Nasm x86_64

Guida Rapida

Creato da

Leonardo De Faveri

Introduzione al Nasm

Netwide Assembler, NASM, è un assembler 80x86 e x86-64 progettato nell'ottica della portabilità e della modularità. Supporta una vasta gamma di formati di file oggetto, inclusi Linux e * BSD a.out, ELF, COFF, Mach-O, formato OBJ (OMF) a 16 e 32 bit, Win32 e Win64.

L'Assembler produce semplici file binari nei formati Intel hex e Motorola S-Record.
La sua sintassi è progettata per essere semplice e facile da capire, è simile alla sintassi nel Manuale per gli sviluppatori del software Intel, ma con una complessità minima. Supporta tutte le estensioni architettoniche x86 attualmente conosciute e ha un forte supporto per le macro.

NASM include anche una serie di utility per la gestione del formato file oggetto personalizzato RDOFF.

Struttura di un programma

Un programma Nasm è suddiviso in 3 sezioni:

- ***.data*** : contiene le variabili inizializzate
- ***.bss*** : contiene le variabili non inizializzate
- ***.text*** : contiene il codice e le istruzioni del programma

Una parte fondamentale di un programma assembly sono le etichette (*label*).
Le etichette servono per etichettare una parte di codice e rendere possibile l'utilizzo di istruzioni di salto che sono usate per muoversi tra le etichette.

Le etichette sono definite in questo modo:

```
<nome_etichetta>:
```

In un programma la più importante delle etichette è l'etichetta *_start*, che identifica il punto d'ingresso del programma (un po' come la funzione *main* nel C++).

Nella sezione *.text* infatti la prima istruzione solitamente è: *global _start*.
L'istruzione *global* serve per far sapere al linker l'indirizzo di un'etichetta e in questo caso specifico fa sapere al linker l'indirizzo del punto di ingresso del programma.

Esistono anche etichette cosiddette locali, che vengono associate alla precedente etichetta non-locale. Le etichette locali vengono definite come le etichette standard, ma il loro nome deve iniziare con un punto.

Schema base di un programma:

```
section .data
    ...

section .bss
    ...

section .text
    global _start

_start:
    ...
```

Registri

I registri sono parti del processore usate per memorizzare temporaneamente dei dati.
Nell'architettura x86_64 i registri sono tutti a 64 bit, ciò significa che possono codificare 2^{64} valori distinti.
I registri messi a disposizione dall'Assembler Nasm sono 16.

8 bit	16 bit	32 bit	64 bit
al	ax	eax	rax
bl	bx	ebx	rbx
cl	cx	ecx	rcx
dl	dx	edx	rdx
sil	si	esi	rsi
dil	di	edi	rdi
dbp	bp	ebp	rbp
spl	sp	esp	rsp
r8b	r8w	r8d	r8
r9b	r9w	r9d	r9
r10b	r10w	r10d	10
r11b	r11w	r11d	r11
r12b	r12w	r12d	r12
r13b	r13w	r13d	r13
r14b	r14w	r14d	r14
r15b	r15w	r15d	r15

I bit di un registro corrispondono ai bit meno significativi del corrispondente registro di ordine superiore

Costanti

Le costanti sono simboli ai quali sono associati dei valori, che non possono essere modificati durante l'esecuzione del programma.

Le costanti vengono definite usando le istruzioni *%define* e *equ* e per convenzione i nomi delle costanti devono essere scritti in maiuscolo.

```
CONST1 equ 3

%define CONST2 4
```

La differenza tra *equ* e *%define* sta nel momento in cui viene valutato il valore della costante.
Con *equ* il valore è valutato nel momento in cui viene definita la costante, mentre con *%define* la valutazione del valore viene fatta nel momento in cui questo deve essere utilizzato.

```
ADDR1 equ $

%define ADDR2 $
```

In questo caso il valore di *ADDR1* sarà la posizione dell'assembly nel momento in cui viene incontrato *equ*, cioè nel punto in cui *ADDR1* è stato definito, mentre il valore di *ADDR2* sarà la posizione dell'assembly nel momento in cui *ADDR2* viene richiamato, quindi avrà valori diversi ad ogni utilizzo.

Variabili

Una variabile è una spazio in memoria RAM identificato da un nome e che può contenere dei valori.

La sintassi per la dichiarazione delle variabili cambia in base alla sezione in cui avviene.
Nella sezione *.data* si utilizzano le istruzioni *db*, *dw*, *dd* e *dq* mentre nella sezione *.bss* si usa l'istruzione *resb*, *resw*, *resd* e *resq*.

```
section .data
    <nome_variabile> db/dw/dd/dq <valore>, <valore>, ...
    variabile1 db 23

section .bss
    <nome_variabile> resb/resw/resd/resq <numero_gruppi_di_byte>
    variabile2 resb 4
```

Istruzione	Significato	Byte allocati
db	Define Byte	1
dw	Define Word	2
dd	Define Double Word	4
dq	Define Quadruple Word	8
resb	Reserve Byte	1
resw	Reserve Word	2
resd	Reserve Double Word	4
resq	Reserve Quadruple Word	8

Stringhe

Nel Nasm le stringhe sono array di byte, nei quali ogni byte definisce un carattere, i caratteri sono infatti codificati con la codifica ASCII. Tutte le stringhe terminano con il carattere 0.

La definizione di una stringa funziona come la definizione di una qualsiasi variabile.

```
<nome> db <contenuto>, <terminatore_di_linea>
  ↓   ↓      ↓                  ↓
testo  db “questo è del testo”, 10
```

Nella tabella ASCII il 10 identifica il carattere terminatore di linea.
Nel momento in cui si dovesse stampare la stringa a schermo o su un file si dovrebbe decidere se, stampare anche il carattere terminatore e creare una nuova linea andando a capo, oppure non stamparlo a restare sulla stessa linea.

Inserire il carattere terminatore di linea nella definizione di una stringa comunque, non è obbligatorio, si potrebbe infatti omettere o inserire altri caratteri purché vengano separati da virgole.

Commenti

Più un programma diventa complicato più è importante che il codice sia correttamente commentato così da renderne più facile la comprensione.

In Nasm i commenti si fanno utilizzando un '*;*' dopo il quale tutto il testo della riga è considerato un commento.

```
;Questo è un commento

section .data
    anni db 4              ;numero di anni
    ;nome db "Franco", 10 Questa istruzione non verrà eseguita in quanto è stata
                           ;commentata
```

Direttive di Sistema

Le direttive di Sistema (System Call) sono servizi (funzioni) del Kernel che possono essere invocati.
A ogni direttiva è associato un ID, ossia un identificativo numerico.

