

Rechnersysteme und -netze

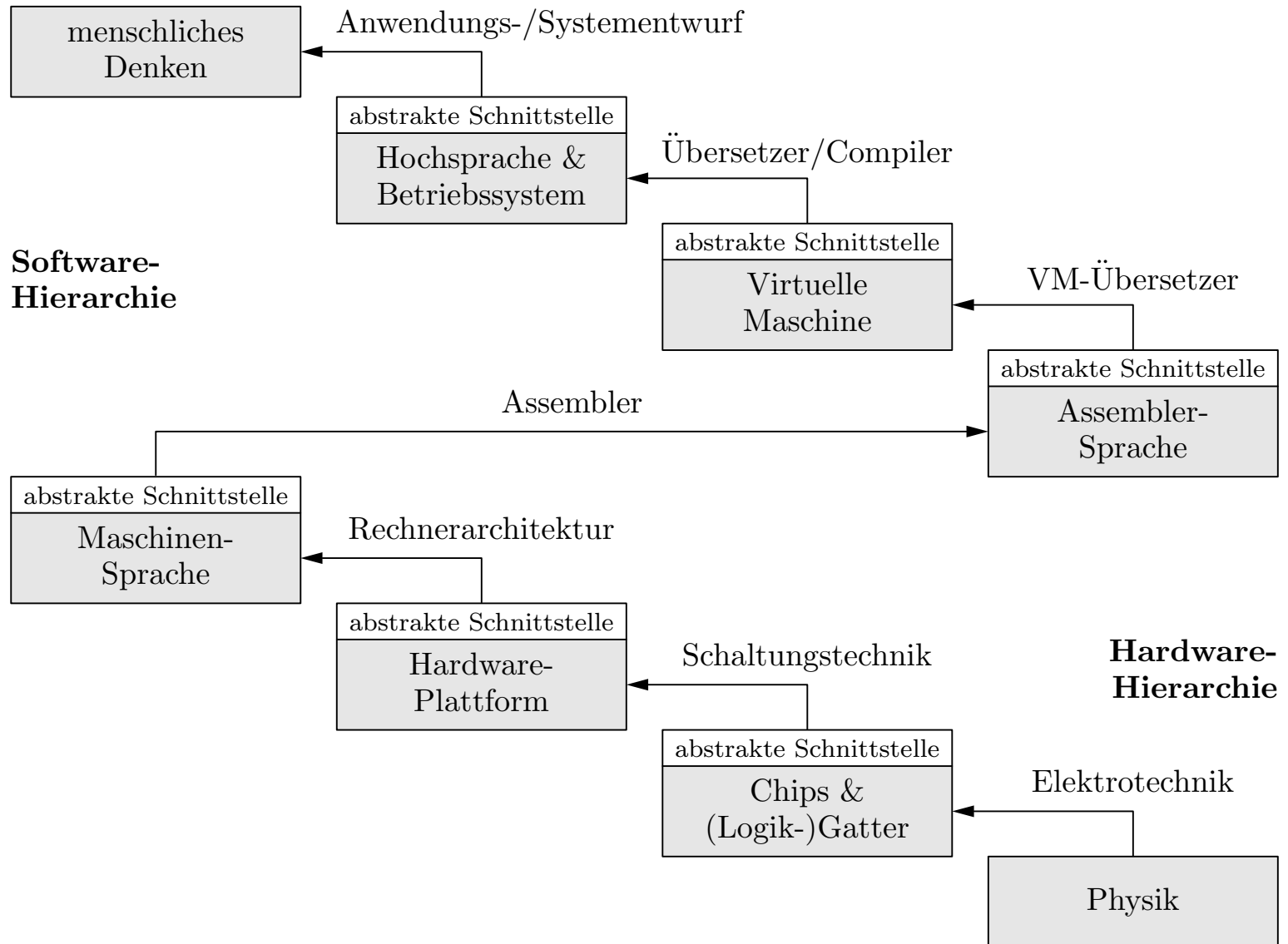
Kapitel 7

Virtuelle Maschine

Bastian Goldlücke

Universität Konstanz
WS 2020/21

Rechnersysteme: Plan der Vorlesung



Erinnerung: Rechnerarchitektur

- **Speicherprogrammierung**

- Festverdrahtete „Prozessoren“
- Konzept der Speicherprogrammierung (stored program concept)
- Befehlsabruf, -dekodierung und -ausführung (fetch-decode-execute cycle)
- Rechnerarchitekturen (Harvard und von Neumann)

- **Die Hack-Plattform**

- Befehls- und Datenspeicher (ROM32K und RAM16K)
- Bildschirm und Bildschirmspeicher (screen)
- Tastatur (keyboard)
- Hauptspeicherorganisation (memory)
- Prozessor (central processing unit, CPU)
- Gesamtsystem (computer on a chip)

Erinnerung: Maschinensprache und Assembler

- **Die Hack-Maschinensprache**
 - A-Anweisungen (address instructions)
 - C-Anweisungen (compute instructions)
- **Assembler und Assemblersprache**
 - Physikalische und symbolische Programmierung
 - Maschinensprache und Assemblerprache
 - Opcodes, mnemonische Symbole (Mnemonics)
 - Die Hack-Assemblersprache
 - Typische Programmstrukturen in der Hack-Assemblersprache
 - Symbole und Symbolverwaltung
 - Programmübersetzung und Assemblerimplementierung
 - Beispiele für Assemblerprogramme: Linux und Windows

1 Höhere Programmiersprachen und Übersetzung

- 1.1 Direkte und zweistufige Übersetzung
- 1.2 Zwischensprache und virtuelle Maschine
- 1.3 Systembasierte und prozeßbasierte virtuelle Maschinen
- 1.4 Übersetzungspfad

2 Virtuelle Maschine des Hack-Systems

- 2.1 Stapel(-speicher) und ihre Operationen
- 2.2 Stapelarithmetik (arithmetische und logische Operationen)
- 2.3 Speicherzugriff, Speicheraufteilung, Speichersegmente
- 2.4 Programmablauf (bedingte Anweisungen und Schleifen)
- 2.5 Objekt- und Arraybehandlung
- 2.6 Funktionsaufrufe, globaler Stapel zur Steuerung
- 2.7 Befehlssatz
- 2.8 Programmstart

Overview

1 Höhere Programmiersprachen und Übersetzung

- 1.1 Direkte und zweistufige Übersetzung
- 1.2 Zwischensprache und virtuelle Maschine
- 1.3 Systembasierte und prozeßbasierte virtuelle Maschinen
- 1.4 Übersetzungspfad

2 Virtuelle Maschine des Hack-Systems

- 2.1 Stapel(-speicher) und ihre Operationen
- 2.2 Stapelarithmetik (arithmetische und logische Operationen)
- 2.3 Speicherzugriff, Speicheraufteilung, Speichersegmente
- 2.4 Programmablauf (bedingte Anweisungen und Schleifen)
- 2.5 Objekt- und Arraybehandlung
- 2.6 Funktionsaufrufe, globaler Stapel zur Steuerung
- 2.7 Befehlssatz
- 2.8 Programmstart

Hochsprachen und Übersetzung



Höhere Programmiersprachen (Hochsprachen)

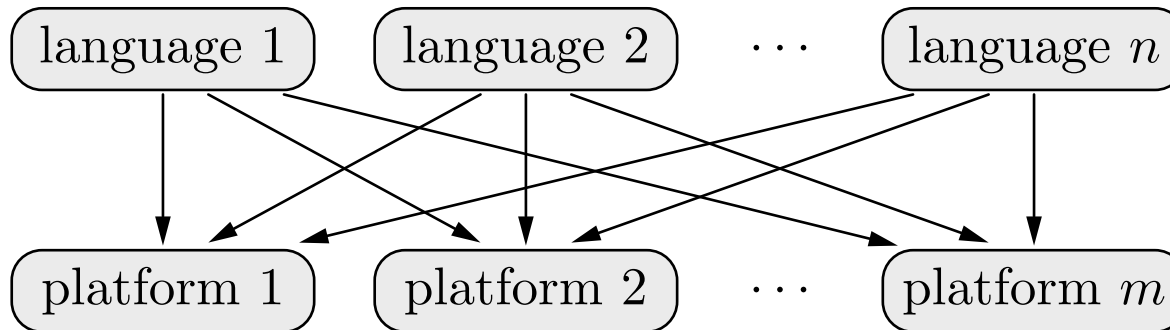
- abstrahieren von den konkreten Eigenschaften des Rechners;
- sind leichter zu verstehen als Sprachen tieferer Ebenen.

Aber: Hochsprachen

- können von einem Rechner nicht direkt ausgeführt werden;
- müssen daher in Sprachen tieferer Ebenen (und schließlich in Assembler-/Maschinensprache) übersetzt werden.

Hochsprachen und Übersetzung

- Problem: Es gibt viele verschiedene (Hoch-)Sprachen und viele verschiedene Hardware-Plattformen.
- Eine **direkte Übersetzung** aus einer Hochsprache in die Maschinensprache einer Hardware-Plattform erfordert sehr viele Übersetzer (compiler).

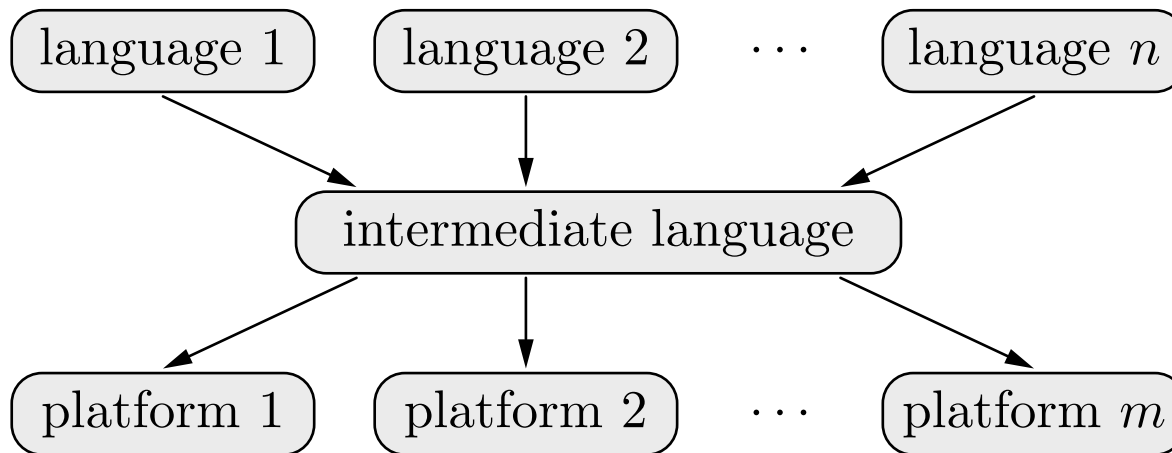


- Für jedes Paar aus einer (Hoch-)Sprache und einer Hardware-Plattform muß ein eigener Übersetzer (compiler) geschrieben werden.
- Nachteil: n Sprachen und m Plattformen erfordern $n \times m$ Übersetzer.

Außerdem: Redundanter Programmieraufwand, da das Einlesen und Analysieren des Hochsprachenquelltextes u.U. mehrfach implementiert wird.

Hochsprachen und Übersetzung

- Problem: Es gibt viele verschiedene (Hoch-)Sprachen und viele verschiedene Hardware-Plattformen.
- Daher: Aufteilung des Übersetzungsvorgangs (**zweistufige Übersetzung**).



- Die (Hoch-)Sprachen werden zunächst in eine **Zwischensprache** übersetzt, die unabhängig von der Hardware-Plattform ist,
- Vorteil: n Sprachen und m Plattformen erfordern nur $n + m$ Übersetzer.

Außerdem: Die Zwischensprache hat eine einfachere Struktur als die Hochsprachen und kann daher leichter in Maschinensprache übersetzt werden.

Hochsprachen und Übersetzung

- Eine **zweistufige Übersetzung** entkoppelt Hochsprachen und Plattformen:
 - Erste Stufe hängt nur von der Hochsprache ab (compilation).
 - Zweite Stufe hängt nur von der Zielmaschine ab (translation/interpretation).
- Die **Zwischensprache** kann als Assembler-/Maschinensprache einer virtuellen oder Pseudo-Hardware-Plattform gesehen werden: Sie bezieht sich auf eine **virtuelle Maschine** (einen virtuellen Rechner).
- Die grundsätzliche Idee virtueller Maschinen als softwaretechnische Kapselung eines Rechnersystems innerhalb eines anderen wurde in den 1970er Jahren entwickelt (z.B. Betriebssystem CP/CMS, Programmiersprache UCSD Pascal).
- Die abstrahierende Schicht zwischen der virtuellen Maschine und dem realen Rechner auf dem die virtuelle Maschine ausgeführt wird, wird **Hypervisor** oder **Virtual Machine Monitor** genannt.
- Man unterscheidet **systembasierte virtuelle Maschinen** und **prozeßbasierte virtuelle Maschinen**.

