Architettura degli Elaboratori
Pratica

Andrea Malvezzi

Novembre 2024 - ...

Contents

1 Che cos'è l'ISA

L'ISA è l'interfaccia tra HW e SW. Costituisce il set di istruzioni con cui opera la CPU e di conseguenza, varia di architettura in architettura.

2 L'architettura HACK

L'architettura HACK non segue né la filosofia CISC né quella RISC e, data la sua semplicità, in essa ad ogni istruzione eseguita corrisponde un ciclo di clock. Qui si usano una RAM (per contenere i dati di un programma in esecuzione) e una ROM (per contenere il programma *stesso*) a 16 bit.
Si utilizzano prevalentemente tre registri di memoria: **A**, **D** ed **M**.
Il registro M è particolare, in quanto contiene il valore del registro di memoria RAM attualmente puntato da A. Questo si indica con RAM[A].
Oltre a questi due si usa inoltre il **Program Counter**, o **PC**. In esso è contenuto l'indirizzo della prossima istruzione da eseguire e si indica con ROM[PC].

3 Tipi di istruzione

Esistono due tipi di istruzioni:

- A-instruction: quelle che lavorano sulla memoria (che caricano valori in A);
- C-instruction: quelle che eseguono un instruzione (sfruttando l'ALU) prelevando gli operandi dai registri A,D ed M.

Le C-instruction solitamente seguono la forma generale dest = comp; jump, dove:

- **dest**: opzionale. Specifica dove memorizzare il risultato dell'operazione;
- **comp**: specifica l'operazione da eseguire;
- **jump**: opzionale. Specifica se, dopo aver eseguito dest = comp occorre fare un salto nel programma alla linea specificata dal valore del registro A (esegue PC = A). Funziona effettivamente come un *if* eseguito sempre rispetto allo zero.

3.1 Tutte le possibili operazioni

A seguire tutte le possibili operazioni eseguibili nell'architettura HACK.

comp = `0, 1, -1, D, A, !D, !A, -D, -A, D+1, A+1, D-1, A-1, D+A, D-A, A-D, D&A, D|A, M, !M, -M, M+1, M-1, D+M, D-M, M-D, D&M, D|M`

dest = `M, D, MD, A, AM, AD, AMD` o nullo (in questo caso dest viene omesso)

jump = `JGT, JEQ, JGE, JLT, JNE, JLE, JMP` o nullo (omesso)

Figure 1: Se un'operazione che si desidera eseguire non compare qui, allora occorre cambiare approcio.

3.1.1 Esempio di operazione

$$@2$$
$$\mathrm{D} = \mathrm{D} + 1; \mathrm{JLT}$$

Questa riga di codice aumenta di 1 il valore del registro D, poi controlla che questo valore sia strettamente minore di 0. Nel caso in cui questa condizione risultasse verificata, verrà effettuato un JUMP alla riga corrispondente al valore corrente di A, quindi, alla seconda.

4 Le etichette

Un'etichetta è un'istruzione composta di due parti: quella di dichiarazione e quella di chiamata.

4.1 Dichiarazione di etichetta

Un'etichetta fornisce un modo per non dover scrivere a mano (e quindi commettere errori) ogni A-instruction prima di effettuare un salto. Difatti, l'etichetta prende il valore del numero di riga in cui viene dichiarata, permettendo poi di saltare a quella eventuale riga qualora risultasse necessario.

4.1.1 Esempio di dichiarazione di etichetta

(MIA_ETICHETTA)

. . . codice . . .

Qui l'etichetta viene definita prima di un blocco di codice e prende il nome di MIA_ETICHETTA. Questa etichetta avrà quindi ora il valore della riga a cui è stata dichiarata (quella prima del blocco di codice).

4.2 Utilizzo dell'etichetta

Questa etichetta contiene ora il numero della riga in cui è stata dichiarata e, in quanto tale, fornisce un modo per saltare a tale riga mediante una istruzione JUMP.

4.2.1 Esempio di utilizzo dell'etichetta

@MIA_ETICHETTA

0; JMP

Questo ci permette di assegnare ad A il valore della riga in cui l'etichetta è stata dichiarata, per poi eseguire un salto incondizionato (0 è sempre zero, quindi il JMP verrà sempre effettuato) alla riga corrispondente all'inizio del blocco di codice sottostante all'etichetta.

