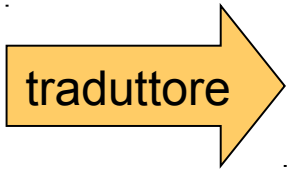
Virtual Machine per il calcolatore Hack

Codice Virtual Machine (esempio)

```
push constant 10
call inc 1
label LOOP
goto LOOP

function inc 0
push argument 0
push constant 1
add
return
```



Codice assembly

```
//bootstrap
      @256
      D=A
      @SP
      M=D
//push constant 10
      @10
      D=A
      @SP
      A=M
      M=D
      @SP
      M=M+1
//call inc 1
      @l0000000000000000
      D=A
      @SP
      A=M
      M=D
      @SP
      M=M+1
      ...
```

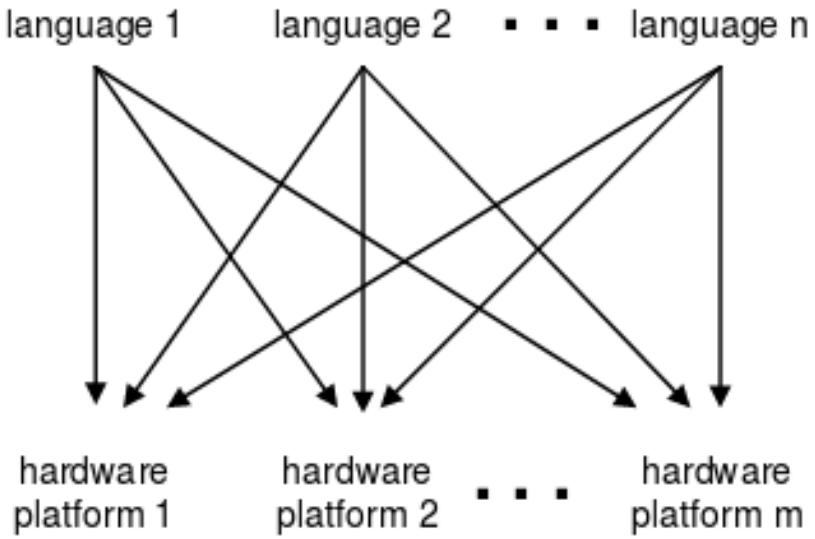
Prof. Ivan Lanese

Virtual machine

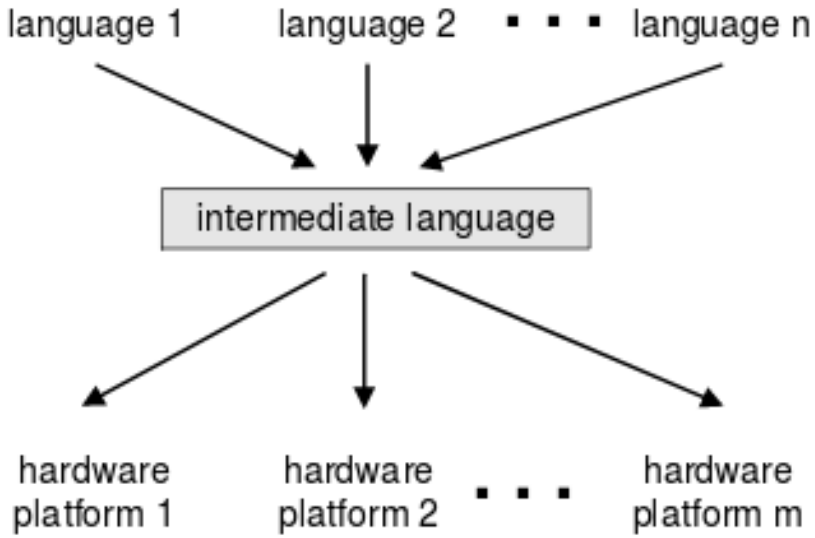
- Una virtual machine è un modello astratto di calcolo
- Non fa riferimento ad una specifica architettura
- Utile step intermedio tra linguaggio ad alto livello e linguaggio assembler
  - Ad esempio, JVM di Java (che è la stessa di Scala e altri, basata su stack) e BEAM di Erlang e Elixir (basata su registri)
- Usata nella compilazione a 2 livelli

Modelli di compilazione

Compilazione diretta



Compilazione a 2 livelli



Compilazione a 2 livelli

- Primo livello: dipende solo dai dettagli del linguaggio sorgente
- Secondo livello: dipende solo dai dettagli del linguaggio target

Il nostro linguaggio per Virtual Machine

Il linguaggio che implementeremo è basato sui seguenti comandi:

Arithmetic / Boolean commands

```
add
sub
neg
eq
gt
lt
and
or
not
```

Memory access commands

```
pop  x   (pop into x, which is a variable)
push y   (y being a variable or a constant)
```

Program flow commands

```
label      (declaration)
goto       (label)
if-goto    (label)
```

Function calling commands

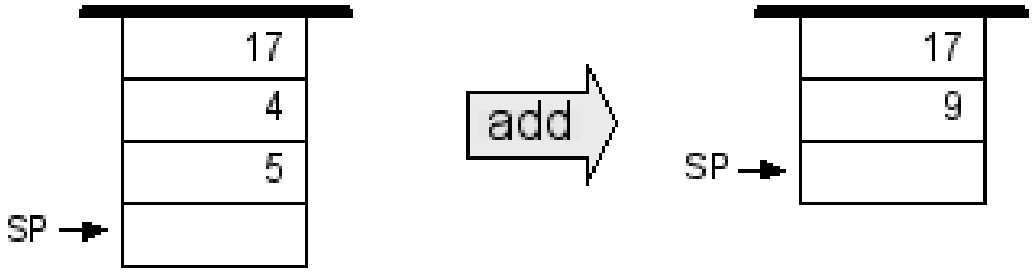
```
function   (declaration)
call       (a function)
return     (from a function)
```

Come procederemo:

(a) specificheremo i comandi a livello di modello

(b) studieremo come implementare i comandi con il computer Hack

Modello a stack



La VM considera un unico tipo di dato a 16 bit che può essere usato per:

- Interi
- Booleani
- Puntatori

Comportamento delle operazioni tipiche della virtual machine:

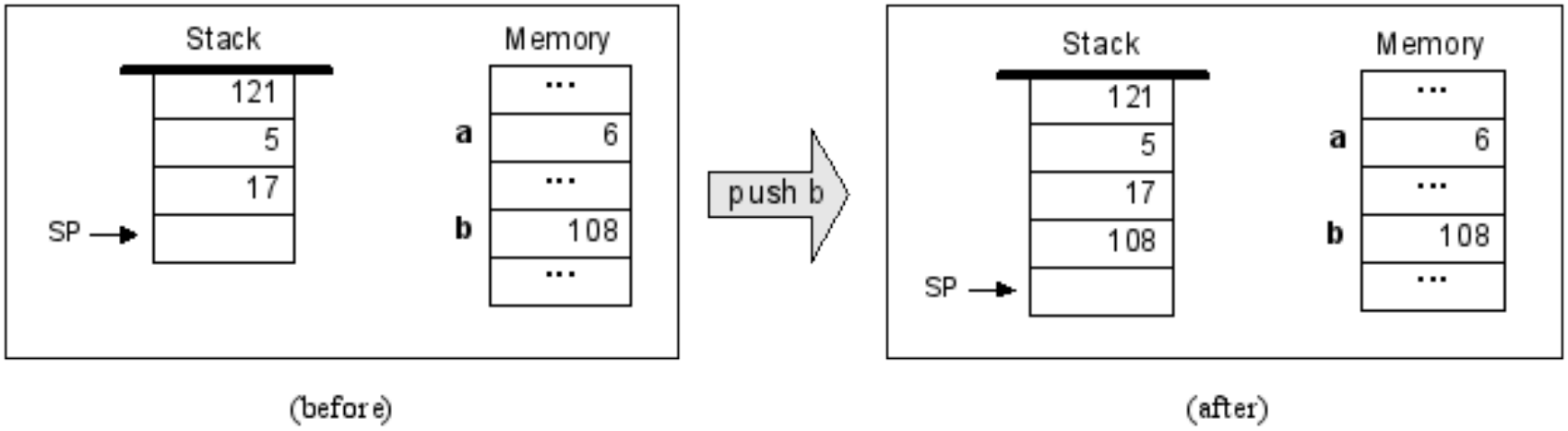
- Pop dei valori x,y dalla cima dello stack
- Calcola il valore di una qualche funzione f(x,y)
- Push del risultato in cima allo stack

(Operazioni unarie sono simili, usando x e f(x) invece di x,y e f(x,y))

Effetto dell'esecuzione dell'operazione:
gli operandi sono rimpiazzati dal risultato dell'operazione

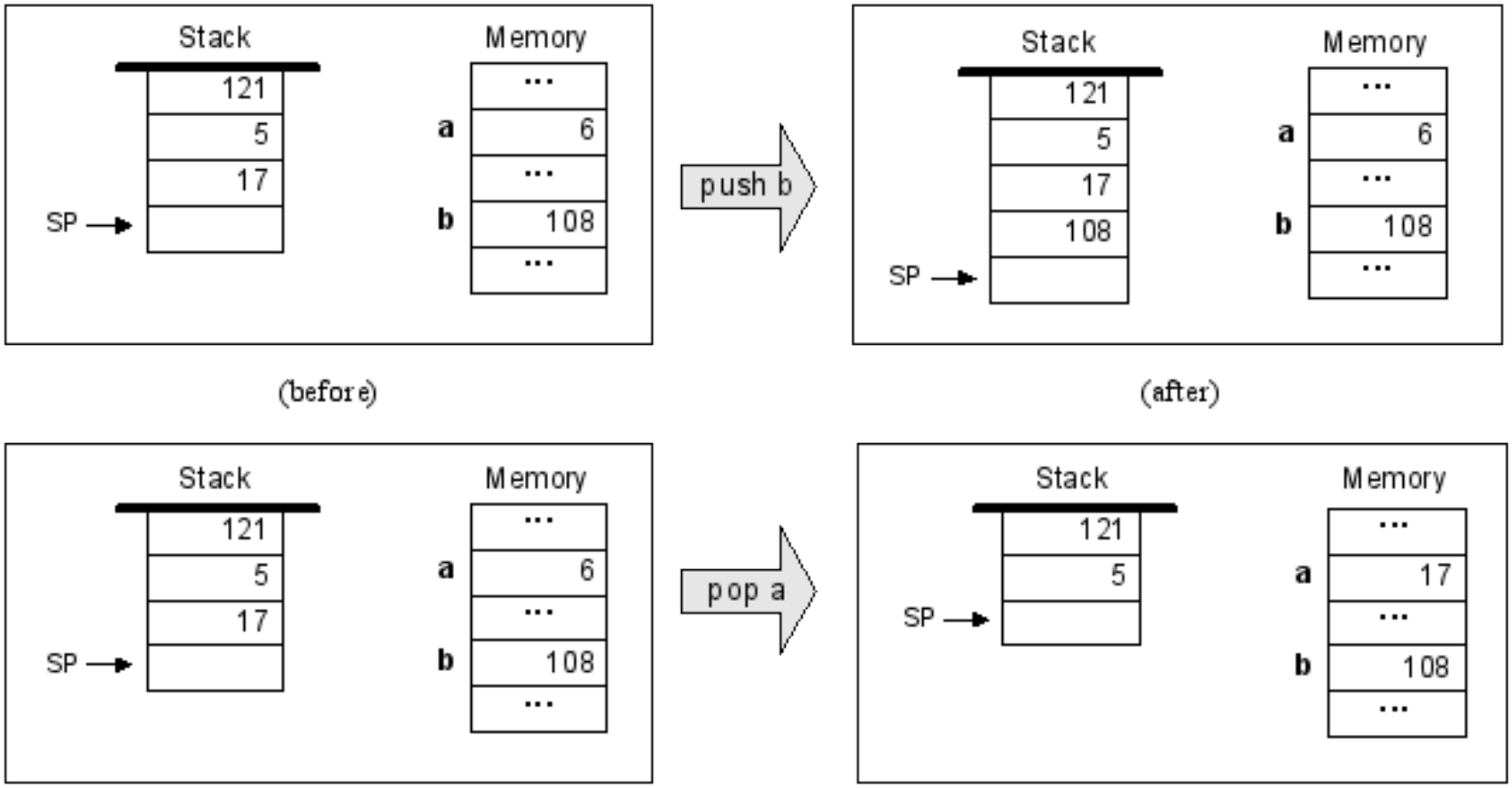
In generale: tutte le operazioni aritmetico-logiche si comportano in questo modo

Comandi di accesso in memoria



(before) (after)