L'invocazione di una direttiva di sistema può prevedere anche il passaggio di parametri, che viene effettuato usando i seguenti registri:

Argomento	Registro
ID	rax
parametro 1	rdi
parametro 2	rsi
parametro 3	rdx
parametro 4	r10
parametro 5	r8
parametro 6	r9

Per passare i parametri ai registri si utilizza l'istruzione *mov*.

```
mov <registro_destinazione>, <registro_sorgente/valore>
```

Una volta passato l'ID e i parametri ai rispettivi registri, la direttiva di sistema viene invocata con l'istruzione *syscall*.

Principali Direttive di Sistema

Syscall	rax	rdi	rsi	rdx	r10	r8	r9
sys_read	0	#file descriptor	$buffer	#count			
sys_write	1	#file descriptor	$buffer	#count			
sys_open	2	#file name	#flags	#mode			
sys_close	3	#file descriptor					
sys_nanosleep	35	$timespec	$timespec				
sys_exit	60	#error code					

Nella tabella degli argomenti i simboli # e *$* hanno dei particolari significati:

- **#**: il parametro è un numero
- ***$***: il parametro è un indirizzo di memoria

Primo programma: Hello World

Per realizzare l'Hello World è necessario utilizzare direttive `sys_write` e `sys_exit` che servono rispettivamente per, scrivere su schermo Hello World e terminare correttamente il programma.

Sys_write:

Syscall	rax	rdi	rsi	rdx	r10	r8	r9
sys_write	1	#file descriptor	$buffer	#count			

Argomento	Descrizione
File descriptor	0. Standard Input (tastiera) 1. Standard Output (terminale) 2. Standard Error
Buffer	Indirizzo di memoria della stringa che si vuole stampare/scrivere
Count	Lunghezza della stringa, cioè il numero di caratteri che la compongono

Se si volesse scrivere “Hello, World!\n” a schermo, l'invocazione corretta di *sys_write* sarebbe:

Syscall	rax	rdi	rsi	rdx	r10	r8	r9
sys_write	1	1	stringa	14			

dove `stringa` è il nome della variabile contenente la stringa e 14 è il numero di caratteri che la compongono incluso il carattere terminatore di linea (10 -> \n).

Sys_exit:

Syscall	rax	rdi	rsi	rdx	r10	r8	r9
sys_exit	60	#error code					

Argomento	Descrizione
Error code	Il codice di errore con il quale terminare il programma. Se non vi sono errori è 0.

Error code può essere qualsiasi numero in quanto è il programmatore ad assegnare ad ogni possibile errore un codice

Quindi per terminare con successo il programma si ha:

Syscall	rax	rdi	rsi	rdx	r10	r8	r9
sys_exit	60	0					

Ora è quindi possibile scrivere il codice per il programma Hello World.

Codice Hello World:

```
SYS_WRITE     equ 1
SYS_EXIT      equ 60

section .data
    message db “Hello, World!”, 10

section .text
    global _start

_start:
    mov     rax, SYS_WRITE
    mov     rdi, 1
    mov     rsi, message
    mov     rdx, 14
    syscall                 ;Invoca la direttiva di sistema sys_write

    mov     rax, SYS_EXIT
    mov     rdi, 0
    syscall                 ;Invoca la direttiva di sistema sys_exit
```

A questo punto per eseguire il programma è necessario compilarlo ed eseguire il linking e per fare ciò si usano i seguenti comandi:

Azione	Comando
Compilazione (release)	nasm -f elf64 -o <file_oggetto>.o <file_sorgente>
Compilazione (debug)	nasm -f elf64 -F DWARF -g <file_sorgente>.asm -o <file_oggetto>.o
Linking	ld <file_oggetto>.o -o <file_eseguibile>

La differenza tra la compilazione (release) e la compilazione (debug) è che la seconda include nell'eseguibile anche i simboli per il debugger, che permettono di eseguire il debug del programma utilizzando dei debugger come ad esempio il comando *gdb* di Linux.

Flag

I flag, come i registri, possono contenere valori, ma siccome sono rappresentati con un solo un bit, i valori che possono assumere possono essere solo valori booleani: `true` o `false`.
Tutti i flag fanno parte di uno stesso registro.

Flag	Descrizione
CF	Overflow
PT	Parità
ZF	Zero
SF	Segno
OF	Overflow
AF	Ausiliario
IF	Interrupt Attivato

I flag CF e OF identificano entrambi l'overflow, ma vengono settati in casi diversi:
CF:
1) ***Il risultato della somma tra due numeri senza segno supera la capacità del registro***
2) ***In una sottrazione tra numeri senza segno, viene sottratto il maggiore al minore***

OF:
1) ***Come il primo caso del CF, ma quando si usano numeri con il segno***

Puntatori

I puntatori (pointers) come i registri immagazzinano dati, ma in realtà non contengono il dato in sé, bensì il suo indirizzo di memoria.

Puntatore	Significato	Descrizione
rip (eip, ip)	Puntatore all'indice	Punta alla prossima istruzione del flusso di esecuzione
rsp (esp, sp)	Puntatore allo stack	Punta al primo indirizzo dello stack
rbp (ebp, bp)	Puntatore alla base dello stack	Punta all'ultimo indirizzo dello stack
...	...	...

Registri come puntatori

I registri possono essere trattati come puntatori e per farlo è necessario circondare il nome del registro con delle parentesi quadre.

Esempio:

```
mov rax, rbx     ;Carica in rax il valore contenuto nel registro rbx
```

Esempio:

```
mov rax, [rbx]   ;Carica in rax il valore a cui sta puntando rbx
```

Flusso di esecuzione

Tutti i programmi sono eseguiti dall'alto verso il basso e la direzione del flusso del programma è detta flusso di esecuzione.
Il puntatore *rip* (*ip* se si lavora a 16 bit, *eip* se si lavora a 32) contiene l'indirizzo della prossima istruzione da eseguire e dopo che l'istruzione è stata eseguita il contenuto del puntatore *rip* viene incrementato di 1 (64bit se si è in un'architettura a 64 bit).
Quindi se è stata eseguita l'istruzione *x*, *rip* punterà all'istruzione *x+1*.

Salti incondizionali e l'istruzione jump:

L'istruzione jump può essere usata per saltare da una parte all'altra del codice e permette quindi, di modificare il flusso di esecuzione del programma.

```
jmp <etichetta>   ;Questa istruzione cerca nel registro rip l'indirizzo
                  ;dell'etichetta
```

Controlli e confronti:

I controlli consentono di modificare il flusso di esecuzione sulla base di determinate condizioni e sono effettuati sui registri.

