Karlsruher Institut für Technologie

Institut für Prozessdatenverarbeitung und Elektronik

Institut für Technik der Informationsverarbeitung

Kapitel 3: Schätzung der Entwurfsqualität

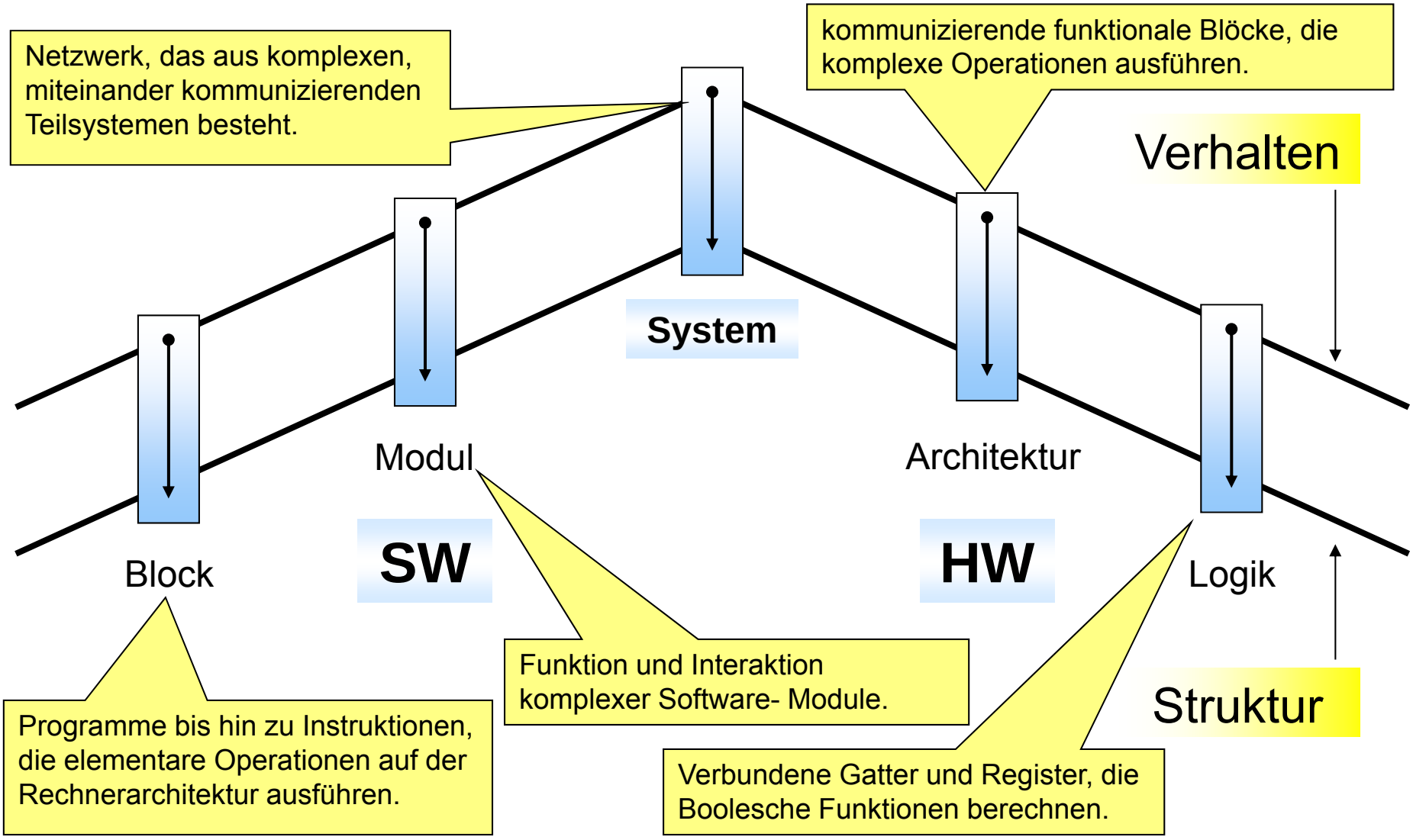
Dr.-Ing. Oliver Sander
Dipl.-Inform. Leonard Masing

Institut für Technik der Informationsverarbeitung (ITIV)

Hardware/Software Co-Design

Inhalt

3.1 Abstraktionsebenen



3.2 Synthese

- Verhalten
  - Funktionen unabhängig von ihrer konkreten Implementierung beschrieben.
- Struktur
  - Kommunizierende Komponenten werden beschrieben.
  - Die Aufteilung und Kommunikation entsprechen der tatsächlichen Implementierung.
- Syntheseaufgaben
  - **Allokation**: Auswahl von Komponenten.
  - **Bindung**: Zuweisung von Funktionen zu Komponenten.
  - **Ablaufplanung**: Festlegen der Ausführungsreihenfolge.

3.2 Systemsynthese (I)

- Kurze Entwurfszyklen:
  - Automatisiertes Entwurfssystem führt Entwurf schneller durch als es ein Entwickler ohne Unterstützung von CAD-Werkzeugen könnte.
  - Beispiel für große Zeitersparnis: Werkzeuge zum automatisierten Platzieren und Verdrahten von Leiterplatten und integrierten Schaltungen.
  - Reduktion der Produktlebensdauer
    - Reduzierte time-to-market feststellbar
    - Einsatz geeigneter CAD-Verfahren unabdingbar.

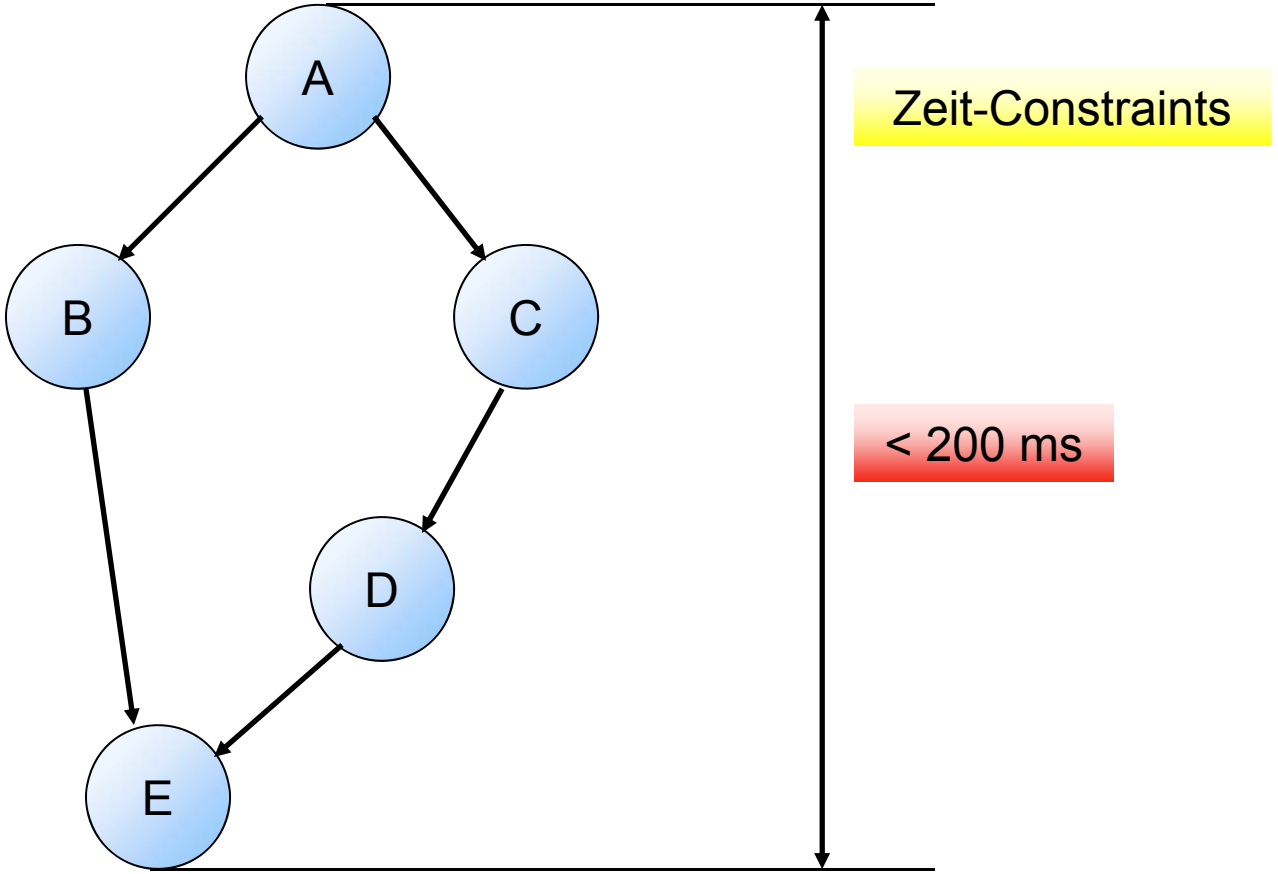
- Reduzierte Entwurfsfehler:
  - Vermeidung kostspieliger Iterationen aufgrund von Entwurfsfehlern durch Entwicklung von Synthesewerkzeugen, die beweisbar korrekte Entwürfe liefern.
  - Verfeinerungsvorgang von Verhaltensbeschreibung hin zur strukturellen Beschreibung
    - Sequenz von Programmtransformationen (formale Verifikationsverfahren)

3.2 Systemsynthese (II)

- Exploration des Entwurfsraumes:
  - Entwurfsentscheidungen auf den obersten Entwurfsebenen bestimmen die Leistungsfähigkeit und Kosten des implementierten Systems.
  - Entwurfswerkzeug auf Systemebene erlaubt schnelle Bewertung unterschiedlicher Realisierungsarten einer Spezifikation unter verschiedenen Optimierungsgesichtspunkten.
  - Die Konsequenzen von Fehlentscheidungen werden in einem frühen Entwurfsstadium deutlich (Bsp. Verletzung von Zeit- oder Kostenbeschränkungen).

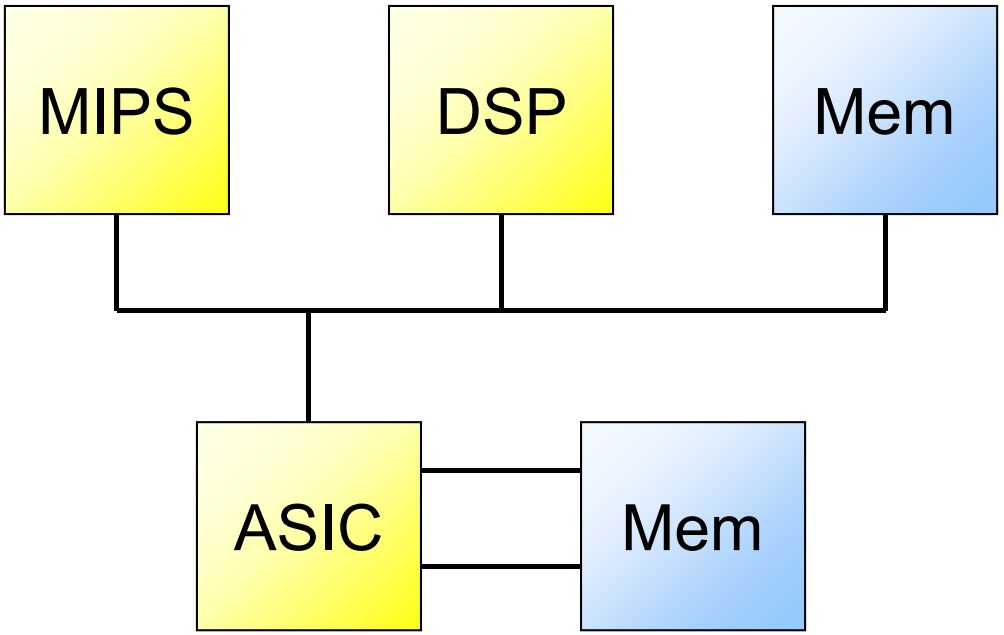
3.2 Spezifikation auf Systemebene

- Taskgraph
  - bestehend aus sequentieller und paralleler Abfolge von Tasks.



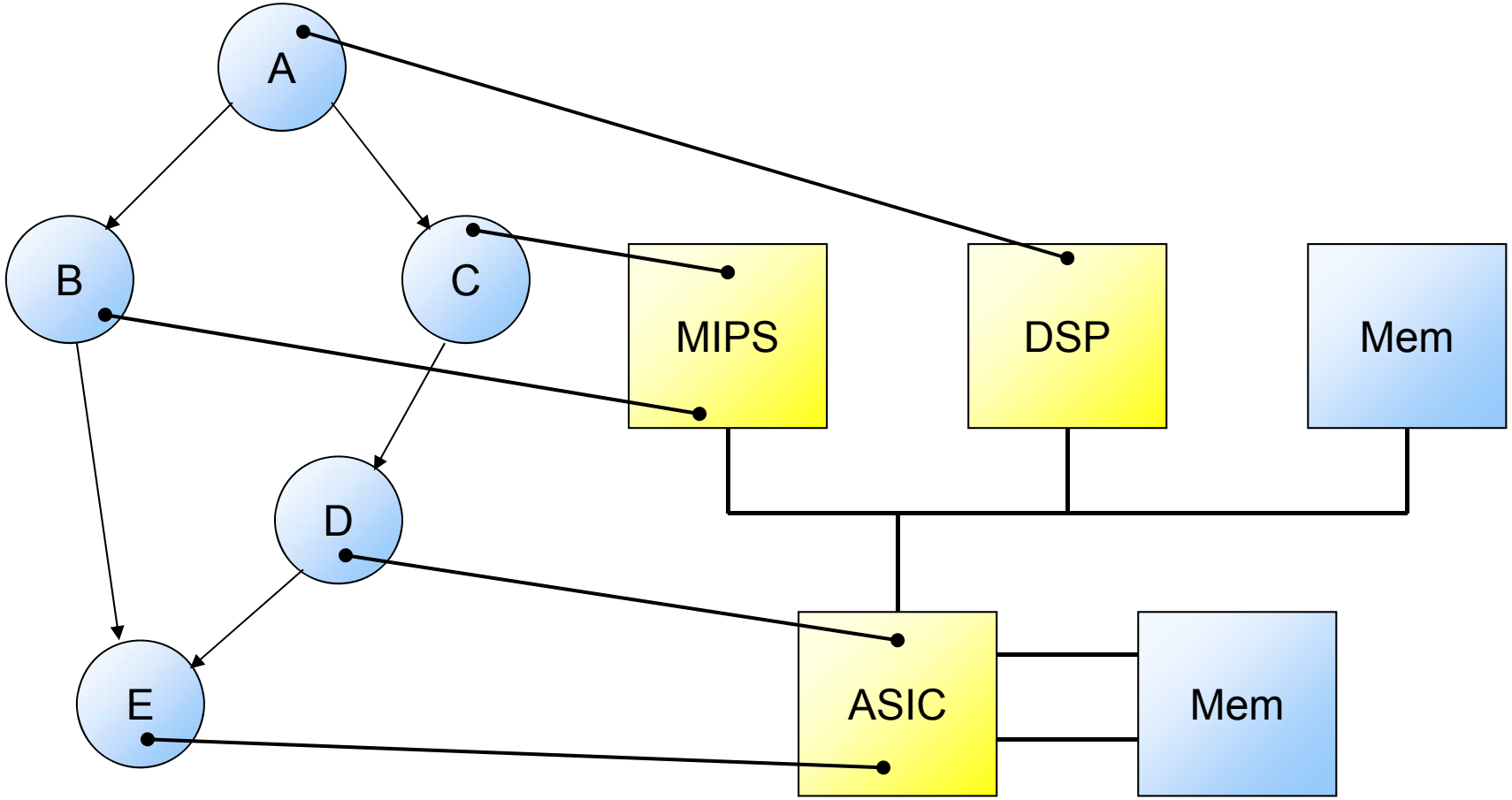
3.2 Allokation auf Systemebene

- Prozessoren, dedizierte Hardwarebausteine
- Speicher, I/O
- Verbindungsstrukturen

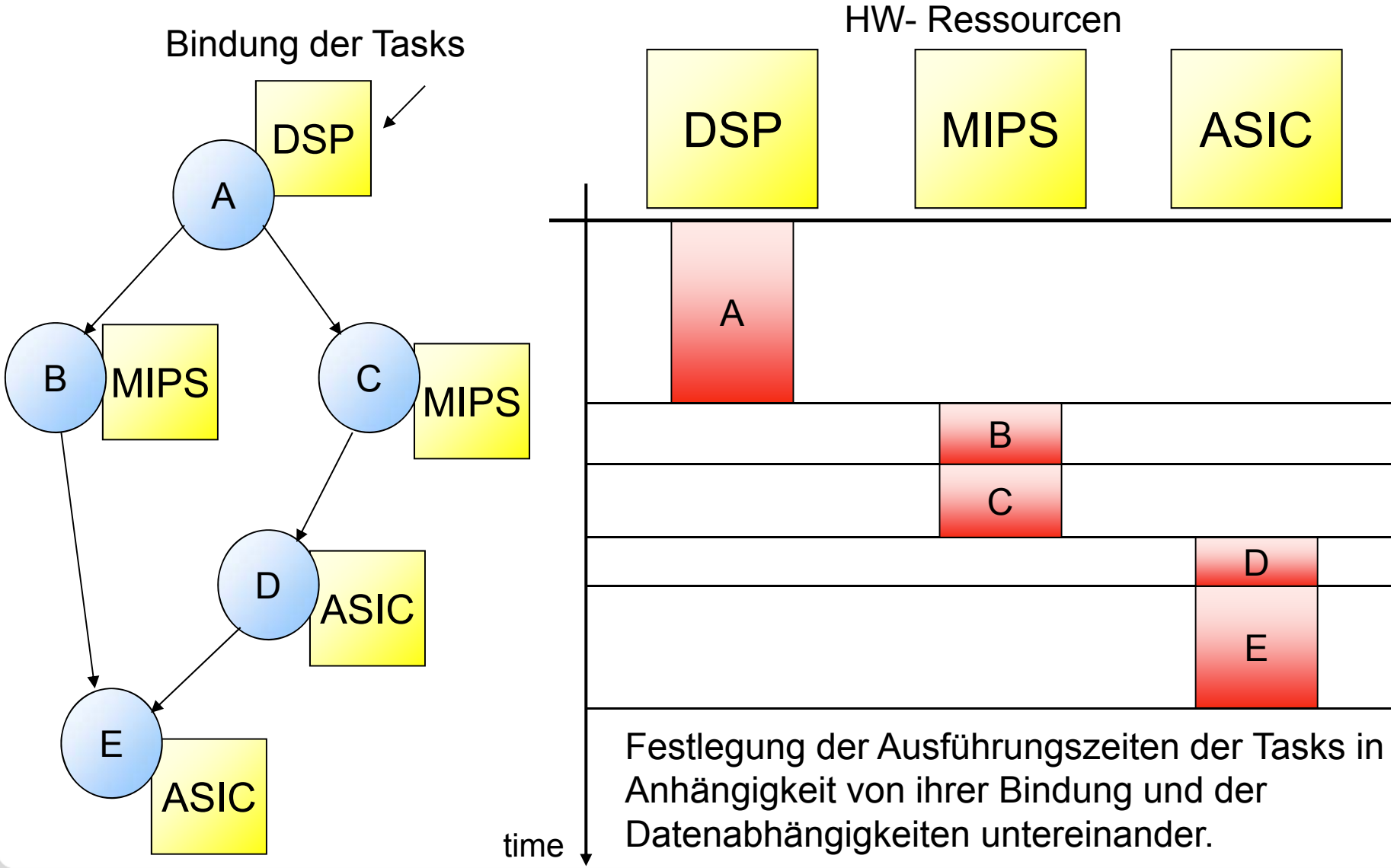


3.2 Bindung auf Systemebene

- Abbildung und Bindung der Tasks auf Prozessoren oder ASICs.



