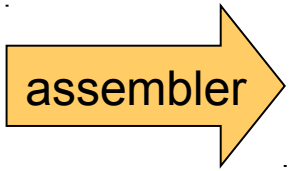
Assemblatore per il linguaggio Hack

Codice sorgente (esempio)

```
// Computes 1+...+RAM[0]
// and stores the sum in RAM[1]
    @i
    M=1    // i = 1

    @sum
    M=0    // sum = 0
(LOOP)
    @i     // if i>RAM[0] goto WRITE
    D=M
    @R0
    D=D-M
    @WRITE
    D;JGT
    ...    // Etc.
```



Codice macchina

```
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
  ...
```

Prof. Ivan Lanese

Assembler

- L'assembler si occupa della traduzione del linguaggio assembly (human readable) al relativo codice in linguaggio macchina (machine executable)
  - Nel caso del linguaggio Hack, ogni istruzione in linguaggio assembly viene tradotta in un codice a 16 bit, ad esclusione delle direttive di dichiarazione di "labels" che semplicemente definiscono dei simboli
  - In input all'assemblatore forniamo un file in linguaggio assembly Hack (estensione .asm), in output l'assemblatore genera un file contenente per ogni riga un numero binario a 16 bit scritto in ASCII (estensione .hack)
    - In un sistema reale non scriverebbe in ASCII ma in binario
  - I file con estensione .hack possono essere caricati in CPUEmulator per essere eseguiti

Programma assembly

```
// Computes 1+...+RAM[0]
// and stores the sum in RAM[1].
    @i
    M=1   // i = 1

    @sum
    M=0   // sum = 0
(LOOP)
    @i    // if i>RAM[0] goto WRITE
    D=M
    @0
    D=D-M
    @WRITE
    D;JGT
    @i    // sum += i
    D=M
    @sum
    M=D+M
    @i    // i++
    M=M+1
    @LOOP // goto LOOP
    0;JMP
(WRITE)
    @sum
    D=M
    @1
    M=D  // RAM[1] = the sum
(END)
    @END
    0;JMP
```

Programma assembly =

file con un sequenza di linee di testo; le linee rilevanti contengono **una** fra le seguenti cose:

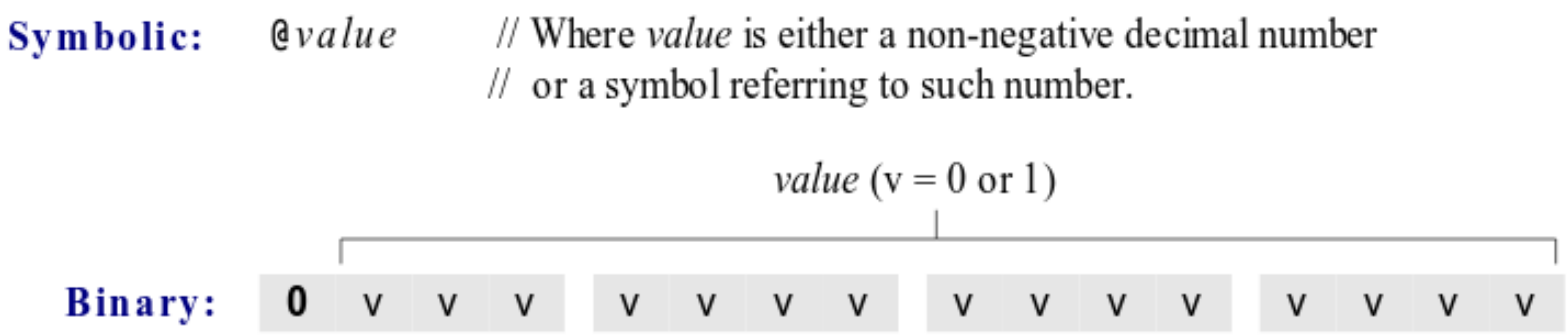
- A-instruction
- C-instruction
- dichiarazione etichette