Typen Virtueller Maschinen

- **Systembasierte virtuelle Maschinen**

- Motivation: mehrere Betriebssysteme gleichzeitig auf einem Rechner.
- So vollständige Nachbildung eines realen Rechners, daß Betriebssysteme, die für diesen realen Rechner entworfen wurden, ausgeführt werden können (z.B. VMwares vSphere, Oracles VirtualBox o.ä.)
- “Eine virtuelle Maschine ist ein effizientes, identisches und isoliertes Duplikat eines echten Prozessors.” [Goldberg und Popek 1972]

- **Prozeßbasierte virtuelle Maschinen**

- Motivation: Programme, die für eine Rechnerarchitektur entwickelt wurden, werden ohne Änderungen auf einer anderen Rechnerarchitektur ausgeführt.
- Einzelne Programme werden abstrahiert von der Ausführungsumgebung einer Rechnerarchitektur ausgeführt, indem eine darauf aufbauende Laufzeitumgebung bereitgestellt wird. [Nelson 1979]
- Beispiele: Java Virtual Machine (JVM, Bytecode)
Common Language Runtime (CLR, .NET Framework)

Vor- und Nachteile Virtueller Maschinen

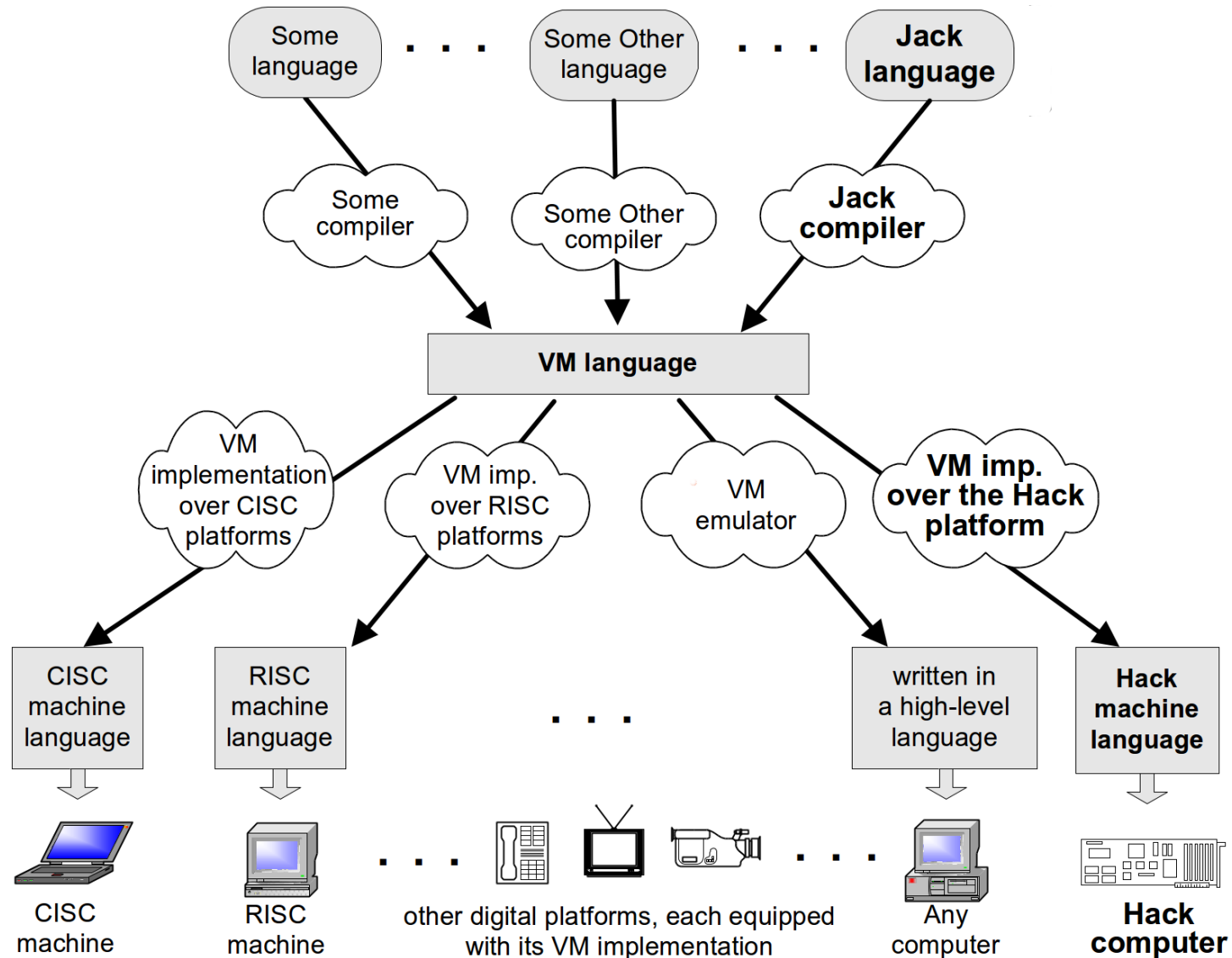
Vorteile:

- Plattform-Unabhängigkeit: Programme für eine virtuelle Maschine laufen auf allen physischen Maschinen, für die die virtuelle Maschine implementiert ist.
- Dynamische Optimierung auf spezielles Zielsystem ist möglich/einfacher.
- Implementierung von Übersetzern wird einfacher.

Nachteile:

- Effizienzverlust gegenüber direkter Übersetzung für das Zielsystem.
- Zusätzliche Indirektionen bei Interpretation, dadurch langsamere Ausführung.
- JIT-Übersetzer (just in time compiler) lösen die meisten Indirektionen auf, benötigen aber Zeit für die Übersetzung.
- Auch kleine Programme benötigen die vollständige virtuelle Maschine.
- Entwickler haben weniger Kontrolle über den Zielcode.

Vorteil virtueller Maschinen



Übersetzungspfad

Jack-Quelltext (Beispiel)

```
class c1 {
  static int x1, x2, x3;
  method int f1 (int x) {
    var int a, b;
    ...
  }
  method int f2 (int x, int y) {
    var int a, b, c;
    ...
  }
  method int f3 (int u) {
    var int x;
    ...
  }
}

class c2 {
  static int y1, y2;
  method int f1 (int u, int v) {
    var int a, b;
    ...
  }
  method int f2 (int x) {
    var int a1, a2;
    ...
  }
}
```

- Jede Klasse hat eine Liste statischer Variablen („globale Variablen“).
- Jede Funktion hat eine Liste von Argumenten.
- Jede Funktion hat eine Liste lokaler Variablen.
- Die Übersetzung muß den Zugriff auf diese Listen organisieren.

Jack-Quelltext (allgemein)

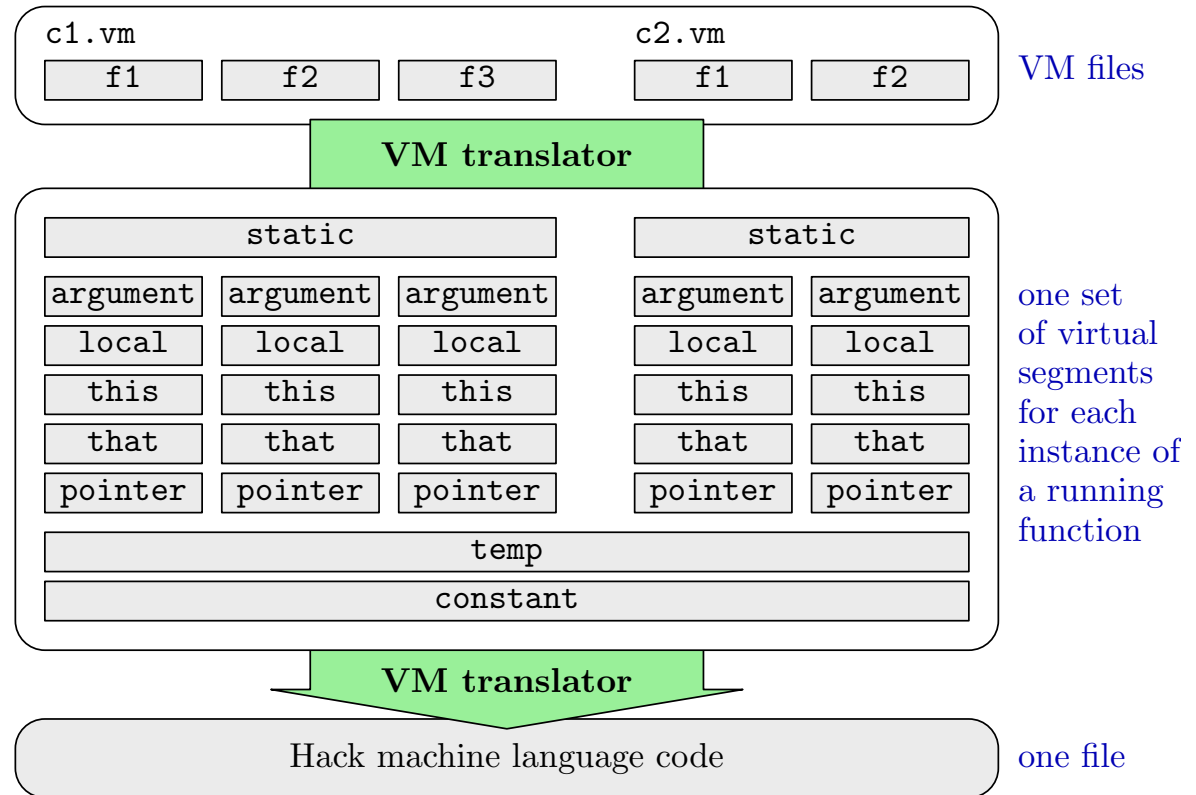
```
class c1 {
  static staticsList;
  method int f1 (argsList) {
    var localsList;
    ...
  }
  method int f2 (argsList) {
    var localsList;
    ...
  }
  method int f3 (argsList) {
    var localsList;
    ...
  }
}

class c2 {
  static staticsList;
  method int f1 (argsList) {
    var localsList;
    ...
  }
  method int f2 (argsList) {
    var localsList;
    ...
  }
}
```

Übersetzungspfad

Jack-Quelltext (allg.)

```
class c1 {  
    static staticsList;  
    method int f1 (argsList) {  
        var localsList;  
        ...  
    }  
    method int f2 (argsList) {  
        var localsList;  
        ...  
    }  
    method int f3 (argsList) {  
        var localsList;  
        ...  
    }  
}  
  
class c2 {  
    static staticsList;  
    method int f1 (argsList) {  
        var localsList;  
        ...  
    }  
    method int f2 (argsList) {  
        var localsList;  
        ...  
    }  
}
```



Durch die Übersetzung werden den verschiedenen Listen der Klassen und Funktionen Speichersegmente zugewiesen. Für die lokalen Variablen werden sie dynamisch bestimmt.