3.2 Ablaufplanung auf Systemebene

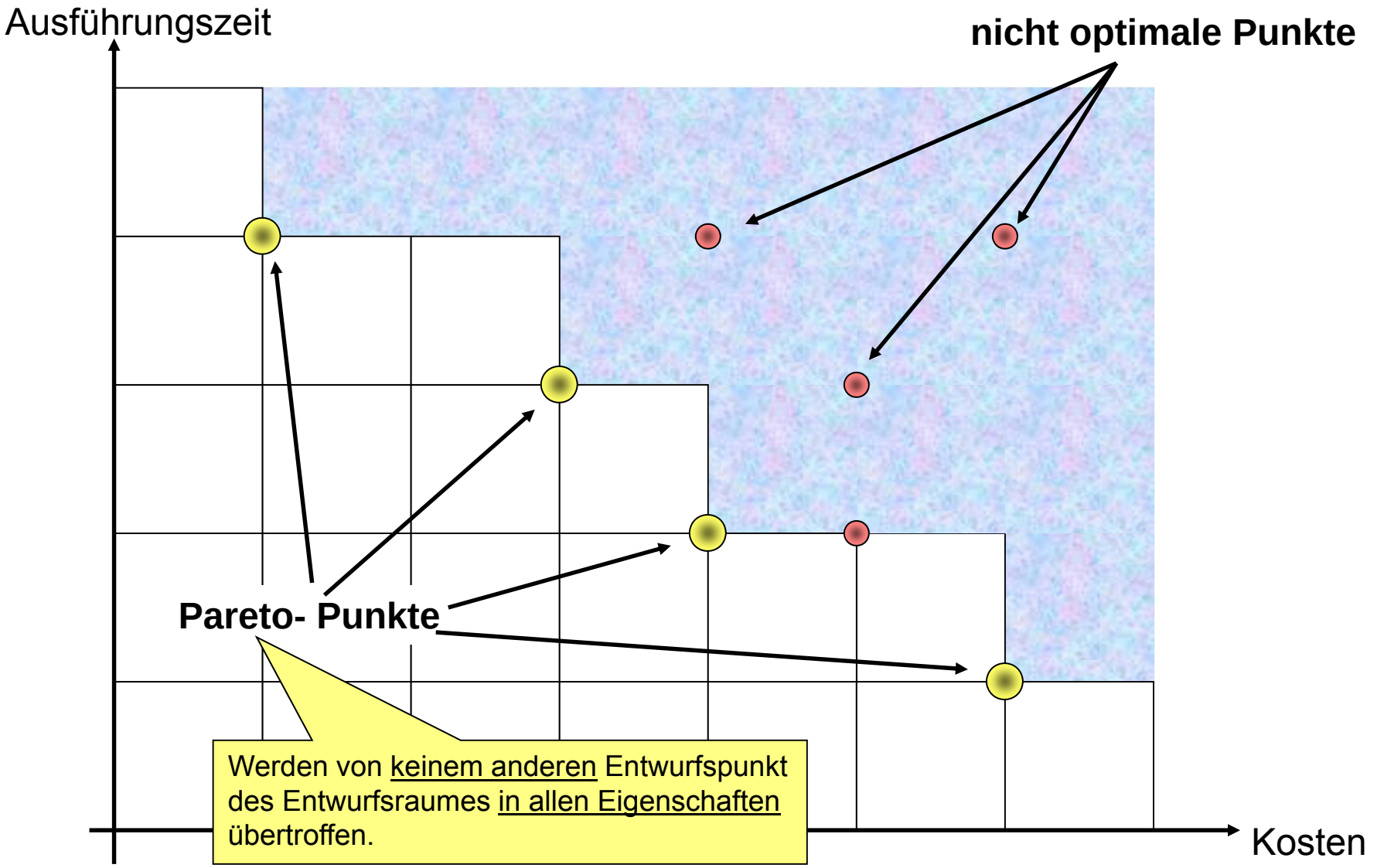


Festlegung der Ausführungszeiten der Tasks in Anhängigkeit von ihrer Bindung und der Datenabhängigkeiten untereinander.

- Was ist Synthese? Welche Aufgaben hat sie?
- Was sind Ziele der Systemsynthese?
- Was sind Allokation und Binding?
- Mehr dazu in der Vorlesung Hardware Synthese & Optimierung (HSO) von Prof. Becker im Sommersemester

Inhalt

3.3 Optimale Entwurfspunkte

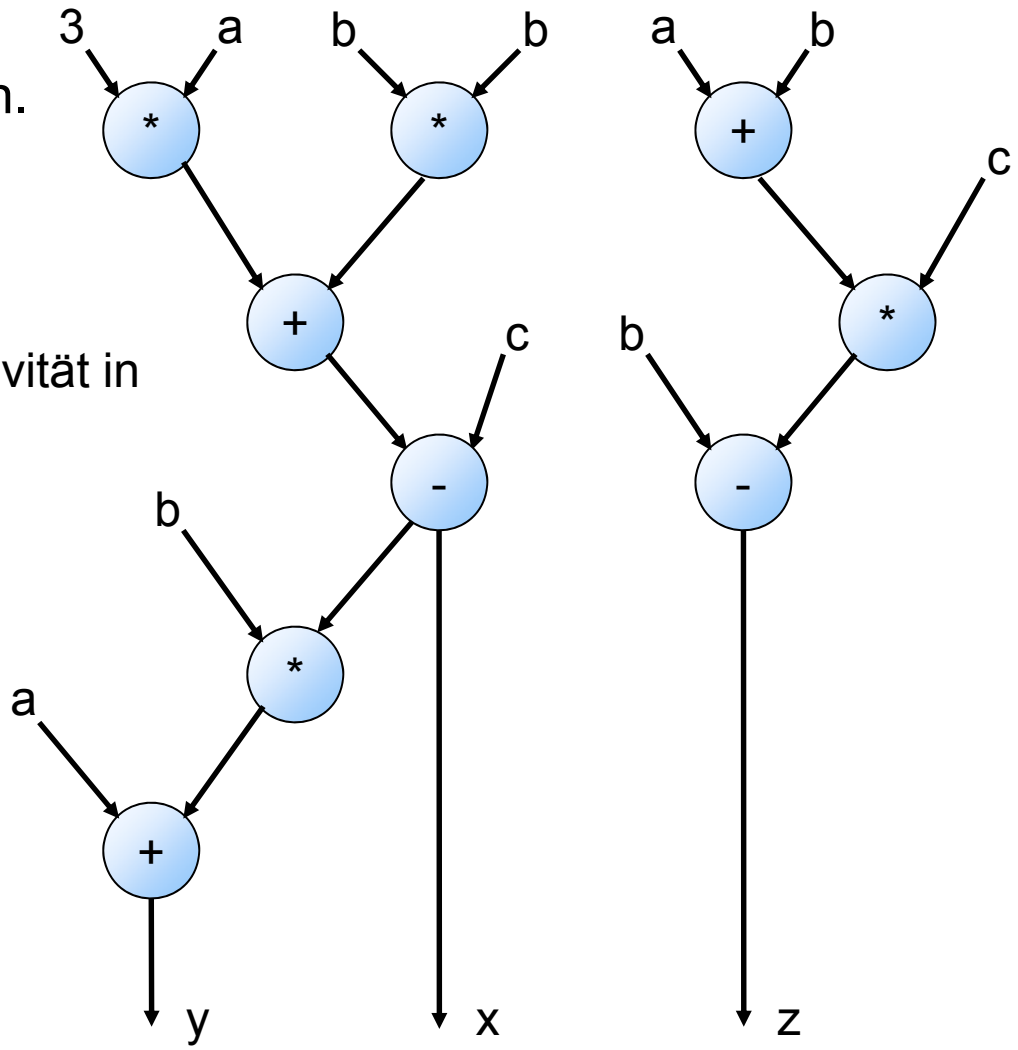


3.3 Graphenmodelle: Kontroll- /Datenfluß

- Graph G(V,E)
  - Menge der Knoten V: Operationen, Tasks
  - Menge der Kanten E: Abhängigkeiten
- Abhängigkeiten
  - Datenabhängigkeit (data dependency)
  - Kontrollflußabhängigkeit (control dependency)
  - Ressourcenkonflikt (entsteht durch die Implementierung)
- Modelle
  - Datenflußgraphen (DFGs)
  - Kontrollflußgraphen (CFGs)
  - gemischte Kontroll/Datenflußgraphen (CDFGs)
    (Bsp.: Sequenzgraphen, nach De Micheli)

3.3 Datenflußgraphen (DFG's)

- Modellierung von Operationen und ihrer Datenabhängigkeiten.
- Nachteile:
  - keine explizite Ausführungs-reihenfolge der Operationen.
  - Kommutativität und Assoziativität in den Ausdrücken erlauben verschiedene DFGs.
  - keine Modellierung von Kontrollstrukturen, wie z.B. Verzweigungen und Iterationen.

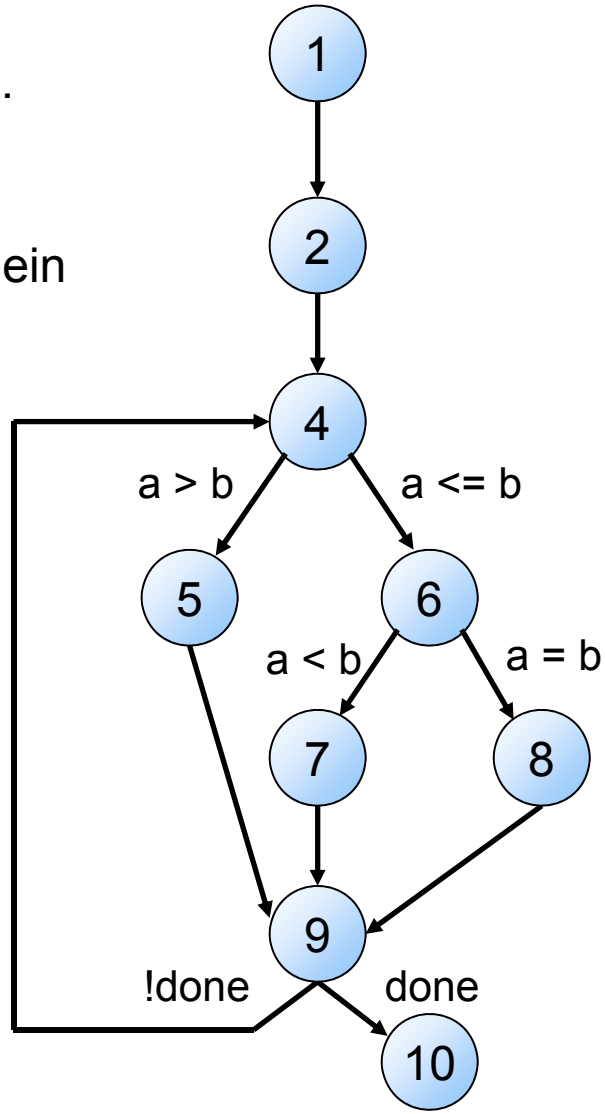


- x = 3 • a + b • b – c;
- y = a + b • x;
- z = b – c • (a + b);

3.3 Kontrollflußgraphen (CFG's)

- Erlaubt Modellierung von Kontrollstrukturen, wie z.B. Verzweigungen und Iterationen.
- Der von einem Verzweigungsknoten ausgehende Programmfluss ist alternativ, d.h. es wird nur genau ein Nachfolgeast durchlaufen.
- Nachteil: Möglichkeit der parallelen Ausführung von Anweisungen wird von vorneherein nicht betrachtet.
- Lösung: hierarchische Sequenzgraphen.

```
1   what_is_this {
2       read (a,b);
3       done = FALSE;
4       repeat {
5           if (a>b)
6               a=a-b;
7           elseif (b>a)
8               b=b-a;
9           else done = TRUE;
10      } until done;
11      write (a);
12  }
```



3.3 Hierarchische Sequenzgraphen (I)

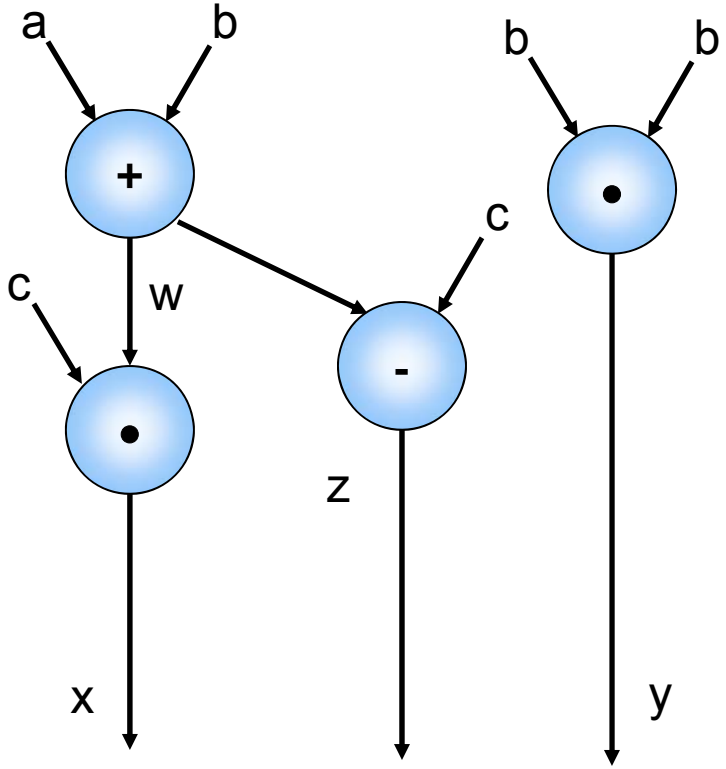
- Hierarchie von verketteten Einheiten
  - die Einheiten modellieren den Datenfluß
  - die Hierarchie modelliert den Kontrollfluß
- Spezielle Knoten
  - Start/Endknoten: NOP (no operation)
  - Verzweigungsknoten (BRanch)
  - Iterationsknoten (LOOP)
  - Modulaufrufknoten (CALL)
- Attribute
  - **Knoten**: Berechnungszeiten, Kosten, …
  - **Kanten**: Bedingungen für Verzweigungen und Iterationen

3.3 Hierarchische Sequenzgraphen (II)

- Aus einem gegebenen Datenflussgraphen erhält man den zugehörigen Sequenzgraphen durch folgende Schritte:
  - Entfernen aller Eingangskanten, die zu Knoten ohne Vorgängerknoten führen.
  - Einfügen des NOP- Startknotens mit je einer Kante zu allen Knoten, die keine Eingangskanten besitzen.
  - Entfernen aller Ausgangskanten, die von Knoten ohne Nachfolgerknoten wegführen.
  - Einfügen eines NOP- Endknotens mit je einer Kante von jedem Knoten ohne Nachfolger zu diesem Endknoten.

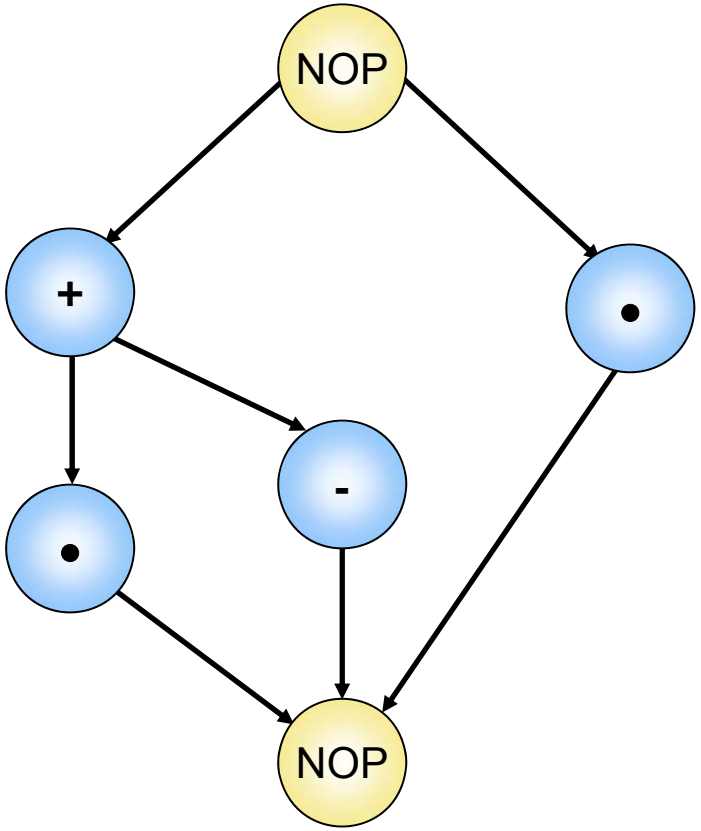
3.3 Sequenzgraphen – Einheit

```
w= a + b;
x= w • c;
y= b • b;
z= w - c;
```



