Introducción a los Computadores

La Máquina Sencilla: Introducción a la Estructura Básica de un Computador

Dpto. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza

1. ESTRUCTURA BASICA DE UN COMPUTADOR

En este tema vamos a estudiar la estructura básica de un computador, es decir, qué bloques funcionales lo constituyen y qué características presentan de cara al diseñador.

Un **computador** es un sistema secuencial más o menos complejo capaz de realizar un conjunto determinado de operaciones básicas que llamaremos **instrucciones.**

Se denomina **programa** a la secuencia de instrucciones que resuelve un determinado problema.

Todo computador consta de:

- **Unidad de Proceso** (UP): conjunto de bloques destinados a realizar operaciones sobre datos. Los elementos básicos de la UP son:
    * **Unidad Aritmético-Lógica** (ALU): conjunto de bloques destinados a realizar operaciones aritméticas (+, -, ...) y lógicas (and, or, xor, ...).
    * **Registros**: básicamente almacenan los datos con que operar y resultados de la operación.
- **Unidad de Control** (UC): conjunto de bloques que gobiernan el funcionamiento de la UP. Para cada instrucción del programa, la UC decide que acciones se deben realizar sobre la UP y resto de bloques del computador.
- **Memoria**: bloque destinado a almacenar información a la espera de ser trasladada a la UP o al exterior del computador (a través del bloque de entrada / salida que se presenta a continuación).
- **Entrada / Salida**: bloque que permite la comunicación con el exterior, haciendo que el computador sea un sistema abierto, es decir, accesible desde el exterior mediante **periféricos.**
- **Buses**: permiten la transferencia de información entre todos los bloques que constituyen el computador.

El conjunto formado por la UC y UP se denomina **Unidad Central de Proceso (CPU) o Procesador.** Para el usuario del computador, la CPU se caracteriza por su **repertorio de instrucciones** (conjunto de operaciones básicas que es capaz de realizar). Para cada instrucción, la UC realiza una secuencia de acciones sobre la UP a fin de realizar la operación indicada en la instrucción.

El bloque de memoria almacena dos tipos de información:

- datos: variables y constantes utilizados en el programa;
- programa: secuencia de instrucciones.

Se utiliza dos tipos de memoria según el tipo de información a almacenar:

- Memoria **RAM**: almacena datos y programas del usuario.
- Memoria **ROM**: almacena datos y programas propios del computador que permiten su funcionamiento como sistema secuencial.

Cada instrucción o dato se almacena en memoria como una secuencia de ceros y unos de longitud determinada. Esta información se referencia con una **dirección** de memoria o posición que ocupa dentro de ella.

Cada periférico del bloque de entrada/salida tambien puede referenciarse con una dirección de periférico.

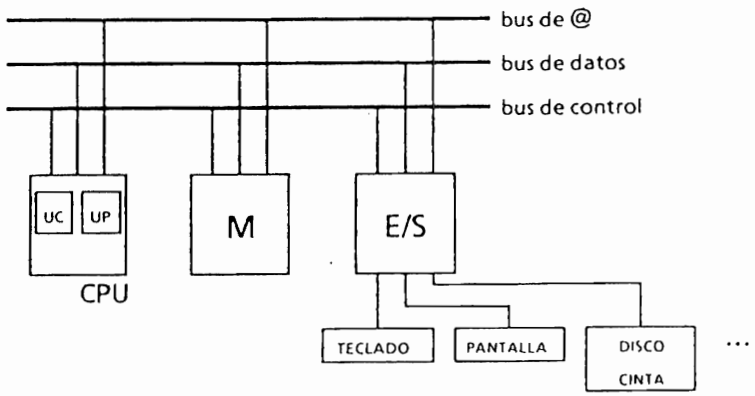
A fin de permitir la transferencia de información entre bloques del computador, este dispone de:

a/ **Bus de direcciones**: a través de él la CPU indica la dirección de memoria donde se encuentra el dato o instrucción que desea acceder.

b/ **Bus de datos**: por él se transfiere el contenido de la posición de memoria seleccionada. Es un bus bidireccional, es decir, permite la transferencia de información desde memoria a CPU (operación de lectura) o desde CPU a memoria (operación de escritura).

c/ **Bus de control**: conjunto de líneas que controlan la transferencia a realizar. Cabe destacar, de entre otras, las líneas de lectura/escritura, acceso a memoria/periférico, reloj, ...

La estructura general de un computador sería la mostrada en la siguiente figura:



A continuación pasaremos a diseñar la CPU de un computador sencillo definido por el conjunto de instrucciones que es capaz de ejecutar y memoria que es capaz de direccionar.

2. LA MAQUINA SENCILLA MS1

2.1 Descripción de la arquitectura

Repertorio de instrucciones

Queremos diseñar una CPU capaz de ejecutar 4 instrucciones diferentes:

a/ suma: suma el contenido de dos posiciones de memoria D y F dejando el resultado en D:

$$ADD\ F, D \qquad D \leftarrow (D) + (F).$$

Además se modifica un indicador interno, bandera o FLAG que denominaremos FZ en función del resultado de la operación. Este indicador se activa a 1 si el resultado es cero o se activa a 0 en caso contrario, es decir,

$$FZ \leftarrow (D)+(F)=0\ .$$

b/ movimiento: transferencia del contenido de una posición de memoria F a otra D:

$$MOV\ F, D \qquad D \leftarrow (F);$$
$$FZ \leftarrow (F)=0\ .$$

c/ comparación: compara el contenido de dos posiciones de memoria D y F:

$$CMP\ F,D \qquad (F) - (D);$$
$$FZ \leftarrow (F) - (D) = 0.$$

No almacena el resultado de la operación en el destino.

d/ salto: si FZ = 1, pasa a ejecutar la instrucción que se encuentra en la dirección D. En caso contrario, continua con la siguiente instrucción de memoria.

$$BEQ\ \ D \qquad Si\ FZ=1 \qquad PC \leftarrow D$$

Esta instrucción permite romper el **secuenciamiento** implícito de las instrucciones de un programa.

Por ejemplo, vamos a ver cual sería la secuencia de instrucciones que debería ejecutar la CPU para realizar una multiplicación de dos variables *a* y *b* dejando el resultado almacenado en la variable *c*. El algoritmo escrito en lenguaje de alto nivel sería, por ejemplo:

```
begin
    c := 0;
    i := 0;
    while i < b do
        begin
        c := c + a;
        i := i + 1
        end
end.
```

La traducción de este algoritmo utilizando el repertorio de instrucciones definido para nuestro computador sería:

```
begin:  MOV 0, c        ;c := 0
        MOV 0, i        ;i := 0
while:  CMP i, b        ;mientras i < b
        BEQ end
        ADD a, c        ;c := c + a
        ADD 1, i        ;i := i + 1
        CMP X, X
        BEQ while
end:
```

Fijémonos que la última instrucción de salto debe ser incondicional, es decir, debe saltar en cualquier caso. Para ello, previamente se hace una instrucción de comparación que fuerze FZ = 1.

Memoria

Tanto el programa anterior como las variables y constantes que utiliza deberán residir en memoria.

En el diseño de este computador supondremos que el bloque de memoria RAM dispone de una capacidad máxima de 128 palabras, tanto para almacenar instrucciones, datos o constantes.

Si por ejemplo suponemos que las variables y constantes se almacenan a partir de la dirección 100 y el programa a partir de la dirección 0, tendremos las siguientes direcciones de memoria asignadas a las variables, constantes e instrucciones del programa, suponiendo que cada una de ellas ocupa una palabra de memoria:

```
a     →  @100
b     →  @101
c     →  @102
i     →  @103
1     →  @104
0     →  @105
begin:   @0
while:   @2
end:     @8
```

Formato de instrucción

Si las instrucciones deben estar almacenadas en memoria, deberán codificarse en binario. Se denomina **formato** de una instrucción a la organización en campos de bits de toda la información que requiere la instrucción para ser ejecutada. Este formato deberá indicar:

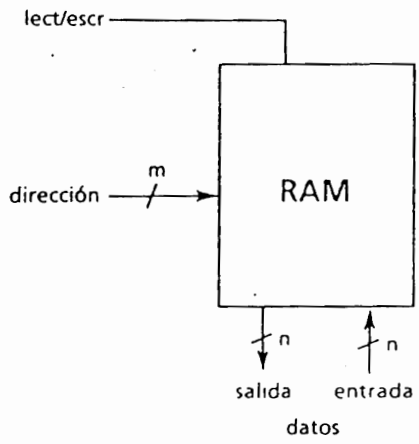
* de qué instrucción se trata (código de operación CO);
* dónde se encuentran los operandos que necesita y dónde almacenar el resultado de la instrucción (referencia a operandos). Nuestra instrucción especificará los operandos dando la dirección de memoria en que se encuentran. A este mecanismo se denomina **direccionamiento absoluto o directo.**

Si nuestra CPU dispone sólo de 4 operaciones distintas, el campo que indique el tipo de instrucción a ejecutar deberá tener una anchura de 2 bits. Los códigos de operación asignados a cada instrucción podrían ser los siguientes:

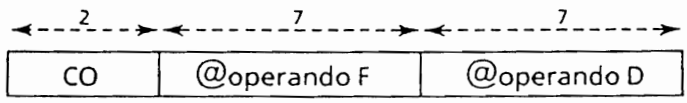
CO_1	CO_0	operación
0	0	ADD
0	1	CMP
1	0	MOV
1	1	BEQ

Todas las instruciones anteriores, además del CO deberán indicar donde están los operandos. Las instrucciones de que dispone nuestra CPU poseen como máximo dos operandos (fuente y/o destino). El operando destino puede indicar además donde dejar el resultado de la instrucción. Por lo tanto necesitaremos dos campos en nuestro formato de instrucción para indicar esta información. La anchura de estos campos dependerá del tamaño de memoria disponible, es decir, del número de posiciones direccionables en la memoria

RAM. Si la memoria RAM está constituida por 2^m registros de n bits cada uno, nuestros campos de dirección necesitarán m bits.



