

# Appunti e Documentazione Assembly per Esercitazioni di Reti Logiche A.A. 2025/26

Raffaele Zippo

15 ottobre 2025

# Indice

<b>1 Esercitazioni di Reti Logiche</b>	<b>5</b>
1.1 Chi tiene il corso . . . . .	5
<b>2 Introduzione</b>	<b>6</b>
2.1 Perché compilare, testare, debuggare . . . . .	6
2.2 Ambienti utilizzati . . . . .	6
2.3 Domande e ricevimenti . . . . .	6
<b>3 Ambienti di sviluppo</b>	<b>7</b>
3.1 Editor . . . . .	7
3.2 Ambiente assembler . . . . .	7
3.3 Ambiente Verilog . . . . .	8
3.4 Versioni dell'ambiente e alternative . . . . .	8
3.5 Ambiente per Windows 11 + WSL2 . . . . .	8
3.6 Ambiente per Linux nativo o devcontainer . . . . .	9
3.6.1 Utilizzo nativo . . . . .	9
3.6.2 Utilizzo tramite devcontainer . . . . .	10
3.7 Alternative fai da te . . . . .	10
3.8 Testare gli ambienti . . . . .	10
3.8.1 Assembler . . . . .	10
3.8.2 Verilog . . . . .	10
<b>4 Essere efficienti con VS Code</b>	<b>12</b>
4.1 Le basi elementari . . . . .	12
4.2 Le basi un po' meno elementari . . . . .	12
4.3 Editing multi-caret . . . . .	13
<b>I Assembler - Introduzione</b>	<b>14</b>
<b>5 Ambiente di sviluppo</b>	<b>15</b>
5.1 Struttura dell'ambiente . . . . .	15
5.2 Lanciare l'ambiente e primo programma . . . . .	15
<b>II Assembler - Esercitazioni</b>	<b>18</b>
<b>6 Esercitazione 1</b>	<b>19</b>
6.1 Premesse per programmi nell'ambiente del corso . . . . .	19
6.2 Esercizio 1.1 . . . . .	20
6.3 Uso del debugger . . . . .	22
6.4 Domande a risposta multipla . . . . .	25
6.4.1 15/07/2025, domanda 9 . . . . .	25
6.4.2 24/06/2025, domanda 9 . . . . .	26
6.4.3 09/09/2025, domanda 3 . . . . .	27
6.5 Esercizi per casa . . . . .	27
6.5.1 Esercizio 1.2: istruzioni stringa . . . . .	27
6.5.2 Esercizi 1.3 e 1.4 . . . . .	27
6.5.3 Esercizio 1.5 . . . . .	27
6.5.4 Esercizio 1.6 . . . . .	28

<b>7 Esercitazione 2</b>	<b>29</b>
7.1 Esercizio 2.1: esercizio d'esame 2025-09-09	29
7.1.1 Richiami su sottoprogrammi	29
7.1.2 indigit_b7	30
7.1.3 innumber_b7	31
7.1.4 Divisibilità per 64	35
7.2 Domande a risposta multipla	36
7.2.1 2025-01-08, domanda 3	36
7.2.2 2025-06-04, domanda 2	36
7.2.3 2025-02-11, domanda 1	37
<b>8 Esercitazione 3</b>	<b>38</b>
8.1 Soluzioni passo-passo esercizi per casa	38
8.1.1 Esercizio 1.2: istruzioni stringa, soluzione passo-passo	38
8.1.2 Esercizio 1.6: esercizio di debugging, soluzione passo-passo	39
8.2 Esercizio 3.1: esercizio d'esame 2021-01-08	43
8.3 Esercizi per casa	43
8.3.1 Esercizio 3.2	43
8.3.2 Esercizio 3.3	43
8.3.3 Esercizio 3.4	48
8.3.4 Esercizio 3.5	50
<b>III Assembler - Documentazione</b>	<b>52</b>
<b>9 Architettura x86</b>	<b>53</b>
9.1 Registri	53
9.2 Memoria	54
9.3 Spazio di I/O	54
9.4 Condizioni al reset	54
<b>10 Sezione .data</b>	<b>55</b>
10.1 Direttive di allocazione	55
10.2 Valori letterali	55
<b>11 Istruzioni processore x86</b>	<b>57</b>
11.1 Immediati	57
11.2 Spostamento di dati	58
11.3 Aritmetica	58
11.4 Logica binaria	60
11.5 Traslazione e Rotazione	61
11.6 Controllo di flusso	62
11.7 Operazioni condizionali	62
11.8 Istruzioni stringa	64
11.8.1 Repeat Instruction	65
11.9 Altre istruzioni	65
<b>12 Sottoprogrammi di utility</b>	<b>67</b>
12.1 Terminologia	67
12.2 Caratteri speciali	67
12.3 Sottoprogrammi	68
<b>13 Debugger gdb</b>	<b>69</b>
13.1 Controllo dell'esecuzione	69
13.1.1 Problemi con next	70
13.2 Ispezione dei registri	70
13.3 Ispezione della memoria	70
13.4 Gestione dei breakpoints	71
13.4.1 Conditional Breakpoints	71
13.4.2 Watchpoints	72
<b>14 Tabella ASCII</b>	<b>73</b>

<b>15 Ambiente d'esame e i suoi script</b>	<b>75</b>
15.1 Aprire l'ambiente	75
15.2 Il terminale Powershell	76
15.3 Eseguire gli script	77
15.3.1 assemble.ps1	77
15.3.2 debug.ps1	77
15.3.3 run-single-test.ps1	77
15.3.4 run-multiple-tests.ps1	77
 <b>IV Assembler - Appendice</b>	 <b>78</b>
<b>16 Problemi comuni</b>	<b>79</b>
16.1 Setup dell'ambiente	79
16.1.1 1. Ho trovato un ambiente assembler per Mac su Github, ma ho problemi ad usarlo	79
16.1.2 2. Ho trovato un ambiente basato su DOS, usato precedentemente all'esame, ma ho problemi ad usarlo	79
16.1.3 3. Lanciando il file assemble.code-workspace, mi appare un messaggio del tipo Unknown distro: Ubuntu	79
16.1.4 4. Sto utilizzando una sistema Linux desktop, come uso l'ambiente senza virtualizzazione?	79
16.2 Uso dell'ambiente	80
16.2.1 5. Se premo Run su VS Code non viene lanciato il programma	80
16.2.2 6. Provando a lanciare ./assemble.ps1 programma.s ricevo un errore del tipo ./assemble.ps1: line 1: syntax error near unexpected token	80
16.2.3 7. Provando ad assemblare ricevo un warning del tipo warning: creating DT_TEXTREL in a PIE	80
16.2.4 8. Provando ad assemblare ricevo un warning del tipo missing .note.GNU-stack section implies executable stack	80
16.2.5 9. Ho modificato il codice per correggere un errore, ma quando assemblo e eseguo il codice, continuo a vedere lo stesso errore.	80
16.2.6 10. Dove trovo i file che scrivo nell'ambiente assembler?	80

\*

# Capitolo 1

## Esercitazioni di Reti Logiche

Questa dispensa contiene appunti e materiali per le esercitazioni del corso di Reti Logiche, Laurea Triennale di Ingegneria Informatica dell'Università di Pisa, A.A. 2025/26.

Il contenuto presume conoscenza degli aspetti teorici già discussi nel corso, ricordando alla bisogna solo gli aspetti direttamente collegati con gli esercizi trattati.

### **Materiale in costruzione**

Questa dispensa contiene materiale in via di stesura, è messa a disposizione per essere utile quanto prima. Potrebbero esserci inesattezze o errori. Siete pregati, nel caso, di [segnalarlo](#).

Il contenuto di partenza è quello dell'anno precedente, e verrà sostituito man mano in base a quello che viene svolto a lezione. Aspettativi quindi che cambi *molto* durante il trimestre del corso.

### 1.1 Chi tiene il corso

Il corso è tenuto dal [Prof. Giovanni Stea](#) . Le esercitazioni sono tenute dal [Dott. Raffaele Zippo](#) .

La pagina ufficiale del corso è [http://docenti.ing.unipi.it/~a080368/Teaching/RetiLogiche/index\\_RL.html](http://docenti.ing.unipi.it/~a080368/Teaching/RetiLogiche/index_RL.html).

# Capitolo 2

## Introduzione

### 2.1 Perché compilare, testare, debuggare

*If debugging is the process of removing bugs, then programming must be the process of putting them in.*  
Edsger W. Dijkstra

Si parta dal presupposto che fare errori *succede*. Meno è banale il progetto o esercizio, più è facile che da qualche parte si sbagli. La parte importante è riuscire a cogliere e rimuovere questi errori prima che sia troppo tardi, sia che si tratti di rilasciare un software in produzione o di consegnare l'esercizio a un esame. In queste esercitazioni vedremo questo processo in contesti specifici (software scritto in assembler e reti logiche descritte in Verilog) ma la linea si applica in generale in tutti gli altri ambiti dell'ingegneria informatica. Dunque il codice, di qualunque tipo sia, non va solo scritto, va *provato*. Come identificare, trovare e rimuovere gli errori è invece una capacità pratica che va *esercitata*.

### 2.2 Ambienti utilizzati

Gli strumenti a disposizione per provare e testare il codice, così come la loro praticità d'uso possono cambiare molto in base ad architettura, sistema operativo, e generale potenza delle macchine utilizzate.

Dato che il corso è collegato a un esame, ci si concentrerà sullo stesso ambiente che sarà disponibile all'esame, che è dunque basato su PC desktop con Windows 11 e architettura x86. Il software e le istruzioni a disposizione riguarderanno questa combinazione.

Per altre architetture e sistemi operativi, il supporto è sporadico e *best effort*, con nessuna garanzia da parte dei docenti che funzioni. Dovrete, con molta probabilità, litigare con il vostro computer per far funzionare il tutto.

Una introduzione generale alle opzioni è in [Ambienti software](#).

### 2.3 Domande e ricevimenti

Siamo a disposizione per rispondere a domande, spiegare esercizi, colmare lacune. Gli orari ufficiali di ricevimento sono comunicati durante il corso e tenuti aggiornati sulle pagine personali. È sempre una buona idea scrivere prima, via email o Teams, per evitare impegni concomitanti o risolvere più rapidamente in via testuale. In caso di dubbi su esercizi, aiuta molto allegare il testo dell'esercizio (foto o pdf) e il codice sorgente (sempre e solo file testuale, non foto o file binari).

Non è raro che gli studenti si sentano in imbarazzo o comunque evitino di fare domande, quindi ci spendo qualche parola in più. Fuori dall'esame, è nostro *compito* insegnare, e questo include rispondere alle domande. È un *diritto* degli studenti chiedere ricevimenti e avere risposte. Avere dubbi o lacune è in questo contesto *positivo*, perché sapere di non sapere qualcosa è un primo passo per imparare.

## Capitolo 3

# Ambienti di sviluppo

In questo corso, scriveremo codice per programmi assembler e per descrivere reti logiche in Verilog. Per entrambi, utilizziamo un ambiente software che è lo stesso (o estremamente simile) a quello che si troverà all'esame.

### 3.1 Editor

Nelle esercitazioni e nella documentazione faremo riferimento a [VS Code](#), che è l'unico editor che si potrà utilizzare all'esame.

Non c'è però nessun obbligo a usare VS Code per le esercitazioni personali, qualunque editor di file di testo andrà bene. Anche un editor da terminale come `nano` o `vim`.

### 3.2 Ambiente assembler

Programmare in assembler vuol dire programmare per una specifica architettura di processori. L'architettura x86 è stata rimpiazzata nel tempo da x64, a 64 bit, che è del tutto retrocompatibile. Altre architetture (in particolare, ARM) hanno istruzioni, registri e funzionamento completamente diversi e non sono compatibili con x86. Usare una macchina con architettura diversa è inevitabilmente fonte di problemi.

L'ambiente fornito funziona con Linux x86 (o x64 o amd64, che significano la stessa cosa). Non funziona invece per processori arm64, come quelli usati da Mac o Windows on ARM.

Da una parte, si potrebbe pensare di esercitarsi scrivendo assembler per la propria architettura, anziché quella usata nel corso. Sorgono diversi problemi:

- dover imparare sintassi, meccanismi, registri completamente diversi;
- dover fare a meno o reingegnerizzarsi la libreria usata per l'input-output a terminale;
- dover comunque imparare l'assembler mostrato nel corso, perché quella sarà richiesta all'esame e supportata dalle macchine in laboratorio.

La seconda opzione è usare strumenti di virtualizzazione capaci di far girare un sistema operativo con architettura diversa. Sorge come principale problema l'ergonomicità ed efficienza di questa soluzione, che dipende molto dagli strumenti che si trovano e dalle caratteristiche hardware della macchina, che potrebbero essere non sufficienti.

Per chi ha una macchina ARM, sarà necessario trovare soluzioni di virtualizzazione o usare un'altra macchina dedicata (va bene qualunque cosa di qualunque potenza, purché x86). In ogni caso, *non offriamo nessun supporto diretto* a tali macchine. Lo ribadisco in rosso, perché chiesto spesso.

#### **Nessun supporto diretto per Mac con ARM**

Non testiamo né supportiamo ambienti per Mac con ARM, che non abbiamo a disposizione. *Ci è stato detto* che [UTM](#) può emulare l'architettura x86, affermazione che riportiamo senza alcuna garanzia. Non risponderemo a ulteriori domande a riguardo, soprattutto se parte delle [domande frequenti](#).

Oltre a questioni di architettura, abbiamo anche il sistema operativo, che è rilevante per gestire input e output da terminale. I programmi che scriveremo ed eseguiremo, così come quelli utilizzati per assemblare, gireranno in un terminale Linux. Nei pacchetti forniti e in sede di esame, si usa in particolare Ubuntu 24.04.

**Perché Linux?**

Perché è molto più facile virtualizzare un ambiente Linux moderno in Windows o Mac che il contrario. In precedenza si usava MS-DOS, un sistema del 1981 facilmente emulabile, ma molto limitante data l'età.

Per assemblare, si usa `gcc`, per debuggare `gdb`. Per usarli però sono necessari comandi *lunghi*, che semplifichiamo usando script Powershell `assemble.ps1` e `debug.ps1`.

**Perché Powershell?**

Perché Powershell (2006) è object-oriented, e permette di scrivere script leggibili e manutenibili, in modo semplice. Bash (1989) è invece text-oriented, con una [lunga lista di trappole da saper evitare](#).

**32 vs 64 bit**

In realtà, i processori x86 a soli 32 bit non sono più in commercio da vent'anni. I processori che si trovano oggi sono x64, a 64 bit, e sono in grado di eseguire codice a 32 bit per retrocompatibilità. Nel corso, continuiamo ad usare l'istruzione set a 32 bit perché

2. è di complessità ridotta e sufficiente per i nostri scopi didattici,
2. il vecchio ambiente DOS, che qualcuno può trovare ancora utile, supporta *solo* x86.

### 3.3 Ambiente Verilog

L'ambiente Verilog non ha i problemi di quello assembler, perché quel che compiliamo (una rete simulabile) non è legato sistema operativo o all'architettura della CPU. Basta che si riescano ad installare

- `iverilog` e `vvp`
- GTKWave

### 3.4 Versioni dell'ambiente e alternative

L'ambiente dell'A.A. 2025/26 è leggermente diverso da quello degli anni precedenti. Le differenze riguardano solo aspetti di installazione e configurazione, il modo di utilizzo rimane pressoché invariato.

Se si ha già un ambiente funzionante, non c'è bisogno di fare nulla.

L'ambiente è fornito in due versioni:

- Windows 11 + WSL2
- Linux nativo o devcontainer

Questi contengono sia istruzioni per installazione e configurazione, sia le cartelle `assembler` e `verilog` con i file necessari per scrivere codice.

Tenere presente che non c'è bisogno di utilizzare lo stesso tipo di pacchetto o macchina per assembler e Verilog, le due scelte sono indipendenti.

### 3.5 Ambiente per Windows 11 + WSL2

#### Download

Questo pacchetto supporta macchine Windows 11 x64, utilizza WSL2 per virtualizzare un sistema Ubuntu 24.04 per assembler, e applicazioni native per Verilog.

WSL2 è un sottosistema di Windows che permette di virtualizzare macchine Linux in modo semplice, e l'integrazione con VS Code tramite [l'estensione WSL](#) permette di scrivere codice *fuori* dalla macchina virtuale ed assemblare ed eseguire *dentro* la macchina virtuale. Questo ci permette di mantenere un ambiente grafico moderno mentre si lavora con un terminale Linux virtualizzato.



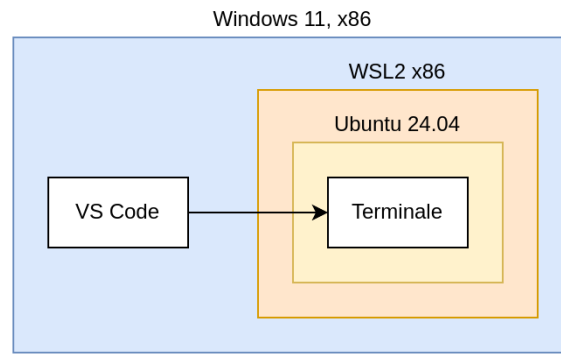


Figura 3.1: Schema dell'ambiente usato all'esame.

Il pacchetto dell'ambiente contiene le istruzioni passo passo per installare e configurare la macchina virtuale su una macchina Windows 11 con architettura x86.

Per l'ambiente Verilog, invece, ci sono sia installer precompilati [qui](#) che codice sorgente [qui](#) e [qui](#).

## 3.6 Ambiente per Linux nativo o devcontainer

### Download

Questo pacchetto supporta due scenari: una macchina con Linux x64, oppure [devcontainers](#) tramite Docker. Il pacchetto contiene le cartelle `assembler` e `verilog` con i file necessari per scrivere codice.

### 3.6.1 Utilizzo nativo

Per assembler, l'ambiente Linux deve essere in grado di

- Eseguire gli script `powershell` dell'ambiente
- Assemblare, usando `gcc`, programmi x86 scritti con sintassi GAS
- Eseguire programmi x86
- Debuggarli usando `gdb`

Per far questo su Ubuntu 24.04, i pacchetti da installare sono

- `build-essential`
- `gcc-multilib`
- `gdb`
- `powershell` ([guida](#))

Per Verilog, l'ambiente Linux deve essere in grado di

- Compilare simulazioni con `iverilog`
- Eseguire simulazioni con `vvp`
- Visualizzare waveform con `gtkwave`

Per far questo su Ubuntu 24.04, i pacchetti da installare sono

- `iverilog`
- `gtkwave`

#### Altro software per installazioni minime

Script e istruzioni si basano anche su due altri programmi: `wget` e `file`. Di solito sono inclusi di default per installazioni Desktop, ma su installazioni minime (come l'immagine Docker di Ubuntu 24.04) vanno installati

manualmente.

Una volta installato il software richiesto, per sviluppare basterà aprire le cartelle con VS Code.

### 3.6.2 Utilizzo tramite devcontainer

I [devcontainer](#) sono un'altra forma di virtualizzazione integrata in VS Code, basata su Docker anziché WSL. Il pacchetto include, nelle cartelle `.devcontainer`, i `Dockerfile` che installano il software necessario su immagini Ubuntu 24.04.

Una volta aperta la cartella con VS Code, usare il comando “Riapri in devcontainer”.

## 3.7 Alternative fai da te

Un'altra opzione molto utile di VS Code è lo sviluppo remoto tramite [SSH](#) usando [questa estensione](#). In questo caso, invece di collegarsi a un ambiente di sviluppo virtualizzato, questo risiede su un'altra macchina a cui ci si collega aprendo un terminale SSH.

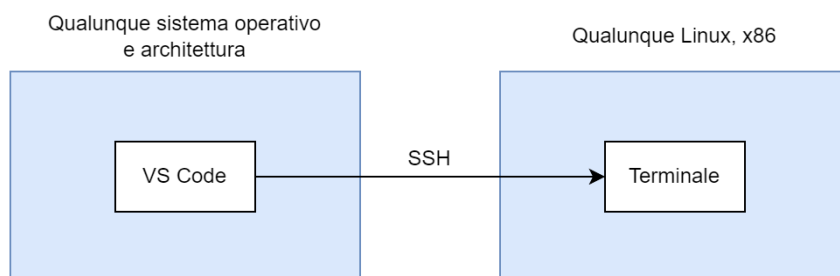


Figura 3.2: Schema di un ambiente che usa SSH.

Da notare che le macchine sono distinte “concettualmente”: niente ci vieta di avere una macchina virtuale (e.g. VirtualBox) al posto di una macchina fisicamente distinta.

## 3.8 Testare gli ambienti

I pacchetti includono dei file per *testare* che l'ambiente sia utilizzabile.

### 3.8.1 Assembler

Il file `test-ambiente.s` contiene il codice di un semplice programma che si limita a stampare `Ok..`. Provare ad assemblarlo, eseguirlo e debuggarlo.

```
PS /workspaces/assembler> ./assemble.ps1 ./test-ambiente.s
PS /workspaces/assembler> ./test-ambiente
Ok.
PS /workspaces/assembler> ./debug.ps1 ./test-ambiente
GNU gdb (Ubuntu 15.0.50.20240403-0ubuntu1) 15.0.50.20240403-git
[output di poca utilità]
Breakpoint 1, _main () at /workspaces/assembler/test-ambiente.s:7
7      _main:  nop
(gdb) qq
PS /workspaces/assembler>
```

### 3.8.2 Verilog

Il file `test-ambiente.v` contiene il codice di una semplice testbench con registro da 1 bit che cambia valore, e stampa `Ok.` a terminale prima di terminare. Provare a compilare ed eseguire la simulazione, e poi osservarne la waveform.

```
PS /workspaces/verilog> iverilog -o sim ./test-ambiente.v
PS /workspaces/verilog> vvp ./sim
VCD info: dumpfile waveform.vcd opened for output.
Ok.
./test-ambiente.v:10: $finish called at 20 (1s)
PS /workspaces/verilog> gtkwave ./waveform.vcd
[output di poca utilità]
PS /workspaces/verilog>
```

## Capitolo 4

# Essere efficienti con VS Code

VS Code è l'editor disponibile in sede d'esame e mostrato a lezione. Come ogni strumento di lavoro, è una buona idea imparare ad usarlo bene per essere più rapidi ed efficaci. Questo si traduce, in genere, nel prendere l'abitudine di usare meno il mouse e più la tastiera, usando le dovute scorciatoie e combinazioni di tasti.

In questa documentazione ci focalizziamo sulle combinazioni per Windows, che sono quelle che troverete all'esame. Evidenzierò con una ☆ le combinazioni più importanti e probabilmente meno note.

### Salvare i file

Fra le cause dei vari errori per cui riceviamo richieste d'aiuto, una delle più frequenti è che i file modificati non sono stati salvati. Un file modificato ma non salvato è indicato da un pallino nero nella tab in alto, e le modifiche non saranno visibili a altri programmi come gcc e iverilog.

Si consiglia di salvare spesso e abitualmente, usando `ctrl + s`.

## 4.1 Le basi elementari


Quando si scrive in un editor, il testo finisce dove sta il cursore (in inglese *caret*). È la barra verticale che indica dove stiamo scrivendo. Si può spostare usando le frecce, non solo destra e sinistra ma anche su e giù. Usando font monospace, infatti, il testo è una matrice di celle delle stesse dimensioni, ed è facile prevedere dove andrà il caret anche mentre ci si sposta tra le righe.

Vediamo quindi le combinazioni più comuni.

	Tasti	Cosa fa
	Tenere premuto shift	Seleziona il testo seguendo il movimento del cursore.
	<code>ctrl + c</code>	Copia il testo selezionato.
	<code>ctrl + v</code>	Incolla il testo selezionato.
	<code>ctrl + x</code>	Taglia (cioè copia e cancella) il testo selezionato.
	<code>ctrl + f</code>	Cerca all'interno del file.
	<code>ctrl + h</code>	Cerca e sostituisce all'interno del file.
☆	<code>ctrl + s</code>	Salva il file corrente.
	<code>ctrl + shift + p</code>	Apri la <i>Command Palette</i> di VS Code.

## 4.2 Le basi un po' meno elementari

Si può spostare il cursore in modo ben più rapido che un carattere alla volta.

	Tasti	Cosa fa
☆	<code>ctrl + freccia sx o dx</code>	Sposta il cursore di un <i>token</i> (in genere una parola, ma dipende dal contesto).
	<code>home</code> (inizio in italiano, più spesso  )	Sposta il cursore all'inizio della riga.
	<code>end</code> (fine in italiano)	Sposta il cursore alla fine della riga.
	<code>ctrl + shift + f</code>	Cerca all'interno della cartella/progetto/...
	<code>ctrl + shift + h</code>	Cerca e sostituisce all'interno della cartella/progetto/...
	<code>alt + freccia su/giù</code>	Sposta la riga corrente (o le righe selezionate) verso l'alto/basso.
☆	<code>ctrl + alt + freccia su/giù</code>	Copia la riga corrente (o le righe selezionate) verso l'alto/basso.

