

# Konzepte systemnaher Programmierung

Technische Hochschule Mittelhessen

Andre Rein

— Unterprogrammaufrufe und  
Rücksprünge —

# Unterprogramm-Aufrufe und Rücksprünge

Beispiel eines Unterprogrammaufrufs in C :

```
int main(int argc, char *argv[]) {  
    printf("Hallo"); ①  
    printf("Welt\n"); ②  
    return 0;  
}
```

- 1 Springe zur Funktion `printf()`, führe alle Instruktionen von `printf()` aus und kehre **anschließend** wieder zur `main`-Funktion zurück um
- 2 die zweite `printf()`-Funktion auszuführen.



Wir haben die 2 bedingten Sprünge ( `brf` , `brt` ) und einen unbedingten Sprung ( `jmp` ) besprochen. Bei der Verwendung geht allerdings die Information verloren, **von welcher Position** aus wir gesprungen sind

# Unterprogramm-Aufrufe und Rücksprünge

Einführung neuer Instruktionen für Unterprogrammaufrufe:

- `call <n> → ... -> ... ra` – legt eine **Rücksprungadresse** `ra` auf den Stack und setzt den Programmzähler auf `<n>` (`PC=n`)
- `ret → ... ra -> ...` – nimmt die Rücksprungadresse `ra` vom Stack und setzt den Programmzähler auf den Wert von `ra` (`PC=ra`)



Insgesamt gibt es 4 Varianten von Funktionsaufrufen: Mit und ohne **Funktionsparameter** jeweils mit und ohne **Rückgabewerte**.

# Funktionsaufrufe ohne Argumente und Rückgabewerte

- Notation: **Caller** Aufrufende Funktion
- Notation: **Callee** Aufgerufene Funktion
- Funktionslabel: Ähnlich zum normalen Label *eine Adresse im Programm*

Caller

```
call Funktionslabel; ①  
add; // beliebige nachfolgende  
// Instruktion ②
```

① Legt die Adresse  
von ② auf den  
Stack

Callee

```
Funktionslabel:  
    asf <numLocals> // numLocals bezeichnet die Anzahl der  
                      // benötigten lokalen Variablen der Funktion  
    <body>           // unbestimmte Anzahl an Instruktionen  
    rsf              // zurücksetzen des aktuellen Stackframes ③  
    ret              // zurückkehren zum Aufrufer (Caller!) ④
```

③ Auf dem Stack liegt nun die Rücksprungadresse *ra*  
(②) als oberster Wert!

④ ret nimmt die Rücksprungadresse und springt  
zurück in den Caller zur Position ②.

# Funktionsaufruf: Beispiel

```
pushc 12
rdint
mul
wrint
call new_line
halt
new_line:
    asf 0
    pushc 10
    wrchr
    rsf
    ret
```

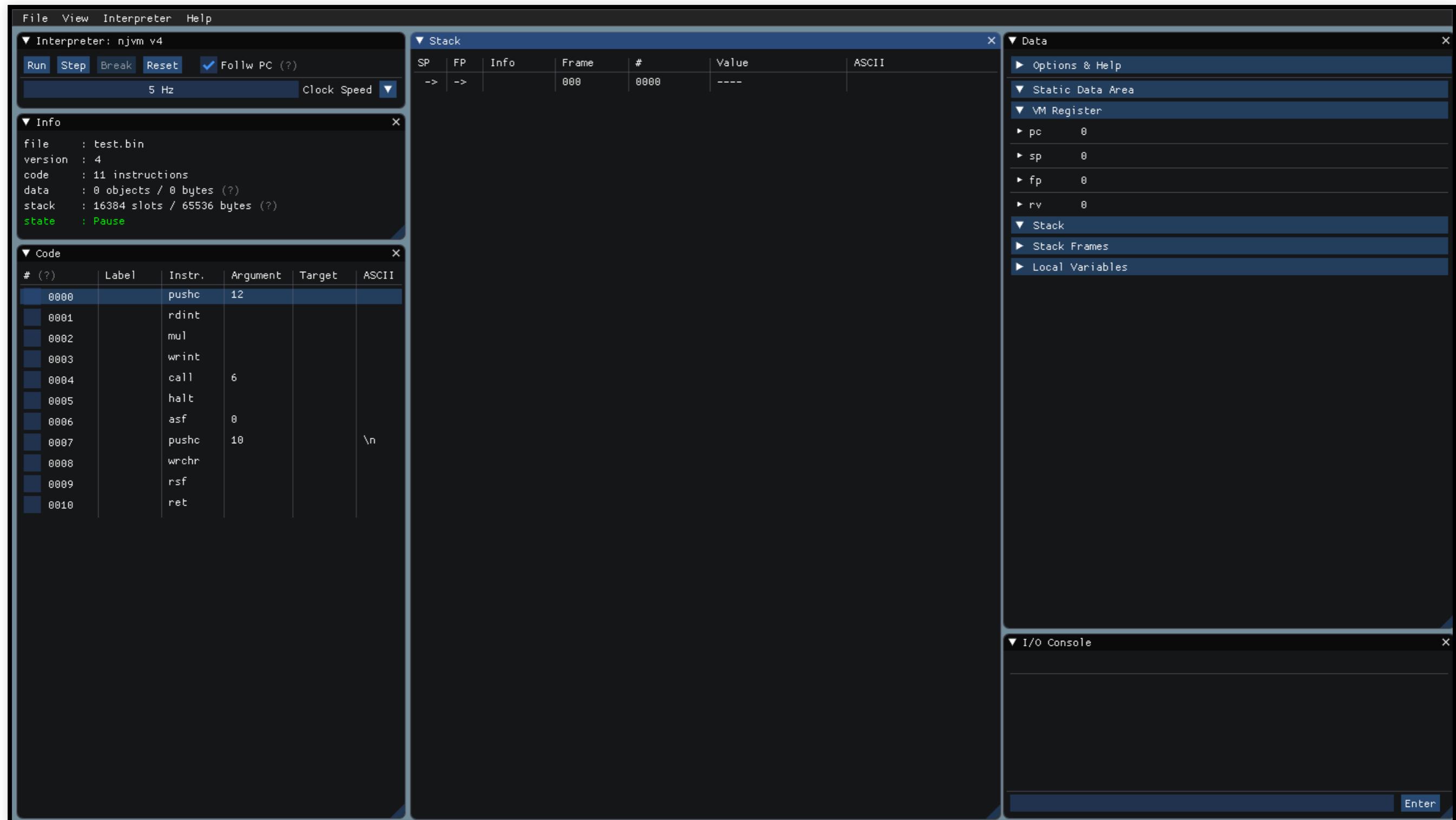
Frage: Was macht diese Funktion?

Einlesen einer Zahl, Multiplikation mit 12, Ausgabe der Zahl, Ausgabe \n

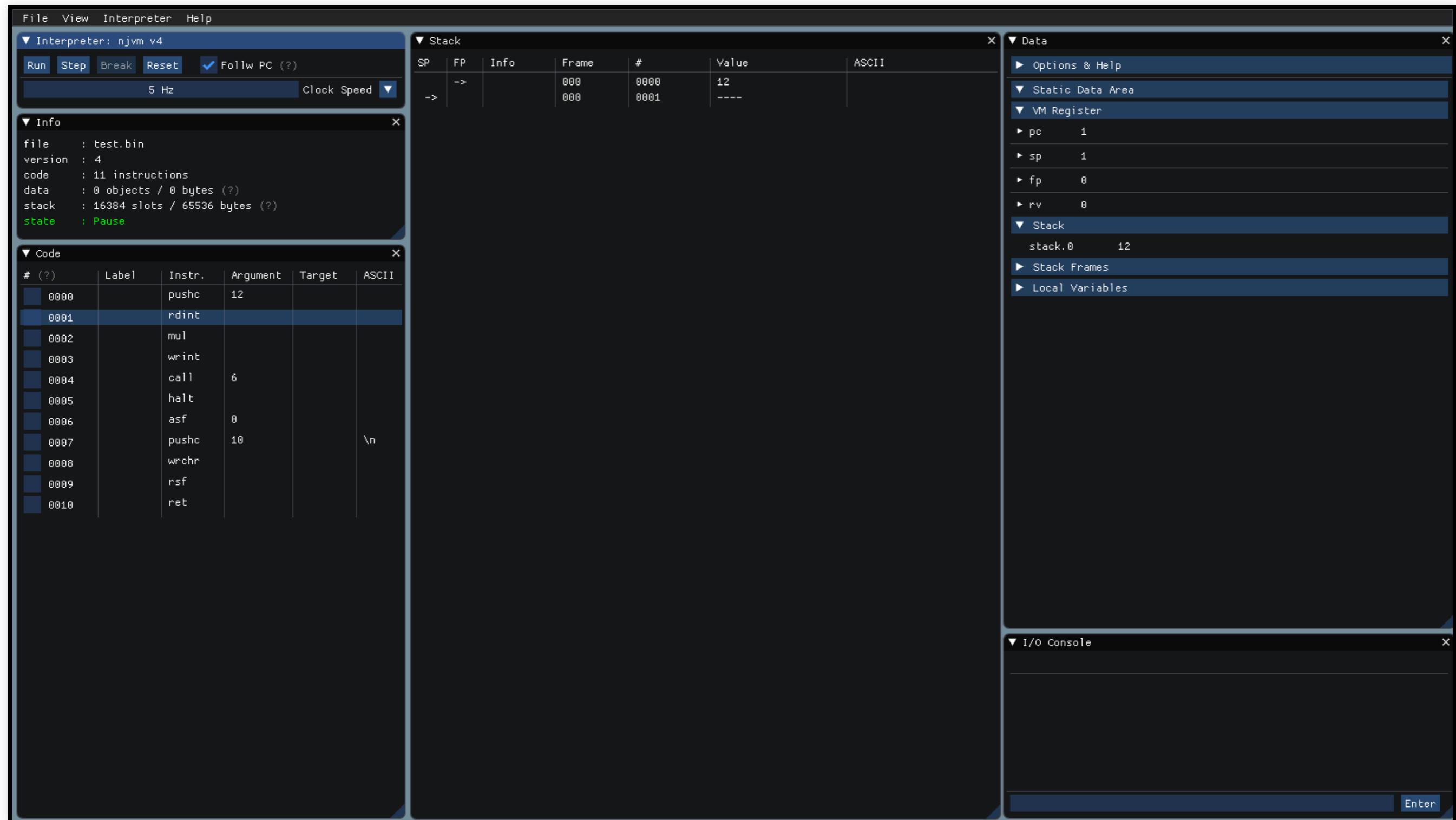
# Ninja Debugger

[https://homepages.thm.de/~arin07/lectures/ksp/njdb/?font\\_size=23&interp=4](https://homepages.thm.de/~arin07/lectures/ksp/njdb/?font_size=23&interp=4)

