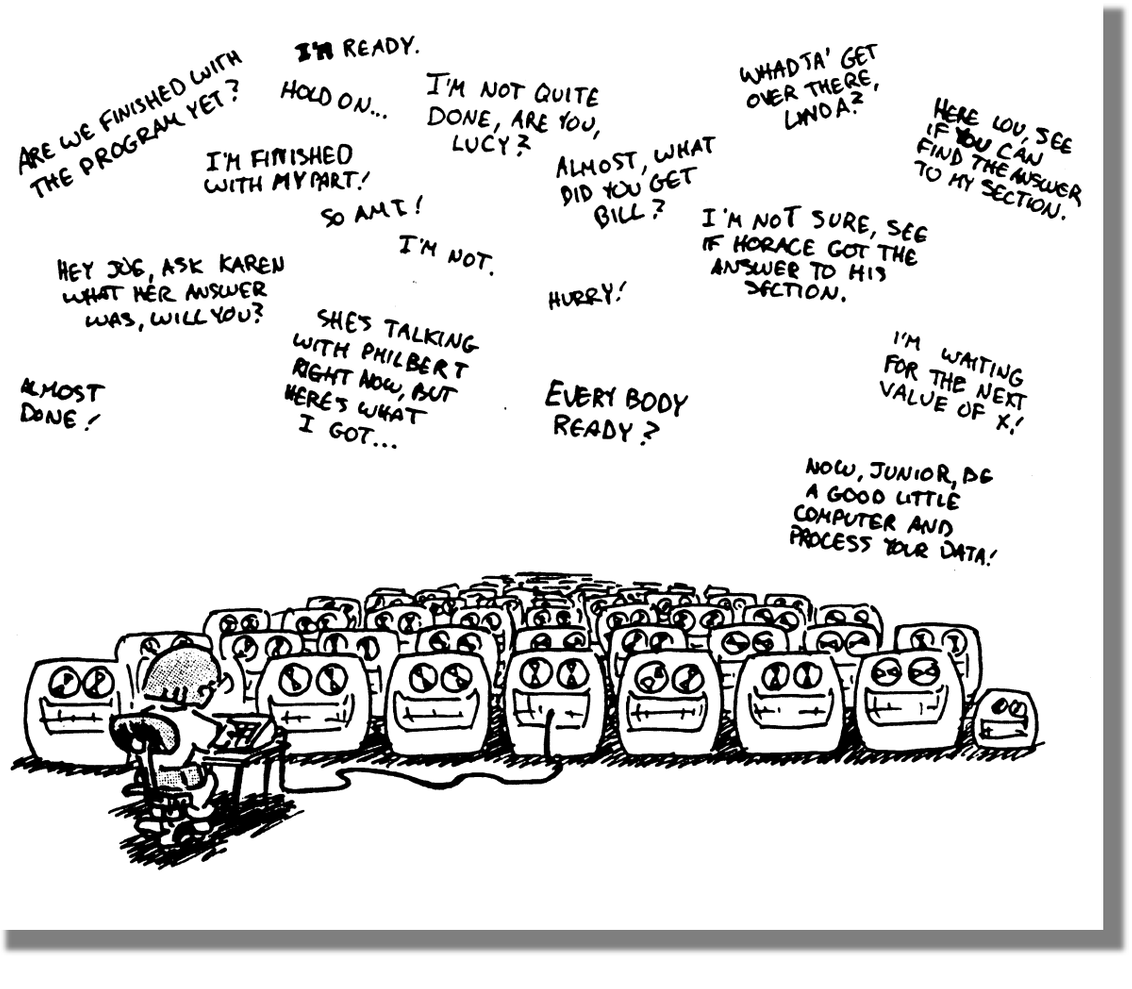
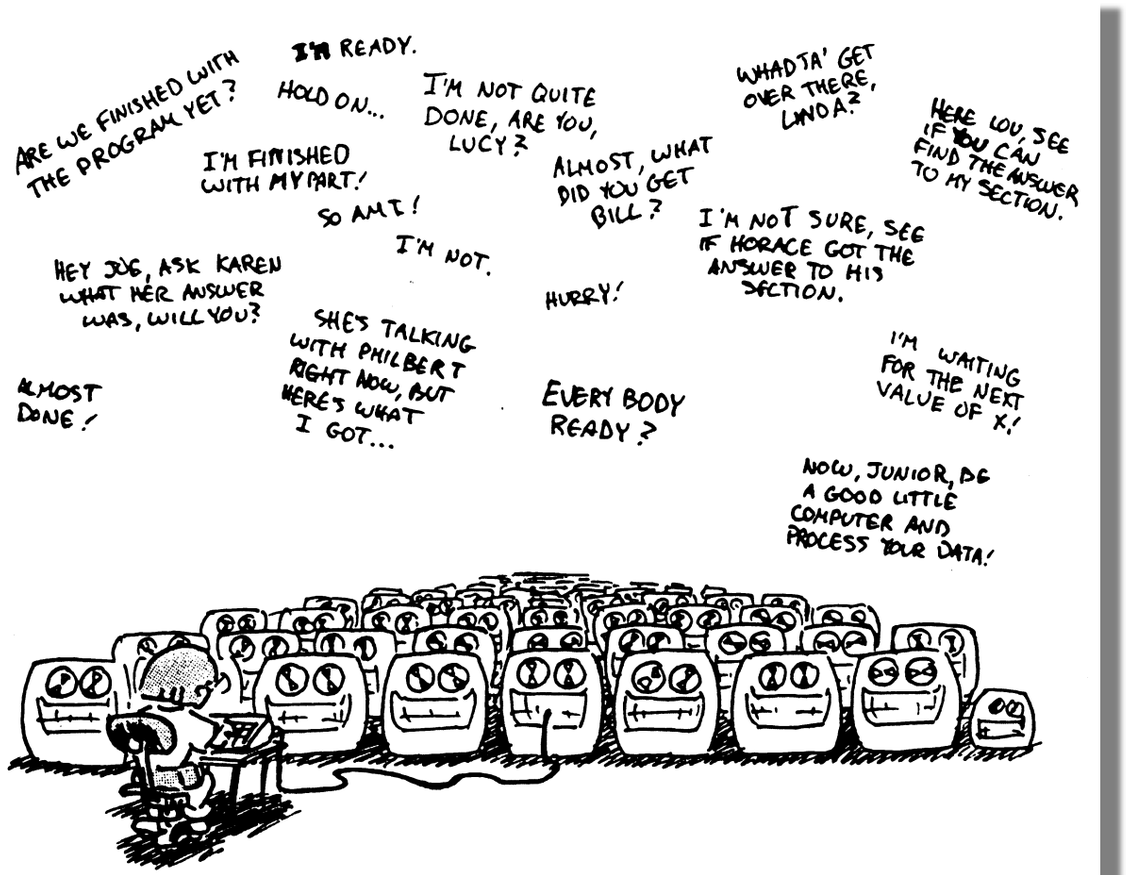
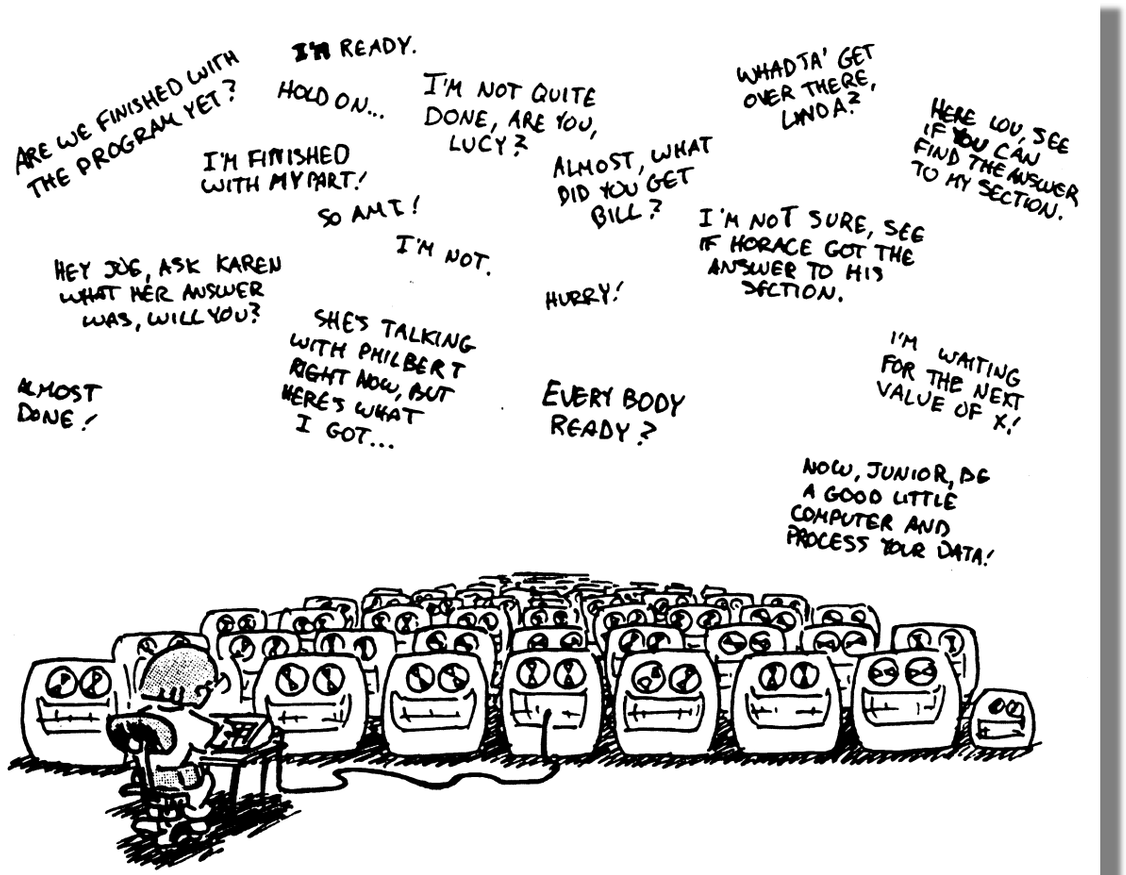
Kapitel 4

Parallele Algorithmen


ARE WE FINISHED WITH THE PROGRAM YET?
I'M READY.
HOLD ON...
I'M NOT QUITE DONE, ARE YOU, LUCY?
WHADTA' GET OVER THERE, LINDA?
HERE LOU, SEE IF YOU CAN FIND THE ANSWER TO MY SECTION.
I'M FINISHED WITH MY PART!
ALMOST, WHAT DID YOU GET BILL?
SO AM I!
I'M NOT SURE, SEE IF HORACE GOT THE ANSWER TO HIS SECTION.
I'M NOT.
HEY JOE, ASK KAREN WHAT HER ANSWER WAS, WILL YOU?
HURRY!
SHE'S TALKING WITH PHILBERT RIGHT NOW, BUT HERE'S WHAT I GOT...
I'M WAITING FOR THE NEXT VALUE OF X!
ALMOST DONE!
EVERY BODY READY?
NOW, JUNIOR, BE A GOOD LITTLE COMPUTER AND PROCESS YOUR DATA!



- Unter parallelen und verteilten Algorithmen wird die Erweiterung von klassischen, für Einprozessormaschinen konzipierten Algorithmen auf viele miteinander kooperierende Prozessoren verstanden.



- Parallele Algorithmen beruhen in der Regel auf Speicher- oder Rendezvous-Synchronisation und haben eine synchrone Ablaufsemantik.

Informatischen Kolloquium Hamburg

Montag, 11. Januar 2010
um 17 Uhr c.t.
Vogt-Kölln-Str. 30
Konrad-Zuse-Hörsaal
Gebäude B

Prof. Dr. Thomas Ludwig
Wissenschaftl.-techn. Geschäftsführer
Deutsches Klimarechenzentrum (DKRZ) Hamburg

Werkzeugentwicklung für Höchstleistungsrechner
==========

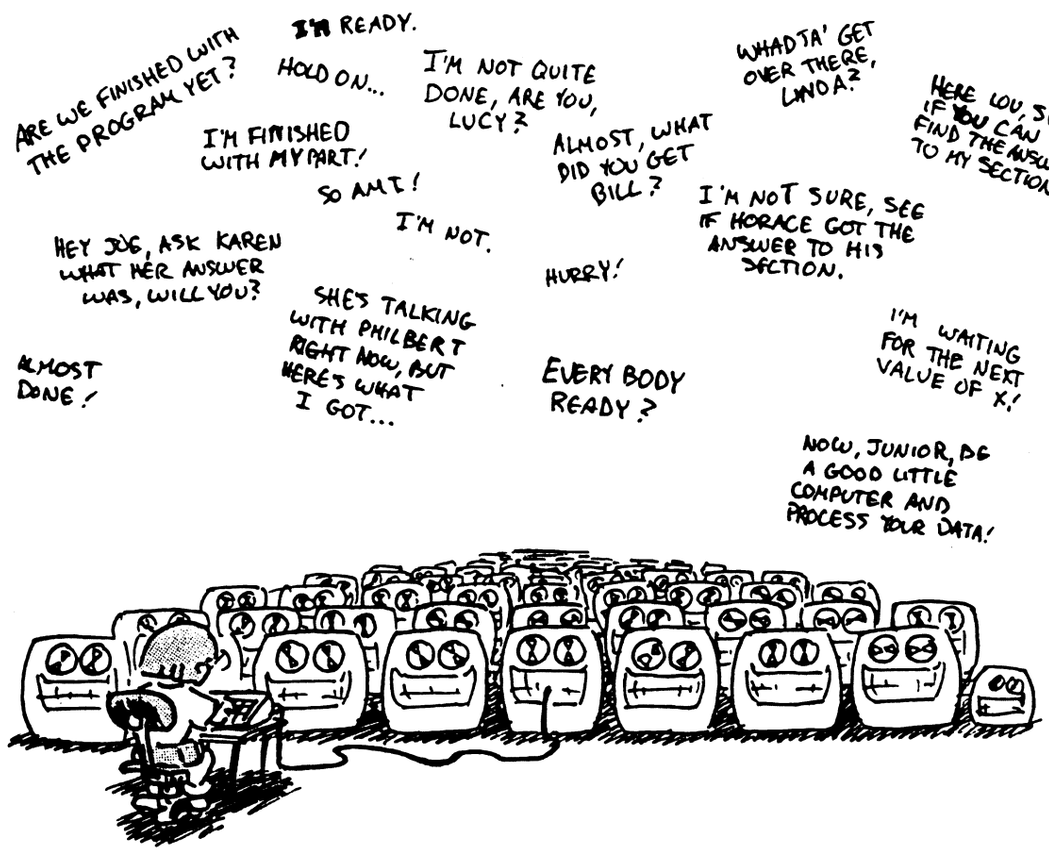
Die effiziente Nutzung von Höchstleistungsrechensystemen bietet viele interessante Herausforderungen für die Informatik. Neben den Fragestellungen zur Rechnerarchitektur und zur Programmiermethodik ist vor allem das Problem der optimalen Anpassung von parallelen Programmen an die Rechner von großer Bedeutung.

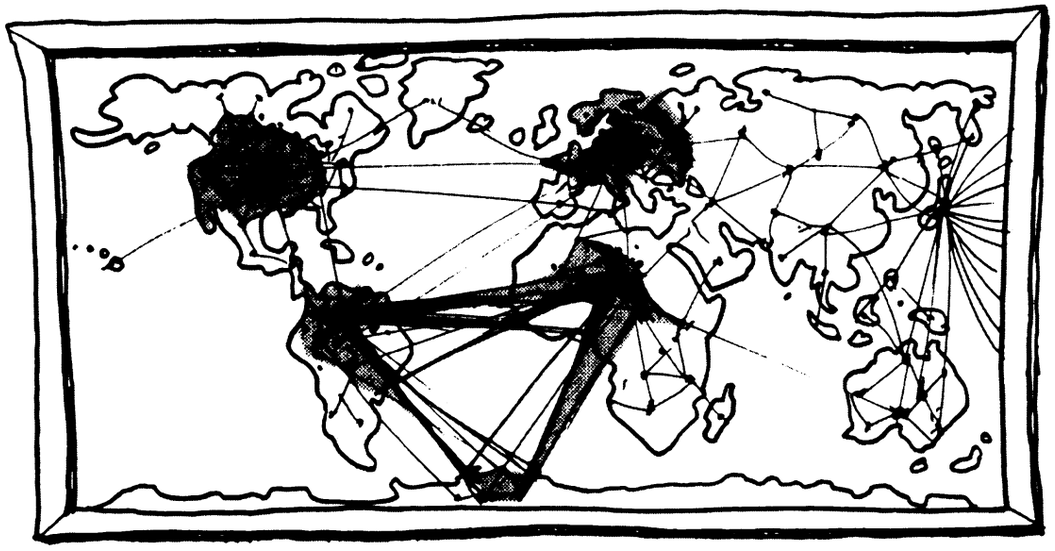
Der Anwender erhält hier durch eine Vielzahl von Werkzeugen Unterstützung. Diese werden interaktiv von ihm genutzt, wie z.B. Werkzeuge zur Fehlersuche und zur Leistungsoptimierung, oder arbeiten automatisch im Hintergrund, wie z.B. Lastausgleichssysteme. Allen diesen Werkzeugen liegt zugrunde, dass sie als Softwarekomponenten zwischen dem Betriebssystem und den parallelen Programmen zu liegen kommen. Trotz unterschiedlicher Funktionen, die sie anbieten, finden wir in ihrem Aufbau gewisse Regelmäßigkeiten. So reagieren diese Werkzeuge z.B. auf Ereignisse im Rechnersystem und zeichnen diese auf und kommunizieren mit dem Anwender und bieten ihm z.B. Möglichkeiten, den Ablauf seines Programmes zu beeinflussen.

Die neue Arbeitsgruppe "Wissenschaftliches Rechnen" von Prof. Ludwig verfügt über einen langjährigen Hintergrund in der Entwicklung solcher Werkzeuge zu verschiedenen Einsatzwecken. Ältere Konzepte für Werkzeuge zur Fehlersuche und Leistungsanalyse paralleler Programme wurden übertragen auf Werkzeuge zur Optimierung von Eingabe-/Ausgabesystemen und aktuell zur Erfassung und Verminderung des Energieverbrauchs beim Programmablauf.

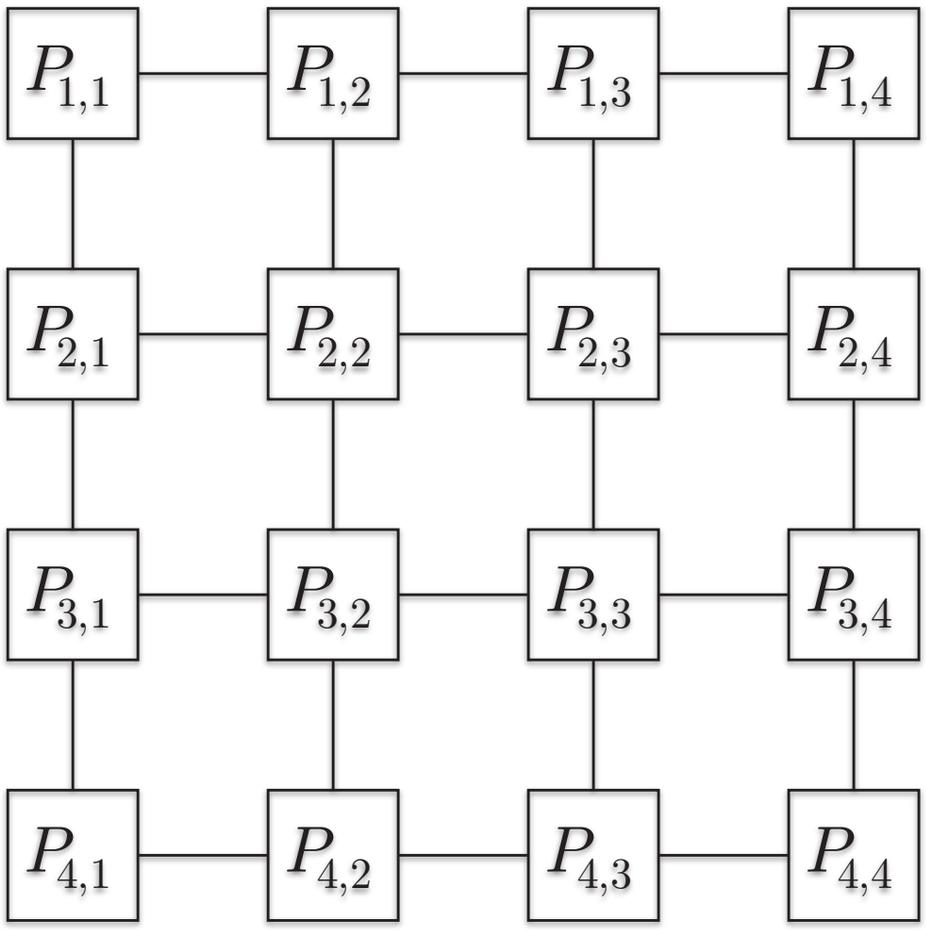
Im Vortrag werden wichtige Strukturen solcher Werkzeugumgebungen erläutert und es wird aufgezeigt, welche Bedeutung sie in der Praxis des Hochleistungsrechnens haben.

der Regel auf
hronisation und
antik.

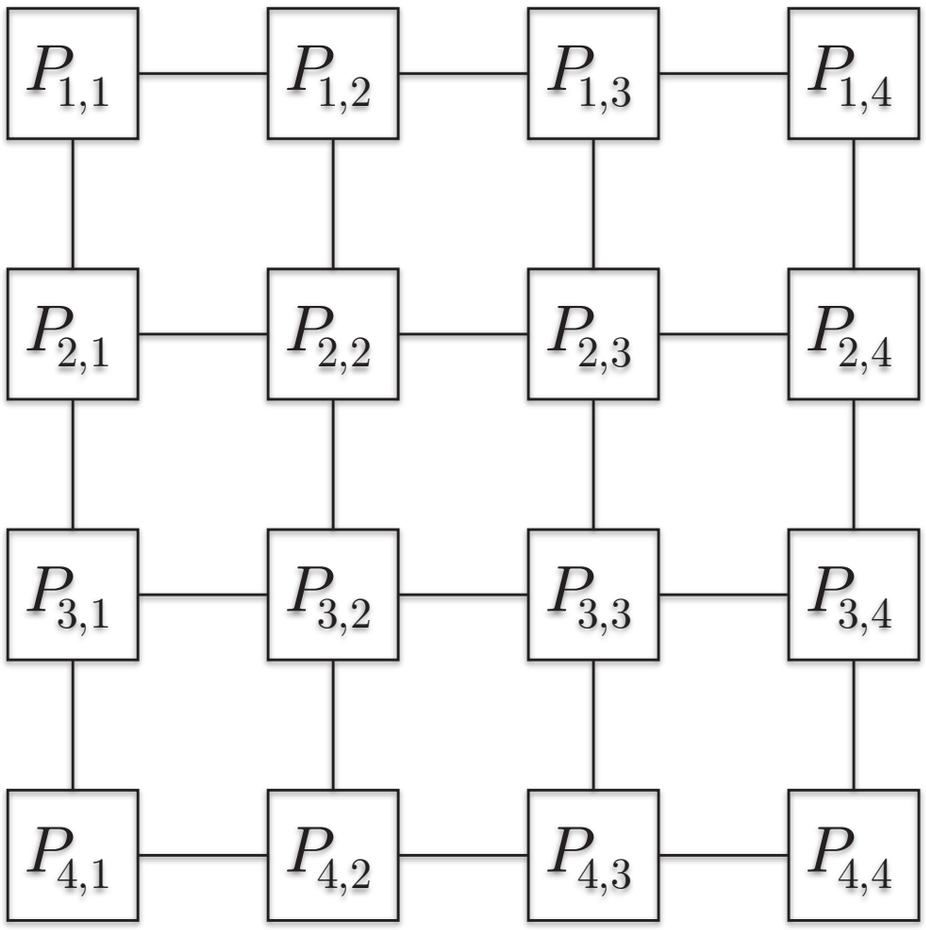




- Charakteristisch für die verteilten Algorithmen ist dagegen Nachrichten-Synchronisation.

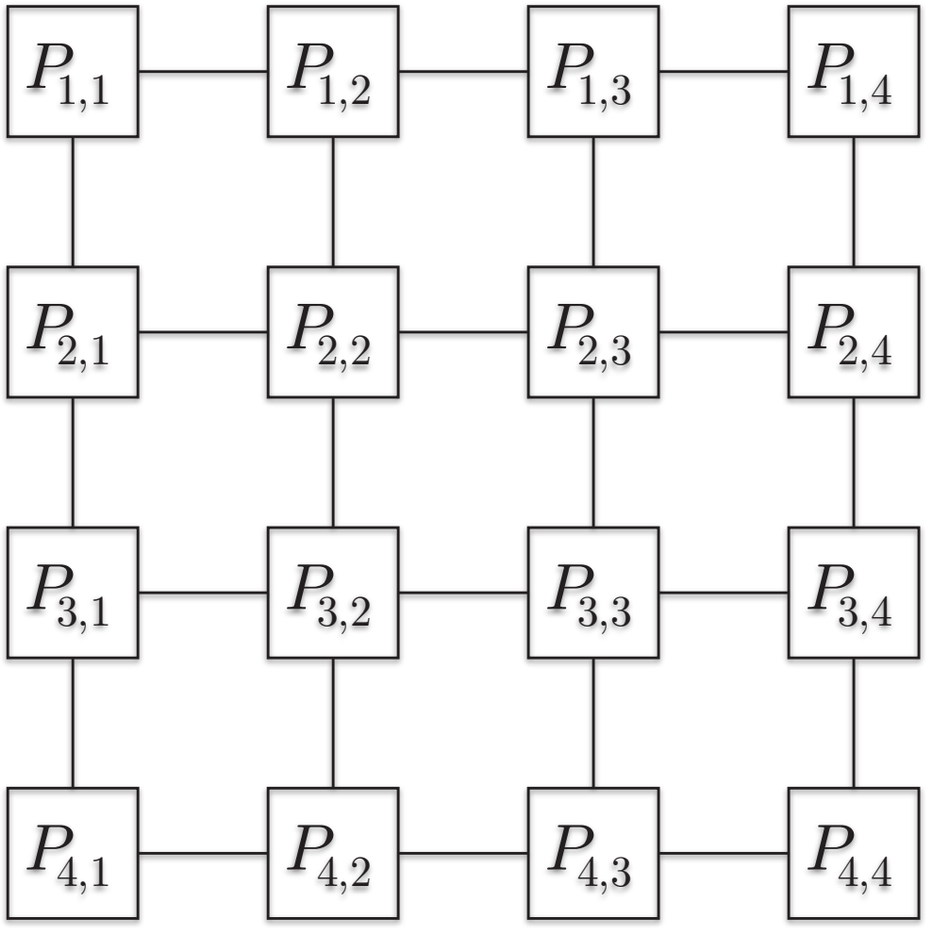


2-dimensionales Feld



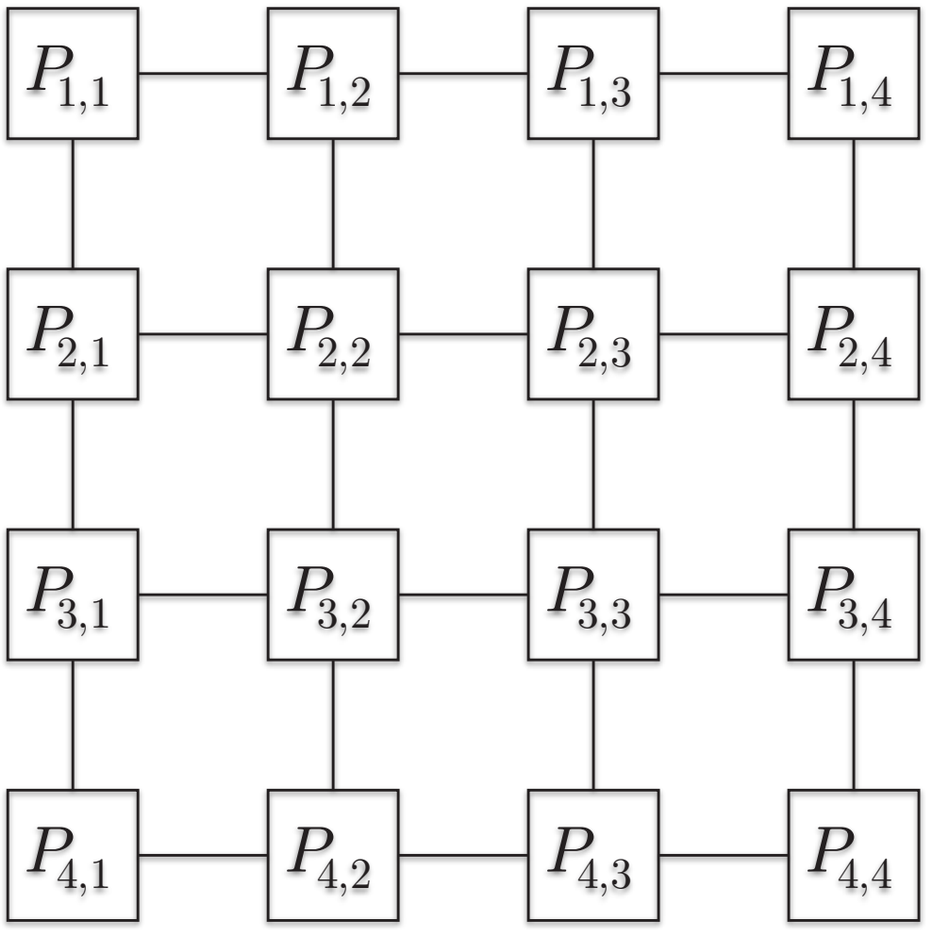
- Ring

2-dimensionales Feld



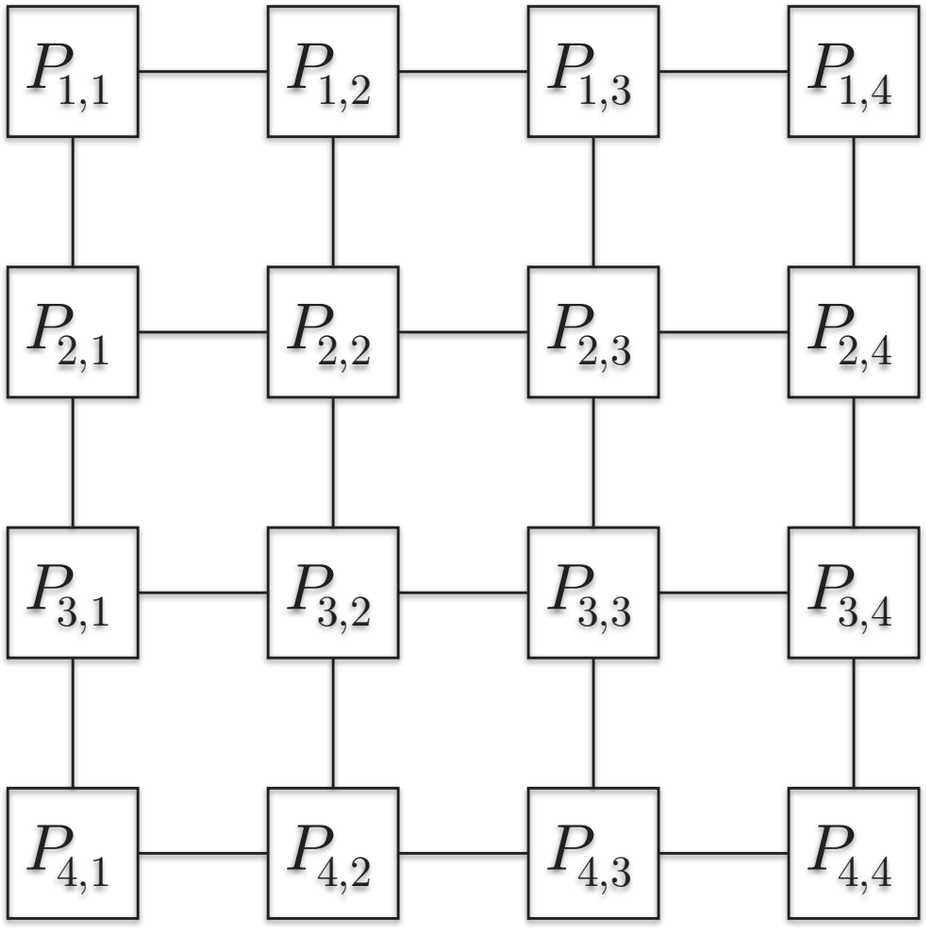
- Ring
- Baum

2-dimensionales Feld



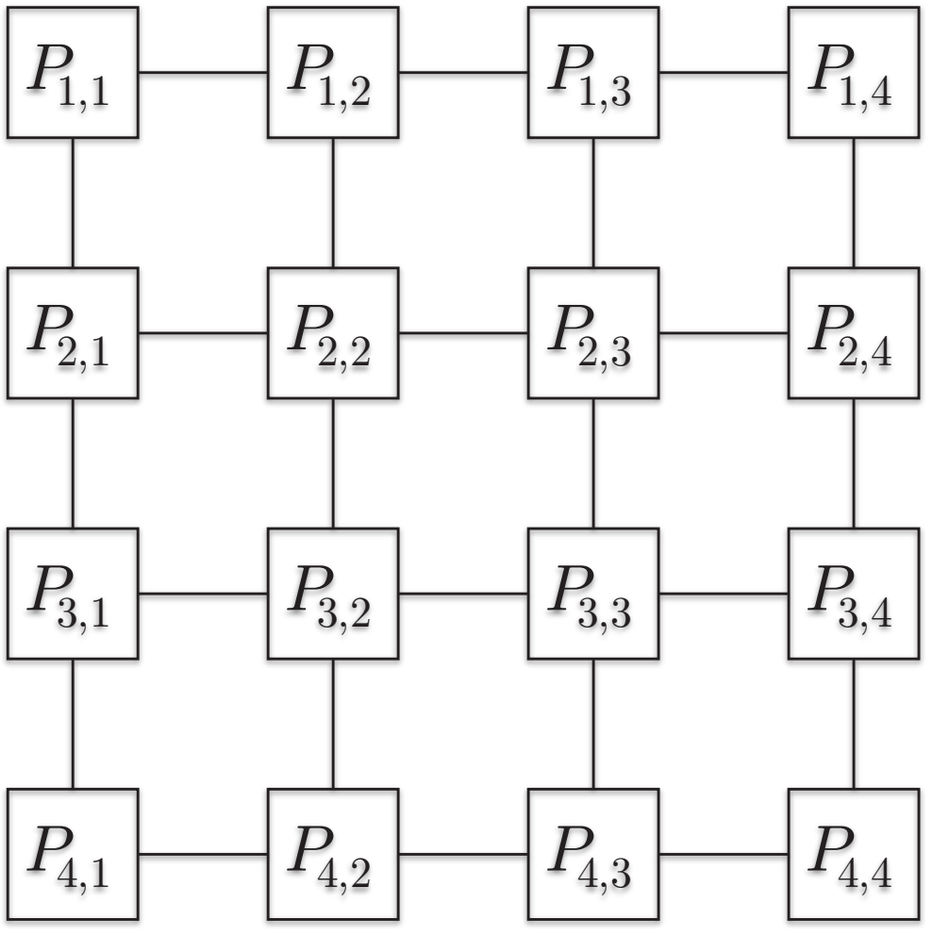
- Ring
- Baum
- Feld (array)

2-dimensionales Feld



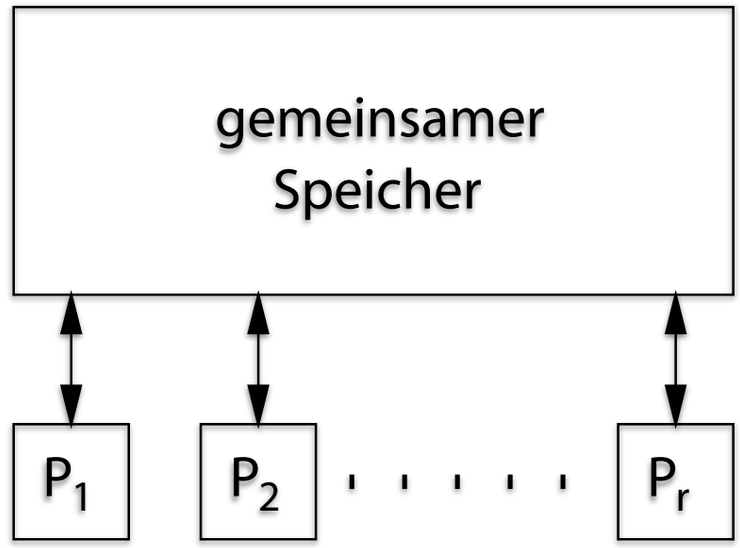
- Ring
- Baum
- Feld (array)
- ...

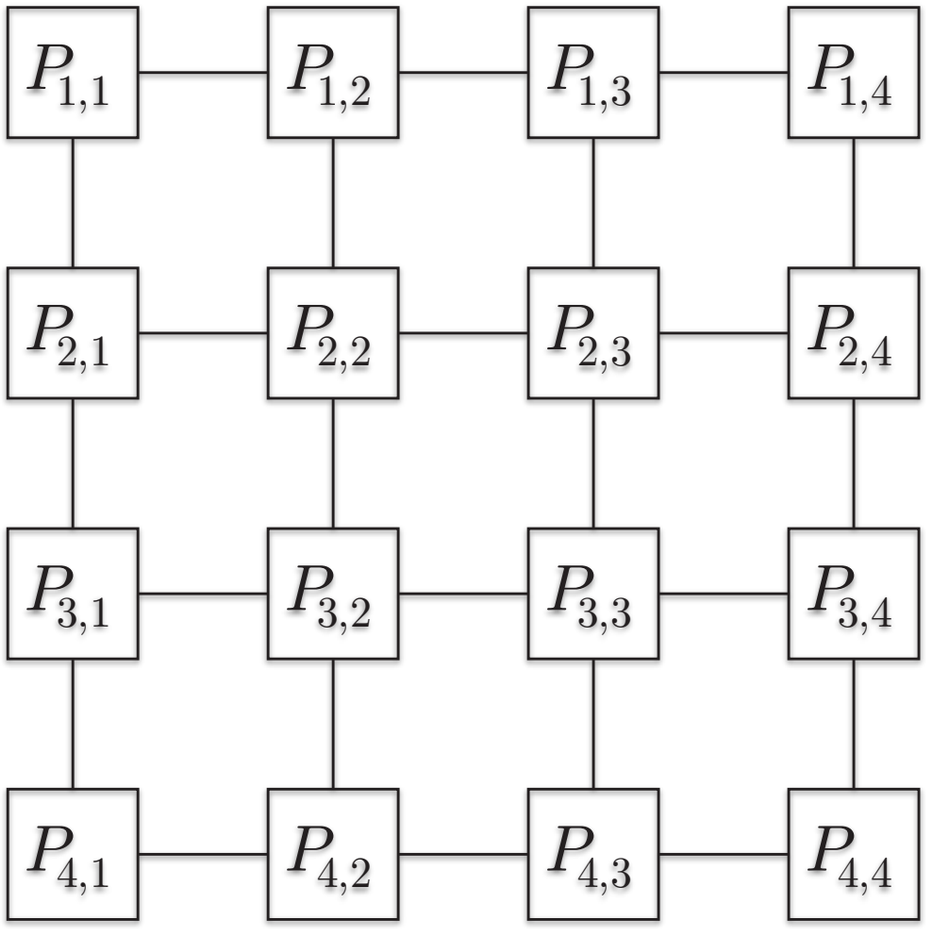
2-dimensionales Feld



2-dimensionales Feld

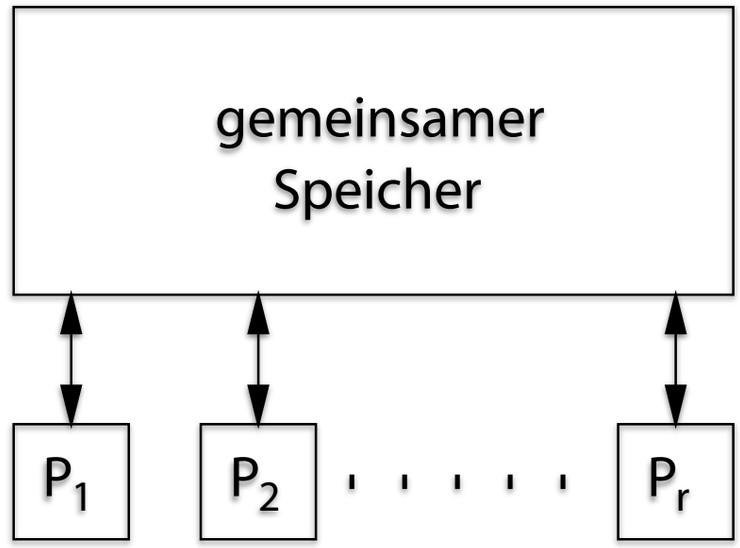
- Ring
- Baum
- Feld (array)
- ...



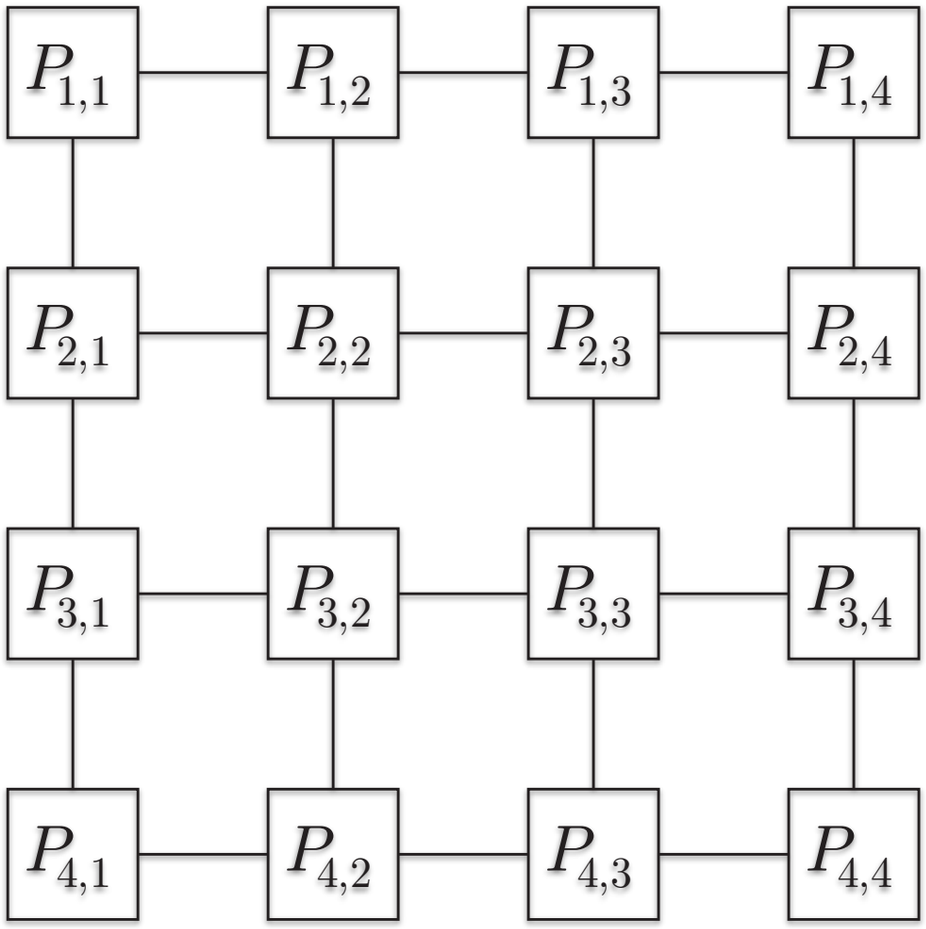


2-dimensionales Feld

- Ring
- Baum
- Feld (array)
- ...

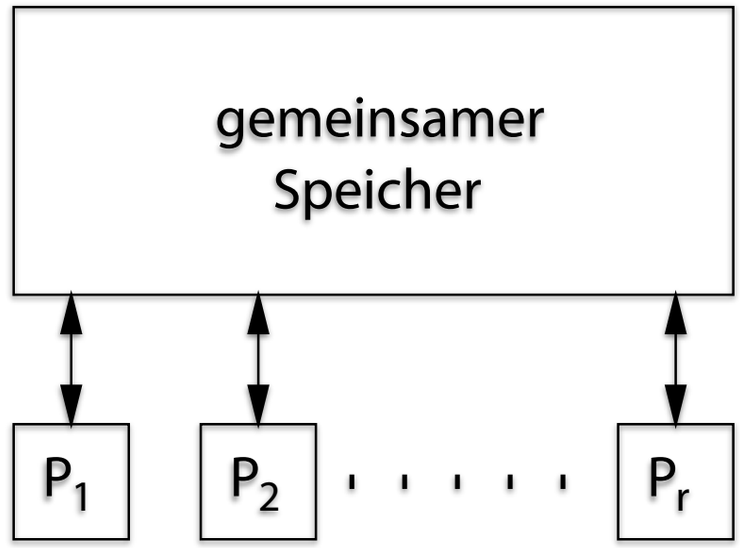


P RAM



2-dimensionales Feld

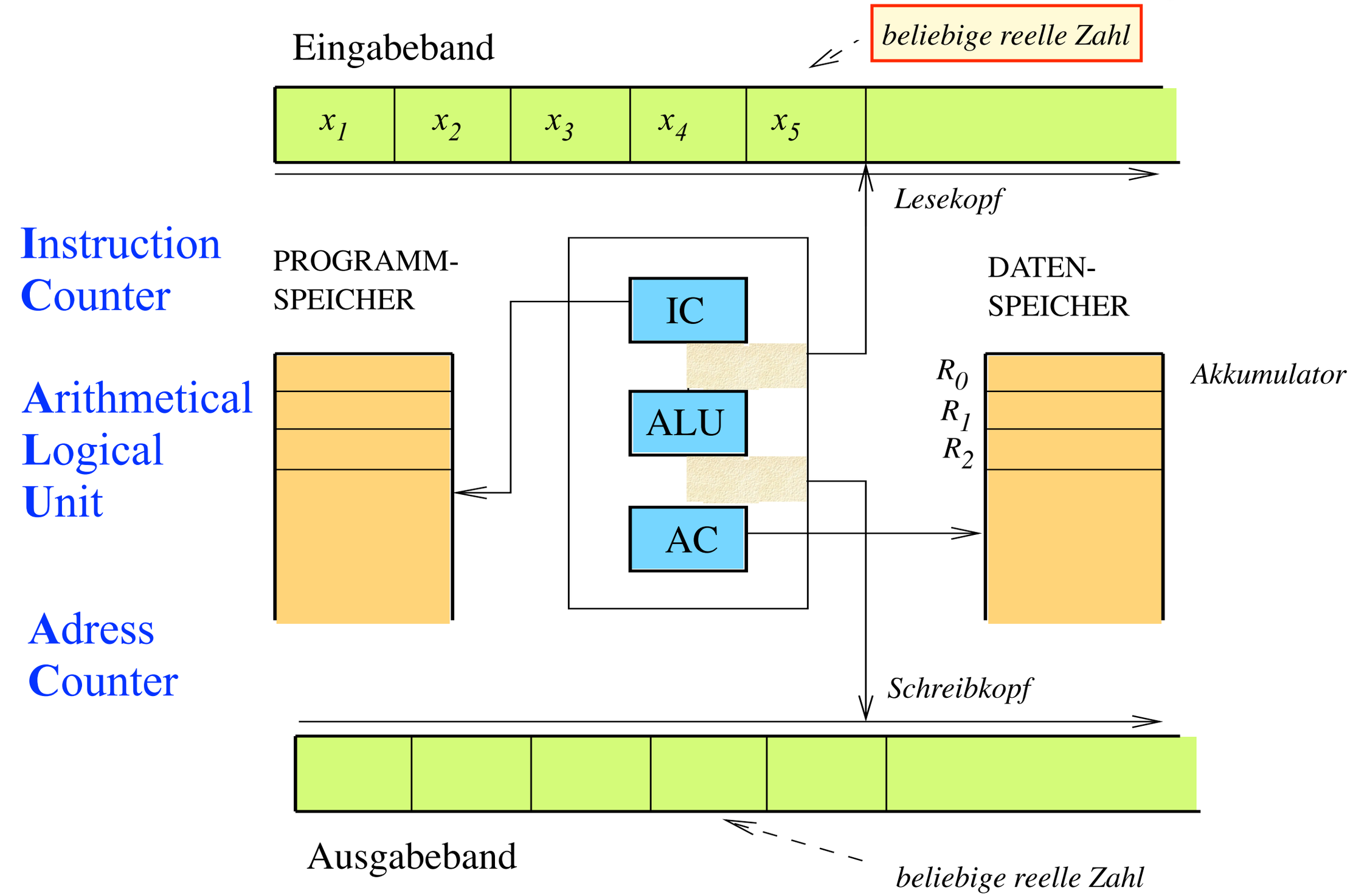
- Ring
- Baum
- Feld (array)
- ...

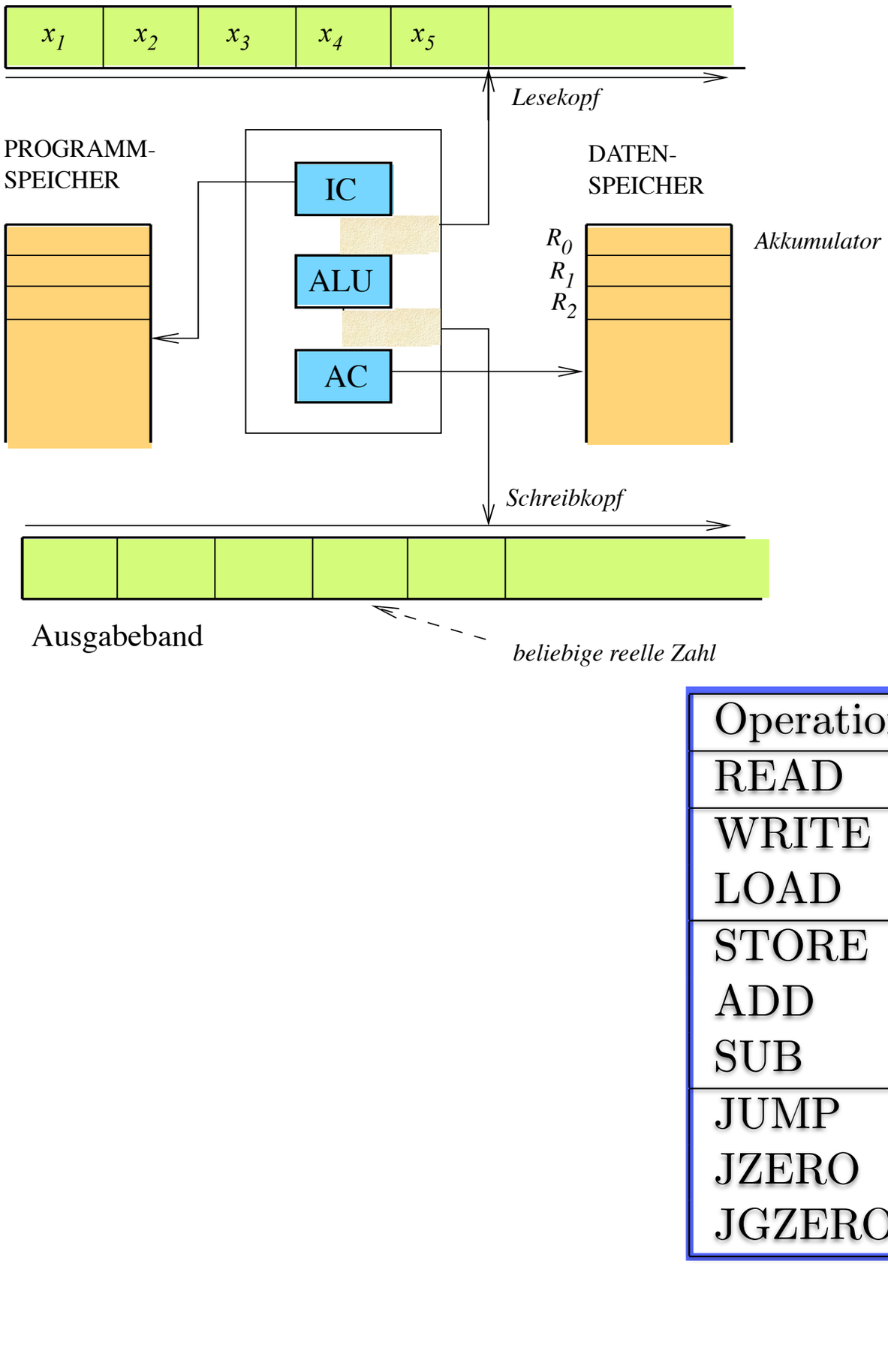


P RAM

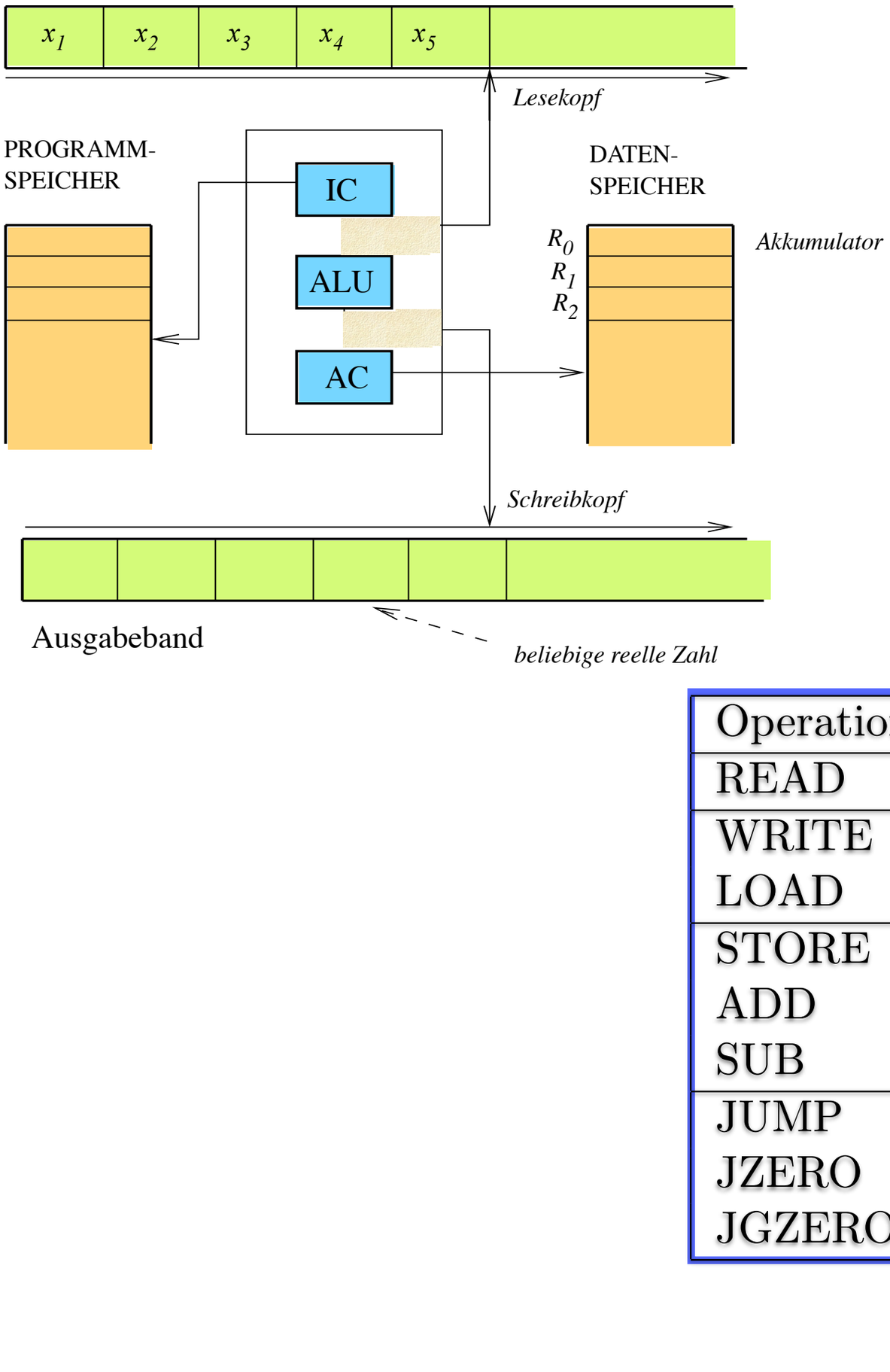
***P**arallel **R**andom-**A**ccess **M**achine*

4.1 Random-Access-Maschine (RAM)





Operationen	
READ	op
WRITE	op
LOAD	op
STORE	op
ADD	op
SUB	op
JUMP	label
JZERO	label
JGZERO	label



Operationen		Operand
READ	op	$= i$ – die Zahl i
WRITE	op	i – Inhalt des
LOAD	op	Registers R_i
STORE	op	$*i$ – indirekte Adresse
ADD	op	(Inhalt des Registers R_j, wobei j
SUB	op	der Inhalt des Registers R_i ist)
JUMP	label	
JZERO	label	label – Befehlsadresse
JGZERO	label	

Tabelle **4.1** Die Befehle einer RAM

	Instruktion	Bedeutung
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	$c(0) \leftarrow c(0) \times v(a)$
6.	DIV a	$c(0) \leftarrow \lfloor c(0) \div v(a) \rfloor$

c – Speicherabbildung

$c(i)$ Der Inhalt des Registers R_i.

$v(a)$ – Der Wert von a

$$v(=i) = i$$
$$v(i) = c(i)$$
$$v(*i) = c(c(i))$$

Tabelle **4.1** Die Befehle einer RAM

	Instruktion	Bedeutung
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	$c(0) \leftarrow c(0) \times v(a)$
6.	DIV a	$c(0) \leftarrow \lfloor c(0) \div v(a) \rfloor$
7.	READ i	$c(i) \leftarrow$ derzeitiges Eingabesymbol
	READ $*i$	$c(c(i)) \leftarrow$ derzeitiges Eingabesymbol. Der Kopf auf dem Eingabeband geht in beiden Fällen um ein Feld nach rechts.
8.	WRITE a	$v(a)$ wird in das Feld auf dem Ausgabeband geschrieben über dem sich gerade der Bandkopf befindet. Danach wird der Kopf um ein Feld nach rechts bewegt.
9.	JUMP b	Der *location counter* (Befehlsregister) wird auf die mit b markierte Anweisung gesetzt.
10.	JGTZ b	Der *location counter* wird auf die mit b markierte Anweisung gesetzt, wenn $c(0) > 0$ ist, sonst auf die nachfolgende Anweisung.
11.	JZERO b	Der *location counter* wird auf die mit b markierte Anweisung gesetzt, wenn $c(0) = 0$ ist, sonst auf die nachfolgende Anweisung.
12.	HALT	Programmausführung beenden.

c – Speicherabbildung

$c(i)$ Der Inhalt des Registers R_i.

