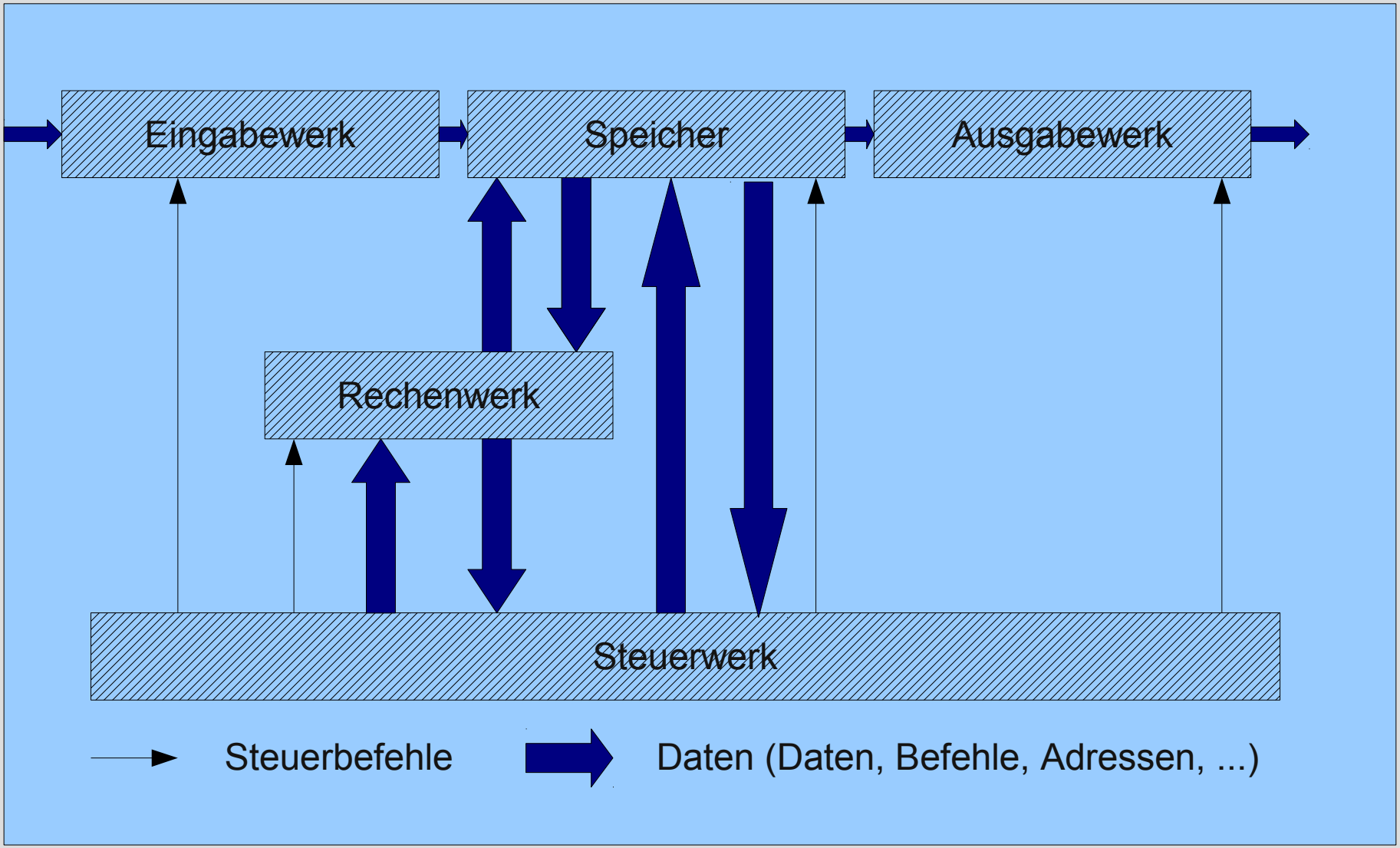
Parallele Rechnerarchitekturen

- Bisher behandelte von Neumann-Architektur:

v.-Neumann-Architektur



Parallele Rechnerarchitekturen

- Bisher behandelte von Neumann-Architektur
  - entscheidender Nachteil:
  - zu einem Zeitpunkt kann nur ein Maschinenbefehl geholt und verarbeitet werden.
  - diese Sequentialität ist aber nicht zwingend.

Parallele Rechnerarchitekturen

- Bisher behandelte von Neumann-Architektur
  - entscheidender Nachteil:
    - zu einem Zeitpunkt kann nur ein Maschinenbefehl geholt und verarbeitet werden.
    - diese Sequentialität ist aber nicht zwingend.
  - Ausweg:
    - statt *sequentieller* Ausführung **parallele** Ausführung.

