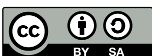
La arquitectura CISCA

Miquel Albert Orenga
Gerard Enrique Manonellas

PID_00218266

UOC
Universitat Oberta de Catalunya



Los textos e imágenes publicados en esta obra están sujetos –excepto que se indique lo contrario– a una licencia de Reconocimiento-Compartir igual (BY-SA) v.3.0 España de Creative Commons. Se puede modificar la obra, reproducirla, distribuirla o comunicarla públicamente siempre que se cite el autor y la fuente (FUOC. Fundació per a la Universitat Oberta de Catalunya), y siempre que la obra derivada quede sujeta a la misma licencia que el material original. La licencia completa se puede consultar en: http://creativecommons.org/licenses/by-sa/3.0/es/legalcode.ca

Índice

Introducción 5

Objetivos 6

1. Organización del computador 7
- 1.1. Procesador 8
  - 1.1.1. Organización de los registros 8
  - 1.1.2. Unidad aritmética y lógica 10
  - 1.1.3. Unidad de control 10
- 1.2. Memoria principal 12
  - 1.2.1. Memoria para la pila 12
  - 1.2.2. Memoria para la tabla de vectores de interrupción 13
- 1.3. Unidad de entrada/salida (E/S) 14
- 1.4. Sistema de interconexión (bus) 14

2. Juego de instrucciones 15
- 2.1. Operandos 15
- 2.2. Modos de direccionamiento 15
- 2.3. Instrucciones 18
  - 2.3.1. Instrucciones de transferencia de datos 18
  - 2.3.2. Instrucciones aritméticas 19
  - 2.3.3. Instrucciones lógicas 21
  - 2.3.4. Instrucciones de ruptura de secuencia 22
  - 2.3.5. Instrucciones de entrada/salida 24
  - 2.3.6. Instrucciones especiales 24
- 2.4. Estructuras de control 24
  - 2.4.1. Estructura if 24
  - 2.4.2. Estructura if-else 25
  - 2.4.3. Estructura while 26
  - 2.4.4. Estructura do-while 27
  - 2.4.5. Estructura for 27
  - 2.4.6. Estructura switch-case 28

3. Formato y codificación de las instrucciones 30
- 3.1. Codificación del código de operación. Byte B0 31
- 3.2. Codificación de los operandos. Bytes B1-B10 32
- 3.3. Ejemplos de codificación 37

4. Ejecución de las instrucciones 41
- 4.1. Lectura de la instrucción 41
- 4.2. Lectura de los operandos fuente 42

4.3. Ejecución de la instrucción y almacenamiento del operando destino 43
4.3.1. Operaciones de transferencia 44
4.3.2. Operaciones aritméticas y lógicas 45
4.3.3. Operaciones de ruptura de secuencia 46
4.3.4. Operaciones de Entrada/Salida 47
4.3.5. Operaciones especiales 48
4.4. Comprobación de interrupciones 48
4.5. Ejemplos de secuencias de microoperaciones 48
4.6. Ejemplo de señales de control y temporización 50

Introducción

Dada la gran variedad de procesadores comerciales y dada la creciente complejidad de estos, hemos optado por definir una máquina de propósito general que denominaremos **Complex Instruction Set Computer Architecture** (CISCA) y que utilizaremos en los ejemplos que nos ayudarán a entender mejor los conceptos que trataremos en esta asignatura y en los ejercicios que iremos proponiendo.

La arquitectura CISCA se ha definido siguiendo un modelo sencillo de máquina, del que solo definiremos los elementos más importantes; así será más fácil entender las referencias a los elementos del computador y otros conceptos referentes a su funcionamiento.

Se ha definido una arquitectura para trabajar los conceptos teóricos generales lo más parecida posible a la arquitectura x86-64 para facilitar el paso a la programación sobre esta arquitectura real. Esta será la arquitectura sobre la que se desarrollarán las prácticas.

Objetivos

Con los materiales didácticos de este módulo se pretende que los estudiantes alcancen los objetivos siguientes:

1. Conocer los elementos básicos de un computador sencillo y comprender su funcionamiento.

2. Conocer el juego de instrucciones de una arquitectura concreta con unas especificaciones propias.

3. Aprender los conceptos básicos de programación a partir de una arquitectura sencilla pero próxima a una arquitectura real.

4. Ser capaz de convertir el código ensamblador que genera el programador en código máquina que pueda interpretar el computador.

5. Entender qué hace cada una de las instrucciones de un juego de instrucciones en ensamblador y ver qué efectos tiene sobre los diferentes elementos del computador.

6. Entender el funcionamiento de una unidad de control microprogramada para una arquitectura concreta.

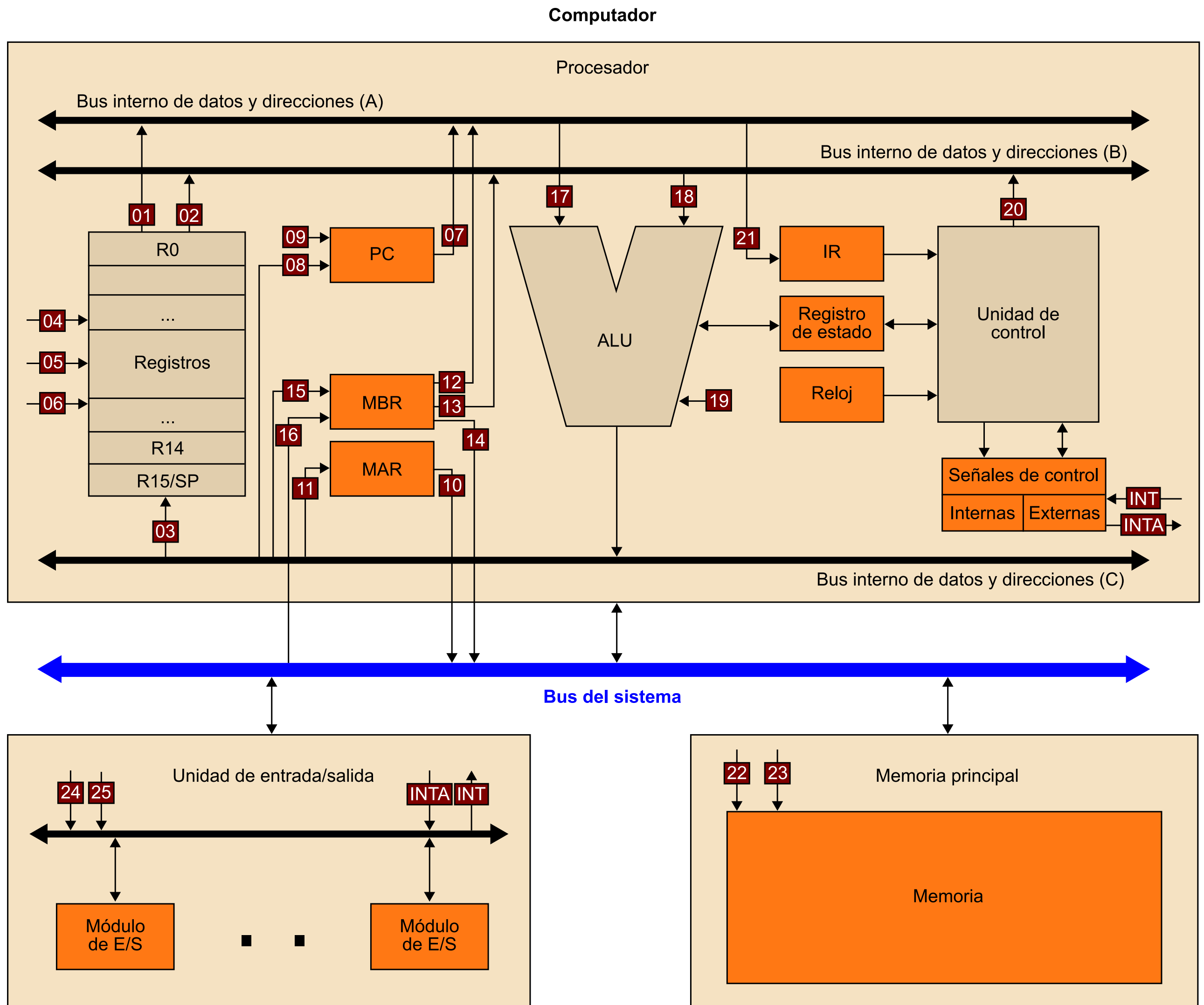
1. Organización del computador

El computador se organiza en unidades funcionales que trabajan independientemente y que están interconectadas por líneas, habitualmente denominadas buses, lo que nos permite describir el comportamiento funcional del computador, que da lugar a las especificaciones de la arquitectura del computador.

Tal como se ve en la figura siguiente, las unidades funcionales principales de un computador son:

- Procesador (CPU)
  - Registros
  - Unidad aritmética y lógica (ALU)
  - Unidad de control (UC)

- Unidad de memoria principal (Mp)
- Unidad de entrada/salida (E/S)
- Sistema de interconexión (Bus)



1.1. Procesador

1.1.1. Organización de los registros

Todos los registros son de 32 bits. Los bits de cada registro se numeran del 31 (bit de más peso) al 0 (bit de menos peso). Los registros del procesador (CPU) son de cuatro tipos:

1) Registros de propósito general. Hay 16 registros de propósito general, de R0 a R15. El registro R15 es especial. Este registro se utiliza de manera implícita en las instrucciones PUSH, POP, CALL y RET, pero también se puede utilizar como registro de propósito general.

> **Registro R15**
>
> El registro R15 se puede denominar también StackPointer (SP).

2) Registros de instrucción. Los dos registros principales relacionados con el acceso a las instrucciones son el contador de programa (PC) y el registro de instrucción (IR).

El registro PC tendrá un circuito autoincrementador. Dentro del ciclo de ejecución de la instrucción, en la fase de lectura de la instrucción, el PC quedará incrementado en tantas unidades como bytes tiene la instrucción. El valor del PC a partir de este momento, y durante el resto de las fases de la ejecución de la instrucción, se denota como PC_{up} (PC updated) y apunta a la dirección de la instrucción siguiente en la secuencia.

3) Registros de acceso a memoria. Hay dos registros necesarios para cualquier operación de lectura o escritura en memoria: el registro de datos de la memoria (MBR) y el registro de direcciones de la memoria (MAR).

4) Registros de estado y de control. Los bits del registro de estado son modificados por el procesador como resultado de la ejecución de instrucciones aritméticas o lógicas. Estos bits son parcialmente visibles al programador mediante las instrucciones de salto condicional.

El registro de estado incluye los bits de resultado de cero, transporte, desbordamiento y signo.

Registros visibles al programador

Los registros de propósito general son los únicos registros visibles al programador, el resto de los registros que se explican a continuación no son visibles.

