Come usare il Simple Assembly Simulator

Alessandro Bugatti (alessandro.bugatti@unibs.it)

23 settembre 2018

Sommario

Questa breve dispensa ha lo scopo di spiegare l'utilizzo del software di simulazione che può essere utilizzato per testare i programmi scritti in Assembly durante il corso di "Fondamenti di Informatica e Programmazione" che si tiene all'Università degli Studi di Brescia, Facoltà di Ingegneria.

1 Perchè imparare l'assembly

Storicamente l'assembly, o per meglio dire i linguaggi assembly, nascono per permettere ai programmatori di scrivere del codice che non sia una sequenza di istruzioni macchina, ma che associ delle etichette linguistiche alle varie istruzioni, il che, insieme ad altre migliorie, consente agli sviluppatori di scrivere programmi più complessi cone meno fatica.

Essendo comunque un linguaggio di basso livello, in cui le istruzioni sono quasi direttamente traducibili in istruzioni macchina, i programmi scritti in assembly permettono un controllo completo su cosa accade all'interno della macchina. Come si può vedere in figura 1, dove è mostrato un esempio di codice reale, le istruzioni in linguaggio macchina sono di difficile interpretazione, mentre le istruzioni in assembly, pur rimanendo di non semplice interpretazione, offrono sicuramente un'astrazione maggiore, permettendo ad esempio di capire più velocemente che il codice in figura somma il numero 2 con il numero 3.

I linguaggi di tipo assembly sono strettamente legati all'architettura della macchina per cui devono essere scritti i programmi e questo ha portato alla proliferazione di linguaggi concettualmente simili, ma tra loro incompatibili.

Per evitare di appesantire inutilmente l'apprendimento usando un linguaggio assembly reale, ad esempio quello di Intel per la piattaforma x86, si è preferito utilizzare un linguaggio inventato, del tutto verosimile e con un insieme di istruzioni minimale ma sufficiente a comprendere il funzionamento "interno" di un computer, che poi è lo scopo dello studio di questo argomento.

Va infine detto che nella pratica reale il linguaggi assembly sono rimasti appannaggio di alcune nicchie di programmazione (sistemi operativi, driver di periferica, compilatori, ...), ma anche in quei casi l'utilizzo rimane molto limitato in parte a causa della velocità dei computer moderni, che rendono meno rilevante la ricerca di prestazioni spinte utilizzando del codice assembly ottimizzato, dall'altro esistono linguaggi di più alto livello (C, C++, Go, ...) che producono del codice macchina estremamente efficiente senza costringere il programmatore a scrivere in assembly.

2 L'Assembly visto a lezione

Per comprendere le istruzioni che verranno adesso spiegate, può essere utile ripassare alcuni concetti fondamentali visti a lezione:

Address	Machine Language				Assembly Language
0000 0000	0000	0000	0000	0000	TOTAL .BLOCK 1
0000 0001	0000	0000	0000	0010	ABC .WORD 2
0000 0010	0000	0000	0000	0011	XYZ .WORD 3
0000 0011	0001	1101	0000	0001	LOAD REGD, ABC
0000 0100	0001	1110	0000	0010	LOAD REGE, XYZ
0000 0101	0101	1111	1101	1110	ADD REGF, REGD, REGE
0000 0110	0010	1111	0000	0000	STORE REGF, TOTAL
0000 0111	1111	0000	0000	0000	HALT

Figura 1: Esempio di un assembly reale

- i **registri** possono essere visti come delle memorie molto piccole, ma veloci, ognuno dei quali è contraddistinto da un *nome* e può memorizzare al suo interno un *valore*. A parte il nome diverso, ogni registro è funzionalmente equivalente a qualunque altro, a differenza delle architetture reali dove spesso registri diversi fanno cose diverse.
- tutte le operazioni (somme, differenze, confronti, ...) avvengono tra valori contenuti nei registri, **non** possono essere fatte direttamente nella memoria
- la **memoria** è un insieme di "caselle" consecutive, ognuna contraddistinta da un *indirizzo*, che servono a contenere i dati su cui deve operare il programma (il fatto che contenga anche le istruzioni è del tutto irrilevante per chi scrive il programma)
- da quanto detto sopra un programma, in generale, prenderà dei dati dalla memoria per metterli in uno o più registri (operazione detta di *LOAD*), li elaborerà in modo opportuno e poi sposterà i risultati dai registri alla memoria (operazione detta di *STORE*), ripetendo eventualmente queste operazioni tante volte quante servono.
- l'architettura di questo calcolatore è a 16 bit, cioè ogni dato numerico occupa due byte, quindi due caselle di memoria adiacenti.

