Estructura de computadores

Miquel Albert Orenga
Gerard Enrique Manonellas

PID_00177069

Material docente de la UOC

Universitat Oberta de Catalunya
www.uoc.edu

Miquel Albert Orenga

Licenciado en Informática por la Universidad Autónoma de Barcelona. Profesor de la Escuela Universitaria Tomàs Cerdà (centro adscrito a la UAB). Desarrolla su actividad docente en las áreas de estructura y arquitectura de computadores, redes y bases de datos.

Gerard Enrique Manonellas

Licenciado en Informática por la Universidad Autónoma de Barcelona. Profesor de la Escuela Universitaria Tomàs Cerdà (centro adscrito a la UAB). Desarrolla su actividad docente en las áreas de estructura y arquitectura de computadores y SOA.

El encargo y la creación de este material docente han sido coordinados por los profesores: Montse Serra Vizern, David Bañeres Besora (2011)

Primera edición: septiembre 2011

Av. Tibidabo, 39-43, 08035 Barcelona
Diseño: Manel Andreu
Realización editorial: Eureca Media, SL
Depósito legal: B-23.646-2011

Contenidos

Módulo didáctico 6

Programación en ensamblador (x86-64)

Miquel Albert Orenga y Gerard Enrique Manonellas

1. Arquitectura del computador
2. Lenguajes de programación
3. El lenguaje de ensamblador para la arquitectura x86-64
4. Introducción al lenguaje C
5. Conceptos de programación en ensamblador y C
6. Anexo: manual básico del juego de instrucciones

Módulo didáctico 7

La arquitectura CISCA

Miquel Albert Orenga y Gerard Enrique Manonellas

1. Organización del computador
2. Juego de instrucciones
3. Formato y codificación de las instrucciones
4. Ejecución de las instrucciones

Bibliografía

Angulo, J. M. (2003). *Fundamentos y Estructura de Computadores* (2.ª edición). Paraninfo.

Angulo, J. M. (2006). *Microcontroladores PIC. Diseño práctico de aplicaciones. Segunda parte: PIC16F87X, PIC18FXXXX* (2.ª edición). McGraw-Hill.

Angulo, J. M. (2007). *Microcontroladores PIC. Primera parte* (4.ª edición). McGraw-Hill.

Dandamundi, S. (2005). *Guide to Assembly Language Programming in Linux* (1.ª edición). Springer.

Charte, F. (2003). *Ensamblador para DOS, Linux y Windows* (1.ª edición). Anaya Multimedia.

Duntemann, J. (2009). *8088-8086/8087 Assembly Language Step-by-Step. Programming with Linux* (3.ª edición). John Wiley Publishing.

Hamacher, C.; Vranesic, Z.; Zaky, S. (2003). *Organización de computadores* (5.ª edición). McGraw-Hill.

Hennessy, John L.; Patterson, David A. (2002). *Arquitectura de computadores. Un enfoque cuantitativo* (1.ª edición). McGraw-Hill.

Miguel, Pedro de (2004). *Fundamentos de los computadores* (9.ª edición). Thomson-Paraninfo.

Patterson, David A.; Hennessy, John L. (2009). *Computer organization and design. The hardware/software interface* (4.ª edición). Morgan Kaufmann.

Prieto, A.; Lloris, A.; Torres, J. C. (2006). *Introducción a la Informática* (4.ª edición). McGraw-Hill.

Stallings, W. (2006). *Organización y arquitectura de computadores* (7.ª edición). Prentice Hall.

Stallings, W. (2009). *Computer Organization and Architecture: Designing for Performance* (8.ª edición). Prentice Hall.

Documentos electrónicos

AMD64. *Architecture Programmer's Manual*. Volumen 1: *Application Programming*.
<http://support.amd.com/us/Processor_TechDocs/24592.pdf>

AMD64. *Architecture Programmer's Manual*. Volumen 2: *System Programming*.
<http://support.amd.com/us/Processor_TechDocs/24593.pdf>

AMD64. *Architecture Programmer's Manual*. Volumen 3: *General-Purpose and System Instructions*.

<http://support.amd.com/us/Processor_TechDocs/24594.pdf>

Intel® 64; IA-32. *Architectures Software Developer's Manuals*.
<http://www.intel.com/products/processor/manuals/>

El computador

Miquel Albert Orenga
Gerard Enrique Manonellas

PID_00177070

Universitat Oberta de Catalunya
www.uoc.edu

Índice

Introducción

En este módulo se describe el concepto de computador y también su organización interna, los elementos que forman parte de él, el funcionamiento general que tienen y cómo se interconectan.

Se explican los dos tipos de organizaciones principales, Von Neumann y Harvard, y se ven cuáles son las características de cada una de estas organizaciones.

Asimismo, se presentan los dos tipos de computadores que utilizan habitualmente arquitectura Harvard:

- Los microcontroladores.
- Los procesadores digitales de señales (DSP).

Finalmente, se realiza una explicación breve sobre la evolución de los computadores desde las primeras máquinas electrónicas de cálculo hasta los computadores actuales, y se muestra la organización de los microprocesadores multinúcleo.

Objetivos

Con el estudio de este módulo se pretende que el estudiante alcance los objetivos siguientes:

1. Entender el concepto de computador.

2. Conocer los dos tipos de organizaciones de un computador más habituales: Von Neumann y Harvard.

3. Conocer la estructura de las dos aplicaciones más habituales de la arquitectura Harvard: microcontroladores y DSP.

4. Conocer de manera general cómo ha evolucionado el concepto y la estructura del computador a lo largo del tiempo.

1. El computador

Un computador se puede definir como una máquina electrónica capaz de hacer las tareas siguientes:

- Aceptar información.
- Almacenarla.
- Procesarla según un conjunto de instrucciones.
- Producir y proporcionar unos resultados.

Un computador dispone de tres componentes principales para efectuar las tareas descritas anteriormente:

1) Unidades de E/S para aceptar información y comunicar los resultados.

2) Un procesador para procesar la información.

3) Una memoria para almacenar la información y las instrucciones.

Es necesario un cuarto componente que conecte entre sí el resto de los componentes: un sistema de interconexión que permita mover la información entre los tres componentes del computador.

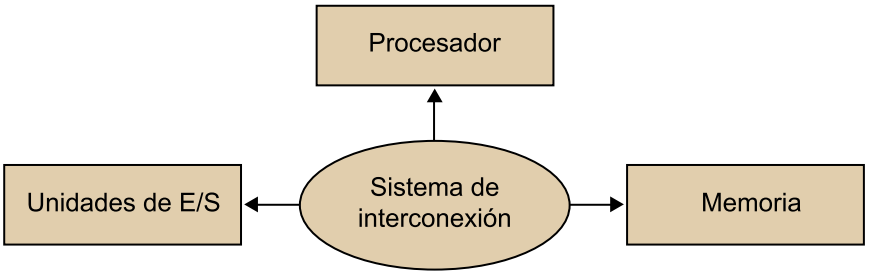
Resumimos a continuación las tareas que debe realizar cada uno de los componentes del computador:

- **Procesador**: se encarga de gestionar y controlar las operaciones del computador.

- **Memoria**: almacena información (los programas y los datos necesarios para ejecutarlos).

- **Sistema de E/S**: transfiere los datos entre el computador y los dispositivos externos, permite comunicarse con los usuarios del computador, introduciendo información y presentando resultados, y también permite comunicarse con otros computadores.

- **Sistema de interconexión**: proporciona los mecanismos necesarios para interconectar todos los componentes.



1.1. Arquitectura y organización del computador

La arquitectura y la organización del computador son conceptos que habitualmente se confunden o se utilizan de manera indistinta, aunque según la mayoría de los autores tienen significados diferentes. Es interesante dejar claros los dos conceptos.

La **arquitectura del computador** hace referencia al conjunto de elementos del computador que son visibles desde el punto de vista del programador de ensamblador.

Los elementos habituales asociados a la arquitectura del computador son los siguientes:

- Juego de instrucciones y modos de direccionamiento del computador.
- Tipos y formatos de los operandos.
- Mapa de memoria y de E/S.
- Modelos de ejecución.

Ved también

Estos conceptos se estudian en el módulo "Juego de instrucciones".

La **organización o estructura del computador** se refiere a las unidades funcionales del computador y al modo como están interconectadas. Describe un conjunto de elementos que son transparentes al programador.

Los elementos habituales asociados a la arquitectura del computador son los siguientes:

- Sistemas de interconexión y de control.
- Interfaz entre el computador y los periféricos.
- Tecnologías utilizadas.

Teniendo en cuenta esta diferencia, podemos tener computadores con una organización diferente, pero que comparten la misma arquitectura.

Por ejemplo, los microprocesadores Intel64 tienen una organización diferente de los microprocesadores AMD64, sin embargo, comparten una misma arquitectura (excepto ciertas diferencias), la arquitectura que se denomina *x86-64*.

1.2. Tipos de arquitecturas

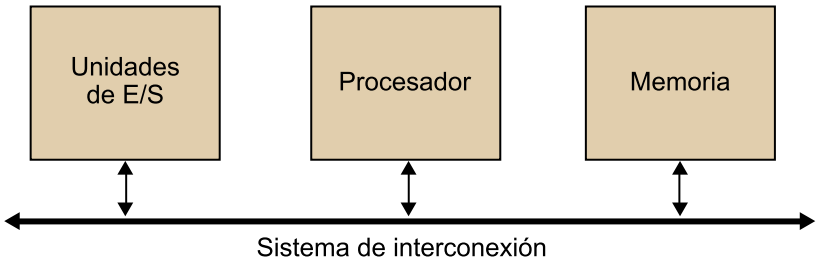
Aunque hablamos de conceptos de organización de los computadores, se mantiene tradicionalmente el término de *arquitectura* para distinguir los dos tipos de organización más habituales: la arquitectura Von Neumann y la arquitectura Harvard.

Se puede decir que la mayoría de los computadores actuales utilizan la arquitectura Von Neumann, o una arquitectura Von Neumann modificada, ya que a medida que los computadores han evolucionado se le ha añadido a esta características procedentes de la arquitectura Harvard.

La diferencia principal entre las dos arquitecturas se encuentra en el mapa de memoria: mientras que en la arquitectura Von Neumann hay un único espacio de memoria para datos y para instrucciones, en la arquitectura Harvard hay dos espacios de memoria separados: un espacio de memoria para los datos y un espacio de memoria para las instrucciones.

2. Arquitectura Von Neumann

Como ya se ha comentado en la descripción de un computador hecha en el apartado 1 del módulo, un computador está compuesto por los elementos siguientes: un procesador, una memoria, unidades de E/S y un sistema de interconexión. Todos estos elementos están presentes en la arquitectura Von Neumann.



En una máquina Von Neumann, la manera de procesar la información se especifica mediante un programa y un conjunto de datos que están almacenados en la memoria principal.

Los programas están formados por instrucciones simples, denominadas *instrucciones máquina*. Estas instrucciones son básicamente de los tipos siguientes:

- Transferencia de datos (mover un dato de una localización a otra).
- Aritméticas (suma, resta, multiplicación, división).
- Lógicas (AND, OR, XOR, NOT).
- Ruptura de secuencia (salto incondicional, salto condicional, etc.).

La arquitectura Von Neumann se basa en tres propiedades:

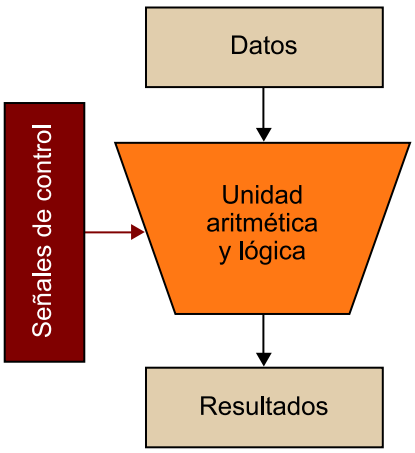
1) Hay un único espacio de memoria de lectura y escritura, que contiene las instrucciones y los datos necesarios.
2) El contenido de la memoria es accesible por posición, independientemente de que se acceda a datos o a instrucciones.
3) La ejecución de las instrucciones se produce de manera secuencial: después de ejecutar una instrucción se ejecuta la instrucción siguiente que hay en la memoria principal, pero se puede romper la secuencia de ejecución utilizando instrucciones de ruptura de secuencia.

El objetivo de la arquitectura Von Neumann es construir un sistema flexible que permita resolver diferentes tipos de problemas. Para conseguir esta flexibilidad, se construye un sistema de propósito general que se pueda programar para resolver los diferentes tipos de problemas. Para cada problema concreto se define un programa diferente.

2.1. Procesador

Un sistema de propósito general debe ser capaz de hacer unas operaciones aritméticas y lógicas básicas, a partir de las cuales se puedan resolver problemas más complejos.

Para conseguirlo, el procesador ha de disponer de una **unidad aritmética y lógica** (ALU) que pueda hacer un conjunto de operaciones. La ALU realiza una determinada operación según unas señales de control de entrada. Cada operación se lleva a cabo sobre un conjunto de datos y produce resultados. Por lo tanto, los resultados son generados según las señales de control y de los datos.



Desde el punto de vista de las instrucciones, cada instrucción máquina que se ejecuta en el procesador genera un determinado conjunto de señales a fin de que la ALU haga una operación determinada.

Desde el punto de vista de las operaciones que lleva a cabo la ALU, se puede decir que cada operación consiste en activar un conjunto de señales de control. Si se codifica cada conjunto de señales de control con un código, obtenemos un conjunto de códigos. Este conjunto de códigos define el conjunto de instrucciones con el que se puede programar el computador.

No todas las instrucciones corresponden a operaciones de la ALU. Las instrucciones de transferencia de datos, por ejemplo, pueden mover datos entre diferentes localizaciones del computador sin la intervención de la ALU.

Dentro del procesador es necesaria una unidad, denominada **unidad de control**, que sea capaz de interpretar las instrucciones para generar el conjunto de señales de control necesarias para gobernar la ejecución de las instrucciones.

También es necesario que el procesador disponga de un conjunto de registros (elementos de almacenamiento de información rápidos pero de poca capacidad) con los que sea capaz de trabajar la ALU, de donde leerá los datos necesarios para ejecutar las operaciones y donde almacenará los resultados de las operaciones hechas.

2.2. Memoria y unidades de E/S

Si analizamos el proceso de ejecución de las instrucciones, veremos que son necesarios otros elementos para construir un computador: la memoria principal y las unidades de E/S.

Las instrucciones que ejecuta el computador y los datos necesarios para cada instrucción están almacenados en la memoria principal, pero para introducirlos en la memoria es necesario un dispositivo de entrada. Una vez ejecutadas las instrucciones de un programa y generados unos resultados, estos resultados se deben presentar a los usuarios y, por lo tanto, es necesario algún tipo de dispositivo de salida.

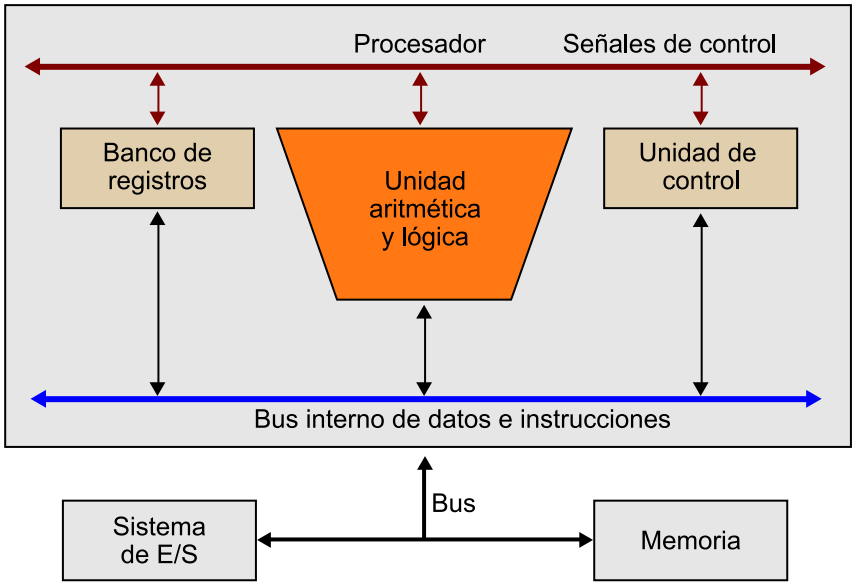
En un computador con arquitectura Von Neumann, además del procesador, son necesarios otros elementos:

- Dispositivos de entrada.
- Memoria principal.
- Dispositivos de salida.

Normalmente los dispositivos de entrada y de salida se tratan agrupados y se habla de *dispositivos de E/S*.

2.3. Sistema de interconexión

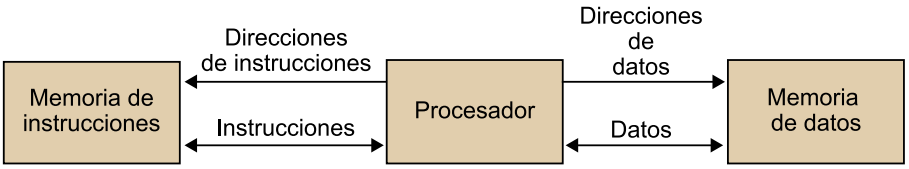
El medio de interconexión habitual en la arquitectura Von Neumann es el bus, un medio de comunicación compartido o multipunto donde se conectan todos los componentes que se quiere interconectar. Como se trata de un medio compartido, es necesario un mecanismo de control y acceso al bus. El sistema de interconexión es necesario pero generalmente no se considera una unidad funcional del computador.


Procesador
Señales de control
Banco de registros
Unidad aritmética y lógica
Unidad de control
Bus interno de datos e instrucciones
Bus
Sistema de E/S
Memoria

3. Arquitectura Harvard

La organización del computador según el modelo Harvard, básicamente, se distingue del modelo Von Neumann por la división de la memoria en una memoria de instrucciones y una memoria de datos, de manera que el procesador puede acceder separada y simultáneamente a las dos memorias.

Arquitectura Harvard



El procesador dispone de un sistema de conexión independiente para acceder a la memoria de instrucciones y a la memoria de datos. Cada memoria y cada conexión pueden tener características diferentes; por ejemplo, el tamaño de las palabras de memoria (el número de bits de una palabra), el tamaño de cada memoria y la tecnología utilizada para implementarlas.

Debe haber un mapa de direcciones de instrucciones y un mapa de direcciones de datos separados.

Los microcontroladores y el DSP (procesador de señales digitales o *digital signal processor*) son dos tipos de computadores que utilizan arquitectura Harvard. Veamos a continuación las características más relevantes de estos dos tipos de computadores de uso específico.

Usos de la arquitectura Harvard

La arquitectura Harvard no se utiliza habitualmente en computadores de propósito general, sino que se utiliza en computadores para aplicaciones específicas.

3.1. Microcontroladores

Un controlador o microcontrolador es un sistema encargado de controlar el funcionamiento de un dispositivo, como, por ejemplo, controlar que el nivel de un depósito de agua esté siempre entre un nivel mínimo y un nivel máximo o controlar las funciones de un electrodoméstico.

Actualmente se implementan utilizando un único circuito integrado, y por este motivo se denominan *microcontroladores* en lugar de simplemente *controladores*.

Un microcontrolador se considera un computador dedicado. Dentro de la memoria se almacena un solo programa que controla un dispositivo.

Un microcontrolador normalmente es un circuito integrado de dimensiones reducidas que se puede montar en el mismo dispositivo que ha de controlar (microcontrolador incrustado).

Aplicaciones de los microcontroladores

Algunos de los campos de aplicación más habituales de los microcontroladores son los siguientes:

- **Telecomunicaciones**. En el campo de las telecomunicaciones, los productos que utilizan frecuentemente microcontroladores son los teléfonos móviles.
- **Productos de gran consumo**. En los productos de gran consumo se utilizan microcontroladores en muchos electrodomésticos de línea blanca (lavadoras, lavavajillas, microondas, etc.) y de línea marrón (televisores, reproductores de DVD, aparatos de radio, etc.).
- **Automoción**. En la industria del automóvil se utilizan microcontroladores para controlar buena parte de los sistemas del coche; por ejemplo, para controlar los *airbags*, o el frenado.
- **Informática**. En la industria informática hay muchos dispositivos periféricos que integran microcontroladores: ratones, teclados, impresoras, escáneres, discos duros, etc.
- **Industria**. En el mundo industrial se utilizan en diferentes ámbitos, como la robótica o el control de motores.

3.1.1. Estructura de un microcontrolador

Un microcontrolador incorpora en un único circuito integrado todas las unidades necesarias para que funcione. Se trata de un computador completo pero de prestaciones limitadas.

Tal como se puede ver en la figura siguiente, las unidades que forman un microcontrolador se pueden agrupar en tres bloques principales:

1) Unidad de proceso:

- Procesador
- Memoria de programa
- Memoria de datos
- Líneas de interconexión

2) Dispositivos de E/S:

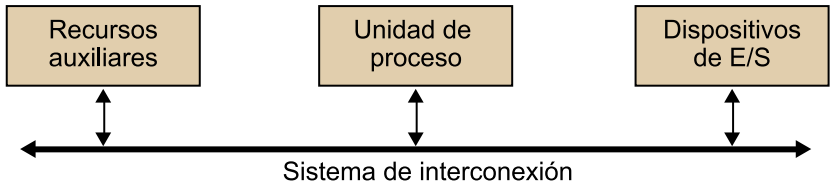
- Temporizadores
- Convertidores analógico-digital
- Comparadores analógicos
- Puertos de comunicación

3) Recursos auxiliares:

- Circuito de reloj

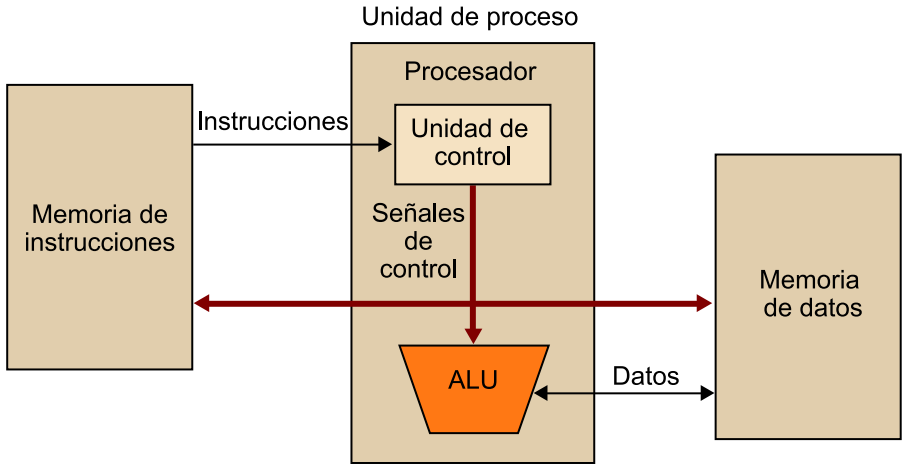
- Modos de bajo consumo
- Temporizador de vigilancia o *watchdog*
- Reinicialización o *reset*

Estructura de un microcontrolador



3.1.2. Organización de la unidad de proceso

A continuación se describe de manera general cada uno de los elementos que forman la **unidad de proceso** de un microcontrolador:



1) Procesador. De manera parecida a los procesadores de otros tipos de computadores, dispone de dos unidades funcionales principales: una unidad de control y una unidad aritmética y lógica.

Para ejecutar una instrucción, la unidad de control lee la instrucción de la memoria de instrucciones, genera las señales de control necesarias para obtener los operandos de la memoria de datos y después ejecuta la instrucción mediante la ALU y almacena el resultado producido en la memoria de datos.

2) Memoria de instrucciones. Es la memoria donde se almacenan las instrucciones del programa que debe ejecutar el microcontrolador. El tamaño de las palabras de la memoria se adapta al número de bits de las instrucciones del microcontrolador.

La memoria de instrucciones se implementa utilizando memorias no volátiles: ROM, PROM, EPROM, EEPROM o flash.

Si el programa que ha de ejecutar el microcontrolador es siempre el mismo, la capacidad de la memoria se adecua al tamaño previsto que tendrán los programas que tiene que ejecutar, con el fin de optimizar el espacio.

3) Memoria de datos. En esta memoria se almacenan los datos utilizados por los programas. Los datos varían continuamente y, por lo tanto, hay que implementarla utilizando memorias volátiles, memoria RAM, sobre la cual se pueden realizar operaciones de lectura y escritura. Habitualmente se utiliza SRAM (memoria RAM estática o *static RAM*). Si es necesario guardar algunos datos de manera permanente o que varíen poco (configuración o estado del microcontrolador), se utiliza memoria EEPROM o flash.

4) Líneas de interconexión. Son las líneas que interconectan los diferentes elementos que forman la unidad de proceso.

3.1.3. Dispositivos de E/S y recursos auxiliares

Aparte de la unidad de proceso, un microcontrolador utiliza dispositivos de E/S y otros recursos auxiliares. Según la aplicación del microcontrolador, son necesarios unos recursos u otros. Los recursos más habituales que hay en la mayoría de los microcontroladores son los siguientes:

- **Circuito de reloj**: genera los pulsos para sincronizar todo el sistema.
- **Temporizadores**: permiten contar el tiempo y establecer retardos.
- **Temporizador de vigilancia**: circuito temporizador que provoca una reinicialización del sistema si el programa se bloquea por alguna condición de fallo.
- **Convertidores analógico-digital (ADC)** y **digital-analógico (DAC)**.
- **Comparadores analógicos**: permiten tratar señales analógicas.
- **Sistema de protección para posibles fallos de la alimentación**.
- **Modos de funcionamiento de bajo consumo**.
- **Módulos de comunicación**: en serie, paralelo, USB, etc. Mediante estos módulos se obtienen o se envían datos de los dispositivos externos al microcontrolador.

3.2. Procesador de señales digitales

Un procesador de señales digitales o *digital signal processor* (DSP) es un dispositivo capaz de procesar en tiempo real señales procedentes de diferentes fuentes.

Un DSP tiene características propias de los microcontroladores y también de los microprocesadores. Esto provoca que muchas veces sea difícil distinguir estos tres conceptos.

Dispone de un procesador con gran potencia de cálculo preparado para tratar señales en tiempo real y puede hacer operaciones aritméticas a gran velocidad; generalmente, también dispone de convertidores de señales analógicas a digitales (ADC) o convertidores de señales digitales a analógicas (DAC).

Una de las características principales de los DSP es que implementan muchas operaciones por hardware que otros procesadores hacen por software, e incorporan habitualmente unidades específicas para realizar sumas y productos. Por este motivo, el hardware del procesador puede ser más complejo que el de algunos microcontroladores o microprocesadores.

Otra de las diferencias importantes entre los DSP y otros procesadores es que están diseñados para que sean escalables y para trabajar en paralelo con otros DSP. Esto hace necesario disponer de elementos para sincronizar el funcionamiento de diferentes DSP.

Aplicaciones de los DSP

Algunas de las aplicaciones más habituales de los DSP son el procesamiento de audio digital, la compresión de audio, el procesamiento de imágenes digitales, la compresión de vídeo, el procesamiento de voz, el reconocimiento de voz, las comunicaciones digitales, el radar, el sonar, la sismología y la medicina.

Algunos ejemplos concretos de estas aplicaciones son los teléfonos móviles, los reproductores de audio digital (MP3), los módems ADSL, los sistemas de telefonía de manos libres (con reconocimiento de voz) y los osciloscopios.

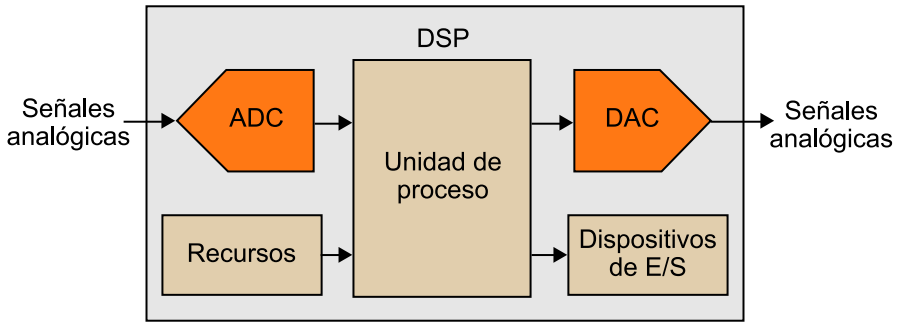
3.2.1. Organización de un DSP

La estructura interna corresponde básicamente a una arquitectura de tipo Harvard, muchas veces mejorada para acelerar la ejecución de las instrucciones y la realización de operaciones aritméticas.

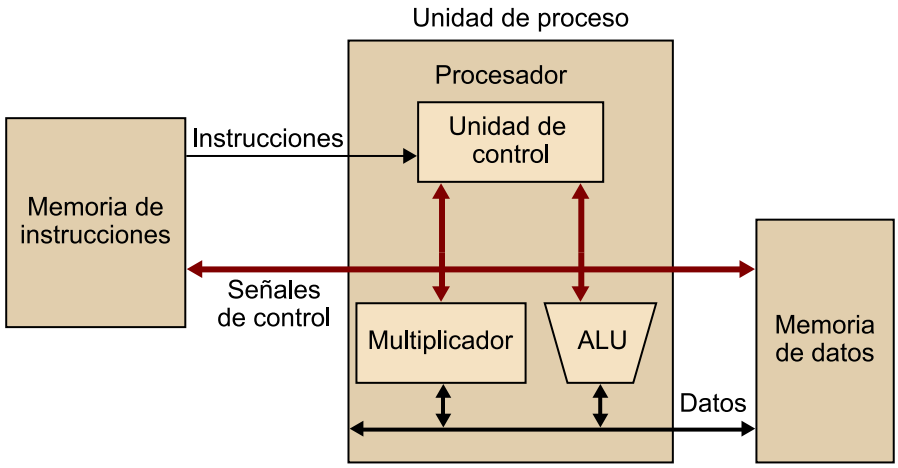
Mejoras de un DSP

Las mejoras que pueden incluir un DSP son varias: se incluyen buses para transferir instrucciones y datos de tamaño superior al necesario, más de un bus de direcciones y de datos para acceder a los datos, implementación de técnicas de paralelismo para permitir la segmentación de la ejecución de las instrucciones y hacer varias operaciones elementales por ciclo, operaciones lógicas y aritméticas complejas, etc.

Organización interna de un DSP



El procesador habitualmente dispone de múltiples ALU y multiplicadores que son capaces de hacer distintas operaciones aritméticas en un solo ciclo de reloj del sistema.



4. Evolución de los computadores

Las primeras máquinas de cálculo, los primeros computadores de la historia, estaban construidos a partir de válvulas de vacío y se programaban mecánicamente mediante interruptores. Ocupaban espacios muy grandes y tenían una capacidad de cálculo muy limitada. Los primeros computadores de este tipo fueron el ENIAC y el IAS.

La segunda generación de computadores se basaba en el uso de transistores (hechos a partir de silicio), que sustituyen a las válvulas de vacío. Se trataba de computadores mucho más pequeños y económicos. Los primeros ejemplos de computadores basados en transistores fueron el IBM 7000 y el DEC PDP1.

Las siguientes generaciones de computadores han basado la construcción en transistores y en la microelectrónica, que ha permitido integrar cantidades elevadas de transistores en un solo circuito integrado (chip).

La integración de transistores empieza a mediados de los años sesenta y, a medida que pasa el tiempo, se consiguen niveles de integración más elevados. Según el número de transistores que se puede incluir en un chip, se definen los niveles o escalas de integración siguientes:

- *small scale integration* (SCI): hasta 100 transistores en un solo chip,
- *medium scale integration* (MSI): por encima de 100 transistores en un chip,
- *large scale integration* (LSI): por encima de 1.000 transistores en un chip,
- *very large scale integration* (VLSI): más de 10.000 transistores en un chip y
- *ultra large scale integration* (ULSI): por encima de 1.000.000 de transistores en un chip.

En los procesadores actuales, el número de transistores en un chip está por encima de los 100 millones y, en algunos casos, llega a estar por encima de los 1.000 millones de transistores.

La evolución en la escala de integración ha afectado a la evolución de los microprocesadores y también a los sistemas de memoria, que se han beneficiado de los aumentos en la escala de integración.

Con respecto a la organización del computador, a lo largo de la historia de los computadores, aunque se ha mantenido la organización básica del modelo Von Neumann, se han ido añadiendo nuevos elementos y se ha ido modificando el modelo original.

A continuación se comenta la evolución de los elementos que forman el computador: procesador, memoria, sistema de E/S y sistema de interconexión.

4.1. Evolución del procesador

En las primeras generaciones de computadores los elementos que formaban el procesador eran elementos independientes, fabricados utilizando diferentes chips e interconectados con un bus. A medida que creció la escala de integración, cada vez hubo más unidades funcionales que se fueron integrando utilizando menos chips, hasta la aparición de lo que se denominó *microprocesador*.

El microprocesador es un procesador que integra en un solo chip todas las unidades funcionales.

Intel 4004

El primer microprocesador lo desarrolló Intel en 1971. Se trataba del Intel 4004, un microprocesador de 4 bits que podía dirigir una memoria de 640 bytes y que se había construido utilizando 2.300 transistores.

Hoy en día es equivalente hablar de *procesador* o de *microprocesador*, ya que todos los procesadores actuales se construyen como microprocesadores. Actualmente, además de incluir todas las unidades funcionales, se incluye un nivel de memoria caché o más de uno.

4.2. Evolución del sistema de memoria

Una de las mejoras más importantes ha sido la aparición de la jerarquía de memorias, con la incorporación de memorias cachés. La memoria caché es una memoria más rápida que la memoria principal, pero también de coste mucho más elevado. Por este motivo tiene un tamaño más reducido que la memoria principal.

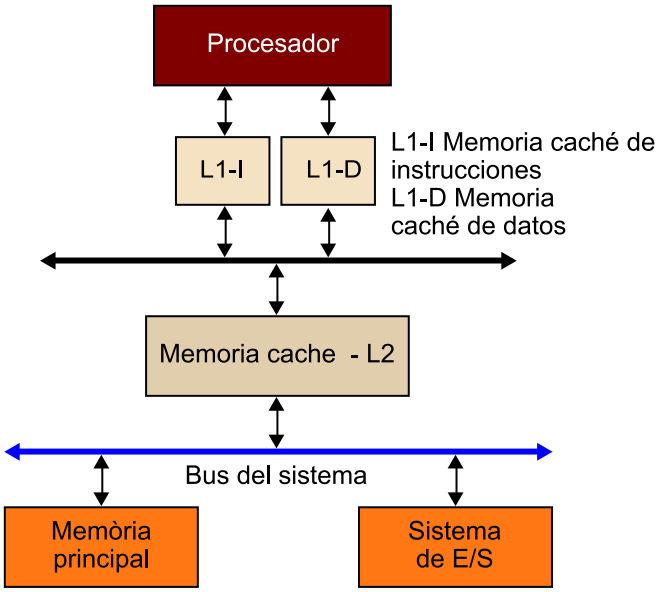
La memoria caché se coloca como una memoria intermedia entre la memoria principal y el procesador. Cuando el procesador necesita un dato o una instrucción, primero se comprueba si está en la memoria caché y solo en caso de que no lo esté se debe traer de la memoria principal para acceder a él.

La utilización de las memorias cachés ha ido evolucionando incorporando diferentes niveles de memoria caché. Actualmente se trabaja con tres niveles, denominados *L1*, *L2* y *L3*. Algunos niveles o todos juntos se pueden integrar en el mismo chip del microprocesador.

La memoria caché puede estar dividida en dos partes: una memoria caché de instrucciones y una de datos. Desde este punto de vista, se puede decir que los computadores con memoria caché dividida utilizan una arquitectura Harvard, o una arquitectura Harvard modificada, ya que la separación de la memoria solo existe en algunos niveles de la memoria caché, pero no en la memoria principal.

Ejemplo de memoria caché

En el ejemplo siguiente se muestra un computador con dos niveles de memoria caché (L1 y L2), en el que el primer nivel de memoria caché está dividido en una memoria caché de instrucciones y una memoria caché de datos.



4.3. Evolución del sistema de interconexión

El sistema de interconexión también ha evolucionado. En los primeros computadores, consistía en un solo bus al que se conectaban todos los elementos del computador. Este sistema facilitaba la conexión de los diferentes elementos del computador, pero como se trataba de un solo bus que todos tenían que utilizar, se generaba un cuello de botella que hacía reducir las prestaciones del computador.

En los computadores actuales se ha ampliado y diversificado el número y tipo de sistemas de interconexión. Actualmente se utiliza una jerarquía de buses separados parecida a la jerarquía de memoria con el objetivo de aislar los dispositivos más rápidos de los más lentos.

Las tendencias actuales pasan por utilizar buses de tipo serie de alta velocidad en lugar de buses paralelos y también por utilizar interconexiones punto a punto, que permiten eliminar los problemas de compartir un bus entre diferentes elementos del computador. Un diseño cada vez más habitual es el de disponer de una conexión directa entre el sistema de memoria y el procesador.

4.4. Evolución del sistema de E/S

Inicialmente, la comunicación del procesador con los periféricos se efectuaba utilizando programas que accedían directamente a los módulos de E/S; posteriormente se introdujo la técnica de E/S por interrupciones, en la que el procesador no necesitaba esperar a que el periférico estuviera disponible para hacer la transferencia. La siguiente mejora fue introducir el acceso directo a me-

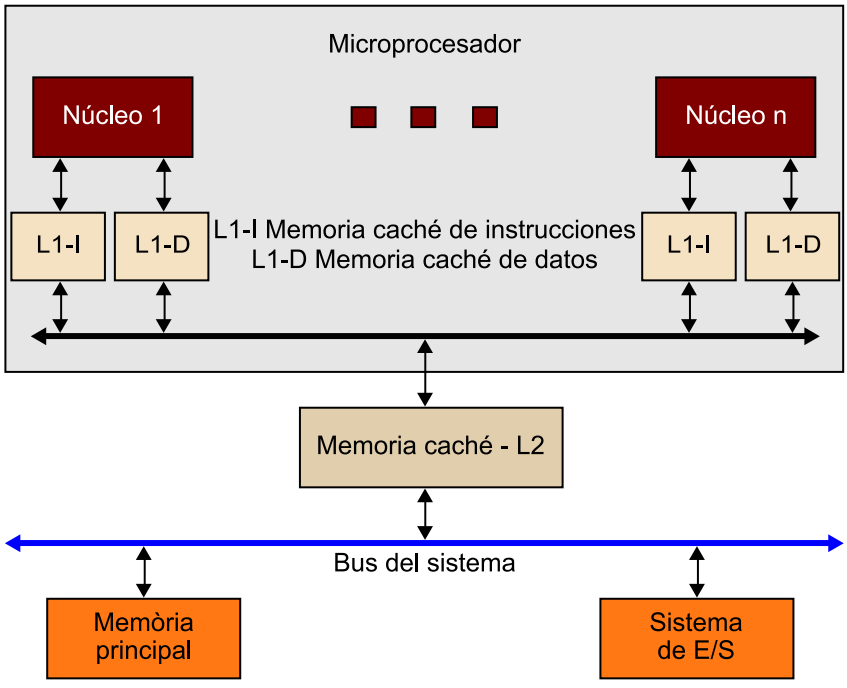
moria (DMA), que permite transferir bloques de datos entre el periférico y la memoria sin la intervención del procesador utilizando controladores de DMA. Estos controladores han evolucionado y se comportan como procesadores específicos de E/S, denominados también *canales de E/S*.

Los sistemas de interconexión externos, entre el computador y los dispositivos periféricos, también han ido evolucionando. En los primeros diseños se utilizaban básicamente sistemas de interconexión multipunto (buses) que habitualmente tenían múltiples líneas de datos (paralelas). Los sistemas de interconexión actuales incluyen buses de tipo serie (una única línea de datos) de alta velocidad, como Firewire o USB, y también sistemas de interconexión punto a punto o sistemas inalámbricos, como Bluetooth y Ethernet inalámbrica.

4.5. Microprocesadores multinúcleo

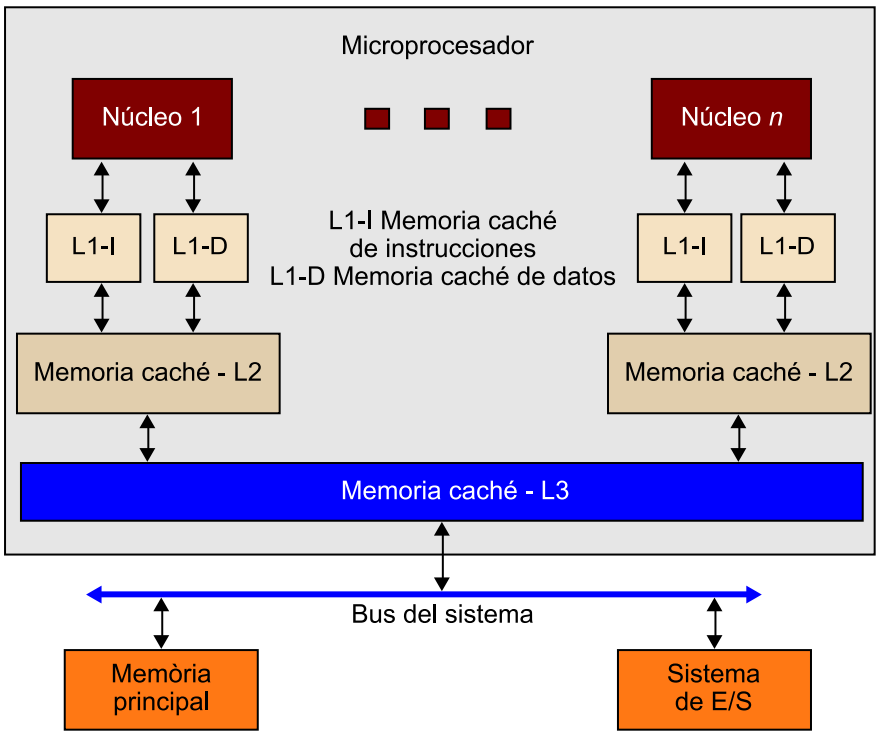
La evolución de los microprocesadores pasa por incluir en un solo chip varios núcleos, donde cada núcleo incluye todas las unidades funcionales de un procesador (registros, ALU y unidad de control), lo que da lugar a lo que se conoce como *procesador multinúcleo*.

Inicialmente, dentro del microprocesador se disponía de una memoria caché de primer nivel (denominada *L1*) para cada núcleo, habitualmente dividida en memoria caché de instrucciones y memoria caché de datos; fuera del microprocesador se disponía de una memoria caché de segundo nivel (L2) unificada (para instrucciones y datos) y compartida por todos los núcleos.



Esta organización ha variado mucho. Una primera evolución consistió en incorporar dentro del microprocesador el segundo nivel de memoria caché, y apareció un tercer nivel (L3) fuera del procesador.

Actualmente, dentro del microprocesador pueden estar los tres niveles de memoria caché (L1, L2 y L3). Dispone de una memoria caché de primer nivel para cada núcleo, dividida en memoria caché de instrucciones y memoria caché de datos, una memoria caché unificada de segundo nivel para cada núcleo y una memoria caché de tercer nivel unificada y compartida por todos los núcleos.



Resumen

En este módulo se ha explicado el concepto de computador de manera genérica y se han diferenciado los conceptos de arquitectura y de organización.

Se han visto brevemente los elementos principales que forman un computador y la organización que tienen.

A continuación se han descrito los dos tipos de arquitecturas más habituales: la arquitectura Von Neumann y la arquitectura Harvard.

Dentro de la arquitectura Von Neumann se han estudiado los elementos que componen un computador que utilice esta arquitectura y las características principales que tiene:

- Procesador
- Memoria
- Unidades de E/S
- Sistema de interconexión

De la arquitectura Harvard se han visto las características que la diferencian de la arquitectura Von Neumann y se han descrito los dos tipos de computadores que utilizan habitualmente esta arquitectura:

- Microcontroladores
- DSP

Finalmente, se ha llevado a cabo una descripción breve de la evolución que han tenido los computadores, analizando las mejoras que se han ido introduciendo en cada uno de los elementos que los componen: procesador, sistema de memoria, sistema de interconexión y sistema de E/S. Se ha acabado comentando la organización de los microprocesadores actuales: los microprocesadores multinúcleo.

Juego de instrucciones

Miquel Albert Orenga
Gerard Enrique Manonellas

PID_00177071

UOC
Universitat Oberta de Catalunya
www.uoc.edu

Índice

Introducción

El juego de instrucciones es el punto de encuentro entre el diseñador del computador y el programador. Desde el punto de vista del diseñador, es un paso más en la explicitación del procesador de un computador y, desde el punto de vista del programador en ensamblador, la herramienta básica para acceder a los recursos disponibles del computador.

Para definir un juego de instrucciones hay que saber cómo se ejecutan las instrucciones, qué elementos las forman y cuáles son las diferentes maneras de acceder a los datos y a las instrucciones. Estos son los conceptos que trataremos en este módulo y lo haremos de manera genérica para poderlos aplicar a todo tipo de problemas reales y de procesadores comerciales.

Veremos cómo estos conceptos afectan a los diferentes parámetros de la arquitectura del computador a la hora de definir el juego de instrucciones, qué restricciones nos imponen y cómo podemos llegar a un compromiso entre facilitar la tarea al programador y dar una eficiencia máxima al computador.

Objetivos

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

1. Saber cómo se define el juego de instrucciones de un procesador.
2. Conocer los tipos de instrucciones más habituales en el juego de instrucciones de un computador de propósito general.
3. Saber cómo funciona cada una de las instrucciones del juego de instrucciones.
4. Aprender las nociones básicas necesarias para utilizar un juego de instrucciones de bajo nivel.
5. Familiarizarse con las maneras de referenciar un dato en una instrucción.

1. Juego de instrucciones

La mayoría de los programas se escriben en lenguajes de alto nivel, como C++, Java o Pascal. Para poder ejecutar un programa escrito en un lenguaje de alto nivel en un procesador, este programa se debe traducir primero a un lenguaje que pueda entender el procesador, diferente para cada familia de procesadores. El conjunto de instrucciones que forman este lenguaje se denomina *juego de instrucciones* o *repertorio de instrucciones*.

Para poder definir un juego de instrucciones, habrá que conocer bien la arquitectura del computador para sacarle el máximo rendimiento; en este módulo veremos cómo queda definido un juego de instrucciones según una arquitectura genérica.

1.1. Ciclo de ejecución

Ejecutar un programa consiste en ejecutar una secuencia de instrucciones, de manera que cada instrucción lleva a cabo un ciclo de ejecución. Esta secuencia de instrucciones no será la secuencia escrita que hacemos de un programa, sino la secuencia temporal de ejecución de las instrucciones considerando las instrucciones de salto.

El **ciclo de ejecución** es la secuencia de operaciones que se hacen para ejecutar cada una de las instrucciones y lo dividiremos en cuatro 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.

Diferencias en las fases del ciclo de ejecución

Las fases del ciclo de ejecución son comunes en la mayoría de los computadores actuales y las diferencias principales se encuentran en el orden en el que se llevan a cabo algunas de las operaciones de cada fase o en el que se hacen algunas de las operaciones en otras fases.

En la tabla siguiente podéis ver de manera esquemática las operaciones que se realizan en cada fase:

Inicio del ciclo de ejecución	
Fase 1 Lectura de la instrucción	Leer la instrucción.
	Descodificar la instrucción.

Inicio del ciclo de ejecución	
	Actualizar el PC.
Fase 2 Lectura de los operandos fuente	Calcular la dirección y leer el primer operando fuente.
	Calcular la dirección y leer el segundo operando fuente.
Fase 3 Ejecución de la instrucción y almacenamiento del operando destino	Ejecutar la instrucción.
Fase 4 Comprobación de interrupciones	Comprobar si algún dispositivo ha solicitado una interrupción.

1.2. Arquitectura del juego de instrucciones

Las instrucciones de una arquitectura son autocontenidas; es decir, incluyen toda la información necesaria para su ejecución.

Uno de los elementos necesarios en cualquier instrucción es el **conjunto de operandos**. Los operandos necesarios para ejecutar una instrucción pueden encontrarse explícitamente en la instrucción o pueden ser implícitos.

La localización dentro del procesador de los operandos necesarios para ejecutar una instrucción y la manera de explicitarlos dan lugar a arquitecturas diferentes del juego de instrucciones.

Según los criterios de localización y la explicitación de los operandos, podemos identificar las arquitecturas del juego de instrucciones siguientes:

- **Arquitecturas basadas en pila**: los operandos son implícitos y se encuentran en la pila.
- **Arquitecturas basadas en acumulador**: uno de los operandos se encuentra de manera implícita en un registro denominado *acumulador*.
- **Arquitecturas basadas en registros de propósito general**: los operandos se encuentran siempre de manera explícita, ya sea en registros de propósito general o en la memoria.

Dentro de las arquitecturas basadas en registros de propósito general, podemos distinguir tres subtipos:

- **Registro-registro (o *load-store*)**. Solo pueden acceder a la memoria instrucciones de carga (*load*) y almacenamiento (*store*).
- **Registro-memoria**. Cualquier instrucción puede acceder a la memoria con uno de sus operandos.

- **Memoria-memoria**. Cualquier instrucción puede acceder a la memoria con todos sus operandos.

Arquitectura memoria-memoria

La arquitectura memoria-memoria es un tipo de arquitectura que prácticamente no se utiliza en la actualidad.

Ventajas de los subtipos de arquitecturas

El subtipo registro-registro presenta las ventajas siguientes respecto a los registro-memoria y memoria-memoria: la codificación de las instrucciones es muy simple y de longitud fija, y utiliza un número parecido de ciclos de reloj del procesador para ejecutarse. Como contrapartida, se necesitan más instrucciones para hacer la misma operación que en los otros dos subtipos.

Los subtipos registro-memoria y memoria-memoria presentan las ventajas siguientes respecto al registro-registro: no hay que cargar los datos en registros para operar en ellos y facilitan la programación. Como contrapartida, los accesos a memoria pueden crear cuellos de botella y el número de ciclos de reloj del procesador necesarios para ejecutar una instrucción puede variar mucho, lo que dificulta el control del ciclo de ejecución y lentifica su ejecución.

Podemos hablar de cinco tipos de arquitectura del juego de instrucciones:

1) Pila
2) Acumulador
3) Registro-registro
4) Registro-memoria
5) Memoria-memoria

Presentamos gráficamente las diferencias entre estos tipos de arquitectura y vemos cómo resuelven una misma operación: sumar dos valores (A y B) almacenados en memoria y guardar el resultado en una tercera posición de memoria (C).

Pila	Acumulador	Registro-registro	Registro-memoria	Memoria-memoria
PUSH [A]	LOAD [A]	LOAD R1 [A]	MOV R1 [A]	MOV [C] [A]
PUSH [B]	ADD [B]	LOAD R2, [B]	ADD R1, [B]	ADD [C], [B]
ADD	STORE [C]	ADD R2, R1	MOV [C], R2	
POP [C]		STORE [C], R2		

Hemos supuesto que en las instrucciones de dos operandos, el primer operando actúa como operando fuente y operando destino en el ADD y solo como operando destino en el resto, el segundo operando actúa siempre como operando fuente.
A, B y C son las etiquetas que representan las direcciones de memoria donde están almacenados los valores que queremos sumar y su resultado, respectivamente.
Los corchetes indican que tomamos el contenido de estas direcciones de memoria.

1.3. Representación del juego de instrucciones

El juego de instrucciones de una arquitectura se representa desde dos puntos de vista:

1) Desde el punto de vista del computador, cada instrucción se representa como una secuencia de bits que se divide en campos, en los que cada campo corresponde a uno de los elementos que forman la instrucción. Cada bit es la

representación de una señal eléctrica alta o baja. Esta manera de representar el juego de instrucciones se suele denominar *código de máquina* o *lenguaje de máquina*.

2) Desde el punto de vista del programador, las instrucciones se representan mediante símbolos y se utilizan expresiones mnemotécnicas o abreviaturas. Hay que tener presente que es muy difícil trabajar con las representaciones binarias propias del código de máquina. Esta manera de representar el juego de instrucciones se suele denominar *código de ensamblador* o *lenguaje de ensamblador*.

Arquitectura	Instrucción	Operación
CISCA	ADD R1, R2	Suma
Intel x86-64	MUL RAX	Multiplicación
MIPS	lw $S1, 100($S2)	Carga en un registro el valor almacenado en una posición de memoria

Ejemplo para la arquitectura CISCA

Instrucción para sumar dos números que están almacenados en los registros R1 y R2. El resultado quedará almacenado en R1.

Desde el punto de vista del programador: *código de ensamblador*

Código de operación	Operando 1	Operando 2
ADD	R1	R2

Desde el punto de vista del computador: *código de máquina*

Código de operación	Operando 1	Operando 2
20h	11h	12h
0010 0000	0001 0001	0001 0010
8 bits	8 bits	8 bits
← 24 bits→		

Ved también

La arquitectura CISCA es un modelo sencillo de máquina definida en el módulo 7 de esta asignatura.

1.4. Formato de las instrucciones

1.4.1. Elementos que componen una instrucción

Los elementos que componen una instrucción, independientemente del tipo de arquitectura, son los siguientes:

- **Código de operación**: especifica la operación que hace la instrucción.

- **Operando fuente**: para hacer la operación pueden ser necesarios uno o más operandos fuente; uno o más de estos operandos pueden ser implícitos.

- **Operando destino**: almacena el resultado de la operación realizada. Puede estar explícito o implícito. Uno de los operandos fuente se puede utilizar también como operando destino.

- **Dirección de la instrucción siguiente**: especifica dónde está la instrucción siguiente que se debe ejecutar; suele ser una información implícita, ya que el procesador va a buscar automáticamente la instrucción que se encuentra a continuación de la última instrucción ejecutada. Solo las instrucciones de ruptura de secuencia especifican una dirección alternativa.

1.4.2. Tamaño de las instrucciones

Uno de los aspectos más importantes a la hora de diseñar el formato de las instrucciones es determinar su tamaño. En este sentido, encontramos dos alternativas:

- **Instrucciones de tamaño fijo**: todas las instrucciones ocuparán el mismo número de bits. Esta alternativa simplifica el diseño del procesador y la ejecución de las instrucciones puede ser más rápida.

- **Instrucciones de tamaño variable**: el tamaño de las instrucciones dependerá del número de bits necesario para cada una. Esta alternativa permite diseñar un conjunto amplio de códigos de operación, el direccionamiento puede ser más flexible y permite poner referencias a registros y memoria. Como contrapartida, aumenta la complejidad del procesador.

Es deseable que el tamaño de las instrucciones sea múltiplo del tamaño de la palabra de memoria.

Para determinar el tamaño de los campos de las instrucciones utilizaremos:

1) Código de operación. La técnica más habitual es asignar un número fijo de bits de la instrucción para el código de operación y reservar el resto de los bits para codificar los operandos y los modos de direccionamiento.

Ejemplo

Si se destinan N bits para el código, tendremos disponibles 2^N códigos de operación diferentes; es decir, 2^N operaciones diferentes.

En instrucciones de tamaño fijo, cuantos más bits se destinen al código de operación, menos quedarán para los modos de direccionamiento y la representación de los operandos.

Se puede ampliar el número de instrucciones diferentes en el juego de instrucciones sin añadir más bits al campo del código de operación. Solo hay que añadir un campo nuevo, que llamaremos *expansión del código de operación*. Evidentemente, este campo solo se añade a las instrucciones en las que se haga ampliación de código.

Si de los 2^N códigos de los que disponemos reservamos x para hacer la expansión de código y el campo de expansión es de k bits, tendremos 2^k instrucciones adicionales por cada código reservado. De esta manera, en lugar de tener un total de 2^N instrucciones, tendremos $(2^n - x) + x \cdot 2^k$.

Ejemplo

Tenemos $n = 3$, $x = 4$ y $k = 2$. En lugar de disponer de $2^3 = 8$ instrucciones diferentes, tendremos $(2^3 - 4) + 4 \cdot 2^2 = 20$.

Para hacer la expansión del código de operación debemos elegir 4 códigos. En este caso hemos elegido los códigos (010, 011, 100, 101).

Códigos de operación de 3 bits		Códigos de operación con expansión del código	
		000	100 00
000		001	100 01
001		010 00	100 10
010		010 01	100 11
011		010 10	101 00
100		010 11	101 01
101		011 00	101 10
110		011 01	101 11
111		011 10	110
		011 11	111

Evidentemente, en las instrucciones de tamaño fijo, las instrucciones que utilicen el campo de expansión del código de operación dispondrán de menos bits para codificar el resto de los campos de la instrucción.

2) Operandos y modos de direccionamiento. Una vez fijados los bits del código de operación, podemos asignar los bits correspondientes a los operandos y a los modos de direccionamiento. Las dos maneras más habituales de indicar qué modo de direccionamiento utiliza cada uno de los operandos de la instrucción son las siguientes:

a) Con el código de operación: esta técnica se utiliza habitualmente cuando no se pueden utilizar todos los modos de direccionamiento en todas las instrucciones. Esta técnica es la utilizada principalmente en las arquitecturas PowerPC, MIPS y SPARC.

b) En un campo independiente: de esta manera, para cada uno de los operandos añadimos un campo para indicar qué modo de direccionamiento utiliza. El tamaño de este campo dependerá del número de modos de direccionamiento que tengamos y de si se pueden utilizar todos los modos en todos los operandos. Esta técnica es la utilizada principalmente en las arquitecturas Intel x86-64 y VAX.

Para codificar los operandos según los modos de direccionamiento que utilizan, hay que considerar una serie de factores:

a) Número de operandos de la instrucción: cuántos operandos tienen las instrucciones y si son siempre del mismo tamaño.
b) Uso de registros o de memoria como operandos: cuántos accesos a registros y a memoria puede haber en una instrucción.
c) Número de registros del procesador.
d) Rango de direcciones del computador: cuántos bits se necesitan para especificar una dirección que puede ser completa o parcial (como en el caso de memoria segmentada).
e) Número de bits para codificar los campos de desplazamiento o valores inmediatos.

Especificidades de la arquitectura CISCA

Las siguientes son las especificidades de la arquitectura CISCA:

- El número de operandos puede ser 0, 1 o 2 y de tamaño variable.
- No puede haber dos operandos que hagan referencia a memoria.
- Dispone de un banco de 16 registros.
- Memoria de 2^{32} direcciones de tipo byte (4 Gbytes). Necesitamos 32 bits para codificar las direcciones.
- Los desplazamientos se codifican utilizando 16 bits.
- Los valores inmediatos se codifican utilizando 32 bits.

1.5. Operandos

1.5.1. Número de operandos

Las instrucciones pueden utilizar un número diferente de operandos según el tipo de instrucción del que se trate.

Ejemplo

Hay que tener presente que una misma instrucción en máquinas diferentes puede utilizar un número diferente de operandos según el tipo de arquitectura del juego de instrucciones que utilice la máquina.

La instrucción aritmética de suma (C = A + B) utiliza dos operandos fuente (A y B) y produce un resultado que se almacena en un operando destino (C):

- En una arquitectura basada en pila, los dos operandos fuente se encontrarán en la cima de la pila y el resultado se almacenará también en la pila.
- En una arquitectura basada en acumulador, uno de los operandos fuente se encontrará en el registro acumulador, el otro estará explícito en la instrucción y el resultado se almacenará en el acumulador.
- En una arquitectura basada en registros de propósito general, los dos operandos fuente estarán explícitos. El operando destino podrá ser uno de los operandos fuente (instrucciones de dos operandos) o un operando diferente (instrucciones de tres operandos).

Según la arquitectura y el número de operandos de la instrucción podemos tener diferentes versiones de la instrucción de suma, tal como se ve en la tabla siguiente.

Pila	ADD	Suma los dos valores de encima de la pila
Acumulador	ADD R1	Acumulador = Acumulador + R1
Registro-registro	ADD R1, R2	R1 = R1 + R2
	ADD R3, R1, R2	R3 = R1 + R2
Registro-memoria	ADD R1, [A01Bh]	R1 = R1+ M(A01Bh)
	ADD R2, R1, [A01Bh]	R2 = R1+ M(A01Bh)
Memoria-memoria	ADD [A01Dh], [A01Bh]	M(A01Dh) = M(A01Dh) + M(A01Bh)

Hemos supuesto que, en las instrucciones de dos operandos, el primer operando actúa como operando fuente y también como operando destino. Los valores entre corchetes expresan una dirección de memoria.

1.5.2. Localización de los operandos

Los operandos representan los datos que hemos de utilizar para ejecutar una instrucción. Estos datos se pueden encontrar en lugares diferentes dentro del computador. Según la localización podemos clasificar los operandos de la siguiente manera:

- **Inmediato**. El dato está representado en la instrucción misma. Podemos considerar que se encuentra en un registro, el registro IR (*registro de instrucción*), y está directamente disponible para el procesador.

- **Registro**. El dato estará directamente disponible en un registro dentro del procesador.

- **Memoria**. El procesador deberá iniciar un ciclo de lectura/escritura a memoria.

En el caso de operaciones de E/S, habrá que solicitar el dato al módulo de E/S adecuado y para acceder a los registros del módulo de E/S según el mapa de E/S que tengamos definido.

Ved también

En el módulo "Sistemas de entrada/salida" veremos que a los registros del módulo de E/S, denominados *puertos de E/S*, podemos acceder como si fueran posiciones de memoria.

Según el lugar donde esté el dato, el procesador deberá hacer tareas diferentes con el fin de obtenerlo: puede ser necesario hacer cálculos y accesos a memoria indicados por el *modo de direccionamiento* que utiliza cada operando.

Ved también

Los modos de direccionamiento se estudian en el apartado 2 de este mismo módulo didáctico.

1.5.3. Tipo y tamaño de los operandos

Los operandos de las instrucciones sirven para expresar el lugar donde están los datos que hemos de utilizar. Estos datos se almacenan como una secuencia de bits y, según la interpretación de los valores almacenados, podemos tener tipos de datos diferentes que, generalmente, también nos determinarán el tamaño de los operandos. En muchos juegos de instrucciones, el tipo de dato que utiliza una instrucción viene determinado por el código de operación de la instrucción.

Hemos de tener presente que un mismo dato puede ser tratado como un valor lógico, como un valor numérico o como un carácter, según la operación que se haga con él, lo que no sucede con los lenguajes de alto nivel.

A continuación presentamos los tipos generales de datos más habituales:

1) Dirección. Tipo de dato que expresa una dirección de memoria. Para operar con este tipo de dato, puede ser necesario efectuar cálculos y accesos a memoria para obtener la dirección efectiva.

Ved también

Trataremos la manera de operar con las direcciones en el apartado 2 de este mismo módulo didáctico.

La manera de expresar y codificar las direcciones dependerá de la manera de acceder a la memoria del computador y de los modos de direccionamiento que soporte el juego de instrucciones.

2) Número. Tipo de dato que expresa un valor numérico. Habitualmente distinguimos tres tipos de datos numéricos (y cada uno de estos tipos se puede considerar con signo o sin signo):

a) Números enteros.

b) Números en punto fijo.
c) Números en punto flotante.

Como disponemos de un espacio limitado para expresar valores numéricos, tanto la magnitud de los números como su precisión también lo serán, lo que limitará el rango de valores que podemos representar.

Según el juego de instrucciones con el que se trabaja, en el código ensamblador, se pueden expresar los valores numéricos de maneras diferentes: en decimal, hexadecimal, binario, utilizando puntos o comas (si está en punto fijo) y con otras sintaxis para números en punto flotante; pero en código de máquina siempre los codificaremos en binario utilizando un tipo de representación diferente para cada tipo de valor. Cabe señalar, sin embargo, que uno de los más utilizados es el conocido como *complemento a 2*, que es una representación en punto fijo de valores con signo.

Ejemplo

Si tenemos 8 bits y el dato es un entero sin signo, el rango de representación será [0,255], y si el dato es un entero con signo, utilizando complemento a 2 el rango de representación será [-128,127].

Forma de expresarlo		Forma de codificarlo con 8 bits	
En decimal	En hexadecimal	Entero sin signo	Entero con signo en Ca2
12	0Ch	0000 1100	0000 1100
-33	DFh	No se puede (tiene signo)	1101 1111
150	96h	1001 0110	No se puede (fuera de rango)

3) Carácter. Tipo de dato que expresa un carácter. Habitualmente se utiliza para formar cadenas de caracteres que representarán un texto.

Aunque el programador representa los caracteres mediante las letras del alfabeto, para poder representarlos en un computador que utiliza datos binarios será necesaria una codificación. La codificación más habitual es la ASCII, que representa cada carácter con un código de 7 bits, lo que permite obtener 128 caracteres diferentes. Los códigos ASCII se almacenan y se transmiten utilizando 8 bits por carácter (1 byte), cuyo octavo bit tiene la función de paridad o control de errores.

El octavo bit

En los sistemas más actuales se utiliza el octavo bit para ampliar el juego de caracteres y disponer de símbolos adicionales, como letras que no existen en el alfabeto inglés (*ç*, *ñ*, etc.), letras con acento, diéresis, etc.

Hay otras codificaciones que se pueden utilizar sobre todo en lenguajes de alto nivel, como por ejemplo Unicode, que utiliza 16 bits y es muy interesante para dar soporte multilingüe.

4) Dato lógico. Tipo de dato que expresa un conjunto de valores binarios o booleanos; generalmente cada valor se utiliza como una unidad, pero puede ser interesante tratarlos como cadenas de *N* bits en las que cada elemento de la cadena es un valor booleano, un bit que puede valer 0 o 1. Este tratamiento es útil cuando se utilizan instrucciones lógicas como AND, OR o XOR.

Eso también permite almacenar en un solo byte hasta 8 valores booleanos que pueden representar diferentes datos, en lugar de utilizar más espacio para representar estos mismos valores.

1.6. Tipos de instrucciones

En general, los códigos de operación de un juego de instrucciones varían de una máquina a otra, pero podemos encontrar los mismos tipos de instrucciones en casi todas las arquitecturas y los podemos clasificar de la manera siguiente:

- Transferencia de datos
- Aritméticas
- Lógicas
- Transferencia del control
- Entrada/salida
- Otros tipos

1.6.1. Bits de resultado

Los bits de resultado nos dan información de cómo es el resultado obtenido en una operación realizada en la unidad aritmeticológica (ALU) del procesador. Al ejecutar una instrucción aritmética o lógica, la unidad aritmeticológica hace la operación y obtiene un resultado; según el resultado, activa los bits de resultado que corresponda y se almacenan en el registro de estado para poder ser utilizados por otras instrucciones. Los bits de resultado más habituales son los siguientes:

- **Bit de cero (Z)**: se activa si el resultado obtenido es 0.

- **Bit de transporte (C)**: también denominado *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, también se puede deber a una operación de desplazamiento. Se activa si al final de la operación nos llevamos una según el algoritmo de suma y resta tradicional operando en binario o si el último bit que desplazamos se copia sobre el bit de transporte y este es 1.

Nota

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

Bit de transporte

Cuando operamos con números enteros sin signo, el bit de transporte es equivalente al bit de desbordamiento; pero con número con signo (como es el caso del Ca2), el bit de

transporte y el bit de desbordamiento no son equivalentes y la información que aporta el bit de transporte sobre el resultado no es relevante.

- **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 que estamos utilizando, necesitaríamos más bits de los disponibles.

- **Bit de signo (S)**: activo si el resultado obtenido es negativo, el bit más significativo del resultado es 1.

Ejemplo

En este ejemplo se muestra cómo funcionan los bits de estado 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. En los subapartados siguientes también se explica de una manera más general el funcionamiento de estas instrucciones.

Consideramos 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 para cada instrucción es 0.

	R1 = 7, R2 = 1					**R1 = –1, R2 = –7**				
	Resultado	**Z**	**S**	**C**	**V**	**Resultado**	**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	0	0	R2= –6	0	1	1	0
NEG R1	R1= –7	0	0	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
SAL R2,1	R2= +2	0	0	0	0	R2= +2	0	0	1	1
SAR R2,1	R2= 0	1	0	1	0	R2= –4	0	1	1	0

Tabla

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
–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 sumamos 1, 1010 (–6).
- (–3) 1101, lo negamos, 0010 y sumamos 1, 0011 (+3).
- (+0) 0000, lo negamos, 1111 y sumamos 1, 0000 (–0). Queda igual.
- (-8) 1000, lo negamos, 0111 y 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.

Bit de transporte en la resta

El bit de transporte, también denominado *borrow*, se genera según el algoritmo convencional de la resta. Pero si para hacer la resta A – B, lo hacemos 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, donde tanto el *carry* como el *borrow* son iguales y valen 0.

1.6.2. Instrucciones de transferencia de datos

Las instrucciones de transferencia de datos transfieren un dato de una localización a otra dentro del computador.

El tipo y el tamaño de los datos pueden estar especificados de manera implícita en el código de operación. Eso hará que tengamos códigos de operación diferentes según el tipo y el tamaño de los datos que se deben transferir y para indicar implícitamente la localización del operando fuente o destino.

Tamaño de los operandos

Normalmente, cada dirección de memoria se corresponde con una posición de memoria de tamaño 1 byte. Si hacemos STORW [1000], R1, y R1 es un registro de 16 bits, en realidad estamos utilizando dos direcciones de memoria, la 1000 y la 1001, ya que el dato ocupa 2 bytes en R1.

STORx: Indica que transferimos un dato hacia la memoria.
STORB [1000], 020h: Indica que transferimos 1 byte a la posición de memoria 1000.
STORW [1000], R1: Indica que transferimos 2 bytes (1 *word*), el contenido del registro R1, a las posiciones de memoria 1000 y 1001.

En las arquitecturas de tipo registro-registro se utilizan las instrucciones LOAD y STORE cuando hay que efectuar transferencias de datos con la memoria.

Ejemplo de instrucciones de transferencia de datos en la arquitectura MIPS

Instrucción	Ejemplo	Operación
LOAD destino, fuente	LW $s1,100($s2)	Mueve el contenido de la posición de memoria M($s2+100) al registro $s1
STORE fuente, destino	SW $s1,100($s2)	Mueve el contenido del registro $s1 a la posición de memoria M($s2+100)
MOVE destino, fuente	MOVE $s1,$s2	Mueve el contenido del registro $s2 al registro $s1

En las arquitecturas de tipo registro-memoria o memoria-memoria se utilizan instrucciones MOV cuando hay que llevar a cabo transferencias de datos con la memoria. Es el caso de las arquitecturas Intel x86-64 y CISCA.

Ejemplo de instrucciones de transferencia de datos en la arquitectura Intel x86-64

Instrucción	Operación	Ejemplo
MOV destino, fuente	destino ← fuente	MOV RAX, [num_var]
XCHG destino, fuente	destino ← → fuente Intercambia el contenido de los dos operandos.	XCHG [num_var], RBX
POP destino	destino ← M(SP), actualiza SP. Saca el valor que hay en la cima de la pila.	POP RAX

Ejemplo de instrucciones de transferencia de datos en la arquitectura CISCA

Instrucción	Operación	Ejemplo
MOV destino, fuente	destino ← fuente	MOV R1, [nom_var]
PUSH fuente	Actualiza SP, M(SP) ← fuente. Pone el valor en la cima de la pila.	PUSH R5

1.6.3. Instrucciones aritméticas

Todas las máquinas incluyen instrucciones aritméticas básicas (suma, resta, multiplicación y división) y otras como negar, incrementar, decrementar o comparar.

Cuando hacemos operaciones aritméticas, hemos de tener presentes los tipos de operandos y si los debemos considerar números con signo o sin signo, ya que en algunas operaciones, como la multiplicación o la división, eso puede dar lugar a instrucciones diferentes.

En este subapartado nos centraremos solo en las operaciones aritméticas de números enteros con signo y sin signo. El formato habitual para representar números enteros con signo es el complemento a 2.

Para la representación de números decimales en punto fijo se puede utilizar la misma representación que para los enteros haciendo que la coma binaria esté implícita en alguna posición del número. Para la representación en punto flotante será necesario especificar un formato diferente para representar los números y las instrucciones específicas para operar con ellos.

Suma y resta

Estas operaciones hacen la suma y la resta de dos números; en algunos procesadores se pueden hacer teniendo en cuenta el valor del bit de transporte como bit de transporte inicial.

Ejemplo de suma y resta en la arquitectura CISCA

Instrucción	Operación	Ejemplo
ADD destino, fuente	destino = destino + fuente	ADD R1, R2
SUB destino, fuente	destino = destino – fuente	SUB R1, R2

Ejemplo de suma y resta en la arquitectura Intel x86-64 considerando el bit de transporte inicial

Instrucción	Operación	Ejemplo
ADC destino, fuente	destino = destino + fuente + bit de transporte	ADC RAX, RBX
SBB destino, fuente	destino = destino – fuente – bit de transporte	SBB RAX, RBX

Multiplicación

Esta operación efectúa la multiplicación entera de dos números y hay que tener presente que el resultado que se genera tiene un tamaño superior al de los operandos fuente.

Habrá que disponer de dos operaciones diferentes si se quiere tratar operandos con signo o sin signo.

Multiplicación y operandos

En la mayoría de las arquitecturas, la operación multiplicación utiliza operandos destino implícitos; de esta manera es más fácil representar un resultado de tamaño superior al de los operandos fuente.

Ejemplo de una instrucción de multiplicación en la arquitectura Intel x86-64

Un operando fuente puede ser implícito y corresponder al registro AL (8 bits), al registro AX (16 bits), al registro EAX (32 bits) o al registro RAX (64 bits); el otro operando fuente es explícito y puede ser un operando de 8, 16, 32 o 64 bits.

- Si el operando fuente es de 8 bits, la operación que se hace es AX = AL * fuente, que amplía el resultado a un valor de 16 bits.
- Si el operando fuente es de 16 bits, la operación que se hace es DX,AX = AX * fuente, que amplía el resultado a un valor de 32 bits.
- Si el operando fuente es de 32 bits, la operación que se hace es EDX,EAX = EAX * fuente, que amplía el resultado a un valor de 64 bits.
- Si el operando fuente es de 64 bits, la operación que se hace es RDX,RAX = RAX * fuente, que amplía el resultado a un valor de 128 bits.

Instrucción	Ejemplo	Operación
MUL fuente (operación sin considerar el signo, un operando implícito)	MUL RBX	RDX,RAX = RAX * RBX
IMUL fuente (operación considerando el signo, un operando implícito)	IMUL BL	AX = AL * BL
IMUL destino, fuente (operación considerando el signo, dos operandos explícitos)	IMUL EAX, EBX	EAX = EAX * EBX

Ejemplo de una instrucción de multiplicación en la arquitectura CISCA

Los dos operandos fuente y destino son explícitos, y el resultado se almacena en el primer operando, que hace también de operando destino; todos los operandos son números en complemento a 2 de 32 bits; no se utiliza ningún operando implícito porque no se amplía el tamaño del resultado.

Instrucción	Operación	Ejemplo
MUL destino, fuente	destino = destino * fuente	MUL R1, R2

División

Esta operación hace la división entera de dos números y se obtienen dos resultados: el cociente y el residuo de la división.

Habrá que disponer de dos operaciones diferentes si se quiere tratar operandos con signo o sin signo.

División y operandos

En la mayoría de las arquitecturas, la operación división utiliza operandos destino implícitos para representar los dos resultados que se obtienen y no perder los valores de los operandos fuente.

Ejemplo de una instrucción de división en la arquitectura Intel x86-64

Divide el dividendo implícito entre el divisor explícito.

- Operando fuente de 8 bits: (cociente) AL = AX / fuente, (residuo) AH = AX mod fuente
- Operando fuente de 16 bits: (cociente) AX = DX:AX / fuente, (residuo) DX = DX:AX mod fuente
- Operando fuente de 32 bits: (cociente) EAX = EDX:EAX / fuente, (residuo) EDX = EDX:EAX mod fuente
- Operando fuente de 64 bits: (cociente) RAX = RDX:RAX / fuente, (residuo) RDX = RDX:RAX mod fuente

Instrucción	Ejemplo	Operación
DIV fuente (operación sin considerar el signo)	DIV RBX	RAX = RDX:RAX / RBX RDX = RDX:RAX mod RBX
IDIV fuente (operación considerando el signo)	IDIV EBX	EAX = EDX:EAX / EBX EDX = EDX:EAX mod EBX

Ejemplo de una instrucción de división en la arquitectura CISCA

Los dos operandos fuente y destino son explícitos y son números con signo. El cociente se almacena en el primer operando y el residuo, en el segundo. Los dos operandos hacen de operando fuente y destino.

Instrucción	Ejemplo	Operación
DIV destino, fuente DIV (dividendo/cociente), (divisor/residuo)	DIV R1, R2	R1 = R1 / R2 (cociente) R2 = R1 mod R2 (residuo)

Incremento y decremento

Estas operaciones son un caso especial de la suma y la resta. Se suelen incluir en la mayoría de los juegos de instrucciones, ya que son operaciones muy frecuentes: saber el valor (generalmente, 1) que se debe incrementar o decrementar facilita mucho su implementación en el procesador.

Ejemplo de instrucciones de incremento y decremento en la arquitectura Intel x86-64

Instrucción	Operación	Ejemplo
INC destino	destino = destino + 1	INC RAX
DEC destino	destino = destino – 1	DEC BL

Ejemplo de instrucciones de incremento y decremento en la arquitectura CISCA

Instrucción	Operación	Ejemplo
INC destino	destino = destino + 1	INC R3
DEC destino	destino = destino – 1	DEC R5

Comparación

Esta operación lleva a cabo la comparación entre dos operandos restando el segundo del primero y actualizando los bits de resultado. Es un caso especial de la resta en el que el valor de los operandos no se modifica porque no se guarda el resultado.

Ejemplo de instrucciones de comparación en la arquitectura Intel x86-64

Instrucción	Operación	Ejemplo
CMP destino, fuente	destino – fuente	CMP RAX, RBX

Ejemplo de comparación en la arquitectura CISCA

Instrucción	Operación	Ejemplo
CMP destino, fuente	destino – fuente	CMP R1,R2

Negación

Esta operación cambia el signo del operando. Si los datos están en complemento a 2, es equivalente a hacer una de estas operaciones: 0 – operando, NOT(operando) + 1, operando * (–1).

Ejemplo de una instrucción de negación en la arquitectura Intel x86-64

Instrucción	Operación	Ejemplo
NEG destino	destino = NOT(destino) + 1	NEG EAX

Ejemplo de una instrucción de negación en la arquitectura CISCA

Instrucción	Operación	Ejemplo
NEG destino	destino = NOT(destino) + 1	NEG R7

Todas las instrucciones aritméticas y de comparación pueden modificar los bits de resultado.

Instrucción	Z	S	C	V
Suma	x	x	x	x
Resta	x	x	x	x
Multiplicación	x	x	-	x
División	x	x	-	x
Incremento	x	x	x	x
Decremento	x	x	x	x

Estos son los bits de resultado que se modifican habitualmente, pero eso puede variar ligeramente en algunos procesadores.
Nota: x indica que la instrucción *puede modificar este bit*. – indica que la instrucción *no modifica este bit*.

Instrucción	Z	S	C	V
Comparación	x	x	x	x
Negación	x	x	x	x

Estos son los bits de resultado que se modifican habitualmente, pero eso puede variar ligeramente en algunos procesadores.
Nota: x indica que la instrucción *puede modificar este bit*. – indica que la instrucción *no modifica este bit*.

1.6.4. Instrucciones lógicas

Hay varias instrucciones lógicas:

1) Operaciones lógicas. Las instrucciones que hacen operaciones lógicas permiten manipular de manera individual los bits de un operando. Las operaciones lógicas más habituales son AND, OR, XOR, NOT.

Las instrucciones lógicas hacen la operación indicada bit a bit; es decir, el resultado que se produce en un bit no afecta al resto.

x	y	x AND y	x OR y	x XOR y	NOT x
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

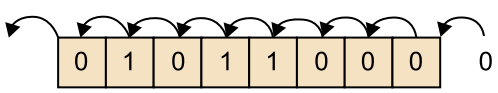
2) Operaciones de desplazamiento y rotación. Dentro de este grupo de instrucciones se incluyen instrucciones de desplazamiento lógico (SHL, SHR), desplazamiento aritmético (SAL, SAR), rotación (ROL, ROR).

Este grupo de instrucciones incluye un operando con el que se hace la operación y, opcionalmente, un segundo operando que indica cuántos bits se deben desplazar/rotar.

Ahora veremos el funcionamiento de las instrucciones lógicas de desplazamiento y rotación:

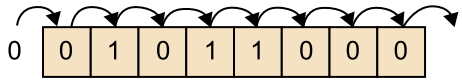
a) Desplazamiento lógico a la izquierda (SHL). Se considera el primer operando como un valor sin signo. Se desplazan los bits a la izquierda tantas posiciones como indica el segundo operando; el bit de más a la izquierda en determinadas arquitecturas se pierde y en otras se copia sobre el bit de transporte y se añaden ceros por la derecha.

Ejemplo de desplazamiento de un bit a la izquierda



b) Desplazamiento lógico a la derecha (SHR). Se considera el primer operando como un valor sin signo; se desplazan los bits a la derecha tantas posiciones como indica el segundo operando, el bit de la derecha en determinadas arquitecturas se pierde y en otras se copia sobre el bit de transporte y se añaden ceros por la izquierda.

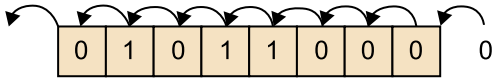
Ejemplo de desplazamiento de un bit a la derecha



c) Desplazamiento aritmético a la izquierda (SAL). Se considera el primer operando como un valor con signo expresado en complemento a 2; se desplazan los bits a la izquierda tantas posiciones como indica el segundo operando, el bit de más a la izquierda en determinadas arquitecturas se pierde y en otras se copia sobre el bit de transporte y se añaden ceros por la derecha. Consideraremos que hay desbordamiento si el signo del resultado es diferente del signo del valor que desplazamos.

Físicamente, es la misma instrucción que la instrucción de desplazamiento lógico a la izquierda.

Ejemplo de desplazamiento de un bit a la izquierda

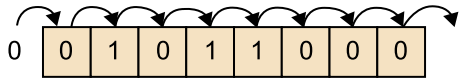


Nota

La operación de desplazamiento aritmético a la izquierda n posiciones es equivalente a multiplicar el valor del primer operando por 2^n.

d) Desplazamiento aritmético a la derecha (SAR). Se considera el primer operando como un valor con signo expresado en complemento a 2. El bit de más a la izquierda, bit de signo, se conserva y se va copiando sobre los bits que se van desplazando a la derecha. Se desplaza a la derecha tantas posiciones como indica el segundo operando y los bits de la derecha en determinadas arquitecturas se pierden y en otras se copian sobre el bit de transporte.

Ejemplo de desplazamiento de un bit a la derecha

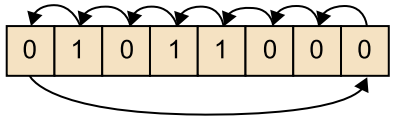


Nota

La operación de desplazamiento aritmético a la derecha n posiciones es equivalente a dividir el valor del primer operando entre 2^n.

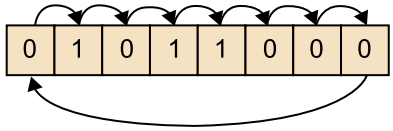
e) Rotación a la izquierda (ROL). Se considera el primer operando como un valor sin signo; se desplazan los bits a la izquierda tantas posiciones como indica el segundo operando, el bit de más a la izquierda se pasa a la posición menos significativa del operando.

Ejemplo de rotación de un bit a la izquierda



f) Rotación a la derecha (ROR). Se considera el primer operando como un valor sin signo; se desplazan los bits a la derecha tantas posiciones como indica el segundo operando, el bit de más a la derecha se pasa a la posición más significativa del operando.

