7

Diseño de una computadora digital

Contenido

Objetivos

- Incorporar el lenguaje técnico.
- Rescatar la importancia del lenguaje ensamblador; para conocer y programar a nivel hardware una computadora. Su inclusión obedece a la consideración de que la programación de máquina permite comprender de manera más eficaz el módulo más importante de la computadora.
- Conocer una máquina Von Neumann para comprender el funcionamiento de una computadora en relación con su diseño interno.
- Comprender la capacidad del hardware para interpretar una secuencia de instrucciones ordenadas lógicamente: el programa.
- Reconocer los componentes de una computadora y el set de instrucciones básicos de la máquina del ejemplo.

En la página Web de apoyo encontrará un breve comentario de la autora sobre este capítulo.

7.1 Introducción

El capítulo siguiente apunta a que se pueda comprender el funcionamiento de una computadora básica en relación con su diseño interno y demostrar la importancia del estudio del Álgebra de Boole y de la teoría de circuitos lógicos como elementos necesarios e imprescindibles para alcanzar ese fin. El modelo de computadora presentado es de una estructura muy sencilla, de manera que permite que se lo estudie en su totalidad. Para lograr entender la estructura interna de esta computadora se hará referencia a algunas instrucciones básicas del lenguaje de programación *Assembler*, que permitirán que se imagine el movimiento de las instrucciones o los datos de un programa a través de los registros internos que componen cada unidad de esta computadora. La verdadera magnitud de los sistemas actuales se desarrolla en capítulos posteriores a éste.

7.2 Módulo de cálculo en una computadora digital

El diseño de una computadora digital es la organización de módulos de hardware relacionados por rutas de control y datos. Su función es permitir el flujo de señales binarias para transformar datos de entrada en información útil al usuario de la computadora.

Un módulo está constituido por una configuración determinada de compuertas. En el diseño de lógica se hace abstracción de los elementos electrónicos que constituyen los circuitos; simplemente se asume que un circuito es una "caja negra" que cumple determinada función lógica. Lo que interesa es relacionar las cajas negras para lograr un producto que, ante determinadas entradas, presente las salidas esperadas.

Cada módulo realiza una o varias operaciones sobre datos codificados en sistema binario, que se almacenan en registros asociados al módulo mientras dura la operación. Una operación aplicada a un registro se denomina **microoperación** (μop) y se activa en un instante de tiempo sincronizado por los pulsos del reloj. El diseño de una computadora es más abstracto que el diseño de la lógica de un módulo y se ocupa de ensamblarlos.

Considérese como ejemplo de módulo de cálculo el sumador binario paralelo de la figura 7.1.

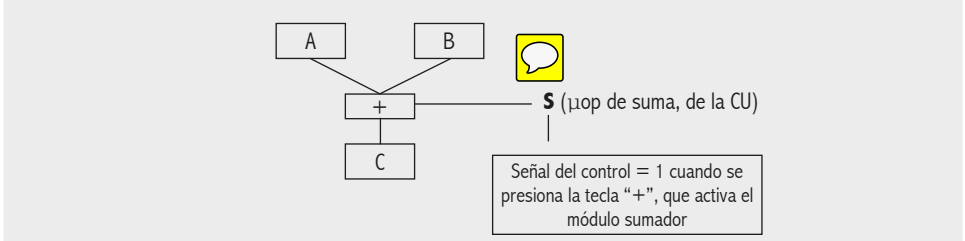


Fig. 7.1. Diagrama de bloque de un dispositivo que suma los bits de los registro A y B.

Su función es operar datos binarios para obtener en la salida el resultado de su suma. Los datos de entrada se almacenan en forma temporal en los registros A y B y, tras la orden de comando S, el resultado se obtiene sobre el registro C. La orden de comando es la microoperación de suma que habilita al registro C para que actúe de receptor del resultado. La orden puede ser una señal "*1*" generada por otro módulo, cuya función es "dar órdenes" en el caso de que este sumador pertenezca a una computadora.

7.3 Relación entre el diseño del hardware y la ejecución de instrucciones

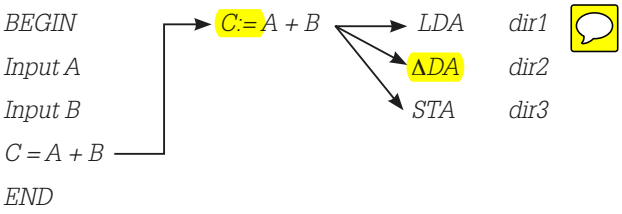
La programación en lenguaje de máquina implica el conocimiento del tipo de instrucciones en **código de máquina** (o **código nativo**) a las que obedece la computadora. Para simplificar la tarea de recordar secuencias binarias tediosas a la hora de programar, se ha desarrollado un lenguaje simbólico de bajo nivel conocido extensamente como **Assembler**. Cada computadora (y sus compatibles) tiene su propio Assembler, que tiene una íntima relación con su diseño. Ciertos autores consideran que su estudio no es muy necesario para abarcar el conocimiento de la computadora en su ámbito físico y lo declaran parte de las materias de programación. Su inclusión obedece a la consideración de que la programación de máquina permite comprender de manera más eficaz el módulo más importante de la computadora, el que entiende instrucciones y genera microoperaciones para su ejecución, llamado unidad de control (CU). Además, según se mencionó, permite el esclarecimiento de un concepto muy importante, que es la interacción entre lo físico y lo lógico, la capacidad del hardware como elemento que soporta e interpreta una secuencia de instrucciones ordenadas de manera lógica: el programa.

En síntesis, si en una computadora se ingresa un programa en lenguaje simbólico de alto nivel que indique:

```
BEGIN
Input A
Input B
C = A + B
Output C
END
```

este programa será convertido en programa ejecutable con la ayuda de un programa "traductor", donde para cada sentencia del programa fuente (o *source*) corresponderán n instrucciones en lenguaje de máquina. En este caso, las sentencias "**entrada A**" y "**entrada B**" solicitan el ingreso de las variables que se han de sumar (que aún no se describirán); la sentencia $C = A + B$ equivale a ordenarle a la computadora que sume la variable A a la variable B y guarde el resultado en la locación de memoria asignada para la variable C. La sentencia "**salida C**" permite mostrar el resultado en un dispositivo de salida.

Si, por el contrario, en la computadora se ingresara sólo la sentencia aritmética para el mismo programa pero en lenguaje simbólico de bajo nivel –como lo es el Assembler–, esto demandaría tres instrucciones para obtener el mismo resultado, en una relación *1* a *1* con las instrucciones en código de máquina:



Tanto el **lenguaje simbólico de alto nivel**, que es aquel que está estructurado de una manera más cercana al hombre, como el **lenguaje simbólico de bajo nivel**, que se generó de una forma más cercana a la computadora, permiten obtener las mismas instrucciones en **lenguaje de máquina**, o sea, las que el procesador "entiende" y puede ejecutar.

En la figura 7.2 se pueden apreciar estas relaciones.

Para un experto en Informática, dos buenas herramientas son el conocimiento de las características de diseño de su computadora y el aprovechamiento de las facilidades del sistema operativo, o sea, maximizar la eficiencia de la computadora.

La programación en lenguaje de máquina permite aumentar las prestaciones del sistema operativo, por ejemplo, al crear rutinas no definidas, que no podrían programarse en los lenguajes de alto nivel. Por lo tanto, debe considerarse satisfactorio el estudio de las instrucciones de máquina que entiende la CPU de su computadora. En este capítulo se creará un "set" o conjunto de instrucciones para el modelo propuesto, pero con otro objetivo: conocer, de la manera más simple, la forma en que estas secuencias binarias generan órdenes a los distintos módulos del hardware en un modelo de computadora básico.

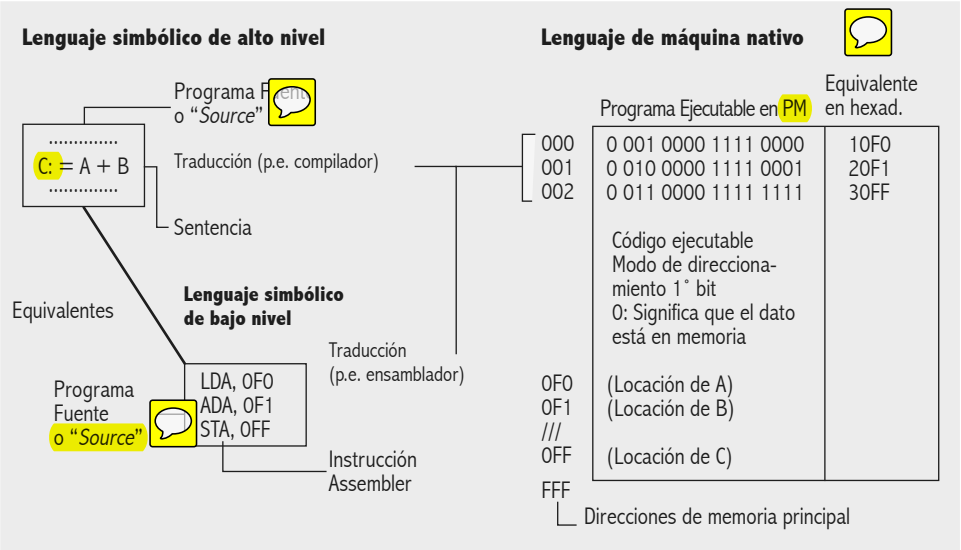


Fig. 7.2. Sentencia de ejemplo.

7.3.1 Instrucciones

Cuando la computadora realiza una tarea compleja, a pedido del usuario, ejecuta una serie de pasos simples representados por su propio juego de instrucciones. Estas instrucciones constituyen su lenguaje de máquina o lenguaje nativo. Como ya se indicó, no es usual que el programador plantee la tarea en términos de secuencias binarias, sino que se utiliza un lenguaje simbólico más orientado a su modalidad de expresión que a la de la computadora. Sin embargo, todo programa que utiliza un lenguaje simbólico debe traducirse a código de máquina antes de su ejecución.

Por el momento no se entrará en detalle respecto de esta herramienta "que traduce" instrucciones simbólicas a instrucciones de máquina. Considérese la notación simbólica como una forma alternativa para representar instrucciones binarias, teniendo siempre presente que la computadora sólo ejecuta códigos de instrucción en lenguaje de máquina.

Así como se establece un código de representación de caracteres, unidades elementales que constituyen los datos (que ingresan, por ejemplo, por teclado), también hay un código de representación de instrucciones, unidades elementales que constituyen los programas.

El **código de una instrucción** es la combinación de bits que la unidad de control de la CPU interpreta para generar las microoperaciones que permitan su ejecución.

La forma de agrupar estos bits en entidades diferenciadas determina la estructura de la instrucción y se define como **formato de la instrucción**. Una misma unidad de control puede "comprender" distintos formatos de instrucción.

El formato de instrucción más simple es el que asigna un grupo de bits para representar una "acción" y otro grupo para representar el "dato" al que afecta esta acción. Como se ve en la figura 7.3, el primer grupo de bits se denomina código de operación (**OPCODE**). La cantidad de bits del COP determina el número de acciones distintas que se podrían definir, según la fórmula siguiente:

"n bits" determinan "2^n códigos de operaciones distintos".

Fig. 7.3. Formato de instrucción.

Por lo general, el segundo grupo de bits hace referencia a un dato en memoria, por lo tanto, determina la dirección de la posición de memoria (locación) donde se aloja el dato. La cantidad de bits debe permitir hacer referencia a cualquier posición de memoria.

Sin embargo, no siempre los datos se encuentran en memoria. Un dato puede estar almacenado en un registro de CPU y, en este caso, los bits de dato deben poder hacer referencia a ese registro. Incluso hay códigos de operación que no afectan dato alguno y, si es así, el segundo grupo de bits puede aparecer como redundante o tomarse como una extensión del código de operación.

Según se indicaba antes, una computadora de propósito general tiene definida la tarea que se ha de realizar según las instrucciones de un programa almacenado en memoria, que es intercambiable. La memoria de lectura/escritura está dividida lógicamente en memoria asignada a programa y memoria asignada a datos y constituye el módulo de almacenamiento de la computadora. La unidad de control lee una instrucción de la memoria, la aloja en un registro interno (que llamaremos en este capítulo registro de instrucción) e interpreta si el código de operación afecta a un dato almacenado en memoria, en cuyo caso provoca su lectura. Cuando una instrucción está alojada en la unidad de control se afirma que está en **estado de ejecución** y su código binario indica dónde está el dato, cuál es la acción que lo afecta y, por lo tanto, qué módulo del hardware la llevará a cabo.

Conocer la naturaleza del juego de instrucciones de máquina es una de las mejores formas de aprovechar (con programas mas simples de ejecutar) la potencia de la computadora y comprender la relación entre los módulos que la constituyen.

En ocasiones se hace difícil encontrar un límite entre las capacidades del hardware y las del software, ya que algunas funciones no definidas por uno pueden ser provistas por el otro. Los diseños actuales muestran la tendencia para desarrollar la mayor cantidad de funciones sobre hardware, porque esto incrementa la velocidad de procesamiento, acompañada de un declive constante de su costo. Hasta principios de la decada del *80*, la tendencia era hacer el hardware más complejo (más funciones en hardware); desde aquellos años hasta hoy, se usa un criterio cuantitativo, que indica que se implementan en hardware las funciones que se utilizan en forma mas frecuente y el resto se implementan en software.