Verranno adesso spiegate le istruzioni assembly racchiuse nella tabella 1

Istruzione	Significato	Indirizzamenti
LOADiL RX Val	$RX_{1-8} \leftarrow$ Val	Registro/Immediato con codifica esadecimale
LOADiH RX Val	$RX_{9-16} \leftarrow$ Val	Registro/Immediato con codifica esadecimale
ADD RX RY Val	RX $\leftarrow$ RY $+$ Val	Registro/Registro/Immediato con codifica esadecimale
ADD RX RY RZ	RX $\leftarrow$ RY $+$ RZ	Registro/Registro/Registro
SUB RX RY Val	RX $\leftarrow$ RY $-$ Val	Registro/Registro/Immediato con codifica esadecimale
SUB RX RY RZ	RX $\leftarrow$ RY $-$ RZ	Registro/Registro/Registro
MUL RX RY Val	RX $\leftarrow$ RY $\times$ Val	Registro/Registro/Immediato con codifica esadecimale
MUL RX RY RZ	RX $\leftarrow$ RY $\times$ RZ	Registro/Registro/Registro
DIV RX RY Val	RX $\leftarrow$ RY / Val	Registro/Registro/Immediato con codifica esadecimale
DIV RX RY RZ	RX $\leftarrow$ RY / RZ	Registro/Registro/Registro
MOVE RX RY	RX$\leftarrow$ RY	Registro/Registro
LOAD RX RY	RX$\leftarrow$ M[RY]	Registro/Memoria indiretto rispetto a registro
STORE RX RY	M[RX]$\leftarrow$ RY	Memoria indiretto rispetto a registro/Registro
BLT RX RY label	PC $\leftarrow$ label se RX $<$ RY	Registro/Registro/Etichetta
BLE RX RY label	PC $\leftarrow$ label se RX $\leq$ RY	Registro/Registro/Etichetta
BGT RX RY label	PC $\leftarrow$ label se RX $>$ RY	Registro/Registro/Etichetta
BGE RX RY label	PC $\leftarrow$ label se RX $\geq$ RY	Registro/Registro/Etichetta
BEQ RX RY label	PC $\leftarrow$ label se RX $=$ RY	Registro/Registro/Etichetta
BNE RX RY label	PC $\leftarrow$ label se RX $\neq$ RY	Registro/Registro/Etichetta
B label	PC $\leftarrow$ label	
END		

Tabella 1: Istruzione assembly

2.1 LOADiL e LOADiH

Tutti i programmi che si vedranno hanno la necessità di caricare inizialmente nei registri dei valori che fanno parte dei dati del problemi da risolvere. Solitamente questi dati possono essere o dei numeri o degli indirizzi di memoria, per l'istruzione questo non fa differenza, ovviamente farà differenza il modo in cui saranno usati. Per rendere l'architettura più realisticaa si è supposto che i registri abbiano una dimensione di 2 byte e che per caricarli sia necessario caricare il byte "basso" (con LOADiL, gli 8 bit meno significativi, dall'1 all'8) e il byte "alto" (LOADiH, gli 8 bit più significativi, dal 9 al 16).

La sintassi prevede due parametri dopo il nome dell'istruzione:

- il primo è un nome di registro, quindi $R_n$con $1 \leq n \leq 16$
- il secondo è un valore esadecimale, quindi con il prefisso 0x, che può rappresentare un valore o un indirizzo

Esempio

Problema: *si prenda il valore memorizzato all'indirizzo di memoria 0x480, gli si sommi 7 e il risultato lo si memorizzi nella cella di memoria all'indirizzo 0x500.*

Nel testo di questo problema sono presenti 3 dati:

- l'indirizzo della cella di memoria dove si trova il numero a cui sommare 7
- il numero 7 stesso
- l'indirizzo della cella di memoria dove memorizzare il risultato

Le istruzioni che permetteranno di farlo sono quindi le seguenti:

```
1 LOADiL R1 0x80
2 LOADiH R1 0x04
3 LOADiL R2 0x07
4 LOADiH R2 0x00
5 LOADiL R3 0x00
6 LOADiH R3 0x05
```

Nelle righe 1 - 2 viene caricato il valore 0x480, che rappresenta un indirizzo di memoria, nel registro R1: come si può vedere viene caricata la parte bassa con il primo byte, cioè 0x80 (si ricorda che una cifra esadecimale occupa 4 bit e quindi due cifre esadecimali sono esattamente 1 byte) e poi la parte alta, cioè 0x04. L'ordine con il quale vengono eseguite queste due istruzioni è indifferente.

Nelle righe 3 - 4 viene inserito il numero 7 nel registro R2, sempre con lo stesso meccanismo. Si ponga attenzione al fatto che per l'istruzione è indifferente che il numero caricato rappresenti un valore (come il 7) o un indirizzo (come 0x480), sarà poi l'uso che se ne farà che sarà diverso. Comunque sia i valori che gli indirizzi devono essere dei numeri esadecimali (quindi con il prefisso 0x) per poter essere caricati dal simulatore: nei testi dei problemi, mentre per gli indirizzi si usano sempre dei valori esadecimali, per i valori come il 7 a volte si usa la notazione decimale (come in questo esempio). Al momento della scrittura del programma il numero va comunque convertito in esadecimale per poter essere caricato (nel caso del 7 è ovviamente equivalente a 0x7).