5 Costrutti della programmazione

5.1 If-then-else

Si può ricreare il costrutto if-then-else tramite JUMP, A/C-instruction ed etichette.

```
    // Costrutto if-then-else
    D=1             // Dato da testare

    @CASE_FALSE     // A=riga del caso false
    D;JLE           // Se D <= 0, JUMP al valore di A

    // Se salto, questo blocco non viene eseguito
    ...caso true...

    @END            // A=riga fine codice
    0;JMP           // Salto incondizionato

    (CASE_FALSE)    // Dichiaro l'etichetta
    ...caso false...

    (END)           // Dichiaro l'etichetta
    ...fine codice...
```

5.2 Ciclo while

Anche il ciclo while si può ricreare mediante JUMP, A/C-instruction ed etichette, seppure in maniera leggermente più complessa.

```
1   // Ciclo while
2   D=10              // Variabile per iterazione
3
4   (LOOP)           // Dichiaro inizio loop
5   D; JLE           // JMP se D<=0
6
7   ...codice...     // Se salto, questo viene saltato
8
9   @LOOP            // Al termine del codice, rifai
10  0;JMP            // Salto incondizionato
11
12  (EXIT)           // Dichiaro uscita loop
13  ...altro codice...
```

6 Cose a cui prestare attenzione

6.1 Fine dei programmi

Di base i programmi non finiscono, ma continuano ad incrementare il PC fino a che non si esaurisce la ROM. Per evitare questa behaviour, è buona pratica terminare un programma con un ciclo dummy che previene quindi di andare ad eseguire istruzioni scritte nelle celle della ROM successive.

```
// Ciclo dummy
...codice programma...

// Dichiaro un ciclo infinito
(END)       // Dichiaro fine programma
    @END    // A=Riga precedente
    0;JMP   // Salto incondizionato
```

6.2 Comparazioni tra valori diversi da 0

Come visto in precedenza, esistono solo comparazioni rispetto lo 0. Per ovviare a questa limitazione, si possono usare la somma e la sottrazione.

```
// A <= 10?
@2      // A=2
D=10    // D=10
D=D-A   // D=D-A, quindi D=10-2

@ELSE   // Preparo A per saltare all'else case
D;JLE   // Se D<=0, JUMP

...caso true...
@END    // Termina programma
0;JMP   // Salto incondizionato

(ELSE)  // Dichiaro else case
    ...caso false...

(END)   // Dichiaro ciclo dummy finale
    @END
    0;JMP
```

7 Le variabili

Anche in asm HACK si hanno le variabili, anche se funzionano in modo leggermente diverso. Qui si definiscono come A-instruction e si va a modificare il valore interno al loro registro.

```
    // Esempio di uso delle variabili
    @i          // Variabile i
    M=1         // Registro di i pari ad 1 (RAM[i]=1)

    (LOOP)      // Dichiaro inizio loop
        @i      // uso i
        D=M     // D=Valore registro di i
        @10     // A=10
        D=D-A   // D=D-A, quindi D=i-10

        @END    // Preparo a terminare il programma
        D;JGT   // Se D>0, JUMP

        ...codice loop...
        @i      // Uso i
        M=M+1   // Aumento di 1 il valore di RAM[i]
        ...codice loop...

        @LOOP   // Preparo per ricominciare
        0;JMP   // Salto incondizionato

        (END)   // Dichiaro ciclo dummy finale
            @END
            0;JMP

        // La variabile i serve ad iterare nel loop
        // fino a quando non equivale ad 11 (i-10>10)
```

Esistono inoltre variabili pre-definite per accedere in maniera più esplicita alla memoria RAM, come @Rn, dove n va da 0 a 15.

8 Cose a cui prestare attenzione

8.1 Naming conventions

Generalmente le variabili hanno nomi in minuscolo, mentre le etichette in maiuscolo.