Dado que la memoria máxima direccionable es de 128 palabras (m = 7), tendremos que nuestras instrucciones tendrán un formato de 16 bits. Este formato podría ser el siguiente:



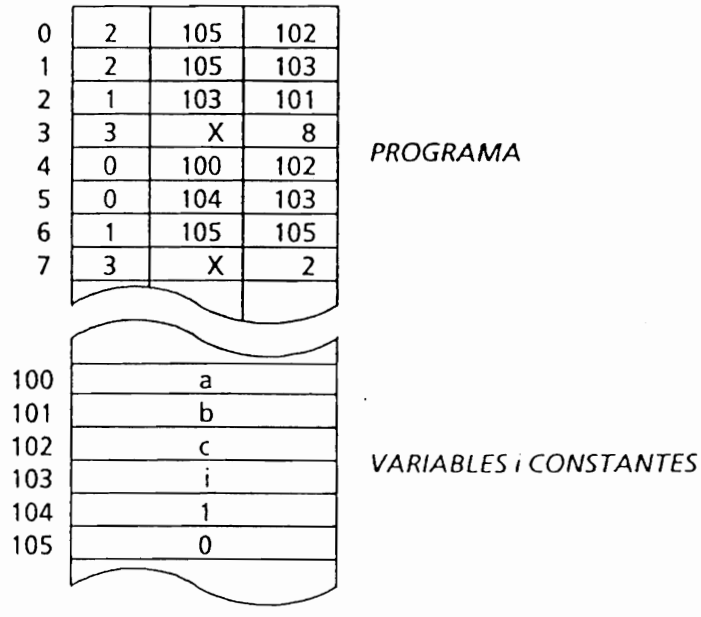
Para ejecutar un programa, la CPU lee de memoria, una a una, las instrucciones que constituyen dicho programa. Si las instrucciones precisan datos, sus direcciones vendrán indicadas después del código de operación.

Datos

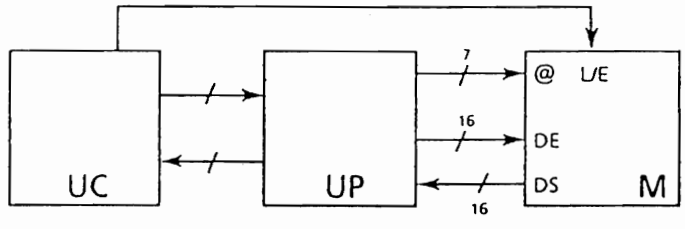
Supondremos que los datos que nuestra máquina va a utilizar son números enteros positivos codificados con 16 bits.

De esta manera, tenemos que la memoria RAM utilizada para almacenar tanto instrucciones como datos estará organizada como 128 palabras de 16 bits cada una.

Con este formato para instrucciones y datos, el programa ejemplo anterior quedaría almacenado en memoria tal como muestra la siguiente figura:



Hasta este punto tenemos definida la estructura general de nuestro computador:



No se incluye por razones de sencillez el diseño del bloque de entrada/salida.

2.2 Diseño de la Unidad de Proceso

A continuación procederemos a diseñar la CPU de nuestro computador. El diseño de la CPU se puede dividir en dos partes:

* diseño de la UP, donde se realizarán operaciones (diseño de la ALU) y guardará determinada información (registros internos de la CPU);
* diseño de la UC, encargada de gobernar el funcionamiento de los bloques que constituyen la UP.

Diseño de la ALU

Por un lado, la ALU deberá permitir realizar las operaciones que se necesitan para poder ejecutar las cuatro instrucciones anteriores:

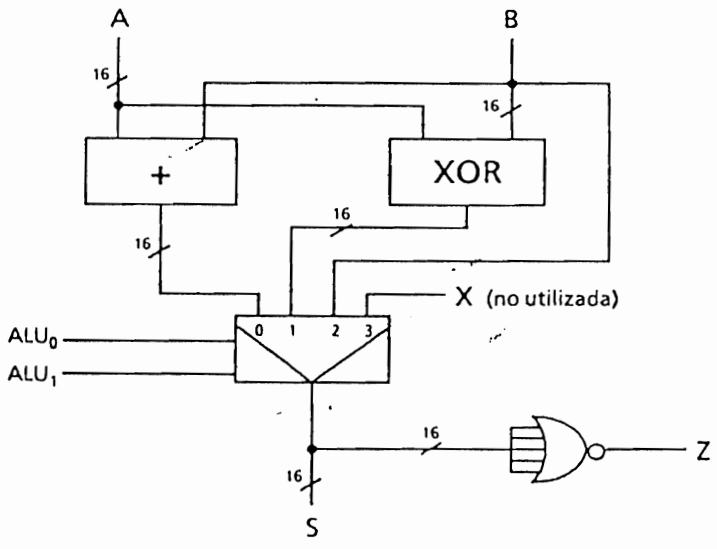
- sumar: necesitaremos un sumador de dos valores de 16 bits;
- comparar: se puede realizar aritméticamente, mediante un restador y detector de cero, o lógicamente, haciendo la comparación bit a bit con puertas XOR. En nuestro diseño adoptaremos la segunda solución.
- ser transparente a un dato de entrada, o sea, dejar pasar un dato desde la entrada a la salida.

Además la ALU debe detectar si el resultado de la operación es cero o no (salida *Z* de la ALU).

La operación que debe realizar la ALU en cada instante vendrá controlada por dos bits ALU_1 y ALU_0, tal como se muestra a continuación:

ALU_1	ALU_0	operación
0	0	A+B
0	1	A xor B (bit a bit)
1	0	B
1	1	no definida

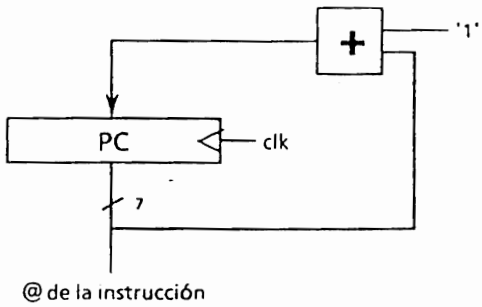
El diseño a nivel de bloques de la ALU se muestra en la figura siguiente:



Registros

La UP, además de la ALU, necesita registros donde almacenar temporalmente determinada información.

En primer lugar, nos hace falta un registro que guarde la dirección donde se encuentra la siguiente instrucción a ejecutar. Este registro se denomina **Contador de Programa** PC. Hemos visto que las instrucciones del programa se almacenan de forma secuencial en memoria, con lo cual el registro PC debe incrementarse en 1 cada vez que se ejecute una instrucción. A este mecanismo se denomina 'secuenciamiento implícito'. La lógica de la UP necesaria para realizar este mecanismo es la siguiente:



Este secuenciamiento implícito obliga la existencia de instrucciones de salto en el repertorio del computador a fin de permitir la ruptura de secuencia de instrucciones del programa.

Durante la ejecución de la instrucción, la UC realiza varios accesos a memoria para conseguir toda la información que requiere para ser ejecutada. Por ejemplo, la instrucción ADD F,D hace los siguientes accesos a memoria:

1er acceso: buscar instrucción;

2o " : buscar operando fuente utilizando la dirección F que se encuentra en la instrucción;

3er " : buscar operando destino utilizando la dirección D;

4o " : guardar resultado en la dirección D.

Por lo tanto hacen falta 3 registros adicionales que guarden:

* instrucción a ejecutar: registro de instrucción IR
* operando F: registro de datos B
* operando D: registro de datos A.