Infine nelle righe 5 - 6 si carica l'indirizzo 0x500 nel registro R3.

2.2 ADD

L'istruzione ADD è un'istruzione di tipo matematico che, come ovvio, somma tra loro due numeri e che è implementata in hardware all'interno della ALU. Qua si parlerà solo di questa istruzione, il funzionamento delle altre funzioni matematiche (SUB, MUL, DIV) è assolutamente lo stesso, a parte l'operazione svolta. Tutte le operazioni matematiche lavorano solo su numeri interi e quindi anche i risultati saranno sempre interi.

La sintassi prevede due forme possibile, entrambe con 3 parametri, con i primi due aventi lo stesso significato:

- il primo parametro è il nome del registro dove verrà memorizzato il risultato dell'operazione
- il secondo parametro è il primo addendo dell'addizione
- il terzo parametro è il secondo addendo dell'addizione e può essere direttamente un numero oppure il numero contenuto all'interno di un registro.

2.2.1 Esempio

Problema: *si sommino i valori contenuti all'interno dei registri R1 e R2 e si memorizzi il risultato nel registro R3, successivamente gli si aggiunga il valore contenuto nel registro R4 e infine lo si incrementi di 1.*

Ignoriamo come siano stati inseriti i valori in R1, R2 e R4, che quindi assumiamo contengano già dei valori, allora il programma sarà il seguente:

```
1 ADD R3 R1 R2
2 ADD R3 R3 R4
3 ADD R3 R3 0x01
```

Alla prima riga vengono presi i valori contenuti in R1 e R2, vengono sommati e successivamente memorizzati in R3, nella seconda riga viene sommato il valore contenuto in R4 a quello già presente in R3: si noti infatti che R3 compare sia in prima che in seconda posizione, il che significa che prima verrà sommato il valore contenuto in R3 con quello in R4 e successivamente il risultato verrà memorizzato in R3, sovrascrivendo il valore precedente.

Infine nella terza riga, utilizzando stavolta un valore come terzo parametro, viene aggiunto 1 al valore contenuto in R3.

Se i valori iniziali contenuti in R1, R2 e R4 fossero stati rispettivamente 12, 4 e 9, dopo aver eseguito le istruzioni precedenti in R3 sarebbe stato memorizzato il valore 26.

2.3 MOVE

Questa istruzione copia i valori contenuti in un registro all'interno di un altro registro.

La sintassi prevede due parametri, entrambi che rappresentano un registro

- il primo parametro è il registro **in** cui verrà copiato il valore
- il secondo parametro è il registro **da** cui verrà copiato il valore

2.3.1 Esempio

Problema: *si copi il valore contenuto in R5 all'interno del registro R2.*

```
1 MOVE R2 R5
```

Come si vede questa istruzione è di immediata comprensione e di semplice utilizzo e non fa altro che copiare il valore che si trova nel registro R5 all'interno del registro R2.

2.4 LOAD

Questa istruzione è una delle due fondamentali (l'altra è STORE) che permette di far dialogare la CPU con la memoria attraverso i registri, in questo caso prelevando un valore da una cella di memoria e copiandolo in un registro.

La sintassi prevede due parametri, ambedue registri:

- il primo è il registro nel quale verrà copiato il valore che si vuole recuperare dalla memoria
- il secondo è un registro che deve contenere l'indirizzo di memoria della cella di memoria da cui si intende copiare il valore

2.4.1 Esempio

Problema: *si copi il valore contenuto in memoria all'indirizzo 0x300 all'interno del registro R1.*

```
1 LOADiL R2 0x00
2 LOADiH R2 0x03
3 LOAD R1 R2
```

Le istruzioni alle righe 1 - 2 servono per caricare nel registro R2 (ma se ne sarebbe potuto anche scegliere un altro) l'indirizzo di memoria che ci interessa, in quanto la cella individuata da quell'indirizzo contiene il dato da utilizzare. A questo punto l'istruzione LOAD caricherà il valore contenuto nella memoria alla locazione 0x300 all'interno del registro R1, in modo che possa essere poi elaborato secondo le richieste del programma.

2.5 STORE

L'istruzione complementare di LOAD è STORE e serve a trasferire un valore che si trova all'interno di un registro verso una cella di memoria individuata dal suo indirizzo.