Ejemplo de rotación de un bit a la derecha



Hay instrucciones de rotación que utilizan el bit de transporte:

- Cuando se rota a la derecha, el bit menos significativo del operando se copia sobre el bit de transporte y el bit de transporte original se copia sobre el bit más significativo del operando.
- Cuando se rota a la izquierda, el bit más significativo del operando se copia sobre el bit de transporte y el bit de transporte se copia sobre el bit menos significativo.

Las instrucciones lógicas también pueden modificar los bits de resultado del procesador.

Instrucción	Z	S	C	V
AND	x	x	0	0
OR	x	x	0	0
XOR	x	x	0	0
NOT	–	–	–	–
Desplazamientos lógicos	x	x	x	0
Desplazamientos aritméticos	x	x	x	x
Rotaciones	–	–	x	x

Nota: x indica que la instrucción *puede modificar el bit de resultado*. – indica que la instrucción *no modifica el bit de resultado*. 0 indica que la instrucción *pone el bit de resultado a 0*.

Ejemplo de instrucciones lógicas de desplazamiento y rotación en la arquitectura Intel x86-64

Instrucción	Operación	Ejemplo
AND destino, fuente	destino = fuente AND destino	AND AL,BL
OR destino, fuente	destino = fuente OR destino	OR RAX,RBX

Instrucción	Operación	Ejemplo
SHL destino, fuente	destino = destino * 2^{fuente} (considerando destino como un valor sin signo)	SHL AL, 4
SHR destino, fuente	destino = destino / 2^{fuente} (considerando destino como un valor sin signo)	SHR RAX, CL
RCL destino, fuente	(CF = bit más significativo de destino, destino = destino * 2 + $CF_{original}$) haremos eso tantas veces como indique el operando fuente.	RCL EAX, CL
RCR destino, fuente	(CF = bit menos significativo de destino, destino = destino / 2 + $CF_{original}$) haremos eso tantas veces como indique el operando fuente.	RCR RBX, 1

Ejemplo de instrucciones lógicas en la arquitectura CISCA

Instrucción	Operación	Ejemplo
XOR destino, fuente	destino = fuente AND destino	XOR R2,R7
NOT destino	destino negado bit a bit	NOT R3
SAL destino, fuente	destino = destino * 2^{fuente}	SAL R9, 3
SAR destino, fuente	destino = destino / 2^{fuente}	SAR R9, 2

1.6.5. Instrucciones de ruptura de secuencia

Las instrucciones de ruptura de secuencia permiten cambiar la secuencia de ejecución de un programa. En el ciclo de ejecución de las instrucciones, el PC se actualiza de manera automática apuntando a la instrucción almacenada a continuación de la que se está ejecutando; para poder cambiarla, hay que saber cuál es la siguiente instrucción que se debe ejecutar para cargar la dirección en el PC.

Entre las instrucciones de ruptura de secuencia encontramos las siguientes:

- Instrucciones de salto o bifurcación.
  - Instrucciones de salto incondicional.
  - Instrucciones de salto condicional.

- Instrucciones de llamada y retorno de subrutina.
- Instrucciones de interrupción de software y retorno de una rutina de servicio de interrupción.

Instrucciones de salto incondicional

Las instrucciones de salto incondicional cargan en el registro del PC la dirección especificada por el operando, que se expresa como una etiqueta que representa la dirección de la instrucción que se debe ejecutar.

Si se codifica el operando como una dirección, actualizaremos el registro PC con esta dirección. Si se codifica el operando como un desplazamiento, deberemos sumar al PC este desplazamiento y, como veremos más adelante, esta manera de obtener la dirección de salto se denomina *direccionamiento relativo a PC*.

Ejemplo de instrucciones de ruptura de secuencia en la arquitectura CISCA

Instrucción	Operación	Ejemplo
JMP etiqueta	[PC] ← etiqueta Salta a la instrucción indicada por la etiqueta	JMP bucle

```
bucle: MOV R0, R1 ; Esto es un bucle infinito
          JMP bucle
          MOV R2, 0 ; No se ejecuta nunca
```

bucle es el nombre de la etiqueta que hace referencia a la dirección de memoria donde se almacena la instrucción MOV R0, R1.

Instrucciones de salto condicional

Las instrucciones de salto condicional cargan en el registro PC la dirección especificada por el operando si se cumple una condición determinada (la condición es cierta); en caso contrario (la condición es falsa), el proceso continúa con la instrucción siguiente de la secuencia. Este operando se expresa como una etiqueta que representa la dirección de la instrucción que se debe ejecutar, pero habitualmente se codifica como un desplazamiento respecto al registro PC. Esta manera de expresar una dirección de memoria, como veremos más adelante, se denomina *direccionamiento relativo a PC*.

Las condiciones se evalúan comprobando el valor actual de los bits de resultado; los más habituales son cero, signo, transporte y desbordamiento. Una condición puede evaluar un bit de resultado o más.

Las instrucciones de salto condicional se han de utilizar inmediatamente después de una instrucción que modifique los bits de resultado, como las aritméticas o lógicas; en caso contrario, se evaluarán los bits de resultado de la última instrucción que los haya modificado, situación nada recomendable.

La tabla siguiente muestra los bits de resultado que hay que comprobar para evaluar una condición determinada después de una instrucción de comparación (CMP). Estos bits de resultado se activan según los valores de los operandos comparados.

Condición	Bits que comprueba con operandos sin signo	Bits que comprueba con operandos con signo
A = B	Z = 1	Z = 1
A ≠ B	Z = 0	Z = 0
A > B	C = 0 y Z = 0	Z = 0 y S = V
A ≥ B	C = 0	S = V
A < B	C = 1	S ¹ V
A ≤ B	C = 1 o Z = 1	Z = 1 o S¹ V

Nota

Hay que tener presente que se activará un conjunto de bits diferente según si los operandos comparados se consideran con signo o sin él.

Ejemplo de instrucciones de salto condicional comunes a la arquitectura Intel x86-64 y CISCA

Instrucción	Operación	Ejemplo
JE etiqueta (Jump Equal – Salta si igual)	Salta si Z = 1	JE eti3
JNE etiqueta (Jump Not Equal – Salta si diferente)	Salta si Z = 0	JNE eti3
JG etiqueta (Jump Greater – Salta si mayor)	Salta si Z = 0 y S = V (operandos con signo)	JG eti3
JGE etiqueta (Jump Greater or Equal – Salta si mayor o igual)	Salta si S = V (operandos con signo)	JGE eti3
JL etiqueta (Jump Less – Salta si más pequeño)	Salta si S≠ V (operandos con signo)	JL eti3
JLE etiqueta (Jump Less or Equal – Salta si más pequeño o igual)	Salta si Z = 1 o S¹ V (operandos con signo)	JLE eti3

```
      MOV R2, 0
eti3: INC R2 ; Esto es un bucle de 3 iteraciones
      CMP R2, 3
      JL eti3
   MOV R2, 7
```

Si consideramos que la instrucción INC R2 está en la posición de memoria 0000 1000h, la etiqueta eti3 hará referencia a esta dirección de memoria, pero cuando codificamos la instrucción JL eti3, no codificaremos esta dirección, sino que codificaremos el desplazamiento respecto al PC. El desplazamiento que hemos de codificar es eti3 – PC.

Ejemplo de instrucciones de salto condicional específicas de la arquitectura Intel x86-64

Instrucción	Operación	Ejemplo
JA etiqueta (Jump Above – Salta si superior)	Salta si C = 0 y Z = 0 (operandos sin signo)	JA bucle
JAE etiqueta (Jump Above or Equal – Salta si superior o igual)	Salta si C = 0 (operandos sin signo)	JAE bucle
JB etiqueta (Jump Below – Salta si inferior)	Salta si C = 1 (operandos sin signo)	JB bucle
JBE etiqueta (Jump Below or Equal – Salta si inferior o igual)	Salta si C = 1 o Z = 1 (operandos sin signo)	JBE bucle

Instrucción	Operación	Ejemplo
JV etiqueta	Salta si V = 1	JV bucle

Instrucciones de salto implícito

Hay un tipo de instrucciones de salto denominadas *skip* o *de salto implícito* que según una condición saltan una instrucción. Si no se cumple la condición, ejecutan la instrucción siguiente y si se cumple la condición, no ejecutan la instrucción siguiente, se la saltan, y ejecutan la instrucción que hay a continuación. En algunos casos también tienen una operación asociada (como incrementar o decrementar un registro). Estas instrucciones las suelen tener computadores muy simples o microcontroladores con un juego de instrucciones reducido y con instrucciones de tamaño fijo.

Por ejemplo, tenemos la instrucción *DSZ Registro* que decrementa *Registro* y salta si es cero.

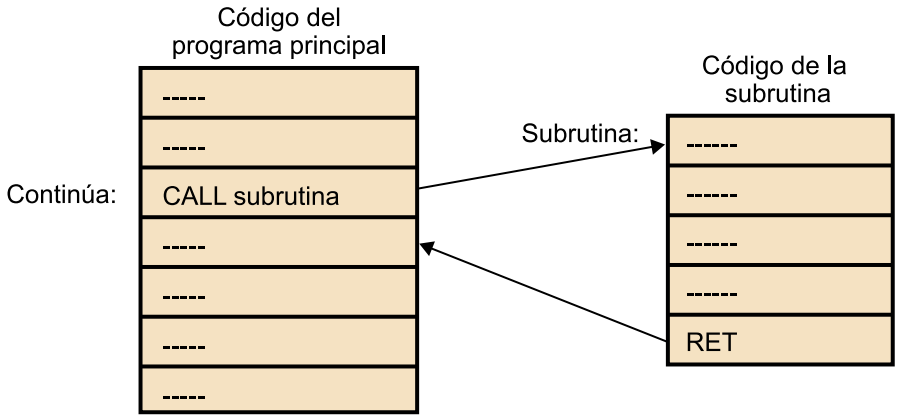
```
iteracion:
        ...
        DSZ R1
        JMP iteración
continuar:
        ...
```

Si R1 = 1 (al decrementar valdrá 0), irá a la etiqueta *continuar:*, si no, ejecuta el JMP (salto incondicional) y salta a la etiqueta *iteracion:*.

Instrucciones de llamada y retorno de subrutina

Una subrutina es un conjunto de instrucciones que hace una función concreta y al que normalmente se llama varias veces dentro del programa principal.

Tanto las instrucciones de salto como las de llamada y retorno a subrutina permiten romper la secuencia de ejecución de un programa, pero las últimas garantizan la vuelta al punto donde se ha roto la secuencia una vez ha finalizado la ejecución.



La **llamada a subrutina** es una instrucción que transfiere el control a la subrutina de manera que se pueda garantizar el retorno al punto donde se encontraba la secuencia de ejecución al llamarla.

La instrucción tiene un único operando que especifica la dirección de inicio de la subrutina. Este operando se expresa como una etiqueta que representa la dirección de la primera instrucción de la subrutina que se debe ejecutar.

Las llamadas a subrutinas llevan a cabo dos operaciones:

1) Guardar el valor del PC en una localización conocida para que cuando finalice la ejecución de la subrutina pueda retornar al punto de ejecución donde se encontraba y continuar la secuencia de ejecución. Las localizaciones donde se puede almacenar el PC (dirección de retorno de la subrutina) son un registro, al principio de la subrutina o a la pila. En la mayoría de los computadores se utiliza la pila para almacenar la dirección de retorno.

2) Cargar en el PC la dirección expresada por el operando de la instrucción para transferir el control a la subrutina.

El **retorno de subrutina** es una instrucción que se utiliza para devolver el control al punto de ejecución desde donde se ha hecho la llamada. Para hacerlo, recupera el valor del PC del lugar donde lo haya almacenado la instrucción de llamada a subrutina. Tiene que ser la última instrucción de una subrutina y no tiene ningún operando explícito.

Es necesario que el programador se asegure de que, antes de ejecutar el retorno de subrutina, en la cima de la pila esté la dirección de retorno. Por ello es muy importante hacer una buena gestión de la pila dentro de las subrutinas.

Ejemplo de instrucciones de llamada y retorno de subrutina comunes a la arquitectura Intel x86-64 y CISCA

Instrucción	Operación	Ejemplo
CALL etiqueta	Transfiere el control a la subrutina.	CALL subrutina
RET	Retorna de la subrutina	RET

Ejemplo de llamada y un retorno de subrutina en la arquitectura CISCA

En esta arquitectura, la pila se implementa a memoria y crece hacia direcciones bajas, por lo tanto, en la instrucción CALL se decrementa SP (R15 en esta arquitectura) y en la instrucción RET se incrementa. El incremento y el decremento que habrá que hacer del registro SP es de 4 unidades, ya que cada posición de la pila almacena 1 byte y la dirección ocupa 4.

```
CALL 'nom_subrutina'
    SP = SP – 4
    M(SP) = PC
    PC = Dirección inicio 'nombre-subrutina'

RET
    PC = M(SP)
    SP = SP + 4
```

Instrucciones de interrupción de software y retorno de una rutina de servicio de interrupción

La **interrupción de software** es un tipo de instrucción que implementa una interrupción llamando a una rutina de servicio de interrupción (RSI). Estos servicios generalmente son rutinas del sistema operativo para facilitar al programador las operaciones de E/S y de acceso a diferentes elementos del hardware.

Este tipo de instrucción tiene un solo operando que especifica un valor inmediato que identifica el servicio solicitado; en los sistemas con las interrupciones vectorizadas, este valor especifica un índice dentro de la tabla de vectores de interrupción gestionada por el sistema operativo.

La ejecución de una instrucción de software lleva a cabo dos operaciones:

1) Guardar el valor del registro de estado y del PC en la pila del sistema para que cuando finalice la ejecución de la RSI pueda retornar al punto de ejecución donde se encontraba y continuar la secuencia de ejecución.

2) Cargar en el PC la dirección donde se encuentra el código de la RSI. Si es un sistema con las interrupciones vectorizadas, la dirección la obtendremos de la tabla de vectores de interrupción.

Ejemplo de instrucción de interrupción de software en la arquitectura Intel x86-64

Instrucción	Operación	Ejemplo
INT servicio	Llamada al sistema	INT 80 h

Ejemplo de instrucción de interrupción de software en la arquitectura CISCA

Instrucción	Operación	Ejemplo
INT servicio	Llamada al sistema	INT 1 h

Respecto al **retorno de una rutina de servicio de interrupción**, se debe tener en cuenta que antes de finalizar la ejecución de una rutina de servicio de interrupción (RSI), hemos de restaurar el estado del procesador para devolver el control al programa que se estaba ejecutando, por ello hay que recuperar los registros de estado y el PC guardados en la pila del sistema y poder reanudar la ejecución del programa que habíamos detenido.

Ejemplo de instrucciones de llamada y retorno de subrutina comunes a la arquitectura Intel x86-64 y CISCA

Instrucción	Operación	Ejemplo
IRET	Retorno de una rutina de servicio de interrupción	IRET

1.6.6. Instrucciones de entrada/salida

Las instrucciones de entrada/salida permiten leer y escribir en un puerto de E/S. Los puertos de E/S se utilizan para acceder a un registro del módulo de E/S que controla un dispositivo o más, para consultar su estado o programarlo.

Según la arquitectura del computador, podemos tener un mapa común de memoria y E/S o un mapa independiente de E/S:

- **Mapa común de memoria y E/S**: se utilizan las mismas instrucciones de transferencia de datos explicados anteriormente.

- **Mapa independiente de E/S**: las instrucciones utilizadas para acceder a los puertos de E/S son diferentes de las de transferencia de datos. Las más habituales son:
  - **IN**. Permite leer de un puerto de E/S. Utiliza dos operandos: un número de puerto y una localización para almacenar el valor leído (normalmente un registro).
  - **OUT**. Permite escribir en un puerto de E/S. Utiliza dos operandos: un número de puerto y la localización del valor que se debe escribir en el puerto, habitualmente este valor se encuentra en un registro.

Ejemplos de mapa común

LOAD, STORE y MOV son instrucciones que se utilizan en el mapa común de memoria y E/S.

Ejemplo de instrucciones de entrada/salida en la arquitectura Intel x86-64

Instrucción	Operación	Ejemplo
IN destino, fuente	destino ← puerto de E/S indicado por el operando fuente	IN AX, 20h
OUT destino, fuente	puerto de E/S indicado por el operando destino ← fuente	OUT 20h, EAX

Ejemplo de instrucciones de entrada/salida en la arquitectura CISCA

Instrucción	Operación	Ejemplo
IN destino, fuente	destino ← puerto de E/S indicado por el operando fuente	IN R5, 01h
OUT destino, fuente	puerto de E/S indicado por el operando destino ← fuente	OUT 02h, R8

1.6.7. Otros tipos de instrucciones

En un juego de instrucciones suele haber otros tipos de instrucciones de carácter más específico; algunas pueden estar restringidas al sistema operativo para ser ejecutado en modo privilegiado.

Las instrucciones de control del sistema, normalmente, son instrucciones privilegiadas que en algunas máquinas se pueden ejecutar solo cuando el procesador se encuentra en estado privilegiado, como por ejemplo HALT (detiene la ejecución del programa), WAIT (espera por algún acontecimiento para sincronizar el procesador con otras unidades) y otras para gestionar el sistema de memoria virtual o acceder a registros de control no visibles al programador.

Instrucciones específicas para trabajar con números en punto flotante (juego de instrucciones de la FPU) o para cuestiones multimedia, muchas de tipo SIMD (*single instruction multiple data*).

U otras, como, por ejemplo, la instrucción NOP (no operación), que no hace nada. Aunque el abanico puede ser muy amplio.

1.7. Diseño del juego de instrucciones

El juego de instrucciones de una arquitectura es la herramienta que tiene el programador para controlar el computador; por lo tanto, debería facilitar y simplificar la tarea del programador. Por otra parte, las características del juego de instrucciones condicionan el funcionamiento del procesador y, por ende, tienen un efecto significativo en el diseño del procesador. Así, nos encontramos con el problema de que muchas de las características que facilitan la tarea al programador dificultan el diseño del procesador; por este motivo, es necesario llegar a un compromiso entre facilitar el diseño del computador y satisfacer las necesidades del programador. Una manera de alcanzar este compromiso es diseñar un juego de instrucciones siguiendo un criterio de ortogonalidad.

La **ortogonalidad** consiste en que todas las instrucciones permitan utilizar como operando cualquiera de los tipos de datos existentes y cualquiera de los modos de direccionamiento, y en el hecho de que la información del código de operación se limite a la operación que debe hacer la instrucción.

Si el juego de instrucciones es ortogonal, será regular y no presentará casos especiales, lo que facilitará la tarea del programador y la construcción de compiladores.

Los aspectos más importantes que hay que tener en cuenta para diseñar un juego de instrucciones son los siguientes:

- **Conjunto de operaciones**: identificación de las operaciones que hay que llevar a cabo y su complejidad.
- **Tipos de datos**: identificación de los tipos de datos necesarios para llevar a cabo las operaciones.
- **Registros**: identificación del número de registros del procesador.
- **Direccionamiento**: identificación de los modos de direccionamiento que se pueden utilizar en los operandos.
- **Formato de las instrucciones**: longitud y número de operandos, y tamaño de los diferentes campos.

Diseño de un juego de instrucciones

A causa de la fuerte interrelación entre los distintos aspectos que hay que tener presentes para diseñar un juego de instrucciones, diseñarlo se convierte en una tarea muy compleja y no es el objetivo de esta asignatura abordar esta problemática.

2. Modos de direccionamiento

Los operandos de una instrucción pueden expresar directamente un dato, la dirección, o la referencia a la dirección, donde tenemos el dato. Esta dirección puede ser la de un registro o la de una posición de memoria, y en este último caso la denominaremos *dirección efectiva*.

Entendemos por **modo de direccionamiento** las diferentes maneras de expresar un operando en una instrucción y el procedimiento asociado que permite obtener la dirección donde está almacenado el dato y, como consecuencia, el dato.

Memoria virtual

Si tenemos un sistema con memoria virtual, la dirección efectiva será una dirección virtual y la correspondencia con la dirección física dependerá del sistema de paginación y será transparente al programador.

Los juegos de instrucciones ofrecen maneras diferentes de expresar los operandos por medio de sus modos de direccionamiento, que serán un compromiso entre diferentes factores:

- El rango de las direcciones que hemos de alcanzar para poder acceder a todo el espacio de memoria dirigible con respecto a programación.
- Con respecto al espacio para expresar el operando, se intentará reducir el número de bits que hay que utilizar para codificar el operando y, en general, las diferentes partes de la instrucción para poder construir programas que ocupen menos memoria.
- Las estructuras de datos en las que se pretenda facilitar el acceso o la manera de operar, como listas, vectores, tablas, matrices o colas.
- La complejidad de cálculo de la dirección efectiva que expresa el operando para agilizar la ejecución de las instrucciones.

En este apartado analizaremos los modos de direccionamiento más comunes, que se recogen en el esquema siguiente:

Clasificación de modos de direccionamiento

La clasificación de modos de direccionamiento que presentamos aquí se ha efectuado a partir de los más utilizados en los juegos de instrucciones de máquinas reales, pero en muchos casos estas utilizan variantes de los modos de direccionamiento explicados o los expresan de manera diferente.

Hay que consultar el manual de referencia del juego de instrucciones de cada máquina para saber qué modos de direccionamiento se pueden utilizar en cada operando de cada instrucción y su sintaxis.

- Direccionamiento inmediato
- Direccionamiento directo:
  - Direccionamiento directo a registro
  - Direccionamiento directo a memoria
- Direccionamiento indirecto:
  - Direccionamiento indirecto a registro
  - Direccionamiento indirecto a memoria
- Direccionamiento relativo:
  - Direccionamiento relativo a registro base o relativo
  - Direccionamiento relativo a registro índice (RI) o indexado

  - Direccionamiento relativo a PC

- Direccionamiento implícito
  - Direccionamiento a pila (indirecto a registro SP)

Hay que tener presente que cada operando de la instrucción puede tener su propio modo de direccionamiento, y no todos los modos de direccionamiento de los que dispone un juego de instrucciones se pueden utilizar en todos los operandos ni en todas las instrucciones.

Existe una cuestión, a pesar de ser transparente al programador, que hay que conocer y tener presente porque indirectamente sí que le puede afectar a la hora de acceder a la memoria. Se trata de la ordenación de los bytes de un dato cuando este tiene un tamaño superior al tamaño de la palabra de memoria.

En la mayoría de los computadores la memoria se dirige en bytes, es decir, el tamaño de la palabra de memoria es de un byte. Cuando trabajamos con un dato formado por varios bytes habrá que decidir cómo se almacena el dato dentro de la memoria, es decir, qué byte del dato se almacena en cada posición de la memoria.

Se pueden utilizar dos sistemas diferentes:

- **little-endian**: almacenar el byte de menos peso del dato en la dirección de memoria más baja.

- **big-endian**: almacenar el byte de más peso del dato en la dirección de memoria más baja.

Una vez elegido uno de estos sistemas, habrá que tenerlo presente y utilizarlo en todos los accesos a memoria (lecturas y escrituras) para asegurar la coherencia de los datos.

Ejemplo

Supongamos que queremos almacenar el valor hexadecimal siguiente: 12345678h. Se trata de un valor de 32 bits, formado por los 4 bytes 12h 34h 56h y 78h. Supongamos también que se quiere almacenar en la memoria a partir de la dirección 200h. Como cada posición de la memoria permite almacenar un solo byte, necesitaremos 4 posiciones de memoria, correspondientes a las direcciones 200h, 201h, 202h y 203h.

Little-endian		**Big-endian**	
Dirección	**Contenido**	**Dirección**	**Contenido**
200h	78h	200h	12h
201h	56h	201h	34h
202h	34h	202h	56h
203h	12h	203h	78h

2.1. Direccionamiento inmediato

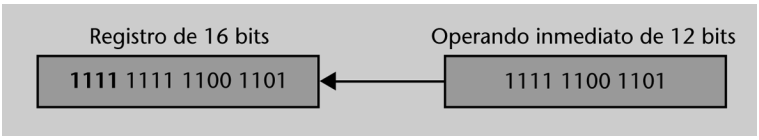
En el direccionamiento inmediato, el operando expresa el valor del dato que se quiere utilizar; es decir, el dato está dentro de la instrucción y su valor es fijo.

Este modo de direccionamiento se suele utilizar en operaciones aritméticas o lógicas, transferencias en las que se quiere inicializar registros y, de manera general, para definir y utilizar constantes.

El valor del dato se representa, normalmente, en complemento a 2 y cuando se transfiere a un registro o a una posición de memoria se hace la extensión de signo replicando el bit de signo a la izquierda hasta llenar el operando destino.

Ejemplo

Tenemos un operando inmediato de 12 bits en complemento a 2 y queremos transferir el número -52_{10} a un registro de 16 bits.



La ventaja principal de este modo de direccionamiento es que no es necesario ningún acceso adicional a memoria para obtener el dato, lo que agiliza la ejecución de la instrucción.

Las desventajas principales son que el valor del dato es constante y el rango de valores que se pueden representar está limitado por el tamaño de este operando, que no suele ser demasiado grande $[-2^n, 2^n - 1]$ si se representa en complemento a 2, donde n es el número de bits del operando.

Instrucciones de salto incondicional

En las instrucciones de salto incondicional, como veremos más adelante, el operando puede expresar la dirección donde se quiere saltar; en este caso, esta dirección es el dato, ya que la función de esta instrucción es cargar este valor en el PC y no hay que hacer ningún acceso a la memoria para obtenerlo. Por este motivo, se considera un modo de direccionamiento inmediato, aunque el operando exprese una dirección, como en el direccionamiento directo a memoria.

2.2. Direccionamiento directo

En el direccionamiento directo el operando indica dónde se encuentra el dato que se quiere utilizar. Si hace referencia a un registro de la máquina, el dato estará almacenado en este registro y hablaremos de **direccionamiento directo a registro**; si hace referencia a una posición de memoria, el dato estará almacenado en esta dirección de memoria (dirección efectiva) y hablaremos de **direccionamiento directo a memoria**.

Terminología

Podéis encontrar los modos de direccionamiento directos referenciados con otros nombres:

- El direccionamiento directo a registro como directo absoluto a registro o, simplemente, a registro.
- El direccionamiento directo a memoria como absoluto o directo absoluto.

Ejemplo de direccionamiento a registro y a memoria en la arquitectura CISCA

Operandos de 32 bits en complemento a 2 y utilizando formato *little-endian*

	Código oper.	Destino	Fuente
Instrucción	MOV	R2	[0AB0 0100h]
Función	R2 ← [0AB0 0100h] => R2 ← 01234567h		

Memoria		
Dirección	**Antes**	**Después**
0000 0000h		
...		
0AB0 0100h	67h	67h
0AB0 0101h	45h	45h
0AB0 0102h	23h	23h
0AB0 0103h	01h	01h
...		
0AB0 0200h		
0AB0 0201h		
0AB0 0202h		
0AB0 0203h		
...		
FFFF FFFFh		

Registros		
Registro	**Antes**	**Después**
R0		
R1		
R2	0000 0024h	0123 4567h
R3		
R4		
R5		
...		
R12		
R13		
R14		
R15/SP		
PC	0000 0100h	0000 0107h

Con la ejecución de esta instrucción queremos transferir el contenido de la dirección de memoria 0AB00100h al registro R2; el operando fuente hace referencia a la dirección 0AB00100h, donde se tiene que leer el dato 01234567h. Antes de la ejecución de la instrucción, el registro R2 contiene el valor 00000024h y después, el valor 01234567h, que es el dato que teníamos en la dirección de memoria 0AB00100h. En el operando destino tendremos un direccionamiento directo a registro, y en el operando fuente, un direccionamiento directo a memoria.

Estos modos de direccionamiento tienen una forma muy simple y no hay que hacer cálculos para obtener la dirección efectiva donde está el dato.

El tamaño del operando, en el caso del direccionamiento directo a registro, dependerá del número de registros que tenga la máquina, que suele ser relativamente reducido y, por lo tanto, se necesitan pocos bits; en el caso del direccionamiento directo a memoria, dependerá del tamaño de la memoria. En las máquinas actuales, el operando deberá ser muy grande para poder dirigir toda la memoria, lo que representa uno de los inconvenientes principales de este modo de direccionamiento.

Ejemplo

En una máquina con 32 registros y 4 Mbytes de memoria ($4 \cdot 2^{20}$ bytes) necesitaremos 5 bits para codificar los 32 registros ($32 = 2^5$) y 22 bits para codificar $4 \cdot 2^{20}$ direcciones de memoria de 1 byte ($4\ M = 2^{22}$) y se expresarán en binario puro, sin consideraciones de signo.

Las ventajas principales del direccionamiento directo a registro es que el tamaño del operando es muy pequeño y el acceso a los registros es muy rápido, por lo que es uno de los modos de direccionamiento más utilizado.

2.3. Direccionamiento indirecto

En el direccionamiento indirecto, el operando indica dónde está almacenada la dirección de memoria (dirección efectiva) que contiene el dato que queremos utilizar. Si hace referencia a un registro de la máquina, la dirección de memoria (dirección efectiva) que contiene el dato estará en este registro y hablaremos de **direccionamiento indirecto a registro**; si hace referencia a una posición de memoria, la dirección de memoria (dirección efectiva) que contiene el dato estará almacenada en esta posición de memoria y hablaremos de **direccionamiento indirecto a memoria**.

Ya hemos comentado que uno de los problemas del direccionamiento directo a memoria es que se necesitan direcciones muy grandes para poder acceder a toda la memoria; con el modo de direccionamiento indirecto esto no sucede: se puede guardar toda la dirección en un registro o en la memoria utilizando las posiciones que sean necesarias.

Si se guardan en registros, puede suceder que no todos los registros del procesador se puedan utilizar para almacenar estas direcciones, pero en este caso siempre habrá algunos especializados para poder hacerlo, ya que son imprescindibles para que el procesador funcione.

Generalmente, las direcciones que se expresan en el modo de direccionamiento indirecto a memoria son de tamaño inferior a las direcciones reales de memoria; por este motivo, solo se podrá acceder a un bloque de la memoria, que habitualmente se reserva como tabla de punteros en las estructuras de datos que utiliza el programa.

Este convenio será el mismo para cada máquina y siempre con el mismo valor o la misma operación; por este motivo solo se puede acceder a un bloque de la memoria direccionable del programa.

Ejemplo de direccionamiento indirecto a registro en la arquitectura CISCA

Operandos de 32 bits en complemento a 2 y utilizando formato *little-endian*

	Código oper.	Destino	Fuente
Instrucción	MOV	[R1]	R12
Función	[0AB00100h] ← 01234567h		

Memoria		
Dirección	**Antes**	**Después**
0000 0000h		
...		
0AB0 0100h	00h	67h
0AB0 0101h	00h	45h
0AB0 0102h	00h	23h
0AB0 0103h	00h	01h
...		
0AB0 0200h		
0AB0 0201h		
0AB0 0202h		
0AB0 0203h		
...		
FFFF FFFFh		

Registros		
Registro	**Antes**	**Después**
R0		
R1		
R2	0AB0 0100h	0AB0 0100h
R3		
R4		
R5		
...		
R12		
R13	0123 4567h	0123 4567h
R14		
R15/SP		
PC	0000 0100h	0000 0103h

Con la ejecución de esta instrucción se quiere transferir el contenido del registro R12 a la dirección de memoria 0AB00100h. El operando destino hace referencia al registro R1, que contiene la dirección 0AB00100h, donde se tiene que guardar el dato; el operando fuente hace referencia al registro R12, donde se tiene que leer el dato 01234567h. En el operando destino tendremos un direccionamiento indirecto a registro y en el operando fuente, un direccionamiento directo a registro.

Ejemplo de direccionamiento indirecto a memoria utilizando el formato de la arquitectura CISCA

Tened presente que esta instrucción sigue el formato de la arquitectura CISCA, pero no forma parte de su juego de instrucciones porque el modo de direccionamiento indirecto a memoria no se ha definido.

Operandos de 32 bits en complemento a 2 y utilizando formato *little-endian*

	Código oper.	Destino	Fuente
Instrucción	MOV	[[0AB00100h]]	R12
Función	[0AB00200h] ← 01234567h		

Memoria		
Dirección	**Antes**	**Después**
0000 0000h		
...		
0AB0 0100h	00h	00h
0AB0 0101h	02h	02h
0AB0 0102h	B0h	B0h
0AB0 0103h	0Ah	0Ah
...		
0AB0 0200h	00h	67h
0AB0 0201h	00h	45h
0AB0 0202h	00h	23h
0AB0 0203h	00h	01h
...		
FFFF FFFFh		

Registros		
Registro	**Antes**	**Después**
R0		
R1		
R2		
R3		
R4		
R5		
...		
R12	0123 4567h	0123 4567h
R13		
R14		
R15/SP		
PC	0000 0100h	0000 0107h

Con la ejecución de esta instrucción se quiere transferir el contenido del registro R12 a la dirección de memoria 0AB00200h. El operando destino hace referencia a la dirección 0AB00100h. En esta dirección está almacenada la dirección 0AB00200h, que es donde se debe guardar el dato, el operando fuente hace referencia al registro R12, donde se tiene que leer el dato 01234567h. En el operando destino tendremos un direccionamiento indirecto a memoria y en el operando fuente, un direccionamiento directo a registro.

La desventaja principal de este modo de direccionamiento es que necesita un acceso más a memoria que el directo. Es decir, un acceso a memoria para el direccionamiento indirecto a registro y dos accesos a memoria para el direccionamiento indirecto a memoria; por este motivo este segundo modo de direccionamiento no se implementa en la mayoría de las máquinas.

2.4. Direccionamiento relativo

En el direccionamiento relativo, el operando expresará dos valores, una dirección de memoria y un desplazamiento respecto a esta dirección (salvo los casos en los que uno de los dos sea implícito). La dirección de memoria (dirección efectiva) donde tendremos el dato la obtendremos sumando el desplazamiento a la dirección de memoria. Las variantes más utilizadas de este modo de direccionamiento son **direccionamiento relativo a registro base**, **direccionamiento relativo a registro índice** y **direccionamiento relativo a PC**.

Terminología

Podéis encontrar el direccionamiento relativo, y sus variantes, referenciado como direccionamiento con desplazamiento o direccionamiento indexado.

Estos modos de direccionamiento son muy potentes, no requieren accesos extras a la memoria, como sucede con el indirecto, pero hay que hacer una suma, que no retrasa casi la ejecución de la instrucción, especialmente en las máquinas que tienen una unidad específica para el cálculo de estas direcciones.

Desde el punto de vista del programador, estos modos de direccionamiento son muy útiles para acceder a estructuras de datos como vectores, matrices o listas, ya que se aprovecha la localidad de los datos, dado que la mayoría de las referencias en la memoria son muy próximas entre sí.

Estos modos de direccionamiento también son la base para hacer que los códigos sean reentrantes y reubicables, ya que permiten cambiar las referencias a las direcciones de memoria cambiando simplemente el valor de un registro, para acceder tanto a los datos como al código mediante las direcciones de las instrucciones de salto o llamadas a subrutinas; también son útiles en algunas técnicas para proteger espacios de memoria y para implementar la segmentación. Estos usos se aprovechan en la compilación y en la gestión que hace el sistema operativo de los programas en ejecución y, por lo tanto, son transparentes al programador.

2.4.1. Direccionamiento relativo a registro base

En el direccionamiento relativo a registro base, la dirección de memoria se almacena en el registro que denominaremos *registro base* (RB) y el desplazamiento se encuentra explícitamente en la instrucción. Es más compacto que el modo de direccionamiento absoluto a memoria, ya que el número de bits utilizados para el desplazamiento es inferior al número de bits de una dirección de memoria.

En algunas instrucciones el registro base se utiliza de manera implícita y, por lo tanto, solo habrá que especificar explícitamente el desplazamiento.

Este modo de direccionamiento se utiliza a menudo para implementar la segmentación. Algunas máquinas tienen registros específicos para eso y se utilizan de manera implícita, mientras que en otras máquinas se pueden elegir los registros que se quieren utilizar como base del segmento, pero, entonces, hay que especificarlo explícitamente en la instrucción.

Segmentación de los Intel x86-64

En el caso del Intel x86-64, se implementa la segmentación utilizando registros de segmento específicos (CS:codigo, DS:datos, SS:pila y ES, FS, GS:extra), pero también se tiene la flexibilidad de poder reasignar algunos, tanto si es de manera explícita en las instrucciones o mediante directivas de compilación como la ASSUME.

2.4.2. Direccionamiento relativo a registro índice

En el direccionamiento relativo a registro índice, la dirección de memoria se encuentra explícitamente en la instrucción y el desplazamiento se almacena en el registro que denominaremos *registro índice* (RI). Justo al contrario que en el direccionamiento relativo a registro base.

Este modo de direccionamiento se utiliza a menudo para acceder a estructuras de datos como vectores, matrices o listas, por lo que es muy habitual que después de cada acceso se incremente o decremente el registro índice un valor constante (que depende del tamaño y el número de posiciones de memoria de los datos a los que se accede). Por este motivo, algunas máquinas hacen esta operación de manera automática y proporcionan diferentes alternativas que denominaremos *direccionamientos relativos autoindexados*. Si la autoindexación se lleva a cabo sobre registros de carácter general, puede requerir un bit extra para indicarlo en la codificación de la instrucción.

Es muy habitual permitir tanto el autoincremento como el autodecremento, operación que se puede realizar antes de obtener la dirección efectiva o después. Por lo tanto, habrá cuatro alternativas de implementación, aunque un mismo juego de instrucciones no las tendrá simultáneamente:

1) Preautoincremento: RI se incrementa antes de obtener la dirección.

2) Preautodecremento: RI se decrementa antes de obtener la dirección.

3) Postautoincremento: RI se incrementa después de obtener la dirección.

4) Postautodecremento: RI se decrementa después de obtener la dirección.

Ejemplo de direccionamiento relativo a registro base en la arquitectura CISCA

Operandos de 32 bits en complemento a 2 y utilizando formato *little-endian*

	Código oper.	Destino	Fuente
Instrucción	MOV	[R2+0100h]	R12
Función	[0AB0 0000h+0100h] ← 01234567h		

Memoria		
Dirección	**Antes**	**Después**
0000 0000h		
...		
0AB0 0100h	00h	67h
0AB0 0101h	00h	45h
0AB0 0102h	00h	23h
0AB0 0103h	00h	01h
...		
0AB0 0200h		
0AB0 0201h		
0AB0 0202h		
0AB0 0203h		
...		
FFFF FFFFh		

Registros		
Registro	**Antes**	**Después**
R0		
R1		
R2	0AB0 0000h	0AB0 0000h
R3		
R4		
R5		
...		
R12	0123 4567h	0123 4567h
R13		
R14		
R15/SP		
PC	0000 0100h	0000 0105h

Con la ejecución de esta instrucción se quiere transferir el contenido del registro R12 a la dirección de memoria 0AB00100h. El operando destino hace referencia al registro R2, que hace de registro base y contiene la dirección 0AB00000h, y al desplazamiento 0100h, que, sumado a la dirección que hay en R2, da la dirección de memoria donde se debe guardar el dato (posición 0AB00100h); el operando fuente hace referencia al registro R12, donde se tiene que leer el dato 01234567h. En el operando destino tendremos un direccionamiento relativo a registro base y en el operando fuente, un direccionamiento directo a registro.

Ejemplo de direccionamiento relativo a registro índice en la arquitectura CISCA

Operandos de 32 bits en complemento a 2 y utilizando formato *little-endian*

	Código oper.	Destino	Fuente
Instrucción	MOV	[0AB00000h+R1]	R12
Función	[0AB0 0000h+0200h] ← 01234567h		

Memoria		
Dirección	**Antes**	**Después**
0000 0000h		
...		
0AB0 0100h		
0AB0 0101h		
0AB0 0102h		
0AB0 0103h		
...		
0AB0 0200h	00h	67h
0AB0 0201h	00h	45h
0AB0 0202h	00h	23h
0AB0 0203h	00h	01h
...		
FFFF FFFFh		

Registros		
Registro	**Antes**	**Después**
R0		
R1	0000 0200h	0000 0200h
R2		
R3		
R4		
R5		
...		
R12	0123 4567h	0123 4567h
R13		
R14		
R15/SP		
PC	0000 0100h	0000 0107h

Con la ejecución de esta instrucción se quiere transferir el contenido del registro R12 a la dirección de memoria 0AB00200h. El operando destino hace referencia a la dirección 0AB00000h y al registro R1, que hace de registro índice y contiene el desplazamiento 00000200h, que, sumados, dan la dirección de memoria donde se debe guardar el dato (posición 0AB00200h); el operando fuente hace referencia al registro R12, donde se debe leer el dato 01234567h. En el operando destino tendremos un direccionamiento relativo a registro índice y en el operando fuente, un direccionamiento directo a registro.

Muchas máquinas habitualmente implementan variantes de estos modos de direccionamiento, combinando el direccionamiento relativo a registro base y el direccionamiento relativo a registro índice. Utilizan dos registros o más y el desplazamiento, lo que evita poner la dirección de memoria. Tienen tanta o más potencia de acceso y se puede reducir bastante el tamaño de la instrucción, ya que solo hay que identificar registros o registros y un desplazamiento.

2.4.3. Direccionamiento relativo a PC

El direccionamiento relativo a PC es equivalente al relativo a registro base, con la diferencia de que utiliza el registro contador de programa (PC) de manera implícita en la instrucción y solo hay que expresar, mediante una etiqueta, el desplazamiento para calcular la dirección de memoria (dirección efectiva) a la que se quiere acceder.

Otra característica que diferencia el modo de direccionamiento relativo a PC es la manera de expresar el desplazamiento: como puede ser un valor tanto positivo como negativo, se suele representar en complemento a 2.

El direccionamiento relativo a PC se utiliza a menudo en las instrucciones de ruptura de secuencia, generalmente en los saltos condicionales, que suelen ser cortos y, en consecuencia, el tamaño del campo no tendrá que ser muy grande.

Ejemplo de direccionamiento relativo a registro PC en la arquitectura CISCA

Tenemos el fragmento de código siguiente:

Memoria		
Dirección	**Etiqueta**	**Instrucción**
0001 1000h		MOV R0,0
0001 1007h	bucle:	ADD R0,1
0001 100Eh		CMP R0,8
0001 1015h		JE bucle
0001 1019h		MOV R0,–1

En este código, cada instrucción ocupa 7 bytes, salvo la instrucción JE bucle, que ocupa 4 bytes. La etiqueta *bucle* hace referencia a la dirección de memoria 00001007h. La instrucción *JE bucle* (salta si el bit de cero está activo) utiliza direccionamiento relativo a PC y, por lo tanto, no codifica la dirección de memoria a la que se quiere saltar, sino el desplazamiento respecto al PC actualizado utilizando 16 bits (desplazamiento = etiqueta-$PC_{updated}$).

En el ciclo de ejecución de la instrucción, la actualización del PC se realiza en las primeras fases de este ciclo, por lo tanto, al final de la ejecución de la instrucción (que es cuando sabremos si hemos de saltar) el PC no apuntará a la instrucción de salto (JE bucle; PC=00011015h), sino a la instrucción siguiente (MOV R0,–1; $PC_{updated}$ = 00011019h); es decir, deberemos saltar tres instrucciones atrás y, como la instrucción JE bucle ocupa 4 bytes y el resto, 7 bytes cada una, el desplazamiento será de –18; para obtener este valor restaremos de la dirección de la etiqueta *bucle* el valor de $PC_{updated}$ (00011007h – 000110119h = FFEEh(–18 en Ca2)). Este valor, FFEEh, es el que se codificará como desplazamiento en la instrucción de salto condicional.

Entonces, cuando se ejecuta una instrucción de salto condicional, si se cumple la condición se actualiza el PC sumándole el desplazamiento que tenemos codificado. En nuestro caso, después de ejecutar la instrucción de salto condicional *JE bucle* si el bit de cero C está activo, saltamos y PC valdrá 00011007h y si el bit de cero C no está activo, saltamos y PC valdrá 00011019h.

El compilador lleva a cabo la conversión de las etiquetas utilizadas en un programa a un desplazamiento de manera transparente al programador porque el modo de direccionamiento está implícito en las instrucciones de salto.

Es muy importante para el programador saber qué modo de direccionamiento utiliza cada instrucción de ruptura de secuencia porque el desplazamiento que se puede especificar en el direccionamiento relativo a PC no es muy grande y puede suceder que no permita llegar a la dirección deseada; entonces habrá

que modificar el programa para poder utilizar otra instrucción de ruptura de secuencia con un direccionamiento que permita llegar a todo el espacio dirigible.

Ejemplo

En el ejemplo anterior se utilizan 16 bits para el desplazamiento, por lo tanto, se pueden saltar 32.767 posiciones adelante y 32.768 posiciones atrás. Así pues, el rango de direcciones al que se puede acceder desde la dirección apuntada por el $PC_{updated}$ (00011019h) es desde la dirección 00009019h a la dirección 00019018h; si se quiere saltar fuera de este espacio, no se puede utilizar esta instrucción porque utiliza direccionamiento relativo a PC.

2.5. Direccionamiento implícito

En el direccionamiento implícito, la instrucción no contiene información sobre la localización del operando porque este se encuentra en un lugar predeterminado, y queda especificado de manera implícita en el código de operación.

2.5.1. Direccionamiento a pila

El direccionamiento a la 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 por medio de registros de la máquina; habitualmente uno de estos registros se conoce como *stack pointer* (SP) y se utiliza para apuntar a la cima de la pila.

Pila

Una pila es una lista de elementos con el acceso restringido, de manera que solo se puede acceder a los elementos de un extremo de la lista. Este extremo se denomina *cima de la pila*. El último elemento que entra a la pila es el primero que sale (LIFO). A medida que se van añadiendo elementos, la pila crece y según la implementación lo hará hacia direcciones más pequeñas (método más habitual) o hacia direcciones mayores.

Como es un modo de direccionamiento implícito, solo se utiliza en instrucciones determinadas, las más habituales de las cuales son PUSH (poner un elemento en la pila) y POP (sacar un elemento de la pila).

Este modo de direccionamiento se podría entender como un direccionamiento indirecto a registro añadiendo la funcionalidad de autoindexado que ya hemos comentado. Es decir, se accede a una posición de memoria identificada por un registro, el registro SP, que se actualiza, antes o después del acceso a memoria, para que apunte a la nueva cima de la pila.

Ejemplo

Supongamos que cada elemento de la pila es una palabra de 2 bytes, la memoria se dirige a nivel de byte y SP apunta al elemento que está en la cima de la pila.

Para poner un elemento en la pila (PUSH X) habrá que hacer lo siguiente:

Crece hacia direcciones más pequeñas (preautodecremento)	Crece hacia direcciones mayores (preautoincremento)
SP = SP – 2 M[SP] = X	SP = SP + 2 M[SP] = X

Para sacar un elemento de la pila (POP X) habrá que hacer lo siguiente:

Crece hacia direcciones más pequeñas (postautoincremento)	Crece hacia direcciones mayores (postautodecremento)
X = M[SP] SP = SP + 2	X = M[SP] SP = SP – 2

Ejemplo de direccionamiento a pila en la arquitectura CISCA

Operandos de 32 bits en complemento a 2 y utilizando formato *little-endian*. La pila crece hacia direcciones pequeñas

	Código oper.	Fuente
Instrucción	PUSH	R2
Función	[SP – 4] ← 01234567h	

Memoria		
Dirección	**Antes**	**Después**
0000 0000h		
...		
0AB0 0100h		
0AB0 0101h		
0AB0 0102h		
0AB0 0103h		
...		
FFFF FFFAh		
FFFF FFFBh		
FFFF FFFCh	00h	67h
FFFF FFFDh	00h	45h
FFFF FFFEh	00h	23h
FFFF FFFFh	00h	01h

Registros		
Registro	**Antes**	**Después**
R0		
R1		
R2	01234567h	01234567h
R3		
R4		
R5		
...		
R12		
R13		
R14		
R15/SP	0000 0000h	FFFF FFFCh
PC	0000 0100h	0000 0102h

Con la ejecución de esta instrucción se quiere transferir el contenido del registro R2 a la cima de la pila. El operando fuente hace referencia al registro R2, donde se ha de leer el dato 01234567h, que queremos poner en la pila. Para guardar este dato en la pila, primero se decrementa en 4 el registro SP (en CISCA, el registro R15), porque los datos son de 32 bits. El registro SP inicialmente vale 00000000h, 00000000h – 4 = FFFFFFFCh (–4 en Ca2). Después, de la misma manera que en un direccionamiento indirecto a registro, utilizaremos esta dirección que tenemos en el registro SP para almacenar el valor que tenemos en R2 en la memoria, y SP quedará apuntando a la cima de la pila. Si SP vale 0, querrá decir que la pila está vacía.

Resumen

Hemos empezado hablando de las características principales de los juegos de instrucciones, de los cuales hemos destacado diferentes puntos.

Hemos visto que el ciclo de ejecución de la instrucción se divide en cuatro fases:

Fase 1 Lectura de la instrucción

Fase 2 Lectura de los operandos fuente

Fase 3 Ejecución de la instrucción y almacenamiento del operando destino

Fase 4 Comprobación de interrupciones

El tipo de arquitectura del juego de instrucciones depende de la localización de los operandos y a partir de aquí hemos definido cinco tipos de arquitecturas: pila, acumulador, registro-registro, registro-memoria, memoria-memoria.

La representación del juego de instrucciones se efectúa desde dos puntos de vista:

- El punto de vista del programador, que denominamos *lenguaje de ensamblador.*

- El punto de vista del computador, que denominamos *lenguaje de máquina*.

Con respecto al formato de las instrucciones, hemos visto lo siguiente:

- Los elementos que componen la instrucción: código de operación, operandos fuente, operando destino y dirección de la instrucción siguiente.

- El tamaño de las instrucciones, que puede ser fijo o variable, y cómo determinar el tamaño de los diferentes campos.

Con respecto a los operandos de la instrucción, hemos analizado el número de operandos que puede tener y su localización, así como el tipo y el tamaño de los datos que tratamos con los operandos.

Los tipos de instrucciones vistos en el módulo aparecen referenciados en la tabla siguiente.

Tipos de instrucciones		Ejemplos
Instrucciones de transferencia de datos		MOV AL, 0
		MOV R1, R2
Instrucciones aritméticas	Suma	ADD R1, R2
	Resta	SUB R1, 2
	Multiplicación	MUL R1, R2
	División	DIV R2, R3
	Incremento	INC RAX
	Decremento	DEC RAX
	Comparación	CMP RAX, RBX
	Negación	NEG RAX
Instrucciones lógicas	AND	AND R1, 1
	OR	OR R1, R2
	XOR	XOR R1, R2
	NOT	NOT R1
	Desplazamiento lógico a la izquierda	SHL RAX, 1
	Desplazamiento lógico a la derecha	SHR RAX, 1
	Desplazamiento aritmético a la izquierda	SAL RAX, 1
	Desplazamiento aritmético a la derecha	SAR RAX, 1
	Rotación a la izquierda	ROL RAX, 1
	Rotación a la derecha	ROR RAX, 1
Instrucciones de ruptura de secuencia	Salto incondicional	JMP etiqueta
	Salto condicional	JE etiqueta
	Instrucciones de llamada y retorno de subrutina	CALL etiqueta
	Instrucciones de interrupción de software	INT 80h
	Instrucciones de retorno de interrupción	IRET
Instrucciones de entrada/salida	Entrada	IN AL, 20h
	Salida	OUT 20h, AL
Otros tipos de instrucciones		NOP

También se han explicado detalladamente los diferentes modos de direccionamiento que puede tener un juego de instrucciones, que están resumidos en la tabla siguiente.

Direccionamiento	Sintaxis	Que expresa OP	Cómo obtenemos la dirección	Cómo obtenemos el dato	Observaciones
Inmediato	Valor	OP = Dato		Dato = OP	
Directo a registro	R	OP = R		Dato = R	
Directo a memoria	[A]	OP = A	AE = A	Dato = M[A]	
Indirecto a registro	[R]	OP = R	AE = R	Dato = M[R]	
Indirecto a memoria	[[A]]	OP = A	AE = M[A]	Dato = M[M[A]]	
Relativo a RB	[RB + Desp.]	OP = RB + Desp.	AE = RB + Desp.	Dato = M[RB + Desp.]	
Relativo a RI	[A + RI]	OP = EN + RI	AE = EN + RI	Dato = M[EN + RI]	Con preautoincremento o postautoincremento RI = RI ± 1
Relativo a PC	A	OP = Desp.		Dato = PC + Desp.	PC es implícito y Dato es la dirección de la instrucción siguiente por ejecutar.
A pila	Valor o R o A	OP = Dato o R o A	AE = SP	Dato = M[SP]	Direccionamiento implícito

Abreviaturas de la tabla. Dato: con el que queremos operar; OP: información expresada en el operando de la instrucción; A: dirección de memoria; AE: dirección efectiva de memoria (dirección donde está el dato); R: referencia de un registro;
Desp.: desplazamiento; [R]: contenido de un registro; M[A]: contenido de una dirección de memoria; [ ]: para indicar acceso a memoria.

El procesador

Miquel Albert Orenga
Gerard Enrique Manonellas

PID_00177072

Universitat Oberta de Catalunya
www.uoc.edu

Índice

Introducción

En este módulo estudiaremos el componente principal de un computador: el procesador, unidad central de proceso o CPU (siglas de la expresión inglesa *central processing unit*).

La función principal que tiene es procesar los datos y transferirlos a los otros elementos del computador. Estas tareas se llevan a cabo mediante la ejecución de instrucciones. Por este motivo, el objetivo principal a la hora de diseñar un procesador es conseguir que las instrucciones se ejecuten de la manera más eficiente posible.

En este módulo nos centraremos en analizar los elementos principales del procesador desde el punto de vista funcional y no entraremos a analizar los aspectos relacionados con la mejora de rendimiento.

Los elementos básicos del procesador que estudiaremos son los siguientes:

- Conjunto de registros.
- Unidad aritmética y lógica.
- Unidad de control.

Del conjunto de registros describiremos la función principal y el tipo de información que pueden almacenar.

De la unidad aritmética y lógica (ALU) veremos la función principal y el tipo de datos con los que se trabaja habitualmente. No entraremos en más detalle porque las operaciones aritméticas y lógicas ya se han estudiado ampliamente en asignaturas previas de estos estudios.

Finalmente, haremos un estudio detallado de la unidad de control en el que veremos que esta es el elemento clave para el funcionamiento correcto del procesador y la ejecución de las instrucciones.

Objetivos

Con el estudio de este módulo se pretende que el estudiante alcance los objetivos siguientes:

1. Entender la organización de un procesador en lo relativo a las unidades funcionales que lo componen: registros, ALU y unidad de control.
2. Entender cómo ejecuta una instrucción un procesador.
3. Conocer la organización del conjunto de registros del procesador.
4. Ver los conceptos básicos relacionados con la ALU.
5. Entender el funcionamiento de la unidad de control del procesador.
6. Conocer las ideas clave para diseñar una unidad de control.
7. Saber las diferencias principales entre computadores RISC y CISC.

1. Organización del procesador

La función principal de un procesador es ejecutar instrucciones y la organización que tiene viene condicionada por las tareas que debe realizar y por cómo debe hacerlo.

Los procesadores están diseñados y operan según una señal de sincronización. Esta señal, conocida como *señal de reloj*, es una señal en forma de onda cuadrada periódica con una determinada frecuencia. Todas las operaciones hechas por el procesador las gobierna esta señal de reloj: un ciclo de reloj determina la unidad básica de tiempo, es decir, la duración mínima de una operación del procesador.

Frecuencia de la señal de reloj

La frecuencia de la señal de reloj se define como el número de impulsos por unidad de tiempo, se mide en ciclos por segundo o hercios (Hz) y determina la velocidad de operación del procesador.

Para ejecutar una instrucción, son necesarios uno o más ciclos de reloj, dependiendo del tipo de instrucción y de los operandos que tenga.

Las prestaciones del procesador no las determina solo la frecuencia de reloj, sino otras características del procesador, especialmente del diseño del juego de instrucciones y la capacidad que tiene para ejecutar simultáneamente múltiples instrucciones.

Para ejecutar las instrucciones, todo procesador dispone de tres componentes principales:

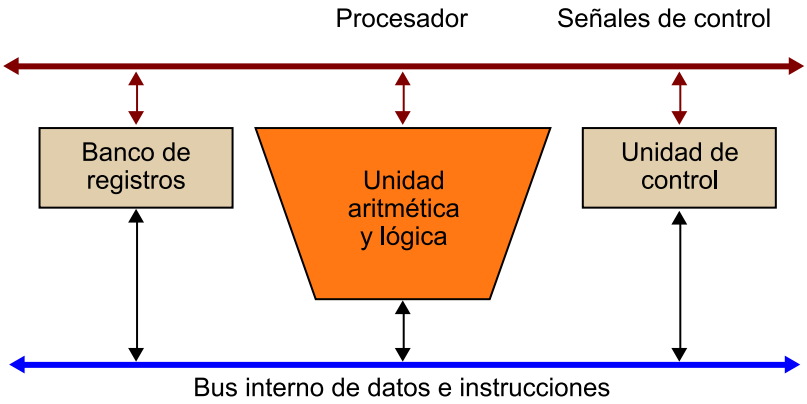
1) Un **conjunto de registros**: espacio de almacenamiento temporal de datos e instrucciones dentro del procesador.

2) Unidad aritmética y lógica o ALU[1]: circuito que hace un conjunto de operaciones aritméticas y lógicas con los datos almacenados dentro del procesador.

3) Unidad de control: circuito que controla el funcionamiento de todos los componentes del procesador. Controla el movimiento de datos e instrucciones dentro y fuera del procesador y también las operaciones de la ALU.

(1) *ALU* son las siglas de la expresión inglesa *arithmetic logic unit*.

La organización básica de los elementos que componen un procesador y el flujo de información entre los diferentes elementos se ve en el esquema siguiente:



Como se observa, aparte de los tres componentes principales, es necesario disponer de un sistema que permita interconectar estos componentes. Este sistema de interconexión es específico para cada procesador. Distinguimos dos tipos de líneas de interconexión: líneas de control, que permiten gobernar el procesador, y líneas de datos, que permiten transferir los datos y las instrucciones entre los diferentes componentes del procesador. Este sistema de interconexión tiene que disponer de una interfaz con el bus del sistema.

El término *procesador* actualmente se puede entender como microprocesador porque todas las unidades funcionales que forman el procesador se encuentran dentro de un chip, pero hay que tener presente que, por el aumento de la capacidad del nivel de integración, dentro de los microprocesadores se pueden encontrar otras unidades funcionales del computador. Por ejemplo:

- **Unidades de ejecución SIMD**: unidades especializadas en la ejecución de instrucciones SIMD (*single instruction, multiple data*), instrucciones que trabajan con estructuras de datos vectoriales, como por ejemplo instrucciones multimedia.

- **Memoria caché**: prácticamente todos los procesadores modernos incorporan dentro del propio chip del procesador algunos niveles de memoria caché.

- **Unidad de gestión de memoria** o ***memory management unit*** **(MMU)**: gestiona el espacio de direcciones virtuales, traduciendo las direcciones de memoria virtual a direcciones de memoria física en tiempo de ejecución. Esta traducción permite proteger el espacio de direcciones de un programa del espacio de direcciones de otros programas y también permite separar el espacio de memoria del sistema operativo del espacio de memoria de los programas de usuario.

- **Unidad de punto flotante** o ***floating point unit*** **(FPU)**: unidad especializada en hacer operaciones en punto flotante; puede funcionar de manera autónoma, ya que dispone de un conjunto de registros propio.

2. Ciclo de ejecución de las instrucciones

El ciclo de ejecución es la secuencia de operaciones que se hace para ejecutar cada una de las instrucciones. Lo dividimos en cuatro fases principales:

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

Orden de operaciones

Las fases principales del ciclo de ejecución son comunes a la mayoría de los computadores actuales y las diferencias principales se encuentran en el orden en el que se realizan algunas de las operaciones de cada fase o en el que se realizan algunas de las operaciones en otras fases.

En cada fase se incluyen una o varias operaciones que hay que hacer. Para llevar a cabo estas operaciones, nombradas *microoperaciones*, es necesaria una compleja gestión de las señales de control y de la disponibilidad de los diferentes elementos del procesador, tarea realizada por la unidad de control.

Veamos las operaciones principales que se realizan habitualmente en cada una de las cinco fases del ciclo de ejecución de las instrucciones:

Fase 1. Lectura de la instrucción. Las operaciones realizadas en esta fase son las siguientes:

a) Leer la instrucción. Cuando leemos la instrucción, el registro contador del programa (PC) nos indica la dirección de memoria donde está la instrucción que hemos de leer. Si el tamaño de la instrucción es superior a la palabra de la memoria, hay que hacer tantos accesos a la memoria como sean necesarios para leerla completamente y cargar toda esta información en el registro de instrucción (IR).

b) Descodificar la instrucción. Se identifican las diferentes partes de la instrucción para determinar qué operaciones hay que hacer en cada fase del ciclo de ejecución. Esta tarea la realiza la unidad de control del procesador leyendo la información que hemos cargado en el registro de instrucción (IR).

c) Actualizar el contador del programa. El contador del programa se actualiza según el tamaño de la instrucción, es decir, según el número de accesos a la memoria que hemos hecho para leer la instrucción.

Actualización del PC

La actualización del contador del programa es una operación que se puede encontrar en otras fases de la ejecución de la instrucción.

Las operaciones de esta fase son comunes a los diferentes tipos de instrucciones que encontramos en el juego de instrucciones.

Fase 2. Lectura de los operandos fuente. Esta fase se debe repetir para todos los operandos fuente que tenga la instrucción.

Las operaciones que hay que realizar en esta fase dependen del modo de direccionamiento que tengan los operandos: para los más simples, como el inmediato o el directo a registro, no hay que hacer ninguna operación; para los indirectos o los relativos, hay que hacer cálculos y accesos a memoria. Si el operando fuente es implícito, vamos a buscar el dato en el lugar predeterminado por aquella instrucción.

Ved también

Los modos de direccionamiento se explican en el apartado 2 del módulo "Juego de instrucciones".

Es fácil darse cuenta de que dar mucha flexibilidad a los modos de direccionamiento que podemos utilizar en cada instrucción puede afectar mucho al tiempo de ejecución de una instrucción.

Por ejemplo, si efectuamos un ADD R3,3 no hay que hacer ningún cálculo ni ningún acceso a memoria para obtener los operandos; en cambio, si efectuamos un ADD [R1], [R2+16], hay que hacer una suma y dos accesos a memoria.

Fase 3. Ejecución de la instrucción y almacenamiento del operando de destino (resultado)

Las operaciones que hay que realizar en esta fase dependen del código de operación de la instrucción y del modo de direccionamiento que tenga el operando destino.

Ejecución de la instrucción

Las operaciones que se llevan a cabo son diferentes para cada código de operación. Durante la ejecución, además de obtener el resultado de la ejecución de la instrucción, se pueden modificar los bits de resultado de la palabra de estado del procesador.

Almacenamiento del operando de destino (resultado)

La función básica es recoger el resultado obtenido durante la ejecución y guardarlo en el lugar indicado por el operando, a menos que el operando sea implícito, en cuyo caso se guarda en el lugar predeterminado por aquella instrucción. El operando de destino puede ser uno de los operandos fuente; de esta manera, no hay que repetir los cálculos para obtener la dirección del operando.

Fase 4. Comprobación de interrupciones

Las interrupciones son el mecanismo mediante el cual un dispositivo externo al procesador puede interrumpir el programa que está ejecutando el procesador con el fin de ejecutar otro programa (una rutina de servicio a la interrupción o RSI) para dar servicio al dispositivo que ha producido la interrupción.

La petición de interrupción se efectúa activando alguna de las líneas de petición de las que dispone el procesador.

Ved también

El concepto de interrupción se explica en detalle en el apartado 3 dentro del módulo "Sistema de entrada/salida".

En esta fase se verifica si se ha activado alguna línea de petición de interrupción del procesador en el transcurso de la ejecución de la instrucción. Si no se ha activado ninguna, continuamos el proceso normalmente; es decir, se acaba la ejecución de la instrucción en curso y se empieza la ejecución de la instrucción siguiente. En caso contrario, hay que transferir el control del procesador a la rutina de servicio de interrupción que debe atender esta interrupción y hasta que no se acabe no podemos continuar la ejecución de la instrucción en curso.

Para atender la interrupción, es necesario llevar a cabo algunas operaciones (intentaremos almacenar la mínima información para que el proceso sea tan rápido como se pueda) para transferir el control del procesador a la rutina de servicio de interrupciones, de manera que, después, lo podamos recuperar con la garantía de que el procesador estará en el mismo estado en el que estaba antes de transferir el control a la rutina de servicio de interrupción:

- Almacenar el contador del programa y la palabra de estado (generalmente, se guardan en la pila).
- Almacenar la dirección donde empieza la rutina para atender la interrupción en el contador del programa.
- Ejecutar la rutina de servicio de interrupción.
- Recuperar el contador del programa y la palabra de estado.

Esta fase actualmente está presente en todos los computadores, ya que todos utilizan E/S para interrupciones y E/S para DMA.

Ved también

En el módulo didáctico "Sistema de entrada/salida" veremos que los ordenadores requieren líneas de petición de interrupción que también se pueden utilizar para otras tareas que no son específicas de E/S.

2.1. Segmentación de las instrucciones

La segmentación de las instrucciones (*pipeline*) consiste en dividir el ciclo de ejecución de las instrucciones en un conjunto de etapas. Estas etapas pueden coincidir o no con las fases del ciclo de ejecución de las instrucciones.

El objetivo de la segmentación es ejecutar simultáneamente diferentes etapas de distintas instrucciones, lo cual permite aumentar el rendimiento del procesador sin tener que hacer más rápidas todas las unidades del procesador (ALU, UC, buses, etc.) y sin tener que duplicarlas.

La división de la ejecución de una instrucción en diferentes etapas se debe realizar de tal manera que cada etapa tenga la misma duración, generalmente un ciclo de reloj. Es necesario añadir registros para almacenar los resultados intermedios entre las diferentes etapas, de modo que la información generada en una etapa esté disponible para la etapa siguiente.

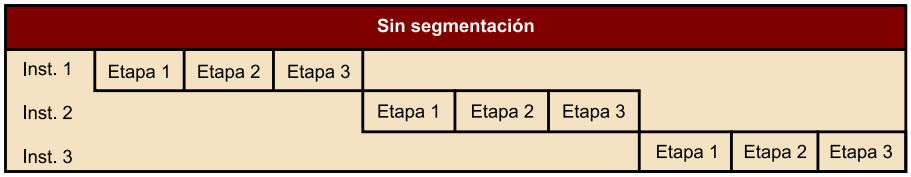
Ejemplo de segmentación de instrucciones

La segmentación es como una cadena de montaje. En cada etapa de la cadena se lleva a cabo una parte del trabajo total y cuando se acaba el trabajo de una etapa, el producto pasa a la siguiente y así sucesivamente hasta llegar al final.

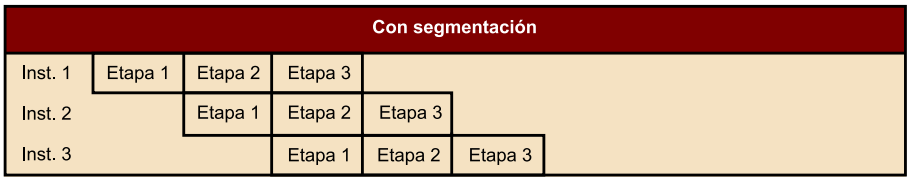
Si hay *N* etapas, se puede trabajar sobre *N* productos al mismo tiempo y, si la cadena está bien equilibrada, saldrá un producto acabado en el tiempo que se tarda en llevar a cabo una de las etapas. De esta manera, no se reduce el tiempo que se tarda en hacer un producto, sino que se reduce el tiempo total necesario para hacer una determinada cantidad de productos porque las operaciones de cada etapa se efectúan simultáneamente.

Si consideramos que el producto es una instrucción y las etapas son cada una de las fases de ejecución de la instrucción, que llamamos *etapa de segmentación*, hemos identificado los elementos y el funcionamiento de la segmentación.

Veámoslo gráficamente. Presentamos un mismo proceso con instrucciones de tres etapas, en el que cada etapa tarda el mismo tiempo en ejecutarse, resuelto sin segmentación y con segmentación:



Son necesarias nueve etapas para ejecutar las tres instrucciones.



Son necesarias cinco etapas para ejecutar las tres instrucciones.

Se ha reducido el tiempo total que se tarda en ejecutar las tres instrucciones, pero no el tiempo que se tarda en ejecutar cada una de las instrucciones.

Si se quiere implementar la segmentación para mejorar el rendimiento de un procesador, hay que tener en cuenta algunas cuestiones que implicarán cambios en el diseño del procesador. Las más importantes son las siguientes:

- ¿Cómo hay que hacerlo para que las etapas estén bien equilibradas (que todas tengan la misma duración)?

- ¿Qué recursos son necesarios en cada etapa para que todas se puedan ejecutar simultáneamente?

- ¿Cómo hay que tratar las instrucciones de salto para minimizar los efectos en el rendimiento de la segmentación?

Nota

Estas cuestiones no son nada sencillas de contestar, pero no es el objetivo de este curso entrar en el detalle de esta problemática. Solo pretendemos que entendáis el funcionamiento de la ejecución de las instrucciones utilizando segmentación.

3. Registros

Los registros son, básicamente, elementos de memoria de acceso rápido que se encuentran dentro del procesador. Constituyen un espacio de trabajo para el procesador y se utilizan como un espacio de almacenamiento temporal. Se implementan utilizando elementos de memoria RAM estática (*static RAM*). Son imprescindibles para ejecutar las instrucciones, entre otros motivos, porque la ALU solo trabaja con los registros internos del procesador.

El conjunto de registros y la organización que tienen cambia de un procesador a otro; los procesadores difieren en el número de registros, en el tipo de registros y en el tamaño de cada registro.

Organización de registros

Para ver cómo se organizan los registros y cómo se utilizan en una arquitectura concreta, podéis consultar los módulos "Programación en ensamblador (x86-64)" y "Arquitectura CISCA".

Una parte de los registros pueden ser visibles para el programador de aplicaciones, otra parte solo para instrucciones privilegiadas y otra solo se utiliza en el funcionamiento interno del procesador.

Una posible clasificación de los registros del procesador es la siguiente:

- Registros de propósito general.
- Registros de instrucción.
- Registros de acceso a memoria.
- Registros de estado y de control.

3.1. Registros de propósito general

Los registros de propósito general son los registros que suelen utilizarse como operandos en las instrucciones del ensamblador. Estos registros se pueden asignar a funciones concretas: datos o direccionamiento. En algunos procesadores todos los registros se pueden utilizar para todas las funciones.

Los registros de datos se pueden diferenciar por el formato y el tamaño de los datos que almacenan; por ejemplo, puede haber registros para números enteros y para números en punto flotante.

Los registros de direccionamiento se utilizan para acceder a memoria y pueden almacenar direcciones o índices. Algunos de estos registros se utilizan de manera implícita para diferentes funciones, como por ejemplo acceder a la pila, dirigir segmentos de memoria o hacer de soporte en la memoria virtual.

3.2. Registros de instrucción

Los dos registros principales relacionados con el acceso a las instrucciones son:

- ***Program counter*** **(PC)**: registro contador del programa, contiene la dirección de la instrucción siguiente que hay que leer de la memoria.
- ***Instruction register*** **(IR)**: registro de instrucción, contiene la instrucción que hay que ejecutar.

3.3. Registros de acceso a memoria

Hay dos registros necesarios para cualquier operación de lectura o escritura de memoria:

- ***Memory address register*** **(MAR)**: registro de direcciones de memoria, donde ponemos la dirección de memoria a la que queremos acceder.
- ***Memory buffer register*** **(MBR)**: registro de datos de memoria; registro donde la memoria deposita el dato leído o el dato que queremos escribir.

La manera de acceder a memoria utilizando estos registros es la siguiente:

1) En una **operación de lectura**, se realiza la secuencia de operaciones siguiente:

a) El procesador carga en el registro MAR la dirección de la posición de memoria que se quiere leer.
b) El procesador coloca en las líneas de direcciones del bus el contenido del MAR y activa la señal de lectura de la memoria.
c) El MBR se carga con el dato obtenido de la memoria.

2) En una **operación de escritura**, se realiza la secuencia de operaciones siguiente:

a) El procesador carga en el registro MBR la palabra que quiere escribir en la memoria.
b) El procesador carga en el registro MAR la dirección de la posición de memoria donde se quiere escribir el dato.
c) El procesador coloca en las líneas de direcciones del bus el contenido del MAR y en las líneas de datos del bus, el contenido del MBR, y activa la señal de escritura de la memoria.

3.4. Registros de estado y de control

La información sobre el estado del procesador puede estar almacenada en un registro o en más de uno, aunque habitualmente suele ser un único registro denominado *registro de estado*.

Los bits del registro de estado son modificados por el procesador como resultado de la ejecución de algunos tipos de instrucciones, por ejemplo instrucciones aritméticas o lógicas, o como consecuencia de algún acontecimiento, como las peticiones de interrupción. Estos bits son parcialmente visibles para el programador, en algunos casos mediante la ejecución de instrucciones específicas.

Cada bit o conjunto de bits del registro de estado indica una información concreta. Los más habituales son:

- **Bit de cero**: se activa si el resultado obtenido es 0.
- **Bit de transporte**: se activa si en el último bit que operamos en una operación aritmética se produce transporte; también puede deberse a una operación de desplazamiento.
- **Bit de desbordamiento**: se activa si la última operación ha producido un resultado que no se puede representar en el formato que estamos utilizando.
- **Bit de signo**: se activa si el resultado obtenido es negativo.
- **Bit de interrupción**: indica si las interrupciones están habilitadas o inhibidas.
- **Bit de modo de operación**: indica si la instrucción se ejecuta en modo supervisor o en modo usuario. Hay instrucciones que solo se ejecutan en modo supervisor.
- **Nivel de ejecución**: indica el nivel de privilegio de un programa en ejecución. Un programa puede desalojar el programa que se ejecuta actualmente si su nivel de privilegio es superior.

Los **registros de control** son los que dependen más de la organización del procesador. En estos registros se almacena la información generada por la unidad de control y también información específica para el sistema operativo. La información almacenada en estos registros no es nunca visible para el programador de aplicaciones.

4. Unidad aritmética y lógica

La unidad aritmética y lógica o ALU[2]es un circuito combinacional capaz de realizar operaciones aritméticas y lógicas con números enteros y también con números reales. Las operaciones que puede efectuar vienen definidas por el conjunto de instrucciones aritméticas y lógicas de las que dispone el juego de instrucciones del computador.

(2)*ALU* es la abreviatura de *arithmetic logic unit*.

Veamos primero cómo se representan los valores de los números enteros y reales con los que puede trabajar la ALU y, a continuación, cuáles son las operaciones que puede hacer.

1) Números enteros. Los números enteros se pueden representar utilizando diferentes notaciones, entre las cuales hay signo magnitud, complemento a 1 y complemento a 2. La notación más habitual de los computadores actuales es el complemento a 2 (Ca2).

Representación de la información

En este módulo no analizaremos en detalle la representación de números ni la manera de operar con ellos. Este tema lo hemos tratado con detenimiento en el módulo "Representación de la información" de la asignatura *Fundamentos de computadores*.

Todas las notaciones representan los números enteros en binario. Según la capacidad de representación de cada computador, se utilizan más o menos bits. El número de bits más habitual en los computadores actuales es de 32 y 64.

2) Números reales. Los números reales se pueden representar básicamente de dos maneras diferentes: en punto fijo y en punto flotante. El computador es capaz de trabajar con números reales representados en cualquiera de las dos maneras.

En la **notación en punto fijo** la posición de la coma binaria es fija y se utiliza un número concreto de bits tanto para la parte entera como para la parte decimal.

En la **notación en punto flotante** se representan utilizando tres campos, esto es: signo, mantisa y exponente, donde el valor del número es $\pm \text{mantisa} \cdot 2^{\text{exponente}}$.

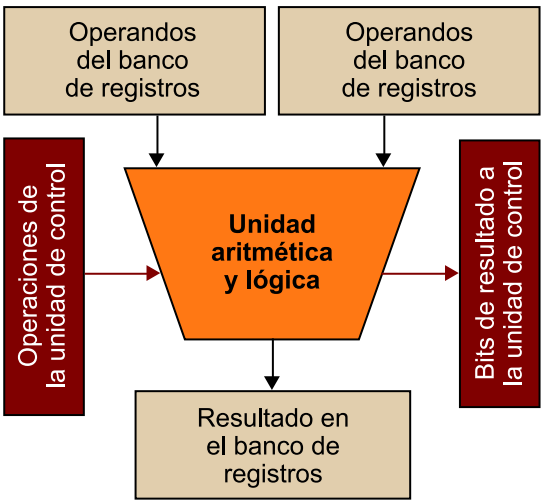
El IEEE ha definido una norma para representar números reales en punto flotante: IEEE-754. La norma define diferentes formatos de representación de números binarios en punto flotante. Los más habituales son los siguientes:

- **Precisión simple**: números binarios en punto flotante de 32 bits, utilizan un bit de signo, 8 bits para el exponente y 23 para la mantisa.
- **Doble precisión**: números binarios en punto flotante de 64 bits, utilizan un bit de signo, 11 bits para el exponente y 52 para la mantisa.
- **Cuádruple precisión**: números binarios en punto flotante de 128 bits, utilizan un bit de signo, 15 bits para el exponente y 112 para la mantisa.

La norma define también la representación del cero y de valores especiales, como infinito y NaN (*not a number*).

Las operaciones aritméticas habituales que puede hacer una ALU incluyen suma, resta, multiplicación y división. Además, se pueden incluir operaciones específicas de incremento positivo (+1) o negativo (–1).

Dentro de las operaciones lógicas se incluyen operaciones AND, OR, NOT, XOR, operaciones de desplazamiento de bits a la izquierda y a la derecha y operaciones de rotación de bits.



En los primeros computadores se implementaba la ALU como una única unidad funcional capaz de hacer las operaciones descritas anteriormente sobre números enteros. Esta unidad tenía acceso a los registros donde se almacenaban los operandos y los resultados de cada operación.

Para hacer operaciones en punto flotante, se utilizaba una unidad específica denominada *unidad de punto flotante*[3] o *coprocesador matemático*, que disponía de sus propios registros y estaba separada del procesador.

[3] En inglés, *floating point unit* (FPU).

La evolución de los procesadores que pueden ejecutar las instrucciones encabalgadamente ha llevado a que el diseño de la ALU sea más complejo, de manera que ha hecho necesario replicar las unidades de trabajo con enteros para permitir ejecutar varias operaciones aritméticas en paralelo y, por otra parte, las unidades de punto flotante continúan implementando en unidades separadas pero ahora dentro del procesador.

5. Unidad de control

La unidad de control se puede considerar el cerebro del computador. Como el cerebro, está conectada al resto de los componentes del computador mediante las señales de control (el sistema nervioso del computador). Con este símil no se pretende humanizar los computadores, sino ilustrar que la unidad de control es imprescindible para coordinar los diferentes elementos que tiene el computador y hacer un buen uso de ellos.

Es muy importante que un computador tenga unidades funcionales muy eficientes y rápidas, pero si no se coordinan y no se controlan correctamente, es imposible aprovechar todas las potencialidades que se habían previsto en el diseño.

Consiguientemente, muchas veces, al implementar una unidad de control, se hacen evidentes las relaciones que hay entre las diferentes unidades del computador y nos damos cuenta de que hay que rediseñarlas, no para mejorar el funcionamiento concreto de cada unidad, sino para mejorar el funcionamiento global del computador.

La función básica de la unidad de control es la ejecución de las instrucciones, pero su complejidad del diseño no se debe a la complejidad de estas tareas (que en general son muy sencillas), sino a la sincronización que se debe hacer de ellas.

Aparte de ver las maneras más habituales de implementar una unidad de control, analizaremos el comportamiento dinámico, que es clave en la eficiencia y la rapidez de un computador.

5.1. Microoperaciones

Como ya sabemos, ejecutar un programa consiste en ejecutar una secuencia de instrucciones, y cada instrucción se lleva a cabo mediante un ciclo de ejecución que consta de las fases principales siguientes:

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

Cada una de las operaciones que hacemos durante la ejecución de una instrucción la denominamos *microoperación*, y estas microoperaciones son la base para diseñar la unidad de control.

5.1.1. Tipos de microoperaciones

La función básica de las microoperaciones es la transferencia de información de un lugar del computador a otro, generalmente de un registro a otro, tanto si son internos al procesador como externos. Este proceso de transferencia puede implicar solo mover la información pero también transformarla. Identificamos tres tipos básicos de microoperaciones:

1) Transferencia interna: operaciones de transferencia entre registros internos del procesador.

2) Transferencia interna con transformación: operaciones aritméticas o lógicas utilizando registros internos del procesador.

3) Transferencia externa: operaciones de transferencia entre registros internos del procesador y registros externos al procesador o módulos externos al procesador (como el bus del sistema o la memoria principal).

Ejemplos de transferencia

Una transferencia interna puede consistir en cargar el contenido del registro PC en el registro MAR para obtener la siguiente instrucción que hemos de ejecutar; una transferencia interna con transformación de información puede consistir en incrementar un registro, llevando el contenido del registro a la ALU y recoger el resultado para guardarlo en otro registro, y una transferencia externa puede consistir en llevar el contenido de un registro de estado de un dispositivo de E/S a un registro del procesador.

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

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

Nota

Los registros, tanto de origen como de destino, pueden ser también de módulos externos al procesador.

5.1.2. Ciclo de ejecución

Las microoperaciones sirven de guía para diseñar la unidad de control, pero antes de entrar en el detalle de la implementación, analizaremos la secuencia de microoperaciones que habitualmente se producen en cada fase del ciclo de ejecución de las instrucciones.

Esta secuencia puede variar de una arquitectura a otra e, incluso, puede haber microoperaciones que estén en fases diferentes. Eso depende en buena parte de las características de la arquitectura: el número de buses, a qué buses tienen acceso los diferentes registros, si hay unidades funcionales específicas como registros que se puedan autoincrementar sin hacer uso de la ALU, la manera de acceder a los elementos externos al procesador, etc.

A continuación veremos las microoperaciones que se llevan a cabo en cada una de las fases del ciclo de ejecución para una arquitectura genérica desde el punto de vista funcional: cuáles se deben realizar y en qué orden. En el próximo apartado analizaremos con más detalle la dependencia temporal entre las microoperaciones en razón de los recursos que ha utilizado cada una.

Lectura de la instrucción

La fase de lectura de la instrucción consta básicamente de cuatro pasos:

1) MAR ← PC: se pone el contenido del registro PC en el registro MAR.

2) MBR ← Memoria: se lee la instrucción.

3) PC ← PC + Δ : se incrementa el PC tantas posiciones de memoria como se han leído (Δ posiciones).

4) IR ← MBR: se carga la instrucción en el registro IR.

Hay que tener presente que si la instrucción tiene un tamaño superior a una palabra de memoria, este proceso se debe repetir tantas veces como sea necesario.

Las diferencias principales que encontramos entre diferentes máquinas en esta fase son cómo y cuándo se incrementa el PC, ya que en algunas máquinas se utiliza la ALU y en otras se puede utilizar un circuito incrementador específico para el PC.

La información almacenada en el registro IR se descodifica para identificar las diferentes partes de la instrucción y determinar las operaciones necesarias que hay que efectuar en las fases siguientes.