$v(a)$ – Der Wert von a

$$v(=i) = i$$
$$v(i) = c(i)$$
$$v(*i) = c(c(i))$$

```
            READ    1          read r1
            LOAD    1
            JGTZ    pos        } if r1 ≤ 0 then write 0
            WRITE   =0
            JUMP    endif
pos:        LOAD    1          } r2 ← r1
            STORE   2
            LOAD    1
            SUB     =1         } r3 ← r1 – 1
            STORE   3
while:      LOAD    3
            JGTZ    continue   } while r3 > 0 do
            JUMP    endwhile
continue:   LOAD    2
            MULT    1          } r2 ← r2 * r1
            STORE   2
            LOAD    3
            SUB     =1         } r3 ← r3 – 1
            STORE   3
            JUMP    while
endwhile:   WRITE   2          write r2
endif:      HALT
```

Abbildung **4.3** $f(n) = n^n$ (RAM–Programm)

```
        READ    1          read r1
        LOAD    1
        JGTZ    pos        if r1 ≤ 0 then write 0
        WRITE   =0
        JUMP    endif
pos:    LOAD    1
        STORE   2          r2 ← r1
        LOAD    1
        SUB     =1         r3 ← r1 – 1
        STORE   3
while:  LOAD    3
        JGTZ    continue   while r3 > 0 d
        JUMP    endwhile
continue: LOAD  2
        MULT    1          r2 ← r2 * r1
        STORE   2
        LOAD    3
        SUB     =1         r3 ← r3 – 1
        STORE   3
        JUMP    while
endwhile: WRITE 2          write r2
endif:  HALT
```

Abbildung **4.3** $f(n) = n^n$ (RAM–Programm)

```
BEGIN
  READ r1;
    IF r1 ≤ 0 THEN WRITE 0
    ELSE
      BEGIN
        r2 ← r1;
        r3 ← r1 - 1;
        WHILE r3 > 0 DO
          BEGIN
            r2 ← r2 * r1;
            r3 ← r3 - 1
          END;
        WRITE r2
      END
END
```

Abbildung 7.4: $f(n) = n^n$ (Hochsprache)

6.1.2 Komplexitätsmaße für die RAM

Uniformes Zeitmaß:	Uniformes Platzmaß:
1 Schritt = 1 Zeiteinheit	1 Register = 1 Platzeinheit

6.1.2 Komplexitätsmaße für die RAM

Uniformes Zeitmaß:	Uniformes Platzmaß:
1 Schritt = 1 Zeiteinheit	1 Register = 1 Platzeinheit

das *logarithmische Komplexitätsmaß*

$$l(i) = \begin{cases} \lfloor \log i \rfloor + 1 & i \neq 0 \\ 1 & i = 0 \end{cases}$$

6.1.2 Komplexitätsmaße für die RAM

Uniformes Zeitmaß:	Uniformes Platzmaß:
1 Schritt = 1 Zeiteinheit	1 Register = 1 Platzeinheit

das *logarithmische Komplexitätsmaß*

$$l(i) = \begin{cases} \lfloor \log i \rfloor + 1 & i \neq 0 \\ 1 & i = 0 \end{cases}$$

Operand a	Kosten t(a)	
=i	l(i)	*Wert*
i	l(i) + l(c(i))	*Wert* + *Adresse*
*i	l(i) + l(c(i)) + l(c(c(i)))	*Wert* + *Adresse* + *Adresse*

Definition 4.4 *Die* uniforme (logarithmische) Zeitkomplexität *eines RAM–Programms A ist die maximale Anzahl $T_A(n)$ der Summen der uniformen (logarithmischen) Kosten aller Schritte des Programms bis zur Termination, wobei das Maximum über alle Eingaben der uniformen (logarithmischen) Größe n gebildet wird.*

Definition 4.4 *Die* uniforme (logarithmische) Zeitkomplexität *eines RAM–Programms A ist die maximale Anzahl $T_A(n)$ der Summen der uniformen (logarithmischen) Kosten aller Schritte des Programms bis zur Termination, wobei das Maximum über alle Eingaben der uniformen (logarithmischen) Größe n gebildet wird.*

Die uniforme (logarithmische) Platzkomplexität *eines RAM–Programms A ist die maximale Summe $S_A(n)$ der uniformen (logarithmischen) Kosten aller Register, die dieses Programm bis zur Termination addressiert, wobei das Maximum über alle Eingaben der uniformen (logarithmischen) Größe n gebildet wird.*

	uniforme Kosten	logarith. Kosten
Zeitkomplexität	$O(n)$	$O(n^2 \log n)$
Platzkomplexität	$O(1)$	$O(n \log n)$

```
        READ    1                   read r1
        LOAD    1
        JGTZ    pos                 if r1 ≤ 0 then write 0
        WRITE   =0
        JUMP    endif
pos:    LOAD    1
        STORE   2                   r2 ← r1
        LOAD    1
        SUB     =1                  r3 ← r1 – 1
        STORE   3
while:  LOAD    3
        JGTZ    continue            while r3 > 0 do
        JUMP    endwhile
continue: LOAD  2
        MULT    1                   r2 ← r2 * r1
        STORE   2
        LOAD    3
        SUB     =1                  r3 ← r3 – 1
        STORE   3
        JUMP    while
endwhile: WRITE 2                   write r2
endif:  HALT
```

Abbildung 4.3 $f(n) = n^n$ (RAM–Programm)

	uniforme Kosten	logarith. Kosten
Zeitkomplexität	$O(n)$	$O(n^2 \log n)$
Platzkomplexität	$O(1)$	$O(n \log n)$

```
        READ    1                   read r1
        LOAD    1
        JGTZ    pos                 if r1 ≤ 0 then write 0
        WRITE   =0
        JUMP    endif
pos:    LOAD    1
        STORE   2                   r2 ← r1
        LOAD    1
        SUB     =1                  r3 ← r1 – 1
        STORE   3
while:  LOAD    3
        JGTZ    continue            while r3 > 0 do
        JUMP    endwhile
continue: LOAD  2
        MULT    1                   r2 ← r2 * r1
        STORE   2
        LOAD    3
        SUB     =1                  r3 ← r3 – 1
        STORE   3
        JUMP    while
endwhile: WRITE 2                   write r2
endif:  HALT
```

Abbildung 4.3 $f(n) = n^n$ (RAM–Programm)

```
BEGIN
  READ r1;
    IF r1 ≤ 0 THEN WRITE 0
    ELSE
      BEGIN
        r2 ← r1;
        r3 ← r1 - 1;
        WHILE r3 > 0 DO
          BEGIN
            r2 ← r2 * r1;
            r3 ← r3 - 1
          END;
        WRITE r2
      END
END
```

	uniforme Kosten	logarith. Kosten
Zeitkomplexität	$O(n)$	$O(n^2 \log n)$
Platzkomplexität	$O(1)$	$O(n \log n)$

```
        READ    1          read r1
        LOAD    1
        JGTZ    pos        if r1 ≤ 0 then write 0
        WRITE   =0
        JUMP    endif
pos:    LOAD    1
        STORE   2          r2 ← r1
        LOAD    1
        SUB     =1         r3 ← r1 – 1
        STORE   3
while:  LOAD    3
        JGTZ    continue   while r3 > 0 do
        JUMP    endwhile
continue: LOAD  2
        MULT    1          r2 ← r2 * r1
        STORE   2
        LOAD    3
        SUB     =1         r3 ← r3 – 1
        STORE   3
        JUMP    while
endwhile: WRITE 2          write r2
endif:  HALT
```

Abbildung 4.3 $f(n) = n^n$ (RAM–Programm)

```
BEGIN
  READ r1;
    IF r1 ≤ 0 THEN WRITE 0
    ELSE
      BEGIN
        r2 ← r1;
        r3 ← r1 - 1;
        WHILE r3 > 0 DO
          BEGIN
            r2 ← r2 * r1;
            r3 ← r3 - 1
          END;
        WRITE r2
      END
END
```

	uniforme Kosten	logarith. Kosten
Zeitkomplexität	$O(n)$	$O(n^2 \log n)$
Platzkomplexität	$O(1)$	$O(n \log n)$

```
        READ    1          read r1
        LOAD    1
        JGTZ    pos        if r1 ≤ 0 then write 0
        WRITE   =0
        JUMP    endif
pos:    LOAD    1
        STORE   2          r2 ← r1
        LOAD    1
        SUB     =1         r3 ← r1 – 1
        STORE   3
while:  LOAD    3
        JGTZ    continue   while r3 > 0 do
        JUMP    endwhile
continue: LOAD  2
        MULT    1          r2 ← r2 * r1
        STORE   2
        LOAD    3
        SUB     =1         r3 ← r3 – 1
        STORE   3
        JUMP    while
endwhile: WRITE 2          write r2
endif:  HALT
```

Abbildung 4.3 $f(n) = n^n$ (RAM–Programm)

```
BEGIN
  READ r1;
    IF r1 ≤ 0 THEN WRITE 0
    ELSE
      BEGIN
        r2 ← r1;
        r3 ← r1 - 1;
        WHILE r3 > 0 DO
          BEGIN
            r2 ← r2 * r1;
            r3 ← r3 - 1
          END;
        WRITE r2
      END
END
```

	uniforme Kosten	logarith. Kosten
Zeitkomplexität	$O(n)$	$O(n^2 \log n)$
Platzkomplexität	$O(1)$	$O(n \log n)$

```
        READ    1          read r1
        LOAD    1
        JGTZ    pos        if r1 ≤ 0 then write 0
        WRITE   =0
        JUMP    endif
pos:    LOAD    1
        STORE   2          r2 ← r1
        LOAD    1
        SUB     =1         r3 ← r1 – 1
        STORE   3
while:  LOAD    3
        JGTZ    continue   while r3 > 0 do
        JUMP    endwhile
continue: LOAD  2
        MULT    1          r2 ← r2 * r1
        STORE   2
        LOAD    3
        SUB     =1         r3 ← r3 – 1
        STORE   3
        JUMP    while
endwhile: WRITE 2          write r2
endif:  HALT
```

Abbildung 4.3 $f(n) = n^n$ (RAM–Programm)

$$n^i \cdot n$$

```
BEGIN
  READ r1;
    IF r1 ≤ 0 THEN WRITE 0
    ELSE
      BEGIN
        r2 ← r1;
        r3 ← r1 - 1;
        WHILE r3 > 0 DO
          BEGIN
            r2 ← r2 * r1;
            r3 ← r3 - 1
          END;
        WRITE r2
      END
END
```

	uniforme Kosten	logarith. Kosten
Zeitkomplexität	$O(n)$	$O(n^2 \log n)$
Platzkomplexität	$O(1)$	$O(n \log n)$

```
        READ    1          read r1
        LOAD    1
        JGTZ    pos        if r1 ≤ 0 then write 0
        WRITE   =0
        JUMP    endif
pos:    LOAD    1
        STORE   2          r2 ← r1
        LOAD    1
        SUB     =1         r3 ← r1 – 1
        STORE   3
while:  LOAD    3
        JGTZ    continue   while r3 > 0 do
        JUMP    endwhile
continue: LOAD  2
        MULT    1          r2 ← r2 * r1
        STORE   2
        LOAD    3
        SUB     =1         r3 ← r3 – 1
        STORE   3
        JUMP    while
endwhile: WRITE 2          write r2
endif:  HALT
```

Abbildung 4.3 $f(n) = n^n$ (RAM–Programm)

$$n^i \cdot n$$

$$l(n^i) + l(n) \approx (i+1) \log n$$

```
BEGIN
  READ r1;
    IF r1 ≤ 0 THEN WRITE 0
    ELSE
      BEGIN
        r2 ← r1;
        r3 ← r1 - 1;
        WHILE r3 > 0 DO
          BEGIN
            r2 ← r2 * r1;
            r3 ← r3 - 1
          END;
        WRITE r2
      END
END
```

	uniforme Kosten	logarith. Kosten
Zeitkomplexität	$O(n)$	$O(n^2 \log n)$
Platzkomplexität	$O(1)$	$O(n \log n)$

```
        READ    1           read r1
        LOAD    1
        JGTZ    pos         if r1 ≤ 0 then write 0
        WRITE   =0
        JUMP    endif
pos:    LOAD    1
        STORE   2           r2 ← r1
        LOAD    1
        SUB     =1          r3 ← r1 – 1
        STORE   3
while:  LOAD    3
        JGTZ    continue    while r3 > 0 do
        JUMP    endwhile
continue: LOAD  2
        MULT    1           r2 ← r2 * r1
        STORE   2
        LOAD    3
        SUB     =1          r3 ← r3 – 1
        STORE   3
        JUMP    while
endwhile: WRITE 2           write r2
endif:  HALT
```

Abbildung 4.3 $f(n) = n^n$ (RAM–Programm)

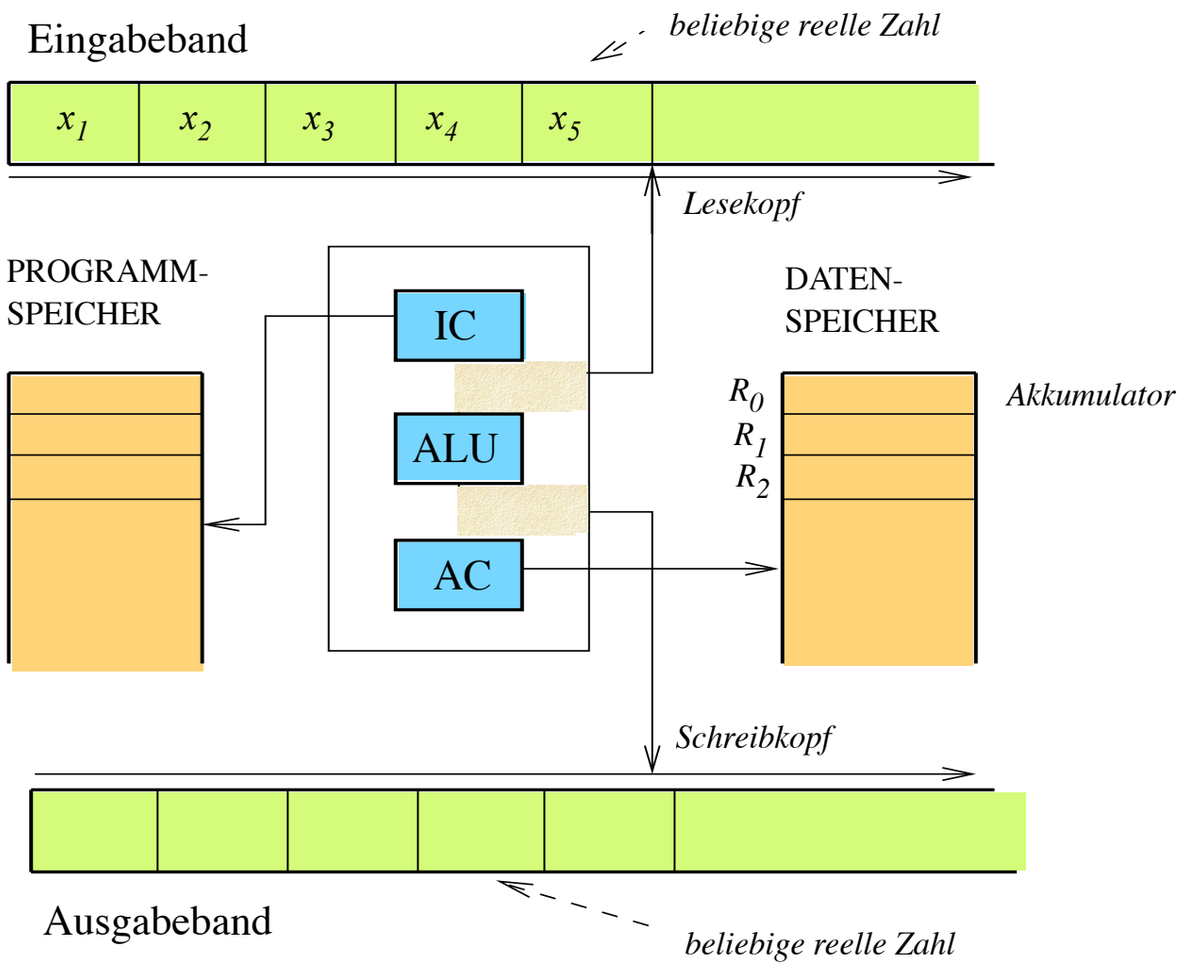
$$n^i \cdot n$$

$$l(n^i) + l(n) \approx (i+1)\log n$$

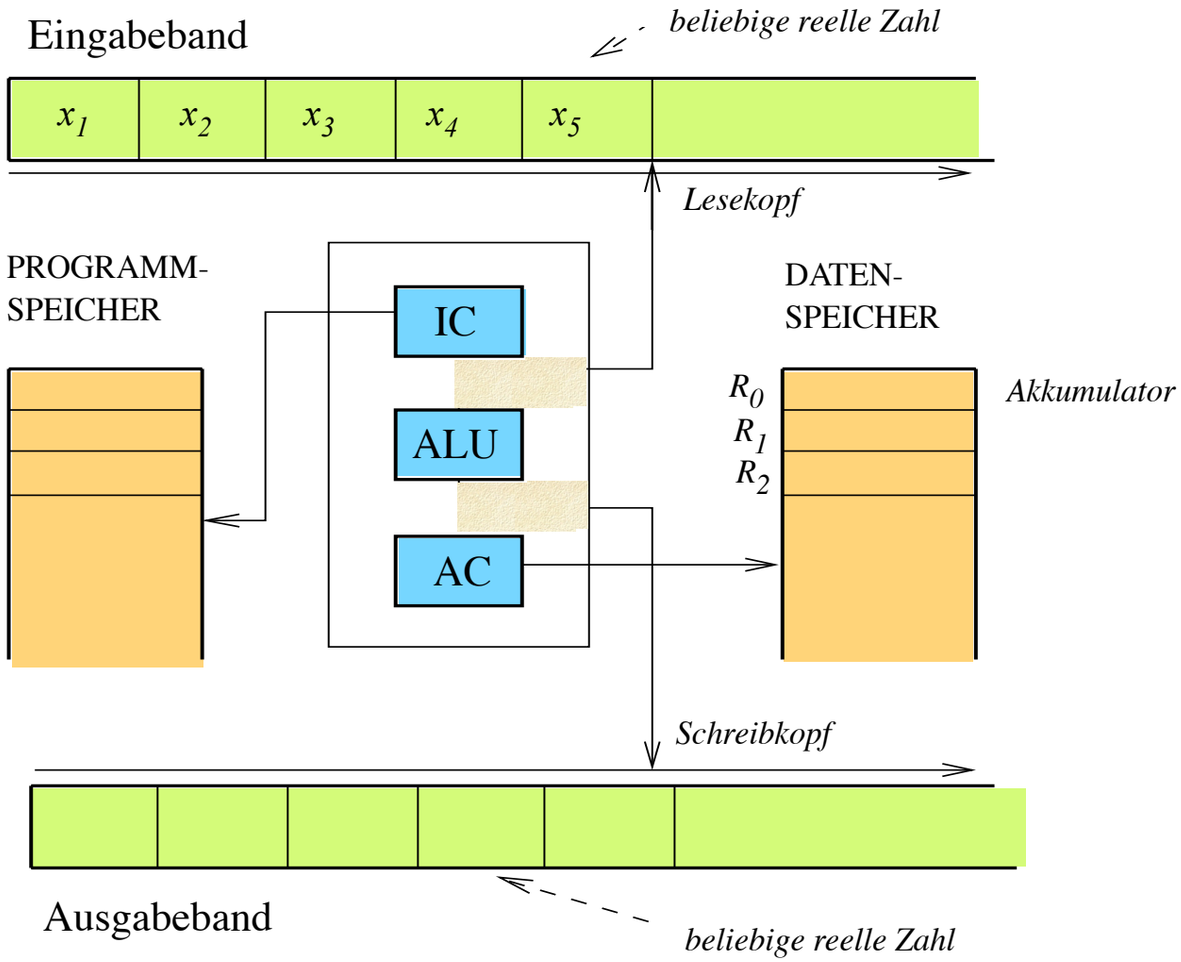
$$\sum_{i=1}^{n-1} (i+1)\log n = O(n^2 \log n)$$

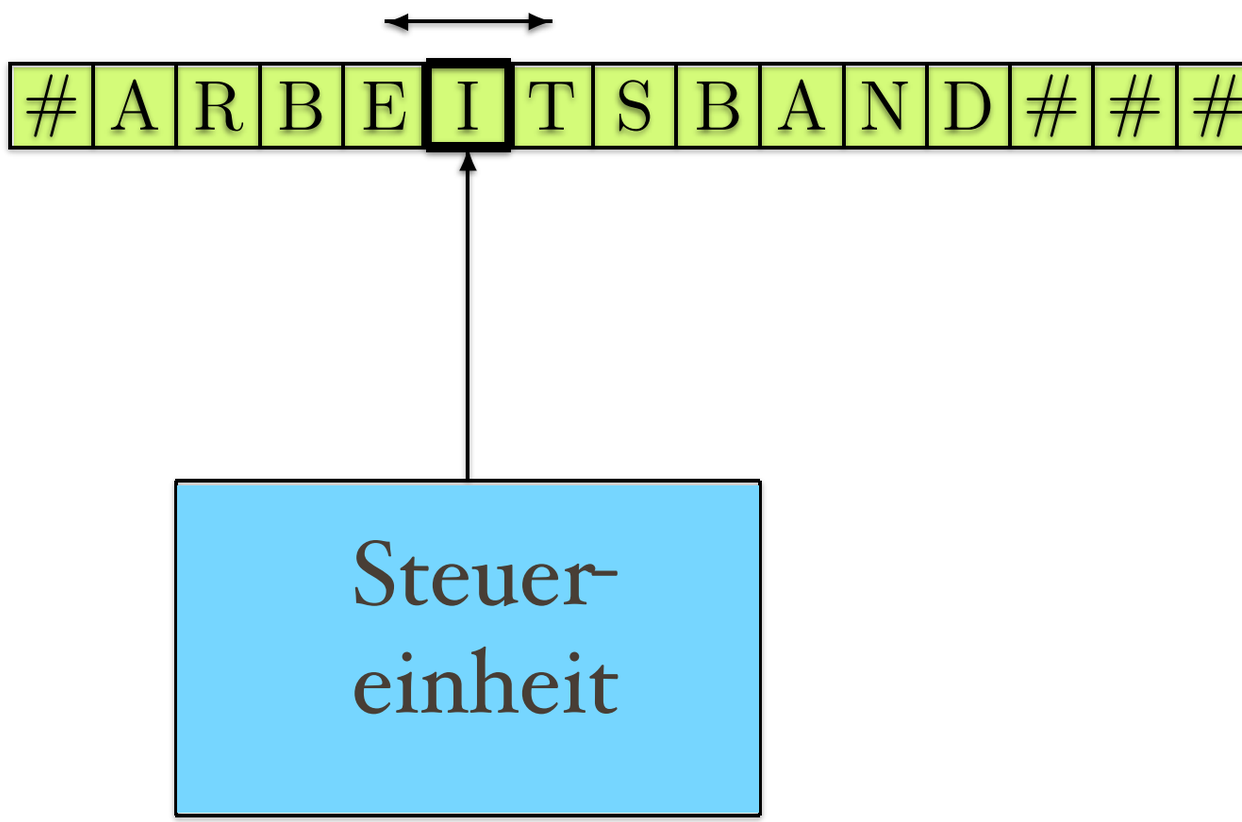
```
BEGIN
  READ r1;
    IF r1 ≤ 0 THEN WRITE 0
    ELSE
      BEGIN
        r2 ← r1;
        r3 ← r1 - 1;
        WHILE r3 > 0 DO
          BEGIN
            r2 ← r2 * r1;
            r3 ← r3 - 1
          END;
        WRITE r2
      END
END
```

4.1.3 Wechselseitige Simulation einer RAM und einer Turing-Machine

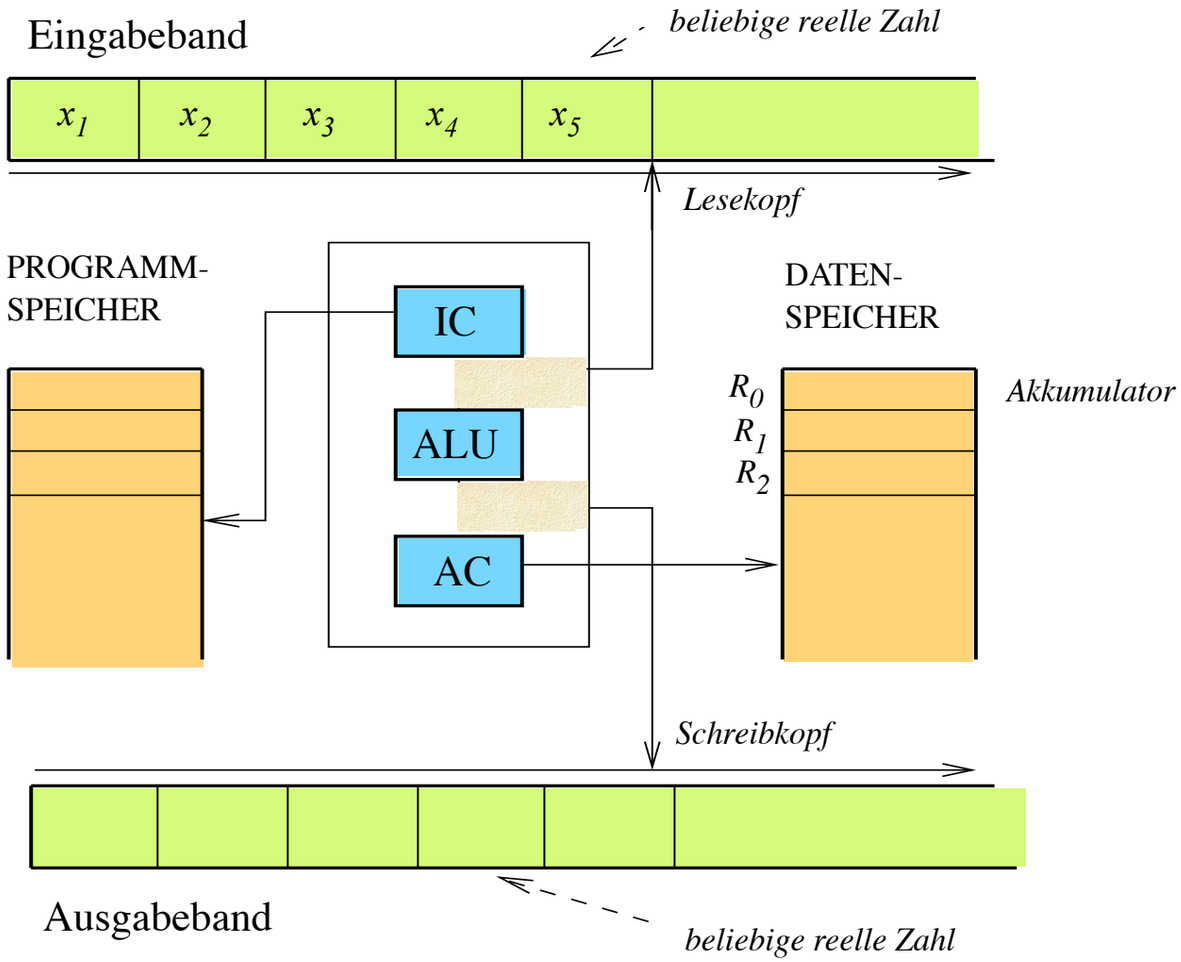


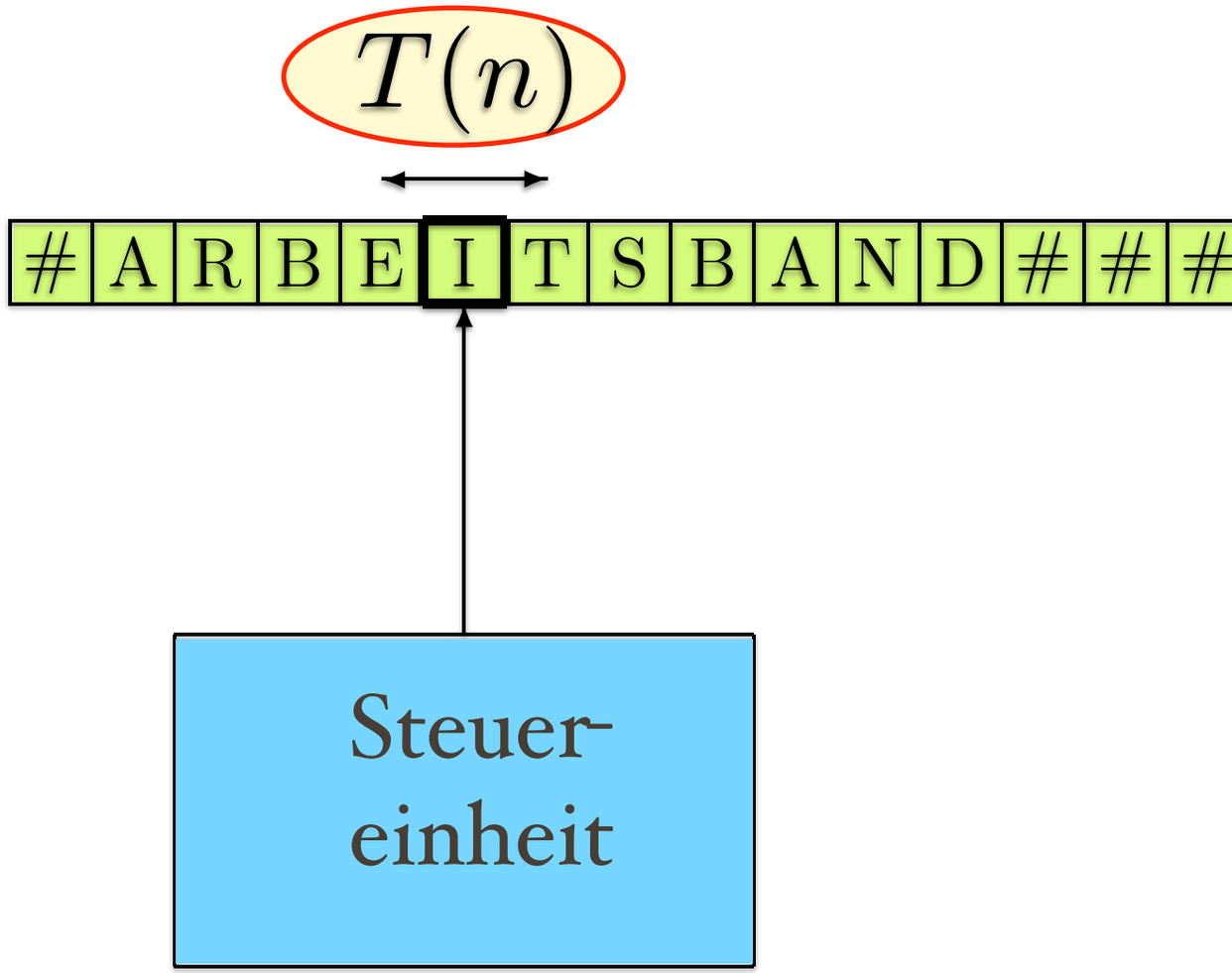
4.1.3 Wechselseitige Simulation einer RAM und einer Turing-Machine





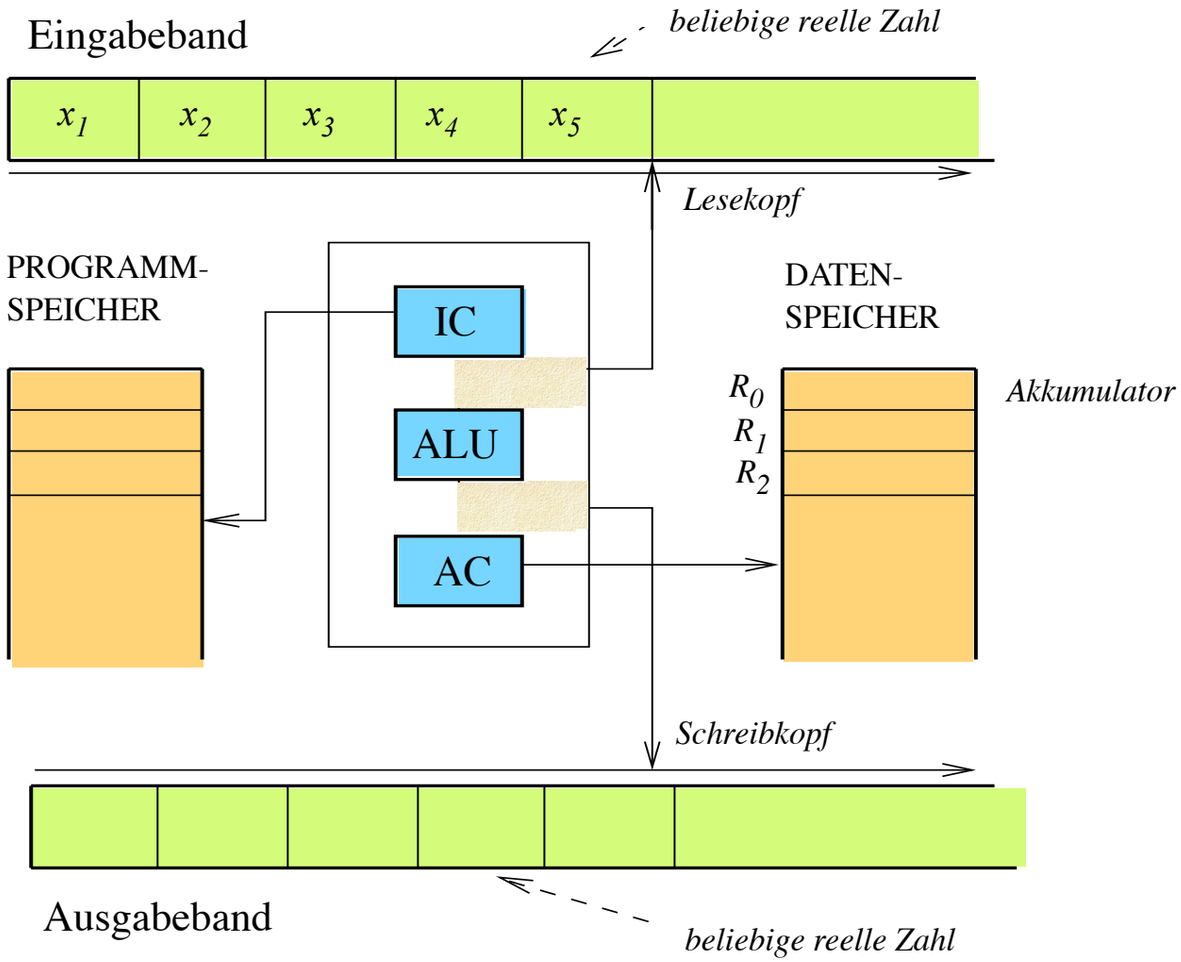
4.1.3 Wechselseitige Simulation einer RAM und einer Turing-Machine

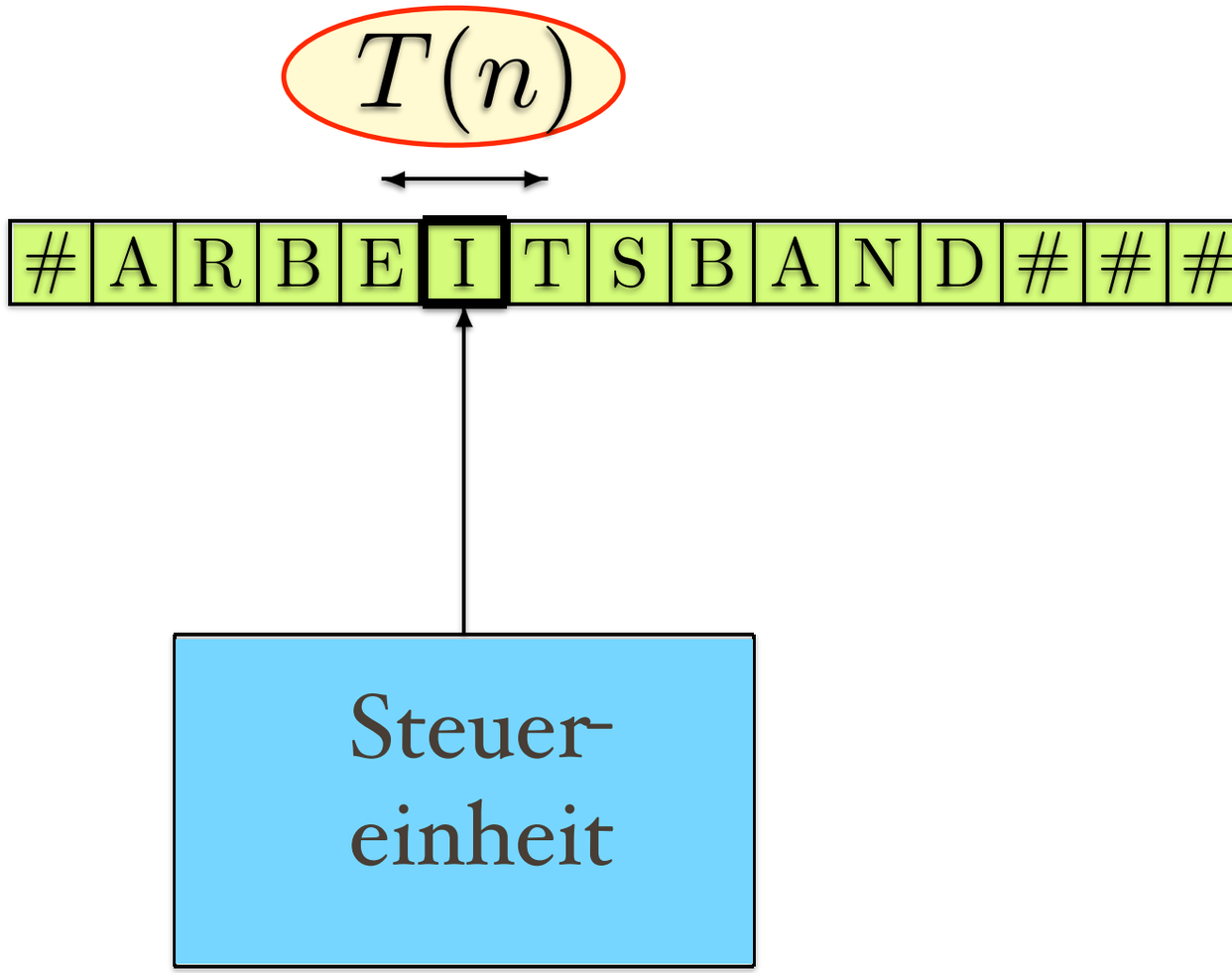




?

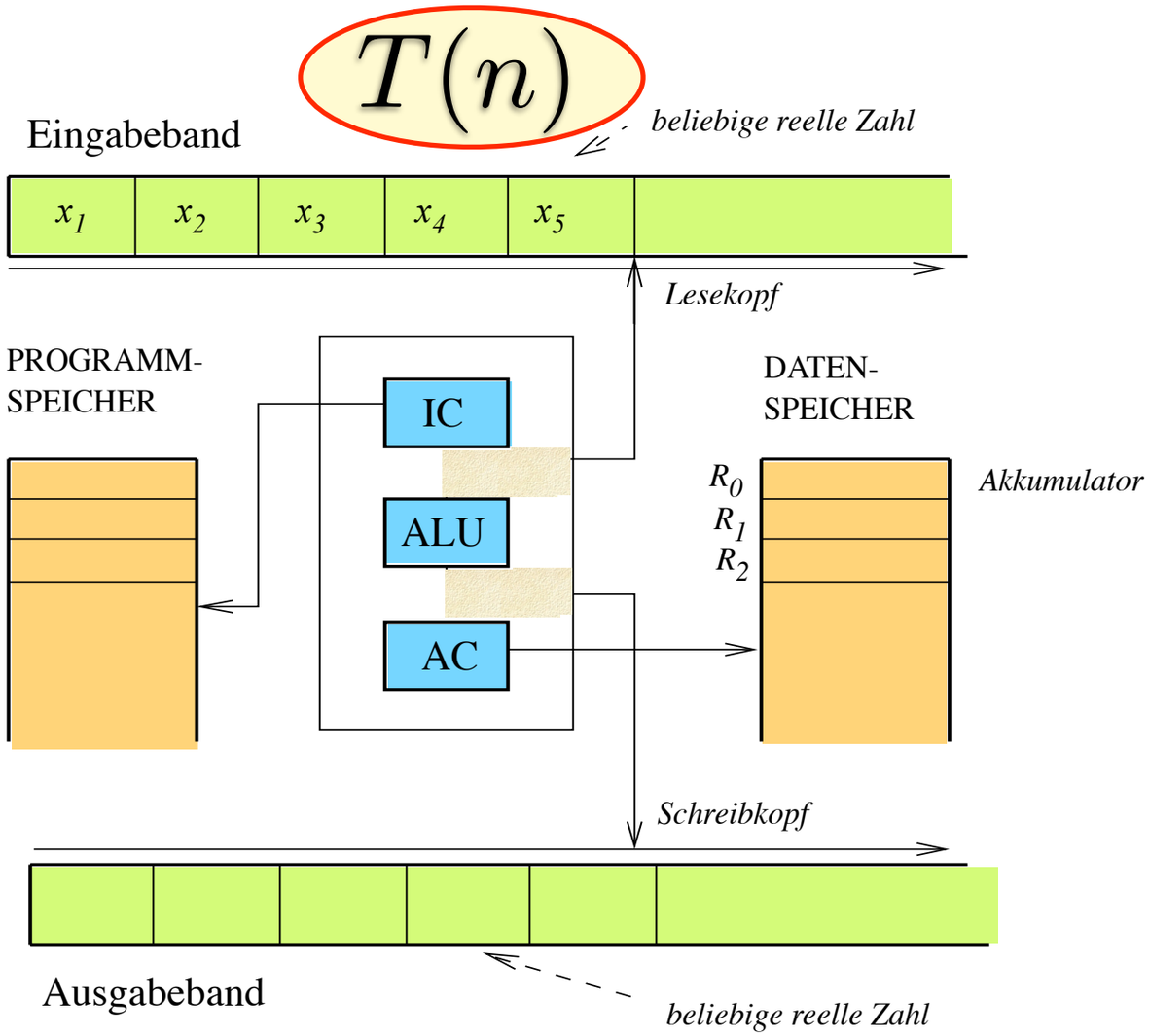
4.1.3 Wechselseitige Simulation einer RAM und einer Turing-Machine

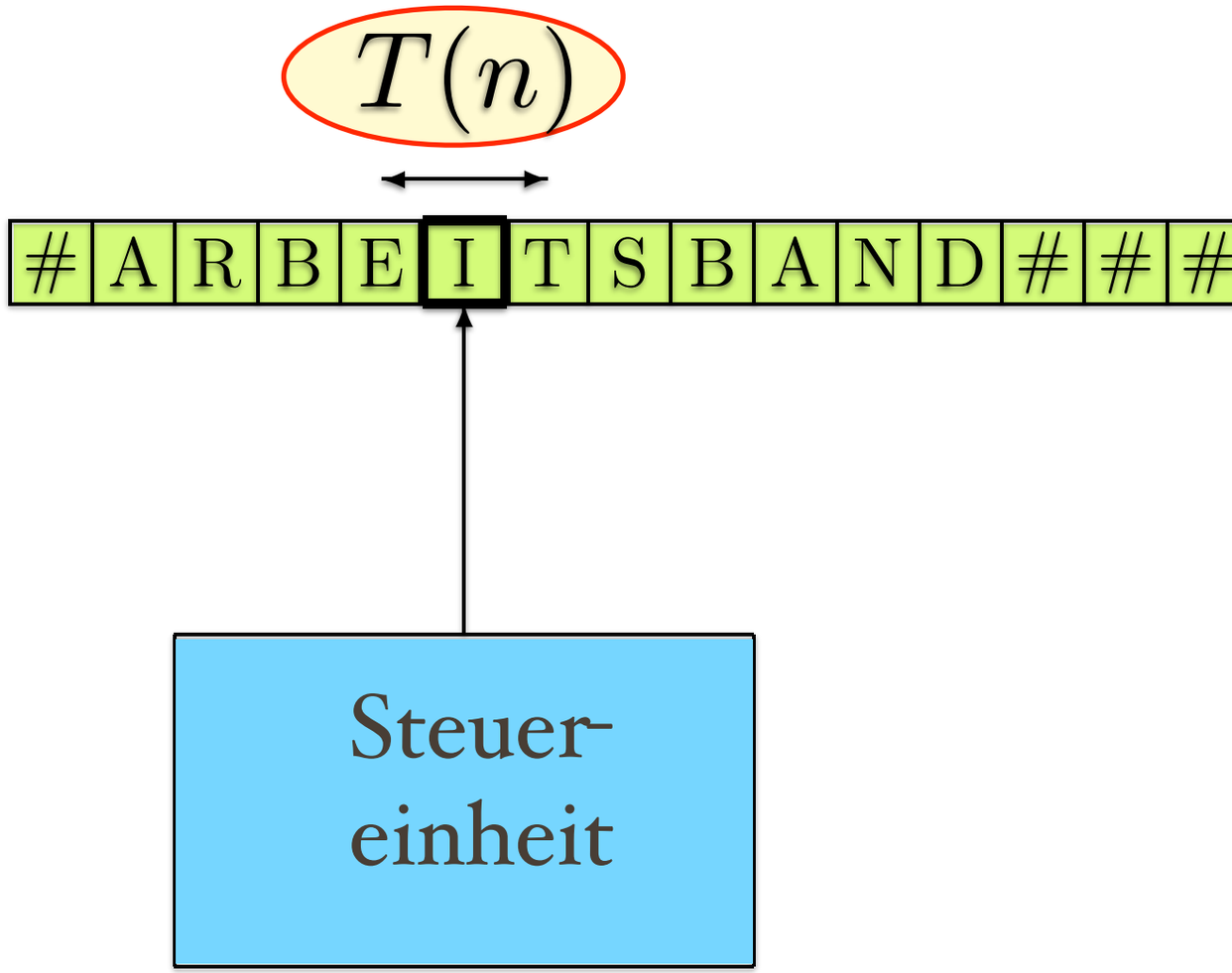




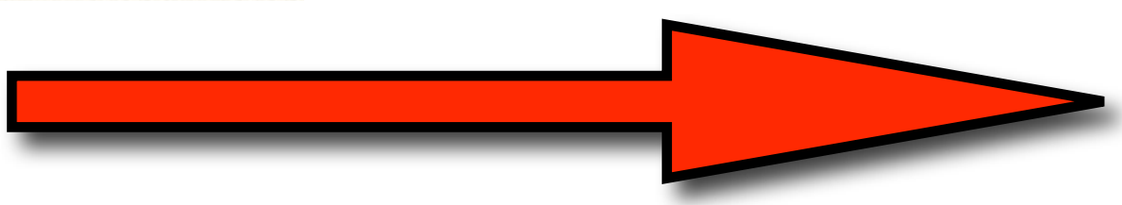
$O(T(n) \log T(n))$

4.1.3 Wechselseitige Simulation einer RAM und einer Turing-Machine



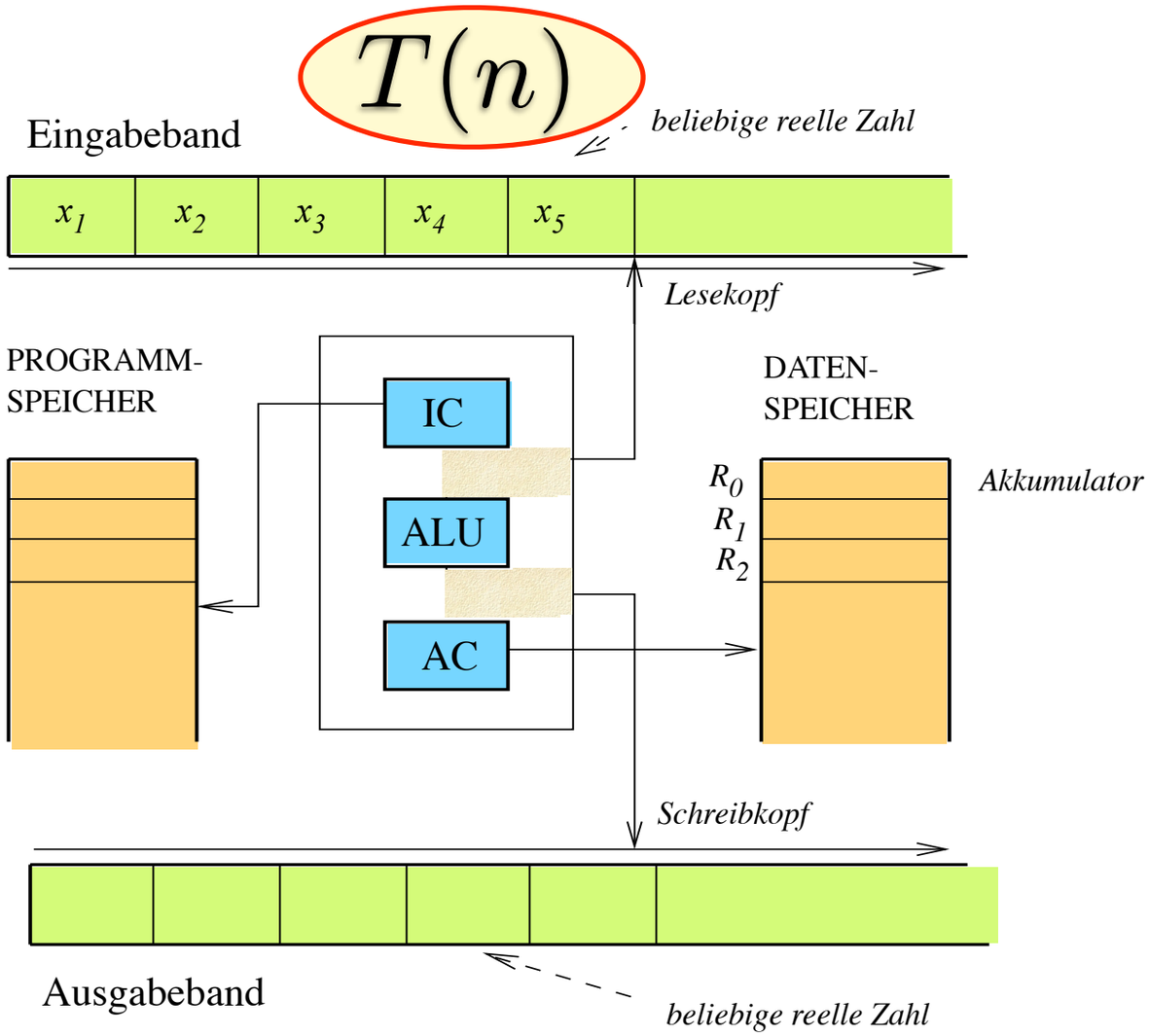


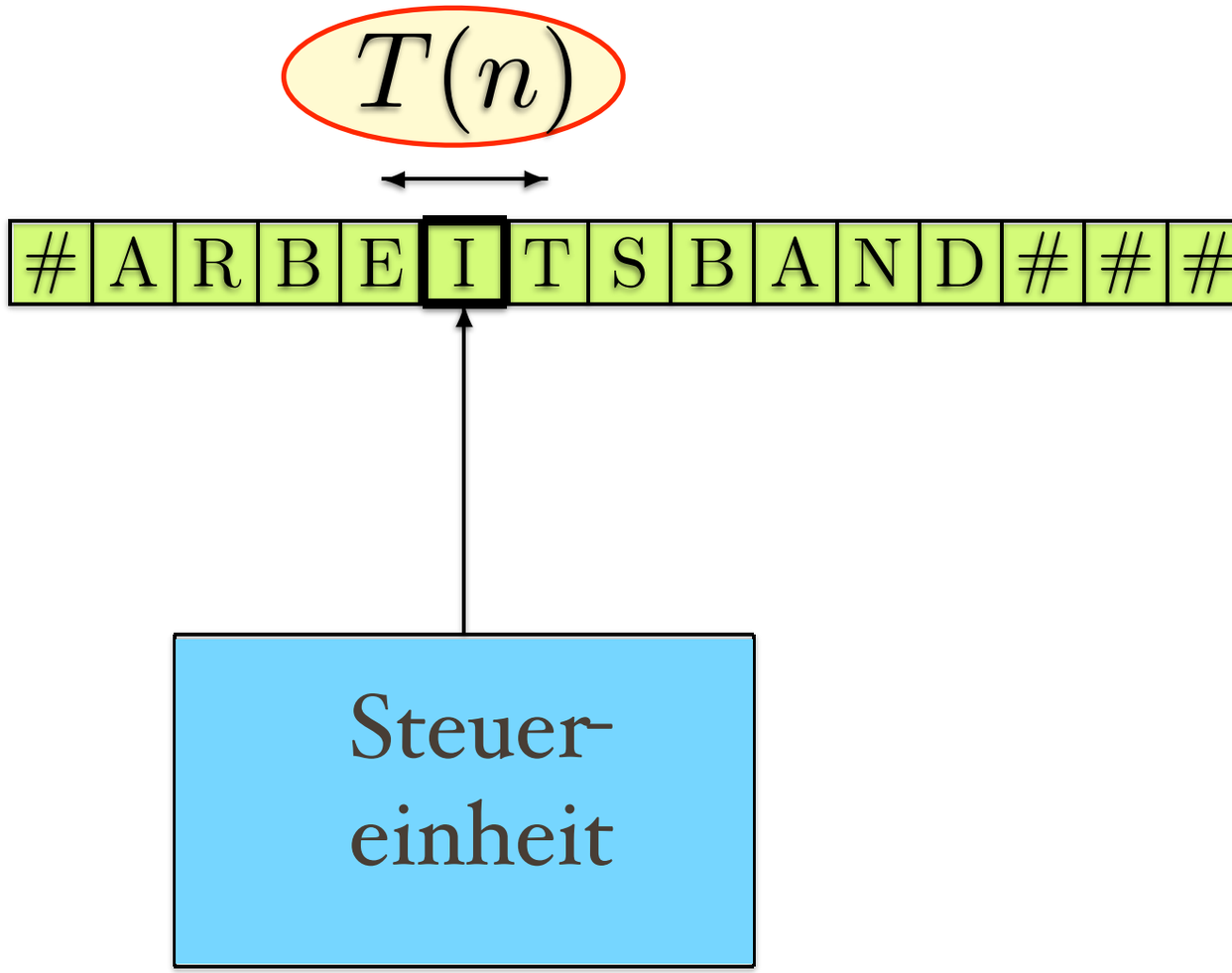
$O(T(n) \log T(n))$



?