La carga de estos registros vendrá controlada por las señales $M \rightarrow IR$, $M \rightarrow B$ y $M \rightarrow A$ respectivamente.

Indicador de cero

Recordemos tambien que la instrucción BEQ utiliza el indicador de cero FZ como operando implícito para decidir si ha de pasar a ejecutar las instrucciones almacenadas a partir de la dirección D o no. Hace falta un registro que guarde el bit Z de salida de la ALU de manera que pueda ser consultado durante la ejecución de la siguiente instrucción (si es BEQ). La carga de este registro vendrá controlada por la señal $Z \rightarrow FZ$.

Direcciones de memoria

La dirección de memoria que puede lanzar la CPU puede proceder de:

* registro PC, a fin de buscar la instrucción a ejecutar;
* campo F de la instrucción, en la búsqueda del operando fuente;
* campo D de la instrucción, en la búsqueda del operando destino o para guardar el resultado de la instrucción.

Con el fin de seleccionar una de las tres direcciones anteriores usaremos un multiplexor de 4 canales controlado por 2 líneas de control, que llamaremos MX_1 y MX_0:

MX1	MX0	@
0	0	PC
0	1	no utilizada
1	0	F
1	1	D

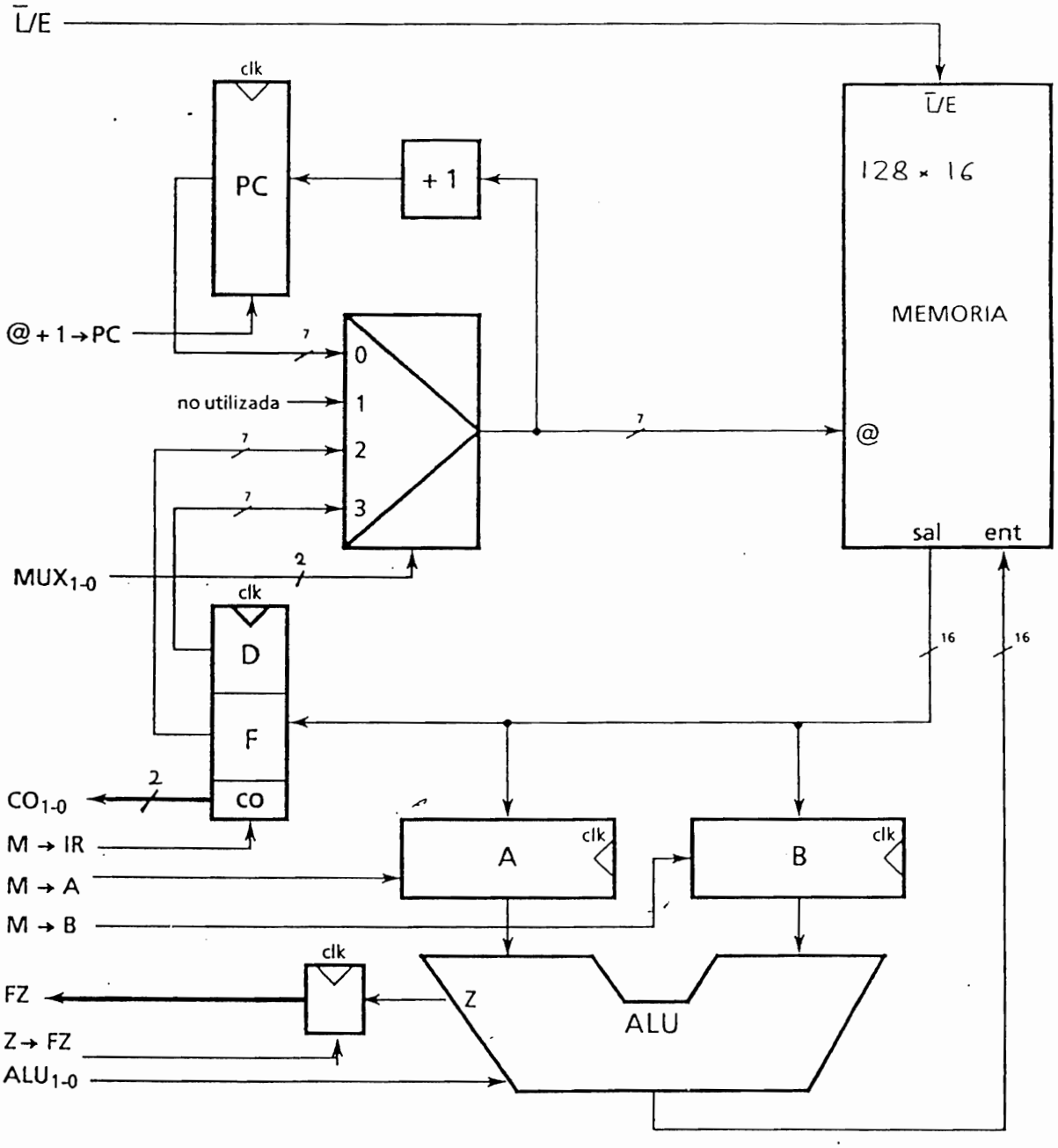
Cuando @=PC debemos incrementar su valor para que apunte a la siguiente instrucción, según el mecanismo de secuenciamiento implícito antes mencionado. La señal que controla la carga de este registro es $@+1 \rightarrow PC$.

Tipo de acceso

La CPU deberá indicar a la memoria el tipo de acceso que desea realizar, es decir, si hace una transferencia de memoria a CPU (lectura) o de CPU a memoria (escritura). La señal *L/E* controlará el tipo de transferencia.

Esquema de la UP

Con todo lo anterior ya tenemos diseñada la UP de nuestra máquina sencilla cuyo esquema a nivel de bloques que se muestra a continuación:



2.3 Diseño de la Unidad de Control.

Una vez diseñada la UP pasaremos a diseñar la UC encargada de controlar toda la lógica anterior. La UC puede diseñarse según dos filosofías básicas:

- **UC cableada:** diseño de un sistema secuencial definido por un grafo de estados asociados a cada fase de ejecución de las instrucciones. Cada estado lleva asociado un vector de salidas que controlan la UP y el resto de bloques del computador.

- **UC microprogramada:** diseño de un autómata controlado por un microprograma residente en una memoria interna que, para cada instrucción del programa, genera una secuencia de microinstrucciones asociadas.

En este apartado abordaremos la primera filosofía de diseño. Posteriormente en el apartado 4 se estudiará su diseño microprogramado.

La UC se diseñará como un sistema secuencial con 3 señales de entrada procedentes de la UP y 10 de salida que controlan los bloques descritos en la sección anterior.

Fases de ejecución

En general podemos decir que la ejecución de una instrucción está dividida en 4 fases:

Fase 1: búsqueda en memoria de la instrucción, almacenamiento en IR e incremento del PC (secuenciamiento implícito). A esta fase se la denomina fase de **fetch**.

Fase 2: **decodificación** de la instrucción.

Fase 3: **búsqueda de operandos** en memoria o evaluación de FZ.

Fase 4: **ejecución** de la instrución y almacenamiento de resultados en memoria.

El correcto secuenciamento de fases y la activación de los bloques adecuados de la UP lo realizará la UC siguiendo un determinado grafo de estados. Así por ejemplo, para ejecutar cada una de las instrucciones definidas se requieren los siguientes estados:

ADD

Definición de estados:

$estado_0$	(fase 1):	IR ← (PC);	PC = PC + 1 ;
$estado_1$	(fase 2):	evaluación de CO_0 y CO_1 ;	
$estado_2$,	(fase 3):	B ← (F);	
$estado_6$	(fase 3):	A ← (D);	
$estado_7$,	(fase 4):	D ← A + B ;	FZ ← Z

CMP

Definición de estados:

$estado_0$	(fase 1):	IR ← (PC);	PC = PC + 1 ;
$estado_1$	(fase 2):	evaluación de CO_0 y CO_1 ;	
$estado_3$	(fase 3):	B ← (F);	
$estado_8$	(fase 3):	A ← (D);	
$estado_9$	(fase 4):	A xor B ;	FZ ← Z

MOV

Definición de estados:

$estado_0$	(fase 1):	IR ← (PC);	PC = PC + 1 ;
$estado_1$	(fase 2):	evaluación de CO_0 y CO_1 ;	
$estado_4$	(fase 3):	B ← (F);	
$estado_{10}$	(fase 4):	D ← B ;	FZ ← Z

