Einführung in die Informatik II
Maschinenahe Programmierung: Rechenstrukturen

Prof. Bernd Brügge, Ph.D
Institut für Informatik
Technische Universität München

Sommersemester 2004

8. Juli 2004

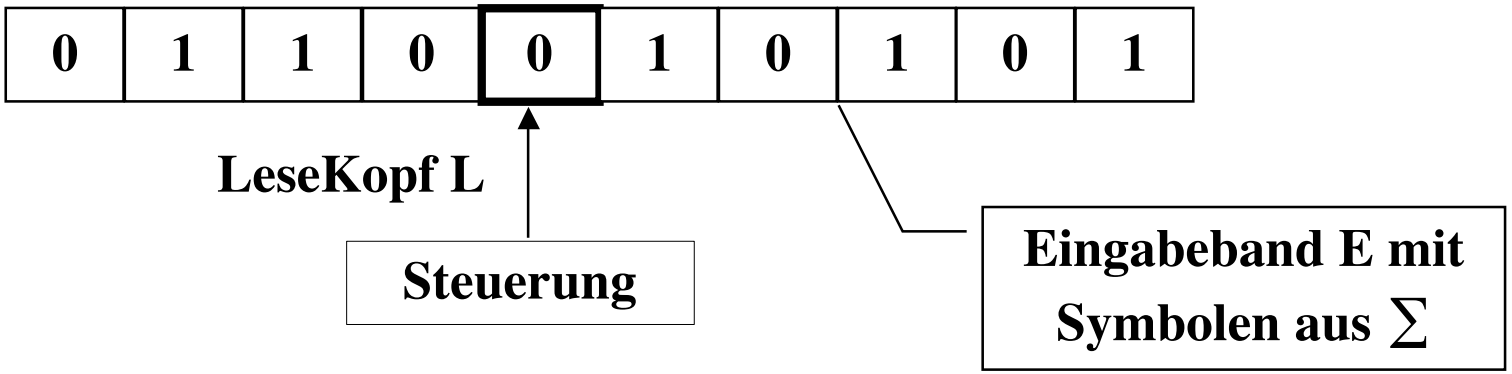
Organisatorisches: Planung der restlichen Vorlesungen

- Heute: Überleitung zu maschinennaher Programmierung
  - Übersetzung von Java in eine maschinenahe Sprache
- 13. Juli: Vorlesung fällt aus
- 15. Juli: Maschinennahe Programmierung
  - Laufzeitkeller, Unterprogrammaufruf, Halde
- 20. Juli: Codierung
- 20. Juli: Zentralübung: Fragestunde zur Klausur.
  - Alle Fragen bis zum 19. Juli per E-Mail an herzog@in.tum.de werden berücksichtigt.
- 22. Juli: Vergleich von Modellierungs-&Programmierkonzepten, Abschluss
- 24. Juli: Klausur, zwischen 13:00 - 16:00,
  - Klausurmerkblatt wird am 20. Juli verteilt

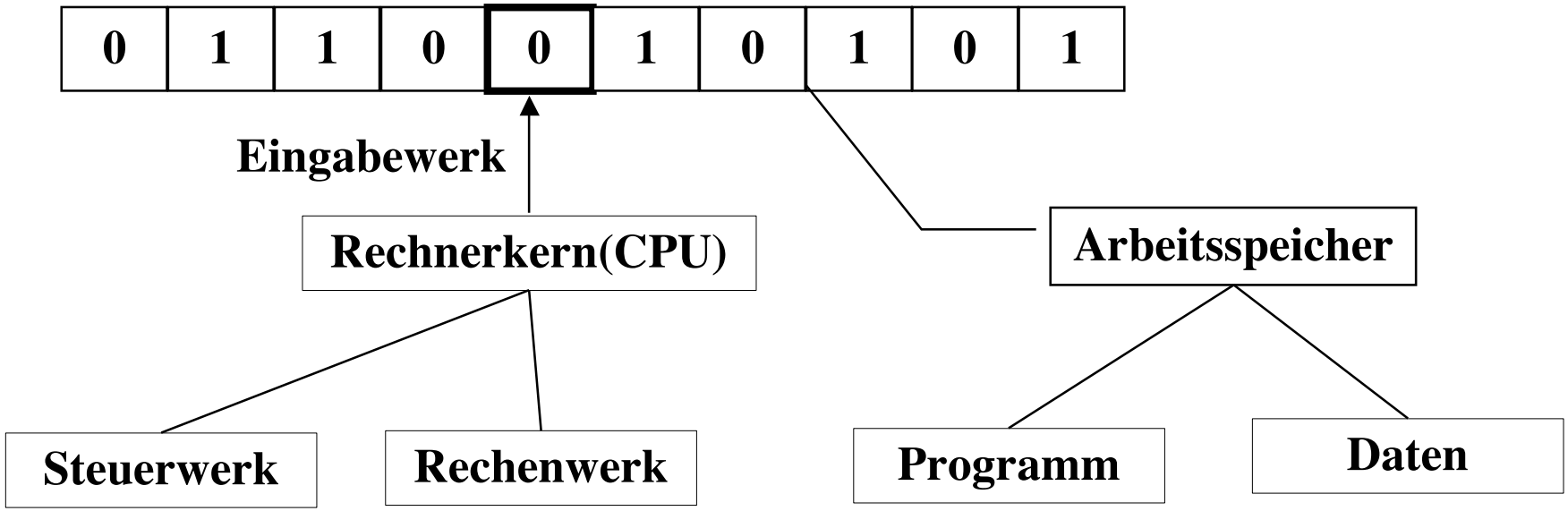
Gliederung dieses Vorlesungsblockes

- **Teil 1: Struktur von Rechenanlagen**
  - Von endlichen Automaten zur von-Neumann-Maschine
  - Aufbau der von-Neumann-Maschine
  - Modellierung einer primitiven von-Neumann-Maschine: PMI
  - Konzept der Programmsteuerung
  - Assembler
- **Teil 2: Übersetzung von Konstrukten höherer Programmiersprachen in maschinennahe Sprachen**
  - Ausdrücke , Zuweisung, While-Schleife
  - Struktur des Laufzeitstapels
  - Methodenaufruf ("Unterprogrammsprung"),
  - Rekursion
  - Rekursive Datenstrukturen

Vom endlichen Automaten ...



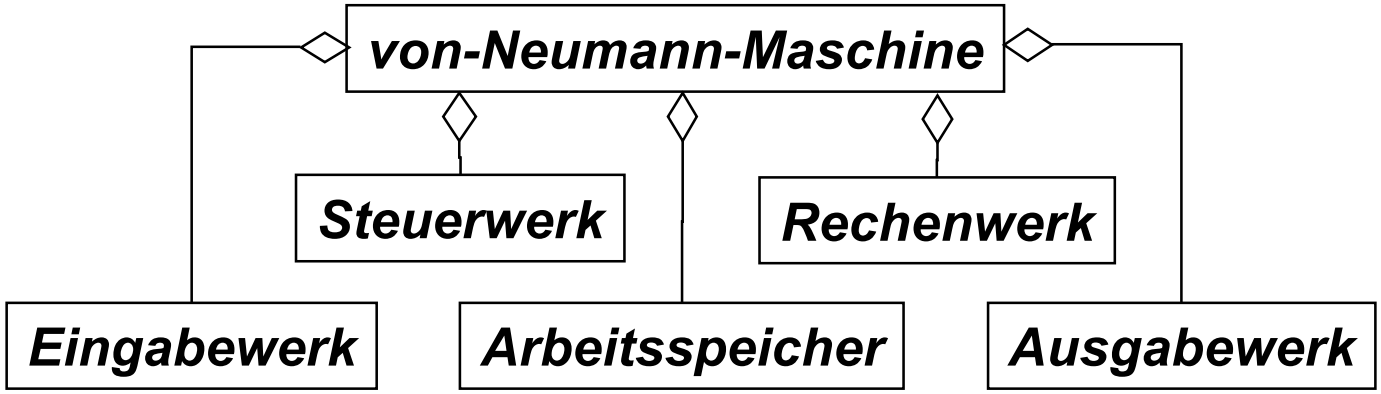
Vom endlichen Automaten zum Rechner



Aufbau von Rechenanlagen

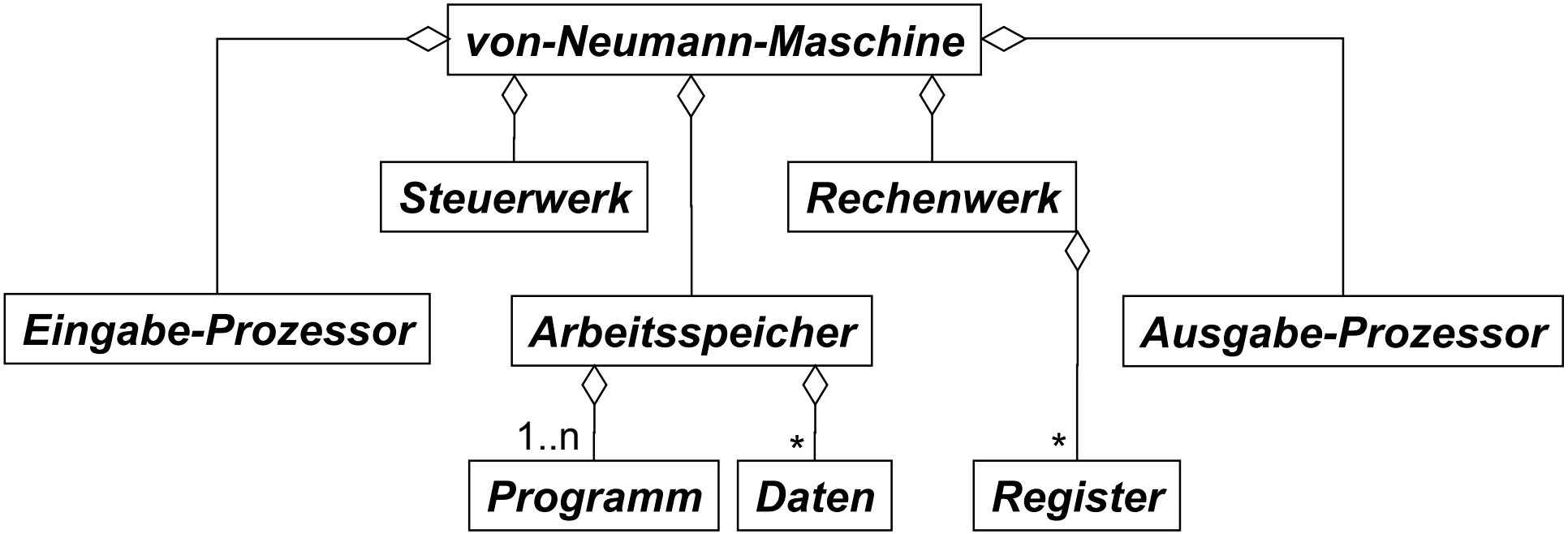
- Im folgenden besprechen wir den Aufbau von Rechenanlagen, wie sie prinzipiell zur Ausführung von Informatik-Systemen verwendet werden. Wir arbeiten dabei einige wichtige Konzepte heraus, die vielen Rechenanlagen gemeinsam sind:
  - ***Die von-Neumann-Maschine***
  - ***Das Prinzip der Programmsteuerung***
  - ***Speicherverwaltung***
- Zur Erklärung der Konzepte haben wir nicht eine kommerziell verfügbare Rechenanlage gewählt, sondern eine hypothetische Rechenanlage namens PMI.
  - Anhand von PMI sind die Konzepte leichter zu verstehen.
  - Die Konzepte gelten für die meisten heute kommerziell verfügbaren Rechenanlagen.

Die Komponenten einer von-Neumann-Maschine



- **Steuerwerk**: Steuerung des Ablaufs der Operationen eines Programms. Wird auch ***Befehlsprozessor*** genannt.
- **Rechenwerk**: Ausführung von Rechenoperationen, die während des Ablaufs durchgeführt werden müssen. Auch ***Datenprozessor*** genannt.
- **Arbeitsspeicher**: Speicherung eines Programms und seiner Daten
- **Eingabewerk**: Einlesen des Programms und der Daten in den Arbeitsspeicher (Eingabeoperationen). Auch ***Eingabe-Prozessor.***
- **Ausgabewerk**: Ausgabe von Daten aus dem Arbeitsspeicher (Ausgabeoperationen). Auch ***Ausgabe-Prozessor***.

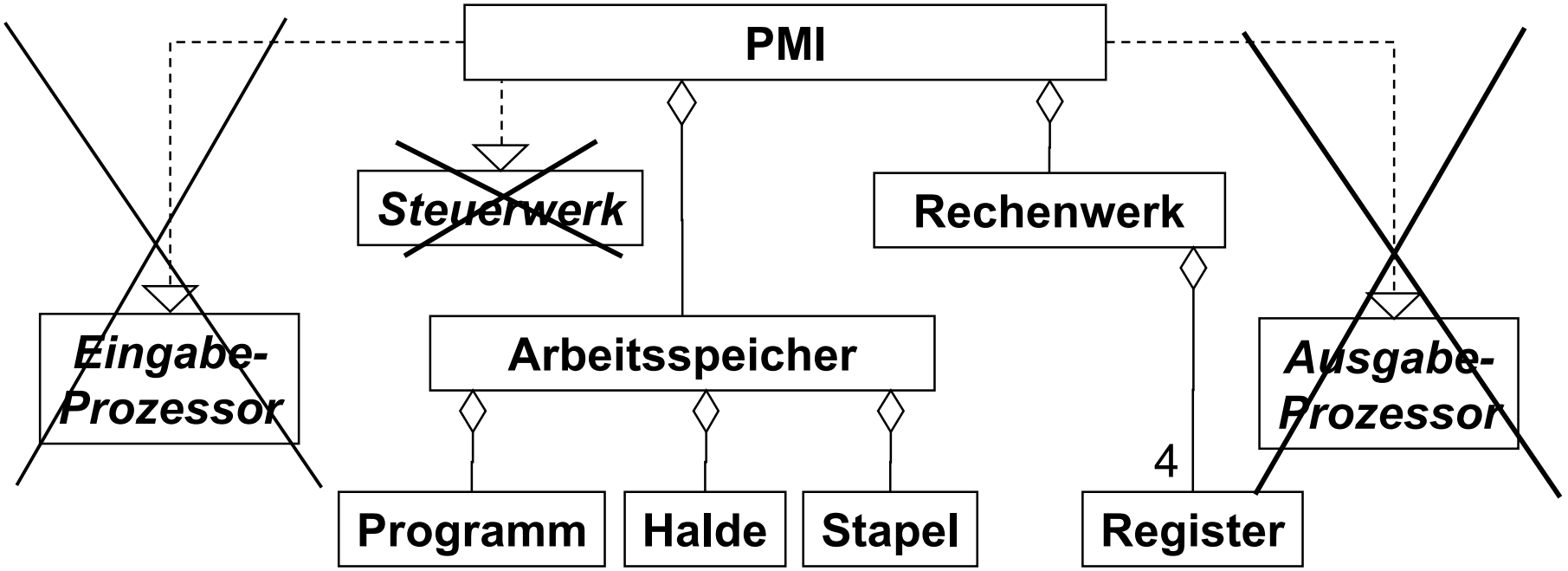
Wichtige Eigenschaften von von-Neumann-Maschinen



- **Programme und Daten** werden im Arbeitsspeicher gehalten
- Die Abarbeitung von Programmen erfolgt zustandsorientiert. Sie ist
  - abhängig von den Werten im **Arbeitsspeicher**
  - abhängig von den Werten der **Register**

Beispiel einer von-Neumann-Maschine: PMI

- Als Beispiel einer von-Neumann-Maschine führen wir nun ***PMI*** (*Primitive Maschine für die Informatik-Ausbildung*) ein:



- Info III benutzt ***MI***. Die PMI ist weniger leistungsfähig (*"primitiver"*) als die MI, aber einfacher zu programmieren.

Entwicklung von Rechenanlagen

- Rechenanlagen sind Teil von Informatik-Systemen.
  - Man kann ***Rechenanlagen*** also ***modellieren*** und ***implementieren***.
- Es gibt viele Spezial-Sprachen (z.B. VHDL), mit denen man Rechenanlagen auf Analyse- und Entwurfsniveau modellieren kann.
  - Vertiefung im Hauptstudium: Rechnerarchitekturen (Bode)
- Im ***Implementierungsmodell*** ist eine Rechenanlage oft ein umfangreiches Schaltwerk mit vielen Komponenten. Das Schaltwerk wird dann mit Hardware-Komponenten realisiert:
    - Schaltwerke und ihre Realisierung wurden in TGI behandelt.
- Für die ***Modellierung*** der ***PMI-Maschine*** nehmen wir ***UML:***
  - Die entsprechenden Modelle führen wir heute schrittweise ein.
- Für die ***Implementierung*** von ***PMI*** haben wir keine Hardware-Komponenten benutzt, sondern ***Java***.
  - Dies bedeutet natürlich, dass PMI auf einer anderen Rechenanlage ausgeführt werden muss, die Java-Code ausführen kann.