Cada computadora está diseñada para abastecer las necesidades de un grupo determinado de usuarios en el mercado total, lo que implica que las bondades y las limitaciones de un diseño siempre son características relativas a las aplicaciones para las que sirven mejor. Computadoras muy buenas para aplicaciones comerciales pueden ser bastante inútiles para aplicaciones científicas. Organizar el diseño del hardware de la computadora y de su software de base es una empresa cuyo objetivo es lograr la mayor eficiencia a menor costo en el mercado al que apunta.

Von Neuman (1903-1957). Matemático húngaro-estadounidense que realizó contribuciones importantes en Física cuántica, análisis funcional, teoría de conjuntos, Informática, Economía, análisis numérico, Hidrodinámica, Estadística y muchos otros campos de la Matemática.
Fue pionero de la computadora digital moderna y publicó un artículo acerca del almacenamiento de programas. El concepto de programa almacenado permitió la lectura de un programa dentro de la memoria de la computadora y, después, la ejecución de las instrucciones del mismo, sin tener que volverlas a escribir. EDVAC (*Electronic Discrete Variable Automatic Computer*) fue la primera computadora que usó este concepto desarrollado por Von Neumann, Eckert y Mauchly. Los programas almacenados dieron a las computadoras flexibilidad y confiabilidad, haciéndolas más rápidas y menos sujetas a errores que los programas mecánicos.

7.4 Presentación del modelo de estudio

La presentación del material aquí expuesto es de lectura necesaria si se desea conocer un diseño completo, aunque elemental, de una computadora. La computadora presentada no es un modelo concreto del mercado actual, sino que fue pensada para que usted se acerque lo suficiente, y de la forma más simple, a las características de diseño de la mayoría de las computadoras. La disposición esquemática de los componentes no tiene relación con su emplazamiento físico real; el tamaño de los bloques en cada figura no sigue una escala fija, incluso el tamaño mayor de alguno de ellos respecto de los demás, sólo pretende resaltar sus características.

Se decidió llamar "X" a esta computadora de propósito general, cuyo diseño se observa en la figura 7.4 y encuadra en los parámetros definidos para una arquitectura Von Neumann.

"X" tiene un módulo de almacenamiento de lectura/escritura de *4* Kpalabras de *16* bits cada una, con acceso aleatorio a cada palabra identificada por una dirección de *12* bits que se notará siempre en hexadecimal, de modo que el rango de direcciones varía entre *000H* y *FFFH (0-4095)*. La memoria está dividida "lógicamente" en memoria asignada a programa y memoria asignada a datos. "X" puede ejecutar un programa por vez, o sea que su modalidad de procesamiento es la monoprogramación.

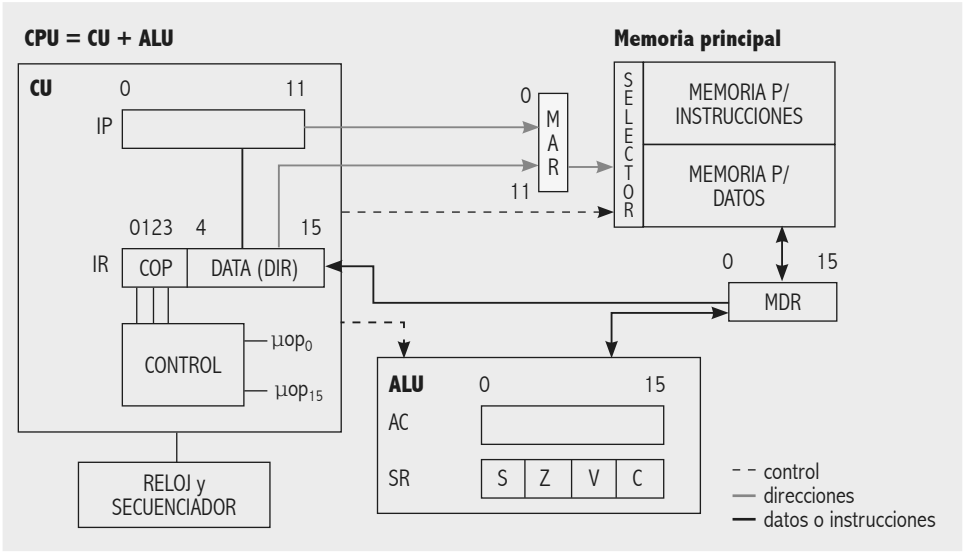


Fig. 7.4. Arquitectura básica de "X".

El modelo plantea cómo evoluciona la ejecución de las instrucciones sin mencionar cómo llegaron a la memoria desde el exterior.

Formato de datos:

Los datos son del tipo enteros signados de *16* bits (*1* para el signo y *15* para la magnitud), según se puede observar en la figura 7.5.

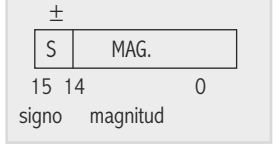


Fig. 7.5. Formato de datos.

Formato de instrucción:

Las instrucciones se almacenan en palabras consecutivas a partir de la palabra *000H*. El código de instrucción es de *16* bits, con un formato único de instrucción, el más simple, representado en la figura 7.6, donde los primeros *4* bits definen el código de operación, y los *12* restantes definen la dirección de un dato (en memoria asignada a datos), de lo contrario, son ignorados.

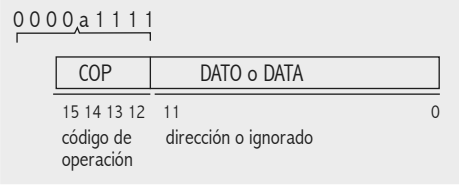


Fig. 7.6. Formato de instrucción de la CPU de "X".

La CPU de "X" (unidad central de procesamiento) se encarga de buscar las instrucciones de la memoria, interpretarlas y gestionar su ejecución, mientras que los datos son operados aritmética o lógicamente en la ALU.

La función de la CPU se puede separar en dos partes:

1. Tratamiento de instrucciones.
2. Operación de los datos.

Del tratamiento de instrucciones se encarga un conjunto de componentes denominado **unidad de control** (CU o *Control Unit*), sincronizado por el generador de pulsos de reloj, y de la operación de los datos se encarga, como se indicó, la **unidad aritmético-lógica** (ALU o *Aritmetic Logic Unit*). De modo que en la figura 7.7 se muestra un esquema simplificado de "X".

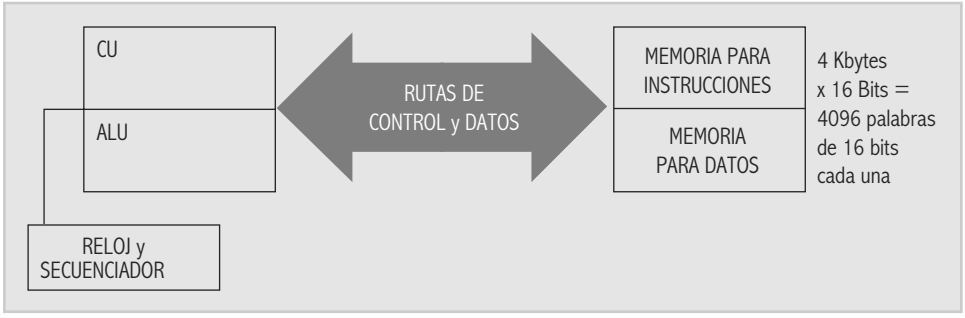


Fig. 7.7. Esquema simplificado de "X".

La memoria de "X" no necesita más detalle del que se aportó. El medio de comunicación entre la memoria y la CPU es un bus que transmite órdenes y datos, interpretando como bus de datos el camino que permite la transferencia de grupos de bits que identifican el contenido de una posición de memoria.

La CPU merece una dedicación especial, ya que su función es gestionar, precisamente, el procesamiento de información. Consideremos primero cómo toma la CU las instrucciones del programa almacenado en memoria para interpretarlas y ejecutarlas. Este proceso se puede dividir en las etapas siguientes:

1. Búsqueda de la instrucción en memoria.
2. Interpretación del código de instrucción.

3. Búsqueda del dato afectado (si afecta a dato) por la instrucción.
4. Generación de órdenes al módulo que opera sobre ese dato.

La etapa *1* también se denomina **fase de búsqueda** o **fase *fetch***, mientras que las otras tres se agrupan en la llamada **fase de ejecución** o ***execute***. Se puede afirmar que una vez que la computadora comienza a funcionar, su CPU se encontrará siempre en una de estas dos fases. Para organizar el estudio de estas etapas, la atención se debe centrar primero en qué elementos intervienen en la gestión y luego en cómo operan para llevarla a cabo.

7.4.1 Fase *fetch*: búsqueda de una instrucción en memoria

Cuando la CU ejecuta cada instrucción de un programa debe alternar sus etapas *fetch* (f) y *execute* (e) desde la primera instrucción hasta la última, esto es, $f_{(I_0)}$, $e_{(I_0)}$; $f_{(I_1)}$, $e_{(I_1)}$; .....; $f_{(In)}$, $e_{(In)}$.

La secuencia del ciclo $f_{(Ii)}$; $e_{(Ii)}$ se denomina **ciclo de instrucción**.

Para cada fase $f_{(Ii)}$ la CU debe enviar a la memoria la dirección de la palabra donde se encuentra la instrucción, una orden de lectura y una orden de transferencia de la instrucción a la CU. La CU retiene la dirección de la instrucción en un registro especial denominado "puntero de instrucción" (IP o *Instruction Pointer*) o "contador de programa" (PC o *Program Counter*). Considérese este registro como el señalador de páginas del libro que usted está leyendo; el señalador le permite cerrar el libro en forma distraída y luego retomar en la página correcta. El IP cumple la misma función al permitir que la CU "se distraiga" de la secuencia de la próxima instrucción del programa que se ha de ejecutar. La longitud del IP depende de la cantidad de bits que se necesiten para direccionar cualquier instrucción en la memoria asignada al programa.

Considérese que para "X" una instrucción I_n podría, "supuestamente," alojarse en la palabra *FFFH* que es la última, por lo tanto, el IP mide *12* bits como máximo, que permiten 2^{12} direcciones numeradas de *0* a $4095_{(10)}$. O sea que este puntero puede hacer referencia a las *4*K direcciones de memoria, la última de ellas se representa con doce *1* binarios (*FFFH*).

Se indicó que I_o siempre se carga en la palabra *000H* (esta condición es válida para "X", pero no para todas las computadoras), de modo que éste es el valor inicial del IP cuando se arranca el funcionamiento de la computadora. Por otra parte, las instrucciones se almacenan en palabras sucesivas y ocupan en este caso una sola palabra; por esta razón, el IP debe poder incrementarse en una unidad para señalar siempre la próxima instrucción para buscar, luego de una ejecución. Más adelante veremos que el IP también puede aceptar un valor cualquiera impuesto por una instrucción que provoca una ruptura de secuencia en el programa. En la figura 7.8 se muestran las posibilidades de actualización y función de este registro.

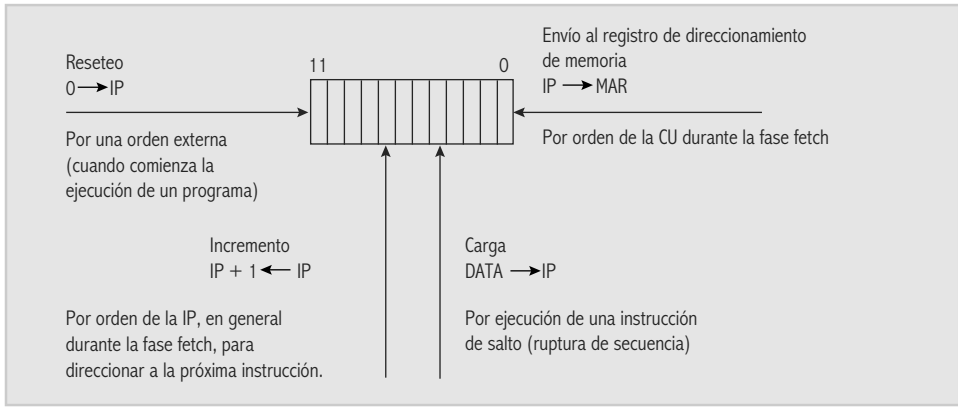


Fig. 7.8. Registro IP.

Una vez que la CU envía a memoria el contenido del IP, da la orden de lectura para que la palabra implicada, que llamaremos *W,* se almacene en el registro de palabra de memoria (MDR o *Memory Data Register*).Usualmente, la unidad de acceso a memoria es el bit; aquí la computadora "X" tiene una memoria organizada por palabras de *2* bytes y la unidad de direccionamiento es entonces la palabra (W).

Por último, la etapa *fetch* termina con la transferencia de la instrucción leída a un registro interno de la CU, donde la instrucción permanece almacenada mientras dure su ejecución. Este registro se denomina registro de instrucción (IR o *Instruction Register*) y su capacidad soporta el conjunto de bits del código de instrucción ejemplificado, en este caso, *16* bits.

Como se observa en la figura 7.9, IR se relaciona directamente con el hardware que genera microoperaciones de ejecución y que denominaremos CU.

