6.5 Streuen und Sammeln

Sortiere Werte, die in einem festen Intervall [UNT, OB] liegen: $a_1, a_2, ..., a_n$ mit $UNT \leq a_i \leq OB$. O.B.d.A seien die a_i natürliche Zahlen und UNT = 0 und OB = m-1. Bucketsort: Verteile ("streue") die n Elemente A(1..n) auf m nacheinander angeordnete Fächer bucket(0..m-1) ("Eimer", daher "bucketsort"), hole sie anschließend in der Reihenfolge der Fächer wieder heraus und lege sie im Feld A ab.
Aufwand: O(n+m). Programm zum Sortieren (Listenbearbeitung selbst hinzufügen):

```
... A: array (1..n) of integer;  bucket: array (0..m-1) of "liste von integer"; ...
begin
for j in 0..m-1 loop bucket(j) := "leer"; end loop;
for i in 1..n loop                                   -- streuen
        bucket(A(i)) := "hänge A(i) an bucket(A(i)) an"; end loop;
i := 0;
for j in 0..m-1 loop                                 -- sammeln
        while "bucket(j) nicht leer" loop
                i := i+1; A(i) := "erstes Element von bucket(j)";
                bucket(j) := "bucket(j) ohne das erset Element"; end loop;
end loop;
end
```


6.6 Paralleles Sortieren

Das Sortieren von n Elementen kostet mindestens n·log(n) Vergleiche, wenn nur ein Prozessor vorhanden ist. Ein schneller Rechner kann heute einige Millionen Vergleiche pro Sekunde durchführen, sofern man 32-stellige Zahlen als Schlüssel verwendet. Für die Umspeicherungen usw. setzen wir den Faktor 10 an. Will man maximal eine Stunde auf das Ergebnis warten, so lassen sich also 10^9 Vergleiche ausführen, womit man ungefähr 40.000.000 Elemente sortieren kann (vorausgesetzt, sie passen alle in den Arbeitsspeicher).

Solche Größenordnungen sind selten, so dass im Prinzip das Sortieren heute kein Problem mehr darstellen sollte. Allerdings entstehen Probleme, wenn eine Sortierung sehr schnell erfolgen muss, weil sicherheitskritische Systeme hiervon abhängen, oder aus anderen Gründen.

Will man also schneller sortieren, so kann man entweder auf noch schnellere Rechner warten oder man kann eine Beschleunigung des Sortierens durch paralleles Vorgehen erhoffen. Wir stellen hierzu zwei „einfache“ Verfahren vor. (Weitere Verfahren finden Sie in dem Buch von I. Wegener, "Effiziente Algorithmen für grundlegende Funktionen", Teubner-Verlag.)

Verfahren 1:

Lineare Kette mit n Prozessoren.

Verfahren 2:

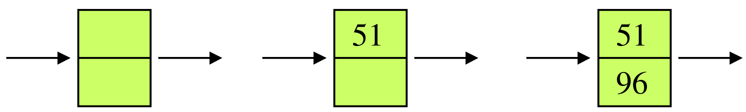
Divide and Conquer Verfahren mit $(1/4) \cdot n \cdot \log^2(n)$ Prozessoren.

Verfahren 1: Lineare Kette

Hierzu betrachten wir einen Prozessor, der

zwei Speicherplätze für Zahlen,
einen Eingang (links) und
einen Ausgang (rechts)

besitzt:



Grundbaustein, leer — Grundbaustein mit einer Zahl — Grundbaustein mit zwei Zahlen

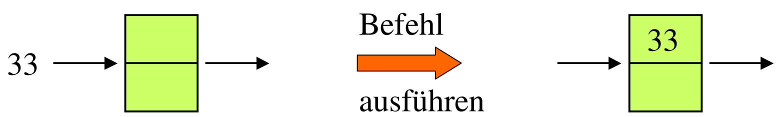
(Die Zahl darf auch unten stehen.)

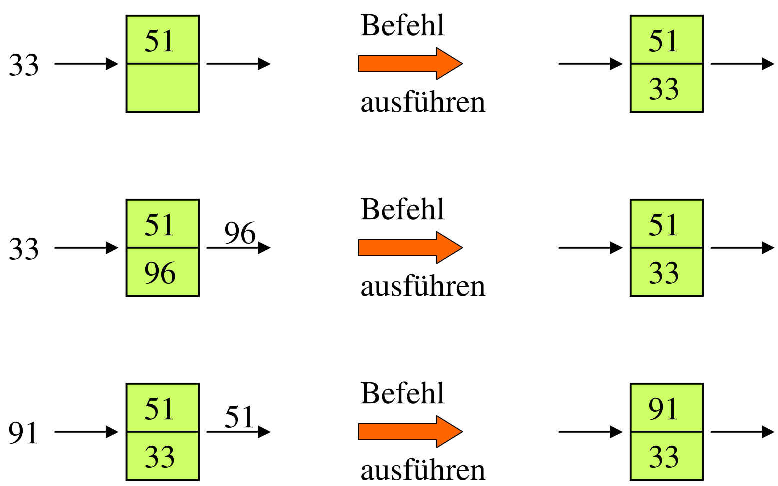
Der Grundbaustein kann nur folgenden Befehl ausführen:

1. Falls er zwei Zahlen besitzt, gibt er die größere der beiden nach rechts aus.
2. Falls eine Zahl von links kommt, legt er sie in einen seiner freien Speicherplätze.

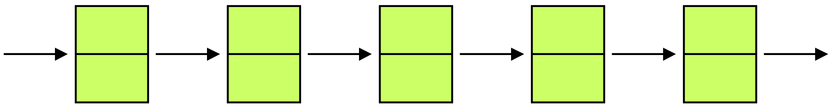
Die Befehlsteile 1. und 2. werden gleichzeitig ausgeführt!

Wenn die Bedingung in 1. oder in 2. zutrifft, dann muss der Befehl auch ausgeführt werden.

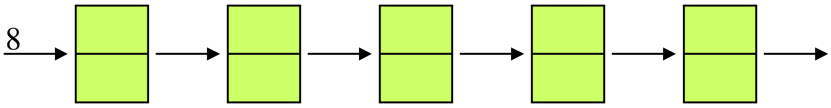


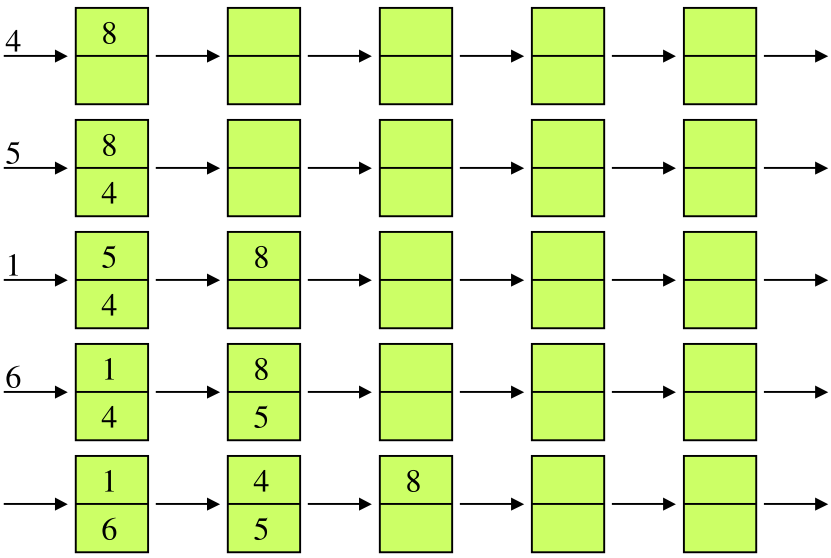


Nun koppeln wir n=5 solche Grundbausteine zu einer Kette aneinander:

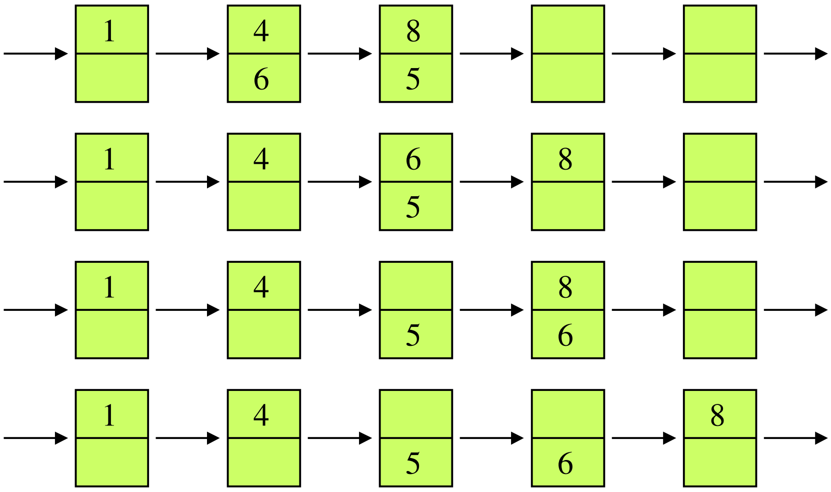


Wir verlangen, dass alle n Bausteine synchron arbeiten. Nun geben wir in n=5 Takten die Zahlen 8, 4, 5, 1, 6 von links ein:





Die Eingabe ist beendet. Die Kette arbeitet noch weiter, bis in jedem Baustein genau eine Zahl steht.



Nach 2n-1 Takten endet das Verfahren und jeder Baustein gibt im 2n-ten Takt seine Zahl aus. Diese Folge ist sortiert.

Wie viele Schritte benötigt dieses Verfahren?

Offensichtlich sind n Elemente nach **2n-1** Schritten sortiert.

Hierfür benötigt man n Prozessoren.
Für große Werte von n ist dies eine deutliche Verbesserung gegenüber den bisherigen $O(n \cdot \log(n))$-Verfahren.

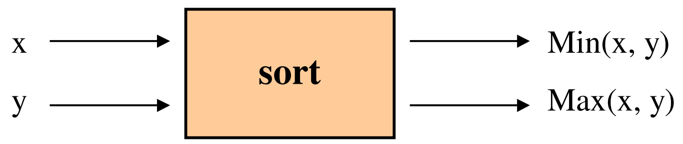
Gibt es bessere Verfahren? Möglichst solche, die <u>viel</u> schneller als in $O(n)$ Schritten arbeiten - dafür dürfen sie dann auch mehr als $O(n)$ Prozessoren besitzen ...

Verfahren 2: Divide and Conquer Ansatz

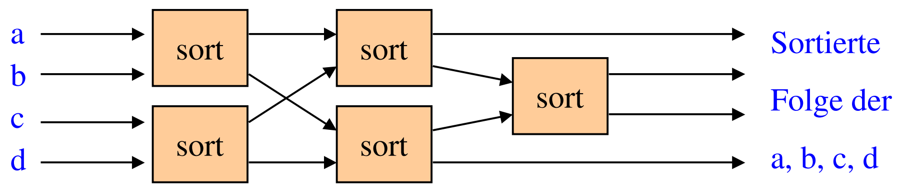
Der Divide-and-Conquer-Ansatz lautet:
Zerlege das Problem in z.B. zwei Teilprobleme, löse diese und setze aus den Teillösungen die Gesamtlösung zusammen.

Wir gehen nun von einer allgemeinen Struktur aus, bei der die zu sortierenden Elemente nicht nacheinander, sondern parallel zueinander eingegeben werden. Dies bedeutet, dass wir mit mindestens O(n) Grundbausteinen rechnen müssen, um n Elemente zu sortieren.

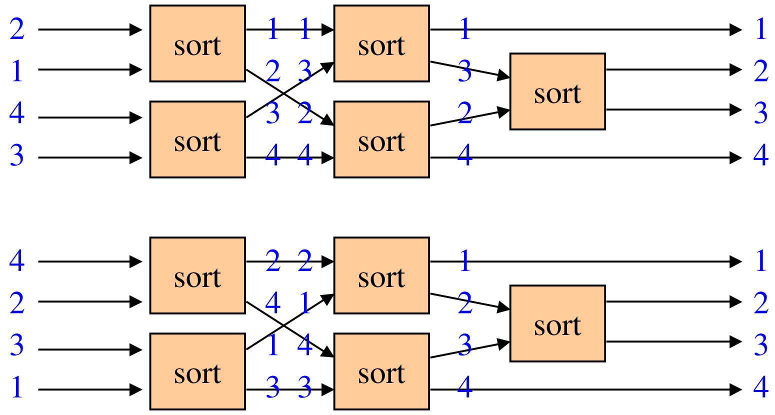
Als Grundbaustein verwenden wir einen elementaren Sortierbaustein **sort** für zwei Werte:



Hiermit kann man beispielsweise n = 4 Werte wie folgt sortieren:

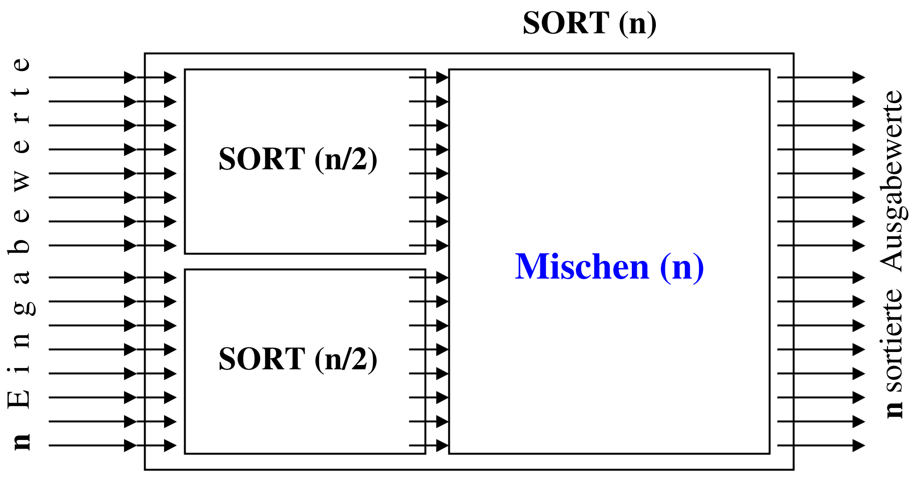


Beispiele:



4 Elemente werden in 3 Schritten mit 5 Bausteinen sortiert.

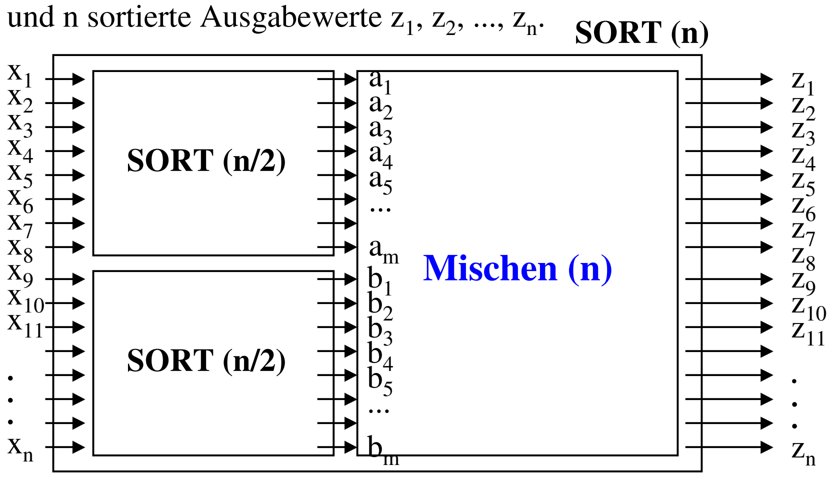
Divide and Conquer Ansatz für n Eingabewerte:



Genauer: Gegeben sind n Eingabewerte $x_1, x_2, ..., x_n$.
Wir erhalten 2 sortierte Folgen mit je m Elementen:

$a_1, a_2, ..., a_m$ und $b_1, b_2, ..., b_m$ $(m = n/2)$

und n sortierte Ausgabewerte $z_1, z_2, ..., z_n$.