Bits de resultado

- **Bit de cero (Z)**: se activa si el resultado obtenido es cero.
- **Bit de transporte (C)**: también llamado *carry* en la suma y *borrow* en la resta.
  Se activa si en el último bit que operamos en una operación aritmética se produce transporte. Se activa si al final de la operación nos llevamos una según el algoritmo de suma y resta tradicional operando en binario.

Bit de transporte en la resta

El bit de transporte, también llamado *borrow*, se genera según el algoritmo convencional de la resta. Pero si hacemos la resta A – B sumando el complementario del sustraendo, A + (–B), el bit de transporte de la resta (*borrow*) será el bit de transporte (*carry*) negado obtenido de hacer la suma con el complementario (*borrow = carry negado*), salvo los casos en los que B = 0, en los que entonces tanto el *carry* como el *borrow* son iguales y valen 0.

- **Bit de desbordamiento (V)**: también denominado *overflow*. Se activa si la última operación ha producido desbordamiento según el rango de representación utilizado. Para representar el resultado obtenido, en el formato de complemento a 2 con 32 bits, necesitaríamos más bits de los disponibles.
- **Bit de signo (S)**: activo si el resultado obtenido es negativo. Si el bit más significativo del resultado es 1.

Bits de resultado activos

Consideramos que los bits de resultado son activos cuando valen 1, e inactivos cuando valen 0.

- **Bit para habilitar las interrupciones (IE)**: si está activo, permite las interrupciones; si está inactivo, no se permiten las interrupciones.

- **Bit de interrupción (IF)**: si hay una petición de interrupción se activa.

1.1.2. Unidad aritmética y lógica

La unidad aritmética y lógica (ALU) es la encargada de hacer las operaciones aritméticas y las operaciones lógicas, considerando números de 32 bits en complemento a 2 (Ca2). Para hacer una operación, tomará los operandos fuente del bus interno A y B y depositará el resultado en el bus interno C.

Cuando se ejecuta una instrucción que hace una operación aritmética o lógica, la unidad de control deberá determinar qué operación hace la ALU, pero también qué registro deposita el dato en el bus A, qué registro en el bus B y en qué registro se almacenará el resultado generado sobre el bus C.

Cada una de las operaciones que hará la ALU puede ser implementada de diferentes maneras y no analizaremos la estructura de cada módulo, sino que nos centraremos solo en la parte funcional descrita en las propias instrucciones de esta arquitectura.

1.1.3. Unidad de control

La unidad de control (UC) es la unidad encargada de coordinar el resto de los componentes del computador mediante las señales de control.

Esta arquitectura dispone de una unidad de control microprogramada en la que la función básica es coordinar la ejecución de las instrucciones, determinando qué operaciones (denominadas *microoperaciones*) se hacen y cuándo se hacen, activando las *señales de control* necesarias en cada momento.

Los tipos de señales que tendremos en la unidad de control son:

1) Señales de entrada.

a) Temporización.
b) Registro de instrucción (IR).
c) Registro de estado.
d) Señales externas de la CPU.

2) Señales de salida y de control.

a) Internas a la CPU:

- Acceso a los buses internos.
- Control de la ALU.
- Control de otros elementos de la CPU.

b) Externas a la CPU:

- Acceso al bus externo.
- Control de la memoria.
- Control de los módulos de E/S.

La tabla siguiente muestra las señales más importantes para el control del computador en esta arquitectura.

	Señal	Observaciones	Dispositivos afectados
01	$R_{outAenable}$		Banco de registros
02	$R_{outBenable}$		
03	$R_{inCenable}$		
04	Ri_{outA}	Son 16 * 3 = 48 señales. Una señal de cada tipo y para cada registro.	
05	Ri_{outB}		
06	Ri_{inC}		
07	PC_{outA}		Registro PC
08	PC_{inC}		
09	$PC_{+\Delta}$	Para indicar incremento del registro PC (0-4)	
10	MAR_{outEXT}		Registro MAR
11	MAR_{inC}		
12	MBR_{outA}		Registro MBR
13	MBR_{outB}		
14	MBR_{outEXT}		
15	MBR_{inC}		
16	MBR_{inEXT}		
17	ALU_{inA}		ALU
18	ALU_{inB}		
19	ALU_{op}	Señales que codifican las operaciones de la ALU. 4bits→ 2^4 = 16 operaciones diferentes	

	Señal	Observaciones	Dispositivos afectados
20	IR_{outB}	Para poder poner los valores inmediatos de la instrucción en el bus interno B.	Registro IR
21	IR_{inA}		
22	Read		Memoria
23	Write		
24	Read E/S		Sistema de E/S
25	Write E/S		

1.2. Memoria principal

Hay 2^{32} posiciones de memoria de un byte cada una (4 GBytes de memoria). A los datos siempre se accede en palabras de 32 bits (4 bytes). El orden de los bytes en un dato de 4 bytes es en formato Little-Endian, es decir, el byte de menos peso se almacena en la dirección de memoria más pequeña de las 4.

Formato Little-Endian

Queremos guardar el valor 12345678h en la dirección de memoria 00120034h con la siguiente instrucción:

MOV [00120034h], 12345678h

Como este valor es de 32 bits (4 bytes) y las direcciones de memoria son de 1 byte, necesitaremos 4 posiciones de memoria para almacenarlo, a partir de la dirección especificada (00120034h). Si lo guardamos en formato Little-Endian, quedará almacenado en la memoria de la siguiente manera:

Memoria	
Dirección	**Contenido**
00120034h	78 h
00120035h	56 h
00120036h	34 h
00120037h	12 h

1.2.1. Memoria para la pila

Se reserva para la pila una parte de la memoria principal (Mp), desde la dirección FFFF0000h a la dirección FFFFFFFFh, disponiendo de una pila de 64 Kbytes. El tamaño de cada dato que se almacena en la pila es de 32 bits (4 bytes).

El registro SP (registro R15) apunta siempre a la cima de la pila. La pila crece hacia direcciones pequeñas. Para poner un elemento en la pila primero decrementaremos el registro SP y después guardaremos el dato en la dirección de

memoria que indique el registro SP, y si queremos sacar un elemento de la pila, en primer lugar leeremos el dato de la dirección de memoria que indica el registro SP y después incrementaremos el registro SP.

El valor inicial del registro SP es 0; eso nos indicará que la pila está vacía. Al poner el primer elemento en la pila, el registro SP se decrementa en 4 unidades (tamaño de la palabra de pila) antes de introducir el dato; de este modo, al poner el primer dato en la pila esta quedará almacenada en las direcciones FFFFFFFCh - FFFFFFFFh en formato Little-Endian, y el puntero SP valdrá FFFFFFFCh (= 0 - 4 en Ca2 utilizando 32 bits), el segundo dato en las direcciones FFFFFFF8h - FFFFFFFBh y SP valdrá FFFFFFF8h, y así sucesivamente.

Memoria principal (4Gbytes)	
Dirección	**Contenido**
00000000h	Tabla de vectores de interrupción (256 bytes)
...	
00000FFh	
0000100h	Código y datos
...	
FFFEFFFFh	
FFFF0000h	Pila (64Kbytes)
...	
FFFFFFFFh	

1.2.2. Memoria para la tabla de vectores de interrupción

Se reserva para la tabla de vectores una parte de la memoria principal, desde la dirección 00000000h a la dirección 000000FFh, por lo que se dispone de 256 bytes para la tabla de vectores de interrupción. En cada posición de la tabla almacenaremos una dirección de memoria de 32 bits (4 bytes), dirección de inicio de cada RSI, pudiendo almacenar hasta 64 (256/4) direcciones diferentes.

1.3. Unidad de entrada/salida (E/S)

La memoria de E/S dispone de 2^{32} puertos de E/S. Cada puerto corresponde a un registro de 32 bits (4 bytes) ubicado en uno de los módulos de E/S. Cada módulo de E/S puede tener asignados diferentes registros de E/S. Podemos utilizar todos los modos de direccionamiento disponibles para acceder a los puertos de E/S.

1.4. Sistema de interconexión (bus)

En este sistema dispondremos de dos niveles de buses: los buses internos del procesador, para interconectar los elementos de dentro del procesador, y el bus del sistema, para interconectar el procesador, la memoria y el sistema de E/S:

- Bus interno del procesador (tendremos 3 buses de 32 bits que los podremos utilizar tanto para datos como para direcciones).

- Bus externo del procesador (tendremos 1 bus de 32 bits que los podremos utilizar tanto para datos como para direcciones).

- Líneas de comunicación o de E/S (tendremos 1 bus de 32 bits que los podremos utilizar tanto para datos como para direcciones).

2. Juego de instrucciones

Aunque el juego de instrucciones de esta arquitectura tiene pocas instrucciones, posee muchas de las características de una arquitectura CISC, como instrucciones de longitud variable, operando destino implícito igual al primer operando fuente explícito, posibilidad de hacer operaciones aritméticas y lógicas con operandos en memoria, etc.

Nota

El juego de instrucciones de esta arquitectura tiene pocas instrucciones a fin de que la arquitectura sea pedagógica y sencilla.

2.1. Operandos

Las instrucciones pueden ser de 0, 1 o 2 operandos explícitos, y al ser una arquitectura con un modelo registro-memoria, en las instrucciones con dos operandos explícitos solo uno de los dos operandos puede hacer referencia a la memoria; el otro será un registro o un valor inmediato. En las instrucciones de un operando, este operando puede hacer referencia a un registro o a memoria.

Podemos tener dos tipos de operandos:

1) Direcciones: valores enteros sin signo [0... (2^{32} – 1)] se codificarán utilizando 32 bits.

2) Números: valores enteros con signo [–2^{31} ... + (2^{31} – 1)] se codifican utilizando Ca2 en 32 bits. Un valor será negativo si el bit de signo (bit de más peso, bit 31) es 1, y positivo en caso contrario.

2.2. Modos de direccionamiento

Los modos de direccionamiento que soporta CISCA son los siguientes:

1) Inmediato. El dato está en la propia instrucción.

El operando se expresa indicando:

- Número decimal. Se puede expresar un valor negativo añadiendo el signo '–' delante del número.
- Número binario finalizado con la letra 'b'.
- Número hexadecimal finalizado con la letra 'h'.
- Etiqueta, nombre de la etiqueta sin corchetes.
- Expresión aritmética.

Expresiones aritméticas

En los diferentes modos de direccionamiento se pueden expresar direcciones, valores inmediatos y desplazamientos, como expresiones aritméticas formadas por etiquetas, valores numéricos y operadores (+ – * /). Pero hay que tener presente que el valor que representa esta expresión se debe poder codificar en 32 bits si es una dirección o un inmediato,

y en 16 bits si es un desplazamiento. La expresión se debe escribir entre paréntesis. Por ejemplo:

```
MOV R1 ((00100FF0h+16)*4)
MOV R2, [Vec+(10*4)]
MOV [R8+(25*4)],R1
```

Cuando se expresa un número no se hace extensión de signo, se completa el número con ceros; por este motivo, en binario y hexadecimal, los números negativos se deben expresar en Ca2 utilizando 32 bits (32 dígitos binarios u 8 dígitos hexadecimales respectivamente).