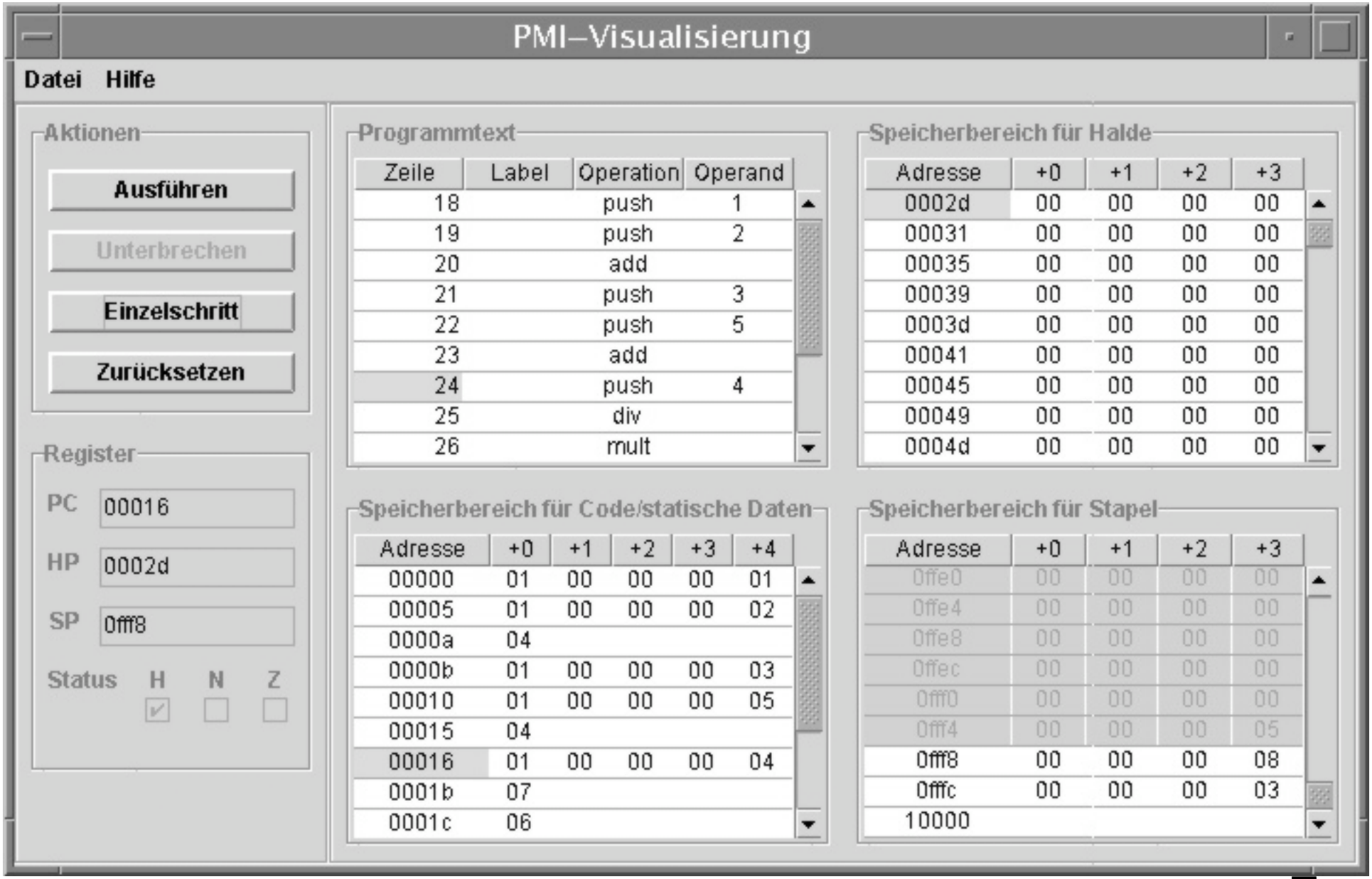
Die PMI-Maschine als Modell

- Die PMI ist ein von-Neumann-Rechner.
- Die PMI ist auch ein Informatik-System. Wir können deshalb ihre funktionellen, statischen und dynamischen Aspekte beschreiben.
  - ***Funktionelles Modell:***
    - Operationen zum Betreiben der Rechenanlage und Operationen ("Maschinenbefehle") zum Ausführen von Programmen.
  - ***Statisches Modell***:
    - Speicher, Prozessor, Adressregister, Statusregister, und die Semantik von Maschinenbefehlen
  - ***Dynamisches Modell:***
    - Arbeitsweise des Steuerwerkes, insbesondere die Exekution von Maschinenbefehlen.
- Zur Veranschaulichung der PMI-Komponenten wählen wir außerdem eine geeignete ***Sicht:*** $\Rightarrow$ PMI-Visualisierung

Funktionelle Anforderungen an die Steuerung der PMI-Maschine

- ***Beginnen und Beenden der Visualisierung:***
  - **Start**: Visualisierung wird gestartet.
  - **Beendigung der PMI-Visualisierung:** Visualisierung wird beendet
- ***Laden von PMI-Programmen:***
  - **Programm laden:** Lädt ein in einer externen Datei gespeichertes PMI-Programm in den Arbeitsspeicher. Der Arbeitsspeicher wird initialisiert.
  - **Programm erneut laden:** Setzt das PMI-Programm wieder auf den Anfangszustand. Der Arbeitsspeicher wird initialisiert.
- ***Exekution von PMI-Programmen:***
  - **Ausführen*:*** Beginnt die Exekution eines PMI-Programms.
  - **Unterbrechen*:*** Stoppt die Exekution des Programms.
  - **Einzelschritt*:*** Die nächste PMI-Instruktion wird ausgeführt
  - **Zurücksetzen*:*** Das gerade geladene Programm wird auf die erste Instruktion zurückgesetzt. Arbeitsspeicher wird ***nicht*** neu initialisiert.

Graphische Benutzerschnittstelle der PMI-Maschine



Maschinenprogramm und Befehlssatz

- Hauptzweck von Rechenanlagen ist die Ausführung von Maschinenprogrammen.
- **Definition:** Ein ***Maschinenprogramm*** ist eine Folge von Maschinenbefehlen, d.h. Operationen, die von der Rechenanlage ausgeführt werden können.
- **Definition:** Der ***Befehlssatz*** einer Rechenanlage ist die Menge der in einem Maschinenprogramm verwendbaren Maschinenbefehle.
- Allgemein unterscheiden wir folgende Arten von Maschinenbefehlen:
  - Transport- und Ladebefehle
  - Arithmetische Befehle (für Festkommaarithmetik und Gleitkommarithmethik)
  - Logische Operationen (auf Binärwortern)
  - Operationen mit Adressen
  - Schiebebefehle
  - Programmablaufbefehle, Sprungbefehle, Unterprogrammbefehle
  - Ein/Ausgabebefehle
  - Steuerungsbefehle (Priviligierte Befehle)

Funktionelle Anforderungen an PMI-Programme

- ***Für den PMI-Befehlssatz definieren wir insgesamt 14 Befehle:***
  - **1 Programmablaufbefehl**: Zum Anhalten der Maschine
  - **3 Transport und Ladebefehle**: Zur Manipulation des Stapels im Arbeitsspeicher
  - **4 arithmetische Befehle**: Plus, Minus, Division, Multiplikation
  - **1 logischen Befehl**: Vergleich des Operanden mit 0
  - **3 Sprungbefehle**: 1 unbedingter Sprung, 2 bedingte Sprünge,
  - **2 Unterprogramm-Befehle**: Sprung zum Unterprogramm, Rückkehr
- PMI hat also keine Schiebebefehle, keine logische Operationen und keine Ein-/Ausgabebefehle.

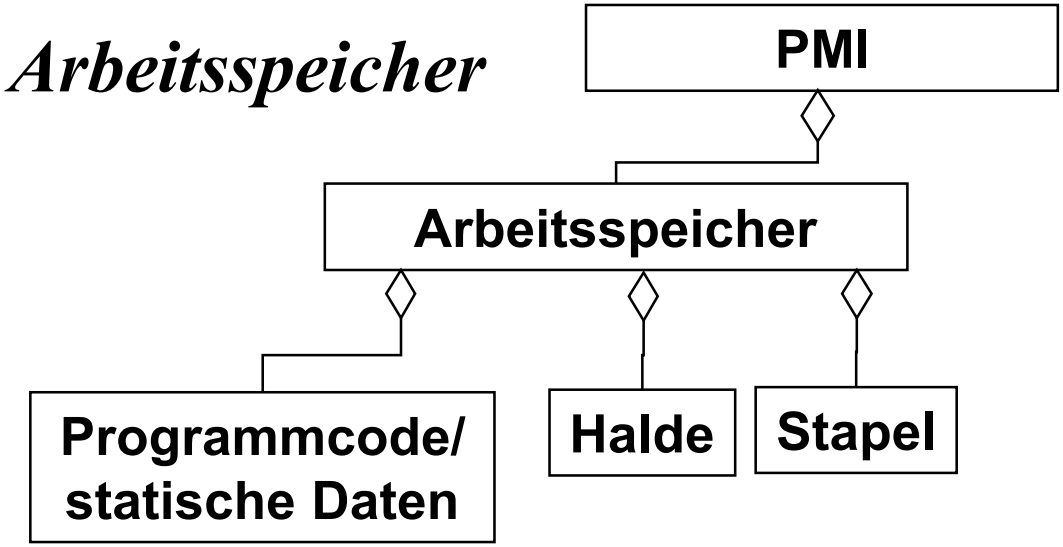
PMI-Modellierung

- **Statisches Modell:**
  - PMI Komponenten:
    - Arbeitsspeicher 
    - Prozessor 
      - Adressregister
      - Statusregister
  - Speicherorganisation 
  - Verwaltung des Arbeitspeichers mit Arbeitsregistern
  - Modellierung eines PMI-Programms 
  - Aufbau von PMI-Programmen 
- **Dynamisches Modell:**
  - Maschinenbefehlszyklus 
  - Adressierungsarten

Programmierung von PMI

- PMI-Assembler 

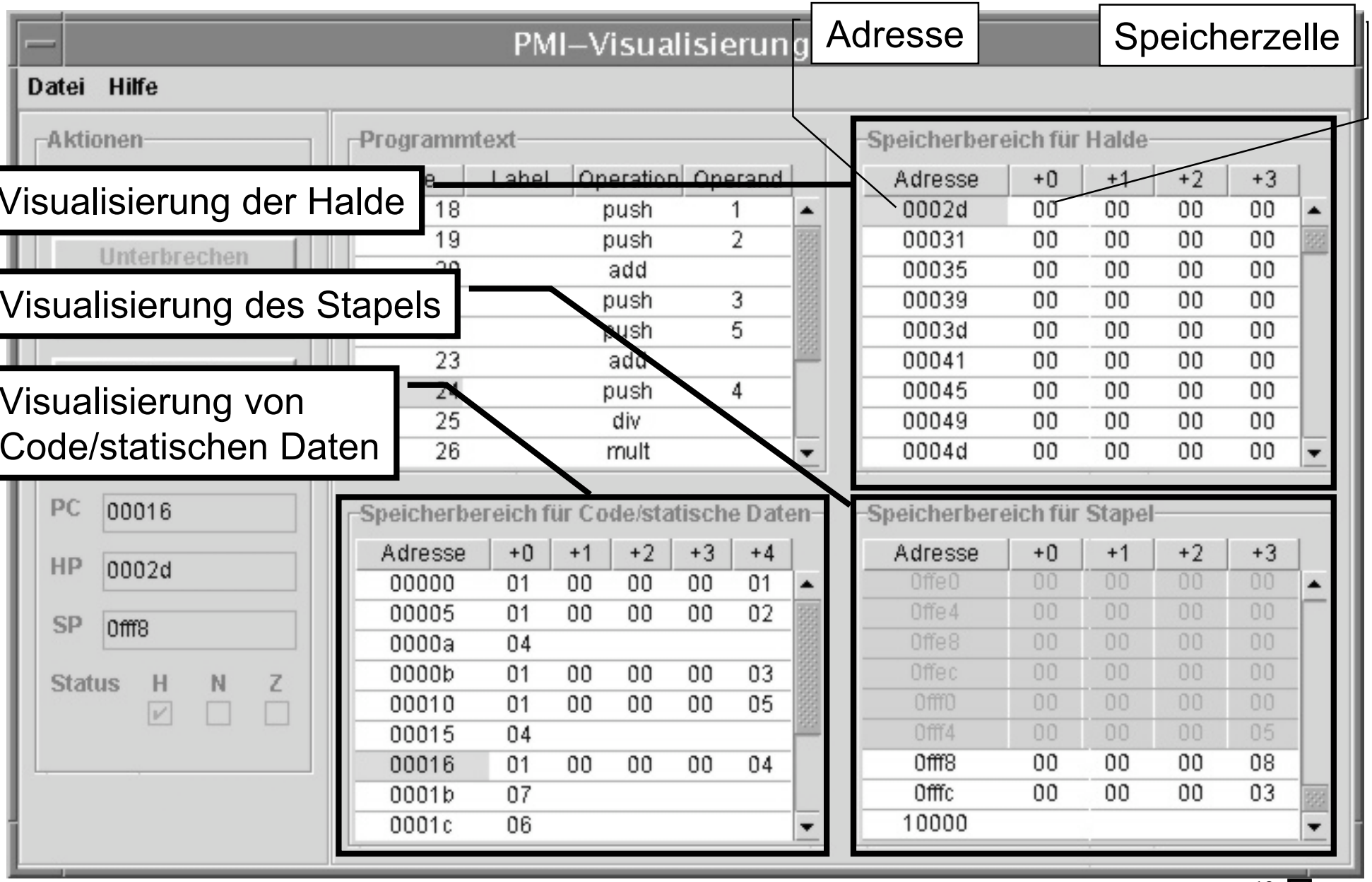
Arbeitsspeicher



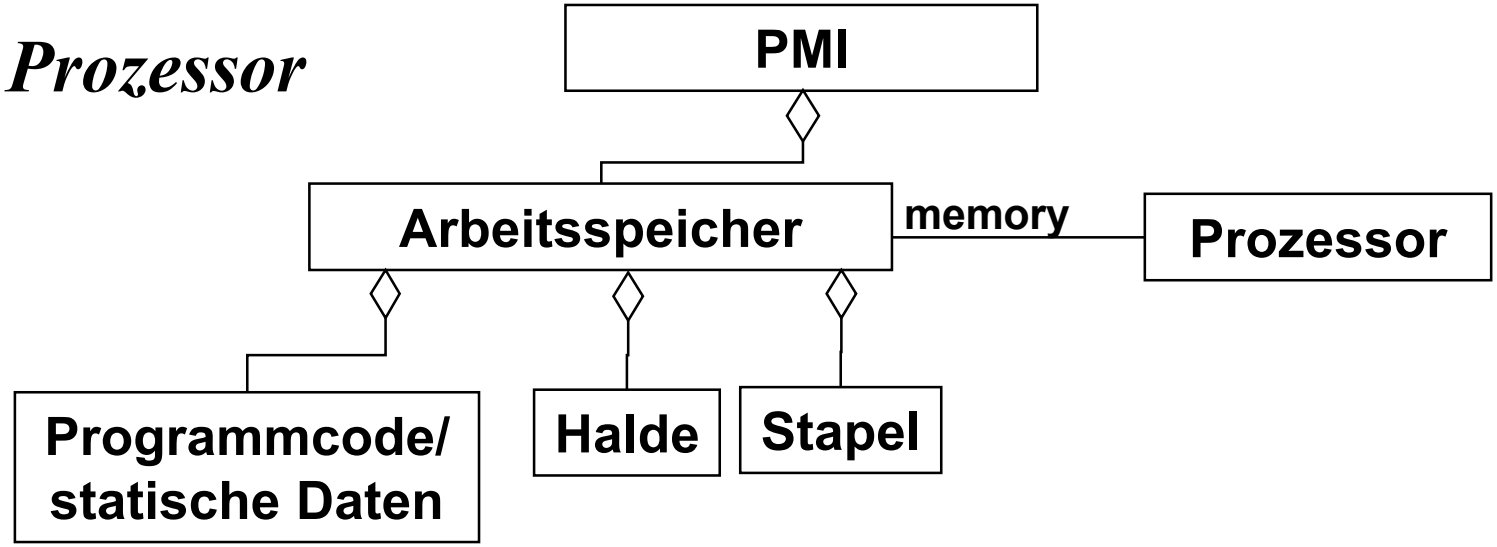
Der Arbeitsspeicher ist in drei Bereiche unterteilt:

- **Code/statische Daten**:
  - (binärer) Programmcode
  - *statische* Daten
- **Halde** (heap):
  - *dynamische* Daten
- **Stapel** (stack):
  - Operanden und Ergebnisse von PMI-Operationen
  - Parameter und lokale Variablen für Unterprogramme

Visualisierung des Arbeitsspeichers

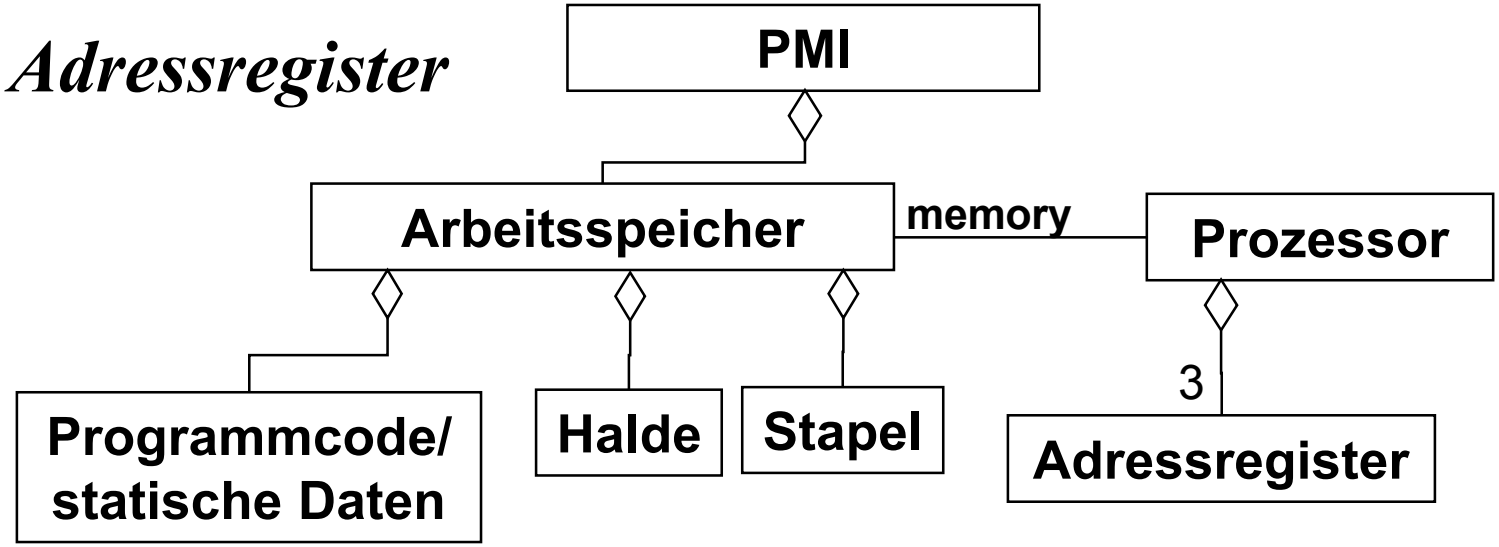


Prozessor



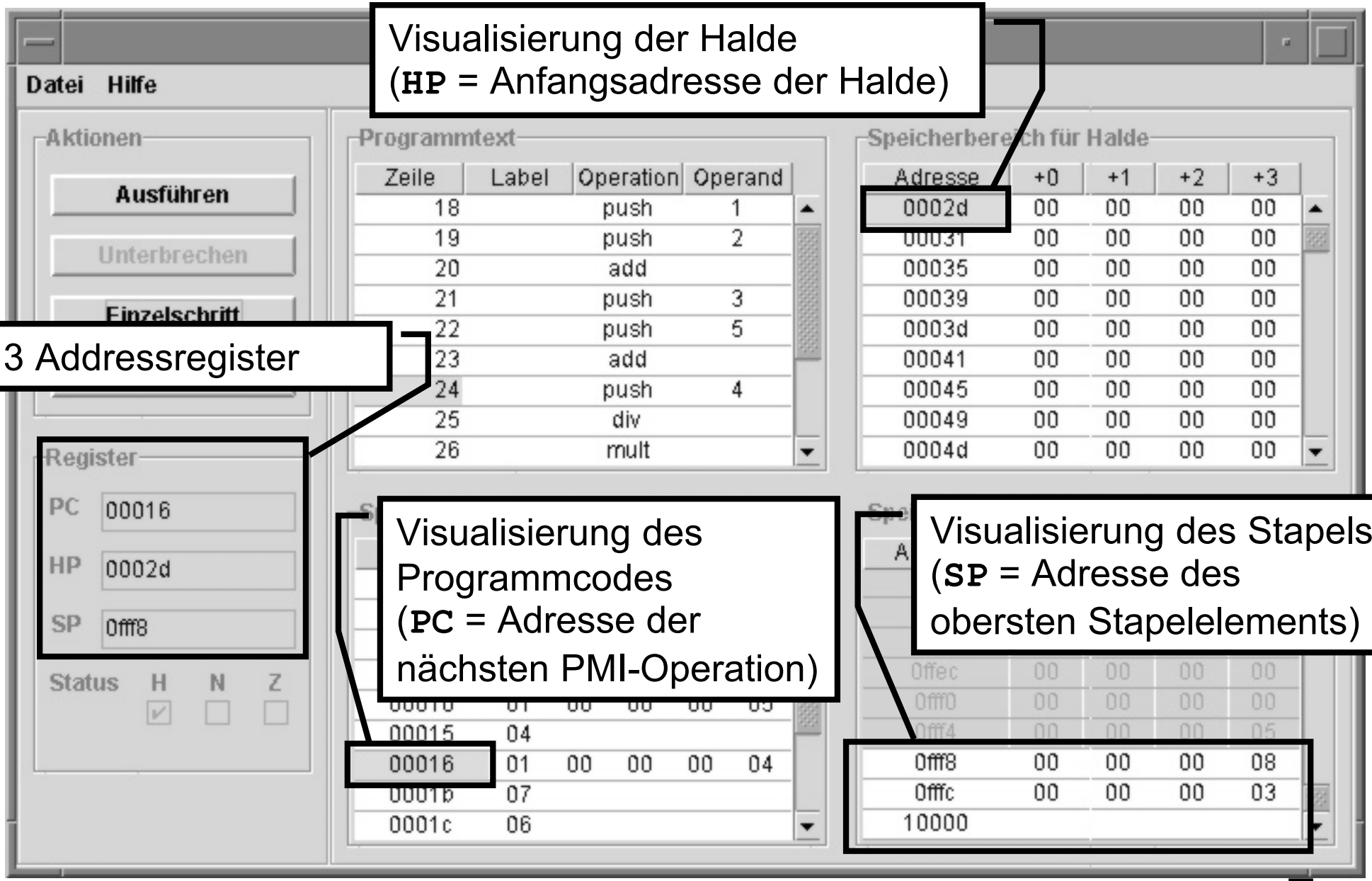
- Der **`Prozessor`** greift auf den **`Arbeitspeicher`** über die Assoziation namens **`memory`** zu.

