

Einführung in die Informatik II Struktur von Rechenanlagen

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

Sommersemester 2001

11. - 16. Juli 2001

Gliederung des nächsten Vorlesungsblockes

✓ 9. Juli: Übersetzung von Modellierungssprachen (UML) in Programmiersprachen (Java)

❖ 11. Juli (Heute): Struktur von Rechenanlagen

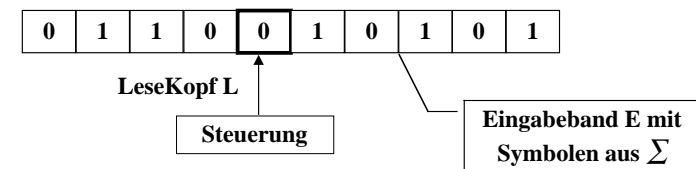
- Von endlichen Automaten zur von-Neumann-Maschine
- Aufbau der von-Neumann-Maschine:
- Modellierung einer primitiven von-Neumann-Maschine: PMI
 - Komponenten der PMI-Maschine
 - Befehlsatz der PMI-Maschine
 - Operandenspezifikation und Adressrechnung
 - Befehlszyklus
- Das Konzept der Programmsteuerung
- PMI-Assembler
- "*Hinter dem Vorhang*": Implementierung von PMI

Vorausschau

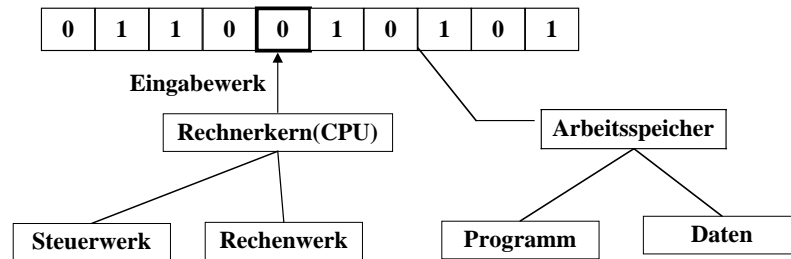
❖ 16. Juli (Montag): Maschinennahe Programmierung und Übersetzung von Quellsprachen in Maschinennahe Sprachen

- Maschinennahe Programmierung von PMI
 - Einige Beispiele
- Manuelle Übersetzung von
 - Ausdrücken
 - While Schleifen
 - Methodenaufruf ("Unterprogrammprung")
 - Rekursion
- Struktur des Laufzeitstapels
 - Aktivierungssegmente
- Assembler vs Compiler

Vom endlichen Automaten ...



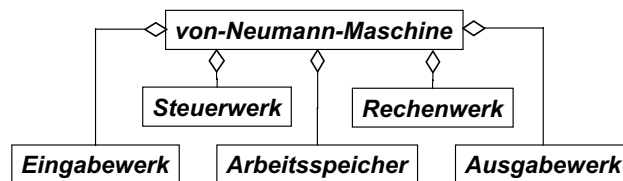
Vom endlichen Automaten zum Rechner



Aufbau von Rechenanlagen

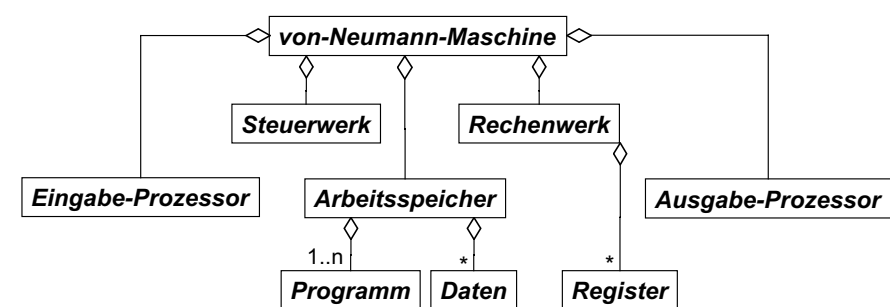
- ❖ Im folgenden besprechen wir den Aufbau von Rechenanlagen, wie sie prinzipiell zur Ausführung von Informatik-Systemen verwendet werden. Wir arbeiten dabei einige wichtige Konzepte heraus, die vielen Rechenanlagen gemeinsam sind:
 - *Die von-Neumann-Maschine*
 - *Das Prinzip der Programmsteuerung*
 - *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*.

