	0	1	2
0		2	1
1	2		0
2	1	0	

Offenbar hängt das Ergebnis nur von der Summe der beiden Argumente ab ...

	0	1	2
0		1	2
1	1		3
2	2	3	

Um solche Tabellen leicht implementieren zu können, stellt Java das `switch`-Statement zur Verfügung:

```
public static byte free (byte a, byte b) {
   switch (a+b) {
   case 1:   return 2;
   case 2:   return 1;
   case 3:   return 0;
   default:  return -1;
   }
}
```

Allgemeine Form eines `switch`-Statements:

```
switch ( expr ) {
case const_0 :    ss_0 ( break; ) ?
case const_1 :    ss_1 ( break; ) ?
    ...
case const_{k-1} :    ss_{k-1} ( break; ) ?
( default:    ss_k ) ?
}
```

- expr sollte eine ganze Zahl (oder ein `char`) sein.
- Die const_i sind ganz-zahlige Konstanten.
- Die ss_i sind die alternativen Statement-Folgen.

- `default` beschreibt den Fall, bei dem keiner der Konstanten zutrifft.
- Fehlt ein `break`-Statement, wird mit der Statement-Folge der nächsten Alternative fortgefahren.

- `default` beschreibt den Fall, bei dem keiner der Konstanten zutrifft.
- Fehlt ein `break`-Statement, wird mit der Statement-Folge der nächsten Alternative fortgefahren.

Eine einfachere Lösung in unserem Fall ist :

```
public static byte free (byte a, byte b) {
   return (byte) (3-(a+b));
   }
```

Für einen Turm der Höhe $h = 4$ liefert das:

```
move 0 to 1
move 0 to 2
move 1 to 2
move 0 to 1
move 2 to 0
move 2 to 1
move 0 to 1
move 0 to 2
move 1 to 2
move 1 to 0
move 2 to 0
move 1 to 2
move 0 to 1
move 0 to 2
move 1 to 2
```

Bemerkungen:

- `move()` ist rekursiv, aber nicht end-rekursiv.
- Sei $N(h)$ die Anzahl der ausgegebenen Moves für einen Turm der Höhe $h \geq 0$. Dann ist

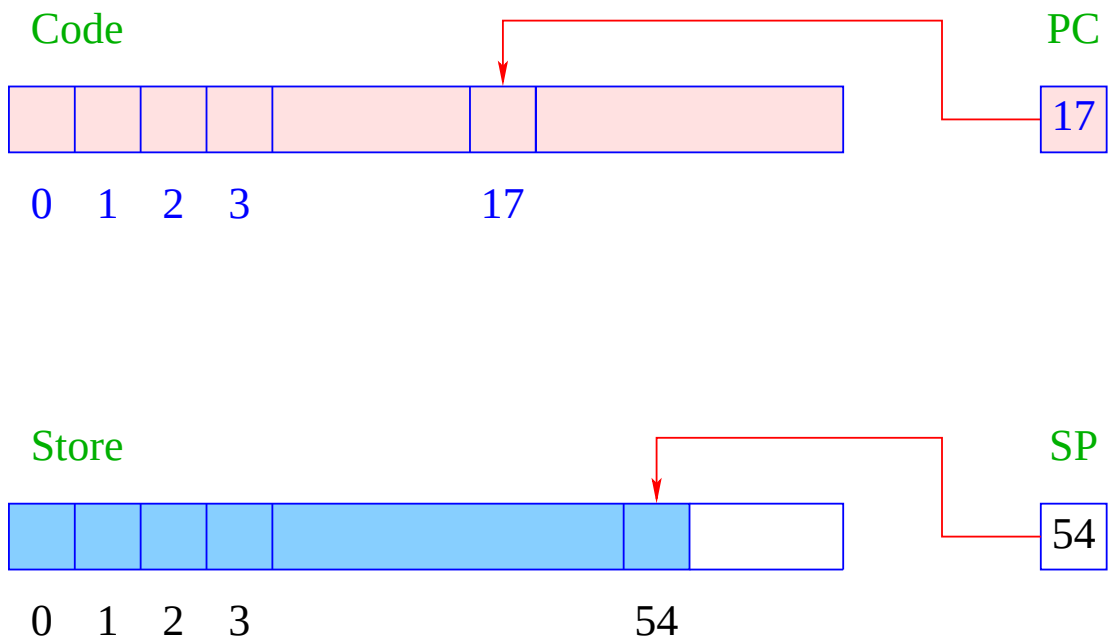
$$\begin{aligned} N(0) &= 0 \qquad\qquad \text{und für } h > 0, \\ N(h) &= 1 + 2 \cdot N(h-1) \end{aligned}$$

- Folglich ist $N(h) = 2^h - 1$.
- Bei genauerer Analyse des Problems lässt sich auch ein nicht ganz so einfacher nicht-rekursiver Algorithmus finden ... (wie könnte der aussehen?)

  Hinweis: Offenbar rückt die kleinste Scheibe in jedem zweiten Schritt eine Position weiter ...

9 Von MiniJava zur JVM

Architektur der JVM:

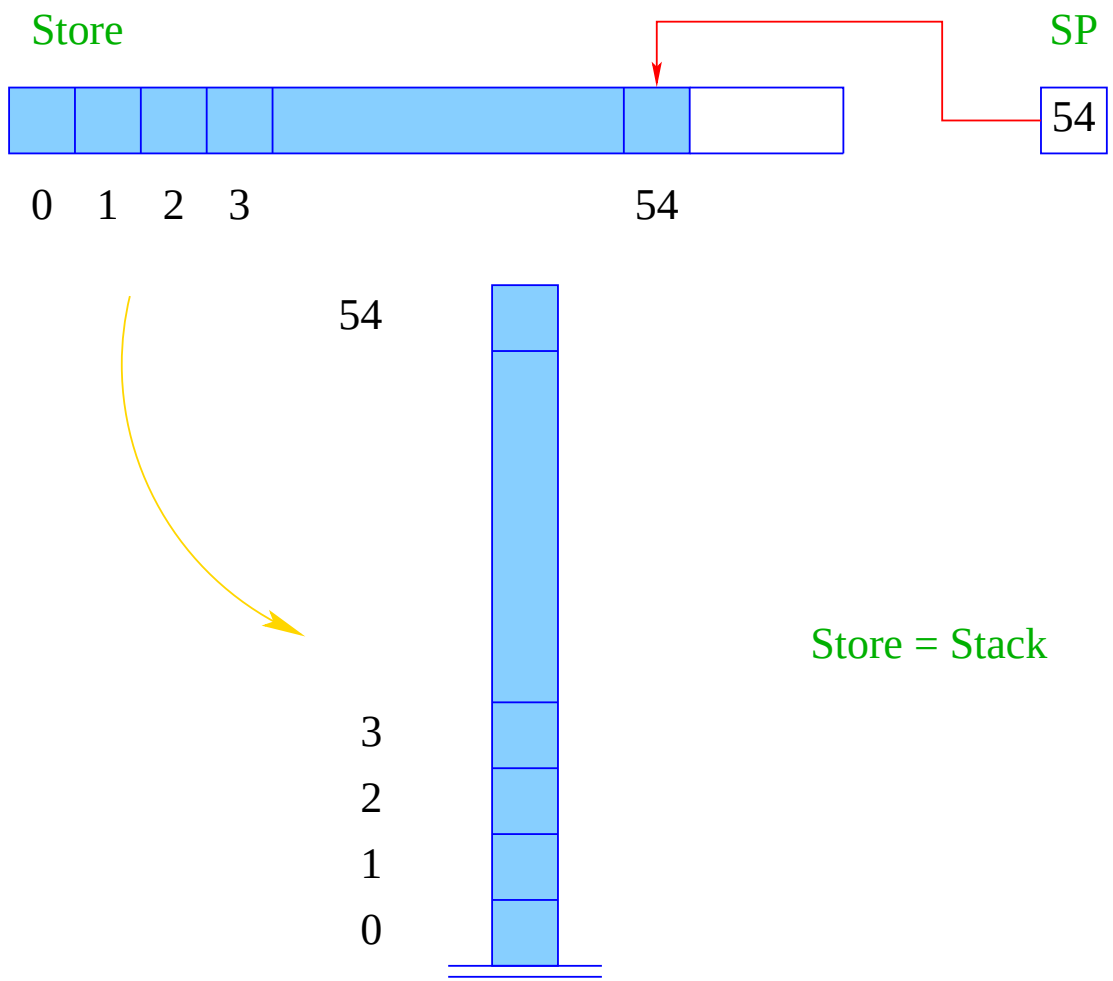


Code	=	enthält JVM-Programm;
		jede Zelle enthält einen Befehl;
PC	=	Program Counter –
		zeigt auf nächsten auszuführenden Befehl;
Store	=	Speicher für Daten;
		jede Zelle kann einen Wert aufnehmen;
SP	=	Stack-Pointer –
		zeigt auf oberste belegte Zelle.

Achtung:

- Programm wie Daten liegen im Speicher – aber in verschiedenen Abschnitten.
- Programm-Ausführung holt nacheinander Befehle aus Code und führt die entsprechenden Operationen auf Store aus.

Konvention:



Befehle der JVM:

`int`-Operatoren:	NEG, ADD, SUB, MUL, DIV, MOD
`boolean`-Operatoren:	NOT, AND, OR
Vergleichs-Operatoren:	LESS, LEQ, EQ, NEQ
Laden von Konstanten:	CONST i, TRUE, FALSE
Speicher-Operationen:	LOAD i, STORE i
Sprung-Befehle:	JUMP i, FJUMP i
IO-Befehle:	READ, WRITE
Reservierung von Speicher:	ALLOC i
Beendung des Programms:	HALT

Ein Beispiel-Programm:

```
      ALLOC 2
      READ
      STORE 0
      READ
      STORE 1
A:    LOAD 0
      LOAD 1
      NEQ
      FJUMP D
      LOAD 0
      LOAD 1
      LESS
      FJUMP B
      LOAD 1
      LOAD 0
      SUB
      STORE 1
      JUMP C
B:    LOAD 0
      LOAD 1
      SUB
      STORE 0
C:    JUMP A
D:    LOAD 1
      WRITE
      HALT
```

- Das Programm berechnet den GGT :-)
- Die Marken (Labels) A, B, C, D bezeichnen symbolisch die Adressen der zugehörigen Befehle:

$$\begin{aligned} \mathsf{A} &= 5 \\ \mathsf{B} &= 18 \\ \mathsf{C} &= 22 \\ \mathsf{D} &= 23 \end{aligned}$$

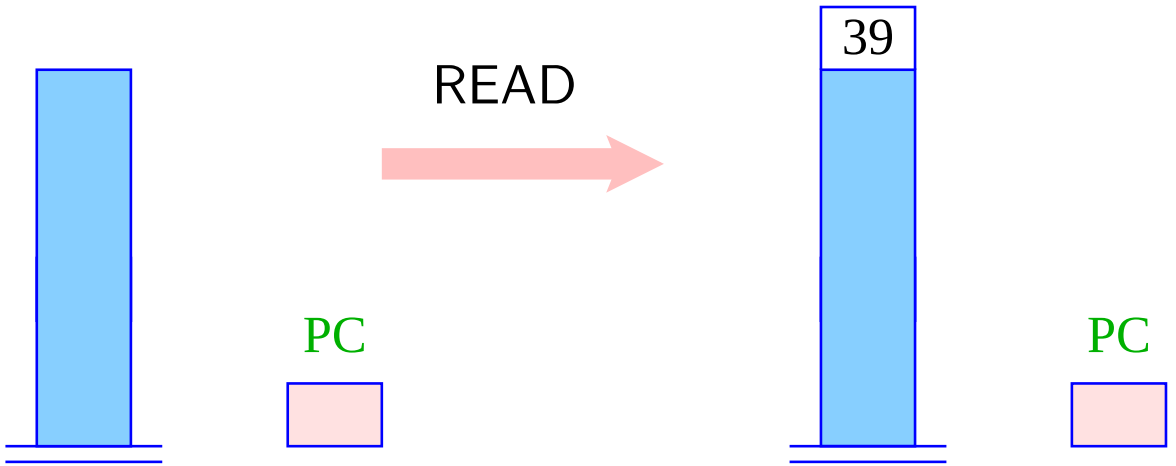
- ... können vom Compiler leicht in die entsprechenden Adressen umgesetzt werden (wir benutzen sie aber, um uns besser im Programm zurechtzufinden :-)

Bevor wir erklären, wie man MiniJava in JVM-Code übersetzt, erklären wir, was die einzelnen Befehle bewirken.

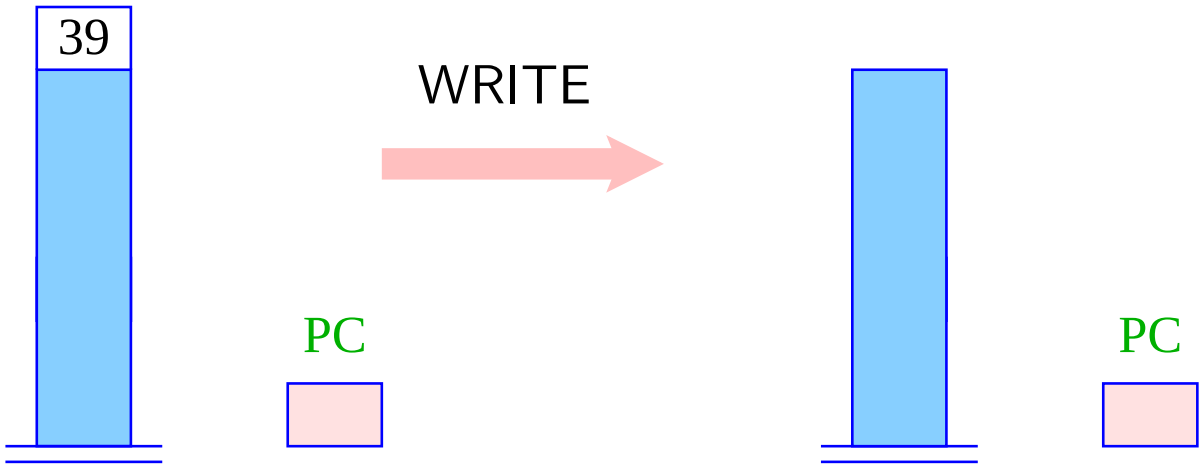
Idee:

- Befehle, die Argumente benötigen, erwarten sie am oberen Ende des Stack.
- Nach ihrer Benutzung werden die Argumente vom Stack herunter geworfen.
- Mögliche Ergebnisse werden oben auf dem Stack abgelegt.