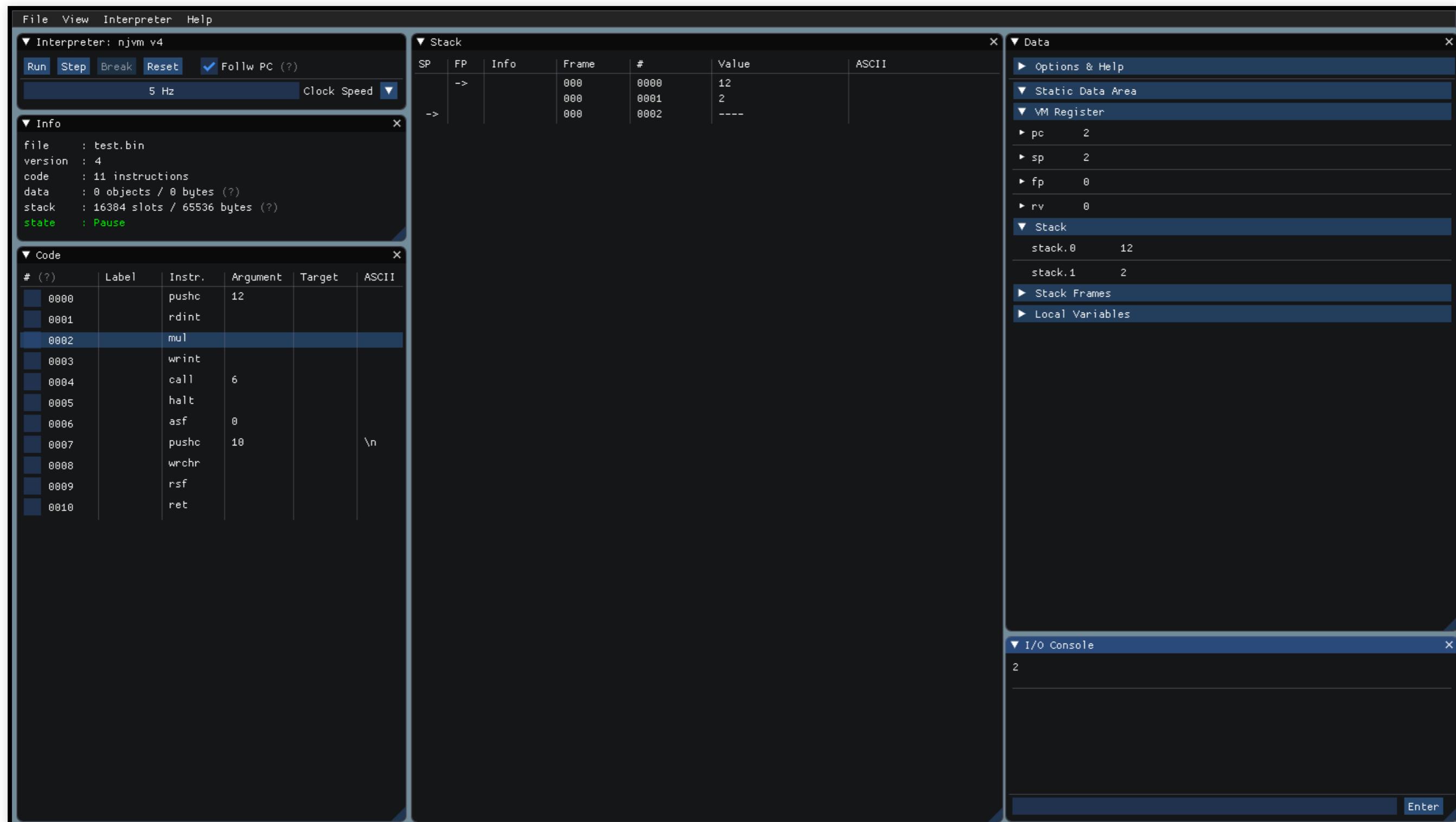
# Funktionsaufruf: Beispiel



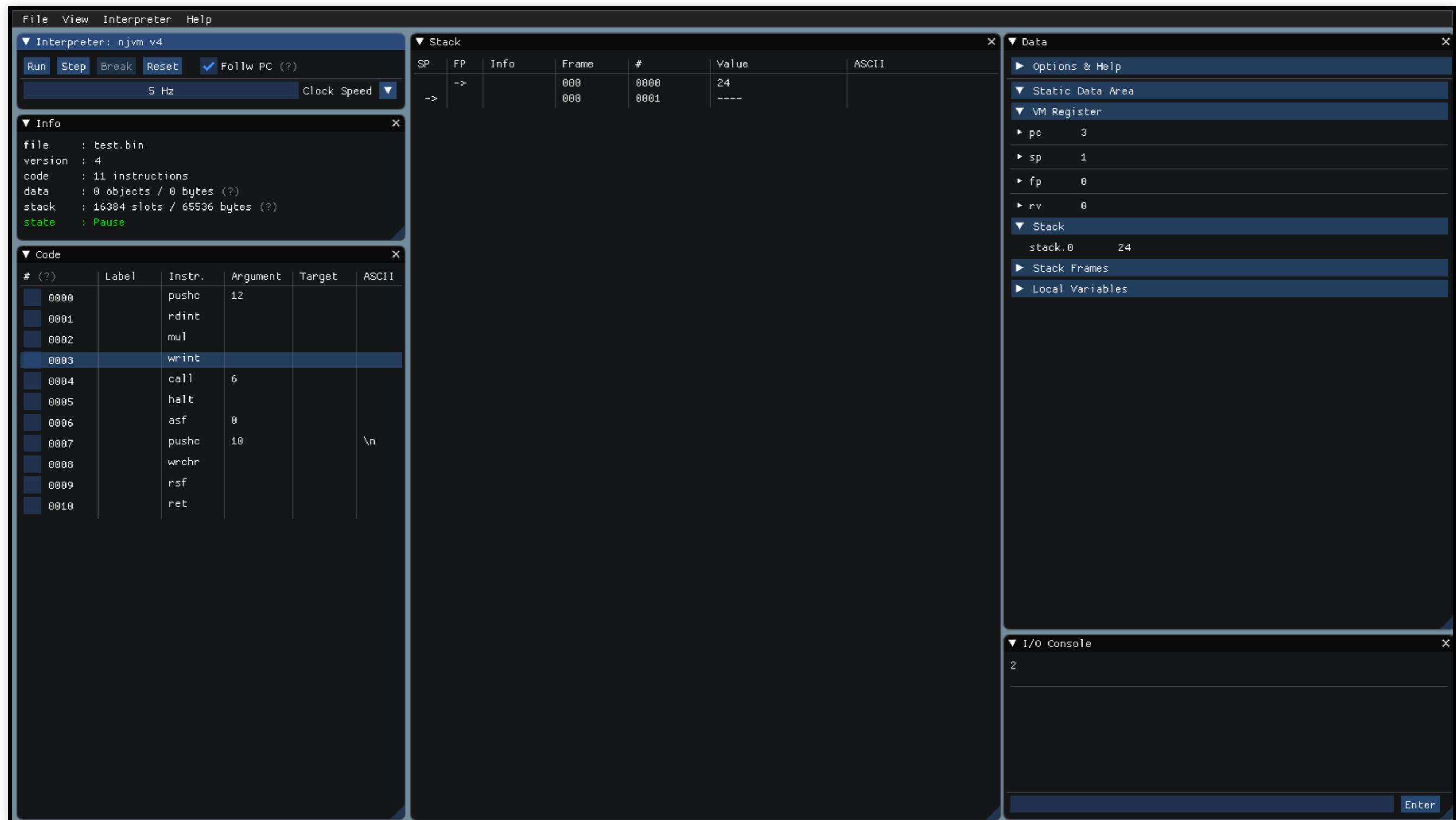
# Funktionsaufruf: Beispiel



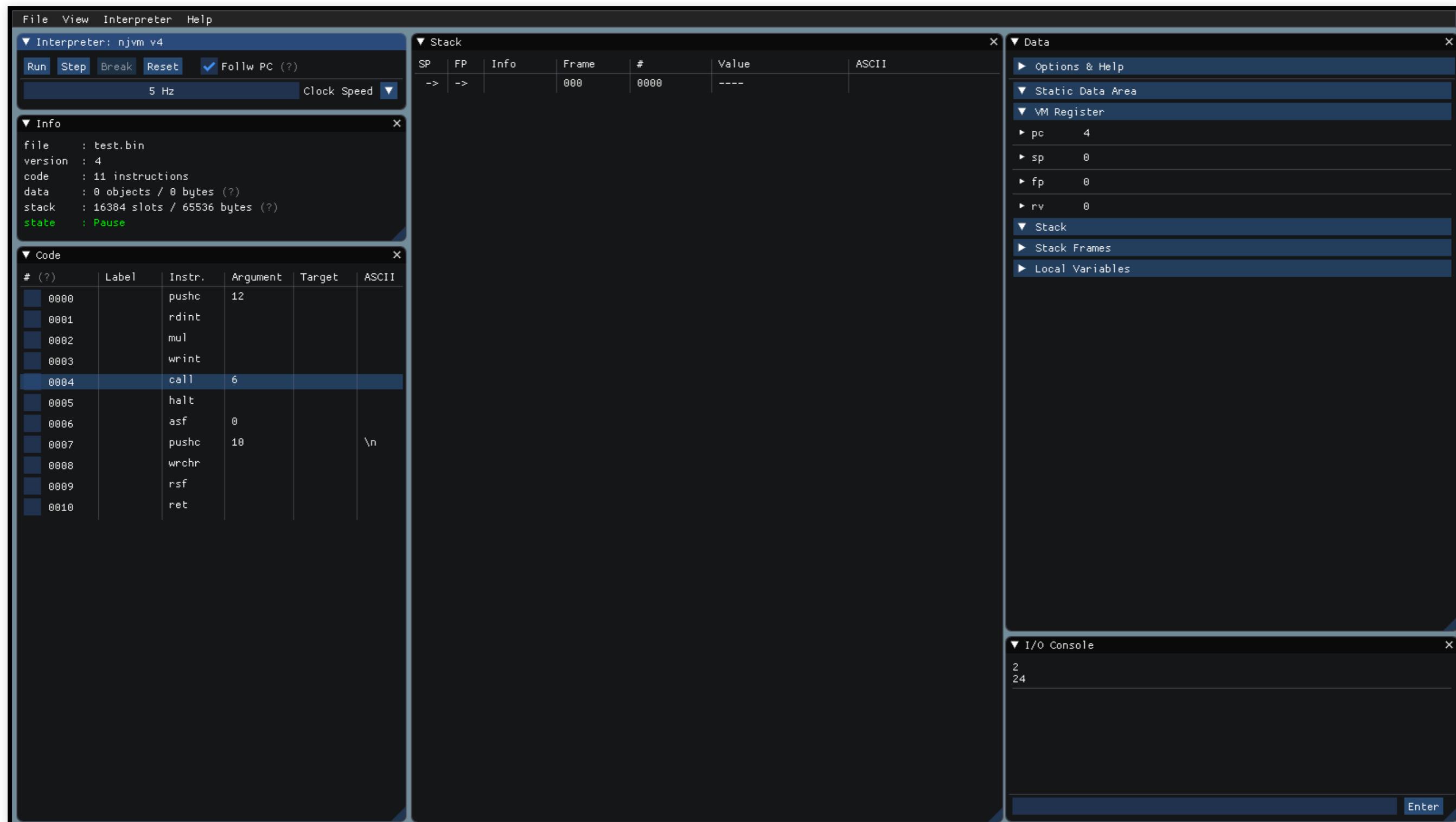
# Funktionsaufruf: Beispiel



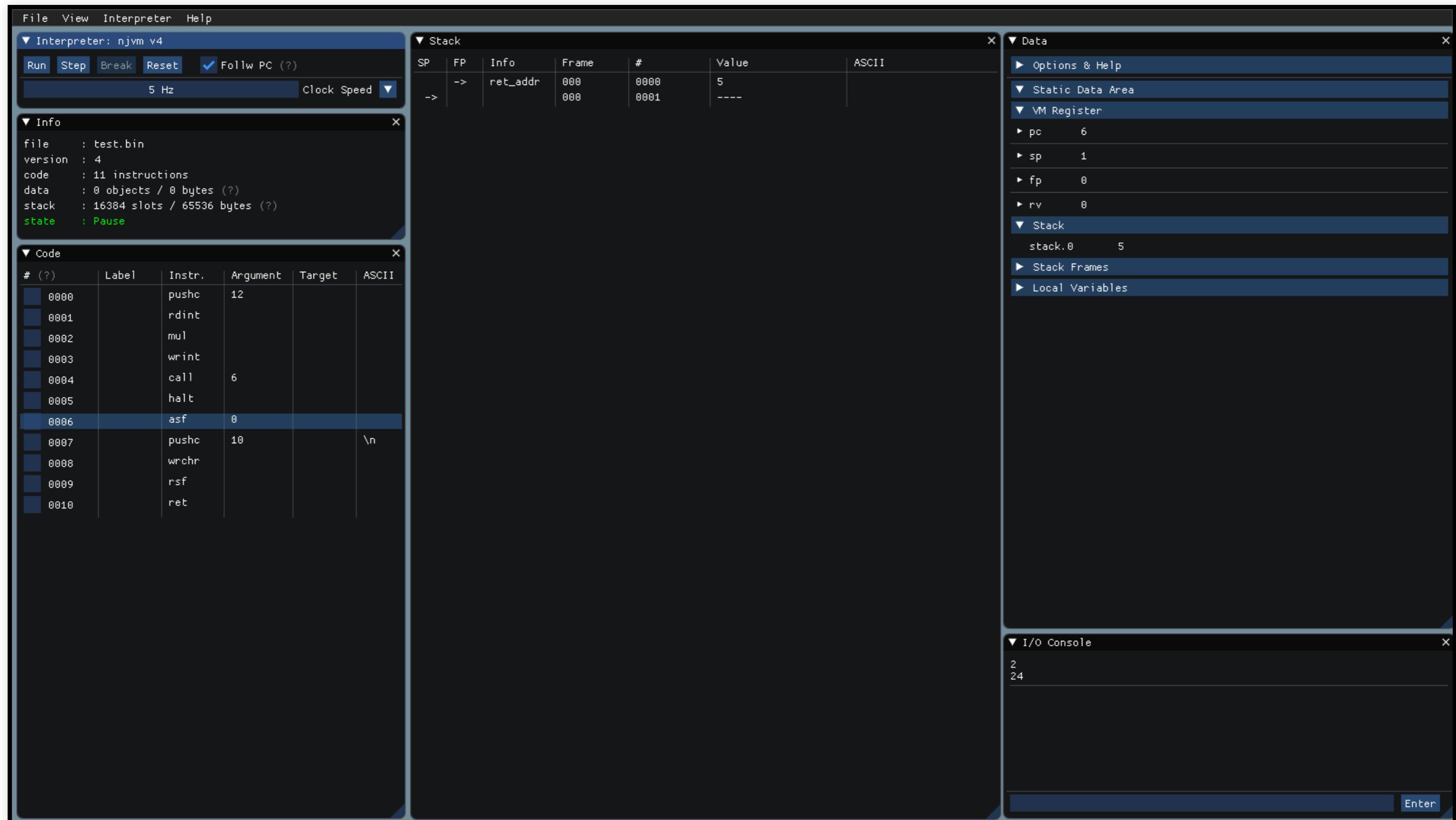
# Funktionsaufruf: Beispiel



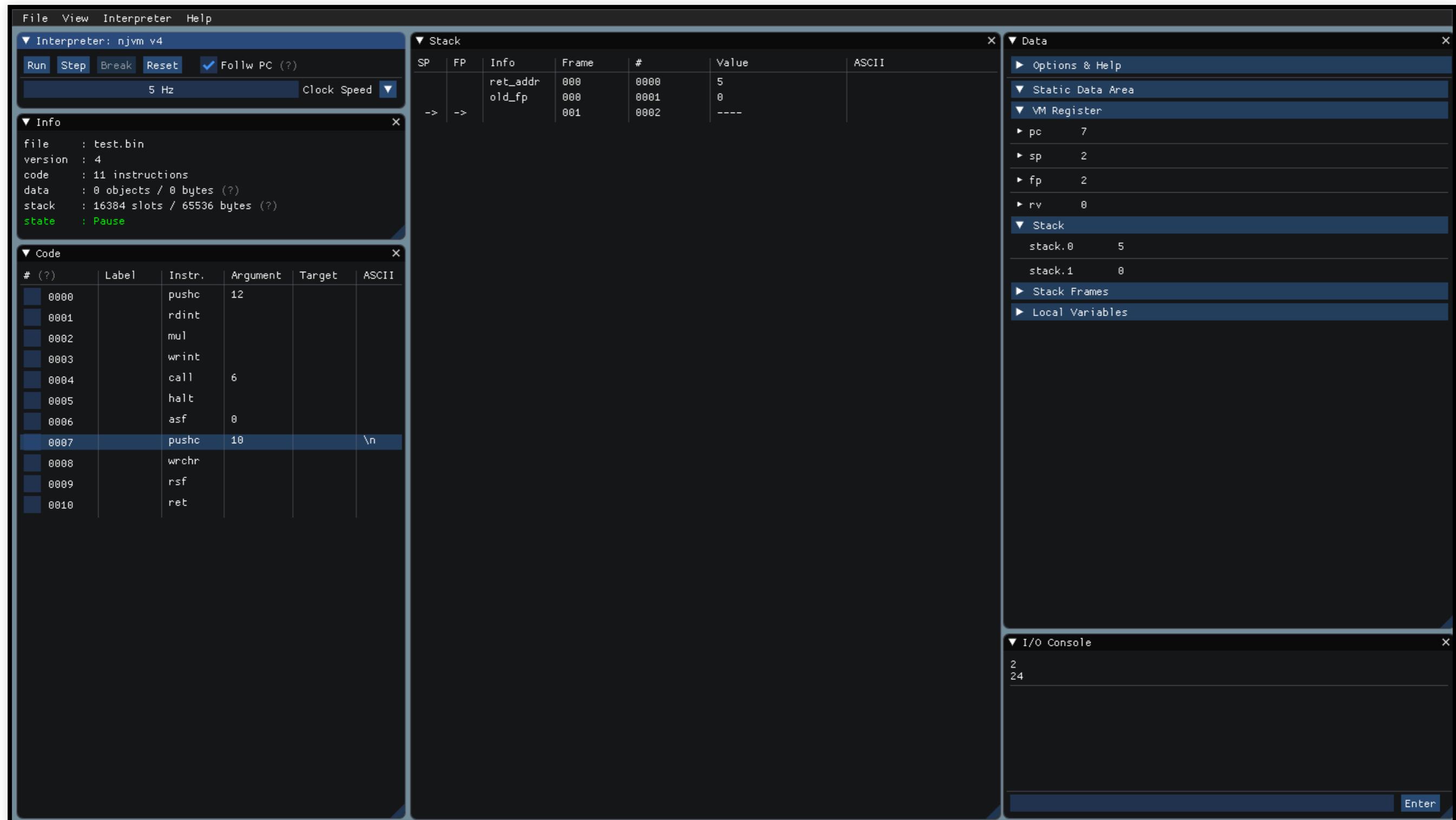
# Funktionsaufruf: Beispiel



# Funktionsaufruf: Beispiel



# Funktionsaufruf: Beispiel

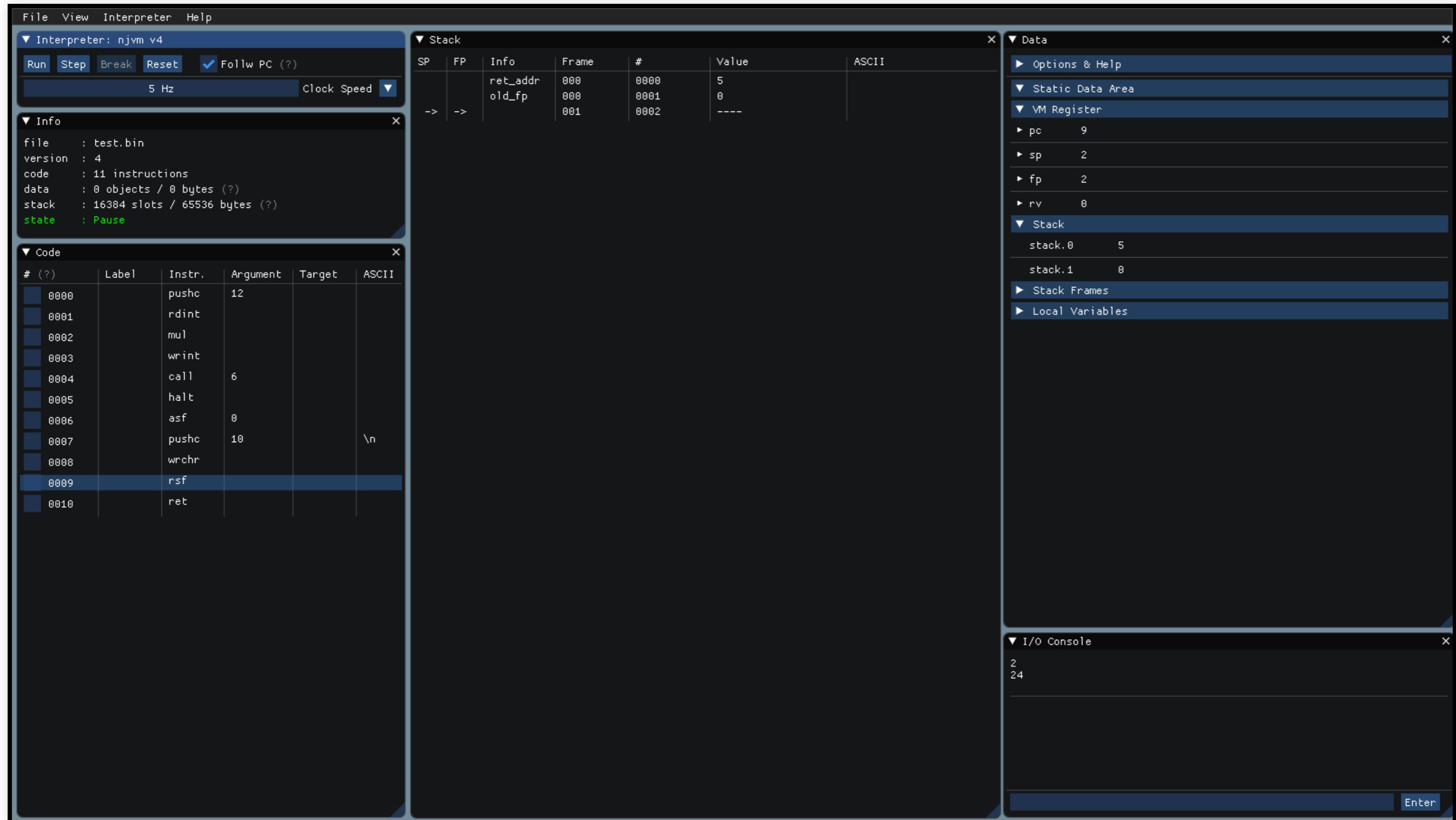


# Funktionsaufruf: Beispiel

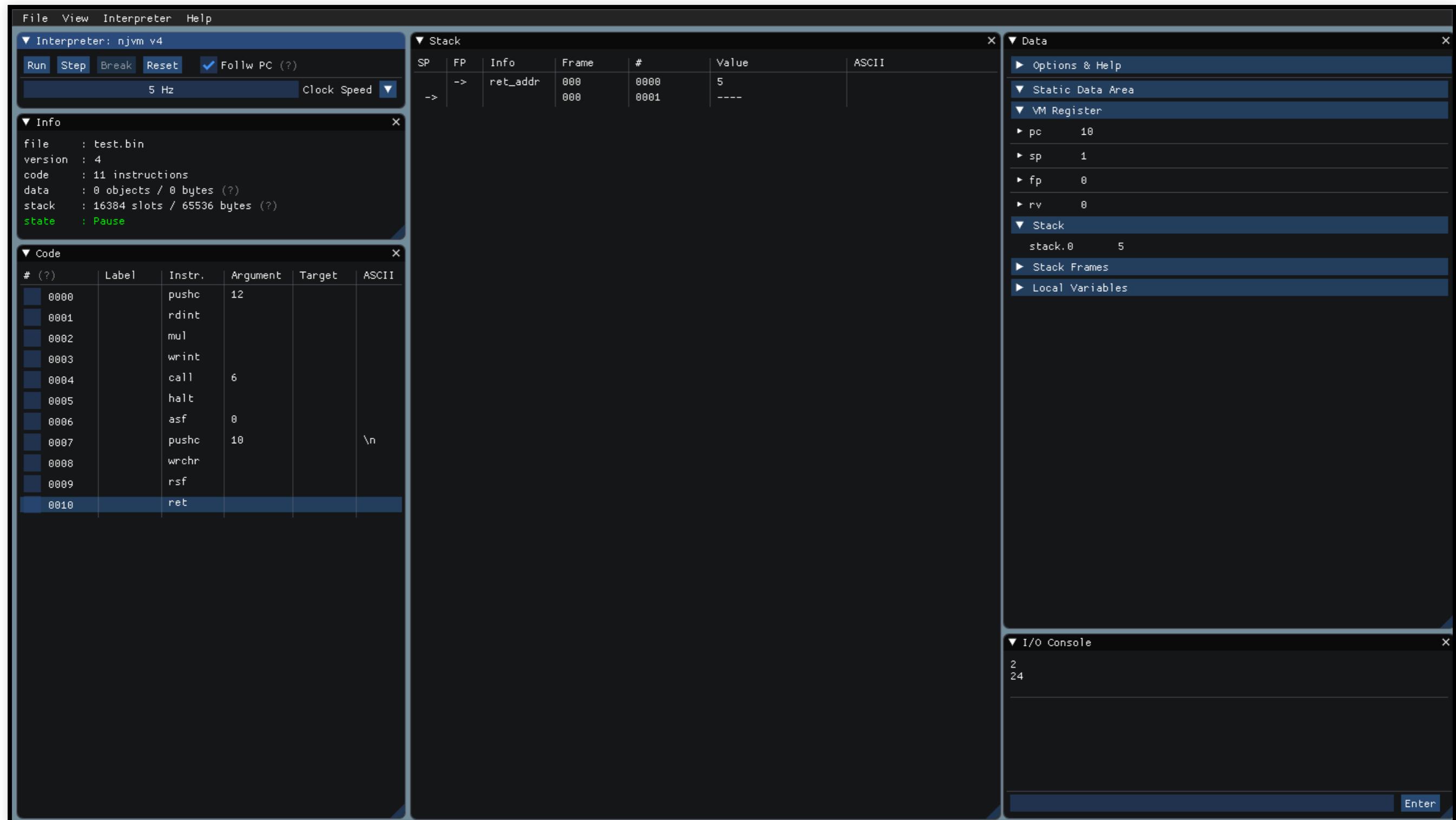
The screenshot shows the njvm v4 debugger interface with several windows open:

- Interpreter: njvm v4**: Top-left window with controls for Run, Step, Break, Reset, and Follow PC. It also shows the clock speed as 5 Hz.
- Info**: Shows file: test.bin, version: 4, code: 11 instructions, data: 0 objects / 0 bytes (?), stack: 16384 slots / 65536 bytes (?), and state: Pause.
- Code**: A table showing assembly instructions. The instruction at address 0008 is highlighted: wrchr (Target: \n).
- Stack**: A table showing the stack structure. The stack grows downwards, with the top of the stack at address 0000. The current frame is 001, containing the return address (ret\_addr) at 0000 with value 5, and the old fp at 0001 with value 0. The previous frame (Frame 000) contains the new fp at 0002 with value 10, and the previous frame (Frame 000) contains the old fp at 0003 with value ----.
- Data**: A tree view of the VM's memory space. It includes sections for Options & Help, Static Data Area, VM Register, Stack, Stack Frames, Local Variables, and I/O Console.
- I/O Console**: Bottom-right window showing the output 2 24.

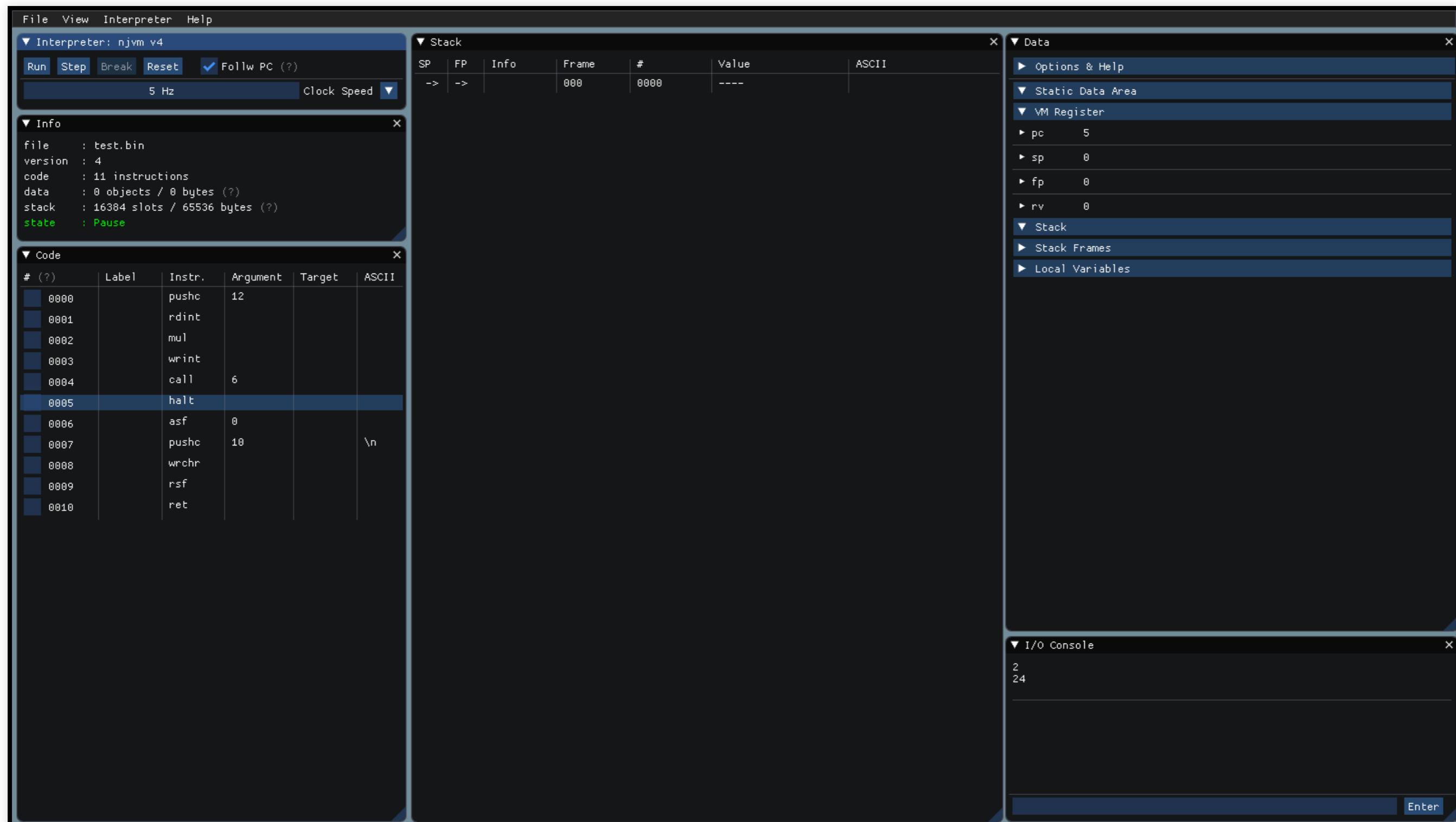
# Funktionsaufruf: Beispiel



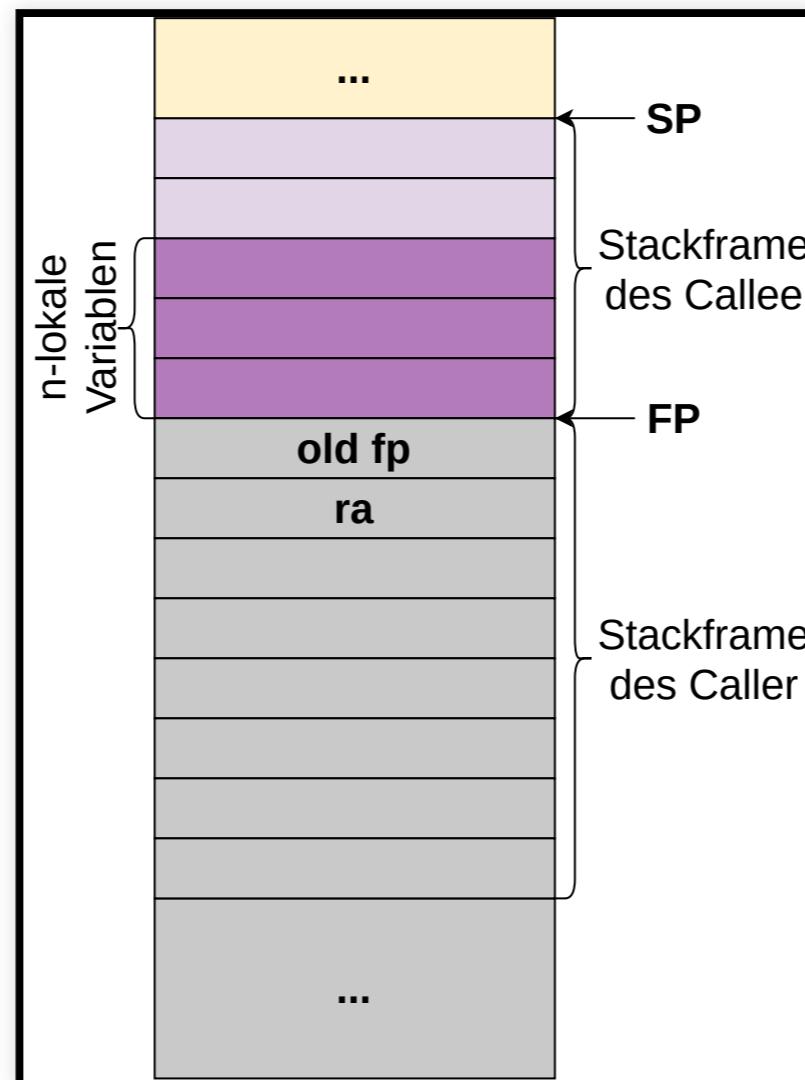
# Funktionsaufruf: Beispiel



# Funktionsaufruf: Beispiel



# Stack bei Funktionsaufruf ohne Argumente



# Funktionsaufruf mit Argumenten, ohne Rückgabewert

Problem: Wie können Argumente an eine Funktion übergeben werden?

Grundsätzlich gibt es verschiedene Varianten, wie eine Argumentübergabe erfolgen kann. Im Sprachdesign von Ninja wurde sich dafür entschieden, dass die Argumentübergabe vollständig über den Stack erfolgt. Diese Designentscheidung nennt man die **Aufrufkonvention** (engl. **Calling Convention**).

Beispiele aus der realen Welt:

- X86 32-Bit Architektur implementiert ebenfalls die Argumentübergabe mittels Stack.
- X86 64-Bit Architektur implementiert eine Mischform, die ersten 6 Argumente werden mittels Registern übergeben und die restlichen auf dem Stack.



Wenn in einer Sprache die Anzahl der übergebenen Parameter nicht begrenzt ist, bzw. die Anzahl der potentiell zu übergebenen Argumente größer ist, als Register zur Verfügung stehen, ist es sinnvoll den Stack für die *restlichen* Werte zu verwenden.

# Funktionsaufruf mit Argumenten, ohne Rückgabewert

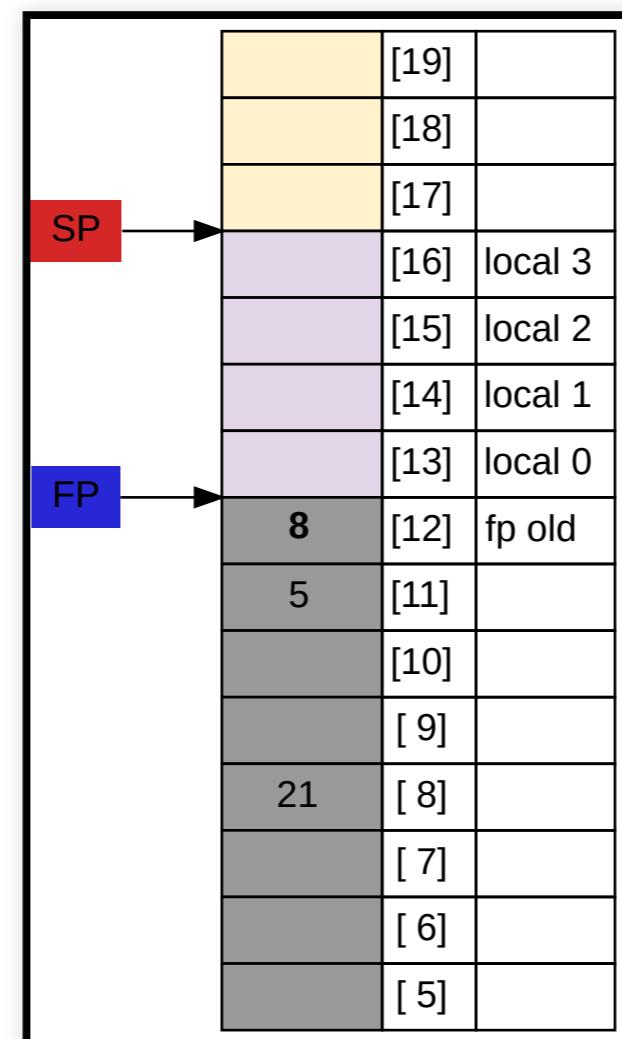
Das Problem der Argumentübergabe mittels Stack ist die **Adressierung** der einzelnen Argumente.

- Unter der Annahme, dass je nach Funktion eine **unterschiedliche** Anzahl an Argumenten übergeben wird, erlaubt hierbei **keine feste** oder gar **absolute** Zuordnung zwischen Argument und Adresse.
- Den Trick den man verwendet ist:
  - Man sucht sich einen **festen Bezugspunkt**, dessen Adresse während der Dauer des Funktionsaufrufs nicht verändert wird, und
  - adressiert die Argumente **relativ** zu diesem Bezugspunkt.

# Funktionsaufruf mit Argumenten, ohne Rückgabewert

*Wir haben dies schon einmal gemacht, erinnern Sie sich wobei?*

Wir adressieren die **lokalen Variablen** als positiven **Offset** zum Framepointer ( FP ).



# Funktionsaufruf mit Argumenten, ohne Rückgabewert

Angenommen wir haben  $n$ -Argumente, dann nummerieren wir diese durch. Das 1. Argument (links in der Liste), ist das 0-te Argument und das  $n$ -te Argument (rechts in der Liste) hat die Nummer  $n-1$ .

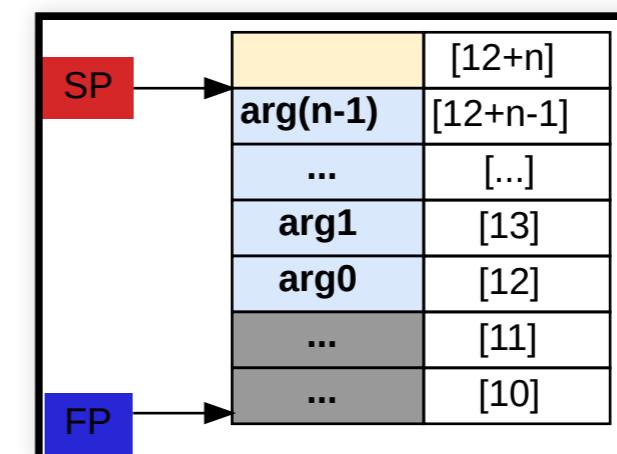
Beispiel:

$f(arg_0, arg_1, arg_2, \dots, arg_{n-1}) \rightarrow n$ -Argumente

Caller:

```
// FP = 10, SP = 12
push arg0
push arg1
...
push arg(n-1)
// FP = 10, SP = 12+n
call Funktionslabel
// FP = 10, SP = 12+n
```

Stack push  $arg(n-1)$



Der Stack sieht vor und nach dem Funktionsaufruf `call Funktionslabel` identisch aus!

# drop <n>-Instruktion

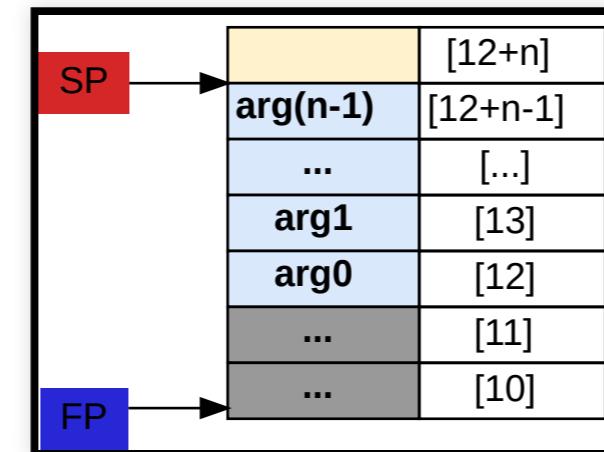
Nach dem Funktionsaufruf (call Funktionslabel) werden die Argumente des Funktionsaufrufs allerdings nicht länger benötigt und werden mit der Instruktion drop <n> verworfen/gelöscht.

drop <n> → ... a<sub>0</sub> a<sub>1</sub>...a<sub>(n-1)</sub> -> ...

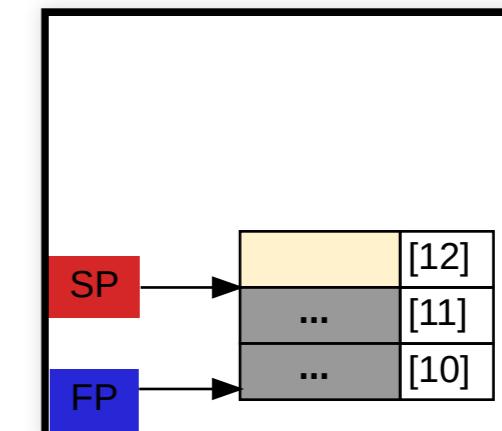
Caller:

```
// FP = 10, SP = 12
push arg0
push arg1
...
push arg(n-1)
// FP = 10, SP = 12+n
call Funktionslabel
// FP = 10, SP = 12+n
drop n //löscht n-Einträge v.Stack
// FP = 10, SP = 12
```

Stack push arg(n-1)



Stack (drop n)



Nach dem Funktionsaufruf (call Funktionslabel) werden die Argumente nicht länger benötigt und mit drop <n> verworfen.

# Funktionsaufruf mit Argumenten, ohne Rückgabewert

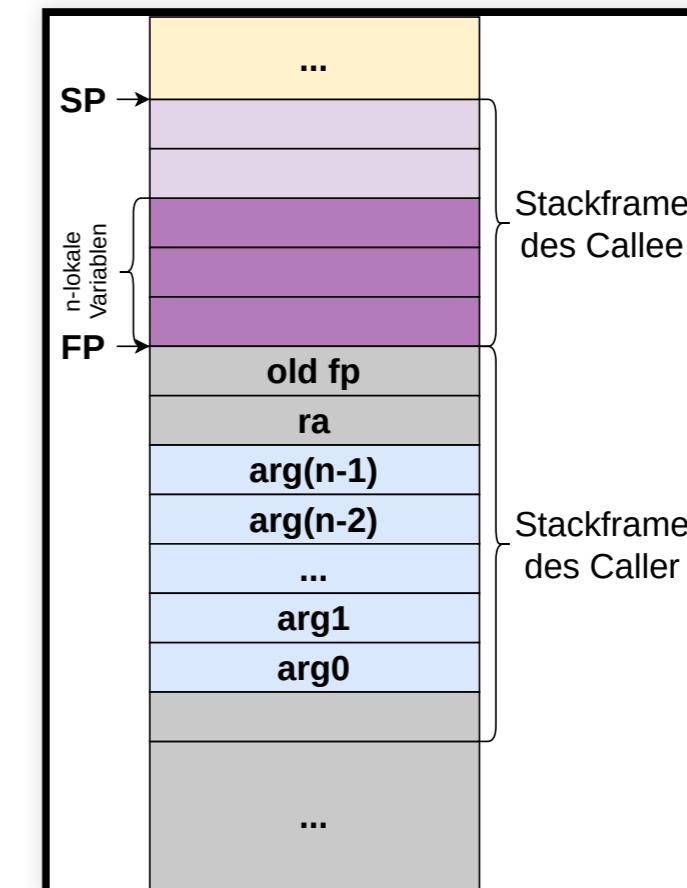
Callee

```
Funktionslabel:  
  asf <numLocals> // numLocals bezeichnet die Anzahl der  
                    // benötigten lokalen Variablen der Funktion  
  <body>           // unbestimmte Anzahl an Instruktionen  
  rsf              // zurücksetzen des aktuellen Stackframes  
  ret              // zurückkehren zum Aufrufer (Caller!)
```

Der Zugriff auf die Argumente erfolgt über einen negativen Offset an `pushl <offset>` und `popl <offset>` als Immediate Wert.

Die Berechnung des negativen Offsets für ein Argument  $i$  erfolgt durch `stack[FP - 2 - n + i]`. Da `popl` und `pushl` bereits eine relative Adressierung mittels Framepointer verwenden, ist für uns nur die Berechnung des Offset-Wertes ( $-2 - n + i$ ) von Interesse, da dieser Wert das negative Offset *relativ* zum aktuellen FP angibt.

