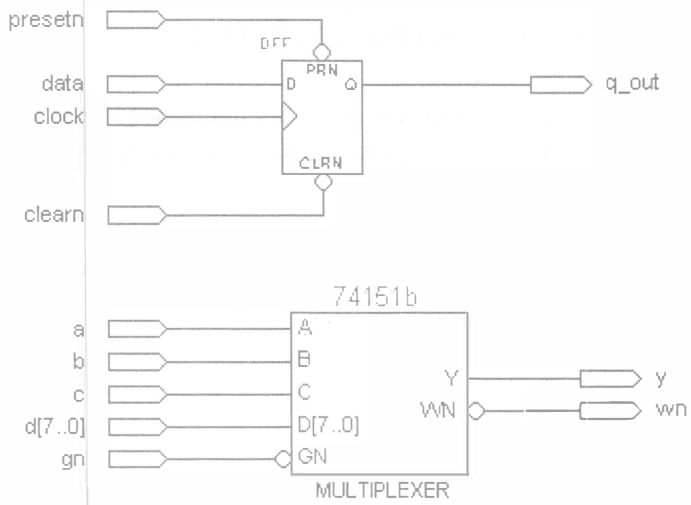


Figura A.17. Circuito equivalente utilizando el editor gráfico de MAX+PLUS II.

A.7.2 CÓMO EMPLEAR INSTANCIAS DE COMPUERTAS LÓGICAS DENTRO DE UN PROYECTO DE VERILOG

En Verilog también se pueden utilizar instancias de compuertas lógicas. El siguiente ejemplo muestra como hacer esto.

```
module Example_8(a, b, c, d, x, y);
    input   a, b, c, d;
    output  x, y;

    and     myand ( x, a, b, c );
    not     not1 ( y, d );
endmodule
```

Se utilizan dos instancias de compuertas lógicas, una **and** o un **not**. Los parámetros pasados a cada instancia son las señales de entrada/salida de nuestro módulo que serán asignadas a los puertos correspondientes de las compuertas. Observe que los puertos de salida en ambas instancias son listados antes que los puertos de entrada, este orden está especificado en el prototipo de la función.

El siguiente diagrama corresponde al circuito equivalente.

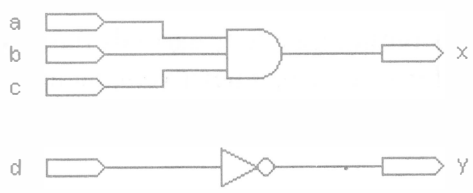


Figura A.18. Circuito equivalente empleando el editor gráfico de MAX+PLUS II.

A.7.3 CÓMO UTILIZAR FUNCIONES DE USUARIO DENTRO DE UN PROYECTO DE VERILOG

Si deseamos utilizar instancias de nuestros módulos es necesario crear un archivo Include con el prototipo de la función. Este archivo puede crearse mediante la opción **Create Default Include File** ubicada en el menú **File** de la barra de herramientas.

Ejemplo. Teclee el siguiente código en el editor de texto de MAX+PLUS II. Este programa creará un registro de 12 bits muy sencillo.

```
module Register_12bit(d, clk, q);
    input   [11:0]  d;
    input           clk;
    output  [11:0]  q;
    reg     [11:0]  q;

    always @ (posedge clk)
        q = d;
endmodule
```

A continuación guarde el archivo con el nombre 'Register_12bit.v'. Abra el menú 'FILE' de la barra de herramientas y elija la opción 'Create Default Include File'. Entonces aparecerá la interfaz del compilador mostrando el avance en la creación del archivo Include y presentará un cuadro de diálogo una vez que lo haya terminado.

Ahora teclee el siguiente código, guárdelo con el nombre 'Register24_bit.v' y compílelo. Este programa creará un registro de 24 bits utilizando instancias del módulo Register_12bit.

```
module Register_24bit(d, clk, q);
    input   [23:0]  d;
    input           clk;
    output  [23:0]  q;

    register_12bit      reg12a (.q  (q[11:0]),  .d (d[11:0]),  .clk (clk) );
    register_12bit      reg12b (.q (q[23:12]), .d (d[23:12]), .clk (clk) );
endmodule
```

Asegúrese de indicarle al compilador la ruta en donde se encuentra el archivo 'Register_12bit.v' y el archivo 'Register_12bit.inc', ya que de lo contrario será incapaz de ligar ambos programas. La declaración de esta ruta podrá omitirse si los programas Register_12bit y Register_24bit fueron guardados en el mismo directorio.

Para colocar el path o ruta de las bibliotecas de usuario diríjase al menú '*Options*' de la barra de herramientas y seleccione '*User Libraries*'. Una nueva ventana aparecerá en donde podrá agregar o borrar los directorios con las bibliotecas de usuario.

A.7.4 CÓMO EMPLEAR FUNCIONES PARAMETRIZADAS EN VERILOG

En Verilog también se pueden utilizar instancias de funciones parametrizadas. Entre ellas destacan las funciones parametrizadas definidas por el usuario y las funciones de la biblioteca de módulos parametrizados (LPM).

Declarar instancias de funciones parametrizadas y de funciones normales es muy parecido, salvo por los detalles que se enuncian a continuación.

- La instancia de la función parametrizada utiliza declaraciones del tipo **defparam** para definir los valores de sus parámetros. En ocasiones estos valores se pueden omitir, ya que algunos parámetros tienen asociados valores por default.

  La sintaxis de la declaración defparam es la siguiente:

  defparam *<nombre instancia>* **.** *<nombre parámetro>* = *<valor parámetro>*;

  El valor de un parámetro puede ser del tipo entero, una expresión aritmética o una cadena de texto.

- Una declaración defparam sólo puede pasar parámetros a archivos de diseño de un nivel menor que el archivo actual. Por ejemplo, si el archivo *top.v* crea una instancia de *middle.v* y éste último crea una instancia de *bottom.v*, *top.v* puede pasar parámetros a *middle.v*, pero no a *bottom.v*. Si desea pasar este tipo de parámetros es necesario que *middle.v* los pase explícitamente hacia *bottom.v*.

- Como se mencionó anteriormente, no todos los parámetros de una función parametrizada tienen valores por default, por lo tanto, es importante asegurar que al menos los parámetros indispensables sí los tienen, de lo contrario, habrá errores en el diseño. También hay funciones parametrizadas en las que no se requieren conectar todos los puertos de entrada, por ello, incorporan un conjunto de parámetros que deshabilitan a esos puertos, o bien, les asignan valores por default. Algo similar ocurre para las primitivas y las macrofunciones sólo que estas sí tienen valores asignados para entradas sin conectar y no es necesario preocuparnos por asignarles unos.

Para ilustrar el uso de las funciones parametrizadas crearemos un registro de 24 bits como el visto en el ejemplo anterior, pero utilizando la función parametrizada **lpm_dff** que corresponde a un registro flip-flop tipo D.

```
module Example_9 (d, clk, q);
    input   [23:0] d;
    input          clk;
    output  [23:0] q;

    lpm_dff        reg12a (  .q (q[11:0]),  .data (d[11:0]), .clock (clk) );
                   defparam  reg12a.lpm_width = 12;

    lpm_dff        reg12b ( .q (q[23:12]), .data (d[23:12]), .clock (clk) );
                   defparam  reg12b.lpm_width = 12;
endmodule
```

Note que el parámetro **lpm_width** especifica el ancho de los buses de entrada y de salida, o visto de otra manera, el número de flip-flops que integran a cada instancia. La función lpm_dff también cuenta con otros parámetros, sin embargo, no fueron declarados porque ya tienen asignados valores por default.

4.7.5 CÓMO CREAR MEMORIAS ROM Y RAM EN VERILOG

Altera no recomienda crear funciones personalizadas para implantar memorias, por ello, el software MAX+PLUS II cuenta con varias funciones **LPM** y megafunciones que permiten crear memorias RAM y ROM en los dispositivos de Altera.[15] Dichas funciones pueden incorporarse fácilmente en los archivos de diseño de Verilog.

lpm_ram_dq	RAM síncrona o asíncrona con puertos de entrada/salida separados
lpm_ram_io	RAM síncrona o asíncrona con un sólo puerto de entrada/salida (bidir)
lpm_rom	ROM síncrona o asíncrona
csfifo	Buffer de tipo FIFO (first-in first-out)

El siguiente ejemplo crea una memoria RAM síncrona de 8x8 (8 palabras de 8 bits cada una). El parámetro **lpm_width** sirve para especificar el número de líneas de los buses de entrada y salida de datos, mientras que el parámetro **lpm_widthad** especifica el número de líneas del bus de direcciones.

```
module Example_10 (data, address, r_w, clk1, clk2, contents);
    input   [7:0]  data;
    input   [2:0]  address;
    input          r_w, clk1, clk2;
    output  [7:0]  contents;

    lpm_ram_dq  myram ( .q(contents), .data(data), .address(address), .we(r_w), .inclock(clk1),
                .outclock(clk2) );
                defparam myram.lpm_width = 8;
                defparam myram.lpm_widthad = 3;
endmodule
```

[15] Algunas memorias sólo pueden implantarse en los dispositivos de la familia **FLEX10K** de Altera o superiores.

APÉNDICE B
CONSTRUCCIÓN DE UN MICROPROCESADOR UTILIZANDO VERILOG HDL Y AHDL

B.1 NOTA IMPORTANTE

Esta sección intenta explicar cómo fueron implantados los componentes del microprocesador de 8 bits estudiado en los capítulos 5 y 6. Pero antes, es necesario aclarar ciertos aspectos de diseño que fueron modificados para concluir con éxito la construcción del microprocesador.

1. Todos los buses bidireccionales que se manejaron en la literatura fueron divididos en dos partes: el bus de entrada por donde los datos ingresan al registro, y el bus de salida por donde los datos contenidos en el registro salen hacia el resto de la arquitectura. Esta división fue necesaria debido a que las celdas lógicas internas de los circuitos programables de Altera, no permiten el manejo de buses bidireccionales. Este manejo se restringe sólo a los pines de salida del circuito.

2. Todos los circuitos latches fueron sustituidos por flip-flops debido a los problemas que se presentaron durante las simulaciones. Entre estos problemas destacan: la inestabilidad del circuito para tiempos de carga muy pequeños y los constantes glitches (transitorios).

3. El último aspecto se refiere a los lenguajes de descripción de hardware utilizados: Verilog HDL y Altera HDL. Verilog HDL fue elegido debido a su universalidad en el campo de los circuitos digitales programables. En cambio, AHDL se eligió porque el software MAX+PLUS II genera código optimizado para los circuitos programables de Altera, circuitos que empleamos para realizar las pruebas físicas. Además, otra razón de utilizar AHDL fue que el software MAX+PLUS II en su versión 10 no tiene implantadas todas las funciones y capacidades del lenguaje Verilog, como por ejemplo, no permite definir primitivas que son tipos de datos definidos por el usuario.

La construcción de este microprocesador la realizaremos en tres etapas.

En la primera etapa, implantaremos el hardware que realiza las operaciones lógico/aritméticas y todos los registros que componen al microprocesador, como son los registros índices, los acumuladores, el contador de programa, el apuntador de pila, el registro de banderas, los registros de interrupciones y el registro de direcciones. En la segunda etapa implantaremos la circuitería necesaria para controlar el hardware de la primera fase, de manera que todo funcione automáticamente. Esta circuitería estará formada por la unidad de control y la unidad de control de interrupciones. La tercera etapa estará dedicada a enlazar los dos módulos anteriores, para así obtener un sólo módulo que opere de manera similar al microcontrolador M68HC11.

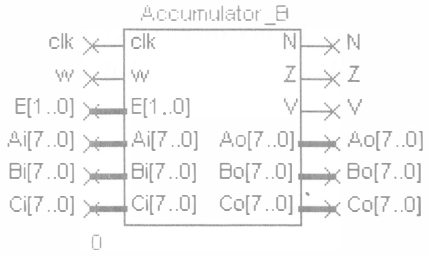
B.2 ETAPA 1: CAMINO DE DATOS

B.2.1 REGISTROS ACUMULADORES

Básicamente, los acumuladores funcionan como dispositivos de almacenamiento. Estos registros almacenan los datos que la ALU operará y guardan el resultado de dicha operación.

Cómo funciona el registro acumulador

El siguiente diagrama muestra las entradas y las salidas que componen al registro acumulador y se explica brevemente la función de cada una de ellas. No está de más mencionar, que todos los diagramas presentados en estos apéndices se generan tras la compilación de sus respectivos códigos fuente.



Entradas:

clk	Señal de reloj (disparo con flanco positivo)
w	Si w es igual a '0' lógico, el registro se habilita para escritura (carga de datos) Si w vale 1, estará habilitado para lectura
E[1..0]	Permite la selección de uno de los buses de entrada ('01' = Ai, '10' = Bi, '11' = Ci)
Ai[7..0]	Bus de entrada de 8 bits que junto con Ao forman el bus bidireccional A
Bi[7..0]	Bus de entrada de 8 bits que junto con Bo forman el bus bidireccional B
Ci[7..0]	Bus de entrada de 8 bits que junto con Co forman el bus bidireccional C

Salidas:

N	Bandera de negativo, el resultado es negativo (el bit 7 vale '1' lógico)
Z	Bandera de cero, resultado igual a cero (todos los bits valen '0' lógico)
V	Bandera de overflow, los bits fueron insuficientes para representar cierto valor
Ao[7..0]	Bus de salida de 8 bits que junto con Ai forman el bus bidireccional A
Bo[7..0]	Bus de salida de 8 bits que junto con Bi forman el bus bidireccional B
Co[7..0]	Bus de salida de 8 bits que junto con Ci forman el bus bidireccional C

Análisis del módulo Accumulator_B.v

Observemos detenidamente cada línea de este programa. Las líneas,

```
/*************************************************************************
 * Módulo Accumulator B -> accumulator_b.v                               *
 *************************************************************************/
```

son un comentario. En Verilog, los comentarios empiezan con la secuencia /* y terminan con */. Todo lo que esté entre los delimitadores de principio y fin de comentario será ignorado por el compilador.

Verilog también permite los comentarios de una línea, los cuales comienzan con la secuencia //. Lo que esté escrito después de la secuencia // será ignorado por el compilador hasta llegar al final de

la línea. De la misma manera, las líneas en blanco son admisibles y no tienen efecto sobre el programa.

La siguiente línea,

```
module accumulator_b (clk, E, w, Ai, Bi, Ci, Ao, Bo, Co, N, Z, V);
```

declara un módulo, que es la unidad básica de un diseño en Verilog. Un módulo es simplemente un bloque de construcción que ejecuta ciertas tareas dentro del diseño. Note que la definición del módulo consta de un nombre y de la lista de parámetros necesarios (señales de entrada y de salida).

Las siguientes líneas en el código son:

```
/*************************************************************************
 * Parámetros:                                                           *
 *          clk         - señal de reloj                                 *
 *          E           - selecciona alguno de los buses de entrada      *
 *          w           - señal de escritura del registro                *
 *          Ai/Bi/Ci    - buses de entrada de datos                      *
 *          Ao/Bo/Co    - buses de salida de datos                       *
 *          N           - bandera de negativo                            *
 *          Z           - bandera de cero (resultado igual a cero)       *
 *          V           - bandera de sobreflujo                          *
 *************************************************************************/
```

que también son un comentario.

A continuación son declaradas las señales de entrada y de salida, mismas que se enunciaron en la lista de parámetros del módulo. Las señales de entrada están precedidas por la palabra **input** y las señales de salida por la palabra **output**.

Si alguna de las señales está formada por más de una línea, entonces se trata de un vector y es necesario definir su tamaño antes del nombre de la señal. El tamaño es un rango de valores que se escribe dentro de corchetes; por ejemplo, la señal Ai es una señal de entrada formada por ocho líneas cuyos índices son: 0, 1, 2, 3, 4, 5, 6 y 7, por lo tanto, el tamaño de la señal es [7:0]. Observe que es posible declarar varias variables del mismo tipo si éstas se separan mediante comas.

```
input       clk, w;
input [1:0] E;
input [7:0] Ai, Bi, Ci;
output      N, Z, V;
output[7:0] Ao, Bo, Co;
```

La siguiente línea define dos variables internas del tipo **reg** para el manejo interno del acumulador. En particular, la variable *temp* almacena el contenido del registro y la variable *t2* guarda el dato multiplexado antes de cargarlo en el registro.

```
reg   [7:0] temp, t2;
```

La siguiente construcción **always** ejecuta la función de un multiplexor:

```
always @ (E) begin
        case (E)
                2'b01:  t2 = Ai;        // Selecciona el bus A
                2'b10:  t2 = Bi;        // Selecciona el bus B
                2'b11:  t2 = Ci;        // Selecciona el bus C
                default:t2 = temp;      // Mantiene el último valor
        endcase
end
```

El bloque anterior define un procedimiento, es decir, una tarea que se ejecutará cada vez que la señal *E* cambia de valor. Si recordamos, la señal *E* selecciona uno de los buses de entrada, por lo tanto, si esta señal cambia de valor, la variable *t2* guardará el dato del bus seleccionado.

La definición del procedimiento comienza con las palabras **always @**, y enseguida, entre paréntesis, la señal a sensar. La etiqueta **begin** que sucede a la definición le indica al compilador el inicio del procedimiento, el cual concluirá al momento que se alcance su respectiva etiqueta **end**.

A continuación aparece una sentencia **case** que va comparando sucesivamente el valor de la señal *E* contra una lista de constantes. Cuando se encuentra una coincidencia, se ejecuta la sentencia o bloque de sentencias asociadas a esa constante. El final de la sentencia case lo establece la etiqueta **endcase**. Por ejemplo, supongamos que la señal *E* cambia su valor a 10 binario; por lo tanto, el procedimiento entra en acción. Al llegar a la sentencia case, se compara el valor de *E* contra los valores de las constantes declaradas, como *E* iguala a la constante 2'b10, entonces, el valor de la señal *Bi* es cargado en la variable *t2*.

Recuerde que las constantes en Verilog son declaradas de la siguiente manera: primero se coloca el número de bits necesarios para representar la constante, a continuación se coloca un apóstrofo seguido de la letra que indica el sistema de numeración de la constante (b-binario, d-decimal, o-octal, h-hexadecimal), y por último la constante en el formato definido.

Si continuamos con la revisión del programa notaremos que está declarado otro procedimiento. Ahora la señal a sensar es el reloj, de manera que cuando se registra un flanco de subida en él, el procedimiento es ejecutado.

```
always @ (posedge clk) begin
        if (!w)
                temp = t2;      // El dato multiplexado se carga en el registro
end
```

La sentencia **if** se comporta de manera similar a la de otros lenguajes de programación. Si la condición que se evalúa en el if tiene un valor verdadero ('1' lógico), entonces, la o las sentencias que constituyan el bloque if serán ejecutadas. En caso contrario, si existe, se ejecutará el código asociado al **else**, pero nunca los dos.

Supongamos que se registra un flanco de subida en el reloj, y además la señal *w* está en '0' lógico, es decir, el registro se habilita para cargar un dato. Cuando se entra al procedimiento, la

condición del if se vuelve verdadera (debido a la negación del valor de *w*) y el contenido de *t2* es cargado en la variable *temp*. En caso de que *w* valiera '1' lógico no se ejecutaría la sentencia del bloque if, pues la condición sería falsa ('1' lógico negado produce '0' lógico).

Las líneas siguientes del código calculan las banderas del registro y las asignan a sus respectivas señales de salida.

```
// Calcula los valores de las banderas
assign N = temp[7];
assign Z = ~(| temp);
assign V = (~temp[7]) & (& temp[6:0]);
```

La bandera de negativo se calcula con el valor del bit más significativo, y se asigna a la señal de salida *N*. La bandera de cero se calcula al realizar la operación lógica OR entre los valores de los 8 bits, y se niega con el fin de que produzca un '1' lógico cuando todos los bits valgan cero. Finalmente, la bandera de overflow se calcula al realizar la operación AND entre los 7 bits menos significativos y el octavo bit negado.

Recuerde que en Verilog, la operación lógica AND se representa con el operador &, la operación lógica OR con el operador |, y la operación lógica NOT con el operador ~ ó !.

A continuación se asignan las señales de salida *Ao*, *Bo* y *Co*. La función que se utiliza para ello es un método de Verilog, **bufif1( )**, que actúa como un circuito tres estados, de manera que si la expresión evaluada es verdadera, el circuito permite la salida de los datos; en caso contrario, el circuito permanecerá en alta impedancia.

Para el caso de la señal de salida *Ao*, la expresión a evaluar es ~*E*[1]&*E*[0]&*w*, que será verdadera cuando *E*[1] valga '0', *E*[0] valga '1' y *w* valga '1'; es decir, el bus *A* esté seleccionado y la señal *w* indique lectura de datos.

```
// Asigna la salida Ao
bufif1 (Ao[7], temp[7], ~E[1] & E[0] & w);
bufif1 (Ao[6], temp[6], ~E[1] & E[0] & w);
bufif1 (Ao[5], temp[5], ~E[1] & E[0] & w);
bufif1 (Ao[4], temp[4], ~E[1] & E[0] & w);
bufif1 (Ao[3], temp[3], ~E[1] & E[0] & w);
bufif1 (Ao[2], temp[2], ~E[1] & E[0] & w);
bufif1 (Ao[1], temp[1], ~E[1] & E[0] & w);
bufif1 (Ao[0], temp[0], ~E[1] & E[0] & w);
```

Algo similar ocurre con la salida *Bo* sólo que en esta ocasión el bus seleccionado es B (la señal **E** vale '10' y *w* vale '1' lógico).

```
// Asigna la salida Bo
bufif1 (Bo[7], temp[7], E[1] & ~E[0] & w);
bufif1 (Bo[6], temp[6], E[1] & ~E[0] & w);
bufif1 (Bo[5], temp[5], E[1] & ~E[0] & w);
bufif1 (Bo[4], temp[4], E[1] & ~E[0] & w);
bufif1 (Bo[3], temp[3], E[1] & ~E[0] & w);
```

```
bufif1 (Bo[2], temp[2], E[1] & ~E[0] & w);
bufif1 (Bo[1], temp[1], E[1] & ~E[0] & w);
bufif1 (Bo[0], temp[0], E[1] & ~E[0] & w);
```

La salida *Co* no utiliza un circuito tres estados, la variable *temp* se asigna directamente a *Co*.

```
// Asigna la salida Co
assign Co = temp;
```

Finalmente, debemos concluir la definición del módulo con la etiqueta **endmodule**, que le indica al compilador la terminación del mismo.

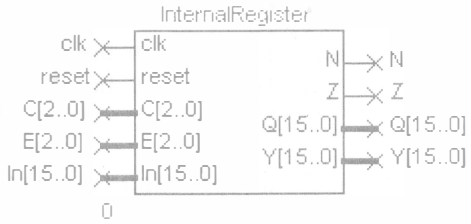
```
endmodule
```

B.2.2 REGISTRO CONTADOR DE 16 BITS

Este tipo de registro también funciona como una unidad de almacenamiento; sin embargo, el contador de 16 bits integrado le da mayor funcionalidad, pues le permite incrementar o decrementar el valor almacenado cuando sea necesario.

Cómo funciona el registro contador

El siguiente diagrama muestra las entradas y las salidas que componen al registro contador. Además, se explica brevemente la función de cada una de ellas.



Entradas:

clk	Señal de reloj (disparo con flanco positivo)
reset	Coloca el contenido del registro a ceros
C[2..0]	Controlan el funcionamiento del contador, de manera que permita realizar incrementos, decrementos y cargas de datos
E[2..0]	Permite seleccionar uno ó varios buses de salida
In[15..0]	Bus de entrada de 16 bits. Los 8 bits más significativos de este bus corresponden al bus de entrada C y los 8 bits menos significativos al bus de entrada D[16]

Salidas:

N	Bandera de negativo, el resultado es negativo (el bit 7 vale '1' lógico)

[16] Estos buses se estudiaron en el Capítulo 5 en el apartado del registro contador.

Z	Bandera de cero, el resultado es igual a cero
Q[15..0]	Bus de salida de 16 bits. Los 8 bits más significativos de este bus corresponden al bus de salida C y los 8 bits menos significativos al bus de salida D
Y[15..0]	Bus de salida de 16 bits correspondiente al bus de salida E

Análisis del módulo InternalRegister.v

Las primeras líneas del código son comentarios.

```
/*************************************************************************
 * Module: Registro Interno -> internalregister.v                        *
 *************************************************************************/
```

Enseguida, se declara un módulo llamado *internalregister* y el conjunto de señales que lo componen.

```
module internalregister (clk, C, reset, In, E, N, Z, Q, Y);
```

Las siguientes líneas también son comentarios.

```
/*************************************************************************
 * Parámetros:                                                           *
 *          clk         - señal de reloj                                 *
 *          C           - señales de control                             *
 *          reset       - señal de clear síncrona                        *
 *          In          - bus de entrada de datos                        *
 *          E           - señales de habilitación para los buses de salida *
 *          N           - bandera de negativo                            *
 *          Z           - bandera de cero                                *
 *          Q           - bus de salida hacia el bus de datos interno    *
 *          Y           - bus de salida hacia el bus de direcciones      *
 *************************************************************************/
```

A continuación, se define el tipo y tamaño de las señales declaradas previamente.

```
input          clk, reset;
input  [ 2:0]  C, E;
input  [15:0]  In;
output         N, Z;
output [15:0]  Q, Y;
```

Nuevamente hacemos uso de variables temporales del tipo **reg** para manipular algunos datos importantes. Por ejemplo, la variable *add_sub* indica qué operación debemos realizar: una suma para calcular el incremento, o bien, una resta para el decremento. La variable *Fa* sirve para almacenar temporalmente el resultado de la suma ó de la resta; la variable *Cin* guarda el valor del acarreo; y *temp* mantiene almacenado el contenido del registro.

```
reg            Z, add_sub, Cin;
reg    [15:0]  temp, Fa;
```

La siguiente línea declara una instancia del módulo **lpm_add_sub** llamada *addsub1*. Este módulo viene integrado con el software MAX+PLUS II y es un sumador/restador.

