Note esercitazioni Assembler

Raffaele Zippo

10 ottobre 2022

Indice

1 Introduzione ambiente di sviluppo

In questo corso, programmeremo assembly usando la sintassi GAS (anche nota come AT&T). Per assemblare e debuggare questi programmi, utilizzeremo degli script `assemble` e `debug`. Questi script non fanno che invocare, con gli opportuni parametri, rispettivamente `gcc` e `gdb`.

L'ambiente è fornito in due versioni, DOS e Linux. *Normalmente*, questi sistemi operativi sono molto diversi e programmi scritti per uno non funzionano per l'altro. *In questo caso*, i file forniti si occupano di rendere omogeneo il comportamento in modo che siano intercambiabili.
Quindi, quale che sia l'ambiente utilizzato per esercitarsi il codice da scrivere sarà lo stesso.

Nota: non ho Mac, e l'ambiente non è quindi regolarmente testato su Mac.
Le istruzioni relative a Mac, sia Intel che M1, sono basati su esperienze e indicazioni condivise da studenti, che ringrazio tutti.

2 Ambiente DOS

Questo ambiente si basa su DOSBox, un emulatore del sistema operativo DOS. L'ambiente include i binari per DOS di gcc e gdb, e i binari DOSBox per Windows.

L'ambiente è fornito in un file .zip che, estratto nella posizione che si preferisce, avrà la seguente struttura:

- files
  - DOSBox
  - GAS
  - utility.s
- ASSEMBLE.BAT
- DEBUG.BAT
- runDosBox.bat
- CWSDPMI.EXE
- dosbox.conf

Fare doppio click su runDosBox.bat (o lanciarlo da terminale) per avviare DOSBoX. Questo script si occupa di lanciare l'emulatore e fargli montare la cartella corrente come C:\.
Per scrivere il codice, si può usare un qualunque editor di testo. Durante le esercitazioni, verrà mostrato Visual Studio Code. L'editor dovrà girare in Windows, e modificare file sorgente nella cartella root. I sorgenti possono essere salvati sia nella root dell'ambiente che in sottocartelle, ma è bene *evitare* di spostarsi (comando cd) dalla root del progetto.
Gli script forniti sono

- ASSEMBLE.BAT, da eseguire in DOS per assemblare il proprio codice. Esempio:

  ```
  C:\> ASSEMBLE.BAT FILE.S
  ```

  Questo produrrà, se non ci sono errori, l'eseguibile FILE.EXE e l'output dell'assemblatore in LISTATO.TXT

- DEBUG.BAT, da eseguire in DOS per debuggare l'eseguibile. Esempio:

  ```
  C:\> DEBUG.BAT FILE.EXE
  ```

  Questo avvierà gdb.

Gli script, così come i programmi assemblati, vanno avviati da linea di comando DOS, nella finestra di DOSBoX. Non è possibile utilizzare il terminale integrato di VSCode.

2.1 Problemi noti

- Per via delle versioni datate dei software per DOS, può capitare che l'assemblatore corrompa il file sorgente. Si consiglia di stare attenti a come viene lasciato il file `.S` dopo aver chiamato `ASSEMBLE.BAT`, e tenere lasciato aperto l'editor di testo per poter annullare le eventuali modifiche indesiderate.
- Sempre per via della versione datata, alcune feature di `gdb` non sono presenti:
  - non è presente una cronologia dei comandi, che ci permette (su Linux) di ripetere i comandi precedenti usando i tasti freccia invece di riscriverli daccapo.
  - il comando `info register` non mostra registri da meno di 32 bit, come per esempio `info register cl`.
  - il comando `info register` non mostra il registro `eflags` come set di flag a 1.
  - alcuni comandi possono portare ad errori interni, per i quali `gdb` suggerisce di terminare il debugging. In tal caso, si può *provare* a continuare, ma non è possibile dire quanto questi errori influiscano sulla normale esecuzione del programma che si sta debuggando.
- I nomi di cartelle sono limitati a 8 caratteri, mentre quelli di file a 8 caratteri più 3 di estensione. Per i nomi di più di 8 caratteri, la parte finale è troncata con il carattere `~`. Per esempio `FILE12345.S` diventa `FILE123~.S`. Utilizzare l'autocompletamento (tasto TAB della tastiera) per essere sicuri di fornire path validi.
- Il terminale `DOSBoX` non permette di cambiare il numero di caratteri interno, scorrere le righe per vedere l'output precedente, copiare o incollare del testo.
  Per quanto riguarda la *dimensione* della finestra di DOSBoX, è possibile indicare la risoluzione desiderata modificando `dosbox.conf`. La risoluzione interna di DOS rimarrà comunque 400x300.