Lectura de los operandos fuente

El número de pasos que hay que hacer en esta fase depende del número de operandos fuente y de los modos de direccionamiento utilizados en cada operando. Si hay más de un operando, hay que repetir el proceso para cada uno de los operandos.

El modo de direccionamiento indica el lugar en el que está el dato:

- Si el dato está en la instrucción misma, no hay que hacer nada porque ya lo tenemos en la misma instrucción.
- Si el dato está en un registro, no hay que hacer nada porque ya lo tenemos disponible en un registro dentro del procesador.
- Si el dato está en la memoria, hay que llevarlo al registro MBR.

Ejemplo

Veamos ahora algún ejemplo de ello:

- Inmediato: el dato está en la misma instrucción y, por lo tanto, no hay que hacer nada: IR (operando).
- Directo a registro: el dato está en un registro y, por lo tanto, no hay que hacer nada.
- Relativo a registro índice:
  - **MAR ← IR(Dirección operando) + Contenido IR(registro índice)**
  - **MBR ← Memoria**: leemos el dato.

IR(campo)

Campo, en IR(campo), es uno de los campos de la instrucción que acabamos de leer y que tenemos guardado en el registro IR.

La mayor parte de los juegos de instrucciones no permiten especificar dos operandos fuente con acceso a la memoria, ya que el dato obtenido se deja en el MBR y, por lo tanto, si se especificaran dos operandos de memoria, se debería guardar el primero temporalmente en otro registro, lo que causaría un retraso considerable en la ejecución de la instrucción.

En arquitecturas con un único bus interno (o de más de un bus, pero con determinadas configuraciones de acceso a los buses) también hay que añadir microoperaciones para guardar temporalmente información en registros cuando se trabaja con más de un dato al mismo tiempo, como por ejemplo cuando se hace una suma.

Ejecución de la instrucción y almacenamiento del operando de destino

El número de pasos que hay que realizar en esta fase depende del código de operación de la instrucción y del modo de direccionamiento utilizado para especificar el operando de destino. Se necesita, por lo tanto, una descodificación para obtener esta información.

Para ejecutar algunas instrucciones es necesaria la ALU. Para operar con esta, hay que tener disponibles al mismo tiempo todos los operandos que utiliza, pero la ALU no dispone de elementos para almacenarlos; por lo tanto, se deben almacenar en registros del procesador. Si no hay un bus diferente desde el que se pueda captar cada uno de los operandos fuente y donde se pueda dejar el operando de destino, se necesitan registros temporales (transparentes al programador) conectados directamente a la ALU (entrada y salida de la ALU) y disponibles al mismo tiempo, lo que implica el uso de microoperaciones adicionales para llevar los operandos a estos registros temporales.

Si los operandos fuente se encuentran en registros disponibles para la ALU, la microoperación para hacer la fase de ejecución es de la manera siguiente:

Registro de destino ← Registro de origen <operación> Registro de origen o Valor

Si el operando de destino no es un registro, hay que resolver antes el modo de direccionamiento para almacenar el dato, de manera muy parecida a la lectura del operando fuente.

Veamos cómo se resolverían algunos de estos modos de direccionamiento.

Directo a memoria:

- MBR ← <Resultado ejecución>
- MAR ← IR(Dirección operando)
- Memoria ← MBR

Relativo a registro base:

- MBR ← <Resultado ejecución>
- MAR ← Contenido IR(RB) + IR(Desplazamiento operando)
- Memoria ← MBR

Hay que tener presente que en muchas arquitecturas el operando de destino es el mismo que uno de los operandos fuente y, por lo tanto, los cálculos consecuencia del modo de direccionamiento ya están resueltos, es decir, ya se sabe dónde se guarda el dato y no hay que repetirlo.

Comprobación de interrupciones

En esta fase, si no se ha producido ninguna petición de interrupción, no hay que ejecutar ninguna microoperación y se continúa con la ejecución de la instrucción siguiente; en el caso contrario, hay que hacer un cambio de contexto. Para hacer un cambio de contexto hay que guardar el estado del procesador (generalmente en la pila del sistema) 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.

- **MBR** ← **PC**: se pone el contenido del PC en el registro MBR.
- **MAR** ← **Dirección de salvaguarda**: se indica dónde se guarda el PC.
- **Memoria** ← **MBR**: se guarda el PC en la memoria.

- **PC ← Dirección de la rutina**: se posiciona el PC al inicio de la rutina de servicio de la interrupción.

Ved también

Este punto lo veremos con más detalle en el módulo "Sistema de entrada/salida" de esta misma asignatura.

5.2. Señales de control y temporización

Hemos visto que cada microoperación hace una tarea determinada dentro del ciclo de ejecución de una instrucción. A pesar de la simplicidad de estas microoperaciones, llevarlas a cabo implica la activación de un conjunto de señales de control.

De manera general, entendemos una **señal de control** como una línea física que sale de la unidad de control y va hacia uno o más dispositivos del computador por los que circula una señal eléctrica que representa un valor lógico 0 o 1 y según cuáles sean los dispositivos a los que está conectado, es activo por flanco o por nivel.

Ved también

Los conceptos relacionados con las señales eléctricas los hemos trabajado en el módulo de circuitos lógicos de la asignatura *Fundamentos de computadores* y no los analizaremos en más detalle.

La mayor parte de los computadores tienen un funcionamiento síncrono, es decir, la secuencia de operaciones es gobernada por una señal de reloj. El período de esta señal de reloj (el tiempo que tarda en hacer una oscilación entera), llamado también *ciclo de reloj*, determina el tiempo mínimo necesario para hacer una operación elemental en el computador. Consideraremos que esta operación elemental es una microoperación.

Operación elemental y microoperación

La identificación "operación elemental – microoperación", en los computadores actuales, no siempre es tan fácilmente generalizable, en gran medida debido a los conceptos más avanzados de arquitectura del computador que no hemos tratado y que, por lo tanto, no tendremos en cuenta.

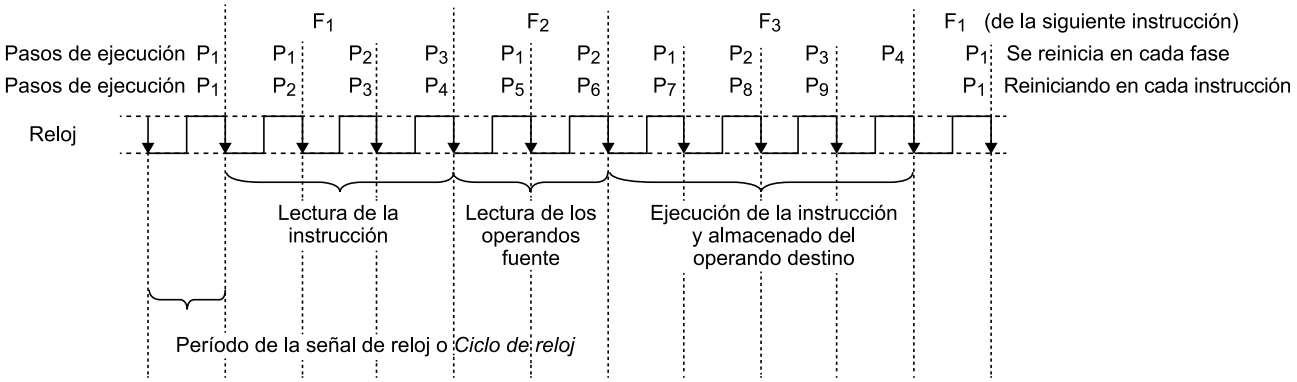
Para llevar a cabo el control de la ejecución de una instrucción, hay que realizar las cuatro fases siguientes:

- F_1: lectura de la instrucción.
- F_2: lectura de los operandos fuente.
- F_3: ejecución de la instrucción y almacenamiento del operando de destino.
- F_4: comprobación de interrupciones.

En cada una de estas fases, se efectúa un conjunto de microoperaciones y, por lo tanto, se requiere un ciclo o más de un ciclo de reloj para ejecutarlas (o ninguno si no se hace ninguna microoperación en aquella fase). Para controlar la ejecución de las microoperaciones en cada fase, se divide cada fase en una secuencia de pasos de ejecución.

Un paso de ejecución (P_i) es el conjunto de microoperaciones que se pueden ejecutar simultáneamente en un ciclo de reloj. El número de pasos de ejecución que se realizan en cada fase puede ser diferente.

El hecho de dividir el ciclo de ejecución en fases y pasos permite sistematizar el funcionamiento de la unidad de control y simplificar su diseño. Al hacerlo de esta manera, para controlar las fases y los pasos de ejecución, solo es necesario tener un contador para las fases que se reinicia al empezar cada instrucción y un contador de pasos que, en algunas máquinas, se reinicia al empezar cada instrucción y en otras se reinicia al comenzar cada fase.



Para optimizar el tiempo de ejecución de cada fase y, por lo tanto, de la instrucción, hay que intentar ejecutar simultáneamente dos o más microoperaciones en un mismo paso de ejecución. Hasta ahora, hemos visto, por una parte, que la ejecución de las microoperaciones implica la utilización de determinados recursos del computador y, por otra parte, que estas microoperaciones se ejecutan durante cierto tiempo, generalmente un ciclo de reloj. Para asegurarnos de que dos o más microoperaciones se pueden ejecutar al mismo tiempo, debemos tener en cuenta qué recursos se utilizan y durante cuántos ciclos de reloj.

CISCA

En el módulo "Arquitectura CISCA" explicamos con detalle la secuencia de microoperaciones de la ejecución de una instrucción concreta.

La agrupación de microoperaciones debe seguir básicamente dos reglas:

1) Debe seguir la secuencia correcta de acontecimientos. No se puede ejecutar una microoperación que genera u obtiene un dato al mismo tiempo que otra microoperación que lo ha de utilizar.

Por ejemplo, no se puede hacer una suma al mismo tiempo que se lee uno de los operandos de memoria, ya que no se puede asegurar que este operando esté disponible hasta el final del ciclo y, por lo tanto, no se puede garantizar que la suma se haga con el dato esperado.

2) Hay que evitar los conflictos. No se puede utilizar el mismo recurso para dos microoperaciones diferentes.

Por ejemplo, no se puede utilizar el mismo bus para transferir dos datos o direcciones diferentes al mismo tiempo.

5.3. Unidad de control cableada

Hasta ahora, hemos visto la unidad de control desde un punto de vista funcional: entradas, salidas, tareas que hace y cómo interacciona con el resto de los elementos del computador. A continuación nos centraremos en las técnicas de diseño e implementación de la unidad de control.

Estas técnicas se pueden clasificar en dos categorías:

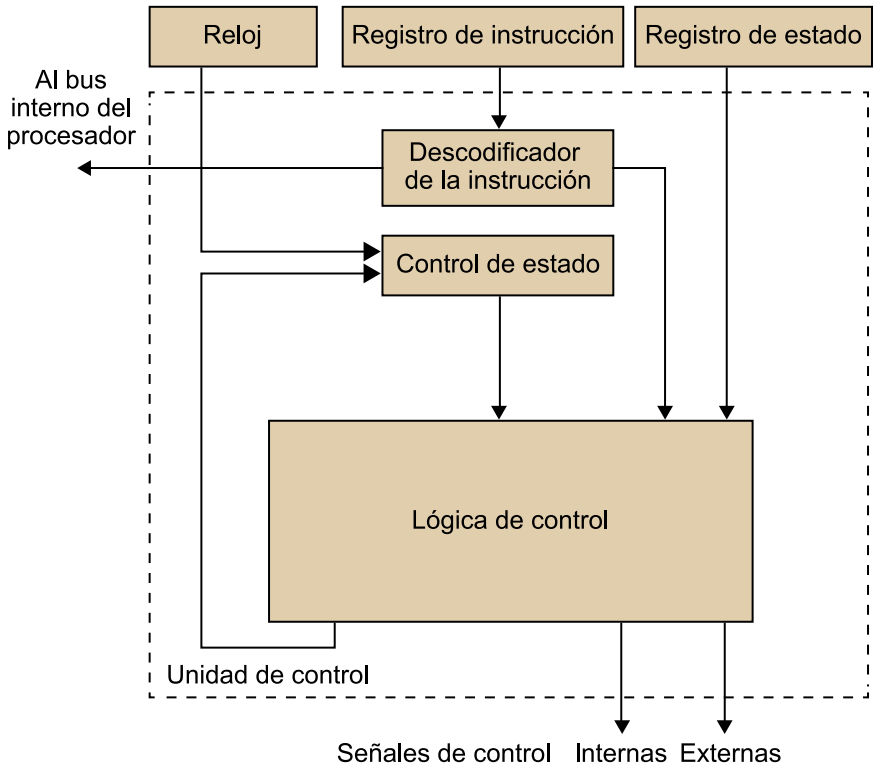
- Control cableado.
- Control microprogramado.

La unidad de control cableada es, básicamente, un circuito combinacional que recibe un conjunto de señales de entrada y lo transforma en un conjunto de señales de salida que son las señales de control. Esta técnica es la que se utiliza típicamente en máquinas RISC.

Para analizar con más detalle la unidad de control cableada, estudiaremos los elementos básicos que forman parte de ella.

5.3.1. Organización de la unidad de control cableada

En el diagrama siguiente se muestran los módulos que forman la unidad de control y la interconexión entre unos y otros.



A continuación, se describen detalladamente cada uno de los módulos que componen esta unidad de control:

1) Descodificador de la instrucción. La función básica de este módulo es descodificar la instrucción. Obtiene la información del registro IR y proporciona una salida independiente para cada instrucción de máquina, y también se encarga de suministrar los valores inmediatos (especificados en la instrucción misma) al bus interno del procesador para trabajar con estos valores.

2) Control de estado. La función de este módulo es contar los pasos dentro de cada fase del ciclo de ejecución de la instrucción. Recibe la señal de reloj (secuencia repetitiva de impulsos que se utiliza para delimitar la duración de las microoperaciones) y las señales de la lógica de control necesarias para controlar las fases del ciclo de ejecución, y genera una señal diferente para cada paso dentro de cada fase del ciclo de ejecución y las señales para identificar la fase dentro del ciclo de ejecución.

3) Lógica de control. Este módulo es un circuito combinacional que efectúa las operaciones lógicas necesarias para obtener las señales de control correspondientes a una microoperación. Recibe las señales de los otros dos módulos de la unidad de control cableada y del registro de estado y suministra información del resultado de esta microoperación al módulo que efectúa el control de estado.

Este circuito se puede implementar mediante dos niveles de puertas lógicas, pero a causa de diferentes limitaciones de los circuitos lógicos se hace con más niveles. Sin embargo, continúa siendo muy rápido. Hay que tener presente que las técnicas modernas de diseño de circuitos VLSI facilitan mucho el diseño de estos circuitos.

Ejemplo

Utilizando de referencia las señales de control de la arquitectura CISCA, y la secuencia de microoperaciones que tienen para ejecutar las instrucciones:

$$\text{MBRoutA} = F_1 \cdot P_3 + I_k \cdot F_3 \cdot P_1 + ...$$

Donde I_k son las instrucciones de transferencia del juego de instrucciones donde se transfiera un dato que se ha leído de memoria a un registro.

De esta expresión interpretamos que la señal de control MBRoutA es activa cuando estamos en el paso 3 de la fase 1 de cualquier instrucción (no especifica ninguna, ya que la fase de lectura de la instrucción es común a todas las instrucciones), o en el paso 1 de la fase 3 de la instrucción I_k, o en el paso..., y se continúa de la misma manera para el resto de los casos en los que es necesario activar la señal.

Como se puede ver, es un trabajo muy complejo y laborioso y el hecho de rediseñar el computador puede implicar cambiar muchas de estas expresiones.

Nota

No analizaremos con más detalle esta implementación cableada, ya que tendríamos que analizar más exhaustivamente la problemática del diseño de circuitos lógicos, lo que no es el objetivo de estos materiales. Encontraréis información acerca de la técnica de diseño en bibliografía específica sobre este tema.

5.4. Unidad de control microprogramada

La microprogramación es una metodología para diseñar la unidad de control propuesta por M. V. Wilkes en 1951, que pretende ser un método de diseño organizado y sistemático que evite las complejidades de las implementaciones cableadas.

Esta metodología se basa en la utilización de una memoria de control que almacena las microoperaciones necesarias para ejecutar cada una de las instrucciones y en el concepto de **microinstrucción** que veremos más adelante.

En los años cincuenta la propuesta pareció poco viable, ya que hacía necesario disponer de una memoria de control que fuera, al mismo tiempo, rápida y económica. Con la mejora de velocidad de los circuitos de memoria y su abaratamiento, devino una metodología muy utilizada, especialmente en el diseño de unidades de control complejas, como es el caso de las arquitecturas CISC.

Como ya sabemos, la ejecución de una instrucción implica realizar una serie de microoperaciones, y cada microoperación implica la activación de un conjunto de señales de control, de las cuales cada una puede ser representada por un bit.

La representación del conjunto de todas las señales de control (combinación de ceros y unos) da lugar a lo que denominamos **palabra de control**.

Palabra de control

Hay que tener presente que una palabra de control determina todas las señales de control que genera la unidad de control, ya que se utilizan tanto para indicar qué recursos se utilizan como para asegurar que el resto de los recursos no interfieren en la ejecución de las microoperaciones en curso.

Si dos o más microoperaciones se pueden ejecutar en un mismo período, se pueden activar las señales de control necesarias en una misma palabra de control.

5.4.1. Microinstrucciones

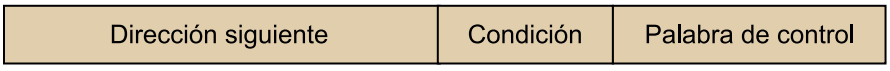
Llamamos **microinstrucción** a la notación utilizada para describir el conjunto de microoperaciones que se realizan simultáneamente en un mismo período y se representan con una palabra de control.

La ejecución de una microinstrucción implica activar las señales de control correspondientes a las microoperaciones que efectúa y determinar la dirección de la microinstrucción siguiente.

Cada microinstrucción se almacena en una memoria de control y, por lo tanto, cada una tiene asignada una dirección de memoria diferente.

Para especificar la secuencia de ejecución de las microinstrucciones, se necesita más información que la misma palabra de control. Esta información adicional da lugar a dos tipos básicos de microinstrucciones:

1) Microinstrucción horizontal. Se añade a la palabra de control un campo para expresar una condición y un campo para especificar la dirección de la microinstrucción siguiente que se tiene que ejecutar cuando se cumpla la condición.



Los bits de la palabra de control representan directamente las señales de control.

2) Microinstrucciones verticales. Para reducir el tamaño de las microinstrucciones horizontales, se codifican los bits de la palabra de control, de manera que se reduce el número de bits necesarios.

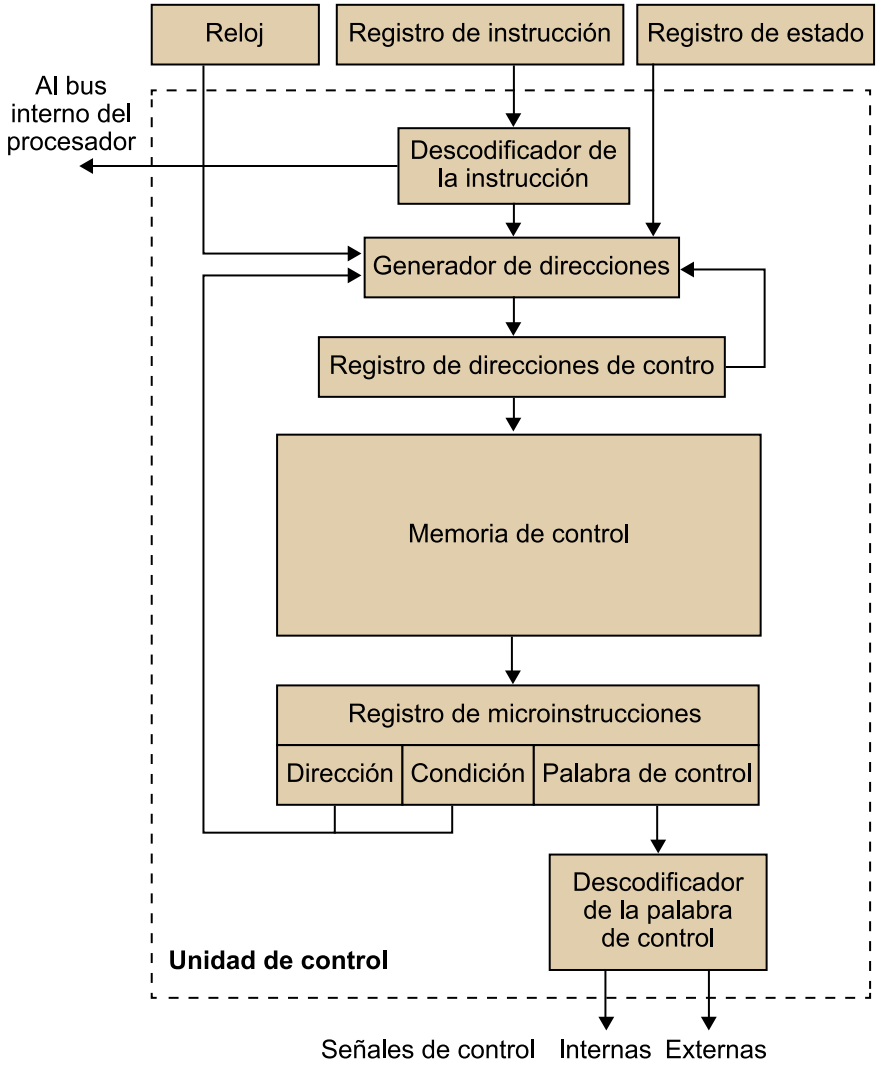
Si tenemos una palabra de control de 2^N bits se puede codificar utilizando N bits. Por ejemplo, una palabra de control de $2^4 = 16$ bits se puede codificar utilizando solo 4 bits.

El inconveniente de este tipo de microinstrucciones es que los bits de la palabra de control no representan directamente las señales de control, y hace falta un descodificador para traducir los códigos que aparecen en la microinstrucción de señales de control individuales.

El conjunto de microinstrucciones necesarias para ejecutar una instrucción da lugar a un **microprograma** (microcódigo o *firmware*). Como cada instrucción realiza tareas diferentes, es necesario un microprograma diferente para cada instrucción del juego de instrucciones de la arquitectura.

5.4.2. Organización de una unidad de control microprogramada

En el diagrama siguiente se muestran los módulos que forman la unidad de control y la interconexión entre unos y otros.



A continuación, se describe con detalle cada uno de los módulos que componen esta unidad de control:

1) Descodificador de la instrucción. Este módulo identifica los diferentes campos de la instrucción y envía la información necesaria al generador de direcciones para el control de los microprogramas (generalmente, la dirección de inicio del microprograma de la instrucción en curso) y suministra los valores inmediatos (especificados en la misma instrucción) al bus interno del procesador para que pueda trabajar con ellos.

2) Generador de direcciones. Este módulo genera la dirección de la microinstrucción siguiente y la carga en el registro de direcciones de control.

3) Registro de direcciones de control. En este registro se carga la dirección de memoria de la microinstrucción siguiente que se ejecutará.

4) Memoria de control. Almacena el conjunto de microinstrucciones necesarias para ejecutar cada instrucción o microprograma.

5) Registro de microinstrucciones. Este registro almacena la microinstrucción que se acaba de leer de la memoria de control.

6) Descodificador de la palabra de control. Este módulo, cuando se utilizan microinstrucciones verticales, traduce la palabra de control incluida en la microinstrucción a las señales de control individuales que representa.

5.4.3. Funcionamiento de la unidad de control microprogramada

Las dos tareas que hace la unidad de control microprogramada son la secuenciación de las microinstrucciones y la generación de las señales de control.

Secuenciación de las microinstrucciones

Una de las tareas más complejas que hace la unidad de control microprogramada es determinar la dirección de la microinstrucción siguiente que se ha de ejecutar, tarea que lleva a cabo el generador de direcciones.

La dirección de la microinstrucción siguiente la determinan estos factores:

- **El registro de instrucción**, que solo se utiliza para determinar el microprograma propio de cada instrucción.

- **El registro de estado**, que solo se utiliza cuando hay que romper la secuencia normal de ejecución de un programa, como sucede cuando se han de evaluar las condiciones en una instrucción de salto condicional.

- **El campo de condición de la microinstrucción**. Es el factor que se utiliza más frecuentemente. Este es el factor que hay que evaluar para saber si se debe continuar en secuencia o hacer una bifurcación:
  - Si la condición indicada es falsa, la microinstrucción siguiente que se ejecutará es la que sigue en secuencia y la dirección de esta microinstrucción puede estar especificada en la misma microinstrucción o se puede obtener incrementando el valor actual del registro de direcciones de control.

  - Si la condición indicada es cierta, la microinstrucción siguiente que se ejecutará es la que indica la dirección de bifurcación y esta dirección se obtiene de la misma microinstrucción.

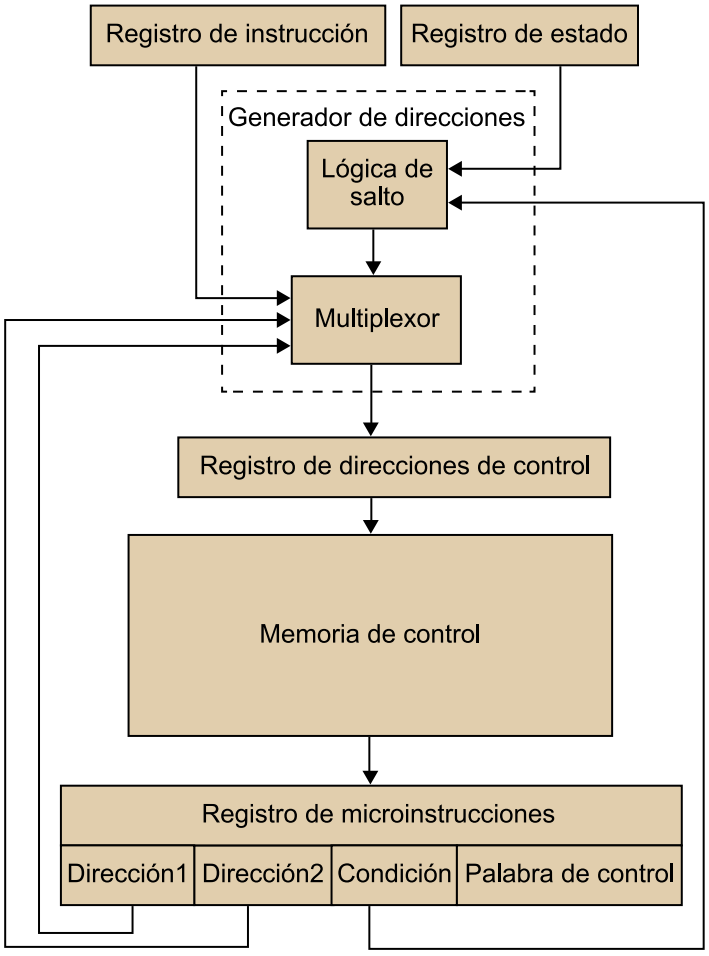
Formatos del campo de dirección y condición de la microinstrucción

El campo de dirección y condición de las microinstrucciones puede tener los formatos siguientes:

1) Dos campos explícitos. En la microinstrucción se especifican dos direcciones explícitas: la dirección de la microinstrucción que sigue en secuencia y una dirección de bifurcación. Según si se cumple o no la condición, se carga una dirección o la otra en el registro de direcciones de control.

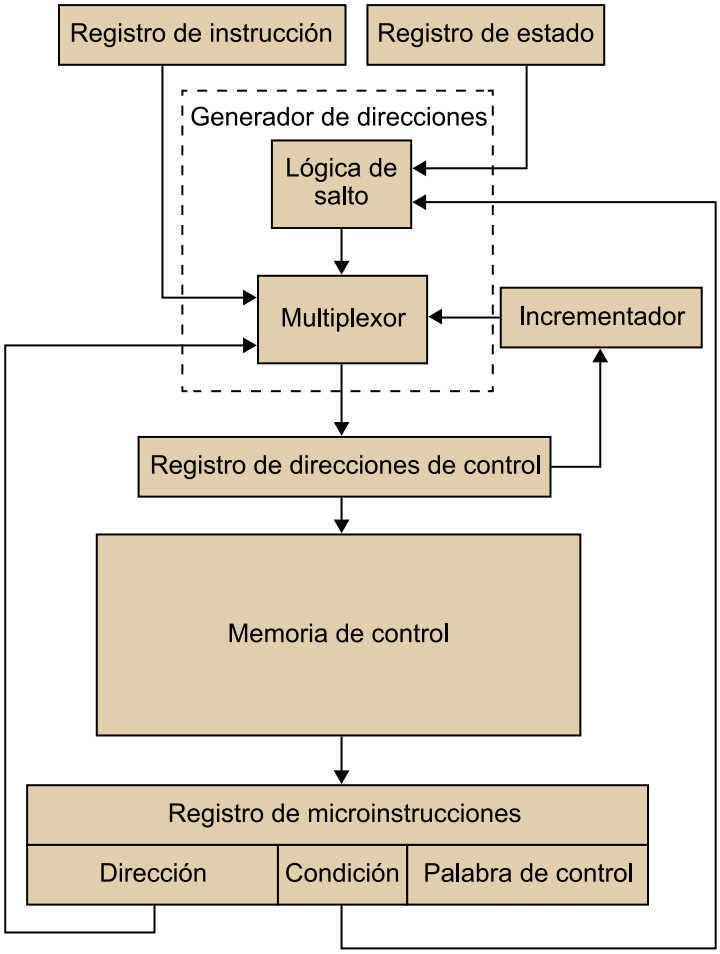
Con esta técnica no se necesita una lógica para incrementar la dirección actual y pasar a la microinstrucción siguiente que sigue en secuencia pero requiere más bits que otros formatos.

Ejemplo de circuito generador de direcciones con dos campos explícitos



2) Un campo explícito. En la microinstrucción se especifica explícitamente una dirección: cuando se cumple la condición indicada, se carga el registro de direcciones de control con la dirección explícita. Cuando no se cumple la condición, se carga el registro de direcciones de control con el valor incrementado del registro de direcciones de control.

Ejemplo de circuito generador de direcciones con un campo explícito



3) Microinstrucciones de formato variable. En este formato solo están el campo de condición y el campo de palabra de control. En las microinstrucciones de salto, se utiliza una parte del campo de la palabra de control como dirección de bifurcación, lo que obliga a utilizar un bit de la microinstrucción para indicar si es necesario interpretar esta parte del campo de la palabra de control como la dirección de la microinstrucción siguiente o como parte de la palabra de control.

La ventaja principal de este método es que nos permite reducir el tamaño de las microinstrucciones y no complica excesivamente la lógica de control de la unidad de control.

Generación de las señales de control

Generalmente las unidades de control microprogramadas no utilizan un formato totalmente horizontal de la palabra de control, sino un cierto nivel de codificación para ahorrar espacio en la memoria de control.

Lo más habitual es agrupar los bits de la palabra de control en campos; cada campo contiene un código que, una vez descodificado, representa un conjunto de señales de control.

Un campo con k bits puede contener 2^k códigos diferentes. Cada código simboliza un estado diferente de las señales de control que representa e indica, para cada una, la activación o la desactivación.

Consideramos que las señales de control son activas si el bit que las representa vale 1 e inactivas si este bit vale 0.

A la hora de elegir una codificación debemos tener presente cuáles son las diferentes combinaciones de señales de control que tienen que ser activas o no en un mismo instante.

La codificación puede ser *directa* (un solo nivel de codificación) o *indirecta* (dos niveles de codificación). En el segundo caso el valor descodificado que toma un campo sirve para saber qué señales de control representa un segundo campo, ya que pueden ser diferentes según el valor que tome el primer campo.

La agrupación de los bits de la palabra de control en campos puede responder a diferentes criterios; lo más habitual es agruparlos según los recursos que representan: se tratan todos los recursos de la máquina de manera independiente y se asigna un campo a cada uno: un campo para la ALU, un campo para la entrada/salida, un campo para la memoria, etc.

5.5. Comparación: unidad de control microprogramada y cableada

A continuación presentamos las ventajas e inconvenientes de la unidad de control microprogramada respecto a la unidad de control cableada.

Las ventajas son:

- Simplifica el diseño.
- Es más económica.
- Es más flexible:
  - Adaptable a la incorporación de nuevas instrucciones.
  - Adaptable a cambios de organización, tecnológicos, etc.

- Permite disponer de un juego de instrucciones con gran número de instrucciones. Solo hay que poseer una memoria de control de gran capacidad.
- Permite disponer de varios juegos de instrucciones en una misma máquina (emulación de máquinas).
- Permite ser compatible con máquinas anteriores de la misma familia.

- Permite construir máquinas con diferentes organizaciones pero con un mismo juego de instrucciones.

Los inconvenientes son:

- Es más lenta: implica acceder a una memoria e interpretar las microinstrucciones necesarias para ejecutar cada instrucción.
- Es necesario un entorno de desarrollo para los microprogramas.
- Es necesario un compilador de microprogramas.

Intel y AMD

Intel y AMD utilizan diseños diferentes para sus procesadores con organizaciones internas diferentes pero, gracias a la utilización de unidades de control microprogramadas, los dos trabajan con el mismo juego de instrucciones.

6. Computadores CISC y RISC

En el diseño del procesador hay que tener en cuenta diferentes principios y reglas, con vistas a obtener un procesador con las mejores prestaciones posibles.

Las dos alternativas principales de diseño de la arquitectura del procesador son las siguientes:

- Computadores CISC[4], cuyas características son:
  - El formato de instrucción es de longitud variable.
  - Dispone de un gran juego de instrucciones, habitualmente más de cien, para dar respuesta a la mayoría de las necesidades de los programadores.
  - Dispone de un número muy elevado de modos de direccionamiento.
  - Es una familia anterior a la de los procesadores RISC.
  - La unidad de control es microprogramada; es decir, la ejecución de instrucciones se realiza descomponiendo la instrucción en una secuencia de microinstrucciones muy simples.
  - Procesa instrucciones largas y de tamaño variable, lo que dificulta el procesamiento simultáneo de instrucciones.

(4) *CISC* es la abreviatura de *complex instruction set computer*; en español, computador con un juego de instrucciones complejo.

- Computadores RISC[5], que tienen las características siguientes:
  - El formato de instrucción es de tamaño fijo y corto, lo que permite un procesamiento más fácil y rápido.
  - El juego de instrucciones se reduce a instrucciones básicas y simples, con las que se deben implementar todas las operaciones complejas. Una instrucción de un procesador CISC se tiene que escribir como un conjunto de instrucciones RISC.
  - Dispone de un número muy reducido de modos de direccionamiento.
  - La arquitectura es de tipo *load-store* (carga y almacena) o registro-registro. Las únicas instrucciones que tienen acceso a memoria son LOAD y STORE, y el resto de las instrucciones utilizan registros como operandos.
  - Dispone de un amplio banco de registros de propósito general.
  - Casi todas las instrucciones se pueden ejecutar en pocos ciclos de reloj.
  - Este tipo de juego de instrucción facilita la segmentación del ciclo de ejecución, lo que permite la ejecución simultánea de instrucciones.
  - La unidad de control es cableada y microprogramada.

(5) *RISC* es la abreviatura de *reduced instruction set computer*; en español, computador con un juego de instrucciones reducido.

La segmentación permite que la ejecución de una instrucción empiece antes de acabar la de la anterior (se encabalgan las fases del ciclo de ejecución de varias instrucciones), gracias a que se reduce el tiempo de ejecución de las instrucciones.

Los procesadores actuales no son completamente CISC o RISC. Los nuevos diseños de una familia de procesadores con características típicamente CISC incorporan características RISC, de la misma manera que las familias con características típicamente RISC incorporan características CISC.

Ejemplo

PowerPC, que se puede considerar un procesador RISC, incorpora características CISC, tales como un juego de instrucciones muy amplio (más de doscientas instrucciones).

Los últimos procesadores de Intel (fabricante de procesadores típicamente CISC) también incorporan características RISC.

Resumen

La función principal del procesador es procesar los datos y transferirlos a los otros elementos del computador. Está formado por los elementos básicos siguientes:

- Conjunto de registros.
- Unidad aritmética y lógica.
- Unidad de control.

El ciclo de ejecución de la instrucción se divide en cuatro fases. En la tabla se pueden ver de manera esquemática las operaciones que se realizan en cada fase:

Inicio del ciclo de ejecución	
Fase 1: lectura de la instrucción	Leer la instrucción.
	Descodificar la instrucción.
	Actualizar el PC.
Fase 2: lectura de los operandos fuente	Calcular la dirección y leer el primer operando fuente.
	Calcular la dirección y leer el segundo operando fuente.
Fase 3: ejecución de la instrucción y almacenamiento del operando de destino	Ejecutar la instrucción.
Fase 4: comprobación de interrupciones	Comprobar si algún dispositivo ha solicitado una interrupción.

Los registros se han clasificado en los tipos siguientes:

- Registros de propósito general.
- Registros de instrucción.
- Registros de acceso a memoria.
- Registros de estado y de control.

La unidad aritmética y lógica hace operaciones aritméticas y lógicas con números enteros y números reales en punto fijo y en punto flotante.

Hemos estudiado con detalle la unidad de control y hemos visto los aspectos siguientes:

- Microoperaciones.
- Señales de control y temporización.

- Unidad de control cableada.
- Unidad de control microprogramada.
- Ventajas e inconvenientes de la unidad de control microprogramada con respecto a la implementación cableada.

Finalmente, hemos hablado de las características principales de los computadores CISC y RISC.

Sistema de memoria

Miquel Albert Orenga
Gerard Enrique Manonellas

PID_00177073

Universitat Oberta de Catalunya
www.uoc.edu

Índice

Introducción

Todo computador necesita un sistema de memoria para almacenar los programas que se ejecutan y los datos necesarios para ejecutar estos programas. Desde el punto de vista del programador, sería deseable disponer de cantidades ilimitadas de memoria y de velocidad ilimitada, si fuera posible, para almacenar el programa que se quiere ejecutar y los datos necesarios; eso permitiría al programador hacer la tarea de escribir programas sin tener que enfrentarse a ningún tipo de limitación. Lógicamente, este deseo no es factible y las cantidades de memoria de que dispone un computador tienen una limitación en capacidad y velocidad.

La cantidad de memoria que puede tener un computador responde básicamente a un factor de coste: cuanta más memoria instalada, más elevado es el coste. De manera parecida, la velocidad de la memoria también depende del coste. Las memorias más rápidas tienen un coste más elevado, pero no se puede conseguir toda la velocidad necesaria simplemente incrementando el coste; hay además un factor tecnológico que limita la velocidad de la memoria: no podemos adquirir memoria más rápida que la que está disponible en el mercado en un momento dado.

Existen diferentes tipos de memorias, con capacidades y tiempos de acceso diferentes. En general, cuanta más capacidad de almacenamiento tiene una memoria, mayor es el tiempo de acceso. Es decir, las memorias con gran capacidad son memorias lentas, mientras que las memorias rápidas (tiempo de acceso pequeño) suelen tener poca capacidad de almacenamiento. Las memorias rápidas son más caras que las memorias lentas. Por ello, los diseñadores de computadores deben llegar a un compromiso a la hora de decidir cuánta memoria ponen en sus diseños y de qué velocidad o tiempo de acceso.

En los últimos años, la evolución de la tecnología ha permitido reducir mucho el espacio necesario para almacenar un bit de información. Eso ha originado que el tamaño de las memorias aumente mucho con relación al espacio físico que ocupan y que se reduzca el precio que se ha de pagar por un bit de información. Así, los discos duros han pasado de los 20 Mbytes de capacidad a mediados de década de los ochenta a los 2.000 Gbytes a finales del 2010 (100.000 veces más), aunque ocupan el mismo espacio físico (incluso son un poco más pequeños) y cuestan casi lo mismo. Esto ha representado una reducción importante en el precio por bit. Este ha sido un factor muy importante para que los computadores actuales incorporen mucha más memoria que los computadores de hace treinta años. Por lo tanto, a la hora de diseñar un sistema de memoria, hay que tener presentes las características de capacidad, velocidad (y tiempo de acceso) y coste por bit.

Otras cuestiones también importantes que cabe considerar son la localización, la organización, el método de acceso o la tecnología de fabricación.

Objetivos

Con el estudio de este módulo se pretende que el estudiante alcance los objetivos siguientes:

1. Conocer las características básicas de una memoria.

2. Comprender los conceptos básicos sobre la organización de la jerarquía de memoria de un computador.

3. Analizar cómo se gestionan los datos entre los diferentes niveles de la jerarquía de memorias, especialmente en la memoria caché.

4. Conocer las diferentes tecnologías utilizadas para implementar los diferentes tipos de memorias utilizados en un computador.

1. Características de las memorias

Las características más importantes de los diferentes tipos de memoria son la localización, la capacidad, el método de acceso, la organización de los datos en una memoria, el tiempo de acceso y velocidad, y el coste. Las estudiaremos en este apartado.

1.1. Localización de la memoria

Podemos clasificar los tipos de memoria según su localización dentro del computador. Básicamente, se pueden distinguir: memoria dentro del chip del procesador, memoria interna (memoria en la placa base del computador) y memoria externa.

Dentro del chip del procesador habitualmente están los registros y uno o varios niveles de memoria caché.

La memoria interna corresponde a la memoria principal (memoria RAM del computador) y adicionalmente un nivel de memoria caché o varios.

La memoria externa corresponde a los dispositivos de almacenamiento secundario, como discos duros, unidades ópticas (CD-ROM, DVD, o Blu-ray), unidades de cinta, etc.

Ved también

En este módulo profundizaremos en el estudio del sistema de memoria formado por la memoria que se encuentra en el procesador y la memoria interna, y haremos una revisión más general de la memoria externa.

1.2. Capacidad de la memoria

La capacidad (o tamaño de la memoria) hace referencia a la cantidad de información que se puede almacenar. La unidad utilizada para especificar la capacidad de almacenamiento de información es el byte (1 byte = 8 bits), y a la hora de indicar la capacidad, se utilizan diferentes prefijos que representan múltiplos del byte.

En el sistema internacional de medidas (SI) se utilizan prefijos que representan múltiplos y submúltiplos de una unidad; estos prefijos SI corresponden siempre a potencias de 10.

Prefijos SI

Cada prefijo del sistema internacional recibe un nombre diferente y utiliza un símbolo para representarlo. Los prefijos que utilizaremos más habitualmente son:

10^{-12}	pico (p)
10^{-9}	nano (n)
10^{-6}	micro (μ)

10^{-3}	mili (m)
10^{3}	kilo (K)
10^{6}	mega (M)
10^{9}	giga (G)
10^{12}	tera (T)

Ejemplos de prefijos SI

10^3 bytes = 1.000 bytes = 1 Kilobyte (KB o Kbyte)

10^6 bytes = 10^3 KB = 1.000 KB = 1 Megabyte (MB o Mbyte)

10^9 bytes = 10^3 MB = 1.000 MB = 1 Gigabyte (GB o Gbyte)

10^{12} bytes = 10^3 GB = 1.000 GB = 1 Terabyte (TB o Tbyte)

Ahora bien, en informática, la capacidad de almacenamiento habitualmente se indica en múltiplos que sean potencias de 2; en este caso se utilizan los prefijos definidos por la International Electrotechnical Commission (IEC).

Prefijos IEC

Los prefijos IEC representan múltiplos para las unidades de información bit y byte. Los nombres se han creado añadiendo el término *binario* a los prefijos SI. Por ejemplo *kibi* sería la contracción de *kilo binario*. Los prefijos que utilizaremos más habitualmente son:

2^{10}	kibi (Ki)
2^{20}	mebi (Mi)
2^{30}	gibi (Gi)
2^{40}	tebi (Ti)

Ejemplos de prefijos IEC

2^{10} bytes = 1.024 bytes = 1 KiB (kibibyte)

2^{20} bytes = 1.024 KiB = 1 MiB (mebibyte)

2^{30} bytes = 1.024 MiB = 1 GiB (gibibyte)

2^{40} bytes = 1.024 GiB = 1 TiB (tebibyte)

La industria utiliza mayoritariamente las unidades SI. Por ejemplo, si nos fijamos en las características de un disco duro que se comercialice con 1 TB de capacidad, realmente la capacidad del disco será de 1.000 GB = 1.000.000 MB = 1.000.000.000 KB = 1.000.000.000.000 bytes.

En cambio, cuando conectamos este disco a un computador y mostramos las propiedades del dispositivo, veremos que en la mayoría de los sistemas operativos se nos mostrará la capacidad en unidades IEC; en este caso, 976.562.500 KiB = 953.674 MiB = 931 GiB = 0,91 TiB.

1.3. Métodos de acceso

Cada tipo de memoria utiliza un método a la hora de acceder a las posiciones de memoria. Hay métodos de acceso diferentes característicos de cada tipo de memoria:

1) Secuencial. Se accede desde la última posición a la que se ha accedido, leyendo en orden todas las posiciones de memoria hasta llegar a la posición deseada. El tiempo de acceso depende de la posición a la que se quiere acceder y de la posición a la que se ha accedido anteriormente.

Usos del acceso secuencial

El acceso secuencial se utiliza básicamente en dispositivos de cinta magnética.

2) Directo. La memoria se organiza en bloques y cada bloque de memoria tiene una dirección única, se accede directamente al principio de un bloque y dentro de este se hace un acceso secuencial hasta llegar a la posición de memoria deseada. El tiempo de acceso depende de la posición a la que se quiere acceder y de la última posición a la que se ha accedido.

Usos del acceso directo

El acceso directo es un método de acceso que se utiliza en discos magnéticos.

3) Aleatorio. La memoria se organiza como un vector, en el que cada elemento individual de memoria tiene una dirección única. Se accede a una posición determinada proporcionando la dirección. El tiempo de acceso es independiente de la posición a la que se ha accedido y es independiente de la última posición a la que se ha accedido.

Las operaciones básicas utilizadas cuando trabajamos con la memoria son:

a) Operación de lectura: en esta operación hay que proporcionar a la memoria la dirección donde se encuentra la información deseada. La acción que hace la memoria consiste en suministrar la información contenida en la dirección indicada.

b) Operación de escritura: en esta operación hay que suministrar a la memoria la información que se debe almacenar y la dirección de memoria donde se la quiere almacenar. La acción que se lleva a cabo consiste en registrar la información en la dirección especificada.

Usos del acceso aleatorio

El acceso aleatorio se suele utilizar en memorias RAM y ROM.

4) Asociativo. Se trata de un tipo de memoria de acceso aleatorio donde el acceso se hace basándose en el contenido y no en la dirección. Se especifica el valor que se quiere localizar y se compara este valor con una parte del contenido de cada posición de memoria; la comparación se lleva a cabo simultáneamente con todas las posiciones de la memoria.

Usos del acceso asociativo

Este método de acceso se suele utilizar en las memorias caché.

1.4. Organización de los datos de una memoria

En este subapartado solo nos referiremos a la manera de organizar los datos en memorias que se encuentren en el chip del procesador y en la memoria interna. La organización de la memoria externa se lleva a cabo de manera diferente.

Básicamente, los elementos que hemos de tener en cuenta son los siguientes:

1) Palabra de memoria. Es la unidad de organización de la memoria desde el punto de vista del procesador; el tamaño de la palabra de memoria se especifica en bytes o bits. Es el número de bytes máximo que se pueden leer o escribir en un solo ciclo de acceso a la memoria.

Ejemplo

Memoria de 2Kbytes con una palabra de memoria de 2 bytes. Por lo tanto, necesitaremos 10 bits para poder hacer referencia a las 1.024 (2^{10}) posiciones de memoria que almacenarán 2 bytes (16 bits) cada una.

Memoria interna

Dirección← (10 bits) →	Palabra← (16 bits) →
0000000000	**00000000 00000000**
0000000001	**00000000 00000000**
0000000010	**00000000 00000000**
0000000011	**00000000 00000000**
0000000100	**00000000 00000000**
...	**...**
↑ (1.024 direcciones) ↓	**contenido almacenado en la memoria**
...	**...**
1111111100	**00000000 00000000**
1111111101	**00000000 00000000**
1111111110	**00000000 00000000**
1111111111	**00000000 00000000**

2) Unidad de direccionamiento. La memoria interna se puede ver como un vector de elementos, una colección de datos contiguos, en la que cada dato es accesible indicando su posición o dirección dentro del vector.

La unidad de direccionamiento especifica cuál es el tamaño de cada elemento de este vector; habitualmente a la memoria se accede como un vector de bytes –cada byte tendrá su dirección–, aunque puede haber sistemas que accedan a la memoria como un vector de palabras, en los que cada dirección corresponda a una palabra.

El número de bits utilizados para especificar una dirección de memoria fija el límite máximo de elementos dirigibles, el tamaño del mapa de memoria; si tenemos n bits para las direcciones de memoria, el número máximo de elementos dirigibles será de 2^n.

3) Unidad de transferencia. En un acceso a memoria se puede acceder a un byte o a varios, con un máximo que vendrá determinado por el número de bytes de una palabra de memoria; es decir, en un solo acceso se leen o escriben uno o varios bytes.

Cuando se especifica la dirección de memoria a la que se quiere acceder, se accede a partir de esta dirección a tantos bytes como indique la operación de lectura o escritura.

En memoria externa, se accede habitualmente a un bloque de datos de tamaño muy superior a una palabra. En discos es habitual transferir bloques del orden de los Kbytes.

Ejemplo

En los procesadores x86 de 32 y 64 bits, la unidad de direccionamiento es de un byte, pero el tamaño de la palabra de memoria es de 4 bytes (32 bits).

Los registros del procesador (accesibles para el programador) habitualmente tienen un tamaño igual al tamaño de la palabra de memoria; por ejemplo, en un procesador de 32 bits (como el Intel 386) el tamaño de los registros era de 32 bits (4 bytes).

Los procesadores x86-64 son procesadores con registros de 64 bits, pero en cambio el tamaño de la palabra de memoria continúa siendo de 32 bits; eso es así para mantener la compatibilidad con procesadores anteriores. No hay que olvidar que la arquitectura x86-64 es una extensión de la arquitectura de 32 bits x86-32.

El tamaño de la palabra de memoria de los procesadores x86 de 32 y 64 bits es de 32 bits (4 bytes) y en un ciclo de memoria se puede acceder a 1, 2 o 4 bytes.

En la arquitectura CISCA el tamaño de palabra es también de 32 bits pero accedemos siempre a una palabra de 4 bytes.

1.4.1. Ordenación de los bytes en memoria

Aunque normalmente la unidad de direccionamiento de la memoria es el byte, es habitual que se puedan hacer accesos a memoria en múltiplos de byte, hasta el tamaño de la palabra (2, 4, e incluso 8 bytes). En este caso, solo se indica la dirección del primer byte de la palabra y se utilizan dos métodos a la hora de acceder a la palabra:

- ***Big-endian***: la dirección especificada corresponde al byte de más peso de la palabra.
- ***Little-endian***: la dirección especificada corresponde al byte de menos peso de la palabra.

Ejemplo

Supongamos una memoria de capacidad reducida (solo 256 bytes) en la que el tamaño de la palabra a la que se puede acceder es de 16 bits (2 bytes). Las direcciones de memoria serán de 8 bits; para acceder a una palabra de memoria se indica solo la dirección del primer byte.

Dirección		Valor
0	00000000	01100110
1	00000001	11100011
...	...	...
14	**00001110**	**00000000**
15	**00001111**	**11111111**
...	...	...
254	11111110	00001111
255	11111111	11001100

Si indicamos la dirección 00001110 (dirección 14), obtenemos los valores de las posiciones de memoria 14 y 15 así:

1) ***Little-endian***: la dirección 14 corresponde al byte de menos peso de la palabra de 16 bits. Obtenemos el valor:

1111111100000000 (direcciones 15 y 14)

2) ***Big-endian***: la dirección 14 corresponde al byte de más peso de la palabra de 16 bits. En este caso obtenemos el valor:

0000000011111111 (direcciones 14 y 15)

1.5. Tiempo de acceso y velocidad

En memorias de acceso aleatorio, memoria RAM, el **tiempo de acceso** (o **latencia**) es el tiempo que transcurre desde que una dirección de memoria es visible para los circuitos de la memoria hasta que el dato está almacenado (escritura) o está disponible para ser utilizado (lectura). En memorias de acceso no aleatorio (discos) es el tiempo necesario para que el mecanismo de lectura o escritura se sitúe en la posición necesaria para empezar la lectura o escritura.

En memorias de acceso aleatorio, el tiempo de un **ciclo de memoria** se considera el tiempo de acceso más el tiempo necesario antes de que se pueda empezar un segundo acceso a la memoria.

Finalmente, la **velocidad de transferencia** es la velocidad a la que se puede leer o escribir un dato de memoria. En las memorias de acceso aleatorio será el inverso del tiempo de ciclo.

Unidades de la velocidad de transferencia

La velocidad de transferencia se mide en bytes por segundo; es habitual indicar la velocidad de una memoria en MiB/segundo o GiB/segundo.

1.6. Coste

Consideramos el coste por unidad de almacenamiento (coste por bit). Podemos observar que existe una relación directamente proporcional entre la velocidad y el coste/bit: a medida que aumenta la velocidad aumenta también el coste/bit. Eso implica que, con un presupuesto fijado, podremos adquirir memorias muy rápidas pero relativamente pequeñas, o memorias más lentas, pero de mucha más capacidad.

Ejemplo de coste

En el año 2010 con 100 € se podía conseguir una memoria RAM de 4GB con un tiempo de acceso de 5ns o un disco magnético de 1 TB (1.000 GB) con un tiempo de acceso de 5 ms (5.000.000 ns), por lo tanto, al mismo coste podemos tenemos una memoria mil veces más grande, pero un millón de veces más lenta.

1.7. Características físicas

La memoria se puede clasificar según características físicas diferentes; básicamente podemos distinguir dos clasificaciones. La primera distingue entre:

- **Memoria volátil**: memoria que necesita una corriente eléctrica para mantener su estado; estas memorias incluyen registros, memoria caché y memoria principal.

- **Memoria no volátil**: mantiene el estado sin necesidad de corriente eléctrica, incluye memorias de solo lectura, memorias programables, memoria flash, dispositivos de almacenamiento magnético y óptico.

La segunda clasificación distingue entre:

- **Memoria de semiconductores**: es una memoria que utiliza elementos semiconductores, transistores, en su construcción; incluye: registros, memoria caché, memoria principal, memorias de solo lectura, memoria flash.

- **Memoria magnética**: utiliza superficies imantadas para guardar la información; dentro de esta categoría se incluyen básicamente discos y cintas magnéticas.

- **Memoria óptica**: utiliza elementos de almacenamiento que pueden ser leídos y escritos mediante luz láser; se incluyen dispositivos de CD, DVD, Blu-ray.

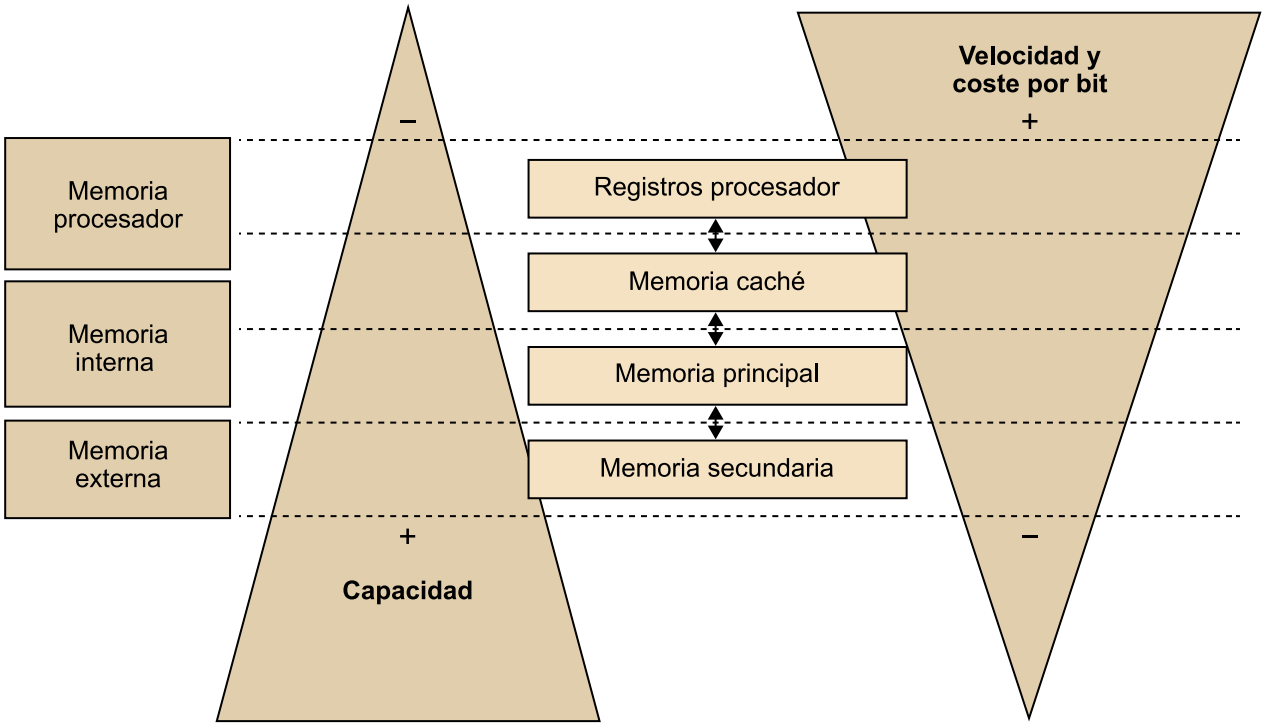
2. Jerarquía de memorias

El objetivo en el diseño del sistema de memoria de un computador es que tenga una gran capacidad y un tiempo de acceso reducido con el precio más bajo posible. Como no hay ninguna tecnología que cumpla simultáneamente estos requisitos, la memoria del computador se estructura en varios niveles con el objetivo de conseguir unas prestaciones mejores, y forma lo que denominamos **jerarquía de memorias**.

En una jerarquía de memorias se utilizan varios tipos de memoria con distintas características de capacidad, velocidad y coste, que dividiremos en niveles diferentes: memoria del procesador, memoria interna y memoria externa.

Cada nivel de la jerarquía se caracteriza también por la distancia a la que se encuentra del procesador. Los niveles más próximos al procesador son los primeros que se utilizan; eso es así porque también son los niveles con una velocidad más elevada.

A continuación se muestra cuál es la variación de la capacidad, velocidad y coste por bit para los niveles típicos de una jerarquía:



El objetivo final de la jerarquía de memorias es conseguir que, cuando el procesador acceda a un dato, este se encuentre en el nivel más rápido de la jerarquía. Obtenemos así una memoria a un coste moderado, con una velocidad próxima a la del nivel más rápido y la capacidad del nivel más alto.

Cada nivel de la jerarquía de la memoria se relaciona solo con los niveles superior e inferior, salvo casos excepcionales. El procesador tiene acceso solamente a los registros y obtiene los datos de memoria mediante la memoria caché.

Por ello, cuando el procesador necesita un dato y este no está disponible en la memoria caché, se tendrá que llevar a ella desde el nivel en el que esté disponible.

Por otra parte, si el procesador modifica un dato en un nivel de la jerarquía de memorias, hay que garantizar que la modificación se efectúe en el resto de los niveles en los que el dato se encuentre almacenado. Si esto no se hiciera así, la siguiente vez que se accediera a este dato, se podría tomar un valor incorrecto. Este problema se denomina *coherencia*.

Como los niveles de memoria más próximos al procesador no son muy grandes, se podría pensar que se pierde mucho tiempo trasladando los datos desde un nivel hasta otro, ya que este movimiento tiene que ser constante. En realidad, eso no es cierto: los datos se reutilizan con mucha frecuencia, por lo que resulta útil que estén en el nivel más próximo al procesador. Más adelante estudiaremos por qué se produce esta reutilización y, por lo tanto, por qué es efectiva la jerarquía de memorias.

A continuación se ofrece una descripción de las características principales de los diferentes niveles de una jerarquía de memoria.

2.1. Registros

El registro es el espacio de memoria que se encuentra dentro del procesador, integrado dentro del mismo chip de este. Se utilizan celdas de memoria de tipo estático, SRAM, para su implementación.

Es el espacio de memoria en el cual el procesador puede acceder más rápidamente a los datos. Este espacio de memoria es accesible al programador de lenguaje de ensamblador y, si se gestiona bien, permite minimizar el número de accesos a la memoria interna, que son bastante más lentos.

2.2. Memoria interna

La memoria interna en un computador moderno está formada típicamente por dos niveles fundamentales: memoria caché y memoria principal. En los computadores actuales es frecuente encontrar la memoria caché también dividida en niveles.

2.2.1. Memoria caché

Las **memorias caché** son memorias de capacidad reducida, pero más rápidas que la memoria principal, que utilizan un método de acceso asociativo. Se pueden encontrar dentro del chip del procesador o cerca de él y están diseñadas para reducir el tiempo de acceso a la memoria. En la memoria caché se almacenan los datos que se prevé que se utilizarán más habitualmente, de manera que sea posible reducir el número de accesos que debe hacer el procesador a la memoria principal (ya que el tiempo de acceso a la memoria principal siempre es superior al tiempo de acceso a la memoria caché).

No es accesible por parte del programador, es gestionada por el hardware y el sistema operativo y se implementa utilizando tecnología SRAM.

Los procesadores modernos utilizan diferentes niveles de memoria caché, lo que se conoce como **memoria caché de primer nivel**, **segundo nivel**, etc. Actualmente es habitual disponer de hasta tres niveles de memoria caché, referidos como L1, L2 y L3. Cada vez es más frecuente que algunos de estos niveles se implementen dentro del chip del procesador y que el nivel más próximo al procesador esté dividido en dos partes: una dedicada a las instrucciones y otra dedicada a los datos.

Memoria caché de los procesadores de Intel y AMD

Los últimos procesadores de Intel y AMD incluyen tres niveles de memoria caché: un primer nivel (L1) dividido en memoria caché de instrucciones y memoria caché de datos, y los otros niveles (L2 y L3), unificados. Los procesadores actuales tienen un diseño multinúcleo (*multicore*); un procesador integra en un solo chip varios núcleos completamente funcionales, cada núcleo dispone de una memoria caché de primer nivel (L1) y de segundo nivel (L2), y la memoria caché de tercer nivel (L3) es compartida por todos los núcleos del procesador. En estos procesadores toda la memoria caché se integra dentro del chip del microprocesador.

2.2.2. Memoria principal

En la memoria principal se almacenan los programas que se deben ejecutar y sus datos, es la memoria visible para el programador mediante su espacio de direcciones.

La memoria principal se implementa utilizando diferentes chips conectados a la placa principal del computador y tiene una capacidad mucho más elevada que la memoria caché (del orden de Gbytes o de Tbytes en supercomputadores).

Utiliza tecnología DRAM (Dynamic RAM), que es más lenta que la SRAM, pero con una capacidad de integración mucho más elevada, hecho que permite obtener más capacidad en menos espacio.

2.3. Memoria externa

La memoria externa corresponde a dispositivos de almacenamiento secundario: discos magnéticos, cintas magnéticas, discos ópticos, dispositivos de memoria flash, etc., y también se pueden considerar sistemas de almacenamiento en red.

Estos dispositivos son gestionados por el sistema de ficheros del sistema operativo mediante el sistema de entrada/salida.

Los dispositivos que forman la memoria externa se conectan al computador con algún tipo de bus (serie o paralelo). Estos dispositivos se pueden encontrar físicamente dentro del computador conectados por buses internos del computador (IDE, SATA, SCSI, etc.) o pueden estar fuera del computador conectados por buses externos (USB, Firewire, eSATA, Infiniband, etc.).

2.4. Memoria virtual

Decimos que un computador utiliza memoria virtual cuando las direcciones de memoria de los programas se refieren a un espacio de memoria superior al espacio de memoria físico, espacio de memoria principal.

La memoria virtual libera al programador de las restricciones de la memoria principal. En estos computadores diferenciamos entre el mapa de direcciones lógicas o virtuales (las direcciones que utilizan los programas) y el mapa de direcciones físicas o reales (las direcciones de la memoria principal). El espacio de memoria virtual utiliza como soporte un dispositivo de almacenamiento externo (habitualmente un disco magnético), mientras que el espacio de memoria físico se corresponde con la memoria principal del computador.

En la actualidad, prácticamente todos los computadores utilizan memoria virtual. La utilización de la memoria virtual implica resolver dos problemas: la traducción de direcciones lógicas a direcciones físicas y la asignación de espacio de memoria físico a los programas que se deben ejecutar. La gestión de la memoria virtual la efectúa el sistema operativo.

Nota

En este módulo nos centraremos en analizar cómo se gestiona el espacio de direcciones físicas del computador; la gestión del espacio de direcciones virtuales corresponde al ámbito de los sistemas operativos y no la analizaremos.

2.5. Funcionamiento de la jerarquía de memorias

Los dos factores básicos que provocan que el esquema de jerarquía de memorias funcione satisfactoriamente en un computador son los siguientes:

- El flujo de datos entre los niveles de la jerarquía de memorias se puede hacer en paralelo con el funcionamiento normal del procesador.
- El principio de **proximidad referencial** de los programas.

El código de los programas se organiza en subrutinas, tiene estructuras iterativas y trabaja con conjuntos de datos agrupados. Esto, unido al hecho de que la ejecución del código es secuencial, lleva a que durante un intervalo de tiempo determinado se utilice solo una pequeña parte de toda la información almacenada: este fenómeno se denomina **proximidad referencial**.

A causa de esta característica, se ha probado empíricamente que aproximadamente el 90% de todas las instrucciones ejecutadas corresponden al 10% del código de un programa.

Distinguimos dos tipos de proximidad referencial:

1) Proximidad temporal. Es cuando, en un intervalo de tiempo determinado, la probabilidad de que un programa acceda de manera repetida a las mismas posiciones de memoria es muy grande.

La proximidad temporal se debe principalmente a las estructuras iterativas; un bucle ejecuta las mismas instrucciones repetidamente, de la misma manera que las llamadas repetitivas a subrutinas.

2) Proximidad espacial. Es cuando, en un intervalo de tiempo determinado, la probabilidad de que un programa acceda a posiciones de memoria próximas es muy grande.

La proximidad espacial se debe principalmente al hecho de que la ejecución de los programas es secuencial –se ejecuta una instrucción detrás de la otra salvo las bifurcaciones– y también a la utilización de estructuras de datos que están almacenados en posiciones de memoria contiguas.

3. Memoria caché

La memoria caché se sitúa entre la memoria principal y el procesador, puede estar formada por uno o varios niveles. En este apartado explicaremos el funcionamiento de la memoria caché considerando un único nivel, pero el funcionamiento es parecido si tiene varios.

La memoria caché tiene un tiempo de acceso inferior al de la memoria principal con el objetivo de reducir el tiempo de acceso medio a los datos, pero también tiene un tamaño mucho más reducido que la memoria principal. Si un dato está en la memoria caché, es posible proporcionarlo al procesador sin acceder a la memoria principal, si no, primero se lleva el dato de la memoria principal a la memoria caché y después se proporciona el dato al procesador.

Si, en la mayoría de los accesos a memoria, el dato está en la memoria caché, el tiempo de acceso medio será próximo al tiempo de acceso a la memoria caché. Eso es factible gracias a la característica de proximidad referencial de los programas.

Para trabajar con memoria caché, la memoria principal se organiza en **bloques** de palabras, de manera que cuando hay que trasladar datos de la memoria principal a la memoria caché se lleva un bloque entero de palabras de memoria, no se trabaja con palabras individuales.

La memoria caché también se organiza en bloques que se denominan **líneas**. Cada línea está formada por un conjunto de palabras (el mismo número de palabras que tenga un bloque de memoria principal), más una etiqueta compuesta por unos cuantos bits. El contenido de la etiqueta permitirá saber qué bloque de la memoria principal se encuentra en cada línea de la memoria caché en un momento dado.

Dirección	Memoria principal	Bloque	Palabra
0		Bloque 0	0
1			1
...			...
$k-1$			$k-1$
k		Bloque 1	0
$k+1$			1
...			...
$2k-1$			$k-1$
...	...	...	...
2^n-k		Bloque $(2^n/k)-1$	0
			1
...			...
2^n-1			$k-1$

Memoria caché				
	Etiqueta del bloque x	Palabras de la línea		
Línea		0	...	$k-1$
0			...	
1			...	
...	...		...	
$m-1$			...	

A la izquierda, memoria principal de 2^n palabras, organizada en $(2^n)/k$ bloques de k palabras. Memoria caché con m líneas, con k palabras por línea.
A la derecha, memoria caché con m líneas, con k palabras por línea, la *etiqueta del bloque x* identifica qué bloque de la memoria principal tenemos almacenado en aquella línea de la memoria caché.

3.1. Aciertos y fallos

Cada vez que el procesador quiere acceder a una palabra de memoria, primero se accede a la memoria caché; si la palabra de memoria se encuentra almacenada en la memoria caché, se proporciona al procesador y diremos que se ha producido un **acierto**. En caso contrario, se lleva el bloque de datos de la memoria principal que contiene la palabra de memoria hacia la memoria caché y, cuando la palabra ya está en la memoria caché, se proporciona al procesador; en este caso diremos que se ha producido un **fallo**.

Cuando hay un fallo, el hardware de la memoria caché debe realizar la secuencia de tareas siguiente:

1) Solicitar a la memoria principal el bloque en el que está el dato que ha producido el fallo.
2) Llevar el bloque de datos solicitado a la memoria caché. Las operaciones realizadas en esta tarea dependerán de las políticas de asignación y algoritmos de reemplazo que veremos más adelante.
3) El procesador obtiene el dato de la memoria caché como si se hubiera producido un acierto.

Un acceso con fallo en la memoria caché puede ser bastante más costoso en tiempo que un acceso con acierto, por lo que es muy importante tener un número reducido de fallos.

3.2. Rendimiento de la memoria caché

A partir del concepto de acierto y fallo se definen los parámetros que utilizaremos para evaluar el rendimiento de una memoria caché: tasa de fallos y tiempo medio de acceso.

La tasa de fallos se define de la manera siguiente:

T_f = Número de fallos / Número de accesos a la memoria

Por otra parte se define la tasa de aciertos así:

T_e = Número de aciertos / Número de accesos a la memoria = $1 - T_f$

Uno de los objetivos del diseño del sistema de memoria es obtener una tasa de fallos tan baja como sea posible. Generalmente se espera que sea inferior al 10%.

Se puede calcular también el tiempo medio de acceso t_m a partir de la tasa de fallos y de la tasa de aciertos, conociendo el tiempo de acceso en caso de aciertos t_e y el tiempo de acceso en caso de fallo t_f, ya que el tiempo de fallo tiene en cuenta el tiempo necesario para llevar todo un bloque de la memoria principal a la memoria caché y el tiempo de acceso al dato.

$$t_m = T_f \text{ x } t_f + T_e \text{ x } t_e = T_f \text{ x } t_f + (1 - T_f) \text{ x } t_e = T_f \text{ x } (t_f - t_e) + t_e$$

Si la tasa de fallos es cero, el tiempo medio de acceso a memoria es igual al tiempo de acceso a la memoria caché.

3.3. Línea de memoria caché

Hemos visto que la memoria caché se organiza en líneas; una línea está formada básicamente por un conjunto de palabras más una etiqueta que identifica qué bloque de la memoria principal ocupa aquella línea de la memoria caché.

La línea de memoria caché es la unidad de transferencia entre la memoria caché y la memoria principal. El tamaño de la línea es uno de los parámetros fundamentales del diseño de la memoria caché. Hay que decidir cuántas palabras se almacenarán en una línea de memoria caché, es decir, cuál es el tamaño de una línea.

Hemos visto que los datos se trasladan de la memoria principal a la memoria caché cuando hay un fallo. Si se produce un fallo, se lleva a la memoria caché el dato que lo ha provocado y el resto de los datos del bloque de memoria donde se encuentra este dato. De esta manera, se espera que los accesos siguientes sean aciertos en la memoria caché.

El tamaño de la línea es de unos cuantos bytes de información (un tamaño habitual está entre los 32 bytes y 128 bytes). Aumentar el tamaño de la línea permite aprovechar la localidad espacial, pero hasta cierto punto. Cuando se produce un fallo, el tiempo necesario para trasladar una línea más grande aumenta; además, disminuye el número de líneas disponibles de la memoria caché (el tamaño de la memoria caché es fijo) y tendremos más competencia para conseguir un bloque, lo que hará que se saquen de la caché líneas que todavía no se han utilizado en su totalidad y se reducirá el efecto de la localidad espacial, y todo ello puede representar un aumento en la tasa de fallos.

3.4. Políticas de asignación

El número de líneas disponibles en la memoria caché es siempre mucho más pequeño que el número de bloques de memoria principal. En consecuencia, la memoria caché, además de la información almacenada, debe mantener alguna información que relacione cada posición de la memoria caché con su dirección en la memoria principal.

Para acceder a un dato se especifica la dirección en la memoria principal; a partir de esta dirección hay que verificar si el dato está en la memoria caché. Esta verificación la haremos a partir del campo *etiqueta* de la línea de la memoria caché que indica qué bloque de memoria principal se encuentra en cada una de las líneas de la memoria caché.

La política de asignación determina dónde podemos colocar un bloque de la memoria principal dentro de la memoria caché y condiciona cómo encontrar un dato dentro de la memoria caché.

Se definen tres políticas de asignación diferentes para almacenar datos dentro de una memoria caché:

1) Política de asignación directa: un bloque de la memoria principal solo puede estar en una única línea de la memoria caché. La memoria caché de asignación directa es la que tiene la tasa de fallos más alta, pero se utiliza mucho porque es la más barata y fácil de gestionar.

2) Política de asignación completamente asociativa: un bloque de la memoria principal puede estar en cualquier línea de la memoria caché. La memoria caché completamente asociativa es la que tiene la tasa de fallos más baja. No obstante, no se suele utilizar porque es la más cara y compleja de gestionar.

3) Política de asignación asociativa por conjuntos: un bloque de la memoria principal puede estar en un subconjunto de las líneas de la memoria caché, pero dentro del subconjunto puede encontrarse en cualquier posición.

La memoria caché asociativa por conjuntos es una combinación de la memoria caché de asignación completamente asociativa y la memoria caché de asignación directa. El número de elementos de cada subconjunto no suele ser muy grande, un número habitual de elementos es entre 4 y 64. Si el número de elementos del subconjunto es *n*, la memoria caché se denomina *n*-asociativa.

Líneas de la memoria caché donde podemos asignar el bloque x según las diferentes políticas de asignación

Dirección	Memoria principal	Bloque
0		Bloque 0
1		
...		
$k-1$		
k		Bloque 1
$k+1$		
...		
$2 \cdot k-1$		
...	...	...
$x \cdot k+0$		Bloque x
$x \cdot k+(k-1)$		
...	...	...
2^n-k		Bloque $(2^n / k)-1$
...		
2^n-1		

Memoria caché de acceso directo		
Línea	**Etiqueta**	**Palabras del bloque**
0		
...		...
Línea asignada al bloque x		
...		...
$m-1$		

Memoria caché de asignación completamente asociativa		
Línea		**Contenido de la caché**
0		
...	...	...
Bloque x asignado a cualquier línea		
...	...	...
$m-1$		

Memoria caché de asignación asociativa por conjuntos		
Línea		**Contenido de la caché**
0		
...	...	...
...	...	...
Conjunto de líneas asignado al bloque x		
		...
...	...	...
...	...	...
$m-1$		

3.4.1. Memoria caché de asignación directa

Para utilizar este tipo de memoria caché, se asigna cada bloque de la memoria principal a una única línea de la memoria caché.

Para relacionar una línea de la memoria caché con un bloque de la memoria principal a partir de la dirección especificada para acceder a una palabra de la memoria principal, hemos de determinar a qué bloque pertenece la dirección.

Se divide la dirección en dos partes: número de bloque, que corresponde a la parte más significativa de la dirección, y número de palabra, que corresponde a la parte menos significativa.

Si tenemos una memoria principal de 2^n palabras y una memoria caché de 2^m líneas de 2^k palabras por línea, la memoria principal se dividirá en bloques de 2^k palabras. Una dirección de memoria estará formada por n bits, utilizará los k bits menos significativos para el número de palabra y los $n - k$ bits restantes para el número de bloque.

Dirección de memoria	
Número de bloque	Número de palabra
$n - 1$ ... k	$k - 1$ ... 0
← ($n - k$ bits) →	← (k bits) →

Cálculo del número de bloque

A partir de una dirección de memoria a se puede calcular el número de bloque b realizando la operación siguiente: $b = a$ div 2^k, donde *div* es la división entera.

El número de palabra p se puede calcular mediante la operación siguiente: $p = a$ mod 2^k, donde *mod* es el residuo de la división entera.

Para determinar a qué línea de la memoria caché podemos asignar cada bloque, hay que dividir el número de bloque en dos partes: una etiqueta, que corresponde a la parte más significativa del número de bloque, y un número de línea, que corresponde a la parte menos significativa.

Si tenemos un número de bloque que utiliza $n - k$ bits, de estos $n - k$ bits utilizaremos m bits para especificar el número de línea y el resto de los bits ($n - k - m$) para especificar la etiqueta.

Dirección de memoria		
Número de bloque		Número de palabra
Etiqueta	Número de línea	
$n - 1$ ... $k + m$	$k + m - 1$ ... k	$k - 1$ ... 0
← ($n - k - m$ bits) →	← (m bits) →	← (k bits) →

Cálculo del número de etiqueta

A partir del número de bloque b se puede calcular la etiqueta e haciendo la operación siguiente: $e = b$ div 2^m, donde *div* es la división entera.

El número de línea l se puede calcular realizando la operación siguiente: $l = b$ mod 2^m, donde *mod* es el residuo de la división entera.

Tenemos 2^{n-k} bloques en la memoria principal y 2^m líneas en la memoria caché ($2^{n-k} > 2^m$), por lo tanto a cada línea de la memoria caché podemos asignar 2^{n-k-m} (= $2^{n-k} / 2^m$) bloques diferentes. Solo uno de estos 2^{n-k-m} puede estar en la memoria caché en cada momento.

El número de línea indicará en cuál de las 2^m líneas de la memoria caché se puede encontrar el bloque de datos al que queremos acceder de la memoria principal. La etiqueta nos permitirá saber si el bloque al que queremos acceder de la memoria principal es el bloque que en este momento está almacenado en aquella línea de la memoria caché.

Cuando se lleva un bloque de la memoria principal a la línea correspondiente de la memoria caché, el número de la etiqueta del bloque se almacena en el campo etiqueta de la línea, así podremos saber cuál de los 2^{n-k-m} bloques está almacenado en esta línea de la caché. El campo etiqueta es el que nos permite identificar de manera única cada uno de los bloques que podemos asignar a una misma línea de la memoria caché.

De manera general, se puede decir que si tenemos una memoria caché de 2^m líneas, los bloques de memoria principal que se pueden encontrar en cada una de las líneas de la memoria caché son los que se muestran en la tabla siguiente.

Número de línea	Bloques asignados
0	0, 2^m, 2x (2^m), ...
1	1, $2^m + 1$, 2x (2^m) + 1, ...
2	2, $2^m + 2$, 2x (2^m) + 2, ...
...	...
$2^m - 1$	$2^m - 1$, $2^m + (2^m - 1)$, 2x $2^m + (2^m - 1)$, ...

Para determinar si un acceso a una dirección de memoria produce un acierto en la memoria caché, hay que hacer lo siguiente: a partir de la dirección de memoria se determina cuál es su número de línea (bits $k + m - 1$ .. k), el cual se utiliza como índice para acceder a la caché y obtener la etiqueta que identifica el bloque almacenado en esta línea y que se compara con el campo etiqueta de la dirección (bits $n - 1$ .. $k + m$); si coinciden, se trata de un acierto, entonces se utiliza el número de palabra (bits $k - 1$ .. 0) para obtener la palabra solicitada y servirla al procesador.

Si la etiqueta de la dirección y la etiqueta de la línea no coinciden, se trata de un fallo y habrá que trasladar todo el bloque de memoria principal a la memoria caché, reemplazando el bloque que tenemos actualmente almacenado.

Ejemplo

Si tenemos una memoria principal de 2^{16} (64 K) palabras y una memoria caché de 2^{10} (1.024) palabras organizada en 2^4 (16) líneas de 2^6 (64) palabras por línea, la memoria principal se dividirá en bloques de 2^6 (64) palabras. Una dirección de memoria tendrá 16 bits, los 6 bits menos significativos para el número de palabra y los 16 – 6 = 10 bits restantes para el número de bloque; en total tendremos 2^{10} (1.024) bloques de 2^6 (64) palabras. Las direcciones se dividirán de la manera siguiente:

Dirección de memoria	
Número de bloque	Número de palabra
15 6	5 0

El número de bloque de 10 bits se divide en etiqueta y número de línea: 4 bits para la línea, ya que hay 2^4 líneas y 10 – 4 = 6 bits para la etiqueta. Así las direcciones de memoria principal se dividirán de la manera siguiente:

Dirección de memoria		
Número de bloque		Número de palabra
Etiqueta	Número de línea	
15 10	9 6	5 0

La asignación de bloques de la memoria principal a la memoria caché sería de la manera siguiente:

Número de línea	Bloques asignados
0	0, 16, 32, 48, 64, ..., 1008
1	1, 17, 33, 49, 65, ..., 1009
2	2, 18, 34, 50, 66, ..., 1010
...	...
15	15, 31, 63, 79, ..., 1023

A cada línea de la memoria caché le podemos asignar 2^6 = 64 bloques diferentes.

El mapa de direcciones de memoria principal quedaría de la manera siguiente:

	Dirección de memoria		
	Número de bloque		Número de palabra
	Etiqueta	Número de línea	
Bloque 0	0	0	0
			...
			63 ($2^6 - 1$)
Bloque 1	0	1	0
			...
			63 ($2^6 - 1$)
...			
Bloque 15	0	15	0
			...
			63 ($2^6 - 1$)
Bloque 16	1	0	0
			...
			63 ($2^6 - 1$)
...			
...			
...			
Bloque 1007	62	15	0
			...
			63 ($2^6 - 1$)
Bloque 1008	63	0	0
			...
			63 ($2^6 - 1$)
...			
Bloque 1023	63	15	0
			...
			63 ($2^6 - 1$)

Se puede observar que todos los bloques que pueden estar en una misma línea de la caché tienen un valor de etiqueta diferente; se podrá utilizar el valor de la etiqueta para saber qué bloque en concreto se encuentra en cada línea de la memoria caché. La tabla anterior muestra que todos los bloques sombreados están asignados a la línea 0 de la memoria caché y podremos saber cuál se encuentra en la memoria caché gracias al número de la etiqueta.

A continuación, se muestra un contenido posible de la memoria caché:

	Número de bloque	
	Etiqueta	Línea
Bloque 16	1	0
Bloque 1	0	1
	...	...
Bloque 1023	63	15

Memoria caché				
Línea	Etiqueta	Palabras de la línea		
		0	...	$2^k - 1$
0	1	M(64)	...	M(127)
1	0	M(1024)	...	M(1087)
...	...	...	...	...
15	63	M(65472)	...	M(65535)

En la línea 0 tenemos el bloque 16 (etiqueta: 1), en la línea 1 tenemos el bloque 1 (etiqueta: 0) y en la línea 15 tenemos el bloque 1023 (etiqueta: 63). M(x) indica que en esta palabra de la línea de la caché se ha almacenado la palabra de memoria con la dirección x.

A continuación, se muestra la descomposición de una de las direcciones del bloque 16 que se encuentra en la memoria caché.