Possono poi essere presenti commenti, spazi bianchi, simboli di tabulazione '\t', o simboli di fine linea come '\r' o '\n' :

- Commenti iniziano con: //

Sfida: Tradurre tali file in sequenze di istruzioni a 16 bit eseguibili dal nostro calcolatore Hack

Traduzione di A-instructions



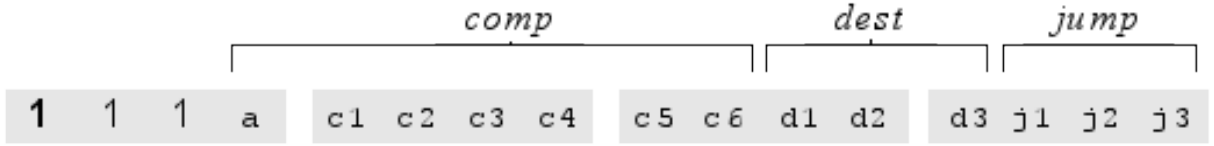
Traduzione in codice macchina:

- Se *value* è un numero decimale non-negativo, basta considerare per la sua sequenza "v v … v" la rappresentazione di *value* in notazione posizionale binaria tramite 15 bit
- Se *value* è un simbolo, vedremo dopo come procedere

Traduzione di C-instructions

Symbolic: *dest*=*comp*;*jump* // Either the *dest* or *jump* fields may be empty.
// If *dest* is empty, the "=" is ommitted;
// If *jump* is empty, the ";" is omitted.

Binary:



(when a=0) *comp*	c1	c2	c3	c4	c5	c6	(when a=1) *comp*
0	1	0	1	0	1	0	
1	1	1	1	1	1	1	
-1	1	1	1	0	1	0	
D	0	0	1	1	0	0	
A	1	1	0	0	0	0	M
!D	0	0	1	1	0	1	
!A	1	1	0	0	0	1	!M
-D	0	0	1	1	1	1	
-A	1	1	0	0	1	1	-M
D+1	0	1	1	1	1	1	
A+1	1	1	0	1	1	1	M+1
D-1	0	0	1	1	1	0	
A-1	1	1	0	0	1	0	M-1
D+A	0	0	0	0	1	0	D+M
D-A	0	1	0	0	1	1	D-M
A-D	0	0	0	1	1	1	M-D
D&A	0	0	0	0	0	0	D&M
D|A	0	1	0	1	0	1	D|M

d1	d2	d3	*Mnemonic*	*Destination (where to store the computed value)*
0	0	0	null	The value is not stored anywhere
0	0	1	M	Memory[A] (memory register addressed by A)
0	1	0	D	D register
0	1	1	MD	Memory[A] and D register
1	0	0	A	A register
1	0	1	AM	A register and Memory[A]
1	1	0	AD	A register and D register
1	1	1	AMD	A register, Memory[A], and D register

j1 (*out* < 0)	j2 (*out* = 0)	j3 (*out* > 0)	Mnemonic	Effect
0	0	0	null	No jump
0	0	1	JGT	If *out* > 0 jump
0	1	0	JEQ	If *out* = 0 jump
0	1	1	JGE	If *out* ≥ 0 jump
1	0	0	JLT	If *out* < 0 jump
1	0	1	JNE	If *out* ≠ 0 jump
1	1	0	JLE	If *out* ≤ 0 jump
1	1	1	JMP	Jump

Gestione dei simboli

- I simboli sono costiutiti dai seguenti possibili caratteri:
  - Lettere minuscole (comprese fra 'a' e 'z')
  - Lettere maiuscole (comprese fra 'A' e 'Z')
  - Cifre decimali (comprese fra '0' e '9')
  - Simboli particolari: '_', '.', '$'
  - Il primo carattere non può essere una cifra decimale
- I simboli sono usati per identificare indirizzi di istruzioni in ROM (destinazione dei jump) oppure indirizzi in memoria RAM (variabili)
  - Alcuni simboli sono già automaticamente prestabiliti:

R0	0
R1	1
R2	2
...	
R15	15
SCREEN	16384
KBD	24576

SP	0
LCL	1
ARG	2
THIS	3
THAT	4

Simboli definiti dal programmatore

Etichette:

Usate per identificare i punti di destinazione delle istruzioni di salto (vedi LOOP e END nell'esempio). Definite tramite la direttiva (YYY) che definisce l'etichetta YYY a cui verrà assegnato come valore l'indirizzo di memoria ROM in cui verrà caricata la prima istruzione successiva alla direttiva (YYY)

Variabili:

Usate nelle A-instruction per identificare celle di memoria RAM da dedicare a specifiche variabili (vedi "counter" e "x" nell'esempio). Riceveranno un valore che va da 16 in poi. I valori vengono assegnati seguendo l'ordine in cui tali variabili appaiono (nell'esempio "counter" varrà 16, mentre "x" varrà 17)

```
    @R0
    D=M
    @END
    D;JLE
    @counter
    M=D
    @SCREEN
    D=A
    @x
    M=D
(LOOP)
    @x
    A=M
    M=-1
    @x
    D=M
    @32
    D=D+A
    @x
    M=D
    @counter
    MD=M-1
    @LOOP
    D;JGT
(END)
    @END
    0;JMP
```

Simboli predefiniti

Registri virtuali:

I simboli R0,…, R15 sono automaticamente predefiniti per far riferimento agli indirizzi di memoria RAM 0,…,15

Puntatori per I/O:

I simboli SCREEN and KBD fanno riferimento agli indirizzi RAM 16384 e 24576, rispettivamente (indirizzi base della mappa in memoria dello schermo, e della tastiera)

Puntatori di controllo della Virtual Machine:

I simboli SP, LCL, ARG, THIS, and THAT (assenti nell'esempio a fianco) sono automaticamente predefiniti per far riferimento agli indirizzi di RAM 0, 1, 2, 3, e 4, rispettivamente

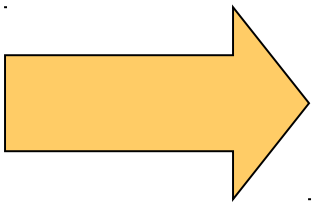
(I puntatori di controllo della VM saranno discussi più avanti quando parleremo di Virtual Machine)

```
  @R0
    D=M
    @END
    D;JLE
    @counter
    M=D
    @SCREEN
    D=A
    @x
    M=D
(LOOP)
    @x
    A=M
    M=-1
    @x
    D=M
    @32
    D=D+A
    @x
    M=D
    @counter
    MD=M-1
    @LOOP
    D;JGT
(END)
    @END
    0;JMP
```

Gestione dei simboli tramite “symbol table”

Codice sorgente (esempio)

```
// Computes 1+...+RAM[0]
// and stores the sum in RAM[1]
    @i
    M=1   // i = 1
    @sum
    M=0   // sum = 0
(LOOP)
    @i    // if i>RAM[0] goto WRITE
    D=M
    @R0
    D=D-M
    @WRITE
    D;JGT
    @i    // sum += i
    D=M
    @sum
    M=D+M
    @i    // i++
    M=M+1
    @LOOP // goto LOOP
    0;JMP
(WRITE)
    @sum
    D=M
    @R1
    M=D  // RAM[1] = the sum
(END)
    @END
    0;JMP
```



Symbol table

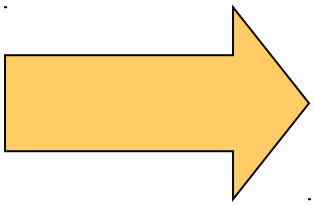
R0	0
R1	1
R2	2
...	...
R15	15
SCREEN	16384
KBD	24576
SP	0
LCL	1
ARG	2
THIS	3
THAT	4
LOOP	4
WRITE	18
END	22
i	16
sum	17