Adressregister

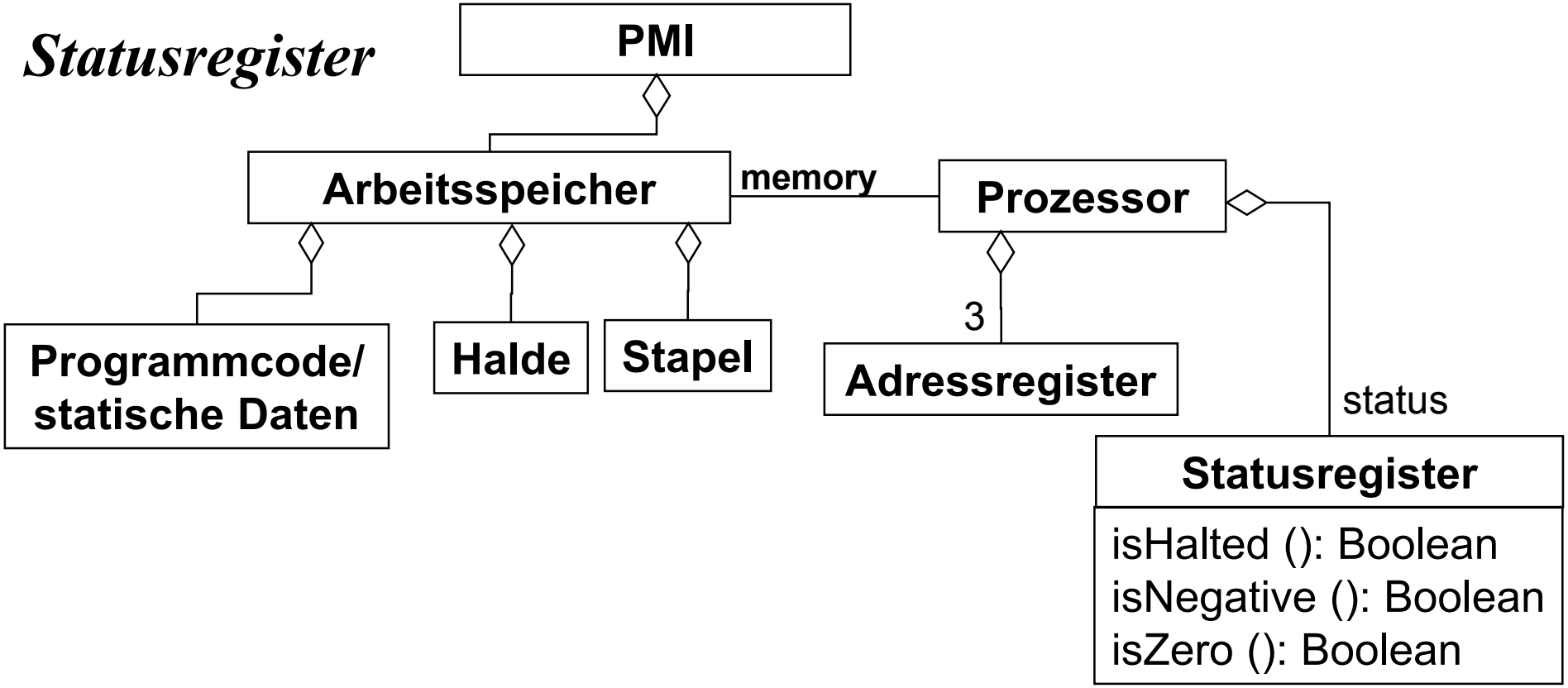


- Der Prozessor der PMI verfügt über ***3 Adressregister***, die zur ***Navigation*** innerhalb der einzelnen Speicherbereiche dienen:
  - **Programmzähler** (program counter, pc): Adresse der nächsten auszuführenden PMI-Operation
  - **Haldenzeiger** (heap pointer, hp): Anfangsadresse der Halde
  - **Stapelzeiger** (stack pointer, sp): Adresse des obersten Elements auf dem Stapel

Visualisierung der Adressregister

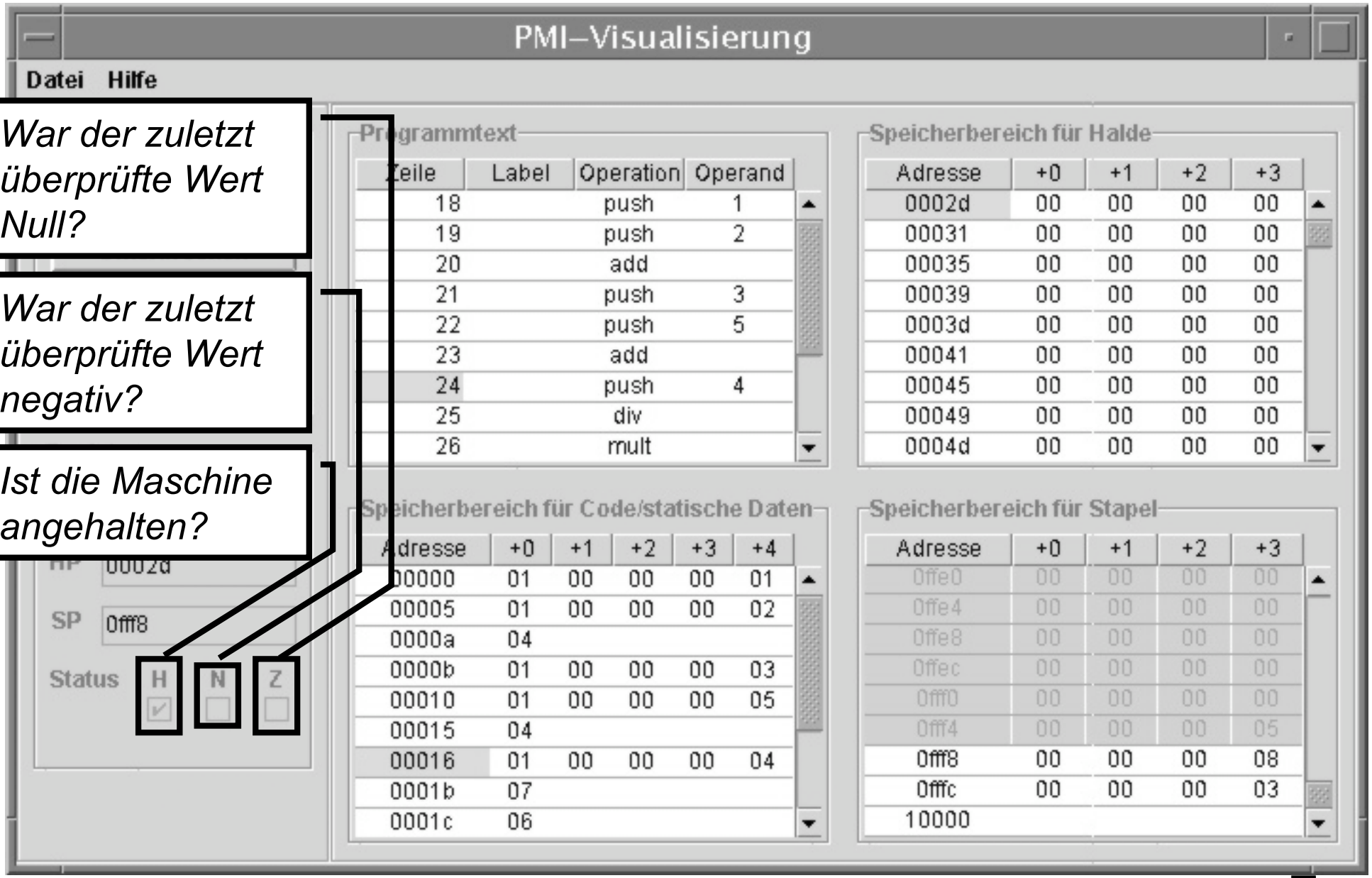


Statusregister



- Der PMI-Prozessor verfügt zusätzlich über ein ***Statusregister***, das Information über den aktuellen Zustand der Maschine liefert:
  - *"Ist die Maschine angehalten?"*
  - *"War der zuletzt überprüfte Wert negativ?"*
  - *"War der zuletzt überprüfte Wert Null?"*

Visualisierung des Statusregisters

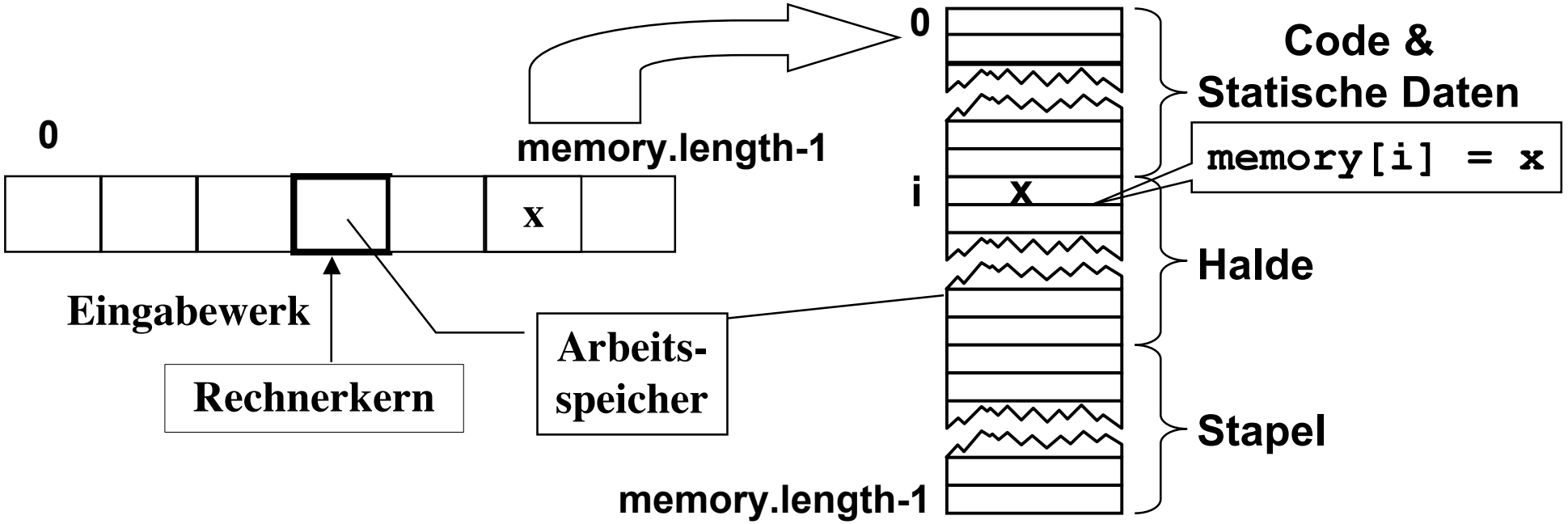


Speicherorganisation der PMI

- Die einzelnen Speicherzellen (Größe: 1 Byte) des Arbeitsspeichers der PMI sind linear angeordnet, d.h. der Speicher kann als Reihung von Speicherzellen modelliert werden:

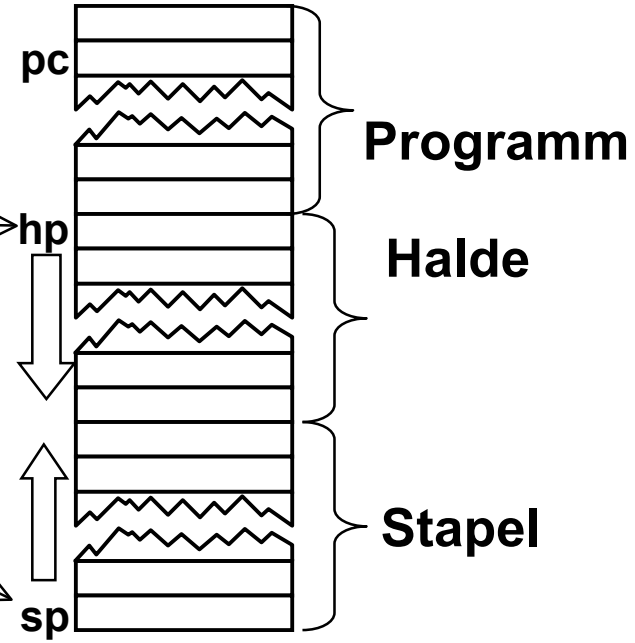
```
byte[] memory;
```

- Die Index-Position **`i`** einer Speicherzelle **`x`** ist die **Adresse** von **`x`**.



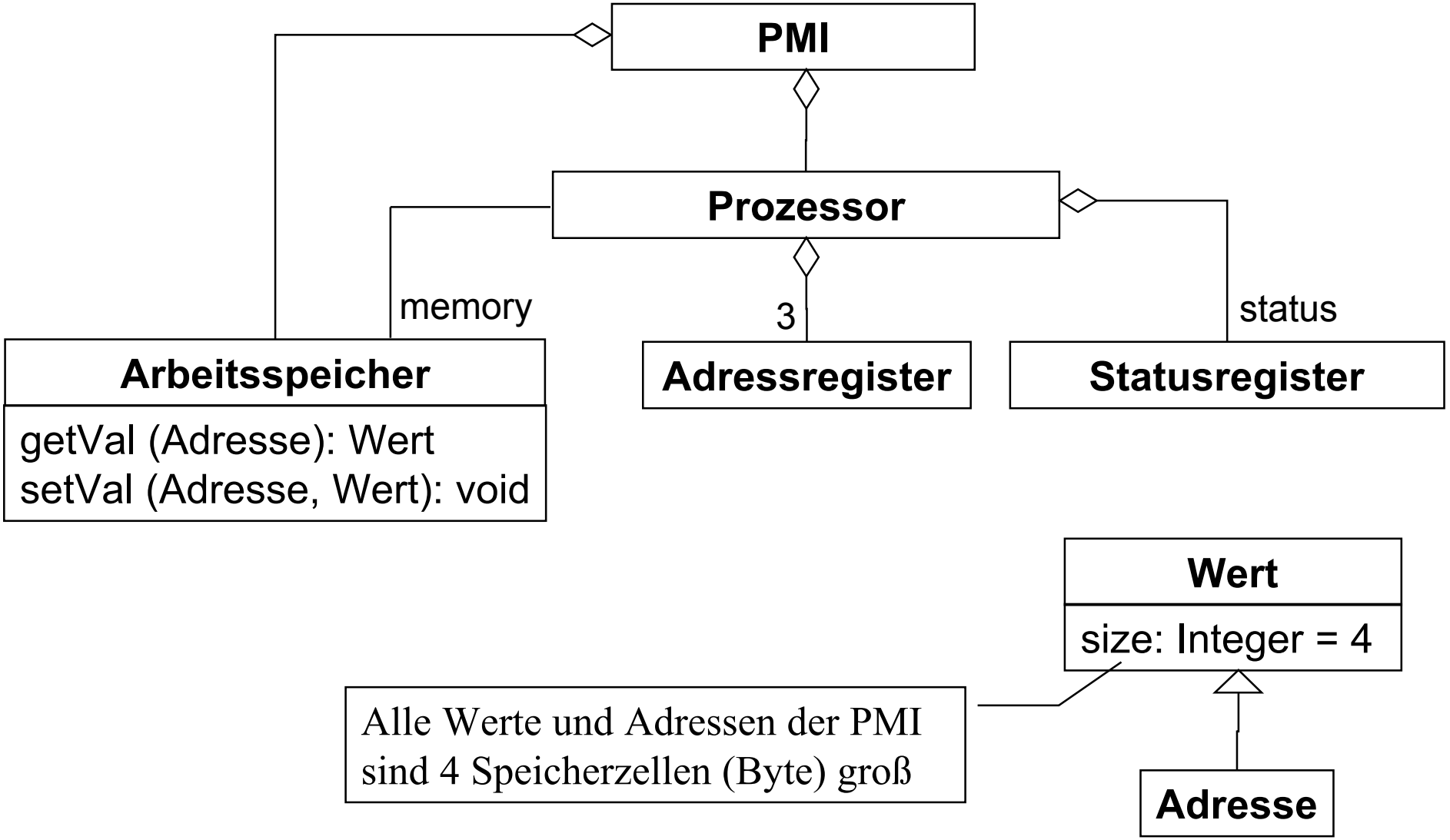
Verwaltung des Arbeitsspeichers mit Adressregistern

- Die ***3 Adressregister*** verwalten die Bereiche im Arbeitsspeicher:
  - **`pc`**: Adresse der nächsten auszuführenden Operation
  - **`hp`**: Anfangsadresse der Halde.
    Die Halde wächst von *"oben nach unten"*
    (von niedrigen Adressen zu hohen Adressen)
  - **`sp`**: Adresse des obersten Stapel-Elementes.
    Der Stapel wächst von *"unten nach oben"*
    (von hohen Adressen zu niedrigen Adressen)



- **Wichtig:** Die 3 Adressregister der PMI können nicht direkt gesetzt werden.
  - PMI-Befehle können Adressregister nur lesen.
  - Die Zuweisung von Registerwerten ist nur innerhalb der Maschine bei der Ausführung von PMI-Befehlen möglich.

