1 Lezione del 01-10-24

1.1 Istruzioni per sottoprogrammi

Nei sottoprogrammi vengono coninvolte due istruzioni `CALL`, e `RET`. Entrambe si riferiscono alla pila.

1.1.1 CALL

- **Formato:** `CALL %EIP +/- $displacement`, `CALL *extended_register`, `CALL *memory`
- **Azione:** effettua la chiamata di un sottoprogramma, ovvero:
  - Salva il valore corrente di EIP nella pila;
  - Modifica EIP come farebbe JMP.
- **Flag:** nessuno.

Operandi	Esempi
Displacement	`CALL 0x00400010`
Registro	`CALL *%EAX`
Memoria	`CALL *0x00400010`

1.1.2 RET

- **Formato:** `RET`
- **Azione:** ritorna da un sottoprogramma, ovvero:
  - Rimuove un long dalla pila;
  - Lo inserisce in EIP.
- **Flag:** nessuno.

Esistono poi altre istruzioni di controllo, ovvero:

1.1.3 NOP

- **Formato:** `NOP`
- **Azione:** è l'istruzione nulla.
- **Flag:** nessuno.

1.1.4 HLT

- **Formato:** `HLT`
- **Azione:** arresta l'esecuzione fino al prossimo interrupt.
- **Flag:** nessuno.

1.1.5 HLF

- **Formato:** `HLF`
- **Azione:** arresta l'esecuzione e causa l'autocombustione del processore.
- **Flag:** nessuno.

1.2 Istruzioni privilegiate

Il codice in assembler può girare secondo due modalità sul sistema:

- **Sistema:** con accesso totale a tutte le istruzioni;
- **Utente:** senza l'accesso ad alcune istruzioni dette privilegiate.

Tra le istruzioni privilegiate ci sono `HLT`, `IN` e `OUT`. La `HLT` non è un grande problema, ma lo sono `IN` e `OUT`. Per ottenere input e output dal sistema, adoperiamo quindi determinati sottoprogrammi di servizio atti a fornire esattamente queste informazioni.

L'uso di sottoprogrammi di servizio per l'input/output è dovuto al fatto che le interfacce sono sistemi complessi, facili da portare in stato inconsistente, mentre i sottoprogrammi si assicurano di farne un corretto uso.

1.3 Struttura di un programma assembler

Vediamo adesso come strutturare un programma assembler scritto nell'ambiente GAS (Gnu Assembler). Un programma assembler è diviso in due sezioni

- **Sezione dati:** qui si dichiarano le variabili, ergo nomi simbolici per indirizzi di memoria che contengono i dati del programma;
- **Sezione codice:** istruzioni.

  In un programma abbiamo bisogno di:

- **Istruzioni**, viste finora;
- **Direttive**, necessarie all'assemblaggio e alla dichiarazione di variabili.

Ad esempio, potremo avere:

```
1 .GLOBAL _main
2
3 .DATA
4 ...
5
6 .TEXT
7 _main:  NOP
8 ...
9         RET
```

Le linee che iniziano col punto sono direttive, le altre istruzioni. Una riga qualsiasi del codice è fatta come:

```
1 nome: OPCODE operandi # commento [\CR]
```

dove abbiamo una label, l'istruzione e un commento.

Tutto qui può mancare, tranne il ritorno carrello. Tutte le righe, inclusa l'ultima, vanno terminate. Inoltre, l'ultima riga dovrebbe essere una RET, che restituisce l'esecuzione al chiamante (qui l'ambiente).

Conviene iniziare il programma con una NOP, per assicurarsi che in fase di inizializzazione esso non faccia effettivamente nulla.

