SISTEMAS DIGITALES

C. Baena, J.I. Escudero, I. Gómez y M. Valencia

UNIVERSIDAD DE SEVILLA
DEPARTAMENTO DE TECNOLOGÍA ELECTRÓNICA

Carmen Baena

José Ignacio Escudero

Isabel Gómez

Manuel Valencia

Sistemas Digitales

Departamento de Tecnología Electrónica
ETS Ingeniería Informática. Universidad de Sevilla
Avda. Reina Mercedes, s/n
41012 Sevilla
Tfos. + 34 954 552 785, + 34 954 556 159. Fax: +34 954 552 764

Primera edición. 1997 (Nº TESP - 9702 -012) Revisión actual: 2008

Este documento es una actualización de la primera edición de 1997 de:

Circuitos y Sistemas Digitales I
Parte 4: Sistemas Digitales

Nº TESP - 9702 - 012

Prólogo (actualizando la ed. 1997)

En los estudios de Informática en la Universidad de Sevilla, inicialmente en la asignatura *Circuitos y Sistemas Digitales I* y, tras el cambio de los planes de estudio de 1997, en las asignaturas *Estructura de Computadores* (de I.I. y de ITIS) y *Estructura y Tecnología de Computadores 2* (de ITIG), se desarrolla la materia sobre **sistemas digitales a nivel RT** y su aplicación al **diseño de computadores**, necesariamente simples. En 1997 no conocíamos la existencia de ningún texto en el que se pudiera estudiar adecuadamente esta materia. Esto nos condujo, en dicha fecha, a desarrollar este "libro", con el principal objetivo de disponer de un texto donde estudiar y aprender esta interesante e importante materia en la que, además de establecer metodologías y realizar diseños complejos, se tiende el puente entre el *hardware* y el *software*.

En los tres capítulos del libro los autores intentaron aportar su dilatada experiencia como profesores de las mencionadas asignaturas. No obstante, aunque el libro lo firmaron sólo cuatro autores, no hubiera sido posible sin la aportación y colaboración de los restantes profesores: así que, muchas gracias a Manolo Bellido, Alberto Molina, Pilar Parra y Paco Pérez.

Desde que se dio por finalizada la primera edición de 1997, el libro ha sido puesto a libre disposición de los alumnos en la copistería del centro todos los cursos, sin más coste que el propio de su reproducción en fotocopias. En los más de 10 años transcurridos, centenares de alumnos del primer curso de cualquiera de las titulaciones de Informática han estudiado dichas materias en este libro, localizando varias erratas.

En la presente edición no sólo se pretende corregir los fallos detectados sino, sobre todo, actualizar los contenidos con lo aportado durante el periodo de tiempo transcurrido. No obstante, hemos deseado mantener la estructura original, añadiendo anexos como forma de incluir los nuevos contenidos. El original de 1997 había sido creado con diferentes procesadores de texto, cuya traslación de uno a otro no es directa. En esta edición, pues, ha sido necesario volver a editar buena parte de la materia en el nuevo procesador.

El presente libro expone los aspectos teóricos de los sistemas digitales a nivel RT. Aunque contiene algunos ejercicios, su estudio debe completarse con la realización de problemas (cuyos enunciados no se incluyen en esta obra).

Al estudiar esta nueva edición del libro posiblemente que se detectarán erratas, partes desarrolladas defectuosamente o fallos de cualquier tipo. Estaríamos muy agradecidos si nos comunican todos los errores que encuentren (p. ej., enviando un mail a manolov@dte.us.es).

Los tres capítulos incluidos son:

1. **Introducción a los sistemas digitales.** En él se plantean las formas de diseñar, describir y realizar los sistemas digitales a nivel RT. Como ejemplo de diseño, se desarrolla como una calculadora de sumas y restas.

2. **Diseño de unidades de control.** Se aborda la realización del controlador de los sistemas digitales, principalmente utilizando la técnica de un biestable por estado, pero presentando diversas opciones más, entre las que se incluye el control microprogramado con ROM y con PLA.

3. **Diseño a nivel RT de un computador simple.** Se introduce el concepto y modo de operación de los computadores desde una perspectiva de diseño, partiendo de la calculadora diseñada en el capítulo 1. Se desarrollan dos computadores simples. El segundo de ellos tiene un juego de instrucciones que permite ejecutar programas de bajo nivel para resolver tareas medianamente complejas. De este computador simple se dispone de un emulador que será utilizado en las prácticas de programación en ensamblador.

Anexos:

*** Ensamblador del Computador Simple**: Se describe el ensamblador y se realizan múltiples ejemplos de programación con este lenguaje ensamblador. También se presenta el emulador que permite ejecutar estos programas y ver el flujo de datos tanto instrucción a instrucción (nivel ISP) como μoperación a μoperación (nivel RT). Este emulador se utiliza en prácticas de laboratorio.

*** Multiplicación de magnitudes**: Se describen diferentes formas de multiplicar en binario. Tres de ellas se aplicarán en prácticas de laboratorio.

*** Glosario**: Un extenso número de conceptos son explicados brevemente en este anexo.

*** Referencias**: Contiene las referencias a la bibliografía consultada para elaborar el libro.

Índice

CAPÍTULO 1: INTRODUCCIÓN A LOS SISTEMAS DIGITALES

CAPÍTULO 2: DISEÑO DE UNIDADES DE CONTROL

CAPÍTULO 3: DISEÑO A NIVEL RT DE UN COMPUTADOR SIMPLE

CAPÍTULO 1: Introducción a los sistemas digitales

1.1 INTRODUCCIÓN

El aumento de la complejidad al evolucionar desde los CIRCUITOS hasta los SISTEMAS digitales tiene múltiples consecuencias en distintos niveles, algunos de los cuales se resumen en la Fig. 1.1. Una de las primeras es la necesidad de incrementar el nivel de abstracción de la información que se manipula. En nuestro caso esto significa que debemos pasar de manejar variables binarias (0 y 1) a agrupaciones de estas señales, agrupaciones en las que la información significativa es el dato que llevan. Con esta nueva perspectiva, la funcionalidad del Sistema Digital consiste en el procesado que se realiza sobre los datos. Por ejemplo, si se desea sumar el dato A con el dato B, lo que es significativo desde el nivel de Sistema es la operación A + B sin importar el valor binario concreto de A o de B.

	CIRCUITOS	SISTEMAS
Información	0,1	Palabras de datos
Nivel/Lenguaje	De conmutación	RT(Register Transfer)
Funcionalidad	Máquinas de estados finitos	Operaciones (instrucciones)
Componentes	Puertas y biestables	MUX, ALU, ..., registros,...
Conexión	Líneas (cables)	Buses
Organización	Combinacional y almacenamiento (memoria)	Procesado de datos y control

Figura 1.1: Circuitos versus Sistemas.

Por otra parte, el lenguaje de conmutación (combinacional y secuencial) muy útil al manejar 0's y 1's, no puede describir adecuadamente el procesado entre datos. Surge, pues, la necesidad de emplear un nuevo lenguaje, más abstracto, apropiado para dicho procesado. Este lenguaje es denominado de transferencia entre registros (RT). Este nombre procede del que reciben genéricamente los dispositivos (registros) que almacenan datos. El lenguaje de descripción establece pues, otra diferencia: mientras que los circuitos ocupan el denominado nivel de conmutación, los sistemas se sitúan en el nivel RT.

Desde la perspectiva funcional, los circuitos realizan máquinas de estados finitos. Estrictamente hablando, los sistemas también. Sin embargo, como la utilidad principal de estos sistemas consiste en el procesado de datos, es preferible describir su funcionalidad en términos de operaciones entre datos, también llamados macrooperaciones o instrucciones del sistema. Ello permite, por una parte, manejar funciones más complejas ya que es posible encadenar secuencias de estas instrucciones (hacer un "programa") para resolver problemas mucho más complicados que los resueltos por cada instrucción por separado. Por otra parte, también se consigue aproximar el lenguaje de la máquina al del ser humano, aunque simultáneamente esto signifique un alejamiento entre el lenguaje de descripción usado y la ejecución real de las tareas por los circuitos. El nivel de la macrooperación se sitúa en un punto intermedio entre las operaciones de conmutación del hardware y las instrucciones software. Además, la descripción funcional mediante operaciones entre datos comparte la resolución algorítmica de problemas con otras muchas aproximaciones (como la de la programación). Con todo ello se tiende un importante puente de conexión entre el hardware y el software.

Particularizando sobre la realización de los Sistemas Digitales, el nuevo enfoque supone cambios en los componentes de diseño y en las conexiones entre ellos. En relación a los componentes, se utilizan subsistemas (secuenciales y combinacionales) preferentemente a puertas y biestables. Con ello se consigue una mayor aproximación entre el lenguaje de descripción y los componentes utilizados, a la vez que se aprovecha la mayor potencia y flexibilidad de los subsistemas frente a las puertas y biestables. Además, el uso de estos componentes está en consonancia con los propios cambios en los criterios de diseño en los que ahora priman los aspectos de modularidad, sencillez en el proceso de diseño, fiabilidad, testabilidad, etc., sobre el del coste en número de puertas. Por otra parte, la interconexión entre los componentes se realiza mediante buses (conjunto de líneas con un significado global claro: p. ej., bus de datos) pero en el que pierde sentido cada una de sus líneas en concreto. Es por estos buses por donde viajan los datos sin importar mucho si una de estas líneas lleva un 1 ó un 0.

De todo lo anterior surge un cambio sustancial en la organización del Sistema en relación a la del Circuito, tal como se muestra en la Fig. 1.2. En vez de diferenciar las partes combinacional (funciones de próximo estado y salida) y secuencial (elementos de almacenamiento de estado), un Sistema Digital se organiza en una "Unidad de Procesado de Datos"(UPD) y una "Unidad de Control" (UC). La UPD también llamada simplemente Unidad de Procesado o Unidad de Datos, es la parte del Sistema que:

- recibe los datos de entrada $D_{IN.}$
- procesa esos datos, para lo cual realiza las transferencias entre registros indicadas en el algoritmo que se desea ejecutar.
- saca al exterior los resultados de salida D_{OUT}.

Por su parte, la UC es la parte del Sistema que controla las tareas que realiza la UPD. En particular las funciones de la UC son:

- generar la señales de control (Z) que necesitan los componentes de la unidad de datos, para realizar el proceso correspondiente. Asímismo, deberá generar las posibles salidas al exterior que correspondan a señales de control (Z_{OUT}), como es, por ejemplo, una señal de fin de tarea. A las señales de salida del controlador se les denominan **comandos**.
- establecer la secuencia de acciones a la que obliga el algoritmo que se ejecuta. Esto significa que, para todo estado, además de generar los comandos, la UC debe conocer su próximo estado y alcanzarlo en el siguiente ciclo. La posible evolución de la secuencia de estados dependerá de un conjunto de señales (X, X_{IN}) que actúan como entradas de control y a las que se denomina **cualificadores**. Al igual que los comandos, los cualificadores pueden provenir del exterior (X_{IN}) o de la unidad de datos (X) en cuyo caso a veces se les denomina "señales

de estado", nombre que a su vez alude al estado del procesado: si hay acarreo, resultado nulo, ... (En inglés, procede de "status" que no debe confundirse con el estado interno de una máquina secuencial : "state").

Cada unidad del sistema, tanto si se trata de la de datos como si es la de control, es en realidad una máquina de estados, por lo que realiza funciones combinacionales y secuenciales y pueden, por tanto, ser estudiadas como circuito secuencial. Esta perspectiva será, en efecto, útil para tratar algunas formas de realización de unidades de control. Sin embargo, es de nula utilidad al describir las unidades de procesado de datos.

A lo largo de este texto y salvo expresa indicacion en contra, los Sistemas Digitales que manejaremos serán síncronos y la misma señal de reloj gobernará ambas unidades: en la de datos el reloj controlará las distintas operaciones de escritura en los registros; en la de control, gobernará los cambios de estado mediante los que se establece la secuencia de transferencias de datos entre los registros. El tratamiento de operaciones asíncronas no será, pues, objeto de estudio en este texto.

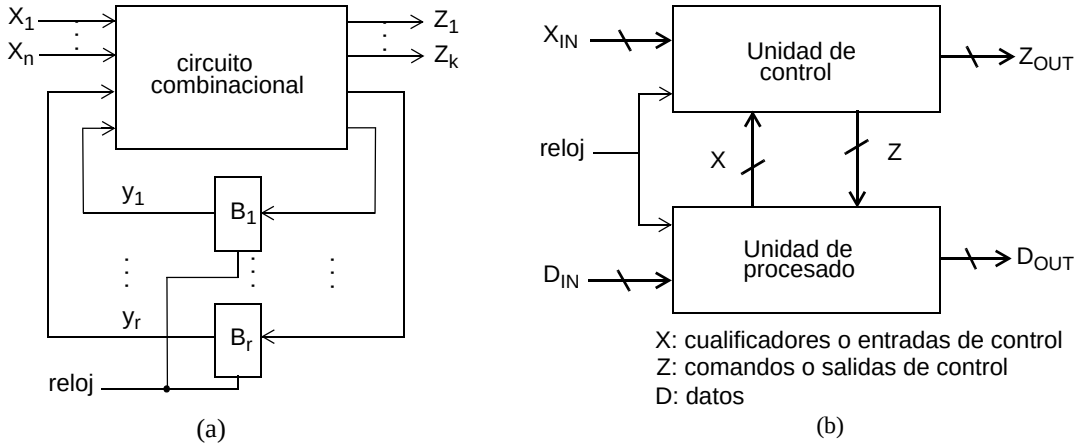


Figura 1.2: (a)Estructura general de una FSM (Máquina de estados finitos). (b)Estructura general de un sistema digital.

1.2 EL NIVEL DE TRANSFERENCIA ENTRE REGISTROS (RT)

El nivel RT (*Register Transfer*) es una forma de descripción de sistemas digitales mediante datos almacenados en registros. El término registro, en el nivel RT de los sistemas digitales, es un término más amplio que el asociado al subsistema secuencial concreto del que toma el nombre. En efecto, con registro nos referimos a cualquier dispositivo capaz de almacenar datos, englobando: los biestables como registros de 1 bit; los contadores como registros que incrementan/decrementan el dato almacenado; las memorias como banco de registros; y los propios registros (de carga en paralelo y de desplazamiento) que son los que dan nombre a este nivel de descripción.

1.2.1 Descripción de registros a nivel RT.

Al ser el componente básico de este nivel, el registro debe ser adecuadamente conocido y descrito bajo las perspectivas de uso en los sistemas digitales: 1) como bloque o componente del sistema (representación estructural); 2) como elemento de almacenamiento de la información (representación de datos); y 3) como circuito que tiene una forma dada de operar (representación funcional). A conti-

nuación se detallan estas tres perspectivas.

- Representación estructural.

A nivel estructural un registro genérico es (Fig. 1.3) un bloque de *n* bits con *m* señales de control: s_1, ..., s_m, que dependiendo de la combinación binaria que posean, harán que el registro realice una operación u otra. Además poseerá un conjunto de líneas de entrada de datos (de las que en la Fig. 1.3 se han representado las *n* líneas de entrada en paralelo), así como otro conjunto de líneas de salida de datos por donde se podrá acceder al contenido (salidas en paralelo en la Fig. 1.3). Además, al ser dispositivos secuenciales, poseerán su entrada de reloj que, salvo indicación en contra, supondremos en adelante que será activa en el flanco de subida.

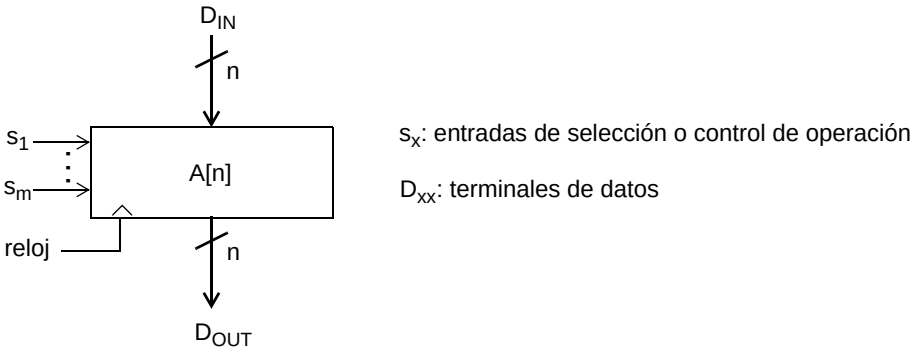


Figura 1.3: Representación estructural de un registro.

- Representación de datos.

Con esta representación nos referimos al contenido que tiene almacenado el registro y que, en general, es un “dato”. A este se puede acceder completamente, esto es, a toda la palabra binaria en él almacenada, o parcialmente, es decir, sólo a algunos bits del dato.

La representación del dato se esquematiza en la Fig. 1.4, en la que hemos supuesto que el nombre del registro con el que trabajamos es A. Si hacemos referencia al dato almacenado al completo lo llamaremos [A] o, cuando no haya posibilidad de confundir el dato y el dispositivo, simplemente A. Sin embargo, si queremos referirnos a bits sueltos del dato lo haremos escribiendo los subíndices correspondientes teniendo en cuenta que en este texto representamos a la derecha el bit menos significativo (posición 0). Así, $[A_{i\text{-}j,k}] = [A]_{i\text{-}j,k}$ hace meción a los bits correlativos desde el i hasta el j, y, además, al bit k-ésimo. Al igual que antes, si no hay confusión entre líneas y bits de datos, también puede ponerse $A_{i\text{-}j,k}$.

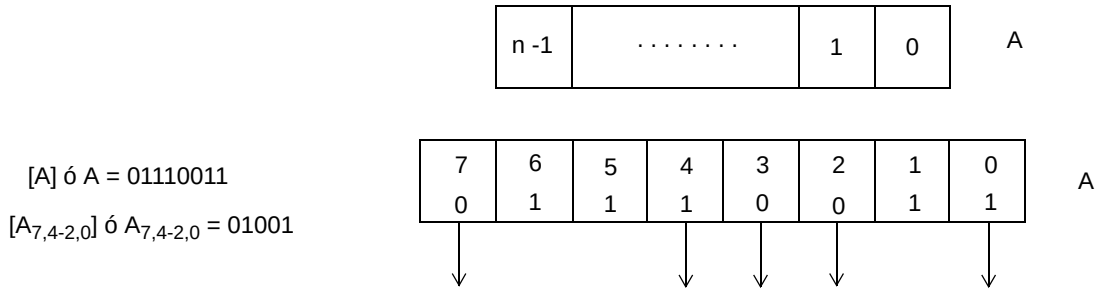


Figura 1.4: Representación de datos.

- Representación funcional.

La representación funcional debe describir la operación y suele ser la representación más compleja del registro. En general se pueden distinguir tres tipos de funcionalidades: la operación de escritura, la operación de lectura y la activación de las señales de control. Pasemos a describirlas en más detalle.

- Escritura en R: aquellas acciones sobre el registro que conducen a una modificación del dato almacenado.

R ← nuevo dato

Se trata de una operación de tipo secuencial. Es decir, el nuevo dato se carga en el registro R de forma sincronizada con su señal de reloj.

Las operaciones más comunes de escritura y su expresión a nivel RT se muestran en la Tabla 1.1.

Operación	Notación RT
Carga en paralelo	$A \leftarrow D_{IN}$
Desplazamiento a derecha	$A_{n-1} \leftarrow D_r$, $A_i \leftarrow A_{i+1}$ $\forall i \neq n-1$; $A \leftarrow SHR(A,D_r)$
Desplazamiento a izquierda	$A_0 \leftarrow D_l$, $A_i \leftarrow A_{i-1}$ $\forall i \neq 0$; $A \leftarrow SHL(A,D_l)$
Incremento (Decremento)	$A \leftarrow A + 1$, $(A \leftarrow A - 1)$
Puesta a 0 (ó 1)	$A_i \leftarrow 0$ $\forall i$ ó $A \leftarrow 0$ $(A_i \leftarrow 1$ $\forall i$ ó $A \leftarrow 1 \ldots 1)$
Inhibición (NOP)	$A \leftarrow A$

Tabla 1.1: Principales operaciones de un registro a nivel RT.

En la operación de carga en paralelo, el nuevo dato del registro es el que hay en las entradas D_{IN} (Fig. 1.3) cuando se activa el flanco de reloj.

Las operaciones de desplazamiento a derecha y a izquierda (Fig. 1.5), además del desplazamiento interno de los bits, supone la entrada de un nuevo bit por una línea (D_r y D_l, respectivamente) que se almacena en una de las celdas extremas del registro (A_{n-1} y A_0, respectivamente). El nuevo dato que se almacena está constituido, pues, por *n*-1 bits del dato antiguo, pero ocupando ahora las posiciones consecutivas, más el nuevo bit que entra por la línea D_r ó D_l. En notación RT estas operaciones tienen dos formas de escribirse: una, expresando la operación de las distintas celdas individualmente; la otra, englobándolas bajo la operación SHR ó SHL(registro, bit)[1].

Por su parte, las operaciones típicas del contador, de incremento y de decremento, se describen sumando o restando la unidad al dato presente, sin especificar nada más ya que se asume que son contadores de magnitud y que hacen la cuenta.

La operación de borrado o puesta a 0 se escribe poniendo 0 como nuevo dato. La operación de puesta a 1 no puede ponerse escribiendo 1 como nuevo dato ya que el nuevo dato es “todos los biestables a 1”, con magnitud de 2^n-1; en su lugar es simple poner “11...11” como nuevo dato. Por último, la operación de “inhibición” o de no-operación (NOP) deja en el registro

1. Nótese que el significado de A varía según donde aparezca: si está a la derecha de la flecha (posición “fuente”) representa el dato actualmente almacenado en el registro, mientras que si está a la izquierda (posición “destino”) indica el registro que, tras la actuación del flanco activo de reloj, almacenará el nuevo dato.

el mismo dato que había.

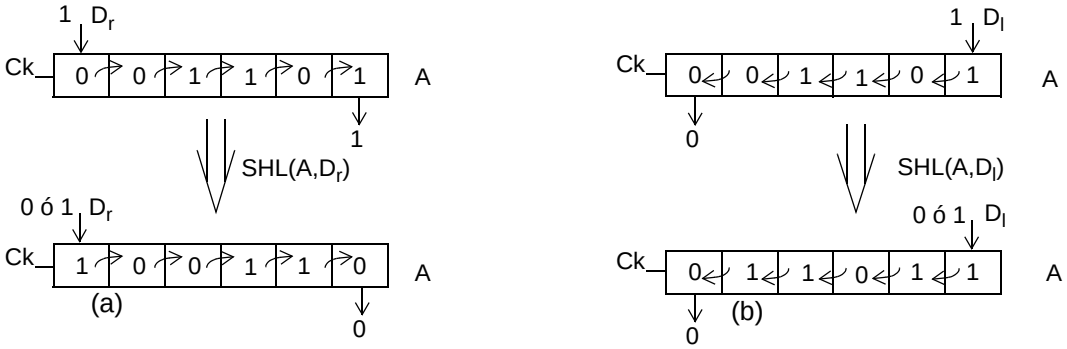


Figura 1.5: Operaciones de desplazamiento: (a) A derecha; (b) A izquierda.

- Lectura de R: básicamente consiste en acceder al dato almacenado o a alguna función combinacional del mismo a través de las líneas de salida del registro. En notación RT se escribe:

SAL = dato/función del dato

donde SAL es el nombre de las salidas.

El símbolo "=" hace alusión a que la lectura es una operación de tipo combinacional. Esto es, salvo los retrasos de propagación de la lógica, las salidas SAL muestran el valor actual del dato o de la función del dato. Por ejemplo, si el registro de la Fig. 1.3 se lee en paralelo cuando su contenido es el de la Fig. 1.4, la operación de lectura en notación RT es:

$$D_{OUT} = 73$$

donde el número 73 está en hexadecimal, que es la notación habitual para dar datos binarios cuando no se usa el propio valor binario.

Un ejemplo de lectura de una función del dato se muestra en la Fig. 1.6. La salida CERO se activa cuando el registro A está borrado. A nivel RT se escribe:

$$CERO = NOR(A_{n-1}, ..., A_0)$$

aunque también puede escribirse como:

$$CERO = 0 \quad \text{si } A \neq 0$$
$$CERO = 1 \quad \text{si } A = 0$$

Otros casos de salida tipo función del dato son las señales de fin de ciclo de cuenta de los contadores (llamadas ¨carry¨ en los ascendentes y ¨borrow¨ en los descendentes).

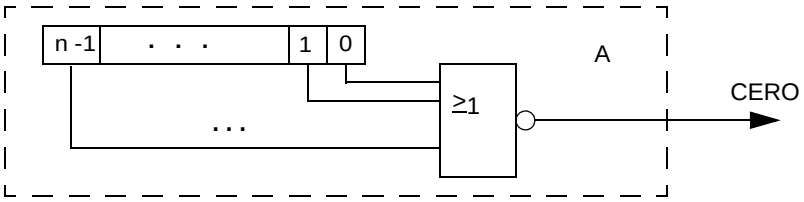


Figura 1.6: Lectura tipo función del dato.

La operación de lectura, sea del dato o de una función del dato, puede ser incondicional o condicional.

- Lectura incondicional. Es la que ocurre si el acceso desde la salida no está condicionado por ninguna señal: las salidas del registro muestran siempre el dato almacenado (Fig. 1.7-a)

o la función combinacional del dato.

- Lectura condicional. En este caso sólo se podrá acceder al dato almacenado en el registro si cierta condición de control, que aquí llamaremos señal de lectura R (read), es verdadera. En notación RT se escribirá:

  R = 1 $D_{OUT} = A$

  R = 0 $D_{OUT} = ?$

  Por lo general, cuando el registro comparte el bus desde el que se accede a sus datos (véase apartado 1.3), en sus líneas de salida se muestra el estado HI. Otras alternativas son fijar las líneas de salida al valor 0 ó al 1. En la Fig. 1.7.b-d se muestran las distintas implementaciones de salida de una etapa típica de un registro según tenga salida condicional de uno u otro tipo, así como su descripción RT.

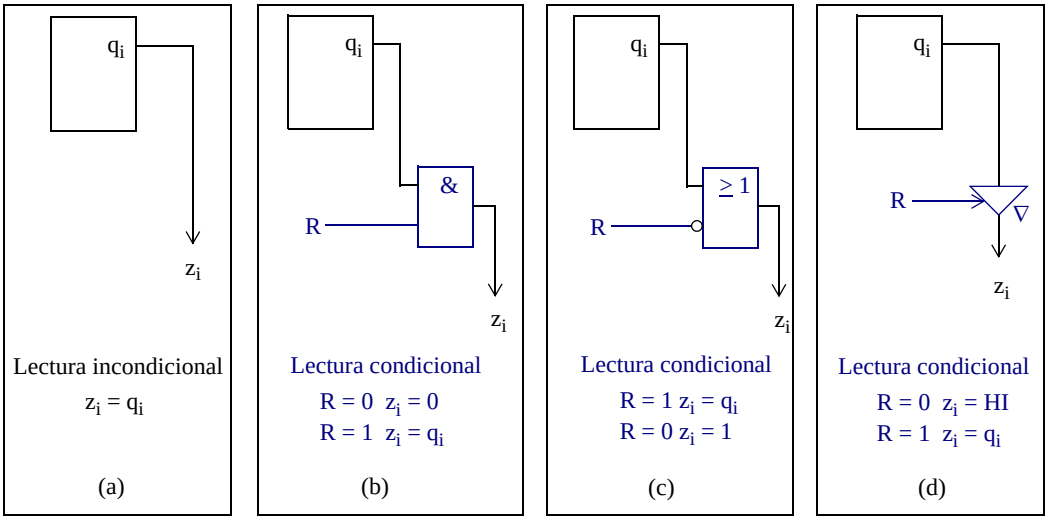


Figura 1.7: Implementaciones para la lectura sobre un registro

- Control. En general, un dispositivo-registro tendrá un conjunto de señales de control (s) que seleccionarán cuál de las operaciones posibles en el registro es la que se va a realizar. Cada vez que cierta función combinacional de las señales se haga verdadera, se realizará una operación en particular. A nivel RT se escribe:

  f(s): operación

  Para el manejo personal es muy útil describir las actuaciones de control mediante una tabla, en la que para cada combinación significativa de las señales de control se especifican todas las acciones “secuenciales” (escritura) como “combinacionales” (lectura). En muchos de los ejemplos que siguen se usa la descripción mediante tablas ya que es muy clara de entender.

  Con mucha frecuencia cada operación del registro es controlada por una única señal que es activada cuando se desea ejecutar dicha operación. En estos casos el nombre que se le da a la señal suele “recordar” la operación que controla.

  En la tabla Tabla 1.2 se han representado algunos nombres frecuentes, casi siempre procedentes de la terminología inglesa. Así, para la carga en paralelo se usa W (Write), junto con T (Transfer) y L (Load). Para las operaciones de desplazamiento se usa S (Shift)

junto con R (Right) si es a la derecha o L (Left) si es a la izquierda. Las operaciones típicas del contador usan I (Increment) o UP y D (Decrement) o DOWN. El borrado o puesta a 0 usa CL (CLear) o Z (Zero), mientras que S (Set) indica la puesta a 1. La inhibición ocurre si no se activa ninguna operación, por lo que estará indicada cuando no hay ninguna señal de control activa; también se indica cuando no se activa la señal de selección de chip CS (Chip Selection) o de habilitación EN (ENable) o cuando está activa la señal de deshabilitación DIS (DISable). Por último, para la lectura se suele usar R (Read).

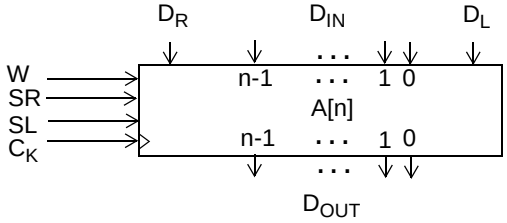
Operación	Señal de control
Carga en paralelo	W (T,L)
Desplazamiento derecha/izquierda	SR/SL
Incrementa/Decrementa	I/D (UP/DOWN)
Puesta a 0/1	CL, (Z) / S
Inhibición	Ninguna señal activa (CS,EN,DIS)
Lectura	R

Tabla 1.2: Algunas señales frecuentes para controlar operaciones.

A continuación aplicaremos la descripción de registros a nivel RT sobre diversos dispositivos como ejemplo.

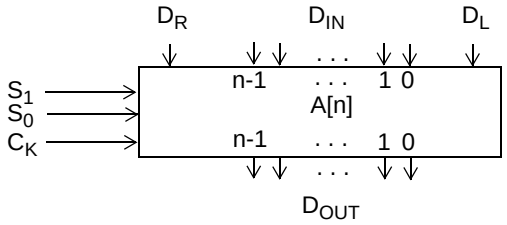
Ejemplo1. Descripción de un registro de *n* bits bidireccional con carga en paralelo.

Las figuras 1.8 y 1.9 muestran dos formas distintas del mismo registro: uno de *n* bits, con carga en paralelo y desplazamiento bidireccional con dos entradas de dato serie (D_R y D_L) y cuya lectura es incondicional. Su diferencia está únicamente en cómo se selecciona cada operación.



W SR SL	Escritura en A		Lectura de A
0 0 0	A ← A	Inhibición	$D_{OUT} = A$
1 0 0	A ← D_{IN}	Carga en paralelo	
0 1 0	A ← SHR(A,D_R)	Desplazamiento a la derecha	
0 0 1	A ← SHL(A,D_L)	Desplazamiento a la izquierda	
Otras	Prohibidas		

Figura 1.8: Registro del Ejemplo 1 con señal de control específica para cada operación



S_1	S_0	Escritura en A		Lectura de A
0	0	$A \leftarrow A$	Inhibición	$D_{OUT} = A$
0	1	$A \leftarrow D_{IN}$	Carga en paralelo	
1	0	$A \leftarrow SHR(A,D_R)$	Desplazamiento a la derecha	
1	1	$A \leftarrow SHL(A,D_L)$	Desplazamiento a la izquierda	

Figura 1.9: Registro del Ejemplo1 con señales de selección de operación mediante código de valores.

En el caso de la Fig. 1.8 cada operación sobre el registro lleva una señal de control asociada, de forma que cuando esta señal vale 1 se realiza la operación correspondiente. Asímismo, como se prohibe la realización de más de una operación sobre el registro simultáneamente entre las señales de control sólo una podrá tener el valor lógico 1 cada vez. Se dice a veces que, en este tipo de registros, las señales de control están decodificadas. La situación “ninguna activa” indica la inhibición.

La forma algebraica de describir el registro de la Fig. 1.8 a nivel RT, bajo las dos suposiciones ya comentadas (prohibir dos o más señales activas y que la inhibición es no activar señales), es:

$$W: A \leftarrow D_{IN}$$
$$SR: A \leftarrow SHR(A,D_R)$$
$$SL: A \leftarrow SHL(A,D_L)$$

De forma análoga, pueden encontrarse dispositivos cuyas señales de control “estén codificadas”. Esto significa que, en cada combinación binaria distinta de las señales de control S_1, ..., S_m se realiza una operación diferente sobre el registro, pudiendo estar más de una señal de control activa simultáneamente. Así, cada operación se ejecuta en un código de entrada. De esta forma podrá elegirse siempre el mínimo número de señales de control necesarias para codificar al número de operaciones distintas que se quieran realizar sobre el registro. En la Fig. 1.9 se aplica esta alternativa a un registro como el anterior (Fig. 1.8). Obsérvese que ahora sólo se usan dos señales de control y que cada código de dichas señales selecciona una operación. De forma algebraica a nivel RT la descripción será:

$$\overline{S}_1\overline{S}_0: A \leftarrow D_{IN}$$
$$S_1\overline{S}_0: A \leftarrow SHR(A,D_R)$$
$$S_1S_0: A \leftarrow SHR(A,D_L)$$

donde hemos asumido que el cuarto código ($\overline{S}_1\overline{S}_0$), al causar una no operación (inhibición), no es necesario explicitarlo.

Ejemplo2. Descripción de una RAM comercial: RAM 2114.

Como segundo ejemplo de descripción a nivel RT se presenta la de un dispositivo comercial relativamente complejo: el CI[1] 2114 (Fig. 1.10). Se trata de una RAM de 10 líneas de dirección, 4 líneas bidireccionales de datos, con una capacidad de 4096 bits (organizados como 1K × 4) y con dos señales de control: una activa en baja para seleccionar el chip y la otra que controla si la operación es de lectura de la RAM (R/$\overline{W}$ = 1) o de escritura en ella (R/$\overline{W}$ = 0). Obsérvese que el dato que se encuentra en las salidas $D_{3\text{-}0}$ durante la lectura es el que se encuentra almacenado en la palabra de la RAM direccionada por las líneas de dirección A (A = $A_{9:0}$).

$$D = [RAM(A)]$$

Como las líneas de datos D son bidireccionales, durante la operación de escritura actúan como entradas hacia la RAM "portando" el dato D_{IN} a escribir, dato que habrá situado otro circuito no contemplado aquí. El dato D_{IN} se escribirá en la palabra seleccionada por las líneas de dirección A que actúa como "registro" en el que escribir:

$$RAM(A) \leftarrow D_{IN}$$

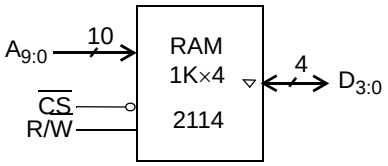
entendiendo que el resto de "registros" de la RAM no cambian:

$$RAM(B) \leftarrow RAM(B) \qquad \forall B \neq A$$

De forma algebraica se escribirá:

$$\overline{\overline{CS}} \cdot R/\overline{W}: D = [RAM(A)]$$
$$\overline{\overline{CS}} \cdot \overline{R/\overline{W}}: RAM(A) \leftarrow DIN$$

entendiendo que los casos no especificados están inhibidos (no hay cambio de datos y las salidas en HI).



$\overline{CS}$	R/$\overline{W}$	RAM	$D_{3\text{-}0}$	Comentarios
1	-	RAM ← RAM	HI	-
0	1	RAM ← RAM	[RAM(A)]	Lectura
0	0	RAM(A) ← D	$D_{3\text{-}0}$	Escritura

Figura 1.10: Descripción RT de la RAM 2114.

1. CI: Circuito Integrado

1.2.2 Operaciones de transferencias entre registros.

La unidad de procesado de datos (Fig. 1.2-b) de un sistema digital genérico contiene varios registros interconectados entre sí a través de líneas conductoras y circuitos combinacionales. Como el propio nombre indica, a nivel RT la principal operación dentro de dicha unidad es la transferencia entre registros, nombre con el que se denomina al movimiento y transformación de datos desde uno o más registros "fuentes" a un registro "destino".

En general, la descripción a nivel RT de una transferencia entre registros se podría expresar de la siguiente forma:

$$\mathbf{f(x): A \leftarrow G(B, C, ...)} \qquad (1.1)$$

donde f(x) expresa la condición que debe satisfacerse para realizar la transferencia, A es el registro destino del dato-resultado y G(B, C, ...) indica la operación que se realiza sobre los datos fuentes.

El sistema digital tiene que estar construido para poder realizar cualquiera de las transferencias entre registros que precise la solución del problema a resolver. De aquí que haya una fuerte relación entre las sentencias RT (ecuación 1.1) y la realización del sistema digital. En lo que sigue estableceremos dicha realización.

De la ecuación 1.1 se entiende que los argumentos B, C, ... son registros que participan, mediante la operación de lectura de sus datos, en la instrucción (registros fuentes). La función G es una función combinacional que maneja los datos almacenados en dichos registros (B, C, ...). En el registro A (registro destino) se realiza la carga en paralelo del dato resultante de la función G sólo si la función f(x) tiene el valor lógico correcto. La función f(x) tiene carácter combinacional y su evaluación corresponde a la Unidad de Control del Sistema Digital. Por tanto, para poder realizar una transferencia entre registros se necesitan una serie de requisitos: 1) de **componentes**, como son los propios registros y los dispositivos combinacionales para realizar las funciones f(x) y G; 2) de **conexión**, para el camino de los datos desde las fuentes hasta su destino y para las señales de control de todo el proceso; y 3) de **organización**, mediante la cual se realiza la distribución de componentes y tareas, así como se hace el diseño arquitectural del sistema. En general son posibles muchas alternativas de solución para el mismo problema.

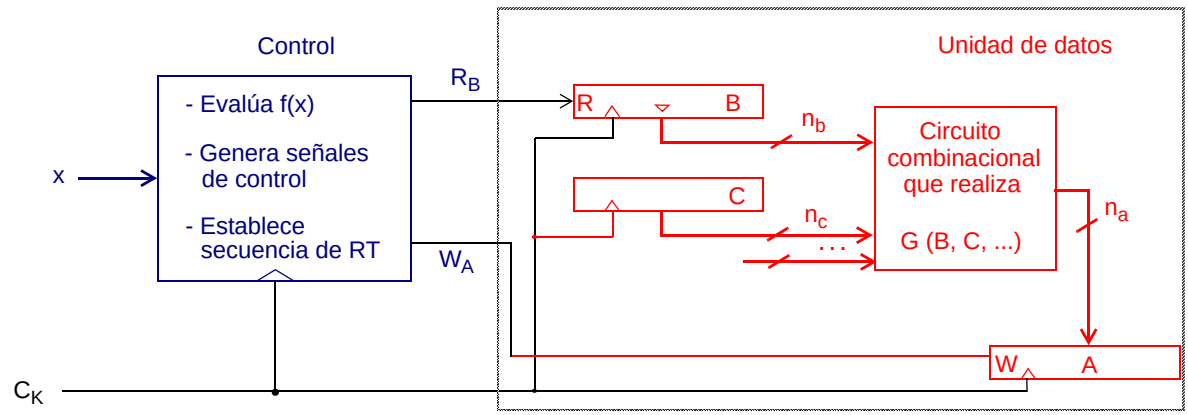


Figura 1.11: Sistema digital para la sentencia RT de la ecuación 1.1.

A modo de ejemplo, en la Fig. 1.11 aparece un sistema digital que permite ejecutar la sentencia RT de la ecuación 1.1. El sistema digital está dividido en dos grandes bloques, la unidad de control y la de datos.

La unidad de control se encarga, a partir de sus entradas x, de evaluar la función f(x) en cada momento. Si esta función toma el valor adecuado, deberá hacer efectiva la transferencia entre registros en curso. Tras ello, la unidad de control debe saber qué nueva transferencia ha de ejecutarse a continuación. Esto es, la unidad de control establece la secuencia de RT. Nótese que, por ser una operación secuencial (escritura en registro) se realizará coincidiendo con un flanco activo de reloj C_k. De aquí que la unidad de control sea un circuito secuencial (máquina de estados finitos).

Para llevar a cabo una transferencia, la unidad de control activará las señales necesarias en los componentes que participan en la transferencia. En la unidad de datos están estos componentes junto con la ruta de datos adecuada para conectar las fuentes y el destino. Por ejemplo, en la Fig. 1.11 tanto el registro B (de n_b bits) como el C (de n_c bits) tienen sus salidas conectadas a las entradas del circuito combinacional que realiza la función G(B,C, ...). A su vez las salidas de éste están conectadas a las n_a entradas de carga en paralelo del registro A. Dicho registro tiene una entrada de control de escritura (W_A) que le es suministrada desde la unidad de control. El reloj de todo el sistema es único, gobernando tanto los cambios de estado en la unidad de control como los instantes en que se almacenan los nuevos datos. Obsérvese que, en esta unidad de datos, el registro B posee salida triestado por lo que la lectura de B está controlada por la señal de lectura R_B. La unidad de control debe generar también dicha señal R_B.

La Fig. 1.12 muestra cómo es la operación en el tiempo. Se ha supuesto que la función G(B, C, ...) es simplemente la suma aritmética de B y C y, para dar valores concretos, que inicialmente A = F1, B = 72 y C = 24. La Unidad de Control activa las señales R_B y W_A sólo en el ciclo de reloj donde ha de ejecutarse la sentencia RT del ejemplo (A ← B + C). Así, sólo en ese ciclo las salidas de B muestran el contenido del registro B y, con ello, las salidas del bloque G muestran el valor deseado (96 = 72 + 24) que se sitúa por tanto como entrada de carga en paralelo del registro A. Como, además, este registro tiene su entrada de escritura activada (W_A = 1), cuando llega el flanco activo de reloj se produce la carga con lo que queda ejecutada la sentencia RT. Simultáneamente la Unidad de Control habrá cambiado al próximo estado en el que ejecutará la sentencia RT posterior.

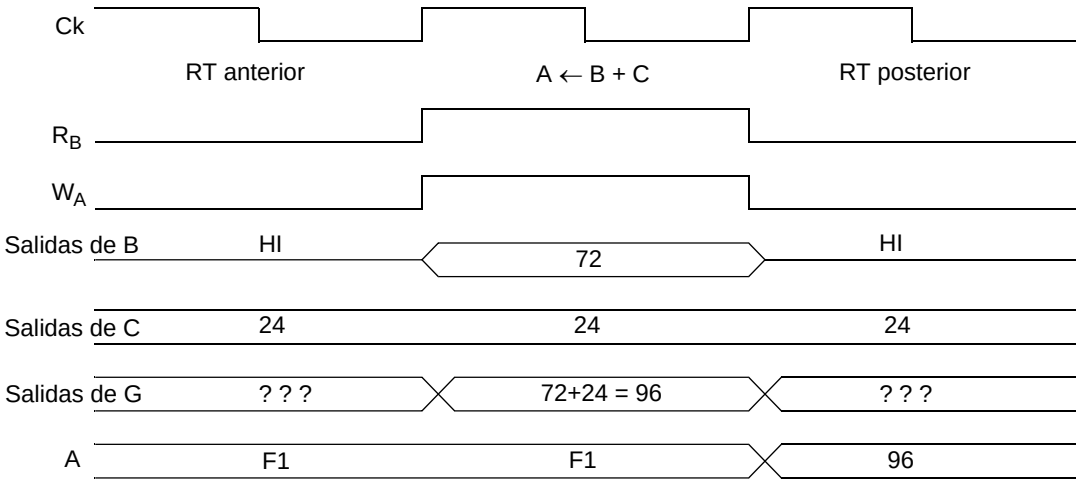


Figura 1.12: Diagrama temporal para ilustrar A ← B + C.

Aunque teóricamente son posibles transferencias entre registros muy complejas que involucran a múltiples datos, las de interés a nivel RT sólo afectan a 1, 2, ó 3 registros. Así, las transferencias entre registros a nivel RT pueden ser clasificadas en:

- Transferencias entre varios registros (típicamente 2 ó 3):
  - De movimiento de datos: $A \leftarrow B$
  - Aritméticas: $A \leftarrow B + C$ $A \leftarrow B - 1$
  - Lógicas: $A \leftarrow B \oplus C$ $A \leftarrow \overline{C}$

  Estas transferncias exigen una ruta de datos entre los registros fuentes y el destino, ruta que incluye circuitos combinacionales y buses de interconexión. El registro destino operará en escritura y los fuentes en lectura. Si un registro es a la vez fuente y destino, como es el caso del "Acumulador" de los procesadores, deberá estar diseñado para que sea posible operar en lectura y en escritura simultáneamente.

- Sobre un único registro:
  - Desplazamiento: a derecha o a izquierda
  - Cuenta: ascendente o descendente
  - De inicialización: típicamente de Puesta a 0 o de Puesta a 1

  Las transferencias de un único registro no exigen más que el registro tenga definida la operación correspondiente.

Ejemplos. Transferencias condicionales entre registros y unidades de datos que las realizan.

La Fig. 1.13 muestra cuatro ejemplos de transferencias entre registros cuando se cumplen las condiciones X_1, X_2, X_3 y X_4, respectivamente, junto con un posible circuito que realiza esa transferencia.

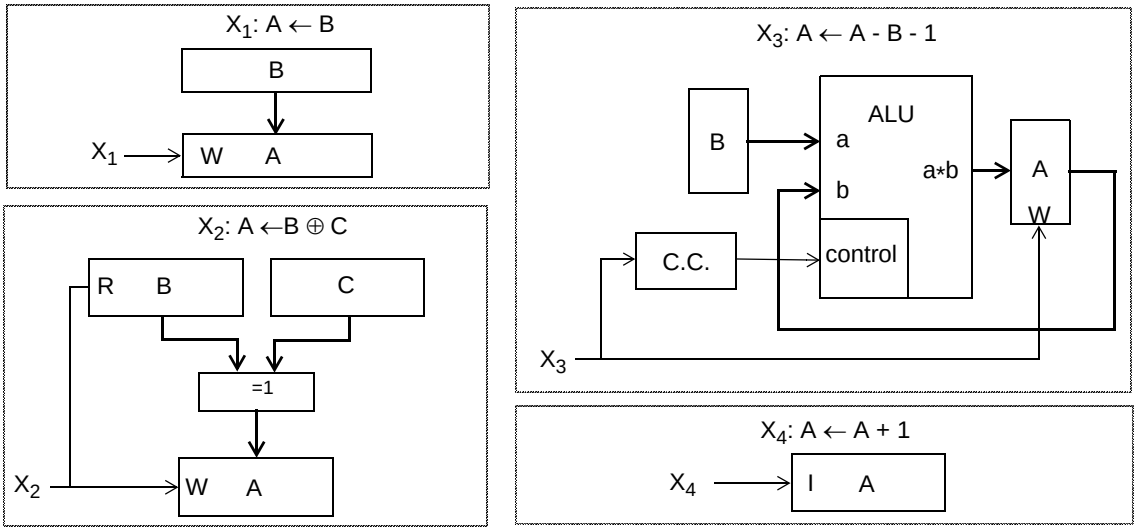


Figura 1.13: Algunas sentencias RT y posibles circuitos que las realizan.

a) En el primer caso se mueve el dato de B (lectura incondicional) hacia A (escritura);

b) en el segundo hay una operación lógica XOR entre los datos de B y de C, que precisa la lectura del registro B pero no la de C;

c) en el tercero hay dos aspectos de interés: 1) que el registro A es fuente y destino, y 2) que se ha usado una ALU para realizar la operación aritmética por lo que la señal X_3 a través del circuito combinacional CC, debe seleccionar la operación adecuada en la ALU;

d) en el cuarto caso el registro A es un contador ascendente simplemente.

1.3 TÉCNICAS DE INTERCONEXIÓN MEDIANTE BUSES

Las operaciones básicas a nivel RT son las transferencias entre registros. En un sistema digital genérico hay múltiples transferencias de unos registros a otros, siendo frecuente que un registro dado sea a veces emisor del dato (registro fuente) y otras, receptor de un resultado (registro destino). Surge así la necesidad de muy diversas rutas de datos. El sistema digital tiene que estar construido de forma que sean transitables todas las rutas de datos necesarias. Desde la perspectiva del hardware esto significa que el sistema deberá estar dotado con la adecuada conexión entre los componentes almacenadores y procesadores de los datos (esto es, los registros y los bloques combinacionales). La presentación de algunas técnicas de interconexión es el propósito de este apartado para lo cual inicialmente se describirán los términos básicos relacionados con los buses de interconexión; a continuación se considerarán las operaciones con los buses y, por último, se presentarán algunos ejemplos típicos de interconexiones entre registros.

La entrada y salida de datos de los registros que lleva asociada una transferencia estándar utiliza los buses como principal vía de interconexión. Entendemos por bus el conjunto de líneas de conexión entre dispositivos que se utiliza para transmitir información. Todas las líneas del bus son similares entre sí salvo en el propio orden de cada línea respecto a las demás. Así, hay buses de líneas de dirección como son las usadas en los mapas de memoria, buses de datos externos a una CPU, buses de datos internos en una CPU, etc. La Fig. 1.14 muestra algunas de las representaciones habituales para un bus, denominado B, de *n* líneas . Siempre es conveniente explicitar el nombre y el número de líneas al representar un bus. Si se quiere aludir a una línea concreta basta indicarlo subindicando el nombre del bus (por ejemplo, $B_{3\text{-}5}$ alude a las líneas 3, 4 y 5 del bus B).

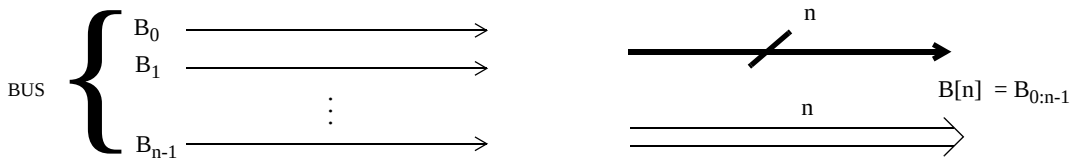


Figura 1.14: Representación del bus B de *n* líneas.

La clasificación de los buses puede hacerse atendiendo a distintos conceptos, como son:

- Según el sentido en el que fluye la información en el bus podemos distinguir:
  - buses unidireccionales: la información va en un único sentido, de un registro fuente a uno o varios registros destino (Fig. 1.15.a).
  - buses bidireccionales: la información puede fluir en cualquier dirección. Cada registro

puede actuar de fuente o destino según el caso (Fig. 1.15.b). Obsérvese que los terminales de los registros conectados a buses bidireccionales deben ser triestado para evitar colisiones entre los datos de entrada y de salida de cada registro.

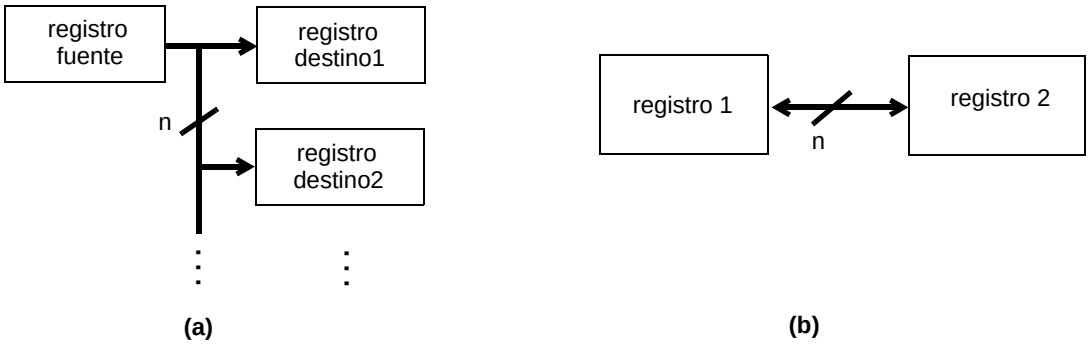


Figura 1.15: (a) Bus unidireccional. (b) Bus bidireccional.

- Atendiendo a si la información que transmite el bus es compartida sólo por dos registros o puede serlo por más, podemos clasificar los buses en dedicados y compartidos (Fig. 1.16).

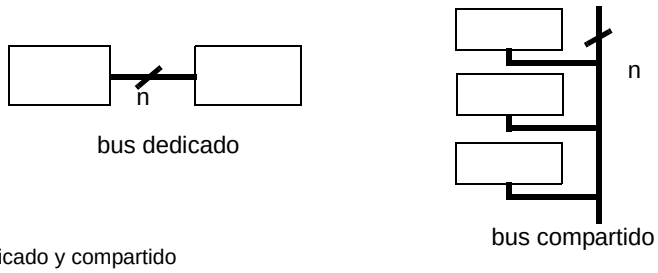


Figura 1.16: Buses dedicado y compartido

- Atendiendo ahora a las operaciones que se pueden realizar sobre un bus, éstas son de lectura y de escritura. La operación de lectura de un bus por parte de un registro lleva asociado el almacenamiento en el registro del dato que porta el bus. Para realizar esta acción debe existir, pues, un camino desde el bus a las líneas de entrada del registro y, a su vez, que se realice una operación de escritura en el registro mediante la cual la información quede almacenada en él. Hay que tener precaución con la terminología ya que la lectura del bus es escritura en el registro.

Para realizar la conexión necesaria que lleve a cabo esta operación pueden plantearse dos circunstancias. En la primera un registro lee de un único bus. La conexión del bus con las entradas en este caso es directa a través de conductores (Fig. 1.17). De esta forma, cuando la señal de control de escritura del registro W esté activa se realizará la operación de carga en paralelo del dato leído del bus.

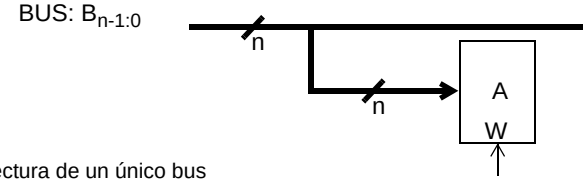


Figura 1.17: Conexión para la lectura de un único bus

Sin embargo, la lectura por parte del registro puede hacerse de más de un bus. Para ello la circuitería debe permitir un camino de conexión entre cada uno de los diversos buses y las líneas de

entrada del registro. Además, para elegir en cada ocasión cuál de entre los posibles buses va a ser leído, se utilizarán unas señales de selección. Una solución muy fácil es la del multiplexor, en la que serán *n* multiplexores los que permitan pasar de varios buses de entrada a uno en particular dependiendo de las entradas de selección (Fig. 1.18). Si hay *k* buses habrá que utilizar como mínimo *s* señales de selección con $k \leq 2^S$ (por ejemplo, si *k* = 3 ó 4, *s* = 2). Los *n* multiplexores tendrán *k* ó más canales de entrada. Por su parte, la señal de control de escritura W del registro deberá activarse con las *k* combinaciones válidas de las señales de selección.

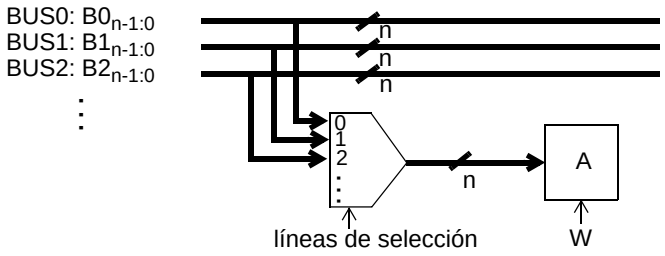


Figura 1.18: Conexión para la lectura desde varios buses

Un mismo bus puede ser leído simultáneamente por múltiples registros sin que existan problemas a nivel lógico. Así, por ejemplo, la Fig. 1.19 muestra un caso en que BUS1 es el único bus que pueden leer los registros A, B, ..., mientras que el registro R también puede leer al BUS2. Si las señales de control poseen el valor indicado todos los registros leen BUS1 sobre el flanco activo de reloj. El único problema que puede existir no es de tipo lógico sino de tipo eléctrico; en concreto, que las líneas BUS1 tengan una carga superior al fan-out máximo en cuyo caso, además de las líneas de conducción, BUS1 deberá incorporar buffers.

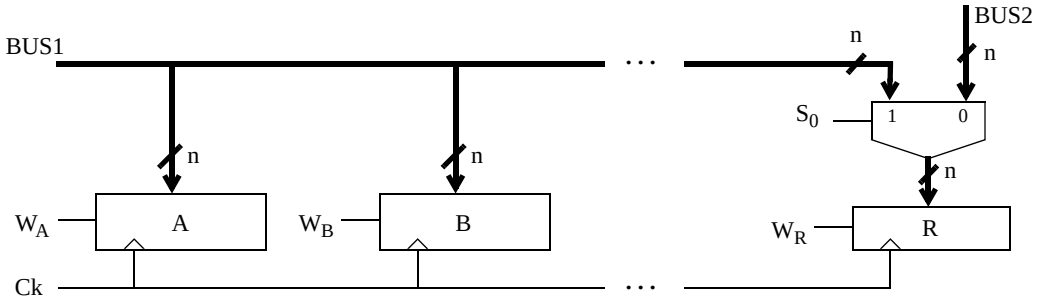


Figura 1.19: Lectura de un bus único por parte de diferentes registros

En cuanto a la operación de escritura en bus, ésta va asociada a la lectura de un registro: el dato almacenado en el registro al ser leído, es el que se escribe en el bus. La operación es

BUS = [REGISTRO]

En general, se pueden hacer consideraciones similares a las de la operación de lectura de buses. Sin embargo hay una diferencia sustancial en cuanto a realizar la operación simultáneamente con múltiples registros: mientras que en el caso anterior (lectura de bus y escritura en registros) no existe problema a nivel lógico, ahora (escritura en bus y lectura de registro) en cada instante sólo un registro puede poner sus datos en el bus para evitar colisiones de datos.

Respecto a buses unidireccionales con un solo registro fuente se dan dos posiblilidades: 1) escritura en un único bus, y 2) escritura en un determinado bus de entre varios (Fig. 1.20). En este último caso, el bus concreto se selecciona mediante un demultiplexor. Dado que la información procede de un único registro de donde se lee, es suficiente con que este dispositivo tenga líneas de salida estándares,

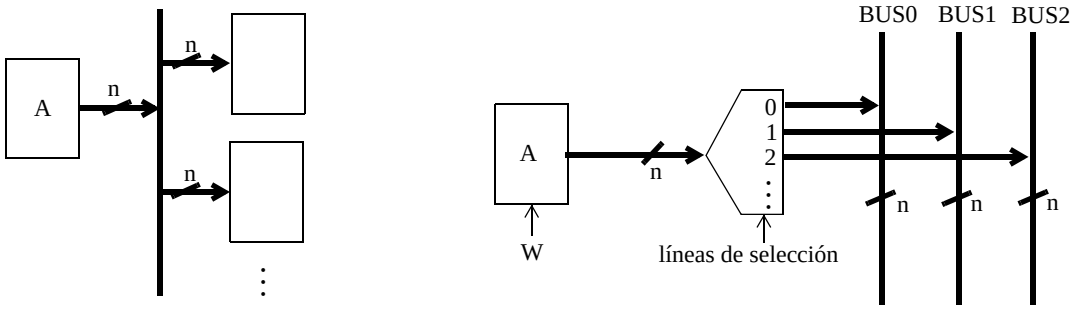


Figura 1.20: Conexión para la escritura en un bus usando buses unidireccionales

sin necesidad de buffers triestado.

En el caso en el que los buses utilizados sean bidireccionales y, en general, cuando haya dos o más dispositivos que puedan escribir sobre el mismo bus, la conexión para poder realizar escritura en un bus se muestra en la Fig. 1.21 . Puede comprobarse que en este caso, como diversos registros pueden escribir en un mismo bus, estos registros necesitan líneas de salidas con buffers triestado, de forma que cuando no se esté leyendo el contenido de un registro éste tendrá su salida en HI y así, no entrará en conflicto con otros contenidos que pueden estar presentes en ese momento en el bus. Por tanto una restricción de operación es que sólo se puede realizar la lectura de un único registro en cada ocasión. Así, en el circuito de la Fig. 1.21 sólo una señal de lectura (R_{A1}, R_{A2} ó R_{A3}) puede estar activa en cada instante.

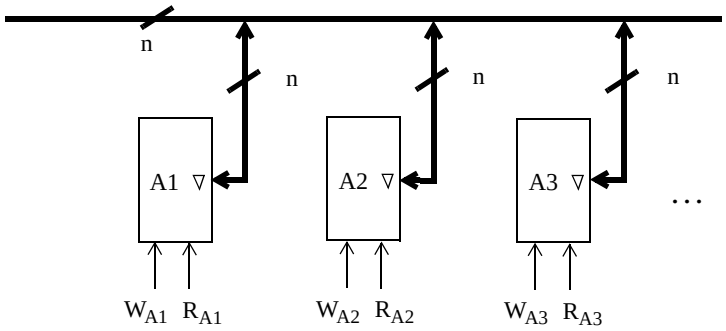


Figura 1.21: Operación de escritura en un bus usando buses bidireccionales

Ejemplo de interconexión:

Se pretende determinar el interconexionado que se requiere para poder intercambiar información entre cuatro registros cualesquiera.

Sean A_0, A_1, A_2 y A_3 dichos registros. A nivel RT, la operación a realizar se puede expresar como

$$A_D \leftarrow A_F$$

donde con A_D se indica el registro destino y con A_F el fuente, y cualquiera de los dos papeles puede ser asignado a cualquiera de los cuatro registros del problema. Para elegir el que actuará como registro destino de la transferencia se dispone de dos líneas D_1 y D_0, de forma que cada una de las cuatro combinaciones binarias de dichas variables selecciona a un único registro. De forma análoga, las líneas F_1 y F_0 determinarán el registro fuente.

Para resolver el problema se presentarán tres estrategias de conexionado distintas, cada una de las cuales relacionará un tipo de bus y un tipo de registro. Las tres soluciones son:

- Solución multiplexada; registros con terminales de entrada I y de salida O separados.
- Solución con bus compartido; registros con terminales de entrada I y de salida O separados pero con salidas triestado.
- Solución con un único bus compartido bidireccional; registros con terminales de entrada/salida I/O.

En la primera solución al problema (Fig. 1.22) se usa un multiplexor a través del cual se realiza la escritura en el bus unidireccional de entrada B_I del dato surgido en la operación de lectura de los distintos registros. Con esta forma de conexionado, los registros vierten su contenido a buses dedicados diferentes, terminando cada uno de ellos en uno de los canales de entrada de los multiplexores. Para elegir el registro fuente las líneas F_1 y F_0 controlarán las entradas de selección del multiplexor. De esta forma, según la combinación binaria que tengan estas señales sólo el contenido del registro apuntado pasará a la salida del multiplexor. Las salidas de éste irán conectadas al bus B_I, a su vez directamente conectado con las líneas de entrada de los registros. Sólo aquel que actúa de registro destino verá activa su señal de control de escritura W_D, con lo que será el único que realizará la operación de lectura del bus B_I y cargará el contenido que en él hubiera. La elección del registro destino la realizará un circuito combinacional, en este caso un decodificador 2:4, de forma que cada una de sus cuatro líneas de salida va conectada a la línea de control de escritura del correspondiente registro.

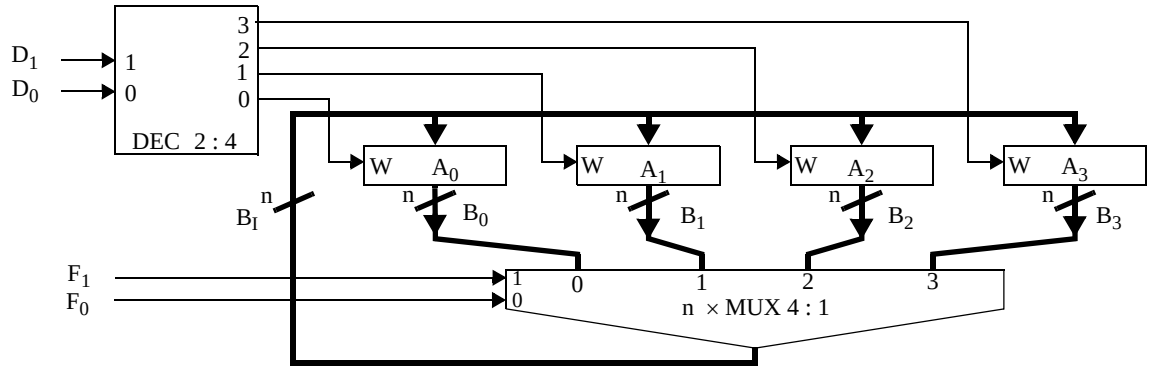


Figura 1.22: Ejemplo de interconexión mediante multiplexado

En la segunda solución que presentamos Fig. 1.23 los registros poseen terminales de entrada I y de salida O separados, estos últimos salidas triestados. Todos los registros vierten su contenido a un único bus interno, B_{INT}, compartido por todos ellos. Debido a este hecho, los registros necesitan disponer de salida tipo tres-estados con la correspondiente señal de control para la lectura. De esta forma, el registro que tenga activa su señal de control para lectura, volcará su contenido en el bus. Si por el contrario tuviera dicha señal desactivada las líneas de salida se encontrarían en estado de alta impedancia no afectando, pues, al dato escrito por el registro fuente. La restricción de que una y sólo una de las señales de lectura de los registros puede estar activa en cada ocasión, se cumple en este circuito al generarse las señales de lectura mediante un decodificador 2:4 de la señales F_1 F_0. De esta forma, según la combinación de estas dos señales, sólo una de las salidas del decodificador se activa. Dado que cada una de ellas se conecta con la señal de lectura R de un registro, se garantiza así que sólo se lee de uno, el que será el registro fuente. De forma análoga al caso anterior, para determinar el registro destino de la transferencia, se usará también un decodificador 2:4 que decodifica el valor de D_1 D_0, determinando así cuál de las señales de escritura W de los registros está activa. Aquél registro que tenga dicha señal activa será el que actúe como registro destino.

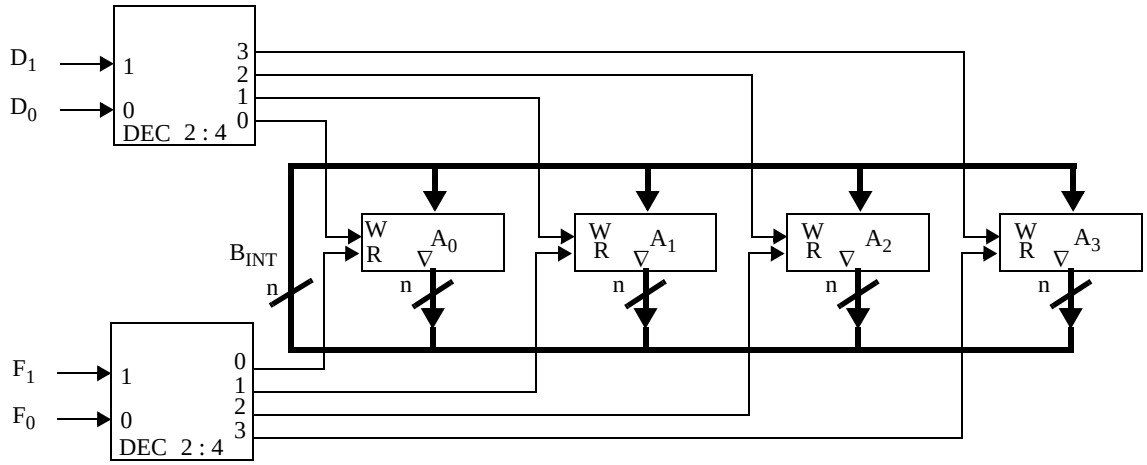


Figura 1.23: Ejemplo de transferencia entre registros con terminales separados de entrada y salida y bus compartido.

Por último, en la tercera solución al problema se utiliza un bus compartido bidireccional. Los registros tienen terminales de entrada/salida y dos señales de control (R, W) para poder realizar adecuadamente las operaciones de lectura y escritura. Al igual que en la solución anterior, dado que el bus es compartido por todos los registros, en cada momento sólo puede realizarse la operación de lectura sobre uno de ellos. Para llevar a cabo la transferencia de información, la elección de los registros fuente y destino se realiza también mediante dos decodificadores 2:4 uno con entradas D_1 D_0 y el otro con F_1 F_0. Con estos decodificadores se controla la activación del grupo de líneas de lectura (para determinar el registro fuente) y de escritura (para seleccionar el registro destino).

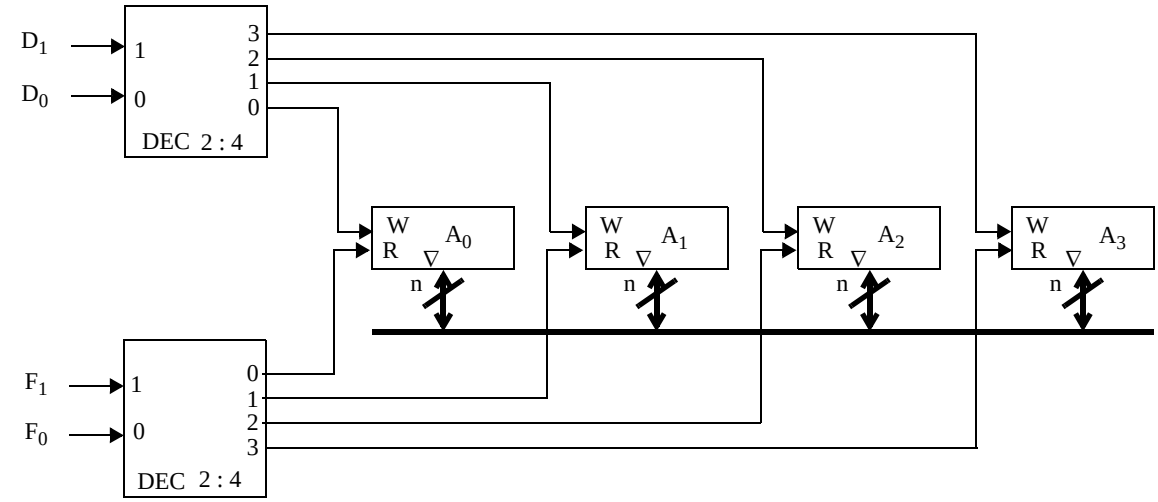


Figura 1.24: Ejemplo de conexión entre registros mediante bus único

1.4 REALIZACIÓN DE SISTEMAS DIGITALES

En este apartado se aborda la tarea de diseño de un sistema digital. Debido a que un sistema digital se organiza mediante la unión de dos grandes bloques, la unidad de procesado de datos y la unidad de control, la tarea de diseño es en realidad doble, una por cada una de estas unidades. En general, cada sistema podrá ejecutar una serie de instrucciones, llamadas también macrooperaciones. Por una parte, la unidad de procesado deberá estar construida posibilitando todas las transferencias entre registros que se necesitan para realizar las operaciones de datos exigidas por las macrooperaciones. Por otra parte, la unidad de control se realizará para que ejerza el gobierno adecuado en la activación de la unidad de datos: dirección de la secuencia, generación de señales, etc.

El diseño de sistemas digitales es una tarea compleja para la que no existe ningún método sistemático eficaz, por lo que la experiencia del diseñador se convierte en un parámetro de suma eficacia, prácticamente decisivo. Al ser éste un capítulo de introducción al diseño de los sistemas digitales, nuestro principal propósito es aportar experiencia de diseño, más que discutir técnicas avanzadas, problemas de optimización, etc. Con el fin de aportar esa experiencia, en el desarrollo del tema utilizaremos un ejemplo sobre el cual se presentarán los principales conceptos de la realización de sistemas. Este ejemplo es simple pero posee gran valor en tanto que se utilizará en éste y en los siguientes capítulos. Además de desarrollar los aspectos concretos del ejemplo, en este apartado también realizaremos una aproximación a la generalización sobre el proceso de diseño.

El punto de partida suele ser una especificación verbal más o menos detallada. Así, nuestro ejemplo de sistema digital será una calculadora de sumas y restas. Este sistema realiza distintas operaciones entre dos datos A y B que están almacenados en sendos registros. El resultado de la operación se almacena en uno de estos dos registros.

El primer paso del proceso de diseño consiste en especificar el sistema a alto nivel. Se trata de definir, con el mayor rigor y formalidad posibles, las características generales y globales del sistema, en concreto:

- cuál es la arquitectura u organización del sistema en términos de “grandes bloques” funcionales.
- cuál es el conjunto de macrooperaciones o instrucciones que el sistema va a entender y a ejecutar. Esto se denomina describirlo a nivel ISP (Instruction Set Processor).
- cuál es el modo de ejecución de esas instrucciones desde la perspectiva del usuario. Aquí se trata de establecer qué es lo que el usuario debe hacer y qué es lo que el sistema realizará automáticamente.

Comenzamos, pues, definiendo nuestro sistema en alto nivel. Para ello presentamos el conjunto de instrucciones (nivel ISP) que, en este caso, será un conjunto de ocho instrucciones distintas (cada una de ellas es una macrooperación).

$A \leftarrow A + B$ $B \leftarrow A + B$ $A \leftarrow A - B$ $B \leftarrow A - B$

$A \leftarrow -A + B$ $B \leftarrow -A + B$ $A \leftarrow -A - B$ $B \leftarrow -A - B$

En nuestro nivel la organización básica de un sistema es la de la Fig. 1.2-b. Además, tras la definición a nivel ISP sabemos que, en nuestro caso, para llevar a cabo este conjunto de macrooperaciones se necesitará una unidad de datos en donde al menos se disponga de los dos registros donde están almacenados A y B. En la Fig. 1.25 se muestra, a nivel de bloques, la organización del sistema.