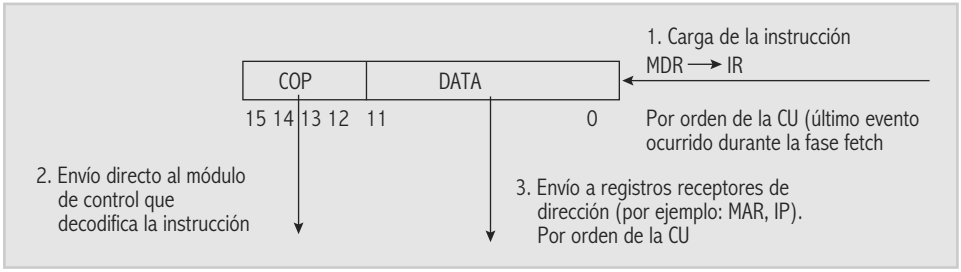


Fig. 7.9. Código de instrucción.

En este caso, la interpretación del COP la realiza un decodificador $4 \cdot 2^4 = 4 \cdot 16$, lo que significa que con *4* dígitos binarios se generan *16* instrucciones referenciadas por una de las salidas del circuito COP_i.

La zona denominada DATA es de *12* bits; se relaciona con el MAR cuando se debe tomar un dato de memoria y con el IP cuando se debe romper la secuencia normal del programa. Esta zona se ignora cuando la instrucción no afecta dato alguno. La CU actualiza el IP y entra en la fase *execute* de la instrucción buscada. En una *flag* llamada F (fase) se retiene un "*1*" durante la fase de búsqueda que cambia a *0* cuando comienza la fase de ejecución. Su función es importantísima para la CU de "X", porque le permite alternar de una fase a la otra, inhibiendo órdenes de búsqueda durante la fase de ejecución, y al revés.

Ahora que se conocen los dos registros de la CU de "X" implicados en la búsqueda de la instrucción, veremos quién y cómo se realiza su operación. En el caso de "X", el que genera la secuencia de microoperaciones de las fases *fetch* y *execute* es el circuito denominado control de instrucción. Aunque su nombre lo asocia a la segunda fase, gestiona la fase *fetch* comandado por el valor "*1*" de la *flag* F y por señales de tiempo que le permiten sincronizar las microoperaciones de búsqueda.

Por el momento no se entrará en detalle respecto de la estructura interna del control ni del sistema de reloj, pero se debe considerar que las microoperaciones obviamente no pueden enviarse simultáneamente. Por lo tanto, los componentes de hardware implicados en la fase *fetch* y la secuencia de sus eventos, que se muestran en la figura 7.10, son los siguientes:

1. *IP* ⟶ *MAR*
2. *Word* ⟶ *MDR (por orden de lectura)*
3. *MDR* ⟶ *IR*
4. *IP + 1* ⟶ *IP*
5. *0* ⟶ *F*

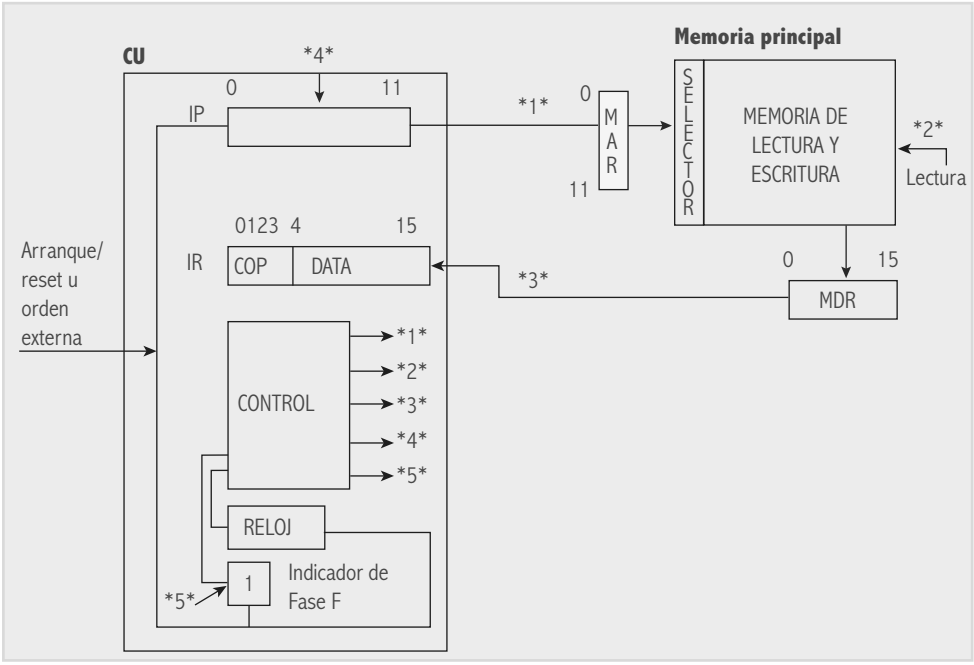


Fig. 7.10. Fase *fetch*.

F es una *flag* cuyo valor realimenta como información de entrada al control, de modo de inhibir la búsqueda de una instrucción nueva a partir del evento *5* de la fase *fetch* y, de esta manera, delimitar esta fase con la de ejecución.

Las microoperaciones enumeradas antes definen la fase *fetch* y se producen cuando el biestable *F* está en *1* en los pulsos de reloj t_i, t_{i+1}, ..., t_{i+n}.

La primera es una transferencia del contenido del puntero IP al MAR para seleccionar la palabra que se ha de leer.

La segunda es una orden de lectura a la memoria para hacerla efectiva.

La tercera es una transferencia de la instrucción al IR.

La cuarta corresponde a la actualización del IP para que señale la próxima instrucción para ejecutar.

La quinta corresponde a la actualización del indicador de fase F para entrar en fase *execute* (nótese que la cuarta microoperación puede producirse al mismo tiempo con la segunda o la tercera, porque no afecta los mismos registros; no así con la primera, porque se corre el riesgo de incrementar el IP antes de acceder a memoria). Si se quiere tener una idea de la estructura de hardware del control que gestiona este nivel de microoperaciones, la forma más simple es armar un circuito cuyas funciones de salida dependan de las variables de tiempo t_i y de la variable de fase F.

7.4.1.1. Diseño parcial del módulo del control asociado a la fase fetch

Las microoperaciones para completar esta fase se encuentran definidas en el control, como se verifica en la figura 7.11. El método que utilizaremos para el diseño final del control es el de asociar, mediante compuertas lógicas variables de tiempo, variables de estado de la CU y variables generadas por la instrucción para ejecutar.

Para este ejemplo se debe asumir que el tiempo de respuesta de memoria es igual a cuatro pulsos del reloj (*4* t_1) y que ninguna ejecución de instrucción supera los ocho tiempos del ciclo de la computadora.

Como ya se mencionó, en la fase de búsqueda el conjunto de compuertas depende sólo de dos variables: la de tiempo y la de control de fase; esta última en su estado *1*, por lo tanto:

$f_1 = F \cdot t_0$ $\quad IP \longrightarrow MAR$

$f_2 = F \cdot t_1$ $\quad W \longrightarrow MDR$

$f_3 = F \cdot t_5$ $\quad MDR \longrightarrow IR$

$f_4 = F \cdot t_1$ $\quad IP + 1 \longrightarrow IP$

$f_5 = F \cdot t_7$ $\quad 0 \longrightarrow F$

f_4 se solapa con f_2 ya que es una microoperación independiente de la anterior

Nótese que el tiempo que transcurre entre la orden de lectura (f_2) y la transferencia del contenido de la palabra (f_3) coincide con el tiempo de respuesta de la memoria. En el tiempo t_7 se resetea la *flag* de estado de la CU para forzar la etapa de ejecución de la instrucción.

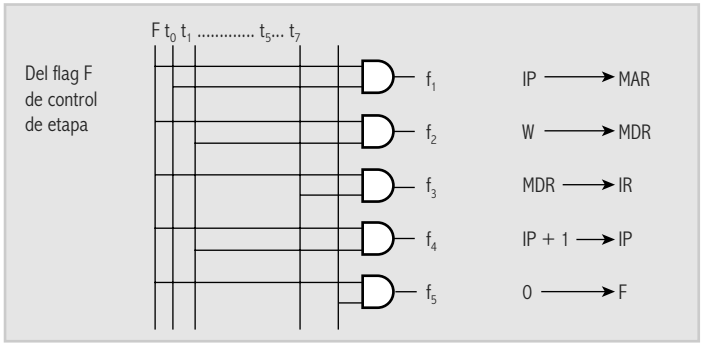


Fig. 7.11. Estructura de una parte del módulo de control para la fase *fetch*.

Ahora es oportuno que se analice cómo se comporta el control cuando decodifica las instrucciones del programa y genera las microoperaciones de ejecución.

Supóngase que el programa almacenado es muy elemental: sumar dos datos almacenados en memoria en las palabras *A0A* y *A0B* y calcular su promedio, almacenando el resultado en la palabra *A0C*.

Veamos cómo se resolvería si se propone como ejercicio práctico el mismo enunciado para realizar bajo las condiciones siguientes: *A* y *B* son binarios enteros signados que, representados en hexadecimal, asumen los valores *0004H* y *0002H*, respectivamente. La única herramienta de trabajo es una calculadora binaria, cuya capacidad máxima es de *16* bits, con un solo visor de *16* bits y un registro interno (que no se ve) también de *16* bits (en el caso de las calculadoras, el contenido del visor pasa al registro interno y se visualiza el segundo operando, mientras que en el caso de la computadora, el registro interno retiene el segundo operando). El teclado tiene dos teclas de entrada de dato (*0* y *1*), suma, desplazamiento a derecha (para dividir por la base sucesivamente) y cuatro luces que se prenden según el estado final de la operación.

En la figura 7.12 se observa un ejemplo de operación con una calculadora elemental.

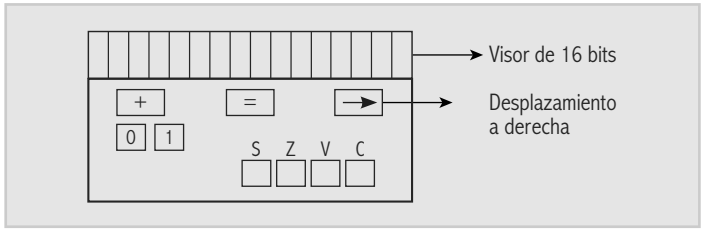
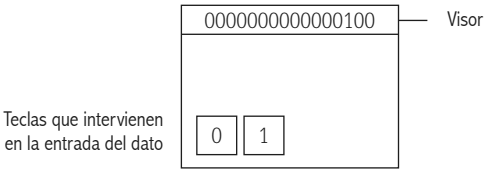
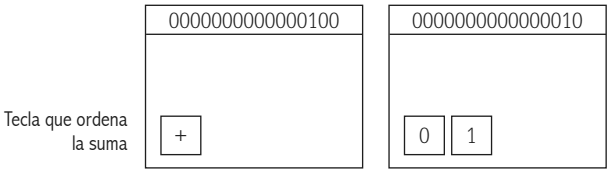


Fig. 7.12. Calculadora elemental.

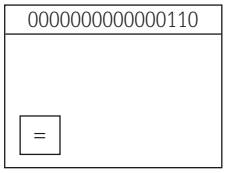
Su primera acción es ingresar el primer valor (*A = 0004*).



Su segunda acción es presionar la tecla "suma" e ingresar el segundo valor (*B = 0002*).



Al presionar la tecla "+"se obtiene:



El estado de las luces le indicará ciertas condiciones de la operación que podrá tomar en cuenta o no:

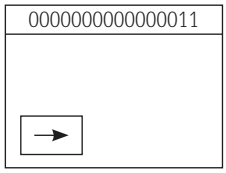
- La luz S se prende si el resultado de la operación fue negativo.
- La luz Z se prende si el resultado de la operación fue cero.
- La luz V se prende si el resultado excedió la capacidad del visor.
- La luz C se prende si el último acarreo,"el que me llevo", fue igual a *1*.

Por lo tanto, para el juego de valores operados todas las luces permanecerán apagadas.

C *0 0*

	0000 0000 0000 0100	*V = 0*
+	*0000 0000 0000 0010*	*Z= 0*
	0000 0000 0000 0110	*C = 0*
S	***Resultado ≠ 0***	*S = 0*

Su última acción será presionar la tecla de desplazamiento a derecha y verá el resultado final.



Esta calculadora elemental permite realizar operaciones muy simples. Esto se debe a su escasa cantidad de teclas "de control" (suma, igual y desplazamiento), por lo que se deduce que cuanto más deficiente sea la capacidad de "hardware", tanto más acciones serán necesarias para realizar lógicamente tareas complejas.

Esta calculadora pretende dar una noción del dispositivo de cálculo de nuestra computadora "X", su unidad aritmético-lógica. Llamemos acumulador a un registro de *16* bits que cumple la función del visor. Las teclas *0* y *1* no tienen sentido práctico, ya que los datos son traídos de memoria; las teclas *+*, ÷, → son órdenes que recibe la ALU por parte del control.

Las luces podrán, eventualmente, ser consultadas y permitirán que el programa lleve un control de qué ocurrió con la operación. Las llamaremos *flags* o banderas de estado; si el bit almacenado es *0*, la condición no se satisface; si el bit almacenado es *1*, sí. Ahora cada acción se transforma en una instrucción del programa P. A cada instrucción se le asigna un nombre simplificado y un código binario que se mostrará en hexadecimal.

