Programació en assemblador (x86-64)

Miquel Albert Orenga
Gerard Enrique Manonellas

PID_00218252

UOC
Universitat Oberta de Catalunya

Índex

Introducció

En aquest mòdul ens centrarem en la programació a baix nivell per tal de conèixer les especificacions més rellevants d'una arquitectura real concreta. En el nostre cas l'arquitectura x86-64 (també anomenada *AMD64* o *Intel 64*).

El llenguatge utilitzat per a programar a baix nivell un computador és el llenguatge d'assemblador, però per a facilitar el desenvolupament d'aplicacions i certes operacions d'E/S utilitzarem un llenguatge d'alt nivell, el llenguatge C, d'aquesta manera, podrem organitzar els programes segons les especificacions d'un llenguatge d'alt nivell, que són més flexibles i potents, i implementar certes funcions amb assemblador per a treballar a baix nivell els aspectes més rellevants de l'arquitectura de la màquina.

La importància del llenguatge d'assemblador rau en el fet que és el llenguatge simbòlic que treballa més a prop del processador. Pràcticament totes les instruccions d'assemblador tenen una correspondència directa amb les instruccions binàries del codi màquina que utilitza directament el processador. Això fa que el llenguatge sigui relativament senzill, però que tingui un gran nombre d'excepcions i regles definides per la mateixa arquitectura del processador, i a l'hora de programar, a més de conèixer les especificacions del llenguatge, cal conèixer també les especificacions de l'arquitectura.

Així doncs, aquest llenguatge permet escriure programes que poden aprofitar totes les característiques de la màquina, cosa que facilita la comprensió de l'estructura interna del processador. També pot aclarir algunes de les característiques dels llenguatges d'alt nivell que queden amagades en la seva compilació.

El contingut del mòdul es divideix en set apartats que tracten els següents temes:

- Descripció de l'arquitectura x86-64, des del punt de vista del programador, i en concret del mode de 64 bits, mode en el qual es desenvoluparan les pràctiques de programació.

- Descripció de l'entorn de programació amb què es treballarà: edició, compilació i depuració dels programes en assemblador i en C.

- Descripció dels elements que componen un programa en assemblador.

- Introducció a la programació en llenguatge C.

- Conceptes de programació en llenguatge d'assemblador i en llenguatge C, i com utilitzar funcions escrites en assemblador dins de programes en C.

- També s'inclou una referència de les instruccions més habituals del llenguatge d'assemblador amb exemples d'ús de cadascuna.

Aquest mòdul no pretén ser un manual que expliqui totes les característiques del llenguatge d'assemblador i del llenguatge C, sinó una guia que permeti iniciar-se en la programació a baix nivell (fer-hi programes) ajudant-se d'un llenguatge d'alt nivell com el C.

Objectius

Amb l'estudi d'aquest mòdul es pretén que l'estudiant assoleixi els objectius següents:

1. Aprendre a utilitzar un entorn de programació amb llenguatge C i llenguatge d'assemblador.

2. Conèixer el repertori d'instruccions bàsiques dels processadors de la família x86-64, les diferents formes d'adreçament per a tractar les dades i el control de flux dins d'un programa.

3. Saber estructurar en subrutines un programa i passar i rebre paràmetres.

4. Tenir una idea global de l'arquitectura interna del processador segons les característiques del llenguatge d'assemblador.

1. Arquitectura del computador

En aquest apartat es descriuran a grans trets els modes d'operació i els elements més importants de l'organització d'un computador basat en l'arquitectura x86-64 des del punt de vista del joc d'instruccions utilitzat pel programador.

x86-64 és una ampliació de l'arquitectura x86. L'arquitectura x86 va ser llançada per Intel amb el processador Intel 8086 l'any 1978 com una arquitectura de 16 bits. Aquesta arquitectura d'Intel va evolucionar a una arquitectura de 32 bits quan va aparèixer el processador Intel 80386 l'any 1985, anomenada inicialment *i386* o *x86-32* i finalment *IA-32*. Del 1999 al 2003 AMD va ampliar aquesta arquitectura de 32 bits d'Intel a una de 64 bits i la va anomenar *x86-64* en els primers documents i posteriorment *AMD64*. Intel aviat va adoptar les extensions de l'arquitectura d'AMD sota el nom d'*IA-32e* o *EM64T*, finalment la va anomenar *Intel 64*.

IA-64

Cal notar que, amb el processador Itanium, Intel ha llançat una arquitectura anomenada *IA-64*, derivada de la IA-32 però amb especificacions diferents de l'arquitectura x86-64 i que no és compatible amb el joc d'instruccions des d'un punt de vista natiu de les arquitectures x86, x86-32 o x86-64. L'arquitectura IA-64 es va originar a Hewlett-Packard (HP) i més tard va ser desenvolupada conjuntament amb Intel per a ser utilitzada en els servidors empresarials i sistemes de computació d'alt rendiment.

L'arquitectura x86-64 (AMD64 o Intel 64) de 64 bits dóna un suport molt més gran a l'espai d'adreces virtuals i físiques, proporciona registres de propòsit general de 64 bits i altres millores que coneixerem.

Qualsevol processador actual també disposa d'una sèrie d'unitats específiques per a treballar amb nombres en punt flotant (joc d'instruccions de la FPU), i d'extensions per a treballar amb dades multimèdia (joc d'instruccions MMX i SSE en el cas d'Intel, o 3DNow! en el cas d'AMD). Aquestes parts no es descriuen en aquests materials ja que queden fora de l'abast de l'assignatura.

Ens centrarem, per tant, en les parts relacionades amb el treball amb enters i dades alfanumèriques.

1.1. Modes d'operació

Els processadors amb arquitectura x86-64 mantenen la compatibilitat amb els processadors de l'arquitectura IA-32 (x86-32). Per aquest motiu disposen dels mateixos modes d'operació de l'arquitectura IA-32, que permeten mantenir la compatibilitat i executar aplicacions de 16 i 32 bits, però afegeixen un mode nou anomenat *mode estès* (o mode *IA-32e*, en el cas d'Intel), dins del qual es pot treballar en mode real de 64 bits.

Els processadors actuals suporten diferents modes d'operació, però com a mínim disposen d'un mode protegit i d'un mode supervisor.

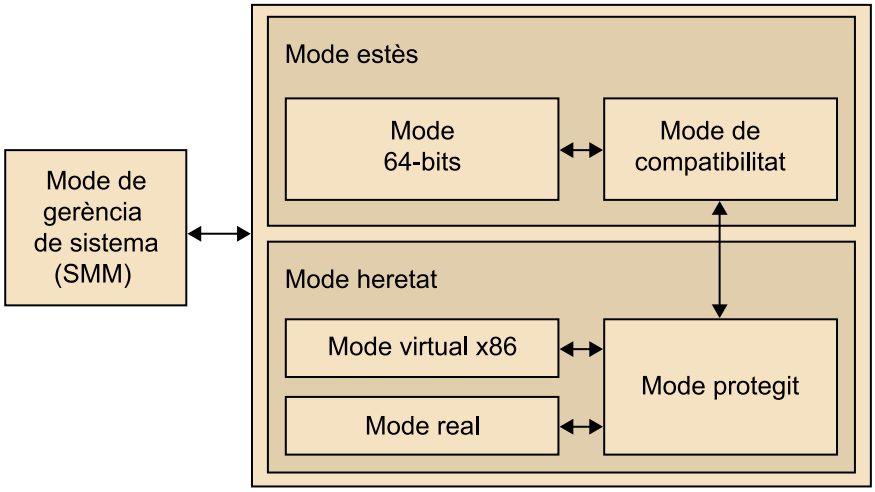
El mode de supervisor és utilitzat pel nucli del sistema per a les tasques de baix nivell que necessiten un accés sense restriccions al maquinari, com pot ser el control de la memòria o la comunicació amb altres dispositius. El mode protegit, en canvi, s'utilitza per a gairebé tota la resta.

Quan executem programes en mode protegit, només podrem utilitzar el maquinari fent crides al sistema operatiu, que és qui el pot controlar en mode supervisor. Hi pot haver altres modes similars al protegit, com el mode virtual, que s'utilitza per a emular altres processadors de la mateixa família, i d'aquesta manera mantenir la compatibilitat amb els processadors anteriors.

Quan un equip s'inicia per primera vegada s'executen els programes de la BIOS, del gestor d'arrencada i del sistema operatiu que tenen accés il·limitat al maquinari; quan l'equip s'ha iniciat, el sistema operatiu pot passar el control a un altre programa i posar el processador en mode protegit.

En mode protegit, els programes tenen accés a un conjunt més limitat d'instruccions i només podran deixar el mode protegit fent una petició d'interrupció que torna el control al sistema operatiu; d'aquesta manera es garanteix el control per a accedir al maquinari.

Modes d'operació de l'arquitectura x86-64



Característiques dels dos modes principals d'operació en l'arquitectura x86-64

Mode d'operació		Sistema operatiu	Les aplicacions necessiten recompilació	Per defecte		Mida dels registres de propòsit general
				Mida (en bits) de les adreces	Mida (en bits) dels operands	
Mode estès	Mode de 64 bits	Sistema operatiu de 64 bits	sí	64	32	64

Mode d'operació		Sistema operatiu	Les aplicacions necessiten recompilació	Per defecte		Mida dels registres de propòsit general
				Mida (en bits) de les adreces	Mida (en bits) dels operands	
	Mode compatibilitat		no	32		32
				16	16	16
Mode heretat	Mode protegit	Sistema operatiu de 32 bits	no	32	32	32
				16	16	
	Mode virtual-8086			16	16	16
	Mode real	Sistema operatiu de 16 bits				

1.1.1. Mode estès de 64 bits

El mode estès de 64 bits és utilitzat pels sistemes operatius de 64 bits. Dins d'aquest mode general es disposa d'un mode d'operació de 64 bits i d'un mode de compatibilitat amb els modes d'operació de les arquitectures de 16 i 32 bits.

En un sistema operatiu de 64 bits, els programes de 64 bits s'executen en mode de 64 bits i les aplicacions de 16 i 32 bits s'executen en mode de compatibilitat. Els programes de 16 i 32 bits que s'hagin d'executar en mode real o virtual x86 no es podran executar en mode estès si no són emulats.

Mode de 64 bits

El mode de 64 bits proporciona accés a 16 registres de propòsit general de 64 bits. En aquest mode s'utilitzen adreces virtuals (o lineals) que per defecte són de 64 bits i es pot accedir a un espai de memòria lineal de 2^{64} bytes.

La mida per defecte dels operands es manté en 32 bits per a la majoria d'instruccions.

La mida per defecte pot ser canviada individualment en cada instrucció mitjançant modificadors. A més a més, suporta adreçament relatiu a PC (RIP en aquesta arquitectura) en l'accés a les dades de qualsevol instrucció.

Mode de compatibilitat

El mode de compatibilitat permet a un sistema operatiu de 64 bits executar directament aplicacions de 16 i 32 bits sense necessitat de recompilar-les.

En aquest mode, les aplicacions poden utilitzar adreces de 16 i 32 bits, i poden accedir a un espai de memòria de 4 Gbytes. La mida dels operands pot ser de 16 i 32 bits.

Des del punt de vista de les aplicacions es veu com si s'estigués treballant en el mode protegit dins del mode heretat.

1.1.2. Mode heretat de 16 i 32 bits

El mode heretat de 16 i 32 bits és utilitzat pels sistemes operatius de 16 i 32 bits. Quan el sistema operatiu utilitza els modes de 16 bits o de 32 bits, el processador actua com un processador x86 i només es pot executar codi de 16 o 32 bits. Aquest mode només permet utilitzar adreces de 32 bits de manera que limita l'espai d'adreces virtual a 4 GB.

Dins d'aquest mode general hi ha tres modes:

1) Mode real. Implementa el mode de programació de l'Intel 8086, amb algunes extensions, com la capacitat de poder passar al mode protegit o al mode de gestió del sistema. El processador es col·loca en mode real en iniciar el sistema i quan es reinicia.

És l'únic mode d'operació que permet utilitzar un sistema operatiu de 16 bits.

El mode real es caracteritza perquè disposa d'un espai de memòria segmentat d'1 MB amb adreces de memòria de 20 bits i accés a les adreces del maquinari (sistema d'E/S). No proporciona suport per a la protecció de memòria en sistemes multitasca ni a codi amb diferents nivells de privilegi.

2) Mode protegit. Aquest és el mode per defecte del processador. Permet utilitzar característiques com la memòria virtual, la paginació o la computació multitasca.

Entre les capacitats d'aquest mode hi ha la possibilitat d'executar codi en mode real, mode virtual-8086, en qualsevol tasca en execució.

3) Mode virtual 8086. Aquest mode permet executar programes de 16 bits com a tasques dins del mode protegit.

1.1.3. El mode de gestió de sistema

El mode de gestió de sistema o system management mode (SMM) és un mode d'operació transparent al programari convencional (sistema operatiu i aplicacions). En aquest mode se suspèn l'execució normal (incloent el sistema operatiu) i s'executa un programari especial d'alt privilegi dissenyat per a controlar el sistema. Tasques habituals d'aquest mode són la gestió d'energia, tasques de depuració assistides per maquinari, execució de microprogramari o un programari assistit per maquinari. Aquest mode és utilitzat bàsicament per la BIOS i pels controladors de dispositiu de baix nivell.

Accedim al SMM mitjançant una interrupció de gestió del sistema (SMI, *system management interrupt*). Una SMI pot ser generada per un esdeveniment independent o disparada pel programari del sistema per l'accés a una adreça d'E/S considerada especial per la lògica de control del sistema.

1.2. El mode de 64 bits

Nota

És important per al programador de baix nivell conèixer les característiques més rellevants d'aquest mode, ja que serà el mode en què es desenvoluparan les pràctiques de programació.

En aquest subapartat analitzarem amb més detall les característiques més importants del mode de 64 bits de l'arquitectura x86-64.

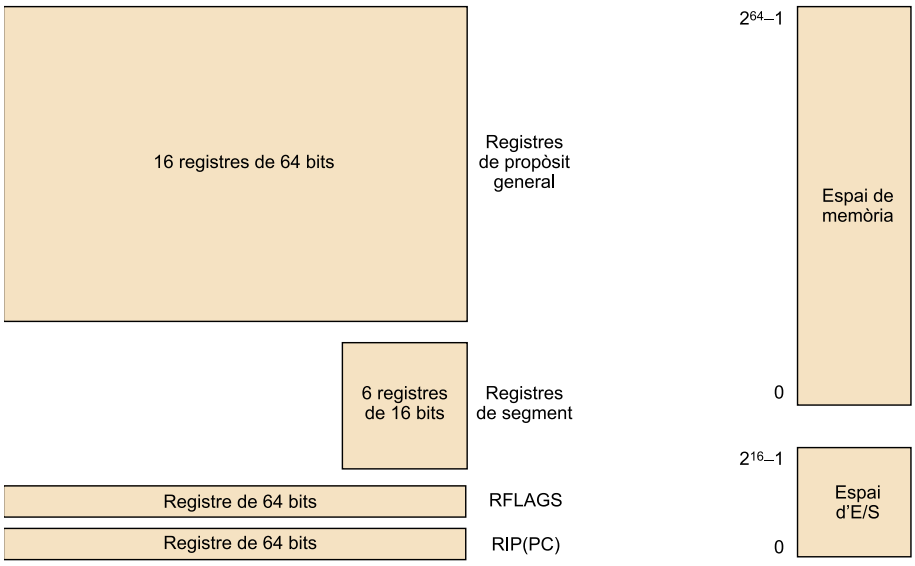
Els elements que des del punt de vista del programador són visibles en aquest mode d'operació són els següents:

1) Espai de memòria: un programa en execució en aquest mode pot accedir a un espai d'adreces lineal de 2^{64} bytes. L'espai físic que realment pot adreçar el processador és inferior i depèn de la implementació concreta de l'arquitectura.

2) Registres: hi ha 16 registres de propòsit general de 64 bits, que suporten operacions de byte (8 bits), word (16 bits), double word (32 bits) i quad word (64 bits).

- El registre comptador de programa (RIP, *instruction pointer register*) és de 64 bits.
- El registre de bits d'estat també és de 64 bits (RFLAGS). Els 32 bits de la part alta estan reservats; els 32 bits de la part baixa són accessibles, corresponen als mateixos bits de l'arquitectura IA-32 (registre EFLAGS).
- Els registres de segment en general no s'utilitzen en el mode de 64 bits.

Entorn d'execució del mode de 64 bits



1.2.1. Organització de la memòria

El processador accedeix a la memòria utilitzant adreces físiques de memòria. La mida de l'espai d'adreces físic accessible pels processadors depèn de la implementació: supera els 4 Gbytes, però és inferior als 2^{64} bytes possibles.

En el mode de 64 bits, l'arquitectura proporciona suport a un espai d'adreces virtual o lineal de 64 bits (adreces de 0 a $2^{64} - 1$), però com que l'espai d'adreces físic és inferior a l'espai d'adreces lineal és necessari un mecanisme de correspondència entre les adreces lineals i les adreces físiques (mecanisme de paginació).

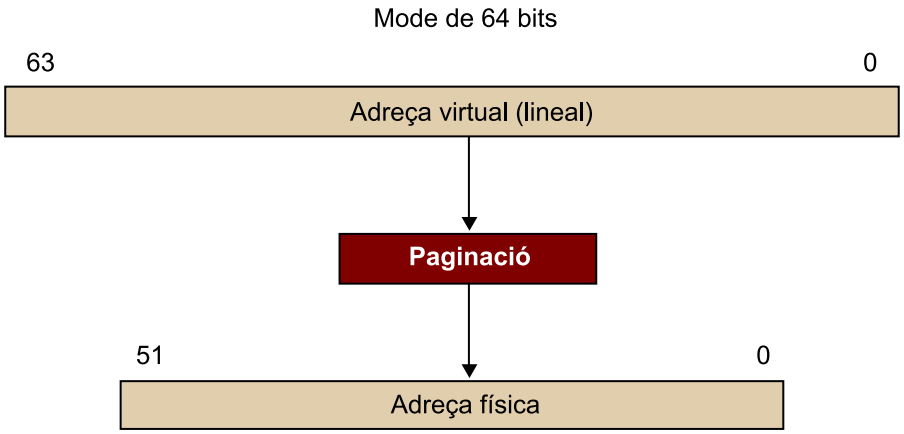
En treballar en un espai lineal d'adreces no s'utilitzen mecanismes de segmentació de la memòria, de manera que no són necessaris els registres de segments, tret dels registres de segment FS i GS, que es poden utilitzar com a registre base en el càlcul d'adreces dels modes d'adreçament relatiu.

Paginació

Aquest mecanisme és transparent als programes d'aplicació, i per tant al programador, i és gestionat pel maquinari del processador i el sistema operatiu.

Les adreces virtuals són traduïdes a adreces físiques de memòria utilitzant un sistema jeràrquic de taules de traducció gestionades pel programari del sistema (sistema operatiu).

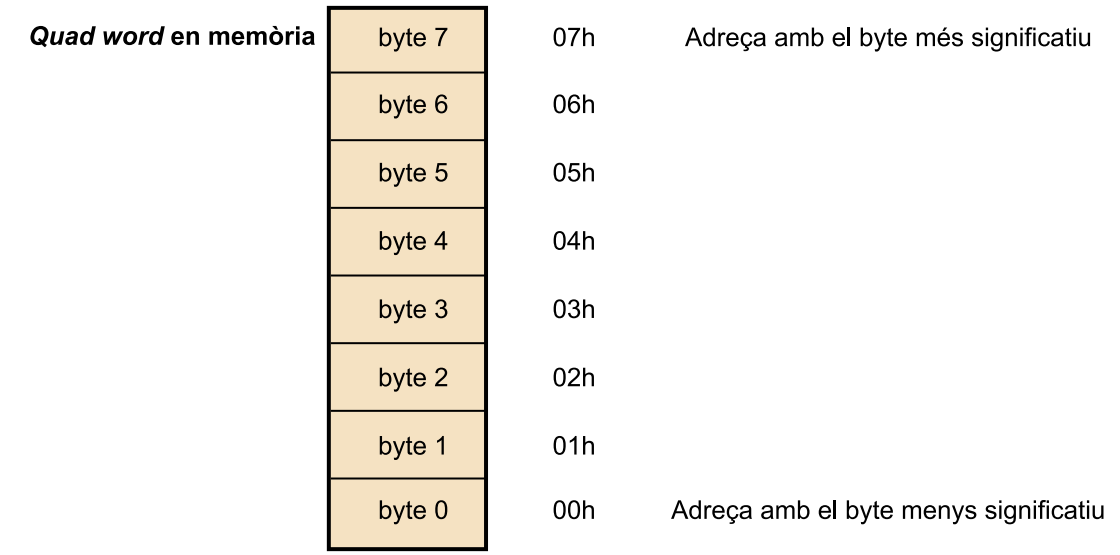
Bàsicament una adreça virtual es divideix en camps, i cada camp actua com a índex dins d'una de les taules de traducció. Cada valor en la posició indexada actua com a adreça base de la taula de traducció següent.

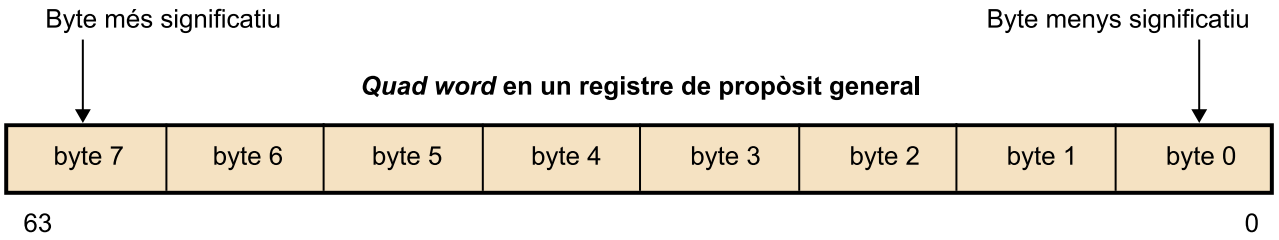


Ordre dels bytes

Els processadors x86-64 utilitzen un sistema d'ordenació dels bytes quan s'accedeix a dades que es troben emmagatzemades a la memòria. En concret s'utilitza un sistema *little-endian*; en aquest sistema el byte de menys pes d'una dada ocupa l'adreça més baixa de memòria.

En els registres també s'utilitza l'ordre *little-endian* i per aquest motiu el byte menys significatiu d'un registre s'anomena *byte 0*.





Mida de les adreces

Els programes que s'executen en el mode de 64 bits generen directament adreces de 64 bits.

Mode compatibilitat

Els programes que s'executen en el mode compatibilitat generen adreces de 32 bits. Aquestes adreces són esteses afegint zeros als 32 bits més significatius de l'adreça. Aquest procés és gestionat pel maquinari del processador i és transparent al programador.

Mida dels desplaçaments i dels valors immediats

En el mode de 64 bits els desplaçaments utilitzats en els adreçaments relatius i els valors immediats són sempre de 32 bits, però són estesos a 64 bits mantenint el signe.

Hi ha una excepció a aquest comportament: en la instrucció MOV es permet especificar un valor immediat de 64 bits.

1.2.2. Registres

Els processadors de l'arquitectura x86-64 disposen d'un banc de registres format per registres de propòsit general i registres de propòsit específic.

De registres de propòsit general n'hi ha 16 de 64 bits, i de propòsit específic hi ha 6 registres de segment de 16 bits, un registre d'estat de 64 bits (RFLAGS) i un registre comptador de programa també de 64 bits (RIP).

Registres de propòsit general

Són 16 registres de dades de 64 bits (8 bytes): RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP i R8-R15.

Els 8 primers registres s'anomenen de manera semblant als 8 registres de propòsit general de 32 bits disponibles en l'arquitectura IA-32 (EAX, EBX, ECX, EDX, ESI, EDI, EBP i ESP).

Els registres són accessibles de quatre maneres diferents:

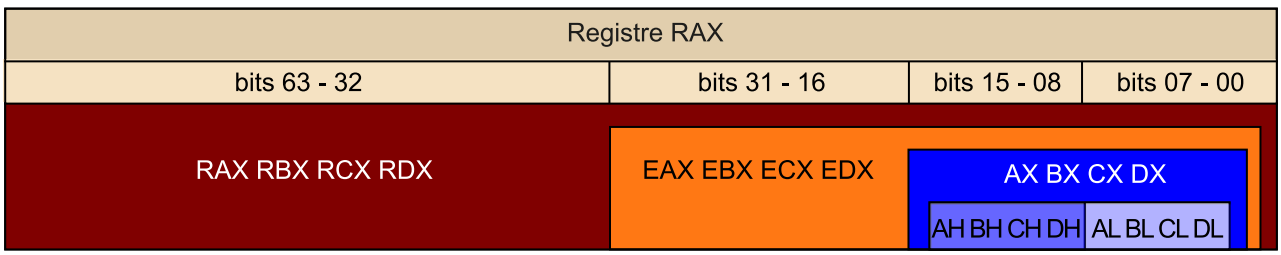
1) Com a registres complets de 64 bits (quad word).

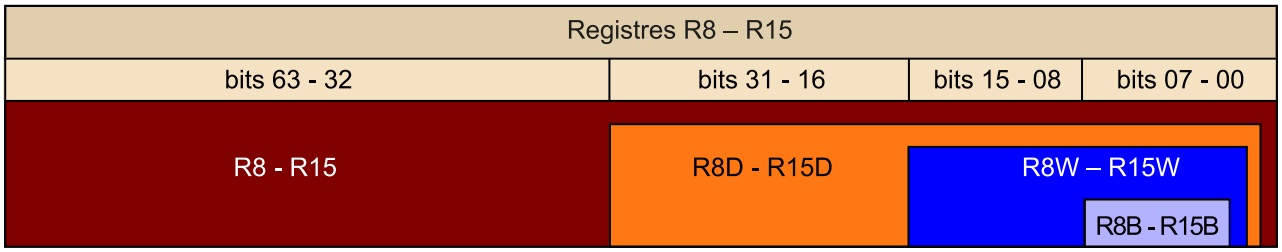
2) Com a registres de 32 bits (double word), accedint als 32 bits de menys pes.

