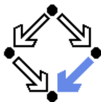
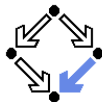
Formale Grundlagen der Informatik 2
Random Access Machines

Wolfgang Schreiner
Wolfgang.Schreiner@risc.uni-linz.ac.at

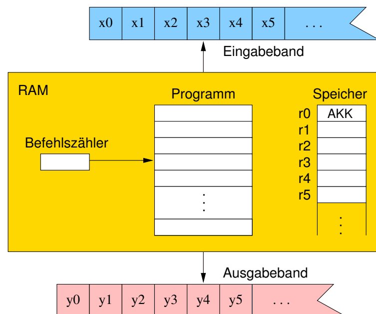
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
http://www.risc.uni-linz.ac.at



Random Access Machine (RAM)

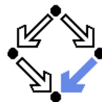


- Eingabeband und Ausgabeband.
    - Unendliche Folge von Feldern.
    - Jedes Feld kann eine ganze Zahl enthalten.
- Speicher.
    - Unendliche Folge von Registern.
    - Jedes Register kann eine ganze Zahl enthalten.
    - Register 0 dient als Akkumulator.
- Programm.
    - Endliche Folge von Instruktionen.
    - Der Befehlszähler verweist auf die aktuelle Instruktion.



Modell eines Computers.

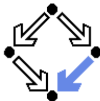
Ausführungsmodell der RAM



- Zu Beginn:
  - Alle Register enthalten 0.
  - Der Befehlszähler zeigt auf die erste Instruktion.
  - Alle Ausgabefelder sind leer.
  - Nur endliche viele Eingabefelder sind nicht leer.
  - Lese- und Schreibkopf zeigen auf das erste Feld des jeweiligen Bandes.
- Jeder Schritt:
  - Es wird die Instruktion ausgeführt, auf die der Befehlszähler verweist.
  - Danach wird der Befehlszähler um eins erhöht.
    - Verweis auf die nächste Instruktion in Folge.
    - Ausnahme: Instruktionen `JUMP`, `JGTZ`, `JZERO`, `HALT` (siehe später).
- Ausführung der Instruktion `HALT`:
  - Die Berechnung endet.

Modell der Arbeit eines Computers.

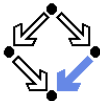
Instruktionen der RAM



- Aufbau einer Instruktion:
  - Eine optionale Marke.
  - Ein Operationscode.
  - Eine Adresse (Operand oder Marke).
- Mögliche Formen von Operanden:
  - `=i` . . . die ganze Zahl *i*.
    - *Unmittelbare* Adressierung.
  - `i` . . . der Inhalt des Registers mit der Nummer *i*.
    - *Direkte* Adressierung.
  - `*i` . . . der Inhalt desjenigen Registers, dessen Nummer im Register mit der Nummer *i* steht.
    - *Indirekte* Adressierung.

Eine einfache Maschinensprache.

Instruktionen der RAM



- 12 Instruktionen:

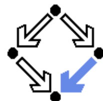
  Operationscodes LOAD, STORE, ADD, SUB, MULT, DIV, READ, WRITE, JUMP, JGTZ, JZERO, HALT.

- Speicherfunktion $c : \mathbb{N} \to \mathbb{Z}$
  - $c(i)$ . . . der Inhalt des Registers r_i.
- $v : \mathit{Operand} \to \mathbb{Z}$
  - $v(a)$ . . . der Wert des Operanden a.

$$v(\texttt{=}i) = i$$
$$v(i) = c(i)$$
$$v(\texttt{*}i) = c(c(i))$$

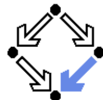
Die Bedeutung der Instruktionen wird mit Hilfe der Speicherfunktion und des Werts der Operanden definiert.

Instruktionen der RAM



1. `LOAD a`: $c(0) \leftarrow v(a)$
2. `STORE i`: $c(i) \leftarrow c(0)$
   `STORE *i`: $c(c(i)) \leftarrow c(0)$
3. `ADD a`: $c(0) \leftarrow c(0) + v(a)$
4. `SUB a`: $c(0) \leftarrow c(0) - v(a)$
5. `MULT a`: $c(0) \leftarrow c(0) * v(a)$
6. `DIV a`: $c(0) \leftarrow \lfloor c(0)/v(a) \rfloor$
7. `READ i`: $c(i) \leftarrow$ *Eingabefeld* (Lesekopf rückt um eine Position nach rechts)
   `READ *i`: $c(c(i)) \leftarrow$ *Eingabefeld* (Lesekopf rückt um eine Position nach rechts)
8. `WRITE a`: *Ausgabefeld* $\leftarrow v(a)$ (Schreibkopf rückt um eine Position nach rechts)
9. `JUMP b`: Befählszähler wird auf die Nummer der Instruktion mit Marke b gesetzt.
10. `JGTZ b`: wenn $c(0) > 0$, wird der Befählszähler auf die Nummer der Instruktion mit der Marke b gesetzt; ansonsten wird der Befehlszähler um 1 erhöht.
11. `JZERO b`: wenn $c(0) = 0$, wird der Befählszähler auf die Nummer der Instruktion mit der Marke b gesetzt; ansonsten wird der Befehlszähler um 1 erhöht.
12. `HALT`: die Berechnung endet.