Betrachten wir als Beispiele die IO-Befehle READ und WRITE.



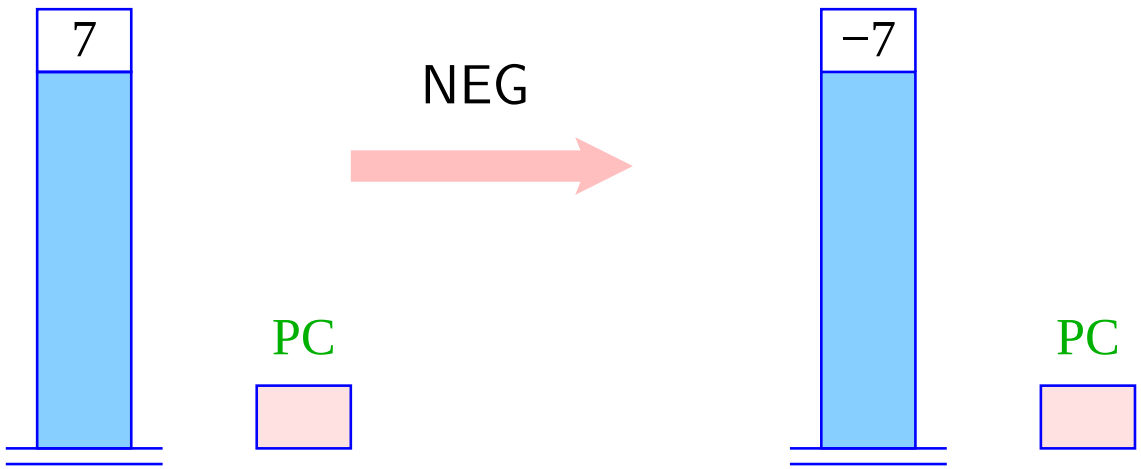
... falls 39 eingegeben wurde

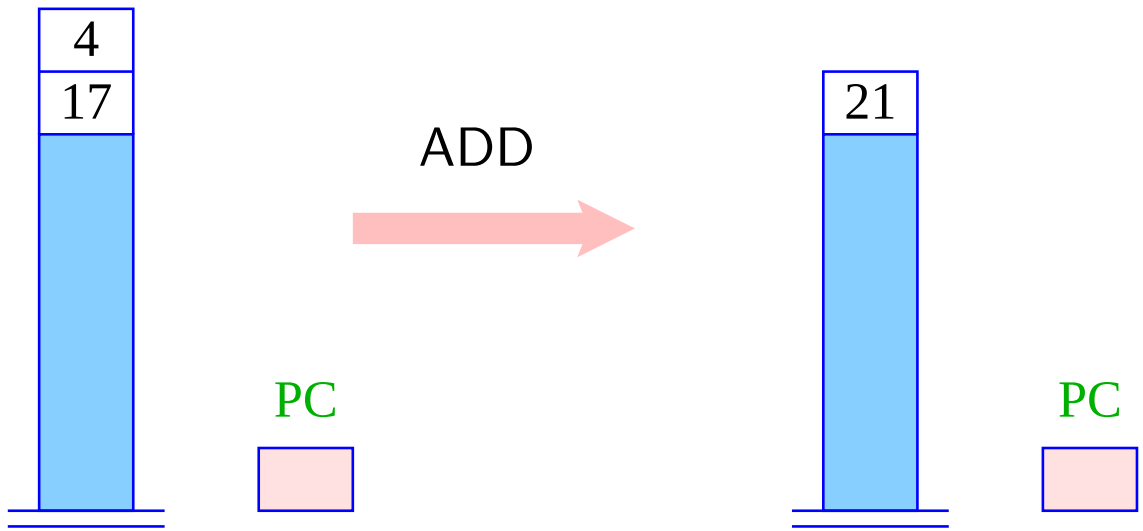


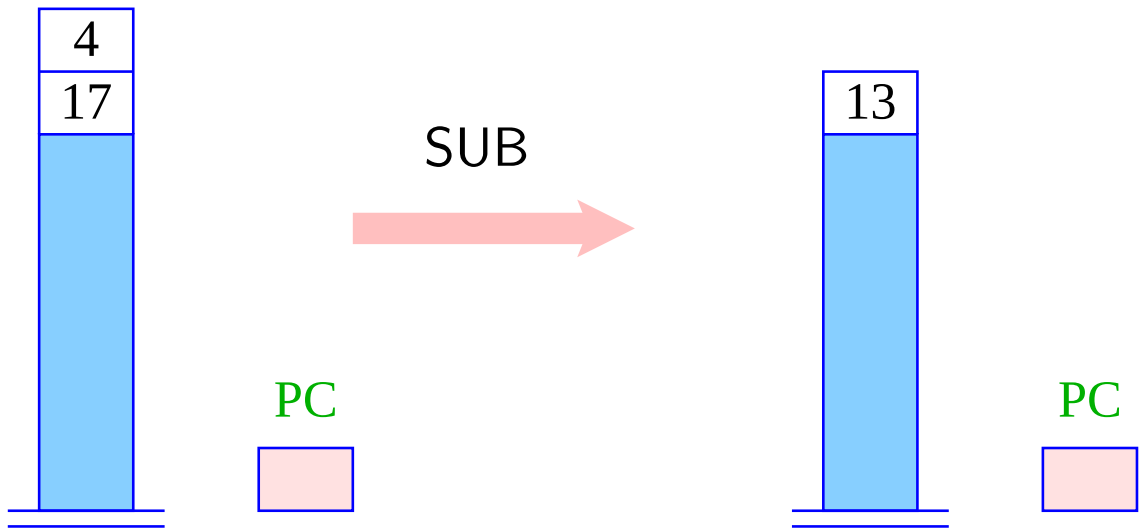
... wobei 39 ausgegeben wird

Arithmetik

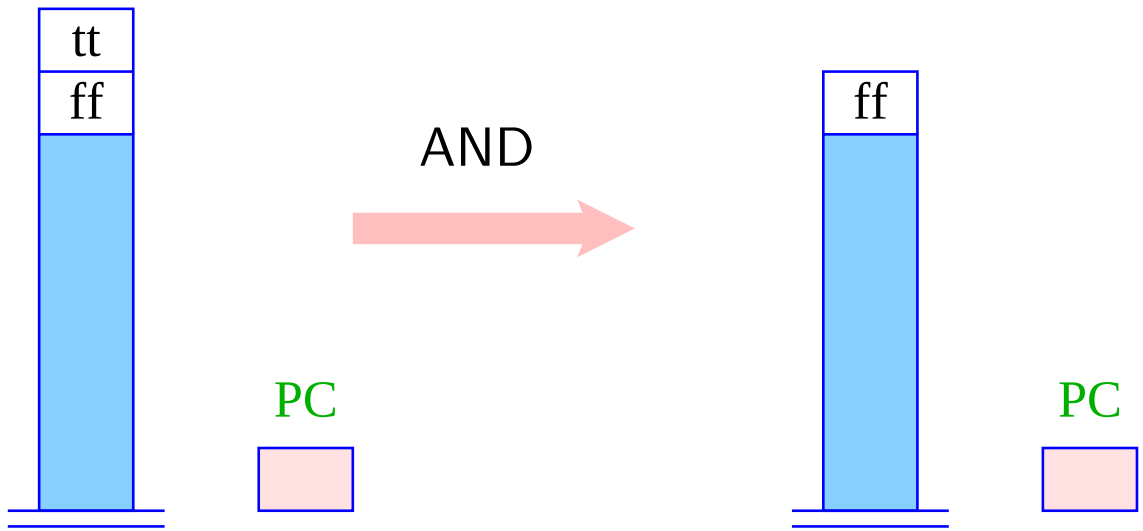
- Unäre Operatoren modifizieren die oberste Zelle.
- Binäre Operatoren verkürzen den Stack.

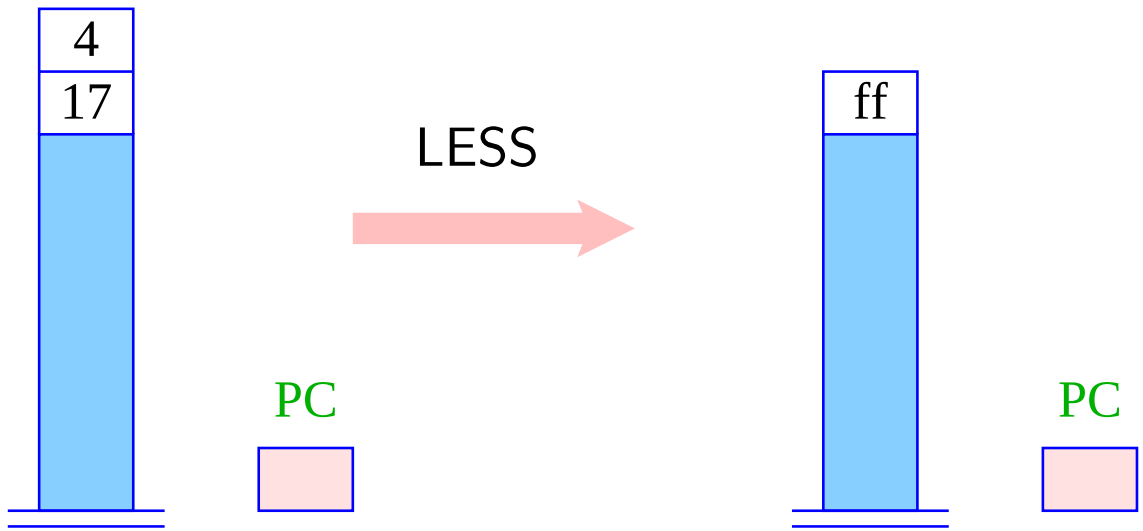

7
NEG
−7
PC
PC