## 4.3 Editing multi-caret

Normalmente c'è *un* cursore, e ogni modifica fatta viene applicata dov'è quel *singolo* cursore.

Negli esempi che seguono, userò `|` per indicare un cursore, e coppie di `_` come delimitatori del testo selezionato.

```
Contenu|to dell'editor
```

Premendo A

```
ContenuA|to dell'editor
```

L'idea del multi-caret è di avere più di un cursore, per modificare più punti del testo allo stesso tempo. Questo è utile se abbiamo più punti del testo con uno stesso *pattern*.

	Tasti	Cosa fa
☆	ctrl + d	Aggiunge un cursore alla fine della prossima occorrenza del testo selezionato.
	esc	Ritorno alla modalità con singolo cursore.

Vediamo un esempio.

```
Prima |riga dell'editor
Seconda riga dell'editor
Terza riga dell'editor
```

Si comincia selezionando del testo.

```
Prima _riga_| dell'editor
Seconda riga dell'editor
Terza riga dell'editor
```

Usiamo ora `ctrl + d` per mettere un nuovo caret dopo la prossima occorrenza di "riga".

```
Prima _riga_| dell'editor
Seconda _riga_| dell'editor
Terza riga dell'editor
```

Abbiamo ora due caret e se facciamo una modifica verrà fatta in tutti e due i punti. Premendo per esempio `e`, andremo a sovrascrivere la parola "riga" in entrambi i punti.

```
Prima e| dell'editor
Seconda e| dell'editor
Terza riga dell'editor
```

Entrambi i cursori seguiranno indipendentemente anche gli altri comandi: movimento per caratteri, movimento per token, selezione, copia e incolla.

Per sfruttare questo, conviene scrivere codice secondo pattern in modo da facilitare questo tipo di modifiche. Per esempio, è utile avere cose che vorremmo poi modificare contemporaneamente su righe diverse, in modo da sfruttare `home` e `end` in modalità multi-cursore.

Vedremo in particolare come la sintesi di reti sincronizzate diventa molto più semplice se si sfrutta appieno l'editor.

## **Parte I**

# **Assembler - Introduzione**

## Capitolo 5

# Ambiente di sviluppo

In questo corso, programmeremo assembler per architettura x86, a 32 bit. Useremo la sintassi GAS (anche nota come AT&T), usando la linea di comando in un sistema Linux. Utilizzeremo degli script appositi per assemblare, testare e debuggare. Questi script non fanno che chiamare, semplificandone l'uso, `gcc` e `gdb`. Per istruzioni per installare e configurare il proprio ambiente, vedere [qui](#). Qui vedremo più da vicino il sistema utilizzato all'esame, basato su Windows 11 + WSL.

### Informazione a rischio aggiornamento

Fino all'A.A. 2024/25, nei laboratori si è utilizzato Windows + WSL come spiegato qui. Ciò è ancora da confermare per l'A.A. 2025/26. Ogni eventuale cambiamento sarà a impatto pratico minimo.

## 5.1 Struttura dell'ambiente

I programmi che scriveremo ed eseguiremo, così come quelli utilizzati per assemblare, gireranno in un terminale Linux.

Nell'ambiente d'esame, si usa un Ubuntu 24.04 virtualizzato tramite [WSL](#) su macchina Windows 11. Come editor usiamo [Visual Studio Code](#) con l' [estensione per lo sviluppo in WSL](#).

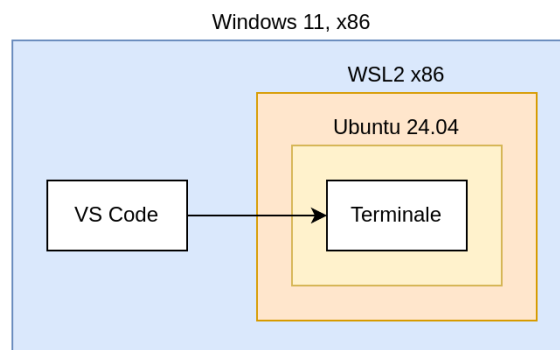
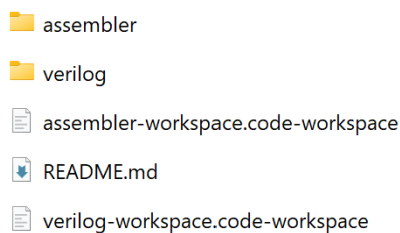


Figura 5.1: Schema dell'ambiente usato all'esame.

Questo ci permette di mantenere un ambiente grafico moderno mentre si lavora con un terminale Linux virtualizzato.

## 5.2 Lanciare l'ambiente e primo programma

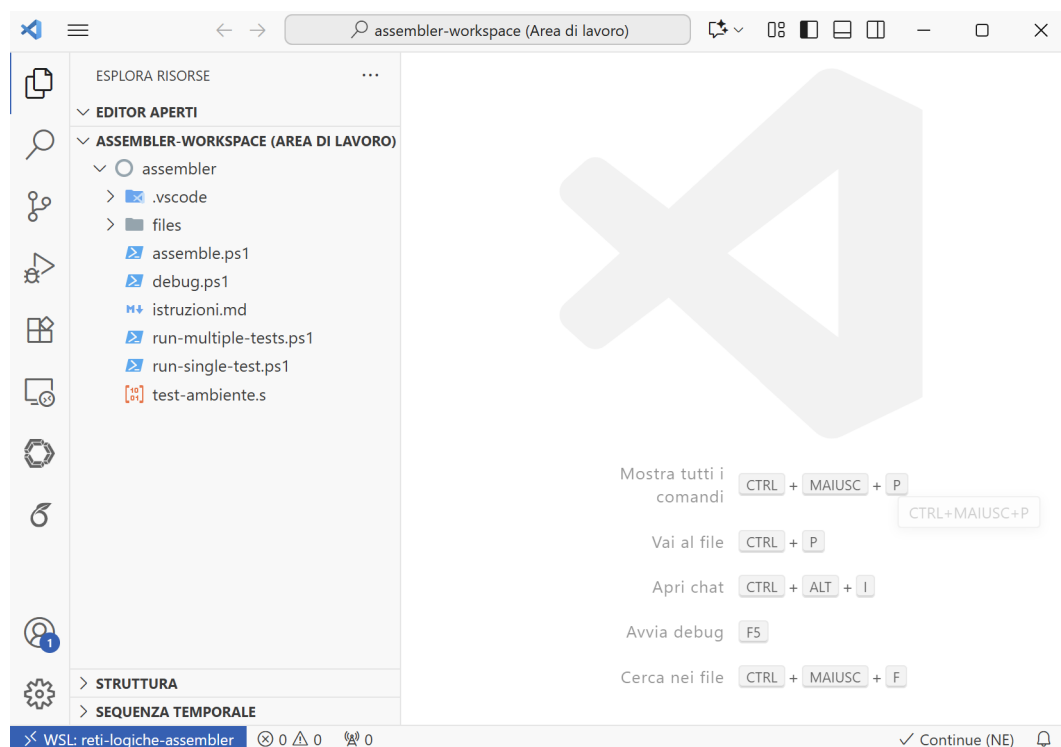
Una volta eseguiti i passi dell'installazione, avremo una cartella `C:/reti_logiche` con contenuto come da figura.



Il file `assembler-workspace.code-workspace` lancerà VS Code, collegandosi alla macchina virtuale WSL e la cartella di lavoro `C:/reti_logiche/assembler`.

Questo file è configurato per l'ambiente Windows + WSL, per automatizzare l'avvio. Se si usa un ambiente diverso, il file andrà modificato di conseguenza.

La finestra VS Code che si aprirà sarà simile alla seguente.



Nell'angolo in basso a sinistra, `WSL: reti-logiche-assembler` sta a indicare che l'editor è correttamente connesso alla macchina virtuale (compare una dicitura simile se si usa SSH).

I file e cartelle mostrati nell'immagine sono quelli che ci si deve aspettare dall'ambiente vuoto.

Il file `test-ambiente.s` è un semplice programma per verificare che l'ambiente funzioni.

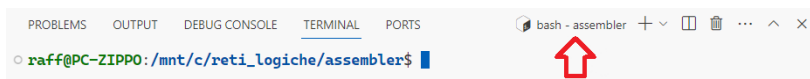
```
.include "./files/utility.s"
```

```
.data
messaggio: .ascii "Ok.\r"
```

```
.text
_main:
    nop
    lea messaggio, %ebx
    call outline
    ret
```

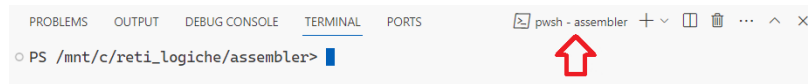
Apriamo quindi un terminale in VS Code (Terminale > Nuovo Terminale). Per poter lanciare gli script, il terminale deve essere Powershell, non bash.

Non così:





Ma così:



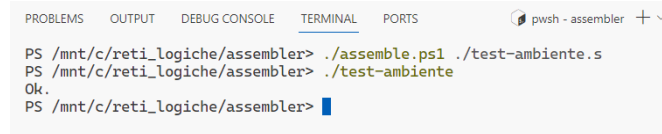
Per cambiare shell si può usare il bottone + sulla destra, o lanciare il comando pwsh senza argomenti.

Se si preferisce, in VS Code si può aprire un terminale anche come tab dell'editor, o spostandolo al lato anziché in basso.

A questo punto possiamo lanciare il comando per assemblare il programma di test.

```
./assemble.ps1 ./test-ambiente.s
```

Dovremmo adesso vedere, tra i file, il binario test-ambiente. Lo possiamo eseguire con ./test-ambiente, che dovrebbe stampare Ok..



## **Parte II**

# **Assembler - Esercitazioni**

# Capitolo 6

## Esercitazione 1

La caratteristica principale del programmare in assembler è che le operazioni a disposizione sono solo quelle messe a disposizione dal processore. Infatti, l'assemblatore fa molto poco: dopo aver sostituito le varie label con indirizzi, traduce ciascuna istruzione, nell'ordine in cui sono presenti, nel diretto corrispondente binario (il cosiddetto linguaggio macchina). Questo binario è poi eseguito direttamente dal processore. Dato un algoritmo per risolvere un problema, i passi base di questo algoritmo *devono* essere istruzioni comprese dal processore, e siamo quindi limitati dall'hardware e le sue caratteristiche.

Per esempio, dato che il processore non supporta `mov` da un indirizzo di memoria a un altro indirizzo di memoria, non possiamo fare questa operazione con una sola istruzione: dobbiamo invece scomporre in `mov src, %eax` `mov %eax, dest`, assicurandoci nel frattempo di non aver perso alcun dato importante prima contenuto in `%eax`.

Per svolgere gli esercizi, bisogna quindi imparare a scomporre strutture di programmazione già note (come if-then-else, cicli, accesso a vettore) nelle operazioni elementari messe ad disposizione dal processore, usando il limitato numero di registri a disposizione al posto di variabili, e tenendo presente quali operazioni da fare con quali dati, senza un sistema di tipizzazione ad aiutarci.

### 6.1 Premesse per programmi nell'ambiente del corso

Unica eccezione alla logica di cui sopra sono i sottoprogrammi di ingresso/uscita, forniti tramite `utility.s`: questi interagiscono con il terminale tramite il *kernel* usando il meccanismo delle *interruzioni*, concetti che avrete il tempo di esplorare in corsi successivi. Qui ci limiteremo a seguirne le specifiche, documentate [qui](#), per leggere o stampare a video numeri, caratteri, o stringhe. Per esempio, parte di queste specifiche è l'uso del carattere di ritorno carrello `\r` come terminatore di stringa. Per usarli, però, va istruito l'assemblatore di aggiungere questi sottoprogrammi al nostro codice, con

```
.include "../files/utility.s"
```

Un altro aspetto importante è dove comincia e finisce il nostro programma: *nell'ambiente del corso*, il punto di ingresso è la label `_main` e quello di uscita è la corrispondente istruzione `ret`. Per motivi di debugging, che saranno chiari più avanti, si tende a cominciare il programma con una istruzione `nop`.

Inoltre, la distinzione tra zona `.data` e `.text` è importante. Dato che durante l'esecuzione sono *entrambi* caricati in memoria, per motivi di sicurezza il kernel Linux ci impedirà di *eseguire* indirizzi in `.data` o di *scrivere* in indirizzi in `.text`. Dimenticarsi di dichiararli porta a eccezioni durante l'esecuzione.

Infine, l'assemblatore non vede di buon occhio la mancanza di una riga vuota alla fine del file. Per evitare messaggi di warning inutili, meglio aggiungerla.

Detto ciò, possiamo quindi comprendere il [programma di test](#), che non fa che stampare "Ok." a terminale e poi termina:

```
.include "../files/utility.s"
```

```
.data
messaggio: .ascii "Ok.\r"
```

```
.text
_main:
    nop
    lea messaggio, %ebx
    call outline
    ret
```

**Queste premesse si applicano solo a questo corso**

Le istruzioni di questa sezione sono relative all'ambiente del corso. La direttiva `.include "../files/utility.s"` ricopia il codice del file `utility.s`, fornito nell'ambiente del corso. Le specifiche dei sottoprogrammi (uso dei registri, `\r` come carattere di terminazione, etc.) sono conseguenza di come è scritto questo codice, che ha a che fare con scelte fatte da noi, per esempio per mantenere la retrocompatibilità con il vecchio ambiente DOS utilizzato precedentemente sempre in questo corso. L'uso di `_main` e `ret` (peraltro, senza alcun valore di ritorno), così come il comportamento del terminale, sono anche questi relativi all'ambiente usato.

Non sono assolutamente concetti validi in generale, per altri assembler e altri ambienti. Tenete questo ben presente nel caso vi avvicinaste allo sviluppo di assembler in altri contesti.

## 6.2 Esercizio 1.1

Partiamo da un esercizio con le seguenti specifiche

1. Leggere messaggio da terminale.
2. Convertire le lettere minuscole in maiuscolo.
3. Stampare messaggio modificato.

Per i passi 1 e 3 possiamo usare i sottoprogrammi di `utility` `inline` e `outline` ( [documentazione](#) ). Cominciamo riservando in memoria, nella sezione `data`, spazio per le due stringhe.

```
.data
```

```
msg_in: .fill 80, 1, 0
msg_out: .fill 80, 1, 0
```

Per la lettura useremo

```
mov $80, %cx
lea msg_in, %ebx
call inline
```

Per la scrittura invece useremo

```
lea msg_out, %ebx
call outline
```

Quel che manca ora è il punto 2. Dobbiamo (capire come) fare diverse cose:

- ricopiare `msg_in` in `msg_out` carattere per carattere
- controllare tale carattere, per capire se è una lettera minuscola
- se sì, cambiare tale carattere nella corrispondente maiuscola

Cominciamo dal capire il primo punto, cioè come ricopiare il messaggio, ignorando per ora la gestione dei caratteri minuscoli.

Come scorrere i due vettori? Abbiamo due opzioni: usare un indice per accesso indicizzato, o due puntatori da incrementare. Anche sulla condizione di terminazione abbiamo due opzioni: fermarsi dopo aver processato il carattere di ritorno carrello `\r`, o dopo aver processato 80 caratteri.

Per questo esercizio, scegliamo la prima opzione per entrambe le scelte. Se usassimo C, scriveremmo qualcosa simile a questo:

```
char[] msg_in, msg_out;
...
int i = 0;
char c;
do c = msg_in[i]; // si può trasformare c qui msg_out[i] = c; i++; while (c != '\r')
```

In assembler, questo si può scrivere così:

```
lea msg_in, %esi
lea msg_out, %edi
mov $0, %ecx
loop:
movb (%esi, %ecx), %al
# si può trasformare %al qui
movb %al, (%edi, %ecx)
inc %ecx
cmp $0x0d, %al # $0x0d è equivalente a $'\r'
jne loop
```

Ci sono diversi aspetti da sottolineare. Il primo è che nell'accesso con indice, a differenza del C, abbiamo completo controllo sia di come è calcolato l'indirizzo di accesso, sia sulla dimensione della lettura in memoria.

Prendiamo il caso di `movb (%esi, %ecx), %al`. Ricordiamo che il formato dell'indirizzazione con indice è `offset(%base, %indice, %scala)`, dove l'indirizzo è calcolato come `offset + %base + (%indice * scala)`. Dunque `(%esi, %ecx)` è, implicitamente, `0(%esi, %ecx, 1)`, dove l'1 indica il fatto che ci spostiamo di un byte alla volta. Dato l'indirizzo, però, in abbiamo controllo di quanti byte leggere, questa volta tramite il suffisso `b` o, implicitamente, tramite la dimensione del registro di destinazione `%al`.

In C, tutti questi aspetti sono gestiti automaticamente come conseguenza dell'uso del tipo `char`, che è appunto di 1 byte. In assembler, dobbiamo starci attenti noi. Infatti, il processore esegue le istruzioni senza alcun controllo (né cognizione) su che tipo di dato stiamo cercando di accedere e dove.

Prima di passare al resto del punto 2, vale la pena provare a comporre il programma così com'è, testarlo ed eseguirlo. Infatti, è sempre una buona idea trovare i bug quanto prima, e quanto più è semplice il codice scritto tanto più lo è trovare la fonte del bug. Il codice scritto finora è scaricabile [qui](#), e questo è l'output che otteniamo provando ad assemblarlo ed eseguirlo.

```
PS /mnt/c/reti_logiche/assembler> ./assemble.ps1 ./esercitazioni/1/maiusc_p1.s
PS /mnt/c/reti_logiche/assembler> ./esercitazioni/1/maiusc_p1
questo E' UN test
questo E' UN test
PS /mnt/c/reti_logiche/assembler>
```

Passiamo adesso ai punti ignorati prima, ossia controllare che il carattere letto sia una minuscola, e nel caso cambiarla in maiuscola. Per controllare che un carattere sia una lettera minuscola, ci basta ricordare che i caratteri ASCII hanno una codifica binaria ordinata: `char c` è minuscola se `c >= 'a' && c <= 'z'`.

Per cambiare invece una minuscola e maiuscola, notiamo sempre dalla tabella ASCII che, per lo stesso motivo, la distanza tra 'a' e 'A' è la stessa di qualunque altra coppia di maiuscola-minuscola. Tale distanza è 32: ci basta infatti sottrarre 32 a una minuscola per ottenere la corrispondente maiuscola, e aggiungere 32 per fare il contrario. Guardando alla rappresentazione in base 2, notiamo che l'operazione è (non per caso) ancora più semplice: essendo  $32 = 2^5$ , si tratta di mettere il bit in posizione 5 a 0 o 1, usando `and`, `or` o `xor` con maschere appropriate.

Detto ciò, il codice C diventa:

```
char[] msg_in, msg_out;
...
int i = 0;
char c;
do c = msg_in[i]; if(c >= 'a' && c <= 'z') c = c & 0xdf; msg_out[i] = c; i++; while (c != '\r')
```

La notazione esadecimale `0xdf` corrisponde a `1101 1111`. Fare un `and` con tale maschera lascia tutti i bit invariati tranne quello in posizione 5, che viene resettato. Per esempio

```
0x63 0110 0011 'c'
AND 0xdf 1101 1111
-----
= 0x43 0101 0011 'C'
```

### Il controllo non è opzionale

Domanda: se vogliamo che tutte le lettere siano maiuscole, non basta resettare il bit 5 a prescindere, e non fare il controllo?

Risposta: no, perché ci sono altri caratteri ASCII con il bit 5 a 1 che non sono affatto lettere. Per esempio, il carattere spazio di codifica `0x20`.

Questo si traduce nel seguente assembler:

```
lea msg_in, %esi
lea msg_out, %edi
mov $0, %ecx
loop:
  movb (%esi, %ecx), %al
  cmp $'a', %al
  jb post_check
  cmp $'z', %al
  ja post_check

  and $0xdf, %al      # 1101 1111 -> l'and resetta il bit 5

post_check:
  movb %al, (%edi, %ecx)
  inc %ecx
  cmp $0x0d, %al
  jne loop
```

Notiamo che le due condizioni nell'if vanno rimaneggiate per essere tradotte da C ad assembler, infatti saltiamo a `post_check`, dopo l'istruzione di conversione, se le condizioni *non* sono verificate. Il codice finale è quindi il seguente, scaricabile [qui](#) come file sorgente.

```
.include "./files/utility.s"

.data
msg_in: .fill 80, 1, 0
msg_out: .fill 80, 1, 0

.text
_main:
    nop
punto_1:
    mov $80, %cx
    lea msg_in, %ebx
    call inline
    nop
punto_2:
    lea msg_in, %esi
    lea msg_out, %edi
    mov $0, %ecx
loop:
    movb (%esi, %ecx), %al
    cmp $'a', %al
    jb post_check
    cmp $'z', %al
    ja post_check
    and $0xdf, %al
post_check:
    movb %al, (%edi, %ecx)
    inc %ecx
    cmp $0x0d, %al
    jne loop
punto_3:
    lea msg_out, %ebx
    call outline
    nop
fine:
    ret
```

Le label `punto_1`, `punto_2`, `punto_3` e `fine` sono, come è facile verificare, del tutto opzionali. Sono però utili ai fini del debugging, che presentiamo ora.

Sono da notare le `nop` aggiunte prima tra le `call` alle righe 13 e 33 e le successive label: queste sono un workaround per ovviare a un problema di `gdb`, che spiegherò più avanti.

## 6.3 Uso del debugger

*Debugging is like being the detective in a crime movie where you are also the murderer.*

Filipe Fortes

La parola *debugger* suggerisce da sé che sia uno strumento per rimuovere bug ma, purtroppo, questo non vuol dire che lo strumento li rimuove da solo. Infatti, quello in cui ci è utile il debugger è *trovare* i bug, seguendo l'esecuzione del programma passo passo e controllando il suo stato per capire dov'è che il suo comportamento differisce da quanto ci aspettiamo. Da lì, spesso indagando a ritroso e con un po' di intuito, si può trovare le istruzioni incriminate e correggerle.

### Uno strumento per essere più efficienti

Domanda: sembra complicato, non è più facile rileggere il codice?

Risposta: sì, lo è. Ma, in genere, quando basta rileggere è perché si è fatto un errore di digitazione, non di ragionamento. Saper usare il debugger significa sapersi tirare fuori *velocemente* da errori che richiederebbero rileggere *a fondo* tutto il codice.

Il debugger che usiamo è `gdb`, che funziona da linea di comando. Questo parte da un binario eseguibile, che verrà eseguito passo passo come da noi indicato.

Per semplicità d'uso, l'ambiente ha uno script `debug.ps1`, da lanciare con

```
./debug.ps1 nome-eseguibile
```

Lo script fa dei controlli, tra cui assicurarsi che si sia passato *l'eseguibile* e non *il sorgente*, lancia il debugger con alcuni comandi tipici già inseriti (imposta un breakpoint a `_main` e lancia il programma), e ne definisce altri per comodità d'uso (`rr` e `qq`, per riavviare il programma o uscire senza dare conferma).

Vediamo come usarlo, lanciando il debugger sul programma realizzato nell'esercizio precedente. Dopo un sezione di presentazione del programma, abbiamo del testo del tipo

```
Breakpoint 1, _main () at /mnt/c/reti_logiche/assembler/lezioni/1/maiusc.s:9
9      nop
(gdb)
```

Un breakpoint è un punto del programma, in genere una linea di codice, dove si desidera che il debugger fermi l'esecuzione. Avendo impostato il primo breakpoint a `_main`, vediamo infatti che il programma si ferma alla prima istruzione relativa, che è appunto la `nop`. Importante: il debugger si ferma *prima* dell'esecuzione della riga indicata. Vediamo poi che il debugger richiede input: infatti possiamo interagire con il debugger *solo* quando il programma è fermo. Possiamo fare tre cose in particolare:

- Osservare il contenuto di registri e indirizzi di memoria (`info registers` e `x`),
- Impostare nuovi breakpoints (`break`),
- Continuare l'esecuzione in modo controllato (`step` e `next`) o fino al prossimo breakpoint (`continue`)

Vediamoli in azione. Cominciamo con il proseguire fino alla riga 13.

```
Breakpoint 1, _main () at /mnt/c/reti_logiche/assembler/lezioni/1/maiusc.s:9
9      nop
(gdb) step
punto_1 () at /mnt/c/reti_logiche/assembler/lezioni/1/maiusc.s:11
11     mov $80, %cx
(gdb) s
12     lea msg_in, %ebx
(gdb) s
13     call inline
(gdb)
```

Notiamo che `gdb` accetta sia comandi per esteso sia abbreviati, per esempio per `step` va bene anche `s`. Con questi 3 `step`, abbiamo eseguito le prime tre istruzioni ma *non* la `call` a riga 13. Possiamo controllare lo stato dei registri usando `info registers`, abbreviabile con `i r`.

```
(gdb) i r
eax            0x66                102
ecx            0x50                80
edx            0x2d                45
ebx            0x56559066          1448448102
esp            0xffffc06c          0xffffc06c
ebp            0xffffc078          0xffffc078
esi            0xf7fb2000          -134537216
edi            0xf7fb2000          -134537216
eip            0x5655676e          0x5655676e <punto_1+10>
eflags        0x282                [ SF IF ]
cs             0x23                35
ss             0x2b                43
ds             0x2b                43
es             0x2b                43
fs             0x0                 0
gs             0x63                99
(gdb)
```

Notare: è un caso trovare i registri già inizializzati a 0, come qui mostrato.