La etiqueta, sin corchetes, especifica una dirección que se debe codificar utilizando 32 bits, tanto si representa el número de una variable como el punto del código al que queremos ir. Salvo las instrucciones de salto condicional, la etiqueta se codifica como un desplazamiento y utiliza direccionamiento relativo a PC, como se verá más adelante.

```
MOV R1, 100            ; carga el valor 100 (00000064h) en el registro R1.
MOV R1, -100           ; carga el valor -100 (FFFFFF9Ch) en el registro R1.
MOV R1, FF9Ch.         ; carga el valor F9Ch (0000FF9Ch) en el registro R1.
MOV R1, FFFFFF9Ch      ; es equivalente a la instrucción donde cargamos -100 en R1.
MOV R1, 1001 1100b.    ; carga el valor 9Ch (0000009Ch) en el registro R1.

MOV R1 1111 1111 1111 1111 1111 1111 1001 1100b ; carga el valor FFFFFF9Ch en el
                      ; registro R1, es equivalente a la instrucción en la que cargamos -100 en R1.
MOV R1, var1;         ; carga la dirección, no el valor que contiene la variable, en el registro R1.
JMP bucle             ; carga la dirección que corresponde al punto de código en el que hemos puesto
                       ; la etiqueta bucle en el registro PC.
MOV R1 ((00100FF0h+16)*4); carga en R1 el valor 00404000h
```

2) Registro. El dato está almacenado en un registro.

El operando se expresa indicando el nombre de un registro: Ri.

```
INC R2       ; el contenido del registro R2 se incrementa en una unidad.
ADD R2, R3   ; se suma el contenido del registro R3 al registro R2: R2 = R2 + R3.
```

3) Memoria. El dato está almacenado en la memoria.

El operando se expresa indicando la dirección de memoria en la que está el dato, una etiqueta como número de una variable o de manera más genérica una expresión aritmética entre corchetes: [dirección] [nombre_variable] [(expresión)].

```
MOV R1, [C0010020h]    ; como cada dirección de memoria se corresponde a 1 byte y
                       ; el operando destino, R1, es de 4 bytes, los valores almacenados
                       ; en las posiciones C0010020h - C0010023h se mueven hacia R1.
MOV R1, [var1];        ; carga el contenido de la variable var1 en el registro R1.
```

```
MOV R2, [Vec+(10*4)]   ; carga el contenido de la dirección Vec+40 en el registro R2.
```

4) Indirecto. El dato está almacenado en la memoria.

El operando se expresa indicando un registro que contiene la dirección de memoria en la que está el operando: [Registro].

```
MOV R3, var1     ; carga la dirección, no el valor que contiene, en el registro R3.
MOV R1, [R3]     ; R3 contiene la dirección de la posición de memoria del dato que se debe cargar
                 ; en el registro R1. R1 = M(R3). Como en R3 hemos cargado la dirección de var1
MOV R1, [var1]   ; es equivalente a cargar el contenido de la variable var1 en el registro R1
```

Nota

var1: direccionamiento inmediato; [R3] direccionamiento indirecto; [var1] direccionamiento a memoria.

5) Relativo. El dato está almacenado en la memoria.

La dirección en la que está el operando se determina sumando el valor del registro y el desplazamiento indicado de 16 bits. En ensamblador se indica utilizando [Registro + Desplazamiento]. El desplazamiento se puede escribir como una expresión aritmética.

```
MOV R1,[R2 + 100]     ; si R2 = 1200, el operando es M(1200 + 100)
                      ; es decir, el dato está en la dirección de memoria M(1300).
MOV [R2+(25*4)], R8   ; carga el valor del registro R8 en la dirección de memoria M(1200+100)
```

6) Indexado. El dato está almacenado en la memoria.

La dirección en la que está el operando se determina sumando la dirección de memoria indicada de 32 bits y el valor del registro. En ensamblador se indica utilizando: [Dirección + Registro] [nombre_variable+registro] [(expresión)+Registro]

```
MOV R1, [BC000020h + R5]      ; si R5 = 1Bh, el operando está en la dirección de memoria M(BC00003Bh).
MOV R1, [vector + R3]         ; si R3 = 08h y la dirección de vector=AF00330Ah, el valor que
                              ; cargamos en R1 está en la dirección de memoria M(AF003312h).
MOV R1 [(vector+00100200h) + R3] ; cargamos en R1 el valor que se encuentra en M(AF103512h).
```

7) Relativo a PC. Este modo de direccionamiento solo se utiliza en las instrucciones de salto condicional.

El operando se expresa indicando la dirección de memoria a la que se quiere dar el salto, una etiqueta que indique una posición dentro del código o, de manera más genérica, una expresión aritmética: etiqueta o (expresión).

```
JE etiqueta     ; se carga en el PC la dirección de la instrucción indicada por la etiqueta.
```

8) A pila. El direccionamiento a pila es un modo de direccionamiento implícito; es decir, no hay que hacer una referencia explícita a la pila, sino que trabaja implícitamente con la cima de la pila a través del registro SP (R15).

Al ser un modo de direccionamiento implícito, solo se utiliza en las instrucciones PUSH (poner un elemento en la pila) y POP (sacar un elemento de la pila).

La instrucción **PUSH fuente** hace lo siguiente:

SP = SP – 4

M[SP] = fuente

La instrucción **POP destino** hace lo siguiente:

destino = M[SP]

SP = SP + 4

```
PUSH 00123400h
PUSH R1
POP [var1]
POP [R2+16]
```

2.3. Instrucciones

2.3.1. Instrucciones de transferencia de datos

- **MOV destino, fuente**. Mueve el dato al que hace referencia el operando fuente a la ubicación en la que especifica el operando destino (destino ← fuente).

- **PUSH fuente**. Almacena el operando fuente (que representa un dato de 32 bits) en la cima de la pila. Primero decrementa SP (registro R15) en 4 unidades y a continuación guarda el dato que hace referencia al operando fuente en la posición de la pila apuntada por SP.

- **POP destino**. Recupera sobre el operando destino el valor almacenado en la cima de la pila (que representa un dato de 32 bits). Recupera el contenido de la posición de la pila que apunta a SP (registro R15) y lo guarda donde indique el operando destino; después incrementa SP en 4 unidades.

2.3.2. Instrucciones aritméticas

Las instrucciones aritméticas y de comparación operan considerando los operandos y el resultado como enteros de 32 bits en Ca2. Activan los bits de resultado según el resultado obtenido. Estos bits de resultado los podremos consultar utilizando las instrucciones de salto condicional.

- **ADD destino, fuente**. Hace la operación destino = destino + fuente.

- **SUB destino, fuente**. Hace la operación destino = destino – fuente2.

- **MUL destino, fuente**. Hace la operación destino = destino * fuente. Si el resultado no se puede representar en 32 bits, se activa el bit de desbordamiento.

- **DIV destino, fuente**. Hace la operación destino/fuente, división entera que considera el residuo con el mismo signo que el dividendo. El cociente se guarda en destino y el residuo se guarda en R0. Solo se produce desbordamiento en el caso $-2^{31}/-1$. Si fuente = 0 no es por la división, y para indicarlo se activa el bit de transporte (en los procesadores reales este error genera una excepción que gestiona el sistema operativo).

- **INC destino**. Hace la operación destino = destino + 1.

- **DEC destino**. Hace la operación destino = destino – 1.

- **CMP destino, fuente**. Compara los dos operandos mediante una resta: destino – fuente, y actualiza los bits de resultado. Los operandos no se modifican y el resultado no se guarda.

- **NEG destino**. Hace la operación destino = 0 – destino.

Bits de resultado

Todas las instrucciones aritméticas pueden modificar los bits de resultado del registro de estado según el resultado obtenido.

Instrucción	Z	S	C	V
ADD	x	x	x	x
SUB	x	x	x	x
MUL	x	x	-	x
DIV	x	x	x	x
INC	x	x	x	x
DEC	x	x	x	x

Notación: x significa que la instrucción modifica el bit de resultado; - significa que la instrucción no modifica este bit; 0 indica que la instrucción pone este bit a 0.

Instrucción	Z	S	C	V
CMP	x	x	x	X
NEG	x	x	x	x

Notación: x significa que la instrucción modifica el bit de resultado; - significa que la instrucción no modifica este bit; 0 indica que la instrucción pone este bit a 0.

Ejemplo

En este ejemplo se muestra cómo funcionan los bits de resultado de acuerdo con las especificaciones de la arquitectura CISCA, pero utilizando registros de 4 bits en lugar de registros de 32 bits para facilitar los cálculos. Consideremos los operandos R1 y R2 registros de 4 bits que representan valores numéricos en complemento a 2 (rango de valores desde –8 hasta +7). El valor inicial de todos los bits de resultado por cada instrucción es 0.

	Resultado	R1 = 7, R2 = 1				Resultado	R1 = –1, R2 = –7			
		Z	S	C	V		Z	S	C	V
ADD R1, R2	R1 = –8	0	1	0	1	R1 = –8	0	1	1	0
SUB R1, R2	R1 = +6	0	0	0	0	R1 = +6	0	0	0	0
SUB R2, R1	R2 = –6	0	1	1	0	R2 = –6	0	1	1	0
CMP R2, R1	R2 = 1	0	1	1	0	R2 = –7	0	1	1	0
NEG R1	R1 = –7	0	1	1	0	R1 = +1	0	0	1	0
INC R1	R1 = –8	0	1	0	1	R1 = 0	1	0	1	0
DEC R2	R2 = 0	1	0	0	0	R2 = –8	0	1	0	0

Valor decimal	Codificación en 4 bits y Ca2
+7	0111
+6	0110
+5	0101
+4	0100
+3	0011
+2	0010
+1	0001
0	0000
–1	1111
–2	1110
–3	1101
–4	1100
–5	1011

Valor decimal	Codificación en 4 bits y Ca2
–6	1010
–7	1001
–8	1000

Tened en cuenta que para obtener el número negado de un número en complemento a 2 hemos de negar el número bit a bit y sumarle 1. Por ejemplo:

- (+6) 0110, lo negamos, 1001 y le sumamos 1, 1010 (–6).
- (–3) 1101, lo negamos, 0010 y le sumamos 1, 0011 (+3).
- (+0) 0000, lo negamos, 1111 y le sumamos 1, 0000 (–0). Queda igual.
- (–8) 1000, lo negamos, 0111 y le sumamos 1, 1000 (–8). Queda igual, +8 no se puede representar en Ca2 utilizando 4 bits.

Una manera rápida de obtener el número negado manualmente es negar a partir del primer 1.

2.3.3. Instrucciones lógicas

Las instrucciones lógicas operan bit a bit y el resultado que se produce en un bit no afecta al resto. Activan los bits de resultado según el resultado obtenido. Estos bits de resultado los podremos consultar utilizando las instrucciones de salto condicional.