4
17
ADD
21
PC
PC


4
17
SUB
13
PC
PC

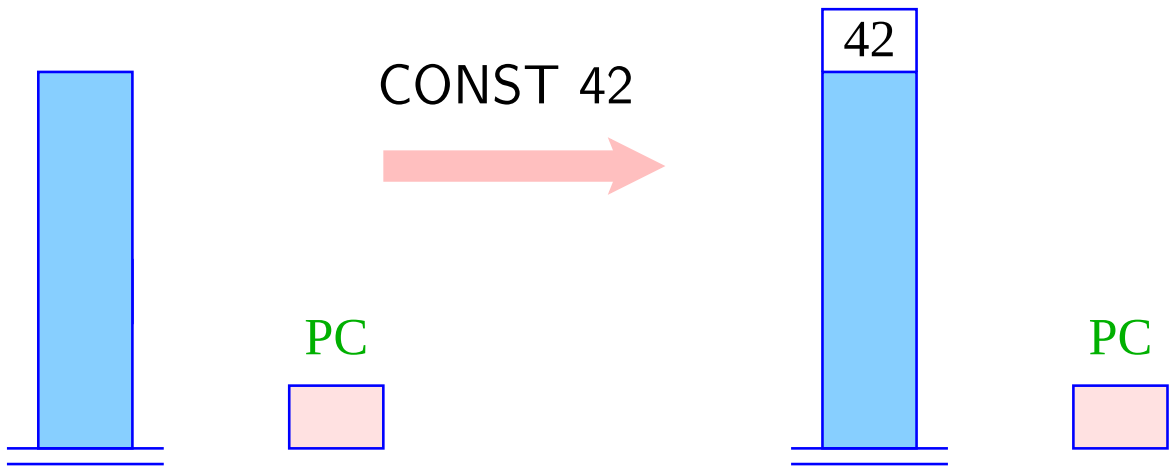
- Die übrigen arithmetischen Operationen MUL, DIV, MOD funktionieren völlig analog.
- Die logischen Operationen NOT, AND, OR ebenfalls – mit dem Unterschied, dass sie statt mit ganzen Zahlen, mit Intern-Darstellungen von `true` und `false` arbeiten (hier: “tt” und “ff”).
- Auch die Vergleiche arbeiten so – nur konsumieren sie ganze Zahlen und liefern einen logischen Wert.