La symbol table viene generata dall’assemblatore e viene usata per tradurre i simboli nei relativi valori

Gestione dei simboli tramite “symbol table”

Codice sorgente (esempio)

```
// Computes 1+...+RAM[0]
// and stores the sum in RAM[1]
    @i
    M=1   // i = 1
    @sum
    M=0   // sum = 0
(LOOP)
    @i    // if i>RAM[0] goto WRITE
    D=M
    @R0
    D=D-M
    @WRITE
    D;JGT
    @i    // sum += i
    D=M
    @sum
    M=D+M
    @i    // i++
    M=M+1
    @LOOP // goto LOOP
    0;JMP
(WRITE)
    @sum
    D=M
    @R1
    M=D  // RAM[1] = the sum
(END)
    @END
    0;JMP
```



Symbol table

R0	0
R1	1
R2	2
...	...
R15	15
SCREEN	16384
KBD	24576
SP	0
LCL	1
ARG	2
THIS	3
THAT	4
LOOP	4
WRITE	18
END	22
i	16
sum	17

Gestione della symbol table:

- iniziale inserimento dei simboli predefiniti
- **prima passata del file in input**: inserimento delle etichette dichiarate tramite direttiva (YYY) con relativo valore
- **seconda passata del file in input**: i simboli in A-instruction non ancora in symbol table vengono aggiunti con relativi valori

Il processo di assemblaggio

- Inizializzazione:
  - Apertura in lettura del file in input
  - Inizializzazione della symbol table con i simboli predefiniti
- Prima passata:
  - Si scorre l'input per inserire le etichette in symbol table: per sapere che valore assegnare alle etichette si contano le A- e le C-instructions (iniziando da 0) e quando si incontra una etichetta le si assegna il valore di tale contatore in quel momento
- Seconda passata:
  - Si apre in scrittura il file di output e si scorre di nuovo l'input:
    - per A- e C-instruction si scrive in output il relativo codice
      - Per i simboli in A-instruction: si cerca il valore in symbol table; se il simbolo è assente si assegna un valore (partendo da 16) e si memorizza il valore in symbol table

Il risultato

Codice sorgente (esempio)

```
// Computes 1+...+RAM[0]
// and stores the sum in RAM[1]
    @i
    M=1   // i = 1
    @sum
    M=0   // sum = 0
(LOOP)
    @i    // if i>RAM[0] goto WRITE
    D=M
    @R0
    D=D-M
    @WRITE
    D;JGT
    @i    // sum += i
    D=M
    @sum
    M=D+M
    @i    // i++
    M=M+1
    @LOOP // goto LOOP
    0;JMP
(WRITE)
    @sum
    D=M
    @R1
    M=D  // RAM[1] = the sum
(END)
    @END
    0;JMP
```

Codice macchina

```
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
1111000010001000
0000000000010000
1111110111001000
0000000000000100
1110101010000111
0000000000010001
1111110000010000
0000000000000001
1110001100001000
0000000000010110
1110101010000111
```

Nota: solo le linee con A- o C-instructions generano un relativo codice macchina

Qualche consiglio

- Procedere per fasi:
  - Implementare prima un assemblatore che non considera i simboli
    - Non serve quindi la symbol table
    - Basta una sola passata
    - Deve generare un file con il medesimo nome dell'input con l'estensione .asm sostituita da .hack
    - Controllare la correttezza usando dei programmi in assembly che non utilizzano simboli
  - Passare poi all'implementazione completa:
    - Implementare (separatamente) la struttura dati symbol table
    - Estendere l'implementazione precedente per fare le due passate come da slide precedenti
    - Controllare correttezza (Pong.asm come esempio non banale)
    - Caricate il file Pong.hack generato dal vostro assemblatore in CPUEmulator e … buon divertimento!