4.1.3 Wechselseitige Simulation einer RAM und einer Turing-Machine



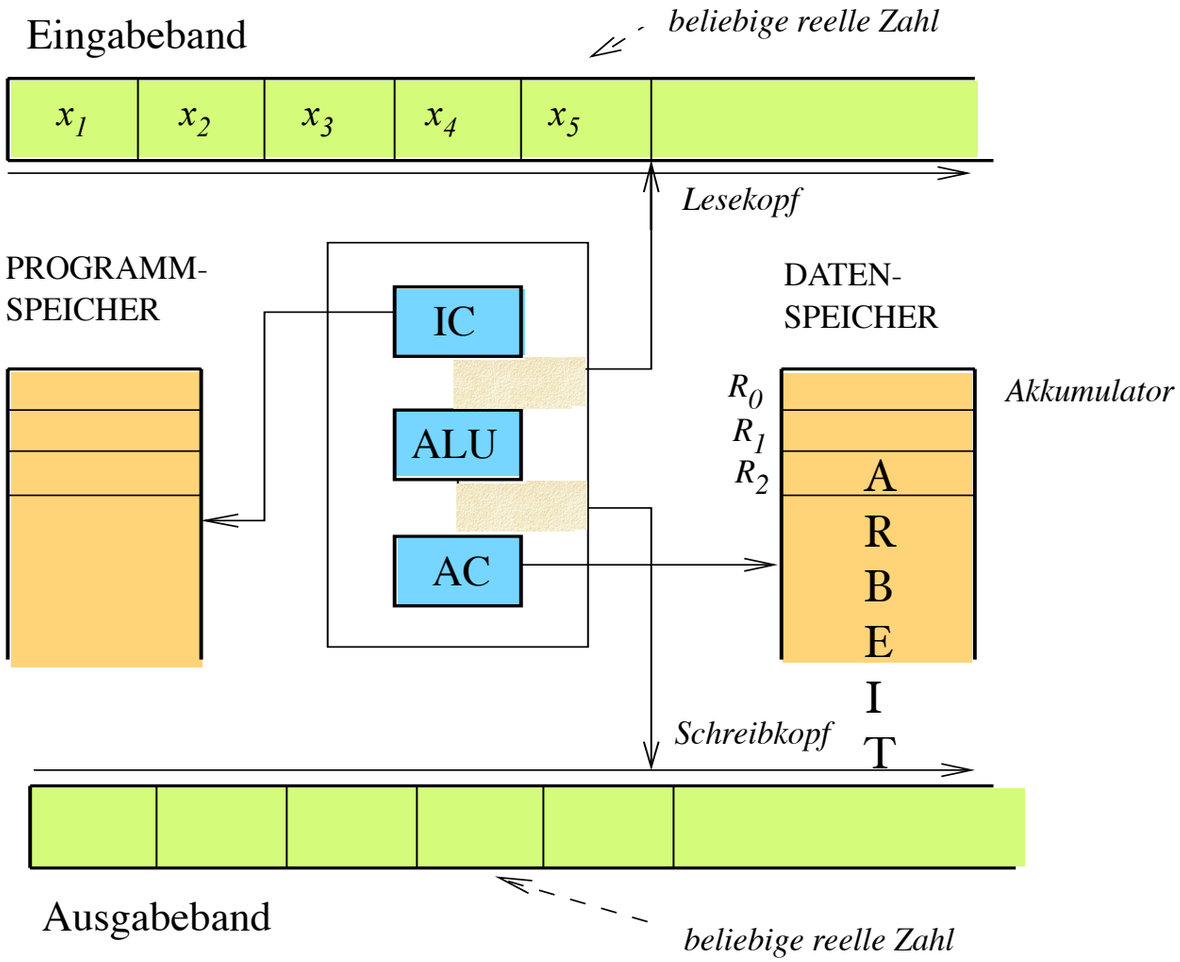


$O(T(n) \log T(n))$

$O(T^2(n))$

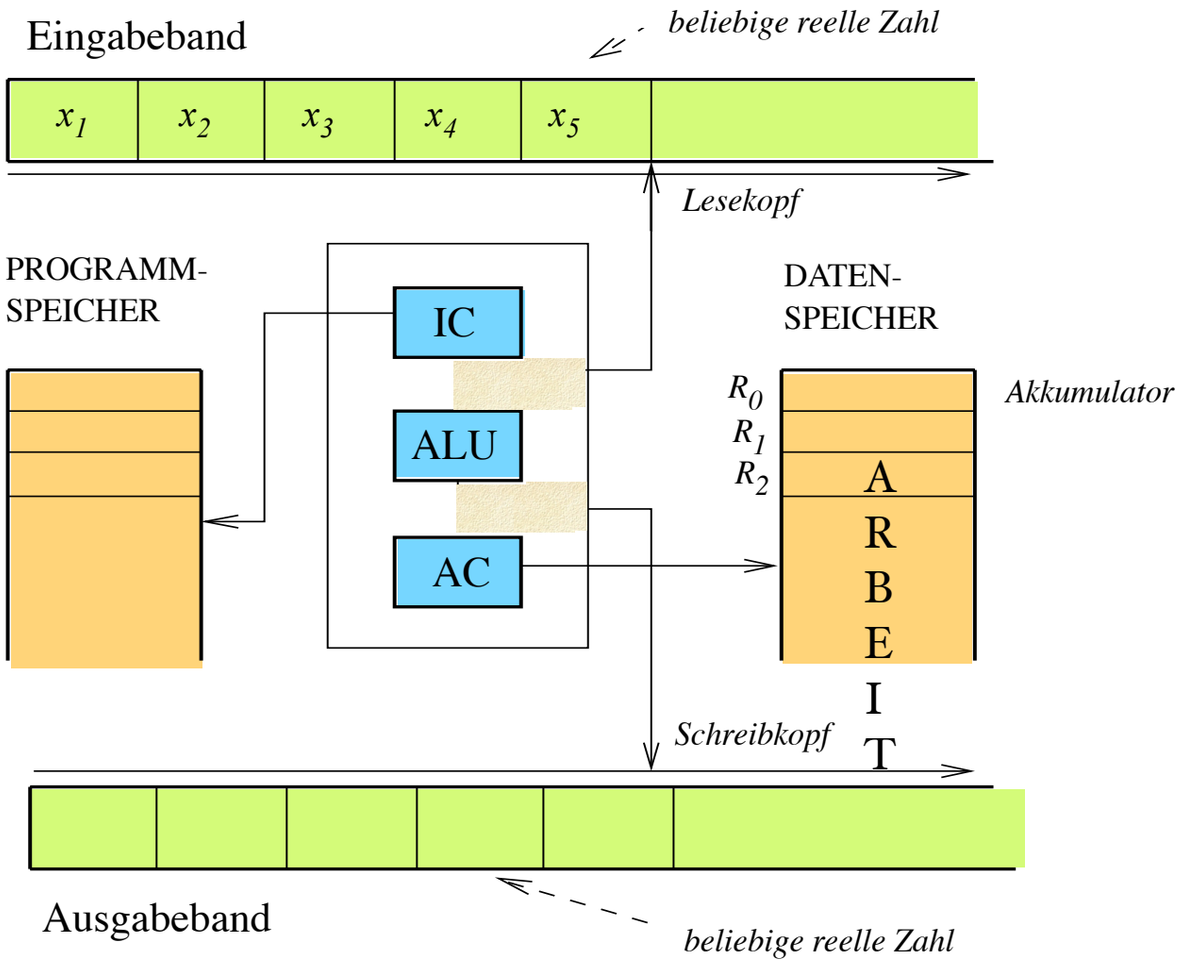
RAM$_+$
ohne MULT, DIV

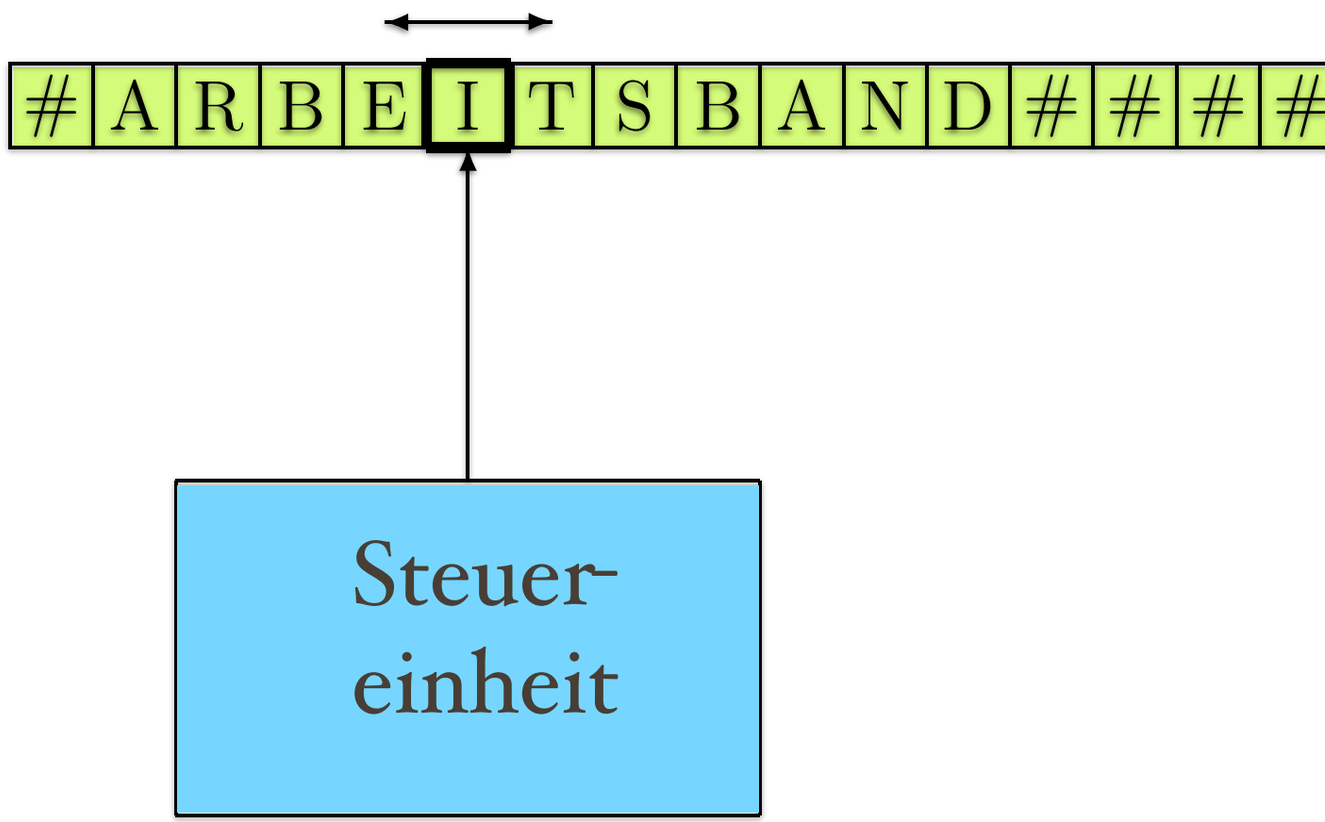
Satz 4.7 *Eine $T(n)$–zeitbeschränkte Mehrband DTM M kann durch eine RAM_+ simuliert werden, die im uniformen Maß $O(T(n))$-zeitbeschränkt ist und die im logarithmischen Maß $O(T(n)\log T(n))$-zeitbeschränkt ist.*



RAM_+
ohne MULT, DIV

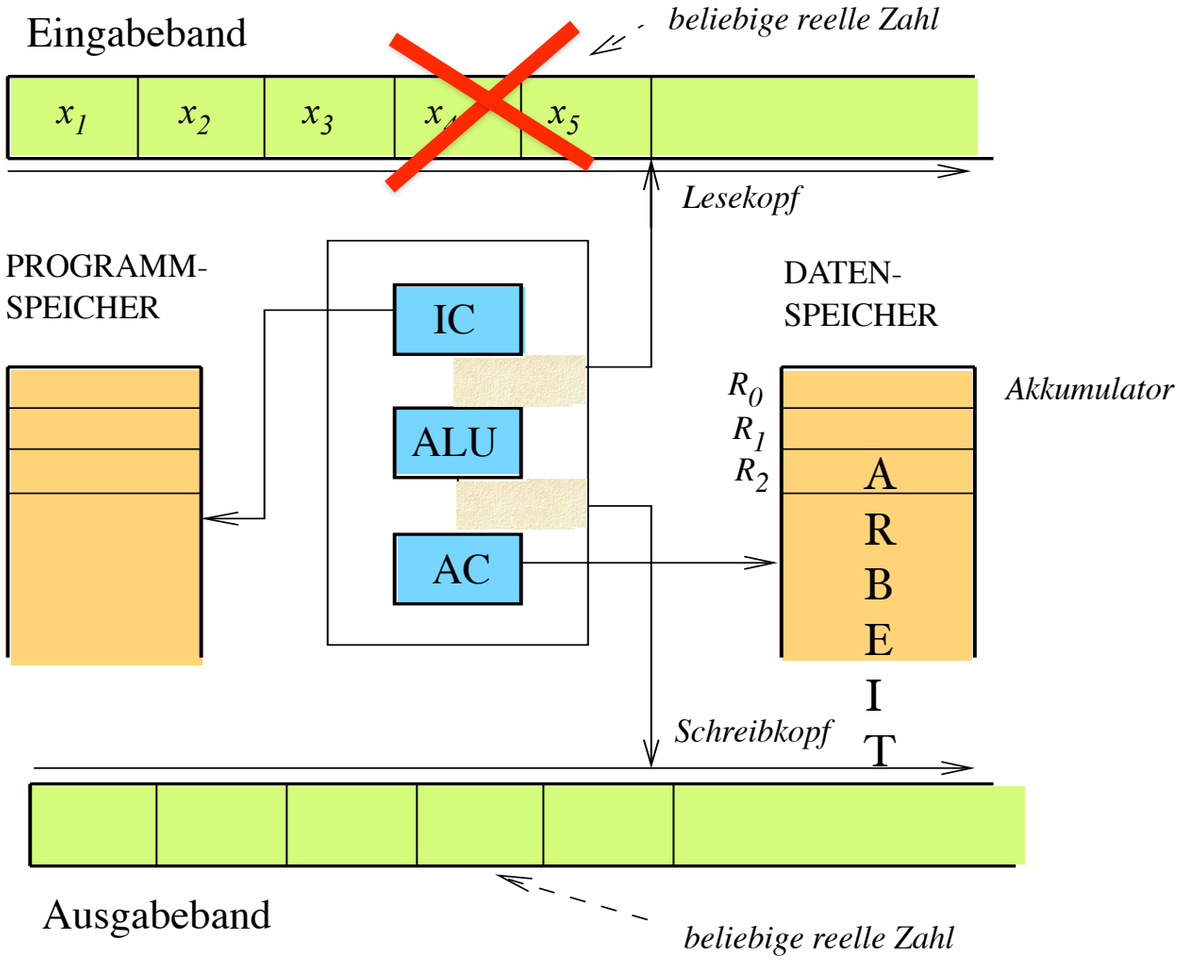
Satz 4.7 *Eine $T(n)$–zeitbeschränkte Mehrband DTM M kann durch eine RAM_+ simuliert werden, die im uniformen Maß $O(T(n))$-zeitbeschränkt ist und die im logarithmischen Maß $O(T(n)\log T(n))$-zeitbeschränkt ist.*

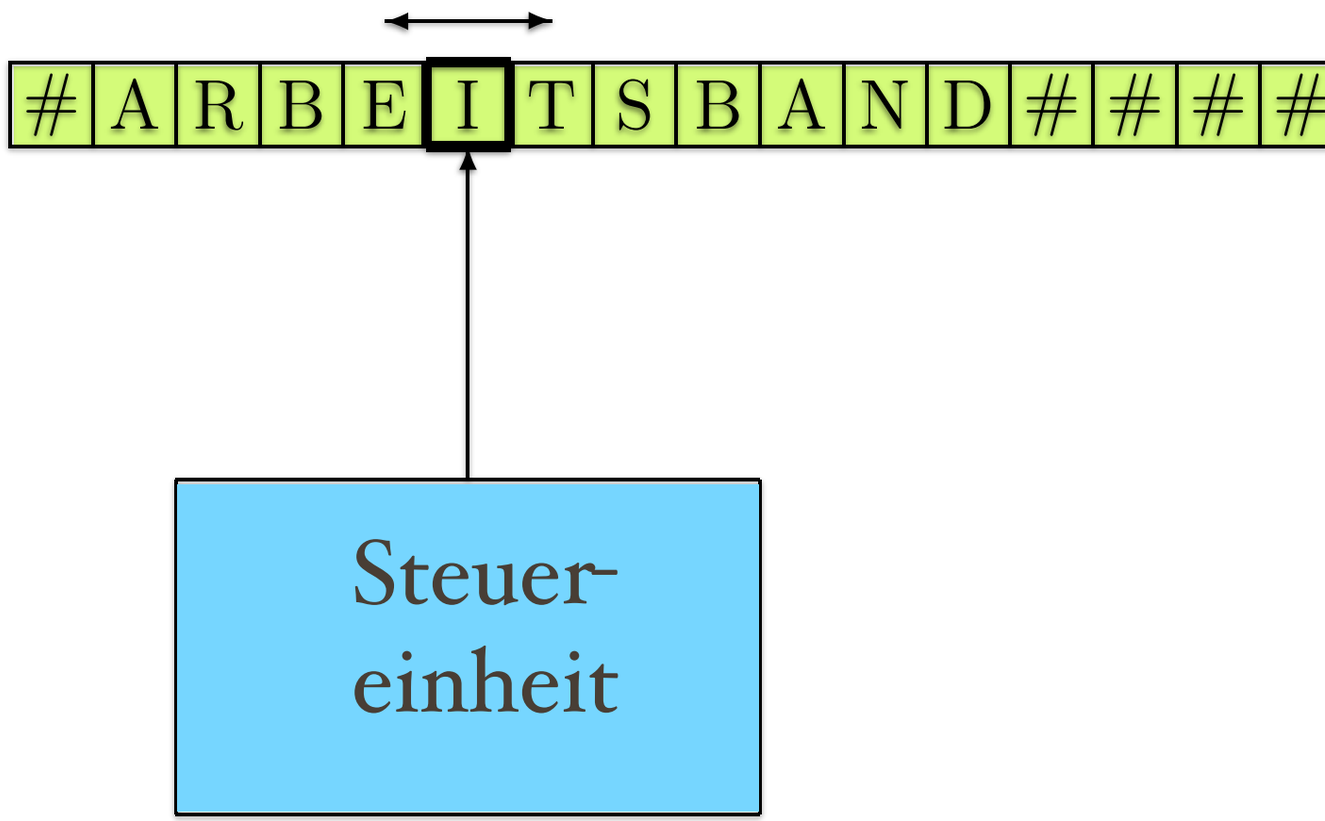




RAM_+
ohne MULT, DIV

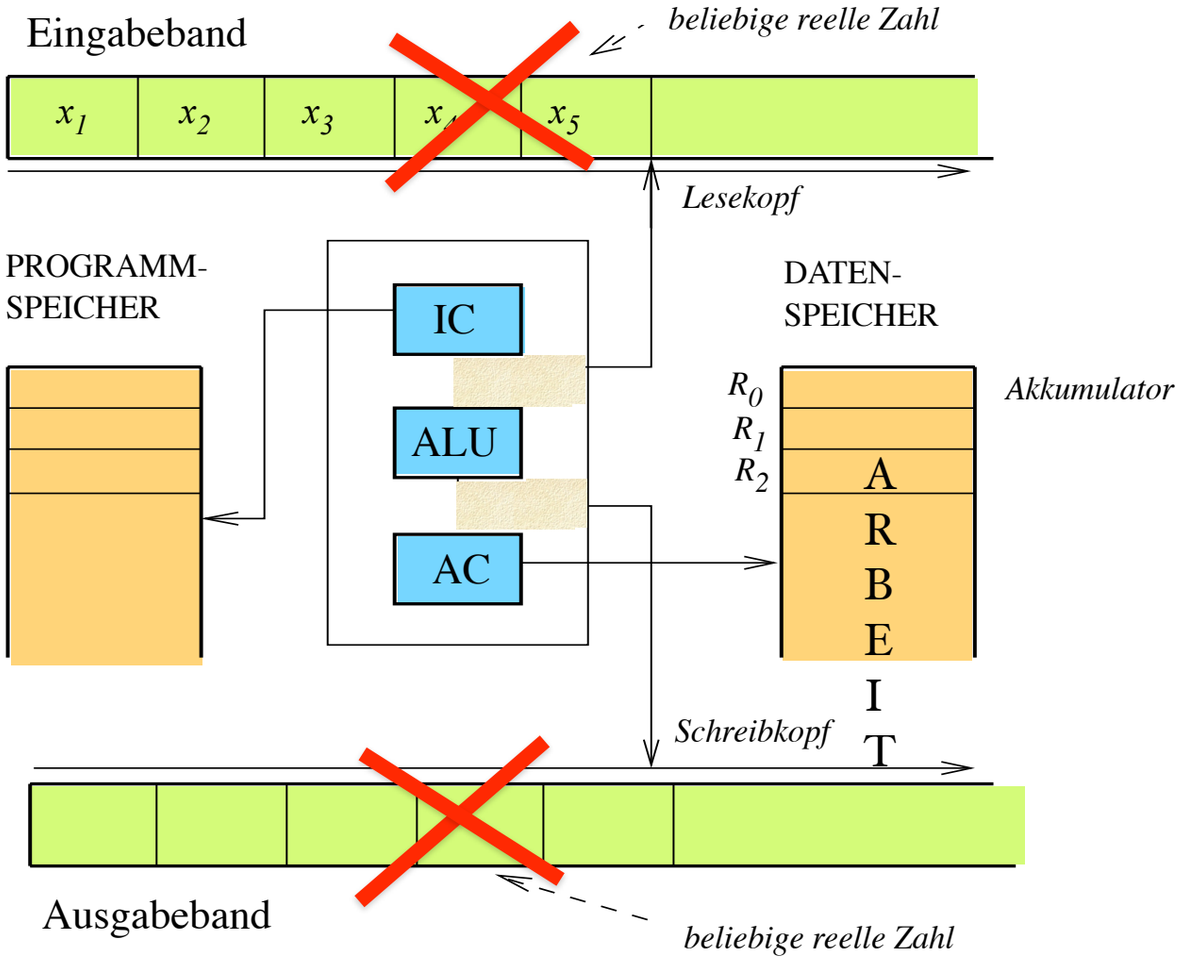
Satz 4.7 *Eine $T(n)$–zeitbeschränkte Mehrband DTM M kann durch eine RAM_+ simuliert werden, die im uniformen Maß $O(T(n))$-zeitbeschränkt ist und die im logarithmischen Maß $O(T(n)\log T(n))$-zeitbeschränkt ist.*

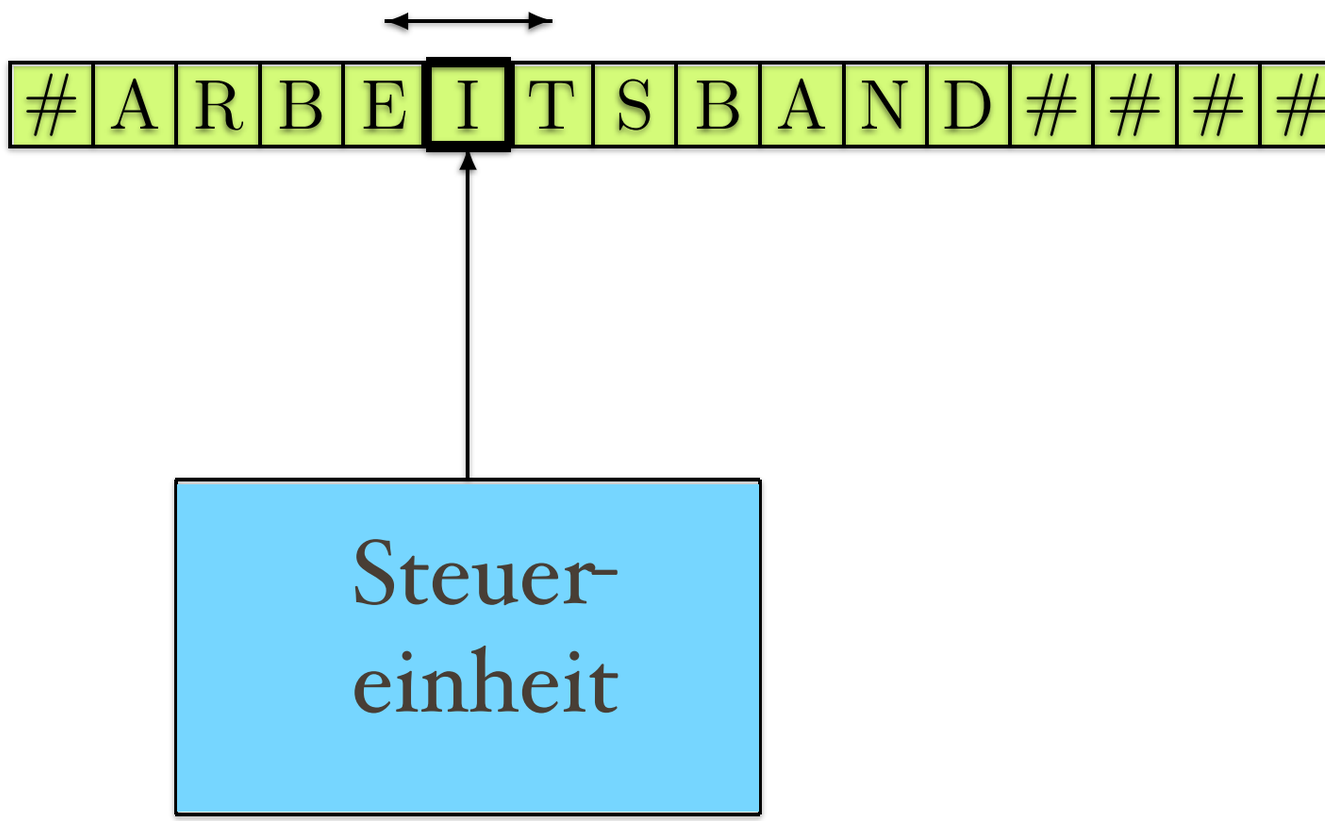




RAM$_+$
ohne MULT, DIV

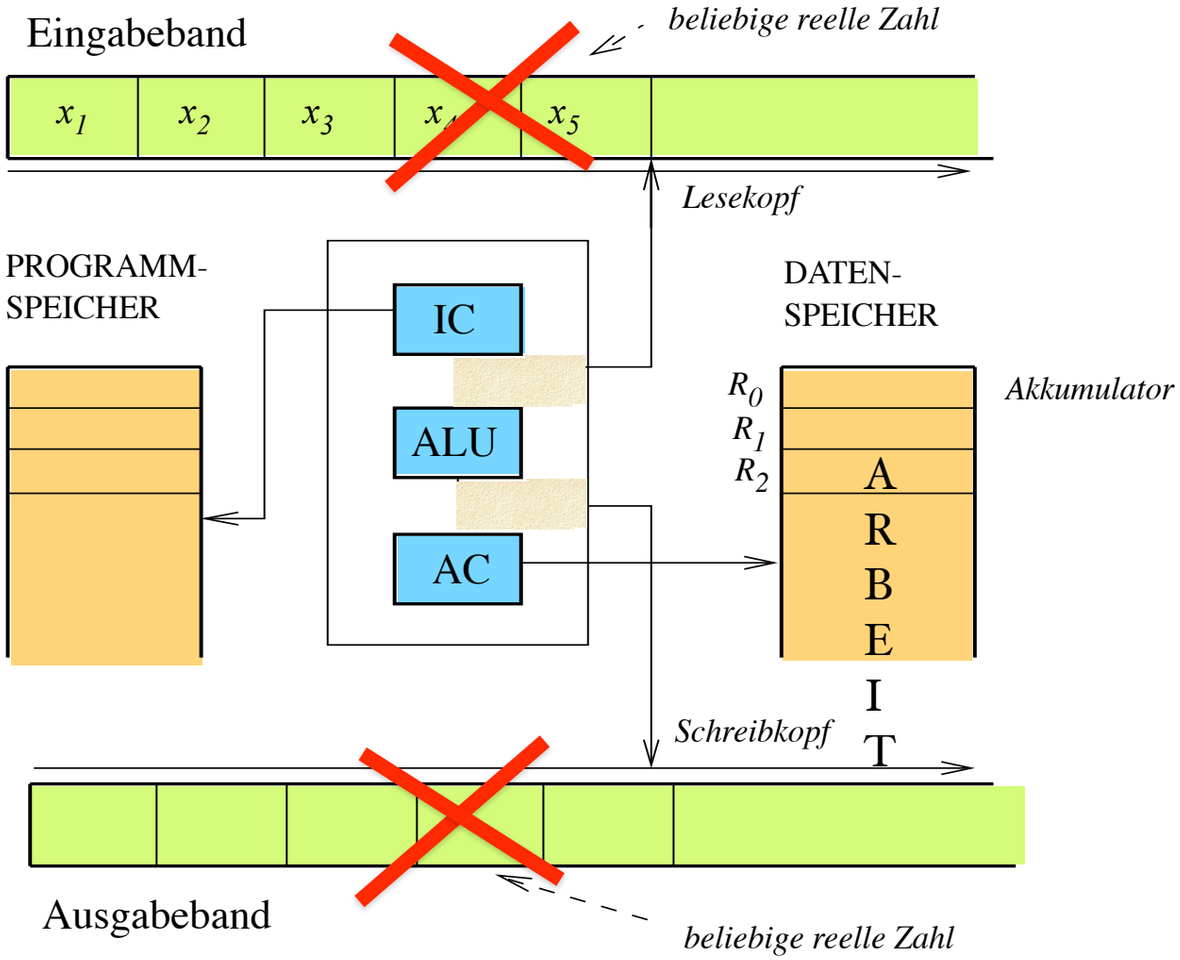
Satz 4.7 *Eine $T(n)$–zeitbeschränkte Mehrband DTM M kann durch eine RAM_+ simuliert werden, die im uniformen Maß $O(T(n))$-zeitbeschränkt ist und die im logarithmischen Maß $O(T(n)\log T(n))$-zeitbeschränkt ist.*

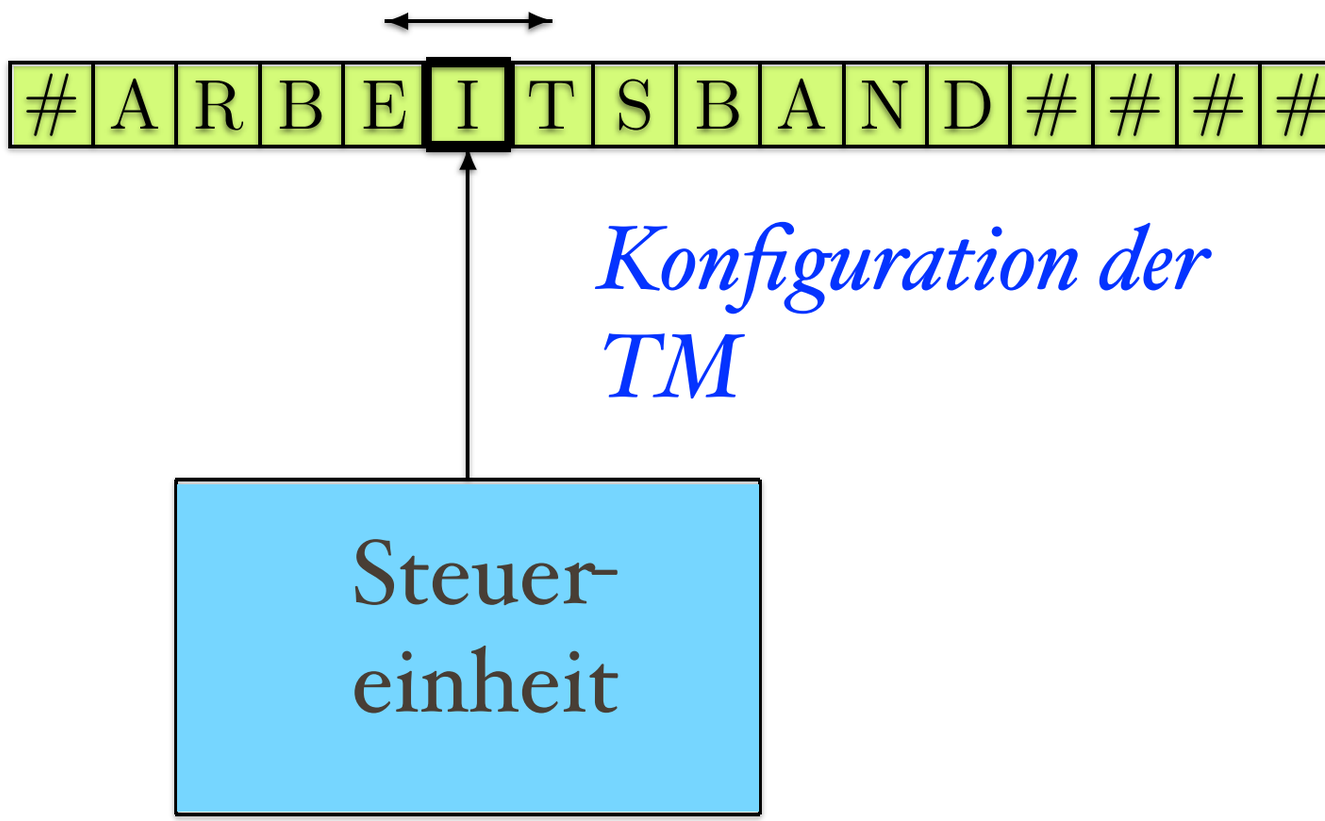




RAM_+
ohne MULT, DIV

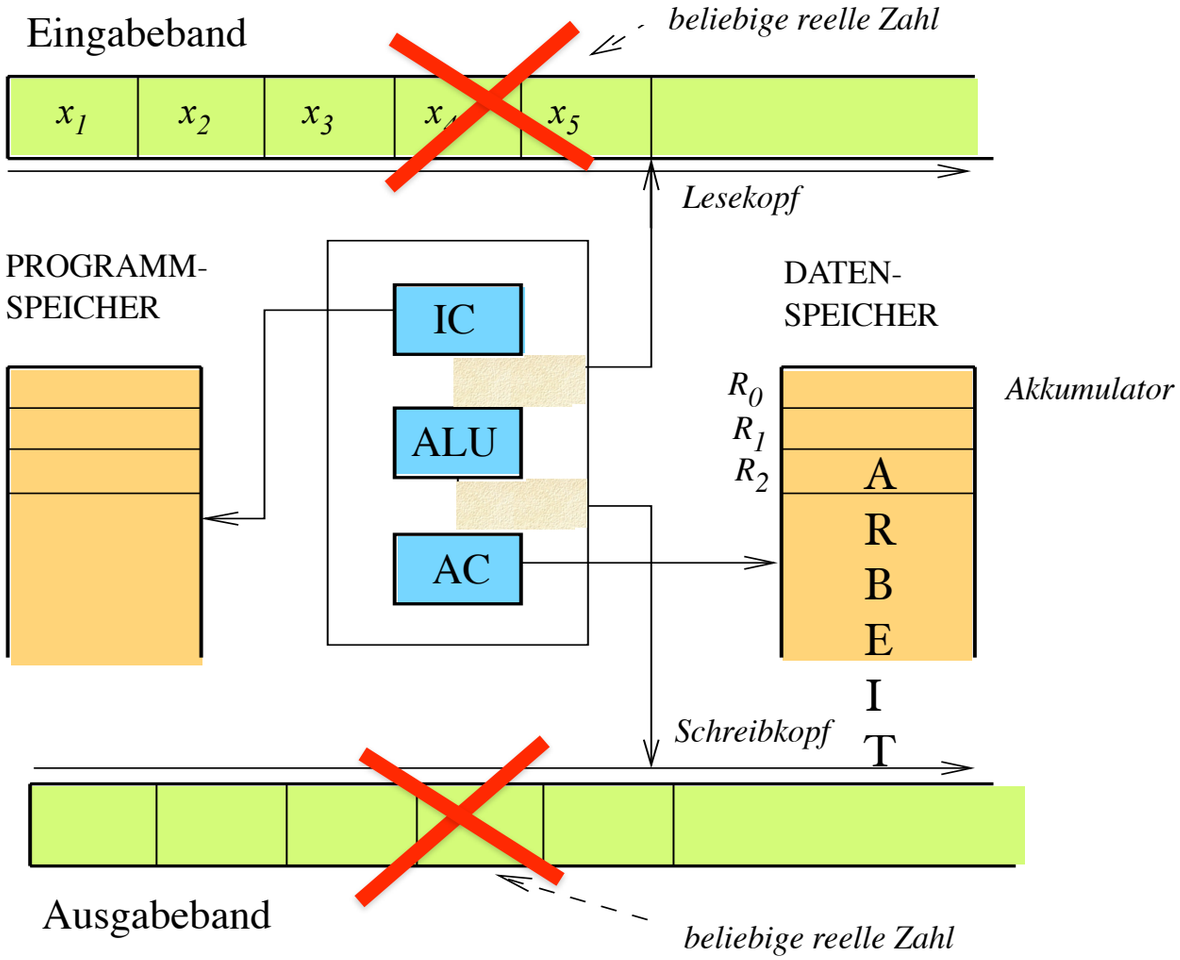
Satz 4.7 *Eine $T(n)$–zeitbeschränkte Mehrband DTM M kann durch eine RAM_+ simuliert werden, die im uniformen Maß $O(T(n))$-zeitbeschränkt ist und die im logarithmischen Maß $O(T(n)\log T(n))$-zeitbeschränkt ist.*

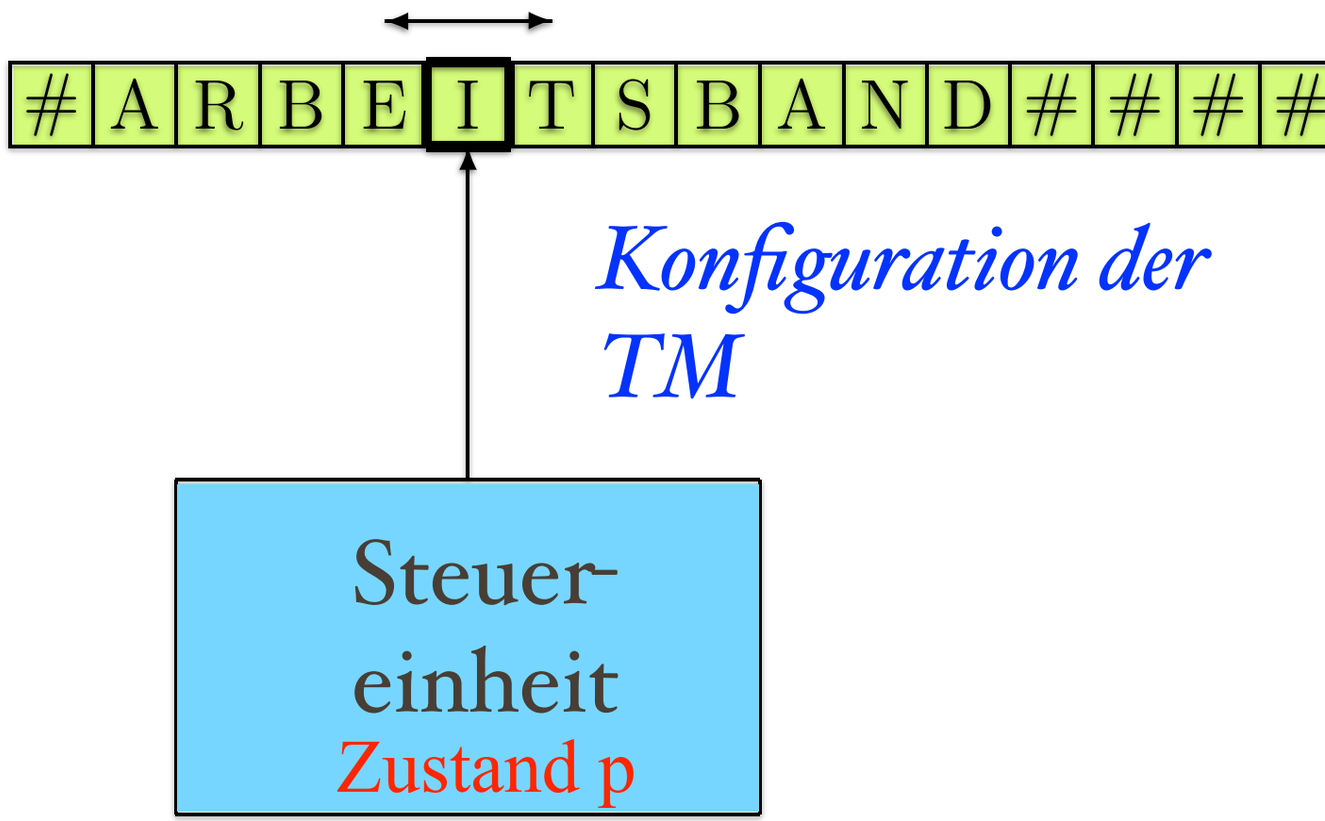




RAM_+
ohne MULT, DIV

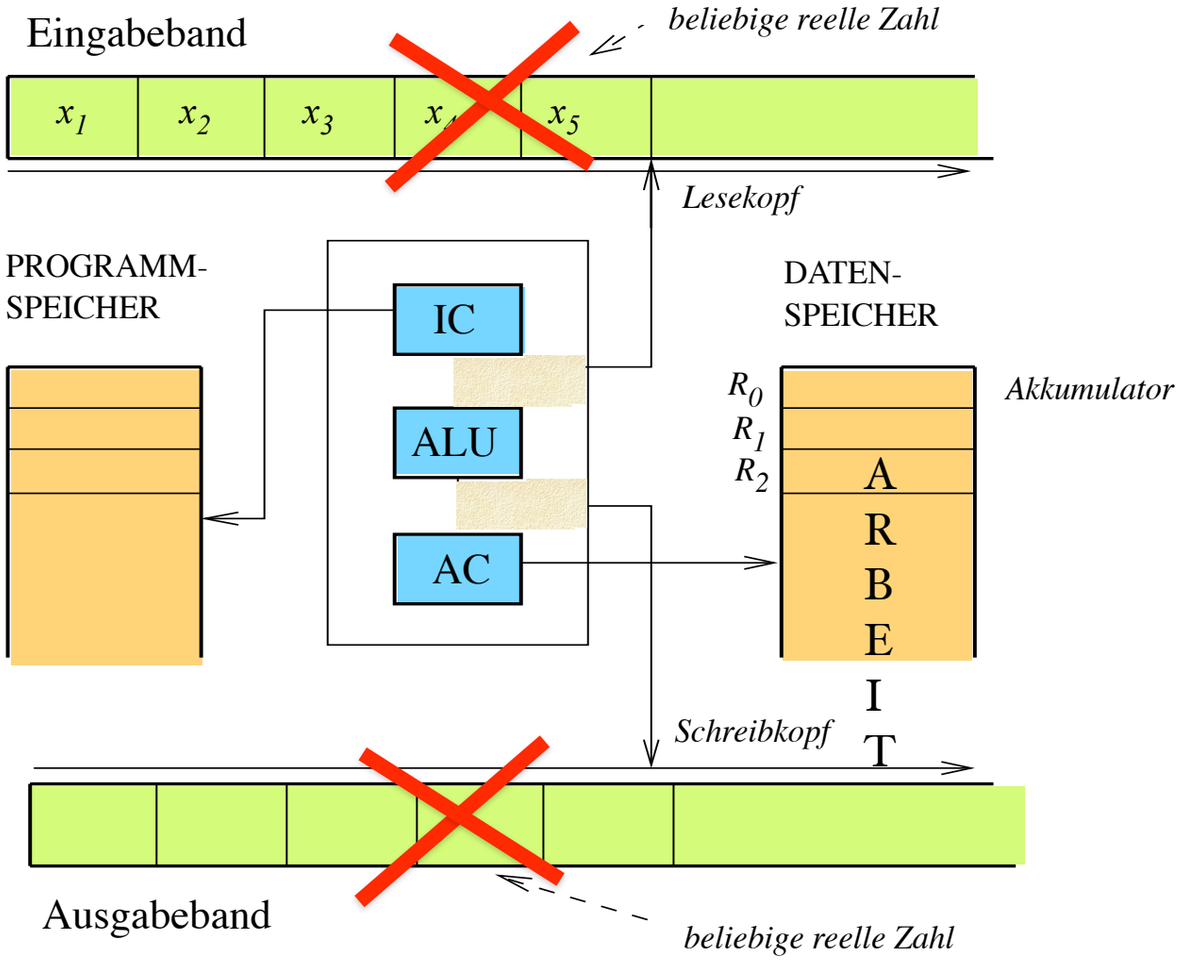
Satz 4.7 *Eine $T(n)$–zeitbeschränkte Mehrband DTM M kann durch eine RAM_+ simuliert werden, die im uniformen Maß $O(T(n))$-zeitbeschränkt ist und die im logarithmischen Maß $O(T(n)\log T(n))$-zeitbeschränkt ist.*



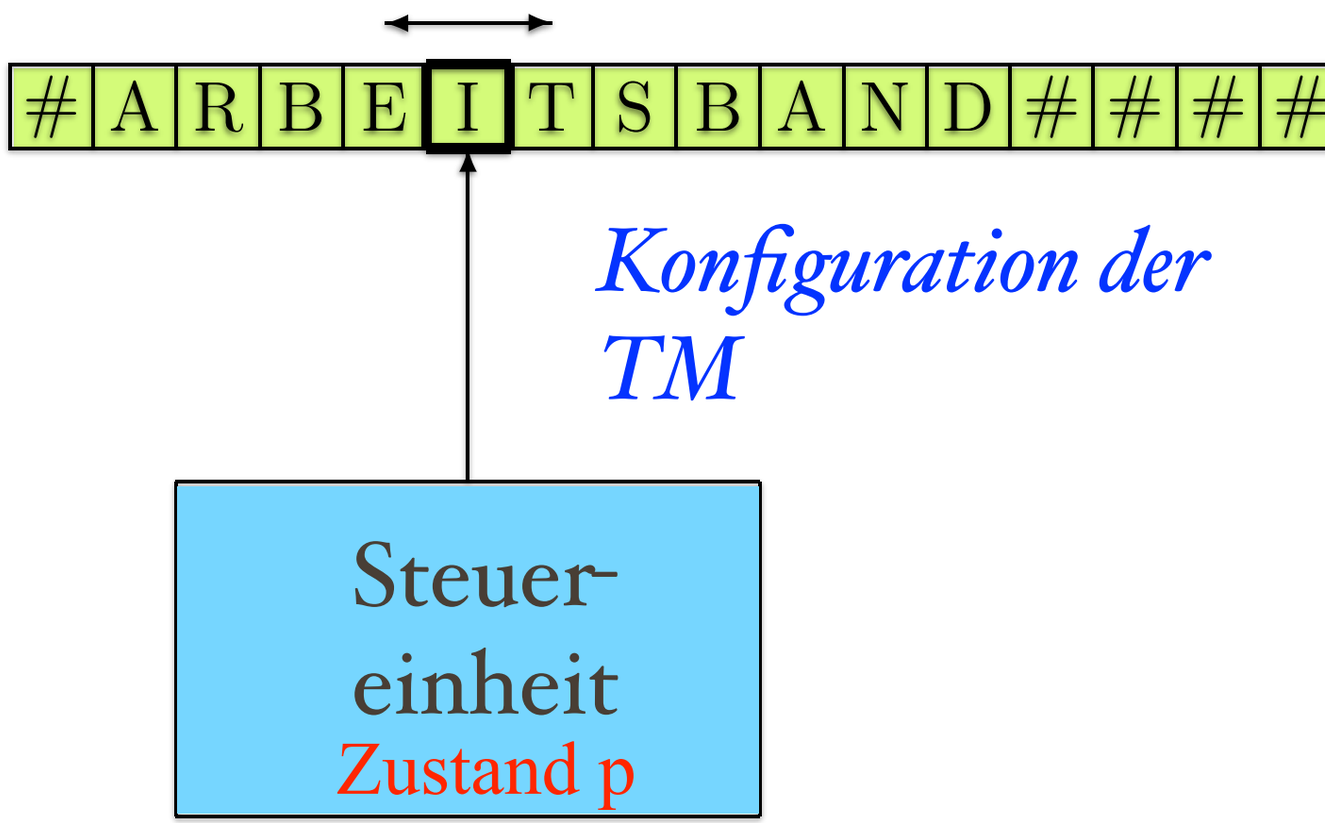


RAM_+
ohne MULT, DIV

Satz 4.7 *Eine $T(n)$–zeitbeschränkte Mehrband DTM M kann durch eine RAM_+ simuliert werden, die im uniformen Maß $O(T(n))$-zeitbeschränkt ist und die im logarithmischen Maß $O(T(n)\log T(n))$-zeitbeschränkt ist.*



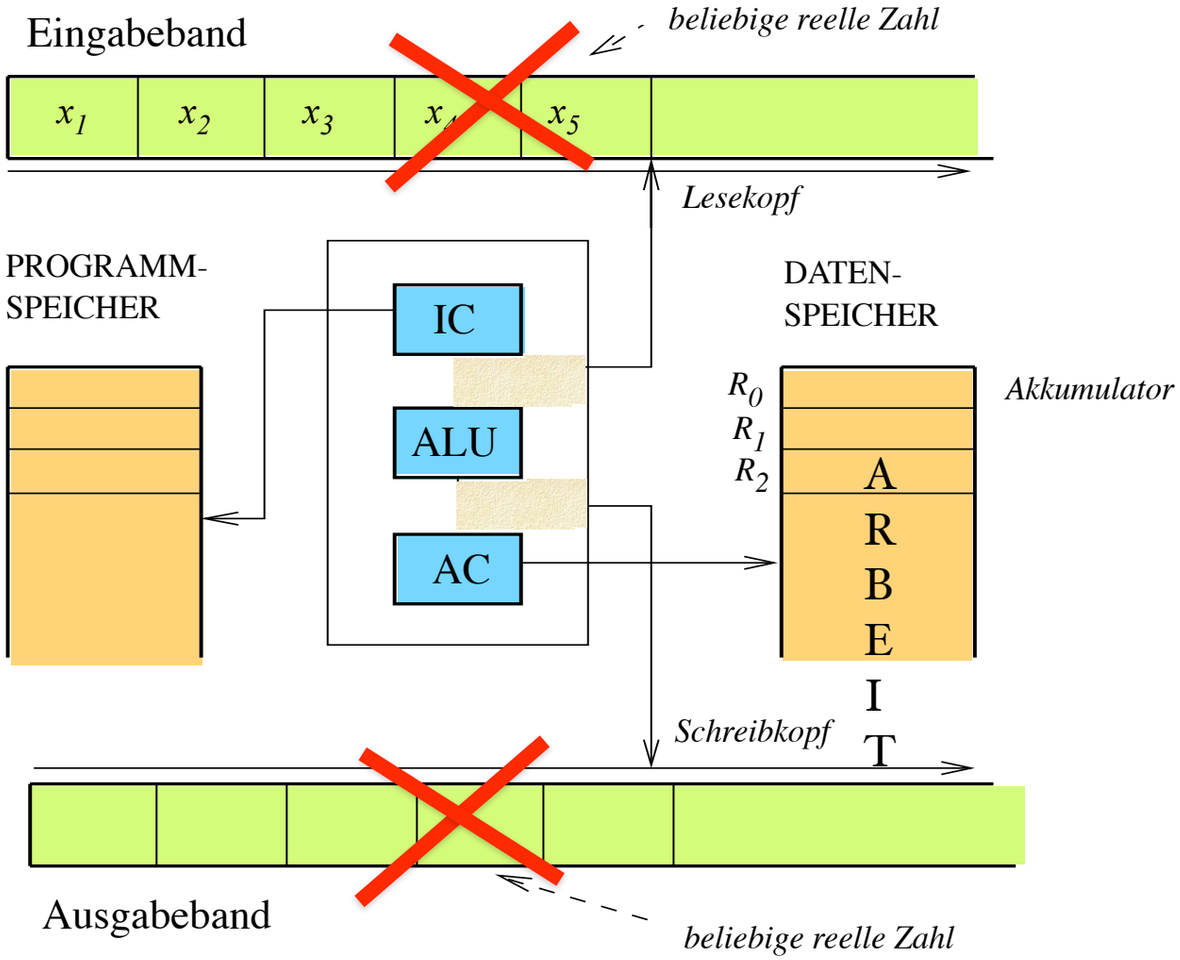
Position des Kopfes: j = 6

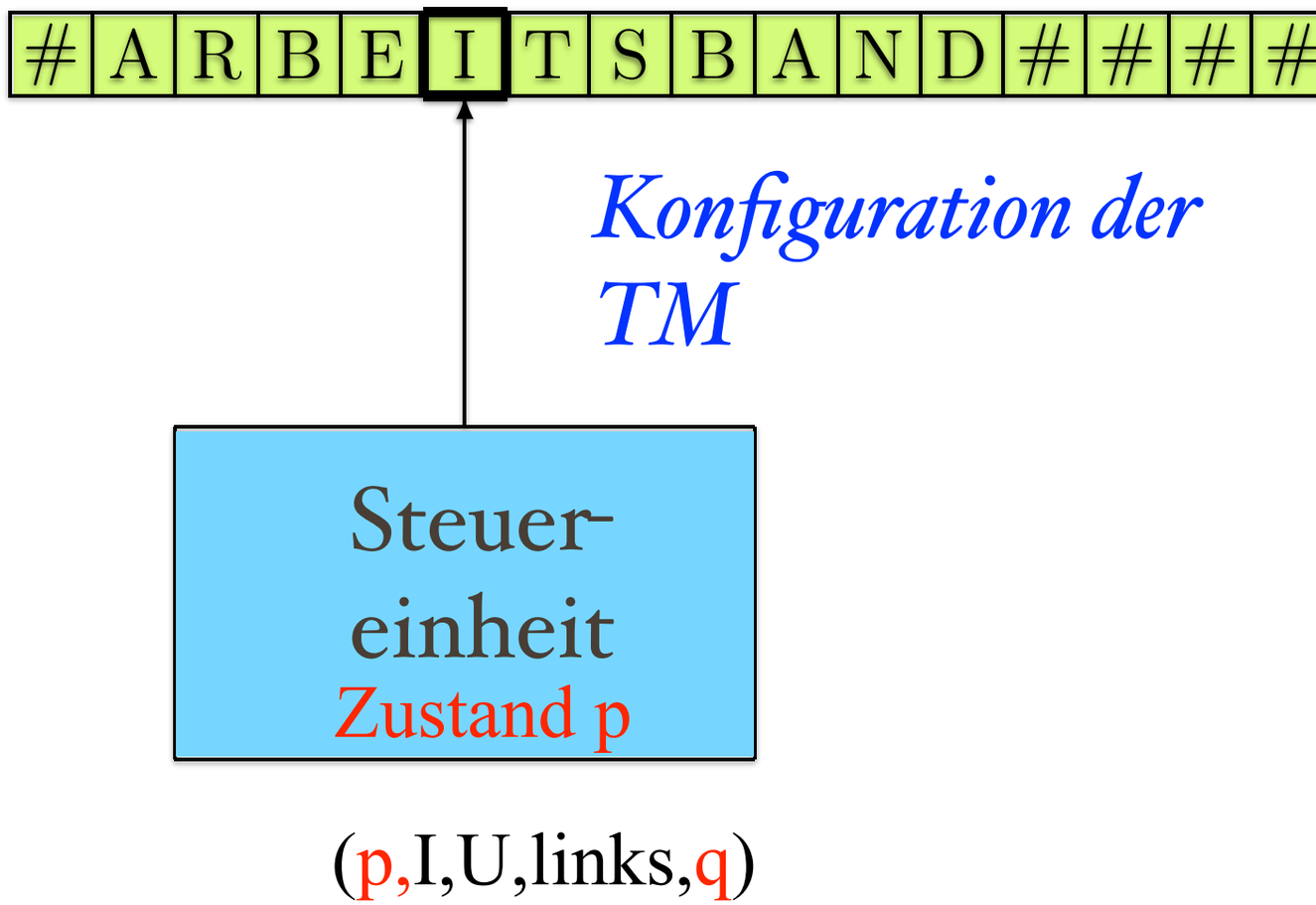


RAM+
ohne MULT, DIV

Satz 4.7 *Eine $T(n)$–zeitbeschränkte Mehrband DTM M kann durch eine RAM_+ simuliert werden, die im uniformen Maß $O(T(n))$-zeitbeschränkt ist und die im logarithmischen Maß $O(T(n)\log T(n))$-zeitbeschränkt ist.*

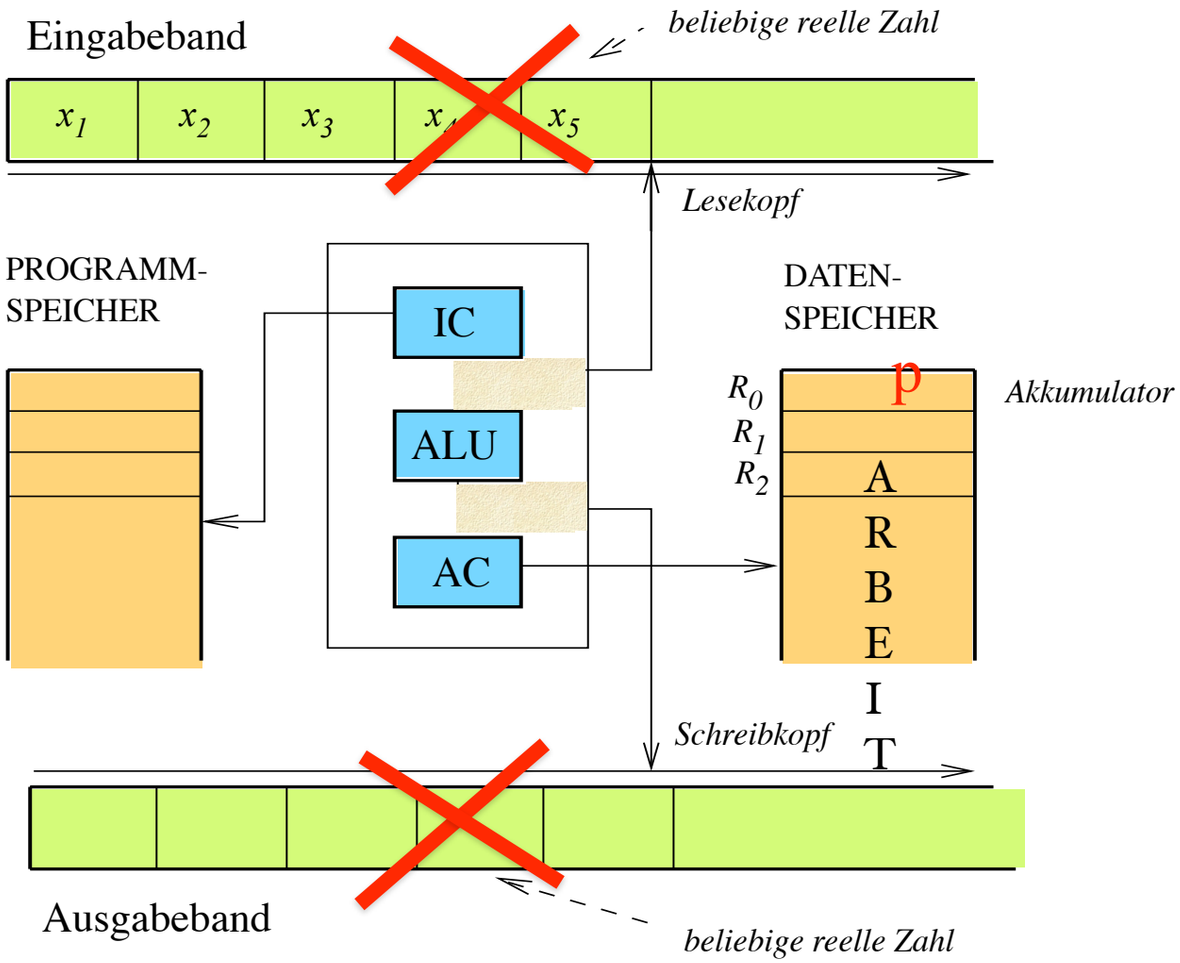
Position des Kopfes: j = 6



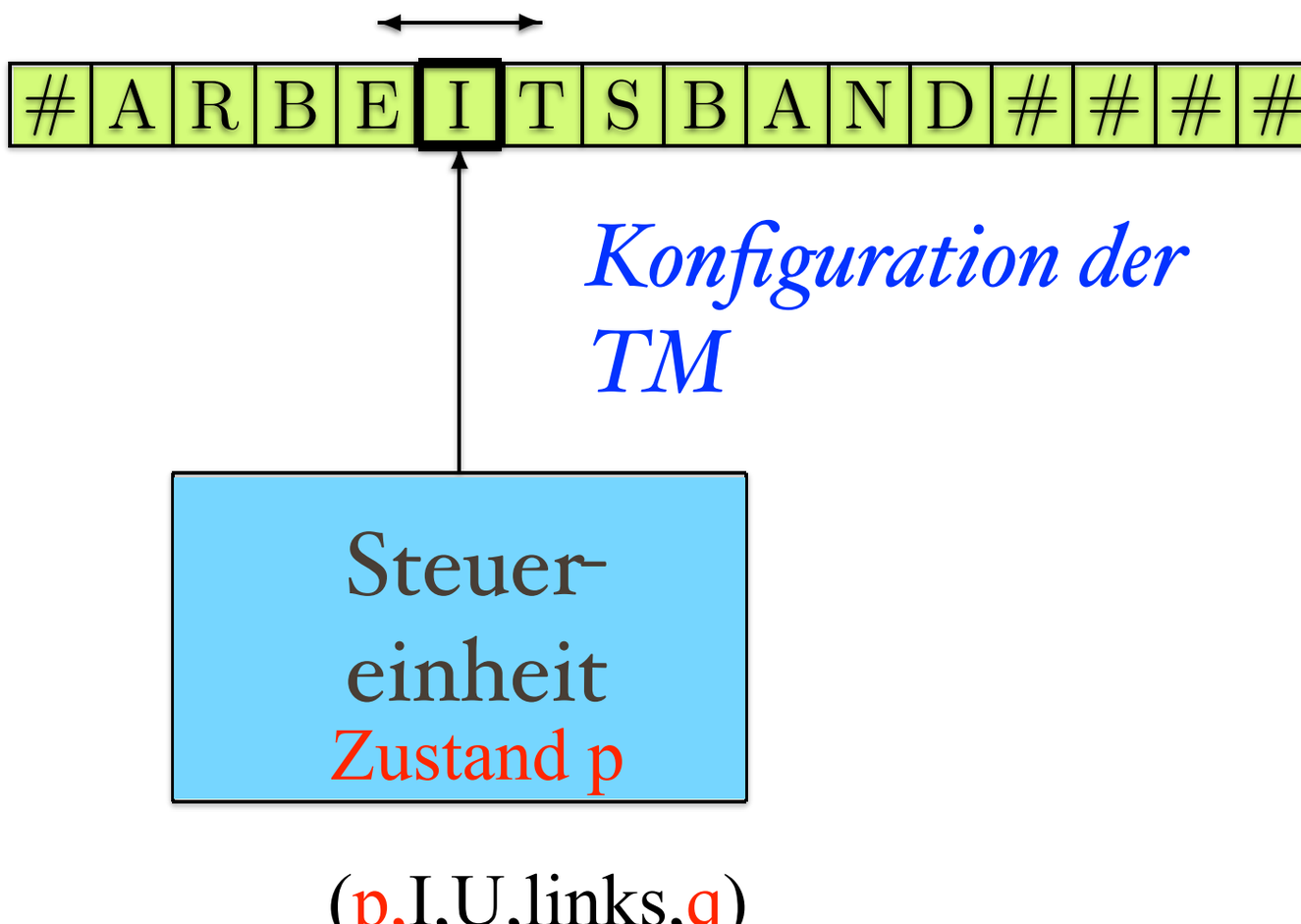


RAM_+
ohne MULT, DIV

Satz 4.7 *Eine $T(n)$–zeitbeschränkte Mehrband DTM M kann durch eine RAM_+ simuliert werden, die im uniformen Maß $O(T(n))$-zeitbeschränkt ist und die im logarithmischen Maß $O(T(n)\log T(n))$-zeitbeschränkt ist.*