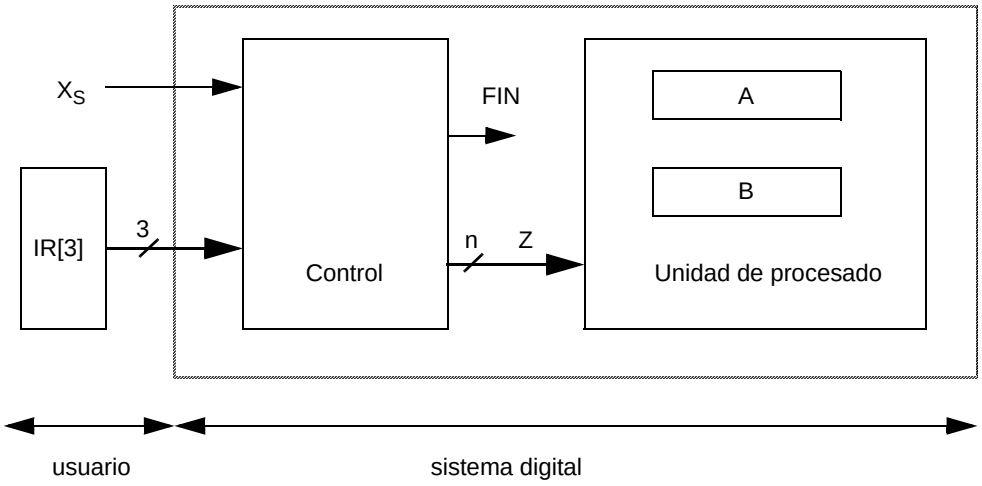


Figura 1.25: Organización del sistema digital ejemplo

A continuación describiremos el modo de operación sistema-usuario. El usuario que opera con el sistema se comunica con él mediante una serie de señales. Con un pulso positivo en la entrada X_S se ordena al sistema que ejecute una de las instrucciones. La selección de la misma se realiza mediante el registro de instrucciones IR (Instruction Register). En nuestro ejemplo (Fig. 1.25), se trata de un dispositivo de tres bits y, dependiendo de la combinación binaria que en él escriba el usuario éste comunica al sistema cuál de las ocho macrooperaciones se ha de realizar. Con esta información el sistema pasa a realizar la secuencia de acciones necesarias para llegar al resultado final. En el momento de finalizar la ejecución se activa la señal de FIN y se regresa a un estado de espera aguardando una nueva señal de comienzo. De esta forma el usuario podría proceder a repetir los pasos dados ($1^{\underline{o}}$, escritura del código en IR; $2^{\underline{o}}$, activación de X_S) para conseguir llevar a cabo una nueva operación.

Para continuar con el desarrollo del sistema, es necesario realizar su especificación a nivel RT, lo que conlleva un doble proceso fuertemente acoplado: por una parte hay que traducir las instrucciones ISP a operaciones de transferencia de datos y, por otra, hay que diseñar una unidad de datos que permita ejecutar todas esas transferencias.

En general, las macrooperaciones realizadas por un sistema digital se llevan a cabo mediante una secuencia de operaciones de transferencia entre registros. El conjunto de transferencias a nivel RT que se realizan en un mismo ciclo de reloj se llama microoperación (μop). Una μop puede consistir en una, en dos, en tres, ..., transferencias entre registros distintas; en el caso límite, en ninguna, llamándose entonces No-OPeración (NOP). Lo que caracteriza a una μop es que se realiza en un solo ciclo de reloj. Con ello, cada macrooperación del sistema será equivalente a una secuencia de microoperaciones, cada una de ellas realizadas en un ciclo de reloj (Fig. 1.26).

El conjunto de microoperaciones de que consta una determinada macrooperación está íntimamente relacionado con la unidad de procesado del sistema. Esto quiere decir que la misma macrooperación precisará más o menos microoperaciones, dependiendo de cuál sea el diseño de la unidad de procesado. Por ejemplo, la macrooperación A $\leftarrow$ A+B del sistema de ejemplo se realizará con cuatro microoperaciones para la unidad que utilizaremos (Fig. 1.28), mientras que en la unidad de la Fig. 1.27 se realizaría con sólo una microoperación.

La relación macrooperación/microoperación/unidad de procesado pone de manifiesto la gran importancia de esta fase del proceso de diseño. Hablando en términos más precisos, cada unidad de

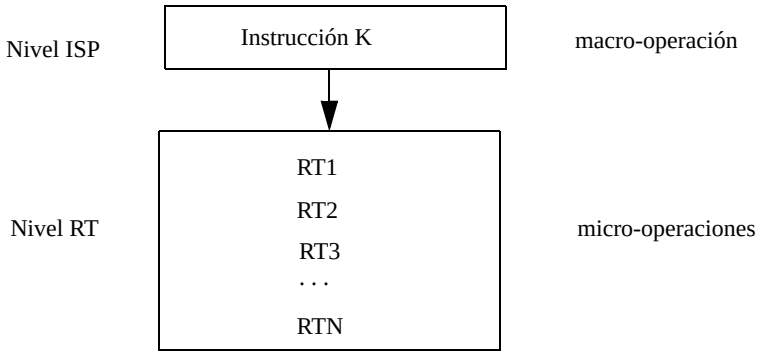


Figura 1.26: Relación entre macrooperación y microoperación

procesado proporciona un conjunto de primitivas RT (que son las microoperaciones realizables en esa unidad) con las cuales hay que obtener el algoritmo de realización de las macrooperaciones. En un proceso de diseño genérico, la conversión desde macrooperación a secuencia de microoperaciones (Fig. 1.26) va imponiendo unos componentes determinados a la Unidad de Datos (para que cada microoperación sea, de hecho, una primitiva RT). En cada instante del proceso, además, el conjunto de primitivas ya disponibles en la Unidad de Datos permite corregir la traducción desde la macrooperación hasta su secuencia de μops. De esta forma, esta fase de diseño obliga a ir teniendo en cuenta tanto el algoritmo de traducción de la macrooperación como el diseño de la Unidad de Datos con componentes RT.

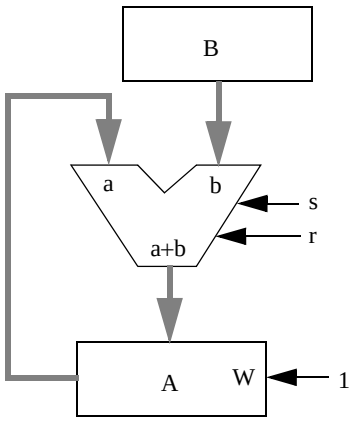


Figura 1.27: Unidad para A ← A+B en una μop

Llevar a cabo adecuadamente esta doble tarea es ciertamente difícil, siendo ésta la fase en la que se necesita gran experiencia. De aquí que en el desarrollo que sigue no afrontemos esta etapa de diseño sino que, sin diseñarla, propondremos una unidad de datos válida para nuestro sistema.

Antes de continuar señalemos otra relación de gran interés que se produce al considerar el dominio temporal y que permite conectar las unidades de datos y de control. El tiempo viene caracterizado por los ciclos de reloj. En cada ciclo, por una parte se ejecuta una μop en la unidad de datos y, por otra, la unidad de control está en un “estado”. Entonces, la funcionalidad de la unidad de control en ese estado aparece de forma clara:

1. durante ese estado deberá activar las entradas de control involucradas en la ejecución de la μop en la unidad de datos;

2. cuando ocurra la transición al siguiente ciclo, debe cambiar al estado que corresponda a la siguiente μop de la secuencia.

La Unidad de Control es, pues, una máquina de estados en la que su secuencia de estados sigue a la secuencia de μops y sus salidas activan las señales de control de los dispositivos de la Unidad de Datos.

Para nuestro ejemplo, la unidad de procesado adoptada será ampliamente descrita y apoyándonos en ella se desarrollará cada macro-operación. La descripción gráfica de nuestra unidad de datos es la presentada en la Fig. 1.28.

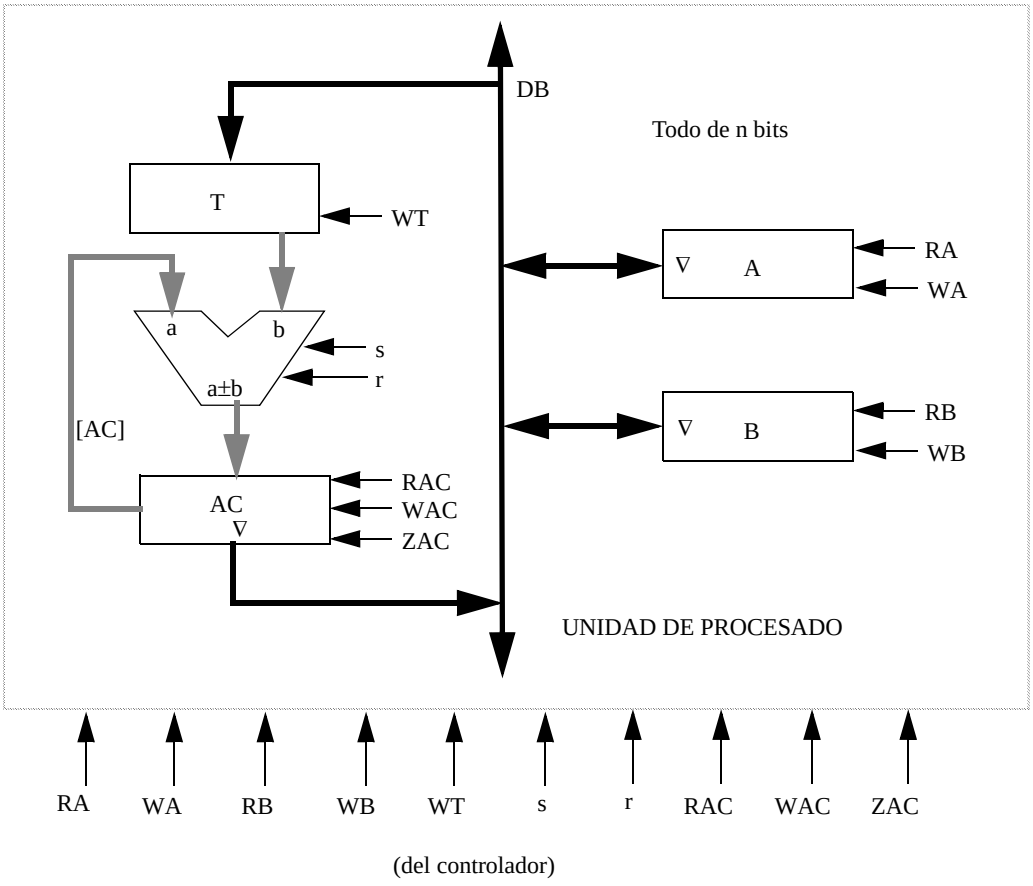


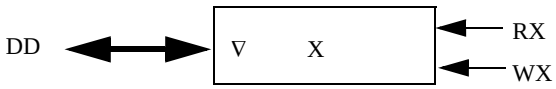
Figura 1.28: Unidad de procesado de una calculadora de sumas y restas

En la Fig. 1.28 pueden apreciarse dos grandes bloques separados por un bus interno de datos (DB) compartido por varios dispositivos. En la parte derecha del bus hay dos elementos de memoria, registros A y B, donde se almacenan los datos de nuestro problema. En la parte izquierda del bus es donde están el resto de los dispositivos que nos servirán para llevar a cabo cada una de las macrooperaciones del sistema. Como se observa, esta subunidad consta de: un registro T que lee datos del bus DB y que suministra uno de los operandos; un sumador-restador entre el contenido del acumulador AC y del registro T; un registro acumulador AC que almacena los resultados de la operación y, en su caso, los transfiere al bus DB. La unidad de procesado dispondrá de dispositivos de *n* bits y, así mismo, todos

los buses tendrán igualmente esa dimensión. En cuanto a los buses, los hay de diversos tipos. Uno de ellos, el bus DB, es un bus compartido que es a través del cual se pasa la información de una parte a otra dentro de la unidad de datos. En la Fig. 1.28 aparece dibujado con trazo fuerte. Los otros, en trazo débil, son buses dedicados cada uno comunicando dos dispositivos entre sí únicamente.

El paso siguiente a la descripción gráfica de nuestro sistema es la descripción detallada de cada uno de sus dispositivos. Esta descripción debe ser lo más formal y rigurosa posible y, para ello, de cada uno se dará su tabla de comportamiento a nivel RT.

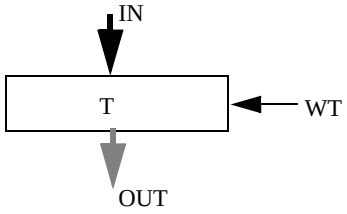
La descripción de los registros A y B se representa en la tabla de la Fig. 1.29. Cada uno dispone de dos señales de control, una de escritura WX y otra de lectura RX, dado que su salida está conectada a un bus compartido por otros dispositivos. Su comunicación con el bus interno de la unidad es mediante un bus bidireccional por lo que la activación simultánea de las líneas de lectura y escritura está prohibida.



RX	WX	X ←	DD =
0	0	X	HI
0	1	DD	entrada
1	0	X	[X]
1	1	Prohibida	

Figura 1.29: Descripción de componentes de la unidad de procesado. Los registros A y B

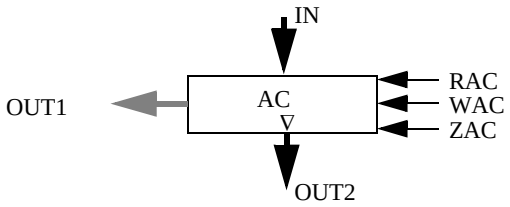
De forma similar se describe el registro T (registro tampón). Como se ve en la Fig. 1.30, el registro T, cuya misión será almacenar datos temporalmente, posee dos buses separados, uno de entrada y otro de salida. El dispositivo dispone de una única señal de control, la de escritura WT, y su contenido siempre está presente en el bus de salida, es decir, es un registro con lectura incondicional.



WT	T ←	OUT =
0	T	[T]
1	IN	[T]

Figura 1.30: Descripción de componentes de la unidad de procesado. El registro tampón.

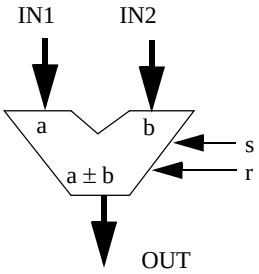
El dispositivo AC es el registro acumulador. Este es el registro en donde se vierten los resultados de las operaciones realizadas por el sumador restador. Como se refleja en la Fig. 1.31, el registro acumulador dispone de un bus de entrada y dos buses de salidas. Uno de ellos, OUT1, es el que vierte su contenido nuevamente a una de las entradas de datos del sumador-restador. Es un bus dedicado, por tanto en él siempre está presente el contenido del acumulador, es decir, el acceso al dato almacenado en AC vía OUT1 es incondicional. El otro bus de salida, OUT2, se conecta al bus interno compartido de la unidad de datos, DB. Por tanto, el registro tendrá una señal de control de lectura, RAC, y sólo cuando ésta esté activa se podrá acceder al dato almacenado en el acumulador a través de OUT2. Las otras líneas de control del dispositivo son las de escritura (WAC) y la de puesta a cero (ZAC). Mediante la activación de esta última señal, se logra hacer un *clear* sobre el contenido del registro, consiguiendo el dato 0 0....0.



ZAC	RAC	WAC	AC ←	OUT1 =	OUT2 =
0	0	0	AC	[AC]	HI
1	0	0	0	[AC]	HI
0	1	0	AC	[AC]	[AC]
0	0	1	IN	[AC]	HI
Otras			Prohibidas		

Figura 1.31: Descripción de componentes de la unidad de procesado. El registro acumulador

Por último describiremos el subsistema sumador-restador (que, en general, podría ser una ALU). Este componente combinacional tendrá dos señales de control, *s* y *r*, que especifican si la operación llevada a cabo es una suma de los datos de entrada o una resta. En la (Fig. 1.32) se muestra su diagrama de bloque y su comportamiento. (Ni el valor sr = 00 ni sr = 11 tienen significado por lo que ambos “no importan”; sin embargo como sr = 11 indicaría que se ordenan dos acciones incompatibles, este valor se “prohibe”).



s	r	OUT =
0	0	No importa
0	1	IN1 – IN2
1	0	IN1 + IN2
1	1	Prohibida

Figura 1.32: Descripción de componentes de la unidad de procesado. La ALU.

Una vez descritos todos los componentes de nuestra unidad de datos del sistema del ejemplo, el siguiente paso será encontrar para cada macrooperación el conjunto de microoperaciones que la realizan. En la Fig. 1.33 se presenta una tabla en donde cada una de las macrooperaciones del sistema es desarrollada en sus microoperaciones. Obsérvese que se ha intentado compartir al máximo las microoperaciones entre las distintas instrucciones del sistema. De esta forma se simplifica la descripción del comportamiento de nuestro sistema como se verá en el siguiente apartado y se reflejará igualmente en un mejor diseño de la unidad de control, lo que se tratará en el siguiente tema.

A modo de ejemplo se comenta detalladamente los pasos dados para desarrollar una macrooperación, en este caso A←A+B. Se pretende cargar en el registro A el resultado de la suma de los datos A y B. Para ello se tiene la necesidad de llevar ambos datos a las dos entradas de la ALU, ya que es ella la que realiza la operación de suma. Una vez realizado esto, se le dirá a la ALU que sume ambos datos llevándose este resultado al registro A que actúa en este caso como registro destino. En la unidad

de datos de nuestro sistema (Fig. 1.28) las líneas de entrada de la ALU provienen del registro tampón T y del registro acumulador AC. Por tanto, serán esos dos registros los que tengan que recibir los datos A y B para que puedan ser sumados. El registro acumulador en nuestra arquitectura sólo recibe los datos de las líneas de salida de la ALU. Por tanto, si queremos que uno de los datos iniciales, por ejemplo A, sea cargado en el acumulador, tendrá que ser como resultado de una operación con la ALU, en nuestro caso, mediante la de suma del dato con cero. Para ello inicialmente habrá que efectuar dos transferencias entre registros:

- poner a 0 el registro acumulador, lo que es posible activando la señal ZAC (Fig. 1.28);
- escribir el dato A en el registro T, lo que es posible leyendo A y escribiendo en T (esto, es activando RA y WT).

Ambas transferencias pueden ser realizadas simultáneamente en el mismo ciclo de reloj y constituyen la primera microoperación:

1. $\mathrm{AC} \leftarrow 0$, $\mathrm{T} \leftarrow \mathrm{A}$

A continuación, como cuando empiece el ciclo de reloj siguiente ya se han efectuado todas las transferencias entre registros ordenadas en el ciclo anterior ya estamos en disposición de realizar una operación de suma con la ALU, dado que los datos que sumará, A y cero, ya están en los registros adecuados. Además, en este mismo ciclo de reloj se puede ir adelantando una nueva transferencia RT consistente en grabar el dato B en el registro T, De esta forma, cuando empiece un nuevo ciclo de reloj ya se tendrá al dato B en el registro T y al dato A en el acumulador. Ambas transferencias pueden realizarse en el mismo ciclo ya que el camino utilizado para cada una de ellas es distinto y constituyen la segunda microoperación:

2. $\mathrm{T} \leftarrow$ B , $\mathrm{AC} \leftarrow$ AC+T

En el tercer ciclo de reloj se efectuará de nuevo una suma con los datos existentes en ese momento en el registro T (dato B) y en el acumulador (dato A), almacenándose el resultado (A+B) en el acumulador. Así, la tercera microoperación es:

3. $\mathrm{AC} \leftarrow$ AC+T

Por último, en el cuarto ciclo de reloj, una vez que ha sido grabado en el acumulador el resultado de la suma de A con B, éste se manda al registro destino, en este caso el registro A. Dado que ambos registros están comunicados por el bus interno DB, la operación a realizar es simplemente un movimiento de datos de un registro a otro. Con ello, se tiene la cuarta y última microoperación.

4. $\mathrm{A} \leftarrow \mathrm{AC}$

De esta forma, en cuatro ciclos de reloj, cuatro microoperaciones, se ha conseguido realizar con esta unidad de datos la primera de las macrooperaciones del sistema del ejemplo. De forma análoga se procedería con las restantes.

Tras haber desarrollado separadamente cada macrooperación en sus microoperaciones, el siguiente paso que hay que dar es ensamblarlas y depurarlas con el fin de obtener un único conjunto de microoperaciones. En su obtención normalmente se persigue compartir el mayor número posible de microoperaciones entre distintas macrooperaciones. De esta forma suele reducirse la unidad de control. En nuestro ejemplo, la Fig. 1.33 ya recoge en buena parte cómo se comparten: por ejemplo la primera μop es común a las ocho instrucciones, en la cuarta sólo hay dos μops distintas (cada una compartida por cuatro instrucciones), etc.

μOP	**A ← A+B**	**B ← A+B**	**A ← A-B**	**B ← A-B**
1	AC←0, T←A			
2	T←B , AC←AC+T			
3	AC←AC+T		AC←AC-T	
4	A←AC	B←AC	A←AC	B←AC

μOP	**A←(-A)+B**	**B←(-A)+B**	**A←(-A)-B**	**B←(-A)-B**
1	AC←0, T←A			
2	T←B , AC←AC-T			
3	AC←AC+T		AC←AC-T	
4	A←AC	B←AC	A←AC	B←AC

Figura 1.33: Conjunto de microoperaciones del sistema del ejemplo

En general, sin embargo, tablas como la de la Fig. 1.33 no reflejan adecuadamente la secuencia de microoperaciones que forman el algoritmo global del sistema. Como veremos próximamente, existen otras formas de representar el algoritmo global, formas que dan una imagen mucho más descriptiva de cómo opera el sistema entero. Esas formas de representación deben permitir, además, describir adecuadamente la secuencia de microoperaciones no sólo bajo la perspectiva de procesamiento de datos, que es como está en la Fig. 1.33, sino también bajo la perspectiva del controlador; esto es, hace falta describir la secuencia en que se han de ir activando las diferentes señales de control de los registros y subsistemas que componen la unidad de datos.

Continuando con el proceso de diseño genérico, una vez descritos los microprogramas de datos y de control, hay que realizar el diseño de las unidades correspondientes. Para ello, en el caso de la Unidad de Procesado de Datos habrá que elegir el hardware concreto que se utilizará. En el caso de la Unidad de Control se aplicarán técnicas de diseño que, en nuestro caso, se tratarán en el siguiente capítulo. El último paso del proceso de diseño consiste en implementar el sistema y verificar que opera correctamente.

Antes de proseguir queremos prestar atención a un aspecto que, aunque siempre importante, lo es mucho más en los Sistemas Digitales. Nos referimos al uso de formas de descripción adecuadas. En los Sistemas Digitales la utilidad es doble:

- en el propio proceso de diseño, ya que lo hace más fácil y directo y, además, reduce mucho el tiempo de localización de posibles errores;
- en la documentación que obligadamente acompaña a sistemas de esta complejidad, ya que las formas de descripción adecuadas constituyen normalmente la parte más inequívoca, precisa y clara de documentación.

Los principales puntos a describir cuidadosamente son: el sistema digital globalmente, la Unidad de Datos (registros, buses) y la de Control, y los microprogramas de datos y de Control.

Todos estos puntos deben ser descritos desde al menos dos puntos de vista:

- estructural o de circuito, esto es, indicando qué componentes tiene y cómo se conectan entre sí;
- funcional o de comportamiento, esto es, explicando qué operaciones realiza y como lo hace.

En esta obra, en la que nos centramos en el nivel RT, se utilizarán esencialmente dos formas de descripción rigurosas:

- Las de tipo gráfico, que a su vez pueden desglosarse en otras dos:
  - Las orientadas al nivel estructural: son los diagramas de bloques y de circuitos tales como los de las Figuras 1.25 y 1.28.
  - Las orientadas al nivel funcional: son las denominadas cartas ASM (Algorithmic State Machine) que presentaremos en el siguiente apartado.
- Las de tipo lenguaje. Generalmente se trata de unos lenguajes de programación, más o menos específicos para describir circuitos y que reciben el nombre de HDL (Hardware Description Language). Nosotros utilizaremos un HDL muy simple, que se explicará más adelante.

Por último, dada la importancia y complejidad de la materia tratada en este apartado, vamos a hacer un resumen sobre el diseño de Sistemas Digitales. En primer lugar, recordar que la experiencia como diseñador es determinante para lograr un buen producto[1] y, a veces, incluso para obtener un sistema que simplemente funcione. De aquí que sea muy importante adquirir experiencia. Incluso con experiencia suficiente, el diseño debe efectuarse bajo las siguientes “guías”:

- Afrontar el diseño siguiendo una metodología. Quizá la mejor metodología sea la denominada *top-down* (desde arriba hacia abajo) que concretamos tras estas guías.
- Dividir con claridad el sistema entre la parte de procesado de datos y la de control.
- Desarrollar la arquitectura de la unidad de procesado y el microprograma (de datos y de control) antes de elegir los dispositivos de hardware concretos.
- Utilizar con rigor las formas de descripción adecuadas y documentar bien todo el proceso.

En cuanto a la aplicación de la metodología top-down al diseño de sistemas digitales, los principales pasos a seguir son:

1. Especificar el sistema globalmente a alto nivel; en concreto,
   - su conjunto de instrucciones (ISP)
   - su modo de operación (uso y ejecución)
2. Para cada instrucción (macrooperación) obtener un algoritmo con primitivas RT (secuencia de microoperaciones) que la realice.
3. Ensamblar y depurar todos los algoritmos de las instrucciones del sistema para obtener:
   - un único microprograma de datos y una Unidad de Procesado que pueda ejecutar todas las primitivas utilizadas en él, (en su caso, regresar al punto 2 para modificar lo que corres-

1. “EN esto , el diseñador puede crear elegancia y belleza o quebraderos de cabeza y caos”. [Pros87].

ponda)
- el microprograma de control asociado.

4. Realizar la Unidad de Procesado y de Control.

5. Implementar y verificar.

1.5 CARTAS ASM

Las cartas ASM(Algorithmic State Machine) son formas de descripción de tipo gráfico especialmente enfocadas, como indica su nombre, a representar algoritmos secuenciales. Se trata de una herramienta prácticamente idónea para la materia que estamos estudiando puesto que:

- El desarrollo de macrooperaciones en microoperaciones es un proceso algorítmico secuencial, por lo que la descripción de Sistemas Digitales a nivel RT cae plenamente dentro de la materia representada con cartas ASM.
- Es una herramienta que da información sobre la estructura y sobre el comportamiento dinámico del sistema que se describe con ella, aspectos ambos de sumo interés.
- La carta ASM proporciona información tanto del algoritmo con los datos como de la secuencia de control, por lo que la propia herramienta está muy próxima a las implementaciones hardware de las Unidades de Datos y de Control.
- Se trata de una herramienta muy intuitiva, fácil de aprender y muy adecuada para trabajar a mano.
- La herramienta tiende un doble puente: 1) hacia niveles de abstracción más bajos, en concreto con los modelos de máquinas de estado que son tan útiles a nivel de conmutación; y 2) hacia niveles más abstractos, como con la representación mediante grafos de flujo de programas a nivel ISP.

El propósito de este apartado es presentar las cartas ASM con un cierto nivel de detalle, con el fin de que el lector pueda utilizarlas adecuadamente de manera inmediata. Para ello, en sucesivos epígrafes mostraremos: sus definiciones básicas, su relación con las tablas de estado (de Moore y de Mealy), diferentes ejemplos de uso en los que se detallarán algunos errores comunes que se cometen al construir cartas ASM, se considerará cómo está contemplada la dimensión temporal en ellas y, por último, regresaremos al ejemplo de sistema digital cuyo diseño iniciamos en el apartado anterior para describirlo con cartas ASM.

1.5.1 Definiciones

Los términos particulares que vamos a definir se refieren a los componentes primarios (o cajas de estado, de decisión y de acción condicional), a la celda básica (llamada bloque ASM) y al ente global (o carta ASM).

Una carta ASM es un grafo orientado y cerrado compuesto por un número variable de bloques ASM interconectados entre sí. Un bloque ASM que es equivalente a lo que se entendía como un estado en las máquinas de estado finito corresponde a todas aquellas acciones que tengan lugar en un mismo ciclo de reloj y sus componentes básicos son:

- caja de estado

- caja de decisión
- caja de acción condicional.

La caja de estado vendrá representada como se muestra en la Fig. 1.34. Se trata de una caja rectangular con un camino de entrada y otro de salida. Dentro de esta caja se especificarán todas las acciones (en nuestro caso, transferencias entre registros o señales a activar) que puedan realizarse en un mismo ciclo de reloj y que no dependan de ninguna condición de entrada. La caja de estado es la caja fundamental en cuanto que identifica a un bloque: cada caja de estados pertenece a un bloque (y sólo a uno) y cada bloque posee una y sólo una caja de estados. Asimismo, cada caja podrá tener asignado un símbolo o un código binario que distinga ese bloque ASM de los restantes.

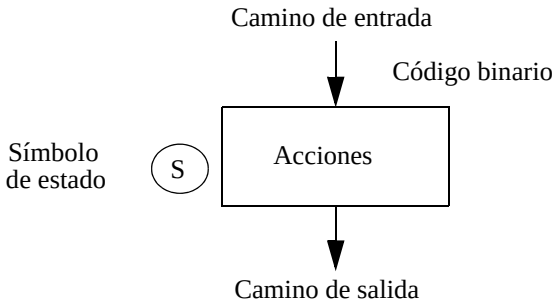


Figura 1.34: Caja de estado

En la caja de decisión (Fig. 1.35) es donde se pregunta, en un sentido lógico, si es verdadera o no cierta condición de entrada que hace modificar el algoritmo del sistema para cada caso, ocurriendo así bifurcaciones en el grafo. Una caja de decisión posee un camino de entrada y varios caminos de salida. Lo usual es que la condición que se interroga posea dos valores (0 y 1) por lo que hay sólo dos caminos de salida. A veces, sin embargo, es aceptable interrogar sobre dos o más variables de conmutación, con lo que el número de caminos de salida puede ser 3, 4, ...; en la Fig. 1.36 se representan algunos ejemplos de estas cajas de decisión junto con sus equivalentes funcionales con cajas de dos salidas.

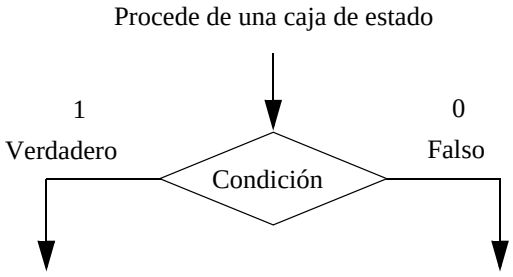


Figura 1.35: Caja de decisión

La caja de acción condicional (Fig. 1.37) contiene las acciones a realizar dependientes de que cierta condiciones se cumplan o no. Estas cajas vendrán a continuación de cajas de decisión, ya que en esta última se pregunta sobre si la condición se cumple o no, y así en cada caso las acciones a realizar podrán ser diferentes. Las acciones contenidas en las cajas de acción condicional son similares a las acciones de las cajas de estado pudiendo ser, por tanto, transferencias RT o activación de señales.

Se entiende por bloque ASM el conjunto formado por una única caja de estado y un número no determinado tanto de cajas de decisión como de cajas de acción condicional. Por tanto, un bloque ASM (Fig. 1.38) poseerá un único camino de entrada y uno o varios de salida. La caja de estado de un bloque

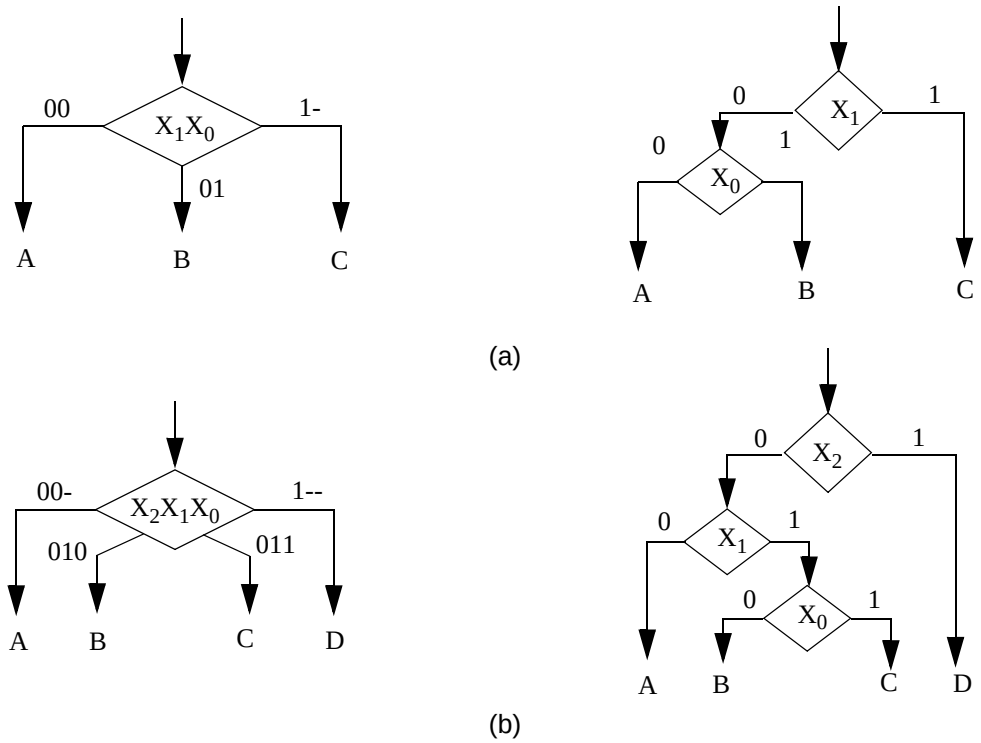


Figura 1.36: Cajas de decisión con 3 (a) y 4 (b) caminos de salida junto con sus equivalentes funcionales con cajas de dos salidas.

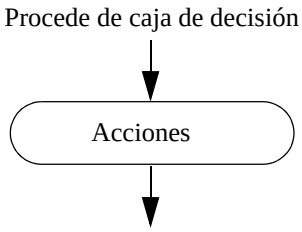


Figura 1.37: Caja de acción condicional

no puede ser compartida por ningún otro bloque. La asociación bloque-caja de estados es, pues, biunívoca. Sí se pueden compartir los otros tipos de cajas por varios bloques, aunque debe hacerse con sumo cuidado pues, además de aumentar la dificultad de la carta ASM, pueden derivarse problemas funcionales y temporales. Todo el conjunto de acciones y de decisiones que tienen lugar dentro de un mismo bloque ASM se desarrollan en el mismo ciclo de reloj. En relación a las acciones tipo señales a activar el convenio que vamos a seguir para simplificar el grafo es que las señales que aparecen como acciones en las cajas de estados o de acción condicional son aquellas que deben de activarse a 1. Aquellas que en el ciclo de reloj correspondiente se mantengan a 0 no aparecerán como acción. Distintos caminos de cajas pueden unirse en un punto de unión (llamado también de acumulación, de interconexión, ...) para originar un único camino. En esta unión el sentido de recorrido de todos los caminos debe ser coherente.

Por último, se denomina carta ASM a un grafo orientado y cerrado que interconecta bloques ASM. Obsérvese que hay dos aspectos que destacar:

1. La carta ASM es un grafo orientado, lo que viene marcado por el sentido de las flechas. El flujo

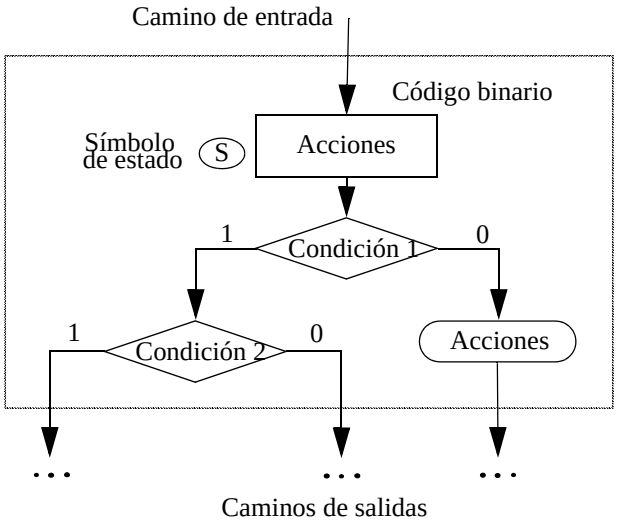


Figura 1.38: Bloque ASM

del algoritmo descrito por la carta sigue esa orientación. Recordando además, que cada bloque ASM representa “lo que ocurre en un ciclo de reloj”, la orientación de la carta al ir viajando de bloque a bloque va representando cómo fluye el sistema ciclo a ciclo.

2. La carta ASM es un grafo cerrado, lo que quiere decir que no existen caminos de entrada a la carta ni de salida de la misma. O, de otra forma, en una carta después de un bloque ASM viene siempre otro bloque ASM. Como el bloque ASM representa el “estado” de una máquina secuencial, el carácter “cerrado” de la carta equivale a que todo estado tiene siempre un próximo estado dentro de un conjunto finito de estados.

La Fig. 1.39.a muestra un ejemplo de carta ASM. Esta carta posee dos cajas de estado (A y B) en ninguna de las cuales se ordena ninguna acción; una caja de acción condicional (C) desde donde se ordena que se active XCICLO; y dos cajas de decisión desde las que se pregunta el valor de LISTO y de YA, respectivamente. Existen dos bloques ASM, uno que corresponde a la caja de estado A y a la de decisión LISTO, y el otro que corresponde a la caja de estado B junto con la decisión YA y la de acción condicional C. Además, hay dos puntos de acumulación, P1 y P2.

Interpretemos el significado de esta carta (Fig. 1.39.a). El sistema se describe en ella sólo en términos de señales ya que no aparece ninguna transferencia de datos. Además de la señal de reloj Ck que siempre está en los sistemas que estudiamos, el sistema posee dos entradas que son las que se interrogan en las cajas de decisión (LISTO y YA), y una salida que es la única señal que hay que activar. Ello da una estructura como la mostrada en el circuito de la Fig. 1.39.b. El diagrama temporal que aparece en ella refleja la operación funcional del sistema. Recordemos que cada bloque ASM representa lo que ocurre en un ciclo de reloj. Comenzando por el bloque A, el sistema mantiene inactiva la señal de salida (XCICLO = 0) mientras interroga a la entrada LISTO. Mientras esta permanezca a 0 se queda en el propio bloque A. Por el contrario, si LISTO pasa a valer 1, en el siguiente ciclo de reloj se ejecuta el bloque B. Estando en éste, se permanecerá en él sin activarse ninguna señal mientras la señal YA esté a 0. Cuando YA vale 1 se activa la salida XCICLO que, por tanto, será 1 sólo durante este ciclo[1]. En el ciclo siguiente, valga YA lo que valga se pasa al bloque A, en el que no se activa XCICLO y del que sólo se sale cuando la señal LISTO sea 1.

Globalmente la Fig. 1.39 describe un circuito secuencial que genera una señal de un ciclo

1. Si la caja C fuera de estado se garantizaría que XCICLO durara en 1 exactamente un ciclo de reloj.

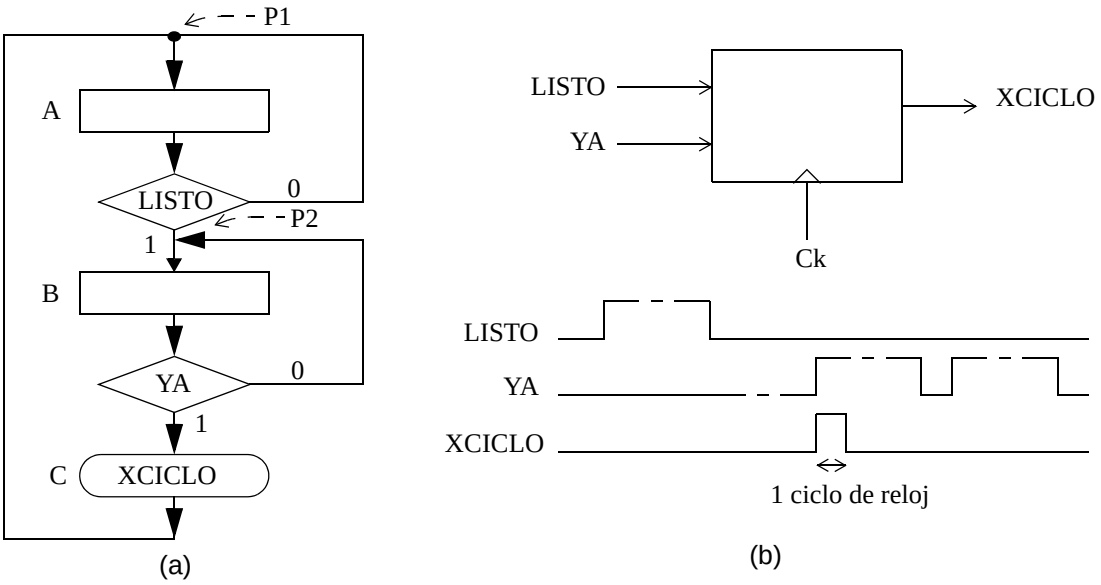


Figura 1.39: (a) Ejemplo de carta ASM. (b) Interpretación.

(XCICLO) mediante la estrategia de prepararla activando LISTO y ejecutarla al activar YA.

1.5.2 Relación entre cartas ASM y tablas de estado

Hasta ahora, la forma de expresar el comportamiento de máquinas secuenciales era a través de diagramas o tablas de estado. Conocido ya el significado de las cartas ASM, es fácil comprobar que con esta herramienta también se puede describir el comportamiento de dichas máquinas. Existe, por tanto, un paralelismo entre una y otra forma de descripción, el cual ahora vamos a presentar (Fig. 1.40).

TABLA DE ESTADOS / SALIDAS	CARTA ASM
Entradas	Cualificadores (condiciones en cajas de decisión)
Salidas	Comandos (acciones)
- tipo Moore	- Acciones en cajas de estado
- tipo Mealy	- Acciones en cajas de acción condicional
Estados	Bloque ASM
- presente	- Bloque actualmente considerado
- próximo	- Bloque al que apunta el camino de salida que esta habilitado

Figura 1.40: Relaciones entre tablas de estado y cartas ASM.

En términos de máquinas de estados finitos, la conducta de un circuito secuencial queda descrita por su tabla (o grafo) de estados/salidas; su equivalente en términos de sistemas digitales es la carta ASM. El circuito se encuentra en cada ciclo en uno de sus estados internos, llamado estado presente y localizado en una de las filas de la tabla de estados; el equivalente es que, en cada ciclo, el sistema se encuentra en un bloque ASM. La situación concreta se completa cuando se conoce el estado (o valor) de entrada en ese ciclo, lo que junto al estado presente constituye el estado total y se localiza en la celda correspondiente (fila de estado y columna de entrada) de la tabla de estados- salidas; el equivalente es conocer el valor de los cualificadores o condiciones de entradas que son interrogados en todas las cajas de decisión del bloque ASM en cuestion. Conocido el estado total de un circuito queda unívocamente determinada su evolución interna al próximo estado y el valor de las salidas generadas por el circuito en ese ciclo; el equivalente ahora es que, conocidos los cualificadores y el bloque ASM, queda unívocamente determinado el siguiente bloque ASM y el valor de las acciones tanto incondicionales como condicionales. Así, lo que eran próximos estados en una tabla de estado será lo equivalente a caminos de salida de cada bloque, las salidas tipo Mealy serán las acciones que se activan en las cajas de acción condicional y las tipo Moore las que lo hacen en las cajas de estado. Por extensión diremos que una acción es tipo Mealy cuando su activación en un determinado ciclo de reloj (bloque ASM) dependa de las condiciones de entrada. Asimismo, por acción tipo Moore entendemos las que se activan dependiendo únicamente del bloque ASM. Por tanto, la activación de acciones de Moore se produce síncronamente con el flanco de reloj y duran todo un ciclo, mientras que la activación de una señal de entrada puede llevar a una activación inmediata de acciones Mealy que se producirán en medio de un ciclo de reloj sin haber cambiado de bloque ASM. La Fig. 1.40 muestra resumidamente las principales relaciones que se han comentado y en la Fig. 1.41 se presenta una tabla de estado y su correspondiente carta ASM.

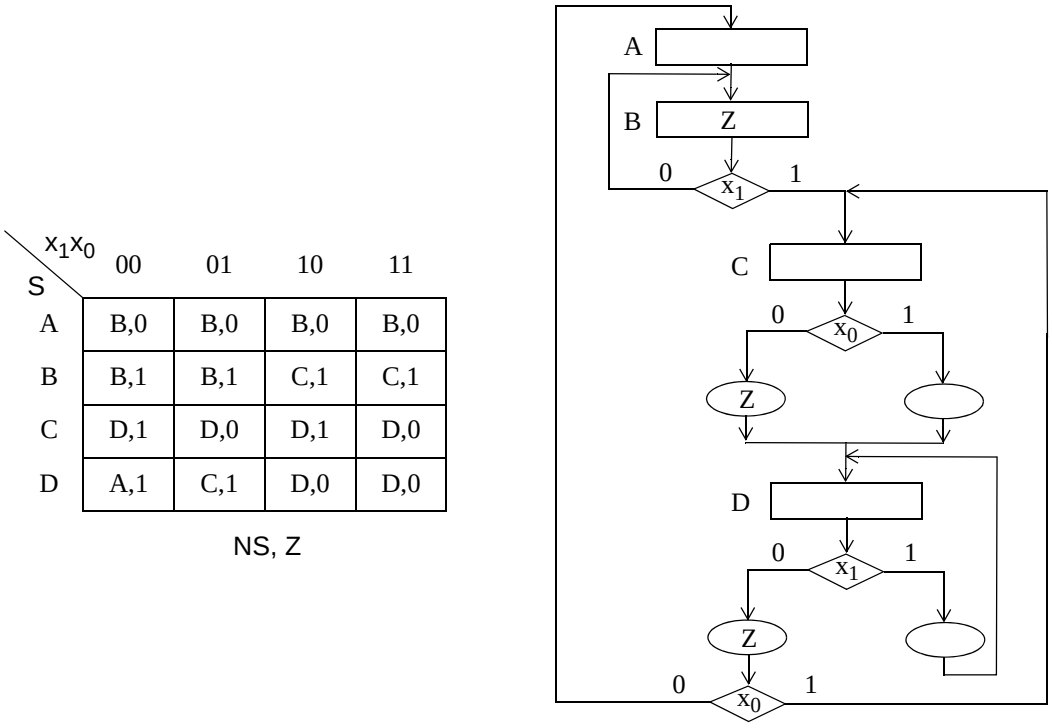


Figura 1.41: Ejemplo de tabla de estados/salidas y carta ASM equivalentes.

Pese a la equivalencia entre tablas de estados/salidas y cartas ASM, ambas son diferentes en cuanto a que cada una de ellas se adecúa mejor a un tipo de aplicación. Así, las tablas de estados/salidas resultan muy útiles para describir máquinas con pocas señales y estados (aunque la secuencia de cambios de estados sea compleja) y para obtener diseños de bajo coste. Por su parte, las cartas ASM son muy intuitivas y útiles para representar procesos secuenciales algorítmicos, para manejar operaciones con datos y para representar sistemas con un gran número de entradas y salidas de las cuales sólo unas cuantas son significativas ("se activan") en cada ciclo.

1.5.3 Ejemplos de cartas ASM

A continuación se presentan una serie de ejemplos de cartas ASM, al final de los cuales se presentan algunos de los errores comunes que suelen cometerse.

➪ Biestable JK (Fig. 1.42).

Un biestable posee dos estados estables, cada uno de los cuales vendrá representado por un bloque ASM. Será la señal de salida *q* la variable que defina el estado del biestable y del bloque ASM. Partiendo de uno de ellos como estado inicial analizaremos las condiciones de entrada que le hacen permanecer en ese estado o cambiar al otro. Si el estado presente es q = 0 y siempre que J = 0 el biestable permanecerá en ese estado en el ciclo de reloj siguiente, por tanto, permanece en el mismo bloque ASM. Sin embargo, si J = 1 el biestable pasa al estado q = 1 en el ciclo siguiente, es decir, cambia de bloque ASM. Estando en este bloque será la entrada K la que decide si permanece en ese estado (K = 0) o si cambia al otro estado en el siguiente ciclo de reloj (K = 1). Expresando este razonamiento en una carta ASM podemos tener:

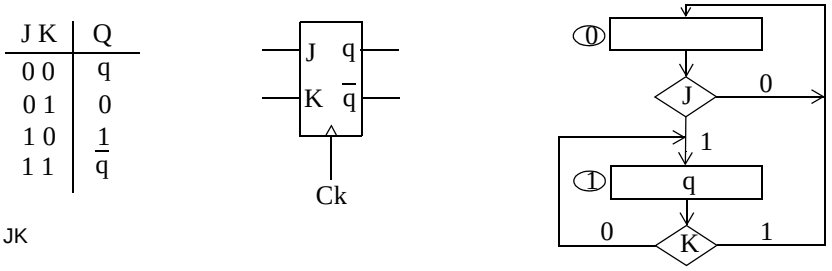


Figura 1.42: Biestable JK

➪ Contadores módulo-8 (Fig. 1.43).

Para describir el funcionamiento de un contador módulo 8 se utiliza una carta ASM con 8 bloques ASM, uno por cada estado de cuenta. En cada caja de estado de cada bloque se irán activando las señales de salida del contador de forma que vayan dando el código de cuenta elegido. En la Fig. 1.43 se muestran las cartas ASM para el contador módulo 8 con código de cuenta en binario ascendente, código Gray y binario descendente.

Dos cosas son de interés. Primero que, al ser una función sin entradas, la carta ASM no tiene ni cajas de decisión ni de acción condicional; el algoritmo es simplemente un ciclo entre los ocho bloques. Segundo que, la acción a realizar en cada bloque coincide con las salidas que se activan en ese bloque; así, en el primer bloque de las cartas "UP" y "GRAY" no se activa nada puesto que el estado de cuenta es el 000, mientras que en el primero de la carta "DOWN" se activan z_2, z_1 y z_0 ya que están representando el estado 111.

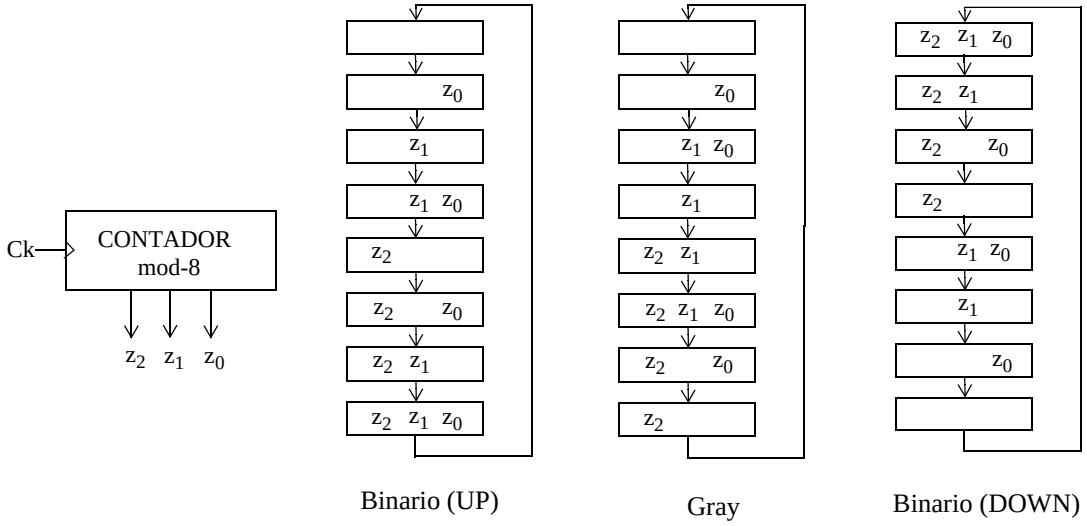


Figura 1.43: Distintos tipos de contadores binarios módulo 8.

⇨ Contador Binario/Gray (Fig. 1.44).

En el ejemplo anterior presentábamos las descripciones de tres contadores módulo 8 distintos, de los cuales uno contaba en código binario y otro en código Gray. En este ejemplo desarrollaremos la carta ASM de un contador módulo 8 que realice su cuenta en código binario ascendente o Gray dependiendo del valor de una línea de entrada G. Se elegirá G = 0 para realizar la cuenta en binario y G = 1 para la cuenta en código Gray. En la Fig. 1.44, se muestran las cartas ASM según el modelo de Moore y según el modelo de Mealy. Obsérvese la carta según Moore, en la que hemos etiquetado los bloques ASM del 0 al 7 para seguir mejor la operación del contador. Si sólo atendemos a los caminos que tienen lugar para G = 0 comprobamos que la secuencia de bloques es 0, 1, 2, 3, 4, 5, 6, 7, 0, 1, ...; esto es, que el circuito descrito opera como contador binario ascendente módulo 8. Si seguimos ahora los caminos para G = 1, la secuencia es 0, 1, 2, 3, 6, 7, 5, 4, 0, 1, ..., que es la que corresponde a un contador Gray. Las salidas del contador no dependen nada más que del bloque ASM presente puesto que se activan en las cajas de estado, por lo que sólo cambian con los flancos activos de reloj.

Analicemos ahora la carta ASM según Mealy. Los bloques ASM se han etiquetado por a, b, ..., h. La secuencia de bloques es en este caso siempre la misma (a, b, ..., h, ..., a, b, ...); lo que cambia según el valor de la entrada G es el conjunto de salidas que se activan. Consideremos, por ejemplo, el estado "c"; en él, siempre $z_1 = 1$ y, si G = 1, además $z_0 = 1$. Esto es, si G = 0 las salidas son 010 y estaríamos en el estado de cuenta 2, que es el que efectivamente sigue al "a" (0) y "b" (1) en un contador Gray; recuérdese que, como el bloque ASM, ocurre en el mismo ciclo de reloj, la activación de z_0 y de z_1 ocurren a la vez, no la de z_1 (caja de estados) antes que la de z_0 (caja de acción condicional). Por último, debe observarse que ahora el valor de las salidas depende no sólo del bloque ASM sino también del valor de la entrada G.

⇨ Generador de señal de comienzo (Figuras 1.45 y 1.46).

En este ejemplo se quiere obtener una señal tipo pulso positivo de duración un ciclo de reloj (Xciclo), a partir de otro pulso positivo pero de una duración mayor (Xlarga). Como se ilustra en la Fig. 1.45 la señal Xciclo está sincronizada con el reloj Ck y se activa cuando hay un pulso en Xlarga sea de la duración que sea.

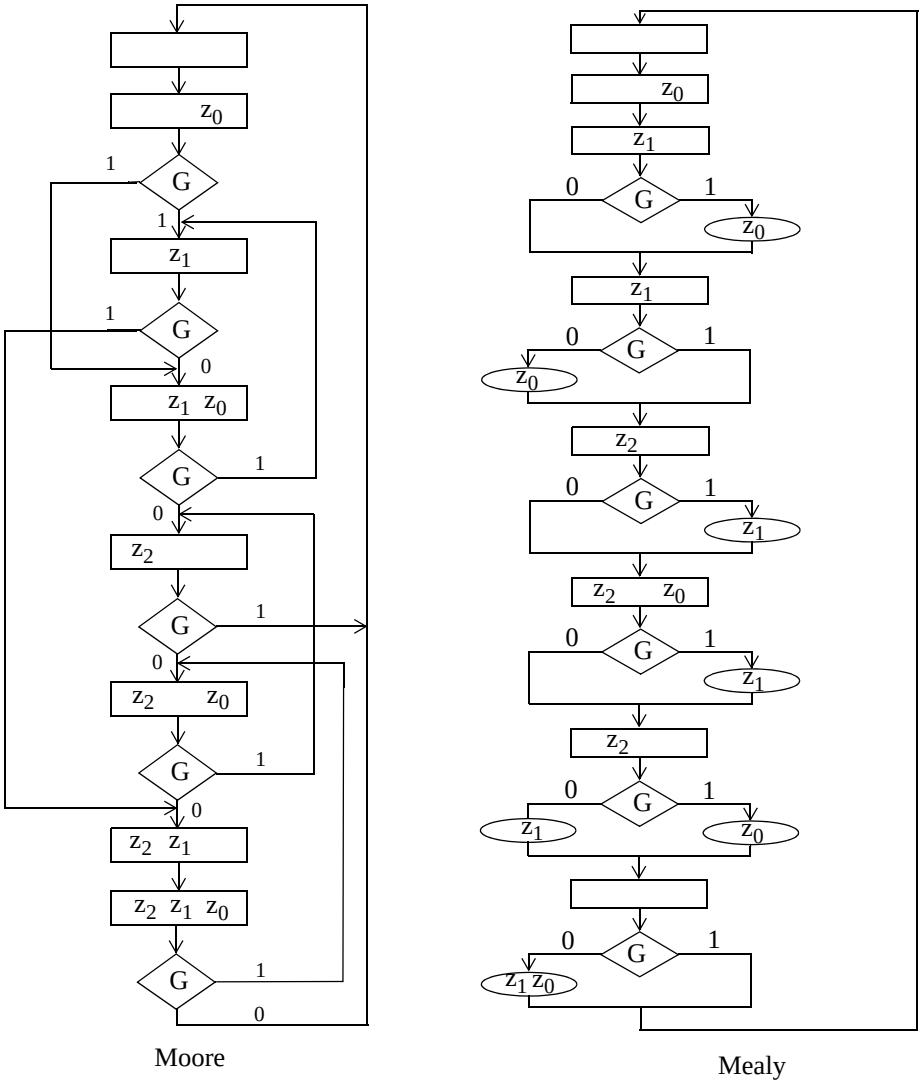


Figura 1.44: Contador ascendente módulo 8 Binario/Gray.

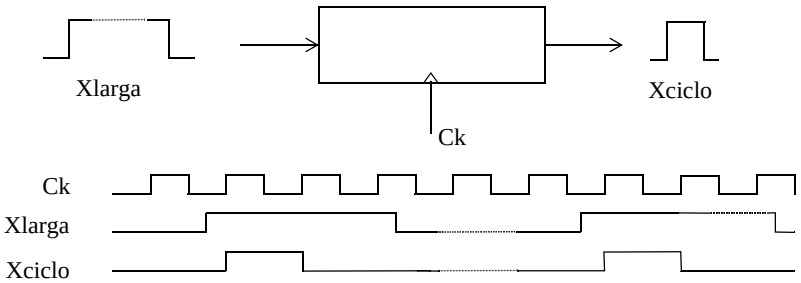


Figura 1.45: Bloque del circuito para obtener un pulso de un ciclo.

El comportamiento del sistema será el siguiente (Fig. 1.46). En el bloque ASM “A” no hay señalada ninguna acción (por tanto la salida del sistema, Xciclo, es 0) pero se interroga por el estado de la entrada Xlarga que, mientras esté a cero, hace que el sistema permanezca en el estado ocioso A. Cuando la señal de entrada cambie a 1, durante el ciclo siguiente al del comienzo del pulso en la

entrada, será la salida Xciclo la que se mantenga a 1. Pasado dicho ciclo la salida deberá volver a cero tanto si la entrada Xlarga sigue en 1 como si ya ha regresado a 0. Para ello, estando en el bloque B se interroga por el valor de Xlarga: si ya vale 0 se va al estado ocioso A y si aún vale 1 se evoluciona al bloque C que proporciona salida Xciclo = 0 y del que sólo se sale cuando Xlarga = 0 para ir al estado ocioso (bloque A). En consecuencia, la salida Xciclo se mantendrá en cero hasta que comience un nuevo pulso en la entrada. Se procederá entonces de la misma forma según se ha descrito.

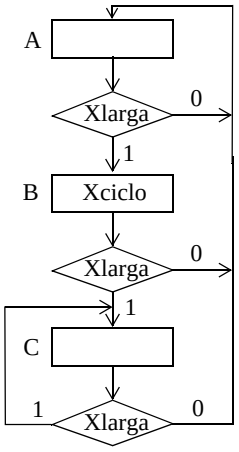


Figura 1.46: Carta ASM del generador de señal de comienzo.

⇨ Generador de ondas programable (Fig. 1.47).

En este problema existe un sistema con dos señales de control S_1 S_0 y una única salida Z que se comportará de forma distinta según el valor de S_1 y S_0. En concreto, la salida Z será periódica pero con distinto periodo y distinta forma de onda, según se muestra en la tabla de funcionamiento de la Fig. 1.47.a.

La carta ASM se muestra en la Fig. 1.47.b. Comenzando por el ciclo marcado como 1 en la Fig. 1.47.a observamos que Z = 1 en las cuatro formas de onda. De aquí que en el bloque ASM 1 se active Z sin interrogar el valor de S_1 y S_0. En el siguiente ciclo Z permanece a 1 salvo que estemos en el caso S_1S_0 = 00; de aquí que en el bloque 2 de la carta ASM no se active nada en la caja de estado y sólo se activa Z bajo la condición de que S_1+S_0 = 1 (donde + es la operación OR, por lo que sólo vale 0 si S_1S_0 = 00). Por otra parte, si S_1S_0 = 00 el próximo bloque es el 1, con lo cual la salida Z tiene dos ciclos de reloj como periodo. En el otro caso, $S_1S_0 \neq 00$, se evoluciona al bloque 3, en el que la salida Z sólo se activa si S_1 = 1 y S_0 = 0, que es lo que marca la tabla de funcionamiento (Fig. 1.47) para el ciclo 3. Además, si S_1S_0 =11, el próximo bloque es el 1 ya que en este caso la salida tiene tres ciclos de reloj como periodo. En los otros dos casos (S_1S_0 = 01 y 10) se pasa por el bloque 4 donde no se activa Z (Z = 0) y desde el que se vuelve al bloque 1 originando un periodo de cuatro ciclos de reloj.

Obsérvese que en el bloque 3 existe un sólo camino para S_1 = 0. Esto incluye el caso deseado (S_1S_0 = 01) pero también a otro caso distinto (S_1S_0 = 00). La razón de incluir este último es que, si el sistema opera bajo S_1S_0 = 00, su secuencia de bloques sería del 1 al 2 y del 2 al 1, por lo que nunca alcanzaría el bloque 3. Así, desde la perspectiva de este bloque, el caso S_1S_0 = 00 nunca ocurre y, por tanto, se convierte en una inespecificación.

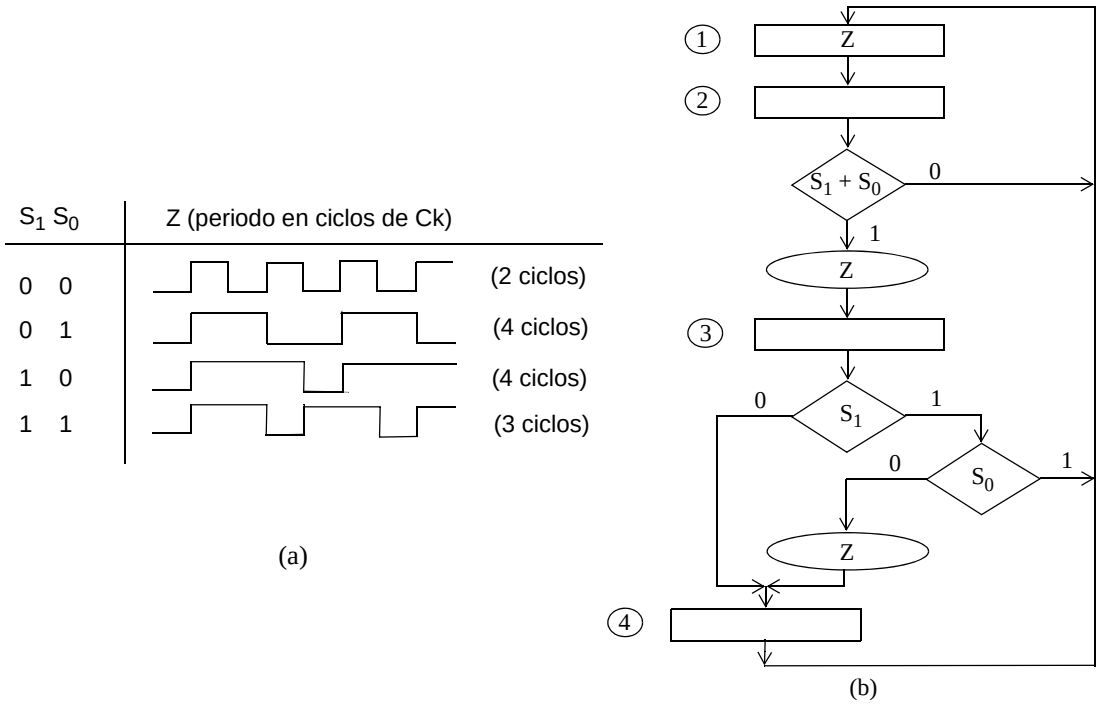
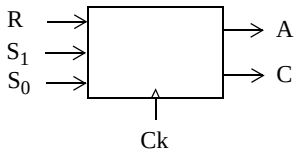


Figura 1.47: Generador de ondas programable.

➪ Máquina expendedora (Figuras 1.48 y 1.49).

Se trata de describir una máquina expendedora de un cierto producto cuyo valor es de 400 pesetas. La máquina admite monedas de 100, 200 y 500 pesetas. En este último caso y sólo en él, la máquina devolverá las 100 pesetas del cambio. En cuanto a las señales que actúan como entradas y salidas de la parte de control serán las siguientes: dos entradas de selección S_1 y S_0 que, en función de los sensores de la máquina, indican la cantidad de monedas (ver Fig. 1.48); otra entrada, R, que se activará en caso de petición de retorno de monedas y anulará el proceso de compra; una salida de acceso, A, que se activa cuando se proporciona el producto porque se ha alcanzado o sobrepasado la cantidad de 400 pesetas; y, por último, una salida C que se activa para devolver el cambio porque se ha recibido de entrada 500 pesetas.



S_1	S_0	Moneda
0	0	Ninguna
0	1	100
1	0	200
1	1	500

Figura 1.48: Máquina expendedora

En la Fig. 1.49 se muestra la carta ASM del sistema que así se comporta. En todos los bloques se interroga la entrada R y, si está activa, el sistema pasa al estado inicial de espera de un proceso de compra (bloque 0). (Suponemos que la propia señal R es la que activa los actuadores de la máquina que le retornan las monedas que haya echado hasta ese momento; en éste ejemplo sólo nos interesamos por la secuencia de control). En este bloque de espera se permanece si no se han echado monedas (S_1S_0 = 00) o se cambia al bloque correspondiente a la moneda que haya sido introducida: 100 (S_1S_0 = 01), 200 (S_1S_0 = 10) ó 500 (S_1S_0 = 11). En este último caso, bloque 500, se activa el acceso al

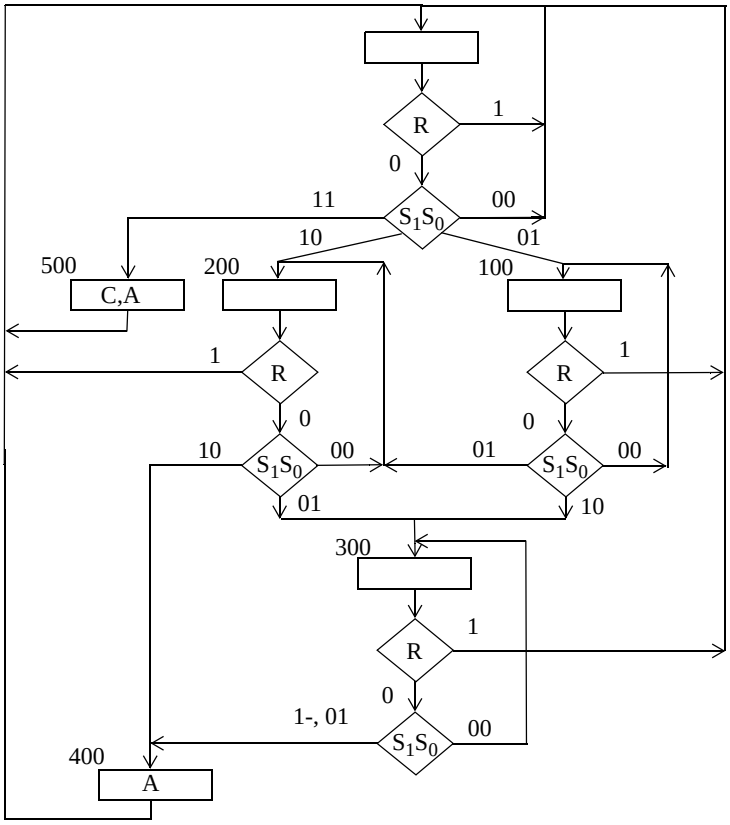


Figura 1.49: Carta ASM para la máquina expendedora

producto y la devolución del cambio C volviéndose después al estado de espera 0. En los otros bloques (100 ó 200) se espera mientras no se introduzcan más monedas (S_1S_0 = 00); cuando se introduzcan se evoluciona al bloque que corresponde en cada caso al valor total acumulado y que puede ser el bloque 200, 300 ó 400. En éste último, bloque 400, se da acceso al producto (A = 1) y se vuelve después al estado de espera (0).

✶ Ejemplos de errores comunes (Figuras 1.50, 1.51 y 1.52)

La construcción de cartas ASM no es una tarea difícil de realizar, pero en ella se pueden cometer algunos errores. Buena parte de los errores más comunes proceden de la diferencia sustancial que existe entre la carta ASM y los organigramas de programación , en cuanto a la temporización en la ejecución del mostrado en el grafo. Así, en la carta ASM cada bloque, aunque posea varias cajas, es un solo estado y se ejecuta en un solo ciclo de reloj. Esto no ocurre con los organigramas de programación, donde el paso de una caja a otra lleva aparejada una secuencia de acciones (primero la primera caja, después la segunda) y donde la ejecución de cada caja se realiza en el tiempo necesario para ello, sin más restricciones. Por otra parte, como segunda gran fuente de fallos de construcción de cartas ASM, está el que las decisiones se tomen sobre funciones lógicas expresadas a veces de forma poco natural, lo que hace difícil detectar algunos errores. A modo de ejemplo presentaremos a continuación algunos errores concretos típicos.

La Fig. 1.50 muestra dos cartas con error. En ambos casos se trata de que, tras el estado A, no está determinado cuál es el siguiente estado que toma el sistema. En la Fig. 1.50(a) este hecho está muy claramente reflejado ya que A apunta tanto a B como a C, pero sólo uno de ellos puede ser el

próximo estado. Más confuso es el caso de la Fig. 1.50(b) donde parece que si x = 0 se pasa a B, si y = 1 se pasará a D y si x = 1 e y = 0 se sigue por C, sin que haya error. Sin embargo, sí los hay: si simultáneamente x = 0 e y = 1 no está determinado si el próximo bloque es B o D; si x = 0 e y = 0 no se sabe si B o C; y si x = 1 e y = 1, se desconoce si C o D.

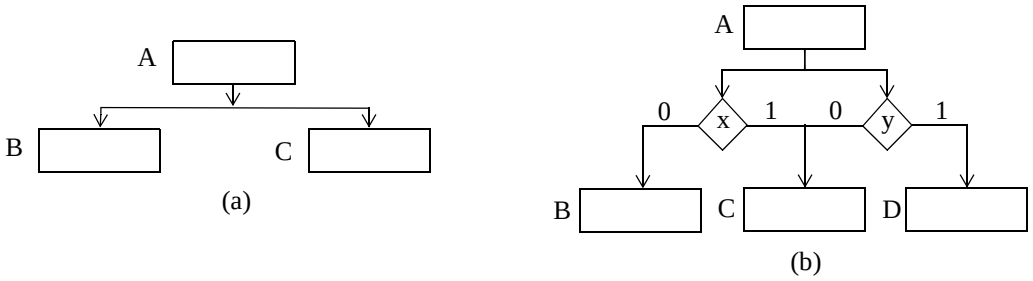


Figura 1.50: Errores en cartas ASM: indeterminación del próximo estado

El error de cerrar un lazo sin que exista una caja de estado dentro del lazo es ilustrado en los dos casos de la Fig. 1.51. En el primer caso Fig. 1.51(a) el camino x = 0 conduce de nuevo a la caja de decisión sin pasar por la caja de estado A, lo que es una situación sin sentido, ya que el sistema tiene que evolucionar siempre de un estado a otro. Lo mismo ocurre en el caso de la Fig. 1.51(b) aunque esta vez sea para x = 1 y el lazo contenga la caja de acción condicional. Para que sea correcto un lazo debe contener al menos una caja de estado.

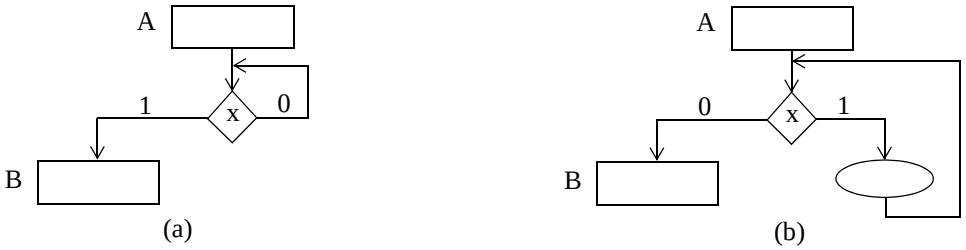


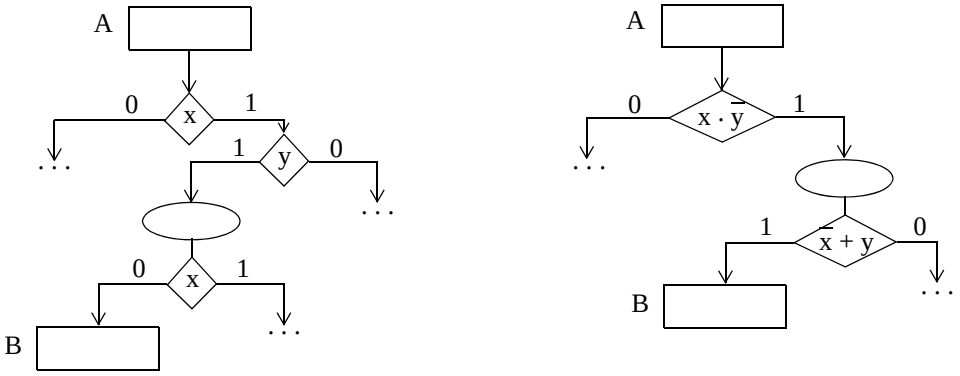
Figura 1.51: Errores de cartas ASM: lazos sin cajas de estado

Al construir una carta ASM todos los caminos que se planteen deben tener la posibilidad de ejecutarse ante alguna combinación de entradas. En caso contrario hay error bien porque ese camino es innecesario, bien porque el constructor de la carta no ha sabido poner bien las condiciones en que se ejecuta. Dos ejemplos de este tipo de error se muestran en la Fig. 1.52. En el primer caso (Fig. 1.52(a)) el camino desde A hacia B es imposible pues exige x = 1 en la primera caja de decisión y x = 0 en la última. Por su parte el segundo caso (Fig. 1.52(b)) también es imposible puesto que $x \cdot \overline{y} = 1$ implica que x = 1 e y = 0, por lo que $\overline{x} + y = 0 + 0$ nunca sería 1.

1.5.4 Consideraciones temporales

En esta sección nos centramos en los aspectos temporales de un bloque ASM ya que, aunque por su gran importancia deben dejarse claros, algunos de ellos ya han sido comentados previamente.

En términos discretos, la dimensión temporal aparece en las cartas ASM a través del sentido en el que se recorren los bloques ASM. Como los sistemas que estudiamos están gobernados por reloj, la unidad temporal es el ciclo de reloj y en cada uno de ellos el sistema estará en uno y sólo uno de sus bloques ASM. El tiempo fluye según se salta de un bloque ASM a otro. (Posteriormente - ver Fig. 1.55



(a) **(b)**

Figura 1.52: Errores en cartas ASM: caminos sin posibilidad lógica de recorrerse

- trataremos la dimensión temporal en términos continuos).