Vediamo ad esempio il programma visto prima per il conteggio degli uni, reso in questa struttura:

```
.GLOBAL _main
.DATA
dato:       .LONG 0x0F0F0101
conteggio:  .BYTE 0x00

.TEXT
_main:      NOP
            MOVB $0x00, %CL
            MOVL dato, %EAX
comp:       CMPL $0x00, %EAX
            JE fine
            SHRL %EAX
            ADCB $0x00, %CL
            JMP comp
fine:       MOVB %CL, conteggio
            RET
```

1.3.1 Direttive

Tutte le direttive iniziano con il carattere punto. Esse sono:

- **Dichiarazione di variabili:** Variabili dichiarate di seguito sono sempre consecutive in memoria. Si ha, di base:
  - `.BYTE`: riserva 1 byte;
  - `.WORD`: riserva 2 byte;
  - `.LONG`: riserva 4 byte.

  Esempi

  ```
  var0: .WORD                 # scalare, 2 byte, valore 0x0000
                              # (considerato brutto, non inizializzare
                              #  si fa con .FILL)
  var1: .BYTE 0x30            # scalare, 1 byte, valore 0x30
  var2: .BYTE 0x30,0x31       # vettore, 2 componenti da 1 byte,
                              # valore 0x30 e 0x31
  var3: .WORD 0x1020, 0x32AB  # vettore, 2 componenti da 2 byte,
                              # valore 0x1020e 0x32AB
  var4: .LONG var3+2          # scalare, 4 byte, valore 0xAB
  ```

  Esistono altri modi di inizializzare variabili particolari:

  - `.FILL numero, dim, espressione`: dichiara `numero` variabili di lunghezza `dim` e le inizializza ad espression (0 di default). Dim può essere 1, 2 o 4.
  - **ASCII**: si può usare la codifica ASCII fra single tick ', coi caratteri speciali dopo sequenze di escape, per indicare singoli byte. Ad esempio:

```
var5: .BYTE 'S', 'o', 'n', 'n', 'o'        # vettore, 4 componenti
                                           # da 1 byte
var6: .BYTE 0x53, 0x6F, 0x6E, 0x6E, 0x6F   # vettore, 4 componenti
                                           # da 1 byte
var7: .ASCII "Stea"                        # vettore, 4 componenti
                                           # da 1 byte
var8: .ASCIZ "Stea"                        # vettore, 5 componenti
                                           # da 1 byte (include il
                                           # terminatore)

```

- **Altre direttive:**
  - `.INCLUDE "path"`: include un sorgente nel presente file, prima dell'assemblamento;
  - `.SET nome, espressione`: serve a creare **costanti simboliche**. Tali costanti hanno nome `nome` e valore `espressione`. Ad esempio:

```
.SET dimensione, 4
.SET n_iter, (100 * dimensione)
...
MOV $n_iter, %CX  # e' accesso immediato
```

1.4 Costanti numeriche

Possiamo indicare costanti numeriche attraverso le seguenti convenzioni:

- **Naturali:** non hanno segno, e vengono convertite nella loro rappresentazione in base 2;
- **Intere:** hanno un segno + o - davanti, e vengono convertite nella loro rappresentazione in complemento a 2.

Inoltre possiamo scrivere costanti in base 2, 8, 10 e 16 attraverso i prefissi `0b`, `0`, nessun prefisso e `0x`.

Le variabili, quando non sono della dimensione giusta, vengono solitamente troncate (con avviso dall'assemblatore) o estese (senza avvisi dall'assemblatore).

1.5 Controllo di flusso

I costrutti di flusso a cui siamo abituati vengono implementati attraverso istruzioni di salto. Conviene comunque ragionare in costrutti ad alto livello, e limitarsi a tradurli in assembler. Da qui in puoi useremo una sintassi pseudo-C per indicare questi costrutti ad alto livello.

1.5.1 If-then-else

Prendiamo la sintassi:

```
if(%AX < variabile) {
  //ramo if
  ...
} else {
  //ramo else
  ...
}
```

```
//prosegui
...
```

potremo tradurla in due modi:

- Invertendo i rami then e else:

```
          CMP variabile, %AX
          JB ramothen
ramoelse: ... # ramo else
          JMP segue
ramothen: ... # ramo then
segue:    # prosegui
          ...
```

- Invertendo la condizione:

```
          CMP variabile, %AX
          JAE ramoelse
ramothen: ... # ramo then
          JMP segue
ramoelse: ... # ramo else
segue:    ... # prosegui
```