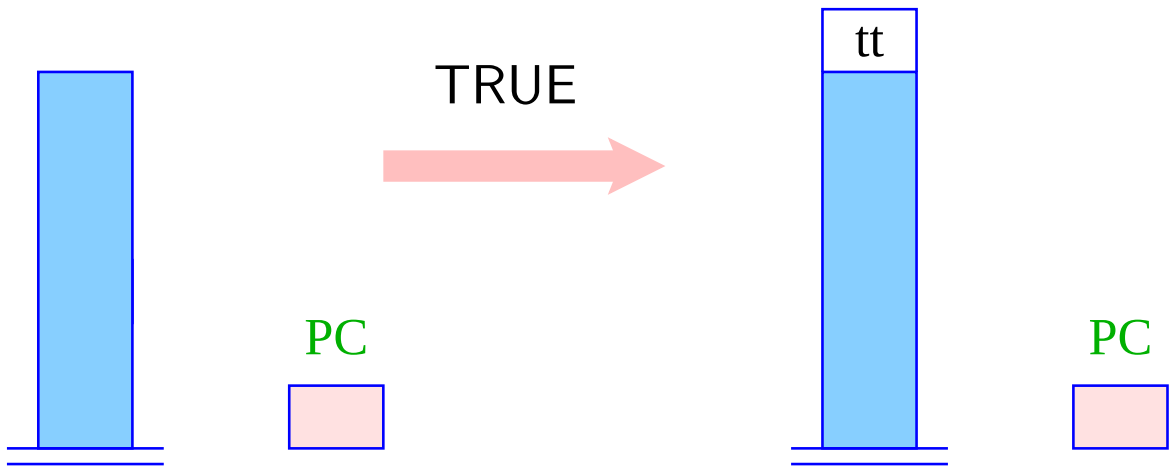

tt
ff
AND
ff
PC
PC

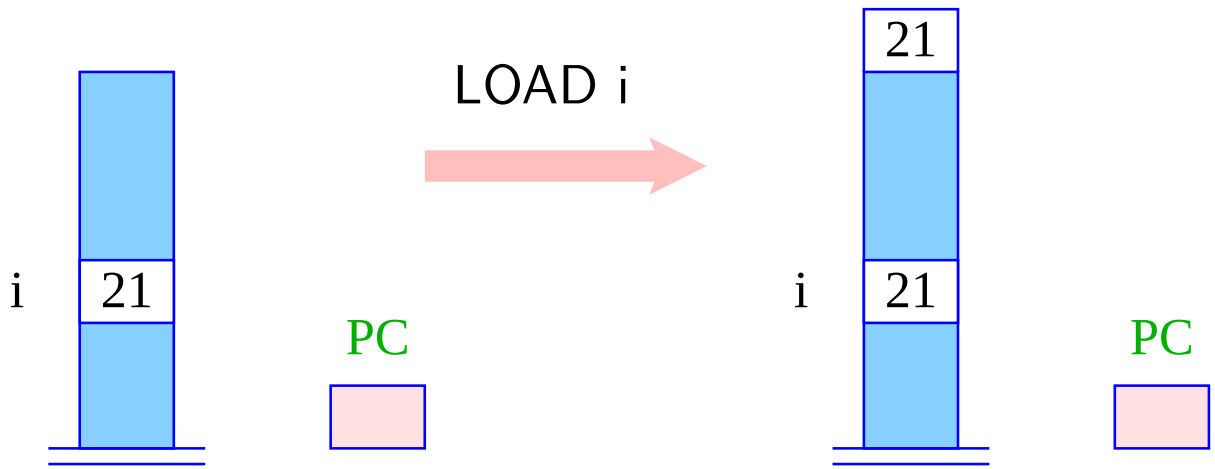

4
17
LESS
ff
PC
PC

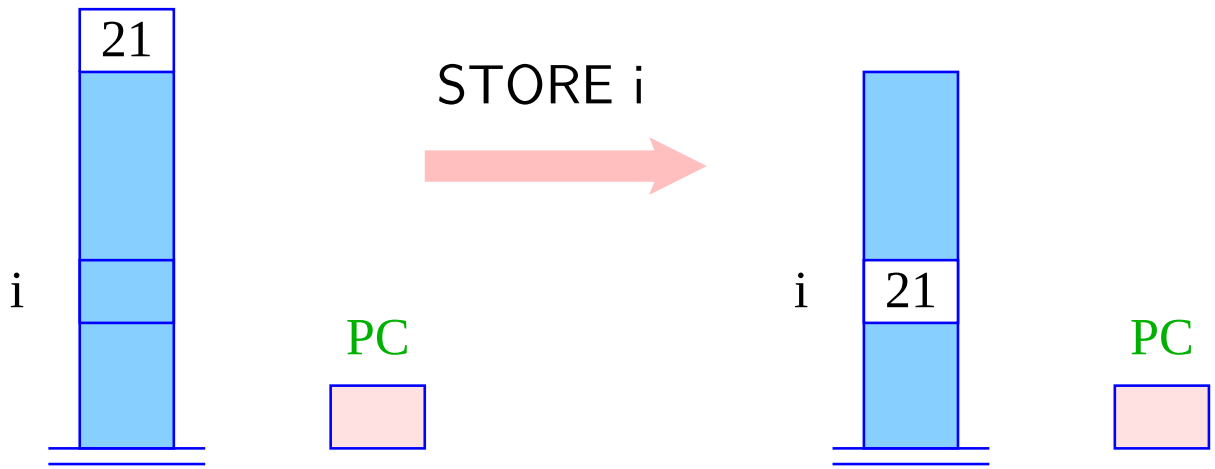
Laden und Speichern

- Konstanten-Lade-Befehle legen einen neuen Wert oben auf dem Stack ab.
- LOAD i legt dagegen den Wert aus der i-ten Zelle oben auf dem Stack ab.
- STORE i speichert den obersten Wert in der i-ten Zelle ab.