Dentro de un bloque ASM, todas las acciones y decisiones se realizan en el mismo ciclo de reloj. El orden en el que se dispongan las cajas de decisión y/o de acción condicional dentro del bloque es indiferente respecto al orden temporal de ejecución. Para ilustrar esto, a continuación (Fig. 1.53), se muestran dos formas diferentes de expresar un mismo bloque ASM. Esto es, aunque de aspecto diferente, los dos bloques describen lo mismo: en el ciclo en el que el estado sea A se activará la o las señales XT, YT y ZT según estén activas las entradas X, Y y Z respectivamente.

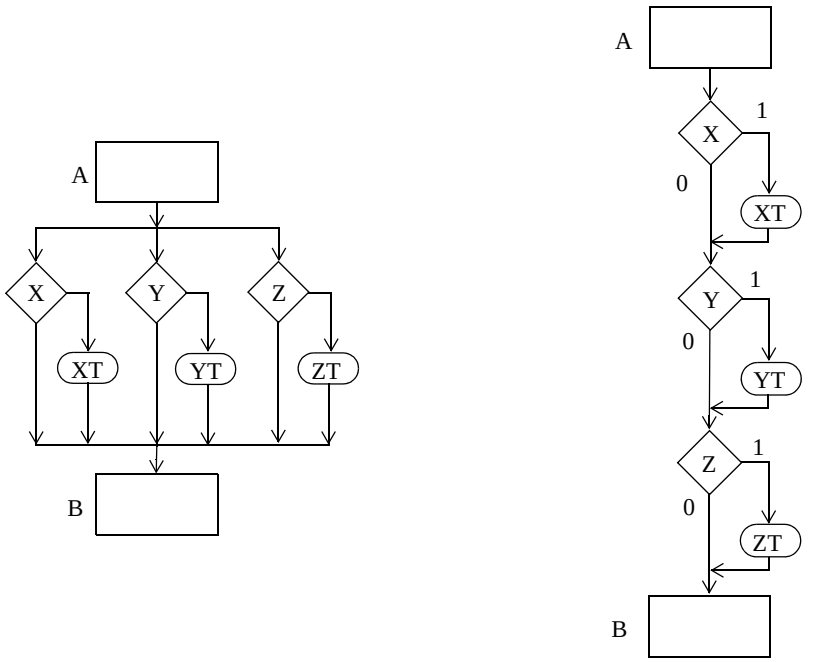


Figura 1.53: Dos formas distintas pero funcionalmente equivalentes de bloque ASM.

Esta es una de las principales diferencias entre las cartas ASM y los organigramas de programación, en donde el orden en el que se plantean las diferentes cajas de preguntas y de decisión es totalmente influyente en el algoritmo que se desarrolla. La secuencia de acciones sigue al flujo de todas las

cajas en los organigramas, mientras que sólo sigue al flujo entre bloques en las cartas ASM.

A continuación nos centraremos en el tiempo como variable continua y distinguiremos entre acciones de Moore y de Mealy en las cartas ASM. Para ello nos apoyaremos en la Fig. 1.54 donde se muestra un determinado bloque ASM. Este bloque pertenece a una carta que podría corresponder a un determinado algoritmo realizado sobre el sistema digital, también representado en la figura. Asimismo se adjunta el comportamiento de las principales señales en el tiempo. Los momentos de cambio de cada una de ellas clarifican la diferencia entre acciones de Moore y de Mealy.

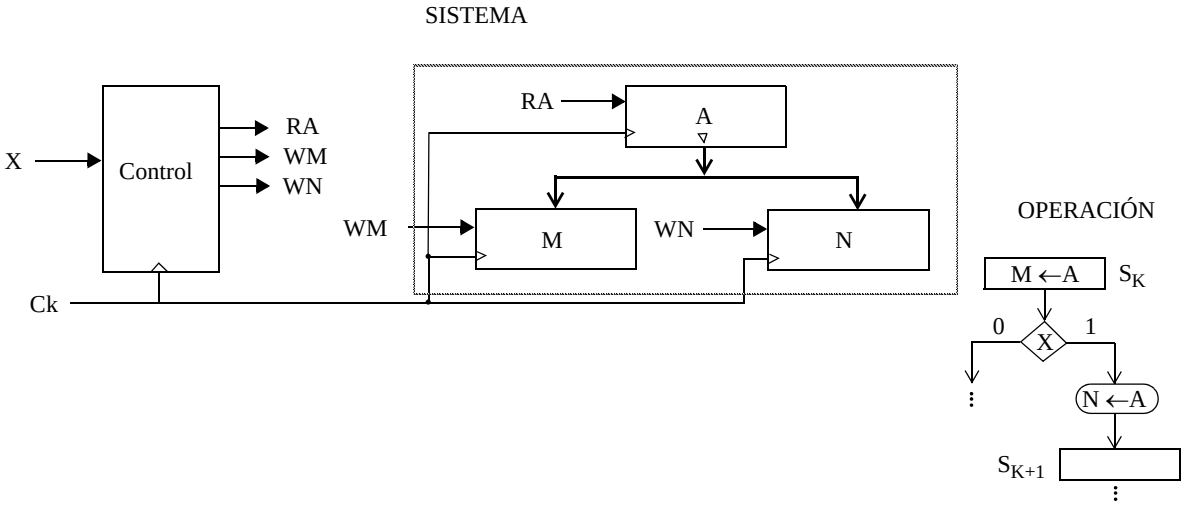


Figura 1.54: Temporización entre las señales de control en un bloque ASM

Se supone que inicialmente los contenidos de los tres registros son A_0, M_0 y N_0 respectivamente. En el bloque ASM bajo estudio y dentro de la caja de estados se ordena la operación de tranferencia entre los registros A y M. Concretamente M se cargará con el contenido de A (M ← A). Al estar ordenada dentro de una caja de estado, esta acción se realiza como acción de Moore, es decir, la señal de lectura RA del registro A y la de escritura WM sobre el registro M deberán estar activas durante todo el ciclo de reloj S_k con independencia de los cualificadores de entrada.

Por otra parte, en ese mismo bloque ASM que transcurre durante el ciclo S_k del reloj, se realizará la acción condicional de transferencia de datos entre el registro N y el A, es decir N ← A, sólo si la línea de entrada X está activa (X = 1). Desde el punto de vista de las señales, para llevar a cabo esta transferencia la señal de escritura WN del registro N debe activarse si estamos en el estado S_k y la entrada X es 1. Se trata, pues, de una acción de Mealy. En la Fig. 1.55, se muestra la temporización de las señales de control de lectura y escritura de cada uno de los registros, apreciándose que WN espera a que X sea 1 para activarse en el ciclo S_k (línea a trazos), mientras que ocupa activa todo el ciclo S_k si X ya se encontraba activa (línea continua en la figura).

Resumiendo, la transferencia llevada a cabo en la caja de estado se realizará siempre que estemos en ese bloque, independientemente de cualquier condición sobre las entradas. Sin embargo, la transferencia en una caja de acción condicional, aun estando en el mismo bloque ASM, sólo se llevará a cabo si se cumple la condición. Respecto a las señales de control, las involucradas en la transferencia de cajas de estado se activan durante todo el periodo de reloj (por ejemplo, RA y WM), mientras que las de cajas de acción condicional requieren además que se cumpla la condición (por ejemplo, WN) por lo que su duración en el tiempo depende de cuándo se cumple la condición de entrada.

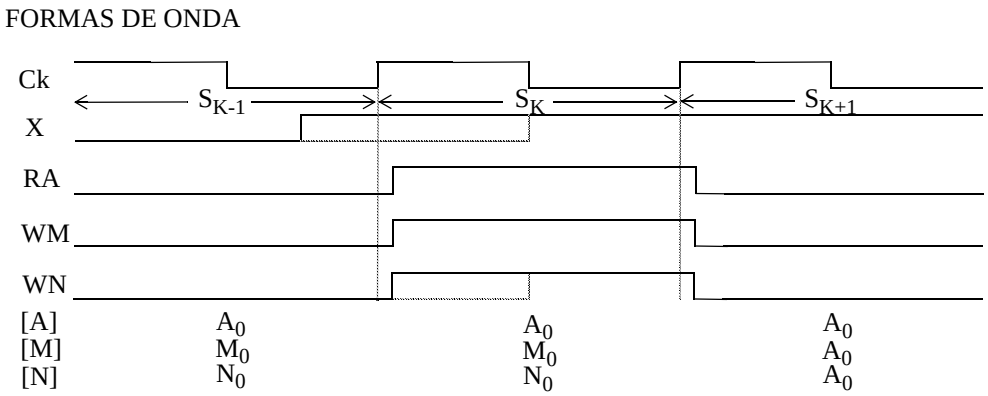


Figura 1.55: Temporización entre las señales de control en un bloque ASM (cont.)

Por último, observemos cuándo se produce un cambio de datos. Si nos basamos en la Fig. 1.55, la señal de lectura del dato fuente (RA) se activa recién comenzado el ciclo S_k, por lo que el dato A_0 contenido en A se situará en su bus de salida y aparecerá en las entradas de M y de N un cierto tiempo después. Este tiempo depende del retraso con que la señal RA generada por el control aparezca en la entrada del registro A, del retraso de propagación en este registro (desde que RA = 1 hasta que el dato A_0 aparece en la salida) y del retraso en la propagación por el bus de conexión ente A y las entradas a los registros destino (M y N). Este tiempo está acotado y, en general, suele despreciarse. En el mismo ciclo S_k, la unidad de control ha activado las señales de escritura en los registros destino (WM = WN = 1) de forma que, cuando llega el flanco de subida que separa el ciclo S_k y el S_{k+1}, estos registros cargan en paralelo el dato de entrada (que es A_0) . Así, con un tiempo de propagación de retraso, el valor A_0 es almacenado en M y N durante el ciclo S_{k+1}. Esto es, la transferencia de datos se ordena en un ciclo y queda ejecutada en el siguiente.

1.5.5 Carta ASM del ejemplo

Una vez conocida la teoría sobre cartas ASM, su estructura, temporización, etc. aplicamos este conocimiento para obtener la carta ASM del ejemplo que se presentó en un apartado anterior.

Se trata de un sistema digital que opera como calculadora que suma/resta dos datos de entrada A y B y el resultado lo carga en A o B según cada caso. A continuación mostraremos la tabla de funcionamiento para cada valor de las variables de selección.

IR_2 IR_1 IR_0	MOP	IR_2 IR_1 IR_0	MOP
0 0 0	A ← A + B	1 0 0	A ← - A + B
0 0 1	B ← A + B	1 0 1	B ← - A + B
0 1 0	A ← A - B	1 1 0	A ← - A - B
0 1 1	B ← A - B	1 1 1	B ← - A - B

Para cada combinación de las variables de selección se realiza una macrooperación distinta. Cada una de ellas se llevará a cabo con una serie de microoperaciones como ya habíamos visto anteriormente. A continuación se repetirá el conjunto de microoperaciones para que nos sirva de ayuda para

obtener la carta ASM.

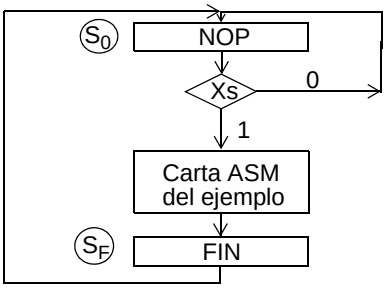
μOP	A ← A + B	B ← A + B	A ← A - B	B ← A - B
1	AC ← 0, T ← A			
2	T ← B, AC ← AC + T			
3	AC ← AC + T		AC ← AC - T	
4	A ← AC	B ← AC	A ← AC	B ← AC

μOP	A ← - A + B	B ← - A + B	A ← - A - B	B ← - A - B
1	AC ← 0, T ← A			
2	T ← B, AC ← AC - T			
3	AC ← AC + T		AC ← AC - T	
4	A ← AC	B ← AC	A ← AC	B ← AC

Toda carta ASM tendrá un bloque ASM inicial en donde se espera que cierta señal de comienzo Xs se active. Cuando eso ocurra, se comenzará a pasar por cada uno de los bloques ASM que componen el algoritmo en sí de nuestro sistema digital, para nuestro ejemplo, el conjunto de microoperaciones que en su totalidad conforman cada una de las 8 macrooperaciones de la calculadora bajo estudio.

De igual forma, todas nuestras cartas ASM acabarán con un bloque ASM dedicado a la activación de una señal de FIN, que le indique a la unidad de datos que la operación solicitada se efectuó y se volvería al bloque ASM inicial en espera de una nueva activación de la señal de comienzo.

Esquemáticamente, la carta ASM quedaría:



Centrándonos en nuestro problema, tenemos que desarrollar la parte central de la carta anterior. Para ello, nos iremos fijando en la tabla previa en donde se expresaban cada una de las microoperaciones. Cada ciclo de reloj corresponderá a un nuevo bloque ASM, y dentro de cada uno de ellos iremos colocando las distintas transferencias entre registros que debamos ir realizando. Todas aquellas comunes para todas las macrooperaciones, se colocarán dentro de la caja de estado del bloque correspondiente, y aquellas que sólo se realicen en unos casos si y en otros no, se realizarán como acciones condicionales, es decir, previamente, se preguntará en una caja de decisión por las variables de selección necesarias para diferenciar una macrooperación de otra, y en cada caso se hará efectiva una microoperación determinada.

Aplicando todo ello a nuestro ejemplo podemos llegar a la carta ASM de la Fig. 1.56.

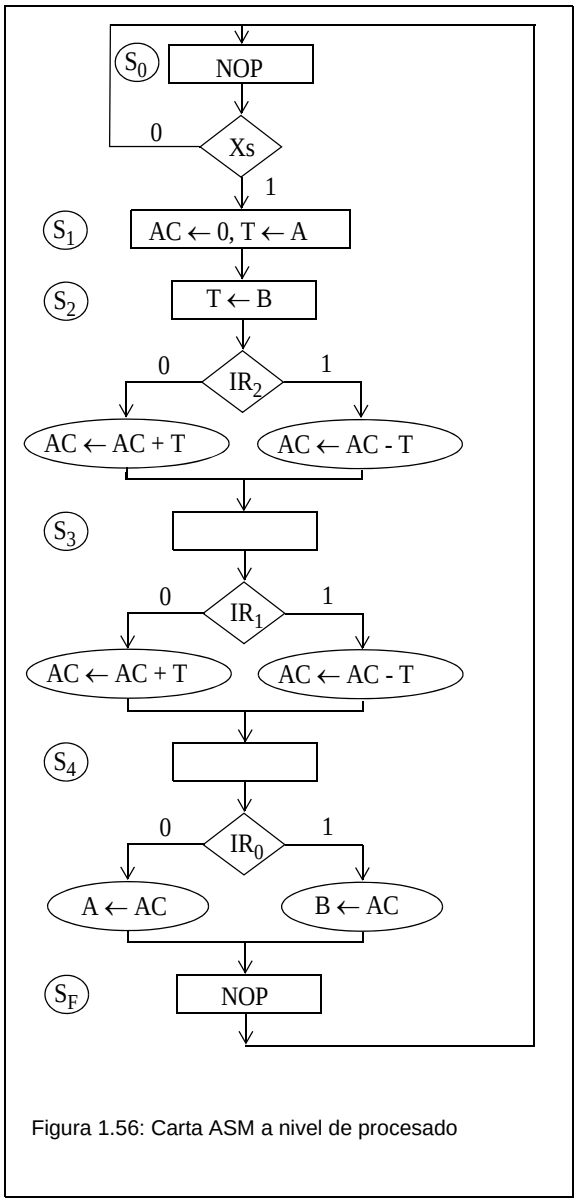


Figura 1.56: Carta ASM a nivel de procesado

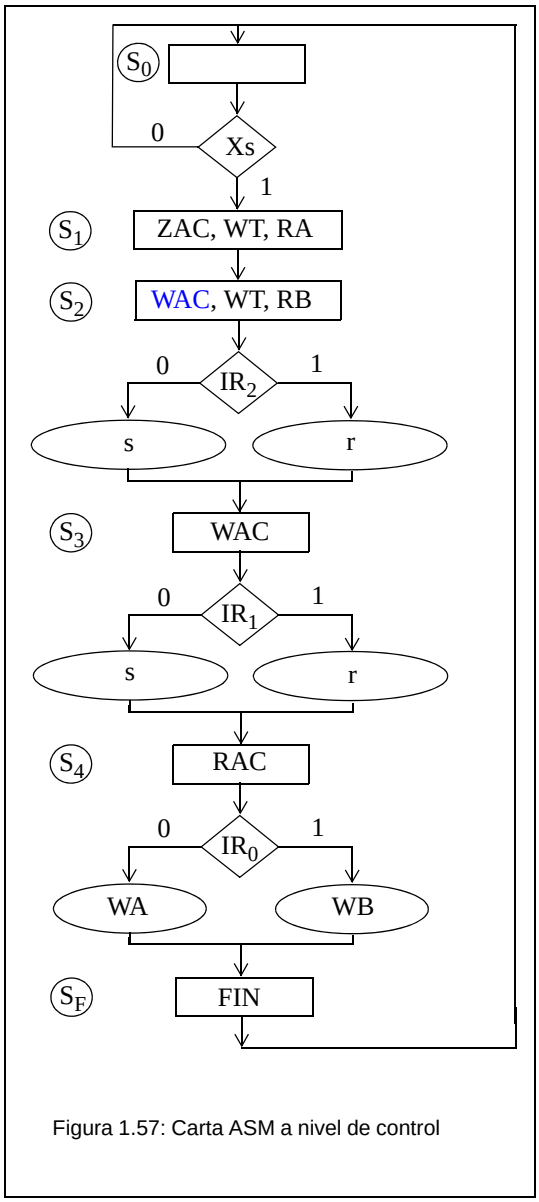


Figura 1.57: Carta ASM a nivel de control

Esa carta ASM muestra el conjunto de transferencias entre registros que se necesitan realizar para cada una de las macrooperaciones de nuestro sistema. Además de esta carta podríamos hacer una carta ASM relacionada con la anterior en donde en vez de expresar a nivel RT las acciones necesarias a realizar en cada ciclo de reloj (o bloque ASM), lo que señalamos es el conjunto de líneas de control que deben ser activadas por el controlador para que se puedan llevar a cabo cada una de las transferencias de la carta anterior. A esta nueva carta se le llamará la carta ASM de control y para nuestro ejemplo tendrá la forma de la Fig. 1.57.

Considéres la carta ASM de control anterior. Si nos fijamos en el bloque ASM denominado S_3 de la carta a nivel de transferencia entre registros (Fig. 1.56), se comprueba que las acciones a tomar en ese ciclo de reloj son acciones de Mealy, puesto que dependiendo del valor de la señal de control IR_1 se realizaba una transferencia (AC ← AC + T) u otra (AC ← AC - T), quedando la caja de estado vacía puesto que no había en ese ciclo ninguna acción tipo Moore (independiente de las líneas de control).

Cuando se pasa a la carta ASM de control (Fig. 1.57), si nos fijamos en ese mismo bloque ASM, para cualquiera de las dos transferencias que tienen que realizarse hay señales comunes que debes ser activadas. Ese conjunto de estas señales se pueden incluir en la caja de estado pues su activación resulta independiente de la señal de control IR_1. Asimismo, con esta alternativa se consiguen simplificaciones en el diseño final de la unidad de control cuando se aplique una de las estrategias para pasar de la carta de control al diseño del controlador.

1.5.6 Unión entre cartas ASM

En este punto del tema trataremos distintas formas de asociar o unir distintas cartas ASM donde cada una de ellas por separado, desde su principio a su fin, realizarán una tarea determinada que no es de nuestro interés en este apartado.

Se disponen de dos cartas ASM que representan cada una de ellas el funcionamiento de dos máquinas cualesquiera llamadas A y B. Asímismo, se tienen dos señales CLA y CLB que sólo cuando tengan valor lógico 1 se permitirá el desarrollo de la secuencia de microperaciones que posea cada carta ASM respectivamente. Con esta introducción nos planteamos tres formas de asociar cartas ASM:

a) Asociación Serie:

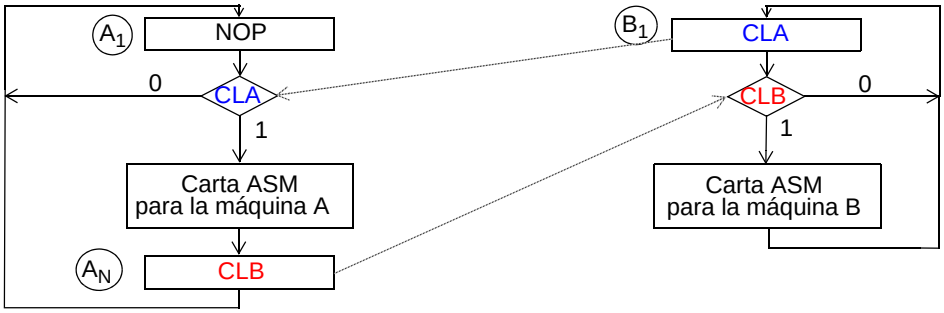


Figura 1.58: Asociación de cartas ASM en serie

La Fig. 1.58 muestra la asociación serie. En esta carta ASM se puede comprobar que, partiendo del bloque ASM A_1, una vez que se active la señal CLA se procede al desarrollo de la carta ASM de la máquina A. Una vez que éste ha concluido se activará la señal CLB permitiendo así que comience entonces la otra carta, la correspondiente a la máquina B. De igual forma, a su finalización se activará nuevamente la señal CLA volviendo al desarrollo de la primera. Como puede verse, nunca las señales CLA y CLB están activas simultáneamente, de ahí que una carta de paso a la otra y se ejecuten en "serie".

b) Asociación Parelelo:

En este caso, mostrado en la Fig. 1.59, una vez que se hayan activado simultáneamente las señales de control CLA y CLB, se da paso a ejecutar al unísono ambas cartas ASM. Es por tanto un proceso en el que ambas máquinas trabajan en "paralelo".

c) Subrutinas:

Para explicar este punto, suponemos que la carta ASM de la máquina A desarrolla un programa

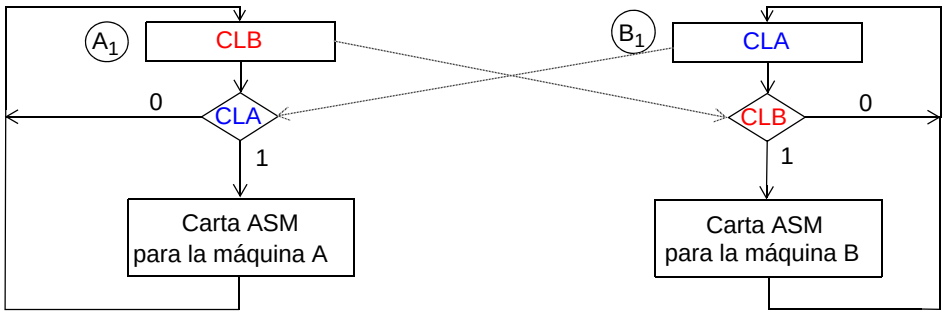


Figura 1.59: Asociación de cartas ASM en paralelo

principal, y es la correspondiente a la máquina B la que desarrollará un programa al que nos referiremos como subrutina. En general, en el desarrollo del programa principal se irán realizando distintas llamadas a la subrutina, se desarrollará esta, y una vez finalizado su conjunto de acciones se volverá al programa principal en aquel punto en donde nos habíamos quedado cuando se llamó a la subrutina. Se seguirá avanzando por el programa principal hasta que nuevamente puedan aparecer nuevas llamadas a la subrutina. Esta explicación queda representada en la Fig. 1.60.

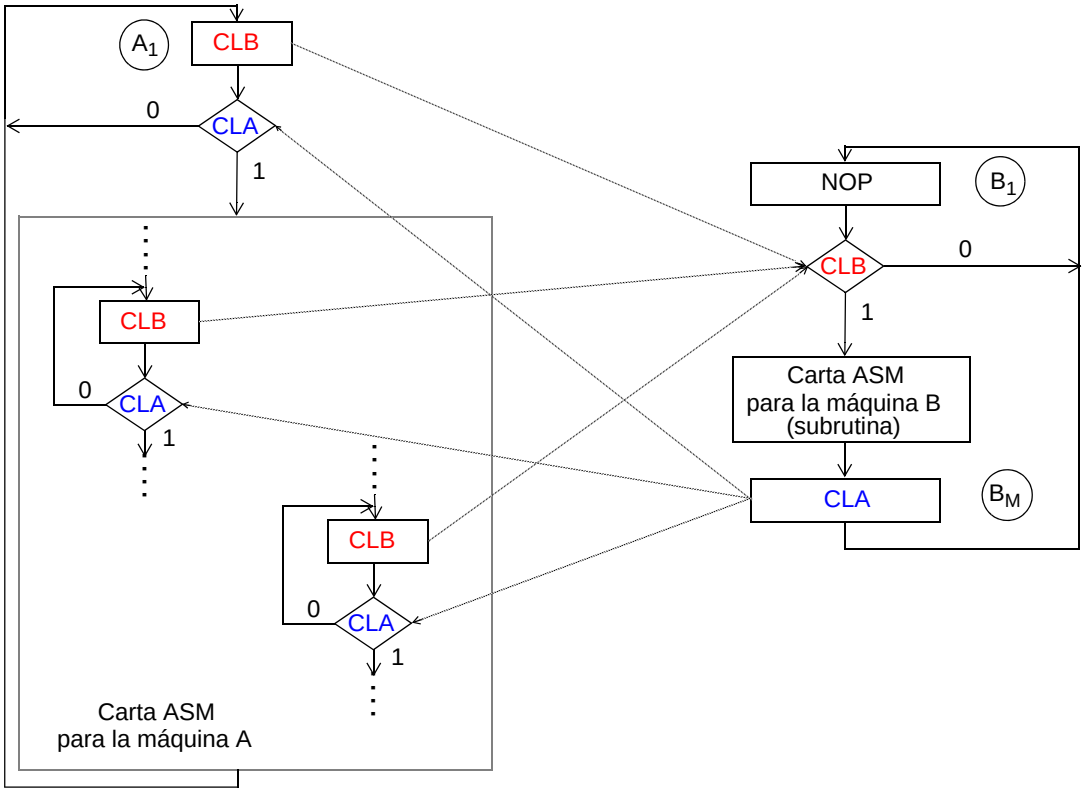


Figura 1.60: Asociación entre cartas ASM. Subrutinas

1.6 LENGUAJE DE DESCRIPCIÓN DE HARDWARE (HDL) SIMPLIFICADO

Hasta ahora, la descripción del funcionamiento de un sistema digital se ha expresado mediante la carta ASM. Se trata de una forma gráfica de referenciar cada una de las microoperaciones que desarrolla el sistema digital. A continuación presentamos una alternativa a la carta ASM donde la descripción se realiza en forma de programa: los lenguajes de descripción de hardware (*Hardware Description Languaje*, HDL). En nuestro caso presentaremos un lenguaje muy simplificado, ya que los lenguajes estándares de descripción de hardware tienen una excesiva complejidad para el fin al que queremos llegar en este texto.

1.6.1 Descripción del HDL

En el lenguaje HDL simplificado que vamos a presentar, cada instrucción se ejecuta en un ciclo de reloj. El formato general de la instrucción es el que se muestra:

N	$f_0(x)$	T_0/z_0	N_0
	$f_1(x)$	T_1/z_1	N_1
	.....		
	$f_{n-1}(x)$	T_{n-1}/z_{n-1}	N_{n-1}

El primer campo que aparece es un número natural N y corresponde al número de la instrucción en la que estamos. El segundo campo que aparece está constituido por una serie de funciones combinacionales $f_i(x)$ de las entradas de control de nuestro sistema. En cada una de ellas se pregunta si una determinada función lógica es verdadera o no. Si resulta verdadera esa función lógica, se llevará a cabo la transferencia entre registros T_i que viene a continuación. También existe la alternativa de, en vez de expresar la tranferencia entre registro T_i, listar las señales de control z_i de los dispositivos necesaria para llevar a cabo dicha transferencia (equivalente a lo que ya se comentó con las cartas ASM a nivel de control). Hay que tener en cuenta que en cada instrucción hay que garantizar que el conjunto de funciones $f_i(x)$ sea adecuado. Es decir, por una parte, para cada combinación de entradas de control X que tenga el sistema, deben quedar determinadas unívocamente las acciones a tomar, por lo que sólo una de las $f_i(x)$ que compongan la instrucción se hará verdadera. Por otra, para cada valor de entrada X al menos una de las $f_i(x)$ debe ser verdadera. En consecuencia, para que el conjunto de funciones $f_i(x)$ sea correcto deberá cumplirse:

$$\sum_{\forall i} f_i(x) = 1$$
$$f_i \cdot f_j \overset{OR}{=} 0 \qquad \forall (i \neq j)$$

Por último, el campo que queda en cada línea es el número N_i de la instrucción a la que se llegará si $f_i = 1$ y que se ejecutará en siguiente ciclo de reloj.

Descrito ya el formato general del lenguaje HDL que vamos a usar hay que comentar algunas simplificaciones que pueden realizarse sobre dicho formato. Hablaremos primero de campos que pueden eliminarse:

a) Si se cumple que la próxima dirección es la siguiente en orden numérico ($N_i = N + 1$), este campo puede desaperecer. Por ej., en la instrucción

N	$f_0(x)$	T_0/z_0

$f_1(x)$ T_1/z_1 N_1

se entendería que la próxima instrucción que se realizará si $f_0(x)$ hubiera sido verdadera sería N + 1, sin necesidad de indicarlo. Sin embargo, si hubiera sido cierta la función $f_1(x)$ iríamos a la instrucción N_1, que no es la correlativa a la que estábamos.

b) Hay veces en que “las funciones $f_i(x)$” resultan siempre verdaderas, en el sentido de que las transferencias entre registros se realizan siempre que se esté en esa instrucción, independientemente de las variables de entrada de control. Serían las que entendíamos como acciones de Moore. En este caso, se puede rellenar ese campo por la letra "t" (*true*), sin necesidad de expresar ninguna función combinacional. Por ej., la instrucción

N t T_0/z_0 N_0

indica que siempre se ejecutarán las transferencias T_0/z_0 y se seguirá por la instrucción N_0.

c) Si para deteminadas condiciones dentro de una instrucción no debe realizarse ninguna transferencia, el tercer campo de la instrucción puede quedar en blanco o rellenado con la palabra "NOP" (No Operación). Por ej.

N $f_0(x)$ NOP N_0

$f_1(x)$ T_1/z_1 N_1

Una instrucción que recoge las tres eliminaciones de campos es,

N t NOP

Esta instrucción “siempre” “no haría nada” y prosigue en la instrucción N+1. Aunque aparentemente inútil, con esta instrucción se permite “esperar” 1 ciclo de reloj.

Por último comentamos algunos cambios que pueden realizarse al formato general del lenguaje HDL que hemos definido:

a) Puede utilizarse la instrucción de salto incondicional

N GOTO N'

en vez de escribir

N t NOP N'

b) De forma análoga, se podría incorporar también la siguiente instrucción:

N IF $f_i(x)$ THEN z_i GOTO N_1

ELSE w_i GOTO N_2

en vez de

N $f_i(x)$ z_i N_1

$\overline{f_i(x)}$ w_i N_2

De esta forma, este lenguaje de descripción HDL, que de por sí se asemeja a los lenguajes de programación tradicionales de software, queda aún más cercano en tanto que se admiten "macros" como los dos ejemplos anteriores, que son típicos en cualquier lenguaje.

Una vez que conocemos las características del lenguaje HDL presentado, podemos realizar un paralelismo con las cartas ASM. Además de las referencias que se han ido comentando en los párrafos anteriores, ahora queremos centrarnos en las acciones incluidas en las cajas de las cartas ASM. En un

bloque ASM las acciones que llamábamos de Moore porque no dependían de las líneas de entrada del sistema, se trasladarán en HDL a acciones que se realizan en la instrucción correspondiente independientemente de cualquier función $f_i(x)$. De forma análoga, las acciones de Mealy en una carta ASM, que eran las que se activaban en un bloque ASM dentro de una caja de acción condicional sólo si se cumplía la correspondiente condición, son las señales que se denominan z_i en la instrucción HDL y que, en general, están asociadas a una de las funciones $f_i(x)$.

Para ilustrar estas equivalencias considérese la Fig. 1.61. El bloque inicial de espera S_0 se transforma en la instrucción inicial de espera 0, desde la cual se regresa a la propia instrucción 0 si Xs = 0 (línea $\overline{Xs}$ - 0) o se prosigue a la siguiente instrucción, si Xs = 1 (línea Xs -). El bloque final S_F también es especial ya que su instrucción (la número F) siempre activará la salida de control FIN y proseguirá con la instrucción 0 (línea [F] t FIN 0). En cuanto al bloque genérico S_k, que se transforma en la instrucción k, puede observarse que:

1. La acción de Moore “T” se activa en todas las líneas (esto es, para todas las combinaciones de las entradas x e y).

2. La “decisión” sobre x = 0, que no implica hacer más acciones que ir al bloque S_M, se traduce en la línea $\overline{x}$ T M (M es la insrtucción para S_M).

3. La acción condicional U, que se ejecutará cuando x = 1 sin importar el valor de y, aparece en las líneas “x·y” y “x·$\overline{y}$” de la instrucción.

4. El próximo bloque en el caso x y = 1 0 es S_L, por lo que aparece L en el campo de próxima instrucción de la línea “x·$\overline{y}$”, mientras que en el caso x y = 1 1 es S_{k+1}, por lo que dicho campo está vacío en la línea “x·y”.

1.6.2 Programa HDL de la calculadora del ejemplo

El ejemplo que veníamos trabajando en este capítulo (apartado 1.4) hacía referencia a una calculadora simple. Al igual que hicimos cuando estudiábamos las cartas ASM (apartado 1.5.5), ahora escribiremos el programa HDL para nuestro ejemplo, tanto a nivel de procesado como a nivel de control

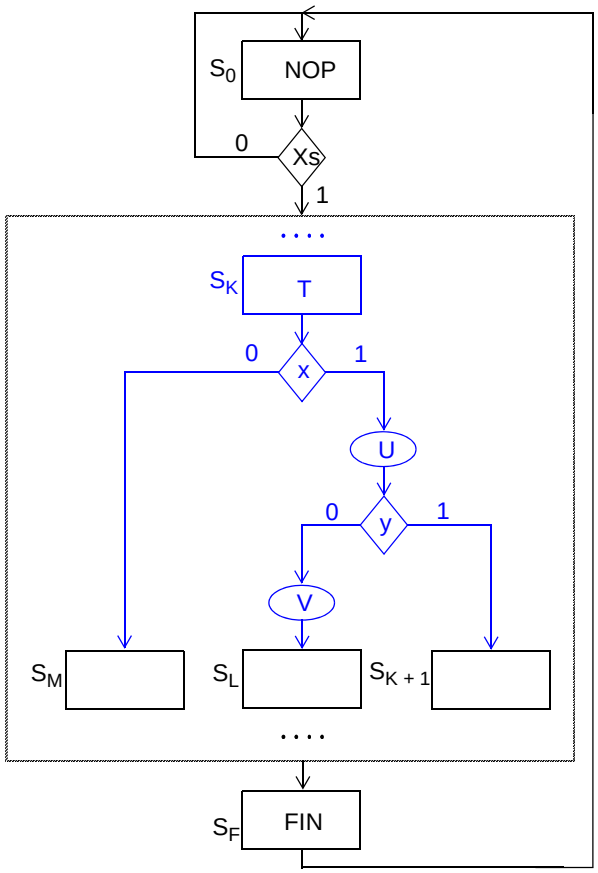
El programa HDL a nivel de procesado se muestra en la Fig. 1.62(a) y en él se recogen la secuencia de transferencias entre registros de la calculadora. Este programa se obtiene fácilmente de la carta ASM mostrada en la Fig. 1.56 siguiendo las indicaciones comentadas en al apartado anterior. Salvo en la forma, ambas representaciones del sistema, la de la carta ASM y la del programa HDL, son idénticas.

A nivel de control, las acciones del programa HDL representan las señales de control de los componentes de la unidad de dato que hay que activar. En nuestro caso, el programa HDL se presenta en la Fig. 1.62(b) y, como antes, muestra la misma representación del sistema que hay en la correspondiente carta ASM (Fig. 1.57).

1.7 EL DISEÑO DE LA UNIDAD DE CONTROL

Retomemos de nuevo la realización de los sistemas digitales. Un resumen del proceso seguido hasta ahora es el que sigue.

Una vez diseñada la unidad de datos (tal como la de la Fig. 1.28) se obtiene el conjunto de microoperaciones (Fig. 1.33) que permiten ejecutar las instrucciones o macrooperaciones definidas para el sistema. Ese conjunto de microoperaciones es ensamblado y en su caso depurado hasta quedar como una única secuencia de transferencias entre registros, la cual se representa por medio de una carta



0	$\overline{Xs}$	-	0
	Xs	-	
....			
K	$\overline{x}$	T	M
	xy	T/U	(K+1)
	$x\overline{y}$	T/U/V	L
K + 1	....		
	....		
L	....		
M	....		
	....		
F	t	FIN	0

Figura 1.61: Equivalencia entre carta ASM y programa HDL

0	$\overline{Xs}$	NOP	0
	Xs	NOP	
1	t	AC ← 0/T ← A	
2	$\overline{IR_2}$	T ← B / AC ← AC + T	
	IR_2	T ← B / AC ← AC - T	
3	$\overline{IR_1}$	AC ← AC + T	
	IR_1	AC ← AC - T	
4	$\overline{IR_0}$	A ← AC	
	IR_0	B ← AC	
5	t	NOP	0

(a) HDL a nivel de procesado

0	$\overline{Xs}$	NOP	0
	Xs	NOP	
1	t	ZAC/WT/RA	
2	$\overline{IR_2}$	WT/RB/WAC/s	
	IR_2	WT/RB/WAC/r	
3	$\overline{IR_1}$	WAC/s	
	IR_1	WAC/r	
4	$\overline{IR_0}$	RAC/WA	
	IR_0	RAC/WB	
5	t	FIN	0

(b) HDL a nivel de control

Figura 1.62: Programas HDL de la calculadora

ASM (Fig. 1.56) o de un programa HDL (Fig. 1.62(a)). A partir de estas representaciones, junto con el diseño de la unidad de datos, se derivan las correspondientes carta ASM y programa HDL de control (Figuras 1.57 y 1.62(b)).

Esta carta ASM y programa HDL de control son el punto de partida para el diseño de la unidad de control del sistema digital. Básicamente se trata de una máquina de estados, cada bloque ASM o instrucción es un estado de la máquina cuyas entradas son los cualificadores (variables sobre las que se toman decisiones) y cuyas salidas son los comandos (o señales a activar por el controlador) del sistemas digital. El diseño de la unidad de control es, pues, el diseño de un circuito secuencial cuya conducta viene dada por una carta ASM o por un programa HDL. Para realizar eficientemente el diseño existen diversas técnicas que serán tratadas en el siguiente capítulo, por lo que nos remitimos a él para conocer cómo se acaba de diseñar un sistema digital.

1.8 EL USO DEL SISTEMA DEL EJEMPLO

Tras completar la realización del sistema digital, en nuestro caso la calculadora de sumas y restas, ya está en condiciones de ser utilizado para operar con él. Desde la perspectiva del usuario el sistema se ve como se ilustra en la Fig. 1.63, figura que retoma la organización ya presentada en la Fig. 1.25. El usuario puede cargar en el registro IR el código de una instrucción, dar la orden de ejecución activando Xs y conocer el final de operación mediante FIN.

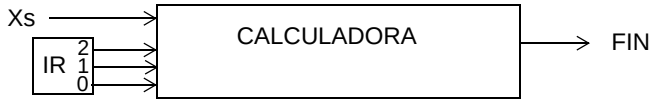


Figura 1.63: Visión de la calculadora desde la perspectiva del usuario

Pese a la gran sencillez de la calculadora y a lo limitado de su conjunto de instrucciones (sólo posee 8), con ella se pueden resolver problemas mas complejos "programando" una secuencia de éstas. Esto es, el sistema puede ser utilizado a nivel ISP (Instructor Set Processor), tal como se hace en los procesadores reales.

Como ejemplo del uso de esta calculadora mostraremos cómo puede resolver la siguiente operación:

$A \leftarrow 3A - B$.

Obviamente esa operación no puede realizarse con una sola de las instrucciones definidas en la calculadora por lo que habrá que seguir una secuencia de intrucciones (programa) que, en su conjunto, son las que permitirán obtener el resultado final.

Supongamos que inicialmente los registros A y B poseen los datos A_0 y B_0. En la Fig. 1.64 se muestra la secuencia de instrucciones que vamos a ir realizando, señalándose después de cada una de ella qué resultado parcial hay en cada registro. El programa comienza restando B a A y almacenando el resultado en B. Así, tras ejecutarse esta instrucción, el contenido del registro será A_0-B_0, mientras que el de A seguirá siendo A_0 ya que A no se modifica en la instrucción ejecutada. La segunda instrucción, sumar A y B en B, tampoco modifica el contenido de A mientras que deja a B con $A_0+(A_0-B_0) = 2A_0-B_0$. Por último, la tercera instrucción, sumar A y B (produciendo como resultado $3A_0-B_0$) en A, con lo que queda ejecutada la operación que se quería.

Una vez que se sabe cuáles son las instrucciones del programa que resuelve el problema, tenemos que poner en funcionamiento a la calculadora para ejecutarlo. Recordemos que para que se ejecute cada instrucción hay que poner su código binario en las tres señales de control $IR_{2\text{-}0}$; esto es, hay

		A	B
		A_0	B_0
Instrucción 1:	B ← A - B	A_0	$A_0 - B_0$
Instrucción 2:	B ← A + B	A_0	$2A_0 - B_0$
Instrucción 3:	A ← A + B	$3A_0 - B_0$	$2A_0 - B_0$

Figura 1.64: Resolución de A ← 3A-B con la calculadora

que almacenar el código de instrucción en el registro IR. Además, para que el sistema ejecute una instrucción, el controlador del sistema necesita recibir la señal de comienzo Xs. Por tanto, para cada instrucción del programa habrá un "operador" que realice ambas tareas. Así, el operador en nuestro caso procederá de la siguiente forma:

1º Fija $IR_{2-0} = 011$ (B ← A - B)

2º Activa la señal de comienzo Xs (y espera a que se active FIN)

3º Fija $IR_{2-0} = 001$ (B ← A + B)

4º Activa la señal de comienzo Xs (y espera a que se active FIN)

5º Fija $IR_{2-0} = 000$ (A ← A + B)

6º Activa la señal de comienzo Xs (y espera a que se active FIN).

Al igual que los procesadores reales, la calculadora diseñada en este capítulo es un ejemplo de sistema digital que puede ser utilizado a un nivel de abstracción muy alto, concretamente el de las instrucciones (ISP). Esto les proporciona dos características de sumo interés: 1) que con este tipo de sistema se pueden resolver por programación problemas mucho más complejos de los que resuelven individualmente las instrucciones para las que se diseñó el sistema; y 2) que el usuario del sistema no necesita ser especialista en la electrónica con la que se realizó el sistema. Ambas características hacen que estos sistemas sean prácticamente universales.

Sin embargo, al comparar la calculadora diseñada con los procesadores reales, se ve que están separados tanto por la distinta complejidad de las instrucciones como, sobre todo, por la forma de operar. Refiriéndonos al proceso de operación con la calculadora que acabamos de presentar, está claro que posee grandes inconvenientes como son la imposibilidad de almacenar el programa y el no poder ejecutar más que una instrucción cada vez. Es el usuario el que necesita estar atento y actuar sobre las entradas de control del sistema en cada momento, introduciendo el código de instrucción y generando los pulsos en la señal de comienzo Xs cada vez que se quiera ejecutar la nueva instrucción.

Todas estas características son las que separan nuestro sistema del ejemplo de un procesador real, mucho más complejo y cuya forma de operar se basa en ejecutar automáticamente el programa que previamente le ha sido almacenado. Tras el próximo capítulo, donde se presenta el diseño de unidades de control, diseñaremos un sistema digital que opera como los procesadores reales.

CAPÍTULO 2: Diseño de unidades de control

2.1 INTRODUCCIÓN

En este capítulo se describirán distintas formas de diseñar la unidad de control de un sistema digital. Recordemos que un sistema digital típico está formado por una unidad de datos (llamada también de procesado) y una unidad de control o controlador, tal como se esquematiza en la Fig. 2.1. La unidad de datos procesa los datos que tiene a su entrada proporcionando los correspondientes resultados (datos de salida). Para ello debe ejecutar una secuencia de microoperaciones que en conjunto realizan la operación deseada (macrooperación).

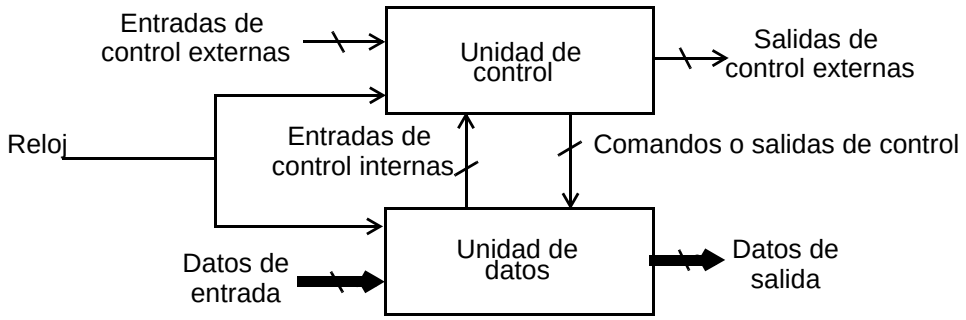


Figura 2.1: Estructura típica de un sistema digital

El proceso de diseño de la unidad de control toma como punto de partida el microprograma de control utilizando, en nuestro caso en concreto, las cartas ASM presentadas en el capítulo anterior. A partir de ellas, se diseña la unidad de control correspondiente a la unidad de procesado que se haya diseñado.

Como ejemplo para desarrollar en este capítulo los distintos tipos de realización, se va a considerar la carta ASM de control de la Fig. 1.57. Dicha carta ASM corresponde a la "calculadora" de sumas y restas diseñada en el capítulo anterior (Fig. 1.28). Por conveniencia, reperimos ambas figuras en la Fig. 2.2. Recordemos que en la calculadora se realiza una de las ocho operaciones posibles de sumar o restar dos datos almacenados en dos registros (A y B) y guardar el resultado en uno de ellos, A,B ±A±B. Además de los registros A y B, la unidad de datos posee una pequeña subunidad de cálculo (con un registro "tampón", un sumador-restador y un registro acumulador). Los registros A y B se comunican mediante un único bus. La operación a realizar viene dada por el valor de las variables de instrucción

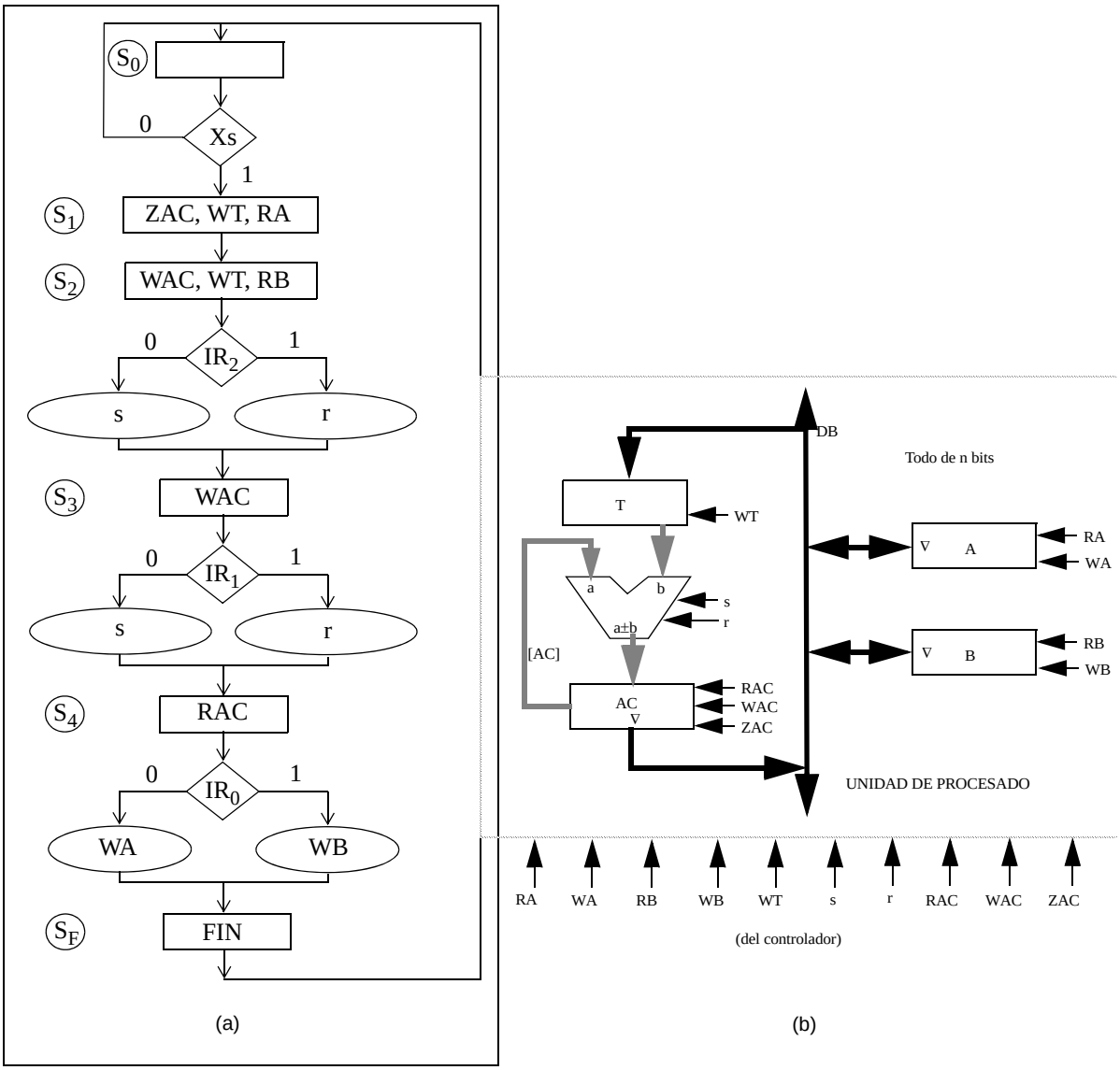


Figura 2.2: Calculadora como ejemplo de sistema digital a nivel RT: a)Carta ASM; b)Unidad de datos

(IR2,IR1,IR0) y se efectúa cuando se activa una señal de comienzo (XS). Una señal de FIN indica que la operación ha sido completada.

Este capítulo está organizado de la siguiente manera. En primer lugar nos centraremos en las estrategias de diseño de controladores. A continuación daremos la solución usando la lógica discreta en nuestra calculadora de ejemplo. Seguidamente presentaremos, con mucho nivel de detalle, la técnica de realización de controladores mediante un biestable por estado, que proporciona un magnífico compromiso entre la facilidad y sencillez del proceso de diseño y el coste del mismo. Un aspecto muy puntual, pero importante para garantizar que la técnica anterior funciona, es que la señal de comienzo tenga exactamente un ciclo de duración. En el penúltimo apartado se explica cómo conseguirlo. Por último, presentamos otras realizaciones basadas en Mux (multiplexores) e introducimos el diseño de control microprogramado mediante ROM y mediante PLA.

2.2 ESTRATEGIAS DE REALIZACIÓN DE CONTROLADORES

Las unidades de control pueden realizarse con una amplia variedad de estrategias. En esta sección comentaremos, en primer lugar, los objetivos y criterios de diseño y, posteriromente, presentaremos una clasificación de las diferentes posibilidades

2.2.1 Objetivos y criterios de diseño

La unidad de control tiene tres funciones principales:

- Llevar a efecto la secuencia de micro-operaciones
- Generar las señales de control de los componentes de la unidad de datos
- Evaluar las entradas de control tanto externas como internas

Así, el controlador es el circuito encargado de suministrar a la unidad de datos los niveles de señal apropiados en los tiempos apropiados para que ejecute dicha secuencia de microoperaciones (a estos niveles lógicos se les denomina comandos en la Fig. 2.1). Los comandos son por tanto salidas del controlador: se trata de las señales a activar en la unidad de procesado que vienen dadas en la carta ASM correspondiente (en nuestro ejemplo de la Fig. 2.2 son: ZAC, WT, RA, WAC, RB, s, r, RAC, WA y WB). En general, el controlador necesita información del estado de la unidad de datos, información que recibe mediante las señales internas de control que, por ello, también se denominan *señales de estados* (Fig. 2.1). Estas señales son variables de decisión para el algoritmo de control (en nuestro ejemplo de la Fig. 2.2 no existen señales de estados). Además, el controlador actuará en función de algunos "estímulos" externos de entrada, que también son variables de decisión para su algoritmo (XS, IR2, IR1, IR0 en la Fig. 2.2), así como suministrará otras señales de información al exterior (FIN en la Fig. 2.2).

En consecuencia, el controlador es un circuito secuencial cuyas entradas de control son las variables de decisión (señales de estados y entradas externas), representadas en las cajas de decisión de su carta ASM; sus estados son las cajas de estado de la carta ASM (S_0, S_1, S_2, S_3, S_4 y S_F en la Fig. 2.2) y sus salidas son las señales a activar representadas en las cajas de estado y de acción condicional de la carta ASM. Se trata pues de diseñar un circuito secuencial (controlador) cuya descripción está dada en la carta ASM correspondiente.

El objetivo de este tema es presentar distintas técnicas de realización de controladores así como los criterios de diseño que las soportan. Los criterios de diseño utilizados pueden ser de diversa índole:

- Reducción del número de biestables.
- Reducción del coste de la parte combinacional (próximo estado y salida).
- Modularidad (en el sentido de usar repetidamente un mismo componente o módulo básico).
- Programabilidad (en el sentido de que basta cambiar datos almacenados, sin modificar el hardware, para realizar distintos controladores).

Estos criterios orientan en gran medida la estrategia de implementación usada para el diseño del controlador.

2.2.2 Una clasificación de estrategias de implementación

Hay dos grandes líneas de diseño.

➢ Cableados:

- Sus componentes básicos son biestables y puertas o subsistemas combinacionales.
- Son diseños no-programables, sino "personalizados" al sistema digital concreto que controlan.
- Tienden a reducir el coste del número de biestables y/o de la lógica combinacional.
- Para cambiar la secuencia de operaciones, es necesario modificar el cableado.

➢ Microprogramables:

- Sus componentes son un secuenciador (que es básicamente un registro) y una ROM o PLA.
- La estructura del circuito de control es universal y la "personalización" a cada problema concreto se realiza almacenando "datos" en un elemento programable (ROM o PLA).
- Lo que se almacena en la ROM o PLA constituye el microprograma. El coste está relacionado con el tipo de "lenguaje" usado en la estructura del microprograma.
- Es posible cambiar la secuencia de operaciones sin tener que modificar el cableado.

Para cada una de las líneas de realización podemos considerar diversos criterios de diseño.

En la realización cableada, la reducción del coste (tanto de la parte combinacional como de la secuencial) es el principal criterio que orienta el diseño con lógica discreta.

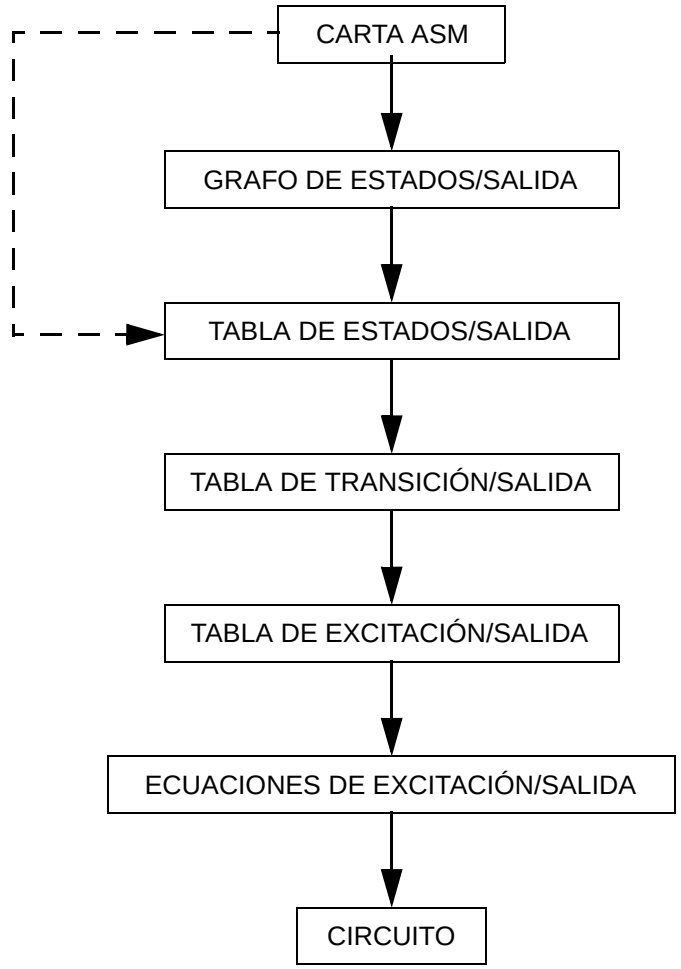
La modularidad, en el sentido de usar siempre el mismo circuito combinacional o secuencial, es el criterio primordial usado en las realizaciones basadas en un registro de desplazamiento y en multiplexores. Este criterio conduce a procesos de diseño más cortos, fáciles y sistemáticos, con menos pasos desde la carta ASM a la realización del circuito. Sin embargo, son diseños más costosos en cuanto al número de elementos usados.

En los diseños microprogramables el criterio básico es la programabilidad. También se tiene en cuenta el coste en el sentido de reducir el tamaño de la ROM o PLA. Esto puede realizarse añadiendo circuitería combinacional o usando un "lenguaje" adecuado para la programación, es decir, escribir las instrucciones usando un formato determinado. La modularidad también orienta este tipo de diseño. En este caso el módulo básico es un conjunto mínimo de instrucciones distintas con las cuales se realizan todos los programas.

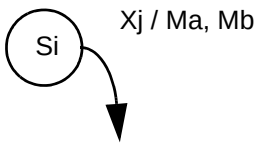
2.3 REALIZACIÓN CON LÓGICA DISCRETA

Se usa la técnica estudiada en temas anteriores para el diseño de circuitos secuenciales síncronos. La codificación de estados se realiza con el mínimo número de variables, esto es, de forma que para N estados necesitamos n variables de estado, donde $n = \lceil \log_2 N \rceil$, siendo $\lceil x \rceil$ el entero por exceso de x.

Los pasos a seguir partiendo de una carta ASM son:



El circuito secuencial, que en definitiva es una carta ASM, puede ser representado mediante un pseudografo de estados-salida usando la notación de la Fig. 2.3. Así, basándonos en la carta ASM de nuestro ejemplo es casi inmediato obtener el pseudografo de estados-salida mostrado en la Fig. 2.4. Para ello basta asignar un estado a cada bloque ASM y seguir el flujo de la carta de la Fig. 2.2. Las salidas asociadas al pseudografo de estados-salida recogen tanto las que están dentro de la caja de estado como las que ocurren en las cajas de acción condicional de la trayectoria que se cumpla en la entrada de control chequeada en cada caso.



Si: Estado asociado a la micro-operación (bloque ASM)
Xj: Entrada de control chequeada
Ma, Mb: Salidas de control activas

Figura 2.3: Notación utilizada en el pseudografo de estados/salida

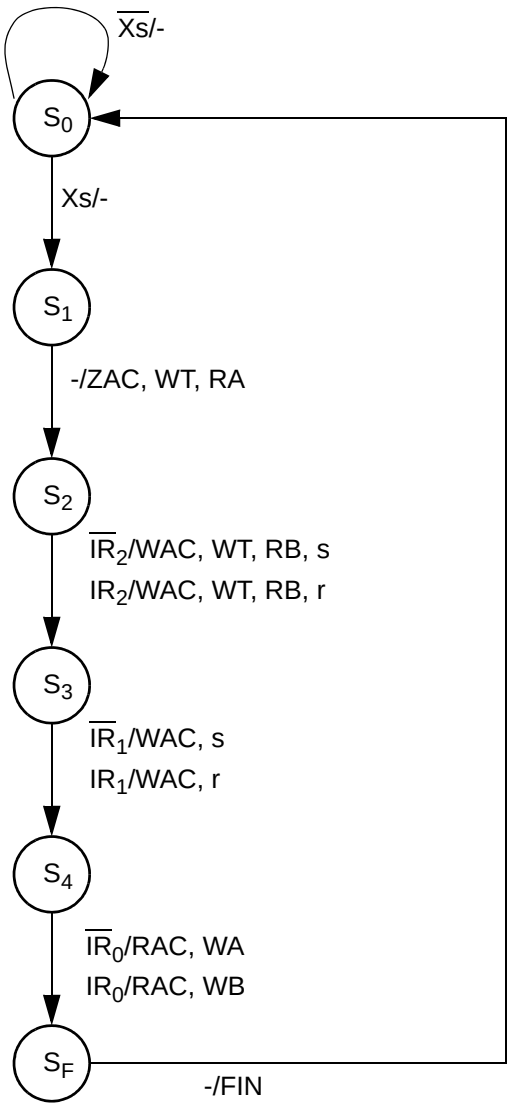


Figura 2.4: Diagrama de estados para el control de la calculadora

Este paso, sin embargo, puede ser omitido ya que es fácil pasar directamente de la carta ASM a la tabla de estados/salida. Para ello se procede de igual forma que para obtener el pseudografo de estados-salida pero escribiendo una tabla en vez de un grafo.

En cualquier caso, a través del pseudografo o directamente de la cartaASM, podemos obtener la siguiente tabla de estados:

ENTRADAS				Estado Presente	Estado Próximo	SALIDAS										
Xs	IR_2	IR_1	IR_0			WAC	RAC	ZAC	WT	RA	WA	RB	WB	s	r	FIN
0	-	-	-	S_0	S_0	0	0	0	0	0	0	0	0	0	0	0
1	-	-	-	S_0	S_1	0	0	0	0	0	0	0	0	0	0	0
-	-	-	-	S_1	S_2	0	0	1	1	1	0	0	0	0	0	0
-	0	-	-	S_2	S_3	1	0	0	1	0	0	1	0	1	0	0
-	1	-	-	S_2	S_3	1	0	0	1	0	0	1	0	0	1	0
-	-	0	-	S_3	S_4	1	0	0	0	0	0	0	0	1	0	0
-	-	1	-	S_3	S_4	1	0	0	0	0	0	0	0	0	1	0
-	-	-	0	S_4	S_F	0	1	0	0	0	1	0	0	0	0	0
-	-	-	1	S_4	S_F	0	1	0	0	0	0	0	1	0	0	0
-	-	-	-	S_0	S_0	0	0	0	0	0	0	0	0	0	0	1

Vamos a realizar la siguiente asignación arbitraria: $S_0 = 000$; $S_1 = 001$; $S_2 = 010$; $S_3 = 011$; $S_4 = 100$; y $S_F = 101$. Se obtiene la siguiente tabla de transición:

ENTRADAS				Estado Presente	Estado Próximo	SALIDAS										
Xs	IR_2	IR_1	IR_0	$q_2q_1q_0$	$Q_2Q_1Q_0$	WAC	RAC	ZAC	WT	RA	WA	RB	WB	s	r	FIN
0	-	-	-	000	000	0	0	0	0	0	0	0	0	0	0	0
1	-	-	-	000	001	0	0	0	0	0	0	0	0	0	0	0
-	-	-	-	001	010	0	0	1	1	1	0	0	0	0	0	0
-	0	-	-	010	011	1	0	0	1	0	0	1	0	1	0	0
-	1	-	-	010	011	1	0	0	1	0	0	1	0	0	1	0
-	-	0	-	011	100	1	0	0	0	0	0	0	0	1	0	0
-	-	1	-	011	100	1	0	0	0	0	0	0	0	0	1	0
-	-	-	0	100	101	0	1	0	0	0	1	0	0	0	0	0
-	-	-	1	100	101	0	1	0	0	0	0	0	1	0	0	0
-	-	-	-	000	000	0	0	0	0	0	0	0	0	0	0	1

Si se usan biestables JK para la realización de este diseño, hemos de tener en cuenta la tabla de transición de este último, que viene dada por:

$q \rightarrow Q$	J K
$0 \rightarrow 0$	0 x
$0 \rightarrow 1$	1 x
$1 \rightarrow 0$	x 1
$1 \rightarrow 1$	x 0

Con ello se obtiene la siguiente tabla de excitación:

Xs IR_2 IR_1 IR_0	q_2 q_1 q_0	J_2K_2 J_1K_1 J_0K_0
0 - - -	0 0 0	0 x 0 x 0 x
1 - - -	0 0 0	0 x 0 x 1 x
- - - -	0 0 1	0 x 1 x x 1
- - - -	0 1 0	0 x x 0 1 x
- - - -	0 1 1	1 x x 1 x 1
- - - -	1 0 0	x 0 0 x 1 x
- - - -	1 0 1	x 1 0 x x 1

Basándonos en estas tablas obtenemos las ecuaciones de excitación y de salida que vienen dadas a continuación:

Ecuaciones de Excitación:

$J_2 = q_1 \cdot q_0$ $\quad K_2 = q_0$

$J_1 = \overline{q}_2 \cdot q_0$ $\quad K_1 = q_0$

$J_0 = Xs + q_2 + q_1$ $\quad K_0 = 1$

Ecuaciones de Salida:

$WAC = q_1$; $ZAC = RA = \overline{q}_2 \cdot \overline{q}_1 \cdot q_0$; $RAC = q_2 \cdot \overline{q}_0$

$WT = \overline{q}_2 \cdot \overline{q}_1 \cdot q_0 + q_1 \cdot \overline{q}_0$; $WA = \overline{IR}_0 \cdot q_2 \cdot \overline{q}_0$; $RB = q_1 \cdot \overline{q}_0$

$WB = IR_0 \cdot q_2 \cdot \overline{q}_0$; $s = q_1 (\overline{IR}_2 \cdot \overline{q}_0 + \overline{IR}_1 \cdot q_0)$; $r = q_1 (IR_2 \cdot \overline{q}_0 + IR_1 \cdot q_0)$

$FIN = q_2 \cdot q_0$

Mediante estas ecuaciones realizamos el circuito que se muestra en la Fig. 2.5.

Comentarios:

La realización con lógica discreta tiene la ventaja de que se realizan circuitos de bajo coste, pero el inconveniente de que no hay correspondencia obvia entre la realización y el algoritmo que la representa. Además un cambio en el algoritmo requiere un rediseño completo del circuito.

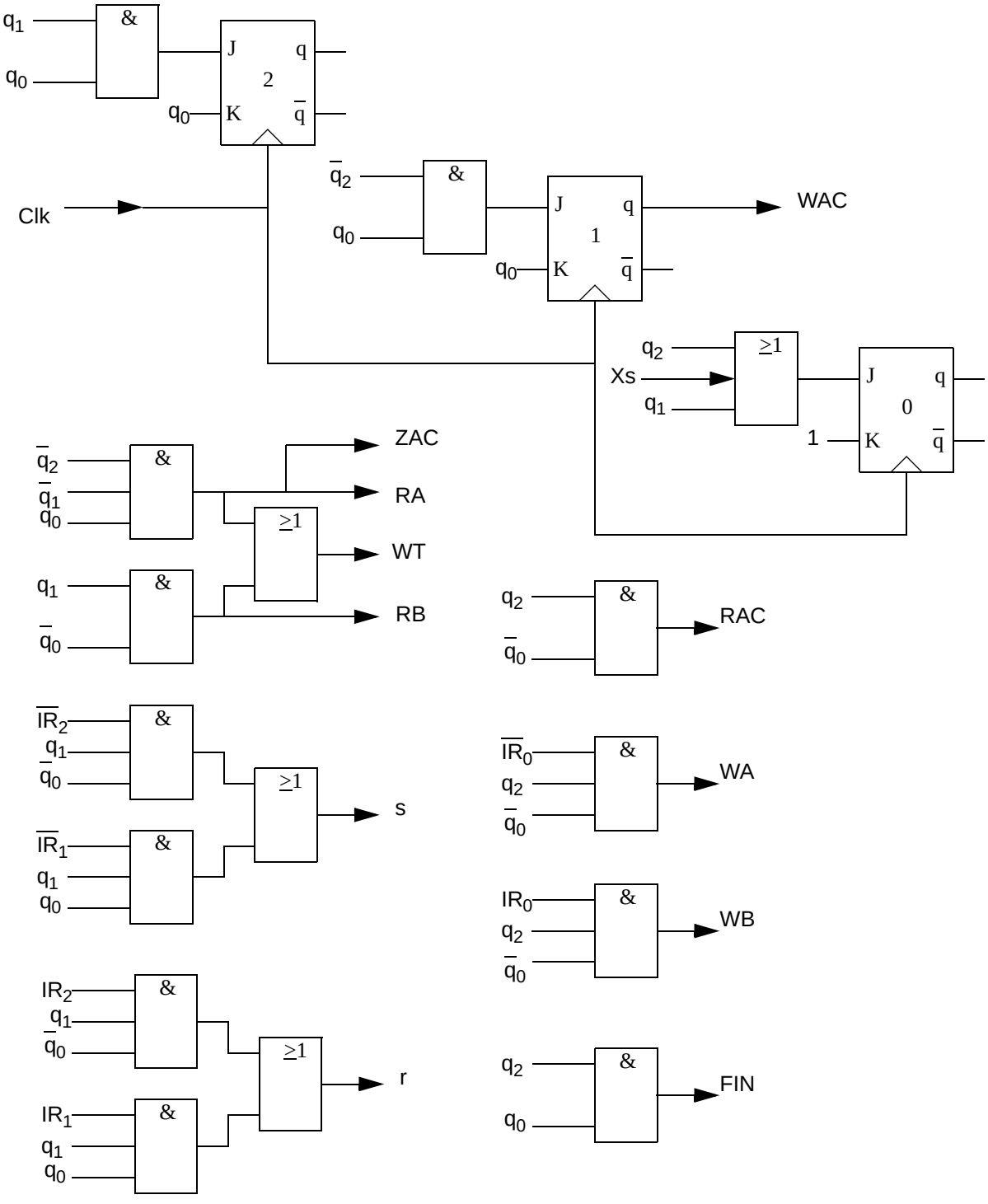


Figura 2.5: Realización de la Unidad de Control de la Calculadora con Lógica Discreta

2.4 REALIZACIÓN BASADA EN UN BIESTABLE POR ESTADO.

Este tipo de diseño pertenece a la línea cableada. Su realización es un proceso sistemático (y formal) a partir de la carta ASM, lo que lo hace simple y con "realización-conectada-al-algoritmo". Es modular, lo que permite hacer cambios con poco esfuerzo de diseño (sí a nivel de placa). No reduce (sino maximiza) el número de estados, pero simplifica en general la lógica de excitación de biestables y de salida.

La realización de la carta ASM presenta varias alternativas, pudiendo plantearse como Moore o Mealy e intentando o no compartir microoperaciones entre operaciones, lo que nos lleva a distintos diseños del controlador. En general, si no se tiene un control adecuado sobre las entradas, es conveniente utilizar la opción Moore ya que las salidas del controlador son señales que normalmente operan de acuerdo con el reloj de la unidad de datos y éstas no deben presentar pulsos espurios ni errores. En el autómata de Moore las salidas sólo dependen de los estados, con lo que se puede asegurar que van a permanecer constantes durante un ciclo de reloj.

2.4.1 Fundamentos

Cada microoperación, salvo la de espera (S_0 en nuestro ejemplo de la Fig. 2.2), tiene asociada una variable de estado que se "activa" cuando se realiza esa microoperación; en los otros ciclos permanece inactiva. A nivel de asignación de estados esto equivale a una codificación "one-hot". El estado de espera tiene asociado el código correspondiente a todas las variables a 0 que significa "ninguna microoperación". El asignamiento correspondiente a nuestro ejemplo sería pues:

ESTADO	q_1 q_2 q_3 q_4 q_F
S_0	0 0 0 0 0
S_1	1 0 0 0 0
S_2	0 1 0 0 0
S_3	0 0 1 0 0
S_4	0 0 0 1 0
S_F	0 0 0 0 1

Excepto la transición $S_0 \rightarrow S_1$, que consiste en introducir Xs = 1 en el primer biestable, cada transición de estados $S_k \rightarrow S_{k+1}$ es un desplazamiento a la derecha del 1 almacenado en el biestable k-ésimo, esto se representa esquemáticamente en la Fig. 2.6.

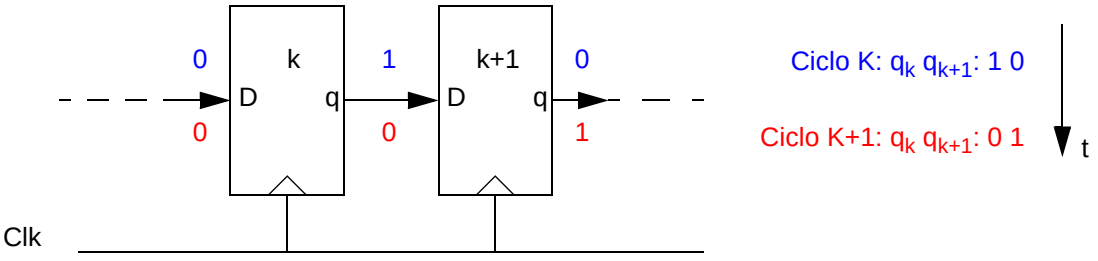


Figura 2.6: Transición de estados

Con ello, la unidad de control consiste, básicamente, en un registro de desplazamiento con leves modificaciones que dan cabida a saltos en la secuencia normal (desviación de caminos, repeticiones, etc.).

Las salidas a activar en cada microoperación coinciden con la salida q_k del biestable de esa microoperación. La señal de salida se obtiene mediante la operación OR de las salidas q_k de los biestables correspondientes a las microoperaciones que activan esa salida. Por ejemplo, para el fragmento de microprograma mostrado a continuación, los valores de las variables de estado y de la salida σ_R son:

MICROPROGRAMA	
...	...
μop k	σ_R...
μop k+1	...(no σ_R)
μop k+2	σ_R...

... q_k q_{k+1} q_{k+2} ...	σ_R
...	...
... 1 0 0 ...	1
... 0 1 0 ...	0
... 0 0 1 ...	1
...	...

De las tablas anteriores podemos deducir la siguiente ecuación para la salida:

$$\sigma_R = ... + q_k + q_{k+2} + ...$$

En la Fig. 2.7 se muestra el una parte del controlador correspondiente a esta salida[1].