Inhalt

1 Höhere Programmiersprachen und Übersetzung

- 1.1 Direkte und zweistufige Übersetzung
- 1.2 Zwischensprache und virtuelle Maschine
- 1.3 Systembasierte und prozeßbasierte virtuelle Maschinen
- 1.4 Übersetzungspfad

2 Virtuelle Maschine des Hack-Systems

- 2.1 Stapel(-speicher) und ihre Operationen
- 2.2 Stapelarithmetik (arithmetische und logische Operationen)
- 2.3 Speicherzugriff, Speicheraufteilung, Speichersegmente
- 2.4 Programmablauf (bedingte Anweisungen und Schleifen)
- 2.5 Objekt- und Arraybehandlung
- 2.6 Funktionsaufrufe, globaler Stapel zur Steuerung
- 2.7 Befehlssatz
- 2.8 Programmstart

Eine virtuelle Maschine für das Hack-System

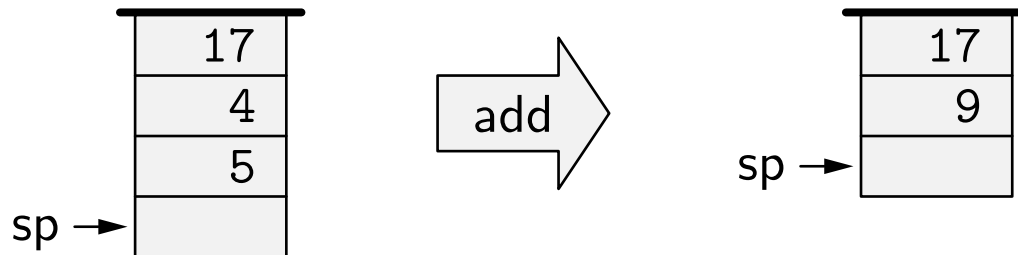
- Wir betrachten im folgenden eine virtuelle Maschine für das Hack-System, die mit einem Stapel als zentraler Datenstruktur und Funktionsaufrufen arbeitet (weitgehend analog zur virtuellen Maschine von Java).
- Ein Programm für diese virtuelle Maschine besteht aus einer Sammlung von Dateien mit der Endung `.vm`
Jede dieser Dateien enthält eine oder mehrere Funktionen (in grober Näherung analog zu Java Bytecode).
- Es wird nur ein einziger 16-Bit-Datentyp verwendet, der sowohl Ganzzahlen und Boolesche Werte als auch Zeiger darstellt.
- Wir betrachten die folgenden Elemente dieser virtuellen Maschine:
 - Arithmetisch-logische Operationen
 - Speicherzugriff
 - Programmablaufsteuerung
 - Funktionsaufrufe

Stapel(-speicher) / Keller(-speicher)

- Ein **Stapel(-speicher)** oder **Keller(-speicher)** (englisch: stack) ist eine häufig verwendete dynamische abstrakte Datenstruktur, die Daten nach dem LIFO-Prinzip speichert (last in, first out).
- Ein Stapel/Keller stellt zwei bzw. drei Operationen zur Verfügung
 - **push** (deutsch: „einkellern“)
Ein Objekt wird oben auf den Stapel gelegt.
 - **pop** (abgeleitet von pull operation/operand(s), deutsch: „auskellern“)
Das auf dem Stapel zuoberst liegende Objekt wird vom Stapel entfernt und zurückgegeben.
 - **top** oder **peek** (deutsch: „nachsehen“, optionale Operation)
Das auf dem Stapel zuoberst liegende Objekt wird zurückgegeben, aber nicht vom Stapel entfernt.
- Diese Datenstruktur und ihre zugehörigen Operationen werden von den meisten Mikroprozessoren direkt in Hardware unterstützt.

Stapelarithmetik

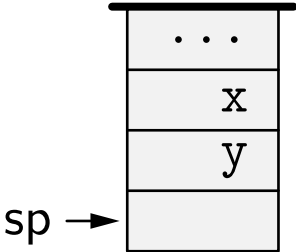
- Typische arithmetisch-logische Operation mit einem Stapel/Keller:
 - Hole die beiden obersten Werte x und y vom Stapel (pop).
 - Berechne den Wert einer Funktion $z = f(x, y)$.
 - Lege das Ergebnis z auf dem Stapel ab (push).
- Beispiel einer Addition:



- Größere Bekanntheit erhielt diese Art der Berechnung in den 1980er Jahren, weil sie von Taschenrechnern der Firma Hewlett-Packard verwendet wurde.
Sie entspricht der Schreibweise arithmetisch-logischer Operationen in **Postfixnotation** oder **umgekehrter polnischer Notation** (UPN) (englisch: reverse polish notation, RPN).

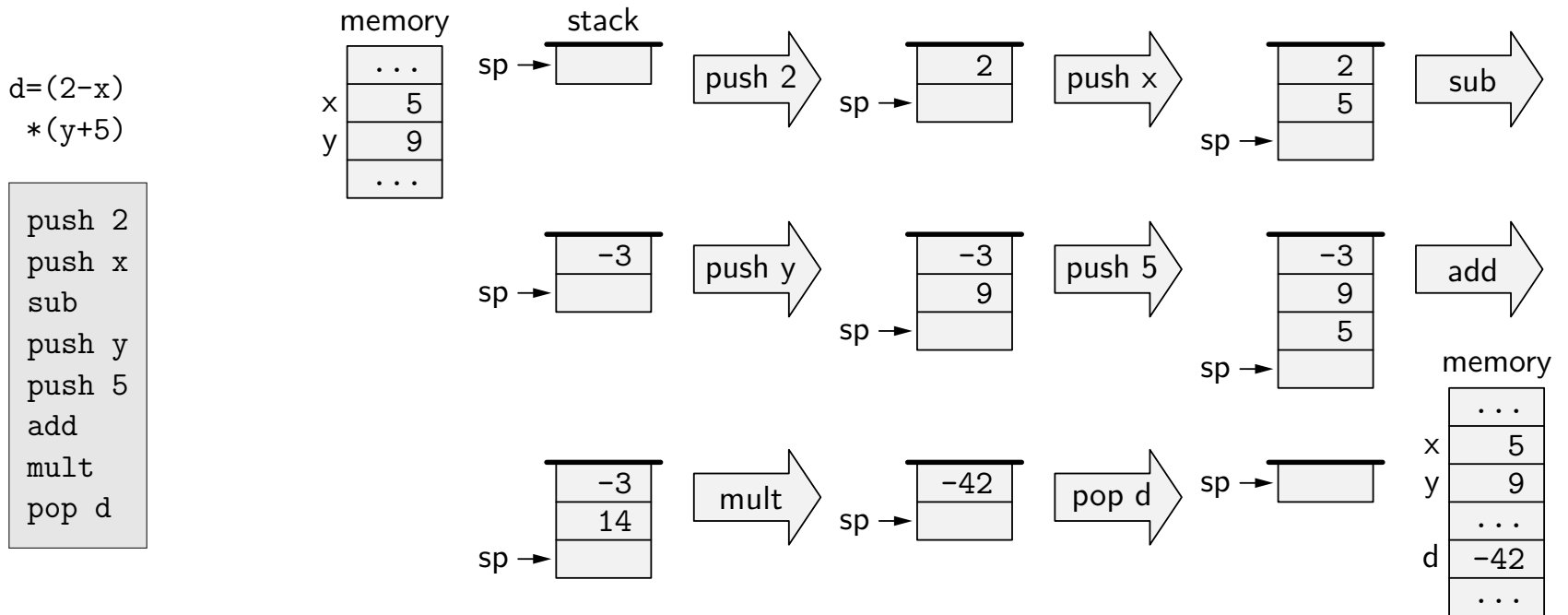
Virtuelle Maschine: Arithmetisch-logische Operationen

Anweisung	Rückgabewert	Kommentar
add	$x + y$	Ganzzahladdition (Zweierkomplement)
sub	$x - y$	Ganzzahlsubtraktion (Zweierkomplement)
neg	$-y$	arithmetische Negation (Zweierkomplement)
eq	-1 falls $x = y$, sonst 0	Test auf Gleichheit
gt	-1 falls $x > y$, sonst 0	Test auf größer
lt	-1 falls $x < y$, sonst 0	Test auf kleiner
and	$x \& y$	bitweises Und
or	$x \mid y$	bitweises Oder
not	$\sim x$	bitweise Negation



- Beachte: Die Operationen der virtuellen Maschine sind gegenüber der Assemblersprache eingeschränkt.
- Anweisungen wie `le`, `ge` etc. ließen sich zwar leicht hinzufügen, können aber auch anders erzeugt werden.
- Die Übersetzung in die Assemblersprache kann/wird noch optimieren.

Auswertung arithmetischer Ausdrücke



- Der zu berechnende Ausdruck wird aus Infix- in Postfixnotation umgeschrieben:

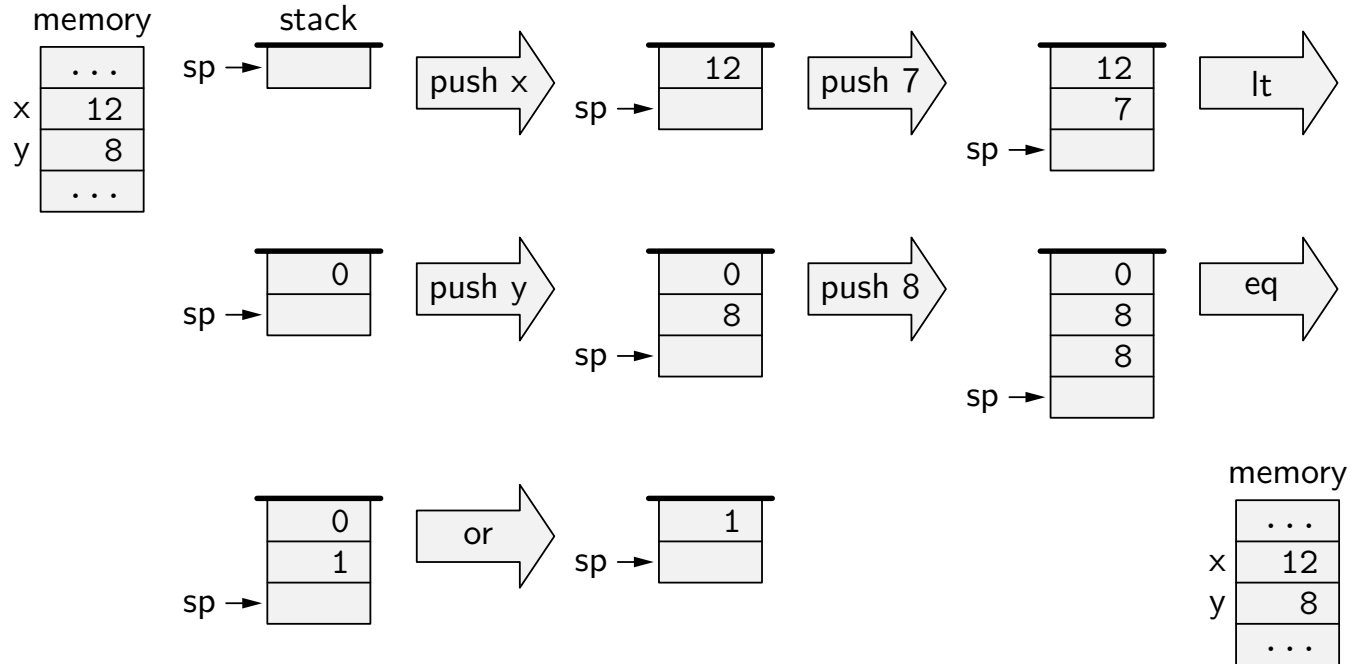
$$2 \ x \ - \ y \ 5 \ + \ * \ = \ d$$

- Durch diese Umstellung kann er leicht mit Hilfe eines Stapels berechnet werden.

Auswertung logischer Ausdrücke

if (x<7)
or (y=8)

```
push x
push 7
lt
push y
push 8
eq
or
if-goto
```



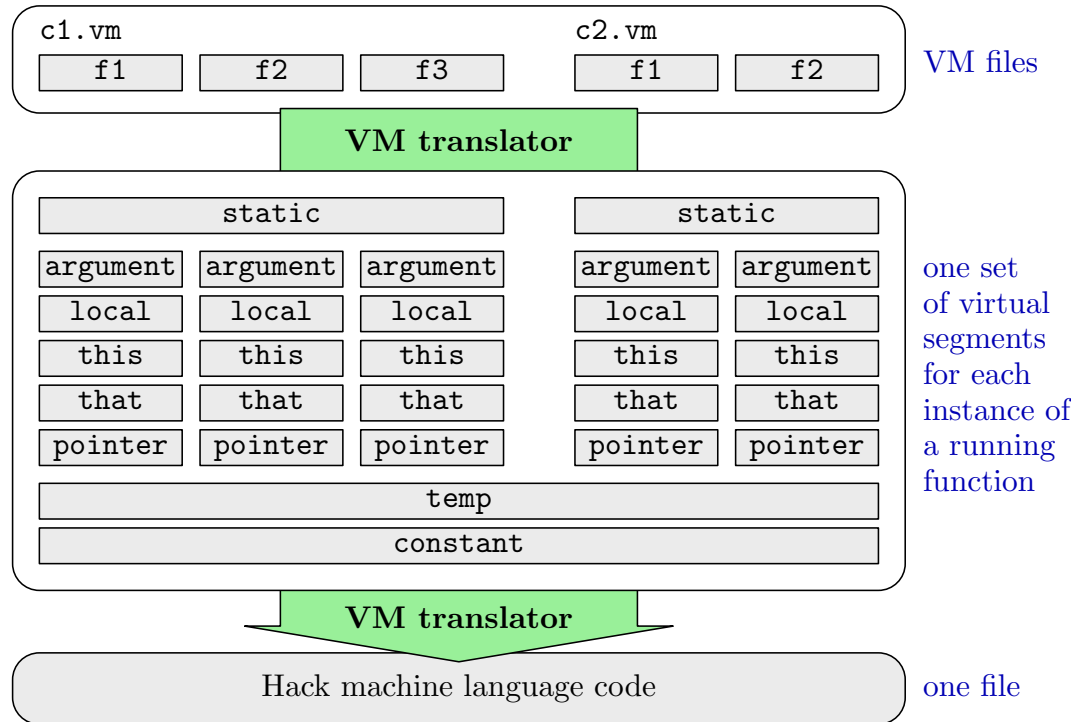
- Der zu berechnende Ausdruck wird aus Infix- in Postfixnotation umgeschrieben:

$x \ 7 < y \ 8 = \text{or if}$

- Durch diese Umstellung kann er leicht mit Hilfe eines Stapels berechnet werden.