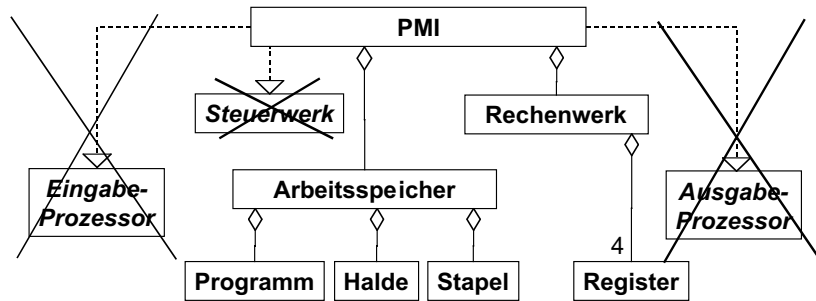
Wichtige Eigenschaften von von-Neumann-Maschinen



- ❖ Wichtige Eigenschaften der von-Neumann-Maschine:
 - **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:



- ❖ **Bemerkung:** In Info III wird für die Systemprogrammierung die **MI** als Maschine eingeführt. Die PMI ist weniger leistungsfähig ("primitiver") als die MI, aber dafür 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.

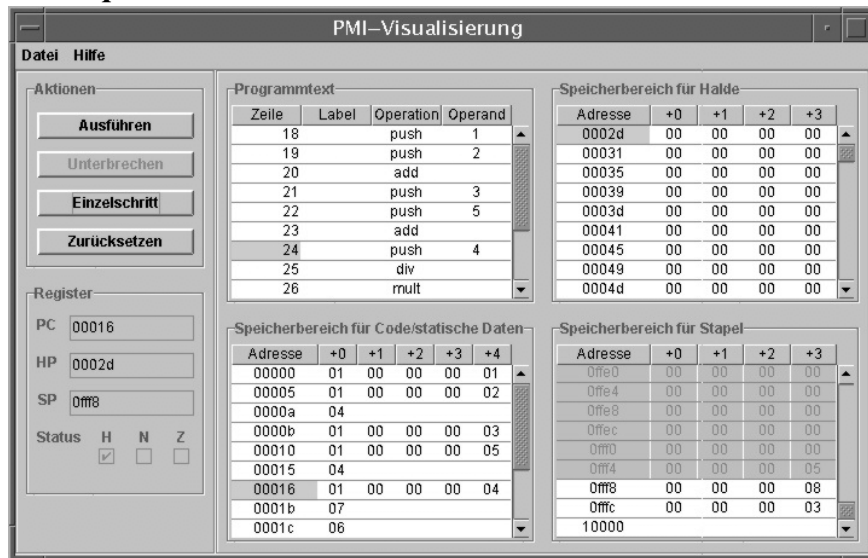
Die PMI-Maschine als Modell

- ❖ Die PMI ist ein Modell einer sehr einfachen Rechenanlage vom Typ eines von-Neumann-Rechners.
- ❖ Die PMI ist auch ein Informatik-System. Wir können deshalb ihre funktionellen, statischen und dynamischen Aspekte beschreiben.
 - Das **funktionelle PMI-Modell** beschreibt die Funktionalität der PMI, insbesondere Operationen zum Betreiben der Rechenanlage und Operationen ("Maschinenbefehle") zum Ausführen von Programmen.
 - Das **statische PMI-Modell** beschreibt die statischen Aspekte: Speicher, Prozessor, Adressregister, Statusregister, und die Semantik von Maschinenbefehlen (als Vor- und Nachbedingungen).
 - Im **dynamischen PMI-Modell** beschreiben wir die Arbeitsweise des Steuerwerkes, insbesondere die Exekution von Maschinenbefehlen.
- ❖ Zur Veranschaulichung der PMI-Komponenten wählen wir außerdem eine geeignete **Sicht**: ⇒ 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



Copyright 2001 Bernd Brügge

Einführung in die Informatik II: TUM Sommersemester 2001

15:36 13

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 Gleitkommaarithmetik)
 - Logische Operationen (auf Binärworten)
 - Operationen mit Adressen
 - Schiebepbefehle
 - Programmablaufbefehle, Sprungbefehle, Unterprogrammbefehle
 - Ein-/Ausgabebefehle
 - Steuerungsbefehle (Privilegierte Befehle)

Copyright 2001 Bernd Brügge

Einführung in die Informatik II: TUM Sommersemester 2001

15:36 14

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 Schiebepbefehle, keine logische Operationen und keine Ein-/Ausgabebefehle.