- **AND destino, fuente**. Hace la operación destino = destino AND fuente. Hace una 'y' lógica bit a bit.

- **OR destino, fuente**. Hace la operación destino = destino OR fuente. Hace una 'o' lógica bit a bit.

- **XOR destino, fuente**. Hace la operación destino = destino XOR fuente. Hace una 'o exclusiva' lógica bit a bit.

- **NOT destino**. Hace la negación lógica bit a bit del operando destino.

- **SAL destino, fuente**. Hace un desplazamiento aritmético a la izquierda de los bits destino, desplaza tantos bits como indique fuente y llena los bits de menos peso con 0. Se produce desbordamiento si el bit de más peso (bit 31) cambia de valor al finalizar los desplazamientos. Los bits que se desplazan se pierden.

- **SAR destino, fuente**. Hace un desplazamiento aritmético a la derecha de los bits de destino, desplaza tantos bits como indique fuente. Conserva el bit de signo de destino; es decir, copia el bit de signo a los bits de más peso. Los bits que se desplazan se pierden.

- **TEST destino, fuente**. Comparación lógica que realiza una operación lógica AND actualizando los bits de resultado que corresponda según el resultado generado pero sin guardar el resultado. Los operandos no se modifican y el resultado no se guarda.

Bits de resultado

Todas las instrucciones lógicas (excepto NOT), pueden modificar los bits de resultado del registro de estado según el resultado obtenido.

Instrucción	Z	S	C	V
AND	x	x	0	0
OR	x	x	0	0
XOR	x	x	0	0
NOT	-	-	-	-
SAL	x	x	-	x
SAR	x	x	-	0
TEST	x	x	0	0

Notación: x significa que la instrucción modifica el bit de resultado; - significa que la instrucción no modifica este bit; 0 indica que la instrucción pone este bit a 0.

Ejemplo

En este ejemplo se muestra cómo funcionan los bits de resultado de acuerdo con las especificaciones de la arquitectura CISCA, pero utilizando registros de 4 bits en lugar de registros de 32 bits para facilitar los cálculos. Consideremos los operandos R1 y R2 registros de 4 bits que representan valores numéricos en complemento a 2 (rango de valores desde –8 hasta +7). El valor inicial de todos los bits de resultado por cada instrucción es 0.

	Resultado	R1 = 0110, R2 = 0001				Resultado	R1 = 1111, R2 = 1001			
		Z	S	C	V		Z	S	C	V
AND R1, R2	R1 = 0000	1	0	0	0	R1 = 1001	0	1	0	0
OR R1, R2	R1 = 0111	0	0	0	0	R1 = 1111	0	1	0	0
XOR R1, R2	R1 = 0111	0	0	0	0	R1 = 0110	0	0	0	0
SAL R2,1	R2 = 0010	0	0	–	0	R2 = 0010	0	0	–	1
SAR R2,1	R2 = 0000	1	0	–	0	R2 = 1100	0	1	–	0

2.3.4. Instrucciones de ruptura de secuencia

Podemos distinguir entre:

1) Salto incondicional

- **JMP etiqueta**. *etiqueta* indica la dirección de memoria donde se quiere saltar. Esta dirección se carga en el registro PC. La instrucción que se ejecutará después de *JMP etiqueta* siempre es la instrucción indicada por *etiqueta* (JMP es una instrucción de ruptura de secuencia incondicional). El modo de direccionamiento que utilizamos en esta instrucción es el direccionamiento inmediato.

Etiqueta

Para especificar una etiqueta dentro de un programa en ensamblador lo haremos poniendo el nombre de la etiqueta seguido de ": ". Por ejemplo:

eti3: JMP eti3

2) Saltos condicionales. En las instrucciones de salto condicional (JE, JNE, JL, JLE, JG, JGE) se ejecutará la instrucción indicada por *etiqueta* si se cumple una condición; en caso contrario, continuará la secuencia prevista. El modo de direccionamiento que utilizamos en estas instrucciones es el direccionamiento relativo a PC.

- **JE etiqueta** (Jump Equal – Salta si igual). Si el bit Z está activo, carga en el PC la dirección indicada por *etiqueta*; en caso contrario, continúa la secuencia prevista.

- **JNE etiqueta** (Jump Not Equal – Salta si diferente). Si el bit Z está inactivo, carga en el PC la dirección indicada por *etiqueta*; en caso contrario, continúa la secuencia prevista.

- **JL etiqueta** (Jump Less – Salta si más pequeño). Si $S \neq V$, carga en el PC la dirección indicada por *etiqueta*; en caso contrario, continúa la secuencia prevista.

- **JLE etiqueta** (Jump Less or Equal – Salta si más pequeño o igual). Si $Z = 1$ o $S \neq V$, carga en el PC la dirección indicada por *etiqueta*; en caso contrario, continúa la secuencia prevista.

- **JG etiqueta** (Jump Greater – Salta si mayor). Si $Z = 0$ y $S = V$, carga en el PC la dirección indicada por *etiqueta*; en caso contrario, continúa la secuencia prevista.

- **JGE etiqueta** (Jump Greater or Equal – Salta si mayor o igual). Si $S = V$, carga en el PC la dirección indicada por *etiqueta*; en caso contrario, continúa la secuencia prevista.

3) Llamada y retorno de subrutina

- **CALL etiqueta** (llamada a la subrutina indicada por etiqueta). *etiqueta* es una dirección de memoria en la que empieza la subrutina. Primero se decrementa SP en 4 unidades, se almacena en la pila el valor PC_{up} y el registro PC se carga con la dirección expresada por la etiqueta. El modo de direccionamiento que utilizamos en estas instrucciones es el direccionamiento inmediato.

- **RET** (retorno de subrutina). Recupera de la pila el valor del PC e incrementa SP en 4 unidades.

4) Llamada al sistema y retorno de rutina de servicio de interrupción

- **INT servicio** (interrupción de software o llamada a un servicio del sistema operativo). *servicio* es un valor que identifica el servicio solicitado. El modo de direccionamiento que utilizamos en estas instrucciones es el direccionamiento inmediato.

- **IRET** (retorno de una rutina de servicio de interrupción). Recupera de la pila del sistema el valor del PC y el registro de estado; el registro SP queda incrementado en 8 unidades.

2.3.5. Instrucciones de entrada/salida

- **IN Ri, puerto**. Mueve el contenido del puerto de E/S especificado en registro Ri.

- **OUT puerto, Ri**. Mueve el contenido del registro Ri al puerto de E/S especificado.

Puerto

Puerto hace referencia a un puerto de entrada salida, a un registro de un módulo de E/S. Para acceder a un puerto podremos utilizar los mismos modos de direccionamiento que para acceder a memoria.

2.3.6. Instrucciones especiales

- **NOP**. No hace nada. El procesador pasa el tiempo de ejecutar una instrucción sin hacer nada.

- **STI**. Habilita las interrupciones, activa (pone a 1) el bit IE del registro de estado. Si las interrupciones no están habilitadas, el procesador no admitirá peticiones de interrupción.

- **CLI**. Inhibe las interrupciones, desactiva (pone a 0) el bit IE del registro de estado.

2.4. Estructuras de control

En este subapartado veremos cómo diferentes estructuras de control en lenguajes de alto nivel, expresadas en C, se pueden traducir a lenguaje ensamblador CISCA.

2.4.1. Estructura if

Estructura condicional expresada en C:

```
if (expresión) sentencia;
```

Ejemplo

```
if (x>y) maxx = 1;
```

Se puede traducir a lenguaje ensamblador CISCA de la manera siguiente:

```
if:   mov r1,[x]
      cmp r1,[y]
      jg set
      jmp endif
set:  mov [maxx],1
endif:
```

Se puede optimizar el número de instrucciones del código si se utiliza la condición contraria a la que aparece en el código C (`x<=y`)

```
if:   mov r1,[x]
      cmp r1, [y]
      jle endif    ;condición contraria
      mov [maxx],1
endif:
```

2.4.2. Estructura if-else

Estructura condicional expresada en C, incluyendo la condición alternativa:

```
if (expresión) sentencia1;
   else sentencia2;
```

Ejemplo

```
if (x>y) max = x;
   else max = y;
```

Se puede traducir a lenguaje ensamblador CISCA de la manera siguiente:

```
if:   mov r1,[x]
      mov r2,[y]
      cmp r1,r2
      jg true
      jmp else
true: mov [max],r1
      jmp endif
else: mov [max],r2
endif:
```

Se puede optimizar el número de instrucciones del código si se utiliza la condición contraria a la que aparece en el código C (x<=y)

```
if:   mov r1,[x]
      mov r2,[y]
      cmp r1,r2
      jle else ;condición contraria
      mov [max],r1
      jmp endif
else: mov [max],r2
endif:
```

2.4.3. Estructura while

Estructura iterativa en C controlada por una condición expresada al principio:

```
while (expresión)
   sentencia;
```

con más de una sentencia:

```
while (expresión) {
   sentencia1;
   sentencia2;
}
```

Ejemplo

```
while(num > 0){
   i = i*num;
   num = num - 1;
}
resul = i;
```

Se puede traducir a lenguaje ensamblador CISCA de la manera siguiente:

```
       mov r1,[i]
       mov r2,[num]
while: cmp r2,0
       jg cont
       jmp end_w
cont:  mul r1,r2
       sub r2,1
       jmp while
end_w: mov [resul],r1
       mov [i],r1
```

Se puede optimizar el número de instrucciones del código si se utiliza la condición contraria a la que aparece en el código C (`num<=0`)

```
       mov r1,[i]
       mov r2,[num]
while: cmp r2,0
       jle end_w    ;condición contraria
       mul r1,r2
       sub r2,1
       jmp while
end_w: mov [resul],r1
       mov [i],r1
```

2.4.4. Estructura do-while

Estructura iterativa en C controlada por una condición expresada al final:

```
do
   sentencia;
while (expresión);
```

con más de una sentencia:

```
do {
   sentencia1;
   sentencia2;
} while (expresión);
```

Ejemplo

```
do {
   sum = sum + i;
   i = i + 1
} while (i <= valor);
```

Se puede traducir a lenguaje ensamblador CISCA de la manera siguiente:

```
    mov r1,[i]
do: add [sum],r1
    add r1,1
    cmp r1,[valor]
    jle do
    mov [i],r1
```

2.4.5. Estructura for

Estructura iterativa utilizando la orden for:

```
for (expr1; expr2; expr3)
```

```
      sentencia;
```

es equivalente a:

```
expr1;
while (expr2) {
   sentencia;
   expr3;
}
```

donde:

```
expr1: inicialización de un contador
expr2: condición de salida de la estructura iterativa
expr3: actualización del contador
```

Ejemplo

```
for (i=0; i<=valor; i++)
   sum = sum + i;
```

Se puede traducir a lenguaje ensamblador CISCA de la manera siguiente:

```
      mov r1,0
for:  cmp r1,[valor]
      jle incr
      jmp endf
incr: add [sum],r1
      add r1,1    ; equivalente a inc r1
      jmp for
endf: mov [i],r1
```