Virtuelle Maschine: Speicherzugriff

- Bisher: Zugriff auf globalen Speicher.
- Jetzt: Die virtuelle Maschine verwaltet bis zu 8 verschiedene (virtuelle) Speichersegmente (wieder recht ähnlich zur virtuellen Maschine von Java).
- Virtuelles Speichersegment: Ein Abschnitt des Speichers, der einem bestimmten Zweck gewidmet ist.



- push segment index
Ablegen des Inhalts von segment[index] auf dem Stapel.
- pop segment index
Speichern des obersten Stapelelementes in segment[index].

Virtuelle Maschine: Speicheraufteilung

RAM

RAM[0 - 15] virtual registers

RAM[16 - 255] static variables

RAM[256 - 2047] stack

RAM[2048 - 16383] heap

RAM[16384 - 24575] memory mapped I/O

- Der Hauptspeicher des Hack-Systems wird in fünf Abschnitte eingeteilt.
- Jeder dieser Abschnitte dient speziellen Zwecken.
- Der erste und der letzte Abschnitt sind bereits aus der Betrachtung des Hack-Systems und -Assemblers bekannt.

RAM

RAM[0 - 15] virtual registers

RAM[16 - 255] static variables

RAM[256 - 2047] stack

RAM[2048 - 16383] heap

RAM[16384 - 24575] memory mapped I/O

RAM[0] stack pointer

RAM[1] points to base of function locals

RAM[2] points to base of function arguments

RAM[3] points to base of current this segment

RAM[4] points to base of current that segment

RAM[5-12] hold the content of the temp segment

RAM[13-15] can be used as general purpose registers

Virtuelle Maschine: Speicheraufteilung

RAM

RAM[0 - 15] virtual registers

RAM[16 - 255] static variables

RAM[256 - 2047] stack

RAM[2048 - 16383] heap

RAM[16384 - 24575] memory mapped I/O

Static variables are variables that are shared by all functions

Static variables of all VM functions in the program are located here

RAM

RAM[0 - 15] virtual registers

RAM[16 - 255] static variables

RAM[256 - 2047] stack

RAM[2048 - 16383] heap

RAM[16384 - 24575] memory mapped I/O

Working memory of VM operations

Data values do not jump from one segment to another - they are passed through the stack.

Central role in the VM architecture

Virtuelle Maschine: Speicheraufteilung

RAM

RAM[0 - 15] virtual registers

RAM[16 - 255] static variables

RAM[256 - 2047] stack

RAM[2048 - 16383] heap

RAM[16384 - 24575] memory mapped I/O

RAM area dedicated to storing objects and arrays.

Objects and arrays can be manipulated by VM commands.

RAM

RAM[0 - 15] virtual registers

RAM[16 - 255] static variables

RAM[256 - 2047] stack

RAM[2048 - 16383] heap

RAM[16384 - 24575] memory mapped I/O

Screen

Keyboard

Virtuelle Maschine: Abbildung der Speichersegmente

- `local`, `argument`, `this`, `that`
 - Direkte Abbildung auf festen Speicherbereich.
 - Positionen im Speicher werden in `RAM[1..4]` gehalten (`LCL`, `ARG`, `THIS`, `THAT`).
- `pointer`, `temp`
 - `pointer` wird auf `RAM[3..4]` abgebildet (`this`, `that`)
 - `temp` wird auf `RAM[5..12]` abgebildet.
- `constant`
 - Tatsächlich virtuell (kein Speicherbereich zugeordnet)
 - Die virtuelle Maschine bearbeitet einen Zugriff auf `constant i`, indem sie die Konstante `i` liefert.
- `static`
 - Verfügbar für alle Dateien mit Endung `.vm`
 - Statische Variablen werden ab `RAM[16]` zugeordnet.

Virtuelle Maschine: Speichersegmente

Segment	Verwendungszweck	Kommentare
argument	Speichert die Argumente einer Funktion.	Wird von der VM dynamisch angelegt, wenn die Funktion aufgerufen wird.
local	Speichert die lokalen Variablen einer Funktion.	Wird von der VM dynamisch angelegt, wenn die Funktion aufgerufen wird.
static	Speichert statische Variablen, die von allen Funktionen einer VM-Datei geteilt werden.	Wird von der VM für jede VM-Datei angelegt; kann von allen Funktionen in der VM-Datei benutzt werden.
constant	Pseudo-Segment, das alle Konstanten 0...32767 enthält.	Wird durch die VM emuliert; kann von allen Funktionen des Programms gesehen werden.
this, that	Mehrzwecksegmente; können auf verschiedene Bereiche des Heaps verweisen.	Jede VM-Funktion kann diese Segmente verwenden, um ausgewählte Bereiche des Heaps zu verändern.
pointer	Ein Zwei-Zellen-Segment, das die Basisadressen der this und that Segmente enthält.	Jede VM-Funktion kann pointer 0 (oder 1) auf eine Adresse setzen; dies bewirkt, daß das this (oder that) Segment auf den Speicherbereich ausgerichtet wird, der an dieser Adresse anfängt.
temp	Festes Acht-Zellen-Segment, das temporäre Variablen zur allgemeinen Verwendung enthält.	Kann von jeder VM-Funktion für beliebige Zwecke verwendet werden; wird von allen Funktionen des Programms geteilt.

Virtuelle Maschine: Programmablaufsteuerung

- `label c`
Definiert eine Marke im Programmtext, z.B. als Sprungziel.
- `goto c`
Springt zu einer Marke im Programmtext (unbedingter Sprung).
- `if-goto c`
Springt zu einer Marke im Programmtext, wenn das oberste Stapелеlement verschieden von 0 ist. (Dieses Element wird vom Stapel entfernt.)

Implementierung:

(durch Übersetzen in Assembler)

Einfach, da Markerdefinitionen und Sprünge direkt durch Assembleranweisungen ausgedrückt werden können.

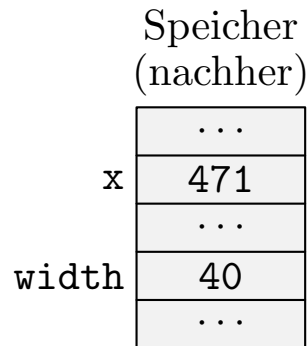
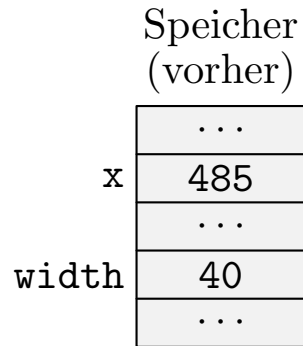
Beispiel:

```
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
```

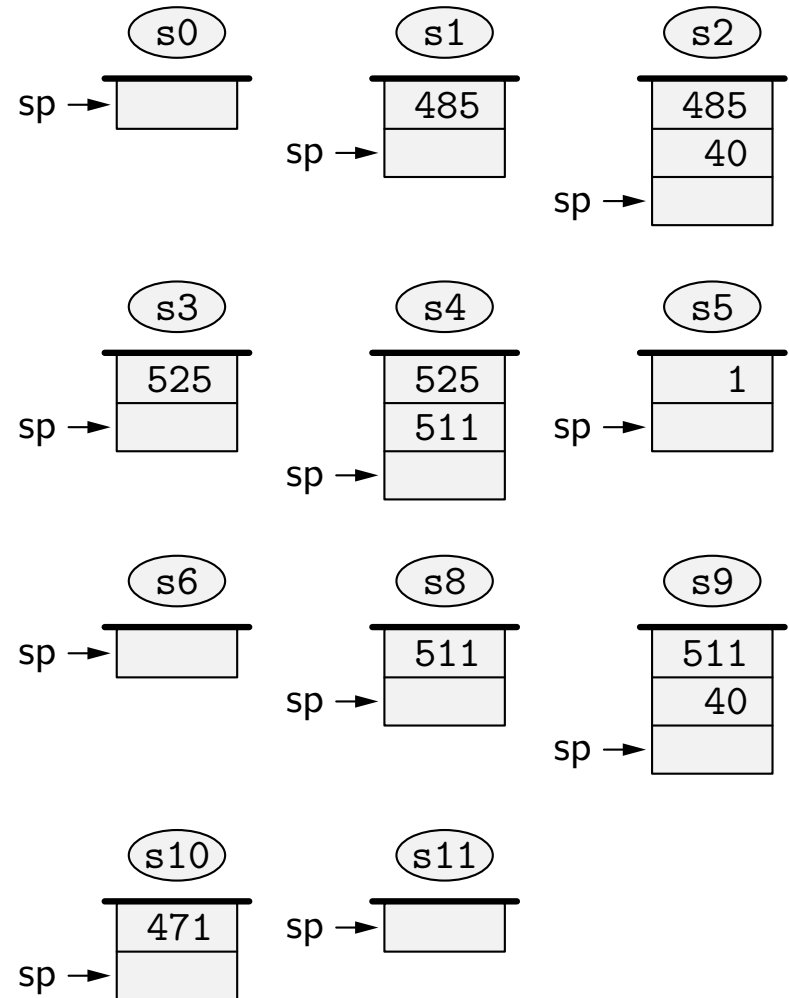
Virtuelle Maschine: Beispiel

```
// Jack source code
if ((x + width) > 511) {
  let x = 511 - width;
}
```

```
// VM code
push x           // s1
push width       // s2
add              // s3
push 511         // s4
gt               // s5
if-goto L1       // s6
goto L2         // s7
label L1
push 511         // s8
push width       // s9
sub              // s10
pop x            // s11
label L2
...
```



Stapel (stack; sp: stack pointer)



Virtuelle Maschine: Beispiel

Jack-Quelltext

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

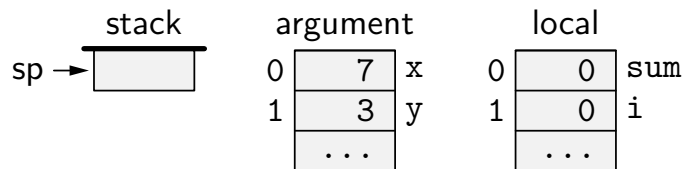
VM-Code (Näherung)

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

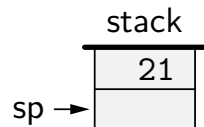
VM-Code

```
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
```

at start of mult(7,3)

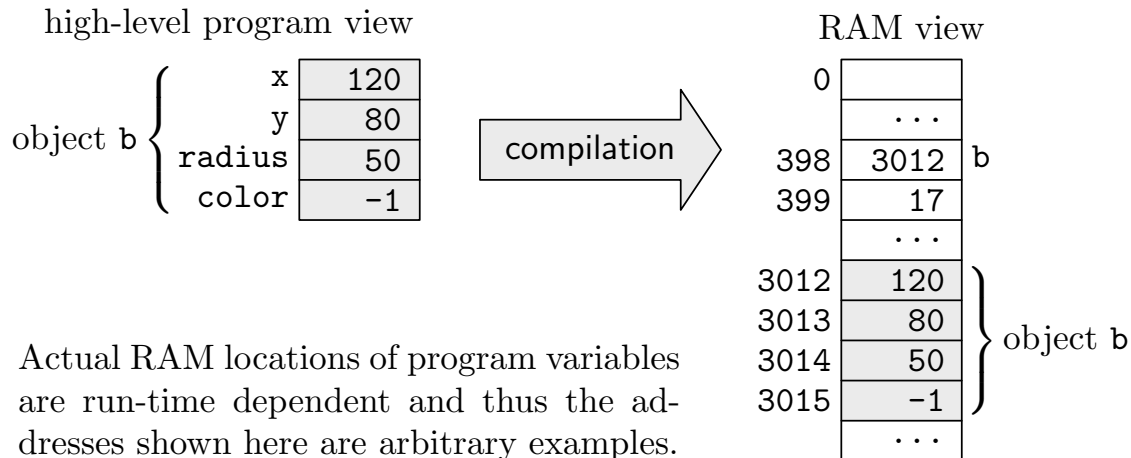


after mult(7,3) returns



Virtuelle Maschine: Objektbehandlung

Hintergrund: Wir haben ein Objekt `b` vom Typ `Ball`, das Felder `x`, `y`, `radius` und `color` besitzt.