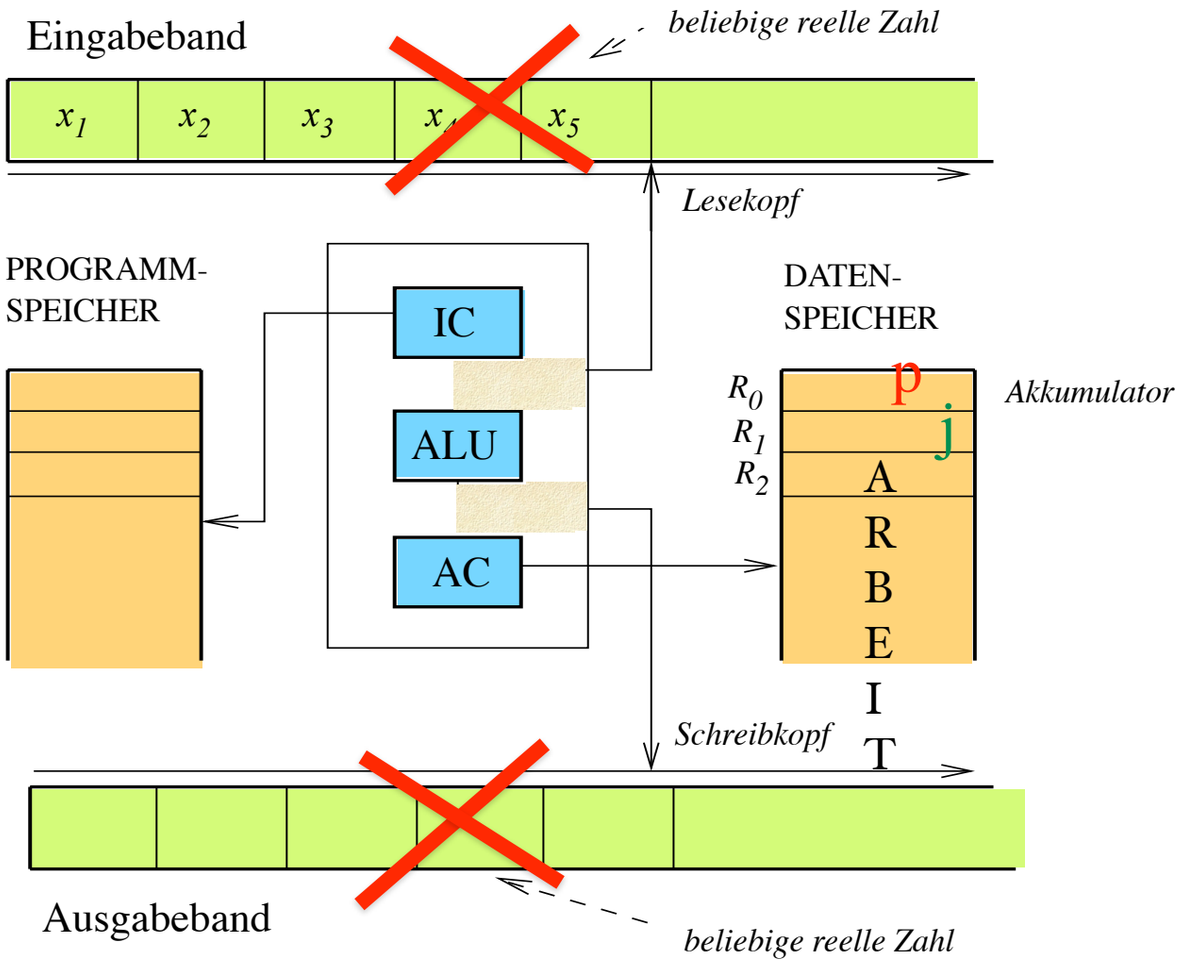


Position des Kopfes: j = 6

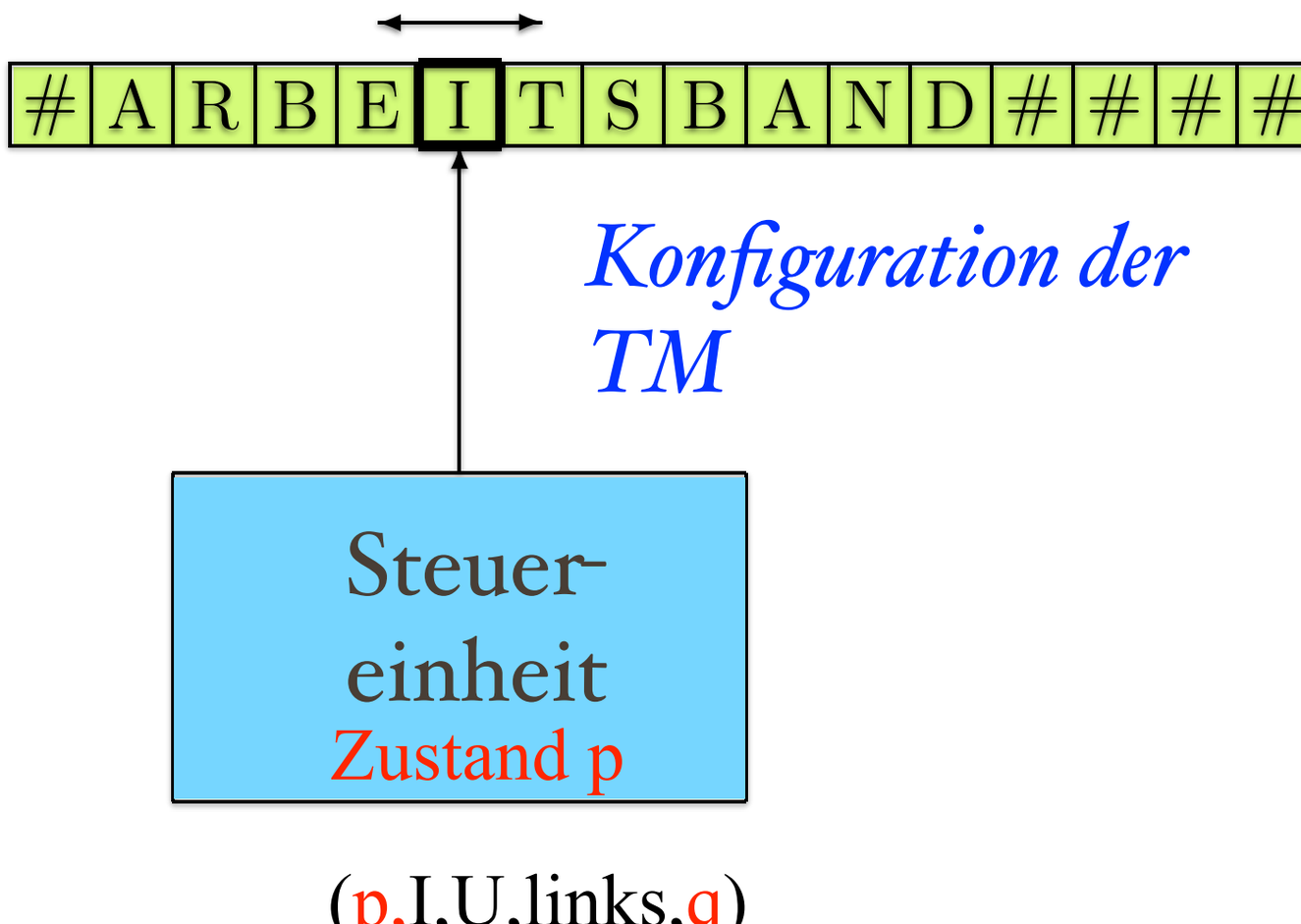


RAM_+
ohne MULT, DIV

Satz 4.7 *Eine $T(n)$–zeitbeschränkte Mehrband DTM M kann durch eine RAM_+ simuliert werden, die im uniformen Maß $O(T(n))$-zeitbeschränkt ist und die im logarithmischen Maß $O(T(n)\log T(n))$-zeitbeschränkt ist.*



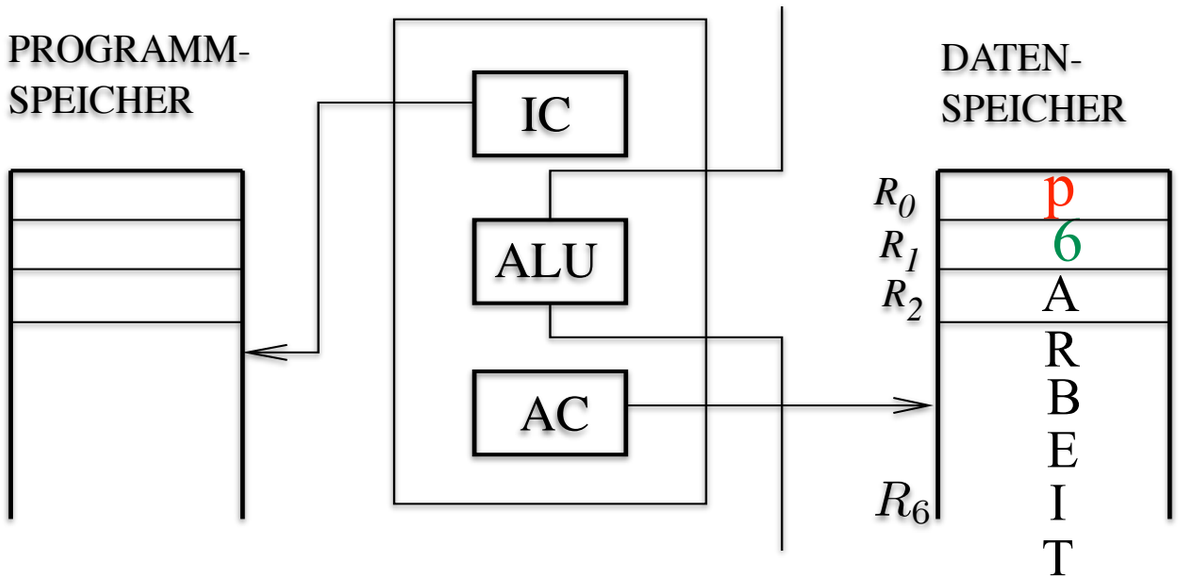
Position des Kopfes: j = 6



Konfiguration der TM

RAM_+
ohne MULT, DIV

Satz 4.7 *Eine $T(n)$–zeitbeschränkte Mehrband DTM M kann durch eine RAM_+ simuliert werden, die im uniformen Maß $O(T(n))$-zeitbeschränkt ist und die im logarithmischen Maß $O(T(n)\log T(n))$-zeitbeschränkt ist.*



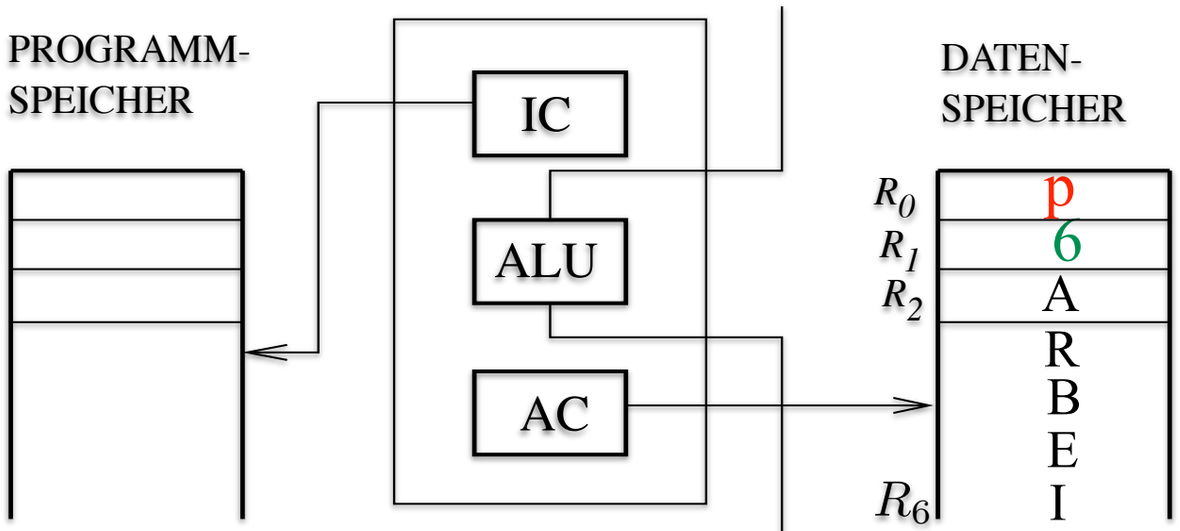
Position des Kopfes: j = 6

Steuer-
einheit
Zustand p

(p,I,U,links,q)

RAM_+
ohne MULT, DIV

Satz 4.7 *Eine $T(n)$–zeitbeschränkte Mehrband DTM M kann durch eine RAM_+ simuliert werden, die im uniformen Maß $O(T(n))$-zeitbeschränkt ist und die im logarithmischen Maß $O(T(n)\log T(n))$-zeitbeschränkt ist.*



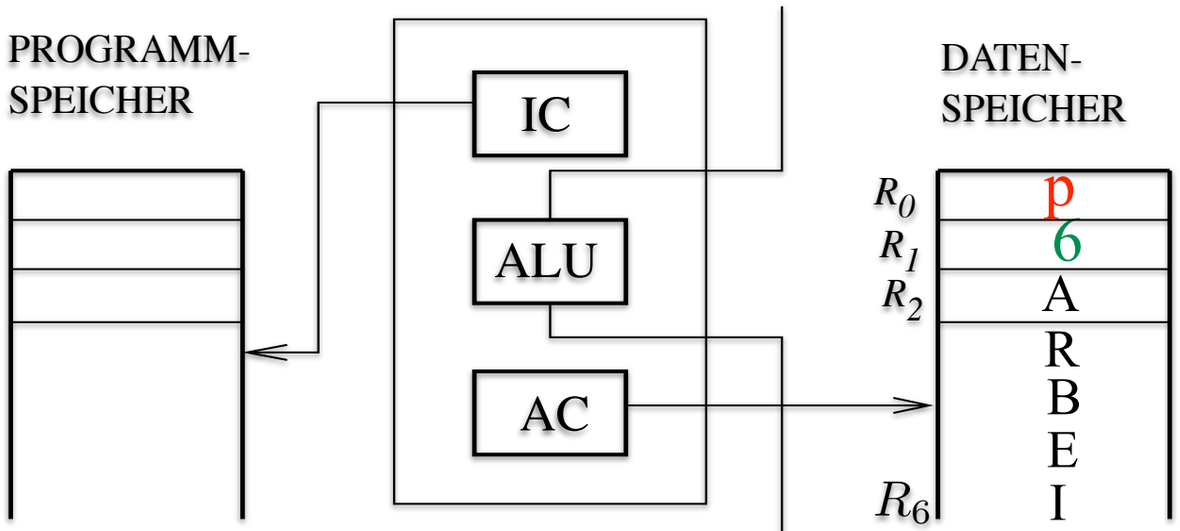

```
if r0 ='p'
  ...
  ...
fi
fi
```

Steuer-
einheit
Zustand p

(p,I,U,links,q)

RAM_+
ohne MULT, DIV

Satz 4.7 *Eine $T(n)$–zeitbeschränkte Mehrband DTM M kann durch eine RAM_+ simuliert werden, die im uniformen Maß $O(T(n))$-zeitbeschränkt ist und die im logarithmischen Maß $O(T(n)\log T(n))$-zeitbeschränkt ist.*



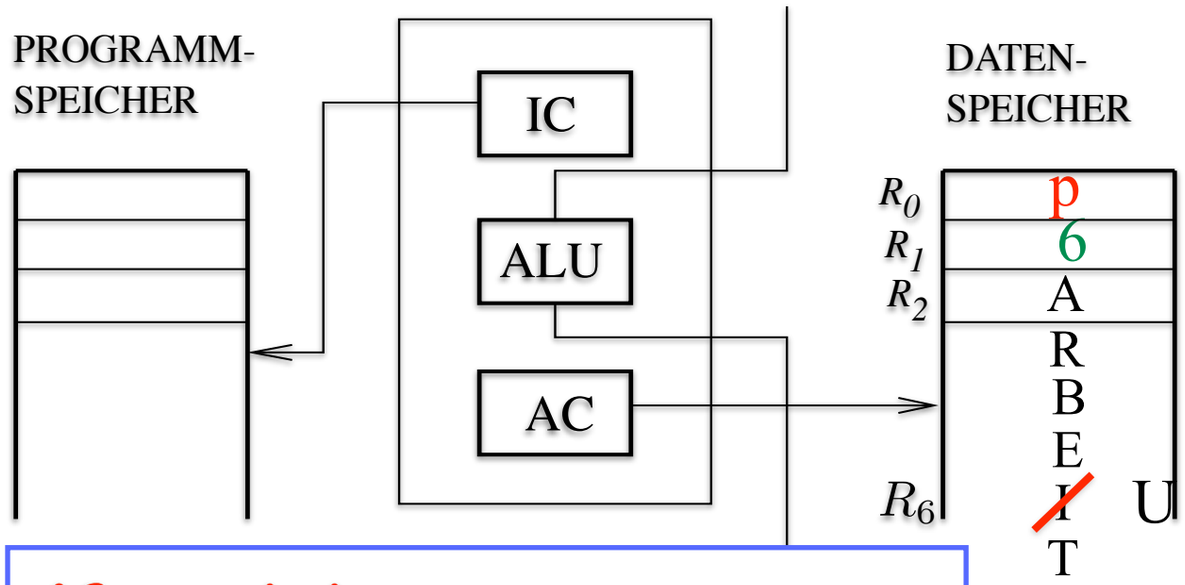

```
if r_0 ='p'
   ...
   if r_c(r_1) ='I'

                    fi
   ...
fi
```

Steuer-
einheit
Zustand p

(p,I,U,links,q)

RAM_+
ohne MULT, DIV

Satz 4.7 *Eine $T(n)$–zeitbeschränkte Mehrband DTM M kann durch eine RAM_+ simuliert werden, die im uniformen Maß $O(T(n))$-zeitbeschränkt ist und die im logarithmischen Maß $O(T(n)\log T(n))$-zeitbeschränkt ist.*



Position des Kopfes: j = 6

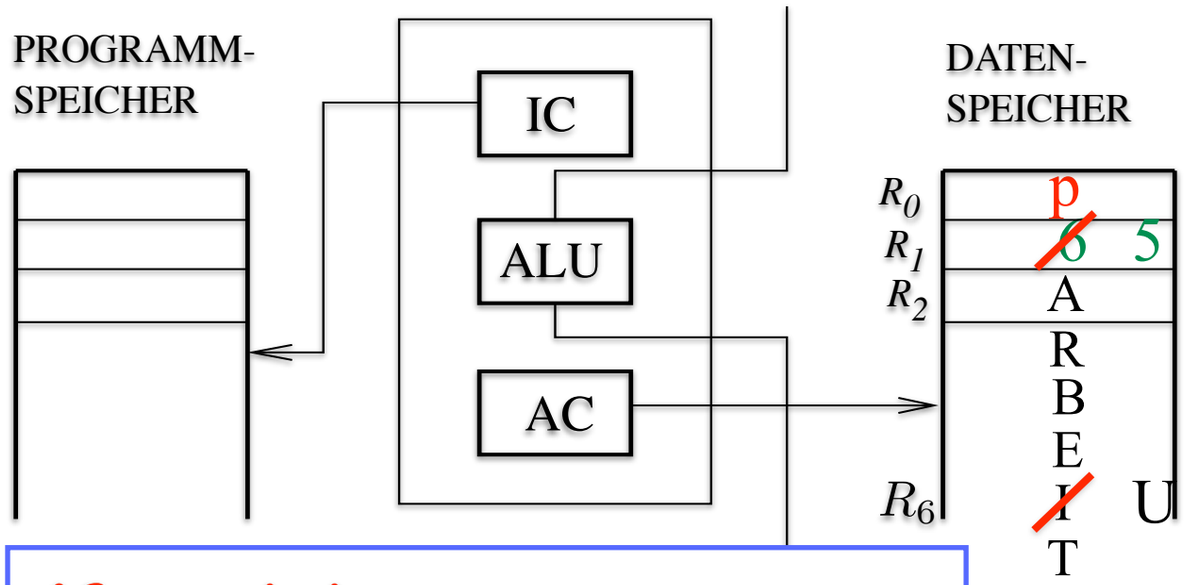
Steuer-
einheit
Zustand p

(p,I,U,links,q)

```
if r_0 = 'p'
   ...
   if r_c(r_1) = 'I'
    then r_c(r_1) ← 'U';
                                fi
   ...
fi
```

RAM_+
ohne MULT, DIV

Satz 4.7 *Eine $T(n)$–zeitbeschränkte Mehrband DTM M kann durch eine RAM_+ simuliert werden, die im uniformen Maß $O(T(n))$-zeitbeschränkt ist und die im logarithmischen Maß $O(T(n)\log T(n))$-zeitbeschränkt ist.*



Position des Kopfes: j = 6

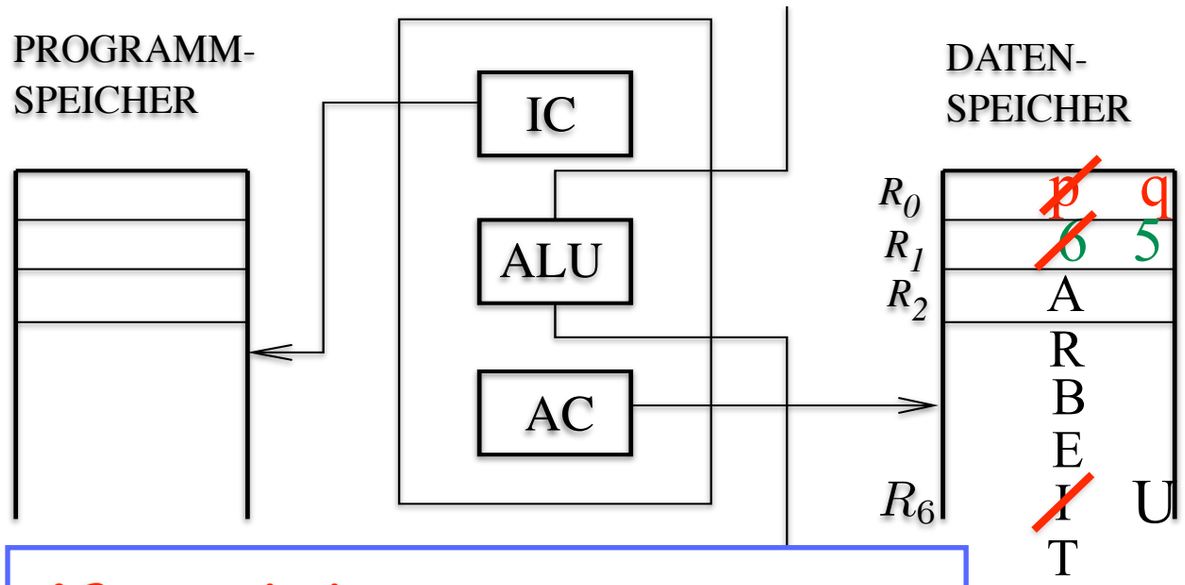

```
if r_0 ='p'
   ...
   if r_c(r_1) ='I'
    then r_c(r_1) ←'U';
                 r_1 ← r_1 − 1;
                              fi
   ...
fi
```

Steuer-
einheit
Zustand p

(p,I,U,links,q)

RAM_+
ohne MULT, DIV

Satz 4.7 *Eine $T(n)$–zeitbeschränkte Mehrband DTM M kann durch eine RAM_+ simuliert werden, die im uniformen Maß $O(T(n))$-zeitbeschränkt ist und die im logarithmischen Maß $O(T(n)\log T(n))$-zeitbeschränkt ist.*

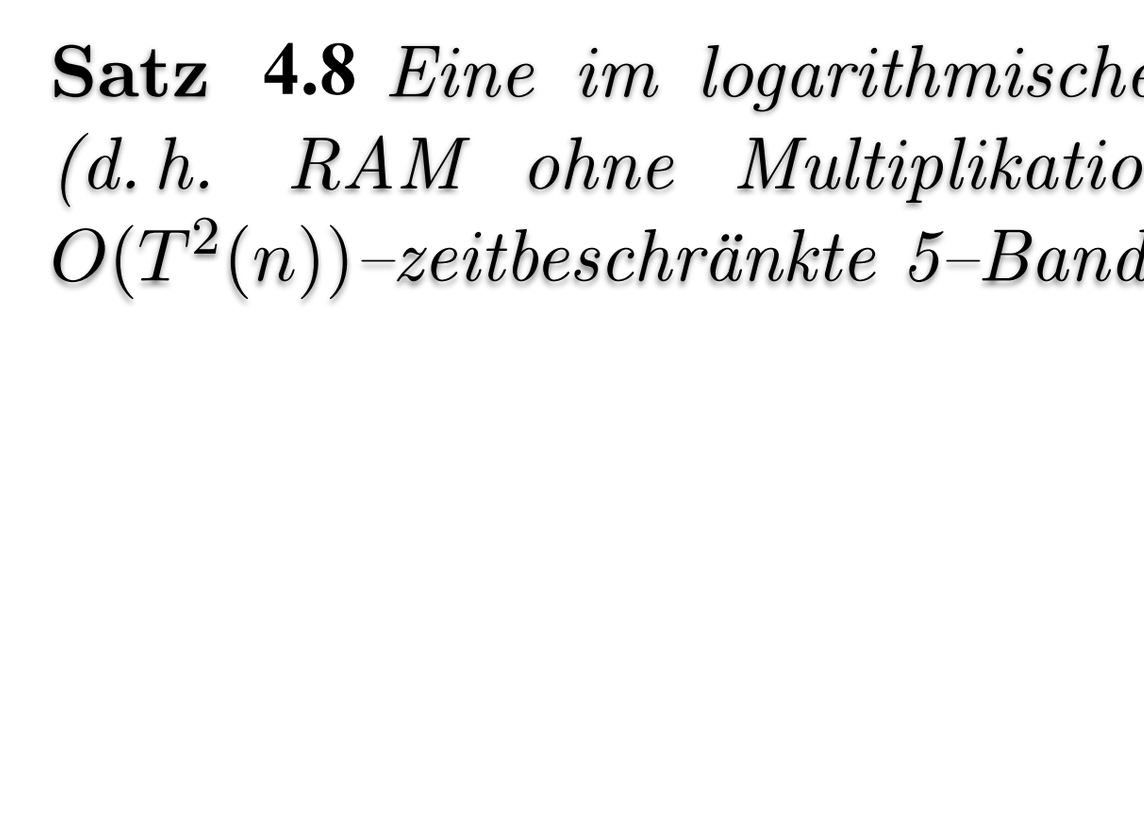


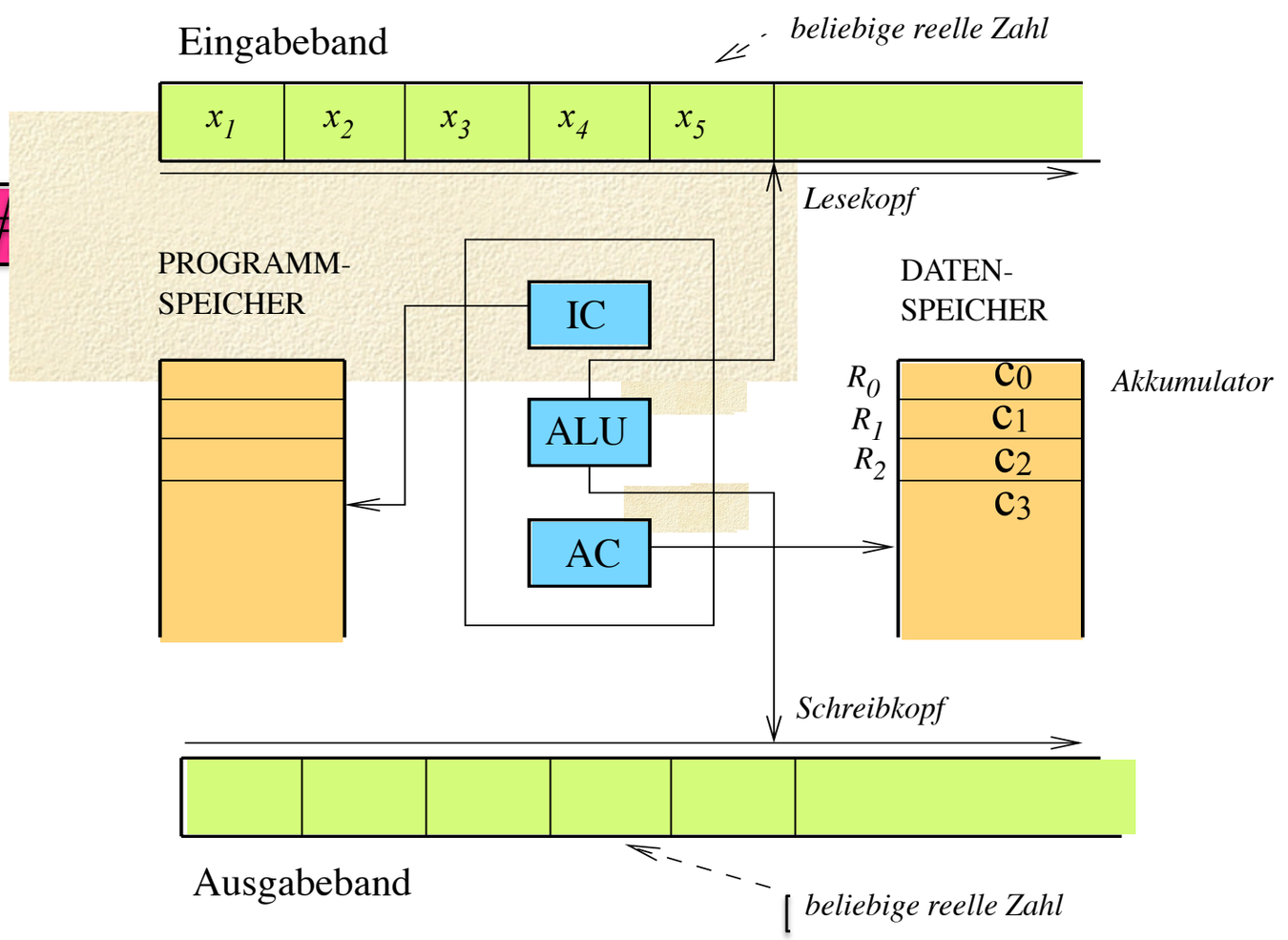

```
if r_0 ='p'
   ...
   if r_c(r_1) ='I'
    then r_c(r_1) ←'U';
                 r_1 ← r_1 − 1;
                  r_0 ←'q'  fi
   ...
fi
```

Steuer-
einheit
Zustand p

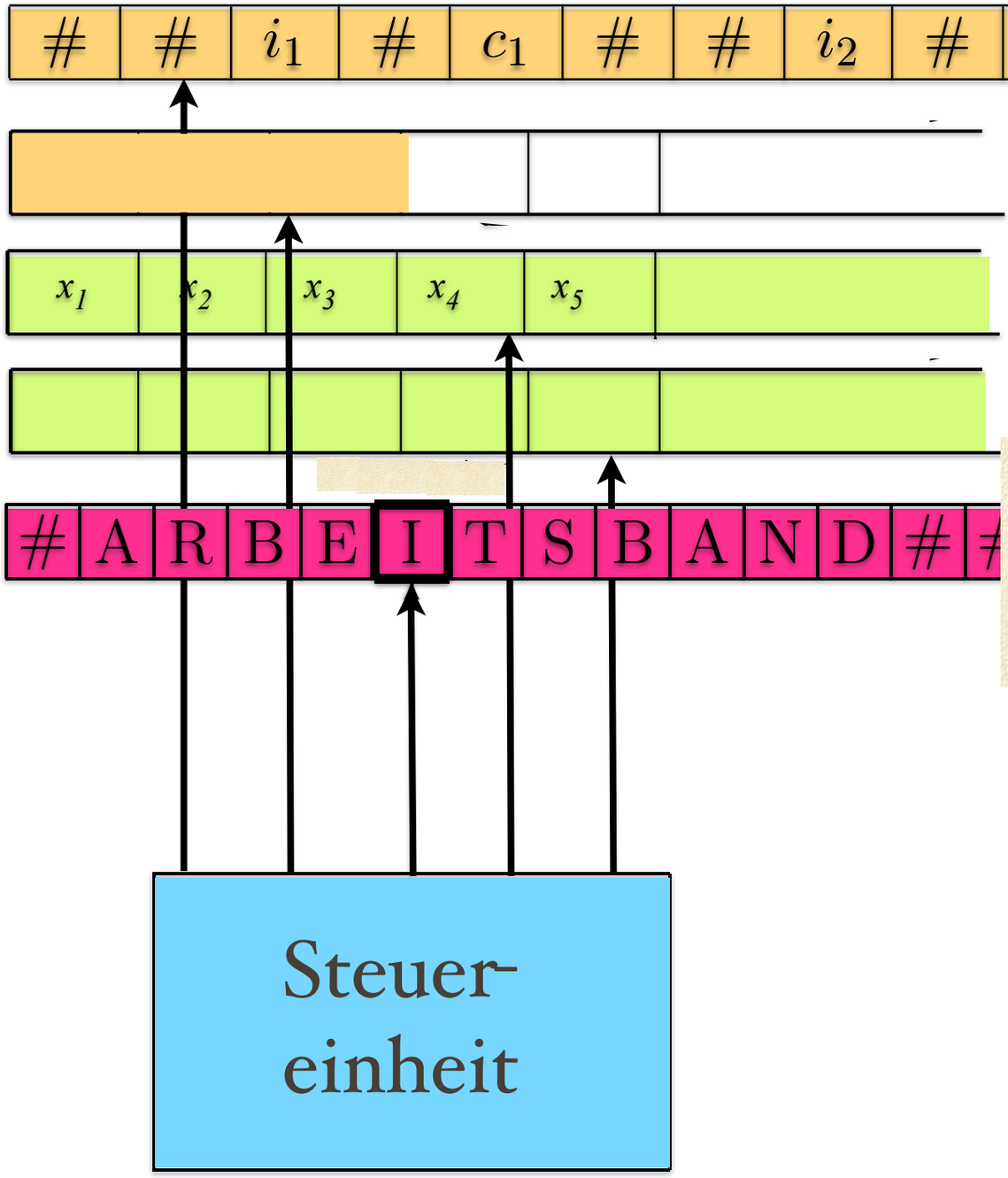
(p,I,U,links,q)

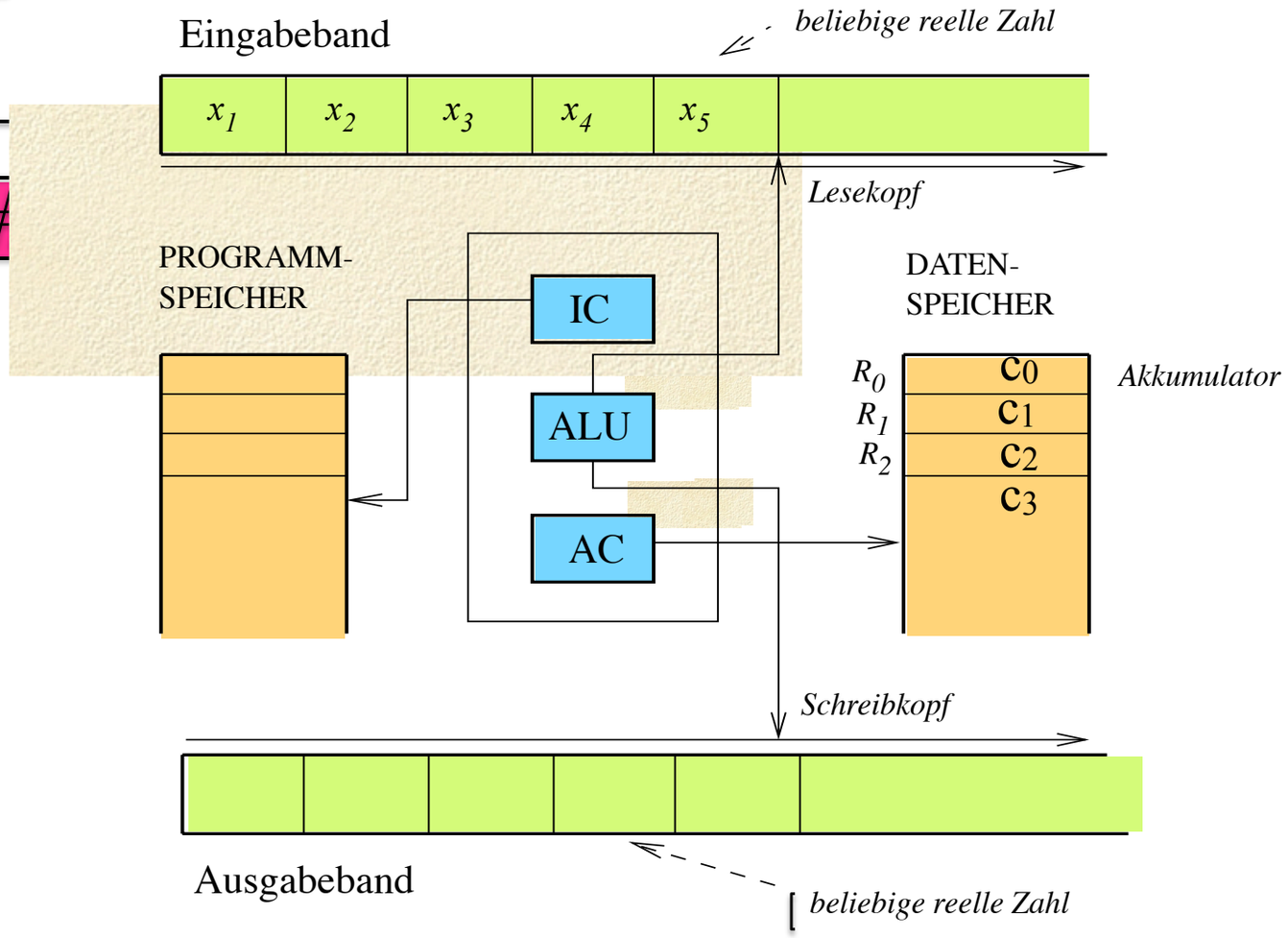
Satz 4.8 *Eine im logarithmischen Maß $T(n)$–zeitbeschränkte RAM_+ (d. h. RAM ohne Multiplikation und Division) kann durch eine $O(T^2(n))$–zeitbeschränkte 5–Band DTM M simuliert werden.*



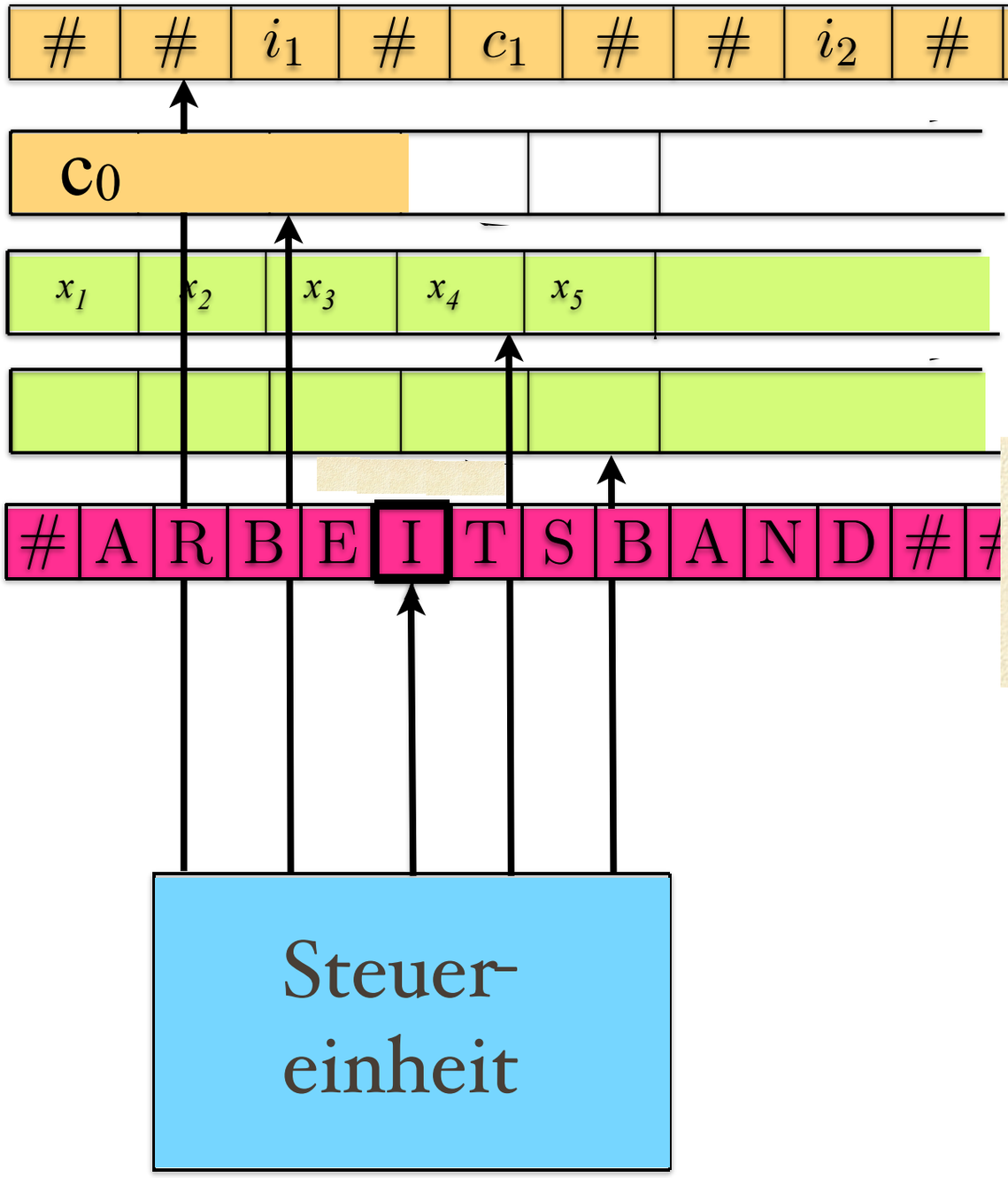


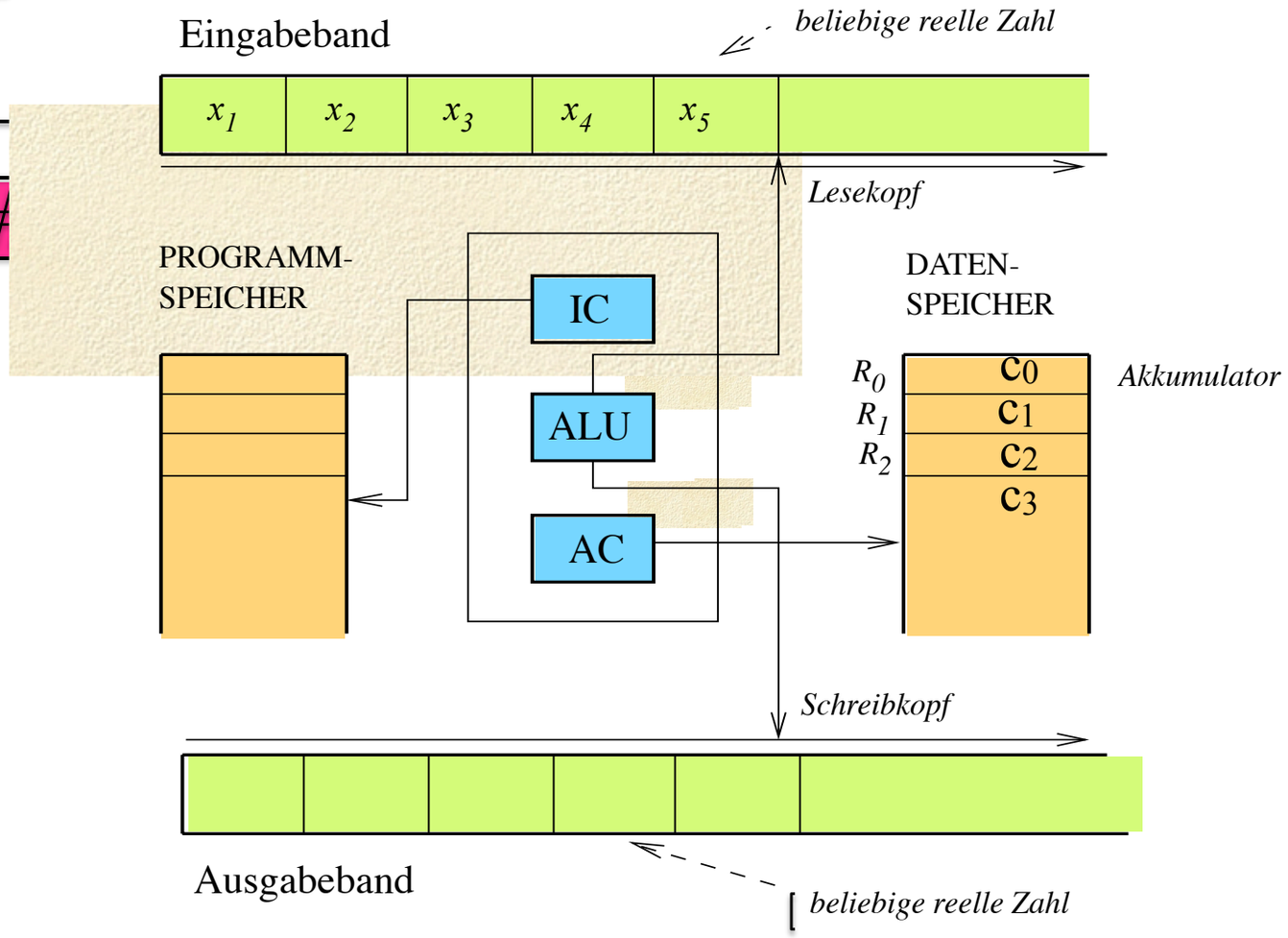
Satz 4.8 *Eine im logarithmischen Maß $T(n)$–zeitbeschränkte RAM_+ (d. h. RAM ohne Multiplikation und Division) kann durch eine $O(T^2(n))$–zeitbeschränkte 5–Band DTM M simuliert werden.*



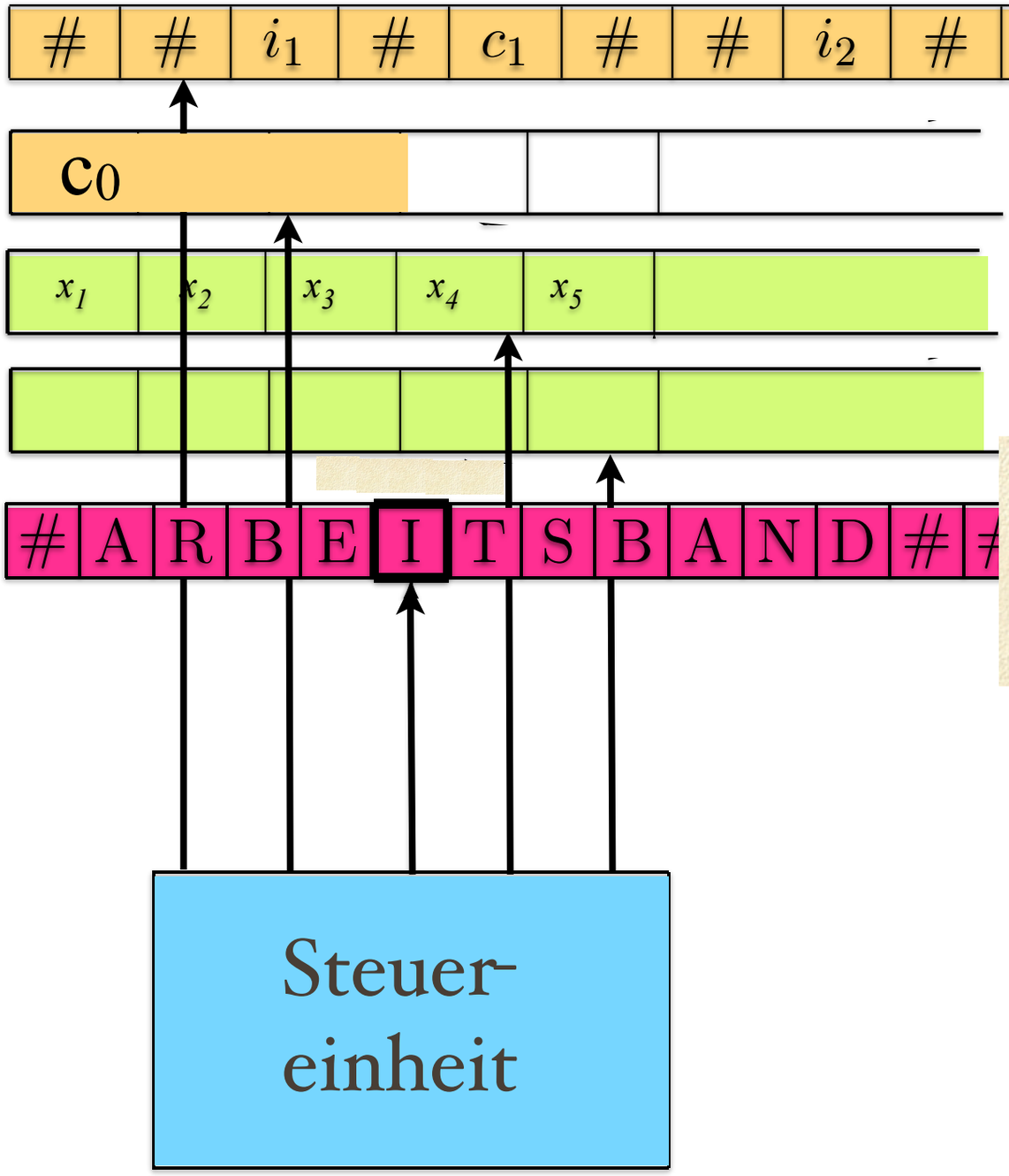


Satz 4.8 *Eine im logarithmischen Maß $T(n)$–zeitbeschränkte RAM_+ (d. h. RAM ohne Multiplikation und Division) kann durch eine $O(T^2(n))$–zeitbeschränkte 5–Band DTM M simuliert werden.*

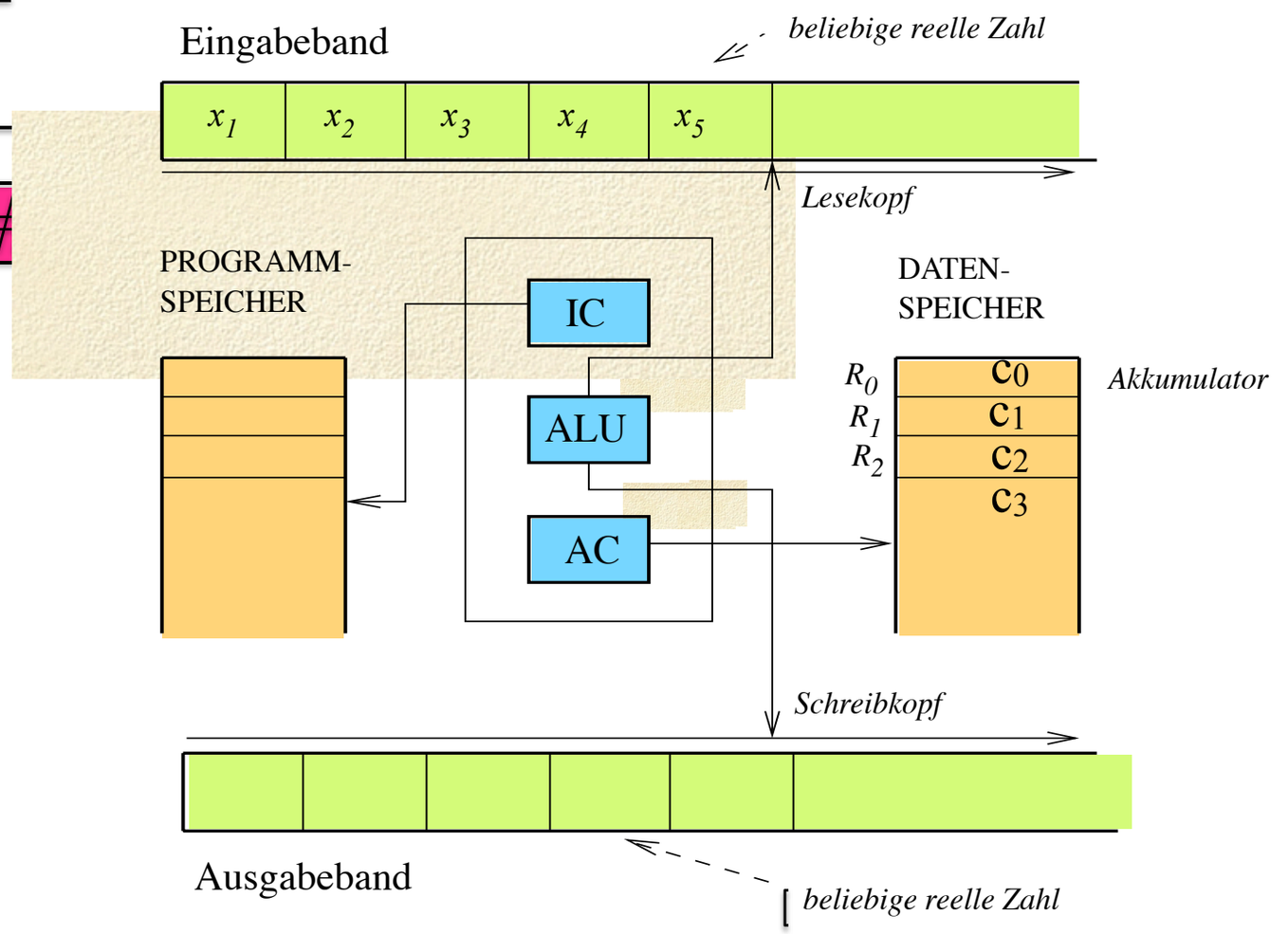




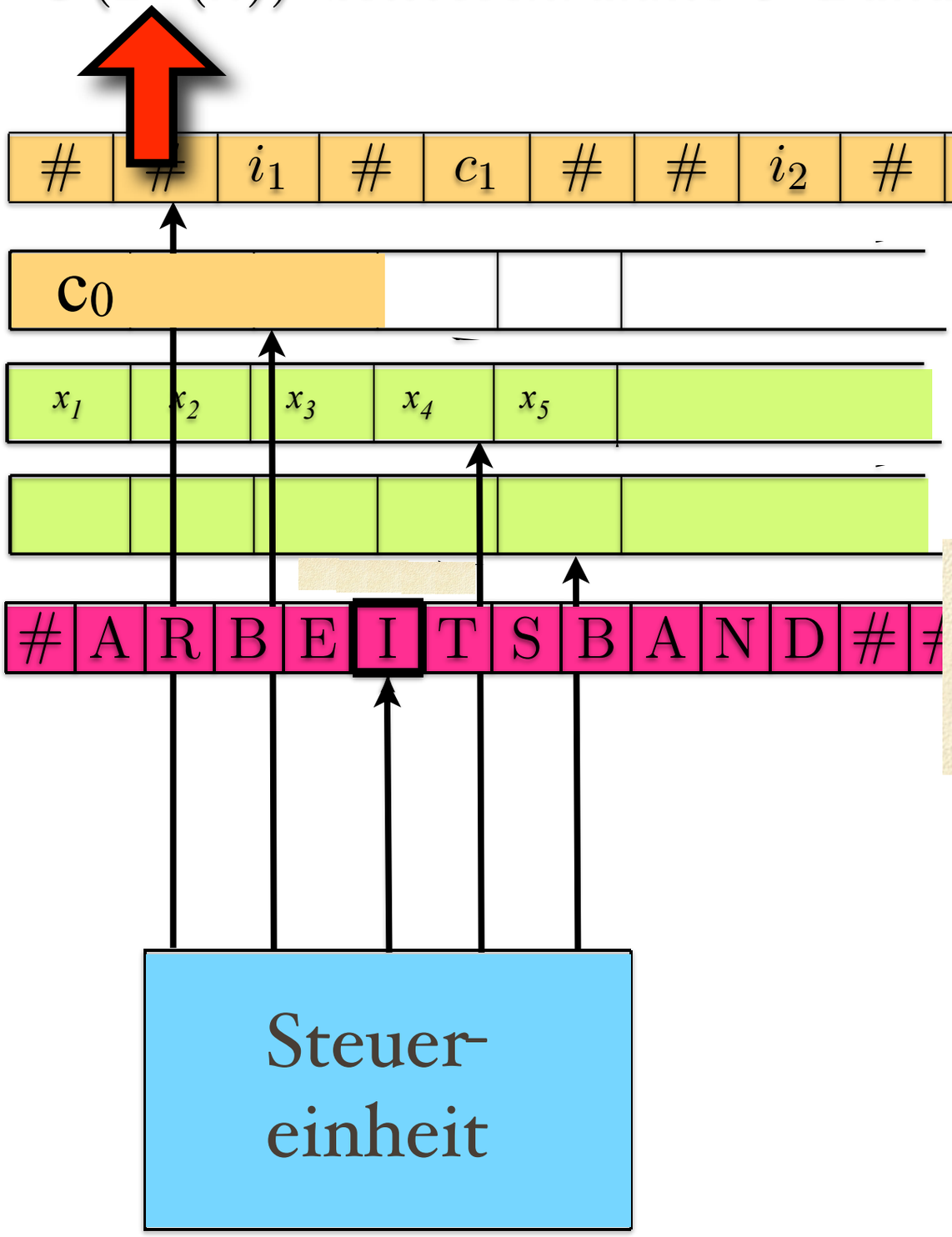
Satz 4.8 *Eine im logarithmischen Maß $T(n)$–zeitbeschränkte RAM_+ (d. h. RAM ohne Multiplikation und Division) kann durch eine $O(T^2(n))$–zeitbeschränkte 5–Band DTM M simuliert werden.*



