Corso di Architettura degli Elaboratori
Programmazione Assembly: Progetto
21 Dicembre 2006

Lo studente implementi un **traduttore**, che chiameremo `ijvm2as`, scritto nel linguaggio Assembly 8088, che trasformi un generico programma IJVM16 (chiamato programma sorgente) in un equivalente programma nel linguaggio Assembly 8088 (chiamato programma oggetto).

1 Il Linguaggio IJVM16

Il linguaggio IJVM16 è una semplice variante del linguaggio IJVM, che abbiamo studiato nell'ultima parte del corso. Le principali differenze tra IJVM16 e IJVM sono le seguenti:

- La memoria di IJVM16 è composta da 2^{16} byte oppure da da 2^{15} parole da 16 bit. Per indirizzare un byte occorrono quindi 16 bit, mentre per indirizzare una parola bastano 15 bit.
- Anche in IJVM16 come in IJVM esistono i registri `LV`, `SV` e `PC`, che però sono registri a 16 bit anziché a 32 bit. Non esiste, invece, la porzione costante di memoria e, di conseguenza, non esiste il registro `CPP`.
- Le istruzioni disponibili in IJVM16 sono le stesse disponibili in IJVM, con alcune, semplici variazioni:
  - Per semplicità, non sono disponibili le istruzioni costruite tramite il prefisso `WIDE`.
  - L'istruzione `LDC_W` non fa parte dell'insieme di IJVM16, non esistendo la porzione costante nel relativo modello di memoria.
  - Esiste una nuova istruzione `EXIT` che ha l'effetto di terminare l'esecuzione del programma.
  - Esistono due nuove istruzioni per gestire l'I/O. L'istruzione `IN` legge una costante intera dallo standard input e la inserisce in cima allo stack degli operandi, l'istruzione `OUT` stampa sullo standard output il contenuto dell'operando in cima allo stack.

`ijvm2as` riceve in input un programma IJVM16 dallo standard input e produce un programma 8088 (equivalente al programma in input) nel file `output.s`. Sia il programma in input che il programma in output non sono scritti in forma esplicita (non sono, quindi, sequenze binarie), ma sono sequenze di caratteri alfanumerici, spazi e punteggiatura. Si supponga in tal senso di utilizzare la codifica ASCII. A tal fine, occorre stabilire alcune convenzioni per la scrittura del codice IJVM16:

- Ogni istruzione occupa una riga.
- Ogni istruzione può essere preceduta da un'etichetta, ovvero da una sequenza alfabetica lunga al più 8 caratteri, seguita dal carattere ":".
- Le costanti necessarie nelle istruzioni `BIPUSH` e `IINC` sono espresse come numeri in notazione decimale di valore compreso tra 0 e 255.
- Gli spiazzamenti necessari nelle istruzioni `IINC`, `ILOAD`, `ISTORE` e `LDC` sono numeri in notazione decimale di valore compreso tra 0 e 255.
- Gli spiazzamenti necessari nei salti condizionati e incondizionati sono invece espressi attraverso etichette.
- L'istruzione `INVOKEVIRTUAL`, che in IJVM prende come argomento uno spiazzamento relativo a `CPP`, ha tre argomenti in IJVM16:
  - Due costanti, entrambe numeri in notazione decimale di valore compreso tra 0 e 255 che specificano il numero di argomenti e il numero di variabili locali della procedura chiamata.
  - Un'etichetta, che specifica la prima istruzione del corpo della procedura.

Un esempio di programma IJVM16 potrebbe quindi essere il seguente:

```
          BIPUSH 5
          INVOKEVIRTUAL 1 2 PRODOTTO
          EXIT
PRODOTTO: BIPUSH 0
```

```
          ISTORE 1
          BIPUSH 0
          ISTORE 2
CICLO:    ILOAD 1
          ILOAD 0
          IF_CMPEQ FINE
          ILOAD 1
          BIPUSH 1
          IADD
          ISTORE 1
          ILOAD 0
          ILOAD 2
          IADD
          ISTORE 2
          GOTO CICLO
FINE:     ILOAD 2
          IRETURN
```

2 Suggerimenti

Il programma `ijvm2as` può essere implementato come un traduttore ad una sola passata. Per ogni riga del programma sorgente IJVM16, il traduttore deve:

1. Decodificare l'istruzione corrente
2. Tradurre in maniera opportuna l'istruzione IJVM16 in una sequenza di istruzioni assembly
3. Scrivere la sequenza di istruzioni assembly su un file di output.