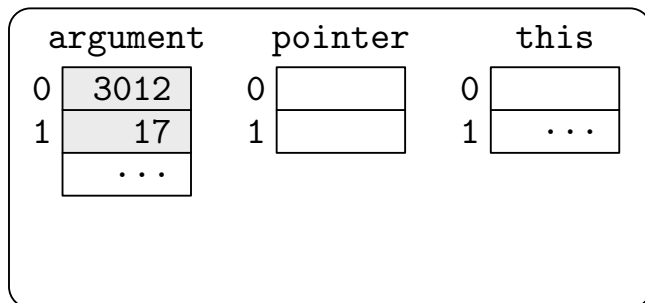


```
/* Assume that b and r were
   passed to the function as
   its first two arguments.
   The following code
   implements the operation
   b.radius = r. */

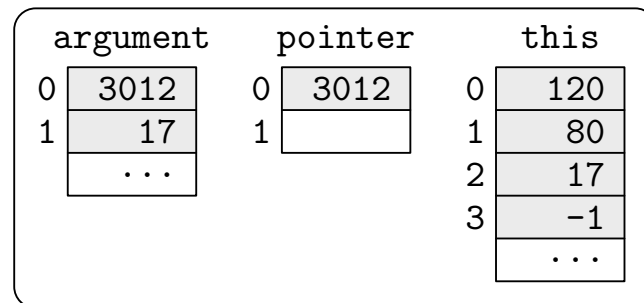
// get b's base address:
push    argument 0
// point the this segment to b:
pop     pointer 0
// get r's value
push    argument 1
// set b's third field to r:
pop     this 2
```

Actual RAM locations of program variables are run-time dependent and thus the addresses shown here are arbitrary examples.

Virtual memory segments just before the operation `b.radius = 17`:



Virtual memory segments just after the operation `b.radius = 17`:



this 0 is now aligned with RAM[3012]

Virtuelle Maschine: Arraybehandlung 1

Hintergrund: Wir haben ein Array a vom Typ int und wollen a[2] auf den Wert 27 setzen.

high-level program view

array a	0	7
	1	53
	2	121
	3	8

	9	19

compilation

RAM view

0		
	...	
412	4315	a
	...	
4315	7	array a
4316	53	
4317	121	
4318	8	
	...	
4324	19	
	...	

```
/* Assume that a is the
   first local variable
   declared in the program.
   The following code
   implements the operation
   a[2] = 27 or *(a+2) = 27 */

// get a's base address:
push    local 0
// point the that segment to a:
pop     pointer 1
// get value to store in a[2]:
push    constant 27
// set a[2] to 27:
pop     that 2
```

Actual RAM locations of program variables are run-time dependent and thus the addresses shown here are arbitrary examples.

Virtual memory segments just before the operation a[2] = 27:

	local	pointer	that
0	4315		
1

Virtual memory segments just after the operation a[2] = 27:

	local	pointer	that
0	4315		
1	...	4315	
			7
			53
			27
			...

that 0 is now aligned with RAM[4315]

Virtuelle Maschine: Arraybehandlung 2

Hintergrund: Wir betrachten nun $a[i] = x$, wobei $i = 2$ und $x = 31$ die ersten beiden Funktionsargumente sind.

high-level program view

array a

0	7
1	53
2	121
3	8
...	...
9	19

compilation

RAM view

0		
	...	
412	4315	a
	...	
4315	7	array a
4316	53	
4317	121	
4318	8	
	...	
4324	19	
	...	

```
/* Assume that a is the
first local variable and
i and x are the first
two function arguments.
the following implements
a[i] = x or *(a+i) = x */

// get a's base address:
push    local 0
// add index i to a's address:
push    argument 0
add
// point that segment to a+i:
pop     pointer 1
// get value to store in a[i]:
push    argument 1
// set a[i] to x:
pop     that 0
```

Actual RAM locations of program variables are run-time dependent and thus the addresses shown here are arbitrary examples.

Virtual memory segments just before the operation $a[i] = x$:

	local		pointer		that
0	4315	0		0	
1	...	1		1	...

Virtual memory segments just after the operation $a[i] = x$:

	local		pointer		that
0	4315	0		0	31
1	...	1	4317	1	8
					...
		7		7	19

that 0 is now aligned with $RAM[4315+i]$ (for $i = 2$)

Virtuelle Maschine: Funktionsaufrufe

Beispielrechnung:

(Lösung quadratischer Gleichung $ax^2 + bx + c = 0$)

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

```
if (a != 0)
  x = (-b+sqrt(power(b,2) -4*a*c)) / (2*a)
else
  x = -c/b
```

- Um solchen Hochsprachen-Quelltext in VM-Code zu übersetzen, benötigen wir Funktionsaufrufe (hier: sqrt, power)
- **Funktionen** sind ein wesentliches Programmiersprachenelement, möglicherweise die wichtigste Abstraktion in Programmiersprachen.
- Eine einfache Sprache kann durch Funktionen beliebig erweitert werden (benutzerdefinierte Kommandos, Funktionen, Methoden, ...).
- Ziel: Transparente Implementierung, so daß sich (Benutzer-)Funktionen und Basisanweisungen i.w. gleich verhalten (gleicher “look and feel”).

Virtuelle Maschine: Funktionsaufrufe

```
// 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
```

Konventionen für Aufruf und Rücksprung:

- Die aufrufende Funktion legt die Argumente auf den Stapel, ruft die Funktion auf, und wartet dann, bis die aufgerufene Funktion zurückkehrt.
- Bevor die aufgerufene Funktion zurückkehrt, muß sie ein Ergebnis/einen Rückgabewert auf dem Stapel ablegen.
- Beim Rücksprung werden der von der aufgerufenen Funktion benutzte Speicher freigegeben und der Zustand der aufrufenden Funktion wiederhergestellt.
- Endeffekt: Die Argumente der aufgerufenen Funktion werden durch den Rückgabewert ersetzt (wie bei Basisanweisungen).

Virtuelle Maschine: Funktionsaufrufe

```
// 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
```

Implementierung:

- Das Freigeben des lokal benutzten Speichers und vor allem das Wiederherstellen des Zustands der aufrufenden Funktion erfordern große Sorgfalt.
- Die virtuelle Maschine (oder der Compiler) sollten diese Aufgaben übernehmen, um Fehlerquellen so weit wie möglich auszuschließen.
- Wir benutzen hier i.w. den Stapel, um dies zu erreichen:

Der aktuelle Zustand wird auf dem Stapel gemerkt,
lokale Variablen werden auf dem Stapel angelegt.

Virtuelle Maschine: Funktionsaufrufe

VM-Sprachelemente für Funktionsaufrufe:

- `function g nVars`
Definiert (den Start) eine(r) Funktion `g`,
die `nVars` lokale Variablen hat.
- `call g nArgs`
Ruft die Funktion `g` auf, um sie auszuführen;
es wurden `nArgs` Argumente auf dem Stapel abgelegt.
- `return`
Beendet die Ausführung einer Funktion und
gibt die Kontrolle an die aufrufende Funktion zurück.

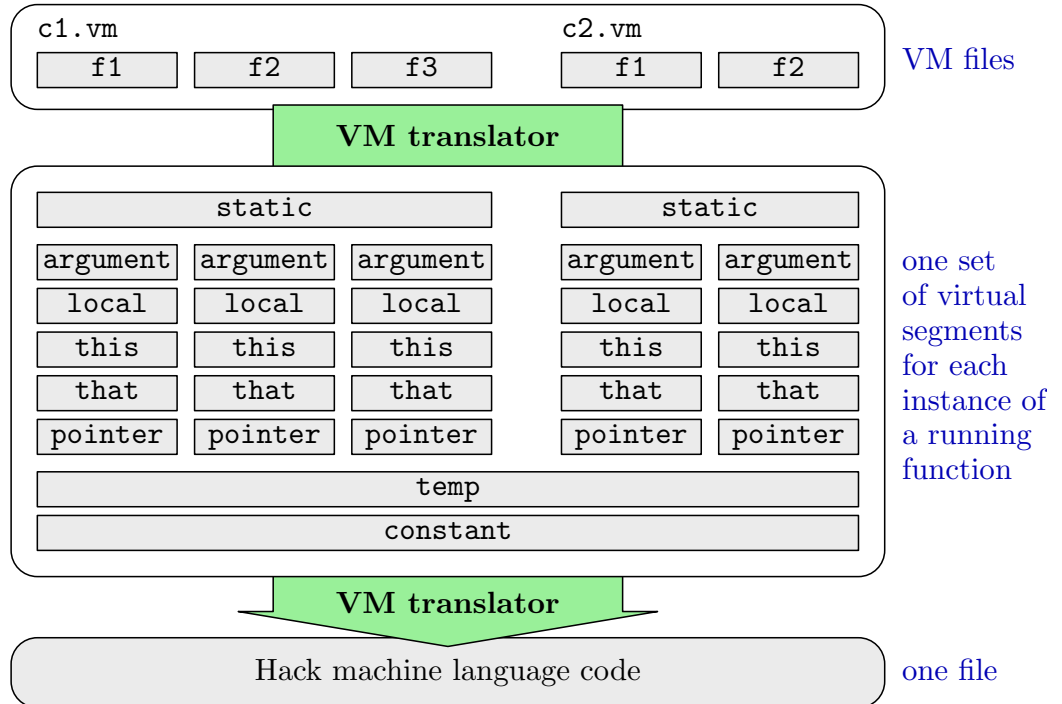
**A well-designed system consist of
a collection of black box modules,
each executing its effect like magic.**

(Steven Pinker: “How the Mind Works”, 1999)

Erinnerung: Übersetzungspfad

Jack-Quelltext (allg.)

```
class c1 {  
    static staticsList;  
    method int f1 (argsList) {  
        var localsList;  
        ...  
    }  
    method int f2 (argsList) {  
        var localsList;  
        ...  
    }  
    method int f3 (argsList) {  
        var localsList;  
        ...  
    }  
}  
  
class c2 {  
    static staticsList;  
    method int f1 (argsList) {  
        var localsList;  
        ...  
    }  
    method int f2 (argsList) {  
        var localsList;  
        ...  
    }  
}
```



Durch die Übersetzung werden den verschiedenen Listen der Klassen und Funktionen Speichersegmente zugewiesen. Für die lokalen Variablen werden sie dynamisch bestimmt.

Virtuelle Maschine: Ablauf eines Funktionsaufrufs

Wenn eine Funktion f eine Funktion g aufruft:

- wird die Rücksprungadresse abgespeichert,
- werden die virtuellen Segmente der Funktion f gesichert,
- werden die lokalen Variablen der Funktion g angelegt und mit 0 initialisiert,
- werden die Segmente `argument` und `local` für g gesetzt,
- wird die Kontrolle an die Funktion g übergeben.

Wenn die Funktion g fertig ist und zur Funktion f zurückgekehrt werden soll:

- werden die lokalen Variablen und die Argumente der Funktion g gelöscht,
- werden die virtuellen Segmente der Funktion f wiederhergestellt,
- wird die Kontrolle an die Funktion f zurückgeben
(zur Rücksprungadresse gesprungen).