Datenflußgraph

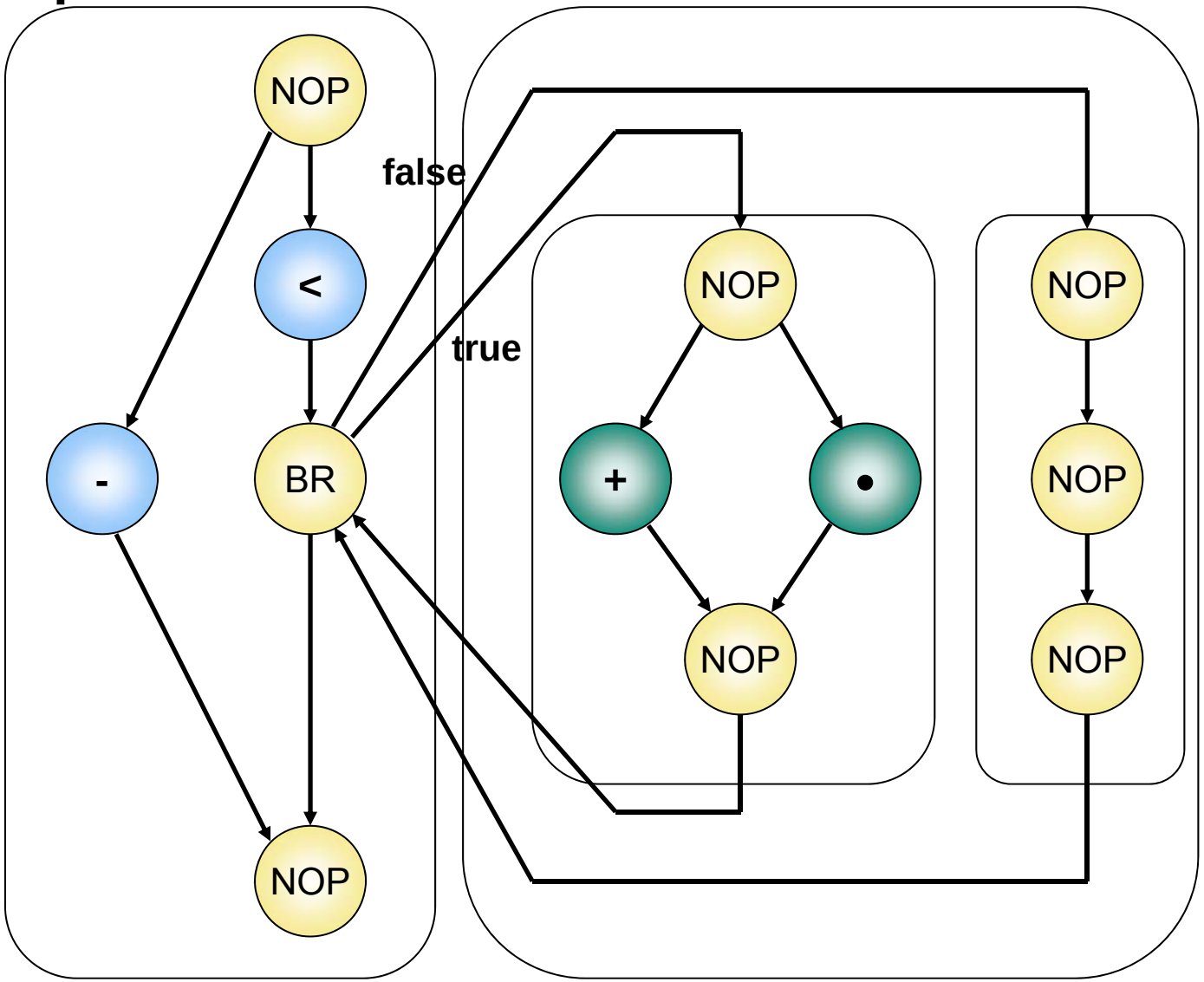
Sequenzgraph – Einheit



Azyklischer polarer Graph

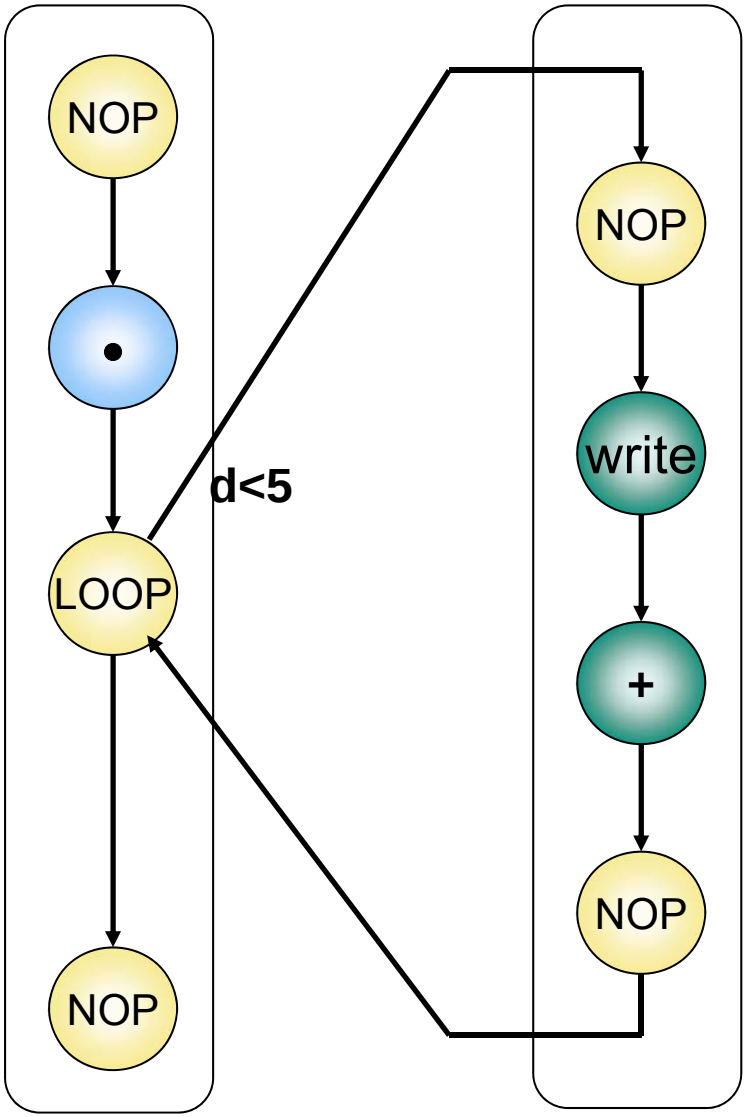
3.3 Sequenzgraphen - Branch

```
c=a<b;
IF ( c ) THEN
    p=m+n;
    q=m*n;
ENDIF
x=a-b;
```



3.3 Sequenzgraphen – Loop

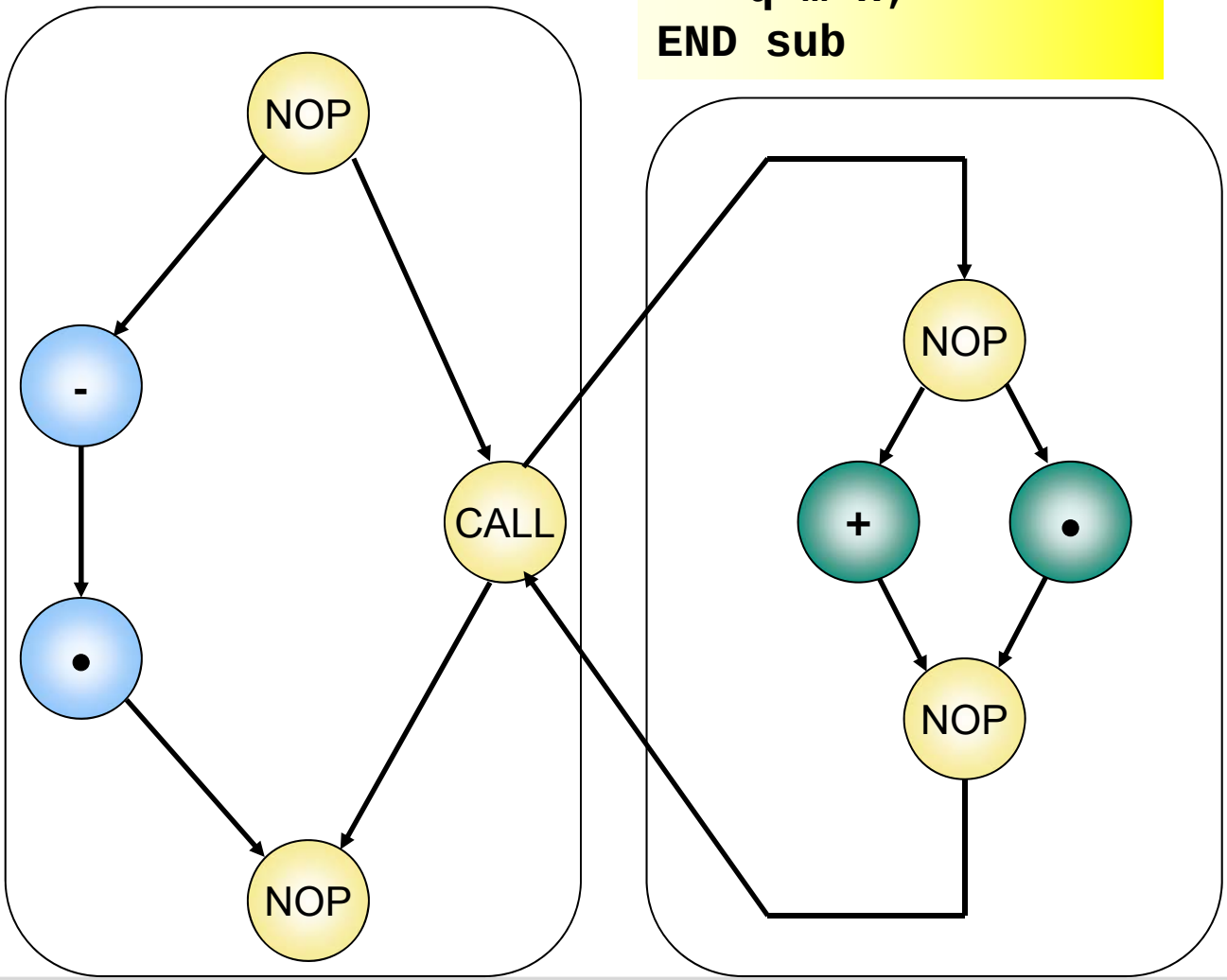
```
d= 2*x;
WHILE (d<5) DO
    write(d);
    d=d+1;
ENDWHILE
```



3.3 Sequenzgraphen - Call

```
PROCEDURE sub(m,n)
    p=m+n;
    q=m*n;
END sub
```

```
d=x-y;
e=d*x;
sub(x,y);
...
```



- Was ist Pareto-optimal?
- Welche Graphenmodelle gibt es? Welche Eigenschalften haben sie?
- In welchen Graphen kann Kontrollfluss abgebildet werden?
- Wie wird Programmcode in Graphen dargestellt?

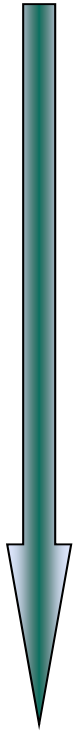
Inhalt

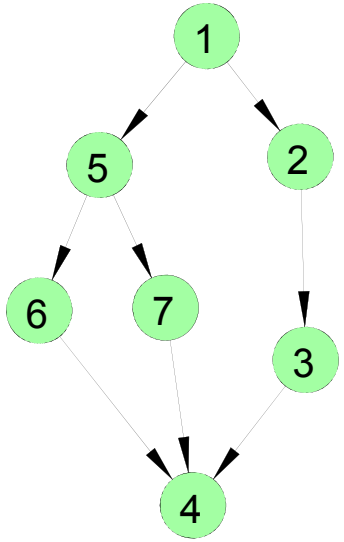
3.4 Abschätzung der Entwurfsqualität

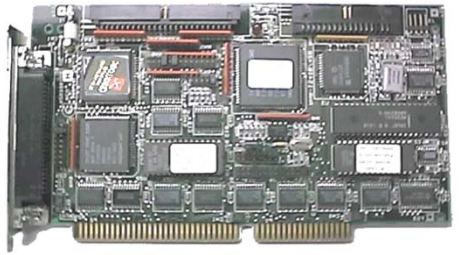
- **Motivation**: Feststellen der Systemparameter, ohne es zu implementieren
  - unterstützt Entwurfsentscheidungen
  - ermöglicht Entwurfsraumexploration
  - Ansatz für Systemoptimierungen

Schätzung
- Analyse
- Simulation
- Emulation /
- Rapid Prototyping (meist rekonf. HW)

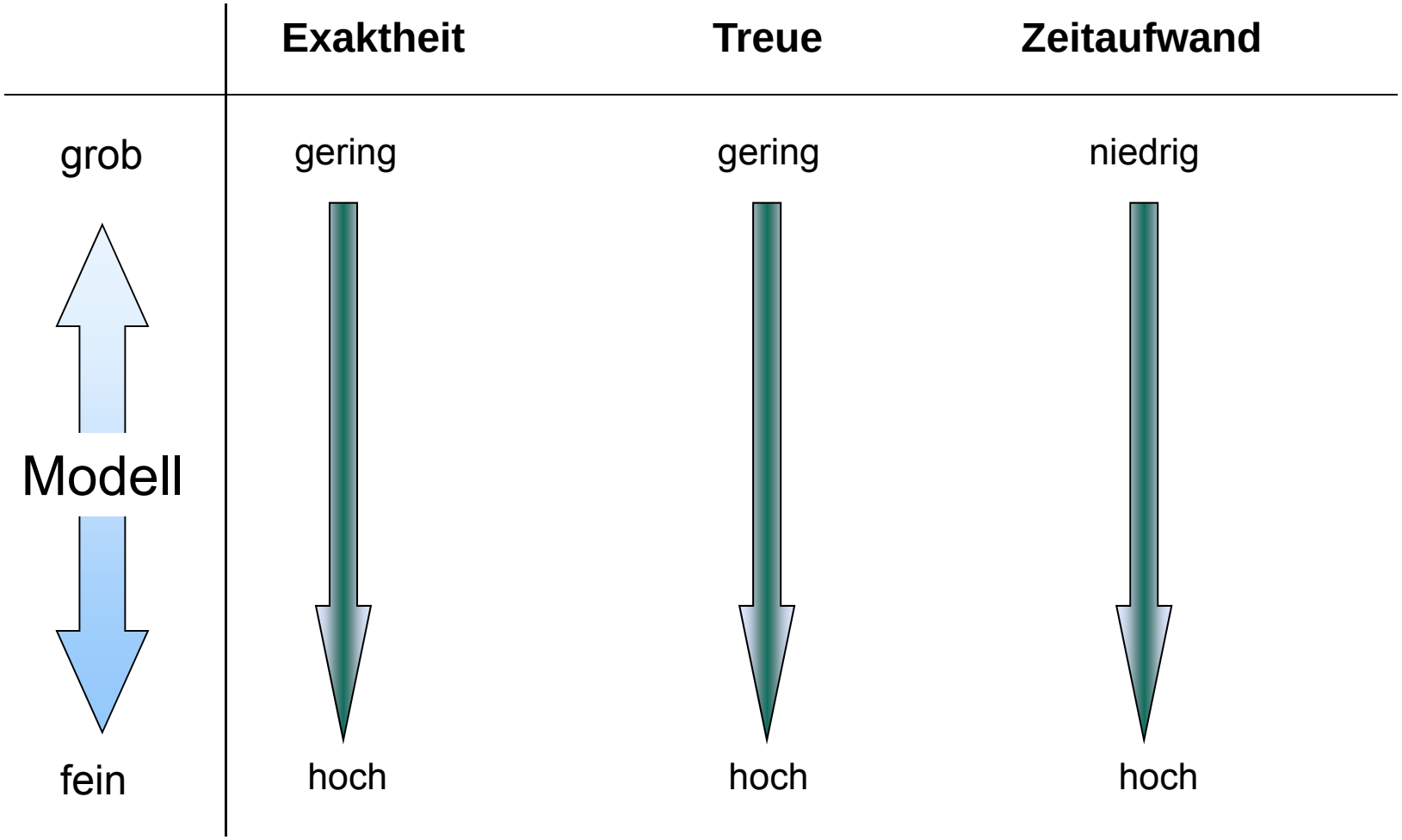
Messung
- Implementierung







3.4 Parameter von Schätzverfahren



3.4 Metriken und Exaktheit der Schätzung

- Qualitätsmaße/Metriken:
  - Performance, Kosten (Fläche), Leistungsaufnahme, Energiebedarf, Zuverlässigkeit, Testbarkeit, Time-to-market, ...

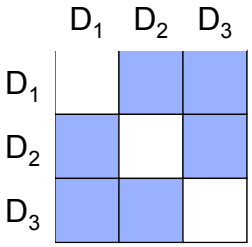
Built in Self-tests, Scan-Path Schieberegister

- Definition : Exaktheit
  - Sei E(D) eine abgeschätzte und M(D) die exakte (gemessene) Metrik einer Implementierung D.
  - Die Exaktheit A der Abschätzung ist gegeben durch :

$$A = 1 - \frac{|E(D) - M(D)|}{|M(D)|}$$

3.4 Treue der Schätzung

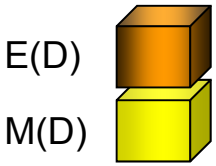
- Definition :
  - Sei D = { D1, D2 ,..., Dn } eine Menge von Implementierungen
    - Bsp. D1= ASIC, D2= FPGA
  - Die Treue F ist ein Maß für die Zuverlässigkeit einer Schätzmethode, welche die Implementierungen Di gegeneinander vergleicht:

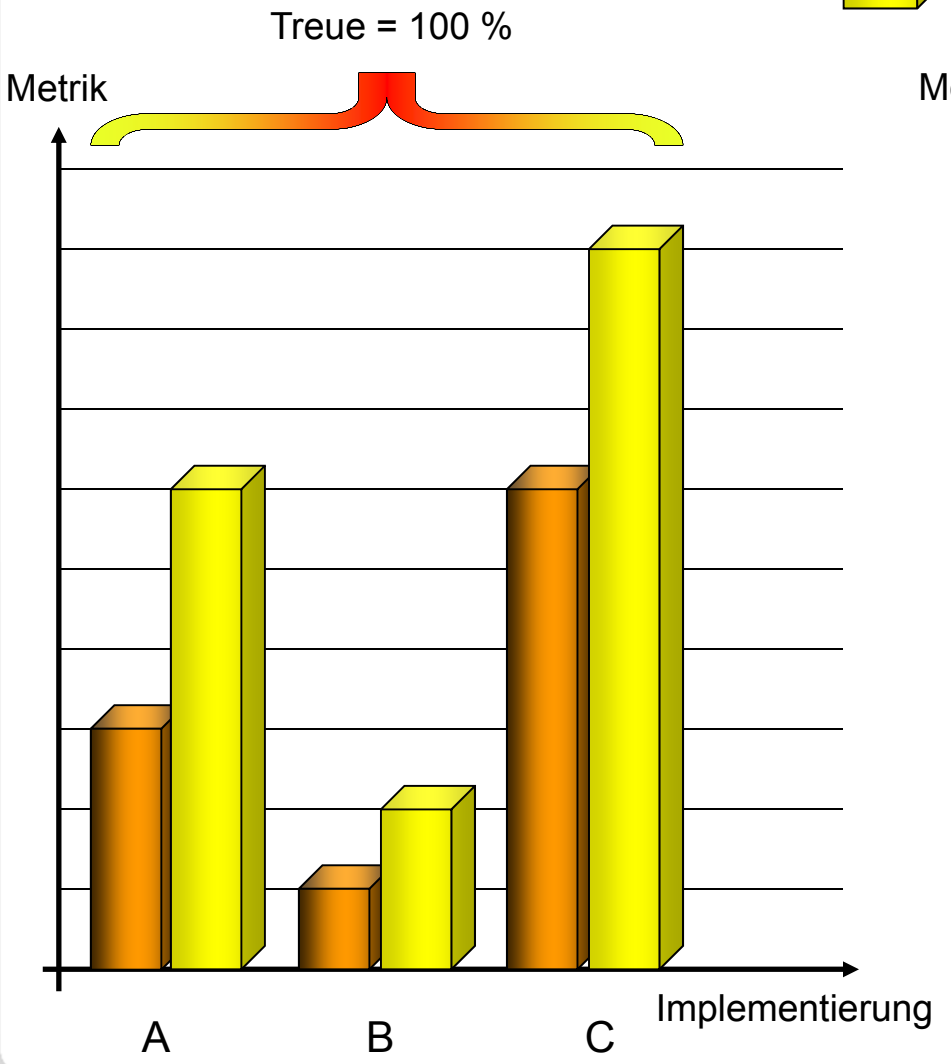


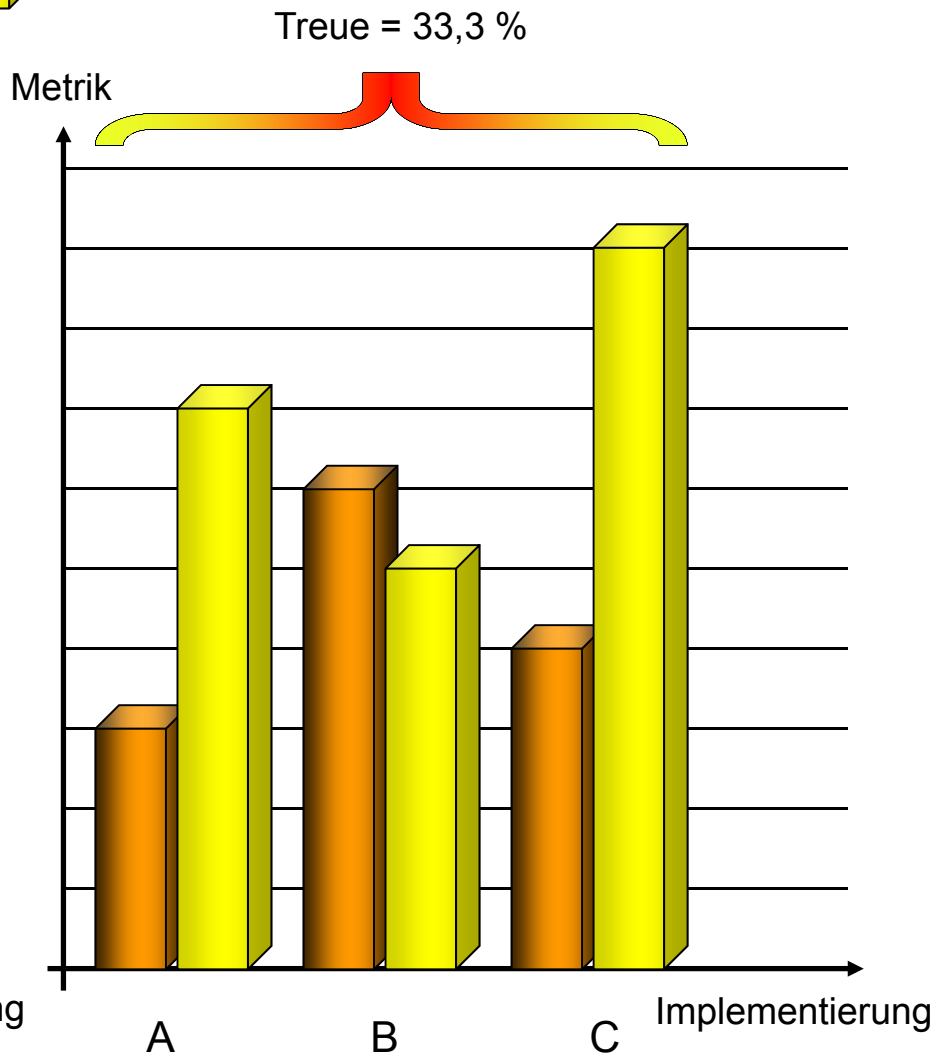
$$F = 100\% \cdot \frac{2}{n(n-1)} \cdot \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mu_{i,j}$$

$$\mu_{i,j} = \begin{cases} 1 & if \quad \left(E(D_i) > E(D_j) \wedge M(D_i) > M(D_j)\right) \vee \\ & \quad \left(E(D_i) < E(D_j) \wedge M(D_i) < M(D_j)\right) \vee \\ & \quad \left(E(D_i) = E(D_j) \wedge M(D_i) = M(D_j)\right) \\ 0 & else \end{cases}$$

3.4 Beispiel: Treue





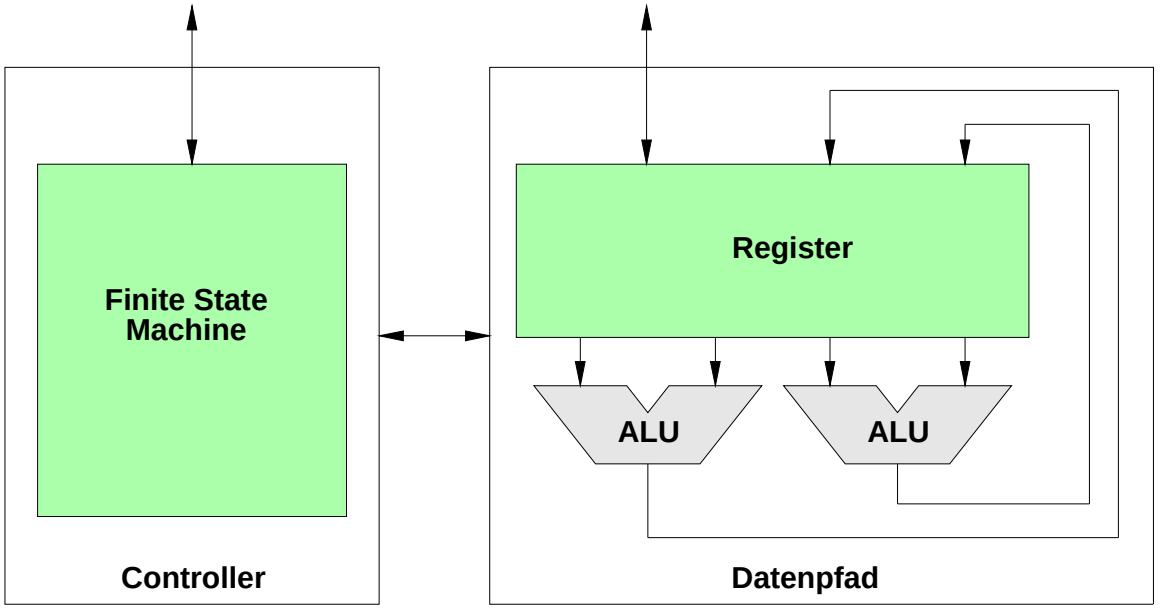


- Warum erfolgt eine Schätzung der Entwurfsqualität?
- Welche Metriken können geschätzt werden?
- Was ist Exaktheit, was Treue?
- Mehr dazu in der Vorlesung Hardware Synthese & Optimierung (HSO) von Prof. Becker im Sommersemester

Inhalt

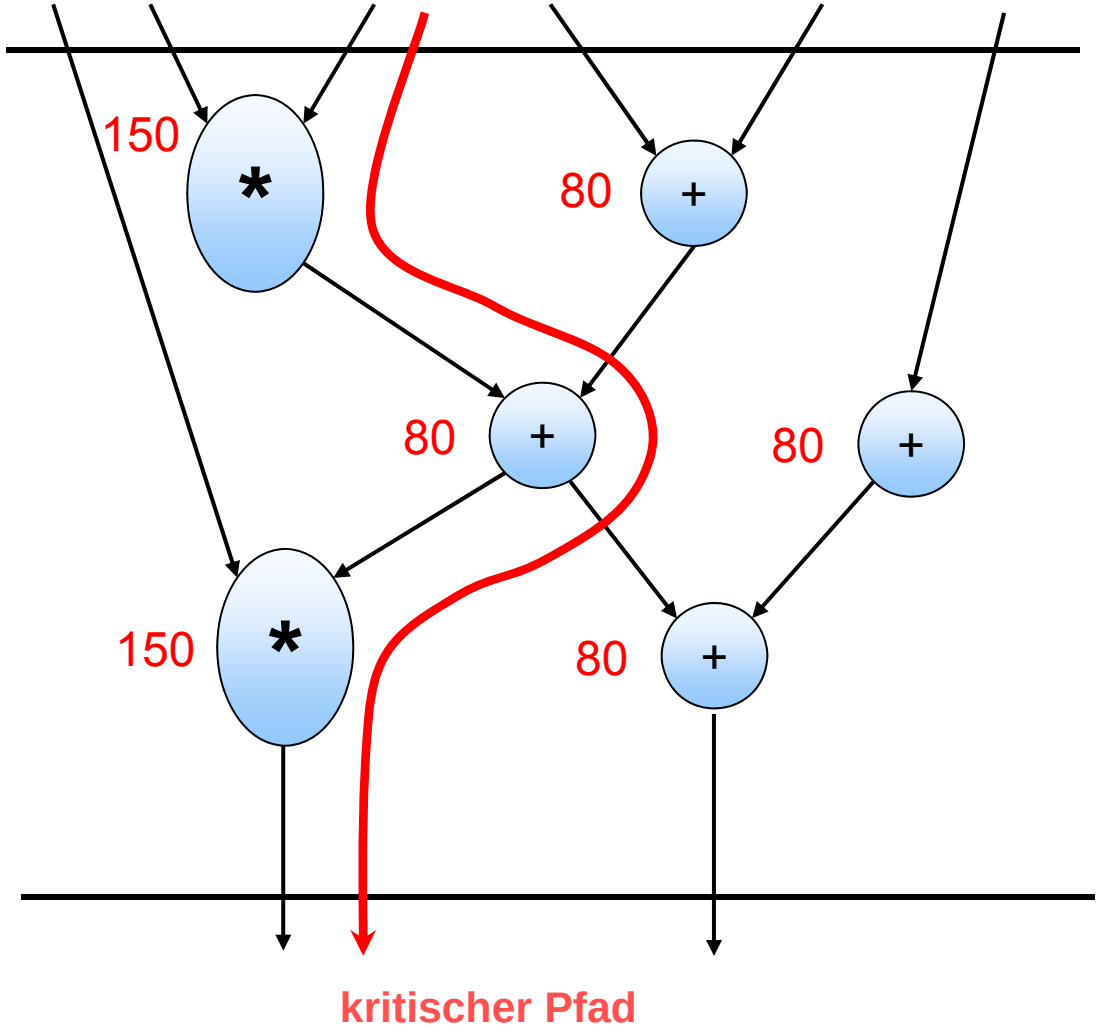
3.5 Hardware - Performanz

- Taktperiode T
  - Beeinflusst von Technologie, Ressourcen
- Latenz L
  - Anzahl der Taktschritte, abhängig von Datenpfadoptimierung (Logikminimierung, Scheduling, Retiming)



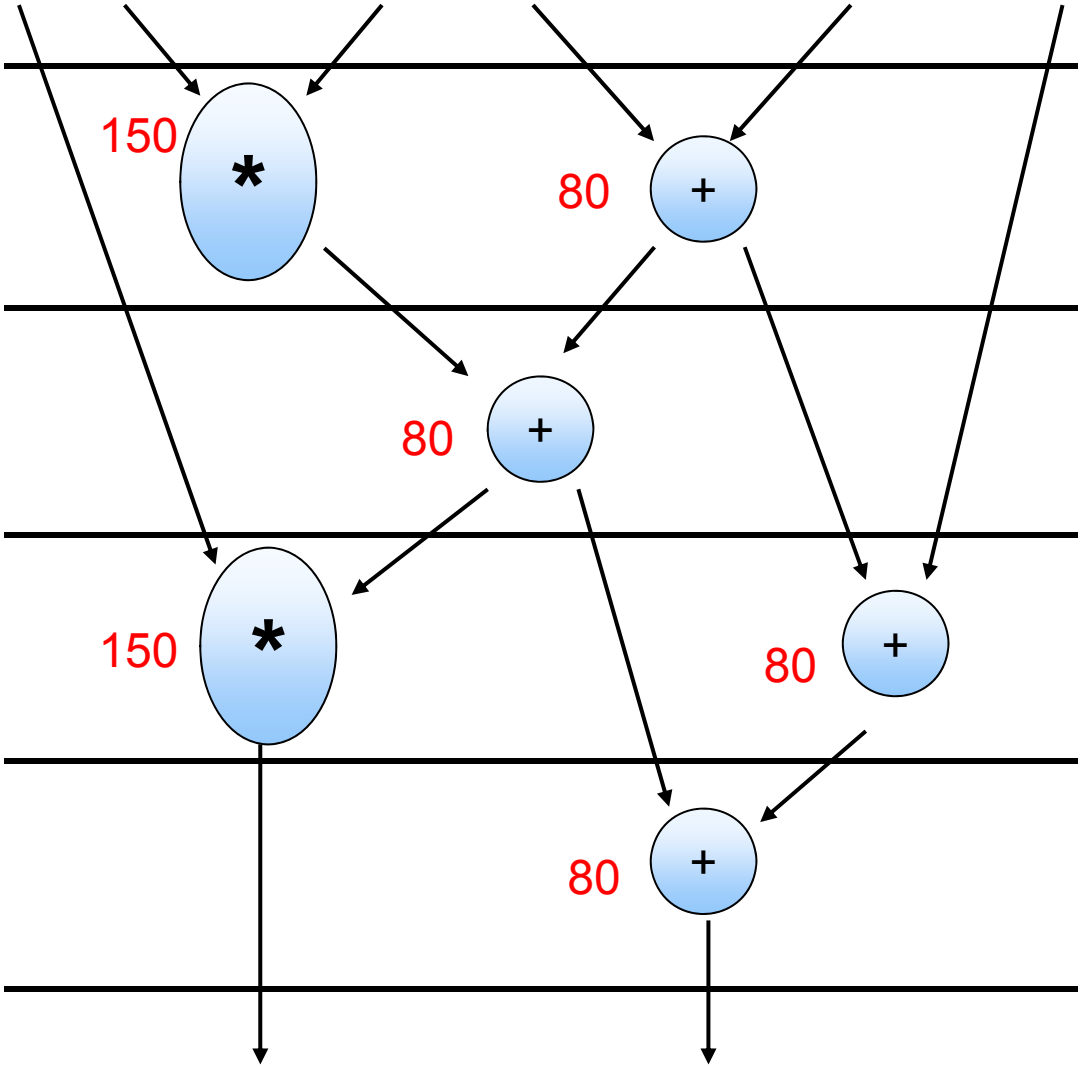
- Ausführungszeit:
  - $T_{ex} = T \cdot L$
- Durchsatz:
  - $R = 1 / T_{ex}$
    (entspr. zeitl. Häufigkeit des Berechnungsergebnisses)

3.5 Hardware-Performanz: Beispiel (I)



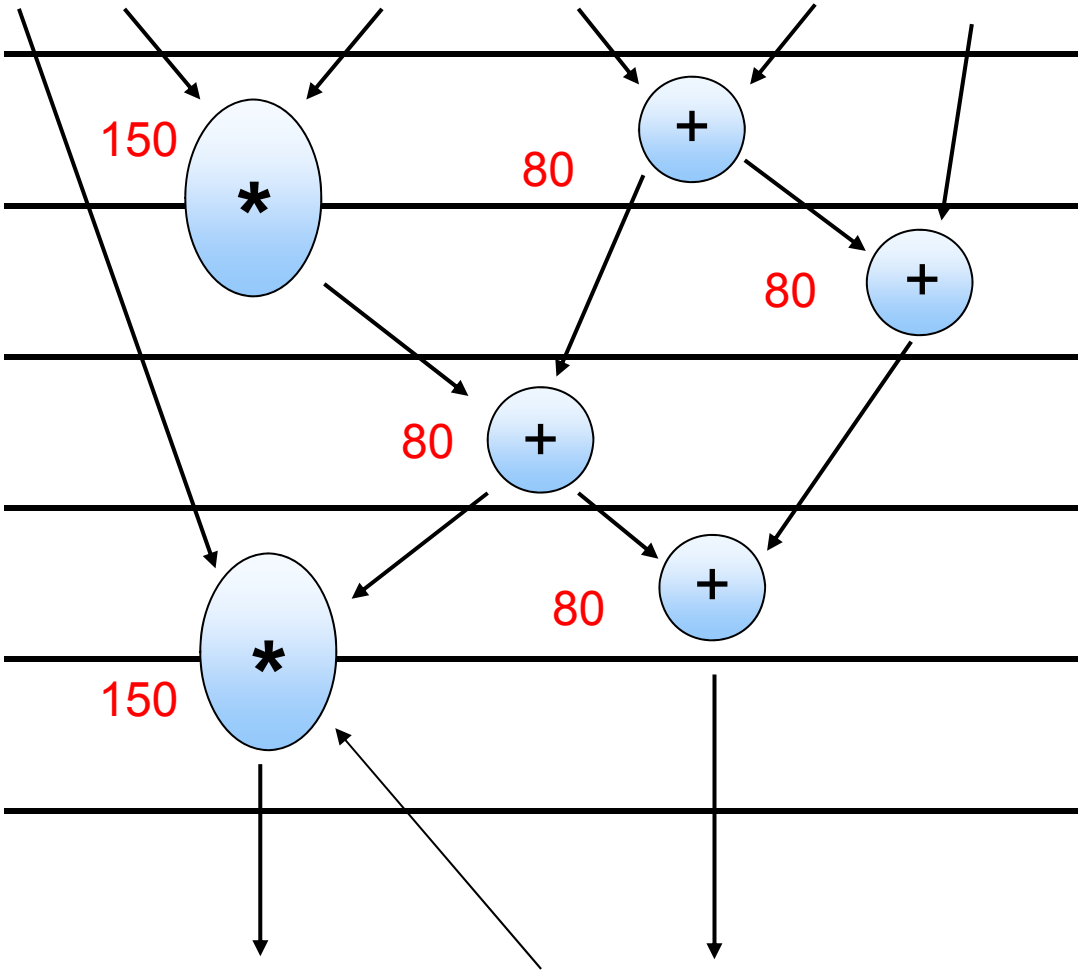
- Taktperiode T=380 ns
- Latenz L=1
- Ausführungszeit Tex= 380 ns
- Ressourcen: 2 * , 4 +

3.5 Hardware-Performanz: Beispiel (II)



- Taktperiode T=150 ns
- Latenz L=4
- Ausführungszeit Tex= 600 ns
- Ressourcen: 1 * , 1 +
- Keine Änderung der Inputs über die 4 Takte.
- Erst danach neue Inputs anlegen.