La lista de parámetros que pasamos al módulo lpm_add_sub son las señales necesarias que requiere el módulo para operar. Por ejemplo, las señales *dataa* y *datab* son los operandos a sumar o restar, *cin* es el acarreo de entrada, *add_sub* selecciona la operación a realizar, y *result* entrega el resultado de la operación. Cada una de estas señales se asocia a una señal o variable en nuestro módulo.

```
lpm_add_sub addsub1 ( .result (Fa), .cin (Cin), .dataa (temp), .datab (16'h0000),
                      .add_sub (add_sub) );
```

Hasta el momento hemos declarado la utilización del módulo de suma/resta, sin embargo, para que ejecute operaciones de 16 bits es necesario ajustar el valor de un parámetro extra: el número de bits de los operandos y del resultado. La siguiente línea precisamente establece este valor, para ello, modificamos un parámetro predeterminado llamado **lpm_width**, que indica el tamaño en bits de los operandos del módulo lpm_add_sub.

```
defparam addsub1.lpm_width = 16;
```

El siguiente fragmento del código declara un procedimiento que se ejecutará cada vez que la señal de control *C* cambia de valor.

Básicamente lo que realiza el siguiente bloque es establecer los valores adecuados para las variables *Cin* y *add_sub* de nuestro módulo. Por ejemplo, si el valor de *C* es igual a 1, significa que el registro debe realizar un incremento, por lo tanto, los valores de *Cin* y *add_sub* se colocan a '1'. De esta forma, los parámetros recibidos por el módulo lpm_add_sub le indicarán que ejecute la suma. Si *C* vale 2 (se tratará de un decremento), los valores de *Cin* y *add_sub* se colocarán a '0' y el módulo lpm_add_sub ejecutará una resta. Recuerde que el valor de la operación ejecutada por el módulo lpm_add_sub será asignado a la variable *Fa*.

Si el valor de *C* no corresponde a alguno de los valores anteriores, entonces, no se asignan nuevos valores a las variables *Cin* y *add_sub*.

```
// Suma o resta
always @ (C) begin
        if (C==1) begin
                Cin = 1; add_sub = 1;
        end
        else if (C==2) begin
                Cin = 0; add_sub = 0;
        end
end
```

El siguiente bloque declara otro procedimiento que se ejecutará cada vez que se registre un flanco de subida en el reloj.

El procedimiento comienza revisando el valor de la señal de *reset*. Si éste vale '1' lógico, entonces el contenido del registro es colocado a ceros, en caso contrario, se ejecuta el bloque de sentencias que componen al **else**. Dentro del else se encuentra una sentencia **case**, que va comparando el valor de la señal *C* con las constantes definidas. Si encuentra una correspondencia, se ejecuta el bloque de sentencias asociado con el valor de la constante. Si no existe correspondencia con alguna de las constantes, entonces se ejecuta el bloque de sentencias asociado con la etiqueta **default**.

Si observamos detenidamente cada sentencia dentro del case notaremos que sólo se realizan asignaciones de las diferentes fuentes de datos a la variable *temp*, fuentes que tendrán los valores correctos al momento de ejecutar este procedimiento.

```
always @ (posedge clk) begin
        if (reset)                                          // Limpia el registro
                temp = 0;
        else begin
                case (C)
                        3'b000:   temp = temp;              // Mantiene el valor
                        3'b001:   temp = Fa;                // Incrementa
                        3'b010:   temp = Fa;                // Decrementa
                        3'b011:   temp[7:0] = In[7:0];      // Carga un byte en la parte baja
                        3'b100:   temp[15:8] = In[15:8];    // Carga un byte en la parte alta
                        3'b101:   temp = In;                // Carga un dato de 16-bit
                        default:  temp = temp;              // Mantiene el valor
                endcase
        end // Fin del else
```

Este procedimiento termina calculando la bandera de cero, para ello, ejecuta una operación lógica OR entre los 16 bits que componen al registro contador, y niega el valor obtenido para hacer que Z valga '1' cuando todos los bits del registro valgan '0'.

```
        // Calcula la bandera de cero
        Z = ~(| temp);
end
```

En las siguientes líneas se calcula el valor de la bandera de negativo que corresponde al bit más significativo del registro (el bit de signo).

```
// Calcula la bandera de negativo
assign N = temp[15];
```

A continuación se hace uso de un método incorporado en el lenguaje de Verilog, **bufif0( )**, el cual funciona como un circuito tres estados. Este método evalúa una condición, si ésta resulta falsa ('0' lógico), el circuito se habilita para la salida de datos. Si resulta verdadera, el circuito permanece en alta impedancia.

Por ejemplo, para habilitar la salida de datos por el bus *Y* se evalúa la condición *E*[2]. De manera que si *E*[2] vale '0', entonces los datos de *temp* son asignados a *Y*. En caso contrario, *Y* permanece en alta impedancia.

```
// Habilita o deshabilita la salida Y
bufif0 (Y[15], temp[15], E[2]);
bufif0 (Y[14], temp[14], E[2]);
bufif0 (Y[13], temp[13], E[2]);
bufif0 (Y[12], temp[12], E[2]);
bufif0 (Y[11], temp[11], E[2]);
bufif0 (Y[10], temp[10], E[2]);
bufif0 ( Y[9],  temp[9], E[2]);
bufif0 ( Y[8],  temp[8], E[2]);
bufif0 ( Y[7],  temp[7], E[2]);
bufif0 ( Y[6],  temp[6], E[2]);
bufif0 ( Y[5],  temp[5], E[2]);
bufif0 ( Y[4],  temp[4], E[2]);
bufif0 ( Y[3],  temp[3], E[2]);
bufif0 ( Y[2],  temp[2], E[2]);
bufif0 ( Y[1],  temp[1], E[2]);
bufif0 ( Y[0],  temp[0], E[2]);
```

Algo similar ocurre para el bus *Q*. Si *E*[1] vale '0' la parte alta del bus *Q* estará habilitada. Si *E*[0] vale '0' la parte baja de *Q* también se habilitará. Y si alguna de estas señales estuviera a '1' lógico una parte del circuito permanecería deshabilitada (en alta impedancia).

```
// Habilita o deshabilita la salida Q
bufif0 (Q[15], temp[15], E[1]);
bufif0 (Q[14], temp[14], E[1]);
bufif0 (Q[13], temp[13], E[1]);
bufif0 (Q[12], temp[12], E[1]);
bufif0 (Q[11], temp[11], E[1]);
bufif0 (Q[10], temp[10], E[1]);
bufif0 ( Q[9],  temp[9], E[1]);
bufif0 ( Q[8],  temp[8], E[1]);
bufif0 ( Q[7],  temp[7], E[0]);
bufif0 ( Q[6],  temp[6], E[0]);
bufif0 ( Q[5],  temp[5], E[0]);
bufif0 ( Q[4],  temp[4], E[0]);
bufif0 ( Q[3],  temp[3], E[0]);
bufif0 ( Q[2],  temp[2], E[0]);
bufif0 ( Q[1],  temp[1], E[0]);
bufif0 ( Q[0],  temp[0], E[0]);
```

El programa termina con la etiqueta **endmodule** que indica la finalización de un módulo.

```
endmodule
```

B.2.3 REGISTRO DE ESTADOS O DE BANDERAS

El registro de estados está compuesto por 8 flip-flops que guardan el valor de ciertas variables llamadas banderas, las cuales indican el estado de los componentes de la arquitectura.

C	Bit 0	Bandera de acarreo
V	Bit 1	Bandera de overflow (sobreflujo)
Z	Bit 2	Bandera de cero
N	Bit 3	Bandera de negativo
I	Bit 4	Bandera de interrupción $\overline{\text{IRQ}}$
H	Bit 5	Bandera de medio acarreo
X	Bit 6	Bandera de interrupción $\overline{\text{XIRQ}}$
S	Bit 7	Bandera de stop

Cómo funciona el registro de banderas

El siguiente diagrama muestra las entradas y las salidas que componen a este registro; además, se explica brevemente la función de cada una de ellas.

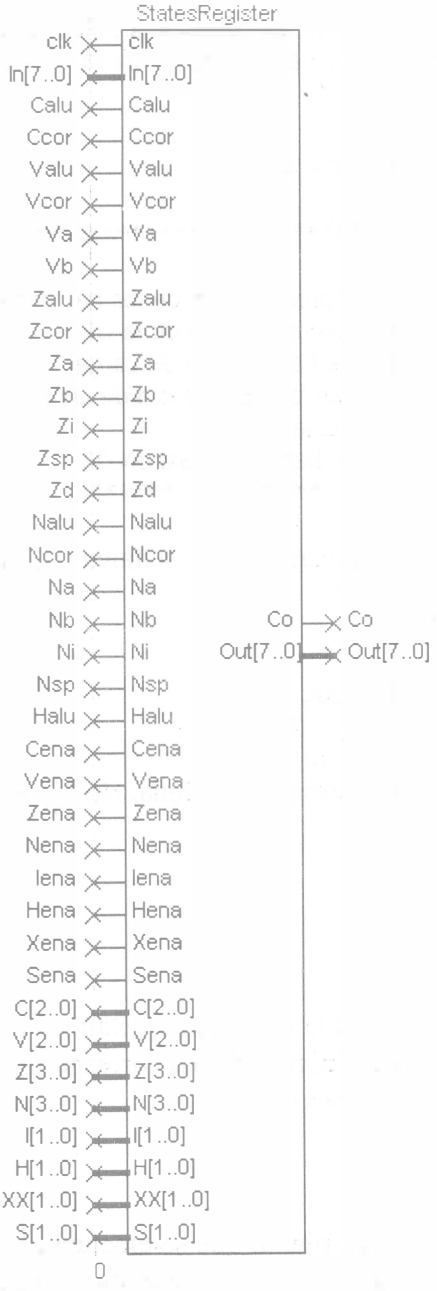
Salidas:

Co	Línea de salida con el valor de la bandera de acarreo
Out[7..0]	Bus de salida hacia la parte baja del bus de datos con las 8 banderas del registro

Entradas:

clk	Señal de reloj (disparo con flanco positivo)
In[7..0]	Bus de 8 bits conectado a la parte baja del bus de datos interno
Calu, ...	Calu y Ccor son las banderas de acarreo provenientes de la ALU (unidad lógica aritmética) y del registro de corrimiento de la UPA, respectivamente
Valu, ...	Valu, Vcor, Va y Vb son las banderas de overflow provenientes de la ALU, del registro de corrimiento de la UPA, del registro acumulador A y del registro acumulador B, respectivamente
Zalu, ...	Zalu, Zcor, Za, Zb, y Zsp son las banderas de cero provenientes de la ALU, del registro de corrimiento de la UPA, del registro acumulador A, del registro acumulador B y del apuntador de pila, respectivamente
Zi	Bandera de cero proveniente del registro índice IX ó IY
Zd	Bandera de cero proveniente del registro D El registro D se forma al concatenar el acumulador A (parte alta) con B (parte baja)
Nalu, ...	Nalu, Ncor, Na, Nb y Nsp son las banderas de negativo provenientes de la ALU, del registro de corrimiento de la UPA, del registro acumulador A, del registro acumulador B y del apuntador de pila, respectivamente
Ni	Bandera de negativo proveniente del registro índice IX ó IY
Halu	Bandera de medio acarreo proveniente de la ALU
Cena, ...	Las señales Cena, Vena, Zena, Nena, Iena, Hena, Xena y Sena habilitan la carga de datos en el registro de banderas adecuado (registros C, V, Z, N, I, H, X y S, respectivamente) La carga de datos se habilita con un nivel lógico bajo presente en estas señales
C[2..0], ...	C[2..0], V[2..0], Z[3..0], N[3..0], I[1..0], H[1..0], XX[1..0] y S[1..0] nos permiten seleccionar la procedencia de las banderas C, V, Z, N, I, H, X y S Los valores de selección son los siguientes:

0 h – La bandera proviene de la parte baja del bus de datos
1 h – El valor a almacenar es cero
2 h – El valor a almacenar es uno
3 h – La bandera proviene de la ALU
4 h – La bandera proviene del registro de corrimiento de la UPA
5 h – La bandera proviene del acumulador A
6 h – La bandera proviene del acumulador B
7 h – La bandera proviene del registro índice IX ó IY
8 h – La bandera proviene del doble acumulador (D=A:B)
9 h – La bandera proviene del apuntador de pila



Análisis del módulo StatesRegister.v

Las primeras líneas del código son comentarios.

```
/*************************************************************************
 * Module: Registro de Estados -> statesregister.v                       *
 *************************************************************************/
```

Enseguida, se declara un módulo llamado *statesregister* y el conjunto de señales que lo componen.

```
module statesregister (clk, In, Calu, Ccor, Valu, Vcor, Va, Vb, Zalu, Zcor, Za, Zb, Zi, Zsp, Zd,
      Nalu, Ncor, Na, Nb, Ni, Nsp, Halu, Cena, C, Vena, V, Zena, Z, Nena, N, Iena, I, Hena, H,
      Xena, XX, Sena, S, Co, Out);
```

Las siguientes líneas también son comentarios.

```
/*************************************************************************
 * Parámetros:                                                           *
 *          Calu..  - banderas procedentes de los registros internos,    *
 *                    de los acumuladores y de la UPA                    *
 *          Cena..  - señales de habilitación de carga                   *
 *          C, V..  - seleccionan la procedencia de las banderas         *
 *          Co      - acarreo de salida                                  *
 *          Out     - valores de banderas (SXHINZVC)                     *
 *************************************************************************/
```

A continuación, se define el tipo y tamaño de las señales declaradas en el módulo.

```
input         clk;
input  [7:0]  In;
input         Calu, Ccor, Valu, Vcor, Va, Vb;
input         Zalu, Zcor, Za, Zb, Zi, Zsp, Zd;
input         Nalu, Ncor, Na, Nb, Ni, Nsp;
input         Halu, Cena, Vena, Zena, Nena, Iena, Hena, Xena, Sena;
input  [2:0]  C, V;
input  [3:0]  Z, N;
input  [1:0]  I, H, XX, S;
output        Co;
output [7:0]  Out;
```

La siguiente línea declara una variable del tipo **reg** llamada *Out*, que almacenará los valores de las ocho banderas. El bit más significativo de Out corresponde a la bandera de stop (S) y el bit menos significativo a la bandera de acarreo (C).

```
reg    [7:0]  Out;
```

A continuación, se define un procedimiento que se ejecutará cada vez que se registre un flanco de subida en la señal de reloj. El procedimiento comienza revisando los valores de las señales de habilitación, de manera que si registra un cero en algunas de ellas, se debe actualizar el valor de la

bandera correspondiente. En caso de que alguna o algunas señales de habilitación no valgan '0', entonces no se ejecutará el bloque de sentencias asociado a cada **if**.

Por ejemplo, el siguiente bloque verifica si la señal *Cena* vale '0'; si es así, compara el valor de *C* con alguna de las constantes definidas. Si encuentra alguna correspondencia, el valor de la bandera seleccionada se carga en la posición adecuada de la variable *Out*, si no se encuentra correspondencia, el contenido del registro no es alterado.

```
always @ (posedge clk) begin

    // Actualiza la bandera de acarreo
    if (Cena==0) begin // Habilita al registro para carga
        case (C)
            3'h0:   Out[0] = In[0];     // C = bit 0 del bus de datos
            3'h1:   Out[0] = 0;         // C = 0
            3'h2:   Out[0] = 1;         // C = 1
            3'h3:   Out[0] = Calu;      // C = bandera de la alu
            3'h4:   Out[0] = Ccor;      // C = bandera del registro de corrimiento
        endcase
    end
```

Ahora revisamos el valor de la señal de habilitación *Vena*. Si éste vale '0', se actualiza el registro de la bandera overflow con el valor adecuado; en caso contrario, no se ejecuta el bloque de sentencias asociado al if y el valor de esta bandera no es modificado.

```
    // Actualiza la bandera de sobreflujo
    if (Vena==0) begin // Habilita el registro para carga
        case (V)
            3'h0:   Out[1] = In[1];     // V = bit 1 del bus de datos
            3'h1:   Out[1] = 0;         // V = 0
            3'h2:   Out[1] = 1;         // V = 1
            3'h3:   Out[1] = Valu;      // V = bandera de la alu
            3'h4:   Out[1] = Vcor;      // V = bandera del registro de corrimiento
            3'h5:   Out[1] = Va;        // V = bandera del acumulador A
            3'h6:   Out[1] = Vb;        // V = bandera del acumulador B
        endcase
    end
```

De manera similar revisamos el resto de las señales de habilitación y actualizamos el valor de la bandera correspondiente.

```
    // Actualiza la bandera de cero
    if (Zena==0) begin // Habilita el registro para carga
        case (Z)
            4'h0:   Out[2] = In[2];     // Z = bit 2 del bus de datos
            4'h1:   Out[2] = 0;         // Z = 0
            4'h2:   Out[2] = 1;         // Z = 1
            4'h3:   Out[2] = Zalu;      // Z = bandera de la alu
            4'h4:   Out[2] = Zcor;      // Z = bandera del registro de corrimiento
            4'h5:   Out[2] = Za;        // Z = bandera del acumulador A
            4'h6:   Out[2] = Zb;        // Z = bandera del acumulador B
```

```
                4'h7:   Out[2] = Zi;        // Z = bandera del registro índice
                4'h8:   Out[2] = Zd;        // Z = bandera del doble acumulador
                4'h9:   Out[2] = Zsp;       // Z = bandera del apuntador de pila
        endcase
end

// Actualiza la bandera de negativo
if (Nena==0) begin // Habilita el registro para carga
        case (N)
                4'h0:   Out[3] = In[3];     // N = bit 3 del bus de datos
                4'h1:   Out[3] = 0;         // N = 0
                4'h2:   Out[3] = 1;         // N = 1
                4'h3:   Out[3] = Nalu;      // N = bandera de la alu
                4'h4:   Out[3] = Ncor;      // N = bandera del registro de corrimiento
                4'h5:   Out[3] = Na;        // N = bandera del acumulador A
                4'h6:   Out[3] = Nb;        // N = bandera del acumulador B
                4'h7:   Out[3] = Ni;        // N = bandera del registro índice
                4'h9:   Out[3] = Nsp;       // N = bandera del apuntador de pila
        endcase
end

// Actualiza la bandera de interrupción IRQ
if (Iena==0) begin // Habilita el registro para carga
        case (I)
                2'h0:   Out[4] = In[4];     // I = bit 4 del bus de datos
                2'h1:   Out[4] = 0;         // I = 0
                2'h2:   Out[4] = 1;         // I = 1
        endcase
end

// Actualiza la bandera de medio acarreo
if (Hena==0) begin // Habilita el registro para carga
        case (H)
                2'h0:   Out[5] = In[5];     // H = bit 5 del bus de datos
                2'h1:   Out[5] = 0;         // H = 0
                2'h2:   Out[5] = 1;         // H = 1
                2'h3:   Out[5] = Halu;      // H = bandera de la alu
        endcase
end

// Actualiza la bandera de interrupción XIRQ
if (Xena==0) begin // Habilita el registro para carga
        case (XX)
                2'h0:   Out[6] = In[6];     // X = bit 6 del bus de datos
                2'h1:   Out[6] = 0;         // X = 0
                2'h2:   Out[6] = 1;         // X = 1
        endcase
end

// Actualiza la bandera de stop
if (Sena==0) begin // Habilita el registro para carga
        case (S)
                2'h0:   Out[7] = In[7];     // S = bit 7 del bus de datos
```

```
                2'h1:   Out[7] = 0;          // S = 0
                2'h2:   Out[7] = 1;          // S = 1
            endcase
        end
```

Una vez revisadas todas las señales de habilitación y actualizados los valores de las banderas, concluimos la definición del procedimiento.

```
    end
```

Por último, asignamos a *Co* el valor de *Out*[0], es decir, el valor de la bandera de acarreo. Y terminamos la definición del módulo.

```
    assign Co = Out[0];

endmodule
```

B.2.4 UNIDAD DE PROCESOS ARITMÉTICOS

Esta unidad se encarga de ejecutar las operaciones lógico aritméticas básicas, para ello, cuenta con una unidad lógico aritmética (ALU) que le permite ejecutar operaciones tales como suma, resta, and, or, or-exclusivo, entre otras.

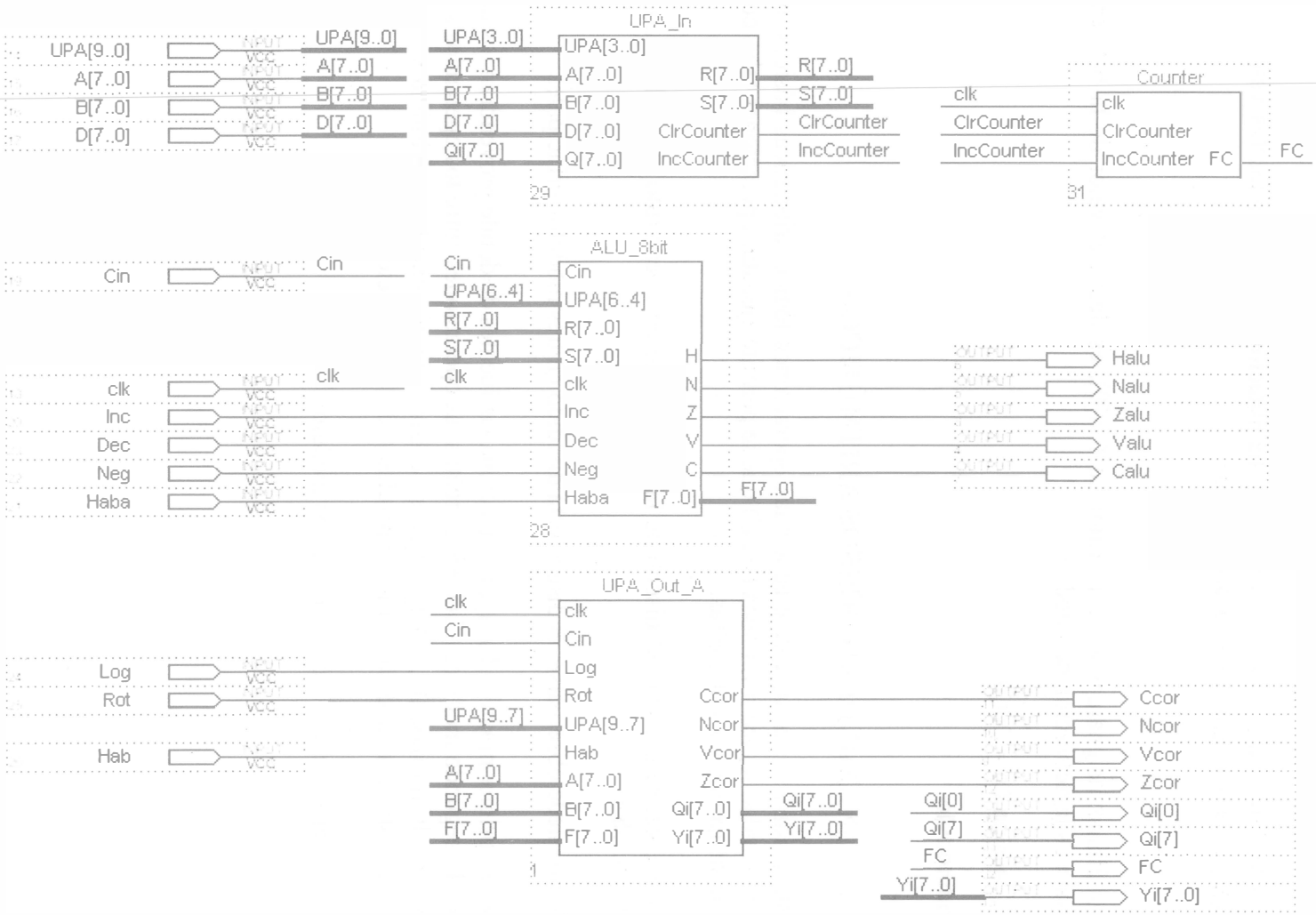
Enseguida, se presenta el diseño de la UPA (visto en el capítulo 5) implantado con el software MAX+PLUS II.

Como puede observarse, el diseño está dividido en tres módulos principales.

- El primer módulo, *UPA_in*, selecciona las fuentes de donde provendrán los datos a operar.
- El segundo módulo, *ALU_8bit*, efectúa las operaciones entre los operandos seleccionados.
- El último módulo, *UPA_out_A*, se encarga de efectuar los corrimientos, y de colocar en el registro adecuado (destino) el resultado de las operaciones.

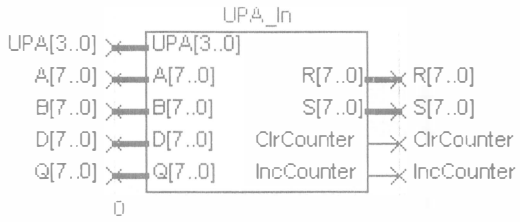
Ahora desglosemos cada uno de estos módulos.

MÓDULO UNIDAD DE PROCESOS ARITMÉTICOS



Cómo funciona el módulo UPA_In

El módulo *upa_in* selecciona la procedencia de los datos a operar. El siguiente diagrama muestra las entradas y las salidas que componen a este módulo; además, se explica brevemente la función de cada una de ellas.