Die Rekursion sorgt dafür, dass die Folgen $a_1, a_2, ..., a_m$ und $b_1, b_2, ..., b_m$ sortiert sind $(m=n/2)$.
Wenn es einen "guten" Baustein **Mischen (n)** gibt, dann lässt sich hieraus auch ein "gutes" Sortierwerk für $n = 2^s$ Eingabewerte, für jede natürliche Zahl s, konstruieren.

In der Tat gibt es eine Technik, um zwei sortierte Folgen *parallel* in relativ wenigen Schritten zu einer gemeinsamen sortierten Folge zu mischen.

Diese Technik nennt sich "odd-even-merge" (dtsch.: gerade-ungerad-Mischen) und wird ebenfalls rekursiv definiert.

Definition: odd-even-merge "Mischen(2m)"

Gegeben: zwei sortierte Folgen (für $m = 2^k$ für eine natürliche Zahl $k \geq 0$) $a_1, a_2, ..., a_m$ und $b_1, b_2, ..., b_m$.

Ergebnis: die zugehörige sortierte Folge $z_1, z_2, ..., z_{2m}$.

Vorgehen zur Durchführung von "Mischen (2m)":

$m = 1$: Vergleiche mit dem Baustein "**sort**"; fertig.

$m > 1$:

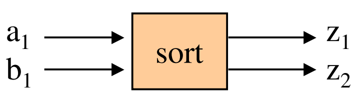
(1) Mische rekursiv die ungeraden Glieder der a- und der b-Folge sowie die geraden Glieder der a- und der b-Folge, d.h., mische die sortierten Folgen $a_1, a_3, a_5, ..., a_{m-1}$ und $b_1, b_3, b_5, ..., b_{m-1}$ zur sortierten Folge $u_1, u_2, u_3,..., u_m$ und mische die sortierten Folgen $a_2, a_4, a_6, ..., a_m$ und $b_2, b_4, b_6, ..., b_m$ zur sortierten Folge $v_1, v_2, v_3,..., v_m$.

(2) $z_1 = u_1$, $z_{2m} = v_m$ und führe parallel m-1 Vergleiche durch: $z_{2i} = \text{Min}(u_{i+1}, v_i)$, $z_{2i+1} = \text{Max}(u_{i+1}, v_i)$ für i=1,2,..,m-1.

Wir konstruieren zunächst das Mischen für einige Werte von m.

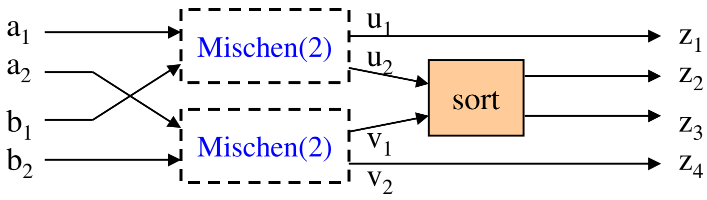
Der Fall m=1 benötigt genau einen Sortierbaustein:

m = 1:

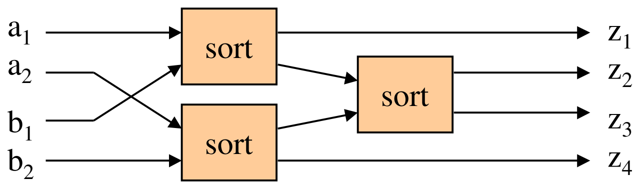


Dies ist der Baustein Mischen(2).

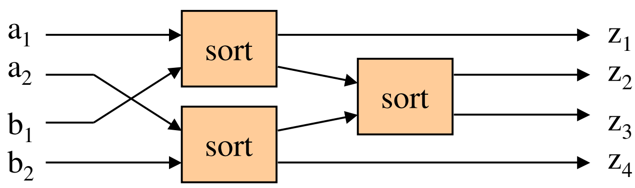
m = 2: (beachte, dass a_1, a_2 bzw. b_1, b_2 sortiert sind.)

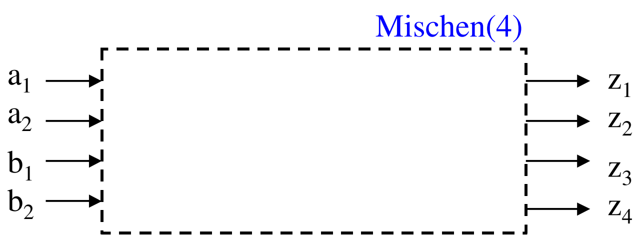


Einsetzen von Mischen(2) ergibt den Baustein Mischen(4):

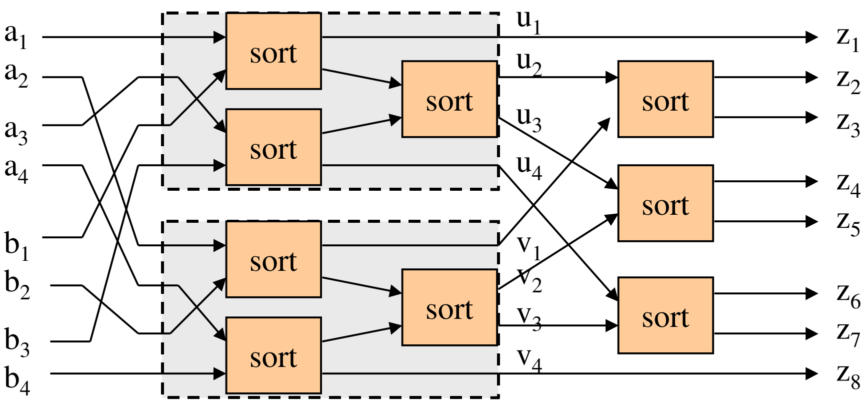


Wir kürzen diesen Baustein durch Mischen(4) ab:

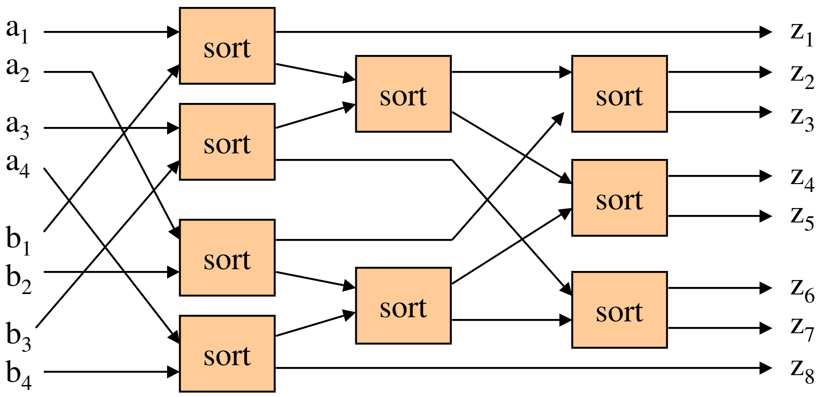




m = 4:

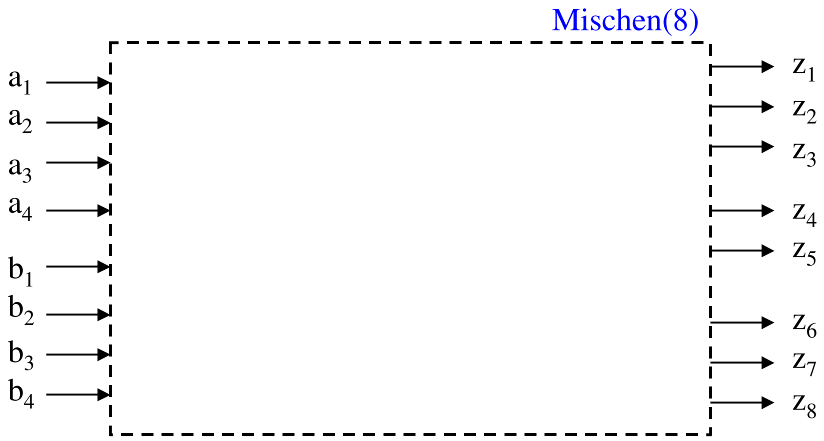


m = 4: Bereinigung der Verbindungen ergibt:

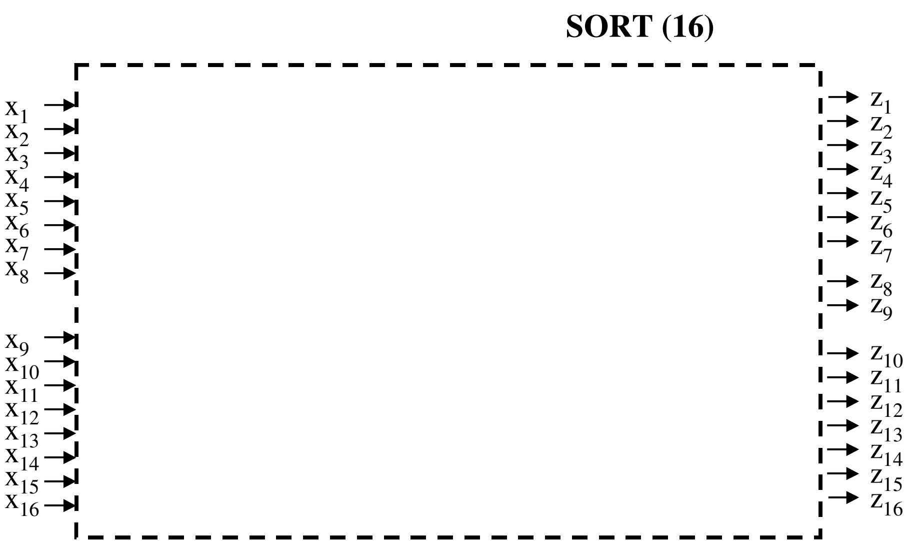


Beachte: a_1, a_2, a_3, a_4 bzw. b_1, b_2, b_3, b_4 sind sortierte Folgen.

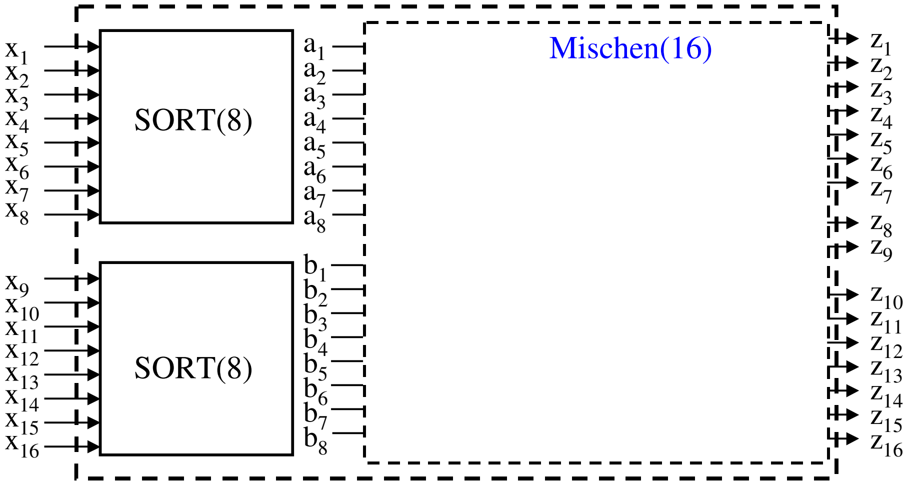
m = 4: Dies kürzen wir erneut ab durch



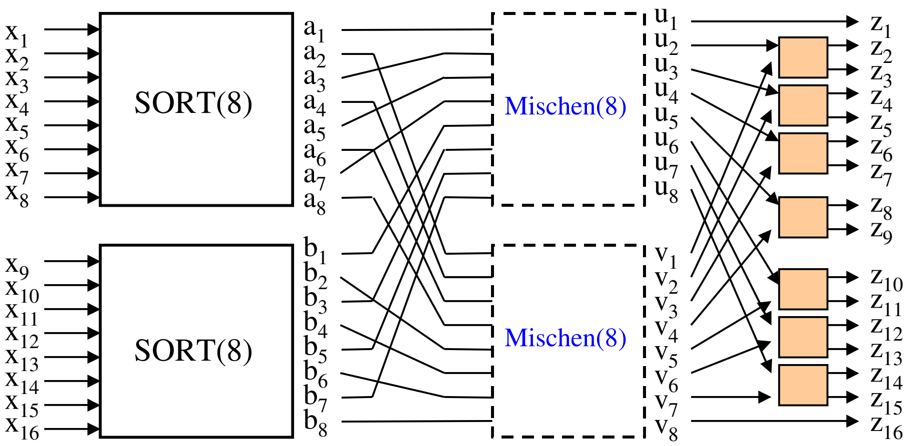
Wir betrachten nun den Gesamtalgorithmus SORT für n=16:



Rekursives Ersetzen für SORT(16) ergibt:

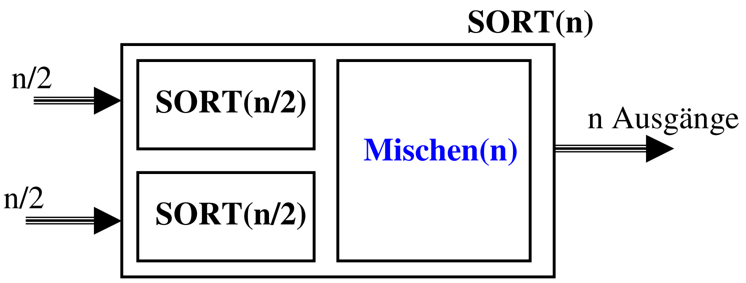


Einsetzen von Mischen(16) ergibt die Struktur:



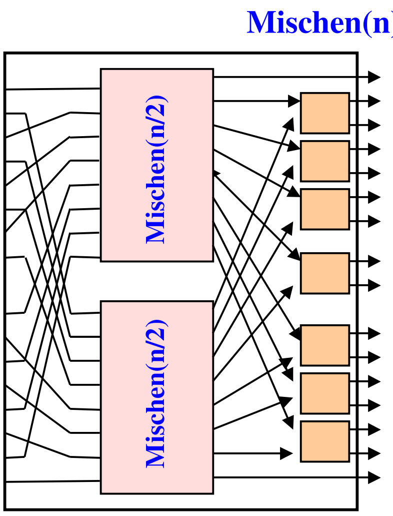
Setzt man alle kleineren schon konstruierten Bausteine ein, so erhält man den auführbaren Algorithmus (selbst durchführen, vgl. dann Folie 55).

Zusammenfassung der rekursiven Struktur des parallelen Sortierens:



Anzahl G(n) der Bausteine **sort**: G(2) = 1 und für $n=2^k$, k>1:
$G(n) = 2 \cdot G(n/2) + M(n)$,
wobei M(n) die Anzahl **sort** in **Mischen(n)** ist.

Mischen(n) ist ebenfalls rekursiv definiert (nächste Folie):



Anzahl M(n) der Bausteine **sort** im Algorithmus Mischen(n):
$M(2) = 1$ und für n>2: $M(n) = 2 \cdot M(n/2) + n/2-1$.

Nachdem der Aufbau des parallelen Sortieralgorithmus klar ist, müssen wir beweisen, dass die Technik "odd-even-merge" die beiden geordneten Folgen $a_1, a_2, ..., a_m$ und $b_1, b_2, ..., b_m$ tatsächlich zu der geordneten Folge $z_1, z_2, ..., z_{2m-1}, z_{2m}$ zusammenmischt.

Satz:

odd-even-merge sortiert die geordneten Folgen $a_1, a_2, ..., a_m$ und $b_1, b_2, ..., b_m$ zur geordneten Folge $z_1, z_2, ..., z_{2m-1}, z_{2m}$.