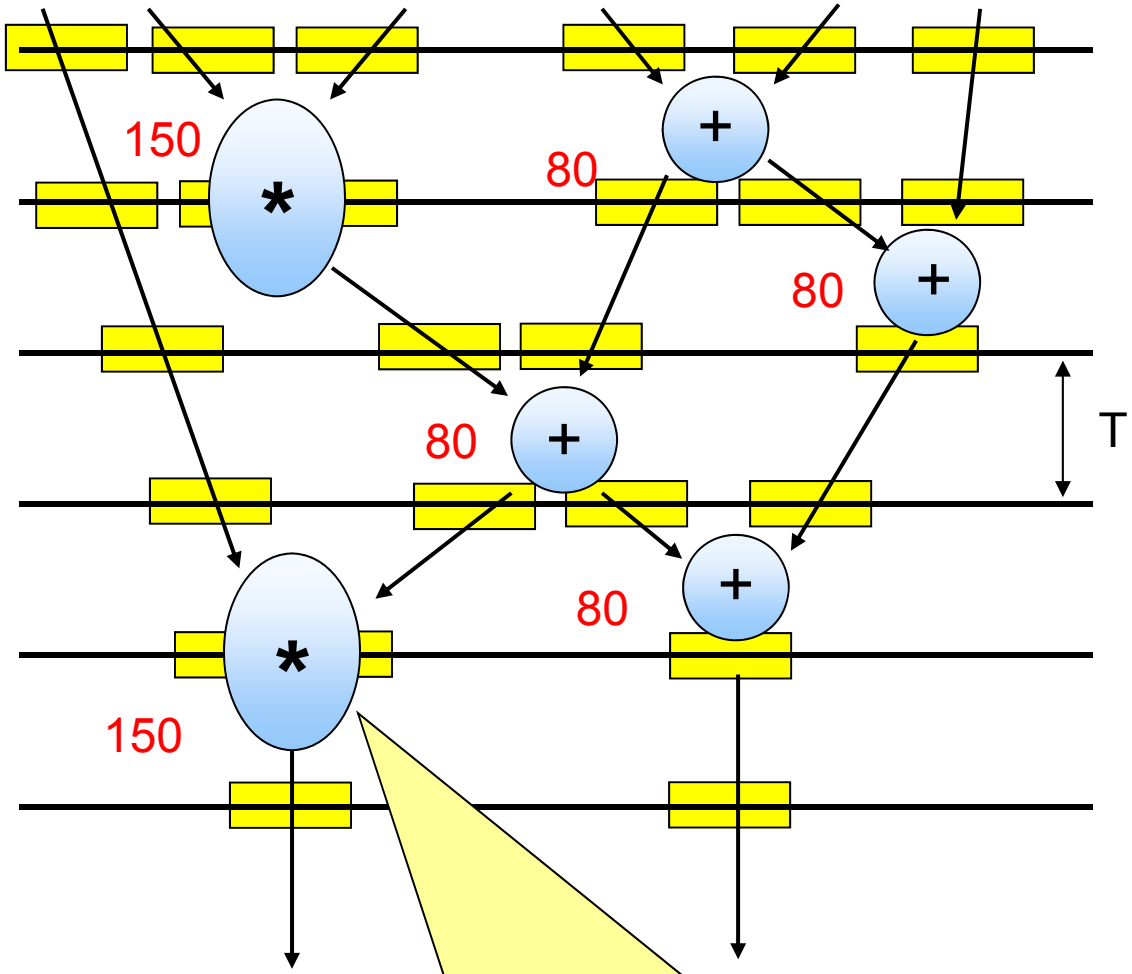
3.5 Hardware-Performanz: Beispiel (III)



Multi-cycle Operation

- Taktperiode T=80 ns
- Latenz L=5
- Ausführungszeit Tex= 400 ns
- Ressourcen: 1 * , 1 +
- **Vergleich mit Beispiel 1:**
  - Weniger Ressourcen nötig
  - Taktperiode kleiner
  - T_{ex} etwas größer

3.5 Hardware-Performanz: Pipelining



Pipelining mit P gleich langen Stufen

$R = P / T_{ex}$

T_{ex}

$R = 5 / T_{ex}$

Multiplizierer mit arithmetischer Pipeliningunterstützung

3.5 Hardware-Performanz: Schätzung der Taktperiode

- Funktionale Einheiten v_k mit delays delay(v_k)
  - Methode der maximalen Operatorverzögerung

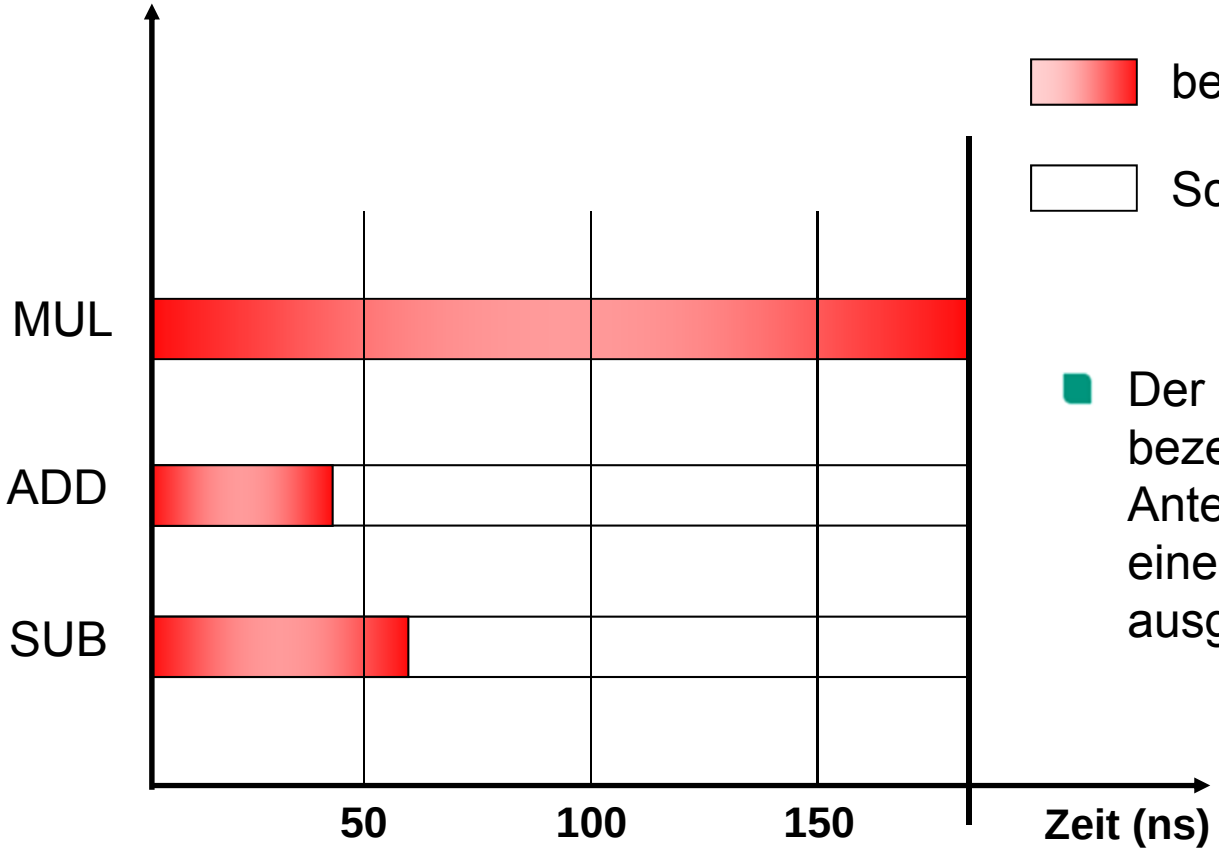
$$T = \max_k \left(delay\left(v_k \right) \right)$$

    - Nachteil: es muß mit einer erheblichen Unterauslastung der schnelleren Funktionseinheiten gerechnet werden.

  - Methode der Minimierung des *Clockschlupfs*
    - Suche im Intervall T_{min} … T_{max} nach der Taktperiode T mit maximaler Taktauslastung (minimalem *Clockschlupf*), Scheduling oft als Nachfolgeschritt zur Bestimmung der Gesamtausführungszeit T_{exec}.

  - ILP-Suche:
    - Modellierung eines Latenzminierungsproblems als ILP für diskrete Werte der Taktperiode zur Minimierung von T_{exec}.

3.5 Hardware-Performanz: *Taktschlupf*



- Der **Clockschlupf** (clock slack) bezeichnet den proportionalen Anteil einer Taktperiode, in dem eine funktionale Einheit v_k nicht ausgenutzt wird.

$$slack(T, v_k) = \left(\lceil delay(v_k)/T \rceil \right) \cdot T - delay(v_k)$$

3.5 Hardware-Performanz: Taktschlupfminimierung

- Mit occ(vk), der Anzahl der Operationen vom Typ vk, und |VT|, der Anzahl unterschiedlicher Operationstypen, ist der mittlere Schlupf für eine Taktperiode T:

$$avgslack(T) = \frac{\sum_{k=1}^{|V_T|} (occ(v_k) \cdot slack(T, v_k))}{\sum_{k=1}^{|V_T|} occ(v_k)}$$

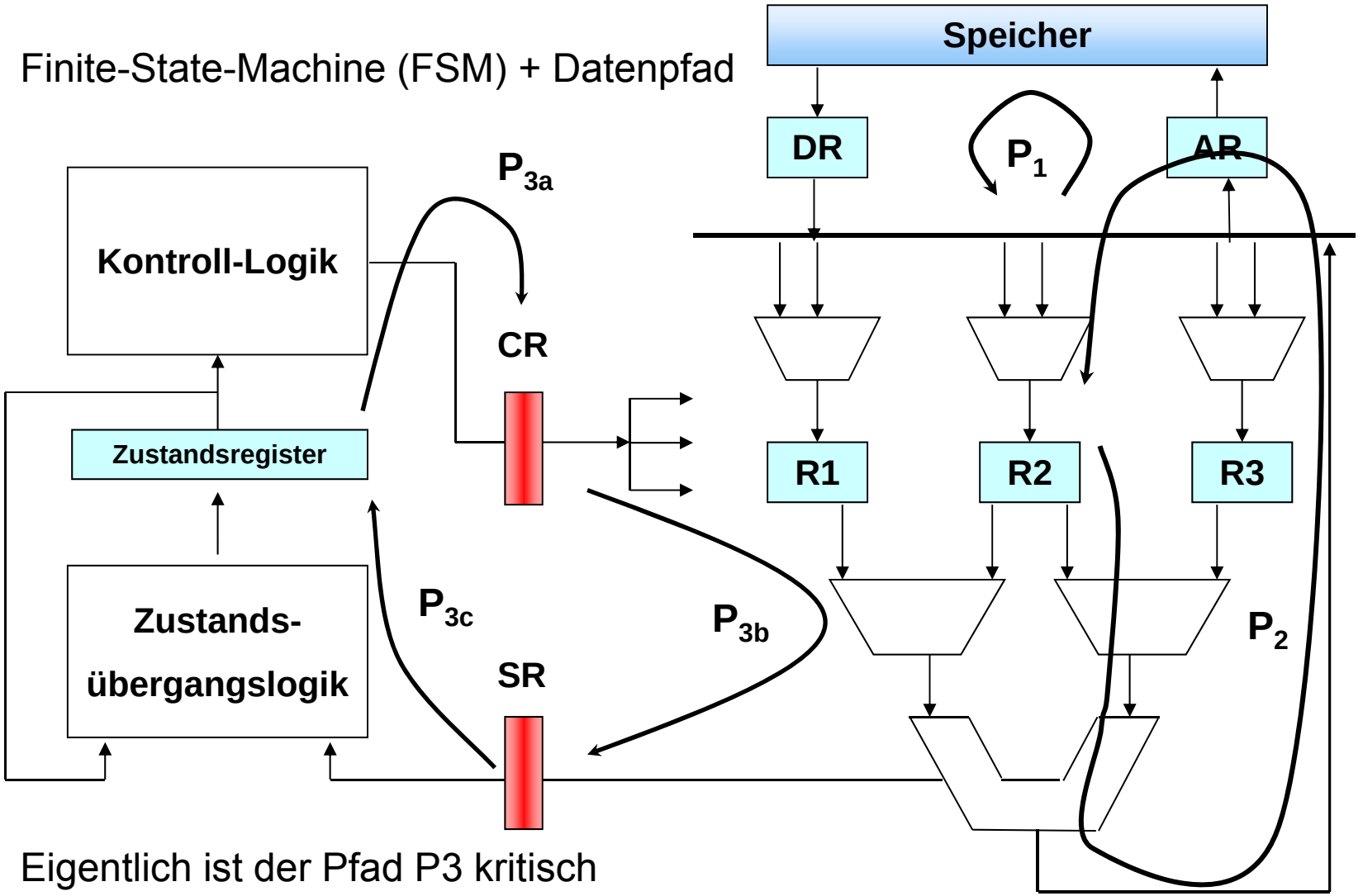
- Ein geringerer mittlerer Taktschlupf impliziert auch eine geringere Ausführungszeit für eine feste Anzahl Ressourcen.
- **Taktauslastung**:

$$util(T) = 1 - \frac{avgslack(T)}{T}$$

  - bezeichnet die prozentuale mittlere Auslastung aller Funktionseinheiten.

3.5 Hardware-Performanz: FSMD - Modell

- Finite-State-Machine (FSM) + Datenpfad



- Eigentlich ist der Pfad P3 kritisch
- Einführen zusätzlicher Register ⇒ Control Pipelining

3.5 Hardware - Kostenmetriken

- Metriken proportional zur Siliziumfläche
  - mm^2, λ^2 $\Rightarrow$ Design Rules
  - Anzahl Transistoren, Anzahl Gatter
  - Anzahl von logic blocks (FPGAs)
- Package, Anzahl von I/O-Pins
- FSMD Modell
  - Datenpfad: Register, Funktionelle Einheiten, Logik, Verdrahtung
  - Controller: Zustandsregister, Kontroll- Logik, Zustands- Übergangslogik

- Was ist Hardware-Performanz? Wie setzt sich diese zusammen?
- Wie kann Pipelining ausgenutzt werden?
- Wieso wird die Taktperiode geschätzt? Wie kann sie optimiert werden?
- Wieso wird zwischen Hardware-Leistung und Energie unterschieden?
- Mehr dazu in der Vorlesung Hardware Synthese & Optimierung (HSO) von Prof. Becker im Sommersemester

Inhalt

3.6 Software - Performanz

- Ausführungszeit T_{exec}
  - $T_{exec} = I_c \times CPI \times \tau = (I_c \times CPI) / f$
  - I_c Instruction-Count eines Programms
  - CPI Cycles Per Instruction (Mittelwert)
    (bei Pipelining + Multiskalar 0,6 $\Rightarrow$ 0,2 und kleiner)
  - τ Taktperiode = T_{clock}
  - f Taktfrequenz ($1/\tau$)
- Beispiel:
  - $I_c = 2000$, CPI = 0.4, f = 400 MHz $\Rightarrow T_{exec} = 2\ \mu s$

3.6 Software–Performanz: Beispiel (I)

- MIPS-Rate (million instructions per second)
  - MIPS = I_c / ($T_{exec} \cdot 10^6$) = f / (CPI $\cdot 10^6$)
  - **Beispiel:**
    - Prozessor mit einer Taktfrequenz von 500 MHz
    - 3 Instruktionsklassen A, B, C mit $CPI_A = 1$, $CPI_B = 2$, $CPI_C = 3$
    - 2 verschiedene Compiler erzeugen für das gleiche Programm folgende Instruktionszahlen I_c:

	A	B	C
Compiler 1	$5 \cdot 10^9$	$1 \cdot 10^9$	$1 \cdot 10^9$
Compiler 2	$10 \cdot 10^9$	$1 \cdot 10^9$	$1 \cdot 10^9$

3.6 Software–Performanz: Beispiel (II)

- **Taktzyklen: L**
  - Programm1: $(5 \cdot 1 + 1 \cdot 2 + 1 \cdot 3) \cdot 10^9 = 10 \cdot 10^9$
  - Programm2: $(10 \cdot 1 + 1 \cdot 2 + 1 \cdot 3) \cdot 10^9 = 15 \cdot 10^9$
- **Ausführungszeiten: L/f**
  - Programm 1: $(10 \cdot 10^9) / (500 \cdot 10^6) = 20$ sec
  - Programm 2: $(15 \cdot 10^9) / (500 \cdot 10^6) = 30$ sec
- **MIPS-Raten: ΣI_c / ($T_{exec} \cdot 10^6$)**
  - Programm1: $((5 + 1 + 1) \cdot 10^9) / (20 \cdot 10^6) = 350$ MIPS
  - Programm2: $((10 + 1 + 1) \cdot 10^9) / (30 \cdot 10^6) = 400$ MIPS
- Programm 2 hat eine **höhere MIPS-Rate,** Programm1 **läuft schneller**, da weniger Code.
- MIPS eher als Maß für die Komplexität eines Algorithmus, bzw. dessen Umsetzung.
- Nur T_{exec} ist **wirklich aussagefähig**.

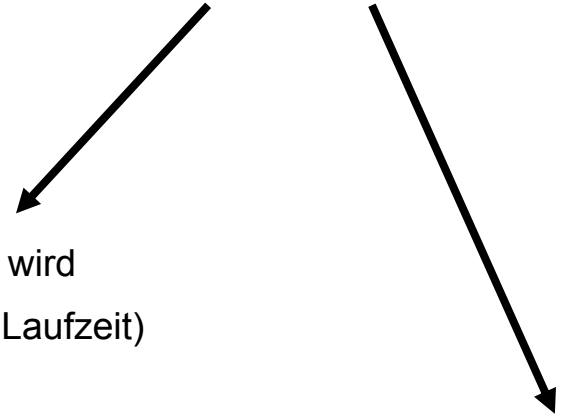
3.6 Software-Performanz: Metriken

- **MIPS** (million instructions per second)
- **MFLOPS** (million floating-point operations per second)
- **MACS** (million multiply & accumulates per second)
  - wichtig bei DSPs
- **MOPS** (million operations per second)
  - alle Operationen zusammengezählt: ALUs, Adressrechnungen, DMA, …
  - bei allen:
    - Paralleloperationen berücksichtigt
    - optimale Belegung vorausgesetzt

- **Ausführungszeit**
  - Profiling: Compilation und möglichst viele Testläufe
    ⇒ statistische Aussagen
  - Analytische Schätzung: auf Basis des Quell- / Zwischen- / Zielcodes
    ⇒ wichtig bei Echtzeitsystemen mit harten Zeitschranken:
    ⇒ **worst-case execution time (WCET)**

3.6 Software-Performanz: Worst-Case Execution Time (WCET)

- kann nicht durch *Profiling* bestimmt werden
- Schätzung mittels **Programmanalysetechniken**



Programmpfadanalyse

Welche Reihenfolge von Instruktionen wird im worst-case ausgeführt? (längste Laufzeit)

Problem: die Anzahl der möglichen Programmpfade wächst exponentiell mit der Programmlänge

Modellierung der Zielarchitektur

Berechnung der geschätzten WCET für ein spezifisches Prozessormodell

Probleme: Compileroptimierungen, dynamische Effekte durch Pipelining, Caches

Vereinfachte Annahmen nötig.

3.6 Software-Performanz: Programmpfadanalyse

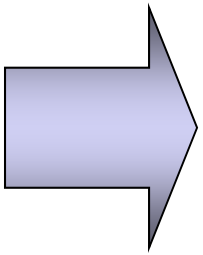
- Die geschätzte WCET
  - ist immer größer als die tatsächliche WCET,
  - eine gute Schätzung approximiert die tatsächliche WCET aber möglichst nahe.