8.2 Nota sull'uso delle variabili

In termini pratici, i primi 16 indirizzi della memoria RAM (da @R0 a @R15) sono riservati per l'esecuzione del programma in esecuzione. Tutti gli indirizzi successivi saranno utilizzati da eventuali variabili, a scelta dell'assemblatore. Per questo motivo, qualora in un programma ci si ritrovasse a dover scrivere qualcosa come @16 oppure un qualunque valore superiore a 15, allora non si dovranno utilizzare variabili, in quanto queste rischierebbero di essere sovrascritte.

9 Il computer HACK

Il computer HACK sfrutta l'architettura Von Neumann a 16-bit, con una feature di quella Harvard: la memoria dati viene separata da quella programma, in modo da poter caricare contemporaneamente dati e istruzioni.
Questo viene reso possibile dal bus dati che, mentre nelle architetture moderne svolge due compiti distinti e quindi non eseguibili in parallelo, qui ne svolge solamente uno. Conseguentemente per eseguire un'istruzione basta un ciclo di clock.

10 La memoria

10.1 La ROM

La ROM corrisponde ad un chip built-in che dato un address 15-bit codifica in binario un'istruzione da eseguire, quindi: out = ROM32K[$address$].

10.2 Chip memory

Il chip memory è composto da un totale di 3 componenti, per un totale di 24K indirizzi:

- la RAM da 16K;
- il chip **Screen** da 8K, usato per mappare lo schermo;
- un singolo registro **Keyboard** per leggere il tasto premuto;

Tuttavia, avendo in ingresso un address da 15 bit da convertire in binario, si sprecano molti indirizzi così facendo: $2^{15} = 32767$ meno i 24576 occupati dal chip memory, per un totale di 8191 indirizzi sprecati. Cosa accade a questi?

10.2.1 Il chip Screen

All'interno del chip Screen si ha una corrispondenza diretta tra il singolo bit e un pixel sullo schermo. In base al valore di questi bit, lo schermo viene costsantemente refreshato colorando di bianco (0) o di nero (1) ogni singolo pixel.
Per impostare il pixel in posizione (row, col) dello schermo ad un certo colore, occorrerà quindi settare il bit *col*%16 della word all'indirizzo Screen[row*32 + col/16] a 1 oppure a 0.
Ma come mai questi numeri? Anzitutto, occorre specificare che nel computer HACK lo schermo è composto da 256 righe e 512 colonne.
Questo significa che in una singola riga ci saranno 512 caratteri, raggruppati in parole (che in informatica hanno spesso una grandezza di 16 bit), da cui si evince che in una singola riga si avranno 32 parole (512/16 = 32). Ed ecco spiegato il **row*32**.
Lo stesso ragionamento va applicato alle righe: avendo 256 righe, si avranno 16 (256/16 = 16) parole per colonna, da cui **col/16**.

10.2.2 Esempio di codice per disegnare un rettangolo largo 16 pixel e alto MEM[0] pixel

```
1    @0                // A=0
2    D=M               // D=RAM[0]
3
4    @INFINITE_LOOP    // Preparo il salto
```

```
5       D; JGE          // Se D>0
6
7       @counter        // Variabile counter per iterare
8       M=D             // RAM[counter]=RAM[0]
9
10      @SCREEN         // Predefinita dal linguaggio
11      D=A             // D=Indirizzo Screen
12      @address        // Variabile address
13      M=D             // RAM[address]=Indirizzo Screen
14
15      (LOOP)
16          @address    // Variabile address
17          A=M         // A=Indirizzo Screen
18          M=-1        // MEM[Indirizzo Screen]=-1
19
20          @32         // A=32
21          D=A         // D=32
22          @address    // Variabile address
23          M=M+D       // Address+=32
24
25          @counter
26          MD=M-1      // Counter--, D per check sul jump
27
28          @LOOP
29          D;JGT       // Torno all'inizio
30
31      (END)           // Ciclo dummy finale
32          @END
33          0;JMP
```

10.2.3 Il chip Keyboard

Come annunciato precedentemente questo chip è composto da un solo registro e fornisce in output la codifica ASCII estesa del tasto premuto, oppure 0 se non si preme alcun tasto.
Il registro è read-only e vi sono alcuni tasti con codifiche particolari, come lo spazio.