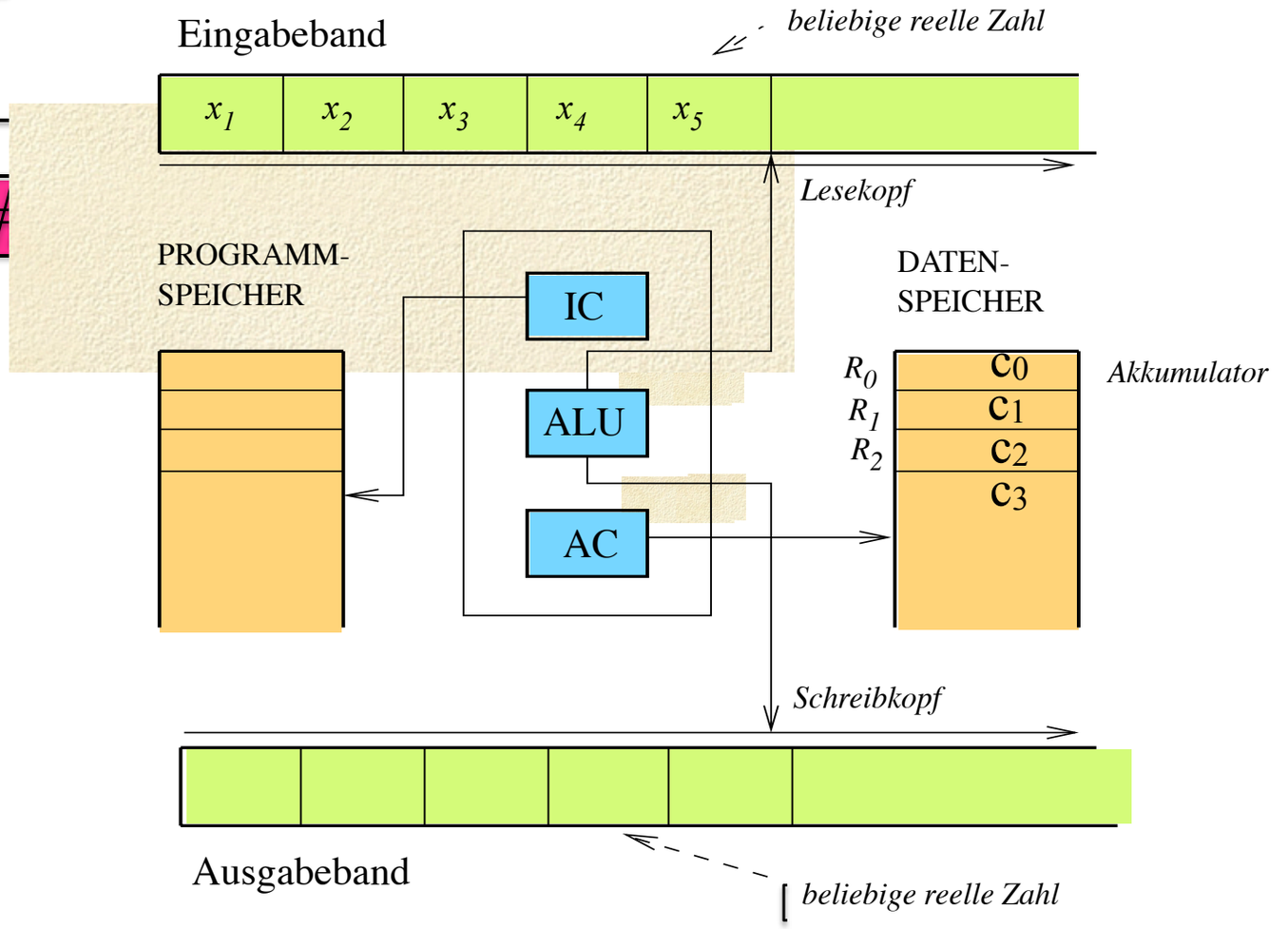
Simulation von ADD *25



Satz 4.8 *Eine im logarithmischen Maß $T(n)$–zeitbeschränkte RAM_+ (d. h. RAM ohne Multiplikation und Division) kann durch eine $O(T^2(n))$–zeitbeschränkte 5–Band DTM M simuliert werden.*

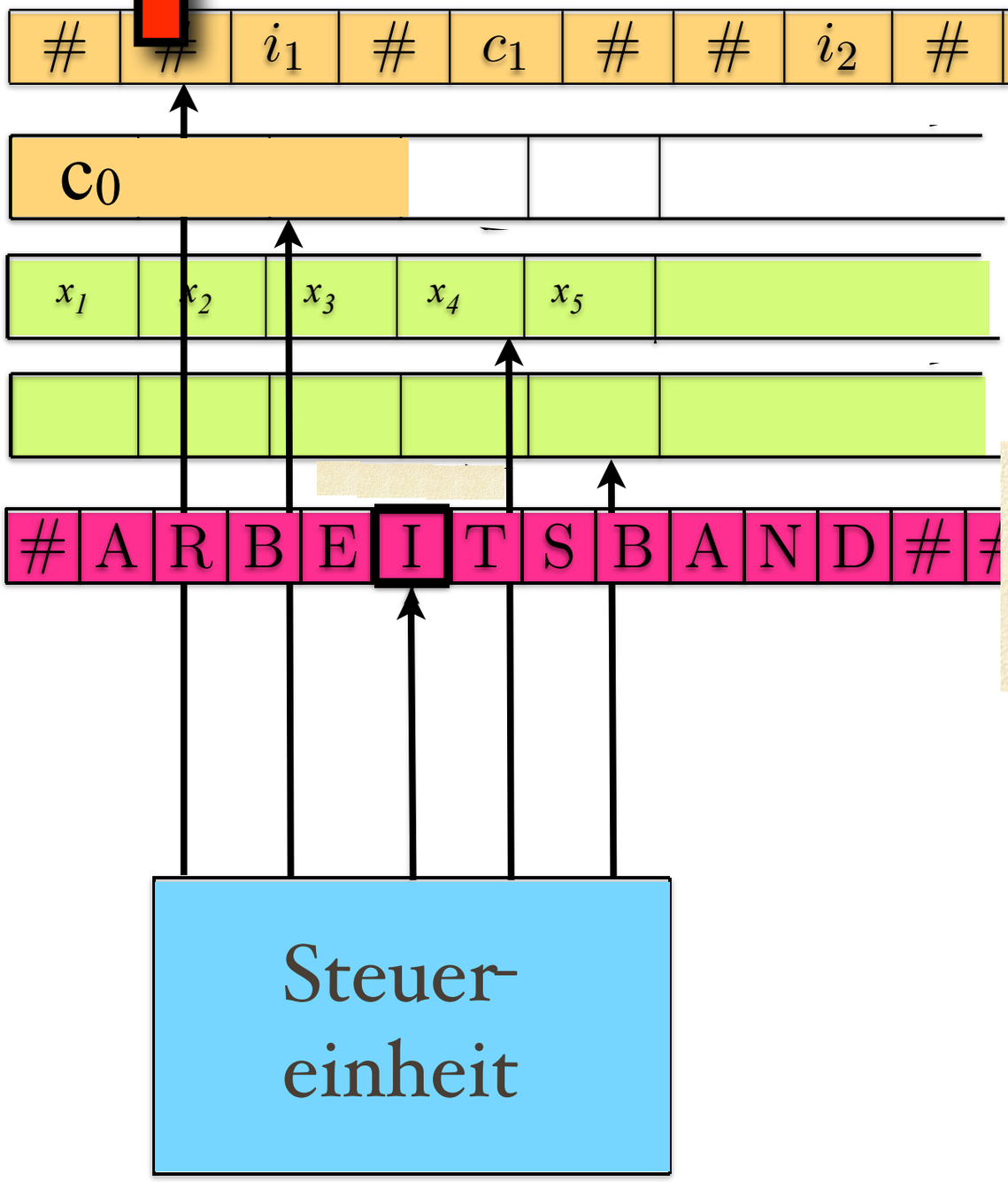


Simulation von ADD *25

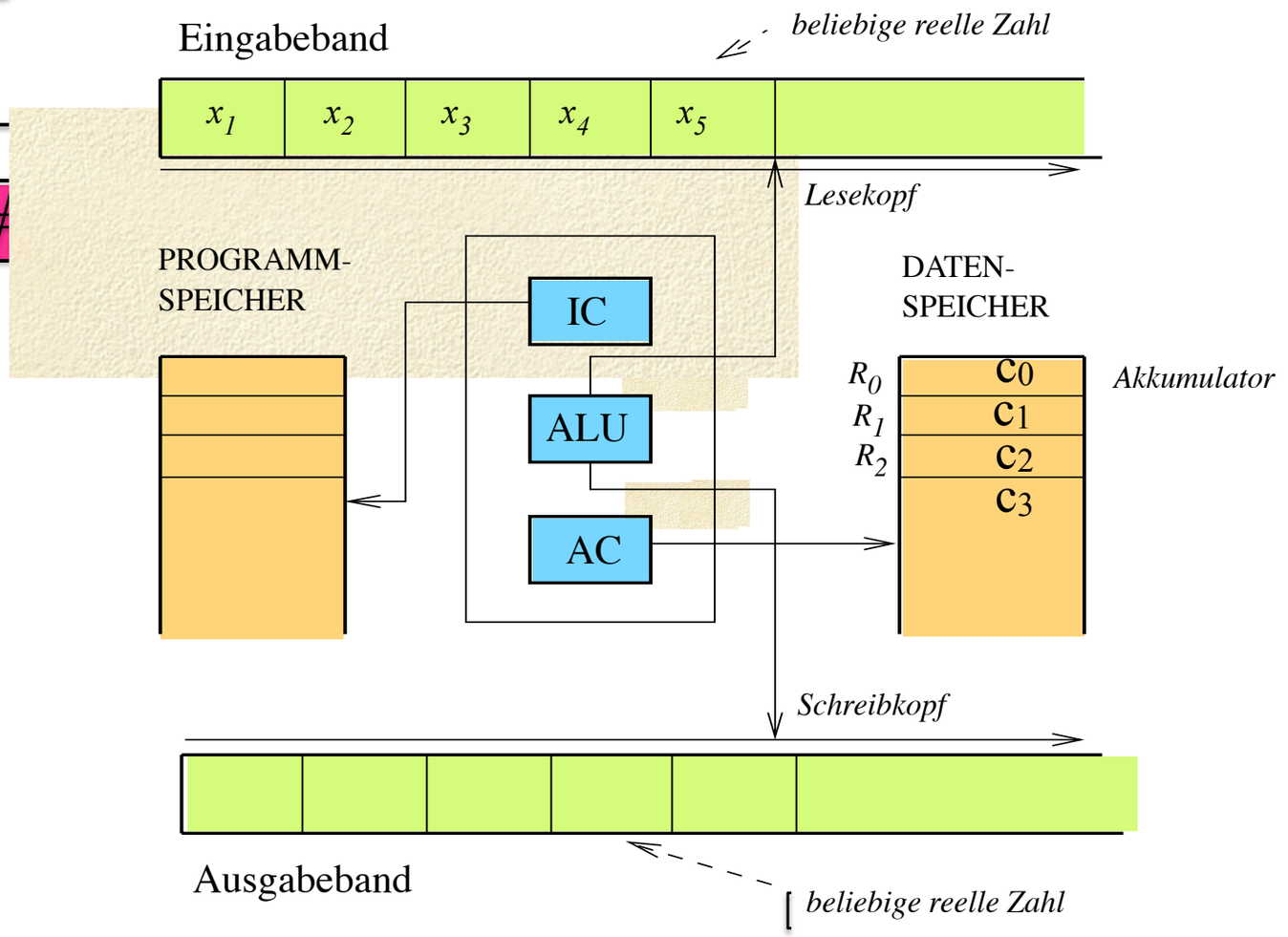


Satz 4.8 *Eine im logarithmischen Maß $T(n)$–zeitbeschränkte RAM_+ (d. h. RAM ohne Multiplikation und Division) kann durch eine $O(T^2(n))$–zeitbeschränkte 5–Band DTM M simuliert werden.*

$$T(n) \times S(n)$$

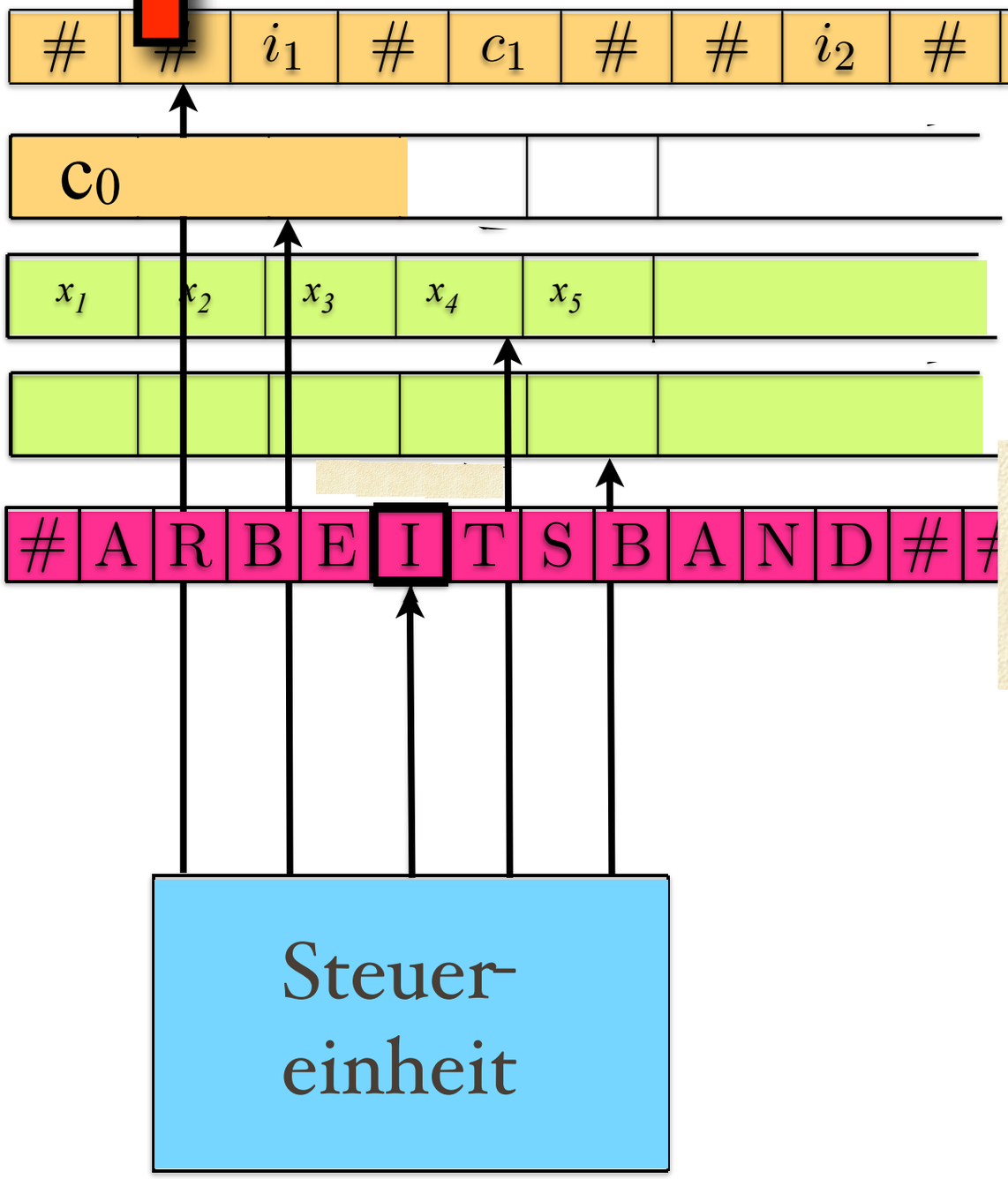


Simulation von ADD *25

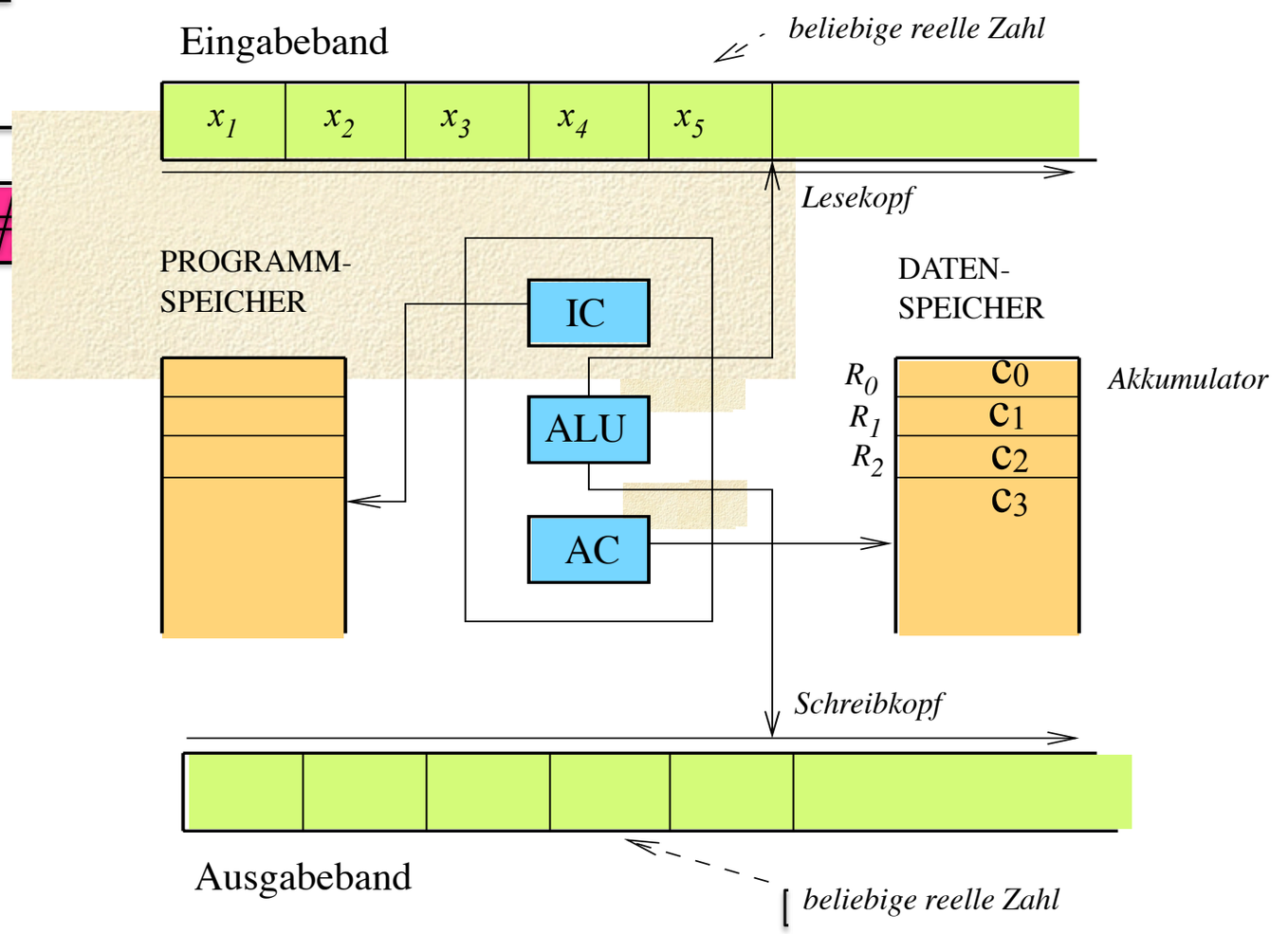


Satz 4.8 *Eine im logarithmischen Maß $T(n)$–zeitbeschränkte RAM_+ (d. h. RAM ohne Multiplikation und Division) kann durch eine $O(T^2(n))$–zeitbeschränkte 5–Band DTM M simuliert werden.*

$$T(n) \times S(n) \leq T(n) \times T(n)$$



Simulation von ADD *25



"Random access stored program machine"

Invariance Thesis: *There exists a standard class of machine models, which includes among others all variants of Turing Machines, all variants of RAM's ~~and RASP's~~ with logarithmic time measures, and also the RAM's ~~and RASP's~~ in the uniform time measure and logarithmic space measure, provided only standard arithmetical instructions of additive type are used. Machine models in this class simulate each other with polynomically bounded overhead in time, and constant factor overhead in space.*

"Random access stored program machine"

Invariance Thesis: *There exists a standard class of machine models, which includes among others all variants of Turing Machines, all variants of RAM's ~~and RASP's~~ with logarithmic time measures, and also the RAM's ~~and RASP's~~ in the uniform time measure and logarithmic space measure, provided only standard arithmetical instructions of additive type are used. Machine models in this class simulate each other with polynomically bounded overhead in time, and constant factor overhead in space.*

die 1. Maschinenklasse

"Random access stored program machine"

Invariance Thesis: *There exists a standard class of machine models, which includes among others all variants of Turing Machines, all variants of RAM's ~~and RASP's~~ with logarithmic time measures, and also the RAM's ~~and RASP's~~ in the uniform time measure and logarithmic space measure, provided only standard arithmetical instructions of additive type are used. Machine models in this class simulate each other with polynomically bounded overhead in time, and constant factor overhead in space.*

die 1. Maschinenklasse

$$\mathrm{RAM}_{+}\text{–}\mathrm{Time}(POL) = \mathrm{RAM}\text{–}\mathrm{Time}_{log}(POL) = \mathcal{P}$$
$$= \mathrm{DTM}\text{–}\mathrm{Time}(POL) = \mathrm{DTM}_1\text{–}\mathrm{Time}(POL)$$

"Random access stored program machine"

Invariance Thesis: *There exists a standard class of machine models, which includes among others all variants of Turing Machines, all variants of RAM's ~~and RASP's~~ with logarithmic time measures, and also the RAM's ~~and RASP's~~ in the uniform time measure and logarithmic space measure, provided only standard arithmetical instructions of additive type are used. Machine models in this class simulate each other with polynomically bounded overhead in time, and constant factor overhead in space.*

die 1. Maschinenklasse

jede log-time beschränkte RAM

$$\text{RAM}_{+}\text{–Time}(POL) = \text{RAM–Time}_{log}(POL) = \mathcal{P}$$
$$= \text{DTM–Time}(POL) = \text{DTM}_1\text{–Time}(POL)$$

"Random access stored program machine"

Invariance Thesis: *There exists a standard class of machine models, which includes among others all variants of Turing Machines, all variants of RAM's ~~and RASP's~~ with logarithmic time measures, and also the RAM's ~~and RASP's~~ in the uniform time measure and logarithmic space measure, provided only standard arithmetical instructions of additive type are used. Machine models in this class simulate each other with polynomically bounded overhead in time, and constant factor overhead in space.*

die 1. Maschinenklasse

jede log-time beschränkte RAM

$$\mathrm{RAM}_{+}\text{–}\mathrm{Time}(POL) = \mathrm{RAM}\text{–}\mathrm{Time}_{log}(POL) = \mathcal{P}$$
$$= \mathrm{DTM}\text{–}\mathrm{Time}(POL) = \mathrm{DTM}_1\text{–}\mathrm{Time}(POL)$$

nur ein Band

"Random access stored program machine"

Invariance Thesis: *There exists a standard class of machine models, which includes among others all variants of Turing Machines, all variants of RAM's ~~and RASP's~~ with logarithmic time measures, and also the RAM's ~~and RASP's~~ in the uniform time measure and logarithmic space measure, provided only standard arithmetical instructions of additive type are used. Machine models in this class simulate each other with polynomically bounded overhead in time, and constant factor overhead in space.*

die 1. Maschinenklasse

jede log-time beschränkte RAM

$$\mathrm{RAM}_{+}\text{–}\mathrm{Time}(POL) = \mathrm{RAM}\text{–}\mathrm{Time}_{log}(POL) = \mathcal{P}$$
$$= \mathrm{DTM}\text{–}\mathrm{Time}(POL) = \mathrm{DTM}_1\text{–}\mathrm{Time}(POL)$$

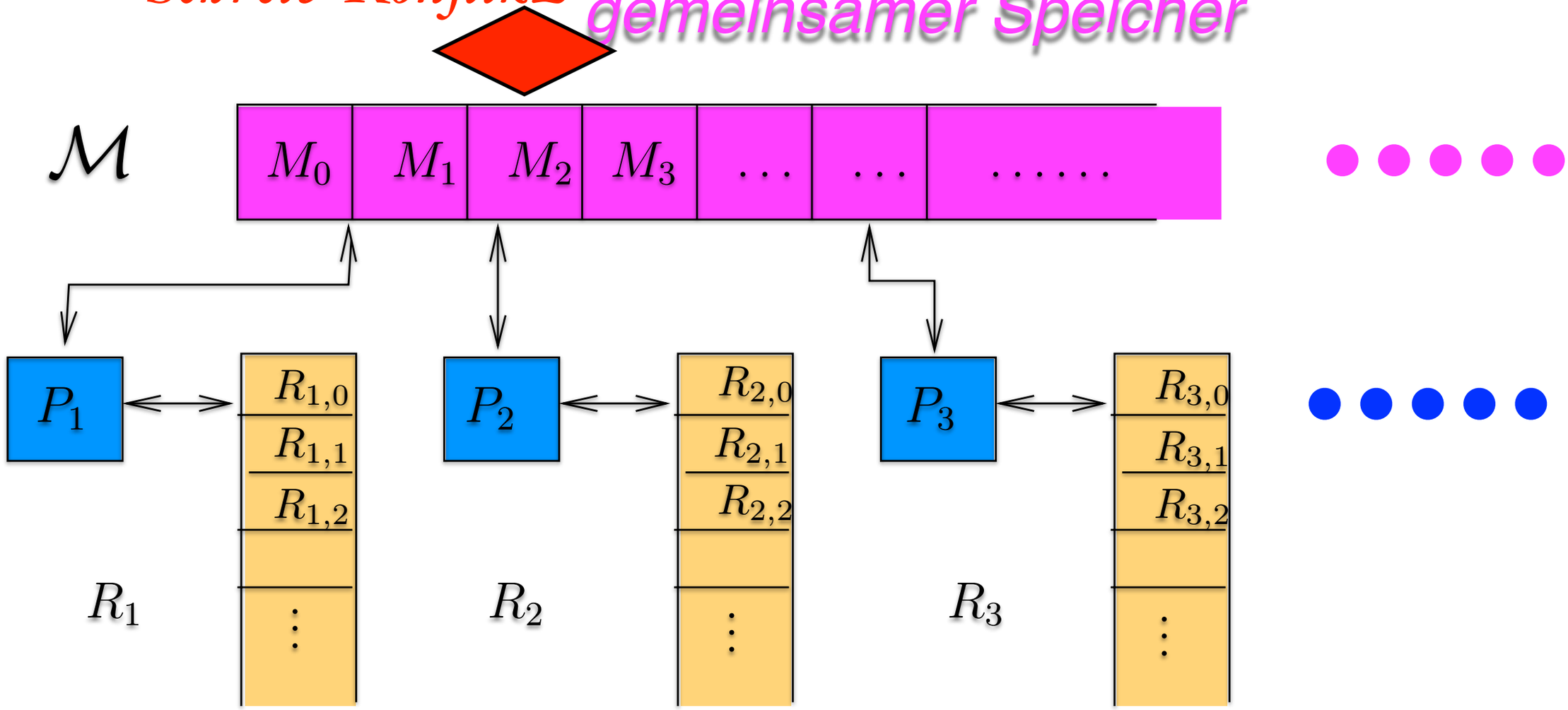
nur ein Band

die 2. Maschinenklasse ⇒ *Parallelrechner*

4.2 Parallele Random-Access-Maschine (PRAM)

Schreib-Konflikt

gemeinsamer Speicher



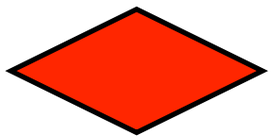
lokaler Speicher

Die Programme sind für alle Prozessoren identisch!

aber neuer Befehl: $LOAD(PID)$

4.2.2 **EREW PRAM** (**E**xclusive **R**ead, **E**xclusive **W**rite):

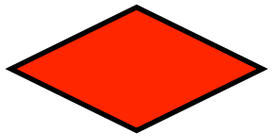
Simultane *READ*s und *WRITE*s sind nicht erlaubt.



Schreib-Konflikt

4.2.2

EREW PRAM (**E**xclusive **R**ead, **E**xclusive **W**rite):
Simultane *READ*s und *WRITE*s sind nicht erlaubt.

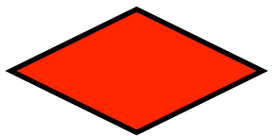


Schreib-Konflikt

CREW PRAM (**C**oncurrent **R**ead, **E**xclusive **W**rite):
Gleichzeitiges Lesen erlaubt, jedoch nur exklusives Schreiben.

4.2.2

EREW PRAM (**E**xclusive **R**ead, **E**xclusive **W**rite):
Simultane *READ*s und *WRITE*s sind nicht erlaubt.



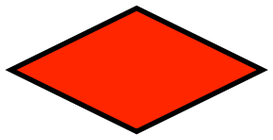
Schreib-Konflikt

CREW PRAM (**C**oncurrent **R**ead, **E**xclusive **W**rite):
Gleichzeitiges Lesen erlaubt, jedoch nur exklusives Schreiben.

CRCW PRAM (**C**oncurrent **R**ead, **C**oncurrent **W**rite):
Simultanes Lesen ist uneingeschränkt erlaubt, simultanes Schreiben ist ebenfalls erlaubt.

4.2.2

EREW PRAM (**E**xclusive **R**ead, **E**xclusive **W**rite):
Simultane *READ*s und *WRITE*s sind nicht erlaubt.



Schreib-Konflikt

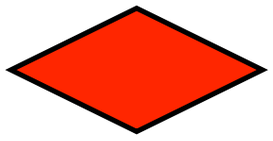
CREW PRAM (**C**oncurrent **R**ead, **E**xclusive **W**rite):
Gleichzeitiges Lesen erlaubt, jedoch nur exklusives Schreiben.

CRCW PRAM (**C**oncurrent **R**ead, **C**oncurrent **W**rite):
Simultanes Lesen ist uneingeschränkt erlaubt, simultanes Schreiben ist ebenfalls erlaubt.

CRCW$^{\text{com}}$ PRAM (**com**mon):
Ein gleichzeitiges Schreiben in ein Register M_j ist nur zulässig, wenn alle beteiligten Prozessoren versuchen denselben Wert zu schreiben.

4.2.2

EREW PRAM (**E**xclusive **R**ead, **E**xclusive **W**rite):
Simultane *READ*s und *WRITE*s sind nicht erlaubt.



Schreib-Konflikt

CREW PRAM (**C**oncurrent **R**ead, **E**xclusive **W**rite):
Gleichzeitiges Lesen erlaubt, jedoch nur exklusives Schreiben.

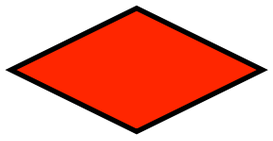
CRCW PRAM (**C**oncurrent **R**ead, **C**oncurrent **W**rite):
Simultanes Lesen ist uneingeschränkt erlaubt, simultanes Schreiben ist ebenfalls erlaubt.

CRCW^{com} PRAM (**com**mon):
Ein gleichzeitiges Schreiben in ein Register M_j ist nur zulässig, wenn alle beteiligten Prozessoren versuchen denselben Wert zu schreiben.

CRCW^{arb} PRAM (**arb**itrary):
Wenn einige Prozessoren versuchen, in ein Register zu schreiben, ist einer erfolgreich. Es ist aber nicht a priori festgelegt, welcher.

4.2.2

EREW PRAM (**E**xclusive **R**ead, **E**xclusive **W**rite):
Simultane *READ*s und *WRITE*s sind nicht erlaubt.



Schreib-Konflikt

CREW PRAM (**C**oncurrent **R**ead, **E**xclusive **W**rite):
Gleichzeitiges Lesen erlaubt, jedoch nur exklusives Schreiben.

CRCW PRAM (**C**oncurrent **R**ead, **C**oncurrent **W**rite):
Simultanes Lesen ist uneingeschränkt erlaubt, simultanes Schreiben ist ebenfalls erlaubt.

CRCW^{com} PRAM (**com**mon):
Ein gleichzeitiges Schreiben in ein Register M_j ist nur zulässig, wenn alle beteiligten Prozessoren versuchen denselben Wert zu schreiben.

CRCW^{arb} PRAM (**arb**itrary):
Wenn einige Prozessoren versuchen, in ein Register zu schreiben, ist einer erfolgreich. Es ist aber nicht a priori festgelegt, welcher.

CRCW^{pri} PRAM (**pri**ority):
Wenn einige Prozessoren versuchen, in ein Register zu schreiben, ist der mit dem kleinsten Index erfolgreich (Man sagt, dieser besitzt die höchste Priorität).

4.2.3 Komplexitätsmaße der PRAM

$T_M(n)$

Definition 4.10 *Die uniforme (logarithmische)* Zeitkomplexität *$time_M(x)$ einer PRAM M auf eine Eingbe x ist entsprechend wie für die RAM in Definition 7.4 definiert, wobei die synchronen Schritte bis zur Termination gerechnet werden. Die Zeitkomplexität $T_M(n)$ von M ist wieder das Maximum aller $time_M(x)$, wobei das Maximum über alle Eingaben x der uniformen (logarithmischen) Größe n gerechnet wird.*

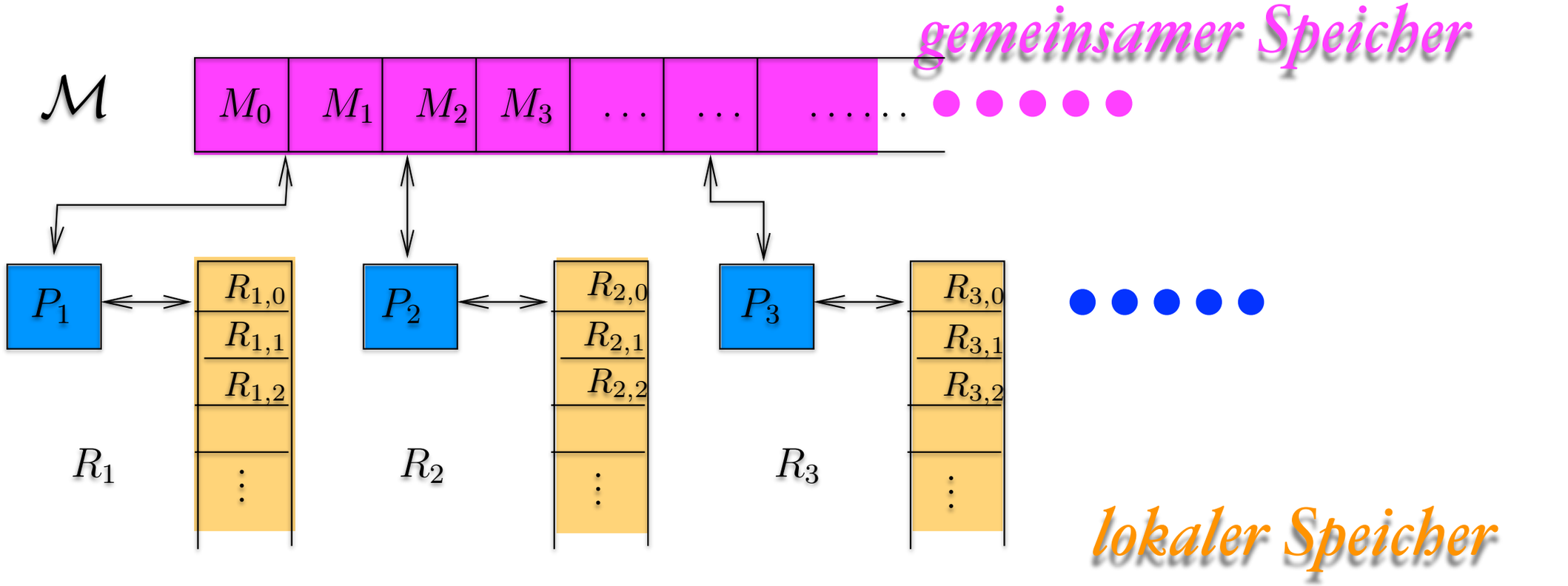
$n \in \mathbb{N}$

$P_M(n)$

Definition Die *Prozessorkomplexität* $proc_M(x)$ einer PRAM M auf einer Eingabe x ist:

$$proc_M(x) = max\{i \mid i = 1 \text{ oder } P_i \text{ führt ein } WRITE \text{ aus}\}$$

Die *Prozessorkomplexität* $P_M(n)$ von M ist das Maximum aller $proc_M(x)$, wobei das Maximum über alle Eingaben der uniformen (logarithmischen) Größe n gebildet wird,



Definition 4.12 *Zwei weitere wichtige Komplexitätsmaße für PRAMs sind das Prozessor-Zeit-Produkt für die Eingabe x:*

$$pt_M(x) = time_M(x) \cdot proc_M(x)$$

,

Definition 4.12 *Zwei weitere wichtige Komplexitätsmaße für PRAMs sind das Prozessor-Zeit-Produkt für die Eingabe x:*

$$pt_M(x) = time_M(x) \cdot proc_M(x)$$

und Work, das die Gesamtzahl der Schritte aller Prozessoren, die während der Berechnung aktiv sind, nach oben hin abschätzt. Letztere wird als

$$work_M(x)$$

Definition 4.12 *Zwei weitere wichtige Komplexitätsmaße für PRAMs sind das Prozessor-Zeit-Produkt für die Eingabe x:*

$$pt_M(x) = time_M(x) \cdot proc_M(x)$$

und Work, das die Gesamtzahl der Schritte aller Prozessoren, die während der Berechnung aktiv sind, nach oben hin abschätzt. Letztere wird als

$$work_M(x)$$

Operationenkomplexität

$W_M(n)$

Definition 4.12 *Zwei weitere wichtige Komplexitätsmaße für PRAMs sind das Prozessor-Zeit-Produkt für die Eingabe x:*

$$pt_M(x) = time_M(x) \cdot proc_M(x)$$

und Work, das die Gesamtzahl der Schritte aller Prozessoren, die während der Berechnung aktiv sind, nach oben hin abschätzt. Letztere wird als

$$work_M(x)$$

Operationenkomplexität

$W_M(n)$

bezeichnet. Die Operationenkomplexität *$W_M(n)$ ist dann wieder das Maximum aller $work_M(x)$, wobei das Maximum über alle Eingaben x der uniformen (logarithmischen) Größe n gerechnet wird.*

ein Beispiel

Definition 4.12 *Zwei weitere wichtige Komplexitätsmaße für PRAMs sind das Prozessor-Zeit-Produkt für die Eingabe x:*

$$pt_M(x) = time_M(x) \cdot proc_M(x)$$

und Work, das die Gesamtzahl der Schritte aller Prozessoren, die während der Berechnung aktiv sind nach oben hin abschätzt. Letztere wird als

$$work_M(x)$$

Operationenkomplexität

$W_M(n)$

ein Beispiel

Definition 4.12 *Zwei weitere wichtige Komplexitätsmaße für PRAMs sind das Prozessor-Zeit-Produkt für die Eingabe x:*

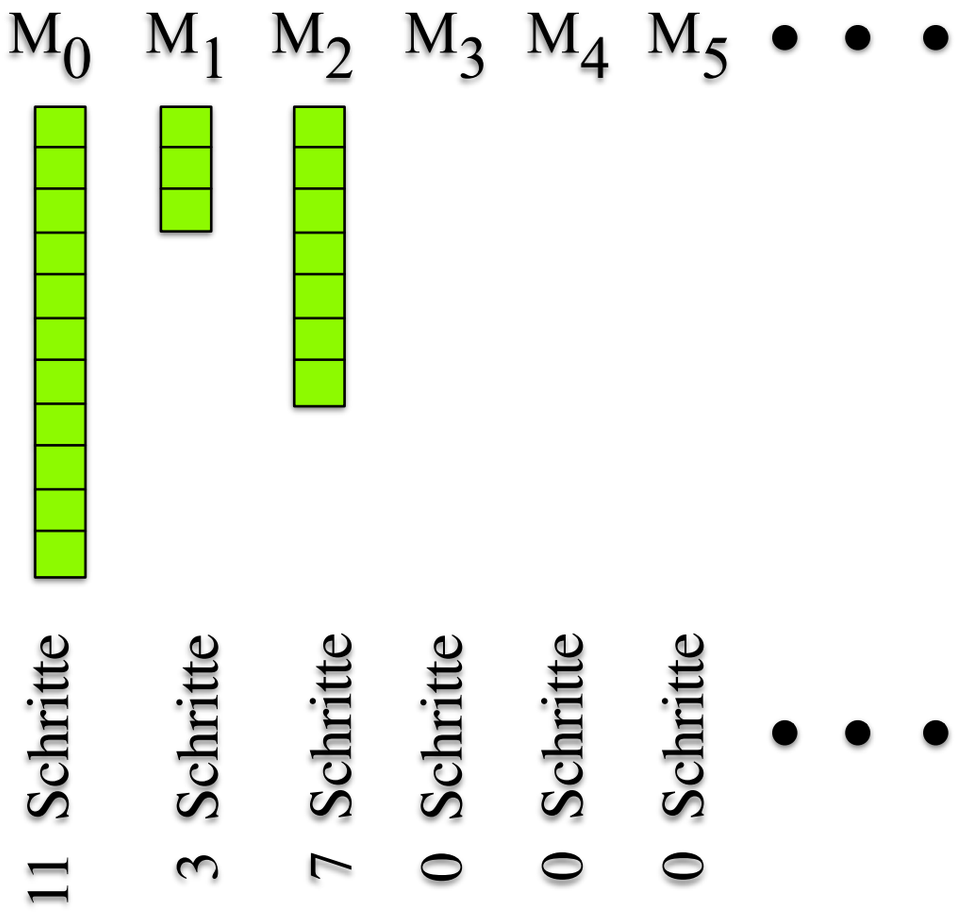
$$pt_M(x) = time_M(x) \cdot proc_M(x)$$

und Work, das die Gesamtzahl der Schritte aller Prozessoren, die während der Berechnung aktiv sind nach oben hin abschätzt. Letztere wird als

$$work_M(x)$$

Operationenkomplexität

$W_M(n)$



ein Beispiel

Definition 4.12 *Zwei weitere wichtige Komplexitätsmaße für PRAMs sind das Prozessor-Zeit-Produkt für die Eingabe x:*

$$pt_M(x) = time_M(x) \cdot proc_M(x)$$

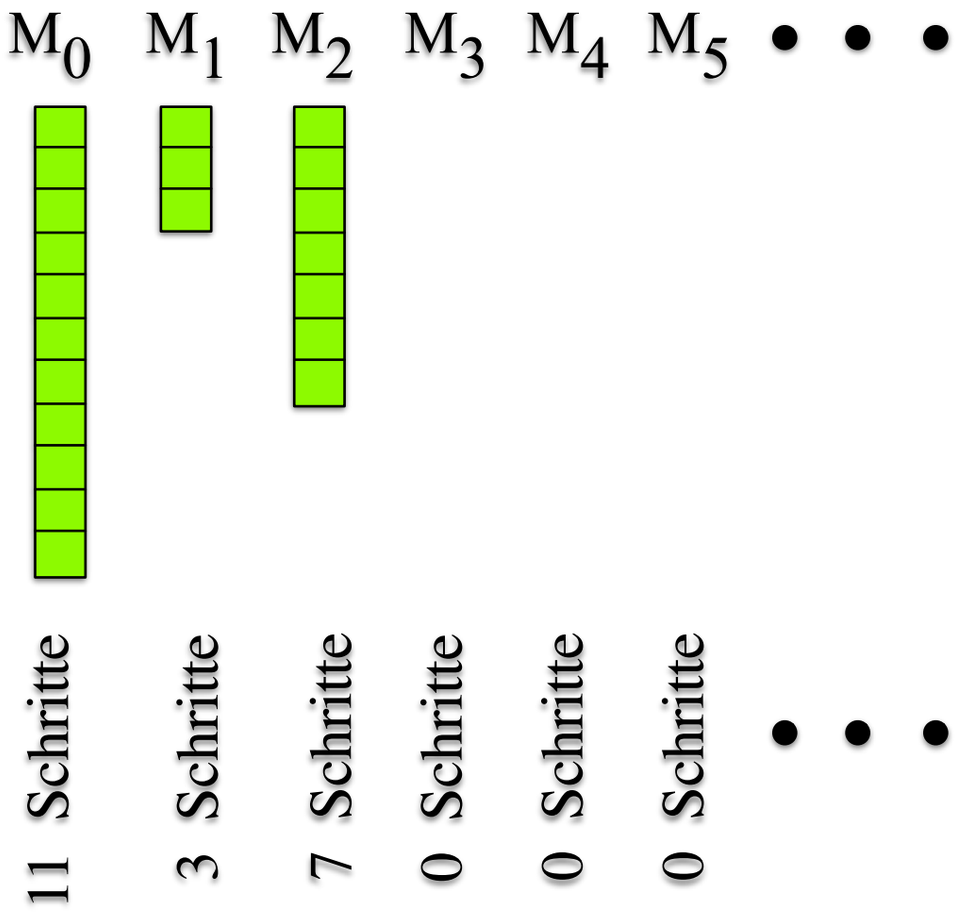
und Work, das die Gesamtzahl der Schritte aller Prozessoren, die während der Berechnung aktiv sind nach oben hin abschätzt. Letztere wird als

$$work_M(x)$$

$proc_M(x) = P_M(n) =$

Operationenkomplexität

$W_M(n)$



ein Beispiel

Definition 4.12 *Zwei weitere wichtige Komplexitätsmaße für PRAMs sind das Prozessor-Zeit-Produkt für die Eingabe x:*

$$pt_M(x) = time_M(x) \cdot proc_M(x)$$

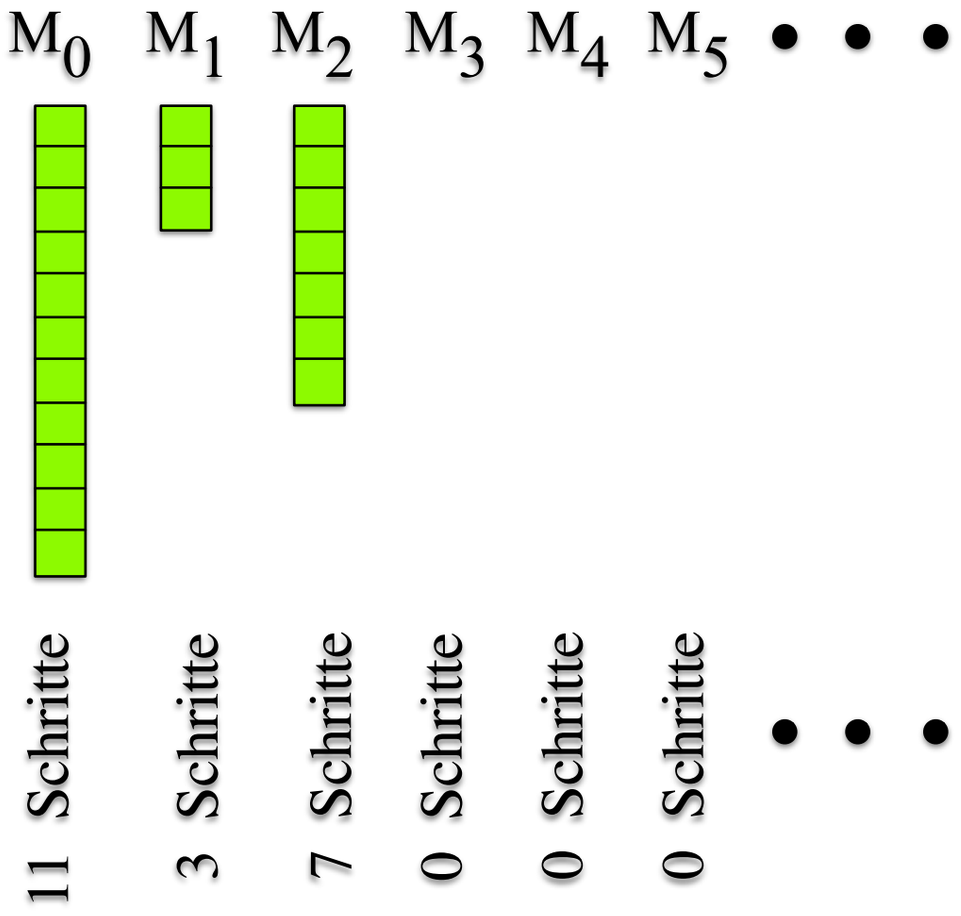
und Work, das die Gesamtzahl der Schritte aller Prozessoren, die während der Berechnung aktiv sind nach oben hin abschätzt. Letztere wird als

$$work_M(x)$$

Operationenkomplexität

$W_M(n)$

$proc_M(x) = P_M(n) = 3$



ein Beispiel

Definition 4.12 *Zwei weitere wichtige Komplexitätsmaße für PRAMs sind das Prozessor-Zeit-Produkt für die Eingabe x:*

$$pt_M(x) = time_M(x) \cdot proc_M(x)$$

und Work, das die Gesamtzahl der Schritte aller Prozessoren, die während der Berechnung aktiv sind nach oben hin abschätzt. Letztere wird als

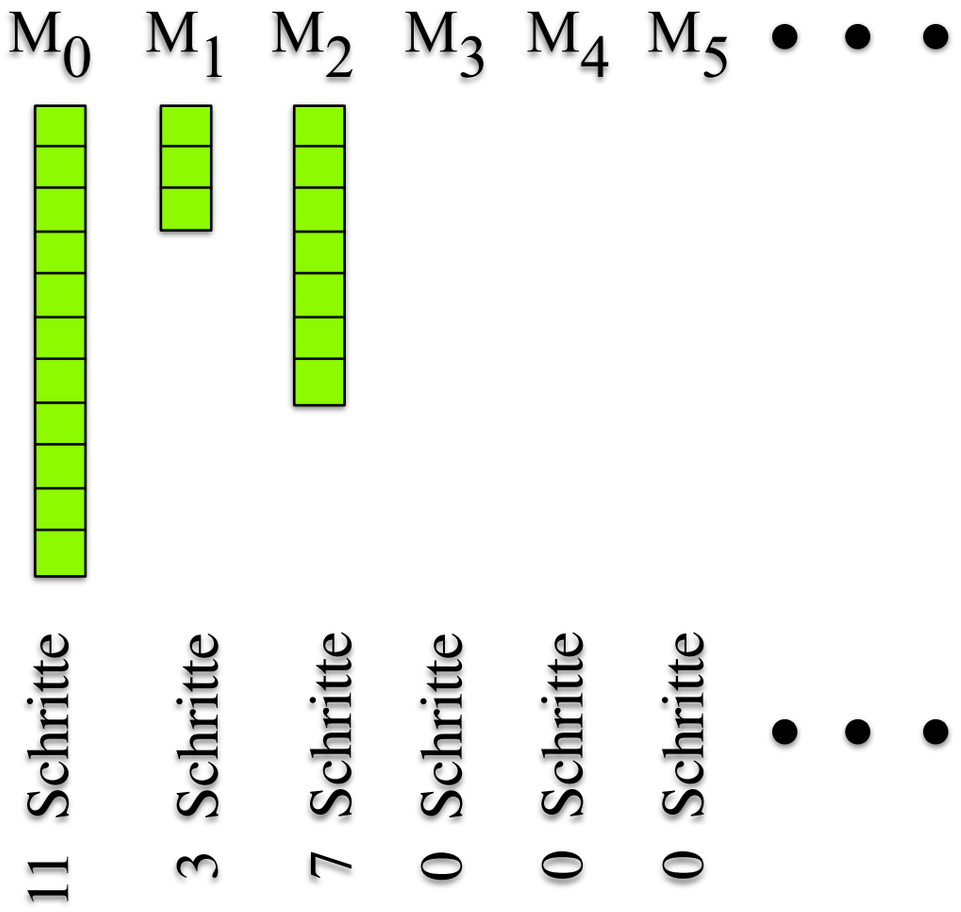
$$work_M(x)$$

Operationenkomplexität

$W_M(n)$

$proc_M(x) = P_M(n) = 3$

$time_M(x) = T_M(n) =$



ein Beispiel

Definition 4.12 *Zwei weitere wichtige Komplexitätsmaße für PRAMs sind das Prozessor-Zeit-Produkt für die Eingabe x:*

$$pt_M(x) = time_M(x) \cdot proc_M(x)$$

und Work, das die Gesamtzahl der Schritte aller Prozessoren, die während der Berechnung aktiv sind nach oben hin abschätzt. Letztere wird als

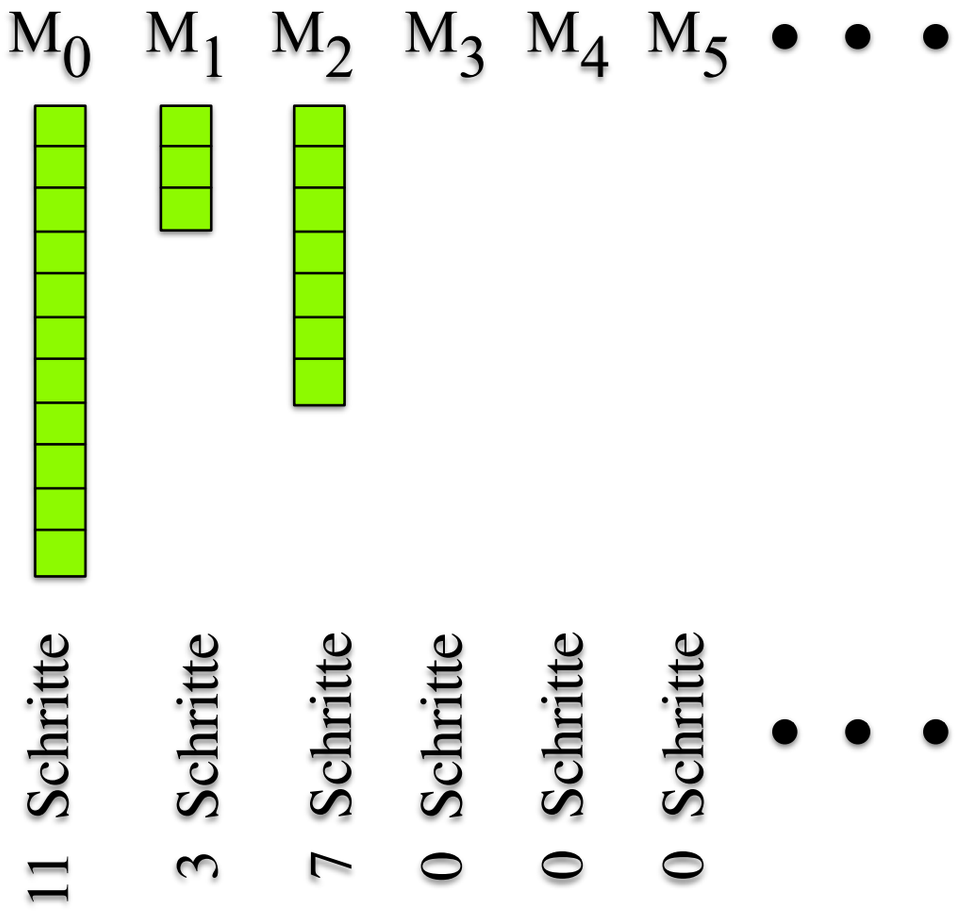
$$work_M(x)$$

Operationenkomplexität

$W_M(n)$

$proc_M(x) = P_M(n) = 3$

$time_M(x) = T_M(n) = 11$



ein Beispiel

Definition 4.12 *Zwei weitere wichtige Komplexitätsmaße für PRAMs sind das Prozessor-Zeit-Produkt für die Eingabe x:*

$$pt_M(x) = time_M(x) \cdot proc_M(x)$$

und Work, das die Gesamtzahl der Schritte aller Prozessoren, die während der Berechnung aktiv sind nach oben hin abschätzt. Letztere wird als

$$work_M(x)$$

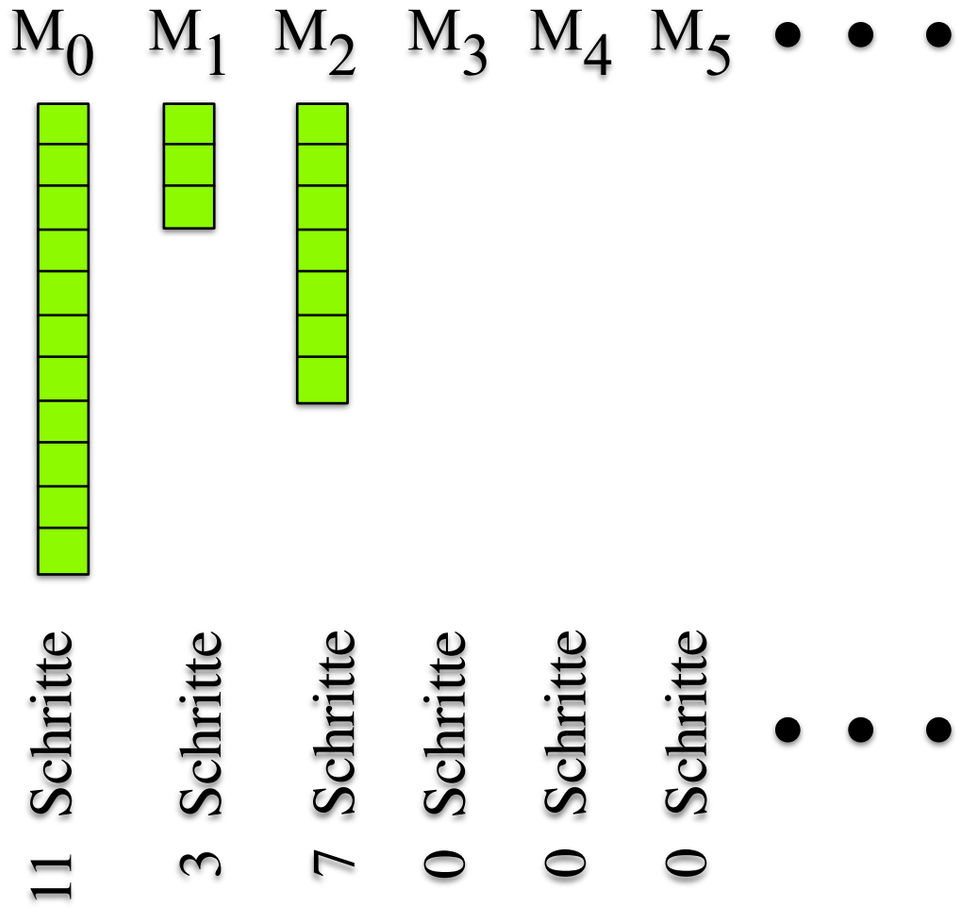
Operationenkomplexität

$W_M(n)$

$proc_M(x) = P_M(n) = 3$

$time_M(x) = T_M(n) = 11$

$work_M(x) = W_M(n) =$



ein Beispiel

Definition 4.12 *Zwei weitere wichtige Komplexitätsmaße für PRAMs sind das Prozessor-Zeit-Produkt für die Eingabe x:*

$$pt_M(x) = time_M(x) \cdot proc_M(x)$$

und Work, das die Gesamtzahl der Schritte aller Prozessoren, die während der Berechnung aktiv sind nach oben hin abschätzt. Letztere wird als

$$work_M(x)$$

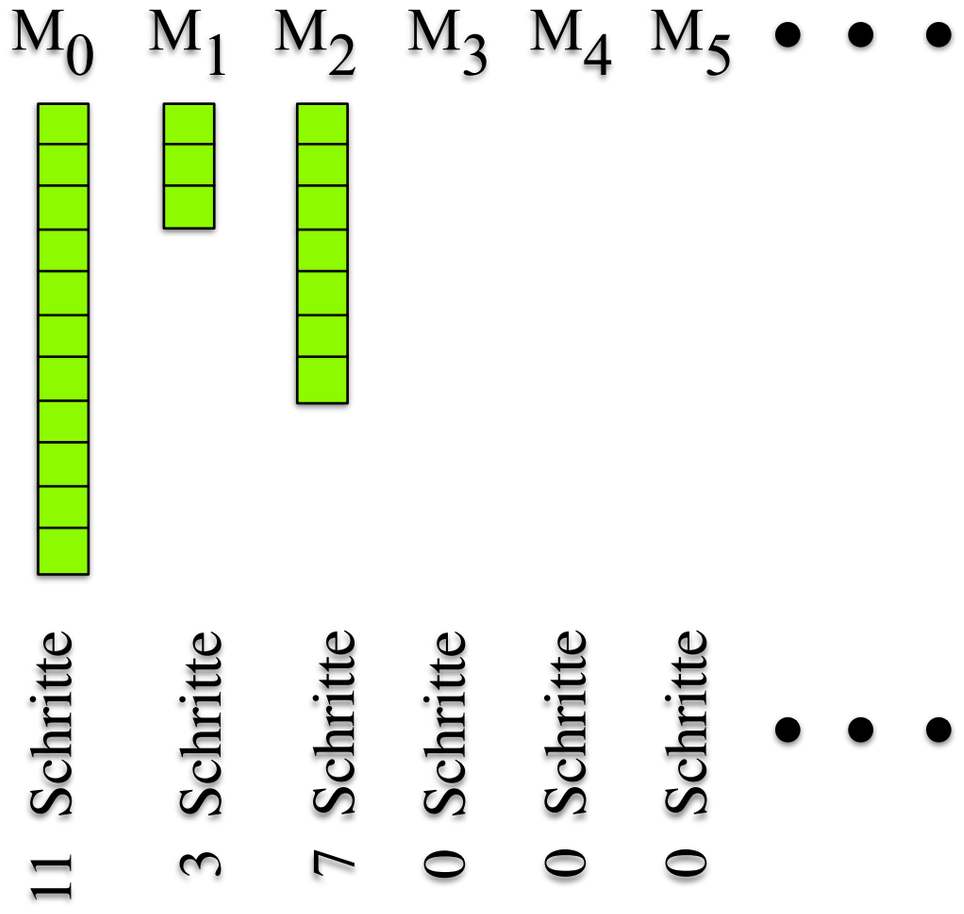
Operationenkomplexität

$W_M(n)$

$proc_M(x) = P_M(n) = 3$

$time_M(x) = T_M(n) = 11$

$work_M(x) = W_M(n) = 21$



ein Beispiel

Definition 4.12 *Zwei weitere wichtige Komplexitätsmaße für PRAMs sind das Prozessor-Zeit-Produkt für die Eingabe x:*