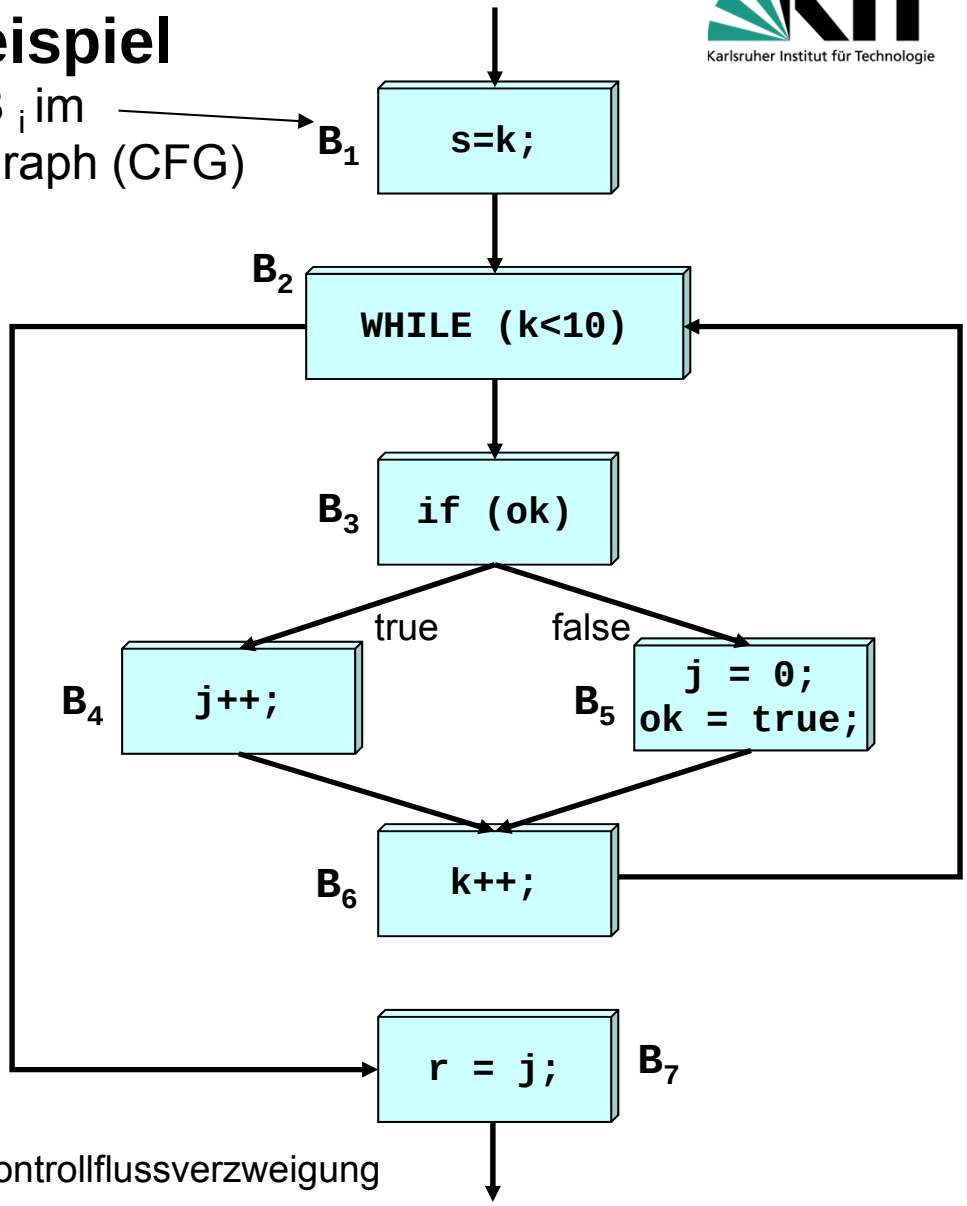
- Prozessormodell
  - ein Prozessor (eine skalare Einheit)
  - keine Interrupts
  - kein Betriebssystem (keine Preemption)

- Programmiermodell
  - keine rekursiven Funktionsaufrufe (direkt und indirekt)
  - keine Pointeroperationen
  - Schleifen müssen beschränkt sein

3.6 Software-Performanz: Programmpfadanalyse - Beispiel

```
/* k >= 0 */
s = k;
WHILE (k < 10) {
    IF (ok)
        j++;
    ELSE {
        j = 0;
        ok = true;
    }
    k ++;
}
r = j;
```





Basisblock B $_i$: Sequenz von Instruktionen bis Kontrollflussverzweigung

3.6 Software-Performanz: Strukturelle Beschränkungen

- **Flussgleichungen**:

$d_1 = d_2 = x_1$

$d_2 + d_8 = d_3 + d_9 = x_2$

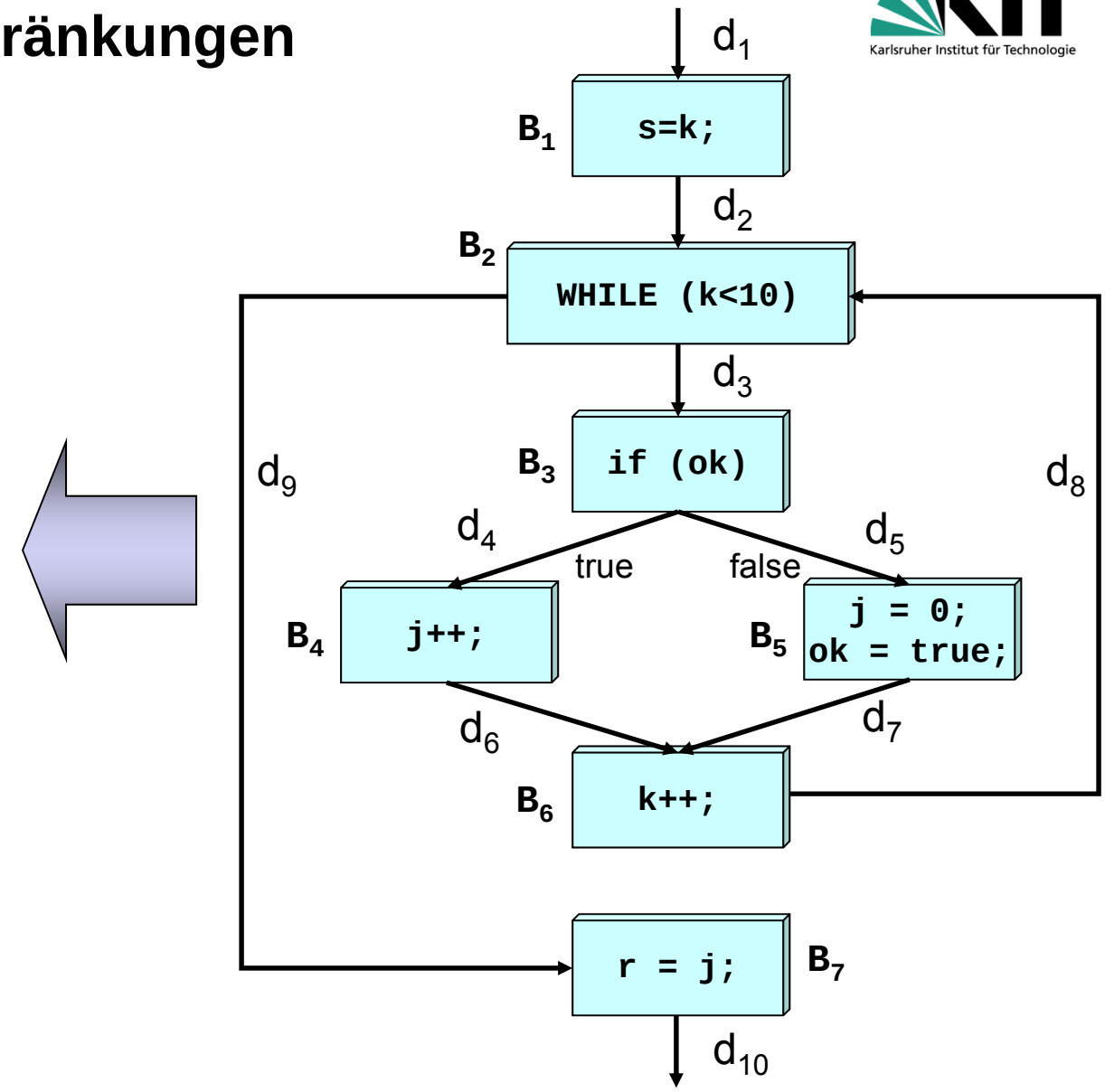
$d_3 = d_4 + d_5 = x_3$

$d_4 = d_6 = x_4$

$d_5 = d_7 = x_5$

$d_6 + d_7 = d_8 = x_6$

$d_9 = d_{10} = x_7$



3.6 Software-Performanz: Berechnung der WCET

- **Definition:**
  - Ein Programm besteht aus N Grundblöcken, wobei jeder Grundblock B_i eine Worst-Case Ausführungszeit c_i hat und genau x_i mal ausgeführt wird. Dann ist die

$$WCET = \sum_{i=1}^{N} c_i \cdot x_i$$

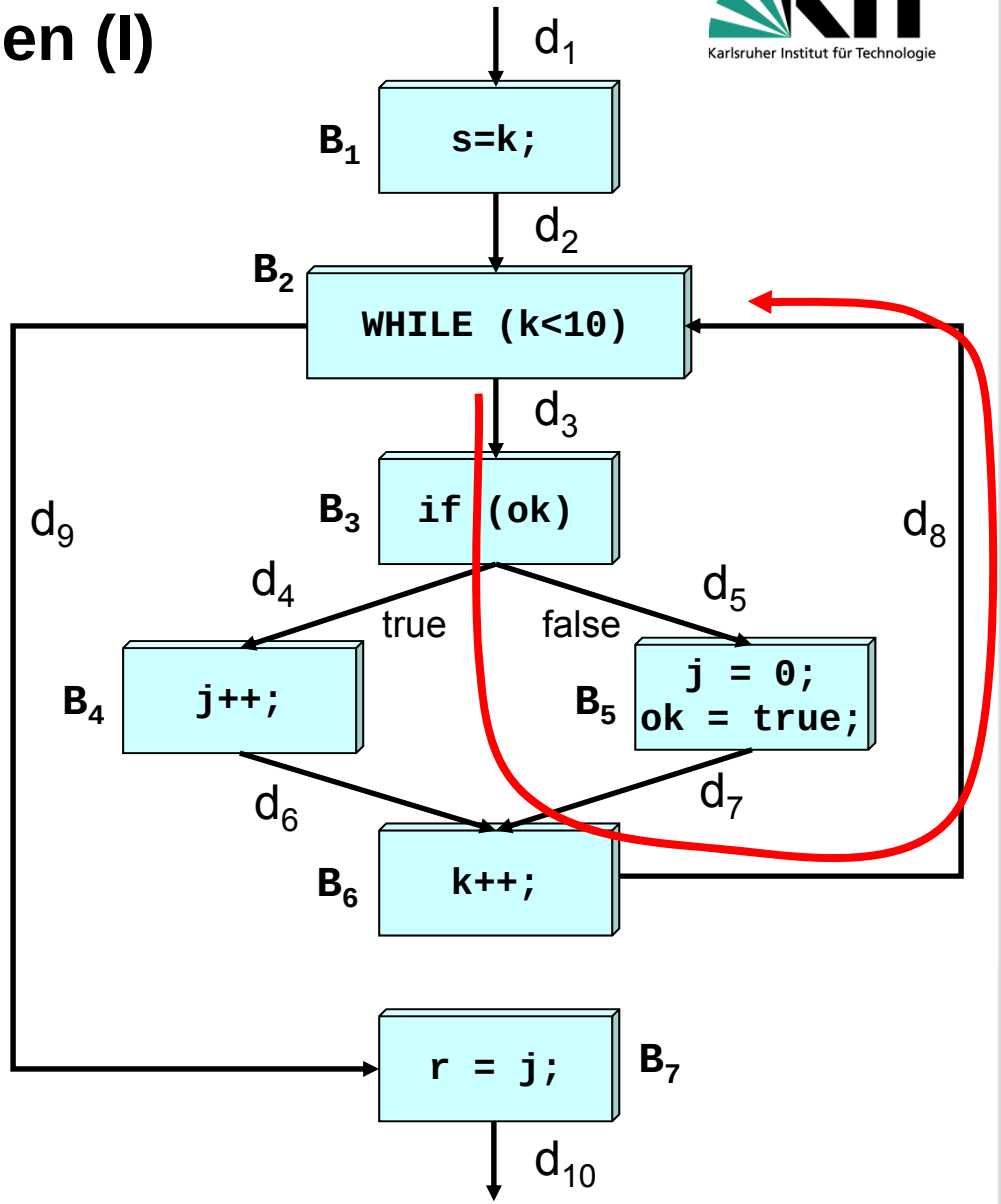
  - Die c_i können abgeschätzt werden, da die Sequenz der Instruktionen bekannt ist (Grundblock-Definition).
- Wie berechnet man die xi ?
  - strukturelle Constraints durch Programmstruktur gegeben
  - funktionale Constraints durch Programmierer gegeben (Schleifengrenzen, etc.)
- **Anmerkung:** alles auf vom Compiler erzeugten Assemblercode bezogen; Source Code aber einfacher nachzuvollziehen

3.6 Software-Performanz: Funktionale Beschränkungen (I)

- Die **While-Schleife** wird maximal **10 mal** durchlaufen:
- $0 \cdot x_1 \leq x_3 \leq 10 \cdot x_1$

- **B_5** wird **maximal einmal** durchlaufen:

- $x_5 \leq 1 \cdot x_1$



3.6 Software-Performanz: Funktionale Beschränkungen (II)

- Werden durch den Programmierer definiert
  - durch Schranken für Schleifenzähler
  - durch Kenntnis des Programmkontextes
- Können komplex sein
  - **Beispiel (angenommen!):**

    “Wird der ELSE-Zweig in der Schleife ausgeführt, so wird die Schleife genau 5 mal durchlaufen.”

Spaltet die funktionalen Constraints in zwei Sets ⇒ **getrennte ILPs ausführen**.

$$(x_5 = 0) \;||\; (x_5 \geq 1)\ \&\ (x_3 = 5 \cdot x_1)$$

set #1 *set* #2

3.6 Worst-Case Execution Time (WCET) – ILP Formulierung

- **ILP mit strukturellen und funktionalen Beschränkungen:**

$$WCET = \max\left\{\sum_{i=1}^{N} c_i \cdot x_i\right\}$$

- **Nebenbedingungen:**

einmaliger Programmaufruf

$$(d_1 = 1) \wedge$$

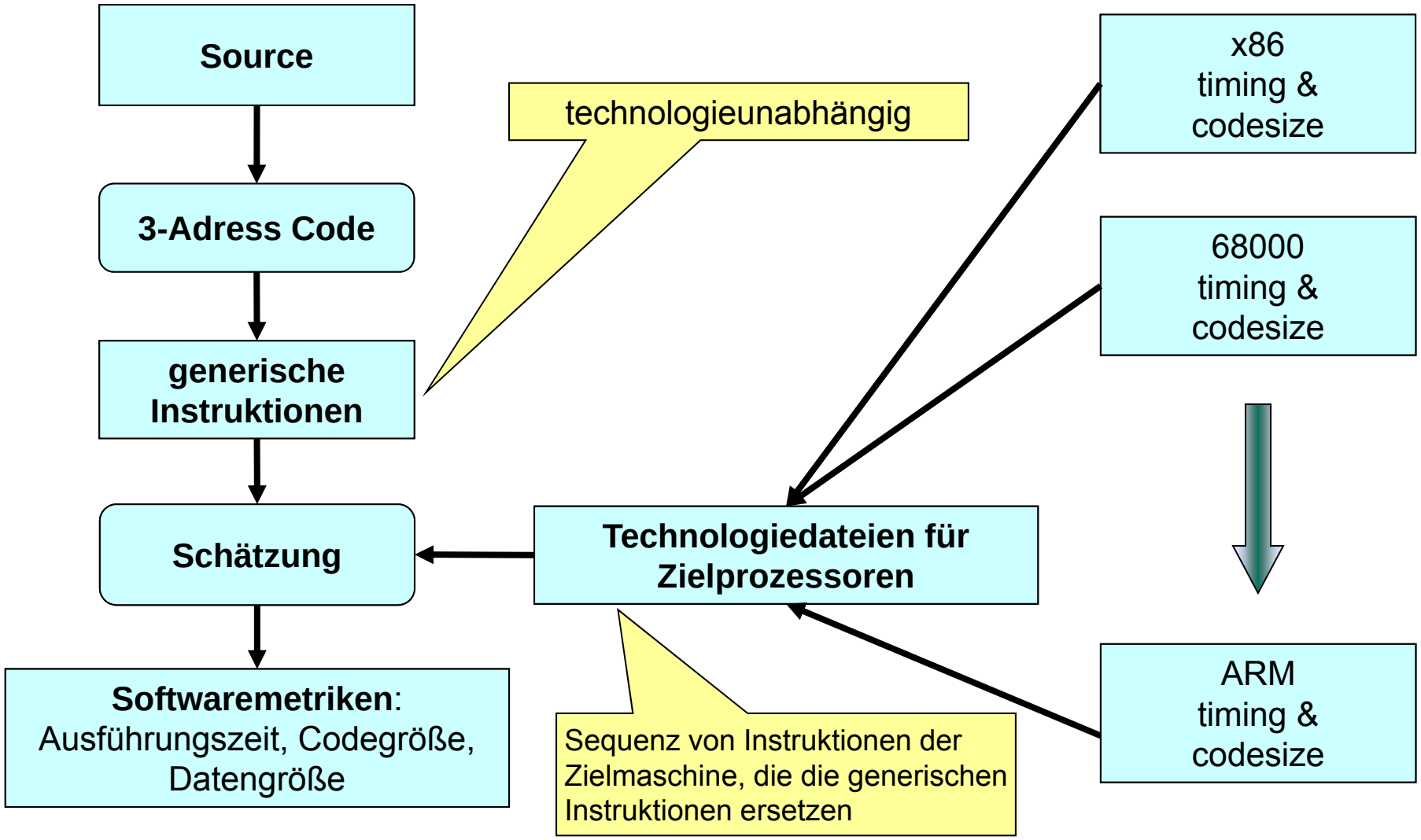
strukturelle Constraints

$$\left(\sum_{j \in Inputs(B_i)} d_j = \sum_{k \in Outputs(B_i)} d_k = x_i \quad , i = 1 \ldots N \right) \wedge$$

ein ILP pro Set von Constraints

$$(\text{funktionale Constraints})$$

3.6 Software-Performanz: Modellierung der Zielarchitektur



3.6 Software-Performanz: Modellierung von Caches (I)

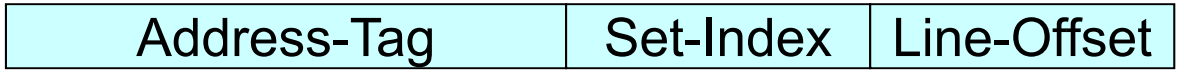
- Lokalitätseigenschaft des Programmcodes:
  - 90% der Zeit verbringt ein Programm in 10% des Codes.
- Hauptspeicherzugriffe kosten i.d.R. sehr viel Zeit.
  - Entwicklung schnellerer (und billigerer) Speichermodule (DDR, RAMBUS)
  - Einführung von kleineren und sehr schnellen SRAM- Blöcken für den Schnellzugriff $\Rightarrow$ **Caches**.
  - Anordnung der Caches in Zugriffs- Hierarchien (1st Level, 2nd Level Cache).
  - Von-Neumann-Bottleneck: kein gleichzeitiger Zugriff auf Befehle und Daten
  - Trennung von Befehls- und Datenspeicher
    - Übergang „Von Neumann“ $\Rightarrow$ Harvard Architektur
    - **Instruktions-Cache** (1st Level), i.d.R. Nur-Lese- Betrieb.
    - **Daten- Cache** (1st Level) i.d.R. Schreib/Lese- Betrieb.
      - Gut für Pipelinebetrieb, erlaubt simultanes Lesen von Instruktionen und Daten.
    - **Unified- Cache** (2nd Level) für Instruktionen und Daten.