Copyright 2001 Bernd Brügge

Einführung in die Informatik II: TUM Sommersemester 2001

15:36 15

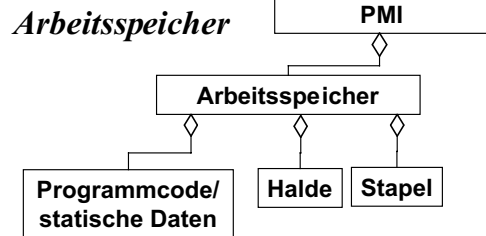
PMI-Modellierung

- ❖ **Statisches Modell:**
 - PMI Komponenten:
 - Arbeitsspeicher
 - Prozessor
 - Adressregister
 - Statusregister
 - Speicherorganisation
 - Verwaltung des Arbeitsspeichers mit Arbeitsregistern
 - Modellierung eines PMI-Programms
 - Modellierung und Realisierung eines PMI-Befehls
- ❖ **Dynamisches Modell:**
 - Maschinenbefehlszyklus
 - Adressierungsarten

Copyright 2001 Bernd Brügge

Einführung in die Informatik II: TUM Sommersemester 2001

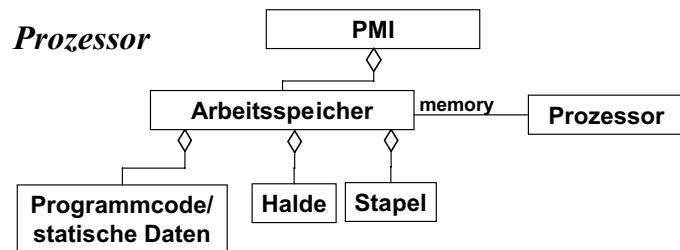
15:36 16



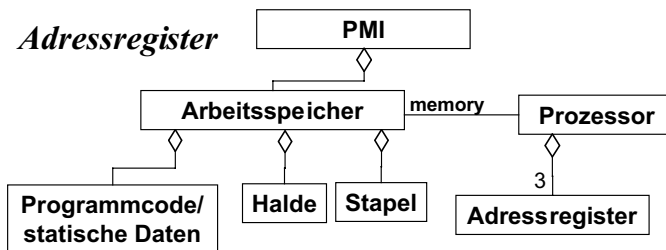
Der Arbeitspeicher ist in drei Bereiche unterteilt:

- **Code/statische Daten:**
 - (binärer) Programmcode
 - *statische* (vor dem Programmablauf) erzeugte Daten
- **Halde (heap):**
 - *dynamisch* (während des Programmablaufs) erzeugte Daten
- **Stapel (stack):**
 - Operanden und Ergebnisse von PMI-Operationen
 - Parameter und lokale Variablen für Unterprogramme

Visualisierung des Arbeitspeichers



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



❖ 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

Visualisierung der Halde (hp = Anfangsadresse der Halde)

Direkte Visualisierung der Registerwerte

Visualisierung des Programmcodes (pc = Adresse der nächsten PMI-Operation)

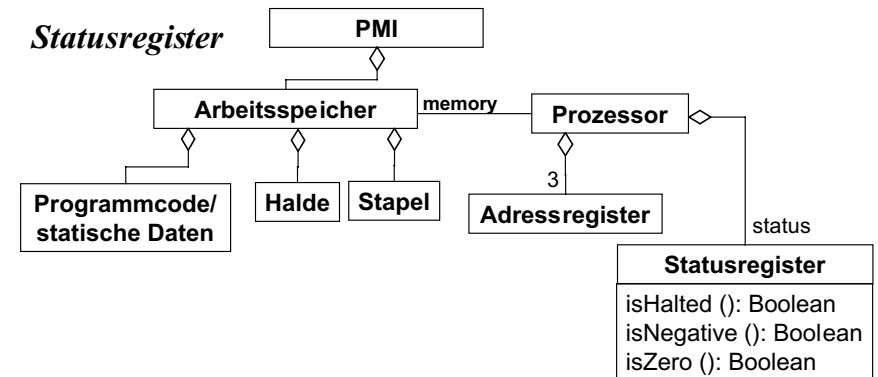
Visualisierung des Stapels (sp = Adresse des obersten Stapелеlements)

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	

Adresse	+0	+1	+2	+3
0002d	00	00	00	00
00031	00	00	00	00
00035	00	00	00	00
00039	00	00	00	00
0003d	00	00	00	00
00041	00	00	00	00
00045	00	00	00	00
00049	00	00	00	00
0004d	00	00	00	00

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				

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

War der zuletzt überprüfte Wert Null?