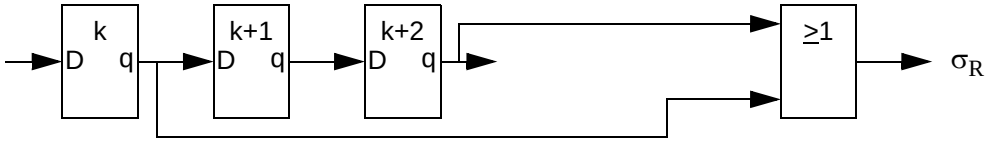


Figura 2.7: Obtención de la salida

Estado de espera, S_0

El estado de espera S_0 es especial. De hecho, este estado S_0 es el único que no tiene un biestable asociado. Vamos a discutir sus tres aspectos relevantes: cómo se consigue que sea de no operación, cómo se consigue salir de él para alcanzar el primer estado (S_1) y cómo se regresa a S_0 desde el último estado (que asumiremos es S_0).

El estado de espera es un estado de no operación. La asignación binaria que posee S_0 es q_1=q_2=...=0. Como todas las salidas se obtienen mediante la OR de las variables q_i ($\forall$j, σ_j=OR(q_i)), en S_0 todas las salidas serán σ_j=0 ($\forall$j), y, por tanto, no se activa ninguna señal de control. Así, S_0 es un estado de no operación.

La solución para la consecución del primer cambio de estado, de S_0 a S_1, viene dada en la Fig. 2.8. La señal de entrada que marca el comienzo, Xs, es un pulso positivo de un ciclo de reloj. Como esta señal está conectada a la entrada D del biestable 1, el biestable 1 capturará 0's hasta que Xs se active. En ese momento capturará un 1 con lo que se habrá alcanzado el estado S_1. En el ciclo siguiente, en el que Xs ya habrá regresado a 0, el biestable 1 volverá a capturar un 0 mientras que su salida q_1=1 habrá sido capturada por el biestable 2 (alcanzándose el estado S_2).

1. De ahora en adelante no se dibujará la señal de Clk común a los biestables, salvo que sea conveniente.

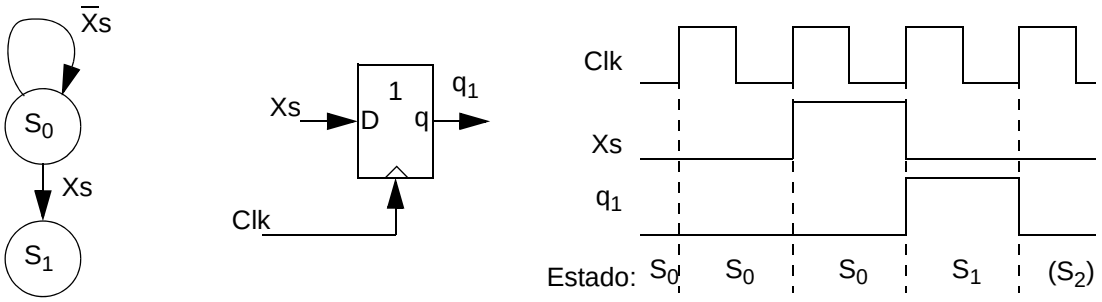


Figura 2.8: Solución para el estado inicial de espera y su transición al primer estado activo.

El paso de la última microoperación (S_F) al estado de espera, simplemente consiste en "perder" el 1 del biestable final F en el desplazamiento correspondiente. Esta situación se ilustra en la Fig. 2.9. Con ella se quiere representar que las variables de estado van tomando el valor "1" durante un ciclo de reloj de forma sucesiva. La última en tomar este valor es q_F y, como todas las variables de estado anteriores ya valen "0", en el siguiente ciclo de reloj también q_F se hace "0" volviendo, por tanto, al estado de espera S_0.

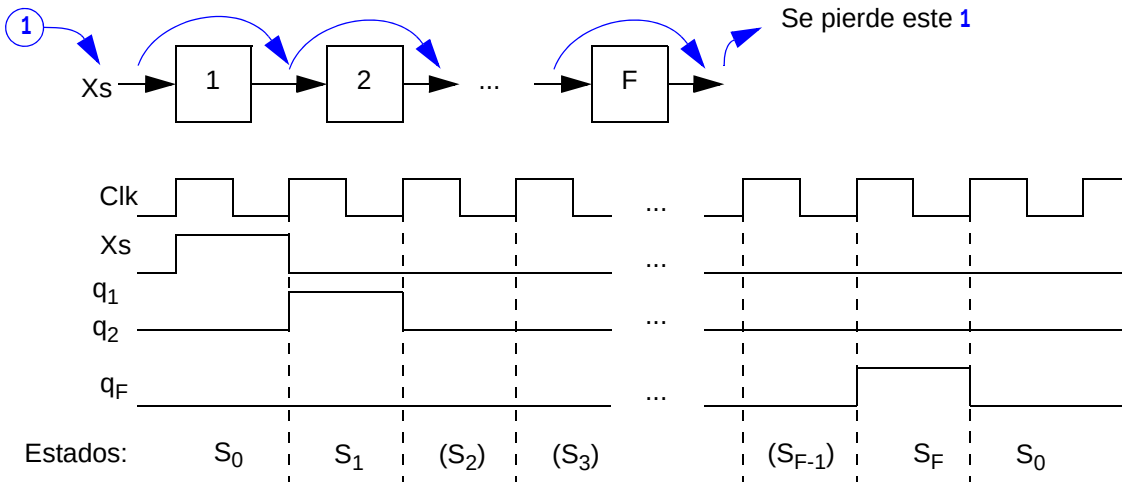


Figura 2.9: Transición de regreso al estado inicial de espera

2.4.2 Asociación carta ASM con circuito de control

La implementación basada en desplazamiento se obtiene directamente de la carta ASM, sin necesidad de pasos intermedios. Esto es lo que llamamos aproximación formal, que consiste en asociar cada bloque ASM con un biestable y un circuito combinacional de salida de la siguiente forma:

- Cada caja de estados del bloque ASM se realiza con un biestable (normalmente de tipo D).
- Cada caja de decisión se implementa con un demultiplexor controlado por la condición.
- Las acciones de las cajas de acción condicional son las salidas del demultiplexor citado anteriormente.

- Si se ‘unen’ dos señales en el mismo punto, la unión se consigue con una OR de esas señales.

Como ejemplo de los tres primeros casos, podemos ver cómo se implementaría el estado 2 de la carta ASM de la Fig. 2.2. Esto se muestra en la Fig. 2.10, para la máquina de Mealy. Por otra parte, en la Fig. 2.11 se muestra el ejemplo anterior pero realizado como máquina de Moore.

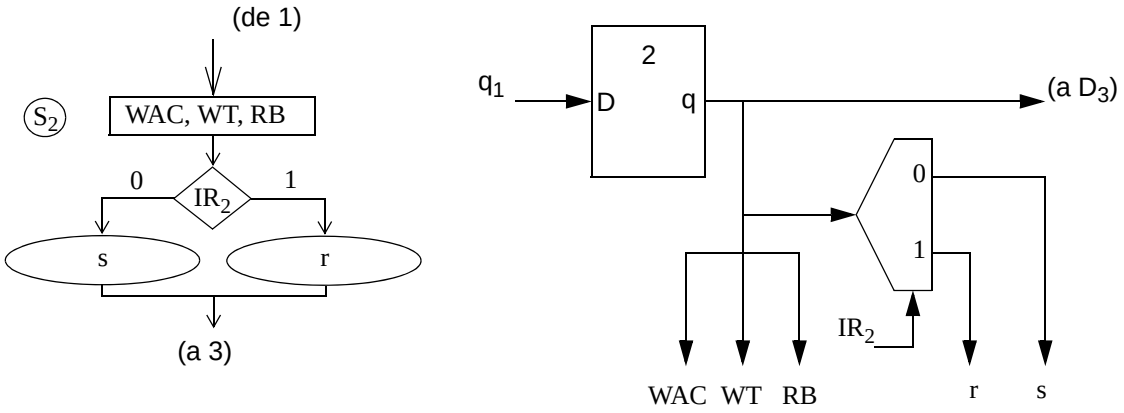


Figura 2.10: Aproximación formal en caso de Mealy

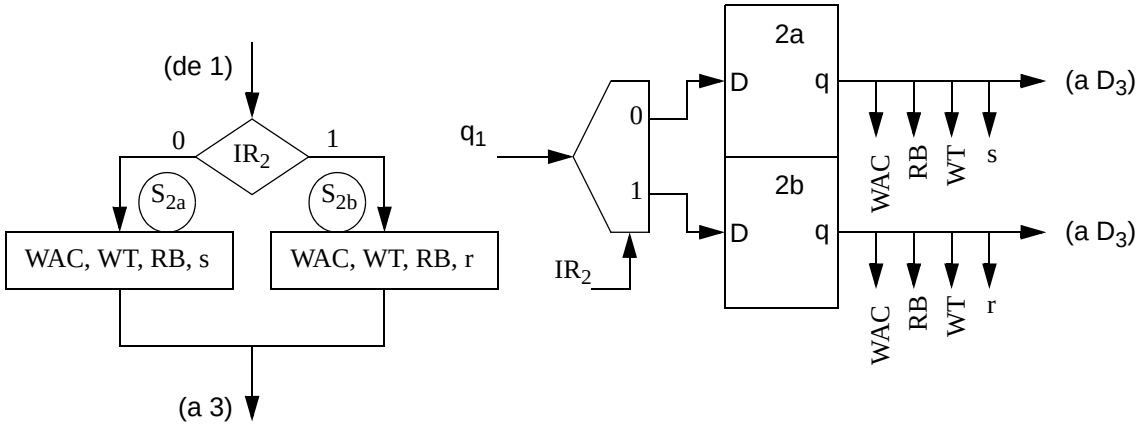


Figura 2.11: Aproximación formal en caso de Moore equivalente al anterior

Por último, con respecto al uso de una puerta OR para la unión de señales, podemos ver en la Fig. 2.12 un caso de punto de acumulación de caminos. En concreto, desde los lugares “a”, “b” y “g” de una carta ASM, hay que continuar por el bloque S_k. Una OR(a, b, g) resuelve esto.

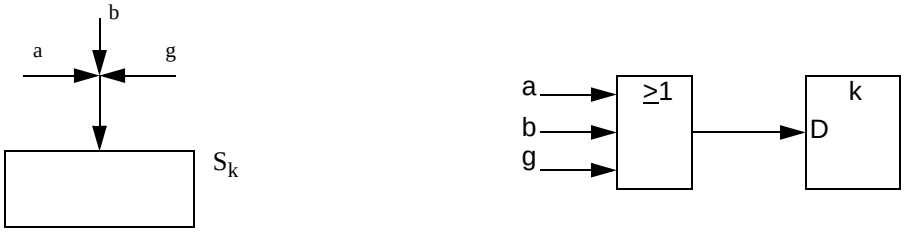


Figura 2.12: Ejemplo de solución para la acumulación de caminos

En algunos casos concretos es posible reducir el circuito mediante acciones puntuales. Por ejemplo, si la OR de algunas o todas las variables es igual a una "q", debe eliminarse la OR y sustituirse por la q. Esta simplificación ocurre cuando todas las salidas de un bloque ASM van a la misma caja de estados. En el caso de la carta ASM de la Fig. 2.2 se producen varias acumulaciones de caminos y siempre la OR de los distintos caminos coinciden con la q del estado anterior: por ejemplo, para el estado 2 se cumple: $D_3 = s + r = q_2 \cdot \overline{IR_2} + q_2 \cdot IR_2 = q_2$
Es por ello que podemos eliminar la OR tal como se muestra en la Fig. 2.13.

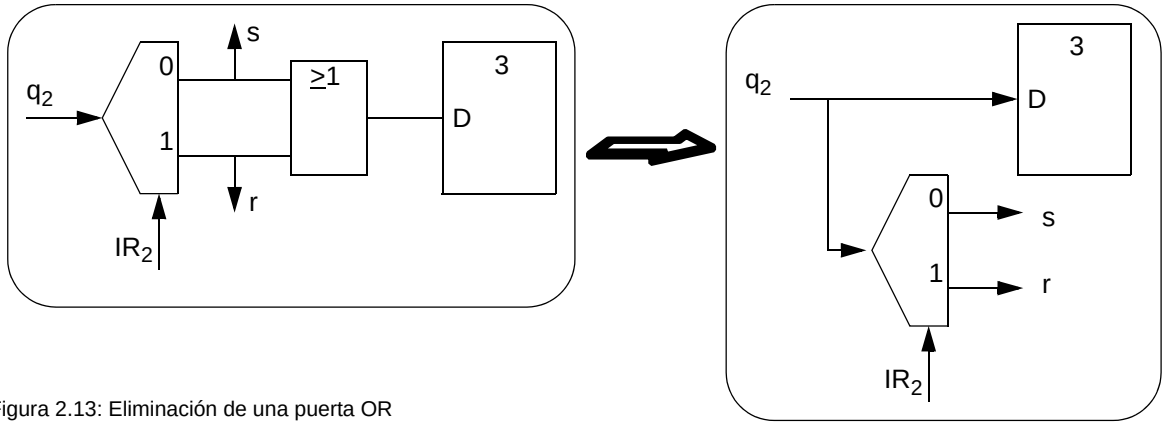


Figura 2.13: Eliminación de una puerta OR

2.4.3 Casos particulares

Desde la perspectiva de los algoritmos a desarrollar, hay algunos casos particulares de interés a la hora de obtener una única carta ASM que represente el microprograma de control, casos que vamos a considerar a continuación. La implementación en el controlador de cada uno de estos casos sigue las reglas de asociaciación generales que acabamos de ver, pero su presentación nos permitirá aplicar esas reglas a esos casos concretos. Se trata de: 1/la bifurcación de acciones en una microoperación, 2/el desarrollo de varias macrooperaciones de longitud variable, 3/la repetción de una secuencia de microoperaciones (lo que sería un bucle en el microprograma); y 4/la anulación de comandos.

2.4.3.1 Bifurcación de Acciones en una Microoperación

Cuando en una microoperación pueden tomarse dos o más caminos para realizar acciones diferentes en cada uno, se produce una bifurcación de acciones en esa microoperación. Puede resolverse de dos formas:

- Un bloque ASM único que contiene las acciones condicionales (solución que corresponde la máquina de Mealy).
- Una decisión que conduce a varios bloques ASM (solución que corresponde la máquina de Moore).

Esto ha sido mostrado en las Fig. 2.10 y Fig. 2.11, respectivamente.

Puede haber más de dos bifucarciones en una microoperación, lo que ocurre cuando éstas se establecen sobre más de una variable de entrada al controlador. La solución es un circuito combinacional que resuelva la condición de entrada que ocurre. No hay, en general, un criterio ni método de opti-

mización. Sin embargo, el coste varía dependiendo de la forma elegida, como muestra el siguiente caso.

Las dos propuestas mostradas en la Fig. 2.14 son equivalentes. Sus soluciones formales son mostradas en la Fig. 2.15, junto a una tercera solución a nivel de puertas. Puede observarse que el coste de la implementación cambia. Así, si la solución se basa en DEMUX, la solución Fig. 2.15.b es menos costosa que la Fig. 2.15.a. Esta solución de menor coste debe ser elegida tanto si la carta ASM se ha desarrollado de una como de la otra forma mostrada en la Fig. 2.14.

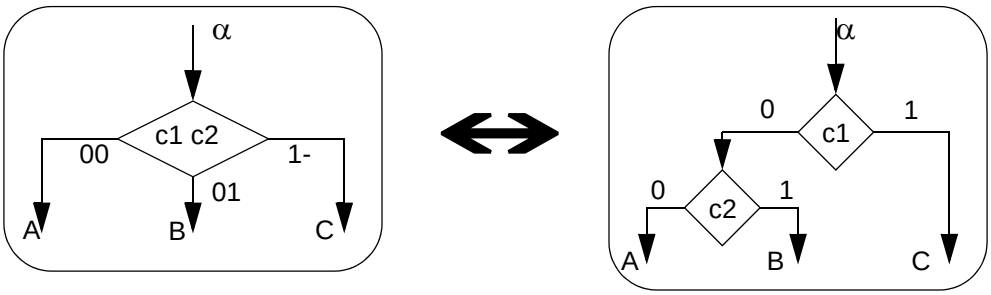


Figura 2.14: Bifurcación con más de dos caminos: opciones equivalentes en carta ASM

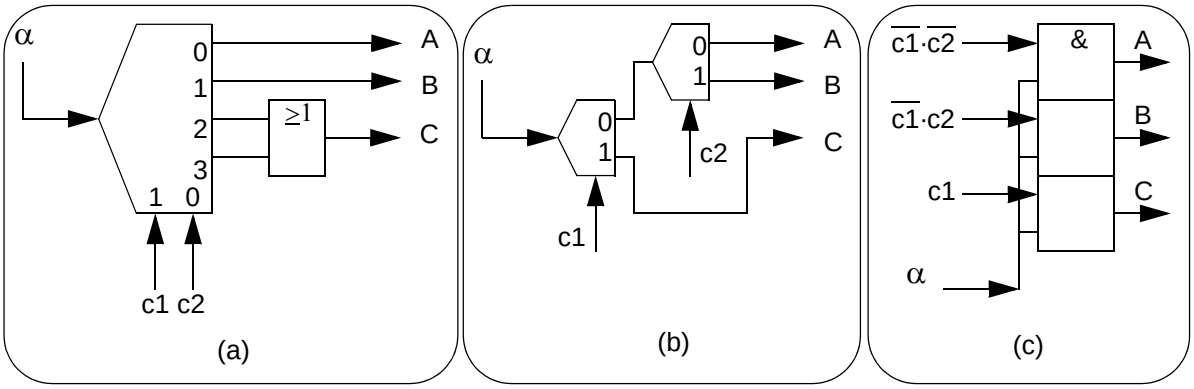


Figura 2.15: Solución de distinto coste para las bifurcaciones de la Fig. 2.14: a) Con DEMUX 1:4; b) con DEMUX 1:2; c)con puertas AND

2.4.3.2 Macrooperaciones de longitud variable

Considérese la situación que se muestra en la Fig. 2.16. Hay dos macrooperaciones que comparten microoperaciones hasta la “k-1” y a partir de la “k+1”. Sin embargo el fragmento “k” tiene longitud variable: una macrooperación sólo tiene la microoperación K0, mientras que la otra posee n microoperaciones (k11, k12, ..., k1n).

La Fig. 2.16 muestra el fragmento de carta ASM y la solución del controlador, con dos caminos de biestables que se bifurcan para después volver a unirse mediante una OR.

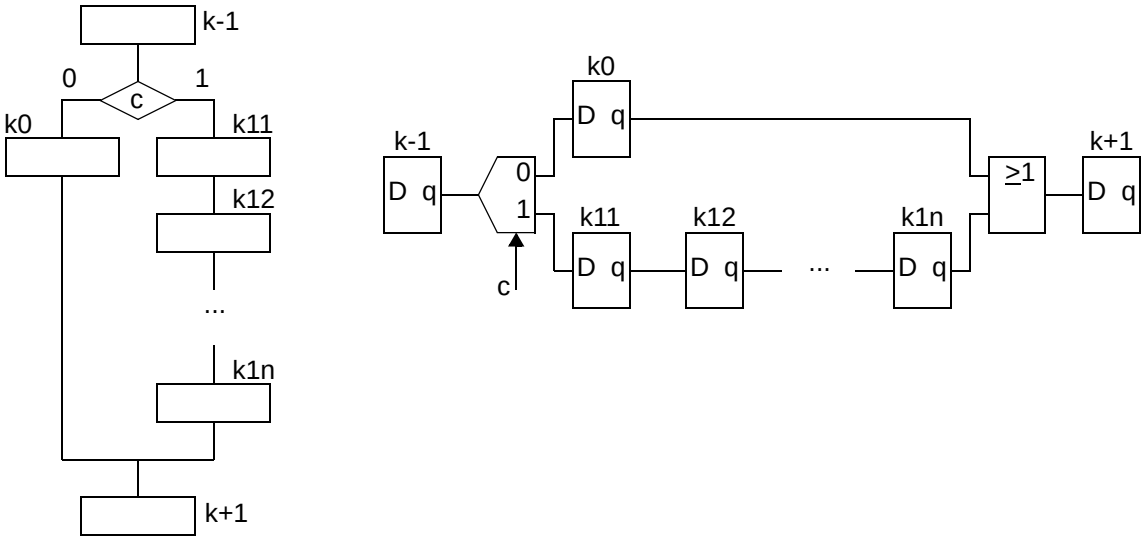


Figura 2.16: Carta ASM y solución de control para macrooperaciones de longitud variable

2.4.3.3 Repetición de microoperaciones

El desarrollo de una macrooperación consiste a veces en un bucle de microoperaciones que se repite hasta el cumplimiento de determinado valor de entrada. Un ejemplo de cómo se implementa este caso está dado en la Fig. 2.17.

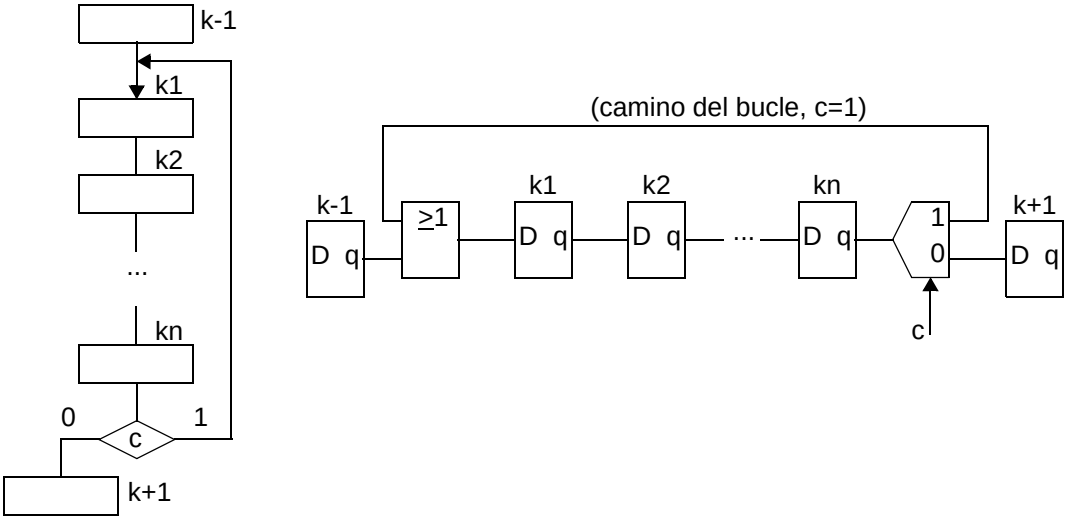


Figura 2.17: Carta ASM y solución de control para repetición de microoperaciones

2.4.3.4 Anulación de comandos

En algunas microoperaciones, dependiendo del valor de una señal de entrada, debe realizar algún comando o no. Se dice a veces que dicha señal anula ese comando. Un ejemplo de esta situación esta dado en la Fig. 2.18, donde en la microoperación k se hace el comando de dos acciones S1 y S2 o, si la entrada es c=0, sólo se hace S1 (S2 ha sido anulado).

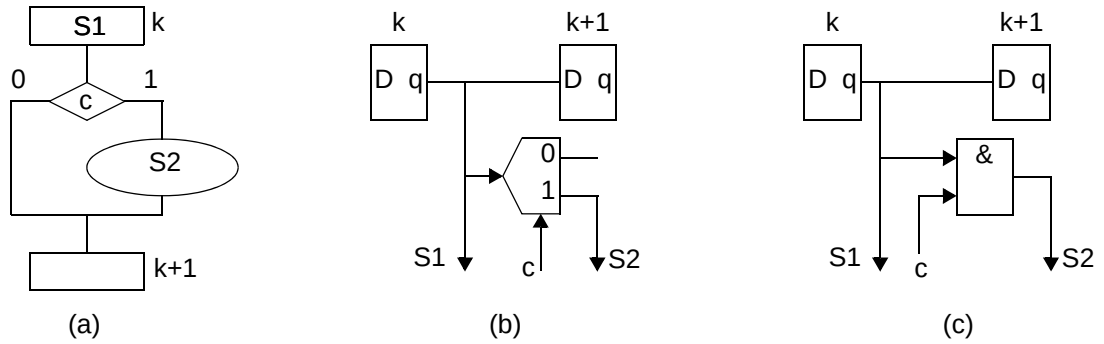


Figura 2.18: Anulación de comandos: a)ASM; b)opción estándar con DEMUX; c)opción de coste reducido con AND

Obsérvese que la solución hardware mediante puerta AND optimiza el diseño estándar con DEMUX, por lo que hay que adoptarla.

2.4.4 Solución a la unidad de control de la calculadora de nuestro ejemplo

Como aplicación de todo lo que hemos visto, realizamos la implementación del controlador de la calculadora con su carta ASM correspondiente (Fig. 2.2), el cual está dado en la Fig. 2.19.

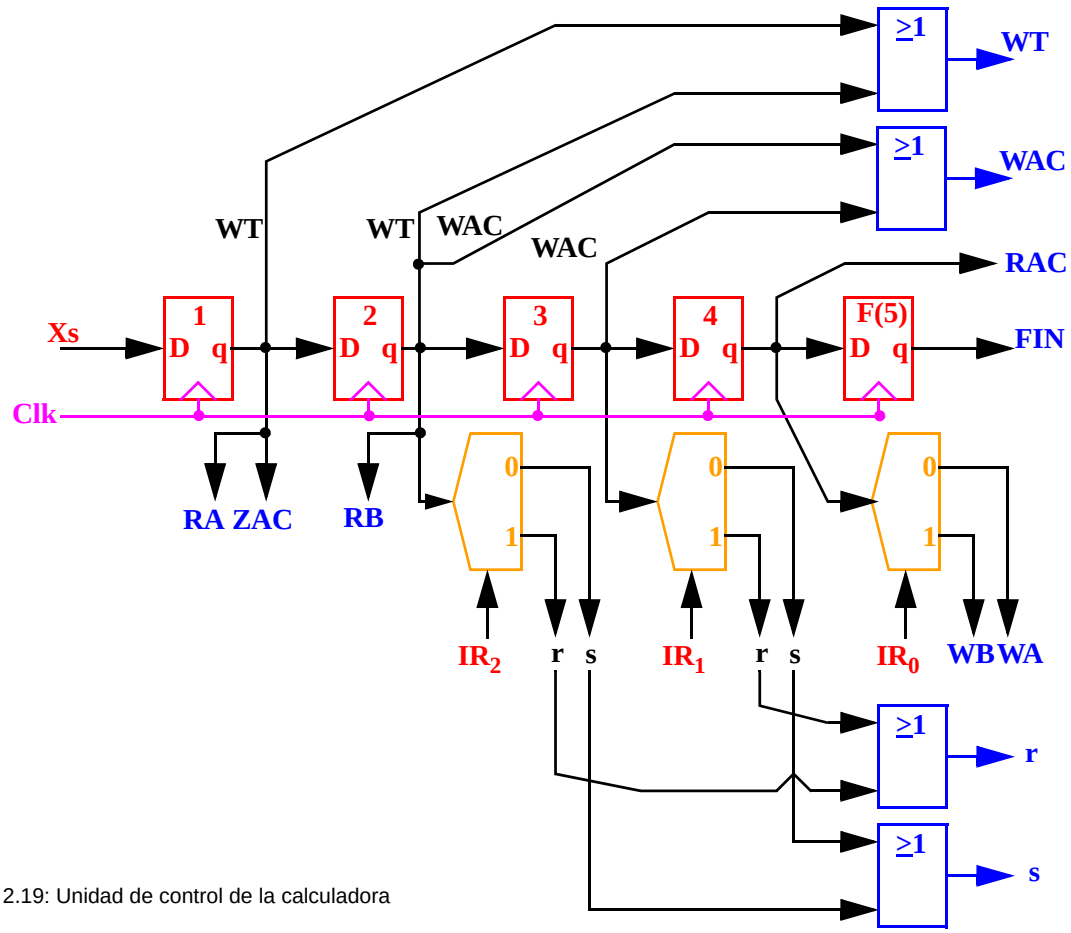


Figura 2.19: Unidad de control de la calculadora

Comentarios:

El diseño con registros de desplazamiento emplea más flip-flops, pero generalmente menos lógica combinacional, es decir, menos puertas para generar funciones lógicas. Aunque el diseño no es tan económico en hardware como la realización discreta, es más ordenado y sistemático en el sentido que podemos determinar fácilmente y de forma precisa lo que hace cada flip-flop.

Una de las grandes ventajas que posee este diseño es que es directamente implementable de la carta ASM. Una desventaja es que hay que tener especial cuidado en la inicialización, problema que es tratado en el apartado siguiente.

2.5 PROBLEMAS DEL COMIENZO

En el comienzo de la operación hay dos problemas:

- Uno es conducir al controlador al estado S_0. Comenzar en el estado S_0 sólo es problema tras "poner en marcha" al sistema (OFF $\rightarrow$ ON), ya que posteriormente, tras cada microoperación final, se vuelve a S_0 automáticamente: se "pierde" el 1 que se desplaza, tal como se explicó en el apartado 2.4.1 y en la Fig. 2.9.
- Otro es generar la señal XS adecuada. Este problema aparece al adaptar una señal de comienzo demasiado corta o demasiado larga a la duración correcta para nuestro controlador.

Obtener S_0 como estado incicial

Una forma de solucionar el primero de los problemas es generar una señal, RESET, al principio de la operación del controlador de forma que todos los biestables que lo componen sean puestos a cero asíncronamente, con lo que llevamos al sistema al estado S_0. Esta solución esta esquematizada en la Fig. 2.20.

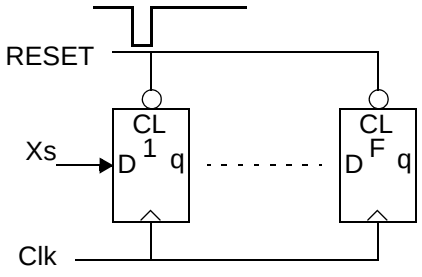


Figura 2.20: Inicio en S_0

En el instante inicial el sistema está apagado (OFF). Tras el encendido (puesta en ON), la fuente de alimentación genera V_{cc} que va subiendo hasta su valor estable. A partir de cierto valor de V_{cc}, todos los dispositivos están operativos, RESET se hace/mantiene a 0 por lo que el controlador está en S_0. Transcurrido un pequeño intervalo de tiempo, RESET pasa a 1 por lo que la unidad de control empieza a funcionar síncronamente estando en S_0.

Adecuación de la señal de comienzo Xs

La señal de comienzo Xs es aquella que, al activarse, saca al sistema digital del estado de espera S_0 y causa que comience la ejecución de la operación deseada. En la Fig. 2.21 se recuerda el papel de la señal Xs y la solución propuesta para la inicialización de la operación. La señal Xs es un pulso que:

- No puede durar menos de un pulso de reloj, ya que de esta forma puede darse el caso de que el primer biestable no capte el uno, lo cual está representado en la Fig. 2.22.a.
- No puede ser arbitrariamente larga, en el sentido de que dure muchos ciclos de reloj, porque

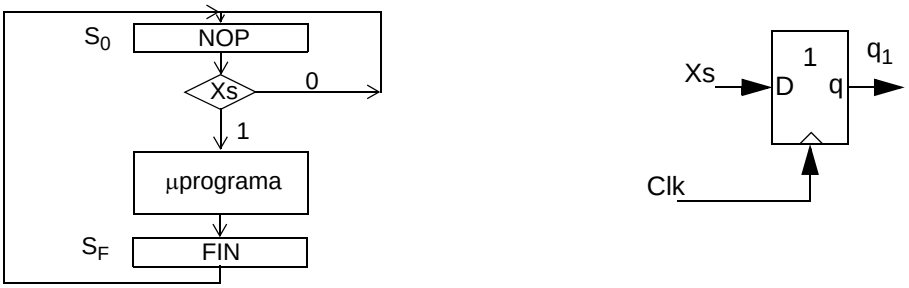


Figura 2.21: Inicio de la operación cuando Xs se activa y solución de circuito adoptada

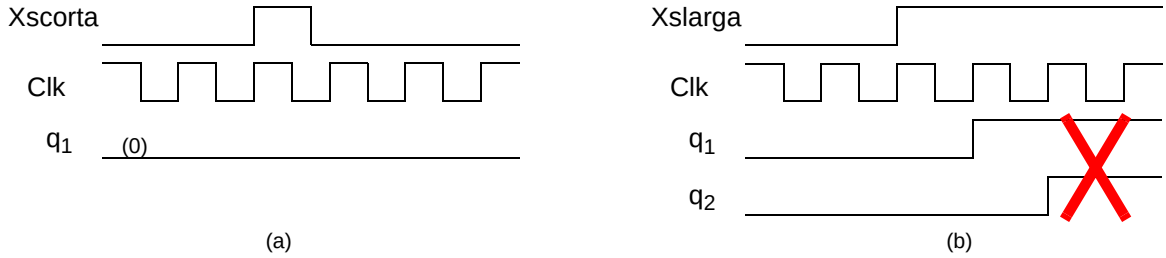


Figura 2.22: Anchuras de pulso no válidos en Xs: a)demasiado corto; b)demasiado largo

puede darse el caso de que más de un biestable esté a uno con lo que se está violando el asignamiento "one-hot" y, por lo tanto, también se violan todas las consideraciones seguidas en la obtención de las salidas adecuadas en el controlador. En la Fig. 2.22.b se ilustra cómo si se capturan dos unos consecutivos de Xs se produce la situación prohibida de q_1q_2= 11

En consecuencia, la duración adecuada en Xs es de un ciclo de reloj para que: 1) Siempre sea captada por el primer biestable, que es el que define el estado S_1, y 2) no se produzcan varias transferencias entre registros en el mismo ciclo de reloj con potenciales colisiones, lo que no está permitido.

Vamos a describir a continuación soluciones que nos permiten obtener la señal Xs adecuada (Xsciclo) tanto si disponemos de una señal demasiado larga (Xslarga) como si ésta es demasiado corta (Xscorta). Adelantemos que para estas últimas, la solución es obtener una Xslarga a partir de la señal Xs corta y, después, acondicionar Xslarga a que dure sólo un ciclo de reloj.

Solución a señales arbitrariamente largas

Sea Xslarga la señal arbitrariamente larga y Xs la señal Xsciclo que deseamos obtener. En el ejemplo de la carta ASM de la Fig. 1.39 se planteaba una forma de obtener Xsciclo a partir de un protocolo de dos señales, LISTO y YA, ambas arbitrariamente largas. Ahora planteamos una solución basada en dos biestables D para obtener Xs a partir de Xslarga. El circuito mostrado en la Fig. 2.23 proporciona la señal correcta, tal como se observa en el diagrama temporal de la misma figura.

En esta solución, si se desea, la señal XS puede actuar como variable q_1.

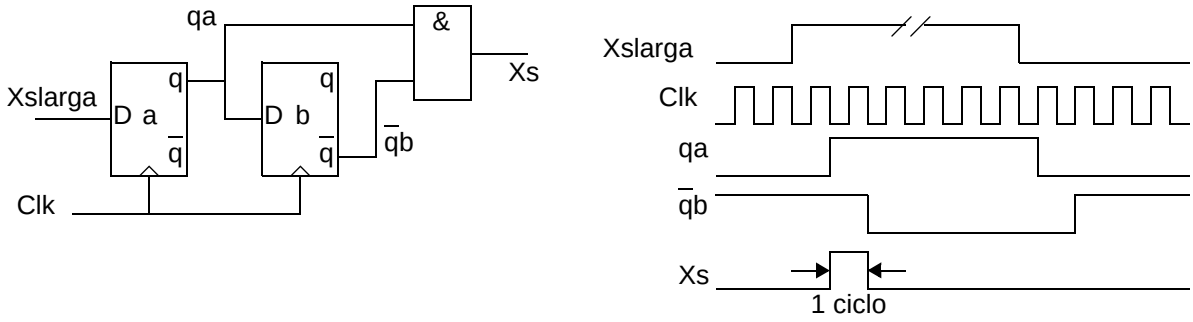


Figura 2.23: Solución para señal arbitrariamente larga, Xslarga

Solución a señales arbitrariamente cortas

Sea XC la señal arbitrariamente corta (pero siempre mayor que una anchura mínima). La solución consiste en, a partir de ella, obtener una señal arbitrariamente larga Xslarga y aplicar la solución anterior. Hay varias formas de realizar esto:

- Usando un monoestable. Un monoestable o “one shot” es un circuito al que se le conectan externamente una resistencia (R) y un condensador (C), y que, ante un pulso de entrada, responde con un pulso de salida de duración T cuyo valor es función del producto RC, T=f(RC) (ver Fig. 2.24). En nuestro caso se eligen R y C de forma que sea mayor que un ciclo de reloj del sistema.

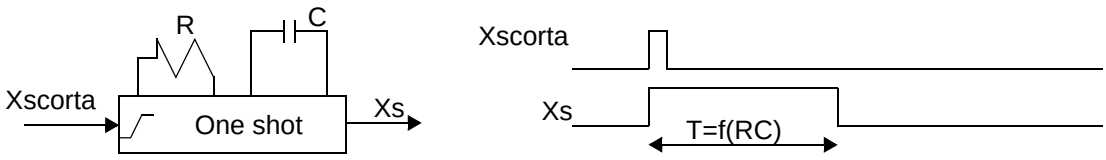


Figura 2.24: Monoestable usado para alargar el pulso de Xscorta y su forma de onda entrada-salida

Sin embargo, existen varios inconvenientes en esta solución. Uno de ellos es que los valores de R y C son poco precisos y varían mucho con las condiciones de operación (temperatura, frecuencia, ...). En particular para diseños a integrar (hacer un chip), estos problemas se agravan y tienen el inconveniente añadido del coste generalmente alto en área de chip para integrar R's y C's. Además la dependencia de T con RC no es lineal. Por último, no es fácil la descripción lógica del comportamiento del monoestable.

- Usando un biestable asíncrono y una señal de BORRAR. La señal Xscorta cambia el estado del biestable asíncrono; el estado adquirido se mantendrá hasta que el biestable no reciba una "orden" de cambiar al estado inicial con la señal de BORRAR (la orden la da el sistema). El circuito y el diagrama temporal correspondiente a esta solución se muestra en la Fig. 2.25.

- Usando un biestable tipo D síncrono y una señal de BORRAR: La señal Xscorta se utiliza como "reloj" del biestable síncrono para cargar el estado 1 (dato de entrada constante); el sistema vuelve al estado 0 inicial de forma asíncrona con BORRAR. La solución se muestra en la Fig. 2.26.

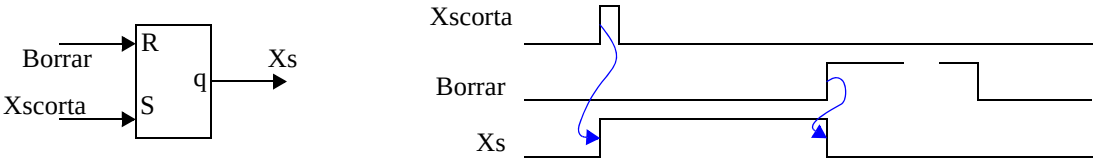


Figura 2.25: Solución mediante un biestable asíncrono

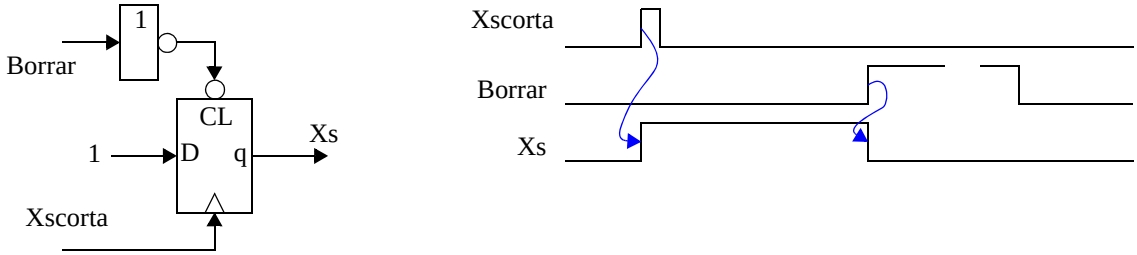


Figura 2.26: Solución mediante un biestable síncrono

Las dos últimas soluciones tienen como rasgo que hay que generar la señal BORRAR. Proponemos las siguientes soluciones para la generación de esta señal.

- Solución 1: Utilizar la propia señal XS como señal de BORRAR. En la Fig. 2.27 se muestra un diagrama temporal que ilustra esta solución, hay que tener en cuenta el circuito de la Fig. 2.23 donde se obtenía la señal XS a partir de Xslarga. Obsérvese que la señal XL podría durar menos de un ciclo de reloj. Ello no es problema pues Xslarga sólo vuelve a 0 cuando su valor 1 ya ha sido captado por el biestable "a" de la Fig. 2.23.

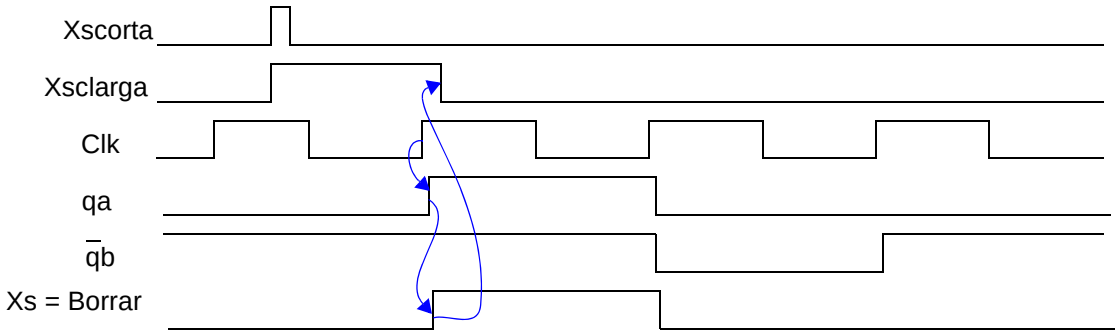


Figura 2.27: Solución 1. Se usa Xs como señal ‘Borrar’ en los circuitos de Fig. 2.25 o Fig. 2.26

- Solución 2: Que sea la misma unidad de control la que genere la señal Borrar. Se haría en cualquier microoperación común a todos los caminos del microprograma, por ejemplo la última microoperación del sistema (FIN = Borrar). Esta solución es ilustrada en la Fig. 2.28.

Para terminar este apartado vamos a estudiar el problema de cómo una persona puede generar la señal de comienzo. El mecanismo de actuación es que la persona acciona un botón que puede ser un pulsador de tres terminales, es decir que con una sola acción pasa de una posición a otra, volviendo al cabo de cierto tiempo a la posición inicial. Un mecanismo de este tipo presenta problemas de balanceo ya que al establecer el contacto y al abandonarlo se producen muchos "rebotes", cuyo número y duración, aunque acotados, son desconocidos. Para eliminar los "rebotes" se utiliza un biestable RS

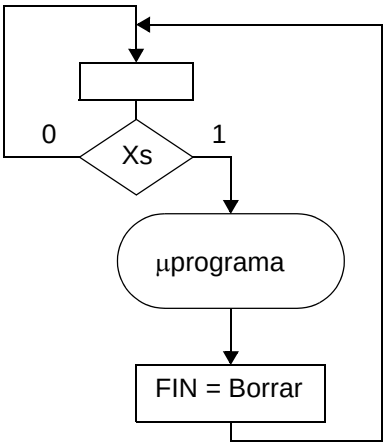


Figura 2.28: Solución 2: Borrar se obtiene de la señal de FIN

asíncrono[1] de la forma mostrada en la Fig. 2.29. Asumamos que en el instante inicial el botón está en la posición de RESET: En (1) hay un RESET por lo que q es 0. En (2) la placa-contacto del pulsador se junta/separa de A; en el biestable genera R S = x 0, por lo que q es 0 tanto si R es 0 como si es 1. En (3) la placa está entre A y B por lo que RS = 00 y el estado próximo es igual al estado presente (Q=q). En (4) la placa se junta/separa de B; en el biestable, la primera vez que S = 1 produce que q se haga 1 y después, durante los rebotes será R S = 0 x, por lo que q sigue en 1. En (5) hay un buen contacto entre la placa y B, por lo que RS = 01 y q es 1. Así q = Xslarga es una señal libre de rebotes.

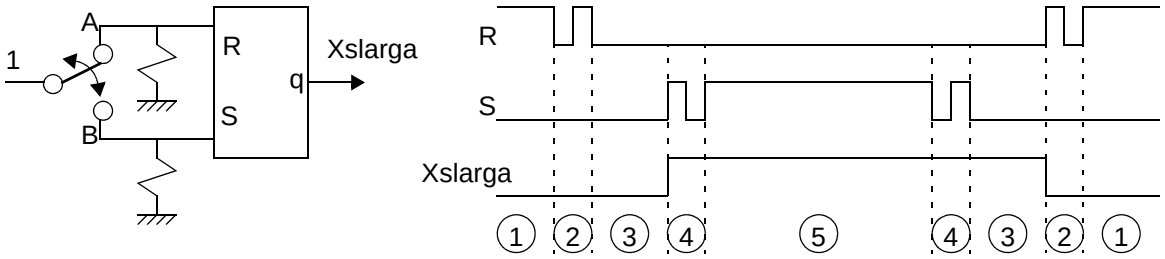


Figura 2.29: Solución sin rebote ante botones que sí los presentan

De esta forma, por la acción de la persona se genera una señal larga Xslarga que, junto al circuito de la Fig. 2.23, proporciona la señal de comienzo adecuada Xs.

1. Una solución alternativa es mediante software. En efecto, aunque desconocida, la duración de los rebotes (2+3+4) puede acotarse t_{234}; en cualquier caso esta duración es mucho menor que la de la región 5 (o 1). La solución software consiste en detectar el valor en A (B) y, cuando hay un 1, espera un tiempo mayor que t_{234}, vuelve a detectar el valor de A (B) y, sólo si vuelve a detectar un 1, acepta ese 1 como valor válido.

2.6 OTROS TIPOS DE REALIZACIÓN

En esta sección presentamos otras formas de realización de unidades de control. En primer lugar plantearemos cómo desarrollar controladores usando el número mínimo de biestables y multiplexores (MUX) para generar fácilmente el flujo de estados. Posteriormente introducimos los controladores microprogramados considerendo los diseños con registro y PLA, con registro y PAL, y con registro y ROM, terminando con una breve introducción al *firmware* (*firmware*: es un programa de microinstrucciones para propósitos específicos, grabado en un dispositivo programable; ocupa un nivel intermedio entre el *sotware* y el *hardware*).

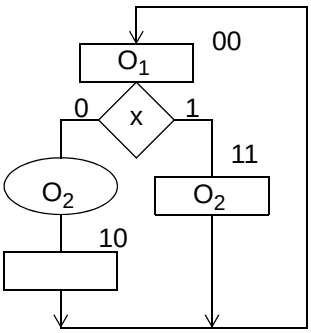
2.6.1 Implementación con multiplexores y flip-flops D

En este tipo de diseño el asignamiento se realiza de forma similar a la del apartado 2.3. Seleccionamos un número mínimo (n=$\lceil \log_2 N \rceil$) de flip-flops D para las variables de estado y hacemos una asignación de valores binarios para los estados de la carta ASM.

Dada la codificación del estado presente y las variables de entrada, debemos producir el valor adecuado en la codificación para el próximo estado y las activaciones de las salidas. Podemos distinguir, pues, dos partes en la tarea de diseño:

- La función de próximo estado se realiza mediante un multiplexor conectado a la entrada de cada flip-flop. El código del estado presente (almacenado en éstos) alimenta las entradas de selección de los multiplexores que, en consecuencia, tendrán que ser MUX 2^n:1. La obtención de las entradas de datos en cada multiplexor se realiza construyendo tablas de transición a partir de la carta ASM de la forma indicada más adelante, en la realización de ejemplos.
- La obtención y realización de las ecuaciones de salida del controlador. La realización de las salidas puede hacerse en dos niveles de puertas tal como vimos en la lógica discreta. Pero hay otra alternativa, que es utilizando un decodificador de forma que nos permita decodificar las variables de estado presente con lo que la salidas del decodificador se activarán para cada uno de los estados del controlador. De esta forma hemos convertido la codificación para los estados en una codificación 'one-hot' y por lo tanto a partir de las salidas del decodificador, las salidas del controlador se obtienen de la misma forma que en el apartado 2.4.

Antes de enfrentarnos a la tarea de diseñar el controlador de la calculadora, tal como hemos venido haciendo hasta ahora, vamos a realizar un ejemplo más sencillo. Consideremos la carta ASM de la Fig. 2.30, donde la codificación de cada estado se muestra en el bloque correspondiente (en esta sección no discutiremos cómo afecta la asignación al coste). Las salidas son O_1 y O_2 y la entrada es x.



Estado Presente q_1 q_0	Próximo Estado Q_1 Q_0	Condición para transición
0 0	1 0	X'
0 0	1 1	X
1 0	0 0	1
1 1	0 0	1

Figura 2.30: Ejemplo 1 con MUX: Carta ASM y tabla de transición

Para la realización de nuestro diseño necesitamos dos biestables D y, por tanto, dos multiplexores de 4 canales, además de la lógica combinacional para la obtención de las salidas. De la carta ASM se pasa directamente a la tabla de transición de la Fig. 2.30.

La primera variable de estado tomará el valor $Q_1 = 1$ cuando el estado presente sea q_1q_0=00 y el valor 0, en el caso q_1q_0=1-. El valor de Q_0 dependerá de "x" cuando el estado presente sea q_1q_0=00 y será 0 en el caso q_1q_0=1-. Como utilizamos biestables D, las entradas de excitación de éstos coinciden con el valor de próximo estado (Di=Qi), con lo que la siguiente tabla refleja las entradas de datos que hay que poner en cada MUX:

q_1q_0	Q_1	Q_0
0 0	1	x
0 1	-	-
1 0	0	0
1 1	0	0

En cuanto a las salidas, O_1 se activa (su valor es 1), cuando el estado del circuito es el 00. Por su parte, O_2 se activa cuando el estado es el 00 y x = 0 y cuando el estado es el 11. En la implementación del circuito (mostrada en la Fig. 2.31) hemos usado un decodificador y puertas para la obtención de las salidas.

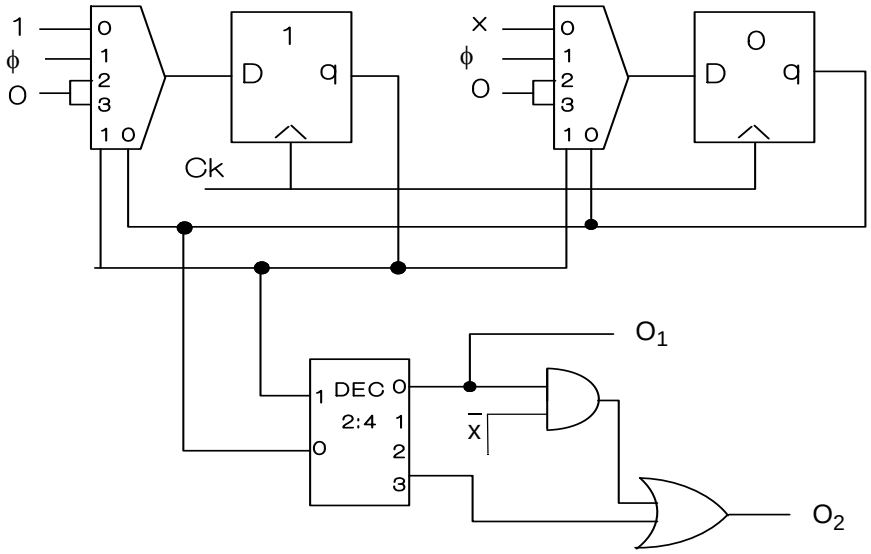


Figura 2.31: Ejemplo 1 de la realización con MUX

Hagamos ahora un diseño de este tipo para la carta ASM de nuestro ejemplo de la Fig. 2.2. El asignamiento para los estados es:

S_0: 000 S_1: 001 S_2: 010 S_3: 011 S_4: 100 S_F: 101

Necesitamos tres biestables D y tres multiplexores de ocho canales. De la carta ASM de la Fig. 2.2 obtenemos la tabla de transición que se muestra a continuación:

Estado Presente $q_2\ q_1\ q_0$	Estado Próximo $Q_2\ Q_1\ Q_0$	Condición
0 0 0	0 0 0	x'_s
0 0 0	0 0 1	x_s
0 0 1	0 1 0	1
0 1 0	0 1 1	1
0 1 1	1 0 0	1
1 0 0	1 0 1	1
1 0 1	0 0 0	1

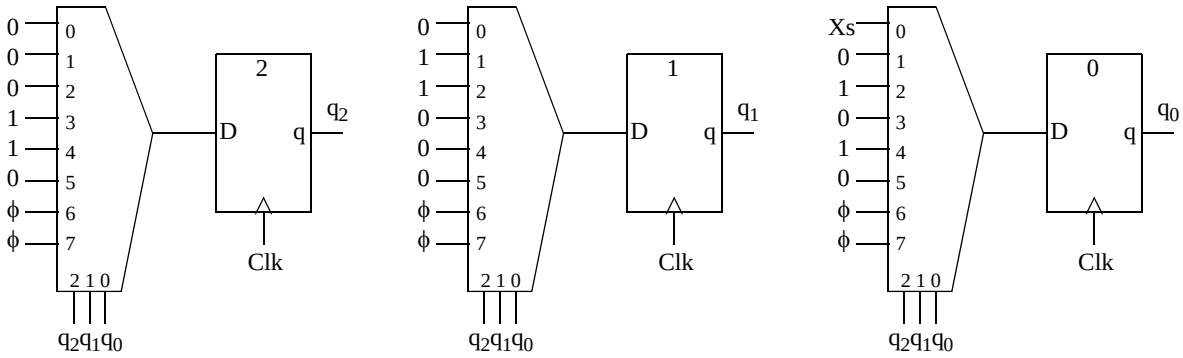
A partir de esta tabla se obtiene la siguiente, que muestra la lógica de excitación de los biestables mediante MUX..

$q_2q_1q_0$	Q_2	Q_1	Q_0
0 0 0	0	0	x_s
0 0 1	0	1	0
0 1 0	0	1	1
0 1 1	1	0	0
1 0 0	1	0	1
1 0 1	0	0	0
1 1 0	-	-	-
1 1 1	-	-	-

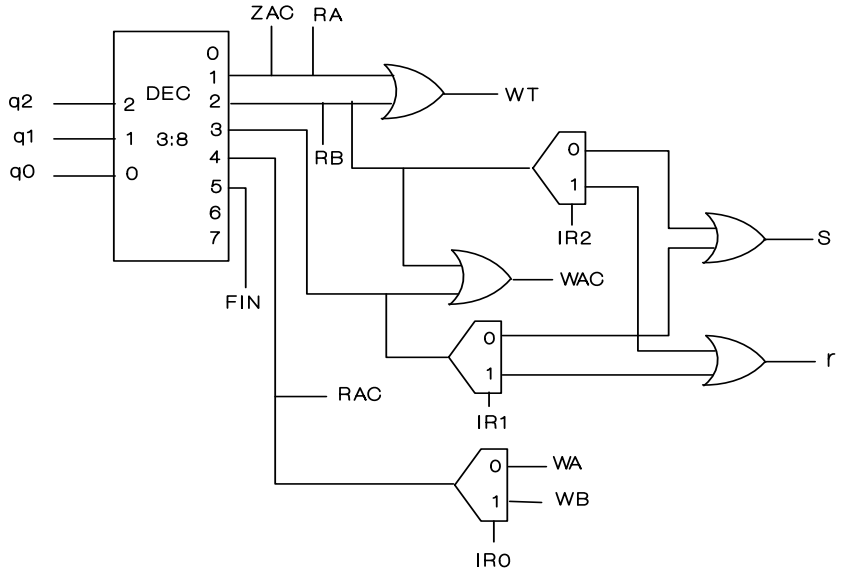
De ella se deriva la implementación mostrada en la Fig. 2.32.a. Para la realización de las salidas hemos utilizado un DEC 3:8 en el nivel de entrada. Se puede observar que las salidas 1, 2, 3, 4 y 5 del mismo (Fig. 2.32.b) tienen los mismos valores que las salidas de los biestables de la realización basada en un registro de desplazamiento (q_1, q_2, q_3, q_4, q_F, en la Fig. 2.19). La lógica combinacional usada en el segundo nivel para la obtención de las salidas del controlador en la Fig. 2.32.b coincide, pues, con la lógica combinacional para la obtención de las salidas de la Fig. 2.19.

Comentarios:

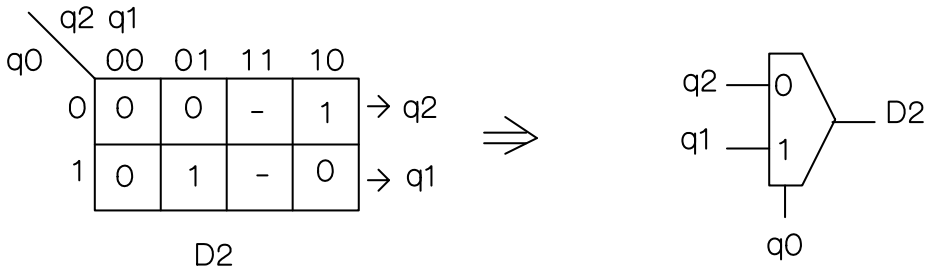
Este método es de fácil realización. Sin embargo, es poco económico ya que en muchas ocasiones no se hace un buen uso de los multiplexores. Por ejemplo el MUX (8:1) que proporciona la entrada D2 en la Fig. 2.32.a puede ser sustituido por un MUX (2:1) como se muestra en la Fig. 2.32.c. Además el circuito no puede ser realizado directamente a partir de la carta ASM, puesto que hay hay dos pasos previos que son las tablas de transición. En algunas ocasiones, para obtener el próximo estado, además de multiplexores habrá que poner uno o varios niveles previos a éstos de puertas. El controlador resultante está "alejado del algoritmo".



(a)



(b)



(c)

Figura 2.32: Controlador de la calculadora: a)Realización de estados con MUX y biestables D; b)Señales de salida; c)Reducción en el MUX del biestable 2

2.6.2 Realización con dispositivos lógicos programables PLDs.

El término PLD (*Programmable Logic Design*) engloba a un importante grupo de dispositivos digitales cuyo hardware puede personalizado por el usuario según la aplicación concreta que desee implementar. Esto es, estos dispositivos pueden ser programados, lo que da lugar a la *lógica programada*, mecanismo de implementación digital alternativo a la *lógica cableada*. Los principales PLDs de operación simple son: PLA (*Programmable Logic Array*), PLA (*Programmable Array Logic*), GAL (*Generic Array Logic*) y ROM (*Read Only Memory*); los más complejos son: CPLD (*Complex PLD*) y FPGA (*Field Programmable Logic Array*), llegando a tener millones de puertas.

En esta sección desarrollaremos las técnicas de diseño de unidades de control basadas en un PLD simple (PLA, PAL o ROM), en el que *se escribirá* el microprograma de control, y un registro de carga en paralelo, en el que se almacenará el código de la microoperación presente.

2.6.2.1 Realización con PLA

En el diseño con PLA's, éste se utiliza para realizar las funciones combinacionales (de excitación y de salida), mentras que un conjunto de biestables D (o registro de carga paralelo/paralelo) se usa para el almacenamiento de estados. La principal consideración de diseño es reducir el número de terminales (de entrada y de salida) del PLA, para lo cual:

- Los biestables son tipo D con el fin de suministrar sólo una señal de excitación por biestable.
- La asignación de estados se realiza utilizando el menor número de variables, minimizando así el número de biestables y, por lo tanto, el de terminales de entrada y de salida .

La arquitectura del controlador con PLA es la mostrada en la Fig. 2.33. Las n+k entradas al PLA corresponden a las n entradas de control y a las k variables de estado presente (salidas q de los biestables) utilizadas en la carta ASM. Las k+m salidas proporcionan las m salidas de control (señales de comandos) y el código del próximo estado (cuyo número y valor coincide con las entradas de excitación al usar biestables D: Qi=Di para los k biestables).

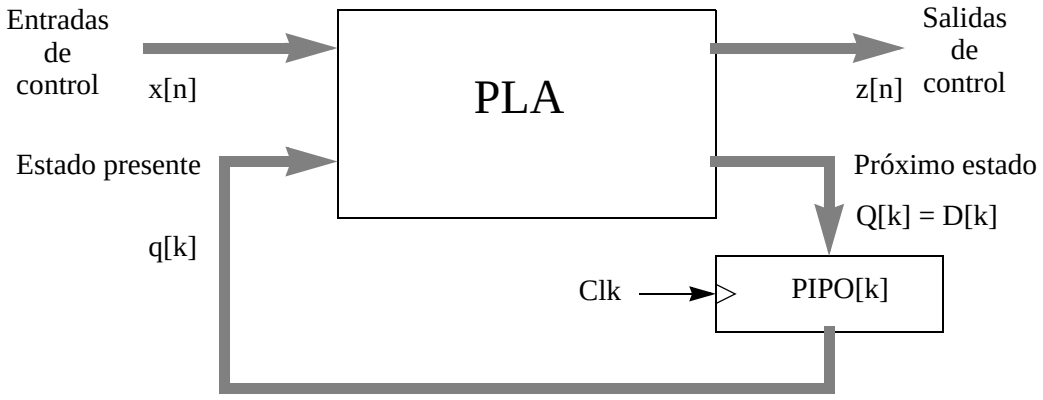


Figura 2.33: Arquitectura del controlador con PLA

La realización del controlador admite dos estrategias:

1. Partiendo de la carta ASM se puede obtener directamente la personalización del PLA, tal como explicaremos a continuación. El PLA resultante no está optimizado. Sin embargo, la asignación de estados no influye en el coste de diseño, por lo que no hay que tenerla en cuenta.

2. Partiendo de la carta ASM se procede como en el diseño con lógica discreta (apartado 2.3). Para obtener el PLA mínimo hay que hacer una asignación de estados óptima y obtener las expresiones mínimas en dos niveles de las k+m funciones de salida (ecuaciones de excitación/salida) cada una de n+k variables de entrada. Este proceso es, en general muy complejo y costoso en tiempo, aumentando la complejidad de forma no lineal con k, n y m.

La diferencia de coste entre las dos soluciones está en el número de términos productos a generar en el plano AND del PLA, ya que el número de terminales permanece constante en ambos métodos. No se puede generalizar diciendo qué solución es mejor cuando se incorpora como criterio la facilidad del proceso de diseño, mejor en la estrategia 1 que en la 2. En general, dependerá de cada caso y de las herramientas de síntesis que se disponga. Aquí nos limitaremos a la aplicación a un ejemplo concreto y la comparación de las dos soluciones en ese caso.

Estrategia 1

Describamos ahora cómo se efectúa directamente de la carta ASM el diseño del PLA. La Fig. 2.34.a muestra un "trozo" de carta ASM de la que se obtiene la solución PLA de la Fig. 2.34.b de la siguiente forma:

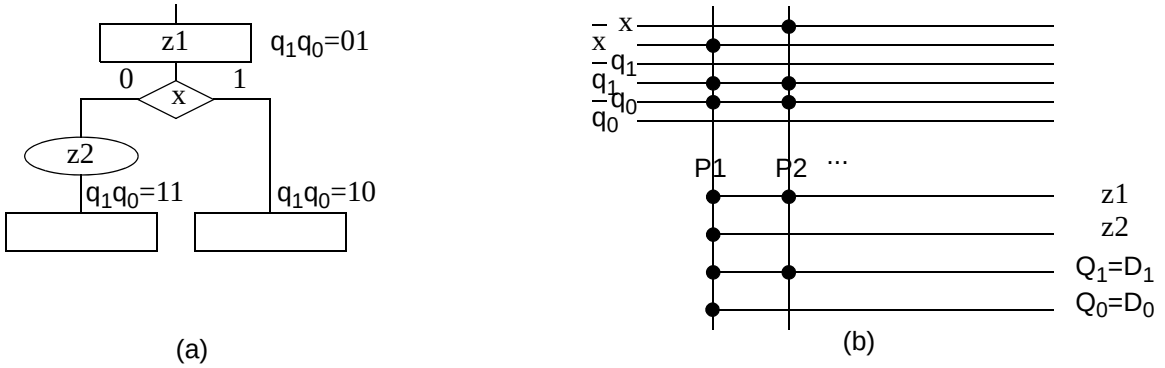


Figura 2.34: Ilustración del paso directo de a)una carta ASM a b)la personalización sobre PLA

- Para cada bloque ASM se generan tantos términos productos (AND) como caminos distintos haya en el bloque. En la figura 34a, el bloque con estado 01 tiene dos caminos, por lo que se generan dos términos producto, P1 y P2. Cada uno de ellos se activa en el correspondiente estado (en este caso, q_1q_0=01) y valores de control activos en ese camino (en este caso, x=0 para un camino y x=1 para el otro). Así:

$$P1 = \overline{x} \cdot \overline{q}_1 \cdot q_0$$
$$P2 = x \cdot \overline{q}_1 \cdot q_0$$

- Cada variable de salida, tantro las de tipo comando (en este caso, z1 y z2) como las de excitación de próximo estado que se active en cada camino contiene el término producto correspondiente (en este caso, Q_1 en ambos casos y Q_0 sólo en P2). Es importante hacer notar que si una salida se activa en varios caminos del bloque debe contener todos los términos produc-

tos respectivos: por ejemplo z1 debe contener P1 y P2.

Como ejemplo de realización de un controlador de este tipo volvamos a la carta ASM de la Fig. 2.2. La arquitectura es la representada en la Fig. 2.33. La asignación binaria considerada es la habitualmente usada (S_0 : 000; S_1 : 001; S_2 : 010; S_3 : 011; S_4 : 100; y S_F : 101) por lo que el registro PIPO es de 3 bits. Necesitamos un PLA con 7 entradas (XS, IR_2, IR_1, IR_0, q_2, q_1 y q_0) y 14 salidas (los 11 comandos: ZAC, WT, RA, RB, WAC, s, r, RAC, WA, WB y FIN, y 3 las entradas de carga en paralelo del registro PIPO: D_2, D_1 y D_0).

La personalización de esta unidad de control se realiza a partir de la carta ASM aplicando el método expuesto. Considerando todos los caminos de cada bloque ASM de esa carta, se obtienen los siguientes términos producto. La notación utilizada es la siguiente: en primer lugar aparece el bloque ASM de procedencia con su código binario y variable de decisión; después, el término producto; por último, una lista de las señales que debe activar el término producto.

$S_k=q_2q_1q_0$	x	Término P_K	Salidas afectadas
$S_0 = 000$	$\overline{Xs}$	(Ninguno)	(Ninguna)
	Xs	$P_1 = X_s\overline{q_2}\,\overline{q_1}\,\overline{q_0}$	Q_0
$S_1 = 001$	t	$P_2 = \overline{q_2}\,\overline{q_1}q_0$	Q_1, ZAC/WT/RA
$S_2 = 010$	$\overline{IR_2}$	$P_3 = \overline{IR_2}\,\overline{q_2}q_1\overline{q_0}$	Q_1Q_0, WT/RB/WAC/s
	IR_2	$P_4 = IR_2\overline{q_2}q_1\overline{q_0}$	Q_1Q_0, WT/RB/WAC/r
$S_3 = 011$	$\overline{IR_1}$	$P_5 = \overline{IR_1}\,\overline{q_2}q_1q_0$	Q_2, WAC/s
	IR_1	$P_6 = IR_1\overline{q_2}q_1q_0$	Q_2, WAC/r
$S_4 = 100$	$\overline{IR_0}$	$P_7 = \overline{IR_0}q_2\overline{q_1}\,\overline{q_0}$	Q_2Q_0, RAC/WA
	IR_0	$P_8 = IR_0q_2\overline{q_1}\,\overline{q_0}$	Q_2Q_0, RAC/WB
$S_5 = 101$	t	$P_9 = q_2\overline{q_1}q_0$	FIN

Los términos producto son programados en el plano AND del PLA que tendrá que ser como mínimo de 9 términos. El plano OR se programa para que cada salida contenga sus términos producto correspondientes. El resultado se muestra en la Fig. 2.35.a, donde el registro PIPO ha sido desarrollado en sus tres biestables.

Estrategia 2

En el caso de nuestro ejemplo se ha obtenido otro diseño mediante la estrategia de realización 2 explicada anteriormente, consistente en optimizar conjuntamente las expresiones dos niveles de las salidas afectadas como un problema de optimización multisalida. Para asegurarnos de que la minimización es la óptima y debido a la complejidad del sistema se ha usado un paquete 'software' (ESPRESSO) que nos da las ecuaciones de excitación/salida de menor coste basándose en el método de Quine-McCluskey para la minimización de funciones multisalida.

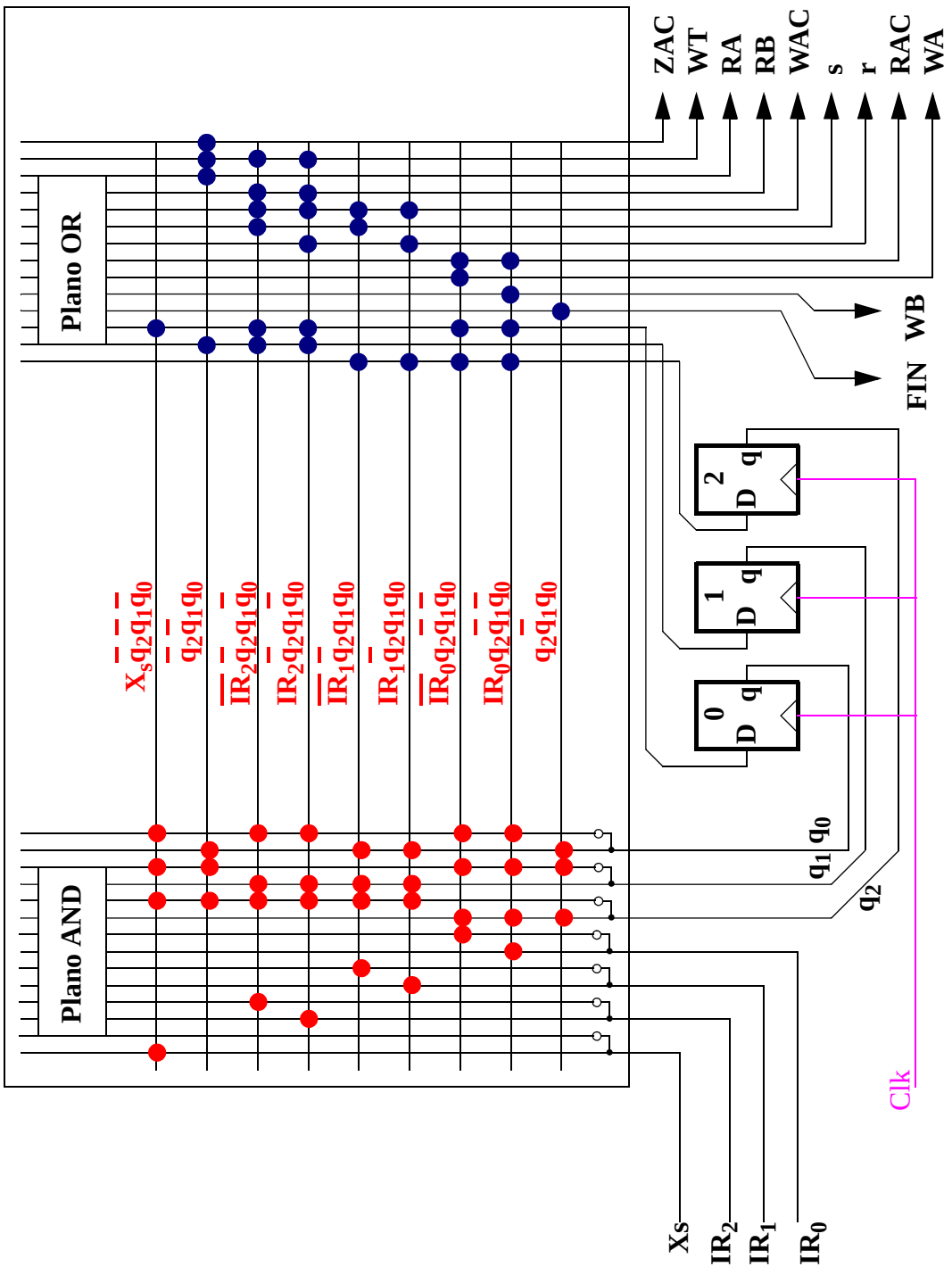


Figura 2.35: a)Realización con PLA del controlador de la calculadora por la estrategia 1 (método directo)

Las ecuaciones obtenidas y la correspondiente realización mediante el PLA son:

$D_2 = Xs\cdot\overline{q}_1\cdot\overline{q}_0 + IR_2\cdot\overline{q}_2\cdot q_0 + \overline{IR}_2\cdot\overline{q}_2\cdot q_0 + IR_0\cdot\overline{q}_2\cdot q_1 + \overline{IR}_0\cdot\overline{q}_2\cdot q_1$

$D_1 = Xs\cdot\overline{q}_1\cdot\overline{q}_0 + IR_2\cdot\overline{q}_2\cdot q_0 + \overline{IR}_2\cdot\overline{q}_2\cdot q_0 + IR_1\cdot q_2\cdot\overline{q}_0 + \overline{IR}_1\cdot q_2\cdot q_0$

$D_0 = Xs\cdot\overline{q}_1\cdot\overline{q}_0 + \overline{IR}_0\cdot\overline{q}_2\cdot q_1 + IR_0\cdot\overline{q}_2\cdot q_1 + q_1\cdot q_0$

$WAC = IR_2\cdot\overline{q}_2\cdot q_0 + \overline{IR}_2\cdot\overline{q}_2\cdot q_0 + IR_1\cdot q_2\cdot\overline{q}_0 + \overline{IR}_1\cdot q_2\cdot\overline{q}_0$

$WT = IR_2\cdot\overline{q}_2\cdot q_0 + \overline{IR}_2\cdot\overline{q}_2\cdot q_0 + q_1\cdot q_0$

$RB = IR_2\cdot\overline{q}_2\cdot q_0 + \overline{IR}_2\cdot\overline{q}_2\cdot q_0$

$r = IR_2\cdot\overline{q}_2\cdot q_0 + IR_1\cdot q_2\cdot\overline{q}_0$

$ZAC = q_1\cdot q_0$

$RAC = IR_0\cdot\overline{q}_2\cdot q_1 + \overline{IR}_0\cdot\overline{q}_2\cdot q_1$

$RA = q_1\cdot q_0$

$WA = \overline{IR}_0\cdot\overline{q}_2\cdot q_1$

$WB = IR_0\cdot\overline{q}_2\cdot q_1$

$s = \overline{IR}_2\cdot\overline{q}_2\cdot q_0 + \overline{IR}_1\cdot q_2\cdot\overline{q}_0$

$FIN = q_2\cdot\overline{q}_1$

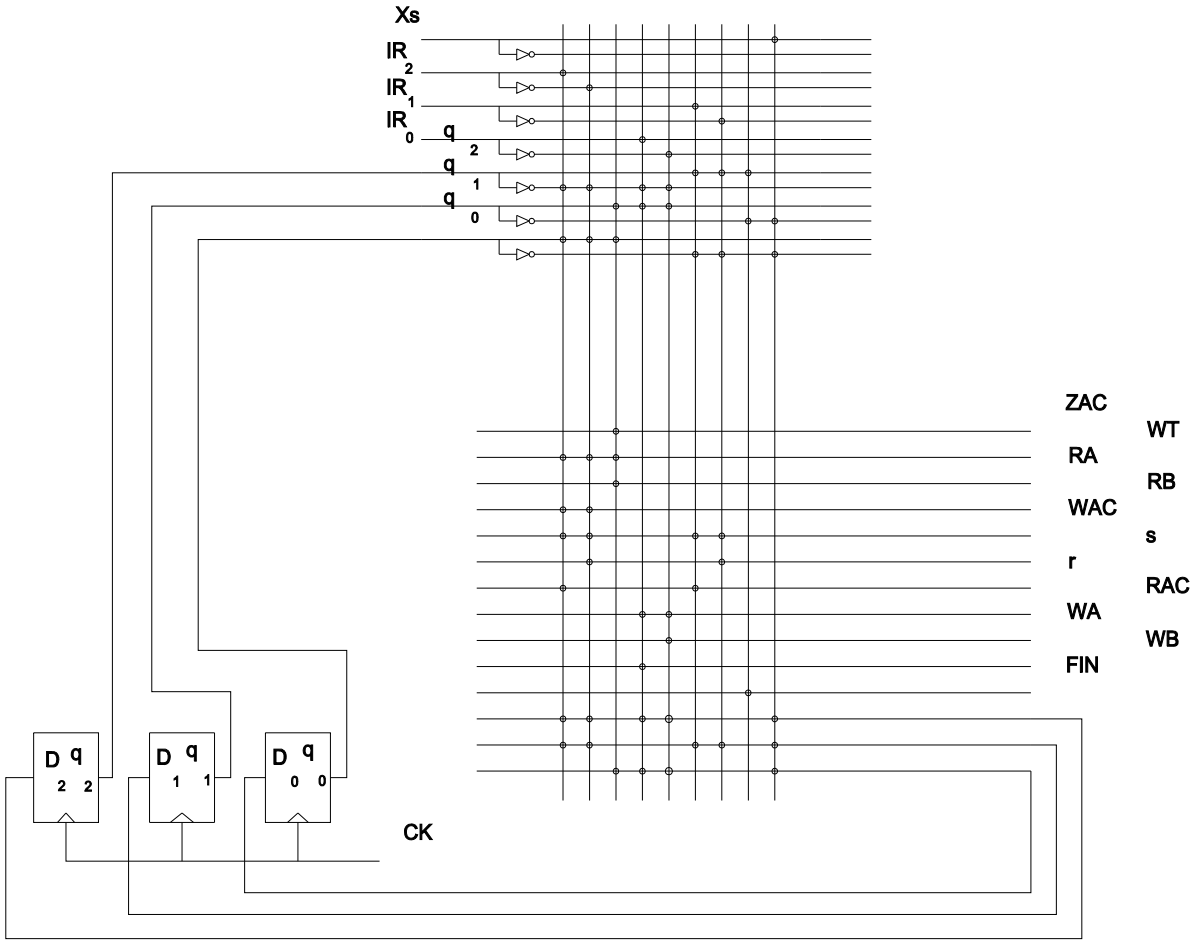


Figura 2.35: b)Realización con PLA por la estrategia 2 (optimización con ESPRESSO)

Puede observarse que la optimización con ESPRESSO ha dado lugar en este caso a numerosos términos producto compartidos entre las funciones de salida, lo cual es un indicador de una buena optimización. Sin embrago, el tamaño del PLA ha resultado ser el mismo que para la realización anterior (Fig. 2.35.a), incluso en número de líneas AND ocupadas. Por otra parte, la optimización con ESPRESSO resulta mucho más costoso en tiempo incluso con la ayuda de ese *software*.