CONST 42
42
PC
PC

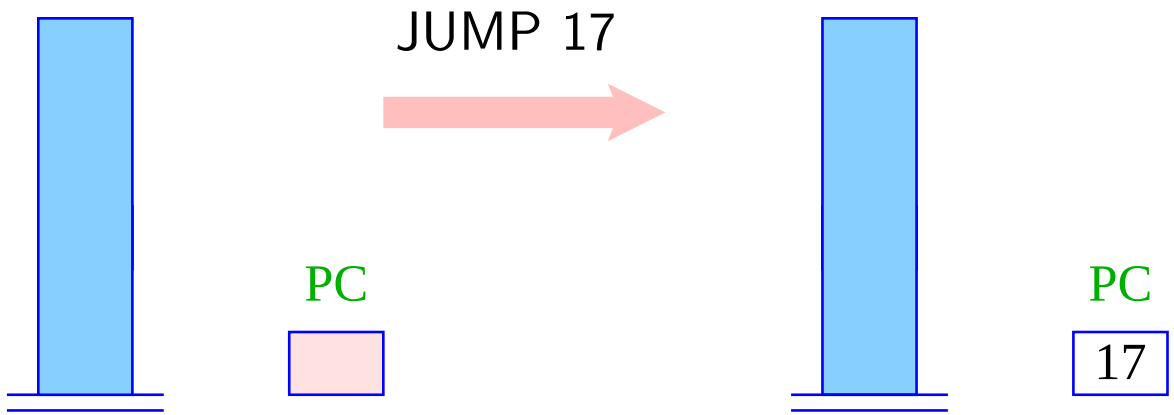

TRUE
tt
PC
PC

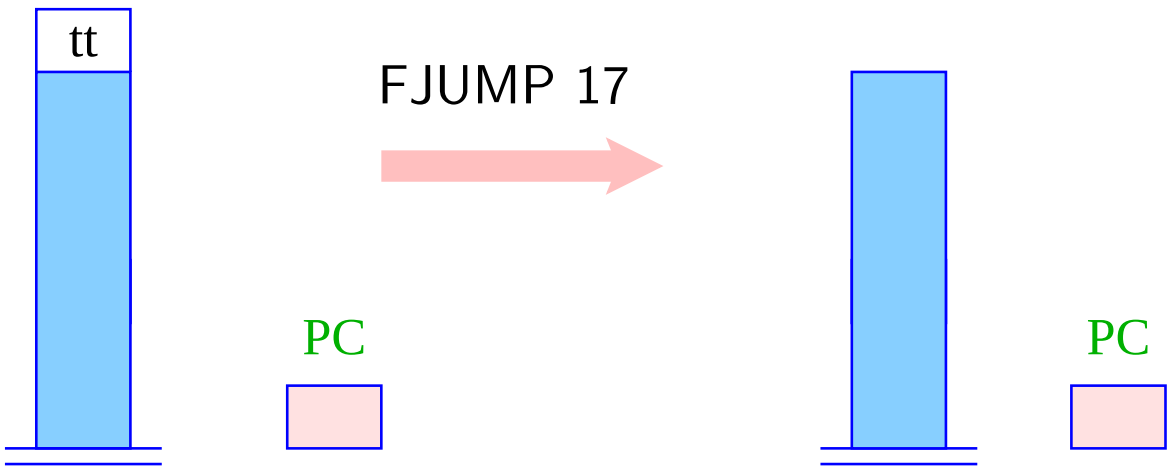

LOAD i
i
21
PC
21
i
21
PC

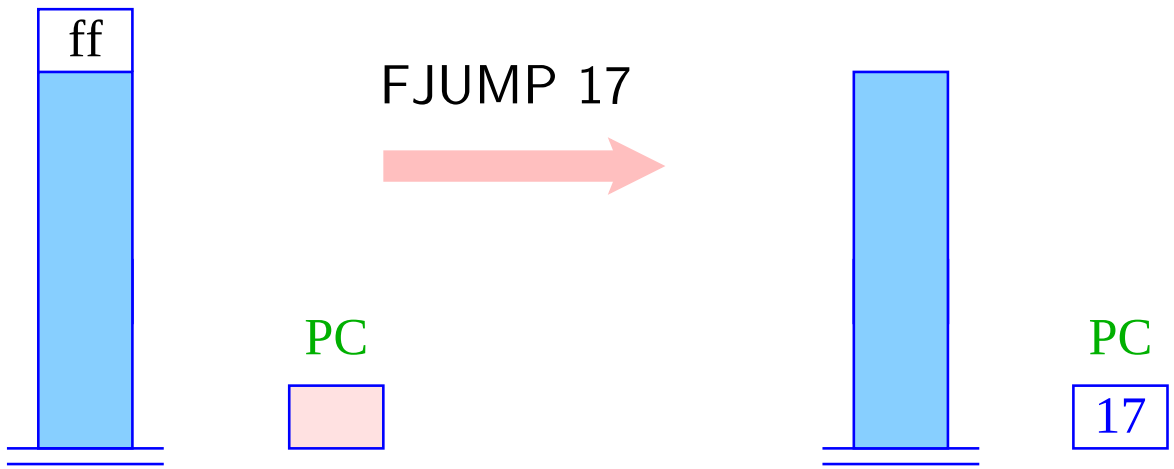

21
STORE i
i
PC
i
21
PC

Sprünge

- Sprünge verändern die Reihenfolge, in der die Befehle abgearbeitet werden, indem sie den PC modifizieren.
- Ein unbedingter Sprung überschreibt einfach den alten Wert des PC mit einem neuen.
- Ein bedingter Sprung tut dies nur, sofern eine geeignete Bedingung erfüllt ist.