Parallele Rechnerarchitekturen

- Bisher behandelte von Neumann-Architektur
  - entscheidender Nachteil:
    - zu einem Zeitpunkt kann nur ein Maschinenbefehl geholt und verarbeitet werden.
    - diese Sequentialität ist aber nicht zwingend.
  - Ausweg:
    - statt *sequentieller* Ausführung **parallele** Ausführung.
  - 2 Ausprägungen der Parallelität:
    - zeitliche Parallelität
    - räumliche Parallelität

Zeitliche Parallelität

- Datenflussanalyse von Algorithmen
- Ziel:
  - besseres Verstehen von
    - Problemen
    - Lösungsansätzen

Zeitliche Parallelität

- Datenflussanalyse von Algorithmen
- Ziel:
  - besseres Verstehen von
    - Problemen
    - Lösungsansätzen
  - Auffinden von Operationen, die nicht zwingend in der Reihenfolge ausgeführt werden müssen, in der sie im Programmtext auftreten.

Zeitliche Parallelität (2)

Programmfragment

```
x := y+3;                  {1}
if x < 5 then begin        {2}
    a := x – 2;            {3}
    b := x + 6;            {4}
    c := b / x;            {5}
    b := 3 * y;            {6}
    a := 11 * x;           {7}
  end;
else
     c := 6;               {8}
```

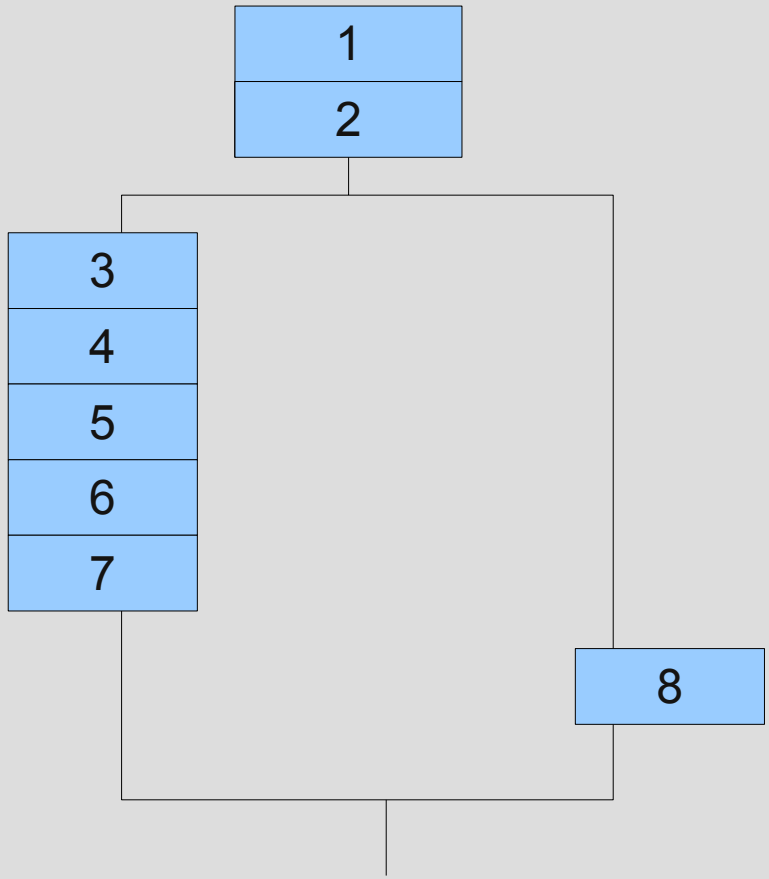
- Datenflussgraph:

Zeitliche Parallelität (2)

Programmfragment

```
x := y+3;                  {1}
if x < 5 then begin        {2}
    a := x – 2;            {3}
    b := x + 6;            {4}
    c := b / x;            {5}
    b := 3 * y;            {6}
    a := 11 * x;           {7}
  end;
else
     c := 6;               {8}
```

Datenflussgraph(PAP)



Zeitliche Parallelität (3)

- es läßt sich leicht folgendes erkennen:
  - {1} muss immer vor {2} ausgeführt werden
  - die Abarbeitungsreihenfolge von {3} und {4} ist egal

Zeitliche Parallelität (4)

- allgemeiner:
  - es lassen sich Abhängigkeiten finden, die der parallelen Ausführung und damit der Aussage: „Reihenfolge ist beliebig“ entgegenstehen:

Zeitliche Parallelität (4)

Steuerflussabhängigkeit (*control*)

Die Ausführung von x entscheidet darüber, ob y ausgeführt wird oder nicht. ⟶ x muss immer vor y ausgeführt werden.