2.2 DOSBox su altri sistemi operativi

L'emulatore DOSBox è disponibile anche per Linux e Mac (sia Intel che M1). Una volta installato DOSBox, per usare questo ambiente, bisognerà estrarre tutti i file in dos.zip eccetto per la cartella `DOSBox` (che contiene DOSBox per Windows) e convertire lo script `runDosBox.bat` nell'equivalente per la propria piattaforma.
Una volta avviato correttamente DOSBox, dovrebbe essere possibile utilizzare gli script e i programmi dell'ambiente così come descritto per Windows. Allo stesso modo, i file sorgenti andranno editati con un editor nativi per la propria piattaforma.

È sempre possibile, anche se inutilmente più complesso, utilizzare una macchina virtuale con Windows ed utilizzare direttamente il contenuto del pacchetto dos.zip

3 Ambiente Linux

Questo ambiente si basa direttamente su Linux. Può essere utilizzato direttamente su una macchina con Linux, o con una macchina virtuale. Per le esercitazioni, verrà utilizzato Ubuntu 20.04 virtualizzato in WSL2.

L'ambiente non include tutti i binari. Su Ubuntu 20.04, si dovranno installare:

- build-essential
- gcc-multilib
- musl-dev
- gdb
- powershell

L'ambiente è fornito in un file `.zip` che, estratto nella posizione che si preferisce, avrà la seguente struttura:

- 📁`files`
  - `gdb_startup`
  - `main.c`
  - `utility.s`
- `assemble.ps1`
- `debug.ps1`
- `test.ps1`

Per utilizzare l'ambiente basterà aprire un terminale Powershell[1] nella cartella root dell'ambiente. Per scrivere il codice, si può usare un qualunque editor di testo. Durante le esercitazioni, verrà mostrato Visual Studio Code. I sorgenti possono essere salvati sia nella root dell'ambiente che in sottocartelle, ma è bene *evitare* di spostarsi (comando `cd`) dalla root del progetto.

Gli script forniti sono

- `assemble.ps1`, da eseguire per assemblare il proprio codice. Esempio:

  ```
  > ./assemble.ps1 esercizio.s
  ```

  Questo produrrà, se non ci sono errori, l'eseguibile `esercizio` e l'output dell'assemblatore in `esercizio.lst`

- `debug.ps1`, da eseguire per debuggare l'eseguibile. Esempio:

  ```
  > ./debug.ps1 esercizio
  ```

  Questo avvierà `gdb`.

- `test.ps1`, di utilità per fare test con file di input e (opzionalmente) salvare l'output su file

[1] Per cambiare shell, basta eseguire il comando `pwsh` senza argomenti

```
> ./test.ps1 esercizio input.txt

> ./test.ps1 esercizio input.txt output.txt
```

Gli script controllano che i file passati siano testo od eseguibili, per evitare errori comuni con l'autocompletamento.

```
> ./assemble.ps1 esercizio
The file is an executable
> ./debug.ps1 esercizio.s
The file is not an executable
```

Gli script, così come i programmi assemblati, possono essere avviati da qualsiasi terminale della macchina Linux, purché si utilizzi la shell Powershell[2]. È possibile utilizzare il terminale integrato di VSCode.