Stack



# Anmerkung: Offset-Berechnung



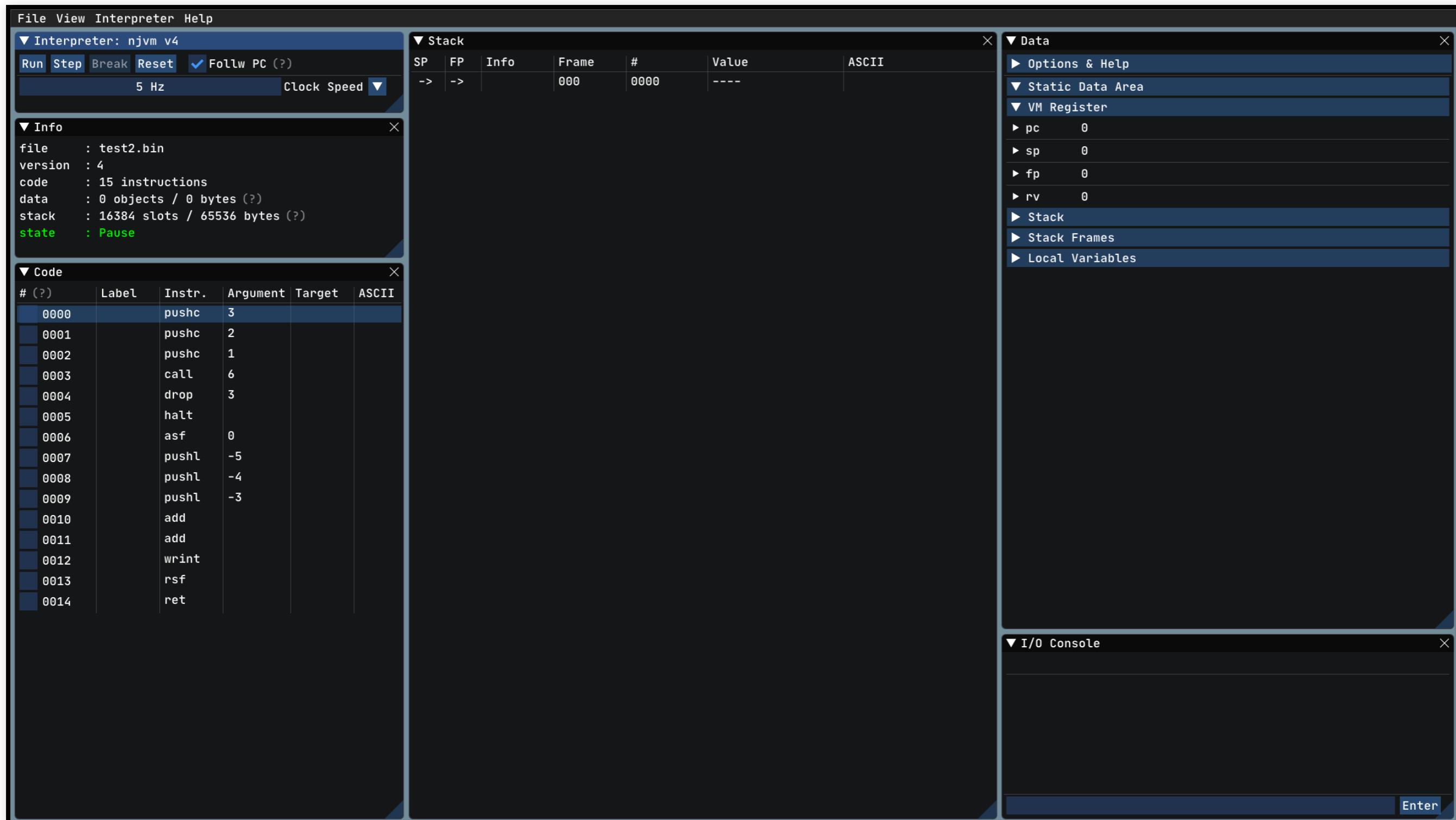
Die Berechnung des korrekten (negativen) Offset-Wertes an `pushl` und `popl` wird vom Compiler berechnet. Wir müssen uns um die Anpassung also nur dann kümmern, wenn wir direkt Ninja-Assembler Code schreiben und diesen mit `nja` in Bytecode umwandeln.

# Beispiel: Funktionsaufruf mit 3 Argumenten

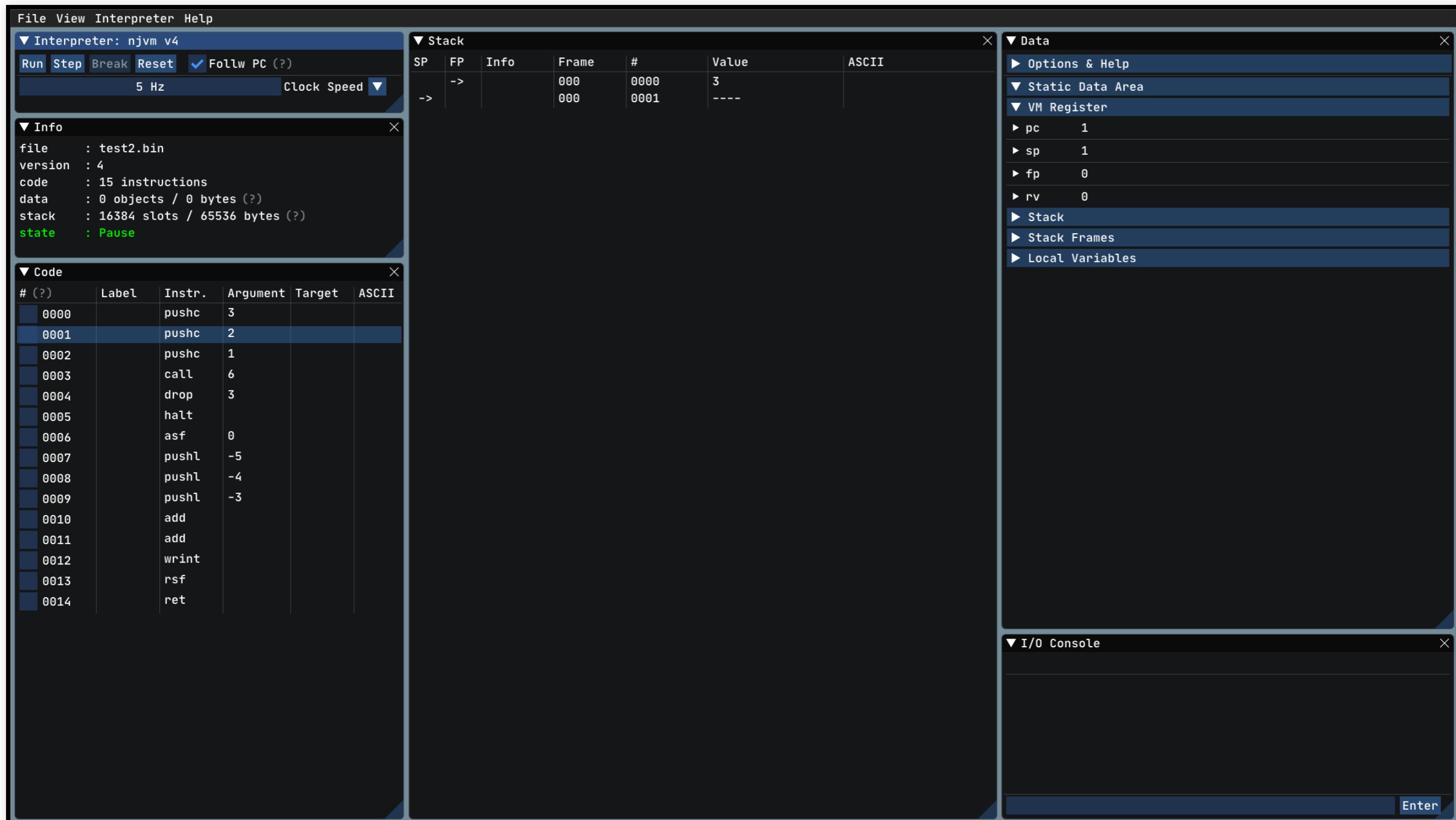
**Beispiel:** Ausgabe von  $f(3, 2, 1)$  mit  $f(a, b, c)$  {`wrint (a+b+c);`}

```
pushc 3
pushc 2
pushc 1
call f
drop 3
halt
f:
    asf 0
    pushl -5
    pushl -4
    pushl -3
    add
    add
    wrint
    rsf
    ret
```

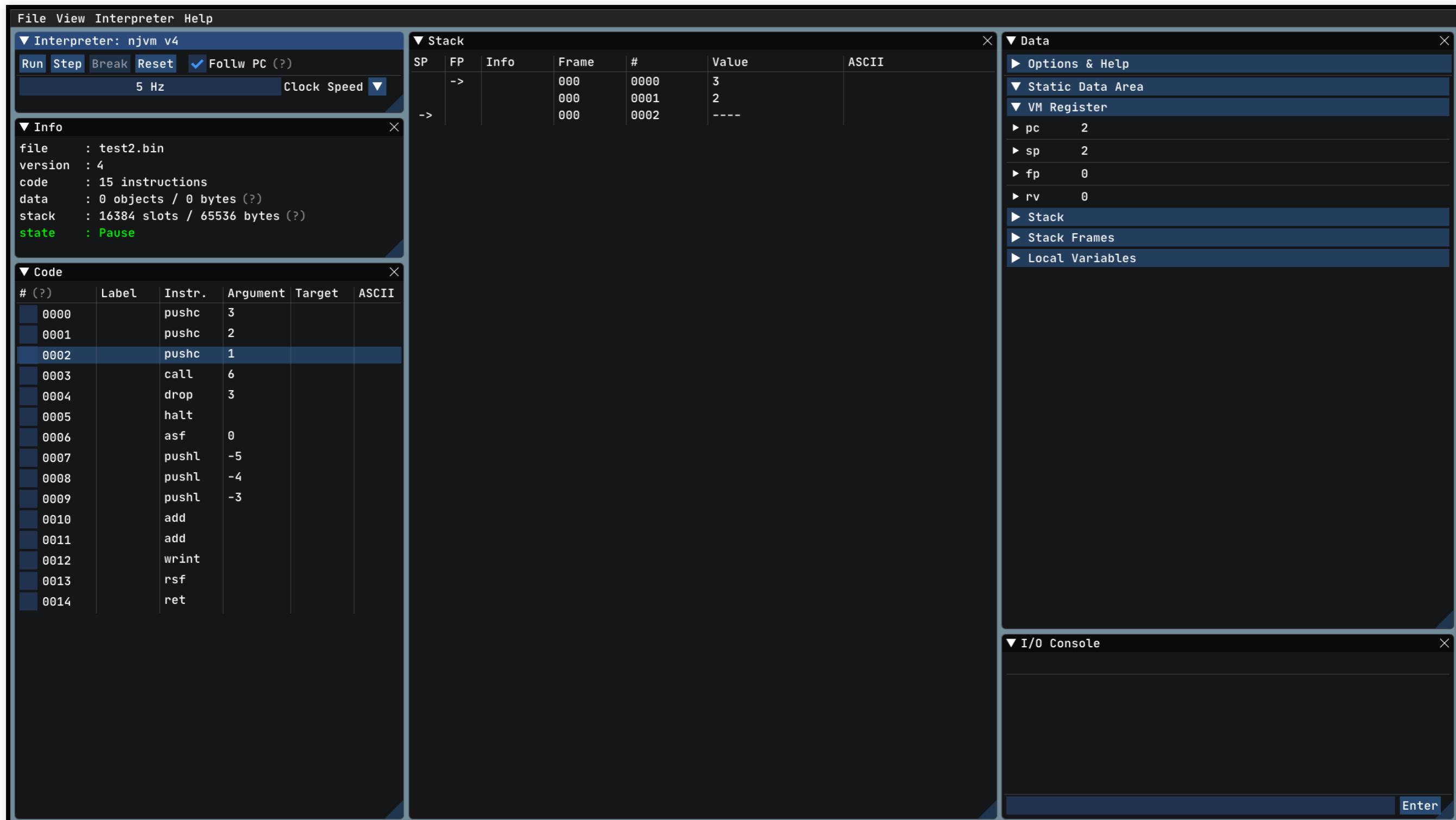
# Beispiel mit Argumenten, ohne Rückgabewert



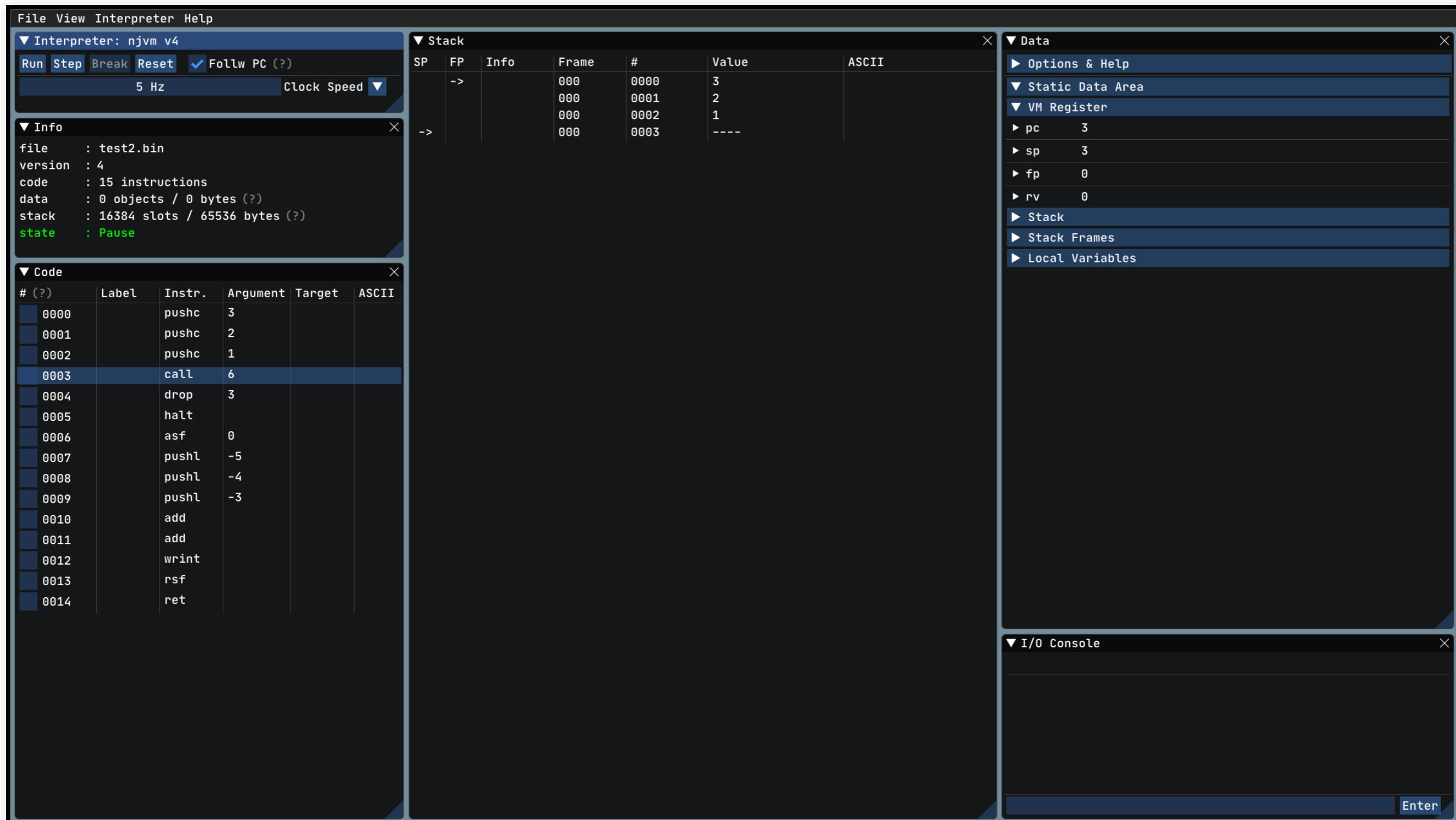
# Beispiel mit Argumenten, ohne Rückgabewert



# Beispiel mit Argumenten, ohne Rückgabewert



# Beispiel mit Argumenten, ohne Rückgabewert



# Beispiel mit Argumenten, ohne Rückgabewert

The screenshot shows the njvm v4 debugger interface with several panes:

- Interpreter: njvm v4**: Top-left pane with buttons: Run, Step, Break, Reset, Follow PC (?), Clock Speed (5 Hz).
- Info**: Top-right pane displaying file: test2.bin, version: 4, code: 15 instructions, data: 0 objects / 0 bytes (?), stack: 16384 slots / 65536 bytes (?), state: Pause.
- Stack**: Middle-right pane showing the stack structure. The stack grows downwards. It has columns: SP, FP, Info, Frame, #, Value, ASCII. The stack contains values 3, 2, 1, 4 at addresses 0000, 0001, 0002, 0003 respectively. The return address (ret\_addr) is at 0000, pointing to address 000. The stack frame is at 0000.
- Data**: Rightmost pane listing VM register values: pc: 6, sp: 4, fp: 0, rv: 0. It also includes sections for Options & Help, Static Data Area, VM Register, Stack, Stack Frames, and Local Variables.
- Code**: Bottom-left pane showing assembly-like code. The code consists of 15 instructions. Instruction 0006 is highlighted with a blue background. The columns are: # (?), Label, Instr., Argument, Target, ASCII.
- I/O Console**: Bottom-right pane for I/O operations, with an Enter button.

# (?)	Label	Instr.	Argument	Target	ASCII
0000		pushc	3		
0001		pushc	2		
0002		pushc	1		
0003		call	6		
0004		drop	3		
0005		halt			
0006		ASF	0		
0007		pushl	-5		
0008		pushl	-4		
0009		pushl	-3		
0010		add			
0011		add			
0012		wrint			
0013		rsf			
0014		ret			

# Beispiel mit Argumenten, ohne Rückgabewert

The screenshot shows the njvm v4 debugger interface with several panes:

- Interpreter: njvm v4**: Top-left pane with buttons for Run, Step, Break, Reset, and Follow PC. It also shows Clock Speed set to 5 Hz.
- Info**: Shows file: test2.bin, version: 4, code: 15 instructions, data: 0 objects / 0 bytes, stack: 16384 slots / 65536 bytes, and state: Pause.
- Code**: A table showing assembly instructions. The instruction at address 0007 is highlighted: pushl -5. Other instructions include pushc, call, drop, halt, asf, add, add, wrint, rsf, and ret.
- Stack**: A table showing the stack structure. The stack grows downwards. The current frame (Frame 001) contains the following values:

#	Value	ASCII
0000	0000	3
0001	0001	2
0002	0002	1
ret_addr	000	
old_fp	000	
0004	0004	0
001	0005	----
- Data**: A tree view of memory areas. Selected items include pc (7), sp (5), fp (5), and rv (0).
- I/O Console**: Bottom-right pane for input and output.

# Beispiel mit Argumenten, ohne Rückgabewert

The screenshot shows the njvm v4 debugger interface with several panes:

- Interpreter: njvm v4**: Top-left pane with buttons: Run, Step, Break, Reset, Follow PC (?), Clock Speed (5 Hz).
- Info**: Top-right pane displaying file: test2.bin, version: 4, code: 15 instructions, data: 0 objects / 0 bytes (?), stack: 16384 slots / 65536 bytes (?), state: Pause.
- Stack**: Middle-right pane showing the stack structure. The stack grows downwards. It has columns: SP, FP, Info, Frame, #, Value, ASCII. Rows include 000, 000, 000, 000, 000, 3; 000, 0001, 000, 0002, 0001, 2; 000, 0002, 000, 0003, 0003, 1; ret\_addr, 000, 000, 0004, 0004, 4; old\_fp, 000, 000, 0005, 0005, 0; ->, 001, 001, 0006, 0006, ----. The row with frame 001 is highlighted.
- Data**: Rightmost pane listing VM register values: pc: 8, sp: 6, fp: 5, rv: 0. It also includes sections for Options & Help, Static Data Area, VM Register, Stack, Stack Frames, and Local Variables.
- Code**: Bottom-left pane showing assembly-like code. Columns: # (?), Label, Instr., Argument, Target, ASCII. Rows include pushc 3, pushc 2, pushc 1, call 6, drop 3, halt, asf 0, pushl -5, pushl -4 (highlighted), pushl -3, add, add, wrint, rsf, ret.
- I/O Console**: Bottom-right pane for I/O operations, with an Enter button at the bottom right.

# Beispiel mit Argumenten, ohne Rückgabewert

The screenshot shows the njvm v4 debugger interface with several panes:

- Interpreter: njvm v4**: Top-left pane with buttons: Run, Step, Break, Reset, Follow PC (?), Clock Speed (5 Hz).
- Info**: Top-right pane displaying file: test2.bin, version: 4, code: 15 instructions, data: 0 objects / 0 bytes (?), stack: 16384 slots / 65536 bytes (?), state: Pause.
- Code**: Bottom-left pane showing assembly-like code:

# (?)	Label	Instr.	Argument	Target	ASCII
0000		pushc	3		
0001		pushc	2		
0002		pushc	1		
0003		call	6		
0004		drop	3		
0005		halt			
0006		asf	0		
0007		pushl	-5		
0008		pushl	-4		
0009		pushl	-3		
0010		add			
0011		add			
0012		wrint			
0013		rsf			
0014		ret			
- Stack**: Middle-right pane showing the stack structure:

SP	FP	Info	Frame	#	Value	ASCII
			000	0000	3	
			000	0001	2	
			000	0002	1	
		ret_addr	000	0003	4	
		old_fp	000	0004	0	
->			001	0005	3	
->			001	0006	2	
->			001	0007	----	
- Data**: Rightmost pane showing VM register values:
  - pc: 9
  - sp: 7
  - fp: 5
  - rv: 0
- I/O Console**: Bottom-right pane with an Enter button.

# Beispiel mit Argumenten, ohne Rückgabewert

The screenshot shows the njvm v4 debugger interface with several panes:

- Interpreter: njvm v4**: Top-left pane with buttons: Run, Step, Break, Reset, Follow PC (?), Clock Speed (5 Hz).
- Info**: Displays file: test2.bin, version: 4, code: 15 instructions, data: 0 objects / 0 bytes (?), stack: 16384 slots / 65536 bytes (?), state: Pause.
- Code**: Shows assembly-like code with columns: # (?), Label, Instr., Argument, Target, ASCII. The instruction at address 0010 is highlighted.
- Stack**: Shows the stack memory layout with columns: SP, FP, Info, Frame, #, Value, ASCII. The stack grows downwards.
- Data**: Shows VM register values: pc: 10, sp: 8, fp: 5, rv: 0.
- I/O Console**: Bottom-right pane with an Enter button.

Code pane details:

# (?)	Label	Instr.	Argument	Target	ASCII
0000		pushc	3		
0001		pushc	2		
0002		pushc	1		
0003		call	6		
0004		drop	3		
0005		halt			
0006		asf	0		
0007		pushl	-5		
0008		pushl	-4		
0009		pushl	-3		
0010		add			
0011		add			
0012		wrint			
0013		rsf			
0014		ret			

# Beispiel mit Argumenten, ohne Rückgabewert

The screenshot shows the njvm v4 debugger interface with several panes:

- Interpreter: njvm v4**: Top-left pane with buttons for Run, Step, Break, Reset, and Follow PC (checked). It also shows a clock speed of 5 Hz.
- Info**: Shows file: test2.bin, version: 4, code: 15 instructions, data: 0 objects / 0 bytes (?), stack: 16384 slots / 65536 bytes (?), and state: Pause.
- Code**: A table of assembly instructions:

# (?)	Label	Instr.	Argument	Target	ASCII
0000		pushc	3		
0001		pushc	2		
0002		pushc	1		
0003		call	6		
0004		drop	3		
0005		halt			
0006		asf	0		
0007		pushl	-5		
0008		pushl	-4		
0009		pushl	-3		
0010		add			
0011		add			
0012		wrint			
0013		rsf			
0014		ret			

- Stack**: A table showing the stack structure:

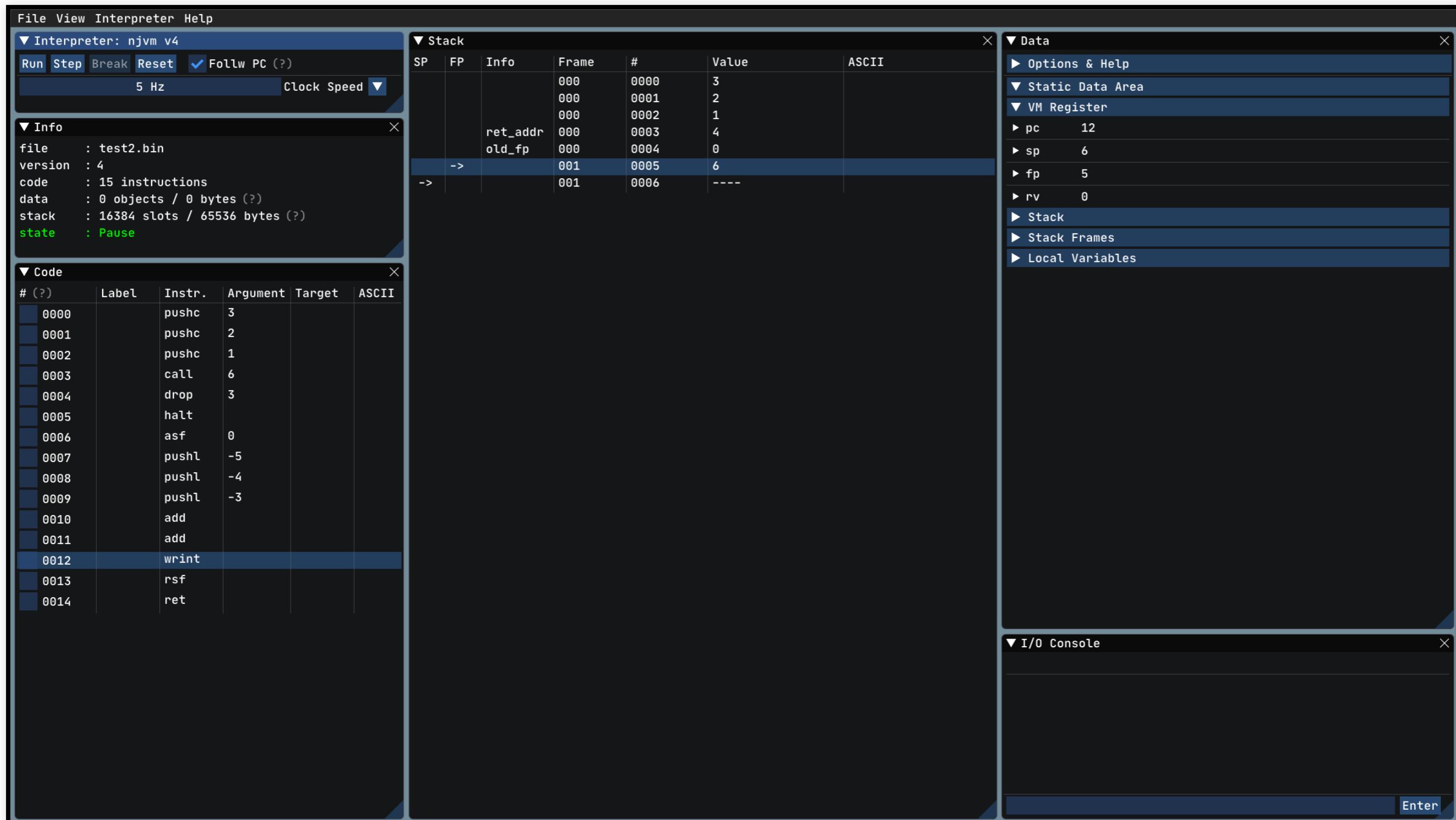
SP	FP	Info	Frame	#	Value	ASCII
			000	0000	3	
			000	0001	2	
			000	0002	1	
		ret_addr	000	0003	4	
		old_fp	000	0004	0	
->			001	0005	3	
->			001	0006	3	
->			001	0007	----	

- Data**: Shows VM register values:

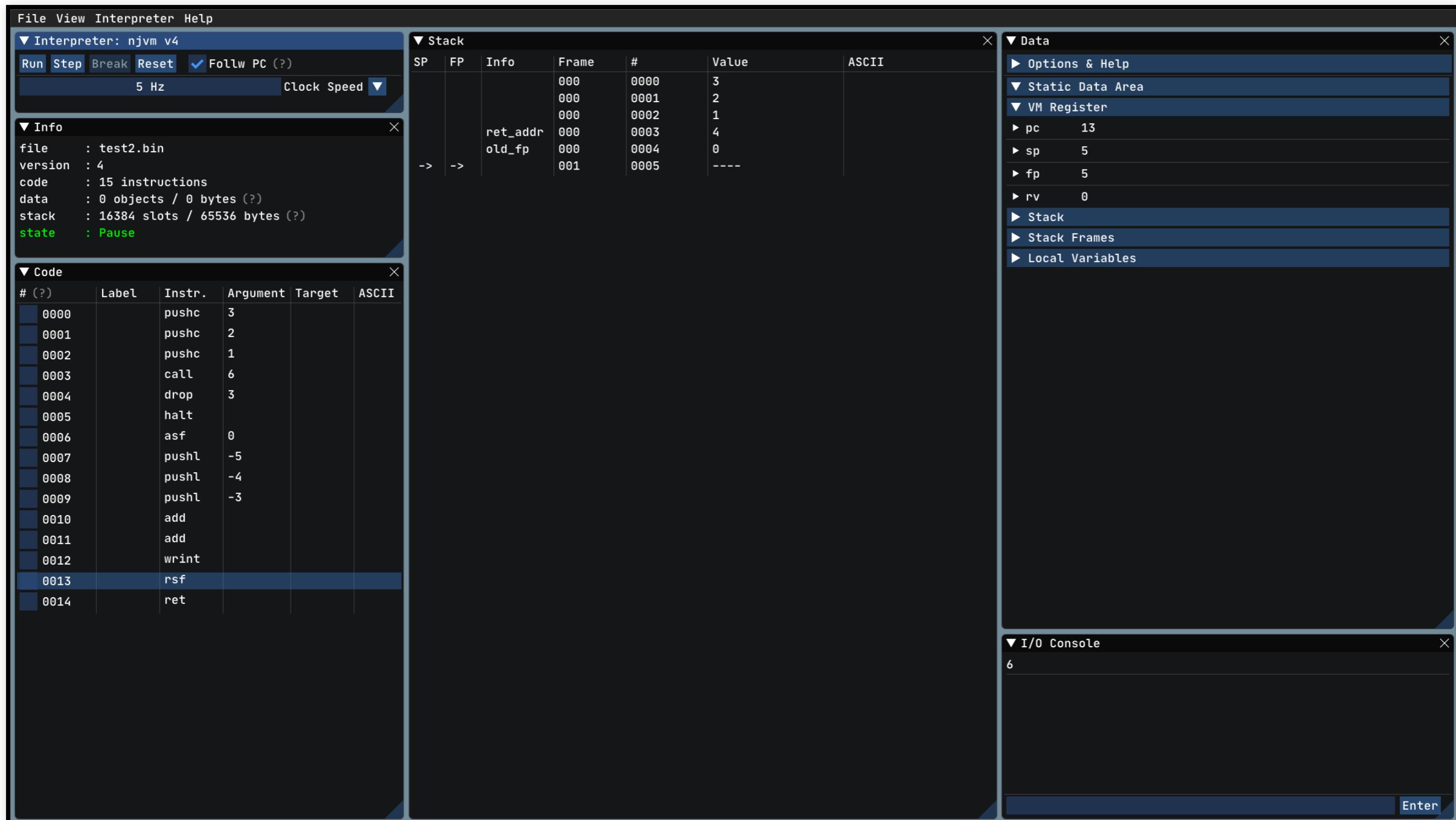
  - pc: 11
  - sp: 7
  - fp: 5
  - rv: 0

- I/O Console**: Bottom-right pane with an Enter button.