2.1 Operazioni su stringhe

Nel caso generale, una riga del file sorgente si presenta nel modo seguente:

`[etichetta:]  opcode [operando/i]`

dove le parentesi quadre denotano elementi opzionali. Occorre determinare un metodo corretto ed efficiente per effettuare il parsing di ogni riga, in modo da eliminare gli spazi vuoti e da estrarre i token (etichetta, nome istruzione, operandi) che formano ciascuna riga del file sorgente. Qualche suggerimento utile:

- La chiamata di sistema `_GETCHAR` consente di leggere un carattere dallo standard input, lo salva in AL e azzera AH. Ad esempio, il frammento seguente legge caratteri dallo standard input finchè non incontra un carattere di terminazione di linea, ed evita di processare i caratteri vuoti.

```
ciclo:    PUSH _GETCHAR  ;Legge un carattere dallo standard input
          SYS
          CMPB AL,10     ;Confronta il carattere letto con "\n"
          JE fine        ;Se il carattere letto è "\n", esce dal ciclo
          CMPB AL,32     ;Controlla se il carattere letto è uno spazio (" ")
          JNE faiQualcosa ;Se il carattere letto non è uno spazio, lo processa
          JMP ciclo
          ....
          ....
faiQualcosa: ...
```

- Le istruzioni assembly `MOVS(B)`, `LODS(B)`, `STOS(B)`, `SCAS(B)`, `CMPS(B)` combinate con l'istruzione `REPxxx` consentono di effettuare operazioni di confronto, copia e scansione di stringhe. In particolare, l'istruzione:

  `REPxxx string_instr`, con **string_instr**={`MOVS|LODS|STOS|SCAS|CMPS`}

ripete l'esecuzione dell'istruzione **string_istr** per un numero di volte pari al contenuto del registro CX. Operandi impliciti:

- CX deve contenere la lunghezza della stringa.
- L'indirizzo della stringa sorgente deve essere contenuto in SI, l'indirizzo della stringa destinazione in DI.

Le istruzioni `REPE` e `REPNE` sono due varianti che possono essere utilizzate abbinate alle istruzioni di confronto e ricerca. In particolare:

- L'istruzione `REPE` ripete il ciclo finchè il registro CX ha un valore diverso da 0 e le parole confrontate sono uguali.
- L'istruzione `REPNE` ripete il ciclo finchè il registro CX ha valore diverso da 0 e le parole confrontate sono diverse.

Esempio di frammento assembly che confronta una stringa sorgente con la keyword `"EXIT"`. (L' indirizzo della stringa sorgente è contenuto in AX, il numero di caratteri è in CX)

```
 MOV SI, AX   ; Metto in SI  l'indirizzo della stringa sorgente
 MOV DI, exit ; Metto in DI  l'indirizzo della stringa destinazione
 REPE CMPSB
 JE equal   ; Se le stringhe sono uguali, salto all'etichetta "equal"
 .....      ; Se mi trovo qui, le stringhe confrontate sono diverse
 .....
equal:
    ....
.SECT .DATA
exit: .ASCII "EXIT"
```

La sezione C.4.3 descrive le principali istruzioni su stringhe; la sezione C.8.5 contiene un esempio (assembly) di manipolazione di stringhe.