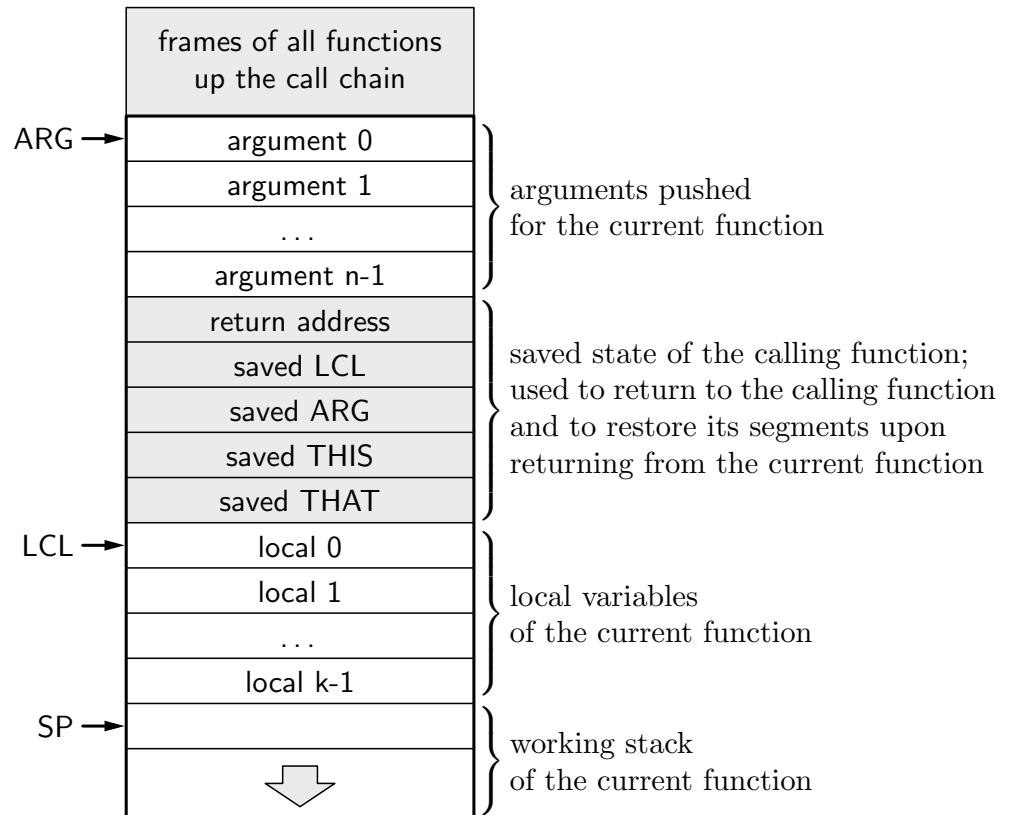
Funktionsaufrufe: Speicherverwaltung

- Bisher: **Arbeitsstapel** (Stapel als Arbeitsspeicher)
 - Ablegen von Argumenten von Basisoperationen mit `push`.
 - Berechnungen mit Basisoperationen (z.B. `add`).
 - Auslesen von Ergebnissen von Basisoperationen mit `pop`.
- Jetzt: **Globaler Stapel** (zur Programmablaufsteuerung)
 - Speicherbereich der die Rahmen (frames) der aktuellen Funktionen enthält (aktiv oder wartend).
 - Der Arbeitsstapel befindet sich am Ende des globalen Stacks.
- **Rahmen einer Funktion** (frame):

○ Funktionsargumente	VM: <code>argument</code>	Assembler: <code>ARG</code>
○ Lokale Variablen	VM: <code>local</code>	Assembler: <code>LCL</code>
○ Andere Speichersegmente	VM: <code>this, that</code>	Assembler: <code>THIS, THAT</code>
○ Arbeitsstapel	VM: implizit	Assembler: <code>SP</code>

Funktionsaufrufe: Speicherverwaltung

- Zu jedem Zeitpunkt warten einige Funktionen und nur die aktuelle Funktion ist aktiv.
- Graue Bereiche: für die aktuelle Funktion nicht von Bedeutung.
- Die aktuelle Funktion sieht nur das Stapelende (Arbeitsstapel).
- Der Rest des Stapels enthält die „eingefrorenen“ Zustände aller Funktionen weiter oben in der Aufrufhierarchie.



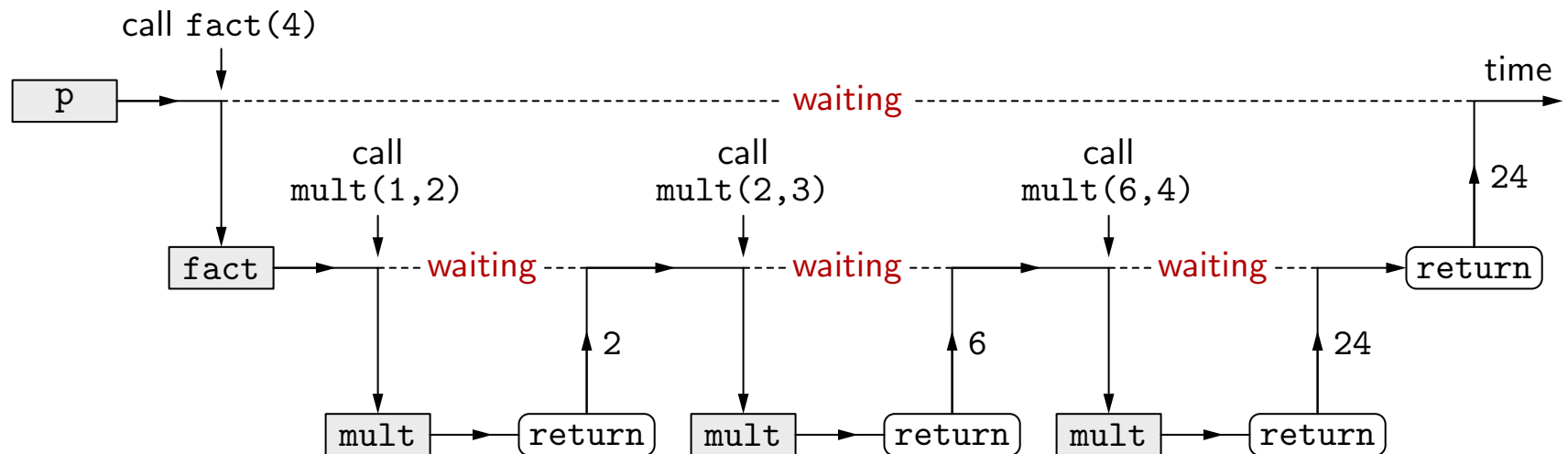
- Die Details der Speicherstruktur hängen von der Implementierung der virtuellen Maschine ab.

Virtuelle Maschine: Funktionsaufrufe

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

```
function fact (int n) {  
  vars res, i;  
  res = 1; i = 1;  
  while (i < n) {  
    i = i+1;  
    res = mult(res,i);  
  }  
  return res;  
}
```

```
function mult (int x, int y) {  
  vars sum, i;  
  sum = 0; i = 0;  
  while (i < y) {  
    sum = sum+x;  
    i = i+1;  
  }  
  return sum;  
}
```

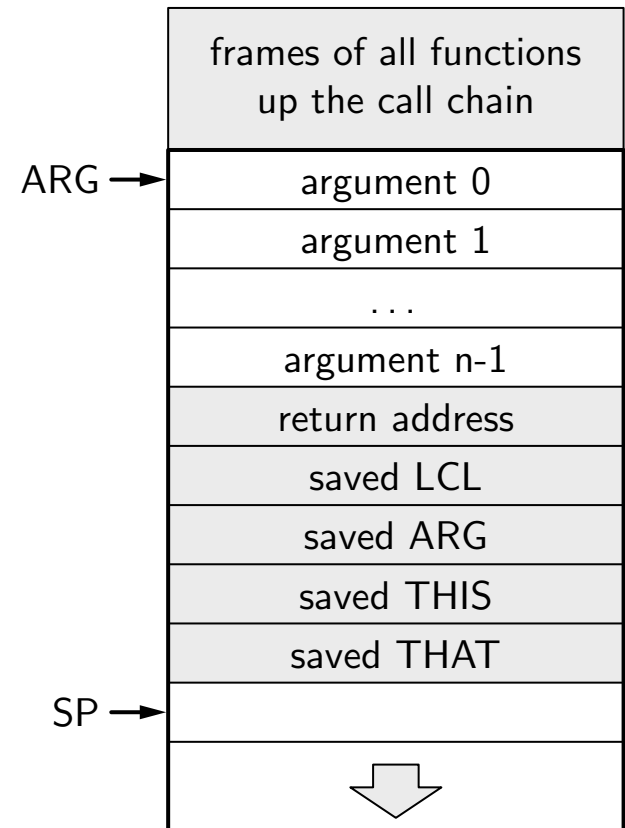


Virtuelle Maschine: Funktionsaufrufe

```
call f n
// call the function f after n arguments
// have been pushed onto the stack

push  retaddr // (use label declared below)
push  LCL     // save LCL  of caller
push  ARG     // save ARG  of caller
push  THIS    // save THIS of caller
push  THAT    // save THAT of caller
ARG = SP-5-n  // n = number of args.
goto  f       // transfer control to f
label retaddr // declare label for return
```

- Falls die virtuelle Maschine implementiert ist als ein Programm, das VM-Code in Assemblercode übersetzt, sollte der Übersetzer die oben dargestellte Logik in Assemblercode erzeugen.

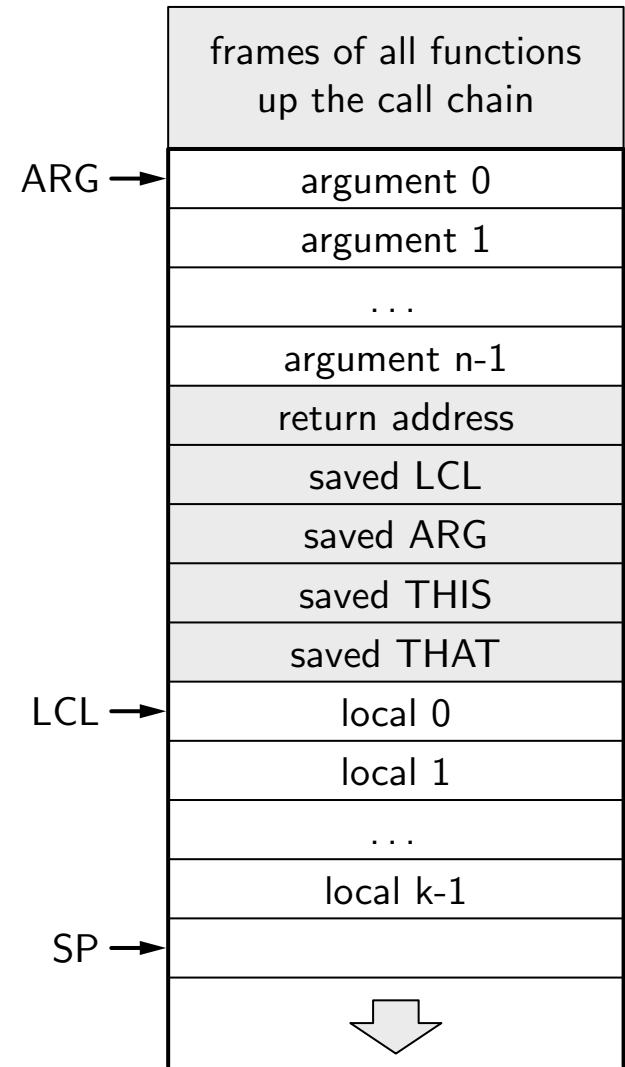


Virtuelle Maschine: Funktionsaufrufe

```
function f k
// declare a function f
// that has k local variables

label f          // declare function label
LCL = SP         // set LCL of f
repeat k times:  // k = number of local vars.
push 0           // init. local variables
```

- Falls die virtuelle Maschine implementiert ist als ein Programm, das VM-Code in Assemblercode übersetzt, sollte der Übersetzer die oben dargestellte Logik in Assemblercode erzeugen.

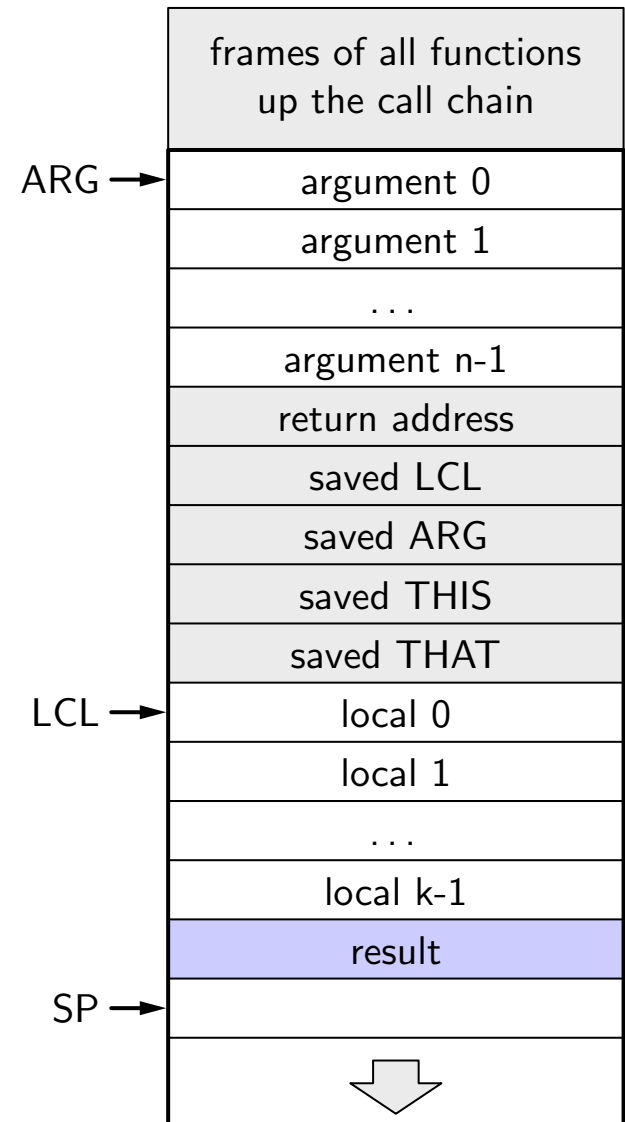


Virtuelle Maschine: Funktionsaufrufe

```
return
// return from a function

pop  ARG 0      // store function result
SP = ARG+1      // restore SP of caller
FRAME = LCL     // FRAME is temp. var.
THAT = *(FRAME-1) // restore THAT of caller
THIS = *(FRAME-2) // restore THIS of caller
ARG = *(FRAME-3) // restore ARG of caller
LCL = *(FRAME-4) // restore LCL of caller
RET = *(FRAME-5) // RET is temp. var.
goto RET        // go to return address
```

- Falls die virtuelle Maschine implementiert ist als ein Programm, das VM-Code in Assemblercode übersetzt, sollte der Übersetzer die oben dargestellte Logik in Assemblercode erzeugen.