	Dirección de memoria		
	64 = 000001 0000 000000		
	Número de bloque		Número de palabra
	Etiqueta	Número de línea	
Bloque 16	1 = 000001	0000	000000

3.4.2. Memoria caché completamente asociativa

A diferencia de la memoria caché directa, un bloque de memoria principal se puede encontrar en cualquier línea de la memoria caché.

Para relacionar una línea de la memoria caché con un bloque de la memoria principal a partir de la dirección especificada para acceder a una palabra de la memoria principal, hemos de determinar a qué bloque pertenece la dirección. Se divide la dirección en dos partes: número de bloque que corresponde a la parte más significativa de la dirección y número de palabra que corresponde a la parte menos significativa.

Si tenemos una memoria principal de 2^n palabras y una memoria caché de 2^m líneas de 2^k palabras por línea, la memoria principal se dividirá en bloques de 2^k palabras. Una dirección de memoria estará formada por n bits y utilizará los k bits menos significativos para el número de palabra y los $n - k$ bits restantes para el número de bloque.

Dirección de memoria	
Número de bloque	Número de palabra
$n - 1$ k	$k - 1$ 0
← ($n - k$ bits) →	← (k bits) →

Cálculo del número de bloque

A partir de una dirección de memoria a se puede calcular el número de bloque b haciendo la operación siguiente: $b = a \text{ div} 2^k$, donde *div* es la división entera.

El número de palabra p se puede calcular haciendo la operación siguiente: $p = a \bmod 2^k$, donde *mod* es el residuo de la división entera.

Cabe tener presente que a cada línea de la memoria caché le podemos asignar cualquier bloque de la memoria principal y debemos poder saber cuál se encuentra en cada momento en la memoria caché.

Cuando se traslada un bloque de memoria principal a la memoria caché, hay que decidir qué línea reemplazamos con el nuevo bloque de datos; para tomar esta decisión, se pueden utilizar diferentes algoritmos de reemplazo que

explicaremos más adelante. El número de bloque de la dirección de memoria se almacena en el campo etiqueta de la línea; así podremos saber qué bloque está almacenado en cada una de las líneas de la caché.

Para determinar si un acceso a una dirección de memoria produce un acierto en la memoria caché, hay que hacer lo siguiente:

A partir de la dirección de memoria se determina cuál es su número de bloque (bits $n - 1$ .. k) y se compara simultáneamente el número de bloque de esta dirección con el campo etiqueta de todas las líneas de la memoria caché; si se produce una coincidencia significa que hay un acierto, entonces se utiliza el número de palabra (bits $k - 1$ .. 0) para obtener la palabra solicitada y servirla al procesador.

Si el número de bloque de la dirección no coincide con ninguna etiqueta de la memoria caché, se trata de un fallo y habrá que llevar todo el bloque de memoria principal a la memoria caché reemplazando uno de los bloques que tenemos actualmente almacenados.

Ejemplo

Tenemos una memoria principal de 2^{16} (64 K) palabras y una memoria caché de 2^{10} (1.024) palabras organizada en 2^6 (64) líneas de 2^4 (16) palabras por línea.

La memoria principal se dividirá en bloques de 2^4 (16) palabras. Una dirección de memoria tendrá 16 bits, los 4 bits menos significativos para el número de palabra y los 16 – 4 = 12 bits restantes para el número de bloque; en total tendremos 2^{12} (4.096) bloques de 2^4 (16) palabras. Las direcciones se dividirán de la manera siguiente:

Dirección de memoria	
Número de bloque	Número de palabra
15 4	3 0

A continuación se muestra un posible contenido de la memoria caché:

	Número de bloque
	Etiqueta
Bloque 4095	4095
Bloque 1024	1024
	...
Bloque 1	1

Memoria caché				
Línea	Etiqueta	Palabras de la línea		
		0	...	$2^k - 1$
0	4095	M(65520)	...	M(65535)
1	1024	M(16384)	...	M(16400)
...	...	...	...	...
63	1	M(16)	...	M(31)

M(x) indica que en esta palabra de la línea de la caché está almacenada la palabra de memoria con la dirección x.

A continuación se muestra la descomposición de una de las direcciones del bloque 16 que se encuentra en la memoria caché.

	Dirección de memoria	
	16384 = 010000000000 0000	
	Número de bloque	Número de palabra
Bloque 1024	1024 = 010000000000	000000

3.4.3. Memoria caché asociativa por conjuntos

Un bloque de la memoria principal puede encontrarse en un único conjunto de líneas de la memoria caché, pero dentro del conjunto puede encontrarse en cualquier línea.

Para relacionar una línea de la memoria caché con un bloque de la memoria principal a partir de la dirección especificada para acceder a una palabra de la memoria principal, hemos de determinar a qué bloque pertenece la dirección. Se divide la dirección en dos partes: número de bloque que corresponde a la parte más significativa de la dirección y número de palabra que corresponde a la parte menos significativa.

Si tenemos una memoria principal de 2^n palabras y una memoria caché de 2^m líneas de 2^k palabras por línea, la memoria principal se dividirá en bloques de 2^k palabras. Una dirección de memoria estará formada por n bits, utilizará los k bits menos significativos para el número de palabra y los $n - k$ bits restantes para el número de bloque.

Dirección de memoria	
Número de bloque	Número de palabra
$n - 1$ k	$k - 1$ 0
← ($n - k$ bits) →	← (k bits) →

Cálculo del número de bloque

A partir de una dirección de memoria a se puede calcular el número de bloque b haciendo la operación siguiente: $b = a$ div2^k, donde *div* es la división entera.

El número de palabra p se puede calcular mediante la operación siguiente: $p = a \ mod \ 2^k$, donde *mod* es el residuo de la división entera.

En una memoria caché asociativa por conjuntos hay que organizar la memoria caché en conjuntos; se tiene que dividir las 2^m líneas de la memoria caché en 2^c conjuntos de $\omega = 2^{m-c} = (2^m/2^c)$ líneas cada uno, y de esta manera diremos que es una memoria caché ω -asociativa.

Si tenemos tantos conjuntos como líneas ($2^c = 2^m$) y cada conjunto tiene una sola línea ($\omega = 1$), estamos ante el mismo caso que una memoria caché de asignación directa; si tenemos un solo conjunto ($2^c = 1$) de 2^m líneas ($\omega = 2^m$), se trata de una memoria completamente asociativa.

Para determinar a qué conjunto de la memoria caché podemos asignar cada bloque de la memoria principal, hay que dividir el número de bloque en dos partes: una etiqueta que corresponde a la parte más significativa del número de bloque y un número de conjunto correspondiente a la parte menos significativa.

Si tenemos un número de bloque que utiliza $n - k$ bits, de estos $n - k$ bits utilizaremos c bits para especificar el número de conjunto y el resto de los bits ($n - k - c$), para especificar la etiqueta.

Dirección de memoria		
Número de bloque		Número de palabra
Etiqueta	Número de conjunto	
$n - 1$ $k + c$	$k + c - 1$ k	$k - 1$ 0
← ($n - k - c$ bits) →	← (c bits) →	← (k bits) →

Cálculo del número de etiqueta

A partir del número de bloque b se puede calcular la etiqueta e haciendo la operación siguiente: $e = b$ div 2^c, donde *div* es la división entera.

El número de línea l se puede calcular haciendo la operación siguiente: $l = b$ mod 2^c, donde *mod* es el residuo de la división entera.

Tenemos 2^{n-k} bloques de la memoria principal y 2^c conjuntos de la memoria caché ($2^{n-k} > 2^c$), por lo tanto a cada conjunto de la memoria caché podemos asignar 2^{n-k-c} ($= 2^{n-k}/2^c$) bloques diferentes. Como cada conjunto dispone de ω líneas, solo ω bloques de los 2^{n-k-c} pueden encontrarse en un conjunto de la memoria caché en cada momento.

El número de conjunto de la dirección indicará en cuál de los 2^c conjuntos de la memoria caché se puede encontrar el bloque al que queremos acceder de la memoria principal. La etiqueta nos permitirá saber, comparando simultáneamente todas las etiquetas de las líneas que forman el conjunto, si el bloque al que queremos acceder de la memoria principal es uno de los bloques que en este momento están almacenados en una línea de aquel conjunto de la memoria caché.

Cuando se traslada un bloque de memoria principal a la memoria caché, el campo etiqueta del bloque se almacena en el campo etiqueta de la línea seleccionada dentro del conjunto que le corresponda (según el número de conjunto que especifica la dirección), de esta manera podremos saber qué bloque está almacenado en cada una de las líneas de la caché.

De manera general se puede decir que, si tenemos una memoria caché de 2^m líneas, los bloques de memoria principal que se pueden encontrar en cada una de las líneas de la memoria caché son los siguientes:

Líneas	Número de conjunto	Bloques asignados
0, ..., ω – 1	0	0, 2^c, 2x (2^c), 3x (2^c) ...
ω , ..., 2ω – 1	1	1, 2^c + 1, 2x (2^c) + 1, 3x (2^c) + 1 ...
	...	...
(2^c – 1) ω , ..., $2^{c\,\omega}$ – 1	2^c – 1	2^c – 1, 2^c + (2^c – 1), 2x 2^c + (2^c – 1), ..., 3x 2^c + (2^c – 1)

Para determinar si un acceso a una dirección de memoria produce un acierto en la memoria caché, hay que hacer lo siguiente: a partir de la dirección de memoria se determina cuál es su número de conjunto (bits $k + c - 1$ .. k). Este número de conjunto se utiliza como índice para acceder a las etiquetas de las líneas que identifican los bloques que están almacenados en este conjunto y que se comparan simultáneamente con el campo etiqueta de la dirección (bits $n - 1$ .. $k + c$). Si hay una coincidencia significa que se ha producido un acierto y entonces se utiliza el número de palabra (bits $k - 1$ .. 0) para obtener la palabra solicitada y servirla al procesador.

Si la etiqueta de la dirección no coincide con ninguna etiqueta del conjunto, se trata de un fallo y habrá que llevar todo el bloque de memoria principal a una de las líneas de este conjunto de la memoria caché, reemplazando uno de los bloques que tenemos actualmente almacenados. Como un bloque de memoria principal puede ir a cualquier línea del conjunto, hay que decidir qué línea reemplazaremos con el nuevo bloque de datos; para tomar esta decisión se pueden utilizar diferentes algoritmos de reemplazo que explicaremos más adelante.

Ejemplo

Si tenemos una memoria principal de 2^{16} (64 K) palabras y una memoria caché de 2^{10} (1.024) palabras organizada en 2^6 (64) líneas de 2^4 (16) palabras por línea, dividimos las líneas de la caché en 2^4 (16) conjuntos de 2^2 (4) líneas; por lo tanto, tendremos ω = 4 y diremos que es una memoria caché 4-asociativa.

La memoria principal se dividirá en bloques de 2^4 (16) palabras. Una dirección de memoria tendrá 16 bits, los 4 bits menos significativos para el número de palabra y los 16

– 4 = 12 bits restantes para el número de bloque, en total tendremos 2^{12} (4.096) bloques de 2^4 (16) palabras. Las direcciones se dividirán de la manera siguiente:

Dirección de memoria	
Número de bloque	Número de palabra
15 4	3 0

El número de bloque de 12 bits se divide en etiqueta y número de conjunto: 4 bits para el número de conjunto, ya que hay 2^4 conjuntos, y 12 – 4 = 8 bits para la etiqueta. Así las direcciones de memoria principal se dividirán de la manera siguiente:

Dirección de memoria		
Número de bloque		Número de palabra
Etiqueta	Número de conjunto	
15 8	7 4	3 0

La asignación de bloques de la memoria principal a la memoria caché sería de la manera siguiente:

Líneas	Número de conjunto	Bloques asignados
0, 1, 2, 3	0	0, 16, 32, 48, 64, ..., 4080
4, 5, 6, 7	1	1, 17, 33, 49, 65, ..., 4081
...	...	...
60, 61,62, 63	15	15, 31, 47, 63, 79, ..., 4095

El mapa de direcciones de la memoria principal quedaría de la manera siguiente:

	Dirección de memoria		
	Número de bloque		Número de palabra
	Etiqueta	Número de conjunto	
Bloque 0	0	0	0
			...
			15 ($2^4 - 1$)
Bloque 1	0	1	0
			...
			15 ($2^4 - 1$)
...			
Bloque 15	0	15	0
			...
			15 ($2^4 - 1$)
Bloque 16	1	0	0
			...
			15 ($2^4 - 1$)
...			
...			
...			
Bloque 4079	254	15	0
			...
			15 ($2^4 - 1$)
Bloque 4080	255	0	0
			...
			15 ($2^4 - 1$)
...			
Bloque 4095	255	15	0
			...
			15 ($2^4 - 1$)

Se puede observar que todos los bloques que se pueden encontrar en un mismo conjunto de la caché tienen un valor de etiqueta diferente; se puede utilizar el valor de la etiqueta para saber qué bloque en concreto se encuentra en cada línea de la memoria caché. La tabla anterior muestra que los bloques sombreados están asignados todos al conjunto 0 de la memoria caché y podremos saber cuál se encuentra en la memoria caché gracias al número de la etiqueta.

A continuación se muestra un posible contenido de la memoria caché:

	Número de bloque		Conjunto	Memoria caché				
				Línea	Etiqueta	Palabras de la línea		
	Etiqueta	Conjunto				0	...	$2^k - 1$
Bloque 16	1	0	0	0	1	M(256)	...	M(271)
Bloque 0	0	0		1	0	M(0)	...	M(15)
Bloque 4080	255	0		2	255	M(65280)	...	M(65295)
Bloque 1008	63	0		3	63	M(16128)	...	M(16143)
Bloque 1	0	1	1	4	0	M(16)	...	M(31)
	...	...		...	...	...	...	...
	...	...	...	...	...	...	...	...
	...	...	15	...	...	...	...	...
Bloque 4095	255	15		63	255	M(65520)	...	M(65535)

M(x) indica que en esta palabra de la línea de la caché se ha almacenado la palabra de memoria con la dirección x.

A continuación se muestra la descomposición de una de las direcciones del bloque 1008 que se encuentra en la memoria caché.

	Dirección de memoria		
	16128 = 00111111 0000 0000		
	Número de bloque		Número de palabra
	Etiqueta	Número de conjunto	
Bloque 1008	63 = 00111111	0000	0000

3.5. Algoritmos de reemplazo

Cuando se produce un fallo y se tiene que llevar a la memoria caché un bloque de memoria principal determinado, si este bloque de memoria se puede almacenar en más de una línea de la memoria caché, hay que decidir en qué línea de todas las posibles se pone, y sobreescribir los datos que se encuentran en aquella línea. El algoritmo de reemplazo se encarga de esta tarea.

En una memoria caché directa no es necesario ningún algoritmo de reemplazo, ya que un bloque solo puede ocupar una única línea dentro de la memoria caché. En una memoria caché completamente asociativa, solo se aplica el algoritmo de reemplazo para seleccionar una de las líneas de la memoria caché. En una memoria caché asociativa por conjuntos, solo se aplica el algoritmo de reemplazo para seleccionar una línea dentro de un conjunto concreto.

Para que estos algoritmos de reemplazo no penalicen el tiempo medio de acceso a memoria, se deben implementar en hardware y, por lo tanto, no deberían ser muy complejos.

A continuación se describen de manera general los algoritmos de reemplazo más comunes, pero se pueden encontrar otros algoritmos o variantes de estos.

1) FIFO (*first in first out*). Para elegir la línea se utiliza una cola, de manera que la línea que hace más tiempo que está almacenada en la memoria caché será la reemplazada. Este algoritmo puede reducir el rendimiento de la memoria caché porque la línea que se encuentra almacenada en la memoria caché desde hace más tiempo no tiene que ser necesariamente la que se utilice menos.

Se puede implementar fácilmente utilizando técnicas de *buffers* circulares (o *round-robin*): cada vez que se debe sustituir una línea se utiliza la línea del *buffer* siguiente, y cuando se llega a la última, se vuelve a empezar desde el principio.

2) LFU (*least frequently used*). En este algoritmo se elige la línea que hemos utilizado menos veces.

Se puede implementar añadiendo un contador del número de accesos a cada línea de la memoria caché.

3) LRU (*least recently used)*. Este algoritmo elige la línea que hace más tiempo que no se utiliza. Es el algoritmo más eficiente, pero el más difícil de implementar, especialmente si hay que elegir entre muchas líneas. Se utiliza habitualmente en memorias caché asociativas por conjuntos, con conjuntos pequeños de 2 o 4 líneas.

Para memorias cachés 2-asociativas, se puede implementar añadiendo un bit en cada línea; cuando se hace referencia a una de las dos líneas, este bit se pone a 1 y el otro se pone a 0 para indicar cuál de las dos líneas ha sido la última que se ha utilizado.

4) Aleatorio. Los algoritmos anteriores se basan en factores relacionados con la utilización de las líneas de la caché; en cambio, este algoritmo elige la línea que se debe reemplazar al azar. Este algoritmo es muy simple y se ha demostrado que tiene un rendimiento solo ligeramente inferior a los algoritmos que tienen en cuenta factores de utilización de las líneas.

3.6. Comparativa entre diferentes sistemas de memoria caché

Utilizaremos un ejemplo para ver cómo los accesos a memoria de un programa pueden producir aciertos y fallos en la memoria caché y cómo modifican el contenido de la memoria caché.

Supongamos una memoria principal de 2^{10} (1.024) palabras, en la que cada dirección de memoria corresponde a una palabra, y una memoria caché de 2^4 (4) líneas de 2^2 (4) palabras; por lo tanto, la memoria principal también estará organizada en bloques de tamaño de 4 palabras.

Para determinar el número de bloque que corresponde a una dirección de memoria, dividimos la dirección *d* entre el número de palabras de un bloque:

$$b = d \text{ div } 2^k = d \text{ div } 4$$

Este número de bloque se aplica a todas las organizaciones de la memoria caché.

3.6.1. Memoria caché de asignación directa

En una memoria caché directa un bloque de memoria solo se puede encontrar en una única línea de la memoria caché. A un bloque de memoria *b* le corresponderá la etiqueta *e* y lo asignaremos a la línea *l* de la memoria caché, para determinar la etiqueta y la línea, dividimos el número de bloque *b* entre el número de líneas de la memoria caché:

$$e = b \text{ div } 2^m = b \text{ div } 4$$
$$l = b \text{ mod } 2^m = b \text{ mod } 4$$

Por lo tanto, los bloques de memoria principal que se pueden asignar a cada línea de la memoria caché son los siguientes:

l: número de línea	Bloques	Bloque: etiqueta (6 bits) línea (2 bits)
0 (00)	0, 4, 8, 12, ..., 252	0(000000) 0(00), 1(000001) 0(00), 2(000010) 0(00), ..., 63(111111) 0(00)
1 (01)	1, 5, 9, 13, ..., 253	0(000000) 1(01), 1(000001) 1(01), 2(000010) 1(01), ..., 63(111111) 1(01)
2 (10)	2, 6, 10, 14, ..., 254	0(000000) 2(10), 1(000001) 2(10), 2(000010) 2(10), ..., 63(111111) 2(10)
3 (11)	3, 7, 11, 15, ..., 255	0(000000) 3(11), 1(000001) 3(11), 2(000010) 3(11), ..., 63(111111) 3(11)

Mostramos a qué líneas de la memoria caché se asignan los primeros 16 bloques de memoria principal con las direcciones de memoria que contiene el bloque:

l: número de línea	b:e (a_0 ,a_1 ,a_2 ,a_3): bloque asignado : etiqueta (direcciones del bloque)			
0	0:0 (0,1,2,3)	4:1 (16,17,18,19)	8:2 (32,33,34,35)	12:3 (48,49,50,51)
1	1:0 (4,5,6,7)	5:1 (20,21,22,23)	9:2 (36,37,38,39)	13:3 (52,53,54,55)
2	2:0 (8,9,10,11)	6:1 (24,25,26,27)	10:2 (40,41,42,43)	14:3 (56,57,58,59)

l: número de línea	b:e (a_0 ,a_1 ,a_2 ,a_3): bloque asignado : etiqueta (direcciones del bloque)			
3	3:0 (12,13,14,15)	7:1 (28,29,30,31)	11:2 (44,45,46,47)	15:3 (60,61,62,63)

Hay que remarcar que especificamos el número de bloque y el número de etiqueta, pero que el valor que tendremos realmente almacenado en la caché es tan solo la etiqueta asociada al bloque.

Supongamos que durante la ejecución de un programa se accede a las direcciones de memoria siguientes: 1, 2, 4, 10, 15, 1, 26, 27, 28, 29, 36, 37, 38, 40, 10, 11, 12, 13, 9, 30, 8, 12, 40, 17, 40.

La tabla siguiente muestra la evolución del contenido de la memoria caché e indica el número de bloque, la etiqueta del bloque y las direcciones de memoria del bloque que hay en cada una de las 4 líneas de la memoria caché. Inicialmente, la memoria caché está vacía. Cuando se produce un acierto, se indica con una *E* la línea donde se ha producido el acierto. Cada vez que hay un fallo, se indica con una letra *F* qué línea de la memoria caché se reemplazará y se actualiza el contenido llevando el nuevo bloque de memoria principal a esta línea de la memoria caché.

	Estado inicial	1	Fallo	2	4	Fallo	10	Fallo	15	Fallo
Línea 0		F	0:0 (0, 1, 2, 3)	E		0:0 (0, 1, 2, 3)		0:0 (0, 1, 2, 3)		0:0 (0, 1, 2, 3)
Línea 1					F	1:0 (4, 5, 6, 7)		1:0 (4, 5, 6, 7)		1:0 (4, 5, 6, 7)
Línea 2							F	2:0 (8, 9, 10, 11)		2:0 (8, 9, 10, 11)
Línea 3									F	3:0 (12, 13, 14, 15)

	1	26	Fallo	27	28	Fallo	29	36	Fallo
Línea 0	E		0:0 (0, 1, 2, 3)			0:0 (0, 1, 2, 3)			0:0 (0, 1, 2, 3)
Línea 1			1:0 (4, 5, 6, 7)			1:0 (4, 5, 6, 7)		F	9:2 (36, 37, 38, 39)
Línea 2		F	6:1 (24, 25, 26, 27)	E		6:1 (24, 25, 26, 27)			6:1 (24, 25, 26, 27)
Línea 3			3:0 (12, 13, 14, 15)		F	7:1 (28, 29, 30, 31)	E		7:1 (28, 29, 30, 31)

	37	38	40	Fallo	10	Fallo	11	12	Fallo	13	9
Línea 0				0:0 (0, 1, 2, 3)		0:0 (0, 1, 2, 3)			0:0 (0, 1, 2, 3)		
Línea 1	E	E		9:2 (36, 37, 38, 39)		9:2 (36, 37, 38, 39)			9:2 (36, 37, 38, 39)		
Línea 2			F	10:2 (40, 41, 42, 43)	F	2:0 (8, 9, 10, 11)	E		2:0 (8, 9, 10, 11)		E
Línea 3				7:1 (28, 29, 30, 31)		7:1 (28, 29, 30, 31)		F	3:0 (12, 13, 14, 15)	E	

	30	Fallo	8	12	Fallo	40	Fallo	17	Fallo	40

Línea 0		0:0 (0, 1, 2, 3)			0:0 (0, 1, 2, 3)		0:0 (0, 1, 2, 3)	F	4:1 (16, 17, 18, 19)	
Línea 1		9:2 (36, 37, 38, 39)			9:2 (36, 37, 38, 39)		9:2 (36, 37, 38, 39)		9:2 (36, 37, 38, 39)	
Línea 2		2:0 (8, 9, 10, 11)	E		2:0 (8, 9, 10, 11)	F	10:2 (40, 41, 42, 43)		10:2 (40, 41, 42, 43)	E
Línea 3	F	7:1 (28,29,30,31)		F	3:0 (12, 13, 14, 15)		3:0 (12, 13, 14, 15)		3:0 (12, 13, 14, 15)	

3.6.2. Memoria caché completamente asociativa

Utilizamos ahora una memoria caché completamente asociativa. En una memoria caché completamente asociativa, un bloque de memoria principal se puede encontrar en cualquier línea de la caché. Las direcciones de memoria se dividen en número de bloque y número de palabra.

Mostramos los primeros 16 bloques de la memoria principal con las direcciones de memoria que contiene el bloque:

b (a_0,a_1 ,a_2 ,a_3): bloque (direcciones del bloque)			
0 (0,1,2,3)	4 (16,17,18,19)	8 (32,33,34,35)	12 (48,49,50,51)
1 (4,5,6,7)	5 (20,21,22,23)	9 (36,37,38,39)	13 (52,53,54,55)
2 (8,9,10,11)	6 (24,25,26,27)	10 (40,41,42,43)	14 (56,57,58,59)
3 (12,13,14,15)	7 (28,29,30,31)	11 (44,45,46,47)	15 (60,61,62,63)

Cabe remarcar que el número de bloque es el número de etiqueta que tendremos almacenado en la memoria caché.

1) FIFO. Utilizamos este algoritmo de reemplazo y la misma secuencia de direcciones de memoria que en el caso anterior: 1, 2, 4, 10, 15, 1, 26, 27, 28, 29, 36, 37, 38, 40, 10, 11, 12, 13, 9, 30, 8, 12, 40, 17, 40.

La tabla siguiente muestra la evolución del contenido de la memoria caché e indica el número de bloque y las direcciones de memoria del bloque que hay en cada una de las 4 líneas de la memoria caché. Inicialmente la memoria caché está vacía. Cuando se produce un acierto, se indica con una E la línea donde se ha producido el acierto. Cada vez que hay un fallo, se indica con una letra F qué línea de la memoria caché se reemplazará y se actualiza el contenido llevando el nuevo bloque de memoria principal a esta línea de la memoria caché.

	Estado inicial	1	Fallo	2	4	Fallo	10	Fallo	15	Fallo
Línea 0		F	0 (0, 1, 2, 3)	E		0 (0, 1, 2, 3)		0 (0, 1, 2, 3)		0 (0, 1, 2, 3)
Línea 1					F	1 (4, 5, 6, 7)		1 (4, 5, 6, 7)		1 (4, 5, 6, 7)
Línea 2							F	2 (8, 9, 10, 11)		2 (8, 9, 10, 11)

Línea 3									F	3 (12, 13, 14, 15)

	26	Fallo	27	28	Fallo	29	36	Fallo	37	38
Línea 0	F	6 (24, 25, 26, 27)	E		6 (24, 25, 26, 27)			6 (24, 25, 26, 27)		
Línea 1		1 (4, 5, 6, 7)		F	7 (28, 29, 30, 31)	E		7 (28, 29, 30, 31)		
Línea 2		2 (8, 9, 10, 11)			2 (8, 9, 10, 11)		F	9 (36, 37, 38, 39)	E	E
Línea 3		3 (12, 13, 14, 15)			3 (12, 13, 14, 15)			3 (12, 13, 14, 15)		

	40	Fallo	10	Fallo	11	12	Fallo	13	9
Línea 0		6 (24, 25, 26, 27)	F	2 (8, 9, 10, 11)	E		2 (8, 9, 10, 11)		E
Línea 1		7 (28, 29, 30, 31)		7 (28, 29, 30, 31)		F	3 (12, 13, 14, 15)	E	
Línea 2		9 (36, 37, 38, 39)		9 (36, 37, 38, 39)			9 (36, 37, 38, 39)		
Línea 3	F	10 (40, 41, 42, 43)		10 (40, 41, 42, 43)			10 (40, 41, 42, 43)		

	30	Fallo	8	12	40	17	Fallo	40	Fallo
Línea 0		2 (8, 9, 10, 11)	E				2 (8, 9, 10, 11)	F	10 (40, 41, 42, 43)
Línea 1		3 (12, 13, 14, 15)		E			3 (12, 13, 14, 15)		3 (12, 13, 14, 15)
Línea 2	F	7 (28, 29, 30, 31)					7 (28, 29, 30, 31)		7 (28, 29, 30, 31)
Línea 3		10 (40, 41, 42, 43)			E	F	4 (16, 17, 18, 19)		4 (16, 17, 18, 19)

2) LRU. Utilizamos ahora el algoritmo de reemplazo LRU y la misma secuencia que en los casos anteriores: 1, 2, 4, 10, 15, 1, 26, 27, 28, 29, 36, 37, 38, 40, 10, 11, 12, 13, 9, 30, 8, 12, 40, 17, 40.

La tabla siguiente muestra la evolución del contenido de la memoria caché e indica el número de bloque y las direcciones de memoria del bloque que hay en cada una de las 4 líneas de la memoria caché. Inicialmente la memoria caché está vacía. Cuando se produce un acierto, se indica con una E la línea donde se ha producido el acierto. Cada vez que hay un fallo, se indica con una letra F qué línea de la memoria caché se reemplazará y se actualiza el contenido llevando el nuevo bloque de memoria principal a esta línea de la memoria caché.

	Estado inicial	1	Fallo	2	4	Fallo	10	Fallo	15	Fallo
Línea 0		F	0 (0, 1, 2, 3)	E		0 (0, 1, 2, 3)		0 (0, 1, 2, 3)		0 (0, 1, 2, 3)
Línea 1					F	1 (4, 5, 6, 7)		1 (4, 5, 6, 7)		1 (4, 5, 6, 7)
Línea 2							F	2 (8, 9, 10, 11)		2 (8, 9, 10, 11)

Línea 3										F	3 (12, 13, 14, 15)

	1	26	Fallo	27	28	Fallo	29	36	Fallo	37	38
Línea 0	E		0 (0, 1, 2, 3)			0 (0, 1, 2, 3)			0 (0, 1, 2, 3)		
Línea 1		F	6 (24, 25, 26, 27)	E		6 (24, 25, 26, 27)			6 (24, 25, 26, 27)		
Línea 2			2 (8, 9, 10, 11)		F	7 (28, 29, 30, 31)	E		7 (28, 29, 30, 31)		
Línea 3			3 (12, 13, 14, 15)			3 (12, 13, 14, 15)		F	9 (36, 37, 38, 39)	E	E

	40	Fallo	10	Fallo	11	12	Fallo	13	9
Línea 0	F	10 (40, 41, 42, 43)		10 (40, 41, 42, 43)			10 (40, 41, 42, 43)		
Línea 1		6 (24, 25, 26, 27)	F	2 (8, 9, 10, 11)	E		2 (8, 9, 10, 11)		E
Línea 2		7 (28, 29, 30, 31)		7 (28, 29, 30, 31)		F	3 (12, 13, 14, 15)	E	
Línea 3		9 (36, 37, 38, 39)		9 (36, 37, 38, 39)			9 (36, 37, 38, 39)		

	30	Fallo	8	12	40	17	Fallo	40
Línea 0		10 (40, 41, 42, 43)			E		10 (40, 41, 42, 43)	E
Línea 1		2 (8, 9, 10, 11)	E				2 (8, 9, 10, 11)	
Línea 2		3 (12, 13, 14, 15)		E			3 (12, 13, 14, 15)	
Línea 3	F	7 (28, 29, 30, 31)				F	4 (16, 17, 18, 19)	

3.6.3. Memoria caché asociativa por conjuntos

En una memoria caché asociativa por conjuntos con dos conjuntos de dos líneas, un bloque de memoria se puede encontrar en un único conjunto y dentro del conjunto en cualquier línea. A un bloque de memoria b le corresponderá la etiqueta e y lo asignaremos al conjunto j de la memoria caché, para determinar la etiqueta y el conjunto, dividimos el número de bloque b entre el número de líneas de cada conjunto:

$$e = b \text{ div } 2^c = \text{b div } 2$$
$$j = b \bmod 2^c = b \bmod 2$$

Por lo tanto, los bloques de memoria principal que se pueden asignar a cada conjunto de la memoria caché son los siguientes:

j: número de conjunto	Línea	Bloques	Bloque: etiqueta (7 bits) conjunto de 1 bit
0	0	0, 2, 4, 6, 8, ..., 254	0:0(0000000) 0, 2:1(0000001) 0, 4:2(0000010) 0, ..., 254:127(1111111) 0
	1		

j: número de conjunto	Línea	Bloques	Bloque: etiqueta (7 bits) conjunto de 1 bit
1	2	1, 3, 5, 7, 9, ..., 255	1:0(0000000) 1, 3:1(0000001) 1, 5:2(0000010) 1, ..., 255:127(1111111) 1
	3		

Mostramos a qué conjuntos de la memoria caché se asignan los primeros 16 bloques de memoria principal con las direcciones de memoria que contiene el bloque:

j: número de conjunto	b:e (a_0 ,a_1 ,a_2 ,a_3): bloque asignado : etiqueta (direcciones del bloque)			
0	0:0 (0,1,2,3) 2:1 (8,9,10,11)	4:2 (16,17,18,19) 6:3 (24,25,26,27)	8:4 (32,33,34,35) 10:5 (40,41,42,43)	12:6 (48,49,50,51) 14:7 (56,57,58,59)
1	1:0 (4,5,6,7) 3:1 (12,13,14,15)	5:2 (20,21,22,23) 7:3 (28,29,30,31)	9:4 (36,37,38,39) 11:5 (44,45,46,47)	13:6 (52,53,54,55) 15:7 (60,61,62,63)

Hay que remarcar que especificamos el número de bloque y el número de etiqueta, pero que el valor que tendremos realmente almacenado en la caché es tan solo la etiqueta asociada al bloque.

1) LRU. Utilizamos el algoritmo de reemplazo LRU y la misma secuencia que en los casos anteriores: 1, 2, 4, 10, 15, 1, 26, 27, 28, 29, 36, 37, 38, 40, 10, 11, 12, 13, 9, 30, 8, 12, 40, 17, 40.

La tabla siguiente muestra la evolución del contenido de la memoria caché e indica el número de bloque, la etiqueta del bloque y las direcciones de memoria del bloque que hay en cada una de las 4 líneas de la memoria caché. Inicialmente la memoria caché está vacía. Cuando se produce un acierto, se indica con una E la línea donde se ha producido el acierto. Cada vez que hay un fallo, se indica con una letra F qué línea de la memoria caché se reemplazará y se actualiza el contenido llevando el nuevo bloque de memoria principal a esta línea de la memoria caché.

	Estado inicial	1	Fallo	2	4	Fallo	10	Fallo	15	Fallo
Línea 0		F	0:0 (0, 1, 2, 3)	E		0:0 (0, 1, 2, 3)		0:0 (0, 1, 2, 3)		0:0 (0, 1, 2, 3)
Línea 1							F	2:1 (8, 9, 10, 11)		2:1 (8, 9, 10, 11)
Línea 2					F	1:0 (4, 5, 6, 7)		1:0 (4, 5, 6, 7)		1:0 (4, 5, 6, 7)
Línea 3									F	3:1 (12, 13, 14, 15)

	1	26	Fallo	27	28	Fallo	29	36	Fallo
Línea 0	E		0:0 (0, 1, 2, 3)			0:0 (0, 1, 2, 3)			0:0 (0, 1, 2, 3)
Línea 1		F	6:3 (24,25,26,27)	E		6:3 (24,25,26,27)			6:3 (24,25,26,27)
Línea 2			1:0 (4, 5, 6, 7)		F	7:3 (28,29,30,31)	E		7:3 (28,29,30,31)

Línea 3			3:1 (12, 13, 14, 15)			3:1 (12, 13, 14, 15)		F	9:4 (36,37,38,39)

	37	38	40	Fallo	10	Fallo	11	12	Fallo	13	9
Línea 0			F	10:5 (40,41,42,43)		10:5 (40,41,42,43)			10:5 (40,41,42,43)		
Línea 1				6:3 (24,25,26,27)	F	2:1 (8,9,10,11)	E		2:1 (8,9,10,11)		E
Línea 2				7:3 (28,29,30,31)		7:3 (28,29,30,31)		F	3:1 (12,13,14,15)	E	
Línea 3	E	E		9:4 (36,37,38,39)		9:4 (36,37,38,39)			9:4 (36,37,38,39)		

	30	Fallo	8	12	40	17	Fallo	40
Línea 0		10:5 (40,41,42,43)			E		10:5 (40,41,42,43)	E
Línea 1		2:1 (8,9,10,11)	E			F	4:2 (16,17,18,19)	
Línea 2		3:1 (12,13,14,15)		E			3:1 (12,13,14,15)	
Línea 3	F	7:3 (28,29,30,31)					7:3 (28,29,30,31)	

Podemos comparar las tasas de fallos de los casos anteriores y observar que, en esta secuencia de accesos, en el algoritmo LRU es donde se obtienen menos fallos, mientras que la memoria caché de asignación directa es la que obtiene más fallos.

La tasa de fallos en cada caso es:

Asignación directa:	$T_f = 14/25 = 0{,}56$
Completamente asociativa con FIFO:	$T_f = 13/25 = 0{,}52$
Completamente asociativa con LRU:	$T_f = 12/25 = 0{,}48$
Asociativa por conjuntos con LRU:	$T_f = 12/25 = 0{,}48$

3.7. Políticas de escritura

Cuando accedemos a la memoria caché, podemos hacer lecturas o escrituras; hasta ahora hemos visto la problemática de acceder a la memoria caché para leer un dato. Cuando se debe realizar una operación de escritura, aparecen nuevos problemas porque los datos que tenemos en la memoria caché son una copia de los datos que tenemos en la memoria principal y hay que garantizar la coherencia de los datos.

Analizaremos el caso con un único procesador y un único nivel de memoria caché entre el procesador y la memoria principal. Si tenemos más de un procesador con una memoria caché local para cada procesador, la modificación de un dato en una de estas memorias caché invalida el valor del dato en la memoria principal, pero también invalida el valor del dato si se encuentra en otra memoria caché. De manera parecida, si tenemos otros dispositivos que

puedan modificar directamente un dato de la memoria principal, el valor de este dato queda invalidado en las memorias caché donde se pueda encontrar. La gestión de la coherencia en estos sistemas es más compleja y no se analizará aquí.

A continuación comentaremos diferentes políticas para gestionar las escrituras y mantener la coherencia entre los datos de la memoria caché y la memoria principal:

1) Escritura inmediata (*write trough*): cuando se escribe en la memoria caché, también se escribe en la memoria principal transfiriendo todo el bloque que contiene el dato modificado; de esta manera en todo momento la copia que tenemos en la caché es idéntica a la que tenemos en la memoria principal. La política de escritura inmediata es la más fácil de implementar, pero su inconveniente es que produce un gran flujo de información entre la memoria caché y la memoria principal.

2) Escritura aplazada (*write back*): las escrituras se efectúan solo sobre la memoria caché. La memoria principal se actualiza cuando se elimina una línea de la memoria caché que ha sido modificada. Eso implica añadir algunos bits a cada línea de la memoria caché para saber si la línea se ha modificado o no.

Si hay que reemplazar una línea que ha sido modificada, primero es necesario copiar la línea modificada a la memoria principal y a continuación llevar el nuevo bloque, lo que aumenta significativamente el tiempo para acceder al dato.

Fallo en la escritura

Cuando se quiere hacer una escritura de una dirección que no está en la memoria caché, se producirá un fallo; este fallo se puede tratar de diferentes maneras:

a) Escribir directamente en memoria principal y no llevar el dato a la memoria caché. Esta técnica se puede utilizar en la escritura inmediata. Evitamos transferir el bloque a la caché pero no se tiene en cuenta la proximidad referencial, ya que es muy probable que haya nuevos accesos al mismo bloque de datos.

b) Llevar el bloque a la memoria caché y escribir simultáneamente en la caché y en la memoria principal. Esta técnica es la que se utiliza habitualmente en la escritura inmediata.

c) Llevar el bloque a la memoria caché y escribir solo en la caché. Esta técnica se utiliza habitualmente en escritura aplazada.

En máquinas reales se utilizan cada vez más políticas de escritura aplazada, pero el tratamiento de los fallos en caso de escritura es diferente, principalmente porque se consideran diferentes niveles de memoria caché y porque múltiples dispositivos (procesadores, DMA, canales de E/S) pueden acceder a la memoria.

4. Memoria interna

Todos los tipos de memoria interna se implementan utilizando tecnología de semiconductores y tienen el transistor como elemento básico de su construcción.

El elemento básico en toda memoria es la celda. Una celda permite almacenar un bit, un valor 0 o 1 definido por una diferencia de potencial eléctrico. La manera de construir una celda de memoria varía según la tecnología utilizada.

La memoria interna es una memoria de acceso aleatorio; se puede acceder a cualquier palabra de memoria especificando una dirección de memoria.

Una manera de clasificar la memoria interna según la perdurabilidad es la siguiente:

- Memoria volátil
  - SRAM (*static random access memory*)
  - RAM (*dynamic random access memory*)

- Memoria no volátil
  - ROM (*read only memory*)
  - PROM (*programmable read only memory*)
  - EPROM (*erasable programmable read only memory*)
  - EEPROM (*electrically erasable programmable read only memory*)

- Memoria flash

4.1. Memoria volátil

La memoria volátil es la memoria que necesita una corriente eléctrica para mantener su estado, de manera genérica denominada *RAM*.

Las memorias volátiles pueden ser de dos tipos:

1) SRAM. La memoria estática de acceso aleatorio (SRAM) implementa cada celda de memoria utilizando un flip-flop básico para almacenar un bit de información, y mantiene la información mientras el circuito de memoria recibe alimentación eléctrica.

Para implementar cada celda de memoria son necesarios varios transistores, típicamente seis, por lo que la memoria tiene una capacidad de integración limitada y su coste es elevado en relación con otros tipos de memoria RAM, como la DRAM; sin embargo, es el tipo de memoria RAM más rápido.

Usos de la memoria SRAM

La memoria SRAM se utiliza en la construcción de los registros del procesador y en la memoria caché.

2) DRAM. La memoria dinámica implementa cada celda de memoria utilizando la carga de un condensador. A diferencia de los flip-flops, los condensadores con el tiempo pierden la carga almacenada y necesitan un circuito de refresco para mantener la carga y mantener, por lo tanto, el valor de cada bit almacenado. Eso provoca que tenga un tiempo de acceso mayor que la SRAM.

Cada celda de memoria está formada por solo un transistor y un condensador; por lo tanto, las celdas de memoria son mucho más pequeñas que las celdas de memoria SRAM, lo que garantiza una gran escala de integración y al mismo tiempo permite hacer memorias más grandes en menos espacio.

Usos de la memoria DRAM

La memoria DRAM se utiliza en la construcción de la memoria principal del computador.

4.2. Memoria no volátil

La memoria no volátil mantiene el estado sin necesidad de corriente eléctrica.

Las memorias no volátiles pueden ser de diferentes tipos:

1) Memoria de solo lectura o ROM (*read only memory*). Tal como indica su nombre, se trata de memorias de solo lectura que no permiten operaciones de escritura y, por lo tanto, la información que contienen no se puede borrar ni modificar.

Este tipo de memorias se pueden utilizar para almacenar los microprogramas en una unidad de control microprogramada; también se pueden utilizar en dispositivos que necesitan trabajar siempre con la misma información.

La grabación de la información en este tipo de memorias forma parte del proceso de fabricación del chip de memoria. Estos procesos implican la fabricación de un gran volumen de memorias ROM con la misma información; es un proceso costoso y no es rentable para un número reducido de unidades.

2) Memoria programable de solo lectura o PROM (*programmable read only memory*). Cuando hay que fabricar un número reducido de memorias ROM con la misma información grabada, se recurre a otro tipo de memorias ROM: las memorias ROM programables (PROM).

A diferencia de las anteriores, la grabación no forma parte del proceso de fabricación de los chips de memoria, sino que se efectúa posteriormente con un proceso eléctrico utilizando un hardware especializado para la grabación de memorias de este tipo.

Como el proceso de programación no forma parte del proceso de fabricación, el usuario final de este tipo de memorias puede grabar el contenido según sus necesidades.

Cabe destacar que, tal como sucede con las memorias ROM, el proceso de grabación o programación solo se puede realizar una vez.

Este tipo de memoria tiene unas aplicaciones parecidas a las de las memorias ROM.

3) Memoria reprogramable mayoritariamente de lectura. Esta puede ser de tres tipos:

a) EPROM (*erasable programmable read only memory*). Se trata de memorias en las que habitualmente se hacen operaciones de lectura, pero cuyo contenido puede ser borrado y grabado de nuevo.

Hay que destacar que el proceso de borrar es un proceso que borra completamente todo el contenido de la memoria; no se puede borrar solo una parte. Para borrar, se aplica luz ultravioleta sobre el chip de memoria EPROM; para permitir este proceso, el chip dispone de una pequeña ventana sobre la cual se aplica la luz ultravioleta.

La grabación de la memoria se hace mediante un proceso eléctrico utilizando un hardware específico.

Tanto para el proceso de borrar como para el proceso de grabar hay que sacar el chip de memoria de su localización de uso habitual, ya que la realización de estas dos tareas implica la utilización de hardware específico.

b) EEPROM (*electrically erasable programmable read only memory*). Tiene un funcionamiento parecido a la EPROM, permite borrar el contenido y grabar información nueva; sin embargo, a diferencia de las memorias EPROM, todas las operaciones son realizadas eléctricamente.

Para grabar datos no hay que borrarlos previamente; se permite modificar directamente solo uno o varios bytes sin modificar el resto de la información.

Son memorias mayoritariamente de lectura, ya que el proceso de escritura es considerablemente más lento que el proceso de lectura.

c) Memoria flash. La memoria flash es un tipo de memoria parecida a las memorias EEPROM, en las que el borrado es eléctrico, con la ventaja de que el proceso de borrar y grabar es muy rápido. La velocidad de lectura es superior a la velocidad de escritura, pero las dos son del mismo orden de magnitud.

Este tipo de memoria no permite borrar la información byte a byte, sino que se deben borrar bloques de datos enteros; eso lleva a que todo el contenido de la memoria se pueda borrar en pocos segundos, pero también modera el proceso de escritura, ya que para escribir un dato nuevo hay que borrar previamente todo el bloque.

Tiene una capacidad de integración muy elevada y por este motivo también se utiliza para dispositivos de almacenamiento externo.

5. Memoria externa

La memoria externa está formada por dispositivos de almacenamiento secundario (discos magnéticos, CD, DVD, Blu-ray, etc.). Estos dispositivos se pueden encontrar físicamente dentro o fuera del computador.

La memoria externa es de tipo no volátil; por lo tanto, los datos que se quieran mantener durante un tiempo indefinido o de manera permanente se pueden almacenar en dispositivos de memoria externa.

El método de acceso varía según el dispositivo: generalmente los dispositivos basados en disco utilizan un método de acceso directo, mientras que otros dispositivos, como las cintas magnéticas, pueden utilizar acceso secuencial.

Los datos almacenados en la memoria externa son visibles para el programador en forma de bloques de datos, no como datos individuales (bytes), normalmente en forma de registros o ficheros. El acceso a estos dispositivos se lleva a cabo mediante el sistema de E/S del computador y es gestionado por el sistema operativo.

Ved también

La manera de acceder a los dispositivos de almacenamiento secundario se verá con más detalle cuando se analice el sistema de E/S.

5.1. Discos magnéticos

Los discos magnéticos son dispositivos formados por un conjunto de platos con superficies magnéticas y un conjunto de cabezales de lectura y escritura. La información se graba en estas superficies. Un solo dispositivo integra varios platos, que habitualmente utilizan las dos caras para almacenar la información.

Los platos y los cabezales son accionados por motores eléctricos. Los platos hacen un movimiento de rotación continuo y los cabezales se pueden mover de la parte más externa del disco a la parte más interna, lo que permite un acceso directo a cualquier posición del disco.

Son los dispositivos de almacenamiento secundario más importantes en cualquier computador y constituyen la base de cualquier sistema de memoria externa.

Son los dispositivos de memoria externa que proporcionan más capacidad de almacenamiento y los que tienen las prestaciones más elevadas. La capacidad de los discos magnéticos es del orden de Tbytes, el tiempo de acceso medio es de pocos milisegundos y pueden llegar a velocidades de transferencia del orden de un Gbyte por segundo.

5.1.1. RAID

Un sistema RAID[1] consiste en utilizar una colección de discos que trabajan en paralelo con el objetivo de mejorar el rendimiento y la fiabilidad del sistema de almacenamiento.

(1)RAID son las siglas de *redundant array of independent discs*, en español: matriz redundante de discos independientes.

Un conjunto de operaciones de E/S puede ser tratado en paralelo si los datos a los que se ha de acceder en cada operación se encuentran en diferentes discos; también una sola operación de E/S puede ser tratada en paralelo si el bloque de datos al cual hay que acceder se encuentra distribuido entre varios discos.

Un RAID está formado por un conjunto de discos que el sistema operativo ve como un solo disco lógico.

Los datos se pueden distribuir entre los discos físicos según configuraciones diferentes. Se utiliza información redundante para proporcionar capacidad de recuperación en el caso de fallo en algún disco.

La clasificación del tipo de RAID original incluye 7 niveles, del RAID 0 al RAID 6, en los que cada uno necesita un número diferente de discos y utiliza diferentes sistemas de control de la paridad y de detección y corrección de errores.

El control de un sistema RAID se puede llevar a cabo mediante software o hardware, con un controlador específico.

5.2. Cinta magnética

La cinta magnética es un dispositivo que utiliza una tecnología de almacenamiento parecida a la de los discos magnéticos; la diferencia básica es que la superficie magnética en la que se guarda la información se encuentra sobre una cinta de poliéster. Ya que es una cinta, se utiliza un método de acceso secuencial.

Son dispositivos lentos y se utilizan para hacer copias de seguridad de grandes volúmenes de datos o para almacenar datos a los que se accede con poca frecuencia.

5.3. Memoria flash

Las tendencias actuales incluyen dispositivos de almacenamiento construidos a partir de circuitos de memoria flash. El objetivo es sustituir los discos magnéticos ofreciendo características parecidas en cuanto el tiempo de acceso, tasa de transferencia de datos y capacidad de almacenamiento.

Como las memorias flash no tienen partes mecánicas ni superficies magnéticas, son más tolerantes a fallos y más adecuadas para entornos en los que la fiabilidad es muy importante. También hay dispositivos de este tipo para el gran público, pero que actualmente no rivalizan con los discos magnéticos, especialmente con respecto a la capacidad.

Usos de la memoria flash

La memoria flash se utiliza habitualmente en diferentes tipos de dispositivos de almacenamiento externo: tarjetas de memoria (Compact Flash, Secure Digital, etc.), memorias USB (*pendrive*) y unidades de estado sólido (SSD).

5.4. Disco óptico

Los discos ópticos son unidades de almacenamiento que utilizan luz láser para realizar operaciones de lectura y escritura sobre un soporte extraíble. Estas unidades pueden ser internas (conectadas a un bus interno del computador) o externas (conectadas por un bus externo).

Básicamente, se distinguen tres tipos de soportes: CD, DVD y Blu-ray (BD). Su capacidad máxima varía según el tipo de soporte; es del orden de los centenares de Mbytes en el caso del CD, del orden Gbytes en el caso de los DVD y de decenas de Gbytes en el caso de los Blu-ray.

El tipo de operaciones que se pueden realizar sobre el disco depende de su tipo: hay discos de solo lectura (CD-ROM, DVD-ROM, BD-ROM), discos que se pueden escribir solo una vez (CD-R, DVD+R, DVD-R, BD-R) y discos que se pueden escribir varias veces (CD-RW, DVD+RW, DVD-RW, BD-RE).

Habitualmente, una misma unidad es capaz de trabajar con soportes de diferentes tipos. Por ejemplo, una grabadora de DVD es capaz de leer CD-ROM, DVD-ROM y de leer y escribir CD-R, DVD+R, DVD-R, CD-RW, DVD+RW y DVD-RW.

La velocidad es inferior a la de los discos magnéticos. Están diseñados básicamente para hacer operaciones de lectura, ya que escribir implica un proceso de grabación relativamente lento, del orden de minutos, dependiendo de la cantidad de datos que se quiere almacenar.

5.5. Red

Utilizando los recursos de redes LAN o Internet, se puede disponer de almacenamiento de gran capacidad. Existen tipos diferentes, como SMB, NFS o SAN. Se puede acceder a cualquier dato almacenado en un ordenador conectado a la Red, sin límite de capacidad, y para ampliar la capacidad solo hay que añadir ordenadores nuevos a la red con capacidad de almacenamiento.

Es habitual medir la velocidad de transferencia en bits por segundo y está limitada por el ancho de la banda de la red.

Es adecuado utilizar este tipo de sistema cuando queremos gestionar grandes volúmenes de datos, los soportes físicos que se utilizan para almacenar los datos son discos magnéticos; si el objetivo es simplemente hacer copias de seguridad, se suelen utilizar cintas magnéticas.

Resumen

En este módulo, primero se ha ofrecido una introducción al sistema de memoria de un computador y se han explicado las características principales de los diferentes tipos de memorias:

- localización,
- capacidad,
- métodos de acceso,
- organización de los datos de una memoria,
- tiempo de acceso y velocidad,
- coste y
- características físicas.

Se ha introducido el concepto de jerarquía de memorias con el objetivo de conseguir que el procesador, cuando accede a un dato, este se encuentre en el nivel más rápido y así conseguir tener una memoria a un coste moderado, con una velocidad próxima al nivel más rápido y la capacidad del nivel mayor.

La jerarquía de memorias de un computador se organiza en niveles diferentes:

- Registros.
- Memoria interna.
  - Memoria caché.
  - Memoria principal.

- Memoria externa.

Hemos visto el concepto de la proximidad referencial, gracias al cual la jerarquía de memorias funciona, y hemos distinguido dos tipos:

- proximidad temporal y
- proximidad espacial.

Se han explicado detalladamente las características y el funcionamiento de la memoria caché:

- La organización de la memoria caché.
- El concepto de acierto y fallo y los índices para medir el rendimiento.
- Las diferentes políticas de asignación que hay a la hora de llevar un bloque de memoria principal a la memoria caché:
  - asignación directa,
  - asignación completamente asociativa y

  - asignación asociativa por conjuntos.

- Los diferentes algoritmos de reemplazo que se pueden utilizar cuando hay que sustituir el contenido de una línea de la memoria caché:
  - FIFO,
  - LFU,
  - LRU y
  - aleatorio.

- La comparativa entre los diferentes sistemas de memoria caché.
- Cómo gestionar las escrituras en una memoria caché.

A continuación se han explicado los diferentes tipos de memoria interna y cómo se pueden clasificar según su perdurabilidad en memorias volátiles y no volátiles, y los diferentes tipos que podemos encontrar dentro de cada categoría.

Finalmente, en el último apartado, se han descrito los dispositivos más habituales que conforman la memoria externa de un computador y que son gestionados por el sistema operativo mediante el sistema de E/S.

Sistema de entrada/salida

Miquel Albert Orenga
Gerard Enrique Manonellas

PID_00177074

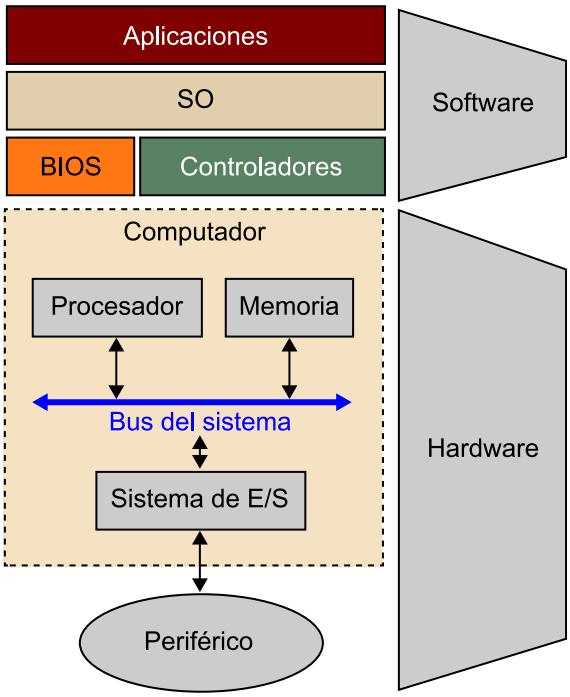
Universitat Oberta de Catalunya
www.uoc.edu

Índice

Introducción

Todo computador necesita llevar a cabo intercambio de información con personas u otros computadores mediante unos dispositivos que denominamos de manera genérica **dispositivos periféricos**. Para hacer una operación de E/S entre el computador y un periférico, es necesario conectar estos dispositivos al computador y gestionar de manera efectiva la transferencia de datos. Para hacerlo, el computador dispone del **sistema de entrada/salida (E/S)**.

Este sistema de E/S es la interfaz que tiene el computador con el exterior y el objetivo que tiene es facilitar las operaciones de E/S entre los **periféricos** y la **memoria** o los **registros del procesador**. Para gestionar las operaciones de E/S es necesario un hardware y la ayuda de un software.



Dada la gran variedad de periféricos, es necesario dedicar un hardware y un software específicos para cada uno. Por este motivo se ha intentado normalizar la interconexión de los periféricos y el computador mediante lo que se denomina **módulos de E/S** o **controladores de E/S**. Eso nos permite tener, por una parte, una conexión, entre el módulo de E/S y el periférico, específica y con unas características propias que difícilmente se pueden generalizar para utilizarlas en otros dispositivos y, por otra parte, una conexión entre los módulos de E/S y el computador común a todos los controladores, pero estos módulos, además de permitir la conexión de los periféricos al computador, disponen de la lógica necesaria para tener cierta capacidad de procesamiento y gestionar las transferencias de información.

Hay que tener presente que la gestión global del sistema de E/S de un computador la hace el sistema operativo (SO). Las técnicas para controlar este sistema de E/S las utiliza el SO y el programador cuando quieren acceder al periférico, pero en las máquinas actuales, a causa de la complejidad de controlar y gestionar los periféricos, el acceso se lleva a cabo generalmente mediante llamadas al SO, que es quien gestiona la transferencia. El conjunto de rutinas que permiten controlar un determinar periférico es lo que denominamos habitualmente **programas controladores** o ***drivers*** y cuando el SO quiere hacer una operación de E/S con un periférico llama a una de estas rutinas.

Este módulo se centra en el estudio del sistema de E/S y hablaremos de las principales técnicas utilizadas y de qué características debe tener el hardware y el software necesario para gestionar las diferentes maneras de realizar la transferencia de información entre el computador y el periférico.

Objetivos

Con el estudio de este módulo se pretende que el estudiante alcance los objetivos siguientes:

1. Conocer los aspectos básicos del sistema de E/S de un computador.

2. Aprender las técnicas básicas de E/S.

3. Entender las ventajas de cada una de estas técnicas para mejorar el rendimiento del computador.

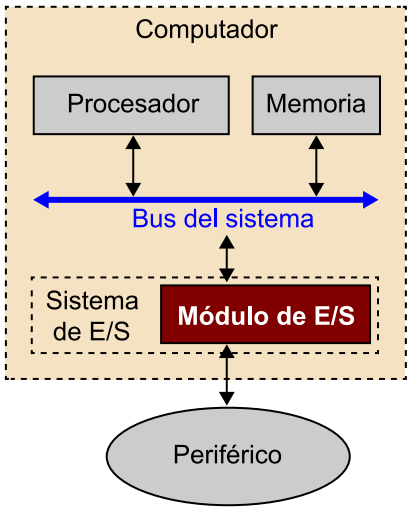
4. Tener unos conocimientos básicos de los tipos de dispositivos que podemos conectar al computador y cómo se comunican con el procesador mediante el sistema de E/S.

1. Aspectos básicos del E/S

Cuando hablamos de E/S de información entre un computador y un periférico lo hacemos siempre desde el punto de vista del computador. Así, decimos que es una **transferencia de entrada** cuando el periférico es el emisor de la información y tiene como receptor el computador (procesador o memoria) y decimos que es una **transferencia de salida** cuando el computador es el emisor de la información y tiene como receptor el periférico.

De manera más concreta, toda operación de E/S que se lleva a cabo entre el computador y un periférico es solicitada y gobernada desde el procesador, es decir, es el procesador quien determina en qué momento se debe hacer y con qué periférico, si la operación es de lectura o escritura, qué datos se han de transferir, y también quién da la operación por acabada.

Para llevar a cabo la operación de E/S, hemos de conectar el periférico al computador. Para hacerlo, es necesario que el computador disponga de unos dispositivos intermedios por donde ha de pasar toda la información que intercambia el computador con el periférico y que nos permite hacer una gestión y un control correctos de la transferencia. Estos dispositivos los llamamos de manera genérica **módulo de E/S**.



Puede parecer lógico conectar el periférico directamente al bus del sistema del computador, pero esta opción no es factible básicamente por dos razones:

- La necesidad de gestionar una gran variedad de periféricos con unas características muy específicas y diferenciadas. Esto hace muy complejo añadir

la lógica necesaria dentro del procesador para gestionar esta gran diversidad de dispositivos.

- La diferencia de velocidad entre sí, en la que, salvo casos excepcionales, el procesador es mucho más rápido que el periférico. Por un lado, hay que asegurar que no se pierdan datos y, por otro, garantizar principalmente la máxima eficiencia del procesador, pero también de los otros elementos del computador.

Así pues, para hacer una operación de E/S, el módulo de E/S nos debe permitir establecer, por una parte, **mecanismos de control** para determinar el inicio y el final de la operación de E/S, la cantidad de información que hay que transmitir, la detección de errores, etc., y, por otra parte, **mecanismos para hacer la transferencia de datos** considerando aspectos como la manera de dirigir el periférico, la conversión serie/paralela de la información, la conversión de códigos, la sincronización, etc. Estos mecanismos se reparten entre la unidad de control del procesador, el módulo de E/S y los programas de E/S.

Cuando queremos hacer la operación de E/S, hemos de diferenciar el caso de una transferencia individual, en la que se transmite un solo dato y el control de la transferencia es muy simple (leer una tecla, mirar si se ha hecho un clic en el ratón), y la transferencia de bloques, que se basa en una serie de transferencias individuales y en la que se necesita un control mayor de todo el proceso (leer un fichero, actualizar el contenido de la pantalla).

Otro aspecto importante que hay que considerar, dado que podemos tener conectados al computador una gran variedad de periféricos, es que si se desencadenan operaciones de E/S de manera simultánea, el sistema de E/S del computador debe disponer de los mecanismos necesarios para gestionarlas sin que se pierdan datos.

1.1. Estructura del sistema de E/S del computador

Los elementos principales que forman el sistema de E/S son los siguientes:

- los periféricos,
- los módulos de E/S,
- los sistemas de interconexión externos y
- el mapa de memoria e instrucciones de E/S.

A continuación haremos una breve descripción de estos elementos y de cómo interactúan entre sí.

1.1.1. Periféricos

Los periféricos son dispositivos que se conectan al computador mediante los módulos de E/S y que sirven para almacenar información o para llevar a cabo un tipo determinado de comunicación con el exterior con humanos, con máquinas o con otros computadores.

La clasificación más habitual es la siguiente:

- Para la interacción con humanos:
  - Entrada.
  - Salida.

- Para la interacción con otros computadores o sistemas físicos (en los que las operaciones que se hacen son generalmente de E/S):
  - Almacenamiento.
  - Comunicación.

En un periférico distinguimos habitualmente dos partes: una parte mecánica y una parte electrónica. La parte mecánica hace funcionar los elementos principales que forman el periférico, como el motor para hacer girar un disco o mover el cabezal de una impresora, el botón de un ratón o el láser de un dispositivo óptico. La parte electrónica nos permite, por una parte, generar las señales eléctricas para gestionar los elementos mecánicos y, por otra parte, hacer la conversión de los datos provenientes del computador a señales eléctricas o al revés.

La conexión física entre un periférico y el computador se lleva a cabo mediante lo que denominamos **sistema de interconexión de E/S**. Este sistema de interconexión de E/S nos permite hacer la gestión de las señales de control, de estado y de datos necesarias para llevar a cabo una transferencia de información que, como veremos más adelante, es gestionada desde el módulo de E/S del computador.

En este módulo nos centraremos en analizar la transferencia de información entre un periférico y el computador mediante los módulos de E/S.

Nota

En este módulo no analizaremos la estructura y el funcionamiento del periférico; habrá suficiente con ver el periférico como un elemento capaz de enviar o recibir una determinada cantidad de información, a una determinada velocidad, mediante el sistema de interconexión de E/S. No analizaremos tampoco las características técnicas que permiten al periférico tener estas prestaciones.

La cantidad de información que puede enviar o recibir el periférico por unidad de tiempo la denominamos **velocidad de transferencia** y generalmente se expresa en bits o bytes por segundo.

La velocidad de transferencia puede ir de unos pocos bits por segundo a gigabytes por segundo, pero hemos de tener presente que un computador puede llegar a trabajar a velocidades bastante superiores y hemos de garantizar que durante una transferencia no se pierdan datos.

1.1.2. Módulos de E/S

Un módulo de E/S es un controlador de uno o varios periféricos que establece una interfaz entre el periférico y el computador (procesador y memoria) para facilitar la comunicación entre el uno y el otro de manera que buena parte de los detalles técnicos del periférico queden ocultos al resto del computador.

Nota

La complejidad de un módulo de E/S puede variar mucho. Por este motivo, aquí haremos una descripción general de las partes y características básicas más importantes. Veremos otras características más específicas cuando analicemos las diferentes técnicas de E/S.

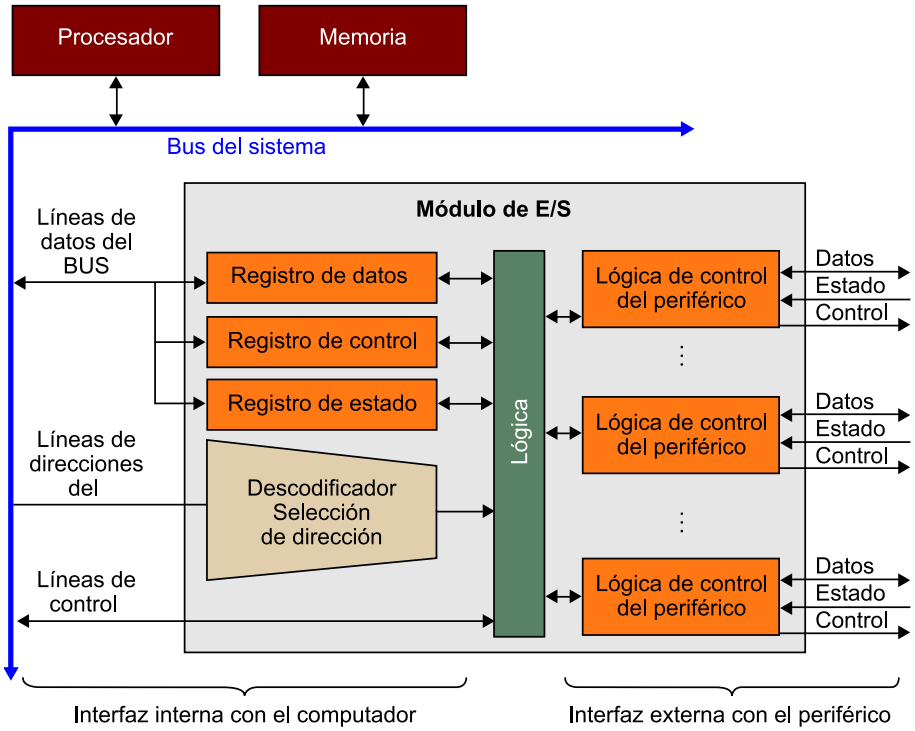
Del módulo de E/S distinguimos tres partes básicas:

1) Una interfaz interna normalizada con el resto del computador mediante el bus de sistema que nos da acceso al banco de registros del módulo de E/S.

2) Una interfaz externa específica para el periférico que controla. Habitualmente la conexión con el periférico se realiza mediante un sistema de interconexión normalizado de E/S.

3) La lógica necesaria para gestionar el módulo de E/S. Es responsable del paso de información entre la interfaz interna y externa.

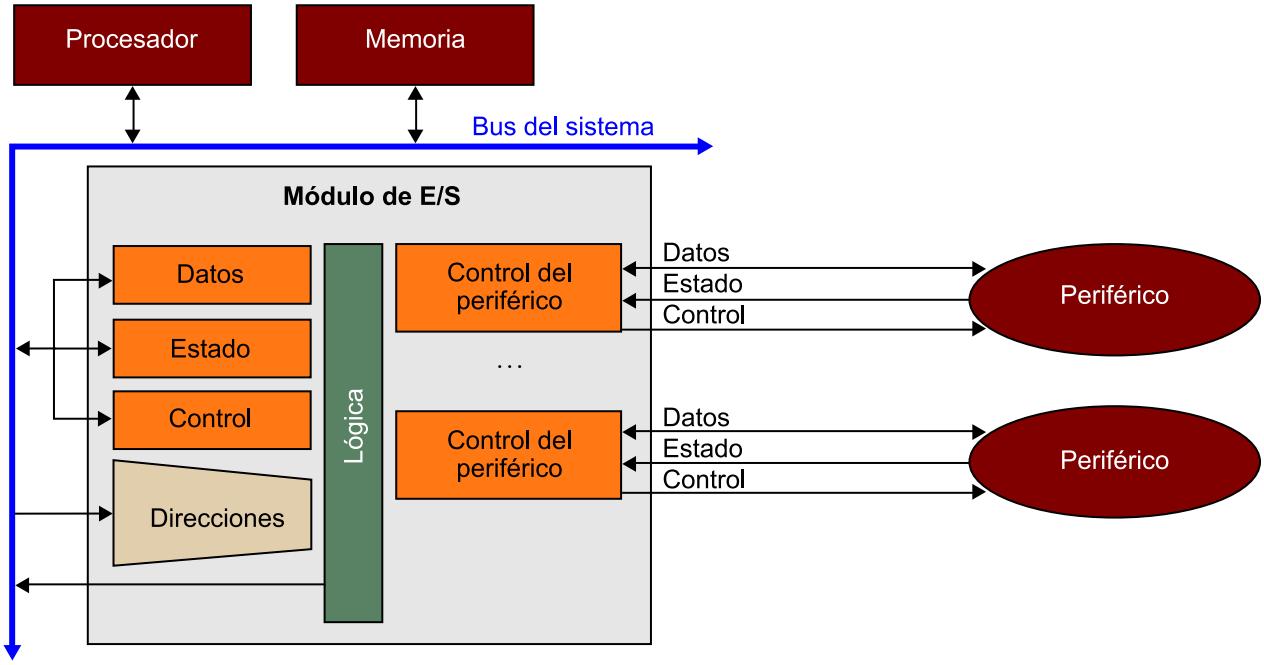
En la siguiente figura podéis ver el esquema general de un módulo de E/S.



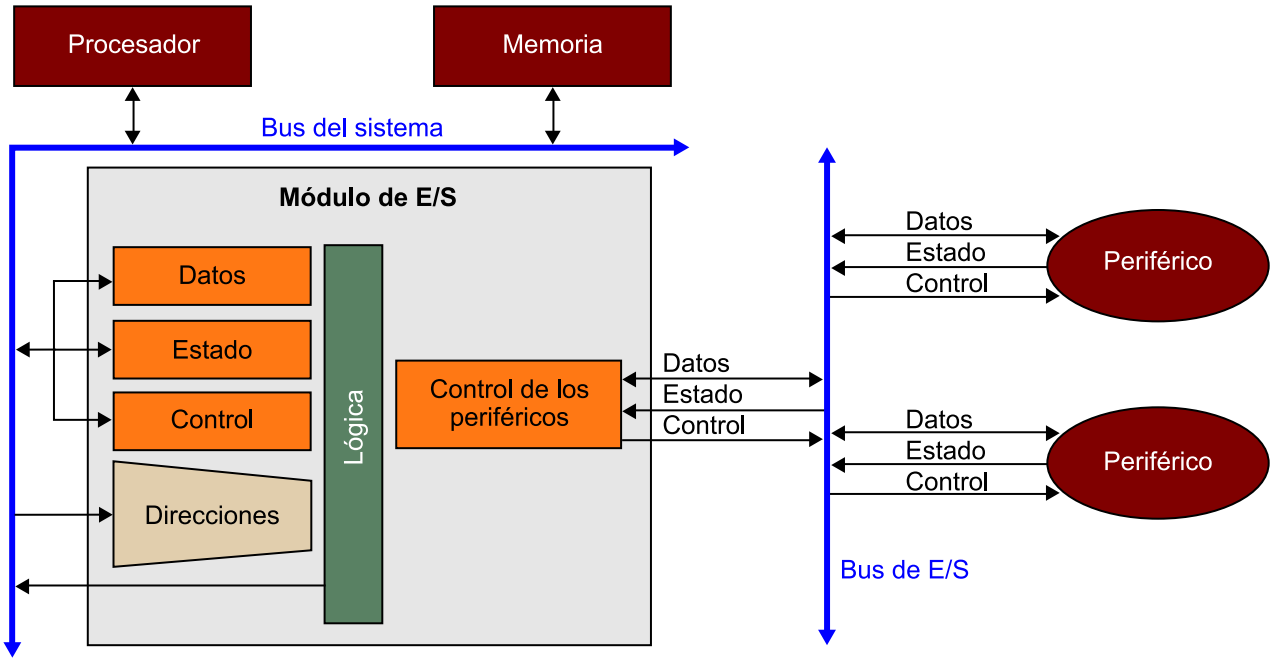
La forma de comunicación entre el módulo de E/S y el periférico es específica para cada periférico. Lógicamente, depende de las características del periférico que queremos controlar, pero también del sistema de interconexión utilizado para comunicarse. Esta conexión tiene habitualmente unas especificaciones normalizadas y adaptadas al tipo de transferencia que se debe realizar y lo denominamos **sistema de interconexión de E/S**. Esto hace que la interfaz externa tenga unas características propias que difícilmente se pueden generalizar.

Cuando un módulo de E/S gestiona más de un periférico, hay dos configuraciones básicas, la conexión punto a punto y la multipunto, aunque las configuraciones que encontramos en máquinas reales son muy variadas. En la conexión punto a punto el módulo de E/S gestiona la comunicación con cada periférico individualmente; no es un bus de E/S, pero sí que tiene unas especificaciones normalizadas de la conexión, de manera parecida a las de un bus normalizado de E/S. En la conexión multipunto el módulo de E/S gestiona la comunicación con los periféricos mediante un bus normalizado de E/S y hay que añadir la lógica para acceder al bus.

Conexión punto a punto entre el módulo de E/S y el periférico



Conexión multipunto entre el módulo de E/S y el periférico



La comunicación entre los módulos de E/S y el computador es siempre la misma para todos los módulos. Esta comunicación se establece mediante el bus del sistema, de modo que el procesador ve el módulo de E/S como un espacio de memoria, pero estas direcciones, físicamente, corresponden (están mapeadas) a cada uno de los registros que tiene el módulo de E/S del computador y se denominan habitualmente **puertos de E/S**. De esta manera conseguimos que la comunicación entre el computador y el módulo de E/S se lleve a cabo mediante instrucciones de transferencia para leer y escribir en sus registros, de una manera muy parecida a como hacemos para acceder a la memoria.

Estos registros se pueden agrupar según el tipo de señales o el tipo de información que necesitamos para hacer una gestión correcta del periférico:

- Registros de control.
- Registros de estado.
- Registros de datos.

Para gestionar la comunicación entre el procesador y el módulo de E/S son necesarios diferentes tipos de señales.

Las **señales de control** las utilizamos generalmente para dar órdenes al módulo de E/S, como empezar o parar una transferencia, seleccionar modos de operación del periférico o indicar acciones concretas que debe hacer el periférico, como comprobar si está disponible. Estas señales se pueden recibir directamente de las líneas de control del bus del sistema o de las líneas de datos del bus del sistema y se almacenan en el *registro de control*.

Las **señales de estado** nos dan información del estado del módulo de E/S, como saber si el módulo está disponible o está ocupado, si hay un dato preparado, si se ha acabado una operación, si el periférico está puesto en marcha o parado, qué operación está haciendo, o si se ha producido algún error y qué tipo de error. Estas señales se actualizan generalmente mediante la lógica del módulo de E/S y se almacenan en el *registro de estado*.

Los **datos** son la información que queremos intercambiar entre el módulo de E/S y el procesador mediante las líneas de datos del bus del sistema y se almacenan en el *registro de datos*.

Las **direcciones** las pone el procesador en el bus de direcciones y el módulo de E/S debe ser capaz de reconocer estas direcciones (direcciones de los puertos de E/S) correspondientes a los registros de este módulo. Para saber si la dirección corresponde a uno de los registros del módulo utilizamos un descodificador. Este descodificador puede formar parte del módulo de E/S o de la misma lógica del bus del sistema.

Hay que tener presente que un computador puede tener definidos diferentes tipos de conexiones normalizadas entre el módulo de E/S y el resto del computador. Tanto el módulo de E/S como el computador se deben adaptar a estos tipos de conexión, de modo que tenemos módulos de E/S adaptados a las diferentes normas, y eso tiene implicaciones con respecto al hardware y a la manera de gestionar las operaciones de E/S, como veremos más adelante cuando analicemos las técnicas básicas de E/S.

1.1.3. Sistemas de interconexión externos

En un computador distinguimos dos tipos básicos de sistemas de interconexión: los internos del computador, que nos permiten conectar el procesador, la memoria y el sistema de E/S y que denominamos **bus del sistema**, y los externos al computador, que nos permiten conectar el sistema de E/S con los diferentes periféricos y que denominamos **sistemas de interconexión de E/S** o **buses de E/S**.

Desde el punto de vista del sistema de E/S, el **bus del sistema** nos permite la comunicación entre los módulos de E/S y el resto del computador. Este bus tiene una estructura jerárquica formada por diferentes tipos de buses para aislar los elementos más rápidos de los más lentos y, de esta manera, mejorar las prestaciones del sistema.

Los **sistemas de interconexión de E/S** o **buses de E/S** nos permiten la comunicación de los módulos de E/S con los periféricos o dispositivos con suficiente autonomía para gestionar una operación de E/S y los módulos de E/S. Las características de estos sistemas se adaptan al tipo de dispositivos que hemos de conectar.

Físicamente, un sistema de interconexión está formado por un conjunto de hilos conductores o líneas que interconectan diferentes dispositivos. Por estas líneas circulan señales eléctricas que los dispositivos que tenemos conectados pueden interpretar como señales binarias. Hay tres tipos de señales básicas: señales de datos, de direcciones y de control.

Las siguientes son las características principales de los sistemas de interconexión externos:

- **Ancho de banda**: la cantidad máxima de información que podemos transmitir por unidad de tiempo. Se expresa en bits o bytes por segundo.

- **Serie/paralelo**: en una interconexión paralela hay varias líneas que conectan el módulo de E/S y el periférico y pueden transmitir varios bits simultáneamente mediante las líneas de datos. En una interconexión serie solo hay una línea para transmitir los datos y los bits se han de transmitir uno a uno. Tradicionalmente las interconexiones de tipo serie eran para dispositivos lentos y las de tipo paralelo, para dispositivos más rápidos, pero con las nuevas generaciones de sistemas de interconexión serie de alta velocidad las paralelas cada vez son menos utilizadas.

- **Punto a punto/multipunto**: una interconexión punto a punto tiene un enlace dedicado entre el módulo de E/S y el periférico. En una interconexión multipunto, que habitualmente se denomina *bus de E/S* y que dispone

de un enlace compartido entre diferentes periféricos y el módulo de E/S, el hecho de tener múltiples dispositivos conectados a un mismo conjunto de líneas hace necesario establecer mecanismos para controlar el acceso.

Otras características típicas de los buses de E/S son:

- **Modo de operación síncrono/asíncrono/semisíncrono**: si el control de los accesos al bus es controlado o no por un reloj.

- **Multiplexación de datos y direcciones**: si las líneas del bus están dedicadas a datos y direcciones o si se comparten las mismas líneas para datos y para direcciones.

- **Arbitraje centralizado y distribuido**: es centralizado cuando un único árbitro o controlador determina quién tiene que acceder al bus en cada momento y es distribuido cuando los dispositivos conectados al bus disponen de capacidad de controlar el acceso al bus.

- **Tipos de operaciones de lectura/escritura**: diferentes maneras de hacer las operaciones de lectura y escritura, como la transferencia de bloques o la combinación de operaciones de lectura y escritura.

- **Esquema de direccionamiento**: hay dos tipos básicos: el *direccionamiento lógico*, que es cuando el espacio de direccionamiento de memoria es común a todos los dispositivos y cada uno dispone de un rango de direcciones único y los dispositivos para descodificar la dirección para saber si esta dirección está dentro de su rango; el *direccionamiento geográfico*, que es cuando cada dispositivo tiene una dirección propia y se separa la identificación del módulo de la selección de la dirección dentro del módulo.

Normalización de un bus

La normalización de un bus consiste en dar una descripción detallada y precisa de las características que tiene a diferentes niveles. Los niveles principales son el mecánico (tamaños y conectores), el eléctrico (tipos de señales eléctricas que circulan por las líneas), el lógico (descripción de todas las señales: direcciones, datos y control) y el cronograma (cuál es la secuencia de señales para hacer la transferencia de un dato o de un bloque).

La normalización de un bus facilita la divulgación y también el diseño de los dispositivos que hemos de conectar. Generalmente, la mayoría de las normalizaciones son definidas por organismos internacionales. El más conocido en el ámbito de la electricidad y la electrónica es el IEEE.

1.1.4. Mapa de memoria e instrucciones de E/S

Tal como hemos explicado, el procesador ve el banco de registros del módulo de E/S como un espacio de memoria dirigible, de manera que cada registro del módulo de E/S tiene asociada (mapeada) una dirección única. Veamos ahora cómo hemos de acceder a estas direcciones, que denominamos **puertos de E/S**.

Para identificar los registros del módulo de E/S hay dos posibilidades:

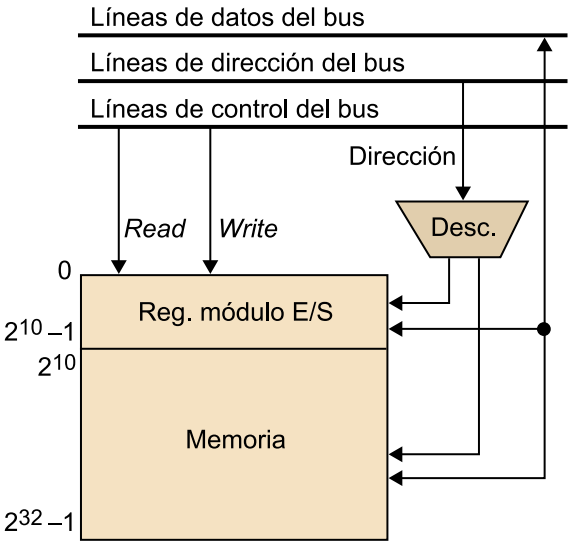
1) Mapa común de memoria y E/S. No hay distinción entre direcciones de memoria y registros de E/S. Para acceder a los registros se utilizan descodificadores que se activan a partir de las líneas del bus de direcciones y se utilizan las mismas señales de control (READ/WRITE) que se emplean para seleccionar la

memoria. Podemos acceder a los puertos de E/S con las mismas instrucciones de transferencia que utilizamos para acceder a memoria (MOV y las variantes que tiene).

Este sistema tiene la ventaja de que nos permite aprovechar el amplio conjunto de instrucciones del que dispone el procesador para acceder a memoria y podemos hacer programas más eficientes. La principal desventaja es que hemos de dedicar una parte del valioso espacio de memoria a las direcciones de E/S y hay que ser cuidadosos con la cantidad de espacio asignado, ya que este espacio va en detrimento del espacio de memoria disponible, pero cada vez este problema es menos importante a causa del incremento del espacio de memoria dirigible.

Ejemplo de mapa común de memoria

En la siguiente figura tenéis un ejemplo de conexión de una memoria de 2^{32} direcciones y 2^{10} puertos de E/S utilizando un mapa común y un mapa independiente de memoria y E/S.