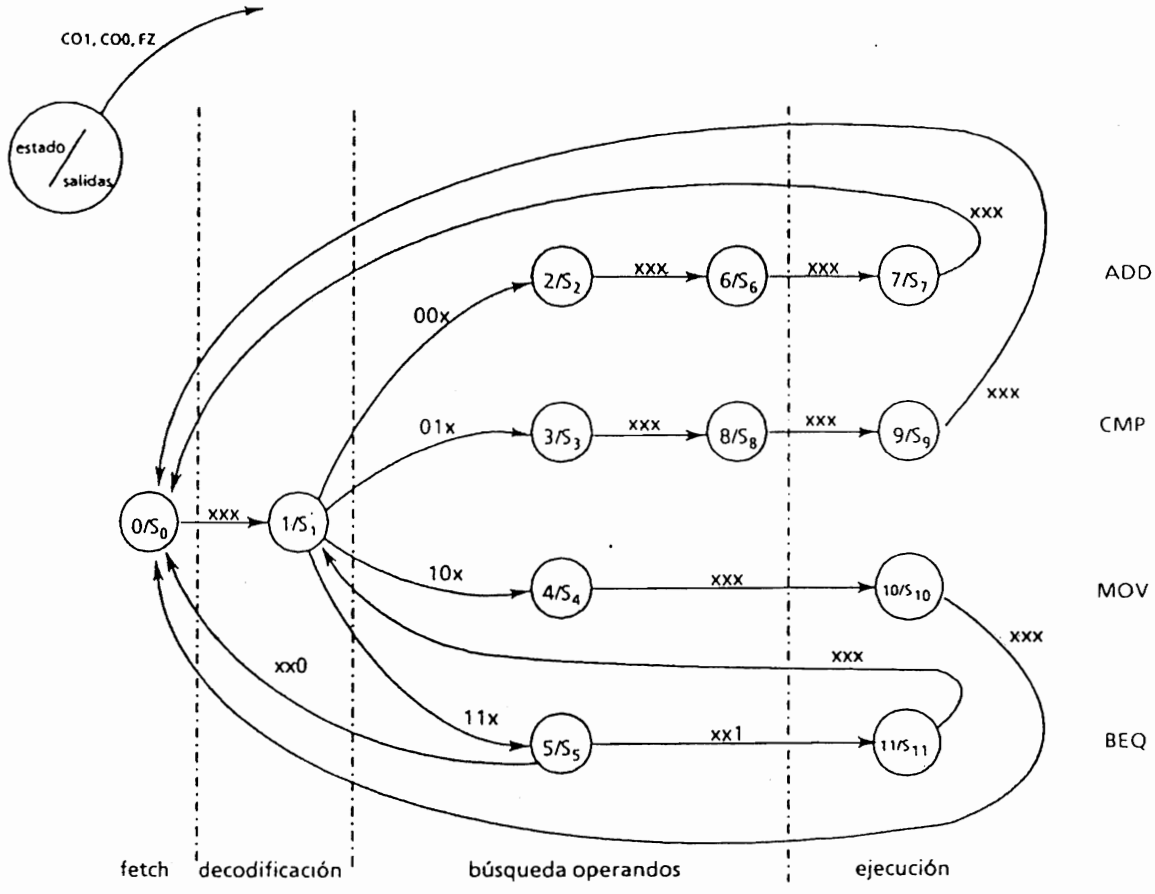
BEQ

Definición de estados:

$estado_0$	(fase 1):	IR ← (PC);	PC = PC + 1 ;
$estado_1$	(fase 2):	evaluación de CO_0 y CO_1 ;	
$estado_5$	(fase 3):	consulta de FZ ;	
$estado_{11}$	(fase 1):	IR ← (D) ;	PC ← D + 1

Observar que en el $estado_{11}$ se realiza el fetch de la instrucción que se encuentra en la dirección D, y por lo tanto, el PC debe quedar preparado apuntando a la siguiente instrucción en secuencia, es decir, D + 1.

Las transiciones entre estados del grafo vienen controladas por las señales de entrada a la UC. El grafo de estados completo se muestra en la siguiente figura:



En este grafo, cada estado lleva asociado un vector binario de salida con los valores mostrados en la siguiente tabla de salidas:

	S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	S_{11}
MX1	0	X	1	1	1	X	1	1	1	X	1	1
MX0	0	X	0	0	0	X	1	1	1	X	1	1
ALU1	X	X	X	X	X	X	X	0	X	0	1	X
ALU0	X	X	X	X	X	X	X	0	X	1	0	X
L/E	0	0	0	0	0	0	0	1	0	0	1	0
PC←@+1	1	0	0	0	0	0	0	0	0	0	0	1
IR←M	1	0	0	0	0	0	0	0	0	0	0	1
A←M	0	0	0	0	0	0	1	0	1	0	0	0
B←M	0	0	1	1	1	0	0	0	0	0	0	0
FZ←Z	0	0	0	0	0	0	0	1	0	1	1	0

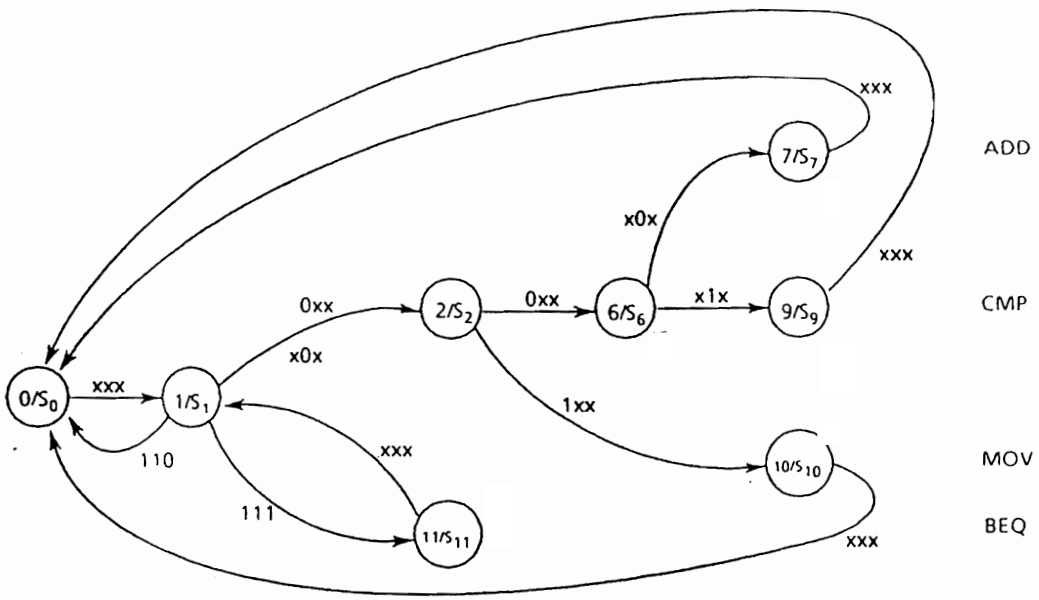
En principio tenemos un grafo con 12 estados aunque es posible simplificarlo, teniendo en cuenta las siguientes consideraciones:

* las instrucciones ADD, MOV y CMP poseen una fase común que es la busqueda del 1er operando. Por lo tanto, podemos juntar estos 3 estados idénticos (S_2, S_3, S_4) en uno sólo.

* las instrucciones ADD y CMP poseen otra fase común que es la búsqueda del 2º operando. Por lo tanto se pueden unificar los dos estados (S_6, S_8) en uno sólo.

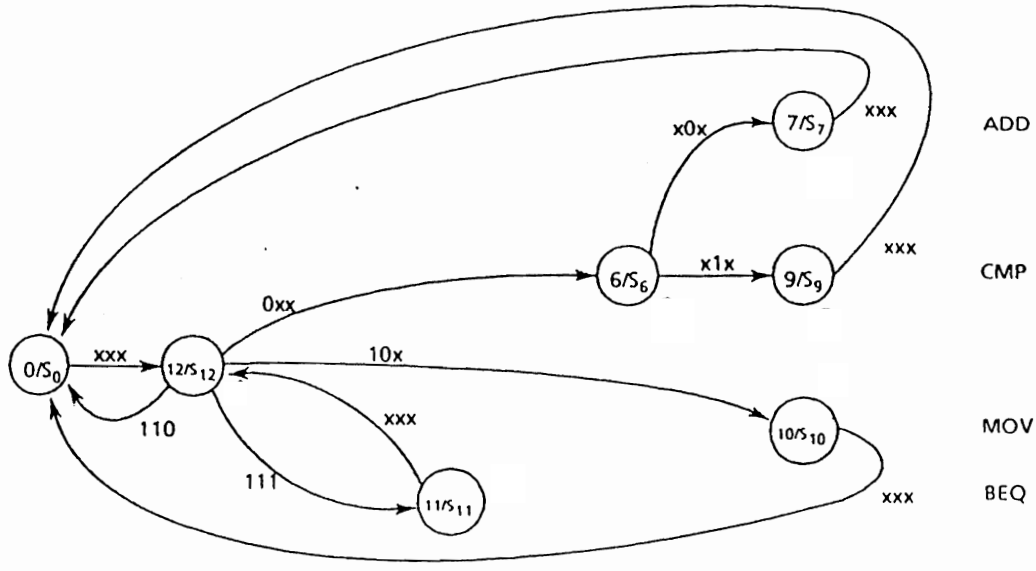
* la consulta de FZ para decidir si se efectúa el salto puede realizarse en S_1, evitandonos el estado S_5.