Beweis:
Wenn m=1 ist, dann werden zwei Zahlen durch den Sortierbaustein **sort** geordnet und der Satz ist richtig.

Sei daher m>1. Gegeben sind zwei sortierte Folgen $a_1, a_2, ..., a_m$ und $b_1, b_2, ..., b_m$. Nach Definition von odd-even-merge werden die sortierten Teilfolgen mit jeweils m/2 Elementen $a_1, a_3, a_5, ..., a_{m-1}$ und $b_1, b_3, b_5, ..., b_{m-1}$ zur sortierten Folge $u_1, u_2, u_3,..., u_m$ bzw. $a_2, a_4, a_6, ..., a_m$ und $b_2, b_4, b_6, ..., b_m$ zur sortierten Folge $v_1, v_2, v_3,..., v_m$ rekursiv gemischt.

Die Ergebnisfolge ist definiert durch $z_1 = u_1$, $z_{2m} = v_m$ und $z_{2i} = \mathrm{Min}(u_{i+1}, v_i)$, $z_{2i+1} = \mathrm{Max}(u_{i+1}, v_i)$ für i=1,2,..,m-1. Wir zeigen, dass die z-Folge hierdurch korrekt sortiert ist.

u_1 muss das Minimum der beiden Folgen sein, da a_1 und b_1 die Minima der a- bzw. b-Folge und beide Elemente in der u-Folge sind. Analog gilt, dass v_m das Maximum der Ergebnisfolge sein muss. Also sind z_1 und z_{2m} richtig bestimmt worden.

Um zu zeigen, dass $z_{2i} = \text{Min}(u_{i+1}, v_i)$, $z_{2i+1} = \text{Max}(u_{i+1}, v_i)$ richtig festgelegt wurden, genügt es zu zeigen, dass sich das Element u_{i+1} in der sortierten Ergebnisfolge an der Position 2i oder 2i+1 befinden muss (analog muss man dies für das Element v_i nachweisen). Wir beweisen dies in acht Schritten (a) bis (h).

Wir betrachten u_{i+1} für ein i zwischen 1 und m-1. O.B.d.A. nehmen wir an, dass dieses Element aus der a-Folge stammt und das j-te Element der Teilfolge mit den ungeraden Indizes ist, d.h., es gibt ein j mit $1 \leq j \leq m/2$ mit $u_{i+1} = a_{2j-1}$.

(a) Wieviele der Elemente der a-Folge sind kleiner als u_{i+1}?
Genau 2j-2 Elemente;
denn weil $u_{i+1} = a_{2j-1}$ ist, müssen die Elemente $a_1, a_2, ..., a_{2j-2}$ der geordneten a-Folge kleiner als u_{i+1} sein.

(b) Wieviele der Elemente der b-Folge sind kleiner als u_{i+1}?
Mindestens 2i-2j+1 Elemente. Beweis hierfür:
Wegen $u_{i+1} = a_{2j-1}$ und wegen $a_1 \leq a_3 \leq a_5 \leq ... \leq a_{2j-3}$ müssen sich unter den ersten i Elementen $u_1, u_2, ..., u_i$ der u-Folge genau diese (j-1) Elemente aus der a-Folge befinden. Folglich müssen unter diesen ersten i Elementen der u-Folge genau i-(j-1) = i-j+1 Elemente der b-Folge sein. Da in der u-Folge aber nur die Elemente mit ungeradem Index sind, müssen dies genau die Elemente $b_1, b_3, b_5, ..., b_{2(i-j+1)-1}$ sein. Da die b-Folge sortiert ist, müssen daher mindestens die Elemente $b_1, b_2, b_3, ..., b_{2(i-j+1)-1}$ kleiner als u_{i+1} sein. Ihre Anzahl ist 2i-2j+1.

(c) Folglich stehen in der sortierten Ergebnisfolge mindestens
2j-2 + 2i-2j+1 = 2i-1 Elemente <u>vor</u> u_{i+1}, d.h., in der Ergebnisfolge kann u_{i+1} frühestens das Element z_{2i} sein.

(d) Wieviele der Elemente der a-Folge sind größer als u_{i+1}?
Genau m-2j+1 Elemente,
denn wegen $u_{i+1} = a_{2j-1}$ müssen $a_{2j}, a_{2j+1}, ..., a_m$ größer als u_{i+1} sein.

(e) Wieviele der Elemente der b-Folge sind größer als u_{i+1}?
Mindestens m-2i+2j-2 Elemente. Beweis hierfür:
Wegen (b) muss $u_{i+1} \leq b_{2(i-j+1)+1}$ sein; denn sonst wäre u_{i+1} nicht das (i+1)-te, sondern ein späteres Element der u-Folge. Also müssen mindestens die Elemente $b_{2(i-j+1)+1}, b_{2(i-j+1)+2}, ..., b_m$ größer als u_{i+1} sein. Ihre Anzahl ist m - (2i-2j+3) + 1 = m-2i+2j-2.

(f) Folglich stehen in der sortierten Ergebnisfolge mindestens
m-2j+1 + m-2i+2j-2 = 2m-2i-1 Elemente <u>hinter</u> u_{i+1}, d.h.,
in der sortierten Ergebnisfolge muss u_{i+1} spätestens das Element $z_{2m-(2m-2i-1)} = z_{2i+1}$ sein.

(g) Nun führe man genau die gleiche Untersuchung für v_i durch.
Auch hier erhält man, dass v_i frühestens das Element z_{2i} und spätestens das Element z_{2i+1} in der Ergebnisfolge sein kann.
(Führen Sie diesen Beweis selbst analog zu (a) bis (f).)

(h) Aus (f) und (g) folgt nun, dass die Elemente u_{i+1} und v_i sich an den Positionen 2i und 2i+1 der z-Folge befinden müssen.
Daher muss $z_{2i} = \text{Min}(u_{i+1}, v_i)$, $z_{2i+1} = \text{Max}(u_{i+1}, v_i)$ für die sortierte Ergebnisfolge $z_1, z_2, ..., z_{2m}$ gelten.

Damit ist der Satz bewiesen.

Skizze zum Beweis: O.B.d.A. sei $u_{i+1} = a_{2j-1}$.

Es sind kleiner als u_{i+1} | Es sind größer als u_{i+1}

$a_1 \leq a_2 \leq \ldots \leq a_{2j-2}$ | $a_{2j} \leq a_{2j+1} \leq \ldots \leq a_m$

$\leq u_{i+1} \leq$

$b_1 \leq b_2 \leq \ldots \leq b_{2(i-j+1)-1}$ | $b_{2(i-j+1)+1} \leq b_{2(i-j+1)+2} \leq \ldots \leq b_m$

$z_1, z_2, \ldots, z_{2i-2}, z_{2i-1}$ $\quad z_{2i} \quad z_{2i+1} \quad$ $z_{2i+2}, z_{2i+3}, \ldots, z_{2m}$

[Wenn $u_{i+1} = a_{2j-1}$ ist, so ist $v_i = b_{2(i-j+1)}$. Beide müssen an den Positionen 2i oder 2i+1 in der sortierten z-Ergebnisfolge stehen. Es ist nur noch zu prüfen, ob a_{2j-1} kleiner oder größer als $b_{2(i-j+1)}$ ist. Folglich gilt $z_{2i} = \mathrm{Min}(u_{i+1}, v_i)$, $z_{2i+1} = \mathrm{Max}(u_{i+1}, v_i)$.]

Nun zur Komplexität:
Wie groß, wie breit und wie tief ist dieser parallele Sortieralgorithmus?

"Größe" soll ein Maß für die "Herstellungskosten" sein, wenn man den Algorithmus hardwaremäßig realisiert:
G(n) = Anzahl der Bausteine **sort** in **SORT(n)**.

"Tiefe" soll die Laufzeit des Sortierens angeben:
T(n) = Länge des längsten gerichteten Weges durch Bausteine **sort** in **SORT(n)**.