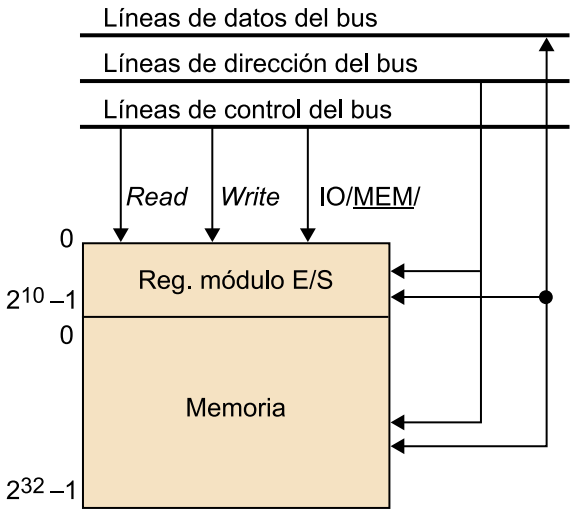


2) Mapa independiente de E/S. Hay distinción entre direcciones de memoria y registros de E/S. Las líneas de direcciones se suelen compartir, pero hay que añadir algunas líneas de control para distinguir si un acceso es a memoria o a un puerto de E/S. También son necesarias instrucciones específicas de E/S. Las instrucciones utilizadas habitualmente son IN (para leer del puerto de E/S) y OUT (para escribir en el puerto de E/S).

Este sistema tiene la ventaja de que la memoria dispone de todo el rango de direcciones y la clara desventaja de que dispone de un reducido número de instrucciones específicas de E/S que solo disponen de los modos de direccionamiento más básicos para acceder a los puertos de E/S.

Ejemplo de mapa independiente

En la siguiente figura tenéis un ejemplo de conexión de una memoria de 2^{32} direcciones y 2^{10} puertos de E/S utilizando un mapa independiente de memoria y E/S.



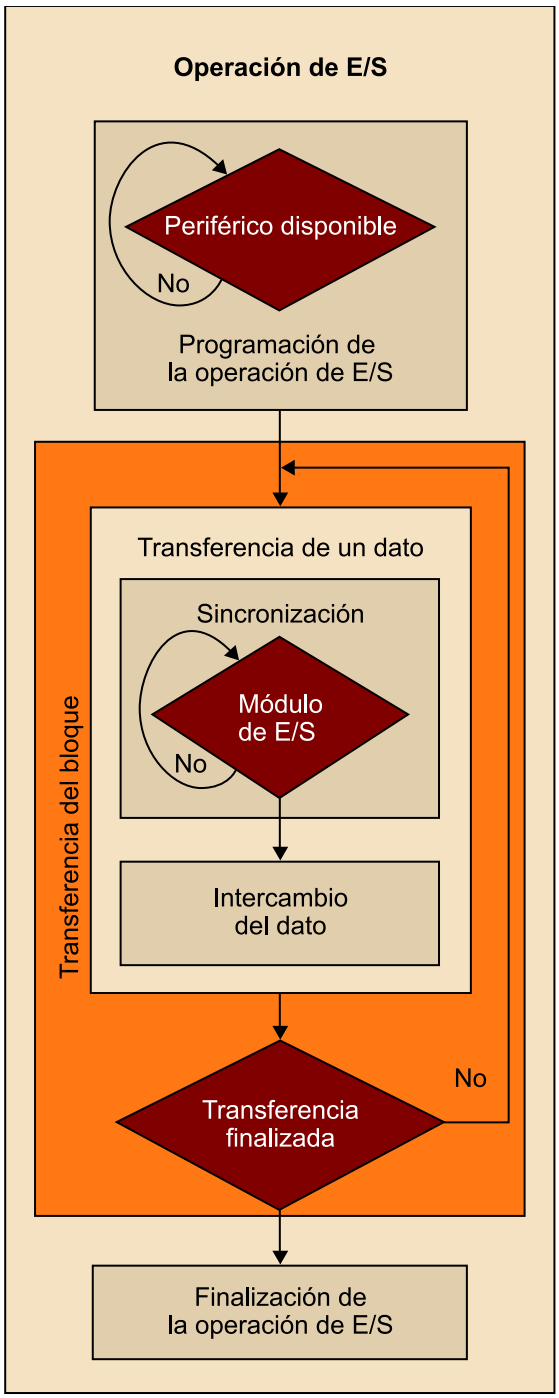
Una cuestión interesante que hay que considerar es que si utilizamos instrucciones específicas de E/S, el procesador puede saber en la fase de descodificación de la instrucción que haremos una operación de E/S. Esto hace más fácil la gestión de las señales de control para llevar a cabo esta operación y también hace más fácil establecer un mecanismo de protección para controlar que estas instrucciones solo se puedan ejecutar en modo supervisor. Si no tenemos instrucciones específicas de E/S, hasta que no se hace el acceso al puerto de E/S (se descodifica la dirección), no podemos saber si aquel acceso se debe permitir o no y eso complica la gestión.

1.2. Operación de E/S

Para garantizar que una transferencia de datos se realiza con éxito, es necesario definir una serie de pasos y establecer una serie de mecanismos que nos permitan controlar en todo momento la operación de E/S.

En el proceso global de una operación de E/S distinguimos los pasos siguientes:

1) Programación de la operación de E/S.
2) Transferencia de un bloque. Transferencia de un dato (para cada dato del bloque):
a) Sincronización.
b) Intercambio del dato.
3) Finalización de la operación de E/S.



1.2.1. Programación de la operación de E/S

La programación de la operación de E/S es el proceso para indicar al módulo de E/S cómo se debe llevar a cabo la transferencia.

Esta programación consiste en ejecutar un pequeño conjunto de instrucciones que verifican si el periférico está disponible para iniciar una transferencia de datos, actualizan los registros del módulo de E/S, principalmente los registros

de control para dar instrucciones al periférico, y también nos puede servir para inicializar las variables o registros que necesite el procesador para llevar a cabo la transferencia de datos.

1.2.2. Transferencia de datos

La transferencia de datos es la fase donde se hace realmente la transferencia de información entre el procesador y el módulo de E/S y es la fase que veremos con más detalle analizando diferentes técnicas.

De manera genérica, en la transferencia de cada dato dentro de un bloque distinguimos dos fases principales: la sincronización y el intercambio. Estas dos fases se repiten para cada dato del bloque.

La **sincronización** es donde se establece un mecanismo para conseguir que el dispositivo más rápido espere que el dispositivo más lento esté preparado para llevar a cabo el *intercambio del dato*.

Este mecanismo nos garantiza que no se dejen datos sin procesar, pero la consecuencia es que la transferencia de datos se realiza a la velocidad del dispositivo más lento.

Durante la fase de sincronización, el procesador (o el elemento que controle la transferencia) debe ser capaz de detectar cuándo está disponible el periférico para hacer el intercambio de un dato. El módulo debe informar de que el periférico está preparado.

Ejemplo del proceso de sincronización

Supongamos que el procesador envía diez datos (D0, D1, D2, D3, D4, D5, D6, D7, D8, D9) a un periférico. El procesador puede enviar un dato cada 2 ms y el periférico tarda 5 ms en procesar cada dato.

	Tiempo (ms)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Sin sincronización	Proc. envía	D0		D1		D2		D3		D4		D5		D6		D7		D8		D9		FI
	Perif. lee	↓ D0					↓ D2					↓ D5					↓ D7					↓ FI
Con sincronización	Proc. envía	D0					D1					D2					D3					...
	Perif. lee	↓ D0					↓ D1					↓ D2					↓ D3					↓ ...

Cuando no hay sincronización, observamos que hay datos que se pierden (D1, D3, D4, D6, D8, D9). Cuando el periférico quiere leer el dato siguiente, el procesador ya ha enviado nuevos y los anteriores se han perdido porque se han sobreescrito en el registro de datos del módulo de E/S. Con sincronización, el procesador se espera 3 ms (zona sombreada) y, cuando el periférico está preparado, el procesador envía el nuevo dato. De esta manera, no se pierde ningún dato, pero la transferencia se hace al ritmo que marca el periférico, que es el más lento.

Memorias intermedias

Este problema se puede reducir utilizando una pequeña memoria intermedia (generalmente denominada *buffer*) para almacenar temporalmente la entrada o salida de datos. Pero si la diferencia de velocidad es grande o los bloques de datos son grandes, la única manera de garantizar que no se pierdan datos es tener un mecanismo de sincronización.

El **intercambio del dato** es la fase en la que se envía realmente el dato. Sabiendo que tanto el emisor como el receptor están preparados (se ha hecho la *sincronización*), se verifica que el dato se ha recibido correctamente, se deja todo preparado y se indica que ya se puede iniciar una nueva transferencia.

En una operación de entrada (lectura), el periférico envía el dato al módulo de E/S, que hace una verificación para detectar posibles errores. Generalmente, esta verificación es una validación muy simple, como verificar el bit de paridad, y deja el dato disponible para el procesador en el registro de datos del módulo de E/S para que el procesador lo pueda leer.

En una operación de salida (escritura), el procesador escribe el dato en el registro de datos del módulo de E/S y después el módulo de E/S lo envía al periférico.

Veamos cómo sería una operación de E/S en la que el procesador lee un dato de un periférico:

Nota

Hay que tener presente que el proceso que aquí describimos puede variar ligeramente dependiendo del tipo de periférico con el que hacemos la transferencia o de la técnica de E/S utilizada.

1) El procesador comprueba el estado del periférico, lee el registro de estado del módulo de E/S y mira los bits que indican si el periférico está disponible. Puede ser necesario dar una orden accediendo al registro de control para que el módulo de E/S consulte el estado del periférico y actualice el registro de estado.

2) Si el periférico está disponible, el procesador solicita el dato enviando una orden al módulo de E/S. El procesador debe acceder al puerto que corresponde al registro de control del módulo de E/S para escribir en él y así indicar la operación que queremos que haga. Hemos programado la transferencia.

Acceso a los puertos de E/S

Recordad que para acceder a un puerto de E/S (registro de un módulo de E/S), utilizamos instrucciones de transferencia si el procesador tiene un mapa común de memoria y E/S e instrucciones específicas de E/S si el computador tiene un mapa independiente de memoria y E/S.

En estas instrucciones debemos especificar la dirección de uno de los registros del módulo de E/S donde tenemos conectado el periférico con el que queremos llevar a cabo la operación de E/S.

El módulo, por su parte, ha de ser capaz de reconocer esta dirección que el procesador ha puesto en las líneas de direcciones del bus y leer el dato de las líneas de datos del bus para actualizar el registro correspondiente si se trata de una operación de escritura o poner la información de este registro en las líneas de datos del bus si estamos haciendo una operación de lectura.

3) La lógica del módulo se encarga de identificar la operación que tiene que hacer e inicia la transferencia con el periférico, de manera que el periférico deja de estar disponible y cambia el estado a ocupado hasta que la transferencia se finalice. Empieza la transferencia del dato con la sincronización.

4) El módulo obtiene el dato del periférico, lo guarda en el registro de datos para que el procesador lo pueda leer y actualiza el registro de estado o avisa al procesador para indicar que el dato está disponible, y consigue de esta manera la sincronización con el procesador. Si ha habido errores al obtener el dato del periférico, también lo indica en el registro de estado. Mientras el módulo de E/S obtiene el dato del periférico y valida la información, el procesador se ha de esperar. Se acaba la sincronización y empieza el intercambio del dato.

5) Una vez el dato está preparado en el módulo de E/S y el procesador está enterado de ello, el procesador obtiene el dato y se hace el intercambio del dato. El procesador debe acceder al puerto correspondiente al registro de estado si quiere validar que el dato recibido es correcto y al puerto correspondiente al registro de datos para leer el dato. Se acaba el intercambio del dato.

6) Como solo se tiene que transferir un dato, se indica que el estado del periférico es de disponible para hacer una nueva operación de E/S. Se acaba la operación de E/S.

1.2.3. Finalización de la operación de E/S

La finalización de la operación de E/S es un proceso parecido a la programación de la operación de E/S. También consiste en ejecutar un pequeño conjunto de instrucciones que actualizan los registros del módulo de E/S, pero ahora para indicar que se ha finalizado la transferencia y que el módulo de E/S queda disponible para atender a otra operación de E/S. También se puede consultar el estado del módulo de E/S para verificar que la transferencia se ha hecho correctamente.

Hay que remarcar que tanto la programación como la finalización de la operación de E/S son tareas que siempre efectúa el procesador, ya que es quien solicita la operación de E/S de información, y tienen realmente sentido cuando queremos enviar un bloque de datos y el módulo de E/S tiene autonomía para gestionar la transferencia. En cambio, no tienen tanto sentido para enviar datos unitarios. La complejidad de este proceso depende de la complejidad del periférico y de la técnica de E/S que utilizamos.

1.3. Gestión de múltiples dispositivos

Todos los computadores deben gestionar más de un periférico y estos pueden trabajar al mismo tiempo; por ejemplo, estamos imprimiendo un documento que tenemos guardado en el disco mientras escribimos un texto con el teclado que se muestra por pantalla. Por lo tanto, hemos de prever que nuestro

sistema de E/S pueda gestionar transferencias de E/S con dos o más periféricos simultáneamente. Eso quiere decir que de manera simultánea dos o más módulos de E/S deben estar preparados para hacer la transferencia de datos con el procesador, pero la transferencia no la podemos hacer al mismo tiempo. Por este motivo, hemos de disponer de un sistema que, primero, nos permita determinar cuáles son los módulos a los que tenemos que atender (**identificar los módulos de E/S** que están preparados para la operación de E/S) y, segundo, nos permita decidir a quién atendemos primero, teniendo en cuenta que si ya atendemos a otro periférico o hacemos otra tarea más prioritaria no la podemos interrumpir (**establecer una política de prioridades**).

Tanto la identificación del módulo de E/S que está preparado para transferir un dato como la manera de establecer una política de prioridades dependen de la técnica de E/S que utilicemos para gestionar las operaciones de E/S y lo hemos de analizar con detalle en cada caso.

Las políticas de prioridades básicas son:

- **Prioridad fija**: cuando el procesador está preparado para atender una petición, siempre empieza la consulta por el periférico que tiene más prioridad. El periférico con más prioridad siempre es atendido el primero. Si hay muchas peticiones de periféricos prioritarios, los menos prioritarios puede ser que hayan de esperar mucho tiempo para ser atendidos.

- **Prioridad rotativa**: cuando el procesador está preparado para atender una petición, consulta al periférico que hay a continuación según un número de orden preestablecido. A los periféricos se les asigna un orden pero todos tienen la misma prioridad.

Normalmente las políticas de prioridad se definen según las necesidades de cada dispositivo y las restricciones específicas que muchas veces impone el sistema de E/S utilizado en un computador. Y se pueden utilizar diferentes políticas de prioridades en un mismo sistema.

1.4. Técnicas de E/S

Hemos visto de manera genérica cuáles son los pasos para hacer una operación de E/S. El primer paso y el último, la programación y la finalización de la operación de E/S, siempre son responsabilidad del procesador y la complejidad que tienen depende en gran medida de las características del periférico y del sistema de interconexión de E/S que utilicemos, y no tanto, aunque también, de la técnica de E/S que utilicemos.

Por este motivo, en los apartados siguientes nos centraremos en la fase de **transferencia de un dato** (sincronización e intercambio del dato). Analizaremos qué dispositivos nos permiten liberar o descargar al procesador de las diferentes tareas que se tienen que hacer en este proceso y distinguiremos las técnicas básicas de E/S siguientes:

- E/S programada.
- E/S por interrupciones.
- E/S por DMA.
- Canales de E/S.

En la tabla que hay a continuación vemos cómo quedan repartidas las responsabilidades en una transferencia de E/S según la técnica de E/S que utilicemos.

	Programación	**Sincronización**	**Intercambio**	**Finalización**
E/S programada	Procesador	Procesador	Procesador	Procesador
E/S por interrupciones	Procesador	Módulo de E/S	Procesador	Procesador
E/S por DMA	Procesador	DMA	DMA bloquea el procesador	Procesador
Canales de E/S	Procesador/Canal de E/S	Canal de E/S	Canal de E/S bloquea el procesador	Procesador

2. E/S programada

Para hacer la operación de E/S entre el procesador y el módulo de E/S, el procesador ejecuta un programa que controla toda la operación de E/S (programación, transferencia de datos y finalización).

A continuación, analizamos con más detalle la transferencia de un dato:

1) Sincronización. Durante la sincronización, el procesador, como responsable de la transferencia, ejecuta un programa que mira constantemente el estado del periférico consultando el registro de estado del módulo de E/S. Este programa tiene un bucle que se ejecuta continuamente hasta que detecta el cambio de estado e indica que el periférico está preparado. Este método de sincronización se denomina **sincronización por encuesta** o **espera activa**.

Mientras se lleva a cabo la sincronización, el procesador está dedicado al cien por cien a esta tarea y, por lo tanto, no puede atender a otros procesos o aplicaciones. Si esta espera es muy larga, puede degradar el nivel de prestaciones de todo el sistema. Por lo tanto, es recomendable que las transferencias hechas utilizando esta técnica sean cortas y rápidas.

2) Intercambio del dato. Durante el intercambio del dato, si es una operación de lectura (entrada), el procesador lee el registro de datos del módulo de E/S para recoger el dato enviado por el periférico, y lo guarda en memoria; si es una operación de escritura (salida), el procesador toma de la memoria el dato que queremos enviar al periférico y lo escribe en el registro de datos del módulo de E/S.

Programa para atender a un periférico utilizando E/S programada

Queremos enviar un dato (escritura) a un periférico controlado por un módulo de E/S que tiene mapeados los registros de control, de estado y de datos en las direcciones 0120h, 0124h, 0128h, respectivamente.

SINCRO:	**IN**	R0,[0000 0124h]	0000 0124h: dirección del registro de estado del módulo de E/S. Leemos el estado y lo guardamos en R0.
	AND	R0, 00000004h	00000004h es una máscara para mirar el bit 3 del registro R0; en este caso es el bit que indica si el periférico está disponible para hacer la transferencia de datos.
	JE	SINCRO	Si el bit 3 está a cero, volvemos a SINCRO: y continuamos esperando hasta que el periférico esté preparado para recibir el dato y el módulo active este bit; si vale 1, continuamos y vamos a INTERCAMBIO: para hacer el intercambio del dato.
INTERCAMBIO:	**MOV**	R1, [Dato]	Tomamos el dato que queremos enviar al periférico y lo ponemos en el registro R1.
	OUT	[0000 0128h], R1	Copiamos el dato que tenemos en el registro R1 en el registro de datos del módulo de E/S (0000 0128h) para que este lo envíe al periférico.
	RET		Devolvemos el control al programa que ha llamado a esta rutina para hacer la operación de E/S.

Tanto este ejemplo como la descripción del proceso de sincronización y transferencia de datos se ha hecho siguiendo un modelo simplificado de operación de E/S. Hay que tener presente que para programar una rutina que lleve a cabo una operación de E/S con un periférico concreto hay que conocer las direcciones de los puertos de E/S asociadas a los registros del módulo de E/S que gestiona este periférico y el significado o la utilidad de cada uno de los bits. Dada la gran variedad de periféricos y su creciente complejidad, interpretar toda esta información es una tarea bastante complicada y generalmente son los propios fabricantes quienes desarrollan estas rutinas.

2.1. Gestión de múltiples dispositivos

No es habitual gestionar operaciones de E/S con múltiples dispositivos utilizando E/S programada. En caso de ser necesario, hay que tener presente que durante la sincronización se debe realizar la encuesta de todos los periféricos implicados estableciendo una política de prioridades implementada en forma de programa durante la sincronización. Este sistema de identificar qué periférico necesita ser atendido se denomina **encuesta** o ***polling***.

En el momento en el que se detecta que uno de los periféricos está preparado llamamos a una subrutina para hacer el intercambio del dato y volvemos al bucle de sincronización para seguir haciendo la encuesta según la política de prioridades implementada hasta que se hayan acabado todas las operaciones de E/S que están programadas.

3. E/S con interrupciones

En este apartado veremos el E/S por interrupciones. Esta técnica de E/S pretende evitar que el procesador tenga que estar parado o haciendo trabajo improductivo mientras espera a que el periférico esté preparado para hacer una nueva operación de E/S y pueda aprovechar este tiempo para ejecutar otros programas.

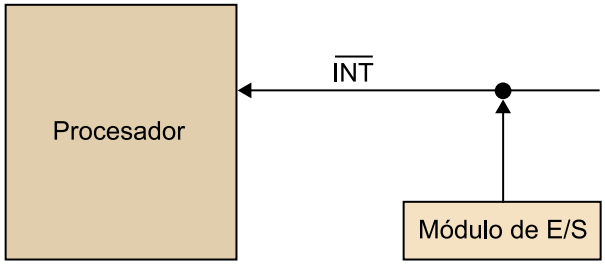
Utilizando la técnica de E/S por interrupciones se descarga al módulo de E/S de la responsabilidad de llevar a cabo la sincronización entre el periférico y el procesador.

Para utilizar esta técnica de E/S en un computador, es necesario considerar tanto aspectos del software como del hardware.

Como parte del hardware, es necesario que el computador disponga de una línea especial que tiene que formar parte del conjunto de líneas de control del bus del sistema y que denominamos **línea de petición de interrupción (INT)**. El módulo de E/S avisa al procesador mediante esta línea e indica que está preparado para hacer la transferencia. La señal INT la activa el módulo de E/S y la recibe el procesador. Es una señal activa a la baja. El procesador debe tener un punto de conexión de entrada por donde llegarán las interrupciones y el módulo de E/S debe tener un punto de conexión de salida por donde generará las interrupciones.

Señal activa a la baja

La señal INT es activa a la baja. Se considera activa si tenemos un 0 en la línea y no se considera activa si tenemos un 1.



Para hacer una operación de E/S utilizando esta técnica se siguen los mismos pasos que en la E/S programada: se programa la operación de E/S, se realiza la transferencia de datos y se finaliza la operación de E/S. La diferencia principal la tenemos durante la transferencia de datos, en la que en la fase de sincronización debemos hacer la gestión de las interrupciones y eso también afecta en cierta medida al intercambio de datos, como veremos más adelante.

Durante la fase de sincronización, una vez hecha la programación de la operación de E/S, el procesador ejecuta otro programa (según la política de gestión de procesos del sistema operativo) de manera que el procesador estará ocupado haciendo trabajo productivo hasta que el módulo de E/S esté preparado y active la señal de petición de interrupción (INT).

De entrada, el procesador no sabe en qué momento se producirá esta petición; por lo tanto, ha de comprobar periódicamente si el módulo de E/S pide la atención del procesador, sin que ello afecte a la dedicación que tiene. Esta comprobación el procesador la hace dentro del ciclo de ejecución de cada instrucción. Es una operación muy rápida que incluso se puede encabalgar con el comienzo de la lectura de la instrucción siguiente para que no afecte al rendimiento del procesador. Los procesadores que han de gestionar interrupciones deben tener en el ciclo de ejecución de instrucción una **fase de comprobación de interrupciones**.

Ciclo de ejecución de instrucción

Recordad que las cuatro fases principales para ejecutar una instrucción son:

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.

En el momento en el que el procesador reconoce que ha llegado una petición de interrupción, empieza un **ciclo de reconocimiento de interrupción** para detener la ejecución del programa actual y transferir el control a la **rutina de servicio de la interrupción (RSI)**, rutina que accede al módulo de E/S correspondiente para llevar a cabo la transferencia de datos y, una vez se acabe la ejecución de la RSI, continuar la ejecución del programa que habíamos detenido haciendo el **retorno de interrupción**.

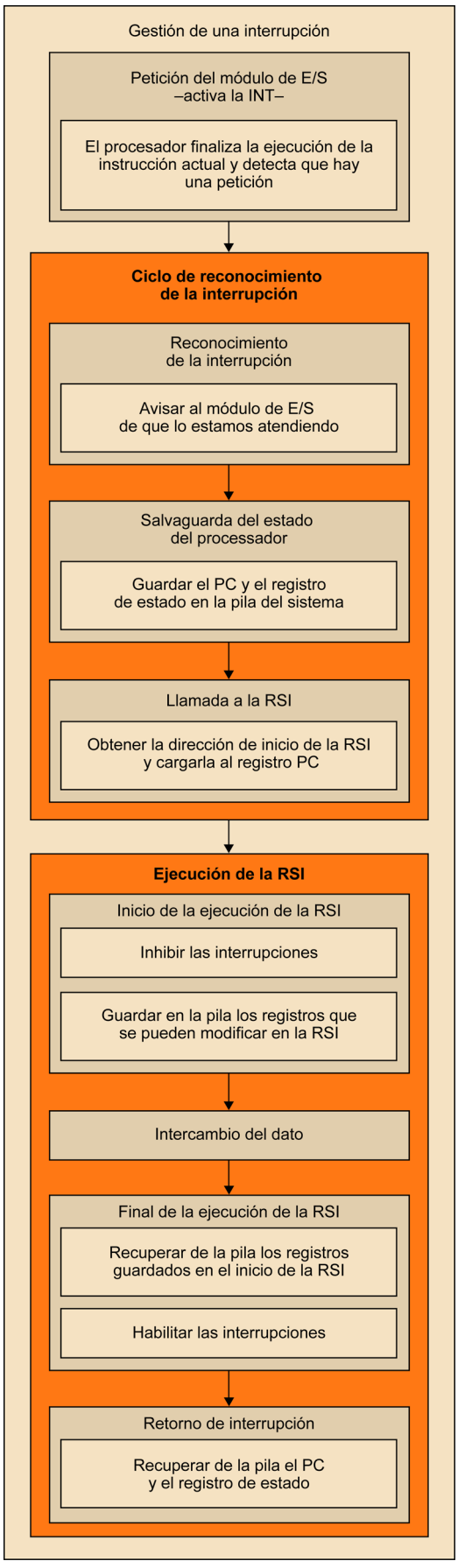
En este apartado nos centraremos en el análisis de interrupciones provocadas por elementos externos al procesador (interrupciones del hardware), pero también hay un tipo de interrupciones producidas por acontecimientos internos al procesador y denominadas **interrupciones de software**, **excepciones** o ***traps***.

Las interrupciones del software se gestionan de una manera muy parecida a las interrupciones externas al procesador, pero sin ser necesario dedicar una línea del bus del sistema para gestionarlas. Para tratarlas, no hay que esperar a la fase de comprobación de interrupción, sino que se pueden tratar en el mismo momento en el que se produce.

Las más habituales son estas: operaciones que la ALU no puede hacer, como dividir por 0, no se reconoce el código de operación de la instrucción que queremos ejecutar, acceso a recursos restringidos o áreas privilegiadas de memoria y también peticiones del propio programador con instrucciones específicas del juego de instrucciones para hacer peticiones al sistema operativo.

3.1. Gestión de una interrupción con un único módulo de E/S

Veamos cuáles son los pasos básicos para la gestión de una interrupción en un sistema con una única línea de interrupción y un único módulo de E/S. Más adelante analizaremos diferentes mejoras de este sistema para gestionar múltiples módulos de E/S y múltiples líneas de interrupción y cómo afecta eso a este proceso. Primero, sin embargo, hay que entender bien el caso más simple.


Gestión de una interrupción
Petición del módulo de E/S
–activa la INT–
El procesador finaliza la ejecución de la instrucción actual y detecta que hay una petición
Ciclo de reconocimiento de la interrupción
Reconocimiento de la interrupción
Avisar al módulo de E/S de que lo estamos atendiendo
Salvaguarda del estado del processador
Guardar el PC y el registro de estado en la pila del sistema
Llamada a la RSI
Obtener la dirección de inicio de la RSI y cargarla al registro PC
Ejecución de la RSI
Inicio de la ejecución de la RSI
Inhibir las interrupciones
Guardar en la pila los registros que se pueden modificar en la RSI
Intercambio del dato
Final de la ejecución de la RSI
Recuperar de la pila los registros guardados en el inicio de la RSI
Habilitar las interrupciones
Retorno de interrupción
Recuperar de la pila el PC y el registro de estado

Una vez el procesador ha solicitado una operación de E/S, ha programado la transferencia y ya ejecuta otro programa mientras espera a que el módulo de E/S esté preparado. En el momento en el que el módulo de E/S pide la atención del procesador (el módulo activa la INT), se produce una secuencia de acontecimientos que el procesador ha de gestionar para atender a esta petición del módulo de E/S, garantizando que después podrá devolver el control al programa cuya ejecución detenemos para atender la petición del módulo de E/S.

Los pasos son los siguientes:

1) Petición del módulo de E/S. El módulo de E/S está preparado para hacer una transferencia y activa la INT. Entonces, cuando el procesador acaba la ejecución de la instrucción actual, en la última fase del ciclo de ejecución de la instrucción, la fase de comprobación de interrupción, detecta que se ha hecho una petición.

2) Ciclo de reconocimiento de la interrupción. Esta es seguramente la parte más compleja de la gestión de interrupciones porque se producen muchos acontecimientos en poco tiempo y, como veremos más adelante, algunos de estos acontecimientos se pueden producir encabalgadamente en el tiempo y hay que estar atento a quién genera la acción, quién la recibe y qué respuesta da.

a) Reconocimiento de la interrupción. Si las interrupciones están habilitadas, el procesador acepta la petición (cuando hay más de un dispositivo se debe determinar si es suficientemente prioritario para ser atendido, situación que analizaremos más adelante, ya que tiene otras implicaciones en el proceso). Entonces, es necesario que el procesador avise al módulo de E/S para que sepa que lo está atendiendo y para que desactive la INT, y también es necesario que el procesador inhiba las interrupciones para evitar nuevas peticiones.

Si las interrupciones están inhibidas, el procesador no atiende la petición y continúa ejecutando el programa. El periférico se tiene que esperar a ser atendido y ha de dejar la INT activa hasta que el procesador las vuelva a habilitar.

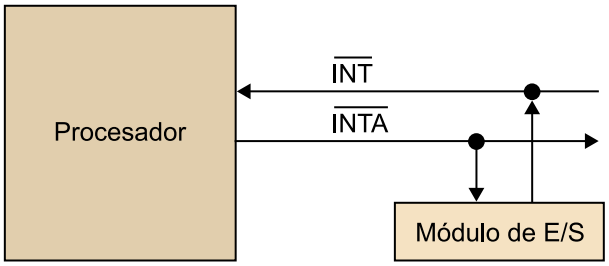
Para **inhibir las interrupciones**, hay básicamente dos maneras de hacerlo: con un hardware específico que las inhiba automáticamente hasta que se acabe la atención de la interrupción o dejar que sea responsabilidad de la propia RSI, que tiene que ejecutar una primera instrucción para inhibir las interrupciones y al acabar ejecutar otra instrucción para volver a habilitarlas.

Inhibir las interrupciones consiste en modificar uno o más bits del registro de la palabra de estado del procesador (algunos procesadores pueden tener registros específicos para la gestión de interrupciones). Estos bits son los que se

consultan en la fase de comprobación de interrupciones durante la ejecución de una instrucción para saber si las interrupciones están habilitadas y se tiene que aceptar o no una petición.

Si el procesador no inhibe las interrupciones, al ejecutar la primera instrucción de la RSI, en la fase de comprobación de interrupción volverá a aceptar la petición y podría entrar en un bucle infinito ejecutando esta primera instrucción indefinidamente, ya que volverá a empezar un ciclo de reconocimiento de la misma interrupción.

De manera análoga a como se hace en la inhibición de las interrupciones, se puede avisar al módulo de E/S de dos maneras: la más habitual es utilizando un hardware específico que incluye una **línea de reconocimiento de interrupción** (INTA o INTACK, del inglés *interrupt acknowlege*) que activa el procesador, igual que la INT es una señal activa a la baja, o dejar que sea responsabilidad de la propia RSI, que ha de ejecutar una o varias instrucciones para acceder a los registros del módulo de E/S y de esta manera indicarle que ya lo atiende.



b) Salvaguarda del estado del procesador. Ahora el procesador ha reconocido que hay una petición de un módulo de E/S, la ha aceptado y tiene que ejecutar la rutina de servicio de interrupción (RSI) para atenderlo, pero recordemos que el procesador está ejecutando otro programa y hay que reanudar la ejecución una vez ha acabado la ejecución de la RSI. Por este motivo se tiene que hacer la salvaguarda del estado del procesador, que consiste en almacenar la información necesaria para garantizar que podremos reanudar la ejecución del programa en las mismas condiciones. Esta información se almacena en la memoria del computador, generalmente en la pila de sistema.

El estado del procesador lo determina el valor de todos sus registros, pero para reanudar la ejecución del programa solo hay que guardar los registros que pueda modificar la RSI. Una RSI puede modificar cualquier registro del procesador, pero realmente suelen ser rutinas relativamente simples que solo modifican algunos de estos registros. Si solo se guardan los registros que se modifican dentro de la RSI, la salvaguarda y la restauración posterior serán mucho más eficientes en tiempo y espacio, ya que reducimos los accesos a la memoria y el espacio necesario para guardar al estado.

La manera más habitual de hacer la salvaguarda es almacenar de manera automática la información mínima necesaria para llamar a la RSI y almacenar el resto de la información dentro de la misma rutina, y es responsabilidad del programador de la RSI determinar qué registros hay que guardar.

La información mínima que se debe guardar es el **registro contador de programa** (PC) y el **registro de estado**. El registro PC tiene la dirección de la instrucción siguiente que queremos ejecutar. El registro de estado contiene información importante de control y de estado del procesador que puede ser modificada por cualquier instrucción.

Nota

Recordad que el valor del PC se actualiza en la fase de lectura de la instrucción del ciclo de ejecución.

Con respecto al resto de la información, como el procesador no puede saber qué registros modificará la RSI, deja que sea responsabilidad de la propia RSI.

c) Llamada a la RSI. Para empezar a ejecutar la rutina de servicio de interrupción (RSI), primero de todo, hay que saber la dirección de memoria donde tenemos que iniciar la ejecución de la RSI. En caso de gestionar un único dispositivo, esta dirección puede ser fija (cuando tenemos más de un dispositivo, en la fase de reconocimiento de la interrupción tenemos que haber identificado qué dispositivo pide atención y utilizaremos esta información para saber la dirección de la RSI correspondiente al dispositivo al que hemos de atender).

Una vez sabemos la dirección de inicio de la RSI, se carga en el registro PC y se inicia el ciclo de ejecución de la primera instrucción de la RSI. La RSI se ejecuta en modalidad supervisor.

3) Ejecución de la rutina de servicio de interrupción

a) Inicio de la ejecución de la RSI. Si no se ha hecho la inhibición de las interrupciones en la fase de reconocimiento de la interrupción, hay que hacerlo al inicio de la RSI. Esta inhibición se efectúa utilizando una instrucción específica del juego de instrucciones.

Después se deben guardar los registros que puedan ser modificados en la pila (el programador de la RSI sabe qué registros se cambian y, por lo tanto, puede determinar qué registros se deben guardar) y antes de acabar la ejecución hay que restaurarlos con los valores originales. Cuando se programa una RSI, se debe ser muy cuidadoso con la salvaguardia de los registros porque puede provocar que otros programas (los que detenemos para ejecutar la RSI) funcionen mal.

b) Intercambio del dato. La parte principal del código de la RSI tiene que ser específico para el periférico al que atendemos. De manera general se debe acceder al registro de datos para hacer el intercambio del dato. Si es necesario, se puede acceder primero a los registros de estado y de control para saber si la operación de E/S se ha hecho correctamente o se han producido errores.

Para programar el código de una RSI, hay que conocer con detalle el funcionamiento del periférico y qué función tiene cada uno de los bits de los registros del módulo de E/S que se utiliza para controlarlo. Generalmente, esta información que hay que conocer es bastante compleja cuando se debe analizar con detalle y puede variar mucho de un dispositivo a otro, incluso, en evoluciones de un mismo tipo de dispositivo.

c) Finalización de la ejecución de la RSI. Hay que recuperar el valor de los registros del procesador que se han guardado al principio de la RSI a la pila (en orden inverso al modo como se ha guardado, ya que lo tenemos guardado en la pila).

Si se han inhibido las interrupciones al principio de la RSI, hay que habilitarlas utilizando una instrucción específica del juego de instrucciones.

d) Retorno de interrupción. Antes de acabar la ejecución de la RSI, hemos de restaurar el estado del procesador para devolver el control al programa que se estaba ejecutando. Para ello, hemos de recuperar de la pila del sistema la información que se ha guardado de manera automática (los registros de estado y el PC). Esta información la recuperamos al ejecutar la última instrucción de la RSI, que es la instrucción de retorno de interrupción, habitualmente denominada *IRET*.

Una vez restaurado el estado del procesador, se reanuda la ejecución del programa que hemos detenido en las mismas condiciones iniciando un nuevo ciclo de ejecución de la instrucción. Y la petición del módulo de E/S ha quedado atendida.

Ejemplo

En este ejemplo se observa cómo se atiende una interrupción. Es especialmente importante fijarse en cómo queda el registro PC después de ejecutar cada fase y la evolución de la pila durante este proceso.

Las abreviaturas que se utilizan son:

- SP: *stack pointer* (puntero en la cima de la pila).
- PC: *program counter* (dirección de la instrucción que estamos ejecutando).
- RSI: rutina de servicio de interrupción.
- INT: señal que activa el módulo de E/S para pedir atención al procesador.


1. Petición del módulo de E/S. Al acabar la ejecución de la instrucción actual el procesador mira si la INT está activa y pasa a atender la petición.
Rutina para atender una INT
RSI: Inhibir INT
PUSH Regs.
...
Intercambio del dato
...
POP Regs.
Habilitar INT
IRET
...
Pila del sistema
SP → Cima de la pila
...
Programa en ejecución
...
PC → Instrucción actual
Instrucción siguiente
...
arriba
← INT
2. Reconocimiento de la interrupción, salvaguarda del estado (guarda PC y registro de estado) y llamada a la RSI (actualiza PC con la dirección de inicio de la RSI).
Rutina para atender una INT
PC → RSI Inhibir INT
PUSH Regs.
...
Intercambio del dato
...
POP Regs.
Habilitar INT
IRET
...
Pila del sistema
SP → PC (Inst. siguiente)
Reg. STATUS
Cima de la pila
...
Programa en ejecución
...
Instrucción actual
Instrucción siguiente
...
3. Se inicia la ejecución de la RSI (inhibir las interrupciones y guardar los registros que pueden ser modificados por la RSI).
Rutina para atender una INT
RSI: Inhibir INT
PUSH Regs.
PC → ...
Intercambio del dato
...
POP Regs.
Habilitar INT
IRET
...
Pila del sistema
SP → Regs. de l'RSI
PC (Inst. siguiente)
Reg. STATUS
Cima de la pila
...
Programa en ejecución
...
Instrucción actual
Instrucción siguiente
...
4. Intercambio del dato.
Rutina para atender una INT
RSI: Inhibir INT
PUSH Regs.
...
Intercambio del dato
...
PC → POP Regs.
Habilitar INT
IRET
...
Pila del sistema
SP → Reg. de la INT
PC (Inst. siguiente)
Reg. STATUS
Cima de la pila
...
Programa en ejecución
...
Instrucción actual
Instrucción siguiente
...
5. Final de la ejecución del RSI. Recuperar de la pila los registros guardados y habilitar las interrupciones.
Rutina para atender una INT
RSI: Inhibir INT
PUSH Regs.
...
Intercambio del dato
...
POP Regs.
Habilitar INT
PC → IRET
...
Pila del sistema
SP → PC (Inst. siguiente)
Reg. STATUS
Cima de la pila
...
Programa en ejecución
...
Instrucción actual
Instrucción siguiente
...
6. Retorno de interrupción. Recuperar el registro de estado y el PC de la pila, y dejar el procesador en el mismo estado y el PC apuntando a la instrucción siguiente.
Rutina para atender una INT
RSI: Inhibir INT
PUSH Regs.
...
Intercambio del dato
...
POP Regs.
Habilitar INT
IRET
...
Pila del sistema
SP → Cima de la pila
...
Programa en ejecución
...
Instrucción actual
PC → Instrucción siguiente
...

3.2. Gestión de interrupciones con múltiples módulos de E/S

Hasta ahora hemos estudiado el caso más simple, en el que solo tenemos un módulo que funciona por interrupciones, pero la realidad es que un sistema de E/S por interrupciones tiene que gestionar múltiples dispositivos con capacidad de generar interrupciones. Veamos cómo afecta esto a la gestión de interrupciones y cómo lo hemos de implementar.

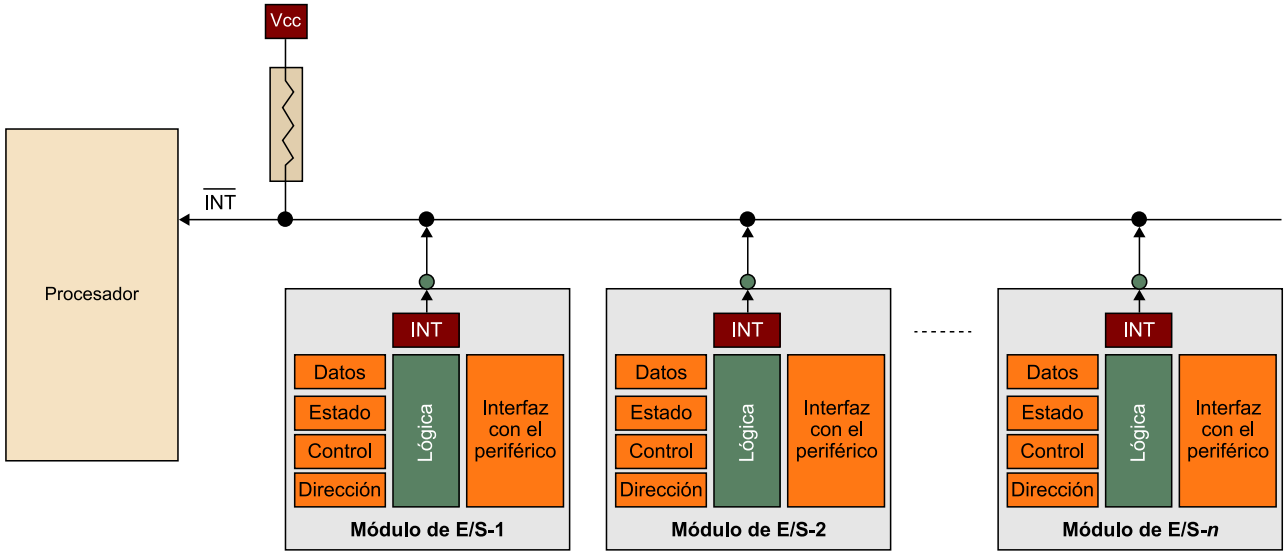
Cuando el procesador recibe una petición, sabe que algún módulo de E/S pide atención pero no sabe cuántos piden atención, ni cuáles son. Para identificar al que pide atención y decidir cuál se tiene que atender, las cuestiones principales que hay que resolver son:

- Identificar cuál es el periférico que pide atención.
- Determinar qué periférico hemos de atender primero en caso de que dos o más periféricos soliciten atención al mismo tiempo.
- Gestionar, si el sistema permite la nidificación, las prioridades para determinar si el periférico que solicita atención es más prioritario que otro al que ya atendemos.
- Obtener la dirección de la RSI correspondiente al periférico al que hemos de atender.

Estas cuestiones afectan principalmente al ciclo de reconocimiento de la interrupción y hacen este proceso bastante más complejo. El objetivo principal de los sistemas que nos permiten gestionar más de un módulo de E/S mediante interrupciones es conseguir que este proceso sea tan eficiente como sea posible, añadiendo mejoras al hardware que da soporte al sistema de atención de interrupciones.

3.3. Sistema con una única línea de petición de interrupción

Este es el caso más simple, en el que todos los módulos de E/S se conectan en colector abierto, utilizando un PULL-UP, a una única línea de petición de interrupción que llega al procesador.



La resistencia conectada a Vcc (tensión alta) sirve para implementar un PULL-UP, que da la funcionalidad de OR-CABLEADA considerando que la línea es activa a la baja. Si tenemos un 0 en la línea, se considera activa y si tenemos un 1, no se considera activa. Esto permite que los módulos de E/S estén conectados en colector abierto a la línea, de manera que si ningún módulo activa la salida, el *pull-up* mantiene la línea a 1 (tensión alta), condición de reposo de todos los dispositivos, y si algún módulo activa la salida, poniendo un 0 (tensión baja), la línea cambia de estado y el procesador puede reconocer que hay una petición pendiente de ser atendida.

La gestión de una interrupción en este sistema es análoga a la gestión de una interrupción con un único módulo de E/S. Ahora bien, para identificar qué periférico pide atención y gestionar las prioridades si más de un módulo de E/S ha pedido atención al mismo tiempo, el procesador ejecuta una única RSI que tiene una dirección de inicio fija. La RSI, mediante código y accediendo a los registros de estado de cada módulo de E/S, determina el módulo de E/S al que debe atender, de manera muy parecida a la encuesta (o *polling*) que se hace en E/S programada cuando tenemos más de un módulo de E/S. El código que utilizamos para atender al periférico es una subrutina de la RSI.

Las ventajas principales que ofrece este sistema son que:

- Como hacemos la encuesta de todos los módulos por programa, es muy flexible determinar las prioridades porque podemos implementar diferentes políticas de prioridades simplemente modificando el código de la RSI.

- No hay que hacer cambios en el hardware.

Las desventajas principales son que:

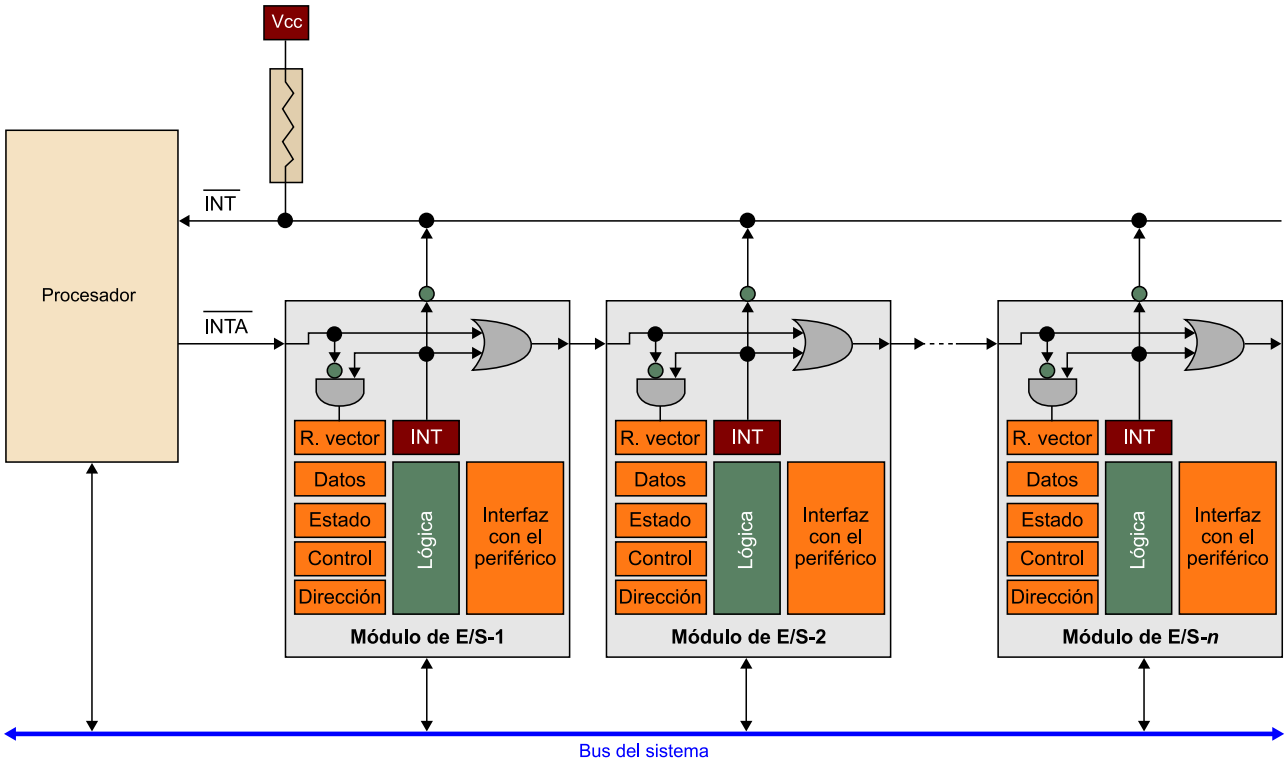
- La ejecución del código para hacer la encuesta de los módulos y la gestión de prioridades es muy costosa en tiempo, ya que estamos ejecutando un pequeño programa que accede a todos los módulos de E/S conectados a la línea.

- Como hay una sola línea de petición de interrupción, se tienen que inhibir las interrupciones y, por lo tanto, mientras atendemos una petición no podemos atender otras aunque sean más prioritarias.

3.4. Sistema con una línea de petición de interrupción y una línea de reconocimiento con encadenamiento

Este sistema con encadenamiento también se denomina ***daisy-chain***. Los módulos de E/S se conectan al procesador con una línea de petición de interrupción (INT) en colector abierto y una línea de reconocimiento de interrupción (INTA) que genera el procesador para indicar al módulo de E/S que se atiende la petición que ha hecho. Esta señal INTA se propaga mediante los módulos.

Es un sistema de conexión que permite, por una parte, reducir el tiempo necesario para gestionar las prioridades si más de un módulo de E/S ha pedido atención al mismo tiempo y, por otra parte, identificar qué periférico pide atención, sin tener que ejecutar un programa de encuesta que accede a todos los módulos de E/S.



Para implementar este sistema, hay que hacer algunos cambios en el hardware: el procesador necesita disponer de un punto de conexión por donde generar la señal de reconocimiento de interrupción (INTA); los módulos de E/S deben disponer de un circuito lógico muy simple para propagar la señal INTA al siguiente módulo de E/S dejando todos los módulos de E/S conectados uno tras otro y disponer de un nuevo registro, que denominamos **registro vector**, donde almacenamos un valor llamado **vector de interrupción** que enviamos al procesador mediante el bus del sistema. Este valor que almacena cada registro vector tiene que ser único, ya que el procesador lo utiliza para identificar al periférico que pide atención.

La gestión de una interrupción en este sistema es análoga a la gestión de una interrupción con un único módulo de E/S, y varía el ciclo de reconocimiento de la interrupción para gestionar las prioridades si más de un módulo de E/S ha pedido atención al mismo tiempo e identificar qué periférico demanda atención.

3.4.1. Interrupciones vectorizadas

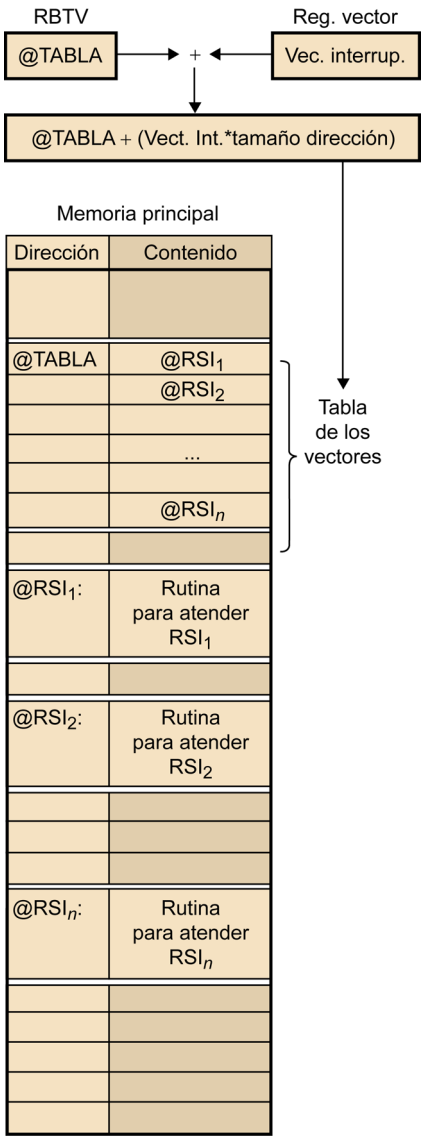
Se denomina **vectorización** a la técnica en la que el procesador identifica al módulo de E/S mediante la información que envía el mismo módulo de E/S. Cuando para tratar las interrupciones utilizamos esta técnica para identificar a quien hace la petición decimos que tenemos las **interrupciones vectorizadas**.

Una vez un módulo de E/S o más de uno han hecho la petición al procesador activando la INT y el procesador acepta la petición, activa la INTA y empieza el proceso para saber a cuál se debe atender.

Tratamiento de la señal INTA

La señal INTA tiene un funcionamiento parecido a la señal READ para leer un dato de memoria. Cuando el módulo recibe la señal INTA y tiene la INT activa, deposita el vector de interrupción en el bus del sistema para que el procesador lo pueda leer.

Esta señal INTA proveniente del procesador, activa a la baja, llega al primer módulo. Si este módulo ha solicitado atención, bloquea la propagación de la señal INTA y deposita en el bus del sistema el vector de interrupción, valor almacenado en el registro vector del módulo de E/S; si no ha solicitado atención (no tiene activa la INT), deja pasar la señal INTA al módulo siguiente, que hace lo mismo hasta que llega al último módulo conectado a esta línea.



Como vemos, los módulos de E/S quedan encadenados por esta señal INTA. El primer módulo de la cadena es el primero en recibir la señal y, por lo tanto, el más prioritario, y la propaga hasta el último, que es el menos prioritario. Si queremos cambiar las prioridades, nos veremos obligados a cambiar el orden de los módulos dentro de la cadena físicamente.

El procesador lee el vector de interrupción del bus del sistema. Con este valor identifica el periférico al que tiene que atender y lo utiliza para obtener la dirección de inicio de la RSI (esta dirección es la que hemos de cargar en el registro PC para hacer la llamada a la RSI).

La manera más habitual de obtener la dirección de inicio de la RSI es utilizar una **tabla de vectores de interrupción** almacenada en la memoria principal, donde tenemos guardadas las direcciones de inicio de las RSI asociadas a cada petición de interrupción a la que tenemos que atender. El vector de interrupción lo utilizamos para acceder a la tabla. Esta tabla puede estar almacenada en una dirección fija de memoria o en una dirección programable almacenada en el **registro base de la tabla de vectores (RBTV)**. Si tenemos la tabla

de vectores en una dirección fija, hemos de determinar la posición dentro de la tabla de vectores, a partir del vector de interrupción. Si tenemos la tabla de vectores en una dirección programable, hemos de determinar la posición dentro de la tabla de vectores sumando el RBTV a un índice obtenido a partir del vector de interrupción.

Desplazamiento dentro de la tabla de vectores

Una manera de obtener el desplazamiento dentro de la tabla de vectores a partir del vector de interrupción, como el vector de interrupción identifica el dispositivo (y no la dirección dentro de la tabla o la dirección de la rutina misma), es multiplicar el vector de interrupción por las posiciones de memoria que ocupa una dirección dentro de la tabla de vectores. Si las direcciones de memoria almacenadas en la tabla de vectores ocupan 4 bytes (32 bits) y cada posición de memoria es de un byte, el vector de interrupción se tiene que multiplicar por 4. Hacer esta operación en binario es muy simple: solo hay que desplazar el vector de interrupción dos posiciones a la izquierda (añadir dos ceros a la parte derecha) para obtener el desplazamiento dentro de la tabla. En sistemas más complejos puede ser necesario hacer otro tipo de operaciones.

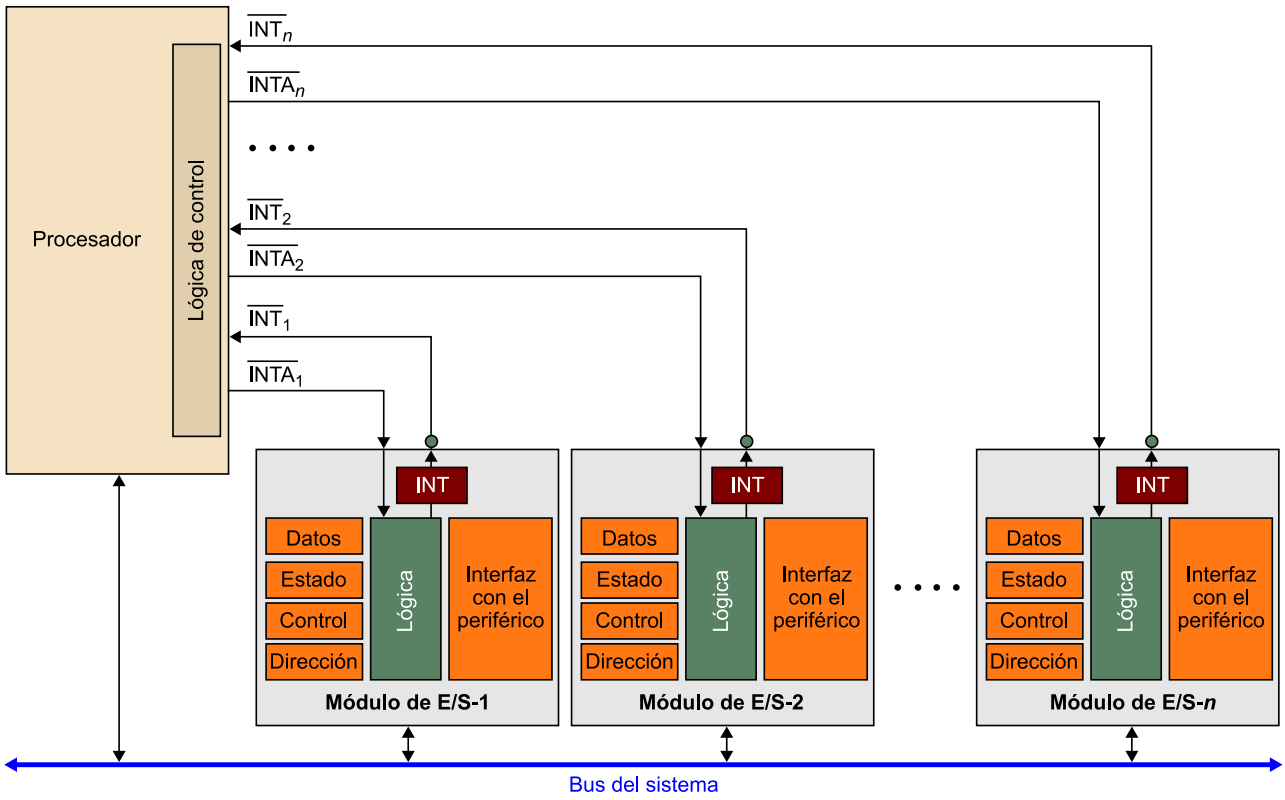
Las principales ventajas de este sistema con encadenamiento son que la identificación se efectúa en un tiempo parecido a la duración de un ciclo de bus, que es mucho menos costoso en tiempo que hacer la encuesta a todos los módulos de E/S y que también resuelve la gestión de prioridad cuando hay peticiones simultáneas.

Las desventajas principales son que el sistema de prioridades es fijo e inalterable –cambiar el orden de prioridades implica modificar el hardware– y que, como tiene una única línea de petición de interrupciones, no permite que otros periféricos más prioritarios sean atendidos si ya atendemos a una petición de interrupción sea cual sea la prioridad que tiene.

3.5. Sistema con líneas independientes de petición de interrupciones y de reconocimiento

En este sistema tenemos una línea de petición de interrupción y una de reconocimiento de interrupción para cada módulo de E/S. De esta manera conseguimos que la identificación del periférico sea inmediata y permitimos la **nidificación de interrupciones**, es decir, una interrupción puede interrumpir la ejecución de una RSI que atiende una petición menos prioritaria que se ha producido con anterioridad.

Para implementar este sistema es necesario que el procesador disponga de los puntos de conexión necesarios para conectar las señales de petición y reconocimiento de cada uno de los módulos de E/S que puedan generar una interrupción hacia el procesador y un circuito específico para hacer el control de estas líneas de petición y reconocimiento de interrupciones. Los módulos de E/S solo deben poder generar una petición de interrupción y recibir la señal de reconocimiento de interrupción generada por el procesador.



La gestión de una interrupción en este sistema es análoga a la gestión de una interrupción con un único módulo de E/S, y varía la fase de reconocimiento de la interrupción para gestionar las prioridades e identificar qué periférico pide atención.

El proceso para hacer el ciclo de reconocimiento de la interrupción es el siguiente: una vez uno o más módulos de E/S han hecho la petición al procesador activando la INT, el procesador decide si acepta la petición, inhibe de manera selectiva las interrupciones, activa la INTA correspondiente y obtiene la dirección de la RSI para atender aquella petición.

En un sistema con una única línea de petición de interrupción hemos visto que cuando aceptamos una petición hay que inhibir las interrupciones y, por lo tanto, no podemos aceptar nuevas peticiones hasta que se ha acabado la atención de esta petición. En un sistema con múltiples líneas de petición y reconocimiento de interrupción también hemos de inhibir las peticiones de interrupción de la línea de la que hemos aceptado la petición, pero hay que tener un mecanismo para decidir si se deben permitir peticiones de las otras líneas para enmascarar las interrupciones de manera selectiva según una política de prioridades.

Hay 2 formas básicas para enmascarar de manera selectiva las interrupciones:

- Enmascaramiento individual.
- Enmascaramiento por nivel.

En el **enmascaramiento individual** disponemos de un **registro de interrupciones**, que tiene 2 bits por cada línea de petición de interrupción: un bit de máscara que nos permite habilitar o inhibir las peticiones de una línea y opcionalmente un bit de interrupción que nos indica si se ha aceptado la petición, es decir, si ha habido una petición del módulo de E/S y están habilitadas las peticiones de interrupciones para aquella línea.

Este sistema de enmascaramiento también dispone de 1 bit para el enmascaramiento general de interrupciones, un bit de máscara que nos permite habilitar o inhibir todas las peticiones de interrupción enmascarables. Para aceptar una petición de una línea, tienen que estar activos los bits de la línea y también el bit general de interrupciones.

Si el procesador en la fase de comprobación de interrupción detecta que hay alguna petición de interrupción, hace un ciclo de reconocimiento de interrupción y se determina qué petición se tiene que atender accediendo a los bits de máscara del registro de interrupciones. Si se acepta una petición, se deben inhibir las interrupciones de aquella línea y de las líneas menos prioritarias modificando los bits de máscara correspondientes. En computadores simples o microcontroladores, el registro de interrupción es visible al programador y hemos de identificar quién pide atención y decidir a quién atendemos primero mediante un programa; en sistemas más complejos generalmente se dispone de un hardware específico.

Esta forma de enmascaramiento se suele utilizar en sistemas con pocas líneas de petición de interrupción porque, a pesar de ser muy flexible, si no se dispone de un hardware específico la gestión de prioridades se debe realizar por programa y eso la hace más lenta. Es una situación parecida a la encuesta que hacemos cuando tenemos una única línea de petición de interrupción, pero en lugar de tener que acceder a los registros de los módulos de E/S mediante el bus del sistema, accedemos a un registro del procesador que es mucho más fácil y rápido.

En el **enmascaramiento por nivel**, a cada línea de petición de interrupción se asigna un **nivel de interrupción**, denominado también **nivel de ejecución**. Si el número de niveles es inferior al número de líneas de petición de interrupción, se agrupan varias líneas en un mismo nivel.

Este nivel de ejecución se utiliza para gestionar las prioridades, de manera que si ya atendemos una petición de interrupción, solo aceptamos una nueva petición y detenemos la atención si esta petición tiene un nivel de ejecución más prioritario.

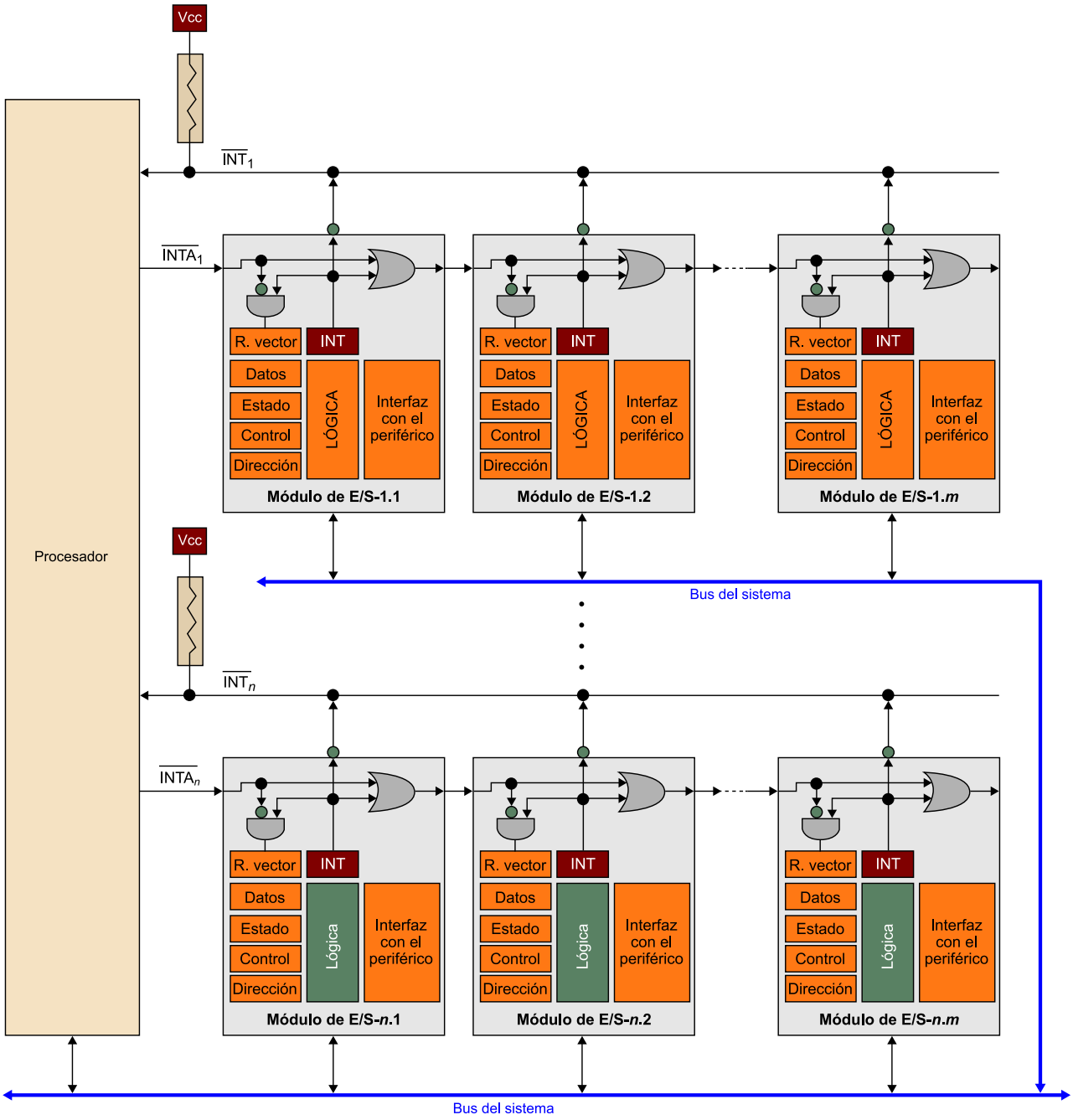
El nivel de ejecución se puede obtener fácilmente conectando las líneas de petición de interrupción a un codificador de prioridad y el código generado se almacena en un registro del procesador, generalmente en el registro de estado del procesador. Para decidir si aceptamos una determinada petición, con un circuito comparador, hemos de comparar el valor obtenido del codificador de prioridad con el nivel de ejecución que tenemos almacenado. Si es más prioritario el valor obtenido del codificador, aceptamos la petición y utilizamos este valor como un vector de interrupción para obtener la dirección de la RSI para atender a esta petición. El nuevo nivel de ejecución se actualiza de manera automática una vez hecha la salvaguarda del estado del procesador (se ha almacenado el PC y el registro de estado en la pila del sistema) y empieza la ejecución de la RSI. De esta manera, cuando acabamos la ejecución de la RSI que atiende esta petición más prioritaria y restauramos el estado del procesador para reanudar la ejecución del programa parado, recuperamos también el nivel de ejecución que tiene, ya que queda guardado con el resto de la información del registro de estado.

Esta forma de enmascaramiento es la más habitual en este tipo de sistemas.

Por otra parte, para garantizar un funcionamiento correcto del computador, el procesador debe disponer de líneas de petición de interrupción que no se puedan enmascarar. Solo están inhibidas en momentos concretos durante el ciclo de reconocimiento de las interrupciones y durante el retorno de interrupción para garantizar la estabilidad del sistema.

Las principales ventajas de este sistema con líneas independientes de petición de interrupciones y de reconocimiento son que permite la nidificación, que la identificación del periférico es muy rápida y que la gestión de prioridades resulta muy flexible.

La desventaja principal es que no se puede aumentar el número de módulos de E/S a los que podemos atender, ya que implica un rediseño del procesador o, como se puede ver en el esquema siguiente, utilizar sistemas híbridos estableciendo por ejemplo un sistema de encadenamiento (*daisy-chain*) en cada una de las líneas de petición y reconocimiento de interrupción, aunque esta opción hace la gestión de las interrupciones más compleja y lenta.



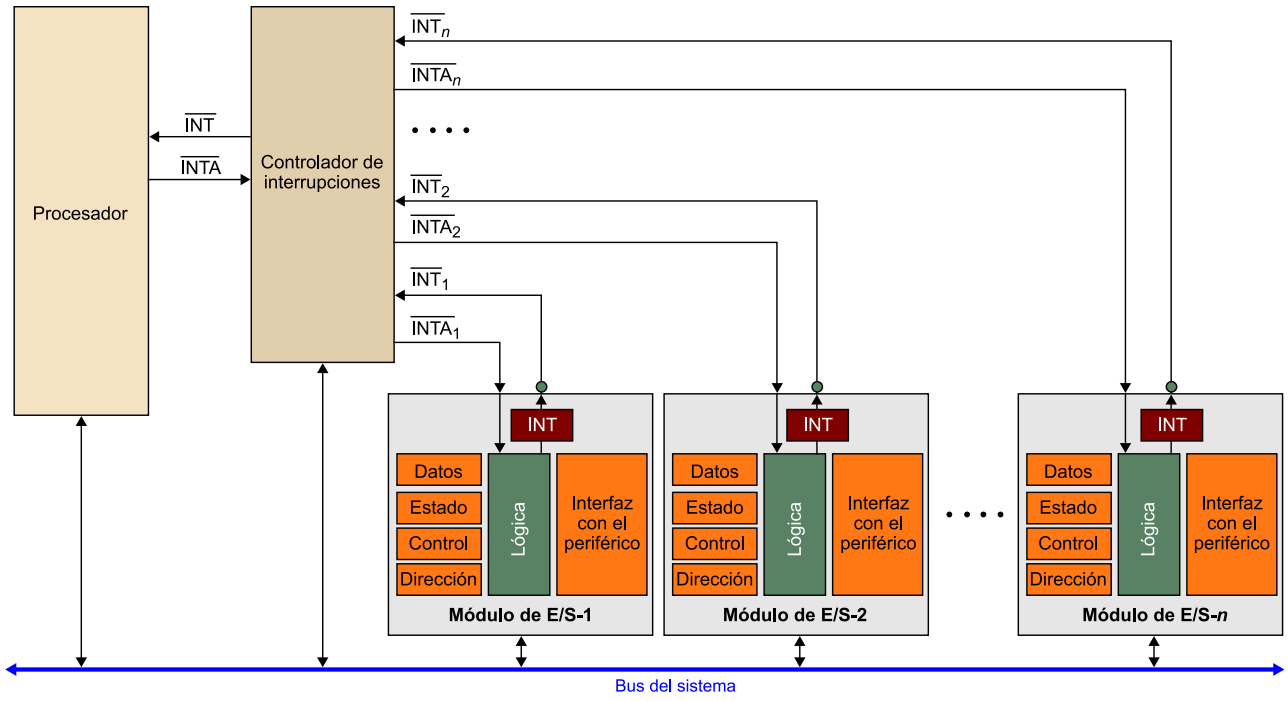
3.6. Sistema con controladores de interrupciones

En un sistema con controladores de interrupciones se añade un nuevo elemento especializado para gestionar las peticiones de interrupción que denominamos **controlador de interrupciones**. Las líneas de petición de interrupción y de reconocimiento de cada módulo de E/S están conectadas al controlador de interrupciones. De esta manera el procesador solo debe tener una línea de petición de interrupción y una de reconocimiento conectadas al controlador de interrupción para gestionar las peticiones provenientes de los módulos de E/S.

Las funciones del controlador de interrupción son las siguientes:

- Definir una política de prioridades para los módulos de E/S conectados al controlador.
- Identificar qué módulo de E/S pide atención e informar el procesador.

La gestión de una interrupción en este sistema es análoga a la gestión de una interrupción con un único módulo de E/S, y varía la fase de reconocimiento de la interrupción para gestionar las prioridades e identificar qué periférico pide atención.



El proceso para hacer el reconocimiento de la interrupción es el siguiente: para identificar el periférico que pide atención y obtener la dirección de la RSI que tiene que atender la petición, utilizamos un sistema de vectorización muy parecido al descrito en el *daisy-chain*, pero ahora no es necesario que el módulo de E/S tenga un registro vector. El controlador de interrupciones dispone de un conjunto de registros vector donde se guardan los vectores de interrupción asociados a cada línea.

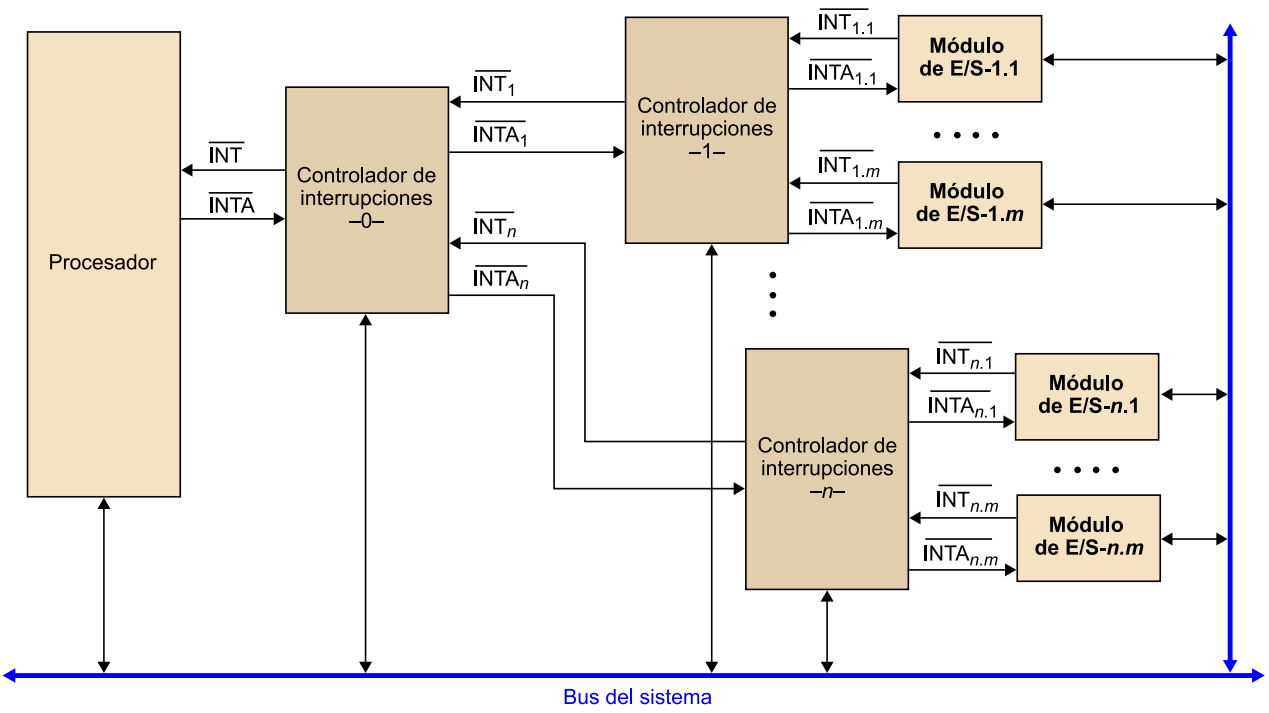
Cuando llegue una petición de interrupción de un módulo de E/S muy prioritario, el controlador de interrupciones hace la petición al procesador activando la señal INT y el procesador contesta activando la señal INTA. Cuando el controlador de interrupciones recibe la señal INTA coloca el vector de interrupción, correspondiente al módulo de E/S que ha hecho la petición, en el bus de datos y el procesador lee el vector. De esta manera, el procesador puede obtener la dirección de la RSI y empezar la ejecución de la rutina para

hacer la transferencia de datos, comunicándose directamente con el módulo de E/S. La transferencia de datos no se gestiona mediante el controlador de interrupciones.

Las principales ventajas de este sistema con un controlador de interrupción son que la identificación del periférico es bastante rápida y la gestión de prioridades flexible, y también que el procesador solo necesita disponer de una línea INT y una INTA para gestionar múltiples módulos de E/S.

La desventaja principal es que a causa de la gestión de prioridades que hace el controlador no se puede hacer la salvaguarda del nivel de ejecución de manera automática, sino que se debe realizar ejecutando un pequeño programa que acceda al controlador de interrupciones y hace el proceso de salvaguarda del estado más lento.

Si tenemos un sistema que utiliza controladores de interrupción y queremos aumentar el número de módulos de E/S, hemos de conectar el controlador de interrupción en cascada como se muestra a continuación, pero es necesario que los controladores estén diseñados específicamente para conectarlos de esta manera.



4. E/S con acceso directo a memoria

Las técnicas de E/S que hemos visto hasta ahora requieren una dedicación importante del procesador (ejecutar un fragmento de código) para hacer simples transferencias de datos. Si queremos transferir bloques de datos, estas técnicas todavía ponen más en evidencia la ineficiencia que tienen. En E/S programada implica que el procesador no pueda hacer nada más y en E/S por interrupciones descargamos el procesador de la sincronización a costa de hacer las rutinas de atención más largas para garantizar el estado del procesador, lo que limita la velocidad de transferencia.

En este apartado describiremos una técnica mucho más eficiente para transferir bloques de datos, el **acceso directo a memoria** (**DMA**). En esta técnica el procesador programa la transferencia de un bloque de datos entre el periférico y la memoria encargando a un nuevo elemento conectado al bus del sistema hacer toda la transferencia. Una vez acabada, este nuevo elemento avisa el procesador. De esta manera, el procesador puede dedicar todo el tiempo que dura la transferencia del bloque a otras tareas. Este nuevo elemento que gestiona toda la transferencia de datos entre el periférico y la memoria principal lo denominamos **módulo** o **controlador de DMA** o también en versiones más evolucionadas **canal** o **procesador de E/S**.

Utilizando la técnica de E/S por DMA se descarga al procesador de la responsabilidad de llevar a cabo la sincronización y el intercambio de datos entre el periférico y la memoria.

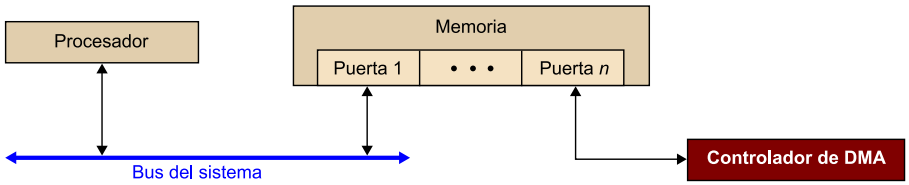
Por otra parte, nos aparece una nueva problemática en el computador, ya que hay dos dispositivos –el procesador y el controlador de DMA– que tienen que acceder de manera concurrente a la memoria y hay que establecer un mecanismo para resolver este conflicto.

4.1. Acceso concurrente a memoria

Básicamente hay dos maneras de resolver el acceso concurrente a memoria:

1) Conexiones independientes, utilizando memorias multipuerta. Hay una conexión independiente para cada dispositivo que tiene que acceder a la memoria. Eso permite al controlador de DMA acceder a la memoria sin que el procesador tenga que intervenir. Aunque la afectación al procesador es muy

baja, esta solución no se suele utilizar porque hace aumentar el coste de la memoria. También aumenta el tiempo de acceso, sobre todo cuando hay más de una puerta que quiere acceder al mismo bloque de información.



2) Conexión compartida, utilizando robo de ciclo. En este caso, la memoria solo necesita una única puerta y tanto el procesador como el controlador de DMA comparten el bus del sistema para acceder a la memoria. En la mayoría de los sistemas, el procesador es quien controla el bus; por lo tanto, hay que establecer un mecanismo para que el procesador pueda ceder el bus al controlador de DMA y este controlador pueda hacer el intercambio de los datos con la memoria. Este mecanismo se denomina *robo de ciclo* y es el que se utiliza más frecuentemente para gestionar el acceso concurrente a la memoria.

Para controlar el acceso al bus, son necesarias dos señales, BUSREQ y BUSACK (parecidas a las señales INT e INTA utilizadas en interrupciones). Con la señal BUSREQ el controlador de DMA solicita el control del bus y el procesador cede el bus activando la señal BUSACK.

> La diferencia más importante entre el acceso directo a memoria y la gestión de interrupciones es que el procesador puede inhibir las interrupciones total o parcialmente, mientras que la cesión del bus no la puede inhibir y está obligado a ceder siempre el control del bus cuando el controlador de DMA lo solicita.

Las principales ventajas de este sistema son que la atención es muy rápida porque no se puede inhibir la cesión del bus y que no es necesario guardar al estado del procesador, que solo queda parado durante un tiempo muy breve mientras el controlador de DMA tiene el control del bus.

La cesión del bus no es inmediata. El procesador solo puede ceder el bus al acabar cada una de las fases del ciclo de ejecución de las instrucciones. Una vez el controlador de DMA libera el bus, de manera que se acaba el robo de ciclo, el procesador continúa la ejecución de la siguiente fase de la instrucción en curso. Por lo tanto, el robo de ciclo se puede producir en diferentes puntos dentro del ciclo de ejecución de una instrucción.

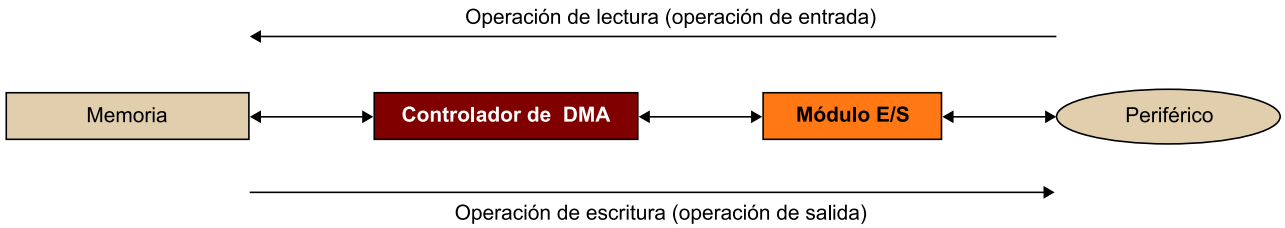
La principal desventaja de este sistema es que se alarga el tiempo de ejecución de las instrucciones a causa de los posibles robos de ciclo que se pueden producir.

4.2. Operación de E/S con acceso directo a memoria

El proceso para hacer una transferencia de E/S utilizando esta técnica es el siguiente:

1) Programación de la operación de E/S: el procesador envía la información necesaria al controlador de DMA para que este controlador pueda gestionar toda la transferencia de datos. Una vez acaba la programación de la operación de E/S, el procesador puede ejecutar otros programas mientras se realiza la transferencia.