PMI Werte und Adressen



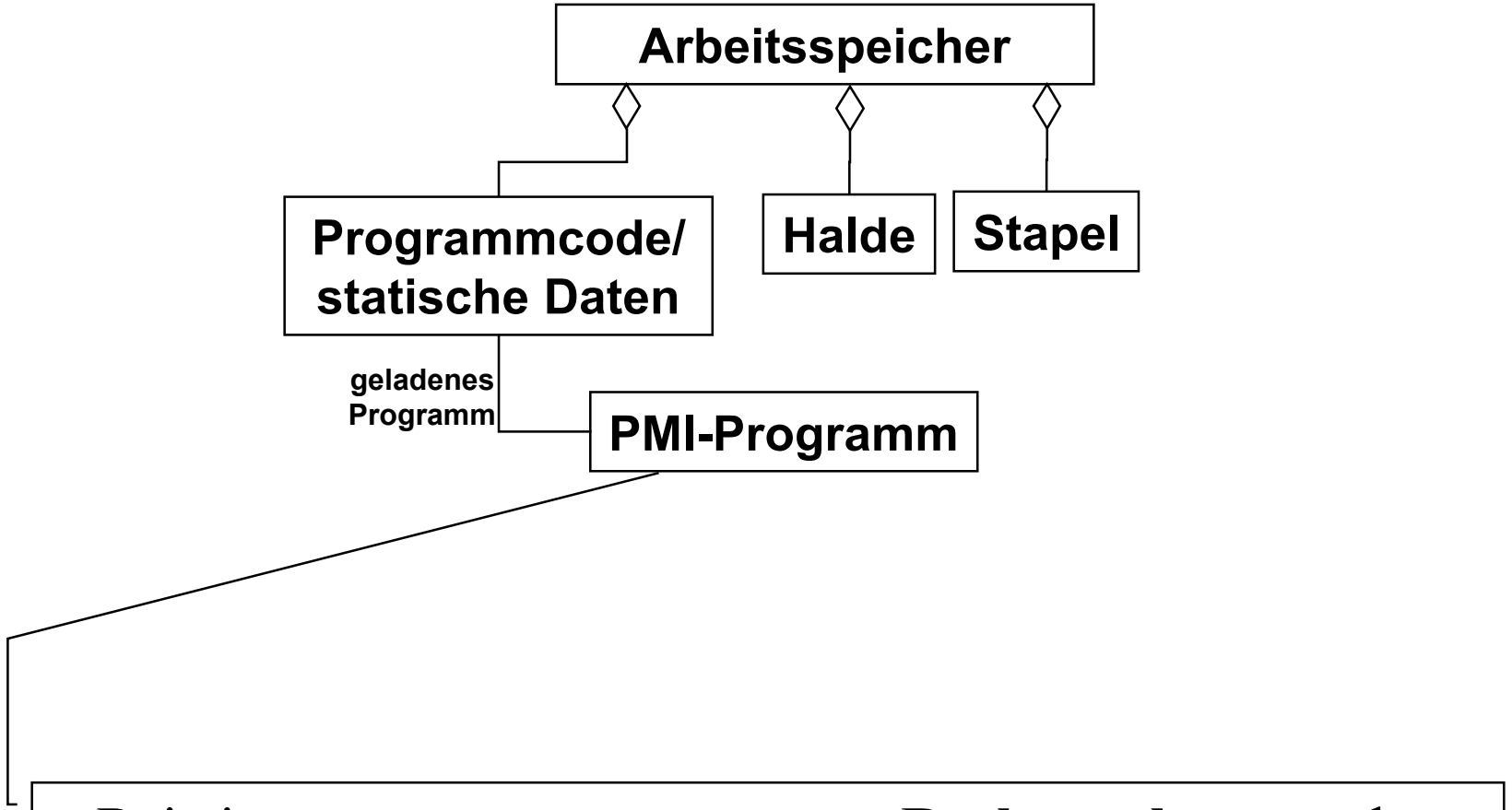
Befehle und Zustände

- Durch die Beschreibung aller vorhandenen Register und Speicherzellen wird die Menge der möglichen Zustände einer Maschine beschrieben.
- Es ist damit noch nicht festgelegt, in welcher Weise von einem Zustand zu einem anderen übergegangen wird. Diese Übergänge werden durch die Ausführung von **Befehlen** (engl instructions) bewirkt, die aus dem Speicher gelesen werden und im Steuerwerk ausgeführt werden.
- **Definition:** Ein **PMI Programm** (allgemeiner: Maschinen-Programm) besteht aus einer Menge von PMI-Befehlen (Maschinen-Befehlen).
- Jeder Befehl bewirkt die Änderung gewisser Speicherzellen bzw. Register oder die Übertragung von Werten in oder aus Speicherzellen bzw. Registern.
  Die Ausführung eines Befehls entspricht einem Zustandsübergang.
  - Der Befehlsvorrat beschreibt also die möglichen Zustandsübergänge.
  - Umgekehrt wird die Semantik (Wirkung) eines Befehls hinreichend durch den bewirkten Zustandsübergang charakterisiert.

Was kommt jetzt?

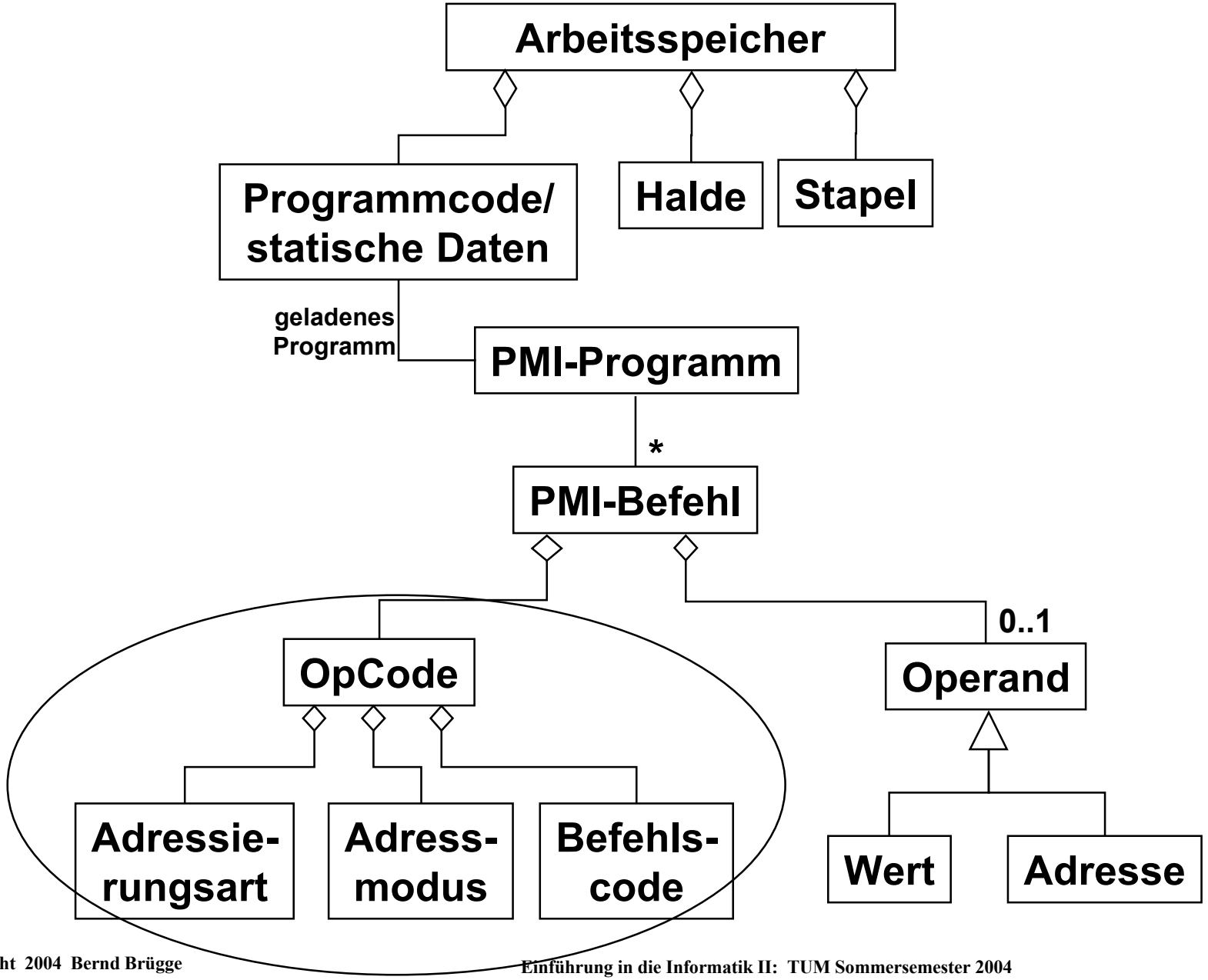
- Wir werden jetzt den Aufbau von PMI-Programmen modellieren
- Dann werden wir die Semantik der einzelnen Operationen des Befehlssatzes beschreiben.
  - Wir benutzen dazu das Konzept des Vertrages.
    Ein Befehl wird durch Vor- und Nachbedingungen beschrieben.
- Den gesamten Befehlssatz führen wir schrittweise anhand von einigen kleinen PMI-Programmen ein.

Modellierung von PMI-Programmen



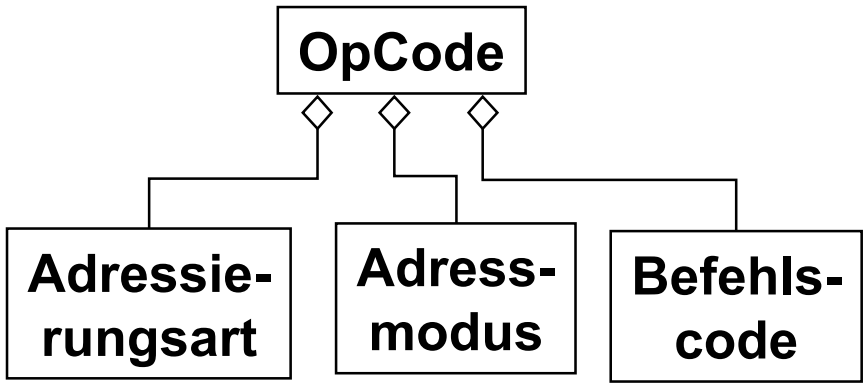
Bei einer **programmgesteuerten Rechenanlage** werden Programme im Arbeitsspeicher gespeichert

Modellierung von PMI-Programmen (2)

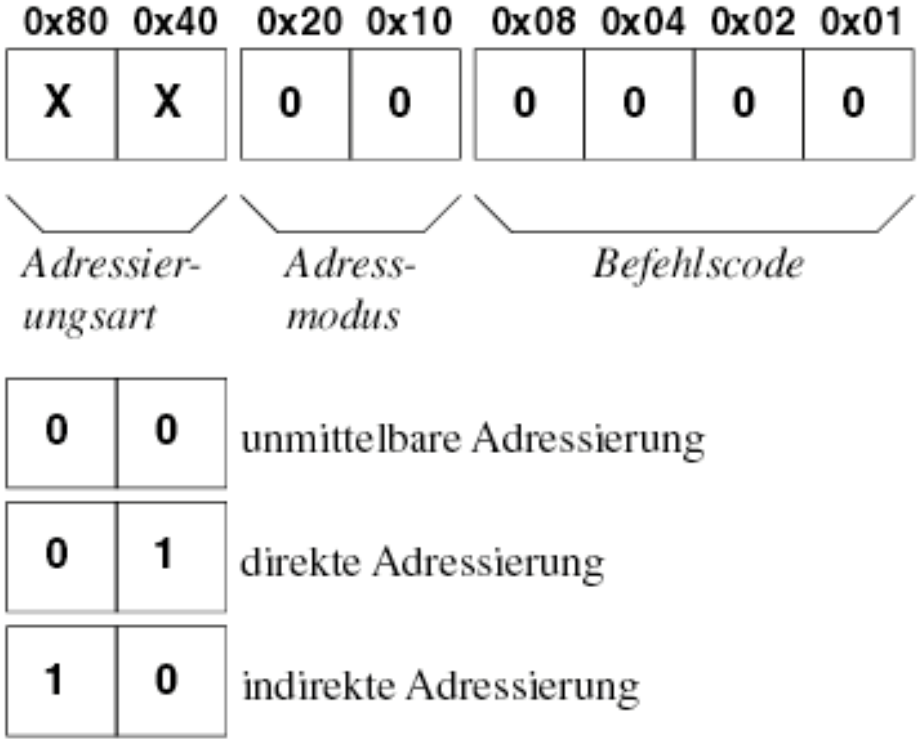


OpCode

- ***Analyse-Modell:*** Alle PMI-Befehle haben einen ***OpCode.*** Der OpCode enthält die Adressierungsart, den Adressmodus und den Befehlscode.



- ***Implementierungs-Modell:*** Realisiert ist der OpCode in einem Byte. Das Byte ist dabei folgendermaßen aufgebaut (die Wertigkeit der einzelnen Bits ist hexadezimal angegeben):



Implementierung des OpCode als Java-Schnittstelle

```
public interface PMIInstructionSet {
  // Adressierungsart
    int IMMEDIATE_ADDR = 0x00;
    int DIRECT_ADDR    = 0x40;
    int INDIRECT_ADDR  = 0x80;

  // Adressmodus
    int PC_REL_ADDR = 0x10;
    int HP_REL_ADDR = 0x20;
    int SP_REL_ADDR = 0x30;

  // Befehlscode
    int HALT = 0x00;
    int PUSH = 0x01;
    int POP  = 0x02;
    int DEL  = 0x03;
    int ADD  = 0x04;
    int SUB  = 0x05;
    int MULT = 0x06;
    int DIV  = 0x07;
    int CMP  = 0x08;
    int JMP  = 0x09;
    int JN   = 0x0A;
    int JZ   = 0x0B;
    int JSR  = 0x0C;
    int RET  = 0x0D;
    int INSTR_MAX = RET;
}
  // PMIInstructionSet
```

Java Dokumentation: `$PMIDOC/model/PMIInstructionSet.html`

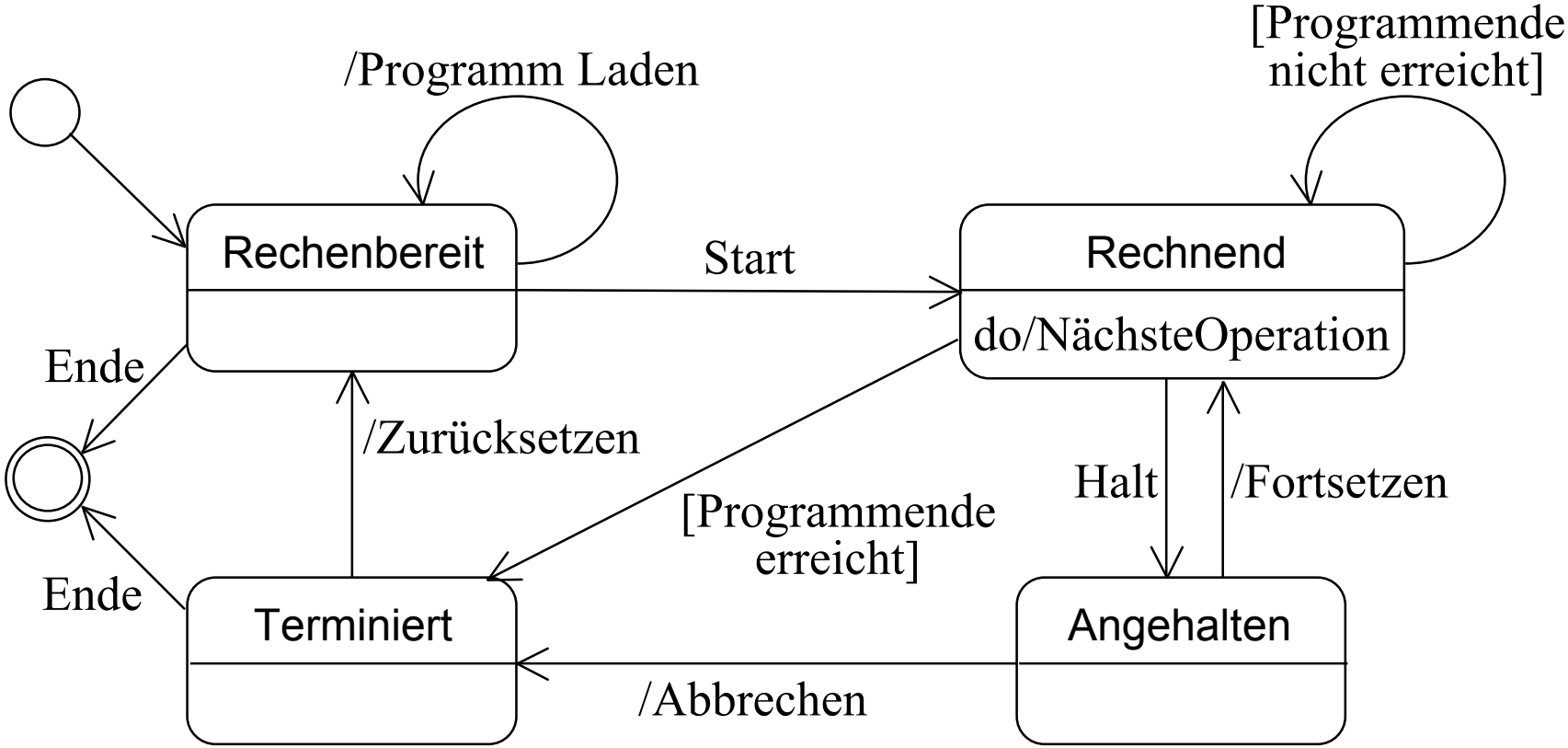
Implementierung des PMI-Befehlssatzes (`Prozessor.java`)

- Die Klasse **`Prozessor`** realisiert den PMI-Befehlssatz als eine Menge von öffentlichen Methoden:

	Prozessor
Anhalten der Maschine	halt ()
Manipulation des Stapels	push (Wert)
	pop (Adresse)
	delete ()
arithmetische Operationen	add ()
	sub ()
	mult ()
	div ()
Vergleich	compare ()
Sprünge	jump (Adresse)
	jumpZero (Adresse)
	jumpNegative (Adresse)
Unterprogramm-Aufruf	jumpSubroutine (Adresse)
	return ()

Dynamisches Verhalten der PMI

- Für die Benutzung der PMI sind die dynamischen Aspekte des Modells von großem Interesse:
  - *"Wie werden die Abläufe der Maschine gesteuert?"*
  - *"Welche Aktionen kann die Maschine ausführen?"*



Maschinenbefehlszyklus

Jeder PMI-Befehl besteht aus einer Operation (**`opCode`**) und 0 oder 1 Operanden (**`operand`**). Er wird nach folgendem Schema ausgeführt:

1. **Einlesen des OpCodes der Operation aus dem Speicher:**

   ```
   byte opCode = memory[pc];
   ```

2. **Bestimmung der dem OpCode entsprechenden Operation (z.B. `add` oder `halt`)**

3. **Bestimmen des Operandenwerts** (für Befehle mit einem Operanden):

   a) Bestimme den Adressmodus des Operanden 

   b) Bestimmen die Adressierungsart (übernächste Folie) 

4. **Ausführen des Befehls:** Wenn **`Operation == halt`**, halte die Maschine an, sonst verändere Speicher- und/oder Registerwerte entsprechend der Spezifikation des Befehls.