11 Comportamento della CPU

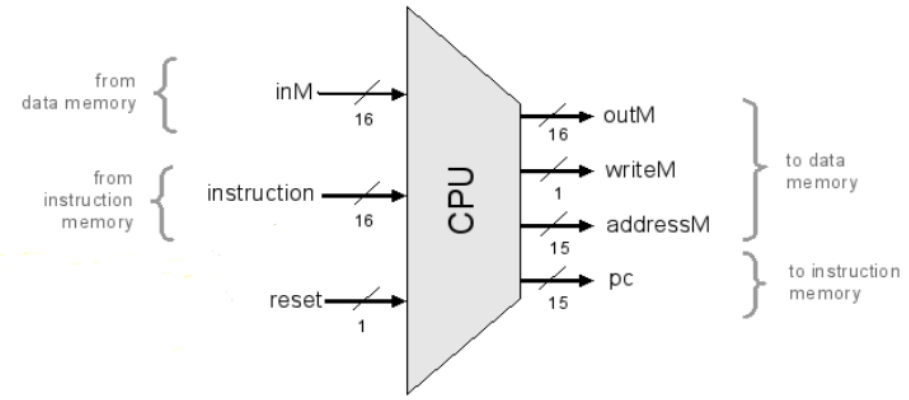


Figure 2: Schema logico CPU nell'architettura HACK.

La CPU è composta da 4 componenti: l'ALU e 3 registri, quali A, D, PC. Questa esegue istruzioni seguendo le specifiche del linguaggio HACK (nel nostro caso). Ovvero:

- i valori D ed A, se presenti, sono letti e/o scritti nei registri A ed M;
- il valore M, se presente nella right side dell'istruzione da eseguire, viene LETTO da inM (input-M);
- se invece il valore M risulta presente nella left side dell'istruzione da eseguire, allora l'output verrà scritto in outM, A prenderà il valore di addressM e writeM verrà alzato ad 1.

12 Codifica A-Instruction

In HACK le A-instruction sono codificate nella maniera seguente: il bit di segno (o modulo) è sempre pari a 0, mentre i 15 valori a seguire saranno cifre binarie.

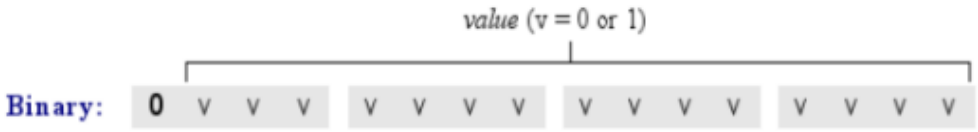


Figure 3: Codifica delle A-instruction nell'architettura HACK.

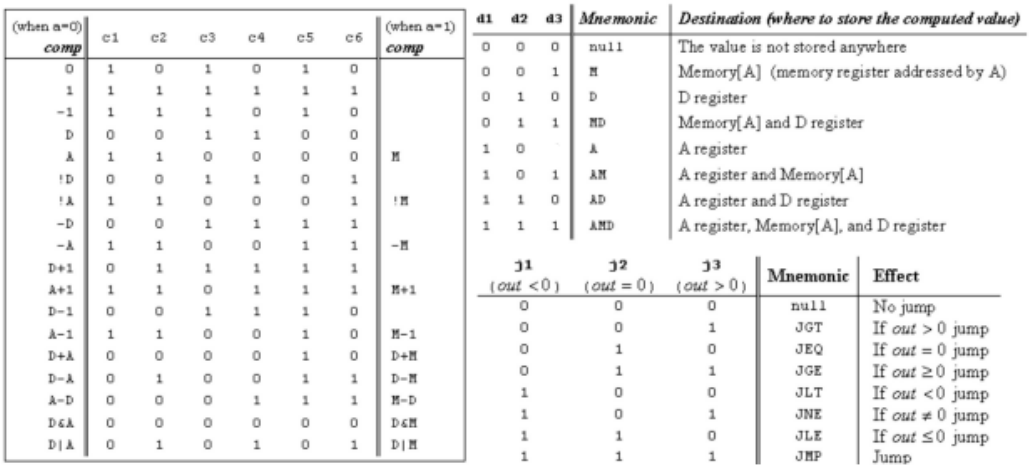
Tramite questo approcio si possono codificare tutti i valori non negativi codificabili in complemento a 2 con 16 bit (si ha effettivamente un bit in meno, quello modulo).