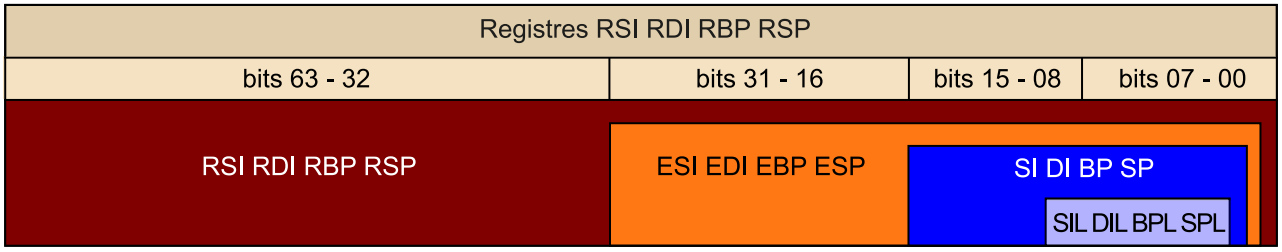
3) Com a registres de 16 bits (word), accedint als 16 bits de menys pes.

4) Com a registres de 8 bits (byte), permetent accedir individualment a un, o dos, dels bytes de menys pes depenent del registre.

Els registres de propòsit general es divideixen en parts (i es dóna un nom a cada part), cosa que facilita treballar amb diferents tipus de dades.







Exemple

RAX[63-0]				Registre de 8 bytes
EAX[31-0]	= RAX[31-0]			Registre de 4 bytes
AX[15-0]	= EAX[15-0]	= RAX[15-0]		Registre de 2 bytes
AH[7-0]	= AX[15-8]	= EAX[15-8]	= RAX[15-8]	Registre d'1 byte
AL[7-0]	= AX[7-0]	= EAX[7-0]	= RAX[7-0]	Registre d'1 byte

Els registres EAX, AX, AH, AL no són registres independents, sinó que són parts del registres RAX. Això passa amb tots els registres de propòsit general: RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, R8, R9, R10, R11, R12, R13, R14 i R15.

Hi ha algunes limitacions en l'ús dels registres de propòsit general:

- En una mateixa instrucció no es pot usar un registre del conjunt AH, BH, CH, DH juntament amb un del conjunt SIL, DIL, BPL, SPL, R8B – R15B.

- Registre RSP: té una funció especial, funciona com a apuntador de pila, conté sempre l'adreça del primer element de la pila. Si l'utilitzem amb altres finalitats perdrem l'accés a la pila.

- Quan s'utilitza un registre de 32 bits com a operand destinació d'una instrucció, la part alta del registre és fixada a 0.

Registres de propòsit específic

Podem distingir diversos registres de propòsit específic:

1) Registres de segment: hi ha 6 registres de segment de 16 bits.

- CS: *code segment*

- DS: *data segment*
- SS: *stack segment*
- ES: *extra segment*
- FS: *extra segment*
- GS: *extra segment*

Aquests registres s'utilitzen bàsicament en els models de memòria segmentat (heretat de l'arquitectura IA-32). En aquests models, la memòria es divideix en segments, de manera que en un moment donat el processador només és capaç d'accedir a sis segments de la memòria utilitzant cadascun dels sis registres de segment.

En el mode de 64 de bits, aquests registres pràcticament no s'utilitzen, ja que es treballa amb el model de memòria lineal, i el valor d'aquests registres es troba fixat a 0 (a excepció dels registres FS i GS que poden ser utilitzat com a registres base en el càlcul d'adreces).

2) Registre d'instrucció o *Instruction pointer* (RIP): és un registre de 64 bits que actua com a registre comptador de programa (PC), conté l'adreça efectiva (o adreça lineal) de la instrucció següent que s'ha d'executar.

Cada cop que es llegeix una instrucció nova de la memòria s'actualitza amb l'adreça de la instrucció següent que s'ha d'executar; també es pot modificar el contingut del registre durant l'execució d'una instrucció de ruptura de seqüència (crida a subrutina, salt condicional o incondicional).

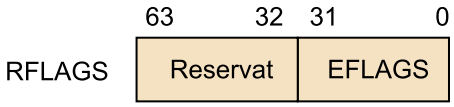
3) Registre d'estat o *Flags register* (RFLAGS): és un registre de 64 bits que conté informació sobre l'estat del processador i informació sobre el resultat de l'execució de les instruccions.

Només s'utilitza la part baixa del registre (bits 31 a 0), que correspon al registre EFLAGS de l'arquitectura IA-32; la part alta del registre està reservada i no s'utilitza.

Bits de resultat

Les instruccions de salt condicional consulten alguns dels bits del registre, anomenats *bits de resultat*. Hi ha més informació sobre el tema al manual bàsic del joc d'instruccions en l'apartat 7 d'aquest mòdul.

L'ús habitual del registre d'estat consisteix a consultar el valor individual dels seus bits; això es pot aconseguir amb instruccions específiques, com, per exemple, les instruccions de salt condicional que consulten un bit o més per a determinar si salten o no segons com han deixat aquests bits la darrera instrucció que els ha modificat (no ha de ser la darrera instrucció que s'ha executat).



A la següent taula es descriu el significat dels bits d'aquest registre.

bit 0	CF	Carry Flag	0 = no s'ha produït transport; 1 = sí que s'ha produït transport
bit 1		No definit	
bit 2	PF	Parity Flag	0 = nombre de bits 1 és senar; 1 = nombre de bits 1 és parell
bit 3		No definit	
bit 4	AF	Auxiliary Carry Flag	0 = no s'ha produït transport en operacions BCD; 1 = sí que s'ha produït transport en operacions BCD
bit 5		No definit	
bit 6	ZF	Zero Flag	0 = el resultat no ha estat zero; 1 = el resultat ha estat zero
bit 7	SF	Sign Flag	0 = el resultat no ha estat negatiu; 1 = el resultat ha estat negatiu
bit 8	TF	Trap Flag	Facilita l'execució pas a pas.
bit 9	IF	Interrupt Enable Flag	Reservat per al sistema operatiu en mode protegit.
bit 10	DF	Direction Flag	0 = autoincrement cap a adreces altes; 1 = autoincrement cap a adreces baixes
bit 11	OF	Overflow Flag	0 = no s'ha produït sobreeiximent; 1 = sí que s'ha produït sobreeiximent
bit 12	IOPL	I/O Privilege Level	Reservat per al sistema operatiu en mode protegit.
bit 13	IOPL	I/O Privilege Level	Reservat per al sistema operatiu en mode protegit.
bit 14	NT	Nested Task Flag	Reservat per al sistema operatiu en mode protegit.
bit 15		No definit	
bit 16	RF	Resume Flag	Facilita l'execució pas a pas.
bit 17	VM	Virtual-86 Mode Flag	Reservat per al sistema operatiu en mode protegit.
bit 18	AC	Alignment Check Flag	Reservat per al sistema operatiu en mode protegit.
bit 19	VIF	Virtual Interrupt Flag	Reservat per al sistema operatiu en mode protegit.
bit 20	VIP	Virtual Interrupt Pending	Reservat per al sistema operatiu en mode protegit.
bit 21	ID	CPU ID	Si el bit pot ser modificat pels programes en l'espai d'usuari, CPUID està disponible.
bit 22-31		No definits	

2. Llenguatges de programació

Un programa és un conjunt d'instruccions que segueixen unes normes sintàctiques estrictes especificades en un llenguatge de programació concret i dissenyat de manera que, quan s'executa sobre una màquina concreta, fa una tasca determinada sobre un conjunt de dades. Els programes, per tant, estan formats per codi (instruccions) i dades. De manera genèrica, el fitxer que conté el conjunt d'instruccions i la definició de les dades que utilitzarem s'anomena **codi font**.

Per a poder executar un programa, l'haurem de traduir a un llenguatge que pugui entendre el processador; aquest procés s'anomena habitualment *compilació*. Convertim el codi font en **codi executable**. Aquest procés per a generar el codi executable normalment es descompon en dues fases: en la primera fase el codi font es tradueix a un **codi objecte**, i en la segona fase s'enllaça aquest codi objecte i altres codis objecte del mateix tipus que ja tinguem generats, si cal, per a generar el codi executable final.

Codi objecte

El codi objecte és un codi de baix nivell format per una col·lecció organitzada de seqüències de codis seguint un format estàndard. Cada seqüència, en general, conté instruccions per a la màquina en què s'ha d'executar el codi per a dur a terme alguna tasca concreta; també pot tenir un altre tipus d'informació associada (per exemple, informació de reubicació, comentaris o els símbols del programa per a la depuració).

Per a iniciar l'execució, cal que tant el codi com les dades (almenys una part) estiguin carregats a la memòria del computador.

Per a escriure un programa, cal utilitzar un llenguatge de programació que ens permeti especificar el funcionament que es vol que tingui el programa. Hi ha molts llenguatges de programació i, segons la funcionalitat que vulguem donar al programa, serà millor utilitzar-ne un o un altre. En molts casos, escollir-ne un no és una elecció fàcil i pot condicionar força el desenvolupament i el funcionament. Hi ha moltes maneres de classificar-los, però per l'interès d'aquesta assignatura i com que és una de les maneres més generals de fer-ho, els classificarem segons el nivell d'abstracció (proximitat a la màquina) en dos llenguatges:

1) Llenguatges de baix nivell

S'anomenen *llenguatges de baix nivell* perquè depenen de l'arquitectura del processador en què volem executar el programa i no disposen de sentències amb una estructura lògica que facilitin la programació i la comprensió del codi al programador, sinó que estan formats per una llista d'instruccions específiques d'una arquitectura.

Podem distingir entre dos llenguatges:

a) Llenguatge de màquina. Llenguatge que pot interpretar i executar un processador determinat. Aquest llenguatge està format per instruccions codificades en binari (0 i 1). És generat per un compilador a partir de les especificacions d'un altre llenguatge simbòlic o d'alt nivell. És molt difícil d'entendre per al programador i seria molt fàcil cometre errors si s'hagués de codificar.

b) Llenguatge d'assemblador. Llenguatge simbòlic que s'ha definit perquè es puguin escriure programes amb una sintaxi propera al llenguatge de màquina, però sense haver d'escriure el codi en binari, sinó utilitzant una sèrie de mnemònics més fàcils d'entendre per al programador. Per a executar aquests programes també cal un procés de traducció, generalment anomenat *assemblatge*, però més senzill que en els llenguatges d'alt nivell.

2) Llenguatges d'alt nivell. Els llenguatges d'alt nivell no tenen relació directa amb un llenguatge de màquina concret, no depenen de l'arquitectura del processador en què s'executaran i disposen de sentències amb una estructura lògica que faciliten la programació i la comprensió del codi al programador; les instruccions habitualment són mots extrets d'un llenguatge natural, generalment l'anglès, perquè el programador les pugui entendre millor. Per a poder executar programes escrits en aquests llenguatges cal un procés previ de compilació per a passar de llenguatge d'alt nivell a llenguatge de màquina; el codi generat en aquest procés sí que dependrà de l'arquitectura del processador en què s'executarà.

2.1. Entorn de treball

L'entorn de treball que utilitzarem per a desenvolupar els problemes i les pràctiques de programació serà un PC basat en processadors x86-64 (Intel64 o AMD64) sobre el qual s'executarà un sistema operatiu Linux de 64 bits.

L'entorn de treball es podrà executar nativament sobre un PC amb un processador amb arquitectura x86-64, amb sistema operatiu Linux de 64 bits, o utilitzant algun programari de virtualització que permeti executar un sistema operatiu Linux de 64 bits.

Els llenguatges de programació que utilitzarem per a escriure el codi font dels problemes i de les pràctiques de programació de l'assignatura seran: un llenguatge d'alt nivell, el llenguatge C estàndard, per a dissenyar el programa principal i les operacions d'E/S i un llenguatge de baix nivell, el llenguatge assemblador x86-64, per a implementar funcions concretes i veure com treballa aquesta arquitectura a baix nivell.

Hi ha diferents sintaxis de llenguatge assemblador x86-64, però utilitzarem la sintaxi NASM (Netwide Assembler) basada en la sintaxi Intel.

3. El llenguatge d'assemblador per a l'arquitectura x86-64

3.1. Estructura d'un programa en assemblador

Un programa assemblador escrit amb sintaxi NASM està format per tres seccions o segments: *.data* per a les dades inicialitzades, *.bss* per a les dades no inicialitzades i *.text* per al codi:

```
section .data

section .bss

section .text
```

Per a definir una secció es poden utilitzar indistintament les directives *section* i *segment*.

La secció *.bss* no és necessària si s'inicialitzen totes les variables que es declaren a la secció *.data*.

La secció *.text* permet també definir variables i, per tant, permetria prescindir també de la secció *.data*, encara que no és recomanable. Per tant, aquesta és l'única secció realment obligatòria en tot programa.

La secció *.text* ha de començar sempre amb la directiva *global*, que indica al GCC quin és el punt d'inici del codi. Quan s'utilitza GCC per a generar un executable, el nom de l'etiqueta on s'inicia l'execució del codi ha de tenir obligatòriament el nom *main*; també ha d'incloure una crida al sistema operatiu per a finalitzar l'execució del programa i retornar el control al terminal des d'on hem cridat el programa.

En el programa d'exemple `hola.asm` utilitzat anteriorment hem destacat la declaració de les seccions *.data* i *.text*, la directiva global i la crida al sistema operatiu per a finalitzar l'execució del programa.

```
                        ;1: fitxer hola.asm
section .data           ;2: Inici de la  secció de dades
  msg db "Hola!",10     ;3:
                        ;4: El 10 correspon al codi ASCII del salt de línia.
                        ;5:
section .text           ;6: Inici de la secció de codi.
  global main           ;7: Aquesta directiva és per a fer visible
                        ;8: una etiqueta pel compilador de C.
                        ;9:
  main:                 ;10: Per defecte el compilador de C reconeix com a
                        ;11: punt d'inici del programa l'etiqueta main.
                        ;12: Mostrar un missatge
  mov rax,4             ;13: Posa el valor 4 al registre rax
                        ;14: per a fer la crida a la funció write (sys_write)
  mov rbx,1             ;15: Posa el valor 1 al registre RBX
                        ;16: per a indicar el descriptor que fa referència
                        ;17: a la sortida estàndard.
  mov rcx,msg           ;18: Posa l'adreça de la variable msg
                        ;19: al registre RCX
  mov rdx,6             ;20: Posa la longitud del missatge inclòs el 10
                        ;21: del final al registre RDX
  int 80h               ;22: crida al sistema operatiu
                        ;23:
                        ;24: retorna el control al terminal del sistema operatiu.
  mov rax,1             ;25: Posa el valor 1 al registre rax
                        ;26: per a fer la crida a la funció exit (sys_exit)
  mov rbx,0             ;27: Posa el valor 0 al registre RBX
                        ;28: per a indicar el codi de retorn (0=sense errors)
  int 80h               ;29: crida al sistema operatiu
                        ;30:
```

A continuació es descriuen els diferents elements que es poden introduir dins les seccions d'un programa assemblador.

3.2. Directives

Les directives són pseudooperacions que només són reconegudes per l'assemblador. No s'han de confondre amb les instruccions, tot i que en alguns casos sí que poden afegir codi al nostre programa. La seva funció principal és declarar certs elements del nostre programa per a poder ser identificats més fàcilment pel programador i també per a facilitar la tasca d'assemblatge.

A continuació explicarem algunes de les directives que podem trobar en un programa amb codi assemblador i que ens caldrà utilitzar: definició de constants, definició de variables i definició d'altres elements.

3.2.1. Definició de constants

Una constant és un valor que no pot ser modificat per cap instrucció del codi del programa. Realment una constant és un nom que es dóna per a referir-se a un valor determinat.

La declaració de constants es pot fer en qualsevol part del programa: al principi del programa fora de les seccions *.data*, *.bss*, *.text* o dins de qualsevol de les seccions anteriors.

Les constants són útils per a facilitar la lectura i possibles modificacions del codi. Si, per exemple, utilitzem un valor en molts llocs d'un programa, com podria ser la mida d'un vector, i volem provar el programa amb un valor diferent, haurem de canviar aquest valor a tots els llocs on l'hem utilitzat amb la possibilitat de deixar-ne un per modificar i, per tant, que alguna cosa no funcioni; en canvi, si definim una constant amb aquest valor i en el codi utilitzem el nom de la constant, modificant el valor assignat a la constant, quedarà tot modificat.

Per a definir constants s'utilitza la directiva `equ`, de la manera següent:

```
nom_constant equ valor
```

Exemples de definicions de constants

```
midaVec equ 5

ServeiSO equ 80h

Missatge1 equ 'Hola'
```

3.2.2. Definició de variables

La declaració de variables en un programa en assemblador es pot incloure a la secció *.data* o a la secció *.bss*, segons l'ús de cada una.

Secció *.data*, variables inicialitzades

Les variables d'aquesta secció es defineixen utilitzant les següents directives:

- `db`: defineix una variable de tipus byte, 8 bits.
- `dw`: defineix una variable de tipus paraula (word), 2 bytes = 16 bits.
- `dd`: defineix una variable de tipus doble paraula (double word), 2 paraules = 4 bytes = 32 bits.
- `dq`: defineix una variable de tipus quàdruple paraula (quad word), 4 paraules = 8 bytes = 64 bits.

El format utilitzat per a definir una variable utilitzant qualsevol de les directives anteriors és el mateix:

```
nom_variable directiva valor_inicial
```

Exemples

```
var1 db 255                   ;defineix una variable amb el valor FFh
Var2 dw 65535                   ;en hexadecimal FFFFh
var4 dd 4294967295              ;en hexadecimal FFFFFFFFh
var8 dq 18446744073709551615    ;en hexadecimal FFFFFFFFFFFFFFFFh
```

Es pot utilitzar una constant per a inicialitzar variables. Només cal que la constant s'hagi definit abans de la seva primera utilització.

```
midaVec equ 5
indexVec db midaVec
```

Els valors inicials de les variables i constants es poden expressar en bases diferents com decimal, hexadecimal, octal i binari, o també com a caràcters i cadenes de caràcters.

Si es volen separar els dígits dels valors numèrics inicials per a facilitar-ne la lectura es pot utilitzar el símbol '_', sigui quina sigui la base en què estigui expressat el nombre.

```
var4 dd 4_294_967_295
var8 dq FF_FF_FF_FF_FF_FF_FF_FFh
```

Els valors numèrics es consideren per defecte en decimal, però també es pot indicar explícitament que es tracta d'un **valor decimal** finalitzant el nombre amb el caràcter *d*.

```
var db 67   ;el valor 67 decimal
var db 67d   ;el mateix valor
```

Els **valors hexadecimals** han de començar per *0x*, *0h* o *$* o bé han de finalitzar amb una *h*.

Si s'especifica el valor *amb* $ al principi o amb una *h* al final, el nombre no pot començar per una lletra; si el primer dígit hexadecimal ha de ser una lletra (*A, B, C, D, E, F*) cal posar un 0 al davant.

```
var db 0xFF
var dw 0hA3
var db $43
var dw 33FFh
```

Les definicions següents són incorrectes:

```
var db $FF       ;hauríem de posar: var db $0FF
var db FFh       ;hauríem de posar: var db 0FFh
```

Els **valors octals** han de començar per *0o* o *0q* o bé han de finalitzar amb el caràcter *o* o *q*.

```
var db 103o
var db 103q
var db 0o103
var db 0q103
```

Els **valors binaris** han de començar per *0b* o bé han de finalitzar amb el caràcter *b*.

```
var db 0b01000011
var dw 0b0110_1100_0100_0011
var db 01000011b
var dw 0110_0100_0011b
```

Els **caràcters** i les **cadenes** de caràcters han d'anar entre cometes simples (''), dobles ("") o cometes obertes *backquotes* (` `):

```
var db 'A'
var db "A"
var db `A`
```

Les cadenes de caràcters (*strings*) es defineixen de la mateixa manera:

```
cadena db 'Hola'   ;defineix una cadena formada per 4 caràcters
cadena db "Hola"
cadena db `Hola`
```

Les cadenes de caràcters també es poden definir com una sèrie de caràcters individuals separats per comes.

Les definicions següents són equivalents:

```
cadena db 'Hola'
cadena db 'H','o','l','a'
cadena db 'Hol','a'
```

Les cadenes de caràcters poden incloure caràcters no imprimibles i caràcters especials; aquests caràcters es poden incloure amb la seva codificació (ASCII), també poden utilitzar el codi ASCII per a qualsevol altre caràcter encara que sigui imprimible.

```
cadena db 'Hola',10   ;afegeix el caràcter ASCII de salt de línia
cadena db 'Hola',9   ;afegeix el caràcter ASCII de tabulació
cadena db 72,111,108,97 ;defineix igualment la cadena 'Hola'
```

Si la cadena es defineix entre cometes obertes (` `) també s'admeten algunes seqüències d'*escape* començades amb \:

\n: salt de línia
\t: tabulador
\e: el caràcter ESC (ASCII 27)
\0x[valor]: [valor] ha de ser 1 byte en hexadecimal, expressat amb 2 dígits hexadecimals.

\u[valor]: [valor] ha de ser la codificació hexadecimal d'un caràcter en format UTF.

```
cadena db `Hola\n`   ;afegeix el caràcter de salt de línia al final
cadena db `Hola\t`   ;afegeix el caràcter de salt de línia al final
cadena db `\e[10,5H` ;defineix una seqüència d'escape que comença amb
                     ;el caràcter ESC = \e = ASCII(27)
cadena db `Hola \u263a` ;mostra: Hola ☺
```

Si volem definir una cadena de caràcters que inclogui un d'aquests símbols (', ", `) s'ha de delimitar la cadena utilitzant unes cometes d'altre tipus o posar-hi una contrabarra (\) al davant.

```
"L'assemblador","L`assemblador",'L`assemblador',
`L'assemblador`, `L\`assemblador`,
'Vocals accentuades "àèéíòóú", amb dièresi "ïü" i símbols "ç€@#".'
`Vocals accentuades "àèéíòóú", amb dièresi "ïü" i símbols "ç€@#".`
```

Si volem declarar una variable inicialitzada amb un valor que es repeteix un conjunt de vegades podem utilitzar la directiva *times*.

```
cadena times 4 db 'PILA'       ;defineix una variable inicialitzada
                               ;amb el valor 'PILA' 4 vegades
cadena2 db 'PILAPILAPILAPILA'  ;és equivalent a la declaració anterior
```

Vectors

Els vectors en assemblador es defineixen amb un nom de variable i indicant a continuació els valors que formen el vector.

```
vector1 db 23, 42, -1, 65, 14  ;vector format per 5 valors de tipus
                               ;byte
vector2 db 'a', 'b', 'c', 'd'  ;vector format per 4 bytes,
                               ;inicialitzat usant caràcters
vector3 db 97, 98, 99, 100       ;és equivalent al vector anterior
vector4 dw 1000, 2000, -1000, 3000 ;vector format per 4 paraules
```

També podem utilitzar la directiva *times* per a inicialitzar vectors, però aleshores totes les posicions tindran el mateix valor.

```
vector5 times 5 dw 0 ;vector format per 5 paraules inicialitzades a 0
```

Valors i constants numèriques

En assemblador, tots els valors s'emmagatzemen com a valors amb signe expressats en complement a 2.

Secció *.bss*, variables no inicialitzades

Dins d'aquesta secció es declaren i es reserva espai per a les variables del nostre programa per a les quals no volem donar un valor inicial.

Cal utilitzar les directives següents per a declarar variables no inicialitzades:

- `resb`: reserva espai en unitats de byte
- `resw`: reserva espai en unitats de paraula, 2 bytes
- `resd`: reserva espai en unitats de doble paraula, 4 bytes
- `resq`: reserva espai en unitats de quàdruple paraula, 8 bytes

El format utilitzat per a definir una variable utilitzant qualsevol de les directives anteriors és el mateix:

```
nom_variable directiva multiplicitat
```

La multiplicitat és el nombre de vegades que reservem l'espai definit pel tipus de dada que determina la directiva.

Exemples

```
section .bss

var1 resb 1 ;reserva 1 byte
var2 resb 4 ;reserva 4 bytes
var3 resw 2 ;reserva 2 paraules = 4 bytes, equivalent al cas anterior
var3 resd 1 ;reserva una quàdruple paraula = 4 bytes
            ;equivalent als dos casos anteriors
```

3.2.3. Definició d'altres elements

Altres elements són:

1) extern. Declara un símbol com a extern. L'utilitzem si volem accedir a un símbol que no es troba definit en el fitxer que estem assemblant, sinó en un altre fitxer de codi font, en què haurà d'estar definit i declarar amb la directiva *global*.

En el procés d'assemblatge, qualsevol símbol declarat com a extern no generarà cap error; és durant el procés d'enllaçament que, si no hi ha un fitxer de codi objecte en què aquest símbol estigui definit, donarà error.

La directiva té el format següent:

```
extern símbol1, símbol2, ..., símbolN
```

En una mateixa directiva *extern* es poden declarar tants símbols com es vulgui separats per comes.

2) global. És la directiva complementària d'*extern*. Permet fer visible un símbol definit en un fitxer de codi font a altres fitxers de codi font; d'aquesta manera, ens podrem referir a aquest símbol en altres fitxers utilitzant la directiva *extern*.

El símbol haurà d'estar definit en el mateix fitxer de codi font on sigui la directiva global.

Hi ha un ús especial de *global*: declarar una etiqueta que s'ha d'anomenar *main* perquè el compilador de C (GCC) pugui determinar el punt d'inici de l'execució del programa.

La directiva té el format següent:

```
global símbol1, símbol2, ..., símbolN
```

En una mateixa directiva *global* es poden declarar tants símbols com es vulgui separats per comes.

Generació de l'executable

Cal fer l'assemblatge dels dos codis font assemblador, i enllaçar els dos codis objecte generats per a obtenir l'executable.

Exemple d'utilització de les directives extern i global

Prog1.asm

```
global printHola, msg   ;fem visibles la subrutina
                           ;printHola i la variable msg.
section .data
   msg   db      "Hola!",10
   ...

section .text
printHola:
   mov rax,4
   mov rbx,1
   mov rcx,msg
   mov rdx,6
   int 80h
   ret
```

Prog2.asm

```
extern printHola, msg ;indiquem que estan declarades
                           ;en un altre fitxer.
global main         ;fem visible l'etiqueta main
                    ;per a poder iniciar l'execució.
section .text
main:
   call printHola   ;executem el codi del Prog1.asm

   mov rax,4         ;executem aquest codi però
   mov rbx,1         ;utilitzem la variable definida
   mov rcx,msg       ;al fitxer Prog1.asm
   mov rdx,6
   int 80h

   mov rax,1
   mov rbx,0
   int 80h
```

3) section. Defineix una secció dins del fitxer de codi font. Cada secció farà referència a un segment de memòria diferent dins l'espai de memòria assignat al programa. Hi ha tres tipus de seccions:

a) `.data`: secció en què es defineixen dades inicialitzades, dades a les quals donem un valor inicial.

b) `.bss`: secció en què es defineixen dades sense inicialitzar.

c) `.text`: secció en què s'inclouen les instruccions del programa.