Questo ci dà info su diversi registri, molti dei quali non ci interessano. Possiamo specificare quali registri vogliamo, anche di dimensioni minori di 32 bit.

```
(gdb) i r cx ebx
cx            0x50                80
ebx            0x56559066          1448448102
(gdb)
```

La prossima istruzione, se lasciamo il programma eseguire, è una `call`. In questo caso, abbiamo due scelte: proseguire *nella* chiamata al sottoprogramma (andando quindi alle istruzioni di `inline`, definite in `utility.s`), od *oltre* la chiamata, andando quindi direttamente alla riga 14. Questa è la differenza fra `step` e `next`: `step` prosegue dentro i sottoprogrammi, mentre `next` prosegue finché il sottoprogramma non ritorna.

È qui però che è rilevante la presenza della `nop` aggiunta a riga 14, prima di `parte_2`. `next` infatti continua fino alla

prossima istruzione della *sezione corrente* del codice, che è in questo caso `punto_1`. Se però tale sezione termina subito dopo la `call`, e non esiste quindi una successiva istruzione nella stessa sezione, allora usando `next` il programma continuerà fino alla terminazione. Aggiungere la `nop` ovvia al problema essendo una successiva istruzione ancora parte di `punto_1`.

```
13      call inline
(gdb) n
questo e' un test
14      nop
(gdb)
```

Da notare che “questo e' un test” è proprio l'input inserito da tastiera durante l'esecuzione di `inline`. Eseguire il programma un'istruzione alla volta può risultare molto lento. Per esempio, quando vogliamo osservare cosa succede a una particolare iterazione di un loop. Per questo ci aiutano `break` e `continue`. Nell'esempio che segue, sono usati per raggiungere rapidamente la quarta iterazione.

```
(gdb) b loop
Breakpoint 2 at 0x56556785: file /mnt/c/reti_logiche/assembler/lezioni/1/maiusc.s, line 20.
(gdb) c
Continuing.

Breakpoint 2, loop () at /mnt/c/reti_logiche/assembler/lezioni/1/maiusc.s:20
20      movb (%esi, %ecx), %al
(gdb) i r ecx
ecx      0x0      0
(gdb) c
Continuing.

Breakpoint 2, loop () at /mnt/c/reti_logiche/assembler/lezioni/1/maiusc.s:20
20      movb (%esi, %ecx), %al
(gdb) i r ecx
ecx      0x1      1
(gdb) c
Continuing.

Breakpoint 2, loop () at /mnt/c/reti_logiche/assembler/lezioni/1/maiusc.s:20
20      movb (%esi, %ecx), %al
(gdb) c
Continuing.

Breakpoint 2, loop () at /mnt/c/reti_logiche/assembler/lezioni/1/maiusc.s:20
20      movb (%esi, %ecx), %al
(gdb) i r ecx
ecx      0x3      3
(gdb)
```

L'ultima operazione base da vedere è osservare valori in memoria. Il comando `x` sta per *examine memory* ma, a differenza degli altri comandi, esiste solo in forma abbreviata. Il comando ha 4 argomenti:

- `N`, il numero di “celle” consecutive della memoria da leggere;
- `F`, il formato con cui interpretare il contenuto di tali “celle”, per esempio `d` per decimale e `c` per ASCII;
- `U`, la dimensione di ciascuna “cella”: `b` per 1 byte, `h` per 2 byte, `w` per 4 byte;
- `addr`, l'indirizzo in memoria da cui cominciare la lettura.

Il formato del comando è `x/NFU addr`. Gli argomenti `N`, `F` e `U` sono, di default, *gli ultimi utilizzati*. Questo è infatti un comando *con memoria*. Quando non sono specificati, si dovrà omettere anche lo `/`. L'argomento `addr` si può passare come

- costante esadecimale, per esempio `x 0x56559066`;
- label preceduta da `&`, per esempio `x &msg_in`;
- registro preceduto da `$`, per esempio `x $esi`;
- espressione basata su aritmetica dei puntatori, per esempio `x (int*)&msg_in+$ecx`.

L'ultima opzione è abbastanza ostica da sfruttare, vedremo come evitarla con una tecnica alternativa. Vediamo degli esempi tornando al debugging del nostro primo programma:



```
(gdb) x/20cb &msg_in
0x56559066: 113 'q' 117 'u' 101 'e' 115 's' 116 't' 111 'o' 32 ' ' 101 'e'
0x5655906e: 39 '\ ' 32 ' ' 117 'u' 110 'n' 32 ' ' 116 't' 101 'e' 115 's'
0x56559076: 116 't' 13 '\r' 10 '\n' 0 '\000'
(gdb) x/20cb &msg_out
0x565590b6: 81 'Q' 85 'U' 69 'E' 0 '\000' 0 '\000' 0 '\000' 0 '\000' 0 '\000'
0x565590be: 0 '\000' 0 '\000' 0 '\000' 0 '\000' 0 '\000' 0 '\000' 0 '\000' 0 '\000'
0x565590c6: 0 '\000' 0 '\000' 0 '\000' 0 '\000'
(gdb) x/20cb $esi
0x56559066: 113 'q' 117 'u' 101 'e' 115 's' 116 't' 111 'o' 32 ' ' 101 'e'
0x5655906e: 39 '\ ' 32 ' ' 117 'u' 110 'n' 32 ' ' 116 't' 101 'e' 115 's'
0x56559076: 116 't' 13 '\r' 10 '\n' 0 '\000'
```

In questo programma usiamo un'indirizzazione con indice per leggere e scrivere lettere nei vettori. Infatti, vediamo che il registro `esi` punta sempre alla prima lettera del vettore, e abbiamo bisogno di usare anche `ecx` per sapere qual è la lettera che il programma intende processare in questa iterazione del loop.

Per usare la sintassi menzionata sopra, dovremmo ricordarci come tradurre (`%esi`, `%ecx`) in un'espressione di aritmetica dei puntatori. Una alternativa molto agevole è invece la scomposizione dell'istruzione `movb (%esi, %ecx), %al` in due: una `lea` e una `mov`. Infatti, ricordiamo che la `lea` ci permette di calcolare un indirizzo, anche se con composto con indice, e salvarlo in un registro. Possiamo per esempio scrivere

```
lea (%esi, %ecx), %ebx
movb (%ebx), %al
```

In questo modo, l'indirizzo della lettera da leggere sarà contenuto in `ebx`, cosa che possiamo sfruttare nel debugger con il comando `x/1cb $ebx`.

Come ultime indicazioni sul debugger, menzioniamo il comando `layout regs`, che mostra a ogni passo i registri e il codice da eseguire, e i comandi `r`, per riavviare il programma e `q`, per terminare il debugger. Le versioni `qq` e `rr`, definite ad hoc nell'ambiente di questo corso, fanno lo stesso senza richiedere conferma.

## 6.4 Domande a risposta multipla

Vedremo ora delle domande a risposta multipla, riguardanti assembler e aritmetica. Prima però un disclaimer, che sembra purtroppo necessario.

### Studiare per l'esame, non l'esame

Le domande e gli esercizi dell'esame di Reti Logiche sono pensati perché, con la dovuta conoscenza e comprensione degli argomenti del programma del corso, sia agevole arrivare alla risposta/soluzione. Cioè domanda + conoscenze

È invece difficile e controproducente cercare di fare il contrario: non basta fissare domande e risposte per riuscirne a derivare conoscenze di alcun tipo. Anzi, le conoscenze necessarie sono quasi del tutto assenti dal testo delle domande e delle risposte, quelle le trovate nel *materiale di studio* del corso. Questo disclaimer è dato nella speranza di scongiurare il frequente caso di studenti che ignorano lezioni, dispense, libri di testo e ricevimenti, cercando invece di trovare autonomamente "strategie più dirette". Ripetendo poi l'esame più e più volte.

### 6.4.1 15/07/2025, domanda 9

```
mov 0x00, 0xFF
```

Nell'architettura x86 l'istruzione scritta sopra:

- copia la costante `0x00` (su 8 bit) nella cella di memoria di indirizzo `0xFF`
- copia il contenuto della cella di memoria di indirizzo `0x00` dentro la cella di indirizzo `0xFF`
- viene accettata dall'assemblatore solo se completata con un suffisso (`b`, `w`, `l`)
- nessuna delle precedenti

Per prima cosa, ricordiamo le sintassi per operandi *immediati*: con `$0x00` si rappresenta il *byte* `0x00`, mentre con solo `0x00` si rappresenta l'*indirizzo* `0x00`. Verrebbe quindi da rispondere *b*, o *c* se ci si accorge che non c'è nulla a indicare la dimensione dello spostamento.

Tuttavia il dubbio è inutile, perché la `mov` nell'architettura x86 non supporta affatto lo spostamento tra un indirizzo di memoria a un altro; serve l'istruzione `movs` per farlo (che richiede infatti di esplicitare sempre `b`, `w` o `l`). La risposta giusta è quindi la *d*.

## 6.4.2 24/06/2025, domanda 9

```

var0: .byte 0x30, 0x31
var1: .word 0x100, 0x120
var2: .long var0+3
...
mov var2, %ebx
mov (%ebx), %al

```

Alla fine del segmento di codice scritto sopra, `al` contiene

- b. `0x01`
- b. `0x20`
- b. `var0+3`
- b. Nessuna delle precedenti

Questo è un genere di esercizio che può trarre in inganno perché la risposta *non si trova affatto* sempre allo stesso modo.

Ciò che è *in comune* sono le cose che vanno sapute fare:

- ricostruire il layout in memoria dei dati, e quindi la corrispondenza tra un indirizzo e il byte corrispondente
- distinguere le varianti di operandi immediati e forme di indirizzamento
- distinguere `lea` e `mov`

A fini didattici, svolgerò per intero la ricostruzione del layout in memoria, *poi* guarderò a cosa fa il programma. All'esame, fate il contrario.

Cominciamo dalla prima riga: all'indirizzo `var0` c'è il byte `0x30`, a `var0+1` il byte `0x31`.

Passiamo quindi alla seconda riga: deriviamo innanzitutto che, se questi nuovi dati cominciano a `var1`, allora dev'essere `var1 = var0 + 2`.

In questa riga i dati sono *word*, ossia valori scritti su due byte. Ricordiamo che l'architettura è *little-endian*, ossia **little end first**: il byte meno significativo viene scritto prima.

Dunque, la word `0x0100` (il primo `0` è implicito) viene suddivisa nei due byte `0x01` e `0x00`, e salvati in memoria nell'ordine `0x00` (a `var0 + 2`) `0x01` (a `var0 + 3`). Seguono, per la stessa ragione, `0x20` (a `var0 + 4`) e `0x01` (a `var0 + 5`). Infine, alla terza riga abbiamo `var2: .long var0+3`: questo è il *valore di un indirizzo*, non il *contenuto di un indirizzo*. Questo valore verrà calcolato *poi* dall'assemblatore (o altri... è complicato) in base all'indirizzo a partire dal quale verrà allocata questa sezione `.data`. Dunque non possiamo prevederne il valore a priori, ma possiamo prevedere che punterà al byte più significativo della prima word in `var1`: `0x01`.

Ora abbiamo un'idea completa di questa sezione `.data`, e possiamo passare allo svolgimento del programma.

La prima istruzione è `mov var2, %ebx`, dove il primo operando è un *indirizzo immediato*. Quello che fa la `mov`, quindi, è copiare il *valore all'indirizzo 'var2'* nel registro `%ebx`. Dato che `%ebx` è a 32 bit, tale copia sarà a 32 bit (cioè, è implicitamente una `movl`). Dunque, `%ebx` conterrà l'indirizzo `var0+3`.

La seconda istruzione è `mov (%ebx), %al`, dove il primo operando è un *indirizzamento tramite registro*. Quello che fa la `mov`, quindi, è copiare il *valore all'indirizzo contenuto in '%ebx'* in `%al`. Dato che `%al` è a 8 bit, questa è implicitamente una `movb`. Dunque, è `0x01` che viene copiato in `%al`.

La risposta corretta è la *a*.

#### Andare dritti al punto

Per svolgere il ragionamento di sopra, e tutte le sue varianti, c'è bisogno di padroneggiare i *pochi* concetti elencati sopra.

In un vero contesto d'esame, è consigliato partire dal programma (*poche* istruzioni) e svolgere direttamente e solo i calcoli richiesti da tale programma. Per esempio, in questo esercizio è stato del tutto inutile discutere dei byte a `var0` e della seconda word di `var1`.

#### Debugger come strumento di verifica

Il modo più diretto per controllare un esercizio di questo tipo è assemblarlo e vedere con il debugger il contenuto di registri e memoria. Per esempio, con il comando `x/4xb $var1` si può verificare che i byte a partire da `var1` sono proprio quelli detti sopra.

### 6.4.3 09/09/2025, domanda 3

```
add %al, %bl
```

Dopo l'istruzione riportata sopra, quale delle seguenti configurazioni degli operandi scrive 1 dentro OF, e 1 dentro CF ?

- c.  $al = 0100\_0000, bl = 0100\_0000$
- c.  $al = 1000\_0000, bl = 1000\_0000$
- c.  $al = 1111\_1111, bl = 0000\_0001$
- c. Nessuna delle precedenti

Per rispondere, ricapitoliamo come si comportano l'istruzione `add` e i flag OF e CF.

Si parte dal fatto che la somma di numeri naturali e numeri interi in complemento alla radice (CR) eseguono *esattamente le stesse operazioni*. Quindi abbiamo una sola istruzione `add` che va usata sia che gli operandi vadano interpretati come naturali, sia che vadano interpretati come interi.

Per il processore, però, non c'è nulla che indichi se siamo nell'uno o nell'altro caso. Dunque, i flag relativi vengono popolati in ogni caso, e sta a noi controllare i flag giusti in base all'operazione svolta.

Il flag CF viene settato a 1 (0 altrimenti) se la somma, interpretata come somma fra naturali, produce riporto. In altre parole, se il risultato naturale non sta sul numero di bit previsto, in questo caso 8, che può contenere solo naturali tra 0 e 255. Questo è il caso delle risposte *b* e *c*: per *b* abbiamo  $128 + 128 = 256$ , per *c*  $255 + 1 = 256$ , dove  $256 = 1\_0000\_0000$  non sta su 8 bit. Invece, per *a* abbiamo  $64 + 64 = 128$ , ossia  $1000\_0000$ , che sta tranquillamente su 8 bit.

Il flag OF viene settato a 1 (0 altrimenti) se la somma, interpretata come somma fra interi, produce overflow. In altre parole, se il risultato intero non sta sul numero di bit previsto, in questo caso 8, che può contenere solo interi (in CR) tra -128 e +127. Questo è il caso delle risposte *a* e *b*: per *a* abbiamo  $64 + 64 = 128$ , che in CR si rappresenta con  $0\_1000\_0000$ , per *b* abbiamo  $-128 + (-128) = -256$ , che in CR si rappresenta come  $1\_0000\_000$ . Invece, per *c* abbiamo  $-1 + 1 = 0$ , ossia  $0000\_0000$ .

Dato che la *b* verifica entrambi i criteri, è la risposta giusta.

## 6.5 Esercizi per casa

Parte fondamentale delle esercitazioni è *fare pratica*. Per questo, vengono lasciati alcuni esercizi per casa. Le soluzioni di alcuni di questi saranno discusse nelle esercitazioni successive.

### 6.5.1 Esercizio 1.2: istruzioni stringa

L'esercizio 1.1 compie un'operazione ripetuta su vettori. Legge da un vettore, una cella alla volta, ne manipola il contenuto, poi lo scrive su un altro vettore. Questo genere di operazioni è adatto per l'uso delle *istruzioni stringa*. Riscrivere quindi il programma sfruttando questo set di istruzioni:

1. Leggere messaggio da terminale.
2. Convertire le lettere minuscole in maiuscolo, usando le istruzioni stringa.
3. Stampare messaggio modificato.

### 6.5.2 Esercizi 1.3 e 1.4

Scrivere dei programmi che si comportano come gli esercizi 1.1 e 1.2, tranne che per il fatto di convertire da maiuscolo in minuscolo anziché il contrario.

### 6.5.3 Esercizio 1.5

Scrivere un programma che, a partire dalla sezione `.data` che segue (e scaricabile [qui](#)), conta e stampa il numero di occorrenze di `numero` in array.

```
.include "../files/utility.s"
```

```
.data
```

```
array:      .word 1, 256, 256, 512, 42, 2048, 1024, 1, 0
array_len:  .long 9
numero:     .word 1
```

### 6.5.4 Esercizio 1.6

Quello che segue (e scaricabile [qui](#)) è un tentativo di soluzione dell'esercizio precedente. Contiene tuttavia uno o più bug. Trovarli e correggerli.

```
.include "./files/utility.s"

.data
array:      .word 1, 256, 256, 512, 42, 2048, 1024, 1, 0
array_len:  .long 9
numero:     .word 1

.text

_main:
    nop
    mov $0, %cl
    mov numero, %ax
    mov $0, %esi

comp:
    cmp array_len, %esi
    je fine
    cmpw array(%esi), %ax
    jne poi
    inc %cl

poi:
    inc %esi
    jmp comp

fine:
    mov %cl, %al
    call outdecimal_byte
    ret
```

# Capitolo 7

## Esercitazione 2

### 7.1 Esercizio 2.1: esercizio d'esame 2025-09-09

Vediamo ora un esercizio d'esame, del 09 Settembre 2025 (l'ultimo appello). Il testo con soluzione si trova [qui](#).

#### Provare da sé

Provare a svolgere da sé l'esercizio, prima di guardare la soluzione o andare oltre per la discussione.

L'esercizio ci chiede di

- Leggere un numero di 4 cifre in base 7
- Stamparlo in notazione decimale (base 10)
- Testare e stampare se è divisibile per 64, senza usare `div`

In particolare, ci è chiesto di risolvere il primo punto scrivendo due sottoprogrammi:

- `indigit_b7`, per la lettura di una cifra in base 7,
- `innumber_b7`, per la lettura di un numero a 4 cifre in base 7.

Per entrambi, dovremo gestire l'input ignorando caratteri inattesi: cioè, il programma deve comportarsi come se non fosse stato premuto niente, non stampando nulla e restando in attesa di un carattere corretto (in questo caso, un numero da 0 a 6).

#### 7.1.1 Richiami su sottoprogrammi

Partiamo da cosa significa scrivere un sottoprogramma. Un sottoprogramma è un blocco di istruzioni riutilizzabile: si entra nel sottoprogramma con una `call` e, alla fine di questo, tramite `ret` si ritorna al chiamante riprendendo dall'istruzione successiva alla `call`. È infatti questo il meccanismo che sfruttiamo quando utilizziamo i sottoprogrammi di utility, come nello snippet che segue.

```
...
mov $'h', %ah
mov $'l', %al
call outchar # chiamata a sottoprogramma
cmp $'h', %ah
je ok
...
```

Il sottoprogramma `outchar` ci dice, nella sua documentazione, che si occupa di stampare il carattere che trova in `%al`, in questo caso `l`.

Parte di ciò che rende un sottoprogramma *utile* è che faccia quello che dice, e non altro: per esempio, da questa lettura della documentazione ci aspettiamo che il contenuto di `%ah` non sia modificato, e che quindi la `je` avrà sempre successo.

Elenchiamo quindi gli aspetti principali di un sottoprogramma:

1. Ci si entra con una `call`, si esce con una `ret`;
1. Ha input e output (registri, memoria, I/O) chiaramente documentati;
1. Non modifica alcun registro, locazione di memoria o I/O al di fuori quanto documentato.

Quando si viola il terzo punto si parla di effetti collaterali, che è un errore.

## 7.1.2 indigit\_b7

Cominciamo quindi a delineare la nostra `indigit_b7` partendo da la sua struttura e documentazione:

```
# Legge e fa eco, ignorando caratteri inattesi, di una cifra in base 7
# Ne lascia il valore in %al
indigit_b7:
    ...
    ret
```

Guardiamo ora al requisito di ignorare caratteri inattesi: il sottoprogramma dovrebbe leggere un carattere e controllare se “va bene”, se sì lo stampa e altrimenti torna a leggere un altro carattere. Questo non è che un ciclo.

Per quanto riguarda il criterio, si tratta di fare un confronto tra caratteri ASCII, dato che i valori sono consecutivi: tutte le cifre tra 0 e 6 sono, nella [tabella ASCII](#), tra i valori 0x30 e 0x36.

Riassumiamo vedendo come si farebbe in pseudo-C.

```
bool continua = true;
char c;
while(continua)    c = inchar(); // legge un carattere senza farne eco  if(c < '0' || c > '6')    continua = true;
outchar(c); // stampa il carattere letto
```

Traducendo questo loop in assembler, otteniamo:

```
# Legge e fa eco, ignorando caratteri inattesi, di una cifra in base 7
# Ne lascia il valore in %al
indigit_b7:
    # ...
indigit_b7_loop:
    call inchar
    cmp '$'0'', %al
    jb indigit_b7_loop
    cmp '$'6'', %al
    ja indigit_b7_loop
    # arrivati qui, il carattere è ok
    call outchar
    # ...
    ret
```

Adesso abbiamo un carattere tra 0 e 6, dobbiamo convertirlo in un *valore* tra 0 e 6. Ricordiamo infatti che il valore di un carattere ASCII è un byte generalmente diverso da quello che rappresenta, cioè '\$'0' non è uguale a \$0.

Per convertire da uno all'altro, ricordiamo che le rappresentazioni dei caratteri sono ordinate, e quindi possiamo ottenere un indice per *differenza* : per esempio '1' - '0' = 1. Aggiungendo questa sottrazione, otteniamo:

```
# Legge e fa eco, ignorando caratteri inattesi, di una cifra in base 7
# Ne lascia il valore in %al
indigit_b7:
    # ...
indigit_b7_loop:
    call inchar
    cmp '$'0'', %al
    jb indigit_b7_loop
    cmp '$'6'', %al
    ja indigit_b7_loop
    # arrivati qui, il carattere è ok
    call outchar
    sub '$'0'', %al
    # ...
    ret
```

Quello che manca è controllare gli effetti collaterali. In questo caso non ce ne sono: le istruzioni di `indigit_b7`, così come la `inchar`, sporcano solo `%al` che è lo stesso registro che utilizziamo per l'output. `outchar` invece non modifica nessun registro. Quindi, per questo sottoprogramma, non c'è bisogno di aggiungere `push` e `pop`.

```
# Legge e fa eco, ignorando caratteri inattesi, di una cifra in base 7
# Ne lascia il valore in %al
indigit_b7:
indigit_b7_loop:
    call inchar
    cmp '$'0'', %al
    jb indigit_b7_loop
    cmp '$'6'', %al
    ja indigit_b7_loop
    # arrivati qui, il carattere è ok
    call outchar
    sub '$'0'', %al
    ret
```

**Perché usare due label per `indigit_b7` e `indigit_b7_loop`?**

Nel caso visto sopra è chiaramente una distinzione inutile. Consideriamo però un sottoprogramma leggermente diverso, che usi un altro registro come output, per esempio `%bl`, senza distinguere le due label. L'usare un altro registro significa che il valore di `%al` va preservato, usando `push` e `pop`.

```
# Legge e fa eco, ignorando caratteri inattesi, di una cifra in base 7
# Ne lascia il valore in %bl
indigit_b7:
push %ax      # push/pop sono solo da 32 o 16 bit, non 8
call inchar
cmp '$0', %al
jb indigit_b7
cmp '$6', %al
ja indigit_b7
# arrivati qui, il carattere è ok
call outchar
sub '$0', %al
pop %ax
ret
```

Questo codice ha un bug: con questa struttura, facciamo una nuova `push` ogni volta che si inserisce un carattere non riconosciuto. Però, in fondo, si ha solo una `pop`. Questo significa che si arriva alla `ret` con lo stack sporco: questo causa crash del programma (se va bene e non si trovano istruzioni nel punto a caso in cui salta).