	Nombre	*Código (representa al código de máquina)*
- Cargar el acumulador con la palabra A0A	LDA A0A *Load accumulator*	1A0A
- Sumar al acumulador la palabra A0B	ADA A0B *Add accumulator*	2A0B
- Desplazar el acumulador	SHR *Shift right*	AXXX
- Almacenar el acumulador en la palabra A0C	STA A0C *Store accumulator*	3A0C
- Fin del programa	HLT	0XXX

En la primera columna se detalla verbalmente cada instrucción; en la segunda se le asigna un mnemónico de tres letras, (se respetaron los mnemónicos más empleados en la descripción simbólica de instrucciones); la tercera columna muestra el código binario, que es, en definitiva, el código que entiende el control de "X", o código ejecutable.

7.4.2 Fase *execute*

Una vez que se hace efectiva la búsqueda de alguna de las cuatro instrucciones de *P*, la CU entra en fase de ejecución *(0—F)*. Cada código de instrucción tiene asignada una secuencia de microoperaciones definida en el control. Veamos qué sucede entonces en el estado de ejecución de cada una de ellas.

LDA A0A

La interpretación de su COP (*0001*) permite la transferencia del dato almacenado en la palabra *A0A* al acumulador de la ALU.

Las dos primeras microoperaciones hacen efectiva la lectura del dato de memoria y la tercera procede a la carga en sí.

Convengamos en representar los valores binarios en hexadecimal para mejorar la claridad de los ejemplos (fig. 7.13).

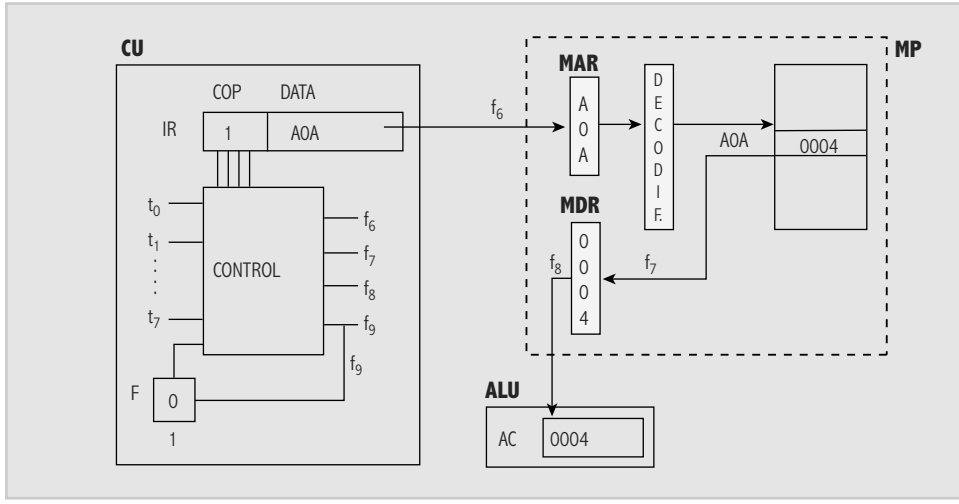


Fig. 7.13. Fase *execute* LDA.

$f_6 = \overline{F} \cdot t_0 \cdot COP_1$ $\quad DATA \rightarrow MAR$

$f_7 = \overline{F} \cdot t_1 \cdot COP_1$ $\quad WORD \rightarrow MDR$

$f_8 = \overline{F} \cdot t_5 \cdot COP_1$ $\quad MDR \rightarrow AC$

$f_9 = \overline{F} \cdot t_7 \cdot COP_1$ $\quad 1 \rightarrow F$

COP_1 es la variable que representa al código de operación 0001

$\overline{F}$ es la variable que representa no fetch, esto es, execute

f_6: Microoperación que habilita la transferencia de la dirección del dato indicada por la instrucción al registro de direccionamiento de memoria.

f_7: Microoperación que habilita la lectura de la unidad de memoria.

f_8: Microoperación que habilita la transferencia del dato leído al acumulador.

f_9: Microoperación que actualiza la *flag* controladora de fase y que provoca que el control se "entere" que debe producir microoperaciones de búsqueda para la instrucción siguiente.

ADA A0B

En el esquema para la ejecución de esta instrucción se incluye como dispositivo de cálculo un sumador binario paralelo, que se analizó con anterioridad en el capítulo Álgebra de Boole. Los bits A_i corresponden al dato almacenado en el acumulador (primer operando). Los bits B_i corresponden al dato leído de la palabra *A0B*, que permanece en el MDR (segundo operando).

Las salidas S_i están listas cuando la microoperación final habilita al acumulador como receptor de la suma, la misma microoperación habilita al *status register* para que se actualice.

Como en la instrucción anterior, las dos primeras microoperaciones permiten la lectura del dato almacenado en la palabra *A0B* (fig. 7.14).

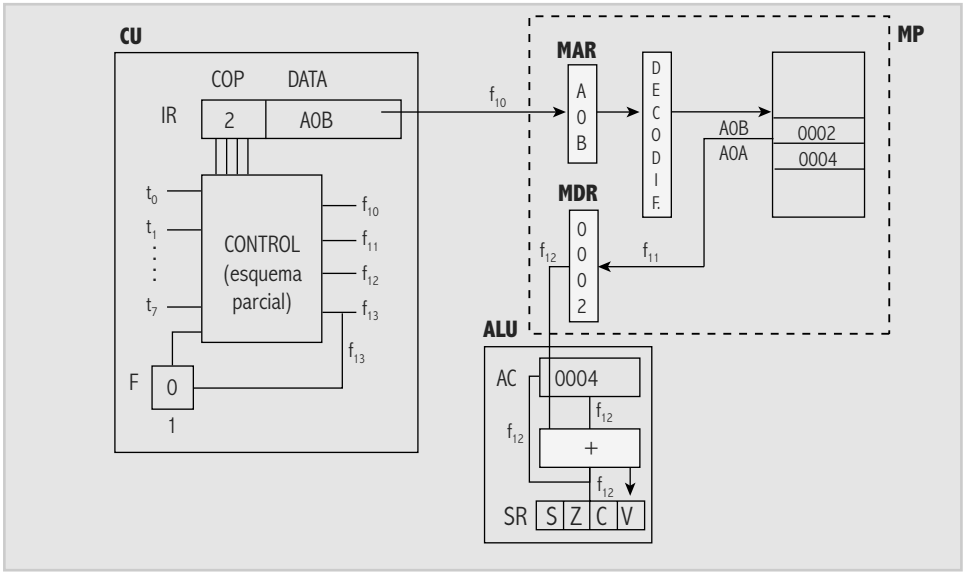


Fig. 7.14. Fase *execute* ADA.

$$f_{10} = \overline{F} \cdot t_0 \cdot COP_2 \quad DATA \longrightarrow MAR$$
$$f_{11} = \overline{F} \cdot t_1 \cdot COP_2 \quad WORD \longrightarrow MDR$$
$$f_{12} = \overline{F} \cdot t_5 \cdot COP_2 \quad MDR + AC \rightarrow AC$$
$$f_{13} = \overline{F} \cdot t_7 \cdot COP_2 \quad 1 \longrightarrow F$$

f_{10}. Similar a la microoperación f_6.

f_{11}. Similar a la microoperación f_7.

f_{12}. Microoperación que habilita al acumulador como receptor del resultado (*0006H*) y al SR, o *status register,* como receptor de la información del estado final de la operación (signo, cero, *carry* y *overflow*).

f_{13}. Similar a la microoperación f_9.

Nótese que en este ejemplo el MDR actúa como registro interno o segundo registro de la calculadora.

SHR

El control genera una microoperación que ordena al acumulador desplazar su información un bit a derecha (*shift right*). Incluir esta instrucción en este punto del programa implica lograr dividir el resultado almacenado en AC por dos, para calcular el promedio de los datos sumados.

Esta única microoperación afecta una entrada especial del registro acumulador que hace posible habilitar la función "desplazar". Más adelante se verá que ésta es una de las varias funciones propias que tiene este registro (fig. 7.15).

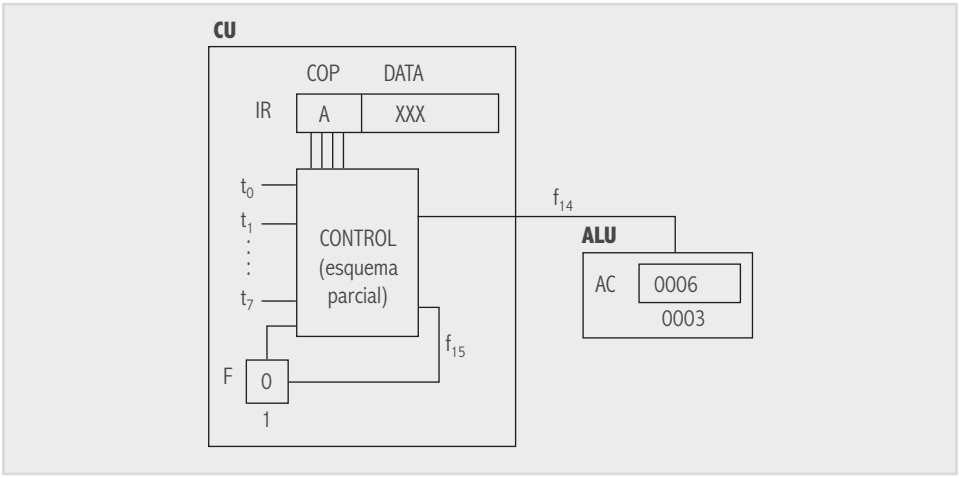


Fig. 7.15. Fase *execute* SHR.

$$f_{14} = \overline{F} \cdot t_0 \cdot COP_{10} \qquad \overrightarrow{AC} \longrightarrow AC$$
$$f_{15} = \overline{F} \cdot t_7 \cdot COP_{10} \qquad 1 \longrightarrow F$$

f_{14}. Microoperación que permite desplazar la información del AC una posición a la derecha, con lo que divide por la base.

f_{15}. Similar a la microoperación f_9.

STA A0C

Esta instrucción se incluyó en el programa para almacenar el resultado final en memoria.

El control genera microoperaciones que permiten la escritura del promedio calculado en la palabra de memoria *A0C* (fig. 7.16).

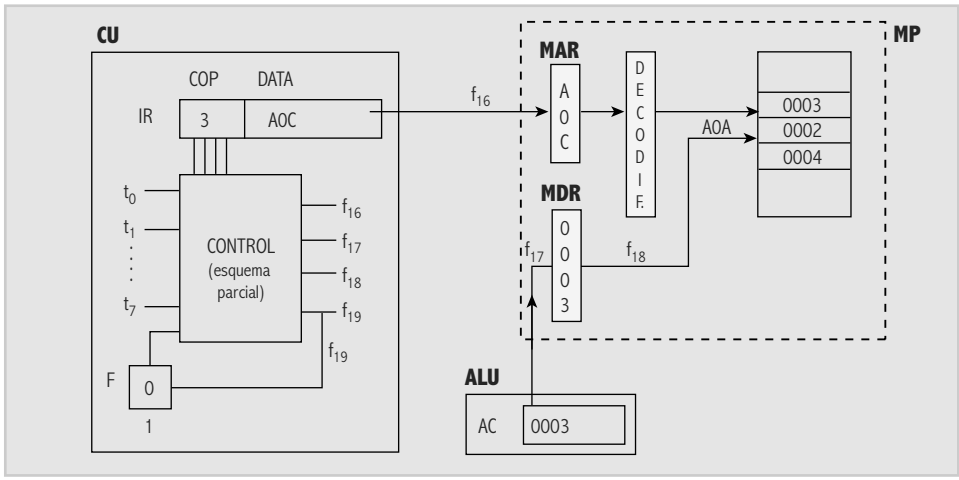


Fig. 7.16. Fase *execute* STA.

$f_{16} = \overline{F} \cdot t_0 \cdot COP_3$ $\quad DATA \longrightarrow MAR$

$f_{17} = \overline{F} \cdot t_1 \cdot COP_3$ $\quad AC \longrightarrow MDR$

$f_{18} = \overline{F} \cdot t_2 \cdot COP_3$ $\quad MDR \longrightarrow WORD$ *Tiempo necesario de escritura (el mismo asignado a la lectura)*

$f_{19} = \overline{F} \cdot t_7 \cdot COP_3$ $\quad 1 \longrightarrow F$

f_{16}. Microoperación que habilita la transferencia de la dirección de la palabra donde se va a escribir el dato, indicada por la instrucción, al registro de direccionamiento de memoria.

f_{17}. Microoperación que habilita la transferencia del dato almacenado en el acumulador al registro de palabra de memoria.

f_{18}. Microoperación que habilita la escritura en la unidad de memoria.

f_{19}. Actualización de la *flag* F para entrar en una nueva fase de búsqueda.

($1 \text{—} \overline{F}$).

HLT

La instrucción HLT (*halt* = parada) inhibe la generación de microoperaciones dejando a "X" suspendida. Esta instrucción es muy importante, puesto que de no existir se produce una etapa *fetch,* considerando la siguiente posición de memoria como una instrucción. Esto implica que el programa continúe de manera imprevista y provoque el fracaso de su ejecución. Para el modelo presentado, la ejecución de HLT genera una sola microoperación f_{20} que inactiva la generación de secuencias de tiempo t_i, con lo que se logra el efecto buscado. Todas las microoperaciones dependen de algún t_i, por lo tanto, si $t_i = 0$, todas las funciones de salida quedan en cero (fig. 7.17).