Comandi di accesso in memoria

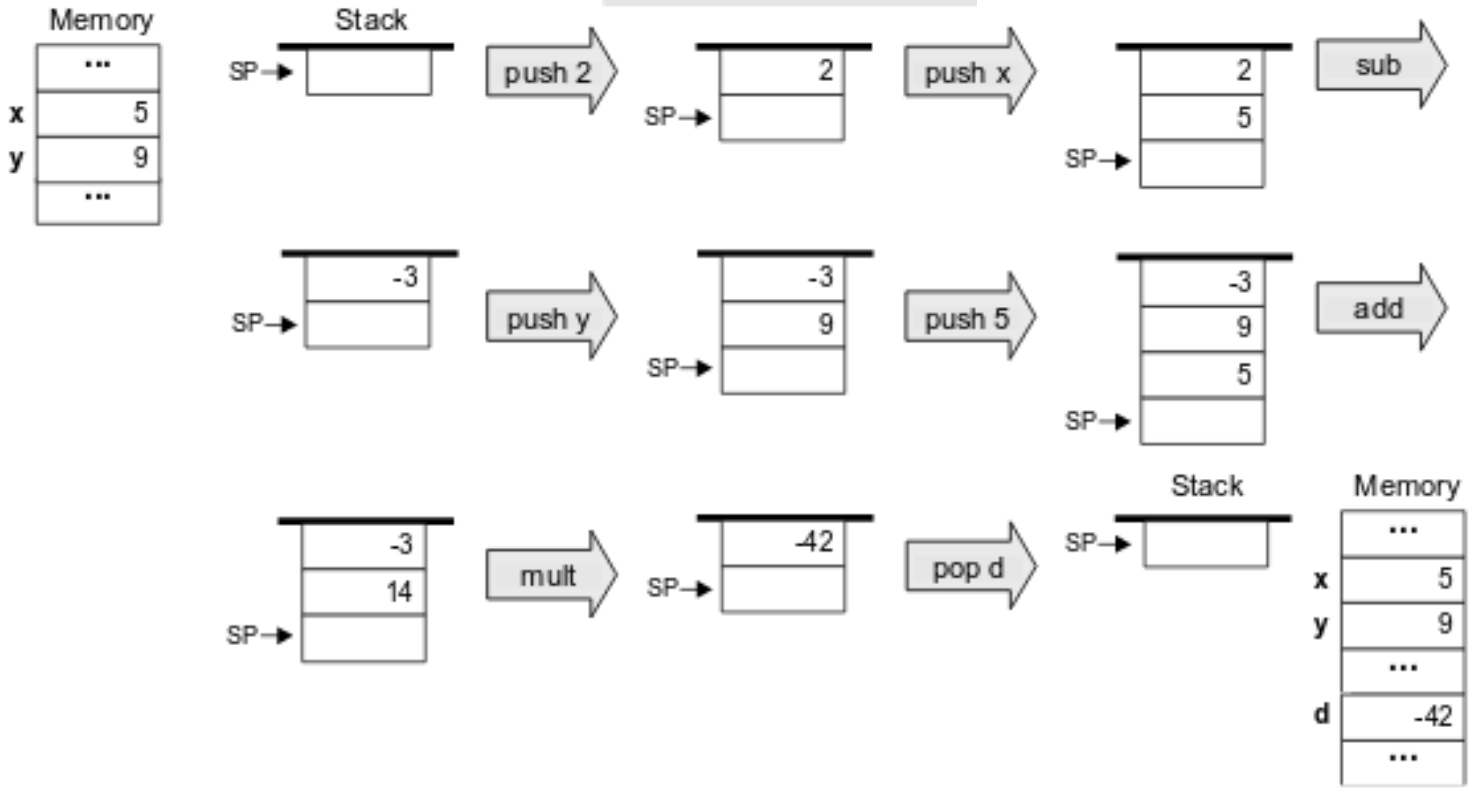


Vedremo che in realtà la gestione della memoria è leggermente più complessa

Valutazione di una espressione aritmetica

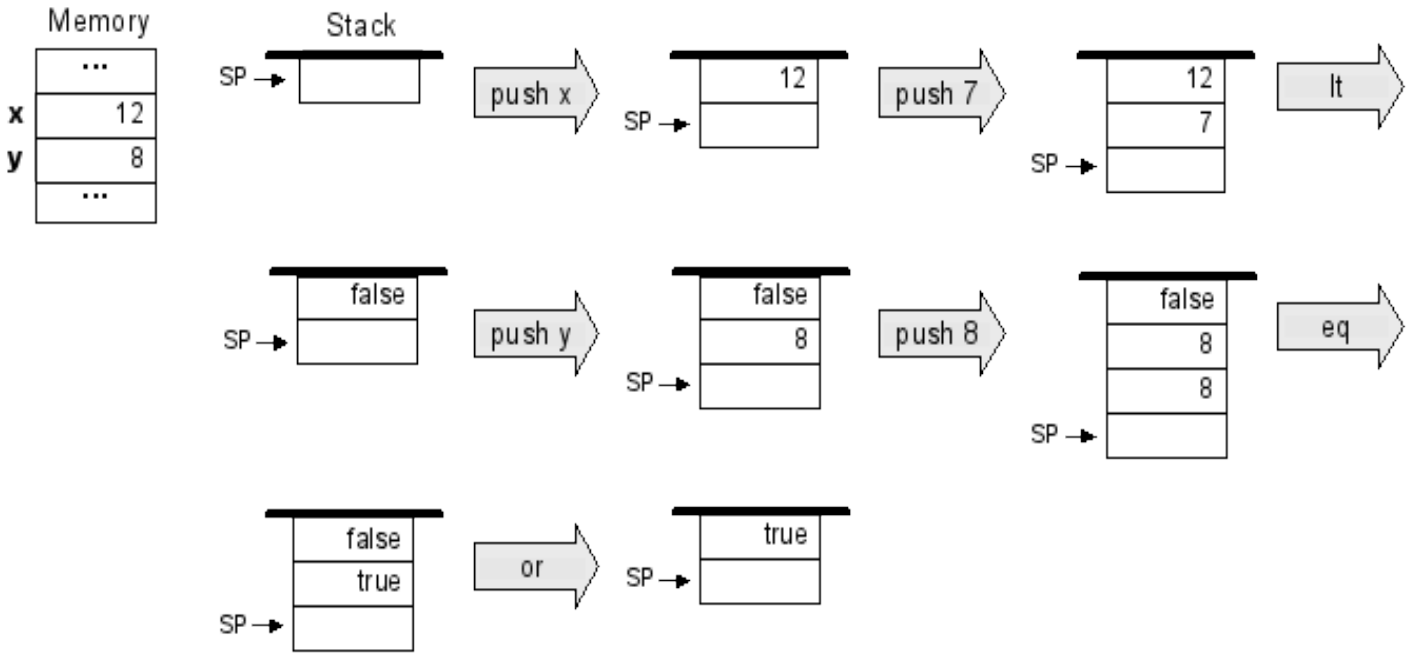
Per comodità assumiamo l'esistenza di una operazione "mult" che nel nostro caso non è presente (definiremo più avanti una funzione "mult" per la moltiplicazione)

```
// d=(2-x)*(y+5)
push 2
push x
sub
push y
push 5
add
mult
pop d
```