La sintassi prevede due parametri, ambedue registri:

- il primo è il registro che contiene l'indirizzo di memoria in cui si vuole copiare il dato che interessa
- il secondo è il registro che contiene il dato che si desidera copiare in memoria

2.5.1 Esempio

Problema: *si copi il valore contenuto nel registro R1 all'indirizzo di memoria 0x300.*

```
1 LOADiL R2 0x00
2 LOADiH R2 0x03
3 STORE R2 R1
```

Le istruzioni alle righe 1 - 2 servono per caricare nel registro R2 (ma se ne sarebbe potuto anche scegliere un altro) l'indirizzo di memoria che ci interessa, che quindi individuerà la cella di memoria dove si desidera memorizzare il dato. Alla riga 3 l'istruzione STORE prenderà il dato presente nel registro R1 e lo copierà quindi nella cella 0x300.

3 Un esempio completo

Prima di spiegare il funzionamento delle istruzioni di salto (branch), verrà analizzata la soluzione di un problema semplice ma completo e successivamente si vedrà come eseguire questo programma sul simulatore.

Problema: *si calcoli l'area di un rettangolo, considerando che il valore della base si trova all'indirizzo 0x100, quello dell'altezza all'indirizzo 0x200 e si vuole che il risultato venga scritto in memoria all'indirizzo 0x300.*

Come si può vedere questo problema richiede di applicare tutte le istruzioni viste in precedenza per arrivare alla soluzione, che può essere scritta così:

```
1  LOADiL R1 0x00
2  LOADiH R1 0x01
3  LOADiL R2 0x00
4  LOADiH R2 0x02
5  LOADiL R3 0x00
6  LOADiH R3 0x03
7  LOAD R4 R1
8  LOAD R5 R2
9  MUL R6 R4 R5
10 STORE R3 R6
```

Le prime 6 righe permettono di caricare in alcuni registri gli indirizzi delle aree di memoria che il problema richiede di leggere o scrivere: infatti in R1 viene memorizzato l'indirizzo dove si trova il valore della base, in R2 quello dell'altezza e in R3 l'indirizzo della cella di memoria dove dovrà essere posto il risultato una volta eseguita la computazione.

Le righe 7-8 permettono di caricare i valori della base e dell'altezza rispettivamente nel registro R4 e nel registro R5, in modo che poi si possa eseguire la moltiplicazione su quei valori, che viene fatta nella riga successiva. Come si nota il risultato viene scritto in R6 e l'ultima riga provvede poi a copiarlo in memoria all'indirizzo contenuto in R3, cioè 0x300.

Recapitolando sono stati usati i seguenti registri con questo scopo:

- R1: contiene l'indirizzo della cella di memoria dove si trova il valore della base
- R2: contiene l'indirizzo della cella di memoria dove si trova il valore dell'altezza
- R3: contiene l'indirizzo della cella di memoria dove verrà copiato il risultato, cioè l'area
- R4: contiene il valore della base
- R5: contiene il valore dell'altezza
- R6: contiene il valore dell'area

Vale la pena notare due cose:

- si sarebbe potuta fare "economia" nell'utilizzo dei registri, ad esempio memorizzando l'area in R1, poichè dopo il suo utilizzo alla riga 7 il dato contenuto (0x100) poteva tranquillamente essere sovrascritto. Per questo corso si sconsiglia comunque di utilizzare lo stesso registro per scopi diversi in momenti diversi, in modo da privilegiare la chiarezza, indicando sempre lo scopo di ogni registro
- come già detto in precedenza, i registri possono contenere sia valori che indirizzi, poichè entrambi sono numeri, la differenza sarà soltanto nell'utilizzo che poi ne farà il programma.

4 Come recuperare e installare il simulatore

Per poter provare a correttezza di quanto scritto, è necessario avere sul proprio computer il simulatore scritto per questo corso. Il sito dove si possono trovare sia il software che le istruzioni per installarlo e utilizzarlo si trova all'indirizzo https://github.com/alessandro-bugatti/simple_assembler_simulator e chi fosse già in grado di utilizzare *git* e gli strumenti di compilazione del linguaggio C++ è invitato a seguire le istruzioni, in modo da avere la versione sempre aggiornata e seguirne un eventuale sviluppo futuro. Per tutti gli altri[1], verrà indicata una procedura molto più semplice, che dovrebbe essere alla portata di chiunque abbia un computer[2].

Andando in una sottosezione del sito precedente, in particolare quella indicata nel tab Release, il cui link diretto è https://github.com/alessandro-bugatti/simple_assembler_simulator/releases è possibile trovare un archivio zip che contiene, sia per Windows che per Linux, il programma vero e proprio, una serie di esempi e questo tutorial.

