La macchina IJVM

Daniele Paolo Scarpazza
Dipartimento di Elettronica e Informazione
Politecnico di Milano

1 Giugno 2004

Materiale di riferimento

- Testo: Tanenbaum, *Structured Computer Organization*
- Software disponibile a: www.ontko.com/mic1/
  - Comprende i seguenti materiali:
    - Assemblatore e micro-assemblatore (scaricare e installare)
    - mic-1 User Guide (studiare)
    - IJVM Assembly Language Specification (studiare)
  - E' richiesto un ambiente Java funzionante, per esempio il JRE, disponibile all'indirizzo: java.sun.com/j2se/1.4.2/download.html

Assembly IJVM

- Costanti:
  - si dichiarano globalmente
  - sintassi:
    ```
    .constant
    constant1 value1
    constant2 value2
    .end-constant
    ```
  - si referenziano per nome con `LDC_W`
- Variabili:
  - solamente locali
  - si possono dichiarare solo in `.main` o in ciascun `.method`
  - sintassi:
    ```
    .var
    var1
    var2
    .end-var
    ```
  - si referenziano per nome con `ILOAD`, `ISTORE`
- Etichette:
  - marcano punti del programma a cui si può saltare;
  - sono accessibili solo all'interno del metodo in cui appaiono
  - esempio:
    ```
                IFLT lt_max
                GOTO gte_max
    lt_max:     HALT
    gte_max:
                ERR
    ```
- Main
  - esattamente uno per programma
  - sintassi
    ```
    .main
    -- variable declaration --
    -- program contents --
    .end-main
    ```

Assembly IJVM

- Metodi:
  - sintassi:

```
.method method_name(par1,par2,...)
-- variable declaration --
-- method contents --
.end-method
```

  - il nome del metodo dichiarato si aggiunge alla tabella globale delle costanti
  - e quindi può essere referenziato per nome con `INVOKEVIRTUAL`
  - l'invocazione di un metodo richiede le seguenti azioni:

```
PUSH objref                (ignorato, ma necessario per compatibilità)
PUSH par1
PUSH par2
...
INVOKEVIRTUAL method_name
```

  - i parametri sono automaticamente aggiunti alle variabili locali del metodo e possono essere referenziate con `ILOAD par1`
  - i metodi devono essere dichiarati dopo il main;

Assemblaggio e simulazione

- Scompattare `mic1win.exe` in una directory a piacere, per esempio `C:\MyProjects\IJVM`
- Modificare `env.bat`, in accordo con i vostri percorsi:

```
path "C:\Program Files\Java\j2re1.4.2_04\bin";%path%
set CLASSPATH=C:\MyProjects\IJVM\classes.zip
```

- Trascrivere il programma di esempio in `prova.jas`
- Con l'assemblatore (lanciare con `ijvmasm.bat`)
  - specificare file di ingresso e uscita `prova.jas` e `prova.ijvm` rispettivamente;
  - cliccare [Compile];
- Con il simulatore (lanciare con `mic1sim.bat`)
  - scegliere File > load microprogram > `mic1ijvm.mic1`
  - scegliere File > load macroprogram > `prova.ijvm`
  - cliccare [Run];

```
// programma d'esempio;
// mostra car. ASCII 32-126
.constant
        one 1
        start 32
        stop 126
.end-constant
.main
        LDC_W start
next:   DUP
        OUT
        DUP
        LDC_W stop
        ISUB
        IFEQ done
        LDC_W one
        IADD
        GOTO next
done:   POP
        HALT
.end-main
```