Se puede optimizar el número de instrucciones del código si se utiliza la condición contraria a la que aparece en el código C (`i>valor`)

```
      mov r1,0
for:  cmp r1,[valor]
      jg endf     ; condición contraria
      add [sum],r1
      add r1,1    ; equivalente a inc r1
      jmp for
endf: mov [i],r1
```

2.4.6. Estructura switch-case

Estructura de selección expresada en C, que amplía el número de opciones de la estructura if-else, permitiendo múltiples opciones. Se utiliza una variable que se compara por igualdad sucesivamente con diferentes valores constantes,

se permite indicar una o varias sentencias por cada caso, finalizadas por la sentencia *break*, y especificar una o varias sentencias que se ejecuten en caso de no coincidir con ninguno de los valores indicados, *default*.

```
switch (variable) {
  case valor1:   sentencia1;
         break;
  case valor2:   sentencia2;
         ...
         break;
  ...
  default;      sentenciaN;
}
```

Ejemplo

```
switch (var) {
  case 1:   a=a+b;
      break;
  case 2:   a=a-b;
      break;
  case 3:   a=a*b;
      break;
  default:   a=-a;
}
```

Se puede traducir a lenguaje ensamblador CISCA de la siguiente manera:

```
switch:  mov r1,[var]
         cmp r1, 1
         jne case2
         mov r2, [b]
         add [a], r2
         jmp end_s
case2:   cmp r1, 2
         jne case3
         mov r2, [b]
         sub [a], r2
         jmp end_s
case3:   cmp r1, 3
         jne default
         mov r2, [b]
         mul [a], r2
         jmp end_s
default: neg [a]
end_s:
```

3. Formato y codificación de las instrucciones

Las instrucciones de esta arquitectura tienen un formato de longitud variable, como suele suceder en todas las arquitecturas CISC. Tenemos instrucciones de 1, 2, 3, 4, 5, 6, 7, 9 o 11 bytes, según el tipo de instrucción y los modos de direccionamiento utilizados.

> Denotaremos cada uno de los bytes que forman una instrucción como B0, B1, ... (Byte 0, Byte 1, ...). Si la instrucción está almacenada a partir de la dirección @ de memoria, B0 es el byte que se encuentra en la dirección @; B1, en la dirección @ + 1; etc. Por otro lado, los bits dentro de un byte se numeran del 7 al 0, siendo el 0 el de menor peso. Denotamos B*k*<*j*..*i*> con *j* > *i* el campo del byte *k* formado por los bits del *j* al *i*. Escribiremos los valores que toma cada uno de los bytes en hexadecimal.

Para codificar una instrucción, primero codificaremos el código de operación en el byte B0 y, a continuación, en los bytes siguientes (los que sean necesarios), los operandos de la instrucción con su modo de direccionamiento.

En las instrucciones de 2 operandos se codificará un operando a continuación del otro, en el mismo orden que se especifica en la instrucción. Hay que recordar que al ser una arquitectura registro-memoria solo un operando puede hacer referencia a memoria. De este modo tendremos las combinaciones de modos de direccionamiento que se muestran en la siguiente tabla.

Observación

En las instrucciones de dos operandos, el operando destino no podrá utilizar un direccionamiento inmediato.

Operando destino	Operando fuente
Registro (direccionamiento directo en registro)	Inmediato
	Registro (direccionamiento directo a registro)
	Memoria (direccionamiento directo a memoria)
	Indirecto (direccionamiento indirecto a registro)
	Relativo (direccionamiento relativo a registro base)
	Indexado (direccionamiento relativo a registro índice)
Memoria (direccionamiento directo a memoria)	Inmediato
	Registro (direccionamiento directo a registro)

Operando destino	Operando fuente
Indirecto (direccionamiento indirecto a registro)	Inmediato
	Registro (direccionamiento directo a registro)
Relativo (direccionamiento relativo a registro base)	Inmediato
	Registro (direccionamiento directo a registro)
Indexado (direccionamiento relativo a registro índice)	Inmediato
	Registro (direccionamiento directo a registro)

3.1. Codificación del código de operación. Byte B0

El byte B0 representa el código de operación de las instrucciones. Esta arquitectura no utiliza la técnica de expansión de código para el código de operación.

Codificación del byte B0

B0 (Código de operación)	Instrucción
Especiales	
00h	NOP
01h	STI
02h	CLI
Transferencia	
10h	MOV
11h	PUSH
12h	POP
Aritméticas	
20h	ADD
21h	SUB
22h	MUL
23h	DIV
24h	INC
25h	DEC
26h	CMP
27h	NEG
Lógicas	
30h	AND

B0 (Código de operación)	Instrucción
31h	OR
32h	XOR
33h	TEST
34h	NOT
35h	SAL
36h	SAR
Ruptura de secuencia	
40h	JMP
41h	JE
42h	JNE
43h	JL
44h	JLE
45h	JG
46h	JGE
47h	CALL
48h	RET
49h	INT
4Ah	IRET
Entrada/Salida	
50h	IN
51h	OUT

3.2. Codificación de los operandos. Bytes B1-B10

Para codificar los operandos deberemos especificar el modo de direccionamiento, si este no está implícito; así como la información para expresar directamente un dato, la dirección, o la referencia a la dirección en la que tenemos el dato.

Para codificar un operando podemos necesitar 1, 3 o 5 bytes, que denotaremos como Bk, Bk+1, Bk+2, Bk+3, Bk+4.

En el primer byte (Bk), codificaremos el modo de direccionamiento (Bk<7..4>) y el número de registro si este modo de direccionamiento utiliza un registro, o ceros si no utiliza un registro (Bk<3..0>).

Bk<7..4>	Modo de direccionamiento
0h	Inmediato
1h	Registro (direccionamiento directo a registro)
2h	Memoria (direccionamiento directo a memoria)
3h	Indirecto (direccionamiento indirecto a registro)
4h	Relativo (direccionamiento relativo a registro base)
5h	Indexado (direccionamiento relativo a registro índice)
6h	EN PC (direccionamiento relativo a registro PC)
De 7h a Fh	Códigos no utilizados en la implementación actual. Quedan libres para futuras ampliaciones del lenguaje.

Bk<7..4>: modo de direccionamiento
Bk<3..0>: 0 (no utiliza registros)
Bk+1, Bk+2, Bk+3, Bk+4: valor inmediato de 32 bits en Ca2

Cuando debamos codificar un valor inmediato o una dirección en la que tenemos el dato, lo codificaremos utilizando los bytes (Bk+1, Bk+2, Bk3, Bk+4) en formato Little-Endian; si tenemos que codificar un desplazamiento, utilizaremos los bytes (Bk+1, Bk+2) en formato Little-Endian.

Expresiones aritméticas

Si se ha expresado un operando utilizando una expresión aritmética, antes de codificarlo se deberá evaluar la expresión y representarla en el formato correspondiente: una dirección utilizando 32 bits, un valor inmediato utilizando 32 bits en Ca2 y un desplazamiento utilizando 16 bits en Ca2.

Recordad que en las instrucciones con dos operandos explícitos solo uno de los dos operandos puede hacer referencia a la memoria; el otro será un registro o un valor inmediato. Hay que codificar cada operando según el modo de direccionamiento que utilice:

1) Inmediato

Formato: número decimal, binario o hexadecimal o etiqueta.

El dato se codifica en Ca2 utilizando 32 bits en formato Little-Endian. No se hace extensión de signo, se completa el número con ceros; por este motivo, en binario y hexadecimal los números negativos se deben expresar en Ca2 utilizando 32 bits (32 dígitos binarios u 8 dígitos hexadecimales, respectivamente).

En las instrucciones de transferencia en las que se especifica una etiqueta sin corchetes que representa el nombre de una variable o el punto del código al que queremos ir, se codifica una dirección de memoria de 32 bits (dirección de la variable o dirección a la que queremos ir, respectivamente). Dado que no debemos hacer ningún acceso a memoria para obtener el operando, consideraremos que estas instrucciones utilizan un direccionamiento inmediato.

Ejemplo

Sintaxis	Valor que codificar	Codificación operando					
		Bk<7..4>	Bk<3..0>	Bk+1	Bk+2	Bk+3	Bk+4
0	00000000h	0h	0h	00h	00h	00h	00h
100	00000064h	0h	0h	64h	00h	00h	00h
–100	FFFFFF9Ch	0h	0h	9Ch	FFh	FFh	FFh
156	0000009Ch	0h	0h	9Ch	00h	00h	00h
9Ch	0000009Ch	0h	0h	9Ch	00h	00h	00h
FF9Ch	0000FF9Ch	0h	0h	9Ch	FFh	00h	00h
FFFFFF9Ch	FFFFFF9Ch	0h	0h	9Ch	FFh	FFh	FFh
1001 1100b	0000009Ch	0h	0h	9Ch	00h	00h	00h
1111 1111 1111 1111 1111 1111 1001 1100b	FFFFFF9Ch	0h	0h	9Ch	FFh	FFh	FFh
var1 La etiqueta *var1* vale 00AB01E0h	00AB01E0h	0h	0h	E0h	01h	ABh	00h
bucle La etiqueta *bucle* vale 1FF00230h	1FF00230h	0h	0h	30h	02h	F0h	1Fh

Bk<7..4>: modo de direccionamiento
Bk<3..0>: 0 (no utiliza registros)
Bk+1, Bk+2, Bk+3, Bk+4: valor inmediato de 32 bits en Ca2

2) Registro (direccionamiento directo a registro)

Formato: Ri

El registro se codifica utilizando 4 bits.

Ejemplo

Sintaxis	Valor que codificar	Codificación operando	
		Bk<7..4>	Bk<3..0>
R0	0h	1h	0h
R10	Ah	1h	Ah

Bk<7..4>: modo de direccionamiento
Bk<3..0>: número de registro

3) Memoria (direccionamiento directo a memoria)

Formato:[dirección] o [nombre_variable]

La dirección de memoria se codifica utilizando 32 bits en formato Little-Endian (8 dígitos hexadecimales), necesarios para acceder a los 4Gbytes de la memoria principal. Si ponemos nombre_variable, para poder hacer la codificación hay que conocer a qué dirección de memoria hace referencia.

Ejemplo

Sintaxis	Valor que codificar	Codificación operando					
		Bk<7..4>	Bk<3..0>	Bk+1	Bk+2	Bk+3	Bk+4
[00AB01E0h]	00AB01E0h	2h	0h	E0h	01h	ABh	00h
[var1] La etiqueta *var1* vale 00AB01E0h	00AB01E0h	2h	0h	E0h	01h	ABh	00h

Bk<7..4>: modo de direccionamiento
Bk<3..0>: 0 (no utiliza registros)
Bk+1, Bk+2, Bk+3, Bk+4: dirección de 32 bits

4) Indirecto (direccionamiento indirecto a registro)

Formato: [Ri]

El registro se codifica utilizando 4 bits.