2) Transferencia del bloque de datos: las dos operaciones básicas que hace el controlador de DMA son la lectura de un bloque de datos de un periférico y la escritura de un bloque de datos en un periférico, aunque también puede hacer otras operaciones. Todos los datos de la transferencia pasan por el controlador de DMA.



3) Finalización de la operación de E/S: cuando se ha acabado la transferencia del bloque, el controlador de DMA envía una petición de interrupción al procesador para informar de que se ha acabado la transferencia de datos.

Nota

Para utilizar esta técnica de E/S en un computador, es necesario considerar tanto aspectos del software como del hardware.

4.3. Controladores de DMA

En un sistema de este tipo las implicaciones con respecto al hardware y al sistema de conexión de este hardware son diversas.

El procesador debe disponer de un sistema para gestionar interrupciones, como hemos explicado en el apartado anterior. El controlador de DMA avisa al procesador de que se ha acabado la transferencia del bloque de datos mediante una petición de interrupción.

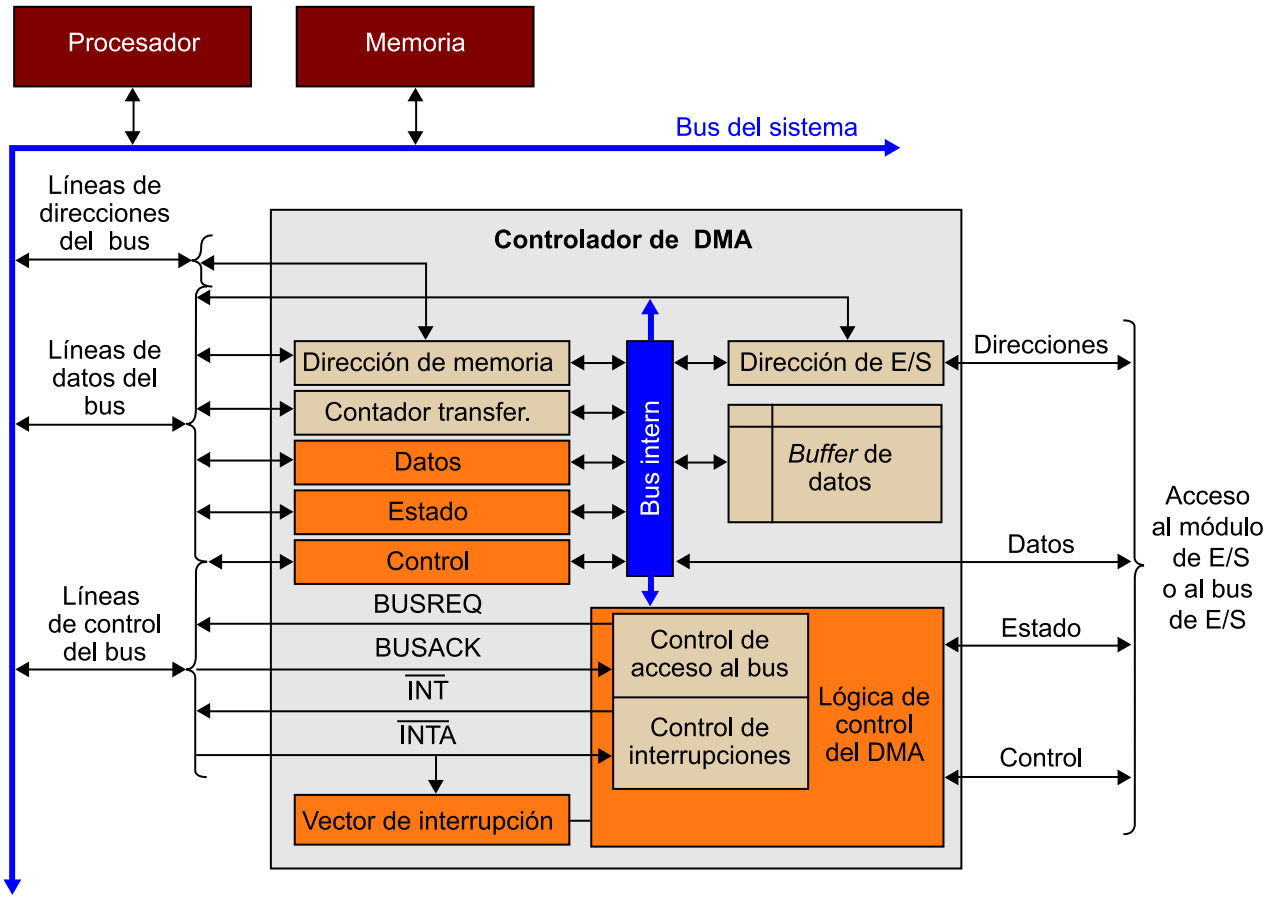
Los elementos básicos que debe tener el controlador de DMA para gestionar una transferencia de datos entre el periférico y la memoria son los siguientes:

- Un banco de registros para gestionar las operaciones de E/S.
- La lógica de control necesaria para gestionar las transferencias entre la memoria y el módulo de E/S.
- La lógica necesaria para gestionar la interrupción de la finalización de la operación de E/S.

- Señales de control para acceder al bus del sistema y para la gestión de interrupciones (BUSREQ, BUSACK, INT, INTA, etc.).

El banco de registros del controlador de DMA está formado por los registros siguientes:

- **Registro de control**: se utiliza generalmente para dar las órdenes al controlador.
- **Registro de estado**: da información del estado de la transferencia de datos.
- **Registro de datos**: almacena los datos que se quieren intercambiar.
- **Registro de direcciones de memoria**: indica la dirección de memoria donde se leerán o se escribirán los datos que se tienen que transferir en cada momento.
- **Registro contador**: indica inicialmente el número de transferencias que se deben hacer, se irá actualizando durante la transferencia hasta que valga cero e indicará en cada momento el número de transferencias que quedan por hacer.
- **Registro de direcciones de E/S**: indica la posición dentro del periférico donde se leerán o se escribirán los datos que se tienen que transferir en cada momento.

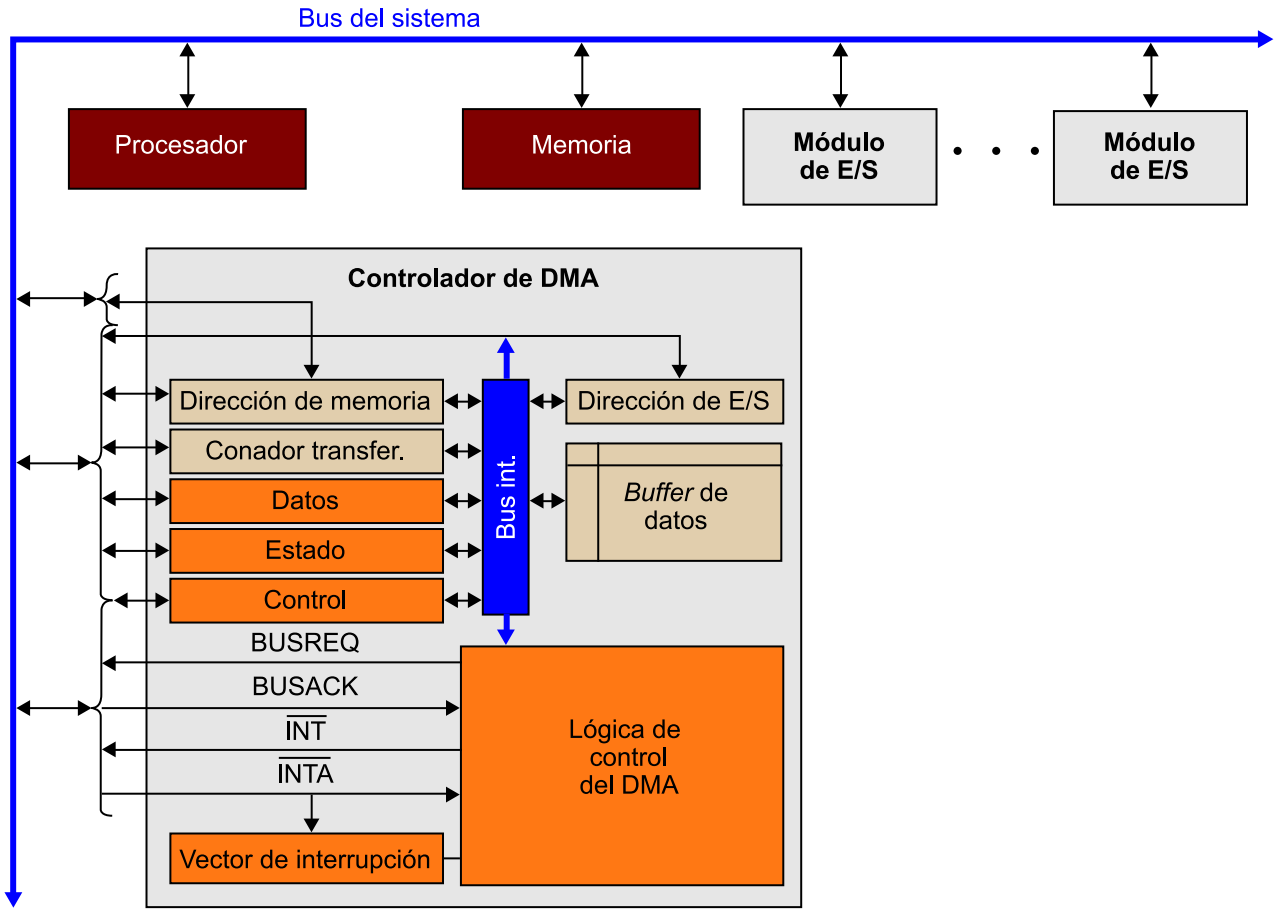


4.3.1. Formas de conexión de los controladores de DMA

A continuación trataremos las configuraciones más habituales del controlador de DMA.

La configuración siguiente es la más simple pero también la más ineficiente porque se tienen que hacer dos accesos al bus: uno para acceder a memoria y otro para acceder al módulo de E/S.

Conexión del controlador de DMA y de los módulos de E/S con un único bus



En caso de tener más de un controlador de DMA conectado al bus, la gestión de prioridades para decidir a quién se cede el bus se hace de manera muy parecida a la gestión de prioridades explicada en la E/S por interrupciones:

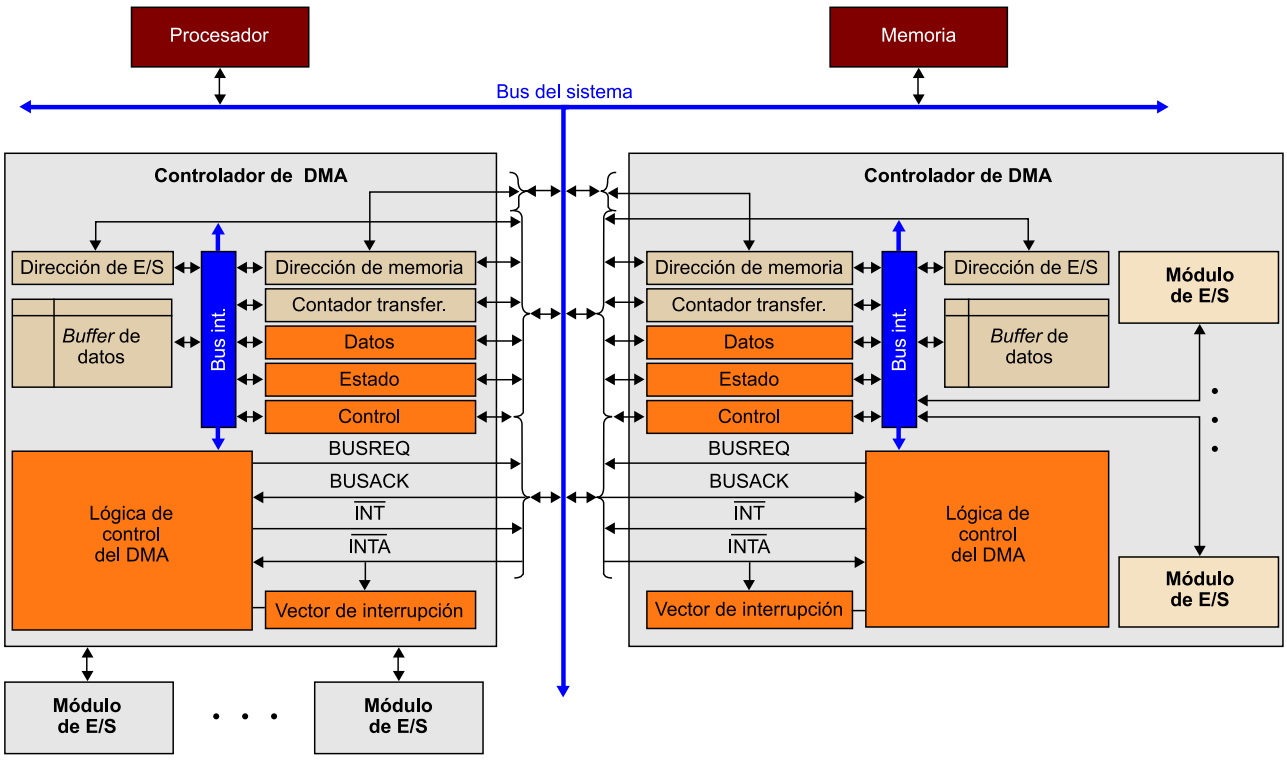
- Sistema con encadenamiento (*daisy-chain*), en el que se hace el encadenamiento de la señal BUSACK.

- Utilizar un controlador de DMA que gestione múltiples transferencias, de manera parecida al controlador de interrupciones, pero ahora la transferencia de datos con la memoria se efectúa mediante el controlador de DMA y no directamente entre el módulo de E/S y el procesador, como sucede cuando utilizamos controladores de interrupciones. En este caso, podemos

encontrar tanto conexiones independientes entre el controlador y cada módulo de E/S, como todos los módulos conectados a un bus específico de E/S.

Una mejora de la configuración anterior consiste en conectar los módulos de E/S directamente al controlador de DMA, de manera que necesitamos solo un acceso al bus del sistema. Esta configuración también permite conectar controladores de DMA que integran la lógica del módulo de E/S.

Conexión punto a punto entre el controlador de DMA y los módulos de E/S

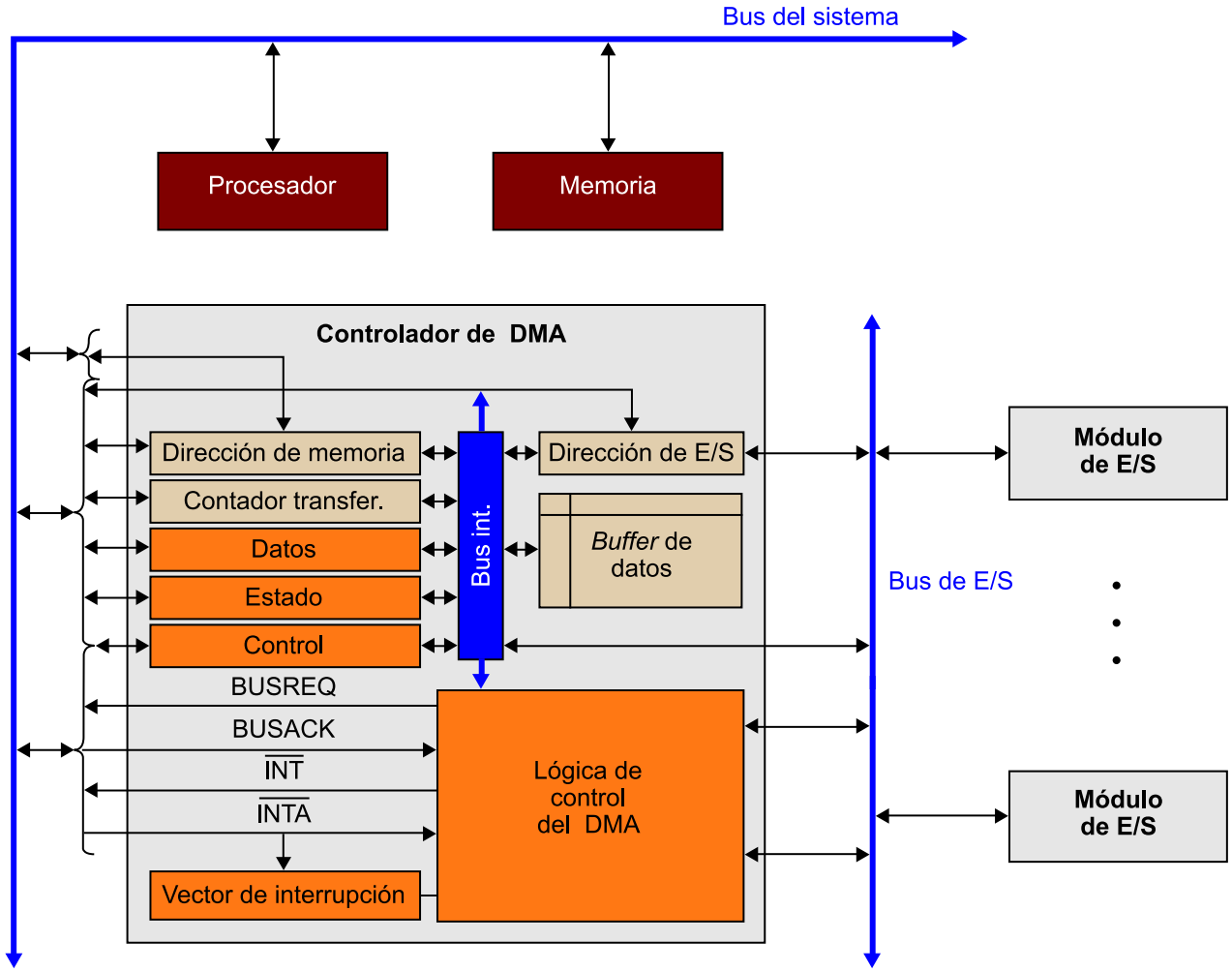


Transferencias simultáneas

Si se tienen que hacer las operaciones de E/S con varios periféricos simultáneamente, hemos de replicar parte del conjunto de registros (direcciones, contador, bits de control, etc.) tantas veces como transferencias de datos simultáneos queramos gestionar.

Otra configuración posible consiste en conectar los módulos de E/S al controlador de DMA mediante un bus de E/S. El flujo de información entre el controlador de DMA y los módulos de E/S no interfiere con los accesos al bus del sistema.

Conexión del controlador de DMA y de los módulos de E/S mediante un bus de E/S



4.3.2. Operación de E/S mediante un controlador de DMA

Para hacer una transferencia de E/S utilizando esta técnica se siguen los pasos siguientes:

1) Programación de la operación de E/S. El procesador tiene que enviar la información necesaria al controlador de DMA para que este controlador pueda hacer la transferencia de manera autónoma. Esta información se debe escribir en el banco de registros del controlador de DMA.

La información imprescindible que se ha de escribir es la siguiente:

a) Operación que hay que hacer. Las operaciones básicas son de lectura o escritura, pero también se pueden hacer otros tipos de operaciones de control del periférico. Esta información se escribe en el registro de control mediante las líneas de control o las líneas de datos del bus de sistema.

b) Dirección de memoria. Dirección inicial de memoria donde se leerán o se escribirán los datos que se han de transferir considerando que el bloque de datos que moveremos está en direcciones contiguas de memoria. Esta información se escribe en el registro de direcciones de memoria.

c) Tamaño del bloque de datos. Número de transferencias que se tiene que hacer mediante el bus del sistema para transferir todo el bloque de datos. Esta información se escribe en el registro contador.

d) Dirección del periférico. Posición inicial dentro del periférico donde se leerán o se escribirán los datos que se han de transferir. Esta información se escribe en el registro de direcciones de E/S. Esta información depende del periférico y en cierta medida del tipo de información que se tiene que transmitir. Si es un dispositivo de almacenamiento, hemos de saber dónde está guardado para recuperar después los datos.

2) Transferencia del bloque de datos. Dado que el procesador ha delegado al controlador de DMA la operación de E/S, el controlador de DMA debe realizar la transferencia de datos de todo el bloque. La transferencia se hace dato a dato accediendo directamente a memoria sin la intervención del procesador.

Los pasos para una operación de lectura y para una operación de escritura son los siguientes:

Después de transferir cada dato, el controlador de DMA decrementa el registro contador y actualiza el registro de direcciones de memoria con la dirección donde tenemos que leer o almacenar el dato siguiente. Cuando el registro contador llega a cero, el controlador de DMA genera una petición de interrupción para avisar al procesador de que se ha acabado la transferencia del bloque.

3) Finalización de la operación de E/S. Una vez el controlador de DMA envía la petición de interrupción al procesador para informar de que se ha acabado la transferencia del bloque de datos, el procesador ejecuta la RSI correspondiente para acabar la operación de E/S.

Cabe señalar que el controlador de DMA se debe dotar del hardware necesario para que pueda generar la petición de interrupción según el sistema de interrupciones que utiliza el procesador; por ejemplo, si utilizamos un sistema con encadenamiento (*daisy-chain*), hemos de poder generar una INT, recibir una INTA y la lógica para propagarla y un registro vector para identificarse.

4.4. Controlador de DMA en modo ráfaga

Una manera de optimizar las operaciones de E/S por DMA consiste en reducir el número de cesiones y recuperaciones del bus. Para hacerlo, en lugar de solicitar y liberar el bus para cada dato que se tiene que transferir, se solicita y se libera el bus para transferir un conjunto de datos de manera consecutiva. Esta modalidad de transferencia se llama **modo ráfaga**.

Para hacer la transferencia de este conjunto de datos, que denominamos **ráfaga**, el controlador de DMA tiene que disponer de una memoria intermedia (*buffer*), de modo que la transferencia de datos entre la memoria y el controlador de DMA se pueda hacer a la velocidad que permita la memoria y no quedando limitada a la velocidad del periférico.

Este modo de funcionamiento no afecta a la programación ni a la finalización de la operación de E/S descrita anteriormente, pero sí que modifica la transferencia de datos.

El funcionamiento de la transferencia del bloque de datos es el siguiente: en el caso de la lectura, cada vez que el módulo de E/S tiene un dato disponible, el controlador de DMA lo almacena en la memoria intermedia y decrementa el registro contador. Cuando la memoria intermedia está llena o el contador ha llegado a cero, solicita el bus. Una vez el procesador le cede el bus, escribe en memoria todo el conjunto de datos almacenados en la memoria intermedia, hace tantos accesos a memoria como datos tenemos y actualiza el registro de direcciones de memoria en cada acceso. Al acabar la transferencia del conjunto de datos, libera el bus.

En el caso de la escritura, el controlador de DMA solicita el bus y, cuando el procesador cede el bus, el controlador de DMA lee un dato de la memoria, lo almacena en la memoria intermedia, decrementa el registro contador y actualiza el registro de direcciones de memoria. Cuando la memoria intermedia está llena o el contador ha llegado a cero, libera el bus. A continuación transfiere todo este conjunto de datos almacenados en la memoria intermedia al módulo de E/S y espera para cada dato a que el módulo de E/S esté preparado.

Una vez acabada una ráfaga, si el registro contador no ha llegado a cero, empieza la transferencia de una nueva ráfaga.

4.5. Canales de E/S

Los canales de E/S son una mejora de los controladores de DMA. Pueden ejecutar instrucciones que leen directamente de memoria. Eso permite gestionar con más autonomía las operaciones de E/S y de esta manera se pueden controlar múltiples operaciones de E/S con dispositivos con una mínima intervención del procesador.

Estos canales todavía se pueden hacer más complejos añadiendo una memoria local propia que los convierte en procesadores específicos de E/S.

La programación de la operación de E/S por parte del procesador se realiza escribiendo en memoria los datos y las instrucciones que necesita el canal de E/S para gestionar toda la operación de E/S. La información que se especifica incluye el dispositivo al que tenemos que acceder, la operación que se debe realizar indicando el nivel de prioridad, la dirección del bloque de datos donde tenemos que leer o escribir los datos que se han de transferir, el tamaño del bloque de datos que se tienen que transferir, cómo se tiene que hacer el tratamiento de errores y cómo se ha de informar al procesador del final de la operación de E/S.

Cuando se acaba la operación de E/S, el canal de E/S informa al procesador de que se ha acabado la transferencia y de posibles errores mediante la memoria. También se puede indicar el final de la transferencia mediante interrupciones.

Las dos configuraciones básicas de canales de E/S son las siguientes:

- **Canal selector**: está diseñado para periféricos de alta velocidad de transferencia y solo permite una operación de transferencia simultánea.

- **Canal multiplexor**: está diseñado para periféricos más lentos de transferencia y puede combinar la transferencia de bloques de datos de diferentes dispositivos.

Las ventajas principales de los canales de E/S respecto a los controladores de E/S son las siguientes:

- Permiten controlar operaciones de E/S simultáneas.

- Se pueden programar múltiples operaciones de E/S sobre diferentes dispositivos o secuencias de operaciones sobre el mismo dispositivo, mientras el canal de E/S efectúa otras operaciones de E/S.

5. Comparación de las técnicas de E/S

Para analizar las prestaciones de un sistema de E/S, compararemos las diferentes técnicas de E/S estudiadas y así profundizaremos un poco más en el estudio del funcionamiento de estas técnicas, y también definiremos los parámetros básicos de una transferencia de E/S para determinar la dedicación del procesador en cada una de las técnicas de E/S.

Para realizar el análisis de las prestaciones, consideramos que hacemos la transferencia de un bloque de datos entre un periférico y el computador.

$$N_{datos} = m_{bloque} / m_{dato}$$

Donde

- m_{dato}: tamaño de un dato expresado en bytes. Entendemos *dato* como la unidad básica de transferencia.
- m_{bloque}: tamaño del bloque de datos que queremos transferir entre el computador y el periférico expresado en bytes.
- N_{datos}: número de datos que forman el bloque que queremos transferir. Nos indica también el número de transferencias que se tienen que hacer.

Periférico

Primero hay que analizar cómo condiciona la transferencia de E/S el periférico. Para hacerlo definimos los parámetros siguientes:

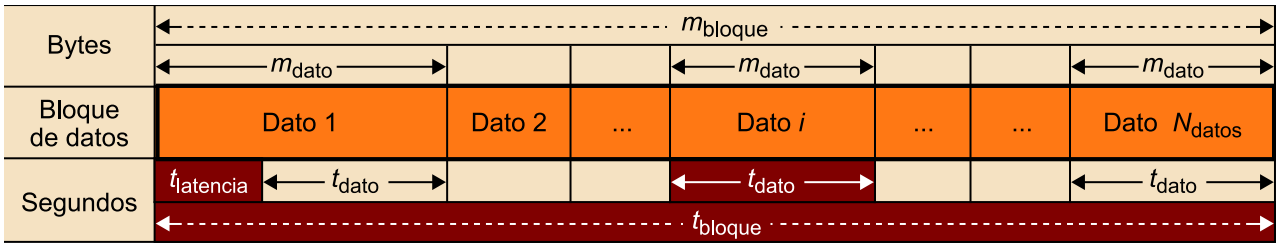
- v_{transf}: velocidad de transferencia media del periférico expresada en bytes por segundo.
- t_{dato}: tiempo de transferencia de un dato (entre el periférico y el módulo de E/S) expresado en segundos.

$$t_{dato} = m_{dato} / v_{transf}$$

- $t_{latencia}$: tiempo de latencia, tiempo que necesita el periférico para iniciar la primera transferencia expresado en segundos. En algunos periféricos este tiempo puede ser significativo.

- t_{bloque}: tiempo de transferencia de los datos de todo el bloque (entre el periférico y el módulo de E/S) expresado en segundos.

$$t_{\text{bloque}} = t_{\text{latencia}} + (N_{\text{datos}} \cdot t_{\text{dato}})$$



Definimos a continuación los parámetros básicos que determinan de alguna manera la transferencia de datos en las diferentes técnicas de E/S.

Procesador

- t_{inst}: tiempo medio de ejecución de una instrucción expresado en segundos.

Bus del sistema y memoria

- $t_{\text{cesión}}$: tiempo necesario para que el procesador haga la cesión del bus expresado en segundos.
- t_{recup}: tiempo necesario para que el procesador haga la recuperación del bus expresado en segundos.
- t_{mem}: tiempo medio para hacer una lectura o una escritura en la memoria principal mediante el bus del sistema expresado en segundos.

Transferencia de E/S

Tal como hemos definido anteriormente en una transferencia de E/S, la programación y la finalización de la transferencia son pasos comunes a todas las técnicas de E/S y son responsabilidad del procesador. Hay que tener presente que en E/S por DMA la programación de la transferencia es estrictamente necesaria y tiene un peso más importante que en el resto de las técnicas.

- t_{prog}: tiempo necesario para hacer la programación de la transferencia de E/S expresado en segundos.

- t_{final}: tiempo necesario para hacer la finalización de la transferencia de E/S expresado en segundos.

Estos tiempos se pueden calcular a partir del número de instrucciones que hay que ejecutar para hacer la programación (N_{prog}) o la finalización de la transferencia (N_{final}), multiplicando el número de instrucciones necesarias por el tiempo medio de ejecución de una instrucción (t_{inst}).

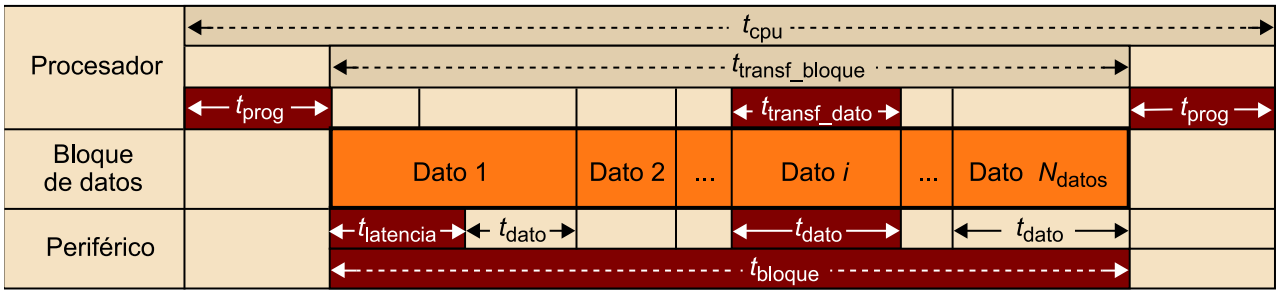
- t_{transf_dato}: tiempo, expresado en segundos, que el procesador está ocupado durante la transferencia de un dato con el módulo de E/S o parado mientras el controlador de DMA hace la transferencia de un dato.

- t_{transf_bloque}: tiempo, expresado en segundos, que el procesador está ocupado durante la transferencia de datos de todo el bloque con el módulo de E/S o parado mientras el DMA hace la transferencia de datos de todo el bloque.

$$t_{transf\_bloque} = N_{datos} \cdot t_{transf\_dato}$$

- t_{cpu}: tiempo que el procesador está ocupado o parado durante la transferencia de E/S expresado en segundos.

$$t_{cpu} = t_{prog} + t_{transf\_bloque} + t_{final}$$



Hasta ahora hemos analizado los parámetros básicos que condicionan los pasos principales en una transferencia de E/S. A continuación, veremos cómo afectan las diferentes técnicas de E/S al tiempo que el procesador está ocupado o parado (t_{transf_bloque} y t_{cpu}).

E/S programada

Cuando utilizamos esta técnica el procesador está dedicado el cien por cien del tiempo a esta tarea; por lo tanto, no tiene tiempo para ejecutar otras instrucciones:

$$t_{transf\_dato} = t_{dato}$$
$$t_{transf\_bloque} = t_{bloque}$$
$$t_{cpu} = t_{prog} + t_{transf\_bloque} + t_{final}$$

Cabe señalar que estos tiempos no dependen del número de instrucciones del bucle de sincronización, ni del código para hacer la transferencia del dato, sino que solo dependen de la velocidad de transferencia del periférico.

E/S por interrupciones

Cuando utilizamos E/S por interrupciones, la sincronización es responsabilidad del módulo de E/S y el procesador solo es responsable de la operación de transferencia de los datos. La transferencia se realiza cuando llega la petición de interrupción (INT) del módulo de E/S y el procesador ejecuta la RSI para atenderla.

Para determinar el tiempo que dedica el procesador a atender la transferencia de un dato definimos los parámetros siguientes:

- t_{rec_int}: tiempo que pasa desde que el procesador reconoce que hay una petición de interrupción hasta que se puede iniciar la ejecución de la RSI expresado en segundos. Este tiempo incluye el reconocimiento de la interrupción y la salvaguarda del estado del procesador.

- t_{rsi}: tiempo de ejecución de la RSI expresado en segundos. Este tiempo incluye inhibir las interrupciones si es necesario, guardar en la pila los registros que se pueden modificar, acceder al módulo de E/S para hacer la transferencia del dato, restaurar de la pila los registros, volver a habilitar las interrupciones y hacer el retorno de interrupción.

- N_{rsi}: número de instrucciones de la RSI.

Tanto el tiempo de reconocimiento de la interrupción y salvaguarda del estado (t_{rec_int}) como el tiempo de ejecución de la RSI (t_{rsi}) dependen de diferentes factores que hay que tener en cuenta si se tienen que calcular.

De manera general decimos que:

$$t_{rsi} = N_{rsi} \cdot t_{inst}$$

El tiempo total que el procesador está ocupado durante toda la transferencia de E/S se puede calcular de la manera siguiente:

$$t_{transf\_dato} = t_{rec\_int} + t_{rsi}t$$
$$_{transf\_bloque} = N_{datos} \cdot t_{transf\_dato}$$
$$t_{cpu} = t_{prog} + t_{transf\_bloque} + t_{final}$$

E/S por DMA

En esta técnica de E/S se descarga el procesador de la responsabilidad de llevar a cabo la transferencia de los datos. Esta responsabilidad se deja al controlador de DMA. Este controlador hace la transferencia de los datos, pero para hacerlo tiene que utilizar el bus del sistema con la técnica del robo de ciclo.

Veamos cuánto tiempo está ocupado o parado el procesador mientras dura la transferencia de los datos. El procesador está ocupado durante la programación y la finalización de la transferencia y está parado mientras el controlador de DMA utiliza el bus del sistema para hacer la transferencia de los datos. Para hacerlo utilizamos los parámetros siguientes:

$$t_{transf\_dato} = t_{cesión} + t_{mem} + t_{recup}$$
$$t_{transf\_bloque} = N_{datos} \cdot t_{transf\_dato}$$
$$t_{cpu} = t_{prog} + t_{transf\_bloque} + t_{final}$$

E/S por DMA en modo ráfaga

En este caso, se considera que la unidad de transferencia del DMA está formada por un conjunto de datos. Cada vez que se solicita el bus, en lugar de transferir un único dato, se transfiere un conjunto de datos. Cada transferencia de un conjunto de datos la denominamos *ráfaga*. Esto se puede hacer si el controlador de DMA dispone de una memoria intermedia (*buffer*) para almacenar el conjunto de datos que transferimos en cada ráfaga.

- $N_{ráfaga}$: número de datos que forman una ráfaga.

Si sabemos el tamaño de la memoria intermedia, podemos obtener el número de datos de la ráfaga de la manera siguiente:

- m_{buffer}: tamaño de la memoria intermedia de datos expresada en bytes.

$$N_{ráfaga} = m_{buffer} / m_{dato}$$

El tiempo que el procesador está ocupado o parado durante la transferencia se puede calcular de la manera siguiente:

$$t_{transf\_ráfaga} = t_{cesión} + (N_{ráfaga} \cdot t_{mem}) + t_{recup}$$
$$t_{transf\_bloque} = (N_{datos} / N_{ráfaga}) \cdot t_{transf\_ráfaga}$$
$$t_{cpu} = t_{prog} + t_{transf\_bloque} + t_{final}$$

Ocupación del procesador

Veamos cuál es la proporción de tiempo que el procesador dedica a la transferencia de E/S respecto al tiempo que tarda el periférico en hacer toda la transferencia, esto es, el porcentaje de ocupación del procesador.

Como el periférico, salvo casos excepcionales, es más lento que el procesador, es el periférico quien determina el tiempo necesario para hacer la transferencia de un bloque de datos.

$$t_{transf\_bloque} < t_{bloque}$$

Considerando toda la transferencia de E/S:

$$\%_{ocupación} = (t_{cpu} \cdot 100) / t_{bloque}$$

- t_{disp}: tiempo que el procesador está libre para hacer otras tareas durante la transferencia de E/S.

$$t_{disp} = t_{bloque} - t_{cpu}$$

Ejemplo

Hacemos una comparación de las tres técnicas de E/S utilizando datos concretos para transferir un bloque de datos.

Definimos primero los parámetros básicos de esta transferencia:

m_{dato}	tamaño del dato (bus de datos y registros)	4 Bytes
m_{bloque}	tamaño del bloque que se quiere transferir	1 MByte

Procesador:

t_{inst}	tiempo de ejecución de las instrucciones	16 ns
	Frecuencia del reloj del procesador	125 MHz
	Ciclos de reloj para ejecutar una instrucción	2

Bus del sistema y memoria:

$t_{cesión}$	tiempo para ceder el bus	2 ns
t_{recup}	tiempo para liberar el bus	2 ns
t_{mem}	tiempo de lectura/escritura en memoria	4 ns

Periférico:

Recordad

Submúltiplos de la unidad de tiempo (segundos):

10^{-3}: mili (m)

10^{-6}: micro (m)

10^{-9}: nano (n)

10^{-12}: pico (p)

Múltiplos de las unidades de datos (bits o bytes).

2^{10}: kilo (K)

2^{20}: mega (M)

2^{30}: giga (G)

2^{40}: tera (T)

Frecuencia

La frecuencia nos indica el número de ciclos de reloj por segundo. Calculando la inversa de la frecuencia obtenemos el tiempo de un ciclo de reloj del procesador.

t_{ciclo} = 1/frecuencia

v_{transf}	velocidad de transferencia	10 MBytes/s
$t_{latencia}$	latencia	7 ms

Técnicas de E/S

E/S programada:

N_{prog}	programar la transferencia	2 instrucciones
N_{final}	finalizar la transferencia	0 instrucciones

E/S por interrupciones:

N_{prog}	programar la transferencia	5 instrucciones
N_{final}	finalizar la transferencia	1 instrucción
t_{rec_int}	tiempo de reconocimiento de la interrupción	48 ns
N_{rsi}	número de instrucciones de la RSI:	12 instrucciones

E/S por DMA:

N_{prog}	programar la transferencia	15 instrucciones
N_{final}	finalizar la transferencia	15 instrucciones
m_{buffer}	tamaño de la memoria interna en modo ráfaga	16 Bytes

Calculamos el número de datos que forman el bloque que queremos transferir, que nos indica también el número de transferencias que se tienen que hacer.

$$N_{datos} = m_{bloque} / m_{dato} = 2^{20}/4$$

Calculamos el tiempo de transferencia de un dato y a partir de esta el tiempo de transferir un bloque.

v_{transf}	10 MBytes/s	$10 \cdot 2^{20}$ Bytes/s
t_{dato}	m_{dato} / v_{transf}	$4 / 10 \cdot 2^{20}$ s
$t_{latencia}$	7 ms	0,007 s
t_{bloque}	$t_{latencia} + (N_{datos} \cdot t_{dato})$	$0,007 + (2^{20}/4) \cdot (4/10 \cdot 2^{20}) = 0,007 + 0,1 = 0,107$ s

Ahora analizamos los tiempos de transferencia para cada una de las técnicas de E/S.

E/S programada:

t_{prog}	$N_{prog} \cdot t_{inst}$	$2 \cdot 16$ ns = 32 ns
t_{final}	$N_{final} \cdot t_{inst}$	$0 \cdot 16$ ns = 0 ns
t_{transf_dato}	t_{dato}	$4 / 10 \cdot 2^{20}$ B
t_{transf_bloque}	t_{bloque} = 0,107 s	107.000.000 ns
t_{cpu}	32 ns + 107.000.000 ns + 0 ns	107.000.032 ns = 107 ms

E/S por interrupciones:

t_{prog}	$N_{prog} \cdot t_{inst}$	$5 \cdot 16$ ns = 80 ns
t_{final}	$N_{final} \cdot t_{inst}$	$1 \cdot 16$ ns = 16 ns
t_{rsi}	$N_{rsi} \cdot t_{inst}$	$12 \cdot 16$ ns = 192 ns
t_{transf_dato}	$t_{rec\_int} + t_{rsi}$	48 ns + 192 ns = 240 ns
t_{transf_bloque}	$N_{dades} \cdot t_{transf\_dada}$	$(2^{20} / 4) \cdot 240$ ns = $2^{20} \cdot 60$ ns = 62.914.560 ns
t_{cpu}	$t_{prog} + t_{transf\_bloc} + t_{final}$	80 ns + 62.914.560 ns + 16 ns = 62.914.656 ns = 62,9 ms

E/S por DMA:

t_{prog}	$N_{prog} \cdot t_{inst}$	$15 \cdot 16$ ns = 240 ns
t_{final}	$N_{final} \cdot t_{inst}$	$15 \cdot 16$ ns = 240 ns
t_{transf_dato}	$t_{cesión} + t_{mem} + t_{recup}$	2 ns + 4 ns + 2 ns = 8 ns
t_{transf_bloque}	$N_{datos} \cdot t_{transf\_dato}$	$(2^{20} / 4) \cdot 8$ ns = $2 \cdot 2^{20}$ ns
t_{cpu}	$t_{prog} + t_{transf\_bloque} + t_{final}$	240 ns + $2 \cdot 2^{20}$ ns + 240 ns = 2.097.632 ns = 2,09 ms

E/S por DMA en modo ráfaga:

$N_{ráfaga}$	m_{buffer} / m_{dato}	16 Bytes / 4 Bytes = 4
$t_{transf\_ráfaga}$	$t_{cesión} + (N_{ráfaga} \cdot t_{mem}) + t_{recup}$	2 ns + (4 · 4 ns) + 2 ns = 20 ns
t_{transf_bloque}	$(N_{datos} / N_{ráfaga}) \cdot t_{transf\_ráfaga}$	$((2^{20} / 4) / 4) \cdot 20$ ns = $2^{16} \cdot 20$ ns = 1.310.720 ns
t_{cpu}	$t_{prog} + t_{transf\_bloque} + t_{final}$	240 ns + 1.310.720 ns + 240 ns = 1.311.200 ns = 1,31 ms

Comparamos el tiempo de ocupación del procesador en cada una de las técnicas de E/S.

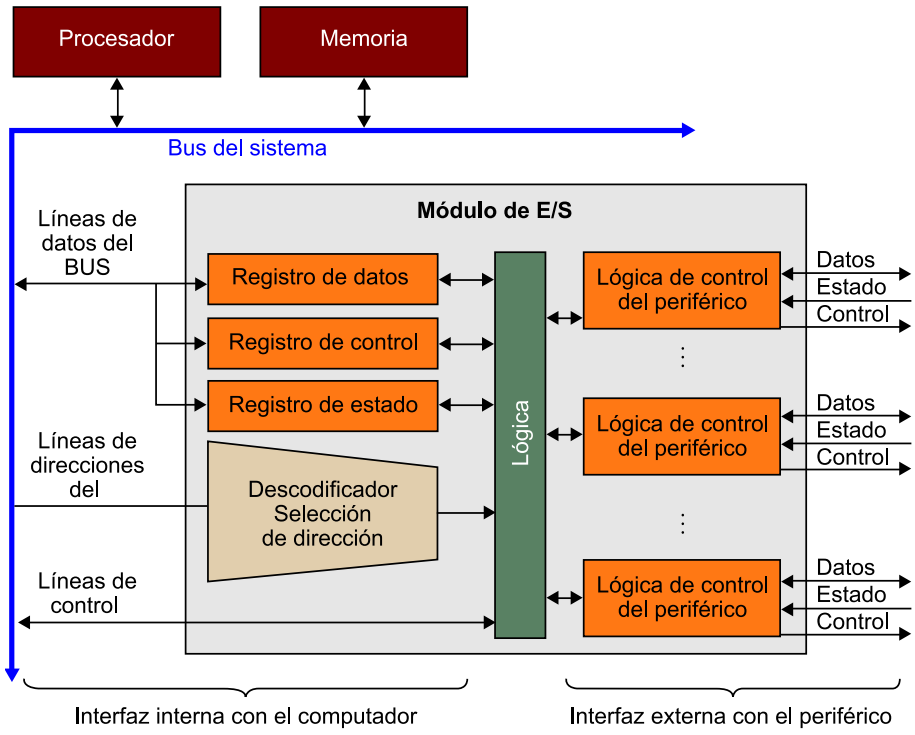
Periférico:

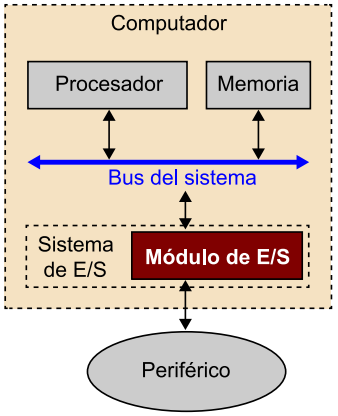
t_{bloque}	107 ms
$\%_{ocupación}$	$(t_{cpu} \cdot 100) / t_{bloque}$
t_{disp}	$t_{bloque} - t_{cpu}$

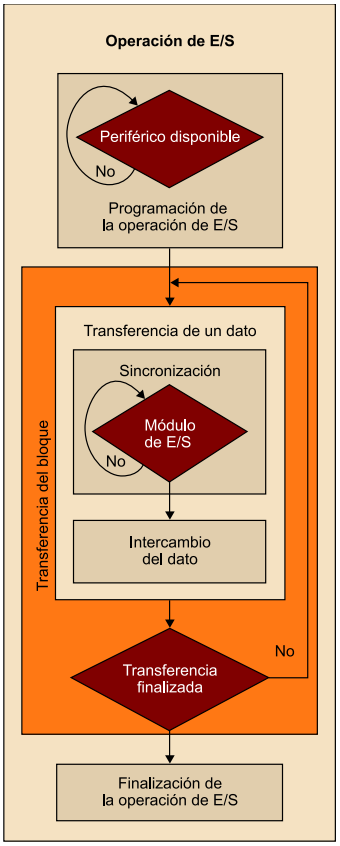
	E/S programada	**E/S por interrupciones**	**E/S por DMA**	**E/S por DMA en modo ráfaga**
t_{cpu}	107 ms	62,9 ms	2,09 ms	1,31 ms
$\%_{ocupación}$	100%	58,78%	1,95%	1,22%
t_{disp}	0 ms	44,1 ms	104,91 ms	105,69 ms

Resumen

En este módulo se han explicado en primer lugar los aspectos básicos del sistema de E/S de un computador. La estructura del sistema de E/S está formada por los periféricos, los módulos de E/S y los sistemas de interconexión externos como elementos principales.







A continuación se han descrito las fases que componen una operación básica de E/S.

1) Programación de la operación de E/S.

2) Transferencia de datos.

3) Finalización de la operación de E/S.

Para hacer la transferencia de datos se han descrito las principales técnicas de E/S:

- E/S programada.
- E/S por interrupciones.
- E/S por DMA.

Para cada una de estas técnicas se ha explicado el hardware necesario, cómo funciona la transferencia de un dato individual y de un bloque entre el computador y un periférico, y también cómo se debe gestionar la transferencia cuando tenemos más de un periférico conectado al computador y se han de gestionar simultáneamente múltiples transferencias, en cuyo caso los problemas principales que hay que resolver son la identificación del periférico con el que se ha de realizar la transferencia y la gestión de prioridades.

Finalmente, se ha hecho una comparación de las diferentes técnicas de E/S para analizar las prestaciones de un sistema de E/S y de esta manera profundizar un poco más en el funcionamiento de estas técnicas.

Programación en ensamblador (x86-64)

Miquel Albert Orenga
Gerard Enrique Manonellas

PID_00178132

Universitat Oberta de Catalunya
www.uoc.edu

Índice

Introducción

En este módulo nos centraremos en la programación de bajo nivel con el fin de conocer las especificaciones más relevantes de una arquitectura real concreta. En nuestro caso se trata de la arquitectura x86-64 (también denominada *AMD64* o *Intel 64*).

El lenguaje utilizado para programar a bajo nivel un computador es el lenguaje de ensamblador, pero para facilitar el desarrollo de aplicaciones y ciertas operaciones de E/S, utilizaremos un lenguaje de alto nivel, el lenguaje C; de esta manera, podremos organizar los programas según las especificaciones de un lenguaje de alto nivel, que son más flexibles y potentes, e implementar ciertas funciones con ensamblador para trabajar a bajo nivel los aspectos más relevantes de la arquitectura de la máquina.

La importancia del lenguaje de ensamblador radica en el hecho de que es el lenguaje simbólico que trabaja más cerca del procesador. Prácticamente todas las instrucciones de ensamblador tienen una correspondencia directa con las instrucciones binarias del código máquina que utiliza directamente el procesador. Esto lleva a que el lenguaje sea relativamente sencillo, pero que tenga un gran número de excepciones y reglas definidas por la misma arquitectura del procesador, y a la hora de programar, además de conocer las especificaciones del lenguaje, hay que conocer también las especificaciones de la arquitectura.

Así pues, este lenguaje permite escribir programas que pueden aprovechar todas las características de la máquina, lo que facilita la comprensión de la estructura interna del procesador. También puede aclarar algunas de las características de los lenguajes de alto nivel que quedan escondidas en su compilación.

El contenido del módulo se divide en siete apartados que tratan los siguientes temas:

- Descripción de la arquitectura x86-64 desde el punto de vista del programador, y en concreto del modo de 64 bits, en el cual se desarrollarán las prácticas de programación.

- Descripción del entorno de programación con el que se trabajará: edición, compilación y depuración de los programas en ensamblador y en C.

- Descripción de los elementos que componen un programa en ensamblador.

- Introducción a la programación en lenguaje C.

- Conceptos de programación en lenguaje de ensamblador y en lenguaje C, y cómo utilizar funciones escritas en ensamblador dentro de programas en C.

- También se incluye una referencia de las instrucciones más habituales del lenguaje de ensamblador con ejemplos de uso de cada una.

Este módulo no pretende ser un manual que explique todas las características del lenguaje de ensamblador y del lenguaje C, sino una guía que permita iniciarse en la programación a bajo nivel (hacer programas) con la ayuda de un lenguaje de alto nivel como el C.

Objetivos

Con el estudio de este módulo se pretende que el estudiante alcance los objetivos siguientes:

1. Aprender a utilizar un entorno de programación con lenguaje C y lenguaje de ensamblador.

2. Conocer el repertorio de instrucciones básicas de los procesadores de la familia x86-64, las diferentes formas de direccionamiento para tratar los datos y el control de flujo dentro de un programa.

3. Saber estructurar en subrutinas un programa; pasar y recibir parámetros.

4. Tener una idea global de la arquitectura interna del procesador según las características del lenguaje de ensamblador.

1. Arquitectura del computador

En este apartado se describirán a grandes rasgos los modos de operación y los elementos más importantes de la organización de un computador basado en la arquitectura x86-64 desde el punto de vista del juego de instrucciones utilizado por el programador.

x86-64 es una ampliación de la arquitectura x86. La arquitectura x86 fue lanzada por Intel con el procesador Intel 8086 en el año 1978 como una arquitectura de 16 bits. Esta arquitectura de Intel evolucionó a una arquitectura de 32 bits cuando apareció el procesador Intel 80386 en el año 1985, denominada inicialmente *i386* o *x86-32* y finalmente *IA-32*. Desde 1999 hasta el 2003, AMD amplió esta arquitectura de 32 bits de Intel a una de 64 bits y la llamó *x86-64* en los primeros documentos y posteriormente *AMD64*. Intel pronto adoptó las extensiones de la arquitectura de AMD bajo el nombre de *IA-32e* o *EM64T*, y finalmente la denominó *Intel 64*.

IA-64

Hay que apuntar que, con el procesador Itanium, Intel ha lanzado una arquitectura denominada *IA-64*, derivada de la IA-32 pero con especificaciones diferentes de la arquitectura x86-64 y que no es compatible con el juego de instrucciones desde un punto de vista nativo de las arquitecturas x86, x86-32 o x86-64. La arquitectura IA-64 se originó en Hewlett-Packard (HP) y más tarde fue desarrollada conjuntamente con Intel para ser utilizada en los servidores empresariales y sistemas de computación de alto rendimiento.

La arquitectura x86-64 (AMD64 o Intel 64) de 64 bits da un soporte mucho mayor al espacio de direcciones virtuales y físicas, proporciona registros de propósito general de 64 bits y otras mejoras que conoceremos más adelante.

Cualquier procesador actual también dispone de una serie de unidades específicas para trabajar con números en punto flotante (juego de instrucciones de la FPU), y de extensiones para trabajar con datos multimedia (juego de instrucciones MMX y SSE en el caso de Intel, o 3DNow! en el caso de AMD). Estas partes no se describen en estos materiales, ya que quedan fuera del programa de la asignatura.

Nos centraremos, por lo tanto, en las partes relacionadas con el trabajo con enteros y datos alfanuméricos.

1.1. Modos de operación

Los procesadores con arquitectura x86-64 mantienen la compatibilidad con los procesadores de la arquitectura IA-32 (x86-32). Por este motivo, disponen de los mismos modos de operación de la arquitectura IA-32, lo que permite

mantener la compatibilidad y ejecutar aplicaciones de 16 y 32 bits, pero además añaden un modo nuevo denominado *modo extendido* (o modo *IA-32e*, en el caso de Intel), dentro del cual se puede trabajar en modo real de 64 bits.

Los procesadores actuales soportan diferentes modos de operación, pero como mínimo disponen de un modo protegido y de un modo supervisor.

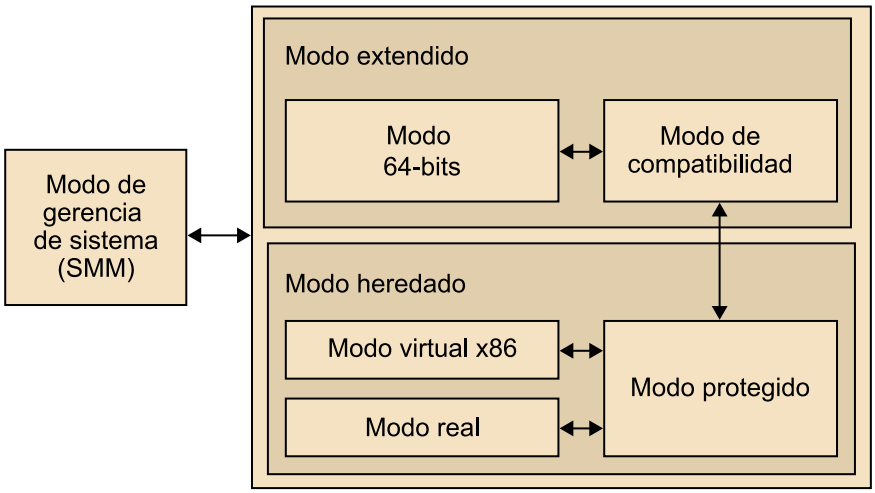
El modo de supervisor es utilizado por el núcleo del sistema para las tareas de bajo nivel que necesitan un acceso sin restricciones al hardware, como puede ser el control de la memoria o la comunicación con otros dispositivos. El modo protegido, en cambio, se utiliza para casi todo el resto de las tareas.

Cuando ejecutamos programas en modo protegido, solo podremos utilizar el hardware haciendo llamadas al sistema operativo, que es el que lo puede controlar en modo supervisor. Puede haber otros modos similares al protegido, como el modo virtual, que se utiliza para emular otros procesadores de la misma familia, y de esta manera mantener la compatibilidad con los procesadores anteriores.

Cuando un equipo se inicia por primera vez se ejecutan los programas de la BIOS, del gestor de arranque y del sistema operativo que tienen acceso ilimitado al hardware; cuando el equipo se ha iniciado, el sistema operativo puede pasar el control a otro programa y poner el procesador en modo protegido.

En modo protegido, los programas tienen acceso a un conjunto más limitado de instrucciones y solo podrán dejar el modo protegido haciendo una petición de interrupción que devuelve el control al sistema operativo; de esta manera se garantiza el control para acceder al hardware.

Modos de operación de la arquitectura x86-64



Características de los dos modos principales de operación en la arquitectura x86-64

Modo de operación		Sistema operativo	Las aplicaciones necesitan recompilación	Por defecto		Tamaño de los registros de propósito general
				Tamaño (en bits) de las direcciones	Tamaño (en bits) de los operandos	
Modo extendido	Modo de 64 bits	Sistema operativo de 64 bits	sí	64	32	64
	Modo compatibilidad		no	32		32
				16	16	16
Modo heredado	Modo protegido	Sistema operativo de 32 bits	no	32	32	32
				16	16	
	Modo virtual-8086			16	16	16
	Modo real	Sistema operativo de 16 bits				

1.1.1. Modo extendido de 64 bits

El modo extendido de 64 bits es utilizado por los sistemas operativos de 64 bits. Dentro de este modo general, se dispone de un modo de operación de 64 bits y de un modo de compatibilidad con los modos de operación de las arquitecturas de 16 y 32 bits.

En un sistema operativo de 64 bits, los programas de 64 bits se ejecutan en modo de 64 bits y las aplicaciones de 16 y 32 bits se ejecutan en modo de compatibilidad. Los programas de 16 y 32 bits que se tengan que ejecutar en modo real o virtual x86 no se podrán ejecutar en modo extendido si no son emulados.

Modo de 64 bits

El modo de 64 bits proporciona acceso a 16 registros de propósito general de 64 bits. En este modo se utilizan direcciones virtuales (o lineales) que por defecto son de 64 bits y se puede acceder a un espacio de memoria lineal de 2^{64} bytes.

El tamaño por defecto de los operandos se mantiene en 32 bits para la mayoría de las instrucciones.

El tamaño por defecto puede ser cambiado individualmente en cada instrucción mediante modificadores. Además, soporta direccionamiento relativo a PC (RIP en esta arquitectura) en el acceso a los datos de cualquier instrucción.

Modo de compatibilidad

El modo de compatibilidad permite a un sistema operativo de 64 bits ejecutar directamente aplicaciones de 16 y 32 bits sin necesidad de recompilarlas.

En este modo, las aplicaciones pueden utilizar direcciones de 16 y 32 bits, y pueden acceder a un espacio de memoria de 4 Gbytes. El tamaño de los operandos puede ser de 16 y 32 bits.

Desde el punto de vista de las aplicaciones, se ve como si se estuviera trabajando en el modo protegido dentro del modo heredado.

1.1.2. Modo heredado de 16 y 32 bits

El modo heredado de 16 y 32 bits es utilizado por los sistemas operativos de 16 y 32 bits. Cuando el sistema operativo utiliza los modos de 16 bits o de 32 bits, el procesador actúa como un procesador x86 y solo se puede ejecutar código de 16 o 32 bits. Este modo solo permite utilizar direcciones de 32 bits, de manera que limita el espacio de direcciones virtual a 4 GB.

Dentro de este modo general hay tres modos:

1) Modo real. Implementa el modo de programación del Intel 8086, con algunas extensiones, como la capacidad de poder pasar al modo protegido o al modo de gestión del sistema. El procesador se coloca en modo real al iniciar el sistema y cuando este se reinicia.

Es el único modo de operación que permite utilizar un sistema operativo de 16 bits.

El modo real se caracteriza por disponer de un espacio de memoria segmentado de 1 MB con direcciones de memoria de 20 bits y acceso a las direcciones del hardware (sistema de E/S). No proporciona soporte para la protección de memoria en sistemas multitarea ni de código con diferentes niveles de privilegio.

2) Modo protegido. Este es el modo por defecto del procesador. Permite utilizar características como la memoria virtual, la paginación o la computación multitarea.

Entre las capacidades de este modo está la posibilidad de ejecutar código en modo real, modo virtual-8086, en cualquier tarea en ejecución.

3) Modo virtual 8086. Este modo permite ejecutar programas de 16 bits como tareas dentro del modo protegido.

1.1.3. El modo de gestión de sistema

El modo de gestión de sistema o *system management mode* (SMM) es un modo de operación transparente del software convencional (sistema operativo y aplicaciones). En este modo se suspende la ejecución normal (incluyendo el sistema operativo) y se ejecuta un software especial de alto privilegio diseñado para controlar el sistema. Tareas habituales de este modo son la gestión de energía, tareas de depuración asistidas por hardware, ejecución de microhardware o un software asistido por hardware. Este modo es utilizado básicamente por la BIOS y por los controladores de dispositivo de bajo nivel.

Accedemos al SMM mediante una interrupción de gestión del sistema (SMI, *system management interrupt*). Una SMI puede ser generada por un acontecimiento independiente o ser disparada por el software del sistema por el acceso a una dirección de E/S considerada especial por la lógica de control del sistema.

1.2. El modo de 64 bits

En este subapartado analizaremos con más detalle las características más importantes del modo de 64 bits de la arquitectura x86-64.

Nota

Es importante para el programador de bajo nivel conocer las características más relevantes de este modo, ya que será el modo en el que se desarrollarán las prácticas de programación.

Los elementos que desde el punto de vista del programador son visibles en este modo de operación son los siguientes:

1) Espacio de memoria: un programa en ejecución en este modo puede acceder a un espacio de direcciones lineal de 2^{64} bytes. El espacio físico que realmente puede dirigir el procesador es inferior y depende de la implementación concreta de la arquitectura.

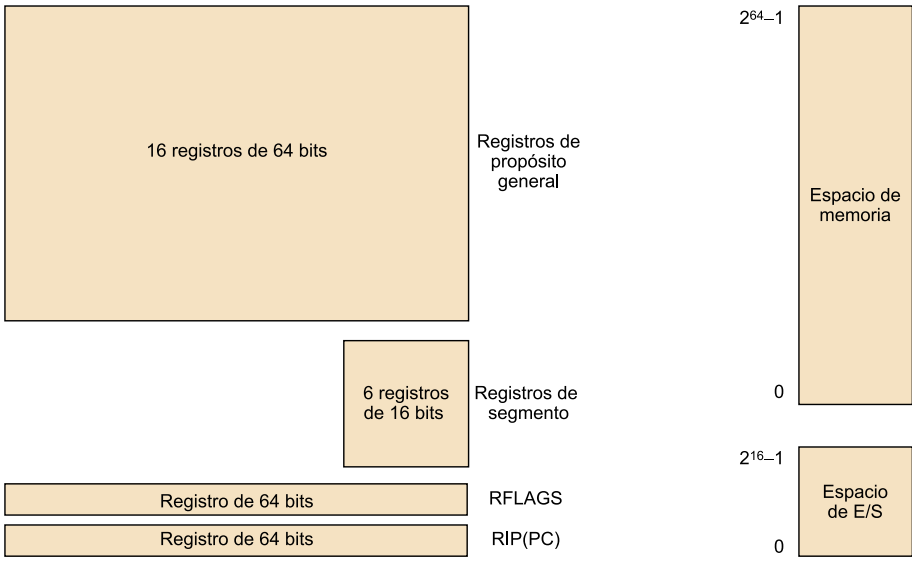
2) Registros: hay 16 registros de propósito general de 64 bits, que soportan operaciones de byte (8 bits), word (16 bits), double word (32 bits) y quad word (64 bits).

- El registro contador de programa (RIP, *instruction pointer register*) es de 64 bits.

- El registro de bits de estado también es de 64 bits (RFLAGS). Los 32 bits de la parte alta están reservados; los 32 bits de la parte baja son accesibles y corresponden a los mismos bits de la arquitectura IA-32 (registro EFLAGS).

- Los registros de segmento en general no se utilizan en el modo de 64 bits.

Entorno de ejecución en el modo de 64 bits



1.2.1. Organización de la memoria

El procesador accede a la memoria utilizando direcciones físicas de memoria. El tamaño del espacio de direcciones físico accesible para los procesadores depende de la implementación: supera los 4 Gbytes, pero es inferior a los 2^{64} bytes posibles.

En el modo de 64 bits, la arquitectura proporciona soporte a un espacio de direcciones virtual o lineal de 64 bits (direcciones de 0 a $2^{64} - 1$), pero como el espacio de direcciones físico es inferior al espacio de direcciones lineal, es necesario un mecanismo de correspondencia entre las direcciones lineales y las direcciones físicas (mecanismo de paginación).

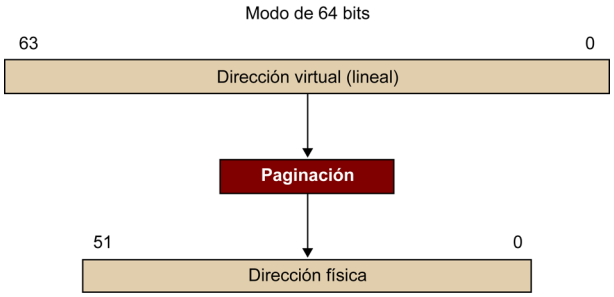
Al trabajar en un espacio lineal de direcciones, no se utilizan mecanismos de segmentación de la memoria, de manera que no son necesarios los registros de segmentos, excepto los registros de segmento FS y GS, que se pueden utilizar como registro base en el cálculo de direcciones de los modos de direccionamiento relativo.

Paginación

Este mecanismo es transparente para los programas de aplicación, y por lo tanto para el programador, y viene gestionado por el hardware del procesador y el sistema operativo.

Las direcciones virtuales son traducidas a direcciones físicas de memoria utilizando un sistema jerárquico de tablas de traducción gestionadas por el software del sistema (sistema operativo).

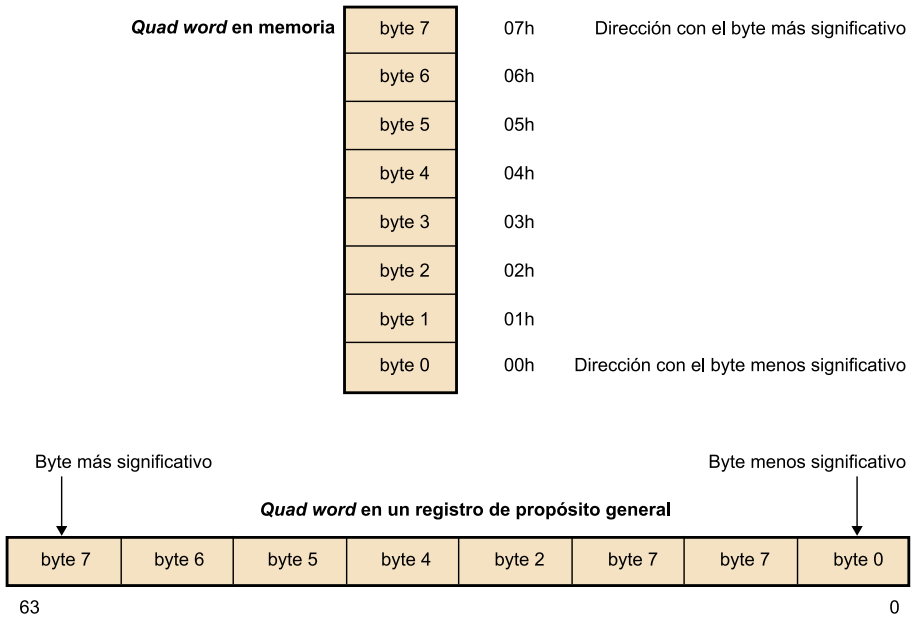
Básicamente, una dirección virtual se divide en campos, y cada campo actúa como índice dentro de una de las tablas de traducción. Cada valor en la posición indexada actúa como dirección base de la tabla de traducción siguiente.



Orden de los bytes

Los procesadores x86-64 utilizan un sistema de ordenación de los bytes cuando se accede a los datos que se encuentran almacenados en la memoria. En concreto, se utiliza un sistema *little-endian*, en el cual el byte de menos peso de un dato ocupa la dirección más baja de memoria.

En los registros también se utiliza el orden *little-endian* y por este motivo el byte menos significativo de un registro se denomina *byte 0*.



Tamaño de las direcciones

Los programas que se ejecutan en el modo de 64 bits generan directamente direcciones de 64 bits.

Modo compatibilidad

Los programas que se ejecutan en el modo compatibilidad generan direcciones de 32 bits. Estas direcciones son extendidas añadiendo ceros a los 32 bits más significativos de la

dirección. Este proceso es gestionado por el hardware del procesador y es transparente para el programador.

Tamaño de los desplazamientos y de los valores inmediatos

En el modo de 64 bits los desplazamientos utilizados en los direccionamientos relativos y los valores inmediatos son siempre de 32 bits, pero vienen extendidos a 64 bits manteniendo el signo.

Hay una excepción a este comportamiento: en la instrucción MOV se permite especificar un valor inmediato de 64 bits.

1.2.2. Registros

Los procesadores de la arquitectura x86-64 disponen de un banco de registros formado por registros de propósito general y registros de propósito específico.

Registros de propósito general hay 16 de 64 bits y de propósito específico hay 6 registros de segmento de 16 bits, también hay un registro de estado de 64 bits (RFLAGS) y un registro contador de programa también de 64 bits (RIP).

Registros de propósito general

Son 16 registros de datos de 64 bits (8 bytes): RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP y R8-R15.

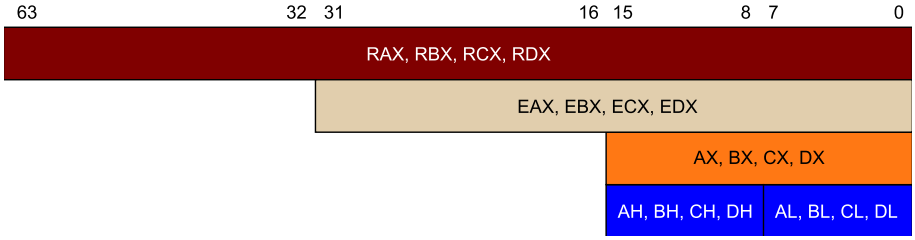
Los 8 primeros registros se denominan de manera parecida a los 8 registros de propósito general de 32 bits disponibles en la arquitectura IA-32 (EAX, EBX, ECX, EDX, ESI, EDI, EBP y ESP).

Los registros son accesibles de cuatro maneras diferentes:

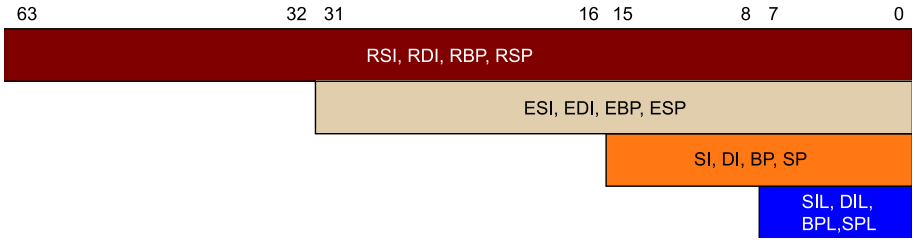
1) Como registros completos de 64 bits (quad word).
2) Como registros de 32 bits (double word), accediendo a los 32 bits de menos peso.
3) Como registros de 16 bits (word), accediendo a los 16 bits de menos peso.
4) Como registros de 8 bits (byte), permitiendo acceder individualmente a uno o dos de los bytes de menos peso según el registro.

El acceso a registros de byte tiene ciertas limitaciones según el registro. A continuación se presenta la nomenclatura que se utiliza según si se quiere acceder a registros de 8, 16, 32 o 64 bits y según el registro.

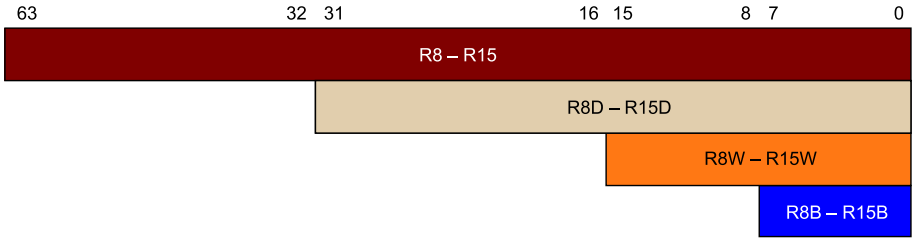
Registros RAX, RBX, RCX y RDX



Registros RSI, RDI, RBP, RSP



Registros R8-R15



Hay algunas limitaciones en el uso de los registros de propósito general:

- En una misma instrucción no se puede usar un registro del conjunto AH, BH, CH, DH junto con uno del conjunto SIL, DIL, BPL, SPL, R8B – R15B.
- Registro RSP: tiene una función especial, funciona como apuntador de pila, contiene siempre la dirección del primer elemento de la pila. Si lo utilizamos con otras finalidades, perderemos el acceso a la pila.
- Cuando se utiliza un registro de 32 bits como operando destino de una instrucción, la parte alta del registro está fijada en 0.

Registros de propósito específico

Podemos distinguir varios registros de propósito específico:

1) Registros de segmento: hay 6 registros de segmento de 16 bits.

- CS: *code segment*
- DS: *data segment*
- SS: *stack segment*
- ES: *extra segment*
- FS: *extra segment*
- GS: *extra segment*

Estos registros se utilizan básicamente en los modelos de memoria segmentados (heredados de la arquitectura IA-32). En estos modelos, la memoria se divide en segmentos, de manera que en un momento dado el procesador solo es capaz de acceder a seis segmentos de la memoria utilizando cada uno de los seis registros de segmento.

En el modo de 64 de bits, estos registros prácticamente no se utilizan, ya que se trabaja con el modelo de memoria lineal y el valor de estos registros se encuentra fijado en 0 (excepto en los registros FS y GS, que pueden ser utilizados como registros base en el cálculo de direcciones).

2) Registro de instrucción o *instruction pointer* (RIP): es un registro de 64 bits que actúa como registro contador de programa (PC) y contiene la dirección efectiva (o dirección lineal) de la instrucción siguiente que se ha de ejecutar.

Cada vez que se lee una instrucción nueva de la memoria, se actualiza con la dirección de la instrucción siguiente que se tiene que ejecutar; también se puede modificar el contenido del registro durante la ejecución de una instrucción de ruptura de secuencia (llamada a subrutina, salto condicional o incondicional).

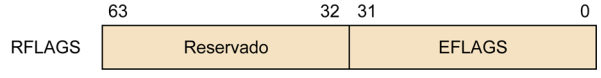
3) Registro de estado o *Flags register* (RFLAGS): es un registro de 64 bits que contiene información sobre el estado del procesador e información sobre el resultado de la ejecución de las instrucciones.

Solo se utiliza la parte baja del registro (bits de 31 a 0), que corresponde al registro EFLAGS de la arquitectura IA-32; la parte alta del registro está reservada y no se utiliza.

Bits de resultado

Las instrucciones de salto condicional consultan algunos de los bits del registro, denominados *bits de resultado*. Hay más información sobre el tema en el manual básico del juego de instrucciones en el apartado 7 de este módulo.

El uso habitual del registro de estado consiste en consultar el valor individual de sus bits; eso se puede conseguir con instrucciones específicas, como, por ejemplo, las instrucciones de salto condicional que consultan uno o más bits para determinar si saltan o no según cómo ha dejado estos bits la última instrucción que los ha modificado (no ha de ser la última instrucción que se ha ejecutado).



En la siguiente tabla se describe el significado de los bits de este registro.

bit 0	CF	Carry Flag	0 = no se ha producido transporte; 1 = sí que se ha producido transporte
bit 1		No definido	
bit 2	PF	Parity Flag	0 = número de bits 1 es impar; 1 = número de bits 1 es par

bit 3		No definido	
bit 4	AF	Auxiliary Carry Flag	0 = no se ha producido transporte en operaciones BCD; 1 = sí que se ha producido transporte en operaciones BCD
bit 5		No definido	
bit 6	ZF	Zero Flag	0 = el resultado no ha sido cero; 1 = el resultado ha sido cero
bit 7	SF	Sign Flag	0 = el resultado no ha sido negativo; 1 = el resultado ha sido negativo
bit 8	TF	Trap Flag	Facilita la ejecución paso a paso.
bit 9	IF	Interrupt Enable Flag	Reservado para el sistema operativo en modo protegido.
bit 10	DF	Direction Flag	0 = autoincremento hacia direcciones altas; 1 = autoincremento hacia direcciones bajas
bit 11	OF	Overflow Flag	0 = no se ha producido desbordamiento; 1 = sí que se ha producido desbordamiento
bit 12	IOPL	I/O Privilege Level	Reservado para el sistema operativo en modo protegido.
bit 13	IOPL	I/O Privilege Level	Reservado para el sistema operativo en modo protegido.
bit 14	NT	Nested Task Flag	Reservado para el sistema operativo en modo protegido.
bit 15		No definido	
bit 16	RF	Resume Flag	Facilita la ejecución paso a paso.
bit 17	VM	Virtual-86 Mode Flag	Reservado para el sistema operativo en modo protegido.
bit 18	AC	Alignment Check Flag	Reservado para el sistema operativo en modo protegido.
bit 19	VIF	Virtual Interrupt Flag	Reservado para el sistema operativo en modo protegido.
bit 20	VIP	Virtual Interrupt Pending	Reservado para el sistema operativo en modo protegido.
bit 21	ID	CPU ID	Si el bit puede ser modificado por los programas en el espacio de usuario, CPUID está disponible.
bit 22-31		No definidos	

2. Lenguajes de programación

Un programa es un conjunto de instrucciones que siguen unas normas sintácticas estrictas especificadas por un lenguaje de programación concreto y diseñado de manera que, cuando se ejecuta en una máquina concreta, realiza una tarea determinada sobre un conjunto de datos. Los programas, por lo tanto, están formados por código (instrucciones) y datos. De manera genérica, el fichero que contiene el conjunto de instrucciones y la definición de los datos que utilizaremos se denomina **código fuente**.

Para poder ejecutar un programa, lo hemos de traducir a un lenguaje que pueda entender el procesador; este proceso se llama habitualmente *compilación*. Convertimos el código fuente en **código ejecutable**. Este proceso para generar el código ejecutable normalmente se descompone en dos fases: en la primera fase el código fuente se traduce a un **código objeto** y en la segunda fase se enlaza este código objeto y otros códigos objeto del mismo tipo que ya tengamos generados, si es necesario, para generar el código ejecutable final.

Código objeto

El código objeto es un código de bajo nivel formado por una colección organizada de secuencias de códigos siguiendo un formato estándar. Cada secuencia, en general, contiene instrucciones para la máquina en la que se tiene que ejecutar el código para llevar a cabo alguna tarea concreta; también puede tener otro tipo de información asociada (por ejemplo, información de reubicación, comentarios o los símbolos del programa para la depuración).

Para iniciar la ejecución, es necesario que tanto el código como los datos (al menos una parte) estén cargados en la memoria del computador.

Para escribir un programa, hay que utilizar un lenguaje de programación que nos permita especificar el funcionamiento que se quiere que tenga el programa. Hay muchos lenguajes de programación y, según la funcionalidad que queramos dar al programa, será mejor utilizar uno u otro. En muchos casos, elegir uno no es una tarea fácil y puede condicionar mucho el desarrollo y el funcionamiento. Hay muchas maneras de clasificarlos, pero por el interés de esta asignatura y como es una de las maneras más generales de hacerlo, los clasificaremos según el nivel de abstracción (proximidad a la máquina) en dos lenguajes:

1) Lenguajes de bajo nivel

Se denominan *lenguajes de bajo nivel* porque dependen de la arquitectura del procesador en el que queremos ejecutar el programa y porque no disponen de sentencias con una estructura lógica que faciliten la programación y la comprensión del código para el programador, sino que están formados por una lista de instrucciones específicas de una arquitectura.

Podemos distinguir entre dos lenguajes:

a) Lenguaje de máquina. Lenguaje que puede interpretar y ejecutar un procesador determinado. Este lenguaje está formado por instrucciones codificadas en binario (0 y 1). Es generado por un compilador a partir de las especificaciones de otro lenguaje simbólico o de alto nivel. Es muy difícil de entender para el programador y sería muy fácil cometer errores si se tuviera que codificar.

b) Lenguaje de ensamblador. Lenguaje simbólico que se ha definido para que se puedan escribir programas con una sintaxis próxima al lenguaje de máquina, pero sin tener que escribir el código en binario, sino utilizando una serie de mnemónicos más fáciles de entender para el programador. Para ejecutar estos programas también es necesario un proceso de traducción, generalmente denominado *ensamblaje*, pero más sencillo que en los lenguajes de alto nivel.

2) Lenguajes de alto nivel. Los lenguajes de alto nivel no tienen relación directa con un lenguaje de máquina concreto, no dependen de la arquitectura del procesador en el que se ejecutarán y disponen de sentencias con una estructura lógica que facilitan la programación y la comprensión del código para el programador; las instrucciones habitualmente son palabras extraídas de un lenguaje natural, generalmente el inglés, para que el programador las pueda entender mejor. Para poder ejecutar programas escritos en estos lenguajes, es necesario un proceso previo de compilación para pasar de lenguaje de alto nivel a lenguaje de máquina; el código generado en este proceso dependerá de la arquitectura del procesador en el que se ejecutará.

2.1. Entorno de trabajo

El entorno de trabajo que utilizaremos para desarrollar los problemas y las prácticas de programación será un PC basado en procesadores x86-64 (Intel64 o AMD64) sobre el cual se ejecutará un sistema operativo Linux de 64 bits.

El entorno de trabajo se podrá ejecutar de forma nativa sobre un PC con un procesador con arquitectura x86-64, con sistema operativo Linux de 64 bits, o utilizando algún software de virtualización que permita ejecutar un sistema operativo Linux de 64 bits.

Los lenguajes de programación que utilizaremos para escribir el código fuente de los problemas y de las prácticas de programación de la asignatura serán: un lenguaje de alto nivel, el lenguaje C estándar, para diseñar el programa principal y las operaciones de E/S y un lenguaje de bajo nivel, el lenguaje ensamblador x86-64, para implementar funciones concretas y ver cómo trabaja esta arquitectura a bajo nivel.

Hay diferentes sintaxis de lenguaje ensamblador x86-64; utilizaremos la sintaxis NASM (Netwide Assembler), basada en la sintaxis Intel.

En el espacio de recursos del aula podéis encontrar documentación sobre la instalación y utilización de las herramientas necesarias para editar, compilar o ensamblar, enlazar, y depurar o ejecutar los problemas y las prácticas que se plantearán en esta asignatura, tanto para trabajar de forma nativa como desde un entorno de virtualización.

Máquina virtual

En el espacio de recursos de la asignatura os podéis descargar una imagen de una máquina virtual que ya tiene instalado un Linux de 64 bits y todas las herramientas del entorno de trabajo. Esta máquina virtual se podrá ejecutar en un sistema operativo Linux, o Windows; para poder ejecutarla, se debe instalar el software de virtualización.

3. El lenguaje de ensamblador para la arquitectura x86-64

3.1. Estructura de un programa en ensamblador

Un programa ensamblador escrito con sintaxis NASM está formado por tres secciones o segmentos: *.data* para los datos inicializados, *.bss* para los datos no inicializados *y .text* para el código:

```
section .data

section .bss

section .text
```

Para definir una sección se pueden utilizar indistintamente las directivas *section* y *segment*.

La sección *.bss* no es necesaria si se inicializan todas las variables que se declaran en la *sección .data*.

La sección *.text* permite también definir variables y, por lo tanto, permitiría prescindir también de la sección *.data*, aunque no es recomendable. Por lo tanto, esta es la única sección realmente obligatoria en todo programa.

La sección *.text* debe empezar siempre con la directiva *global*, que indica al GCC cuál es el punto de inicio del código. Cuando se utiliza GCC para generar un ejecutable, el nombre de la etiqueta donde se inicia la ejecución del código se ha de denominar obligatoriamente *main*; también ha de incluir una llamada al sistema operativo para finalizar la ejecución del programa y devolver el control al terminal desde el que hemos llamado al programa.