3.6 Software-Performanz: Modellierung von Caches (II)

- Caches sind so organisiert, daß der Speicherzugriff auf seine Inhalte sehr schnell ausgeführt werden kann.
- **Cache Typen:**
  - **Direkt abbildender Cache**:
    - Speicher als eine Anzahl von Cache- Sets organisiert,
    - Jeder Set enthält nur eine Cache- Zeile.
  - **N- Wege assoziativer Cache**:
    - Jeder Set enthält N Cache- Zeilen.
    - Entspricht ersten Fall für N=1.
    - Sind teurer, haben aber eine bessere Hit-Rate.
- Die Größe einer Cache- Zeile ist fest.
  - Typische Zeilengrößen reichen von 4 bis 128 Bytes.
  - *Größe des Cache = # Cache-Sets • N • Zeilengröße*
- Cache Hitrate: 100% ·(# CacheHits / # Gesamtzugriffe)

3.6 Software-Performanz: Modellierung von Caches (III)

- Zuordnung eines Speicherblocks zur Cache-Zeile erfolgt über dessen Adresse
  - Anzahl der Cache-Sets und Zeilengröße sind immer Zweierpotenzen.
  - Zuordnung möglich durch Betrachtung der niederen Bits der Adresse (Set-Index).



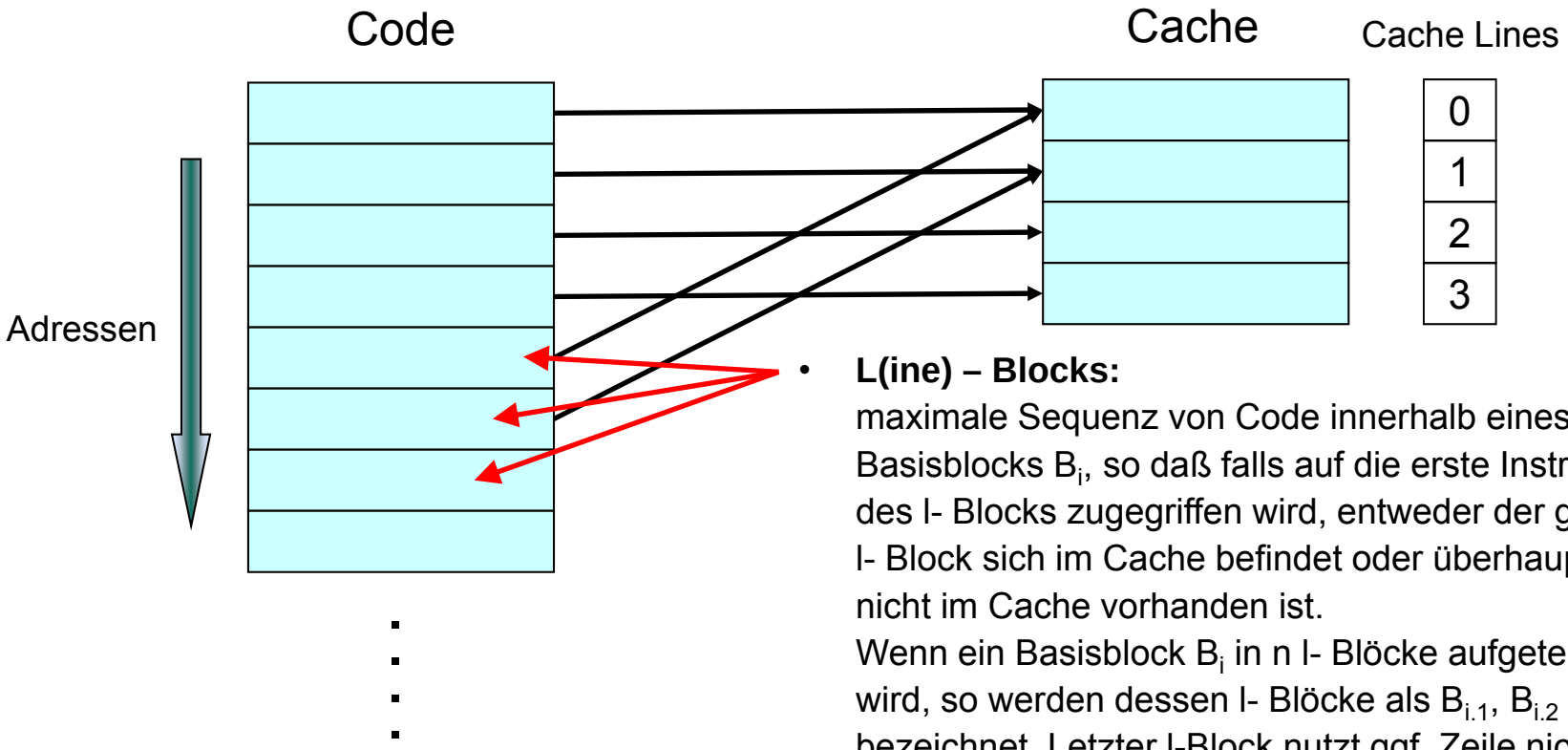
Set-Index	Valid	Adress-Tag	Cache-Inhalte	Cache-Lines
0	1			0
	1			1
1	0			2
	1			3
2	0			4
	0			5
3	1			6
	1			7

2^m-1 … 0

- Cachezugriff ist erfolgreich (Cache-Hit), wenn Set-Index in der Adresse mit dem im Cache-Eintrag übereinstimmt und der dazugehörige Adress-Tag gefunden wurde.
- Ansonsten Cache-Miss mit Zugriff auf niedrigere Speicherebene.
- Ersetzung alter Cache-Einträge durch neue Speicherblöcke mittels Least-Recently-Used (LRU)oder Variante (LFU, Random) .
  (Nur bei N-Wege Cache!)

3.6 Software-Performanz: Modellierung von Caches (IV)

- **Voraussetzung:** Compilation in die Zielsprache
- Direct-mapped Instruction-Caches (N=1)



- **L(ine) – Blocks:**
  maximale Sequenz von Code innerhalb eines Basisblocks B_i, so daß falls auf die erste Instruktion des l- Blocks zugegriffen wird, entweder der ganze l- Block sich im Cache befindet oder überhaupt nicht im Cache vorhanden ist.
  Wenn ein Basisblock B_i in n l- Blöcke aufgeteilt wird, so werden dessen l- Blöcke als $B_{i.1}$, $B_{i.2}$ usw. bezeichnet. Letzter l-Block nutzt ggf. Zeile nicht ganz $\Rightarrow$ Überlappungen mit anderen B_j möglich.

3.6 Software-Performanz: Erweiterung WCET ILP-Modell

- jeder Grundblock B_i besteht aus n_i I-Blöcken, d.h. jeder $B_{i,j}$ hat das gleiche x_i
- jede Ausführung eines I-Blocks führt entweder zu einem Cache-Hit oder einem Cache-Miss

$$x_i = x_{i,j} = x_{i,j}^{hit} + x_{i,j}^{miss} \quad , j = 1 \ldots n_i$$

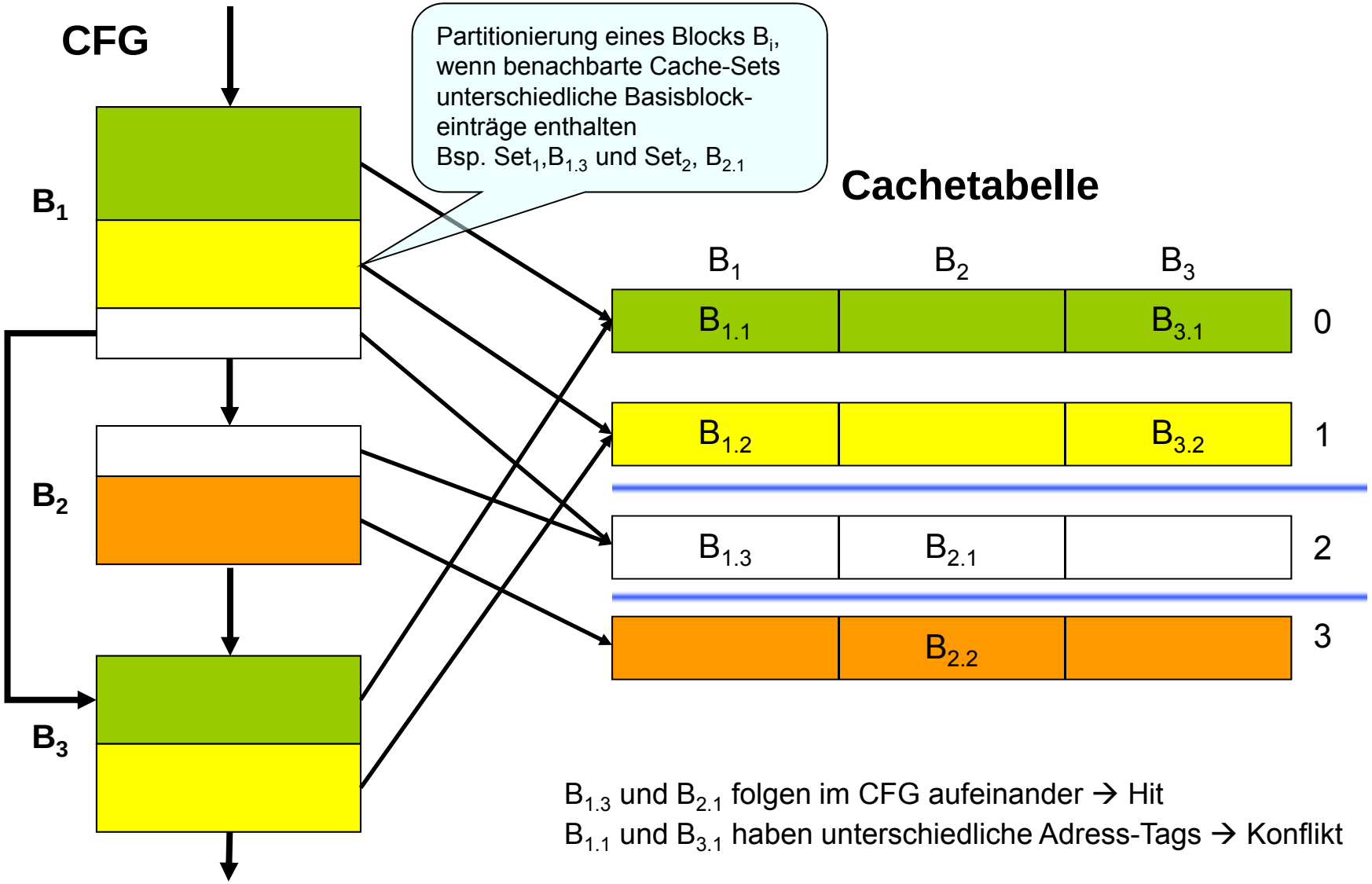
- **WCET**

Kosten für einen Cache- Hit

Kosten für einen Cache- Miss

$$WCET = \sum_{i=1}^{N} \sum_{j=1}^{n_i} \left(c_{i,j}^{hit} \cdot x_{i,j}^{hit} + c_{i,j}^{miss} \cdot x_{i,j}^{miss} \right)$$

3.6 Software-Performanz: Beispiel



$B_{1.3}$ und $B_{2.1}$ folgen im CFG aufeinander → Hit
$B_{1.1}$ und $B_{3.1}$ haben unterschiedliche Adress-Tags → Konflikt

3.6 Software-Performanz: Constraints durch Cachekonflikte

Karlsruher Institut für Technologie

- **Definition:**
  - Zwei l-Blöcke sind in Konflikt, wenn die Ausführung eines l-Blocks zu einer Ersetzung des anderen l-Blocks im Cache führt.
    - $B_{2.2}$ ist mit keinem anderen l-Block in Konflikt (höchstens 1 Cachemiss).

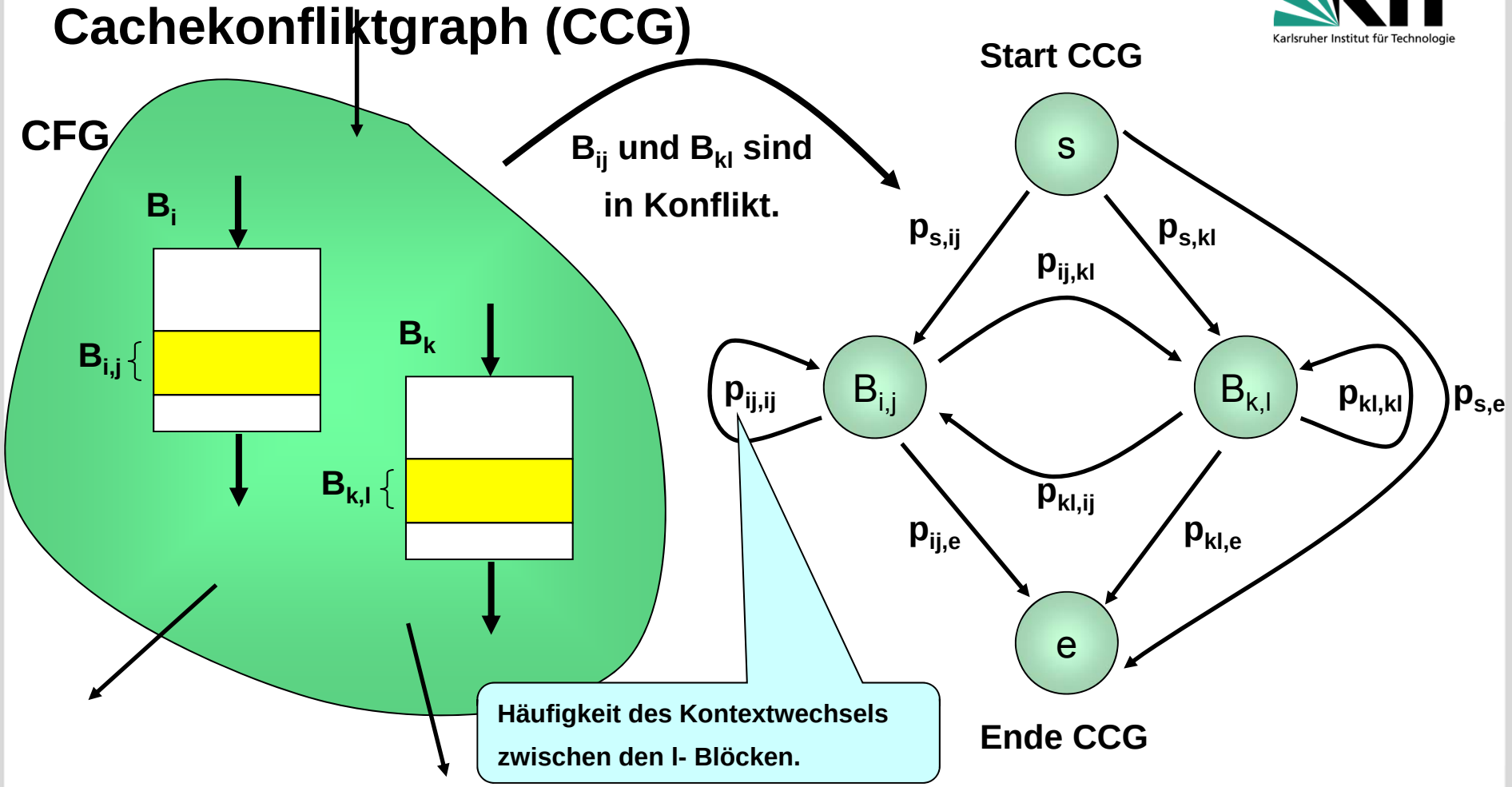
$$x_{2,2}^{miss} \leq 1$$

    - $B_{1.3}$ und $B_{2.1}$ sind nicht in Konflikt: ein Cachemiss bei der Ausführung eines der beiden Blöcke lädt automatisch den anderen Block mit in den Cache.

$$x_{1,3}^{miss} + x_{2,1}^{miss} \leq 1$$

    - $B_{1.1}$ und $B_{3.1}$ sind in Konflikt: ILP-Beschränkungen sind aus dem **Cachekonfliktgraph** ableitbar.

3.6 Software-Performanz: Cachekonfliktgraph (CCG)



- CCG nicht für Multitasking Systeme. Bilde CCG stets für l-Block-Paare, die im Konflikt sind.
- Die Kanten pij,kl bzw. pkl,ij entsprechen einem Kontrollflusspfad von Bi nach Bk bzw. Bk nach Bi im CFG, wobei keiner der auf diesen Kontrollflusspfaden liegenden anderen l-Blöcke (Bi,j , Bk,l ausgenommen) auch auf die gleiche Cache-Zeile wie Bi,j , Bk,l abgebildet wird.