War der zuletzt überprüfte Wert negativ?

Ist die Maschine angehalten?

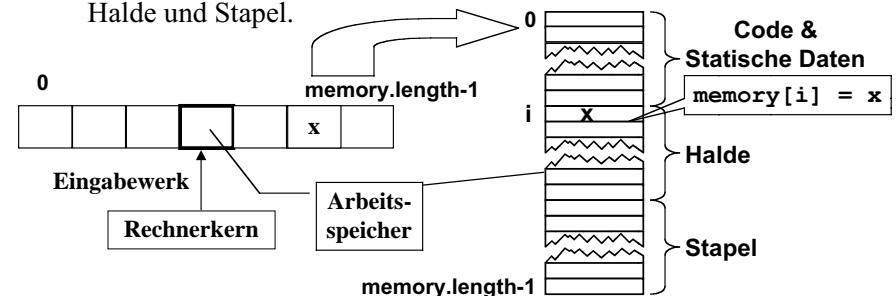
Zeile	Label	Operation	Operand
18		push	1
19		push	2
20		add	
21		push	3
22		push	5
23		add	
24		push	4
25		div	
26		mult	

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				

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:

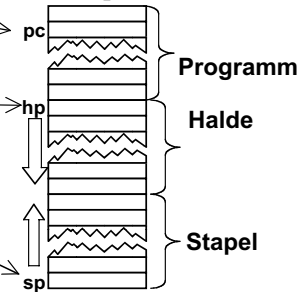

```
byte[] memory;
```
- Die Index-Position **i** einer Speicherzelle **x** ist die **Adresse** von **x**.
- Der Arbeitsspeicher ist aufgeteilt in einen Programmcode-Bereich, Halde und Stapel.



Verwaltung des Arbeitsspeichers mit Adressregistern

❖ Die 3 **Adressregister** verwalten die Bereiche im Arbeitsspeicher:

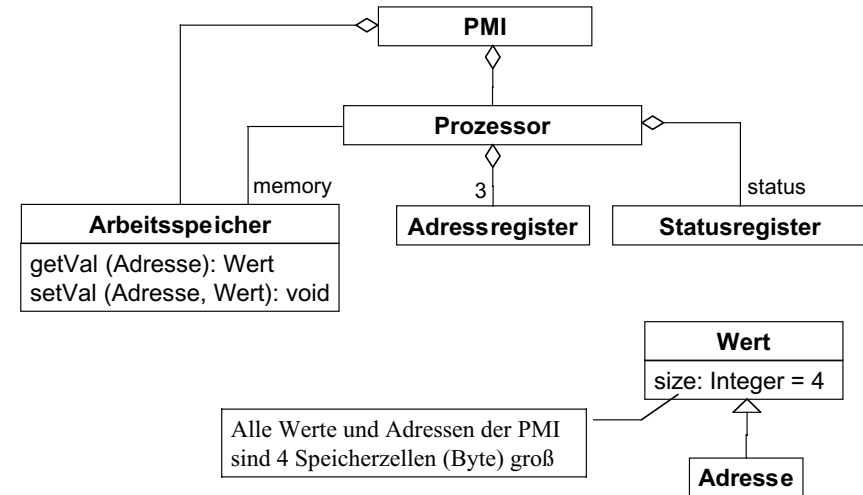
- **pc**: Adresse der nächsten auszuführenden Operation
- **hp**: Anfangsadresse der Halde. Die Halde wächst von "oben nach unten" (von niedrigen Adressen zu hohen Adressen)
- **sp**: Adresse des obersten Stapel-Elementes. Der Stapel wächst von "unten nach oben" (von hohen Adressen zu niedrigen Adressen)



❖ **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



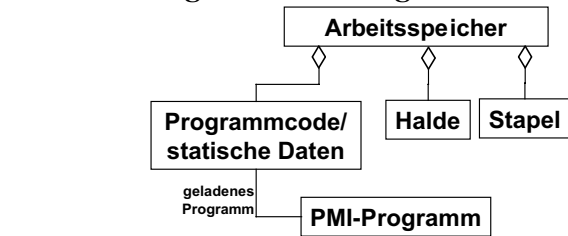
Befehle und Zustände

- ❖ 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?