È infatti possibile confrontare il contenuto di 2 registri:

```
cmp rax, rbx  ;Confronto il valore contenuto in rbx con quello contenuto in rax
```

...o il contenuto di un registro ed un valore:

```
cmp rax, 23       ;Confronto il valore contenuto in rax con il 23
```

Dopo ogni controllo il valore di alcuni flag viene modificato:

cmp a, b	
a = b	ZF = 1
a ≠ b	ZF = 0
msb(a - b)	SF = 1/0
...	...

Il caso msb(a - b) significa che viene fatta la differenza tra a e b e se il bit più significativo del risultato è 1, SF=1 altrimenti, SF=0

Salti condizionali:

I salti condizionali possono essere effettuati dopo un controllo in quanto si basano sullo stato dei flag.
I salti condizionali si scrivono come i salti incondizionali, ma l'istruzione *jmp* è sostituita da un simbolo per il salto condizionale.

Simbolo (con segno)	Simbolo (senza segno)	risultato di: `cmp a,b`
je	-	a = b
jne	-	a ≠ b
jg	ja	a > b
jge	jae	a ≥ b
jl	jb	a < b
jle	jbe	a ≤ b
jz	-	a = 0
jnz	-	a ≠ 0
jo	-	C'è stato overflow
jno	-	Non c'è stato overflow
js	-	C'è il segno
jns	-	Non c'è il segno

Le istruzioni di salto `js` e `jns` vengono eseguite se il flag del segno è settato o meno

Esempio:

```
cmp rax, 23       ;Questo codice salterà all'indirizzo dell'etichetta _doThis
je _doThis        ;se e solo se il valore contenuto in rax è uguale a 23
```

Esempio:

```
cmp rax, rbx      ;Questo codice salterà all'indirizzo dell'etichetta _doThis se
jg _doThis        ;e solo se il valore contenuto in rax è maggiore di quello
                  ;contenuto in rbx
```

Subroutine e l'istruzione call

Una subroutine è una sequenza di istruzioni che è possibile richiamare all'interno del codice.
Per richiamare una subroutine si utilizza l'istruzione *call*.

```
call <nome_etichetta>    ;Istruzione per richiamare una subroutine
```

L'istruzione *call* funziona come l'istruzione *jmp*, tranne per il fatto che con *call* è possibile ritornare al punto di partenza.
L'ultima istruzione di ogni subroutine infatti dovrebbe essere l'istruzione *ret* che consente di ritornare all'istruzione *call* che ha richiamato la subroutine e riprendere l'esecuzione del programma dall'istruzione successiva.

Esempio:

```
section .data
    text db “Hello, World!”, 10

section .text
    global _start

_start:
    call _printHello  ;Richiama la subroutine _printHello

    mov rax, 60
    mov rdi, 1
    syscall


_printHello:          ;Inizio della subroutine
    mov rax, 1
    mov rdi, 1
    mov rsi, text
    mov rdx, 14
    syscall
    ret               ;La prossima istruzione eseguita sarà: mov rax, 60
```

Interrogare l'utente e la direttiva sys_read

Un programma nella sua definizione più semplice è in insieme di istruzioni le quali sulla base di dati forniti in input producono un output.
Per produrre un output è stata utilizzata la direttiva *sys_write*, mentre per richiedere un input si utilizza la direttiva *sys_read*.

Syscall	rax	rdi	rsi	rdx	r10	r8	r9
sys_read	0	#file descriptor	$buffer	#count			

Argomento	Descrizione
File descriptor	0. Standard Input (tastiera) 1. Standard Output (terminale) 2. Standard Error
Buffer	Indirizzo di memoria del buffer in cui salvare i caratteri letti
Count	Lunghezza del buffer o numero di caratteri da leggere

Se si volesse ricevere in input il nome di una persona e salvarlo in un buffer di 10 byte si avrebbe:

Syscall	rax	rdi	rsi	rdx	r10	r8	r9
sys_read	0	0	buffer	10			

dove *buffer* è il nome della variabile in cui salvare i caratteri letti e 10 è il numero massimo di caratteri che possono essere letti.
Se non vengono occupati tutti i byte del buffer viene inserito in automatico alla fine della stringa il carattere terminatore di linea (10).

Codice per ricevere una stringa

```
section .bss
    buffer resb 10       ;Vengono riservati 10 byte da usare per ricevere la stringa

section .text
    global _start

_start:
    mov rax, 0
    mov rdi, 0
    mov rsi, buffer
    mov rdx, 10
    syscall              ;Viene richiamata sys_read

    mov rax, 1
    mov rdi, 1
    mov rsi, buffer
    mov rdx, 10
    syscall              ;Viene richiamata sys_write per stampare la stringa letta

    mov rax, 60
    mov rdi, 0
    syscall              ;Viene richiamata sys_exit
```

Funzioni matematiche

Le operazioni matematiche sono usate per manipolare matematicamente il contenuto dei registri.
Tutte le operazioni hanno la stessa forma generale:

```
<operazione> <operando_principale>, <operando_secondario>
```

Elemento	Descrizione
operazione	È l'istruzione che identifica il tipo di operazione
operando_principale	È sempre un registro ed è il registro nel quale viene salvato il risultato ed è l'unico il cui valore viene modificato
operando_secondario	Può essere un registro o un valore

Esempio:

```
add rax, 5    ;Aggiunge 5 al valore contenuto in rax
sub rbx, rcx  ;Sottrae al valore contenuto in rbx il valore contenuto in rcx
```

Tabella delle operazioni matematiche:

Nome operazione (senza segno)	Nome operazione (con segno)	Descrizione
add a,b	-	a = a + b
sub a, b	-	a = a - b
mul registro	imul registro	rax = rax * registro
div registro	idiv registro	rax = rax / registro
neg registro	-	registro = -registro
inc registro	-	registro = registro + 1
dec registro	-	registro = registro - 1
adc registro	-	a = a + b+ CF
sbb registro	-	a = a - b - CF

Dopo l'operazione `div` nel registro `rdx` viene inserito il resto e i registri `rdx` e `rax` vengono concatenati in un unico registro a 128 bit.
Per evitare questo effetto collaterale, che potrebbe causare errori, si deve mettere a 0 `rdx` prima di eseguire la divisione.