Asi el nuevo grafo de estados sería:



quedando en definitiva 8 estados y la tabla de salidas anterior eliminando los estados redundantes. En este caso, la fase de decodificación queda distribuida a lo largo de todo el grafo de estados. Por ejemplo, hasta que no se alcanza el estado S_6 no se sabe si la instrucción es CMP o ADD.

Una última optimización que podríamos realizar consistiría en unificar la decodificación con la búsqueda del 1er operando en un sólo estado (S_{12}), ya que durante el estado S_1 no se accede a memoria. Si después resulta que la instrucción es BEQ (para la cual no hace falta el operando), lo ignoramos. El grafo de estados reducido es el siguiente:



En este caso, el nuevo estado S_{12} tendrá asociado el siguiente vector de salidas:

	S_{12}
MX1	1
MX0	0
ALU1	X
ALU0	X
L/E	0
PC←@+1	0
IR←M	0
A←M	0
B←M	1
FZ←Z	0

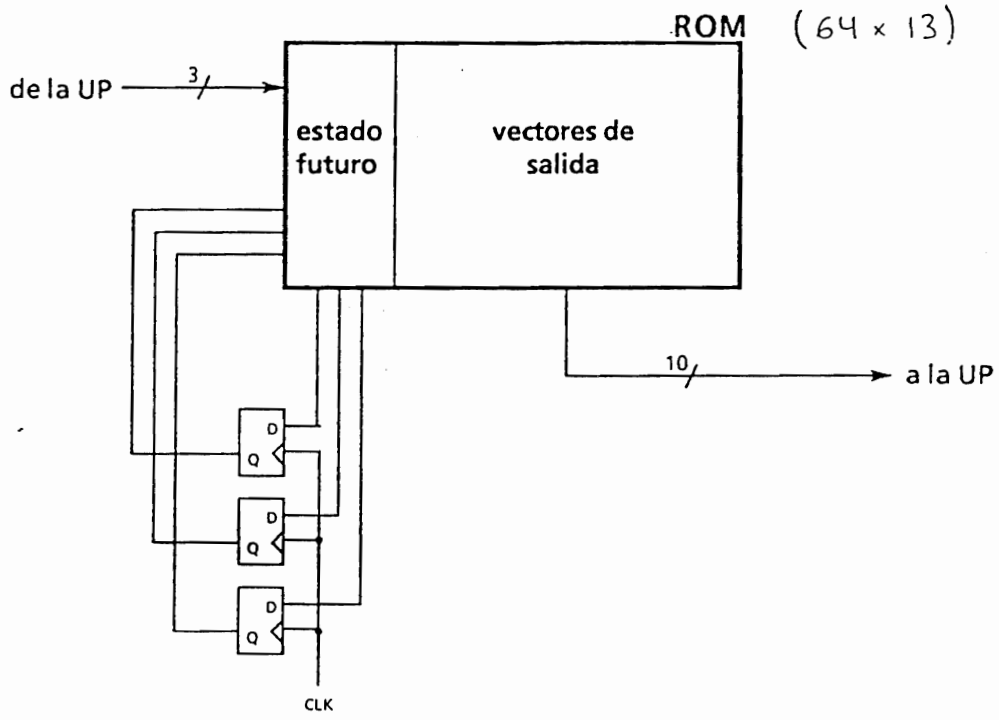
Este grafo de estados nos dice además cuantos ciclos de reloj tardará cada instrucción en ejecutarse (número de arcos en cada camino del grafo):

* ADD y CMP tardan 4 ciclos de reloj;
* MOV tarda 3 ciclos de reloj;
* BEQ tarda 2 ciclos de reloj. En caso de FZ=1, se ejecuta el fetch de la instrucción destino en el estado$_{11}$, contabilizándose como el primer ciclo de la ejecución de dicha instrucción.

y el número de accesos a memoria:

* ADD provoca 4 accesos a memoria;
* CMP y MOV provocan 3 accesos a memoria;
* BEQ provoca 2 accesos a memoria (aunque uno de ellos innecesario en la última versión optimizada).

El diseño de la UC se completa realizando su síntesis utilizando cualquiera de las técnicas estudiadas durante el curso (diseño con lógica discreta, ROM, ...). En este caso, y dado el elevado número de entradas y salidas del sistema, vamos a realizar su diseño utilizando una memoria ROM, tal y como se muestra a continuación:



A continuación se estudia como modificar la MS1 a fin de permitir nuevas instrucciones en el repertorio básico.

3. SOFTWARE, HARDWARE & FIRMWARE

En este apartado estudiamos diferentes alternativas a la hora de realizar modificaciones en el diseño original de nuestra máquina sencilla. Las alternativas posibles son las siguientes:

a) realización software;

b) modificación hardware;

c) modificación firmware.

Sus ventajas e inconvenientes se discutirán al final de este apartado. Para ello supondremos que deseamos realizar una instrucción adicional, que denominaremos *CUAD F, D* y que realiza la siguiente operación:

CUAD F, D $\quad D \leftarrow 4 \cdot (F);$ $FZ = 1 \; si \; (F) = 0$

3.1 Software

En este caso se trata de realizar un programa que usando las instrucciones básicas disponibles realice la operación deseada.

En el ejemplo que se considera, la secuencia de instrucciones que realizan la operación *CUAD a, b* podrían ser:

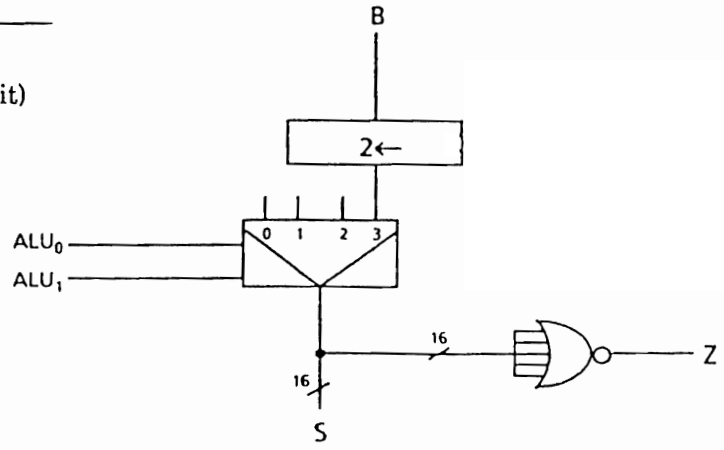
```
CUAD: MOV a, b
      ADD b, b
      ADD b, b
```

3.2 Hardware

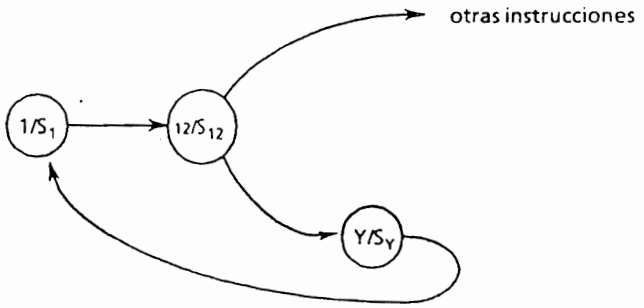
En este caso se trata de modificar la UC y la UP a fin de posibilitar la ejecución de la nueva instrucción.

En principio, para permitir realizar dicha operación es necesario modificar la ALU para que calcule el cuadruple de uno de sus registros de entrada. Esta modificación es posible dado que disponemos una combinación libre de las señales ALU_1 y ALU_0 que se puede utilizar para dicho fin. La multiplicación por cuatro puede realizarse lógicamente desplazando B combinacionalmente dos bits a la izquierda, tal y como muestra la siguiente figura:

ALU_1	ALU_0	operación
0	0	A+B
0	1	A xor B (bit a bit)
1	0	B
1	1	4·B



Por otro lado, es también necesario modificar el grafo de estados de la UC a fin de ejecutar la instrucción requerida. Suponiendo la última versión del grafo de estados, en la que se realiza la decodificación y búsqueda del operando fuente de forma simultanea, se obtiene el siguiente fragmento de grafo de estados:



	S_Y
MX1	1
MX0	1
ALU1	1
ALU0	1
L/E	1
PC←@+1	0
IR←M	0
A←M	0
B←M	0
FZ←Z	1

Observar que el hecho de añadir una nueva instrucción requiere también modificar el formato de la instrucción, entre otras cosas, dado que ahora se necesitan más bits para el código de operación. Estos aspectos serán considerados con más detalle en el siguiente apartado.