È quindi una buona regola, per evitare errori difficili da debuggare, distinguere le label di ingresso dei sottoprogrammi dalle label utilizzate per fare cicli.

Arrivati a questo punto, abbiamo il sottoprogramma `indigit_b7` che possiamo utilizzare per ottenere da terminale una cifra in base 7, e trovarne il valore (compreso tra 0 e 6) in `%al`.

Possiamo verificarne il funzionamento cominciando a scrivere il resto del programma per testarlo ( [download](#) ).

```
.include "./files/utility.s"

.data

.text
_main:
nop
call indigit_b7
call newline
call outdecimal_byte
ret

# Legge e fa eco, ignorando caratteri inattesi, di una cifra in base 7
# Ne lascia il valore in %al
indigit_b7:
indigit_b7_loop:
call inchar
cmp '$0', %al
jb indigit_b7_loop
cmp '$6', %al
ja indigit_b7_loop
# arrivati qui, il carattere è ok
call outchar
sub '$0', %al
ret
```

Eseguendo questo programma, otteniamo

```
PS /mnt/c/reti_logiche/assembler> ./assemble.ps1 ./lezioni/2/2025-09-09-p1.s
PS /mnt/c/reti_logiche/assembler> ./lezioni/2/2025-09-09-p1
5
5
PS /mnt/c/reti_logiche/assembler>
```

### 7.1.3 innumber\_b7

Passiamo ora a scrivere `innumber_b7`, per la lettura di un numero a 4 cifre in base 7. Definiamo prima cosa vogliamo che faccia. Sarebbe infatti utile che questo sottoprogramma si occupi di convertire la sequenza di 4 cifre nel numero naturale rappresentato.

Bisogna prima però chiedersi: quanto sarà grande questo naturale, quanti bit servono? Questo si chiama *fare il dimensionamento*, ed è una parte importante per tutti gli esercizi che toccano l'aritmetica. Il numero naturale più

grande che si può scrivere con 4 cifre in base 7 è  $7^4 - 1 = 2400$ . Quindi non ci basta un registro a 8 bit ( $2^8 - 1 = 255$ ), ma ce ne basta uno a 16 bit ( $2^{16} - 1 = 65535$ ).

```
# Legge e fa eco, ignorando caratteri inattesi, di un numero naturale di 4 cifre in base 7
# Ne lascia il valore in %ax
innumber_b7:
    ...
    ret
```

Siano  $b_3, b_2, b_1, b_0$  le 4 cifre in base 7, ciascuna compresa tra 0 e 6. Il numero naturale rappresentato da queste 4 cifre sarà  $b_3 \cdot 7^3 + b_2 \cdot 7^2 + b_1 \cdot 7^1 + b_0 \cdot 7^0$ . Abbiamo quindi bisogno di

- leggere le 4 cifre (possiamo usare `indigit_b7`)
- moltiplicare ciascuna cifra per una potenza di 7
- poi sommare questi valori tra di loro. Cominciamo a vedere, in pseudo-C, come potremmo fare questa cosa, ricordandoci che in assembler possiamo fare operazioni aritmetiche (`add, mul`) solo fra due operandi alla volta.

```
// per semplicità, il codice che segue non tiene in considerazione i dimensionamenti per le mul...
int b_3 = indigit_b7();
int b_2 = indigit_b7();
int b_1 = indigit_b7();
int b_0 = indigit_b7();

int p_3 = b_3 * 343; // 7*7*7
int p_2 = b_2 * 49;  // 7*7
int p_1 = b_1 * 7;
int p_0 = b_0;

int n = ((p_3 + p_2) + p_1) + p_0;
```

Questa scomposizione funziona, e potremmo effettivamente tradurla in una implementazione. C'è un vincolo importante, però: dove mettiamo tutte queste variabili intermedie? Ricordiamo che abbiamo sempre 3 opzioni:

- registri
- memoria statica (`.data`)
- stack

La prima opzione è preferibile (anche per performance) ma i registri sono limitati, e potrebbe essere difficile gestirli. La seconda opzione funziona, ma è la meno elegante: richiede che il sottoprogramma abbia il proprio spazio di memoria dedicato *sempre* allocato (equivale ad una funzione C che utilizzi variabili globali). La terza opzione è invece la migliore quando si tratta di usare la memoria in sottoprogrammi: non a caso, un compilatore C utilizza per le variabili locali proprio lo stack.

A fini didattici, vediamo prima come fare questo con lo stack. Dobbiamo prima riordinare le operazioni del nostro pseudo-C per rendere l'idea fattibile. Attenzione: possiamo riordinare i calcoli, ma non l'ordine di input, perché l'utente scriverà sempre il numero partendo dalla cifra più significativa.

### push e pop non a 8 bit

Ricordiamo che le istruzioni `push` e `pop` supportano solo operandi a 16 e 32 bit, non 8. Dobbiamo quindi estendere su almeno 16 bit per utilizzare lo stack.

```
// per semplicità, il codice che segue non tiene in considerazione i dimensionamenti per le mul...

// fase 1: calcolo prodotti e push
al = indigit_b7();
ax = al * 343;
push(ax);

al = indigit_b7();
ax = al * 49;
push(ax);

al = indigit_b7();
ax = al * 7;
push(ax);

al = indigit_b7();
ax = al;
push(ax);
```



```
// fase 2: pop e sommatoria
ax = 0;
// b_0
bx = pop();
ax += bx;
// += b_1 * 7
bx = pop();
ax += bx;
// += b_2 * 7 * 7
bx = pop();
ax += bx;
// += b_3 * 7 * 7 * 7
bx = pop();
ax += bx;
```

Per tradurre questo in assembler, dobbiamo risolvere i problemi di dimensionamento per utilizzare correttamente la `mul`. Infatti, la `mul` a 8 bit accetta operandi a 8 bit, lasciando un risultato a 16 bit in `%ax`. La `mul` a 16 bit accetta operandi a 16 bit, lasciando un risultato a 32 bit in `%dx_%ax`.

Noi però vogliamo moltiplicare, ad un certo punto, per 343, che non sta su 8 bit ( $> 255$ ). Quello che dobbiamo fare, almeno per quel passaggio, è quindi utilizzare la `mul` a 16 bit ignorando la parte alta del risultato in `%dx` (sappiamo, per dimensionamento, che sarà `0x0000`).

#### Utilizzare costanti quando possibile

In questo esercizio abbiamo potenze di 7, come  $7^3$ . Si potrebbe pensare di calcolare questo *nel* programma, con una serie di `mul`. Sarebbe però uno sforzo del tutto inutile, sia in termini di codice che di cicli del processore.

Se infatti possiamo calcolare in anticipo il risultato (la calcolatrice si può usare) è meglio scriverlo direttamente come costante nel codice, e usare i commenti per dire qual'è il calcolo che vi sta dietro.

```
# Legge e fa eco, ignorando caratteri inattesi, di un numero naturale di 4 cifre in base 7
# Ne lascia il valore in %ax
innumber_b7:
    # push dei registri sporcati
    push %bx
    push %dx    # sporcato dalla mul a 16 bit

    # fase 1: calcolo prodotti e push
    call indigit_b7
    mov $0, %ah
    mov $343, %bx
    mul %bx
    push %ax

    call indigit_b7
    mov $49, %bl
    mul %bl
    push %ax

    call indigit_b7
    mov $7, %bl
    mul %bl
    push %ax

    call indigit_b7
    mov $0, %ah
    push %ax

    # fase 2: pop e sommatoria
    mov $0, %ax
    # b_0
    pop %bx
    add %bx, %ax
    # += b_1 * 7
    pop %bx
    add %bx, %ax
    # += b_2 * 7 * 7
    pop %bx
    add %bx, %ax
    # += b_3 * 7 * 7 * 7
    pop %bx
    add %bx, %ax

    # pop dei registri sporcati
    pop %dx
```

```

pop %bx
ret

```

Questa implementazione del sottoprogramma è la migliore? No. Però funziona, cosa che è l'obiettivo principale da raggiungere.

Infatti, possiamo verificarlo con un programma di test ( [download](#) ). Notiamo che, visto che il risultato è un naturale su 16 bit, ci basterà usare `outdecimal_word` per stamparne la rappresentazione decimale.

### Versione senza stack

Il sottoprogramma scritto sopra utilizza lo stack per gestire i quattro valori intermedi da calcolare e, infine, sommare. Questa tecnica è utile in generale, soprattutto per conti più complessi che richiedono molte più istruzioni (e registri) per ciascun passaggio intermedio.

Tuttavia, è facile osservare che questo calcolo non è *così* complesso. Infatti, potremmo usare un altro registro come appoggio per calcolare la somma *mentre* leggiamo nuove cifre, senza passare per lo stack.

```

# Legge e fa eco, ignorando caratteri inattesi, di un numero naturale di 4 cifre in base 7
# Ne lascia il valore in %ax
innumber_b7:
    # push dei registri sporcati
    push %bx
    push %cx
    push %dx      # sporcato dalla mul a 16 bit

    # inizializzazione registro d'appoggio
    mov $0, %cx

    call indigit_b7
    mov $0, %ah
    mov $343, %bx
    mul %bx
    add %ax, %cx

    call indigit_b7
    mov $49, %bl
    mul %bl
    add %ax, %cx

    call indigit_b7
    mov $7, %bl
    mul %bl
    add %ax, %cx

    call indigit_b7
    mov $0, %ah
    add %ax, %cx

    # riprendiamo il risultato dal registro d'appoggio
    mov %cx, %ax

    # pop dei registri sporcati
    pop %dx
    pop %cx
    pop %bx

    ret

```

[Qui](#) il programma di test per questa versione.

### Versione con loop scalabile

Le versioni sopra hanno un grosso limite: tutti e 4 i casi per le 4 cifre vengono gestiti “a mano”. Passare a un numero  $n$  di cifre richiederebbe scrivere  $n$  blocchi di codice simili, ma con costanti diverse.

Per trovare un'alternativa *iterativa* dobbiamo partire dalla formula, capendo come trasformarla per renderla iterativa.

$$\begin{aligned}
 & b_3 \cdot 7^3 + b_2 \cdot 7^2 + b_1 \cdot 7^1 + b_0 \\
 &= (b_3 \cdot 7^2 + b_2 \cdot 7 + b_1) \cdot 7 + b_0 \\
 &= ((b_3 \cdot 7 + b_2) \cdot 7 + b_1) \cdot 7 + b_0
 \end{aligned}$$

Possiamo generalizzare questo per una qualunque base  $\beta$  e numero di cifre  $n$ . Sia  $x$  il numero naturale di  $n$  cifre in base  $\beta$ , se aggiungo una cifra a destra, sia questa  $b$ , ottengo allora il numero naturale  $x \cdot \beta + b$ .

Riflettendoci, c'è anche un modo più immediato per arrivarci: aggiungere una cifra a destra equivale a fare uno shift a sinistra (base  $\beta$ ) delle cifre che si hanno già, a cui poi sommiamo la nuova cifra.

Possiamo quindi scrivere un algoritmo iterativo che, utilizzando questa espressione, calcola man mano il numero naturale inserito dall'utente. Questo significa però che se *almeno una* delle moltiplicazioni dovrà essere a 16 bit, allora tutte le mul del ciclo dovranno esserlo.

# Legge e fa eco, ignorando caratteri inattesi, di un numero naturale di 4 cifre in base 7

# Ne lascia il valore in %ax

innumber\_b7:

push %bx

push %cx

push %dx # sporcato dalle mul a 16 bit

mov \$0, %bx

mov \$4, %cl;

innumber\_b7\_loop:

# check fine ciclo

cmp \$0, %cl

je innumber\_b7\_fine

# shift a sinistra (base 7) del numero letto finora

mov \$7, %ax

mul %bx

mov %ax, %bx

# leggo la nuova cifra e la sommo

call indigit\_b7

mov \$0, %ah

add %ax, %bx

# nuova iterazione

dec %cl

jmp innumber\_b7\_loop

innumber\_b7\_fine:

mov %bx, %ax

pop %dx

pop %cx

pop %bx

ret

Da notare che questa versione è non solo più breve, ma anche molto più scalabile. Ci basterà infatti cambiare soltanto il valore con cui inizializziamo %cl e il dimensionamento dei registri utilizzati per supportare altri valori di  $n$  diversi da 4: possiamo gestire fino a 5 cifre base 7 mantenendo il risultato su 16 bit, e fino a 11 cifre base 7 passando a 32 bit.

[Qui](#) il programma di test per questa versione.

### 7.1.4 Divisibilità per 64

Ricapitolando, con i sottoprogrammi indigit\_b7, innumber\_b7 e outdecimal\_word possiamo gestire i primi due punti dell'esercizio.

- ☒ Leggere un numero di 4 cifre in base 7
- ☒ Stamparlo in notazione decimale (base 10)
- ☐ Testare e stampare se è divisibile per 64, senza usare div

Ci resta da controllare la divisibilità per 64. Chiediamoci intanto usando la div - che ci è preclusa - come avremmo potuto fare. La divisione tra naturali fatta dalla div produce un quoziente ed un resto: potremmo verificare la divisibilità controllando che il resto della divisione per 64 sia 0.

Notiamo però che 64 non è un numero qualunque, ma equivale a  $2^6$ . In una qualunque base  $\beta$ , un numero è divisibile per  $\beta^n$  se le sue  $n$  cifre meno significative sono 0.

Dobbiamo quindi controllare che le 6 cifre meno significative del nostro numero siano 0. Il modo più efficace per farlo è usare un and con una maschera.

Ricordiamo cosa fa la and con una maschera (valore costante):

- lascia i bit dell'operando invariati in corrispondenza degli 1 nella maschera;
- forza a 0 i bit in corrispondenza degli 0 nella maschera.

Possiamo utilizzare una maschera che forzi a 0 tutti i bit che non ci interessano (nelle posizioni da 15 a 6) e lasci invariati quelli che ci interessano: il risultato sarà 0 solo se i bit che ci interessano sono a 0.

Vediamo degli esempi, per semplicità su 8 bit con  $16 = 2^4$  come divisore.

```

    0110 0000    96 = 6 * 16
AND 0000 1111
-----
=   0000 0000    0

    0110 0110    102, non divisibile per 16
AND 0000 1111
-----
=   0000 0110    != 0

```

Tornando al nostro caso, cioè 16 bit e  $64 = 2^6$  come divisore, la maschera da utilizzare sarà 0x003F.

```

punto_1:
    call innumber_b7
    call newline
punto_2:
    call outdecimal_word
    call newline
punto_3:
    and $0x003f, %ax
    jz divisibile
non_divisibile:
    mov '$0', %al
    jmp stampa
divisibile:
    mov '$1', %al
stampa:
    call outchar

```

Il programma completo (utilizzando la versione iterativa di `innumber_b7`) è scaricabile [qui](#).

## 7.2 Domande a risposta multipla

### 7.2.1 2025-01-08, domanda 3

```

var1: .word 0x1020, 0x32ab
var2: .long var1+2
var3: .byte 0x66

```

Data la dichiarazione di sopra, qual è il contenuto del byte di memoria di indirizzo `var2` ?

- a. 0xab
- a. 0x32
- a. 0x66
- a. Nessuna delle precedenti

#### Risposta

La risposta giusta è la *d*.

I quattro byte a partire da `var2` formano un indirizzo di memoria, calcolato opportunamente dall'assemblatore o chi per lui. Mentre possiamo prevedere *a cosa punti* tale indirizzo (il byte 0xAB, il meno significativo del secondo elemento di `var1`) non abbiamo modo di prevedere il *valore* di tale indirizzo.

### 7.2.2 2025-06-04, domanda 2

```

not %bx
not %ax
or  %bx,%ax
not %ax

```

Il codice sopra scritto calcola:

- b. L' and di `bx` e `ax`

- b. L'or di bx e ax
- b. Il nor di bx e ax
- b. Nessuna delle precedenti

**Risposta**

La risposta giusta è la a.

Basta ricordare il teorema di De Morgan:

$$\overline{\overline{a} + \overline{b}} = \overline{\overline{a} \cdot \overline{b}} \\ = a \cdot b$$

Questo si applica per ogni coppia di bit  $a$  e  $b$ , che siano in posizioni corrispondenti di  $ax$  e  $bx$ . Quindi, per estensione, il codice è equivalente a `and %ax, %bx`.

**7.2.3 2025-02-11, domanda 1**

```
sar %ax
sal %ax
jc dopo
```

Il codice scritto sopra salta all'etichetta dopo :

- c. Sempre
- c. Mai
- c. Se prima che si iniziasse il MSB di  $ax$  valeva 1
- c. Nessuna delle precedenti

**Risposta**

La risposta giusta è la c.

`sar` sta per *shift arithmetic right* : è uno shift aritmetico che preserva il segno dell'operando, cioè se il MSB ( *most significant bit* ) è 1 allora anche il risultato avrà 1 come MSB.

`sal` sta per *shift arithmetic left* : si comporta come lo shift logico ( `shl` ) con l'unica aggiunta di settare *anche* OF se si verifica un cambio di segno. In particolare, come lo shift logico setta CF se il MSB dell'operando è 1. Dunque, il salto `jc` ( *jump if carry* ) ha successo solo se `sal` setta CF, cosa che succede solo se `sar` setta il MSB, cosa che succede solo se l'operando originale aveva il MSB a 1.

Da notare che se la prima istruzione fosse stata uno shift logico ( `shr` ), la risposta corretta sarebbe stata *mai*.

# Capitolo 8

## Esercitazione 3

### 8.1 Soluzioni passo-passo esercizi per casa

#### 8.1.1 Esercizio 1.2: istruzioni stringa, soluzione passo-passo

Ricordiamo la traccia dell'esercizio:

1. Leggere messaggio da terminale.
2. Convertire le lettere minuscole in maiuscolo, usando le istruzioni stringa.
3. Stampare messaggio modificato.

Le istruzioni stringa sono un esempio di set di istruzioni specializzate, cioè istruzioni che non sono pensate per implementare algoritmi generici, ma sono invece pensate per fornire supporto hardware efficiente a uno specifico set di operazioni che alcuni algoritmi necessitano. Infatti, ci si può aspettare che tra due programmi equivalenti, uno scritto con sole istruzioni generali e l'altro scritto con istruzioni specializzate, il secondo sarà molto più performante del primo. Altri esempi comuni sono le istruzioni a supporto di crittografia, encoding e decoding di stream multimediali, e, più recentemente, neural networks.

Questi set di istruzioni sono però più "rigidi" delle istruzioni a uso generale. Ci impongono infatti dei modi specifici di organizzare dati e codice, perché questi devono essere compatibili con il modo in cui l'algoritmo eseguito da un'istruzione è implementato in hardware.

Nell'esercizio 1.1 abbiamo considerato due modi di scorrere i due array. Nel primo, che è quello che abbiamo scelto, si carica l'indirizzo di inizio del vettore, e si usa un altro registro come indice, usando l'indirizzazione con indice. Nel secondo, si usa un registro come puntatore alla cella corrente, inizializzato all'indirizzo di inizio del vettore e poi incrementato (della quantità giusta) per passare all'elemento successivo. In entrambi i casi, siamo liberi di usare i registri che vogliamo, per esempio non abbiamo nessun problema se scriviamo il programma di prima come segue:

```
    lea msg_in, %eax
    lea msg_out, %ebx
    mov $0, %edx
loop:
    movb (%eax, %edx), %cl
    ...
```

Infatti, usare esi ed edi come registri puntatori, ed ecx come registro di indice, è del tutto opzionale.

Tutto questo cambia quando si vogliono usare istruzioni specializzate come le istruzioni stringa. Queste ci impongono di usare esi come puntatore al vettore sorgente, edi come puntatore al vettore destinatario, eax come registro dove scrivere o da cui leggere il valore da trasferire, ecx come contatore delle ripetizioni da eseguire, etc. Una volta scelte le istruzioni da usare, dobbiamo quindi assicurarci di seguire quanto imposto dall'istruzione.

Per questo esercizio siamo interessati alla lods, che legge un valore dal vettore e ne sposta il puntatore allo step successivo, e la stos, che scrive un valore nel vettore. Partiamo dal riscrivere il punto\_2 dell'esercizio 1.1 in modo da rendere l'algoritmo compatibile.

```
...
punto_2:
    lea msg_in, %esi
    lea msg_out, %edi
loop:
    movb (%esi), %al
    inc %esi
    cmp $'a', %al
    jb post_check
    cmp $'z', %al
    ja post_check
    and $0xdf, %al
```

```

post_check:
    movb %al, (%edi)
    inc %edi
    cmp $0x0d, %al
    jne loop
...

```

Abbiamo dunque rimosso l'uso di `ecx` come indice, e usiamo `esi` ed `edi` come puntatori. Il fatto di usare la `inc` è legato alla dimensione dei dati, cioè 1 byte. Dovremmo invece scrivere `add $2, %esi` o `add $4, %esi` per dati su 2 o 4 byte. Altra nota è che *incrementiamo* i puntatori, anziché decrementarli, perché stiamo eseguendo l'operazione da sinistra verso destra.

Siamo pronti adesso a sostituire le istruzioni evidenziate con delle istruzioni stringa. Il sorgente finale è scaricabile [qui](#).

```

...
punto_2:
    lea msg_in, %esi
    lea msg_out, %edi
    // highlight-start
    cld
    // highlight-end
loop:
    lodsb
    cmp 'a', %al
    jb post_check
    cmp 'z', %al
    ja post_check
    and $0xdf, %al
post_check:
    stosb
    cmp $0x0d, %al
    jne loop
...

```

L'istruzione `cld` serve a impostare a 0 il flag di direzione, che serve a indicare alle istruzioni stringa se andare da sinistra verso destra o il contrario. Dato che tutti i registri sono impliciti, dobbiamo sempre specificare la dimensione delle istruzioni, in questo caso `b`.

Come esercizio, può essere interessante osservare con il debugger l'evoluzione dei registri, osservando come si eseguono più operazioni con una sola istruzione.

### 8.1.2 Esercizio 1.6: esercizio di debugging, soluzione passo-passo

Ricordiamo la traccia dell'esercizio:

Scrivere un programma che, a partire dalla sezione `.data` che segue (e scaricabile [qui](#)), conta e stampa il numero di occorrenze di `numero` in `array`.

```

#include "./files/utility.s"

.data
array:      .word 1, 256, 256, 512, 42, 2048, 1024, 1, 0
array_len:  .long 9
numero:     .word 1

```

Questa è invece la soluzione proposta dall'esercizio:

```

#include "./files/utility.s"

.data
array:      .word 1, 256, 256, 512, 42, 2048, 1024, 1, 0
array_len:  .long 9
numero:     .word 1

.text

_main:
    nop
    mov $0, %cl
    mov numero, %ax
    mov $0, %esi

comp:
    cmp array_len, %esi

```

```

    je fine
    cmpw array(%esi), %ax
    jne poi
    inc %cl

poi:
    inc %esi
    jmp comp

fine:
    mov %cl, %al
    call outdecimal_byte
    ret

```

Come prima cosa, cerchiamo di capire, a grandi linee, cosa cerca di fare questo programma.

Notiamo l'uso di `%cl` : dall'inizializzazione a riga 12, l'incremento condizionato a righe 19-21, e la stampa a righe 28-29, si evince che `%cl` è usato come contatore dei successi, ossia di quante volte è stato trovato `numero` in `array`. Notiamo che `%ax` viene inizializzato con `numero` e, prima della stampa, mai aggiornato. Infine, `%esi` viene inizializzato a 0 e incrementato a fine di ogni ciclo, confrontandolo con `array_len` per determinare quando uscire dal loop. Infine, a riga 19 notiamo il confronto tra un valore di `array`, indicizzato con `%esi`, e `%ax`, che contiene `numero`.

Si ricostruisce quindi questa logica: scorro valore per valore `array`, indicizzandolo con `%esi`, e lo confronto con `numero`, che è appoggiato su `%ax` (perché il confronto tra due valori in memoria non è possibile con `cmp`). Utilizzo `%cl` come contatore dei successi, e alla fine dello scorrimento ne stampo il valore.

Fin qui nessuna sorpresa, il programma sembra seguire lo schema che si seguirebbe con un normale programma in C:

```

int cl = 0;
for(int esi = 0; esi < array_len; esi++)    if(array[esi] == numero)        cl++;

```

Proviamo ad eseguire il programma: ci si aspetta che stampi 2. Invece, stampa 3. Dobbiamo passare al debugger.