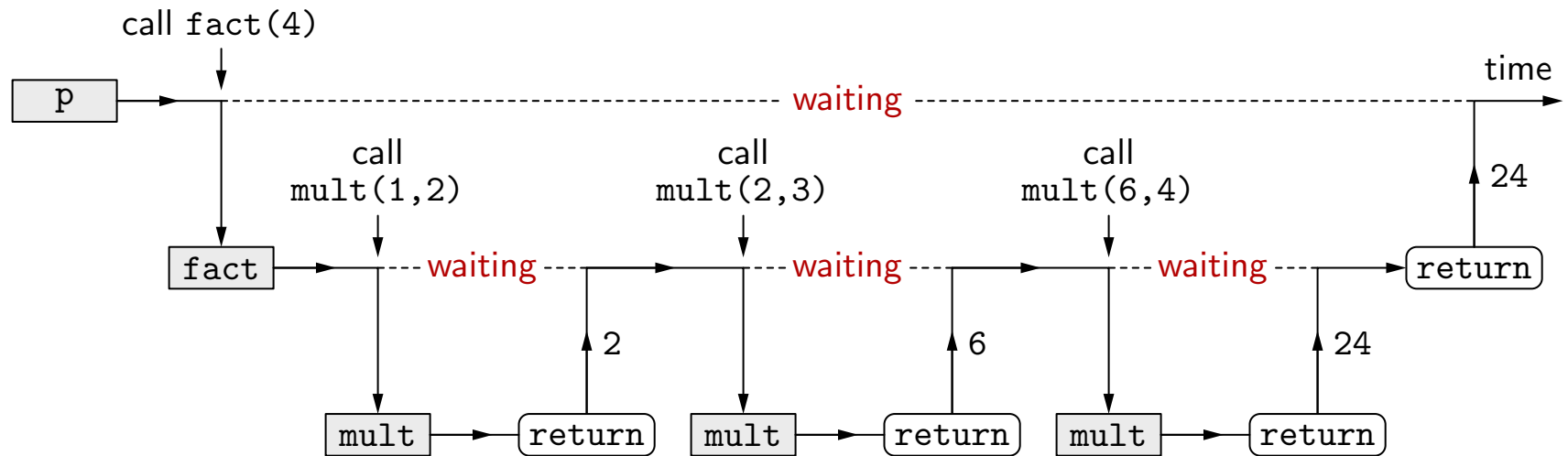


Virtuelle Maschine: Funktionsaufrufe

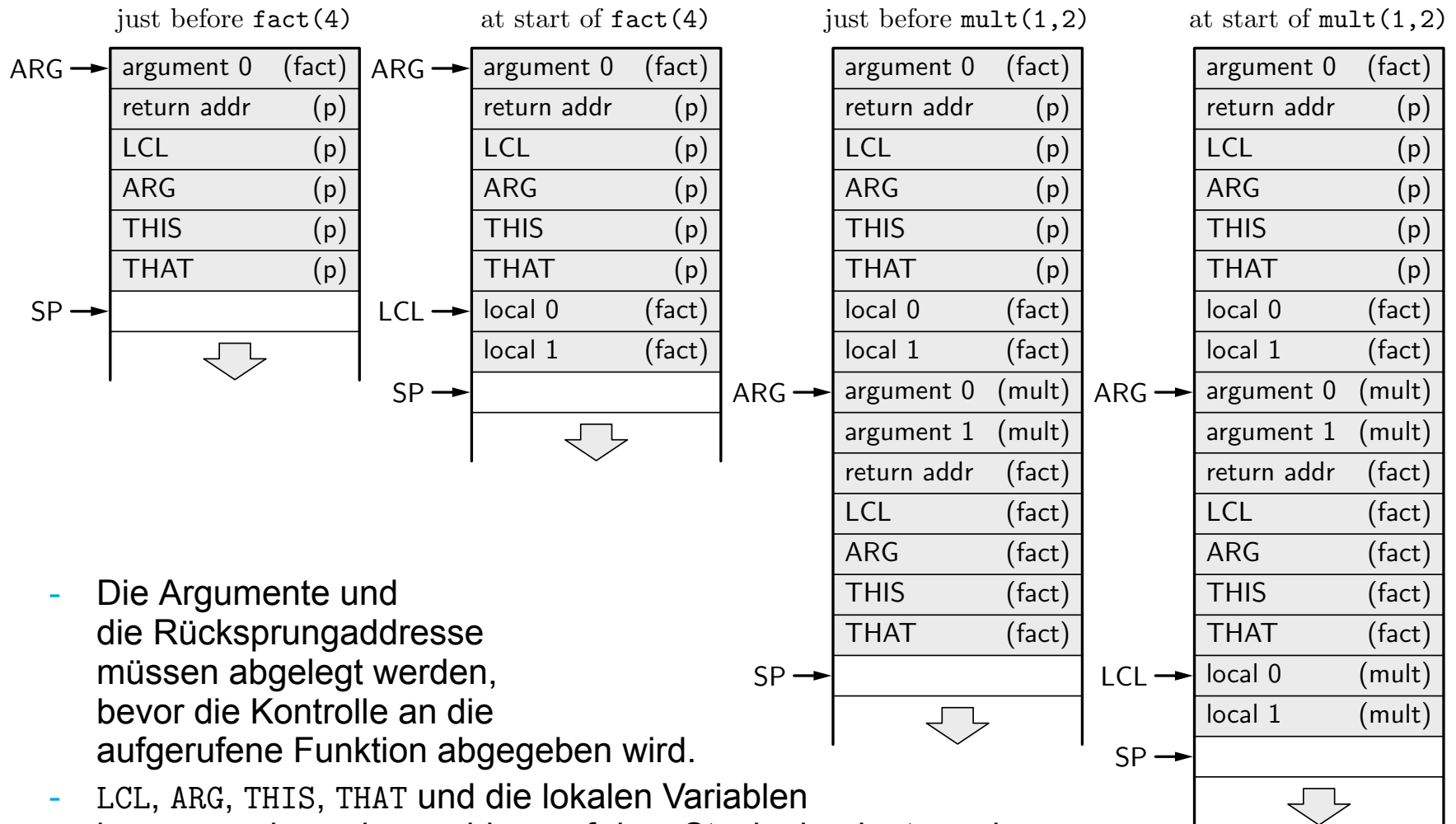
```
function p (...) {  
  ...  
  fact(4);  
  ...  
}
```

```
function fact (int n) {  
  vars res, i;  
  res = 1; i = 1;  
  while (i < n) {  
    i = i+1;  
    res = mult(res,i);  
  }  
  return res;  
}
```

```
function mult (int x, int y) {  
  vars sum, i;  
  sum = 0; i = 0;  
  while (i < y) {  
    sum = sum+x;  
    i = i+1;  
  }  
  return sum;  
}
```

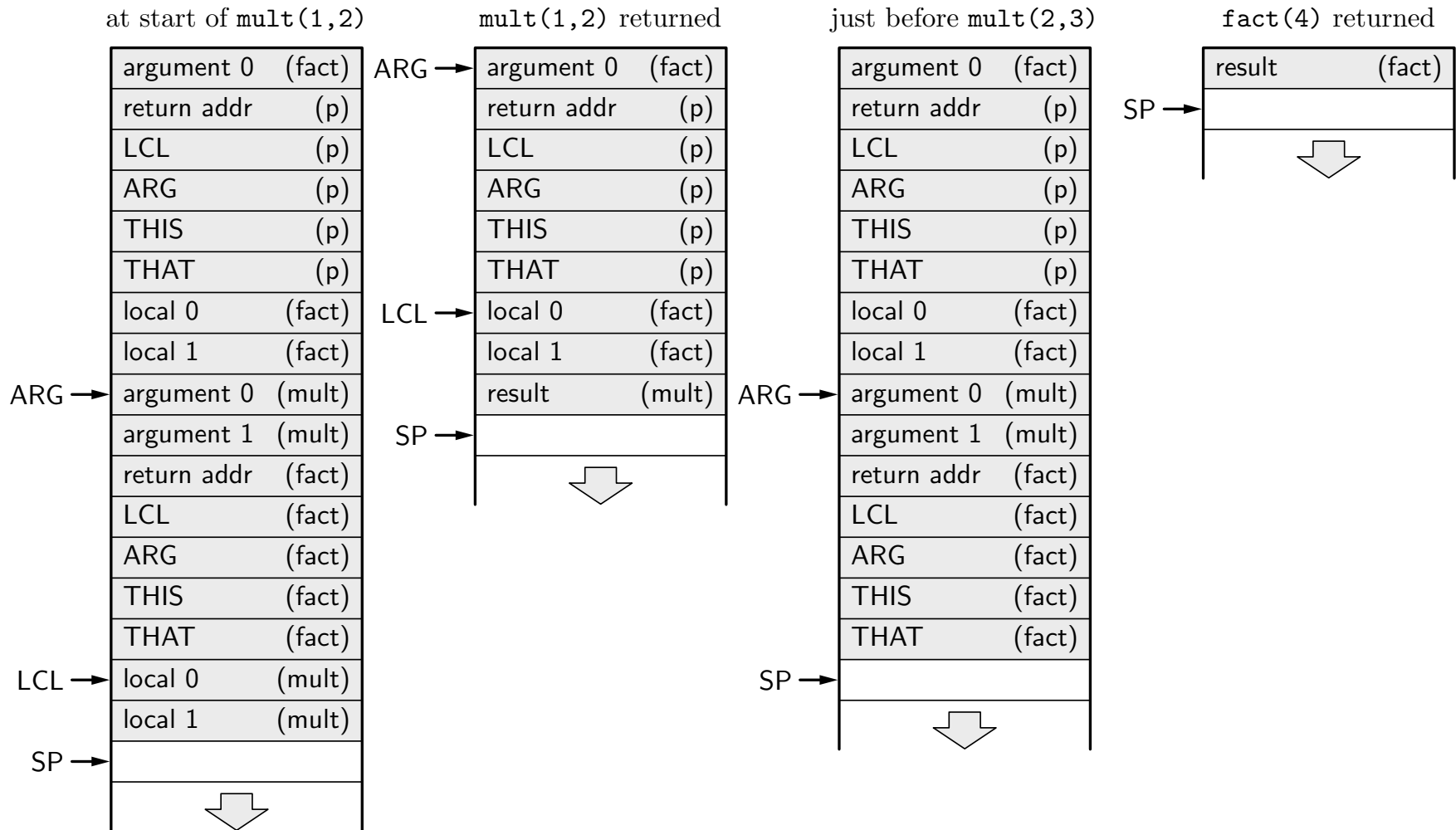


Virtuelle Maschine: Funktionsaufrufe



- Die Argumente und die Rücksprungadresse müssen abgelegt werden, bevor die Kontrolle an die aufgerufene Funktion abgegeben wird.
- LCL, ARG, THIS, THAT und die lokalen Variablen können vorher oder nachher auf dem Stack abgelegt werden. (Darstellung hier: LCL, ARG, THIS, THAT vorher, lokale Variablen nachher)

Virtuelle Maschine: Funktionsaufrufe

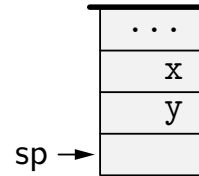


- Nach der Rückkehr einer Funktion steht das Ergebnis auf dem Stapel.
- Argumente, Sicherungen und lokale Variablen wurden vom Stapel entfernt.