Comentarios:

Se obtiene una implementación directa de la carta ASM al PLA. La implementación directa da lugar a PLA no optimizados. La optimización del PLA implica un esfuerzo de diseño (minimización de funciones de múltiples salidas) que, en muchos controladores, no conduce a una ganancia en el plano AND significativa. Ello es debido a que en estos sistemas las decisiones se toman bajo muy pocas variables de entrada (frecuentemente 1 ó 2 de las n), por lo que ya existe una simplificación implícita en el método directo (las entradas que no intervienen ya están "eliminadas").

2.6.2.2 Realización con PLA

El diseño con PAL es equivalente al diseño con PLA realizado en el apartado anterior, con la diferencia de que las PALs son menos flexibles al poseer un plano OR fijo. Esto supone que habría que adaptar las ecuaciones dependiendo del PAL utilizado.

Por otro lado existen PALs que incorporan biestables, lo que puede permitir la implementación del controlador en un único chip.

2.6.2.3 Realización con ROM

La estructura general de un controlador basado en ROM es como la de la Fig. 2.33 sustituyendo el PLA por la ROM. Las n variables de entrada junto con las k variables de estado presente proporcionan la dirección de la palabra de la ROM a la que se accede. El contenido de cada palabra deberá suministrar los k bits de próximo estado y las m señales de comando. Por tanto se requiere una ROM de $2^{k+n}x(k+m)$ bits.

Sin embargo, si hacemos directamente la implementación tal como acabamos de señalar, no es eficiente ni la solución, ni el proceso de diseño.

En efecto, a diferencia de lo que ocurre en el PLA, las entradas a la ROM están totalmente decodificadas habiendo una posición de memoria por cada estado total (entrada y estado presente). Sin embargo, en la mayoría de los microprogramas, en cada microoperación no son significativas la mayoría de las combinaciones de las entradas (control/estado). Esas combinaciones no significativas suponen repetir el contenido de muchas palabras de la ROM innecesariamente. Además, para obtener una realización directa con ROM desde carta ASM necesitamos expandir las entradas para cubrir todas las posibles combinaciones de entrada, lo que es un paso adicional en el proceso de diseño. Por otra parte, en cada microoperación tampoco son significativas todas las salidas, ya que sólo un pequeño conjunto de ellas debe activarse cada vez. Esto implica un mal aprovechamiento de la anchura de la memoria: salvo unos pocos cada vez, gran parte de los bits de cada palabra serán 0. A continuación desarrollaremos más ampliamente estas cuestiones y mostraremos cómo se realiza el control con ROM.

Consideremos la carta ASM de la Fig. 2.36. La carta ASM definida en el ejemplo tiene 3 entradas y tres variables de estado por lo que son posibles $2^6 = 64$ combinaciones de valores. Por otro lado, hay 3 salidas, a las que se añade los 3 bits de próximo estado, haciendo un total de 6 salidas. Se precisa, pues, una ROM de 6 líneas de dirección (64 palabras) de, al menos, 6 bits por palabra.

Realicemos el diseño con registro PIPO (biestables D) y ROM. La tarea de diseño consiste en determinar el contenido de la ROM, lo que puede realizarse casi directamente de la carta ASM de la siguiente forma:

1. En primer lugar, elegimos las entradas de dirección de ROM (A_5:A_0), que serán, desde MSB a

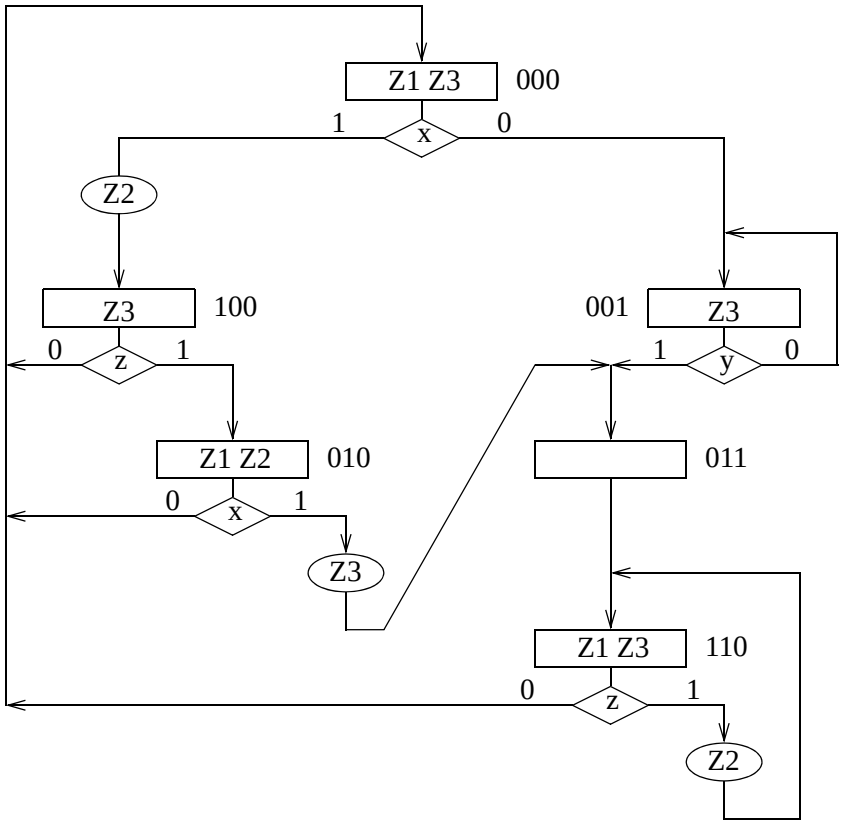


Figura 2.36: Ejemplo de carta ASM para la implementación con ROM

LSB[1]: x, y, z, q_2, q_1 y q_0.

2. Para cada bloque ASM se expanden las variables de decisión del bloque a todas las restantes variables de entrada. Por ejemplo, para el bloque 000 que contiene sólo la variable de decisión x, se expande a:

a) Para x=0: x y z = 000, 001, 010, y 011 y b) Para x=1: x y z = 100, 101, 110 y 111

3. Para cada valor de expansión se le añaden los valores de estado de la microoperación y se forman, así, los conjuntos de direcciones de cada rama. En nuestro caso-ejemplo, el estado es $q_2q_1q_0$ = 000:

   a) Para x=0: A_5:A_0 = 000000, 001000, 010000, y 011000 = $00, 08, 10, y 18

   b) Para x=1: A_5:A_0 = 100000, 101000, 110000 y 111000 = $20, 28, 30, y 38

4. Para cada dirección el contenido de la palabra direccionada de la ROM será el vector de entrada al registro PIPO y de las salidas (D_2, D_1, D_0, Z1, Z2 y Z3) correspondiente. Ese contenido es el mismo para todas palabras de un conjunto de direcciones. Su valor se obtiene de la carta ASM asociando un 1 a las salidas a activar y un 0 a las no activas. En el ejemplo:

   a) $D_2D_1D_0$ Z1 Z2 Z3 = 001101 = $0D, ya que en ese camino (bloque 000 y x=0) el próximo estado es Q = 001, y se activan Z1 y Z3 pero no Z2 (Z1 = 1, Z2=0 y Z3 = 1).

1. MSB (Most Significant Bit) y LSB (Least Significant Bit) son el bit más y menos siginificativo, respectivamente.

b) $D_2D_1D_0$ Z1 Z2 Z3 = 100111 = \$27, ya que, análogamente para x=1, el próximo estado es Q = 100, y ahora se activan las tres salidas (Z1 = Z2 = Z3 = 1)

Así se procede con el resto de los bloques, quedando el controlador mostrado en la fff37. Se puede ver que este diseño está poco optimizado, ya que hay mucha información duplicada correspondiente a combinaciones de entrada que no son significativas.

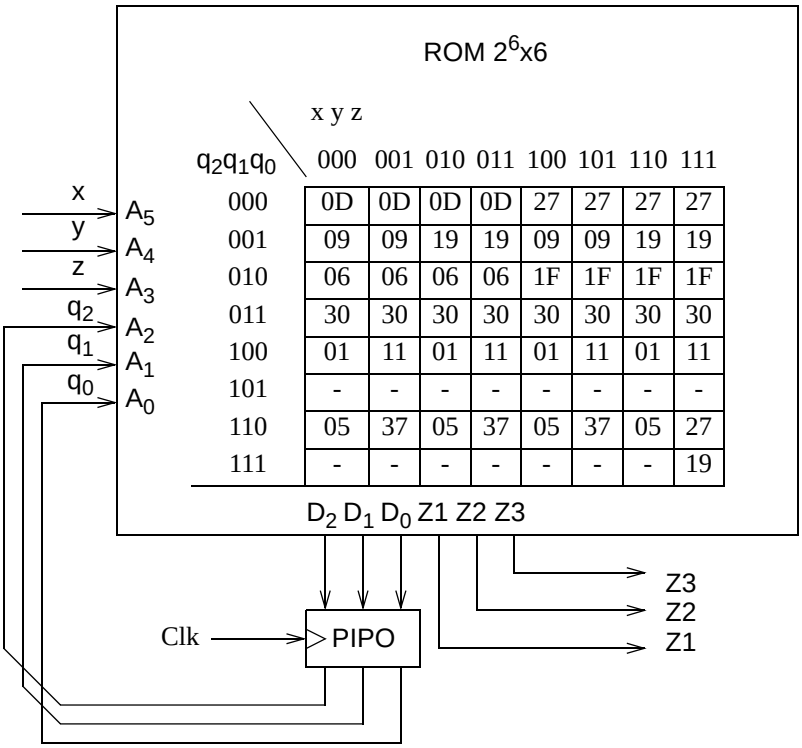


Figura 2.37: Realización del controlador con ROM y registro

En muchos controladores es posible reducir significativamente el tamaño de la ROM añadiendo circuitería combinacional adicional. Esto ocurre, en particular, cuando todos los bloques de la carta ASM contienen únicamente una variable de decisión, condición que se cumplen tanto en nuestro ejemplo de la calculadora (Fig. 2.2) como el de este epígrafe (Fig. 2.36). En estos casos, las salidas de la ROM en cada micropoeración sólo dependen de k+1 variables: las k variables de estado y la variable de decisión que interviene en ese bloque.

Existen dos estructuras generales para aprovechar la dependencia en k+1 variables de estos controladores:

1. La ROM se direcciona con k+1 líneas: las k variables de estado presente y la variable de decisión E de ese bloque, si la tiene. La variable E se selecciona multiplexando el conjunto de variables de entrada al controlador mediante el valor de estado presente, lo que requiere un multiplexor MUX 2^k:1. Por otra parte, el contenido de la palabra es, como en el caso anterior, de (k+m) bits: el valor de próximo estado (k bits) y el del comando de salida (m bits), obtenién-

dose directamente de la carta ASM.

La unidad de control para la carta ASM de la Fig. 2.36 se muestra en la Fig. 2.38, con el MUX 8:1 que necesita en este caso. La ROM resultante tiene una capacidad de 2^4x6 bits, lo que reduce a la cuarta parte el tamaño de la anterior (Fig. 2.37).

En general, los componentes requeridos son:

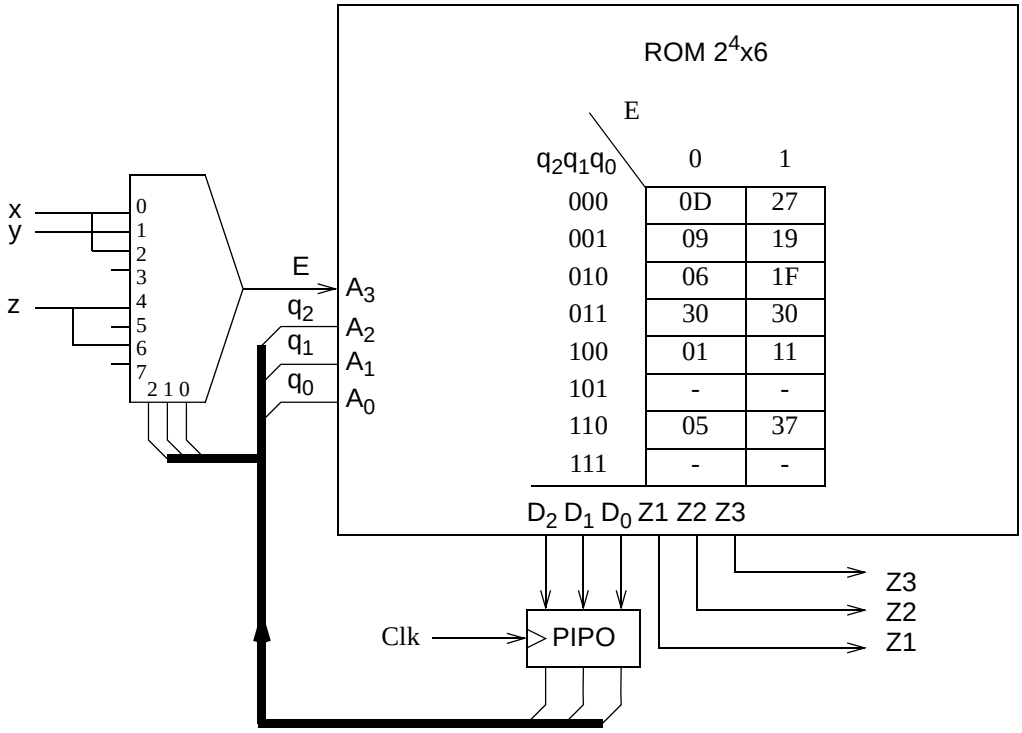


Figura 2.38: Realización del controlador con MUX, ROM y registro

PIPO[k]	ROM 2^{k+1} x (k + m)	MUX 2^k:1

2. La ROM se direcciona únicamente con las k líneas de estado presente, lo que reduce a la mitad el número de palabras de la ROM respecto a la de la solución anterior, aunque eso es a costa de aumentar el número de bits por palabra. El contenido de las palabras tienen 5 campos (ver Fig. 2.39 para la carta ASM de la Fig. 2.36):
   - uno de n bits correspondiente a las entradas de decisión, en el que se señala (con 1) la entrada de decisión que interviene en el bloque correspondiente a ese estado y con cero a las restantes entradas. Por ejemplo, para el bloque 000, como la variable de decisión es x, este campo es xyz = 100;
   - otros dos campos son de k bits, donde se escriben los dos próximos estados según valga 0 ó 1 la variable de decisión. Por ejemplo, para el bloque 000 se escribirá $D_2D_1D_0 = Q_2Q_1Q_0$= = 001 en el campo de x = 0 y $D_2D_1D_0 = Q_2Q_1Q_0$= 100 en el de x = 1;
   - y otros dos campos de m bits, donde se escriben los dos posibles comandos de forma similar a antes. Por ejemplo, para el bloque 000 se escribirá Z1 Z2 Z3 = 101 en x = 0 y Z1 Z2 Z3 = 111 en x = 1.

   Como se observa, para el ejemplo, la ROM es de 2^3 x 15.
   - Además, se necesitarán k multiplexores 2:1 para seleccionar el valor correcto de próximo

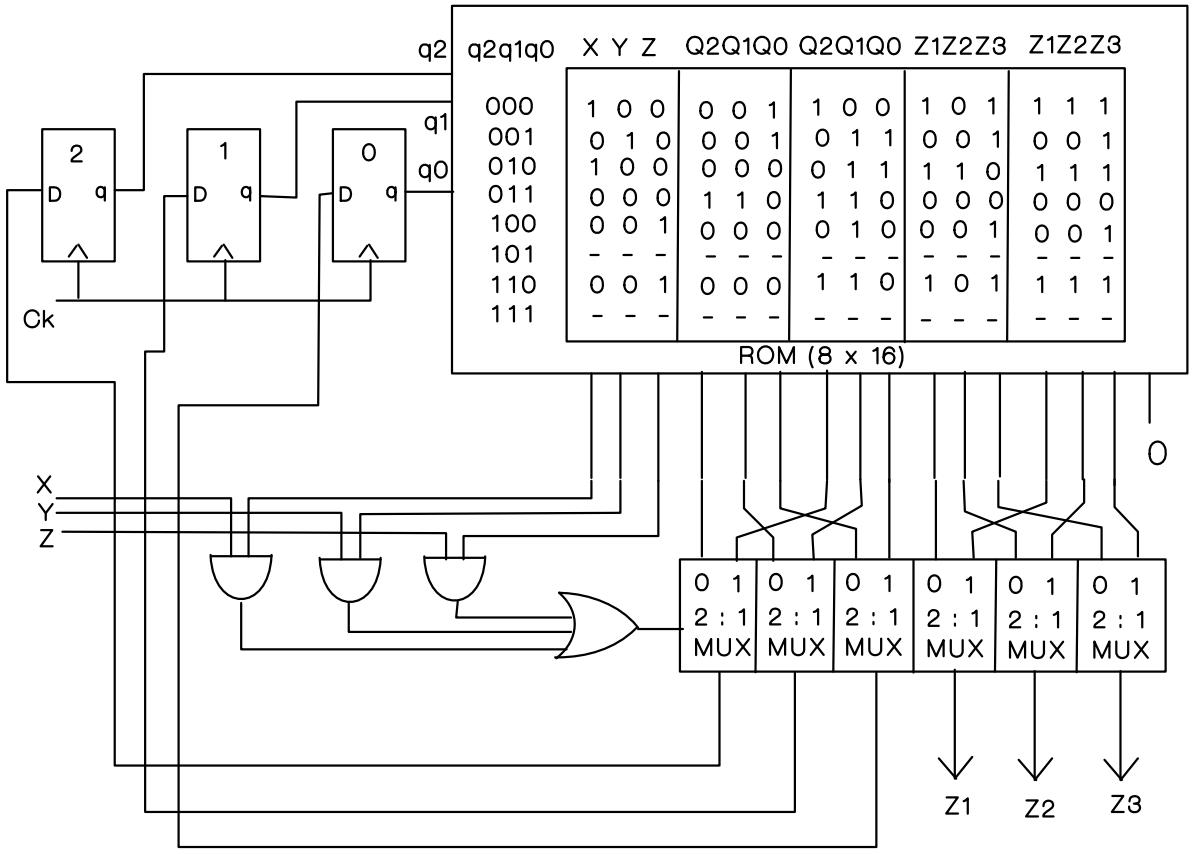


Figura 2.39: Realización del controlador con (k+m)xMUX 2:1, ROM y registro

estado y otros m multiplexores 2:1 para hacer lo propio con las salidas de comando. Un conjunto de puertas generarán la señal de selección de los multiplexores como función de xyz y de las salidas ROM para el campo de xyz.
En general, los componentes requeridos son:

PIPO[k] ROM 2^k x (n + 2k + 2m) (k+m)xMUX 2^2:1, y (n+1) puertas

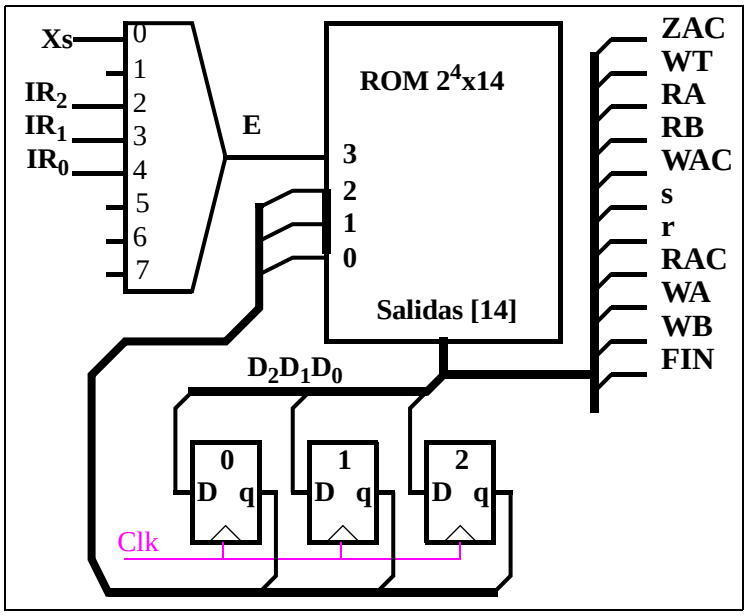
Ambas soluciones permiten un ahorro significativo en hardware con respecto a la solución general. En efecto, en los controladores suele haber muchas entradas (n es alto) y, como ambas soluciones reducen el número de líneas de dirección (en n-1 la solución 1, y en n la solución 2), las ROMs disminuyen su capacidad drásticamente (recuérdese que cada línea de dirección reduce a la mitad el tamaño de una ROM). Ello compensa suficientemente el coste de los multiplexores y puertas adicionales. Aunque parecería que la solución 2 proporciona una solución menor que la de 1 al tener la mitad de palabras que ésta, el coste de la ROM de la solución 2 es siempre mayor que la de la solución 1. En efecto la dimensión de la ROM en la solución 2 puede desarrollarse como:

$$2^k \times (n + 2k + 2m) = 2^k \times n + 2^k \times 2 \times (k+m) = 2^k \times n + 2^{k+1} \times (k+m)$$

Como se ve, comparando esta expresión con la de la capacidad de la ROM de la solución 1, que es $2^{k+1} \times (k+m)$, la solución 2 siempre necesita 2^k x n bits más.

Unidad de control de la calculadora con ROM y registro

Como siempre vamos a considerar ahora la realización mediante ROM del control de la calculadora. En este caso tenemos un cualificador por estado, por lo cual el diseño consistirá en usar la estructura de la Fig. 2.38, con registro, ROM y un multiplexor. Considerando el asignamiento habitual: S_0: 000, S_1: 001, S_2: 010, S_3: 011, S_4: 100, y S_F: 101, Ss obtiene el circuito de la Fig. 2.40, que se ha realizado directamente de la carta ASM de la Fig. 2.2.



$A	$H3H2H1H0
0	0000
1	1700
2	1AE0
3	2060
4	280C
5	0001
6	(0000)
7	(0000)
8	0800
9	1700
A	1AD0
B	2050
C	280A
D	0001
E	(0000)
F	(0000)

Figura 2.40: Realización mediante MUX, ROM y registro de la unidad de control de la calculadora

Cada dirección de la ROM viene dada por las tres variables de estado y el cualificador correspondiente a ese estado, que se selecciona adecuadamente mediante el MUX 8:1, de la forma:

$$A_3A_2A_1A_0 = E\ q_2q_1q_0$$

siendo E = Xs, o IR_2, o IR_1, o IR_0,según el caso

En cada una de esas direcciones se almacenan palabras de 14 bits de la siguiente forma: los tres primeros bits corresponden a valores de próximo estado ($Q_2Q_1Q_0 = D_2D_1D_0$) y los 11 siguientes corresponden a los valores de los comandos (salidas del controlador), concretamente, en el orden:

ZAC, WT, RA, RB, WAC, s, r, RAC, WA, WB y FIN

En la tabla de contenidos de la Fig. 2.40 los valores se han representado en hexadecimal (4 dígitos) correspondientes al orden dado, esto es, a las agrupaciones:

{(00)D_2D_1} {D_0 ZAC WT RA} {RB WAC s r} {RAC WA WB FIN}

Los valores de los comandos y próximos estados proceden de la carta ASM (Fig. 2.2) y, como ya se ha indicado, dependen del estado presente y del cualificador. Por ejemplo, para el estado S_0 (000) el multiplexor hace que E=Xs. En el caso de que Xs no esté activa (Xs=E=0), la dirección es $A_3A_2A_1A_0$ = 0000; el contenido de esta palabra es también 0 debido a que el próximo estado vuelve a ser S_0 (000) y a que no se activa ninguna salida. Sin embargo, para el mismo estado inicial S_0 (000, también E=Xs), si ahora Xs está activa, la dirección es = $8. Por otro lado, en ese camino de la microoperación, el próximo estado es S_1 (por tanto, $D_2D_1D_0$ = 001) y como, de nuevo, no se activa ninguna salida, ningún bit más de esa palabra se activa. Por tanto, el contenido de la palabra $8 es $0800.

Un ejemplo más, en la microoperación S_1 (001) siempre se pasa al estado S_2 (010) y se activa ZAC, WT y RA. Como no depende de entradas, E puede ser 0 o 1 y, como el estado presente es 001, da lugar a dos direcciones, la $1 y la $9. En ambas el contenido (próximo estado y salidas) será el mismo: 010 11100000000000 = $1700.

Comentarios:

El diseño con ROM es también directamente realizable de la carta ASM. Tiene el inconveniente de que, en general, no se aprovecha bien el espacio de direcciones de la ROM. Para cartas ASM con sólo una variable de decisión en cada bloque se puede hacer un aprovechamiento mejor de la ROM incluyendo un multiplexor. En todo caso el asignamiento de estados no influye en el coste.

La elección entre hacer un diseño basado en PLA o en ROM depende en gran medida de las funciones que intervienen, aunque hay algunas consideraciones que pueden afectar a esta decisión. En general, para una tecnología determinada, los PLAs son más rápidos que las ROMs pero poseen un consumo mayor. Por otro lado, es más fácil acceder a ROMs reprogramables (EPROMs) que a PLAs reprogramables [1]. La elección de uno u otro dispositivo dependerá de cuál de esas características es prioritaria en un desarrollo concreto.

2.6.2.4 Introducción al Control Microprogramado

Los diseños mediante ROM y PLA descritos en los apartados anteriores son ejemplos simples de control microprogramado. En este tipo de unidad de control, la estructura hardware es única para unos valores dados de entrada, variables de estado y salidas (n,k y m fijos). Dentro de estos límites, lo que caracteriza a un controlador en concreto es el programa almacenado en ROM (PLA). A este programa se le denomina microprograma o *firmware*. El cambio de microprograma altera completamente la funcionalidad del sistema digital, aunque se mantenga sin cambios el hardware de control.

El control microprogramado es muy utilizado en sistemas digitales complejos o que den lugar a distintas versiones del producto inicial. Ello ocurre, por ejemplo, con los microprocesadores. La razón de su uso es precisamente la facilidad con la que se cambia la funcionalidad del sistema, modificando sólo el firmware con lo que el resto de la realización es reutilizable.

Teniendo en cuenta, por otra parte, la necesidad de describir la función del controlador de forma fácilmente computable para usar así herramientas CAD/CAEE , el control microprogramado está enfocado hacia el uso de lenguajes de descripción de hardware (HDL). Esta herramienta de descripción es más fácil de interpretar por el computador que las cartas ASM, ya que es en todos los sentidos un lenguaje de programación. Nuestro propósito en este epígrafe es introducir algunos aspectos básicos del diseño de controladores microprogramados a partir de HDL.

1. Estas características no pueden tomarse en términos absolutos ya que dependen del estado tecnológico. Así, cada vez hay más PLAs reprogramables y las ROMs son más rápidas.

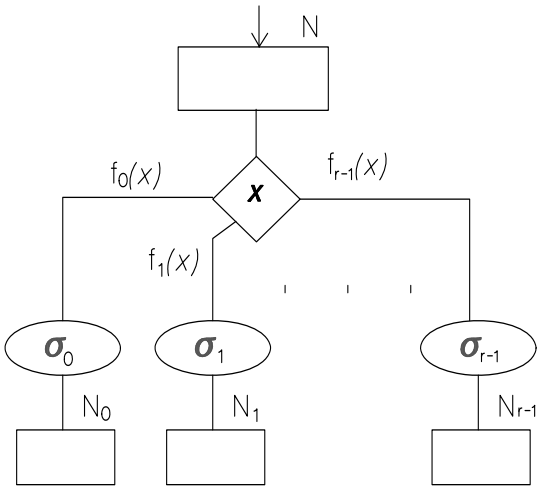


Figura 2.41: Bloque ASM genérico

En primer lugar, volvamos a considerar la conversión de una descripción ASM a su equivalente HDL en el lenguaje básico utilizado. La Fig. 2.41 muestra un bloque ASM general (estado N), cuyas decisiones se toman sobre un conjunto de funciones booleanas de las entradas ($f_0(x)$,...,$f_{r-1}(x)$) y que, en cada caso, deberá suministrar el correspondiente comando (σ_0,..., σ_{r-1}) y evolucionar al bloque siguiente (N_0,..., N_{r-1}). Este bloque es traducido al formato general de instrucción siguiente

N $f_0(x)$ σ_0 N_0

$f_1(x)$ σ_1 N_1

...

$f_{r-1}(x)$ σ_{r-1} N_{r-1}

donde:

- N representa la microinstrucción actual, normalmente mediante una palabra de dirección.
- $f_i(x)$ son funciones booleanas dependientes de las variables de condición o cualificadores *x*.
- σ_i son comandos.
- N_i son etiquetas de microinstrucción, normalmente dirección de una posible próxima microinstrucción.

La ejecución de la instrucción N es la siguiente:

- En una primera fase, llamada de evaluación, las funciones f_i son evaluadas en el punto *x*.
- En una segunda fase, llamada de ejecución, se ejecutan los comandos de acuerdo con el resultado de la fase de evaluación. Así en el caso que f_i sea cierto se ejecuta el comando σ_i y la siguiente instrucción a realizar será N_i.

La arquitectura general que se deriva de la estructura de la instrucción anterior se muestra en la

Fig. 2.42. Esta arquitectura se basa en dividir las palabras de la memoria de control en campos. Básicamente se distinguen dos campos: uno reservado a la función de transición (próxima dirección), y el otro reservado a la función de salida (comandos). La función de transición se almacena en el registro RDI (Registro de Dirección de Instrucción) de manera que la próxima dirección se obtiene a partir del contenido de RDI y el valor de los cualificadores *x*.

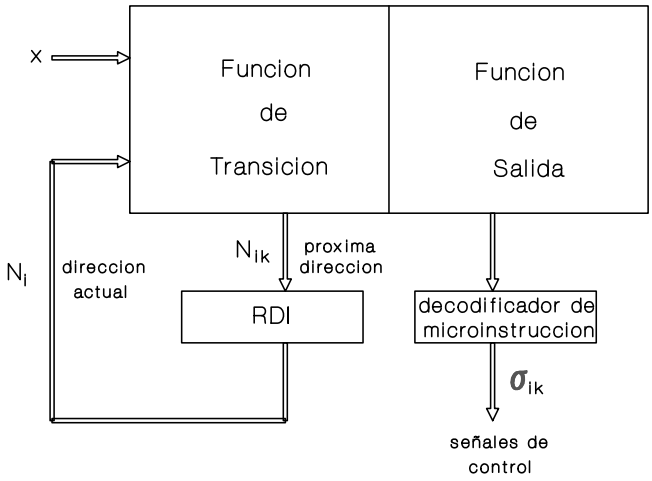


Figura 2.42: Estructura de una Unidad de Control Microprogramada

Obsérvese que la Fig. 2.42 coincide con la realización general de controlador estudiado en el apartado anterior, salvo el bloque "decodificador de microinstrucción". Este bloque se añade en algunas ocasiones cuando el número de comandos de salida puede codificarse con menos bits que el número de señales de salida a suministrar, con lo cual se reduce el tamaño de la ROM (menos bits por palabra).

A partir de la organización básica de la Fig. 2.42 se pueden construir diferentes tipos de controladores microprogramados, según los criterios de diseño que se utilicen: alta velocidad, bajo coste, descomposición en subsistemas funcionales, etc. Así, en particular, las dos soluciones particulares presentadas en el apartado anterior son otros tantos tipos de controlador microprogramado , ya que se puede demostrar que cualquier programa de control puede ser convertido a otro equivalente en el que la decisión sobre cada microinstrucción afecte a sólo una variable de entrada.

Las unidades de control microprogramadas poseen la estructura general que se muestra en la fff43. El estado de control de la máquina está identificado por la dirección actual de la memoria microprogramada (contiene el microprograma), el contenido de esta dirección es la microinstrucción que suministra la información necesaria para establecer los valores de las señales de control y elegir la siguiente dirección. El conjunto de tareas que se ejecutan en una microinstrucción son completadas en una unidad de tiempo (ciclo de reloj). La dirección donde reside la siguiente microinstrucción se genera en el secuenciador del microprograma (define el estado de control siguiente) en base a la información de la siguiente dirección que reciba de su estado de control actual, de las entradas actuales del código de operación de la macroinstrucción y/o de las entradas de condición que reciba del resto del sistema.

Varios conjuntos de microinstrucciones (microrrutinas) son necesarias para el desarrollo de sistemas complejos. El código de operación de una macroinstrucción, cuando es decodificado ordenadamente en la sección de control, señala la microrrutina apropiada incluida en la memoria microprogramada. La ejecución de la microrrutina por la unidad de control conducirá a la realización de

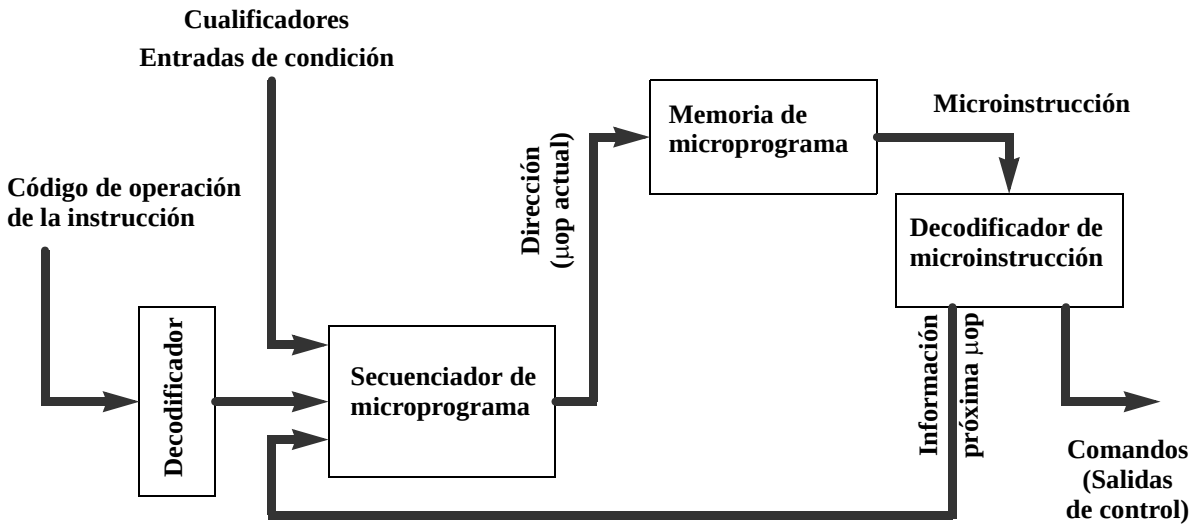


Figura 2.43: Unidad de control microprogramada

todas las operaciones elementales (microoperaciones) en la unidad de procesado.

El criterio de coste básico que se tiene en cuenta en la realización del control microprogramado es la modularidad. La modularidad desde el punto de vista del programa de control supone reducir el número de instrucciones distintas en lo posible. Ello nos lleva a realizar programas basados en un conjunto mínimo de instrucciones distintas. Estas instrucciones básicas pertenecen a los distintos tipos de "lenguajes" que dan lugar a diferentes programas (condicionales, condicionales en sentido estricto, incrementales) que proporcionan distintas realizaciones de la unidad de control[1].

Un aspecto importante es la estructura del secuenciador del microprograma, puesto que es éste el que determina las características generales de la sección de control y los mecanismos de secuenciamiento disponibles empleados para la introducción de la modularidad en los microprogramas. La mayoría de los microprogramas pasan de una instrucción a la siguiente numéricamente hablando, por lo que, básicamente, el secuenciador es un contador. Sin embargo, para asegurar la modularidad y diferentes flujos del microprograma, la unidad de control deberá permitir la realización de estructuras básicas de control, tales como facilitar la ejecución secuencial y no secuencial (capacidad de salto) de microinstrucciones, por lo que el secuenciador es algo más que un contador.

El criterio de coste de la modularidad va a repercutir sobre el tamaño de la memoria de microcontrol y el tiempo de ejecución de microprogramas. Cualquier tarea de diseño se enfrentará a un compromiso tiempo-tamaño-coste.

1. Para una mayor profundización se da bibliografía referente al control microprogramado.

2.7 RESUMEN

En este capítulo se ha presentado el diseño de la unidad de control de los sistemas digitales diseñados a nivel RT con Unidades de Datos (*Data Path*) y de Control. El punto de partida es la descrpción formal del microprograma de control, bien mediante su carta ASM, bien mediante su equivalente HD, que es donde se identifica la información del circuito a diseñar, sus entradas, sus salidas y sus estados.

Se han planteado las dos grandes líneas de diseño (cableadas y microprogramadas) y los criterios de diseño asociados a ellas. A continuación se han descrito los distintos tipos de realizaciones. La primera corresponde a una realización discreta cuya metodología corresponde al diseño estándar de circuitos secuenciales síncronos con biestables y puertas. La siguiente está basada en un registro de desplazamiento y se detalla su aproximación formal desde la carta ASM, es decir, su realización directa a partir de la misma. Este diseño presenta especificaciones estrictas en la señal de inicialización, por lo que se han descrito diferentes soluciones para generar dicha señal a través de diferentes mecanismos de incialización. Por último se han considerado otros tipos de realizaciones: con MUX y flip-flops D, con PLA y con ROM. Las dos últimas constituyen una pauta para introducir el control microprogramado, describiendo algunos conceptos y generalidades sobre el mismo.

CAPÍTULO 3: Diseño a nivel RT de un computador sencillo

3.1 INTRODUCCIÓN

En los dos temas anteriores se desarrolló un sistema digital que permitía obtener cualquier operación de suma o resta de los contenidos de dos registros A y B, almacenando el resultado de la misma en cualquiera de los dos registros señalados. La estructura del sistema se puede ver en la Fig. 3.1 en donde se aprecian claramente los componentes de la unidad de datos. Este sistema digital permite ejecutar las ocho operaciones diferentes siguientes:

IR_0 \ $IR_{2,1}$	00	01	10	11
0	A ← A+B	A ← A-B	A ← -A+B	A ← -A-B
1	B ← A+B	B ← A-B	B ← -A+B	B ← -A-B

Recordemos que su funcionamiento consiste en interpretar el código asociado a la operación requerida, código que se encuentra en el registro IR. De esa manera la unidad de control comienza a generar una serie de señales (las de selección de operación de los dispositivos de la unidad de datos), ordenadas en el tiempo, de manera que se lleva a cabo la operación elegida. Sin embargo, para realizar varias de estas operaciones seguidas, el mismo operador del sistema, el usuario, es el encargado de introducir los datos a manejar en los registros A y B, de introducir en el registro IR el código correspondiente a cada operación a realizar, e indicar, para cada operación, la señal de comienzo de la ejecución XS.

El modo de operación de este sistema es tipo "calculadora". Dicho modo se caracteriza porque puede ejecutar automáticamente una instrucción, pero sólo una cada vez, requiriéndose la intervención del usuario para preparar manualmente la instrucción (operandos y tipo de operación) y dar la señal de comienzo. No puede, en consecuencia, ejecutar automáticamente un programa (secuencia ordenada de instrucciones cuya ejecución global resuelve el problema deseado).

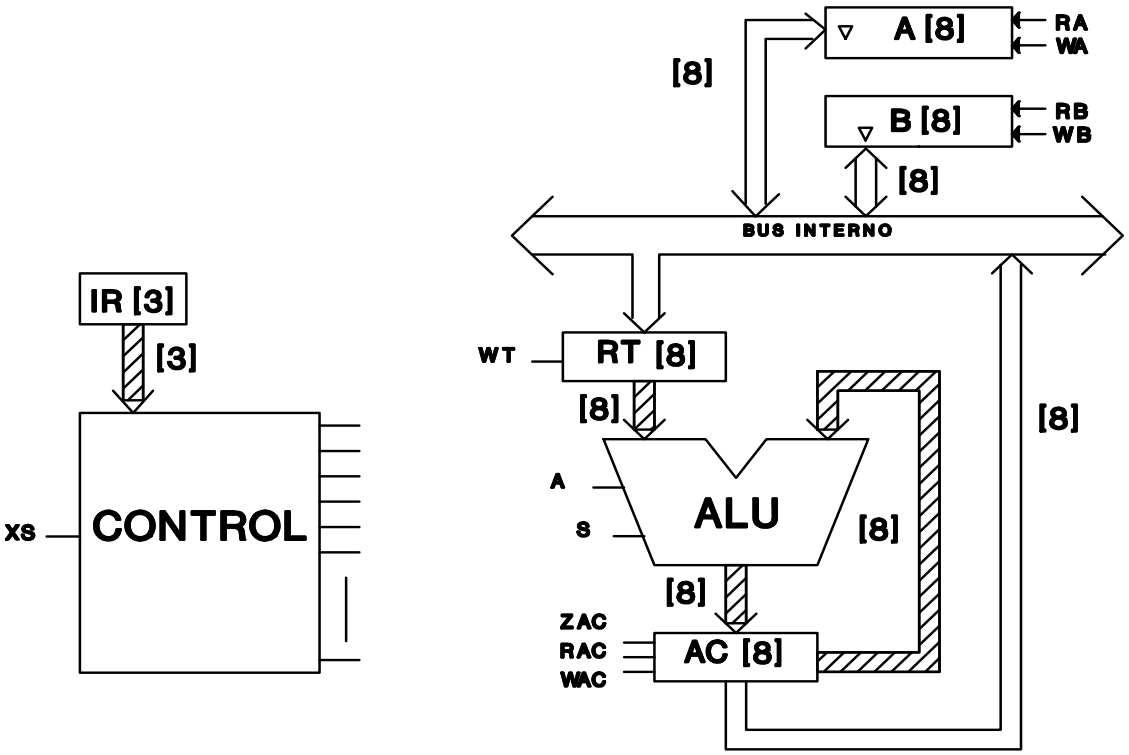


Figura 3.1: “Calculadora” de sumas y restas (desarrollada en las secciones anteriores)

Este tema tiene como propósito primordial desarrollar sistemas digitales cuyo modo de operación sea tipo "computador". Las principales innovaciones que hay que introducir sobre la calculadora son:

- Los nuevos sistemas deberán estar facultados para ejecutar automáticamente un programa. Esto significa, por una parte, que hay que dotar al sistema de la capacidad de tener un programa almacenado. Así, el sistema podrá buscar en memoria las instrucciones que debe ir ejecutando sin que el usuario tenga que irlas introduciendo una a una. Por otra parte, deberá automatizar la ejecución, lo cual significa no tener que activar la señal de comienzo XS con cada instrucción, sino sólo al principio, al ordenar la ejecución del programa.
- Hay que incrementar las prestaciones del sistema lo que lleva consigo, entre otras posibles mejoras: aumentar el número de datos que se puedan manejar; diseñar el procesador con un conjunto de instrucciones (ISP, *Instruction Set Processor*) suficientemente amplio en tipos de operación y suficientemente versátil en la forma de acceder a los operandos; e incluir la comunicación con el mundo "exterior" posibilitando el diálogo con los periféricos de entrada/salida.

Desde el punto de vista organizativo el tema se desarrolla con el fin de evolucionar poco a poco desde la bien conocida calculadora hasta la presentación, a modo de introducción, del concepto de computador. En particular, el tema está estructurado de la siguiente forma:

- En la próxima sección (Apartado 3.2) se diseña el llamado computador simple 1. El objetivo principal es solucionar la ejecución automática de programas. Para ello se presentan conceptos muy importantes (la estructura de la instrucción como código máquina, los ciclos de opera-

ción, etc.) y sus implicaciones sobre el hardware del sistema digital, incluyendo el diseño completo a nivel RT, tanto de la Unidad de Datos como de la Unidad de Control. Pese al extremadamente simple ISP (sólo 4 instrucciones), también se considera el uso de este computador desde el propio nivel ISP realizando un pequeño programa.

- En el Apartado 3.3 se diseña un computador algo más complejo, el computador simple 2. El principal objetivo es dotarlo de un ISP que sea representativo del conjunto de instrucciones existentes en los procesadores comerciales reales. Para ello se proponen instrucciones aritméticas, lógicas, de "estado", de salto, etc., así como se incluyen modos de direccionamiento directo e indirecto para algunas de ellas. Además del desarrollo hardware, se presentan varios ejemplos de su uso a nivel ISP.
- Por último, en el Apartado 3.4 se terminan de introducir las principales prestaciones de los procesadores reales, estableciendo el concepto de (micro)computador. Básicamente, en este apartado se presentan las instrucciones cuyo código completo reside en más de una palabra, los circuitos de interfaz para las operaciones de entrada-salida y los mecanismos para la interrupción de un proceso en curso y la ejecución del proceso que atiende al recurso que ha solicitado la interrupción.

3.2 COMPUTADOR SIMPLE 1 (CS1)

En este apartado se aborda el diseño a nivel RT de un sistema digital simple cuyas principales características son la ejecución automática del programa y el manejo de un número alto de datos. De esta forma, el sistema va a operar según el modo de "computador", por lo que lo denominaremos Computador Simple 1 (CS1).

Aunque CS1 no es un sistema real, comparte ya muchos aspectos con los computadores reales. Así, por una parte, en el epígrafe 2.1 se le dotará de una memoria para el almacenamiento del programa (instrucciones) y de los datos (operandos); conceptualmente, esta memoria corresponde a la "memoria principal", unidad funcional básica de los computadores actuales. En segundo lugar (epígrafe 2.2), CS1 incluye la ejecución lineal de las instrucciones que forman el programa almacenado; esta capacidad se deriva de la descomposición de la operación en dos ciclos, de búsqueda y de ejecución, que se encadenen uno tras otro, lo cual permite, de un lado, distinguir entre instrucción y operando en los accesos a memoria y, de otro, ejecutar sucesivamente las instrucciones del programa.

Las cuestiones tratadas en los epígrafes 2.1 y 2.2 son de carácter general. A continuación (epígrafe 2.3) se especificará a nivel ISP un conjunto de 4 instrucciones con el fin de diseñar CS1 completamente (incluyendo el controlador). Por último (epígrafe 2.4) se plantea un problema complejo, para ilustrar cómo se resuelve mediante un programa ejecutable por CS1; aquí se introducen los diferentes niveles o jerarquías de los lenguajes contrastando el código máquina (entendible por el hardware) y el uso de *mnemónicos* (dirigidos por el programador).

3.2.1 Unidad de datos

Si se parte de la estructura de la "calculadora" que se mostraba en la Fig. 3.1 y se desea aumentar el número de datos a manejar por el sistema, existen dos alternativas claramente diferenciadas:

a) Añadir tantos registros tipo A/B como se deseen.
b) Sustituir los registros A/B por memoria de tipo aleatorio (RAM/ROM).

La ventaja más importante de la alternativa primera es la mayor velocidad de acceso a los registros frente a las memorias RAM/ROM. Las principales desventajas son el mayor coste y el elevado número de líneas de control que se necesitan; así, por ejemplo para 256 palabras, se precisan 2 x 256 = 512 líneas para la primera alternativa, mientras que bastarían 10 (habilitación, lectura/escritura y 8 de direcciones) para la segunda. Debe, pues, establecerse un compromiso entre ambas estableciéndose los distintos niveles de jerarquía de memoria (ver Tema...). En los computadores reales se incluyen unos cuantos registros siguiendo la alternativa a) y se opta por la alternativa b) para almacenar cientos de miles de palabras.

En nuestro diseño de CS1 adoptaremos, en consecuencia, la inclusión de una memoria (i.e. tipo RAM) para aumentar el número de datos. Esto se representa en la Fig. 3.2, en donde el bus de datos se conecta al bus interno del sistema y en el bus de direcciones se colocaría la dirección donde, dentro de la RAM, se encuentra el dato con el cual queremos trabajar. La capacidad de la memoria, por motivos que más adelante veremos, será de sesenta y cuatro palabras de ocho bits (2^6 x 8) en nuestro CS1. En general, sería de 2^K x N, donde K es la anchura del bus de direcciones y N la del bus de datos, pudiendo incluir tanto dispositivos RAM como ROM.

Con el fin de poder almacenar el programa, hay que incluir una memoria donde residan sus instrucciones. De nuevo hay dos alternativas para esta memoria de programa:

a) Que sea distinta de la memoria de datos.

b) Que sea la misma memoria, la cual quedará así compartida por instrucciones y por operandos.

La principal característica de la solución basada en dos memorias diferentes es que permite

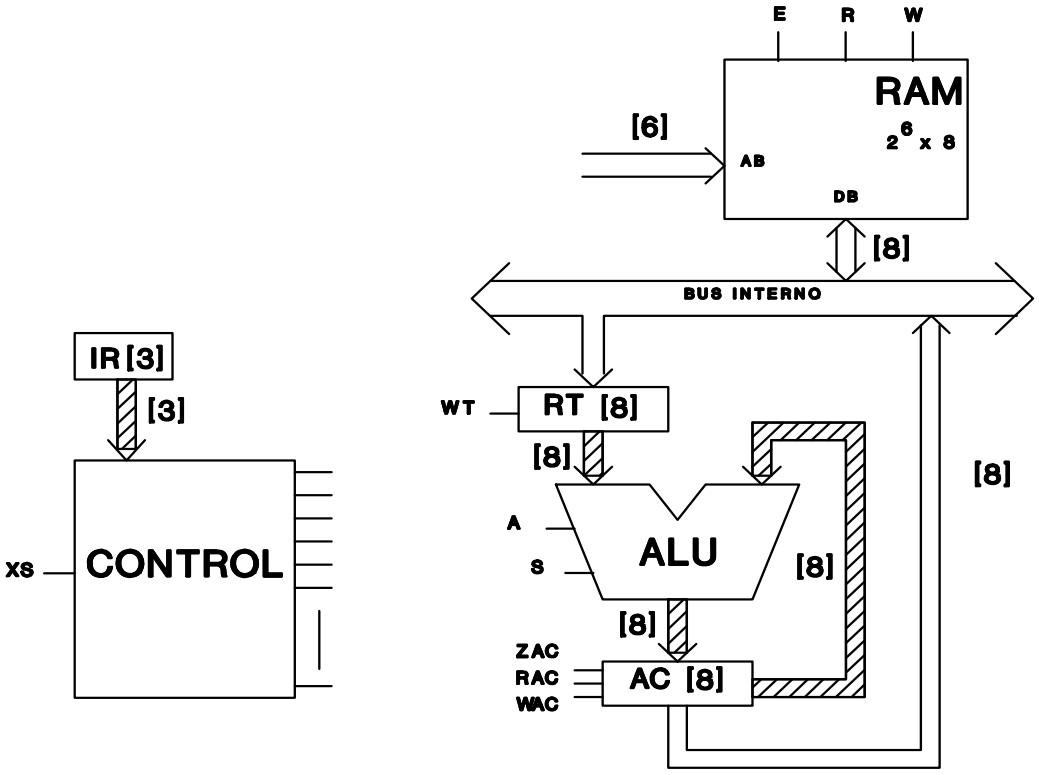


Figura 3.2: Sustitución de los registros A y B por una memoria RAM

separar claramente el diseño del procesado de datos del de las instrucciones. Por ejemplo, podría trabajarse con datos de 8 bits junto a instrucciones de 12 bits. Además, el acceso a los operandos y a las instrucciones seguirían rutas obviamente diferenciadas. A pesar de estas ventajas, disponer de dos memorias diferentes presenta fuertes inconvenientes como es la obligada duplicación de los buses de dirección y de datos, así como del trasvase de programas/datos desde las unidades de memoria periféricas.

La solución más habitual es la de compartir la memoria para programa y datos. Esta solución es la que adoptamos para CS1. El propio circuito de la Fig. 3.2 admite dicha solución.

Centremos ahora nuestra atención en el propio concepto de instrucción. En ella se suministran dos informaciones fundamentales: el tipo de operación que ha de realizarse y los datos sobre los que se realiza la operación. Desde la perspectiva del computador la instrucción es una palabra binaria, en la que cabe distinguir dos "partes" o "campos":

1. Campo del código de operación, CO. Corresponde a los bits que codifican las diferentes operaciones definidas para el computador. El número de bits o anchura de este campo, N_{CO}, debe ser suficiente para distinguir todas las operaciones.

2. Campo de la dirección (del operando), CD. En estos bits se identifican los operandos. Puesto que los datos se almacenan en una memoria, la forma de identificar un operando es suministrar la dirección de la palabra donde se encuentra dicho dato. De aquí que se denomine campo de la dirección. Su anchura será N_{CD} bits.

Vamos a especificar estas relaciones para diseñar nuestro CS1. En primer lugar, cada dato de CS1 corresponderá a una palabra de memoria (concretamente N=8); ésta es una simplificación del caso real ya que los computadores existentes permiten el manejo de datos multipalabras. En segundo lugar, el campo de direcciones suministrará todas las direcciones de memoria (concretamente, $N_{CD} = K = 6$); de nuevo se trata de una simplificación del caso real ya que los computadores existentes incluyen múltiples formas o modos de direccionamiento (ver Aptº 3.1.2), siendo el modo "directo", también llamado "exhaustivo", el que se ha elegido para el CS1. Por último, CS1 entenderá sólo instrucciones monopalabras (concretamente $N_{CO} + N_{CD} = 8$); esta elección simplifica otra vez el caso real que permite instrucciones de distintas anchuras de palabras (ver Aptº 3.4.1).

Retomando la incidencia sobre el hardware en el proceso de paso entre calculadora y computador simple, habría que colocar en el bus de direcciones de la RAM dos direcciones: 1ª la que contiene la instrucción que se pretende realizar; y 2ª la dirección donde se encuentra el dato con el cual se va a llevar a cabo la operación. También para leer la instrucción hay que conectar el bus interno del sistema con el registro IR, que a su vez, aumenta el número de bits para poder acoger ambos campos de instrucción. Centrándonos en IR, además tiene que poseer, ver Fig. 3.3, dos buses de salida, uno dirigido al controlador para indicarle a éste el código de operación que hay que realizar y el otro, con la dirección del operando, hacia el bus de direcciones de memoria.

En nuestro caso concreto, el registro tiene la forma que se observa en la Fig. 3.4, en donde se puede ver que el registro IR es de ocho bits, de los cuales, los dos más significativos indicarán la operación a realizar (campo de operación) y los seis restantes indicarán la dirección del operando (campo de dirección). El haber reservado sólo dos bits para las operaciones nos indica ya que el número de posibles operaciones será muy reducido, en este caso, sólo cuatro operaciones diferentes.

Viendo de nuevo la Fig. 3.3 parece que ya se puede conectar el bus de seis bits de salida del registro IR con el bus de direcciones de la memoria RAM. Como a este bus hay que conectar, además la dirección de la instrucción y debido también a otras consideraciones (eléctricas, de partición en cir-

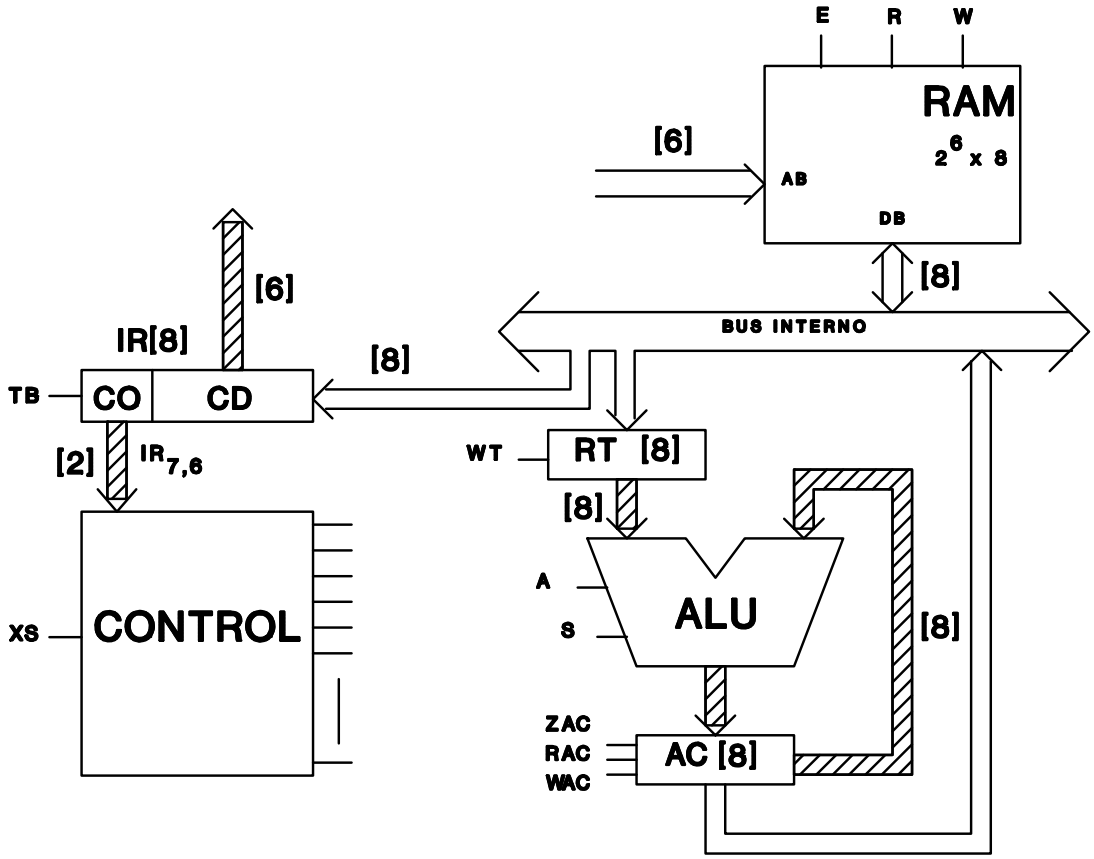


Figura 3.3: Se incorpora el registro IR, cuya estructura está dividida en campos y que se conecta al bus interno

Figura 3.4: Estructura del nuevo IR: Campo de operación y campo de operando o dirección de memoria

cuitos integrados, etc.) es conveniente añadir un registro específico que almacene las direcciones y las suministre a las memorias de manera adecuada. A ese registro , que se llama MAR (Memory Address Register), se le conecta como entrada el bus de seis líneas que procede del registro IR (Fig. 3.5) y su salida proporciona el bus de direcciones de la memoria.

Si suponemos que la instrucción a realizar ya está almacenada en el registro IR, el sistema de la Fig. 3.5 permite su ejecución para un número elevado de operandos. En efecto, una vez grabada esa información en el registro IR, sus dos bits más significativos se encargan de indicar al controlador qué operación queremos llevar a cabo, mientras los seis bits restantes se transfieren al registro MAR con la idea de buscar en la RAM el dato requerido.

Sin embargo, la Unidad de Datos de la Fig. 3.5 no permite acceder a la instrucción que hay que

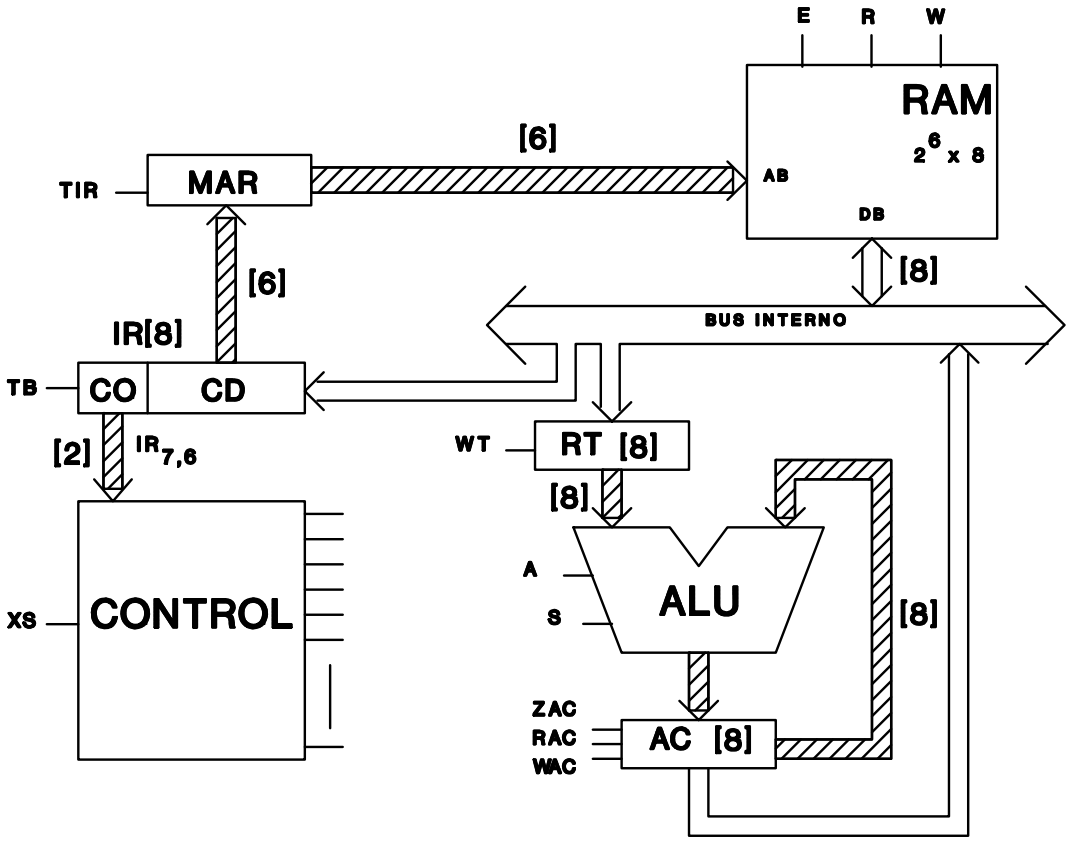


Figura 3.5: Se introduce el registro MAR (*Memory Address Register*)

ejecutar y que se encuentra almacenada en la memoria. La razón es que aún falta un dispositivo que actúe como puntero de instrucciones, esto es, que suministre la dirección de la palabra de la memoria en la que se encuentra la instrucción. De esta forma, la instrucción almacenada en la memoria puede cargarse en el registro IR a través del "bus interno". Este puntero es un registro denominado Contador de Programa (PC, Program Counter) debido a que su principal operación, como justificaremos en el próximo epígrafe, es la de contar. En tanto que suministra direcciones, el registro PC está dimensionado al bus de direcciones y sus salidas están conectadas al registro MAR. De esta forma, la Unidad de Datos del CS1 quedará tal como aparece en la Fig. 3.6**.** .

3.2.2 Ejecución automática del programa

La Unidad de Datos de la Fig. 3.6 permite el acceso a una instrucción (vía PC). Esta instrucción es una de las que, globalmente, forman el programa que suponemos almacenado en la memoria. El problema que vamos a resolver en este apartado es cómo puede conseguirse ejecutar todo el programa de forma automática.

La ejecución del programa requiere que el sistema digital busque la instrucción actual y que la ejecute. Después tendrá que buscar la siguiente instrucción, la ejecutará y así sucesivamente hasta completar todo el programa. Está claro que el funcionamiento de este sistema digital se basa en la repetición cíclica en el tiempo de dos ciclos ó fases de operación (Fig. 3.7)..

En la primera de ellas, denominada de búsqueda (Fetch), el contenido del registro PC se trans-

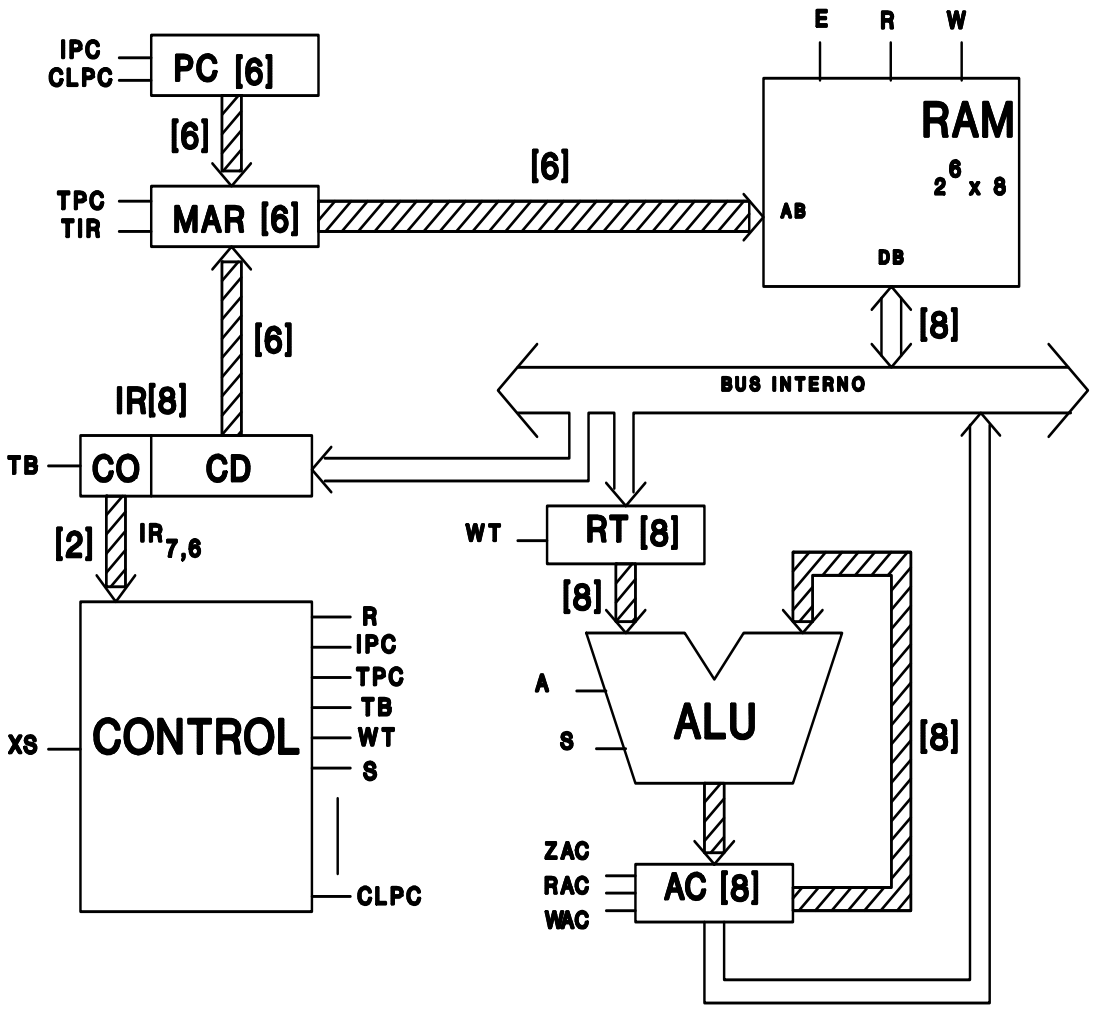


Figura 3.6: Unidad de datos del Computador Simple 1

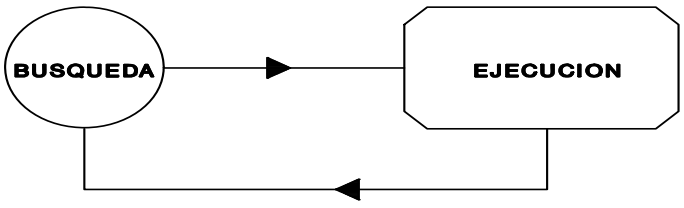


Figura 3.7: Fases de búsqueda (*Fetch*) y de ejecución (*Execution*)

fiere al registro MAR, para indicarle a la memoria del sistema en qué dirección se encuentra la próxima instrucción a realizar por el mismo. Posteriormente, la información almacenada en la memoria en esa dirección se trasvasa, mediante el bus interno del sistema, al registro IR. Con ello se finaliza la fase de búsqueda. En la segunda fase de operación, denominada de ejecución (Execute) la palabra grabada en el registro IR informa, tanto al controlador sobre qué operación es la que hay que realizar, como al registro MAR, y a través de éste a la memoria, sobre en qué dirección se encuentra el dato con el que vamos

a llevar a cabo la operación anterior. De esta forma la instrucción puede ser ejecutada.

Tras finalizar la ejecución se vuelve a realizar una nueva fase de búsqueda continuando el proceso cíclicamente (Fig. 3.7). Al conjunto formado por ambas fases se le denomina ciclo de instrucción.

Si nos fijamos de nuevo en la Fig. 3.7, vemos que el proceso no tiene fin, lo cual no tiene utilidad ninguna ya que todo programa debe tener final. Por eso hay que exigir siempre que una de las instrucciones del sistema digital sea la de parada (STOP), de manera que el controlador saque al sistema de ese ciclo sin fin cuando reciba el código correspondiente a esa operación.

En la Fig. 3.8 se representa la carta ASM (Algorithmic State Machine) del sistema para la ejecución automática del programa. La activación de la señal de comienzo XS significa ahora la orden de ejecutar el programa almacenado. Cuando ello ocurre, el contador de programa se inicia con la dirección de la primera instrucción (acción condicional que carga en PC el valor inicial deseado, PCinicial). Después comienzan los ciclos de instrucción hasta encontrar la instrucción STOP que lleva al sistema otra vez al estado inicial de espera (S_0).

Para ejecutar el conjunto de instrucciones de manera sucesiva, el contenido del registro PC debe ir variando, de modo que al comienzo de cada ciclo de búsqueda su contenido sea la dirección de la nueva instrucción a realizar. Para conseguir esto de forma automática, hay que establecer dos especificaciones: 1ª) La posición en memoria de la primera instrucción del programa, que en nuestro caso será la dirección $0; y 2ª) El tipo de flujo admisible en la ejecución de las instrucciones (o sea, si es lineal, si posee saltos,...), que en este primer diseño de computador será lineal exclusivamente. De acuerdo con

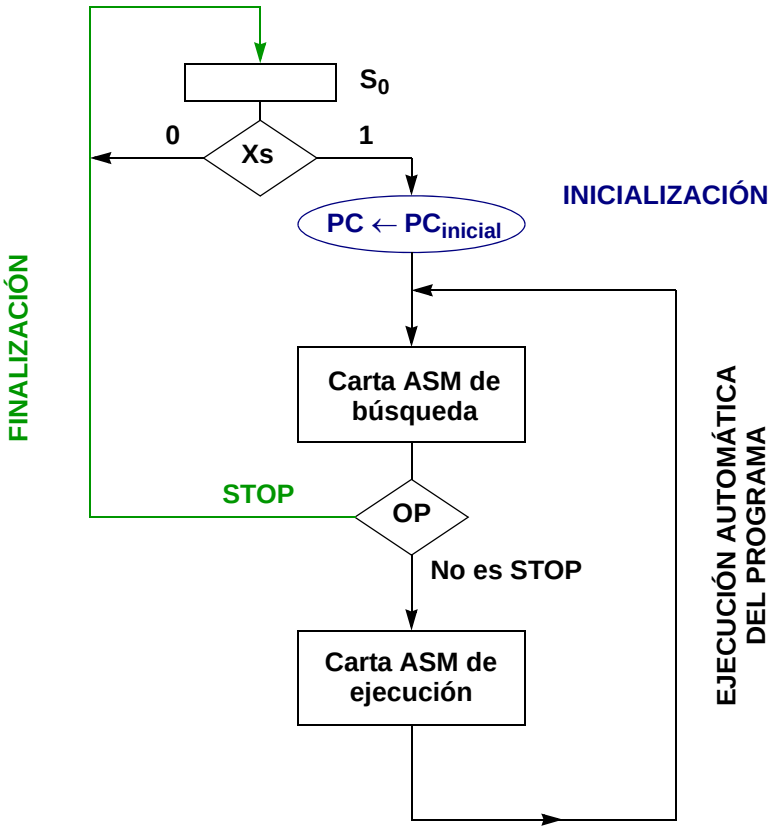


Figura 3.8: Carta ASM de la operación en modo computador: Inicio, Ejecución automática del programa y fin.

esto, las instrucciones se almacenan en la memoria de forma ordenada y consecutiva, según se muestra en la Fig. 3.9. Así, partiendo de la posición cero de la memoria, se van recorriendo una tras otra las sucesivas posiciones de la misma.