Dopo aver scaricato l'archivio opportuno, si proceda con l'estrazione ottenendo una cartella che contiene quanto detto sopra. Facendo doppio click sull'unico eseguibile presente (*simple_assembler_simulator.exe*, l'estensione nei sistemi Windows potrebbe anche non essere visibile), si aprirà una finestra a caratteri che costituisce il simulatore. Attenzione: alcuni sistemi Windows e/o alcuni antivirus potrebbero lamentarsi del fatto che stiate per mandare in esecuzione un programma scaricato da Internet, di fonte ignota. Se vi fidate di me ignorate gli avvisi, altrimenti leggetevi i sorgenti del programma e compilatevelo sul vostro sistema[3].

5 Come utilizzare il programma

Se dopo aver fatto doppio click si aprirà la finestra a caratteri, dovrebbe anche comparire una scritta come questa:

```
Assembly simulator
Version 1.1.0.0 Builds count:175
Copyright 2018 - Alessandro Bugatti
Load file:
```

con il cursore lampeggiante in attesa che voi inseriate il nome di un file contenente un programma scritto con il linguaggio assembly appena visto. Per verificare che tutto funzioni si può scrivere il nome *test.asm* (che è un file già compreso nell'archivio scaricato) e dovrebbe apparire questo menù:

```
1) Run
2) Show registers
3) Show memory
0) Exit
```

In questo modo il programma viene caricato nel simulatore ed è pronto per essere eseguito. Le voci di menù permettono di fare le seguenti cose:

1. **Run**: manda in esecuzione il programma, eseguendolo riga per riga. Il programma può essere uno degli esempi presenti nell'archivio compresso oppure essere scritto ex-novo. Se nel programma fossero presenti degli errori sintattici il programma mostrerà un messaggio in cui viene spiegato l'errore e, dopo 5 secondi, si chiuderà automaticamente. Se invece il programma viene eseguito fino alla fine, mostrando eventualmente dell'output a video, oppure niente, se non ci sono istruzioni di output: in ogni caso alla fine dell'esecuzione verrà mostrato un messaggio che indica che tutto è andato a buon fine e verrà mostrato nuovamente il menù.
2. **Show registers**: mostra il contenuto dei registri, sia in formato decimale che esadecimale. Viene richiesto un intervallo e vengono mostrati i valori all'interno di quell'intervallo di registri. Se ad esempio interessano i registri da R1 a R4, dopo aver inserito gli estremi dell'intervallo, apparirà un output simile al seguente:

```
Insert the range to view the values of registers (e.g. from 1 to 5 will
   show R1, R2, R3, R4, R5)
From: 1
To: 4
    Register |  Value(hex) |  Value(dec) |
-----------------------------------------
          R1 |       0x3e8 |        1000 |
```

[1] Temo la stragrande maggioranza...

[2] D'altronde state facendo Ingegneria, ci si aspetta che siate piuttosto tosti...

[3] Una soluzione ancora più drastica è quella di non scaricarlo affatto, ma la sconsiglio vivamente

```
R2 |       0x3e6 |         998 |
R3 |       0x76c |        1900 |
R4 |       0x1c2 |         450 |
```

1. **Show memory**: la stessa cosa dell'opzione precedente, solo per la memoria. In questo caso viene richiesto un intervallo di celle di memoria, che può essere espresso in forma decimale o esadecimale, il cui risultato sarà simile al seguente:

```
Insert the range to view the content of the memory, the values can be
   either hex or decimal number (e.g. from 256 (0x100) to 512 (0x200) will
    show the content of the memory from address 256 (0x100) to address 512
    (0x200) included)
From: 0x100
To: 0x104
   Addr(hex) |   Addr(dec) |  Value(hex) |  Value(dec) |
-------------------------------------------------------
       0x100 |         256 |        0x4f |          79 |
       0x101 |         257 |           0 |           0 |
       0x102 |         258 |        0x50 |          80 |
       0x103 |         259 |           0 |           0 |
       0x104 |         260 |        0x51 |          81 |
```

Come si può vedere ogni cella di memoria contiene al suo interno un singolo byte: poichè i registri sono a 16 bit nelle operazioni di LOAD e STORE verrà comunque spostato il contenuto di due celle di memoria adiacenti.

Lo scopo delle istruzioni che mostrano memoria e registri è quello di verificare, una volta fatto il *run* del programma, che i dati siano quelli che ci si aspetta, non avendo questo programma un'interfaccia grafica che possa essere aggiornata durante l'esecuzione del codice assembly.