$v_1(x) \xrightarrow{c} v_2(y)$

im Beispiel: $v_1(2) \xrightarrow{c} v_2(3)$

....

$v_1(2) \xrightarrow{c} v_2(8)$

Zeitliche Parallelität (5)

Datenflussabhängigkeit

Echte Datenflussabhängigkeit (True)

$v_1(x) \longrightarrow_t v_2(y)$

es gibt wenigsten eine Variable z, die in beiden Anweisungen x und y auftritt, wobei der Zugriff auf z in x schreibend (i.d.R. steht z auf der linken Seite der Anweisung) und in y lesend ist, d.h. x muss stets vor y ausgeführt werden.

im Beispiel:

$v_1(1) \longrightarrow_t v_2(3)$

Zeitliche Parallelität (6)

Datenflussabhängigkeit

Ausgangsdatenabhängigkeit (Output)

$v_1(x) \longrightarrow {}_o v_2(y)$

es gibt wenigstens eine Variable z, die in beiden Anweisungen x und y auftritt, wobei der Zugriff auf z in beiden Anweisungen schreibend erfolgt, d.h. x **muss** stets vor y ausgeführt werden.

im Beispiel:

$v_1\{3\} \longrightarrow {}_o v_2\{7\}$ aber

$v_1\{5\} \longrightarrow {}_o v_2\{8\}$ entfällt, da {5} und {8} alternativ ausgeführt werden

Zeitliche Parallelität (7)

Datenflussabhängigkeit

Anti-Datenabhängigkeit (Anti)

$v_1(x) \xrightarrow{}{}_a v_2(y)$

es gibt wenigstens eine Variable z, die in beiden Anweisungen x und y auftritt, wobei der Zugriff auf z in x lesend und in y schreibend ausgeführt werden. Auch hier muss x stets vor y ausgeführt werden, damit sichergestellt wird, dass in x der „alte“ Wert von z gelesen wird bevor er in y mit einem „neuen“ Wert überschrieben wird.

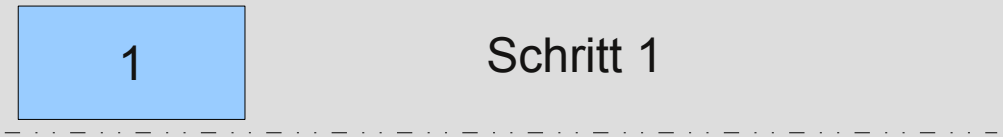
$v_1(5) \xrightarrow{}{}_a v_2(6)$ (paralleles Ausführen ist möglich, wenn Lesen im gleichen Takttyklus vor Schreiben erfolgt, siehe auch Abschnitt Pipelining)

Zeitliche Parallelität (Übung)

- Aufgabe:
  Ermitteln Sie alle Steuer- und
  Datenflussabhängigkeiten für das gegebene
  Programmfragment

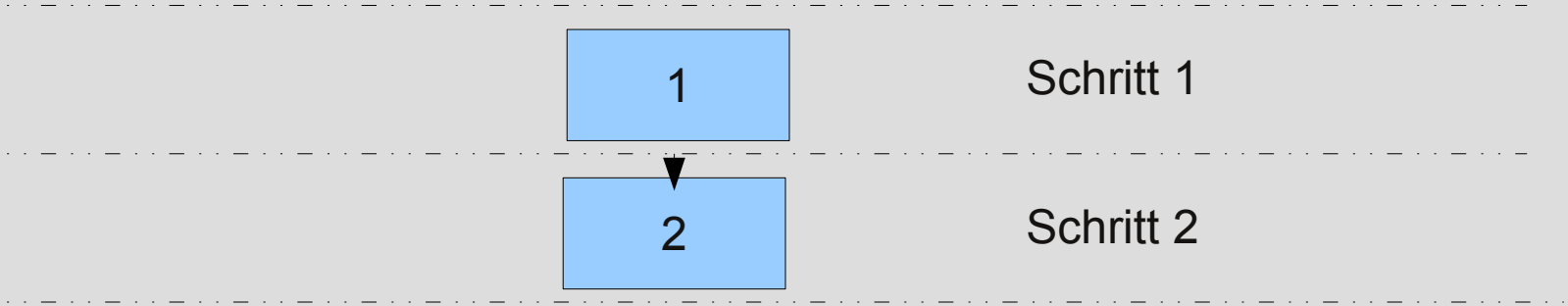
Zeitliche Parallelität (8)

Zusammenfassung:
maximal paralleler Datenflussgraph:



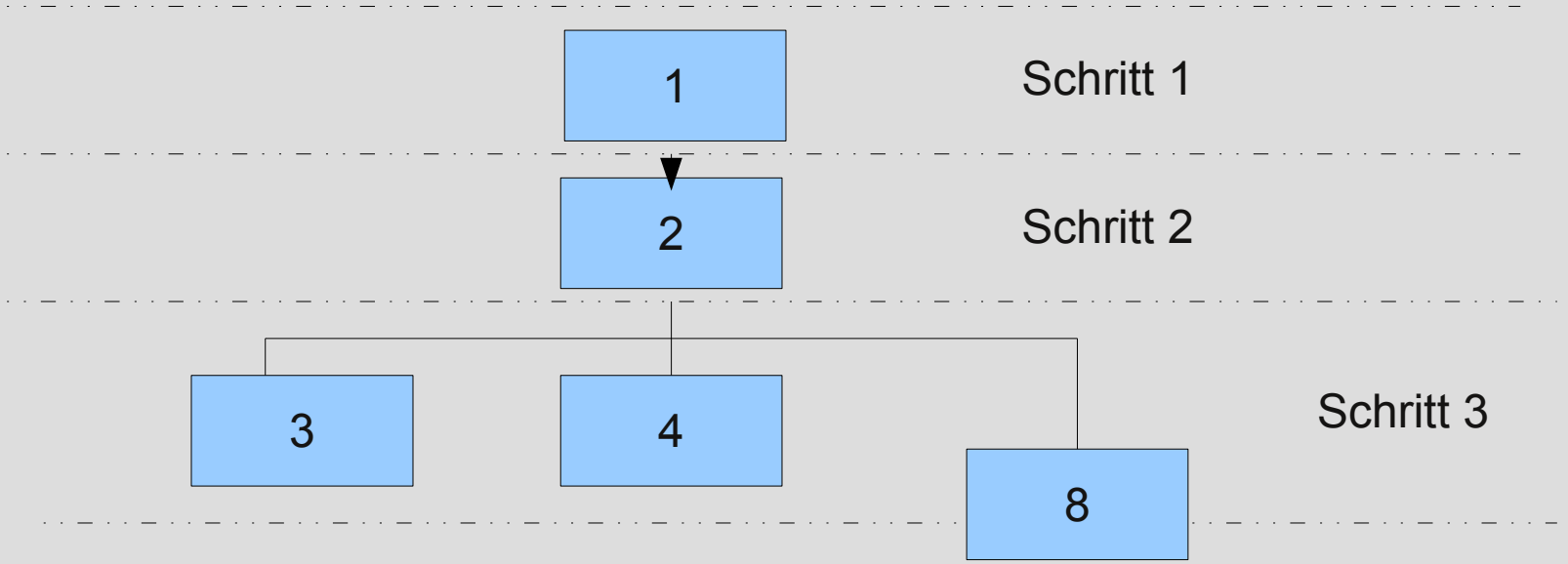
Zeitliche Parallelität (8)

Zusammenfassung:
maximal paralleler Datenflussgraph:



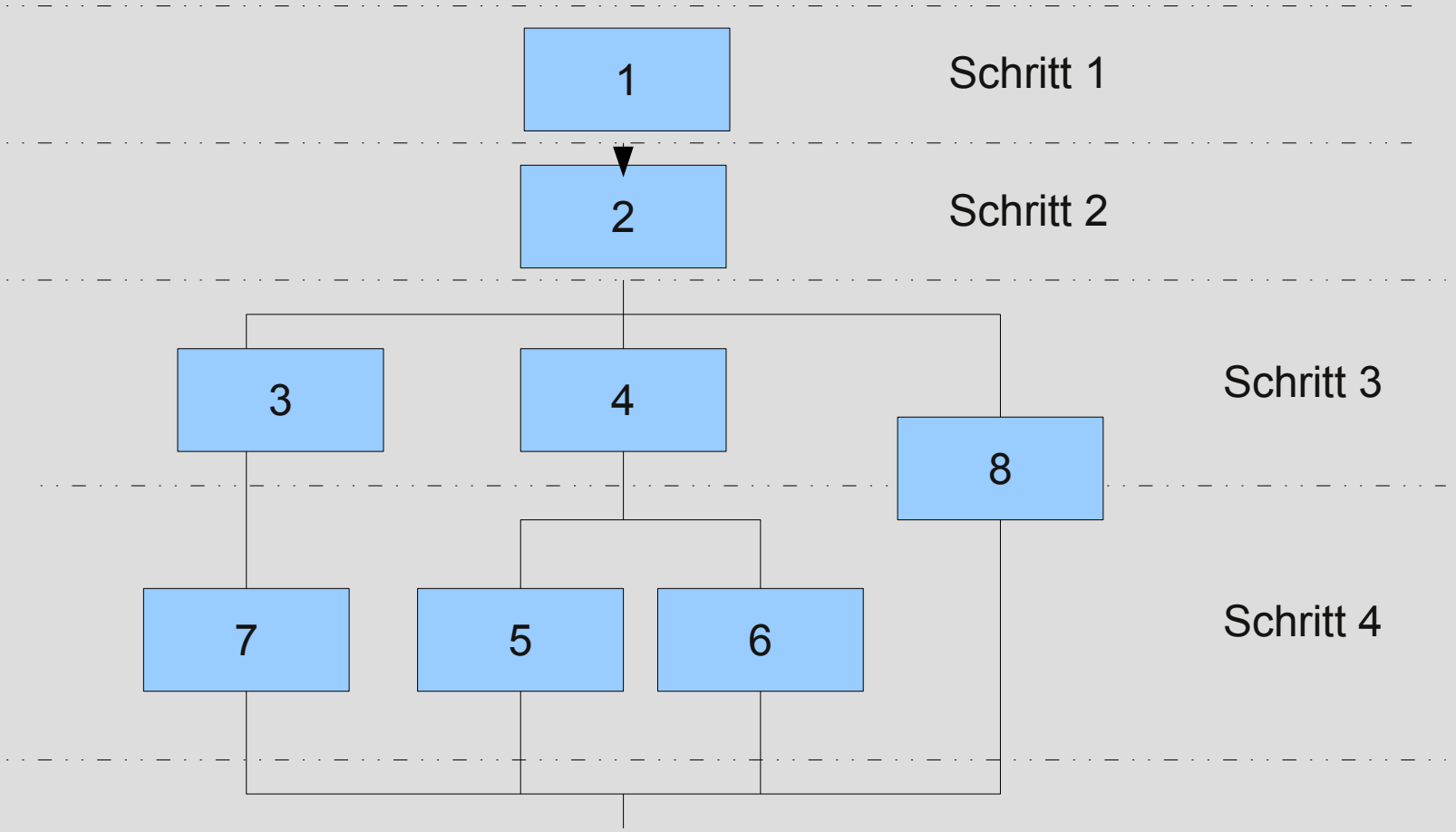
Zeitliche Parallelität (8)

Zusammenfassung:
maximal paralleler Datenflussgraph:



Zeitliche Parallelität (8)

Zusammenfassung:
maximal paralleler Datenflussgraph:



Zeitliche Parallelität (9)

Zusammenfassung:

- um {3} und {4} parallel ausführen zu können, benötigt man 2 ALUs
- um {5}, {6} und {7} parallel ausführen zu können, benötigt man 3 ALUs
- da {8} alternativ zu {3} bis {7} ausgeführt wird, kann {8} entweder ausgeführt werden, wenn {3} und {4} oder wenn {5}, {6} und {7} ausgeführt würden

Zeitliche Parallelität (10)

Zusammenfassung:

- jeder Algorithmus, jedes Programm hat ein gewisses Potential, einige Operationen zeitlich parallel ausführen zu können
- Zeitliche Parallelität hat ihren Preis
- der Datenflussanalyse ist eingeeignetes Werkzeug, die maximal zeitliche Parallelität zu ermitteln
- mathematische Grundlagenforschung zum Entwurf von Algorithmen, die sich besonders gut parallelisieren lasse, ist notwendig (FEM, Bildverarbeitung, SFB 393 an der TU Chemnitz)

Zeitliche Parallelität (Übung 2)

- Aufgabe:

  Gegeben sei das Programmfragment (folgende Seite). Ermitteln Sie die Steuer- und Datenflussabhängigkeiten sowie den maximal parallelen Datenflussgraphen

Zeitliche Parallelität (Übung 2)

- Programmfragment: {a, b, c, d seien definiert)

```
cin >> a;
b = 7;
c = a % b;
if (!c) {
   d = a – b;
   cout << d;
   cin >> a; }
else {
   d = b – a + c;
   cout << d;
   cin >> b; }
```

Zeitliche Parallelität (Übung 2)

- Programmfragment: {a, b, c, d seien definiert)

```
cin >> a;                 //1
B = 7;                    //2
c = a % b;                //3
if (!c) {                 //4
   d = a – b;             //5
   cout << d;             //6
   cin >> a; }            //7
else {
   d = b – a + c;         //8
   cout << d;             //9
   cin >> b; }            //10
```