5. **Weiterschalten des Programmzählers**:

   ```
   pc = pc + 1 + Wert.size;// bei Operationen ohne
                           // Operanden ist Wert.size=0
   ```

Bestimmung des Adressmodus für den Operanden

- Der Adressmodus für die Bestimmung des Operanden-Wertes wird nach folgender Regel berechnet:

```
Wert operandCode = basisAdresse + memory.getVal(pc + 1);
```

- Die **`basisAdresse`** ist 0 oder ein Registerwert. Wir unterscheiden:
  - ***Absolute Adressierung:*** Die Basisadresse hat den Wert 0. Der aus dem Programmcode gelesene Wert ist bereits der Ausgangswert:

```
operandCode = memory.getVal(pc + 1);
```

  - ***Relative Adressierung:*** Die Basisadresse ist der Wert eines Adressregisters (**`pc`**, **`hp`** oder **`sp`**) plus dem aus dem Programmcode gelesenen Wert. Beispiel (**`sp`**-relative Adressierung):

```
operandCode = sp + memory.getVal(pc + 1);
```

- Welcher Adressmodus für die Bestimmung des Operanden verwendet wird, ist durch den OpCode eindeutig festgelegt.
- **Bemerkung**: Wenn die Basisadresse ein beliebiger Wert im Speicher sein kann, wird die relative Adressierung auch ***Indexadressierung*** genannt. Indexadressierung wird in PMI *nicht* unterstützt.

Bestimmung der Adressierungsart

- Der durch Auswertung des ***Adressmodus*** ermittelte **`operandCode`** ist der Ausgangswert für die Bestimmung des Operandenwerts.
  Die ***Adressierungsart*** legt fest, wie **`operandCode`** ausgewertet wird.
- Die PMI unterstützt drei verschiedene Adressierungsarten:

1. Unmittelbare Adressierung:

```
Wert operand = operandCode;
```

2. Direkte Adressierung:

```
Wert operand = memory.getVal(operandCode);
```

3. Indirekte Adressierung:

```
Wert operand =
    memory.getVal(memory.getVal(operandCode));
```

- Welche Adressierungsart verwendet wird, ist durch den OpCode eindeutig festgelegt.

Programmierung der PMI: PMI-Assembler

- PMI-Programme werden als Binär-Datei in den Arbeitsspeicher geladen.
- Um die Erstellung von PMI-Programmen etwas zu vereinfachen, wird eine sog. ***Assembler-Sprache*** bei der Formulierung des Programms verwendet, die von einem ***Assembler*** in den Binärcode übersetzt wird.
- Die Aufgaben des Assemblers sind:
  - Übersetzung von symbolischen Operationsbezeichnern
  - Übersetzung von symbolischen Operandenbeschreibungen
- Das in Assembler-Sprache geschriebene Programm wird oft als **Assemblerprogramm** oder Assembler-Code bezeichnet.
  - Für das Assemblerprogramm wird ebenfalls oft die Bezeichnung "Assembler" verwendet. Aus dem Kontext wird gewöhnlich klar, ob das Programm selbst oder der Übersetzer gemeint ist.

PMI-Assemblerprogramm vs PMI-Maschinenprogramm

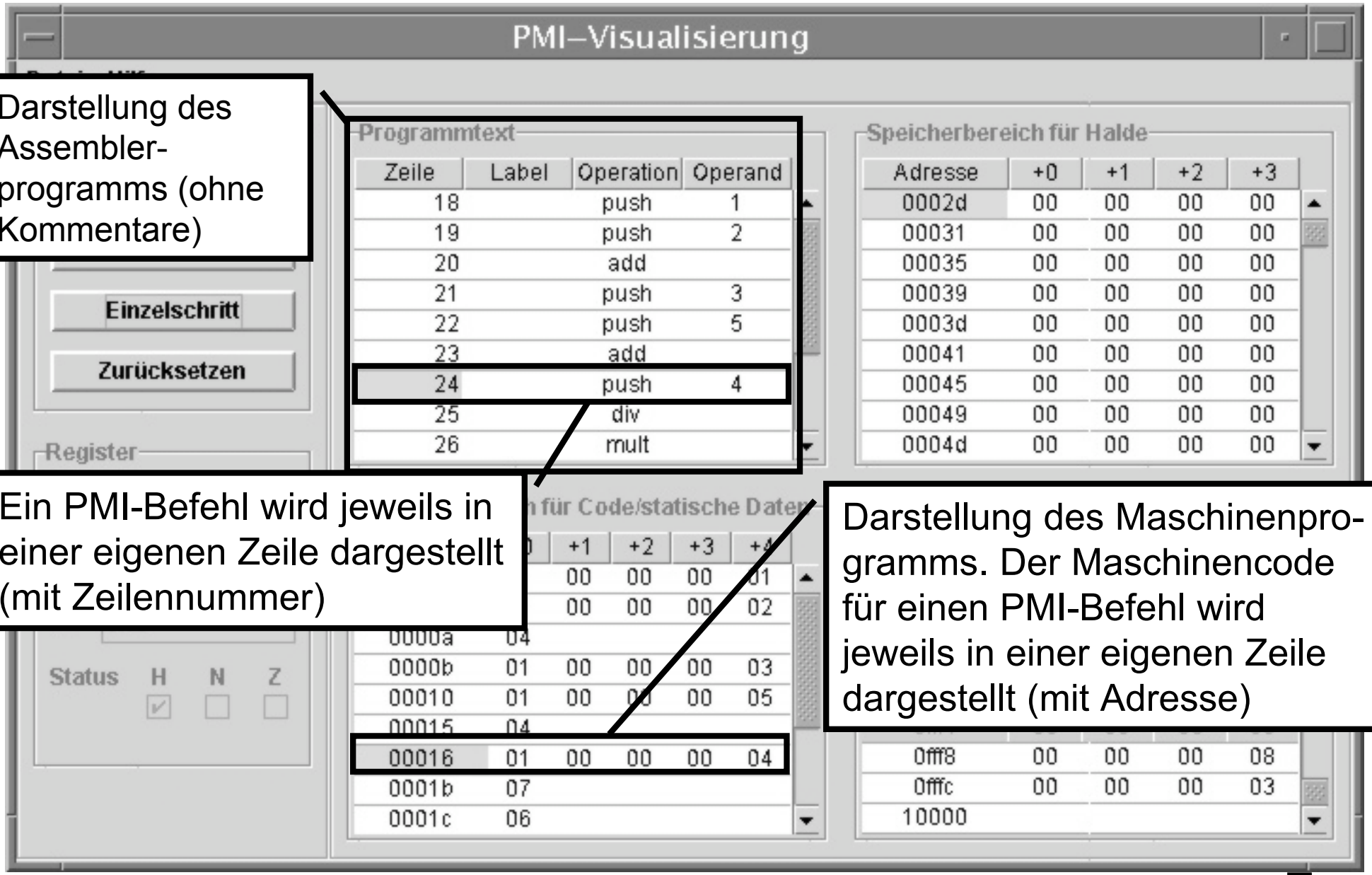
PMI-Assemblerprogramm

Zeile	Label	Operation	Operand
18		push	1
19		push	2
20		add	
21		push	3
22		push	5
23		add	
24		push	4
25		div	
26		mult	
27		push	6
28		mult	
29		pop	i
30		halt	
31	i:	dd	0

PMI-Maschinenprogramm

Adresse	+0	+1	+2	+3	+4
00000	01	00	00	00	01
00005	01	00	00	00	02
0000a	04				
0000b	01	00	00	00	03
00010	01	00	00	00	05
00015	04				
00016	01	00	00	00	04
0001b	07				
0001c	06				
0001d	01	00	00	00	06
00022	06				
00023	02	00	00	00	29
00028	00				
00029	00	00	00	00	

Visualisierung von PMI-Assembler-/Maschinenprogrammen



Darstellung des Assembler-programms (ohne Kommentare)

Ein PMI-Befehl wird jeweils in einer eigenen Zeile dargestellt (mit Zeilennummer)

Darstellung des Maschinenpro-gramms. Der Maschinencode für einen PMI-Befehl wird jeweils in einer eigenen Zeile dargestellt (mit Adresse)

PMI-Assembler: Syntax (in EBNF)

⟨*PMI-Programm*⟩ ::= ⟨*PMI-Programmzeile*⟩*

⟨*PMI-Programmzeile*⟩ ::= ⟨*PMI-Anweisung*⟩ [⟨*Kommentar*⟩] |
⟨*Kommentar*⟩

Leerzeichen

⟨*PMI-Anweisung*⟩ ::= [⟨*Adressbezeichner*⟩ **:**] □$^{+}$ ⟨*PMI-Befehl*⟩

⟨*PMI-Befehl*⟩ ::= ⟨*Operationsbezeichner*⟩ [□$^{+}$ ⟨*Operand*⟩] |
⟨*Datendefinition*⟩

⟨*Operationsbezeichner*⟩ ::= **halt** | **push** | **pop** | **del** | **add** | **sub** | **mult** | **div** | **comp** | **jump** | **jmpn** | **jmpz** | **jsr** | **ret**

⟨*Datendefinition*⟩ ::= **dd** □$^{+}$ ⟨*Operand*⟩

⟨*Operand*⟩ ::= [⟨*Adressierungsart*⟩] ⟨*Adresse*⟩ [⟨*Offset*⟩]

⟨*Adressierungsart*⟩ ::= **@** | **>**

⟨*Adresse*⟩ ::= ⟨*Adressbezeichner*⟩ | ⟨*Register*⟩ | [**-**] ⟨*Zahl*⟩

⟨*Register*⟩ ::= **pc** | **hp** | **sp**

⟨*Offset*⟩ ::= **+** ⟨*Zahl*⟩ | **-** ⟨*Zahl*⟩

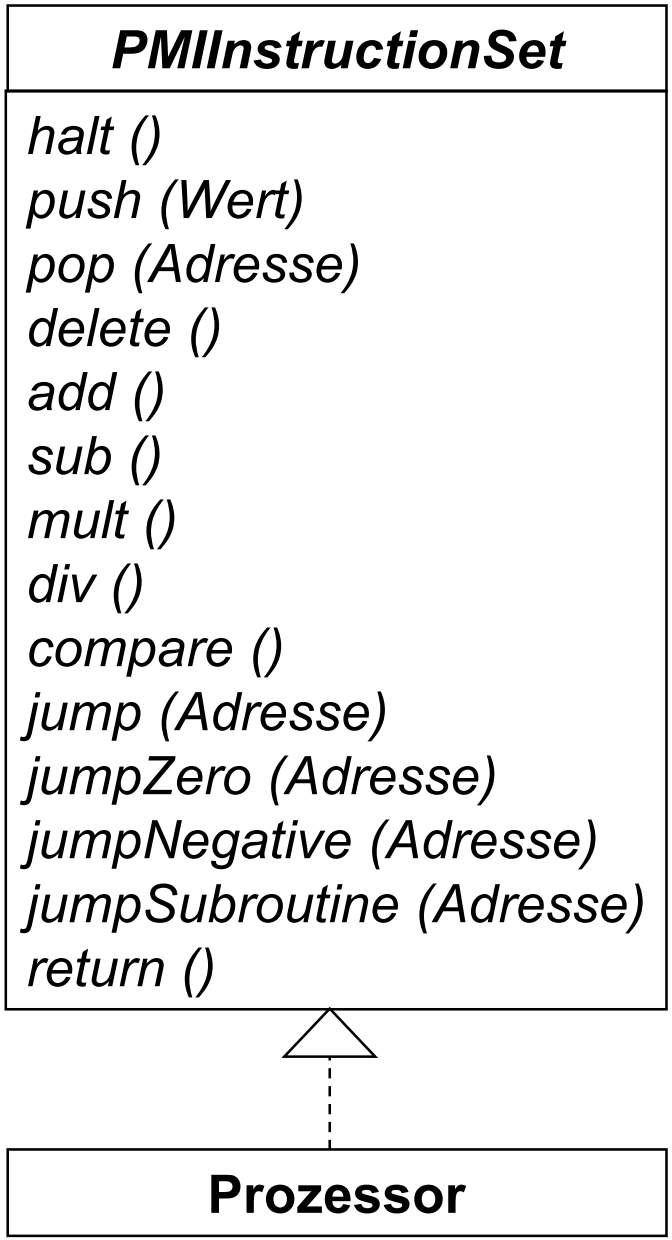
⟨*Kommentar*⟩ ::= // ⟨*Text*⟩

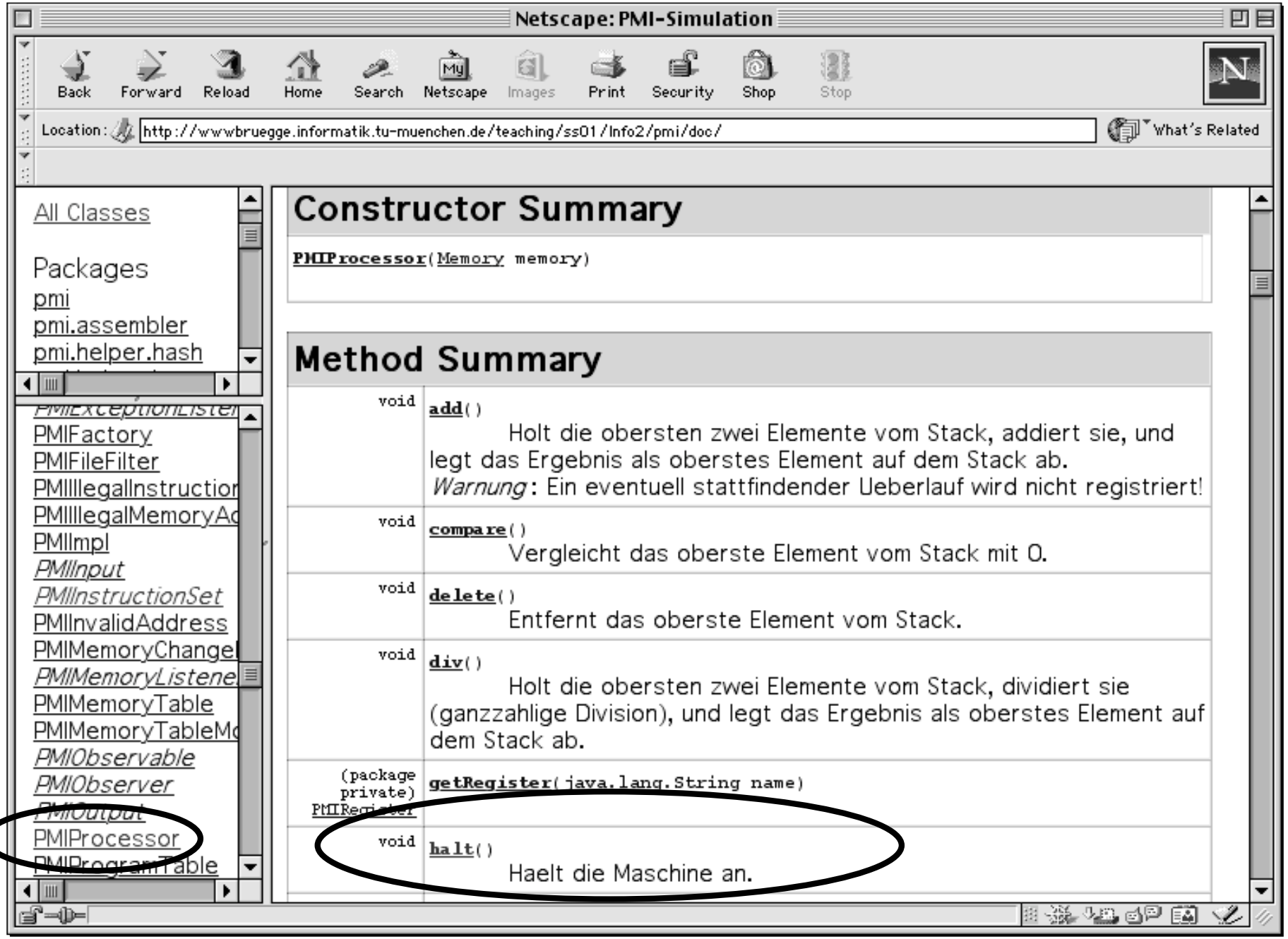
Spezifikation des PMI-Befehlssatzes

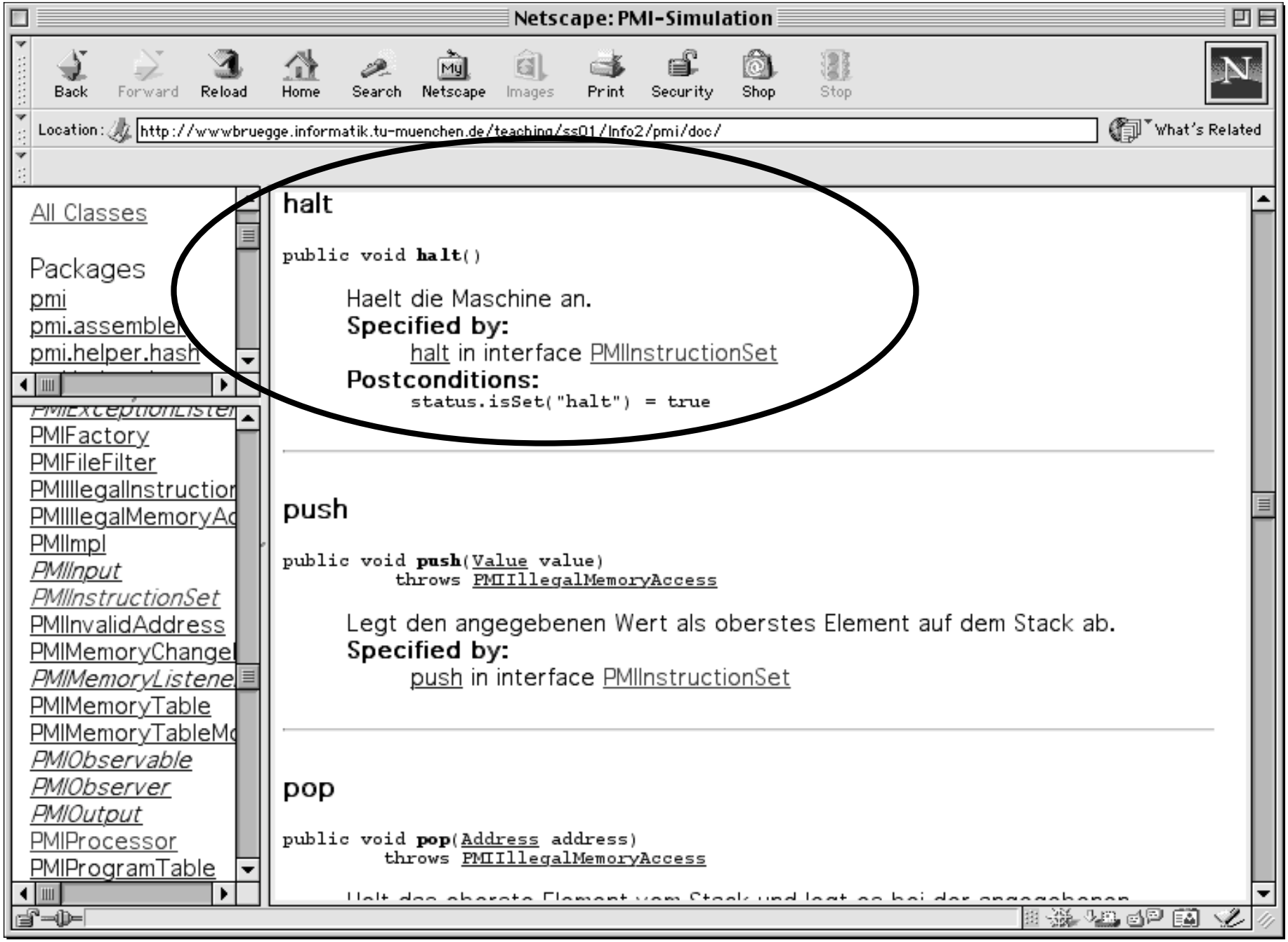
- Wir modellieren den PMI-Befehlssatz in der Schnittstelle **`PMIInstructionSet`** als einen Vertrag für die Klasse **`Prozessor`**.
- Jede öffentliche Methode in **`Prozessor`** ist ein Maschinenbefehl.
- Wir spezifizieren alle Methoden mit Vor- und Nachbedingungen
  - Die Spezifikation des gesamten PMI-Befehlssatzes ist mit Javadoc dokumentiert:

    http://wwwbruegge.in.tum.de/Lehrstuhl/Informatik2SoSe2004`PMI`

    (im weiteren Text mit **`$PMI`** abgekürzt)
- Im Folgenden schauen wir uns diese Spezifikation genauer an.