Virtuelle Maschine: Befehlssatz

Arithmetisch-logische Operationen

add	$x + y$	Ganzzahladdition (Zweierkomplement)
sub	$x - y$	Ganzzahlsubtraktion (Zweierkomplement)
neg	$-y$	arithmetische Negation (Zweierkomplement)
eq	-1 falls $x = y$, sonst 0	Test auf Gleichheit
gt	-1 falls $x > y$, sonst 0	Test auf größer
lt	-1 falls $x < y$, sonst 0	Test auf kleiner
and	$x \& y$	bitweises Und
or	$x \mid y$	bitweises Oder
not	$\sim y$	bitweise Negation



Speicherzugriff

push <u>segment</u> <u>index</u>	Ablegen des Inhalts von <u>segment[index]</u> auf dem Stapel.
pop <u>segment</u> <u>index</u>	Speichern des obersten Stapelelementes in <u>segment[index]</u> .
Speichersegmente:	constant, static, local, argument, this, that, pointer, temp

Programmablaufsteuerung

label <u>labelname</u>	Definiert eine Marke im Programmtext, z.B. als Sprungziel.
goto <u>labelname</u>	Springt zu einer Marke im Programmtext (unbedingter Sprung).
if-goto <u>labelname</u>	Springt zu einer Marke im Programmtext, wenn das oberste Stapelelement verschieden von 0 ist. (Dieses Element wird vom Stapel entfernt.)

Funktionen und Funktionsaufrufe

function <u>fname</u> k	Definiert eine Funktion mit dem Namen <u>fname</u> (k : Anzahl lokaler Variablen).
call <u>fname</u> n	Ruft die Funktion mit dem Namen <u>fname</u> auf (n : Anzahl Funktionsargumente).
return	Kehrt aus einer Funktion zurück (oberstes Stapelelement ist Rückgabewert).

Bemerkung zur Stapelrichtung

- Anschauliche, alltägliche Stapel, wie z.B. Aktenstapel, wachsen von unten nach oben.
- Der **Stapel in einem Rechner wächst** dagegen von **oben nach unten** (d.h., von höheren zu niedrigeren Speicheradressen).
(Siehe C-Programm rechts, sowie die Adressierungsarten $-(An)$ (= push) und $(An)+$ (= pop) des Motorola 68000.)
Die vorangehende Beschreibung kann weitgehend auf beide Weisen aufgefaßt werden.
- **Achtung:** In der virtuellen Maschine des Hack-Systems wächst der Stapel von unten nach oben (aufsteigende Adressen)!
Dies ist bei Emulatorläufen zu beachten!

Einfaches C-Programm zur Demonstration der Stapelrichtung

```
void d (void)
{ int x; printf("%p\n", &x); }

void c (void)
{ int x; printf("%p\n", &x); d(); }

void b (void)
{ int x; printf("%p\n", &x); c(); }

void a (void)
{ int x; printf("%p\n", &x); b(); }

void main (void)
{ int x; printf("%p\n", &x); a(); }
```

Ausgabe des C-Programms:

0x7ffe73a5f4f4	(x in main)
0x7ffe73a5f4d4	(x in a)
0x7ffe73a5f4b4	(x in b)
0x7ffe73a5f494	(x in c)
0x7ffe73a5f474	(x in d)

Virtuelle Maschine: Programmstart

- **Konvention:** Eine Klasse muß den Namen `Main` haben.
und diese Klasse muß mindestens eine Funktion `main` haben.
- **Konvention:** Wenn das Programm ausgeführt werden soll,
wird die Funktion `Main.main` ausgeführt.
- Nachdem das Hochsprachen-Programm übersetzt worden ist,
gibt es für jede Klassendatei eine Datei mit Endung `.vm`.
- Eine der Bibliotheken des Betriebssystems heißt `Sys`.
Diese Bibliothek enthält eine Funktion `Sys.init`,
die einige Anweisungen zur Systeminitialisierung ausführt,
dann `Main.main` aufruft, und in einer Endlosschleife endet.
- Um zu starten, muß die Implementierung der virtuellen Maschine
(in Maschinensprache) die folgenden Operationen ausführen:

```
SP = 256      // init. the stack pointer to 0x0100  
call Sys.init // call the initialization function
```

Virtuelle Maschine: Emulator

Virtual Machine Emulator (2.5) - /home/borgelt/teach/rsn/nand2tetris/src/Main.vm

File View Run Help

Animate: Program flow View: Screen Format: Decimal

Slow Fast

Program

0	function	Main.main 1
1	push	constant 5
2	call	Main.f 1
3	pop	local 0
4	return	
0	function	Main.f 2
1	push	constant 1
2	pop	local 1
	label	Main.f\$LOOP
3	push	local 0
4	push	argument 0
5	push	local 1
6	lt	
7	if-goto	Main.f\$END
8	push	local 1

Static

0	0
1	0
2	0
3	0
4	0

Local

Argument

This

That

Temp

--	--

Stack

Call Stack

Global Stack

256	0
257	0
258	0
259	0
260	0
261	0
262	0
263	0
264	0
265	0
266	0
267	0
268	0
269	0
270	0

RAM

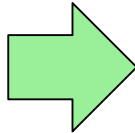
SP:	0	256
LCL:	1	0
ARG:	2	0
THIS:	3	0
THAT:	4	0
Temp0:	5	0
Temp1:	6	0
Temp2:	7	0
Temp3:	8	0
Temp4:	9	0
Temp5:	10	0
Temp6:	11	0
Temp7:	12	0
R13:	13	0
R14:	14	0

Virtuelle Maschine, Assembler und Maschinensprache

Virtuelle Maschine

```
...
push x
push width
add
push 511
gt
if-goto L1
goto L2
L1:
push 511
push width
sub
pop x
L2:
...
```

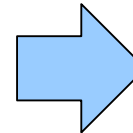
VM-Übersetzer



Assemblersprache

```
// push 511
@511
D = A    // D = 511
@SP
A = M    // A = SP
M = D    // *SP = D
@SP
M = M+1  // SP++
```

Assembler



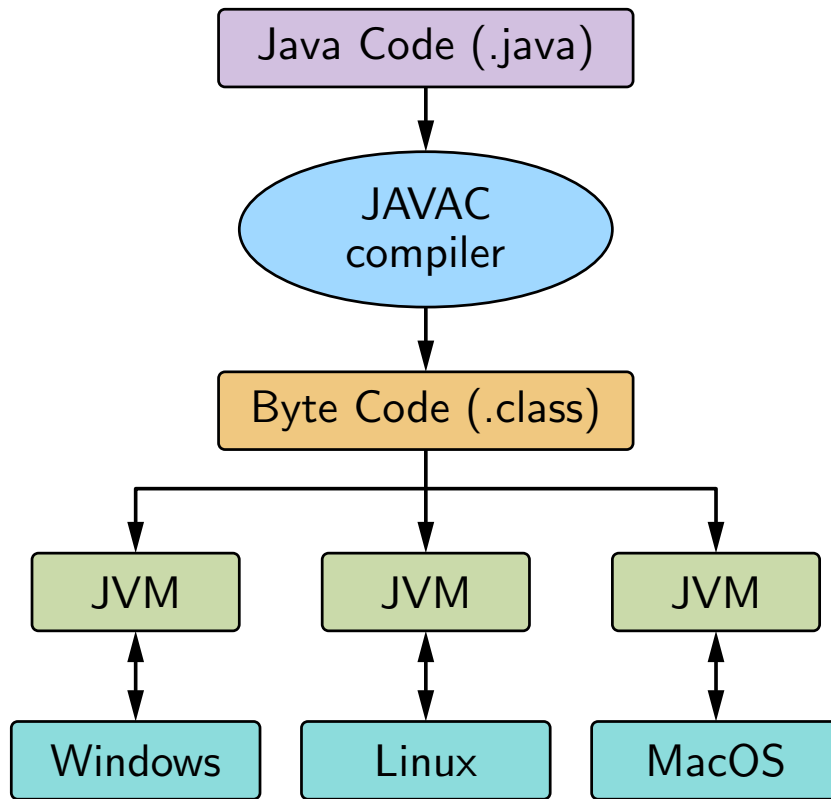
Maschinensprache

```
0000000000000000
1111110111001000
```

Die Maschinensprache besteht nicht mehr aus Symbolen, sondern aus Binärzahlen. Sie ist daher von einem Menschen nur mit großen Schwierigkeiten zu lesen. Deshalb abstrahieren höhere Sprachen von ihr.

- Die **Maschinensprache** kann von einem Rechner direkt ausgeführt werden!

Virtuelle Maschine: Vergleich mit Java

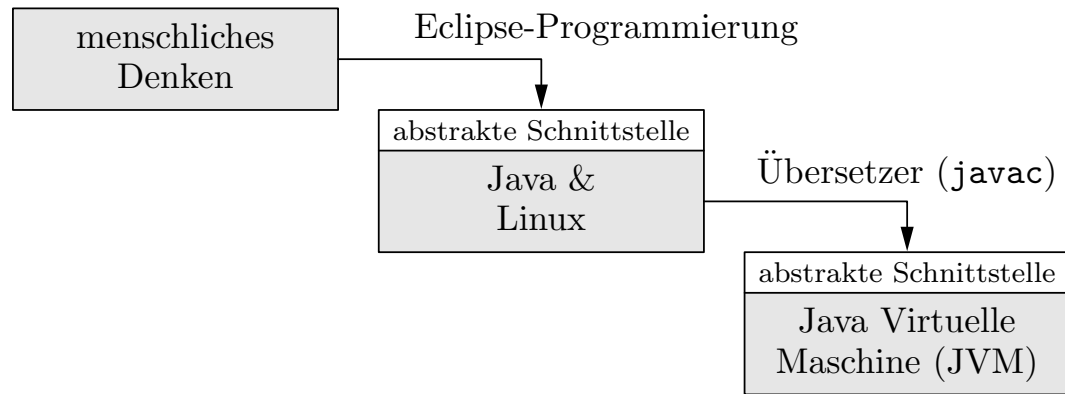


- JVM: Java Virtual Machine
- Die virtuelle Maschine muß auf das Betriebssystem und den verwendeten Rechner (hardware) passen.

- Ein Java-Quelltext (in Dateien mit der Endung `.java`) kann von einem Rechner nicht direkt ausgeführt werden.
- Er wird durch einen Java-Übersetzer (ein Programm namens `javac`) in sogenannten "Byte Code" übersetzt. Dieser entspricht dem Zwischencode mit Stapelbefehlen in unserem Beispiel.
- Der erzeugte "Byte Code" kann ebenfalls nicht direkt von einem Rechner ausgeführt werden.
- Er wird von einer virtuellen Maschine (JVM) auf Befehle des verwendeten Rechners und Aufrufe des verwendeten Betriebssystems abgebildet.

Virtuelle Maschine: Vergleich mit Java

Spezialfall Java und Linux:



Java-Byte-Code Beispiel:

```
// byte code stream
03 3b 84 00 01 1a
05 68 3b a7 ff f9
```

```
// disassembly
iconst_0          // 03
istore_0          // 3b
iinc 0, 1         // 84 00 01
iload_0           // 1a
iconst_2          // 05
imul              // 68
istore_0          // 3b
goto -7           // a7 ff f9
```

Warum so viele Abstraktionsebenen?

- Nur ein Übersetzer für Java in Byte Code nötig, unabhängig von Betriebssystem und Rechner.
- Der Byte Code ist viel leichter in Sprachen tieferer Ebenen zu übersetzen, da er eine wesentlich einfachere Struktur hat als Java.

Der Java-Byte-Code ist viel mächtiger als der Zwischencode aus unserem Beispiel.

Zusammenfassung: Virtuelle Maschine

- **Höhere Programmiersprachen und Übersetzung**
 - Direkte und zweistufige Übersetzung
 - Zwischensprache und virtuelle Maschine
 - Systembasierte und prozeßbasierte virtuelle Maschinen
- **Virtuelle Maschine des Hack-Systems**
 - Stapel(-speicher) und ihre Operationen
 - Stapelarithmetik (arithmetische und logische Operationen)
 - Speicherzugriff, Speicheraufteilung, Speichersegmente
 - Programmablauf (bedingte Anweisungen und Schleifen)
 - Objekt- und Arraybehandlung
 - Funktionsaufrufe, globaler Stapel zur Steuerung
 - Programmstart