Entradas:

A[7..0]	Dato proveniente del acumulador A
B[7..0]	Dato proveniente del acumulador B
D[7..0]	Dato proveniente del bus de datos
Q[7..0]	Dato proveniente del registro de corrimiento Q de la UPA
UPA[3..0]	Señales de control. Permiten elegir la procedencia de los datos

Salidas:

R[7..0]	Primer operando
S[7..0]	Segundo operando
ClrCounter	Limpia contador (coloca a ceros el contenido del registro contador)
IncCounter	Incrementa al registro contador en una unidad

Análisis del módulo UPA_In.v

Las primeras líneas del código son comentarios.

```
/*****************************************************************
 * Module: Selecciona las fuentes de la ALU -> upa_in.v          *
 *****************************************************************/
```

Enseguida, se declara un módulo llamado *upa_in* y el conjunto de señales que lo componen.

```
module upa_in (UPA, A, B, D, Q, R, S, ClrCounter, IncCounter);
```

Las siguientes líneas también son comentarios.

```
/*****************************************************************
 * Parámetros:                                                   *
 *      UPA         - seleccionan la procedencia de los datos    *
 *      A, B, D     - buses de entrada de datos                  *
 *      Q           - bus de entrada de datos del registro de corrimiento *
 *      R, S        - salidas de datos (operandos)               *
```

```
*      ClrCounter     - inicializa el contador                    *
*      IncCounter     - incrementa la cuenta                      *
*****************************************************************/
```

También se define el tipo y tamaño de las señales declaradas previamente.

```
input  [3:0]  UPA;
input  [7:0]  A, B, D, Q;
output [7:0]  R, S;
reg    [7:0]  R, S;
output        ClrCounter, IncCounter;
reg           ClrCounter, IncCounter;
```

A continuación se define una construcción always cuya función es seleccionar la procedencia de los operandos, al igual que un multiplexor. Note que esta construcción es sensible a la señal *UPA*, esto es, cada vez que UPA cambia de valor, el procedimiento se ejecuta.

Por ejemplo, imaginemos que el valor de UPA[3..0] cambia a 4'b0101. Una vez que el procedimiento comienza a ejecutarse, la sentencia **case** compara el valor de *UPA* con los valores de sus constantes. Como el valor de *UPA* es igual a 4'h5, el bloque de sentencias asociado a esta constante es ejecutado; es decir, se seleccionan los datos del bus D y del acumulador A, y se guardan en las variables R y S, respectivamente.

```
always @ (UPA) begin
        case (UPA)
                // Selecciona el bus A[] y cero
                4'h0:   begin   R = A; S = 0; ClrCounter = 0; IncCounter = 0; end
                // Selecciona el bus A[] y el bus B[]
                4'h1:   begin   R = A; S = B; ClrCounter = 0; IncCounter = 0; end
                // Selecciona cero y el bus Q[]
                4'h2:   begin   R = 0; S = Q; ClrCounter = 0; IncCounter = 0; end
                // Selecciona cero y el bus B[]
                4'h3:   begin   R = 0; S = B; ClrCounter = 0; IncCounter = 0; end
                // Selecciona cero y el bus A[]
                4'h4:   begin   R = 0; S = A; ClrCounter = 0; IncCounter = 0; end
                // Selecciona el bus D[] y el bus A[]
                4'h5:   begin   R = D; S = A; ClrCounter = 0; IncCounter = 0; end
                // Selecciona el bus D[] y el bus Q[]
                4'h6:   begin   R = D; S = Q; ClrCounter = 0; IncCounter = 0; end
                // Selecciona el bus D[] y cero
                4'h7:   begin   R = D; S = 0; ClrCounter = 0; IncCounter = 0; end
                // Selecciona el bus D[] y el bus B[]
                4'h8:   begin   R = D; S = B; ClrCounter = 0; IncCounter = 0; end
                // Selecciona el bus Q[] y el bus B[]
                4'h9:   begin   R = Q; S = B; ClrCounter = 0; IncCounter = 0; end
                // Limpia el contenido del registro contador
                4'hA:   begin   R = B; S = 0; ClrCounter = 1; IncCounter = 0;  end
                // Incrementa el contenido del registro contador
                4'hB:   begin   R = B; S = 0; ClrCounter = 0; IncCounter = 1; end
                // Selecciona el bus Q[] y el bus A[]
                4'hC:   begin   R = Q; S = A; ClrCounter = 0; IncCounter = 0; end
```

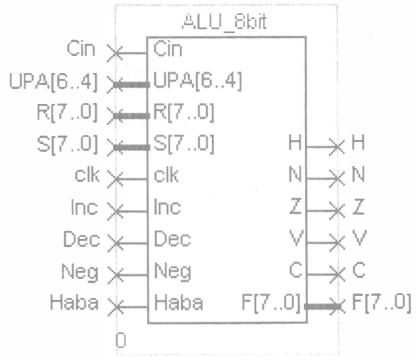
```
        endcase
    end
```

Por último, se concluye la definición del módulo.

```
    endmodule
```

Cómo funciona el módulo ALU_8bit

Este módulo realiza las operaciones lógico aritméticas entre los operandos seleccionados. El siguiente diagrama muestra las entradas y las salidas que componen a este módulo, y se explica brevemente la función de cada una de ellas.



Entradas:

clk	Señal de reloj (disparo con flanco positivo)
Cin	Acarreo de entrada
Inc	Indica que la operación a efectuar es un incremento
Dec	Indica que la operación a efectuar es un decremento
Neg	Indica que la operación a efectuar es una negación
Haba	Si Haba=1, se calculan las banderas de la ALU
R, S	Datos a operar
UPA[6..4]	Señales de control para seleccionar la operación a efectuar

Salidas:

H, N, ...	H, N, Z, V y C son los valores de las banderas de medio acarreo, negativo, cero, overflow y acarreo, generadas tras cualquier operación de la ALU
F[7..0]	Resultado de la operación

Análisis del módulo ALU_8bit.v

Las primeras líneas del código son comentarios.

```
/*****************************************************************
 * Module: 8-bit Logical/Arithmetic Unit -> alu_8bit.v           *
 *****************************************************************/
```

Enseguida, se declara un módulo llamado *alu_8bit* y el conjunto de señales que lo componen.

```
module alu_8bit (clk, Cin, UPA, R, S, Inc, Dec, Neg, Haba, H, N, V, Z, C, F);
```

Las siguientes líneas también son comentarios.

```
/*****************************************************************
 * Parámetros:                                                   *
 *        clk     - señal de reloj                               *
 *        Cin     - acarreo de entrada                           *
 *        Inc     - las banderas a calcular son para un incremento *
 *        Dec     - las banderas a calcular son para un decremento *
 *        Neg     - las banderas a calcular son para una negación *
 *        UPA     - señales de control (seleccionan la operación de la ALU) *
 *        R,S     - operandos                                    *
 *        Haba    - guarda las banderas de la operación de la ALU *
 *        F       - salida de resultados                         *
 *        H,N,V.. - banderas de medio acarreo, negativo, sobreflujo, *
 *                  cero y acarreo                               *
 *****************************************************************/
```

A continuación, se define el tipo y tamaño de las señales declaradas.

```
input          clk, Cin, Inc, Dec, Neg, Haba;
input  [6:4]   UPA;
input  [7:0]   R, S;
output         C, N, Z, V, H;
output [7:0]   F;
```

Además, se hace uso de varias variables del tipo **reg** para guardar los valores de las banderas, así como para almacenar los datos y los resultados generados por la función parametrizada lpm_add_sub.

```
reg            Ct, Vt, Ht, C, V, H, Z, N, add_sub;
reg    [7:0]   F, Fa, R1, S1;
```

Para efectuar el cálculo de las sumas y restas utilizaremos la función parametrizada lpm_add_sub, para ello crearemos una instancia de este módulo, llamada *addsub1*, y asignaremos cada una de nuestras variables al puerto correspondiente en el módulo.

```
lpm_add_sub addsub1 (.result (Fa), .cin (Cin), .dataa (R1), .datab (S1), .add_sub (add_sub));
```

También se requiere definir el número de bits necesarios para representar los valores de los operandos y del resultado, es por ello que colocamos el parámetro **lpm_width** a ocho.

```
defparam addsub1.lpm_width = 8;
```

Una vez declaradas todas nuestras variables crearemos un procedimiento para manipularlas. En resumen, este procedimiento revisa el valor de las señales de control *UPA*[6..4] y ejecuta la operación correspondiente.

Para implantar este procedimiento utilizaremos una construcción **always** sensible a los cambios en las señales *R* y *S* (los operandos), de manera que cuando un operando cambia de valor, el procedimiento se ejecuta. Dentro de la construcción always se hace uso de una sentencia **case** para decodificar el valor de las señales de control y ejecutar la operación lógico/aritmética adecuada.

Si revisa el código de alguna de las etiquetas de la sentencia **case**, observará que primero se efectúa la operación lógico/aritmética y a continuación se calculan las banderas. El cálculo de banderas debe hacerse cuidadosamente porque hay banderas que se calculan de manera diferente, dependiendo de la operación efectuada. Por ejemplo, la bandera de acarreo se calcula de manera diferente para una sustracción, un decremento y una negación, aún cuando las tres operaciones son simples restas.[17]

Revisemos el bloque de sentencias para la etiqueta 3'h0, el cual ejecutará la operación de suma si la señal de control UPA[6..4] vale 3'b000.

Dentro del bloque se establecen los valores correctos para las variables *R1*, *S1* y *add_sub*, de tal manera que el módulo *addsub1* efectúa una suma. Enseguida, se calculan las banderas de medio acarreo y acarreo, según el manual del microcontrolador MC68HC11. Por último, la bandera de overflow se calcula en base a la operación efectuada: una suma ó un incremento. Una metodología similar se sigue para implantar las otras operaciones de la ALU.

```
always @ (R or S) begin
        case (UPA)
                3'h0:   begin // Suma R+S+Cin
                        R1 = R; S1 = S; add_sub = 1; F = Fa;
                        Ht = R[3] & S[3] | S[3] & (~F[3]) | (~F[3]) & R[3];
                        Ct = R[7] & S[7] | S[7] & (~F[7]) | (~F[7]) & R[7];
                        if (Inc) // Banderas para un incremento
                                Vt = (~R[7]) & (& R[6:0]);
                        else
                                Vt = R[7] & S[7] & (~F[7]) | (~R[7]) & (~S[7]) & F[7];
                        end
                3'h1:   begin // Resta S-R-(~Cin)
                        R1 = S; S1 = R; add_sub = 0; F = Fa;
                        Ht = 0;
                        if (Dec==Neg) begin // Banderas para una resta
                                Vt = S[7] & (~R[7]) & (~F[7]) | (~S[7]) & R[7] & F[7];
                                Ct = (~S[7]) & R[7] | R[7] & F[7] | F[7] & (~S[7]);
                        end
                        else if (Dec & ~Neg) begin // Banderas para un decremento
                                Vt = (~F[7]) & (& F[6:0]);
                                Ct = 0;
                        end
                        else if (~Dec & Neg) begin // Banderas para una negación
                                Vt = F[7] & (~F[6]) & (~F[5]) & (~F[4]) & (~F[3]) & (~F[2]) &
                                (~F[1]) & (~F[0]);
                                Ct = (| F);
                        end
```

[17] Si desea mayor información acerca del cálculo de banderas consulte el manual del microcontrolador MC68HC11.

```
                end
        3'h2:   begin // Resta R-S-(~Cin)
                R1 = R; S1 = S; add_sub = 0; F = Fa;
                Ht = 0;
                if (Dec==Neg) begin // Banderas para una resta
                        Vt = R[7] & (~S[7]) & (~F[7]) | (~R[7]) & S[7] & F[7];
                        Ct = (~R[7]) & S[7] | S[7] & F[7] | F[7] & (~R[7]);
                end
                else if (Dec & ~Neg) begin // Banderas para un decremento
                        Vt = (~F[7]) & (& F[6:0]);
                        Ct = 0;
                end
                else if (~Dec & Neg) begin // Banderas para una negación
                        Vt = F[7] & (~F[6]) & (~F[5]) & (~F[4]) & (~F[3]) & (~F[2]) &
                        (~F[1]) & (~F[0]);
                        Ct = (| F);
                end
                end
        3'h3:   begin // Ejecuta R or S
                F = R | S;
                Ht = 0; Ct = 0; Vt = 0;
                end
        3'h4:   begin // Ejecuta R and S
                F = R & S;
                Ht = 0; Ct = 0; Vt = 0;
                end
        3'h5:   begin // Ejecuta ~R and S
                F = ~R & S;
                Ht = 0; Ct = 0; Vt = 0;
                end
        3'h6:   begin // Ejecuta R xor S
                F = R ^ S;
                Ht = 0; Ct = 0; Vt = 0;
                end
        3'h7:   begin // Ejecuta R xnor S
                F = R ~^ S;
                Ht = 0; Ct = 0; Vt = 0;
                end
    endcase
end
```

También se utiliza una construcción **always**, sensible a los flancos de subida del reloj, para guardar el valor de las banderas calculadas. Note que las banderas sólo serán modificadas si la señal *Haba* presenta un nivel lógico alto.

```
always @ (posedge clk) begin
    if (Haba) begin
            // Asigna la bandera de acarreo
            C = Ct;
            // Calcula la bandera de negativo
            N = F[7];
            // Calcula la bandera de cero
            Z = (~(| F));
```

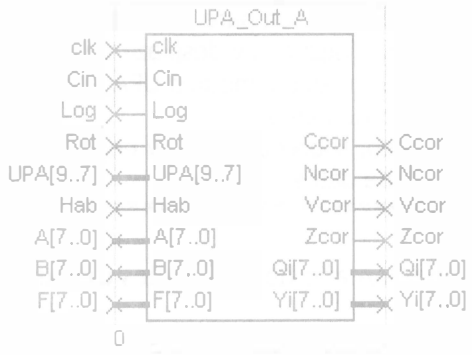
```
                // Asigna la bandera de sobreflujo
                V = Vt;
                // Asigna la bandera de medio acarreo
                H = Ht;
            end
        end
```

Por último, concluimos la definición del módulo.

```
    endmodule
```

Cómo funciona el módulo UPA_out_A

El módulo *upa_out_a* realiza el cálculo de los corrimientos y asigna el valor de éstos a los distintos destinos. El siguiente diagrama muestra las entradas y salidas que componen a este módulo y se explica brevemente la función de cada una de ellas.



Entradas:

clk	Señal de reloj (disparo con flanco positivo)
Cin	Acarreo de entrada
Log	La operación a efectuar es un desplazamiento lógico
Rot	La operación a efectuar es una rotación Si Log=0 y Rot=0, la operación a efectuar es un desplazamiento aritmético Si Log=1 y Rot=1, el corrimiento efectuado es para el cálculo de una multiplicación
Hab	Habilita las operaciones del módulo UPA_Out_A
A, B	Datos provenientes de los acumuladores A y B
F	Resultado de la operación realizada por el módulo ALU_8bit
UPA[9..7]	Señales de control para elegir el tipo de desplazamiento y el destino del resultado

Salidas:

Ccor, ...	Ncor, Zcor, Vcor y Ccor son los valores de las banderas de negativo, cero, overflow y acarreo, generadas tras un corrimiento
Qi[7..0]	Salida con el contenido del registro Q
Yi[7..0]	Salida con el contenido del registro Y

Análisis del módulo UPA_Out_A.v

Las primeras líneas del código son comentarios.

```
/*****************************************************************
 * Module: Salidas y corrimientos de la UPA -> upa_out_a.v       *
 *****************************************************************/
```

Enseguida, se declara un módulo llamado *upa_out_a* y el conjunto de señales que lo componen.

```
module upa_out_a (clk, Cin, Log, Rot, UPA, Hab, A, B, F, Ccor, Ncor, Vcor, Zcor, Qi, Yi);
```

Las siguientes líneas también son comentarios.

```
/*****************************************************************
 * Parámetros:                                                   *
 *      clk     - señal de reloj                                 *
 *      Cin     - acarreo de entrada                             *
 *      Log     - las banderas a calcular son para un corrimiento lógico *
 *      Rot     - las banderas a calcular son para una rotación  *
 *      UPA     - señales de control (destinos y desplazamientos) *
 *      Hab     - habilita las operaciones del módulo UPA_Out_A  *
 *      A,B,F   - buses de entrada de datos                      *
 *      Ccor..  - banderas (acarreo, sobreflujo, cero y negativo) *
 *      Qi,Yi   - buses de salida de datos                       *
 *****************************************************************/
```

A continuación, se define el tipo y tamaño de las señales declaradas previamente.

```
input          clk, Cin, Log, Rot, Hab;
input  [9:7]   UPA;
input  [7:0]   A, B, F;
output         Ccor, Ncor, Zcor, Vcor;
output [7:0]   Qi, Yi;
```

Además, son declaradas algunas variables del tipo **reg** para guardar los valores de las banderas y el resultado de los desplazamientos.

```
reg            Ct, Cm, Qm, Ncor, Zcor, Vcor, Ccor, Mul;
reg    [7:0]   Ql, Yi;
```

Para el cálculo de los desplazamientos y rotaciones, así como para la actualización de los destinos, emplearemos una construcción **always** que se ejecutará al momento de registrar un flanco de subida en el reloj.

Dentro de la construcción se revisa el valor de la señal *Hab* para habilitar la ejecución de la sentencia **case**. Si *Hab*=1, la sentencia case comparará el valor de las señales de control *UPA*[9..7] contra el valor de sus etiquetas, y ejecutará el bloque de sentencias correspondiente.

Por ejemplo, supongamos que *Hab*=1 y *UPA*[9..7]=3'b000. Una vez que el procedimiento se ejecuta se revisa el valor de la señal *Hab* y la sentencia case busca una correspondencia; en este caso, el bloque que se ejecuta es el asociado con la etiqueta 4'h0, el cual asigna el resultado del módulo *alu_8bit* a los destinos *Q*i y *Y*i.

Ahora suponga que *Hab*=1 y *UPA*[9..7]=3'b100; el bloque de sentencias que se ejecuta es el asociado con la etiqueta 4'h4, que calcula un desplazamiento, o bien, una rotación hacia la derecha, del resultado generado por el módulo *alu_8bit*.

Note que un desplazamiento lógico se calcula de manera diferente a un desplazamiento aritmético, o a una rotación; por lo tanto, se recurre a dos señales de control, *Log* y *Rot*, que nos indicarán el tipo de corrimiento para realizar el cálculo correcto.

```
always @ (posedge clk) begin
        Ct = Yi[0]; Cm = Yi[7]; Qm = Qi[7]; Mul = Yi[0];
        if (Hab) begin
                case (UPA)
                        4'h0:   begin // Carga F en Yi y en Qi
                                Ct = 0; Yi = F; Qi = F;
                                end
                        4'h1:   begin // Carga F en Yi, Qi no se modifica
                                Ct = 0; Yi = F;
                                end
                        4'h2:   begin // Carga A en Yi, Qi no se modifica
                                Ct = 0; Yi = A;
                                end
                        4'h3:   begin // Carga B en Yi, Qi no se modifica
                                Ct = 0; Yi = B;
                                end
                        4'h4:   begin // Calcula F/2 y lo guarda en Yi, Qi no se modifica
                                // Desplazamiento lógico hacia la derecha
                                if (Log & ~Rot)
                                        Yi = (F >> 1);
                                // Desplazamiento aritmético hacia la derecha
                                else if (~Log & ~Rot) begin
                                        Yi = (F >> 1); Yi[7] = Cm;
                                end
                                // Rotación hacia la derecha
                                else if (~Log & Rot) begin
                                        Yi = (F >> 1); Yi[7] = Cin;
                                end
                                end
                        4'h5:   begin // Calcula Qi/2 y lo guarda en Qi, Yi no se modifica
                                // Desplazamiento lógico hacia la derecha
                                if (Log & ~Rot)
                                        Qi = (Qi >> 1);
                                // Desplazamiento aritmético hacia la derecha
                                else if (~Log & ~Rot) begin
                                        Qi = (Qi >> 1); Qi[7] = Qm;
                                end
                                // Rotación hacia la derecha
                                else if (~Log & Rot) begin
```

```
                    Qi = (Qi >> 1); Qi[7] = Cin;
                end
                // Desplazamiento hacia la derecha para multiplicaciones
                else if (Log & Rot) begin
                    Qi = (Qi >> 1); Qi[7] = Mul;
                    Yi = (Yi >> 1); Yi[7] = Cin;
                end
                end
        4'h6:   begin // Calcula 2F y lo guarda en Yi, Qi no se modifica
                // Desplazamiento lógico y aritmético hacia la izquierda
                if (~Rot)
                    Yi = (F << 1);
                // Rotación hacia la izquierda
                else begin
                    Yi = (F << 1); Yi[0] = Cin;
                end
                end
        4'h7:   begin // Calcula 2Qi y lo guarda en Qi, Yi no se modifica
                // Desplazamiento lógico y aritmético hacia la izquierda
                if (~Rot)
                    Qi = (Qi << 1);
                // Rotación hacia la izquierda
                else begin
                    Qi = (Qi << 1); Qi[0] = Cin;
                end
                end
    endcase
```

Antes de terminar la definición de la construcción **always**, debemos calcular las nuevas banderas que se generan tras la realización de los corrimientos.[18]

```
// Calcula la bandera de acarreo
Ccor = (UPA[9] & UPA[8]) ? Cm : ( (Log & Rot) ? Qi[7] : Ct );
// Calcula la bandera de negativo
Ncor = Yi[7];
// Calcula la bandera de sobreflujo
Vcor = (UPA[9] & UPA[8]) ? (Yi[7] ^ Cm) : (Yi[7] ^ Ct);
// Calcula la bandera de cero
Zcor = !(| Yi);
```

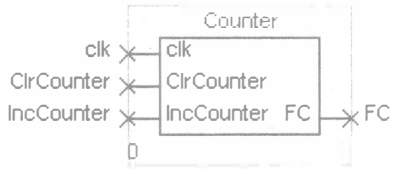
Por último, colocamos etiquetas de finalización para el bloque de sentencias asociado a la condicional if, para la construcción always y para el módulo *UPA_Out_A*.

```
            end
        end

    endmodule
```

[18] Para cualquier duda acerca del cálculo de las banderas es preciso consultar el manual de referencia del microcontrolador MC68HC11.

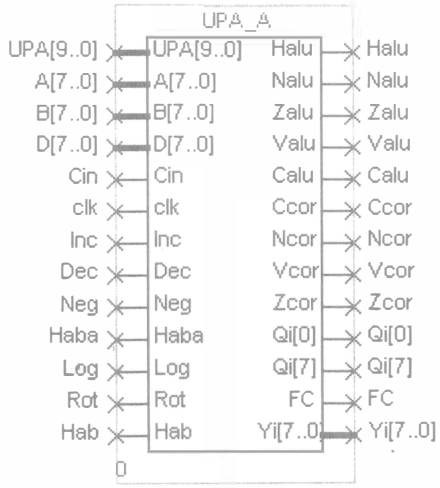
Cómo funciona el módulo Counter.v

Este módulo, como su nombre lo indica, es un contador de 3 bits. Las señales de entrada *ClrCounter* e *IncCounter* permiten inicializar e incrementar el contador, respectivamente. Si el contador alcanza su cuenta máxima, es decir, llega a 7, la señal de salida FC es activada. Debido a la simplicidad de este módulo, no entraremos en mayores detalles sobre su construcción.



Cómo funciona el módulo UPA_A

Una vez construidos los cuatro módulos anteriores, los uniremos para generar una sola entidad: *UPA_A*. El diagrama resultante tras la compilación del diseño, el cual fue visto al principio de esta sección, es el siguiente.