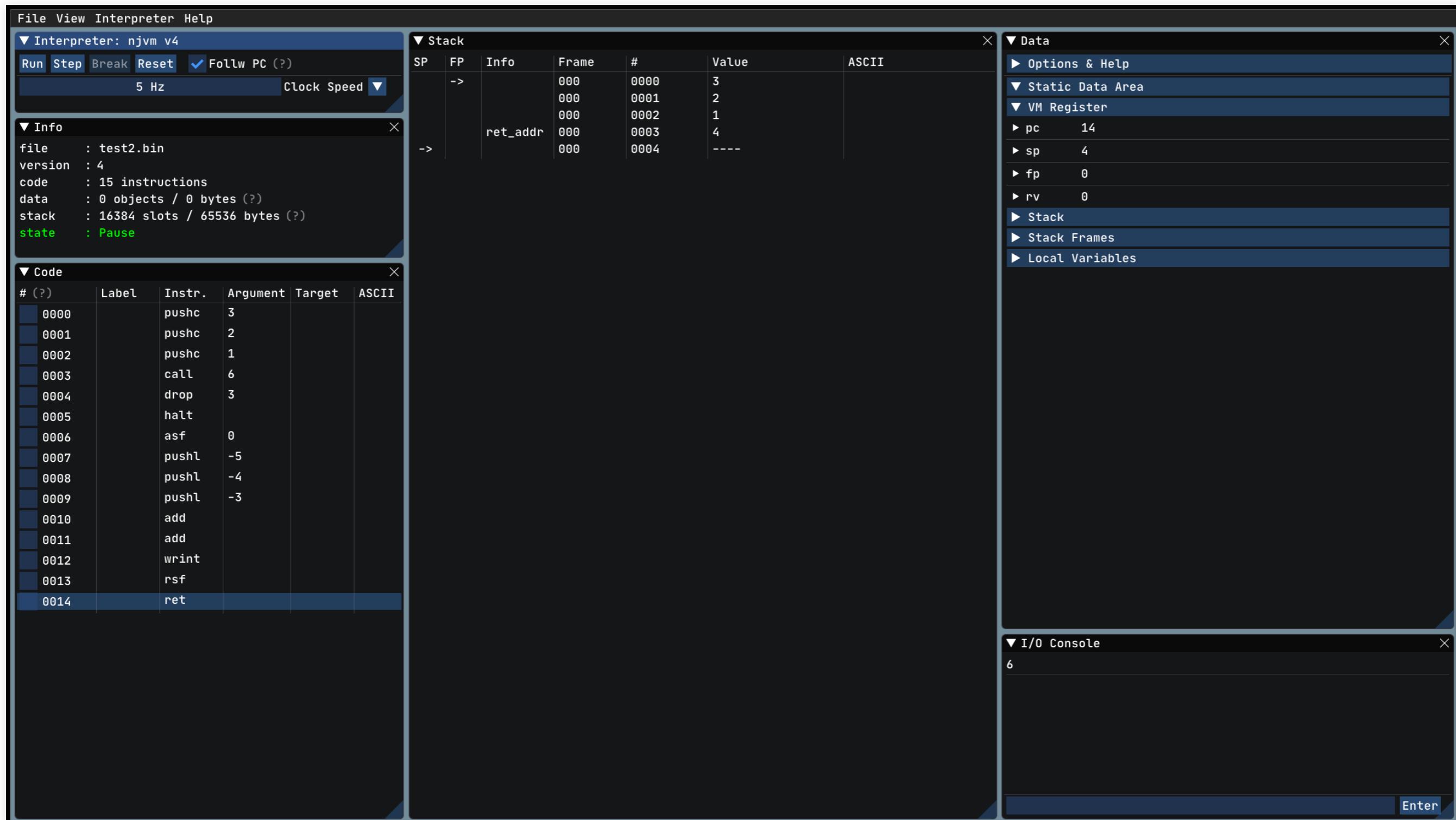
# Beispiel mit Argumenten, ohne Rückgabewert



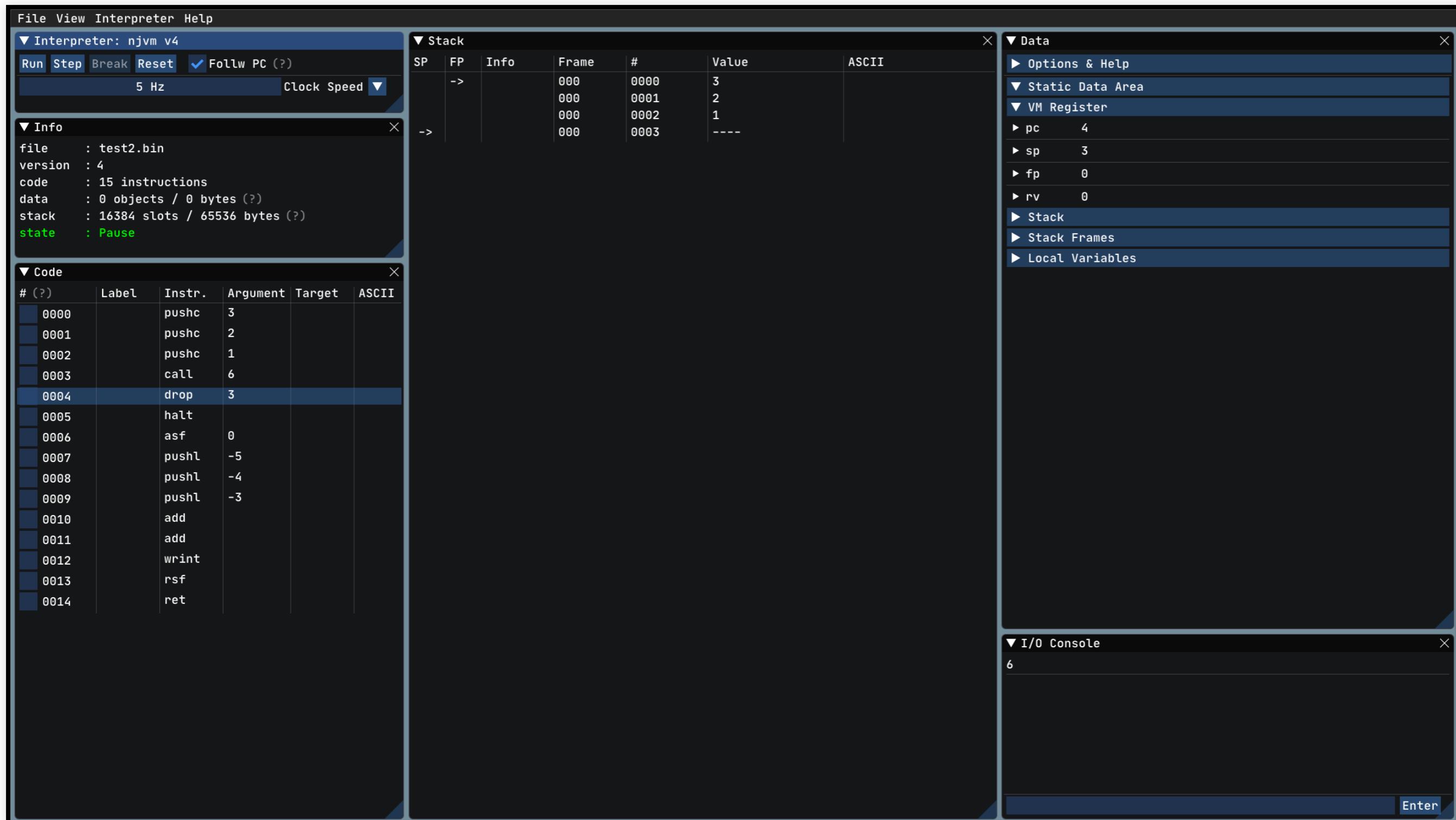
# Beispiel mit Argumenten, ohne Rückgabewert



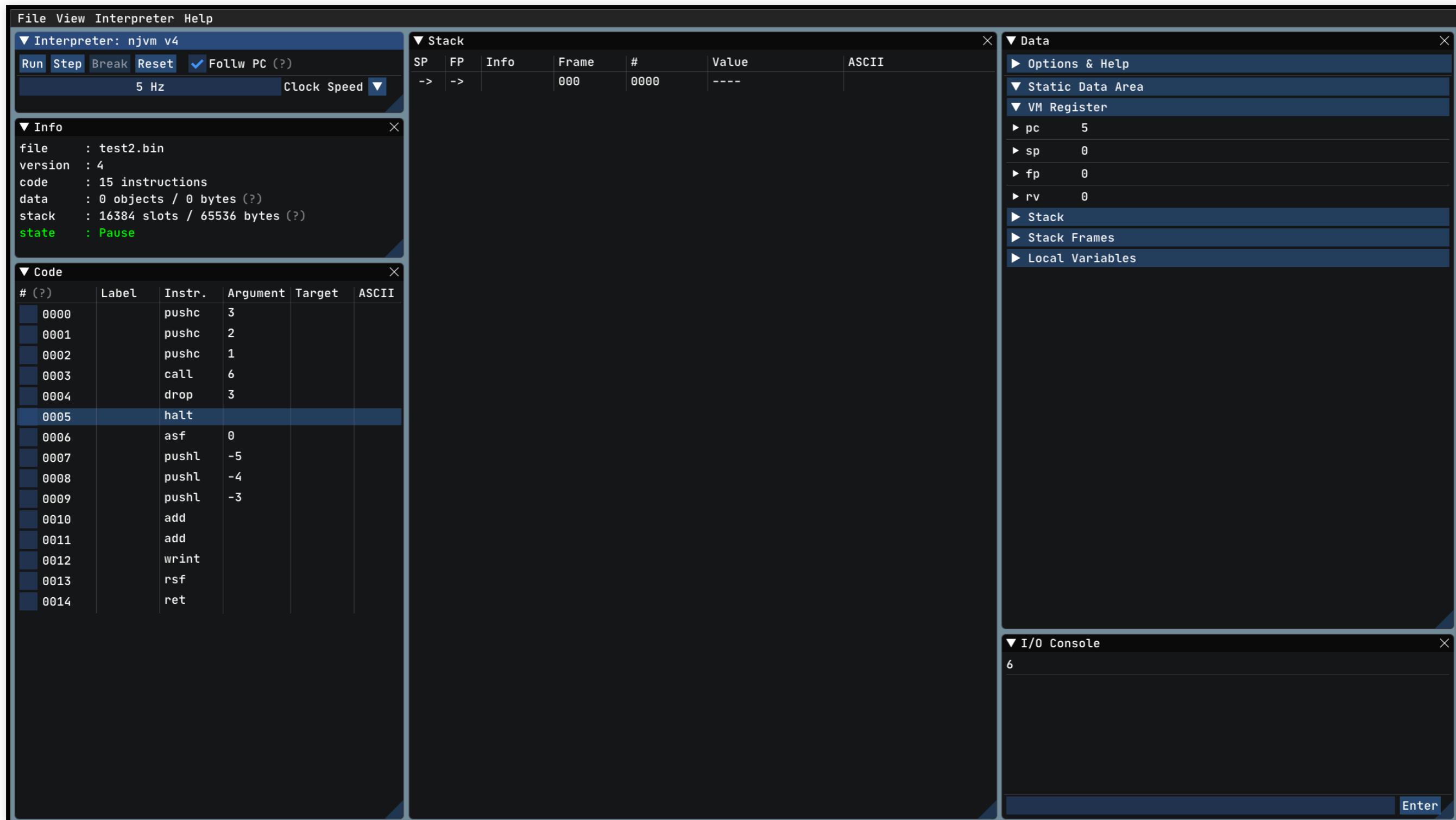
# Beispiel mit Argumenten, ohne Rückgabewert



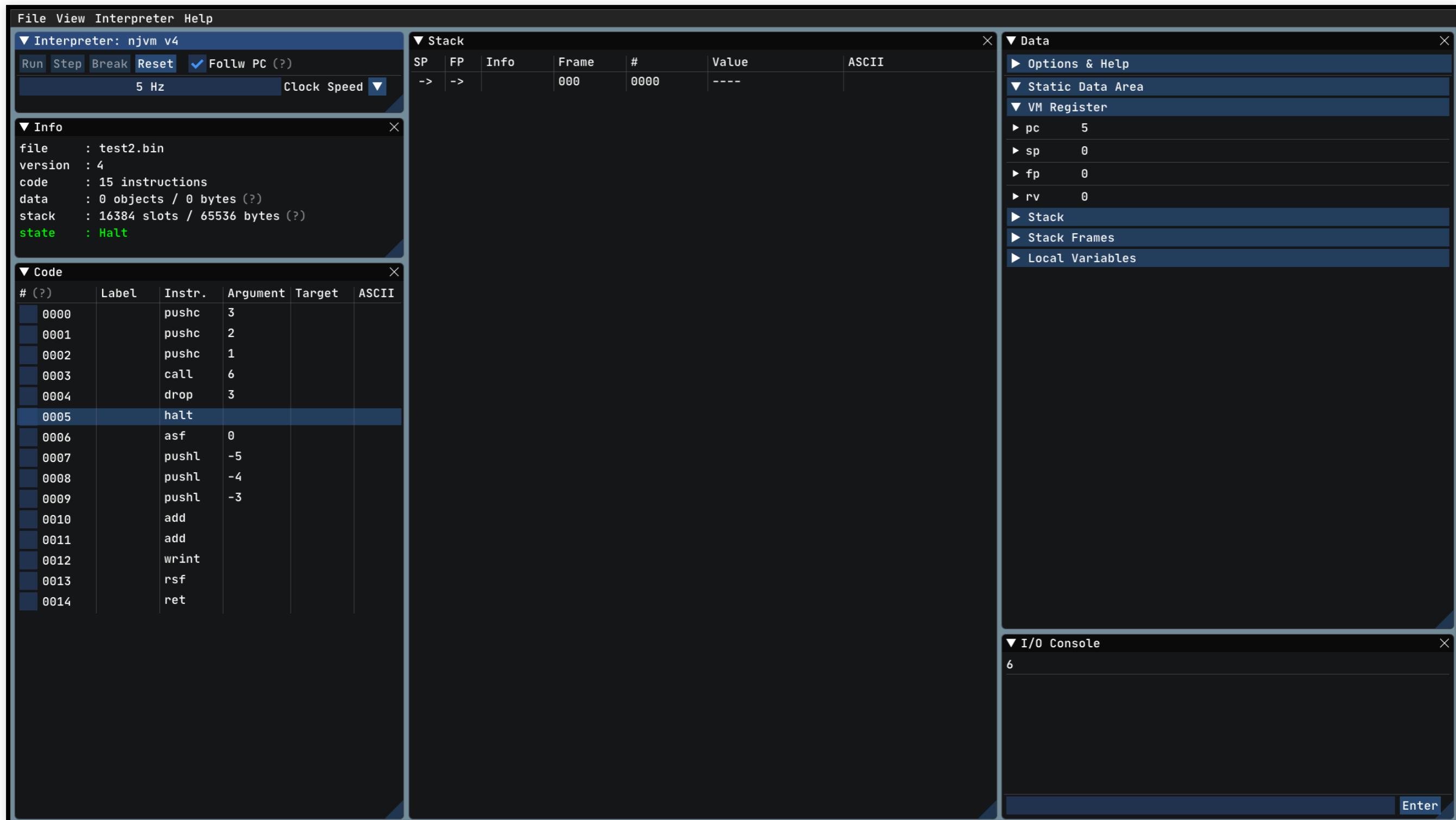
# Beispiel mit Argumenten, ohne Rückgabewert



# Beispiel mit Argumenten, ohne Rückgabewert

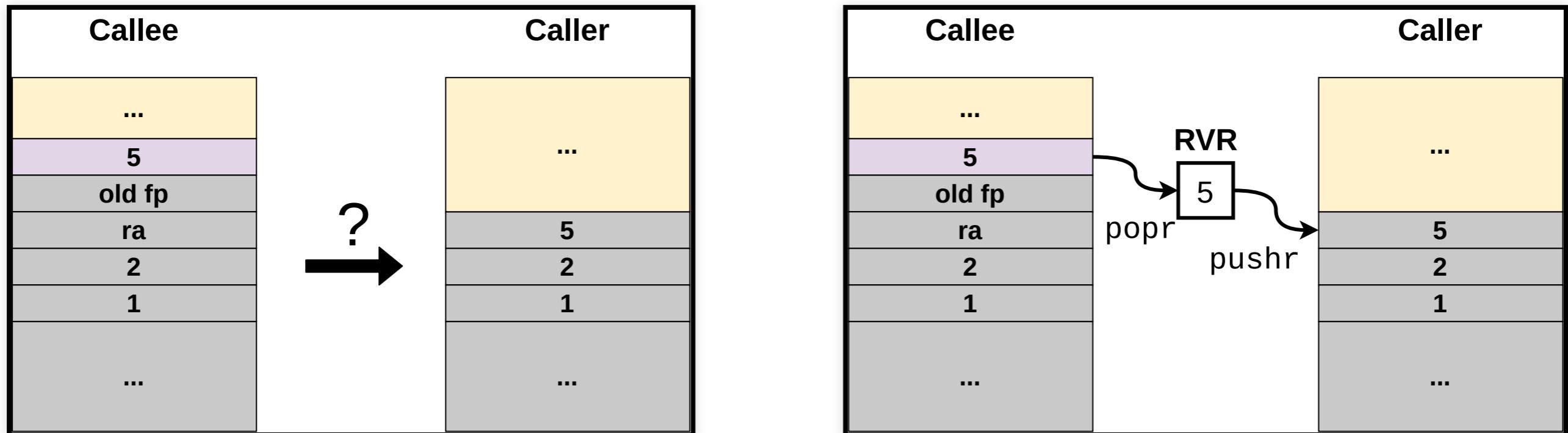


# Beispiel mit Argumenten, ohne Rückgabewert



# Funktionsaufruf ohne Argumenten, mit Rückgabewert

- Beispiel: `1+2*f()`, mit `int f(){ return 5; }`



Wir benötigen 2 neue Instruktionen und das neue **Return Value Register (RVR)**, welches als Zwischenspeicher fungiert.

- `pushr → ... -> ... rv` – holt Wert aus dem RVR-Register und speichert ihn auf dem Stack ab
- `popr → ... rv -> ...` – holt einen Wert vom Stack und speichert ihn im RVR-Register

# Funktionsaufruf ohne Argumenten, mit Rückgabewert

Caller

```
call Funktionslabel  
pushr
```

Callee

```
Funktionslabel:  
    asf <numLocalVars>  
    <body>  
    <push retval> // Vorbereitung für popr  
    popr  
    rsf  
    ret
```

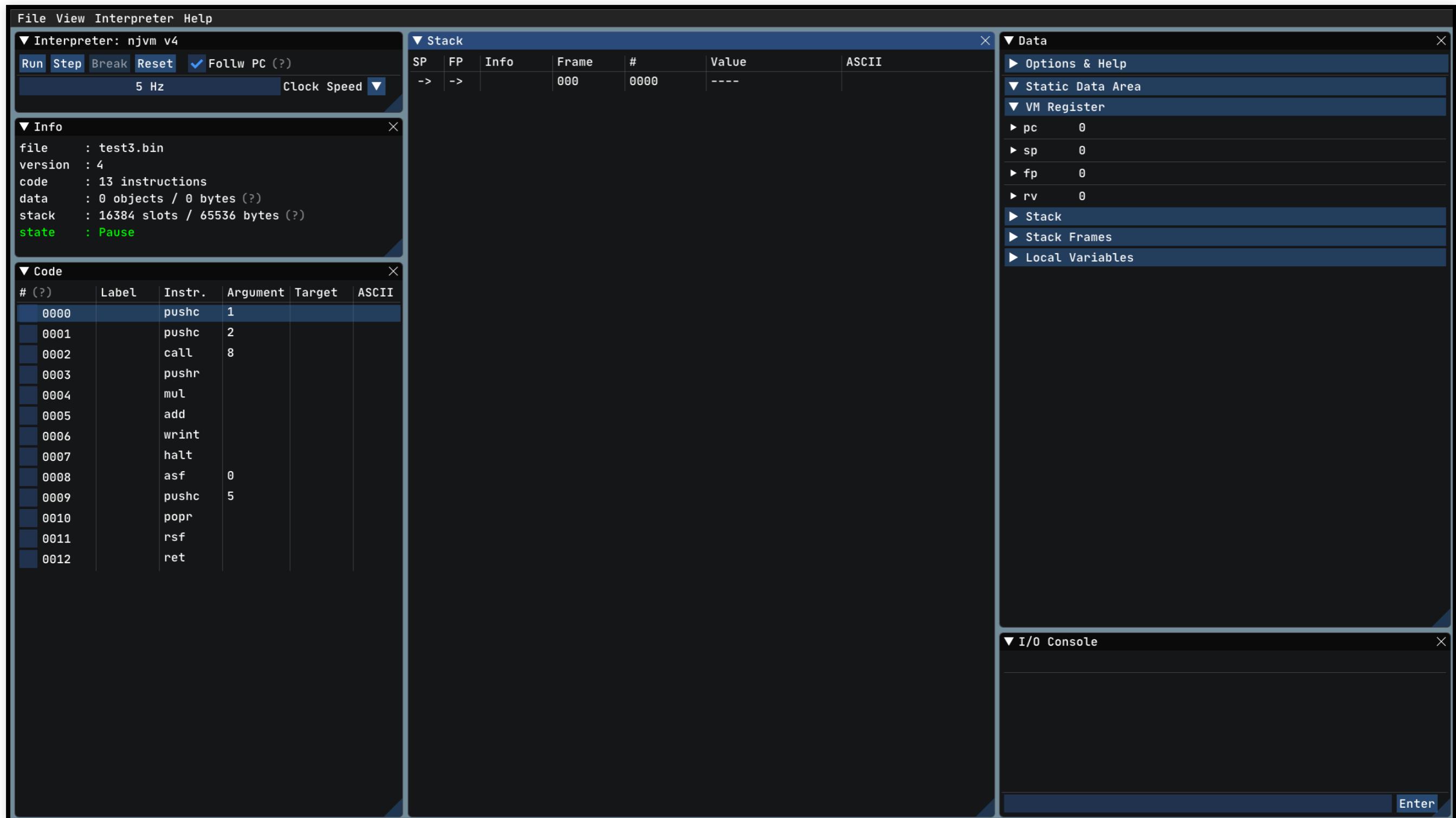
# Beispiel: Funktionsaufruf nur mit Rückgabewert

**Beispiel:** Ausgabe von  $1+2*f()$ , mit int f(){ return 5; }

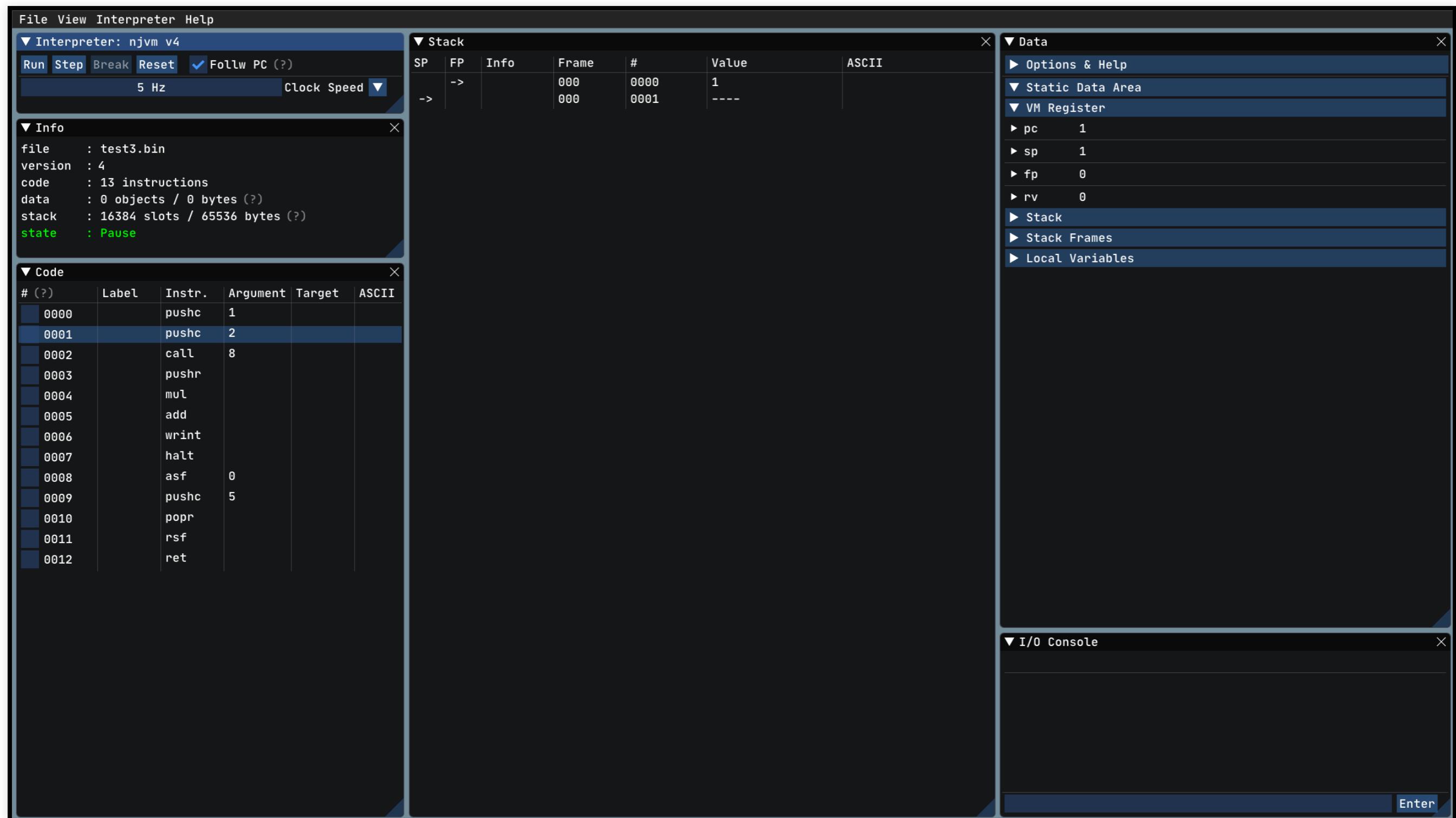
```
pushc 1
pushc 2
call f
pushr
mul
add
wrint
halt
f:
    asf 0
    pushc 5
    popr
    rsf
    ret
```