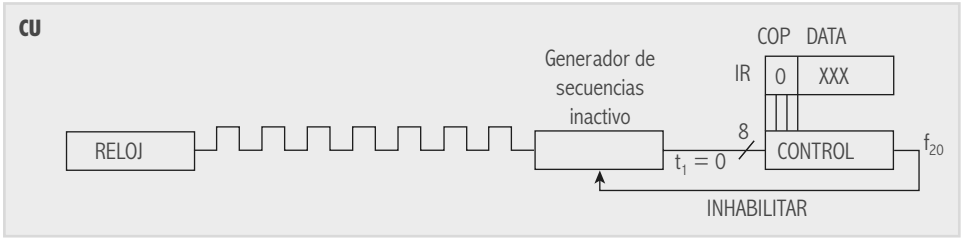


Fig. 7.17. Generador de unidades de tiempo t_i.

7.4.3 Flujo de datos entre los registros de la computadora básica

Por el momento, observe en las siguientes tablas el flujo de datos entre los registros de la computadora "X". Cada etapa de ejecución es precedida por la correspondiente etapa de búsqueda de la instrucción. En la tabla 7-1, que contiene el esquema de la evolución de ejecución de las instrucciones en forma detallada, y en la tabla 7-2, con un esquema simplificado por etapas, se indica cuál fue el efecto sobre los registros implicados.

Aclaraciones:

X: indica estado desconocido.

F: contenido de la *flag* indicadora de fase.

PC: contenido del *Program Counter* (PC) o contador de programa.

MAR (o *Memory Address Register*): contenido del registro de direccionamiento de memoria.

MDR: contenido del registro de palabra de memoria.

AC (o *Accumulator*): contenido del acumulador.

SZVC (SR): estado de las *flags* o banderas contenidas en el *status register*.

A0A: contenido de la palabra de memoria A0A.

A0B: contenido de la palabra de memoria A0B.

A0C: contenido de la palabra de memoria A0C.

000: contenido de la palabra de memoria *000*.

001: contenido de la palabra de memoria *001*.

002: contenido de la palabra de memoria *002*.

003: contenido de la palabra de memoria *003*.

W: WORD indica, en forma genérica, una palabra de memoria cualquiera.

Tabla 7-1. Esquema de la evolución de ejecución de las instrucciones.

	F	*PC*	*IR*	*MAR*	*MDR*	*AC*	*SZVC*	*A0A*	*A0B*	*A0C*	*000*	*001*	*002*	*003*	*004*	*H*
Estado inicial	*1*	*000*						*0004*	*002*		1A0A	2A0B	AXXX	3A0C	0XXX	*0*
IP → MAR W → MDR MDR → IR IP+1 → IP 0 → F	*0*	*001*	1A0A	*000*	1ADA											
DATA→ MAR W → MDR MDR → AC 1 → F	*1*			A0A	*0004*	*0004*										
IP → MAR W → MDR MDR → IR IP+1 → IP 0 → F	*1*	*002*	2A0B	*001*	2A0B											
DATA → MAR W → MDR MDR+AC→ AC 1 → F	*1*			A0B	*0002*	*0006*	*0000*									
IP → MAR W → MDR MDR → IR IP+1 → IP 0 → F	*0*	*003*	AXXX	*002*	AXXX											
AC → AC 1 → F	*1*					*003*										
IP → MAR W → IR MDR → IR IP +1 → IP 0 → F	*0*	*004*	3A0C	*003*	3A0C											
DATA → MAR AC → MDR MDR → W 1 → F	*1*			A0C	*0003*					*003*						
IP → MAR W → MDR MDR → IR IP+1 → IP 0 → F	*1*	*005*	0XXX	*004*	0XXX											

Tabla 7-2. Esquema simplificado por etapas.

	IP	*IR*	*MAR*	*MDR*	*AC*	*SR*	*F*
Estado inicial	*000*	XXXX	XXX	XXXX	XXXX	x	*1*
Fetch	*001*	1A0A	*000*	1A0A			*0*
Execute LDA			A0A	*0004*	*004*		*1*
Fetch	*002*	2A0B	*001*	2A0B			*0*
Execute ADA			A0B	*0002*	*006*	*0*	*1*
Fetch	*003*	AXXX	*002*	AXXX			
Execute SHR			No afecta a mem.		*003*	*0*	*1*
Fetch	*004*	3A0C	*003*	3A0C		*0*	
Execute STA			A0C	*0003*		*0*	
Fetch	*005*	Instrucc de fin	*004*	Instrucc de fin		X	

7.4.4 Juego completo de instrucciones de "X"

Para cubrir las necesidades de programación de "X" a continuación se presentan otras instrucciones y los enunciados de algunas tareas de aplicación de ellas.

"X" tiene tres tipos de instrucciones definidos con claridad:

1. Las instrucciones que afectan datos almacenados en la memoria. Por ejemplo, LDA *A0A*.
2. Las instrucciones que afectan datos almacenados en registros que no pertenecen a la memoria (*AC, STATUS REGISTER*, etc.). Por ejemplo, SHR.
3. Las instrucciones que permiten la entrada de un dato a "X" desde el exterior, o la salida de un dato desde "X" al exterior. Por ejemplo, INP.

Para la representación de instrucciones se adopta un lenguaje que permite que el usuario exprese sus programas, sin necesidad de recordar el código binario asignado a cada una de las instrucciones. Este lenguaje asume, por convención, un mnemónico de tres letras para definir el COP y tres dígitos hexadecimales para definir la dirección del operando en memoria en aquellas instrucciones de referencia a memoria. En la tabla 7-3 se muestra la expresión simbólica, la representación binaria en hexadecimal, el código binario ejecutable y las microoperaciones de ejecución.

Algunas de las instrucciones de la tabla 7-3 fueron analizadas para la ejecución del programa P, las demás, tal vez requieran una explicación adicional.

JMP HHH Es una instrucción de salto incondicional, o sea que usted debe usarla cuando necesite romper el orden secuencial del programa. HHH es la dirección de la palabra de la instrucción a la que quiere saltar. Por eso el control transfiere directamente esta dirección al PC, de modo que la próxima etapa de búsqueda lo encuentre actualizado.

ANA HHH Es una instrucción que permite asociar al acumulador y al dato obtenido mediante el operador lógico AND, de modo que el contenido final del acumulador sea el resultado de la operación "producto lógico" para cada par de valores bit a bit.

XOA HHH Es una instrucción que permite asociar al acumulador y al dato obtenido mediante el operador lógico OR EXCLUSIVE, de modo que el contenido final del acumulador sea el resultado de la operación "suma exclusiva" para cada par de valores bit a bit.

Tabla 7-3. Microoperaciones de ejecución.

	Símbolo	*Taquigráfico decimal*	*Ejecutable bniario*	*Microoperaciones de ejecución*
Referencia a memoria	LDA HHH	1HHH	0001bbbbbbbbbbbb	DATA → MAR W → MDR MDR → AC
	ADA HHH	2HHH	0010bbbbbbbbbbbb	DATA → MAR W → MDR MDR → AC
	STA HHH	3HHH	0011bbbbbbbbbbbb	DATA → MAR AC → MDR MDR → MW
	JMP HHH (*jump*)	4HHH	0100bbbbbbbbbbbb	DATA → IP
	ANA HHH (*and acumulator*)	5HHH	0101bbbbbbbbbbbb	DATA → MAR W → MDR MDR^AC → AC
	XOA HHH (*or exclusive acumulator*)	6HHH	0110bbbbbbbbbbbb	DATA → MAR W → MDR MDR + AC → AC
Referencia a registro	HLT	0XXX	0000XXXXXXXXXXXX	1 → H
	CLA	7XXX	0111XXXXXXXXXXXX	0 → AC
	CMA	8XXX	1000XXXXXXXXXXXX	AC → AC
	INC	9XXX	1001XXXXXXXXXXXX	AC + 1 → AC
	SHR	AXXX	1010XXXXXXXXXXXX	AC → AC
	SNA	BHHH	1011bbbbbbbbbbbb	SI AC < 0 → DATA → IP
	SZA	CHHH	1100bbbbbbbbbbbb	SI AC = 0 → DATA → IP
	SCA	DHHH	1101bbbbbbbbbbbb	SI C_{15} = 1 → DATA → IP
Entrada y salida	INP	EXXX	1110XXXXXXXXXXXX	ENT → AC, 0 → FEN
	OUT	FXXX	1111XXXXXXXXXXXX	AC → SAL, 0 → FSA

HLT Es la instrucción que determina el fin del programa (*halt*). Es imprescindible para que no se sigan buscando posiciones de memoria que no almacenan instrucciones, evitando así resultados imprevistos. "X" lo logra al inhibir la generación de señales de tiempo.

Se ha considerado apropiado asignarle el valor *0*, de modo que si por error se ejecuta una instrucción almacenada en una palabra de memoria puesta a *0*, el control genera la misma microoperación que para ejecutar HALT. Ésta y todas las instrucciones de referencia a registro o de E/S ignoran el campo del código de instrucción DATA.

CLA Permite la puesta a *0* del acumulador (*clear accumulator*).

CMA Permite la inversión de los bits del acumulador.

Calcula el complemento restringido del dato almacenado en el acumulador (*complement accumulator*).

INC Permite sumar una unidad al dato cargado en el acumulador (*increment accumulator*).

SHR Permite desplazar un bit del dato almacenado en el acumulador.

En el caso de SHR (*shift right*) el desplazamiento se produce a derecha.

SNA HHH Es una instrucción de salto condicionado, que permite tomar una decisión en el programa, en función de una condición preestablecida. En este caso, la condición es que el dato almacenado en el acumulador sea negativo, esto es, que para efectuar el salto la instrucción considera el valor *1* de la *flag* "S" en el registro de estado. Si *S = 1*, entonces, modifica el contenido actual del PC con la dirección de salto especificada en el campo DATA, lo que provoca que la próxima fase *fetch* no busque la instrucción siguiente al SNA, sino la indicada en esta instrucción. Si *S = 0*, significa que el dato almacenado en el acumulador no es negativo, por lo tanto, no se satisface la condición y no se produce ninguna microoperación de ejecución, continuando la secuencia normal del programa.

SZA HHH Es una instrucción de salto condicional igual que las anteriores. En este caso, la condición que se ha de verificar es que el dato almacenado en el acumulador sea *0*, por lo tanto, la bandera consultada es la *Z*. Con *Z = 1* la condición se satisface.

SCA HHH Es una instrucción de salto condicional igual que las anteriores. En este caso la condición que se ha de verificar es que el último acarreo sea *1*, para considerar que ésta se encuentra satisfecha.

INP Esta instrucción permite la entrada de un dato (*input*) desde un periférico de entrada. Como se verá más adelante, la transferencia del dato se realiza entre un registro propio del periférico ENT y el acumulador. "X" está dotada de un solo periférico de entrada, por lo tanto, el campo DATA del código de instrucción no identifica direcciones de periféricos, entonces se lo ignora. La siguiente instrucción al *input* no se ejecutará hasta tanto no se haya hecho efectiva la entrada del dato. El control "consulta" la bandera de estado FEN. Si *FEN = 0* no habilita la transferencia, porque significa que el periférico no envió información al registro ENT.

Cuando *FEN = 1*, recién habilita la transferencia y vuelve la bandera FEN a *0*. O sea que FEN se pone en *1* cuando hay información para transferir; en este caso, el que pone FEN en *1* es el periférico. El dato ingresado queda en el acumulador, por lo general, el programador lo transfiere a la memoria con una instrucción STA HHH.

OUT Esta instrucción permite la salida de un dato almacenado con anterioridad en el acumulador hacia un dispositivo periférico. Por lo tanto, la transferencia se realiza entre el acumulador y un registro propio del periférico SAL. "X" sólo tiene asignado un dispositivo de salida, por lo tanto, el campo DATA es ignorado. La instrucción siguiente no se ejecutará hasta que no se haya hecho efectiva la salida. El control "consulta" la bandera FSA. Si *FSA = 0* no habilita la transferencia, porque significa que el periférico aún no está listo. El periférico pone la bandera FSA en *1* cuando está listo para la recepción. La bandera vuelve a *0* cuando se carga la información en el registro SAL para impedir que se siga enviando información antes de que el periférico la muestre.

7.4.5 Unidad de control y sincronización del tiempo

Ahora la pregunta puede ser, ¿cómo se controla el flujo secuencial de microoperaciones para llevar a cabo la tarea del procesamiento?

Piense en la ejecución de la instrucción LDA XXX ¿Cómo se sincronizan las microoperaciones para que la orden de lectura (M → MDR) no se produzca antes de haber seleccionado la

palabra (DATA → MAR) o para que la transferencia al acumulador (MDR → AC) no se produzca antes de que la palabra haya sido cargada en el MDR? Es obvio que cada microoperación debe ser controlada por una variable fundamental: el tiempo. Tomando el último ejemplo, la palabra no estará disponible en el MDR hasta que se haya cumplido el tiempo estimado de respuesta de memoria; antes sería inútil intentar transferir su contenido al acumulador.

Las señales de tiempo que afectan a las microoperaciones están reguladas según el tiempo de respuesta de los registros que involucran.