Netscape: PMI-Simulation
Back
Forward
Reload
Home
Search
Netscape
Images
Print
Security
Shop
Stop
Location: http://wwwbruegge.informatik.tu-muenchen.de/teaching/ss01/Info2/pmi/doc/
What's Related
All Classes
Packages
pmi
pmi.assembler
pmi.helper.hash
PMIFactory
PMIFileFilter
PMIIllegalInstructior
PMIIllegalMemoryAc
PMIImpl
PMIInput
PMIInstructionSet
PMIInvalidAddress
PMIMemoryChange
PMIMemoryListene
PMIMemoryTable
PMIMemoryTableMc
PMIObservable
PMIObserver
PMIProcessor
Constructor Summary
PMIProcessor(Memory memory)
Method Summary
void add()
Holt die obersten zwei Elemente vom Stack, addiert sie, und legt das Ergebnis als oberstes Element auf dem Stack ab.
Warnung: Ein eventuell stattfindender Ueberlauf wird nicht registriert!
void compare()
Vergleicht das oberste Element vom Stack mit 0.
void delete()
Entfernt das oberste Element vom Stack.
void div()
Holt die obersten zwei Elemente vom Stack, dividiert sie (ganzzahlige Division), und legt das Ergebnis als oberstes Element auf dem Stack ab.
(package private) PMIRegister
getRegister(java.lang.String name)
void halt()
Haelt die Maschine an.


Netscape: PMI-Simulation
Back
Forward
Reload
Home
Search
Netscape
Images
Print
Security
Shop
Stop
Location: http://wwwbruegge.informatik.tu-muenchen.de/teaching/ss01/Info2/pmi/doc/
What's Related
All Classes
Packages
pmi
pmi.assembler
pmi.helper.hash
PMIFactory
PMIFileFilter
PMIIllegalInstruction
PMIIllegalMemoryAc
PMIImpl
PMIInput
PMIInstructionSet
PMIInvalidAddress
PMIMemoryChange
PMIMemoryListene
PMIMemoryTable
PMIMemoryTableMo
PMIObservable
PMIObserver
PMIOutput
PMIProcessor
PMIProgramTable
halt
public void halt()
Haelt die Maschine an.
Specified by:
halt in interface PMIInstructionSet
Postconditions:
status.isSet("halt") = true
push
public void push(Value value)
throws PMIIllegalMemoryAccess
Legt den angegebenen Wert als oberstes Element auf dem Stack ab.
Specified by:
push in interface PMIInstructionSet
pop
public void pop(Address address)
throws PMIIllegalMemoryAccess

Spezifikation des Befehlssatzes: Anhalten der Maschine

- **halt ()**:
  Hält die Maschine an.

```
Prozessor::halt():void
post: status.ishalted() = true
post: pc = pc@pre
```

- Die **halt()**-Operation führt unmittelbar zur Terminierung des Programms.
- Wenn die Maschine angehalten wird (egal, ob durch das Programm oder durch die Steuerung der PMI-Maschine), kann sie nur von außen, d.h. durch die Steuerung, wieder in Gang gesetzt werden.

Prozessor
halt ()
push (Wert)
pop (Adresse)
delete ()
add ()
sub ()
mult ()
div ()
compare ()
jump (Adresse)
jumpZero (Adresse)
jumpNegative (Adresse)
jumpSubroutine (Adresse)
return ()

Spezifikation der Stapel-Operationen

- **push (Wert)**:
  Legt den Argument-Wert als oberstes Element auf den Stapel

```
Prozessor::push(w: Wert):void
post: pc = pc@pre + 1 + Wert.size
post: sp = sp@pre - Wert.size
post: mem.getVal(sp) = w
```

- **pop (Adresse)**:
  Nimmt das oberste Element vom Stapel und legt es bei der Argument-Adresse ab

```
Prozessor::pop(a: Adresse):void
post: pc = pc@pre + 1 + Wert.size
post: sp = sp@pre + Wert.size
post: mem.getVal(a) =
        mem@pre.getVal(sp@pre)
```

- **delete ()**:
  Entfernt das oberste Element vom Stapel

```
Prozessor::delete():void
post: pc = pc@pre + 1
post: sp = sp@pre + Wert.size
```

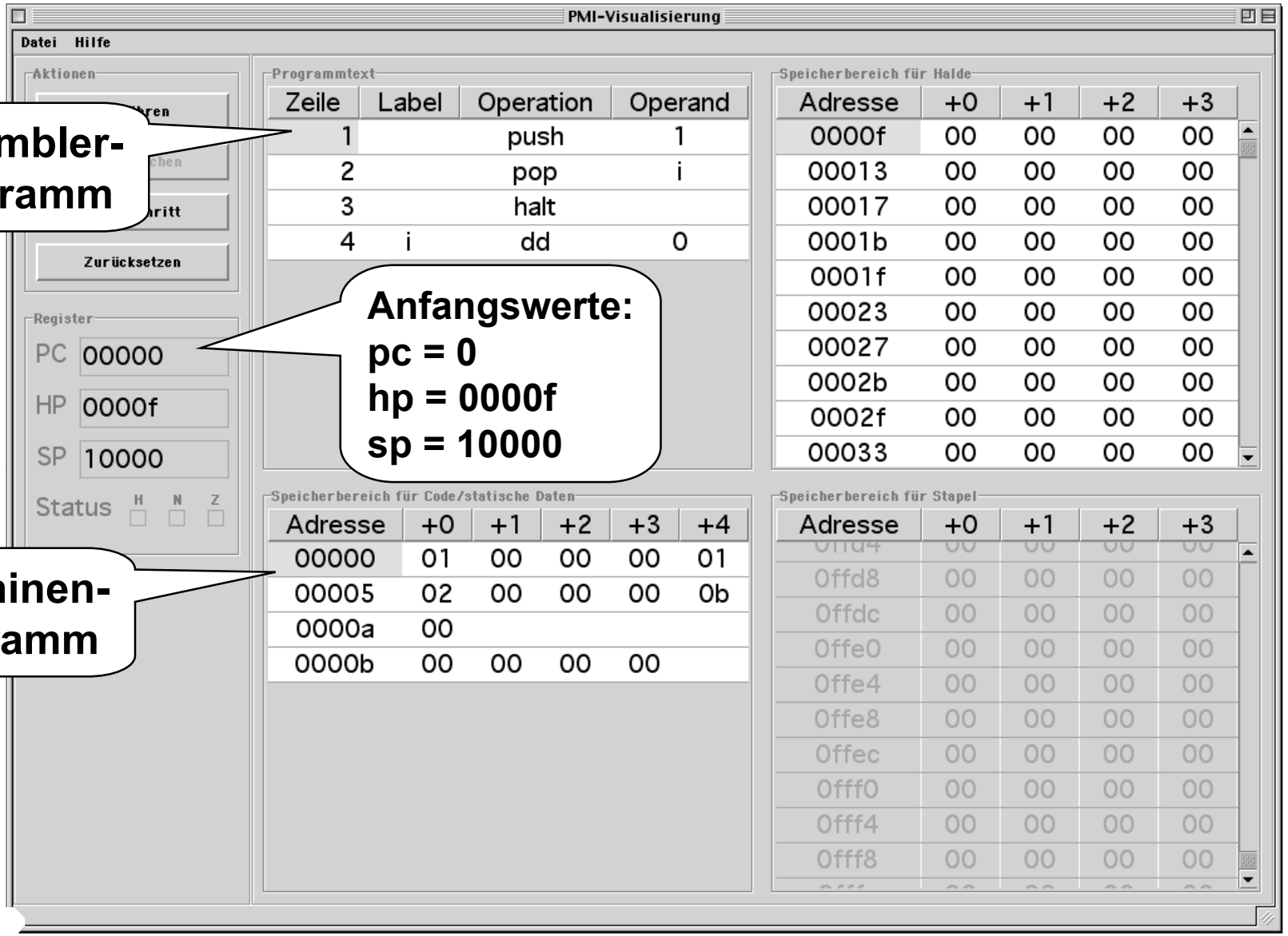
Prozessor
halt ()
push (Wert)
pop (Adresse)
delete ()
add ()
sub ()
mult ()
div ()
compare ()
jump (Adresse)
jumpZero (Adresse)
jumpNegative (Adresse)
jumpSubroutine (Adresse)
return ()

Beispiel: Exekution eines einfachen PMI-Programms

```
      push    1
      pop     i
      halt
i:  dd  0
```

PMI-Maschine

Vor der Ausführung des ersten Befehls



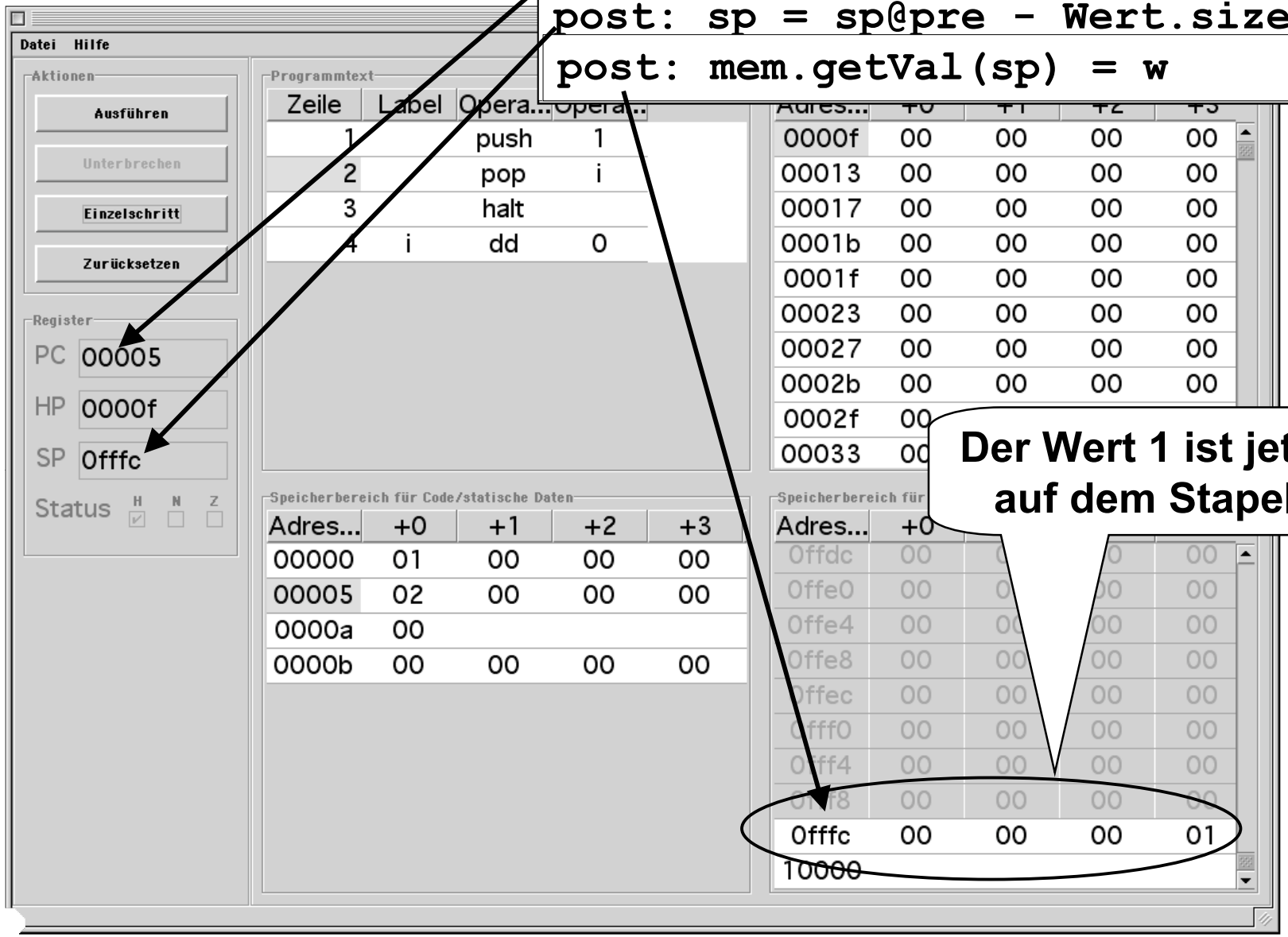
Assembler-Programm

Anfangswerte:
pc = 0
hp = 0000f
sp = 10000

Maschinen-Programm

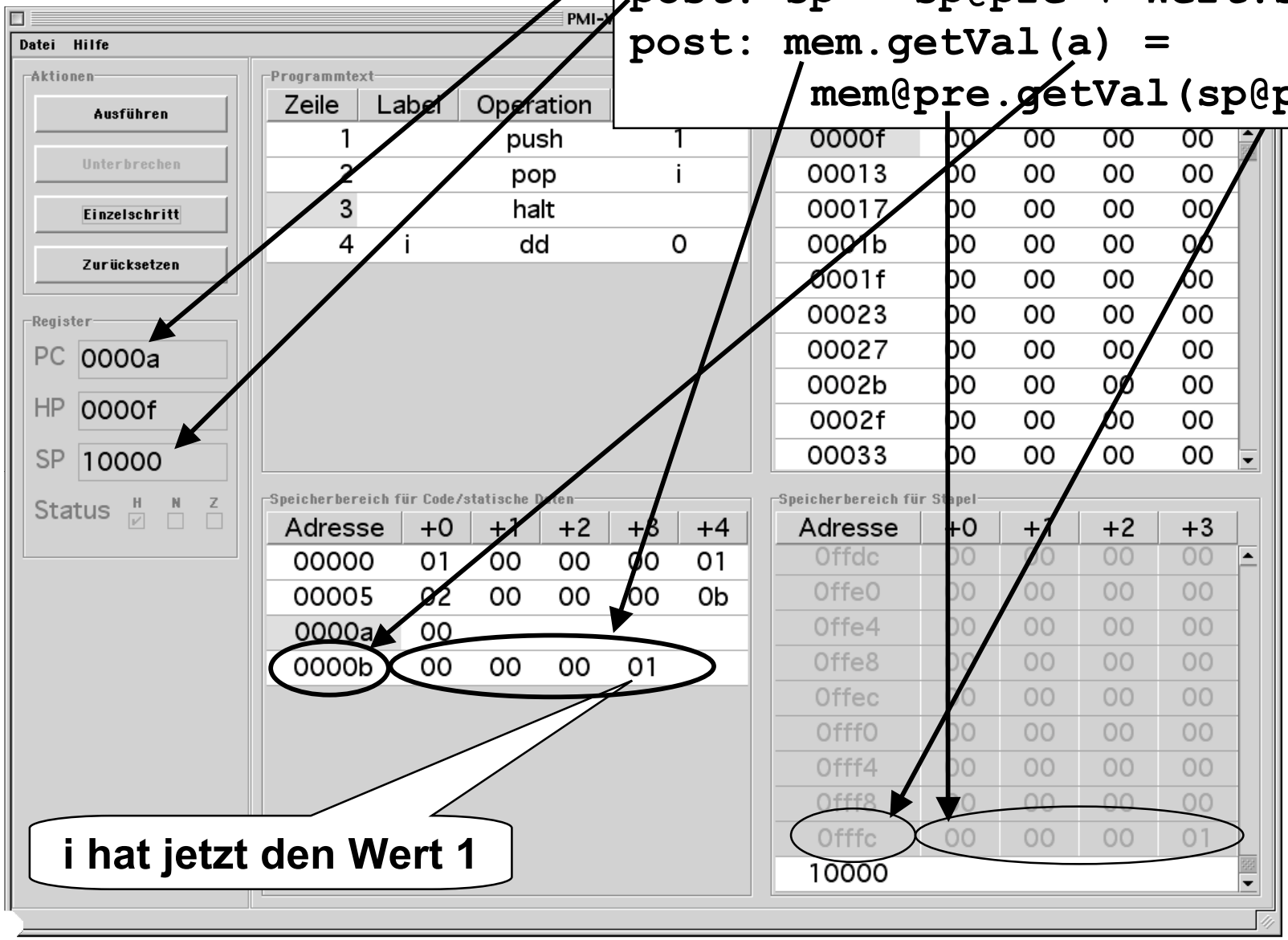
Zustand nach `push 1`

```
Prozessor::push(w: Wert):void
post: pc = pc@pre+1+Wert.size
post: sp = sp@pre - Wert.size
post: mem.getVal(sp) = w
```