Die Bedeutung der RAM

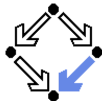


Die Bedeutung einer RAM R ist durch ihre (partielle) Abbildung von Eingabebändern auf Ausgabebänder definiert.

- Interpretation der Abbildung als Funktion:
  - R lese immer n Zahlen $x_1, \ldots, x_n$ vom Eingabeband und schreibe entweder eine Zahl y auf das Ausgabeband oder terminiere nicht.
  - R berechnet damit die folgende Funktion $f : \mathbb{Z}^n \to \mathbb{Z}$:

    $$f(x_1, \ldots, x_n) = \begin{cases} y, & \text{wenn } R \text{ auf } x_1, \ldots, x_n \text{ terminiert und } y \text{ schreibt} \\ \bot, & \text{wenn } R \text{ auf } x_1, \ldots, x_n \text{ nicht terminiert} \end{cases}$$
- Interpretation der Abbildung als Sprache:
  - Für ein gegebenes Alphabet $\Sigma = \{\alpha_1, \ldots, \alpha_k\}$ kann jedes Symbol α_i als die ganze Zahl i codiert werden.
  - Ein Wort $\alpha_{i_1}, \ldots, \alpha_{i_n}$ kann damit als die Folge $i_1, \ldots, i_n, 0$ auf das Eingabeband einer R geschrieben werden.
  - Das Wort wird von R akzeptiert, wenn R die Folge vom Eingabeband liest, eine 1 in das Ausgabeband schreibt und terminiert.
  - Die Sprache von R ist die Menge der von R akzeptierten Wörter.

Beispiel

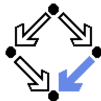


RAM-Programm für die Funktion $f(n) = \begin{cases} n^n, & \text{wenn } n > 0 \\ 0, & \text{sonst} \end{cases}$

read r_1
if $r_1 \leq 0$ **then**
  write 0
else
  $r_2 \leftarrow r_1$
  $r_3 \leftarrow r_1 - 1$
  while $r_3 > 0$ **do**
    $r_2 \leftarrow r_2 \cdot r_1$
    $r_3 \leftarrow r_3 - 1$
  write r_2

	`READ`	`1`	Eingabe von $c(1)$
	`LOAD`	`1`	$c(0) \leftarrow c(1)$
	`JGTZ`	`else`	wenn $c(0) > 0$, `else`
	`WRITE`	`=0`	Ausgabe von 0
	`JUMP`	`halt`	
`else:`	`LOAD`	`1`	$c(0) \leftarrow c(1)$
	`STORE`	`2`	$c(2) \leftarrow c(0)$
	`LOAD`	`1`	$c(0) \leftarrow c(1)$
	`SUB`	`=1`	$c(0) \leftarrow c(0) - 1$
	`STORE`	`3`	$c(3) \leftarrow c(0)$
	...		

Beispiel

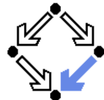


read r_1
if $r_1 \leq 0$ **then**
  write 0
else
  $r_2 \leftarrow r_1$
  $r_3 \leftarrow r_1 - 1$
  while $r_3 > 0$ **do**
    $r_2 \leftarrow r_2 \cdot r_1$
    $r_3 \leftarrow r_3 - 1$
  write r_2

```
        ...
while:  LOAD   3      c(0) ← c(3)
        JGTZ   body   wenn c(0) > 0, body
        JUMP   done
body:   LOAD   2      c(0) ← c(2)
        MULT   1      c(0) ← c(0) · c(1)
        STORE  2      c(2) ← c(0)
        LOAD   3      c(0) ← c(3)
        SUB    =1     c(0) ← c(0) − 1
        STORE  3      c(3) ← c(0)
        JUMP   while
done:   WRITE  2      Ausgabe von c(2)
halt:   HALT
```

Ähnlich einem echten Maschinenprogramm.

Beispiel

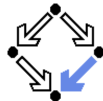


RAM-Programm, das alle Wörter aus 1en und 2en akzeptiert, die aus gleich vielen 1en wie 2en bestehen.