En la mayoría de las computadoras las señales de tiempo son generadas por un sistema de reloj. Éste se encuentra constituido por un oscilador y circuitos asociados que generan pulsos, cuyo ancho y separación son determinados en forma precisa. Se denomina **ciclo de reloj** o **ciclo menor** al intervalo entre dos pulsos consecutivos de reloj (*0* y *1*), como lo muestra la figura 7.18.

Ciclo de reloj: es el tiempo que transcurre entre dos pulsos adyacentes.

Fig. 7.18. Ciclo de reloj.

En el diagrama de la figura 7.18 se ve la oscilación de la señal entre los valores *0* y *1*, la frecuencia del reloj se controla por un oscilador y cada ciclo de reloj se identifica como *1* hercio, Hz (hertz o ciclo). O sea: *1HZ = 1 ciclo / 1 segundo*.

$$frecuencia = \frac{cantidad\ de\ ciclos\ de\ reloj}{seg}$$

Dado que las unidades de tiempo de una CPU están en el orden de los nanosegundos (10^{-9} *seg, ns*), si se quiere averiguar cuántos ns "tarda" un ciclo de reloj para una computadora que trabaja con una frecuencia de *25* millones de Hz o MHZ:

$$25\ MHz = \frac{25.000.000}{seg}$$

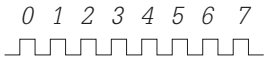
luego

$$1\ ciclo = \frac{1\ seg}{25.000.000} = \frac{1\ seg}{25 \cdot 10^6} = 0{,}04\ seg \cdot 10^{-6} =$$
$$= 40 \cdot 10^{-6} \cdot 10^{-3}\ seg =$$
$$= 40 \cdot 10^{-9}\ seg = 40\ \ nanoseg$$

Una computadora controlada por un sistema de reloj se denomina **sincrónica**.

En estas computadoras se proporcionan secuencias de tiempo repetitivas. Una secuencia repetitiva puede estar formada por *4*, *8* o *16* señales de tiempo, que se diferenciarán por el valor de su subíndice (t_0, t_1, ..., t_{15}). El tiempo de una secuencia repetitiva constituye el **ciclo de la computadora** o **ciclo de máquina**. Éste debería ser compatible con el ciclo de memoria. Recordemos que **ciclo de memoria** es igual al tiempo de acceso a la memoria, si se trata de una memoria de lectura no destructiva, y será igual al tiempo de acceso más el tiempo de restauración, en el caso de una memoria de lectura destructiva.

El **tiempo de acceso a memoria** es el tiempo que tarda la CU en buscar la información en la memoria y dejarla disponible en el MDR.



/-/ "pulso de reloj"

/---/ "ciclo menor" o "ciclo de reloj"

/----------/ "ciclo de la computadora" o "ciclo mayor" o "ciclo de máquina"

"X" es una computadora sincrónica con un sistema de reloj muy elemental, como se observa en la figura 7.19, que genera ocho señales de tiempo: t_0, t_1, t_2, t_3, t_4, t_5, t_6, t_7. El reloj maestro está asociado a un registro de desplazamiento circular, cuya información inicial es *10000000*: por cada pulso del maestro, el único bit "uno" se desplaza un lugar a la derecha. Cada biestable del registro genera una señal de tiempo cada ocho pulsos del reloj maestro.

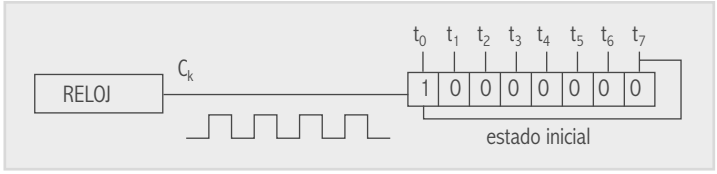


Fig. 7.19. Generación de señales de tiempo.

El registro se carga con el estado inicial mediante un comando externo controlado por el usuario, que indica el comienzo del procesamiento, llamémoslo *ON*, y se resetea mediante otro comando externo, también controlado por el usuario, que indica fin del procesamiento, llamémoslo *OFF*. Puede ser causa de reseteo de este registro un evento interno del procesamiento en curso, por ejemplo, la interpretación de una instrucción de fin de programa. Es obvio que el reseteo de este registro provoca que no se generen señales de tiempo y, por lo tanto, la interrupción de la actividad en "X".

Ahora que está resuelto el problema de sincronización de operaciones de "X", se puede completar el esquema de su unidad de control y evaluar las condiciones que permiten que el control "interprete" los códigos de instrucción y genere las microoperaciones.

Repasemos los componentes de "X": memoria de *4096* palabras de *16* bits cada una; sumador binario asociado al acumulador de *16* bits y al MDR de *16* bits; tres banderas de estado de operación en el *status register* (*S*, *Z* y *C*); puntero de instrucción de *12* bits, llamado contador de programa, que indica la próxima instrucción que ha de ejecutarse.

Convengamos que la primera instrucción del programa siempre se carga en la palabra *000H*, por lo tanto, es éste el valor inicial de PC. El registro de instrucción de *16* bits almacena la instrucción que está siendo decodificada y ejecutada por el control y se encuentra dividido en dos partes: cuatro bits para el código de operación (COP) y *12* bits para la dirección del operando (DATA).

El usuario presiona la tecla *ON* para iniciar la ejecución del programa, lo que desencadena una serie de operaciones internas para la búsqueda de la primera instrucción de memoria. A partir de ese momento, la unidad de control estará en uno de dos estados: búsqueda de la instrucción (o fase *fetch*) o ejecución de la instrucción (o fase *execute*). La unidad de control actualiza la *flag* de control de fase para pasar de un estado a otro. Acordemos que el valor de este biestable es *1* para la fase de búsqueda y *0* para la fase de ejecución. Todos estos componentes brindan cierta información al control que genera, considerando estas variables, las "funciones de control" necesarias para llevar a cabo el procesamiento: las microoperaciones.

El esquema de la figura 7.20 muestra en forma general la CU de "X" formada por dos bloques: el decodificador de instrucción y el secuenciador que genera microoperaciones desfasadas en el tiempo. El primero puede obtenerse a partir de un decodificador *4 · 16*. Las cuatro entradas al circuito varían entre *0000* y *1111*, según la instrucción almacenada en el IR. Las salidas indican con un *1* cuál de las dieciséis combinaciones se dio en la entrada.

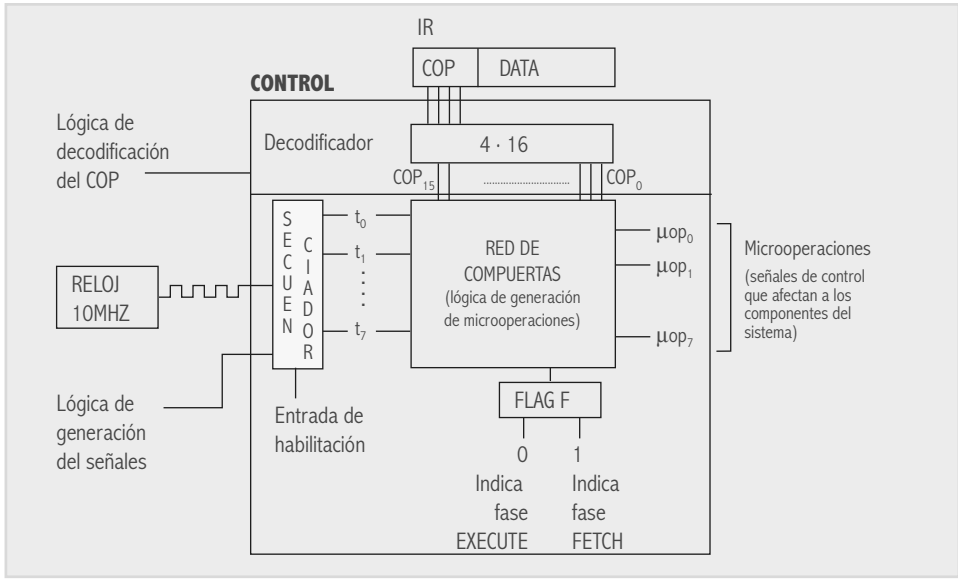


Fig. 7.20. Esquema interno de CU.

7.4.5.1 Diseño parcial de la CU para la orden de lectura de memoria

La variable COP_i está asociada en la lógica de ejecución a un juego determinado de microoperaciones.

En la fase de ejecución las compuertas dependen de la variable de control de fase negada, $\overline{f}$, de las variables que indican cuál de los *16* COP se almacena en el IR y de las variables de tiempo para secuenciar las microoperaciones.

Las instrucciones que requieren un operando de memoria son:

LDA XXX COP_1

ADA XXX COP_2

STA XXX COP_3

ANA XXX COP_5

XOA XXX COP_6

Las microoperaciones que permiten la carga del operando en el MDR para cualquiera de ellas son:

$$DATA \longrightarrow MAR$$

$$WORD \longrightarrow MDR$$

En consecuencia, deben producirse en los primeros dos tiempos del ciclo de la computadora. Las dos compuertas que generan en su salida una de estas microoperaciones dependen, entonces, de las variables siguientes:

$$\mu op_0 = DATA \longrightarrow MAR = \overline{f} \cdot t_0 \cdot (COP_1 + COP_2 + COP_3 + COP_5 + COP_6)$$

$$\mu op_1 = WORD \longrightarrow MDR = \overline{f} \cdot t_1 \cdot (COP_1 + COP_2 + COP_5 + COP_6)$$

$$\mu op_2 = \boxed{} = \overline{f} \cdot t_5 \cdot COP_i$$

ejecución de la instrucción

$$\mu op_3 = 1 \longrightarrow F = \overline{f} \cdot t_7$$

En consecuencia, para la instrucción LDA XXX la

$$\mu op_2 = MDR \longrightarrow AC = \overline{f} \cdot t_5 \cdot COP_1$$

microoperación que habilita la transferencia entre los registros MDR y AC.

Es importante destacar que todas las instrucciones se ejecutan en un ciclo de computadora, aun cuando no utilicen todas las variables de tiempo para su ejecución.

7.4.5.2 Diseño parcial del módulo del control asociado a la fase execute de algunas instrucciones

Para finalizar, en la figura 7.21 se observa un diseño parcial del control considerando solamente la función de control WORD—MDR (orden de lectura de memoria). Si se analizan los casos de lectura en los ejemplos desarrollados, se notará que la CU sólo dispara una orden de lectura en la fase de búsqueda de la instrucción (lectura de la palabra que contiene la instrucción) y en la fase de ejecución de aquellas instrucciones que necesiten un operando de memoria (lectura de la palabra que contiene el operando).

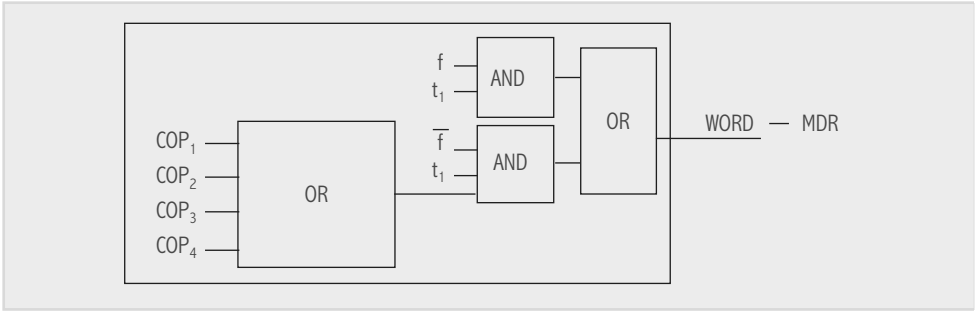


Fig. 7.21. Diseño parcial del control para la orden de lectura de la memoria.

Si la computadora "X" tuviese sólo las cinco instrucciones del programa *P*, podríamos armar la red de compuertas del control para generar las distintas microoperaciones. Es importante analizar que la *flag* indicadora de fase cambia a *0* en el último paso de ejecución de cualquier instrucción, o sea que el estado de *F* es independiente del valor del COP_i y puede producirse en el último t_i del ciclo de la computadora, que puede ser t_7, esto es, $SET\ F = t_7$. El registro MDR recibe la transferencia del PC sólo en la fase de búsqueda, o sea:

$$SETF = f_1 = F \cdot t_0$$

recibe la transferencia del campo DATA en f_6, f_{10} y f_{16}, que se puede expresar como:

$$DATAMAR = f_6 + f_{10} + f_{16} = \overline{F} \cdot t_0 \cdot COP_1 + \overline{F} \cdot t_0 \cdot COP_2 + \overline{F} \cdot t_0 \cdot COP_3 = \\ = \overline{F} \cdot t_0 \ (COP_1 + COP_2 + COP_3)$$

El registro MDR recibe la transferencia de la palabra de memoria en f_2, f_7 y f_{11}, que se puede expresar como:

$$WORDMDR = f_2 + f_7 + f_{11} = F \cdot t_1 + \overline{F} \cdot t_1 \cdot COP_1 + \overline{F} \cdot t_1 \cdot COP_2 = \\ = F \cdot t_1 + \overline{F} \cdot t_1 (COP_1 + COP_2)$$

y recibe la transferencia del acumulador en f_{17}, entonces:

$$ACMDR = \overline{F} \cdot t_1 \cdot COP_3$$

El acumulador recibe una orden de desplazamiento en f_{14}, entonces:

, $$SHRAC = \overline{F} \cdot t_0 \cdot COP_{10}$$

y una orden de recibir el resultado de la suma en f_{12}

$$SUMAC = \overline{F} \cdot t_5 \cdot COP_2$$

El secuenciador de tiempos puede detenerse en el primer tiempo de la fase *execute*, entonces,

$$DISABLE = \overline{F} \cdot t_0$$

7.4.6 El módulo de cálculo: unidad aritmético–lógica

El módulo de tratamiento de datos es la **unidad aritmético-lógica** (ALU). En este módulo los datos se tratan según órdenes de la **unidad de control** (CU), que interpreta la instrucción durante su estado de ejecución.