Entradas:

UPA[9..0]	Señales de control de la UPA
A[7..0]	Dato procedente del acumulador A
B[7..0]	Dato procedente del acumulador B
D[7..0]	Dato procedente del bus de datos interno
clk	Señal de reloj (disparo con flanco positivo)
Cin	Acarreo de entrada
Inc	Las banderas a calcular son para un incremento
Dec	Las banderas a calcular son para un decremento
Neg	Las banderas a calcular son para una negación
Haba	Si Haba=1, las banderas del módulo alu_8bit son calculadas

Log	La operación a efectuar es un desplazamiento lógico
Rot	La operación a efectuar es una rotación
Hab	Habilita las operaciones en el módulo upa_out_a

***S*alidas:**

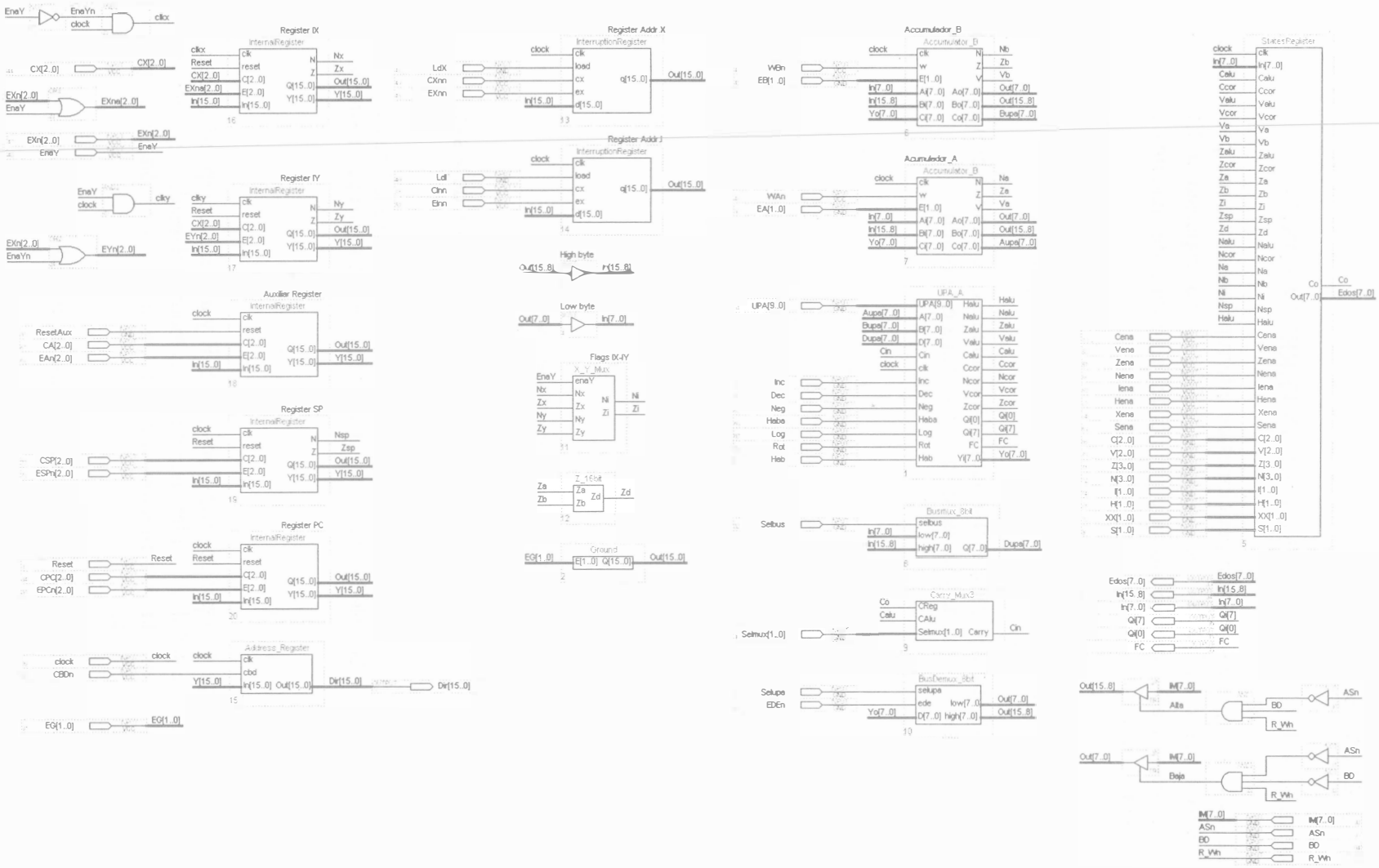
Halu, ...	Halu, Nalu, Zalu, Valu y Calu son los valores de las banderas de medio acarreo, negativo, cero, sobreflujo y acarreo, generadas por la ALU
Ccor, ...	Ncor, Zcor, Vcor y Ccor son los valores de las banderas de negativo, cero, sobreflujo y acarreo, generadas tras un corrimiento
Qi[0]	Bit menos significativo del registro de corrimiento Qi
Qi[7]	Bit más significativo del registro de corrimiento Qi
FC	Si FC=1, el contador ha alcanzado su cuenta máxima
Y[7..0]	Bus de salida con el resultado de la UPA

B.2.5 MÓDULO ARQUITECTURA_WIRE_VERILOG

Este módulo, al igual que el módulo *UPA_A*, fue construido utilizando el editor gráfico de MAX+PLUS II. Como puede verse, este módulo está compuesto por todos los módulos que construimos a lo largo de la etapa 1, a excepción de algunos que no fueron explicados debido a su simplicidad.

A continuación se reproduce un diagrama de este módulo donde se muestran las conexiones entre los diversos submódulos.

MÓDULO ARQUITECTURA_WIRE_VERILOG



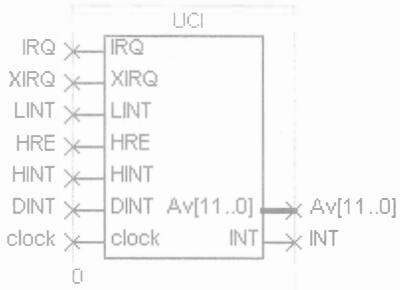
B.3 ETAPA 2: CONTROL

B.3.1 UNIDAD DE CONTROL DE INTERRUPCIONES

La UCI se encarga de recibir peticiones de interrupciones externas procedentes de alguno de los dispositivos conectados a las líneas $\overline{IRQ}$ ó $\overline{XIRQ}$. Como respuesta, la UCI envía una dirección de salto al secuenciador que le indica el inicio del algoritmo de máquina de estados que atiende la interrupción solicitada.

Cómo funciona el módulo Unidad de control de interrupciones

El siguiente diagrama muestra el módulo *UCI*; además, se explica brevemente la función de cada una de las señales de entrada y de salida que integran a este módulo.



Entradas:

clock	Señal de reloj (disparo con flanco positivo)
IRQ	Petición de interrupción externa IRQ
XIRQ	Petición de interrupción externa XIRQ
LINT, HRE, ...	LINT, HINT, HRE y DINT habilitan la generación de la señal INT

Salidas:

Av[11..0]	Dirección de inicio de la subrutina que atiende a la interrupción
INT	Cuando INT=1, existe una petición de una interrupción externa

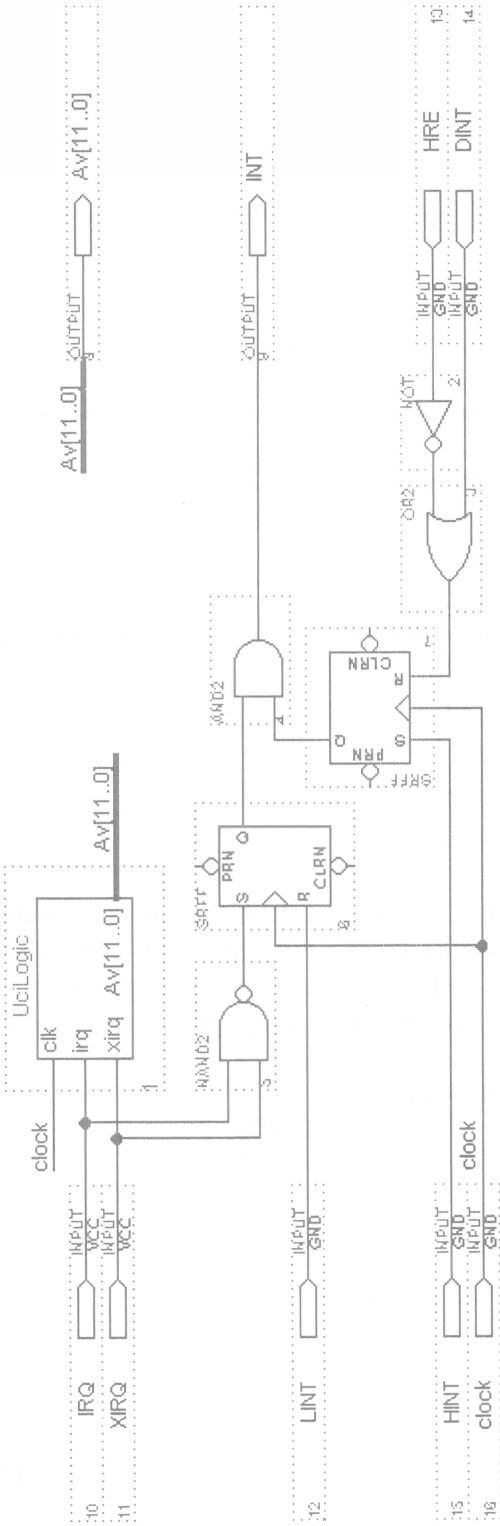
Análisis del módulo UCI.gdf

La UCI está compuesta por dos flip-flops tipo S-R y algunas compuertas lógicas que se encargan de la generación de la señal *INT*; además, integra un módulo llamado *UciLogic* de donde se extrae la dirección de salto para la interrupción adecuada. Este último módulo es muy simple, pero lo analizaremos más adelante.

El siguiente diagrama muestra cómo está construida la unidad de control de interrupciones.

UNIDAD DE CONTROL DE INTERRUPCIONES

UCI



Análisis del módulo UciLogic.v

Las primeras líneas del código son comentarios.

```
/*****************************************************************
 * Module: Lógica UCI (Unidad de Interrupciones) -> ucilogic.v   *
 *****************************************************************/
```

Enseguida, se declara un módulo llamado *ucilogic* y el conjunto de señales que lo componen.

```
module ucilogic (clk, irq, xirq, Av);
```

Las siguientes líneas también son comentarios.

```
/*****************************************************************
 * Parámetros:                                                   *
 *      clk     - señal de reloj                                 *
 *      irq     - señal de interrupción IRQ                      *
 *      xirq    - señal de interrupción XIRQ                     *
 *      Av      - dirección de salto hacia la subrutina de atención *
 *                a la interrupción                              *
 *****************************************************************/
```

A continuación, se define el tipo y tamaño de las señales declaradas previamente.

```
input          clk, irq, xirq;
output [11:0]  Av;
reg    [11:0]  Av;
```

Utilizaremos una construcción **always** sensible a los flancos de subida del reloj para seleccionar la dirección de salto correcta. El valor de la dirección dependerá de las señales *irq* y *xirq*. Cabe mencionar, que las direcciones de salto establecidas en este módulo son totalmente arbitrarias, sólo cuidamos que su valor fuera diferente al de los códigos de operación de las instrucciones.

```
always @ (posedge clk) begin
        if (!irq & xirq)        Av = 12'hF90;
        else if (irq & !xirq)   Av = 12'hFC0;
        else                    Av = Av;
end
```

Por último, concluimos la definición del módulo.

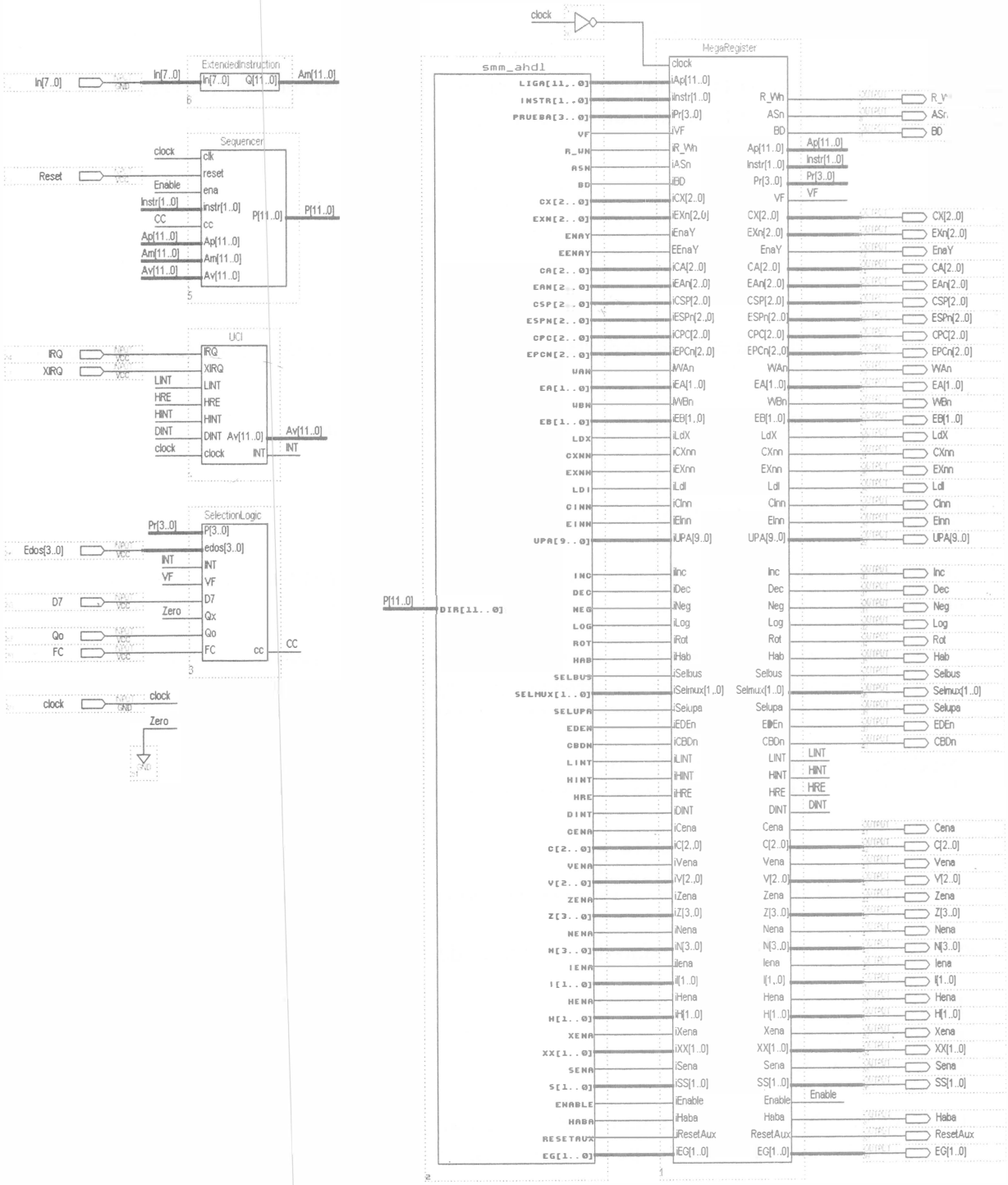
```
endmodule
```

B.3.2 UNIDAD DE CONTROL DE LA COMPUTADORA

La *UCC* es la parte fundamental de la computadora. Su tarea consiste en decodificar las instrucciones en lenguaje ensamblador y ejecutar las micro-operaciones necesarias para llevarlas a

cabo. Los módulos que integran a la unidad de control de la computadora son: el registro de instrucción, el secuenciador, la memoria de microprograma y la lógica de selección.

A continuación se presenta el diagrama del módulo *control_wire_verilog*, que muestra la conexión entre la unidad de control de la computadora y la unidad de control de interrupciones.



B.3.2.1 REGISTRO DE INSTRUCCIÓN

En este registro se almacena el código de operación de la próxima instrucción a ejecutar, el cual representa una dirección de salto que le indica al secuenciador en dónde comienzan las micro-operaciones para dicha instrucción.

Cómo funciona el Registro de Instrucción

A continuación se presenta el diagrama de entradas y salidas del módulo *ExtendedInstruction*, y se explica brevemente la función de cada una de ellas.

Entradas:

In[7..0]	Bus de entrada de 8 bits (conectado a la parte baja del bus de datos interno)

Salidas:

Q[11..0]	Bus de salida de 12 bits (conectado al secuenciador)

Análisis del módulo ExtendedInstruction.v

El programa comienza con la declaración del módulo *ExtendedInstruction* junto con las señales de entrada y salida que lo componen.

```
/*****************************************************************
 * Module: Extended Instruction  -> extendedinstruction.v        *
 *****************************************************************/

module extendedinstruction (In, Q);
/*****************************************************************
 * Parámetros:                                                   *
 *       In      - bus de entrada (8-bit)                        *
 *       Q       - bus de salida (12-bit)                        *
 *****************************************************************/
```

Enseguida, se define el tipo y tamaño de las señales declaradas previamente.

```
input  [ 7:0]  In;
output [11:0]  Q;
reg    [11:0]  Q;
```

El cuerpo del módulo consiste de una construcción **always** sensible a los cambios en la señal *In*. Este procedimiento desplaza cuatro posiciones hacia la izquierda al código de operación de la

instrucción, de manera que el código de operación se extiende de 8 a 12 bits: los 8 bits más significativos corresponden al código de operación y los 4 bits menos significativos son ceros.

```
always @ (In) begin
        Q = (In << 4); // In es desplazado 4 bits hacia la izquierda
end
```

Finalmente, colocamos la etiqueta endmodule para indicar la terminación del módulo.

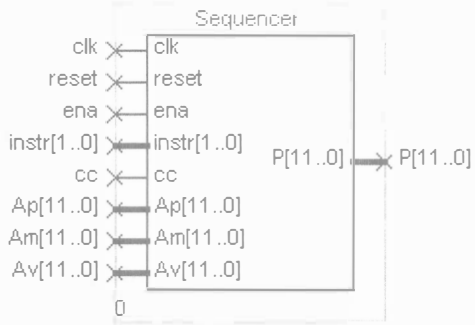
```
endmodule
```

B.3.2.2 SECUENCIADOR

El secuenciador es el orquestador de la unidad de control. Su función es sincronizar y dirigir las tareas de control necesarias para la ejecución de una instrucción. Como puede observar, la importancia del secuenciador es notable, sin embargo, su presencia sería inútil sin el soporte que le ofrecen el resto de los módulos de control.

Cómo funciona el Secuenciador

El siguiente diagrama muestra las entradas y salidas que componen al secuenciador; además, se explica brevemente la función de cada una de ellas.



Entradas:

clk	Señal de reloj (disparo con flanco positivo)
reset	Inicializa al secuenciador
ena	Habilita las operaciones en el secuenciador
instr[1..0]	Le indica al secuenciador qué operación ejecutar
cc	Le indica al secuenciador de dónde proviene la dirección del estado siguiente
Ap[11..0]	Dirección de salto condicional (proviene del registro de liga)
Am[11..0]	Dirección de salto de transformación (proviene del módulo ExtendedInstruction)
Av[11..0]	Dirección de salto condicional (proviene de la UCI)

Salidas:

P[11..0]	Dirección de salida del secuenciador (estado presente)

Análisis del módulo Sequencer.v

Las primeras líneas del código son comentarios.

```
/*********************************************************************
 * Module: Secuenciador -> sequencer.v                               *
 *********************************************************************/
```

Enseguida, se declara un módulo llamado *sequencer* y el conjunto de señales que lo componen.

```
module sequencer (clk, reset, ena, instr, cc, Ap, Am, Av, P);
```

Las siguientes líneas también son comentarios.

```
/*********************************************************************
 * Parámetros:                                                       *
 *      clk     - señal de reloj                                     *
 *      reset   - señal de clear síncrona                            *
 *      ena     - señal de habilitación síncrona                     *
 *      instr   - instrucción a ejecutar                             *
 *      cc      - si cc=1, el dato proviene del registro uPc, de lo contrario,  *
 *                de alguno de los registros conectados al secuenciador  *
 *      Ap      - dato procedente del registro de liga               *
 *      Am      - dato procedente del registro de instrucción        *
 *      Av      - dato procedente de la unidad de interrupciones     *
 *      P       - bus de salida (estado presente)                    *
 *********************************************************************/
```

A continuación, se define el tipo y tamaño de las señales declaradas previamente.

```
input          clk, reset, ena, cc;
input  [ 1:0]  instr;
input  [11:0]  Ap, Am, Av;
output [11:0]  P;
```

Se declaran dos variables del tipo **reg** para el manejo interno del secuenciador. La variable *temp* almacena una dirección de salto y la variable *uPc* la dirección del estado presente más uno.

```
reg    [11:0]  temp, uPc;
```

El comportamiento del secuenciador se implanta por medio de una construcción **always** sensible a los flancos de subida del reloj.

La construcción comienza revisando las señales *ena* y *reset*; si ambas señales presentan un nivel lógico alto, la variable *temp* se coloca a ceros. Si la señal *ena* vale '1' y la señal *reset* vale '0', la dirección del estado presente se incrementa en una unidad y se ejecuta la sentencia **case**.

La sentencia case revisa el valor de la señal *instr* para conocer la próxima instrucción que el secuenciador ejecutará. Las instrucciones válidas para el secuenciador son las siguientes:

Instrucción	Descripción
00 – Continúa	En esta instrucción la dirección del estado siguiente la proporciona el uPc.
01 – Salto Condicional	En esta instrucción se revisa la señal cc, si cc=1 la dirección del estado siguiente la proporciona el uPc, si no, la proporciona Ap.
10 – Salto de Transformación	La dirección del estado siguiente se obtiene del bus Am.
11 – Salto Condicional usando la Dirección de las Interrupciones	En esta instrucción se revisa la señal cc, si cc=1 la dirección del estado siguiente la proporciona el uPc, si no, la proporciona Av.

La construcción always termina con una etiqueta **else** para atender a las combinaciones de las variables *ena* y *reset* no contempladas; en este bloque de sentencias, se asigna el valor de la salida del secuenciador a la variable *temp*, en otras palabras, el valor de la salida *P* no se modifica.

```
always @ (posedge clk) begin
        if (ena & reset) // Inicializa al secuenciador
                temp = 0;
        else if (ena & !reset) begin // Calcula el valor del estado presente más uno
                uPc = P + 1;
                case (instr) // Decodifica la instrucción
                        // Continúa
                        2'b00 : temp = uPc;
                        // Salto condicional
                        2'b01 : if (!cc) temp = Ap;
                                else     temp = uPc;
                        // Salto de transformación
                        2'b10 : temp = Am;
                        // Salto condicional usando la dirección de las interrupciones
                        2'b11 : if (!cc) temp = Av;
                                else     temp = uPc;
                endcase
        end
        else
                temp = P;
end
```

Por último, asignamos a la señal *P* el valor de *temp*, y cerramos la definición del módulo.

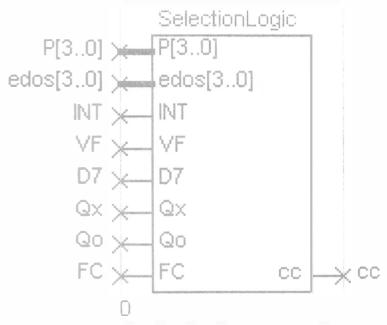
```
assign P = temp;

endmodule
```

B.3.2.3 LÓGICA DE SELECCIÓN

Este módulo mantiene una estrecha relación de trabajo con el secuenciador. Su función es generar el valor de *cc* dependiendo de los valores de los campos *Prueba* y *VF*.

Cómo funciona la Lógica de Selección

El siguiente diagrama muestra las entradas y salidas que componen al módulo *SelectionLogic*; además, se explica brevemente la función de cada una de ellas.



Entradas:

P[3..0]	Campo de prueba
edos[3..0]	Estados de la arquitectura (banderas de negativo, cero, sobreflujo y acarreo)
INT	Señal de interrupción externa
VF	Campo de verdadero-falso
D7	Bit más significativo del registro de corrimiento Qi
Qx	Variable auxiliar para los saltos condicionales
Qo	Bit menos significativo del registro de corrimiento Qi
FC	Estado del contador de la UPA

Salidas:

cc	Señal de salida hacia el secuenciador

Análisis del módulo SelectionLogic.v

Las primeras líneas del código son comentarios.

```
/*****************************************************************
 * Module:  Lógica de Selección -> selectionlogic.v              *
 *****************************************************************/
```

Declaramos un módulo llamado *selectionlogic* y el conjunto de señales que lo componen.

```
module selectionlogic (P, edos, INT, VF, D7, Qx, Qo, FC, cc);
```

Las siguientes líneas también son comentarios.

```
/*****************************************************************
 * Parámetros:                                                   *
 *      P          - selecciona una variable para probar su valor *
 *      edos       - banderas                                     *
 *      INT        - señal de interrupción externa                *
```

```
*       VF, Qx, ...       - señales auxiliares internas                    *
*************************************************************************/
```

Enseguida, se define el tipo y tamaño de las señales declaradas previamente. Cabe recordar que las variables *cc* e *instr* son utilizadas por el secuenciador para elegir la procedencia de la dirección de salto. Si *cc*=1, la dirección provendrá de uPc, en caso contrario, la dirección se toma de uno de los buses conectados al secuenciador.

```
input          INT, VF, D7, Qx, Qo, FC;
input  [3:0]   P, edos;
output         cc;
reg            cc;
```

El cuerpo del módulo *SelectionLogic* se implanta utilizando una construcción **always** sensible a los cambios en la señal *P*. Dentro de la construcción always se define una sentencia **case** que compara el valor de la señal *P* con los valores de sus etiquetas. Cuando encuentra una coincidencia, el bloque de código asociado se ejecuta.

Por ejemplo, supongamos que deseamos verificar que el valor de la bandera de acarreo es 1, por lo tanto, los valores de las señales de control deberán ser *P*[]=4'b0001 y *VF*=1. Una vez que el procedimiento se ejecuta, la sentencia case encuentra como correspondencia a la etiqueta 4'h1. El bloque de código de esta etiqueta valida la igualdad entre el valor de la bandera de acarreo (*edos*[0]) y *VF*; si ambos valores son iguales *cc* se coloca a cero, de lo contrario, *cc* vale uno.

A continuación se anexa una tabla con la información sobre las variables sensadas y su procedencia.