3.1 Altri sistemi operativi

Se non si ha una macchina con Linux, il setup più comodo ed efficiente sarà sicuramente l'uso di una macchina virtuale (anche senza GUI) combinata con l'editing in remoto di Visual Studio Code. Questo ci permette di avere l'editor nativo per la propria piattaforma, mentre i file e i comandi eseguiti saranno nella macchina virtuale.
Avremo bisogno di:

1. Una macchina virtuale che ci permetta di installare ed avviare Linux x86/x64
2. Installare Ubuntu 20.04 (questa è la distro testata), opzionalmente la versione server che non include GUI
3. Installare e configurare ssh-server per connettersi via ssh dalla piattaforma host
4. Usare il remoting ssh di VS Code per editare in remoto sulla macchina virtuale

Su Windows o Mac Intel, l'opzione più comune è VirtualBox.
Su Windows 10, abbiamo l'opzione aggiuntiva di WSL[3], che è più facile da configurare e ci permette di saltare la configurazione di ssh, usando invece il remoting WSL di VS Code.
Su Mac M1, UTM permette di selezionare l'architettura CPU da emulare (selezionare x64). Questo sarà molto meno efficiente della normale emulazione, ma è necessario dato che il codice assembly è, ovviamente, strettamente legato all'architettura CPU.

[2] Se compare un errore del tipo `./assemble.ps1: line 1: syntax error near unexpected token`, siete nella shell sbagliata. Usare il comando `pwsh` per cambiare shell.
[3] Assicurarsi di utilizzare WSL v2, non v1. Quest'ultima non può eseguire programmi a 32 bit

4 Differenze tra gli ambienti

Le differenze principali riguardano l'usabilità e la facilità di configurazione, per il quale si lascia alla scelta personale, soprattutto per le esercitazioni a casa. Dal punto di vista del codice da scrivere, gli ambienti si comportano allo stesso modo: si utilizzano le stesse sintassi, le stesse primitive di input/output, gli stessi caratteri di terminazione riga[1].

4.1 File `utility.s`

Entrambi gli ambienti sono *portabili*, cioè i rispettivi file possono essere salvati dove si preferisce. Per utilizzare i sottoprogrammi di ingresso/uscita, si utilizzerà

```
.INCLUDE "./files/utility.s"
```

Il percorso è relativo alla *root* dell'ambiente, cioè la cartella che conterrà gli script `assemble` e `debug`. È importante che sia da questa cartella che si chiamino gli script.

4.2 Meccanismi di protezione e `segmentation fault`

Marcare le sezioni `.data` e `.text` potrebbe risultare opzionale in DOS, dato che la distinzione non viene applicata dal sistema. Nell'ambiente Linux la distinzione è invece applicata: se non si dichiarano opportunamente le sezioni, il programma potrebbe essere terminato con `segmentation fault`.

4.3 Differenze con l'ambiente degli anni precedenti

Le uniche cose che cambiano rispetto all'ambiente degli anni precedenti sono

- Il percorso del file di utility, che ora va incluso con `.INCLUDE "./files/utility.s"`
- Le sezioni `.data` e `.text`, che, se non dichiarate, possono causare terminazione su Linux

Questa classe di errori non verrà considerata ai fini della valutazione.

[1] In particolare, l'ambiente fa sì che Linux si comporti allo stesso modo di DOS.

5 Debugging: utilizzo di `gdb`

Il *debugging* è il processo di ricerca e rimozione degli errori (bug) di un programma. In linguaggi ad alto livello, il primo strumento utilizzato è la stampa su terminale o su file di log, per individuare rapidamente i punti d'errore. Qui invece questo non è altrettanto facile, e dobbiamo affidarci ad un debugger completo, per l'appunto `gdb`.