"Breite" soll die Anzahl der Mikroprozessoren angeben, die für eine Softwarelösung benötigt werden:
B(n) = Minimale Zahl der Bausteine **sort**, die parallel zueinander liegen müssen, ohne die Tiefe zu verändern.

Siehe Folien 26 und 27:

Anzahl G(n) der Bausteine **sort**: G(2) = 1 und für $n=2^k$, k>1:
$G(n) = 2\cdot G(n/2) + M(n)$,
wobei M(n) die Anzahl **sort** in Mischen(n) ist:

M(2) = 1 und für $n=2^k$, k>1: $M(n) = 2\cdot M(n/2) + n/2-1$.

Für die Tiefe gilt: T(n) = T(n/2) + "Tiefe von Mischen(n)".

Die Breite beträgt B(n) = n/2. Denn von jedem Eingang x_i muss ein längster Weg ausgehen und die Anzahl der Bausteine, die unmittelbar hinter den Eingängen liegen, beträgt n/2. Im weiteren Verlauf der Rekursion kommt man mit dieser Zahl auch aus, siehe die Formeln für SORT(n) und für Mischen(n). Dies lässt sich auch formal beweisen, worauf wir hier verzichten.

Einige Größen der Mischen-Bausteine: M(2) = 1, M(4) = 3, M(8) = 9, M(16) = 25, M(32) = 65. Aus $M(n) = 2\cdot M(n/2) + n/2-1$ gewinnt man durch Einsetzen rasch die Gleichungen:

$$\begin{aligned}
M(n) &= 2\cdot M(n/2) + n/2-1 \\
&= 2\cdot(2\cdot M(n/4) + n/4-1) + n/2-1 \\
&= 4\cdot M(n/4) + 2\cdot n/2 - 1 - 2 \\
&= 4\cdot(2\cdot M(n/8) + n/8-1) + 2\cdot n/2 - 1 - 2 \\
&= 8\cdot M(n/8) + 3\cdot n/2 - 1 - 2 - 4 \\
&= \ldots \\
&= 2^{\log(n)-1}\cdot M(2) + (\log(n)-1)\cdot n/2 - (2^{\log(n)-1} - 1) \\
&= n/2 + n/2\cdot\log(n) - n/2 - n/2 + 1 \\
&= n/2 \cdot (\log(n) - 1) + 1
\end{aligned}$$

Es gilt also $M(n) = n/2 \cdot (\log(n) - 1) + 1$.

Hiermit können wir nun die "Größe" G(n) ausrechnen:

$$
\begin{aligned}
G(n) &= 2\cdot G(n/2) + M(n)\\
&= 2\cdot G(n/2) + n/2 \cdot (\log(n) - 1) + 1\\
&= 2\cdot(2\cdot G(n/4) + n/4 \cdot (\log(n/2) - 1) + 1) + n/2 \cdot (\log(n) - 1) + 1\\
&= 4\cdot G(n/4) + n/2 \cdot (\log(n) - 2) + n/2 \cdot (\log(n) - 1) +1 +2\\
&= 4\cdot(2\cdot G(n/8) + n/8 \cdot (\log(n/4) - 1) + 1)\\
&\qquad + n/2 \cdot (\log(n) - 2) + n/2 \cdot (\log(n) - 1) +1 +2\\
&= 8\cdot G(n/8) + n/2 \cdot (\log(n) - 3)\\
&\qquad + n/2 \cdot (\log(n) - 2) + n/2 \cdot (\log(n) - 1) +1 +2 + 4\\
&= \ldots\\
&= 2^{\log(n)-1}\cdot G(2) + n/2\cdot(1+2+\ldots+(\log(n) - 1)) +1+2+\ldots+2^{\log(n)-2}\\
&= 2^{\log(n)-1} + n/4\cdot\log(n)\cdot(\log(n)-1)) + 2^{\log(n)-1} - 1\\
&= (n/4)\cdot\log(n)\cdot(\log(n)-1)) + n - 1.
\end{aligned}
$$

Ergebnis: $\mathbf{G(n) = (n/4)\cdot\log(n)\cdot(\log(n)-1)) + n - 1} \in O(n\cdot\log^2(n))$.

Entscheidend für den praktischen Einsatz ist die Tiefe T(n).
Hierzu betrachten wir Folie 27:
Die Tiefe von Mischen(n) ist die Tiefe von Mischen(n/2) + 1,
woraus sofort die Tiefe log(n) für den Teil Mischen(n) folgt.

Nach Folie 26 gilt dann:

$$
\begin{aligned}
T(n) &= T(n/2) + \log(n)\\
&= T(n/4) + \log(n/2) + \log(n) = T(n/4) + \log(n)-1 + \log(n)\\
&= T(n/8) + \log(n/4) + \log(n/2) + \log(n)\\
&= \ldots\\
&= T(2) + \log(n/2^{\log(n)-2}) + \log(n/2^{\log(n)-3}) + \ldots + \log(n)-1 + \log(n)\\
&= 1 + 2 + 3 + \ldots + \log(n)\\
&= (1/2)\cdot\log(n)\cdot(\log(n) + 1).
\end{aligned}
$$

Ergebnis: $\mathbf{T(n) = (1/2)\cdot\log(n)\cdot(\log(n) + 1)} \in O(\log^2(n))$.

$G(n) = (n/4) \cdot \log(n) \cdot (\log(n)-1)) + n - 1$

$T(n) = (1/2) \cdot \log(n) \cdot (\log(n) + 1)$

Faustformel:

$G(n) \approx n/2 \cdot T(n)$.

n Eingänge	G(n) Größe	T(n) Tiefe
2	1	1
4	5	3
8	19	6
16	63	10
32	191	15
64	543	21
128	1471	28
256	3839	36
512	9727	45
1024	24063	55

n Eingänge	G(n) Größe	T(n) Tiefe
16	63	10
128	1471	28
1024	24063	55
16384	1490957	105

Mit 24.063 Bausteinen **sort** kann man also in 55 Schritten 1024 Zahlen sortieren.

Mit etwa 100 Millionen Bausteinen **sort** kann man in nur 210 Schritten rund 1 Million Zahlen sortieren.

Solche Algorithmen sind technisch durchaus realisierbar. Sie können in Zukunft die Leistungsfähigkeit beim Sortieren deutlich steigern.

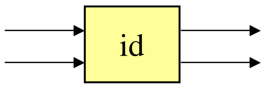
Durchdenken Sie eine mögliche Realisierung weiter, z.B.:

- Gibt es Probleme bei der Hardware-Realisierung? Lassen sich die vielen Leitungen problemlos verschalten / auf Platten drucken? ... Jede Leitung in unserer Skizze besitzt eine "Breite", z.B. 64 Bits zuzüglich Kontrollbits.
- Wie kann man den Sortierbaustein **sort** realisieren? Nehmen Sie an, dass die zu sortierenden Schlüssel k-stellige 0-1-Folgen sind, wobei man wegen der vielfältig möglichen Schlüssel von k=64 ausgehen sollte. Was ist dann die minimale Tiefe (=längster Weg über Gatter von einem Eingang zu einem Ausgang) von **sort**?
- Wie kann man Millionen von Daten *gleichzeitig* an alle Eingänge legen? Welche Strukturen müssen die Speicher hierfür besitzen?

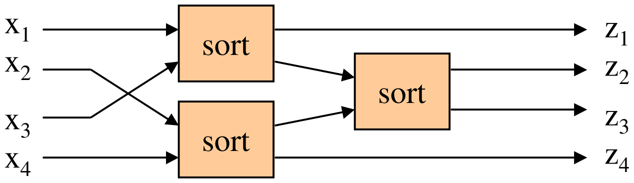
Wie sehen softwaremäßige Realisierungen aus?

Bei n Eingabewerten kann man n/2 Prozeduren definieren, die jeweils "den nächsten Gesamtschritt" (= alle parallel durchführbaren Sortierschritte) ausführen. Hierfür greifen sie immer wieder auf das Feld der zu sortierenden Daten x: <u>array</u>(1..n) <u>of</u> <Elementtyp> zu, aus dem in jedem Schritt gelesen und in das in jedem Schritt geschrieben wird.