Stack

Lo stack (pila) è un altro modo per memorizzare i dati.
Lo stack funziona come una pila; al suo interno i dati vengono impilati uno sopra l'altro in base all'ordine di inserimento, ciò significa che l'ultimo elemento dello stack è il primo ad essere stato inserito, mentre l'elemento in cima allo stack è l'ultimo inserito.
L'unico modo per interagire con un dato all'interno dello stack è rimuovere prima tutti i dati impilati sopra.

Stack terminologia:

Quando si lavora sullo stack è importante utilizzare i termini corretti.

Termine	Descrizione
Pushing	Aggiungere dati in cime allo stack
Popping/Pulling	Rimuovere dati dalla cima dello stack
Peeking	Esaminare il dato in cima allo stack, ma senza rimuoverlo

Stack operazioni:

Operazione	Descrizione
push registro/valore	Aggiunge un valore sulla cima dello stack
pop registro	Rimuove il valore in cima allo stack e lo salva nel registro
mov registro, [rsp]	Legge il valore in cima allo stack e lo salva nel registro, ma senza rimuoverlo

rsp è un registro che punta all'indirizzo del primo elemento dello stack

Esempio:

```
push rax          ;carica nello stack il valore contenuto in rax
```

Esempio:

```
pop rbx           ;Rimuove il valore in cima allo stack e lo salva in rbx
```

Esempio:

```
mov rcx, [rsp]    ;Copia in rcx il valore in cima allo stack
```

Stack e puntatori:

Di solito quando in un'operazione si possono usare i registri è anche possibile usare i puntatori, quindi per salvare direttamente in memoria il valore in cima alla pila sarebbe lecito scrivere:

```
pop [registro]
```

In questo modo il valore in cima allo stack verrebbe salvato direttamente in memoria all'indirizzo contenuto nel registro.

Macro

Una macro è una macro istruzione, cioè un'istruzione che contiene al suo interno un insieme di istruzioni.
Per definire una macro si utilizzano le parole chiave *%macro* e *%endmacro* che indicano rispettivamente l'inizio e la fine della macro.

Scheletro di una macro:

Una macro è sempre definita da un nome, un numero di argomenti e un corpo.

```
%macro <nome> <argc>
    ...
    <corpo>
    ...
%endmacro
```

Elementi	Descrizione
nome	Identifica il nome della macro e sarà usato per richiamarla
argc	Identifica il numero di argomenti della macro
corpo	Contiene un insieme di istruzioni che definiscono cosa farà la macro

Esempio:

```
%macro exit 0     ;La seguente macro si chiama exit, non riceve parametri
    mov rax, 60   ;e se richiamata invoca la direttiva sys_exit e termina il
    mov rdi, 0    ;programma
    syscall
%endmacro
```

Utilizzare i parametri:

Gli argomenti o parametri sono input che possono essere passati ad una macro.
All'interno della macro per fare riferimento ad un argomento si utilizza un simbolo percentuale seguito dal numero dell'argomento (il primo avrà numero 1).

Esempio:

```
%macro exit 1     ;Questa macro riceve un parametro; il codice di errore, e per
    mov rax, 60   ;usarlo all'interno della macro si usa la formula %1
    mov rdi, %1
    syscall
%endmacro
```

Invocare una macro:

Per invocare una macro si deve semplicemente usare il nome della macro eventualmente seguito dai valori dei parametri separati tra loro da una virgola.

Esempio:

```
exit            ;Viene invocata la macro chiamata exit senza passare alcun parametro
```

Esempio:

```
exit rbx        ;Viene invocatala macro exit passando il valore in rbx come parametro
```

Esempio:

```
exit 23         ;Viene invocatala macro exit passando il valore 23 come parametro
```

Ogni volta che si richiama una macro il suo corpo viene inserito nel corpo del programma.

Definire etichette in una macro:

Siccome quando una macro viene richiamata il suo corpo viene inserito all'interno del codice del programma chiamante, se una macro all'interno della quale sono state definite delle etichette viene richiamata più volte sicuramente si verificherà un errore, in quanto, non possono esistere 2 o più etichette con lo stesso nome.

Per risolvere questo problema, quando si definiscono etichette all'interno di una macro bisogna porre due simboli percentuali '%%' davanti al nome dell'etichetta.
Naturalmente le etichette definite all'interno di una macro saranno accessibili solo al codice della macro.

Librerie e file esterni

Un programma assembly può essere suddiviso in più file i quali possono poi essere uniti usando la parola chiave *%include*, che deve essere usata prima di qualsiasi altra istruzione.

```
%include "<percorso_file/percorso_libreria>"
```

Includendo un file in un altro si da la possibilità al codice del file includente di usare il codice del file incluso. Durante la compilazione infatti il codice di un file incluso viene copiato nel codice del file includente nelle posizioni in qui questo è stato utilizzato.
Solitamente le costanti e le macro vengono definite in un file, di estensione *.asm* o *.inc*, che viene poi incluso nel file del programma principale.

Esempio:

```
%include "standardlib.inc"

.section text
    global _start

_start:
    exit        ;La macro exit è stata definita in standardlib.inc
```

Argomenti da linea di comando

Quando si esegue un programma dalla linea di comando è possibile passargli degli argomenti.
Per fare ciò è necessario scrivere dopo il nome del programma il valore dei parametri separandoli con uno spazio.

```
$ ./program arg1 arg2 arg3
```

All'interno del programma tutti i paramatri ottenuto saranno gestiti come stringhe, quindi per i parametri numerici sarà necessaria una conversione da stringa a numero.

Quando il programma viene eseguito tutti i parametri vengono automaticamente caricati nello stack.
Il primo elemento dello stack è il numero di argomenti ed il suo valore è sempre almeno 1, anche nel caso in cui l'utente non fornisca nessun parametro.
Questo avviene perché il sistema operativo fornisce in automatico il percorso del programma, che infatti è il valore successivo al numero di argomenti.
Tutti i valori successivi nello stack sono invece gli argomenti specificati dall'utente ordinati dal primo all'ultimo.

Esempio:

```
$ ./program arg1 25 arg3
```

Stack	Valore
argc	4
*path	“./program”
*arg[1]	“arg1”
*arg[2]	“25”
*arg[3]	“arg3”

Il primo elemento dello stack è quindi il numero di argomenti (*argc*) il cui valore è sempre uno in più rispetto al numero di argomenti forniti dall'utente perché include anche il percorso del programma (*path*) fornito in automatico dal sistema operativo.

Fornire argomenti dalla linea di comando è utile perché consente di passare dei dati al programma ricavati da altri comandi, ad esempio sarebbe possibile fornire in input ad un programma assembly l'output di un comando della shell Linux.