La utilització d'aquestes directives no és imprescindible, però sí recomanable per tal de definir els diferents tipus d'informació que utilitzarem.

4) cpu. Aquesta directiva indica que només es podran utilitzar els elements compatibles amb una arquitectura concreta. Només podrem utilitzar les instruccions definides en aquella arquitectura.

Aquesta directiva s'acostuma a posar al principi del fitxer de codi font, abans de l'inici de qualsevol secció del programa.

El format de la directiva és:

```
cpu tipus_processador
```

Exemple

cpu 386
cpu x86-64

3.3. Format de les instruccions

L'altre element que forma part de qualsevol programa escrit en llenguatge d'assemblador són les instruccions.

Les instruccions en assemblador tenen el format general següent:

```
[etiqueta:] instrucció [destinació[, font]] [;comentari]
```

on *destinació* i *font* representen els operands de la instrucció.

Al final de la instrucció podem afegir un comentari precedit del símbol ;.

Els elements entre [ ] indiquen elements opcionals, per tant, l'únic element imprescindible és el nom de la instrucció.

3.3.1. Etiquetes

Una etiqueta fa referència a un element dins del programa assemblador. Serveix per a facilitar al programador la tasca de fer referència a diferents elements del programa. Les etiquetes serveixen per a definir constants, variables o posicions del codi i les utilitzem com a operands en les instruccions o directives del programa.

Per a definir una etiqueta podem utilitzar nombres, lletres i símbols *_, $, #, @, ~, .* i *?*. Les etiquetes han de començar amb un caràcter alfabètic o amb els símbols *_, ., o ?*, però les etiquetes que comencen amb aquests tres símbols tenen un significat especial dins la sintaxi NASM i per aquest motiu es recomana no utilitzar-los a l'inici de l'etiqueta.

Una qüestió important que cal tenir en compte és que en sintaxi NASM es distingeix entre minúscules i majúscules. Per exemple, les etiquetes següents serien diferents: Etiqueta1, etiqueta1, ETIQUETA1, eTiQueTa1, ETIqueta1.

Exemple

```
Servei equ 80h

section .data
   msg   db      "Hola!",10

section .text
printHola:
   mov rax,4
   mov rbx,1
   mov rcx,msg
   mov rdx,6
   int Servei
   jmp printHola
```

En aquest fragment de codi hi ha definides tres etiquetes: *Servei* defineix una constant, *msg* defineix una variable i *printHola* defineix una posició de codi.

Les etiquetes per a marcar posicions de codi s'utilitzen en les instruccions de salt per a indicar el lloc on s'ha de saltar i també per a definir la posició d'inici d'una subrutina dins del codi font (vegeu instruccions de ruptura de seqüència en el subapartat 3.4.3).

Habitualment quan es defineix una etiqueta per a marcar una posició de codi, es defineix amb una seqüència de caràcters acabada amb el caràcter : (dos punts). Però quan utilitzem aquesta etiqueta com a operand no posarem els dos punts.

Exemple

```
            ...
Etiqueta1:   mov rax,0
            ...
            jmp Etiqueta1   ;Aquesta instrucció salta a la
                           ;instrucció mov rax,0
            ...
```

Una etiqueta es pot escriure en la mateixa línia que ocupa una instrucció o en una altra. El fragment següent és equivalent al fragment anterior:

```
   ...
Etiqueta1:

   mov rax,0
   ...
   jmp Etiqueta1   ;Aquesta instrucció salta a la instrucció mov rax,0
   ...
```

3.4. Joc d'instruccions i modes d'adreçament

Una instrucció en assemblador està formada per un *codi d'operació* (el nom de la instrucció) que determina què ha de fer la instrucció, més un conjunt d'operands que expressen directament una dada, un registre o una adreça de memòria; les diferents maneres d'expressar un operand en una instrucció i el procediment associat que permet obtenir la dada s'anomena *mode d'adreçament*.

Hi ha instruccions que no tenen cap operand i que treballen implícitament amb registres o la memòria; hi ha instruccions que tenen només un operand i hi ha instruccions amb dos operands o més (en aquest cas els operands se separen amb comes ,):

1) Instruccions sense cap operand explícit: *codi_operació*

Exemple

```
ret      ;retorn de subrutina
```

2) Instruccions amb un sol operand: *codi_operació destinació*

```
push rax     ;guarda el valor de rax a la pila, rax és un operand font
pop rax      ;carrega el valor del cim de la pila a rax, rax és un
             ;operand destinació
call subr1   ;crida a la subrutina subr1, subr1 és un operand font
inc rax      ;rax=rax+1, rax fa d'operand font i també d'operand destinació.
```

3) Instruccions amb dos operands: *codi_operació destinació, font*

Exemples

```
mov rax, [vec+rsi];mou el valor de la posició del vector vec
                  ;indicada per rsi, és l'operand font, a rax,
                  ;rax és l'operand destinació
add rax, 4        ;rax=rax+4
```

Operand font i operand destinació

L'operand font especifica un valor, un registre o una adreça de memòria on hem d'anar a buscar una dada que ens fa falta per a executar la instrucció.

L'operand destinació especifica un registre o una adreça de memòria on hem de guardar la dada que hem obtingut en executar la instrucció. L'operand destinació també pot actuar com a operand font i el valor original es perd quan guardem la dada obtinguda en executar la instrucció.

3.4.1. Tipus d'operands de les instruccions x86-64

Operands font i destinació

A les instruccions amb un sol operand, aquest es pot comportar només com a operand font, només com a operand destinació o com a operand font i destinació.

Exemples

```
push rax
```

El registre `rax` és un **operand font**; la instrucció emmagatzema el valor de l'operand font a la pila del sistema, de manera que la pila és un operand destinació implícit.

```
pop rax
```

El registre `rax` es comporta com a **operand destinació**; la instrucció emmagatzema a `rax` el valor que es troba al cim de la pila del sistema, de manera que la pila és un operand font implícit.

```
inc rax
```

El registre `rax` és a la vegada **operand font i destinació**; la instrucció `inc` fa l'operació `rax = rax + 1`, suma el valor original de `rax` amb 1 i torna a guardar el resultat final a `rax`.

En les instruccions amb dos operands, el primer operand es pot comportar com a operand font i/o destinació, i el segon operand es comporta sempre com a operand font.

Exemples

```
mov rax, rbx
```

El primer operand fa només d'operand destinació, la instrucció emmagatzema el valor indicat pel segon operand en el primer operand (`rax = rbx`).

```
add rax, 4
```

El primer operand es comporta alhora com a operand font i destinació; la instrucció `add` fa l'operació `rax = rax + 4`, suma el valor original de `rax` amb el valor 4 i torna a emmagatzemar el resultat a `rax`, i es perd el valor que teníem originalment.

Localització dels operands

Els operands de les instruccions es poden trobar en tres llocs diferents; a la instrucció (valors immediats), en registres o a memòria:

Vegeu també

Per a saber quines instruccions permeten utilitzar un tipus d'operand determinat, ja sigui com a operand font o operand destinació, cal consultar la referència de les instruccions a l'apartat 7 d'aquest mòdul.

1) Immediats. En les instruccions de dos operands es pot utilitzar un valor immediat com a operand font; algunes instruccions d'un operand també admeten un valor immediat com a operand. Els valors immediats es poden expressar com a valors numèrics (decimal, hexadecimal, octal o binari) o com a caràcters o cadenes de caràcters. També es poden utilitzar les constants definides en el programa com a valors immediats.

Per a especificar un valor immediat en una instrucció s'utilitza la mateixa notació que l'especificada en la definició de variables inicialitzades.

Exemples

```
mov al, 10b       ;un valor immediat expressat en binari

cinc equ 5h       ;es defineix una constant en hexadecimal
mov al, cinc      ;s'utilitza el valor de la constant com
                  ;a valor immediat

mov eax, 0xABFE001C ;un valor immediat expressat en hexadecimal.

mov ax, 'HI'      ;un valor immediat expressat com una cadena
                  ;de caràcters
```

2) Registres. Els registres es poden utilitzar com a operand font i com a operand destinació. Podem utilitzar registres de 64 bits, 32 bits, 16 bits i 8 bits. Algunes instruccions poden utilitzar registres de manera implícita.

Exemples

```
mov al, 100
mov ax, 1000
mov eax, 100000
mov rax, 10000000
mov rbx, rax
mul bl    ;ax = al * bl, els registres al i ax són implícits,
          ;al com a operand font i ax com a operand de destinació.
```

3) Memòria. Les variables declarades a memòria es poden utilitzar com a operands font i destinació. En el cas **d'instruccions amb dos operands, només un dels operands pot accedir a la memòria**, l'altre ha de ser un registre o un valor immediat (i aquest serà l'operand font).

Accés a memòria

En sintaxi NASM es distingeix l'accés al contingut d'una variable de memòria, de l'accés a l'adreça d'una variable. Si es vol accedir al contingut d'una variable de memòria cal especificar el nom de la variable entre *[ ]*; si s'utilitza el nom d'una variable sense *[ ]* ens estarem referint a la seva adreça.

Exemple

```
section .data
   var1 dd 100       ;variable de 4 bytes (var1 = 100)

section .text
   mov rax, var1     ;es carrega a rax l'adreça de la variable var1
   mov rbx, [var1]   ;es carrega a rbx el contingut de var1, rbx=100
  mov rbx, [rax]   ;aquesta instrucció farà el mateix que l'anterior.
   mov [var1], rbx   ;es carrega a var1 el contingut del registre rbx
```

Cal notar que quan especifiquem un nom d'una variable sense els claudàtors *[ ]*, no estem fent un accés a memòria, sinó que l'adreça de la variable està codificada en la instrucció mateix i es considera un valor immediat, i per tant podem fer el següent:

```
;si considerem que l'adreça de memòria a qui fa referència var1
;és l'adreça 12345678h
mov QWORD [var2], var1   ;es carrega a var2 l'adreça de var1
mov QWORD [var2],12345678h ;es equivalent a la instrucció anterior.
```

`[var2]` és l'operand que fa l'accés a memòria i `var1` es codifica com un valor immediat.

Mida dels operands

En sintaxi NASM la mida de les dades especificades per un operand pot ser de byte, word, double word i quad word. I cal recordar que ho fa en format *little-endian*.

- BYTE: indica que la mida de l'operand és d'un byte (8 bits).
- WORD: indica que la mida de l'operand és d'una paraula (word) o dos bytes (16 bits).
- DWORD: indica que la mida de l'operand és d'una doble paraula (double word) o quatre bytes (32 bits).
- QWORD: indica que la mida de l'operand és d'una quàdruple paraula (quad word) o vuit bytes (64 bits).

Cal anar molt en compte amb la mida dels operands que estem utilitzant en cada moment, sobretot quan fem referències a memòria, i especialment quan aquesta és l'operand destinació, ja que utilitzarem la part de la memòria necessària per a emmagatzemar l'operand font segons la mida que tingui.

En alguns casos és obligatori especificar la mida de l'operand; això es fa utilitzant els modificadors BYTE, WORD, DWORD i QWORD al davant de la referència a memòria. La funció del modificador és utilitzar tants bytes com indica a partir de l'adreça de memòria especificada, independentment de la mida del tipus de dada (`db`, `dw`, `dd`, `dq`) que haguem utilitzat a l'hora de definir la variable.

En els casos que no és obligatori especificar el modificador es podem utilitzar per a facilitar la lectura del codi o modificar un accés a memòria.

Els casos en què cal especificar obligatòriament la mida dels operands són els següents:

1) Instruccions de dos operands en què el primer operand és una posició de memòria i el segon operand és un immediat; cal indicar la mida de l'operand.

Exemple

```
.data
  var1 dd 100              ;variable definida de 4 bytes
.text
  mov DWORD [var1], 0ABCh  ;s'indica que la variable var1
                           ;és de 32 bits
```

La instrucció següent és incorrecta:

```
mov [var1], 0ABCh
```

En el cas anterior el compilador no seria capaç de saber com ha de copiar aquest valor a la variable `[var1]`, encara que quan l'hem definit se n'hagi especificat el tipus.

2) Instruccions d'un operand en què l'operand sigui una posició de memòria.

Exemple

```
inc QWORD [var1]    ;s'indica que la posició de memòria afectada és
                    ;de mida 8 bytes. [var1]=[var1]+1
push WORD [var1]    ;s'indica que posem 2 bytes a la pila.
```

3.4.2. Modes d'adreçament

Quan un operand es troba a la memòria, cal considerar quin mode d'adreçament utilitzem per a expressar un operand en una instrucció per a definir la manera d'accedir a una dada concreta. A continuació veurem els modes d'adreçament que podem utilitzar en un programa assemblador:

1) Immediat. En aquest cas, l'operand fa referència a una dada que es troba en la instrucció mateix. No cal fer cap accés extra a memòria per a obtenir-la. Només podem utilitzar un adreçament immediat com a operand font. El nombre especificat ha de ser un valor que es pugui expressar amb 32 bits com a màxim, que serà el resultat d'avaluar una expressió aritmètica formada per valors numèrics i operadors aritmètics i també sumar una adreça de memòria representada mitjançant una etiqueta (nom d'una variable), amb l'excepció de la instrucció *mov* quan el primer operand és un registre de 64 bits, al qual podrem especificar un valor que es podrà expressar amb 64 bits.

```
mov rax, 0102030405060708h   ;el segon operand utilitza adreçament
                             ;immediat expressat amb 64 bits.
mov QWORD [var], 100         ;el segon operand utilitza adreçament immediat
                             ;carrega el valor 100 i es guarda a var
mov rbx, var                 ;el segon operand utilitza adreçament immediat
                             ;carrega l'adreça de var al registre rbx
mov rbx, var+16 * 2          ;el segon operand utilitza adreçament immediat
                             ;carrega l'adreça de var+32 al registre rbx
```

Per a especificar un valor immediat en una instrucció s'utilitza la mateixa notació que l'especificada en la definició de variables inicialitzades.

2) Directe a registre. En aquest cas, l'operand fa referència a una dada que es troba emmagatzemada en un registre. En aquest mode d'adreçament podem especificar qualsevol registre de propòsit general (registres de dades, registres índex i registres apuntadors).

Exemple

```
mov rax, rbx       ;els dos operands utilitzen adreçament
                   ;directe a registre, rax = rbx
```

3) Directe a memòria. En aquest cas, l'operand fa referència a una dada que es troba emmagatzemada en una posició de memòria. L'operand haurà d'especificar el nom d'una variable de memòria entre claudàtors *[ ]*, cal recordar que en sintaxi NASM s'interpreta el nom d'una variable sense claudàtors com l'adreça de la variable i no com el contingut de la variable.

Exemples

```
mov rax,[var]    ;el segon operand utilitza adreçament
                 ;directe a memòria,  rax = [var]
add [suma],rcx   ;el primer operand utilitza adreçament
                 ;directe a memòria, [suma]=[suma]+rcx
```

4) Indirecte a registre. En aquest cas, l'operand fa referència a una dada que es troba emmagatzemada en una posició de memòria. L'operand haurà d'especificar un registre entre claudàtors *[ ]*; el registre contindrà l'adreça de memòria a la qual volem accedir.

Exemples

```
mov rbx, var     ;es carrega a rbx l'adreça de la variable var
mov rax, [rbx]   ;el segon operand utilitza l'adreça que tenim a rbx
                ;per a accedir a memòria, es mouen 8 bytes a partir de
                 ;l'adreça especificada per rbx i es guarden a rax.
```

5) Indexat. En aquest cas, l'operand fa referència a una dada que es troba emmagatzemada en una posició de memòria. Direm que un operand utilitza adreçament indexat si especifica una adreça de memòria com a adreça base que pot ser expressada mitjançant un nombre o el nom d'una variable que tinguem definida sumada a un registre que actua com a índex respecte d'aquesta adreça de memòria entre claudàtors *[ ]*.

Exemples

```
mov rax, [vector+rsi]   ;vector conté l'adreça base, rsi actua
                        ;com a registre índex
add [1234h+r9],rax      ;1234h és l'adreça base, r9 actua
                        ;com a registre índex.
```

6) Relatiu. En aquest cas, l'operand fa referència a una dada que es troba emmagatzemada en una posició de memòria. Direm que un operand utilitza adreçament relatiu quan especifica un registre sumat a un nombre entre claudàtors *[ ]*. El registre contindrà una adreça de memòria que actuarà com a adreça base i el nombre com un desplaçament respecte d'aquesta adreça.

Exemples

```
mov rbx, var        ;es carrega a rbx l'adreça de la variable var
mov rax, [rbx+4]    ;el segon operand utilitza adreçament relatiu
                    ;4 és el desplaçament respecte d'aquesta adreça
mov [rbx+16], rcx   ;rbx conté l'adreça base, 16 és el
                    ;desplaçament respecte d'aquesta adreça.
```

Combinacions de l'adreçament indexat i relatiu. La sintaxi NASM ens permet especificar d'altres maneres un operand per a accedir a memòria; el format de l'operand que haurem d'expressar entre claudàtors *[ ]* és el següent:

```
   [ Registre Base + Registre Index * escala + desplaçament]
```

El registre base i el registre índex poden ser qualsevol registre de propòsit general, però han de ser registres de 32 bits o 64 bits, l'escala pot ser 1, 2, 4 o 8 i el desplaçament ha de ser un nombre representable amb 32 bits que serà el resultat d'avaluar una expressió aritmètica formada per valors numèrics i operadors aritmètics; també podem sumar una adreça de memòria representada mitjançant una etiqueta (nom d'una variable). Podem especificar només els elements que ens siguin necessaris.

Exemples

```
[ rbx + rsi * 2 + 4 * (12+127)]
[ r9 + r10 * 8]
[ r9 - 124 * 8]
[ rax * 4 + 12 / 4]
[ vec+16 + rsi * 2 ]
```

Aquesta és la manera general d'expressar un operand per a accedir a memòria; els altres modes especificats anteriorment són casos concrets en què només es defineixen alguns d'aquests elements.

7) Relatiu a PC. En el mode de 64 bits es permet utilitzar adreçament relatiu a PC en qualsevol instrucció; en altres modes d'operació l'adreçament relatiu a PC es reserva exclusivament per a les instruccions de salt condicional.

Aquest mode d'adreçament és equivalent a un adreçament relatiu a registre base en què el registre base és el registre comptador de programa (PC). En l'arquitectura x86-64 aquest registre s'anomena *rip*, i el desplaçament és el valor que sumarem al comptador de programa per a determinar l'adreça de memòria a la qual volem accedir.

Utilitzarem aquest mode d'adreçament habitualment en les instruccions de salt condicional. En aquestes instrucció especificarem una etiqueta que representarà el punt del codi al qual volem saltar. La utilització del registre comptador de programa és implícita, però per a utilitzar aquest mode d'adreçament s'ha de codificar el desplaçament respecte del comptador de programa; el càlcul per a obtenir aquest desplaçament a partir de l'adreça representada per l'etiqueta es resol durant l'assemblatge i és transparent al programador.

Exemple

```
je etiqueta1
```

8) Adreçament a pila. És un adreçament implícit, es treballa implícitament amb el cim de la pila, mitjançant el registre apuntador a pila (*stack pointer*); en l'arquitectura x86-64 aquest registre s'anomena *rsp*. En la pila només podrem emmagatzemar valors de 16 bits i de 64 bits.

Només hi ha dues instruccions específiques dissenyades per a treballar amb la pila:

```
push font   ;posa un dada a la pila
pop  destinació   ;treu una dada de la pila
```

Per a expressar l'operand font o destinació podem utilitzar qualsevol tipus d'adreçament descrit anteriorment, però el més habitual és utilitzar l'adreçament a registre.

Exemples

```
push word 23h
push qword [rax]
pop  qword [var]
pop  bx
```

3.4.3. Tipus d'instruccions

El joc d'instruccions dels processadors x86-64 és molt ampli. Podem organitzar les instruccions segons els tipus següents:

Vegeu també

En l'apartat 6 d'aquest mòdul es descriu amb detall el format i la utilització de les instruccions dels processadors x86-64 que considerem més importants.

1) Instruccions de transferència de dades:

- **mov** ***destinació***, ***font***: instrucció genèrica per a moure una dada des d'un origen a un destinació.
- **push** ***font***: instrucció que mou l'operand de la instrucció al cim de la pila.
- **pop** ***destinació***: mou la dada que es troba al cim de la pila a l'operand destinació.
- **xchg** ***destinació***, ***font***: intercanvia continguts dels operands.

2) Instruccions aritmètiques i de comparació:

- **add** ***destinació***, ***font***: suma aritmètica dels dos operands.
- **adc** ***destinació***, ***font***: suma aritmètica dels dos operands considerant el bit de transport.
- **sub** ***destinació***, ***font***: resta aritmètica dels dos operands.
- **sbb** ***destinació***, ***font***: resta aritmètica dels dos operands considerant el bit de transport.
- **inc** ***destinació***: incrementa l'operand en una unitat.
- **dec** ***destinació***: decrement l'operand en una unitat.
- **mul** ***font***: multiplicació entera sense signe.

- **imul** ***font***: multiplicació entera amb signe.
- **div** ***font***: divisió entera sense signe.
- **idiv** ***font***: divisió entera amb signe.
- **neg** ***destinació***: negació aritmètica en complement a 2.
- **cmp** ***destinació***, ***font***: comparació dels dos operands; fa una resta sense guardar el resultat.

3) Instruccions lògiques i de desplaçament:

a) Operacions lògiques:

- **and** ***destinació***, ***font***: operació 'i' lògica.
- **or** ***destinació***, ***font***: operació 'o' lògica.
- **xor** ***destinació***, ***font***: operació 'o exclusiva' lògica.
- **not** ***destinació***: negació lògica bit a bit.
- **test** ***destinació***, ***font***: comparació lògica dels dos operands; fa una 'i' lògica sense guardar el resultat.

b) Operacions de desplaçament:

- **sal** ***destinació***, ***font*** / **shl** ***destinació***, ***font***: desplaçament aritmètic/lògic a l'esquerra.
- **sar** ***destinació***, ***font***: desplaçament aritmètic a la dreta.
- **shr** ***destinació***, ***font***: desplaçament lògic a la dreta.
- **rol** ***destinació***, ***font***: rotació lògica a l'esquerra.
- **ror** ***destinació***, ***font***: rotació lògica a la dreta.
- **rcl** ***destinació***, ***font***: rotació lògica a l'esquerra considerant el bit de transport.
- **rcr** ***destinació***, ***font***: rotació lògica a la dreta considerant el bit de transport.

4) Instruccions de ruptura de seqüència:

a) Salt incondicional:

- **jmp etiqueta**: salta de manera incondicional a l'etiqueta.

b) Salts que consulten un bit de resultat concret:

- **je** ***etiqueta*** / **jz** ***etiqueta***: salta a l'etiqueta si igual, si bit de zero és actiu.
- **jne** ***etiqueta*** / **jnz** ***etiqueta***: salta a l'etiqueta si diferent, si bit de zero no és actiu.
- **jc** ***etiqueta***: salta a l'etiqueta si bit de transport és actiu.
- **jnc** ***etiqueta***: salta a l'etiqueta si el bit de transport no és actiu.
- **jo** ***etiqueta***: salta a l'etiqueta si el bit de sobreeiximent és actiu.
- **jno** ***etiqueta***: salta a l'etiqueta si el bit de sobreeiximent no és actiu.
- **js** ***etiqueta***: salta a l'etiqueta si el bit de signe és actiu.

- **jns *etiqueta***: salta a l'etiqueta si el bit de signe no és actiu.

c) Salts condicionals sense considerar el signe:

- **jb *etiqueta*** / **jnae *etiqueta***: salta a l'etiqueta si és més petit.
- **jbe *etiqueta*** / **jna *etiqueta***: salta a l'etiqueta si és més petit o igual.
- **ja *etiqueta*** / **jnbe *etiqueta***: salta a l'etiqueta si és més gran.
- **jae *etiqueta*** / **jnb *etiqueta***: salta a l'etiqueta si és més gran o igual.

d) Salts condicionals considerant el signe:

- **jl *etiqueta*** / **jnge *etiqueta***: salta si és més petit.
- **jle *etiqueta*** / **jng *etiqueta***: salta si és més petit o igual.
- **jg *etiqueta*** / **jnle *etiqueta***: salta si és més gran.
- **jge *etiqueta*** / **jnl *etiqueta***: salta si és més gran o igual.

e) Altres instruccions de ruptura de seqüència:

- **loop *etiqueta***: decrementa el registre rcx i salta si rcx és diferent de zero.
- **call *etiqueta***: crida a subrutina.
- **ret**: retorn de subrutina.
- **iret**: retorn de rutina de servei d'interrupció (RSI).
- **int *servei***: crida al sistema operatiu.

5) Instruccions d'entrada/sortida:

- **in *destinació*, *font***: lectura del port d'E/S especificat a l'operand font i es guarda a l'operand destinació.

- **out *destinació*, *font***: escriptura del valor especificat per l'operand font al port d'E/S especificat a l'operand destinació.

4. Introducció al llenguatge C

En aquest apartat es fa una breu introducció al llenguatge C. No pretén ser un manual de programació ni una guia de referència del llenguatge, només explica els conceptes necessaris per a poder desenvolupar petits programes en C i poder fer crides a funcions implementades en llenguatge d'assemblador, amb l'objectiu d'entendre el funcionament del processador i la comunicació entre un llenguatge d'alt nivell i un de baix nivell.

4.1. Estructura d'un programa en C

Un programa escrit en llenguatge C segueix en general l'estructura següent:

- Directives de compilació
- Definició de variables globals
- Declaració i implementació de funcions
- Declaració i implementació de la funció main

L'única part imprescindible en tot programa escrit en C és la funció *main*, encara que molts cops cal incloure algunes directives de compilació.