Ejemplo

Sintaxis	Valor que codificar	Codificación operando	
		Bk<7..4>	Bk<3..0>
[R0]	0h	3h	0h
[R10]	Ah	3h	Ah

Bk<7..4>: modo de direccionamiento
Bk<3..0>: número de registro

5) Relativo (direccionamiento relativo a registro base)

Formato: [Registro + Desplazamiento]

El registro se codifica utilizando 4 bits. El desplazamiento se codifica en Ca2 utilizando 16 bits en formato Little-Endian. No se hace extensión de signo, se completa el número con ceros. Por este motivo, en hexadecimal los números negativos se deben expresar en Ca2 utilizando 16 bits (4 dígitos hexadecimales).

Ejemplo

Sintaxis	Valores que codificar	Codificación operando			
		Bk<7..4>	Bk<3..0>	Bk+1	Bk+2
[R0+8]	0h y 0008h	4h	0h	08h	00h
[R10+001Bh]	Ah y 001Bh	4h	Ah	1Bh	00h
[R11-4]	Bh y FFFCh	4h	Bh	FCh	FFh
[R3+FFFCh]	3h y FFFCh	4h	3h	FCh	FFh

Bk<7..4>: modo de direccionamiento
Bk<3..0>: número de registro
Bk+1, Bk+2: desplazamiento de 16 bits en Ca2

6) Indexado (direccionamiento relativo a registro índice)

Formato: [dirección + registro] [nombre_variable + registro] o [(expresión) + Registro]

El registro se codifica utilizando 4 bits. La dirección de memoria se codifica utilizando 32 bits en formato Little-Endian (8 dígitos hexadecimales), necesarios para acceder a los 4Gbytes de la memoria principal. Si ponemos nombre_variable, para poder hacer la codificación hay que conocer a qué dirección de memoria hace referencia.

Ejemplo

Sintaxis	Valores que codificar	Codificación operando					
		Bk<7..4>	Bk<3..0>	Bk+1	Bk+2	Bk+3	Bk+4
[0ABC0100h+R2]	0ABC0100h y 2h	5h	2h	00h	01h	BCh	0Ah
[vector1+R9] La etiqueta *vector1* vale 0ABC0100h	0ABC0100h y 9h	5h	9h	00h	01h	BCh	0Ah

Bk<7..4>: modo de direccionamiento
Bk<3..0>: número de registro
Bk+1, Bk+2, Bk+3, Bk+4: dirección de 32 bits

7) A PC (direccionamiento relativo a registro PC)

Formato: etiqueta o (expresión).

En este modo de direccionamiento no se codifica la etiqueta, se codifica el número de bytes que hay que desplazar para llegar a la posición de memoria indicada por la etiqueta; este desplazamiento se codifica en Ca2 utilizando 16 bits en formato Little-Endian (4 dígitos hexadecimales).

Para determinar el desplazamiento que debemos codificar (*desp16*), es necesario conocer a qué dirección de memoria hace referencia la etiqueta especificada en la instrucción (*etiqueta*) y la dirección de memoria de la siguiente instrucción (dirección del byte B0 de la siguiente instrucción, PC_{up}).

$$\text{desp16} = \textit{etiqueta} - PC_{up}$$

Si *etiqueta* < PC_{up}, entonces *desp16* será negativo; por lo tanto, daremos un salto hacia atrás en el código.

Si *etiqueta* ≥ PC_{up}, entonces *desp16* será positivo; por lo tanto, daremos un salto hacia delante en el código.

Ejemplo

Sintaxis	Valor que codificar	Codificación operando			
		Bk<7..4>	Bk<3..0>	Bk+1	Bk+2
Inicio La dirección de la etiqueta *Inicio* vale 0AB00030h y PC_{up} = 0AB00150h	FEE0h	6h	0h	E0h	FEh
Fin La dirección de la etiqueta *Fin* vale 0AB00200h y PC_{up} = 0AB00150h	00B0h	6h	0h	B0h	00h

Bk<7..4>: modo de direccionamiento
Bk<3..0>: 0 (no utiliza registros)
Bk+1, Bk+2: desplazamiento de 16 bits en Ca2

3.3. Ejemplos de codificación

Vemos a continuación cómo codificamos algunas instrucciones.

1) PUSH R3

Código de operación: PUSH

- El campo B0 = 11h

Operando: R3 (direccionamiento a registro)

- El campo Bk: (Bk<7..4>) modo de direccionamiento = 1h y (Bk<3..0>) registro = 3h

La codificación en hexadecimal de esta instrucción será:

B0	B1
11h	13h

2) JNE etiqueta

Código de operación: JNE

- El campo B0 = 42h

Operando: etiqueta (direccionamiento relativo a PC)

Suponemos que etiqueta tiene el valor 1FF000B4h y PC_{up} vale 1FF00030h.

- El campo Bk: (Bk<7..4>) modo de direccionamiento = 6h y (Bk<3..0>) sin registro = 0h

- El campo Bk+1, Bk+2: *desp16* = *etiqueta* – PC_{up}.

desp16 = 1FF000B4h – 1FF00030h = 0084h

1FF000B4h > 1FF00030h → desp16 será positivo → salto adelante

La codificación en hexadecimal de esta instrucción será:

B0	B1	B2	B3
42h	60h	84h	00h

3) JL etiqueta

Código de operación: JL

- El campo B0 = 43h

Operando: etiqueta (direccionamiento relativo a PC)

Suponemos que etiqueta tiene el valor 1FF000A0h y PC_{up} vale 1FF00110h.

- El campo Bk: (Bk<7..4>) modo de direccionamiento = 6h y (Bk<3..0>) sin registro = 0h

- El campo Bk+1, Bk+2: *desp16* = *etiqueta* – PC_{up}.

desp16 = 1FF000A0h – 1FF00110h = FF90h

1FF000A0h<1FF00110h → *desp16* es negativo = FF90h (–112 decimal); por lo tanto, daremos un salto hacia atrás en el código.

La codificación en hexadecimal de esta instrucción será:

B0	B1	B2	B3
43h	60h	90h	FFh

4) NOT [0012005Bh]

Código de operación: NOT

- El campo B0 = 34h

Operando: [0012005Bh] (direccionamiento directo a memoria).

- El campo Bk: (Bk<7..4>) modo de direccionamiento = 2h y (Bk<3..0>) sin registro = 0h.

- El campo Bk+1, Bk+2, Bk+3, Bk+4: dirección de memoria codificada con 32 bits en formato Little-Endian.

La codificación en hexadecimal de esta instrucción será:

B0	B1	B2	B3	B4	B5
34h	20h	5Bh	00h	12h	00h

En la tabla siguiente se muestra la codificación de algunas instrucciones. Suponemos que todas las instrucciones empiezan en la dirección @=1FF000B0h (no se debe entender como programa, sino como instrucciones individuales). En los ejemplos en los que sea necesario *etiqueta1* = 0012005Bh y *etiqueta2* = 1FF00030h. El valor de cada uno de los bytes de la instrucción con direcciones @ + *i* para *i* = 0, 1, ... se muestra en la tabla en hexadecimal (recordad que los campos que codifican un desplazamiento en 2 bytes, un valor inmediato o una dirección en 4 bytes lo hacen en formato Little-Endian; hay que tener esto en cuenta y escribir los bytes de dirección más pequeña a la izquierda y los de dirección mayor a la derecha):

	Bk para k=0..10										
Ensamblador	**B0**	**B1**	**B2**	**B3**	**B4**	**B5**	**B6**	**B7**	**B8**	**B9**	**B10**
PUSH R3	11	1D									
JMP etiqueta2	40	00	30	00	F0	IF					
JNE etiqueta2	42	60	7Ch	FFh							
CALL etiqueta2	47	00	30	00	F0	1F					
NOT [etiqueta1]	34	20	5B	00	12	00					
DEC [R4]	25	34									

	Bk para k=0..10										
Ensamblador	**B0**	**B1**	**B2**	**B3**	**B4**	**B5**	**B6**	**B7**	**B8**	**B9**	**B10**
XOR [R13 + 3F4Ah], R12	32	4D	4A	3F	1C						
ADD [8A5165F7h + R1], -4	20	51	F7	65	51	8A	00	FC	FF	FF	FF
RET	48										
MOV [R4+32], 100	10	44	20	00	00	64	00	00	00		

En el byte B0 se codifica el código de operación y en las casillas sombreadas está codificado el modo de direccionamiento (Bk<7..4>) y el número de registro si este modo de direccionamiento utiliza un registro, o cero si no utiliza ningún registro (Bk<3..0>). Por lo tanto, nos indica dónde empieza la codificación de cada operando de la instrucción.

En la instrucción JMP etiqueta2, que es una instrucción de salto incondicional y que utiliza direccionamiento inmediato, se codifica la dirección donde se quiere saltar utilizando 4 bytes, en este caso etiqueta2 = 1FF00030h.

En cambio, en la instrucción JNE etiqueta2, que es una instrucción de salto condicional y que utiliza direccionamiento relativo a PC, se codifica un desplazamiento utilizando 2 bytes, para obtener el desplazamiento se resta PCup – etiqueta2.

En este caso, PCup = dirección de la instrucción + tamaño en bytes de la instrucción codificada = 1FF000B0h + 4 = 1FF000B4h, y etiqueta2 = 1FF00030h. De esta forma PCup – etiqueta2 = 1FF000B4h – 1FF00030h = FFFFFF7Ch, pero como solo se consideran 2 bytes, el desplazamiento será FF7Ch.

4. Ejecución de las instrucciones

La ejecución de una instrucción consiste en realizar lo que denominamos un *ciclo de ejecución*, y este ciclo de ejecución es una secuencia de operaciones que se dividen en 4 fases principales:

1) Lectura de la instrucción.
2) Lectura de los operandos fuente.
3) Ejecución de la instrucción y almacenamiento del operando destino.
4) Comprobación de interrupciones.

Las operaciones que realiza el procesador en cada fase están gobernadas por la unidad de control y se denominan *microoperaciones*. Para determinar esta secuencia de microoperaciones, la unidad de control deberá descodificar la instrucción, es decir, leer e interpretar la información que tendremos en el registro IR.

La nomenclatura que utilizaremos para denotar las microoperaciones será la siguiente:

Registro destino ← Registro origen

Registro destino ← Registro origen <operación> Registro origen / Valor

Vamos a analizar la secuencia de microoperaciones que habitualmente se produce en cada fase del ciclo de ejecución de las instrucciones.