$r_2 \leftarrow 0$
read r_1
while $r_1 \neq 0$ **do**
  if $r_1 \neq 1$
    then $r_2 \leftarrow r_2 - 1$
    else $r_2 \leftarrow r_2 + 1$
  read r_1
if $r_2 = 0$ **then write** 1

```
        LOAD   =0    c(0) = 0
        STORE  2     c(2) = c(0)
        READ   1     Eingabe von r1
while:  LOAD   1     c(0) ← c(1)
        JZERO  done  wenn c(0) = 0, done
        LOAD   1     c(0) ← c(1)
        SUB    =1    c(0) ← c(0) − 1
        JZERO  one   wenn c(0) = 0, one
        LOAD   2     c(0) ← c(2)
        SUB    =1    c(0) ← c(0) − 1
        STORE  2     c(2) = c(0)
        JUMP   read
one:    LOAD   2     c(0) ← c(2)
        ADD    =1    c(0) ← c(0) + 1
        STORE  2     c(2) = c(0)
        ...
```

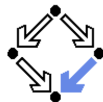
Beispiel



$r_2 \leftarrow 0$
read r_1
while $r_1 \neq 0$ **do**
  if $r_1 \neq 1$
    then $r_2 \leftarrow r_2 - 1$
    else $r_2 \leftarrow r_2 + 1$
  read r_1
if $r_2 = 0$ **then write** 1

```
          ...
read:     READ    1       Eingabe von c(1)
          JUMP    while
done:     LOAD    2       c(0) ← c(2)
          JZERO   write   wenn c(0) = 0, write
          HALT
write:    WRITE   =1      Ausgabe von 1
          HALT
```

Random Access Stored Program Machine



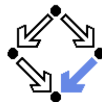
Programm selbst ist im Speicher in Registern abgelegt.

- Random Access Stored Program Machine (RASP)
  - Instruktionen wie RAM.
    - Ausnahme: keine indirekte Adressierung (werden durch Modifikation der Instruktionen während der Programmausführung ersetzt).
  - Jede Instruktion ist in zwei aufeinanderfolgenden Registern codiert:
    - Register i: der Operationscode als ganze Zahl.
    - Register $i+1$: der Operand.

      `LOAD 3`: (1,3), `LOAD =3` (2,3), ...
  - Die Programmausführung ähnlich ist wie bei der RAM.
    - Das Programm ist in einer Folge von Registern abgelegt.
    - Der Befehlszähler verweist auf das Register, das den Operationscode der ersten Instruktion enthält.
    - Nach Ausführung jeder Instruktion wird Befehlszähler um 2 erhöht.

      Ausnahme: Instruktionen `JUMP`, `JGTZ`, `JZERO`, `HALT`.

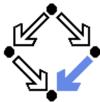
Vergleich von RAM und RASP



- Ist das Konzept der RAM mächtiger?
  - Ist indirekte Adressierung mächtiger als Selbstmodifikation?
- Ist das Konzept der RASP mächtiger?
  - Bieten selbstmodifizierende Programme zusätzliche Möglichkeiten?
- Satz: zu jedem RAM-Programm gibt es ein äquivalentes RASP-Programm und umgekehrt.
  - Beide Konzepte sind gleich mächtig.

RAM-Programme haben die gleiche Mächtigkeit wie RASP-Programme, sind aber in gewissem Sinne "sicherer".

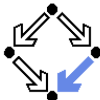
Simulation von RAM durch RASP



Wir konstruieren ein zu einem gegebenen RAM-Programm P äquivalentes RASP-Programm P'.

- P' benötigt r zusätzliche Register:
  - Register 1 für Zwischenspeicherung des Akkumulators von P.
  - Register $2, \ldots, r$ für das Programm selbst.
  - Register $r+1, r+2, \ldots$ simulieren die Register $1, 2, \ldots$ von P.
- Übersetzung einer Instruktion von P ohne indirekte Adressierung:
  - Identische Instruktion in P' (mit Registerzugriffen um r Positionen verschoben).
- Übersetzung einer Instruktion von P mit indirekter Adressierung:
  - Folge von sechs RASP-Instruktionen.
  - Indirekte Adressierung wird durch Instruktionsmodifikation simuliert.
- P mit n Instruktionen, davon m mit indirekter Adressierung.
  - P' hat $(n-m)+6m = n+5m$ Instruktionen.
  - Verschiebungsfaktor $r = 2(n+5m)+1 = 2n+10m+1$.

Simulation von RAM durch RASP

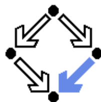


Übersetzung der RAM-Instruktion SUB $*i$ ($i \geq 1$).