3.3 Firmware

En este caso se trata de modificar únicamente la UC, de manera que se realice la operación indicada utilizando los recursos que ofrece la UP, es decir, sin realizar ninguna modificación adicional en ella.

Para la instrucción que se trata, podríamos realizar la siguiente secuencia de estados:

Definición de estados

$estado_0$	(fase 1):	IR←(PC);	PC = PC + 1	
$estado_a$	(fase 2/3):	decodificación;	A←(F);	B←(F);
$estado_b$	(fase 4):	D←A + B		
$estado_c$	(fase 3):	A←(D);	B←(D);	
$estado_d$	(fase 4):	D←A + B		

Observar que, como en el caso anterior, es necesario realizar modificaciones adicionales que serán descritas en el siguiente apartado con más detalle.

3.4 Valoración

En función de las realizaciones anteriores podemos extraer algunas conclusiones de caracter general:

* La solución software es más barata, en cuanto a que no requiere modificar el hardware del computador. Sin embargo es más lenta en cuanto a que requiere un total de 11 ciclos de reloj para ejecutarse (3 ciclos para el MOV y (2·4) ciclos para las ADD).

* La solución hardware es la más cara pues require modificar tanto la UC como la UP. Sin embargo es la más rápida en cuanto a que require únicamente 3 ciclos para ser ejecutada.

* La solución firmware posee un coste intermedio, en cuanto a que únicamente requiere modificar la UC (p.e el contenido y tamaño de la ROM, añadir algún biestable, ...). También se obtiene una rapidez intermedia, dado qué en este caso se requieren 5 ciclos para su ejecución completa.

En el siguiente apartado abordamos una modificación completa eligiendo como opción la modificación hardware de la MS1.

4. MODIFICACION DE MS1

Se desea añadir nuevas instrucciones a la MS1 diseñada en el apartado2. Para ello vamos a realizar modificaciones en la UP y en la UC a fin de permitir la ejecución de las nuevas instrucciones. Se desea mantener el tamaño total de la instrucción y el espacio de direcciones accesible.

Nuevo repertorio de instrucciones

Las instrucciones que se añaden a la MS1 se describen a continuación y están caracterizadas por poseer un único operando en memoria (operando D):

* CLEAR D: puesta a 0 de la posición de memoria D sin alterar el flag FZ;

  $D \leftarrow 0$

* MOVD K, D: carga directa de la constante K en la dirección D;

  $D \leftarrow K;$ $\quad FZ \leftarrow K=0$

* ACUM K, D : acumular la constante K en la dirección D.

  $D \leftarrow (D) + K;$ $\quad FZ \leftarrow (D) + K=0$

Fijémonos que estas 2 últimas instrucciones necesitan una palabra de 16 bits que acompaña a la instrucción para especificar la constante K. Esta forma de indicar un operando en la instrucción se denomina **direccionamiento inmediato**, a diferencia del mecanismo descrito en el apartado anterior que se denomina direccionamiento absoluto o directo (mediante el cual la dirección del operando es la que se especifica en la instrucción).

Nuevos Formatos de instrucción

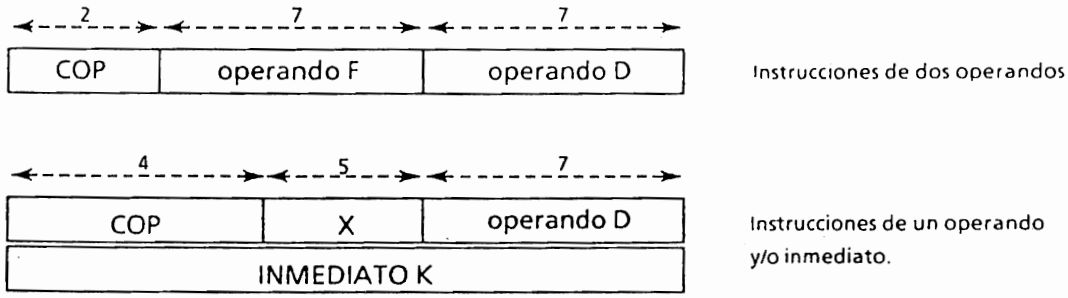
Fijémonos que las nuevas instrucciones no requieren los 7 bits del campo F del formato de instrucción descrito. Por lo tanto, estas instrucciones pueden utilizar estos bits del formato para la codificación del código de operación. Si ambos tipos de instrucciones deben tener el mismo formato de 16 bits, deberemos codificar de alguna manera si la instrucción posee 1 o 2 operandos. Una posible solución, denominada **codificación expandida**, consiste en dedicar una de las combinaciones de los bits de código de operación para diferenciar los dos tipos de instrucciones. Por ejemplo la combinación 11 (en binario) de estos bits puede ser utilizada para indicar que es una instrucción de

un operando y los bits que hagan falta del campo F para diferenciarlas entre ellas.

Una posible codificación para las nuevas instrucciones utilizaría:

- 4 bits de código de operación (los dos de más peso a 11);
- 5 bits no usados;
- 7 bits como @ de operando D.

Las instrucciones con operando en modo inmediato necesitan además 16 bits para la constante. Los nuevos formatos se muestran a continuación:



Los CO asociados a cada instrucción podrían ser los siguientes:

C_1	C_0	E_1	E_0	operación
0	0			ADD
0	1			CMP
1	0			MOV
1	1	0	0	BEQ
1	1	0	1	CLEAR
1	1	1	0	MOVD
1	1	1	1	ACUM

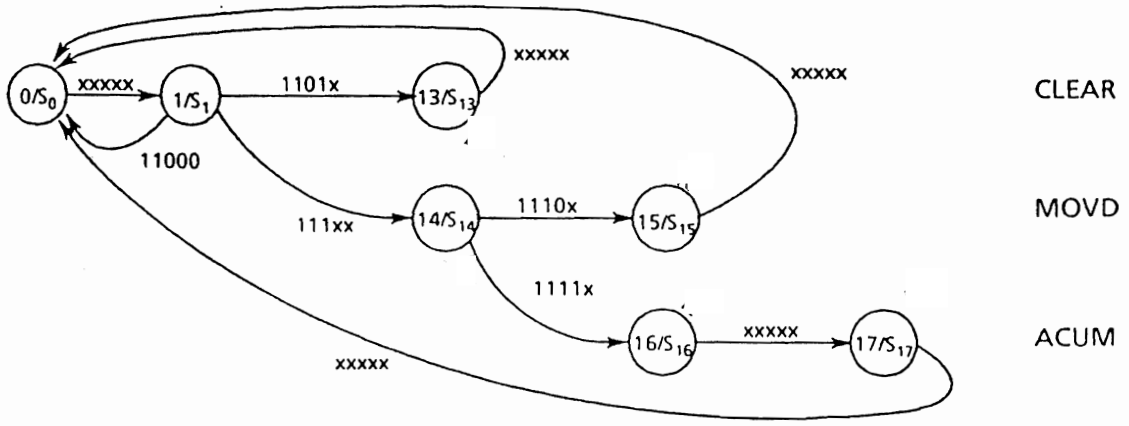
Modificaciones de la UP

Vamos a ver que modificaciones debemos hacer en la UP y UC de la MS1. Debemos modificar la ALU de la UP a fin de que permita poner un cero a su salida. De esta manera, las operaciones disponibles en la ALU serían:

ALU_1	ALU_0	operación
0	0	A+B
0	1	A xor B (bit a bit)
1	0	B
1	1	0

Modificaciones de la UC

Las modificaciones en la UC quedan especificadas en el nuevo grafo de estados en el que sólo se muestran los estados necesarios para las nuevas instrucciones (los arcos vienen etiquetados con CO_1, CO_0, E_1, E_0 y FZ):



La definición de los nuevos estados podría ser la mostrada a continuación:

Definición de estados

estado13	(fase 4):	D←0	
estado14	(fase 3):	B←(PC);	PC=PC+1; (búsqueda de K)
estado15	(fase 4):	D←B	
estado16	(fase 3):	A←(D);	(búsqueda del segundo operando)
estado17	(fase 4):	D←A+B	

y los vectores de salida asociados a dichos estados los siguientes:

	S_{13}	S_{14}	S_{15}	S_{16}	S_{17}
MX1	1	0	1	1	1
MX0	1	0	1	1	1
ALU1	1	X	1	X	0
ALU0	1	X	0	X	0
$\overline{L}$/E	1	0	1	0	1
PC←@+1	0	1	0	0	0
IR←M	0	0	0	0	0
A←M	0	0	0	1	0
B←M	0	1	0	0	0
FZ←Z	0	0	1	0	1

Fijémonos que cualquier modificación en la MS1 supone un rediseño prácticamente total de la UC cableada. A continuación vamos a estudiar una técnica de diseño para la UC que aumenta su flexibilidad a la hora de realizar modificaciones.