13 Codifica C-instruction

Nell'architettura HACK le C-instruction sono codificate nella maniera seguente: il modulo e le due cifre subito dopo di esso sono **sempre** pari ad 1, ed i valori seguenti sono posti ad 1 solamente quando alcune caratteristiche vengono rispettate.

Figure 4: Codifica delle C-instruction nell'architettura HACK.

E come possibile vedere dalla figura, il corpo effettivo dell'istruzione è composto da valori impostati a seconda della presenza di alcune direttive. Ma come sono decisi questi valori? tramite la seguente tabella:



(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		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

Figure 5: Ad esempio un'istruzione contenente un JGT avrebbe gli ultimi 3 bit (quelli di *jump*) pari a 001.

14 Che cos'è l'Assembler

L'assembler si occupa di tradurre da linguaggio Assembly a binario il nostro codice. Si passa quindi da istruzioni, etichette e simboli speciali a sequenze di 16 bit, generando in output un file *.hack*.
Oltre a quanto citato, l'assembler si occupa anche di tradurre spazi, righe vuote e simboli come $\backslash n$ o $\backslash t$ (ignorando quindi i commenti).

15 Traduzione di A-instructions

Ricordando la traduzione delle A-instructions spiegata precedentemente, basta porre il primo bit della sequenza a 0 e poi tradurre il numero specificato nel codice. Ad esempio:

$$@3 \Rightarrow 0000000000000011$$

16 Traduzione di C-instructions

Non occorre imparare a memoria la tabella, ma bensì conoscere a grandi linee il significato di una traduzione in bit, qualora se ne trovasse una. Ad esempio:

$$D = A \Rightarrow 1110011000010000$$

Dove le cifre in rosso sono valori fissi delle C-instructions.
Quelli in blue sono le istruzioni inerenti alla *comp*, ovvero l'istruzione effettiva da eseguire (in questo caso A), con davanti il valore di a (qui è 0).
Quella gialla sono la parte *comp*, ovvero la destinazione dove salvare il risultato dell'operazione (nel nostro caso D).
La parte verde è la sezione per il *jump*. Ovviamente, non avendo un jump, ogni bit sarà posto a 0.

17 Gestione dei simboli

I simboli si usano per identificare certi indirizzi nella ROM (destinazione salti) o nella RAM (dove son contenute le variabili).
Alcuni simboli potrebbero essere le lettere, le cifre o caratteri speciali come _,-,. etc ... L'unica accortezza consiste nel non scrivere una cifra all'inizio di un simbolo (ed ecco spiegato perché non si può cominciare il nome di una variabile con un numero).

17.1 Simboli predefiniti

Alcuni simboli sono pre-esistenti nel linguaggio, ovvero ...

- ... I registri virtuali: I simboli da R0 a R15 fanno riferimento agli indirizzi RAM da 0 a 15;
- ... I puntatori per I/O: I simboli SCREEN e KBD fanno riferimento rispettivamente agli indirizzi RAM 16384 e 24576;
- ... I puntatori di controllo della VM: I simboli SP, LCL, ARG, THIS e THAT fanno riferimento agli indirizzi RAM da 0 a 4, rispettivamente.

17.2 Simboli NON predefiniti

Ci sono poi anche simboli non definiti dal linguaggi, ovvero le etichette e le variabili.
Le prime sono usate per identificare la destinazione dei JUMP e a queste è assegnato come valore l'indirizzo della ROM in cui verrà caricata la prossima istruzione da eseguire. Ad esempio:

```
1    // Valore: numero della riga contenente
2    // l'istruzione seguente
3    (LOOP)
4        istruzione 1...
5        istruzione 2...
6        istruzione 3...
```

Le seconde sono usate nelle A-instructions per identificare celle della memoria RAM a partire dalla 16-esima. Riceveranno quindi un valore che va dal 16 in poi. Ad esempio:

```
1    @counter        // Valore: 16
2    ...codice...
3    @max            // Valore: 17
4    ...codice...
5    @min            // Valore: 18
6    ...codice...
```

18 Procedura di assemblaggio

Il processo di assemblaggio è diviso in 3 fasi:

18.1 Inizializzazione

- si apre in lettura il file .asm in input;
- si inizializza la symbol table dei simboli predefiniti;

18.2 Prima passata

Si scorre l'input per inserire nella symbol table le codifiche delle etichette incontrate. Per farlo si usa un counter del totale delle A/C-instructions

incontrate, assegnando a ogni etichetta il valore corrente del counter +1.