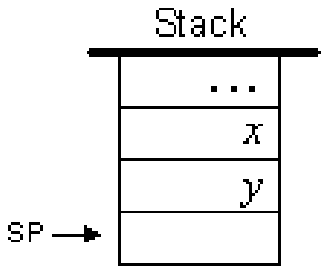
Valutazione di una espressione booleana

```
// if (x<7) or (y=8)
push x
push 7
lt
push y
push 8
eq
or
```



Sommario delle operazioni aritmetico-logiche

Command	**Return value** (after popping the operand/s)	**Comment**
`add`	$x+y$	Integer addition (2's complement)
`sub`	$x-y$	Integer subtraction (2's complement)
`neg`	$-y$	Arithmetic negation (2's complement)
`eq`	true if $x=y$ and false otherwise	Equality
`gt`	true if $x>y$ and false otherwise	Greater than
`lt`	true if $x<y$ and false otherwise	Less than
`and`	x And y	Bit-wise
`or`	x Or y	Bit-wise
`not`	Not y	Bit-wise



Segmenti di memoria

- Utilizzeremo diversi segmenti di memoria
  - Sono spazi di indirizzamento virtuali usati per scopi diversi con dimensioni e caratteristiche differenti
- Ogni funzione in esecuzione avrà due propri segmenti specifici che vengono creati e distrutti prima e dopo l'esecuzione
  - "argument" per gli argomenti (parametri di invocazione)
  - "local" per le proprie variabili locali
- Useremo un segmento "static" condiviso da tutte le funzioni
  - Contiene variabili "globali"
- Per comodità, si assume l'esistenza di un segmento fittizio "constant" da cui si può solo leggere
  - Usato per le costanti (esempi a seguire)

Comandi di accesso alla memoria

Considereremo i seguenti segmenti di memoria virtuale per la nostra macchina virtuale:

`static, local, argument, constant`

Per accedere ai vari segmenti della macchina virtuale si useranno i seguenti comandi:

Memory access command format:

`pop` *segment i*

`push` *segment i*

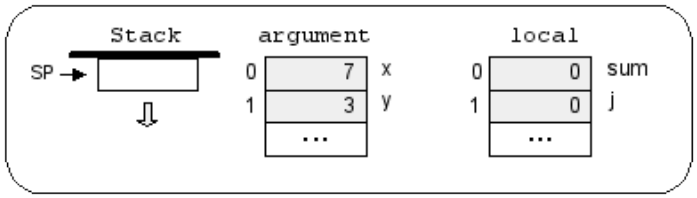
- Al posto di *segment* si mette **`static`**, **`local`**, **`argument`**, oppure **`constant`**
- Al posto di *i* si mette:
  - L'indirizzo di interesse (per local e argument)
  - La costante numerica di interesse (per constant)
  - Per static si usa un valore numerico *n* che viene tradotto nel simbolo *nomeFile.n* assumendo che il file con il programma si chiami *nomeFile.vm*

Esempio di codice per la macchina virtuale

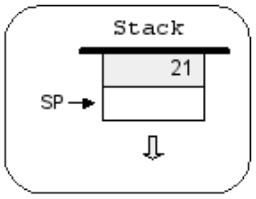
High-level code

```
function mult (x,y) {
  int sum, j;
  sum = 0;
  j = y;
  while (j != 0) {
    sum = sum + x;
    j = j - 1;
  }
  return sum;
}
```

Just after mult(7,3) is entered:



Just after mult(7,3) returns:



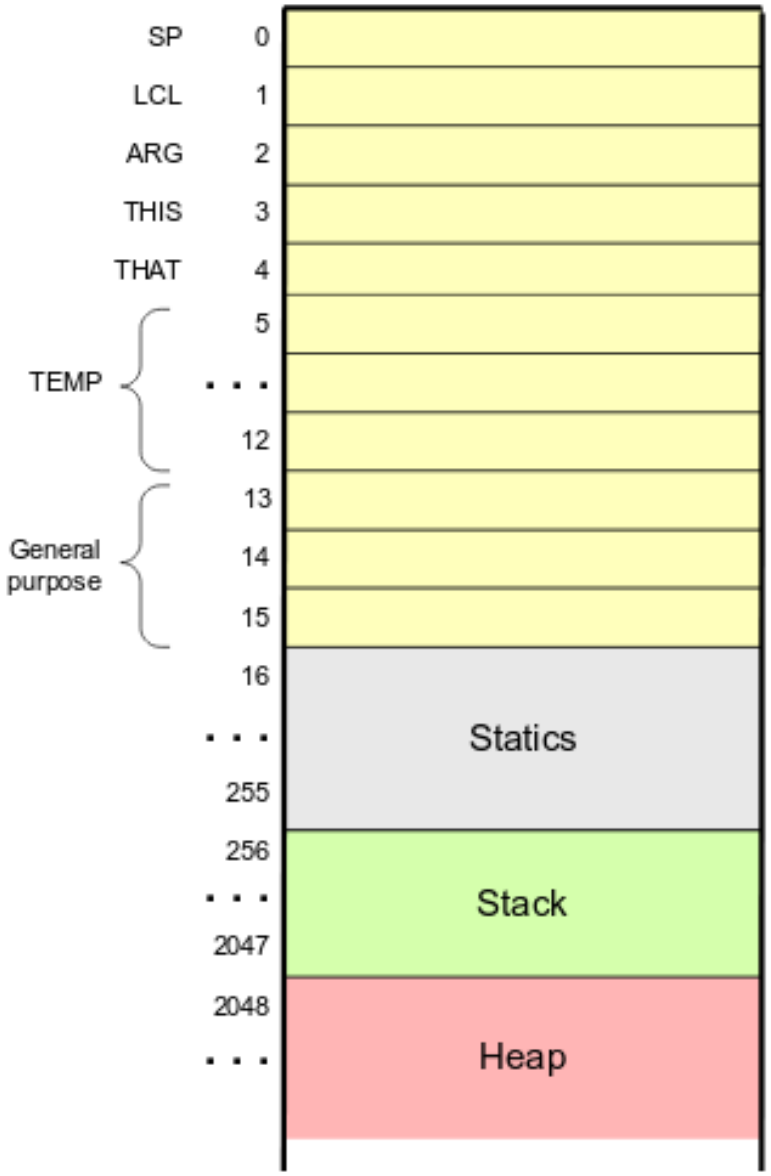
VM code(first approx.)

```
function mult(x,y)
  push 0
  pop  sum
  push y
  pop  j
label  loop
  push j
  push 0
  eq
  if-goto end
  push sum
  push x
  add
  pop  sum
  push j
  push 1
  sub
  pop  j
  goto loop
label  end
  push sum
  return
```