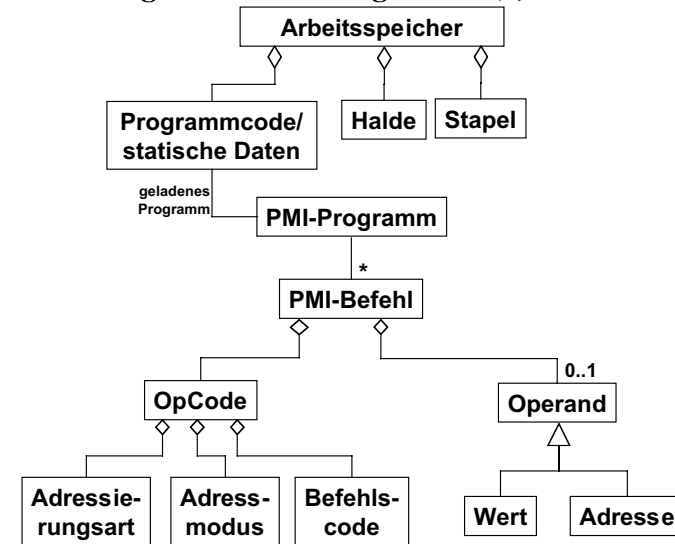
- ❖ Wir werden jetzt den Aufbau von PMI-Programmen modellieren
- ❖ Dann werden wir die Semantik der einzelnen Operationen des Befehlssatzes beschreiben.
 - Wir benutzen dazu das Konzept des Vertrages. Ein Befehl wird durch Vor- und Nachbedingungen beschrieben.
- ❖ Den gesamten Befehlssatz führen wir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

Modellierung von PMI-Programmen



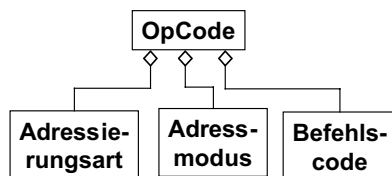
Bei einer **programmgesteuerten Rechenanlage** werden Programme im **Arbeitspeicher** gespeichert

Modellierung eines PMI-Programms (2)

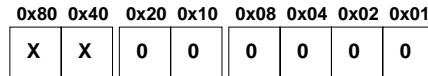


OpCode

❖ **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):



Adressierungsart Adressmodus Befehlscode

0 0 unmittelbare Adressierung

0 1 direkte Adressierung

1 0 indirekte Adressierung

Der gesamte PMI-Befehlssatz als Schnittstelle (PMIInstructionSet.java)

```
public interface PMIInstructionSet {
    // Die einzelnen Instruktionen
    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;

    // 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;
    ....
} // PMIInstructionSet
```

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

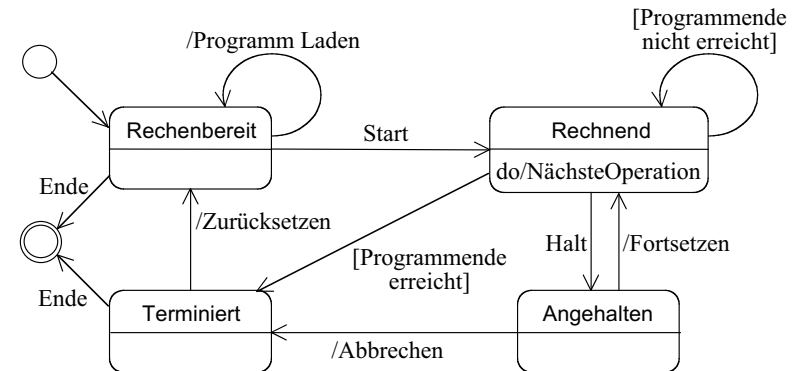
Implementierung des PMI-Befehlssatzes (*Prozessor.java*)

- ❖ 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 ()
	compare ()
Vergleich	jump (Adresse)
	jumpZero (Adresse)
Sprünge	jumpNegative (Adresse)
	jumpSubroutine (Adresse)
	return ()
Unterprogramm-Aufruf	

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:

- Einlesen des OpCodes der Operation aus dem Speicher:**
`byte opCode = memory[pc];`
- Bestimmung der dem OpCode entsprechenden Operation (z.B. add oder halt)**
- Bestimmen des Operandenwerts** (für Befehle mit einem Operanden):
 - Bestimme den Adressmodus des Operanden (nächste Folie)
 - Bestimmen die Adressierungsart (übernächste Folie)
- Ausführen des Befehls:** Wenn **Operation == halt**, halte die Maschine an, sonst verändere Speicher- und/oder Registerwerte entsprechend der Spezifikation des Befehls.
- 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:** **operandCode** ist bereits der Operand.
Wert operand = operandCode ;
 - 2. Direkte Adressierung:** **operandCode** ist die Speicheradresse, an der der Operand gespeichert ist:
Wert operand = memory.getVal(operandCode) ;
 - 3. Indirekte Adressierung:** **operandCode** ist die Speicheradresse, an der die Speicheradresse des Operanden gespeichert ist.
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				PMI-Maschinenprogramm					
Zeile	Label	Operation	Operand	Adresse	+0	+1	+2	+3	+4
18		push	1	00000	01	00	00	00	01
19		push	2	00005	01	00	00	00	02
20		add		0000a	04				
21		push	3	0000b	01	00	00	00	03
22		push	5	00010	01	00	00	00	05
23		add		00015	04				
24		push	4	00016	01	00	00	00	04
25		div		0001b	07				
26		mult		0001c	06				
27		push	6	0001d	01	00	00	00	06
28		mult		00022	06				
29		pop	i	00023	02	00	00	00	29
30		halt		00028	00				
31	i:	dd	0	00029	00	00	00	00	

Visualisierung von PMI-Assembler-/Maschinenprogrammen

The screenshot shows a window titled "PMI-Visualisierung" with several panes:

- Programmtext:** A table showing assembly instructions with their addresses and operands. A callout box points to the first row: "Darstellung des Assemblerprogramms (ohne Kommentare)".
- Speicherbereich für Halde:** A table showing the corresponding machine code in hexadecimal. A callout box points to the first row: "Darstellung des Maschinenprogramms. Der Maschinencode für einen PMI-Befehl wird jeweils in einer eigenen Zeile dargestellt (mit Adresse)".
- Register:** A section for register status.
- Status:** A section for status flags (H, N, Z).
- Buttons:** "Einzelschritt" and "Zurücksetzen" are visible.

Additional callout boxes provide more context:

- "Ein PMI-Befehl wird jeweils in einer eigenen Zeile dargestellt (mit Zeilennummer)" points to the assembly table.
- "für Code/statische Daten" points to the machine code table.