5. MICROPROGRAMACION

Acabamos de ver que una CPU no es más que una Unidad de Control (UC) y una Unidad de Proceso (UP). La UP está formada por un conjunto de bloques (combinacionales unos y secuenciales otros) y la UC es un sistema secuencial (autómata) que se encarga de generar la secuencia de señales necesaria para gobernar la UP de forma que se realize una operación determinada (instrucción).

Para el diseño de la UC se puede usar cualquiera de las técnicas de síntesis de sistemas secuenciales estudiadas durante el curso (síntesis mediante mapas de Karnaugh, ROM etc). Sin embargo, la UC de cualquier computador es un sistema secuencial con una característica muy importante: el número de salidas y de estados internos puede ser muy elevado. Esto implica que el diseño del sistema, utilizando cualquiera de los métodos que conocemos, puede ser muy costoso.

Por ejemplo, parece que la forma más sencilla de realización puede ser el uso de una ROM para construir las funciones combinacionales. En el caso de la máquina sencilla, y partiendo del grafo de estados simplificado, tenemos 7 estados, 3 entradas y 10 salidas. Esto significa que, si usamos una ROM, ésta ha de ser de 64 palabras de 13 bits cada una. Sin embargo, lo que realmente nos interesa de la UC son sus salidas, que gobiernan el funcionamiento de la UP. Vemos que, en nuestro caso, sólo hay 7 combinaciones diferentes de las salidas (dependiendo del estado en que nos encontramos).

Parece, por tanto, muy poco eficiente emplear una memoria con 64 palabras cuando sólo pueden ocurrir 7 cosas diferentes. Esta situación se agrava enormemente en el caso de máquinas más complicadas.

Para solucionar el problema vamos a ver una alternativa de diseño que se denomina **microprogramación**.

Secuencia de operaciones

Sabemos que la máquina sólo puede ejecutar 4 instrucciones diferentes. Para efectuar cada una de estas instrucciones se ha de generar una determinada secuencia de señales de control. Por ejemplo, para ejecutar la instrucción ADD se ha de generar la siguiente secuencia:

ADD

1- @ = PC, ALU = X, PC = @ + 1, IR = M, $\overline{L}$/E = 0 ;Fetch
2- @ = X, ALU = X ; Decodificación
3- @ = F, ALU = X, B = M, $\overline{L}$/E = 0 ; Búsqueda 1º operando
4- @ = D, ALU = X, A = M, $\overline{L}$/E = 0 ; Búsqueda 2 º operando
5- @ = D, ALU = +, $\overline{L}$/E = 1, FZ = Z ; Suma y carga del resultado

La secuencia para ejecutar CMP sería:

CMP

1- @ = PC, ALU = X, PC = @ + 1, IR = M, $\overline{L}$/E = 0 ;Fetch
2- @ = X, ALU = X ; Decodificación
3- @ = F, ALU = X, B = M, $\overline{L}$/E = 0 ; Búsqueda 1º operando
4- @ = D, ALU = X, A = M, $\overline{L}$/E = 0 ; Búsqueda 2 º operando
5- @ = X, ALU = XOR, FZ = Z ; Comparación

La secuencia para ejecutar MOV sería:

MOV

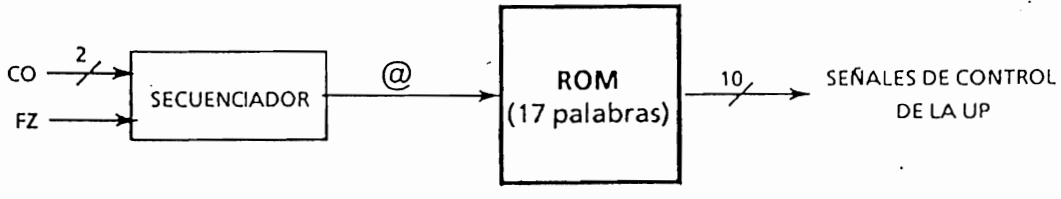
1- @ = PC, ALU = X, PC = @ + 1, IR = M, $\overline{L}$/E = 0 ;Fetch
2- @ = X, ALU = X ; Decodificación
3- @ = F, ALU = X, B = M, $\overline{L}$/E = 0 ; Búsqueda 1º operando
4- @ = D, ALU = B, L/E, FZ = Z, $\overline{L}$/E = 1 ; Mover

Por último, la secuencia para ejecutar BEQ (en caso de salto) sería:

BEQ

1- @ = PC, ALU = X, PC = @ + 1, IR = M, L/E = 0 ;Fetch
2- @ = X, ALU = X ; Decodificación
3- @ = D, ALU = X, PC = @ + 1 , IR = M, L/E = 0 ; Fetch de la siguiente instrucción

Supongamos que tenemos almacenadas en una ROM todas estas palabras (17 palabras de 10 bits cada una). La estructura básica de la unidad de control podría ser:



La parte fundamental es el **secuenciador**. Se trata de un circuito que ha de saber enviar la secuencia de direcciones adecuada para leer de la ROM las palabras de control en el orden correcto. La ROM tiene almacenados 4 grupos de palabras de control (uno para cada instrucción). El secuenciador debe ser capaz de detectar la instrucción que debe ejecutarse en un momento dado y leer de la ROM las palabras de control mediante las que se ejecuta dicha instrucción.

Los 4 grupos de palabras de control (que a partir de ahora llamaremos **microinstrucciones**) tienen una parte común (las microinstrucciones de fetch y decodificación). Podemos escribir las microinstrucciones en la ROM en el siguiente orden:

PALABRA	MICROINSTRUCCIÓN	
0	@ = PC, ALU = X, PC = @ + 1, IR = M, $\overline{L/E}$ = 0	;Fetch
1	@ = X, ALU = X	; Decodificación
2	@ = F, ALU = X, B = M, $\overline{L/E}$ = 0	; ADD
3	@ = D, ALU = X, A = M, $\overline{L/E}$ = 0	
4	@ = D, ALU = +, $\overline{L/E}$ = 1, FZ = Z	
5	@ = F, ALU = X, B = M, $\overline{L/E}$ = 0	;CMP
6	@ = D, ALU = X, A = M, $\overline{L/E}$ = 0	
7	@ = D, ALU = XOR, FZ = Z	
8	@ = F, ALU = X, B = M, $\overline{L/E}$ = 0	;MOV
9	@ = D, ALU = B, $\overline{L/E}$ = 1, FZ = Z	
10	@ = D, ALU = X, PC = @ + 1, IR = M, $\overline{L/E}$ = 0	; BEQ

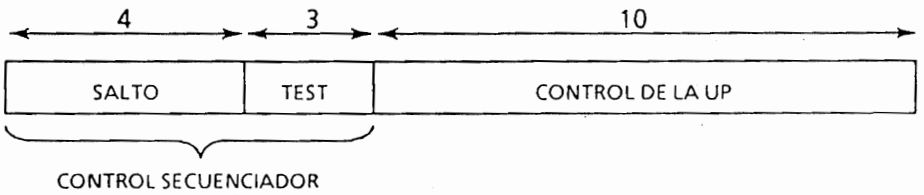
Si en la ROM tenemos estas 11 microinstrucciones el secuenciador se ha de comportar de la siguiente forma: En primer lugar ha de leer las microinstrucciones de fetch y decodificación. En función del valor del código de operación ha de leer la microinstrucción 2 (si CO = ADD), la microinstrucción 5 (si CO = CMP), la microinstrucción 8 (si CO = MOV), la microinstrucción 10 (si CO = BEQ y FZ = 1) o la microinstrucción 0 (si CO = BEQ y FZ = 0).

Si CO = ADD, después de la microinstrucción 2 ha de leer la 3, luego la 4 y finalmente la 0. En general, cuando se acaba la secuencia de microinstrucciones para ejecutar una determinada instrucción, se ha de leer la microinstrucción 0 para realizar el fetch de la siguiente instrucción salvo después de la 10 que va la 1.

Vemos que el secuenciador ha de leer las microinstrucciones de forma secuencial y en ocasiones ha de realizar una salto que rompe esta secuencia. Este salto dependerá a veces de los valores de CO_1, CO_0 y FZ y otras veces será un salto incondicional.

Supongamos que la microinstrucción, además de tener un conjunto de bits (10 bits) que gobiernan la UP, tiene un grupo de bits que van a ayudar al secuenciador a decidir cuál es la siguiente microinstrucción que debe leer.