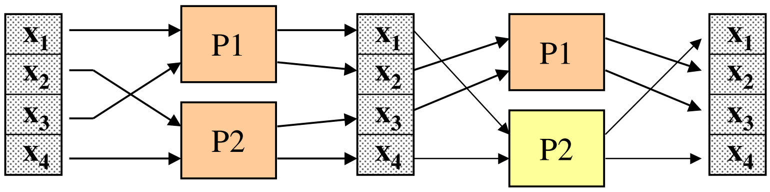
Im Algorithmus gibt es Teile, in denen ein Wert nur weitergereicht wird. Hierfür führen wir den zusätzlichen Baustein "id" (Identität) ein, der zwei Werte einliest und unverändert wieder ausgibt:

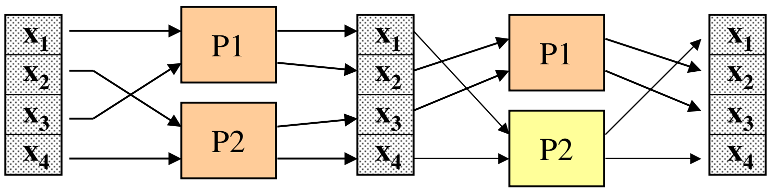


Der Mischbaustein Mischen (4) (siehe Folie 19)



wird dann realisiert durch zwei Prozeduren P1 und P2:

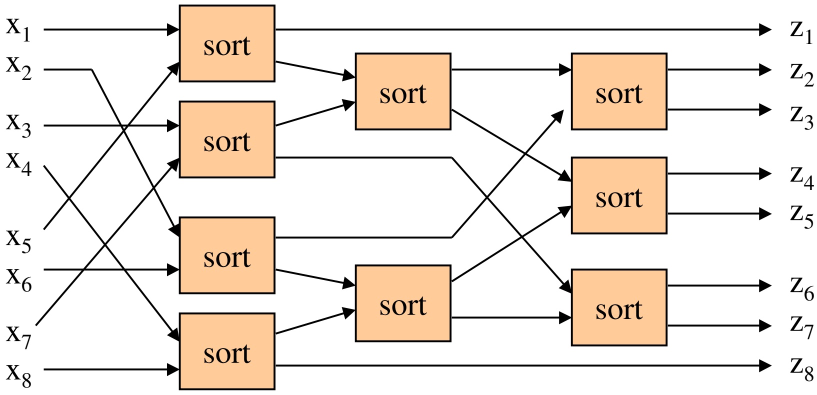




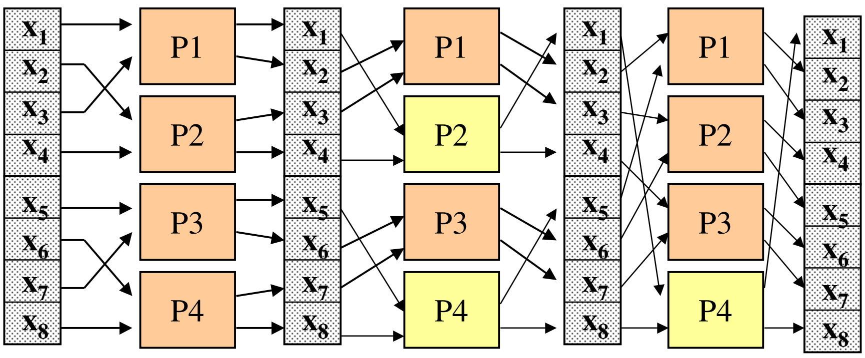

```
procedure P1 is              -- x: array(1..n) of <Elementtyp> ist global
h1, h2: <Elementtyp>;
begin (h1, h2) := sort (x(1), x(3));  x(1) := h1; x(2) := h2; $
      (h1, h2) := sort (x(2), x(3));  x(2) := h1; x(3) := h2;
end;
procedure P2 is              -- x: array(1..n) of <Elementtyp> ist global
h1, h2: <Elementtyp>;
begin (h1, h2) := sort (x(2), x(4));  x(3) := h1; x(4) := h2; $
      (h1, h2) := id (x(1), x(4));    x(1) := h1; x(4) := h2;
end;
```

$ steht für "Synchronisation"

Der Mischbaustein Mischen(8)



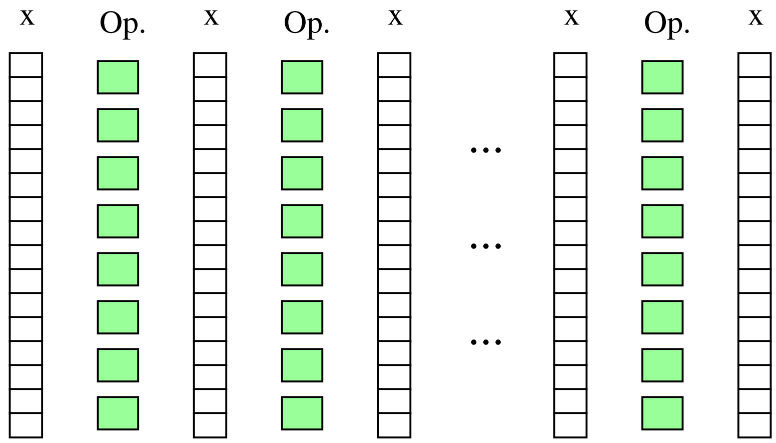
wird dann realisiert durch vier Prozeduren, die ihre Arbeitsweise synchronisieren müssen:



<u>procedure</u> Mischen8 <u>is</u>
    <u>procedure</u> P1 <u>is</u> <u>begin</u> ... <u>end</u>;
    <u>procedure</u> P2 <u>is</u> <u>begin</u> ... <u>end</u>;
    <u>procedure</u> P3 <u>is</u> <u>begin</u> ... <u>end</u>;
    <u>procedure</u> P4 <u>is</u> <u>begin</u> ... <u>end</u>;
<u>begin</u> P1; P2; P3; P4; <u>end</u>;

Die Synchronisiation ist innerhalb der Prozeduren P1 bis P4 sicherzustellen.

Man erhält auf diese Weise eine "normierte Darstellung":



x = Datenschicht, Op. = Operationsschicht. Jeder Operationsbaustein **sort** oder **id** hat zwei einlaufende und zwei ausgehende Verbindungen.

Jede Operationsschicht lässt sich durch n/2 Prozeduren softwaremäßig beschreiben; diese Prozedruen können wir beenfalls für die nächste Operationsschicht verwenden usw., so dass wir aus Software-Sicht folgendes Ergebnis erhalten (die Operationen **id** kann man natürlich im Programm weglassen, nicht aber die Synchronisierung):

Mit n/2 Prozeduren lassen sich n Elemente in
$T(n) = (1/2) \cdot \log(n) \cdot (\log(n) + 1)$
Schritten sortieren, wobei jede der n/2 Prozeduren aus einem sequentiellen Programmstück mit bis zu T(n) einzelnen Sortieroperationen **sort** besteht und T(n)-1 Synchronisations-anweisungen besitzt.

Diese Prozeduren können automatisch durch ein Programm erzeugt werden; sie hängen nur vom Parameter n ab.