In generale la sequenza di operazioni da svolgere quando si desidera provare a scrivere e testare un programma scritto in assembly sarà la seguente:

1. si crei un nuovo file con un editor di testo, il blocco note nella sua ultima versione può andare bene, oppure Scite, Geany o cose del genere
2. si scriva all'interno del file il codice assembly che risolve il problema richiesto (come ad esempio quello del calcolo dell'area del rettangolo)
3. si salvi il file all'interno della cartella ottenuta scompattando l'archivio che contiene l'assemblatore, dandogli l'estensione *.asm*, non è realmente necessario ma permette di vedere al volo che si tratta di un file scritto in assembly.
4. si apra il *simple_assembler_simulator* con il doppio click e alla richiesta *Load file*: si inserisca il nome del file appena salvato e si prema Invio
5. se viene mostrato il menù si mandi in esecuzione il programma premendo il tasto 1 e successivamente Invio
6. se il programma non contiene errori l'esecuzione terminerà correttamente, altrimenti il programma segnalerà un errore e la finestra verrà chiusa. In tal caso sarà necessario rileggere il codice assembly e cercare dove si trova l'errore, per correggerlo e ripetere la procedura dal punto 4.
7. per verificare che il programma, oltre a essere corretto sintatticamente, esegua anche il compito per cui era stato scritto, si visualizzino registri e memoria di interesse e si verifichi che contengano i valori corretti.

Riprendiamo l'esempio del paragrafo 3, con qualche modifica per rendere più semplice la visualizzazione dei risultati.

Problema: *si calcoli l'area di un rettangolo, considerando che il valore della base si trova all'indirizzo 0x100, quello dell'altezza all'indirizzo 0x102 e si vuole che il risultato venga scritto in memoria all'indirizzo 0x104.*

```
1 # Input: the width and the height of a rectangle
2 # Output: the area
3
4 # Preload
5 LOADiL R1 0x00
6 LOADiH R1 0x01
```

```
 7 LOADiL R2 0x07
 8 LOADiH R2 0x00
 9 STORE R1 R2
10 LOADiL R1 0x02
11 LOADiH R1 0x01
12 LOADiL R2 0x04
13 LOADiH R2 0x00
14 STORE R1 R2
15
16 # The solution
17 LOADiL R1 0x00
18 LOADiH R1 0x01
19 LOADiL R2 0x02
20 LOADiH R2 0x01
21 LOADiL R3 0x04
22 LOADiH R3 0x01
23 LOAD R4 R1
24 LOAD R5 R2
25 MUL R6 R4 R5
26 STORE R3 R6
27 END
```

Questa soluzione si trova già all'interno del file *area.asm* presente nella cartella scaricata e contiene anche una parte che **non** viene richiesta nei temi d'esame, cioè le righe dalla 5 alla 14, ma che è necessaria se si vuole simulare l'esecuzione del programma. Questa parte si limita a caricare il valore 7 nell'indirizzo di memoria 0x100 (righe 5-9) e il valore 4 nella cella di memoria 0x102 (righe 10-14), in modo da avere i valori di base e altezza e poter verificare di calcolare il risultato corretto. Il resto del codice, dalla riga 17 alla 27, è quello già visto in precedenza e si occupa di risolvere il problema vero e proprio. Dopo aver mandato in esecuzione questo programma con il comando Run, si può procedere a verificarne la correttezza visualizzando registri e memoria.

Per quanto riguarda i registri si può vedere che il contenuto da R1 a R6 è il seguente:

Register	Value(hex)	Value(dec)
R1	0x100	256
R2	0x102	258
R3	0x104	260
R4	0x7	7
R5	0x4	4
R6	0x1c	28

che è proprio quello che ci si attendeva, essendo il risultato, che si trova in R6, quello corretto.

Per la memoria, le tre locazioni che interessano sono la 0x100, 0x102 e 0x104 e il loro contenuto è il seguente:

Addr(hex)	Addr(dec)	Value(hex)	Value(dec)
0x100	256	0x7	7
0x101	257	0	0
0x102	258	0x4	4
0x103	259	0	0
0x104	260	0x1c	28
0x105	261	0	0

Va notato che viene mostrata anche la cella di indirizzo 0x105 perchè, come già detto, i dati vengono gestiti a blocchi di due byte.

Due ultime osservazioni rispetto a questo codice:

- le righe che iniziano con il simbolo **#** non vengono eseguite, quindi possono contenere qualsiasi cosa, in questo caso dei messaggi di commento. Come si vedrà nel seguito del corso questa è una pratica molto diffusa in tutti i linguaggi per computer e serve agli umani per avere informazioni immediatamente disponibili

- l'ultima riga contiene l'istruzione **END**, che, quando raggiunta, fa fermare il programma. In questo caso se ne sarebbe potuto fare a meno poichè il programma si sarebbe fermato comunque, essendo giunto alla fine, però in altri casi potrebbe essere più utile.

Si vedranno adesso le istruzioni di salto condizionato, che permettono al programma di non dovere per forza seguire un flusso lineare, come visto finora, ma spostarsi da un'istruzione a una qualsiasi altra, non necessariamente quella consecutiva. Questo permetterà di scrivere programmi nei quali si potranno implementare delle *selezioni* e delle *iterazioni*.

6 Istruzioni di branch

Le istruzioni di *branch*[4] sono tutte simili tra di loro, quindi ne verrnno analizzate solo un paio, indicando le differenze con le altre.