Adresse	Opcode	Operand	Beschreibung
	STORE	1	$c(1) \leftarrow c(0)$
	LOAD	r+i	$c(0) \leftarrow c(r+i)$
	ADD	r	$c(0) \leftarrow c(0) + r$
	STORE	a+1	$c(a+1) \leftarrow c(0)$
	LOAD	1	$c(0) \leftarrow c(1)$
a:	SUB	...	$c(0) \leftarrow c(0) - \ldots$

- Inhalt des Akkumulators wird gesichert.
- Im Akkumulator wird $c(r+i)+r$ berechnet.
- Ergebnis wird in den Operandenteil der SUB-Instruktion geschrieben.
- Der Inhalt des Akkumulator wird wieder hergestellt.
- Die modifizierte SUB-Instruktion wird ausgeführt.

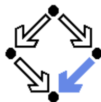
Simulation von RASP durch RAM



Wir konstruieren ein zu einem gegebenen RASP-Programm P äquivalentes RAM-Programm P'.

- Das RAM-Programm ist ein Interpreter für die Sprache der RASP.
  - Liest jede Instruktion des RASP-Programms.
  - Bestimmt den numerischen Operationscode.
  - Verzweigt in eine Instruktionsfolge, die eine RASP-Instruktion mit diesem Operationscode interpretiert.
- Register von P':
  - r_1 ... für indirekte Adressierung (am Anfang 0).
  - r_2 ... der Befehlszähler der RASP (am Anfang 4).
  - r_3 ... der Akkumulator der RASP (am Anfang 0).
  - $(r_4, r_5, \ldots)$ ... das RASP-Programm.

Simulation von RASP durch RAM

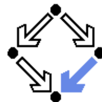


Interpretation der RASP Instruktion `SUB *i`.

`LOAD`	`2`	$c(2) \leftarrow c(2) + 1$
`ADD`	`=1`	Befehlszähler verweist auf Operand i
`STORE`	`2`	
`LOAD`	`*2`	$c(1) \leftarrow c(c(2)) + 3$
`ADD`	`=3`	Inhalt des RASP-Registers i
`STORE`	`1`	
`LOAD`	`3`	$c(3) \leftarrow c(3) - c(c(1))$
`SUB`	`*1`	RASP-Akkumulator um Inhalt des RASP-Registers i vermindert
`STORE`	`3`	
`LOAD`	`2`	$c(2) \leftarrow c(2) + 1$
`ADD`	`=1`	Befehlszähler verweist auf nächste Instruktion
`STORE`	`2`	
`JUMP`	`loop`	Interpretiere die nächste Instruktion

Modell eines *Mikroprogramms*, wie es viele Prozessoren exekutieren.

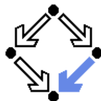
RAM und Endliche Automaten



- Jeder endliche Automat kann durch eine RAM simuliert werden.
  - Es ist nicht schwer, für jeden gegebenen Automaten eine äquivalente RAM zu konstruieren.
  - Das Konzept der RAM ist also mindestens so mächtig wie das der endlichen Automaten.
  - Aber ist es auch mächtiger?
- Sei $L = \{a^m b^m \mid m \in \mathbb{N}\}$.
  - Wir haben gezeigt, dass L keine reguläre Sprache ist.
    - L wird also von keinem endlichen Automaten akzeptiert.
  - Es ist leicht, eine RAM zu konstruieren, die L akzeptiert:
    - Lies alle a (bis zum ersten Symbol, das kein a ist). Bestimme dabei ihre Anzahl n.
    - Lies alle b (bis zum ersten Symbol, das kein b ist). Bestimme dabei ihre Anzahl m.
    - Gib 1 aus, wenn $n = m$ und die Folge zu Ende ist.

Das Konzept der RAM ist mächtiger als das der endlichen Automaten.

RAM und Computer



Was macht die Überlegenheit der RAM aus?

- Die Möglichkeit zu unendlich vielen Zuständen.
  - Register können beliebig große ganze Zahlen speichern.
  - Unendlich viele Register.
- Genau genommen ist ein Computer nur ein endlicher Automat!
  - Fixe Anzahl von Speicherzellen, die nur ganze Zahlen bis zu einem maximalen Absolutbetrag speichern können.
    - Endliche Anzahl von Zuständen.
- Annähernd kann Computer aber auch als RAM betrachtet werden.
  - Folgen von Speicherwörtern können beliebig große Zahlen codieren.
    - Solange der verfügbare Speicher ausreicht.
  - Virtueller Speicher kann beliebig großen Speicher simulieren.
    - Solange der Sekundärspeicher (Plattenplatz) ausreicht.

Unter der Annahme (prinzipiell) beliebig großen Speichers ist ein Computer einer RAM gleichmächtig.