Quello che ci conviene guardare è quello che succede ad ogni loop, in particolare alla riga 19, dove la `cmpw` confronta un valore di `array` con `%ax`, che contiene `numero`. Però, la `cmpw` utilizza un indirizzamento complesso che, come descritto nella [documentazione](#), richiede una sintassi più complicata nel debugger. Cambio quindi quella istruzione in una serie equivalente che sia più facile da osservare col debugger.

```

#include "./files/utility.s"

.data
array:      .word 1, 256, 256, 512, 42, 2048, 1024, 1, 0
array_len:  .long 9
numero:     .word 1

.text

_main:
    nop
    mov $0, %cl
    mov numero, %ax
    mov $0, %esi

comp:
    cmp array_len, %esi
    je fine
    movw array(%esi), %bx
    cmpw %bx, %ax
    jne poi
    inc %cl

poi:
    inc %esi
    jmp comp

fine:
    mov %cl, %al
    call outdecimal_byte
    ret

```

Assemblo, avvio il debugger, e setto un breakpoint alla riga 20 con `break 20`. Lascio girare il programma con `continue`, che quasi immediatamente raggiunge la riga 20 e si ferma. Ricordiamo che il debugger si ferma *prima* di eseguire una istruzione.

Vediamo lo stato dei registri, con i r `ax bx cl esi` (mostra solo quelli che ci interessano).



```
(gdb) i r ax bx cl esi
ax          0x1          1
bx          0x1          1
cl          0x0          0
esi         0x0          0
```

Fin qui, tutto come ci si aspetta: `%ax` che contiene numero, `%bx` contiene il numero alla prima cella di array, i due contatori `%cl` e `%esi` sono a 0. Facciamo step per vedere l'esito del confronto: dopo la riga 21 l'esecuzione giunge alla riga 22, indicando che il salto non è stato fatto perché la `jne` è stata eseguita dopo un confronto tra valori uguali. Continuiamo con step controllando che il comportamento sia quello atteso, fino a giungere di nuovo alla riga 20.

```
(gdb) i r ax bx cl esi
ax          0x1          1
bx          0x0          0
cl          0x1          1
esi         0x1          1
```

Qui abbiamo la prima sorpresa. In `%bx` troviamo 0, ma il secondo valore di array è 256. Se continuiamo, vediamo che 256 compare come terzo valore, poi 1 come quarto, poi 256 come quinto. Abbiamo quindi dei valori aggiuntivi che compaiono durante lo scorrimento del vettore ma che non sono presenti nell'allocazione a riga 4. Continuando ancora, vediamo che i 9 valori coperti dal programma non sono affatto tutti e 9 quelli a riga 4, e che effettivamente il valore 1 compare 3 volte.

Con questo, abbiamo intanto confinato il problema: la logica di confronto e conteggio funziona, il problema è nella lettura di valori da array.

Per capire cosa sta succedendo, dobbiamo ricordare come si comporta l'allocazione in memoria di valori su più byte: abbiamo infatti a che fare con *word*, composte da 2 byte ciascuna, e ciascun indirizzo in memoria corrisponde a una locazione di un solo byte.

L'architettura x86 è *little-endian*, che significa **little end first**, [un riferimento a I viaggi di Gulliver](#). Questo si traduce nel fatto che quando un valore di  $n$  byte viene salvato in memoria a partire dall'indirizzo  $a$ , il byte meno significativo del valore viene salvato in  $a$ , il secondo meno significativo in  $a + 1$ , e così via fino al più significativo in  $a + (n - 1)$ . Possiamo quindi immaginare così il nostro array in memoria.

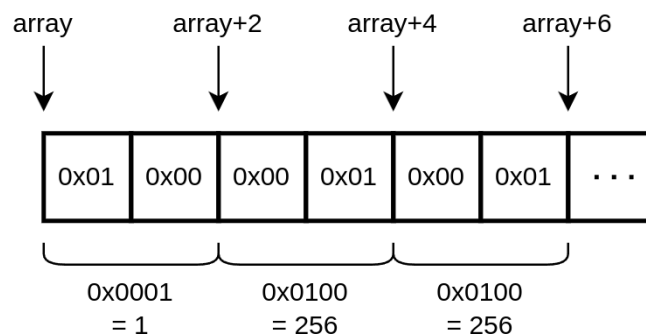


Figura 8.1: Layout di array in memoria.

La lettura di una *word* dalla memoria funziona quindi così: dato l'indirizzo  $a$ , vengono letti i due byte agli indirizzi  $a$  e  $a + 1$  e concatenati nell'ordine  $(a + 1, a)$ . Una istruzione come `movw a, %bx`, quindi, salverà il contenuto di  $a + 1$  in `%bh` e il contenuto di  $a$  in `%bl`.

Per la lettura di più *word* consecutive, dobbiamo assicurarci di incrementare l'indirizzo di 2 alla volta: come mostrato in figura, il secondo valore è memorizzato a partire da `array + 2`, il terzo da `array + 4`, e così via.

Tornando però al codice dell'esercizio, questo non succede:

```
comp:
    cmp array_len, %esi
    je fine
    movw array(%esi), %bx
    cmpw %bx, %ax
    jne poi
    inc %cl

poi:
    inc %esi
    jmp comp
```

Ecco quindi spiegato cosa legge il programma dalla memoria: quando alla seconda iterazione si esegue `movb array(%esi), %bx`, con `%esi` che vale 1, si sta leggendo un valore composto dal byte meno significativo del secondo valore concatenato

con il byte più significativo del primo. Questo valore è del tutto estraneo e privo di senso se confrontato con array così come è stato dichiarato e allocato, ma nell'eseguire le istruzioni il processore non sa e non controlla niente di tutto ciò.

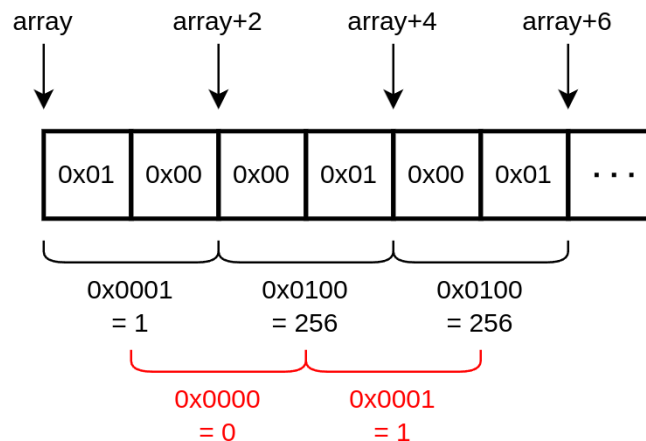


Figura 8.2: Lettura erranea di array : sbagliando l'incremento dell'indirizzo, leggiamo dei byte senza alcuna relazione fra loro dalla memoria e li interpretiamo come parti di una word.

Abbiamo due strade per correggere questo errore. Il primo approccio è quello di incrementare `%esi` di 2 alla volta, così che l'indirizzamento `array(%esi)` risulti corretto. Con questo schema però `%esi` dovrà assumere i valori 0, 2, ... 16, cosa che lo rende non più un contatore confrontabile direttamente con `array_len` come fatto a riga 17. Si dovrà gestire tale confronto in altro modo, per esempio usando un registro separato come contatore.

La seconda strada è quella di usare il fattore di *scala* dell'indirizzamento, che è pensato proprio per essere utilizzato in casi come questo. Infatti, `array(, %esi, 2)` calcolerà l'indirizzo `array + 2 * esi`. Da notare la virgola subito dopo la parentesi, che sta a indicare che il registro *base* è stato omesso, mentre `%esi` è *indice*.

Per concludere, torniamo sul codice C che abbiamo visto prima come modello di questo programma: come viene gestito lì questo problema? In quel codice C non vi è alcun errore perché `array[esi]`, sfruttando la tipizzazione e l'aritmetica dei puntatori, applica sempre i fattori di scala corretti. Infatti, mentre il processore che esegue istruzioni assembler non ha idea del significato di tali istruzioni, linguaggi di livello più alto introducono concetti come i tipi proprio per permettere di esprimere questi significati e occuparsene a tempo di compilazione per produrre assembler corretto.

Il codice finale, scaricabile [qui](#), è il seguente:

```
.include "../files/utility.s"

.data
array:      .word 1, 256, 256, 512, 42, 2048, 1024, 1, 0
array_len:  .long 9
numero:     .word 1

.text

_main:
    nop
    mov $0, %cl
    mov numero, %ax
    mov $0, %esi

comp:
    cmp array_len, %esi
    je fine
    cmpw array(, %esi, 2), %ax
    jne poi
    inc %cl

poi:
    inc %esi
    jmp comp

fine:
    mov %cl, %al
    call outdecimal_byte
    ret
```

## 8.2 Esercizio 3.1: esercizio d'esame 2021-01-08

Il testo con soluzione si trova [qui](#).

### Provare da sé

Provare a svolgere da sé l'esercizio, prima di guardare la soluzione o andare oltre per la discussione.

Questo esercizio pone principalmente tre spunti.

Il primo è la gestione dell'input, da eseguire con un loop di `inchar` e controlli, facendo `outchar` solo quando il carattere è accettato. Questo è stato già visto, per esempio, nell'esercizio 2.1.

Il secondo spunto riguarda il *dimensionamento* dei dati da gestire. Infatti, dobbiamo scegliere se usare 8, 16 o 32 bit, e possiamo farlo solo cercando di capire su quanti bit sta il numero più grande che possiamo gestire.

Data la natura del problema, è facile intuire che questo si trova quando  $N = 9$  e  $k = 9$ . Dovremmo stampare un triangolo 9 righe, ciascuna composta da 1 a 9 numeri, a partire da 1 e di passo 9. Da una parte, potremmo ricordarci questa è una sequenza nota: la somma di  $1 + 2 + \dots + n$  è  $\frac{n(n+1)}{2}$ , quindi abbiamo  $9 \cdot 10/2 = 45$  elementi. Tuttavia, un approccio più semplice porta a un risultato simile: di sicuro il triangolo avrà meno elementi di un quadrato di lato 9, composto da  $9 \cdot 9 = 81$  elementi e, dato che la diagonale è inclusa, avrà più della metà di questo, cioè  $81/2$ . Possiamo quindi dire con questo ragionamento che sono più di 41 elementi e meno di 81, mentre usando la formula esatta troviamo che sono 45.

Dato che incrementiamo di passo 9 ogni volta, il numero di posizione  $j$  sarà  $(j-1) \cdot 9 + 1$ . Considerando per la stima di prima il 41-esimo elemento, abbiamo  $40 \cdot 9 + 1 = 361$ , mentre l'81-esimo elemento (che non sarà mai presente) sarebbe  $80 \cdot 9 + 1 = 721$ . Il valore esatto, se ci ricordiamo la formula di cui sopra, è invece  $44 \cdot 9 + 1 = 397$ . Un tale numero deve essere rappresentato su più di 8 bit, ma sta senza problemi in 16 bit: svolgeremo quindi i nostri calcoli usando delle *word* di 16 bit.

Non resta quindi che fare la stampa del triangolo. Questo si può scrivere come un doppio loop, dove il loop interno usa il contatore esterno per determinare quando uscire stampando una nuova riga, mentre un registro contatore viene utilizzato durante ogni ciclo per calcolare il nuovo numero da stampare. In (pseudo) C, tale ciclo avrebbe una forma simile:

```
short c = 1; // word da 16 bit
for(int i = 0; i < n; i++)    for(int j = 0; j < i + 1; j++)        outdecimal_word(c);        outchar(' ');
```

## 8.3 Esercizi per casa

### 8.3.1 Esercizio 3.2

Scrivere un programma che svolge quanto segue.

# leggere 2 numeri interi in base 10, calcolarne il prodotto, e stampare il risultato.

# lettura:

# come primo carattere leggere il segno del numero, cioè un '+' o un '-'

# segue il modulo del numero, minore di 256

# stampa:

# stampare prima il segno del numero (+ o -), poi il modulo in cifre decimali

### 8.3.2 Esercizio 3.3

Quello che segue (e scaricabile [qui](#)) è un tentativo di soluzione dell'esercizio precedente. Contiene tuttavia uno o più bug. Trovarli e correggerli.

```
.include "./files/utility.s"

mess1: .asciz "inserire il primo numero intero:\r"
mess2: .asciz "inserire il secondo numero intero:\r"
mess3: .asciz "il prodotto dei due numeri e':\r"
a:     .word 0
b:     .word 0

_main:
    nop
    lea mess1, %ebx
    call outline
    call in_intero
    mov %ax, a
    lea mess2, %ebx
```

```

    call outline
    call in_intero
    mov %ax, b

    mov a, %ax
    mov b, %bx
    imul %bx

    lea mess3, %ebx
    call outline
    call out_intero
    ret

# legge un intero composto da segno e modulo minore di 256
# ne lascia la rappresentazione in complemento alla radice base 2 in ax
in_intero:
    push %ebx
    mov $0, %bl
in_segno_loop:
    call inchar
    cmp $'+', %al
    je in_segno_poi
    cmp $'- ', %al
    jne in_segno_loop
    mov $1, %bl
in_segno_poi:
    call outchar
    call indecimal_word
    call newline
    cmp $1, %bl
    jne in_intero_fine
    neg %ax
in_intero_fine:
    pop %ebx
    ret

# legge la rappresentazione di un numero intero in complemento alla radice base 2 in eax
# lo stampa come segno seguito dalle cifre decimali
out_intero:
    push %ebx
    mov %eax, %ebx
    cmp $0, %ebx
    ja out_intero_pos
    jmp out_intero_neg
out_intero_pos:
    mov $'+', %al
    call outchar
    jmp out_intero_poi
out_intero_neg:
    mov $'- ', %al
    call outchar
    neg %ebx
    jmp out_intero_poi
out_intero_poi:
    mov %ebx, %eax
    call outdecimal_long
    pop %ebx
    ret

```

### Soluzione passo-passo

Per brevità, e vista la documentazione dei sottoprogrammi, lascio al lettore l'interpretazione a grandi linee del programma. Passeremo direttamente ai problemi incontrati testando il programma.

```

inserire il primo numero intero:
+30
Segmentation fault

```

L'errore, sicuramente già ben noto, è in realtà un risultato tipico di una vasta gamma di errori. Di per sé significa semplicemente "tentativo di accesso in una zona di memoria a cui non si può accedere per fare quello che si voleva fare". Non spiega, per esempio, cos'è che si voleva fare e perché è sbagliato.

Vediamo tramite il debugger.

```

Program received signal SIGSEGV, Segmentation fault.
_main () at /mnt/c/reti_logiche/assembler/lezioni/2/imul_debug.s:14
14      mov %ax, a

```

Questo ci dice che il problema è il tentativo di scrivere all'indirizzo *a*, che è la word allocata poco più su. Il problema qui è che il programma non ha nessuna distinzione tra `.data` e `.text`: di default è tutto `.text`, dove non si può scrivere perché non ci è permesso, normalmente, sovrascrivere le istruzioni del programma. Il problema inverso si avrebbe tentando di *eseguire* dalla sezione `.data`.

Correggiamo l'errore aggiungendo le dichiarazioni di queste due sezioni.

```
.include "../files/utility.s"

.data
mess1: .asciz "inserire il primo numero intero:\r"
mess2: .asciz "inserire il secondo numero intero:\r"
mess3: .asciz "il prodotto dei due numeri e':\r"
a:     .word 0
b:     .word 0

.text
_main:
...
```

Ritestiamo quindi il programma:

```
inserire il primo numero intero:
+30
inserire il secondo numero intero:
+20
il prodotto dei due numeri e':
+600
```

Fin qui, sembra andare bene. Ricordiamoci però di testare *tutti i casi di interesse*, in particolare i *casi limite*. Le specifiche dell'esercizio ci chiedono di considerare numeri interi di modulo inferiore a 256.

```
inserire il primo numero intero:
+255
inserire il secondo numero intero:
+255
il prodotto dei due numeri e':
+65025
```

Corretto.

```
inserire il primo numero intero:
-255
inserire il secondo numero intero:
+255
il prodotto dei due numeri e':
+511
```

Decisamente non corretto. Verifichiamo col debugger. Per prima cosa, ci assicuriamo che la lettura di numeri negativi sia corretta. Mettiamo un breakpoint a riga 16 (riga 14 prima dell'aggiunta di `.data` e `.text`), e verifichiamo cosa viene letto quando inseriamo -255.

```
(gdb) b 16
Breakpoint 2 at 0x56556774: file /mnt/c/reti_logiche/assembler/lezioni/2/imul_debug.s, line 16.
(gdb) c
Continuing.
inserire il primo numero intero:
-255
```

```
Breakpoint 2, _main () at /mnt/c/reti_logiche/assembler/lezioni/2/imul_debug.s:16
16      mov %ax, a
(gdb) i r ax
ax      0xff01      -255
(gdb)
```

Fin qui è bene, il problema non sembra essere nella lettura di interi da tastiera. Proseguiamo quindi alla moltiplicazione, e controlliamone il risultato. La `imul` utilizzata è a 16 bit, che da documentazione vediamo usa `%ax` come operando implicito, `%bx` come operando esplicito, e `%dx_%ax` come destinatario del calcolo.

```
Breakpoint 3, _main () at /mnt/c/reti_logiche/assembler/lezioni/2/imul_debug.s:25
25      imul %bx
(gdb) i r ax bx
ax      0xff01      -255
bx      0xff       255
(gdb) s
27      lea mess3, %ebx
(gdb) i r dx ax
dx      0xffff      -1
ax      0x1ff       511
(gdb)
```

Concatenando i due registri otteniamo `0xffff01ff`, ricordando, in particolare per `%ax`, che `gdb` omette nelle stampe gli zeri all'inizio di esadecimali. Possiamo verificare questo valore convertendo da esadecimale a decimale con una calcolatrice da programmatore. Dato che si parla di interi, è importante impostare la calcolatrice sul numero di bit giusti, in questo caso 32. Nella calcolatrice di Windows, questo significa impostare la modalità `DWORD`.



Figura 8.3: Uso della calcolatrice da programmatore in Windows 11 per convertire da esadecimale a intero decimale.

Il risultato è `-65025`, che è quello che ci aspettiamo. Anche qui, quindi, è bene: resta allora la stampa di questo valore, cioè il sottoprogramma `out_intero`.

```
# legge la rappresentazione di un numero intero in complemento alla radice base 2 in eax
# lo stampa come segno seguito dalle cifre decimali
out_intero:
...
```

Vediamo qui la prima discrepanza: il sottoprogramma si aspetta il risultato in `%eax`, ma noi sappiamo che la `imul` lo lascia in `%dx_%ax`. Ci si può chiedere quale dei due correggere, se il sottoprogramma o il programma che lo usa. In generale, è una buona cambiare le specifiche di un componente interno (il sottoprogramma) solo quando queste *non hanno senso*. È quindi il componente esterno (il programma) che non rispetta le specifiche d'uso di quello interno, e che va cambiato.

Assicuriamoci allora di lasciare il risultato nel registro giusto prima di `call out_intero`.

```
...
mov a, %ax
mov b, %bx
imul %bx

shl $16, %edx
movw %ax, %dx
movl %edx, %eax

lea mess3, %ebx
call outline
call out_intero
...
```

Riproviamo ad eseguire:

```
inserire il primo numero intero:
-255
inserire il secondo numero intero:
+255
il prodotto dei due numeri e':
+4294902271
```

Il risultato è cambiato, ma è comunque sbagliato. Ritorniamo al debugger, cominciando dalla call di `out_intero`, verificando di avere il valore corretto in `%eax`.

```
Breakpoint 2, _main () at /mnt/c/reti_logiche/assembler/lezioni/2/imul_debug.s:33
33      call out_intero
(gdb) i r eax
eax      0xffff01ff      -65025
(gdb)
```

Il valore in `%eax` è corretto, il problema allora è nel sottoprogramma. Proseguiamo nel sottoprogramma, cercando di capire come funziona e dove potrebbe sbagliare. La prima cosa che notiamo è che `out_intero` ha due rami, `out_intero_pos` e `out_intero_neg`, dove stampa segni diversi e, in caso di numero negativo, usa la `neg` per ottenere l'opposto. Quando si giunge a `out_intero_poi`, stampa il modulo del numero usando `outdecimal_long` (che, ricordiamo, supporta solo numeri naturali). Tuttavia, nella nostra esecuzione abbiamo un negativo che viene stampato come naturale.

Verifichiamo seguendo l'esecuzione con `step`, che entra nel sottoprogramma `out_intero`:

```
(gdb) s
out_intero () at /mnt/c/reti_logiche/assembler/lezioni/2/imul_debug.s:62
62      push %ebx
(gdb) s
63      mov %eax, %ebx
(gdb) s
64      cmp $0, %ebx
(gdb) i r ebx
ebx      0xffff01ff      -65025
(gdb) s
65      ja out_intero_pos
(gdb) s
out_intero_pos () at /mnt/c/reti_logiche/assembler/lezioni/2/imul_debug.s:68
68      mov $'+', %al
(gdb)
```

Effettivamente, nonostante `%ebx` sia un numero negativo, il salto a `out_intero_pos` viene eseguito. Guardiamo però più attentamente: l'istruzione di salto è `ja`, che interpreta il confronto come *tra numeri naturali*. In effetti, qualunque valore di `%ebx` diverso da 0, se interpretato come naturale, risulta maggiore di 0. Correggiamo quindi utilizzando `jg`, e ritestiamo.

```
cmp $0, %ebx
jg out_intero_pos
jmp out_intero_neg
```

```
inserire il primo numero intero:
-255
inserire il secondo numero intero:
+255
il prodotto dei due numeri e':
-65025
```

Si dovrebbe ora continuare con altri test (combinazioni di segni, uso di 0) fino a convincersi che funzioni. Per questa soluzione ci fermiamo qui.

Il codice finale, scaricabile [qui](#), è il seguente:

```
.include "./files/utility.s"

.data
mess1: .asciz "inserire il primo numero intero:\r"
mess2: .asciz "inserire il secondo numero intero:\r"
mess3: .asciz "il prodotto dei due numeri e':\r"
a:     .word 0
b:     .word 0

.text
_main:
nop
lea mess1, %ebx
```

```

call outline
call in_intero
mov %ax, a

lea mess2, %ebx
call outline
call in_intero
mov %ax, b

mov a, %ax
mov b, %bx
imul %bx

shl $16, %edx
movw %ax, %dx
movl %edx, %eax

lea mess3, %ebx
call outline
call out_intero
ret

# legge un intero composto da segno e modulo minore di 256
# ne lascia la rappresentazione in complemento alla radice base 2 in ax
in_intero:
push %ebx
mov $0, %bl
in_segno_loop:
call inchar
cmp $'+', %al
je in_segno_poi
cmp $'-', %al
jne in_segno_loop
mov $1, %bl
in_segno_poi:
call outchar
call indecimal_word
call newline
cmp $1, %bl
jne in_intero_fine
neg %ax
in_intero_fine:
pop %ebx
ret

# legge la rappresentazione di un numero intero in complemento alla radice base 2 in eax
# lo stampa come segno seguito dalle cifre decimali
out_intero:
push %ebx
mov %eax, %ebx
cmp $0, %ebx
jg out_intero_pos
jmp out_intero_neg
out_intero_pos:
mov $'+', %al
call outchar
jmp out_intero_poi
out_intero_neg:
mov $'-', %al
call outchar
neg %ebx
jmp out_intero_poi
out_intero_poi:
mov %ebx, %eax
call outdecimal_long
pop %ebx
ret

```

### 8.3.3 Esercizio 3.4

Quello che segue (e scaricabile [qui](#)) è un tentativo di soluzione per le seguenti specifiche:

```

# Leggere una riga dal terminale, che DEVE contenere almeno 2 caratteri '_'
# Identificare e stampa la sottostringa delimitata dai primi due caratteri '_'

```

Un esempio di output ( [qui](#) in formato txt) è il seguente



```
questa e' una _prova_ !!
prova
```

Contiene tuttavia uno o più bug. Trovarli e correggerli.

```
.include "../files/utility.s"

.data

msg_in: .fill 80, 1, 0

.text
_main:
    nop
    mov $80, %cx
    lea msg_in, %ebx
    call inline

    cld
    mov $'_ ', %al
    lea msg_in, %esi
    mov $80, %cx

    repne scasb
    mov %esi, %ebx
    repne scasb
    mov %esi, %ecx
    sub %ebx, %ecx
    call outline

    ret
```

### Soluzione

Il programma usa `repne scasb` per scorrere il vettore finché non trova il carattere in `%al`, cioè `_`. Dopo la prima scansione, salva l'indirizzo attuale per usarlo come indirizzo di partenza della sottostringa. Dopo la seconda scansione, fa una sottrazione di indirizzi per trovare il numero di caratteri che compongono la sottostringa. Usando indirizzo di partenza e numero caratteri, stampa quindi a terminale.