Il principale scopo del debugger è far eseguire il programma *un passo alla volta*, e permetterci di osservare lo stato dei registri e della memoria a ciascuno di questi passi. Quali sono questi "passi" su cui soffermarsi lo decidiamo noi, definendo dei *breakpoints*: quando il programma giunge ad un breakpoint, il debugger mette in pausa l'esecuzione e lascia a noi il controllo. Possiamo allora, tramite specifici comandi, stampare informazioni sullo stato, proseguire un'istruzione alla volta, far continuare fino al prossimo breakpoint, etc.

5.1 Avvio di `gdb`

La sintassi più semplice è `gdb percorso_eseguibile`.
Lo script di debug nell'ambiente del corso fa dei passi in più, che semplificano l'utilizzo: definisce i comandi `rr` e `qq` (sezione successiva) ed esegue i comandi per mettere un breakpoint a `_main` ed avviare l'esecuzione del programma.

Attenzione a quello che `gdb` stampa all'avvio: se la terzultima riga è la seguente conviene fermarsi subito.

```
warning: Source file is more recent than executable.
```

Come dice l'errore, non abbiamo riassemblato il programma dopo le ultime modifiche.

5.2 Comandi per il controllo dell'esecuzione

Per guidare l'esecuzione del programma si usano i seguenti comandi:[1]

- **f**rame
  Mostra la posizione attuale del programma, ossia l'istruzione che sta per essere eseguita e la riga a cui corrisponde nel file sorgente.
- **b**reak *label*
  Inserisce un breakpoint alla posizione indicata da *label*. L'esecuzione del programma verrà quindi messa in pausa prima di eseguire l'istruzione associata a *label*.
  Se chiamato senza argomento *label*, mostra la lista dei breakpoints inseriti.
- **d**elete **br**eak *label*
  Rimuove il breakpoint alla posizione indicata da *label*.
- **c**ontinue
  Prosegue l'esecuzione fino al prossimo breakpoint.

[1] I caratteri evidenziati in **grassetto** sono il minimo necessario perché `gdb` riconosca il comando. Non c'è quindi bisogno di scriverli per intero.

- **s**tep
  Esegue una singola istruzione.
  Attenzione se questa è una `CALL`: il debugger si fermerà *dentro* il sottoprogramma chiamato.

- **fin**ish
  Prosegue l'esecuzione del sottoprogramma corrente (sia *f*) finché non termina (`RET`). L'esecuzione del programma verrà quindi messa in pausa prima di eseguire l'istruzione successiva all'istruzione `CALL f`.

- **n**ext
  Continua l'esecuzione fino alla successiva istruzione di questo file.
  Questo significa che si comporta come *step*, tranne quando l'istruzione da eseguire è una `CALL`. In tal caso, il sottoprogramma viene eseguito senza pause.

- **r**un
  Avvia l'esecuzione del programma. Se il programma è già in esecuzione, dopo aver chiesto conferma all'utente, riavvia il programma dall'inizio.

- **q**uit
  Termina il debugger. Se il programma è ancora in esecuzione, chiede prima conferma all'utente.

5.2.1 Comandi aggiuntivi

I seguenti comandi non fanno parte dei comandi standard di `gdb`. Sono invece comandi personalizzati definiti all'avvio dallo script di debug fornito nell'ambiente, aggiunti allo scopo di semplificare i casi d'uso più comuni.

- **rr**un
  Riavvia il programma, senza chiedere conferma.

- **qq**uit
  Termina il debugger, senza chiedere conferma.

5.3 Comandi per ispezionare lo stato del programma

Oltre a controllare il percorso seguito dal programma, è ovviamente utile controllare lo stato di registri e memoria prima e dopo l'esecuzione delle istruzioni di interesse. Per far questo useremo i seguenti comandi:

- **i**nfo **r**egister *registro*
  L'argomento *registro* è opzionale, e deve essere in minuscolo e senza caratteri preposti: *eax*, *bx*, *cl*.
  Se specificato, mostra il contenuto del registro, prima in esadecimale e poi in un formato che dipende dal tipo di registro:[2]

  - decimale per registri accumulatori
  - label+offset per *eip*
  - lista dei flag a 1 per *eflags*

  Se non specificato, farà quanto sopra per tutti i registri.