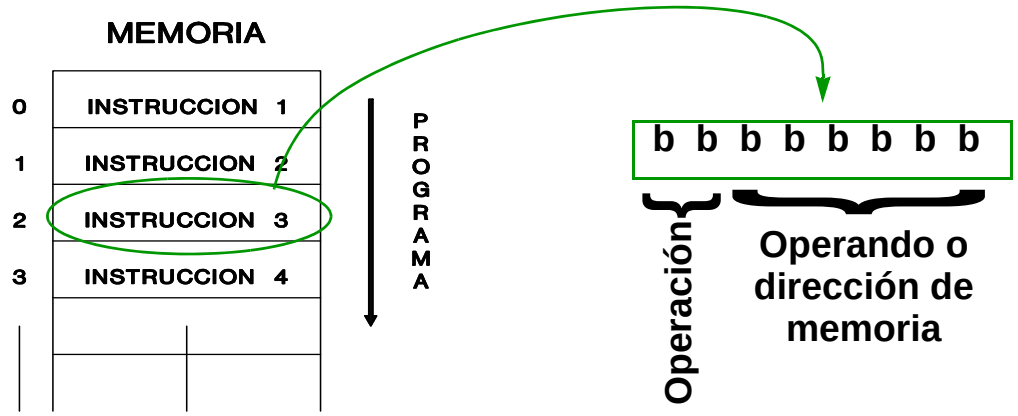


Figura 3.9: Almacenamiento del programa en la memoria RAM comenzando en la palabra 0. Dentro de cada palabra están los 0's y 1's de la instrucción, organizados en código de operación y de operando

Para ello lo único que necesitamos es dotar al registro PC de la posibilidad de inicializarse a 0 y de incrementarse, por lo que se trata de un contador ascendente con puesta a 0. Si el incremento se realiza en cada ciclo de búsqueda, al final del mismo, queda almacenada en él la dirección de memoria que apunta a la siguiente instrucción a realizar. Con ello, automáticamente el sistema queda listo para comenzar una nueva fase de búsqueda tras terminar la búsqueda actual.

3.2.3 El sistema digital CS1

De acuerdo con los apartados anteriores, el computador CS1 posee un programa almacenado linealmente desde la posición 0, con una instrucción en cada palabra de la memoria, que puede ejecutar. La última instrucción del programa será la de STOP. En este apartado ultimaremos el diseño del CS1. Para ello, en primer lugar definiremos el conjunto de instrucciones que entenderá este computador. A continuación describiremos completamente el hardware requerido a nivel RT. Por último, diseñaremos la unidad de control.

3.2.3.1 El conjunto de instrucciones

Con el fin de diseñar completamente el computador CS1 hay que especificar su conjunto de instrucciones a nivel ISP.

Para CS1, cada instrucción es una palabra binaria. La instrucción así descrita se dice que está en "código máquina". Sin embargo, el lenguaje máquina, al ser binario, es poco adecuado para su manejo por las personas. Por ello, desde la perspectiva del programador, cada instrucción se describirá por un *mnemónico* representativo de la operación. Tanto la especificación a nivel ISP de CS1 como su uso en programación la realizaremos usando *mnemónicos*.

Como el campo de código de instrucción para CS1 es de 2 bits (Fig. 3.4), este computador sólo podrá tener 4 instrucciones, que son:

a) La primera de ellas será aquélla que detiene la ejecución del programa, que ya sabíamos que debía estar. En *mnemónico* se caracteriza por **STOP** y le adjudicamos el código de operación **00**. STOP es una instrucción "de control", por lo que no tiene operando y, en consecuencia, el campo de direcciones no está especificado. Su código máquina es:

STOP: 0 0 (x x x x x x)

b) Con el **código 01**, representada en *mnemónico* por **ADD $A**, se realizará la suma del registro acumulador con el contenido de la memoria RAM en su dirección $A. El resultado se almacena en el propio acumulador. Se trata de una instrucción aritmética con sólo un operando programable, ya que el otro, el acumulador, es fijo. La dirección del operando es $A y forma parte explícita del *mnemónico*.
A nivel RT se describe como **AC ← AC + RAM ($A)**
Su código máquina es:

ADD$A: 0 1 A_5 A_4 A_3 A_2 A_1 A_0

c) Con el código **10**, representado por **SUB$A**, se obtendrá el resultado de restarle al contenido del registro acumulador, el contenido de la memoria RAM en su dirección $A. El resultado se almacena en el propio acumulador.
A nivel RT se describe como **AC ← AC - RAM ($A)**
Su código máquina es:

SUB$A: 1 0 A_5 A_4 A_3 A_2 A_1 A_0

d) Por último, se realizará la carga del contenido del registro acumulador en la palabra de dirección $A de la memoria RAM. El código utilizado será el **11**, representando esta operación por **STA$A**: ***ST****ore* ***A****ccumulator*.
La operación a nivel RT es **RAM ($A) ← AC**
Su código máquina es:

STA$A: 1 1 A_5 A_4 A_3 A_2 A_1 A_0

3.2.3.2 Requisitos hardware

En nuestro Computador Simple 1 han aparecido nuevos componentes, con respecto a la "calculadora" de partida. La Fig. 3.6 muestra el sistema digital con el que se pueden realizar todas las operaciones planteadas. La descripción formal de los dispositivos introducidos es la siguiente a nivel RT en la Fig. 3.10:

a) REGISTRO PC, de seis bits, que proporciona la dirección de memoria donde se encuentra la próxima instrucción a realizar. Para llevar a cabo su cometido se le ha dotado de dos señales de control:

CLPC, que coloca a cero el registro

IPC, que lo incrementa en una unidad.

b) REGISTRO MAR, también de seis bits, proporciona la dirección de memoria a ésta. Esta información la recibe tanto del registro IR como del registr PC. Posee dos señales de control:

TPC, transfiere el contenido del registro PC al registro MAR.

TIR, transfiere el contenido de los seis bits menos significativos del registro IR al registro MAR.

c) REGISTRO IR, de ocho bits, recibe las instrucciones procedentes de la memoria a través del bus interno del sistema. A nivel de salida está dividido en dos partes: los dos bits más significativos se envían a la unidad de control y los seis restantes se envían al registro MAR. Sólo posee una señal de control:

TB, transfiere, al registro IR, el contenido del bus interno del sistema.

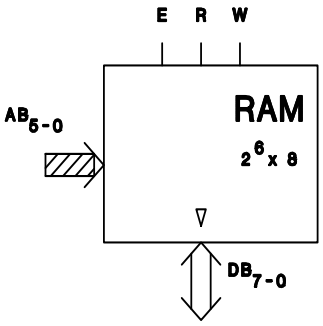
d) MEMORIA RAM, posee un bus de direcciones de seis líneas, un bus de datos, bidireccional, de ocho líneas y tres señales de control:

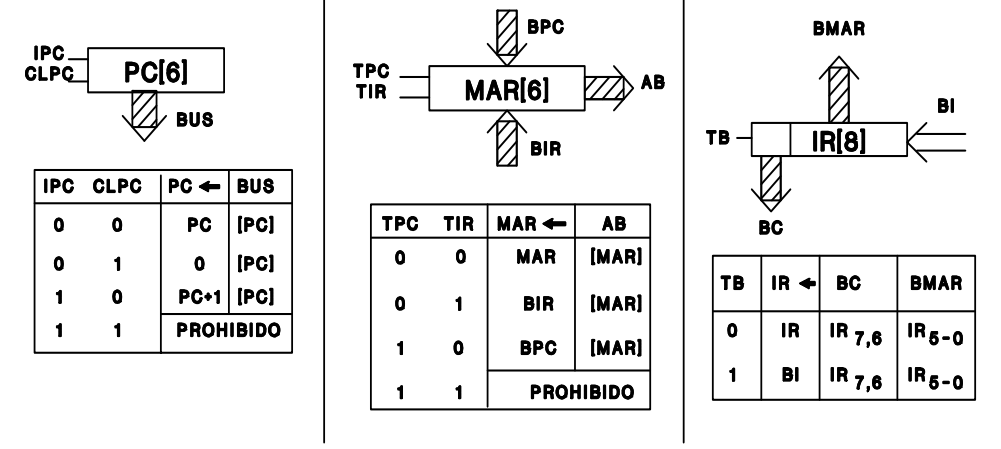
E, habilitador de la memoria.

R, vuelca el contenido de la palabra de memoria que le indica el bus de direcciones en el bus interno del sistema (operación de lectura de RAM).

W, transfiere el dato exsistente en el bus interno del sistema a la palabra de memoria que le indica el bus de direcciones (operación de escritura en RAM)..

E	R	W	RAM	DB
0	X	X	RAM ← RAM	HI
1	0	0	RAM ← RAM	HI
1	0	1	RAM(AB) ← DB	DB
1	1	0	RAM ← RAM	[RAM(AB)]
1	1	1	PROHIBIDO	





IPC	CLPC	PC ←	BUS
0	0	PC	[PC]
0	1	0	[PC]
1	0	PC+1	[PC]
1	1	PROHIBIDO	

TPC	TIR	MAR ←	AB
0	0	MAR	[MAR]
0	1	BIR	[MAR]
1	0	BPC	[MAR]
1	1	PROHIBIDO	

TB	IR ←	BC	BMAR
0	IR	$IR_{7,6}$	IR_{5-0}
1	BI	$IR_{7,6}$	IR_{5-0}

Figura 3.10: Descripción a nivel RT de los nuevos componentes en el computador simple 1

3.2.3.3 Unidad de control

Antes de diseñar la unidad de control se tendrán que obtener, para cada una de las instrucciones a realizar, las distintas microoperaciones. Después, se obtendrá la carta ASM global y, de ella, el circuito

de control.

Se ha visto antes que la ejecución de una instrucción requiere dos fases. La fase de búsqueda, que es igual para todas las instrucciones, y la fase de ejecución que varía con cada instrucción. En la Fig. 3.11, se representa el desarrollo en microoperaciones a nivel RT de estas cuatro operaciones, así como del ciclo de búsqueda. Éste está constituido por dos microoperaciones. En la primera se transfiere el contenido de PC al registro MAR, para lo que se activa la entrada TPC de éste. En la segunda, por su parte, se procede a la lectura de la RAM (accediéndose a la instrucción actual) y a su almacenamiento en el registro IR, para lo cual se activarán las señales E, R y TB; además, simultáneamente se incrementa el contador de programas, para lo que se activa IPC. De esta forma, al acabar el ciclo de búsqueda, la instrucción se encuentra almacenada en IR y PC apunta ya a la próxima instrucción.

Por su parte, las microoperaciones para ejecutar las 4 instrucciones del CS1 son:

- **STOP**:

  Simplemente, no se hace nada (NOP), pero el próximo estado del controlador será el de espera, S_0.

- **ADD$A**:

  En la primera micro-operación (numerada con 3) el campo de direcciones de IR se transfiere al registro MAR (basta activar TIR).

Ciclo de Búsqueda (*Fetch*)

S	μOp	Señales a activar
1	MAR ← PC	TPC
2	IR ← RAM, PC ← PC + 1	E, R, TB, IPC

Ciclo de Ejecución (*Execute*)

S	STOP (00)	ADD (01)	SUB (10)	STA (11)
3	NOP go to S_0	MAR ← IR (TIR)		
4		T ← RAM (E, R, WT)		RAM ← AC, goto S_1 (E, W, RAC)
5		AC ← AC + T, goto S_1 (A, WAC)	AC ← AC - T, goto S_1 (S, WAC)	

Figura 3.11: Descripción a nivel RT de las fases de búsqueda y de ejecución del computador simple 1

Después (op 4) el dato es almacenado en el registro T, sin más que leer de la RAM (E y R se activan) y escribir en paralelo en el registro T (WT se activa).

Por último, se selecciona la operación de suma en la ALU (se activa A) y el resultado se almacena en el acumulador (se activa WAC). El siguiente estado del controlador será el de la primera op del ciclo de búsqueda (1 en Fig. 3.11).

- **SUB$A**:

  Es como ADD$A salvo que se selecciona la resta en la ALU (activándose S en vez de A).

- **STA$A**:

  La primera operación (la número 3) es como antes. La segunda realiza la transferencia del contenido del acumulador en la correspondiente palabra de memoria (se activa RAC, E, y W). Después se continúa por el estado 1.

Como se ha indicado anteriormente existen microoperaciones comunes a varias operaciones, lo cual se tendrá en cuenta a la hora de realizar la carta ASM global, tanto de la unidad de datos (Fig. 3.12), como de la unidad de control (Fig. 3.13). Los estados S_1, S_2, S_3, S_4 y S_5 de estas cartas corresponden a las operaciones 1, 2, 3, 4 y 5 respectivamente.

Las señales a activar se pueden ver en la Tabla 3.1.

Tabla 3.1 Señales a activar por la Unidad de Control

Tabla 3.1.- Señales a activar por la Unidad de Control

SK	COND.	CLPC	IPC	TPC	TIR	TB	E	R	W	WT	A	S	ZAC	RAC	WAC
S0	XS = 1	1													
S1				1											
S2			1			1	1	1							
S3	$IR_{7,6}$ = 00				1										
S4	$IR_{7,6}$ = 11 = 11						1 1	1	1	1				1	
S5	$IR_{7,6}$ = 01 = 01										1	1			1 1

Con todos estos datos y utilizando la realización basada en un biestable por estado, implementamos la unidad de control del Computador Simple 1, ver Fig. 3.14. Para ello hemos hecho uso de la aproximación formal, que nos permite obtener el diseño del controlador directamente de la carta ASM correspondiente, como se mostró en el capítulo anterior.

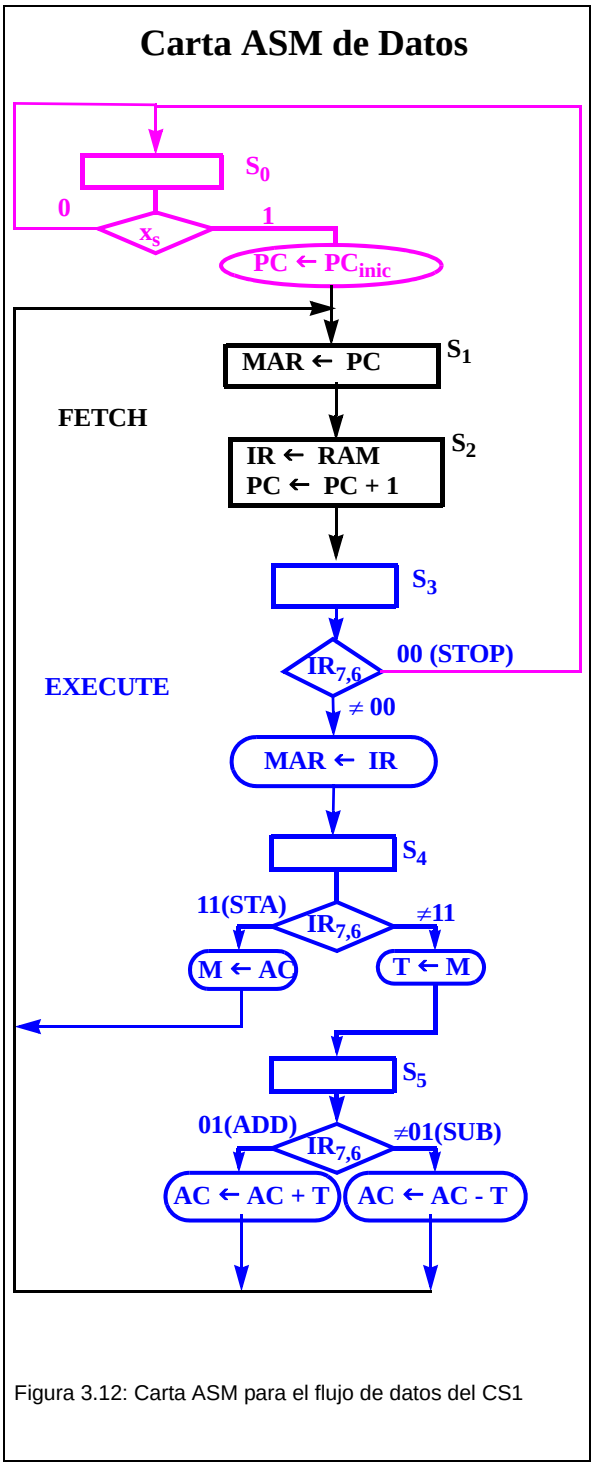


Figura 3.12: Carta ASM para el flujo de datos del CS1

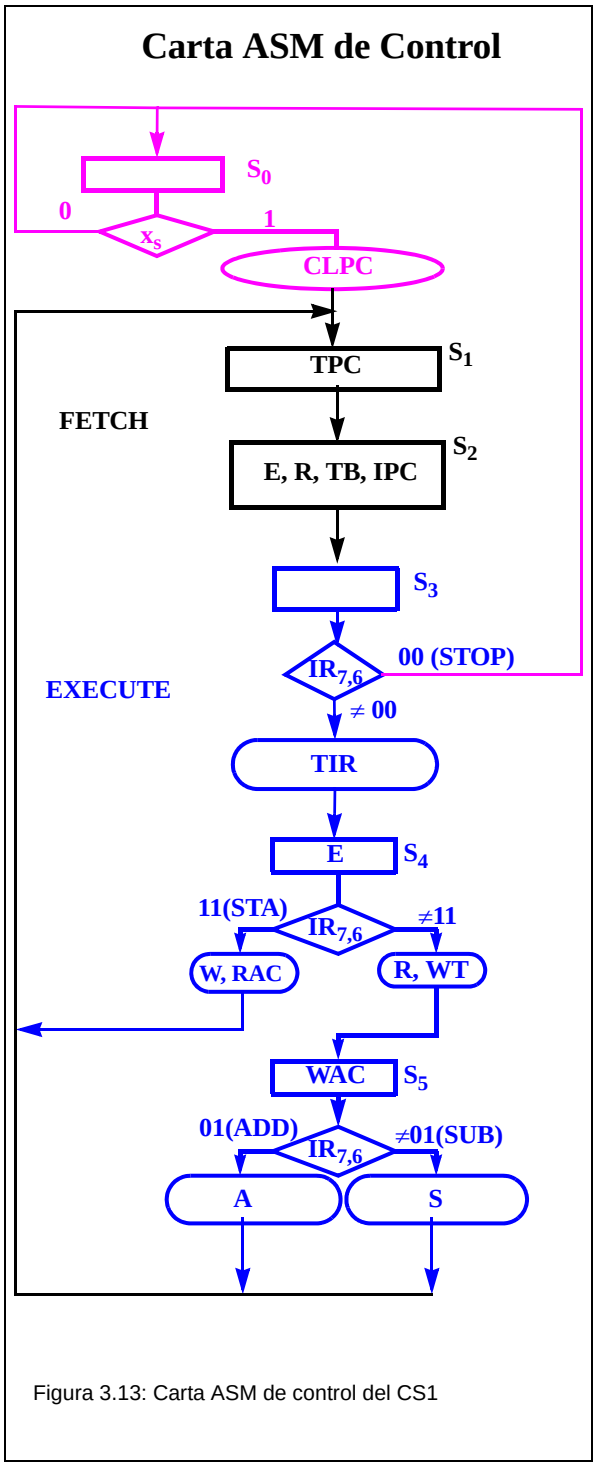


Figura 3.13: Carta ASM de control del CS1

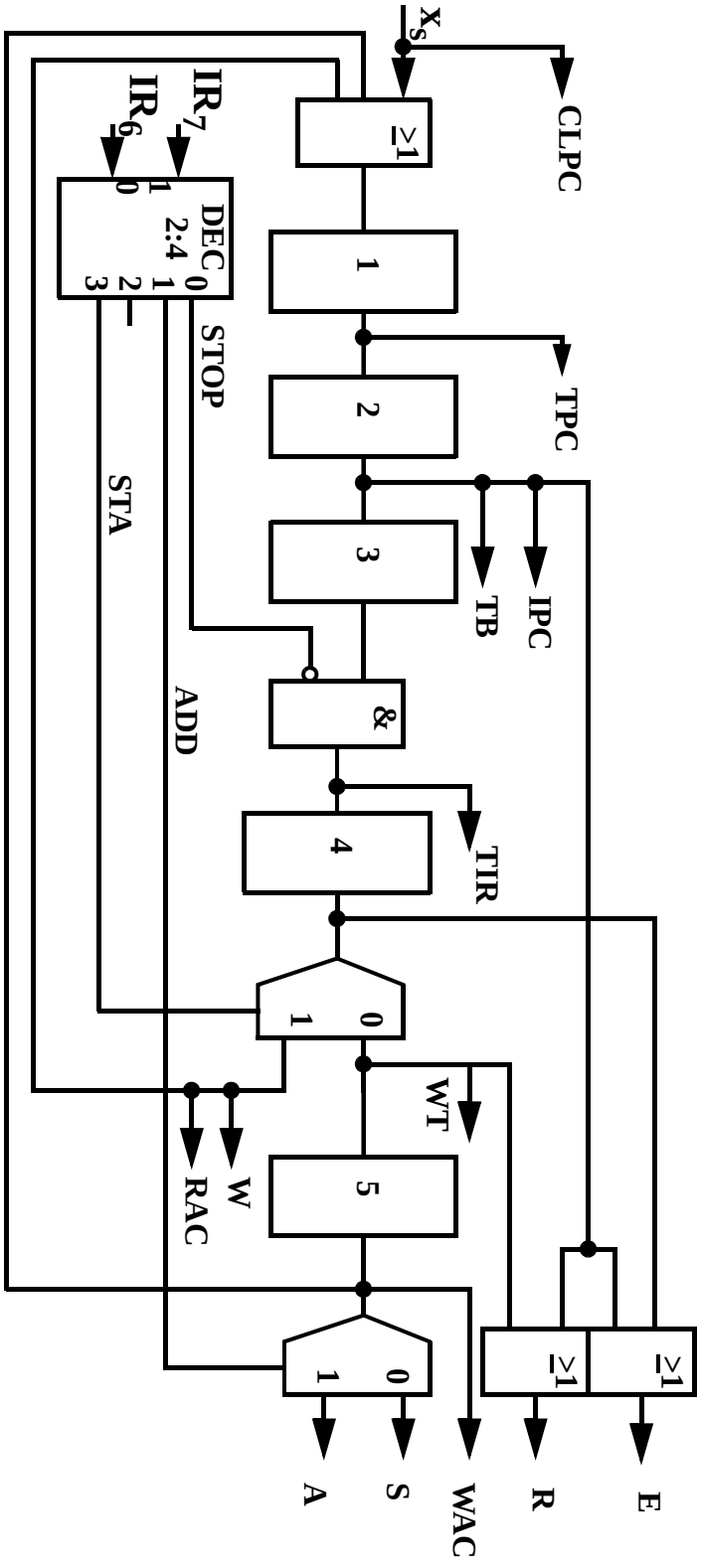


Figura 3.14: Unidad de control del CS1 mediante la técnica de un biestable por estado

3.2.4 Ejemplo de uso del computador simple 1

Vamos a desarrollar un ejemplo con el fin de presentar globalmente el funcionamiento de este Computador Simple 1. Para ello, elaboraremos un programa que sea capaz de llevar a cabo la siguiente función:

Ejemplo: *"Almacenar, en la dirección de memoria $3F, el resultado de sumar los datos almacenados en las direcciones de memoria $3E y $3D, restándole el contenido de la dirección de memoria $3C"*.

Ante este enunciado el usuario de CS1, utilizando su conjunto de instrucciones, elaborará un programa que resuelva el problema planteado. En primer lugar, lo describirá utilizando los mnemónicos de la forma siguiente:

1.- STA $20

2.- SUB $20

3.- ADD $3E

4.- ADD $3D

5.- SUB $3C

6.- STA $3F

7.- STOP

Obsérvese que las dos primeras instrucciones se necesitan para poner el acumulador a 0. (Aunque la Unidad de Datos de la Fig. 3.6 permite borrar el acumulador muy fácilmente, sin más que activar la señal ZAC, esta operación no ha sido definida como instrucción de CS1 y , por tanto, *el programador no puede* utilizarla). Las tres siguientes instrucciones realizan la suma deseada almacenanado el resultado en el acumulador. La instrucción 6 traslada ese resultado a la posición indicada ($3F). Por último, la instrucción 7, STOP, marca el final del programa.

A continuación, el programa debe ser interpretado al código máquina y almacenado en memoria, a partir de la posición $0 y en el mismo orden en que se va a ejecutar. El código máquina resultante se puede ver en la Fig. 3.15, en donde se ha representado el contenido de la memoria RAM. Como se ve, en la columna de la izquierda aparece la dirección de memoria, expresada en hexadecimal, y a la derecha, segunda columna, el contenido de cada una de las palabras de la memoria. Desde la posición $00 hasta la posición $06 está el programa. Por tanto, en todas esas palabras, los dos primeros bits nos indicarán qué operación se va a realizar, y los seis bits restantes qué dirección de memoria está implicada en esa operación.

Por otra parte, en las posiciones que van desde la $3C hasta la $3E están almacenados los datos, de ocho bits, con los cuales se va a trabajar. En la posición $3F se almacenará, al final del proceso, el resultado obtenido. Por último, la posición de memoria $20 se utiliza de manera transitoria para poner a 0 el acumulador.

En la Fig. 3.15 se han puesto los valores correspondientes a un caso particular: en decimal los sumandos son 126 en M($3E), 105 en M($3D) y 188 en M($3C). Tras la suma, el resultado que aparece en M($3F) es 43. Obviamente el CS1 opera en binario, por lo que los correspondientes valores son: 1011 1100 ($BC), 0110 1001 ($69), 0111 1110 ($7E) y 0010 1011 ($2B). Se invita al lector a que indique, instrucción a instrucción, cuál es el contenido de AC, de M($20) y de M($3F).

$A	[M($A)]	Instrucción/Dato	
00	1 1 1 0 0 0 0 0	STA $20	PROGRAMA
01	1 0 1 0 0 0 0 0	SUB $20	
02	0 1 1 1 1 1 1 0	ADD $3E	
03	0 1 1 1 1 1 0 1	ADD $3D	
04	1 0 1 1 1 1 0 0	SUB $3C	
05	1 1 1 1 1 1 1 1	STA $3F	
06	0 0 x x x x x x	STOP	
20	d d d d d d d d	Dato irrelevante	DATOS
3C	1 0 1 1 1 1 0 0	$188_{(10}$ = $BC	
3D	0 1 1 0 1 0 0 1	$105_{(10}$ = $69	
3E	0 1 1 1 1 1 1 0	$126_{(10}$ = $7E	
3F	0 0 1 0 1 0 1 1	$126_{(10}$ + $105_{(10}$ - $188_{(10}$ = $43_{(10}$ = $2 B	

Figura 3.15: Programa en lenguaje máquina del ejemplo (Decimal: 126 + 105 - 188)

3.3 COMPUTADOR SIMPLE 2 (CS2)

El Computador Simple 1 diseñado en el apartado anterior, nos ha permitido conocer los elementos más básicos del funcionamiento de los computadores. Sin embargo, su gran sencillez ha limitado la capacidad y la potencia de su uso.

La diferencia entre CS1 y los computadores reales es todavía sustancialmente grande. Existen varias líneas de progreso pendientes: se puede diversificar el tipo de instrucciones, diversificar el modo de direccionamiento, incluir instrucciones de más de una palabra, flexibilizando sus campos de código de operación y de dirección, incluir capacidad de entrada/salida, incluir control de procesos, etc.

El propósito de este apartado es presentar un nuevo sistema digital al que llamaremos Computador Simple 2 (CS2), que sirva de puente entre CS1 y los computadores reales. En particular, la finalidad de CS2 es enriquecer el juego de instrucciones sin perder la visión de CS2 como un sistema digital global. No obstante será básicamente el mismo de CS1 (1 palabra para instrucción, sólo un proceso en la ejecución, etc.), con lo que CS2 seguirá siendo bastante simple.

El desarrollo de este apartado comienza exponiendo la pluralidad de las instrucciones de los computadores (epígrafe 3.1) para, a continuación, detallar un conjunto de 16 instrucciones que será el que entienda CS2 (epígrafe 3.2). Seguidamente (epígrafe 3.3) se presenta CS2 a nivel estructural, si bien no se llega al diseño RT del controlador. Por último se realizan tres ejemplos de programación con CS2, para ilustrar el uso de su juego de instrucciones.

3.3.1 La pluralidad de instrucciones a nivel ISP

El juego de instrucciones de los computadores se diversifica de dos formas básicamente. La primera es ampliando los tipos de operación; esto es, pluralizando las tareas que puede ejecutar. La segunda es aumentando los modos de direccionamiento; esto es, flexibilizando las maneras en que se indican los operandos. A continuación presentaremos de forma introductoria estos conceptos.

3.3.1.1 Tipos de instrucciones

El tipo de instrucciones diseñadas para el CS1 ha sido muy reducido lo que ha sido debido, básicamente, al hecho de poseer sólo dos bits para indicarle a la unidad de control qué tipo de operación queremos llevar a cabo. Por ello, los computadores reales disponen de un campo de código de operación con un número de bits superior. De esa manera se le pueden definir operaciones de tipos muy diversos, algunas de las cuales se ven a continuación:

- DE TRANSFERENCIA DE DATOS: corresponden a movimientos de datos entre registros y palabras de memoria.
- ARITMÉTICAS: están dirigidas a realizar operaciones aritméticas básicas. Incluyen siempre la suma y la resta binaria, tanto con acarreo como sin él, así como alguna opción para aritmética decimal (BCD). Muy común es también disponer de la multiplicación y, en menor medida, de la división. Operaciones especiales tales como negar un número (complemento a 2), incrementar/decrementar, multiplicar/dividir por potencias de 2 (desplazamientos), comparar magnitudes, etc., también suelen estar definidas como instrucciones a nivel ISP.
- LOGICAS: realizan una operación lógica entre datos. Se suele incluir el conjunto completo de operadores booleanos (AND, OR, NOT) junto con otras operaciones como la exclusive-OR, producir desplazamiento cíclico a derecha o izquierda, etc.

Tanto las instrucciones aritméticas como lógicas no implican cambios sustanciales sobre la Unidad de Datos ya utilizada para CS1 (Fig. 3.6). Para realizarlas basta utilizar una ALU suficientemente potente, junto con un acumulador que incorpore la funcionalidad requerida (desplazamiento, cuenta, etc.).

- DE SALTO: se incluyen para que la ejecución del programa no sea siempre lineal. Generalmente incorporan tanto el salto incondicional como el condicional, esto es, el salto en el programa se realizará siempre (salto tipo "GOTO") ó sólo cuando ocurra determinada condición en el proceso de datos (ramificación o salto tipo "IF ... THEN ... ELSE ...), respectivamente. Para incorporar saltos al recibir el registro IR la operación de salto, a una dirección de memoria determinada, sólo tendrá que transferir esa dirección al registro PC. De esta manera en el siguiente ciclo de búsqueda el ordenador busca la instrucción que hay en esa nueva dirección de memoria.
- DE LLAMADA Y REGRESO DE SUBRUTINAS: estas instrucciones permiten ejecutar un trozo de programa situado en otra parte de la memoria y, a su conclusión, regresar al lugar del pro-

grama principal por donde éste íba. En cierto sentido son también instruciones de salto. Por su especial importancia vamos a detallarlas un poco más.

Supóngase, por ejemplo, que en la instrucción $7 hay una llamada a una subrutina cuyo programa empieza en la instrucción $20 y termina con la instrucción $29 (ésta es necesariamente la instrucción de regreso de subrutina). Tras ejecutarse la instrucción de llamada ($7), deberá seguirse por la $20, ejecutar hasta la $29 y regresar a $8 para seguir el programa. Puesto que el contador de programa apunta en todo momento a la instrucción que ha de ejecutarse, es necesario almacenar la dirección de retorno ($8) cuando se ejcuta la llamada a subrutina (PC se carga, pues, con $20). Así, al realizar el regreso de la subrutina ($29) podrá recuperarse la dirección de retorno $8. Es más, si se encadenan llamadas a subrutinas dentro de subrutinas, el orden en que se almacenan las direcciones de retorno es justamente el inverso al orden en que serán recuperadas. Para ello es necesario contar con una memoria, tipo LIFO (Last In First Out), a la que también se le da el nombre de "pila".

En la Fig. 3.16 se muestra el comportamiento de una pila realizada con una memoria RAM y un contador que indica, en nuestro caso, cuál es la última dirección de la pila que contiene información ($K en Fig. 3.16), se denomina "puntero de pila" y se representa por SP (*Stack Pointer*). Si se quiere introducir un dato en la pila (operación de PUSH), se tendrá que decrementar el puntero para que señale la primera dirección sin información de la pila, transfiriendo a continuación el dato a esa dirección de la RAM.

Por el contrario, si lo que se pretende es leer el contenido de la pila (operación de PULL), se tendrá que transferir el contenido de esa posición de memoria fuera de la pila, seguido del incremento de SP necesario para que el puntero de pila vuelva a señalar la primera posición de la pila que contiene información.

Haciendo uso de la pila el proceso de llamada a subrutina consiste, simplemente, en almacenar (PUSH) el contenido del registro PC en la pila y, tras ello, grabar en el registro PC la dirección de memoria en donde se encuentra la subrutina. Una vez finalizada ésta, para volver a la línea del programa principal por donde éste se encontraba, sólo hay que transferir el contenido de la pila (PULL), al registro PC para seguir la ejecución normal del programa.

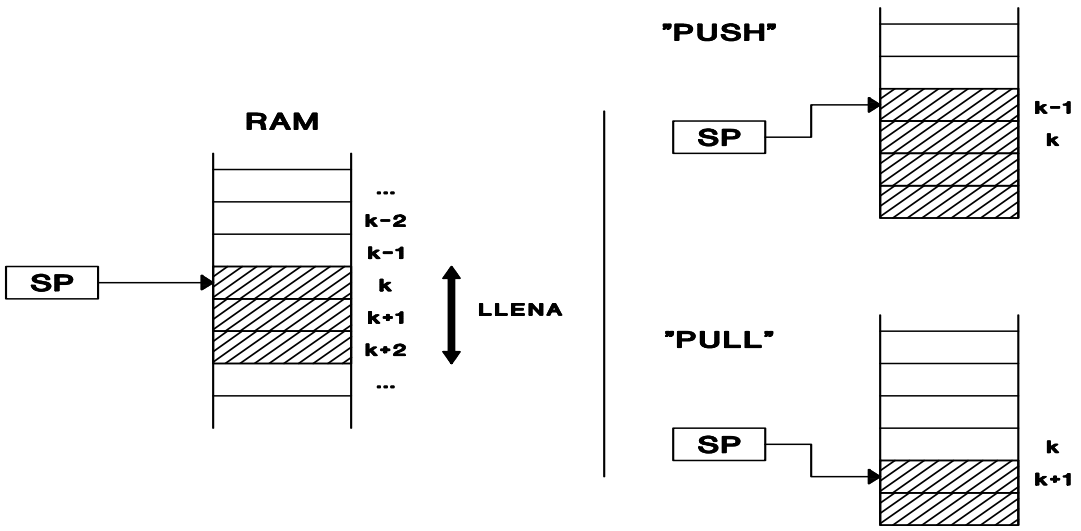


Figura 3.16: Funcionamiento de la pila y de su puntero (SP: *Stack Pointer*) en CS2

- DE CONTROL DE "ESTADO": son instrucciones que actúan sobre señales llamadas "banderines" (flags). Estas señales informan sobre el estado del procesador (p. ej., si ha habido acarreo) y suelen almacenarse en biestables, aunque al conjunto de éstos se les denomina registro de condiciones (o de estados). Algunos de los banderines más frecuentes en la manipulación de datos son: el de acarreo (C, carry), el de desbordamiento (V, overflow), el de signo (S) y el de resultado cero (Z, zero; Z=1 si el resultado es cero y Z=0 en otros casos). Además, se suelen incluir banderines para funciones de control de procesos tales como el tratamiento de las interrupciones (ver Apartado 4). Muchas de las instrucciones de ramificación condicional, tienen en cuenta el valor de estas señales para determinar la secuencia de instrucciones a realizar.

- MISCELÁNEA: aquí se agrupan las restantes instrucciones que no pueden incluirse en los otros grupos. Las hay de muchos tipos y varían de un computador a otro. A título de ejemplo y sólo con el fin de mostrar gran diversidad de casos posibles, podemos mencionar: las de manipulación de registros específicos internos (tales como el puntero de pila), las de entrada/salida de datos, las de operaciones especiales con datos (conversiones a/desde punto flotante en computadores "matemáticos" o generación/detección de paridad en computadores "de comunicación"), etc.

3.3.1.2 Modo de direccionamiento

En el CS1 se utilizaron instrucciones en cuyo campo de direcciones se encontraba la dirección de memoria en donde se localizaba al operando. A pesar de ser un modo de direccionamiento bastante generalizado, no es el único posible. En computadoras, minicomputadoras y microcomputadoras hay una amplia gama de modos de direccionamiento, de los que veremos algunos de ellos a continuación:

- INMEDIATO: en el campo de dirección de la instrucción, no aparece la dirección en memoria del operando, sino directamente el operando.

- DIRECTO: es el utilizado en el CS1. En él, el campo de dirección contiene la dirección de memoria en la cual se encuentra el operando. Existen algunas variaciones sobre este modo de direccionamiento. Supongamos un bus de direcciones de 16 bits (A_{15} - A_0), lo que da un espacio de memoria de 64K palabras. A veces, la memoria se considera divida en secciones o "páginas", por ejemplo, en 256: la primera es la página 0 y está constituida por las 256 palabras cuya dirección tiene A_{15} - A_8 = \$00; la página 1 tiene A_{15} - A_8 = \$01; ... ; y la página 255 tiene A_{15} - A_8 = \$FF. Con este planteamiento, el direccionamiento directo se subdivide en el llamado "extendido", en el que el campo de direcciones de la instrucción ocupa los 16 bits de direcciones, y en el "paginado" (a la página 0, a la página "actual", ...), en el que el campo de direcciones sólo informa de los últimos 8 bits (A_7 - A_0). En este último caso, el operando está en la palabra \$$A_7$ - A_0 de la página correspondiente (la 0, la "actual", etc.).

- INDIRECTO: ahora la parte dirección contiene la dirección de la memoria que contiene la dirección de memoria donde está el operando. En la Fig. 3.17 se muestra la diferencia, al usar el modo directo y el indirecto, al realizar la operación de suma, lo que se llama suma (ADD) y suma indirecta (ADDI).

- RELATIVO: supongamos que el campo de dirección de la instrucción contiene un número N. En el modo relativo la dirección del operando se encuentra sumando ese número N al contenido del registro PC en ese momento. Dicho número N puede ser positivo o negativo. Por lo

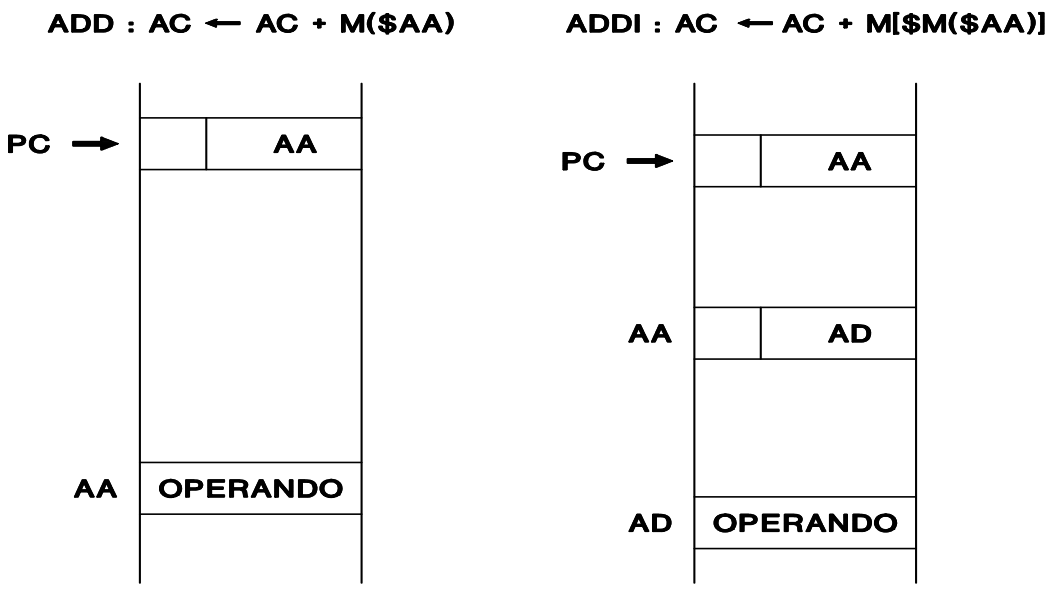


Figura 3.17: Ejemplo de direccionamiento indirecto

tanto, la dirección del operando puede ser mayor (está después) o menor (está antes) que la instrucción.

- INDEXADO: el modo de direccionamiento es similar al modo relativo que vimos antes; es decir, el campo de dirección contiene un número N, positivo o negativo. La diferencia consiste en que este modo indexado requiere la existencia de un registro especial, denominado registro índice (X, Y, ...). La posición de memoria del operando se obtendrá ahora sumando N al contenido de ese registro (X, Y, ...).

- REGISTRO INDIRECTO: ahora se utiliza otro registro especial P, llamado a menudo registro puntero, en el cual se almacena la dirección de la memoria donde se encuentra el operando buscado. Una instrucción que solicite una operación con el modo de direccionamiento mediante registro indirecto, no requiere ningún contenido en el campo de dirección de la misma.

- IMPLÍCITO ó INHERENTE: en este tipo de direccionamiento, la parte de dirección no contiene información, y si la contiene no es de interés. Esto ocurre en aquellas instrucciones que de por sí incluyen, implícitamente, el origen y el destino de la operación. Así, por ejemplo, la operación de "poner a cero" el registro acumulador, o la operación de parada, son operaciones de direccionamiento implícito.

- MISCELÁNEA: en los modos de direccionamiento se encuentra también una gran diversidad de casos específicos cuando se pasa de un computador a otro. Así, otros modos de direccionamiento particulares son: los que están basados en el puntero de pila; los que mezclan dos o más modos anteriores (p. ej., el relativo indirecto); los que permiten direccionar varios operandos simultáneamente (direccionamiento "múltiple"); los que direccionan registros internos (y no posiciones de memoria); etc.

Conceptualmente, las instrucciones de los procesadores reales contienen 3 campos ya que, además del campo de operación desdoblan su campo de direcciones en dos: el del modo de direccionamiento y el de direcciones propiamente.

3.3.2 Conjunto de instrucciones de CS2

Partiendo de la estructura del CS1, y con muy pocas modificaciones, podemos solventar algunas de sus deficiencias. El nuevo computador CS2 surge esencialmente para enriquecer el juego de instrucciones de CS1, incorporando algunos de los aspectos tratados en los apartados anteriores. Este segundo computador simple, no lo detallaremos tanto como lo hicimos con el primero. Para su presentación partiremos del tipo de palabra de instrucción que utiliza, de ahí pasaremos a definir las operaciones que realiza, y por último, ya en el próximo epígrafe, comentaremos la estructura que debe tener.

La palabra de instrucción es una palabra de doce bits, de los cuales los cuatro bits más significativos componen la parte de operación, mientras que la parte de dirección estará formada por los ocho bits restantes. Esto nos lleva a disponer de dieciseis (2^4) operaciones posibles, y a tener que utilizar una memoria RAM de 256 (2^8) direcciones de memoria, cada una de ellas con un tamaño de 12 bits.

Las dieciseis instrucciones elegidas son las recogidas en la Tabla 3.2. Seguidamente desarrollaremos cada una de ellas. De forma global puede observarse que, a pesar de ser sólo 16 instrucciones, las hay de transferencia de datos, aritméticas, lógicas, de salto (incondicional y condicional), de control de estado y de subrutinas. Asímismo se incluyen los modos de direccionamiento siguientes: inmediato (IMmediate) en LAIM; directo en LDA, STA, ADD, SUB, JMP, BCS, DBZ y JSR; indirecto en ADDI; implícito en ROR, ROL, CLC, SEC, STOP y RTS.

En el desarrollo de cada instrucción que se presenta a continuación se indica: sus descripciones en mnemónico, como operación RT y en código máquina; su realización mediante secuencia de microoperaciones con el sistema digital de la Fig. 3.18 (ver próximo epígrafe); y una breve explicación verbal.

- **OPERACIÓN 0 LAIM dd**

Cargar acumulador (modo inmediato) (*Load Accumulator IMmediate*)

$AC \leftarrow AA$ IR: 0 0 0 0 $IR_7 IR_6 IR_5 IR_4$ $IR_3 IR_2 IR_1 IR_0$

Microoperaciones:		
	1. $RT \leftarrow 0_{11\text{-}8}\ IR_{7\text{-}0}$	R8,CIM,WT,ZA
	2. $AC \leftarrow AC + RT$	RT,A,WA

El valor “dd” suministrado en el campo de direcciones de la instrucción (IR_7:IR_0) se almacena en el acumulador. Se usa para almacenar datos conocidos por el programador. Durante su ejecución se pierde el contenido previo del registro RT y del biestable BC.

- **OPERACIÓN 1 LDA AA**

Cargar acumulador (modo directo) (*LoaD Accumulator*)

$AC \leftarrow M$ IR: 0 0 0 1 $A_7\ A_6\ A_5\ A_4\ A_3\ A_2\ A_1\ A_0$

Microoperaciones:		
	1. $MAR \leftarrow IR$	TIR
	2. $RT \leftarrow RAM$; $AC \leftarrow 0$	ZA,WT,R
	3. $AC \leftarrow AC + RT$	RT,A,WA

Tabla 3.2 Conjunto de instrucciones del CS2

CO	MNEM	OPERACIÓN	TIPO DE OPERACIÓN
0	LAIM	AC ← dd	Transf.: cargar AC con el dato dd
1	LDA	AC ← M	Transf.: cargar en AC
2	STA	M ← AC	Transf.: almacenar en M
3	ADD	AC ← AC + M	Arit.: suma
4	SUB	AC ← AC - M	Arit.: resta
5	ADDI	AC ← AC+M($[M]$$_{7-0}$)	Arit.: suma indirecta
6	ROR	SHR(C,AC),C ← AC_0	Lóg.: desplazamiento a derecha (con carry)
7	ROL	SHL(AC,C),C ← AC_{11}	Lóg.: desplazamiento a izquierda (con carry)
8	JMP	GOTO $AA	De salto: incondicional
9	BCS	C=0 : GOTO N+1 C=1 : GOTO $AA	De salto: condicional; si C=1 salta a $AA
A	DBZ	M ← M - 1 Z=0 : GOTO N+1 Z=1 : GOTO N+2	De salto: decrementa y salta si cero
B	CLC	C ← 0	De estado: borrar carry
C	SEC	C ← 1	De estado: carry a "1"
D	STOP	--- (GOTO S_0)	De control: parada
E	JSR	SP ← SP - 1 M(SP) ← PC GOTO $AA	De salto: salto a subrutina
F	RTS	PC ← [M(SP)] SP ← SP + 1	De salto: retorno de subrutina

NOTAS:

- **dd, 8 bits que están en el campo de direcciones de la instrucción.**
- **M es la palabra de memoria cuya dirección se indica en el campo de direcciones de la instrucción.**
- **C es el bit de acarreo (Carry).**
- **N es la dirección donde se encuentra la instrucción actual.**
- **Z es el bit que se activa a 1 cuando el valor del resultado es cero (Zero)**
- **LAIM y LDA borran el bit de acarreo, C.**

Se trata de una operación de transferencia entre registros. Se trasvasa al registro acumulador el contenido de la memoria RAM en la dirección que indica la instrucción, AA.

- **OPERACIÓN 2 STA AA**

Almacenar el acumulador en la memoria (modo directo) (*STore Accumulator*)

M ← AC IR: 0 0 1 0 A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0

Microoperaciones:	1. MAR ← IR	TIR
	2. RAM ← AC	RA,W

Es una operación de transferencia entre registros. El contenido del registro acumulador se trasvasa a la memoria RAM del sistema, en la dirección que indica la instrucción.

- **OPERACIÓN 3 ADD AA**

Suma (modo directo) (*ADDition*)

AC ← AC + M IR: 0 0 1 1 A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0

Microoperaciones:	1. AR ← IR	TIR
	2. RT ← RAM	WT,R
	3. AC ← AC + RT	RT,A,WA

Es una operación aritmética. Se trata de sumar, el contenido del registro acumulador con el dato almacenado en la memoria RAM del sistema, en la dirección que indica la instrucción. El resultado se almacena en el registro acumulador.

- **OPERACIÓN 4 SUB AA**

Resta (modo directo) (*SUBstraction*)

AC ← AC - M IR: 0 1 0 0 A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0

Microoperaciones:	1. AR ← IR	TIR
	2. RT ← RAM	WT,R
	3. AC ← AC - RT	RT,S,WA

Operación aritmética. Al contenido del registro acumulador se le resta el contenido de la memoria RAM, en la dirección que le indica la instrucción. El resultado se almacena en el registro acumulador.

- **OPERACIÓN 5 ADDI AA**

Suma indirecta (modo indirecto) (*ADDition Indirect*)

AC ← AC + M[$[M]$$_{7\text{-}0}$] IR: 0 1 0 1 A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0

Microoperaciones:	1. AR ← IR	TIR
	2. $IR_{7\text{-}0}$ ← $RAM_{7\text{-}0}$	R,W8
	3. AR ← IR	TIR

4. RT ← RAM R,WT

5. AC ← AC + RT RT,A,WA

Operación aritmética. En ella se le suma, al contenido del registro acumulador, el dato cuya dirección de memoria se encuentra almacenada en la dirección de memoria que se indica en la instrucción (Fig. 3.17). El resultado de esta operación se almacena en el registro acumulador.

- **OPERACIÓN 6 ROR**

Rotación a derecha del acumulador (modo implícito) (*ROtate Right*)

AC ← SHR(AC,C) IR: 0 1 1 0 - - - - - - - -

Microoperaciones: 1. AC ← SHR(AC,C); BC ← AC_0 RR

Operación de tipo lógico. Se trata del desplazamiento circular a la derecha del contenido del registro acumulador junto con el del biestable de acarreo. Por ello, el bit menos significativo AC_0 se almacena en el biestable BC, mientras que la entrada serie en el acumulador es el bit C.

- **OPERACIÓN 7 ROL**

Rotación a izquierda del acumulador (modo implícito) (*ROtate Left*)

AC ← SHL(AC,C) IR: 0 1 1 1 - - - - - - - -

Microoperaciones: 1. AC ← SHL(AC,C); BC ← AC_{11} RL

Operación de tipo lógico, similar a la anterior salvo que el desplazamiento circular es a la izquierda.

- **OPERACIÓN 8 JMP AA**

Salto incondicional (modo directo) (*JuMP*)

GOTO $AA (PC← AA) IR: 1 0 0 0 A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0

Microoperaciones: 1. PC ← IR_{7-0} R8,WPC

Operación "de salto". En este caso se trata de que la próxima instrucción a ejecutar es la de la dirección de memoria indicada por la instrucción.

- **OPERACIÓN 9 BCS AA**

Salta si el acarreo es 1 (modo directo) (*Branch if Carry Set*)

C: GOTO $AA (PC← AA) IR: 1 0 0 1 A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0

Microoperaciones: 1. C: PC ← IR_{7-0} R8,WPC

Es una operación "de salto". Es similar a la anterior salvo que es salto condicional, es decir, depende del estado en que se encuentre el biestable BC: si está a "0", no se producirá el salto y seguirá con la instrucción siguiente (N+1); si está a "1" se produce el salto a la instrucción almacenada en $AA.

- **OPERACIÓN A DBZ AA**

Decrementa y salta si es cero (modo directo) (*Decrement and Branch if Zero*)

IF(M-1=0) THEN GOTO N+2 IR: 1 0 1 0 A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0

Microoperaciones:	1. MAR ← IR		TIR
	2. RT ← RAM		R,WT
	3. RT ← RT - 1	DT	
	4. RAM ← RT ;		RT,W
	Z: PC ← PC + 1		IPC

Operación "de salto". Esta operación decrementa en una unidad el contenido de la dirección de memoria determinada por la instrucción y, dependiendo del resultado, ejecuta la instrucción siguiente (N+1), si es no nulo, o salta a la otra que sigue (N+2), si es nulo. En todo caso, el resultado obtenido tras decrementar se vuelve a almacenar en la misma posición de memoria. La variable Z, llamada variable CERO, nos indica si el contenido del registro del cual se toma es cero (Z=1), o no (Z=0) (Fig. 3.18).

- **OPERACIÓN B CLC**

Borrar acarreo (modo implícito) (*CLear Carry*)

C ← 0 IR: 1 0 1 1 - - - - - - - -

Microoperaciones:	1. BC ← 0	CC

Operación "de estado". Coloca a "0" el biestable BC.

- **OPERACIÓN C SEC**

Poner a "1" el acarreo (modo implícito) (*SEt Carry*)

C ← 1 IR: 1 1 0 0 - - - - - - - -

Microoperaciones:	1. BC ← 1	SC

Operación "de estado". Coloca a "1" el biestable BC.

- **OPERACIÓN D STOP**

Parada (modo implícito) (*STOP*)

NOP (and GOTO S_0) IR: 1 1 0 1 - - - - - - - -

Microoperaciones:	---------- (S_0)	--

Operación de control. Se trata de la operación de parada y es la última del programa. Como se ve no hay que hacer ninguna operación; sólo hay que colocar al sistema en el estado de espera S_0, del cual saldrá de nuevo cuando se active la señal de comienzo XS.

- **OPERACIÓN E JSR AA**

Llamada a subrutina (modo directo) (*Jump SubRoutine*)

PUSH[PC], GOTO $AA IR: 1 1 1 0 A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0

Microoperaciones: 1. SP ← SP - 1 DS

2. AR ← SP TSP

3. RAM ← PC RPC,W

4. PC ← IR_{7-0} R8,WPC

Operación "de salto". Es el salto a subrutina. Para ello se almacenará la dirección de retorno en la pila (PUSH [PC]) y, después, se cargará el contador de programa PC con la dirección de comienzo de la subrutina ($AA, dada en la instrucción). (La operación de pila ya ha sido descrita en el epígrafe 3.1.1, Fig. 3.16; en nuestro caso, PUSH [PC] requiere las transferencias: 1º) SP ← SP - 1 y 2º) M(SP) ← PC).

- **OPERACIÓN F RTS**

Retorno de subrutina (modo implícito) (*ReTurn of Subroutine*)

PC ← PULL (pila) IR: 1 1 1 1 - - - - - - - -

Microoperaciones: 1. AR ← SP TSP

2. PC ← RAM ; SP ← SP + 1 R,WPC,IS

Operación "de salto". Se trata del retorno de una subrutina. La instrucción RTS es la última de cualquier subrutina. Su ejecución hace que el contador de programa se cargue con el dato extraido de la pila (PC ← PULL (pila)). (En nuestro caso, PC ← PULL (pila) requiere las transferencias:
1º) PC ← M(SP) y 2º) SP ← SP + 1).

3.3.3 Estructura del computador simple 2

En la Fig. 3.18 se muestra la estructura completa del CS2. En ella se puede comprobar que este esquema es básicamente el del CS1, al que se le han añadido los elementos necesarios para poder llevar a cabo las distintas operaciones que antes se definieron. Para realizar las instrucciones de subrutina se ha añadido el registro SP, de 8 bits, el cual necesita dos señales de control: **IS**, que incrementa su contenido en una unidad y **DS**, la cual decrementa el contenido del registro en una unidad. La aparición de este registro obliga al registro de direcciones de memoria, MAR, que posea un nuevo bus de entrada, procedente del registro SP. Por tanto, MAR tendrá una nueva señal de control, **TSP**, la cual trasferirá el contenido del registro SP al registro MAR.

Otro elemento nuevo que hemos introducido es el biestable de acarreo BC. Este biestable y el registro acumulador forman un "elemento" en el que comparten las operaciones de carga en paralelo (reciben la salida C_{out} y de resultados de la ALU) y de desplazamiento circular a derecha o a izquierda. Como biestable individual, BC tiene otras dos señales de control: **CC**, que coloca el biestable a "0" y **SC**, que lo coloca a "1". Su línea de salida, C, parte hacia el controlador del sistema con la idea de que informe al mismo del estado en que se encuentra el biestable en cada momento. A esta línea se le da el nombre de CARRY y se utiliza en las operaciones: BCS, ROR, ROL, CLC y SEC.

Al registro RT se le ha incorporado la operación "decrementar". Además, se le ha añadido una nueva salida, Z (señal CERO), de manera que informa a la unidad de control sobre si el contenido de ese registro es o no nulo (Z=1 ó Z=0, respectivamente). Esto se utiliza, por ejemplo, en la instrucción DBZ.

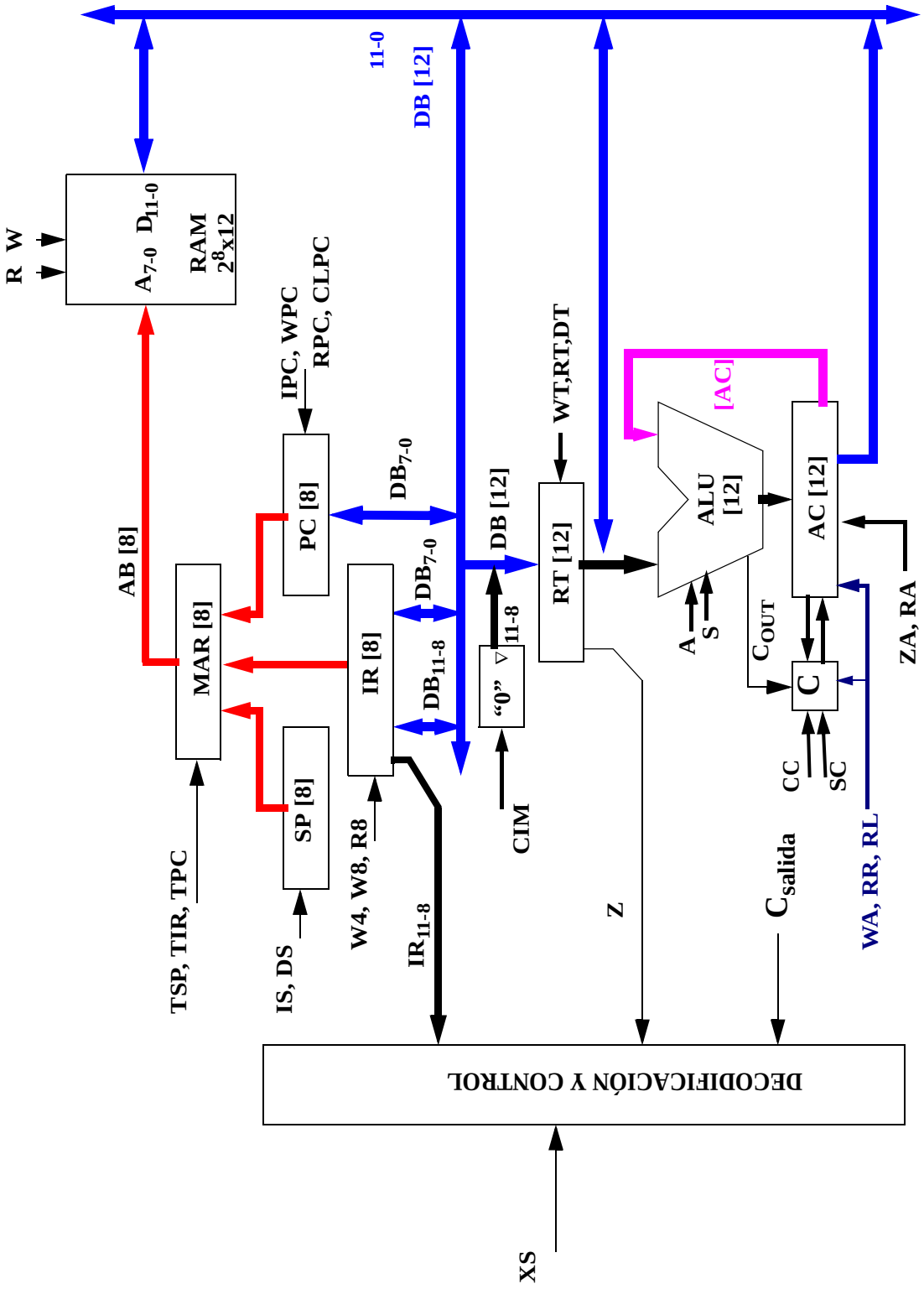


Figura 3.18: Estructura del Computador Simple 2

El registro de instrucciones IR, se ha modificado levemente. Ahora se graban la parte de operación y de dirección por separado, es decir, cada una de ellas necesita activar una señal diferente: **W4**, para la de operación y **W8**, para la de dirección. Además, el bus que accede a la parte de dirección es bidireccional. Por lo tanto hay una señal, **R8**, que permite la lectura de la parte de dirección del registro IR. Esto es necesario, por ejemplo, para poder llevar a cabo la operación de salto incondicional (JMP).

El registro PC ha sido alterado, de manera que ahora posee un bus bidireccional desde el bus interno del sistema. Esto le va a permitir recibir información, **WPC**, de otros registros del sistema, como, por ejemplo, son IR en las operaciones JMP y JSR, o la memoria RAM del sistema en las instrucciones JMPI y RTS. Además puede enviar directamente su contenido a la memoria RAM, como se necesita en la operación de salto a subrutina (JSR).

El bus de salida del registro RT accede directamente al bus de datos, por lo que ahora es condicional. Por ello necesita una señal de control, **RT**, que antes no tenía y que, cuando se activa, produce la lectura del registro tanto hacia la ALU como hacia el bus interno del sistema. Esta conexión resulta necesaria para llevar a cabo la operación DBZ. En esa operación se decrementa en una unidad el contenido de un dato almacenado en la memoria, para lo cual se le añade al registro RT la señal de control, **DT**, que realiza dicha operación con el contenido de ese registro.

Al registro acumulador (AC) se le ha dotado de la posibilidad de desplazamiento a derecha e izquierda, al activar, respectivamente las señales, **RR** y **RL**. También se le ha añadido la señal de puesta a "0", la cual se utiliza en la operación LDA. Por otra parte hay que resaltar el nuevo tamaño de los buses del sistema, ya que aquéllos en los que están implicados datos (como son: el bus interno, la mayoría de los buses asociados a él y los relacionados con la A.L.U.) son ahora de 12 bits. Mientras que los relativos a direcciones (como los que están relacionados con el registro MAR o con el registro PC) son ahora de 8 bits.

Por último, centremos nuestra atención sobre el circuito de control. Su realización para CS2 no presenta ningún cambio conceptual respecto al caso de CS1. Es decir, ahora habría que agrupar todos los desarrollos en microoperaciones de las 16 instrucciones en una sola carta ASM y, de aquí, generar el correspondiente controlador. El mayor cambio de CS2 respecto a CS1 es el aumento de la complejidad: ahora son 16 casos, con decenas de variables de entrada y de salida. Resolver este problema manualmente bajo el criterio de reducir coste requiere un esfuerzo excesivo y posiblemente inviable. Por esta causa, estos problemas se afrontan utilizando herramientas de CAD (Computer Aided Design) que automatizan la tarea de minimización. (Esta opción pone de manifiesto un nuevo problema, de gran actualidad y muy "informático": hay que obtener buenos algoritmos de minimización y desarrollar los correspondientes programas que, además, deberán poseer prestaciones competitivas).

Alternativamente, para resolver el circuito "a mano", se puede optar por simplificar el proceso de diseño a costa de renunciar a la optimización del circuito. Esta "metodología", que se aplica en la realidad en una gran multitud de trabajos, consiste fundamentalmente en particionar el problema global en "sub-problemas". Así, en vez de resolver un todo muy complejo, se resuelven varias partes, cada una de ellas suficientemente simples. Como muestra de esta metodología en su grado más extremo, apliquémosla a la unidad de control CS2.

La carta ASM de la Fig. 3.19 muestra simbólicamente la operación de la unidad de control de CS2. Tras la búsqueda de la instrucción (FETCH) se decodifica su código de operación CO abriéndose tantas líneas de continuación como instrucciones se hayan definido (en nuestro caso, 16). En cada línea se realizaría la secuencia de microoperaciones correspondientes a esa instrucción, tal como han sido desarrollados en el Aptº 3.2. Así, basta sustituir por separado cada "Carta de instrucción" por su bloque ASM correspondiente. El circuito de control, realizado con un biestable por estado, tiene la organización

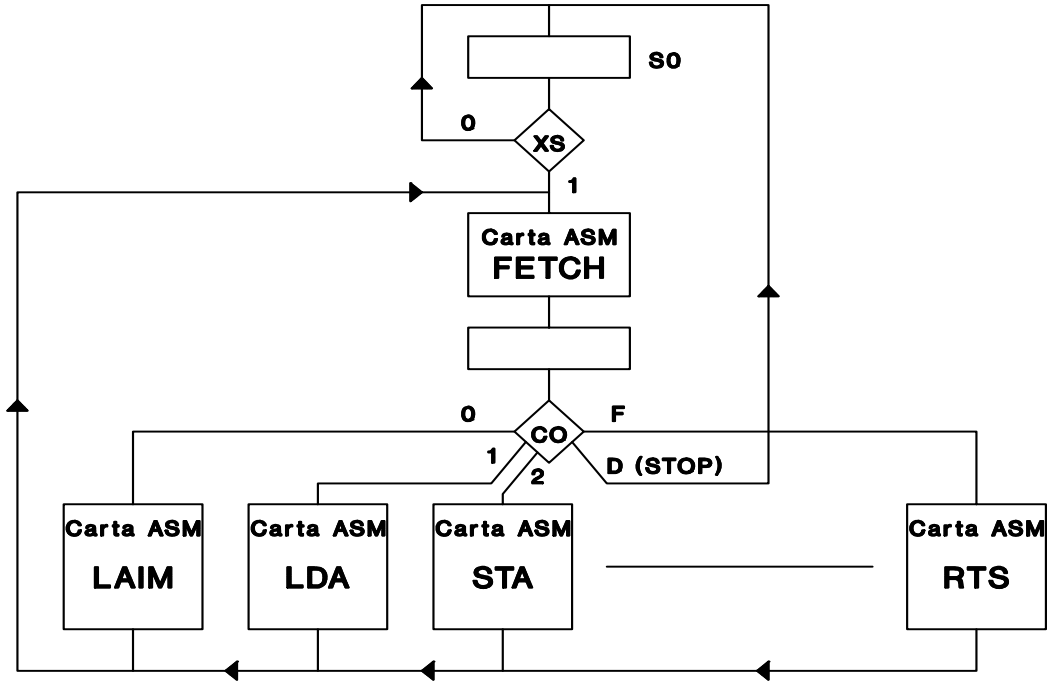


Figura 3.19: Carta ASM del controlador del CS2.

mostrada en la Fig. 3.18 **Fig. 3.20**. En esta figura, cada caja contiene los biestables y salidas que corresponden a la respectiva instrucción.

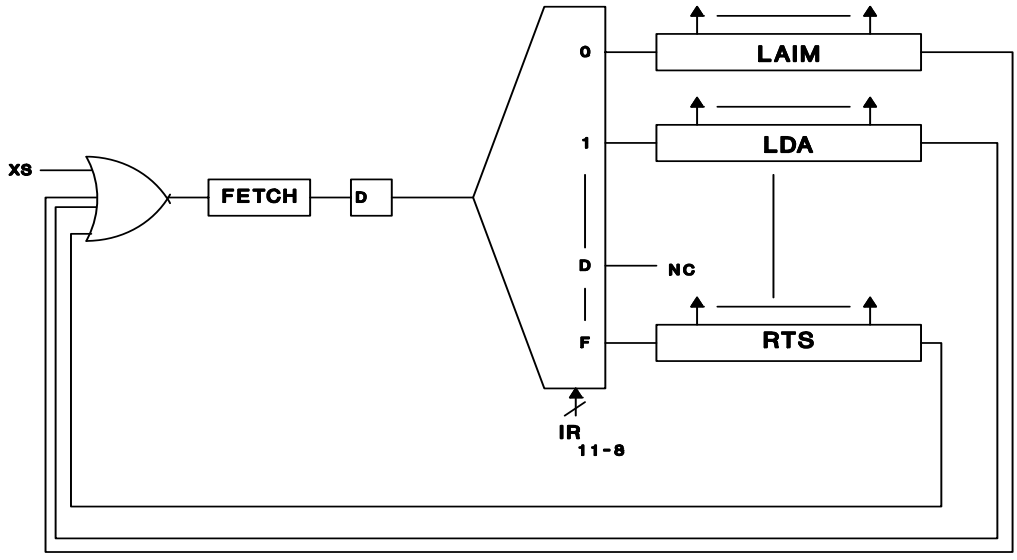


Figura 3.20: Circuito para la unidad de control del CS2.

3.3.4 Ejemplos de uso del computador simple 2

Vamos a desarrollar tres ejemplos que nos permitan comprender el funcionamiento de este CS2. Como se verá, CS2 se maneja como un auténtico computador, aunque todavía se hace necesario establecer algunas limitaciones al uso del mismo.

3.3.4.1 Ejemplo I: Suma de "n" sumandos

Como primer ejemplo se realizará la suma sin signo de "n" sumandos, escribiendo su resultado en una palabra de memoria, que denominaremos Resultado. Un valor 1 en el biestable BC al acabar la ejecución indicará que ha habido desbordamiento. Los números a sumar ocupan n palabras consecutivas de la memoria.

En la Fig. 3.21 se ha representado cómo sería el contenido de la RAM: la parte superior, que ocupa el programa (con las instrucciones del CS2 las cuales se han escrito mediante su mnemónico); algunos punteros o valores necesarios; y la zona de datos. Además, se ha representado el **ordinograma**[1] para la solución de programa adoptada.

Antes de iniciar los comentarios del programa, concretemos las posiciones de los valores que aparecen en el enunciado:

- La dirección del dato a sumar será un *puntero*, que se introducirá en el propio programa y se escribirá en la posición $AA. Inicialmente apuntará al primer sumando.
- El valor del número de sumandos “n” será un *puntero,* que se introducirá en el propio programa y se escribirá en la posición $AB. De esta forma, se puede cambiar el número de datos a sumar modificando el valor de “n” (en el propio programa).
- Los datos a sumar se suponen que ya están escritos y que, por ejemplo, ocupan las últimas posiciones de RAM, desde la $FF hasta la $FF-(n-1): $FF contiene el primer sumando, $FE el segundo y así sucesivamente hasta el último que está en $(FF-n+1). Así, el puntero del sumando será inicialmente $FF. Si los datos estuvieren en otro lugar (p. ej., desde la posición $59 hacia atrás), bastaría reescribir el programa poniendo $59 como valor inicial del puintero de sumandos.
- Arbitrariamente hemos elegido que el resultado se escriba en la dirección $AC.

Antes de alcanzar la respuesta final de un programa, que es la lista ordenada de las instrucciones que lo forman, *se requiere una descripción formal del algoritmo* que se va a desarrollar. Para ello existen diferentes mecanismos, como es el uso de *pseudolenguajes* o de formas visuales, como son los *ordinogramas* (erróneamente también referidos como organigramas). En nuestro caso, el algoritmo desarrollado obedece al *ordinograma* de la Fig. 3.21. Comienza con la **inicialización**, en nuestro caso asignando valor a los dos punteros y poniendo a 0 el acumulador. Seguidamente se entra en un **lazo** (*ciclo de instrucciones*) en el que sucesivamente: se acumula el nuevo sumando preguntando si ha habido desbordamiento; si no lo hay, se busca un nuevo sumando y se cierra el lazo, hasta completar todos los sumandos. La ejecución del programa acaba al completar la suma o cuando se detecta un desbordamiento.

1. En programación, un **ordinograma** representa el orden de los pasos o acciones de un algoritmo de manera gráfica mediante un diagrama de flujo. No se trata de una carta ASM, ya que, aunque también representan un algoritmo, en el ordinograma los distintos símbolos (acciones, decisiones,...) no llevan asociada una duración determinada en ciclos de reloj. Otro término equivalente es **organigrama**.

Direccion RAM	Contenido de la palabra	
$00	LAIM n	PROGRAMA
$01	STA $AB	
$02	LAIM $FF	
$03	STA $AA	
$04	LAIM 0	
$05	ADDI $AA	
$06	BCS $0B	
$07	DBZ $AA	
$08	DBZ $AB	
$09	JMP $05	
$0A	STA $AC	
$0B	STOP	
...		
$AA	FF (después, FE, FD,...)	Punteros
$AB	n (nº de sumandos)	
$AC	Resultado	DATOS
...		
$FF-(N-1)	Sumando n	
...	...	
$FE	Sumando 2	
$FF	Sumando 1	

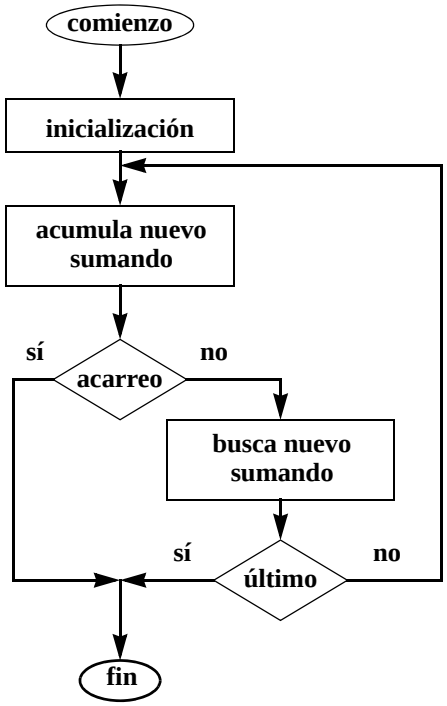


Figura 3.21: Contenido conceptual de la memoria para la suma de “n” sumandos y un ordinograma del ejemplo.

COMENTARIOS:

Las cinco primeras instrucciones del programa inicializan los registros. El número "n" se introduce en la posición $AB mediante dos instrucciones: primero se carga el acumulador con el número "n" usando el modo inmediato (LAIM n) y, después, se almacena el acumulador en la posición deseada (STA $AB). De igual forma se almacena la dirección del primer sumando $FF en la palabra $AA (LAIM FF y STA $AA). Por último, se borra el acumulador almacenando en él un 0 usando el modo inmediato (LAIM 0).