4.1. Lectura de la instrucción

Leemos la instrucción que queremos ejecutar. Esta fase consta básicamente de 4 pasos:

```
MAR ← PC, read  ; Ponemos el contenido de PC en el registro MAR.
MBR ← Memoria   ; Leemos la instrucción.
PC ← PC + Δ     ; Incrementamos el PC en Δ unidades (Δ = tamaño de la instrucción en bytes)
IR ← MBR        ; Cargamos la instrucción en el registro IR.
```

Dado que en esta arquitectura se accede a la memoria en palabras de 4 bytes, si la instrucción tiene un tamaño superior a una palabra de memoria (4 bytes), se tendría que repetir el proceso de lectura de memoria tantas veces como fuera necesario para leer toda la instrucción. Para simplificar el funcionamiento de

la unidad de control se considerará que se puede leer toda la instrucción haciendo un único acceso a memoria, y por lo tanto se especificará el incremento del PC (Δ) según el tamaño en bytes de la instrucción.

La información almacenada en el registro IR se descodificará para identificar las diferentes partes de la instrucción y determinar, así, las operaciones necesarias que habrá que realizar en las fases siguientes.

4.2. Lectura de los operandos fuente

Leemos los operandos fuente de la instrucción. El número de pasos que habrá que realizar en esta fase dependerá del número de operandos fuente y de los modos de direccionamiento utilizados en cada operando.

El modo de direccionamiento indicará el lugar en el que está el dato, ya sea una dirección de memoria o un registro. Si está en memoria habrá que llevar el dato al registro MBR; y si está en un registro, no habrá que hacer nada porque ya lo tendremos disponible en el procesador. Como esta arquitectura tiene un modelo registro-memoria, solo uno de los operandos podrá hacer referencia a memoria.

Vamos a ver cómo se resolvería para diferentes modos de direccionamiento:

1) Inmediato: tenemos el dato en la propia instrucción. No será necesario hacer nada.

2) Directo a registro: tenemos el dato en un registro. No hay que hacer nada.

3) Directo a memoria:

```
MAR ← IR(Dirección), read
MBR ← Memoria
```

4) Indirecto a registro:

```
MAR ← Contenido de IR(Registro), read
MBR ← Memoria
```

5) Relativo a registro índice:

```
MAR ← IR(Dirección operando) + Contenido de IR(Registro índice), read
MBR ← Memoria
```

6) Relativo a registro base:

```
MAR ← Contenido de IR(Registro base) + IR(Desplazamiento), read
MBR ← Memoria
```

7) Relativo a PC:

IR(campo)

En IR(campo) consideraremos que *campo* es uno de los operandos de la instrucción que acabamos de leer y que tenemos guardado en el registro IR.

Read E/S

En la instrucción IN, el operando fuente hace referencia a un puerto de E/S; en este caso, en lugar de activar la señal *read* habrá que activar la señal *read E/S*.

No hay que hacer ninguna operación para obtener el operando, solo en el caso de que se tenga que dar el salto haremos el cálculo. Pero se hará en la fase de ejecución y almacenamiento del operando destino.

8) A pila:

Se utiliza de manera implícita el registro SP. Se asemeja al modo indirecto a registro, pero el acceso a la pila se resolverá en esta fase cuando hacemos un POP (leemos datos de la pila), y se resuelve en la fase de ejecución y almacenamiento del operando destino cuando hacemos un PUSH (guardamos datos en la pila).

```
MAR ← SP, read
MBR ← Memoria
SP  ← SP + 4   ; 4 es el tamaño de la palabra de pila
               ; '+' sumamos porque crece hacia direcciones bajas
```

4.3. Ejecución de la instrucción y almacenamiento del operando destino

Cuando iniciamos la fase de ejecución y almacenamiento del operando destino, tendremos los operandos fuente en registros del procesador Ri o en el registro MBR si hemos leído el operando de memoria.

Las operaciones que deberemos realizar dependerán de la información del código de operación de la instrucción y del modo de direccionamiento utilizado para especificar el operando destino.

Una vez hecha la operación especificada, para almacenar el operando destino se pueden dar los casos siguientes:

a) Si el operando destino es un registro, al hacer la operación ya dejaremos el resultado en el registro especificado.

b) Si el operando destino hace referencia a memoria, al hacer la operación dejaremos el resultado en el registro MBR, y después lo almacenaremos en la memoria en la dirección especificada por el registro MAR:

- Si el operando destino ya se ha utilizado como operando fuente (como sucede en las instrucciones aritméticas y lógicas), todavía tendremos la dirección en el MAR.

- Si el operando destino no se ha utilizado como operando fuente (como sucede en las instrucciones de transferencia y de entrada salida), primero habrá que *resolver el modo de direccionamiento* como se ha explicado anteriormente en la fase de lectura del operando fuente, dejando la dirección del operando destino en el registro MAR.

4.3.1. Operaciones de transferencia

1) MOV destino, fuente

- Si el operando destino es un registro y el operando fuente es un inmediato:

```
Ri ← IR(valor Inmediato)
```

- Si el operando destino es un registro y el operando fuente también es un registro:

```
Ri ← Rj
```

- Si el operando destino es un registro y el operando fuente hace referencia a memoria:

```
Ri ← MBR
```

- Si el operando destino hace referencia a memoria y el operando fuente es un inmediato:

```
MBR ← IR(valor Inmediato)
(Resolver modo de direccionamiento),write
Memoria ← MBR
```

- Si el operando destino hace referencia a memoria y el operando fuente es un registro:

```
MBR ← Ri,
(Resolver modo de direccionamiento),write
Memoria ← MBR
```

2) PUSH fuente

- Si el operando fuente es un registro, primero habrá que llevarlo al registro MBR:

```
MBR ← Ri
SP  ← SP - 4   ; 4 es el tamaño de la palabra de pila
               ; '-' restamos porque crece hacia direcciones bajas
MAR ← SP, write
Memoria ← MBR
```

- Si el operando fuente hace referencia a memoria, ya estará en el registro MBR:

```
SP  ← SP - 4    ; 4 es el tamaño de la palabra de pila
                ; '-' restamos porque crece hacia direcciones bajas
MAR ← SP, write
Memoria ← MBR
```

3) POP destino

- Si el operando destino es un registro:

```
Ri ← MBR
```

- Si el operando destino hace referencia a memoria:

```
(Resolver modo de direccionamiento), write
Memoria ← MBR
```

4.3.2. Operaciones aritméticas y lógicas

Operando destino

Recordad que el operando destino también es operando fuente.

1) ADD destino, fuente; SUB destino, fuente; MUL destino, fuente; DIV destino, fuente; CMP destino, fuente; TEST destino, fuente; SAL destino, fuente; SAR destino, fuente:

- Si el operando destino es un registro y el operando fuente es un inmediato:

```
Ri ← Ri<operación>IR(valor Inmediato)
```

- Si el operando destino es un registro y el operando fuente es también un registro:

```
Ri ← Ri<operación>Ri
```

- Si el operando destino es un registro y el operando fuente hace referencia a memoria:

```
Ri ← Ri<operación>MBR
```

- Si el operando destino hace referencia a memoria y el operando fuente es uno inmediato:

```
MBR ← MBR<operación>IR(valor Inmediato), write
Memoria ← MBR
```

- Si el operando destino hace referencia a memoria y el operando fuente es un registro:

```
MBR ← MBR<operación>Ri, write
Memoria ← MBR
```

Para todas las combinaciones de modos de direccionamiento hace falta un tratamiento especial para la instrucción ***DIV destino, fuente***, puesto que genera dos resultados: el cociente y el resto.

Las microoperaciones que hay especificadas guardan el resultado de la operación en el operando ***destino***, para la instrucción DIV, este valor es el cociente de la división.

Hay que añadir para cada una de las combinaciones de modos de direccionamiento la microoperación siguiente que realiza la operación *módulo*, que genera el resto de la división y lo guarda en el registro R0:

R0 ← destino <mod> fuente.

Ejemplo

Si estamos ejecutando la instrucción ADD R3, 7, la microoperación que se ejecutaría en esta fase sería:

R3 ← R3 + IR(valor inmediato)=7

Si estamos ejecutando la instrucción DIV R1,[var], las microoperaciones que se ejecutarían en esta fase serían:

R1 ← R1 / MBR
R0 ← R1 <mod> MBR

2) INC destino; DEC destino; NEG destino; NOT destino

- Si el operando destino es un registro:

```
Ri ← Ri <operación>
```

- Si el operando destino hace referencia a memoria (*ya tendremos en el MAR la dirección*):

```
MBR ← MBR<operación>, write
Memoria ← MBR
```

4.3.3. Operaciones de ruptura de secuencia

1) JMP etiqueta

```
PC ← IR(Dirección)
```

2) JE etiqueta, JNE etiqueta, JL etiqueta, JLE etiqueta, JG etiqueta, JGE etiqueta:

```
PC ← PC + IR(Desplazamiento)
```

3) CALL etiqueta:

```
MBR ← PC
SP  ← SP - 4     ; 4 es el tamaño de la palabra de pila
                 ; '-' restamos porque crece hacia direcciones bajas
MAR ← SP, write
Memoria ← MBR    ; guardamos en la pila
PC  ← IR(Dirección)
```

4) RET

```
MAR ← SP, read
MBR ← Memoria
SP ← SP + 4   ; 4 es el tamaño de la palabra de pila.
              ; '+', sumamos porque crece hacia direcciones bajas.
PC ← MBR      ; Restauramos el PC
```

5) INT servicio

```
MBR ← Registro de Estado
SP ← SP - 4
MAR ← SP, write
Memoria ← MBR                ;Guardamos el registro de estado en la pila
MBR ← PC
SP ← SP - 4
MAR ← SP, write
Memoria ← MBR                ;Guardamos el PC en la pila
MAR ← IR(servicio)*4 , read  ;4 es el tamaño de cada dirección de la Tabla de Vectores
MBR ← Memoria                ;Leemos la dirección de la RSI de la tabla de vectores
                             ;La tabla de vectores comienza en la dirección 0 de memoria
PC ← MBR                     ;Cargamos en el PC la dirección de la Rutina de Servicio
```

6) IRET

```
MAR ← SP, read
MBR ← Memoria
SP ← SP + 4                ; 4 es el tamaño de la palabra de pila.
                           ; '+', sumamos porque crece hacia direcciones bajas.
PC ← MBR                   ; Restauramos el PC.
MAR ← SP, read
MBR ← Memoria
SP ← SP + 4
Registro de estado ← MBR ; Restauramos el registro de estado.
```

4.3.4. Operaciones de Entrada/Salida

1) IN Ri, puerto

```
Ri ← MBR        ;En el registro MBR tendremos el dato leído del puerto
                ;en el ciclo de lectura del operando fuente
```

2) OUT puerto, Ri

```
MBR ← Ri        ;Ponemos el dato que tenemos en el registro especificado
(Resolver modo de direccionamiento), write E/S
Memoria ← MBR   ;para almacenar como operando destino
```

4.3.5. Operaciones especiales

1) NOP. No hay que hacer nada (hace un ciclo de no operación de la ALU).

2) STI. Activa el bit IE del registro de estado para habilitar las interrupciones.

3) CLI. Desactiva el bit IE del registro de estado para inhibir las interrupciones.