Beispiel ohne Argumenten, mit Rückgabewert

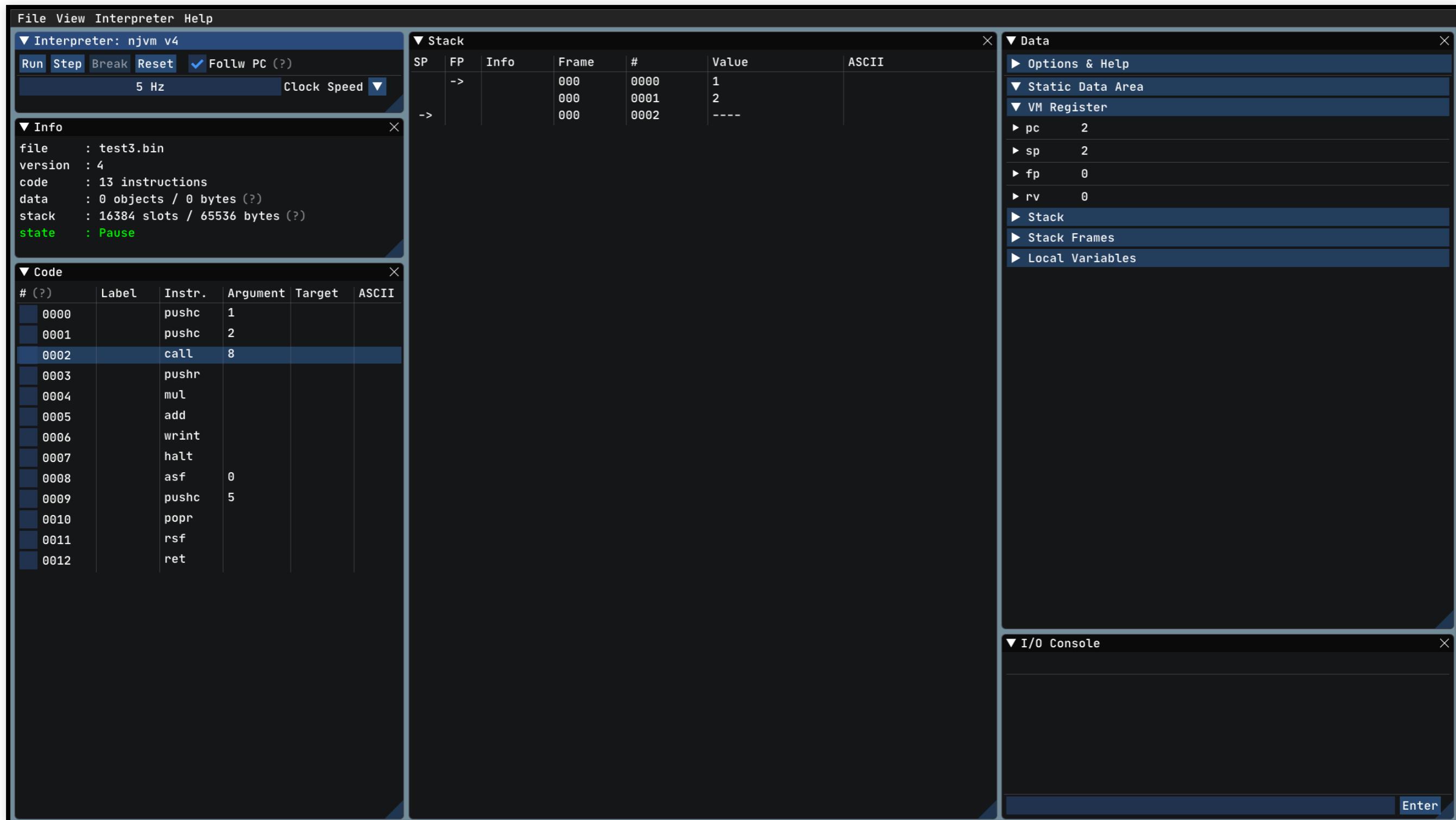
# Beispiel ohne Argumenten, mit Rückgabewert



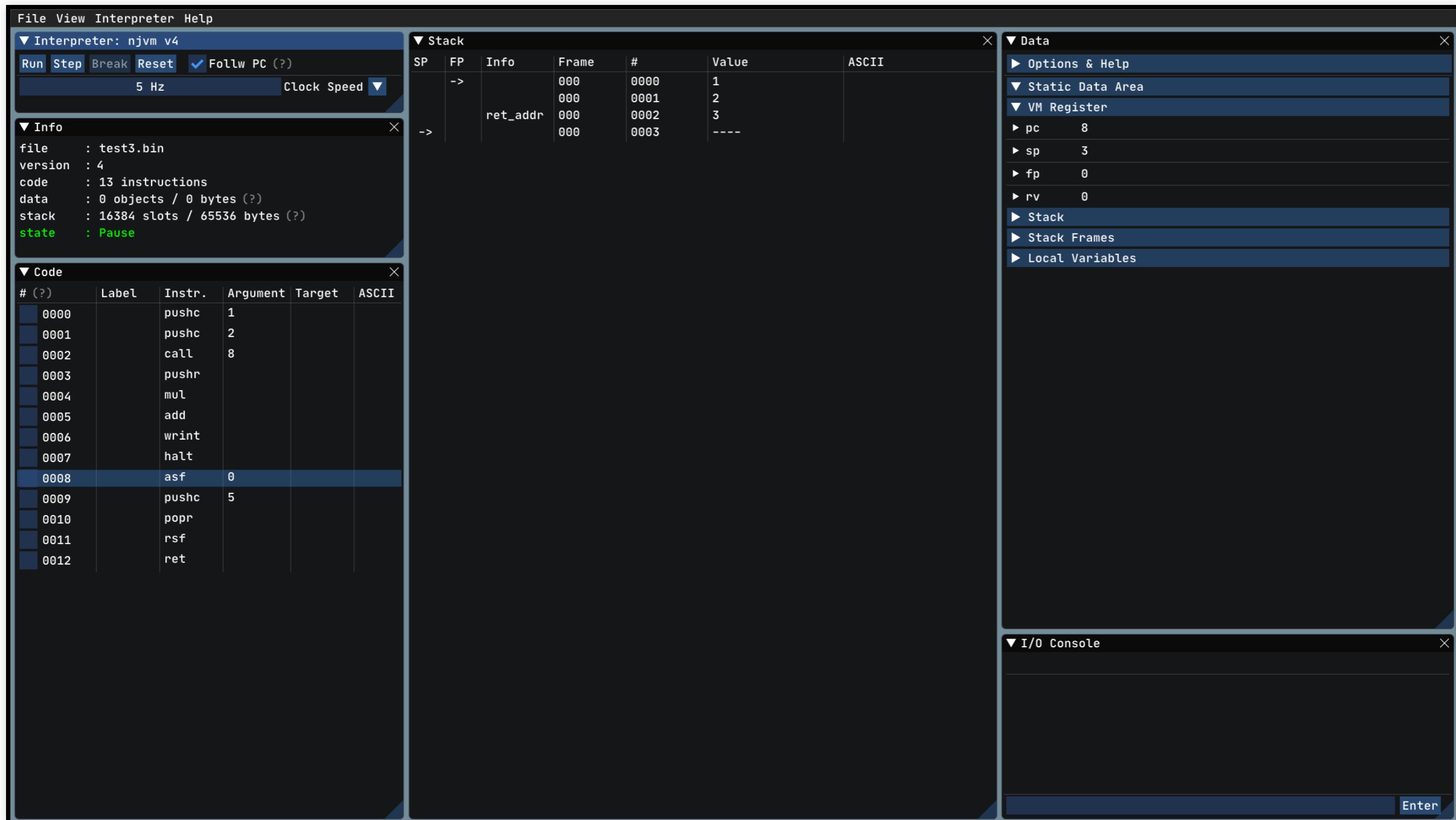
# Beispiel ohne Argumenten, mit Rückgabewert



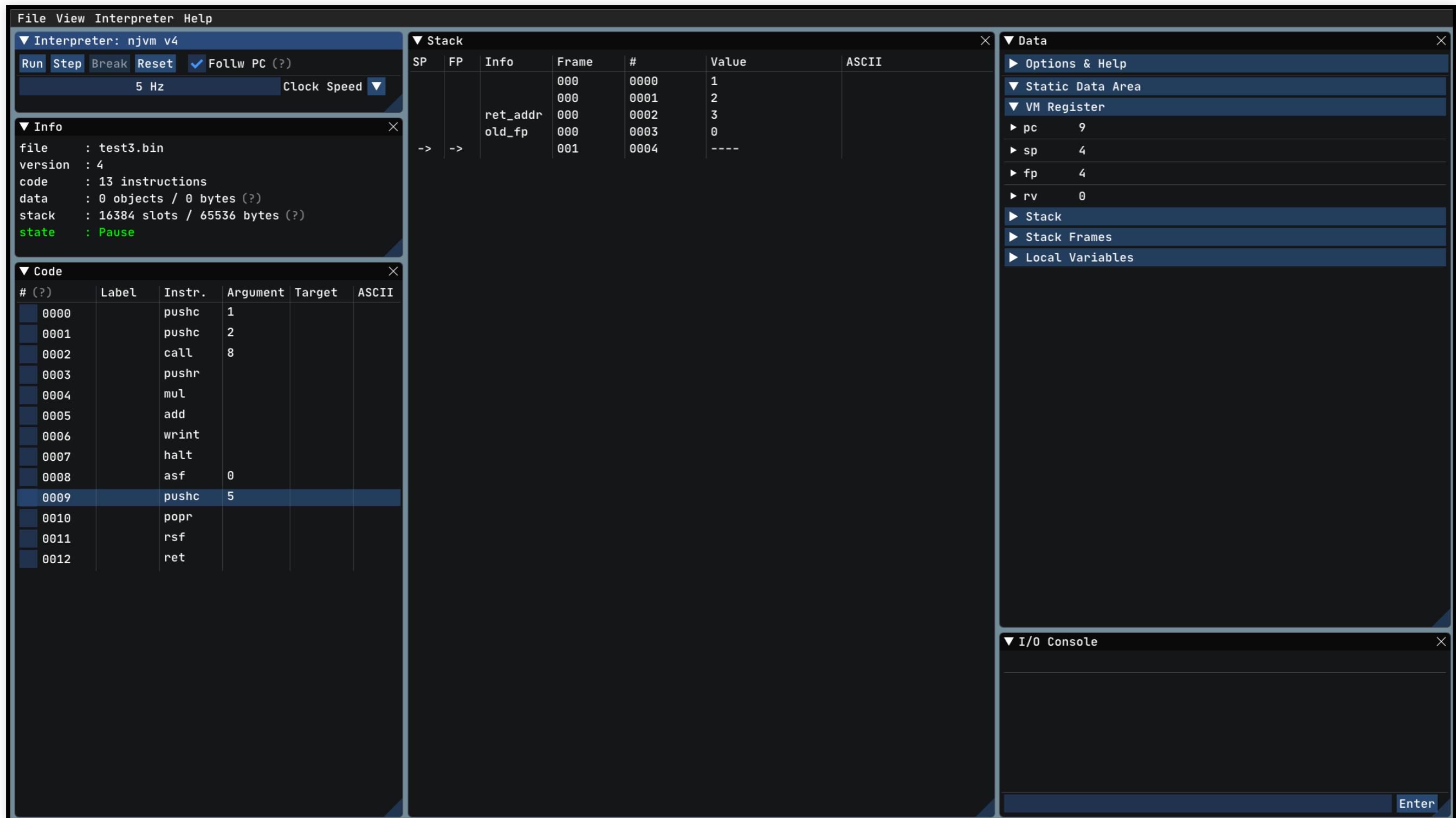
# Beispiel ohne Argumenten, mit Rückgabewert



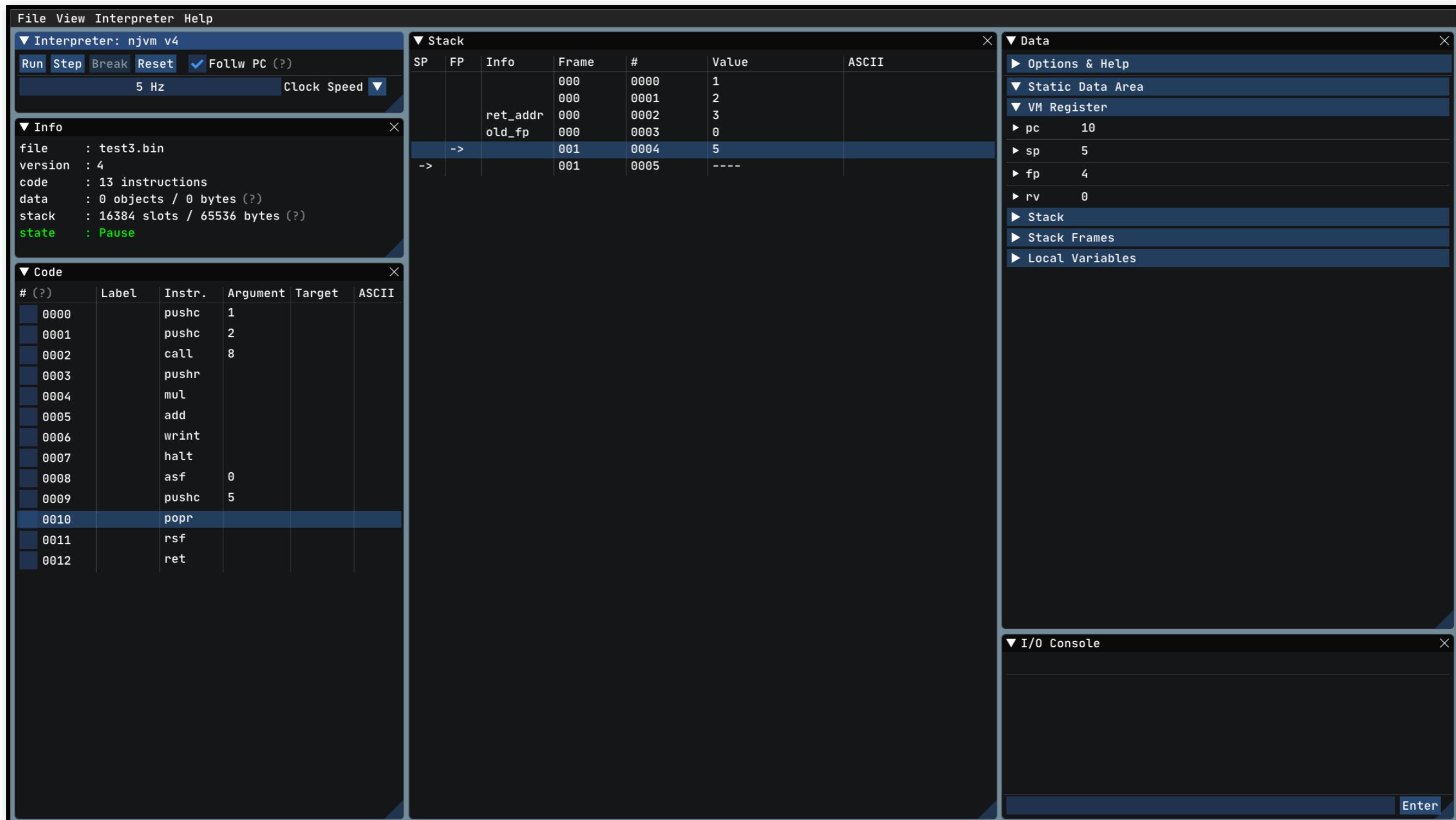
# Beispiel ohne Argumenten, mit Rückgabewert



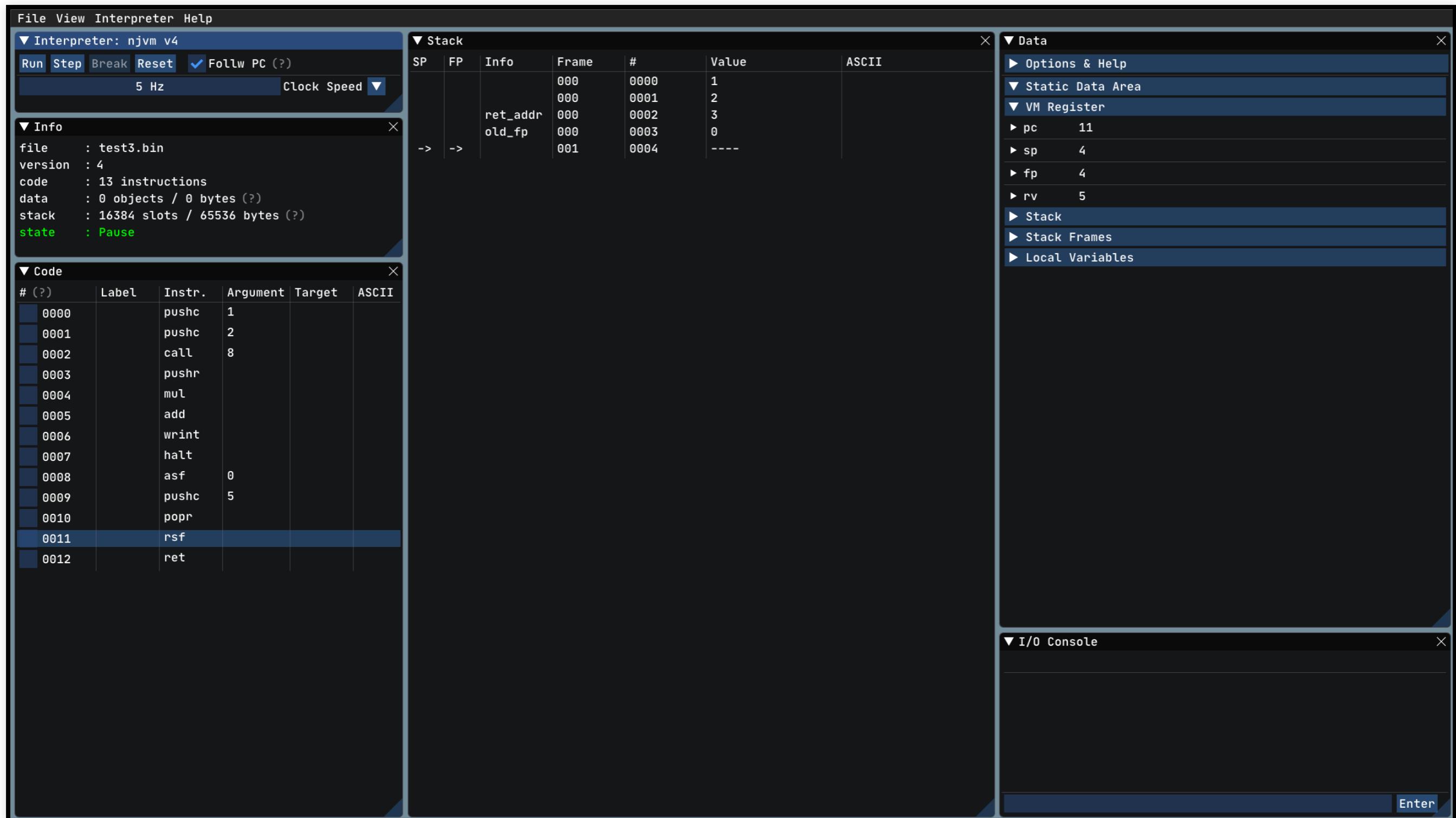
# Beispiel ohne Argumenten, mit Rückgabewert



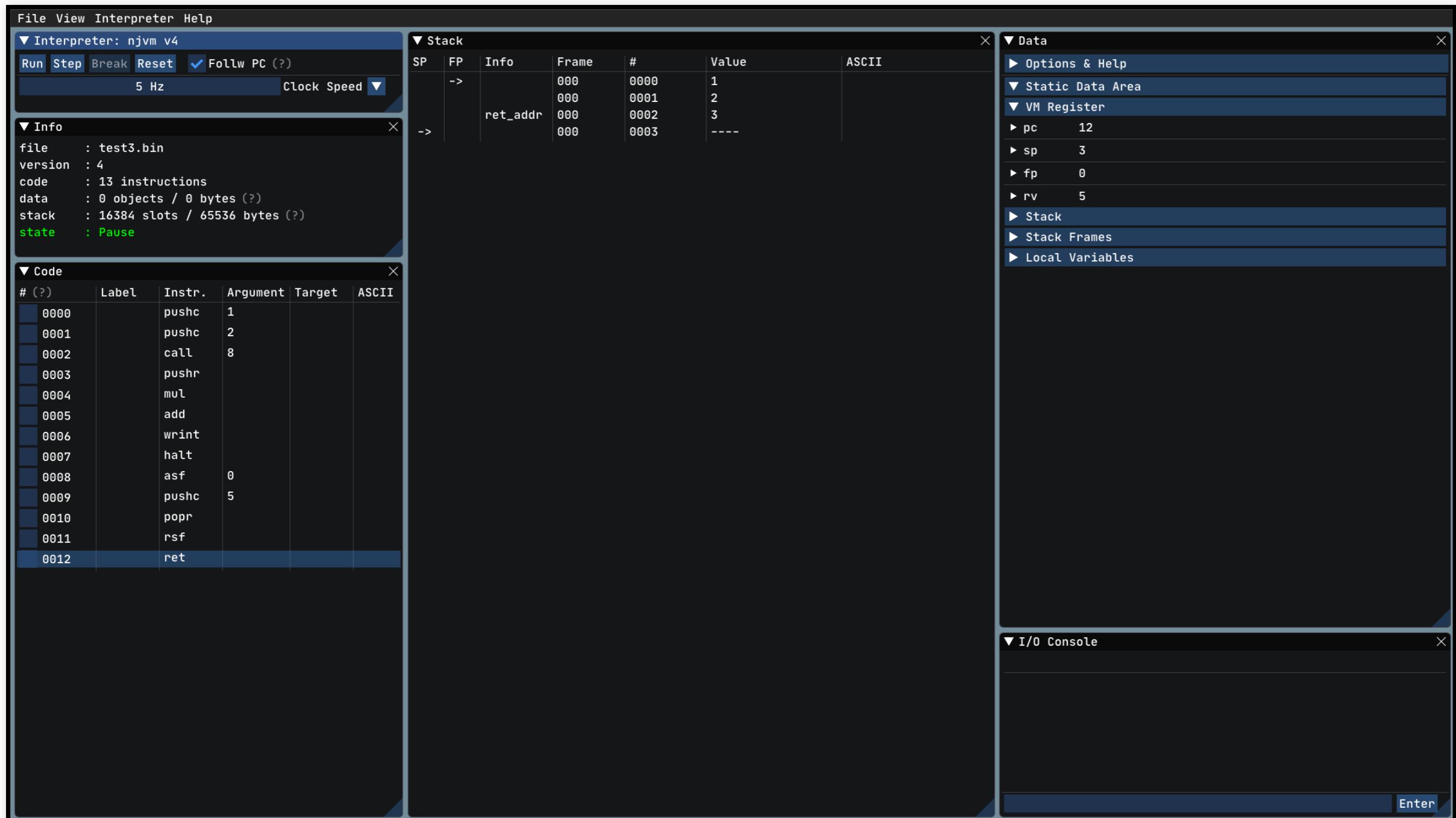
# Beispiel ohne Argumenten, mit Rückgabewert



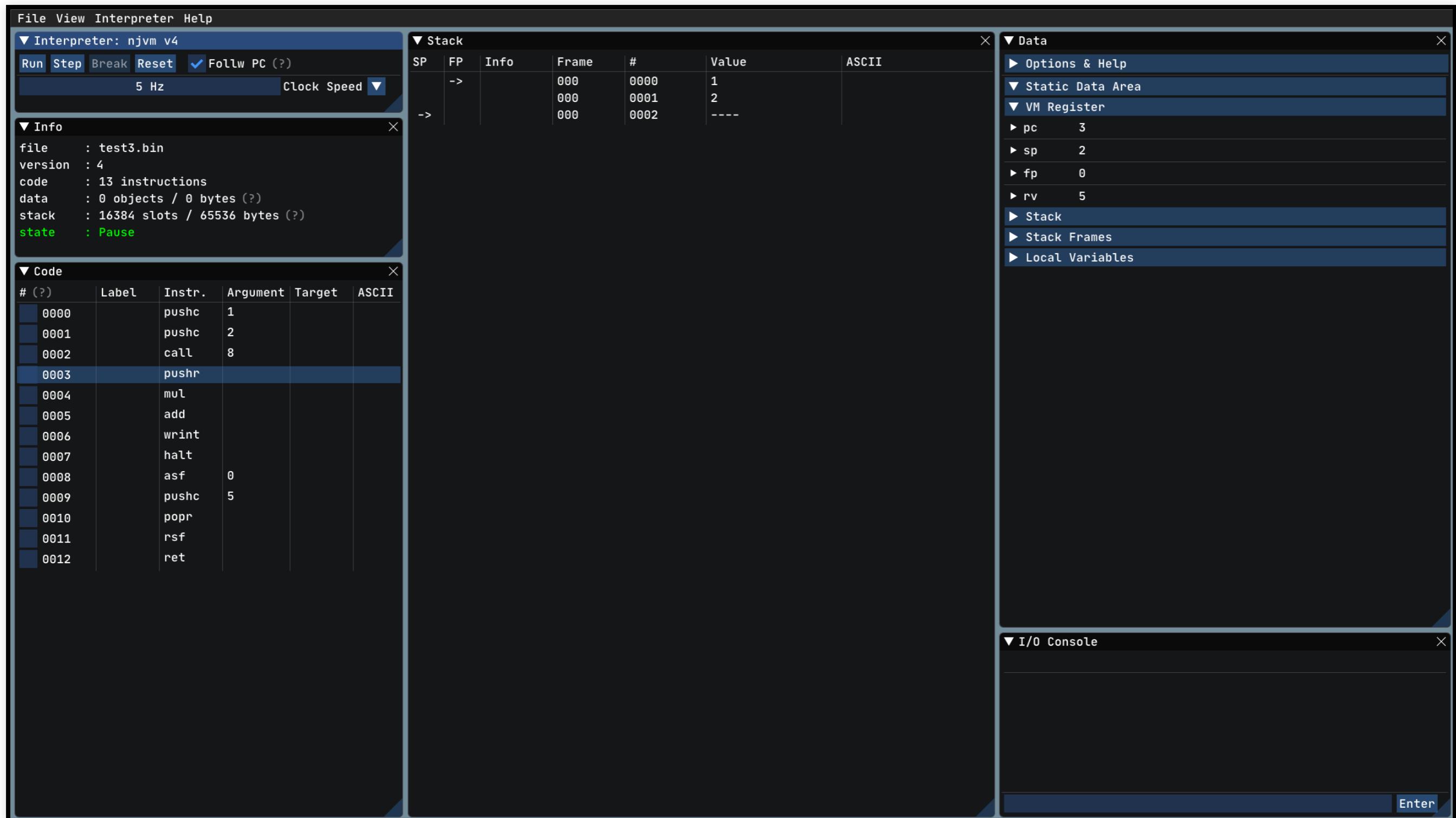
# Beispiel ohne Argumenten, mit Rückgabewert



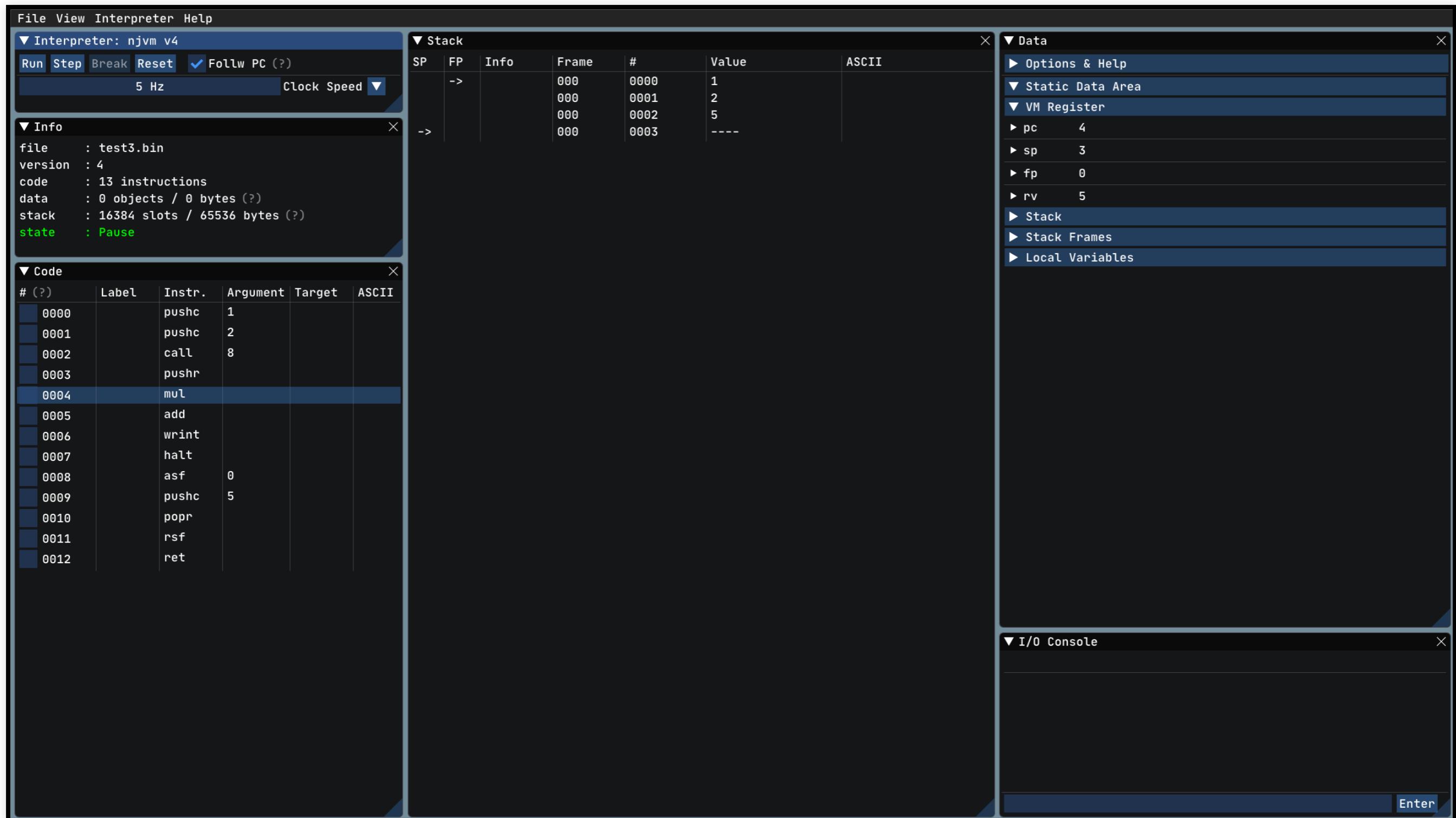
# Beispiel ohne Argumenten, mit Rückgabewert