I bug da trovare sono i seguenti:

- Le istruzioni `rep` utilizzano `%ecx`, ma la riga 17 inizializza solo `%cx`. Questo funziona solo se, per puro caso, la parte alta di `%ecx` è a 0 ad inizio programma.
- L'istruzione `scasb` ha l'indirizzo del vettore come destinatario implicito in `%edi`, non `%esi`.
- La `repne scasb` termina *dopo* aver scansionato il carattere che rispetta l'equivalenza. Questo vuol dire che dopo la prima scansione abbiamo l'indirizzo del carattere dopo il primo `_` (corretto) ma dopo la seconda scansione abbiamo l'indirizzo del carattere dopo il secondo `_`: la sottrazione calcola una sottostringa che include il `_` di terminazione.
- Il sottoprogramma usato è quello sbagliato: `outline` stampa finché non incontra `\r`, per indicare il numero di caratteri da stampare va usato `outmess`.

Il codice dopo le correzioni è quindi il seguente, scaricabile [qui](#).

```
.include "../files/utility.s"

.data

msg_in: .fill 80, 1, 0

.text
_main:
    nop
    mov $80, %cx
    lea msg_in, %ebx
    call inline

    cld
    mov $'_ ', %al
    lea msg_in, %edi
    mov $80, %cx
```

```
repne scasb
mov %edi, %ebx
repne scasb
mov %edi, %ecx
sub %ebx, %ecx
dec %ecx
call outmess

ret
```

Si sottolinea inoltre una debolezza della soluzione: la sottrazione fra puntatori funziona solo perché la scala è 1, cioè maneggiamo valori da 1 byte, per cui c'è corrispondenza fra la differenza di due indirizzi e il numero di elementi fra loro. Una soluzione più robusta è utilizzare la differenza del contatore `%ecx` anziché di puntatori. In alternativa, si può utilizzare shift a destra dopo la sottrazione per tener conto di una scala maggiore di 1, ma è un metodo che rende facile sbagliare (bisogna stare attenti all'ordine tra shift e decremento).

Si può verificare ciò svolgendo un esercizio simile a questo basato ma basato su word, per esempio con serie di valori decimali delimitati da 0.

### 8.3.4 Esercizio 3.5

A partire dalla soluzione dell'esercizio precedente, estendere il programma per rispettare le seguenti specifiche:

```
# Leggere una riga dal terminale
# Identificare e stampa la sottostringa delimitata dai primi due caratteri '_'
# Se un solo carattere '_' e' presente, assumere che la sottostringa cominci
# ad inizio stringa e finisca prima del carattere '_'
# Se nessun carattere '_' e' presente, stampare l'intera stringa
```

#### Soluzione

Il programma dell'esercizio 3.4 viene complicato dalla richiesta di gestire dei valori di default, in caso siano presenti uno o nessun delimitatore `_`. Questo vuol dire gestire il caso in cui una `repne scasb` termina non perché ha trovato il carattere, ma perché `%ecx` è stato decrementato fino a 0.

Questo si implementa come dei semplici check su `%ecx` dopo ciascuna `repne scasb`, in caso sia 0 si va a un branch separato: se succede alla prima scansione non è presente alcun `_` e saltiamo quindi a `print_all`, se succede alla seconda scansione abbiamo solo un `_` e saltiamo quindi a `print_from_start`. Altrimenti, si prosegue con lo stesso codice dell'esercizio 2.2, che nomineremo `print_substr`.

Per `print_all` basta una semplice outline dell'intera stringa. Per `print_from_start`, si fa un ragionamento non dissimile da quanto visto per l'esercizio precedente, dove va usato come inizio l'indirizzo di `msg_in` e il numero di caratteri può essere calcolato, come prima, usando l'indirizzo che troviamo in `%edi` dopo la prima `repne scasb`.

Il codice risultante è il seguente, scaricabile [qui](#).

```
.include "../files/utility.s"

.data

msg_in: .fill 80, 1, 0

.text
_main:
nop
mov $80, %cx
lea msg_in, %ebx
call inline

cld
mov $'_', %al
lea msg_in, %edi
mov $80, %ecx

repne scasb
cmp $0, %ecx
je print_all

mov %edi, %ebx
repne scasb
cmp $0, %ecx
je print_from_start

print_substr:
mov %edi, %ecx
```

```
sub %ebx, %ecx  
dec %ecx  
call outmess  
ret
```

```
print_from_start:  
mov %ebx, %ecx  
lea msg_in, %ebx  
sub %ebx, %ecx  
dec %ecx  
call outmess  
ret
```

```
print_all:  
lea msg_in, %ebx  
call outline  
ret
```

## **Parte III**

# **Assembler - Documentazione**

## Capitolo 9

# Architettura x86

Riportiamo qui una vista *semplificata e riassuntiva* dell'architettura x86 per la quale scriveremo programmi assembler. L'architettura x86 è a 32 bit. Questo implica che i registri generali, così come tutti gli indirizzi per locazioni in memoria, sono a 32 bit. L'evoluzione di questa architettura, x64 a 64 bit, che è quella che troviamo nei processori in commercio, è del tutto retrocompatibile.

### Importanti semplificazioni

La visione del processore che proponiamo è molto limitata, e omette diversi importanti registri, flag e funzionalità che saranno esplorati in corsi successivi. Questi includono, per esempio, il registro `ebp`, la natura dei meccanismi di protezione, il significato di `SEGMENTATION FAULT`, e che cosa sia un *kernel*. Quanto discutiamo è tuttavia sufficiente agli scopi didattici di questo corso.

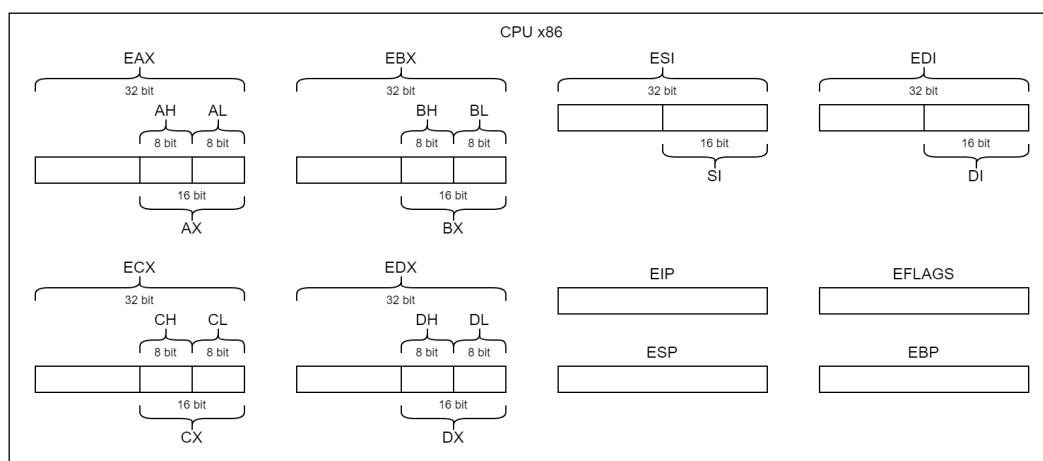
## 9.1 Registri

I registri che utilizzeremo *direttamente* sono 6: `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`. Per i primi quattro di questi, è possibile operare sulle loro porzioni a 16 e 8 bit tramite `ax`, `ah`, `al` e così via. Per i registri `esi` ed `edi` è possibile operare solo sulle porzioni a 16 bit, tramite `si` e `di`. Tipicamente, i registri `eax... edx` sono utilizzati per processare dati, mentre `esi` ed `edi` sono utilizzati come registri puntatori. Questa divisione di utilizzo non è però affatto obbligatoria per la maggior parte delle istruzioni.

Altri registri sono invece utilizzati in modo indiretto:

- `esp` è il registro puntatore per la *cima* dello stack, viene utilizzato da `pop` / `push` per prelevare/spostare valori nella pila, e da `call` / `ret` per la chiamata di sottoprogrammi;
- `eip` è il registro puntatore verso la prossima istruzione da eseguire, viene incrementato alla fine del *fetch* di una istruzione e modificato da istruzioni che cambiano il flusso d'esecuzione, come `call`, `ret` e le varie `jmp`;
- `eflags` è il registro dei flag, una serie di booleani con informazioni sullo stato dell'esecuzione e sul risultato dell'ultima operazione aritmetica. I flag di nostro interesse sono il carry flag `CF` (posizione 0), lo zero flag `ZF` (6), il sign flag `SF` (7), l'overflow flag `OF` (11). Sono tipicamente aggiornati dalle istruzioni aritmetiche, e testati indirettamente con istruzioni condizionali come `jcon`, `set` e `cmov`.

Di seguito uno schema funzionale dei registri del processore x86.



## 9.2 Memoria

Lo spazio di memoria dell'architettura x86 è indirizzato su 32 bit. Ciascun indirizzo corrisponde a un byte, ma è possibile eseguire anche letture e scritture a 16 e 32 bit.

Per tali casi è importante ricordare che l'architettura x86 è *little-endian*, che significa **little end first**, [un riferimento a I viaggi di Gulliver](#). Questo si traduce nel fatto che quando un valore di  $n$  byte viene salvato in memoria *a partire* dall'indirizzo  $a$ , il byte meno significativo del valore viene salvato in  $a$ , il secondo meno significativo in  $a + 1$ , e così via fino al più significativo in  $a + (n - 1)$ .

Questo ordinamento dei bytes in memoria non inficia sulla coerenza dei dati nei registri: eseguendo `movl %eax, a` e `movl a, %eax` il contenuto di `eax` non cambia, e l'ordinamento dei bit rimane coerente.

I *meccanismi di protezione* ci precludono l'accesso alla maggior parte dello spazio di memoria. Potremmo accedere senza incorrere in errori solo

- 2. allo stack
- 2. allo spazio allocato nella sezione `.data`
- 2. alle istruzioni nella sezione `.text`

Queste sezioni tipicamente non includono gli indirizzi “bassi”, cioè a partire da `0x0`.

È importante anche tenere presente che

- 2. non è possibile *eseguire* istruzioni dallo stack e da `.data`
- 2. non è possibile *scrivere* nella sezione `.text`

Vanno quindi opportunamente dichiarate le sezioni, e vanno evitate operazioni di `jmp`, `call` etc. verso locazioni di `.data` così come le `mov` verso locazioni di `.text`.

In caso di violazione di questi meccanismi, l'errore più tipico è `SEGMENTATION FAULT`.

## 9.3 Spazio di I/O

Lo spazio di I/O, sia quello fisico (monitor, speaker, tastiera, etc.) sia quello virtuale (terminale, files su disco, etc.) ci è in realtà precluso tramite *meccanismi di protezione*. Tentare di eseguire istruzioni `in` o `out` porterà infatti al brusco arresto del programma. Il nostro programma può interagire con lo spazio di I/O solo tramite il *kernel* del *sistema operativo*.

Tutta questa complessità è astratta tramite i *sottoprogrammi di input/output* dell'ambiente, documentati [qui](#).

## 9.4 Condizioni al reset

Il reset iniziale e l'avvio del nostro programma sono concetti completamente diversi e scollegati. Non possiamo sfruttare nessuna ipotesi sullo stato dei registri al momento dell'avvio del nostro programma, se non che il registro `eip` punterà a un certo punto alla prima istruzione di `_main`.

Il fatto che `_main` sia l'entry point del nostro programma, così come l'uso di `ret` senza alcun valore di ritorno, è una caratteristica di *questo ambiente*.

# Capitolo 10

## Sezione .data

Un programma assembler è tipicamente diviso in sezione `.data`, dove vengono allocato spazio in memoria a disposizione del programma, e sezione `.text`, dove viene indicata la sequenza di istruzione che compone il programma. La sezione `data` è tipicamente composta da una serie di dichiarazioni nella forma `nomeVariabile: .tipo <parametri di inizializzazione>`. Alcuni esempi:

```
.data
var1: .long 5
var2: .byte 0x2d, 0x01
str: .asciz "Una stringa"
```

Ciascuna direttiva non fa che allocare uno o più blocchi di memoria contigui della dimensione richiesta e con il contenuto iniziale richiesto.

Ciascuna *label* non è che un indirizzo al primo byte di tale blocco contiguo di memoria. Dato che l'architettura x86 è *little-endian*, tale primo byte sarà il meno significativo.

### 10.1 Direttive di allocazione

Tipo	Notazione	Descrizione
byte	<code>.byte V1 [, V2...]</code>	Alloca uno o più byte, inizializzati con i valori forniti.
word	<code>.word V1 [, V2...]</code>	Alloca uno o più word (2 byte), inizializzati con i valori forniti.
long	<code>.long V1 [, V2...]</code>	Alloca uno o più long (4 byte), inizializzati con i valori forniti.
fill	<code>.fill n, l, v</code>	Alloca <i>n</i> locazioni di <i>l</i> byte ciascuno e inizializzati a <i>v</i> . <i>l</i> e <i>v</i> si possono omettere, di default sono 1 e 0.
ascii	<code>.ascii "str"</code>	Alloca la stringa <code>str</code> , 1 byte per carattere.
asciz	<code>.asciz "str"</code>	Alloca la stringa <code>str</code> , 1 byte per carattere, aggiungendo un byte <code>0x00</code> in fondo.

L'assemblatore supporta anche altre direttive e usi più complessi. Per maggiori informazioni, la documentazione ufficiale è [qui](#).

### 10.2 Valori letterali

Il contenuto di ciascuna allocazione è definito tramite valori letterali, che devono essere *costanti* note o derivabili a tempo di compilazione.

Tipo	Esempio	Descrizione
Decimale	<code>.byte 2</code>	Costante in notazione decimale.
Esadecimale	<code>.byte 0x0d</code>	Costante in notazione esadecimale.
Binario	<code>.byte 0b00001101</code>	Costante in notazione binaria.
ASCII	<code>.byte 'a', 'r'</code>	Costante in notazione ASCII, il carattere viene tradotto nel byte corrispondente.
label	<code>.long val0</code>	Indirizzo corrispondente a un'altra label.
label e offset	<code>.long val0+1</code>	Indirizzo corrispondente a un'altra label, più offset. La scala è sempre 1.

**Attenzione alle dimensioni**

I valori letterali vengono automaticamente troncati o estesi per rientrare nelle dimensioni specificate dalla direttiva.

```
.data  
b1: .byte 0xd0e # viene troncato a 0xe  
w1: .word 0xd   # viene esteso a 0x000d  
w2: .word 0xf1  # viene esteso a 0x00f1
```



# Capitolo 11

## Istruzioni processore x86

Le seguenti tabelle sono per *riferimento rapido* : sono utili per la programmazione pratica, ma omettono molteplici dettagli che serve sapere, e che trovate nel resto del materiale.

Si ricorda che utilizziamo la sintassi GAS/AT&T, dove le istruzioni sono nel formato *opcode source destination*. Nella colonna notazione, indicheremo con `[bwl]` le istruzioni che richiedono la specifica delle dimensioni. Quando la dimensione è deducibile dai registri utilizzati, questi suffissi si possono omettere.

Per gli operandi, useremo le seguenti sigle:

- *r* per un registro (come in `mov %eax, %ebx`);
- *m* per un indirizzo di memoria;
- *i* per un valore immediato (come in `mov $0, %eax`).

Per gli indirizzi in memoria, abbiamo a disposizione tre notazioni:

- immediato, come in `mov numero, %eax`;
- tramite registro, come in `mov (%esi), %eax`;
- con indice, come in `mov matrice(%esi, %ecx, 4), %eax`.

Si ricorda che non tutte le combinazioni sono permesse nell'architettura x86: nessuna istruzione generale supporta l'indicazione di *entrambi* gli operandi in memoria (cioè, non si può scrivere `movl x, y` o `mov (%eax), (%ebx)`). Fanno eccezione le istruzioni stringa come la **movs**, usando operandi impliciti.

### 11.1 Immediati

Si parla di immediati quando si usano valori costanti all'interno di una istruzione. Sia l'assemblatore che le istruzioni distinguono due tipi di immediato, indirizzo e valore, e si comportano in modo diverso in base a ciò. Per esempio, una `mov` che ha come sorgente un indirizzo immediato legge il valore contenuto a quell'indirizzo, mentre con un valore immediato legge semplicemente il valore.

Tipo	Esempio	Descrizione
Indirizzo letterale	<code>mov 0x01f2a3b0, %eax</code>	Un indirizzo a 32 bit. Valori più piccoli, come <code>0x0d</code> , vengono automaticamente estesi con 0.
Indirizzo tramite label	<code>mov val0, %eax</code>	Un indirizzo tramite label, per esempio proveniente dalla sezione <code>.data</code> .
Valore decimale	<code>mov \$3, %al</code>	Costante in notazione decimale.
Valore esadecimale	<code>mov \$0x0d, %al</code>	Costante in notazione esadecimale.
Valore binario	<code>mov \$0b00001101, %al</code>	Costante in notazione binaria.
Valore ASCII	<code>mov \$'r', %al</code>	Costante in notazione ASCII, il carattere viene tradotto nel byte corrispondente.

#### Attenti al \$

Nella sintassi GAS che utilizziamo, il modo con cui si dice all'assemblatore che un immediato è un valore, e non un indirizzo, è il `$`. Dimenticarlo è fonte di `SEGMENTATION FAULT` quando va bene, bug molto bizzari quando va male.

## 11.2 Spostamento di dati

Istruzione	Nome esteso	Notazione	Comportamento
mov	Move	mov[bwl] r/m/i, r/m	Scrive il valore sorgente nel destinatario. Non modifica alcun flag.
lea	Load Effective Address	lea m, r	Scrive l'indirizzo m nel registro destinatario.
xchg	Exchange	xchg[bwl] r/m, r/m	Scambia il valore del sorgente con quello del destinatario.
cbw	Convert Byte to Word	cbw	Estende il contenuto di %al su %ax, interpretandone il contenuto come intero.
cwde	Convert Word to Doubleword	cwde	Estende il contenuto di %ax su %eax, interpretandone il contenuto come intero.
push	Push onto the Stack	push[wl] r/m/i	Aggiunge il valore sorgente in cima allo stack (destinatario implicito).
pop	Pop from the Stack	pop[wl] r/m	Rimuove un valore dallo stack (sorgente implicito) lo scrive nel destinatario.

## 11.3 Aritmetica

Istruzione	Nome esteso	Notazione	Comportamento
add	Addition	add[bwl] r/m/i, r/m	Somma sorgente e destinatario, scrive il risultato sul destinatario. Valido sia per naturali che interi. Aggiorna SF, ZF, CF e OF.
sub	Subtraction	sub[bwl] r/m/i, r/m	Sottrae il sorgente dal destinatario, scrive il risultato sul destinatario. Valido sia per naturali che interi. Aggiorna SF, ZF, CF e OF.
adc	Addition with Carry	adc[bwl] r/m/i, r/m	Somma sorgente, destinatario e CF, scrive il risultato sul destinatario. Valido sia per naturali che interi. Aggiorna SF, ZF, CF e OF.
sbb	Subtraction with Borrow	sub[bwl] r/m/i, r/m	Sottrae il sorgente e CF dal destinatario, scrive il risultato sul destinatario. Valido sia per naturali che interi. Aggiorna SF, ZF, CF e OF.
inc	Increment	inc[bwl] r/m	Somma 1 (sorgente implicito) al destinatario. Aggiorna SF, ZF, e OF, ma non CF.
dec	Decrement	dec[bwl] r/m	Sottrae 1 (sorgente implicito) al destinatario. Aggiorna SF, ZF, e OF, ma non CF.
neg	Negation	neg[bwl] r/m	Sostituisce il destinatario con il suo opposto. Aggiorna ZF, SF e OF. Modifica CF.

Le seguenti istruzioni hanno operandi e destinatari impliciti, che variano in base alla dimensione dell'operazione. Usano in oltre composizioni di più registri: useremo %dx\_%ax per indicare un valore i cui bit più significativi sono scritti in %dx e quelli meno significativi in %ax.

Istruzione	Nome esteso	Notazione	Comportamento
mul	Unsigned Multiply, 8 bit	mulb r/m	Calcola su 16 bit il prodotto tra naturali del sorgente e %a1, scrive il risultato su %ax. Se il risultato non è riducibile a 8 bit, mette CF e OF a 1, altrimenti a 0.
mul	Unsigned Multiply, 16 bit	mulw r/m	Calcola su 32 bit il prodotto tra naturali del sorgente e %ax, scrive il risultato su %dx_%ax. Se il risultato non è riducibile a 16 bit, mette CF e OF a 1, altrimenti a 0.
mul	Unsigned Multiply, 32 bit	mull r/m	Calcola su 64 bit il prodotto tra naturali del sorgente e %eax, scrive il risultato su %edx_%eax. Se il risultato non è riducibile a 32 bit, mette CF e OF a 1, altrimenti a 0.
imul	Signed Multiply, 8 bit	imulb r/m	Calcola su 16 bit il prodotto tra interi del sorgente e %a1, scrive il risultato su %ax. Se il risultato non è riducibile a 8 bit, mette CF e OF a 1, altrimenti a 0.
imul	Signed Multiply, 16 bit	imulw r/m	Calcola su 32 bit il prodotto tra interi del sorgente e %ax, scrive il risultato su %dx_%ax. Se il risultato non è riducibile a 16 bit, mette CF e OF a 1, altrimenti a 0.
imul	Signed Multiply, 32 bit	imull r/m	Calcola su 64 bit il prodotto tra interi del sorgente e %eax, scrive il risultato su %edx_%eax. Se il risultato non è riducibile a 32 bit, mette CF e OF a 1, altrimenti a 0.

Istruzione	Nome esteso	Notazione	Comportamento
div	Unsigned Divide, 8 bit	divb r/m	Calcola su 8 bit la divisione tra naturali tra %ax (dividendo implicito) e il sorgente (divisore). Scrive il quoziente su %al e il resto su %ah. Se il quoziente non è rappresentabile su 8 bit, causa <i>crash del programma</i> .
div	Unsigned Divide, 16 bit	divw r/m	Calcola su 16 bit la divisione tra naturali tra %dx_%ax (dividendo implicito) e il sorgente (divisore). Scrive il quoziente su %ax e il resto su %dx. Se il quoziente non è rappresentabile su 16 bit, causa <i>crash del programma</i> .
div	Unsigned Divide, 32 bit	divl r/m	Calcola su 32 bit la divisione tra naturali tra %edx_%eax (dividendo implicito) e il sorgente (divisore). Scrive il quoziente su %eax e il resto su %edx. Se il quoziente non è rappresentabile su 32 bit, causa <i>crash del programma</i> .
idiv	Signed Divide, 8 bit	idivb r/m	Calcola su 8 bit la divisione tra interi tra %ax (dividendo implicito) e il sorgente (divisore). Scrive il quoziente su %al e il resto su %ah. Se il quoziente non è rappresentabile su 8 bit, causa <i>crash del programma</i> .
idiv	Signed Divide, 16 bit	idivw r/m	Calcola su 16 bit la divisione tra interi tra %dx_%ax (dividendo implicito) e il sorgente (divisore). Scrive il quoziente su %ax e il resto su %dx. Se il quoziente non è rappresentabile su 16 bit, causa <i>crash del programma</i> .
idiv	Signed Divide, 32 bit	idivl r/m	Calcola su 32 bit la divisione tra interi tra %edx_%eax (dividendo implicito) e il sorgente (divisore). Scrive il quoziente su %eax e il resto su %edx. Se il quoziente non è rappresentabile su 32 bit, causa <i>crash del programma</i> .

## 11.4 Logica binaria

Le seguenti istruzioni operano *bit a bit* : data per esempio la and, l'i-esimo bit del risultato è l'and logico tra gli i-esimi bit di sorgente e destinatario.

Istruzione	Notazione	Comportamento
not	not[bwl] r/m	Sostituisce il destinatario con la sua negazione.
and	and r/m/i, r/m	Calcola l'and logico tra sorgente e destinatario, scrive il risultato sul destinatario.
or	or r/m/i, r/m	Calcola l'or logico tra sorgente e destinatario, scrive il risultato sul destinatario.
xor	xor r/m/i, r/m	Calcola lo xor logico tra sorgente e destinatario, scrive il risultato sul destinatario.