$$pt_M(x) = time_M(x) \cdot proc_M(x)$$

und Work, das die Gesamtzahl der Schritte aller Prozessoren, die während der Berechnung aktiv sind nach oben hin abschätzt. Letztere wird als

$$work_M(x)$$

Operationenkomplexität

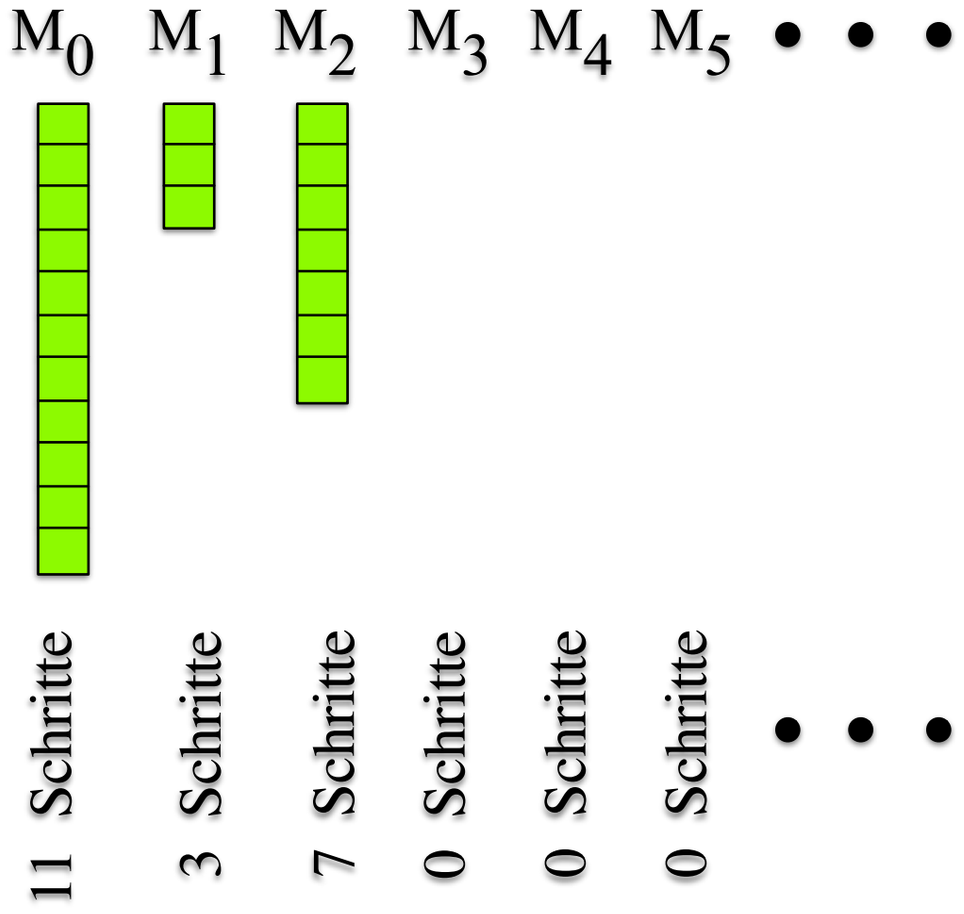
$W_M(n)$

$proc_M(x) = P_M(n) = 3$

$time_M(x) = T_M(n) = 11$

$work_M(x) = W_M(n) = 21$

$pt_M(x) = 33$



ein Beispiel

Definition 4.12 *Zwei weitere wichtige Komplexitätsmaße für PRAMs sind das Prozessor-Zeit-Produkt für die Eingabe x:*

$$pt_M(x) = time_M(x) \cdot proc_M(x)$$

und Work, das die Gesamtzahl der Schritte aller Prozessoren, die während der Berechnung aktiv sind nach oben hin abschätzt. Letztere wird als

$$work_M(x)$$

Operationenkomplexität

$W_M(n)$

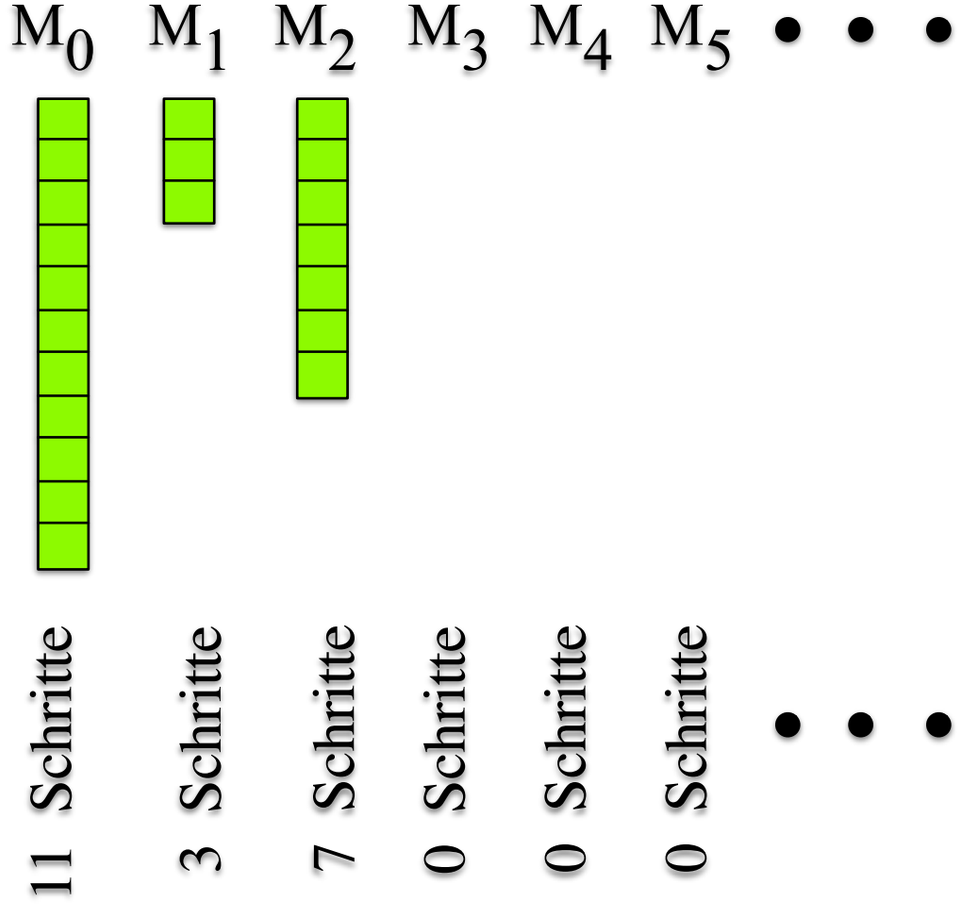
$proc_M(x) = P_M(n) = 3$

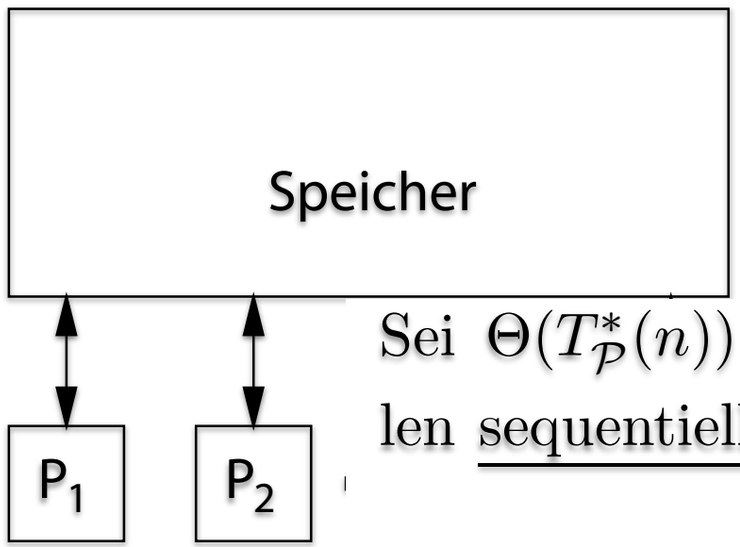
$time_M(x) = T_M(x) = 11$

$work_M(x) = W_M(n) = 21$

$pt_M(x) = 33$

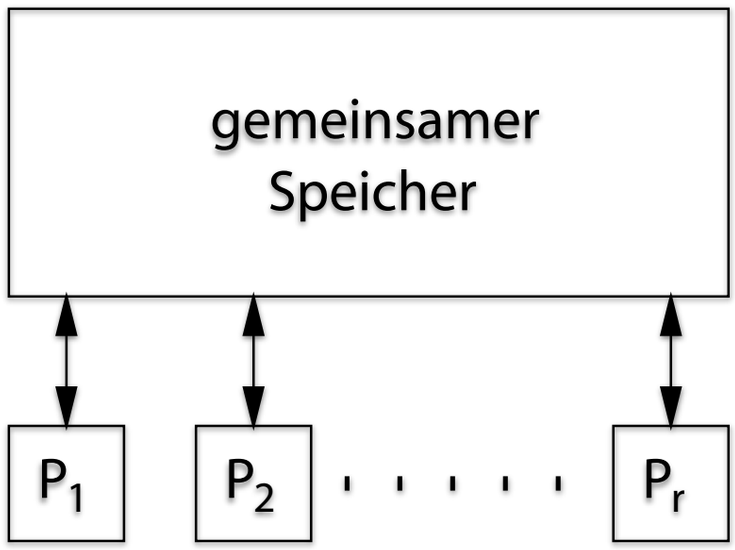
oft: $pt_M(x) \approx work_M(x)$





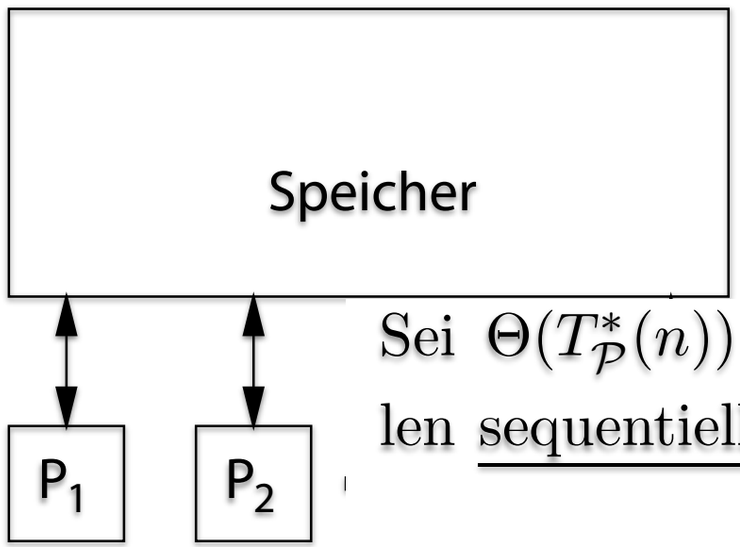
sequentieller Alg. mit T(n)

Sei $\Theta(T^*_{\mathcal{P}}(n))$ die beste Zeitkomplexität für ein Problem $\mathcal{P}$ unter allen sequentiellen Algorithmen, die das Problem $\mathcal{P}$ lösen



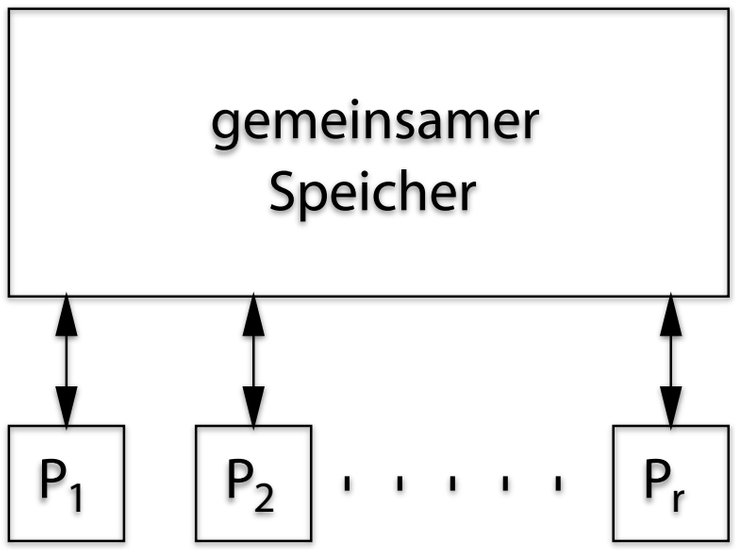
paralleler Alg.

T(n) = ?



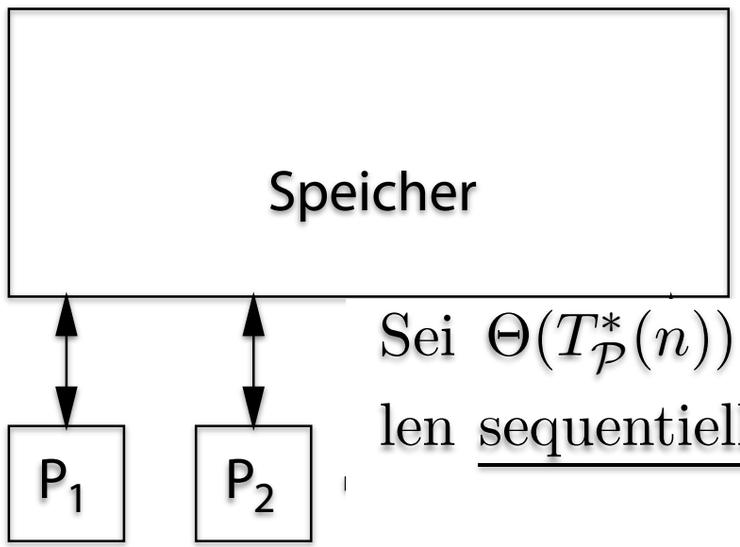
sequentieller Alg. mit T(n)

Sei $\Theta(T^*_{\mathcal{P}}(n))$ die beste Zeitkomplexität für ein Problem $\mathcal{P}$ unter allen sequentiellen Algorithmen, die das Problem $\mathcal{P}$ lösen



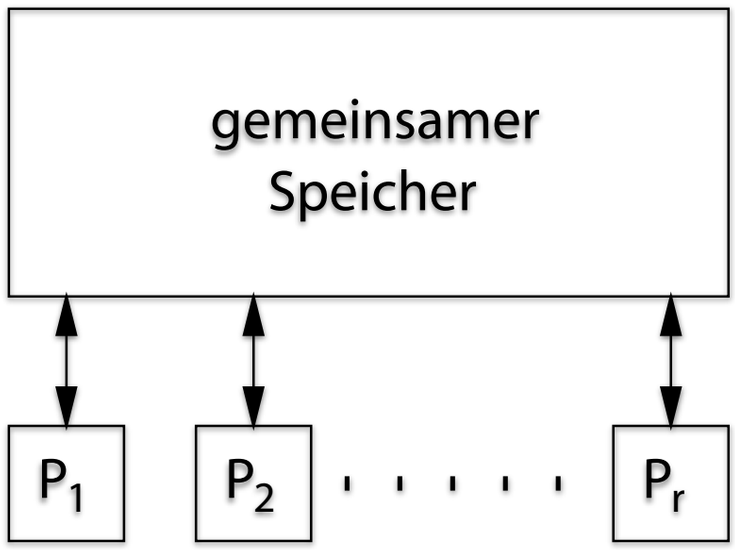
paralleler Alg.

T(n) = ? besser!



sequentieller Alg. mit T(n)

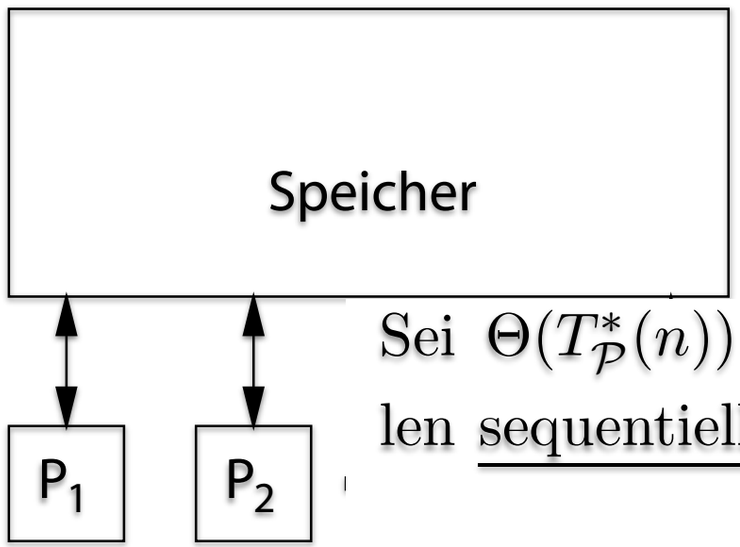
Sei $\Theta(T^*_{\mathcal{P}}(n))$ die beste Zeitkomplexität für ein Problem $\mathcal{P}$ unter allen sequentiellen Algorithmen, die das Problem $\mathcal{P}$ lösen



paralleler Alg.

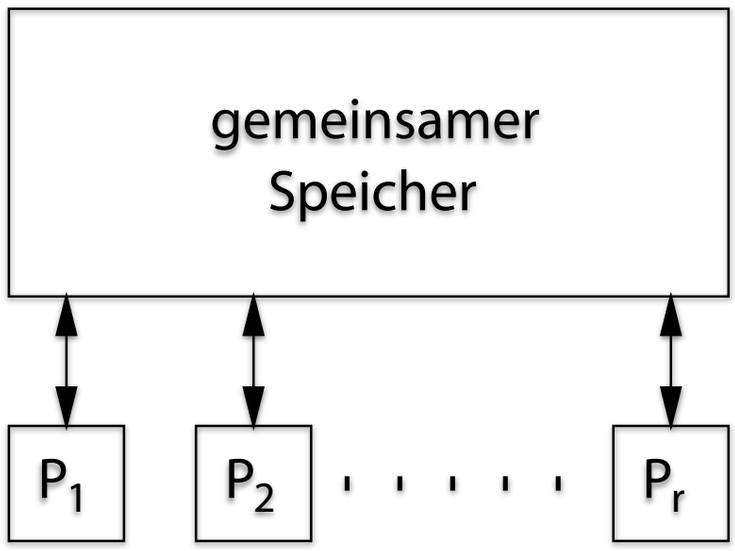
T(n) = ? besser!

optimal ???



sequentieller Alg. mit T(n)

Sei $\Theta(T^*_{\mathcal{P}}(n))$ die beste Zeitkomplexität für ein Problem $\mathcal{P}$ unter allen sequentiellen Algorithmen, die das Problem $\mathcal{P}$ lösen

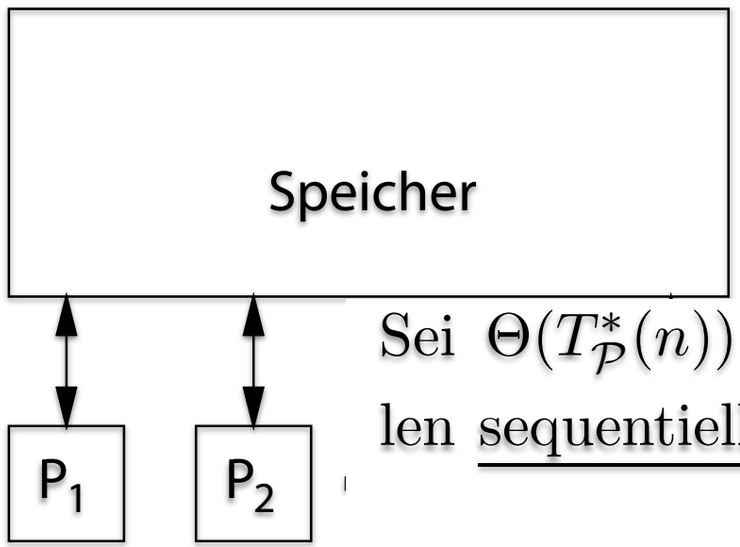


paralleler Alg.

T(n) = ? besser!

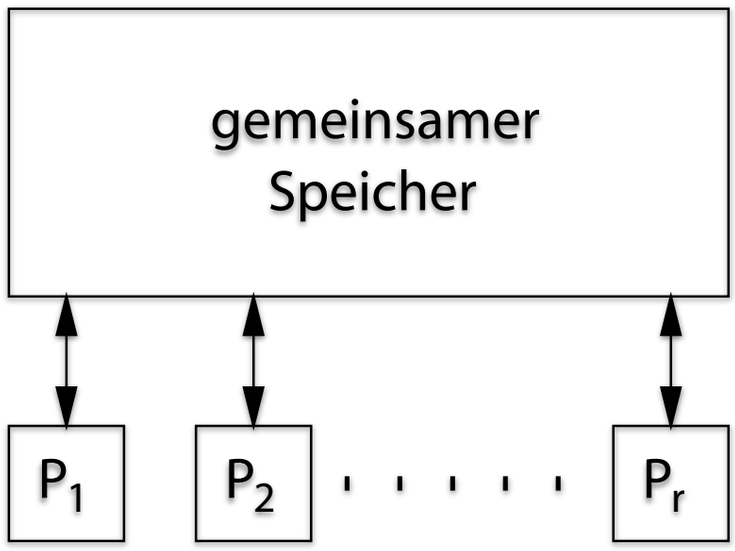
optimal ???

Der Alg. heißt *optimal*, falls



sequentieller Alg. mit T(n)

Sei $\Theta(T^*_{\mathcal{P}}(n))$ die beste Zeitkomplexität für ein Problem $\mathcal{P}$ unter allen sequentiellen Algorithmen, die das Problem $\mathcal{P}$ lösen



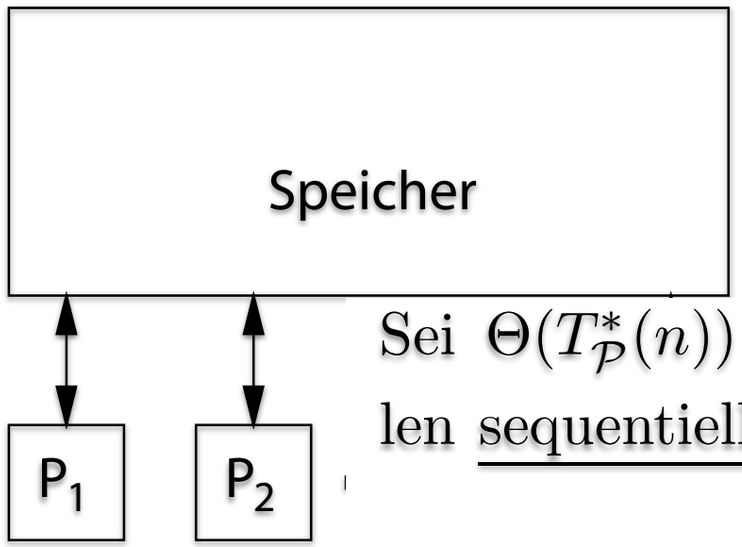
paralleler Alg.

T(n) = ? besser!

optimal ???

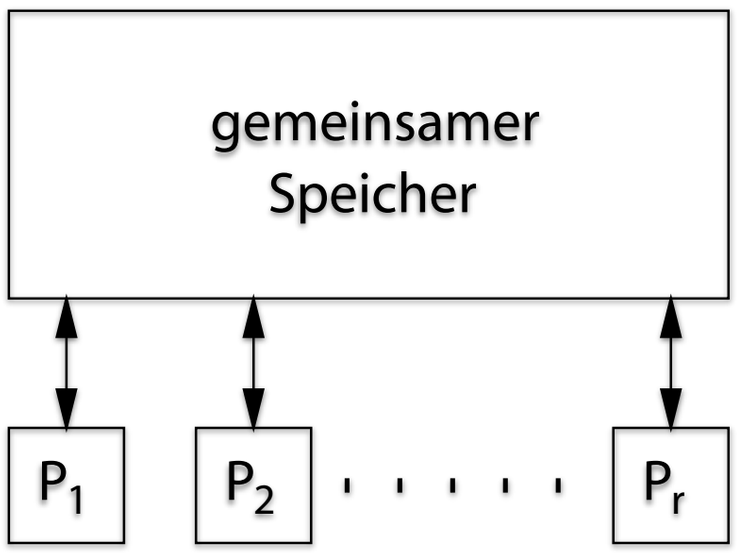
Der Alg. heißt *optimal*, falls

$$W_A(n) = \Theta(T^*_{\mathcal{P}}(n))$$

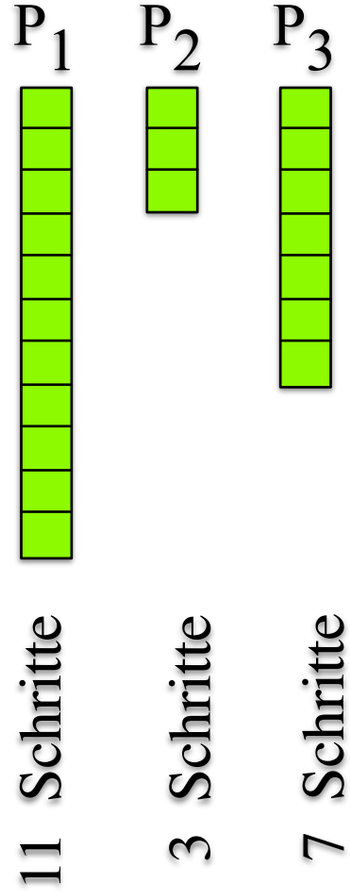


sequentieller Alg. mit T(n)

Sei $\Theta(T^*_{\mathcal{P}}(n))$ die beste Zeitkomplexität für ein Problem $\mathcal{P}$ unter allen sequentiellen Algorithmen, die das Problem $\mathcal{P}$ lösen



$W_A(n) = 21$



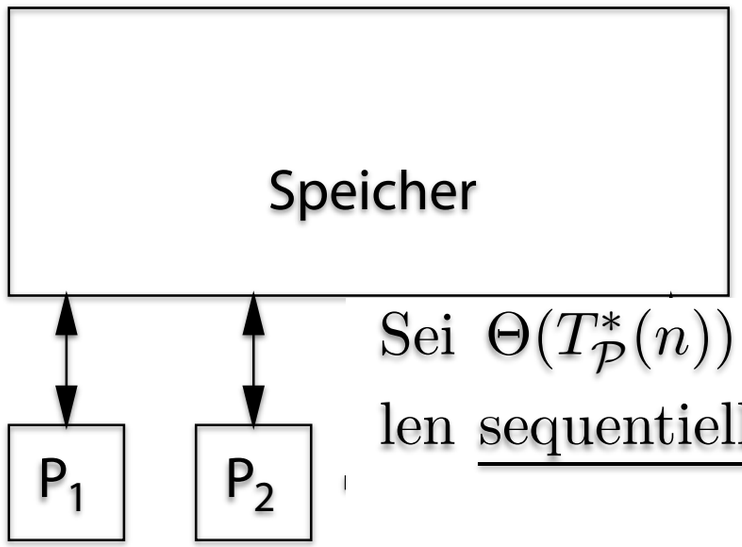
paralleler Alg.

T(n) = ? besser!

optimal ???

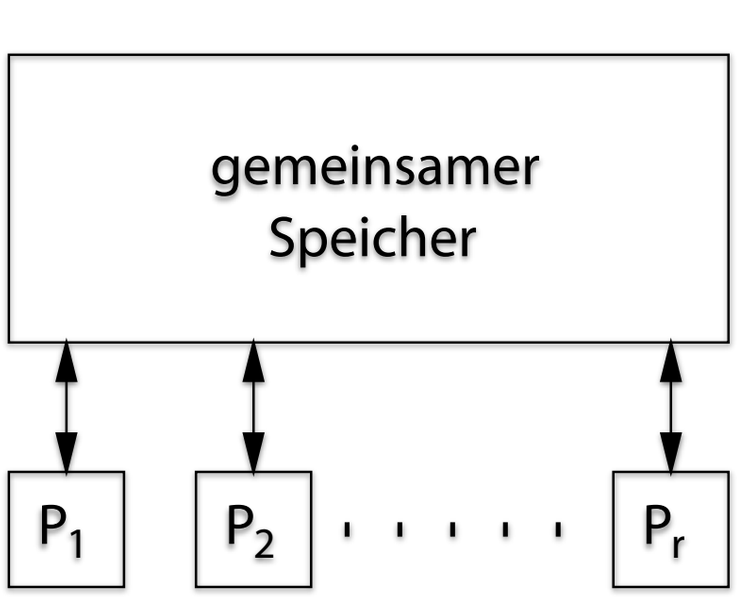
Der Alg. heißt *optimal*, falls

$$W_A(n) = \Theta(T^*_{\mathcal{P}}(n))$$

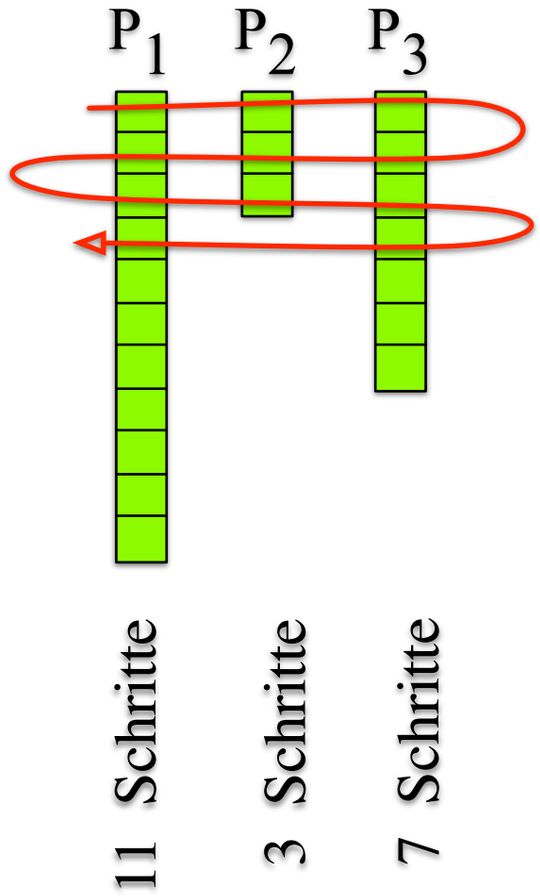


sequentieller Alg. mit T(n)

Sei $\Theta(T^*_{\mathcal{P}}(n))$ die beste Zeitkomplexität für ein Problem $\mathcal{P}$ unter allen sequentiellen Algorithmen, die das Problem $\mathcal{P}$ lösen



$W_A(n) = 21$



paralleler Alg.

T(n) = ? besser!

optimal ???

Der Alg. heißt *optimal*, falls

$$W_A(n) = \Theta(T^*_{\mathcal{P}}(n))$$

Satz 4.13 (Brent's scheduling principle) *Wenn man eine Berechnung auf* $m = max\ x_i$ *Prozessoren in Zeit* t *ausführen kann, wobei* x_i *die Anzahl der Operationen im* i*-ten Schritt ist, dann kann man auf*

$$p < m$$

Prozessoren dieselbe Berechnung in der Zeit

$$t + \lfloor \frac{x}{p} \rfloor$$

ausführen, wobei $x = \sum x_i$

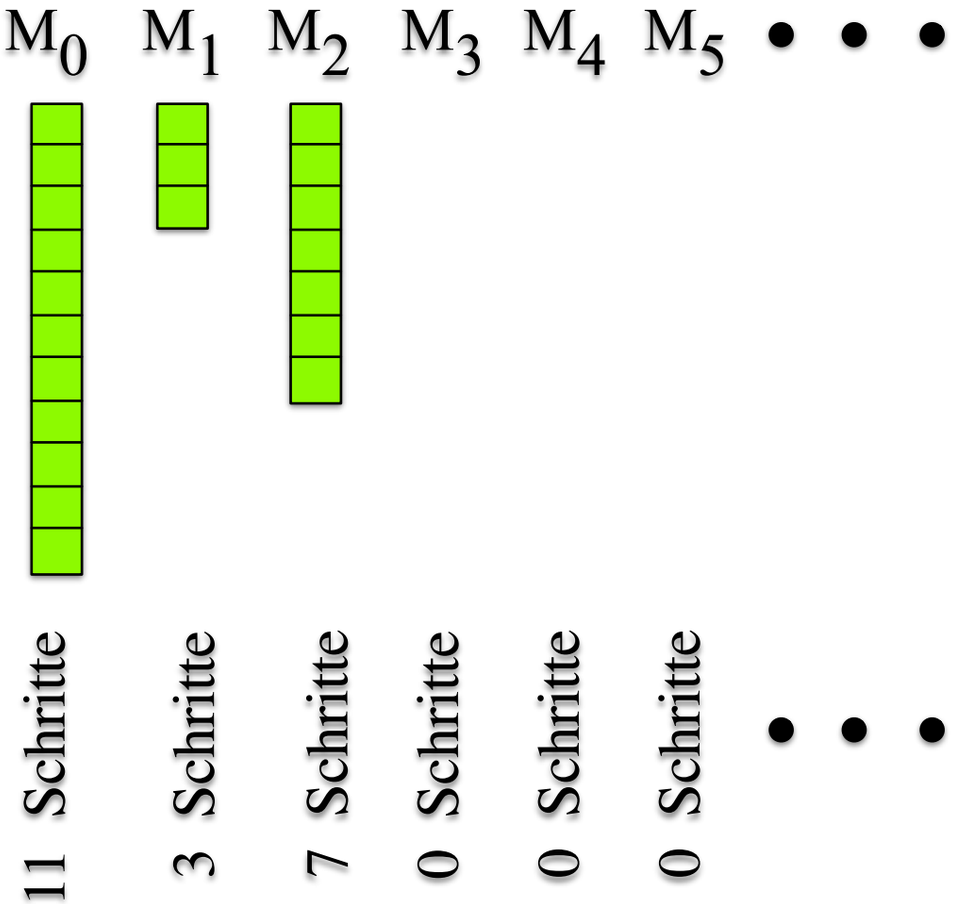
Satz 4.13 (Brent's scheduling principle) *Wenn man eine Berechnung auf $m = max\ x_i$ Prozessoren in Zeit t ausführen kann, wobei x_i die Anzahl der Operationen im i-ten Schritt ist, dann kann man auf*

$$p < m$$

Prozessoren dieselbe Berechnung in der Zeit

$$t + \lfloor \frac{x}{p} \rfloor$$

ausführen, wobei $x = \sum x_i$



Satz 4.13 (Brent's scheduling principle) *Wenn man eine Berechnung auf $m = max\ x_i$ Prozessoren in Zeit t ausführen kann, wobei x_i die Anzahl der Operationen im i-ten Schritt ist, dann kann man auf*

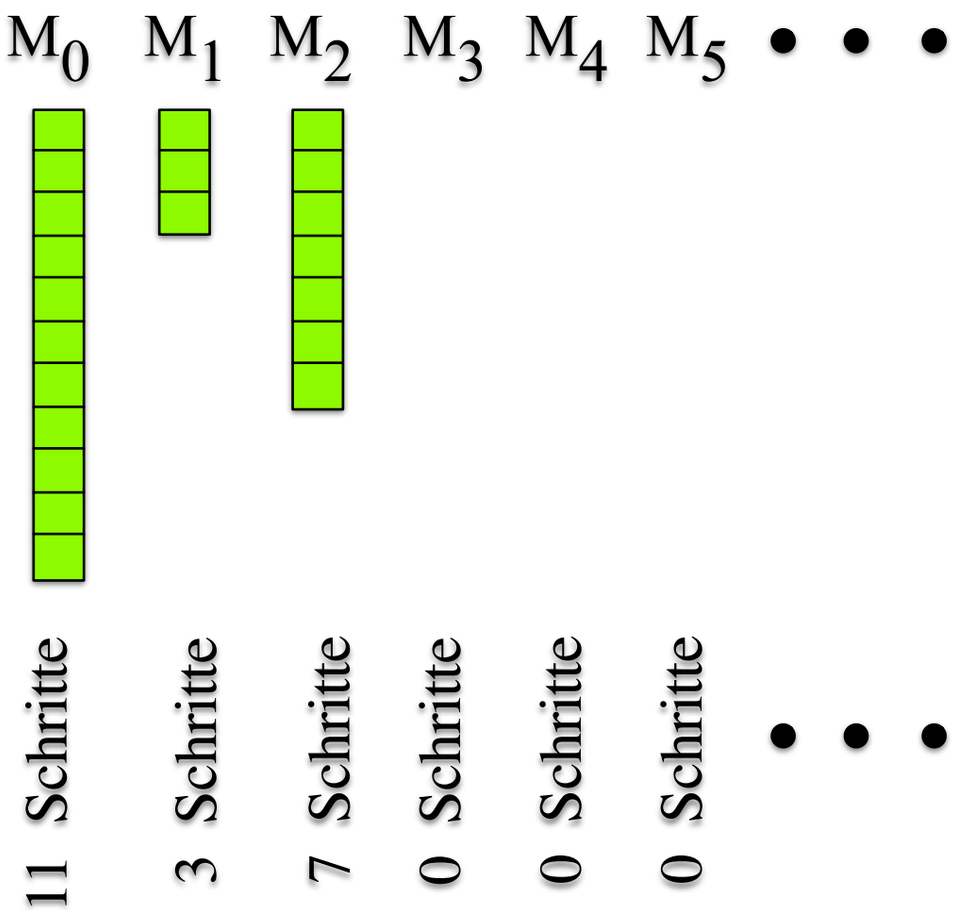
$$p < m$$

3

Prozessoren dieselbe Berechnung in der Zeit

$$t + \lfloor \frac{x}{p} \rfloor$$

ausführen, wobei $x = \sum x_i$



Satz 4.13 (Brent's scheduling principle) *Wenn man eine Berechnung auf $m = max\ x_i$ Prozessoren in Zeit t ausführen kann, wobei x_i die Anzahl der Operationen im i-ten Schritt ist, dann kann man auf*

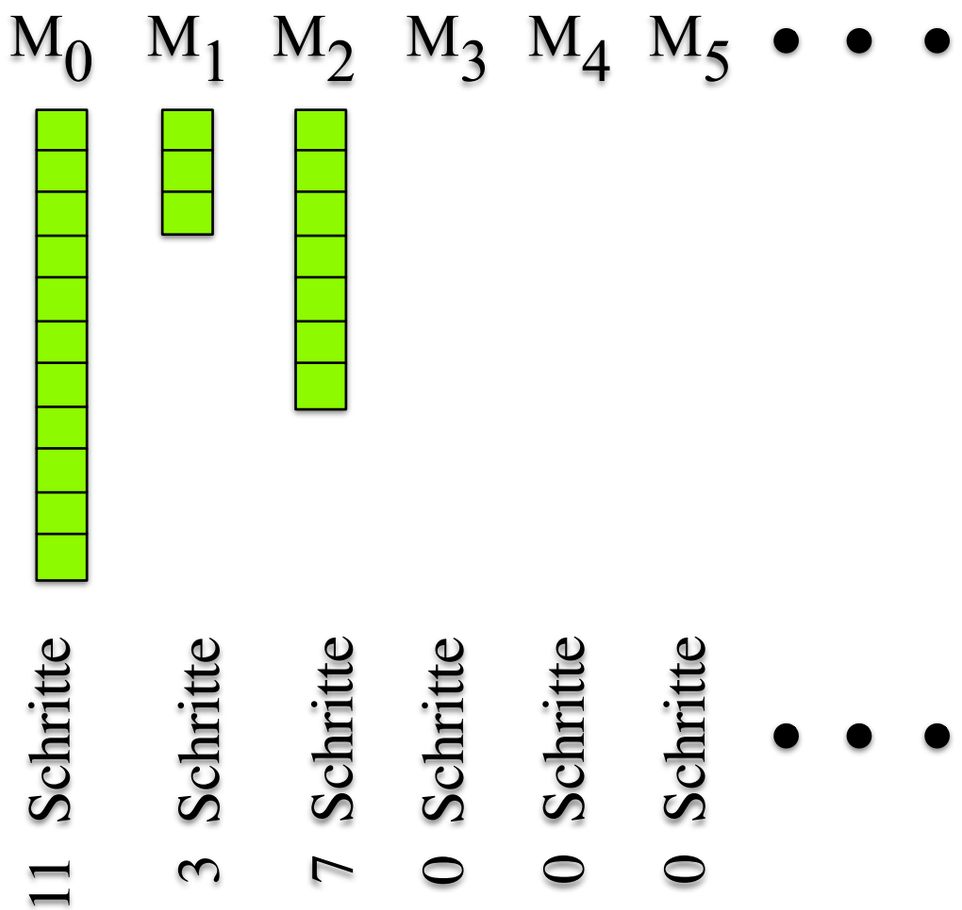
$$p < m$$

2 3

Prozessoren dieselbe Berechnung in der Zeit

$$t + \lfloor \frac{x}{p} \rfloor$$

ausführen, wobei $x = \sum x_i$



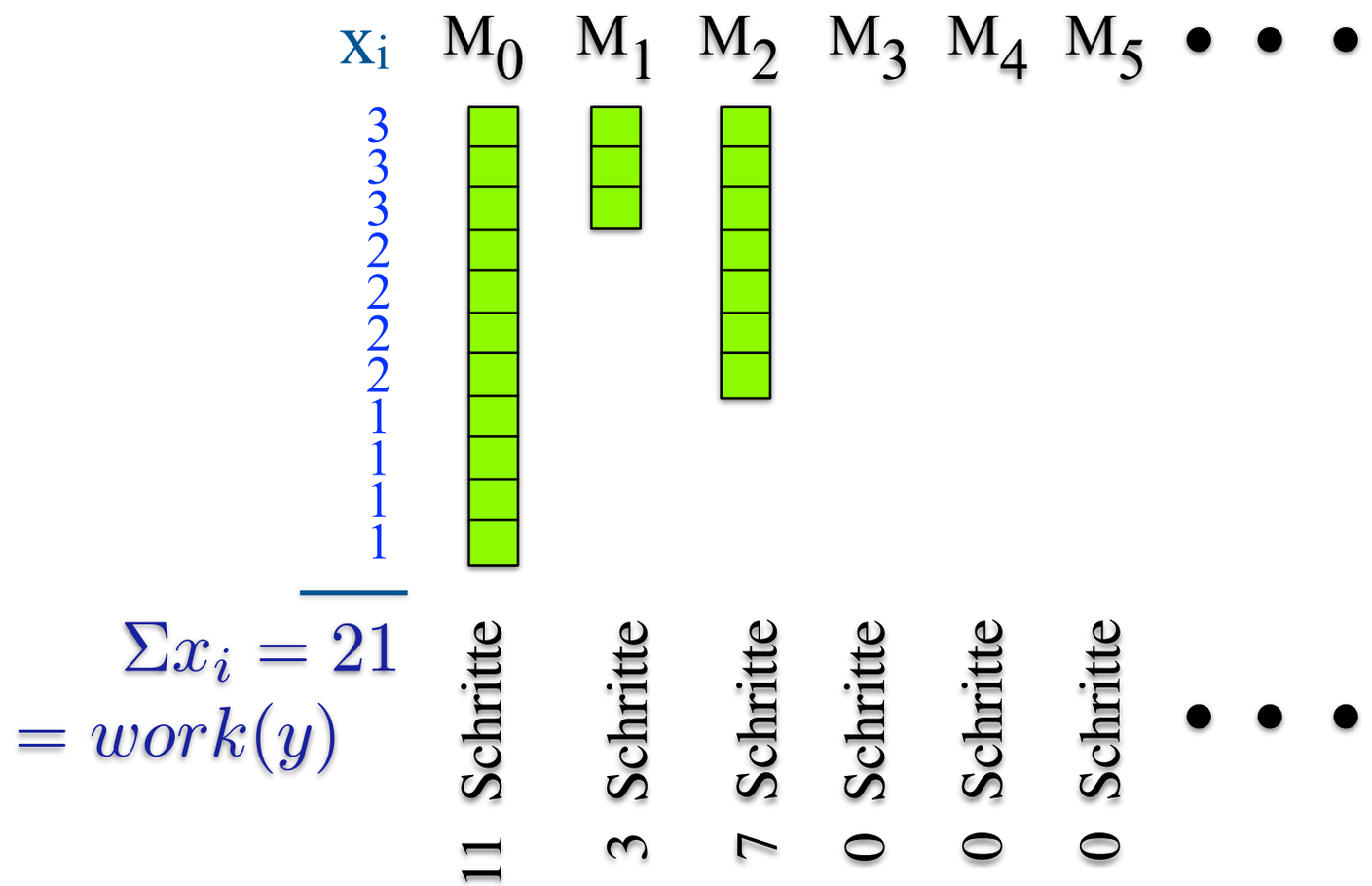
Satz 4.13 (Brent's scheduling principle) *Wenn man eine Berechnung auf $m = max\ x_i$ Prozessoren in Zeit t ausführen kann, wobei x_i die Anzahl der Operationen im i-ten Schritt ist, dann kann man auf*

$$p < m$$
2 3

Prozessoren dieselbe Berechnung in der Zeit

$$t + \lfloor \frac{x}{p} \rfloor$$

ausführen, wobei $x = \sum x_i$



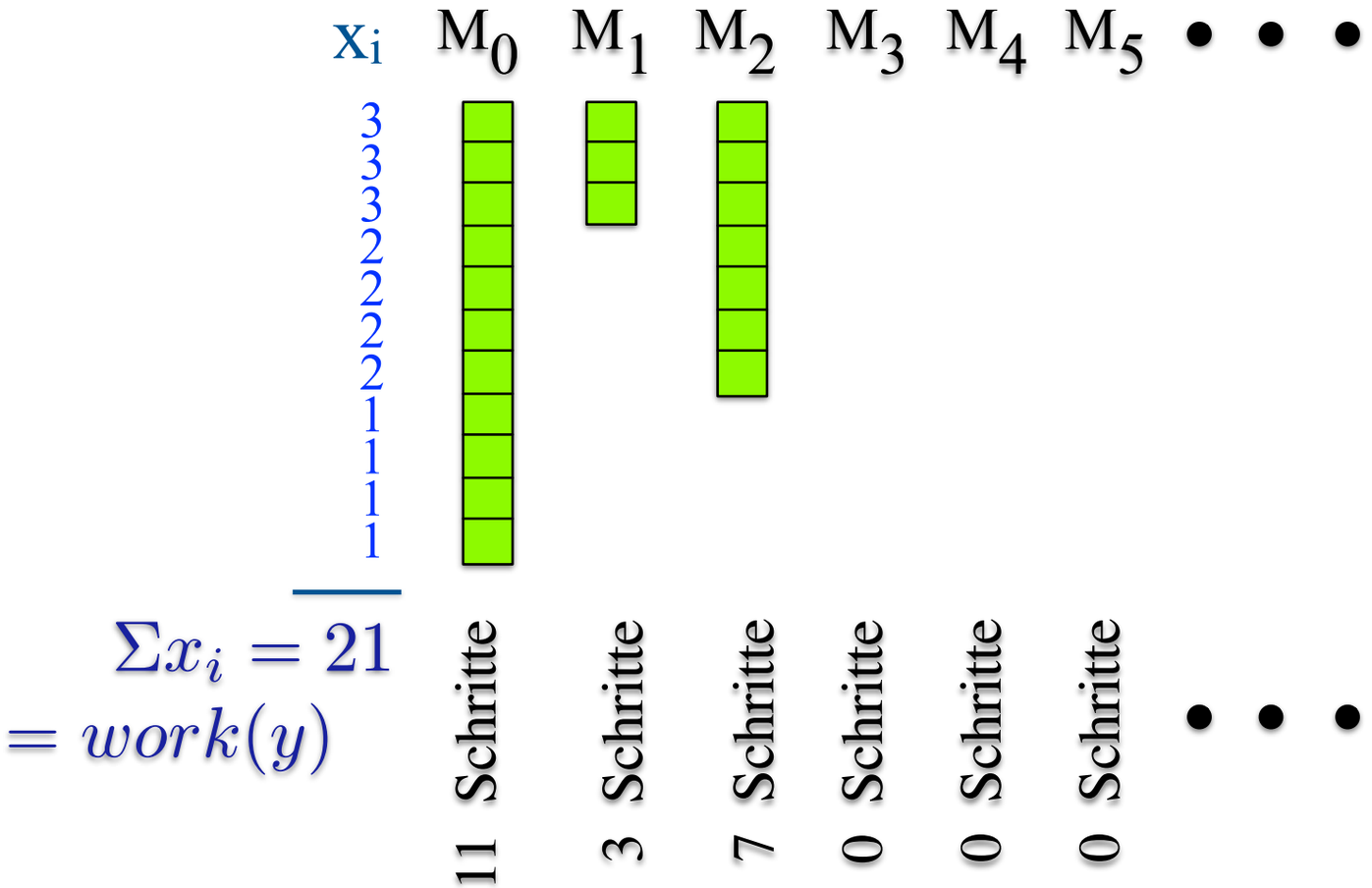
Satz 4.13 (Brent's scheduling principle) *Wenn man eine Berechnung auf $m = max\ x_i$ Prozessoren in Zeit t ausführen kann, wobei x_i die Anzahl der Operationen im i-ten Schritt ist, dann kann man auf*

$$p < m$$
$$2 \quad 3$$

Prozessoren dieselbe Berechnung in der Zeit

$$t + \lfloor \frac{x}{p} \rfloor = 11 + \lfloor \frac{21}{2} \rfloor = 11 + 10 = 21$$

ausführen, wobei $x = \sum x_i$



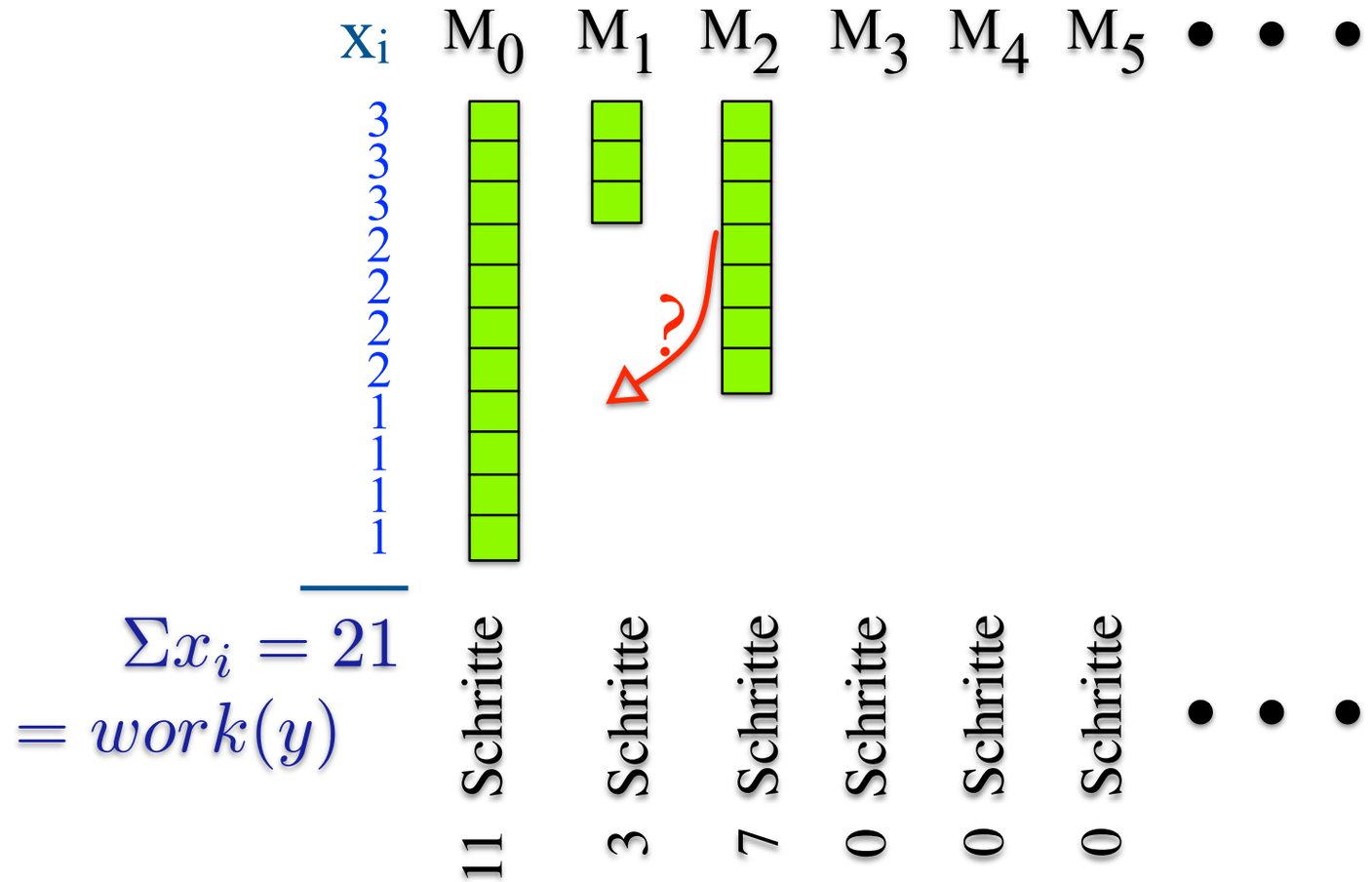
Satz 4.13 (Brent's scheduling principle) *Wenn man eine Berechnung auf $m = max\ x_i$ Prozessoren in Zeit t ausführen kann, wobei x_i die Anzahl der Operationen im i-ten Schritt ist, dann kann man auf*

$$p < m$$
$$2 \quad 3$$

Prozessoren dieselbe Berechnung in der Zeit

$$t + \lfloor \frac{x}{p} \rfloor = 11 + \lfloor \frac{21}{2} \rfloor = 11 + 10 = 21$$

ausführen, wobei $x = \sum x_i$



Satz 4.13 (Brent's scheduling principle) *Wenn man eine Berechnung auf $m = max\ x_i$ Prozessoren in Zeit t ausführen kann, wobei x_i die Anzahl der Operationen im i-ten Schritt ist, dann kann man auf*

1

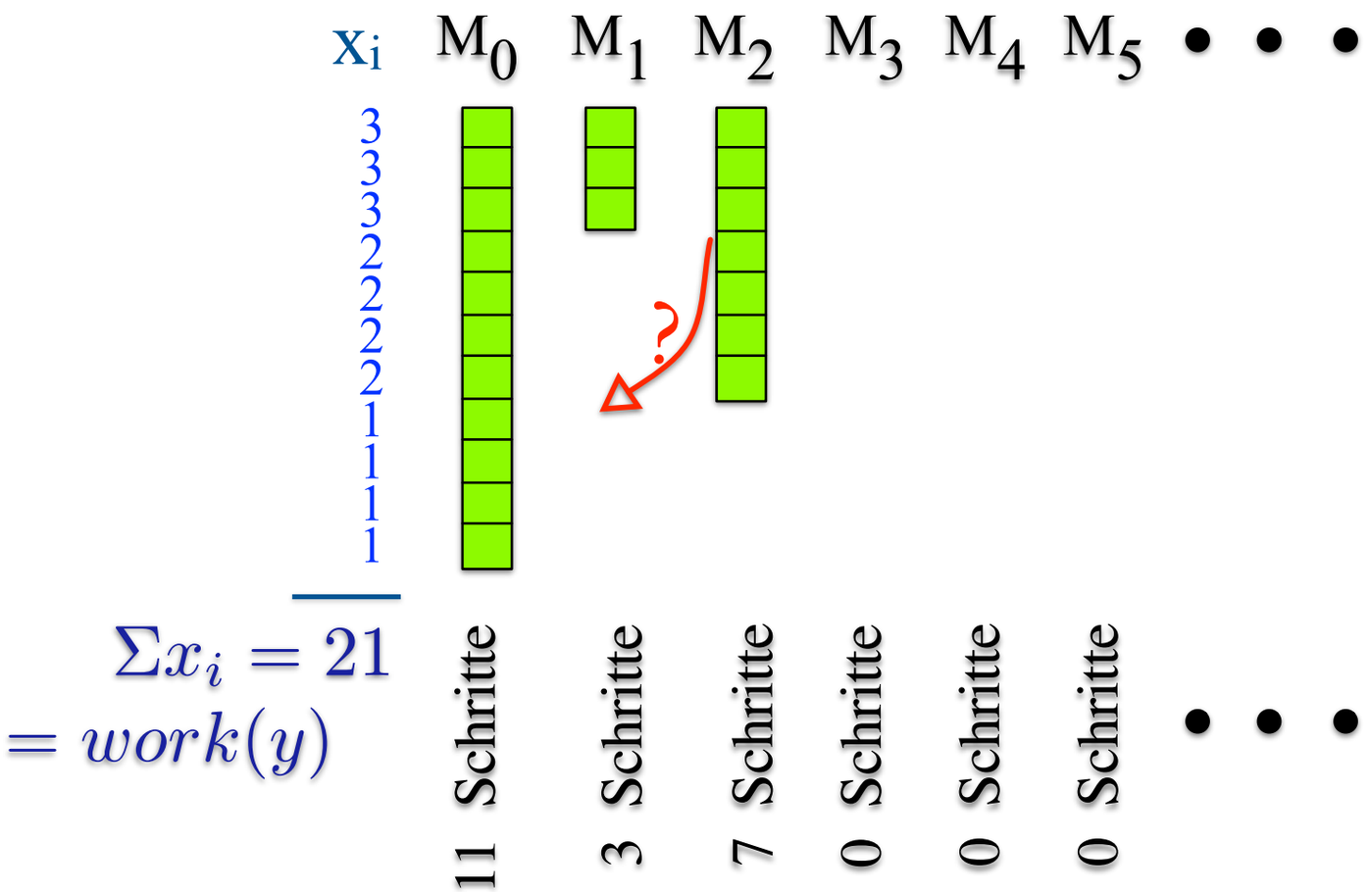
$p < m$

2 3

Prozessoren dieselbe Berechnung in der Zeit

$$t + \lfloor \frac{x}{p} \rfloor = 11 + \lfloor \frac{21}{2} \rfloor = 11 + 10 = 21$$

ausführen, wobei $x = \sum x_i$



Satz 4.13 (Brent's scheduling principle) *Wenn man eine Berechnung auf $m = max\ x_i$ Prozessoren in Zeit t ausführen kann, wobei x_i die Anzahl der Operationen im i-ten Schritt ist, dann kann man auf*