Sommario del set di istruzioni IJVM

Mnemonic	Operands	Description
BIPUSH	byte	Push a byte onto stack
DUP	N/A	Copy top word on stack and push onto stack
ERR	N/A	Print an error message and halt the simulator
GOTO	label name	Unconditional jump
HALT	N/A	Halt the simulator
IADD	N/A	Pop two words from stack; push their sum
IAND	N/A	Pop two words from stack; push Boolean AND
IFEQ	label name	Pop word from stack and branch if it is zero
IFLT	label name	Pop word from stack and branch if it is less than zero
IF_ICMPEQ	label name	Pop two words from stack and branch if they are equal
IINC	var name, byte	Add a constant value to a local variable
ILOAD	var name	Push local variable onto stack
IN	N/A	Reads a character from the keyboard buffer and pushes it; If no character is available, 0 is pushed
INVOKEVIRTUAL	method name	Invoke a method
IOR	N/A	Pop two words from stack; push Boolean OR
IRETURN	N/A	Return from method with integer value
ISTORE	var	Pop word from stack and store in local variable
ISUB	N/A	Pop two words from stack; push their difference
LDC_W	constant name	Push constant from constant pool onto stack
NOP	N/A	Do nothing
OUT	N/A	Pop word off stack and print it to standard out
POP	N/A	Delete word from top of stack
SWAP	N/A	Swap the two top words on the stack
WIDE	N/A	Prefix instruction; next instruction has a 16-bit index

Ripasso: numerazione esadecimale

- Il sistema di numerazione esadecimale:
  - è posizionale a base sedici;
  - ogni cifra può assumere i valori 0,1,2, ... ,9,A,B,C,D,E,F
  - si denota di solito con il prefisso “0x”
- Conversione da base 16 a base 10 e viceversa:
  - 16>10: $a_n a_{n-1} \ldots a_1 a_0 = a_0 + a_1 16^1 + a_2 16^2 + \ldots + a_n 16^n$
  - 10>16: lunga divisione, collezione resti in ordine inverso
- Conversione da base 16 a base 2 e viceversa:
  - 16>2: espansione di ogni cifra in esattamente 4 cifre
    - 0>0000, 1>0001, ..., A>1010, B> 1011, ..., F>1111
  - 2>16: contrazione di ogni gruppo di 4 cifre in una (dx>sx)
    - 0000>0, 0001>1, ..., 1010>A, 1011>B, ..., 1111>F
  - Nota: 1 cifra hex=4 bit, 2 cifre=1 byte, 8 cifre=1 word(32)

Esempio: istruzione IADD

- Stato originario dello stack:
  [0x00000001] 108 ← top
  [0x00000002] 104
  [...] 100

- Prelevamento degli operandi dallo stack:
  [...] 100 ← top

- Impilamento del risultato:
  [0x00000003] 104 ← top
  [...] 100

Esempio: istruzione ISUB

- Per l'istruzione `ISUB`, l'elemento in cima alla pila è il secondo operando (sottraendo), e quello che si trova sotto di esso è il primo operando (minuendo); cioè:
  - Se lo stack si presenta così:

    [y] ← top
    [x]
    [...]
  - L'istruzione `ISUB` lascia lo stack come segue:

    [x-y] ← top
    [...]

- Ad esempio le istruzioni:

```
BIPUSH 4
BIPUSH 3
ISUB
```

  lasciano in cima allo stack il valore 1 (e non il valore -1)

Esercizio 1

Scrivere in assembly IJVM un programma che si comporti come il seguente programma C:

```
char c;
while (1)  {
    printf("?");
    c = getchar();
    if (c!='Q')
        printf("%c\n",c);
    else
        break;
}
```