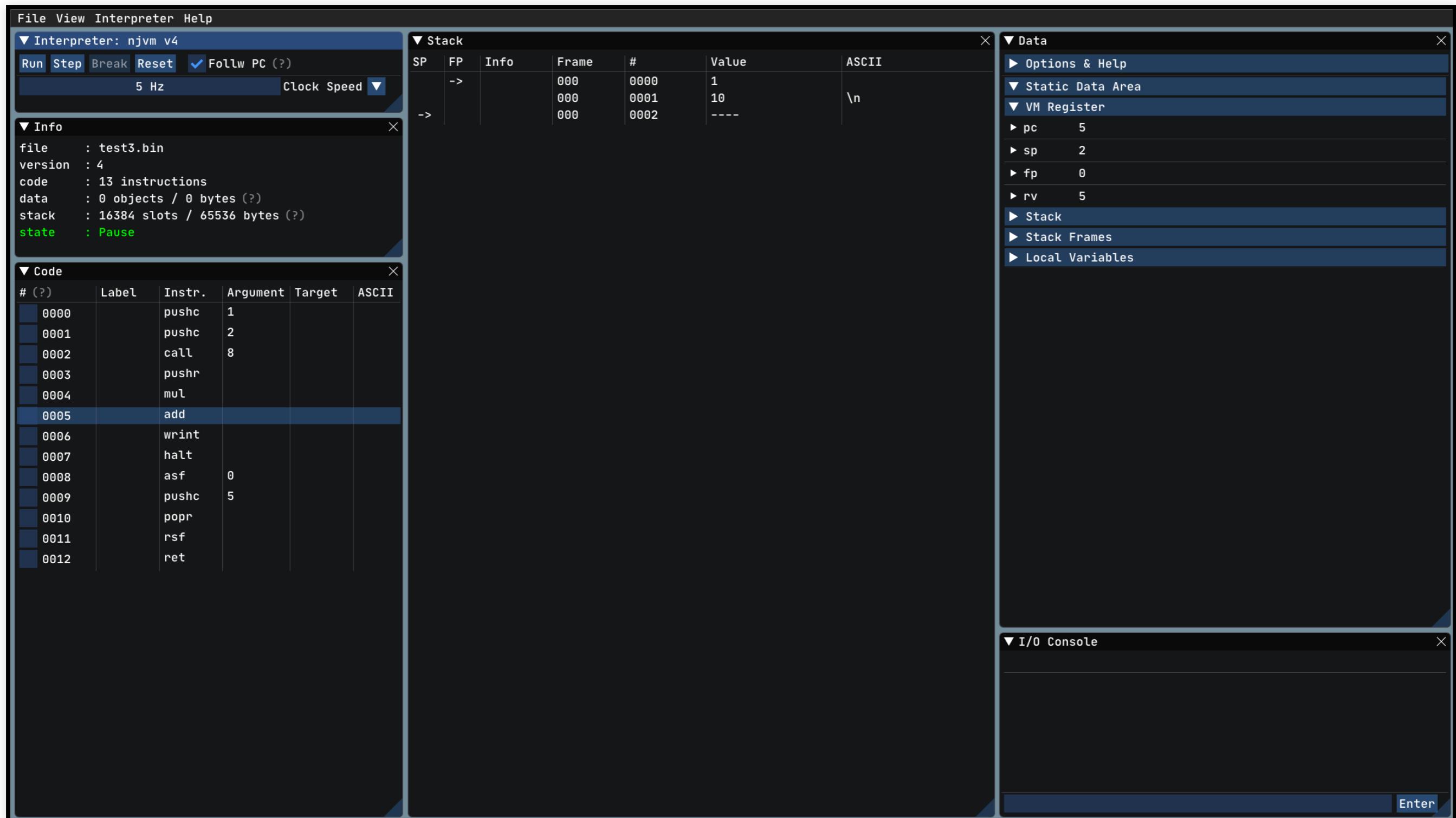
# Beispiel ohne Argumenten, mit Rückgabewert



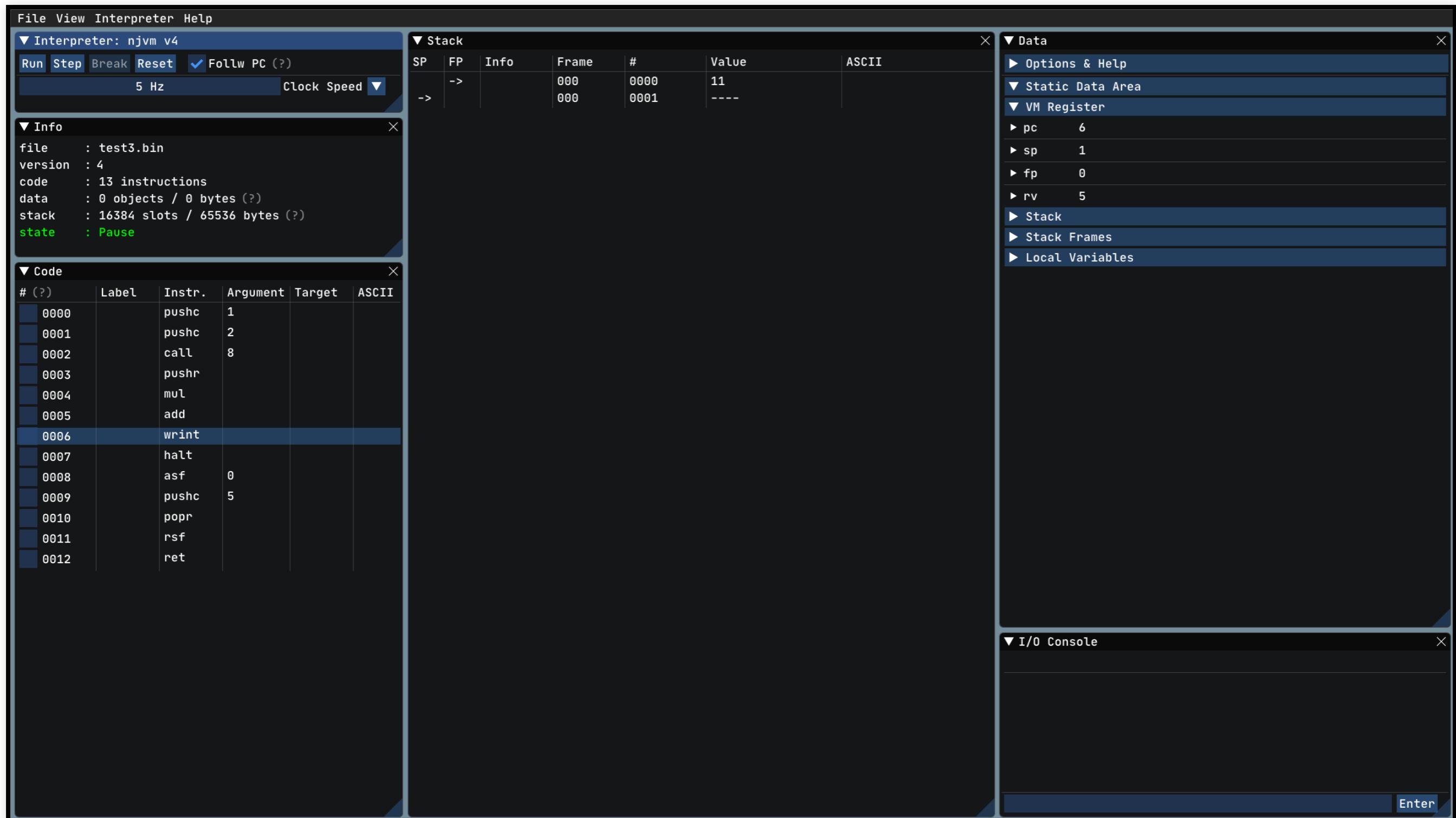
# Beispiel ohne Argumenten, mit Rückgabewert



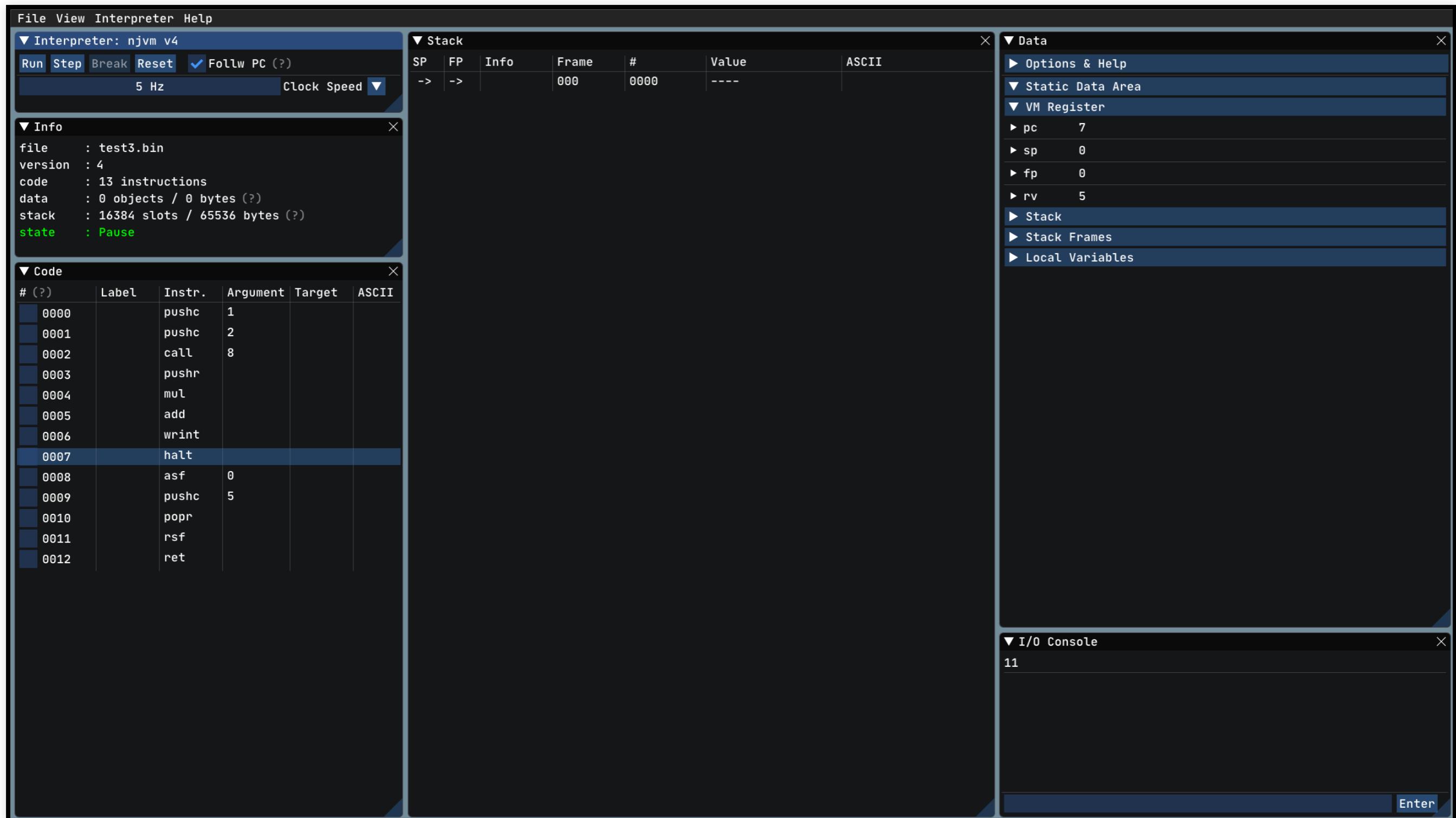
# Beispiel ohne Argumenten, mit Rückgabewert



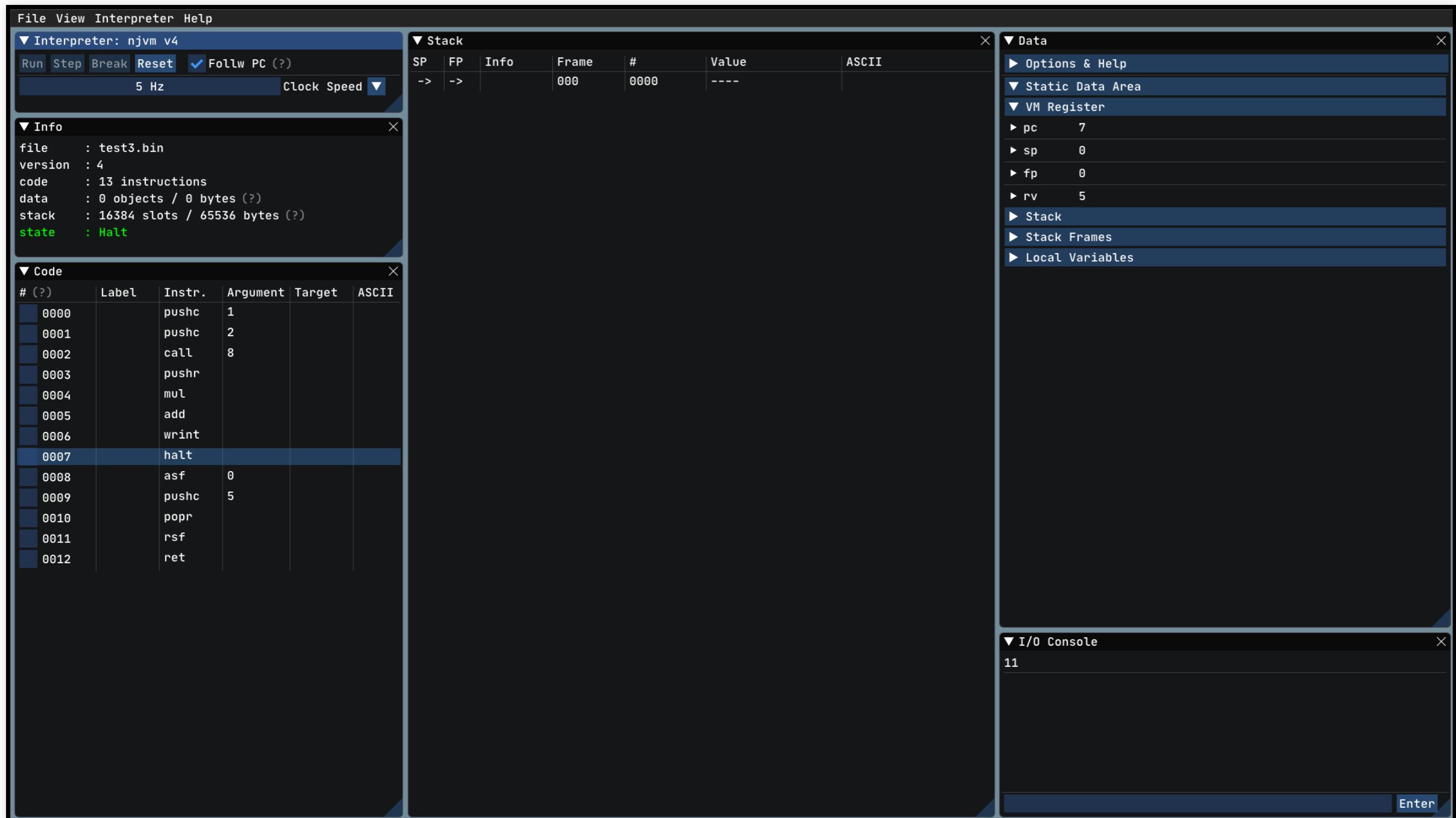
# Beispiel ohne Argumenten, mit Rückgabewert



# Beispiel ohne Argumenten, mit Rückgabewert



# Beispiel ohne Argumenten, mit Rückgabewert



# Funktionsaufruf mit Argumenten, mit Rückgabewert

Der Aufruf einer Funktion **mit Argumenten und mit einem Rückgabewert**, ist die Kombination der zuletzt gezeigten Aufrufe.

Caller:

```
push arg0  
push arg1  
...  
push arg(n-1)  
call Funktionslabel  
drop n  
pushr
```

Callee:

```
Funktionslabel:  
  asf <numLocalVars>  
  <body>  
  <push retval> // Vorbereitung für popr  
  popr  
  rsf  
  ret
```

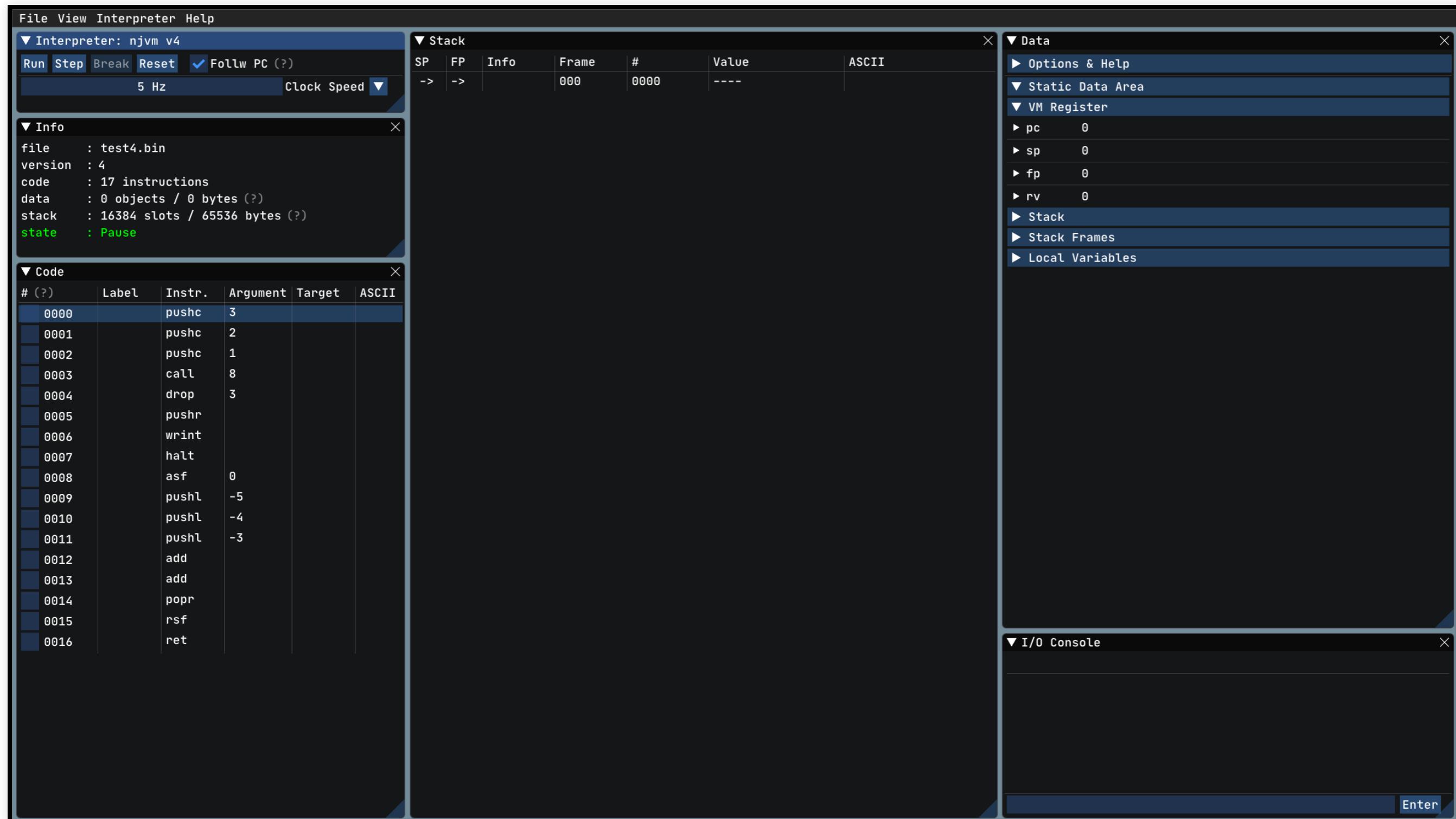
Zugriff auf Argumente erfolgt mit `pushl` und `pushl` mittels negativen Immediate Wert ( $-2-n+i$ ) für  $i$ -tes Argument

# Beispiel: Funktionsaufruf mit Argumenten und Rückgabewert

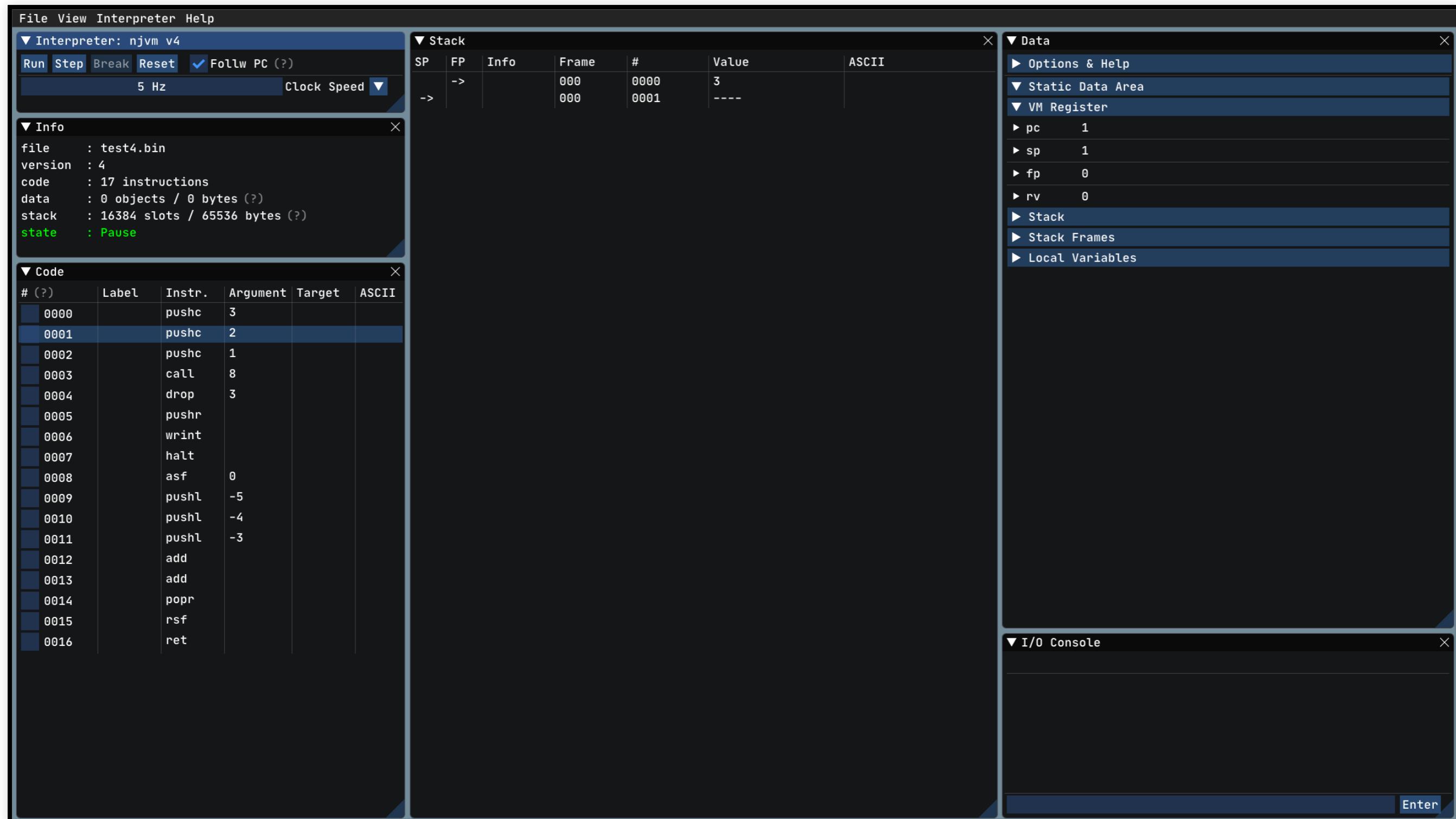
Beispiel: Ausgabe von `wrint f(3,2,1)` mit `int f(a,b,c) {return a+b+c;}`

```
pushc 3
pushc 2
pushc 1
call f
pushr
drop 3
wrint
halt
f:
    asf 0
    pushl -5
    pushl -4
    pushl -3
    add
    add
    popr
    rsf
    ret
```

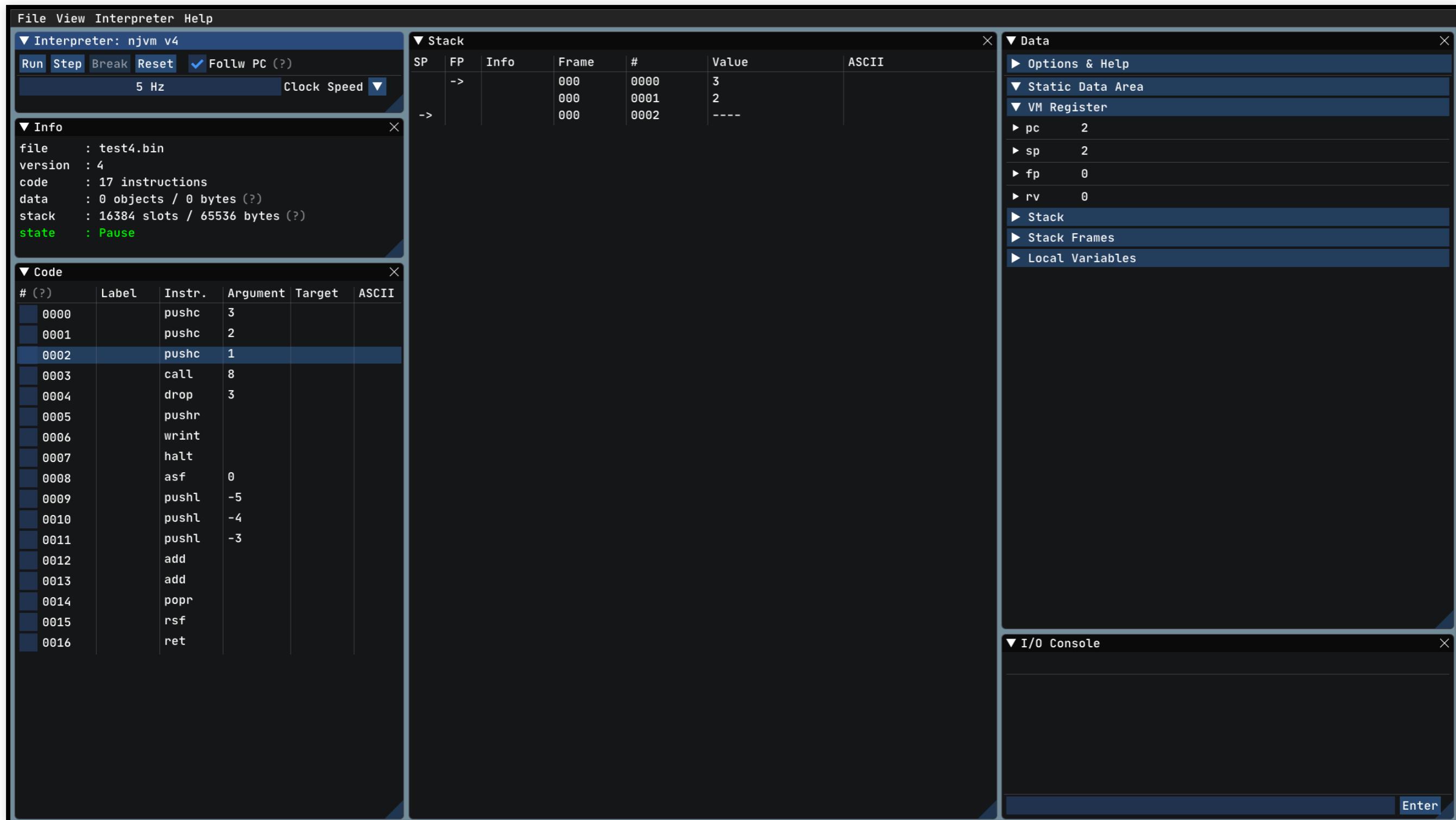
# Beispiel mit Argumenten und Rückgabewert