18.3 Seconda passata

Si apre in writing il file .hack prodotto e si scorre nuovamente l'input. Per ogni A/C-instruction incontrata ne si scrive nell'out la relativa codifica. Inoltre qualora una A-instruction usasse un simbolo, si dovrebbe ricercare questo nella symbol table e, nel caso in cui questo sia assente (quindi si sta parlando di una variabile) si assegna a questa istruzione un valore progressivo a partire da 16, per poi memorizzare questo nella symbol table.

18.4 Esempio di assemblaggio

N.B. questa non è una tipologia di esercizio ma bensì solamente un esempio per comprendere meglio quanto appena spiegato.

```
1     @x
2     M=1
3
4     (CICLO)
5         @1
6         D=M
7         x
8         D=D-M
9         @END
10        D;JLE
11        @CICLO
12        0;JMP
13
14    (END)
15        @END
16        0;JMP
```

Corrisponde alla tabella:

```
1     ...simboli predefiniti...
2     CICLO   2
3     END     15
4     x       16
```

Dove:

- CICLO ha valore di 2 in quanto prima della sua definizione si conta una sola A-instruction, quindi counter + 1 = 2;
- END ha valore di 15 in quanto la riga seguente è la quindicesima;
- x ha valore di 16 in quanto è la prima (ed unica) variabile incontrata nel codice.

19 Esercizi ISA

19.1 Esercizi introduttivi

19.1.1 D=0

```
1    D=0
```

19.1.2 A=1 e D=1

```
1    // Con due istruzioni separate:
2    @0       // A=0
3    D=0      // D=0. Anche D=A andrebbe bene
4
5    // Con la scrittura compatta:
6    AD=0     // Assegno CONTEMPORANEAMENTE A e D a 0
```

19.1.3 D=3

```
1    // Non esiste l'istruzione D=n con n diverso da 1
2    @3  // Prima assegno ad A il valore di 3
3
4    D=A // Poi assegno a D il valore di A
```

19.1.4 D=RAM[3]+3

```
1    @3       // Assegno ad A valore pari a 3
2    D=M      // D=RAM[3]
3    D=D+A    // D=D+A, quindi D=RAM[3]+3
```

19.1.5 D=3+4

```
    @3      // A=3
    D=A     // D=A, quindi D=3
    @4      // A=4
    D=D+A   // D=D+A, quindi D=3+4
```

19.1.6 RAM[3]=7

```
    @7      // A=7
    D=A     // D=7
    @3      // A=3
    M=D     // RAM[3]=7
```

19.1.7 D=RAM[A]+3

```
    D=M     // Supponendo che A sia dichiarata a priori
    @3      // A=3
    D=D+A   // D=D+A, quindi D=RAM[A]+3
```

19.2 Esercizi significativi

19.2.1 D=D-3

```
    @3      // A=3
    D=D-A   // Supponendo che D sia dichiarata a priori
```

19.2.2 D=10-RAM[5]

```
    @10     // A=10
    D=A     // D=A, quindi D=10
    @5      // A=5
    D=D-M   // D=D-M, quindi D=10-RAM[5]
```

19.2.3 RAM[0]=RAM[0]+2

```
@2      // A=2
D=A     // D=A, quindi D=2
@0      // A=0
M=M+D   // M=M+D, quindi RAM[0]=RAM[0]+2
```

19.2.4 RAM[RAM[0]]=RAM[0]

```
@0      // A=0
AD=M    // A e D = M, quindi A e D = RAM[0]
M=D     // M=D, quindi RAM[RAM[0]]=RAM[0]
// M equivale a RAM[RAM[0]] in quanto A=RAM[0]
```