Campo P	Descripción
0	Prueba el valor de Qx, una variable auxiliar útil en los saltos condicionales
1	Prueba el valor de la bandera de Carry
2	Prueba el valor de la bandera de Overflow
3	Prueba el valor de la bandera de Zero
4	Prueba el valor de la bandera de Negative
5	Prueba el valor de la operación lógica: Carry OR Zero
6	Prueba el valor de la operación lógica: Overflow XOR Negative
7	Prueba el valor de la operación lógica: Zero OR (Overflow XOR Negative)
8	Prueba el valor de D7, bit más significativo del registro de corrimiento Q
9	Prueba el valor de INT, interrupción externa
A	Prueba el valor de Qo, bit menos significativo del registro de corrimiento Q
B	Prueba el valor de FC, estado del contador de la UPA (indica si el contador ha alcanzado la cuenta máxima)

```
always @ (P) begin
        case (P)
                // Revisa el valor de Qx (véase el modo de direccionamiento implícito)
                4'h0:   begin
                        if (Qx==VF) cc = 0;
```

```
            else cc = 1;
            end
// Revisa el valor de la bandera de Carry (acarreo)
4'h1:   begin
            if (edos[0]==VF) cc = 0;
            else cc = 1;
            end
// Revisa el valor de la bandera de Overflow (sobreflujo)
4'h2:   begin
            if (edos[1]==VF) cc = 0;
            else cc = 1;
            end
// Revisa el valor de la bandera de Zero (cero)
4'h3:   begin
            if (edos[2]==VF) cc = 0;
            else cc = 1;
            end
// Revisa el valor de la bandera de Negative (bandera de signo)
4'h4:   begin
            if (edos[3]==VF) cc = 0;
            else cc = 1;
            end
// Revisa el valor de la operación [Carry or Zero]
4'h5:   begin
            if ((edos[0] | edos[2])==VF) cc = 0;
            else cc = 1;
            end
// Revisa el valor de la operación [Overflow xor Negative]
4'h6:   begin
            if ((edos[1] ^ edos[3])==VF) cc = 0;
            else cc = 1;
            end
// Revisa el valor de la operación [Zero or (Overflow xor Negative)]
4'h7:   begin
            if ((edos[2] | (edos[1] ^ edos[3]))==VF) cc = 0;
            else cc = 1;
            end
// Revisa el valor de D7, el bit más significativo del registro de corrimiento Qi
4'h8:   begin
            if (D7==VF) cc = 0;
            else cc = 1;
            end
// Revisa el valor de INT (señal de interrupción externa)
4'h9:   begin
            if (INT==VF) cc = 0;
            else cc = 1;
            end
// Revisa el valor de Qo, el bit menos significativo del registro de corrimiento Qi
4'hA:   begin
            if (Qo==VF) cc = 0;
            else cc = 1;
            end
// Revisa el valor de FC (estado del contador)
```

```
            4'hB:   begin
                    if (FC==VF) cc = 0;
                    else cc = 1;
                    end
        endcase
    end
```

Finalmente, cerramos la definición del módulo.

```
    endmodule
```

B.3.2.4 MEMORIA DE MICROPROGRAMA

La memoria de microprograma contiene la información suficiente para el control tanto de la arquitectura interna como de la externa; es decir, la memoria de microprograma sabe qué líneas de control debe activar o desactivar para cada módulo de la arquitectura, en base a la dirección de entrada que recibe del secuenciador.

Cómo construir la Memoria de Microprograma

Este módulo no utiliza Verilog HDL debido a que el software MAX+PLUS II no soporta todas las funciones y capacidades de Verilog; por lo tanto, para construir la memoria de microprograma utilizaremos una construcción **TABLE** del lenguaje AHDL.

Una tabla en AHDL guarda las transiciones entre los niveles lógicos para todos los grupos de nodos o señales declaradas. La definición de una tabla comienza con la etiqueta **TABLE** y finaliza con la etiqueta **END TABLE** seguida de un punto y coma. Las señales y sus transiciones deberán definirse dentro de estas etiquetas.

La construcción TABLE es muy simple: primero se declaran todas las señales de entrada separadas mediante comas, enseguida se colocan los símbolos => y se listan las señales de salida también separadas mediante comas. Una vez declarados todos nombres de nuestras señales terminamos la sentencia con un punto y coma. De manera similar se definen las transiciones para estas señales, no olvide colocar un punto y coma al final de cada línea de transiciones.

Como ejemplo revisemos la línea que le sigue al comentario % Fetch Cycle %. Cuando el secuenciador genere la salida 12'h002, la tabla encontrará como correspondencia a esta línea y los valores de las señales de salida *Liga*, *Instr*, *Prueba*, *VF*, *R_Wn*, *ASn*, *BD*, etcétera, cambiarán a los nuevos valores de transición: Liga[11..0]=H"003", Instr[1..0]=B"00", Prueba[3..0]=H"0", VF=0, R_Wn=1, ASn=1, BD=0, etcétera.

Por cierto, los comentarios en AHDL comienzan y terminan con la secuencia %. Todo lo que esté dentro de los delimitadores de comentarios será ignorado por el compilador.

A continuación se reproduce un pequeño fragmento de la memoria de microprograma.

```
TABLE

    Dir[11..0] => Liga[11..0], Instr[1..0], Prueba[3..0], VF, R_Wn, ASn, BD, ... ;

    % Fetch Cycle %
    H"002"   =>   H"003",   B"00",        H"0",  0,   1,   1,  0, ... ;
    H"003"   =>   H"002",   B"10",        H"0",  0,   1,   0,  0, ... ;

    % Instrucción: ldab (modo inmediato) [ C6 + dato_8bits => C6:ii ] %
    H"860"   =>   H"861",   B"00",        H"0",  0,   1,   1,  0, ... ;
    H"861"   =>   H"862",   B"00",        H"0",  0,   1,   0,  0, ... ;
    H"862"   =>   H"863",   B"11",        H"9",  1,   1,   1,  0, ... ;
    H"863"   =>   H"003",   B"01",        H"0",  0,   1,   1,  0, ... ;
    ...      ...      ...

END TABLE;
```

En resumen, la memoria de microprograma guarda el estado de activación o desactivación en el que se encuentran las líneas de control de cada componente de la arquitectura. Por cuestiones de espacio no se imprime todo el contenido de la memoria de microprograma, pero al menos se indica cómo fueron codificadas algunas de las instrucciones vistas en el capítulo 6. Tenga presente que las instrucciones mostradas en esta sección fueron codificadas utilizando la filosofía de las cartas ASM, no obstante, encontrará varios cambios, principalmente, en la sintaxis utilizada y en algunos nombres de las señales de control. Además, como convención, se muestran los valores de todas las señales de control en formato hexadecimal.

Observe la siguiente instrucción.

```
Instrucción: inx
Incrementa en una unidad al registro índice IX

OPERACIÓN:              IX ← (IX)+1
MODO:                   Inherente
OPCODE:                 08
CICLO FETCH:            fetch   =>   Instr[]=0, EPCn[]=3, CBDn=0
                                     Reg. Direcciones←PC
                        fetch1  =>   Instr[]=2, ASn=0, EEnaY=1, CPC[]=1, ResetAux=1
                                     Se ejecuta un salto de transformación, incrementa PC,
                                     habilita el registro IX y limpia el registro auxiliar
MICRO-OPERACIONES:      inx     =>   Instr[]=Ó, CX[]=1, HINT=1
                                     Incrementa IX y habilita las interrupciones
                        inx1    =>   Instr[]=3, Prueba[]=9, VF=1, Zena=0, Z[]=7
                                     Actualiza banderas en el CCR y revisa interrupciones
                        inx2    =>   Instr[]=1, Prueba[]=0, VF=0, Liga[]=003
                                     EPCn[]=3, CBDn=0
                                     En caso de que no exista interrupción, se ejecutan las
                                     micro-operaciones para el estado fetch y se salta a fetch1
```

Todas las instrucciones codificadas en este apartado comienzan con su nombre, una breve descripción de lo que hacen, el tipo de acceso que se utiliza, su código de operación, y un listado con las micro-operaciones necesarias para ejecutar dicha instrucción. Por otra parte, las micro-

operaciones están compuestas de tres partes: el nombre del estado, las señales que se activan en ese estado y la descripción de la tarea o tareas que ejecuta el microprocesador. Por ejemplo, los estados 'fetch' y 'fetch1' leen de memoria la siguiente instrucción a ejecutar y decodifican dicha instrucción. El estado 'inx', primer estado de la instrucción, activa la señal CX0 la cual incrementa en una unidad al registro índice X, y activa la señal HINT que habilita la generación de la señal INT en la unidad de control de interrupciones. El segundo estado, 'inx1', actualiza el valor de la bandera de cero mediante la activación de las señales Zena, Z2, Z1y Z0, y prepara al secuenciador para ejecutar un salto si existe una interrupción externa. Para esta última tarea, la lógica de selección debe generar la señal *cc* usando los valores de Prueba[]=9 (selecciona la señal INT) y *VF*=1. Si *Prueba* es igual a *VF*, *cc* se coloca a cero y el secuenciador ejecuta un salto hacia la dirección dada por la unidad de control de interrupciones; en caso contrario, el secuenciador toma la dirección del estado siguiente del registro μPC. Finalmente, en el estado 'inx2' se comienza un nuevo ciclo fetch para leer de memoria la siguiente instrucción a ejecutar, y se ejecuta un salto condicional al estado 'fetch1' para continuar con la decodificación de la nueva instrucción.

Una metodología similar se utiliza para codificar el resto de las instrucciones.

Instrucción: xgdx
Intercambian contenido el doble acumulador D y el registro índice IX

OPERACIÓN:	(IX) ⇔ (ACCD)		
MODO:	Inherente		
OPCODE:	8F		
MICRO-OPERACIONES:	xgdx	=>	Instr[]=0, EXn[]=4, CA[]=5
			Reg. Auxiliar←IX
	xgdx1	=>	Instr[]=0, EA[]=2, EB[]=1, CX[]=5, HINT=1
			IX←D y habilita las interrupciones
			Recuerde que el doble acumulador D se forma concatenando
			el contenido del acumulador A con el contenido de B
	xgdx2	=>	Instr[]=3, Prueba[]=9, VF=1, EAn[]=4, EA[]=2
			WAn=0, EB[]=1, WBn=0
			D←Reg. Auxiliar y revisa interrupciones
	xgdx3	=>	Instr[]=1, Prueba[]=0, VF=0, Liga[]=003
			EPCn[]=3, CBDn=0
			En caso de que no exista interrupción, se ejecutan las
			micro-operaciones para el estado fetch y se salta a fetch1

Instrucción: ldab
Carga el contenido de memoria en el acumulador B

OPERACIÓN:	ACCB ← (Memoria)		
MODO:	Inmediato		
OPCODE:	C6		
MICRO-OPERACIONES:	ldab	=>	Instr[]=0, EPCn[]=3, CBDn=0
			Reg. Direcciones←PC
	ldab1	=>	Instr[]=0, ASn=0, CPC[]=1, WBn=0, EB[]=1, HINT=1
			Incrementa PC, ACCB←dato de memoria
			y habilita las interrupciones
	ldab2	=>	Instr[]=3, Prueba[]=9, VF=1
			V[]=1, Z[]=N[]=6, Vena=Nena=Zena=0
			Actualiza banderas en el CCR y revisa interrupciones

ldab3 => Instr[]=1, Prueba[]=0, VF=0, Liga[]=003
EPCn[]=3, CBDn=0
En caso de que no exista interrupción, se ejecutan las
micro-operaciones para el estado fetch y se salta a fetch1

Instrucción: ldaa
Carga el contenido de memoria en el acumulador A

OPERACIÓN: ACCA ← (Memoria)
MODO: Inmediato
OPCODE: 86
MICRO-OPERACIONES:
ldaa => Instr[]=0, EPCn[]=3, CBDn=0
Reg. Direcciones←PC
ldaa1 => Instr[]=0, ASn=0, CPC[]=1, WAn=0, EA[]=1, HINT=1
Incrementa PC, ACCA←dato de memoria
y habilita las interrupciones
ldaa2 => Instr[]=3, Prueba[]=9, VF=1
V[]=1, Z[]=N[]=5, Vena=Nena=Zena=0
Actualiza banderas en el CCR y revisa interrupciones
ldaa3 => Instr[]=1, Prueba[]=0, VF=0, Liga[]=003
EPCn[]=3, CBDn=0
En caso de que no exista interrupción, se ejecutan las
micro-operaciones para el estado fetch y se salta a fetch1

Instrucción: suba
Le resta al contenido del acumulador A el contenido de la memoria

OPERACIÓN: ACCA ← (ACCA) - (Memoria)
MODO: Extendido
OPCODE: B0
MICRO-OPERACIONES:
suba => Instr[]=0, EPCn[]=3, CBDn=0
Reg. Direcciones←PC
suba1 => Instr[]=0, ASn=0, CPC[]=1, CA[]=4, BD=1
Incrementa PC y Reg. Auxiliar←dirección parte alta
suba2 => Instr[]=0, CBDn=0, EPCn[]=3
Reg. Direcciones←PC
suba3 => Instr[]=0, ASn=0, CPC[]=1, CA[]=3
Incrementa PC y Reg. Auxiliar←dirección parte baja
suba4 => Instr[]=0, EAn[]=3, CBDn=0
Reg. Direcciones←Reg. Auxiliar
suba5 => Instr[]=0, ASn=0, EA[]=3, UPA=095, Hab=1, Haba=1,
Selmux[]=1, HINT=1
Calcula (ACCA)-(Memoria) y habilita las interrupciones
suba6 => Instr[]=3, Prueba[]=9, VF=1, EALUn=0, WAn=0
EA[]=3, Nena=Zena=Vena=Cena=0, N[]=Z[]=V[]=C[]=3
Actualiza banderas en el CCR, ACCA←Resultado_UPA
y revisa interrupciones
suba7 => Instr[]=1, Prueba[]=0, VF=0, Liga[]=003
EPCn[]=3, CBDn=0
En caso de que no exista interrupción, se ejecutan las
micro-operaciones para el estado fetch y se salta a fetch1

Instrucción: bra
Salto incondicional

OPERACIÓN: PC ← (PC) + 2 + Desplazamiento
MODO: Relativo
OPCODE: 20
MICRO-OPERACIONES:

bra => Instr[]=0, EPCn[]=3, CBDn=0
Reg. Direcciones←PC

bra1 => Instr[]=0, ASn=0, CPC[]=1, UPA=037, Hab=1, Haba=1
Q←Desplazamiento e incrementa PC

bra2 => Instr[]=0, EPCn[]=6, UPA=086, Hab=1, Haba=1
Calcula PC_baja+Desplazamiento

bra3 => Instr[]=1, Prueba[]=8, VF=1, Liga[]=207
CPC[]=3, EALUn=0, EDEn=0
PC_baja←Resultado_UPA. Si D7=1, entonces se salta al estado bra+7, si no, se ejecuta el estado bra+4

bra4 => Instr[]=0, EPCn[]=5, UPA=087, Hab=1, Haba=1, Selbus=1
Selmux[]=3, HINT=1
Calcula PC_alta+Carry_alu y habilita las interrupciones

bra5 => Instr[]=3, Prueba[]=9, VF=1
CPC[]=4, EALUn=0, Selupa=1, EDEn=0
PC_alta←Resultado_UPA y revisa interrupciones

bra6 => Instr[]=1, Prueba[]=0, VF=0, Liga[]=003
EPCn[]=3, CBDn=0
En caso de que no exista interrupción, se ejecutan las micro-operaciones para el estado fetch y se salta a fetch1

bra7 => Instr[]=1, Liga[]=205, Prueba[]=0, VF=0, EPCn[]=5, UPA=0A7, Hab=1, Haba=1, Selbus=1, Selmux[]=3, HINT=1
Calcula PC_alta-Carry_alu, habilita las interrupciones y salta al estado bra+5

Instrucción: beq
Prueba el estado del bit Z y ocasiona un salto si Z=1

OPERACIÓN: PC ← (PC) + 2 + Desplazamiento, si (Z)=1
MODO: Relativo
OPCODE: 27
MICRO-OPERACIONES:

beq => Instr[]=1, Prueba[]=3, VF=1, Liga[]=201
EPCn[]=3, CBDn=0, HINT=1
Reg. Direcciones←PC, habilita las interrupciones y salta al estado bra1 si se cumple la condición Z=1

beq1 => Instr[]=3, Prueba[]=9, VF=1, CPC[]=1
Si Z=0 se incrementa PC y revisa interrupciones

beq2 => Instr[]=1, Prueba[]=0, VF=0, Liga[]=003
EPCn[]=3, CBDn=0
En caso de que no exista interrupción, se ejecutan las micro-operaciones para el estado fetch y se salta a fetch1

Instrucción: jsr
Salto a subrutina

OPERACIÓN: PC ← (PC) + 3

(SP) ← PC parte baja
SP ← (SP) - 1
(SP) ← PC parte alta
SP ← (SP) - 1
PC ← Dirección de la subrutina

MODO: Extendido
OPCODE: BD
MICRO-OPERACIONES:

jsr => Instr[]=0, EPCn[]=3, CBDn=0
Reg. Direcciones←PC

jsr1 => Instr[]=0, ASn=0, CPC[]=1, CA[]=4, BD=1
Incrementa PC y Reg. Auxiliar←dirección parte alta

jsr2 => Instr[]=0, CBDn=0, EPCn[]=3
Reg. Direcciones←PC

jsr3 => Instr[]=0, ASn=0, CPC[]=1, CA[]=3
Incrementa PC y Reg. Auxiliar←dirección parte baja

jsr4 => Instr[]=0, ESPn[]=3, CBDn=0
Reg. Direcciones←SP

jsr5 => Instr[]=0, CSP[]=2, ASn=0, R_Wn=0, BD=0, EPCn[]=6
Decrementa SP y (Reg. Direcciones)←PC_baja

jsr6 => Instr[]=0, ESPn[]=3, CBDn=0
Reg. Direcciones←SP

jsr7 => Instr[]=0, CSP[]=2, ASn=0, R_Wn=0, BD=1, EPCn[]=5
HINT=1
Decrementa SP, (Reg. Direcciones)←PC_alta
y habilita las interrupciones

jsr8 => Instr[]=3, Prueba[]=9, VF=1, CPC[]=5, EAn[]=4
PC←Reg. Auxiliar y revisa interrupciones

jsr9 => Instr[]=1, Prueba[]=0, VF=0, Liga[]=003
EPCn[]=3, CBDn=0
En caso de que no exista interrupción, se ejecutan las
micro-operaciones para el estado fetch y se salta a fetch1

Instrucción: rts
Regreso de subrutina

OPERACIÓN:

SP ← (SP) + 1
PC ← Dirección de regreso parte alta
SP ← (SP) + 1
PC ← Dirección de regreso parte baja

MODO: Inherente
OPCODE: 39
MICRO-OPERACIONES:

rts => Instr[]=0, CSP[]=1
Incrementa SP

rts1 => Instr[]=0, ESPn[]=3, CBDn=0
Reg. Direcciones←SP

rts2 => Instr[]=0, CSP[]=1, ASn=0, BD=1, CPC[]=4
Incrementa SP y PC_alta←Dir. de regreso parte alta

rts3 => Instr[]=0, ESPn[]=3, CBDn=0, HINT=1
Reg. Direcciones←SP y habilita las interrupciones

rts4 => Instr[]=3, Prueba[]=9, VF=1
ASn=0, BD=0, CPC[]=3
PC_baja←Dir. de regreso parte baja y revisa interrupciones

rts5 => Instr[]=1, Prueba[]=0, VF=0, Liga[]=003

EPCn[]=3, CBDn=0
En caso de que no exista interrupción, se ejecutan las
micro-operaciones para el estado fetch y se salta a fetch1

Instrucción: mul
Multiplica el contenido del acumulador A por el del acumulador B y asigna el resultado en D=A:B

OPERACIÓN: ACCD ← (ACCA) x (ACCB)
MODO: Inherente
OPCODE: 3D
MICRO-OPERACIONES:

mul => Instr[]=0, UPA=030, Hab=1, Haba=1, EA[]=3
Qi←Acumulador A

mul1 => Instr[]=0, UPA=0C3, Hab=1, Haba=1
Yi←0

mul2 => Instr[]=0, EG[]=1, WAn=0, EA[]=2, UPA=18A
Inicializa contador de la UPA y Acumulador A←0

mul3 => Instr[]=1, Prueba[]=A, VF=0, Liga=3D5
Si Qo=0, entonces salta al estado mul5, si no, va a mul4

mul4 => Instr[]=0, UPA=081, Hab=1, Haba=1, EB[]=3, EA[]=3
Y←(A+B)

mul5 => Instr[]=0, UPA=283, Hab=1, Log=1, Rot=1, Selmux[]=3
Q←Q/2 e Y←Y/2

mul6 => Instr[]=1, Prueba[]=B, VF=0, Liga=3D3
UPA=1CB, Haba=1, EALUn=0, WAn=0, EA[]=3
Incrementa contador y Acumulador A←Y
Si FC=0, entonces salta al estado mul3, si no, va a mul7

mul7 => Instr[]=0, UPA=0B2, Haba=1, Hab=1, HINT=1, Cena, C[]=4
Y←Q y habilita las interrupciones

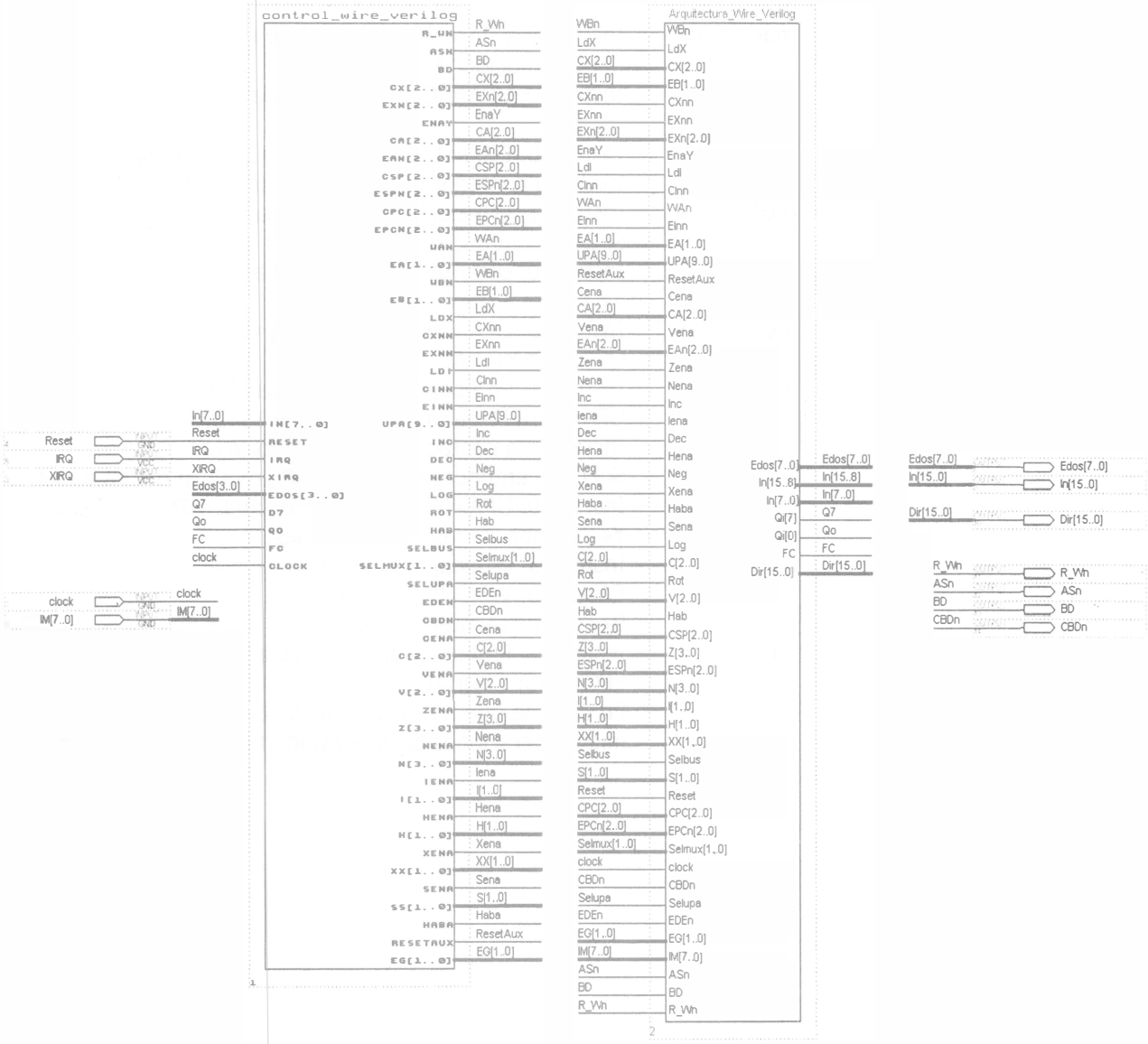
mul8 => Instr[]=3, Prueba[]=9, VF=1
EALUn=0, WBn=0, EB[]=3
B←Y y revisa interrupciones

mul9 => Instr[]=1, Prueba[]=0, VF=0, Liga[]=003
EPCn[]=3, CBDn=0
En caso de que no exista interrupción, se ejecutan las
micro-operaciones para el estado fetch y se salta a fetch1

B.4 ETAPA 3: UN MICROPROCESADOR DE 8 BITS

B.4.1 MODULE_WIRE_VERILOG

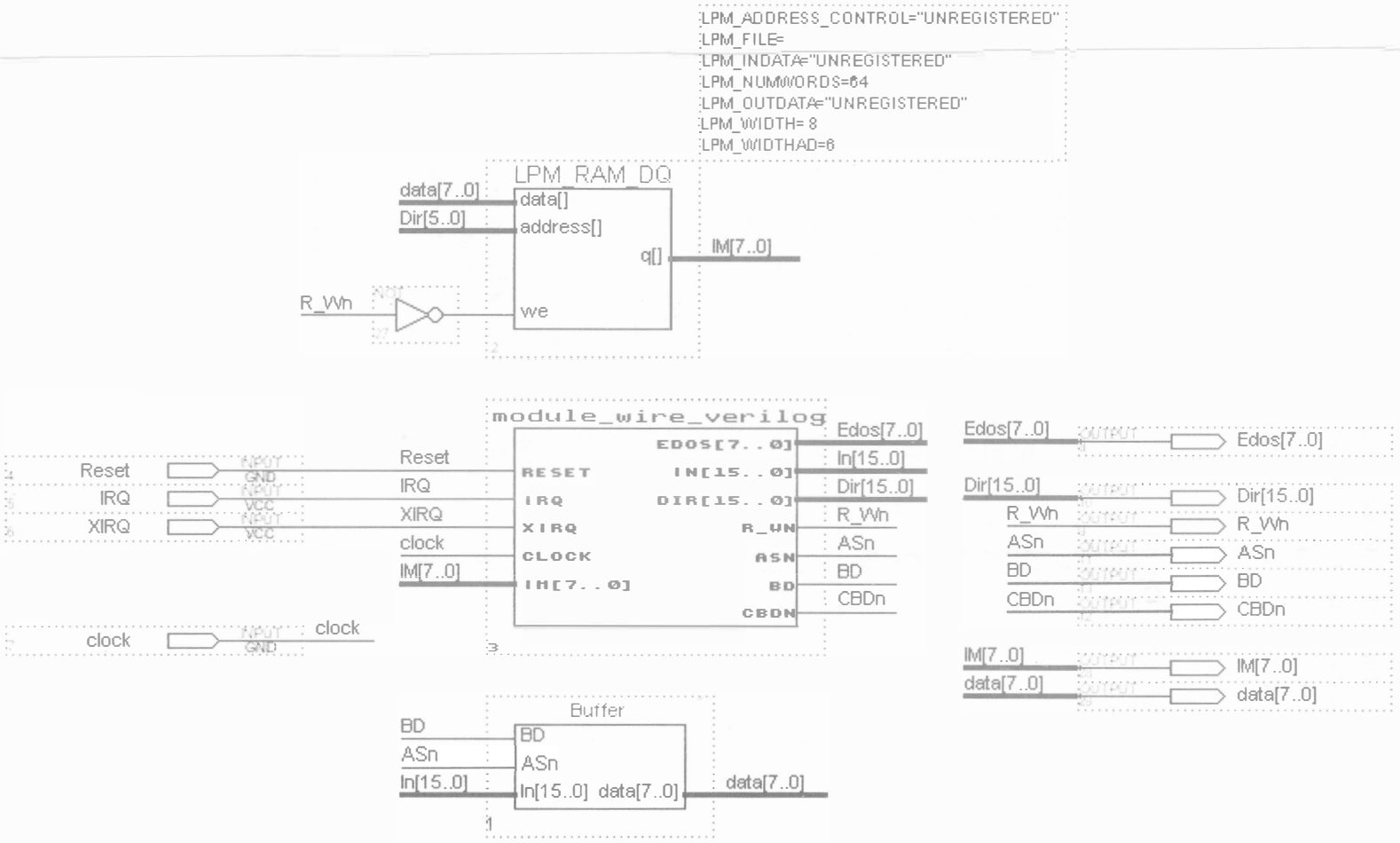
El siguiente diagrama muestra la unión entre el módulo a*rquitectura_wire_verilog* obtenido en la etapa 1 y el módulo c*ontrol_wire_verilog* obtenido en la etapa 2.