Por ejemplo, la microinstrucción puede tener el siguiente formato:



Supongamos también que asignamos a cada una de las señales CO_0, CO_1 y FZ un código. Por ejemplo:

SEÑAL	CODIGO	
CO_0	00	
CO_1	01	
FZ	10	
ZERO	11	;ZERO es una señal que siempre vale 0

Cuando se lee una determinada microinstrucción, los bits del campo CONTROL determinan el funcionamiento de los bloques de la UP mientras que los bits de los campos TEST y SALTO son utilizados por el secuenciador para determinar la dirección de la siguiente microinstrucción a leer.

El cálculo de esta dirección se realiza de la forma siguiente: los dos bits bajos del campo TEST identifican a una de las 4 señales CO_0, CO_1, FZ o ZERO (según la codificación preestablecida). Si el valor de la señal seleccionada coincide con el valor del tercer bit del campo TEST entonces la dirección de la siguiente

microinstrucción está contenida en el campo SALTO. Si los valores indicados no coinciden se lee la siguiente microinstrucción en secuencia.

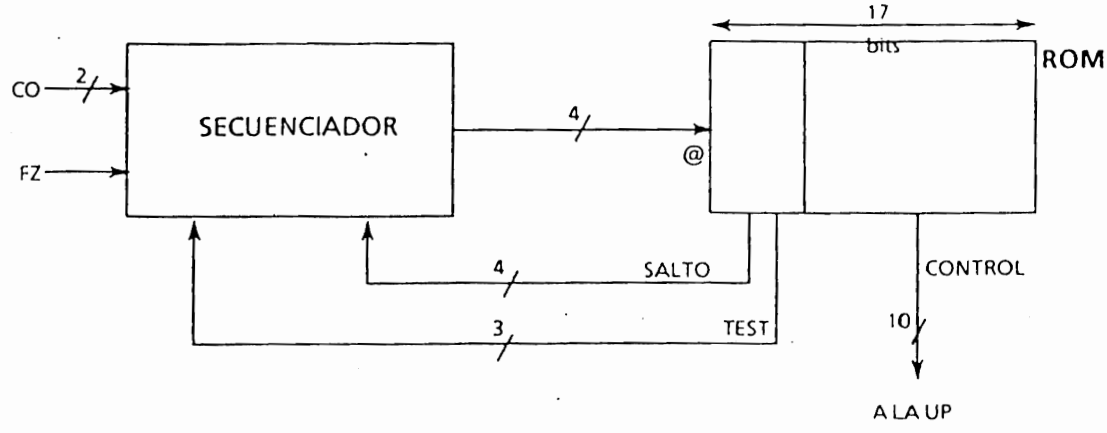
El contenido de la memoria ROM se muestra a continuación:

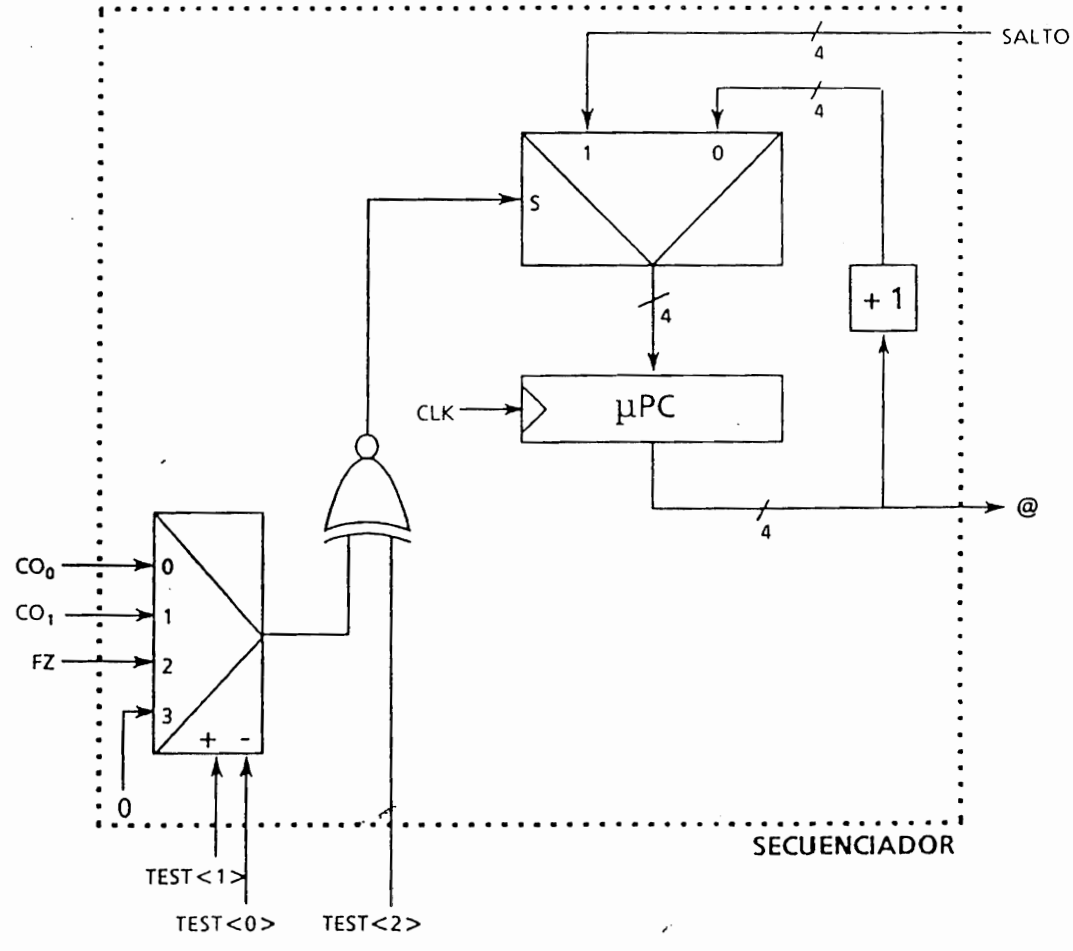
PALABRA	SALTO	TEST	CONTROL											
		2 1 0	MX1	MX0	ALU1	ALU0	L/E	PC	IR	A	B	FZ	
0	X X X X	1 1 1	0	0	X	X	0	1	1	0	0	0	fetch
1	1 0 0 0	1 0 0	X	X	X	X	0	0	0	0	0	0	si CO0 = 1 ir a 8
2	0 1 1 0	1 0 1	X	X	X	X	0	0	0	0	0	0	si CO1 = 1 ir a 6
3	X X X X	1 1 1	1	0	X	X	0	0	0	0	1	0	CO = 00: ADD
4	X X X X	1 1 1	1	1	X	X	0	0	0	1	0	0	buscamos operandos
5	0 0 0 0	0 1 1	1	1	0	0	1	0	0	0	0	1	suma, ir a 0
6	X X X X	1 1 1	1	0	X	X	0	0	0	0	1	0	CO = 10: MOV
7	0 0 0 0	0 1 1	1	1	1	0	1	0	0	0	0	1	mueve, flag FZ, ir a 0
8	1 1 0 0	1 0 1	X	X	X	X	0	0	0	0	0	0	Si CO1 = 1 ir a 12
9	X X X X	1 1 1	1	0	X	X	0	0	0	0	1	0	CO = 01; CMP
10	X X X X	1 1 1	1	1	X	X	0	0	0	1	0	0	buscamos operandos
11	0 0 0 0	0 1 1	X	X	0	1	0	0	0	0	0	1	XOR, flag FZ, ir a 0
12	0 0 0 0	0 1 0	X	X	X	X	0	0	0	0	0	0	CO = 11:BEQ,si FZ = 0 ir a 0
13	0 0 0 1	0 1 1	1	1	X	X	0	1	1	0	0	0	si FZ = 1 fetch, ir a 1

Fijémonos que para forzar la lectura de la siguiente microinstrucción en secuencia hacemos que el campo TEST sea 111. De esta forma, comparamos la señal ZERO con el valor 1. Estos valores son diferentes y por tanto no se efectúa el salto. En este caso, el contenido del campo SALTO es indiferente. Para forzar el salto incondicional hacemos que el campo TEST valga 011 con lo cual comparamos el valor de la señal ZERO con 0. Estos valores son iguales y por tanto se efectúa el salto.

La estructura de la UC y de la estructura interna del secuenciador se muestran en la página siguiente.

Como conclusión podríamos decir que una UC microprogramada será, en general, más lenta que una UC cableada, que puede haber sido diseñada mediante técnicas de minimización de funciones. Sin embargo, cuando las instrucciones del lenguaje máquina son complejas, la técnica de microprogramación permite realizar el diseño con más facilidad. Por otra parte, una UC microprogramada es más flexible a la hora de realizar modificaciones o incluir nuevas instrucciones. Basta para ello con modificar o añadir





microinstrucciones a la ROM que contiene los microprogramas. Sin embargo estas modificaciones en la UC cableada pueden representar prácticamente un rediseño total de la misma.