1.5.2 Ciclo for

Prendiamo:

```
for(int i = 0; i < variabile; i++) {
  //iter
  ...
}
//prosegui
...
```

si rende attraverso il registro CX, come:

```
        MOV $0, %CX
ciclo:  CMP var, %CX
        JE segue
        ... # iter
        INC %CX
        JMP ciclo
segue:  ... # prosegui
```

1.5.3 Ciclo do-while

Prendiamo infine:

```
do {
  //iter
  ...
} while(AX < var)
//prosegui
...
```

si rende come:

```
ciclo:  ... # iter
        CMP var, %AX
        JB ciclo
        ... # prosegui
```

1.5.4 Un piatto di spaghetti

In assembler ci è concesso fare ciò che non è permesso da linguaggi strutturati come il C o il Pascal. In questi linguaggi, un costrutto ha un solo punto di ingresso e un solo punto di uscita.

In assembler, invece, possiamo saltare fuori e dentro cicli e costrutti quando e dove vogliamo, ed è il programmatore che deve pensare a cosa il programma sta effettivamente facendo. Ad esempio, nessuno ci vieta di dire:

```
ciclo:  ... # inizio ciclo
        ...
label1: ... # meta' ciclo
        CMP var, %AX
        JB ciclo
        ...
        JMP label1 # salto dentro un ciclo a meta' esecuzione?
```

In assembler abbiamo a disposizione un'istruzione dedicata per i loop, che è:

1.5.5 LOOP

- **Formato:** `LOOP destination`
- **Azione:** decrementa ECX e salta alla destinazione se ECX $\neq 0$. ECX va inizializzato al numero di iterazioni desiderate, e non va toccato durante il ciclo.
- **Flag:** nessuno.

Si nota che la LOOP decrementa sempre ECX, quindi si applica difficilmente a cicli FOR dove vogliamo che la variabile di controllo incrementi, e ci serve che il suo valore nel corpo del ciclo. Si noti la differenza nei due esempi:

```
for(int i = var; i > 0; i--) {
  //iter (usa i)
}
```

diventa:

```
        MOV var, %ECX
ciclo:  ... # iter
        LOOP ciclo
```

```
for(int i = 0; i < var; i++) {
  //iter (usa i)
}
```

diventa:

```
        MOV $0, %EBX # usa EBX
ciclo:  ... # iter
        INC EBX
        CMP var, %EBX
        JE ciclo
```

1.5.6 LOOP condizionali

Esistono versioni condizionali della LOOP, che sono `LOOPE` e `LOOPNE`, simili alle Jump condizionali. In questo caso, oltre al registro ECX, si verifica la condizione e nel caso si salta. Ad esempio:

```
        MOV $10, %ECX
ciclo:  CMP src, dest
        LOOPcond ciclo
```

Queste istruzioni non sono indispensabili, in quanto possono essere rimpiazzate facilmente dalla `CMP` unita ad un Jump condizionale.

1.6 Passaggio di argomenti a sottoprogrammi

Le CALL e RET prima definite non fornisicono modi per passare parametri ai sottoprogrammi, o restituire valori ai chiamanti.

Dobbiamo quindi stabilire delle convenzioni, scegliendo se:

- Usare locazioni di memoria condivise;
- Usare registri;
- Usare la pila (che non verrà visto nel corso).

In assembler non esiste il concetto di visibilità o variabili locali, tutta la memoria è indirizzabile a qualsiasi livello. Comunque, quando si scrive un sottoprogramma, bisogna specificare i parametri di ingresso e di uscita con un'opportuno commento, come:

```
# sottoprogramma "sottoprog", [descrizione]
# ingresso: %AX, [descrizione]
#           %EBX, [descrizione]
# uscita:   CAX, [descrizione]

sottoprog:  ...
            MOV ..., %CX # preparo il ritorno
            RET
```

adesso potremo usare il sottoprogramma come:

```
MOV ..., %AX # preparo i parametri
MOV ..., %EBX
CALL sottoprog # chiamo
MOV %CX, var # var contiene il ritorno
```