Las instrucciones de "X" que hacen referencia a la ALU son ADA, INC, CMA, SHR y SHL, correspondientes a los operadores aritméticos "suma" (*+*), "incremento" (*+1*), "complemento", "división por la base binaria" y "multiplicación por la base binaria". ANA y XOA responden a los operadores lógicos "y" (AND) y "o excluyente" (*or exclusive*).

Las instrucciones LDA, STA y CLA permiten el control del registro acumulador perteneciente a este módulo, gestionando su carga, descarga y puesta a *0*.

Como se ve, la mayor parte de las instrucciones de "X" requiere la intervención de la ALU.

La unidad aritmético-lógica de "X" está implementada en torno del registro acumulador. Cuando la operación se realiza sobre un dato, éste se almacena en el acumulador y el resultado de la operación permanece en este registro. Las instrucciones que generan operaciones sobre un solo dato en la ALU son:

CLA CMA INC SHR SHL

Cuando aparece el código de operación correspondiente a alguna de ellas, el control genera una señal que habilita alguna de las funciones del acumulador, por lo tanto, este registro debe tener capacidad de "puesta a *0*", "complemento", "incremento", "desplazamiento a derecha" y "desplazamiento a izquierda".

El resto de las operaciones aritméticas o lógicas afectan a dos datos. En consecuencia, el primer dato se almacena en el acumulador y el segundo permanece en el separador de memoria (MDR), mientras que el resultado final se almacena en el acumulador. Las instrucciones de este tipo son

ADA, ANA, XOA

que requieren una lógica adicional en la ALU. Considérese un sumador binario paralelo armado, en este caso, con *16* sumadores completos convencionales. Veamos la segunda opción, de modo que las tres funciones queden implementadas, según la figura 7.22.

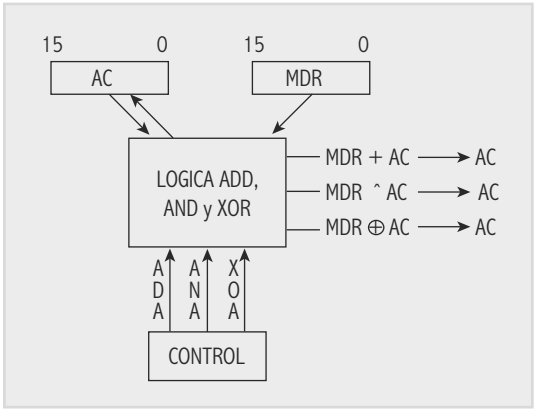


Fig. 7.22. Diagrama de relaciones entre los operandos y las operaciones.

Es importante destacar el valor de las banderas (*flags*) del registro de estado asociado a esta unidad (*status register* o SR). El estado de las banderas permite que el programador controle la condición final de una operación para establecer o no una ruptura de la secuencia normal del programa. Cada bandera se actualiza después de una operación en la ALU (asumiendo un valor *0* o *1*), según la lógica presentada en la figura 7.23.

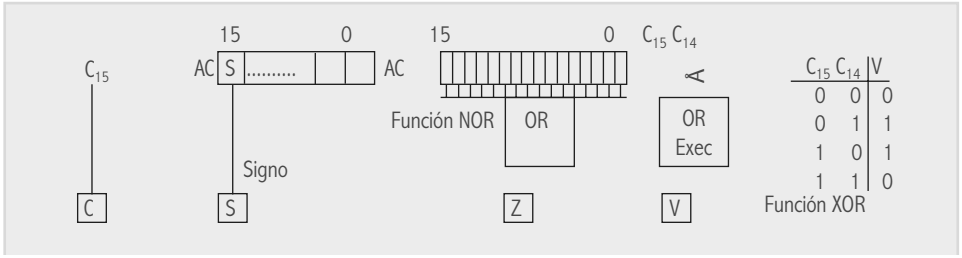


Fig. 7.23. Lógica de actualización de las banderas.

El *status register* de "X" representa sólo tres *flags,* que son:

- *N* –a veces denominado *S* por signo– (resultado del acumulador negativo) que vale *1* cuando $S\ (AC_{15}) = 1$.
- *Z* (resultado = cero) que vale *1* cuando **todos** los AC_i son iguales a *0*.
- *C* (*carry* o acarreo que se produce al sumar el último par de bits del acumulador) que asume el valor de C_{15}.

El *overflow* se produce cuando el resultado excede la capacidad del acumulador y, por lo tanto, es incorrecto; cuando se produce *overflow*, "X" detiene la ejecución del programa; como no hay instrucción de consulta de *overflow*, se puede detectar consultando si $C \neq N$.

Las instrucciones de salto condicional SNA, SZA y SCA provocan salto cuando la condición de la bandera es verdadera. Por lo tanto, la lógica que genera la microoperación en el control depende de ellas y se representa en la figura 7.24.

En la figura 7.25 se presenta el esquema global de la ALU de "X".

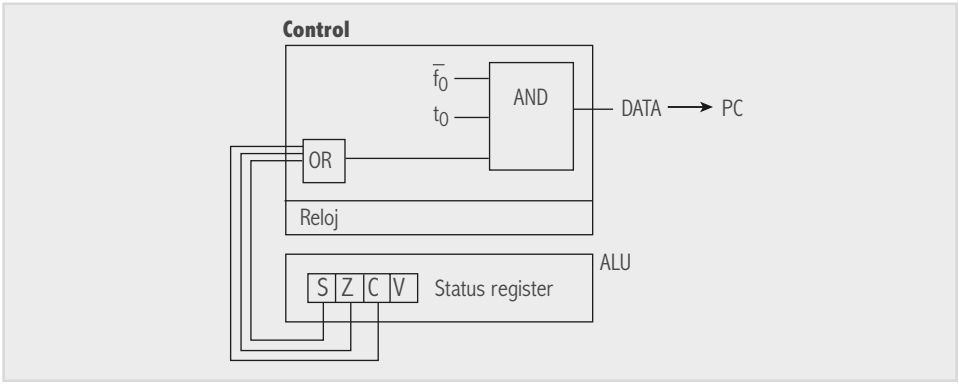


Fig. 7.24. Lógica de actualización de las banderas.

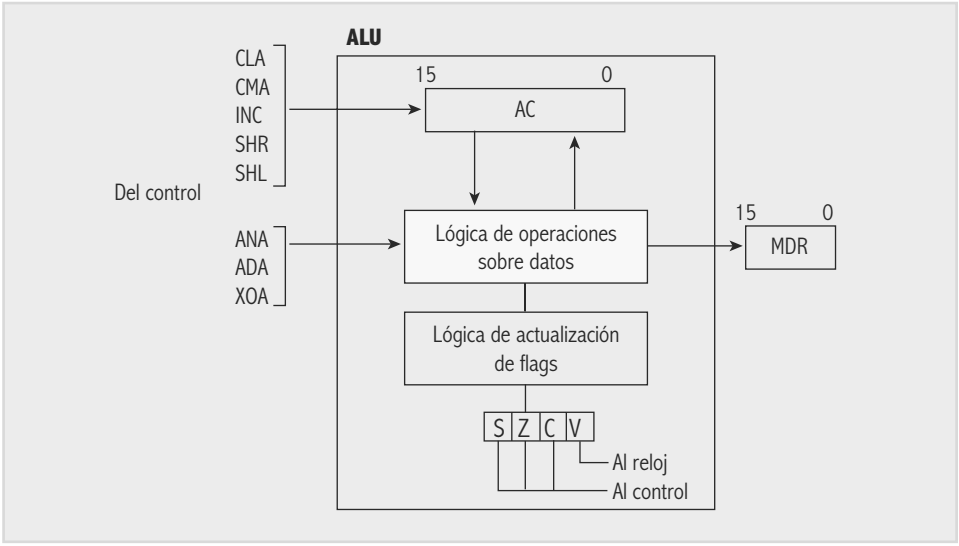


Fig. 7.25. Instrucciones aritméticas y lógicas en la ALU.

El ejemplo siguiente le permitirá ver una aplicación del efecto que causa el estado final de una operación para la toma de decisiones en un programa. Considere una porción de programa Assembler, donde el resultado de comparar dos números por mayor, menor e igual permite que el programador se bifurque a otros puntos del programa. El método usado para la comparación es la resta de los datos.

```
010 LDA 3A2
011 CMA          resta (3A1 – 3A2)
012 INC
013 ADA 3A1
014 SZA 100      si son iguales salta a 100
015 SNA 200      si 3A1 < 3A2 salta a 200 (si Z = 0 y S = 1)
016 JMP 300      salta a 300
```

7.4.6.1 El desplazamiento de bits en los registros de la ALU

Cada desplazamiento a derecha o a izquierda de los bits de un registro implica, en principio, el corrimiento de un solo bit, pero trae aparejado el arrastre de todos los demás. Los desplazamientos a derecha o izquierda son otras dos operaciones accesibles gracias a la ALU, donde los operandos residen temporariamente en registros asociados al sumador.

En este momento, cabe preguntarse por qué razón se desplazan bits en un registro. Básicamente, por tres razones esenciales:

- Para multiplicar.
- Para dividir.
- Para "testear" un bit.

En efecto, usted sabe que hemos remarcado que en las operaciones aritméticas definidas para la ALU la multiplicación y la división no figuran. Una multiplicación se define por medio de una serie de instrucciones en código de máquina, que implican sumas y desplazamientos sucesivos. Por ejemplo, supóngase en un formato de sólo tres dígitos decimales el valor siguiente:

080 *080*

→ *008* ← *800*

Un desplazamiento a derecha divide el valor *080* por *10* (*008*), mientras que uno a izquierda lo multiplica por *10* (*800*). En sistemas de numeración de cualquier base, el desplazamiento permite multiplicar o dividir un número por su base (siempre representada por *10*). De la misma manera, en binario, para multiplicar un número *2* (la base) es necesario agregarle un *0* a derecha, que es equivalente a desplazar todos sus bits un bit a izquierda; para multiplicarlo por *4*, se necesitan dos desplazamientos. Veamos el ejemplo siguiente, que se trata de multiplicar el número *6* por *5*:

$$6 \cdot 5 = ((6 \cdot 2) \cdot 2) + 6 = 30$$

Así, en formato de *8* bits, los registros contendrán los valores que se detallan en la figura 7.26,

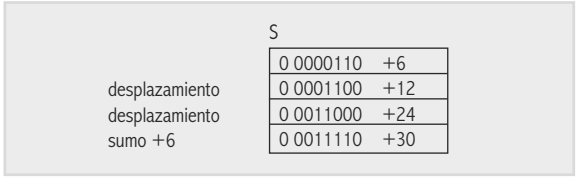


Fig. 7.26. Evolución del cálculo.

lo que nos da un algoritmo válido para la multiplicación, utilizando desplazamientos sucesivos y sumas.

Es posible que en el desplazamiento también intervenga el bit de acarreo. Se pueden hacer tantos desplazamientos como los necesarios para que un bit que se quiera testear llegue al acarreo; de este modo, se pregunta el valor del bit con la instrucción en código de máquina SCA.

7.4.6.2 Comparación mediante resta

La comparación también es función de la ALU. Cuando se quieren comparar dos operandos, se efectúa una sustracción sin necesidad de almacenar el resultado. Las instrucciones en código de máquina permiten preguntar por la bandera indicadora de *zero* o de signo, de modo

que, si $Z = 0$ y $S = 1$, el primer operando es menor que el segundo, y si $Z = 0$ y $S = 0$, el primer operando será mayor que el segundo.

7.4.6.3 Incremento y decremento de un registro

Incrementar el valor de un operando implica aumentarlo en una unidad o, por el contrario, decrementar significa restarle una unidad. La operación se realiza sobre un solo operando y, por lo general, afecta al registro acumulador.

7.4.6.4 Operaciones lógicas

Examinemos las cuatro operaciones lógicas que permite realizar la ALU.

La complementación: es una operación que se aplica a un solo operando; así, si el operando mide un octeto, la operación invertirá el valor de cada uno de los 8 bits:

```
10100101
01011010
```

Una de las aplicaciones tiene lugar en el cálculo de una resta, $X - Y = Z$, mediante la suma del complemento del sustraendo. Ejemplo:

```
LDA, Y
CMA      complemento
INC      suma 1
ADA, X
STA, Z
```

El "y" *lógico*: es una operación que se aplica a dos operandos $A \wedge B = Z$. La operación consiste en hallar el producto lógico (bit a bit), no aritmético, de cada par de bits. Entonces:

```
  00110010 A
∧ 00001111 B
  00000010 Z
```

El producto lógico consiste en multiplicar $a_i \cdot b_i$ para $7 \geq i \geq 0$. Ejemplo:

```
LDA, A
ANA, B
STA, Z
```

Una de las aplicaciones es el uso de máscara para indicar qué grupo de bits del operando A se va a reconocer como información en el operando Z.