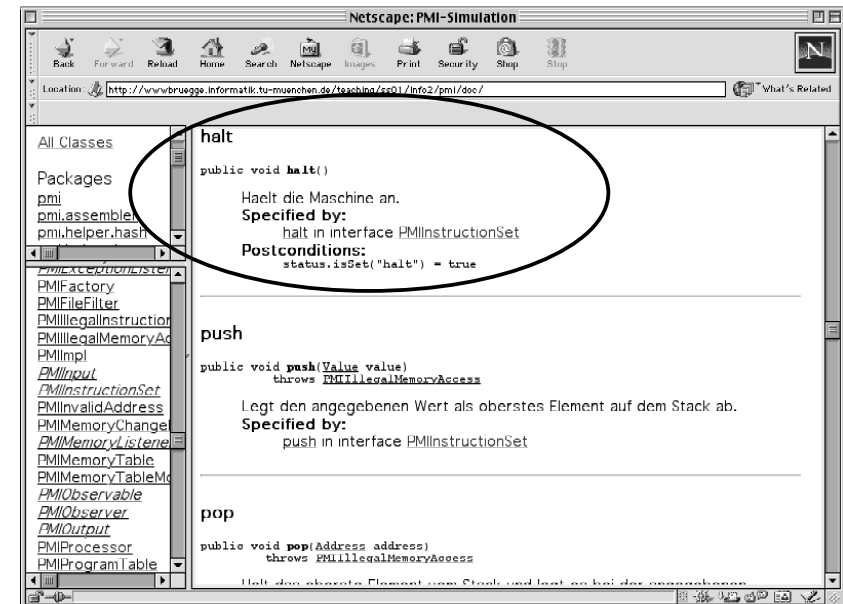
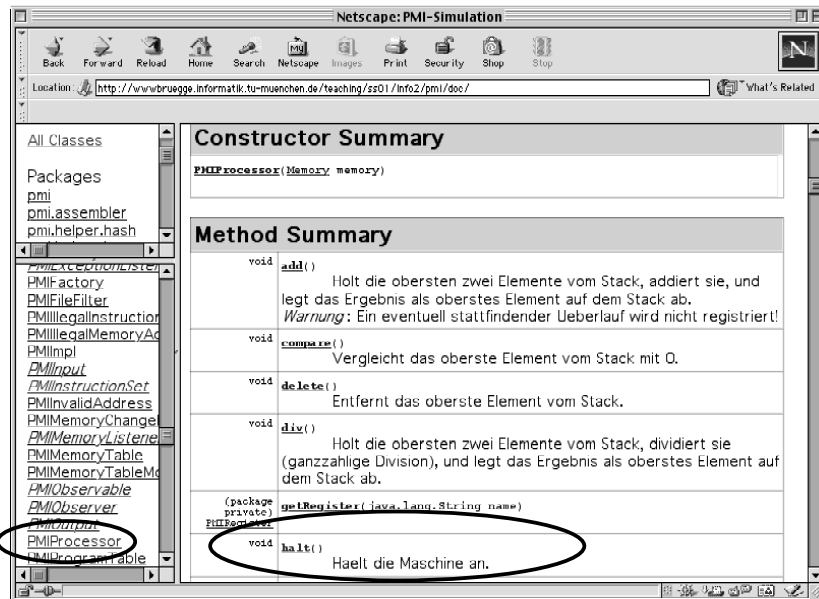
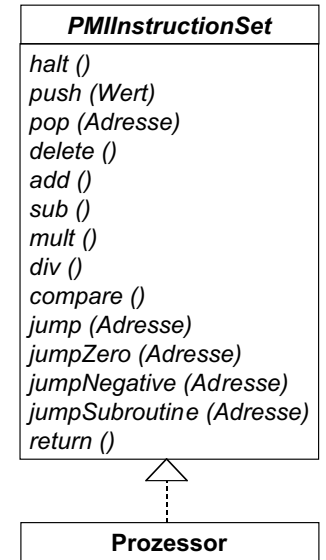
PMI-Assembler: Syntax (in EBNF)

```

⟨PMI-Programm⟩ ::= ⟨PMI-Programmzeile⟩*
⟨PMI-Programmzeile⟩ ::= ⟨PMI-Anweisung⟩ [ ⟨Kommentar⟩ ] |
    ⟨Kommentar⟩
⟨PMI-Anweisung⟩ ::= [ ⟨Adressbezeichner⟩ : ] Leerzeichen ⟨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://www.bruegge.in.tum.de/teaching/ss01/Info2/pmi/doc/> (im weiteren Text mit **\$PMIDOC** abgekürzt)
- ❖ Im Folgenden schauen wir uns diese Spezifikation genauer an.



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

The screenshot shows the PMI-Visualisierung interface with the following components:

- Assembler-Programm:** A table showing the assembly code:

Zeile	Label	Operation	Operand
1		push	1
2		pop	i
3		halt	
4	i	dd	0
- Anfangswerte (Initial Values):**
 - pc = 0
 - hp = 0000f
 - sp = 10000
- Maschinen-Programm (Machine Program):** A table showing the initial memory state:

Adresse	+0	+1	+2	+3	+4
00000	01	00	00	00	01
00005	02	00	00	00	0b
0000a	00				
0000b	00	00	00	00	
- Speicherbereich für Halde (Stack):** A table showing the initial stack state:

Adresse	+0	+1	+2	+3
0000f	00	00	00	00
00013	00	00	00	00
00017	00	00	00	00
0001b	00	00	00	00
0001f	00	00	00	00
00023	00	00	00	00
00027	00	00	00	00
0002b	00	00	00	00
0002f	00	00	00	00
00033	00	00	00	00

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
    
```

Der Wert 1 ist jetzt auf dem Stapel

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)
    
```

i hat jetzt den Wert 1

Zustand nach halt

```

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

Endwerte:
pc = 0a
hp = 0000f
sp = 10000

H = 1: Maschine ist gestoppt

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)

Zeile	Label	Opera
18		push
19		push
20		add
21		push
22		push
23		add
24		push 4
25		div
26		mult
27		push 6