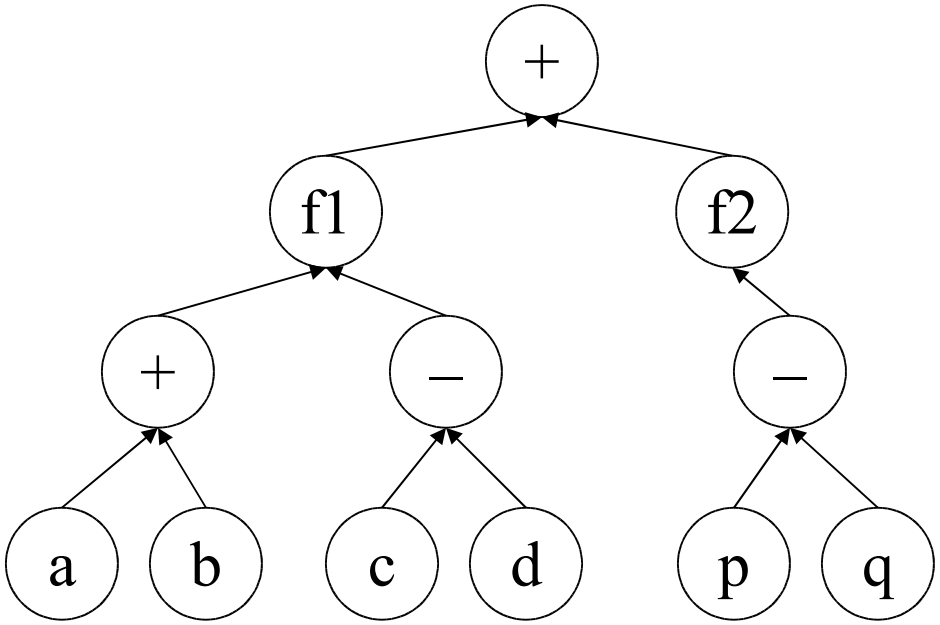
(il programma mostra all'utente un '?', legge un carattere e lo mostra a video finché l'utente inserisce 'Q')

Esercizio 1: soluzione

```
.constant
        prompt  63      // carattere '?'
        qchar   81      // carattere 'Q'
        cr      13      // carattere ritorno carrello
        lf      10      // carattere avanzamento linea
.end-constant

.main
loop:   LDC_W     prompt
        OUT             // mostro il prompt
        BIPUSH    0     // resetto il carattere letto
read:   POP
        IN              // leggo lo stato della tastiera
        DUP
        IFEQ      read  // nessun carattere: torno a leggere
        DUP
        LDC_W     qchar
        IF_ICMPEQ end   // il carattere letto è 'Q', esco
        OUT             // stampo a video il carattere, top
        LDC_W     cr
        OUT             // stampo il ritorno carrello
        LDC_W     lf
        OUT             // stampo l'avanzamento riga
        GOTO      loop  // ripeto il ciclo
end:    HALT
.end-main
```

Macchine a pila e notazione RPN

- Data una espressione in notazione infissa, la maniera più semplice per valutarla su una macchina a pila è convertirla in forma postfissa (notazione polacca inversa, reverse polish notation, RPN);
- Esempio: f1(a + b, c - d) + f2 (p - q)
- Costruisco l'albero sintattico:



- Visita dell'albero postorder: a b + c d - f1 p q - f2 +

- Traduzione:

```
LDC_W       objref
ILOAD       a
ILOAD       b
IADD
ILOAD       c
ILOAD       d
ISUB
INVOKEVIRTUAL f1
LDC_W       objref
ILOAD       p
ILOAD       q
ISUB
INVOKEVIRTUAL f2
IADD
```

Esercizio 2

- Ristrutturare il programma precedente in modo che:
  - la lettura dei caratteri avvenga in un metodo chiamato `getchar`;
  - `getchar` non ritorni fino a quando non ha ricevuto un carattere;

Esercizio 2: soluzione

```
.constant
objref   0
prompt  63
qchar   81
cr      13
lf      10
.end-constant

.main
loop:    LDC_W              prompt
         OUT
         LDC_W              objref
         INVOKEVIRTUAL      getchar
         DUP
         LDC_W              qchar
         IF_ICMPEQ          end
         OUT
         LDC_W              cr
         OUT
         LDC_W              lf
         OUT
         GOTO               loop
end:     HALT
.end-main
```

```
.method getchar ()
         BIPUSH     0
read:    POP
         IN
         DUP
         IFEQ       read
         IRETURN
.end-method
```

Esercizio 3

- Scrivere un programma equivalente a:

```
int a, b;
while (1)  {
      printf("A>");
      scanf("%i", &a);
      if (a==0) break;
      printf("B>");
      scanf("%i", &b);
      printf(">> %i", a+b);
}
```

- Strutturare tale programma nei metodi `getint`, `putint` e `main`;
- Avvalersi del codice di `getchar` già sviluppato nell'esercizio precedente;