En el programa de ejemplo `hola.asm` utilizado anteriormente hemos destacado la declaración de las secciones *.data* y *.text*, la directiva global y la llamada al sistema operativo para finalizar la ejecución del programa.

```
                    ;1: fichero hola.asm
section .data       ;2: Inicio de la sección de datos
  msg db "Hola!",10 ;3:
                    ;4: El 10 corresponde al código ASCII del salto de línea.
                    ;5:
section .text       ;6: Inicio de la sección de código.
  global main       ;7: Esta directiva es para hacer visible
                    ;8: una etiqueta para el compilador de C.
                    ;9:
  main:             ;10: Por defecto el compilador de C reconoce como
                    ;11: punto de inicio del programa la etiqueta main.
                    ;12: Mostrar un mensaje
  mov rax,4         ;13: Pone el valor 4 en el registro rax
                    ;14: para hacer la llamada a la funció write (sys_write)
  mov rbx,1         ;15: Pone el valor 1 en el registro RBX
                    ;16: para indicar el descriptor que hace referencia
                    ;17: a la salida estándar.
  mov rcx,msg       ;18: Pone la dirección de la variable msg
                    ;19: en el registro RCX
  mov rdx,6         ;20: Pone la longitud del mensaje incluido el 10
                    ;21: del final en el registro RDX
  int 80h           ;22: llama al sistema operativo
                    ;23:
                    ;24: devuelve el control al terminal del sistema operativo.
  mov rax,1         ;25: Pone el valor 1 en el registro rax
                    ;26: para hacer la llamada a la función exit (sys_exit)
  mov rbx,0         ;27: Pone el valor 0 en el registro RBX
                    ;28: para indicar el código de retorno (0=sin errores)
  int 80h           ;29: llama al sistema operativo
                    ;30:
```

A continuación, se describen los diferentes elementos que se pueden introducir dentro de las secciones de un programa ensamblador.

3.2. Directivas

Las directivas son pseudooperaciones que solo son reconocidas por el ensamblador. No se deben confundir con las instrucciones, a pesar de que en algunos casos pueden añadir código a nuestro programa. Su función principal es declarar ciertos elementos de nuestro programa para que puedan ser identificados más fácilmente por el programador y también para facilitar la tarea de ensamblaje.

A continuación, explicaremos algunas de las directivas que podemos encontrar en un programa con código ensamblador y que tendremos que utilizar: definición de constantes, definición de variables y definición de otros elementos.

3.2.1. Definición de constantes

Una constante es un valor que no puede ser modificado por ninguna instrucción del código del programa. Realmente una constante es un nombre que se da para referirse a un valor determinado.

La declaración de constantes se puede hacer en cualquier parte del programa: al principio del programa fuera de las secciones *.data, .bss*, *.text* o dentro de cualquiera de las secciones anteriores.

Las constantes son útiles para facilitar la lectura y posibles modificaciones del código. Si, por ejemplo, utilizamos un valor en muchos lugares de un programa, como podría ser el tamaño de un vector, y queremos probar el programa con un valor diferente, tendremos que cambiar este valor en todos los lugares donde lo hemos utilizado, con la posibilidad de que dejemos uno sin modificar y, por lo tanto, de que alguna cosa no funcione; en cambio, si definimos una constante con este valor y en el código utilizamos el nombre de la constante, modificando el valor asignado a la constante, se modificará todo.

Para definir constantes se utiliza la directiva `equ`, de la manera siguiente:

```
nombre_constante equ valor
```

Ejemplos de definiciones de constantes

```
tamañoVec equ 5
ServicioSO equ 80h
Mensaje1 equ 'Hola'
```

3.2.2. Definición de variables

La declaración de variables en un programa en ensamblador se puede incluir en la sección *.data* o en la sección *.bss*, según el uso de cada una.

Sección *.data*, variables inicializadas

Las variables de esta sección se definen utilizando las siguientes directivas:

- `db`: define una variable de tipo byte, 8 bits.
- `dw`: define una variable de tipo palabra (word), 2 bytes = 16 bits.
- `dd`: define una variable de tipo doble palabra (double word), 2 palabras = 4 bytes = 32 bits.
- `dq`: define una variable de tipo cuádruple palabra (quad word), 4 palabras = 8 bytes = 64 bits.

El formato utilizado para definir una variable empleando cualquiera de las directivas anteriores es el mismo:

```
nombre_variable directiva valor_inicial
```

Ejemplos

```
var1 db 255 ;                    define una variable con el valor FFh
Var2 dw 65535 ;                    en hexadecimal FFFFh
var4 dd 4294967295 ;                en hexadecimal FFFFFFFFh
var8 dq 18446744073709551615 ; en            hexadecimal
FFFFFFFFFFFFFFFFh
```

Se puede utilizar una constante para inicializar variables. Solo es necesario que la constante se haya definido antes de su primera utilización.

```
tamañoVec equ 5
indexVec db tamañoVec
```

Los valores iniciales de las variables y constantes se pueden expresar en bases diferentes como decimal, hexadecimal, octal y binario, o también como caracteres y cadenas de caracteres.

Si se quieren separar los dígitos de los valores numéricos iniciales para facilitar la lectura, se puede utilizar el símbolo '_', sea cual sea la base en la que esté expresado el número.

```
var4 dd 4_294_967_295
var8 dq FF_FF_FF_FF_FF_FF_FF_FFh
```

Los valores numéricos se consideran por defecto en decimal, pero también se puede indicar explícitamente que se trata de un **valor decimal** finalizando el número con el carácter *d*.

```
var db 67   ;el valor 67 decimal
var db 67d ;el mismo valor
```

Los **valores hexadecimales** han de empezar por *0x*, *0h* o *$*, o deben finalizar con una *h*.

Si se especifica el valor con *$* al principio o con una *h* al final, el número no puede empezar por una letra; si el primer dígito hexadecimal ha de ser una letra (*A, B, C, D, E, F*), es necesario añadir un 0 delante.

```
var db 0xFF
var dw 0hA3
var db $43
var dw 33FFh
```

Las definiciones siguientes son incorrectas:

```
var db      $FF ;deberíamos escribir: var db $0FF
var db      FFh ;deberíamos escribir: var db 0FFh
```

Los **valores octales** han de empezar por *0o* o *0q*, o deben finalizar con el carácter *o* o *q*.

```
var db 103o
var db 103q
var db 0o103
var db 0q103
```

Los **valores binarios** han de empezar por *0b* o finalizar con el carácter *b*.

```
var db 0b01000011
var dw 0b0110_1100_0100_0011
var db 01000011b
var dw 0110_0100_0011b
```

Los **caracteres** y las **cadenas** de caracteres han de escribirse entre comillas simples (''), dobles ("") o comillas abiertas *backquotes* (` `):

```
var db 'A'
var db "A"
var db `A`
```

Las cadenas de caracteres (*strings*) se definen de la misma manera:

```
cadena db 'Hola' ;define una cadena formada por 4 caracteres
cadena db "Hola"
cadena db `Hola`
```

Las cadenas de caracteres también se pueden definir como una serie de caracteres individuales separados por comas.

Las definiciones siguientes son equivalentes:

```
cadena db 'Hola'
cadena db 'H','o','l','a'
cadena db 'Hol','a'
```

Las cadenas de caracteres pueden incluir caracteres no imprimibles y caracteres especiales; estos caracteres se pueden incluir con su codificación (ASCII), también pueden utilizar el código ASCII para cualquier otro carácter aunque sea imprimible.

```
cadena db 'Hola',10    ;añade el carácter ASCII de salto de línea
cadena db 'Hola',9   ;añade el carácter ASCII de tabulación
cadena db 72,111,108,97 ;añade igualmente la cadena 'Hola'
```

Si la cadena se define entre comillas abiertas (` `), también se admiten algunas secuencias de escape iniciadas con \:

\n: salto de línea
\t: tabulador
\e: el carácter ESC (ASCII 27)
\0x[valor]: [valor] ha de ser 1 byte en hexadecimal, expresado con 2 dígitos hexadecimales.
\u[valor]: [valor] ha de ser la codificación hexadecimal de un carácter en formato UTF.

```
cadena db `Hola\n`   ;añade el carácter de salto de línea al final
cadena db `Hola\t`   ;añade el carácter de salto de línea al final
cadena db `\e[10,5H`   ;define una secuencia de escape que empieza con
                        ;el carácter ESC = \e = ASCII(27)
cadena db `Hola \u263a` ;muestra: Hola☺
```

Si queremos definir una cadena de caracteres que incluya uno de estos símbolos (' , " ,`), se debe delimitar la cadena utilizando unas comillas de otro tipo o poner una contrabarra (\) delante.

```
"El ensamblador","El ensamblador",'El ensamblador',
`El ensamblador´, `El ensamblador`,
'Vocales acentuadas "àèéìòóú", con diéresis "ïü" y símbolos
"ç€@#".'
`Vocales acentuadas "àèéìòóú", con diéresis "ïü" y símbolos
"ç€@#".`
```

Si queremos declarar una variable inicializada con un valor que se repite un conjunto de veces, podemos utilizar la directiva *times*.

```
cadena times 4 db 'PILA' ;define una variable inicializada
                         ;con el valor 'PILA' 4 veces
cadena2 db 'PILAPILAPILAPILA' ;es equivalente a la declaración
anterior
```

Vectores

Los vectores en ensamblador se definen con un nombre de variable e indicando a continuación los valores que forman el vector.

```
vector1 db 23, 42, -1, 65, 14 ;vector formado por 5 valores de tipo
                              ;byte
vector2 db 'a'b', 'c', 'de ;vector formado por 4 bytes,
                           ;inicializado usando caracteres
vector3 db 97, 98, 99, 100   ;es equivalente al vector anterior
vector4 dw 1000, 2000, -1000, 3000 ;vector formado por 4 palabras
```

También podemos utilizar la directiva *times* para inicializar vectores, pero entonces todas las posiciones tendrán el mismo valor.

```
vector5 times 5 dw 0 ;vector formado por 5 palabras inicializadas en 0
```

Valores y constantes numéricas

En ensamblador, todos los valores se almacenan como valores con signo expresados en complemento a 2.

Sección *.bss*, variables no inicializadas

Dentro de esta sección se declaran y se reserva espacio para las variables de nuestro programa para las cuales no queremos dar un valor inicial.

Hay que utilizar las directivas siguientes para declarar variables no inicializadas:

- `resb`: reserva espacio en unidades de byte
- `resw`: reserva espacio en unidades de palabra, 2 bytes
- `resd`: reserva espacio en unidades de doble palabra, 4 bytes
- `resq`: reserva espacio en unidades de cuádruple palabra, 8 bytes

El formato utilizado para definir una variable empleando cualquiera de las directivas anteriores es el mismo:

```
nombre_variable directiva multiplicidad
```

La multiplicidad es el número de veces que reservamos el espacio definido por el tipo de dato que determina la directiva.

Ejemplos

```
section .bss

var1 resb 1 ;reserva 1 byte
var2 resb 4 ;reserva 4 bytes
var3 resw 2 ;reserva 2 palabras = 4 bytes, equivalente al caso anterior
var3 resd 1 ;reserva una cuádruple palabra = 4 bytes
            ;equivalente a los dos casos anteriores
```

3.2.3. Definición de otros elementos

Otros elementos son:

1) extern. Declara un símbolo como externo. Lo utilizamos si queremos acceder a un símbolo que no se encuentra definido en el fichero que estamos ensamblando, sino en otro fichero de código fuente, en el que tendrá que estar definido y declarar con la directiva *global*.

En el proceso de ensamblaje, cualquier símbolo declarado como externo no generará ningún error; es durante el proceso de enlazamiento cuando, si no hay un fichero de código objeto en el que este símbolo esté definido, producirá error.

La directiva tiene el formato siguiente:

```
extern símbolo1, símbolo2, ..., símboloN
```

En una misma directiva *extern* se pueden declarar tantos símbolos como se quiera, separados por comas.

2) global. Es la directiva complementaria de *extern*. Permite hacer visible un símbolo definido en un fichero de código fuente en otros ficheros de código fuente; de esta manera, nos podremos referir a este símbolo en otros ficheros utilizando la directiva *extern*.

El símbolo ha de estar definido en el mismo fichero de código fuente donde se encuentre la directiva global.

Hay un uso especial de *global*: declarar una etiqueta que se debe denominar *main* para que el compilador de C (GCC) pueda determinar el punto de inicio de la ejecución del programa.

La directiva tiene el formato siguiente:

```
global símbolo1, símbolo2, ..., símboloN
```

En una misma directiva *global* se pueden declarar tantos símbolos como se quiera separados por comas.

Generación del ejecutable

Hay que hacer el ensamblaje de los dos códigos fuente ensamblador y enlazar los dos códigos objeto generados para obtener el ejecutable.

Ejemplo de utilización de las directivas *extern* y *global*

Prog1.asm

```
global printHola, msg ;hacemos visibles la subrutina
                       ;printHola y la variable msg.
section .data
   msg db "Hola!",10
   ...

section .text
printHola:
   mov rax,4
   mov rbx,1
   mov rcx,msg
   mov rdx,6
   int 80h
   ret
```

Prog2.asm

```
  extern printHola, msg ;indicamos que están declaradas
                      ;en otro fichero.
  global main ;hacemos visible la etiqueta main
                       ;para poder iniciar la ejecución.
  section. text
  main:
    call printHola ;ejecutamos el código del Prog1.asm

    mov rax,4     ;ejecutamos este código pero
    mov rbx,1     ;utilizamos la variable definida
    mov rcx,msg   ;en el fichero Fichero1.asm
    mov rdx,6
    int 80h

    mov rax,1
    mov rbx,0
    int 80h
```

3) section. Define una sección dentro del fichero de código fuente. Cada sección hará referencia a un segmento de memoria diferente dentro del espacio de memoria asignado al programa. Hay tres tipos de secciones:

a) `.data`: sección en la que se definen datos inicializados, datos a los que damos un valor inicial.

b) `.bss`: sección en la que se definen datos sin inicializar.

c) `.text`: sección en la que se incluyen las instrucciones del programa.

La utilización de estas directivas no es imprescindible, pero sí recomendable con el fin de definir los diferentes tipos de información que utilizaremos.

4) cpu. Esta directiva indica que solo se podrán utilizar los elementos compatibles con una arquitectura concreta. Solo podremos utilizar las instrucciones definidas en esta arquitectura.

Esta directiva se suele escribir al principio del fichero de código fuente, antes del inicio de cualquier sección del programa.

El formato de la directiva es:

```
cpu tipo_procesador
```

Ejemplo

cpu 386

cpu x86-64

3.3. Formato de las instrucciones

El otro elemento que forma parte de cualquier programa escrito en lenguaje de ensamblador son las instrucciones.

Las instrucciones en ensamblador tienen el formato general siguiente:

`[etiqueta:] instrucción [destino[, fuente]] [;comentario]`

donde *destino* y *fuente* representan los operandos de la instrucción.

Al final de la instrucción podemos añadir un comentario precedido del símbolo ;.

Los elementos entre [] son opcionales; por lo tanto, el único elemento imprescindible es el nombre de la instrucción.

3.3.1. Etiquetas

Una etiqueta hace referencia a un elemento dentro del programa ensamblador. Su función es facilitar al programador la tarea de hacer referencia a diferentes elementos del programa. Las etiquetas sirven para definir constantes, variables o posiciones del código y las utilizamos como operandos en las instrucciones o directivas del programa.

Para definir una etiqueta podemos utilizar números, letras y símbolos *_, $, #, @, ~, .* y *?*. Las etiquetas han de comenzar con un carácter alfabético o con los símbolos _, ., o ?, pero las etiquetas que empiezan con estos tres símbolos tienen un significado especial dentro de la sintaxis NASM y por este motivo se recomienda no utilizarlos al inicio de la etiqueta.

Una cuestión importante que hay que tener en cuenta es que en sintaxis NASM se distingue entre minúsculas y mayúsculas. Por ejemplo, las etiquetas siguientes serían diferentes: Etiqueta1, etiqueta1, ETIQUETA1, eTiQueTa1, ETIqueta1.

Ejemplo

```
Servicio equ 80h

section .data
    msg db "Hola!",10

section .text
    printHola:
    mov rax,4
    mov rbx,1
    mov rcx,msg
    mov rdx,6
    int Servicio
    jmp printHola
```

En este fragmento de código hay definidas tres etiquetas: *Servicio* define una constante, *msg* define una variable y *printHola* define una posición de código.

Las etiquetas para marcar posiciones de código se utilizan en las instrucciones de salto para indicar el lugar donde se debe saltar y también para definir la posición de inicio de una subrutina dentro del código fuente (podéis ver instrucciones de ruptura de secuencia en el subapartado 3.4.3).

Habitualmente, cuando se define una etiqueta para marcar una posición de código, se define con una secuencia de caracteres acabada con el carácter : (dos puntos). Pero cuando utilizamos esta etiqueta como operando, no escribiremos los dos puntos.

Ejemplo

```
          ...
Etiqueta1: mov rax,0
          ...
          jmp Etiqueta1 ;Esta instrucción salta a la
                        ;instrucción mov rax,0
          ...
```

Una etiqueta se puede escribir en la misma línea que ocupa una instrucción o en otra. El fragmento siguiente es equivalente al fragmento anterior:

```
    ...
Etiqueta1:

    mov rax,0
    ...
    jmp Etiqueta1 ;Esta instrucción salta a la instrucción mov
rax,0
    ...
```

3.4. Juego de instrucciones y modos de direccionamiento

Una instrucción en ensamblador está formada por un *código de operación* (el nombre de la instrucción) que determina qué debe hacer la instrucción, más un conjunto de operandos que expresan directamente un dato, un registro o una dirección de memoria; las diferentes maneras de expresar un operando en una instrucción y el procedimiento asociado que permite obtener el dato se denomina *modo de direccionamiento*.

Existen instrucciones que no tienen ningún operando y que trabajan implícitamente con registros o la memoria; hay instrucciones que tienen solo un operando y también las hay con dos o más operandos (en este caso los operandos se separan con comas):

1) Instrucciones sin ningún operando explícito: *código_operación*

Ejemplo

```
ret    ;retorno de subrutina
```

2) Instrucciones con un solo operando: *código_operación destino*

```
push rax    ;guarda el valor de rax en la pila; rax es un operando fuente
pop rax     ;carga el valor de la cima de la pila en rax; rax es un
            ; operando destino
call subr1  ;llama a la subrutina subr1, subr1 es un operando fuente
inc rax     ;rax=rax+1, rax hace de operando fuente y también
            de operando destino.
```

3) Instrucciones con dos operandos: *código_operación destino, fuente*

Ejemplos

```
mov rax [vec+rsi];mueve el valor de la posición del vector vec
                  ;indicada por rsi; es el operando fuente, en rax,
                  ;rax es el operando destino
add rax, 4        ;rax=rax+4
```

Operando fuente y operando destino

El operando fuente especifica un valor, un registro o una dirección de memoria donde hemos de ir para buscar un dato que necesitamos para ejecutar la instrucción.

El operando destino especifica un registro o una dirección de memoria donde hemos de guardar el dato que hemos obtenido al ejecutar la instrucción. El operando destino también puede actuar como operando fuente y el valor original se pierde cuando guardamos el dato obtenido al ejecutar la instrucción.

3.4.1. Tipos de operandos de las instrucciones x86-64

Operandos fuente y destino

En las instrucciones con un solo operando, este se puede comportar solo como operando fuente, solo como operando destino o como operando fuente y destino.

Ejemplos

```
push rax
```

El registro `rax` es un **operando fuente**; la instrucción almacena el valor del operando fuente en la pila del sistema, de manera que la pila es un operando destino implícito.

```
pop rax
```

El registro `rax` se comporta como **operando destino**; la instrucción almacena en `rax` el valor que se encuentra en la cima de la pila del sistema, de manera que la pila es un operando fuente implícito.

```
inc rax
```

El registro `rax` es a la vez **operando fuente y destino**; la instrucción `inc` realiza la operación `rax = rax + 1`, suma el valor original de `rax` con 1 y vuelve a guardar el resultado final en `rax`.

En las instrucciones con dos operandos, el primer operando se puede comportar como operando fuente y/o destino, mientras que el segundo operando se comporta siempre como operando fuente.

Ejemplos

```
mov rax, rbx
```

El primer operando se comporta solo como operando destino; la instrucción almacena el valor indicado por el segundo operando en el primer operando (`rax = rbx`).

```
add rax, 4
```

El primer operando se comporta al mismo tiempo como operando fuente y destino; la instrucción `add` lleva a cabo la operación `rax = rax + 4`, suma el valor original de `rax` con el valor 4, vuelve a almacenar el resultado en `rax`, y se pierde el valor que teníamos originalmente.

Localización de los operandos

Los operandos de las instrucciones se pueden encontrar en tres lugares diferentes: en la instrucción (valores inmediatos), en registros o a memoria.

Ved también

Para saber qué instrucciones permiten utilizar un tipo de operando determinado, ya sea como operando fuente u operando destino, hay que consultar la referencia de las instrucciones en el apartado 7 de este módulo.

1) Inmediatos. En las instrucciones de dos operandos, se puede utilizar un valor inmediato como operando fuente; algunas instrucciones de un operando también admiten un valor inmediato como operando. Los valores inmediatos se pueden expresar como valores numéricos (decimal, hexadecimal, octal o binario) o como caracteres o cadenas de caracteres. También se pueden utilizar las constantes definidas en el programa como valores inmediatos.

Para especificar un valor inmediato en una instrucción se utiliza la misma notación que la especificada en la definición de variables inicializadas.

Ejemplos

```
mov al, 10 b      ;un valor inmediato expresado en binario

cinco equ 5 h      ;se define una constante en hexadecimal
mov al, cinco      ;se utiliza el valor de la constante como
                        ;valor inmediato
mov eax, 0xABFE001C ;un valor inmediato expresado en hexadecimal.

mov ax, 'HI'       ;un valor inmediato expresado como una cadena
                       ;de caracteres
```

2) Registros. Los registros se pueden utilizar como operando fuente y como operando destino. Podemos utilizar registros de 64 bits, 32 bits, 16 bits y 8 bits. Algunas instrucciones pueden utilizar registros de manera implícita.

Ejemplos

```
mov al, 100
mov ax, 1000
mov eax, 100000
mov rax, 10000000
mov rbx, rax
mul bl   ;ax = al * bl,
los registros al y ax
son implícitos,
         ;al como operando
fuente y ax como
operando de destino.
```

3) Memoria. Las variables declaradas a memoria se pueden utilizar como operandos fuente y destino. En el caso **de instrucciones con dos operandos, solo uno de los operandos puede acceder a la memoria**, el otro ha de ser un registro o un valor inmediato (y este será el operando fuente).

Acceso a memoria

En sintaxis NASM se distingue el acceso al contenido de una variable de memoria del acceso a la dirección de una variable. Para acceder al contenido de una variable de memoria hay que especificar el nombre de la variable entre *[ ]*; si utilizamos el nombre de una variable sin *[ ]*, nos estaremos refiriendo a su dirección.

Ejemplo

```
section .data
     var1 dd 100      ;variable de 4 bytes (var1 = 100)

section .text
    mov rax, var1     ;se carga en rax la dirección de la variable var1
    mov rbx, [var1] ;se carga en rbx el contenido de var1, rbx=100
    mov rbx, [rax]   ;esta instrucción hará lo mismo que la anterior.
    mov [var1], rbx ;se carga en var1 el contenido del registro rbx
```

Debemos apuntar que, cuando especificamos un nombre de una variable sin los corchetes *[ ]*, no estamos haciendo un acceso a memoria, sino que la dirección de la variable está codificada en la propia instrucción y se considera un valor inmediato; por lo tanto podemos hacer lo siguiente:

```
;si consideramos que la dirección de memoria a la que hace referencia
var1
;es la dirección 12345678h
mov QWORD [var2], var1 ;se carga en var2 la dirección de var1
mov QWORD [var2],12345678h ;es equivalente a la instrucción anterior.
```

`[var2]` es el operando que hace el acceso a memoria y `var1` se codifica como un valor inmediato.

Tamaño de los operandos

En sintaxis NASM el tamaño de los datos especificados por un operando puede ser de byte, word, double word y quad word. Cabe recordar que lo hace en formato *little-endian*.

- BYTE: indica que el tamaño del operando es de un byte (8 bits).
- WORD: indica que el tamaño del operando es de una palabra (word) o dos bytes (16 bits).
- DWORD: indica que el tamaño del operando es de una doble palabra (double word) o cuatro bytes (32 bits).
- QWORD: indica que el tamaño del operando es de una cuádruple palabra (quad word) u ocho bytes (64 bits).

Debemos tener en cuenta el tamaño de los operandos que estamos utilizando en cada momento, sobre todo cuando hacemos referencias a memoria, y especialmente cuando esta es el operando destino, ya que utilizaremos la parte de la memoria necesaria para almacenar el operando fuente según el tamaño que tenga.

En algunos casos es obligatorio especificar el tamaño del operando; esto se lleva a cabo utilizando los modificadores BYTE, WORD, DWORD y QWORD ante la referencia a memoria. La función del modificador es utilizar tantos bytes como indica a partir de la dirección de memoria especificada, independientemente del tamaño del tipo de dato (`db`, `dw`, `dd`, `dq`) que hayamos utilizado a la hora de definir la variable.

En los casos en los que no es obligatorio especificar el modificador, se puede utilizar para facilitar la lectura del código o modificar un acceso a memoria.

Los casos en los que se debe especificar obligatoriamente el tamaño de los operandos son los siguientes:

1) Instrucciones de dos operandos en los que el primer operando es una posición de memoria y el segundo operando es uno inmediato; es necesario indicar el tamaño del operando.

Ejemplo

```
.data
var1 dd 100              ;variable definida de 4 bytes
. text
mov DWORD [var1], 0ABCh  ;se indica que la variable var1
                         ;es de 32 bits
```

La instrucción siguiente es incorrecta:

```
mov [var1], 0ABCh
```

En el caso anterior, el compilador no sería capaz de averiguar cómo ha de copiar este valor a la variable `[var1]`, aunque cuando lo hemos definido se haya especificado el tipo.

2) Instrucciones de un operando en las que el operando sea una posición de memoria.

Ejemplo

```
inc QWORD [var1]  ;se indica que la posición de memoria afectada es
                  ;de tamaño de un byte. [var1]=[var1]+1
push WORD [var1]  ;se indica que ponemos 2 bytes en la pila.
```

3.4.2. Modos de direccionamiento

Cuando un operando se encuentra en la memoria, es necesario considerar qué modo de direccionamiento utilizamos para expresar un operando en una instrucción para definir la manera de acceder a un dato concreto. A continuación, veremos los modos de direccionamiento que podemos utilizar en un programa ensamblador:

1) Inmediato. En este caso, el operando hace referencia a un dato que se encuentra en la instrucción misma. No hay que hacer ningún acceso extra a memoria para obtenerlo. Solo podemos utilizar un direccionamiento inmediato como operando fuente. El número especificado ha de ser un valor que se pueda expresar con 32 bits como máximo, que será el resultado de evaluar una expresión aritmética formada por valores numéricos y operadores aritméticos y también sumar una dirección de memoria representada mediante una etiqueta (nombre de una variable), con la excepción de la instrucción *mov* cuando el primer operando es un registro de 64 bits, para el que podremos especificar un valor que se podrá expresar con 64 bits.

Ejemplos

```
mov rax, 0102030405060708h   ;el segundo operando utiliza direccionamiento inmediato
                             ;expresado con 64 bits.
mov QWORD [var], 100         ;el segundo operando utiliza direccionamiento inmediato
                             ;carga el valor 100 y se guarda en var
mov rbx, var                 ;el segundo operando utiliza direccionamiento inmediato
                             ;carga la dirección de var en el registro rbx
mov rbx, var+16 * 2          ;el segundo operando utiliza direccionamiento inmediato
                             ;carga la dirección de var+32 en el registro rbx
```

Para especificar un valor inmediato en una instrucción se utiliza la misma notación que la especificada en la definición de variables inicializadas.

2) Directo a registro. En este caso, el operando hace referencia a un dato que se encuentra almacenado en un registro. En este modo de direccionamiento podemos especificar cualquier registro de propósito general (registros de datos, registros índice y registros apuntadores).

Ejemplo

```
mov rax, rbx     ;los dos operandos utilizan direccionamiento
                 ;directo a registro, rax = rbx
```

3) Directo a memoria. En este caso, el operando hace referencia a un dato que se encuentra almacenado en una posición de memoria. El operando habrá de especificar el nombre de una variable de memoria entre corchetes *[ ]*; cabe recordar que en sintaxis NASM se interpreta el nombre de una variable sin corchetes como la dirección de la variable y no como el contenido de la variable.

Ejemplos

```
mov rax,[var]      ;el segundo operando utiliza direccionamiento
                   ;directo a memoria, rax = [var]
mov [suma],rcx     ;el primer operando utiliza direccionamiento
                   ;directo a memoria [suma]=[suma]+rcx
```

4) Indirecto a registro. En este caso, el operando hace referencia a un dato que se encuentra almacenado en una posición de memoria. El operando habrá de especificar un registro entre corchetes *[ ]*; el registro contendrá la dirección de memoria a la cual queremos acceder.

Ejemplos

```
mov rbx, var   ;se carga en rbx la dirección de la variable var
mov rax, [rbx] ;el segundo operando utiliza la dirección que tenemos
en rbx
               ;para acceder a memoria, se mueven 4 bytes a partir de
              ;la dirección especificada por rbx y se guardan en rax.
```

5) Indexado. En este caso, el operando hace referencia a un dato que se encuentra almacenado en una posición de memoria. Digamos que un operando utiliza direccionamiento indexado si especifica una dirección de memoria como dirección base que puede ser expresada mediante un número o el nombre de una variable que tengamos definida, sumada a un registro que actúa como índice respecto a esta dirección de memoria entre corchetes *[ ]*.

Ejemplos

```
mov rax, [vector+rsi] ;vector contiene la dirección base, rsi actúa
                      ;como registro índice
add [1234h+r9],rax    ;1234h es la dirección base, r9 actúa
                      ;como registro índice.
```

6) Relativo. En este caso, el operando hace referencia a un dato que se encuentra almacenado en una posición de memoria. Digamos que un operando utiliza direccionamiento relativo cuando especifica un registro sumado a un número entre corchetes *[ ]*. El registro contendrá una dirección de memoria que actuará como dirección base y el número como un desplazamiento respecto a esta dirección.

```
mov rbx, var       ;se carga en rbx la dirección de la variable var
mov rax, [rbx+4]   ;el segundo operando utiliza direccionamiento relativo
                   ;4 es el desplazamiento respecto a esta dirección
mov [rbx+16], rcx  ;rbx contiene la dirección base, 16 es el
                   ;desplazamiento respecto a esta dirección.
```

a) Combinaciones del direccionamiento indexado y relativo. La sintaxis NASM nos permite especificar de otras maneras un operando para acceder a memoria; el formato del operando que tendremos que expresar entre corchetes *[ ]* es el siguiente:

```
[Registro Base + Registro Index * escala + desplazamiento]
```

El registro base y el registro índice pueden ser cualquier registro de propósito general, la escala puede ser 1, 2, 4 u 8 y el desplazamiento ha de ser un número representable con 32 bits que será el resultado de evaluar una expresión aritmética formada por valores numéricos y operadores aritméticos; también podemos sumar una dirección de memoria representada mediante una etiqueta (nombre de una variable). Podemos especificar solo los elementos que nos sean necesarios.

Ejemplos

```
[rbx + rsi * 2 + 4 * (12+127)]
[r9 + r10 * 8]
[r9 - 124 * 8]
[rax * 4 + 12 / 4]
[vec+16 + rsi * 2]
```

Esta es la manera general de expresar un operando para acceder a memoria; los otros modos especificados anteriormente son casos concretos en los que solo se definen algunos de estos elementos.

7) Relativo a PC. En el modo de 64 bits se permite utilizar direccionamiento relativo a PC en cualquier instrucción; en otros modos de operación, el direccionamiento relativo a PC se reserva exclusivamente para las instrucciones de salto condicional.

Este modo de direccionamiento es equivalente a un direccionamiento relativo a registro base en el que el registro base es el registro contador de programa (PC). En la arquitectura x86-64 este registro se denomina *rip*, y el desplazamiento es el valor que sumaremos al contador de programa para determinar la dirección de memoria a la que queremos acceder.

Utilizaremos este modo de direccionamiento habitualmente en las instrucciones de salto condicional. En estas instrucciones especificaremos una etiqueta que representará el punto del código al que queremos saltar. La utilización del registro contador de programa es implícita, pero para emplear este modo de direccionamiento, se ha de codificar el desplazamiento respecto al contador de programa; el cálculo para obtener este desplazamiento a partir de la dirección representada por la etiqueta se resuelve durante el ensamblaje y es transparente para el programador.

Ejemplo

```
je etiqueta1
```

8) Direccionamiento a pila. Es un direccionamiento implícito; se trabaja implícitamente con la cima de la pila, mediante el registro apuntador a pila (*stack pointer*); en la arquitectura x86-64 este registro se llama *rsp*. En la pila solo podremos almacenar valores de 16 bits y de 64 bits.

Solo existen dos instrucciones específicas diseñadas para trabajar con la pila:

```
push fuente   ;coloca un dato en la pila
pop destino   ;extrae un dato de la pila
```

Para expresar el operando fuente o destino podemos utilizar cualquier tipo de direccionamiento descrito anteriormente, pero lo más habitual es utilizar el direccionamiento a registro.

Ejemplos

```
push word 23h
push qword [rax]
pop qword [var]
pop bx
```

3.4.3. Tipos de instrucciones

El juego de instrucciones de los procesadores x86-64 es muy amplio. Podemos organizar las instrucciones según los tipos siguientes:

Ved también

En el apartado 6 de este módulo se describe con detalle el formato y la utilización de las instrucciones de los procesadores x86-64 que consideramos más importantes.

1) Instrucciones de transferencia de datos:

- **mov** ***destino, fuente***: instrucción genérica para mover un dato desde un origen a un destino.
- **push** ***fuente***: instrucción que mueve el operando de la instrucción a la cima de la pila.
- **pop** ***destino***: mueve el dato que se encuentra en la cima de la pila al operando destino.
- **xchg** ***destino, fuente***: intercambia contenidos de los operandos.

2) Instrucciones aritméticas y de comparación:

- **add** ***destino, fuente***: suma aritmética de los dos operandos.
- **adc** ***destino, fuente***: suma aritmética de los dos operandos considerando el bit de transporte.
- **sub** ***destino, fuente***: resta aritmética de los dos operandos.
- **sbb** ***destino, fuente***: resta aritmética de los dos operandos considerando el bit de transporte.
- **inc** ***destino***: incrementa el operando en una unidad.
- **dec** ***destino***: decrementa el operando en una unidad.
- **mul** ***fuente***: multiplicación entera sin signo.
- **imul** ***fuente***: multiplicación entera con signo.
- **div** ***fuente***: división entera sin signo.
- **idiv** ***fuente***: división entera con signo.
- **neg** ***destino***: negación aritmética en complemento a 2.
- **cmp** ***destino, fuente***: comparación de los dos operandos; hace una resta sin guardar el resultado.

3) Instrucciones lógicas y de desplazamiento:

a) Operaciones lógicas:

- **and** ***destino, fuente***: operación 'y' lógica.
- **or** ***destino, fuente***: operación 'o' lógica.
- **xor** ***destino, fuente***: operación `o exclusiva´ lógica.
- **not** ***destino***: negación lógica bit a bit.

- **test** ***destino, fuente***: comparación lógica de los dos operandos; hace una 'y' lógica sin guardar el resultado.

b) Operaciones de desplazamiento:

- **sal** ***destino, fuente*** / **shl** ***destino, fuente***: desplazamiento aritmético/lógico a la izquierda.
- **sar** ***destino, fuente***: desplazamiento aritmético a la derecha.
- **shr** ***destino, fuente***: desplazamiento lógico a la derecha.
- **rol** ***destino, fuente***: rotación lógica a la izquierda.
- **ror** ***destino, fuente***: rotación lógica a la derecha.
- **rcl** ***destino, fuente***: rotación lógica a la izquierda considerando el bit de transporte.
- **rcr** ***destino, fuente***: rotación lógica a la derecha considerando el bit de transporte.

4) Instrucciones de ruptura de secuencia:

a) Salto incondicional:

- **jmp etiqueta**: salta de manera incondicional a la etiqueta.

b) Saltos que consultan un bit de resultado concreto:

- **je** ***etiqueta*** / **jz** ***etiqueta***: salta a la etiqueta si igual, si el bit de cero es activo.
- **jne** ***etiqueta*** / **jnz** ***etiqueta***: salta a la etiqueta si diferente, si el bit de cero no es activo.
- **jc** ***etiqueta*** / **jnc** ***etiqueta***: salta a la etiqueta si el bit de transporte es activo.
- **jnc** ***etiqueta***: salta a la etiqueta si el bit de transporte no es activo.
- **jo** ***etiqueta***: salta a la etiqueta si el bit de desbordamiento es activo.
- **jno** ***etiqueta***: salta a la etiqueta si el bit de desbordamiento no es activo.
- **js** ***etiqueta***: salta a la etiqueta si el bit de signo es activo.
- **jns** ***etiqueta***: salta a la etiqueta si el bit de signo no es activo.

c) Saltos condicionales sin considerar el signo:

- **jb** ***etiqueta*** / **jnae** ***etiqueta***: salta a la etiqueta si es más pequeño.
- **jbe** ***etiqueta*** / **jna** ***etiqueta***: salta a la etiqueta si es más pequeño o igual.
- **ja** ***etiqueta*** / **jnbe** ***etiqueta***: salta a la etiqueta si es mayor.
- **jae** ***etiqueta*** / **jnb** ***etiqueta***: salta a la etiqueta si es mayor o igual.

d) Saltos condicionales considerando el signo:

- **jl** ***etiqueta*** / **jnge** ***etiqueta***: salta si es más pequeño.
- **jle** ***etiqueta*** / **jng** ***etiqueta***: salta si es más pequeño o igual.
- **jg** ***etiqueta*** / **jnle** ***etiqueta***: salta si es mayor.

- **jge *etiqueta*** / **jnl *etiqueta***: salta si es mayor o igual.

e) Otras instrucciones de ruptura de secuencia:

- **loop *etiqueta***: decrementa el registro rcx y salta si rcx es diferente de cero.
- **call *etiqueta***: llamada a subrutina.
- **ret**: retorno de subrutina.
- **iret**: retorno de rutina de servicio de interrupción (RSI).
- **int *servicio***: llamada al sistema operativo.

5) Instrucciones de entrada/salida:

- **in *destino, fuente***: lectura del puerto de E/S especificado en el operando fuente y se guarda en el operando destino.
- **out *destino, fuente***: escritura del valor especificado por el operando fuente en el puerto de E/S especificado en el operando destino.

4. Introducción al lenguaje C

En este apartado se hace una breve introducción al lenguaje C. No pretende ser un manual de programación ni una guía de referencia del lenguaje, solo explica los conceptos necesarios para poder desarrollar pequeños programas en C y poder hacer llamadas a funciones implementadas en lenguaje de ensamblador, con el objetivo de entender el funcionamiento del procesador y la comunicación entre un lenguaje de alto nivel y uno de bajo nivel.

4.1. Estructura de un programa en C

Un programa escrito en lenguaje C sigue en general la estructura siguiente:

- Directivas de compilación.
- Definición de variables globales.
- Declaración e implementación de funciones.
- Declaración e implementación de la función *main*.

La única parte imprescindible en todo programa escrito en C es la función *main*, aunque muchas veces hay que incluir algunas directivas de compilación.

Por ejemplo, es habitual utilizar la directiva *include* para incluir otros ficheros en los que se definen funciones de la biblioteca estándar *glibc* que se quieren utilizar en el programa.

Veamos a continuación un primer programa escrito en C:

```
1 /*Fichero hola.c*/
2 #include <stdio.h>
3 int main(){
4 printf ("Hola!\n");
5 return 0;
6 }
```

La primera línea define un comentario; en C los comentarios se escriben entre los símbolos `/*` y `*/`; también se puede utilizar `//` para escribir comentarios de una sola línea.

Ejemplos de comentarios en C

```
//Esto es un comentario de una línea
/* Esto es un
comentario
de varias líneas */
```

La segunda línea corresponde a una directiva; en lenguaje C, las directivas empiezan siempre por el símbolo *#*.

La directiva *include* indica al compilador que incluya el fichero indicado, *stdio.h*, al compilar el programa.

El fichero *stdio.h* incluye la definición de las funciones más habituales para trabajar con la pantalla y el teclado.

La tercera línea declara la función *main*, función principal de todo programa escrito en C en que se iniciará la ejecución del programa; se marca el inicio de la función con el símbolo {.

La cuarta línea es la primera instrucción del *main*, y corresponde a una llamada a la función *printf* de la biblioteca estándar; esta función está definida en el fichero *stdio.h*. La función *printf* permite escribir en la pantalla; en este caso se escribe la cadena de caracteres indicada. En lenguaje C debemos finalizar cada instrucción con un punto y coma (*;*).

La quinta línea define cuál es el valor que devolverá la función. Como el tipo de retorno es un número entero (*int*), se ha de especificar un valor entero.

Finalmente, en la sexta línea, se cierra el código de la función con el símbolo *}*. El código de una función en C siempre se ha de cerrar entre los símbolos *{* y *}*.

4.1.1. Generación de un programa ejecutable

Para generar un programa ejecutable a partir de un fichero de código fuente C, utilizamos el compilador GCC.

Para compilar el programa anterior, hay que ejecutar la orden siguiente:

```
$ gcc hola.c -o hola -g
```

Para ejecutar el programa solo debemos utilizar el nombre del fichero de salida generado al añadir *./* delante:

```
$ ./hola
Hola!
$ _
```

4.2. Elementos de un programa en C

4.2.1. Directivas

En C existe un conjunto de directivas de compilación muy amplio. A continuación, se describen solo las directivas que utilizaremos. Las directivas empiezan siempre por el símbolo #.

1) ***include***: permite incluir otro fichero, de manera que cuando se llame el compilador, aquel sea compilado junto con el código fuente.

Lo más habitual es incluir los denominados *ficheros de cabecera* (*header*), ficheros con extensión *.h* con las definiciones de las funciones incluidas en las bibliotecas estándar de C, de manera que puedan ser utilizadas en el programa fuente.

El formato de la directiva *include* es:

```
#include <nombre_fichero>
```

Ejemplo

```
#include <stdio.h>
```

2) ***define***: permite definir valores constantes para ser utilizados en el programa. Los valores de las constantes se pueden expresar de maneras diferentes: como cadenas de caracteres o como valores numéricos expresados en binario, octal, decimal o hexadecimal.

El formato de la directiva *define* es:

```
#define nombre_constante valor
```

Ejemplo

```
#define CADENA "Hola"
#define NUMBIN 01110101b
#define NUMOCT 014q
#define NUMDEC 12
#define NUMHEX 0x0C
```

3) ***extern***: declara un símbolo como externo. Lo utilizamos si queremos acceder a un símbolo que no se encuentra definido en el fichero que estamos compilando, sino en otro fichero de código fuente C o código objeto.

En el proceso de compilación, cualquier símbolo declarado como externo no generará ningún error; sin embargo, durante el proceso de enlazamiento, si no existe un fichero de código objeto en el que este símbolo esté definido, producirá error.

La directiva tiene el formato siguiente:

```
extern [tipo_del_retorno] nombre_de_función ([lista_de_tipos]);
extern [tipo] nombre_variable [= valor_inicial];
```

En una misma directiva *extern* se pueden declarar tantos símbolos como se quiera, separados por comas.

Por ejemplo:

```
extern int y;
extern printVariable(int);
```

4.2.2. Variables

En general, las variables se pueden definir de dos maneras:

Identificadores

Los identificadores son los nombres que damos a las variables, constantes, funciones, etiquetas y otros objetos.

En C los identificadores han de empezar por una letra o el símbolo _.

También cabe tener presente que en C se distingue entre mayúsculas y minúsculas, por lo tanto, las variables siguientes son diferentes: VARX, varx, varX, VarX.

1) Globales: accesibles desde cualquier punto del programa.

2) Locales: accesibles solo dentro de un fragmento del programa.

La manera más habitual de definir variables locales es al principio de una función, de modo que solo existen durante la ejecución de una función y solo son accesibles dentro de esta.

Ejemplo

```
int x=0;                                    //x: variable global
int main(){
    int y=1;                                //y: variable local de la función main
     for (y=0; y>3; y++) {int z = y*2; }    //z: variable local del for
}
```

Para definir una variable se ha de especificar primero el tipo de dato de la variable seguido del nombre de la variable; opcionalmente se puede especificar a continuación el valor inicial.

El formato general para definir una variable en C es el siguiente:

```
tipo nombre_variable [= valor_inicial];
```

Los tipos de datos más habituales para definir variables son:

- caracteres (*char*),
- enteros (*int*) y (*long*) y
- números reales (*float*) y (*double*).

Ejemplo

```
/*Una variable de tipo carácter inicializada con el valor 'A'*/
char c='A';
/*Dos variables de tipo entero, una inicializada y otra sin
inicializar*/
int x=0, y;
/*Un número real, expresado en punto fijo*/
float pi=3.1416;
```

4.2.3. Operadores

Los operadores de C permiten realizar operaciones aritméticas, lógicas, relacionales (de comparación) y operaciones lógicas bit a bit. Cualquier combinación válida de operadores, variables y constantes se denomina **expresión**.

Cuando se define una expresión, podemos utilizar paréntesis para clarificar la manera de evaluar y cambiar la prioridad de los operadores.

Prioridad de los operadores

De más prioridad a menos prioridad:

```
! , ~
++, --
*, /, %
+, -
<<, >>
<, <=, >, >=
==, !=
&
^
|
&&
||
=
```

Podemos clasificar los operadores de la siguiente manera:

1) Operador de asignación:

Igualar dos expresiones: =

Ejemplos

```
a = 3;
c = a;
```

2) Operadores aritméticos:

suma: +
resta y negación: –
incremento: ++
decremento: – –
producto: *
división: /
resto de la división: %

Ejemplos

```
a = b + c;
x = y * z
```

3) Operadores de comparación relacionales:

igualdad: = =
diferente: !=
menor: <
menor o igual: <=
mayor: >
mayor o igual: >=

4) Operadores de comparación lógicos:

Y lógica (AND): &&
O lógica (OR): ||
Negación lógica: !

Ejemplos

```
(a = = b) && (c != 3)
(a <= b) || !(c > 10)
```

5) Operadores lógicos:

Operación lógica que se hace bit a bit.

OR (O): |
AND (Y): &
XOR (O exclusiva): ^
Negación lógica: ~
Desplazamiento a la derecha: >>
Desplazamiento a la izquierda: <<

Ejemplos

```
z = x | y;
z = x & y;
z = x ^ y;
z = x >> 2; // desplazar los bits de la variable x 2 posiciones
            // a la derecha y guardar el resultado en z
z = ~x; //z es el complemento a 1 de x
z = -x; //z es el complemento a 2 de x
```

4.2.4. Control de flujo

A continuación, se explican las estructuras de control de flujo más habituales de C:

1) Sentencias y bloques de sentencias. Una sentencia es una línea de código que finalizamos con un punto y coma (;). Podemos agrupar sentencias formando un bloque de sentencias utilizando las llaves ({}).

2) Sentencias condicionales

a) if. Es la sentencia condicional más simple; permite especificar una condición y el conjunto de sentencias que se ejecutarán en caso de que se cumpla la condición:

```
if (condición) {
      bloque de sentencias
}
```

Si solo hemos de ejecutar una sentencia, no es necesario utilizar las llaves:

```
if (condición) sentencia;
```

La condición será una expresión cuyo resultado de la evaluación sea 0 (falsa) o diferente de cero (cierta).

Ejemplos

```
if (a > b) {
    printf("a es mayor que b\n");
    a--;
}

if (a >= b) b++;
```

b) if-else. Permite añadir un conjunto de instrucciones que se ejecutarán en caso de que no se cumpla la condición:

```
if (condición) {
    bloque de sentencias
}
else {
    bloque de sentencias alternativas
}
```

Ejemplo

```
if (a >b) {
    printf("a es mayor que b\n");
    a--;

}
else
    printf(("a no es mayor que b\n");
```

c) if-else-if. Se pueden enlazar estructuras de tipo *if* añadiendo nuevas sentencias *if* a continuación.

```
if (condición 1) {
      bloque de sentencias que se ejecutan
      si se cumple la condición 1
}
else if (condición 2) {
      bloque de sentencias que se ejecutan
      si se cumple la condición 2 y
      no se cumple la condición 1
      }
      else {
         bloque de sentencias que se ejecutan
       si no se cumple ninguna de las condiciones
}
```

Ejemplo

```
if (a > b) {
    printf("a es mayor que b\n");
}
else if (a < b) {
        printf("a es menor que b\n");
     }
     else {
        printf("a es igual que b\n");
}
```

3) Estructuras iterativas

a) for. Permite hacer un bucle controlado por una o más variables que van desde un valor inicial hasta un valor final y que son actualizadas a cada iteración. El formato es:

```
for (valores iniciales; condiciones; actualización) {
      bloque de sentencias que ejecutar
}
```

Si solo hemos de ejecutar una sentencia, no hay que utilizar las llaves.

```
for (valores iniciales; condiciones; actualización) sentencia;
```

Ejemplo

```
//bucle que se ejecuta 5 veces, mientras x<5, cada vez se
incrementa x en 1
int x;
for (x = 0; x < 5 ; x++) {
     printf("el valor de x es: %d\n", x);
}
```

b) while. Permite expresar un bucle que se va repitiendo mientras se cumpla la condición indicada. Primero se comprueba la condición y si esta se cumple, se ejecutan las sentencias indicadas:

```
while (condiciones) {
     bloque de sentencias que ejecutar
}
```

Si solo hemos de ejecutar una sentencia no hay que utilizar las llaves.

```
while (condiciones) sentencia;
```

Ejemplo

```
//bucle que se ejecuta 5 veces, cabe definir previamente la variable x
int x = 0;
while (x < 5) {
   printf("el valor de x es: %d\n", x);
   x++; //necesario para salir del bucle
}
```

c) do-while. Permite expresar un bucle que se va repitiendo mientras se cumpla la condición indicada. Primero se ejecutan las sentencias indicadas y después se comprueba la condición; por lo tanto, como mínimo se ejecutan una vez las sentencias del bucle:

```
do {
   bloque de sentencias a ejecutar
}while (condiciones);
```

Si solo hemos de ejecutar una sentencia, no hay que utilizar las llaves:

```
do sentencia;
while (condiciones);
```

Ejemplo

```
//bucle que se ejecuta 5 veces, cabe definir previamente la variable x
int x = 0;
do {
   printf("el valor de x es: %d\n", x);
   x++;
} while (x < 5);
```

4.2.5. Vectores

Los vectores en lenguaje C se definen utilizando un tipo de dato base junto con el número de elementos del vector indicado entre los símbolos *[ ]*.

El formato para un vector unidimensional es el siguiente:

```
tipo nombre_vector [tamaño];
```

El formato para un vector bidimensional o matriz es el siguiente:

```
tipo nombre_vector [filas][columnas];
```

Siguiendo este mismo formato, se podrían definir vectores de más de dos dimensiones.

Al declarar un vector, se reserva el espacio de memoria necesario para poder almacenar el número de elementos del vector según el tipo.

Ejemplo

```
int vector[5]; // vector de 5 enteros
char cadena[4]; // vector de 4 caracteres
int matriz[3][4]; // matriz de 3 filas y 4 columnas: 12 enteros
```

También se pueden definir vectores dando un conjunto de valores iniciales; en este caso no es necesario indicar el número de elementos, si se hace no se podrá dar un conjunto de valores iniciales superior al valor indicado:

```
int vector[]={1, 2, 3, 4, 5};          // vector de 5 enteros
char cadena[4]={'H', 'o', 'l', 'a'};   // vector de 4 caracteres
int vector2[3]={1, 2};                 // vector de 3 enteros con las dos
                                       // primeras posiciones inicializadas
int vector2[3]={1, 2, 3, 4};           // declaración incorrecta
    int matriz[][]={{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}};
                                       // matriz de enteros de 3 filas y 4 columnas
```

Los vectores de tipo *char* también se pueden inicializar con una cadena de caracteres entre comillas dobles. Por ejemplo:

```
char cadena[]="Hola";
```

Para acceder a una posición del vector, debemos indicar el índice entre corchetes ([ ]). Cabe tener presente que el primer elemento de un vector se encuentra en la posición 0; por lo tanto, los índices irán desde 0 hasta el número de elementos menos uno.

Ejemplo

```
int vector[3]={1, 2, 3}; //índices válidos: 0, 1 y 2
                         //lo que equivale a hacer lo siguiente:
   int vector[3];
   vector[0]=1;
   vector[1]=2;
   vector[2]=3;
                         // o también se puede hacer lo siguiente:
int i;
for (i=0; i<3; i++) vector[i]=i+1;
                         // Si hacemos lo siguiente, no se producirá error, pero realmente
                         // estaremos accediendo fuera del vector, lo que puede ocasionar
                         // múltiples problemas:
   vector[3]=4;
   vector[20]=300;
```

4.2.6. Apuntadores

Un aspecto muy importante del lenguaje C es que permite acceder directamente a direcciones de memoria. Esto se consigue utilizando un tipo de variable que se conoce como *apuntador a memoria*. Un apuntador es una variable que contiene una dirección de memoria.

Un apuntador se define indicando el tipo de dato al cual apunta y añadiendo el símbolo ***** ante el nombre del apuntador.

Cabe tener presente que al definir un apuntador solo se reserva espacio para almacenar una dirección de memoria; no se reserva espacio para poder almacenar un valor de un tipo determinado.

Símbolo * en C

El símbolo * tiene múltiples funciones: definir un apuntador, acceder al contenido de una posición de memoria y también se corresponde con el operador aritmético de multiplicación.

El formato general para definir un apuntador en C es el siguiente:

```
tipo *nombre_apuntador;
```

Utilizamos el operador & para obtener la dirección de memoria donde se encuentra almacenada una variable, no su contenido. Esta dirección la podemos asignar a un apuntador.

Ejemplo

```
int   *p1;    // Se define un apuntador de tipo int
char *p2;     // Se define un apuntador de tipo char

int x=1, y;   // se definen dos variables de tipo int
char c='a';   // se define una variables de tipo char

p1=&x;        // se asigna a p1 la dirección de la variable x
p2=&c;        // se asigna a p2 la dirección de la variable c
y=*p1;        // como p1 apunta a x, es equivalente a y = x.
```

Cuando trabajamos con vectores, el nombre del vector en realidad es un apuntador, una constante que contiene la dirección del primer elemento del vector.

```
   int vector[3]={1, 2, 3};
                    //lo que es equivalente a hacer lo siguiente:
   int vector[3];
   int *v;
   int i;
   v = vector;      //vector i *v son apuntadores a entero.
for (i=0; i<3; i++) *(v+i)=i+1;
                    // *(v+i): indica que estamos accediendo al contenido
                    // de la dirección 'v+i'
```

4.2.7. Funciones

Una función es un bloque de código que lleva a cabo una tarea concreta. Aparte de la función *main*, un programa puede definir e implementar otras funciones.

El formato general de una función es el siguiente:

```
tipo_del_retorno nombre_función(lista_de_parámetros){
    definición de variables;
    sentencias;
    return valor;
}
```

donde:

- `tipo_del_retorno`: tipo del dato que devolverá la función; si no se especifica el tipo de retorno, por defecto es de tipo entero (*int*).
- `lista_de_parámetros`: lista de tipos y nombres de variables separados por comas; no es obligatorio que la función tenga parámetros.
- `return`: finaliza la ejecución de la función y devuelve un valor del tipo especificado en el campo `tipo_del_retorno`. Si no se utiliza esta sentencia o no se especifica un valor, se devolverá un valor indeterminado.

Ejemplo

Definamos una función que sume dos enteros y devuelva el resultado de la suma:

```
int funcioSuma(int a, int b){
    int resultado;     //variable local

    resultado = a + b; //sentencia de la función
    return resultado;  //valor de retorno
}
```

Se muestra a continuación cómo quedaría un programa completo que utilizara esta función.

```
// fichero suma.c
#include <stdio.h>

int funcioSuma(int a, int b){
    int resultado;     //variable local

    resultado = a + b; //sentencia de la función
    return resultado;  //valor de retorno
}

int main(){
  int x, y, r;         //variables locales
  printf ("\nIntroduce el valor de x: ");
  scanf("%d",&x);
  printf ("Introduce el valor de y: ");
  scanf("%d",&y);
  r=funcioSuma(x,y);   //llamamos a la función que hemos definido
  printf("La suma de x e y es: %d\n", r);
}
```

Compilación y ejecución

```
$ gcc -o suma suma.c
$ ./suma
Introduce el valor de x: 3
Introduce el valor de y: 5
La suma de x e y es: 8
$ _
```

4.2.8. Funciones de E/S

Se describen a continuación las funciones básicas de E/S, para escribir en pantalla y para leer por teclado y que están definidas en el fichero *stdio.h*:

Nota

El repertorio de funciones de E/S es muy extenso, podréis encontrar mucha información de todas estas funciones en libros de programación en C y en Internet.

Función *printf*

printf permite escribir en pantalla información formateada, permite visualizar cadenas de caracteres constantes, junto con el valor de variables.

El formato general de la función es el siguiente:

```
printf("cadena de control"[, lista_de_parámetros])
```

La cadena de control incluye dos tipos de elementos:

- Los caracteres que queremos mostrar por pantalla.
- Órdenes de formato que indican cómo se mostrarán los parámetros.

La lista de parámetros está formada por un conjunto de elementos que pueden ser expresiones, constantes y variables, separadas por comas.

Debe existir el mismo número de órdenes de formato que de parámetros, se deben corresponder en orden y el tipo de la orden con el tipo del dato.

Las órdenes de formato empiezan con un símbolo % seguido de uno o más caracteres. Las más habituales son:

`%d`	Para mostrar un valor entero, el valor de una variable de tipo *char* o *int*.
`%ld`	Para mostrar un valor entero largo, el valor de una variable de tipo *long*.
`%c`	Para mostrar un carácter, el contenido de una variable de tipo *char*.

`%s`	Para mostrar una cadena de caracteres, el contenido de un vector de tipo *char*.
`%f`	Para mostrar un número real, el valor de una variable de tipo *float* o *double*. Se puede indicar el número de dígitos de la parte entera y de la parte decimal, utilizando la expresión siguiente: `%[dígitos enteros].[dígitos decimales]f`
`%p`	Para mostrar una dirección de memoria, por ejemplo el valor de un apuntador.

Dentro de la cadena de control se pueden incluir algunos caracteres especiales, empezando con un símbolo \ y seguido de un carácter o más. Los más habituales son:

`\n`	carácter de salto de línea
`\t`	carácter de tabulación

Si solo se quiere mostrar una cadena de caracteres constantes, no es necesario especificar ningún parámetro.

Ejemplos

```
int x=5;
float pi=3.1416;
char msg[]="Hola", c='A';
printf("Hola!" );
printf ("Valor de x: %d. Dirección de x: %p\n", x, &x);
printf ("El valor de PI con 1 entero y 2 decimales: %1.2f\n", pi);
printf ("Contenido de msg: %s. Dirección de msg: %p\n", msg, msg);
printf ("Valor de c: %c. El código ASCII guardado en c: %d\n", c, c);
printf ("Constante entera: %d, y una letra: %c", 100, 'A');
```

Función *scanf*

scanf permite leer valores introducidos por teclado y almacenarlos en variables de memoria.

La lectura de un dato acaba cuando se encuentra un carácter de espacio en blanco, un tabulador o un ENTER. Si la llamada a la función solicita la lectura de varias variables, estos caracteres tienen la función de separadores de campo, pero habrá que finalizar la lectura con un ENTER.

El formato general de la función es:

```
scanf("cadena de control", lista_de_parámetros)
```

La cadena de control está formada por un conjunto de órdenes de formato. Las órdenes de formato son las mismas definidas para la función *printf*.

La lista de parámetros debe estar formada por direcciones de variables en las que se guardarán los valores leídos; las direcciones se separan por comas.

Para indicar la dirección de una variable, se utiliza el operador & delante del nombre de la variable. Recordad que el nombre de un vector se refiere a la dirección del primer elemento del vector, por lo tanto no es necesario el operador &.

Ejemplos

```
int x;
float f;
char msg[5];

scanf("%d %f", &x, &f); // se lee un entero y un real
                        // si escribimos: 2 3.5 y se aprieta ENTER, se asignará x=2 y f=3.5
                        // hará lo mismo si hacemos: 2 y se aprieta ENTER, 3.5 y se
                        // aprieta ENTER
scanf("%s", msg);       // Al leer una cadena se añade un \0 al final
```

5. Conceptos de programación en ensamblador y C

5.1. Acceso a datos

Para acceder a datos de memoria en ensamblador, como sucede en los lenguajes de alto nivel, lo haremos por medio de variables que deberemos definir previamente para reservar el espacio necesario para almacenar la información.

En la definición podemos especificar el nombre de la variable, el tamaño y un valor inicial; para acceder al dato lo haremos con los operandos de las instrucciones especificando un modo de direccionamiento determinado.

Por ejemplo, `var dd 12345678h` es la variable con el nombre *var* de tamaño 4 bytes inicializada con el valor 12345678h. `mov eax, dword[var]` es la instrucción en la que accedemos a la variable *var* utilizando direccionamiento directo a memoria, en la que especificamos que queremos acceder a 4 bytes (dword) y transferimos el contenido, el valor 12345678h, al registro *eax*.

En ensamblador hay que estar muy alerta cuando accedemos a las variables que hemos definido. Las variables se guardan en memoria consecutivamente a medida que las declaramos y no existe nada que delimite las unas de las otras.

Ejemplo

```
.data
  var1 db 0          ;variable definida de 1 byte
  var2 db 61h        ;variable definida de 1 byte
  var3 dw 0200h      ;variable definida de 2 bytes
  var4 dd 0001E26Ch  ;variable definida de 4 bytes
```

Las variables se encontrarán en memoria tal como muestra la tabla.

Dirección	Valor
var1	00h
var2	61h
var3	00h
	02h
var4	6Ch
	E2h
	01h
	00h

Si ejecutamos la instrucción siguiente:

```
mov eax, dword[var1]
```

Cuando accedemos a `var1` como una variable de tipo `DWORD` el procesador tomará como primer byte el valor de `var1`, pero también los 3 bytes que están a continuación, por lo tanto, como los datos se tratan en formato *little-endian*, consideraremos `DWORD [var1] = 02006100h` y este es el valor que llevaremos a EAX (eax=02006100h). Si este acceso no es el deseado, el compilador no reportará ningún error, ni tampoco se producirá un error durante la ejecución; solo podremos detectar que lo estamos haciendo mal probando el programa y depurando.

Por una parte, el acceso a los datos es muy flexible, pero, por otra parte, si no controlamos muy bien el acceso a las variables, esta flexibilidad nos puede causar ciertos problemas.

En lenguaje C tenemos un comportamiento parecido. Supongamos que tenemos el código siguiente:

```
int vec[4]={1,2,3,4};
int x=258; //258=00000102h
char c=3;

int main() {
   int i=0;
   for (i=0;i<5;i++) printf("contenido de vec", vec[i]);
   c = x;
}
```

El índice `i` toma los valores 0, 1, 2, 3 y 4, el índice `i=4` no corresponde a una posición del vector; la primera posición es la 0 y la última es la 3; por lo tanto, estamos accediendo a una posición fuera del vector. Al compilar el código, no se generará ningún error y tampoco se producirá ningún error durante la ejecución. Si la variable `x` ocupara la posición de memoria siguiente a continuación de los 4 elementos que tenemos definidos del vector, se mostraría un 258.

En la asignación `c = x` como los tipos son diferentes, `x` (int de 4 bytes) y `c` (char de 1 byte), asigna el byte menos significativo de `x` a la variable `c`, después de la asignación `c = 2`. Si hiciéramos la asignación al revés, `x = c`, haría la extensión de signo de `c` a 4 bytes y, por lo tanto, después de la asignación `x = 3 (00000003h)`.

A diferencia del ensamblador, en el que las variables se encuentran en la memoria en el mismo orden en el que las declaramos, en C no podemos asegurarlo, ya que la reserva de espacio de memoria para las variables que hace el compilador GCC es diferente.

5.1.1. Estructuras de datos

Veamos cómo acceder a vectores y matrices utilizando lenguaje C y lenguaje de ensamblador.

Definimos en C un vector y una matriz y hacemos la suma de los elementos.

```
int main(){

   int vec[6]={1,2,3,4,5,6},mat[2][3]={{1,2,3},{4,5,6}};
   int i,j, sumaVec=0, sumaMat=0;

   for (i=0;i<6;i++) sumaVec=sumaVec+vec[i];

   for (i=0;i<2;i++){
   for(j=0;j<3;j++) sumaMat=sumaMat+mat[i][j];
  }
}
```

A continuación, veremos cómo hacer lo mismo con lenguaje de ensamblador, utilizando diferentes variantes del direccionamiento indexado (el formato general es `[dirección + Registro Base + Registro Índice * escala]`):

```
 vec dd 1,2,3,4,5,6
  mat dd 1,2,3
     dd 4,5,6

                              ;recorremos el vector para sumarlo
  mov esi,0                   ;esi será el índice para acceder a los datos
  mov eax,0                   ;eax será donde guardaremos la suma
loop_vec:
  add eax, dword[vec+esi*4]   ;multiplican el índice por 4
  inc esi                     ;porque cada elemento ocupa 4 bytes.
  cmp esi, 6                  ;comparamos con 6 porque es el índice del primer
                              ;elemento fuera del vector.
  jl loop_vec

                              ;recorremos la matriz con un solo índice para sumarla
  mov esi,0                   ;esi será el índice para acceder a los datos
  mov ebx,0                   ;ebx será donde guardaremos la suma
loop_mat:
  add ebx, dword[mat+esi*4]   ;multiplican el índice por 4
  inc esi                     ;porque cada elemento ocupa 4 bytes.
  cmp esi, 6                  ;comparamos con 6 porque es el índice del primer elemento
                              ;fuera de la matriz, la matriz tiene 2*3=6 elementos.
jl loop_mat

                              ;recorremos la matriz con dos índices para sumarla
  mov esi,0                   ;esi será el índice de la columna
  mov edi,0                   ;edi será el índice de la fila
  mov ebx,0                   ;ebx será donde guardaremos la suma
```

```
loop_mat2:
  add ebx, dword[mat+edi+esi*4] ;multiplican el índice de la columna
  inc esi                       ;por 4 porque cada elemento ocupa 4 bytes.
  cmp esi, 3                    ;comparamos con 3 porque son los elementos de una fila.
  jl loop_mat2
  mov esi, 0
  add edi, 12                   ;cada fila ocupa 12 bytes, 3 elementos de 4 bytes.
  cmp edi, 24                   ;comparamos con 24 porque es el primer índice
  jl loop_mat2                  ; fuera de la matriz.
```

Como podemos comprobar, el código para acceder al vector y a la matriz utilizando un índice son idénticos; en ensamblador las matrices se ven como vectores y todas las posiciones de memoria son consecutivas; por lo tanto, si queremos acceder a una posición concreta de una matriz, a partir de un número de fila y de columna, deberemos calcular a qué posición de memoria hay que acceder; fila × elementos_de_la_fila × tamaño_del_dato + columna × tamaño_del_dato.

Por ejemplo, si queremos acceder al elemento mat[1][2], segunda fila, tercera columna (las filas y columnas se empiezan a numerar por 0), este elemento estará en la posición $1 \times 3 \times 4 + 2 \times 4 = 20$, [mat+20].

Recordemos que el ensamblador nos ofrece diferentes modos de direccionamiento que nos pueden facilitar el acceso a determinadas estructuras de datos, tales como los vectores y las matrices.

En el ejemplo anterior, en el que se utilizan dos índices para acceder a la matriz, un índice (edi) lo utilizamos para la fila y el otro índice (esi) para la columna, de manera que el recorrido por la matriz es el siguiente:

```
edi = 0
[mat+0+0*4]    [mat+0+1*4]    [mat+0+2*4]
edi = 12
[mat+12+0*4]  [mat+12+1*4]  [mat+12+2*4]
```

5.1.2. Gestión de la pila

La pila es una zona de memoria que se utiliza para almacenar información de manera temporal. La pila también se utiliza habitualmente para pasar parámetros a las subrutinas y para guardar los registros que son modificados dentro de una subrutina, de manera que se pueda restaurar el valor antes de finalizar la ejecución.

Se trata de una estructura de datos de tipo LIFO (*last in first out*): el último elemento introducido, que habitualmente se denomina *cima de la pila*, es el primer elemento que se saca y es el único directamente accesible.

En los procesadores x86-64, la pila se implementa en memoria principal a partir de una dirección base. Se utiliza el registro RSP como apuntador a la cima de la pila.

La pila crece hacia direcciones más pequeñas; es decir, cada vez que se introduce un valor en la pila, este ocupa una dirección de memoria más pequeña, por lo tanto, el registro RSP se decrementa para apuntar al nuevo valor introducido y se incrementa cuando lo sacamos.

Los elementos se introducen y se sacan de la pila utilizando instrucciones específicas: PUSH para introducir elementos y POP para sacar elementos.

En el modo de 64 bits los elementos que se pueden introducir en la pila y sacar de ella han de ser valores de 16 o 64 bits; por lo tanto, el registro RSP se decrementa o incrementa en 2 u 8 unidades respectivamente.

Al ejecutar la instrucción PUSH, se actualiza el valor de RSP decrementándose en 2 u 8 unidades y se traslada el dato especificado por el operando hacia la cima de la pila. Al ejecutar la instrucción POP, se traslada el valor que está en la cima de la pila hacia el operando de la instrucción y se actualiza el valor de RSP incrementándose en 2 u 8 unidades.

Ejemplo

```
  Supongamos que rax = 0102030405060708h (registro de 64 bits),
bx = 0A0Bh (registro de 16 bits) y rsp = 0000000010203050h.

push rax
push bx
pop bx
pop rax
```

Evolución de la pila al ejecutar estas cuatro instrucciones. En la tabla se muestra el estado de la pila después de ejecutar cada instrucción.

Dirección de memoria	Estado inicial	push rax	push bx	pop bx	pop rax
10203044h					
10203045h					
10203046h			**0B**		
10203047h			**0A**		
10203048h		**08**	08	08	
10203049h		**07**	07	07	
1020304Ah		**06**	06	06	
1020304Bh		**05**	05	05	
1020304Ch		**04**	04	04	
1020304Dh		**03**	03	03	
1020304Eh		**02**	02	02	
1020304Fh		**01**	01	01	

Dirección de memoria	Estado inicial	push rax	push bx	pop bx	pop rax
10203050h	xx	xx	xx	xx	xx
RSP	10203050h	10203048h	10203046h	10203048h	10203050h

- `push rax`. Decrementa RSP en 8 unidades y traslada el valor del registro RAX a partir de la posición de memoria indicada por RSP; funcionalmente la instrucción anterior sería equivalente a:

```
sub rsp, 8
mov qword[rsp], rax
```

- `push bx`. Decrementa RSP en 2 unidades y traslada el valor del registro BX a partir de la posición de memoria indicada por RSP; funcionalmente la instrucción anterior sería equivalente a:

```
sub rsp, 2
mov word[rsp], bx
```

- `pop bx`. Traslada hacia BX el valor de 2 bytes almacenado a partir de la posición de memoria indicada por RSP, a continuación se incrementa RSP en 2. Sería equivalente a efectuar:

```
mov bx, word [rsp]
add rsp, 2
```

- `pop rax`. Traslada hacia RAX el valor de 8 bytes almacenado a partir de la posición de memoria indicada por RSP, a continuación se incrementa RSP en 8. Sería equivalente a efectuar:

```
mov rax, qword [rsp]
add rsp, 8
```

5.2. Operaciones aritméticas

En este subapartado hemos de tener presente que en C podemos construir expresiones utilizando variables, constantes y operadores en una misma sentencia; en cambio, hacer lo mismo en ensamblador implica escribir una secuencia de instrucciones en la que habrá que hacer las operaciones una a una según la prioridad de los operadores dentro de la sentencia.

Ejemplo

Sentencia que podemos expresar en lenguaje C:

```
r=(a+b)*4 / (c>>2);
```

Traducción de la sentencia en lenguaje de ensamblador:

```
mov eax [a]
mov ebx, [b]
add eax, ebx  ;(a+b)
imul eax, 4   ;(a+b)*4
mov ecx,[c]
sar ecx, 2    ;(c>>2)
idiv ecx      ;(a+b)*4 / (c>>2)
mov [r], eax  ;r=(a+b)*4 / (c>>2)
```

5.3. Control de flujo

En este subapartado veremos cómo las diferentes estructuras de control del lenguaje C se pueden traducir a lenguaje de ensamblador.

5.3.1. Estructura *if*

Estructura condicional expresada en lenguaje C:

```
if (condición) {
      bloque de sentencias
}
```

Ejemplo

```
if (a > b) {
     maxA = 1;
     maxB = 0;
}
```

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

```
mov rax, qword [a]     ;Se cargan las variables en registros
mov rbx, qword [b]

cmp rax, rbx           ;Se hace la comparación
jg cierto              ;Si se cumple la condición, salta a la etiqueta cierto
jmp fin                ;Si no se cumple la condición, salta a la etiqueta fin

cierto:
    mov byte [maxA], 1 ;Estas instrucciones solo se ejecutan
    mov byte [maxB], 0 ;cuando se cumple la condición

fin:
```

El código se puede optimizar si se utiliza la condición contraria a la que aparece en el código C (a <= b).

```
    mov rax, qword [a] ;Se carga solo la variable a en un registro
    cmp rax, qword [b] ;Se hace la comparación
    jle fin            ;Si se cumple la condición (a<=b) salta a fin

    mov byte [maxA], 1 ;Estas instrucciones solo se ejecutan
    mov byte [maxB], 0 ;cuando se cumple la condición (a > b)
fin:
```

Si tenemos una condición más compleja, primero habrá que evaluar la expresión de la condición para decidir si ejecutamos o no el bloque de sentencias de *if*.

Ejemplo

```
if ((a != b) || (a>=1 && a<=5)) {
    b = a;
}
```

Se puede traducir a lenguaje de ensamblador de esta manera:

```
mov rax, qword [a]     ;Se cargan las variables en registros
mov rbx, qword [b]

cmp rax, rbx           ;Se hace la comparación (a != b)
jne cierto             ;Si se cumple la condición salta a la etiqueta cierto
                       ;Si no se cumple, como es una OR, se puede cumplir la otra condición.
cmp rax, 1             ;Se hace la comparación (a >= 1), si no se cumple,
jl fin                 ;como es una AND, no hay que mirar la otra condición.
cmp rax, 5             ;Se hace la comparación (a <= 5)
jg fin                 ;Si no salta, se cumple que (a>=1 && a<=5)

cierto:
    mov qword [b], rax ;Hacemos la asignación cuando la condición
es cierta.

fin:
```

5.3.2. Estructura *if-else*

Estructura condicional expresada en lenguaje C considerando la condición alternativa:

```
if (condición) {
    bloque de sentencias
}
else {
    bloque de sentencias alternativas
}
```

Ejemplo

```
if (a > b) {
    max = 'a';
}
else { // (a <= b)
    max = 'b';
}
```

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

```
    mov rax, qword [a]  ;Se cargan las variables en registros
    mov rbx, qword [b]
    cmp rax, rbx        ;Se hace la comparación
    jg cierto           ;Si se cumple la condición, se salta a cierto

    mov byte [max], 'b' ;else (a <= b)
    jmp fin

cierto:
    mov byte [max], 'a' ;if (a > b)

fin:
```

5.3.3. Estructura *while*

Estructura iterativa controlada por una condición expresada al principio:

```
while (condiciones) {
    bloque de sentencias que ejecutar
}
```

Ejemplo

```
resultado=1;
while (num > 1){ //mientras num sea mayor que 1 hacer ...
    resultado = resultado * num;
    num--;
}
```

Se puede traducir a lenguaje de ensamblador de esta manera:

```
  mov rax, 1           ;rax será [resultado]
  mov rbx, qword [num] ;Se carga la variable en un registro
while:
  cmp rbx, 1           ;Se hace la comparación
  jg cierto            ;Si se cumple la condición (num > 1) salta a cierto
  jmp fin              ;Si no, salta a fin
cierto:
  imul rax, rbx        ;rax = rax * rbx
  dec rbx
  jmp while
fin:
  mov qword [resultado], rax
  mov qword [num], rbx
```

Una versión alternativa que utiliza la condición contraria (num <= 0) y trabaja directamente con una variable de memoria:

```
  mov rax, 1            ;rax será [resultado]
while:
  cmp qword [num], 1    ;Se hace la comparación
  jle fi                ;Si se cumple la condición (num <= 1) salta a fin
  imul rax, qword [num] ;rax=rax*[num]
  dec qword [num]
  jmp while
fin:
  mov qword [resultado], rax
```

En este caso, reducimos tres instrucciones del código, pero aumentamos los accesos a memoria durante la ejecución del bucle.

5.3.4. Estructura *do-while*

Estructura iterativa controlada por una condición expresada al final:

```
do {
   bloque de sentencias que ejecutar
} while (condiciones);
```

Ejemplo

```
resultado = 1;
do {
   resultado = resultado * num;
   num--;
} while (num > 1)
```

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

```
  mov rax, 1            ;rax será [resultado]
  mov rbx, qword [num]  ;Se carga la variable en un registro
while:
  imul rax, rbx
  dec rbx
  cmp rbx, 1            ;Se hace la comparación
  jg while              ;Si se cumple la condición salta a while

  mov qword [resultado], rax
  mov qword [num], rbx
```

5.3.5. Estructura *for*

Estructura iterativa, que utiliza la orden *for*:

```
for (valores iniciales; condiciones; actualización) {
    bloque de sentencias que ejecutar
}
```

Ejemplo

```
resultado=1;
for (i = num; i > 1; i--)
{
   resultado=resultado*i;
}
```

Se puede traducir a lenguaje de ensamblador de esta manera:

```
  mov rax, 1            ;rax será [resultado]
  mov rcx, qword [num]  ;rcx será [i] que inicializamos con [num]

for:
  cmp rcx, 1            ;Se hace la comparación
  jg cierto             ;Si se cumple la condición, salta a cierto
  jmp fin
cierto:
  imul rax,rcx
  dec rcx
  jmp for
fin:
  mov qword [resultado], rax
  mov qword [i], rcx
```

Al salir de la estructura iterativa, no actualizamos la variable *num* porque se utiliza para inicializar *i* pero no se cambia.

5.4. Subrutinas y paso de parámetros

Una subrutina es una unidad de código autocontenida, diseñada para llevar a cabo una tarea determinada y tiene un papel determinante en el desarrollo de programas de manera estructurada.

Una subrutina en ensamblador sería equivalente a una función en C. En este apartado veremos cómo definir subrutinas en ensamblador y cómo las podemos utilizar después desde un programa en C.

Primero describiremos cómo trabajar con subrutinas en ensamblador:

- Definición de subrutinas en ensamblador.
- Llamada y retorno de subrutina.
- Paso de parámetros a la subrutina y retorno de resultados.

A continuación veremos cómo hacer llamadas a subrutinas hechas en ensamblador desde un programa en C y qué implicaciones tiene en el paso de parámetros.

5.4.1. Definición de subrutinas en ensamblador

Básicamente, una subrutina es un conjunto de instrucciones que inician su ejecución en un punto de código identificado con una etiqueta que será el nombre de la subrutina, y finaliza con la ejecución de una instrucción *ret*, instrucción de retorno de subrutina, que provoca un salto a la instrucción siguiente desde donde se ha hecho la llamada (*call*).

La estructura básica de una subrutina sería:

```
subrutina:
   ;
   ; Instrucciones de la subrutina
   ;
   ret
```

Consideraciones importantes a la hora de definir una subrutina:

- Debemos almacenar los registros modificados dentro de la subrutina para dejarlos en el mismo estado en el que se encontraban en el momento de hacer la llamada a la subrutina, salvo los registros que se utilicen para

devolver un valor. Para almacenar los registros modificados utilizaremos la pila.

- Para mantener la estructura de una subrutina y para que el programa funcione correctamente, no se pueden efectuar saltos a instrucciones de la subrutina; siempre finalizaremos la ejecución de la subrutina con la instrucción *ret*.

```
subrutina:
   ; Almacenar en la pila
   ; los registros modificados dentro de la subrutina.
   ;
   ; Instrucciones de la subrutina.
   ;
   ; Restaurar el estado de los registros modificados
   ; recuperando su valor inicial almacenado en la pila.
   ret
```

Ejemplo de subrutina que calcula el factorial de un número

```
factorial:
   push rax     ; Almacenar en la pila
   push rbx     ; los registros modificados dentro de la subrutina.
                ; Instrucciones de la subrutina
   mov rax, 1   ; rax será el resultado
   mov rbx, 5   ; Calculamos el factorial del valor de rbx (=5)
while:
   imul rax, rbx
   dec rbx
   cmp rbx, 1   ; Se hace la comparación
   jg while     ; Si se cumple la condición salta a while
                ; En rax tendremos el valor del factorial de 5 (=120)

   pop rbx      ; Restauramos el valor inicial de los registros
   pop rax      ; en orden inverso a como los hemos almacenado.
ret
```

5.4.2. Llamada y retorno de subrutina

Para hacer la llamada a la subrutina se utiliza la instrucción `call` y se indica la etiqueta que define el punto de entrada a la subrutina:

```
call factorial
```

La instrucción *call* almacena en la pila la dirección de retorno (la dirección de la instrucción que se encuentra a continuación de la instrucción *call*) y entonces transfiere el control del programa a la subrutina, cargando en el registro RIP la dirección de la primera instrucción de la subrutina.

Funcionalmente, la instrucción *call* anterior sería equivalente a:

```
sub rsp, 8
mov qword[rsp], rip
mov rip, factorial
```

Para finalizar la ejecución de la subrutina, ejecutaremos la instrucción *ret*, que recupera de la pila la dirección del registro RIP que hemos almacenado al hacer *call* y la carga otra vez en el registro RIP; continúa la ejecución del programa con la instrucción que se encuentra después de *call*.

Funcionalmente, la instrucción *ret* sería equivalente a:

```
mov rip, qword[rsp]
add rsp, 8
```

5.4.3. Paso de parámetros a la subrutina y retorno de resultados

Una subrutina puede necesitar que se le transfieran parámetros; los parámetros se pueden pasar mediante registros o la pila. Sucede lo mismo con el retorno de resultados, que puede efectuarse por medio de registro o de la pila. Consideraremos los casos en los que el número de parámetros de entrada y de retorno de una subrutina es fijo.

Paso de parámetros y retorno de resultado por medio de registros

Debemos definir sobre qué registros concretos queremos pasar parámetros a la subrutina y sobre qué registros haremos el retorno; podemos utilizar cualquier registro de propósito general del procesador.

Una vez definidos los registros que utilizaremos para hacer el paso de parámetros, deberemos asignar a cada uno el valor que queremos pasar a la subrutina antes de hacer *call*; para devolver los valores, dentro de la subrutina, tendremos que asignar a los registros correspondientes el valor que se debe devolver antes de hacer *ret*.

Recordemos que los registros que se utilicen para devolver un valor no se han de almacenar en la pila al inicio de la subrutina, ya que no hay que conservar el valor inicial.