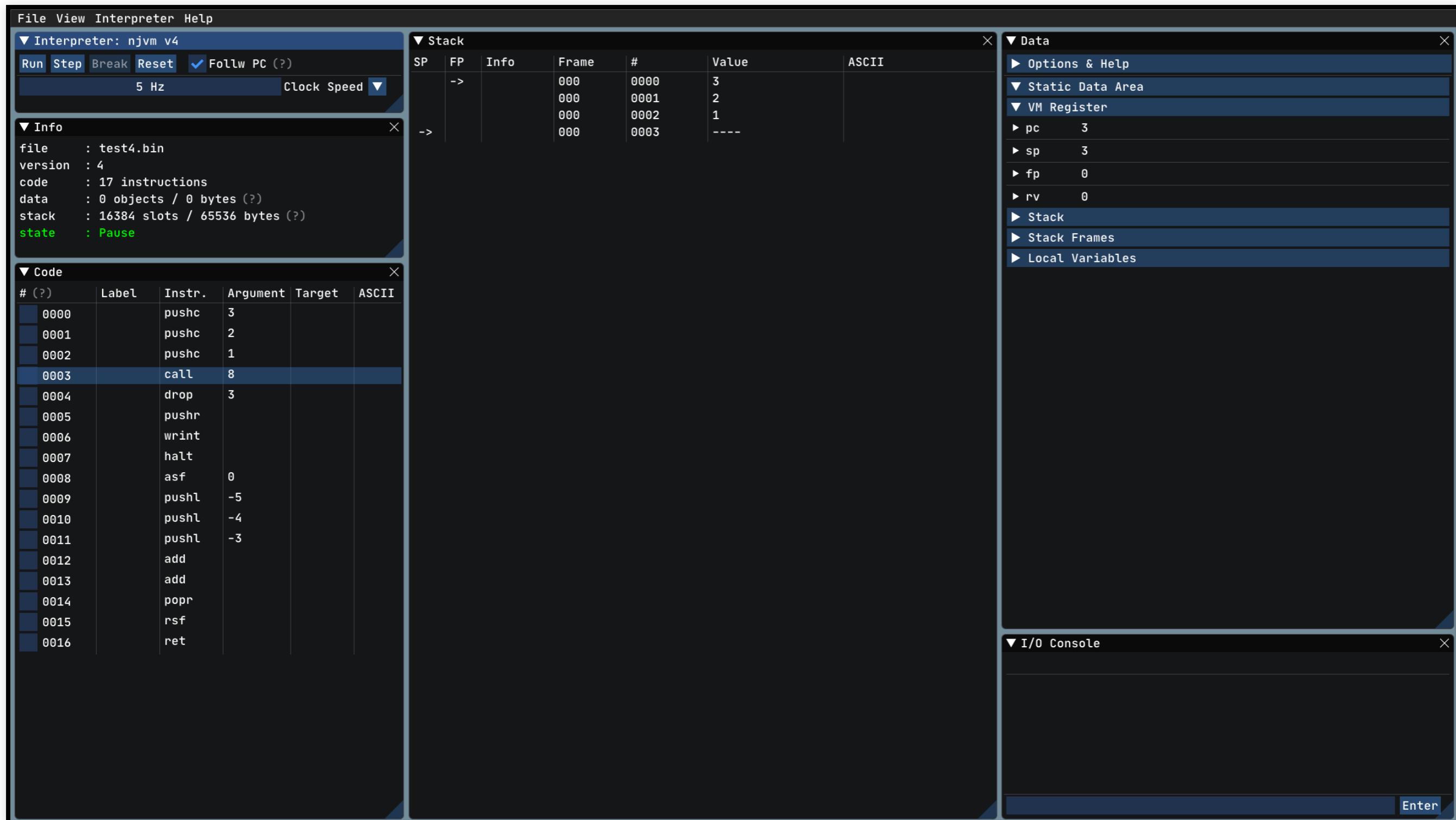
# Beispiel mit Argumenten und Rückgabewert



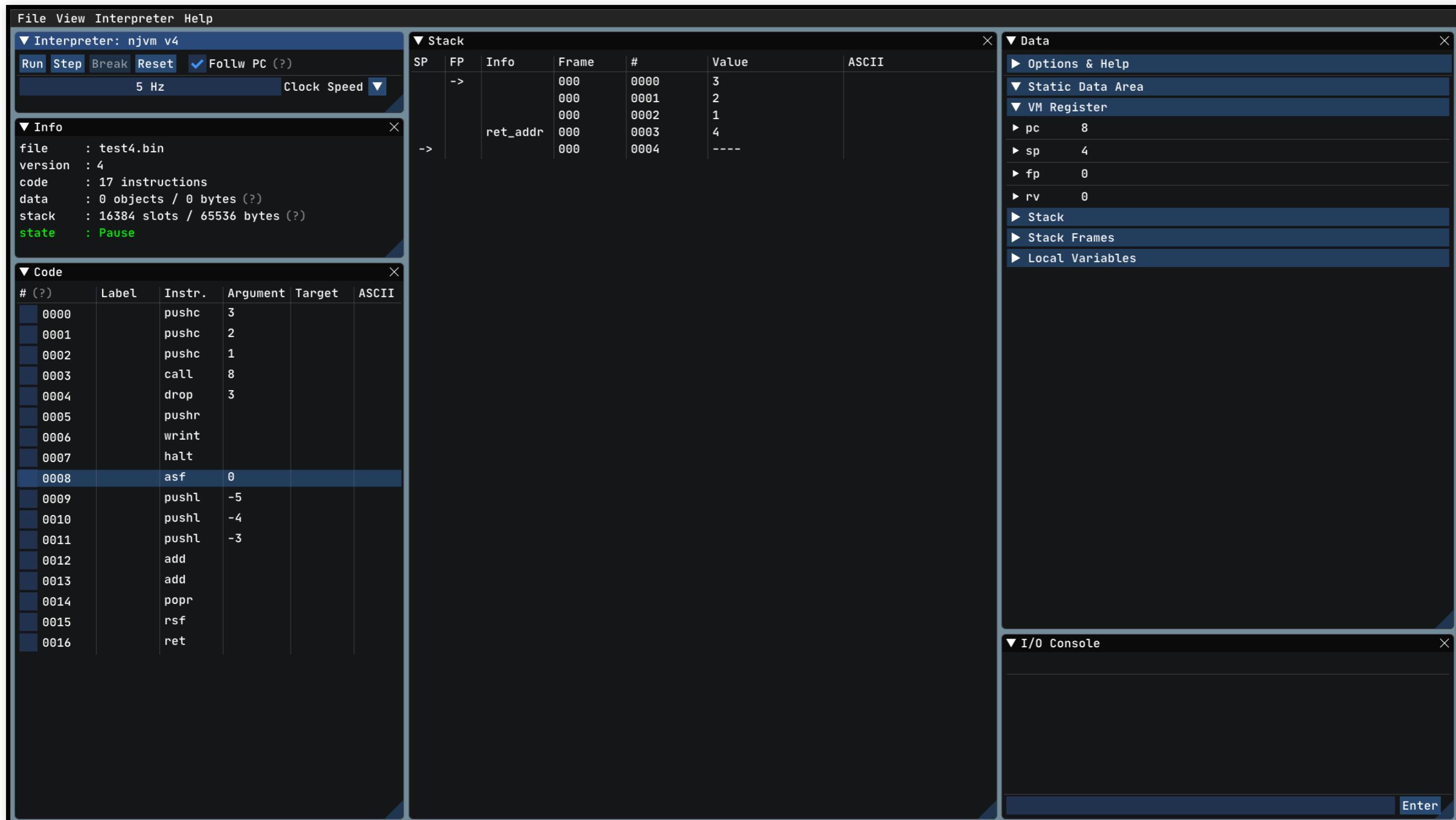
# Beispiel mit Argumenten und Rückgabewert



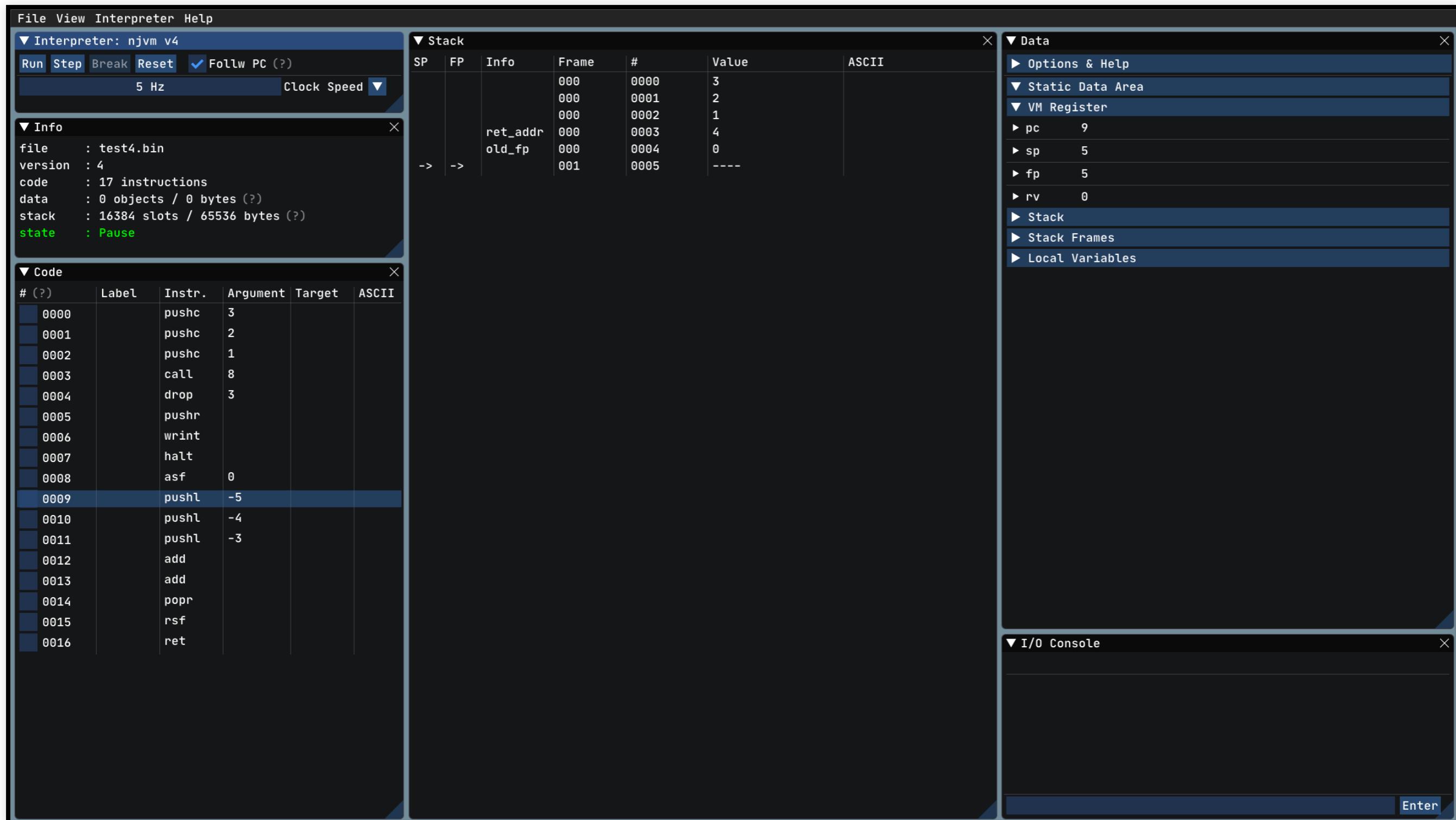
# Beispiel mit Argumenten und Rückgabewert



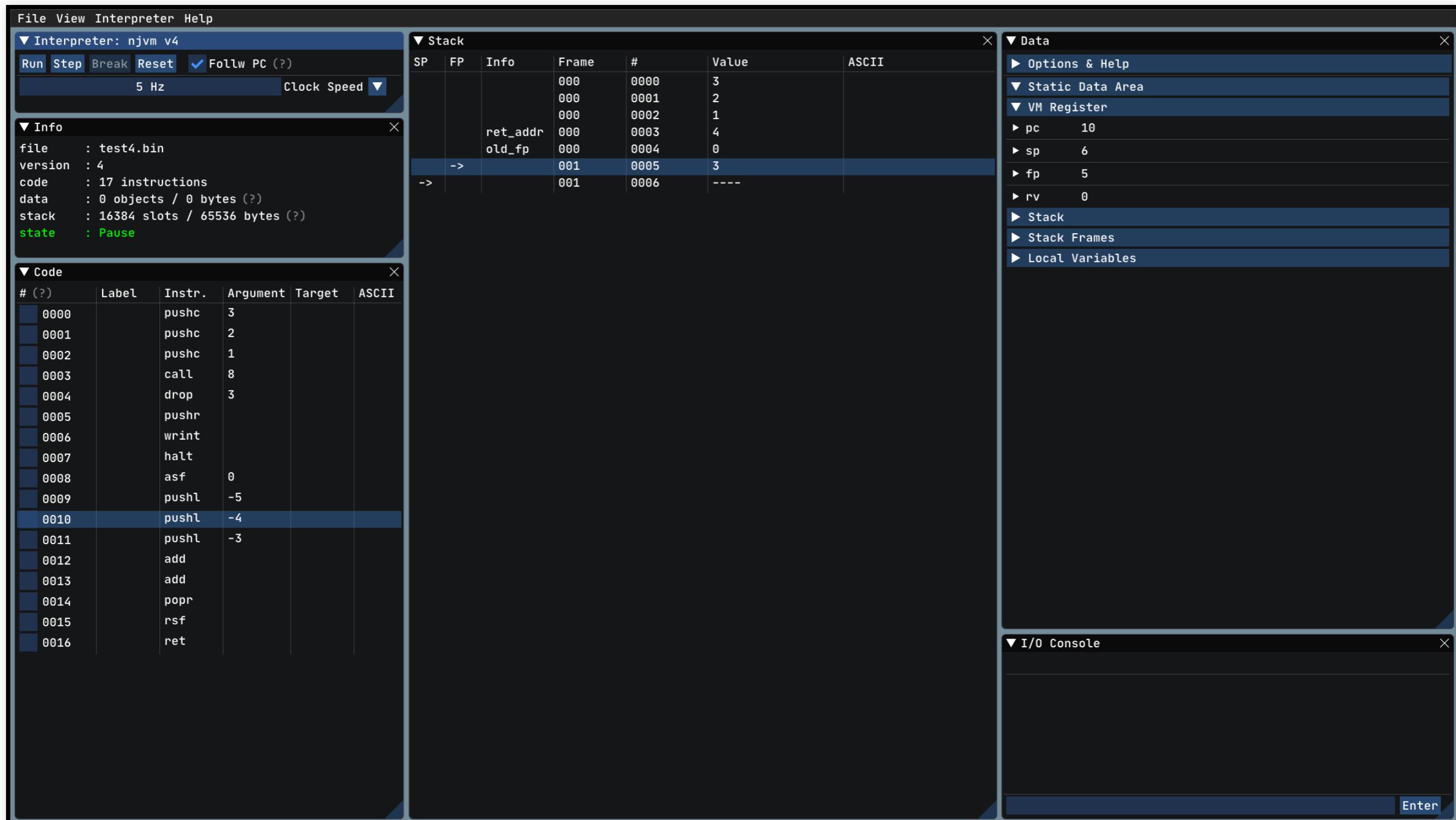
# Beispiel mit Argumenten und Rückgabewert



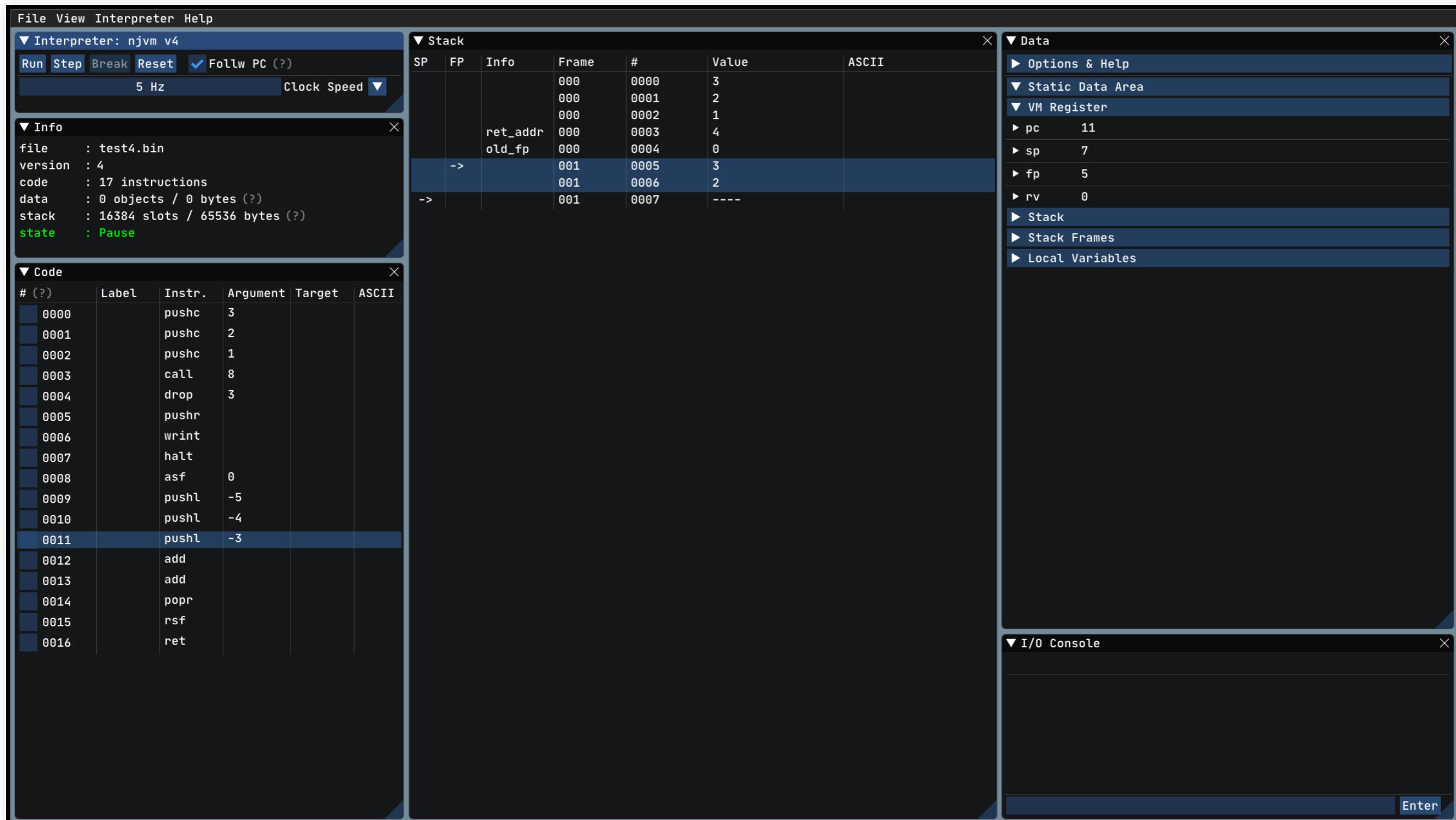
# Beispiel mit Argumenten und Rückgabewert



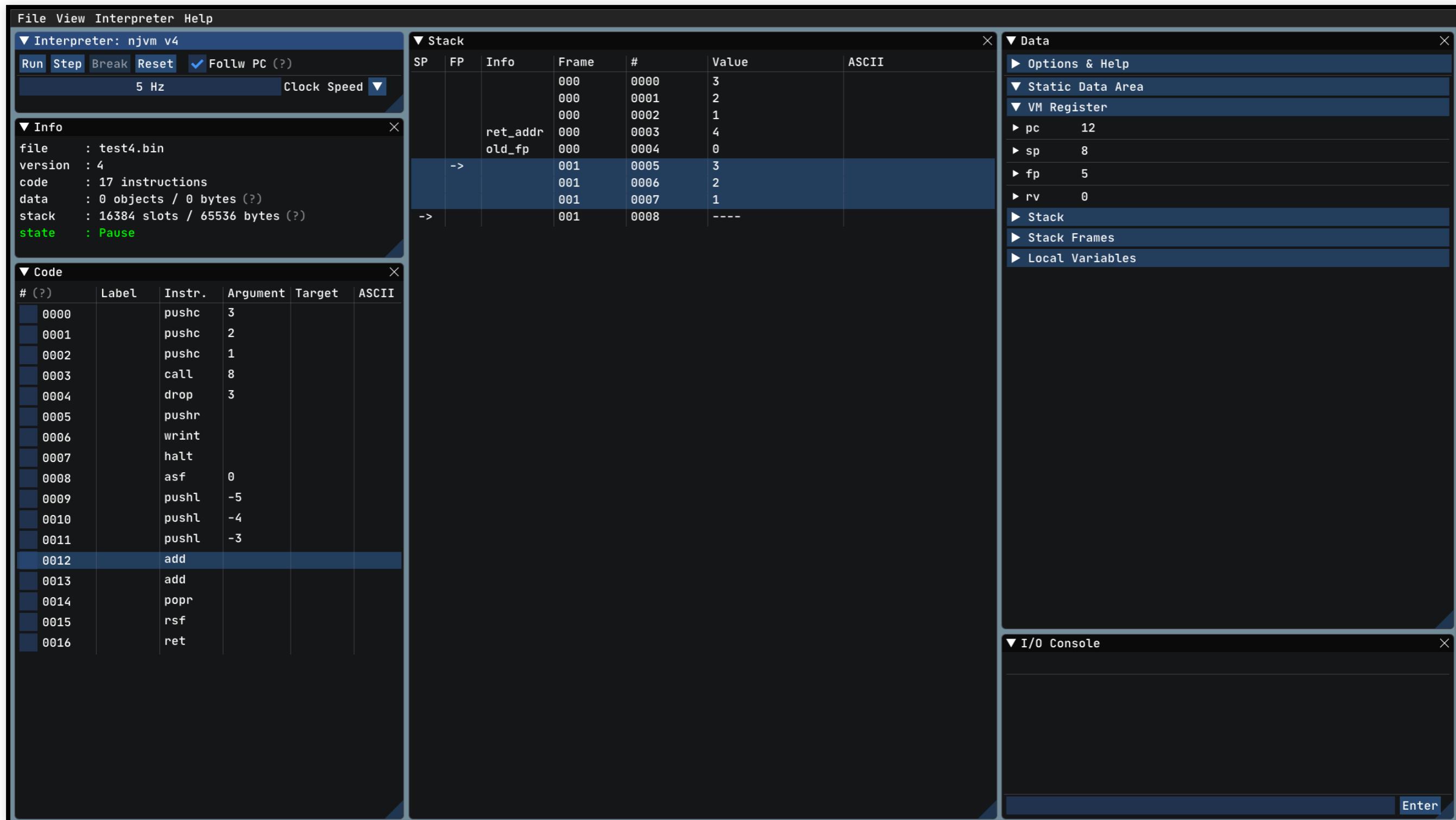
# Beispiel mit Argumenten und Rückgabewert



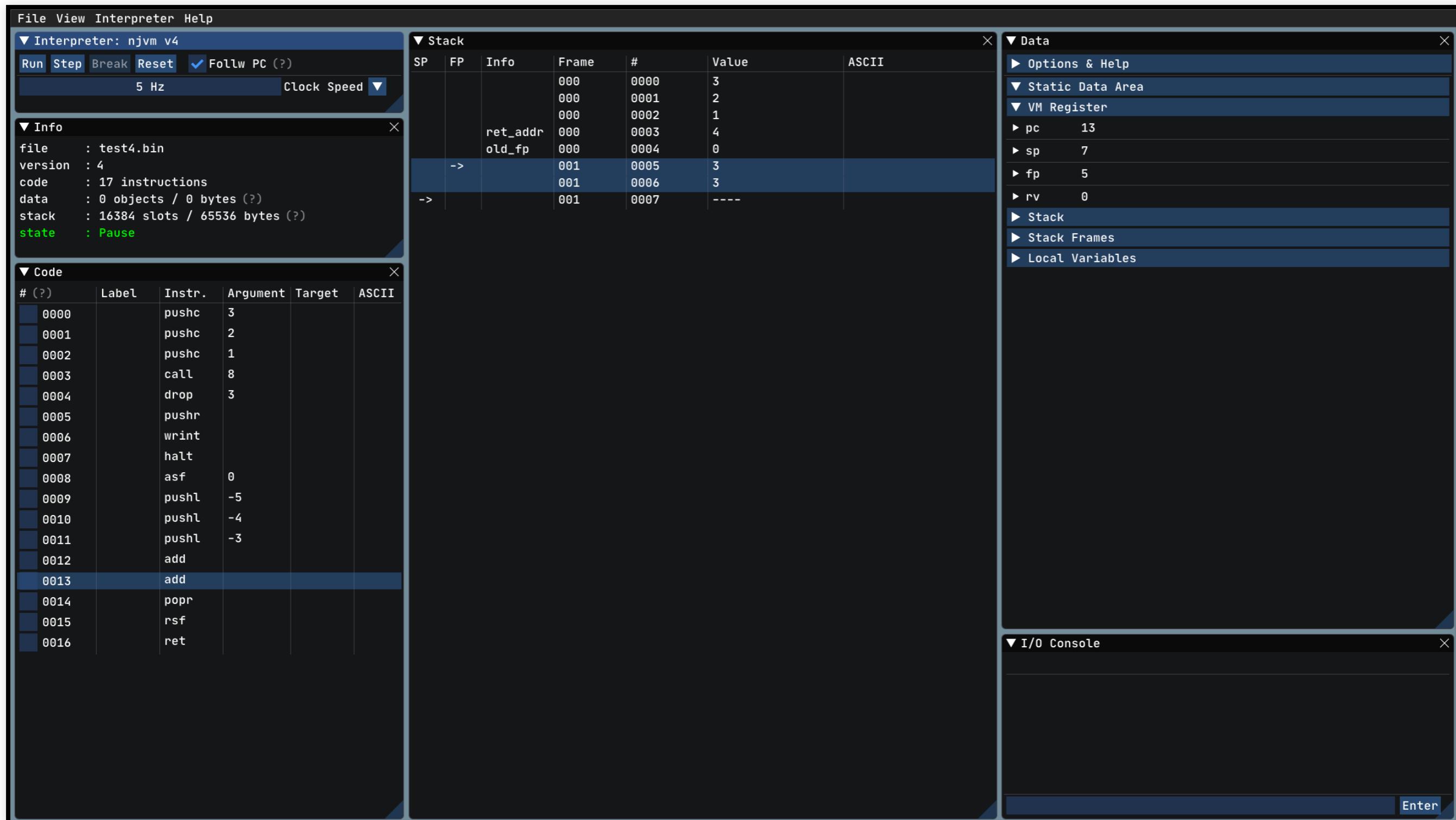
# Beispiel mit Argumenten und Rückgabewert



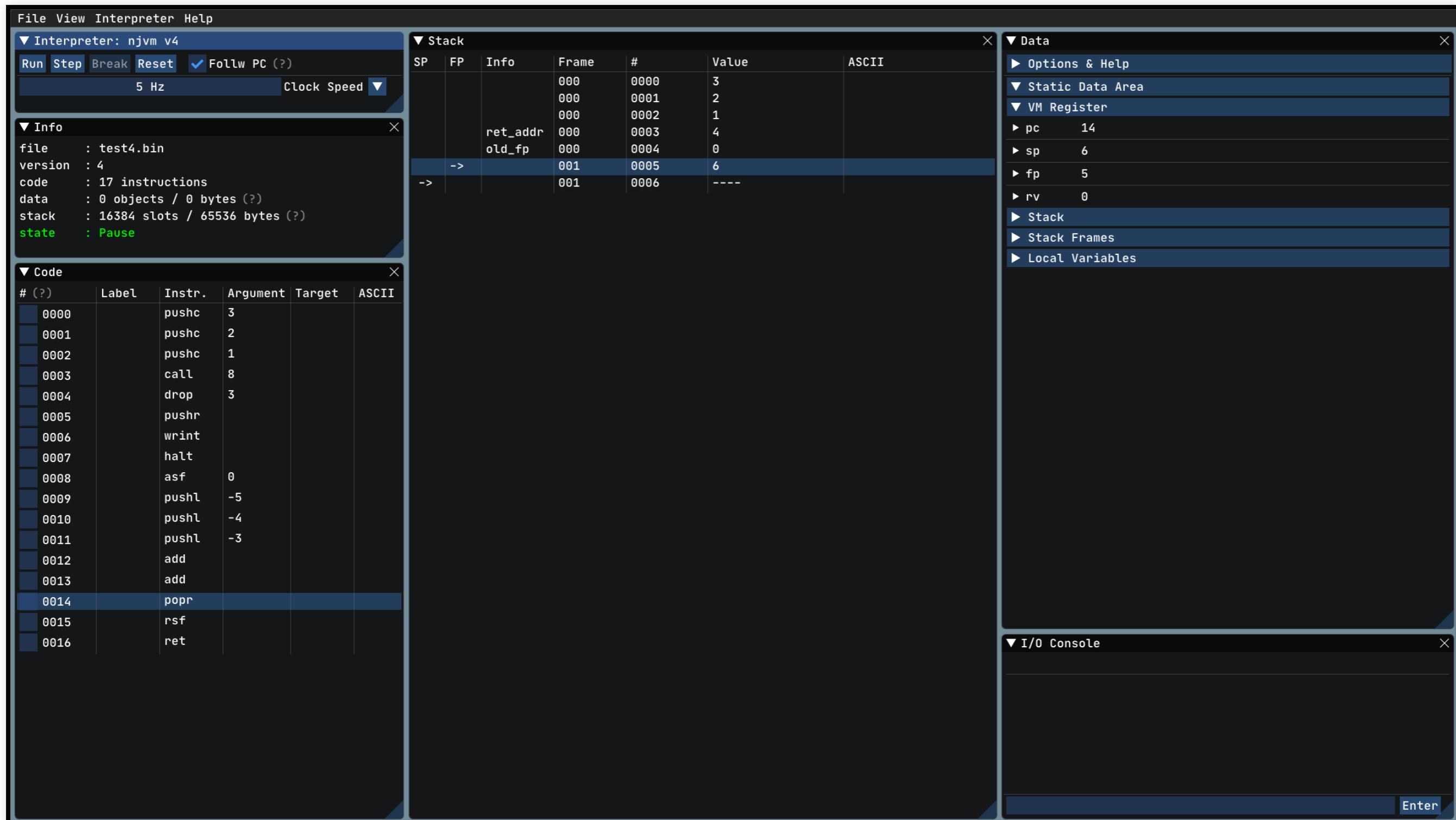
# Beispiel mit Argumenten und Rückgabewert