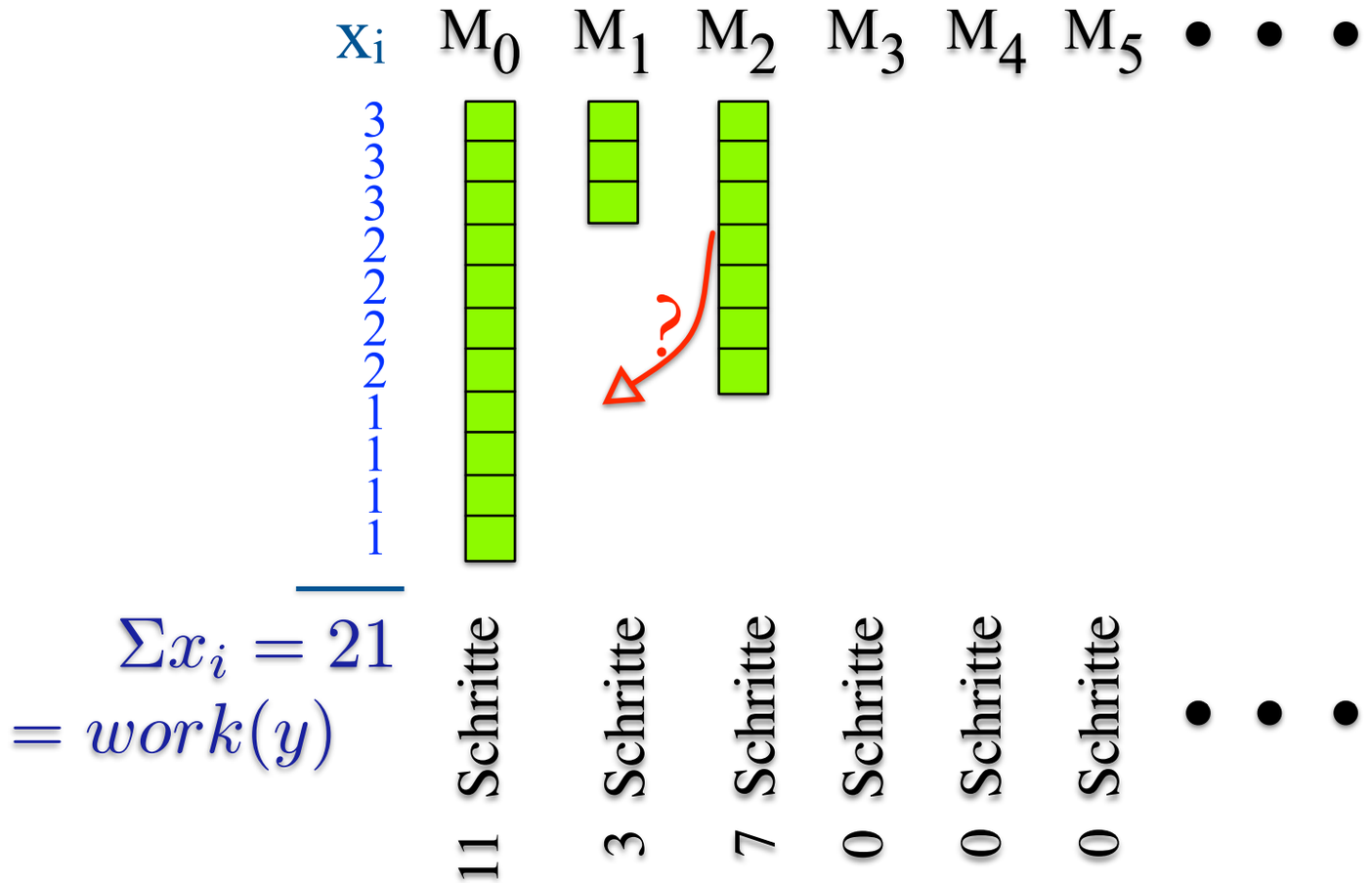
$$p < m$$

Prozessoren dieselbe Berechnung in der Zeit

$$t + \lfloor \frac{x}{p} \rfloor = 11 + \lfloor \frac{21}{2} \rfloor = 11 + 10 = 21$$

$$= 11 + \lfloor \frac{21}{1} \rfloor = 11 + 21 = 32$$

ausführen, wobei $x = \sum x_i$



Satz 4.13 (Brent's scheduling principle) *Wenn man eine Berechnung auf $m = max\ x_i$ Prozessoren in Zeit t ausführen kann, wobei x_i die Anzahl der Operationen im i-ten Schritt ist, dann kann man auf*

1

$p < m$

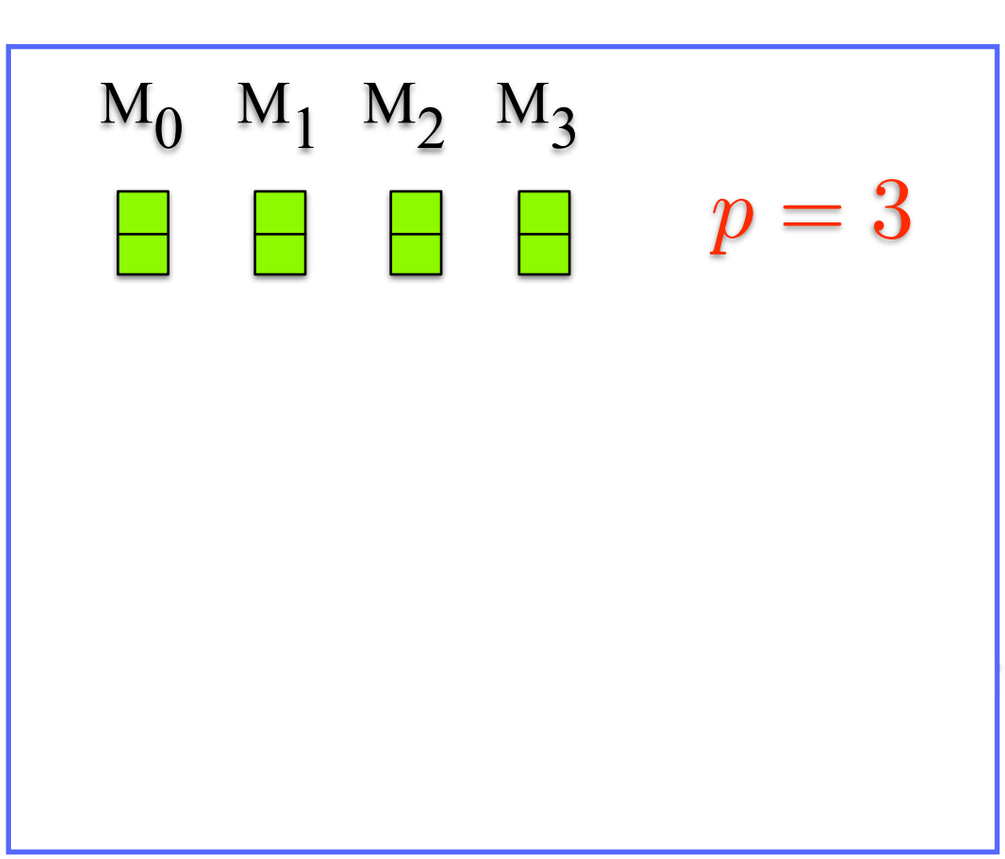
2 3

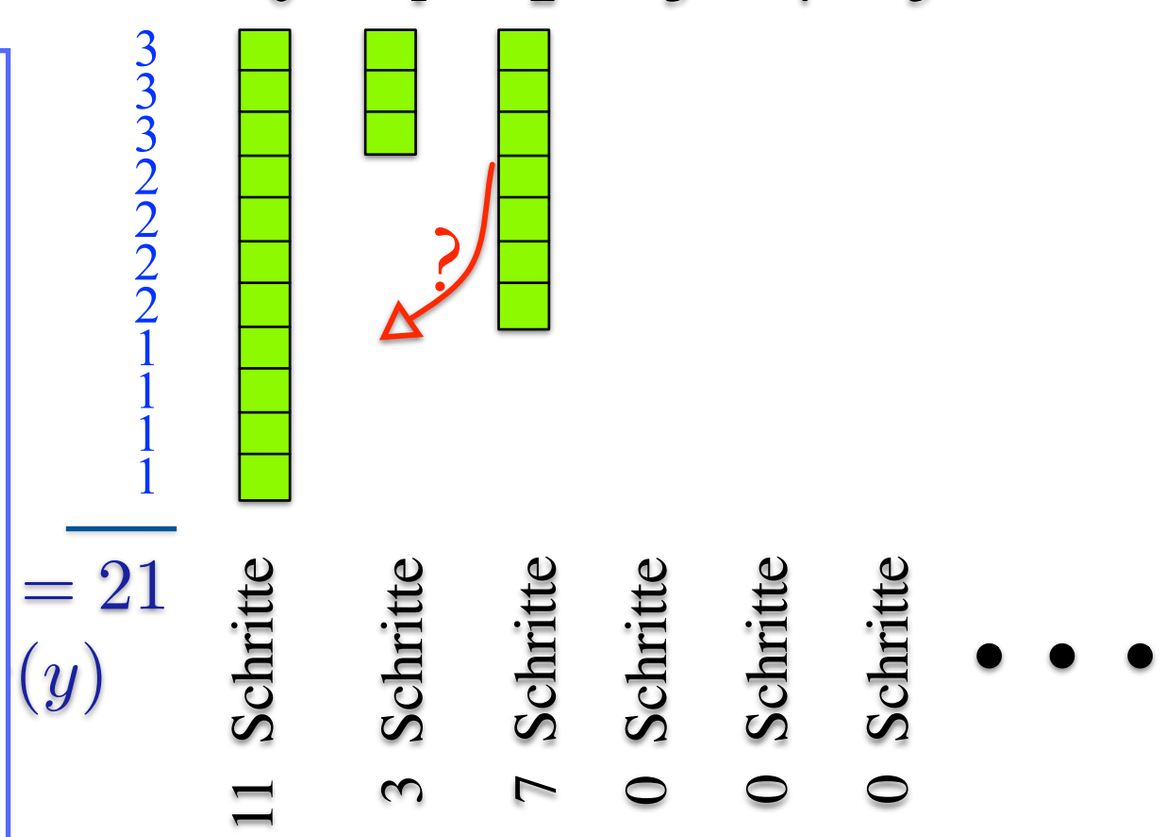
Prozessoren dieselbe Berechnung in der Zeit

$$t + \lfloor \frac{x}{p} \rfloor = 11 + \lfloor \frac{21}{2} \rfloor = 11 + 10 = 21$$

$$= 11 + \lfloor \frac{21}{1} \rfloor = 11 + 21 = 32$$

ausführen, wobei $x = \sum x_i$





Satz 4.13 (Brent's scheduling principle) *Wenn man eine Berechnung auf $m = max\ x_i$ Prozessoren in Zeit t ausführen kann, wobei x_i die Anzahl der Operationen im i-ten Schritt ist, dann kann man auf*

1

$p < m$

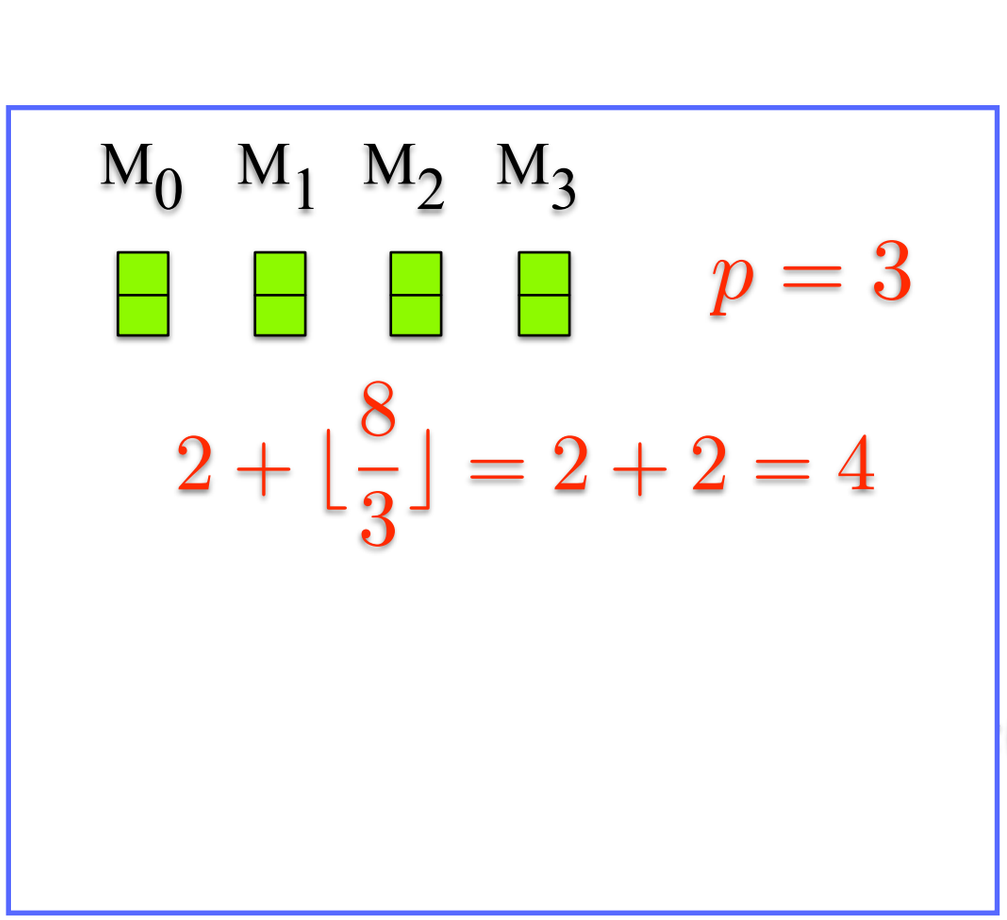
2 3

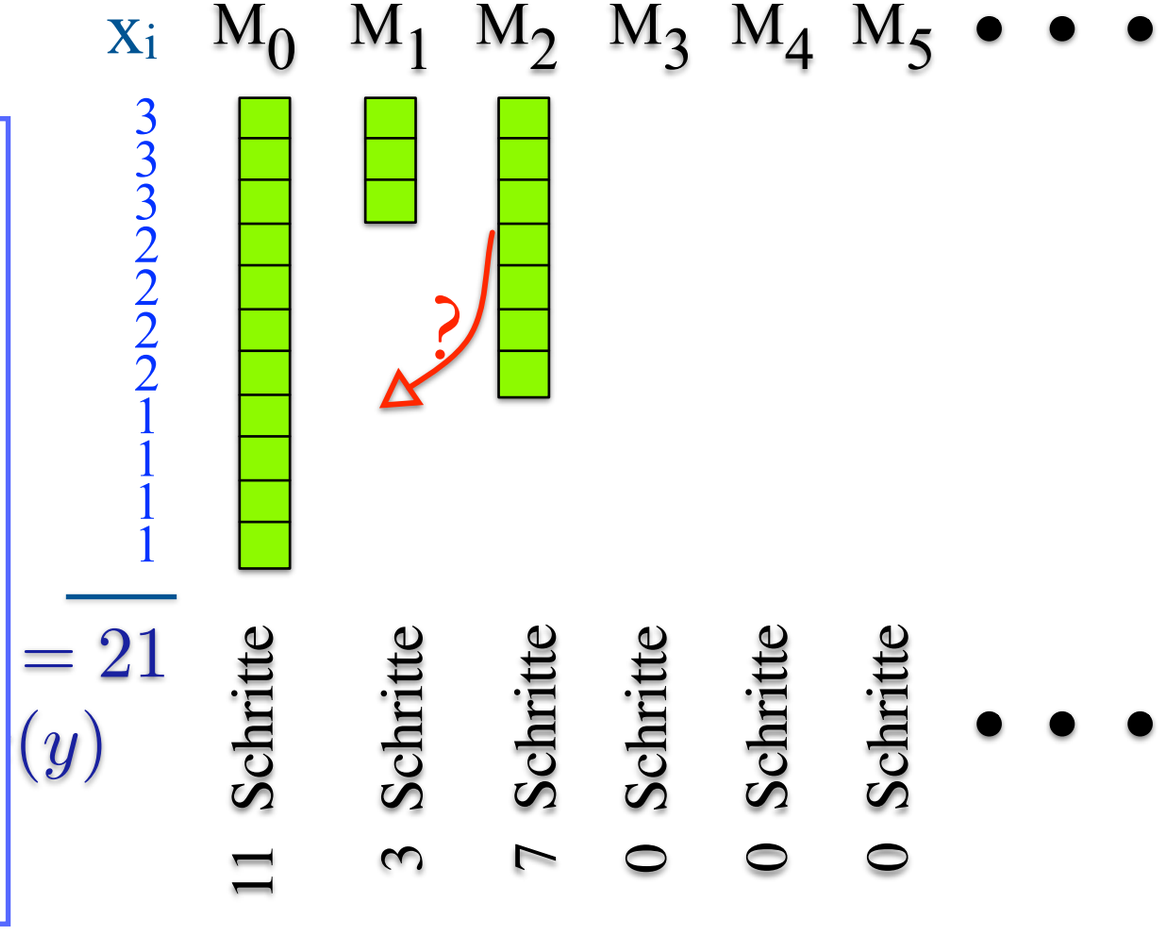
Prozessoren dieselbe Berechnung in der Zeit

$$t + \lfloor \frac{x}{p} \rfloor = 11 + \lfloor \frac{21}{2} \rfloor = 11 + 10 = 21$$

$$= 11 + \lfloor \frac{21}{1} \rfloor = 11 + 21 = 32$$

ausführen, wobei $x = \sum x_i$





Satz 4.13 (Brent's scheduling principle) *Wenn man eine Berechnung auf $m = max\ x_i$ Prozessoren in Zeit t ausführen kann, wobei x_i die Anzahl der Operationen im i-ten Schritt ist, dann kann man auf*

1

$p < m$

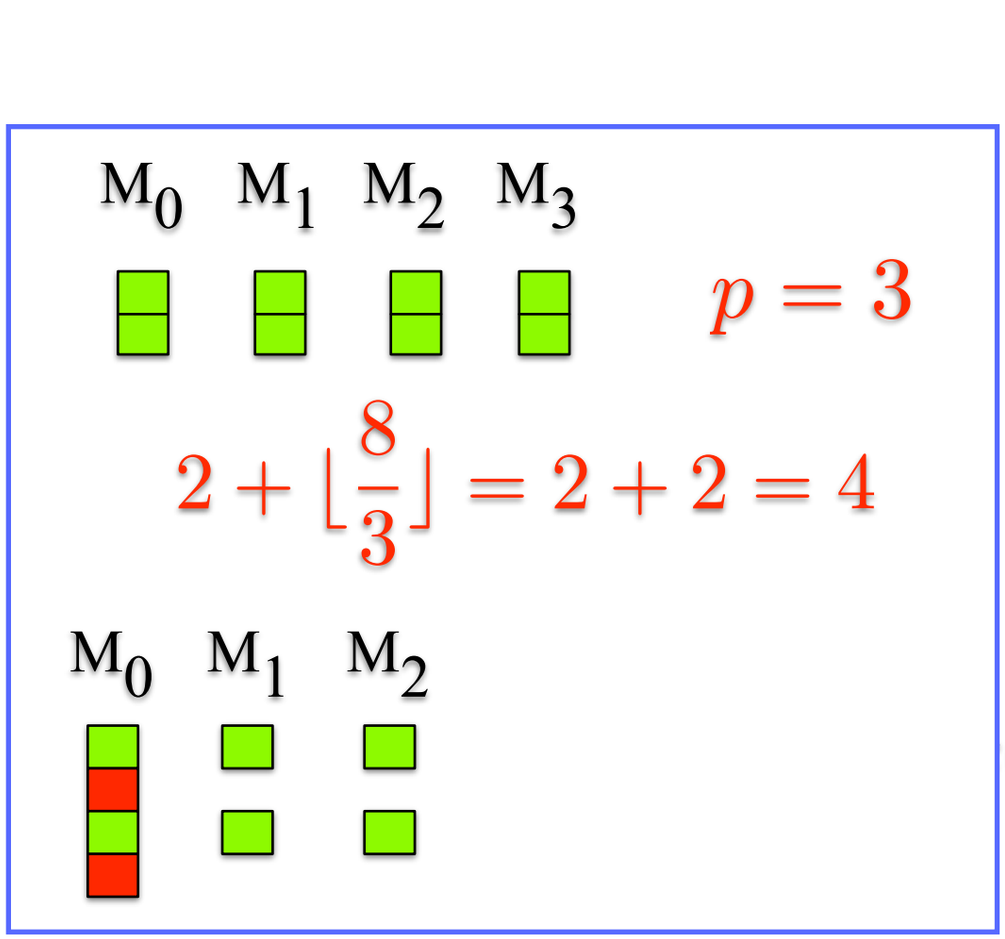
2 3

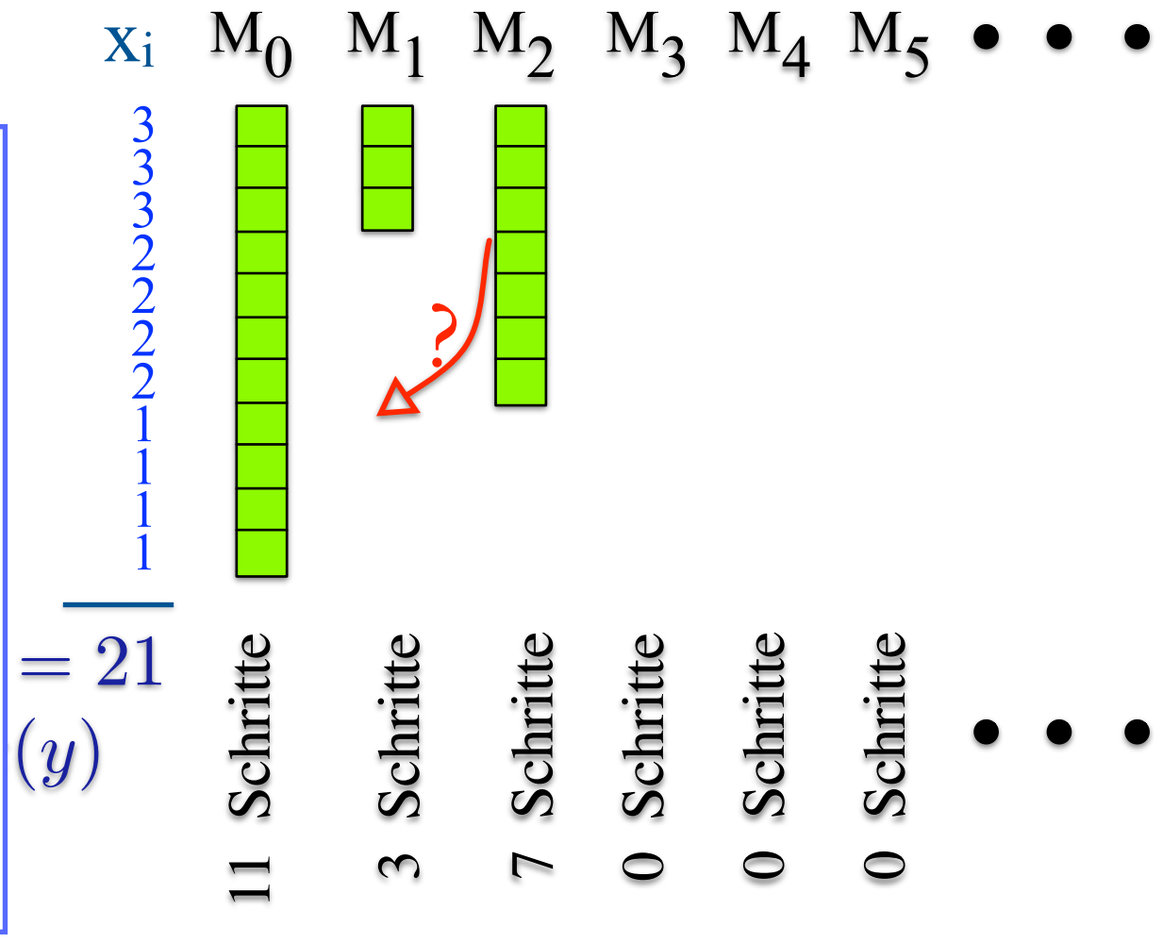
Prozessoren dieselbe Berechnung in der Zeit

$$t + \lfloor \frac{x}{p} \rfloor = 11 + \lfloor \frac{21}{2} \rfloor = 11 + 10 = 21$$

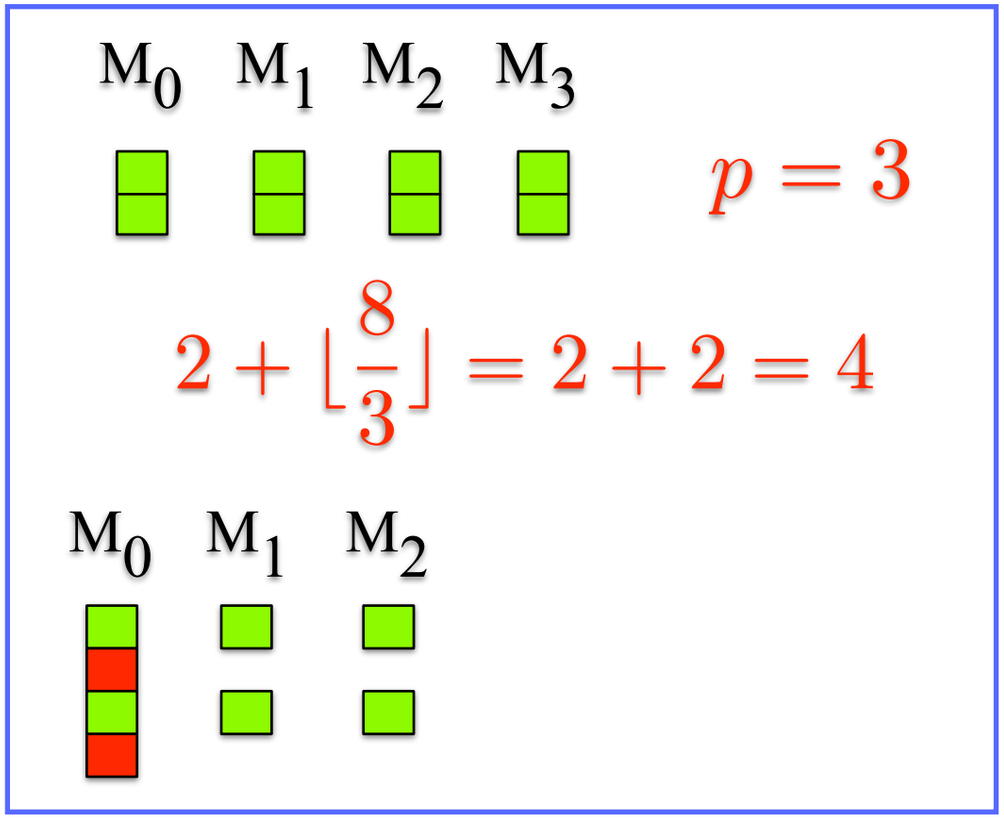
$$= 11 + \lfloor \frac{21}{1} \rfloor = 11 + 21 = 32$$

ausführen, wobei $x = \sum x_i$

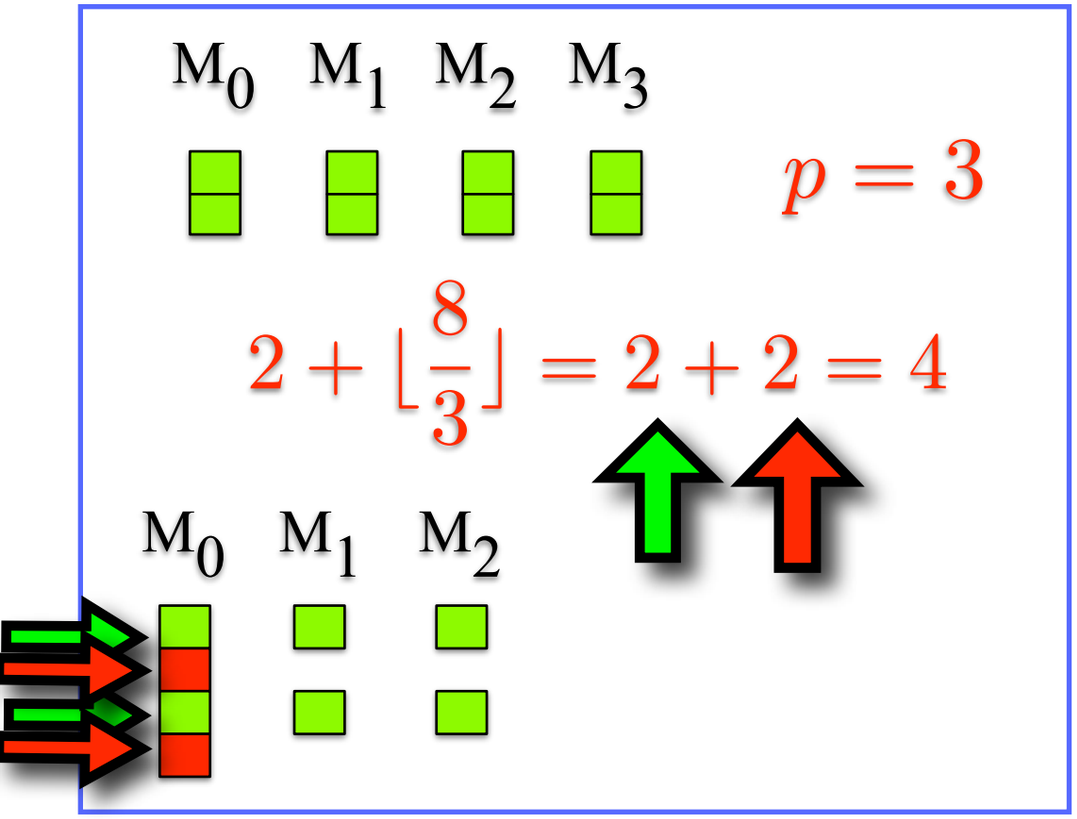




$$\sum_{1}^{t} \lceil \frac{x_i}{p} \rceil = \lfloor \sum_{1}^{t} \lceil \frac{x_i}{p} \rceil \rfloor \leq \lfloor \sum_{1}^{t} (\frac{x_i}{p} + 1) \rfloor = t + \lfloor \frac{x}{p} \rfloor$$



$$\sum_{1}^{t}\lceil\frac{x_i}{p}\rceil = \lfloor\sum_{1}^{t}\lceil\frac{x_i}{p}\rceil\rfloor \leq \lfloor\sum_{1}^{t}(\frac{x_i}{p}+1)\rfloor = t + \lfloor\frac{x}{p}\rfloor$$



Satz 4.13 (Brent's scheduling principle) *Wenn man eine Berechnung auf $m = max\ x_i$ Prozessoren in Zeit t ausführen kann, wobei x_i die Anzahl der Operationen im i-ten Schritt ist, dann kann man auf*

$$p < m$$

Prozessoren dieselbe Berechnung in der Zeit

$$t + \lfloor \frac{x}{p} \rfloor$$

ausführen, wobei $x = \sum x_i$

Korollar 4.14 *Wenn man eine Berechnung, die $O(n)$ sequentielle Zeit braucht, auf n Prozessoren in $O(\log n)$ Zeit ausführen kann, dann kann man diese Berechnung auf $O\left(\frac{n}{\log n}\right)$ Prozessoren in Zeit $O(\log n)$ ausführen (d. h. in* optimaler Zeit: $O(n) = O(\log n) \cdot O\left(\frac{n}{\log n}\right)$*).*

Satz 4.13 (Brent's scheduling principle) *Wenn man eine Berechnung auf* $m = max\ x_i$ *Prozessoren in Zeit* t *ausführen kann, wobei* x_i *die Anzahl der Operationen im* i*-ten Schritt ist, dann kann man auf*

$$p < m$$

Prozessoren dieselbe Berechnung in der Zeit

$$t + \lfloor \frac{x}{p} \rfloor$$

ausführen, wobei $x = \sum x_i$

$O(n)$

Korollar 4.14 *Wenn man eine Berechnung, die* $O(n)$ *sequentielle Zeit braucht, auf* n *Prozessoren in* $O(\log n)$ *Zeit ausführen kann, dann kann man diese Berechnung auf* $O\left(\frac{n}{\log n}\right)$ *Prozessoren in Zeit* $O(\log n)$ *ausführen (d. h. in* optimaler Zeit: $O(n) = O(\log n) \cdot O\left(\frac{n}{\log n}\right)$*).*

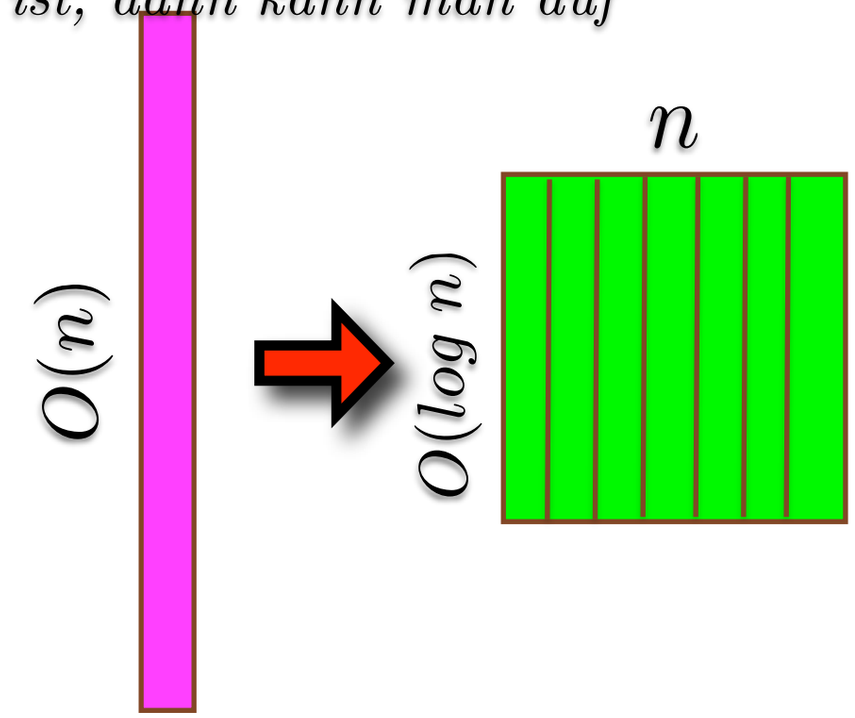
Satz 4.13 (Brent's scheduling principle) *Wenn man eine Berechnung auf* $m = max\ x_i$ *Prozessoren in Zeit* t *ausführen kann, wobei* x_i *die Anzahl der Operationen im* i*-ten Schritt ist, dann kann man auf*

$$p < m$$

Prozessoren dieselbe Berechnung in der Zeit

$$t + \lfloor \frac{x}{p} \rfloor$$

ausführen, wobei $x = \sum x_i$



Korollar 4.14 *Wenn man eine Berechnung, die* $O(n)$ *sequentielle Zeit braucht, auf* n *Prozessoren in* $O(\log n)$ *Zeit ausführen kann, dann kann man diese Berechnung auf* $O\left(\frac{n}{\log n}\right)$ *Prozessoren in Zeit* $O(\log n)$ *ausführen (d. h. in* optimaler Zeit: $O(n) = O(\log n) \cdot O\left(\frac{n}{\log n}\right)$*).*

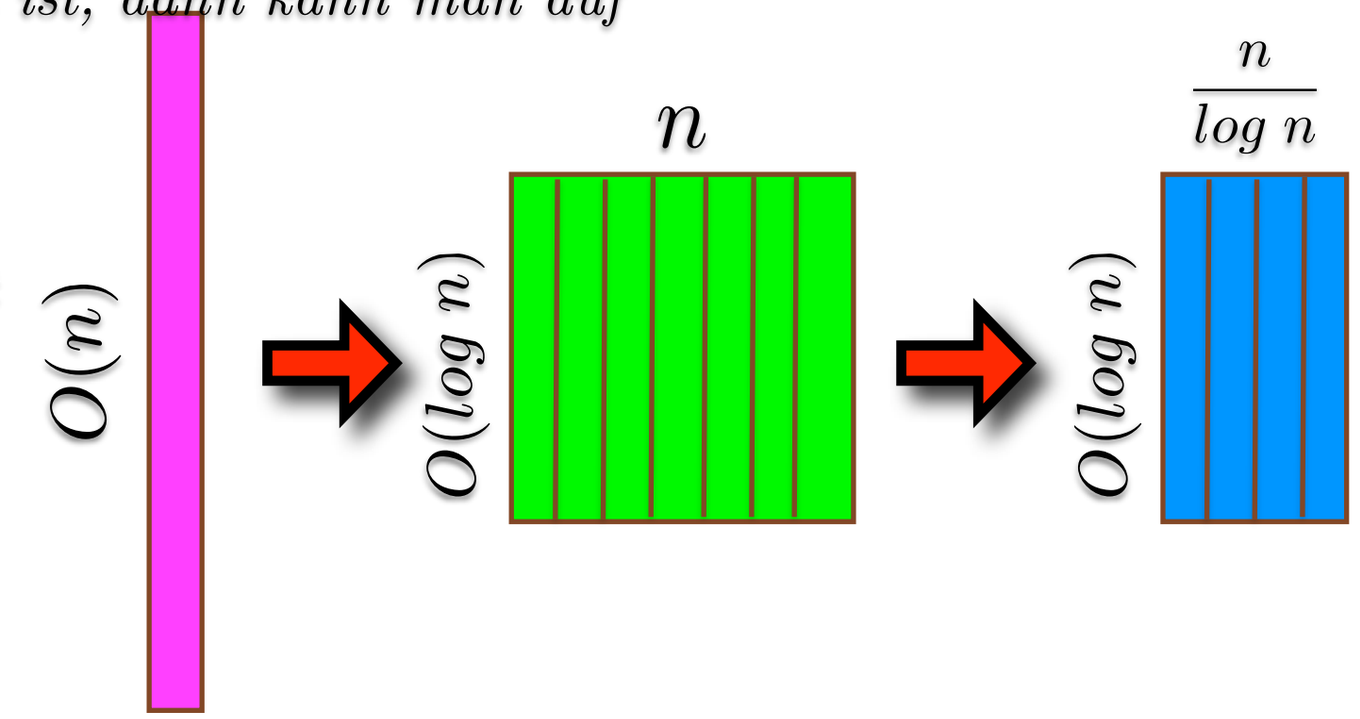
Satz 4.13 (Brent's scheduling principle) *Wenn man eine Berechnung auf* $m = max\ x_i$ *Prozessoren in Zeit* t *ausführen kann, wobei* x_i *die Anzahl der Operationen im i-ten Schritt ist, dann kann man auf*

$$p < m$$

Prozessoren dieselbe Berechnung in der Zeit

$$t + \lfloor \frac{x}{p} \rfloor$$

ausführen, wobei $x = \sum x_i$



Korollar 4.14 *Wenn man eine Berechnung, die* $O(n)$ *sequentielle Zeit braucht, auf* n *Prozessoren in* $O(\log n)$ *Zeit ausführen kann, dann kann man diese Berechnung auf* $O\left(\frac{n}{\log n}\right)$ *Prozessoren in Zeit* $O(\log n)$ *ausführen (d. h. in* optimaler Zeit: $O(n) = O(\log n) \cdot O\left(\frac{n}{\log n}\right)$*).*

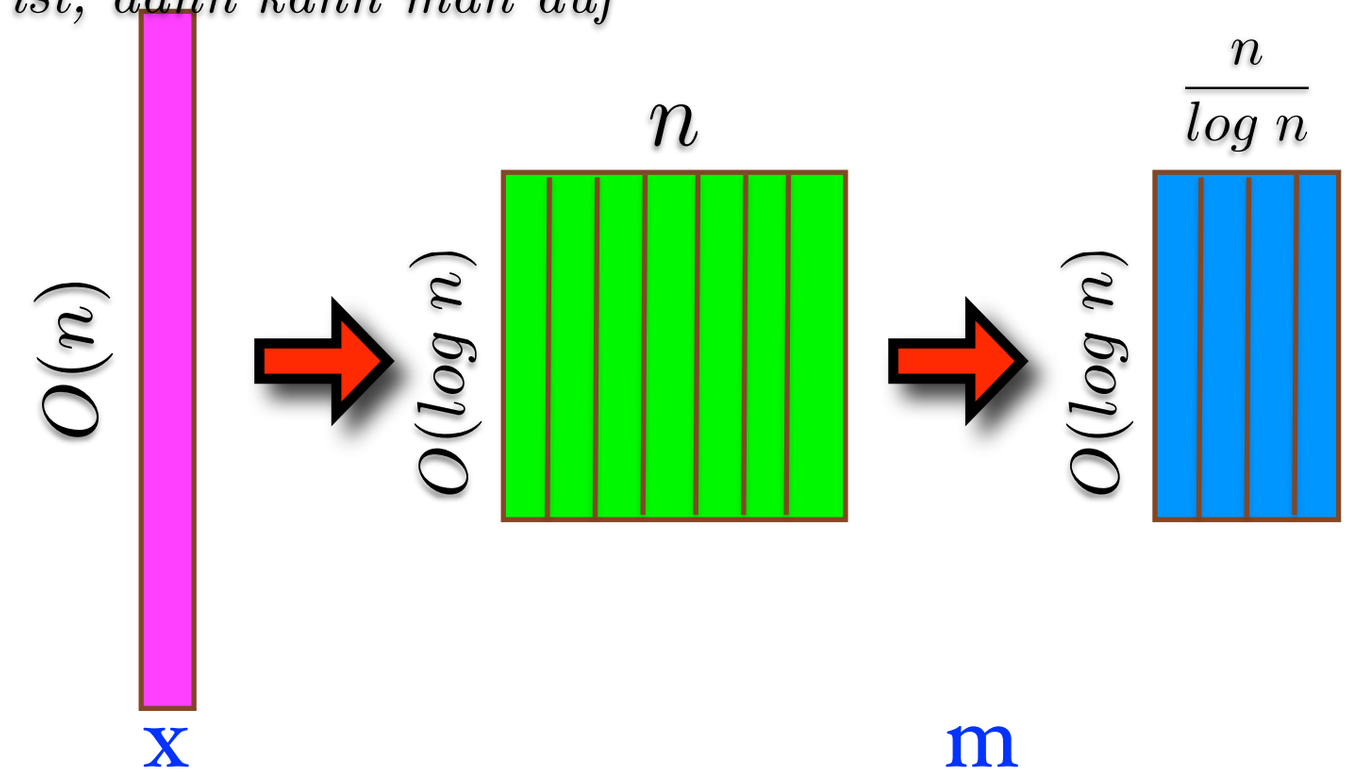
Satz 4.13 (Brent's scheduling principle) *Wenn man eine Berechnung auf* $m = max\ x_i$ *Prozessoren in Zeit* t *ausführen kann, wobei* x_i *die Anzahl der Operationen im i-ten Schritt ist, dann kann man auf*

$$p < m$$

Prozessoren dieselbe Berechnung in der Zeit

$$t + \lfloor \frac{x}{p} \rfloor$$

ausführen, wobei $x = \sum x_i$



Korollar 4.14 *Wenn man eine Berechnung, die* $O(n)$ *sequentielle Zeit braucht, auf* n *Prozessoren in* $O(\log n)$ *Zeit ausführen kann, dann kann man diese Berechnung auf* $O\left(\frac{n}{\log n}\right)$ *Prozessoren in Zeit* $O(\log n)$ *ausführen (d. h. in* optimaler Zeit: $O(n) = O(\log n) \cdot O\left(\frac{n}{\log n}\right)$*).*

t

p

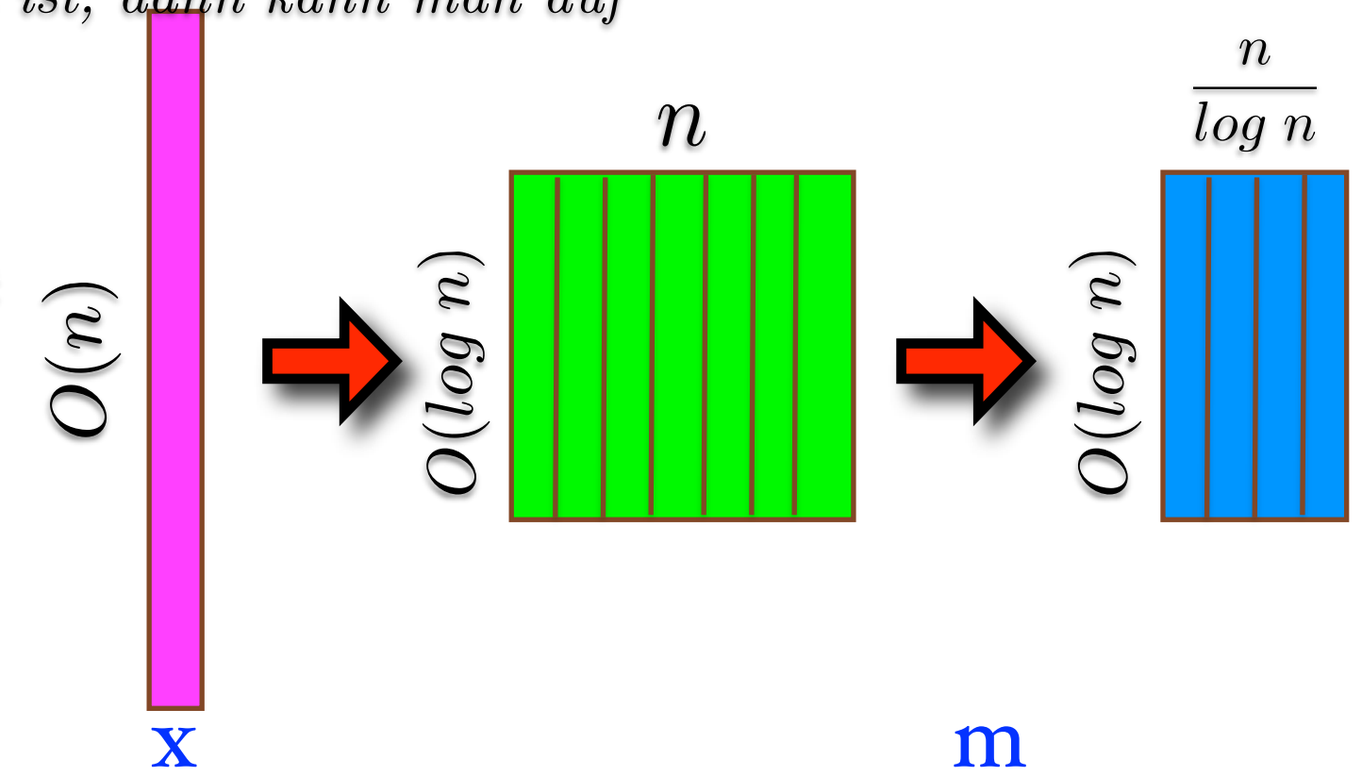
Satz 4.13 (Brent's scheduling principle) *Wenn man eine Berechnung auf $m = max\ x_i$ Prozessoren in Zeit t ausführen kann, wobei x_i die Anzahl der Operationen im i-ten Schritt ist, dann kann man auf*

$$p < m$$

Prozessoren dieselbe Berechnung in der Zeit

$$t + \lfloor \frac{x}{p} \rfloor$$

ausführen, wobei $x = \sum x_i$



Korollar 4.14 *Wenn man eine Berechnung, die $O(n)$ sequentielle Zeit braucht, auf n Prozessoren in $O(\log n)$ Zeit ausführen kann, dann kann man diese Berechnung auf $O\left(\frac{n}{\log n}\right)$ Prozessoren in Zeit $O(\log n)$ ausführen (d. h. in* optimaler Zeit: $O(n) = O(\log n) \cdot O\left(\frac{n}{\log n}\right)$*).*

t

p

$$t + \lfloor \frac{x}{p} \rfloor = log\ n + \lfloor \frac{n}{\frac{n}{log\ n}} \rfloor \in O(log\ n)$$

4.2.4 Beispiele für PRAM-Algorithmen

Um die Algorithmen zu beschreiben, benutzen wir den parallelen Ausdruck

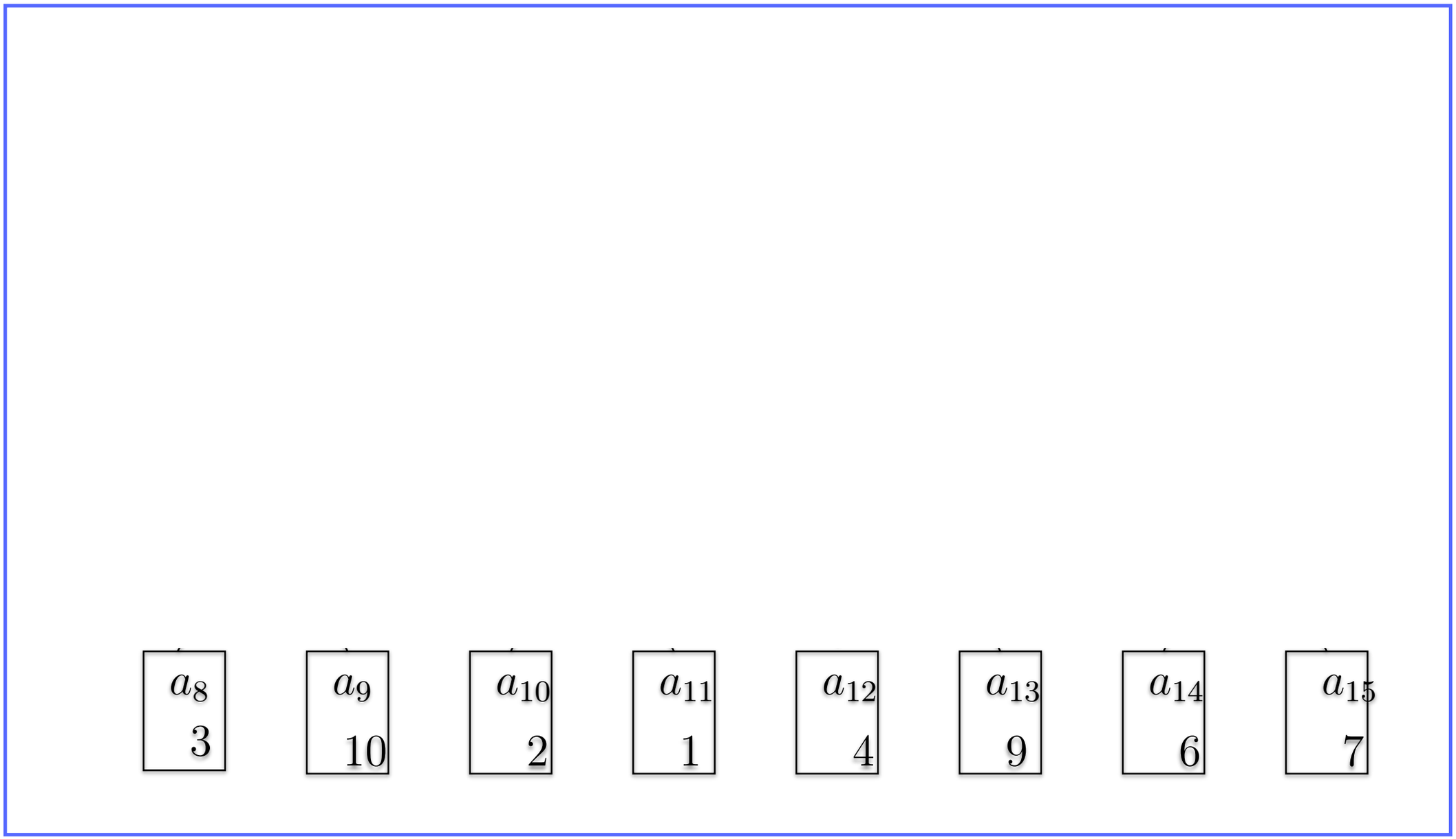
$$\underline{\text{par}}\ [a \leq i \leq b]\ S_i$$

um zu beschreiben, dass die Anweisungen S_i für alle i mit $a \leq i \leq b$ parallel ausgeführt werden sollen.

Beispiel 4.15 Maximum finden *Gegeben:* $n = 2^m$ natürliche Zahlen $a_n, a_{n+1}, \ldots, a_{2n-1}$. Gesucht wird ihr Maximum.

Algorithmus (für EREW-PRAM):

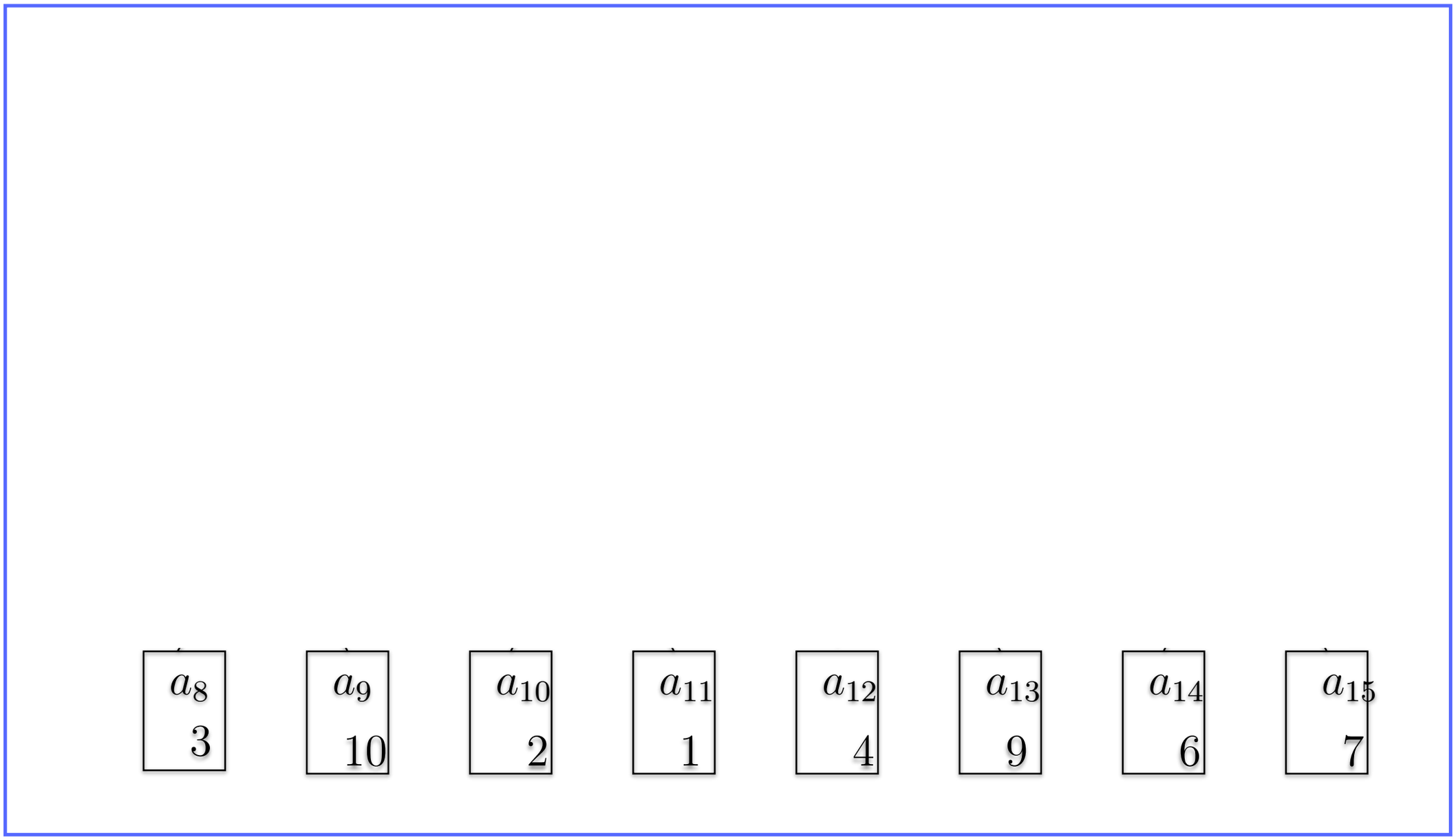
$$\underline{\text{for}}\ l \leftarrow m-1\ \underline{\text{down to}}\ 0\ \underline{\text{do}}$$
$$\quad \underline{\text{par}}\ [2^l \leq j \leq 2^{l+1}-1]\ a_j \leftarrow max\{a_{2j}, a_{2j+1}\}$$



Beispiel 4.15 Maximum finden *Gegeben:* $n = 2^m$ natürliche Zahlen $a_n, a_{n+1}, \ldots, a_{2n-1}$. Gesucht wird ihr Maximum.