Supongamos que en el ejemplo del factorial queremos pasar como parámetro un número cuyo factorial queremos calcular, y devolver como resultado el factorial del número transferido como parámetro, implementando el paso de parámetros y el retorno de resultados por medio de registros.

El número cuyo factorial queremos calcular lo pasaremos por medio del registro RBX y devolveremos el resultado al registro RAX.

La llamada de la subrutina será:

```
    mov rbx, 5
    call factorial
;En rax tendremos el valor del factorial de 5 (=120)
```

Subrutina:

```
factorial:
   push rbx     ; Almacenar en la pila el registro que modificamos
                ; y que no se utiliza para devolver el resultado.

; Instrucciones de la subrutina
  mov rax, 1    ; rax será el resultado
while:
  imul rax, rbx
  dec rbx
  cmp rbx, 1    ; Se hace la comparación
  jg while      ; Si se cumple la condición salta a while
                ; En rax tendremos el valor del factorial de rbx

pop rbx         ; Restauramos el valor inicial del registro

ret
```

Paso de parámetros y retorno de resultado por medio de la pila

Si queremos pasar parámetros y devolver resultados a una subrutina utilizando la pila, y una vez definidos los parámetros que queremos pasar y los que queremos retornar, hay que hacer lo siguiente:

1) Antes de hacer la llamada a la subrutina: es necesario reservar espacio en la pila para los datos que queremos devolver y a continuación introducir los parámetros necesarios en la pila.

2) Dentro de la subrutina: hay que acceder a los parámetros leyéndolos directamente de memoria, utilizando un registro que apunte a la cima de la pila.

El registro apuntador de pila, RSP, siempre apunta a la cima de la pila y, por lo tanto, podemos acceder al contenido de la pila haciendo un direccionamiento a memoria que utilice RSP, pero si utilizamos la pila dentro de la subrutina, no se recomienda utilizarlo.

El registro que se suele utilizar como apuntador para acceder a la pila es el registro RBP. Antes de utilizarlo, lo tendremos que almacenar en la pila para poder recuperar el valor inicial al final de la subrutina, a continuación se carga en RBP el valor de RSP.

RBP no se debe cambiar dentro de la subrutina; al final de esta se copia el valor sobre RSP para restaurar el valor inicial.

3) Después de ejecutar la subrutina: una vez fuera de la subrutina es necesario liberar el espacio utilizado por los parámetros de entrada y después recuperar los resultados del espacio que hemos reservado antes de hacer la llamada.

Supongamos que en el ejemplo del factorial queremos pasar como parámetro el número cuyo factorial queremos calcular y devolver como resultado el factorial del número pasado como parámetro, implementando el paso de parámetros y el retorno de resultados por medio de la pila.

La llamada de la subrutina:

```
   sub rsp,8       ;Reservamos 8 bytes para el resultado que devolvemos
   mov rbx, 5
   push rbx        ;Introducimos el parámetro de entrada en la pila
   call factorial
   add rsp,8       ;Liberamos el espacio utilizado por el parámetro de entrada
   pop rax         ;Recuperamos el resultado devuelto sobre el registro rax
```

Subrutina:

```
factorial:
   push rbp           ; Almacenar el registro que utilizaremos de apuntador en la pila rbp
   mov rbp, rsp       ; Asignar a RBP el valor del registro apuntador RSP
   push rax           ; Almacenar en la pila los registros que modificamos
   push rbx           ; dentro la subrutina
                      ; Instrucciones de la subrutina
   mov rax, 1         ; rax será el resultado
   mov rbx, [rbp+16]  ; [rbp+16] referencia al parámetro de entrada
while:
   imul rax, rbx
   dec rbx
   cmp rbx, 1         ; Se hace la comparación
   jg while           ; Si se cumple la condición salta a while
                      ; En rax tendremos el valor del factorial de rbx
   mov [rbp+24],rax   ; [rbp+24] espacio que hemos reservado en la pila para devolver el resultado
pop rbx               ; Restauramos el valor inicial de los registros
   pop rax
   mov rsp, rbp       ; Restauramos el valor inicial de RSP con RBP
   pop rbp            ; Restauramos el valor inicial de RBP
   ret
```

Cabe recordar que la memoria se direcciona byte a byte; cada dirección corresponde a una posición de memoria de un byte. En este ejemplo, ya que los elementos que ponemos en la pila son de 8 bytes (64 bits), para pasar de un dato al siguiente, tendremos que hacer incrementos de 8 bytes.

Evolución de la pila al ejecutar este código. La tabla muestra el estado de la pila después de ejecutar cada instrucción o conjunto de instrucciones. La dirección inicial de la pila es @.

Dirección	Estado inicial		sub rsp,8		push rbx		call factorial	
	apuntadores	pila	apuntadores	pila	apuntadores	pila	apuntadores	pila
@ - 48								
@ - 40								

Dirección	Estado inicial		sub rsp,8		push rbx		call factorial	
	apuntadores	pila	apuntadores	pila	apuntadores	pila	apuntadores	pila
@ - 32								
@ - 24							rsp→	@retorn
@ - 16					rsp→	5		5
@ - 8			rsp→	----		----		----
@	rsp→	----		----		----		----

	push rbp mov rbp, rsp		push rax push rbx		mov rbx, rbp+16]		mov [rbp+24], rax	
	apuntadores	pila	apuntadores	pila	apuntadores	pila	apuntadores	pila
@ - 48			rsp→	rbx	rsp→	rbx	rsp→	rbx
@ - 40				rax		rax		rax
@ - 32	rsp, rbp→	rbp	rbp→	rbp	rbp→	rbp	rbp→	rbp
@ - 24		@retorn		@retorn		@retorn		@retorn
@ - 16		5		5	rbp+16→	**5**		**5**
@ - 8		----		----		----	rbp+24→	120
@		----		----		----		----

	pop rbx pop rax mov rsp,rbp pop rbp		ret		add rsp,8		pop rax	
	apuntadores	pila	apuntadores	pila	apuntadores	pila	apuntadores	pila
@ - 48								
@ - 40								
@ - 32								
@ - 24	rsp→	@retorn						
@ - 16		**5**	rsp→	**5**		----		
@ - 8		120		120	rsp→	120		----
@		----		----		----	rsp→	----

Hay que tener presente que el procesador también utiliza la pila para almacenar los valores necesarios para poder efectuar un retorno de la subrutina de manera correcta. En concreto, siempre que se ejecuta una instrucción *call*, el procesador guarda en la cima de la pila la dirección de retorno (el valor actualizado del contador de programa RIP).

Es importante asegurarse de que en el momento de hacer *ret*, la dirección de retorno se encuentre en la cima de la pila; en caso contrario, se romperá la secuencia normal de ejecución.

Eso se consigue si no se modifica el valor de RBP dentro de la subrutina y al final se ejecutan las instrucciones:

```
mov rsp, rbp    ;Restauramos el valor inicial de RSP con RBP
pop rbp         ;Restauramos el valor inicial de RBP
```

Otra manera de acceder a los parámetros de entrada que tenemos en la pila es sacar los parámetros de la pila y almacenarlos en registros; en este caso, será necesario primero sacar la dirección de retorno, a continuación sacar los parámetros y volver a introducir la dirección de retorno. Esta manera de acceder a los parámetros no se suele utilizar porque los registros que guardan los parámetros no se pueden utilizar para otros propósitos y el número de parámetros viene condicionado por el número de registros disponibles. Además, la gestión de los valores de retorno complica mucho la gestión de la pila.

```
pop rax ;rax contendrá la dirección de retorno
pop rdx ;recuperamos los parámetros.
pop rcx
push rax    ;Volvemos a introducir la dirección de retorno
```

La eliminación del espacio utilizado por los parámetros de entrada de la subrutina se lleva a cabo fuera de la subrutina, incrementando el valor del registro RSP en tantas unidades como bytes ocupaban los parámetros de entrada.

```
add rsp,8 ;liberamos el espacio utilizado por el parámetro de entrada
```

Esta tarea también se puede hacer dentro de la subrutina con la instrucción *ret*, que permite especificar un valor que indica el número de bytes que ocupaban los parámetros de entrada, de manera que al retornar, se actualiza RSP incrementándolo tantos bytes como el valor del parámetro *ret*.

Poner la instrucción siguiente al final de la subrutina factorial sería equivalente a ejecutar la instrucción `add rsp,8` después de la llamada a subrutina:

```
ret 8
```

Variables locales

En los lenguajes de alto nivel es habitual definir variables locales dentro de las funciones definidas en un programa. Las variables locales ocupan un espacio definido dentro de la pila.

Ahora veremos cómo reservar espacio para variables locales en subrutinas definidas en ensamblador.

Para reservar el espacio necesario, hemos de saber cuántos bytes utilizaremos como variables locales. A continuación, es necesario decrementar el valor del apuntador a pila RSP tantas unidades como bytes se quieran reservar para las variables locales; de esta manera, si utilizamos las instrucciones que trabajan con la pila dentro de la subrutina (push y pop) no sobreescribiremos el espacio de las variables locales. La actualización de RSP se hace justo después de actualizar el registro que utilizamos para acceder a la pila, RBP.

```
subrutina:
   push rbp           ; Almacenar el registro que utilizaremos
                      ; de apuntador a la pila rbp
   mov rbp, rsp       ; Asignar a RBP el valor del registro apuntador RSP
   sub rsp, n         ; n indica el número de bytes reservados para las
                      ; variables locales
                      ;
                      ; Instrucciones de la subrutina
                      ;
      mov rsp, rbp    ; Restauramos el valor inicial de RSP con
RBP
   pop rbp            ; Restauramos el valor inicial de RBP
ret
```

Como las variables locales están en la pila, se utiliza también el registro apuntador RBP para acceder a las variables. Se utiliza un direccionamiento indexado sobre el registro RBP para acceder al espacio reservado, restando un valor al registro RSP.

Ejemplo

```
push rbp                ;Almacenar el registro que utilizaremos
                        ;de apuntador a la pila rbp
mov rbp, rsp            ;Asignar a RBP el valor del registro apuntador RSP
sub rsp, 8              ;reservamos 8 bytes para variables locales

mov al, byte[RBP-1]     ;accedemos a 1 byte de almacenamiento local
mov ax, word[RBP-2]     ;accedemos a 2 bytes de almacenamiento local
mov eax, dword[RBP-4]   ;accedemos a 4 bytes de almacenamiento local
mov rax, qword[RBP-8]   ;accedemos a 8 bytes de almacenamiento local
```

5.4.4. Llamadas a subrutinas y paso de parámetros desde C

En este subapartado se explica cómo llamar a subrutinas escritas en lenguaje de ensamblador desde un programa escrito en C y cómo transferirle parámetros.

Para poder utilizar una función escrita en lenguaje de ensamblador dentro de un programa escrito en C, debemos tener presentes una serie de cuestiones que afectan tanto al código en lenguaje C como al código en lenguaje de ensamblador.

Definición e implementación de funciones en ensamblador

Cualquier función en ensamblador a la que se quiera llamar desde otro programa se tendrá que declarar utilizando la directiva *global*.

Si la función en ensamblador debe recibir parámetros por medio de la pila, habrá que actualizar los apuntadores a la pila al principio de la función:

```
push rbp     ;Almacenar el registro rbp en la pila
mov rbp, rsp ;Asignar el valor del registro apuntador RSP
```

Y habrá que restaurar los valores al final:

```
mov rsp, rbp   ;Restauramos el valor inicial de RSP con RBP
pop rbp     ;Restauramos el valor inicial de RBP
```

Si la función en ensamblador ha de utilizar variables locales, hay que reservar el espacio necesario en la pila. Es necesario decrementar el valor del registro RSP en tantas unidades como bytes utilizamos como variables locales:

```
sub rsp, n
```

La estructura general del código ensamblador sería la siguiente:

```
section .text

                ;debemos declarar con global el nombre de las funciones
                ;global funcio1..., funcioN

funcio1:
   push rbp       ;Almacenar el registro rbp en la pila
   mov rbp,rsp    ;Asignar el valor del registro apuntador RSP
   sub rsp, n     ;Reservar n bytes para variables locales
                  ;
                  ; código de la función
                  ;
   mov rsp, rbp   ;Restauramos el valor inicial de RSP con RBP
   pop rbp        ;Restauramos el valor inicial de RBP
   ret

...

funcioN:
   push rbp       ;Almacenar el registro rbp en la pila
   mov rbp,rsp    ;Asignar el valor del registro apuntador RSP
   sub rsp, n     ;Reservamos n bytes para variables locales
                  ;
                  ; código de la función
                  ;
   mov rsp, rbp   ;Restauramos el valor inicial de RSP con RBP
   pop rbp        ;Restauramos el valor inicial de RBP
   ret
```

Paso de parámetros y retorno de resultado

En el modo de 64 bits, cuando se llama a una función en lenguaje C los seis primeros parámetros se pasan a la función por registro, utilizando los registros siguientes: RDI, RSI, RDX, RCX, R8 y R9. El resto de los parámetros se pasan por medio de la pila. El valor de retorno de la función se pasa siempre por el registro RAX.

Ejemplo

Queremos definir e implementar algunas funciones en ensamblador: *fsuma*, *fproducto*, *factorial*. La función *fsuma* recibirá dos parámetros: hará la suma de los dos parámetros y devolverá la suma; la función *fproducto* también recibirá dos parámetros: hará el producto de los dos parámetros y devolverá el resultado; finalmente, la función *factorial* calcula el factorial de un número recibido como parámetro (corresponde a la misma función factorial del subapartado anterior).

El código ensamblador sería el siguiente:

```
;Fichero funciones.asm
section .text
global fsuma, fproduct, factorial

fsuma:
                ;2 parámetros de entrada: rdi, rsi
                ;no hay variables locales
  mov rax, rdi
  add rax, rsi
  ret           ;retorno de resultado por medio de rax, rax=rdi+rsi

fproducto:
                ;2 parámetros de entrada: rdi, rsi
                ;no hay variables locales
  mov rax, rdi
  imul rax, rsi ;rax=rax*rsi=rdi*rsi
  ret           ;retorno de resultado por medio de rax

factorial:
                ;1 parámetro de entrada: rdi
                ;no hay variables locales
  push rdi      ;rdi es modificado por la subrutina
  mov rax, 1    ;rax será el resultado
while:
  imul rax, rdi
  dec rdi
  cmp rdi, 1    ;Se hace la comparación
  jg while      ;Si se cumple la condición salta a while
                ;En rax tendremos el valor del factorial de rdi
pop rdi         ;restauramos el valor de rdi
ret
```

Llamada a las funciones desde un programa en C

Para utilizar funciones ensamblador dentro de un programa en C debemos definirlas en el programa en C, pero sin implementarlas; solo se incluye la cabecera de las funciones.

Las cabeceras incluyen el tipo de retorno de la función, el nombre de la función y los tipos de dato de cada parámetro de las funciones. Se indica delante de la cabecera de la función que se trata de una función externa, utilizando la palabra reservada *extern* e indicando al compilador de C que el código objeto de la función está en otro fichero.

Una vez definidas las funciones, se pueden llamar como cualquier otra función del programa.

Veamos cómo sería el código C que utilice las funciones *fsuma*, *fproducto* y *factorial*:

```
//Fichero principal.c
#include <stdio.h>

extern int fsuma(int, int);
extern int fproducto(int, int);
extern int factorial(int);

int main()
{
int x, y, result;

  printf("\nIntroduce el valor de x: ");
  scanf("%d", &x);
  printf("Introduce el valor de y: ");
  scanf("%d", &y);
  result = fsuma(x, y);
  printf ("La suma de x e y es %d\n", result);
  result = fproducto(x, y);
  printf ("El producto de x e y es %d\n", result);
  result = factorial(x);
  printf ("El factorial de x es %d\n", result);
  return 0;
}
```

Para ejecutar este código, hay que hacer el ensamblaje del código fuente ensamblador, compilar el código fuente C con el código objeto ensamblador y generar el ejecutable.

Parámetros por referencia

Podemos pasar un parámetro a una función por referencia, de manera que, si dentro de la función se modifica este parámetro, al salir de la función este mantenga el valor obtenido dentro de la función.

Ejemplo

Modificamos la función factorial para pasar el parámetro de entrada por referencia, pasando una dirección de memoria como parámetro (un apuntador de tipo *int*). Llamamos a la función y pasamos como parámetro la dirección de la variable *x*.

```
;Fichero funciones.asm
section .text
global factorial

factorial:
                        ;parámetro entrada: rdi=dirección de una variable
                        ;no hay variables locales
   push rbx             ;rbx es modificado por la subrutina
   mov rbx,0
   mov ebx, dword[rdi]
   mov rax, 1           ;rax será el resultado
while:
   imul rax, rbx
   dec rbx
   cmp rbx, 1           ;Se hace la comparación
   jg while             ;Si se cumple la condición, salta a while
                        ;En rax tendremos el valor del factorial de rdi.
   mov dword[rdi],eax   ;copiamos el resultado a la variable de memoria
   pop rdi              ;restauramos el valor de rdi
   ret

//Fichero principal.c
#include <stdio.h>

extern void factorial(int *); // El parámetro es una dirección
                              // resultado devuelto sobre el parámetro
int main()
{
  int x;

   printf("\nIntroduce el valor de x: ");
   scanf("%d", &x);
   factorial(&x);
    printf ("El factorial de x es "%d\n", x);
}
```

5.5. Entrada/salida

El sistema de E/S en los procesadores de la familia x86-64 utiliza un mapa independiente de E/S.

Se dispone de un espacio separado de direccionamiento de 64 K, accesible como puertos de 8, 16 y 32 bits; según el dispositivo, se podrá acceder a unidades de 8, 16 o 32 bits.

El juego de instrucciones dispone de instrucciones específicas, IN y OUT, para leer de un puerto de E/S y escribir en un puerto de E/S.

5.5.1. E/S programada

Se pueden realizar tareas de E/S programada utilizando las instrucciones IN y OUT de ensamblador, o las instrucciones equivalentes de lenguaje C: *inb*, *outb*.

En los sistemas Linux debemos permitir el acceso a los puertos de E/S utilizando la llamada al sistema *io_perm*, antes de acceder a un puerto de E/S.

Ejemplo

A continuación se muestra un programa escrito en C que accede a los puertos de E/S 70h y 71h correspondientes a los registros de direcciones y de datos del reloj de tiempo real del sistema (RTC).

La dirección 70h corresponde al registro de direcciones, en el que se escribe la dirección del dato del RTC que se quiere leer:

00: segundos
02: minutos
04: hora
06: día de la semana

Después de enviar al registro de direcciones la dirección del dato que se quiere leer, se puede leer el dato mediante el registro de datos.

```
//Fichero: ioprog.c
#include <stdio.h>
#include <stdlib.h>
#include <ys/io.h>

#define RTC_ADDR 0x70 /* registro de direcciones del RTC */
#define RTC_DATA 0x71 /* registro de datos del RTC */

int main()
{
  char dd, hh, mm;
 // Se define cada día de la semana de 10 caracteres, 'viernes' es el más largo
 // y ocupa 10 caracteres incluido un \0 al final
  char semana[7][10]={"domingo", "lunes", "martes", "miércoles",
  "jueves", "viernes", "sábado"};

/* Se proporciona permiso para acceder a los puertos de E/S
  RTC_ADDR dirección inicial a partir de la cual se quiere acceder
  2: se solicita acceso a 2 bytes
  1: se activa el permiso de acceso */
if (ioperm(RTC_ADDR, 2, 1)) {perror("ioperm"); exit(1);}

  // se escribe en el registro de direcciones la dirección 6: día semana
  outb(6, RTC_ADDR);
  // se lee del puerto de datos el día de la semana
  dd=inb(RTC_DATA);
  printf("día semana RTC: %s\n", semana[(int)dd]);

  // Se escribe en el registro de direcciones la dirección 4: hora
  outb(4, RTC_ADDR);
  // Se lee del puerto de datos la hora
  hh=inb(RTC_DATA);
  printf("Hora RTC: %0x": hh);

  // Se escribe en el registro de direcciones la dirección 2: minutos
  outb(2, RTC_ADDR);
  // Se lee del puerto de datos los minutos
  mm=inb(RTC_DATA);
  printf("%0x\n", mm);

  // Se desactivan los permisos para acceder a los puertos poniendo un 0
  if (ioperm(RTC_ADDR, 2, 0)) {perror("ioperm"); exit(1);}

  exit(0);
}
```

Para generar el ejecutable correctamente, hay que añadir la opción `-O2` al compilador:

```
gcc -o ioprog -g -O2 ioprog.c
```

Para poder ejecutar un programa que acceda a puertos de E/S en Linux, es necesario disponer de permisos de superusuario.

Ejemplo

A continuación se muestra otro ejemplo de acceso a puertos de E/S: se trata de leer el registro de datos del teclado, puerto 60h.

El código siguiente hace un bucle que muestra el código de la tecla pulsada (*scancode*) hasta que se pulsa la tecla ESC.

El registro 60h almacena en los 7 bits de menos peso el código de cada tecla que se pulsa.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/io.h>

#define KBD_DATA 0x60

int main(){
   int salir=0;
   char data;

   if (ioperm(KBD_DATA, 1, 1)) {
     perror("ioperm"); exit(1);
  }

  while (salir==0){
  data=inb(KBD_DATA);
  //se muestra el código de la tecla pulsada, scancode
  //se hace una AND lógica ('&') para tomar solo los
  // 7 bits menos significativos del dato leído
  printf("tecla: %0x\n", fecha & 0b01111111);
  //se pulsa ESC para salir del bucle, scancode=1
  if ((data & 0b01111111) == 1) salir=1;
  }
  if (ioperm(KBD_DATA, 1, 0)) {
  perror("ioperm"); exit(1);
  }
}
```

5.6. Controlar la consola

En Linux no existen funciones estándar o servicios del sistema operativo para controlar la consola, por ejemplo, para mover el cursor. Disponemos, sin embargo, de la posibilidad de escribir secuencias de *escape* que nos permiten manipular el emulador de terminal de Linux, entre otras operaciones: mover el cursor, limpiar la consola, etc.

Una secuencia de *escape* es una cadena de caracteres que empieza con el carácter ESC (que corresponde al código ASCII 27 decimal o 1Bh).

Las principales secuencias de *escape* son las siguientes:

1) Mover el cursor. Para mover el cursor hay que considerar que la posición inicial en un terminal Linux corresponde a la fila 0 de la columna 0; para mover el cursor escribiremos la cadena de caracteres siguiente:

```
ESC[F;CH
```

donde *ESC* corresponde al código ASCII del carácter de *escape* 27 decimal o 1Bh, *F* corresponde a la fila donde queremos colocar el cursor, expresada como un valor decimal, y *C* corresponde a la columna donde queremos colocar el cursor, expresada como un valor decimal.

Ejemplo

Podemos mover el cursor a la fila 5 columna 10, escribiendo la secuencia `ESC[05;10H`.

```
section .data
   escSeq db 27,"[05;10H"
   escLen equ 8
section .text
   mov rax,4
   mov rbx,1
   mov rcx, escSeq
   mov rdx, escLen
   int 80h
```

2) Limpiar la consola. Para limpiar la consola debemos escribir la secuencia `ESC[2J`.

```
section .data
   escSeq db 27,"[2J" ;ESC[2J
   escLen equ 4       ; tamaño de la cadena escSeq

section .text
   mov rax,4
   mov rbx,1
   mov rcx, escSeq
   mov rdx, escLen
   int 80h
```

3) Secuencias de *escape* en C. En lenguaje C podemos conseguir el mismo resultado escribiendo la secuencia que emplea la función *printf*; el carácter ESC se puede escribir con su valor hexadecimal 1B.

Ejemplo

```
#include <stdio.h>

int main(){
   printf("\x1B[2J");       //Borra la pantalla
      printf("\x1B[5;10H"); //Sitúa el cursor en la posición (5,10)
}
```

5.7. Funciones del sistema operativo (*system calls*)

Desde un programa en ensamblador podemos hacer llamadas a diferentes funciones del núcleo (*kernel*) del sistema operativo; es lo que se conoce como *system calls* o *kernel system calls*.

El lenguaje de ensamblador proporciona dos mecanismos para poder hacer llamadas al sistema operativo:

1) *int 80h*: este es el mecanismo tradicional en procesadores x86 y, por lo tanto, también está disponible en los procesadores con arquitectura x86-64.

El servicio que se solicita se especifica mediante el registro RAX. Los parámetros necesarios para la ejecución del servicio vienen especificados por medio de los registros RBX, RCX, RDX, RSI, RDI y RBP.

2) ***syscall***: los procesadores de la arquitectura x86-64 proporcionan un mecanismo más eficiente de hacer llamadas al sistema, la instrucción *syscall*.

El servicio solicitado también se especifica por medio de RAX, pero los números que identifican cada servicio son diferentes de los utilizados con la instrucción *int* 80h. Los parámetros se especifican por medio de los registros RDI, RSI, RDX, RCX, R8 y R9.

El sistema operativo proporciona al programador muchas funciones de diferentes tipos; veremos solo las funciones siguientes:

- Lectura del teclado.
- Escritura por pantalla.
- Retorno al sistema operativo.

5.7.1. Lectura de una cadena de caracteres desde el teclado

Lee caracteres del teclado hasta que se pulsa la tecla ENTER. La lectura de caracteres se hace llamando a la función de lectura *read*. Para utilizar esta función hay que especificar el descriptor de archivo que se utilizará; en el caso de una lectura de teclado se utiliza el descriptor correspondiente a la entrada estándar, un 0 en este caso.

Según si se utiliza *int 80h* o *syscall*, los parámetros son los siguientes:

1) ***int 80h***

a) Parámetros de entrada

- RAX = 3
- RBX = 0, descriptor correspondiente a la entrada estándar (teclado)
- RCX = dirección de la variable de memoria donde se guardará la cadena leída
- RDX = número máximo de caracteres que se leerán

b) Parámetros de salida

- RAX = número de caracteres leídos
- La variable indicada se llena con los caracteres leídos.

2) ***syscall***

a) Parámetros de entrada

- RAX = 0
- RDI = 0, descriptor correspondiente a la entrada estándar (teclado)
- RSI = dirección de la variable de memoria donde se guardará la cadena leída
- RDX = número máximo de caracteres que se leerán

b) Parámetros de salida

- RAX = número de caracteres leídos

```
section .bss
   buffer resb 10               ;se reservan 10 bytes para hacer la lectura del
teclado

section .text
                                ;lectura utilizando int 80h
   mov rax, 3
   mov rbx, 0
   mov rcx, buffer              ; se carga la dirección de buffer en rcx
   mov rdx,10                   ; número máximo de caracteres que se leerán
   int 80h                      ; se llama al kernel

                                ;lectura utilizando syscall
   mov rax, 0
   mov rdi, 0
   mov rsi, buffer              ; se carga la dirección de buffer en rsi
   mov rdx,10                   ; número máximo de caracteres que se leerán
   syscall                      ; se llama al kernel
```

5.7.2. Escritura de una cadena de caracteres por pantalla

La escritura de caracteres por pantalla se efectúa llamando a la función de escritura *write*. Para utilizar esta función hay que especificar el descriptor de archivo que se utilizará; en el caso de una escritura por pantalla se utiliza el descriptor correspondiente a la salida estándar, un 1 en este caso.

1) ***int 80h***

a) Parámetros de entrada

- RAX = 4
- RBX = 1, descriptor correspondiente a la salida estándar (pantalla)
- RCX = dirección de la variable de memoria que queremos escribir, la variable ha de estar definida con un byte 0 al final

- RDX = tamaño de la cadena que queremos escribir en bytes, incluido el 0 del final

b) Parámetros de salida

- RAX = número de caracteres escritos

2) ***syscall***

a) Parámetros de entrada

- RAX = 1
- RDI = 1, descriptor correspondiente a la salida estándar (pantalla)
- RSI = dirección de la variable de memoria que queremos escribir, la variable ha de estar definida con un byte 0 al final
- RDX = tamaño de la cadena que queremos escribir en bytes, incluido el 0 del final

b) Parámetros de salida

- RAX = número de caracteres escritos

Ejemplo

```
Section .data
   msg db "Hola!",0
   msgLen db 6

section .text
                     ; escritura utilizando int 80h
   mov rax,4
   mov rbx,1
   mov rcx, msg      ; se pone la dirección de msg1 en rcx
   mov rdx, msgLen   ; tamaño de msg
   int 80h           ; se llama al kernel

;escritura utilizando syscall
   mov rax, 1
   mov rdi, 1
   mov rsi, msg      ; se carga la dirección de msg en rsi
   mov rdx, msglen   ; tamaño de msg
   syscall           ; se llama al kernel
```

Se puede calcular el tamaño de una variable calculando la diferencia entre una posición dentro de la sección data y la posición donde se encuentra declarada la variable:

```
section .data
  msg db "Hola!",0
  msgLen db equ $ - msg ; $ indica la dirección de la posición actual
```

$ define la posición del principio de la línea actual; al restarle la etiqueta *msg*, se le resta la posición donde se encuentra declarada la etiqueta y, por lo tanto, se obtiene el número de bytes reservados a la posición de la etiqueta *msg*, 6 en este caso.

5.7.3. Retorno al sistema operativo (*exit*)

Finaliza la ejecución del programa y retorna el control al sistema operativo.

1) *int 80h*

a) Parámetros de entrada

- RAX = 1
- RBX = valor de retorno del programa

2) *syscall*

a) Parámetros de entrada

- RAX = 60
- RDI = valor de retorno del programa

Ejemplo

```
;retorna al sistema operativo utilizando int 80h
mov rax,1
mov rbx,0  ;valor de retorno 0
int 80h   ; se llama al kernel

           ;retorna al sistema operativo utilizando syscall
mov rax,60
mov rbx,0  ;valor de retorno 0
syscall    ;se llama al kernel
```

6. Anexo: manual básico del juego de instrucciones

En este anexo se describen detalladamente las instrucciones más habituales del lenguaje de ensamblador de la arquitectura x86-64, pero debemos tener presente que el objetivo de este apartado no es ofrecer un manual de referencia completo de esta arquitectura y, por lo tanto, no se describen todas las instrucciones del juego de instrucciones.

Descripción de la notación utilizada en las instrucciones:

1) Bits de resultado (*flags*):

- OF: Overflow flag (bit de desbordamiento)
- TF: Trap flag (bit de excepción)
- AF: Aux carry (bit de transporte auxiliar)
- DF: Direction flag (bit de dirección)
- SF: Sign flag (bit de signo)
- PF: Parity flag (bit de paridad)
- IF: Interrupt flag (bit de interrupción)
- ZF: Cero flag (bit de cero)
- CF: Carry flag (bit de transporte)

2) Tipo de operandos:

a) `imm`: valor inmediato; puede ser un valor inmediato de 8, 16 o 32 bits. Según el tamaño del valor inmediato se podrán representar los intervalos de valores siguientes:

- Inmediato de 8 bits: sin signo [0, 255], con signo en Ca2 [–128, +127]

- Inmediato de 16 bits: sin signo [0, 65.535], con signo en Ca2 [–32.768, +32.767]

- Inmediato de 32 bits: sin signo [0, 4.294.967.295], con signo en Ca2 [–2.147.483.648, +2.147.483.647]

Los valores inmediatos de 64 bits solo se permiten para cargar un registro de propósito general de 64 bits, mediante la instrucción MOV. El intervalo de representación es el siguiente:

- sin signo [0, 18.446.744.073.709.551.615],
- con signo en Ca2 [–9.223.372.036.854.775.808, + 9.223.372.036.854.775.807]

b) `reg`: registro, puede ser un registro de 8, 16, 32 o 64 bits.

c) `tamaño mem`: posición de memoria; se indica primero si se accede a 8, 16, 32 o 64 bits y, a continuación, la dirección de memoria.

Aunque especificar el tamaño no es estrictamente necesario, siempre que se utiliza un operando de memoria lo haremos de esta manera, por claridad a la hora de saber a cuántos bytes de memoria se está accediendo.

Los especificadores de tamaño válidos son:

- BYTE: posición de memoria de 8 bits
- WORD: posición de memoria de 16 bits
- DWORD: posición de memoria de 32 bits
- QWORD: posición de memoria de 64 bits

6.1. ADC: suma aritmética con bit de transporte

ADC destino, fuente

Efectúa una suma aritmética; suma el operando fuente y el valor del bit de transporte (CF) al operando de destino, almacena el resultado sobre el operando destino y sustituye el valor inicial del operando destino.

Operación

destino = destino + fuente + CF

Bits de resultado modificados

OF, SF, ZF, AF, PF, CF

Si el resultado no cabe dentro del operando destino, el bit de transporte se pone a 1. El resto de bits de resultado se modifican según el resultado de la operación.

Formatos válidos

```
ADC reg, reg
ADC reg, tamaño mem
```

```
ADC tamaño mem, reg
```

Los dos operandos han de ser del mismo tamaño.

```
ADC reg, imm
ADC tamaño mem, imm
```

El tamaño del inmediato puede ir desde 8 bits hasta el tamaño del primer operando, 32 bits como máximo.

Ejemplos

```
ADC R9,RAX
ADC RAX, QWORD [variable]
ADC DWORD [variable],EAX
ADC RAX,0x01020304
ADC BYTE [vector+RAX], 5
```

6.2. ADD: suma aritmética

ADD destino, fuente

Efectúa la suma aritmética de los dos operandos de la instrucción, almacena el resultado sobre el operando destino y sustituye el valor inicial del operando destino.

Operación

destino = destino + fuente

Bits de resultado modificados

OF, SF, ZF, AF, PF, CF

Si el resultado no cabe dentro del operando destino, el bit de transporte se pone a 1. El resto de bits de resultado se modifican según el resultado de la operación.

Formatos válidos

```
ADD reg, reg
ADD reg, tamaño mem
ADD tamaño mem, reg
```

Los dos operandos han de ser del mismo tamaño.

```
ADD reg, imm
ADD tamaño mem, imm
```

El tamaño del inmediato puede ir desde 8 bits hasta el tamaño del primer operando, 32 bits como máximo.

Ejemplos

```
ADD R9,RAX
ADD RAX, QWORD [variable]
ADD DWORD [variable],EAX
ADD RAX,0x01020304
ADD BYTE [vector+RAX], 5
```

6.3. AND: Y lógica

AND destino, fuente

Realiza una operación lógica AND ('y lógica') bit a bit entre el operando destino y el operando fuente, el resultado de la operación se guarda sobre el operando destino sobreescribiendo el valor inicial. El valor del operando fuente no se modifica.

Se lleva a cabo una operación AND entre el bit *n* del operando destino y el bit *n* del operando fuente según la tabla de verdad de la función AND:

```
x       y           x AND y
---         ---           -------
0           0             0
0           1             0
1           0             0
1           1             1
```

Operación

destino = destino AND fuente

Bits de resultado modificados

OF=0, SF, ZF, PF, CF=0

Los bits de resultado OF y CF se ponen a 0, el resto se cambia según el resultado de la operación.

Formatos válidos

```
AND reg,reg
AND reg,tamaño mem
AND tamaño mem,reg
```

Los dos operandos han de ser del mismo tamaño.

```
AND reg,imm
AND tamaño mem,imm
```

El tamaño del inmediato puede ir desde 8 bits hasta el tamaño del primer operando, como máximo 32 bits.

Ejemplos

```
AND R9,RAX
AND RAX, QWORD [variable]
AND DWORD [variable],EAX
AND RAX,0x01020304
AND BYTE [vector+RAX], 5
```

6.4. CALL: llamada a subrutina

CALL etiqueta

Llama a la subrutina que se encuentra en la dirección de memoria indicada por la etiqueta. Guarda en la pila la dirección de memoria de la instrucción que sigue en secuencia la instrucción CALL y permite el retorno desde la subrutina con la instrucción RET; a continuación carga en el RIP (*instruction pointer*) la dirección de memoria donde está la etiqueta especificada en la instrucción y transfiere el control a la subrutina.

En entornos de 64 bits, la dirección de retorno será de 64 bits, por lo tanto, se introduce en la pila un valor de 8 bytes. Para hacerlo, primero se actualiza el puntero de pila (registro RSP) decrementándose en 8 unidades, y a continuación se copia la dirección en la cima de la pila.

Operación

RSP=RSP-8
M[RSP]← RIP
RIP ← dirección_etiqueta

Bits de resultado modificados

Ninguno

Formatos válidos

```
CALL etiqueta
```

Existen otros formatos, pero quedan fuera de los objetivos de estos materiales. Podéis consultar las fuentes bibliográficas.

Ejemplos

```
CALL subrutina1
```

6.5. CMP: comparación aritmética

CMP destino, fuente

Compara los dos operandos de la instrucción sin afectar al valor de ninguno de los operandos, actualiza los bits de resultado según el resultado de la comparación. La comparación se realiza con una resta entre los dos operandos, sin considerar el transporte y sin guardar el resultado.

Operación

destino - fuente

Bits de resultado modificados

OF, SF, ZF, AF, PF, CF

Los bits de resultado se modifican según el resultado de la operación de resta.

Formatos válidos

```
CMP reg,reg
CMP reg,tamaño mem
CMP tamaño mem,reg
```

Los dos operandos han de ser del mismo tamaño.

```
CMP reg,imm
CMP tamaño mem,imm
```

El tamaño del inmediato puede ir desde 8 bits hasta el tamaño del primer operando, como máximo 32 bits.

Ejemplos

```
CMP R9,RAX
CMP RAX, QWORD [variable]
CMP DWORD [variable],EAX
CMP RAX,0x01020304
CMP BYTE [vector+RAX], 5
```

6.6. DEC: decrementa el operando

DEC destino

Resta 1 al operando de la instrucción y almacena el resultado en el mismo operando.

Operación

destino = destino – 1

Bits de resultado modificados

OF, SF, ZF, AF, PF

Los bits de resultado se modifican según el resultado de la operación.

Formatos válidos

```
DEC reg
DEC tamaño mem
```

Ejemplos

```
DEC EAX
DEC R9
DEC DWORD [R8]
DEC QWORD [variable]
```

6.7. DIV: división entera sin signo

DIV fuente

Divide el dividendo implícito entre el divisor explícito sin considerar los signos de los operandos.

Si el divisor es de 8 bits, se considera como dividendo implícito AX. El cociente de la división queda en AL y el resto, en AH.

Si el divisor es de 16 bits, se considera como dividendo implícito el par de registros DX:AX; la parte menos significativa del dividendo se coloca en AX y la parte más significativa, en DX. El cociente de la división queda en AX y el resto, en DX.

Si el divisor es de 32 bits, el funcionamiento es similar al caso anterior, pero se utiliza el par de registros EDX:EAX; la parte menos significativa del dividendo se coloca en EAX y la parte más significativa, en EDX. El cociente de la división queda en EAX y el resto, en EDX.

Si el divisor es de 64 bits, el funcionamiento es parecido a los dos casos anteriores, pero se utiliza el par de registros RDX:RAX; la parte menos significativa del dividendo se coloca en RAX y la parte más significativa, en RDX. El cociente de la división queda en RAX y el resto, en RDX.

Operación

Si *fuente* es de 8 bits: AL = AX / *fuente*, AH = AX mod *fuente*

Si *fuente* es de 16 bits: AX = DX:AX / *fuente*, DX = DX:AX mod *fuente*

Si *fuente* es de 32 bits: EAX = EDX:EAX / *fuente*, EDX = EDX:EAX mod *fuente*

Si *fuente* es de 64 bits: RAX = RDX:RAX / *fuente*, RDX = RDX:RAX mod *fuente*

Bits de resultado modificados

OF, SF, ZF, AF, PF, CF

La instrucción DIV no deja información a los bits de resultado, pero estos quedan indefinidos.

Formatos válidos

```
DIV reg
DIV tamaño mem
```

Ejemplos

```
DIV R8B    ; AX / R8B => Cociente en AL; resto en AH
DIV R8W    ; DX:AX / R8W => Cociente en AX; resto en DX
DIV ECX    ; EDX:EAX / ECX => Cociente en EAX; resto en EDX
DIV QWORD [R9] ; RDX:RAX / QWORD [R9] => Cociente en RAX,
resto en RDX
```

6.8. IDIV: división entera con signo

IDIV fuente

Divide el dividendo implícito entre el divisor explícito (fuente) considerando el signo de los operandos. El funcionamiento es idéntico al de la división sin signo.

Si el divisor es de 8 bits, se considera como dividendo implícito AX.

El cociente de la división queda en AL y el resto, en AH.

Si el divisor es de 16 bits, se considera como dividendo implícito el par de registros DX:AX; la parte menos significativa del dividendo se coloca en AX y la parte más significativa, en DX. El cociente de la división queda en AX y el resto, en DX.

Si el divisor es de 32 bits, el funcionamiento es similar al caso anterior, pero se utiliza el par de registros EDX:EAX; la parte menos significativa del dividendo se coloca en EAX, y la parte más significativa, en EDX. El cociente de la división queda en EAX y el resto, en EDX.

Si el divisor es de 64 bits, el funcionamiento es parecido a los dos casos anteriores, pero se utiliza el par de registros RDX:RAX; la parte menos significativa del dividendo se coloca en RAX y la parte más significativa, en RDX. El cociente de la división queda en RAX y el resto, en RDX.

Operación

Si *fuente* es de 8 bits: AL = AX / fuente, AH = AX mod *fuente*

Si *fuente* es de 16 bits: AX = DX:AX / *fuente*, DX = DX:AX mod *fuente*

Si *fuente* es de 32 bits: EAX = EDX:EAX / *fuente*, EDX = EDX:EAX mod *fuente*

Si *fuente* es de 64 bits: RAX = RDX:RAX / *fuente*, RDX = RDX:EAX mod *fuente*

Bits de resultado modificados

OF, SF, ZF, AF, PF, CF

La instrucción IDIV no deja información a los bits de resultado, pero estos quedan indefinidos.

Formatos válidos

```
IDIV reg
IDIV tamaño mem
```

Ejemplos

```
IDIV CH   ; AX / CH => Cociente en AL; resto en AH
IDIV BX   ; DX:AX / BX => Cociente en AX; resto en DX
IDIV ECX   ; EDX:EAX / ECX => Cociente en EAX; resto en EDX
IDIV QWORD [R9] ; RDX:RAX / [R9] => Cociente en RAX, resto en RDX
```

6.9. IMUL: multiplicación entera con signo

IMUL fuente
IMUL destino, fuente

La operación de multiplicación con signo puede utilizar diferente número de operandos; se describirá el formato de la instrucción con un operando y con dos operandos.

6.9.1. IMUL fuente: un operando explícito

Multiplica el operando fuente por AL, AX, EAX, o RAX considerando el signo de los operandos y almacena el resultado en AX, DX:AX, EDX:EAX o RDX:RAX.

Operación

Si *fuente* es de 8 bits AX = AL * *fuente*

Si *fuente* es de 16 bits DX:AX = AX * *fuente*

Si *fuente* es de 32 bits EDX:EAX = EAX * *fuente*

Si *fuente* es de 64 bits RDX:RAX = RAX * *fuente*

Bits de resultado modificados

OF, SF, ZF, AF, PF, CF

Los bits de resultado SF, ZF, AF, y PF quedan indefinidos después de que se ejecute la instrucción IMUL.

CF y OF se fijan en 0 si la parte alta del resultado es 0 (AH, DX, EDX o RDX), en caso contrario se fijan en 1.

Formatos válidos

```
IMUL reg
IMUL tamaño mem
```

Ejemplos

```
IMUL ECX   ; EAX * ECX => EDX:EAX
IMUL QWORD [RBX]   ; AX * [RBX] => RDX:RAX
```

6.9.2. IMUL destino, fuente: dos operandos explícitos

Multiplica el operando fuente por el operando destino considerando el signo de los dos operandos y almacena el resultado en el operando destino; sobreescribe el valor que tuviera.

Operación

destino = fuente * destino

Bits de resultado modificados

OF, SF, ZF, AF, PF, CF

Los bits de resultado SF, ZF, AF, y PF quedan indefinidos después de que se ejecute la instrucción IMUL.

CF y OF se fijan en 0 si el resultado se puede representar con el operando destino; si el resultado no es representable con el rango del operando destino (se produce desbordamiento), CF y OF se fijan en 1.

Formatos válidos

```
IMUL reg, imm
```

El tamaño del inmediato puede ir desde 8 bits hasta el tamaño del primer operando, como máximo 32 bits.

```
IMUL reg, reg
IMUL reg, tamaño mem
```

Los dos operandos han de ser del mismo tamaño.

Ejemplos

```
IMUL EAX, 4
IMUL RAX, R9
IMUL RAX, QWORD [var]
```

6.10. IN: lectura de un puerto de entrada/salida

IN destino, fuente

Lee el valor de un puerto de E/S especificado por el operando fuente y lleva el valor al operando destino.

El operando fuente puede ser un valor inmediato de 8 bits, que permite acceder a los puertos 0-255, o el registro DX, que permite acceder a cualquier puerto de E/S de 0-65535.

El operando destino solo puede ser uno de los registros siguientes:

- AL se lee un byte del puerto
- AX se leen dos bytes
- EAX se leen cuatro bytes

Operación

destino=fuente(puerto E/S)

Bits de resultado modificados

Ninguno

Formatos válidos

```
IN AL, imm8
IN AX, imm8
IN EAX, imm8
IN AL, DX
IN AX, DX
IN EAX, DX
```

Ejemplos

```
IN AL, 60 h
IN AL, DX
```

6.11. INC: incrementa el operando

INC destino

Suma 1 al operando de la instrucción y almacena el resultado en el mismo operando.

Operación

destino = destino + 1

Bits de resultado modificados

OF, SF, ZF, AF, PF

Los bits de resultado se modifican según el resultado de la operación.

Formatos válidos

```
INC reg
INC tamaño mem
```

Ejemplos

```
INC AL
INC R9
INC BYTE [RBP]
INC QWORD [var1]
```

6.12. INT: llamada a una interrupción software

```
INT servicio
```

Llamada a un servicio del sistema operativo, a una de las 256 interrupciones *software* definidas en la tabla de vectores de interrupción. El número de servicio ha de ser un valor entre 0 y 255. Es habitual expresar el número del servicio como un valor hexadecimal, de 00h a FFh.

Cuando se llama a una interrupción, el registro EFLAGS y la dirección de retorno son almacenados en la pila.

Operación

RSP=RSP-8
M(RSP) ← EFLAGS
RSP=RSP-8
M(RSP) ← RIP
RIP← dirección rutina de servicio

Bits de resultado modificados

IF, TF

Estos dos bits de resultado se ponen a 0; poner a 0 el bit de resultado IF impide que se trate otra interrupción mientras se está ejecutando la rutina de interrupción actual.

Formatos válidos

```
INT servicio
```

Ejemplos

```
INT 80h
```

6.13. IRET: retorno de interrupción

```
IRET
```

IRET se debe utilizar para salir de las rutinas de servicio a interrupciones (RSI). La instrucción extrae de la pila la dirección de retorno sobre el registro RIP, a continuación saca la palabra siguiente de la pila y la coloca en el registro EFLAGS.

Operación

RIP ← dirección retorno
RSP=RSP+8
EFLAGS ← M(RSP)
RSP=RSP+8

Bits de resultado modificados

Todos: OF, DF, IF, TF, SF, ZF, AF, PF, CF

Modifica todos los bits de resultado, ya que saca de la pila una palabra que es llevada al registro EFLAGS.

Formatos válidos

```
IRET
```

Ejemplo

```
IRET
```

6.14. Jxx: salto condicional

Jxx etiqueta

Realiza un salto según una condición determinada; la condición se comprueba consultando el valor de los bits de resultado.

La etiqueta codifica un desplazamiento de 32 bits con signo y permite dar un salto de -2^{31} bytes a $+2^{31} - 1$ bytes.

Si la condición se cumple, se salta a la posición del código indicada por la etiqueta; se carga en el registro RIP el valor RIP + desplazamiento,

Operación

RIP = RIP + desplazamiento

Bits de resultado modificados

Ninguno

Formatos válidos

Según la condición de salto, tenemos las instrucciones siguientes:

1) Instrucciones que no tienen en cuenta el signo

```
Instrucción    Descripción                              Condición
---------- ---------------------------------------- --------------
JA/JNBE    (Jump If Above/Jump If Not Below or Equal) CF=0 y ZF=0
JAE/JNB    (Jump If Above or Equal/Jump If Not Below) CF=0
JB/JNAE    (Jump If Below/Jump If Not Above or Equal) CF=1
JBE/JNA    (Jump If Below or Equal/Jump If Not Above) CF=1 o ZF=1
```

2) Instrucciones que tienen en cuenta el signo

```
Instrucción     Descripción                              Condición
---------- ---------------------------------------- --------------
JE/JZ      (Jump If Equal/Jump If Zero)               ZF=1
JNE/JNZ    (Jump If Not Equal/Jump If Not Zero)       ZF=0
JG/JNLE    (Jump If Greater/Jump If Not Less or Equal) ZF=0 y SF=OF
```

```
JGE/JNL    (Jump If Greater or Equal/Jump If Not Less) SF=OF
JL/JNGE    (Jump If Less/Jump If Not Greater or Equal) S≠FOF
JLE/JNG    (Jump If Less or Equal/Jump If Not Greater) ZF=1 o S≠FOF
```

3) Instrucciones que comprueban el valor de un bit de resultado

```
Instrucción     Descripción                              Condición
----------  ---------------------------------------- --------------
JC          (Jump If Carry flag set)                   CF=1
JNC         (Jump If Carry flag Not set)               CF=0
JO          (Jump If Overflow flag set)                OF=1
JNO         (Jump If Overflow flag Not set)            OF=0
JS          (Jump If Sign flag set)                    SF=1
JNS         (Jump If Sign flag Not set)                SF=0
```

Ejemplos

```
JE etiqueta1     ;salta si Z=1
JG etiqueta2     ;salta si Z=0 y SF=OF
JL etiqueta3     ;salta si S≠FOF
```

6.15. JMP: salto incondicional

```
JMP etiqueta
```

Salta de manera incondicional a la dirección de memoria correspondiente a la posición de la etiqueta especificada; el registro RIP toma como valor la dirección de la etiqueta.

Operación

RIP=dirección_etiqueta

Bits de resultado modificados

Ninguno

Formatos válidos

```
JMP etiqueta
```

Ejemplo

```
JMP bucle
```

6.16. LOOP: bucle hasta RCX=0

LOOP etiqueta

La instrucción utiliza el registro RCX.

Decrementa el valor de RCX, comprueba si el valor es diferente de cero y en este caso realiza un salto a la etiqueta indicada.

La etiqueta codifica un desplazamiento de 32 bits con signo y permite efectuar un salto de -2^{31} bytes a $+2^{31} - 1$ bytes.

Operación

La instrucción es equivalente al conjunto de instrucciones siguientes:

```
DEC RCX
JNE etiqueta
```

Bits de resultado modificados

Ninguno

Formatos válidos

```
LOOP etiqueta
```

Ejemplo

```
MOV RCX, 10
bucle:
;
;Las instrucciones se repetirán 10 veces
;
LOOP bucle
```

6.17. MOV: transferir un dato

MOV destino, fuente

Copia el valor del operando fuente sobre el operando destino sobreescribiendo el valor original del operando destino.

Operación

destino = fuente

Bits de resultado modificados

Ninguno

Formatos válidos

```
MOV reg, reg
MOV reg, tamaño mem
MOV tamaño mem, reg
```

Los dos operandos deben ser del mismo tamaño.

```
MOV reg, imm
```

El tamaño del inmediato puede ir desde 8 bits hasta el tamaño del registro; se permiten inmediatos de 64 bits si el registro es de 64 bits.

```
MOV tamaño mem, imm
```

El tamaño del inmediato puede ir desde 8 bits hasta el tamaño del primer operando, como máximo 32 bits.

Ejemplos

```
MOV RAX,R9
MOV RAX,QWORD [variable]
MOV QWORD [variable], RAX
MOV RAX,0102030405060708h
MOV WORD [RAX],0B80h
```

6.18. MUL: multiplicación entera sin signo

MUL fuente

MUL multiplica el operando explícito por AL, AX, EAX o RAX sin considerar el signo de los operandos y almacena el resultado en AX, DX:AX, EDX:EAX o RDX:RAX.

Operación

Si *fuente* es de 8 bits AX = AL * *fuente*

Si *fuente* es de 16 bits DX:AX = AX * *fuente*

Si *fuente* es de 32 bits EDX:EAX = EAX * *fuente*

Si *fuente* es de 64 bits RDX:RAX = RAX * *fuente*

Bits de resultado modificados

OF, SF, ZF, AF, PF, CF

Los bits de resultado SF, ZF, AF, y PF quedan indeterminados después de que se ejecute la instrucción MUL.

CF y OF se fijan a 0 si la parte alta del resultado es 0 (AH, DX, EDX, o RDX); en caso contrario se fijan a 1.

Formatos válidos

```
MUL reg
MUL tamaño mem
```

Ejemplos

```
MUL CH           ; AL * CH --> AX
MUL BX           ; AX * BX --> DX:AX
MUL RCX          ; RAX * RCX --> RDX:RAX
MUL WORD [BX+DI] ; AX * [BX+DI]--> DX:AX
```

6.19. NEG: negación aritmética en complemento a 2

NEG destino

Lleva a cabo una negación aritmética del operando, es decir, hace el complemento a 2 del operando especificado; es equivalente a multiplicar el valor del operando por –1.

Esta operación no es equivalente a complementar todos los bits del operando (instrucción NOT).

Operación

destino = (–1) * destino

Bits de resultado modificados

OF, SF, ZF, AF, PF, CF

Si el operando es 0, el resultado no varía y ZF=1 y CF=0; en caso contrario, ZF=0 y CF=1.

Si el operando contiene el máximo valor negativo (por ejemplo 80000000h si el operando es de 32 bits), el valor del operando no se modifica: OF=1 y CF=1.

SF=1 si el resultado es negativo; SF=0 en caso contrario.

PF=1 si el número de unos del byte de menos peso del resultado es par; PF=0 en caso contrario.

Formatos válidos

```
NEG reg
NEG tamaño mem
```

Ejemplos

```
NEG RCX
NEG DWORD [variable]
```

6.20. NOT: negación lógica (negación en complemento a 1)

NOT destino

Lleva a cabo una negación lógica del operando, es decir, hace el complemento a 1 del operando especificado, y complementa todos los bits del operando.

Operación

destino = –destino

Bits de resultado modificados

Ninguno

Formatos válidos

```
NOT reg
NOT tamaño mem
```

Ejemplos

```
NOT RAX
NOT QWORD [variable]
```

6.21. OUT: escritura en un puerto de entrada/salida

OUT destino, fuente

Escribe el valor del operando fuente en un puerto de E/S especificado por el operando destino.

El operando destino puede ser un valor inmediato de 8 bits, que permite acceder a los puertos 0-255, o el registro DX, que permite acceder a cualquier puerto de E/S de 0-65535.

El operando fuente solo puede ser uno de los registros siguientes:

- AL se escribe un byte
- AX se escriben dos bytes
- EAX se escriben cuatro bytes

Operación

destino(puerto E/S) = fuente

Bits de resultado modificados

Ninguno

Formatos válidos

```
OUT imm8, AL
OUT imm8, AX
OUT imm8, EAX
OUT DX, AL
OUT DX, AX
OUT DX, EAX
```

Ejemplos

```
OUT 60h, AL
OUT DX, AL
```

6.22. OR: o lógica

OR destino, fuente

Realiza una operación lógica OR (o lógica) bit a bit entre el operando destino y el operando fuente; el resultado de la operación se guarda sobre el operando destino, sobreescribiendo el valor inicial. El valor del operando fuente no se modifica.

Se lleva a cabo una operación OR entre el bit *n* del operando destino y el bit *n* del operando fuente según la tabla de verdad de la función OR:

```
x     y      x OR y
---   ---      ------
0     0       0
0     1       1
1     0       1
1     1       1
```

Operación

destino = destino OR fuente

Bits de resultado modificados

OF=0, SF, ZF, PF, CF=0

Los bits de resultado OF y CF se ponen a 0, el resto de los bits de resultado se modifican según el resultado de la operación.

Formatos válidos

```
OR reg, reg
OR reg, tamaño mem
OR tamaño mem, reg
```

Los dos operandos han de ser del mismo tamaño.

```
OR reg, imm
OR tamaño mem, imm
```

El tamaño del inmediato puede ir desde 8 bits hasta el tamaño del primer operando, como máximo 32 bits.

Ejemplos

```
OR R9, RAX
OR RAX, QWORD [variable]
OR DWORD [variable], EAX
OR RAX, 0x01020304
OR BYTE [vector+RAX], 5
```

6.23. POP: extraer el valor de la cima de la pila

POP destino

Extrae el valor que se encuentra en la cima de la pila (copia el valor de la memoria apuntado por el registro RSP) y lo almacena en el operando destino especificado; se extraen tantos bytes de la pila como el tamaño del operando indicado.

A continuación se actualiza el valor del registro apuntador de pila, RSP, incrementándolo en tantas unidades como el número de bytes extraídos de la pila.

El operando puede ser un registro de 16 o 64 bits o una posición de memoria de 16 o 64 bits.

Operación

destino = M[RSP]

La instrucción es equivalente a:

```
MOV destino, [RSP]
ADD RSP, <tamaño del operando>
```

Por ejemplo:

```
POP RAX
```

es equivalente a:

```
MOV RAX, [RSP]
ADD RSP, 8
```

Bits de resultado modificados

Ninguno

Formatos válidos

```
POP reg
POP tamaño mem
```

El tamaño del operando ha de ser de 16 o 64 bits.

Ejemplos

```
POP AX
POP RAX
POP WORD [variable]
POP QWORD [RBX]
```

6.24. PUSH: introducir un valor en la pila

PUSH fuente

Se actualiza el valor del registro apuntador de pila, RSP, decrementándolo en tantas unidades como el tamaño en bytes del operando fuente.

A continuación, se introduce el valor del operando fuente en la cima de la pila, se copia el valor del operando a la posición de la memoria apuntada por el registro RSP y se colocan tantos bytes en la pila como el tamaño del operando indicado.

El operando puede ser un registro de 16 o 64 bits, una posición de memoria de 16 o 64 bits o un valor inmediato de 8, 16 o 32 bits extendido a 64 bits.

Operación

M[RSP]=fuente

La instrucción es equivalente a:

```
SUB RSP, <tamaño del operando>
MOV [RSP], fuente
```

Por ejemplo:

```
PUSH RAX
```

es equivalente a:

```
SUB RSP, 8
 MOV [RSP], RAX
```

Bits de resultado modificados

Ninguno

Formatos válidos

```
PUSH reg
PUSH tamaño mem
```

El operando ha de ser de 16 o 64 bits.

```
PUSH imm
```

El valor inmediato puede ser de 8, 16 o 32 bits.

Ejemplos

```
PUSH AX
PUSH RAX
PUSH WORD [variable]
PUSH QWORD [RBX]
PUSH 0Ah
PUSH 0A0Bh
PUSH 0A0B0C0Dh
```

6.25. RET: retorno de subrutina

```
RET
```

Sale de la subrutina que se estaba ejecutando y retorna al punto donde se había hecho la llamada, a la instrucción siguiente de la instrucción CALL.

Extrae de la pila la dirección de memoria de retorno (la dirección de la instrucción que sigue en secuencia a la instrucción CALL) y la carga en el RIP (*instruction pointer*).

Actualiza el puntero de pila (registro RSP), para que apunte al siguiente elemento de la pila; como la dirección de retorno es de 8 bytes (en modo de 64 bits), incrementa RSP en 8 unidades.

Operación

RIP = M(RSP)
RSP = RSP + 8

Bits de resultado modificados

Ninguno

Formatos válidos

```
RET
```

Ejemplo

```
RET
```

6.26. ROL: rotación a la izquierda

```
ROL destino, fuente
```

Lleva a cabo una rotación de los bits del operando destino a la izquierda, es decir, hacia al bit más significativo; rota tantos bits como indica el operando fuente.

Los bits pasan de la posición que ocupan a la posición de su izquierda; el bit de la posición más significativa pasa a la posición menos significativa del operando.

Por ejemplo, en un operando de 32 bits, el bit 0 pasa a ser el bit 1, el bit 1, a ser el bit 2 y así sucesivamente hasta el bit 30, que pasa a ser el bit 31; el bit más significativo (bit 31) se convierte en el bit menos significativo (bit 0).

El operando fuente solo puede ser un valor inmediato de 8 bits o el registro CL.

Si el operando destino es de 64 bits, se enmascarán los dos bits de más peso del operando fuente, lo que permite rotaciones de 0 a 63 bits.

Si el operando destino es de 32 bits o menos, se enmascarán los tres bits de más peso del operando fuente, lo que permite rotaciones de 0 a 31 bits.

Bits de resultado modificados

OF, CF

El bit más significativo se copia en el bit de transporte (CF) cada vez que se desplaza un bit.

Si el operando fuente vale 1, OF se activa si el signo del operando destino original es diferente del signo del resultado obtenido; en cualquier otro caso, OF queda indefinido.

Formatos válidos

```
ROL reg, CL
ROL reg, imm8
ROL tamaño mem, CL
ROL tamaño mem, imm8
```

Ejemplos

```
ROL RAX,CL
ROL RAX,1
ROL DWORD [RBX],CL
ROL QWORD [variable],4
```

6.27. ROR: rotación a la derecha

ROR destino, fuente

Realiza una rotación de los bits del operando destino a la derecha, es decir, hacia al bit menos significativo; rota tantos bits como indica el operando fuente.

Los bits pasan desde la posición que ocupan a la posición de su derecha; el bit de la posición menos significativa pasa a la posición más significativa del operando.

Por ejemplo, en un operando de 32 bits, el bit 31 pasa a ser el bit 30, el bit 30, a ser el bit 29, y así sucesivamente hasta el bit 1, que pasa a ser el bit 0; el bit menos significativo (bit 0) se convierte en el bit más significativo (bit 31).

El operando fuente solo puede ser un valor inmediato de 8 bits o el registro CL.

Si el operando destino es de 64 bits, se enmascaran los dos bits de más peso del operando fuente, lo que permite rotaciones de 0 a 63 bits.

Si el operando destino es de 32 bits o menos, se enmascaran los tres bits de más peso del operando fuente, lo que permite rotaciones de 0 a 31 bits.

Bits de resultado modificados

OF, CF

El bit menos significativo se copia en el bit de transporte (CF) cada vez que se desplaza un bit.

Si el operando fuente vale 1, OF se actualiza con el resultado de la XOR de los dos bits más significativos del resultado; en cualquier otro caso, OF queda indefinido.

Formatos válidos

```
ROR reg, CL
ROR reg, imm8
ROR tamaño mem, CL
ROR tamaño mem, imm8
```

Ejemplos

```
ROR RAX,CL
ROR RAX,1
ROR DWORD [RBX],CL
ROR QWORD [variable],4
```

6.28. SAL: desplazamiento aritmético (o lógico) a la izquierda

SAL destino, fuente

Lleva a cabo un desplazamiento a la izquierda de los bits del operando destino; desplaza tantos bits como indica el operando fuente.

Los bits pasan de la posición que ocupan a la posición de su izquierda y se van añadiendo ceros por la derecha; el bit más significativo se traslada al bit de transporte (CF).

El operando fuente solo puede ser un valor inmediato de 8 bits o el registro CL.

Si el operando destino es de 64 bits, se enmascaran los dos bits de más peso del operando fuente, lo que permite desplazamientos de 0 a 63 bits.

Si el operando destino es de 32 bits o menos, se enmascaran los tres bits de más peso del operando fuente, lo que permite desplazamientos de 0 a 31 bits.

La operación es equivalente a multiplicar por 2 el valor del operando destino tantas veces como indica el operando fuente.

Operación

destino = destino * 2^N, donde N es el valor del operando fuente

Bits de resultado modificados

OF, SF, ZF, PF, CF

CF recibe el valor del bit más significativo del operando destino cada vez que se desplaza un bit.

Si el operando fuente vale 1, OF se activa si el signo del operando destino original es diferente del signo del resultado obtenido; en cualquier otro caso, OF queda indefinido.

El resto de los bits se modifican según el resultado de la operación.

Formatos válidos

```
SAL reg, CL
SAL reg, imm8
SAL tamaño mem, CL
SAL tamaño mem, imm8
```

Ejemplos

```
SAL RAX,CL
SAL RAX,1
SAL DWORD [RBX],CL
SAL QWORD [variable],4
```

6.29. SAR: desplazamiento aritmético a la derecha

SAR destino, fuente

Lleva a cabo un desplazamiento a la derecha de los bits del operando destino; desplaza tantos bits como indica el operando fuente.

Los bits pasan de la posición que ocupan a la posición de su derecha; el bit de signo (el bit más significativo) se va copiando a las posiciones de la derecha; el bit menos significativo se copia al bit de transporte (CF).

El operando fuente solo puede ser un valor inmediato de 8 bits o el registro CL.

Si el operando destino es de 64 bits, se enmascaran los dos bits de más peso del operando fuente, lo que permite desplazamientos de 0 a 63 bits.

Si el operando destino es de 32 bits o menos, se enmascaran los tres bits de más peso del operando fuente, lo que permite desplazamientos de 0 a 31 bits.

La operación es equivalente a dividir por 2 el valor del operando destino tantas veces como indica el operando fuente.

Operación

destino = destino/2^N, donde N es el valor del operando fuente.

Bits de resultado modificados

OF, SF, ZF, PF, CF

Si el operando fuente vale 1, OF se actualiza con el resultado de la XOR de los dos bits más significativos del resultado; en cualquier otro caso, OF queda indefinido.

CF recibe el valor del bit menos significativo del operando destino, cada vez que se desplaza un bit.

El resto de los bits se modifican según el resultado de la operación.

Formatos válidos

```
SAR reg,CL
SAR reg, imm8
SAR tamaño mem, CL
SAR tamaño mem, imm8
```

Ejemplos

```
SAR RAX,CL
SAR RAX,1
SAR DWORD [RBX],CL
SAR QWORD [variable],4
```

6.30. SBB: resta con transporte (*borrow*)

SBB destino, fuente

Lleva a cabo una resta considerando el valor del bit de transporte (CF). Se resta el valor del operando fuente del operando destino, a continuación se resta del resultado el valor de CF y el resultado final de la operación se guarda sobre el operando destino sobreescribiendo el valor inicial. El valor del operando fuente no se modifica.

Operación

destino = destino – fuente – CF

Bits de resultado modificados

OF, SF, ZF, AF, PF, CF

El bit de resultado CF toma el valor 1 si el resultado de la operación es negativo; el resto de los bits de resultado se modifican según el resultado de la operación.

Formatos válidos

```
SBB reg, reg
SBB reg, tamaño mem
SBB tamaño mem, reg
```

Los dos operandos han de ser del mismo tamaño.

```
SBB reg, imm
SBB tamaño mem, imm
```

El tamaño del inmediato puede ir desde 8 bits hasta el tamaño del primer operando, como máximo 32 bits.

Ejemplos

```
SBB R9,RAX
SBB RAX, QWORD [variable]
SBB DWORD [variable],EAX
SBB RAX,0x01020304
SBB BYTE [vector+RAX], 5
```

6.31. SHL: desplazamiento lógico a la izquierda

Es equivalente al desplazamiento aritmético a la izquierda (podéis consultar la instrucción SAL).

6.32. SHR: desplazamiento lógico a la derecha

```
SHR destino, fuente
```

Realiza un desplazamiento a la derecha de los bits del operando destino; desplaza tantos bits como indica el operando fuente.

Los bits pasan de la posición que ocupan a la posición de su derecha, el bit menos significativo se copia en el bit de transporte (CF) y se van añadiendo ceros a la izquierda.

El operando fuente solo puede ser un valor inmediato de 8 bits o el registro CL.

Bits de resultado modificados

OF, SF, ZF, PF, CF

CF recibe el valor del bit menos significativo del operando destino, cada vez que se desplaza un bit.

Si el operando fuente vale 1, OF se actualiza con el resultado de la XOR de los dos bits más significativos del resultado; en cualquier otro caso, OF queda indefinido.

El resto de los bits se modifican según el resultado de la operación.

Formatos válidos

```
SHR reg, CL
SHR reg, imm8
SHR tamaño mem, CL
SHR tamaño mem, imm8
```

Ejemplos

```
SHR RAX,CL
SHR RAX,1
SHR DWORD [RBX],CL
SHR QWORD [variable],4
```

6.33. SUB: resta sin transporte

SUB destino, fuente

Lleva a cabo una resta sin considerar el valor del bit de transporte (CF). Se resta el valor del operando fuente del operando destino, el resultado de la operación se guarda sobre el operando destino sobreescribiendo el valor inicial. El valor del operando fuente no se modifica.

Operación

destino = destino – fuente

Bits de resultado modificados

OF, SF, ZF, AF, PF, CF

El bit de resultado SF toma el valor 1 si el resultado de la operación es negativo; el resto de los bits de resultado se modifican según el resultado de la operación.

Formatos válidos

```
SUB reg, reg
SUB reg, tamaño mem
SUB tamaño mem, reg
```

Los dos operandos han de ser del mismo tamaño.

```
SUB reg, imm
SUB tamaño mem, imm
```

El tamaño del inmediato puede ir desde 8 bits hasta el tamaño del primer operando, como máximo 32 bits.

Ejemplos

```
SUB R9,RAX
SUB RAX, QWORD [variable]
SUB DWORD [variable],EAX
SUB RAX,0x01020304
SUB BYTE [vector+RAX], 5
```

6.34. TEST: comparación lógica

TEST destino, fuente

Realiza una operación lógica 'y' bit a bit entre los dos operandos sin modificar el valor de ninguno de los operandos; actualiza los bits de resultado según el resultado de la 'y' lógica.

Operación

destino AND fuente

Bits de resultado modificados

OF, SF, ZF, AF, PF, CF

Los bits de resultado CF y OF toman el valor 0, el resto de los bits de resultado se modifican según el resultado de la operación.

Formatos válidos

```
TEST reg, reg
```

```
TEST reg, tamaño mem
TEST tamaño mem, reg
```

Los dos operandos han de ser del mismo tamaño.

```
TEST reg, imm
TEST tamaño mem, imm
```

El tamaño del inmediato puede ir desde 8 bits hasta el tamaño del primer operando, como máximo 32 bits.

Ejemplos

```
TEST R9,RAX
TEST RAX, QWORD [variable]
TEST DWORD [variable],EAX
TEST RAX,0x01020304
TEST BYTE [vector+RAX], 5
```

6.35. XCHG: intercambio de operandos

```
XCHG destino, fuente
```

Se lleva a cabo un intercambio entre los valores de los dos operandos. El operando destino toma el valor del operando fuente, y el operando fuente toma el valor del operando destino.

No se puede especificar el mismo operando como fuente y destino, ni ninguno de los dos operandos puede ser un valor inmediato.

Bits de resultado modificados

No se modifica ningún bit de resultado.

Formatos válidos

```
XCHG reg, reg
XCHG reg, tamaño mem
XCHG tamaño mem, reg
```

Los dos operandos han de ser del mismo tamaño.

Ejemplos

```
XCHG R9, RAX
XCHG RAX, QWORD [variable]
XCHG DWORD [variable], EAX
```

6.36. XOR: o exclusiva

XOR destino, fuente

Realiza una operación lógica XOR ('o exclusiva') bit a bit entre el operando destino y el operando fuente; el resultado de la operación se guarda sobre el operando destino sobreescribiendo el valor inicial. El valor del operando fuente no se modifica.

Se lleva a cabo una operación XOR entre el bit *n* del operando destino y el bit *n* del operando fuente según la tabla de verdad de la función XOR:

```
x     y      x XOR y
---   ---    -------
0     0       0
0     1       1
1     0       1
1     1       0
```

Operación

destino = destino XOR fuente

Bits de resultado modificados

OF=0, SF, ZF, PF, CF=0

Los indicadores OF y CF se ponen a 0; el resto de los indicadores se modifican según el resultado de la operación.

Formatos válidos

```
XOR reg,reg
XOR reg,tamaño mem
XOR tamaño mem,reg
```

Los dos operandos han de ser del mismo tamaño.

```
XOR reg,imm
XOR tamaño mem,imm
```

El tamaño del inmediato puede ir desde 8 bits hasta el tamaño del primer operando, como máximo 32 bits.

Ejemplos

```
XOR R9, RAX
XOR RAX, QWORD [variable]
XOR DWORD [variable], EAX
XOR RAX, 01020304 h
XOR BYTE [vector+RAX], 5
```

La arquitectura CISCA

Miquel Albert Orenga
Gerard Enrique Manonellas

PID_00181526

UOC
Universitat Oberta de Catalunya
www.uoc.edu

Índice

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.

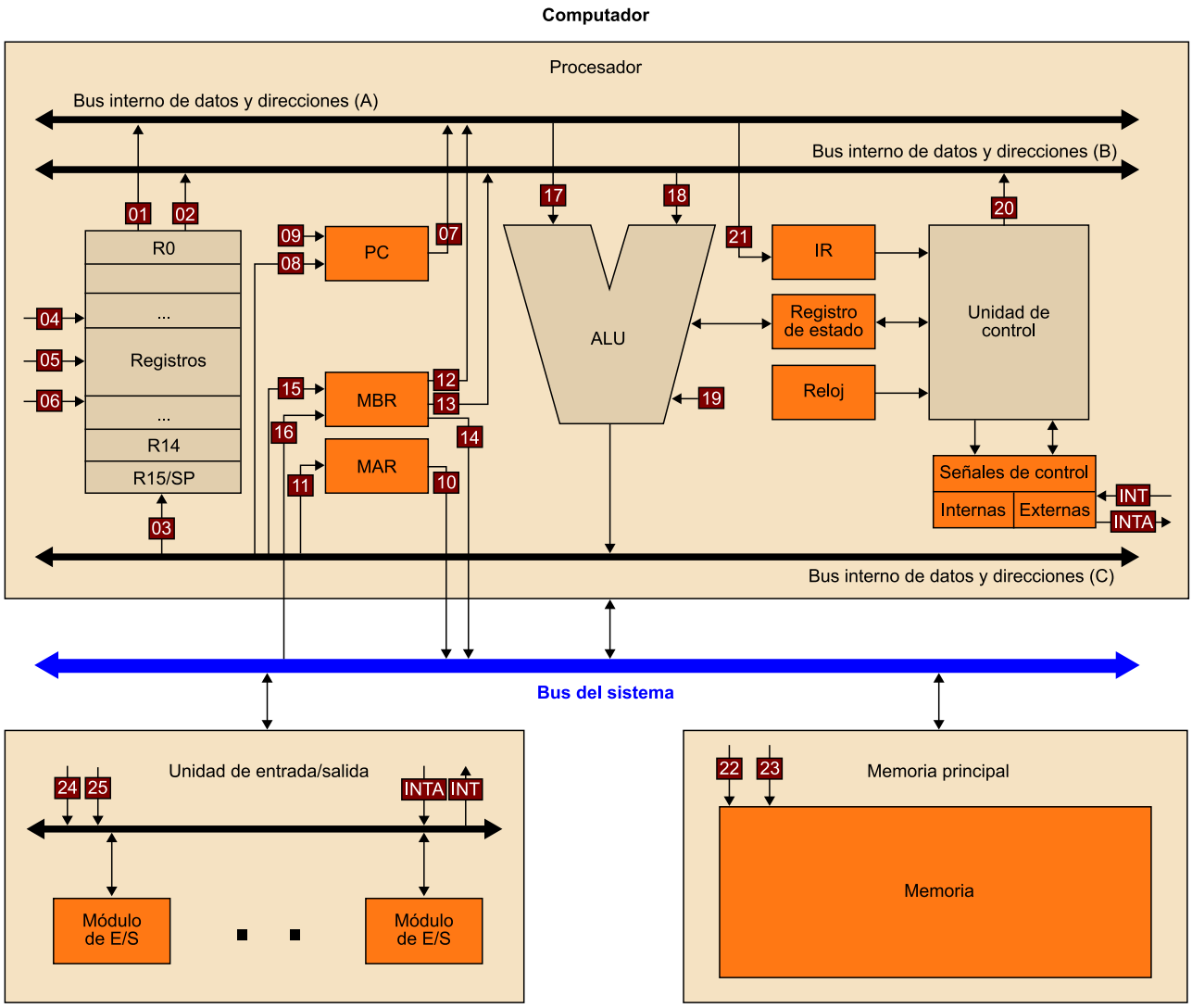
1. Organización del computador

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 15 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).

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.

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 siguientes bits de resultado:

Bits de resultado activos

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

- **Bit de cero (Z)**: se activa si el resultado obtenido es 0.
- **Bit de transporte (C)**: también denominado *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; también puede deberse a una operación de desplazamiento. Se activa si al final de la operación nos llevamos una según el algoritmo de suma y resta tradicional o si el último bit que desplazamos se copia sobre el bit de transporte y este es 1.
- **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.
- **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 = 32 operaciones diferentes	
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		

	Señal	Observaciones	Dispositivos afectados
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(AF10350Ah).
```

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 fuente. 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.

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.

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 ≠ 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 ≠ 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.

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)
Indirecto (direccionamiento indirecto a registro)	Inmediato
	Registro (direccionamiento directo a registro)
Relativo (direccionamiento relativo a registro base)	Inmediato

Operando destino	Operando fuente
	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
31h	OR
32h	XOR
33h	TEST
34h	NOT

B0 (Código de operación)	Instrucción
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)

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

Bk<7..4>	Modo de direccionamiento
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 0001Bh	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	EFh	00h	ABh	00h
[vector1+R9] La etiqueta *vector1* vale 0ABC0100h	0ABC0100h y 9h	5h	9h	EFh	00h	ABh	00h

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}).

$$desp16 = etiqueta - PC_{up}$$

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 R13

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 1FF00030h y PC_{up} vale 1FF000B4h.

- 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 R13	11	1D									
JNE etiqueta2	42	60	84	00							
CALL etiqueta2	47	00	30	00	F0	1F					
NOT [etiqueta1]	34	20	5B	00	12	00					
DEC [R4]	25	34									
XOR [R13 + 3F4Ah], R12	32	4D	4A	3F	1C						

	Bk para k=0..10										
Ensamblador	**B0**	**B1**	**B2**	**B3**	**B4**	**B5**	**B6**	**B7**	**B8**	**B9**	**B10**
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.

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 (1, 2, 3 o 4)
IR  ← MBR         ; cargamos la instrucción en el registro IR.
```

Hay que tener presente que si la instrucción posee un tamaño superior a una palabra de memoria (4 bytes), deberemos repetir el proceso de lectura a memoria las veces necesarias para leer toda la instrucción. Pero para simplificar el funcionamiento de la unidad de control consideraremos que haciendo un solo acceso a memoria leemos toda 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 siguientes fases.

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:

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.

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*.

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 ← RI(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 ← RI(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

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:

Operando destino

Recordad que el operando destino también es operando fuente.

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

```
Ri ← Ri<operación>RI(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>RI(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
```

Ejemplo

Si estamos ejecutando la instrucción ADD R3, 7 la micro-operación que se ejecutaría en esta fase sería: R3 ← R3 + IR(valor inmediato)=7.

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 activo (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.

Si codificamos estas instrucciones en el formato CISCA, veremos que algunas ocupan más de 4 bytes. Como a la memoria se accede siempre con palabras de 4 bytes, para leer toda la instrucción habrá que hacer 2 o más accesos a la memoria. Para simplificar los ejemplos, no consideraremos los siguientes accesos (repetir pasos 1, 2, 3 y 4 de la fase 1).

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

Fase	Microoperación
1	MAR ← PC, read MBR ← Memoria PC ← PC + 4 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)

Fase	Microoperación
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 + 4 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 + 4 IR ← MBR
2	MAR ← 00120034h+R3, read MBR ← Memoria
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 + 3 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; por lo tanto, para leer toda la instrucción deberemos hacer 2 accesos a memoria. Para simplificar el ejemplo, no consideraremos el segundo acceso (repetir pasos 1, 2, 3 y 4 de la fase 1).

Fase	Paso	Microoperación	
1	1	MAR ← PC, read	
	2	MBR ← Memoria	PC ← PC + 4
	3	IR ← MBR	
2	1	MAR ← R1 + 10, read	
	2	MBR ← Memoria	
3	1	R3 ← MBR	
1	1	MAR ← PC, read	Siguiente instrucción

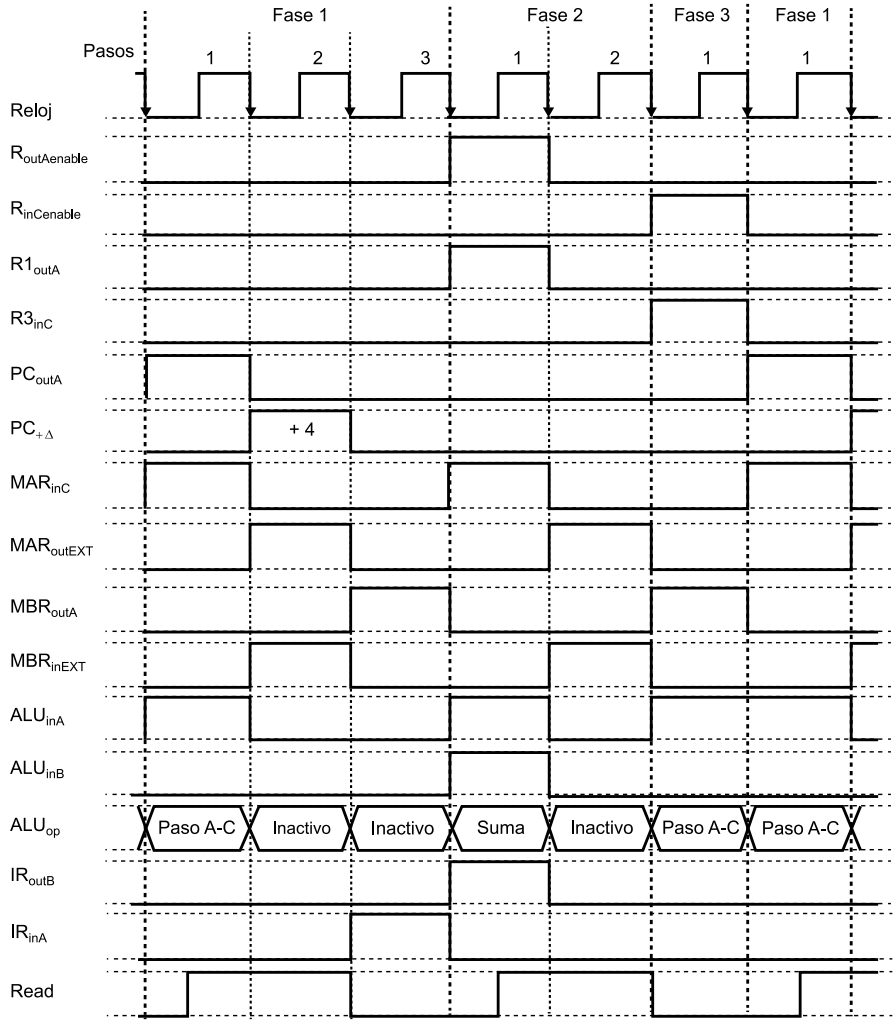
Nota

Consideramos que las microoperaciones

MBR ← Memoria

PC ← PC + 4

se pueden hacer en el mismo paso porque utilizan recursos diferentes del procesador y no se interfieren.



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 + 4

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 en el registro MBR activando la señal MBR_{inEXT}.

Dado que ya tenemos la instrucción en el IR y hemos empezado a descodificar, conocemos su longitud. En este caso es 5, por lo tanto incrementamos el PC en 4, porque solo leemos palabras de 4 bytes. Así, deberíamos hacer una segunda lectura para el quinto byte, que como hemos dicho no consideraremos. Para hacerlo, utilizamos el circuito autoincrementador que tiene el registro PC, activando la señal $PC_{+\Delta}$, que indica un incremento de 4.

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.