Esempio:

```
$ ./program `ls`  ;program riceverà in input i nomi di tutti i file e le cartelle
                  ;presenti nell'attuale working directory
```

File

Un'altra componente importante per un programma assembly sono i file. Leggere e scrivere dati su file è fondamentale per i programmi più complessi.

I permessi:

Non è, però sempre possibile interagire con un file, è infatti necessario che il file acconsenta ad interagire con esso. Ciò che specifica quali azioni possono essere effettuate su un file, sono i permessi.

I permessi indicano a chi è consentito leggere, scrivere e/o eseguire un file.
I permessi vengono rappresentati sotto forma di 3 cifre ottali, le quali indicano rispettivamente i permessi per il proprietario del file, il gruppo di utenti al quale appartiene il proprietario e gli altri utenti.
In realtà è anche possibile utilizzare una quarta cifra ottale per rappresentare i permessi speciali, ma solitamente non è necessario.

Valore	Lettura	Scrittura	Esecuzione
0	×	×	×
1	×	×	✓
2	×	✓	×
3	×	✓	✓
4	✓	×	×
5	✓	×	✓
6	✓	✓	×
7	✓	✓	✓

	Speciali	Proprietario	Gruppo	Altri
1	Sticky bit	Esecuzione	Esecuzione	Esecuzione
2	Setgid	Scrittura	Scrittura	Scrittura
4	Setuid	Lettura	Lettura	Lettura

Esempio:

```
3764o   Assegna I permessi speciali Sticky bit e Setgid, concede tutti i permessi
        al proprietario, i permessi di lettura e scrittura al gruppo e soltanto il
        permesso di lettura agli altri utenti
```

Esempio:

```
0777o   Non assegna nessuno dei permessi speciali, mentre concede a tutti tutti gli
        utenti tutti i permessi.
```

Interagire coi file:

Per interagire coi file si devono usare 5 direttive di sistema.

Syscall	rax	rdi	rsi	rdx	r10	r8	r9
sys_read	0	#file descriptor	$buffer	#count			
sys_write	1	#file descriptor	$buffer	#count			
sys_open	2	$file name	#flags	#mode			
sys_close	3	#file descriptor					
sys_lseek	8	#file descriptor	#offset	#origin			

Aprire un file:

Prima di eseguire qualsiasi tipo di operazione su un file è necessario aprirlo e per farlo si usa la direttiva *sys_open*.

Tipo di argomento	Descrizione
File name	È un puntatore al nome del file, cioè una stringa conclusa dal carattere di terminazione 0
Flags	Identifica i flag associati al file
Mode	Sono le 4 cifre ottali dei permessi

I flag specificano in che modo deve essere aperto un file e sono rappresentati da potenze di 2.
Questo rende possibile combinare i flag sommandoli.

Flag	Valore	$\log_2$(valore)	Descrizione
O_RDONLY	0	null	Solo lettura
O_WRONLY	1	0	Solo scrittura
O_RDWR	2	1	Lettura e scrittura
O_CREAT	64	6	Crea il file se non esiste
O_APPEND	1024	10	Aggiunge il nuovo testo dopo quello già esistente
O_DIRECTORY	65535	16	Apre una cartella
O_PATH	2097152	21	Ottiene un riferimento al file, ma senza aprirlo
O_TMPFILE	4194304	22	Crea un file che non potrà essere aperto da altri processi

La direttiva *sys_open* restituisce nel registro *rax* il file descriptor del file aperto.
Tale valore servirà per interagire con il file.

Leggere e scrivere su un file:

Per leggere e scrivere su file si usano le direttive *sys_read* e *sys_write;* il loro utilizzo è identico a quando le si usa per richiedere un input all'utente o per stampare testo a schermo, l'unica differenza è che come file descriptor bisogna utilizzare il valore restituito da *sys_open*.

Le operazioni di lettura e di scrittura sui file avvengono entrambe sequenzialmente, questo significa che se si scrivono caratteri su un file e dopo si tenta di leggere dal file, i caratteri che sono stati appena scritti non verranno letti in quanto il cursore sarà posizionato dopo.

Per evitare che questo si verifichi è necessario riposizionare all'inizio del file il cursore, invocando la direttiva *sys_lseek*.

sys_lseek:

La direttiva *sys_lseek* può quindi essere usata per spostare il cursore all'interno del file.

Tipo di argomento	Descrizione
File name	È un puntatore al nome del file, cioè una stringa conclusa dal carattere di terminazione 0
Offset	Indica il numero di byte del quale spostare il cursore
Origin	0. SEEK_SET sposta il cursore a partire dall'inizio del file 1. SEEK_CUR sposta il cursore a partire dalla sua attuale posizione 2. SEEK_END sposta il cursore a partire dalla fine del file

Quindi, per spostare il cursore all'inizio del file si ha:

syscall	rax	rdi	rsi	rdx	r10	r9	r8
sys_lseek	8	fileDescriptor	0	0			

dove *fileDescriptor* è una variabile contenente il file descriptor del file.
Con questa invocazione il cursore viene spostato di 0 byte a partire dall'inizio del file, cioè il cursore viene spostato all'inizio del file.

Il cursore può essere spostato sia in avanti che in dietro ed a definire la direzione dello spostamento è il segno dell'`offset`, che se è positivo sposta il cursore in avanti, altrimenti lo sposta all'indietro.

Con questa direttiva è anche possibile spostare il cursore e scrivere oltre la fine del file.
In questo modo vengono a crearsi spazi vuoti all'interno del file e tutti i caratteri all'interno di quegli spazi avranno come codice 0.

La direttiva *sys_lseek* restituisce in *rax* l'offset del cursore dopo lo spostamento e quindi sarà 0 nel caso in cui il cursore sia stato spostato all'inizio del file e corrisponderà alla dimensione del file nel caso in cui sia stato spostato alla fine, mentre negli altri casi l'offset sarà un numero compreso tra 0 e la dimensione in byte del file.

Se, invece, si volesse ottenere l'offset attuale si avrebbe:

syscall	rax	rdi	rsi	rdx	r10	r9	r8
sys_lseek	8	fileDescriptor	0	1			

dove *fileDescriptor* è una variabile contenente il file descriptor del file.
Con questa invocazione il cursore viene spostato di 0 byte a partire dalla posizione attuale del cursore, cioè il cursore non viene spostato e come output in *rax* si otterrà l'offset attuale del cursore.

In caso di errore, in *rax* viene invece inserito il codice di errore.