## 11.5 Traslazione e Rotazione

Istruzione	Nome esteso	Notazione	Comportamento
shl	Shift Logical Left	shl[bwl] i/r r/m	Sia $n$ l'operando sorgente, esegue lo shift a sinistra del destinatario $n$ volte, impostando a 0 gli $n$ bit meno significativi. In ciascuno shift, il bit più significativo viene lasciato in CF. Come registro sorgente si può utilizzare solo %cl. Il sorgente può essere omissso, in quel caso $n = 1$ .
sal	Shift Arithmetic Left	sal[bwl] i/r r/m	Sia $n$ l'operando sorgente, esegue lo shift a sinistra del destinatario $n$ volte, impostando a 0 gli $n$ bit meno significativi. In ciascuno shift, il bit più significativo viene lasciato in CF. Se il bit più significativo ha cambiato valore almeno una volta, imposta OF a 1. Come registro sorgente si può utilizzare solo %cl. Il sorgente può essere omissso, in quel caso $n = 1$ .
shr	Shift Logical Right	shr[bwl] i/r r/m	Sia $n$ l'operando sorgente, esegue lo shift a destra del destinatario $n$ volte, impostando a 0 gli $n$ bit più significativi. In ciascuno shift, il bit meno significativo viene lasciato in CF. Come registro sorgente si può utilizzare solo %cl. Il sorgente può essere omissso, in quel caso $n = 1$ .
sar	Shift Arithmetic Right	sar[bwl] i/r r/m	Sia $n$ l'operando sorgente e $s$ il valore del bit più significativo del destinatario, esegue lo shift a destra del destinatario $n$ volte, impostando a $s$ gli $n$ bit più significativi. In ciascuno shift, il bit meno significativo viene lasciato in CF. Come registro sorgente si può utilizzare solo %cl. Il sorgente può essere omissso, in quel caso $n = 1$ .
rol	Rotate Left	rol[bwl] i/r r/m	Sia $n$ l'operando sorgente, esegue la rotazione a sinistra del destinatario $n$ volte. In ciascuna rotazione, il bit più significativo viene <i>sia</i> lasciato in CF <i>sia</i> ricopiato al posto del bit meno significativo. Come registro sorgente si può utilizzare solo %cl. Il sorgente può essere omissso, in quel caso $n = 1$ .
ror	Rotate Right	ror[bwl] i/r r/m	Sia $n$ l'operando sorgente, esegue la rotazione a destra del destinatario $n$ volte. In ciascuna rotazione, il bit meno significativo viene <i>sia</i> lasciato in CF <i>sia</i> ricopiato al posto del bit più significativo. Come registro sorgente si può utilizzare solo %cl. Il sorgente può essere omissso, in quel caso $n = 1$ .
rcl	Rotate with Carry Left	rcl[bwl] i/r r/m	Sia $n$ l'operando sorgente, esegue la rotazione con carry a sinistra del destinatario $n$ volte. In ciascuna rotazione, il bit più significativo viene lasciato in CF, mentre il valore di CF viene ricopiato al posto del bit meno significativo. Come registro sorgente si può utilizzare solo %cl. Il sorgente può essere omissso, in quel caso $n = 1$ .
rcr	Rotate with Carry Right	rcr[bwl] i/r r/m	Sia $n$ l'operando sorgente, esegue la rotazione con carry a destra del destinatario $n$ volte. In ciascuna rotazione, il bit meno significativo viene lasciato in CF, mentre il valore di CF viene ricopiato al posto del bit più significativo. Come registro sorgente si può utilizzare solo %cl. Il sorgente può essere omissso, in quel caso $n = 1$ .

## 11.6 Controllo di flusso

Istruzione	Nome esteso	Notazione	Comportamento
jmp	Unconditional Jump	jmp m/r	Salta incondizionatamente all'indirizzo specificato.
call	Call Procedure	call m/r	Chiamata a procedura all'indirizzo specificato. Salva l'indirizzo della prossima istruzione nello stack, così che il flusso corrente possa essere ripreso con una ret.
ret	Return from Procedure	ret	Ritorna a un flusso di esecuzione precedente, rimuovendo dallo stack l'indirizzo precedentemente salvato da una call.

La tabella seguente elenca i salti condizionati. I salti condizionati usano i flag per determinare se la condizione di salto è vera. Per un uso sempre coerente, assicurarsi che l'istruzione di salto segua immediatamente una `cmp`, o altre istruzioni che non hanno modificano i flag dopo la `cmp`. Dati gli operandi della `cmp` e una condizione `c`, per esempio `c = "maggiore o uguale"`, la condizione è vera se destinatario `c` sorgente. Nella tabella che segue, quando ci si riferisce a un confronto fra sorgente e destinatario si intendono gli operandi della `cmp` precedente.

Istruzione	Nome esteso	Notazione	Comportamento
cmp	Compare Two Operands	cmp[bwl] r/m/i, r/m	Confronta i due operandi e aggiorna i flag di conseguenza.
je	Jump if Equal	je m	Salta se destinatario == sorgente.
jne	Jump if Not Equal	jne m	Salta se destinatario != sorgente.
ja	Jump if Above	ja m	Salta se, interpretandoli come naturali, destinatario > sorgente.
jae	Jump if Above or Equal	jae m	Salta se, interpretandoli come naturali, destinatario >= sorgente.
jb	Jump if Below	jb m	Salta se, interpretandoli come naturali, destinatario < sorgente.
jbe	Jump if Below or Equal	jbe m	Salta se, interpretandoli come naturali, destinatario <= sorgente.
jg	Jump if Greater	jg m	Salta se, interpretandoli come interi, destinatario > sorgente.
jge	Jump if Greater or Equal	jge m	Salta se, interpretandoli come interi, destinatario >= sorgente.
jl	Jump if Less	jl m	Salta se, interpretandoli come interi, destinatario < sorgente.
jle	Jump if Less or Equal	jle m	Salta se, interpretandoli come interi, destinatario <= sorgente.
jz	Jump if Zero	jz m	Salta se ZF è 1.
jnz	Jump if Not Zero	jnz m	Salta se ZF è 0.
jc	Jump if Carry	jc m	Salta se CF è 1.
jnc	Jump if Not Carry	jnc m	Salta se CF è 0.
jo	Jump if Overflow	jo m	Salta se OF è 1.
jno	Jump if Not Overflow	jno m	Salta se OF è 0.
js	Jump if Sign	js m	Salta se SF è 1.
jns	Jump if Not Sign	jns m	Salta se SF è 0.

## 11.7 Operazioni condizionali

Per alcune operazioni tipiche, sono disponibili istruzioni specifiche il cui comportamento dipende dai flag e, quindi, dal risultato di una precedente `cmp`. Anche qui, quando ci si riferisce a un confronto fra sorgente e destinatario si intendono gli operandi della `cmp` precedente.

La famiglia di istruzioni `loop` supporta i cicli condizionati più tipici. Rimangono d'interesse didattico come istruzioni specializzate ma, curiosamente, nei processori moderni sono generalmente meno performanti degli equivalenti che usino `dec`, `cmp` e salti condizionati.

Istruzione	Nome esteso	Notazione	Comportamento
loop	Unconditional Loop	loop m	Decrementa %ecx e salta se il risultato è (ancora) diverso da 0.
loope	Loop if Equal	loope m	Decrementa %ecx e salta se entrambe le condizioni sono vere: 1) %ecx è (ancora) diverso da 0, 2) destinatario == sorgente.
loopne	Loop if Not Equal	loopne m	Decrementa %ecx e salta se entrambe le condizioni sono vere: 1) %ecx è (ancora) diverso da 0, 2) destinatario != sorgente.
loopz	Loop if Zero	loopz m	Decrementa %ecx e salta se entrambe le condizioni sono vere: 1) %ecx è (ancora) diverso da 0, 2) ZF è 1.
loopnz	Loop if Not Zero	loopnz m	Decrementa %ecx e salta se entrambe le condizioni sono vere: 1) %ecx è (ancora) diverso da 0, 2) ZF è 0.

La famiglia di istruzioni `set` permette di salvare il valore di un confronto in un registro o locazione di memoria. Tale operando può essere solo da 1 byte.

Istruzione	Nome esteso	Notazione	Comportamento
sete	Set if Equal	sete r/m	Imposta l'operando a 1 se destinatario == sorgente, a 0 altrimenti.
setne	Set if Not Equal	setne r/m	Imposta l'operando a 1 se destinatario != sorgente, a 0 altrimenti.
seta	Set if Above	seta r/m	Imposta l'operando a 1 se, interpretandoli come naturali, destinatario > sorgente, a 0 altrimenti.
setae	Set if Above or Equal	setae r/m	Imposta l'operando a 1 se, interpretandoli come naturali, destinatario >= sorgente, a 0 altrimenti.
setb	Set if Below	setb r/m	Imposta l'operando a 1 se, interpretandoli come naturali, destinatario < sorgente, a 0 altrimenti.
setbe	Set if Below or Equal	setbe r/m	Imposta l'operando a 1 se, interpretandoli come naturali, destinatario <= sorgente, a 0 altrimenti.
setg	Set if Greater	setg r/m	Imposta l'operando a 1 se, interpretandoli come interi, destinatario > sorgente, a 0 altrimenti.
setge	Set if Greater or Equal	setge r/m	Imposta l'operando a 1 se, interpretandoli come interi, destinatario >= sorgente, a 0 altrimenti.
setl	Set if Less	setl r/m	Imposta l'operando a 1 se, interpretandoli come interi, destinatario < sorgente, a 0 altrimenti.
setle	Set if Less or Equal	setle r/m	Imposta l'operando a 1 se, interpretandoli come interi, destinatario <= sorgente, a 0 altrimenti.
setz	Set if Zero	setz r/m	Imposta l'operando a 1 se ZF è 1, a 0 altrimenti.
setnz	Set if Not Zero	setnz r/m	Imposta l'operando a 1 se ZF è 0, a 0 altrimenti.
setc	Set if Carry	setc r/m	Imposta l'operando a 1 se CF è 1, a 0 altrimenti.
setnc	Set if Not Carry	setnc r/m	Imposta l'operando a 1 se CF è 0, a 0 altrimenti.
seto	Set if Overflow	seto r/m	Imposta l'operando a 1 se OF è 1, a 0 altrimenti.
setno	Set if Not Overflow	setno r/m	Imposta l'operando a 1 se OF è 0, a 0 altrimenti.
sets	Set if Sign	sets r/m	Imposta l'operando a 1 se SF è 1, a 0 altrimenti.
setns	Set if Not Sign	setns r/m	Imposta l'operando a 1 se SF è 0, a 0 altrimenti.

La famiglia di istruzioni `cmov` permette di eseguire, solo se il confronto ha avuto successo, una `mov` da memoria a registro o da registro a registro. Gli operandi possono essere solo a 2 o 4 byte, non 1.

Istruzione	Nome esteso	Notazione	Comportamento
cmove	Move if Equal	cmove[wl] r/m r	Esegue la mov se destinatario == sorgente, altrimenti non fa nulla.
cmovne	Move if Not Equal	cmovne[wl] r/m r	Esegue la mov se destinatario != sorgente, altrimenti non fa nulla.
cmova	Move if Above	cmova[wl] r/m r	Esegue la mov se, interpretandoli come naturali, destinatario > sorgente, altrimenti non fa nulla.
cmovae	Move if Above or Equal	cmovae[wl] r/m r	Esegue la mov se, interpretandoli come naturali, destinatario >= sorgente, altrimenti non fa nulla.
cmovb	Move if Below	cmovb[wl] r/m r	Esegue la mov se, interpretandoli come naturali, destinatario < sorgente, altrimenti non fa nulla.
cmovbe	Move if Below or Equal	cmovbe[wl] r/m r	Esegue la mov se, interpretandoli come naturali, destinatario <= sorgente, altrimenti non fa nulla.
cmovg	Move if Greater	cmovg[wl] r/m r	Esegue la mov se, interpretandoli come interi, destinatario > sorgente, altrimenti non fa nulla.
cmovge	Move if Greater or Equal	cmovge[wl] r/m r	Esegue la mov se, interpretandoli come interi, destinatario >= sorgente, altrimenti non fa nulla.
cmovl	Move if Less	cmovl[wl] r/m r	Esegue la mov se, interpretandoli come interi, destinatario < sorgente, altrimenti non fa nulla.
cmovle	Move if Less or Equal	cmovle[wl] r/m r	Esegue la mov se, interpretandoli come interi, destinatario <= sorgente, altrimenti non fa nulla.
cmovz	Move if Zero	cmovz[wl] r/m r	Esegue la mov se ZF è 1, altrimenti non fa nulla.
cmovnz	Move if Not Zero	cmovnz[wl] r/m r	Esegue la mov se ZF è 0, altrimenti non fa nulla.
cmovc	Move if Carry	cmovc[wl] r/m r	Esegue la mov se CF è 1, altrimenti non fa nulla.
cmovnc	Move if Not Carry	cmovnc[wl] r/m r	Esegue la mov se CF è 0, altrimenti non fa nulla.
cmovo	Move if Overflow	cmovo[wl] r/m r	Esegue la mov se OF è 1, altrimenti non fa nulla.
cmovno	Move if Not Overflow	cmovno[wl] r/m r	Esegue la mov se OF è 0, altrimenti non fa nulla.
cmovs	Move if Sign	cmovs[wl] r/m r	Esegue la mov se SF è 1, altrimenti non fa nulla.
cmovns	Move if Not Sign	cmovns[wl] r/m r	Esegue la mov se SF è 0, altrimenti non fa nulla.

## 11.8 Istruzioni stringa

Le istruzioni stringa sono ottimizzate per eseguire operazioni tipiche su vettori in memoria. Hanno esclusivamente operandi impliciti, che rende la specifica delle dimensioni *non* opzionale.



Istruzione	Nome esteso	Notazione	Comportamento
cld	Clear Direction Flag	cld	Imposta DF a 0, implicando che le istruzioni stringa procederanno per indirizzi crescenti.
std	Set Direction Flag	std	Imposta DF a 1, implicando che le istruzioni stringa procederanno per indirizzi decrescenti.
lods	Load String	lods[bwl]	Legge 1/2/4 byte all'indirizzo in %esi e lo scrive in %al / %ax / %eax. Se DF è 0, incrementa %esi di 1/2/4, se è 1 lo decrementa.
stos	Store String	stos[bwl]	Legge il valore in %al / %ax / %eax e lo scrive nei 1/2/4 byte all'indirizzo in %edi. Se DF è 0, incrementa %edi di 1/2/4, se è 1 lo decrementa.
movs	Move String to String	movs[bwl]	Legge 1/2/4 byte all'indirizzo in %esi e lo scrive nei 1/2/4 byte all'indirizzo in %edi. Se DF è 0, incrementa %edi di 1/2/4, se è 1 lo decrementa.
cmps	Compare Strings	cmps[bwl]	Confronta gli 1/2/4 byte all'indirizzo in %esi (sorgente) con quelli all'indirizzo in %edi (destinatario). Aggiorna i flag così come fa cmp.
scas	Scan String	scas[bwl]	Confronta %al / %ax / %eax (sorgente) con gli 1/2/4 byte all'indirizzo in %edi (destinatario). Aggiorna i flag così come fa cmp.

### 11.8.1 Repeat Instruction

Le istruzioni stringa possono essere ripetute senza controllo di programma, usando il prefisso rep.

Istruzione	Nome esteso	Notazione	Comportamento
rep	Unconditional Repeat Instruction	rep [opcode]	Dato n il valore in %ecx, ripete l'operazione opcode n volte, decrementando %ecx fino a 0. Compatibile con lods, stos, movs.
repe	Repeat Instruction if Equal	repe [opcode]	Dato n il valore in %ecx, decrementa %ecx e ripete l'operazione opcode finché 1) %ecx è (ancora) diverso da 0, e 2) gli operandi di questa ripetizione erano uguali. Compatibile con cmps e scas.
repne	Repeat Instruction if Not Equal	repne [opcode]	Dato n il valore in %ecx, decrementa %ecx e ripete l'operazione opcode finché 1) %ecx è (ancora) diverso da 0, e 2) gli operandi di questa ripetizione erano disuguali. Compatibile con cmps e scas.

## 11.9 Altre istruzioni

Istruzione	Nome esteso	Notazione	Comportamento
nop	No Operation	nop	Non cambia lo stato del processore in alcun modo, eccetto per il registro %eip.

Le seguenti istruzioni sono di interesse didattico ma non per le esercitazioni, in quanto richiedono privilegi di esecuzione.

Istruzione	Nome esteso	Notazione	Comportamento
in	Input from Port	in r/i r	Legge da una porta di input a un registro.
out	Output to Port	out r r/i	Scrive da un registro a una porta di output.
ins	Input String from Port	ins[bwl]	Legge 1/2/4 byte dalla porta di input indicata in %dx e li scrive nei 1/2/4 byte all'indirizzo in %edi.
outs	Output String to Port	outs[bwl]	Legge 1/2/4 byte all'indirizzo indicato da %esi e li scrive alla porta di output indicata in %dx.
hlt	Halt	hlt	Blocca ogni operazione del processore.

## Capitolo 12

# Sottoprogrammi di utility

Nell'architettura del processore, menzioniamo registri, istruzioni e locazioni di memoria. Quando scriviamo programmi, sfruttiamo però il concetto di *terminale*, un'interfaccia dove l'utente legge caratteri e ne scrive usando la tastiera. Come questo possa avvenire è argomento di altri corsi, dove verranno presentate le *interruzioni*, il *kernel*, e in generale cosa fa un *sistema operativo*.

In questo corso ci limitiamo a sfruttare queste funzionalità tramite del codice ad hoc contenuto in `utility.s`. Queste funzionalità sono fornite come sottoprogrammi, che hanno i loro specifici comportamenti da tenere a mente. Per utilizzare questi sottoprogrammi, utilizziamo la direttiva

```
.include "../files/utility.s"
```

### 12.1 Terminologia

Con *leggere caratteri da tastiera* si intende che il programma resta in attesa che l'utente prema un tasto sulla tastiera, inviando la codifica di quel tasto al programma.

Con *mostrare a terminale* si intende che il programma stampa un carattere a video.

Con *fare eco* di un carattere si intende che il programma, subito dopo aver letto un carattere da tastiera, lo mostra anche a schermo. Questo è il comportamento interattivo a cui siamo più abituati, ma non è automatico.

Con *ignorare caratteri* si intende che il programma, dopo aver letto un carattere, controlli che questo sia del tipo atteso: se lo è ne fa eco o comunque risponde in modo interattivo, se non lo è ritorna in lettura di un altro carattere, mostrandosi all'utente come se avesse, appunto, ignorato il carattere precedente.

### 12.2 Caratteri speciali

Avanzamento linea ( *line feed*, LF): carattere `\n`, codifica `0x0A`.

Ritorno carrello ( *carriage return*, RF): carattere `\r`, codifica `0x0D`.

Il significato di questi ha a che vedere con le macchine da scrivere, dove *avanzare alla riga successiva* e *riportare il carrello a sinistra* erano azioni ben distinte.

## 12.3 Sottoprogrammi

Nome	Comportamento
<code>inchar</code>	Legge da tastiera un carattere ASCII e ne scrive la codifica in <code>%a\</code> . Non mostra a terminale il carattere letto.
<code>outchar</code>	Legge la codifica di un carattere ASCII dal registro <code>%a\</code> e lo mostra a terminale.
<code>inbyte / inword / inlong</code>	Legge dalla tastiera 2/4/8 cifre esadecimali (0-9 e A-F), facendone eco e ignorando altri caratteri. Salva quindi il byte/word/long corrispondente a tali cifre in <code>%a\ / %ax / %eax</code> .
<code>outbyte / outword / outlong</code>	Legge il contenuto di <code>%a\ / %ax / %eax</code> e lo mostra a terminale sotto forma di 2/4/8 cifre esadecimali.
<code>indecimal_byte / indecimal_word / indecimal_long</code>	Legge dalla tastiera fino a 3/5/10 cifre decimali (0-9), o finché non è inserito un <code>\r</code> , facendone eco e ignorando altri caratteri. Interpreta queste come cifre di un numero naturale, e salva quindi il byte/word/long corrispondente in <code>%a\ / %ax / %eax</code> .
<code>outdecimal_byte / outdecimal_word / outdecimal_long</code>	Legge il contenuto di <code>%a\ / %ax / %eax</code> , lo interpreta come numero naturale e lo mostra a terminale sotto forma di cifre decimali.
<code>outmess</code>	Dato l'indirizzo <code>v</code> in <code>%ebx</code> e il numero <code>n</code> in <code>%cx</code> , mostra a terminale gli <code>n</code> caratteri ASCII memorizzati a partire da <code>v</code> .
<code>outline</code>	Dato l'indirizzo <code>v</code> in <code>%ebx</code> , mostra a terminale i caratteri ASCII memorizzati a partire da <code>v</code> finché non incontra un <code>\r</code> o raggiunge il massimo di 80 caratteri.
<code>inline</code>	Dato l'indirizzo <code>v</code> in <code>%ebx</code> e il numero <code>n</code> in <code>%cx</code> , legge da tastiera caratteri ASCII e li scrive a partire da <code>v</code> finché non è inserito un <code>\r</code> o raggiunge il massimo di <code>n - 2</code> caratteri. Pone poi in fondo i caratteri <code>\r\n</code> . Supporta l'uso di backspace per correggere l'input.
<code>newline</code>	Porta l'output del terminale a una nuova riga, mostrando i caratteri <code>\r\n</code> .

# Capitolo 13

## Debugger gdb

`gdb` è un debugger a linea di comando che ci permette di eseguire un programma passo passo, seguendo lo stato del processore e della memoria.

Il concetto fondamentale per un debugger è quello di *breakpoint*, ossia un punto del codice dove l'esecuzione dovrà fermarsi. I breakpoints ci permettono di eseguire rapidamente le parti del programma che non sono di interesse e fermarsi a osservare solo le parti che ci interessano.

Quella che segue è comunque una presentazione sintetica e semplificata. Per altre opzioni e funzionalità del debugger, vedere la documentazione ufficiale o il comando `help`.

### 13.1 Controllo dell'esecuzione

Per istruzione corrente si intende *la prossima da eseguire*. Quando il debugger si ferma a un'istruzione, si ferma *prima* di eseguirla.

Nome completo	Nome scorciatoia	Formato	Comportamento
<code>frame</code>	<code>f</code>	<code>f</code>	Mostra l'istruzione corrente.
<code>list</code>	<code>l</code>	<code>l</code>	Mostra il sorgente attorno all'istruzione corrente.
<code>break</code>	<code>b</code>	<code>b label</code>	Imposta un breakpoint alla prima istruzione dopo <i>label</i> .
<code>continue</code>	<code>c</code>	<code>c</code>	Prosegue l'esecuzione del programma fino al prossimo breakpoint.
<code>step</code>	<code>s</code>	<code>s</code>	Esegue l'istruzione corrente, fermandosi immediatamente dopo. Se l'istruzione corrente è una <code>call</code> , l'esecuzione si fermerà alla prima istruzione del sottoprogramma chiamato.
<code>next</code>	<code>n</code>	<code>n</code>	Esegue l'istruzione corrente, fermandosi all'istruzione successiva del sottoprogramma corrente. Se l'istruzione corrente è una <code>call</code> , l'esecuzione si fermerà <i>dopo</i> il <code>ret</code> di del sottoprogramma chiamato. Nota: aggiungere una <code>nop</code> dopo ogni <code>call</code> prima di una nuova <i>label</i> .
<code>finish</code>	<code>fin</code>	<code>fin</code>	Continua l'esecuzione fino all'uscita dal sottoprogramma corrente ( <code>ret</code> ). L'esecuzione si fermerà alla prima istruzione dopo la <code>call</code> .
<code>run</code>	<code>r</code>	<code>r</code>	Avvia (o riavvia) l'esecuzione del programma. Chiede conferma.
<code>quit</code>	<code>q</code>	<code>q</code>	Esce dal debugger. Chiede conferma.

I seguenti comandi sono *definiti ad hoc nell'ambiente del corso*, e non sono quindi tipici comandi di `gdb`.

Nome completo	Nome scorciatoia	Formato	Comportamento
<code>rrun</code>	<code>rr</code>	<code>rr</code>	Avvia (o riavvia) l'esecuzione del programma, senza chiedere conferma.
<code>qquit</code>	<code>qq</code>	<code>qq</code>	Esce dal debugger, senza chiedere conferma.

### 13.1.1 Problemi con next

Si possono talvolta incontrare problemi con il comportamento di `next`, che derivano da come questa è definita e implementata. Il comando `next` distingue i *frame* come le sequenze di istruzioni che vanno da una label alla successiva. Il suo comportamento è, in realtà, di continuare l'esecuzione finché non incontra di nuovo una nuova istruzione nello stesso *frame* di partenza.