19.2.5 RAM[2]=RAM[0]+RAM[1]

```
@0      // A=0
D=M     // D=RAM[0]
@1      // A=1
D=D+M   // D=RAM[0]+RAM[1]
@2      // A=2
M=D     // M=D, quindi RAM[2]=RAM[0]+RAM[1]
// Nessuno vieta di spezzettare il problema!
```

19.3 Esercizi con salti

19.3.1 Se RAM[0]>0, JUMP all'indirizzo in RAM[1]

```
@0      // A=0
D=M     // D=RAM[0]
@1      // A=1
A=M     // A=RAM[1]
D;JGT   // Se RAM[0]>0, JUMP a RAM[1]
```

19.3.2 D=D+1, poi se D=0 JUMP a istruzione 3

```
1    D=D+1    // Incrementa D
2    @3       // A=3
3    D;JEQ    // Se D=0, JUMP a valore di A (quindi 3)
```

19.3.3 RAM[0]=RAM[5] e se RAM[5]<>0 JUMP a istruzione 3

```
@5       // A=5
D=M      // D=RAM[5]
@0       // A=0
M=D      // RAM[0]=RAM[5]
@3       // A=3
D;JNE    // Se RAM[5] != 0, JUMP a valore di A
```

19.4 Esercizi con le etichette

19.4.1 Implementazione moltiplicazione

```
// Volendo eseguire RAM[2] = RAM[0] * RAM[1]:
@2           // A=2
M=0          // RAM[2]=0
@4           // A=4
M=1          // RAM[4]=1, contatore per loop

(LOOP)       // Inizio loop
    @4       // A=4
    D=M      // D=RAM[4], prendo contatore
    @1       // A=1
    D=D-M    // RAM[4]=RAM[4]-RAM[1]

    @END     // Preparo a terminare
    D;JGT    // Se contatore-RAM[1]>0, termina

    @0       // A=0
    D=M      // D=RAM[0], val da moltiplicare
    @2       // A=2
    M=M+D    // sommo RAM[0] a RAM[2]

    @LOOP    // Preparo a rifare
    0;JMP    // Salto incondizionato

(END)        // Ciclo dummy finale
    @END
    0;JMP
```

19.4.2 Implementazione moltiplicazione senza quarto registro

```
    // Volendo eseguire RAM[2] = RAM[0] * RAM[1]:
    @2          // A=2
    M=0         // RAM[2]=0

    (LOOP)      // Inizio loop
        @0      // A=0
        D=M     // D=RAM[0], val da moltiplicare
        @2      // A=2
        M=M+D   // sommo RAM[0] a RAM[2]

        @1
        DM=M-1
        @LOOP
        D;JGT   // Rifai se RAM[1]-1>0

    ...resto codice...
```

19.4.3 RAM[10]=max(RAM[9..0])

```
    @9      // A=9
    D=A     // D=9
    @11     // A=11
    M=D     // RAM[11]=9, contatore
    @9      // A=9
    D=M     // D=RAM[9], il primo valore equivale a max
    @10     // A=10
    M=D     // RAM[10]=RAM[9]

    (LOOP)          // Dichiaro inizio loop
        @11     // A=11
        D=M     // D=RAM[11], contatore
        @END    // Preparo a terminare
        D;JEQ   // Se D equivale a 0, JUMP

        @11     // Rimetto A=11
        M=M-1   // RAM[11]--
        A=M     // A=RAM[11]
```

```
19          D=M      // D=RAM[A]
20          @10
21          D=M-D    // D=(Max)-(valore corrente)
22          @SET     // Preparo a cambiare max
23          D;JLT    // Se D<0, JUMP
24
25          @LOOP    // Preparo a rifare
26          0;JMP    // Salto incondizionato
27
28      (SET)
29          @11      // A=11
30          D=M      // D=RAM[11], contatore
31          A=D      // A=contatore
32          D=M      // D=(valore corrente)
33          @10      // A=10
34          M=D      // Cambio max
35
36          @LOOP    // Preparo a rientrare nel loop
37          0;JMP    // Salto incondizionato
38
39      (END)        // Ciclo dummy finale
40          @END
41          0;JMP
```

Di questo esercizio si trova una copia del codice eseguibile nell'emulatore nella cartella _esercizi.