Per exemple, és habitual utilitzar la directiva *include* per a incloure altres fitxers en què es defineixen funcions de la biblioteca estàndard *glibc* que es volen utilitzar en el programa.

Vegem a continuació un primer programa escrit en C:

```
1 /*Fitxer hola.c*/
2 #include <stdio.h>
3 int main(){
4   printf ("Hola!\n");
5   return 0;
6 }
```

La primera línia defineix un comentari; en C els comentaris s'escriuen entre els símbols `/*` i `*/`, també es pot utilitzar `//` per a escriure comentaris d'una sola línia.

Exemples de comentaris en C

```
//Això és un comentari d'una línia
/* Això és un
comentari
de diverses línies */
```

La segona línia correspon a una directiva; en llenguatge C, les directives comencen sempre pel símbol *#*.

La directiva *include* indica al compilador que inclogui el fitxer indicat, *stdio.h*, en compilar el programa.

El fitxer *stdio.h* inclou la definició de les funcions més habituals per a treballar amb la pantalla i el teclat.

La tercera línia declara la funció *main*, funció principal de tot programa escrit en C en què s'iniciarà l'execució del programa; es marca l'inici de la funció amb el símbol { .

La quarta línia és la primera instrucció del *main*, i correspon a una crida a la funció *printf* de la biblioteca estàndard; aquesta funció està definida en el fitxer *stdio.h*. La funció *printf* permet escriure en la pantalla; en aquest cas s'escriu la cadena de caràcters indicada. En llenguatge C cal finalitzar cada instrucció amb un punt i coma (*;*).

La cinquena línia defineix quin és el valor que retornarà la funció. Com que el tipus de retorn és un nombre enter (*int*), cal especificar un valor enter.

Finalment, a la sisena línia, es tanca el codi de la funció amb el símbol *}*. El codi d'una funció en C sempre s'ha de tancar entre els símbols *{* i *}*.

4.1.1. Generació d'un programa executable

Per a generar un executable a partir d'un fitxer de codi font C, utilitzem el compilador GCC.

Per a compilar el programa anterior, cal executar l'ordre següent:

```
$ gcc hola.c -o hola -g
```

Per a executar el programa només cal utilitzar el nom del fitxer de sortida generat afegint *./* al davant:

```
$ ./hola
Hola!
$ _
```

4.2. Elements d'un programa en C

4.2.1. Directives

En C hi ha un conjunt de directives de compilació molt ampli. A continuació es descriuen només les directives que utilitzarem. Les directives comencen sempre pel símbol #.

1) ***include***: permet incloure un altre fitxer, de manera que quan es cridi el compilador sigui compilat juntament amb el codi font.

El més habitual és incloure el què s'anomenen *fitxers de capçalera* (*header*), fitxers amb extensió *.h* amb les definicions de les funcions incloses a les biblioteques estàndard de C, de manera que puguin ser utilitzades en el programa font.

El format de la directiva *include* és:

```
#include <nom_fitxer>
```

Exemple

```
#include <stdio.h>
```

2) ***define***: permet definir valors constants per a ser utilitzats en el programa. Els valors de les constants es poden expressar de maneres diferents: com a cadenes de caràcters o com a valors numèrics expressats en binari, octal, decimal o hexadecimal.

El format de la directiva *define* és:

```
#define nom_constant valor
```

Exemple

```
#define CADENA "Hola"
#define NUMBIN 01110101b
#define NUMOCT 014q
#define NUMDEC 12
#define NUMHEX 0x0C
```

3) ***extern***: declara un símbol com a extern. L'utilitzem si volem accedir a un símbol que no es troba definit en el fitxer que estem compilant, sinó en un altre fitxer de codi font C o codi objecte.

En el procés de compilació, qualsevol símbol declarat com a extern no generarà cap error, és durant el procés d'enllaçament que, si no hi ha un fitxer de codi objecte en què aquest símbol estigui definit, donarà error.

La directiva té el format següent:

```
extern [tipus_del_retorn] nom_de_funció ([llista_de_tipus]);
extern [tipus] nom_variable [= valor_inicial];
```

En una mateixa directiva *extern* es poden declarar tants símbols com es vulgui separats per comes.

Per exemple:

```
extern int i;
extern printVariable(int);
```

4.2.2. Variables

En general, les variables es poden definir de dues maneres:

1) Globals: accessibles des de qualsevol punt del programa.

2) Locals: accessibles només dins d'un fragment del programa.

La manera més habitual de definir variables locals és al principi d'una funció, de manera que només existeixen durant l'execució d'una funció i només són accessibles dins d'aquesta.

Identificadors

Els identificadors són els noms que donem a les variables, constants, funcions, etiquetes i altres objectes.

En C els identificadors han de començar per una lletra o el símbol de _.

També cal tenir present que en C es distingeix entre majúscules i minúscules, per tant, les variables següents són diferents: VARX, varx, varX, VarX.

```
int x=0;                                          //x: variable global
int main(){
    int y=1;                                      //y: variable local de la funció main
    for (y=0; y>3; y++) { int z = y*2; }          //z: variable local del for
}
```

Per a definir una variable cal especificar primer el tipus de dada de la variable seguit del nom de la variable; opcionalment es pot especificar a continuació el valor inicial.

El format general per a definir una variable en C és el següent:

```
tipus nom_variable [= valor_inicial];
```

Els tipus de dades més habituals per a definir variables són:

- caràcters (*char*)
- enters (*int*) i (*long*)

- nombres reals (*float*) i (*double*)

```
/*Una variable de tipus caràcter inicialitzada amb el valor 'A'*/
char c='A' ;
/*Dues variables de tipus enter, una d'inicialitzada i una altra sense inicialitzar*/
int x=0, y;
/*Un nombre real, expressat en punt fix*/
float pi=3.1416;
```

Mida de les variables

En la taula següent es poden observar els tipus de dades bàsics en C, la seva mida en bytes, el rang de valors que poden prendre i els tipus equivalent en assemblador.

Tipus de variable en C	Mida	Rang de valors	Tipus assemblador	Modificador
char	1 byte	–127 a +127	db	BYTE
unsigned char	1 byte	0 a 255	db	BYTE
short int	2 bytes	–32.768 a +32.767	dw	WORD
unsigned short int	2 bytes	0 a 65.535	dw	WORD
int	4 bytes	–2.147.483.648 a +2.147.483.647	dd	DWORD
unsigned int	4 bytes	0 a 4.294.967.295	dd	DWORD
long int	8 bytes	–9.223.372.036.854.775.808 a +9.223.372.036.854.775.807	dq	QWORD
unsigned long int	8 bytes	0 a 18.446.744.073.709.551.615	dq	QWORD

4.2.3. Operadors

Els operadors de C permeten fer operacions aritmètiques, lògiques, relacionals (de comparació) i operacions lògiques bit a bit. Qualsevol combinació vàlida d'operadors, variables i constants s'anomena **expressió**.

Quan es defineix una expressió podem utilitzar parèntesis per a clarificar la manera d'avaluar i canviar la prioritat dels operadors.

Prioritat dels operadors

De més prioritat a menys prioritat:

```
! , ~
++, --
*, /, %
+, -
<<, >>
<, <=, >, >=
==, !=
&
^
|
&&
||
=
```

Podem classificar els operadors de la següent manera:

1) Operador d'assignació

Igualar dues expressions: =

Exemples

```
a = 3;
c = a;
```

2) Operadors aritmètics:

suma: +
resta i negació: –
increment: ++
decrement: --
producte: *
divisió: /
resta de la divisió: %

Exemples

```
a = b + c;
x = y * z
```

3) Operadors de comparació relacionals:

igualtat: = =
diferent: !=
més petit: <
més petit o igual: <=
més gran: >
més gran o igual: >=

4) Operadors de comparació lògics:

I lògica (AND): &&
O lògica (OR): ||
Negació lògica: !

Exemples

```
(a = = b) && (c != 3)
(a <= b) || !(c > 10)
```

5) Operadors lògics:

Operació lògica que es fa bit a bit.

OR (O): |
AND (I): &
XOR (O exclusiva): ^
Negació lògica: ~
Desplaçament a la dreta: >>
Desplaçament a l'esquerra: <<

Exemples

```
z = x | y;
z = x & y;
z = x ^ y;
z = x >> 2;   // desplaçar els bits de la variable x 2 posicions
              //   a la dreta i guardar el resultat a z
z = ~x;       //z és el complement a 1 de x
z = -x;       //z és el complement a 2 de x
```

4.2.4. Control de flux

A continuació s'expliquen les estructures de control de flux més habituals de C:

1) Sentències i blocs de sentències. Una sentència és una línia de codi que finalitzem amb un punt i coma (;). Podem agrupar sentències formant un bloc de sentències utilitzant les claus ({ }).

2) Sentències condicionals

a) if. És la sentència condicional més simple; permet especificar una condició i el conjunt de sentències que s'executaran en cas que es compleixi la condició:

```
if (condició) {
   bloc de sentències
}
```

Si només hem d'executar una sentència no cal utilitzar les claus.

```
if (condició) sentència;
```

La condició serà una expressió de la qual el resultat de l'avaluació sigui 0 (falsa) o diferent de zero (certa).

Exemples

```
if (a > b) {
   printf("a és més gran que b\n");
   a--;
}

if (a >= b) b++;
```

b) if-else. Permet afegir un conjunt d'instruccions que s'executaran en cas que no es compleixi la condició:

```
if (condició) {
     bloc de sentències
}
else {
     bloc de sentències alternatives
}
```

Exemple

```
if (a > b) {
     printf("a és més gran que b\n");
   a--;

}
else
   printf(("a no és més gran que b\n");
```

c) if-else-if. Es poden enllaçar estructures de tipus *if*, afegint sentències *if* noves a continuació.

```
if (condició 1) {
   bloc de sentències que s'executen
   si es compleix la condició 1
}
else  if (condició 2) {
         bloc de sentències que s'executen
         si es compleix la condició 2 i
         no es compleix la condició 1
      }
      else {
         bloc de sentències que s'executen
         si no es compleix cap de les condicions
      }
```

Exemple

```
if (a > b) {
  printf("a és més gran que b\n");
}
else  if (a < b) {
           printf("a és més petit que b\n");
      }
      else {
           printf("a és igual que b\n");
      }
```

3) Estructures iteratives

a) for. Permet fer un bucle controlat per una variable o més que van des d'un valor inicial fins a un valor final i que són actualitzades a cada iteració. El format és:

```
for (valors inicials; condicions; actualització) {
   bloc de sentències a executar
}
```

Si només hem d'executar una sentència no cal utilitzar les claus.

```
for (valors inicials; condicions; actualització) sentència;
```

Exemple

```
//bucle que s'executa 5 cops, mentre x<5, cada cop s'incrementa x en 1
int x;
for (x = 0; x < 5 ; x++) {
   printf("el valor de x és: %d\n", x);
}
```

b) while. Permet expressar un bucle que es va repetint mentre es compleixi la condició indicada. Primer es comprova la condició i si es compleix s'executen les sentències indicades:

```
while (condicions) {
  bloc de sentències a executar
}
```

Si només hem d'executar una sentència no cal utilitzar les claus.

```
while (condicions) sentència;
```

Exemple

```
//bucle que s'executa 5 cops, cal definir prèviament la variable x
int x = 0;
while (x < 5) {
  printf("el valor de x és: %d\n", x);
  x++;    //necessari per a sortir del bucle
}
```

c) do-while. Permet expressar un bucle que es va repetint mentre es compleixi la condició indicada. Primer s'executen les sentències indicades i després es comprova la condició; per tant com a mínim s'executen un cop les sentències del bucle:

```
do {
   bloc de sentències a executar
} while (condicions);
```

Si només hem d'executar una sentència no cal utilitzar les claus:

```
do sentència;
while (condicions);
```

Exemple

```
//bucle que s'executa 5 cops, cal definir prèviament la variable x
int x = 0;
do {
  printf("el valor de x és: %d\n", x);
  x++;
} while (x < 5);
```

4.2.5. Vectors

Els vectors en llenguatge C es defineixen utilitzant un tipus de dada base juntament amb el nombre d'elements del vector indicat entre els símbols *[ ]*.

El format per a un vector unidimensional és el següent:

```
tipus nom_vector [mida];
```

El format per a un vector bidimensional o matriu és el següent:

```
tipus nom_vector [files][columnes];
```

Seguint aquest mateix format es podrien definir vectors de més de dues dimensions.

En declarar un vector es reserva l'espai de memòria necessari per a poder emmagatzemar el nombre d'elements del vector segons el tipus.

Exemple

```
   int vector[5] ;   // vector de 5 enters
   char cadena[4];   // vector de 4 caràcters
   int matriu[3][4]; // matriu de 3 files i 4 columnes: 12 enters
```

També es poden definir vectors donant un conjunt de valors inicials; en aquest cas no cal indicar el nombre d'elements, si es fa no es podrà donar un conjunt de valors inicials superior al valor indicat:

```
int vector[]={1, 2, 3, 4, 5};           // vector de 5 enters
char cadena[4]={'H', 'o', 'l', 'a'};    // vector de 4 caràcters
int vector2[3]={1, 2};                  // vector de 3 enters amb les dues
                                        // primeres posicions inicialitzades
int vector2[3]={1, 2, 3, 4};            // declaració incorrecta
   int matriu[][]={{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}} ;
                                      // matriu d'enters de 3 files i 4 columnes
```

Els vectors de tipus *char* també es poden inicialitzar amb una cadena de caràcters entre cometes dobles. Per exemple:

```
char cadena[]="Hola";
```

Per a accedir a una posició del vector cal indicar l'índex entre claudàtors ([ ]). Cal tenir present que el primer element d'un vector es troba a la posició 0, per tant els índexs aniran des de 0 fins al nombre d'elements menys un.

```
int vector[3]={1, 2, 3};          // índexs vàlids: 0, 1 i 2
                                  // això és equivalent a fer el següent:
   int vector[3];
   vector[0]=1;
   vector[1]=2;
   vector[2]=3;
                                  // o també es pot fer el següent:
int i;
```

```
for (i=0; i<3; i++) vector[i]=i+1;
                                    // Si fem el següent, no donarà error però realment estarem
                                    // accedint fora del vector i pot ocasionar múltiples problemes:
   vector[3]=4;
   vector[20]=300;
```

4.2.6. Apuntadors

Un aspecte molt important del llenguatge C és que permet accedir directament a adreces de memòria. Això s'aconsegueix utilitzant un tipus de variable que es coneix com a *apuntador a memòria*. Un apuntador és una variable que conté una adreça de memòria.

Un apuntador es defineix indicant el tipus de dada al qual apunta i posant el símbol * davant del nom de l'apuntador.

Símbol * en C

El símbol * té múltiples funcions: definir un apuntador, accedir al contingut d'una posició de memòria i també es correspon amb l'operador aritmètic de multiplicació.

Cal tenir present que en definir un apuntador només es reserva espai per a emmagatzemar una adreça de memòria; no es reserva espai per a poder emmagatzemar un valor d'un tipus determinat.

El format general per a definir un apuntador en C és el següent:

```
tipus *nom_apuntador;
```

Utilitzem l'operador & per a obtenir l'adreça de memòria on hi ha emmagatzemada una variable, no el seu contingut. Aquesta adreça la podem assignar a un apuntador.

```
int    *p1;          // Es defineix un apuntador a tipus int
char    *p2;         // Es defineix un apuntador a tipus char

int    x=1, y;       // es defineixen dues variables de tipus int
char    c='a';       // es defineix una variable de tipus char

p1=&x;               // s'assigna a p1 l'adreça de la variable x
p2=&c;               // s'assigna a p2 l'adreça de la variable c
y=*p1;               //com que p1 apunta a x, és equivalent a fer y = x.
```

Quan treballem amb vectors, el nom del vector en realitat és un apuntador, una constant que conté l'adreça del primer element del vector.

```
   int vector[3]={1, 2, 3};
                          //això és equivalent a fer el següent:
   int vector[3];
   int *v;
```

```
   int i;
   v = vector;            //vector i *v són apuntadors a enter.
for (i=0; i<3; i++) *(v+i)=i+1;
                          // *(v+i): indica que estem accedint al contingut de l'adreça 'v+i'
```

4.2.7. Funcions

Una funció és un bloc de codi que fa una tasca concreta. A part de la funció *main*, un programa pot definir i implementar altres funcions.

El format general d'una funció és el següent:

```
tipus_del_retorn nom_funció( llista_de_paràmetres ){

     definició de variables;
     sentències;
     return valor;

}
```

on:

- `tipus_del_retorn`: tipus de la dada que retornarà la funció; si no s'especifica el tipus de retorn, per defecte és de tipus enter (*int*).
- `llista_de_paràmetres`: llista de tipus i noms de variables separats per comes; no és obligatori que la funció tingui paràmetres.
- `return`: finalitza l'execució de la funció i retorna un valor del tipus especificat en el camp `tipus_del_retorn`. Si no s'utilitza aquesta sentència o no s'especifica un valor, retornarà un valor indeterminat.

Exemple

Definim una funció que sumi dos enters i retorni el resultat de la suma:

```
int funcioSuma(int a, int b){
   int resultat;       //variable local

   resultat = a + b;   //sentència de la funció
   return resultat;    //valor de retorn
}
```

Es mostra a continuació com quedaria un programa complet que utilitzés aquesta funció.

```
                                  // fitxer suma.c
#include <stdio.h>

int funcioSuma(int a, int b){
   int resultat;                  //variable local

   resultat = a + b;              //sentència de la funció
   return resultat;               //valor de retorn
}

int main(){
  int x, y, r;                    //variables locals
  printf ("\nIntrodueix el valor de x: ");
  scanf("%d",&x);
  printf ("Introdueix el valor de y: ");
  scanf("%d",&y);
  r=funcioSuma(x,y);              //cridem la funció que hem definit
  printf("La suma de x i y és: %d\n", r);
}
```

Compilació i execució

```
$ gcc –o suma suma.c
$ ./suma
Introdueix el valor de x: 3
Introdueix el valor de y: 5
La suma de x i y és: 8
$ _
```

4.2.8. Funcions d'E/S

Es descriuen a continuació les funcions bàsiques d'E/S, per a escriure per pantalla i per a llegir per teclat i que estan definides en el fitxer *stdio.h*:

Nota

El repertori de funcions d'E/S és molt extens, podreu trobar molta informació de totes aquestes funcions en llibres de programació en C i per Internet.

Funció *printf*

printf permet escriure en pantalla informació formatada, permet visualitzar cadenes de caràcters constants, juntament amb el valor de variables.

El format general de la funció és el següent:

```
printf("cadena de control"[, llista_de_paràmetres])
```

La cadena de control inclou dos tipus d'elements:

- Els caràcters que volem mostrar per pantalla.
- Ordres de format que indiquen com es mostraran els paràmetres.

La llista de paràmetres està formada per un conjunt d'elements que poden ser expressions, constants i variables, separades per comes.

Hi ha d'haver el mateix nombre d'ordres de format que de paràmetres, s'han de correspondre en ordre i el tipus de l'ordre amb el tipus de la dada.

Les ordres de format comencen amb un símbol % seguit d'un caràcter o més. Les més habituals són:

`%d`	Per a mostrar un valor enter, el valor d'una variable de tipus *char* o *int*.
`%ld`	Per a mostrar un valor enter llarg, el valor d'una variable de tipus *long*.
`%c`	Per a mostrar un caràcter, el contingut d'una variable de tipus *char*.

`%s`	Per a mostrar una cadena de caràcters, el contingut d'un vector de tipus *char*.
`%f`	Per a mostrar un nombre real, el valor d'una variable de tipus *float* o *double*. Es pot indicar el nombre de dígits de la part entera i de la part decimal, utilitzant l'expressió següent: `%[dígits enters].[dígits decimals]f`
`%p`	Per a mostrar una adreça de memòria, per exemple el valor d'un apuntador.

Dins la cadena de control es poden incloure alguns caràcters especials, començant amb un símbol \ i seguit d'un caràcter o més. Els més habituals són:

`\n`	caràcter de salt de línia
`\t`	caràcter de tabulació

Si només es vol mostrar una cadena de caràcters constant, no cal especificar cap paràmetre.

Exemples

```
int x=5;
float pi=3.1416;
char msg[]="Hola", c='A';

printf("Hola!");
printf ("Valor de x: %d. Adreça de x: %p\n", x, &x);
printf ("El valor de PI amb 1 enter i 2 decimals: %1.2f\n", pi);
printf ("Contingut de msg: %s. Adreça de msg: %p\n", msg, msg);
printf ("Valor de c: %c. El codi ASCII guardat a c: %d\n", c, c);
printf ("Constant entera: %d, i una lletra: %c", 100, 'A');
```

Funció *scanf*

scanf permet llegir valors introduïts per teclat i emmagatzemar-los en variables de memòria.

La lectura d'una dada acaba quan es troba un caràcter d'espai en blanc, un tabulador o un ENTER. Si la crida a la funció sol·licita la lectura de diverses variables, aquests caràcters tenen la funció de separadors de camp, però caldrà finalitzar la lectura amb un ENTER.

El format general de la funció és:

```
scanf("cadena de control", llista_de_paràmetres)
```

La cadena de control està formada per un conjunt d'ordres de format. Les ordres de format són les mateixes definides per a la funció *printf*.

La llista de paràmetres ha d'estar formada per adreces de variables en què es guardaran els valors llegits; les adreces se separen per comes.

Per a indicar l'adreça d'una variable, s'utilitza l'operador & davant del nom de la variable. Recordeu que el nom d'un vector es refereix a l'adreça del primer element del vector, per tant no és necessari l'operador &.

```
int x;
float f;
char msg[5];

scanf("%d %f", &x, &f);     // Es llegeix un enter i un real
                            //si escrivim: 2 3.5 i es prem ENTER, s'assignarà x=2 i f=3.5
                            //farà el mateix si fem: 2 i es prem ENTER, 3.5 i es prem ENTER
scanf("%s", msg);           // En llegir una cadena s'afegeix un \0 al final
```

5. Conceptes de programació en assemblador i C

5.1. Accés a dades

Per a accedir a dades de memòria en assemblador, com passa en els llenguatges d'alt nivell, ho farem per mitjà de variables que caldrà definir prèviament per a reservar l'espai necessari per a emmagatzemar la informació.

En la definició podem especificar el nom de la variable, la mida i un valor inicial; per a accedir a la dada ho farem amb els operands de les instruccions especificant un mode d'adreçament determinat.

Per exemple, `var dd 12345678h` és la variable amb el nom *var* de mida 4 bytes inicialitzada amb el valor 12345678h. `mov eax, dword[var]` és la instrucció en què accedim a la variable *var* usant adreçament directe a memòria, en què especifiquem que volem accedir a 4 bytes (dword) i en transferim el contingut, el valor 12345678h, al registre *eax*.

En assemblador cal estar molt alerta quan accedim a les variables que hem definit. Les variables es guarden a memòria consecutivament a mesura que les declarem i no hi ha res que limiti les unes amb les altres.

Exemple

```
.data
  var1 db 0          ;variable definida d'1 byte
  var2 db 61h        ;variable definida d'1 byte
  var3 dw 0200h      ;variable definida de 2 bytes
  var4 dd 0001E26Ch  ;variable definida de 4 bytes
```

Les variables es trobaran a memòria tal com mostra la taula.

Adreça	Valor
var1	00h
var2	61h
var3	00h
	02h
var4	6Ch
	E2h
	01h
	00h

Si executem la instrucció següent:

```
mov eax, dword[var1]
```

Quan accedim a `var1` com una variable de tipus `DWORD`
el processador agafarà com a primer byte el valor de `var1` però també els 3 bytes que hi ha a continuació, per tant, com que les dades es tracten en format *little-endian*, considerarem `DWORD [var1] = 02006100h` i aquest és el valor que portarem a EAX (eax=02006100h). Si aquest accés no és el desitjat, el compilador no reportarà cap error, ni tindrem un error durant l'execució; només podrem detectar que ho estem fent malament provant el programa i depurant.

D'una banda, tenim que l'accés a les dades és molt flexible, però, de l'altra, que si no controlem molt bé l'accés a les variables, aquesta flexibilitat ens pot portar certs problemes.

En llenguatge C tenim un comportament semblant. Suposem que tenim el codi següent:

```
int vec[4]={1,2,3,4};
int x=258; //258=00000102h
char c=3;

int main() {
   int i=0;
   for (i=0;i<4;i++) printf("contingut de vec ", vec[i]);
   c = x;
}
```

L'índex `i` pren els valors 0, 1, 2, 3 i 4, l'índex `i=4` no correspon a una posició del vector, la primera posició és la 0 i la darrera és la 3; per tant, estem accedint a una posició fora del vector. En compilar el codi no es generarà cap error i tampoc es produirà cap error durant l'execució. Si la variable `x` ocupés la posició de memòria següent a continuació dels 4 elements del vector es mostraria un 258.

En l'assignació `c = x` com que els tipus són diferents, `x` (int de 4 bytes) i `c` (char de 1 byte), assigna el byte menys significatiu de `x` a la variable `c`, després de l'assignació `c = 2`. Si féssim l'assignació al revés, `x = c`, faria l'extensió de signe de `c` a 4 bytes i, per tant, després de l'assignació `x = 3 (00000003h)`.

Però a diferència de l'assemblador, en què les variables es troben a la memòria en el mateix ordre en què les declarem, en C això no ho podem assegurar, ja que la reserva d'espai de memòria per a les variables que fa el compilador GCC és diferent.

En assemblador podem accedir a variables globals definides en un codi escrit en C, si compilem conjuntament el codi C i el codi assemblador. Per accedir a una variable de C només cal declarar-la en assemblador amb la directiva *extern* (vegeu l'apartat 3.2.3).

Per exemple, suposem que tenim definides les següents variables en C:

```
char vec_char[4];
int vec_int[4];
long int l_int;
short int s_int;
```

Per poder accedir a aquestes variables des d'assemblador afegiríem el següent:

```
;Variables definides en C que s'utilitzen en assemblador
extern vec_char, vec_int, l_int, s_int
```

A l'hora d'accedir a les variables definides en C cal tenir present la seva mida (vegeu "mida dels operands" al final de l'apartat 3.4.1. i "mida de les variables" al final de l'apartat 4.2.2).

5.1.1. Estructures de dades

Vegem com accedir a vectors i matrius utilitzant llenguatge C i llenguatge d'assemblador.