2.2 Processo di Traduzione

Solo alcune osservazioni:

- Occorre prima di tutto capire in che modo il modello di memoria di IJVM16 possa essere simulato nel modello di memoria del processore 8088. In particolare, occorre capire quale parte della memoria del processore 8088 simulino lo stack. La risposta è però ovvia, visto che anche nell'8088 esiste uno stack. Occorre poi capire quali registri dell'8088 simulino i registri `SP`, `LV` e `PC` di IJVM16. Ma anche in questo caso, la risposta è molto semplice: `SP`, `BP` e `PC`, rispettivamente.
- Ogni istruzione IJVM16 del programma sorgente viene tradotta in una sequenza di istruzioni 8088 nel programma oggetto. Per esempio, l'istruzione `IADD` potrebbe essere tradotta nella sequenza di istruzioni seguente:

  ```
  pop AX
  pop BX
  add AX,BX
  push AX
  ```

- Sebbene l' istruzione `INVOKEVIRTUAL` del linguaggio IJVM16 produca un effetto simile a quello prodotto dall'istruzione `CALL` in assembly (entrambe sono usate per l'invocazione di procedure), le implementazioni dei due costrutti differiscono notevolmente per quanto riguarda i meccanismi di passaggio di parametri, di accesso ai dati locali e di organizzazione dello stack. La traduzione `INVOKEVIRTUAL` $\rightsquigarrow$ `CALL` non è la scelta più indicata. E' invece più opportuno pensare ad una traduzione di `INVOKEVIRTUAL` in una sequenza di istruzioni che includa solo operazioni logico-aritmetiche, trasferimenti e salti.

2.3 Operazioni su file

L'interprete as88 dispone di sei chiamate di sistema per gestire le operazioni di I/O su file; la tabella C.7 del libro di testo contiene la descrizioni delle chiamate di sistema disponibili, mentre la sezione C.8.7 contiene un esempio (in Assembly) di operazioni di lettura/scrittura su file.
Le routine di sistema che possono essere utili per gli scopi del progetto sono:

- **Creazione di file**: `_CREAT(*name,*mode)`
  La chiamata `CREAT` (codice operativo: 8) crea un file vuoto.
  Argomenti. *name*: indirizzo della stringa contenente il nome del file, *mode*: indirizzo della stringa contenente la stringa di permessi (UNIX) associati al file.
  Valore di ritorno. Restituisce (nel registro AX) il **descrittore** del file creato.
- **Apertura di file**: `_OPEN(*name,mode)`
  La chiamata `OPEN` (codice operativo: 5) apre un file esistente in lettura/scrittura.
  Argomenti. *name*: indirizzo della stringa contenente il nome del file, *mode*: $\in \{0, 1, 2\}$: 0= apre il file in lettura, 1= apre il file in scrittura, 2= apre il file in lettura/scrittura.
  Valore di ritorno. Restituisce (nel registro AX) il descrittore del file.
- **Scrittura su file**: `_WRITE(fd,buf,nbytes)`
  La chiamata `WRITE` (codice operativo: 4) scrive sul file attualmente in uso il contenuto di un buffer passato come argomento.
  Argomenti. *fd*: descrittore del file, *buf*: indirizzo di memoria del buffer da scrivere, *nbytes*: numero di byte da scrivere.
  Valore di ritorno. Restituisce (nel registro AX) il numero di byte correttamente scritti.
- **Chiusura di file**: `_CLOSE(fd)`
  La chiamata `CLOSE` (codice operativo: 6) chiude un file precedentemente aperto.
  Argomenti. *fd*: descrittore del file.
  Valore di ritorno. 0 se l'operazione si è conclusa correttamente.

3 Modalità di Consegna e di Valutazione

La valutazione del progetto dipende strettamente dai seguenti vincoli.

- Il progetto deve essere consegnato entro le 24.00 di Sabato, 3 Febbraio 2007.
- Le modalità di consegna saranno rese note sulle pagine web del corso a partire dalla metà di gennaio.
- **Presenza di commenti**. Il codice assembly deve essere commentato opportunamente in modo da agevolarne la comprensione. **I progetti privi o carenti di commenti non saranno valutati**.
- **Lavoro di gruppo**. Il progetto deve essere svolto in modo collaborativo da tutti i componenti del gruppo. Ogni componente del gruppo deve conoscere le scelte progettuali ed i dettagli implementativi di ogni singolo aspetto del progetto.
- **Funzionamento**. Il progetto consegnato deve essere funzionante, almeno per programmi che contengano un sottoinsieme non banale dell'insieme delle istruzioni di IJVM16. Ovviamente, il traduttore `ijvm2as` deve poter essere assemblato senza errori, in caso contrario il progetto non sarà valutato.