Fondamentalmente una istruzione di branch ha lo scopo di modificare il flusso lineare delle istruzioni che devono essere eseguite, per spostarsi verso un'istruzione in una posizione arbitraria, in modo da poter implementare un processo di *selezione* o un'*iterazione*, come si vedrà nei paragrafi successivi.

Le due parti di un'istruzione di branch sono:

- una **condizione di confronto** ($<, \leq, >, \geq, =, \neq$), applicata tra i valori contenuti in due registri, che, se risulta vera, fa saltare l'esecuzione a un'istruzione indicata da una *label*, altrimenti viene ignorata.
- una **label**, cioè un'etichetta linguistica (una parola) che indica la posizione dove saltare con il flusso di esecuzione. Quello che succede all'interno del calcolatore è che, se si effettua un branch, il *Program Counter*, al posto di essere incrementato di una posizione come succede normalmente, assume il valore dell'indirizzo a cui è posta la label. La label evita al programmatore di conoscere l'indirizzo di memoria, sarà l'assemblatore che si occuperà dei dettagli di basso livello.

6.1 La selezione

Una selezione viene utilizzata per decidere se effettuare o meno un'azione o una serie di azioni, al verificarsi di una certa condizione.

Problema: *si confronti il valore di due numeri naturali che si trovano rispettivamente nelle aree di memoria 0x100 e 0x200: se il primo è maggiore del secondo si scriva la loro differenza nell'area di memoria 0x300*

Come si può vedere in questo problema c'è un'azione (la differenza) che deve essere fatta solo se il primo operando è maggiore del secondo. Supponendo di avere già dei valori contenuti nelle posizione 0x100 e 0x200 una soluzione potrebbe essere la seguente:

```
1          LOADiH R3 0x01
2          LOADiL R3 0x00
3          LOAD R1 R3
4          LOADiH R3 0x02
5          LOADiL R3 0x00
6          LOAD R2 R3
7          BLE R1 R2 FINE
8          SUB R4 R1 R2
9          LOADiH R3 0x03
10         LOADiL R3 0x00
11         STORE R3 R4
12 FINE: END
```

Le righe dalla 1 alla 6 non fanno altro che caricare nel registro R1 il valore contenuto alla posizione di memoria di indirizzo 0x100 e nel registro R2 il valore alla posizione 0x200.

L'istruzione **BLE** (**B**ranch**L**ess than or **E**qual) confronta il valore del primo registro (R1) per vedere se è minore o uguale a quello del secondo (R2) e, se lo è, salta all'etichetta *FINE*, che si trova proprio alla fine del programma. Questo produce l'effetto che le righe dalla 8 alla 11 vengono saltate e quindi non viene effettuata la differenza. Se invece la condizione è falsa (cioè se R1 è maggiore di R2), il salto non viene eseguito e il flusso procede normalmente dalla riga 9 in poi, come se non ci fosse l'istruzione di branch.

Un'altra situazione tipica che si può verificare è quella che, se una condizione è vera, si esegue una determinata azione, altrimenti se ne esegue un'altra.

[4] Il nome branch, ramo in italiano, deriva dal fatto che queste istruzioni permettono di generare dei rami di esecuzione che vengono ercrosi al soddisfacimento o meno di certe istruzioni.

Problema: *si confronti il valore di due numeri naturali che si trovano rispettivamente nelle aree di memoria 0x100 e 0x200: se il primo è maggiore del secondo si scriva la differenza nell'area di memoria 0x300, altrimenti si scriva la differenza fra il secondo e il primo sempre nell'area di memoria 0x300.*

In questo caso si è aggiunta la differenza tra il secondo e il primo registro quando è il secondo registro a essere maggiore o uguale al primo e il programma risulta modificato nel seguente modo:

```
1          LOADiH R3 0x01
2          LOADiL R3 0x00
3          LOAD R1 R3
4          LOADiH R3 0x02
5          LOADiL R3 0x00
6          LOAD R2 R3
7          BGE R2 R1 ELSE
8          SUB R4 R1 R2
9          LOADiH R3 0x03
10         LOADiL R3 0x00
11         STORE R3 R4
12         B FINE
13 ELSE: SUB R4 R2 R1
14         LOADiH R3 0x03
15         LOADiL R3 0x00
16         STORE R3 R4
17 FINE: END
```

Stavolta ci sono due etichette, *ELSE* e *FINE* che servono per effettuare in maniera mutuamente esclusiva due diversi rami di esecuzione. Se la condizione richiesta dal branch alla riga 7, utilizzando stavolta **BGE**[5] (**B**ranch **G**reater than or **E**gual) è vera, vengono eseguite le righe dalla 13 alla 16, che fanno la differenza tra R2 e R1. Se invece la condizione è falsa vengono ancora eseguite le istruzioni dalla 8 alla 11, come nell'esercizio precedente, poi l'istruzione in riga 12, che è un salto incondizionato, fa saltare le istruzioni seguenti e porta direttamente alla fine del programma. Come si vede dunque, o verranno eseguite le istruzioni dalla 8 alla 11 oppure quelle dalla 13 alla 16.