Esercizio 3: soluzione (1/3)

```
.constant
          objref     0
          prompt    62
          achar     65
          bchar     66
          cr        13
          lf        10
          c1         1
          c10       10
          c100     100
          c1000   1000
.end-constant

.main
.var
          a
          b
.end-var

loop:     LDC_W               achar
          OUT
          LDC_W               prompt
          OUT
          LDC_W               objref
          INVOKEVIRTUAL       getint
          DUP
          ISTORE              a
          IFEQ                end

          LDC_W               bchar
          OUT
          LDC_W               prompt
          OUT
          LDC_W               objref
          INVOKEVIRTUAL       getint
          ISTORE              b

          LDC_W               prompt
          OUT
          LDC_W               prompt
          OUT

          ILOAD               a
          ILOAD               b
          IADD
          LDC_W               objref
          SWAP
          INVOKEVIRTUAL       putint

          GOTO                loop
end:      HALT
.end-main

.method getche ()
          BIPUSH              0
read:     POP
          IN
          DUP
          IFEQ                read
          DUP
          OUT
          IRETURN
.end-method
```

Esercizio 3: soluzione (2/3)

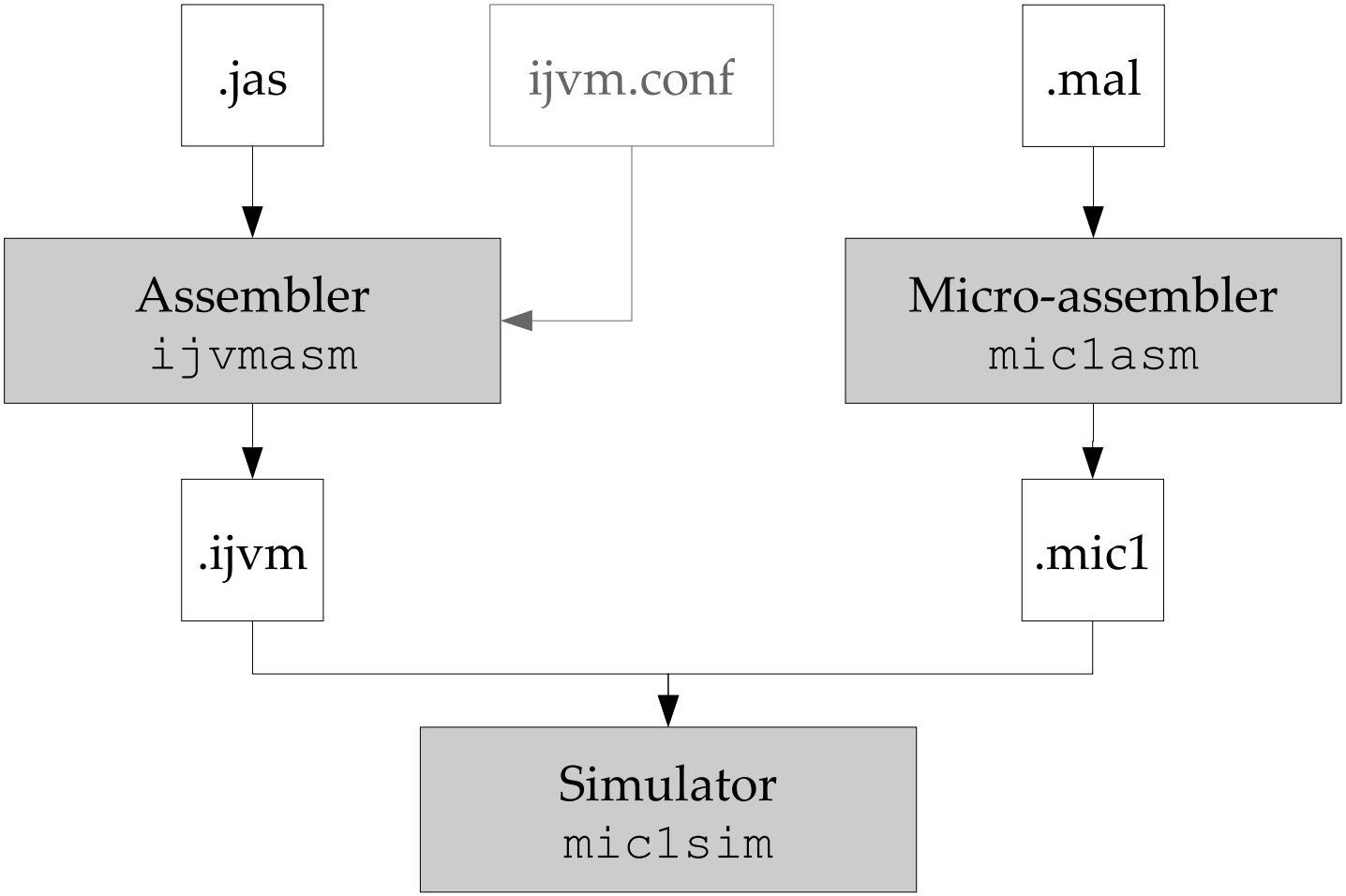
```
.method putint (i)
    // stampo sulla console un
    // intero positivo fra 0 e 9999

.var
          digit
.end-var


          BIPUSH 0
          ISTORE digit
p1000:    ILOAD  i
          LDC_W  c1000
          ISUB
          DUP
          IFLT   e1000
          ISTORE i
          IINC   digit 1
          GOTO   p1000
e1000:    POP
          ILOAD  digit
          BIPUSH 48
          IADD
          OUT

          BIPUSH 0
          ISTORE digit
p100:     ILOAD  i
          LDC_W  c100
          ISUB
          DUP
          IFLT   e100
          ISTORE i
          IINC   digit 1
          GOTO   p100
e100:     POP
          ILOAD  digit
          BIPUSH 48
          IADD
          OUT
          [...]

          BIPUSH 0
          ISTORE digit
p1:       ILOAD  i
          LDC_W  c1
          ISUB
          DUP
          IFLT   e1
          ISTORE i
          IINC   digit 1
          GOTO   p1
e1:       POP
          ILOAD  digit
          BIPUSH 48
          IADD
          OUT

          LDC_W cr
          OUT
          LDC_W lf
          OUT
          IRETURN
.end-method
```