Registor	Wert
PC	00015
HP	0002d
SP	00ff4

Adres...	+0	+1	+2	+3
00000	01	00	00	00
00005	01	00	00	00
0000a	04			
0000b	01	00	00	00
00010	01	00	00	00
00015	04			
00016	01	00	00	00
0001b	07			
0001c	06			
0001d	01	00	00	00
10000				

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)

Zeile	Label	Opera
18		pus
19		pus
20		ad
21		pus
22		pus
23		ad
24		push 4
25		div
26		mult
27		push 6

Registor	Wert
PC	00016
HP	0002d
SP	00ff8

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

Zustand nach 1 2 + 3 5 + 4 /

PMI-Visualisierung

Zeile	Label	Opera...	Opera...
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

Registor	Wert
PC	0001c
HP	0002d
SP	00ff8

Adres...	+0	+1	+2	+3
00000	01	00	00	00
00005	01	00	00	00
0000a	04			
0000b	01	00	00	00
00010	01	00	00	00
00015	04			
00016	01	00	00	00
0001b	07			
0001c	06			
0001d	01	00	00	00
10000				

Spezifikationen der Vergleichs-Operationen

❖ 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 ()

Was macht dieses PMI-Programm?

```

push 10
pop i // i = 10
loop: push @i // @i: direkte Adressierung
      push 1 // operand = memory.getVal(operandCode);
      sub // Welchen Wert würde push i auf den Stapel legen?
      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 = 10; while (i > 0) { i = i -1; }
```

Spezifikation der Unterprogramm-Operationen

❖ jumpSubroutine (Adresse):

Legt die Adresse des nächsten Befehls als oberstes Element auf den Stapel und springt dann zur Argument-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)
```

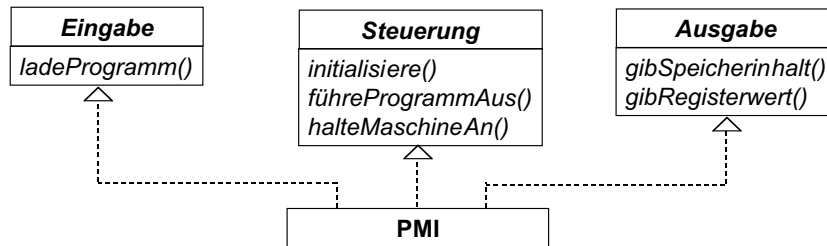
❖ Unterprogrammaufruf

⇒ nächster Vorlesungsabschnitt

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

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 teilweise nachgebildet:



- ❖ **Hinweis:** Diese Einschränkungen gelten nicht für die MI (\Rightarrow Info III)

Implementation der PMI: Modell-Sicht-Steuerung

- ❖ PMI ist als interaktives System implementiert. Die Modellierung und Implementierung der PMI folgt dem **Modell-Sicht-Steuerung**-Konzept:
 - Alle Anwendungs-spezifischen Daten und Berechnungen sind Teil des **Modells (Model)**.
 - Dem Benutzer wird eine **Sicht (View)** auf das Modell präsentiert, in der Daten und Berechnungsergebnisse aufbereitet werden.
 - Eingaben durch den Benutzer werden über die **Steuerung (Controller)** 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).

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



- ❖ **Bemerkung:** Interaktive Anwendungen kombinieren meistens die Komponenten für Datenausgabe (die Sicht) und Dateneingabe (incl. Steuerung) in einer gemeinsamen Bedienoberfläche. Daher sind Sicht und Steuerung häufig eng gekoppelt.
 - Ein Indiz für diese enge Kopplung bei der Implementation der PMI-Visualisierung 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**).

Verfügbarkeit von PMI

- ❖ Den gesamten Java-Quellcode von PMI finden Sie unter <http://www.bruegge.in.tum.de/teaching/ss01/Info2/pmi>
- ❖ **Bemerkung:** Die Implementierung von PMI benutzt einige Java-Konstrukte und Konzepte, die wir in der Vorlesung nicht erklärt haben:
 - Swing-Klassen (graphische Bedienoberfläche)
 - Threads (parallele Abläufe)
 - Synchronisierung (Steuerung paralleler Abläufe \Rightarrow Info 3)

Zusammenfassung

- ❖ Eine ***von-Neumann-Maschine*** (wie z.B. die PMI) besteht aus Eingabewerk, Steuerwerk, Rechenwerk, Arbeitsspeicher 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***.