Por último, falta anexar un módulo muy importante: la memoria de programas y datos; para ello, primero vamos a crear el símbolo del módulo anterior que llamaremos *module_wire_verilog*, y lo conectaremos a una memoria de 64 palabras de 8 bits cada una. También agregaremos un búfer que conmutará los datos de la memoria hacia la parte baja o hacia la parte alta del bus de datos interno. El diagrama final es el siguiente.

MICROPROCESADOR DE 8 BITS USANDO VERILOG HDL

PROTO_WIRE_VERILOG



B.4.2 LA MEMORIA EXTERNA

En la memoria externa se almacenan las instrucciones en lenguaje ensamblador. El microprocesador es capaz de leer estas instrucciones, decodificarlas y ejecutar las micro-operaciones necesarias para su realización. Por el momento, este procesador sólo ejecuta algunas instrucciones del set de instrucciones del microcontrolador MC68HC11; estas instrucciones se enuncian en la siguiente tabla.[19]

Instrucción	Opcode	Descripción
inx	08	Incrementa en una unidad el contenido del registro IX.
iny	18 08	Incrementa en una unidad el contenido del registro IY.
ins	31	Incrementa en una unidad el contenido del apuntador de pila.
ldaa	86	Carga un dato inmediato de 8 bits en el acumulador A.
ldab	C6	Carga un dato inmediato de 8 bits en el acumulador B.
suba	B0	Resta al contenido del acumulador A el contenido de la memoria. El resultado de la resta es colocado nuevamente en A.
bra	20	Salto incondicional.
xgdx	8F	Intercambia el contenido del doble acumulador por el contenido del registro IX.
beq	27	Salto condicional. Prueba el estado de la bandera de cero (Z) y ejecuta el salto si Z=1.
jsr	BD	Salto a subrutina.
rts	39	Regreso de subrutina.
mul	3D	Multiplica dos valores de 8 bits sin signo. Estos valores corresponden a los contenidos de los acumuladores A y B. El resultado, un valor sin signo de 16 bits, es guardado en el doble acumulador D.
staa	B7	Guarda en memoria el contenido del acumulador A.
lds	8E	Carga un dato inmediato de 16 bits en el registro SP (stack pointer).
asla	48	Corrimiento hacia la izquierda del contenido del acumulador A. El resultado es colocado nuevamente en A.
aba	1B	Suma los contenidos de los acumuladores A y B. El resultado es almacenado en A.

Como ejemplos ilustrativos presentamos dos programas que ejecuta el microprocesador usando el conjunto de instrucciones mostrado en la tabla anterior. Además, se imprime el contenido de la memoria externa para cada ejemplo.

Ejemplo Ilustrativo 1

```
Etiquetas   Mnemónicos     Opcode        Descripción

            lds #003F      ; 8E 00 3F    ; Carga el apuntador de pila con el valor 0x003F
INI:        ldaa #36       ; 86 36       ; Carga en el acumulador A el valor 0x36
```

[19] Para obtener información adicional sobre la función y uso de las instrucciones consulte el manual del microcontrolador MC68HC11.

```
        ldab #7F      ; C6 7F     ; Carga en el acumulador B el valor 0x7F
        aba           ; 1B        ; Suma A+B y coloca el resultado en A
        staa 002D     ; B7 00 2D  ; Almacena el contenido del acumulador A en la dirección
                                  ; de memoria 0x002D
        jsr SUB1      ; BD 00 17  ; Salta a la subrutina SUB1
        suba 002F     ; B0 00 2F  ; Resta al acumulador A el contenido de la dirección de
                                  ; memoria 0x002F. El resultado es colocado en A.
        staa 002E     ; B7 00 2E  ; Almacena el contenido del acumulador A en la dirección
                                  ; de memoria 0x002E
        bra INI       ; 20 ED     ; Salta a la etiqueta INI
SUB1:   asla          ; 48        ; Desplazamiento hacia la izquierda del contenido de A
        rts           ; 39        ; Regreso de subrutina
```

Archivo proto_wire_verilog.mif: Contenido de la Memoria

```
WIDTH = 8;
DEPTH = 64;
ADDRESS_RADIX = HEX;
DATA_RADIX = HEX;

CONTENT BEGIN
    0   : 00;   10  : 00;   20  : 00;   30  : 00;
    1   : 8E;   11  : 2F;   21  : 00;   31  : 00;
    2   : 00;   12  : B7;   22  : 00;   32  : 00;
    3   : 3F;   13  : 00;   23  : 00;   33  : 00;
    4   : 86;   14  : 2E;   24  : 00;   34  : 00;
    5   : 36;   15  : 20;   25  : 00;   35  : 00;
    6   : C6;   16  : ED;   26  : 00;   36  : 00;
    7   : 7F;   17  : 48;   27  : 00;   37  : 00;
    8   : 1B;   18  : 39;   28  : 00;   38  : 00;
    9   : B7;   19  : 00;   29  : 00;   39  : 00;
    A   : 00;   1A  : 00;   2A  : 00;   3A  : 00;
    B   : 2D;   1B  : 00;   2B  : 00;   3B  : 00;
    C   : BD;   1C  : 00;   2C  : 00;   3C  : 00;
    D   : 00;   1D  : 00;   2D  : 00;   3D  : 00;
    E   : 17;   1E  : 00;   2E  : 00;   3E  : 00;
    F   : B0;   1F  : 00;   2F  : 80;   3F  : 00;
END;
```

Ejemplo Ilustrativo 2

```
Etiquetas   Mnemónicos    Opcode       Descripción

            lds #003F     ; 8E 00 3F   ; Carga el apuntador de pila con el valor 0x003F
INI:        ldaa #05      ; 86 05      ; Carga el acumulador A con el valor 0x05
            ldab #07      ; C6 07      ; Carga el acumulador B con el valor 0x07
            mul           ; 3D         ; Multiplica A por B y deja el resultado en D
            xgdx          ; 8F         ; Intercambian contenidos los registros D e IX
            bra INI       ; 20 F8      ; Salta a la etiqueta INI
```

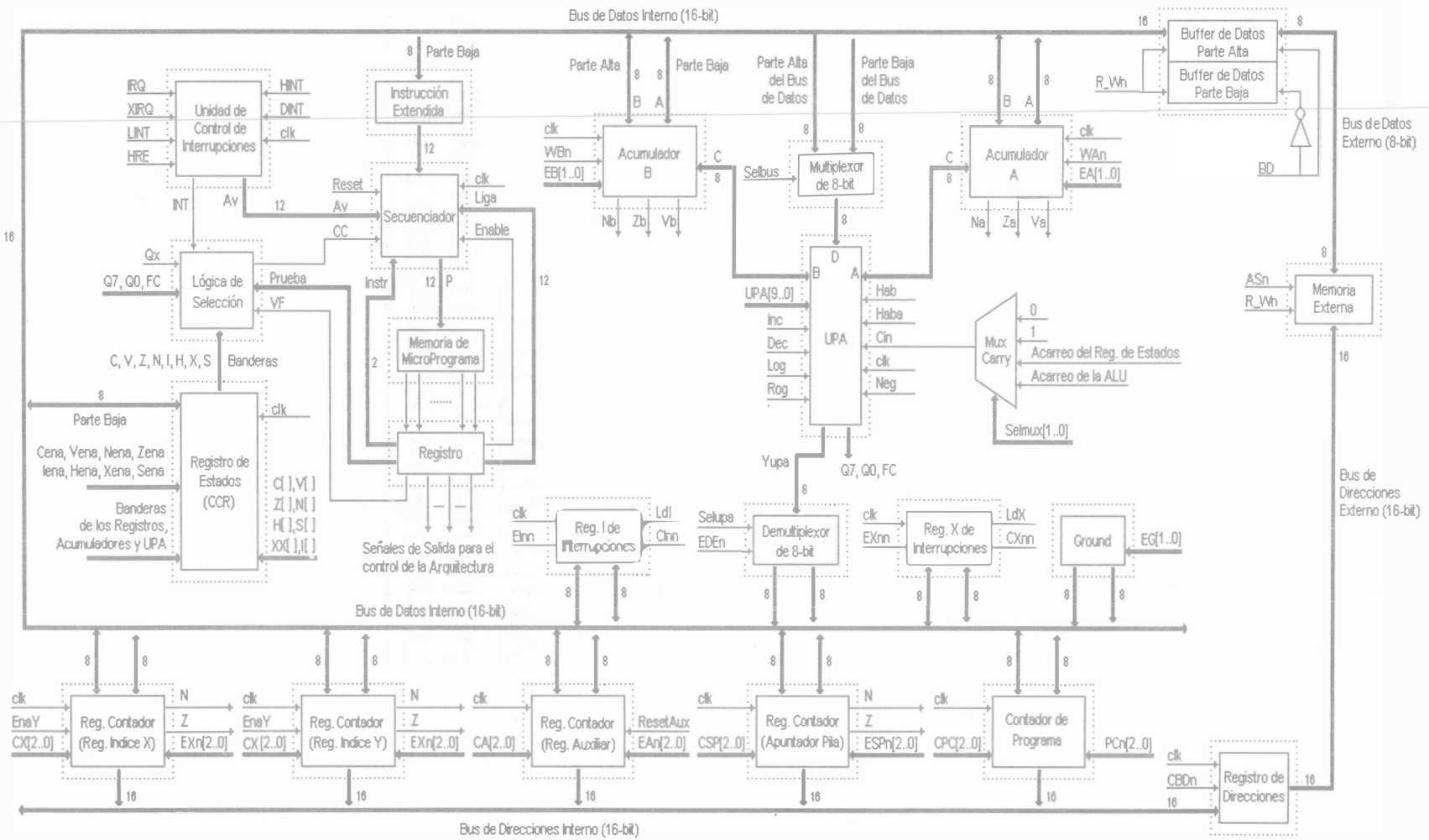
Archivo proto_wire_Verilog_2.mif: Contenido de la Memoria

```
DEPTH = 64;
ADDRESS_RADIX = HEX;
DATA_RADIX = HEX;

CONTENT BEGIN

    0       : 00;       10      : 00;       20      : 00;       30      : 00;
    1       : 8E;       11      : 00;       21      : 00;       31      : 00;
    2       : 00;       12      : 00;       22      : 00;       32      : 00;
    3       : 3F;       13      : 00;       23      : 00;       33      : 00;
    4       : 86;       14      : 00;       24      : 00;       34      : 00;
    5       : 05;       15      : 00;       25      : 00;       35      : 00;
    6       : C6;       16      : 00;       26      : 00;       36      : 00;
    7       : 07;       17      : 00;       27      : 00;       37      : 00;
    8       : 3D;       18      : 00;       28      : 00;       38      : 00;
    9       : 8F;       19      : 00;       29      : 00;       39      : 00;
    A       : 20;       1A      : 00;       2A      : 00;       3A      : 00;
    B       : F8;       1B      : 00;       2B      : 00;       3B      : 00;
    C       : 00;       1C      : 00;       2C      : 00;       3C      : 00;
    D       : 00;       1D      : 00;       2D      : 00;       3D      : 00;
    E       : 00;       1E      : 00;       2E      : 00;       3E      : 00;
    F       : 00;       1F      : 00;       2F      : 00;       3F      : 00;
END;
```

B.4.3 MICROPROCESADOR DE 8 BITS

El diagrama que a continuación se presenta muestra la estructura interna del microprocesador de 8 bits desarrollado en esta sección. Tenga en mente que esta arquitectura es similar a la arquitectura estudiada en el capítulo VI, más no idéntica; por lo tanto, si tiene alguna duda sobre el funcionamiento de un módulo en particular y sus señales de control, deberá consultar la sección, dentro de este apéndice, en donde fue construido ese módulo, o bien, el código fuente que se utilizó para su implantación.

MICROPROCESADOR DE 8 BITS



APÉNDICE C
INTRODUCCIÓN A LOS DISPOSITIVOS LÓGICOS PROGRAMABLES VLSI

C.1 INTRODUCCIÓN

Los dispositivos lógicos programables (PLDs) se utilizan en muchas aplicaciones para reemplazar a los circuitos SSI y MSI, ya que ahorran espacio y reducen el número y el costo de los dispositivos empleados en el diseño. Sin embargo, estos dispositivos son poco útiles cuando se incorporan en el diseño funciones lógicas muy complejas, en este caso, lo más apropiado es utilizar dispositivos con tecnología VLSI (muy alta escala de integración). Un dispositivo VLSI es un circuito que incorpora desde miles hasta millones de compuertas lógicas dentro de un sólo chip.

Gracias a la tecnología VLSI se han producido nuevos dispositivos lógicos programables de alta capacidad; entre ellos encontramos a los CPLDs (Complex Programmable Logic Devices) y a los FPGAs (Field Programmable Gate Arrays). Ambas estructuras comparten las siguientes propiedades:

1. Cuentan con una gran cantidad de compuertas lógicas para implantar lógica combinacional.
2. Cuentan con flip-flops pre-implantados.
3. Tienen el soporte necesario para programar las interconexiones entre la lógica combinacional, los flip-flops y las entradas y salidas del chip.

Estos dispositivos también tienen otras propiedades que varían significativamente de fabricante a fabricante. Entre estas propiedades están la manera de programarlos y el lenguaje empleado para ello. Como ejemplos, estudiaremos brevemente la estructura del CPLD MAX7000S, del PLD FLEX10K[20] y del FPGA XC4000.

C.2 CPLD MAX7000S DE ALTERA

Los MAX son una familia de CPLDs fabricados por Altera que se basan en la tecnología EEPROM de compuerta flotante. La estructura del MAX7000S se muestra en la figura C.1.

El MAX7000S está compuesto de varios bloques de matrices lógicas llamados LABs (Logic Array Blocks). Las salidas de los LABs se conectan a una matriz de interconexiones programables denominada PIA (Programmable Interconnect Array). A esta matriz de interconexión también se conectan las salidas de los bloques de control de entrada/salida, los cuales comunican al circuito con el mundo exterior. La matriz de interconexiones puede re-programarse para conectar las entradas del circuito con los bloques lógicos internos, según las necesidades del diseño.

Cada bloque de matrices lógicas contiene 16 macroceldas, cada una compuesta de tres bloques funcionales: la matriz lógica, la matriz de selección de términos de productos y el registro programable.

[20] Todos los diagramas de estructura de los dispositivos MAX7000S y FLEX10K fueron extraídos de sus respectivas hojas de datos. Para mayor información consulte los documentos "MAX7000 Programmable Logic Device Family Data Sheet" y "FLEX 10K Embedded Programmable Logic Device Family Data Sheet" disponibles en la página web de Altera.

La matriz lógica se utiliza para implantar lógica combinacional, para ello, proporciona cinco términos de productos por macrocelda.

La matriz de selección de términos de productos asigna estos productos a las entradas de la lógica primaria o a las entradas de la lógica secundaria. La lógica primaria, compuesta por las compuertas OR y XOR, sirve para implantar funciones combinacionales, mientras que la lógica secundaria controla las señales de CLEAR, PRESET, CLOCK y CLOCK ENABLE del registro de la macrocelda.

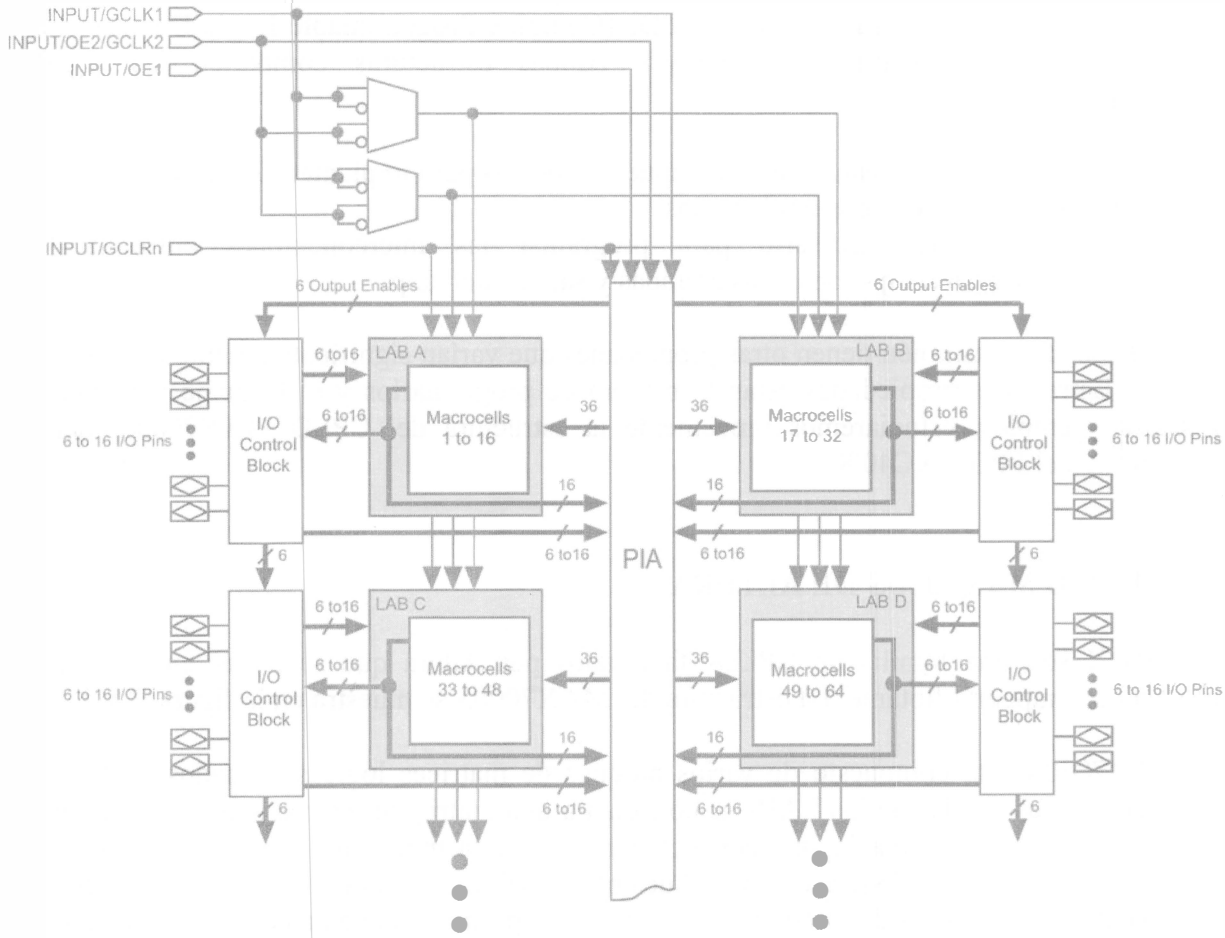


Figura C.1. Estructura del MAX7000S de Altera.

La macrocelda también cuenta con dos tipos de 'expansores' (expanders): los compartidos y los paralelos. Los 'expansores' compartidos invierten los términos de productos de la macrocelda y los realimentan a la matriz lógica. Los 'expansores' paralelos toman prestados los términos de productos de macroceldas adyacentes y los alimentan a la matriz de selección. Gracias a los 'expansores' paralelos una macrocelda puede emplear compuertas AND de macroceldas vecinas.

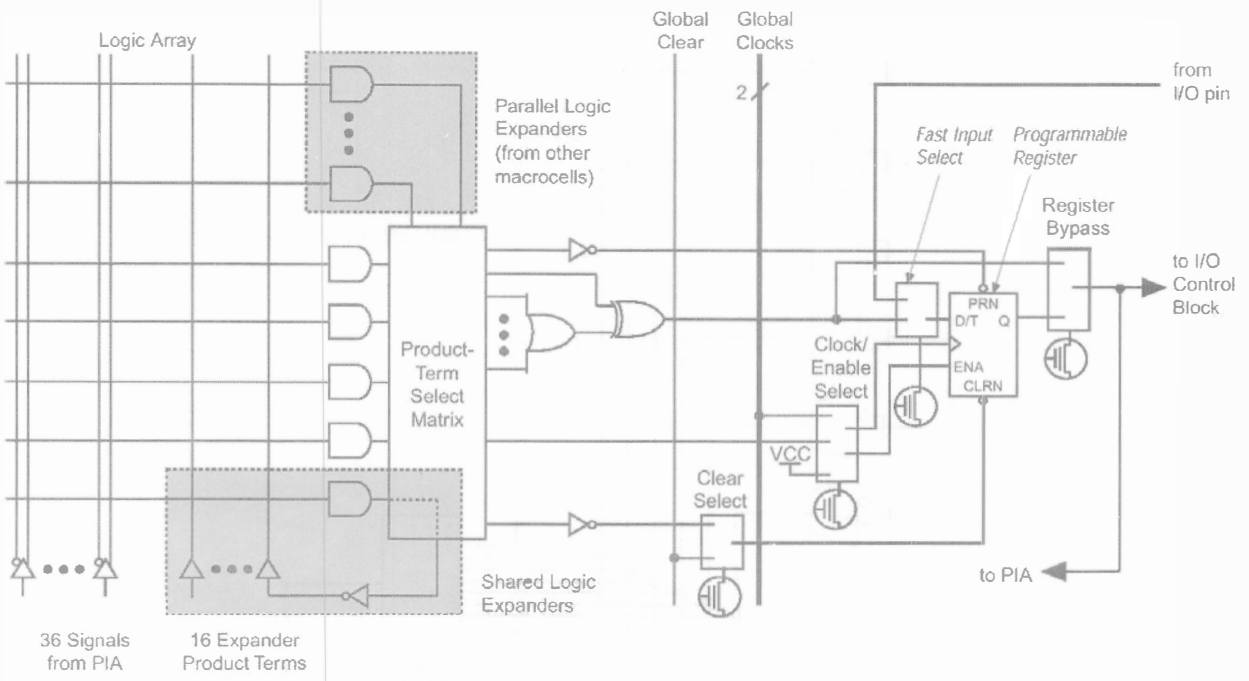


Figura C.2. Estructura de la macrocelda del MAX7000S.

El flip-flop de la macrocelda puede programarse como flip-flop tipo D, T, JK ó SR, con señal de reloj y cualquier combinación de señales PRESET, CLEAR y CLOCK ENABLE. Estas señales son controladas por la lógica secundaria.

Con respecto a la entrada y salida, cada macrocelda de un LAB está conectada a los bloques de control de entrada/salida. Este bloque está compuesto de un pin bidireccional controlado por un búfer tres estados, de manera que un sólo pin sirve como canal de entrada y/o de salida. El control del flujo de datos sobre estos pines lo ejecuta la lógica de las macroceldas.

Todos los MAX7000S tienen canales de entrada de alta velocidad hacia el registro de la macrocelda. A estos canales están conectados los pines de entrada/salida del dispositivo, de manera que el dato pasa directamente a la entrada D del flip-flop evitando pasar por la PIA ó por la lógica combinacional. El dato de una entrada configurada de esta forma tarda 2.5 nanosegundos en llegar al flip-flop.

Como se mencionó al inicio, el MAX7000S utiliza la tecnología EEPROM para almacenar la información de la programación. Dicha información permanecerá en el circuito hasta que éste sea programado nuevamente. La falta de energía en el circuito no implica pérdidas de información.