VM code

```
function mult 2
  push   constant 0
  pop    local 0
  push   argument 1
  pop    local 1
label    loop
  push   local 1
  push   constant 0
  eq
  if-goto end
  push   local 0
  push   argument 0
  add
  pop    local 0
  push   local 1
  push   constant 1
  sub
  pop    local 1
  goto   loop
label    end
  push   local 0
  return
```

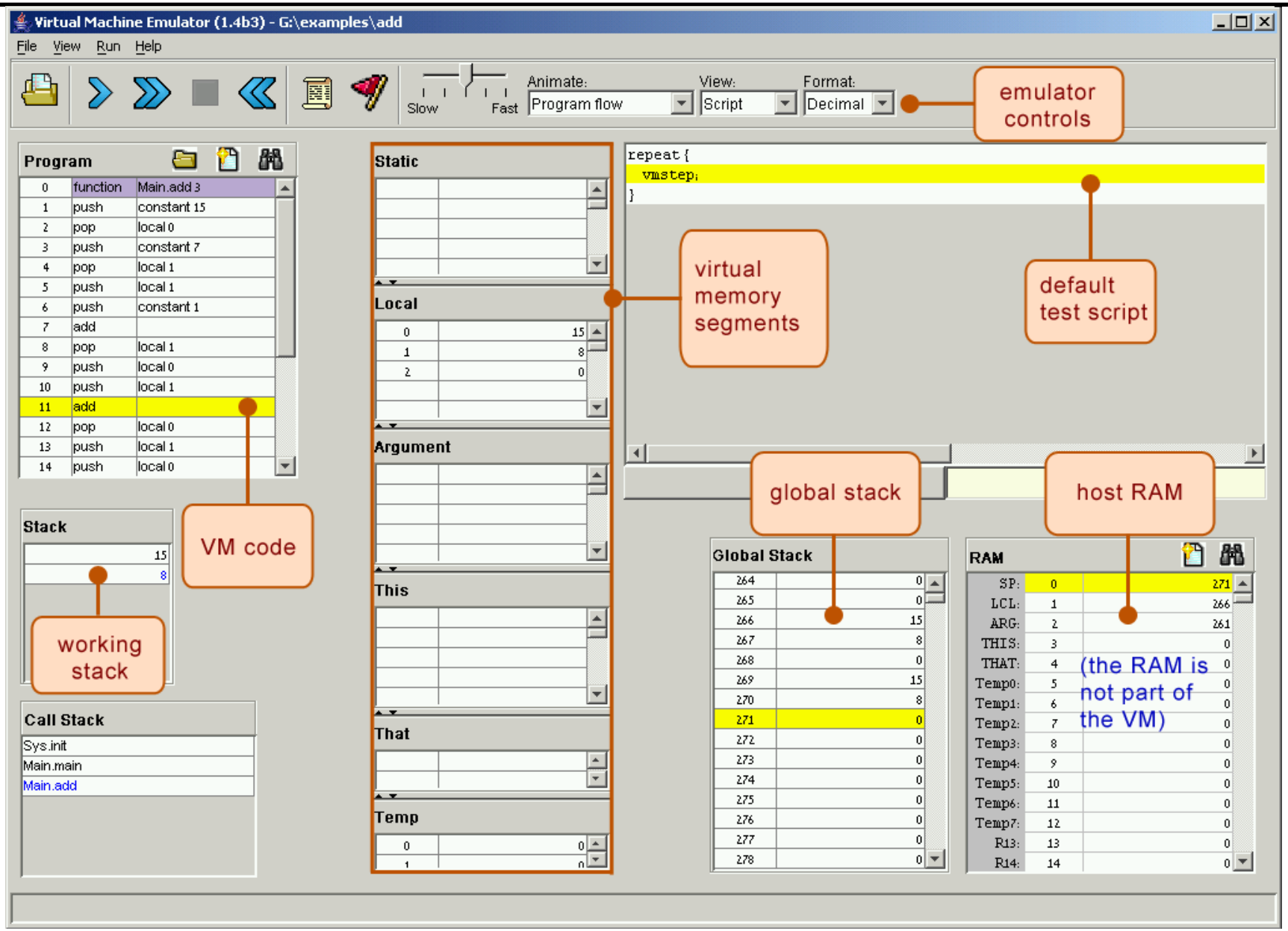
Implementazione della memoria



Dal punto di vista "fisico" la memoria verrà organizzata nel seguente modo:

- Gli indirizzi da 0 a 15 non saranno parte di specifici segmenti, ma saranno usati a livello di implementazione della virtual machine
- In particolare, noi useremo SP (stack pointer), LCL (local segment pointer), ARG (argument segment pointer) (gli altri indirizzi fino a 12 sono usati in una implementazione più completa nel progetto nand2tetris, da 13 a 15 lasciati liberi)
- Gli indirizzi da 16 a 255 saranno usati per il segmento "static"
  - L'accesso a questo segmento si mapperà sulla gestione dei simboli a livello assembler
  - Gli indirizzi da 256 a 2047 saranno usati per lo stack (operazioni push e pop) e per i segmenti "argument" e "local" creati al momento dell'invocazione di funzioni (SP inizializzato a 256)
  - Heap non lo useremo (usato nell'implementazione completa nand2tetris)

VMEmulator (esecutore di VM code presente nella suite nand2tetris)



Note sull'uso del VMEmulator

- Il nome delle funzioni deve essere del tipo nomefile.nomefun
  - Dal nomefile rimuovere l'estensione
- Il VMEmulator inizializza SP a 256
- Il VMEmulator non inizializza i segmenti local e argument
  - Questi vengono inizializzati se fate una chiamata di funzione