Els modes d'adreçament que habitualment s'utilitzen per a accedir a vectors i matrius són l'adreçament indirecte, relatiu, indexat i la combinació de relatiu i indexat.

Per a accedir a cada element del vector o de la matriu cal tenir present la mida del tipus de dada utilitzada per a definir-los, tant per a obtenir la dada d'un element, com per a accedir a l'element següent.

En vectors de tipus `char`, assignarem el valor de cada dada a registres de mida 1 byte i per a passar a l'element següent caldrà fer increments d'una unitat. En tipus `int`, assignarem el valor a registres de mida 4 bytes i caldrà fer increments de 4 unitats.

Vegem alguns exemples de com accedir a vectors definits en C des d'assemblador utilitzant els modes d'adreçament comentats anteriorment.

Utilitzarem el vector `vec_char` de tipus *char*, on cada element ocupa 1 byte (1 posició de memòria) i el vector `vec_int` de tipus *int*, on cada element ocupa 4 bytes (4 posicions de memòria).

```
;adreçament indirecte
```

```
mov edx, vec_int               ;Agafem l'adreça del vector
mov DWORD [edx], 0             ;Posem a 0 el 1r. element del vector vec_int
add edx, 4                     ;Sumem 4 per accedir al següent element del vector
add DWORD [edx], 0             ;Posem a 0 el 2n. element del vector vec_int
add edx, 4                     ;Sumem 4 per accedir al següent element del vector
...

;adreçament relatiu
mov al, byte [vec_char+0]      ;1r. element del vector, vec_char[0]
mov bl, byte [vec_char+1]      ;2n. element del vector, vec_char[1]

mov eax, dword [vec_int+0]     ;1r. element del vector, vec_int[0]
mov ebx, dword [vec_int+12]    ;4r. element del vector, vec_int[3]

;adreçament indexat
   mov esi, 0                  ;Inicialitzem l'índex per a accedir al vector.
  inicialitza:
   mov dword [vec_int+esi],0   ;Posem a 0 un element del vector
   add esi, 4                  ;Sumem 4 per a accedir al següent element del vector
   cmp esi, 12                 ;Accedim a vec_int[0], vec_int[1]
   jle inicialitza             ;vec_int[2] i vec_int[3]

;combinació indexat i relatiu
   mov edx, vec_int            ;Agafem l'adreça del vector
   mov esi, 0                  ;Inicialitzem l'índex per a accedir al vector.
  inicialitza:
   mov dword [edx+esi*4],0     ;Posem a 0 un element del vector
   add esi, 1                  ;Sumem 1 per a accedir al següent element del
                               ;vector perquè utilitzem el factor d'escala 4
   cmp esi, 3                  ;Accedim a vec_int[0], vec_int[1]
   jle inicialitza             ;vec_int[2] i vec_int[3]
```

Definim en C un vector i una matriu i fem la suma dels elements.

```
int main(){

  int vec[6]={1,2,3,4,5,6},mat[2][3]={{1,2,3},{4,5,6}};
  int i,j, sumaVec=0, sumaMat=0;

  for (i=0;i<6;i++) sumaVec=sumaVec+vec[i];

  for (i=0;i<2;i++){
    for(j=0;j<3;j++) sumaMat=sumaMat+mat[i][j];
  }

}
```

A continuació vegem com fer el mateix amb llenguatge d'assemblador utilitzant diferents variants de l'adreçament indexat (el format general és `[adreça + Registre Base + Registre Índex * escala]`):

```
  vec dd 1,2,3,4,5,6
   mat dd 1,2,3
       dd 4,5,6

                                    ;recorrem del vector per a sumar-lo
   mov esi,0                        ;esi serà l'índex per a accedir a les dades
   mov eax,0                        ;eax serà on guardarem la suma
loop_vec:
   add eax, dword[vec+esi*4]        ;multipliquen l'índex per 4
   inc esi                          ;perquè cada element ocupa 4 bytes.
   cmp esi, 6                       ;comparem amb 6 perquè és l'índex del primer
                                    ;element fora del vector.
   jl  loop_vec

                                    ;recorrem la matriu amb un sol índex per a sumar-la
   mov esi,0                        ;esi serà l'índex per a accedir a les dades
   mov ebx,0                        ;ebx serà on guardarem la suma
loop_mat:
   add ebx, dword[mat+esi*4]        ;multipliquen l'índex per 4
   inc esi                          ;perquè cada element ocupa 4 bytes.
   cmp esi, 6                       ;comparem amb 6 perquè és l'índex del primer element
                                    ;fora de la matriu, la matriu té 2*3=6 elements.
   jl  loop_mat

                                    ;recorrem la matriu amb dos índexs per a sumar-la
   mov esi,0                        ;esi serà l'índex de la columna
   mov edi,0                        ;edi serà l'índex de la fila
   mov ebx,0                        ;ebx serà on guardarem la suma
loop_mat2:
   add ebx, dword[mat+edi+esi*4]    ;multipliquen l'índex de la columna
   inc esi                          ;per 4 perquè cada element ocupa 4 bytes.
   cmp esi, 3                       ;comparem amb 3 perquè són els elements d'una fila.
   jl  loop_mat2
   mov esi, 0
   add edi, 12                      ;cada fila ocupa 12 bytes, 3 elements de 4 bytes.
   cmp edi, 24                      ;comparem amb 24 perquè és el primer índex
   jl  loop_mat2                    ;fora de la matriu.
```

Com podem comprovar, el codi per a accedir al vector i a la matriu utilitzant un índex són idèntics, en assemblador les matrius es veuen com a vectors i totes les posicions de memòria són consecutives; per tant, si volem accedir a una posició concreta d'una matriu, a partir d'un número de fila i de columna, haurem de calcular a quina posició de memòria cal accedir; fila × elements_de_la_fila × mida_de_la_dada + columna × mida_de_la_dada.

Per exemple, si volem accedir a l'element mat[1][2], segona fila, tercera columna (les files i columnes es comencen a numerar per 0), aquest element serà a la posició $1 \times 3 \times 4 + 2 \times 4 = 20$, [mat+20].

Cal tenir present que l'assemblador ens ofereix diferents modes d'adreçament que ens poden facilitar l'accés a determinades estructures de dades com són els vectors i les matrius.

En l'exemple anterior, en què s'utilitzen dos índexs per a accedir a la matriu, un índex (edi) l'utilitzem per a la fila i l'altre índex (esi) per a la columna, de manera que el recorregut per la matriu és el següent:

```
edi = 0
[mat+0+0*4]     [mat+0+1*4]     [mat+0+2*4]
edi = 12
[mat+12+0*4]    [mat+12+1*4]    [mat+12+2*4]
```

5.1.2. Gestió de la pila

La pila és una zona de memòria que s'utilitza per a emmagatzemar informació de manera temporal. La pila també s'utilitza habitualment per a passar paràmetres a les subrutines i per a guardar els registres que són modificats dins d'una subrutina, de manera que se'n pugui restaurar el valor abans de finalitzar-ne l'execució.

Es tracta d'una estructura de dades de tipus LIFO (*last in first out*): el darrer element introduït, que habitualment s'anomena *cim de la pila*, és el primer element que es treu i és l'únic directament accessible.

En els processadors x86-64, la pila s'implementa en memòria principal a partir d'una adreça base. S'utilitza el registre RSP com a apuntador al cim de la pila.

La pila creix cap a adreces més petites; és a dir, cada cop que s'introdueix un valor a la pila aquest ocupa una adreça de memòria més petita, per tant, el registre RSP es decrementa per tal que apunti al nou valor introduït i s'incrementa quan el traiem.

Els elements s'introdueixen i es treuen de la pila utilitzant instruccions específiques: PUSH per a introduir elements i POP per a treure elements.

En el mode de 64 bits els elements que es poden posar i treure de la pila han de ser valors de 16 o 64 bits; per tant, el registre RSP es decrementa o incrementa en 2 o 8 unitats respectivament.

En executar la instrucció PUSH, s'actualitza el valor de RSP decrementant-se en 2 o 8 unitats i es porta la dada especificada per l'operand cap al cim de la pila. En executar la instrucció POP, es porta el valor que hi ha al cim de la pila cap a l'operand de la instrucció i s'actualitza el valor de RSP incrementant-se en 2 o 8 unitats.

Exemple

Suposem que rax = 0102030405060708h (registre de 64 bits),i bx = 0A0Bh (registre de 16 bits) i rsp = 0000000010203050h

```
push rax
push bx
pop  bx
pop  rax
```

Evolució de la pila en executar aquestes quatre instruccions. En la taula es mostra l'estat de la pila després d'executar cada instrucció.

Adreça de memòria	Estat inicial	push rax	push bx	pop bx	pop rax
10203044h					
10203045h					
10203046h			**0B**		
10203047h			**0A**		
10203048h		**08**	08	08	
10203049h		**07**	07	07	
1020304Ah		**06**	06	06	
1020304Bh		**05**	05	05	
1020304Ch		**04**	04	04	
1020304Dh		**03**	03	03	
1020304Eh		**02**	02	02	
1020304Fh		**01**	01	01	
10203050h	xx	xx	xx	xx	xx
RSP	10203050h	10203048h	10203046h	10203048h	10203050h

- `push rax`. Decrementa RSP en 8 unitats i porta el valor del registre RAX a partir de la posició de memòria indicada per RSP; funcionalment la instrucció anterior seria equivalent a:

```
sub rsp, 8
mov qword[rsp], rax
```

- `push bx`. Decrementa RSP en 2 unitats i porta el valor del registre BX a partir de la posició de memòria indicada per RSP; funcionalment la instrucció anterior seria equivalent a:

```
sub rsp, 2
mov word[rsp], bx
```

- `pop bx`. Porta cap a BX el valor de 2 bytes emmagatzemat a partir de la posició de memòria indicada per RSP, a continuació s'incrementa RSP en 2. Seria equivalent a fer:

```
mov bx, word [rsp]
add rsp, 2
```

- `pop rax`. Porta cap a RAX el valor de 8 bytes emmagatzemat a partir de la posició de memòria indicada per RSP, a continuació s'incrementa RSP en 8. Seria equivalent a fer:

```
mov rax, qword [rsp]
add rsp, 8
```

5.2. Operacions aritmètiques

En aquest subapartat cal tenir present que en C podem construir expressions utilitzant variables, constants i operadors en una mateixa sentència; en canvi fer el mateix en assemblador implica escriure una seqüència d'instruccions en què caldrà fer les operacions una a una segons la prioritat dels operadors dins la sentència.

Exemple

Sentència que podem expressar en llenguatge C:

```
r=(a+b)*4 / (c>>2);
```

Traducció de la sentència en llenguatge d'assemblador:

```
mov eax, [a]
mov ebx, [b]
add eax, ebx   ;(a+b)
imul eax, 4    ;(a+b)*4
mov ecx,[c]
sar ecx, 2     ;(c>>2)
idiv ecx       ;(a+b)*4 / (c>>2)
mov [r], eax   ;r=(a+b)*4 / (c>>2)
```

5.3. Control de flux

En aquest subapartat veurem com les diferents estructures de control del llenguatge C es poden traduir a llenguatge d'assemblador.

5.3.1. Estructura *if*

Estructura condicional expressada en llenguatge C:

```
if (condició) {
   bloc de sentències
}
```

Exemple

```
if (a > b) {
   maxA = 1;
   maxB = 0;
}
```

Es pot traduir a llenguatge d'assemblador de la manera següent:

```
mov rax, qword [a]   ;Es carreguen les variables en registres
mov rbx, qword [b]

cmp rax, rbx   ;Es fa la comparació
jg cert        ;Si es compleix la condició salta a l'etiqueta cert
jmp fi         ;Si no es compleix la condició salta a l'etiqueta fi

cert:
   mov byte [maxA], 1   ;Aquestes instruccions només s'executen
   mov byte [maxB], 0   ;quan es compleix la condició

fi:
```

El codi es pot optimitzar si s'utilitza la condició contrària a la que apareix en el codi C (a <= b)

```
   mov rax, qword [a]   ;Es carrega només la variable a en un registre
   cmp rax, qword [b]   ;Es fa la comparació
   jle fi              ;Si es compleix la condició (a<=b) salta a fi

   mov byte [maxA], 1   ;Aquestes instruccions només s'executen
   mov byte [maxB], 0   ;quan es compleix la condició (a > b)
fi:
```

Si tenim una condició més complexa, primer caldrà avaluar l'expressió de la condició per a decidir si executem o no el bloc de sentències de *if*.

Exemple

```
if ((a != b) || (a>=1 && a<=5)) {
   b = a;
}
```

Es pot traduir a llenguatge d'assemblador de la manera:

```
mov rax, qword [a]       ;Es carreguen les variables en registres
mov rbx, qword [b]


cmp rax, rbx             ;Es fa la comparació (a != b)
jne cert                 ;Si es compleix la condició salta a l'etiqueta cert
                         ;Si no es compleix com que és una OR es pot complir l'altra condició
cmp rax, 1               ;Es fa la comparació (a >= 1), si no es compleix
jl fi                    ;com que és una AND no cal mirar l'altra condició
cmp rax, 5               ;Es fa la comparació (a <= 5)
jg fi                    ;Si no salta es compleix que (a>=1 && a<=5)


cert:
   mov qword [b], rax    ;Fem l'assignació quan la condició és certa.
```

```
fi:
```

5.3.2. Estructura *if-else*

Estructura condicional expressada en llenguatge C considerant la condició alternativa:

```
if (condició) {
     bloc de sentències
}
else {
     bloc de sentències alternatives
}
```

Exemple

```
if (a > b) {
   max = 'a';
   }
else { // (a <= b)
   max = 'b';
}
```

Es pot traduir a llenguatge d'assemblador de la manera:

```
   mov rax, qword [a]    ;Es carreguen les variables en registres
   mov rbx, qword [b]
   cmp rax, rbx          ;Es fa la comparació
   jg cert               ;Si es compleix la condició se salta a cert

   mov byte [max], 'b'   ;else (a <= b)
   jmp fi

cert:
   mov byte [max], 'a'   ;if (a > b)

fi:
```

5.3.3. Estructura *while*

Estructura iterativa controlada per una condició expressada al principi:

```
while (condicions) {
  bloc de sentències a executar
}
```

Exemple

```
resultat=1;
while (num > 1){ //mentre num sigui més gran que 1 fer ...
   resultat = resultat * num;
   num--;
}
```

Es pot traduir a llenguatge d'assemblador de la manera:

```
  mov rax, 1               ;rax serà [resultat]
  mov rbx, qword [num]     ;Es carrega la variable en un registre
```

```
while:
  cmp rbx, 1               ;Es fa la comparació
  jg cert                  ;Si es compleix la condició (num > 1) salta a cert
  jmp fi                   ;sinó salta a fi
cert:
   imul rax, rbx           ;rax = rax * rbx
   dec rbx
   jmp while
fi:
   mov qword [resultat], rax
   mov qword [num], rbx
```

Una versió alternativa que utilitza la condició contrària (num <= 0) i treballa directament amb una variable de memòria:

```
  mov rax, 1               ;rax serà [resultat]
while:
   cmp qword [num], 1      ;Es fa la comparació
   jle fi                  ;Si es compleix la condició (num <= 1) salta a fi
   imul rax, qword [num]   ;rax=rax*[num]
   dec qword [num]
   jmp while
fi:
   mov qword [resultat], rax
```

En aquest cas, reduïm tres instruccions del codi, però augmentem els accessos a memòria durant l'execució del bucle.

5.3.4. Estructura *do-while*

Estructura iterativa controlada per una condició expressada al final:

```
do {
  bloc de sentències a executar
} while (condicions);
```

Exemple

```
resultat = 1;
do {
   resultat = resultat * num;
   num--;
} while (num > 1)
```

Es pot traduir a llenguatge d'assemblador de la manera:

```
  mov rax, 1            ;rax serà [resultat]
  mov rbx, qword [num]   ;Es carrega la variable en un registre
while:
  imul rax, rbx
  dec rbx
  cmp rbx, 1             ;Es fa la comparació
  jg while               ;Si es compleix la condició salta a while

   mov qword [resultat], rax
   mov qword [num], rbx
```

5.3.5. Estructura *for*

Estructura iterativa, utilitzant l'ordre *for*:

```
for (valors inicials; condicions; actualització) {
   bloc de sentències a executar
}
```

Exemple

```
resultat=1;
for (i = num; i > 1; i--)
{
   resultat=resultat*i;
}
```

Es pot traduir a llenguatge d'assemblador de la manera:

```
   mov rax, 1             ;rax serà [resultat]
   mov rcx, qword [num]   ;rcx serà [i] que inicialitzem amb [num]

for:
  cmp rcx, 1              ;Es fa la comparació
  jg cert                 ;Si es compleix la condició salta a cert
  jmp fi
cert:
  imul rax,rcx
  dec rcx
  jmp for
fi:
   mov qword [resultat], rax
   mov qword [i], rcx
```

En sortir de l'estructura iterativa, no actualitzem la variable *num* perquè s'utilitza per a inicialitzar *i* però no es modifica.

5.3.6. Estructura switch-case

Estructura de selecció, que amplia el nombre d'opcions de l'estructura if-else i permet múltiples opcions. S'utilitza una variable que es compara per igualtat successivament amb diferents valors constants, es permet indicar una o diverses sentències per cada cas, finalitzades per la sentència *break*, i especificar una o diverses sentències que s'executin en cas de no coincidir amb cap dels valors indicats, *default*.

```
switch (variable) {
  case valor1:  sentència1;
```

```
                break;
  case valor2:  sentència2;
                ...
                break;
                ...
  default:      sentènciaN;
}
```

Exemple

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

Es pot traduir a llenguatge assemblador de la manera següent:

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

5.4. Subrutines i pas de paràmetres

Una subrutina és una unitat de codi autocontinguda dissenyada per a dur a terme una tasca determinada i té un paper determinant en el desenvolupament de programes de manera estructurada.

Una subrutina en assemblador seria l'equivalent a una funció en C. En aquest punt veurem com definir subrutines en assemblador i com les podem utilitzar després des d'un programa en C.

Primer descriurem com treballar amb subrutines en assemblador:

- Definició de subrutines en assemblador
- Crida i retorn de subrutina
- Pas de paràmetres a la subrutina i retorn de resultats

A continuació veurem com fer crides a subrutines fetes en assemblador des de codi assemblador i des de codi C i quines implicacions té en el pas de paràmetres.

5.4.1. Definició de subrutines en assemblador

Bàsicament una subrutina és un conjunt d'instruccions que inicien la seva execució en un punt de codi identificat amb una etiqueta que serà el nom de la subrutina, i finalitza amb l'execució d'una instrucció *ret*, instrucció de retorn de subrutina, que provoca un salt a la instrucció següent des d'on s'ha fet la crida (*call*).

L'estructura bàsica d'una subrutina seria:

```
subrutina:
   ;
   ; Instruccions de la subrutina
   ;
   ret
```

Consideracions importants a l'hora de definir una subrutina:

- Cal emmagatzemar els registres modificats dins de la subrutina per a deixar-los en el mateix estat en què es trobaven en el moment de fer la crida a la subrutina, llevat dels registres que s'utilitzin per a retornar un valor. Per a emmagatzemar els registres modificats utilitzarem la pila.

- Per a mantenir l'estructura d'una subrutina i perquè el programa funcioni correctament no es poden fer salts a instruccions fora de la subrutina; sempre finalitzarem l'execució de la subrutina amb la instrucció *ret*.

```
subrutina:
   ; Emmagatzemar a la pila
   ; els registres modificats dins la subrutina
   ;
   ; Instruccions de la subrutina
   ;
   ; Restaurar l'estat dels registres modificats
   ; recuperant el seu valor inicial emmagatzemat a la pila
   ret
```

Exemple de subrutina que calcula el factorial d'un nombre

```
factorial:
  push rax                ; Emmagatzemem a la pila els registres
  push rbx                ; modificats dins la subrutina

                          ; Instruccions de la subrutina
  mov rax, 1              ; rax serà el resultat
  mov rbx, 5              ; Calculem el factorial del valor de rbx (=5).
while:
  cmp rbx, 1              ; Fem la comparació
  jle fi                  ; Si es compleix la condició saltem a fi
  imul rax, rbx
  dec rbx
  jmp while               ; Salta a while
fi:                       ; A rax tindrem el valor del factorial de 5 (=120)
  mov qword[result], rax  ; Emmagatzemem el resultat en la variable 'result'

  pop rbx                 ; Restaurem el valor inicial dels registres
  pop rax                 ; en ordre invers a com els hem emmagatzemat

  ret                     ; Finalitza l'execució de la subrutina
```

5.4.2. Crida i retorn de subrutina en assemblador

Per a fer la crida a la subrutina s'utilitza la instrucció `call`, i s'indica l'etiqueta que defineix el punt d'entrada a la subrutina:

```
call factorial
```

La instrucció *call* emmagatzema a la pila l'adreça de retorn (l'adreça de la instrucció que es troba a continuació de la instrucció *call*) i llavors transfereix el control del programa a la subrutina, carregant al registre RIP l'adreça de la primera instrucció de la subrutina.

Funcionalment la instrucció *call* anterior seria equivalent a:

```
sub rsp, 8
mov qword[rsp], rip
mov rip, factorial
```

Per a finalitzar l'execució de la subrutina, executarem la instrucció *ret*, que recupera de la pila l'adreça del registre RIP que hem emmagatzemat en fer *call* i la carrega un altre cop en el registre RIP; continua l'execució del programa amb la instrucció que hi ha després del *call*.

Funcionalment la instrucció *ret* seria equivalent a:

```
mov rip, qword[rsp]
add rsp, 8
```

Si en fer el *ret* no tenim al cim de la pila l'adreça de retorn que hem emmagatzemat quan hem fet el *call*, per una mala gestió de la pila dins la subrutina, la instrucció *ret* utilitzarà el valor del cim de la pila com a adreça de retorn i es perdrà el fil d'execució del nostre programa.

Per a assegurar-nos que una mala gestió de la pila dins la subrutina no afecti l'execució del codi es recomana guardar l'estat de la pila a l'inici i recuperar-lo abans de sortir:

```
push rbp       ; Emmagatzemar el registre rbp a la pila
mov rbp, rsp   ; Assignar a rbp el valor del registre apuntador rsp


...


mov rsp, rbp   ; Restaurem el valor inicial de rsp amb rbp
pop rbp        ; Restaurem el valor inicial de rbp


ret
```

D'aquesta forma ens assegurem que abans de fer el *ret* tenim al cim de la pila l'adreça de retorn emmagatzemada quan hem fet el *call*.

Exemple de crida d'una subrutina en assemblador

```
section .data
  x   dq 5                ; Declarem les variables que utilitzarem
  result dq 0

section .text

  global main             ; Fem visible l'etiqueta main

factorial:
  push rbp                ; Emmagatzemar el registre que farem servir d'apuntador
                          ; a la pila rbp
  mov rbp, rsp            ; Assignar a rbp el valor del registre apuntador rsp

  push rax                ; Emmagatzemem a la pila els registres
  push rbx                ; modificats dins la subrutina

  mov rax, 1              ; rax serà el resultat
  mov rbx, qword[x]       ; Calculem el factorial de la variable 'x' (=5).
while:
  cmp rbx, 1              ; Fem la comparació
  jle fi                  ; Si es compleix la condició saltem a fi
  imul rax, rbx
  dec rbx
  jmp while               ; Salta a while
fi:                       ; A rax tindrem el valor del factorial de 'x' (=120)
  mov qword[result], rax  ; Emmagatzemem el resultat en la variable 'result'

  pop rbx                 ; Restaurem el valor inicial dels registres
  pop rax                 ; en ordre invers a com els hem emmagatzemat

  mov rsp, rbp            ; Restaurem el valor inicial de rsp amb rbp
  pop rbp                 ; Restaurem el valor inicial de rbp

  ret                     ; Finalitza l'execució de la subrutina


main:
  call factorial          ; Cridem a la subrutina factorial

  mov rax, 1
  mov rbx, 0
  int 80h                 ; Finalitza l'execució del programa
```

5.4.3. Pas de paràmetres a la subrutina i retorn de resultats

Una subrutina pot necessitar que li passin paràmetres; els paràmetres es poden passar mitjançant registres o la pila. Passa el mateix amb el retorn de resultats, pot ser per mitjà de registre o de la pila. Considerarem els casos en què el nombre de paràmetres d'entrada i de retorn d'una subrutina són fixos.

Pas de paràmetres i retorn de resultat per mitjà de registres

Cal definir sobre quins registres concrets volem passar paràmetres a la subrutina i sobre quins registres farem el retorn; podem utilitzar qualsevol registre de propòsit general del processador.

Un cop definits quins registres utilitzarem per a fer el pas de paràmetres, haurem d'assignar a cada un el valor que volem passar a la subrutina abans de fer el *call*; per a retornar els valors, dins de la subrutina, haurem d'assignar als registres corresponents el valor que s'ha de retornar abans de fer el *ret*.

Recordeu que els registres que s'utilitzin per a retornar un valor no s'han d'emmagatzemar a la pila a l'inici de la subrutina, ja que no cal conservar-ne el valor inicial.

Suposem que en l'exemple del factorial volem passar com a paràmetre el nombre del qual volem calcular el factorial i retornar com a resultat el factorial del nombre passat com a paràmetre, implementant el pas de paràmetres i el retorn de resultats per mitjà de registres.

El nombre que volem calcular el factorial el passarem per mitjà del registre RBX i retornarem el resultat sobre el registre RAX.

La crida de la subrutina serà:

```
mov rbx, qword[x]       ; Posem al registre rbx el valor de la variable 'x'
                        ; com a paràmetre d'entrada
call factorial          ; Cridem a la subrutina factorial
                        ; A rax tindrem el valor del factorial de 'x' com a
mov qword[result],rax   ; paràmetre de sortida i l'assignem a la variable 'result'
```

Subrutina:

```
factorial:
  push rbp          ; Emmagatzemar el registre que farem servir d'apuntador a la pila rbp
  mov rbp, rsp      ; Assignar a rbp el valor del registre apuntador rsp

  push rbx          ; Emmagatzemar a la pila el registre que modifiquem
                    ; i que no s'utilitza per a retornar el resultat
```

```
  mov rax, 1        ; rax serà el resultat
while:
  cmp rbx, 1        ; Fem la comparació
  jle fi            ; Si es compleix la condició saltem a fi
  imul rax, rbx
  dec rbx
  jmp while         ; Salta a while
fi:                 ; A rax tindrem el valor del factorial de rbx
  pop rbx           ; Restaurem el valor inicial del registre

  mov rsp, rbp      ; Restaurem el valor inicial de rsp amb rbp
  pop rbp           ; Restaurem el valor inicial de rbp

  ret
```

Pas de paràmetres i retorn de resultat per mitjà de la pila

Per a passar paràmetres i retornar resultats a una subrutina utilitzant la pila, i un cop definits quins paràmetres volem passar i quins volem retornar, cal fer el següent:

1) Abans de fer la crida a la subrutina: cal reservar espai a la pila per a les dades que volem retornar i a continuació introduir els paràmetres necessaris a la pila.