Da questo esempio dovrebbe anche essere chiaro che possono esserci più etichette all'interno di un programma, ma non possono avere lo stesso nome, poichè altrimenti le istruzioni di branch non potrebbero indicare in maniera univoca dove dirottare il flusso di esecuzione.

Per ricordare le altre istruzioni di branch segue un breve elenco:

- **BLT**: **B**ranch **L**ess **T**han
- **BGT**: **B**ranch **G**reater **T**han
- **BLT**: **B**ranch **L**ess **T**han
- **BEQ**: **B**ranch **EQ**ual
- **BNE**: **B**ranch **N**ot **E**qual
- **B**: **B**ranch (senza condizioni)

Che istruzione di branch utilizzare e dove inserire l'etichetta dipende dalle richieste del problema, l'esperienza permette di fare delle scelte ragionevoli che portano al risultato richiesto.

6.2 L'iterazione

Spesso i programmi hanno bisogno di iterare, cioè ripetere, una stessa istruzione o serie di istruzioni per un certo numero di volte, ogni volta con qualche valore differente delle variabili utilizzate. Risolvere questo tipo di problemi richiede di modificare il flusso di un programma in modo che possa "ritornare indietro", cioè eseguire istruzioni già fatte in precedenza e continuare in questo modo fino a raggiungere il risultato desiderato. Anche in questo caso un esempio chiarirà il tipo di effetto che si vuole ottenere attraverso opportune istruzioni di branch.

Problema: *si verifichi se il numero naturale alla cella di memoria 0x100 è divisibile esattamente per il numero che si trova nella cella 0x200: se è divisibile scrivere il valore 1 nella cella di memoria 0x300, altrimenti il valore 0.*

[5]Si sarebbe potuto anche utilizzare BLE come nell'esempio precedente, è stata cambiata solo per evidenziare che, a patto di sapere cosa si sta facendo, possono essere usate istruzioni diverse per risolvere lo stesso problema.

Anche in questo caso si supponga che le celle in posizione 0x100 e 0x200 contengano già dei valori, in particolare 0x15 (21 decimale) e 0x7 (7 decimale) e il programma seguente debba qundi verificare se 0x15 è esattamente divisibile per 0x7. Una possibile soluzione prevede che venga sottratto il secondo valore (il 7 in questo esempio) dal primo valore (lo 0x15), mettendo il risultato al posto del primo valore. Se si procede in questo modo fino a quando il primo valore è maggiore del secondo, questa serie di iterazioni si arresterà quando il primo valore diventerà minore o uguale al secondo. A questo punto sarà possibile rispondere alla domanda iniziale semplicemente verificando se i due valori sono uguali, quindi il primo valore era esattamente divisibile per il secondo, in caso contrario non lo era.

```
1          LOADiH R3 0x01
2          LOADiL R3 0x00
3          LOAD R1 R3
4          LOADiH R3 0x02
5          LOADiL R3 0x00
6          LOAD R2 R3
7          B CONTROLLO
8  CICLO: SUB R1 R1 R2
9  CONTROLLO: BGT R1 R2 CICLO
10         BNE R1 R2 NO
11         LOADiH R4 0x00
12         LOADiL R4 0x01
13         B CARICA
14 NO: LOADiH R4 0x00
15         LOADiL R4 0x00
16 CARICA: LOADiH R3 0x03
17         LOADiL R3 0x00
18         STORE R3 R4
19         END
```

Le prime 6 righe caricano nei registri R1 e R2 i valori contenuti nelle celle 0x100 e 0x200, rispettivamente. Il salto incondizionato alla riga 7 porta subito il programma alla riga 9, dove viene verificata la condizione che il valore contenuto in R1 sia maggiore di quello contenuto in R2. L'istruzione di salto permette quindi di ritornare a un'istruzione precedente (la riga 8), nel caso la condizione sia vera. Con i valori di esempio quindi il valore 0x7 viene sottratto al valore 0x15 contenuto nel registro R1 e si prosegue verificando la condizione alla riga 9. Le righe 8-9 rappresentano così un costrutto iterativo, dove una stessa istruzione viene ripetuta un numero indeterminato di volte. Quando la condizione alla riga 9 diventerà falsa il flusso del programma continuerà alla riga 10, dove verrà effettuata una selezione sulla condizione R1$\neq$R2, inserendo il valore 1 o il valore 0 nella cella di memoria 0x300, in base alla verità o falsità di quella condizione, risolvendo così il problema richiesto.