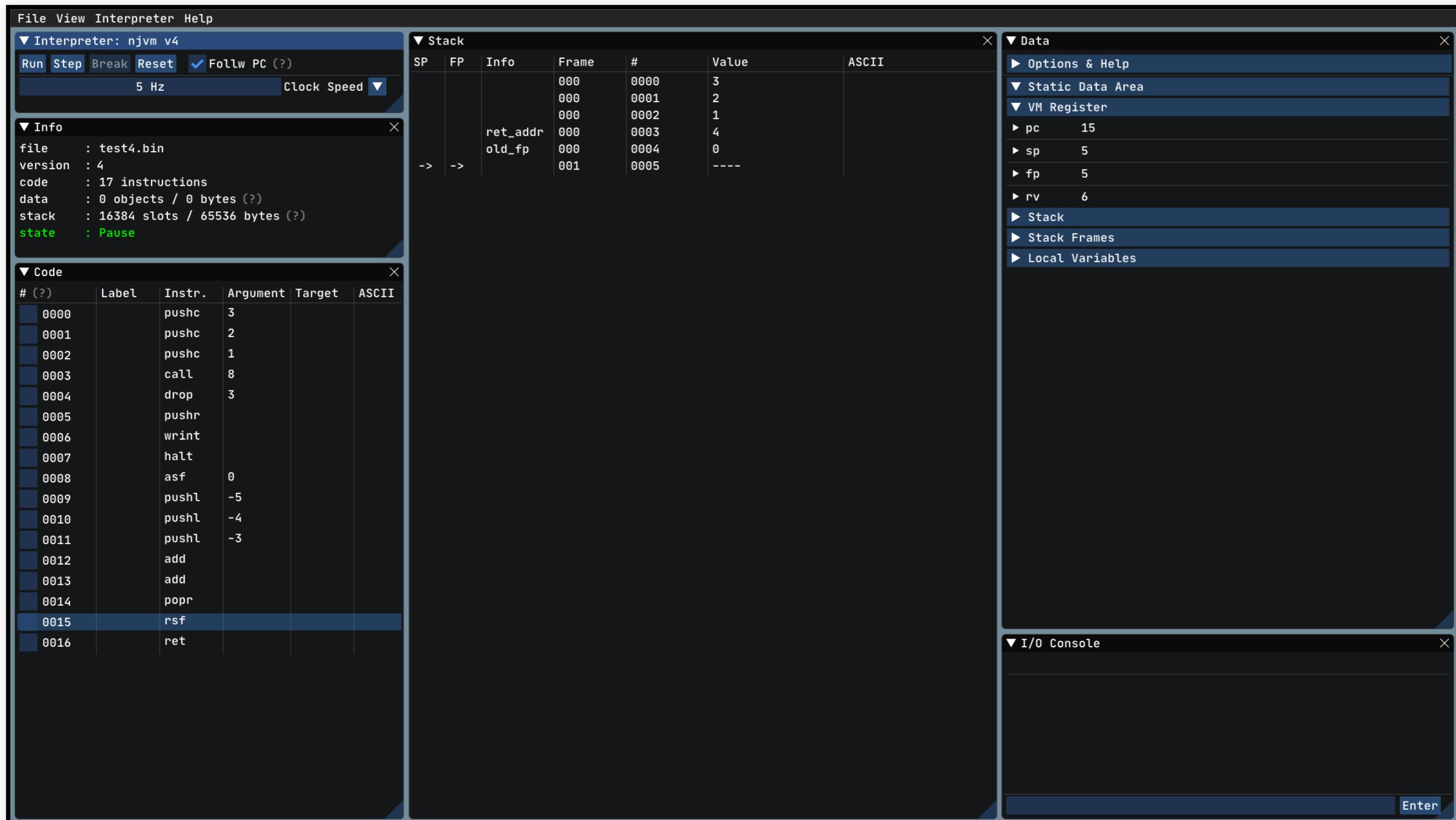
# Beispiel mit Argumenten und Rückgabewert



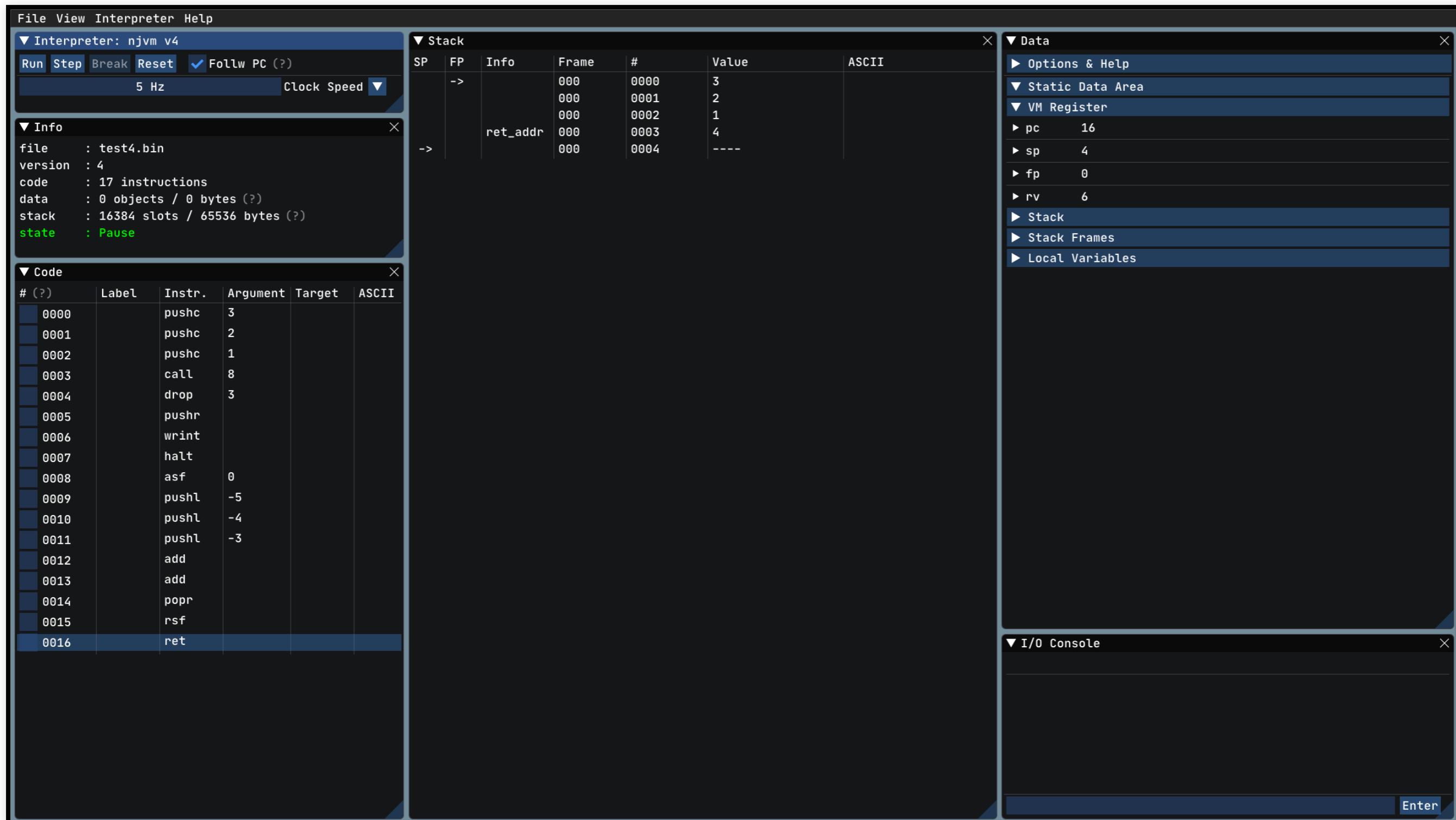
# Beispiel mit Argumenten und Rückgabewert



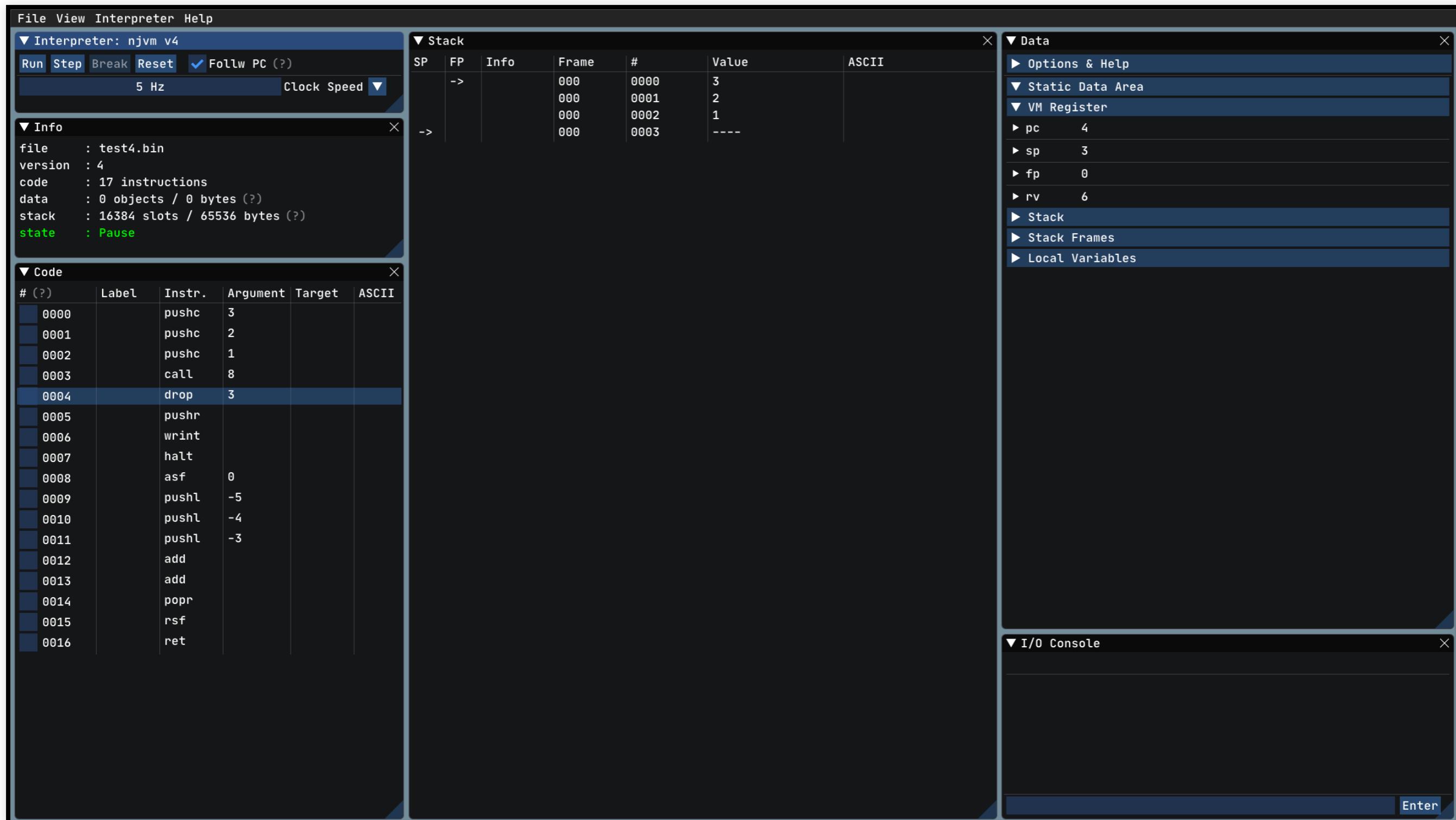
# Beispiel mit Argumenten und Rückgabewert



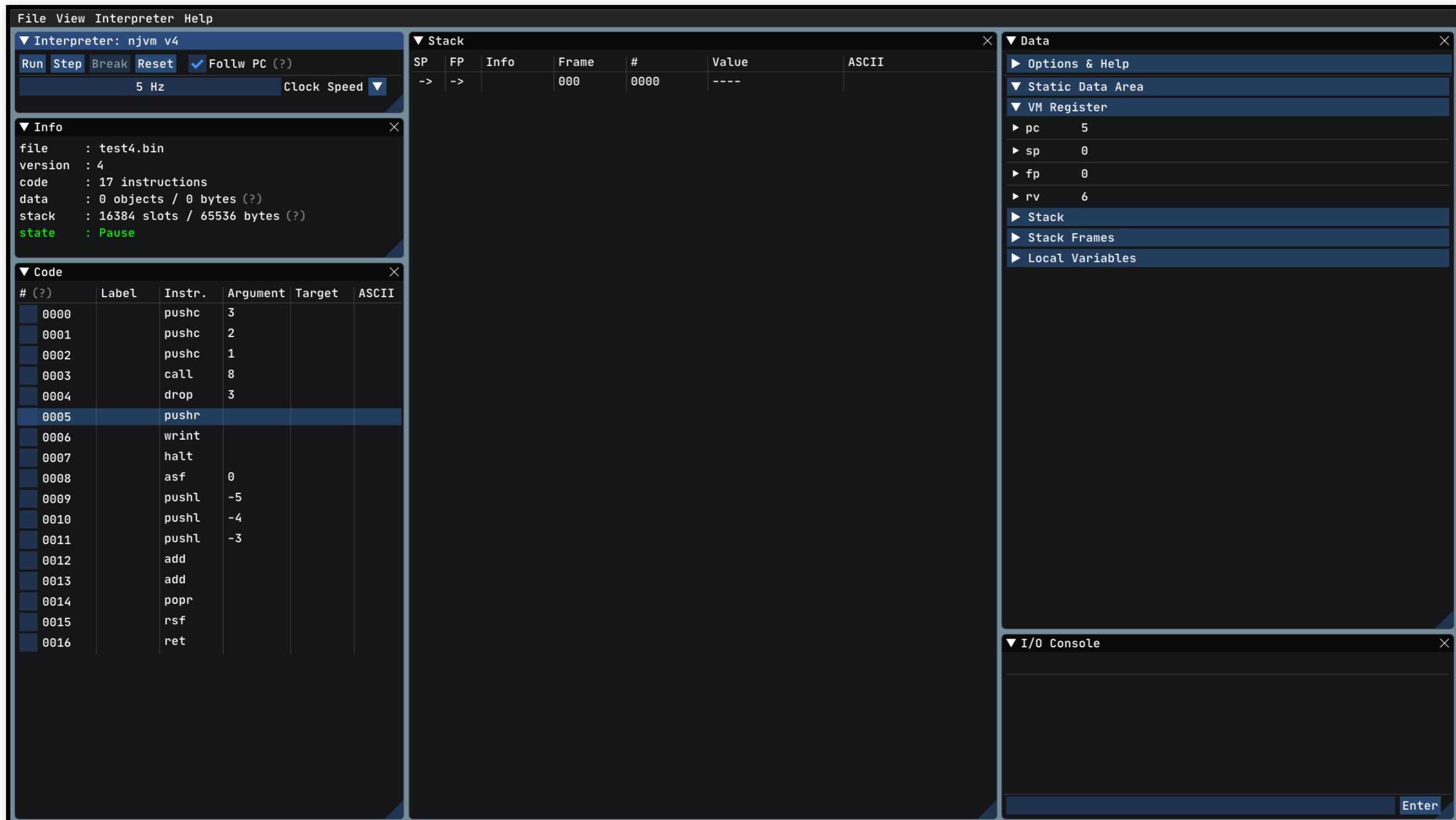
# Beispiel mit Argumenten und Rückgabewert



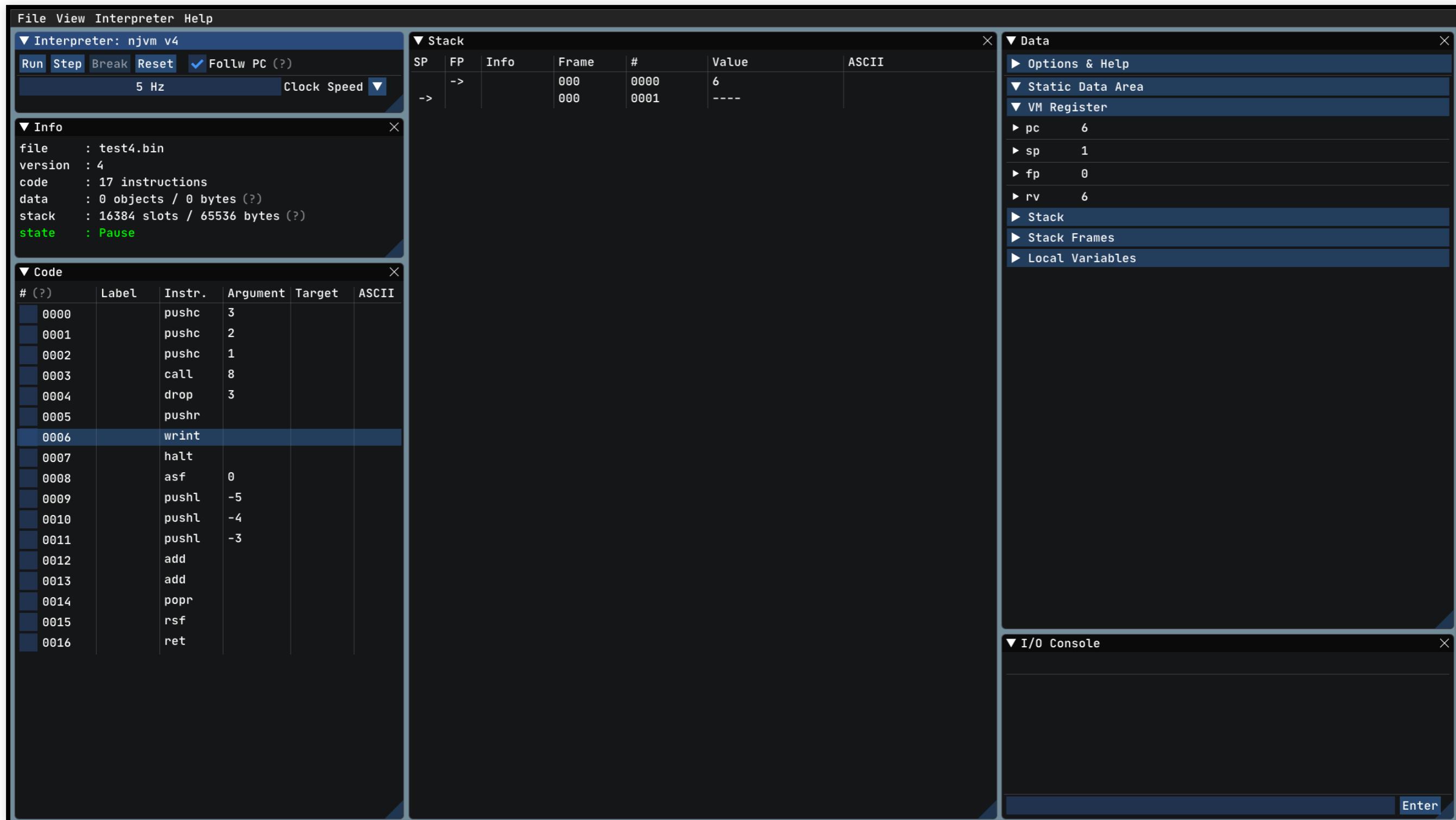
# Beispiel mit Argumenten und Rückgabewert



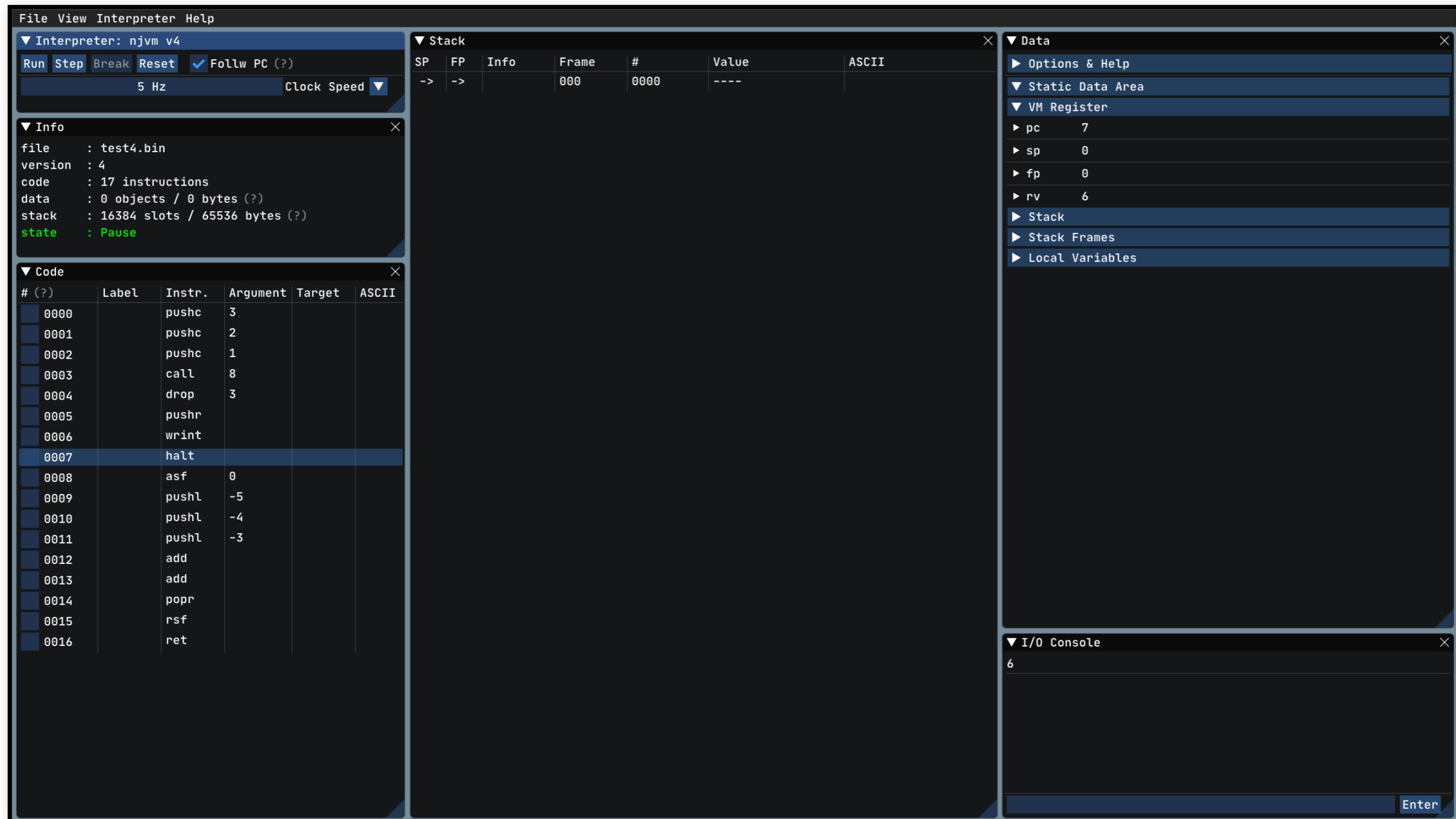
# Beispiel mit Argumenten und Rückgabewert



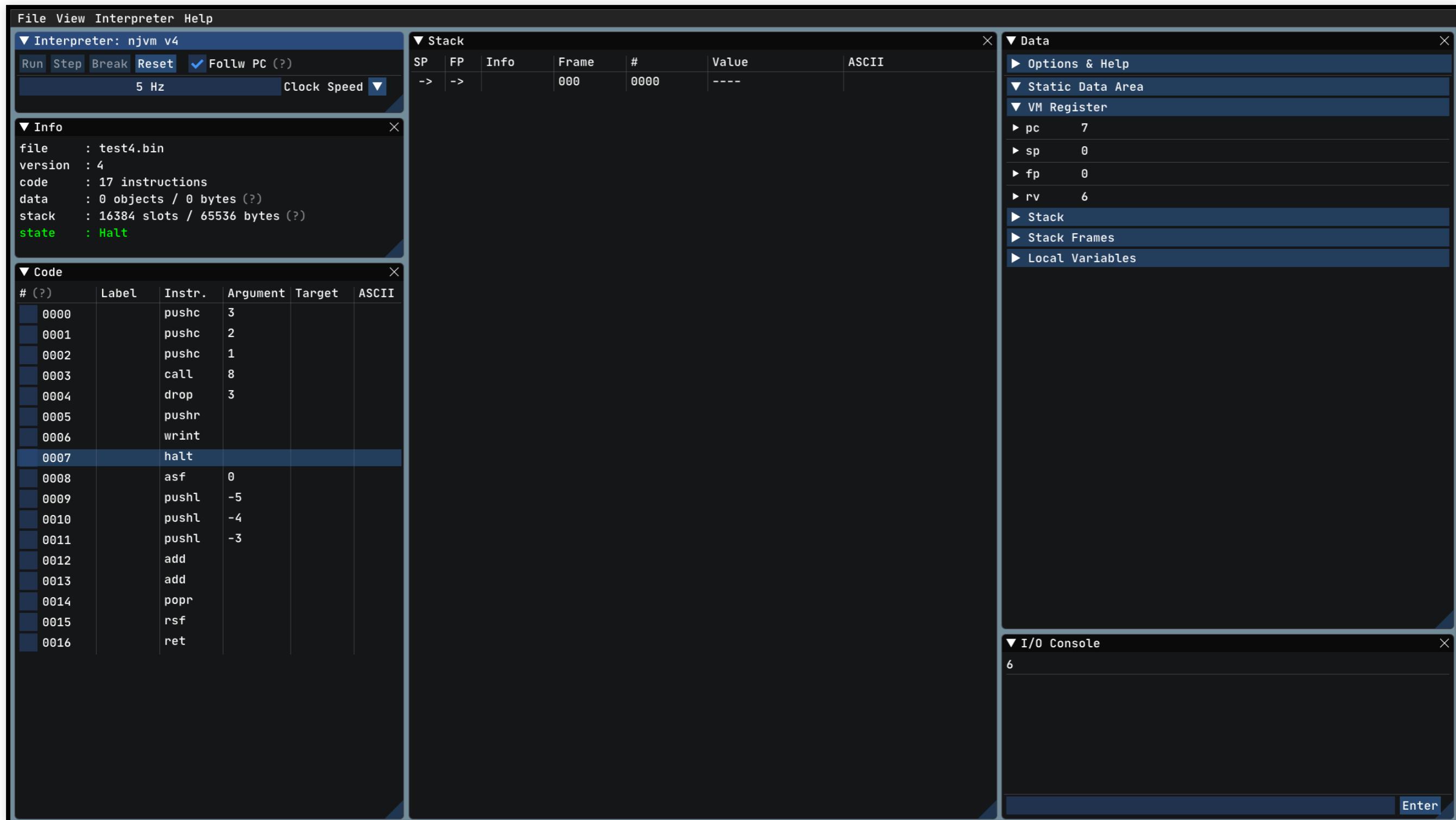
# Beispiel mit Argumenten und Rückgabewert



# Beispiel mit Argumenten und Rückgabewert



# Beispiel mit Argumenten und Rückgabewert



# Übersicht NinjaVM