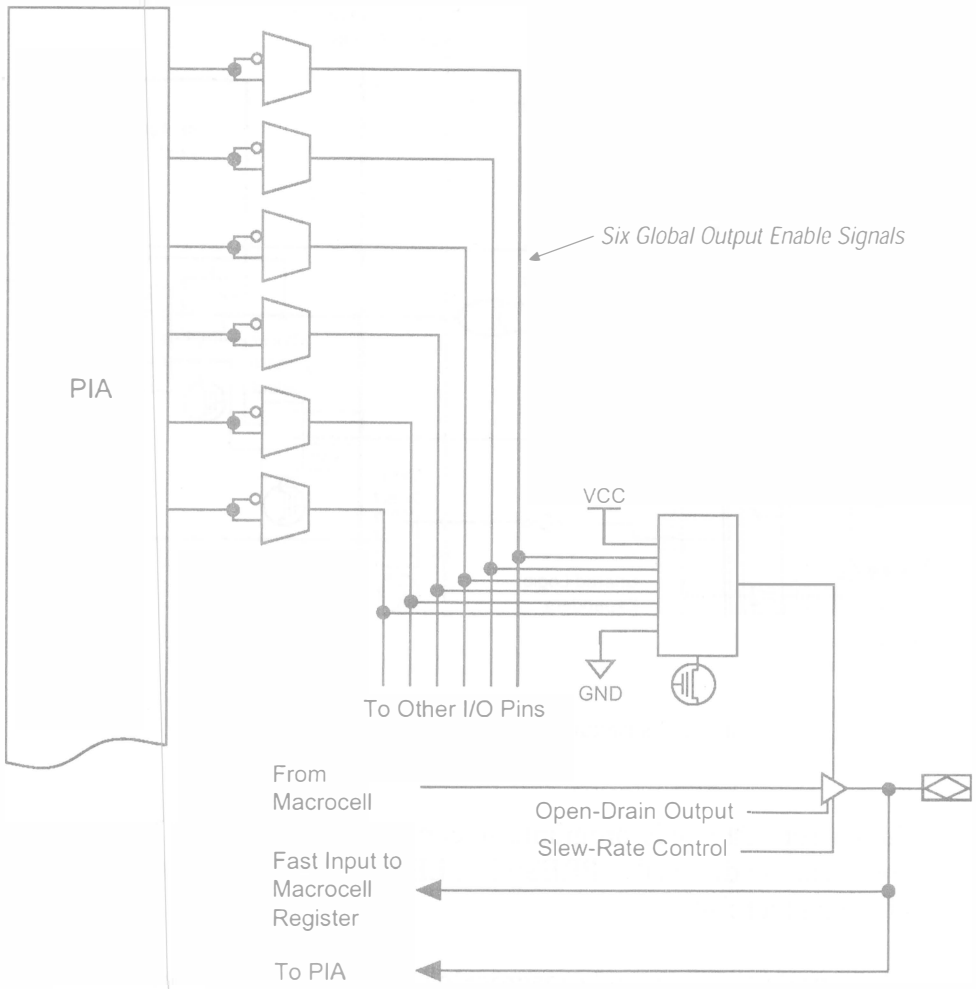


Figura C.3. Bloque de control de E/S del MAX7000S.

C.3 PLD FLEX10K DE ALTERA

Cada dispositivo FLEX10K está formado por una serie de matrices llamadas EABs (Embedded Array Blocks) que sirven para implantar memorias y funciones lógicas especiales. También consta de una serie de matrices lógicas llamadas LABs (Logic Array Blocks) para implantar lógica en general. La figura C.4 muestra el diagrama de bloques de la arquitectura del FLEX10K.

Cuando se crean memorias en los dispositivos FLEX10K cada EAB provee 2048 bits, los cuales permiten implantar memorias RAM, ROM y funciones FIFO (first-in first-out). Si sólo se implanta lógica, cada EAB contribuye de 100 a 600 compuertas que son utilizadas para construir funciones lógicas como multiplicadores, microcontroladores y máquinas de estados. Cada EAB puede utilizarse de manera independiente, o bien, combinarse para implantar funciones más complejas.

Por otra parte, cada LAB contiene ocho LEs (Logic Elements) y un canal local de interconexión. Un LE consiste de una tabla LUT (Look-Up Table) de cuatro entradas, un flip-flop programable y de dos canales dedicados: uno para implantar funciones en cascada y el otro para funciones con

acarreo de entrada. Los ocho LEs pueden emplearse para crear bloques lógicos de tamaño medio como contadores, decodificadores o máquinas de estados.

La interconexión entre las señales internas del dispositivo FLEX10K y los pines de entrada/salida es provista por un canal de interconexiones de alta velocidad (FastTrack Interconnect), una serie de canales de alta velocidad distribuidos en filas y columnas a lo largo y ancho del dispositivo. En cada fila y columna del canal de interconexiones de alta velocidad se localiza un elemento de entrada/salida (IOE). Un IOE está compuesto por un búfer bidireccional de entrada/salida y un flip-flop. Los dispositivos FLEX10K también constan de 6 canales de entrada dedicados al control de los flip-flops. Estos canales aseguran una eficiente distribución de la velocidad que se traduce en menor tiempo de respuesta del circuito.

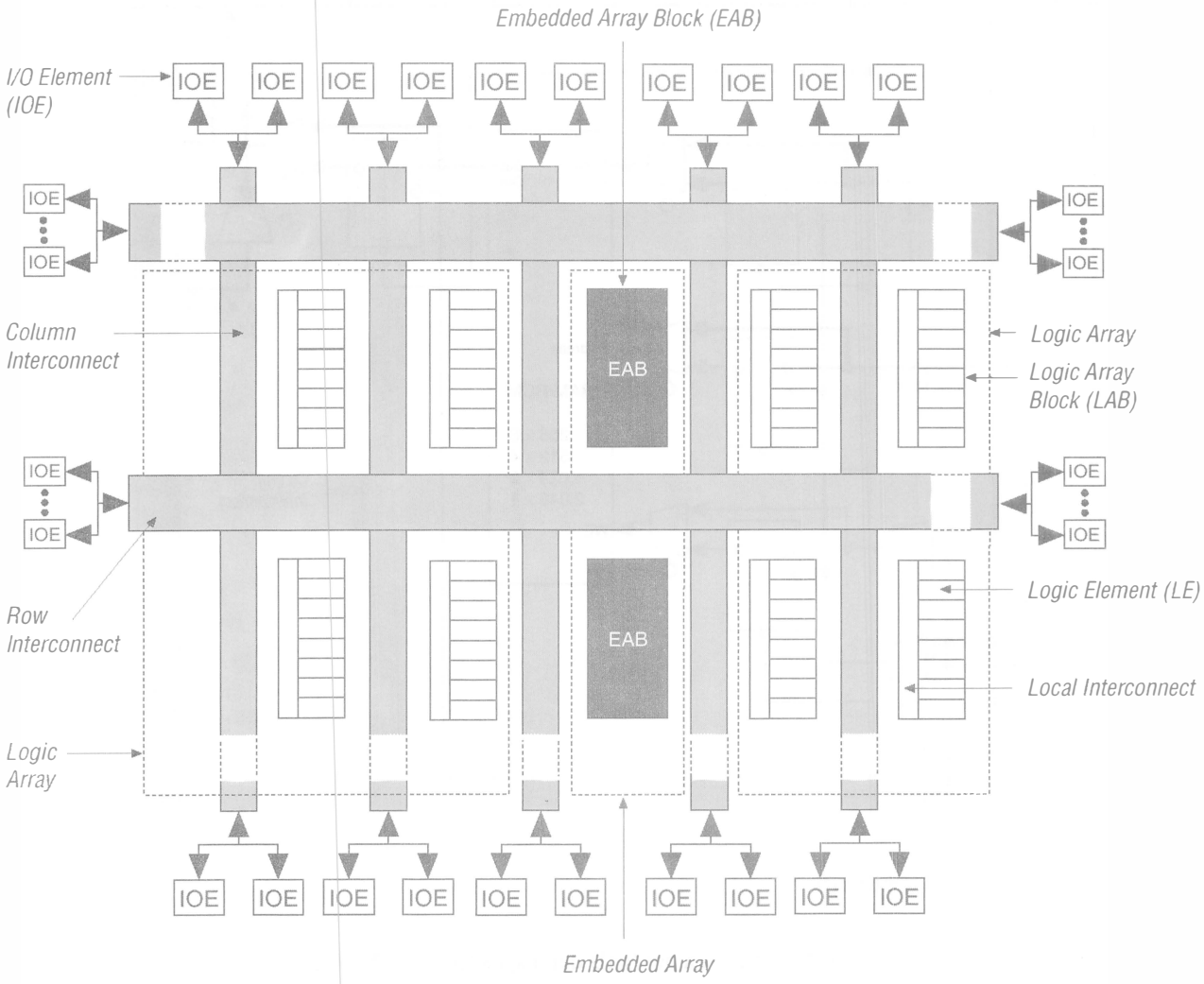


Figura C.4. Diagrama de bloques del dispositivo FLEX10K. Cada grupo de LEs está integrado en un LAB; los LABs están distribuidos en filas y columnas. Cada fila también contiene un EAB. Los LABs y los EABs están conectados mediante un canal de interconexiones de alta velocidad. Los IOEs se localizan en las terminales de este canal de interconexiones.

C.3.1 EAB (Embedded Array Block)

Un EAB consiste de un bloque de memoria RAM con registros flip-flops conectados a los puertos de entrada y salida del bloque de memoria. Los EABs también permiten implantar funciones lógicas, para ello, una tabla LUT es creada durante la configuración del dispositivo. En esta tabla son almacenados los resultados de la función combinacional, de manera que ya no son calculados los resultados sino consultados en la tabla.

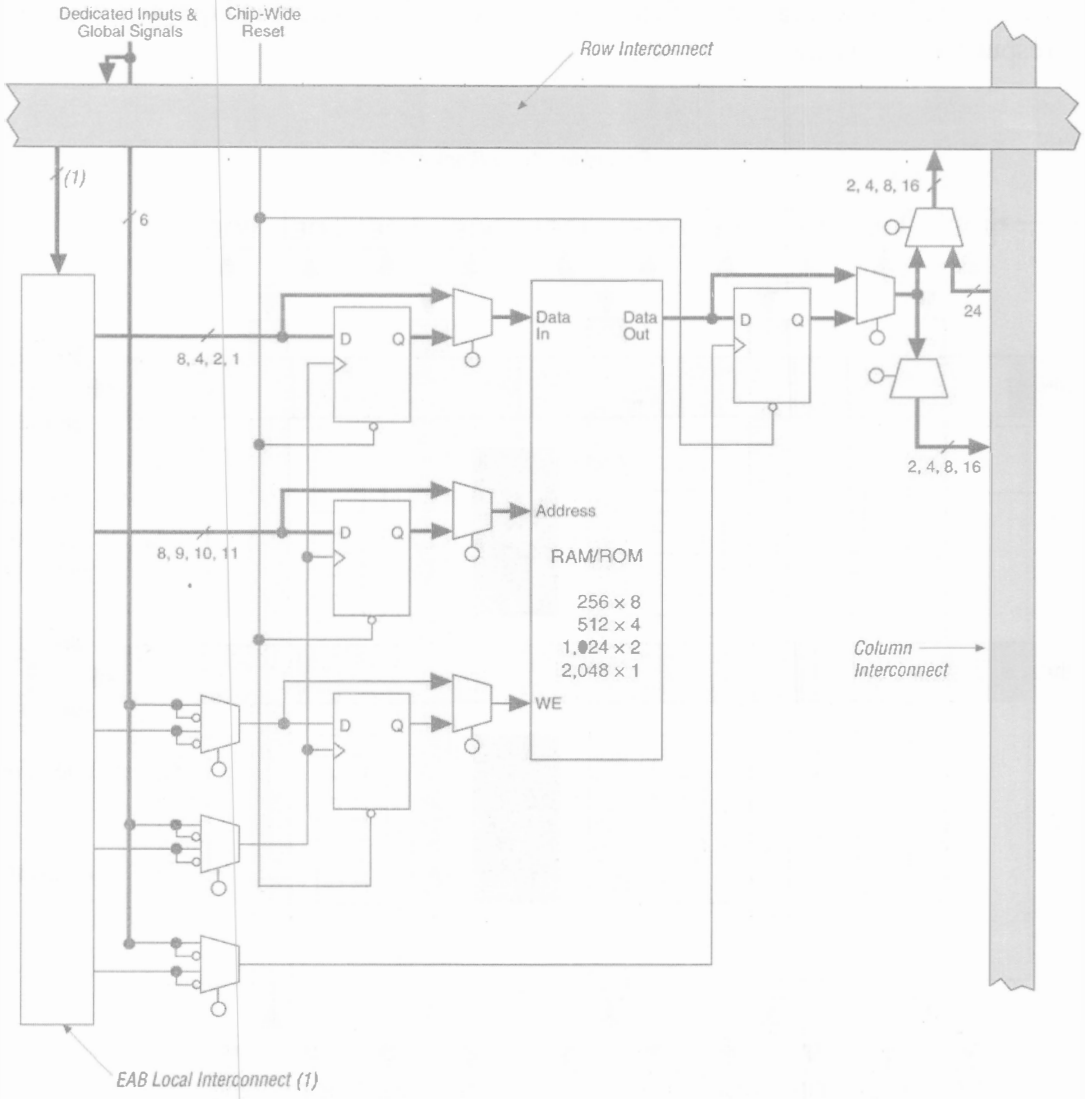


Figura C.5. Estructura del EAB del dispositivo FLEX10K.[21]

[21] (1) En los dispositivos EPF10K10, EPF10K10A, EPF10K20, EPF10K30, EPF10K30A, EPF10K40, EPF10K50 y EPF10K50V el bus de entrada hacia el canal local de interconexiones del EAB está compuesto de 22 líneas; por el contrario, el bus de entrada en los dispositivos EPF10K70, EPF10K100, EPF10K100A, EPF10K130V y EPF10K250A consta de 26 líneas.

Implantar funciones combinacionales en EABs incrementa el rendimiento del circuito, ya que los EABs tienen tiempos de acceso más rápidos que los LABs. Las funciones lógicas complejas también toman ventaja de los EABs pues los tiempos de respuesta son menores comparados con los tiempos de los LEs y de los bloques de RAM de los FPGAs. Esta ventaja se debe a una sencilla razón: los EABs están compuestos de un bloque de memoria RAM de mayor capacidad que elimina cualquier tiempo de retardo; en cambio, los FPGAs constan de bloques de RAM más pequeños, por lo tanto, cuando implantan funciones lógicas muy grandes requieren de mayor número de bloques y los tiempos de retardo para la comunicación entre bloques se incrementa.

Los EABs pueden implantar memorias RAM síncronas, las cuales son mucho más fáciles de utilizar que las RAM asíncronas. Esto se debe a que las RAM síncronas generan automáticamente la señal de habilitación de escritura en la memoria, en cambio, en las RAM asíncronas ésta señal debe generarse vía programación.

Cada EAB puede configurarse para crear memorias de las siguientes capacidades: 256x8, 512x4, 1024x2 y 2048x1. Bloques de RAM de mayor capacidad pueden crearse combinando múltiples EABs; por ejemplo, dos bloques de 256x8 pueden combinarse en un bloque de 256x16, o dos bloques de 512x4 en un bloque de 512x8. Si es necesario, todos los EABs en el dispositivo pueden conectarse en cascada para formar un sólo bloque de RAM. Este bloque de RAM puede estar compuesto de hasta 2048 palabras sin que los tiempos de acceso a la memoria se vean afectados.

Cada EAB es alimentado por una fila de interconexiones y puede direccionar su salida hacia una fila o una columna de interconexiones.

C.3.2 LAB (Logic Array Block)

Un LAB está compuesto de ocho LEs, dos eslabones para implantar funciones en cascada y funciones con acarreos, señales de control del LAB y el canal local de interconexiones del LAB. Además, cada LAB cuenta con la infraestructura suficiente para crear rutas eficientes de comunicación entre LABs, con óptima utilización del dispositivo y alto rendimiento.

Cada LAB proporciona cuatro señales de control con inversión programable que se utilizan como señales de control para los flip-flops de los LEs. Dos de estas señales sirven como relojes, mientras que las otras dos son señales de CLEAR y PRESET. La figura C.6 muestra la estructura de un LAB.

C.3.3 LE (Logic Element)

El LE es la unidad lógica más pequeña en la arquitectura FLEX10K. Cada LE está compuesto de una tabla LUT de cuatro entradas, la cual es un generador de funciones que calcula rápidamente cualquier función de cuatro variables. Además, cada LE contiene un flip-flop programable con señal de habilitación síncrona, un eslabón para funciones en cascada y uno para funciones con acarreo. Todos los LEs están conectados al canal de interconexiones de alta velocidad y a su correspondiente canal local de interconexiones. La figura C.7 muestra la estructura de un LE.

El flip-flop programable de los LEs puede configurarse como tipo D, T, JK ó SR. Las señales de reloj, CLEAR y PRESET para el flip-flop pueden provenir de las señales globales, de los pines de entrada/salida de propósito general ó de la lógica interna. Para funciones combinacionales, el flip-flop es evitado y la salida de la tabla LUT es asignada directamente a la salida del LE.

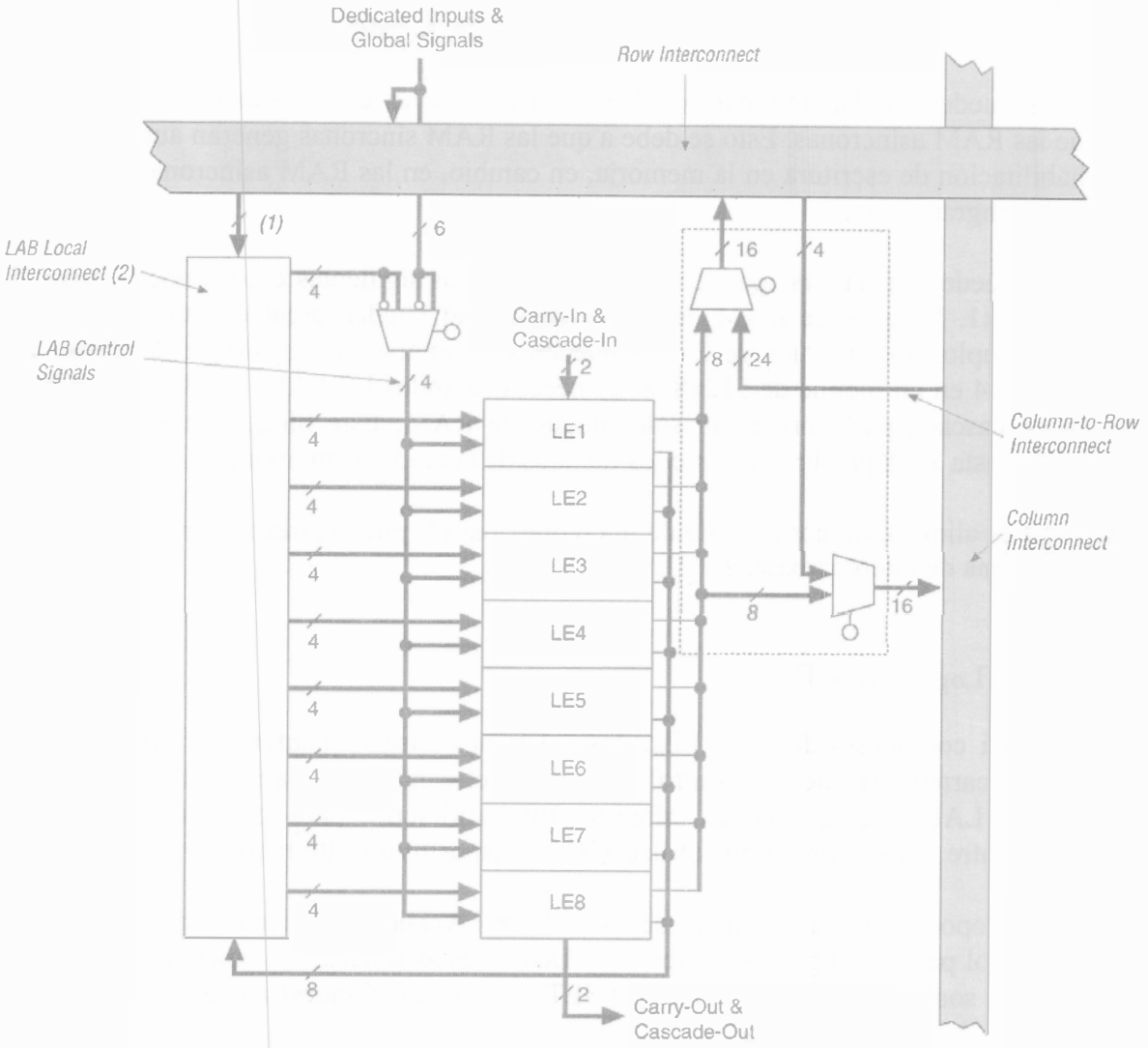


Figura C.6. Estructura del LAB del dispositivo FLEX10K.[22]

[22] (1) En los dispositivos EPF10K10, EPF10K10A, EPF10K20, EPF10K30, EPF10K30A, EPF10K40, EPF10K50 y EPF10K50V el bus de entrada hacia el canal local de interconexiones del LAB consta de 22 líneas; por el contrario, el bus de entrada en los dispositivos EPF10K70, EPF10K100, EPF10K100A, EPF10K130V y EPF10K250A consta de 26 líneas.

(2) En los dispositivos EPF10K10, EPF10K10A, EPF10K20, EPF10K30, EPF10K30A, EPF10K40, EPF10K50 y EPF10K50V el canal local de interconexiones consta de 30 líneas; por el contrario, en los dispositivos EPF10K70, EPF10K100, EPF10K100A, EPF10K130V y EPF10K250A el canal consta de 34 líneas.

El LE tiene dos salidas, una va al canal local de interconexiones y la otra va hacia una fila o una columna del canal de interconexiones de alta velocidad. Ambas salidas son controladas de manera independiente, de manera que la tabla LUT puede emplear una salida y el flip-flop otra. Esta característica hace uso de toda la capacidad del LE, ya que permite utilizar la tabla LUT y el flip-flop en funciones totalmente independientes.

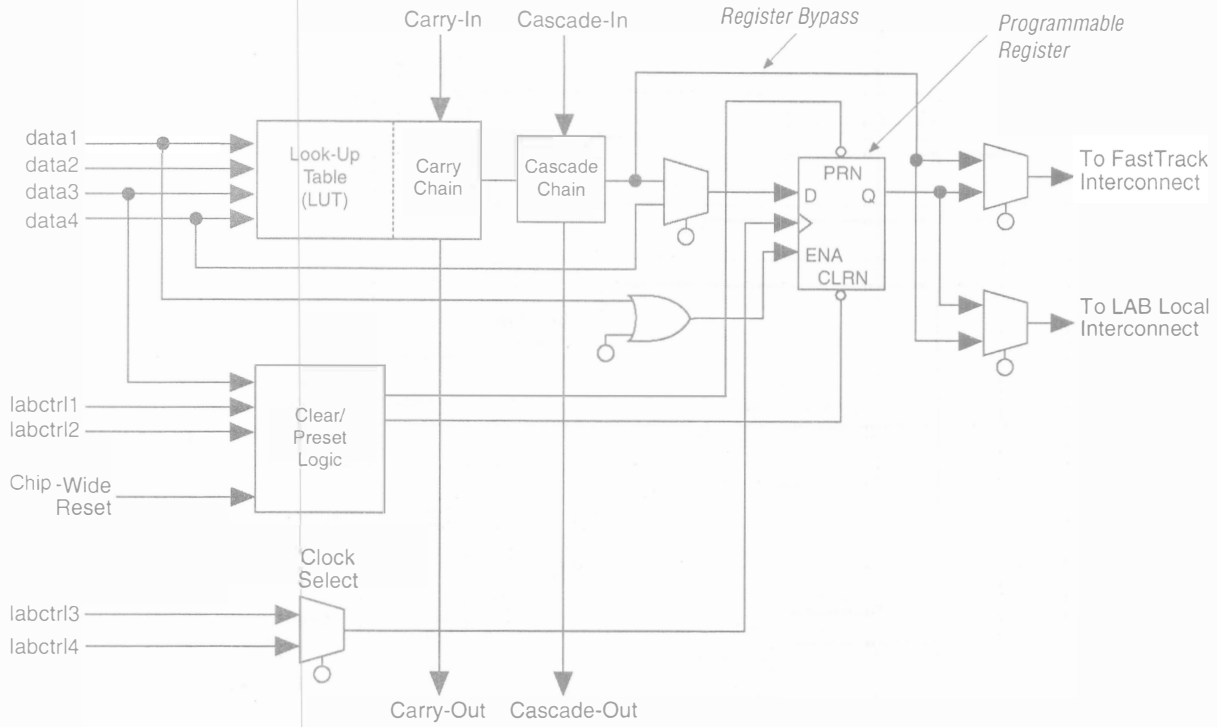


Figura C.7. Estructura del elemento lógico (LE).

La arquitectura del FLEX10K proporciona dos canales de datos de alta velocidad para conectar LEs adyacentes sin emplear el canal local de interconexiones. Estos canales son el eslabón de cascada y el eslabón de acarreo. El eslabón de acarreo soporta contadores y sumadores de alta velocidad, y el eslabón de cascada permite implantar funciones de múltiples entradas con el mínimo retardo. Ambos eslabones están conectados a todos los LEs del LAB y a todos los LABs de una misma fila. El uso excesivo de estos eslabones impacta en la velocidad del circuito.

C.3.4 IOE: Input/Output Element

Un IOE contiene un búfer bidireccional y un registro que puede emplearse como registro de entrada o como registro de salida para datos que requieran alto rendimiento. En ocasiones, utilizar un LE como registro de entrada es mucho más rápido que usar un registro IOE. Los IOEs pueden utilizarse como pines de entrada, de salida o bidireccionales. Si se implanta un pin bidireccional, el registro de salida se coloca en el IOE, y el dato de entrada junto con la señal de habilitación del

registro se colocan en el LE adyacente al pin bidireccional. La figura C.8 muestra los registros de entrada/salida bidireccionales.