2) Dins la subrutina: cal accedir als paràmetres llegint-los directament de memòria utilitzant un registre que apunti al cim de la pila. El registre apuntador de pila, RSP, sempre apunta al cim de la pila i per tant podem accedir al contingut de la pila fent un adreçament a memòria que utilitzi RSP, però si utilitzem la pila dins la subrutina no es recomana utilitzar-lo.

El registre que se sol utilitzar com a apuntador per a accedir a la pila és el registre RBP. Abans d'utilitzar-lo l'haurem d'emmagatzemar a la pila per a poder recuperar-ne el valor inicial al final de la subrutina, a continuació es carrega a RBP el valor de RSP.

RBP no s'ha de modificar dins la subrutina, al final d'aquesta se'n copia el valor sobre RSP, per a restaurar el valor inicial.

```
push rbp      ; Emmagatzemar el registre que farem servir d'apuntador a la pila rbp
mov rbp, rsp  ; Assignar a rbp el valor del registre apuntador rsp

...

mov rsp, rbp  ; Restaurem el valor inicial de rsp amb rbp
pop rbp       ; Restaurem el valor inicial de rbp
```

3) Després d'executar la subrutina: un cop fora de la subrutina cal alliberar l'espai utilitzat pels paràmetres d'entrada i després recuperar els resultats de l'espai que hem reservat abans de fer la crida.

Suposem que en l'exemple del factorial volem passar com a paràmetre el nombre del qual volem calcular el factorial i retornar com a resultat el factorial del nombre passat com a paràmetre, implementant el pas de paràmetres i el retorn de resultats per mitjà de la pila.

La crida de la subrutina:

```
sub rsp,8           ; Reservem 8 bytes per al resultat que retornem
push qword [x]      ; Introduïm com a paràmetre d'entrada a la pila
                    ; el valor de la variable 'x'
call factorial      ; Cridem a la subrutina factorial
                    ; A la pila tindrem el valor del factorial de 'x'
add rsp,8           ; Alliberem l'espai utilitzat pel paràmetre d'entrada
pop qword[result]   ; Recuperem el resultat retornat sobre la variable 'result'
```

Subrutina:

```
factorial:
  push rbp          ; Emmagatzemar el registre que farem servir d'apuntador a la pila rbp
  mov rbp, rsp      ; Assignar a rbp el valor del registre apuntador rsp
  push rax          ; Emmagatzemar a la pila els registres que
  push rbx          ; modifiquem dins la subrutina
                    ; Instruccions de la subrutina
  mov rax, 1        ; rax serà el resultat
  mov rbx, [rbp+16]; [rbp+16] referència al paràmetre d'entrada
while:
  cmp rbx, 1        ; Fem la comparació
  jle fi            ; Si es compleix la condició saltem a fi
  imul rax, rbx
  dec rbx
  jmp while         ; Salta a while
fi:                 ; A rax tindrem el valor del factorial de rbx
  mov [rbp+24], rax; [rbp+24] espai que hem reservat a la pila
                    ; per retornar el resultat
  pop rbx           ; Restaurem el valor inicial del registre
  pop rax
  mov rsp, rbp      ; Restaurem el valor inicial de rsp amb rbp
  pop rbp           ; Restaurem el valor inicial de rbp
  ret
```

Cal recordar que la memòria s'adreça byte a byte; cada adreça correspon a una posició de memòria d'un byte. En aquest exemple, com que els elements que posem a la pila són de 8 bytes (64 bits), per a passar d'una dada a la següent haurem de fer increments de 8 bytes.

Evolució de la pila en executar aquest codi. La taula mostra l'estat de la pila després d'executar cada instrucció o conjunt d'instruccions. L'adreça inicial de la pila és @ i considerem que la variable *x* val 5, i per tant el factorial de 5 és 120.

Adreça	**Estat inicial**		**sub rsp,8**		**push qword[x]**		**call factorial**	
	apuntadors	**pila**	**apuntadors**	**pila**	**apuntadors**	**pila**	**apuntadors**	**pila**
@ - 48								
@ - 40								
@ - 32								
@ - 24							rsp →	@retorn
@ - 16					rsp →	5		5
@ - 8			rsp →	----		----		----
@	rsp →	----		----		----		----

	push rbp mov rbp, rsp		**push rax push rbx**		**mov rbx, [rbp+16]**		**mov [rbp+24], rax**	
	apuntadors	**pila**	**apuntadors**	**pila**	**apuntadors**	**pila**	**apuntadors**	**pila**
@ - 48			rsp →	rbx	rsp →	rbx	rsp →	rbx
@ - 40				rax		rax		rax
@ - 32	rsp, rbp →	rbp	rbp →	rbp	rbp →	rbp	rbp →	rbp
@ - 24		@retorn		@retorn		@retorn		@retorn
@ - 16		5		5	rbp+16 →	**5**		**5**
@ - 8		----		----		----	rbp+24 →	120
@		----		----		----		----

	pop rbx pop rax mov rsp,rbp pop rbp		**ret**		**add rsp,8**		**pop qword[result]**	
	apuntadors	**pila**	**apuntadors**	**pila**	**apuntadors**	**pila**	**apuntadors**	**pila**
@ - 48								
@ - 40								
@ - 32								

	pop rbx pop rax mov rsp,rbp pop rbp		**ret**		**add rsp,8**		**pop qword[result]**	
	apuntadors	**pila**	**apuntadors**	**pila**	**apuntadors**	**pila**	**apuntadors**	**pila**
@ - 24	rsp →	@retorn						
@ - 16		**5**	rsp →	**5**		----		
@ - 8		120		120	rsp →	120		----
@		----		----		----	rsp →	----

Cal tenir present que el processador també utilitza la pila per a emmagatzemar els valors necessaris per a poder fer un retorn de la subrutina de manera correcta. En concret sempre que s'executa una instrucció *call*, el processador guarda al cim de la pila l'adreça de retorn (el valor actualitzat del comptador de programa RIP).

És important assegurar-se que en el moment de fer el *ret* l'adreça de retorn es trobi al cim de la pila; si no, es trencarà la seqüència normal d'execució.

Això s'aconsegueix si no es modifica el valor de RBP dins la subrutina i al final s'executen les instruccions:

```
mov rsp, rbp      ;Restaurem el valor inicial de RSP amb RBP
pop rbp           ;Restaurem el valor inicial de RBP
```

Una altra manera d'accedir als paràmetres d'entrada que tenim a la pila és treure els paràmetres de la pila i emmagatzemar-los en registres; en aquest cas caldrà primer treure l'adreça de retorn, a continuació treure els paràmetres i tornar a introduir l'adreça de retorn. Aquest manera d'accedir als paràmetres no se sol utilitzar perquè els registres que guarden els paràmetres no els podrem utilitzar per a altres propòsits i el nombre de paràmetres està condicionat pel nombre de registres disponibles. A més, la gestió dels valors de retorn complica molt la gestió de la pila.

```
pop rax   ;rax contindrà l'adreça de retorn
pop rdx   ;recuperem els paràmetres.
pop rcx
push rax   ;Tornem a introduir l'adreça de retorn
```

L'eliminació de l'espai utilitzat pels paràmetres d'entrada de la subrutina es fa fora de la subrutina incrementant el valor del registre RSP en tantes unitats com bytes ocupaven els paràmetres d'entrada.

```
add   rsp,8 ;alliberem l'espai utilitzat pel paràmetre d'entrada
```

Aquesta tasca també es pot fer dins la subrutina amb la instrucció *ret*, que permet especificar un valor que indica el nombre de bytes que ocupaven els paràmetres d'entrada, de manera que en retornar actualitza RSP incrementant-lo tants bytes com el valor del paràmetre del *ret*.

Posar la instrucció següent al final de la subrutina factorial seria equivalent a executar la instrucció `add rsp,8` després de la crida a subrutina:

```
ret 8
```

5.4.4. Variables locals en assemblador

En els llenguatges d'alt nivell és habitual definir variables locals dins les funcions definides en un programa. Aquestes variables locals ocupen un espai definit dins la pila.

En assemblador no és habitual definir variables locals, en lloc seu s'utilitzen registres, i en cas de necessitar més espai s'utilitza puntualment la pila.

Exemple

```
long int factorial (long int x) {
  long int j, result;

  result=1;
  for (j = x; j > 1; j--){
    result=result*j;
  }

  return result;
}
```

```
factorial:
  push rbp
  mov rbp, rsp
  push rbx

  mov rax, 1
  mov rbx, rdi
for:
  cmp rbx, 1
  jle fi
  imul rax, rbx
  dec rbx
  jmp for
fi:
  pop rbx
  mov rsp, rbp
  pop rbp
  ret
```

El paràmetre d'entrada *x* del codi C, es rep sobre el registre *rdi* en el codi assemblador (rdi és el registre que s'utilitza com a primer paràmetre d'entrada en les crides en C) i actua com a variable local dins la funció de C.

La variable local *j* del codi C s'implementa utilitzant el registre *rbx* en assemblador.

El paràmetre de sortida del codi C, la variable *result*, que també està declarada com a variable local, s'implementa amb el registre *rax*, que és el registre que s'utilitza per a retornar el resultat en les crides en C.

Si es volen implementar variables locals, tal com es gestionen en C, s'hauria de fer el següent.

Per a reservar l'espai necessari cal saber quants bytes utilitzarem com a variables locals. A continuació cal decrementar el valor de l'apuntador a pila RSP tantes unitats com bytes es vulguin reservar per a les variables locals; d'aquesta

manera si utilitzem les instruccions que treballen amb la pila dins la subrutina (push i pop) no sobreescriurem l'espai de les variables locals. L'actualització de RSP es fa just després d'actualitzar el registre que utilitzem per a accedir a la pila, RBP.

```
subrutina:
   push rbp              ; Emmagatzemar el registre que farem servir
                         ; d'apuntador a la pila rbp
   mov rbp, rsp          ; Assignar a RBP el valor del registre apuntador RSP
   sub rsp, n            ; n indica el nombre de bytes reservats per a les
                         ; variables locals
                         ;
                         ; Instruccions de la subrutina
                         ;
   mov rsp, rbp          ; Restaurem el valor inicial de RSP amb RBP
   pop rbp               ; Restaurem el valor inicial de RBP
   ret
```

Com que les variables locals són a la pila, s'utilitza també el registre apuntador, RBP per a accedir a les variables. S'utilitza un adreçament relatiu sobre el registre RBP, per a accedir a l'espai reservat, restant un valor al registre RBP.

```
push rbp                 ; Emmagatzemar el registre que farem servir
                         ; d'apuntador a la pila rbp
mov rbp, rsp             ; Assignar a RBP el valor del registre apuntador RSP
sub rsp, 8               ; reservem 8 bytes per a variables locals

mov al, byte[RBP-1]      ;accedim a 1 byte d'emmagatzematge local
mov ax, word[RBP-2]      ;accedim a 2 bytes d'emmagatzematge local
mov eax, dword[RBP-4]    ;accedim a 4 bytes d'emmagatzematge local
mov rax, qword[RBP-8]    ;accedim a 8 bytes d'emmagatzematge local
```

5.4.5. Crides a subrutines i pas de paràmetres des de C

En aquest subapartat s'explica com cridar subrutines escrites en llenguatge d'assemblador des d'un programa escrit en C i com passar-li paràmetres.

Per a poder utilitzar una subrutina escrita en llenguatge d'assemblador dins un programa escrit en C, hi ha una sèrie de qüestions que cal tenir presents, i que afecten tant el codi en llenguatge C com el codi en llenguatge d'assemblador.

Definició i implementació de funcions en assemblador

Qualsevol subrutina en assemblador que es vulgui cridar des d'un altre programa s'haurà de declarar utilitzant la directiva *global*.

Es pot accedir a qualsevol variable global de C des d'assemblador declarant-la en el codi assemblador amb la directiva `extern`.

L'estructura general del codi assemblador seria el següent:

```
section .text
                  ;Declararem amb global el nom de les subrutines
                  ;a les que volem accedir des d'altres fitxers de codi font
   global subrutina1, ..., subrutina

                  ;Declarem amb extern les variables globals de C que
                  ;volem accedir des d'assemblador
   extern variable1, ..., variableN

subrutina1:
   push rbp       ;Emmagatzemar el registre rbp a la pila
   mov rbp,rsp    ;Assignar el valor del registre apuntador RSP
   sub rsp, n     ;Reservem n bytes per a variables locals
   ;
   ; codi de la subrutina
   ;
   mov rsp, rbp   ;Restaurem el valor inicial de RSP amb RBP
   pop rbp        ;Restaurem el valor inicial de RBP
   ret

   ...

subrutinaN:
   push rbp       ;Emmagatzemar el registre rbp a la pila
   mov rbp,rsp    ;Assignar el valor del registre apuntador RSP
   sub rsp, n     ;Reservem n bytes per a variables locals
   ;
   ; codi de la subrutina
   ;
   mov rsp, rbp   ;Restaurem el valor inicial de RSP amb RBP
   pop rbp        ;Restaurem el valor inicial de RBP

   ret
```

Exemple de crida a una subrutina d'assemblador des de C i accés a variables globals de C des d'assemblador

Codi assemblador:

```
.text

global factorial       ; Declarem amb global la subrutina d'assemblador
                       ; que volem fer visible al codi C

extern x, result       ; Declarem amb extern les variables globals de C
                       ; a les que volem accedir des d'assemblador

factorial:
   push rbp
   mov rbp, rsp

   push rax            ; Emmagatzemem a la pila els registres
   push rbx            ; modificats dins la subrutina

   mov rax, 1          ; rax serà el resultat
   mov rbx, [x]        ; Calculem el factorial de la variable 'x' (=5)
while:
   cmp rbx, 1          ; Fem la comparació
   jle fi              ; Si es compleix la condició saltem a fi
   imul rax, rbx       ; A rax tindrem el valor del factorial de 'x' (=120)

   dec rbx
   jmp while           ; Salta a while
fi:
   mov [result], rax   ; Emmagatzemem el resultat en la variable result

   pop rbx             ; Restaurem el valor inicial dels registres
   pop rax             ; en ordre invers a com els hem emmagatzemat

   mov rsp, rbp

   pop rbp

   ret
```

Codi C:

```
long int x, result;    //Variables globals a les quals s'accedeix des d'assemblador

int main() {
   x=5;
   factorial();        //Crida a la subrutina d'assemblador

   return 0;
}
```

Pas de paràmetres i retorn de resultat

En el mode de 64 bits, quan es crida des de C, una funció en llenguatge C o una subrutina en assemblador, els sis primers paràmetres es passen per registre utilitzant els registres següents en l'ordre especificat:

RDI, RSI, RDX, RCX, R8 i R9.

La resta de paràmetres es passen per mitjà de la pila.

El valor de retorn es passa sempre per registre i s'utilitza sempre el registre RAX.

Exemple

Volem definir i implementar algunes subrutines en assemblador: *suma*, *producte*, *factorial*. La subrutina *suma* rebrà dos paràmetres, farà la suma dels dos paràmetres i en retornarà la suma; la subrutina *producte* també rebrà dos paràmetres, farà el producte dels dos paràmetres i retornarà el resultat; finalment la subrutina *factorial* farà el factorial d'un nombre rebut com a paràmetre (correspon a la mateixa funció factorial dels subapartats anteriors).

El codi assemblador seria el següent:

```
;Fitxer funcions.asm
section .text
global suma, producte, factorial

suma:
  push rbp
  mov rbp, rsp
                ;2 paràmetres d'entrada: rdi, rsi
  mov rax, rdi
  add rax, rsi

  mov rsp, rbp
  pop rbp

  ret           ;retorn de resultat per mitjà de rax, rax=rdi+rsi


producte:
  push rbp
  mov rbp, rsp
                ;2 paràmetres d'entrada: rdi, rsi
  mov rax, rdi
  imul rax, rsi   ;rax=rax*rsi=rdi*rsi

  mov rsp, rbp
  pop rbp

  ret           ;retorn de resultat per mitjà de rax


factorial:
  push rbp
  mov rbp, rsp
                ;1 paràmetre d'entrada: rdi
                ;no hi ha variables locals
  push rdi      ;rdi és modificat per la subrutina
  mov rax, 1    ;rax serà el resultat
while:
  cmp rdi, 1    ;Fem la comparació
  jle fi        ;Si es compleix la condició saltem a fi
  imul rax, rdi
  dec rdi
  jmp while     ;Salta a while
fi:             ;A rax tindrem el valor del factorial de rbx
  pop rdi       ;restaurem el valor de rdi

  mov rsp, rbp
  pop rbp

  ret
```

Vegem com seria el codi C que crida les subrutines *suma*, *producte* i *factorial*:

```
//Fitxer principal.c
#include <stdio.h>


int main()
```

```
{
   int x, y, result;

   printf("\nIntrodueix el valor de x: ");
   scanf("%d", &x);
   printf("Introdueix el valor de y: ");
   scanf("%d", &y);

   result = suma(x, y); // Cridem la subrutina suma passant les variables
                        // x i y com a paràmetres i retornant
                        // el resultat sobre la variable result
   printf ("La suma de x i y és %d\n", result);

   result = producte(x, y); // Cridem la subrutina producte passant les
                            // variables x i y com a paràmetres i retornant
                            // el resultat sobre la variable result

   printf ("El producte de x i y és %d\n", result);

   result = factorial(x); // Cridem la subrutina factorial passant la
                          // variable x com a paràmetre i retornant
                          // el resultat sobre la variable result

   printf ("El factorial de x és %d\n", result);
   return 0;
}
```

Per a executar aquest codi cal fer l'assemblatge del codi font assemblador, compilar el codi font C amb el codi objecte assemblador i generar l'executable.

Paràmetres per referència

Es poden passar paràmetres per valor o per referència.

En un pas per valor s'està passant a la subrutina un valor per a ser utilitzat dins la subrutina, es passa el contingut d'una variable. En sortir de la subrutina el valor de la variable no s'haurà modificat.

L'altra opció es passar paràmetres a una subrutina per referència, passant a la subrutina l'adreça d'una variable, de manera que, si dins la subrutina es modifica aquest paràmetre, en sortir de la subrutina aquest mantingui el valor obtingut dins la subrutina.

Exemple

Definim dues subrutines inivec i getvec.

inivec rep l'adreça d'un vector (paràmetre per referència) i un índex del vector (paràmetre per valor). Dins la subrutina s'inicialitza l'element del vector a zero i, en sortir de la subrutina, aquest element quedarà assignat a zero.

getvec rep l'adreça d'un vector (paràmetre per referència), un índex (paràmetre per valor) i l'adreça d'una variable (paràmetre per referència). Dins la subrutina la variable s'actualitza amb el valor de l'element indicat per l'índex i, en sortir de la subrutina, la variable quedarà assignada amb el valor obtingut del vector.

Codi assemblador

```
section .text
global inivec, getvec

inivec:
   push rbp
   mov rbp, rsp
                         ; rdi: primer paràmetre, adreça del vector
                         ; rsi: segon paràmetre, índex del vector
   mov byte [rdi+rsi], 0 ; es posa a zero l'element del vector

   mov rsp, rbp
   pop rbp

ret

getvec:
   push rbp
   mov rbp, rsp
                          ; rdi: primer paràmetre, adreça del vector
                          ; rsi: segon paràmetre, índex del vector
                          ; rdx: tercer paràmetre, variable passada per
                          ;      referència
   mov byte al, [rdi+rsi] ; s'assigna al registre 'al' l'element del vector
   mov byte [rdx], al     ; es passa l'element del vector a la variable rebuda
                          ; com a paràmetre
   mov rsp, rbp
   pop rbp

   ret
```

Codi C

```
int main(){
   int j;
   char v[5],x;

   for(j=0;j<5;j++){
      inivec(v,j);    // v es passa per referència, es modifica dins la
                      // subrutina. El nom d'una variable de tipus vector
                      // representa l'adreça del primer element del vector.
                      // j és l'índex del vector al qual volem accedir
                      // i es passa per valor
   }
   j=0;
   getvec(v, j, &x)   // x es passa per referència, es modifica dins la
                      // subrutina amb el valor de l'element que ocupa la
                      // posició j dins del vector v. v es passa per
                      // referència i j es passa per valor
}
```

5.5. Entrada/sortida

El sistema d'E/S en els processadors de la família x86-64 utilitza un mapa independent d'E/S.

Es disposa d'un espai separat d'adreçament de 64 K, accessible com a ports de 8, 16 i 32 bits; depenent del dispositiu es podrà accedir en unitats de 8, 16 o 32 bits.

El joc d'instruccions disposa d'instruccions específiques, IN i OUT, per a llegir d'un port d'E/S i escriure en un port d'E/S.

5.5.1. E/S programada

Es poden fer tasques d'E/S programada utilitzant les instruccions IN i OUT d'assemblador, o bé les instruccions equivalents de llenguatge C: *inb*, *outb*.

En els sistemes Linux cal permetre l'accés als ports d'E/S utilitzant la crida al sistema *io_perm*, abans de fer un accés a un port d'E/S.

Exemple

A continuació es mostra un programa escrit en C que accedeix al ports d'E/S 70h i 71h, corresponents als registres d'adreces i de dades del rellotge de temps real del sistema (RTC).

L'adreça 70h correspon al registre d'adreces, en què s'escriu l'adreça de la dada de l'RTC que es vol llegir:

00: segons
02: minuts
04: hora
08: mes de l'any

```
//Fitxer: ioprog.c

#include <stdio.h>
#include <stdlib.h>
#include <sys/io.h>

#define RTC_ADDR 0x70 // registre d'adreces de l'RTC
#define RTC_DATA 0x71 // registre de dades de l'RTC

int main()
{
   char month, hh, mm;

   // Es defineix cada mes de l'any de 9 caràcters
   // 'setembre', 'novembre' i 'desembre' són els més llargs
   // i ocupen 9 caràcters inclòs un \0 al final
   char mesos[12][9]={"gener", "febrer", "març", "abril", "maig", "juny",
       "juliol", "agost", "setembre", "octubre", "novembre", "desembre"};

   /* Es proporciona permís per a accedir als ports d'E/S
    * RTC_ADDR adreça inicial a partir de la qual es vol accedir
    * 2: es demana accés a 2 bytes
    * 1: s'activa el permís d'accés */
   if (ioperm(RTC_ADDR, 2, 1)) {
      perror("ioperm");
      exit(1);
   }

   outb(8, RTC_ADDR);   // S'escriu al registre d'adreces l'adreça 8: mes
   month=inb(RTC_DATA); // Es llegeix del port de dades el mes de l'any

   // El valor obtingut és el número de mes entre 1 i 12, però l'índex
   // del vector mesos està entre de 0 a 11 i cal restar 1
   printf("Mes any RTC: %d %s\n", month, mesos[month-1]);

   outb(4, RTC_ADDR);  // S'escriu al registre d'adreces l'adreça 4: hora
   hh=inb(RTC_DATA);   // Es llegeix del port de dades l'hora
```

```
    printf("Hora RTC: %0x:", hh);

    outb(2, RTC_ADDR);  // S'escriu al registre d'adreces l'adreça 2: minuts
    mm=inb(RTC_DATA);   // Es llegeixen del port de dades els minuts

    printf("%0x\n", mm);

    // Es desactiven els permisos per a accedir als ports posant un 0
    if (ioperm(RTC_ADDR, 2, 0)) {
       perror("ioperm");
       exit(1);
    }

    return 0;
}
```

Per a poder executar un programa que accedeixi a ports d'E/S en Linux, cal disposar de permisos de superusuari.

Exemple

A continuació es mostra un altre exemple d'accés a ports d'E/S: es tracta de llegir el registre de dades del teclat, port 60h.

El codi següent fa un bucle que mostra el codi de la tecla polsada (*scancode*) fins que es prem la tecla ESC.

El registre 60h emmagatzema en els 7 bits de menys pes el codi de cada tecla que es prem.

```
#include <stdio.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/io.h>

#define KBD_DATA 0x60

int main(){
   int sortir=0;
   char data;

   if (ioperm(KBD_DATA, 1, 1)) {
      perror("ioperm");
      exit(1);
   }
   while (sortir==0){
      data=inb(KBD_DATA);
      // es mostra el codi de la tecla premuda, scancode
      // es fa una AND lògica ('&') per a agafar només els
      // 7 bits menys significatius de la dada llegida
      printf("tecla: %0x\n", data & 0b01111111);
      // es prem ESC per sortir del bucle, scancode=1
      if ((data & 0b01111111) == 1) sortir=1;
   }
   if (ioperm(KBD_DATA, 1, 0)) {
      perror("ioperm");
      exit(1);
   }
   return 0;
}
```

5.6. Controlar la consola

En Linux no hi ha funcions estàndard o serveis del sistema operatiu per a controlar la consola, per exemple, per a moure el cursor. Disposem, però, de la possibilitat d'escriure seqüències d'*escape* que ens permeten manipular l'emulador de terminal de Linux, entre d'altres, moure el cursor, netejar la consola, etc.

> Una seqüència d'*escape* és una cadena de caràcters que comença amb el caràcter ESC (corresponent al codi ASCII 27 decimal o 1Bh).

Les principals seqüències d'*escape* són les següents:

1) Moure el cursor. Per a moure el cursor cal considerar que la posició inicial en un terminal Linux correspon a la fila 0 columna 0; per moure el cursor escriurem la cadena de caràcters següent:

```
ESC[F;CH
```

on *ESC* correspon al codi ASCII del caràcter d'*escape* 27 decimal o 1Bh, *F* correspon a la fila on volem col·locar el cursor, expressada com un valor decimal i *C* correspon a la columna on volem col·locar el cursor, expressada com un valor decimal.