4.4. Comprobación de interrupciones

En esta fase se comprueba si se ha producido una interrupción (para ello, no hay que ejecutar ninguna microoperación); si no se ha producido ninguna interrupción, se continúa con la siguiente instrucción, y si se ha producido alguna, se debe hacer el cambio de contexto, donde hay que guardar cierta información y poner en el PC la dirección de la rutina que da servicio a esta interrupción. Este proceso puede variar mucho de una máquina a otra. Aquí solo presentamos la secuencia de microoperaciones para actualizar el PC cuando se produce el cambio de contexto.

Bit de interrupción (IF)

Bit IF activo (IF=1): se ha producido una interrupción
Bit IF inactivo (IF=0): no se ha producido una interrupción

```
MBR ← Registro de Estado
SP  ← SP - 4
MAR ← SP, write
Memoria ← MBR                 ;Guardamos el registro de estado en la pila
MBR ← PC
SP  ← SP - 4
MAR ← SP, write
Memoria ← MBR                 ;Guardamos el PC en la pila
MAR ← (Índice RSI)*4,read     ;4 es el tamaño de cada dirección de la Tabla de Vectores
MBR ← Memoria                 ;Leemos la dirección de la RSI de la tabla de vectores
                              ;La tabla de vectores empieza en la dirección 0 de memoria
PC ← MBR                      ;Cargamos en el PC la dirección de la Rutina de servicio
```

4.5. Ejemplos de secuencias de microoperaciones

Presentamos a continuación varios ejemplos de secuencias de microoperaciones para algunos casos concretos. En cada caso, se indican las microoperaciones que hay que ejecutar en cada fase y en qué orden.

En cada secuencia, las fases se identifican de la manera siguiente:

- Fase 1: Lectura de la instrucción.
- Fase 2: Lectura de los operandos fuente.
- Fase 3: Ejecución de la instrucción y almacenamiento del operando destino.

1) MOV [R1+10], R3 ; direccionamiento relativo y a registro

Fase	Microoperación
1	MAR ← PC, read MBR ← Memoria PC ← PC + 5 IR ← MBR
2	(no hay que hacer nada, el operando fuente está en un registro)
3	MBR ← R3 MAR ← R1 + 10, write Memoria ← MBR

2) PUSH R4 ; direccionamiento a pila

Fase	Microoperación
1	MAR ← PC, read MBR ← Memoria PC ← PC + 2 IR ← MBR
2	(no hay que hacer nada, el operando fuente está en un registro)
3	MBR ← R4 SP ← SP – 4 MAR ← SP, write Memoria ← MBR

3) ADD [R5], 8 ; direccionamiento indirecto a registro e inmediato

Fase	Microoperación
1	MAR ← PC, read MBR ← Memoria PC ← PC + 7 IR ← MBR
2	MAR ← R5, read MBR ← Memoria
3	MBR ← MBR + 8, write Memoria ← MBR

4) SAL [00120034h+R3],R2 ; direccionamiento indexado y a registro

Fase	Microoperación
1	MAR ← PC, read MBR ← Memoria PC ← PC + 7 IR ← MBR
2	MAR ← 00120034h+R3, read MBR ← Memoria

Fase	Microoperación
3	MBR ← MBR <desp. izquierda> R2, write Memoria ← MBR

5) JE etiqueta ; direccionamiento relativo a PC, donde *etiqueta* está codificada como un desplazamiento

Fase	Micro-operación
1	MAR ← PC, read MBR ← Memoria PC ← PC + 4 IR ← MBR
2	(no hay que hacer nada, se entiende etiqueta como op. fuente)
3	Si bit de resultado Z=1 PC ← PC + etiqueta; si no, no hacer nada

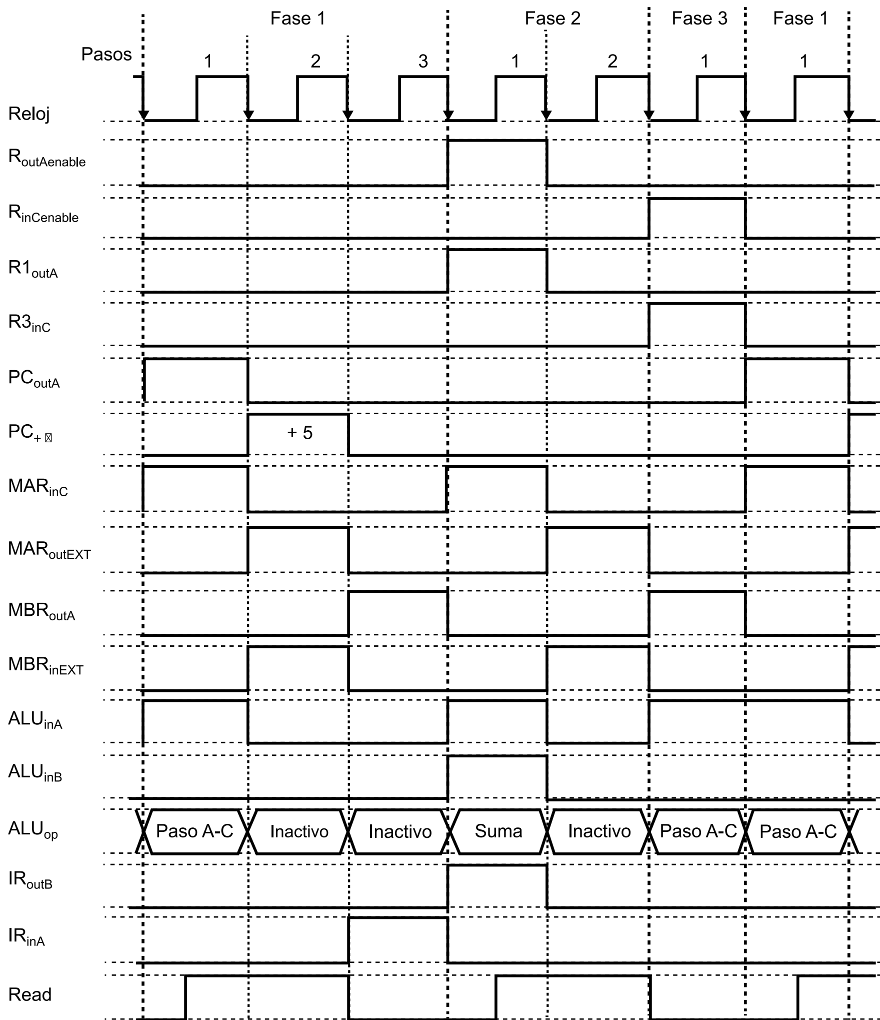
4.6. Ejemplo de señales de control y temporización

Veamos el diagrama de tiempo de la ejecución en CISCA de una instrucción:

MOV R3, [R1+10]

El operando destino utiliza direccionamiento a registro y el operando fuente, direccionamiento relativo a registro base. Si codificamos esta instrucción, veremos que ocupa 5 bytes.

Fase	Paso	Microoperación	
1	1	MAR ← PC, read	
	2	MBR ← Memoria	PC ← PC + 5
	3	IR ← MBR	
2	1	MAR ← R1 + 10, read	
	2	MBR ← Memoria	
3	1	R3 ← MBR	
1	1	MAR ← PC, read	Siguiente instrucción



F1.P1: MAR ← PC, read

Hacemos la transferencia desde el PC al MAR. Para hacer esta transferencia, conectamos la salida del PC al bus A activando la señal PC_{outA}, hacemos pasar el valor del PC al bus C mediante la ALU seleccionando la operación Pas A-C, y activando las señales ALU_{inA}. Para conectar la entrada a la ALU desde el bus A, conectamos la entrada del MAR al bus C al activar la señal MAR_{inC}.

También activamos la señal de lectura de la memoria Read, para indicar a la memoria que iniciamos un ciclo de lectura.

F1.P2: MBR ← Memoria, PC ← PC + 5

Finalizamos la lectura de la memoria manteniendo activa la señal de Read durante todo el ciclo de reloj y transferimos el dato al MBR. Para hacer esta transferencia se activa la señal MAR_{outEXT}, la memoria pone el dato de esta dirección en el bus del sistema y la hacemos entrar directamente al registro MBR activando la señal MBR_{inEXT}.

Como ya tenemos la instrucción en el IR y la hemos empezado a descodificar, sabemos su longitud. En este caso es 5; por lo tanto, incrementamos el PC en 5. Para hacerlo utilizamos el circuito autoincrementador que tiene el registro PC, por medio de la activación de la señala $PC_{+\Delta}$ indicando un incremento de 5.

F1.P3: IR ← MBR

Transferimos el valor leído de la memoria que ya tenemos en el MBR al IR. Conectamos la salida del MBR al bus A, activando la señal MBR_{outA}, y conectamos la entrada del IR al bus A activando la señal IR_{inA}.

Una vez que hemos leído la instrucción, finaliza la fase de lectura de esta y comienza la fase de lectura de los operandos fuente.

F2.P1: MAR ← R1 + 10, read

Calculamos la dirección de memoria en la que está almacenado el operando fuente, y como es un direccionamiento relativo a registro base debemos sumar el contenido del registro base R1 con el desplazamiento, que vale 10 y que tenemos en el IR.

Conectamos la salida de R1 al bus A activando las señales $R_{outAenable}$ para conectar el banco de registros al bus, y $R1_{outA}$ para indicar que el registro que pondrá el dato en el bus A es el R1. También debemos poner el 10 que tenemos en el registro IR en el bus B, activando la señal IR_{outB}. Ya tenemos los datos preparados. Para la suma en la ALU seleccionaremos la operación de suma, activaremos la señal ALU_{inA} para conectar la entrada a la ALU desde el bus A, y activaremos la señal ALU_{inB} para conectar la entrada a la ALU desde el bus B. El resultado quedará en el bus C, que podremos recoger conectando la entrada del MAR al bus C y activando la señal MAR_{inC}.

También activamos la señal de lectura de la memoria Read.

F2.P2: MBR ← Memoria

Finalizamos la lectura de la memoria manteniendo activa la señal de Read durante todo el ciclo de reloj y transferimos el dato al MBR. Para hacer esta transferencia se activa la señal MAR_{outEXT}. La memoria pone el dato de esta dirección de memoria en el bus externo y la hacemos entrar directamente al registro MBR activando la señal MBR_{inEXT}.

Finaliza así la fase de lectura de los operandos fuente y empieza la fase de ejecución de la instrucción y almacenamiento del operando destino.

F3.P1: R3 ← MBR

Transferimos el valor leído de la memoria que ya tenemos en MBR a R3. Conectamos la salida del MBR al bus A activando la señal MBR_{outA}. Hacemos pasar el dato al bus interno C mediante la ALU seleccionando la operación Paso A-C, activando la señal ALU_{inA} para conectar la entrada a la ALU desde el bus A y activando la señal $R_{inCenable}$ para conectar el banco de registros al bus C y $R3_{inC}$ para indicar que el registro que recibe el dato del bus C es el R3.

Finaliza la ejecución de esta instrucción y empieza la ejecución de la instrucción siguiente.