Comandi per il controllo del flusso

- `label` *c*
- `goto` *c*
- `if-goto` *c* // pop l'elemento in cima allo stack;
  // se è true (non zero) salta

Implementazione:

Semplice, usare il meccanismo delle dichiarazioni di label e dei jump messi a disposizione dal linguaggio assembly

Esempio:

```
function mult 2
  push    constant 0
  pop     local 0
  push    argument 1
  pop     local 1
label     loop
  push    local 1
  push    constant 0
  eq
  if-goto end
  push    local 0
  push    argument 0
  add
  pop     local 0
  push    local 1
  push    constant 1
  sub
  pop     local 1
  goto    loop
label     end
  push    local 0
  return
```

Subroutine

- Le subroutine sono un'astrazione fondamentale nei linguaggi di programmazione
- Idea di base: il linguaggio può essere esteso a piacimento con comandi definiti dall'utente (subroutine/funzioni/metodi/...)
- Importante: i comandi primitivi e quelli definiti dall'utente devono essere "simili", per cui le estensioni risultano trasparenti

Subroutine nel linguaggio VM

```
// x+2
push x
push 2
add
...
```

```
// x^3
push x
push 3
call power
...
```

```
// (x^3+2)^y
push x
push 3
call power
push 2
add
push y
call power
...
```

```
// Power function
// result = first arg
// raised to the power
// of the second arg.
function power
// code omitted
push result
return
```

Call-and-return convention

- Il chiamante fa push degli argomenti, fa la "call" del chiamato, e poi aspetta il "return"
- Prima di terminare, il chiamato deve fare push del valore di ritorno
- Al momento della "return" le risorse usate dal chiamato vengono rilasciate ed il chiamante è ripristinato nel medesimo stato in cui era al momento della "call"
- **Effetto netto**: il chiamato consuma gli argomenti e lascia il valore di ritorno in cima allo stack esattamente come succede nelle operazioni di base (add, or, not, ..)

Dettagli implementativi

- Attenzione al rilascio delle risorse del chiamato ed al ripristino dello stato del chiamante.. questa è la parte più critica dell'implementazione!

Comandi relativi alle subroutine

- **`function`** *g nVars*

  (inizio della funzione di nome *g*, che usa *nVars* variabili locali)

- **`call`** *g nArgs*

  (invoca la funzione di nome *g*;
  nArgs argomenti sono già stati inseriti in cima allo stack)

- **`return`**

  (termina l'esecuzione della funzione corrente e
  restituisce il controllo al chiamante)

Q: Perchè questa particolare sintassi?

A: Perchè semplifica l'implementazione

Il protocollo di call-and-return di funzioni

- `function` *g nVars*
- `call` *g nArgs*
- `return`

La vista del chiamante:

- Prima della chiamata devo inserire gli argomenti in cima allo stack
- Successivamente invoco la funzione richiesta con il comando **`call`**
- Dopo che la funzione chiamata ritorna:
  - Gli argomenti che ho inserito nello stack prima della chiamata sono scomparsi; al loro posto è presente in cima allo stack il valore di ritorno (nel nostro linguaggio per VM tale valore di ritorno esiste sempre)
  - Tutti i miei segmenti di memoria (**argument**, **local**, …) sono esattamente gli stessi di prima della chiamata

La vista del chiamato:

- Quando inizio ad eseguire il mio segmento **argument** contiene i parametri attuali passati dal chiamante
- Il mio segmento **local** è presente e contiene tutti valori uguali a zero
- Lo stack che vedo è vuoto
- Prima di eseguire return devo essere sicuro che l'effetto della mia intera esecuzione sia stato quello di inserire nello stack il solo valore di ritorno

Dal punto di vista implementativo

Quando la funzione *f* chiama la funzione *g*, la implementazione della VM deve:

- Salvare il "return address" (indirizzo in ROM della istruzione successiva alla call)
- Salvare i puntatori ai segmenti local e argument di *f*
- Impostare i puntatori local e argument di *g*
- Passare il controllo all'implementazione di *g*

- `function` *g nVars*
- `call` *g nArgs*
- `return`

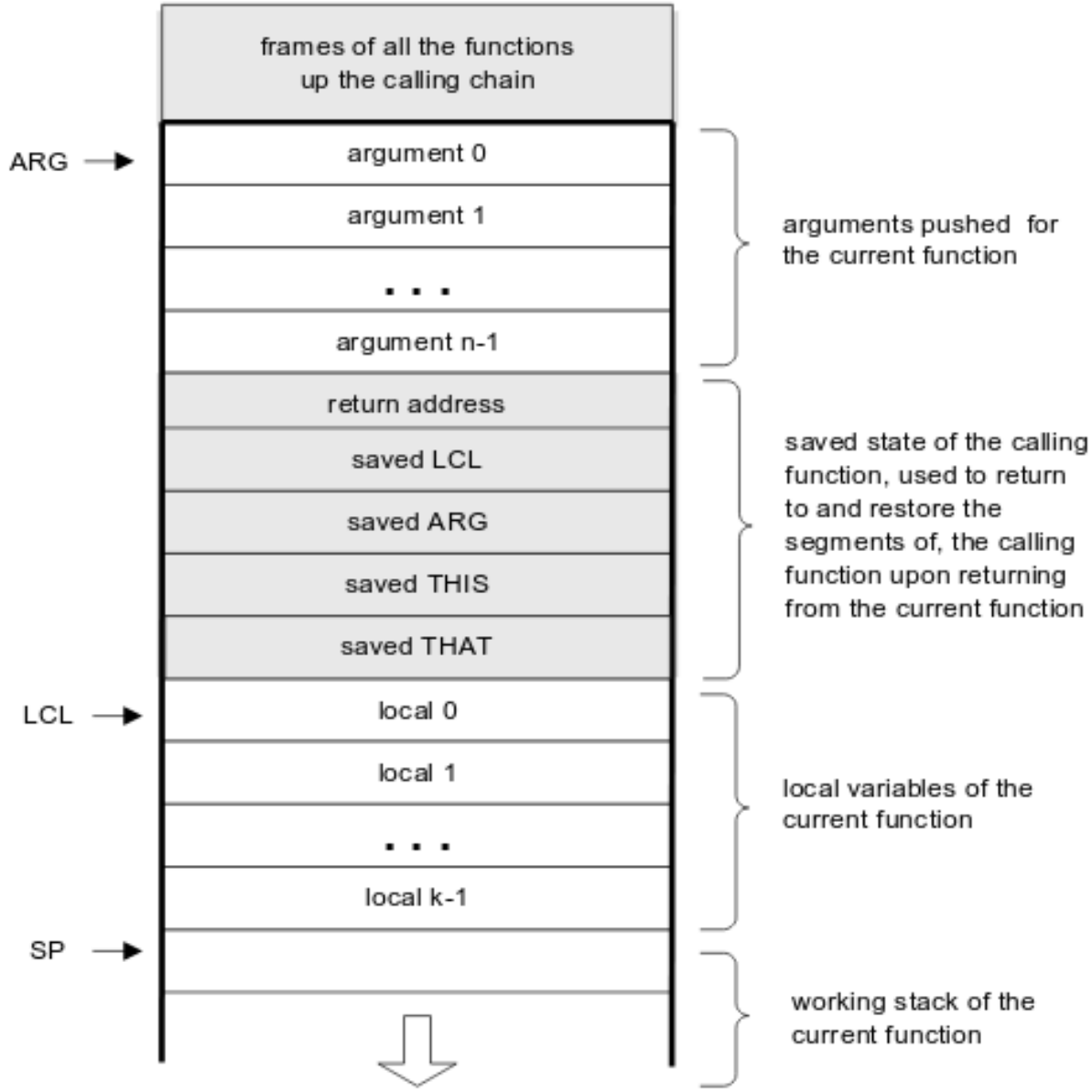
Quando la funzione *g* inizia ad eseguire, l'implementazione della VM deve:

- Predisporre lo spazio per il segmento local (inizializzando tutte le sue celle a 0)

Quando *g* termina ed il controllo deve ritornare a *f*, l'implementazione della VM deve:

- Eliminare i segmenti local e argument lasciando sullo stack solamente il valore di ritorno
- Ripristinare i segmenti local e argument di *f*
- Restituire il controllo a *f* saltando al "return address"

Tutto implementato sullo stack



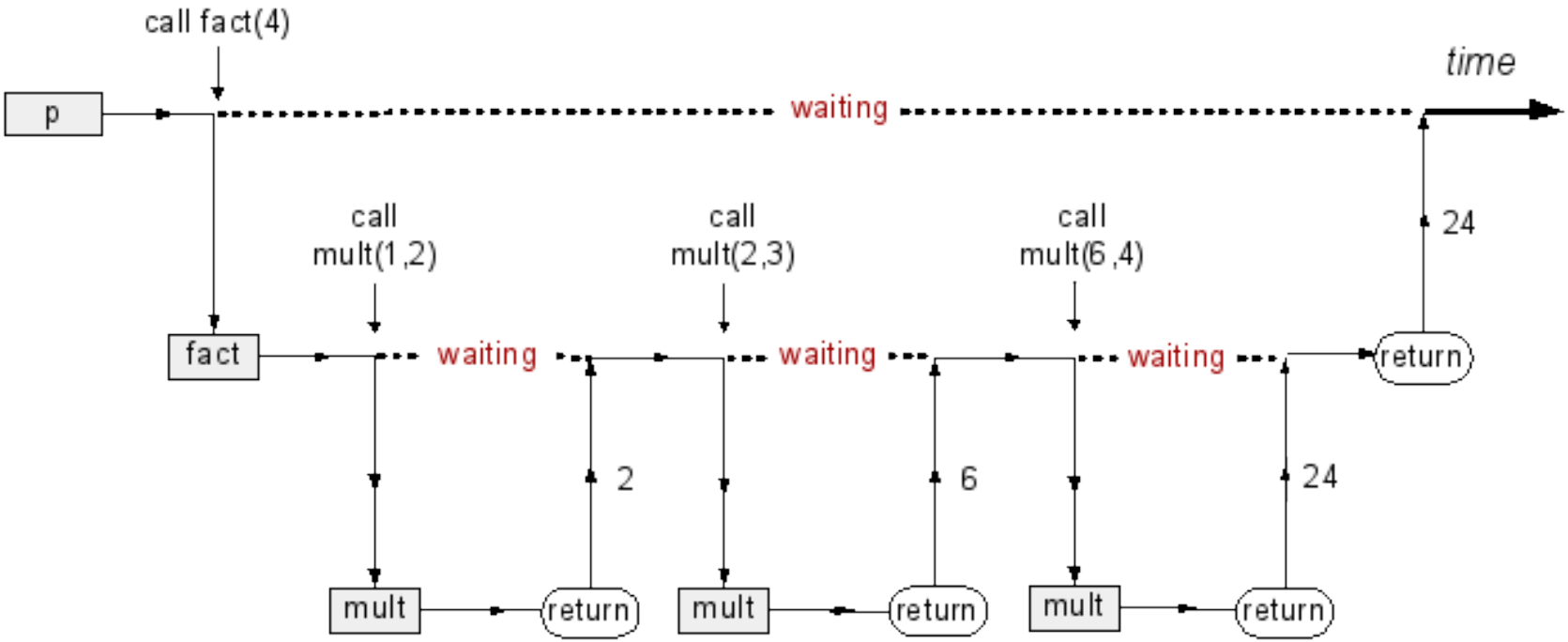
- Usualmente c'è una catena di funzioni in attesa di essere riattivate, ed una sola funzione in esecuzione
- Aree grigie: parti irrilevanti per la funzione in esecuzione
- La funzione corrente si interessa solo della parte più recente dello stack (detto *working stack*)
- Il resto dello stack contiene lo stato "congelato" delle funzioni in attesa di essere riattivate

Esempio di catena di chiamate

```
function p(...) {
...
    ... fact(4) ...
}
```

```
function fact(n) {
    vars result,j;
    result=1; j=2;
    while j<=n {
       result=mult(result,j);
       j=j+1;
    }
     return result;
}
```

```
function mult(x,y) {
    vars sum,j;
    sum=0; j=y;
    while j>0 {
       sum=sum+x;
       j=j+1;
    }
    return sum;
}
```