Exemple

Moure el cursor a la fila 5 columna 10, escrivint la seqüència `ESC[05;10H`.

```
section .data
   escSeq db 27,"[05;10H"
   escLen equ 8
section .text
   mov rax,4
   mov rbx,1
   mov rcx, escSeq
   mov rdx, escLen
   int 80h
```

2) Netejar la consola. Per a netejar la consola cal escriure la seqüència `ESC[2J`.

```
section .data
   escSeq    db 27,"[2J"   ;ESC[2J
   escLen   equ 4          ; mida de la cadena escSeq

section .text
   mov rax,4
   mov rbx,1
   mov rcx, escSeq
   mov rdx, escLen
   int 80h
```

3) Seqüències d'*escape* en C. En llenguatge C podem aconseguir el mateix escrivint la seqüència utilitzant la funció *printf*; el caràcter ESC es pot escriure amb el seu valor hexadecimal 1B.

Exemple

```
#include <stdio.h>

int main(){
       printf("\x1B[2J");      //Esborra la pantalla
     printf("\x1B[5;10H");   //Situa el cursor a la posició (5,10)
  }
```

5.7. Funcions del sistema operatiu (*system calls*)

Des d'un programa en assemblador podem fer crides a diferents funcions del nucli (*kernel*) del sistema operatiu; és el que es coneix com a *system calls* o *kernel system calls*.

El llenguatge d'assemblador proporciona dos mecanismes per a poder fer crides al sistema operatiu:

1) ***int 80h***: aquest és el mecanisme tradicional en processadors x86 i per tant també està disponible en els processadors amb arquitectura x86-64.

El servei que se sol·licita s'especifica mitjançant el registre RAX. Els paràmetres necessaris per a l'execució del servei s'especifiquen per mitjà dels registres RBX, RCX, RDX, RSI, RDI i RBP.

2) ***syscall***: els processadors de l'arquitectura x86-64 proporcionen un mecanisme més eficient de fer crides al sistema, la instrucció *syscall*.

El servei sol·licitat també s'especifica per mitjà de RAX, però els nombres que identifiquen cada servei són diferents dels utilitzats amb la instrucció *int* 80h. Els paràmetres s'especifiquen per mitjà dels registres RDI, RSI, RDX, RCX, R8 i R9.

El sistema operatiu proporciona al programador moltes funcions de diferents tipus; veurem només les funcions següents:

- Lectura del teclat
- Escriptura per pantalla
- Retorn al sistema operatiu

5.7.1. Lectura d'una cadena de caràcters des del teclat

Llegeix caràcters del teclat fins que es prem la tecla ENTER. La lectura de caràcters es fa cridant la funció de lectura *read*. Per a utilitzar aquesta funció cal especificar el descriptor d'arxiu que s'utilitzarà; en el cas d'una lectura de teclat s'utilitza el descriptor corresponent a l'entrada estàndard, un 0 en aquest cas.

Segons si s'utilitza *int 80h* o *syscall* els paràmetres són els següents:

1) ***int 80h***

a) Paràmetres d'entrada

- RAX = 3
- RBX = 0, descriptor corresponent a l'entrada estàndard (teclat)
- RCX = adreça de la variable de memòria on es guardarà la cadena llegida
- RDX = nombre màxim de caràcters que es llegiran

b) Paràmetres de sortida

- RAX = nombre de caràcters llegits
- La variable indicada s'emplena amb els caràcters llegits.

2) ***syscall***

a) Paràmetres d'entrada

- RAX = 0
- RDI = 0, descriptor corresponent a l'entrada estàndard (teclat)
- RSI = adreça de la variable de memòria on es guardarà la cadena llegida
- RDX = nombre màxim de caràcters que es llegiran

b) Paràmetres de sortida

- RAX = nombre de caràcters llegits

Exemple

```
section .bss
  buffer resb 10 ;es reserven 10 bytes per a fer la lectura del teclat

section .text
   ;lectura utilitzant int 80h
   mov rax, 3
   mov rbx, 0
     mov rcx, buffer   ; es carrega l'adreça de buffer a rcx
     mov rdx,10        ; nombre màxim de caràcters que es llegiran
     int 80h           ; es crida el kernel

                       ;lectura utilitzant syscall
   mov rax, 0
   mov rdi, 0
     mov rsi, buffer   ; es carrega l'adreça de buffer a rsi
     mov rdx,10        ; nombre màxim de caràcters que es llegiran
     syscall           ; es crida el kernel
```

5.7.2. Escriptura d'una cadena de caràcters per pantalla

L'escriptura de caràcters per pantalla es fa cridant la funció d'escriptura *write*. Per a utilitzar aquesta funció cal especificar el descriptor d'arxiu que s'utilitzarà; en el cas d'una escriptura per pantalla s'utilitza el descriptor corresponent a la sortida estàndard, un 1 en aquest cas.

1) ***int 80h***

a) Paràmetres d'entrada

- RAX = 4
- RBX = 1, descriptor corresponent a la sortida estàndard (pantalla)
- RCX = adreça de la variable de memòria que volem escriure, la variable ha d'estar definida amb un byte 0 al final
- RDX = mida de la cadena que volem escriure en bytes, inclòs el 0 del final

b) Paràmetres de sortida

- RAX = nombre de caràcters escrits

2) ***syscall***

a) Paràmetres d'entrada

- RAX = 1
- RDI = 1, descriptor corresponent a la sortida estàndard (pantalla)
- RSI = adreça de la variable de memòria que volem escriure, la variable ha d'estar definida amb un byte 0 al final
- RDX = mida de la cadena que volem escriure en bytes, inclòs el 0 del final

b) Paràmetres de sortida

- RAX = nombre de caràcters escrits

Exemple

```
section .data
   msg db "Hola!",0
   msgLen db 6

section .text
                    ; escriptura utilitzant int 80h
   mov rax,4
   mov rbx,1
   mov rcx, msg      ; es posa l'adreça de msg1 a rcx
   mov rdx, msgLen   ; mida de msg
   int 80h           ; es crida el kernel

;escriptura utilitzant syscall
   mov rax, 1
   mov rdi, 1
   mov rsi, msg      ; es carrega l'adreça de msg a rsi
   mov rdx, msglen   ; mida de msg
   syscall           ; es crida el kernel
```

Es pot calcular la mida d'una variable fent la diferència entre una posició dins la secció data i la posició on es troba declarada la variable:

```
section .data
   msg db "Hola!",0
   msgLen db equ $ - msg   ; $ indica l'adreça de la posició actual
```

$ defineix la posició del principi de la línia actual; en restar-li la etiqueta *msg*, se li resta la posició on es troba declarada l'etiqueta i, per tant, s'obté el nombre de bytes reservats a la posició de l'etiqueta *msg*, 6 en aquest cas.

5.7.3. Retorn al sistema operatiu (*exit*)

Finalitza l'execució del programa i retorna el control al sistema operatiu.

1) ***int 80h***

a) Paràmetres d'entrada

- RAX = 1
- RBX = valor de retorn del programa

2) ***syscall***

a) Paràmetres d'entrada

- RAX = 60
- RDI = valor de retorn del programa

Exemple

```
            ;retorna al sistema operatiu utilitzant int 80h
mov rax,1
mov rbx,0   ;valor de retorn 0
int 80h     ;es crida el kernel

            ;retorna al sistema operatiu utilitzant syscall
mov rax,60
mov rbx,0   ;valor de retorn 0
syscall     ;es crida el kernel
```

6. Annex: manual bàsic del joc d'instruccions

En aquest annex es descriuen detalladament les instruccions més habituals del llenguatge d'assemblador de l'arquitectura x86-64, però cal tenir present que l'objectiu d'aquest apartat no és ser un manual de referència complet d'aquesta arquitectura i, per tant, no es descriuen totes les instruccions del joc d'instruccions.

Descripció de la notació utilitzada en les instruccions:

1) Bits de resultat (*flags*):

- OF: Overflow flag (bit de sobreeiximent)
- TF: Trap flag (bit d'excepció)
- AF: Aux carry (bit de transport auxiliar)
- DF: Direction flag (bit de direcció)
- SF: Sign flag (bit de signe)
- PF: Parity flag (bit de paritat)
- IF: Interrupt flag (bit d'interrupció)
- ZF: Zero flag (bit de zero)
- CF: Carry flag (bit de transport)

2) Tipus d'operands:

a) `imm`: valor immediat; pot ser un valor immediat de 8 bits, 16 bits o 32 bits. Segons la mida del valor immediat es podran representar els intervals de valors següents:

- Immediat de 8 bits: sense signe [0, 255], amb signe en Ca2 [–128, +127]
- Immediat de 16 bits: sense signe [0, 65.535], amb signe en Ca2 [–32.768, +32.767]
- Immediat de 32 bits: sense signe [0, 4.294.967.295], amb signe en Ca2 [–2.147.483.648, +2.147.483.647]

Els valors immediats de 64 bits només es permeten per a carregar un registre de propòsit general de 64 bits, mitjançant la instrucció MOV. L'interval de representació és el següent:

- sense signe [0, 18.446.744.073.709.551.615],
- amb signe en Ca2 [–9.223.372.036.854.775.808, + 9.223.372.036.854.775.807]

b) `reg`: registre, pot ser un registre de 8 bits, 16 bits, 32 bits o 64 bits.
`reg8`: registre, ha de ser un registre de 8 bits.

`reg16`: registre, ha de ser un registre de 16 bits.
`reg32`: registre, ha de ser un registre de 32 bits.
`reg64`: registre, ha de ser un registre de 64 bits.

c) `mida mem`: posició de memòria, s'indica primer si s'accedeix a 8, 16, 32 o 64 bits i, a continuació, l'adreça de memòria.

Encara que especificar la mida no és estrictament necessari, sempre que s'utilitza un operand de memòria, es farà d'aquesta manera per claredat a l'hora de saber a quants bytes de memòria s'estan accedint.

Els especificadors de mida vàlids són:

- BYTE: posició de memòria de 8 bits
- WORD: posició de memòria de 16 bits
- DWORD: posició de memòria de 32 bits
- QWORD: posició de memòria de 64 bits

6.1. ADC: suma aritmètica amb bit de transport

ADC destinació, font

Fa una suma aritmètica, suma l'operand font i el valor del bit de transport (CF) a l'operand de destinació, emmagatzema el resultat sobre l'operand destinació, i substitueix el valor inicial de l'operand destinació.

Operació

destinació = destinació + font + CF

Bits de resultat modificats

OF, SF, ZF, AF, PF, CF

Si el resultat no cap dins l'operand destinació, el bit de transport es posa a 1. La resta de bits de resultat es modifiquen segons el resultat de l'operació.

Formats vàlids

```
ADC reg, reg
ADC reg, mida mem
ADC mida mem, reg
```

Els dos operands han de ser de la mateixa mida.

```
ADC reg, imm
ADC mida mem, imm
```

La mida de l'immediat pot anar des de 8 bits fins a la mida del primer operand, 32 bits com a màxim.

Exemples

```
ADC R9,RAX
ADC RAX, QWORD [variable]
ADC DWORD [variable],EAX
ADC RAX,0x01020304
ADC BYTE [vector+RAX], 5
```

6.2. ADD: suma aritmètica

ADD destinació, font

Fa la suma aritmètica dels dos operands de la instrucció, emmagatzema el resultat sobre l'operand destinació, i substitueix el valor inicial de l'operand destinació.

Operació

destinació = destinació + font

Bits de resultat modificats

OF, SF, ZF, AF, PF, CF

Si el resultat no cap dins l'operand destinació, el bit de transport es posa a 1. La resta de bits de resultat es modifiquen segons el resultat de l'operació.

Formats vàlids

```
ADD reg, reg
ADD reg, mida mem
ADD mida mem, reg
```

Els dos operands han de ser de la mateixa mida.

```
ADD reg, imm
ADD mida mem, imm
```

La mida de l'immediat pot anar des de 8 bits fins a la mida del primer operand, 32 bits com a màxim.

Exemples

```
ADD R9,RAX
ADD RAX, QWORD [variable]
ADD DWORD [variable],EAX
ADD RAX,0x01020304
ADD BYTE [vector+RAX], 5
```

6.3. AND: I lògica

AND destinació, font

Fa una operació lògica AND ('i lògica') bit a bit entre l'operand destinació i l'operand font, el resultat de l'operació es guarda sobre l'operand destinació sobreescrivint el valor inicial. El valor de l'operand font no es modifica.

Es fa una operació AND entre el bit *n* de l'operand destinació i el bit *n* de l'operand font segons la taula de veritat de la funció AND:

```
 x   y   x AND y
--- ---  -------
 0   0      0
 0   1      0
 1   0      0
 1   1      1
```

Operació

destinació = destinació AND font

Bits de resultat modificats

OF=0, SF, ZF, PF, CF=0

Els bits de resultat OF i CF es posen a 0, la resta es modifiquen segons el resultat de l'operació.

Formats vàlids

```
AND reg,reg
AND reg,mida mem
AND mida mem,reg
```

Els dos operands han de ser de la mateixa mida.

```
AND reg,imm
AND mida mem,imm
```

La mida de l'immediat pot anar des de 8 bits fins a la mida del primer operand, com a màxim 32 bits.

Exemples

```
AND R9,RAX
AND RAX, QWORD [variable]
AND DWORD [variable],EAX
AND RAX,0x01020304
AND BYTE [vector+RAX], 5
```

6.4. CALL: crida a subrutina

CALL etiqueta

Crida la subrutina que es troba a l'adreça de memòria indicada per l'etiqueta. Guarda a la pila l'adreça de memòria de la instrucció que segueix en seqüència la instrucció CALL, i permet el retorn des de la subrutina amb la instrucció RET; a continuació carrega en el RIP (*instruction pointer*) l'adreça de memòria on hi ha l'etiqueta especificada en la instrucció, i transfereix el control a la subrutina.

En entorns de 64 bits, l'adreça de retorn serà de 64 bits, per tant, s'introdueix a la pila un valor de 8 bytes. Per a fer-ho primer s'actualitza el punter de pila (registre RSP) decrementant-se en 8 unitats, i a continuació es copia l'adreça al cim de la pila.

Operació

RSP=RSP-8
M[RSP] ← RIP
RIP ← adreça_etiqueta

Bits de resultat modificats

Cap

Formats vàlids

```
CALL etiqueta
```

Hi ha altres formats, però queden fora dels objectius d'aquests materials. Podeu consultar les fonts bibliogràfiques.

Exemples

```
CALL subrutina1
```

6.5. CMP: comparació aritmètica

CMP destinació, font

Compara els dos operands de la instrucció sense afectar el valor de cap dels operands, actualitza els bits de resultat segons el resultat de la comparació. La comparació es fa amb una resta entre els dos operands, sense considerar el transport i sense guardar-ne el resultat.

Operació

destinació - font

Bits de resultat modificats

OF, SF, ZF, AF, PF, CF

Els bits de resultat es modifiquen segons el resultat de l'operació de resta.

Formats vàlids

```
CMP reg,reg
CMP reg,mida mem
CMP mida mem,reg
```

Els dos operands han de ser de la mateixa mida.

```
CMP reg,imm
CMP mida mem,imm
```

La mida de l'immediat pot anar des de 8 bits fins a la mida del primer operand, com a màxim 32 bits.

Exemples

```
CMP R9,RAX
CMP RAX, QWORD [variable]
CMP DWORD [variable],EAX
CMP RAX,0x01020304
CMP BYTE [vector+RAX], 5
```

6.6. DEC: decrementa l'operand

DEC destinació

Resta 1 a l'operand de la instrucció, i emmagatzema el resultat en el mateix operand.

Operació

destinació = destinació – 1

Bits de resultat modificats

OF, SF, ZF, AF, PF

Els bits de resultat es modifiquen segons el resultat de l'operació.

Formats vàlids

```
DEC reg
DEC mida mem
```

Exemples

```
DEC EAX
DEC R9
DEC DWORD [R8]
DEC QWORD [variable]
```

6.7. DIV: divisió entera sense signe

DIV font

Divideix el dividend implícit entre el divisor explícit sense considerar els signes dels operands.

Si el divisor és de 8 bits es considera com a dividend implícit AX. El quocient de la divisió queda a AL, i la resta, a AH.

Si el divisor és de 16 bits es considera com a dividend implícit el parell de registres DX: AX, la part menys significativa del dividend es col·loca a AX, i la part més significativa, a DX. El quocient de la divisió queda a AX, i la resta, a DX.

Si el divisor és de 32 bits, el funcionament és semblant al cas anterior, però utilitzant el parell de registres EDX:EAX; la part menys significativa del dividend es col·loca a EAX i la part més significativa a EDX. El quocient de la divisió queda a EAX i la resta a EDX.

Si el divisor és de 64 bits, el funcionament és semblant als dos casos anteriors però utilitzant el parell de registres RDX:RAX, la part menys significativa del dividend es col·loca a RAX, i la part més significativa, a RDX. El quocient de la divisió queda a RAX, i la resta, a RDX.

Operació

Si *font* és de 8 bits: AL = AX / *font*, AH = AX mod *font*

Si *font* és de 16 bits: AX = DX:AX / *font*, DX = DX:AX mod *font*

Si *font* és de 32 bits: EAX = EDX:EAX / *font*, EDX = EDX:EAX mod *font*

Si *font* és de 64 bits: RAX = RDX:RAX / *font*, RDX = RDX:RAX mod *font*

Bits de resultat modificats

OF, SF, ZF, AF, PF, CF

La instrucció DIV no deixa informació als bits de resultat, però aquests queden indefinits.

Formats vàlids

```
DIV reg
DIV mida mem
```

Exemples

```
DIV R8B     ; AX / R8B => Quocient a AL; resta a AH
DIV R8W     ; DX:AX / R8W => Quocient a AX; resta a DX
DIV ECX     ; EDX:EAX / ECX => Quocient a EAX; resta a EDX
DIV QWORD [R9] ; RDX:RAX / QWORD [R9] => Quocient a RAX, resta a RDX
```

6.8. IDIV: divisió entera amb signe

IDIV font

Divideix el dividend implícit entre el divisor explícit (font) considerant el signe dels operands. El funcionament és idèntic al de la divisió sense signe.

Si el divisor és de 8 bits es considera com a dividend implícit AX.

El quocient de la divisió queda a AL i la resta a AH.

Si el divisor és de 16 bits es considera com a dividend implícit el parell de registres DX: AX; la part menys significativa del dividend es col·loca a AX, i la part més significativa, a DX. El quocient de la divisió queda a AX, i la resta, a DX.

Si el divisor és de 32 bits el funcionament és semblant al cas anterior, però utilitzant el parell de registres EDX:EAX; la part menys significativa del dividend es col·loca a EAX, i la part més significativa, a EDX. El quocient de la divisió queda a EAX, i la resta, a EDX.

Si el divisor és de 64 bits, el funcionament és semblant als dos casos anteriors, però utilitzant el parell de registres RDX:RAX; la part menys significativa del dividend es col·loca a RAX, i la part més significativa, a RDX. El quocient de la divisió queda a RAX, i la resta, a RDX.

Operació

Si *font* és de 8 bits: AL = AX / font, AH = AX mod *font*

Si *font* és de 16 bits: AX = DX:AX / *font*, DX = DX:AX mod *font*

Si *font* és de 32 bits: EAX = EDX:EAX / *font*, EDX = EDX:EAX mod *font*

Si *font* és de 64 bits: RAX = RDX:RAX / *font*, RDX = RDX:EAX mod *font*

Bits de resultat modificats

OF, SF, ZF, AF, PF, CF

La instrucció IDIV no deixa informació als bits de resultat, però aquests queden indefinits.

Formats vàlids

```
IDIV reg
IDIV mida mem
```

Exemples

```
IDIV CH    ; AX / CH => Quocient a AL; resta a AH
IDIV BX    ; DX:AX / BX => Quocient a AX; resta a DX
IDIV ECX ; EDX:EAX / ECX => Quocient a EAX; resta a EDX
IDIV QWORD [R9] ; RDX:RAX / [R9] => Quocient a RAX, resta a RDX
```

6.9. IMUL: multiplicació entera amb signe

IMUL font

IMUL destinació, font

L'operació de multiplicació amb signe pot utilitzar diferent nombre d'operands; es descriurà el format de la instrucció amb un operand i amb dos operands.

6.9.1. IMUL font: un operand explícit

Multiplica l'operand font per AL, AX, EAX, o RAX considerant el signe dels operands i emmagatzema el resultat a AX, DX:AX, EDX:EAX o RDX:RAX.

Operació

Si *font* és de 8 bits AX = AL * *font*

Si *font* és de 16 bits DX:AX = AX * *font*

Si *font* és de 32 bits EDX:EAX = EAX * *font*

Si *font* és de 64 bits RDX:RAX = RAX * *font*

Bits de resultat modificats

OF, SF, ZF, AF, PF, CF

Els bits de resultat SF, ZF, AF, i PF queden indefinits després d'executar-se la instrucció IMUL.

CF i OF es fixen a 0 si la part alta del resultat és 0 (AH, DX, EDX o RDX), en cas contrari es fixen a 1.

Formats vàlids

```
IMUL reg
IMUL mida mem
```

Exemples

```
IMUL ECX    ; EAX * ECX => EDX:EAX
IMUL QWORD [RBX]    ; AX * [RBX] => RDX:RAX
```

6.9.2. IMUL destinació, font: dos operands explícits

Multiplica l'operand font per l'operand destinació considerant el signe dels dos operands i emmagatzema el resultat a l'operand destinació; sobreescriu el valor que tingués.

Operació

destinació = font * destinació

Bits de resultat modificats

OF, SF, ZF, AF, PF, CF

Els bits de resultat SF, ZF, AF, i PF queden indefinits després d'executar-se la instrucció IMUL.

CF i OF es fixen a 0 si el resultat es pot representar amb l'operand destinació; si el resultat no és representable amb el rang de l'operand destinació (es produeix sobreeiximent), CF i OF es fixen a 1.

Formats vàlids

```
IMUL reg, imm
```

La mida de l'immediat pot anar des de 8 bits fins a la mida del primer operand, com a màxim 32 bits.

```
IMUL reg, reg
IMUL reg, mida mem
```

Els dos operands han de ser de la mateixa mida.

Exemples

```
IMUL EAX, 4
IMUL RAX, R9
IMUL RAX, QWORD [var]
```

6.10. IN: lectura d'un port d'entrada/sortida

IN destinació, font

Llegeix el valor d'un port d'E/S especificat per l'operand font i porta el valor a l'operand destinació.

L'operand font pot ser un valor immediat de 8 bits, que permet accedir als ports 0-255, o el registre DX, que permet accedir a qualsevol port d'E/S de 0-65535.

L'operand destinació només pot ser un dels registres següents:

- AL es llegeix un byte del port
- AX es llegeixen dos bytes
- EAX es llegeixen quatre bytes

Operació

destinació=font(port E/S)

Bits de resultat modificats

Cap

Formats vàlids

```
IN AL, imm8
IN AX, imm8
IN EAX, imm8
IN AL, DX
IN AX, DX
IN EAX, DX
```

Exemples

```
IN AL, 60h
IN AL, DX
```

6.11. INC: incrementa l'operand

INC destinació

Suma 1 a l'operand de la instrucció, i emmagatzema el resultat en el mateix operand.

Operació

destinació = destinació + 1

Bits de resultat modificats

OF, SF, ZF, AF, PF

Els bits de resultat es modifiquen segons el resultat de l'operació.

Formats vàlids

```
INC reg
INC mida mem
```

Exemples

```
INC AL
INC R9
INC BYTE [RBP]
INC QWORD [var1]
```

6.12. INT: crida a una interrupció software

```
INT servei
```

Crida a un servei del sistema operatiu, a una de les 256 interrupcions *software* definides a la taula de vectors d'interrupció. El número de servei ha de ser un valor entre 0 i 255. És habitual expressar el número del servei com un valor hexadecimal, de 00h a FFh.

Quan es crida una interrupció el registre EFLAGS i l'adreça de retorn són emmagatzemats a la pila.

Operació

RSP=RSP-8
M(RSP) ← EFLAGS
RSP=RSP-8
M(RSP) ← RIP
RIP ← adreça rutina de servei

Bits de resultat modificats

IF, TF

Aquests dos bits de resultat es posen a 0; posar a 0 el bit de resultat IF impedeix que es tracti una altra interrupció mentre s'està executant la rutina d'interrupció actual.

Formats vàlids

```
INT servei
```

Exemples

```
INT 80h
```

6.13. IRET: retorn d'interrupció

```
IRET
```

IRET s'ha d'utilitzar per a sortir de les rutines de servei a interrupcions (RSI). La instrucció treu de la pila l'adreça de retorn sobre el registre RIP, a continuació treu la paraula següent de la pila i la posa al registre EFLAGS.

Operació

RIP ← adreça retorn
RSP=RSP+8
EFLAGS ← M(RSP)
RSP=RSP+8

Bits de resultat modificats

Tots: OF, DF, IF, TF, SF, ZF, AF, PF, CF

Modifica tots els bits de resultat, ja que treu de la pila una paraula que és portada al registre EFLAGS.

Formats vàlids

```
IRET
```

Exemple

```
IRET
```

6.14. Jxx: salt condicional

```
Jxx etiqueta
```

Fa un salt segons una condició determinada; la condició es comprova consultant el valor dels bits de resultat (*flag*).

L'etiqueta codifica un desplaçament de 32 bits amb signe, i permet fer un salt de -2^{31} bytes a $+2^{31} - 1$ bytes.