La operación $05 (ADDI $AA) es una suma indirecta, en donde al contenido del acumulador se le añade el dato que se encuentra almacenado en la dirección de memoria que indica la palabra $AA. Al comienzo de la ejecución, la palabra $AA contiene la dirección de memoria $FF, que es donde se encuentra el primer sumando. La primera ejecución de esta instrucción realizará, por tanto, la suma del acumulador (actualmente a 0) con el primer sumando, resultando un valor que permanecerá almacenado en el acumulador.

A continuación (instrucción $06: BCS $0B) se ejecuta el salto condicional si C es 1. En caso de que haya ocurrido acarreo en la suma anterior (desbordamiento) se salta a la última instrucción. En caso contrario continúa con la instrucción DBZ.

En la instrucción $07 se ejecuta DBZ $AA. En ella, se procede a decrementar en una unidad el contenido de la dirección de memoria $AA y saltar si el resultado ha sido cero. Con ello, tras la primera pasada su contenido será $FE; en las siguientes pasadas será $FD, después $FC, y así sucesivamente. En nuestro caso, como el valor inicial de M($AA) es $FF y obviamente el valor de n es mucho menor que $FF, nunca va a ocurrir su llegada al valor 0 (ver párrafo siguiente). Pero cada vez que se ejecuta esta instrucción la palabra contenida en $AA apuntará al sumando siguiente.

También se realiza la operación DBZ en la instrucción siguiente ($08: DBZ $AB). En este caso decrementa el contenido de la memoria en su dirección $AB, que inicialmente contiene el número de sumandos que se pretenden sumar, n (para que tenga sentido, el número de sumandos deberá ser siempre muy inferior al tamaño de la memoria: n<<$FF). Con cada ejecución, el contenido de $AB va disminuyendo:

- Mientras queden sumandos por añadir, el valor del decremento es distinto de cero, por lo que Z=0 y se ejecuta la siguiente instrucción ($09), que es JMP $05 produciendo un salto incondicional que cierra el lazo. Así, se irá recorriendo n veces el ciclo de sumas.
- Tras sumar los n sumandos, el contenido de M($AB) tras el decremento será cero (Z=1) por lo que, en ese caso, se saltará a la instrucción que hay en en $0A. Aquí se rompe el lazo.

La instrucción $0A es STA $AC, instrucción en la cual se almacena el contenido del registro acumulador, que contiene el resultado de las n sumas, en la posición de memoria elegida arbitrariamente para el resultado ($AC).

A continuación se ejecuta la siguiente instrucción, que está en $0B; a ella se llega también si ha habido desbordamiento (C=1 en BCS $0B). Se trata de la de parada, STOP, cuya misión es únicamente colocar al computador en estado de espera de un nuevo programa a ejecutar.

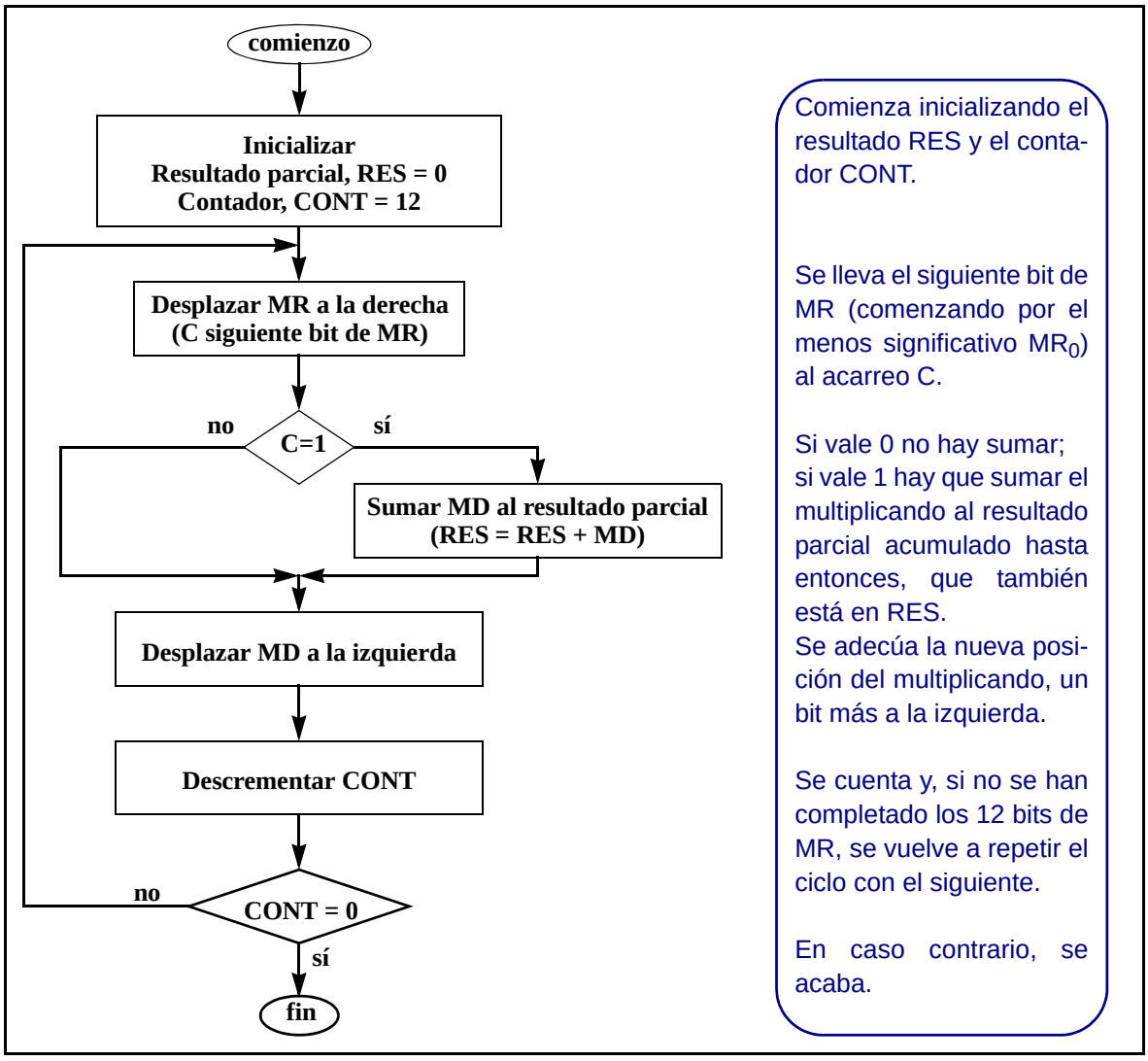
3.3.4.2 Ejemplo II: Multiplicación

En este ejemplo, utilizando el CS2, se va a desarrollar un programa capaz de llevar a cabo la multiplicación de dos números, *multiplicando (*MD) y *multiplicador* (MR). Suponemos que ambos factores se encuentran almacenados en sendas palabras de memoria, a las que nos referiremos como $MD y $MR, respectivamente[1]. El resultado de la multiplicación aparecerá en la palabra de memoria representada por $RES. La multiplicación se llevará a cabo por el algoritmo tradicional de **sumas y desplazamientos a la izquierda** (véase Anexo II). Como en este algoritmo hay que chequear todos los bits de

1. Es obvio que, por ganar simplicidad, se ha renunciando al rigor. Así, en sentido estricto, es erróneo aceptar $MD o $MR ya que $ significa un valor hexadecimal y ni MD ni MR son valores hexadecimales. Aclaremos con más rigor el manejo de los datos, variables y posiciones de memoria. En particular, suponga que se desea multiplicar un multiplicando de valor $6_{(10}$ y un multiplicador de valor $11_{(10}$. Supongamos que las palabras de memoria elegidas para almacenar estos factores son la $90 y $91. El dato $6_{(10}$ es el valor almacenado en la posición de memoria $90: M($90) = $006 (análogamente M($91)=$00B ya que $11_{(10}$=$B). En el texto de esta sección usamos MD para referirnos a 6 (MR para $11_{(10)}$ y $MD para referirnos a $90 ($MR para $91).
Cuando en la práctica normal se use el ensamblador del CS2 (véase Anexo I), se manejarán variables asociadas a direcciones mediante la directiva EQU. Así se definirá, p. ej., MD EQU $90 lo que hará que MD sea $90 tomando, por consiguiente, el sentido que en esta sección se da a $MD.

MR, se usará una variable (a la que denominaremos CONT) que hará la funciones de un contador. Teniendo en cuenta que los datos en CS2 son de 12 bits, la palabra $CONT se inicializará a 12.

Con el único fin de evitar un programa demasiado extenso y, por tanto poco didáctico, una restricción que se impone en la solución que se presenta en esta sección es que el resultado tenga un número de bits menor o igual a 12, que es el tamaño de los buses y registros de datos del sistema. Esta rectricción permite no tener en cuenta si se produce desbordamiento en las operaciones y, así, poder centrarse en la resolución de la multiplicación mediante sumas y desplazamientos a la izquierda. Este algoritmo puede representarse mediante el siguiente ordinograma:.



Ordinograma de la Mulplicación mediante sumas y desplazamientos a la izquierda

El resultado final del producto así como las sumas intermedias de este programa aparecerán en l$RES, donde además se irán almacenado . El listado del programa, asociado a la palabra de la memoria que contiene a cada instrucción, se puede ver en la Fig. 3.22. Con este ejemplo se ve la utilización de la capacidad de desplazamiento del registro acumulador, así como el uso del acarreo C, y su biestable asociado BC, en estos desplazamientos.

Direccion RAM	Contenido de la palabra
$00	LAIM 0
$01	STA $RES
$02	LAIM $C
$03	STA $CONT
$04	LDA $MR
$05	ROR
$06	STA $MR
$07	BCS $9
$08	JMP $C
$09	LDA $MD
$0A	ADD $RES
$0B	STA $RES
$0C	LDA $MD
$0D	ROL
$0E	STA $MD
$0F	DBZ $CONT
$10	JMP $4
$11	STOP
$12	...
...	
$CONT	De $12_{(10}$ hasta 0
$MD	Multiplicando
$MR	Multiplicador
$RES	Sumas parciales y suma final

Figura 3.22: Programa de Mulplicación

COMENTARIOS:

Este programa comienza "vaciando" la posición de memoria, representada por $RES, en la cual se irán almacenando las sumas sucesivas y, al final del proceso, el resultado. Para ello se borra el registro acumulador (LAIM 0) y se almacena éste en la posición $RES (STA $RES), instrucciones que están en las palabras $00 y $01 de la memoria. Con las dos instrucciones siguientes, $02 y $03, se inicializa la palabra $CONT a $C (12 en decimal) . Esta palabra contará el número de desplazamientos para barrer los 12 bits necesarios.

En la instrucción $04, se carga el contenido de la posición de memoria representada por $MR, en donde se encuentra almacenado el multiplicador, en el registro acumulador, instrucción LDA $MR. A continuación, ROR, instrucción $05, se desplaza a la derecha el contenido del registro AC, almacenando el bit menos significativo en el biestable BC. Por último, se almacena en la memoria, en la dirección $MR, el multiplicador ya desplazado ($06 STA $MR).

Con la instrucción BCS $9, que está en $07, se logra la bifurcación según el valor del acarreo C: si C=0 el sistema ejecuta la instrucción siguiente (JMP $C, que es un salto incondicional a la instrucción situada en $0C), mientras que si C=1 sigue por la instrucción escrita en la palabra $09, que es LDA $MD. En este último caso (C=1), tras llevar el multiplicando al acumulador, se suma con el resultado parcial ($0A ADD $RES) y se actualiza el resultado ($0B STA $RES).

A partir de ahora, vuelve a ser común el camino previamente bifurcado.

La palabra $0C contiene una instrucción de carga del multiplicando (LDA $MD), con la que se prepara el desplazamiento a la izquierda del mismo. Este desplazamiento requiere que le entre un 0 por el bit menos significativo. Para ello el bit de acarreo debe ser 0. Sin embargo no hay que incluir ninguna instrucción CLC porque, tal como está construido el CS2, la instrucción LDA siempre borra el bit de acarreo. A continuación de esto, instrucción $0D, se desplaza a la izquierda el contenido del acumulador, operación ROL, y se almacena ese nuevo contenido en la dirección de memoria $MD ($0E STA $MD).

En la instrucción $0F, DBZ $CONT, se decrementa el contenido de la dirección de memoria $CONT, que al comienzo de la ejecución se ha colocado a 12, y se pregunta por el valor de la variable Z, que está conectada al registro RT (Fig. 3.18). Si Z=0, quiere decir que el contenido de RT no es nulo y, por tanto, que aún no se ha terminado la multiplicación. Esto hace que se ejecute la instrucción siguiente $10, JMP $4, lo que hace que se vuelva a la instrucción $04 del programa, para repetir el proceso. Por el contrario si Z=1, quiere decir que ya se han analizado los doce bits del multiplicador y que el proceso se ha terminado. Esto lleva a saltarse la instrucción $10 y pasar a la siguiente, instrucción $11, que será la que finalice el proceso, operación de parada STOP.

3.3.4.3 Ejemplo III: Suma de productos

Con este tercer ejemplo pretendemos mostrar la utilización de subrutinas en el CS2. Para ello vamos a desarrollar un programa en el cual se va a obtener la suma de n sumandos, cada uno de ellos resultado del producto de dos operandos: $S = A_1 \times B_1 + A_2 \times B_2 + A_3 \times B_3 + ... + A_n \times B_n$.

La estructura general del programa será la de calcular el producto de cada una de estas parejas, solicitando el concurso del programa de multiplicación desarrollado en el Ejemplo II anterior, como subrutina de este nuevo ejemplo. Concretemos un poco más las especificaciones del problema a resolver:

- Los valores de los datos a multiplicar (A_i y B_i , para i = 1, 2, ...) están ya almacenados de forma correlativa descendente a partir de una cierta posición de memoria[1], y colocados en parejas. El valor concreto de esa posición de memoria, que en este caso asumiremos que es $EF, será escrita en el programa en una palabra que denominamos $POS. Así, suponemos que A_1 está en $EF, B_1 en $EE, A_2 está en $ED, B_2 en $EC, A_3 está en $EB, B_3 en $EA, etc. En el programa deberemos escribir M($POS) = $EF. De esta forma, si el conjunto de datos estuviera en otra posición, bastaría sustituir $EF por la dirección de esa *otra posición*.
- El valor de cada producto no produce desbordamiento (para usar la misma rutina de multiplicación antes descrita).
- El valor de las sumas parciales y de la suma final total se almacenará en una posición de memoria denominada $TOT. Por simplicidad, tampoco tendrá desbordamiento la suma. (*Ejercicio para el lector: Cuando usted tenga un poco de experiencia programando con el CS2, reforme la solución de este problema para que incorpore desbordamiento en la suma y en la multiplicación.*)
- El número n de sumandos se escribirá en una palabra, $NP. En nuestro caso escribiremos el programa para n = 8

La idea general del programa consiste en el siguiente algoritmo:

1. Inicialización de $NP (a 8), $POS (a $EF) y $TOT (a 0, ya que acumulará las sumas)
2. Comenzando desde M($EF)=$A_1$ y M($EE)=$B_1$, almacenar A_i y B_i en las posiciones de memoria $MD y $MR.
3. Llamar a la subrutina de multiplicación. En $RES aparecerá el producto $A_i \times B_i$.
4. Sumar $RES con $TOT y acumular esta suma en $TOT
5. Decrementar el número de sumandos que queda por sumar ($NP):
   * Si no es cero, volver al punto 2 para una nueva pareja A_i y B_i
   * Si es cero, acabar

La memoria quedará con la siguiente partición:

- Primeras posiciones ($00, $01, etc.): programa principal
- A continuación, subrutina de multiplicación. Esta debe acabar con la instrucción RTS.

1. Recuérdese que las subrutinas usan una pila cuyo primer dato se escribe en $FF y los siguientes, si los hubiere, en las posiciones contiguas decrecientes. Por ello la zona baja de memoria debe reservarse a las operaciones de subrutina.

- Zona de variables: $NP, $POS y $TOT para el programa principal; y $MD, $MR, $CONT y $RES para la subrutina de multiplicación.
- Zona de datos iniciales: como son 2x8=16 datos, estarán desde $E0 a $EF
- Zona de subrutina (pila de direcciones de retorno): final de memoria ($FF, $FE, etc.)

El contenido de la memoria completo se puede ver en la Fig. 3.23.

COMENTARIOS:

Comencemos por el Programa Principal. Las seis primeras instrucciones de este programa tienen por misión inicializar las posiciones de memoria indicadas en el paso 1 del procedimiento, antes de comenzar el proceso iterativo de sumas de productos. Así, $00 y $01 introducen un 8 en la posición de memoria $NP, el número de parejas a sumar en el proceso. Las dos instrucciones siguientes, $02 y $03, hacen que en la posición de memoria $POS se introduzca la posición de memoria más alta que con-

Direccion RAM	Contenido de la palabra
$00	LAIM 8
$01	STA $NP
$02	LAIM $EF
$03	STA $POS
$04	LAIM 0
$05	STA $TOT
$06	LAIM 0
$07	ADDI $POS
$08	STA $MD
$09	DBZ $POS
$0A	LAIM 0
$0B	ADDI $POS
$0C	STA $MR
$0D	DBZ $POS
$0E	JSR $15
$0F	LDA $RES
$10	ADD $TOT
$11	STA $TOT
$12	DBZ $NP
$13	JMP $6
$14	STOP

(a)

Direccion RAM	Contenido de la palabra
$15	LAIM 0
$16	STA $RES
$17	LAIM $C
$18	STA $CONT
$19	LDA $MR
$1A	ROR
$1B	STA $MR
$1C	**BCS $1E**
$1D	**JMP $21**
$1E	LDA $MD
$1F	ADD $RES
$20	STA $RES
$21	LDA $MD
$22	ROL
$23	STA $MD
$24	DBZ $CONT
$25	**JMP $19**
$26	**RTS**

(b)

Direccion RAM	Contenido de la palabra
$NP	Nº sumandos (8)
$POS	Posición mayor de los datos ($EF)
$TOT	Suma parcial y total
$CONT	De $12_{(10}$ hasta 0
$MD	Multiplicando
$MR	Multiplicador
$RES	Sumas parciales y suma final
...	...
($E0)	Zona de Datos A_i y B_i
...	
$EE	
$EF	
....	Retornos de subrutinas
$FE	
$FF	

(c)

Figura 3.23: Visión de la memoria para el Ejemplo III Suma de Productos: a)Programa principal; b)subrutina de multiplicación; c)zona de variables, datos y direcciones de retorno de subrutina

tenga al primer operando (A_1). Las instrucciones \$04 y \$05 ponen a 0 la posición de memoria \$TOT.

A partir de la instrucción \$06 entramos en el bucle iterativo, que corresponde al paso 2 del algoritmo. En primer lugar se carga en el acumulador el dato apuntado por \$POS, que la primera vez es A_1. Esta carga se realiza, tras borrar el acumulador, con la suma indirecta con dirección \$POS: la primera vez, como el contenido de M(\$POS) es \$EF, la instrucción ADDI \$POS hace que se sume el acumulador (que acaba de ser hecho 0) con M(\$EF)=A_1. Después, el acumulador se escribe en \$MD (ahora vale, pues, A_1) y, por último, con DBZ se decrementa \$POS (M(\$POS) pasa a ser \$EE) con lo que \$POS pasa a apuntar al siguiente dato, B_1 (que se encuentra en \$EE). Las siguientes 4 instrucciones (desde la \$0a hasta la \$0D) hacen lo mismo, esta vez metiendo B_1 en \$MR. Obsérvese que, aunque la instrucción DBZ además de decrementar \$POS, consulta el nuevo valor alcanzado para bifurcarse según sea nulo o no, en este caso nunca va a ocurrir que se alcance el 0 ya que sólo avanzaremos hasta \$E0 y, por tanto, la misión de DBZ aquí consiste únicamente en obtener ese decremento de \$POS.

El paso 3 se realiza con la instrucción \$0E JSR \$15. Nótese que para poder conocer cuál es la dirección donde empieza la subrutina, en este caso la \$15, es necesario haber terminado de escribir todo el programa principal.

El paso 4 se realiza en las siguientes 3 instrucciones. En la instrucción \$0F, que es a la que se regresa tras ejecutar RTS en la subrutina, se carga en el acumulador el resultado obtenido por la subrutina, y que se encuentra en la dirección \$RES. En la \$10 se suma con el valor de suma previo y en la \$11 se actualiza dicho valor de suma.

Por último, las instrucciones \$12 a \$ 14 implementan el paso 5: decrementa el número de casos a sumar volviendo a iniciar el ciclo si todavía quedan o acabando si ya se han completado todas las sumas.

La subrutina de multiplicacioón ocupa desde \$15 hasta \$26. Es el mismo programa de la Fig. 3.22con las siguientes salvedades: 1/la instrucción final es RTS en vez de STOP; y 2/se han modificado las direcciones de los saltos para adaptarlas a la nueva ubicación de la rutina, lo que afecta a las instrucciones \$1C, \$1D y \$25 en la Fig. 3.23.

3.4 CONCEPTO DE COMPUTADOR

El sistema digital nominado CS2, desarrollado en el Apartado 3.3 y cuya organización se muestra en la Fig. 3.18, opera ya de una forma muy próxima a los computadores. Hay, sin embargo, ciertos aspectos que los separan, por lo que aún debemos tender puentes entre uno y otros.

Uno de esos aspectos radica en la perspectiva desde la que se los trata. Así, mientras que nosotros hemos afrontado los computadores simples como un problema de diseño de sistema digital a nivel RT, usualmente el estudio de los computadores se realiza desde la perspectiva del usuario (no del diseñador) y se hace a nivel ISP.

Por otra parte, además, un mínimo de la realidad de los computadores debe estar presente en nuestra descripción actual, al menos en sus cuestiones más fundamentales. De ellas hay dos que todavía hemos de incorporar: 1º) el dimensionamiento relativo entre número de palabras y número de bits por palabra de la memoria; y 2º) la interacción del sistema digital con el mundo exterior.

El propósito de este apartado es, por tanto, tender el citado puente entre CS2 y los computadores, lo que haremos centrándonos en su vertiente más conceptual y en su incidencia sobre el sistema digital

ya desarrollado (CS2). Debido tanto a la complejidad como a la diversidad y a la importancia de los computadores, la presentación que realizamos en este apartado es apenas una breve introducción con la que esperamos queden sentadas las bases para comprender los computadores. En todo caso, se deberá completar con un estudio mucho más exhaustivo de computadores reales.

La organización de este apartado es como sigue. Inicialmente presentamos la terminología y arquitectura estándar de los computadores, conectándolo con los sistemas digitales desarrollados hasta ahora. Después, resolvemos la ejecución de instrucciones multipalabras, tipo de instrucción cuya necesidad surge del dimensionamiento usual en la memoria. Por último, prestaremos atención a la interacción con el mundo exterior, para lo que habrá que determinar el mecanismo a través del que se establece el diálogo y posibilitar que el sistema encadene la ejecución de procesos distintos.

3.4.1 Organización básica

Los sistemas digitales que hemos diseñado están subdivididos en "unidad de datos" y "unidad de control". Buena prueba de ello es CS2, cuya estructura se muestra en la Fig. 3.18c. Si atendemos a esta estructura observamos que hay diferentes funciones asociadas a sus componentes, por lo que es fácil particionarla. La forma más simple es dividirla en dos:

- de un lado quedaría la memoria, cuya función es la de almacenar la información, tanto la que representa datos como la que contiene las instrucciones (programas). Obviamente, además de almacenar la información, en la memoria se podrá escribir/leer la información almacenada
- del otro, quedaría "todo lo demás" cuya funcionalidad es doble, procesar la información y controlar la operación del sistema total. A esta parte se le conoce con el nombre de Unidad Central de Proceso, CPU (*Central Process Unit*), o procesador.

Esta división a nivel funcional se realiza también muy fácilmente a nivel estructural: las dos unidades, CPU y memoria, quedarían conectadas a través de los buses de dirección y de datos, junto con algunas señales de control. La Fig. 3.24 muestra a la izquierda un boceto de la Fig. 3.18 que es agrupada según esa división, mostrando a la derecha la nueva visión en CPU, y Memoria.

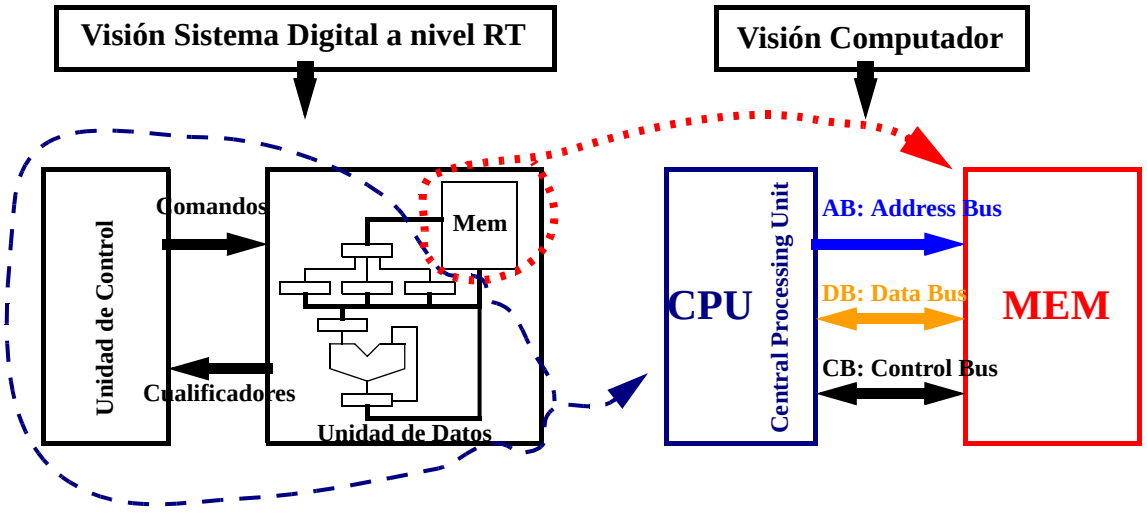


Figura 3.24: Transición de la estructura Datos&Control a la organización CPU&Memoria

Hasta ahora el CS2, al igual que CS1, es un sistema sin conexión con el exterior, cerrado en sí mismo, lo que lo diferencia de un computador real. En éste se establece interacción con el exterior. Esta interacción consiste en transferir información, la cual puede ser en cualquiera de las dos direcciones, desde o hacia el computador. Para llevar a cabo estas transferencias en los computadores existe la llamada unidad de entrada/salida (I/O, Input-Output). Esta unidad está formada, en general, por varios dispositivos de entrada o de salida que se comunican con las unidades anteriores a través de buses similares a los ya utilizados.

Un computador está formado básicamente por estas tres unidades funcionales, interconectadas entre sí a través de buses (conteniendo líneas de direcciones, de datos y de control) tal como se muestra en la Fig. 3.25. El bus dibujado a trazos corresponde al Acceso Directo a Memoria (DMA, *Direct Memory Access*), que no siempre está incluido, pero que con el cual es posible transferir datos entre el mundo exterior y la memoria directamente. Veamos brevemente cada una de estas unidades funcionales.

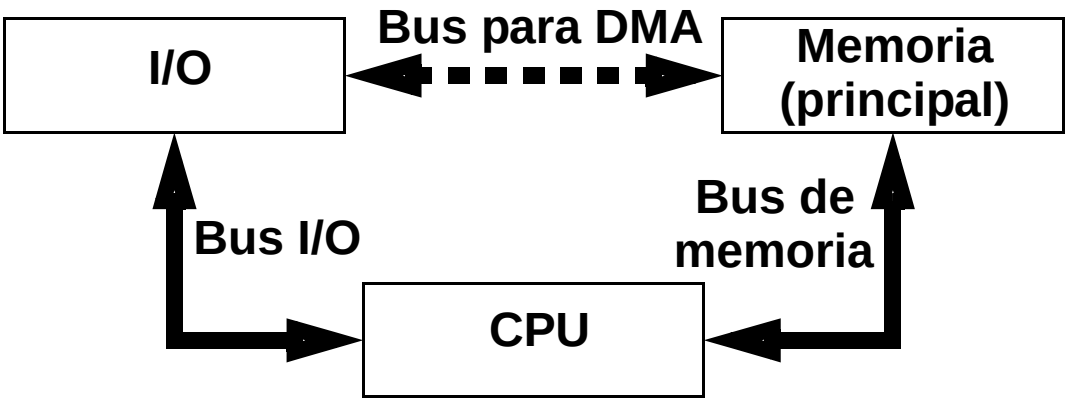


Figura 3.25: Organización básica de un computador.

Unidad Central de Proceso (CPU)

Como hemos indicado, la función del procesador, o CPU, es doble: procesar la información y controlar el sistema. Esto incluye, entre otras tareas, buscar el programa, ejecutarlo de acuerdo con el flujo adecuado, interpretar las instrucciones, manipular los datos y emitir/recibir las señales que gobiernen las interacciones con las otras unidades funcionales. Claramente, la CPU es el "cerebro" del computador.

Una CPU contiene, por una parte, una subunidad de datos (en el sentido dado en los sistemas digitales) y, por otra, una de control (Fig. 3.26). La subunidad de datos, a su vez, consta básicamente de una ALU, con la que se realizan las operaciones entre los datos (esto es, el *procesado* de datos), unos registros de propósito específico (tales como el contador de programas, el puntero de pila ó el registro de estados o de código de condición del procesador) y otros registros de propósito general (acumulador, registro tampón y, frecuentemente, unos pocos de registros más). Por su parte, en la unidad de control se suele incluir tanto el propio circuito de control como el registro de instrucciones y los decodificadores correspondientes; de aquí que a menudo se denomine "decodificación y control" a esta unidad. Todos estos componentes se conectan mediante los buses adecuados. La mayoría de este hardware es fácilmente reconocible en nuestro CS2 (Fig. 3.18). Una CPU integrada en un único circuito recibe el nombre de microprocesador (μP).

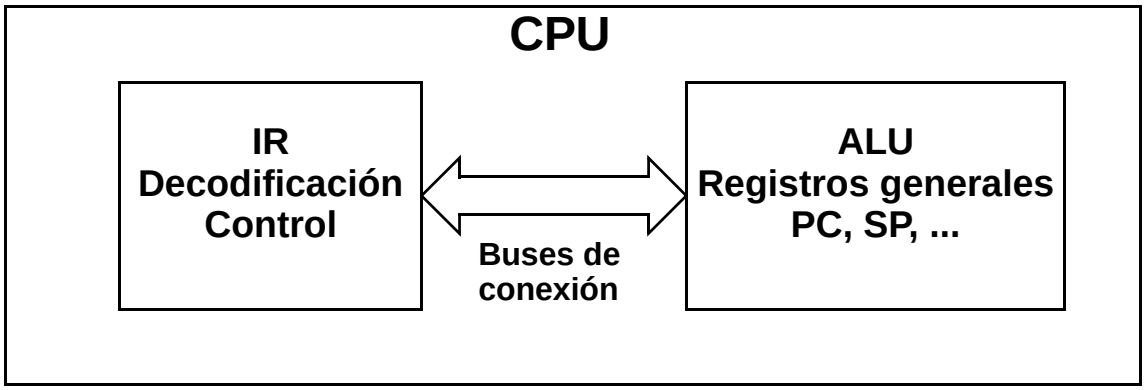
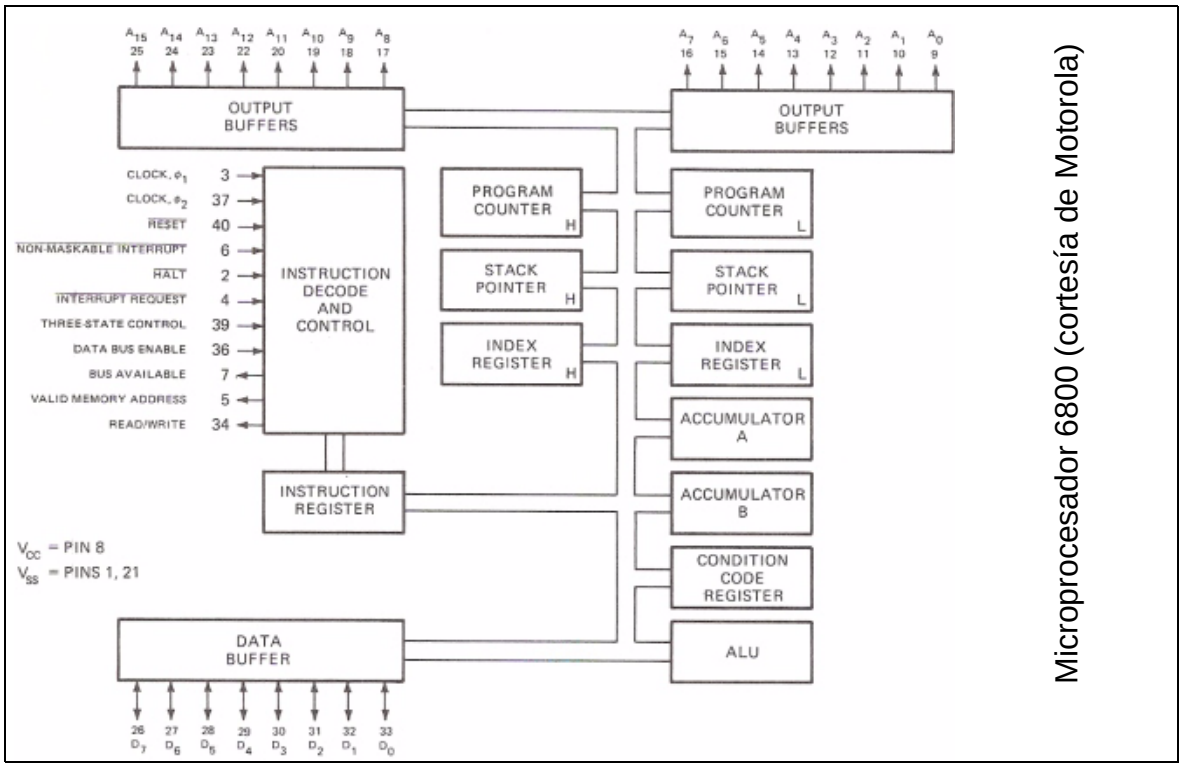


Figura 3.26: Organización de una CPU.

Los fabricantes no suelen dar información sobre el diseño real de la CPU. En su lugar ofrecen el llamado "modelo del usuario" o "modelo de programación". En esta descripción se ofrece, a nivel RT, los componentes que hay que tener en cuenta al utilizar el procesador bien en programación (nivel ISP), bien como "unidad hardware". El modelo de usuario incluye los registros que se ven afectados por el conjunto de instrucciones (p.ej., el acumulador), pero elimina aquéllos que son transparentes al uso (en CS2 sería, p.ej., el de direcciones MAR, de la ecuación 3.18), así como otros detalles concretos como el conexionado, la decodificación o el control. La descripción completa de una CPU incluye el modelo de usuario, el conjunto de instrucciones y diagramas temporales para las señales.

Existe una gran diversidad de procesadores cuya mera clasificación (por familias tecnológicas, por familias comerciales, por prestaciones, por el grado de paralelismo, etc.) cae fuera de los propósitos de esta sección. No obstante, a título de ejemplo, en la Fig. 3.27 se presentan las organizaciones de dos de los procesadores iniciales más básicos: el μP 6800 de Motorola y el μP 8080 de Intel. Comparando nuestro CS2 con estos procesadores, y más particularmente con el más parecido (μP 6800), puede observarse que no hay diferencias sustanciales apreciables. Salvo en lo que atañe a la complejidad (que afecta al conjunto de instrucciones, a la ALU, y a algunos registros), las principales diferencias son las dimensiones relativas entre el bus de direcciones y de datos (16 a 8 en el μP 6800), y la existencia de algunas señales de control a la CPU como son Reset, Interrupt Request, etc. Sobre ambas diferencias volveremos seguidamente.

Hasta aquí hemos presentado lo que se llaman sistemas monoprocesador, que como indica su nombre son aquéllos que sólo tienen una CPU. Antes de abandonar esta breve introducción a las arquitecturas de procesadores, mencionemos la existencia de sistemas multiprocesadores que, obviamente, poseen más de una CPU. Estos sistemas surgen para alcanzar diferentes grados de paralelismo en la ejecución de tareas. La fff ilustra algunas de las posibles arquitecturas de los sistemas multiprocesadores: una con varias CPUs que comparten la memoria principal; otra que representa a los vectores/matrices de procesadores (traducción de *array processor*, término de difícil concreción y con muy discutidas y controvertidas traducciones una de las cuales, procesadores de/en arreglo nos resulta chocante) y, por último, un procesador con *pipeline* (segmentación, escalonamiento, entubamiento,...) en el que las instrucciones tienen diferentes fases de procesado encontrándose varias de ellas ejecutándose simultáneamente: una instrucción en la primera fase, otra en la segunda y así sucesivamente.



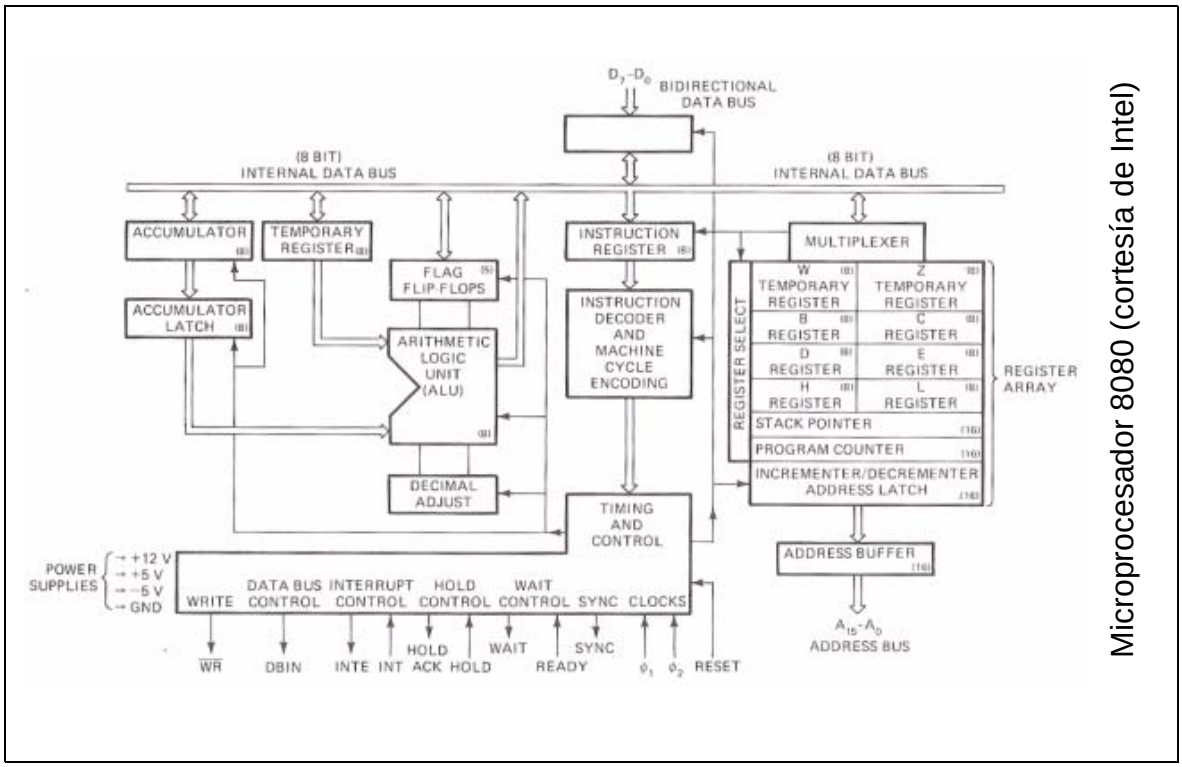


Figura 3.27: Microprocesadores 6800 de Motorola y 8080 de intel

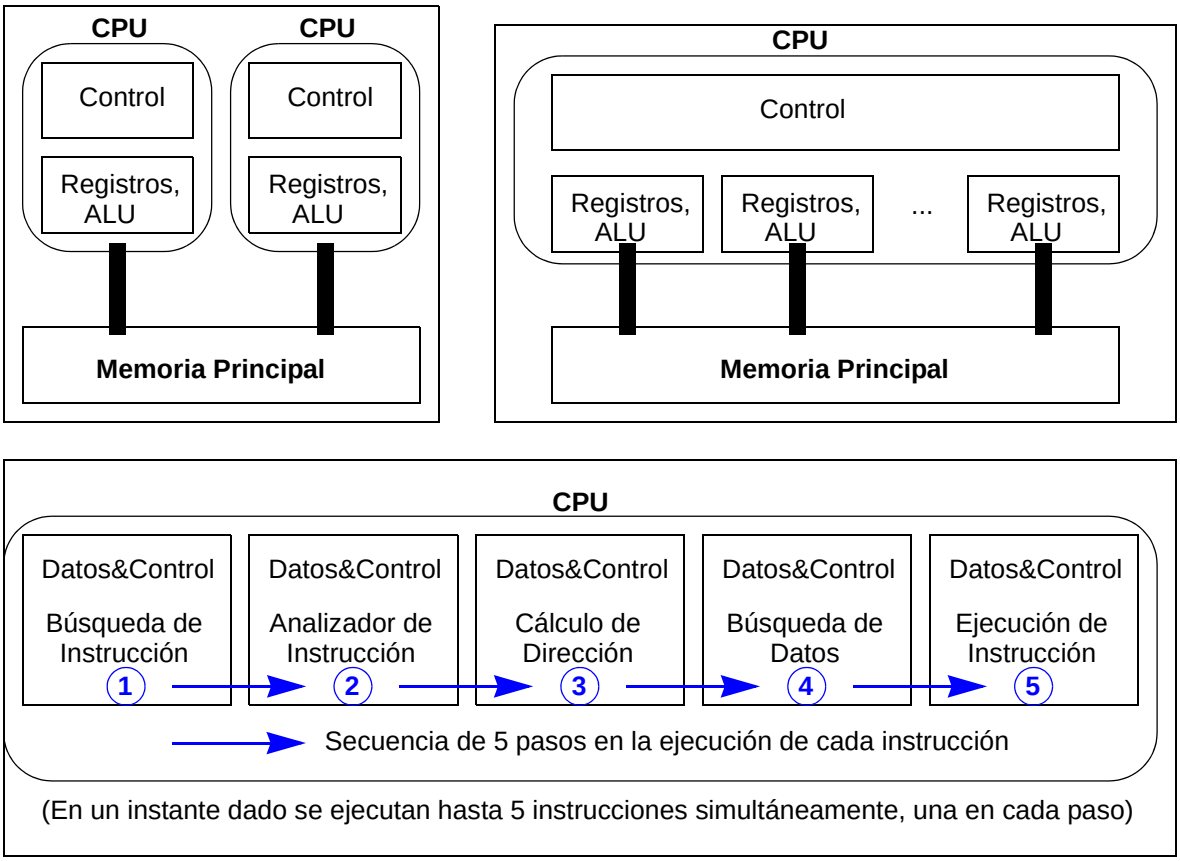


Figura 3.28: Diferentes arquitecturas de sistemas multiprocesadores

Unidad de memoria

La unidad de memoria o memoria principal del computador tiene como función almacenar la información necesaria para la operación del sistema. Esto comprende el almacenamiento de programas y datos, incluyendo los valores de entrada, los resultados intermedios y los de salida. La información contenida en memoria puede ser permanente, o sea, la que es siempre la misma, independientemente del proceso en curso, como ocurre con las rutinas de inicialización o los datos constantes, ó eventual, que es la existente en cada proceso determinado. Las dos operaciones de la memoria son las de lectura y la de escritura de una palabra. Este dato se comunica a la CPU o a los dispositivos externos (a través de la unidad de I/O). Todas sus acciones están controladas por la CPU (salvo excepciones como el acceso directo a memoria).

La memoria principal consiste en una memoria de acceso aleatorio de 2^n x k bits, siendo n el número de líneas de dirección y k el número de líneas de datos. Contiene dispositivos de memoria, normalmente, semiconductora de tipo RAM para almacenar la información eventual y de tipo ROM para la permanente. Se comunica con la CPU a través de los buses de dirección (n bits) y de datos (k bits), junto a las señales de control para la lectura/escritura (R/W, Read-Write). Como unidad del computador su descripción incluye el mapa de memoria sobre el que se especifican las distintas regiones (de usuario para R/W o sólo lectura, de pila, de uso interno, etc.) en el espacio de direccionamiento de la CPU.

Aunque las dimensiones de la memoria (n y k) varían mucho de unos computadores a otros, un hecho general es que el número de líneas de dirección es mayor que el de líneas de datos (n>k); por ejemplo, en computadores basados en el μP 6800 n=16 y k=8, mientras que para el μP 68000,los valores son n=23 y k=16. El acceso del procesador a la memoria principal se hace, en general, palabra a palabra. Así, en el μP 6800 se accede a la palabra de una dirección determinada, p. ej. la $1234, lo que significa el acceso (de lectura o de escritura) a un dato de 8 bits (1 B, B es *Byte*). En el μP 68000 también se accede a la palabra de una dirección determinada, p. ej. la $001234 (los 0's son innecesarios, pero se han puesto para resaltar que el bus de direcciones tiene 23 bits), accediéndose a un dato de 16 bits (que son 2B y, en el lenguaje del μP 68000 se denominan 1W, W es *Word*). A veces hay formas de acceso más complejas: p. ej. en el μP 68000 se puede acceder tanto a la mitad de 1W, esto es a 1 B indivudualmente, como a una doble W (4B consecutivos que se denomina 1L, L es *Long word*).

El hecho de que haya más líneas de direcciones que de datos (n>k) tiene una importante consecuencia en el diseño de un procesador: el número de bits del código de instrucción es mayor que el número de bits k de la palabra de memoria, ya que en general deberá contener los n bits de direcciones más los bits necesarios para el código de operación. De aquí que, en general, se necesiten varias palabras para almacenar una instrucción La manipulación de instrucciones multipalabra será comentada en la siguiente sección (Apartado 3.4.2).

Unidad de Entrada-Salida (I/O)

En un sentido general, la unidad de entrada-salida tiene como función la interacción del computador con el mundo exterior. Esta interacción incluye acciones de comunicación, observación y control. La unidad I/O puede ser subdividida en subsistemas de entrada (el mundo exterior transfiere información hacia el computador), de salida (el computador les suministra la información) y de entrada-salida (la información fluye en ambas direcciones).

Los subsistemas que contiene la unidad de I/O se denominan comúnmente dispositivos periféricos o simplemente periféricos. Existe una variada gama de periféricos que incluyen: monitores o pantallas de visualización, teclados, ratones, impresoras, dispositivos para comunicación, sensores, actuadores, unidades de memoria secundaria, etc. Cada uno de estos grupos, a su vez presenta una gran diversidad. Por ejemplo, consideremos las unidades de memoria secundaria, que son las que proporcionan el almacenamiento masivo de la información. Las principales diferencias entre la memoria masiva y la memoria principal es que en ésta el acceso es aleatorio (en aquélla es secuencial) y su dimensionamiento es compatible con la CPU (mientras que en las memorias masivas la organización de datos es prácticamente independiente de la CPU). Pues bien, existe una gran casuística de sistemas de memoria: discos flexibles, discos duros removibles o no, CD's (Compact Disk), cintas, etc. También existe una gran diversificación entre otros muchos aspectos de los periféricos de un ordenador: Así, la transmisión de datos con algunos debe ser paralela, mientras que en otros es transmisión serie; en unos los datos se envían síncronizados y en otros de forma asíncrona; en unos los datos son digitales mientras que en otros son analógicos (vídeo, audio); los mecanismos de conexión son múltiples; etc.

En resumen, los periféricos constituyen una enorme diversidad de dispositivos con características muy diferenciadas entre sí. Los principales problemas de la conexión entre la CPU y los periféricos, que provienen de la diversidad de éstos entre sí y con la CPU, pueden centrarse en el acoplamiento de: la anchura de la palabra de información, la velocidad de operación y el tipo de soporte de la información. Para la solución de estos problemas es necesario incluir circuitos de interfaz entre la CPU y sus periféricos (Fig. 3.29). Estos circuitos de entrada-salida forman propiamente la unidad I/O de un computador

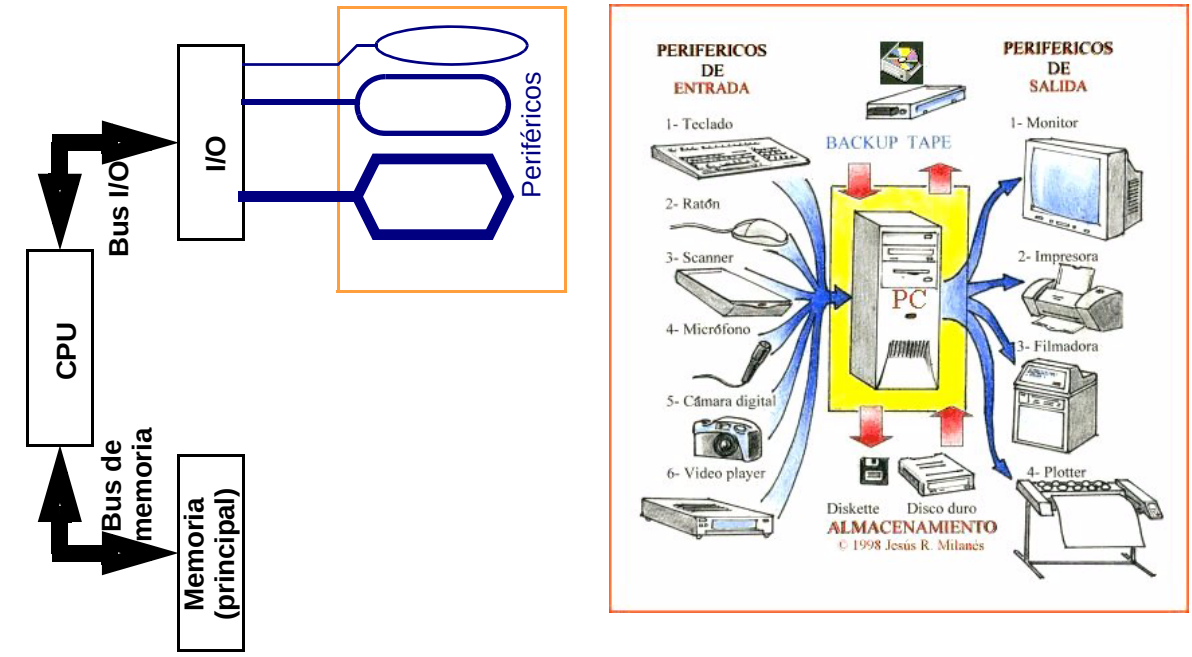


Figura 3.29: Conexiones de periféricos a través de circuitos I/O

y es sobre ellos sobre los que se conectan los periféricos.

Los circuitos de entrada/salida ejecutan dos tareas básicas de acoplo entre CPU y periféricos: la transmisión de la información y el control del diálogo. La transmisión de información incluye acciones múltiples como el reconocimiento del periférico, el almacenamiento temporal de datos, el chequeo de errores en la transmisión, la conversión de datos, etc. Por su parte, el control del diálogo debe resolver el problema del requerimiento del servicio (cómo lo pide y cómo lo reconoce) y el de la sincronización (adecuación de la temporización entre CPU y periféricos).

El estudio detallado de los periféricos, circuitos de interfaz y mecanismos de interacción cae obviamente fuera de esta introducción que aquí hacemos. Afrontaremos, no obstante, una cuestión básica: qué implicación tiene sobre la operación de la CPU la inclusión de la unidad I/O. En particular, al haber muchos de periféricos, uno de ellos que quiera servirse de la CPU deberá hacerle llegar una señal de petición y, una vez que la CPU pueda otorgarle el servicio, se establecerá el intercambio de información. Desde la perspectiva de la CPU esto significa, por una parte, que debe estar dotada de unas señales de control destinadas a las peticiones y, por otra, que el proceso que ejecute actualmente pueda interrumpirse para dejar paso a la ejecución del proceso que preste el servicio requerido.

3.4.2 Instrucciones multipalabra

Los computadores reales deben ser capaces de manejar instrucciones multipalabras. Éstas son aquéllas cuyo código binario ocupa más de una palabra de memoria. Por ejemplo, el µP 6800 (Fig. 3.27) posee un juego de instrucciones en el que existen instrucciones:

- De una sola palabra (1 Byte): corresponden a instrucciones sin operandos, tales como la no-operación ó el retorno de subrutina, o con operandos especiales (acumuladores, registro

de estado, etc.), como son la suma de ambos acumuladores o la puesta a cero del bit de acarreo. En terminología de Motorola, se les denomina instrucciones con direccionamiento implicado.

- De dos palabras (2 Bytes): corresponden a operaciones con una dirección que puede darse como un único Byte. En este microprocesador esto ocurre con todos los modos de direccionamiento salvo el implicado y el extendido. Así, por ejemplo, la carga de una constante en el acumulador o las ramificaciones (*branch*) son instrucciones de dos Bytes: el primero indica el código de operación y el segundo indica la dirección. En memoria, esta instrucción está almacenada en las posiciones N y N+1.
- De tres palabras (3 Bytes): se utiliza en instrucciones con direccionamiento extendido donde el primer Byte contiene el código de operación, el segundo Byte corresponde a los ocho MSB's[1] de la dirección (16 bits) del operando y el tercer Byte corresponde a los ocho LSB's de dicha dirección. En memoria ocuparían tres posiciones consecutivas, N, N+1 y N+2, respectivamente.

La inclusión de instrucciones multipalabras afecta al ciclo de búsqueda que ahora, a diferencia del correspondiente a CS2, tiene que tener en cuenta la posibilidad de que la instrucción sea de una, dos o tres palabras. Para ello, la carta ASM del ciclo de búsqueda debe ser como la representada en la Fig. 3.30. Al comienzo del ciclo de búsqueda (*Fetch cycle*) se ejecuta la caja de estados S1, con la acción denominada ASM-F1, que coincide con el ciclo de búsqueda de CS2: se lee la palabra de memoria almacenada en la posición apuntada por el contador de programas PC, se almacena en el registro de instrucciones IR, y se incrementa PC. Una vez almacenado el código de operación en IR (caja de estado S2) se decodifica para ver si la instrucción es de una, dos o tres palabras. Nótese que el estado S_I es necesario para que la decodificación sea la leida en este ciclo de búsqueda (S_I garantiza que en IR está el valor leído de la memoria). Si la instrucción es de una palabra, se ejecuta. En caso contrario, se busca en memoria la segunda de las palabras, lo cual se ejecuta mediante la acción condicional correspondiente, ASM-F2. En ella, de nuevo, se leerá la palabra apuntada por PC, se almacenará en el registro correspondiente de la CPU (depende de la operación a ejecutar), se incrementará el PC y se pasará a la siguiente caja de estados (S3). En esta se chequea si la instrucción es de 2 palabras, en cuyo caso el control saltará al ciclo de ejecución, o por el contrario es de tres palabras, en cuyo caso se ejecuta ASM-F3 (de forma similar a ASM-F2): se busca la palabra apuntada por PC almacenándola en el registro correspondiente de la CPU e incrementando PC. Seguidamente, se ejecuta la instrucción.

Como ejemplo veremos una instrucción del μP 6800 que tiene tres palabras, con el fin de detallaremos un posible comportamiento del μP. Se trata de la instrucción LDA (*LoaD Accumulator*), que carga en el acumulador (A) el dato almacenado en la palabra de memoria que se indica con direccionamiento directo en la propia instrucción. En concreto, consideremos que el dato a cargar se encuentra en $300:

Dirección de memoria	(HEX)	Mnemónico	Descripción
$N	B6		Cargar el acumulador "A"
$N+1	03	LDA($0300)	con el dato almacenado
$N+2	00		en la dirección $0300

En un momento determinado de la ejecución del programa, en el registro PC se encontrará almacenada la dirección $N. El controlador del sistema ejecuta ASM-F1, con lo que buscará y traerá su contenido $B6 al registro de instrucciones. Este contenido indicará el código de la operación a realizar

1. MSB: *Most Significant Bit*, bit más significativo; LSB: *Least Significant Bit*, bit menos significativo

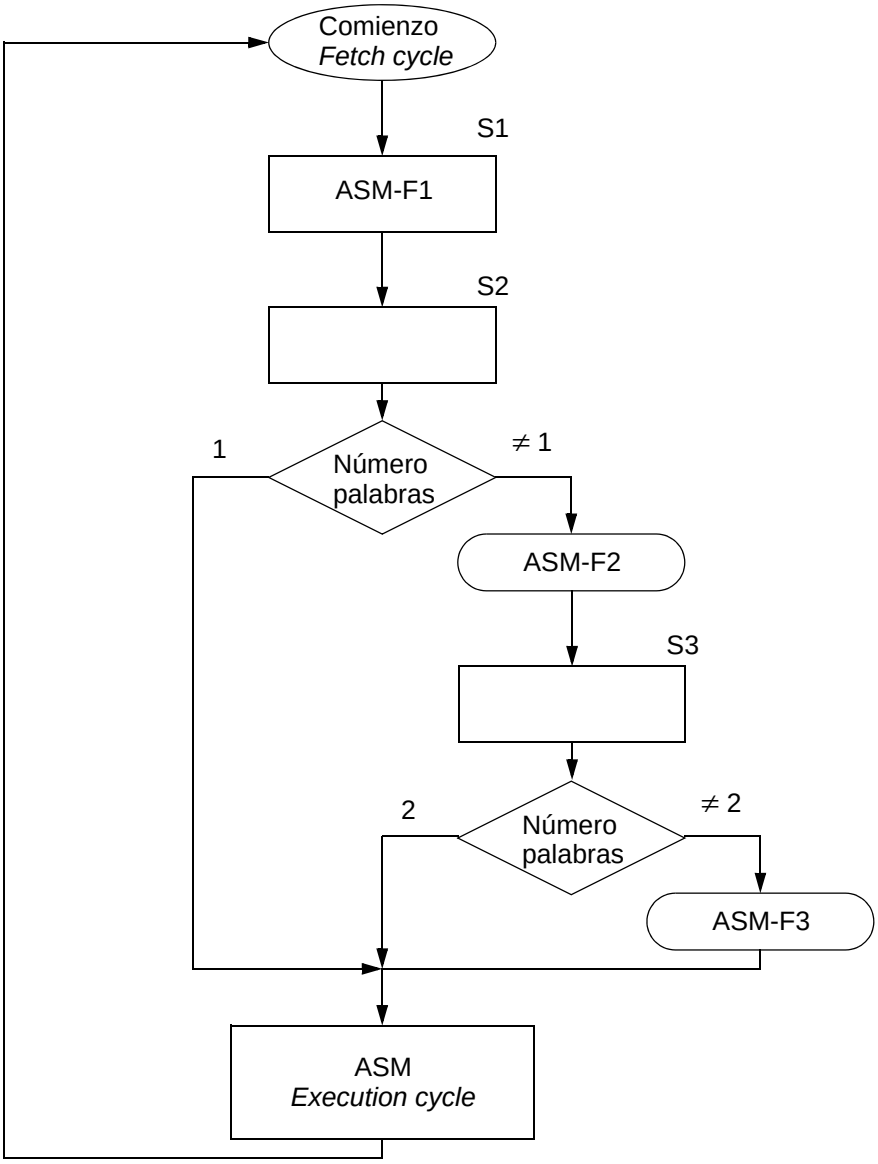


Figura 3.30: Carta ASM para el ciclo de búsqueda (*Fetch cycle*) en procesadores con instrucciones de 1, 2 o 3 palabras.

(LDA) y, además, que la instrucción posee otras dos palabras. En consecuencia, el controlador, pasa a ejecutar la segunda de ellas (ASM-F2 en el esatdo S2). Por ello, el controlador procederá a una nueva lectura (el registro PC ahora apunta a la dirección $(N+1)), almacenando su contenido ($03) como 8 MSBs de la dirección del dato. A continuación, la unidad de control vuelve a incrementar el contenido del registro PC (dirección $(N+2)) y entra en el esatdo S3. En él verifica que esta instrucción es de 3 palabras y vuelve a buscar en la memoria los ocho LSBs de la dirección de memoria del dato ($00). Esto completa el ciclo de búsqueda de la instrucción. A continuación el controlador ejecutará dicha operación(LDA) buscando el dato en la dirección de memoria $0300. Para ello pone en el registro MAR los 8 MSB's y los 8 LSB's almacenados en la CPU durente ASM-F2 y ASM-F3. Tras ello, accede a memoria en modo de lectura, con lo que el dato almacenado (Memoria($0300)) viaja por el bus de datos, de donde lo recoge la CPU almacenándolo en el registro acumulador.

3.4.3 La operación de entrada/salida

Como ya hemos indicado, las operacioens de I/O tienen que resolver, entre otros, el problema de conocer cuándo la CPU debe atender esta operación y, en su caso, cómo presta el servicio en medio de la realización del proceso en curso sin que éste (p. ej. una simulación) se vea afectado por el servicio de I/O. Para controlar las operaciones de I/O, diremos que existen dos métodos básicos: **1/programadas** y **2/controladas por interrupciones**.

- En la primera de ellas todos los pasos relativos a una operación de I/O requieren la ejecución de instrucciones por parte de la CPU, lo cual quiere decir que, en cualquier instante del proceso, la operación de I/O se encuentra bajo el control de un programa. De ahí el nombre de programada.
  Dada la forma de actuar del sistema, la CPU no es capaz de saber en qué momento ha de dar servicio a un dispositivo concreto. Por ello, se ve en la necesidad de ejecutar periódicamente rutinas que "consulten" cada uno de los dispositivos periféricos para constatar su estado. Estos programas o rutinas consumen tiempo, en la mayoría de los casos innecesario, ya que un tanto por ciento muy elevado de las "consultas" serían negativas. Además, los dispositivos ya preparados para enviar/recibir datos tienen que esperar a que les toque el turno para hacer saber a la CPU su disposición. Esto produce una reducción en la velocidad de transferencia de datos. Por todo ello, este método es lento, aunque por otro lado es el más simple y siempre es utilizable en cualquier computador.

- Las deficiencias anteriores se pueden subsanar, en parte, si utilizamos las llamadas I/O **controladas por interrupciones**. En este tipo de control es el propio dispositivo periférico quien avisa a la CPU de su disponibilidad para enviar/recibir datos. Por ello, ya no es necesario que la CPU vaya "consultando" uno a uno todos los dispositivos. Su misión, ahora, consiste en ir ejecutando las tareas que tenga encomendadas (esto es, el proceso actual), interrumpiéndolas cuando vaya a servir al dispositivo que así lo requiera. Sólo entonces pasa a ejecutar la rutina de servicio del periférico. Todo esto lleva a conseguir una mayor velocidad de transferencia de información.

El control de operaciones de I/O controlada por interrupciones ha mostrado ser mucho más eficiente que el método programado[1]. De aquí que detallemos un poco más las bases del control por interrupciones.

El control por interrupciones puede ser bien descrito mediante el caso en que un periférico, a través del circuito de interfaz I/O correspondiente, solicite el servicio de la CPU. El circuito de interfaz activa una señal de control que está conectada a una de las **entradas de interrupción** de la CPU. La CPU, en general, se encontrará ejecutando una instrucción cualquiera del programa correspondiente al proceso en curso. Al activarse su entrada de interrupción, la CPU realizará las siguientes acciones:

1. Interrumpe el proceso de ejecución. Esta interrupción se realiza terminando la ejecución de la instrucción en curso, almacenando todos los registros significativos de la CPU (PC, acumuladores, etc.) y cargando en PC la dirección correspondiente a la atención de interrupciones. Esta dirección es una especificación de la CPU; esto es, se ha previsto a la hora de diseñar la CPU.

1. Nos estamos refiriendo a formas de control realizado por la CPU. Existen otras formas más sofisticadas y eficientes en las que el control de operaciones I/O es realizado por dispositivos inteligentes, tales como el controlador DMA. Estos controladores liberan casi por completo de trabajo I/O a la CPU.

2. Se realiza el proceso de atención al servicio requerido. Se trata de ejecutar un programa especial en que, entre otras tareas, se detecta qué periférico ha solicitado la interrupción lo que, a veces, requiere arbitrar cuál entre varios periféricos va a ser servido. Tras determinar el periférico la CPU ejecuta el programa en particular que servirá para la interacción con el mismo. Estas tareas son dependientes de la configuración concreta dada al computador y, al menos en las instrucciones iniciales, residen en ROM por su carácter permanente. La última instrucción de todo este proceso es una **instrucción de retorno de interrupción**, mediante la cual se vuelven a cargar en los registros de la CPU los valores almacenados en la acción 1ª). De esta forma el proceso interrumpido vuelve a estar en la misma situación previa a la interrupción.

3. Continúa la ejecución del proceso inicial.

En la Fig. 3.31 se muestra, de manera esquemática y para controladores basados en un bit por estado, las modificaciones que hay que añadir a la unidad de control para incorporar la posibilidad de responder a la petición de interrupción. Como se ve, todo depende del estado en que se encuentre el biestable RS. Durante la operación normal (proceso inicial sin interrupción) el biestable tiene almacenado un "0". De esta forma, la salida de la puerta A2 es fijada a 0 con lo que la parte de control denominada "atención a interrupciones" no puede actuar. Por el contrario, al ser Q=1, la puerta A1 cierra el lazo en el "registro" de control que permite encadenar sucesivamente los ciclos de búsqueda y de ejecución. Así, el programa se va ejecutando de la forma ya conocida.

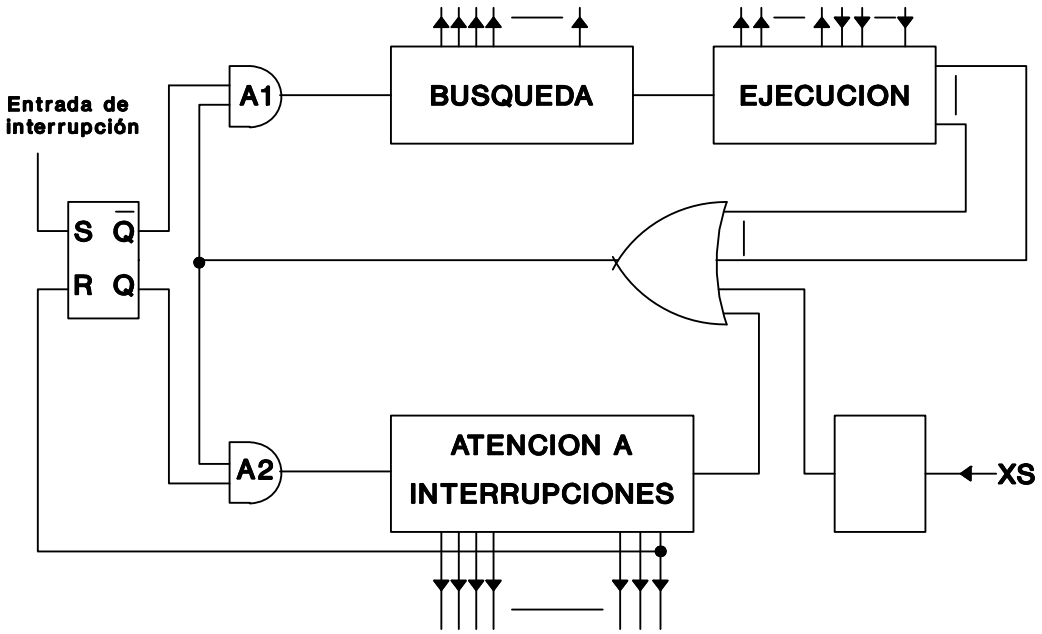


Figura 3.31: Una posible solución al controlador de un procesador con atención a inetrrupciones

Cuando se activa la entrada de interrupción, conectada a la señal de "Set", el biestable se pone a "1" con lo que la puerta AND superior (A1) estará inhabilitada, ya que su salida siempre será un cero lógico. Al mismo tiempo, la puerta AND inferior (A2) estará habilitada, lo que permitirá el paso de un "1" lógico cuando éste aparezca por la otra entrada de dicha puerta. Obviamente, si al ocurrir esto, el "1"

del controlador se encontrara en cualquier lugar de la parte de búsqueda o de ejecución, el proceso de control iniciado continúa hasta que acabe el ciclo de ejecución. Al regresar con la idea de comenzar un nuevo ciclo de búsqueda, éste se impide por la inhabilitación de la puerta AND superior (A1), permitiendo la inferior (A2) la ejecución de la parte de control dedicada al tratamiento de las interrupciones. Una vez terminado éste, el propio controlador activa la señal de "Reset". Así se coloca a "0" el biestable RS, permitiendo de nuevo llevar a cabo el ciclo de búsqueda-ejecución (proceso normal).

Anexo I
Ensamblador del Computador Simple

Ensamblador es un término que sirve para dos conceptos diferentes: lenguaje y programa. En efecto:

- El **lenguaje ensamblador** es un lenguaje de programación y como tal tiene definidas sus reglas (léxicas, sintácticas y semánticas). Usando el lenguaje ensamblador se escriben los denominados *programas en ensamblador*. Estos son programas que se interpretan directamente al lenguaje máquina del procesador y se denominan *programas de bajo nivel*. Así, p. ej. en el Computador Simple 2, las instrucciones del programa en ensamblador se interpretan a alguna de las 16 instrucciones del CS2, dando lugar a la secuencia de códigos de 0's y 1's que forman el *código ejecutable*, y que, tal como se ha diseñado, el CS2 ejecuta.

- El **ensamblador** también es una aplicación software, esto es, un programa, en este caso de alto nivel y escrito en cualquier lenguaje de programación. En este sentido, el **ensamblador** es la aplicación que permite pasar del *programa en ensamblador* al *código ejecutable* correspondiente.

Por ejemplo, si se desea resolver la multiplicación de dos números, usando el lenguaje ensamblador se hará un programa, que será el programa en ensamblador de la multiplicación. Después, el **ensamblador**, a partir de dicho programa en ensamblador, obtendrá el *código ejecutable en la máquina* (la máquina es, p. ej., el CS2) del programa de multiplicación. De esta forma el CS2, al ejecutar ese código máquina, realiza la multiplicación.

En adelante nos centraremos en el Proyecto Fin de Carrera de Delgado y Freniche[1]. En este proyecto se fijó una versión de lenguaje ensamblador del Computador Simple 2, se implementó un ensamblador para dicho lenguaje y se desarrolló un entorno de emulación de la operación del CS2, con el cual es posible no sólo ejecutar programas con el CS2, sino también seguir en detalle cómo opera ese computador.

LENGUAJE ENSAMBLADOR del CS2:

En general, un **lenguaje ensamblador** de un procesador consta de:

- INSTRUCCIONES EJECUTABLES: son todas las instrucciones del conjunto de instrucciones del procesador, se escriben con el mnemónico correspondiente y se interpreta al código ejecutable. Ejemplos de CS2: CLC que, de acuerdo con la Tabla 3.2, se interpretará al código $B00; LAIM $45, que se interpretará al código $045; etc.

- DIRECTIVAS DE ENSAMBLADO O PSEUDOINSTRUCCIONES: son instrucciones del lenguaje ensamblador, pero no de la máquina, por lo que **no dan lugar a código ejecutable**. Las directivas sirven para DOCUMENTAR y para FACILITAR la escritura de programas a través del uso de variables, etiquetas, etc.

---

1. Este programa ha sido desarrollado como Proyecto Fin de carrera por A.L. Delgado y D.I. Freniche dirigidos por los Profs. D. José I. Escudero y D. Alberto J. Molina. Usaron CS1 como nombre, pero se trata del CS2. El programa es de libre distribución, fácil de usar y se puede descargar desde http://www.dte.us.es/docencia/etsii/ii/ec/ (material de laboratorio), desde http://www.dte.us.es/docencia/etsii/itis/ec/Emuladores y desde http://www.dte.us.es/tec_inf/itig/etc2/ (prácticas de laboratorio)

En concreto, el lenguaje ensamblador del CS2 tiene como características principales:

- Cada línea del fichero del programa corresponde a una instrucción o a una directiva
- Contiene las 16 instrucciones del CS2 con el mnemónico ya dado y:

  * direcciones en hexadecimal con el formato $HH: p. ej. BCS $0B

  * datos en decimal (p. ej. LAIM 50), hexadecimal (LAIM $32) u octal (LAIM #062)

- Las **directivas** son solamente 3. Una que permite introducir **comentarios** y que se utiliza para documentar los programas. La segunda que permite definir **variables y asociarlas a direccione**s de memoria; de esta forma se pueden separar dos tareas; la de escritura del programa, el cual se hará con las variables sin preocuparse de dónde se ubicarán, y la de la asignación de zonas de memoria a los datos y programas; así, por ejemplo, la misma rutina puede servir para dos programas que sitúen los datos en lugares diferentes de la memoria. La tercera directiva es la que permite definir **etiquetas**, lo cual libera al programador de la tarea de contar en hexadecimal los lugares de salto. La sintaxis es:

  * comentarios a la derecha de cada “punto y coma“ (**;esto es un comentario**)

  * variables para direcciones: **EQU** nombre **$HH**, donde $HH_{(16}$ es la dirección asignada

  * etiquetas: se definen con un nombre seguido de “ : “ y una instrucción (**nometiq: mnem1**) y se utilizan llamándolas simplemente por el nombre (**mnem2 nometiq**), tanto antes como después de haberla definido.

La Figura I.a muestra un ejemplo de programa que usa estas directivas. En la primera línea simplemente se comenta que se trata de tal ejemplo. En la segunda se asocia ALTA a la dirección $AA, así, cuando en la tercera se escribe la instrucción ADDI ALTA, es como si se hubiera escrito ADDI $AA. Obsérvese que si en otro programa se hace EQU ALTA $98, la misma instrucción de la tercera línea, ADDI ALTA. pasaría a ser como ADDI $98. En la línea 3 también aparece definida una etiqueta, de nombre ciclo1, etiqueta que es usada en la última línea para hacer un salto (JMP ciclo1). Esta instrucción causará un salto a ADDI ALTA con independencia de en qué número de línea de programa se encuentre esa ADDI ALTA (y, por tanto, en qué posición de memoria esté escrita).

```
; esto es un ejemplo de uso de las directivas
            EQU ALTA $AA
            ... ... ... ...
ciclo1:     ADDI ALTA
            ... ... ... ...
            JMP ciclo1
```

Figura I.a Ejemplo de uso de directivas del computador simple

ENSAMBLADOR o programa ensamblador del CS2:

El ensamblador a secas, o programa ensamblador, es una aplicación software que lee un programa escrito en lenguaje ensamblador y, tras un análisis de errores, genera el código ejecutable en el CS2. La Figura I.b ilustra esta operación.