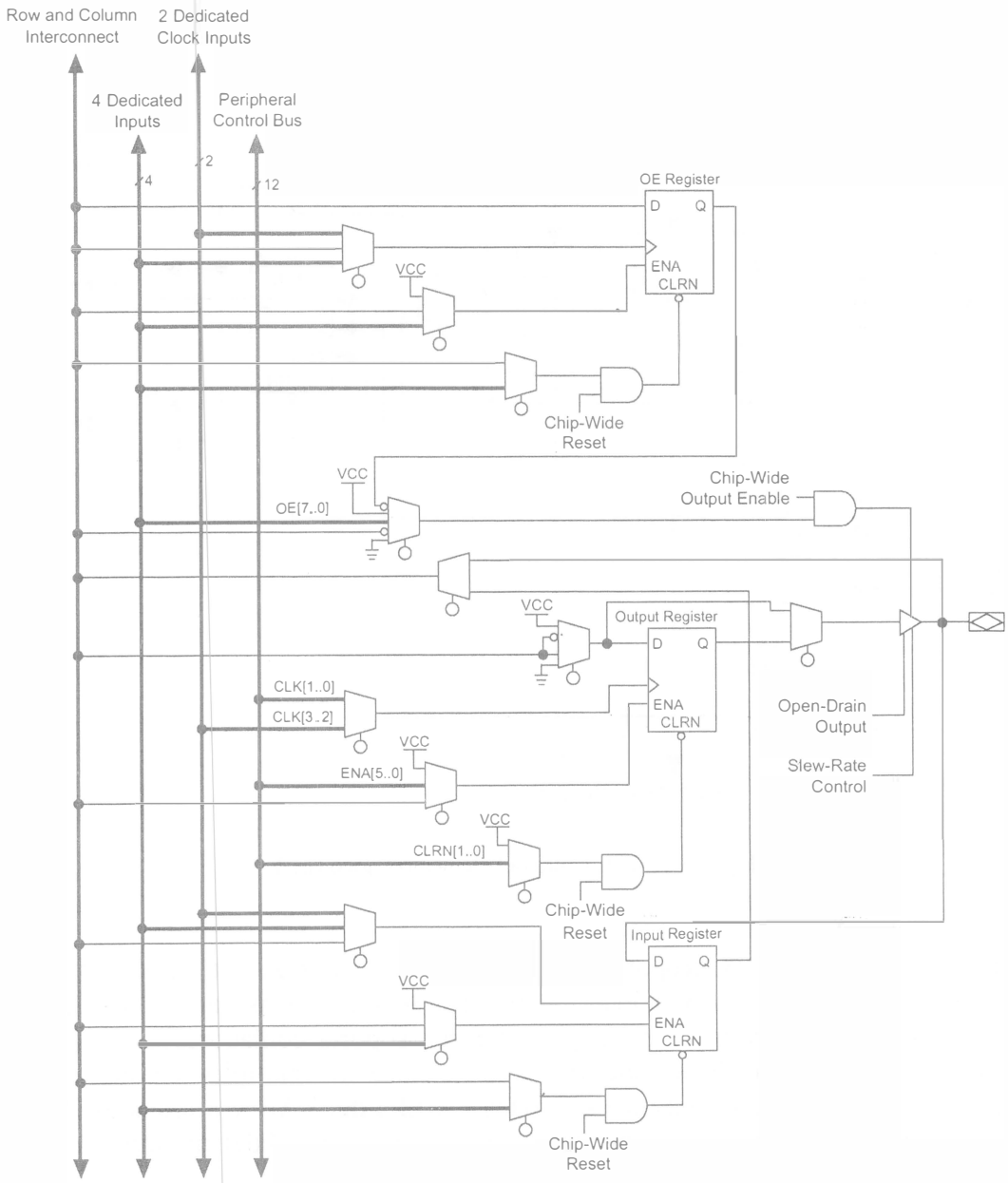


Figura C.8. Estructura del registro de entrada/salida bidireccional.

C.4 FPGA XC4000 DE XILINX

La estructura de este dispositivo se muestra en la siguiente figura.

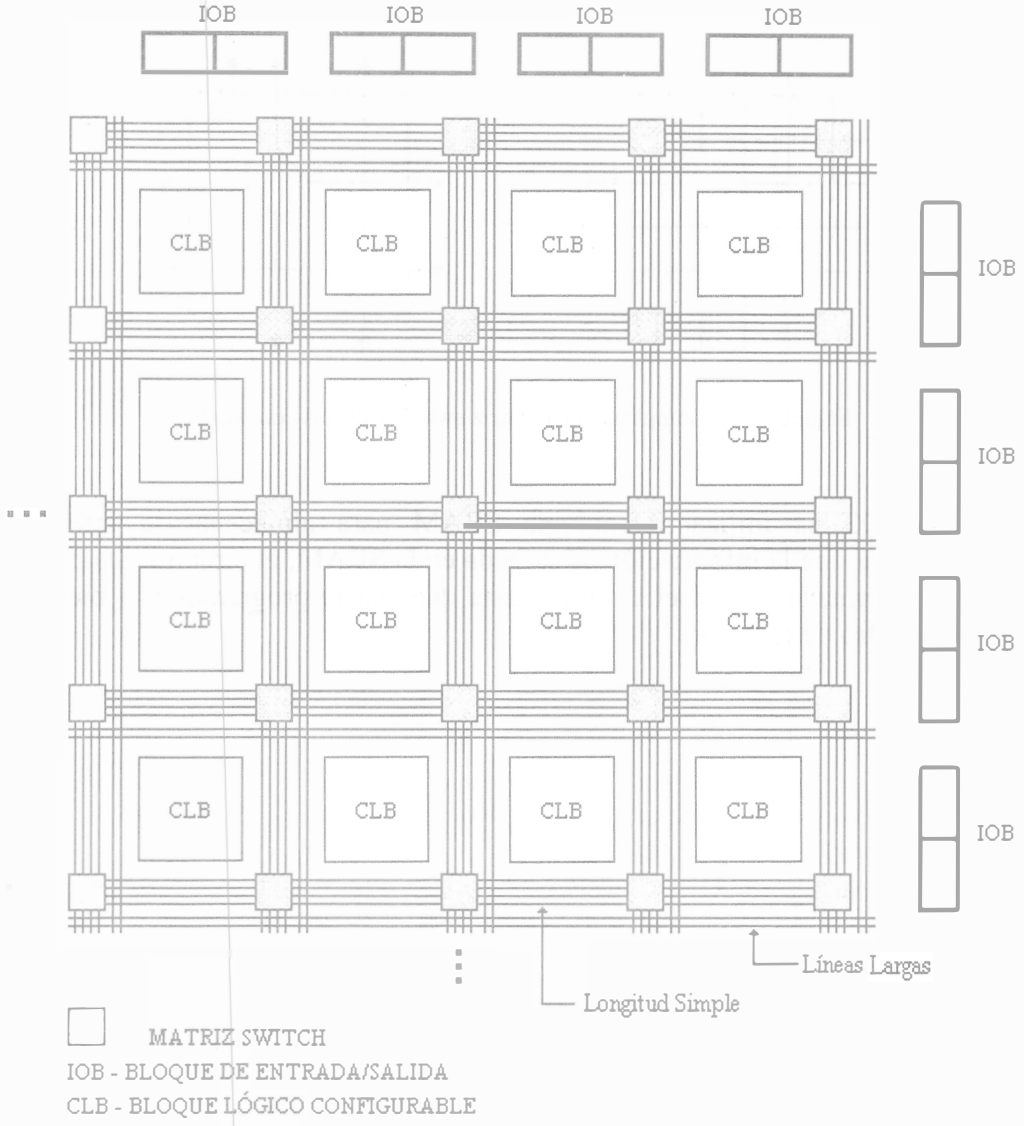


Figura C.9. Estructura del FPGA XC4000 de Xilinx.

La lógica de este FPGA se implanta en una matriz de bloques lógicos programables denominados CLBs (Configurable Logic Blocks). Las líneas de entrada y salida de esta matriz son controladas por medio de los bloques de entrada/salida (IOBs) colocados en los bordes de la matriz.

Los CLBs y los IOBs están interconectados por una gran cantidad de estructuras de interconexiones programables llamadas matrices switch, que pueden programarse para conectar un bloque con otro. Las matrices switch funcionan de manera similar a los cruces de vías de los trenes.

Si el interruptor está 'apagado', el tren continúa su recorrido por la misma vía, de lo contrario, se desvía hacia otro camino. En las matrices switch el interruptor es un transistor, si éste está 'encendido', el camino de los datos está activado, es decir, los datos pueden fluir de un bloque a otro. En caso contrario, el transistor está 'apagado' y no existe camino por donde los datos puedan circular.

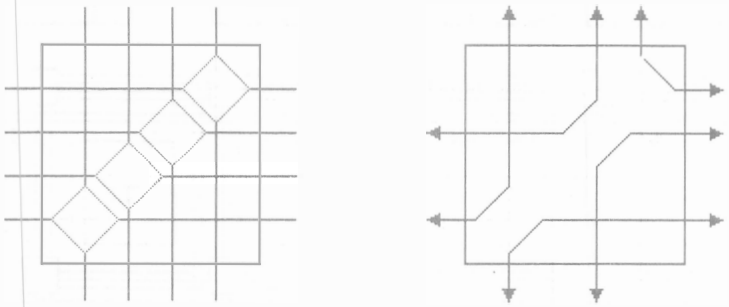


Figura C.10. Ejemplo de una matriz switch y de sus conexiones.

Los dispositivos Xilinx utilizan tecnología SRAM para almacenar la información de la programación. Una vez que la información es cargada en la SRAM, el circuito está listo para operar. Esta configuración permanecerá en el circuito hasta que sea re-programado, o bien, hasta que se interrumpa el suministro de energía.

GLOSARIO

AHDL. Abreviación de Altera Hardware Description Language. Es un lenguaje de descripción de hardware para la programación de PLDs.

Almacenamiento. Capacidad de los dispositivos digitales para guardar y retener información. Proceso de conservación de datos para su uso posterior.

Alta Impedancia. El estado de alta impedancia en un circuito tres estados es aquel en el que la salida se desconecta del resto del circuito.

ALU. Abreviación de Arithmetic Logic Unit/Unidad Lógico Aritmética. Es la parte de la computadora encargada de ejecutar todas las operaciones lógicas y aritméticas. En la actualidad, los procesadores suelen estar compuestos de dos ALU's, una para punto fijo y otra para punto flotante.

AND. Operación lógica básica en la que se obtiene un valor verdadero sólo si todos los operandos son verdaderos, y se obtiene un valor falso para cualquier otro caso.

Arquitectura. Disposición funcional interna de los elementos que dotan a un dispositivo de sus características particulares.

Binario. Que tiene dos valores o estados. Describe un sistema de numeración en el cual hay sólo dos posibles valores para cada dígito: cero y uno.

Bit de Signo. Corresponde al bit de más a la izquierda de un número en formato binario. Si la aritmética que maneja la ALU es signada, entonces este bit designa si el número es positivo (0) ó negativo (1).

Bit. Es la abreviación para dígito binario. Un bit puede presentar el valor de cero ó uno.

Bus. Conjunto de interconexiones que establece la interfaz entre los componentes de un sistema digital.

Byte. Conjunto de 8 bits.

Circuito Integrado. Un tipo de circuito en el que todos sus componentes se encuentran integrados en un único chip semiconductor de tamaño muy pequeño.

Circuito Tres Estados. Un tipo de circuito que posee tres estados de salida: alto, bajo y alta impedancia (abierto/desconectado).

Código. Conjunto de bits ordenados según cierto patrón. Los códigos son utilizados para representar información tales como números, letras y otros símbolos.

Compilador. Es un programa que procesa y traduce las expresiones escritas en un lenguaje de programación en particular a otro lenguaje, generalmente, a lenguaje de máquina.

Complemento a uno. Es el inverso u opuesto de un número. El complemento de uno es cero y de cero es uno. En álgebra booleana es la función inversa, la cual se expresa mediante una barra encima de la variable.

CPU. Abreviación de Central Processing Unit/Unidad Central de Procesamiento. El CPU es el cerebro de toda computadora, en él tienen lugar la mayoría de los cálculos que una computadora lleva a cabo.

Dato. Información en formato numérico, alfabético o de cualquier otro tipo.

Desbordamiento. Overflow. Condición que ocurre cuando una computadora intenta manipular un número cuya representación es demasiado grande. Todas las computadoras tienen definido el rango de valores que pueden representar; si durante la ejecución de

una instrucción se obtiene un número fuera de ese rango, entonces se experimentará una condición de desbordamiento.

Diagrama de Estados. Representación gráfica de una secuencia de estados.

Diagrama de Tiempos. Gráfico de formas de onda digitales que muestra la relación en el tiempo entre las señales de entrada y las señales de salida de un sistema digital.

Digital. Describe cualquier sistema basado en datos o eventos discretos.

Dirección. Posición de una determinada celda de almacenamiento o grupo de celdas.

Expresión Booleana. Expresión algebraica utilizada para analizar el funcionamiento de los circuitos lógicos.

Ciclo Fetch. Proceso de la CPU en el que se obtiene una instrucción de la memoria.

Flip-flop Tipo D. Un tipo de multivibrador biestable en el que la salida sigue al estado de la entrada D.

Flip-flop. Circuito de almacenamiento que puede almacenar sólo un bit a un tiempo; dispositivo biestable síncrono.

Glitch (Transitorio). Pico de voltaje o de corriente de corta duración, no deseado y generalmente producido de forma no intencionada.

HDL. Abreviación de Hardware Description Language/Lenguaje de Descripción de Hardware.

Hexadecimal. Describe un sistema de numeración en base 16.

Hoja de Especificaciones. Documento que establece los valores de los parámetros y las condiciones de funcionamiento de un circuito integrado o de otro dispositivo.

Interrupción. Petición de servicios por parte de un dispositivo periférico.

Inversor (NOT). Circuito que cambia un valor verdadero a falso, y viceversa.

Latch. Dispositivo digital biestable utilizado para almacenar un bit.

Lenguaje de Programación. Es un conjunto de palabras y reglas gramaticales que instruyen a una computadora en la ejecución de tareas específicas. El término lenguaje de programación usualmente se refiere a lenguajes de alto nivel como C, C++, Ada, etc., en nuestro caso, se refiere a VHDL, Verilog HDL y AHDL.

Lenguaje Ensamblador. Es un tipo de lenguaje de programación de computadoras en el que el código se expresa mediante mnemónicos, los cuales representan a las instrucciones.

Lenguaje Máquina. Lenguaje de programación de muy bajo nivel entendible por las computadoras. Las instrucciones escritas en este tipo de lenguaje se representan con códigos binarios (ceros y unos).

LSB. Abreviación de Least Significant Bit/Bit Menos Significativo. Corresponde al bit de más a la derecha de un número en formato binario.

Mapa de Karnaugh. Disposición de celdas que representa la combinación de literales de una expresión booleana y que se utiliza para la simplificación sistemática (minimización) de la expresión.

Microprocesador. Dispositivo VLSI ó ULSI que puede ser programado para realizar operaciones lógicas y aritméticas. También permite procesar datos de la manera como se le especifique.

Minimización. Proceso por el que se obtiene una expresión en forma de suma de productos

o de producto de sumas. Esta expresión se caracteriza por contener el menor número posible de términos con el menor número posible de literales por término.

Mnemónico. En lenguajes de programación de computadoras es la representación taquigráfica de una instrucción.

MSB. Abreviación de Most Significant Bit/Bit Más Significativo. Corresponde al bit de más a la izquierda de un número en formato binario.

Multivibrador. Circuito digital en el que la salida se conecta nuevamente a la entrada. Dependiendo de la configuración del multivibrador es posible producir dos estados estables, un único estado o ningún estado.

NAND. Operación lógica en la que se obtiene un valor falso si todos los operandos son verdaderos.

NOR. Operación lógica en la que se obtiene un valor falso si al menos uno de los operandos es verdadero.

NOR-Exclusiva (XNOR). Operación lógica en la que se obtiene un valor falso si los dos operandos presentan niveles lógicos opuestos. Si los dos operandos presentan niveles lógicos iguales se obtiene un valor verdadero.

OR. Operación lógica básica en la que se obtiene un valor verdadero si al menos uno de los operandos es verdadero, y se obtiene un valor falso para cualquier otro caso.

OR-Exclusiva (XOR). Operación lógica que regresa un valor verdadero cuando los dos operandos tienen niveles lógicos opuestos.

Paralelo. En los circuitos digitales, datos que se producen simultáneamente a través de varias líneas. Transferencia o procesamiento de varios bits simultáneamente.

PLD. Abreviación de Programmable Logic Device/Dispositivo Lógico Programable. Es un circuito integrado que puede ser reprogramado para ejecutar diversas funciones complejas. Un PLD consite de un arreglo de compuertas AND y OR.

Producto de Sumas. Expresión booleana que consiste en multiplicar términos suma.

Registro. Circuito digital capaz de almacenar y desplazar información; típicamente utilizado como dispositivo de almacenamiento temporal.

Reloj. Señal de temporización de un sistema digital.

Segmentación encauzada. Técnica que permite a los microprocesadores trabajar con más de una instrucción a la vez.

Suma de Productos. Expresión booleana que consiste en sumar términos que contienen productos.

Término Producto. Producto booleano de dos o más literales. Equivale a una operación AND.

Término Suma. Suma booleana de dos o más literales. Equivale a una operación OR.

ULSI. Abreviación de Ultra Large Scale Integration/Integración a Ultra Gran Escala. Hace referencia a un nivel de complejidad en los circuitos integrados; los circuitos integrados pertenecientes a esta clasificación cuentan con más de 100,000 compuertas lógicas por chip.

Verilog HDL. Abreviación de Verilog Hardware Description Language. Lenguaje de descripción de hardware para la programación de PLDs.

VHDL. Abreviación de Very High Speed Integrated Circuit Hardware Description

Language. Lenguaje de descripción de hardware para la programación de PLDs.

VLSI. Abreviación de Very Large Scale Integration/Integración a muy Gran Escala. Hace referencia a un nivel de complejidad en los circuitos integrados; los circuitos integrados pertenecientes a esta clasificación presentan entre 10,000 y 99,999 compuertas lógicas por chip.

BIBLIOGRAFÍA

Advanced micro devices
Bipolar microprocessor logic and interface data book
Advanced Micro Devices, Inc. U.S.A., 1985.

Advanced micro devices
PAL device data book bipolar and CMOS
Advanced Micro Devices, Inc. U.S.A., 1990.

Altera Corporation
FLEX 10K embedded programmable logic device family data sheet

Altera Corporation
MAX7000 programmable logic device family data sheet

Altera Corporation
User configurable logic data book
Altera Corporation. Santa Clara, 1988.

Coelho, David R.
The VHDL handbook
Kluwer Academic Publishers. Boston, 1989.

Downs, Thomas and Schulz, Mark F.
Logic design with Pascal: Computer-aided design techniques
Van Nostrand Reinhold. New York, 1988.

Floyd, Thomas L.
Fundamentos de sistemas digitales
Prentice-Hall. Madrid, 1997.

Hayes, John P.
Diseño de sistemas digitales y microprocesadores
McGraw-Hill, Inc. Madrid, 1984.

Heath, Steve
Microprocessor architectures: RISC, CISC and DSP
Butterworth-Heinemann Ltd. Oxford, 1995.

Hennessy, John L. and Patterson David A.
Computer organization & design: The hardware/software interface
Morgan Kaufmann Publishers, Inc. San Francisco, 1994.

Horvath, Ralph.
Introduction to microprocessors using the MC6809 or the MC68000
McGraw-Hill, Inc. Singapore, 1992.

Lynch, Michel A.
Microprogrammed state machine design
CRC Press. Boca Raton, 1993.

Mano, Morris M.
Computer engineering hardware design
Prentice-Hall. Englewood Cliffs, 1988.

Mano, Morris M.
Computer system architecture
Prentice-Hall. Englewood Cliffs, 1982.

Mick, J. and Brick, J.
Bit-slice microprocessor design
McGraw-Hill. New York, 1980.

Motorola HC11
M68HC11 E series technical data
Motorola, Inc. U.S.A., 1995.

Motorola HC11
M68HC11 reference manual
Motorola, Inc. U.S.A., 1991.

Tanenbaum, A. S.
Structured computer organization
Prentice-Hall. Englewood Cliffs, 1981.

ÍNDICE

A

Arquitecturas
de von Neumann · 2
Hardvard · 5
Microprocesador 68HC11 · 84
MIMD · 3, 4
Segmentación del 68HC11 · 141
SIMD · 3
SISD · 2
Arquitecturas MIMD
Características · 7
Arquitecturas SIMD
Características · 6

B

Banderas · 78

C

Cartas ASM
Cartas ASM y secuenciadores · 59
Diferencia entre tipos de salidas · 19
Notación · 13
Representación de decisiones · 13
Representación de estados · 13
Representación de salidas · 14
Circuitos secuenciales · 26
Método de diseño para circuitos secuenciales · 29
Computadora
Computadoras paralelas · 3, 6
Computadoras secuenciales · 2
Controlador de la computadora · 5
Definición · 2
Estructura general de una computadora · 2
Estructura interna básica de una computadora · 65
Tipos de computadoras · 2
Control de riesgos por dependecias de datos · 196
Por medio de anticipaciones · 205
Por medio de detenciones · 198
Por medio de software · 197
Control de riesgos por saltos
Por medio de detenciones · 219
Suponer que el salto no es realizado · 220
CPLDs · 327
Estructura del MAX7000S · 327

D

Diagramas de estados · 12
Direccionamiento entrada-estado · 43
Direccionamiento implícito · 47
Direccionamiento por trayectoria · 41

F

FLEX10K
EABs · 332
IOEs · 335
LABs · 333
LEs · 333
Flip-flop · 28
Flip-flop tipo D · 28
FPGAs · 327
Estructura del XC4000 · 337

I

Instrucciones para el 68HC11 segmentado · 144
ABA (acceso inherente) · 151
ANDB (acceso extendido) · 158
ASL acceso indexado · 164
BRA (acceso relativo) · 178
LDAA (acceso inmediato) · 144
Resumen de instrucciones · 185
STAA (acceso extendido) · 171

L

Latch tipo D · 27
Lógica de selección
Implantación en Verilog · 309

M

Mapas de Karnaugh · 26, 31
Máquina de estados
  Algoritmo de la máquina de estados · 12
  Modelo general · 12
MAX+PLUS II
  Editor de formas de onda · 240
  Editor de texto · 238
  Editor gráfico · 244
  Entorno de trabajo · 236
Memoria de microprograma
  Implantación en AHDL · 313
Microprocesador 68HC11
  Instrucciones de acceso directo · 88
  Instrucciones de acceso extendido · 87
  Instrucciones de acceso indexado · 88
  Instrucciones de acceso inherente · 89
  Instrucciones de acceso inmediato · 87
  Instrucciones de acceso relativo · 88
  Líneas de control · 86
  Modos de direccionamiento · 87
  Pipeline · 130
Microprogramación
  Atención a interrupciones · 120
  Instrucción BEQ · 114
  Instrucción BRA · 110
  Instrucción INX · 90
  Instrucción INY · 102
  Instrucción JSR · 116
  Instrucción LDAA · 107
  Instrucción LDAB · 106
  Instrucción RTI · 123
  Instrucción RTS · 118
  Instrucción SUBA · 108
  Instrucción XGDX · 105
Multiplicación
  Algoritmo · 72

O

OR-Exclusiva · 18

P

PLDs
  Diseño utilizando PLDs · 33
  Estructura del FLEX10K · 330
  Lenguajes de descripción de hardware · 33
  Ventajas · 37

R

Registro acumulador
  Implantación en Verilog · 272
Registro contador de 16 bits
  Implantación en Verilog · 277
Registro de banderas
  Diagrama · 78
  Implantación en Verilog · 282
Registro de instrucción
  Implantación en Verilog · 306
Registros internos
  Acumuladores · 71
  Registros contadores de 16 bits · 76
Riesgos por dependecias de datos · 194
Riesgos por saltos
  Concepto · 215

S

Secuenciador
  Cartas ASM y secuenciadores · 59
  Concepto · 54
  Implantación en Verilog · 307
  Instrucciones del secuenciador · 55
  Secuenciadores y memorias · 58
Segmentación del 68HC11
  Arquitectura segmentada del 68HC11 · 141
  Ejecución de múltiples instrucciones · 186
  Etapa de decodificación · 134
  Etapa de ejecución · 138
  Etapa de lectura de instrucción · 131
  Etapa de post-escritura · 141
  Formato de instrucción · 131
  Módulo de registros internos · 211
Segmentación encauzada · 4
  Anticipación de datos · 205
  Control de riesgos por dependecias de datos · 196
  Definición · 129
  Detenciones · 199
  Detenciones debido a accesos múltiples a memoria · 203
  Representación gráfica de la segmentación encauzada · 143

Riesgos por dependecias de datos · 194
Riesgos por saltos · 215

U

Unidad básica de almacenamiento · 26
Unidad de control de interrupciones
Diagrama · 80
Implantación en Verilog · 302
Unidad de control de la computadora
Diagrama · 66
Implantación en Verilog · 304
Unidad de control de programa
Diagrama · 77
Unidad de procesos aritméticos
Diagrama · 67
Implantación en Verilog · 287
Unidad lógico aritmética
Implantación en Verilog · 291

V

Verilog HDL
Asignaciones continuas · 260
Bloques procedurales · 248
Construcciones always · 260
Contadores · 263
Declaración de módulos · 247
Eventos · 249
Funciones lógicas · 266
Funciones parametrizadas · 269
Latches · 264
Máquinas de estados · 264
Memorias · 270
Multiplexores · 261
Palabras reservadas · 259
Precedencia de operadores · 258
Puertos de entrada/salida · 248
Registros · 262
Sentencias de control de programa · 249
Tipos de datos · 252
Tipos de operadores · 255