Chiudere un file:

Una volta che si ha terminato di interagire con un file è fondamentale chiuderlo correttamente.
Per farlo si utilizza la direttiva *sys_close*, la quale chiude il file associato al file descriptor ricevuto come parametro.

Codice per interagire con un file:

```
SYS_READ   equ 0
SYS_WRITE  equ 1
SYS_OPEN   equ 2
SYS_CLOSE  equ 3
SYS_LSEEK  equ 8
SYS_EXIT   equ 60

O_CREAT    equ 64
O_RDWR     equ 2

section .data
    filename db “file.txt”, 0
    message db “testo da scrivere”, 10

section .bss
    buffer resb 18

section .text
    global _start

_start:
    mov rax, SYS_OPEN
    mov rdi, filename
    mov rsi, O_CREAT+O_RDWR
    mov rdx, 0644o
    syscall              ;Viene richiamata sys_open per aprire il file

    push rax             ;Viene caricato nello stack il file descriptor ottenuto

    mov rax, SYS_WRITE
    mov rdi, [rsp]       ;Viene spostato in rdi il file descriptor salvato
    mov rsi, message
    mov rdx, 18
    syscall              ;Viene richiamata sys_write per scrivere sul file

    mov rax, SYS_LSEEK
    mov rdi, [rsp]
    mov rsi, 0
    mov rdx, 0
    syscall              ;Viene riposizionato il cursore all'inizio del file

    mov rax, SYS_READ
    mov rdi, [rsp]
    mov rsi, buffer      ;La riga letta verrà salvata in buffer
    mov rdx, 18
    syscall              ;Viene richiamata sys_read per leggere una riga dal file

    mov rax, SYS_CLOSE
    pop rdi              ;Il file descriptor viene rimosso dallo stack e caricato
                         ;in rdi
    syscall              ;Viene richiamata sys_close per chiudere il file

    mov rax, SYS_EXIT
    mov rdi, 0
    syscall              ;Viene richiamata sys_exit per terminare il programma
```

Interrompere l'esecuzione di un programma

Certe volte può essere utile interrompere l'esecuzione di un programma per un determinato lasso di tempo.
Per farlo viene utilizzata la direttiva *sys_nanosleep* la quale può interrompere l'esecuzione del programma per un certo periodo di tempo.

Rappresentare il tempo: timespec

Per invocare *sys_nanosleep* è però necessario riuscire a rappresentare una quantità di tempo e per farlo si utilizza la struttura *timespec*.
timespec è una struttura di dati che serve a rappresentare una quantità di tempo ed è definita da 2 valori: *tv_sec* e *tv_nsec*. Entrambi i valori sono interi positivi a 64 bit (*qword*, 8 byte).
Il valore massimo di *tv_nsec* è 999 999 999 perché 1 secondo equivale a 1 000 000 000 nanosecondi.

Dichiarazione di una timespec:

```
section .data
    timespec dq <numero_secondi>, <numero_nanosecondi>
```

sys_nanosleep:

Syscall	rax	rdi	rsi	rdx	r10	r8	r9
sys_nanosleep	35	$timespec	$timespec				

Questa direttiva può quindi essere usata per mettere in pausa il programma per un certo ammontare di tempo con una precisione al nanosecondo. Riceve come parametri 2 riferimenti a strutture di tipo *timespec*.
Il primo riferimento serve per indicare il tempo di pausa, mentre il secondo, che è opzionale, serve eventualmente a contenere lo scarto.

Questa direttiva infatti, può far riprendere anticipatamente l'esecuzione del programma nel caso in cui venga ricevuto un segnale o di terminazione del programma o di invocazione di un handler e in questo caso il tempo restante di pausa che non è stato speso viene salvato nella seconda struttura, la quale potrà poi essere usata per invocare nuovamente *sys_nanosleep* e terminare la pausa.

Codice per mettere in pausa il programma:

```
SYS_NANOSLEEP equ 35

section .data
    delay dq 5, 500000000    ;Definizione della struttura timespec

section .text
    global _start

_start:
    mov rax, SYS_NANOSLEEP   ;Viene interrotta l'esecuzione del programma per
    mov rdi, delay           ;5 secondi e 500 000 000 nanosecondi (5.5 secondi),
    mov rsi, 0               ;passati i quali riprenderà l'esecuzione del
    syscall                  ;programma

    ...
```

Strutture di dati

Si definisce struttura di dati un'entità usata per organizzare un insieme di dati all'interno della memoria del computer, ed eventualmente per memorizzarli in una memoria di massa.
La scelta delle strutture di dati da utilizzare è strettamente legata a quella degli algoritmi, per questo, spesso essi vengono considerati insieme. Infatti, la scelta della struttura di dati influisce inevitabilmente sull'efficienza degli algoritmi che la manipolano.

La struttura di dati è un metodo di organizzazione dei dati e prescinde quindi da ciò che vi è effettivamente contenuto.

I record:

Un record è una struttura dati che può essere eterogenea, se contiene elementi di tipo diverso, o omogenea, se contiene solo elementi dello stesso tipo.
Gli elementi che lo compongono sono detti anche campi, e sono identificati da un nome.

Definizione un record:

La definizione di un record prevede la creazione di un nuovo tipo di dato identificato da un nome e da dei campi i quali sono a loro volta definiti dal loro nome e dal loro tipo.

Per definire un record si usano le macro *`struc`* e *`endstruc`*.
La macro *`struc`* riceve due parametri; il nome del record e l'offset di inizio del record, tuttavia questo secondo parametro è opzionale e se non fornito di default l'offset di partenza è 0.
Il nome di un record è definito come un simbolo, il cui valore è l'offset di inizio del record, ed oltre a questo viene definito anche il simbolo con *`_size`* concatenato al nome del record che ne rappresenta la dimensione in byte.

Con l'invocazione di *`struc`* si avvia la definizione del record, i cui campi sono definiti usando le istruzioni della famiglia *`resb`* (*`resb`*, *`resw`*, *`resd`*, *`resq`*).
Infine con l'invocazione di *`endstruc`* termina la definizione del record.

Se si volesse definire un record chiamato *`my_record_persona`* contenente 3 campi; nome, cognome, eta rispettivamente di tipo stringa di byte, stringa di byte e byte, si avrebbe:

```
struc my_record_persona
    nome:      resb 20
    cognome:   resb 20
    eta:       resw 1
endstruc
```

In questo modo è stato definito un record chiamato `my_record_persona` con 3 campi e un simbolo chiamato `my_record_persona_size` il cui valore sarà 41 (20+20+1), cioè la dimensione in byte del record.