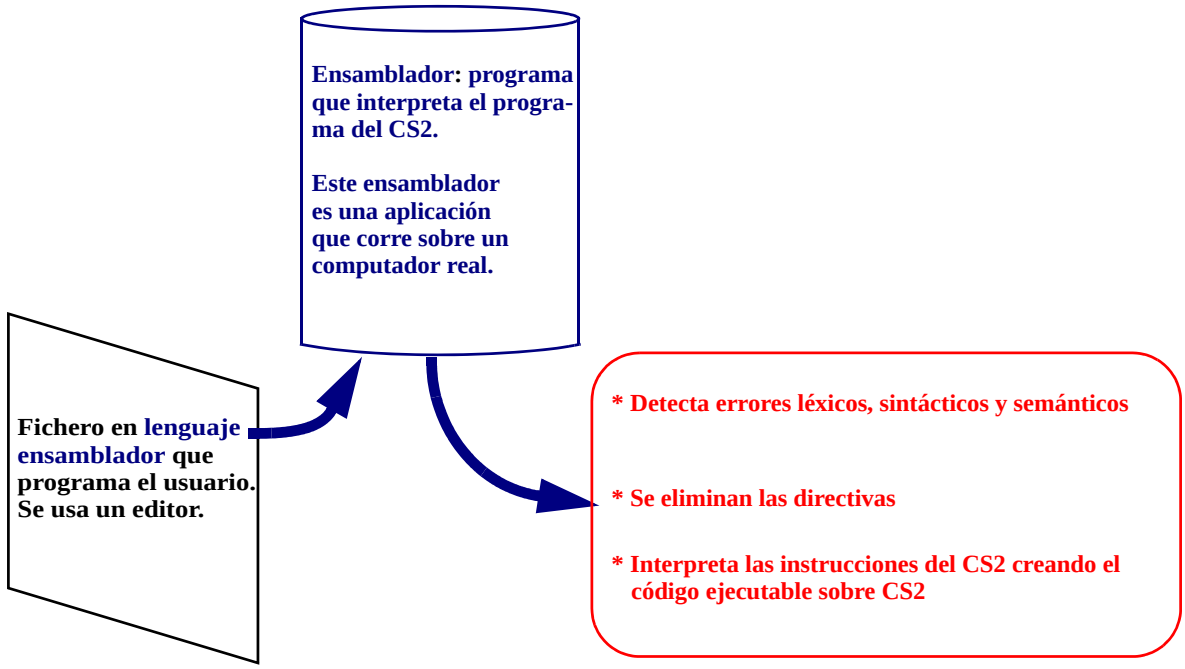


Figura I.b Acciones del programa ensamblador del computador simple

Se asume que existe un cierto fichero escrito por el usuario usando un editor (en este caso, texto plano), fichero que contiene el programa desarrollado en lenguaje ensamblador. El ensamblador lee este fichero y detecta e informa sobre un conjunto de errores del lenguaje, p. ej., da error si se escribe LAIN en vez de LAIM. El chequeo de errores no es funcional, esto es, no sirve para detectar si falta, sobra o hay que modificar una instrucción. Cuando el programa en ensamblador no tiene errores, el ensamblador hace una primera pasada para: 1/eliminar las directivas, 2/anotar qué direcciones se asocian a las variables (EQU), 3/asignar una dirección física de memoria a cada instrucción ejecutable (la primera, la pone en la posición $0, la segunda en $1, etc.) y 4/anotar también las direcciones de las instrucciones con etiquetas. En un segundo paso, sustituye todas las variables y etiquetas por los valores anotados (haciendo, si ha lugar el cambio de base desde la usada originalmente en el programa hacia la base 2) e, instrucción a instrucción, genera su código de operación (según la Tabla 3.2) y los valores binarios del campo de operando según la tabla de anotaciones realizadas previamente.

El resultado es, pues, el código ejecutable.

Ejemplo 1: Suma de “n” sumandos

Retomamos el ejemplo de la suma de “n” sumandos, para el que se daba una solución sin utilizar el ensamblador en el epígrafe 3.3.4.1. La versión escrita en ensamblador es la siguiente:

```
; Suma de n magnitudes (n = 8):
; M($43) = M($EF) + M($EE) + M($ED) + ... + M($E8)
;
; Variables
EQU CONT $AB ; CONT cuenta sumas que faltan
EQU ALTA $AA; ALTA señala al sumador siguiente
;
; Inicialización
        LAIM 8                  ; n = 8
        STA CONT                ; CONT es $AB
        LAIM $EF
        STA ALTA                ; inicializa ALTA = $EF
        LAIM 0                  ; AC = 0
; Programa principal
ciclo1: ADDI ALTA               ; acumula suma
        BCS fin                 ; sale si hay acarreo
        DBZ ALTA                ; siguiente sumando
        DBZ CONT                ; salta (N+2) si último
        JMP ciclo1
        STA $43                 ; almacena resultado
fin:    STOP
```

El ensamblador, una vez comprobado que no hay errores, elimina las líneas de comentario, sustituye las variables definidas en EQU por sus direcciones y las anota en las correspondientes instrucciones, en este caso, CONT=$AB, ALTA=$AA,. Entonces empieza a asignar las direcciones donde se encuentran las instrucciones ejecutables, comenzando en $00:

$00 para LAIM 8;
$01 para STA $AB (sustituyendo CONT por su valor $AB); etc.

Al llegar a la instrucción $05 se encuentra la etiqueta ciclo1, anotando que ciclo1=$05. Después, continúa asignando $06 a BCS fin encontrando una variable desconocida (fin) que es anotada en espera de que aparezca su valor. Esto ocurre al llegar a $0B (STOP), por lo que anotará fin=$0B y, en la segunda pasada, realiza la sustitución, BCS $0B.

También obtiene el código máquina (según codificación de Tabla 3.2), resultando el código ejecutable mostrado en la siguiente tabla:

.

$HH	mnem	[M]
00	LAIM 8	**0000 0000 1000**
01	STA $AB	**0010 1010 1011**
02	LAIM $EF	**0000 1110 1111**
03	STA $AA	**0010 1010 1010**
04	LAIM 0	**0000 0000 0000**
05	ADDI $AA	**0101 1010 1010**
06	BCS $0B	**1001 0000 1011**
07	DBZ $AA	**1010 1010 1010**
08	DBZ $AB	**1010 1010 1011**
09	JMP $05	**1000 0000 0101**
0A	STA $43	**0010 0100 0011**
0B	STOP	**1101 (00 ... 00)**

Ejemplo 2: Multiplicación por sumas sucesivas

Se desea realizar la multiplicación de un multiplicando (→ MD) por un multiplicador (→ MR) guardando el resultado (→ RES). La multiplicación se hará entre dos magnitudes C (multiplicando) y D (multiplicador) mediante sumas sucesivas (véase Anexo II)[1], que serán representadas con una palabra de 12 bits sin signo. Para resolver el problema se propone usar el algoritmo de multiplicación por sumas sucesivas. Una primera solución sencilla en código ensamblador a este problema consiste en:

a) Inicialización: definir variables; escribir datos C (→ MD) y D (→ MR); borrar RES

b) Mientras que MR sea distinto de 0:

   b1 Sumar MD en RES (en los sucesivos ciclos, RES = MD + MD + MD + ...)

   b2 decrementar MR

c) Guardar el resultado en su destino y acabar

Programa ensamblador:

```
        EQU MD $AA
        EQU MR $AB
        EQU RES $AC
; inicialización
        LAIM datoC              ; Obviamente, en vez de datoC se escribirá un número
```

1. En el epígrafe 3.3.4.2 se realiza un programa de multiplicación por sumas y desplazamientos a la izquierda.

```
                STA MD
                LAIM datoD          ; Igual que antes, un número en vez de datoD
                STA MR
                LAIM 0
                STA RES             ; RES se ha iniciado con 0
        ; bucle principal
BUCLE:          LDA MD
                ADD RES             ; Se ha sumado, en AC, un nuevo MD con el anterior RES
                STA RES             ; Se actualiza el valor de RES
                DBZ MR              ; Se decrementa MR. Si no es 0 vuelve a un nuevo ciclo de suma
                JMP BUCLE
                JMP FIN             ; Se podría poner aquí STOP
        ; fin del programa
        FIN:    STOP
```

Si se prueba este programa para valores pequeños de C y D, p. ej. C= 6 y D = 3, se comprobará que opera muy bien, dando un resultado en RES M($AC) = $012 (= $18_{(10)}$). Sin embargo, hay varias cuestiones no resueltas cuando se utilizan otros números. Así,

- Funciona bien si MD = 0, pero lo hace mal[1] si MR = 0. La razón es que, en la solución adoptada, se empieza decrementando MR antes de conocer si es 0, con lo cual, la primera vez, tras sumar MD, MR pasa a se 0-1 = $FFF.
- Si se prueba para valores grandes (p. ej. C = 250 y D = 70) el resultado mostrado en RES es erróneo porque, simplemente, ha habido desbordamiento. El problema de este programa es que no averigua ni avisa de cuándo ocurre.
- Como curiosidad, este es un ejemplo donde se comprueba que en la aritmética digital no se cumplen las propiedades de la aritmética con *papel y lápiz*. En primer lugar, como se destaca en el primero de estos puntos, 0 x 7 (0 en MD y 7 en MR) lo hace bien mientras que 7 x 0 lo hace mal (7 en MD y 0 en MR), por lo que no se cumple la propiedad conmutativa. Otro caso es cuando se multiplican números pequeños por números grandes: mientras que MD = 250 y MR = 9 requiere 9 ciclos por lo que el programa da la respuesta con rapidez, el caso conmutativo (MD= 9 y MR = 250) requiere 250 ciclos y el programa resuelve con mucha lentitud este caso.[2]

1. Medite si lo que se dice en este punto es estrictamente cierto. ¿Funciona mal si MR = 0 quiere decir que hay un error en la multiplicación? Lea el punto 3.
2. Replanteemos qué ocurre si MR = 0. Como se ha indicado antes, el programa inicia RES = MD y se hacen ciclos desde M = 0-1 = 2^{12} - 1. En total, pues, se harán 2^{12} sumas de MD. Esto es, debería aparecer MD·2^{12} que es MD y 12 ceros. Esto es un claro desbordamiento pero que, si se ejecuta enteramente, da lugar a 12 ceros en AC (y, por tanto, en RES), con lo que, si se desprecia el desbordamiento, ¡el resultado es correcto! Sin embargo, para obtenerlo, además de no considerar el desbordamiento, hay que hacer 2^{12} = 4092 ciclos ¡qué lentitud!

Ejemplo 3: Resolución de pequeños problemas con CS2

En este epígrafe se van a resolver diferentes *pequeños problemas* de programación en ensamblador del CS2. En todos ellos supondremos que las variables han sido definidas mediante EQU's (p. ej. EQU WORD $HH, siendo HH un par de dígitos hexadecimales cualesquiera).

En la escritura de números asumimos que es decimal por defecto. Así, N = 96 es lo mismo que poner N = $96_{(10}$.

Escribir un número N en la palabra WORD

Resolveremos el caso de representar números mediante sólo una palabra (12 bits) del CS2.

Caso 3. 1: N es una magnitud que cabe en 8 bits, p. ej. N = 96

```
        LAIM $60            ; $60 = 96(10
        STA WORD
```

Caso 3. 2: N es una magnitud que no cabe en 8 bits, p. ej. N = 596

En primer lugar se escriben los 4 MSB (bits más significativos), con 0's en los 8 LSB (bits menos significativos). Para ello hay que conocer el valor binario de N. Después se escriben los 8 LSB y se les suman a los anteriores. En este caso, 596 = $254, por lo que los 4 MSB son $2 = 0010. Por tanto, hay que situar un 1 en AC_9 para lo cual se usan LAIM o CLC o SEC y ROL o ROR, según convenga.

```
; Escribimos $2 en los 4 MSB de AC
        LAIM $80            ; AC = 0000 1000 0000
        ROL                 ; AC = 0001 0000 0000 pues C=0 tras LAIM
        ROL                 ; $AC = 0010 0000 0000 = $2 0 0
        STA WORD            ; WORD = $2 0 0
; Completamos los 8 LSB
        LAIM $54
        ADD WORD
        STA WORD            ; WORD = $2 5 4
```

Caso 3. 3: N es un número con signo POSITIVO que cabe en 12 bits, p. ej. N = (+) 96

Los números con signo en el CS2 están escritos en Ca2 (así se hace la suma o la resta), por lo que un número es positivo si es 0 su MSB. El mayor número que se puede representar en el CS2 es 2^{11} = 2047 (= $7 F F).

Si N cabe en 12 bits significa que su magnitud es menor que 2^{11}, por lo que $AC_{11} = 0$. Se puede escribir como si fueran magnitudes bien en el caso 3.1 si N < 256, bien en el caso 3.2 si $256 \leq N \leq 2047$. Esto es, hasta el valor 2047 (inclusive) da igual si es magnitud o número con signo positivo.[1]

Para el ejemplo N = (+) 96 la solución es idéntica al caso 3.1:

```
LAIM $60                ; $60 = 96(10
STA WORD
```

Caso 3. 4: N es un número con signo NEGATIVO que cabe en 12 bits, p. ej. N = - 96

Un primer método es que el programador obtenga la representación Ca2:

96 es 0000 0110 $0000_{(2}$, por lo que en Ca2 -96 es Ca2(0000 0110 0000) = 1111 1010 0000

y lo escriba como si fuese una magnitud, en este ejemplo la $F A 0, según el caso 3.2.

Para que el programador no tenga que hacer la conversión a Ca2, recuérdese que si un número con signo en Ca2 es positivo y cabe en 12 bits, también su opuesto (que es el negativo) cabe en 12 bits. Entonces, el opuesto de num puede obtenerse como 0 - num. Así, para escribir los números negativos basta escribir el positivo (casos 3.1 o 3.2) y restárselo a 0.

Para el ejemplo N = -96:

```
LAIM $60                ; $60 = 96(10
STA WORD
LAIM 0
SUB WORD                ; AC = 0 - 96 = - 96 en Ca2
STA WORD
```

Este método tiene única excepción al número extremo negativo representable ($8 0 0, correspondiente a -2048), cuyo positivo no es representable. Así, para escribir el -2048 se hará:

```
LAIM 0
SEC                     ; C = 1
ROR                     ; AC = 1000 0000 0000
STA WORD
```

Utilización de subrutinas

El uso de subrutinas ahorra repetir mucho código, pero su manejo requiere algún cuidado. Las principales ideas a tener en cuenta son:

1. La diferencia entre magnitud y número positivo es que desde 2048 hasta 4095 sólo pueden ser magnitudes si sólo se usa una palabra de representación.

- Terminan con RTS (a diferencia del programa principal, que acaba con STOP)
- En general se intercambian datos entre programa principal y subrutinas, por lo que hay que planificar dicho intercambio de datos. Para ello, usad variables.
- Las direcciones de retorno se almacenan en $FF (después, en $FE, y en $FD, ...). Por tanto, en el uso del CS2 hay que evitar el uso de las últimas direcciones de memoria para poner datos

Caso 3. 5: Dados los números sin signo M, N y P, obtener Prod = M · N · P

Usaremos como subrutina de multiplicación el programa del Ejemplo 2 aún sin arreglar los problemas de multiplicador 0 y de desbordamiento. Suponemos que ya se ha hecho la inicialización y están definidas las variables M, N, P y Prod, así como las de la multiplicación (MD, MR y RES)

```
        ; Lo que sigue es sólo la parte de llamada a subrutina
                LDA M
                STA MD
                LDA N
                STA MR
                JMP SubrMult        ; Al regresar RES = M x N
                LDA RES
                STA MD
                LDA P
                STA MR              ; Ahora está preparadp para hacer (M x N) x P
                JSR SubrMult        ; Al regresar RES = (M x N) x P
                LDA RES
                STA Prod
                STOP
        ; La subrutina que sigue empieza cuando se pone RES a 0 en la rutina del Ejemplo 2.
SubrMult:       LAIM 0
                STA RES             ; RES se ha iniciado con 0
        ; bucle principal
BUCLE:          LDA MD
                ADD RES             ; Se ha sumado, en AC, un nuevo MD con el anterior RES
                STA RES             ; Se actualiza el valor de RES
                DBZ MR              ; Se decrementa MR. Si no es 0 vuelve a un nuevo ciclo de suma
                JMP BUCLE
                RTS                 ; La instrucción RTS indica el final de la subrutina
```

Bifurcaciones por comparación

Una tarea común al desarrollar un algoritmo es decidir entre dos valores. Las formas más típicas de bifurcar un programa en bajo nivel son:

- Comparar un dato con 0. El dato puede ser una posición de memoria o el valor del acumulador.
- Comparar un dato con otro. Los datos pueden ser con o sin signo.

En los siguientes casos trataremos diferentes opciones de comparación, sin perseguir la solución exhaustiva en todos los casos.

Caso 3. 5: Comparación con 0. Siendo MV la posición de memoria a valorar, queremos desarrollar: IF MV = 0 then Tarea1 else Tarea2

Una forma fácil es sumarle 1 y utilizar entonces la instrucción DBZ que nos proporciona directamente los dos caminos:

```
        LAIM 1
        ADD MV
        STA MV          ; MV tiene el valor original más 1
        DBZ MV          ; MV recupera el valor original
        JMP Tarea2      ; Si MV no era 0 salta a Tarea2
        JMP Tarea1      ; Si MV era 0 salta a Tarea1
```

Caso 3. 6: Comparación del AC con 0: queremos desarrollar: IF AC = 0 then Tarea1 else Tarea2

Una forma fácil es almacenar AC en una posición de memoria libre (que llamaremos MV) y aplicar el Caso 3.5. Para preservar en AC el valor inicial, éste debe ser almacenado y, tras el test, las dos tareas deberían empezar por recuperar el valor de AC almacenado previamente. El programa sería:

```
        STA AC
        LAIM 1
        ADD MV
        STA MV          ; MV tiene el valor original de AC 1
        DBZ MV          ; MV recupera el valor original de AC
        JMP Tarea2      ; Si AC no era 0 salta a Tarea2
        JMP Tarea1      ; Si AC era 0 salta a Tarea1
Tarea1: LDA MV          ; AC recuperado al iniciar la Tarea1
```

```
        ... ... ...
Tarea2: LDA MV          ; AC recuperado al iniciar la Tarea2
        ... ... ...
```

Caso 3. 7: Comparación A = B: queremos desarrollar:

IF A = B then Tarea1
else Tarea2

La solución es fácil: se restan, el resultado se almacena en MV y a MV se le aplica el Caso 3.5.

```
        LDA A           ; MV tiene el valor original de AC 1
        SUB B           ; MV recupera el valor original de AC
        STA MV          ; MV tiene el valor original de AC 1
; A partir de aquí se repiten las instrucciones del caso 3.5
```

Caso 3. 8: Comparación A ≥ B para números con signo. Queremos desarrollar:

IF A ≥ B then Tarea1
else Tarea2

Hay dos formas principales: 1/Investigar los signos de A y B y, si son iguales, comparar las magnitudes. 2/Restarlos y verificar si el resultado es mayor o igual que 0 (A ≥ B □□⇔ A-B ≥ 0).

- **Investigando signos de A y de B**

Si A y B tienen signo están representados en Ca2 (el bit de signo es el 11) y puede ocurrir:

* A y B tienen signo diferente: ** $A \geq 0$ y $B < 0 \Leftrightarrow A_{11} = 0$ y $B_{11} = 1 \Leftrightarrow A \geq B$

** $A < 0$ y $B \geq 0 \Leftrightarrow A_{11} = 1$ y $B_{11} = 0 \Leftrightarrow A < B$

* A y B tienen el mismo signo, $A_{11} = B_{11}$. En este caso, A y B en Ca2, tanto si ambos son positivos como si ambos son negativos se cumple que $A \geq B \Leftrightarrow A-B \geq 0$. Por otra parte, si se restan A - B nunca puede haber desbordamiento (*overflow*, ya que se trata de números con signo) ya que el resultado de la resta es siempre menor o, como sumo, igual, al de ambos operandos. Por tanto, tras restar, un resultado positivo significa que el minuendo es mayor que el sustraendo ($A \geq B$).

El siguiente programa implementa esta idea. Para ver el valor del bit 11, éste se lleva al bit de *carry* (C) y se utiliza la instrucción BCS (salta si C = 1).

```
        LDA A
        ROL             ; C = signo de A
        BCS Aneg
        LDA B           ; aquí A es positivo
```

```
                ROL
                BCS Tarea1          ; Salta a Tarea1 pues B es negativo y A positivo
                JMP mismosigno      ; ambos son positivos
Aneg:           LDA B
                ROL
                BCS mismosigno      ; ambos son negativos
                JMP Tarea2          ; Salta a Tarea2 pues B es positivo y A negativo
mismosigno:     LDA A
                SUB B               ; AC = A - B. Ahora averiguaremos el signo de AC
                ROL
                BCS Tarea2          ; la resta resultó negativa (A < B)
                JMP Tarea1          ; la resta resultó positiva (A ≥ B)
```

- **Restando A - B**

Con independencia del signo de A y de B, siempre que no exista desbordamiento (*overflow*), el signo del resultado de A - B indicará quién es mayor, como se ha explicado antes. Sin embargo, si existe *overflow*, el signo de A - B no aparece en A_{11}, sino que es el contrario. Desgraciadamente el CS2 no incorpora una bandera de *overflow*, por lo que averiguar si ha habido o no *overflow* debe hacerse a través de los signos. La solución al problema de conocer si $A \geq B$ es o no fácil según se conozca o no por el contexto si puede existir *overflow* al restar:

* Si se sabe que no puede haber *overflow*, el programa es muy simple y coincide con el caso del mismo signo anterior, esto es:

```
                LDA A
                SUB B               ; AC = A - B. Ahora averiguaremos el signo de AC
                ROL
                BCS Tarea2          ; la resta resultó negativa (A < B)
                JMP Tarea1          ; la resta resultó positiva (A ≥ B)
```

* Si hay que averiguar si ha existido *overflow* en la resta, el programa es más complejo. Sea la resta AC = A - B; puede haber overflow si y sólo si en dos casos: 1)A es positivo, B es negativo y AC resulta negativo; o 2)A es negativo, B es positivo y AC resulta positivo. Averiguar esto por programación con el CS2 tiene una dificultad mayor que el anterior programa correspondiente a investigar los signos.

Por consiguiente, si existe posibilidad de overflow, se recomienda seguir el procedimiento de investigar los signos, mientras que si no existe esa posibilidad, debe hacerse mediante resta.

Caso 3. 9: Comparación A ≥ B para números sin signo. Queremos desarrollar:

IF A ≥ B then Tarea1
else Tarea2

En el ensamblador desarrollado para el CS2, la operación SUB causa que el bit C se ponga a 0 si la magnitud del sustraendo es mayor o igual que la del minuendo, mientras que se pone a 1 si es mayor la magnitud del minuendo[1]. Esto es, como SUB $HH realiza AC - M($HH):

* Si C = 0 ⇨ AC ≥ M($HH) * Si C = 1 ⇨ AC < M($HH)

El programa resultante es muy simple:

```
        LDA A
        SUB B
        BCS Tarea2
Tarea1  mnem            ;primera instrucción de la tarea 1
```

Contadores crecientes y decrecientes

Para recorrer listas *hacia abajo* (direcciones crecientes: ... 34 y, después, 35 y, después, 36 y, después,...) se necesita un puntero creciente mientras que, para recorrerlas *hacia arriba*, se necesita un puntero decreciente. Estos punteros no son, en definitiva, más que unos contadores de subida o de bajada. En ambos casos, puede necesitarse, además, un contador que detecte cuántos valores se han de recorrer con el puntero.

En los ejemplos 1 y 2 ya se han usado estos punteros y contadores decrecientes, los cuales aprovechan la instrucción DBZ, que decrementa y bifurca según se llegue a 0 o no. Ahora simplemente mostramos cómo construir un puntero creciente:

```
        LAIM valorinicial       ; Obviamente valorinicial es un dato numérico
        STA puntero
        LAIM 1
        STA uno
        LDA puntero
        ADD uno
        STA puntero
```

1. Esto es distinto a lo que hacía el sumador/restador desarrollado en *Fundamentos de Computadores*. En éste, el bit C se pone a 1 cuando el minuendo tiene mayor o igual magnitud que el sustraendo. La solución adoptada en el ensamblador de CS2 pone C como si fuera BW_{out} de una resta de magnitudes.

Ejemplo 4: Emular el almacenamiento (*store*) con direccionamiento indirecto

El direccionamiento indirecto permite recorrer posiciones consecutivas de datos mediante una misma instrucción que forma parte de un ciclo de instrucciones. En el caso del CS2 se pueden pasar datos desde la memoria hacia los registros internos (AC) mediante la instrucción ADDI, única que se ha implementado con direccionamiento indirecto. Sin embargo, para pasar datos desde AC hacia la memoria sólo se dispone de STA (direccionamiento directo). La ausencia de un STA con direccionamento indirecto, que bien se podría denominar STAI (STA Indirecto) ha resultado ser la principal carencia del conjunto de instrucciones del CS2.

Supongamos que el contenido del acumulador se va a almacenar repetidamente en posiciones consecutivas de memoria, primero en $9F, después en $9E, después en $9D, etc. Si sólo disponemos de la instrucción STA, en la primera escritura habría que poner STA $9F, en la segunda STA $9E, en la tercera STA $9D, etc., siendo imposible hacer un bucle. Si dispusiéramos de la instrucción STAI se resolvería fácilmente esta cuestión pasando por ella en ciclos sucesivos.

Vamos a presentar cómo *realizar* una STAI en el CS2. Sea stind la posición de referencia del direccionamiento indirecto (STAI stind), que en nuestro ejemplo será $F0 (EQU $F0 stind). Sea también $nn la primera dirección (la mayor) en donde se desea almacenar el acumulador la cual, en el caso propuesto, es $9F. La idea es “emular” STAI stind mediante una subrutina[1] de la siguiente forma:

1. En dos posiciones de memoria consecutivas (stind=$F0 y $F1) se escriben los códigos máquina de STA y de RTS como sigue:
   M($F0) = $2nn, que corresponde a STA $nn. En el caso elegido: M($F0) = $29F
   M($F1) = $F00, que corresponde a RTS

2. Cuando se quiera usar STAI en el programa principal se escribiría:

```
JSR stind       ; se ejecutará M(stind) = STA $9F y, seguidamente, RTS
DBZ stind       ; M(stind) = M(stind) - 1: se apunta a la siguiente dirección
sigue 1         ;   sigue ejecutando la tarea deseada tras “STAI”
sigue 2         ;   etc.
```

- **Ejemplo: Existen 20 ($14) datos almacenados en memoria desde la dirección $EF hacia posiciones decrecientes. Realice un programa CS2 que los reescriba a partir de $9F.**

El programa será:

```
        EQU fuente $80
        EQU cont $81
        EQU stind $F0 ; RTS estará en $F1
; Escribimos código máquina de STA nn = $29F
        LAIM $80
```

1. El autor de este idea es D. Alejandro Muñoz Rivera, cuando fue alumno de esta asignatura

```
                ROL
                ROL
                STA stind ; M(stind) = $200 : STA $00
                LAIM $9F
                ADD stind
                STA stind ; M(stind) = $29F : STA $9F
        ; Escribimos código máquina de RTS: $F00
                LAIM $F0
                ROL
                ROL
                ROL
                ROL
                STA $F1 ; M($F1) = $F00 : RTS
        ; Iniciamos cont a 20 = $14 y fuente a $EF
                LAIM $14
                STA cont ; cont = número de datos
                LAIM $EF
                STA fuente ; fuente = $EF

        ; Ciclo de movimiento de datos
ciclo:          LAIM 0
                ADDI fuente ; AC = Dato_fuente
                DBZ fuente ; nueva dirección fuente (- 1)
                JSR stind ; mueve dato a destino
                DBZ stind ; nueva dirección destino (- 1)
                DBZ cont
                JMP ciclo
                STOP
```

Obviamente, la misma idea puede ser aplicada para almacenar *hacia abajo* (en orden de direcciones crecientes) sin más que hacer en stind un puntero creciente (explicado inmediatamente antes de este ejemplo 4).

EMULADOR del CS2

El proyecto realizado por A.L. Delgado y D.I. Freniche no se limitó a desarrollar el ensamblador del Computador Simple, sino que implementó un emulador donde es posible ejecutar los programas completamente y paso a paso, permitiendo ver los datos de la memoria y los datos de los registros internos incluso ciclo a ciclo de reloj.

A continuación se va a exponer una breve descripción del entorno (basada en un trabajo de D. Enrique Ostúa Arangüena).

En el directorio de trabajo, que es C:\csimple\, al activar el ejecutable aparecerá el entorno correspondiente al programa simulador ocupando toda la pantalla.

• Descripción de la ventana

En esta ventana se pueden distinguir varias zonas: la mitad de la izquierda contiene la unidad de datos del CS2 y en la mitad de la derecha aparecen:

- la ventana "**Registros**": en ella se muestra el contenido de los diferentes registros de la unidad de datos en códigos hexadecimal ($), decimal (!) y octal (#);
- la ventana "**RAM**": en ella se muestra el contenido de la memoria en hexadecimal. Para avanzar por sus distintas direcciones se pueden usar las flechas que aparecen a su derecha, o las teclas del 0-9 y del A-F.
- la ventana "**Instrucciones**": muestra el programa que se está ejecutando, señalando en color rojo la instrucción concreta que se está procesando;
- la ventana "**Microinstrucciones**": muestra la descomposición en microinstrucciones de la instrucción en curso;
- los selectores de modo de ejecución: se encuentran a la derecha de la ventana de "Instrucciones" y son tres:
- **I**nstrucciones: si se pulsa, el programa se ejecutará instrucción a instrucción. Esto permite la depuración de programas ya que se puede seguir la variación de los registros o posiciones de memoria tras cada instrucción.
- **M**icroinstrucciones: si se pulsa, el programa se ejecutará microoperación a microoperación, es decir, ciclo a ciclo de reloj. En este modo se puede seguir el movimiento de los datos en la unidad de procesado en la que se van iluminando en amarillo los buses, registros y señales de control implicados en cada microoperación.
- **G**O STOP: si se pulsa se ejecuta el programa sin detenerse hasta alcanzar su última instrucción.
- otros pulsadores: RESET (borra los registros, la memoria completa y *resetea* el computador), COMIENZO (coloca el PC apuntando a $00), CARGAR, EDITAR, AYUDA, Fetch, Res. Inst., Ver Fuente, Ver out1, Ver out2, EXIT. La función de cada uno de estos se puede consultar desde el propio programa, para ello basta pulsar AYUDA y se mostrará toda la información en una ventana.

Los 3 selectores principales tienen una tecla de acceso rápido, que es la letra iluminada en la palabra que aparece en el pulsador. Por ejemplo, para ejecutar el "**G**O STOP" basta con pulsar la tecla "G". El acceso a distintas zonas de la memoria **RAM** también se puede hacer rápidamente con el

teclado, en este caso con las teclas del “0” al “9” (posiciones $00 a $90, saltos de $10) y las teclas de la “A” a la “F” (que nos llevará a las posiciones $A0 a $F0, en saltos de $10 en $10).

- **Ejemplo de creación de un programa**

Inicialmente no tenemos cargado ningún programa.

Para crear un programa nuevo debemos pinchar en el pulsador “EDITAR”. Esto último abrirá al editor de MSDOS *edit*, con un nuevo documento en blanco llamado “none”. Cuando el programa esté escrito, salve con la opción “Guardar” (o “Guardar Como...” si desea cambiarle el nombre) y a continuación salga del editor. Con esto volverá automáticamente al entorno del simulador y se recargará el programa “none” de nuevo. Tenga en cuenta que si ha guardado el programa con otro nombre distinto, debe entonces cargar con las instrucciones que aparecen a continuación.

- **Ejemplo de carga de un programa**

Cualquier programa escrito en un fichero en texto plano (creado por ejemplo con el editor de MSDOS o con el bloc de notas) puede ser cargado por el emulador. Para cargar uno pulse la tecla “CARGAR” (o pulse F3). Escriba la ruta completa al programa en la ventana que aparece, por ejemplo “C:\CSIMPLE\NONE” (nombre del programa por defecto). Si el fichero tiene extensión (como “.txt”) deberá incluirla también. Al pulsar sobre el icono del disco (o pulsando ENTER), las ventanas del entorno se llenarán con la información correspondiente al programa que se ha cargado el cual podrá ejecutarse en los modos descritos anteriormente.

Si intentó cargar un programa que contiene errores de sintaxis aparecerá una ventana que le informa de la línea en donde se ha encontrado dicho error. En tal caso, proceda a editarlo (botón EDIT o desde un editor externo) y recárguelo en el simulador.

El simulador incluye una carpeta denominada “Examples” que contiene diversos programas de ejemplo para el CS2.

Anexo II

Multiplicación de magnitudes

Sean dos números sin signo, A y B. Aunque no sea necesario, supondremos que ambos son de n bits. El número $A=A_{n-1}\ A_{n-2}\ldots\ldots A_1\ A_0$ actuará de multiplicando y $B=B_{n-1}\ B_{n-2}\ldots\ldots B_1\ B_0$, de multiplicador. El producto $P=A \times B$ es un número sin signo de 2n bits: $P=P_{2n-1}\ P_{2n-2}\ldots\ldots P_1\ P_0$.

El algoritmo de multiplicación mediante suma y desplazamiento de la magnitud se justifica de forma matemática a través de la ecuación 1, desarrollada por la ecuación 2:

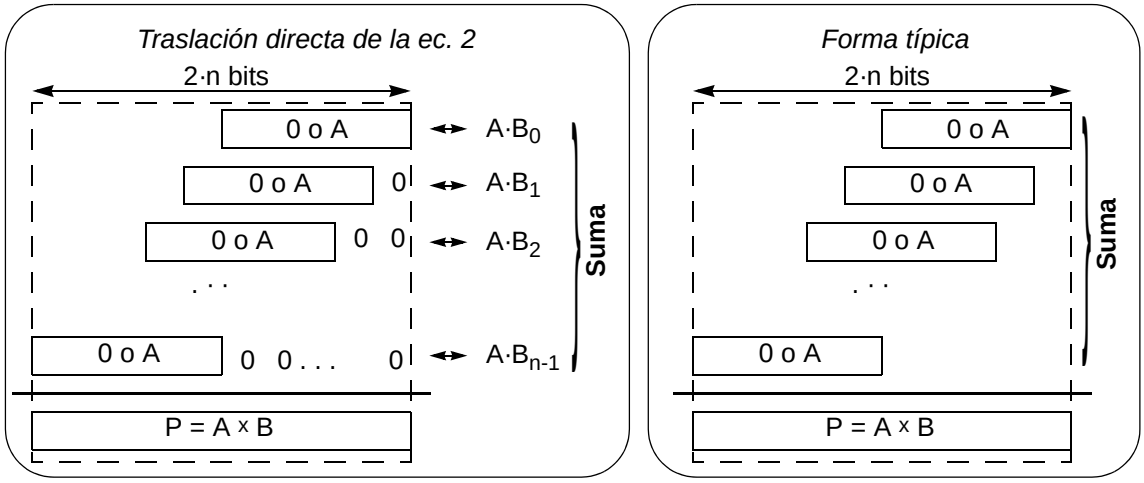
$$A \times B = A \bullet \left(\sum_{i=0}^{n-1} B_i \cdot 2^i \right) = \sum_{i=0}^{n-1} A \cdot B_i \cdot 2^i \qquad (1)$$

$$A \times B = A \cdot B_0 + A \cdot B_1 \cdot 2^1 + A \cdot B_2 \cdot 2^2 + \ldots + A \cdot B_{n-1} \cdot 2^{n-1} \qquad (2)$$

Como se observa en la ecuación (2), el producto se obtiene sumando “n” productos parciales. Cada uno de ellos, se obtiene teniendo en cuenta que:

- Multiplicar A por B_i, es 0 o A según que B_i sea 0 o 1. Esto puede hacerse bien preguntando por el valor de B_i y dar el resultado A o 0 según corresponda, bien aplicando la operación lógica AND de B_i con el multiplicando A.
- Multiplicar por 2^i es “añadir ‘i’ ceros a la derecha”. Para llevar a cabo la multiplicación por 2 se desplaza a la izquierda entrando el valor 0 por la derecha; para multiplicar por 2^i se hacen ‘i’ desplazamientos.

Una visión de este algoritmo es la que da en la siguiente figura: A la izquierda se expone la traslación directa de la ecuación 2 como suma de ‘n’ productos parciales cada uno de los cuales vale 0 o A (según B_i) desplazado ‘i’ veces con lo que se consigue la alineación de las posiciones de los bits en cada producto parcial. A la derecha aparece la forma típica, sin más que eliminar los 0’s a la derecha y mostrando desplazado el valor $A \times B_i$.



En muchos sistemas digitales la suma de los productos parciales se hace mediante un sumador de sólo dos sumandos. En este caso, la multiplicación se realiza mediante una iteración de sumas y desplazamientos a la izquierda de la siguiente forma:

- Se pone a 0 un registro que denominamos Producto Acumulado, PA.
- Comenzando por el bit menos significativo (B_0) y continuando hacia la izquierda de B hasta el más significativo (B_{n-1}), se genera cada producto parcial ($A \times B_i$ que será igual a 0 o a A) y se alinea mediante desplazamiento a la izquierda (para alcanzar el valor correcto $A \times B_i \times 2^i$).
- Ante cada nuevo producto parcial y con la adecuada alineación, se efectúa una "suma intermedia" del resultado anterior PA con este nuevo producto parcial, suma que se acumula en PA.

La multiplicación de dos operandos de n-bits necesita 'n' ciclos de sumas y de desplazamientos, un sumador paralelo de '2·n' bits y registros A y PA de también '2·n' bits. Se producen 'n' productos parciales y la suma de estos productos parciales con su respectiva alineación da lugar al producto final.

Algoritmo con desplazamiento a la derecha

Puede observarse que el bit "0" del producto parcial (PP) $A \times B_0$ no se suma nada más que con 0's, por lo que coincide con el bit 0 del resultado final. Análogamente, el bit 1 generado tras sumar los dos primeros productos parciales ($A \times B_0$ y $A \times B_1$) no se suma en adelante más que con 0's, por lo que coincide con el bit 1 del resultado final. Y así sucesivamente. Entonces, resulta más eficiente sumar sólo 'n' bits cada vez y desplazar el resultado a la derecha para guardar el LSB cada vez, ya que nunca más vuelve a cambiar. Así surge el algoritmo de sumas y desplazamientos a la derecha, que utiliza recursos con la mitad de tamaño que el de desplazamiento a la izquierda.

La siguiente figura muestra los dos algoritmos de desplazamiento en el caso particular del producto 6 x 9.. Los bits del resultado en color verde son los que, tras generarse, no cambian más

A = 0110 B = 1011 ⟹ **P =** 6 x 11 = 66 = **0100 0010**

Desplazamiento a izquierda

1º PA = 0
2º B_0= 1: PA ← PA+A — PA: 0000 0000 + A: 0000 0110 = PA: 0000 0110
3º SHL (A) — A: 0000 1100
4º B_1= 1: PA ← PA+A — PA: 0000 0110 + A: 0000 1100 = PA: 0001 0010
5º SHL (A) — A: 0001 1000
6º B_2= 0: PA ← PA+0 — PA: 0001 0010 + 0 = PA: 0001 0010
7º SHL (A) — A: 0011 0000
8º B_3= 1: PA ← PA+A — PA: 0001 0010 + A: 0011 0000 = PA: 0100 0010
9º SHR (PA) — A: 0110 0000

Desplazamiento a derecha

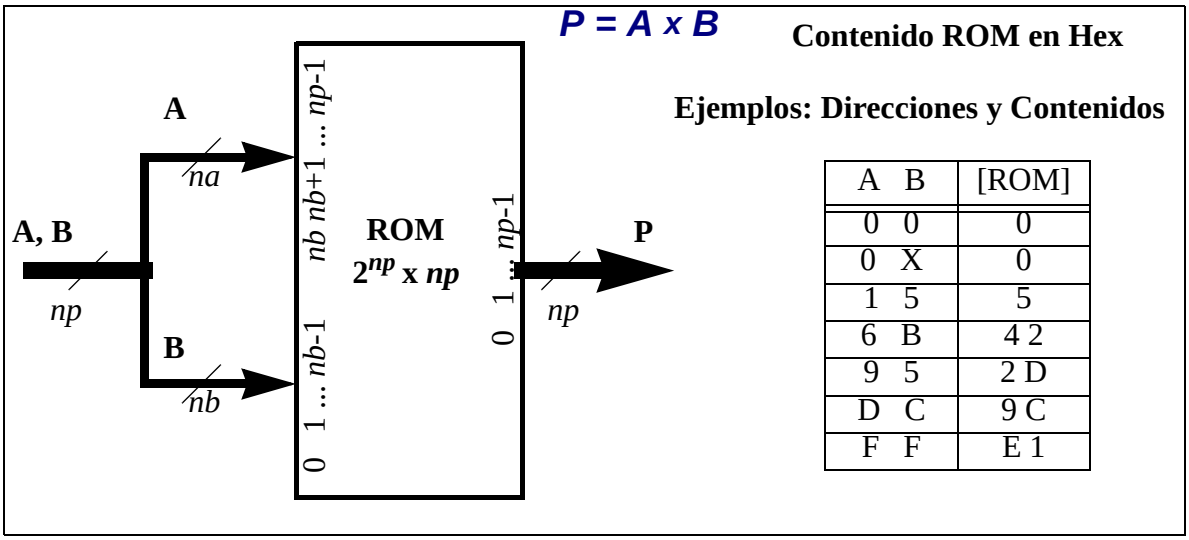
1º PA = 0
2º B_0= 1: PA ← PA+A — PA: 0000 0000 + A: 0110 = PA: 0110 0000
3º SHR (PA) — PA: 0011 0000
4º B_1= 1: PA ← PA+A — PA: 0011 0000 + A: 0110 = PA: 1001 0000
5º SHR (PA) — PA: 0100 1000
6º B_2= 0: PA ← PA+0 — PA: 0100 1000 + 0 = PA: 0100 1000
7º SHR (PA) — PA: 0010 0100
8º B_3= 1: PA ← PA+A — PA: 0010 0100 + A: 0110 = PA: 1000 0100
9º SHR (PA) — PA: 0100 0010

P = 0100 0010

Otras dos formas muy simples de realizar la multiplicación son:

Método de mirar la tabla (LUT: *Look Up Table*)

El **método de mirar la tabla (*Look Up Table*)** consiste en implementar toda la tabla de multiplicar A x B sobre una memoria ROM. Los datos presentes en A y B direccionan una palabra de la ROM en la cual se ha escrito previamente el resultado correspondiente (ver figura siguiente). .



A B	[ROM]
0 0	0
0 X	0
1 5	5
6 B	4 2
9 5	2 D
D C	9 C
F F	E 1

En esta implementación el tiempo de multiplicación es únicamente el tiempo de acceso a ROM. Sin embargo, cuando el número de bits crece, el tamaño de ROM crece exponencialmente y esta solución se vuelve extraordinariamente costosa: aumentar en 1 bit cada dato A y B multiplica por 4 el número de palabras y aumenta en 2 el número de bits por cada palabra. Para el caso genérico de la figura, el tamaño pasaría de ser $T = 2^{np} \times np$ a ser $T' = 2^{np+2} \times (np+2) = 4\ (2^{np} \times np) + 4\ (2^{np} \times 2)$. Nótese que sale una capacidad mucho mayor de 4 veces la original. P. ej., pasar de 4 a 5 bits en A y B lleva a pasar de una ROM (2^8 x 8), de 2Kbits, a una ROM (2^{10} x 10), de 10Kbits.

Sumas sucesivas

Multiplicar A x B es sumar A tantas veces como indique B, según la definición de multiplicar. Este algoritmo es muy fácil de implementar si se dispone de la capacidad de sumar dos operandos -para ir sumando A con el valor acumulado previamente- y de un contador con decremento que, partiendo de B, va contando las veces que se suma A consigo mismo.

Este método de multiplicar tiene como contrapartida que puede durar mucho tiempo, lo que ocurre si el multiplicador es un número grande.

Notas finales

Cuando se multiplican dos números, siempre es conveniente preguntar si alguno vale cero, en cuyo caso, siempre el resultado es 0 y se obtiene rápidamente.

La multiplicación no produce desbordamiento si se dispone del número de bits suficiente. Si no es así, puede producirse bien por acarreo en la suma, bien si se desplaza un “1” a la izquierda.

Anexo III
Glosario

AB, *Address Bus*: Bus de direcciones, véase Bus.

Activa en baja, señal: Una señal se dice que es activa en baja si la acción ocurre cuando la señal posee un nivel bajo. Se representa con un superrayado; p. ej., una señal de selección de chip activa en alta se representa como CS y una activa en baja, como $\overline{\text{CS}}$. Si la señal tiene dos significados, p. ej., de lectura (R) y escritura (W), se superraya la activa en baja: R/$\overline{\text{W}}$ significa que un valor alto causa la lectura y uno bajo, la escritura.

Acumulador: Registro especial de la CPU que aporta uno de los datos y recibe los resultados de la ALU, y sobre el que se realizan muchas de las operaciones sobre un dato (p. ej. los desplazamientos).

Asíncrono: Sin señal de reloj.

ASM, *Algorithmic State Machine*: Máquina de estados para algoritmos (algorítmica). Véase carta ASM.

Alta impedancia: Se representa por HI (*High Impedance*). Una señal binaria posee alta impedancia si se puede conectar/desconectar electrónicamente. La línea tiene tres estados posibles: dos si la línea está conectada (el de 0 y el de 1) y un tercero que es el de desconexión, llamado también tercer estado, o valor HI, el cual no tiene ninguna interpretación binaria (no es 0 ni 1), por lo que no da lugar a interpretación lógica. Las líneas con HI se llaman líneas de 3 estados (tristate). La conexión/desconexión está controlada por una señal de habilitación.

Bandera: *Flag*.

Binario: Que opera con dos valores. En general puede referirse a una información en bits (0/1), a un circuito eléctrico (señal con nivel bajo o alto, L/H, *Low/High*), a una red de conmutación (*switch* o conmutador, OFF/ON), a una función lógica (sentencia falsa o verdadera, F/T: *False*/*True*), etc. En general y salvo que se quieran matizar sus diferencias, los términos binario, digital, lógico o de conmutación suelen utilizarse como sinónimos.

Bit (*Binary digit*): Dígito binario, 0 o 1.

Bits de estado (*Status bits*): Conjunto de bits que señalan algunos aspectos *o estados* tras una operación entre datos, como son, entre otros: **C** (***c**arry*), **V** (*o**v**erflow*), **Z** (***z**ero*), o **S** (***s**ign*).

***Buffer tristate*:** Dispositivo electrónico que permite dotar con HI a un circuito de conmutación.

Bus: Conjunto de líneas de conexión entre dispositivos que se utiliza para transmitir información. Suelen diferenciarse según su funcionalidad. Así, en un computador suelen distinguirse tres buses: el de direcciones (AB, *Address Bus*), el de datos (DB, *Data Bus*) y el de control (CB, *Control Bus*).

Byte, B: Grupo de 8 bits (también llamado octeto).

Búsqueda, ciclo de: *Fetch cycle*. La CPU lee en la memoria la palabra apuntada por el contador de programa, PC, y la guarda en su registro de instrucciones IR. Además, incrementa PC para que éste quede apuntando a la siguiente instrucción que se ha de buscar.

Carta ASM: Grafo orientado y cerrado usado para representar ASM. Además de los arcos orientados, posee tres estructuras básicas: la **caja de estado** (rectangular), la **caja de decisión** (romboi-

dal) y la **caja de acción condicional** (ovalada). Cada caja de estado define un **bloque ASM** que, además, puede contener diversas cajas de decisión y de acción condicional, o no contener ninguna.

***Carry* (C):** Acarreo. El *carry out* o acarreo de salida es un bit de estado que indica el desbordamiento en la suma sin signo.

CB, *Control Bus*: Bus de control, véase Bus.

Ciclo de reloj: Es el periodo de la señal de reloj. Marca el intervalo de tiempo de permanencia en cada estado.

Circuito digital o de conmutación: Circuito electrónico que básicamente opera con valores binarios de señal, alta (H: *High*) o baja (L: *Low*). Sus componentes básicos son las puertas y los biestables, su conexionado se hace vía cables y la función que realizan es combinacional o secuencial.

Circuitos de entrada/salida (I/O *circuits*): Dispositivos o unidades de interfaz entre el mundo exterior (periféricos) y los buses internos de un computador. Hay unidades I/O para transmisión de datos en paralelo o en serie, y de forma síncrona o asíncrona.

Circuito integrado (IC, *Integrated circuit*): Encapsulado con terminales conductores (*pines*) conectados interiormente a un chip que realiza funciones electrónicas.

***Clock*, *Ck*:** Señal de reloj o, simplemente, reloj.

Chip: Semiconductor con circuitos electrónicos implementados en él. Es la parte principal de un circuito integrado.

***Chip Select*, *CS*:** Selección de chip.

Código de dirección o de operando: Campo del código de instrucción donde se representa cuál es el operando o en qué dirección se encuentra. Véase modos de direccionamiento.

Código de instrucción: Son los 0's y 1's que representan la instrucción. Véase formato de instrucción.

Código de operación, CO o COP: Campo del código de instrucción donde se representa qué operación se ha de realizar como, por ejemplo, mover desde memoria a la CPU (*load*) o desde la CPU a la memoria (*store*), sumar (*add*) o restar (*subtract*), poner a cero (*clear*) o a 1 (*set*), saltar (*jump*) o bifurcar (*branch*), parar (*stop*), etc.

Código ejecutable o máquina de un programa: Es el código de instrucciones generado tras traducir o interpretar un programa de un computador. Se trata, pues, de los 0's y 1's que sustituyen a los mnemónicos de las instrucciones del programa. Así, la instrucción del computador simple 2 que en mnemónico se escribe BCS $34, tiene el código ejecutable: 1001 0011 0100.

Comandos: Señales de salida de la unidad de control. Pueden ir a la Unidad de Procesado o al exterior.

Comienzo, señal de: Representada por Xs (Xstart), es un cualificador externo usado para iniciar la ejecución de la macro-operación del sistema digital. En la técnica de un biestable por estado la señal Xs debe durar un ciclo de reloj.

Complemento a 2, operación de: Dada una palabra A de "n" bits, por definición, el complemento a 2 de A es:

$$Ca2(A) = 2^n - A_m \ (\text{mód. } 2^n)$$

donde es A_m representa la magnitud que tiene A si se toma como número sin signo. La palabra A puede empezar por 0 o por 1.

Complemento a 2, representación en: Representación “n” bits de números binarios con signo en la que los números positivos se representan igual que en signo-magnitud (esto es, como 0-magnitud) y los negativos se representan como el complemento a 2 de la magnitud descrita con “n bits”.

Contador de programa, PC: O puntero de programa, es un registro que señala la instrucción que se ha de ejecutar. Más precisamente, durante los ciclos de ejecución el registro PC contiene la dirección donde se encuentra la instrucción que se va buscar en el próximo *fetch cycle*, esto es, apunta a la dirección donde se encuentra la próxima instrucción a buscar.

Control mediante un biestable por estado: Técnica de realización cableada de controladores cuya solución se obtiene directamente de la carta ASM del controlador, de forma fácil y prácticamente sin esfuerzo. Cada estado/bloque ASM tiene su propio biestable, cada caja de decisión se realiza con un DEMUX y la unión de señales o caminos se implementa con una puerta OR. La señal de comienzo es la que dispara la ejecución de la macro-operación.

Control cableado: Técnica de diseño de unidades de control que proporciona realizaciones personalizadas (no microprogramadas) y que usa biestables y puertas. Entre otras técnicas, cabe citar las realizaciones con **lógica discreta**, las que usan **biestables D y multiplexores**, y la que usa **un biestable por estado**.

Control microprogramado: Técnica de diseño de unidades de control donde el hardware tiene una estructura fija de circuito secuencial de propósito general (básicamente, PLA o ROM y registro de carga en paralelo), siendo los módulos programables (PLA o ROM) los que personalizan la implementación del controlador a realizar. La personalización de PLA o ROM constituye el microprograma de control (*firmware*).

Controlador: Unidad de control.

CPU, *Central Processor Unit*: Procesador.

Cualificadores: Señales de entrada a la unidad de control. Pueden venir de la Unidad de Procesado o del exterior.

***Data sheets*:** Hojas de datos de los circuitos.

Dato (binario): Valores (binarios) concretos que toman las variables y palabras que son procesados.

DB, *Data Bus*: Bus de datos, véase “Bus”.

DEMUX, *Demultiplexer*: Demultiplexor.

Descripción estructural: Representa cómo está constituido un componente, sus entradas, sus salidas y, si procede, sus partes (que, a su vez, también son descritas estructuralmente). En esta obra se usa el dibujo de los circuitos como forma de descripción estructural. Se trata de una representación, no de una realidad física.

Descripción funcional: Indica cómo opera un componente. En esta obra se usa frecuentemente una tabla de verdad que desarrolla todas las opciones de las señales de selección de operaciones del componente.

Descripción física: Indica cómo es realmente un componente. En esta obra no suele haber descripcio-

nes físicas que, sin embargo, están siempre presentes en muchos documentos de interés, como son, por ejemplo, las hojas de datos (*data sheets*).

Dinámico: Que varía de valor en el tiempo. Su opuesto es estático.

Dirección (*address*): Indica dónde está una palabra de información (dato o instrucción). En la memoria principal y en los dispositivos de memoria RAM y ROM, la dirección es el valor binario que toma el bus de direcciones, AB. En el código de instrucción, es el campo en el que se indica dónde está el operando.

Directivas de ensamblado: Instrucciones del lenguaje ensamblador que no son ejecutables por el procesador, sino que sirven de ayuda a la programación para documentar y facilitar el programa (p. ej., líneas de comentarios, variables o etiquetas).

DMA, *Direct Access Memory*: Dispositivo para acceder directamente a la memoria principal, sin que la CPU ejerza el control del bus.

Ejecución, ciclo de: *Execution cycle*. La CPU decodifica y ejecuta la instrucción almacenada en su registro de instrucciones IR. Salvo que se trate de la instrucción de parada, tras le ejecución de la instrucción actual se continúa con un nuevo ciclo de búsqueda.

Ensamblador, lenguaje: Lenguaje de programación que utiliza como primitivas las instrucciones de un procesador descritas en mnemónico, a las que se añaden otras **directivas** o **pseudoinstrucciones de ensamblado**.

Ensamblador, programa: Programa que lee un programa escrito en lenguaje ensamblador y lo **interpreta** al código ejecutable del procesador.

Entradas de control: Cualificadores.

Estacionario: Que se mantiene sin cambios en su estado o situación, o que se reproduce en el tiempo sin variar el comportamiento. El concepto contrario es transitorio. Ejemplos de estacionario: una señal estática (p. ej. un bit a 1) está estacionaria; también un oscilador que genera una señal senoidal de 1 KHz está en estado estacionario. Ejemplo estacionario/transitorio: una señal de reloj tiene comportamiento estacionario como señal periódica, pero tiene transitorios cuando conmuta de 0 a 1 y de 1 a 0.

Escritura: Véase **Operaciones RT**.

Estado inicial, S_0: En un sistema digital se refiere al estado de no-operación en el que se encuentra el sistema a la espera de que llegue una señal de comienzo, Xs.

Estático: Con valor constante. Su opuesto es dinámico.

***Execution cycle*:** Ciclo de ejecución.

***Fetch cycle*:** Ciclo de búsqueda.

***Firmware*:** Programas que residen en dispositivos programables no volátiles. (El programa consiste en la adecuada personalización del dispositivo). Es un concepto intermedio al hardware y al software. El control residente en *firmware* se denomina control microprogramado.

Flag: Bandera. Se usan como indicadores de los bits de estado.

Formato de instrucción: Campos en que se divide el código de instrucción para señalar alguna pro-

piedad particular. Típicamente hay dos campos: el del **código de operación** (véase esta entrada) y el del **código de dirección** o de operando (véase esta entrada).

FPGA, Field-Programmable Gate Array: Circuito electrónico programable de alta densidad que puede contener el equivalente a miles de puertas lógicas.

Frecuencia (máxima): La frecuencia es el valor inverso del periodo de reloj y se mide en hercios (Hz, o s^{-1}). La frecuencia máxima de un sistema es el límite superior de la velocidad de operación del sistema.

FSM, *Finite State Machine*: Máquina con número finito de estados. Estructura matemática que se utiliza para describir funciones secuenciales.

Función combinacional: Función de conmutación cuyo valor depende sólo del valor presente en sus variables, salvo en los transitorios.

Función secuencial: Función de conmutación cuyo valor depende, además del valor presente en sus variables, de la secuencia de cambios que han experimentado. Se dice que tienen memoria. Se describen por una máquina con número finito de estados (FSM).

Giga: Ver unidades.

***Hardware*:** Parte tangible de un sistema constituida por los dispositivos físicos: circuitos, cables, etc.

HDL, *Hardware Description Language*: Lenguaje de descripción de hardware.

HI, *High Impedance*: Alta impedancia.

Instrucción: Operación que realiza automáticamente un sistema digital. Puede tardar uno o varios ciclos de reloj. En sistemas digitales que no operan en modo computador, suelen denominarse macro-operaciones.

Interpretar un programa: Sea un programa en un determinado lenguaje de programación, L1, y sea otro lenguaje de programación, L2. Interpretar el programa en L1 consiste en obtener, para cada instrucción de L1, el conjunto de instrucciones equivalentes en L2. Una tarea diferente es **traducir un programa**.

Interrupción: Sistema ideado para que un computador pueda atender a periféricos cuando está ejecutando un programa principal. Cuando un periférico quiere actuar con el computador, activa alguna señal de interrupción. Dependiendo de la prioridad que exista en ese momento, se atenderá la interrupción (ejecutándose la rutina de interrupción correspondiente) o se rechazará dicha interrupción continuando la ejecución del programa principal.

I/O *Input/Output circuit*: Circuito de entrada/salida.

ISP, *Instruction Set Processor*: Conjunto de instrucciones del procesador. Se denomina así al nivel de descripción de computadores cuyas primitivas son fundamentalmente las instrucciones. Es el nivel inmediatamente más abstracto que el nivel RT. Se trata del nivel más bajo de software, el que establece un puente con el hardware. La resolución de un problema a nivel ISP consiste en un programa cuyas instrucciones puede ejecutar el hardware.

Kilo: Ver **unidades**.

Lectura: Véase **Operaciones RT**.

Lenguaje de descripción de hardware: Lenguaje de programación ideado para describir el hardware estructural y funcionalmente. Los dos lenguajes más usados actualmente son VHDL y Verilog y con ellos se puede, además de especificar diseños, realizar simulaciones y efectuar la síntesis del diseño sobre FPGAs o sobre circuitos integrados. En esta obra se utiliza un lenguaje de descripción de hardware extremadamente simple y básico, cuyo único objeto es mostrar cómo describir la secuencia de control de un sistema digital mediante instrucciones.

Lenguaje máquina: Lenguaje que consta de las instrucciones ejecutables por un procesador (o máquina). Su representación mediante 0's y 1's da lugar al código ejecutable.

LIFO, *Last In First Out*: Memoria secuencial tipo **pila** (ver ***stack***). También llamada **FILO, *First In Last Out*.**

LSB, *Least Significant Bit*: Bit menos significativo (posición "0" en las representaciones en punto fijo sin parte fraccionaria). Por defecto, en esta obra se representa en la posición más a la derecha.

Macro-operación: Instrucción.

MAR, *Memory Address Register*: Registro de dirección de memoria.

Mega: Ver unidades.

Memoria: Término con doble significado. Se usa para referirse a los dispositivos físicos que almacenan información, p. ej., una memoria EPROM. También se usa para referirse a la unidad funcional de almacenamiento de la información (datos y programas) en un computador, con independencia de los dispositivos físicos de memoria donde se aloje esa información.

Micro-operación, μop: Operación básica de un sistema digital que se tiene que ejecutar en un solo ciclo de reloj y consistente en una o varias operaciones de transferencias entre registros simultáneas.

Microprocesador: CPU integrada en un solo chip.

Mnemónico o mnemotécnico: Forma muy resumida, muchas veces a base de siglas, con la que se recuerda algo. Es muy usado para representar el juego de instrucciones de un procesador de forma adecuada a las personas. Así, en el computador simple 2, la instrucción de "bifurcación (*branch*) si el acarreo (*carry*) es 1 (*set*)" se escribe como BCS, que es el mnemónico de "*branch if carry set*".

Modelo de usuario: Descripción de los registros de usuario de una CPU, sin incluir detalles del diseño.

Modo de calculadora y modo de computador: En el modo de calculadora el sistema digital ejecuta automáticamente una macro-operación (instrucción) y se detiene a la espera de que se le especifique y ordene ejecutar la siguiente. En el modo de computador existe un programa almacenado que el sistema va a ejecutar ordenadamente hasta la instrucción final (p. ej. STOP), siguiendo la secuencia siguiente: primero realiza el ciclo de búsqueda (*fetch cycle*) en el que el sistema busca la instrucción que ha de ejecutar y la almacena en su CPU; a continuación la decodifica y ejecuta en lo que se llama el ciclo de ejecución (*execution cycle*); después, salvo que haya sido STOP, comienza un nuevo ciclo de búsqueda de instrucción.

Modos de direccionamiento: Son las distintas maneras de indicar la dirección de un operando en una instrucción. Entre otros, se pueden citar:
Implícito o inherente: Cuando o no hay operando o éste no tiene que ser especificado por

estar implícito en el código de operación, p. ej. *clear carry*.
Inmediato: El operando se da en el campo de dirección.
Directo: En el campo de dirección se da la dirección del operando.
Indirecto: En el campo de dirección se da la dirección del registro o palabra de memoria que contiene la dirección del operando. A veces incluye un desplazamiento sobre esa dirección, desplazamiento cuyo valor se da en la propia instrucción. Si se trata de un registro especial, recibe nombre específico: **Relativo** si el registro especial es PC, **Indexado** si se usa un registro índice, etc.

Monoestable: Circuito digital que, ante una excitación de entrada, responde dando un pulso de duración constante. En la terminología sajona se denominan circuitos *one-shot*.

MSB, *Most Significant Bit*: Bit más significativo (posición "n-1" en palabras de "n" bits). Por defecto, en esta obra se representa en la posición más a la izquierda.

MUX, *Multiplexer*: Multiplexor.

Nibble: Palabra de 4 bits.

NOP, *No OPeration*: Inhibición, Sin operación.

Operando: Dato objeto de la operación.

Operaciones RT: Son de dos tipos: de **escritura**, esto es de almacenamiento de un dato, y de **lectura**, o sea, de acceso a un dato almacenado. Las de escritura requieren que ocurra el flanco activo de reloj, ya que por defecto tratamos sólo con sistemas digitales síncronos. Típicamente son la carga en paralelo, los desplazamientos, la cuenta y la puesta a 0 o a 1. En cuanto a las de lectura, el acceso al dato se puede hacer incondicionalmente o bajo alguna condición (típicamente activar una señal de lectura, R: *Read*); también se puede acceder a los "n" bits del dato, a sólo algunos de ellos o a alguna función combinacional de ellos (p. ej., al bit de estado Z).

Ordinograma u organigrama: Grafo orientado usado para expresar el orden en el desarrollo de una tarea o algoritmo.

PC, *Program Counter*: Contador de programa.

PIPO, *Parallel In Parallel Out*: Registro con entrada y salida en paralelo.

PLA, *Programmable Logic Array*: Matriz (arreglo) lógico programable. Es un subsistema combinacional programable que permite realizar múltiples funciones del tipo sumas de productos. Una versión más restringida, en la que sólo se pueden programar términos productos, se denomina PAL, *Programmable Array Logic*. Otra versión, que incorpora biestables, se denomina PLD, *Programmable Logic Device*.

Periférico: Componente de un computador que no se conecta directamente con sus buses internos, sino que lo hace a través de los circuitos de entrada/salida del computador. Ejemplos: teclado, monitor, unidad grabadora de DVD, etc.

Primitivas RT: Son las micro-operaciones utilizables en un sistema digital. En general, el término *conjunto de primitivas* alude al conjunto de entes utilizables en el desarrollo de alguna tarea y tiene un significado concreto según cada caso. Así, en un programa escrito usando el lenguaje "L" las primitivas son las instrucciones del lenguaje "L".

Programa (de computador): Secuencia ordenada de instrucciones que resuelve algún problema y

puede ser ejecutada por un computador.

Pseudoinstrucciones de ensamblado: Directivas de ensamblado.

Pseudolenguaje: Es un lenguaje de programación indefinido, pero de uso muy intuitivo que sirve para describir tareas en el desarrollo de un algoritmo de forma que ese desarrollo sea fácilmente comprensible.

Puntero de pila, SP: Es un registro de la CPU que apunta a la palabra de memoria que actúa como palabra de arriba (*top*) de la pila.

Punto fijo: Representación de números con parte entera y parte fraccionaria.

Punto flotante: Representación binaria de números con notación exponencial o científica: el número se representa a través de una mantisa y del exponente de una base cuyo valor se conviene.

Reloj: Señal de un circuito secuencial síncrono que marca cuándo se cambia el estado. El flanco activo también sirve de referencia de tiempo para todo el sistema.

RAM, *Random Acces Memory*: Memoria de acceso aleatorio, es la memoria de lectura/escritura semiconductora más habitual en la memoria principal. Es volátil.

***Read*, señal de:** Señal de lectura.

Rebotes: En los conmutadores, pulsadores e interruptores reales la señal de salida no pasa limpiamente de 0 a 1, o de 1 a 0, sino que oscila un número determinado de veces entre el valor inicial y el final antes de acabar estabilizándose en este último. Esas oscilaciones se conocen como rebotes y suelen ser una fuente de errores en los circuitos que recogen esas señales de salida. Para la buena operación deben ser filtrados o chequeados sólo después de haberse estabilizado la señal.

Registro: Componente digital que almacena “n” bits (dato). Además del subsistema del mismo nombre, en el nivel RT se entiende como registro cualquier dispositivo de almacenamiento de un dato, como p. ej. un contador (registro con las operaciones de incremento o decremento), un biestable (registro de 1 bit) o una palabra de una memoria RAM o ROM.

Registro de código de condición, CCR, *Condition Code Register*: Registro de la CPU en el que se guardan las banderas o ***flags***: **C**, **V**, etc.

Registro de dirección de memoria, MAR, *Memory Address Register*: Registro de la CPU en el que se guarda la dirección de la palabra de memoria a la que se accede.

Registro de estado, SR, *Status Register*: Registro de **código de condición**.

Registro de instrucción, IR, *Instruction Register*: Registro de la CPU en el que se guarda el código de la instrucción.

Registro de usuario: Son los registros de la CPU que se ven afectados por alguna de las instrucciones del procesador. El valor de un registro de usuario concreto sólo debe verse afectado por las instrucciones específicas que manipulan ese registro en concreto; en la ejecución de las restantes instrucciones debe permanecer sin cambios.

***Reset*:** Señal o estado de inicio. Así, una señal de *Reset* lleva a un sistema digital al estado inicial S_0. Aunque al estado S_0 se le puede decir estado de *Reset*, en los sistemas digitales complejos reales el estado de *Reset* normalmente se refiere a todo un proceso de inicialización con

muchos estados que puede llegar a durar mucho tiempo por lo complicado que resulta.

ROM, *Read Only Memory*: Memoria de sólo lectura, es la memoria de lectura semiconductora más habitual en la memoria principal. Es no volátil.

RT, *Register Transfer*: Transferencia entre registros. Se usa para denominar al lenguaje y al nivel de descripción que opera con datos binarios. En cuanto a su significado como operación, véase **transferencia entre registros**.

Rutina de interrupción: Programa que se ejecuta al atender una interrupción, fuera del programa principal que se estuviera ejecutando en ese momento. Tras terminar la rutina de interrupción se regresa al punto del programa principal donde se atendió la interrupción.

S, *sign*: Bit de estado que indica el signo del resultado (coincide con el MSB).

Salidas de control: Comandos.

Secuenciador: En controladores microprogramados, el secuenciador es el componente que conduce la secuencia de micro-operaciones. Básicamente es un registro PIPO con cuenta ascendente (o, desde otro punto de vista, un contador con carga en paralelo).

Selección de chip, señal de: Señal que permite habilitar la operación de un circuito integrado (típicamente una memoria RAM o ROM).

Signo-magnitud: Representación de números binarios con signo en el que el MSB es el bit de signo y vale 0 en los números positivos y 1, en los negativos.

Síncrono: Posee una señal de reloj que organiza la operación en el tiempo.

***Software*:** Parte lógica e intangible de un sistema digital, constituida básicamente por programas.

***Stack*:** Pila, apilamiento. Memoria en la que se accede a los datos en orden inverso a como han sido escritos. Se llama también memoria tipo LIFO. El acceso es para escribir un dato (*push*) o para extraer un dato tras leerlo (*pull* o *pop*). Se denomina *top* de la pila al registro al que se accede directamente con *push* o con *pull*.

***Stack Pointer, SP*:** Puntero de pila.

Subrutina: Fragmento de programa que es llamado y ejecutado desde otro programa al cual regresa tras terminar la ejecución de dicho fragmento. La ejecución de subrutinas forma parte de la tarea resuelta en el programa principal, característica que distingue a la subrutina de la rutina de interrupción.

Tera: Ver unidades.

Tiempo de acceso: Es el intervalo de tiempo que transcurre entre la orden de acceso, normalmente acceso a un dato almacenado en memoria, y la plena disponibilidad de ese dato. Este concepto debe ser particularizado de varias formas. Así, en un dispositivo de memoria: si el acceso es de lectura o escritura; si se cuenta respecto a la validación de AB o respecto al de alguna señal de selección de operación (como selección de chip, *chip select*, o de escritura, *write*), etc.

Traducir un programa: Sea un programa en un determinado lenguaje de programación, L1, y sea otro lenguaje de programación, L2. Traducir el programa en L1 consiste en obtener, para ese programa en su conjunto, otro programa equivalente en L2. Una tarea diferente es **interpretar un**

programa.

Transferencia entre registros: Operación básica en el nivel RT, mediante la que se almacena en un registro el dato resultante al operar con ciertos datos. Requiere que se pueda realizar en un solo ciclo de reloj. Véase, también, RT y micro-operaciones.

Transitorio: Intervalo de tiempo que transcurre entre dos valores estacionarios consecutivos distintos.

Unidad de control: Parte de un sistema digital dedicada al control. Sus principales tareas son: recibir los cualificadores (entradas de control) y enviar los comandos (salidas de control), tanto al exterior como a la unidad de procesado, y desarrollar ordenadamente la secuencia de micro-operaciones correspondiente a la instrucción (macro-operación) que se está ejecutando.

Unidad de datos: Unidad de procesado.

Unidad de procesado (de datos): Parte de un sistema digital que procesa datos: los recibe del exterior, los almacena, opera con ellos y los saca al exterior.

Unidades, múltiplos y divisores: Las unidades de los parámetros **científicos** (p. ej. el segundo, s, o la frecuencia, Hz) o **lógicos** (número de *Bytes*, B) son a veces muy grandes o muy pequeñas para expresar el valor del parámetro medido, por lo que se usan múltiplos y divisores. Dentro del campo científico:
* los divisores más comunes son: **mili**, m (1 ms = 10^{-3} s); **micro**, μ (1 μs = 10^{-6} s); **nano**, n (1 ns = 10^{-9} s); **pico**, p (1 ps = 10^{-12} s); **femto**, f (1 fs = 10^{-15} s); y **atto**, **a** (1 at = 10^{-18} s);
* los múltiplos más comunes son: **Kilo**, K (1 Ks = 10^{3} s); **Mega**, M (1 Ms = 10^{6} s); **Giga**, G (1 Gs = 10^{9} s); **Tera**, T (1 Ts = 10^{12} s)
En cuanto a los múltiplos lógicos se usan las potencias de base 2 con exponentes múltiplos de 10: **Kilo**, K (1 KB =2^{10} B); **Mega**, M (1 MB = 2^{20} B); **Giga**, G (1 GB = 2^{30} B); **Tera**, T (1 TB = 2^{40} B).
Los valores científico y lógico de los múltiplos son parecidos pero no iguales. Así, el Kilo vale 1000 si es científico (10^{3} = 1000), pero vale 1024 si es lógico (2^{10} = 1024). Usualmente el propio contexto hace que no haya confusión entre ambos. Pero hay algunos casos en los que hay que tener en consideración estas diferencias. Por ejemplo, en un canal de comunicación que transmita 1 B cada ciclo de reloj, con un reloj de 1 GHz en un segundo *no se puede* transmitir una memoria de 1GB, ya que en un segundo hay 10^{9} = 1.000.000.000 de ciclos de reloj y, por tanto, se transportan 1.000.000.000 B, pero la memoria tiene 2^{30} = 1.073.741.824 B, por lo que quedarían 73.741.824 B sin transportar.

V, o*verflow*: Desbordamiento. Bit de estado que indica el desbordamiento en operaciones de números con signo representados en complemento a 2.

***Word*:** Palabra. Referida a una información binaria tiene una dimensión genérica de “n” bits, aunque en un contexto específico, como el del 68000, tiene el valor concreto de 16 bits (2 *Bytes*).

***Write*, señal de:** Señal de escritura.

Xs, *Xstart*: Cualificador (entrada de control) mediante el que se inicia la operación del sistema.

Z, *Zero*: Bit de estado que indica un valor cero en el resultado.

Anexo IV
Referencias

GREEN, D.: "Modern Logic Design". Addison-Wesley, 1986.

* En el cap.4 se trata la síntesis con dispositivos lógicos programables.

HILL, F.J. and PETERSON, G.R.: "Digital Logic and Microprocessors". Wiley, 1984.

* En el cap.9 se trata el tema del diseño basado en cartas ASM, se describe coma pasar a tablas y diagramas de estado a partir de las mismas

MANDADO, E., : "Controladores Lógicos y Autómatas Programables". Marcombo, 1992.

* Las dos primeras partes del libro están dedicadas a controladores. Se trata el tema con una visión diferente a como ha sido tratado aquí, pero de una forma muy amplia y puede ser util para una mayor profundización. En el apéndice 2 se describe un controlador basado en una PROM que ha sido realizado en la Universidad de McGill.

MANO, M.M.: "Digital Design". Prentice-Hall, 1991.

* En el cap.8 se describen distintos ejemplos de realización de controladores basados en una carta ASM.

PROSSER, F.P. and WINKEL, D.E.: "The Art of Digital Design: An Introduction to Top-Down Design". Prentice-Hall, 1987.

* En el cap.5 se describen distintas formas de control.

TAUB,H.: "Circuitos Digitales y Microprocesadores". McGraw-Hill, 1983.

* En el cap.8 se describe ampliamente el diseño de un controlador basado en un registro de desplazamiento.

Bibliografía relativa al control microprogramado:

ANDREWS, M.: "Principles of Firmware Engineering in Microprogram Control". Pitman, 1980.

DAVIO, M., DESCHAMP, J.P., THAYSE, A.: "Digital Systems with Algorithm Implementation". John Wiley, 1983.

DESCHAMPS, J.P., ANGULO, J.M.,: "Diseño de Sistemas Digitales: Metodología Moderna". Paraninfo, 1989.

KRAFT, G.D. and TOY, W.N.: "Microprogramed Control and Reliable Design of Small Computer". Prentice-Hall, 1981.

PÉREZ LÓPEZ, S.A.: "TESIS DOCTORAL: "Nuevos métodos sistemáticos de diseño de controladores lógicos". Universidad de Vigo, 1991.