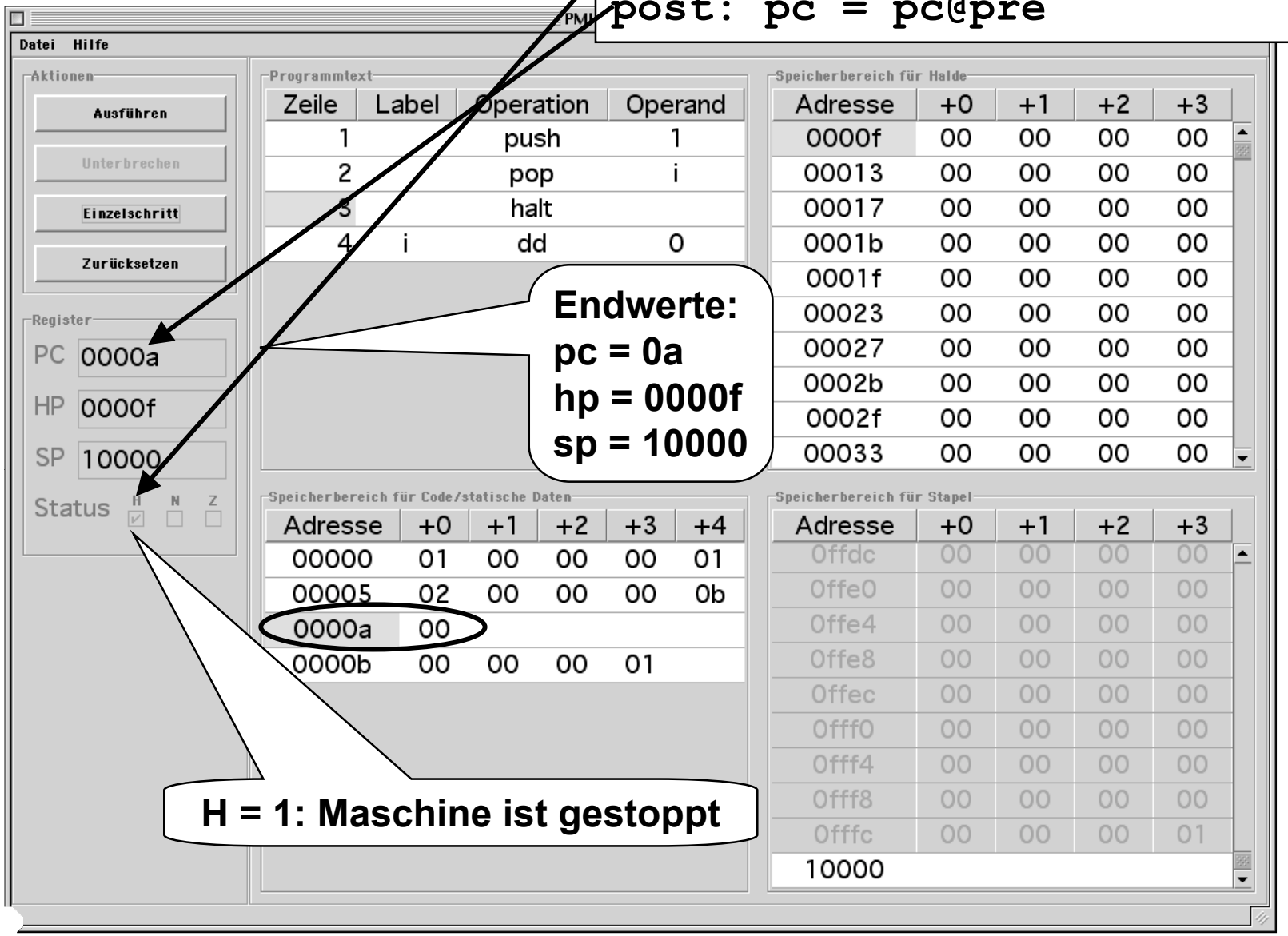
Zustand nach pop i

```
Prozessor::pop(a: Adresse):void
post: pc = pc@pre+1+Wert.size
post: sp = sp@pre + Wert.size
post: mem.getVal(a) =
      mem@pre.getVal(sp@pre)
```



Zustand nach `halt`

```
Prozessor::halt():void
post: status.ishalted() = true
post: pc = pc@pre
```



Spezifikation der arithmetischen Operationen

- **add ()**:
  Nimmt die 2 obersten Elemente vom Stapel, addiert sie und legt das Ergebnis als oberstes Element auf den Stapel.

```
Prozessor::add():void
post: pc = pc@pre + 1
post: sp = sp@pre + Wert.size
post: mem.getVal(sp) =
        mem@pre.getVal(sp@pre +
                       Wert.size)
        + mem@pre.getVal(sp@pre)
```

- **sub ()**:
  Subtraktion (analog zu **add()**)
- **mult()**:
  Multiplikation (analog zu **add()**)
- **div()**:
  Division (analog zu **add()**)

Prozessor
halt ()
push (Wert)
pop (Adresse)
delete ()
add ()
sub ()
mult ()
div ()
compare ()
jump (Adresse)
jumpZero (Adresse)
jumpNegative (Adresse)
jumpSubroutine (Adresse)
return ()

Berechnung eines arithmetischen Ausdrucks

- Gegeben sei der Ausdruck in Postfix-Notation (*"polnische Notation"*):

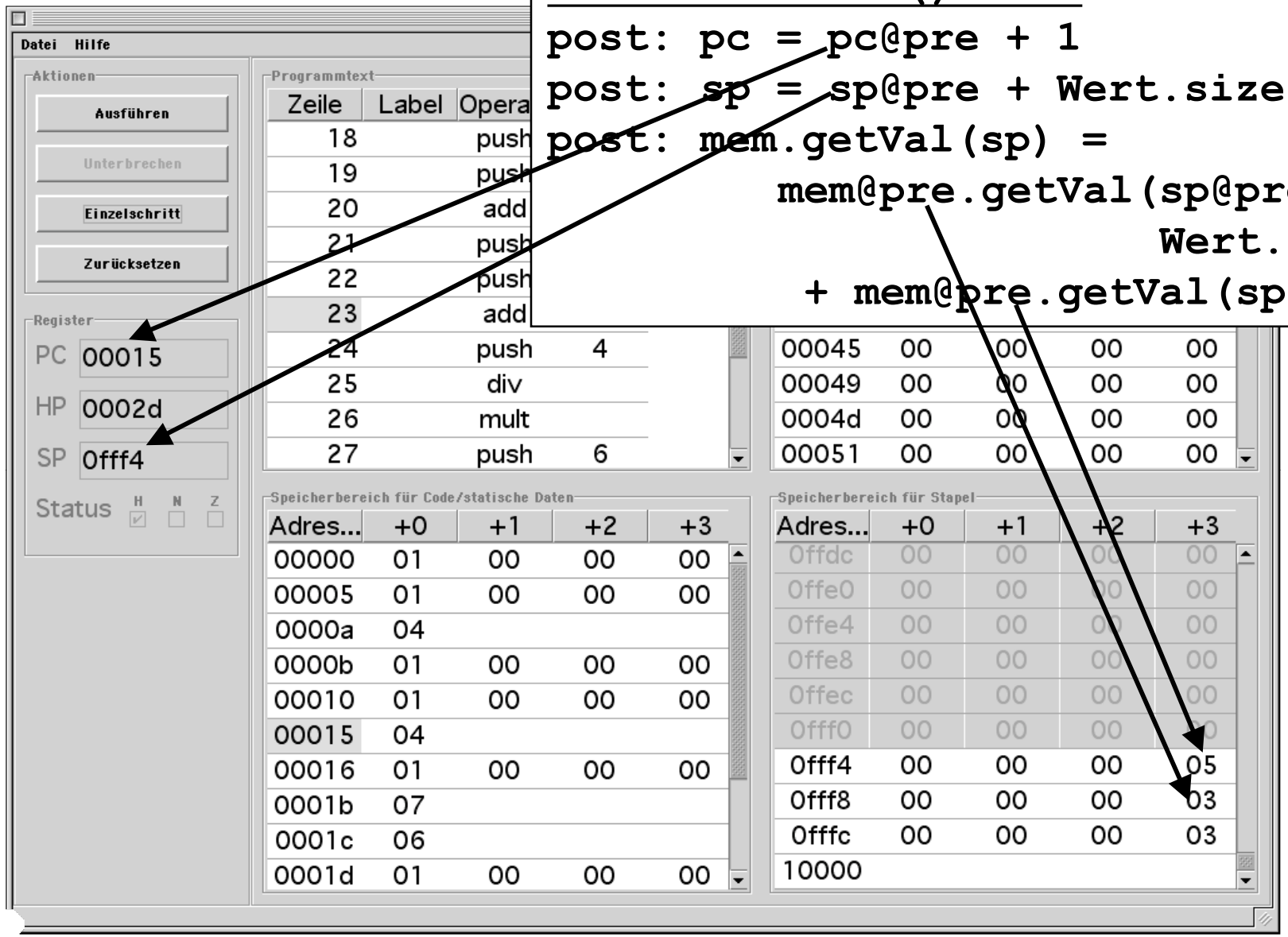
  `1 2 + 3 5 + 4 / * 6 *`

- PMI-Programm, das diesen Ausdruck berechnet:

```
push    1
push    2
add             //    1 2 +
push    3
push    5
add             //    3 5 +
push    4
div             //    3 5 + 4 /
mult            //    1 2 + 3 5 + 4 / *
push    6
mult            // 1 2 + 3 5 + 4 / * 6 *
```

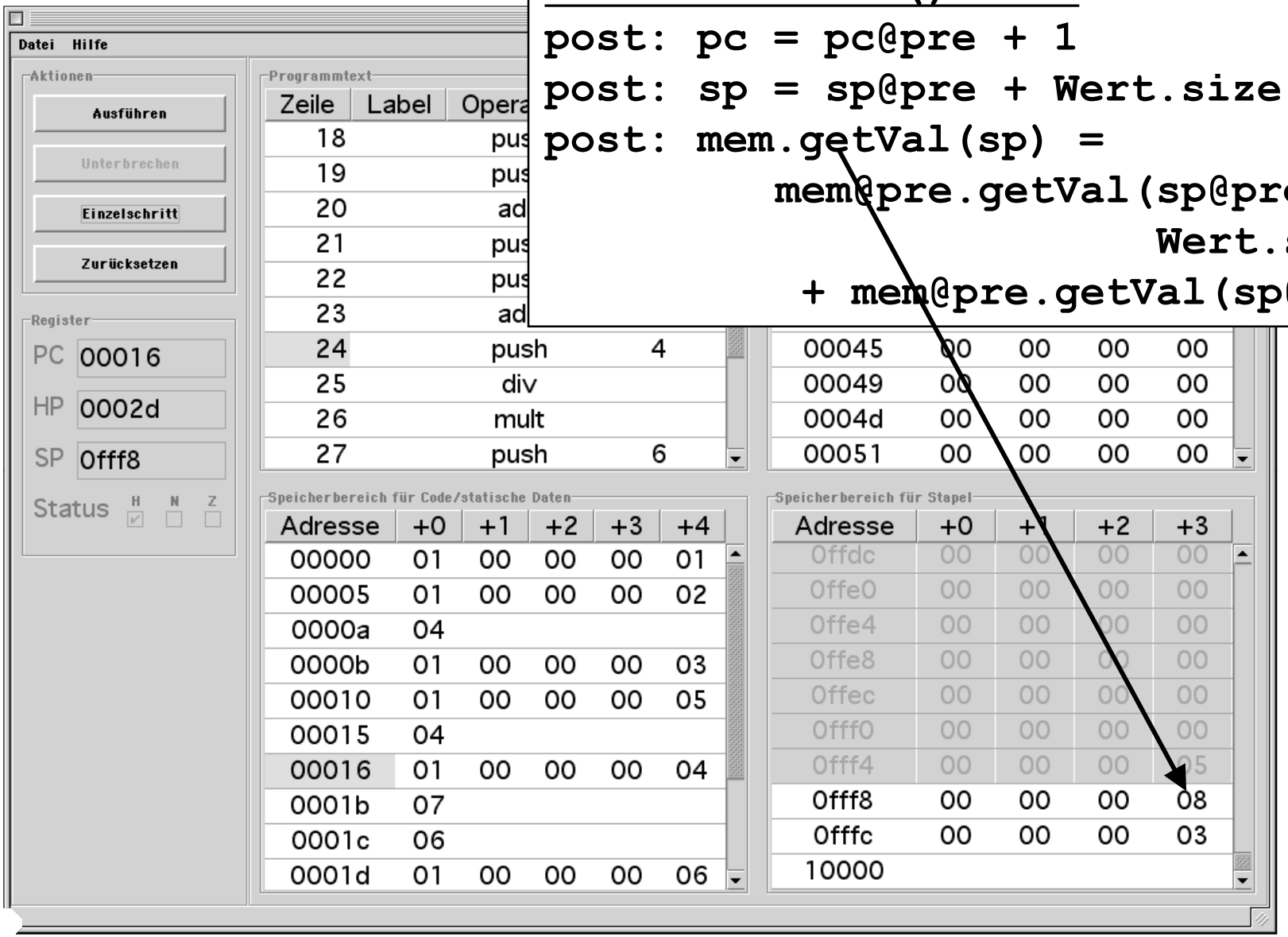
Zustand vor 1 2 + 3 5 +

```
Prozessor::add():void
post: pc = pc@pre + 1
post: sp = sp@pre + Wert.size
post: mem.getVal(sp) =
        mem@pre.getVal(sp@pre +
                       Wert.size)
      + mem@pre.getVal(sp@pre)
```

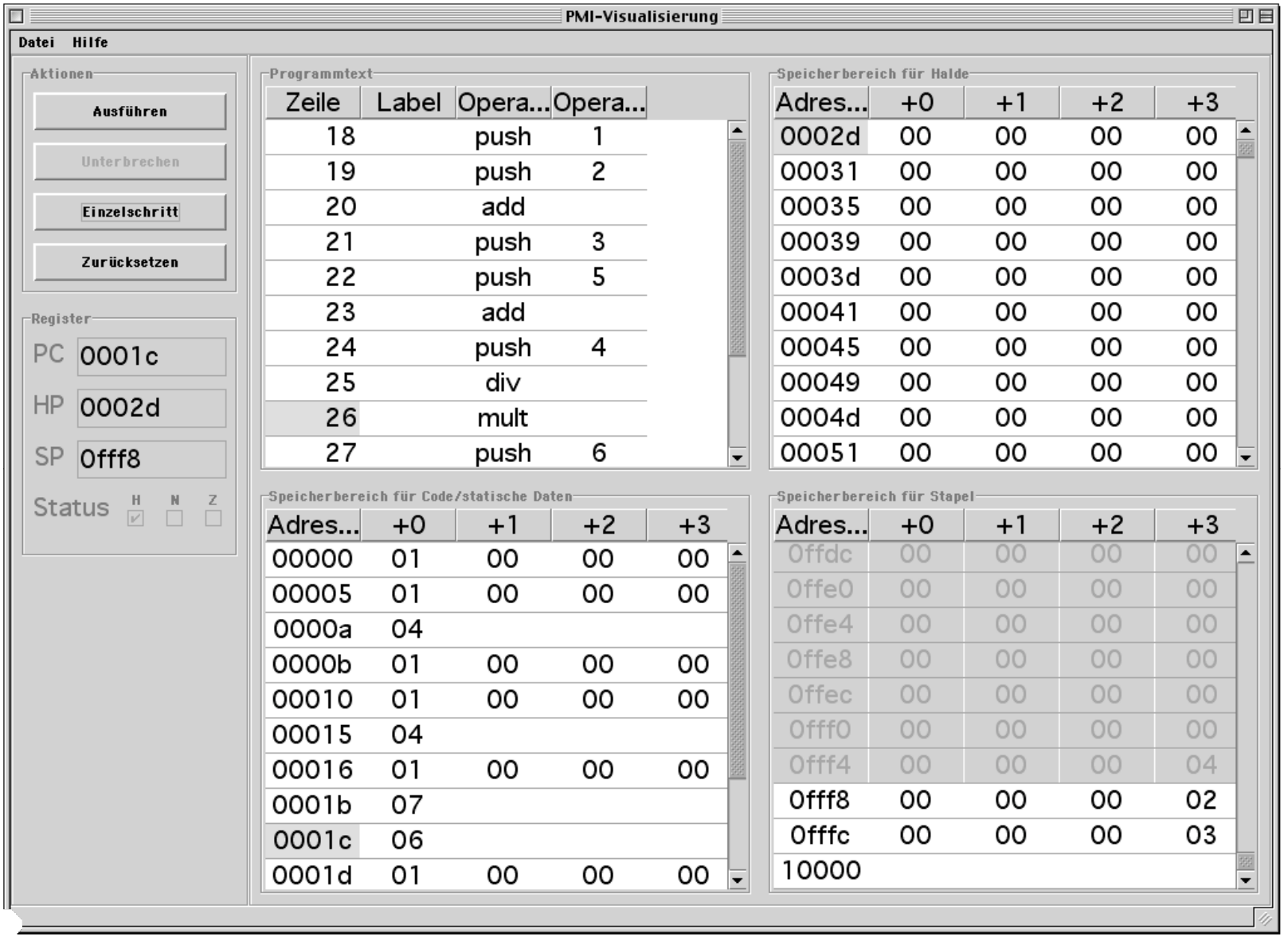


Zustand nach 1 2 + 3 5 +

```
Prozessor::add():void
post: pc = pc@pre + 1
post: sp = sp@pre + Wert.size
post: mem.getVal(sp) =
          mem@pre.getVal(sp@pre +
                                Wert.size)
           + mem@pre.getVal(sp@pre)
```