En este caso, la máscara *0001111* permite que sobre Z se "copien" los primeros 4 bits de A y se anule el resto de la información.

El "o" lógico: es una operación que se aplica a dos operandos $A \vee B = Z$. La operación consiste en hallar la suma lógica (no aritmética) de cada par de bits.

```
  0000 1000 A
v 0011 0000 B
  0011 1000 Z
  └──┘ └──┘
   3    8
```

Aplicación: permite agregar los bits de zona para transformar un dígito BCD en el correspondiente carácter ASCII.

El "o" exclusivo: es una operación lógica que se aplica a dos operandos *A* y *B* = *C*. La operación es similar a la suma aritmética, sólo que no se consideran los acarreos al operar cada par de bits.

10101100 A
v *10101100 B*
00000000 C

Aplicación: permite comparar si dos operandos son iguales, en cuyo caso el resultado es *0*.

7.4.6.5 El registro de estado asociado a la ALU

Cada vez que la ALU realiza una operación elemental, mantiene cierta información que indica el estado final de esa operación. Las instrucciones de código de máquina pueden testear el estado de cada uno de estos bits para tomar decisiones dentro del programa, esto significa que todas estas instrucciones provocan una ruptura de secuencia según sea el valor de la *flag 0* o *1*.

Esta información queda retenida en un "registro", que se denomina **registro de estado** (*status register*), cuya medida en general depende de la medida de palabra de la computadora. En este registro, cada bit se denomina ***flag*** (bandera) e indica con un "*1*" la condición de verdad y con "*0*" la condición de falsedad de un estado. Por ejemplo, en una computadora de palabra de *8* bits (el ejemplo responde al microprocesador *8080*) el registro de estado se encuentra constituido por las *flags* que se detallan en el esquema de la figura 7.27.

Fig. 7.27. Banderas del *status register*.

La *flag S* o *N* es el indicador del signo del resultado. *S = 1* indica que el resultado es negativo y *S = 0*, que es positivo.

La *flag Z* es el indicador de zero. *Z = 1* indica que el resultado es cero y *Z = 0*, que es distinto de *0*.

La *flag V* u *O* es el indicador de *overflow*. *O = 1* indica que el resultado supera la capacidad prevista para ese formato; *O = 0*, que no se produjo desborde.

La *flag H* es el indicador de acarreo de los *4* primeros bits (utilizado cuando la ALU suma dos dígitos BCD); indica el valor que se tomará como acarreo para el próximo par de dígitos BCD.

La *flag P* es el indicador de paridad (par o impar). Indica *1* si la cantidad de *1* del resultado es impar (en el caso de que la ALU opere con paridad par) y *0* en caso contrario.

La *flag I* es el indicador de interrupción e indica con un *1* si hay un requerimiento de atención de la CPU por parte de un dispositivo externo a ella.

La *flag C* es el indicador del último acarreo (*carry*) de una operación y puede llegar a intervenir en los desplazamientos como un bit más asociado al acumulador.

En la presentación de nuestro modelo de arquitectura se indicó que "X" no era una computadora concreta del mercado actual. Suponemos que "X" lo habrá ayudado a familiarizarse con algunos conceptos importantes sobre el procesamiento de datos con computadoras; sin

embargo, es necesario considerar las variaciones que permiten una mayor eficiencia de esta tarea en computadoras que se alejan del modelo básico. Estas variaciones afectan tanto a los elementos de hardware como a los de software.

7.5 Resumen

Los conceptos enunciados en este capítulo son fundamentales para la interpretación de cada uno de los que siguen; es importante destacar que la función de la CPU en cualquier computadora digital es interpretar y ejecutar el código binario de las instrucciones. Estas instrucciones han sido el producto de un proceso de compilación o de interpretación. El programa original o fuente fue escrito por un programador de aplicaciones en lenguaje de alto nivel, que, por ejemplo, hace treinta años fue "Fortran" y que hoy es alguno de los denominados "lenguajes de cuarta generación", como el ABAP; o bien, "lenguajes de quinta generación" que son aquellos orientados a la inferencia de conocimiento. Todos ellos son "compilables" para la generación del código ejecutable. La CPU también interpreta y ejecuta las instrucciones de los programas del sistema operativo, pues aquellas instrucciones reservadas para determinadas operaciones (y vedadas al programador de aplicaciones) están incluidas en el set de instrucciones que interpreta el procesador.

Las CPU actuales utilizan múltiples registros para el almacenamiento temporal de datos y de referencias a memoria. Están categorizados como registros invisibles, registros de uso exclusivo de las instrucciones privilegiadas del sistema operativo, en los cuales los bits almacenados se relacionan con la administración de la dinámica del sistema. Por otra parte, están categorizados como registros visibles o de propósito general, que son aquellos utilizados por las instrucciones de las aplicaciones. También se agregan a los procesadores actuales memorias de almacenamiento ultra-rápidas denominadas "*caché* de instrucciones" o "*caché* de datos"; estos subsistemas de almacenamiento están incluidos con el fin de adelantar la búsqueda de ambos tipos de objetos (instrucciones y datos) desde su almacenamiento en la memoria principal. El almacenamiento en *cachés* brinda a la CPU una disponibilidad casi continua de instrucciones y datos, esto mejora la velocidad de ejecución, pues obtiene objetos en tiempos acordes a sus necesidades (existe una diferencia notable entre ésta y los tiempos de respuesta de las tecnologías que se utilizan en la memoria principal). La organización de las memorias, las diversas tecnologías y la manera en la que el sistema operativo puede dividir territorios en las distintas áreas de almacenamiento se verán en el capítulo Memorias.

En relación con las instrucciones de máquina o nativas, casi todas ellas se pueden clasificar en instrucciones del tipo memoria-registro, o del tipo registro-registro como veremos en el capítulo Instrucciones. Por ejemplo, si previamente se cargan dos operandos en registros de cálculo de la CPU, se utilizan instrucciones de transferencia memoria-registro (MOVER reg*1*, mem), que luego se podrán "operar" con instrucciones aritméticas o lógicas de tipo registro-registro (SUMAR reg*1*, reg*2*). Puede ocurrir también que una única instrucción involucre tanto la carga en CPU de dos operandos, como la operación entre ellos y el posterior almacenamiento del resultado. Cuanto más rápido se pueda realizar este ciclo, mejor. La velocidad de procesamiento no sólo tiene que ver con la velocidad del reloj, también es necesario un modelo de ejecución adecuado. Esto último tiene que ver con los paradigmas presentados para mejorar el ciclo de ejecución de una instrucción. Al paradigma original enunciado por Von Neumann, se agregó un nuevo modelo de ejecución, RISC (*Reduced Set Instruction Set Code*), cuyos referentes principales fueron, David Patterson, de la Universidad de Berkeley, y John Hennessy, de la Universidad de Stanford.

7.6 Ejercicios propuestos

1) Problema de programación:

En la memoria de nuestra computadora ("X") se almacenaron 3 valores en representación de punto fijo con negativos complementados a 2 en las direcciones: 03A, 03B y 03C.

a) Realizar el programa en Assembler, sumando los 3 valores y almacenando el resultado en la dirección 0FE, si el valor es mayor o igual que 0, o en la dirección 0FF, en caso contrario.

b) Codificar el programa en código de máquina representando los valores binarios en hexadecimal.

c) En una tabla indicar las microoperaciones que permiten la ejecución de cada instrucción y representar los contenidos de los registros especiales AC, PC, MAR, MDR, IR, S, Z, V y C, y en las palabras de memoria 03A, 03B, 03C, 0FE y 0FF.

DATOS:

(03A) = 7F00

(03B) = 91A2

(03C) = 6AB1

Palabra inicial de carga de programa A0A.

2) Ejecutar el programa enunciado en 1) bajo *Debug*, haciendo las modificaciones necesarias respecto de las direcciones de memoria y de las instrucciones, ejecutando los *trace* que permitan ver el contenido de los registros luego de cada instrucción.

Comparar con el desarrollo manual respecto del resultado final y enunciar la conclusión que se desprende de la comparación.

3) Diseñar un control (gráfico de compuertas del circuito) que indique las microoperaciones necesarias para obtener el promedio de 2 números, según el programa:

LDA XXX
ADD XXX
SHR XXX
STA XXX.

Tenga en cuenta que cada microoperación tarda un ciclo de computadora (para "X" 8 señales de tiempo).

4) Diseñe un circuito que, acoplado al reloj maestro, genere una secuencia repetitiva de cuatro señales de tiempo.

5) Diseñe un semicircuito incluido en el control para la habilitación de la microoperación PC + 1PC.

6) Programe en Assembler la estructura condicional de un lenguaje de alto nivel similar a la siguiente semántica. Considérese que las referencias a memoria para A, B y C son 300, 301 y 302

IF A = B THEN C=A + B ELSE C=A-B

7) Programe en Assembler la estructura iterativa de un lenguaje de alto nivel similar a la semántica siguiente:

FOR I= 1 to 10
x = x +1
NEXT I

8) Represente el algoritmo de Booth para la suma de números que utilicen negativos complementados a 2 en un diagrama de flujo y realice el programa en el Assembler presentado para esta computadora.

7.7 Contenido de la página Web de apoyo

El material marcado con asterisco (*) sólo está disponible para docentes.

Resumen gráfico del capítulo

Animación

Demostración de las distintas fases de la CPU

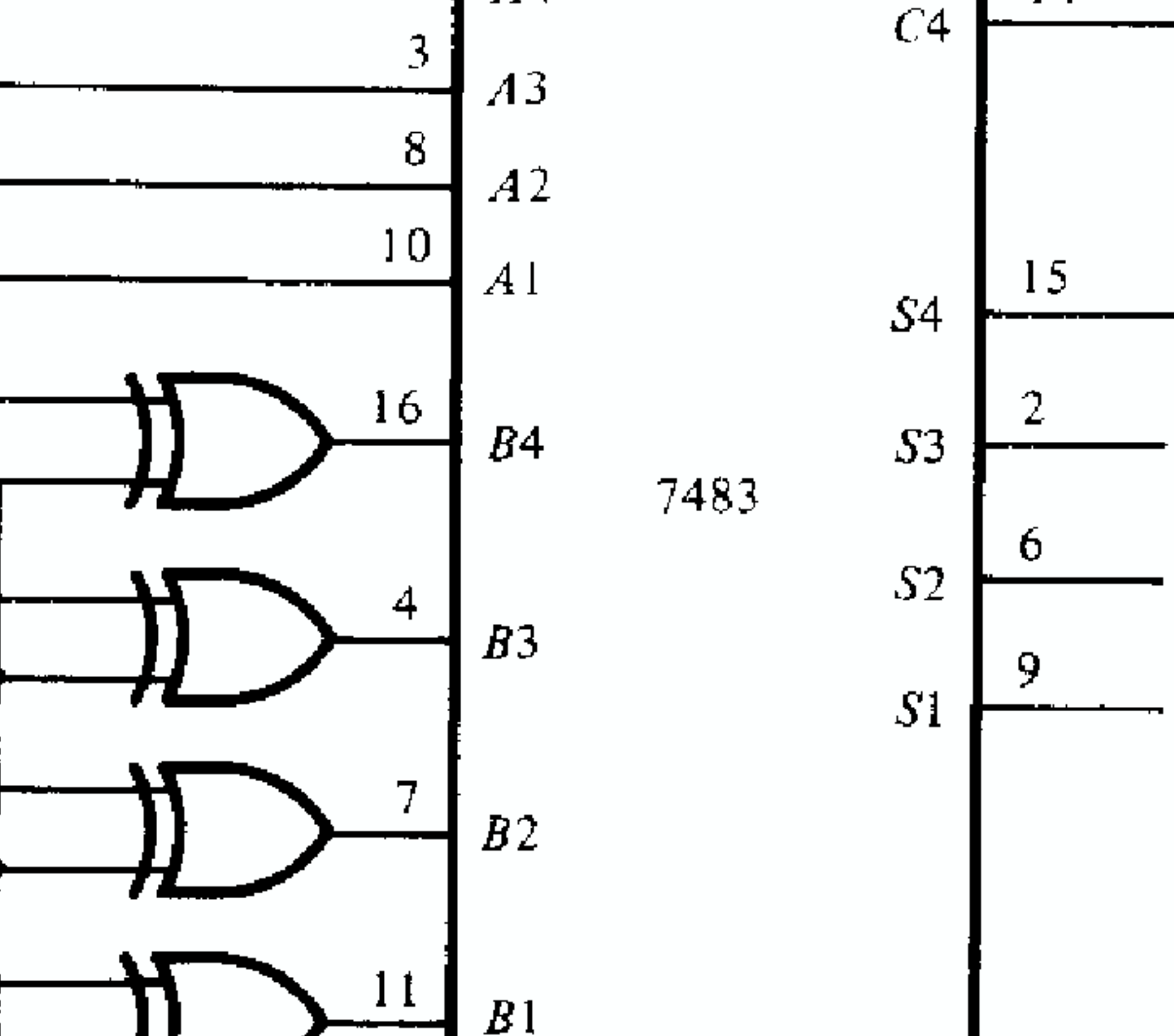
Autoevaluación

Video explicativo (01:53 minutos aprox.)

Audio explicativo (01:53 minutos aprox.)

Evaluaciones Propuestas*

Presentaciones*


3
A3
8
A2
10
A1
16
B4
4
B3
7
B2
11
B1
C4
15
S4
2
S3
6
S2
9
S1
7483