In realtà se nel programma vengono definiti più record i cui nomi dei campi sono uguali, devo usare le etichette locali, ovvero i nomi dei campi devono iniziare con il punto.

Definire un riferimento ad un record senza macro:

In precedenza è stata utilizzata la struttura dati *timespec* per richiamare la *sys_nanosleep*.
Per definire un riferimento a tale struttura è stata utilizzata la seguente istruzione:

```
timespec dq 5, 500000000    ;Definizione della struttura timespec
```

Ma come fare se la struttura è definita da campi di tipo diverso?
In quel caso la sintassi è simile.

Ad esempio se si volesse definire un rifermento al record dell'esempio precedente, si avrebbe:

```
section .data
    <nome_del_riferimento>  db <nome>
                            db <cognome>
                            dw <eta>
```

In questo modo è possibile definire riferimenti a strutture che non sono state definite in precedenza, in quanto non viene indicato a quale struttura punterà il riferimento.

Definire un riferimento ad un record usando le macro:

Esiste anche un altro modo per definire rifermenti a record di cui conosciamo l'implementazione ed è utilizzando le macro *istruc* e *iend*.

Se si utilizzano queste macro la definizione del rifermento diventa:

```
section.data
    <nome_del_rifermento>:
        istruc my_record_persona
            at nome, db <nome>
            at cognome, db <cognome>
            at eta, dw <eta>
        iend
```

Se nella definizione dei campi del record sono state usate le etichette locali il nome del campo può essere scritto anche in forma estesa concatenando il nome del record e il nome del campo con la notazione puntata.
E quindi la definizione del rifermento diventa:

```
section .data

<nome_del_rifermento>:
    istruc my_record_persona
        at my_record_persona.nome, db <nome>
        at my_record_persona.eta, dw <eta>
        at my_record_persona.cognome, db <cognome>
    iend
```

Può anche essere dichiarato un rifermento ad un record, ma senza inizializzarne i campi.
Per farlo si usa, nella sezione *.bss*, l'istruzione *resb* usando come numero di byte da riservare la dimensione del record.

Quindi per dichiarare un riferimento ad un record del tipo dell'esempio precedente, si ha:

```
section .bss
    <nome_del_riferimento> resb my_record_persona_size
```

Accedere ai campi di un record in lettura:

Per accedere ad un campo di un record, il cui riferimento è stato definito senza usare le macro *istruc* e *iend*, è necessario usare i puntatori, conoscere l'offset del campo e la dimensione del dato da leggere o scrivere.

Per leggere e scrivere valori numerici devo conoscere l'esatta dimensione del dato, cioè devo sapere se si tratta di un *byte*, una *word*, una *double word* o una *quadruple word* e devo estrarre dal record il valore contenuto nell'area di memoria e non l'indirizzo.

Quindi il codice per accedere in lettura ai campi del record dell'esempio precedente ipotizzando di avere un riferimento ad esso, sarebbe:

```
section .text
    global _start

_start:
    ;Carica in rbx l'indirizzo della stringa del campo nome
    mov rbx, <nome_del_riferimento>

    ;Carica in rdx l'indirizzo della stringa del campo cognome
    mov rdx, <nome_del_riferimento> + <lunghezza_della_stringa_nome>

    ;Carica nei primi 16 bit di rcx (cx), il valore del campo eta (word)
    mov rcx, 0
    mov cx, [<nome_del_riferimento> + <lunghezza_delle_2_stringhe>]
```

Analizzando le istruzioni per accedere ad ogni campo si nota che:

- Per accedere al primo campo, il campo *nome*, non è stato necessario specificarne l'offset, in quanto, essendo il primo equivale all'offset di partenza del record che viene specificato dal rifermento.
  Inoltre, siccome il campo *nome* contiene una stringa (array di caratteri) nel registro *rbx* è stato caricato l'indirizzo di memoria della prima cella dell'array e non il valore.
- Per accedere al secondo campo, il campo *cognome*, si è specificato come offset la la lunghezza della stringa del primo campo. Trattandosi di un'altra stringa la lettura è stata effettuata come nel primo caso.
- Per accedere al terzo campo, il campo `eta`, è stato necessario specificare come offset la somma delle lunghezze delle due stringhe, *nome* e *cognome*. Questo perché i dati dei campi vengono salvati in memoria RAM in modo sequenziale.
  Inoltre essendo *eta* un campo contenente un singolo valore numerico, nel registro *cx* è stato caricato il valore del campo e non il suo indirizzo.
  Siccome il valore occupava 16 bit (*word*) non è stato caricato in *rcx*, che è a 64 bit, ma solo nei suoi primi 16 bit ai quali si accede usando il registro *cx*.
  Prima di leggere il valore però, nel registro *rcx* è stato caricato lo 0.
  Questo deve essere fatto in quanto se il valore letto non occupa tutti i 64 bit del registro, i bit non sovrascritti dalla lettura rimangono invariati.
  Ciò può condurre ad errori nell'esecuzione del programma in quanto, il valore del registro potrebbe non essere ciò che ci si aspetterebbe, dato che risentirebbe dei bit rimasti nel registro dal suo utilizzo precedente.

Risulta quindi evidente che il processo per accedere in lettura ai campi di un record, il cui riferimento è stato definito senza le macro *istruc* e *iend*, risulta macchinoso e difficoltoso,
in quanto bisogna sapere precisamente quanti `byte` sono stati usati prima di un certo campo.

Questo problema non sussiste nel caso in cui il riferimento sia stato definito usando le macro.
Infatti per accedere in lettura ai campi di quello stesso record, ma da un riferimento definito con le macro *istruc* e *iend*, si dovrebbe usare la seguente sintassi:

```
section .text
    global _start

_start:
    ;Carica in rbx l'indirizzo della stringa del campo nome
    mov rbx, <nome_del_riferimento> + my_record_persona.nome

    ;Carica in rdx l'indirizzo della stringa del campo cognome
    mov rdx, <nome_del_riferimento> + my_record_persona.cognome

    ;Carica nei primi 16 bit di rcx (cx), il valore del campo eta (word)
    mov rcx, 0
    mov cx, [<nome_del_riferimento> + my_record_persona.eta]
```

Se si analizzano di nuovo le istruzioni usate per accedere ad ogni campo si nota che:

- Per accedere al primo campo se ne è specificato comunque l'offset. In realtà questo si sarebbe anche potuto omettere in quanto come prima, l'offset del primo campo è uguale all'offset di partenza del record.
- Per accedere al secondo campo non è stato necessario conoscere la lunghezza delle stringa `nome`.
- Per accedere al terzo campo non è stato necessario sapere nulla riguardo ai campi precedenti.