- **x**/*NFU indirizzo*
  La x sta per *examine memory*, in questo caso non è un'abbreviazione e non si può usare una versione più lunga.
  Notiamo intanto che ci sono diversi argomenti: numero (*N*), formato (*F*), dimensione

[2] Purtroppo, questo comportamento non può essere modificato. Questo significa, per esempio, che non possiamo mostrare il contenuto di *al* come carattere ASCII.

(*U*) e indirizzo. Sono tutti *opzionali*, perché questo è un comando *con memoria*. Verranno infatti ricordati gli ultimi argomenti ed utilizzati come valori di default, ad eccezione di *N* il cui default è sempre 1. Per evitare confusioni, si consiglia comunque di specificare sempre tutto.

L'argomento *indirizzo* indica la locazione da cui *iniziare*. Questo può essere indicato:

- in esadecimale: `x 0x56559066`
- tramite label preceduta da `&`: `x &array`
- tramite registro preceduto da `$`: `x $esi`

L'argomento *N* indica il numero di locazioni da accedere. Questo è un semplice numero decimale. Se negativo, le locazioni saranno mostrate andando all'indietro.

L'argomento *F* indica il formato[3] con cui interpretare il contenuto delle locazioni, e quindi come va stampato a schermo:

- `x` esadecimale
- `d` decimale
- `c` ASCII
- `t` binario
- `s` stringa delimitata da `0x00`[4]

L'argomento *U* indica la dimensione[5] delle locazioni da accedere.

- `b` 1 byte
- `h` word, 2 byte
- `w` long, 4 byte

5.3.1 Espressioni indirizzo complesse

Si prenda il seguente esempio

```
array:  .WORD 1, 256, 512
...
MOV $0, %ESI
...
CMPW array(,%ESI,2), %AX
```

Si potrebbe voler controllare quale valore viene effettivamente confrontato con `%AX`. Abbiamo due metodi a disposizione.

Il primo metodo sfrutta la `LEA` per **scomporre l'istruzione** in modo da avere l'indirizzo calcolato in un registro d'appoggio.

```
array:  .WORD 1, 256, 512
...
MOV $0, %ESI
...
LEA array(,%ESI,2), %EBX
CMPW (%EBX), %AX
```

[3] Molti formati sono stati qui omessi perché, a noi, poco utili. Usare il comando `help x` per una lista esaustiva

[4] La `inline` può lasciare il buffer senza `0x00` come terminazione. Allocare un byte in più se si vuole utilizzare questo comando.

[5] Le sigle sono diverse dal GAS perché diversa è la definizione di "word": h sta per halfword (per noi word), mentre w sta per word (per noi long).

A questo punto basterà usare il comando `x/dh $ebx` per stampare il valore desiderato.

Il secondo metodo sfrutta una sintassi più complessa del comando `x`, che calcola l'indirizzo da accedere in modo simile (ma sintatticamente del tutto diverso) a quanto fatto dall'indirizzamento base-indice-scala.

- `array(,%ESI,1)` →`(char*)&array+$esi`
- `array(,%ESI,2)` →`(short*)&array+$esi`
- `array(,%ESI,4)` →`(int*)&array+$esi`

Il *cast* a `tipo*` serve a indicare a `gdb` la dimensione dei dati puntati dall'indirizzo `array`, così che questa venga usata correttamente per la somma successiva in modo simile a quanto succede con l'aritmetica dei puntatori in C.
Si noti che questa sintassi serve solo al calcolo dell'indirizzo: così come il suffisso dell'istruzione in assembly, è l'argomento *U* a determinare quanti byte vengono letti in memoria con il comando `x`. Per l'esempio di cui sopra, si scriverà quindi

```
x/dh (short*)&array+$esi
```