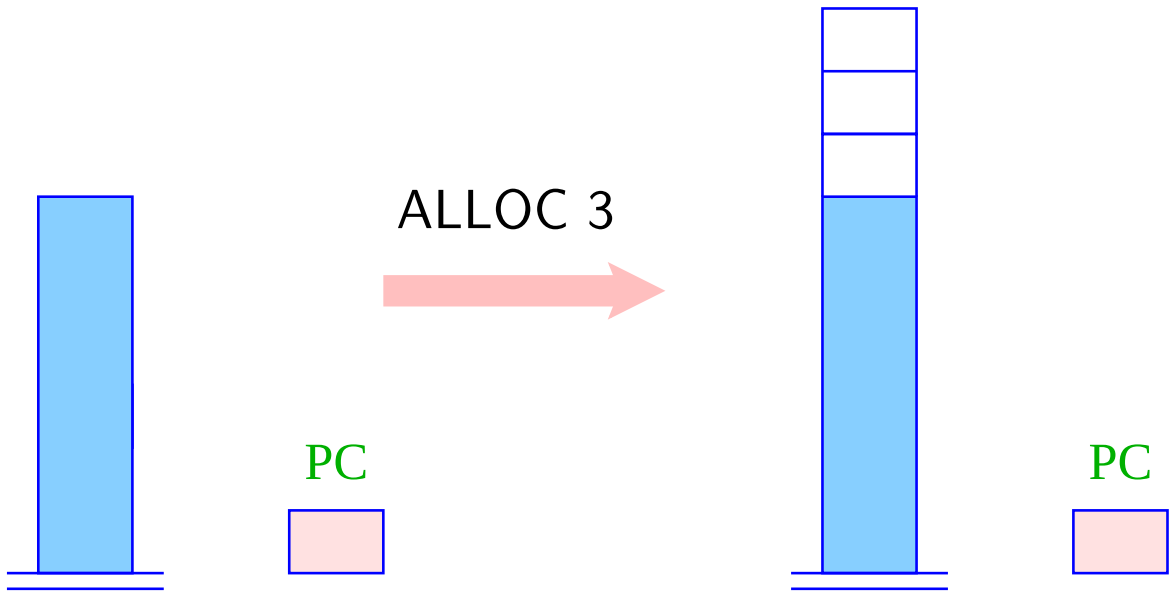

JUMP 17
PC
PC
17


tt
FJUMP 17
PC
PC


ff
FJUMP 17
PC
PC
17

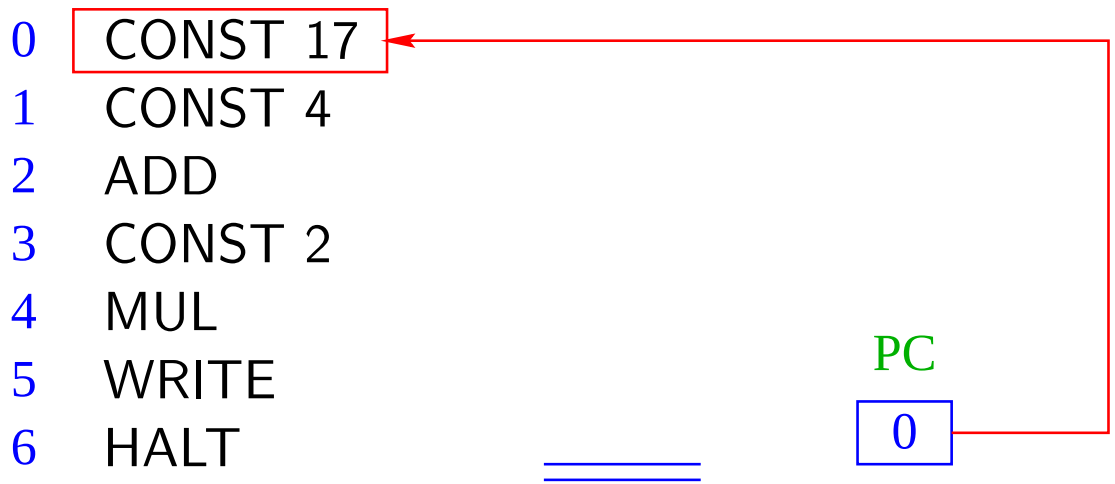
Allokierung von Speicherplatz

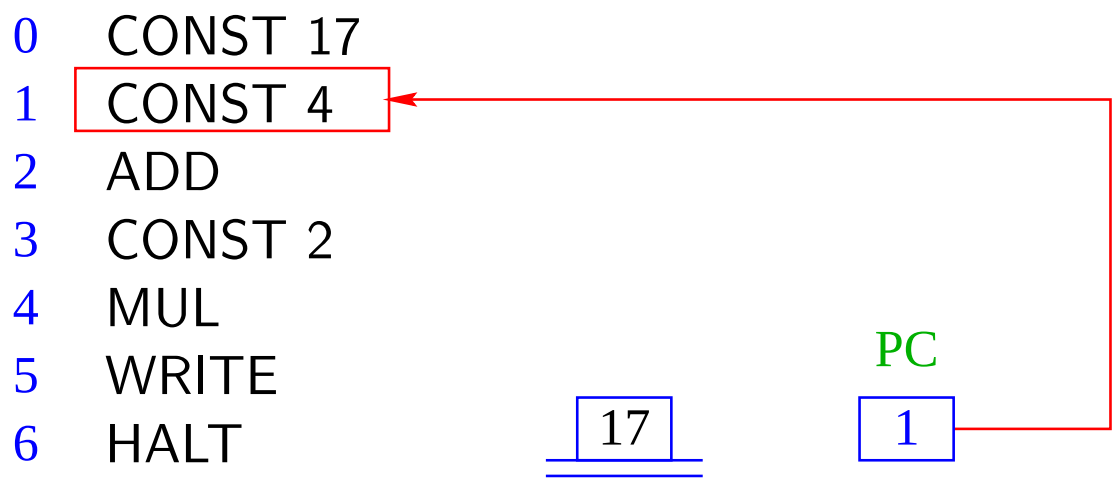
- Wir beabsichtigen, jeder Variablen unseres MiniJava-Programms eine Speicher-Zelle zuzuordnen.
- Um Platz für i Variablen zu schaffen, muss der SP einfach um i erhöht werden.
- Das ist die Aufgabe von ALLOC i.

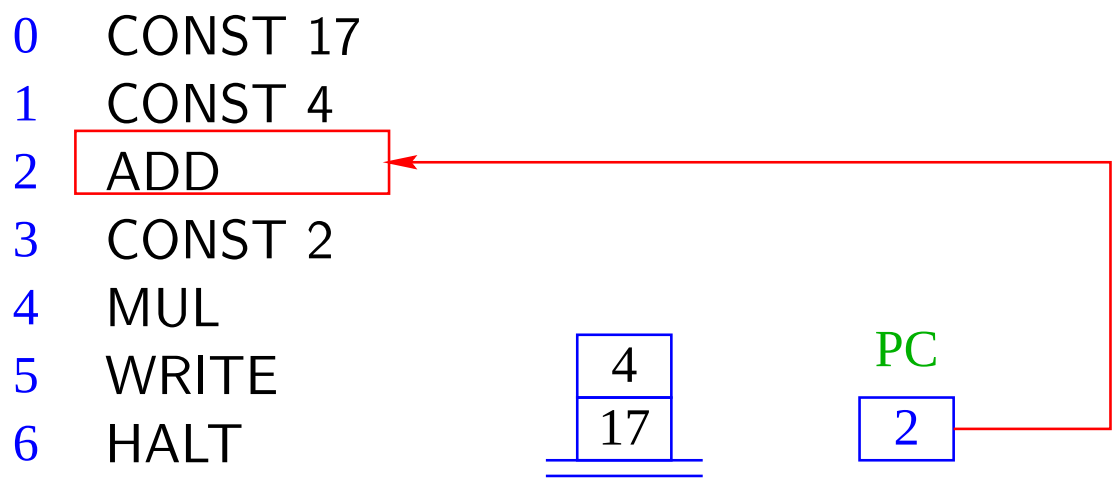

ALLOC 3
PC
PC

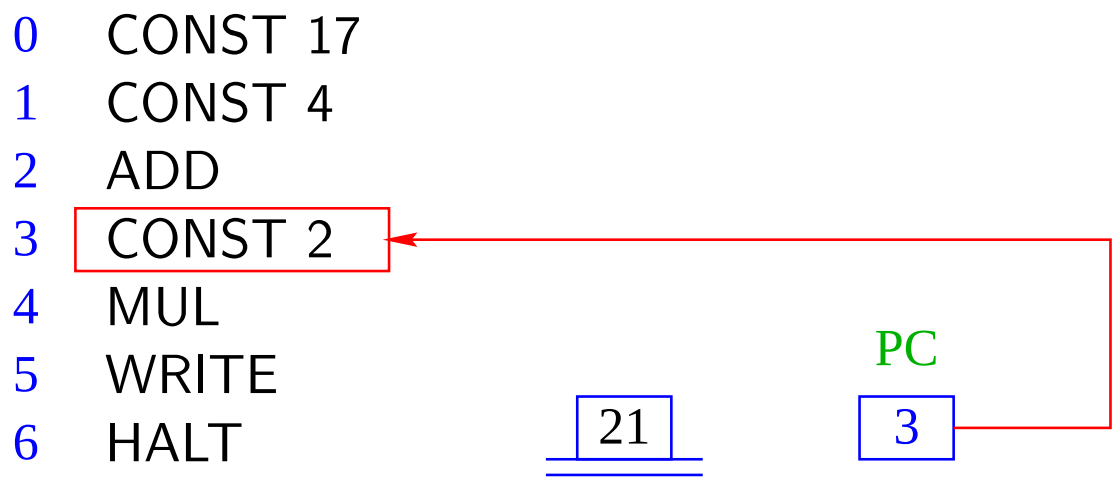
Ein Beispiel-Programm:

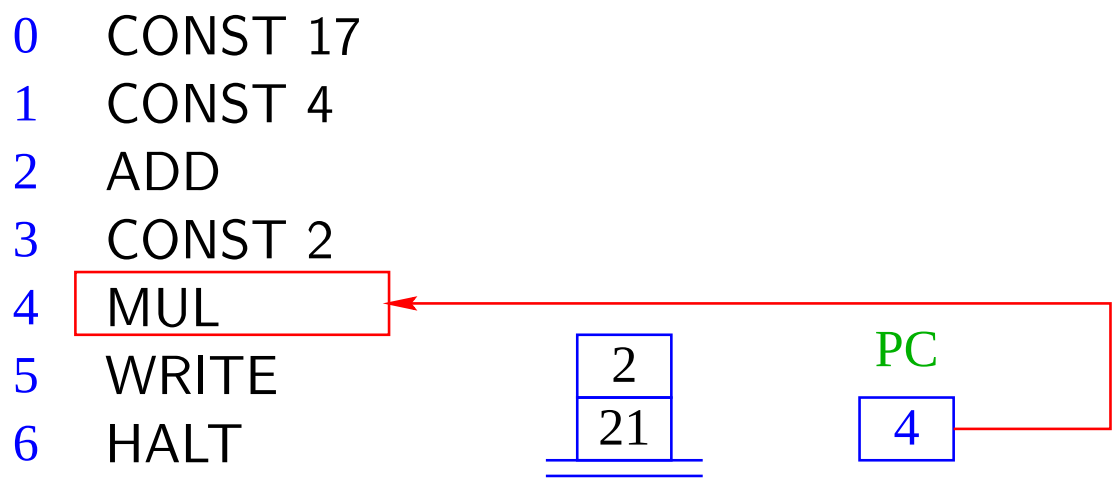
```
CONST 17
CONST 4
ADD
CONST 2
MUL
WRITE
HALT
```

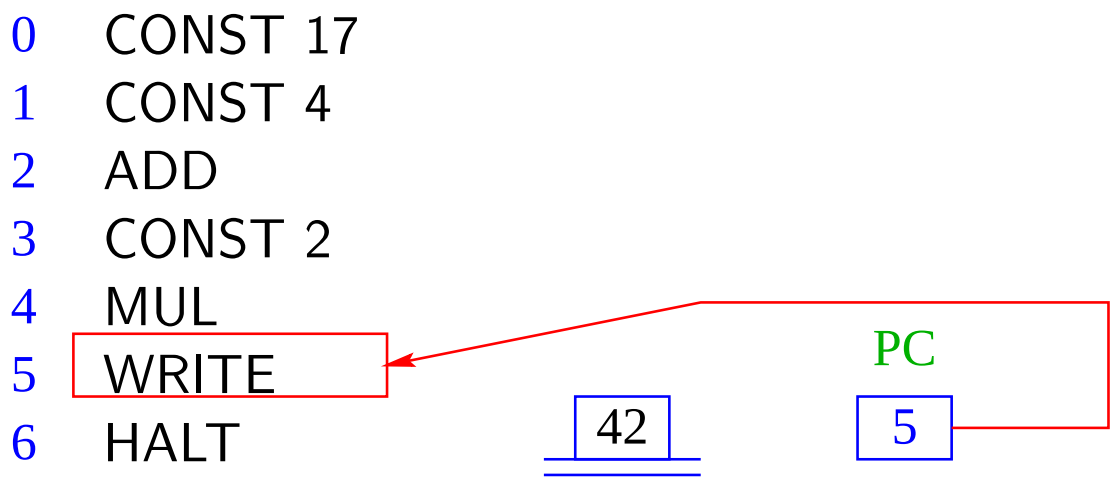

0 CONST 17
1 CONST 4
2 ADD
3 CONST 2
4 MUL
5 WRITE
6 HALT
PC
0

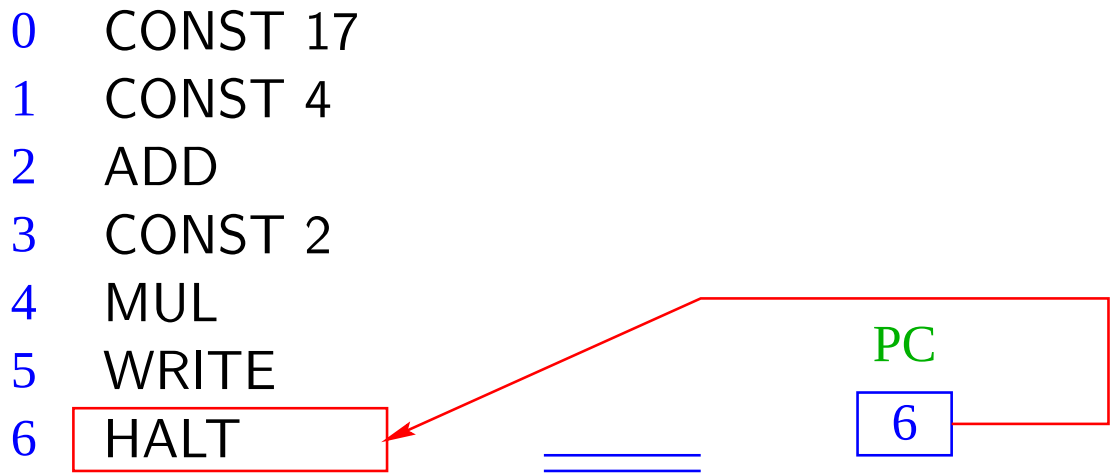

0 CONST 17
1 CONST 4
2 ADD
3 CONST 2
4 MUL
5 WRITE
6 HALT
PC
1
17


0 CONST 17
1 CONST 4
2 ADD
3 CONST 2
4 MUL
5 WRITE
6 HALT
4
17
PC
2


0 CONST 17
1 CONST 4
2 ADD
3 CONST 2
4 MUL
5 WRITE
6 HALT
PC
3
21


0 CONST 17
1 CONST 4
2 ADD
3 CONST 2
4 MUL
5 WRITE
6 HALT
2
21
PC
4


0 CONST 17
1 CONST 4
2 ADD
3 CONST 2
4 MUL
5 WRITE
6 HALT
PC
5
42


0 CONST 17
1 CONST 4
2 ADD
3 CONST 2
4 MUL
5 WRITE
6 HALT
PC
6

Ausführung eines JVM-Programms:

```
PC = 0;
IR = Code[PC];
while (IR != HALT) {
    PC = PC + 1;
    execute(IR);
    IR = Code[PC];
}
```

- IR = Instruction Register, d.h. eine Variable, die den nächsten auszuführenden Befehl enthält.
- `execute(IR)` führt den Befehl in IR aus.
- `Code[PC]` liefert den Befehl, der in der Zelle in Code steht, auf die PC zeigt.