Questo esempio mostra quindi, come sia più immediato l'accesso ai campi di un record, se lo si fa usando un riferimento definito usando le macro.
Anche se è comunque necessario conoscere la dimensione di un campo quando si vuole assegnare il valore in esso contenuto ad un registro.

Accedere ai campi di un record in scrittura:

Il processo per accedere in scrittura ai campi di un record segue le stesse regole del processo per accedere in lettura, ovvero, è necessario conoscere il tipo e le dimensione del dato che si vuole andare a scrivere.
Inoltre, anche il processo per accedere in scrittura è dipendente dal modo in cui il record era stato inizialmente definito.

Quindi per accedere in scrittura ai campi di un record, il cui riferimento è stato definito senza usare le macro, si usa la seguente sintassi:

```
section .text
    global _start

_start:
    ;Scrive nel record un carattere
    mov rbx, <nome_del_riferimento> + <offset>
    mov [rbx], byte "a"

    ;Sintassi alternativa
    mov [<nome_del_riferimento> + <offset>], byte "b"

    ;scrive nel record una word
    mov rcx, 0
    mov rcx, <nome_del_riferimento> + <offset>
    mov cx, word 54321

    ;Sintassi alternativa
    mov rdx, 12345
    mov [<nome_del_riferimento> + <offset>], dx

    ;Per scrivere valori di altre dimensioni la sintassi segue le stesse regole,
    ;ma è necessario usare gli indicatori di dimensione corretti e il registro giusto
```

Se si accedesse ai campi usando un riferimento definito con le macro la sintassi non cambierebbe.

Gli array:

In informatica, con le parole array o vettore ci si riferisce ad una struttura dati complessa, statica e omogenea. Gli array possono essere monodimensionali o multidimensionali e in questo caso vengono anche chiamati, matrici.

Gli array sono contenitori composti da caselle dette celle (o elementi). Ciascuna delle celle si comporta come una variabile tradizionale e tutte le celle sono variabili di uno stesso tipo preesistente, detto tipo base dell'array.

Gli array multidimensionali, invece, sono array i cui elementi solo a loro volta degli array.

Definizione di un array:

Per definire un array è necessario specificare il numero e la dimensione delle celle che lo comporranno.
Il numero di celle definisce la dimensione dell'array.

Quindi se si volesse definire un riferimento ad array di 5 *word* si userebbe la seguente sintassi:

```
section .data
    ;Definizione di un array già inizializzato
    <nome_del_vettore> dw <word1>, <word2>, <word3>, <word4>, <word5>

section .bss
    ;Definizione di un array non inizializzato
    <nome_del_vettore> resw 5
```

Accedere alle celle di un array:

L'accesso alle celle di un array, sia in scrittura che in lettura, è effettuato usando un indice. L'indice non è altro che l'offset della cella alla quale si vuole accedere.

Quindi per accedere in lettura e scrittura ad una cella di un array, si ha:

```
section .text
    global _start

_start:
    ;Lettura e scrittura di valori numerici (word)
    mov ax, [<nome_del_vettore>]   ;Carica in ax il valore della prima cella
    mov bx, [<nome_del_vettore>+2] ;Carica in bx il valore della seconda cella

    mov [<nome_del_vettore>+4], word 4 ;Scrive nella terza cella il valore 4
    mov cx, 17
    mov [<nome_del_vettore+6], cx      ;Scrive nella quarta cella il valore 17

    ;Lettura e scrittura di caratteri (byte)
    mov rdx, <nome_del_vettore>  ;Carica in rdx l'indirizzo del primo carattere
    mov r8, <nome_del_vettore>+2 ;Carica in r8 l'indirizzo del terzo carattere

    mov [nome_del_vettore+1], byte "a" ;Scrive 'a' nella seconda cella
    mov r9b, "b"
    mov [nome_del_vettore+3], r9b      ;Scrive 'b' nella quarta cella

    ;La dimensione dei caratteri in Nasm è sempre un byte
    ;perché sono codificati in ASCII
```

Come si è visto l'offset o indice delle celle dell'array di *word* è il doppio rispetto alle celle dell'array di caratteri (*byte*); questo perché l'indice di una cella è sempre dato dal prodotto tra il numero della cella all'interno dell'array e la dimensione in byte di una cella.
Quindi, siccome una *word* è definita da 2 *byte*, l'indice delle celle di un array di *word* sarà sempre doppio rispetto all'indice di una stessa cella in un array di *byte*.

Inoltre, le celle di un array sono sempre indicizzate a partire dallo 0, quindi per accedere alla prima cella di un vettore non è necessario specificarne l'offset.

Siccome le stringhe, in Nasm, sono array di caratteri è possibile accedere a ciascun carattere della stringa usando la stessa sintassi che usa con gli array

Definizione di un array multidimensionale:

In nasm gli array multidimensionali vengono definiti esattamente come gli array monodimensionali.

Quindi, per definire una matrice 3*5 di byte, cioè un array in grado di contenere 3 array di byte da 5 elementi ciascuno, si avrebbe:

```
section .data
    ;Definizione di una matrice già inizializzata
    <nome_della_matrice> db 0, 1, 2, 3, 4
                         db 5, 6, 7, 8, 9
                         db 10, 11, 12, 13, 14

section .bss
    ;Definizione di una matrice non inizializzata
    <nome_della_matrice> resb 15

    ;Sintassi alternativa (da preferire)
    <nome_della_matrice> resb 3*5
```

Accedere alle celle di una matrice:

Il processo per accedere alle celle di una matrice segue grosso modo quanto visto per gli array, è però necessario effettuare delle operazioni aggiuntive.
Le matrici infatti, memorizzano in memoria RAM i valori di ciascuna cella in modo sequenziale, ciò significa che la matrice dell'esempio precedente viene salvata in memorizzata in questo modo:

0, 1, 2 ,3 ,4 ,5 ,6 ,7 ,8 ,9 ,10 ,11 ,12 ,13 ,14

quindi, per accedere ad una cella bisogna conoscere il numero di elementi che compongono ciascun array di ogni dimensione della matrice.

Nel caso di matrici da 3 o più dimensioni il numero di array da scorrere per arrivare alla cella desiderata sarà 2 o più, in quanto una matrice, altro non è, che un array di array e il numero di dimensioni di una matrice è dato proprio dal numero di livelli di array che la definiscono
(matrice 2D -> array di array, matrice 3D -> array di array di array, ...).

Un'altra importa informazione da conoscere per accedere alla cella corretta è la dimensione in byte di ciascuna cella.