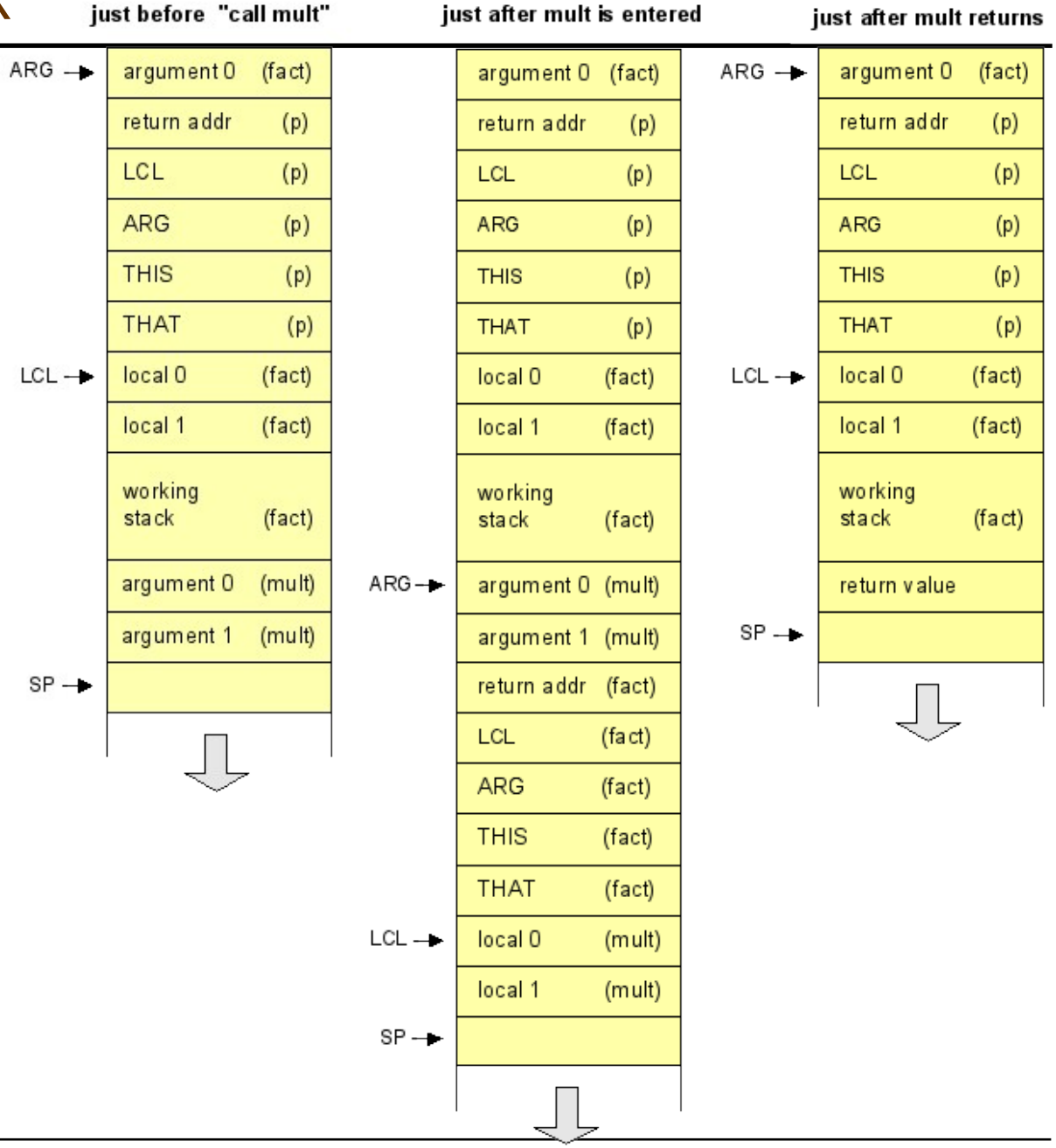


Evoluzione dello stack

```
function p(...) {
...
    ... fact(4) ...
}
```

```
function fact(n) {
    vars result,j;
    result=1; j=1;
    while j<=n {
       result=mult(result,j);
       j=j+1;
    }
     return result;
}
```

```
function mult(x,y) {
    vars sum,j;
    sum=0; j=y;
    while j>0 {
       sum=sum+x;
       j=j+1;
    }
    return sum;
}
```



Implementazione del comando `call f n`

`call f n`

(calling a function `f` after `n` arguments have been pushed onto the stack)

```
push return-address // (Using the label declared below)
push LCL            // Save LCL of the calling function
push ARG            // Save ARG of the calling function
push THIS           // Save THIS of the calling function
push THAT           // Save THAT of the calling function
ARG = SP-n-5        // Reposition ARG (n = number of args)
LCL = SP            // Reposition LCL
goto f              // Transfer control
(return-address)    // Declare a label for the return-address
```

- Questa è la versione completa del progetto nand2tetris: a noi bastano i segmenti local e argument (quindi non sarebbe per noi necessario considerare THIS e THAT)

Implementazione del comando `function f k`

`function f k`

(declaring a function `f` that has `k` local variables)

```
(f)                     // Declare a label for the function entry
   repeat k times:      // k = number of local variables
   PUSH 0               // Initialize all of them to 0
```

Implementazione del comando `call f n`

`return`

(from a function)

```
FRAME=LCL          // FRAME is a temporary variable
RET=*(FRAME-5)     // Put the return-address in a temp. variable
*ARG=pop()         // Reposition the return value for the caller
SP=ARG+1           // Restore SP of the caller
THAT=*(FRAME-1)    // Restore THAT of the caller
THIS=*(FRAME-2)    // Restore THIS of the caller
ARG=*(FRAME-3)     // Restore ARG  of the caller
LCL=*(FRAME-4)     // Restore LCL of the caller
goto RET           // Goto return-address (in the caller's code)
```

- Questa è la versione completa del progetto nand2tetris: a noi bastano i segmenti local e argument (quindi non sarebbe per noi necessario considerare THIS e THAT)

Un ultimo dettaglio

- La traduzione da un programma scritto nel linguaggio della virtual machine al relativo programma assembly deve prevedere di inizializzare l'architettura sottostante per lavorare correttamente
- Nello specifico, si rende necessario inizializzare in modo appropriato lo stack pointer
  - Questo si fa settando a 256 il contenuto della cella di memoria di indirizzo SP come prima operazione assembly
    (che, visto l'assmblatore che abbiamo già implementato, coincide con il mettere 256 in RAM[0])

Esercizio

- Scrivere un programma per la VM che inserisce nello stack i numeri primi minori di 20

Esercizio più divertente

- Scrivere un programma per la VM che stampa un puntino sullo schermo e consente di spostarlo usando le frecce
  - Versione semplificata: sposta un segmento orizzontale lungo 16 pixel (che in orizzontale si sposta di 16 pixel alla volta)
  - Per risolvere questo esercizio occorrono i segmenti pointer e that per accedere a schermo e tastiera
  - that 0 consente di accedere alla locazione di memoria puntata da pointer 1, per cui metto in pointer 1 il puntatore alla locazione alla quale voglio accedere