Questa logica può essere facilmente rotta con del codice come il seguente, dove *non esiste* una istruzione di `punto_1` che viene incontrata dopo la `call`. Quel che ne consegue è che il comando `next` si comporta come `continue`.

```
punto_1:
...
    call newline
punto_2:
...
```

Per ovviare a questo problema, è una buona abitudine quella di aggiungere una `nop` dopo ciascuna `call`. Tale `nop`, appartenendo allo stesso *frame* `punto_1`, farà regolarmente sospendere l'esecuzione.

```
punto_1:
...
    call newline
    nop
punto_2:
...
```

## 13.2 Ispezione dei registri

Nome completo	Nome scorciatoia	Formato	Comportamento
info registers	i r	i r	Mostra lo stato di (quasi) tutti i registri. Non mostra separatamente i sotto-registri, come %ax.
info registers	i r	i r reg	Mostra lo stato del registro <i>reg</i> specificato. <i>reg</i> va specificato in minuscolo senza caratteri preposti, per esempio i r eax. Si possono specificare anche sotto-registri, come %ax, e più registri separati da spazio.

`gdb` supporta viste alternative con il comando `layout` che mettono più informazioni a schermo. In particolare, `layout regs` mostra l'equivalente di `i r` e `l`, evidenziando gli elementi che cambiano ad ogni step di esecuzione.

## 13.3 Ispezione della memoria

Nome completo	Nome scorciatoia	Formato	Comportamento
x	x	x/ N FU addr	Mostra lo stato della memoria a partire dall'indirizzo <i>addr</i> , per le <i>N</i> locazioni di dimensione <i>U</i> e interpretate con il formato <i>F</i> . Comando con memoria, i valori di <i>N</i> , <i>F</i> e <i>U</i> possono essere omessi (insieme allo / ) se uguali a prima.

Il comando `x` sta per *examine memory*, ma differenza degli altri non ha una versione estesa.

Il parametro *N* si specifica come un numero intero, il valore di default (all'avvio di `gdb`) è 1.

Il parametro *F* può essere

- `x` per esadecimale
- `d` per decimale
- `c` per ASCII
- `t` per binario
- `s` per stringa delimitata da `0x00`

Il valore di default (all'avvio di `gdb`) è `x`.

Il parametro `U` può essere

- `b` per byte
- `h` per word (2 byte)
- `w` per long (4 byte)

Il valore di default (all'avvio di `gdb`) è `h`.

L'argomento `addr` può essere espresso in diversi modi, sia usando label che registri o espressioni basate su aritmetica dei puntatori. Per esempio:

- letterale esadecimale: `x 0x56559066`
- label: `x &label`
- registro puntatore: `x $esi`
- registro puntatore e registro indice: `x (char*)$esi + $ecx`

Notare che nell'ultimo caso, dato che ci si basa su aritmetica dei puntatori, il tipo all'interno del cast determina la *scala*, ossia la dimensione di ciascuna delle `$ecx` locazioni del vettore da saltare. Si può usare `(char*)` per 1 byte, `(short*)` per 2 byte, `(int*)` per 4 byte.

Un'alternativa a questo è lo scomporre, anche solo temporaneamente, le istruzioni con indirizzamento complesso. Per esempio, si può sostituire `movb (%esi, %ecx), %al` con `lea (%esi, %ecx), %ebx` seguita da `movb (%ebx), %al`, così che si possa eseguire semplicemente `x $ebx` nel debugger.

## 13.4 Gestione dei breakpoints

Oltre a crearli, i breakpoint possono anche essere rimossi o (dis)abilitati. Questi comandi si basano sulla conoscenza dell' *id* di un breakpoint: questo viene stampato quando un breakpoint viene creato o raggiunto durante l'esecuzione, oppure si possono ristampare tutti usando `info b`.

Nome completo	Nome scorciatoia	Formato	Comportamento
<code>info breakpoints</code>	<code>info b</code>	<code>info b [ id ]</code>	Stampa informazioni sul breakpoint <i>id</i> , o tutti se l'argomento è omissso.
<code>disable breakpoints</code>	<code>dis</code>	<code>dis [ id ]</code>	Disabilita il breakpoint <i>id</i> , o tutti se l'argomento è omissso.
<code>enable breakpoints</code>	<code>en</code>	<code>en [ id ]</code>	Abilita il breakpoint <i>id</i> , o tutti se l'argomento è omissso.
<code>delete breakpoints</code>	<code>d</code>	<code>d [ id ]</code>	Rimuove il breakpoint <i>id</i> , o tutti se l'argomento è omissso.

### 13.4.1 Conditional Breakpoints

In alcuni casi, la complessità del programma, l'uso intensivo di sottoprogrammi o lunghi loop possono rendere molto lungo trovare il punto giusto dell'esecuzione. A questo scopo, è possibile definire dei *breakpoint condizionali*, per far sì che l'esecuzione si interrompa a tale breakpoint solo se la condizione è verificata.

Nome completo	Nome scorciatoia	Formato	Comportamento
<code>condition</code>	<code>cond</code>	<code>cond id cond</code>	Imposta la condizione <i>cond</i> per il breakpoint <i>id</i> .

La sintassi per una condizione è in "stile C", come il comando `x`. Alcuni esempi di questa sintassi:

- `cond 2 $al==5` per far sì che l'esecuzione si fermi al breakpoint 2 solo se il registro `al` contiene il valore 5;
- `cond 2 (short *)$edi== -5` per far sì che l'esecuzione si fermi al breakpoint 2 solo se il registro `edi` contiene l'indirizzo di una word di valore -5;
- `cond 2 (int *)&count!=0` per far sì che l'esecuzione si fermi al breakpoint 2 solo se la locazione di 4 byte a partire da `count` contiene un valore diverso da 0.

Fare attenzione alle conversioni automatiche di rappresentazione: quando si usa la rappresentazione decimale, `gdb` interpreta automaticamente i valori come interi. Una condizione come `cond 2 $al==128`, per quanto

accettata dal debugger, sarà sempre falsa perché la codifica `0x80` è interpretata in decimale come l'intero -128, mai come il naturale 128. È quindi una buona idea usare la notazione esadecimale in casi del genere, cioè quando il bit più significativo è 1.

Una feature disponibile in molti IDE è quello di creare dipendenze tra breakpoint, cioè abilitare un breakpoint solo se è stato prima colpito un altro. Questo però è **fin troppo ostico** da fare in gdb.

### 13.4.2 Watchpoints

I watchpoint sono come dei breakpoint ma per dati (registri e memoria), non per il codice. Si creano indicando l'espressione del dato da controllare. Si gestiscono *con gli stessi comandi per i breakpoint*.

Nome completo	Nome scorciatoia	Formato	Comportamento
watchpoint	watch	watch <i>expr</i>	Imposta un watchpoint per l'espressione <i>expr</i> .
info watchpoints	info wat	info wat [ <i>id</i> ]	Stampa informazioni sul watchpoint <i>id</i> , o tutti se l'argomento è omissso.
disable breakpoints	dis	dis [ <i>id</i> ]	Disabilita il breakpoint o watchpoint <i>id</i> , o tutti se l'argomento è omissso.
enable breakpoints	en	en [ <i>id</i> ]	Abilita il breakpoint o watchpoint <i>id</i> , o tutti se l'argomento è omissso.
delete breakpoints	d	d [ <i>id</i> ]	Rimuove il breakpoint o watchpoint <i>id</i> , o tutti se l'argomento è omissso.

Un watchpoint richiede la specifica di un registro o locazione nella stessa notazione “stile C” del comando `x`, e interrompe l'esecuzione quando tale valore cambia. Per esempio, `watch $eax` crea un watchpoint che interrompe l'esecuzione ogni volta che `eax` cambia valore.



## Capitolo 14

# Tabella ASCII

Dalla tabella seguente sono esclusi caratteri non stampabili che non sono di nostro interesse.

Codifica binaria	Codifica decimale	Codifica esadecimale	Carattere
0000 0000	00	0x00	\0
0000 1000	08	0x08	backspace
0000 1001	09	0x09	\t, Horizontal Tabulation
0000 1010	10	0x0A	\n, Line Feed
0000 1101	13	0x0D	\r, Carriage Return
0010 0000	32	0x20	space
0010 0001	33	0x21	!
0010 0010	34	0x22	"
0010 0011	35	0x23	#
0010 0100	36	0x24	\$
0010 0101	37	0x25	%
0010 0110	38	0x26	&
0010 0111	39	0x27	'
0010 1000	40	0x28	(
0010 1001	41	0x29	)
0010 1010	42	0x2A	*
0010 1011	43	0x2B	+
0010 1100	44	0x2C	,
0010 1101	45	0x2D	-
0010 1110	46	0x2E	.
0010 1111	47	0x2F	/
0011 0000	48	0x30	0
0011 0001	49	0x31	1
0011 0010	50	0x32	2
0011 0011	51	0x33	3
0011 0100	52	0x34	4
0011 0101	53	0x35	5
0011 0110	54	0x36	6
0011 0111	55	0x37	7
0011 1000	56	0x38	8
0011 1001	57	0x39	9
0011 1010	58	0x3A	:
0011 1011	59	0x3B	;
0011 1100	60	0x3C	<
0011 1101	61	0x3D	=
0011 1110	62	0x3E	>
0011 1111	63	0x3F	?
0100 0000	64	0x40	@
0100 0001	65	0x41	A
0100 0010	66	0x42	B

Codifica binaria	Codifica decimale	Codifica esadecimale	Carattere
0100 0011	67	0x43	C
0100 0100	68	0x44	D
0100 0101	69	0x45	E
0100 0110	70	0x46	F
0100 0111	71	0x47	G
0100 1000	72	0x48	H
0100 1001	73	0x49	I
0100 1010	74	0x4A	J
0100 1011	75	0x4B	K
0100 1100	76	0x4C	L
0100 1101	77	0x4D	M
0100 1110	78	0x4E	N
0100 1111	79	0x4F	O
0101 0000	80	0x50	P
0101 0001	81	0x51	Q
0101 0010	82	0x52	R
0101 0011	83	0x53	S
0101 0100	84	0x54	T
0101 0101	85	0x55	U
0101 0110	86	0x56	V
0101 0111	87	0x57	W
0101 1000	88	0x58	X
0101 1001	89	0x59	Y
0101 1010	90	0x5A	Z
0101 1011	91	0x5B	[
0101 1100	92	0x5C	\
0101 1101	93	0x5D	]
0101 1110	94	0x5E	^
0101 1111	95	0x5F	_
0110 0000	96	0x60	`
0110 0001	97	0x61	a
0110 0010	98	0x62	b
0110 0011	99	0x63	c
0110 0100	100	0x64	d
0110 0101	101	0x65	e
0110 0110	102	0x66	f
0110 0111	103	0x67	g
0110 1000	104	0x68	h
0110 1001	105	0x69	i
0110 1010	106	0x6A	j

Codifica binaria	Codifica decimale	Codifica esadecimale	Carattere
0110 1011	107	0x6B	k
0110 1100	108	0x6C	l
0110 1101	109	0x6D	m
0110 1110	110	0x6E	n
0110 1111	111	0x6F	o
0111 0000	112	0x70	p
0111 0001	113	0x71	q
0111 0010	114	0x72	r
0111 0011	115	0x73	s
0111 0100	116	0x74	t
0111 0101	117	0x75	u
0111 0110	118	0x76	v
0111 0111	119	0x77	w
0111 1000	120	0x78	x
0111 1001	121	0x79	y
0111 1010	122	0x7A	z
0111 1011	123	0x7B	{
0111 1100	124	0x7C	
0111 1101	125	0x7D	}
0111 1110	126	0x7E	~

From <https://en.wikipedia.org/wiki/ASCII>

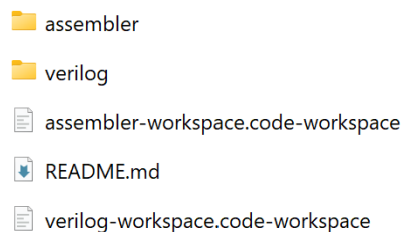
## Capitolo 15

# Ambiente d'esame e i suoi script

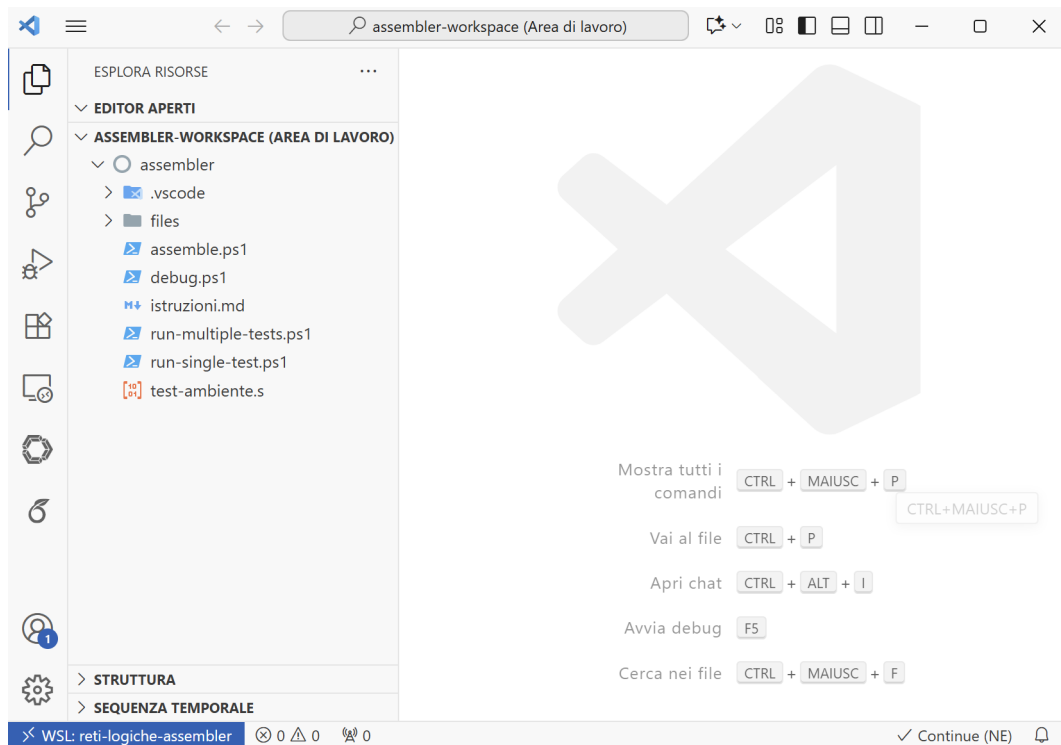
Qui di seguito sono documentati gli script dell'ambiente. I principali sono `assemble.ps1` e `debug.ps1`, il cui uso è mostrato nelle esercitazioni. Gli script `run-test.ps1` e `run-tests.ps1` sono utili per automatizzare i test, il loro uso è del tutto opzionale.

### 15.1 Aprire l'ambiente

Sulle macchine all'esame (o sulla propria, se si seguono tutti i passi indicati nel pacchetto di installazione) troverete una cartella `C:/reti_logiche` con contenuto come da figura.



Facendo doppio click sul file `assembler-workspace.code-workspace` verrà lanciato VS Code, collegandosi alla macchina virtuale WSL e la cartella di lavoro `C:/reti_logiche/assembler`. La finestra VS Code che si aprirà sarà simile alla seguente.



Nell'angolo in basso a sinistra, WSL: reti-logiche-assembler sta a indicare che l'editor è correttamente connesso alla macchina virtuale.

I file e cartelle mostrati nell'immagine sono quelli che ci si deve aspettare dall'ambiente vuoto.

In caso si trovino file in più all'esame, si possono *cancellare*.

Il file test-ambiente.s è un semplice programma per verificare che l'ambiente funzioni. Il contenuto è il seguente:

```
.include "./files/utility.s"

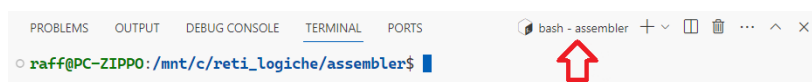
.data
messaggio: .ascii "Ok.\r"

.text
_main:
    nop
    lea messaggio, %ebx
    call outline
    ret
```

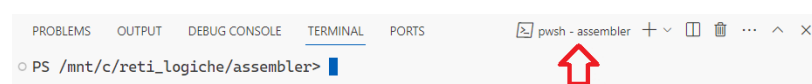
## 15.2 Il terminale Powershell

Per aprire un terminale in VS Code possiamo usare Terminale -> Nuovo Terminale. Per eseguire gli script dell'ambiente c'è bisogno di aprire un terminale *Powershell*. La shell standard di Linux, bash, non è in grado di eseguire questi script.

Non così:



Ma così:



Per cambiare shell si può usare il bottone + sulla sinistra, o lanciare il comando pwsh senza argomenti.

Se si preferisce, in VS Code si può aprire un terminale anche come tab dell'editor, o spostandolo al lato anziché in basso.

#### Perché Powershell?

Perché Powershell (2006) è object-oriented, e permette di scrivere script leggibili e manutenibili, in modo semplice. Bash (1989) è invece text-oriented, con una [lunga lista di trappole da saper evitare](#).

## 15.3 Eseguiare gli script

Gli script forniti permettono di assemblare, debuggare e testare il proprio programma. È importante che vengano eseguiti senza cambiare cartella, cioè non usando il comando `cd` o simili. Ricordarsi anche dei `./`, necessari per indicare al terminale che i file indicati vanno cercati nella cartella corrente.

Il tasto `tab` della tastiera invoca l'autocompletamento, che aiuta ad assicurarsi di inserire percorsi corretti.

Si ricorda inoltre di salvare il file sorgente prima di provare ad eseguire script.

### 15.3.1 assemble.ps1

```
PS /mnt/c/reti_logiche/assembler> ./assemble.ps1 mio_programma.s
```

Questo script assembla un sorgente assembler in un file eseguibile. Lo script controlla prima che il file passato non sia un eseguibile, invece che un sorgente. Poi, il sorgente viene assemblato usando `gcc` ed includendo il sorgente `./files/main.c`, che si occupa di alcune impostazioni del terminale.

### 15.3.2 debug.ps1

```
PS /mnt/c/reti_logiche/assembler> ./debug.ps1 mio_programma
```

Questo script lancia il debugger per un programma. Lo script controlla prima che il file passato non sia un sorgente, invece che un eseguibile. Poi, il debugger `gdb` viene lanciato con il programma dato, includendo le definizioni e comandi iniziali in `./files/gdb_startup`. Questi si occupano di definire i comandi `qquit` e `rrun` (non chiedono conferma), creare un breakpoint in `_main` e avviare il programma fino a tale breakpoint (così da saltare il codice di setup di `./files/main.c`).

### 15.3.3 run-single-test.ps1

```
PS /mnt/c/reti_logiche/assembler> ./run-single-test.ps1 mio_programma input.txt output.txt
```

Lancia un eseguibile usando il contenuto di un file come input, e opzionalmente ne stampa l'output su file. Lo script fa ridirezione di input/output, con alcuni controlli. Tutti i caratteri del file di input verranno visti dal programma come se digitati da tastiera, inclusi i caratteri di fine riga.

### 15.3.4 run-multiple-tests.ps1

```
PS /mnt/c/reti_logiche/assembler> ./run-multiple-tests.ps1 mio_programma cartella_test
```

Testa un eseguibile su una serie di coppie input-output, verificando che l'output sia quello atteso. Stampa riassuntivamente e per ciascun test se è stato passato o meno.

Lo script prende ciascun file di input, con nome nella forma `in_*.txt`, ed esegue l'eseguibile con tale input. Ne salva poi l'output corrispondente nel file `out_*.txt`. Confronta poi `out_*.txt` e `out_ref_*.txt`: il test è passato se i due file coincidono. Nel confronto, viene ignorata la differenza fra le sequenze di fine riga `\r\n` e `\n`.

**Parte IV**

**Assembler - Appendice**

# Capitolo 16

## Problemi comuni

Questa sezione include problemi che è frequente incontrare.

Come regola generale, in sede d'esame rispondiamo a tutte le domande relative a problemi di questo tipo e aiutiamo a proseguire - perché sono relative all'ambiente d'esame e non ai concetti *oggetto* d'esame.

Per altre domande, si può sempre contattare per email o Teams.

### 16.1 Setup dell'ambiente

#### 16.1.1 1. Ho trovato un ambiente assembler per Mac su Github, ma ho problemi ad usarlo

Non abbiamo fatto noi quell'ambiente, non sappiamo come funziona e non offriamo supporto su come usarlo.

#### 16.1.2 2. Ho trovato un ambiente basato su DOS, usato precedentemente all'esame, ma ho problemi ad usarlo

Ha probabilmente incontrato uno dei tanti motivi per cui l'ambiente basato su DOS è stato abbandonato. Questi problemi sono al più *aggirabili*, non *risolvibili*.

#### 16.1.3 3. Lanciando il file `assemble.code-workspace`, mi appare un messaggio del tipo `Unknown distro: Ubuntu`

Il file `assemble.code-workspace` cerca di lanciare via WSL la distro chiamata Ubuntu, senza alcuna specifica di versione. Nel caso la vostra installazione sia diversa, andrà modificato il file. Da un terminale Windows, lanciare `wsl --list -v`, dovreste ottenere una stampa del tipo

```
PS C:\Users\raffa> wsl --list -v
NAME                STATE              VERSION
* Ubuntu             Stopped            2
  Ubuntu-22.04       Stopped            2
```

La parte importante è la colonna `NAME` dell'immagine che vogliamo usare per l'ambiente assembler. Modificare il file `assemble.code-workspace` con un editor di testo (notepad o VS Code stesso, stando attenti ad aprirlo come file di testo e non come workspace) sostituendo tutte le occorrenze di `wsl+ubuntu` con `wsl+NOME-DELLA-DISTRO`. Per esempio, se volessi utilizzare l'immagine `Ubuntu-22.04`, sostituirei con `wsl+Ubuntu-22.04`.

#### 16.1.4 4. Sto utilizzando una sistema Linux desktop, come uso l'ambiente senza virtualizzazione?

Il file `assemble.code-workspace` fa tre cose

- Aprire VS Code nella macchina virtuale WSL
- Aprire la cartella `C:/reti_logiche/assembler` in tale ambiente
- Impostare `pwsh` come terminale default

È possibile fare manualmente gli step 2 e 3, o modificare `assemble.code-workspace` per non fare lo step 1. Per seguire questa seconda opzione, eliminare la riga con `"remoteAuthority":`, e modificare il percorso dopo `"uri":` perché sia semplicemente un percorso sul proprio disco, per esempio `"uri": "/home/raff/reti_logiche/assembler"`.

## 16.2 Uso dell'ambiente

### 16.2.1 5. Se premo *Run* su VS Code non viene lanciato il programma

Non è così che si usa l'ambiente di questo corso. Si deve usare un terminale, assemblare con `./assemble.ps1 programma.s` e lanciare con `./programma`.

### 16.2.2 6. Provando a lanciare `./assemble.ps1 programma.s` ricevo un errore del tipo `./assemble.ps1: line 1: syntax error near unexpected token`

State usando la shell da terminale sbagliata, `bash` invece che `pwsh`. Aprire un terminale Powershell da VS Code o utilizzare il comando `pwsh`.

### 16.2.3 7. Provando ad assemblare ricevo un warning del tipo `warning: creating DT_TEXTREL in a PIE`

Sostituire il file `assemble.ps1` con quello contenuto nel pacchetto più recente tra i file del corso. Oppure modificare manualmente il file, alla riga 29, da

```
gcc -m32 -o ...
```

```
a
```

```
gcc -m32 -no-pie -o ...
```

Riprovare quindi a riassemblare. Se il warning non sparisce, scrivemi. Allegando il sorgente.

### 16.2.4 8. Provando ad assemblare ricevo un warning del tipo `missing .note.GNU-stack section implies executable stack`

Sostituire il file `assemble.ps1` con quello contenuto nel pacchetto più recente tra i file del corso. Oppure modificare manualmente il file, alla riga 29, da

```
gcc -m32 -no-pie -o ...
```

```
a
```

```
gcc -m32 -no-pie -z execstack -o ...
```

Riprovare quindi a riassemblare. Se il warning non sparisce, scrivemi. Allegando il sorgente.

### 16.2.5 9. Ho modificato il codice per correggere un errore, ma quando assemblo e eseguo il codice, continuo a vedere lo stesso errore.

Controllare di aver salvato il file. In alto, nella barra delle tab, VS Code mostra un pallino pieno, al posto della X per chiedere la tab, per i file modificati e non salvati.

### 16.2.6 10. Dove trovo i file che scrivo nell'ambiente assembler?

La cartella `assembler` mostrata in VS Code corrisponde alla cartella `C:/reti_logiche/assembler` su Windows. Troveremo qui sia i file sorgenti (estensione `.s`) che i binari assemblati.



**Windows può nascondere le estensioni dei file**

Nella configurazione default, Windows nasconde le estensioni dei file “noti”. Suggerisco di cambiare questa configurazione per mostrare sempre l'estensione, come indicato [qui](#).