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

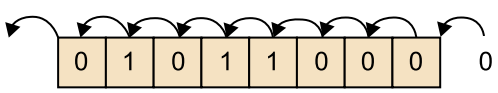
2) Operaciones de desplazamiento y rotación. Dentro de este grupo de instrucciones se incluyen instrucciones de desplazamiento lógico (SHL, SHR), desplazamiento aritmético (SAL, SAR), rotación (ROL, ROR).

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:

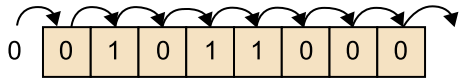
a) Desplazamiento lógico a la izquierda (SHL). Se considera el primer operando como un valor sin signo. Se desplazan los bits a la izquierda tantas posiciones como indica el segundo operando; el bit de más a la izquierda en determinadas arquitecturas se pierde y en otras se copia sobre el bit de transporte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

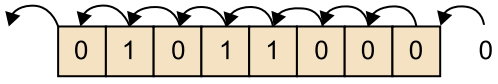
Ejemplo de desplazamiento de un bit a la derecha



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

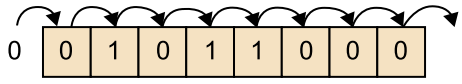


Nota

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

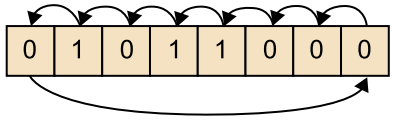


Nota

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

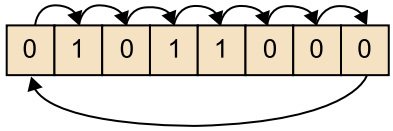
e) Rotación a la izquierda (ROL). Se considera el primer operando como un valor sin signo; se desplazan los bits a la izquierda tantas posiciones como indica el segundo operando, el bit de más a la izquierda se pasa a la posición menos significativa del operando.

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.

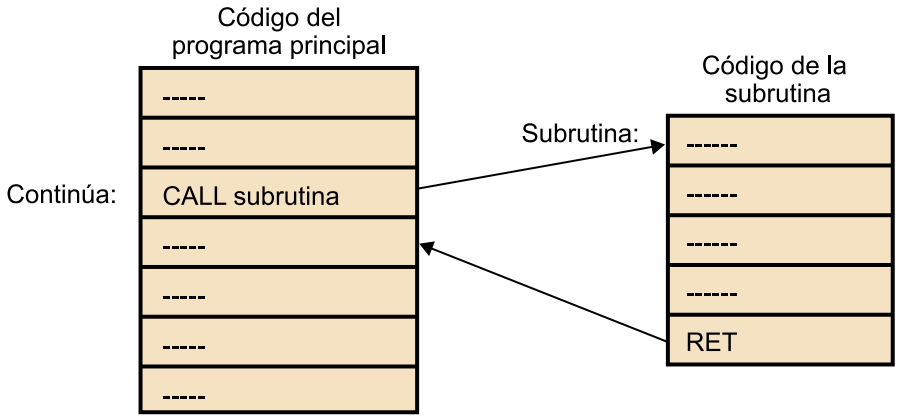
```
iteracion:
        ...
        DSZ R1
        JMP iteración
continuar:
        ...
```

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

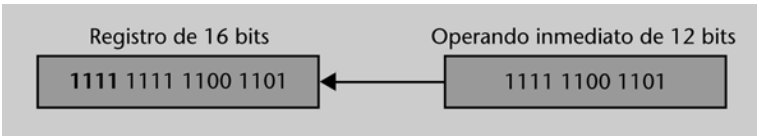
En el direccionamiento inmediato, el operando expresa el valor del dato que se quiere utilizar; es decir, el dato está dentro de la instrucción y su valor es fijo.

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.