Zustand nach 1 2 + 3 5 + 4 /

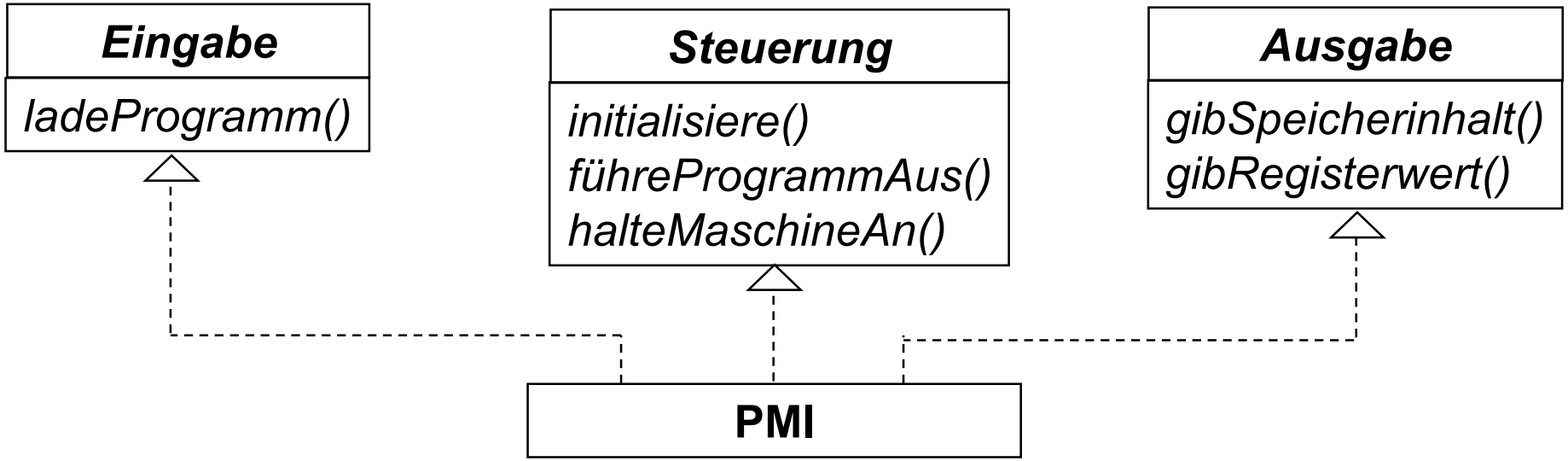


Mehr über PMI 7/8/2004

- Den gesamten Java-Quellcode von PMI finden Sie unter
  http://wwwbruegge.in.tum.de/Lehrstuhl/Informatik2SoSe2004PMI
- Einige Einschränkungen
  - im Ein-/Ausgabebereich
  - Steuerwerk
- Implementierung von PMI mit Modell-Sicht-Steuerungs-Muster.
  - Benutzt ein Java-Konstrukt, das wir in der Vorlesung nicht erklärt haben: Swing-Klassen (graphische Bedienoberfläche)
- Repräsentation und Codierung von Werten in PMI
- Addressierung im PMI-Assembler
- Spezifikation weiterer Operationen
  - Vergleich
  - Sprungoperationen
  - Unterprogrammaufruf

Einschränkungen der PMI

- Die PMI verfügt nicht über alle Komponenten einer vollwertigen von-Neumann-Maschine. Es fehlen insbesondere
  - programmierbare Ein- bzw. Ausgabe-Prozessoren
  - ein programmierbares Steuerwerk
- Die dadurch fehlende Funktionalität wird durch die Implementierung der entsprechenden Schnittstellen in PMI teilweise nachgebildet:

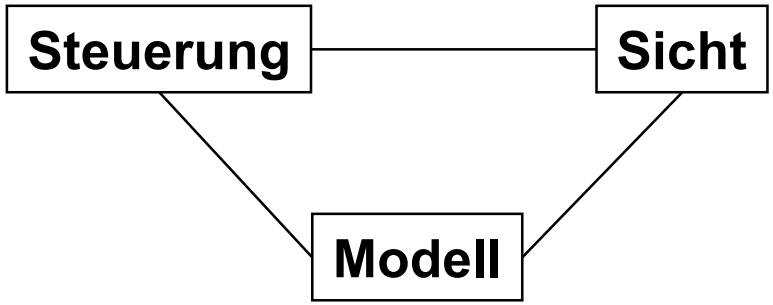


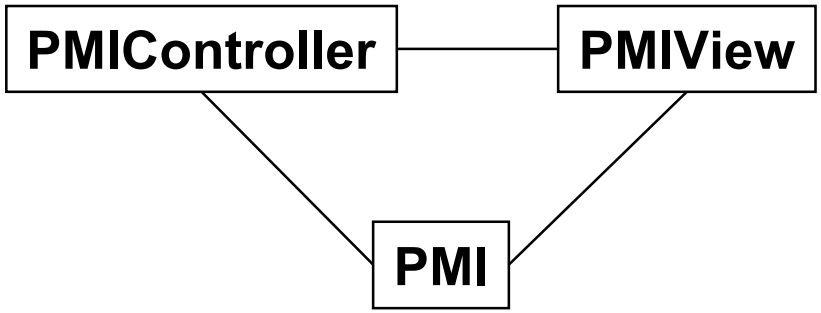
- **Hinweis**: Diese Einschränkungen gelten nicht für die MI (⇒ Info III)

Implementierung der PMI: Modell-Sicht-Steuerung

- Anwendungs-spezifische Daten und Berechnungen sind Teil des ***Modells***.
- Dem Benutzer wird eine ***Sicht*** auf das Modell präsentiert, in der Daten und Berechnungsergebnisse aufbereitet werden.
- Eingaben durch den Benutzer werden über die ***Steuerung*** an das Modell weitergegeben.
- **Vorteile**:
  - verschiedene Sichten auf ein Modell (z.B. GUI, Textausgabe), ohne den Anwendungsablauf (im Modell) verändern zu müssen.
  - gleichzeitige Nutzung eines Modells durch verschiedene Benutzer (Modell-Zugriff wird durch Steuerung koordiniert).

Implementierung der PMI: Modell-Sicht-Steuerung (2)

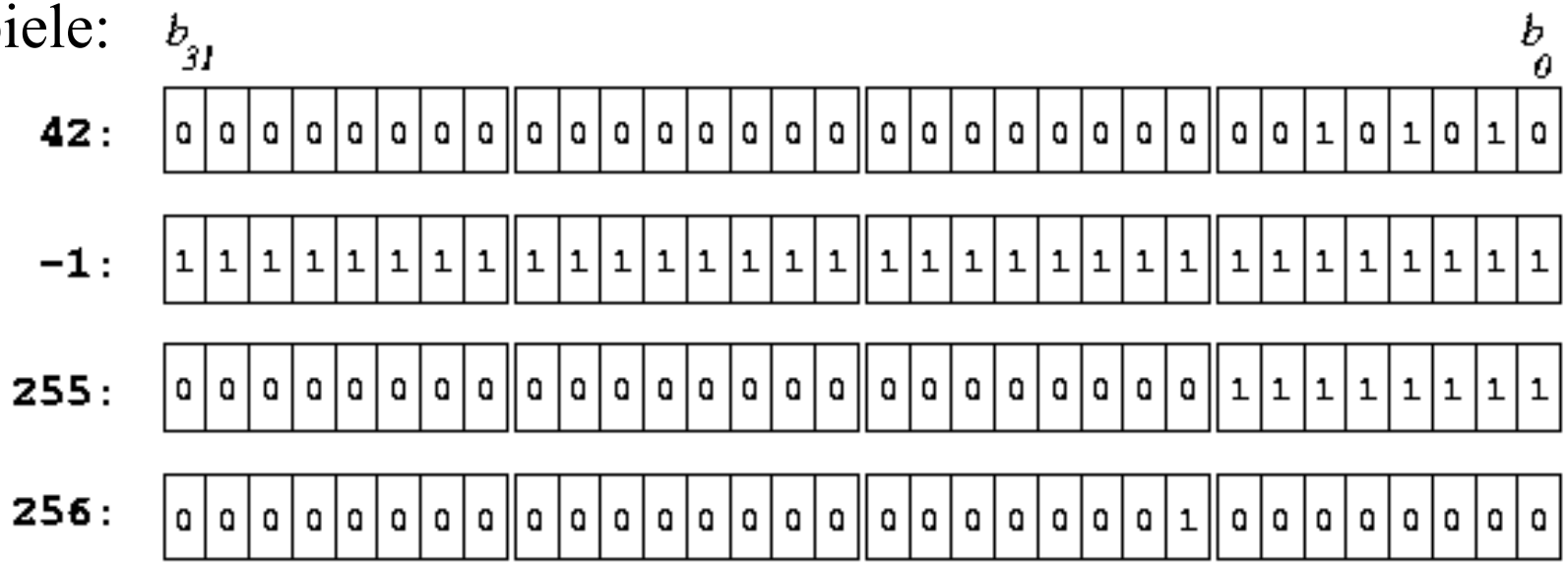




- Interaktive Anwendungen kombinieren oft die Komponenten für Datenausgabe und Dateneingabe incl. Steuerung in einer gemeinsamen Bedienoberfläche.
  - Dadurch sind Sicht und Steuerung eng gekoppelt.
  - Ein Indiz für die enge Kopplung bei der Implementierung von PMI ist, dass die Klassen **`PMIController`** und **`PMIView`** im gleichen Paket (**`pmi.view`**) abgelegt sind.
  - Die Modell-Komponenten (z.B. die Schnittstelle **`PMI`**) befinden sich dagegen in einem anderen Paket (**`pmi.model`**).

Repräsentation und Codierung von Werten in PMI

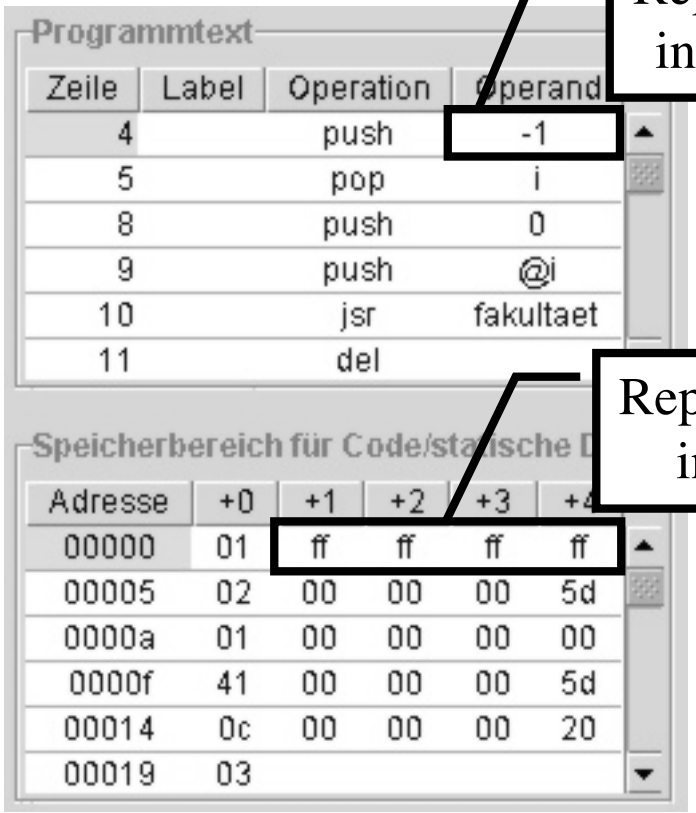
- In PMI-Assemblerprogrammen werden Werte als ganze Zahlen in Dezimal-Darstellung (zur Basis 10) *repräsentiert*.
  - Beispiele: **`42`**, **`-1`**, **`255`**, **`256`**
- Für die Verwendung in der PMI werden diese Werte binär *codiert*. Für jeden Wert stehen dabei 4 Byte (= 32 bit) zur Verfügung.
- Negative Zahlen werden im sog. *Zweier-Komplement* codiert.
- Beispiele:



Visualisierung von Werten im PMI-Arbeitsspeicher

- Der Inhalt einer einzelnen Speicherzelle (1 Byte) des PMI-Arbeitsspeichers wird als vorzeichenlose Zahl in Hexadezimal-Darstellung (Basis 16) repräsentiert ⇒ 2 Ziffern pro Speicherzelle
- Führende Nullen werden angegeben
- Beispiele:

```
  -1 → ffffffff
  93 → 0000005d
 255 → 000000ff
 256 → 00000100
```



Repräsentation des Werts *-1* in PMI-Assemblersprache

Repräsentation des Werts *-1* in PMI-Maschinencode

Adressierung in PMI-Assembler

- **Unmittelbare Adressierung**

```
push    5
push    i
```

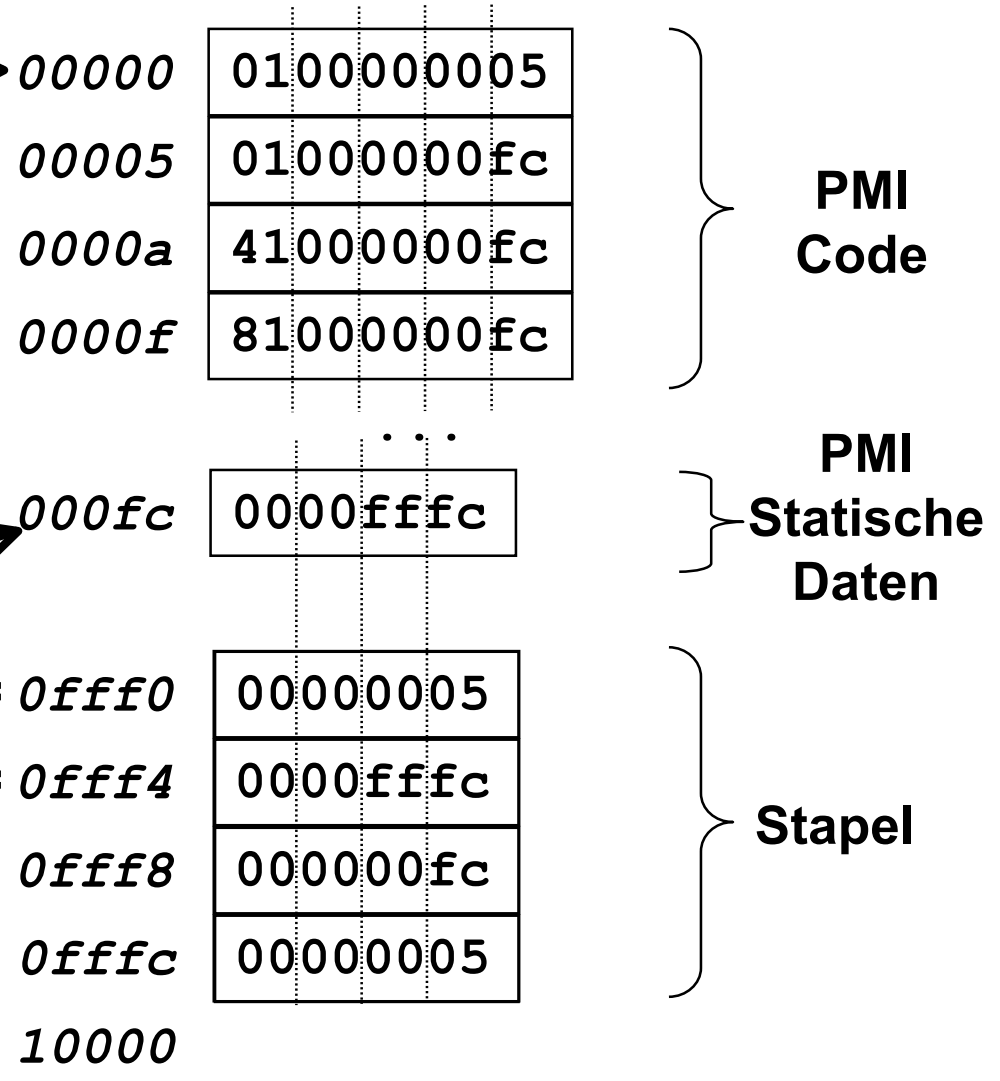
- **Direkte Adressierung:**

```
push    @i
```

- **Indirekte Adressierung:**

```
push    >i
  ...
i: dd 65532
```

```
Adresse von i: 000fc
Wert von i: 65532, x0000fffc
```



Spezifikation der Vergleichs-Operation

- **compare ()**:
  Vergleicht das oberste Element vom Stapel mit 0 und aktualisiert das Statusregister entsprechend dem Vergleichsergebnis.

```
Prozessor::compare():void
post: pc = pc@pre + 1
post: status.isNegative() =
        (mem.getVal(sp) < 0)
post: status.isZero() =
        (mem.getVal(sp) == 0)
```

Prozessor
halt ()
push (Wert)
pop (Adresse)
delete ()
add ()
sub ()
mult ()
div ()
compare ()
jump (Adresse)
jumpZero (Adresse)
jumpNegative (Adresse)
jumpSubroutine (Adresse)
return ()

Spezifikationen der Sprung-Operationen

- **jump (Adresse)**:
  Springt zur Argument-Adresse

```
Prozessor::jump(a:Adresse):void
post: pc = a
```

- **jumpNegative (Adresse)**:
  Springt zur Argument-Adresse, wenn das Statusregister *"negativ"* ist.

```
Prozessor::jumpNegative
 (a:Adresse):void
post: not(status.isNegative())
       implies pc =
        pc@pre + 1 + Wert.size
post: status.isNegative()
       implies pc = a
```

- **jumpZero (Adresse)**:
  Springt zur Argument-Adresse, wenn das Statusregister *"Null"* ist.
  (analog zu **jumpNegative (Adresse)**)

Prozessor
halt ()
push (Wert)
pop (Adresse)
delete ()
add ()
sub ()
mult ()
div ()
compare ()
jump (Adresse)
jumpNegative (Adresse)
jumpZero (Adresse)
jumpSubroutine (Adresse)
return ()

Übersetzung von Java nach PMI: While-Anweisung

```
 i = 10; while (i > 0) { i = i -1; }
```

```
       push   10
       pop    i          // i = 10
loop:  push   @i
       push   1
       sub
       pop    i          // i--
test:  push   @i         // Lege i auf den Stapel
       comp              // Vergleiche i mit 0
       del               // Lösche i vom Stapel
       jmpn   ende       // i<0 → Springe nach ende
       jmpz   ende       // i=0 → Springe nach ende
       jump   loop       // Springe nach loop
ende:  halt
i:     dd     0
```

`@i`: Direkte Adressierung

```
operand = memory.getVal(operandCode);
```

Spezifikation der Unterprogramm-Operationen

- **jumpSubroutine (Adresse)**:
  Legt die Adresse des nächsten Befehls als oberstes Element auf den Stapel („Rücksprungaddresse“) und springt dann nach **Adresse**

```
Prozessor::jumpSubroutine
 (a:Adresse):void
post: sp = sp@pre - Wert.size
post: mem.getVal(sp) =
       pc@pre + 1 + Wert.size
post: pc = a
```

- **return ()**:
  Nimmt das oberste Element als Adresse vom Stapel und springt zu dieser Adresse.

```
Prozessor::return():void
post: sp = sp@pre + Wert.size
post: pc =
       mem@pre.getVal(sp@pre)
```

- Unterprogrammaufruf:wichtiges Konzept.

Prozessor
halt ()
push (Wert)
pop (Adresse)
delete ()
add ()
sub ()
mult ()
div ()
compare ()
jump (Adresse)
jumpNegative (Adresse)
jumpZero (Adresse)
jumpSubroutine (Adresse)
return ()

Zusammenfassung

- Eine ***von-Neumann-Maschine*** (wie z.B. die PMI) besteht aus Eingabewerk, Steuerwerk, Rechenwerk, Arbeitspeicher und Ausgabewerk.
- **Programme und Daten** werden im ***Arbeitsspeicher*** und ***Registern*** gespeichert.
- Das Konzept der ***Programmsteuerung***:
  - *"Programme sind auch nur Daten"*
- Der Zustand einer von-Neumann-Maschine ist die Belegung aller Register und Speicherzellen
  - Ein ***Maschinenprogramm*** besteht aus vielen Maschinenbefehlen.
  - Die Ausführung eines ***Maschinenbefehls*** verändert im allgemeinen den Zustand der Maschine.
- Ein Maschinenbefehl besteht aus ***Opcode*** und ***Operanden***. Zur genaueren Bestimmung des Operanden gibt ***Adressmodi*** und ***Adressierungsarten***.