Algorithmus (für EREW-PRAM):

$$\underline{\text{for}}\ l \leftarrow m-1\ \underline{\text{down to}}\ 0\ \underline{\text{do}}$$
$$\quad \underline{\text{par}}\ [2^l \leq j \leq 2^{l+1}-1]\ a_j \leftarrow max\{a_{2j}, a_{2j+1}\}$$



Beispiel 4.15 Maximum finden *Gegeben:* $n = 2^m$ natürliche Zahlen $a_n, a_{n+1}, \ldots, a_{2n-1}$. Gesucht wird ihr Maximum.

m= 3

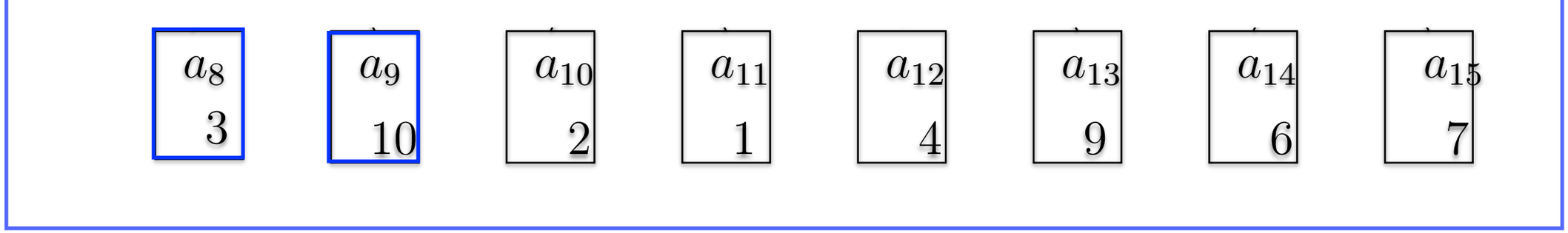
l= 2

j= 4

2j= 8

Algorithmus (für EREW-PRAM):

$$\underline{\text{for}}\ l \leftarrow m-1\ \underline{\text{down to}}\ 0\ \underline{\text{do}}$$
$$\underline{\text{par}}\ [2^l \leq j \leq 2^{l+1}-1]\ a_j \leftarrow max\{a_{2j}, a_{2j+1}\}$$



Beispiel 4.15 Maximum finden *Gegeben:* $n = 2^m$ natürliche Zahlen $a_n, a_{n+1}, \ldots, a_{2n-1}$. Gesucht wird ihr Maximum.

m= 3

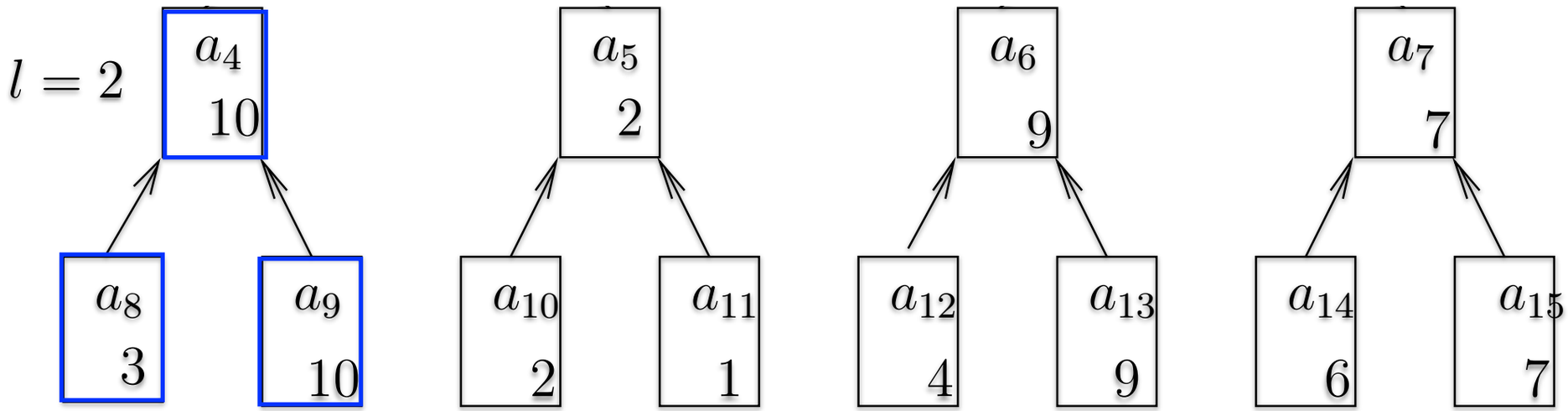
l= 2

j= 4

2j= 8

Algorithmus (für EREW-PRAM):

$$\underline{\text{for}}\ l \leftarrow m-1\ \underline{\text{down to}}\ 0\ \underline{\text{do}}$$
$$\underline{\text{par}}\ [2^l \leq j \leq 2^{l+1}-1]\ a_j \leftarrow max\{a_{2j}, a_{2j+1}\}$$



Beispiel 4.15 Maximum finden *Gegeben:* $n = 2^m$ natürliche Zahlen $a_n, a_{n+1}, \ldots, a_{2n-1}$. Gesucht wird ihr Maximum.

m= 3

l= 2

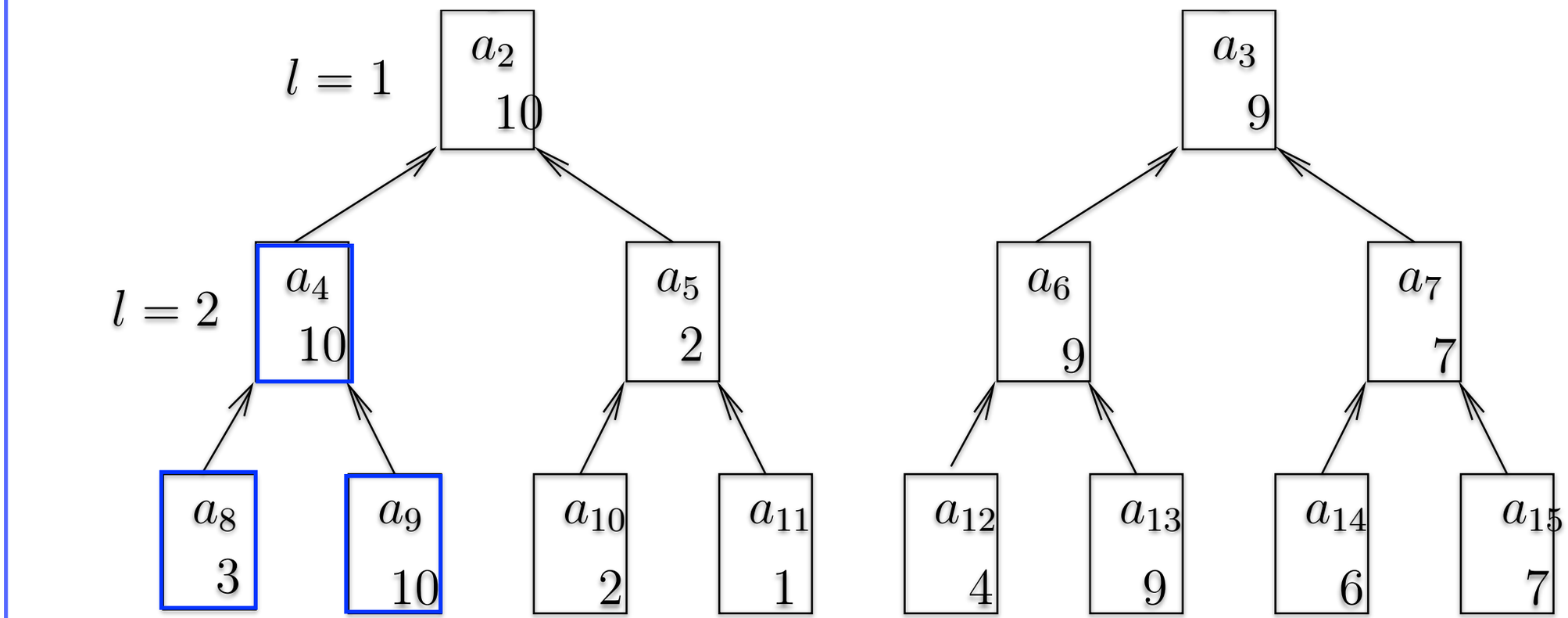
j= 4

2j= 8

Algorithmus (für EREW-PRAM):

$\underline{\text{for}}\ l \leftarrow m-1\ \underline{\text{down to}}\ 0\ \underline{\text{do}}$

$\quad \underline{\text{par}}\ [2^l \leq j \leq 2^{l+1}-1]\ a_j \leftarrow max\{a_{2j}, a_{2j+1}\}$



Beispiel 4.15 Maximum finden *Gegeben:* $n = 2^m$ natürliche Zahlen $a_n, a_{n+1}, \ldots, a_{2n-1}$. Gesucht wird ihr Maximum.

m= 3

l= 2

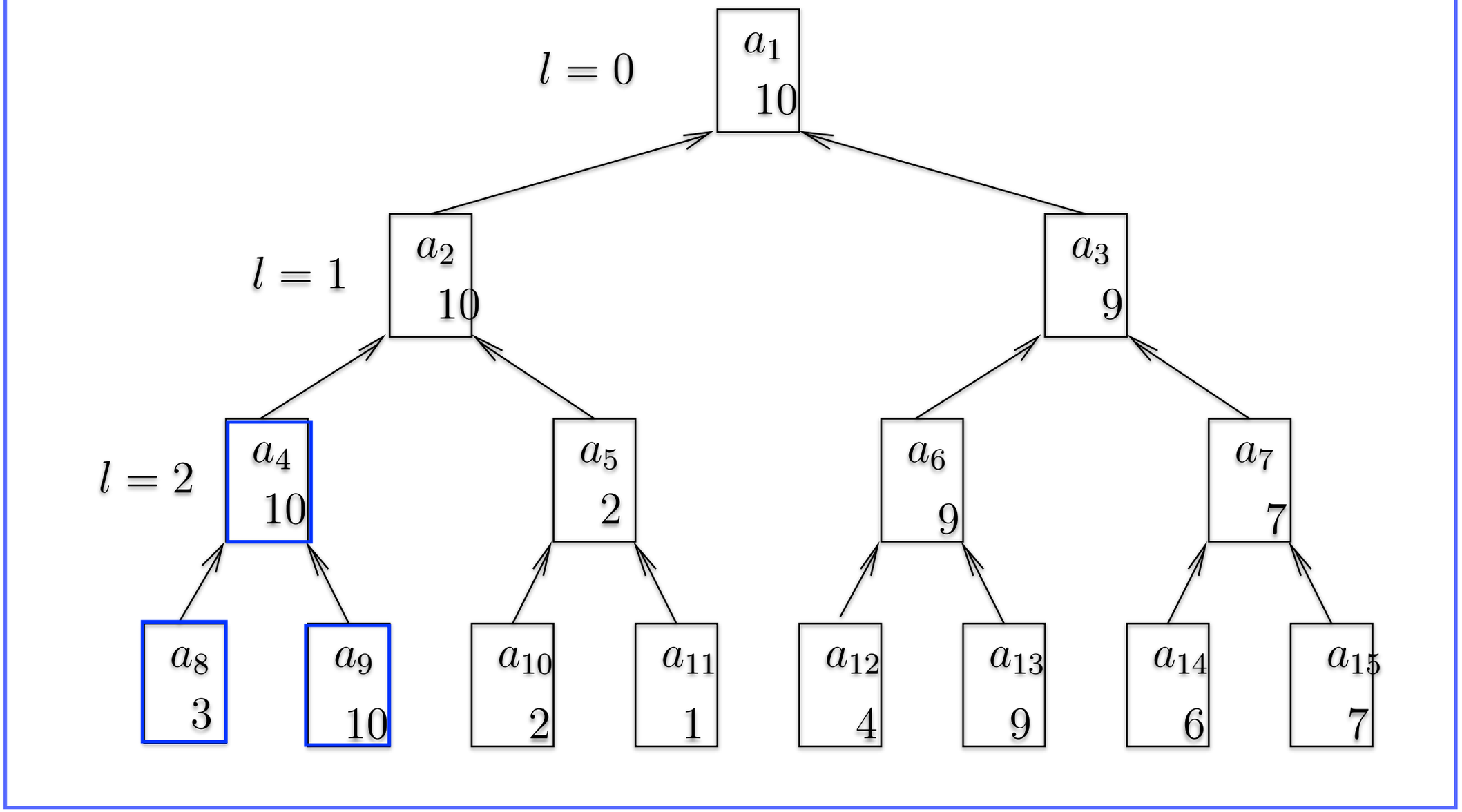
j= 4

2j= 8

Algorithmus (für EREW-PRAM):

$\underline{\text{for}}\ l \leftarrow m-1\ \underline{\text{down to}}\ 0\ \underline{\text{do}}$

$\quad \underline{\text{par}}\ [2^l \leq j \leq 2^{l+1}-1]\ a_j \leftarrow max\{a_{2j}, a_{2j+1}\}$



Beispiel 4.15 Maximum finden *Gegeben:* $n = 2^m$ natürliche Zahlen $a_n, a_{n+1}, \ldots, a_{2n-1}$. Gesucht wird ihr Maximum.

m= 3

l= 2

j= 4

2j= 8

Algorithmus (für EREW-PRAM):

$\underline{\text{for}}\ l \leftarrow m-1\ \underline{\text{down to}}\ 0\ \underline{\text{do}}$

$\quad \underline{\text{par}}\ [2^l \leq j \leq 2^{l+1}-1]\ a_j \leftarrow max\{a_{2j}, a_{2j+1}\}$

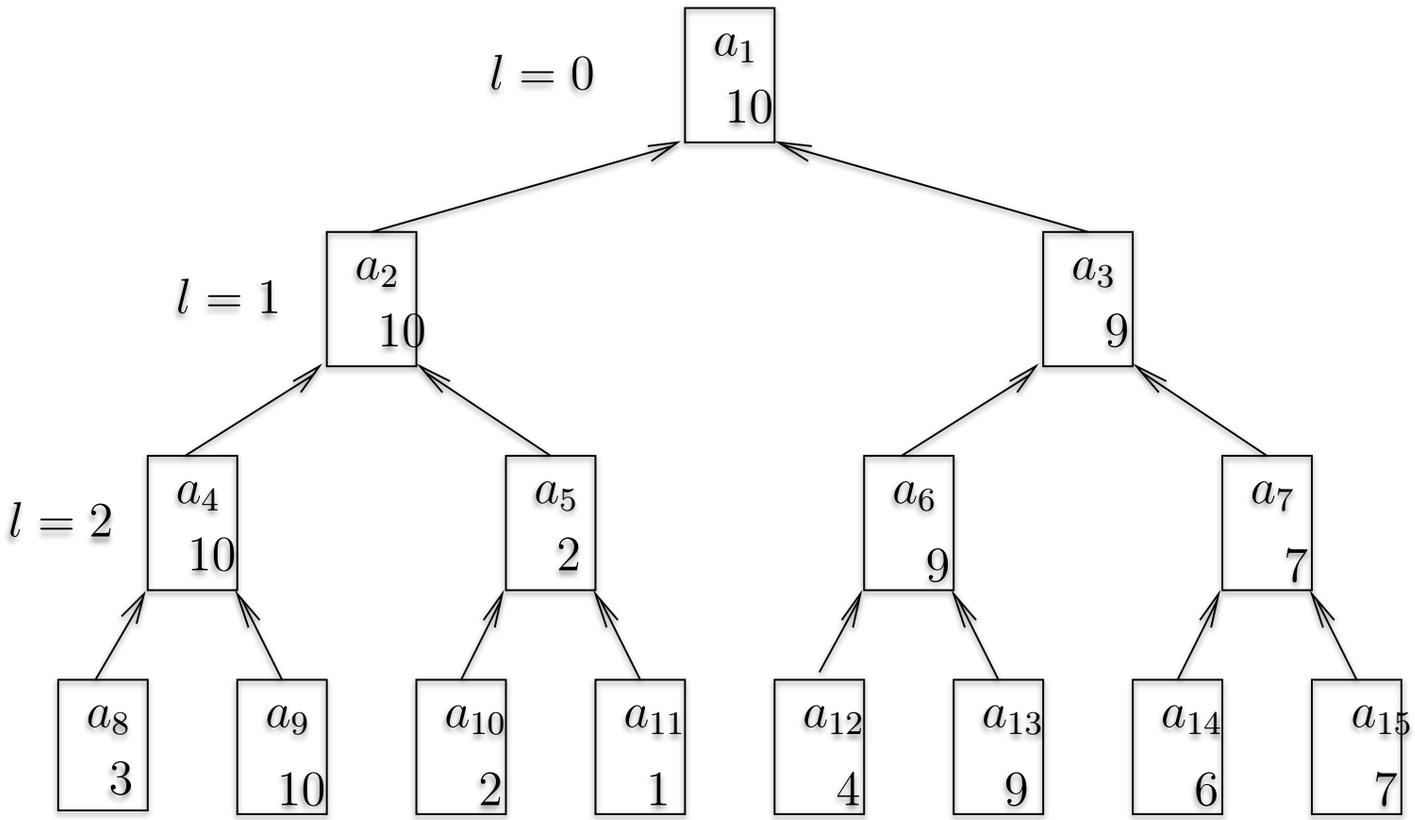


Abbildung 7.9: Maximum finden durch die PRAM

Korollar 4.14 *Wenn man eine Berechnung, die $O(n)$ sequentielle Zeit braucht, auf n Prozessoren in $O(\log n)$ Zeit ausführen kann, dann kann man diese Berechnung auf $O\left(\frac{n}{\log n}\right)$ Prozessoren in Zeit $O(\log n)$ ausführen (d. h. in* optimaler Zeit*: $O(n) = O(\log n) \cdot O\left(\frac{n}{\log n}\right)$).*

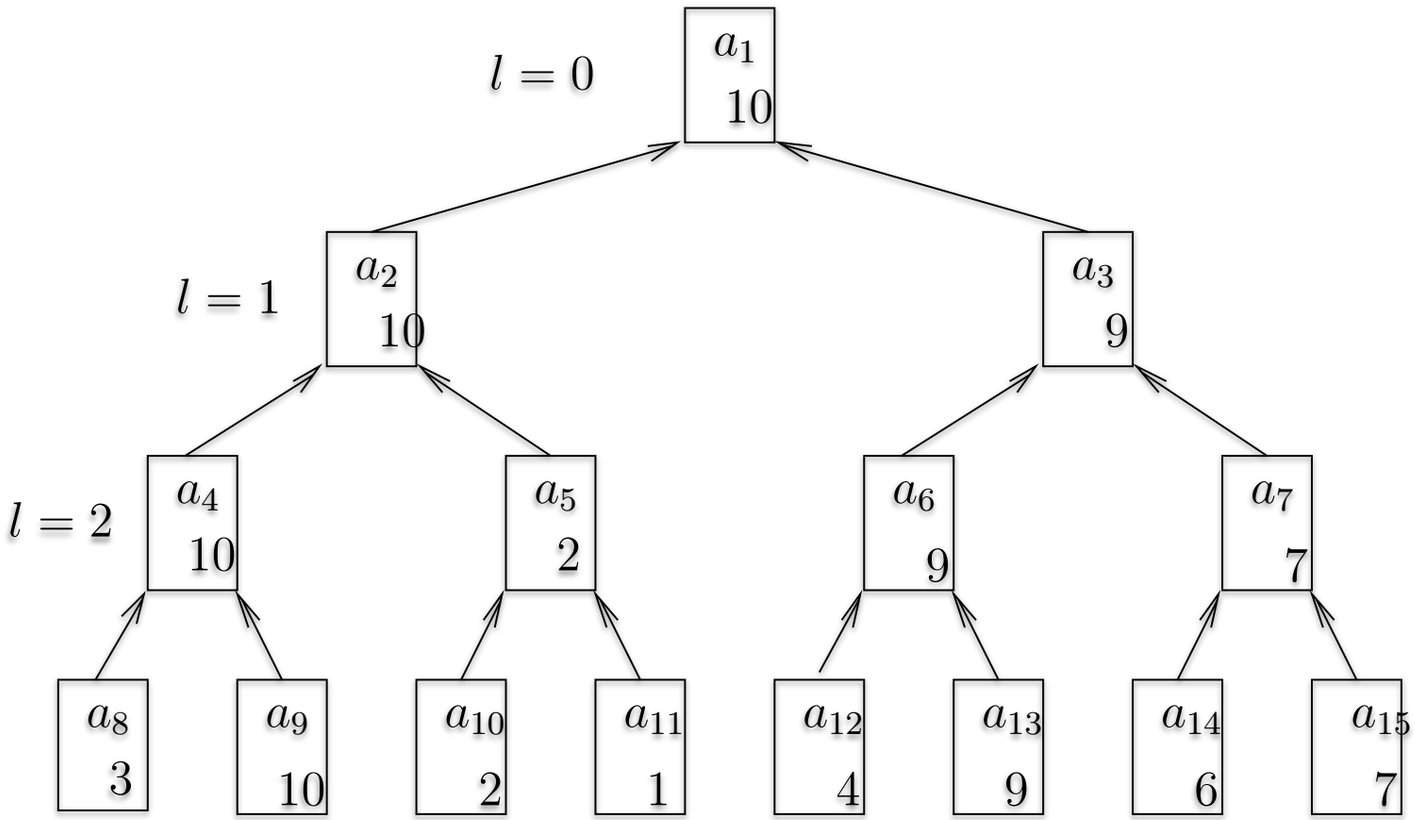


Abbildung 7.9: Maximum finden durch die PRAM

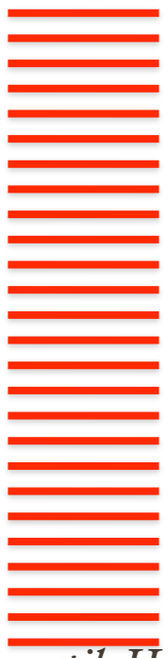
$$n = 8$$

Korollar 4.14 *Wenn man eine Berechnung, die $O(n)$ sequentielle Zeit braucht, auf n Prozessoren in $O(\log n)$ Zeit ausführen kann, dann kann man diese Berechnung auf $O\left(\frac{n}{\log n}\right)$ Prozessoren in Zeit $O(\log n)$ ausführen (d. h. in* optimaler Zeit: $O(n) = O(\log n) \cdot O\left(\frac{n}{\log n}\right)$*).*

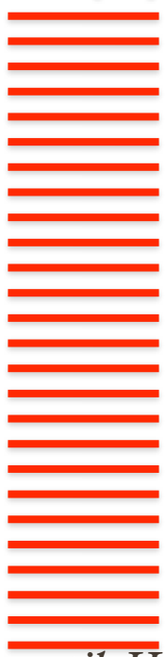
$$\lfloor \frac{8}{3} \rfloor = 2$$

Folgender Algorithmus (für CRCW$^{\text{arb}}$, CRCW$^{\text{com}}$, oder CRCW$^{\text{pri}}$–PRAM) kann mit n^2 Prozessoren das Maximum (Minimum) von n Zahlen in der Zeit $O(1)$ finden.

Folgender Algorithmus (für CRCW$^{\text{arb}}$, CRCW$^{\text{com}}$, oder CRCW$^{\text{pri}}$–PRAM) kann mit n^2 Prozessoren das Maximum (Minimum) von n Zahlen in der Zeit $O(1)$ finden.



Folgender Algorithmus (für $\mathrm{CRCW}^{\mathrm{arb}}$, $\mathrm{CRCW}^{\mathrm{com}}$, oder $\mathrm{CRCW}^{\mathrm{pri}}$–PRAM) kann mit n^2 Prozessoren das Maximum (Minimum) von n Zahlen in der Zeit $O(1)$ finden.



Prozessoren: $P_{ij},\ 1 \leq i, j \leq n$;
Input: $M(0) \leftarrow n$, $M(1) \leftarrow x_1, \ldots, M(n) \leftarrow x_n$. Die Register des globalen Speichers erhalten die Zahlen x_i, unter denen das Maximum gefunden werden soll;
Arbeitsspeicher: $Y(1), \ldots, Y(n)$ – Register des globalen Speichers;
Output: Maximum liegt in $M(0)$.

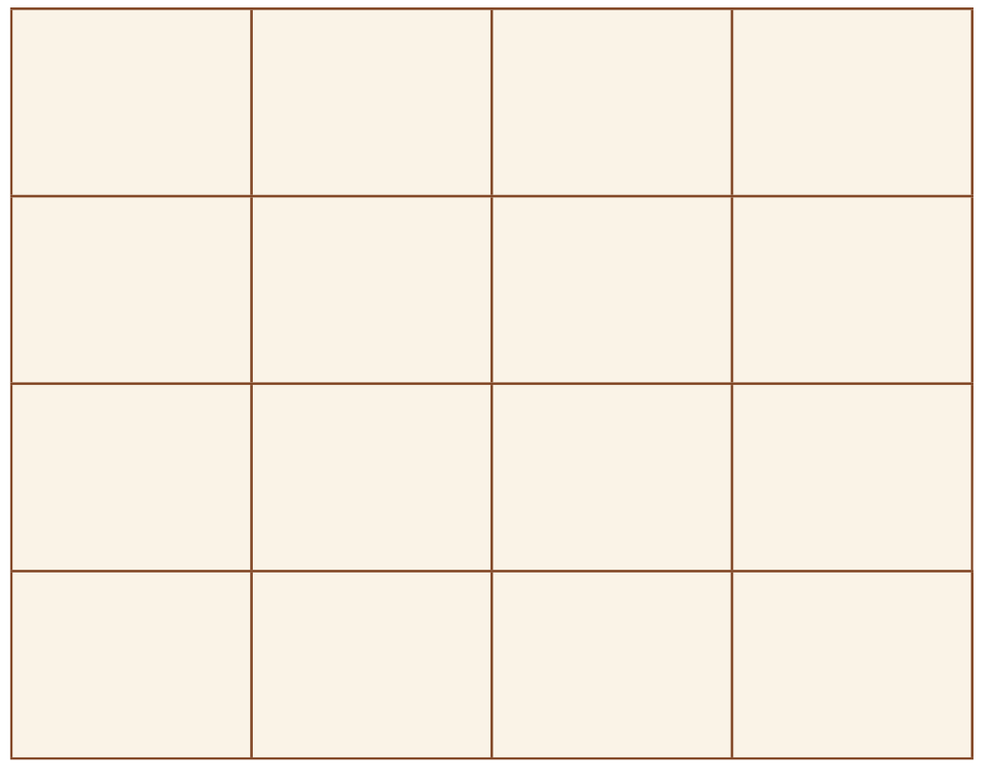
Algorithmus für den Prozessor P_{ij}:

1. $Y(i) \leftarrow 0$;
2. if $M(i) < M(j)$ then $Y(i) \leftarrow 1$;
3. if $Y(i) = 0$ then $M(0) \leftarrow M(i)$

Prozessoren: $P_{ij},\ 1 \leq i, j \leq n$;
Input: $M(0) \leftarrow n$, $M(1) \leftarrow x_1, \ldots, M(n) \leftarrow x_n$. Die Register des globalen Speichers erhalten die Zahlen x_i, unter denen das Maximum gefunden werden soll;
Arbeitsspeicher: $Y(1), \ldots, Y(n)$ – Register des globalen Speichers;
Output: Maximum liegt in $M(0)$.

Algorithmus für den Prozessor P_{ij}:

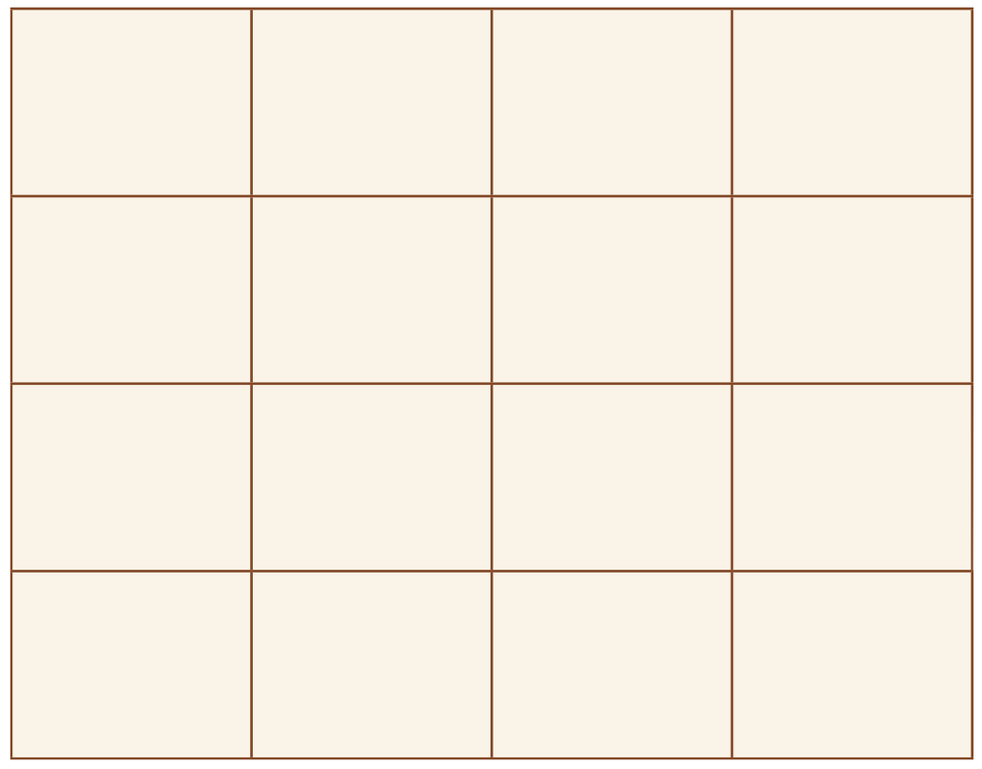
1. $Y(i) \leftarrow 0$;
2. if $M(i) < M(j)$ then $Y(i) \leftarrow 1$;
3. if $Y(i) = 0$ then $M(0) \leftarrow M(i)$



Prozessoren: $P_{ij},\ 1 \leq i, j \leq n$;
Input: $M(0) \leftarrow n$, $M(1) \leftarrow x_1, \ldots, M(n) \leftarrow x_n$. Die Register des globalen Speichers erhalten die Zahlen x_i, unter denen das Maximum gefunden werden soll;
Arbeitsspeicher: $Y(1), \ldots, Y(n)$ – Register des globalen Speichers;
Output: Maximum liegt in $M(0)$.

Algorithmus für den Prozessor P_{ij}:

1. $Y(i) \leftarrow 0$;
2. if $M(i) < M(j)$ then $Y(i) \leftarrow 1$;
3. if $Y(i) = 0$ then $M(0) \leftarrow M(i)$



Prozessoren: $P_{ij},\ 1 \leq i, j \leq n$;
Input: $M(0) \leftarrow n$, $M(1) \leftarrow x_1, \ldots, M(n) \leftarrow x_n$. Die Register des globalen Speichers erhalten die Zahlen x_i, unter denen das Maximum gefunden werden soll;
Arbeitsspeicher: $Y(1), \ldots, Y(n)$ – Register des globalen Speichers;
Output: Maximum liegt in $M(0)$.

Algorithmus für den Prozessor P_{ij}:

1. $Y(i) \leftarrow 0$;
2. if $M(i) < M(j)$ then $Y(i) \leftarrow 1$;
3. if $Y(i) = 0$ then $M(0) \leftarrow M(i)$

	M(0)=	M(1)=	M(2)=	M(3)=	M(4)=
M(1)=					
M(2)=					
M(3)=					
M(4)=					

Prozessoren: $P_{ij},\ 1 \leq i, j \leq n$;
Input: $M(0) \leftarrow n$, $M(1) \leftarrow x_1, \ldots, M(n) \leftarrow x_n$. Die Register des globalen Speichers erhalten die Zahlen x_i, unter denen das Maximum gefunden werden soll;
Arbeitsspeicher: $Y(1), \ldots, Y(n)$ – Register des globalen Speichers;
Output: Maximum liegt in $M(0)$.

Algorithmus für den Prozessor P_{ij}:

1. $Y(i) \leftarrow 0$;
2. if $M(i) < M(j)$ then $Y(i) \leftarrow 1$;
3. if $Y(i) = 0$ then $M(0) \leftarrow M(i)$

	M(0)=4	M(1)=2	M(2)=7	M(3)=3	M(4)=4
M(1)=2					
M(2)=7					
M(3)=3					
M(4)=4					

Prozessoren: $P_{ij},\ 1 \leq i, j \leq n$;
Input: $M(0) \leftarrow n$, $M(1) \leftarrow x_1, \ldots, M(n) \leftarrow x_n$. Die Register des globalen Speichers erhalten die Zahlen x_i, unter denen das Maximum gefunden werden soll;
Arbeitsspeicher: $Y(1), \ldots, Y(n)$ – Register des globalen Speichers;
Output: Maximum liegt in $M(0)$.

Algorithmus für den Prozessor P_{ij}:

1. $Y(i) \leftarrow 0$;
2. if $M(i) < M(j)$ then $Y(i) \leftarrow 1$;
3. if $Y(i) = 0$ then $M(0) \leftarrow M(i)$

M(0)=4	M(1)=2	M(2)=7	M(3)=3	M(4)=4
M(1)=2				
M(2)=7				
M(3)=3				
M(4)=4				

Prozessoren: $P_{ij},\ 1 \leq i, j \leq n$;
Input: $M(0) \leftarrow n$, $M(1) \leftarrow x_1, \ldots, M(n) \leftarrow x_n$. Die Register des globalen Speichers erhalten die Zahlen x_i, unter denen das Maximum gefunden werden soll;
Arbeitsspeicher: $Y(1), \ldots, Y(n)$ – Register des globalen Speichers;
Output: Maximum liegt in $M(0)$.

Algorithmus für den Prozessor P_{ij}:

1. $Y(i) \leftarrow 0$;
2. if $M(i) < M(j)$ then $Y(i) \leftarrow 1$;
3. if $Y(i) = 0$ then $M(0) \leftarrow M(i)$

	M(0)=4	M(1)=2	M(2)=7	M(3)=3	M(4)=4
Y(1)=	M(1)=2				
Y(2)=	M(2)=7				
Y(3)=	M(3)=3				
Y(4)=	M(4)=4				

Prozessoren: $P_{ij},\ 1 \leq i, j \leq n$;
Input: $M(0) \leftarrow n$, $M(1) \leftarrow x_1, \ldots, M(n) \leftarrow x_n$. Die Register des globalen Speichers erhalten die Zahlen x_i, unter denen das Maximum gefunden werden soll;
Arbeitsspeicher: $Y(1), \ldots, Y(n)$ – Register des globalen Speichers;
Output: Maximum liegt in $M(0)$.

Algorithmus für den Prozessor P_{ij}:

1. $Y(i) \leftarrow 0$;
2. if $M(i) < M(j)$ then $Y(i) \leftarrow 1$;
3. if $Y(i) = 0$ then $M(0) \leftarrow M(i)$

	M(0)=4	M(1)=2	M(2)=7	M(3)=3	M(4)=4
Y(1)=	M(1)=2				
Y(2)=	M(2)=7				
Y(3)=	M(3)=3				
Y(4)=	M(4)=4				

Prozessoren: $P_{ij},\ 1 \leq i, j \leq n$;
Input: $M(0) \leftarrow n$, $M(1) \leftarrow x_1, \ldots, M(n) \leftarrow x_n$. Die Register des globalen Speichers erhalten die Zahlen x_i, unter denen das Maximum gefunden werden soll;
Arbeitsspeicher: $Y(1), \ldots, Y(n)$ – Register des globalen Speichers;
Output: Maximum liegt in $M(0)$.

Algorithmus für den Prozessor P_{ij}:

1. $Y(i) \leftarrow 0$;
2. if $M(i) < M(j)$ then $Y(i) \leftarrow 1$;
3. if $Y(i) = 0$ then $M(0) \leftarrow M(i)$

	M(0)=4	M(1)=2	M(2)=7	M(3)=3	M(4)=4
Y(1)=0	M(1)=2				
Y(2)=0	M(2)=7				
Y(3)=0	M(3)=3				
Y(4)=0	M(4)=4				

Prozessoren: $P_{ij},\ 1 \leq i, j \leq n$;
Input: $M(0) \leftarrow n$, $M(1) \leftarrow x_1, \ldots, M(n) \leftarrow x_n$. Die Register des globalen Speichers erhalten die Zahlen x_i, unter denen das Maximum gefunden werden soll;
Arbeitsspeicher: $Y(1), \ldots, Y(n)$ – Register des globalen Speichers;
Output: Maximum liegt in $M(0)$.

Algorithmus für den Prozessor P_{ij}:

1. $Y(i) \leftarrow 0$;
2. if $M(i) < M(j)$ then $Y(i) \leftarrow 1$;
3. if $Y(i) = 0$ then $M(0) \leftarrow M(i)$

	M(0)=4	M(1)=2	M(2)=7	M(3)=3	M(4)=4
Y(1)=0	M(1)=2				
Y(2)=0	M(2)=7				
Y(3)=0	M(3)=3				
Y(4)=0	M(4)=4				

Prozessoren: $P_{ij},\ 1 \leq i, j \leq n$;
Input: $M(0) \leftarrow n$, $M(1) \leftarrow x_1, \ldots, M(n) \leftarrow x_n$. Die Register des globalen Speichers erhalten die Zahlen x_i, unter denen das Maximum gefunden werden soll;
Arbeitsspeicher: $Y(1), \ldots, Y(n)$ – Register des globalen Speichers;
Output: Maximum liegt in $M(0)$.

Algorithmus für den Prozessor P_{ij}:

1. $Y(i) \leftarrow 0$;
2. if $M(i) < M(j)$ then $Y(i) \leftarrow 1$;
3. if $Y(i) = 0$ then $M(0) \leftarrow M(i)$

	M(0)=4	M(1)=2	M(2)=7	M(3)=3	M(4)=4
Y(1)=0	M(1)=2	F	T	T	T
Y(2)=0	M(2)=7	F	F	F	F
Y(3)=0	M(3)=3	F	T	F	T
Y(4)=0	M(4)=4	F	T	F	F

Prozessoren: $P_{ij},\ 1 \leq i, j \leq n$;
Input: $M(0) \leftarrow n$, $M(1) \leftarrow x_1, \ldots, M(n) \leftarrow x_n$. Die Register des globalen Speichers erhalten die Zahlen x_i, unter denen das Maximum gefunden werden soll;
Arbeitsspeicher: $Y(1), \ldots, Y(n)$ – Register des globalen Speichers;
Output: Maximum liegt in $M(0)$.

Algorithmus für den Prozessor P_{ij}:

1. $Y(i) \leftarrow 0$;
2. if $M(i) < M(j)$ then $Y(i) \leftarrow 1$;
3. if $Y(i) = 0$ then $M(0) \leftarrow M(i)$

	M(0)=4	M(1)=2	M(2)=7	M(3)=3	M(4)=4
Y(1)=~~0~~1	M(1)=2	F	T	T	T
Y(2)=0	M(2)=7	F	F	F	F
Y(3)=~~0~~1	M(3)=3	F	T	F	T
Y(4)=~~0~~1	M(4)=4	F	T	F	F

Prozessoren: $P_{ij},\ 1 \leq i, j \leq n$;
Input: $M(0) \leftarrow n$, $M(1) \leftarrow x_1, \ldots, M(n) \leftarrow x_n$. Die Register des globalen Speichers erhalten die Zahlen x_i, unter denen das Maximum gefunden werden soll;
Arbeitsspeicher: $Y(1), \ldots, Y(n)$ – Register des globalen Speichers;
Output: Maximum liegt in $M(0)$.

Algorithmus für den Prozessor P_{ij}:

1. $Y(i) \leftarrow 0$;
2. if $M(i) < M(j)$ then $Y(i) \leftarrow 1$;
3. if $Y(i) = 0$ then $M(0) \leftarrow M(i)$

	M(0)=4	M(1)=2	M(2)=7	M(3)=3	M(4)=4
Y(1)=~~0~~1	M(1)=2	F	T	T	T
Y(2)=0	M(2)=7	F	F	F	F
Y(3)=~~0~~1	M(3)=3	F	T	F	T
Y(4)=~~0~~1	M(4)=4	F	T	F	F

Prozessoren: $P_{ij},\ 1 \leq i, j \leq n$;
Input: $M(0) \leftarrow n$, $M(1) \leftarrow x_1, \ldots, M(n) \leftarrow x_n$. Die Register des globalen Speichers erhalten die Zahlen x_i, unter denen das Maximum gefunden werden soll;
Arbeitsspeicher: $Y(1), \ldots, Y(n)$ – Register des globalen Speichers;
Output: Maximum liegt in $M(0)$.

Algorithmus für den Prozessor P_{ij}:

1. $Y(i) \leftarrow 0$;
2. if $M(i) < M(j)$ then $Y(i) \leftarrow 1$;
3. if $Y(i) = 0$ then $M(0) \leftarrow M(i)$

	M(0)=~~4~~ 7	M(1)=2	M(2)=7	M(3)=3	M(4)=4
Y(1)=~~0~~ 1	M(1)=2	F	T	T	T
Y(2)=0	M(2)=7	F	F	F	F
Y(3)=~~0~~ 1	M(3)=3	F	T	F	T
Y(4)=~~0~~ 1	M(4)=4	F	T	F	F

Prozessoren: $P_{ij},\ 1 \leq i, j \leq n$;
Input: $M(0) \leftarrow n$, $M(1) \leftarrow x_1, \ldots, M(n) \leftarrow x_n$. Die Register des globalen Speichers erhalten die Zahlen x_i, unter denen das Maximum gefunden werden soll;
Arbeitsspeicher: $Y(1), \ldots, Y(n)$ – Register des globalen Speichers;
Output: Maximum liegt in $M(0)$.

Algorithmus für den Prozessor P_{ij}:

1. $Y(i) \leftarrow 0$;
2. if $M(i) < M(j)$ then $Y(i) \leftarrow 1$;
3. if $Y(i) = 0$ then $M(0) \leftarrow M(i)$

$$time_M(n) =$$

	M(0)=~~4~~ 7	M(1)=2	M(2)=7	M(3)=3	M(4)=4
Y(1)=~~0~~ 1	M(1)=2	F	T	T	T
Y(2)=0	M(2)=7	F	F	F	F
Y(3)=~~0~~ 1	M(3)=3	F	T	F	T
Y(4)=~~0~~ 1	M(4)=4	F	T	F	F

Prozessoren: $P_{ij},\ 1 \leq i, j \leq n$;
Input: $M(0) \leftarrow n$, $M(1) \leftarrow x_1, \ldots, M(n) \leftarrow x_n$. Die Register des globalen Speichers erhalten die Zahlen x_i, unter denen das Maximum gefunden werden soll;
Arbeitsspeicher: $Y(1), \ldots, Y(n)$ – Register des globalen Speichers;
Output: Maximum liegt in $M(0)$.

Algorithmus für den Prozessor P_{ij}:

1. $Y(i) \leftarrow 0$;
2. if $M(i) < M(j)$ then $Y(i) \leftarrow 1$;
3. if $Y(i) = 0$ then $M(0) \leftarrow M(i)$

$$time_M(n) = O(1)$$

	M(0)=~~4~~ 7	M(1)=2	M(2)=7	M(3)=3	M(4)=4
Y(1)=~~0~~ 1	M(1)=2	F	T	T	T
Y(2)=0	M(2)=7	F	F	F	F
Y(3)=~~0~~ 1	M(3)=3	F	T	F	T
Y(4)=~~0~~ 1	M(4)=4	F	T	F	F

Prozessoren: $P_{ij},\ 1 \leq i, j \leq n$;
Input: $M(0) \leftarrow n$, $M(1) \leftarrow x_1, \ldots, M(n) \leftarrow x_n$. Die Register des globalen Speichers erhalten die Zahlen x_i, unter denen das Maximum gefunden werden soll;
Arbeitsspeicher: $Y(1), \ldots, Y(n)$ – Register des globalen Speichers;
Output: Maximum liegt in $M(0)$.

Algorithmus für den Prozessor P_{ij}:

1. $Y(i) \leftarrow 0$;
2. if $M(i) < M(j)$ then $Y(i) \leftarrow 1$;
3. if $Y(i) = 0$ then $M(0) \leftarrow M(i)$

$$time_M(n) = O(1)$$
$$proc_M(n) = O(n^2)$$

	M(0)=~~4~~ 7	M(1)=2	M(2)=7	M(3)=3	M(4)=4
Y(1)=~~0~~1	M(1)=2	F	T	T	T
Y(2)=0	M(2)=7	F	F	F	F
Y(3)=~~0~~1	M(3)=3	F	T	F	T
Y(4)=~~0~~1	M(4)=4	F	T	F	F

Prozessoren: $P_{ij},\ 1 \leq i, j \leq n$;
Input: $M(0) \leftarrow n$, $M(1) \leftarrow x_1, \ldots, M(n) \leftarrow x_n$. Die Register des globalen Speichers erhalten die Zahlen x_i, unter denen das Maximum gefunden werden soll;
Arbeitsspeicher: $Y(1), \ldots, Y(n)$ – Register des globalen Speichers;
Output: Maximum liegt in $M(0)$.

Algorithmus für den Prozessor P_{ij}:

1. $Y(i) \leftarrow 0$;
2. if $M(i) < M(j)$ then $Y(i) \leftarrow 1$;
3. if $Y(i) = 0$ then $M(0) \leftarrow M(i)$

$$time_M(n) = O(1)$$
$$proc_M(n) = O(n^2)$$
$$work_M(n) =$$

	M(0)=~~4~~ 7	M(1)=2	M(2)=7	M(3)=3	M(4)=4
Y(1)=~~0~~1	M(1)=2	F	T	T	T
Y(2)=0	M(2)=7	F	F	F	F
Y(3)=~~0~~1	M(3)=3	F	T	F	T
Y(4)=~~0~~1	M(4)=4	F	T	F	F

Prozessoren: $P_{ij},\ 1 \leq i, j \leq n$;
Input: $M(0) \leftarrow n$, $M(1) \leftarrow x_1, \ldots, M(n) \leftarrow x_n$. Die Register des globalen Speichers erhalten die Zahlen x_i, unter denen das Maximum gefunden werden soll;
Arbeitsspeicher: $Y(1), \ldots, Y(n)$ – Register des globalen Speichers;
Output: Maximum liegt in $M(0)$.

Algorithmus für den Prozessor P_{ij}:

1. $Y(i) \leftarrow 0$;
2. if $M(i) < M(j)$ then $Y(i) \leftarrow 1$;
3. if $Y(i) = 0$ then $M(0) \leftarrow M(i)$

	M(0)=~~4~~ 7	M(1)=2	M(2)=7	M(3)=3	M(4)=4
Y(1)=~~0~~1	M(1)=2	F	T	T	T
Y(2)=0	M(2)=7	F	F	F	F
Y(3)=~~0~~1	M(3)=3	F	T	F	T
Y(4)=~~0~~1	M(4)=4	F	T	F	F

$$time_M(n) = O(1)$$
$$proc_M(n) = O(n^2)$$
$$work_M(n) = O(n^2)$$

Prozessoren: $P_{ij},\ 1 \leq i, j \leq n$;
Input: $M(0) \leftarrow n$, $M(1) \leftarrow x_1, \ldots, M(n) \leftarrow x_n$. Die Register des globalen Speichers erhalten die Zahlen x_i, unter denen das Maximum gefunden werden soll;
Arbeitsspeicher: $Y(1), \ldots, Y(n)$ – Register des globalen Speichers;
Output: Maximum liegt in $M(0)$.

Algorithmus für den Prozessor P_{ij}:

1. $Y(i) \leftarrow 0$;
2. if $M(i) < M(j)$ then $Y(i) \leftarrow 1$;
3. if $Y(i) = 0$ then $M(0) \leftarrow M(i)$

	M(0)=~~4~~ 7	M(1)=2	M(2)=7	M(3)=3	M(4)=4
Y(1)=~~0~~ 1	M(1)=2	F	T	T	T
Y(2)=0	M(2)=7	F	F	F	F
Y(3)=~~0~~ 1	M(3)=3	F	T	F	T
Y(4)=~~0~~ 1	M(4)=4	F	T	F	F

$$time_M(n) = O(1)$$
$$proc_M(n) = O(n^2)$$
$$work_M(n) = O(n^2)$$

CRCW^{com} PRAM

CRCW^{arb} PRAM

CRCW^{pri} PRAM

4.2.5 PRAM–Hauptkomplexitätsklassen

Satz 4.19 *Seien $T, P : \mathbb{N} \to \mathbb{N}$ Funktionen und $k : \mathbb{N} \to \mathbb{N}$ eine rekursive Funktion, die auf RAM_+ in der Zeit T berechenbar ist. Für eine feste Konstante c gilt:*

$$\mathcal{M}TimeProc(T, P) \subsetneq \mathcal{M}TimeProc(c \cdot k \cdot T, \lceil \frac{P}{k} \rceil)$$

4.2.5 PRAM–Hauptkomplexitätsklassen

Satz 4.19 *Seien $T, P : \mathbb{N} \to \mathbb{N}$ Funktionen und $k : \mathbb{N} \to \mathbb{N}$ eine rekursive Funktion, die auf RAM_+ in der Zeit T berechenbar ist. Für eine feste Konstante c gilt:*

$$\mathcal{MTimeProc}(T, P) \subsetneq \mathcal{MTimeProc}(c \cdot k \cdot T, \lceil \frac{P}{k} \rceil)$$

4.2.5 PRAM–Hauptkomplexitätsklassen

Satz 4.19 *Seien $T, P : \mathbb{N} \to \mathbb{N}$ Funktionen und $k : \mathbb{N} \to \mathbb{N}$ eine rekursive Funktion, die auf RAM_+ in der Zeit T berechenbar ist. Für eine feste Konstante c gilt:*

$$\mathcal{MTimeProc}(T, P) \subsetneq \mathcal{MTimeProc}(c \cdot k \cdot T, \lceil \frac{P}{k} \rceil)$$

eine PRAM-Maschinenklasse

Zeitkomplexität

4.2.5 PRAM–Hauptkomplexitätsklassen

Satz 4.19 *Seien $T, P : \mathbb{N} \to \mathbb{N}$ Funktionen und $k : \mathbb{N} \to \mathbb{N}$ eine rekursive Funktion, die auf RAM_+ in der Zeit T berechenbar ist. Für eine feste Konstante c gilt:*

$$\mathcal{MTimeProc}(T, P) \subsetneq \mathcal{MTimeProc}(c \cdot k \cdot T, \lceil \frac{P}{k} \rceil)$$

Prozessorkomplexität

Zeitkomplexität

eine PRAM-Maschinenklasse

4.2.5 PRAM–Hauptkomplexitätsklassen

Satz 4.19 *Seien $T, P : \mathbb{N} \to \mathbb{N}$ Funktionen und $k : \mathbb{N} \to \mathbb{N}$ eine rekursive Funktion, die auf RAM_+ in der Zeit T berechenbar ist. Für eine feste Konstante c gilt:*

$$\mathcal{M}TimeProc(T, P) \subsetneq \mathcal{M}TimeProc(c \cdot k \cdot T, \lceil \frac{P}{k} \rceil)$$

Prozessorkomplexität

Zeitkomplexität

eine PRAM-Maschinenklasse

Beweis wie Ablaufmethode von Brendt

(Brent's scheduling principle)

Satz 4.19 *Seien $T, P : \mathbb{N} \to \mathbb{N}$ Funktionen und $k : \mathbb{N} \to \mathbb{N}$ eine rekursive Funktion, die auf RAM_+ in der Zeit T berechenbar ist. Für eine feste Konstante c gilt:*

$$\mathcal{MTimeProc}(T, P) \subsetneq \mathcal{MTimeProc}(c \cdot k \cdot T, \lceil \frac{P}{k} \rceil)$$

Satz 4.19 *Seien $T, P : \mathbb{N} \to \mathbb{N}$ Funktionen und $k : \mathbb{N} \to \mathbb{N}$ eine rekursive Funktion, die auf RAM_+ in der Zeit T berechenbar ist. Für eine feste Konstante c gilt:*

$$\mathcal{M}TimeProc(T, P) \subsetneq \mathcal{M}TimeProc(c \cdot k \cdot T, \lceil \frac{P}{k} \rceil)$$

Für Polynome P, T und $k = P$ ergibt sich aus dem vorigen Satz:

$$\begin{aligned} CRCW_+\ TimeProc(T, P) &\subseteq CRCW_+\ TimeProc(O(P \cdot T), 1) \\ &= RAM_+ - Time(O(T \cdot P)) \subseteq \mathcal{P} \end{aligned}$$

Satz 4.19 *Seien $T, P : \mathbb{N} \to \mathbb{N}$ Funktionen und $k : \mathbb{N} \to \mathbb{N}$ eine rekursive Funktion, die auf RAM_+ in der Zeit T berechenbar ist. Für eine feste Konstante c gilt:*

$$\mathcal{M}TimeProc(T, P) \subsetneq \mathcal{M}TimeProc(c \cdot k \cdot T, \lceil \frac{P}{k} \rceil)$$

Für Polynome P, T und $k = P$ ergibt sich aus dem vorigen Satz:

$$\begin{aligned} CRCW_+\ TimeProc(T, P) &\subseteq CRCW_+\ TimeProc(O(P \cdot T), 1) \\ &= RAM_+ - Time(O(T \cdot P)) \subseteq \mathcal{P} \end{aligned}$$

die 1. Maschinenklasse

$$\text{RAM}_{+}\text{–Time}(POL) = \text{RAM–Time}_{log}(POL) = \mathcal{P}$$
$$= \text{DTM–Time}(POL) = \text{DTM}_1\text{–Time}(POL)$$

Korollar 4.20

$$CRCW_{+}\ TimeProc(\text{POL},\ \text{POL}) = \mathcal{P}$$

die 1. Maschinenklasse

$$\text{RAM}_{+}\text{–Time}(POL) = \text{RAM–Time}_{log}(POL) = \mathcal{P}$$
$$= \text{DTM–Time}(POL) = \text{DTM}_1\text{–Time}(POL)$$

die 2. Maschinenklasse ⇒ *Parallelrechner*

Korollar 4.20

$$CRCW_{+}\ TimeProc(\text{POL},\ \text{POL}) = \mathcal{P}$$

die 1. Maschinenklasse

$$\boxed{\text{RAM}_{+}\text{–Time}(POL)} = \text{RAM–Time}_{log}(POL) = \mathcal{P}$$
$$= \text{DTM–Time}(POL) = \boxed{\text{DTM}_1\text{–Time}(POL)}$$

die 2. Maschinenklasse ⇒ *Parallelrechner*

Korollar 4.20

$CRCW_{+}\ TimeProc(\text{POL}, \text{POL}) = \mathcal{P}$?

die 1. Maschinenklasse

$$\boxed{\mathrm{RAM}_{+}\text{–Time}(POL)} = \mathrm{RAM}\text{–Time}_{log}(POL) = \mathcal{P}$$
$$= \mathrm{DTM}\text{–Time}(POL) = \boxed{\mathrm{DTM}_1\text{–Time}(POL)}$$

die 2. Maschinenklasse ⇒ *Parallelrechner*

Korollar 4.20

$CRCW_{+}\ TimeProc(\mathrm{POL}, \mathrm{POL}) = \mathcal{P}$?

Neue Komplexitätsklasse:

$$PRAM - Time(t(n)) \quad := \quad CRCW_{+}\ TimeProc(t(n), 2^{t(n)})$$

die 1. Maschinenklasse

$$\boxed{\text{RAM}_{+}\text{–Time}(POL)} = \text{RAM–Time}_{log}(POL) = \mathcal{P}$$
$$= \text{DTM–Time}(POL) = \boxed{\text{DTM}_1\text{–Time}(POL)}$$

die 2. Maschinenklasse ⇒ *Parallelrechner*

Korollar 4.20

$CRCW_{+}\ TimeProc(\text{POL}, \text{POL}) = \mathcal{P}$?

Neue Komplexitätsklasse:

$$PRAM - Time(t(n)) \quad := \quad CRCW_{+}\ TimeProc(t(n), 2^{t(n)})$$

$$PRAM - Time(POL) \quad = \quad ?$$

Satz 4.21 $PRAM - Time(POL) = \mathcal{PSPACE}$

Satz 4.21 $PRAM - Time(POL) = \mathcal{PSPACE}$

Benötigte Hilfslemmata (hier ohne Beweis):

Satz 4.21 $PRAM - Time(POL) = \mathcal{PSPACE}$

Benötigte Hilfslemmata (hier ohne Beweis):

$$Space(s(n)) \subseteq PRAM - Time(O(s(n)))$$

Satz 4.21 $PRAM - Time(POL) = \mathcal{PSPACE}$

Benötigte Hilfslemmata (hier ohne Beweis):

$$Space(s(n)) \subseteq PRAM - Time(O(s(n)))$$

$$PRAM - Time(t(n)) \subseteq Space(t^2(n))$$

Satz 4.21 $PRAM - Time(POL) = \mathcal{PSPACE}$

Benötigte Hilfslemmata (hier ohne Beweis):

$$Space(s(n)) \subseteq PRAM - Time(O(s(n)))$$

$$PRAM - Time(t(n)) \subseteq Space(t^2(n))$$

Damit folgt dann:

Satz 4.21 $PRAM - Time(POL) \quad = \quad \mathcal{PSPACE}$

Benötigte Hilfslemmata (hier ohne Beweis):

$$Space(s(n)) \subseteq PRAM - Time(O(s(n)))$$

$$PRAM - Time(t(n)) \subseteq Space(t^2(n))$$

Damit folgt dann:

$$\mathcal{PSPACE} = \bigcup_k PSpace(n^k) \subseteq \bigcup_k PRAM\text{-}Time(O(n^k)) = PRAM - Time(POL)$$

Satz 4.21 $PRAM - Time(POL) \quad = \quad \mathcal{PSPACE}$

Benötigte Hilfslemmata (hier ohne Beweis):

$$Space(s(n)) \subseteq PRAM - Time(O(s(n)))$$

$$PRAM - Time(t(n)) \subseteq Space(t^2(n))$$

Damit folgt dann:

$$\mathcal{PSPACE} = \bigcup_k PSpace(n^k) \subseteq \bigcup_k PRAM\text{-}Time(O(n^k)) = PRAM - Time(POL)$$

$$PRAM - Time(POL) = \bigcup_k PRAM\text{-}Time(n^k) \subseteq \bigcup_k PSpace(n^{2k}) = \mathcal{PSPACE}$$

Satz 4.21 $PRAM - Time(POL) \quad = \quad \mathcal{PSPACE}$

Benötigte Hilfslemmata (hier ohne Beweis):

$$Space(s(n)) \subseteq PRAM - Time(O(s(n)))$$

$$PRAM - Time(t(n)) \subseteq Space(t^2(n))$$

Damit folgt dann:

$$\mathcal{PSPACE} = \bigcup_k PSpace(n^k) \subseteq \bigcup_k PRAM\text{-}Time(O(n^k)) = PRAM - Time(POL)$$

$$PRAM - Time(POL) = \bigcup_k PRAM\text{-}Time(n^k) \subseteq \bigcup_k PSpace(n^{2k}) = \mathcal{PSPACE}$$

die 2. Maschinenklasse ⇒ *Parallelrechner*

Satz 4.21 $PRAM - Time(POL) \quad = \quad \mathcal{PSPACE}$

Benötigte Hilfslemmata (hier ohne Beweis):

$$Space(s(n)) \subseteq PRAM - Time(O(s(n)))$$

$$PRAM - Time(t(n)) \subseteq Space(t^2(n))$$

Damit folgt dann:

$$\mathcal{PSPACE} = \bigcup_k PSpace(n^k) \subseteq \bigcup_k PRAM\text{-}Time(O(n^k)) = PRAM - Time(POL)$$

$$PRAM - Time(POL) = \bigcup_k PRAM\text{-}Time(n^k) \subseteq \bigcup_k PSpace(n^{2k}) = \mathcal{PSPACE}$$

die 2. Maschinenklasse ⇒ *Parallelrechner*

Parallel Computation Thesis *There exists a standard class of (parallel) machine models, which includes among others all variants of PRAM machines, and also APM, for which polynomial time is as powerful as polynomial space for sequencial machines (from the first machine class).*

Satz 4.21 $PRAM - Time(POL) \quad = \quad \mathcal{PSPACE}$

Benötigte Hilfslemmata (hier ohne Beweis):

$$Space(s(n)) \subseteq PRAM - Time(O(s(n)))$$

$$PRAM - Time(t(n)) \subseteq Space(t^2(n))$$

Damit folgt dann:

$$\mathcal{PSPACE} = \bigcup_k PSpace(n^k) \subseteq \bigcup_k PRAM\text{-}Time(O(n^k)) = PRAM - Time(POL)$$

$$PRAM - Time(POL) = \bigcup_k PRAM\text{-}Time(n^k) \subseteq \bigcup_k PSpace(n^{2k}) = \mathcal{PSPACE}$$

die 2. Maschinenklasse ⇒ *Parallelrechner*

Parallel Computation Thesis *There exists a standard class of (parallel) machine models, which includes among others all variants of PRAM machines, and also APM, for which polynomial time is as powerful as polynomial space for sequencial machines (from the first machine class).*

Satz 4.21 $PRAM - Time(POL) \quad = \quad \mathcal{PSPACE}$

Benötigte Hilfslemmata (hier ohne Beweis):

$$Space(s(n)) \subseteq PRAM - Time(O(s(n)))$$

$$PRAM - Time(t(n)) \subseteq Space(t^2(n))$$

Damit folgt dann:

$$\mathcal{PSPACE} = \bigcup_k PSpace(n^k) \subseteq \bigcup_k PRAM\text{-}Time(O(n^k)) = PRAM - Time(POL)$$

$$PRAM - Time(POL) = \bigcup_k PRAM\text{-}Time(n^k) \subseteq \bigcup_k PSpace(n^{2k}) = \mathcal{PSPACE}$$

die 2. Maschinenklasse ⇒ *Parallelrechner*

Parallel Computation Thesis *There exists a standard class of (parallel) machine models, which includes among others all variants of PRAM machines, and also APM, for which polynomial time is as powerful as polynomial space for sequencial machines (from the first machine class).*

4.2.6 Grenzen der Parallelität

Natürliche Frage: Gibt es ein natürliches und einfaches Problem, für das es keinen parallelen Algorithmus gibt, der schneller ist als der schnellste sequentielle Algorithmus?

Satz 4.22 (Kung) *Kein paralleler Algorithmus, der die Operationen* $+, -, *, \div$ *benutzt, kann ein Polynom n-ten Grades schneller als in* $\log n$ *Schritten berechnen.*

Korollar 4.23 *Kein paralleler Algorithmus kann* x^n *schneller als ein sequentieller Algorithmus berechnen.*