Esercizio 3: soluzione (3/3)

```
.method getint ()
    // leggo un intero dalla
    // console
.var
          units
          sum
          temp
.end-var

          BIPUSH 0
          ISTORE units
          BIPUSH 0
          ISTORE sum

loop:     LDC_W objref
          INVOKEVIRTUAL getche
          DUP
          LDC_W lf
          IF_ICMPEQ ret
          BIPUSH 48
          ISUB
          ISTORE units
```

```
    // moltiplico per dieci
    // a somme ripetute
          BIPUSH 0
          ISTORE temp
dec:      ILOAD  sum
          BIPUSH 1
          ISUB
          DUP
          IFLT dec-end
          ISTORE sum
          IINC temp 10
          GOTO dec

dec-end:  POP
          ILOAD temp
          ILOAD units
          IADD
          ISTORE sum
          GOTO loop

ret:      POP
          ILOAD sum
          IRETURN
.end-method
```

Micro- e Macro-programmi



Aggiunta di nuove istruzioni

- Esempio: aggiunta dell'istruzione `IADD3`
- Aggiunta a `ijvm.conf`:

```
.label     iadd3_1          0x01
```

- Aggiunta a `mic1ijvm.mal`:

```
iadd3_1   MAR = SP = SP-1; rd    // Read next-to-top word on stack
iadd3_2   H = TOS                // H = top of stack
iadd3_3   MDR = TOS = MDR+H; wr  // Add top two words; write to top
iadd3_4   MAR = SP = SP-1; rd    // Read next-to-top word on stack
iadd3_5   H = TOS                // H = top of stack
iadd3_6   MDR = TOS = MDR+H; wr; goto Main1
                    // Add top two words; write to top of stack
```