Si la condició es compleix se salta a la posició del codi indicada per l'etiqueta; es carrega al registre RIP el valor RIP + desplaçament,

Operació

RIP = RIP + desplaçament

Bits de resultat modificats

Cap

Formats vàlids

Segons la condició de salt tenim les instruccions següents:

1) Instruccions que no tenen en compte el signe

Instrucció	Descripció	Condició
JA/JNBE	(Jump If Above/Jump If Not Below or Equal)	CF=0 i ZF=0
JAE/JNB	(Jump If Above or Equal/Jump If Not Below)	CF=0
JB/JNAE	(Jump If Below/Jump If Not Above or Equal)	CF=1
JBE/JNA	(Jump If Below or Equal/Jump If Not Above)	CF=1 o ZF=1

2) Instruccions que tenen en compte el signe

Instrucció	Descripció	Condició
JE/JZ	(Jump If Equal/Jump If Zero)	ZF=1
JNE/JNZ	(Jump If Not Equal/Jump If Not Zero)	ZF=0
JG/JNLE	(Jump If Greater/Jump If Not Less or Equal)	ZF=0 i SF=OF
JGE/JNL	(Jump If Greater or Equal/Jump If Not Less)	SF=OF
JL/JNGE	(Jump If Less/Jump If Not Greater or Equal)	SF≠OF
JLE/JNG	(Jump If Less or Equal/Jump If Not Greater)	ZF=1 o SF≠OF

3) Instruccions que comproven el valor d'un bit de resultat

```
Instrucció   Descripció                                  Condició
----------   ---------------------------------------   ---------
JC           (Jump If Carry flag set)                   CF=1
JNC          (Jump If Carry flag Not set)               CF=0
JO           (Jump If Overflow flag set)                OF=1
JNO          (Jump If Overflow flag Not set)            OF=0
JS           (Jump If Sign flag set)                    SF=1
JNS          (Jump If Sign flag Not set)                SF=0
```

Exemples

```
JE etiqueta1      ;salta si Z=1
JG etiqueta2      ;salta si Z=0 i SF=OF
JL etiqueta3      ;salta si SF≠OF
```

6.15. JMP: salt incondicional

JMP etiqueta

Salta de manera incondicional a l'adreça de memòria corresponent a la posició de l'etiqueta especificada; el registre RIP pren com a valor l'adreça de l'etiqueta.

Operació

RIP=adreça_etiqueta

Bits de resultat modificats

Cap

Formats vàlids

```
JMP etiqueta
```

Exemple

```
JMP bucle
```

6.16. LOOP: bucle fins a RCX=0

LOOP etiqueta

La instrucció utilitza el registre RCX.

Decrementa el valor de RCX, comprova si el valor és diferent de zero i en aquest cas fa un salt a l'etiqueta indicada.

L'etiqueta codifica un desplaçament de 32 bits amb signe, i permet fer un salt de -2^{31} bytes a $+2^{31} - 1$ bytes.

Operació

La instrucció és equivalent al conjunt d'instruccions següents:

```
DEC RCX
JNE etiqueta
```

Bits de resultat modificats

Cap

Formats vàlids

```
LOOP etiqueta
```

Exemple

```
MOV RCX, 10
bucle:
;
;Les instruccions es repetiran 10 cops
;
LOOP bucle
```

6.17. MOV: transferir una dada

MOV destinació, font

Copia el valor de l'operand font sobre l'operand destinació sobreescrivint el valor original de l'operand destinació.

Operació

destinació = font

Bits de resultat modificats

Cap

Formats vàlids

```
MOV reg, reg
MOV reg, mida mem
```

```
MOV mida mem, reg
```

Els dos operands han de ser de la mateixa mida.

```
MOV reg, imm
```

La mida de l'immediat pot anar des de 8 bits fins a la mida del registre; es permeten immediats de 64 bits si el registre és de 64 bits.

```
MOV mida mem, imm
```

La mida de l'immediat pot anar des de 8 bits fins a la mida del primer operand, com a màxim 32 bits.

Exemples

```
MOV RAX,R9
MOV RAX,QWORD [variable]
MOV QWORD [variable], RAX
MOV RAX,0102030405060708h
MOV WORD [RAX],0B80h
```

6.18. MOVSX/MOVSXD: transferir una dada amb extensió de signe

MOVSX destinació, font
MOVSXD destinació, font

Copia el valor de l'operand font sobre l'operand destinació sobreescrivint el valor original de l'operand destinació i fent l'extensió de signe. La mida de l'operand font ha de ser inferior a la mida de l'operand destinació.

MOVSXD només s'utilitza per a copiar un valor de 32 bits fent l'extensió sobre un valor de 64 bits.

Operació

destinació = Extensió signe (font)

Bits de resultat modificats

Cap

Formats vàlids

```
MOVSX reg16, reg8
```

```
MOVSX reg16, BYTE imm/mem

MOVSX reg32, reg8
MOVSX reg32, BYTE mem

MOVSX reg64, reg8
;reg8 pot ser qualsevol registre de 8 bits excepte: AH, BH,
CH, DH
MOVSX reg64, BYTE mem

MOVSX reg32, reg16
MOVSX reg32, WORD mem

MOVSX reg64, reg16
MOVSX reg64, WORD mem

MOVSXD reg64, reg32
MOVSXD reg64, DWORD mem
```

Exemples

```
MOVSX AX,BL          ;extensió de signe de 8 bits a 16 bits

MOVSX EAX,BL         ;extensió de signe de 8 bits a 32 bits

MOVSX RAX,BL         ;extensió de signe de 8 bits a 64 bits
MOVSX RAX, BYTE [var]

MOVSX EAX,BX         ;extensió de signe de 16 bits a 32 bits

MOVSX RAX,BX         ;extensió de signe de 16 bits a 64 bits

MOVSXD RAX, EBX      ;extensió de signe de 32 bits a 64 bits
MOVSXD RAX, DWORD [var]
```

6.19. MOVZX: transferir una dada afegint zeros

MOVZX destinació, font

Copia el valor de l'operand font sobre l'operand destinació sobreescrivint el valor original de l'operand destinació, afegint zeros al davant. La mida de l'operand font ha de ser inferior a la mida de l'operand destinació.

Operació

destinació = Extensió amb zeros (font)

Bits de resultat modificats

Cap

Formats vàlids

```
MOVZX reg16, reg8
MOVZX reg16, BYTE mem

MOVZX reg32, reg8
MOVZX reg32, BYTE mem

MOVZX reg64, reg8
;reg8 pot ser qualsevol registre de 8 bits excepte: AH, BH,
CH, DH
MOVZX reg64, BYTE mem

MOVZX reg32, reg16
MOVZX reg32, WORD mem

MOVZX reg64, reg16
MOVZX reg64, WORD mem
```

Exemples

```
MOVZX AX,BL          ;extensió amb zeros de 8 bits a 16 bits

MOVZX EAX,BL         ;extensió amb zeros de 8 bits a 32 bits

MOVZX RAX,BL         ;extensió amb zeros de 8 bits a 64 bits
MOVZX RAX, BYTE [var]

MOVZX EAX,BX         ;extensió amb zeros de 16 bits a 32 bits

MOVZX RAX,BX         ;extensió amb zeros de 16 bits a 64 bits
MOVZX RAX, WORD [var]
```

6.20. MUL: multiplicació entera sense signe

MUL font

MUL multiplica l'operand explícit per AL, AX, EAX o RAX sense considerar el signe dels operands, i emmagatzema el resultat a AX, DX:AX, EDX:EAX o RDX:RAX.

Operació

Si *font* és de 8 bits AX = AL * *font*

Si *font* és de 16 bits DX:AX = AX * *font*

Si *font* és de 32 bits EDX:EAX = EAX * *font*

Si *font* és de 64 bits RDX:RAX = RAX * *font*

Bits de resultat modificats

OF, SF, ZF, AF, PF, CF

Els bits de resultat SF, ZF, AF, i PF queden indeterminats després d'executar-se la instrucció MUL.

CF i OF es fixen a 0 si la part alta del resultat és 0 (AH, DX, EDX, o RDX); en cas contrari es fixen a 1.

Formats vàlids

```
MUL reg
MUL mida mem
```

Exemples

```
MUL CH           ; AL * CH → AX
MUL BX           ; AX * BX → DX:AX
MUL RCX          ; RAX * RCX → RDX:RAX
MUL WORD [BX+DI] ; AX * [BX+DI] → DX:AX
```

6.21. NEG: negació aritmètica en complement a 2

NEG destinació

Fa una negació aritmètica de l'operand, és a dir fa el complement a 2 de l'operand especificat; és equivalent a multiplicar el valor de l'operand per –1.

Aquesta operació no és equivalent a complementar tots els bits de l'operand (instrucció NOT).

Operació

destinació = (–1) * destinació

Bits de resultat modificats

OF, SF, ZF, AF, PF, CF

Si l'operand és 0, el resultat no varia i ZF=1 i CF=0; en cas contrari ZF=0 i CF=1.

Si l'operand conté el màxim valor negatiu (per exemple 80000000h si l'operand és de 32 bits), el valor de l'operand no es modifica però OF=1 i CF=1.

SF=1 si el resultat és negatiu; SF=0 en cas contrari.

PF=1 si el nombre d'uns del byte de menys pes del resultat és parell; PF=0 en cas contrari.

Formats vàlids

```
NEG reg
NEG mida mem
```

Exemples

```
NEG RCX
NEG DWORD [variable]
```

6.22. NOT: negació lògica (negació en complement a 1)

NOT destinació

Fa una negació lògica de l'operand, és a dir fa el complement a 1 de l'operand especificat, i complementa tots els bits de l'operand.

Operació

destinació = –destinació

Bits de resultat modificats

Cap

Formats vàlids

```
NOT reg
NOT mida mem
```

Exemples

```
NOT RAX
NOT QWORD [variable]
```

6.23. OUT: escriptura en un port d'entrada/sortida

OUT destinació, font

Escriu el valor de l'operand font en un port d'E/S especificat per l'operand destinació.

L'operand destinació pot ser un valor immediat de 8 bits, que permet accedir als ports 0-255, o el registre DX, que permet accedir a qualsevol port d'E/S de 0-65535.

L'operand font només pot ser un dels registres següents:

- AL s'escriu un byte
- AX s'escriuen dos bytes
- EAX s'escriuen quatre bytes

Operació

destinació(port E/S) = font

Bits de resultat modificats

Cap

Formats vàlids

```
OUT imm8, AL
OUT imm8, AX
OUT imm8, EAX
OUT DX, AL
OUT DX, AX
OUT DX, EAX
```

Exemples

```
OUT 60h, AL
OUT DX, AL
```

6.24. OR: o lògica

OR destinació, font

Fa una operació lògica OR (o lògica) bit a bit entre l'operand destinació i l'operand font; el resultat de l'operació es guarda sobre l'operand destinació sobreescrivint el valor inicial. El valor de l'operand font no es modifica.

Es fa una operació OR entre el bit *n* de l'operand destinació i el bit *n* de l'operand font segons la taula de veritat de la funció OR:

x	y	x OR y
0	0	0
0	1	1
1	0	1
1	1	1

Operació

destinació = destinació OR font

Bits de resultat modificats

OF=0, SF, ZF, PF, CF=0

Els bits de resultat OF i CF es posen a 0, la resta de bits de resultat es modifiquen segons el resultat de l'operació.

Formats vàlids

```
OR reg, reg
OR reg, mida mem
OR mida mem, reg
```

Els dos operands han de ser de la mateixa mida.

```
OR reg, imm
OR mida mem, imm
```

La mida de l'immediat pot anar des de 8 bits fins a la mida del primer operand, com a màxim 32 bits.

Exemples

```
OR R9, RAX
OR RAX, QWORD [variable]
OR DWORD [variable], EAX
OR RAX, 0x01020304
OR BYTE [vector+RAX], 5
```

6.25. POP: treure valor del cim de la pila

POP destinació

Treu el valor que es troba al cim de la pila (copia el valor de la memòria apuntat pel registre RSP), i l'emmagatzema a l'operand destinació especificat; es treuen tants bytes de la pila com la mida de l'operand indicat.

A continuació s'actualitza el valor del registre apuntador de pila, RSP, incrementant-lo en tantes unitats com el nombre de byte extrets de la pila.

L'operand pot ser un registre de 16 o 64 bits o una posició de memòria de 16 o 64 bits.

Operació

destinació = M[RSP]

La instrucció és equivalent a fer:

```
MOV destinació, [RSP]
ADD RSP, <mida de l'operand>
```

Per exemple:

```
POP RAX
```

és equivalent a:

```
MOV RAX, [RSP]
ADD RSP, 8
```

Bits de resultat modificats

Cap

Formats vàlids

```
POP reg
```

```
POP mida mem
```

La mida de l'operand ha de ser de 16 o 64 bits

Exemples

```
POP AX
POP RAX
POP WORD [variable]
POP QWORD [RBX]
```

6.26. PUSH: introduir un valor a la pila

PUSH font

S'actualitza el valor del registre apuntador de pila, RSP, decrementant-lo en tantes unitats com la mida en bytes de l'operand font.

A continuació s'introdueix el valor de l'operand font al cim de la pila, es copia el valor de l'operand a la posició de la memòria apuntada pel registre RSP, i es posen tants bytes a la pila com la mida de l'operand indicat.

L'operand pot ser un registre de 16 o 64 bits, una posició de memòria de 16 o 64 bits o un valor immediat de 8, 16 o 32 bits estès a 64 bits.

Operació

M[RSP]=font

La instrucció és equivalent a fer:

```
SUB RSP, <mida de l'operand>
MOV [RSP], font
```

Per exemple:

```
PUSH RAX
```

és equivalent a:

```
SUB RSP, 8
MOV [RSP], RAX
```

Bits de resultat modificats

Cap

Formats vàlids

```
PUSH reg
PUSH mida mem
```

L'operand ha de ser de 16 o 64 bits.

```
PUSH imm
```

El valor immediat pot ser de 8, 16 o 32 bits.

Exemples

```
PUSH AX
PUSH RAX
PUSH WORD [variable]
PUSH QWORD [RBX]
PUSH 0Ah
PUSH 0A0Bh
PUSH 0A0B0C0Dh
```

6.27. RET: retorn de subrutina

```
RET
```

Surt de la subrutina que s'estava executant i retorna al punt on s'havia fet la crida, a la instrucció següent de la instrucció CALL.

Treu de la pila l'adreça de memòria de retorn (l'adreça de la instrucció que segueix en seqüència a la instrucció CALL) i la carrega en el RIP (*instruction pointer*).

Actualitza el punter de pila (registre RSP), perquè apunti al següent element de la pila; com que l'adreça de retorn és de 8 bytes (en mode de 64 bits), incrementa RSP en 8 unitats.

Operació

RIP = M(RSP)
RSP = RSP + 8

Bits de resultat modificats

Cap

Formats vàlids

```
RET
```

Exemple

```
RET
```

6.28. ROL: rotació a l'esquerra

ROL destinació, font

Fa una rotació dels bits de l'operand destinació cap a l'esquerra, és a dir, cap al bit més significatiu; rota tants bits com indica l'operand font.

Els bits passen de la posició que ocupen a la posició de la seva esquerra; el bit de la posició més significativa passa a la posició menys significativa de l'operand

Per exemple, en un operand de 32 bits, el bit 0 passa a ser el bit 1, el bit 1 a ser el bit 2, i així successivament fins al bit 30, que passa a ser el bit 31, el bit més significatiu (bit 31) passa a ser el bit menys significatiu (bit 0).

L'operand font només pot ser un valor immediat de 8 bits o el registre CL.

Si l'operand destinació és de 64 bits, s'emmascaren els dos bits de més pes de l'operand font, cosa que permet rotacions de 0 a 63 bits.

Si l'operand destinació és de 32 bits o menys, s'emmascaren els tres bits de més pes de l'operand font, cosa que permet rotacions de 0 a 31 bits.

Bits de resultat modificats

OF, CF

El bit més significatiu es copia en el bit de transport (CF) cada cop que es desplaça un bit.

Si l'operand font val 1, OF s'activa si el signe de l'operand destinació original és diferent del signe del resultat obtingut; en qualsevol altre cas OF queda indefinit.

Formats vàlids

```
ROL reg, CL
ROL reg, imm8
ROL mida mem, CL
ROL mida mem, imm8
```

Exemples

```
ROL RAX,CL
ROL RAX,1
ROL DWORD [RBX],CL
ROL QWORD [variable],4
```

6.29. ROR: rotació a la dreta

ROR destinació, font

Fa una rotació dels bits de l'operand destinació cap a la dreta, és a dir, cap al bit menys significatiu; rota tants bits com indica l'operand font.

Els bits passen des de la posició que ocupen a la posició de la seva dreta; el bit de la posició menys significativa passa a la posició més significativa de l'operand.

Per exemple, en un operand de 32 bits, el bit 31 passa a ser el bit 30, el bit 30 a ser el bit 29, i així successivament fins al bit 1, que passa a ser el bit 0; el bit menys significatiu (bit 0) passa a ser el bit més significatiu (bit 31).

L'operand font només pot ser un valor immediat de 8 bits o el registre CL.

Si l'operand destinació és de 64 bits, s'emmascaren els dos bits de més pes de l'operand font, cosa que permet rotacions de 0 a 63 bits.

Si l'operand destinació és de 32 bits o menys, s'emmascaren els tres bits de més pes de l'operand font, cosa que permet rotacions de 0 a 31 bits.

Bits de resultat modificats

OF, CF

El bit menys significatiu es copia en el bit de transport (CF) cada cop que es desplaça un bit.

Si l'operand font val 1, OF s'actualitza amb el resultat de la XOR dels dos bits més significatius del resultat; en qualsevol altre cas OF queda indefinit.

Formats vàlids

```
ROR reg, CL
ROR reg, imm8
ROR mida mem, CL
ROR mida mem, imm8
```

Exemples

```
ROR RAX,CL
ROR RAX,1
ROR DWORD [RBX],CL
ROR QWORD [variable],4
```

6.30. SAL: desplaçament aritmètic (o lògic) a l'esquerra

SAL destinació, font

Fa un desplaçament a l'esquerra dels bits de l'operand destinació; desplaça tants bits com indica l'operand font.

Els bits passen de la posició que ocupen a la posició de la seva esquerra i es van afegint zeros per la dreta; el bit més significatiu es passa al bit de transport (CF).

L'operand font només pot ser un valor immediat de 8 bits o el registre CL.

Si l'operand destinació és de 64 bits, s'emmascaren els dos bits de més pes de l'operand font, cosa que permet desplaçaments de 0 a 63 bits.

Si l'operand destinació és de 32 bits o menys, s'emmascaren els tres bits de més pes de l'operand font, cosa que permet desplaçaments de 0 a 31 bits.

L'operació és equivalent a multiplicar per 2 el valor de l'operand destinació tants cops com indica l'operand font.

Operació

destinació = destinació * 2^N, en què N és el valor de l'operand font

Bits de resultat modificats

OF, SF, ZF, PF, CF

CF rep el valor del bit més significatiu de l'operand destinació, cada cop que es desplaça un bit.

Si l'operand font val 1, OF s'activa si el signe de l'operand destinació original és diferent del signe del resultat obtingut; en qualsevol altre cas OF queda indefinit.

La resta de bits es modifiquen segons el resultat de l'operació.

Formats vàlids

```
SAL reg, CL
SAL reg, imm8
SAL mida mem, CL
SAL mida mem, imm8
```

Exemples

```
SAL RAX,CL
SAL RAX,1
SAL DWORD [RBX],CL
SAL QWORD [variable],4
```

6.31. SAR: desplaçament aritmètic a la dreta

SAR destinació, font

Fa un desplaçament a la dreta dels bits de l'operand destinació; desplaça tants bits com indica l'operand font.

Els bits passen de la posició que ocupen a la posició de la seva dreta; el bit de signe (bit més significatiu) es va copiant a les posicions de la dreta; el bit menys significatiu es copia al bit de transport (CF).

L'operand font només pot ser un valor immediat de 8 bits o el registre CL.

Si l'operand destinació és de 64 bits, s'emmascaren els dos bits de més pes de l'operand font, cosa que permet desplaçaments de 0 a 63 bits.

Si l'operand destinació és de 32 bits o menys, s'emmascaren els tres bits de més pes de l'operand font, cosa que permet desplaçaments de 0 a 31 bits.

L'operació és equivalent a dividir per 2 el valor de l'operand destinació tants cops com indica l'operand font.

Operació

destinació = destinació/2^N, on N és el valor de l'operand font.

Bits de resultat modificats

OF, SF, ZF, PF, CF

Si l'operand font val 1, OF s'actualitza amb el resultat de la XOR dels dos bits més significatius del resultat; en qualsevol altre cas OF queda indefinit.

CF rep el valor del bit menys significatiu de l'operand destinació, cada cop que es desplaça un bit.

La resta de bits es modifiquen segons el resultat de l'operació.

Formats vàlids

```
SAR reg,CL
SAR reg, imm8
SAR mida mem, CL
SAR mida mem, imm8
```

Exemples

```
SAR RAX,CL
SAR RAX,1
SAR DWORD [RBX],CL
SAR QWORD [variable],4
```

6.32. SBB: resta amb transport (*borrow*)

SBB destinació, font

Fa una resta considerant el valor del bit de transport (CF). Es resta el valor de l'operand font de l'operand destinació, a continuació es resta del resultat el valor de CF, el resultat final de l'operació es guarda sobre l'operand destinació sobreescrivint el valor inicial. El valor de l'operand font no es modifica.

Operació

destinació = destinació – font – CF

Bits de resultat modificats

OF, SF, ZF, AF, PF, CF

El bit de resultat SF pren el valor 1 si el resultat de l'operació és negatiu; la resta dels bits de resultat es modifiquen segons el resultat de l'operació.

Formats vàlids

```
SBB reg, reg
SBB reg, mida mem
```

```
SBB mida mem, reg
```

Els dos operands han de ser de la mateixa mida.

```
SBB reg, imm
SBB mida mem, imm
```

La mida de l'immediat pot anar des de 8 bits fins a la mida del primer operand, com a màxim 32 bits.

Exemples

```
SBB R9,RAX
SBB RAX, QWORD [variable]
SBB DWORD [variable],EAX
SBB RAX,0x01020304
SBB BYTE [vector+RAX], 5
```

6.33. SHL: desplaçament lògic a l'esquerra

És equivalent al desplaçament aritmètic a l'esquerra (vegeu la instrucció SAL).

6.34. SHR: desplaçament lògic a la dreta

```
SHR destinació, font
```

Fa un desplaçament a la dreta dels bits de l'operand destinació; desplaça tants bits com indica l'operand font.

Els bits passen de la posició que ocupen a la posició de la seva dreta, el bit menys significatiu es copia al bit de transport (CF) i es van afegint zeros per l'esquerra.

L'operand font només pot ser un valor immediat de 8 bits o el registre CL.

Bits de resultat modificats

OF, SF, ZF, PF, CF

CF rep el valor del bit menys significatiu de l'operand destinació, cada cop que es desplaça un bit.

Si l'operand font val 1, OF s'actualitza amb el resultat de la XOR dels dos bits més significatius del resultat; en qualsevol altre cas OF queda indefinit.

La resta de bits es modifiquen segons el resultat de l'operació.

Formats vàlids

```
SHR reg, CL
SHR reg, imm8
SHR mida mem, CL
SHR mida mem, imm8
```

Exemples

```
SHR RAX,CL
SHR RAX,1
SHR DWORD [RBX],CL
SHR QWORD [variable],4
```

6.35. SUB: resta sense transport

SUB destinació, font

Fa una resta sense considerar el valor del bit de transport (CF). Es resta el valor de l'operand font de l'operand destinació, el resultat de l'operació es guarda sobre l'operand destinació sobreescrivint el valor inicial. El valor de l'operand font no es modifica.

Operació

destinació = destinació – font

Bits de resultat modificats

OF, SF, ZF, AF, PF, CF

El bit de resultat SF pren el valor 1 si el resultat de l'operació és negatiu; la resta de bits de resultat es modifiquen segons el resultat de l'operació.

Formats vàlids

```
SUB reg, reg
SUB reg, mida mem
SUB mida mem, reg
```

Els dos operands han de ser de la mateixa mida.

```
SUB reg, imm
SUB mida mem, imm
```

La mida de l'immediat pot anar des de 8 bits fins a la mida del primer operand, com a màxim 32 bits.

Exemples

```
SUB R9,RAX
SUB RAX, QWORD [variable]
SUB DWORD [variable],EAX
SUB RAX,0x01020304
SUB BYTE [vector+RAX], 5
```

6.36. TEST: comparació lògica

TEST destinació, font

Fa una operació lògica 'i' bit a bit entre els dos operands sense modificar el valor de cap dels operands; actualitza els bits de resultat segons el resultat de la 'i' lògica.

Operació

destinació AND font

Bits de resultat modificats

OF, SF, ZF, AF, PF, CF

Els bits de resultat CF i OF prenen el valor 0, la resta de bits de resultat es modifiquen segons el resultat de l'operació.

Formats vàlids

```
TEST reg, reg
TEST reg, mida mem
TEST mida mem, reg
```

Els dos operands han de ser de la mateixa mida.

```
TEST reg, imm
TEST mida mem, imm
```

La mida de l'immediat pot anar des de 8 bits fins a la mida del primer operand, com a màxim 32 bits.

Exemples

```
TEST R9,RAX
TEST RAX, QWORD [variable]
TEST DWORD [variable],EAX
TEST RAX,0x01020304
TEST BYTE [vector+RAX], 5
```

6.37. XCHG: intercanvi d'operands

XCHG destinació, font

Es fa un intercanvi entre els valors dels dos operands. L'operand destinació pren el valor de l'operand font, i l'operand font pren el valor de l'operand destinació.

No es pot especificar el mateix operand com a font i destinació, ni cap dels dos operands pot ser un valor immediat.

Bits de resultat modificats

No es modifica cap bit de resultat.

Formats vàlids

```
XCHG reg, reg
XCHG reg, mida mem
XCHG mida mem, reg
```

Els dos operands han de ser de la mateixa mida.

Exemples

```
XCHG R9, RAX
XCHG RAX, QWORD [variable]
XCHG DWORD [variable], EAX
```

6.38. XOR: o exclusiva

XOR destinació, font

Fa una operació lògica XOR ('o exclusiva') bit a bit entre l'operand destinació i l'operand font, el resultat de l'operació es guarda sobre l'operand destinació sobreescrivint el valor inicial. El valor de l'operand font no es modifica.

Es fa una operació XOR entre el bit *n* de l'operand destinació i el bit *n* de l'operand font segons la taula de veritat de la funció XOR:

```
 x   y   x XOR y
--- ---  -------
 0   0      0
 0   1      1
 1   0      1
 1   1      0
```

Operació

destinació = destinació XOR font

Bits de resultat modificats

OF=0, SF, ZF, PF, CF=0

Els indicadors OF i CF es posen a 0; la resta d'indicadors es modifiquen segons el resultat de l'operació.

Formats vàlids

```
XOR reg,reg
XOR reg,mida mem
XOR mida mem,reg
```

Els dos operands han de ser de la mateixa mida.

```
XOR reg,imm
XOR mida mem,imm
```

La mida de l'immediat pot anar des de 8 bits fins a la mida del primer operand, com a màxim 32 bits.

Exemples

```
XOR R9, RAX
XOR RAX, QWORD [variable]
XOR DWORD [variable], EAX
XOR RAX, 01020304h
XOR BYTE [vector+RAX], 5
```