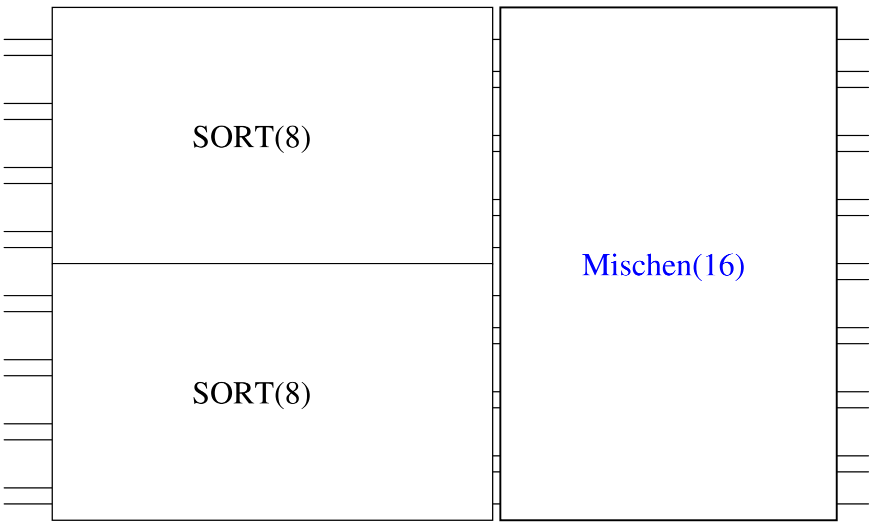
Wie müssen Programmiersprachen beschaffen sein, damit der synchrone Ablauf und die Verzögerungen gesichert werden?

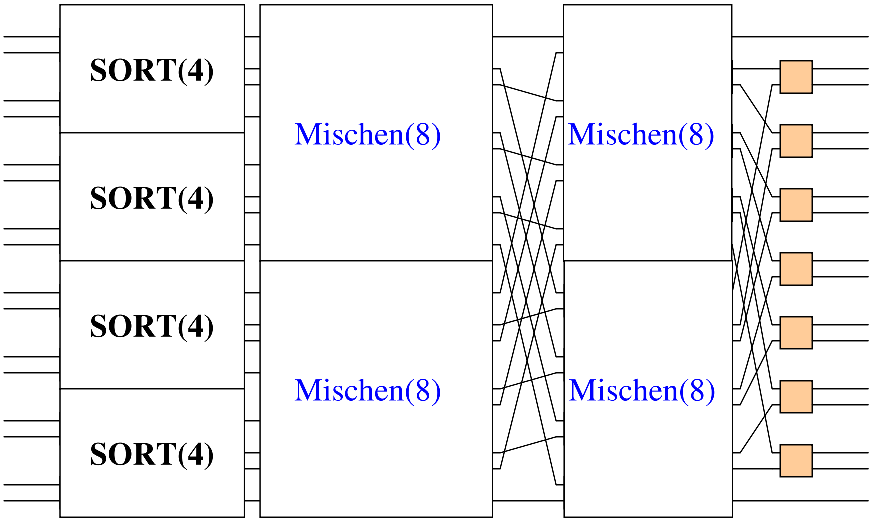
Diese Sprachen müssen in der Regel mehrere Anforderungen erfüllen:

- Teile eines Programms müssen unabhängig voneinander ablaufen können (Nebenläufigkeit, "tasks").
- Es muss Synchronisationsmechanismen geben.
- Man muss Zeitvorgaben (innerhalb von ... Mikrosekunden führe durch ....) machen können und es muss Verzögerungs-elemente (delay) geben.
- Es muss der parallele Zugriff auf Daten möglich sein.
- Es muss ...

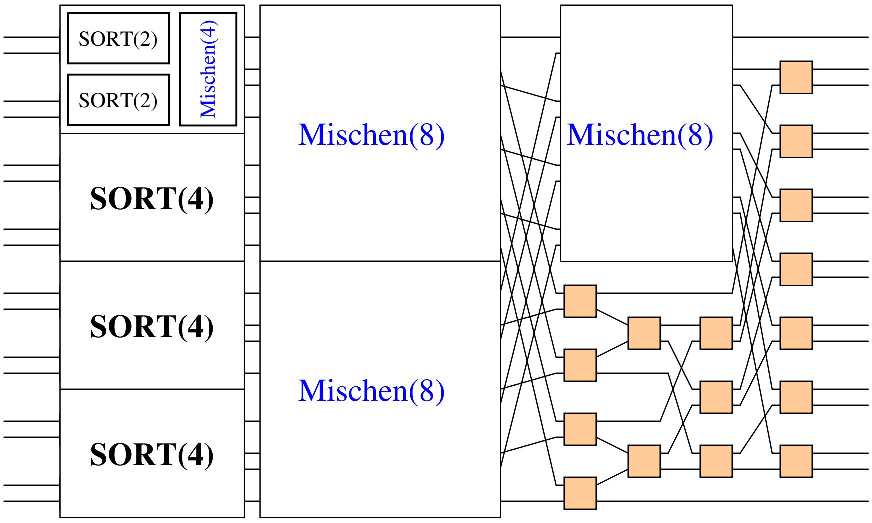
Siehe weiteres Studium, Praktika, Projekte .....

Zum Abschluss das vollständige Beispiel SORT(16)

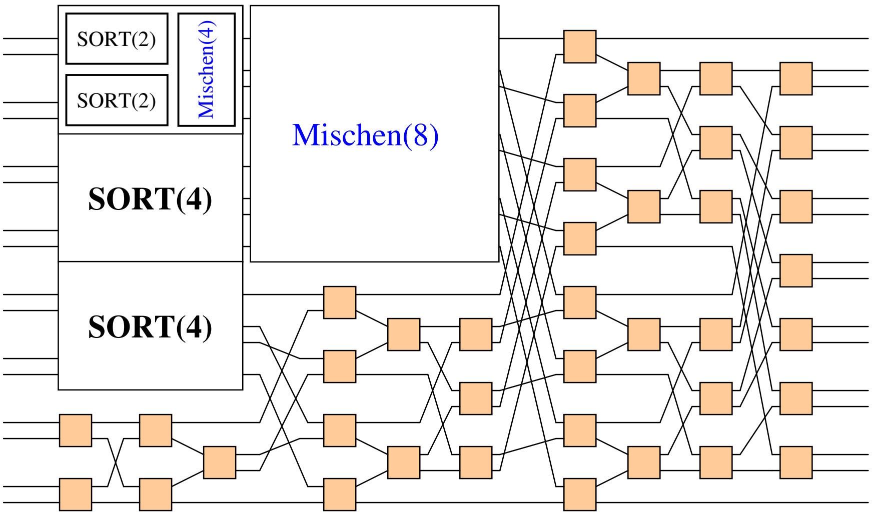



SORT(4)
SORT(4)
SORT(4)
SORT(4)
Mischen(8)
Mischen(8)
Mischen(8)
Mischen(8)

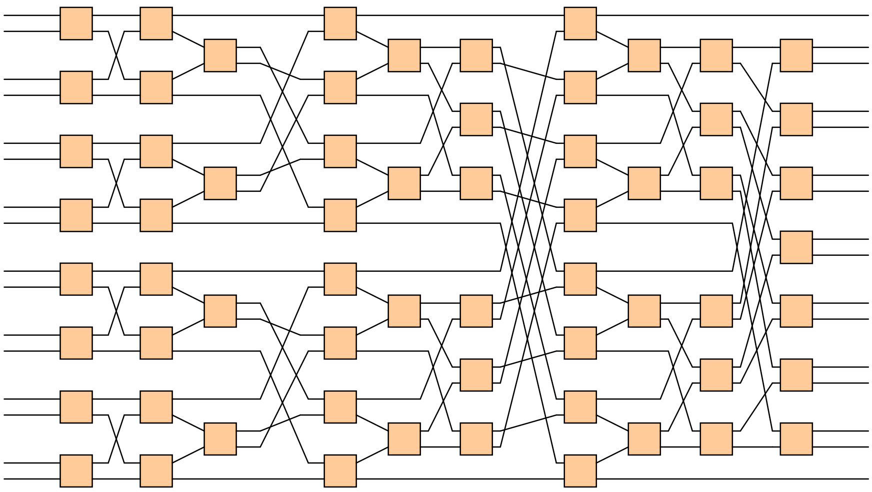
SORT(16)


SORT(2)
SORT(2)
Mischen(4)
SORT(4)
SORT(4)
SORT(4)
Mischen(8)
Mischen(8)
Mischen(8)

SORT(16)



SORT(16)



63 Bausteine **sort**, größte Tiefe: 10, Breite: 8.