3.6 Software-Performanz: Constraints aus Cachekonfliktgraph

- für alle Knoten (außer s, e): **Flußgleichungen**:

$$x_i = x_{i,j} = p_{s,ij} + p_{kl,ij} + p_{ij,ij} = p_{ij,kl} + p_{ij,e} + p_{ij,ij}$$

- Cache-Hits:

$$x^{hit}_{i,j} = p_{ij,ij} \qquad x^{hit}_{k,l} = p_{kl,kl}$$

- Cache-Misses:

Erster Aufruf von $B_{i,j}$

$$x^{miss}_{i,j} = p_{s,ij} + p_{kl,ij}$$

$$x^{miss}_{k,l} = p_{s,kl} + p_{ij,kl}$$

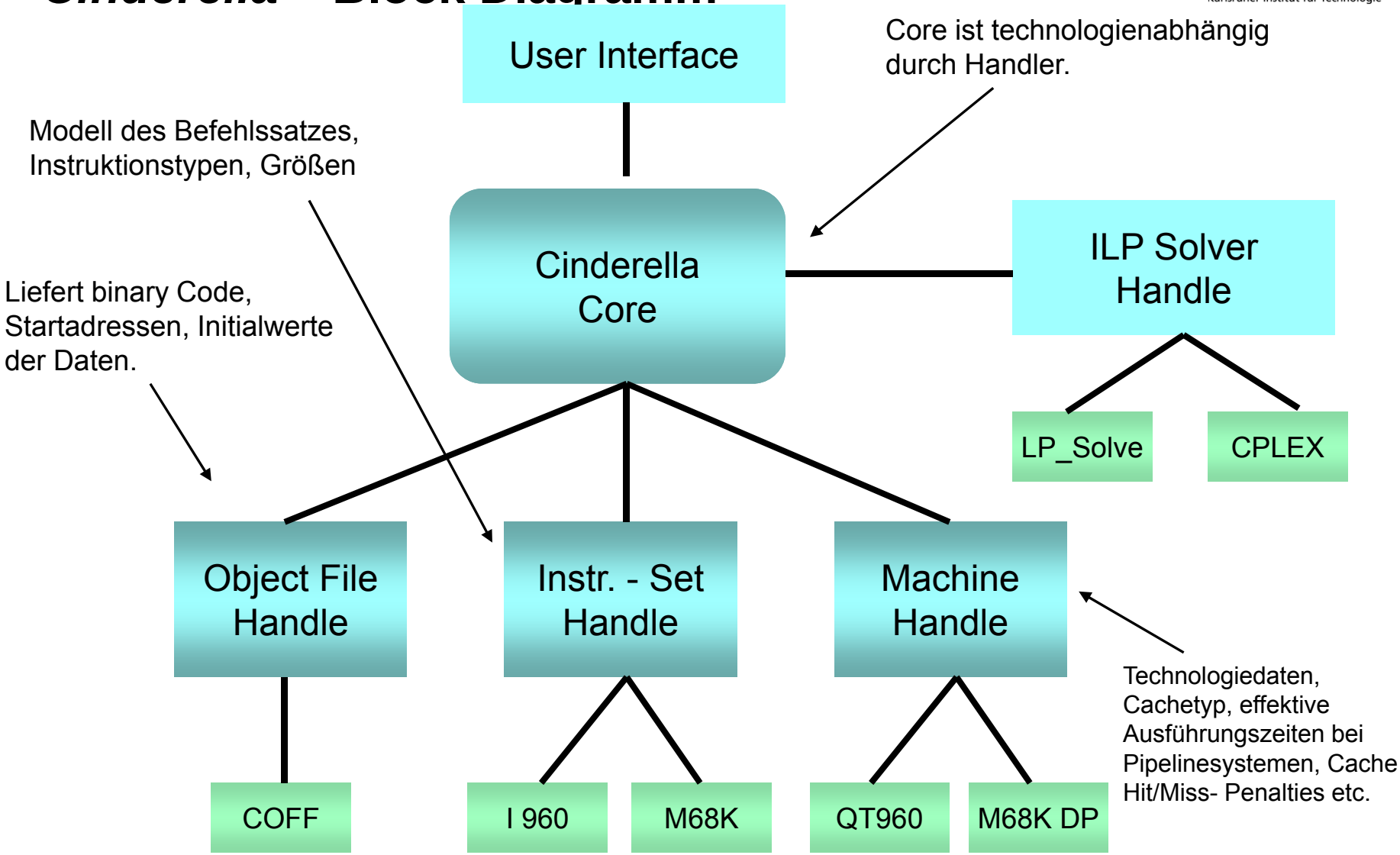
- einmalige Ausführung:

$$p_{s,ij} + p_{s,kl} + \underbrace{p_{s,e}}_{\leq 1} = 1$$

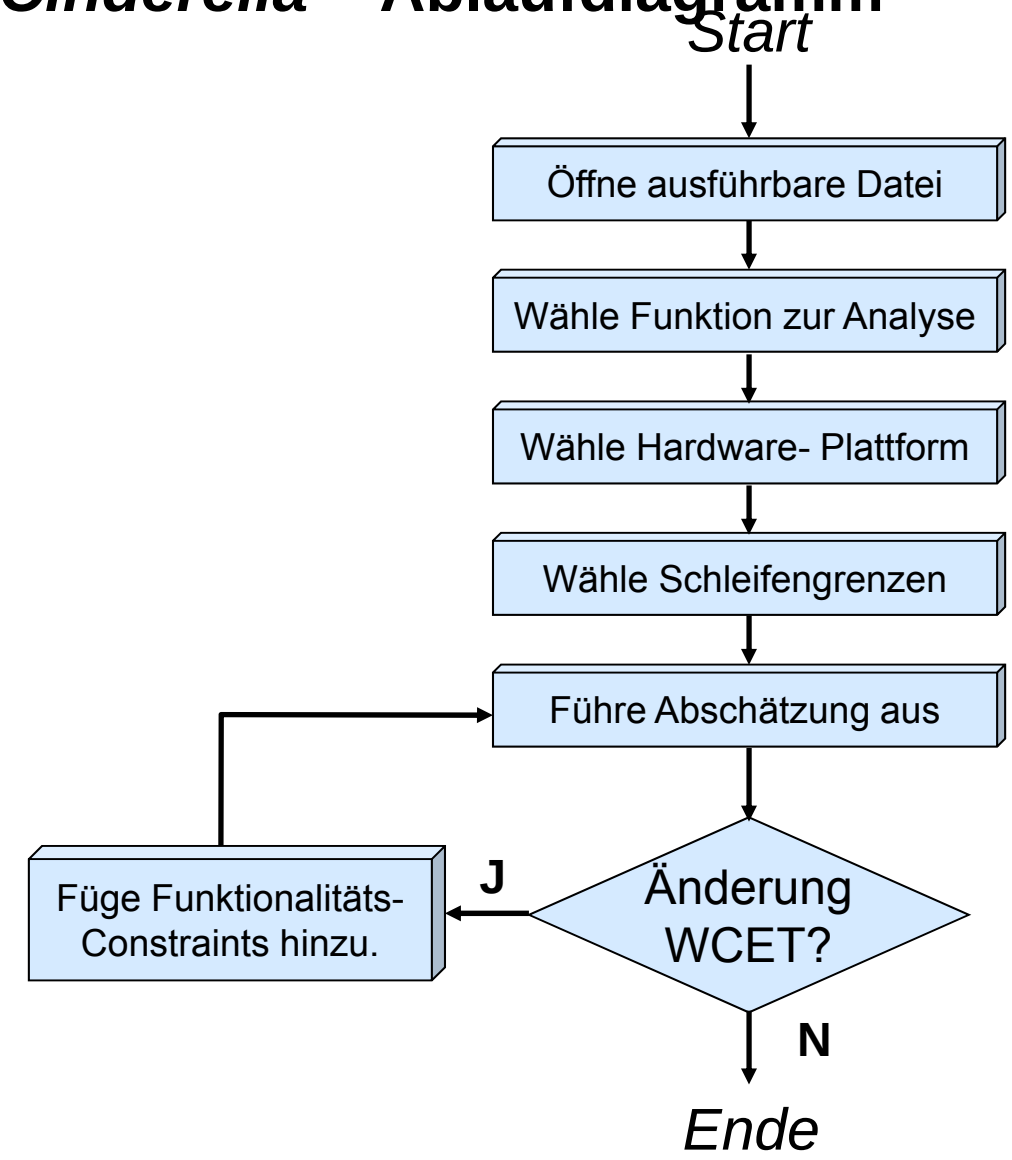
3.6.1 Anwendung: WCET mit Werkzeug *Cinderella*

- **Cinderella:** 1995, Sharad Malik, YauTsun Steven Li
  - Anwendung als Tool zur Cache- und statischen Timing-Analyse für Embedded Software
  - Unterstützt verschiedene Mikro-Architekturen unter Verwendung einer Architekturbeschreibung (zur Zeit nur Intel i960 und M68000)
  - Erlaubt Test / Vergleich der Laufzeiteigenschaften von Programmen auf einer Zielplattform
  - Liest den ausführbaren Assembler-Code eines Programms und konstruiert daraus die CFGs und CCGs
  - Anwender stellt die Schleifengrenzen und Pfadinformationen zur Verfügung
  - Cinderella verwendet den ILP Solver lp_solve zur Lösung der ILP Constraints
- **Verwendete Hardware**
  - Intel QT960 Entwicklungs-Board (CISC Prozessor)
  - Intel i960KB Prozessor (32 bit RISC Prozessor) mit 4–stufiger Instruktions-Pipeline bei 20MHz
  - 128KB Hauptspeicher
  - 512 Byte direct-mapped Instruction-Cache (32 x 16-Byte Zeilen)

3.6.1 Software-Performanz: *Cinderella* – Block Diagramm



3.6.1 Software-Performanz: *Cinderella* – Ablaufdiagramm



Bestimme das Dateiformat
Bestimme die Funktionsnamen

Bestimme ISA
Bestimme die Hardwaremodule

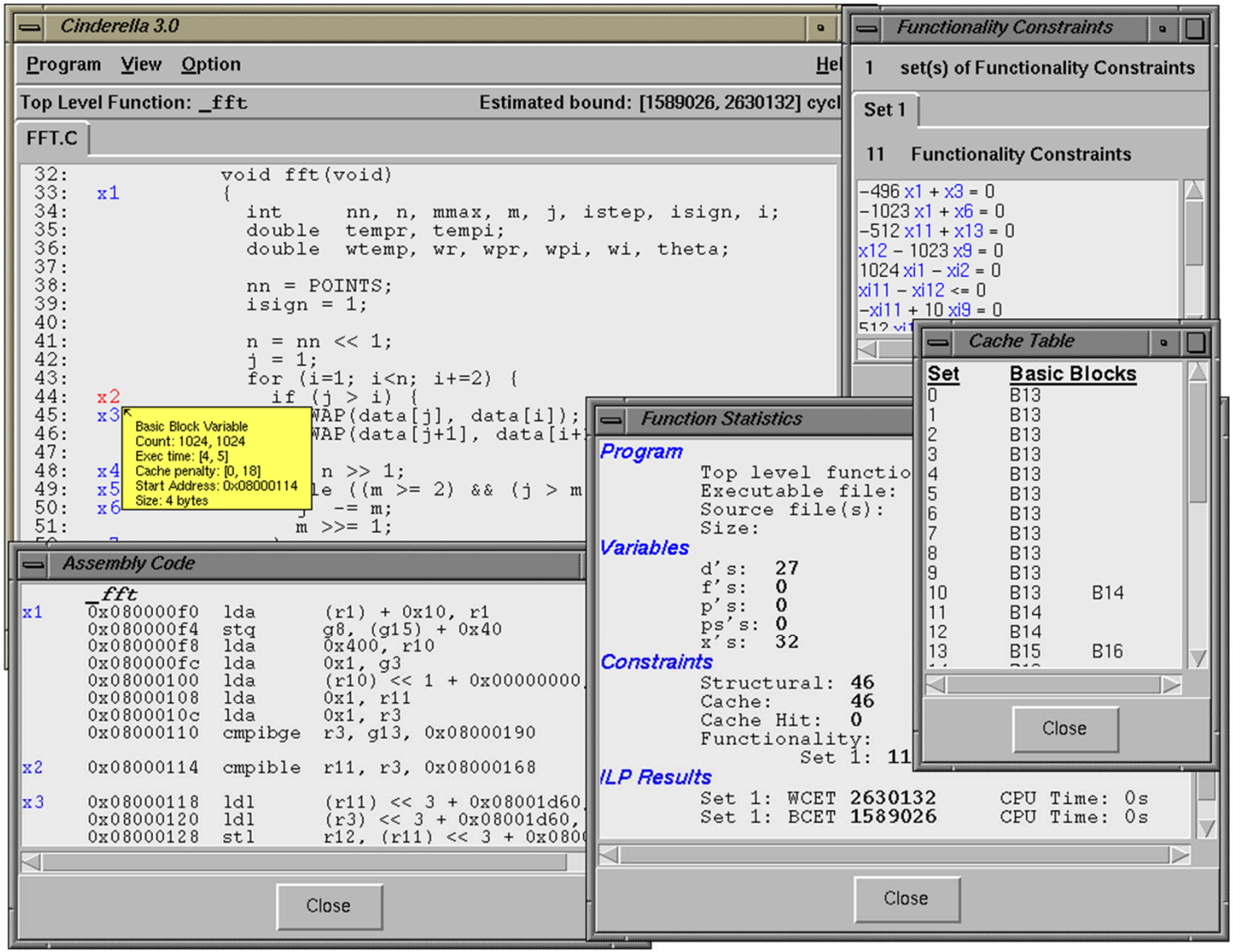
Erzeuge CFG, finde Schleifen
Gebe kommentierte Inform. aus.

Bestimme Block- Kosten
Führe eine Cache- Analyse aus.
Erzeuge Constraints

Starte ILP- Solver

Aktualisiere Constraint- Mengen

3.6.1 Software-Performanz: WCET mit dem Tool *Cinderella*



3.6.1 Software-Performanz: *Cinderella* Benchmarks

Übersicht der ausgeführten Benchmarks

Function	Description	Lines	Bytes
check_data	Example from Park's thesis	23	88
circle	Circle drawing routing in Gupta's thesis	100	1588
des	Data Encryption Standard	192	1852
dhry	Dhrystone benchmark	761	1360
djpeg	Decompression of 128x96 color JPEG	857	5408
fdct	JPEG forward discrete cosine transform	300	996
fft	1024-point Fast Fourier transform	57	500
line	Line drawing routine in Gupta's thesis	165	1556
matcnt	Summation of 2100x100 matrices from Arnold	85	460
matcnt2	Matcnt with inlined functions	73	400
piksrt	Insertion sort	19	104
sort	Bubble sort of 500 elements from Arnold	41	152
sort2	sort with inlined functions	30	148
stats	Sum, mean, var of two 1000 element arrays	100	656
stats2	stats with inlined functions	90	596
whetstone	Whetstone benchmark	196	596

Quelle: Shawn Schaffert

3.6.1 Software-Performanz: Cinderella Benchmarkergebnisse (I)

Karlsruher Institut für Technologie

Vergleich mit den tatsächlichen Programmlaufzeiten

Function	Measured WCET (cycles)	Estimated WCET (cycles)	Ratio
check_data	4.30×10^2	4.91×10^2	1.14
circle	1.45×10^4	1.54×10^4	1.06
des	2.44×10^5	3.70×10^5	1.52
dhry	5.76×10^5	7.57×10^5	1.31
djpeg	3.56×10^7	7.04×10^7	1.98
fdct	9.05×10^3	9.11×10^3	1.01
fft	2.20×10^6	2.63×10^6	1.20
line	4.84×10^3	6.09×10^3	1.26
matcnt	2.20×10^6	5.46×10^6	2.48
matcnt2	1.86×10^6	2.11×10^6	1.13
piksrt	1.71×10^3	1.74×10^3	1.02
sort	9.99×10^6	27.8×10^6	2.78
sort2	6.75×10^6	7.09×10^6	1.05
stats	1.16×10^6	2.21×10^6	1.91
stats2	1.06×10^6	1.24×10^6	1.17
whetstone	6.94×10^6	10.5×10^6	1.51

Quelle: Shawn Schaffert

3.6.1 Software-Performanz: Cinderella Benchmarkergebnisse (II)

- **Vergleich zwischen gemessenen und geschätzten Cache-Misses**

Program	DineroIII Simulation	Estimated Worst-Case Cache Misses	Ratio
circle	443	458	1.03
des	3872	4188	1.08
dhry	8304	8304	1.00
djpeg	230861	316394	1.37
fdct	63	63	1.00
line	99	101	1.02
stats	47	47	1.00
stats2	44	44	1.00
whetstone	18678	18678	1.00

Quelle: Shawn Schaffert

3.6.1 Software-Performanz: Cinderella Benchmarkergebnisse (III)

Karlsruher Institut für Technologie

ILP- Solver Performanz:

Function	d's	f's	p's	x's	Struct.	Cache	Funct.	ILP branches	Time(sec.)
check_data	12	0	0	40	25	21	5+5	1+1	0+0
circle	8	1	81	100	24	186	1	1	0
des	174	11	728	560	342	1059	16+16	13+13	171+197
dhry	102	21	503	504	289	777	24x4+26x4	1x8	0x3+2+0+1x2+4
djpeg	296	20	1816	416	613	2568	64	1	87
fdct	8	0	18	34	16	49	2	1	0
fft	27	0	0	80	46	46	11	1	0
line	31	2	264	231	73	450	2	1	3
matcnt	20	4	0	106	59	61	4	1	0
matcnt2	20	2	0	92	49	54	4	1	0
piksrt	12	0	0	42	22	26	4	1	0
sort	15	1	0	58	35	31	6	1	0
sort2	15	0	0	50	30	27	6	1	0
stats	28	13	75	180	99	203	4	1	0
stats2	28	7	41	144	75	158	4	1	0
whetstone	52	3	301	388	108	739	14	1	2

Quelle: Shawn Schaffert

3.6.2 Software-Performanz: Profiling & Tracing (I)

- „**W**isconsin **A**rchitectural **R**esearch **T**ool **S**et” (WARTS),
  Thomas Ball, James Larus: „Optimally Profiling and Tracing Programs“,
  ACM Transactions on Programming Languages and Systems

- **Tool Sammlung zur Performanceabschätzung** eines Programms durch **Profiling** und **Tracing** sowie deren Analyse nach erfolgter Programmausführung.

- **WARTS** enthält die folgenden Tools:
  - **QPT**, Programm- Profiler und Tracing-System ⇒ näher beleuchtet
  - **CPROF**, ein Cache-Performanz Profiler
  - **Tycho** und **DineroIII**, Cache Simulatoren
  - **EEL**, eine Bibliothek zum Ändern von ausführbaren Dateien

3.6.2 Software-Performanz: Profiling & Tracing (II)

- Das Profiling und Tracing erfolgt i.a. durch Einfügen von zusätzlichen Monitor- Code an bestimmten Stellen, der während der Programmausführung dann das Programmausführungsverhalten protokolliert.

- Das QPT- Tool verwendet dazu zwei Methoden:
  - **Profiling:** Es werden die Verzweigungs- bzw. Ausführungshäufigkeiten jedes sequentiellen Programm-Basisblocks gemessen.
  - **Tracing:** Die Ausführungsreihenfolge der Programm-Basisblöcke wird protokolliert.

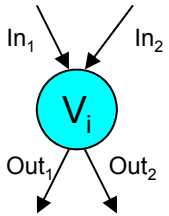
- Diese daraus gewonnenen Daten erlauben die Berechnung der Ausführungskosten der Prozeduren in einem Programm.

3.6.2 Software-Performanz: Profiling & Tracing mit WARTS (I)

- **Definition: Basisblock**
  - Ein Basisblock ist eine Sequenz von aufeinanderfolgenden Befehlen, in welche der Kontrollfluß eintritt und diese wieder verläßt, ohne Halt oder Möglichkeit daraus zu verzweigen, ausgenommen an deren Ende.

- **Definition: Kontrollflußgraph (CFG)**
  - Ein Kontrollflußgraph (CFG) ist ein gerichteter Graph (mit Wurzel) G=(V, E) mit einem speziellen EXIT- Knoten ($\neq$ Wurzel), so daß dieser Graph mit einem Programm auf folgende Weise korrespondiert:
    Jeder Knoten in V repräsentiert einen Basisblock und jede Kante in E repräsentiert eine Kontrollverzweigung von einem Basisblock zu einem anderen.

- **Definition: Gewichte W eines CFG**



  - Als Gewichte W des CFG werden allen Kanten aus E ein nicht-negativer Wert zugeordnet, wobei die Summe der Gewichte der eingehenden bzw. ausgehenden Kanten eines Knotens gleich sind (Bilanz: $\Sigma\ in_i = \Sigma\ out_i$).
  - Das Gewicht eines Knotens aus V ist einfach die Summe der eingehenden bzw. nur ausgehenden Kanten (=Ausführungshäufigkeit B_i).

3.6.2 Software-Performanz: Profiling & Tracing mit WARTS (II)

- **Definition: Kosten von G**
  - Die Kosten cost(G, pl, W) einer Teilmenge pl von Kanten (bzw. Knoten) ergibt sich aus der Summe der Kantengewichte (bzw. Knotengewichte) W.

- **Definition: Aufspannender Baum G´**
  - Ein aufspannender Baum eines gerichteten Graphen G ist ein Untergraph G´=(V´, E´) mit V´=V und E´$\subseteq$ E, so daß für jedes Knotenpaar (v, w) in G´ ein eindeutiger Pfad (nicht unbed. gerichtet) in G´ existiert, der v und w verbindet (keine Zyklen!).
  - Ein maximal aufspannender Baum G´ eines Graphen G mit Gewichten W ist ein aufspannender Baum, der die Kostenfunktion cost(G, E´, W) maximiert.

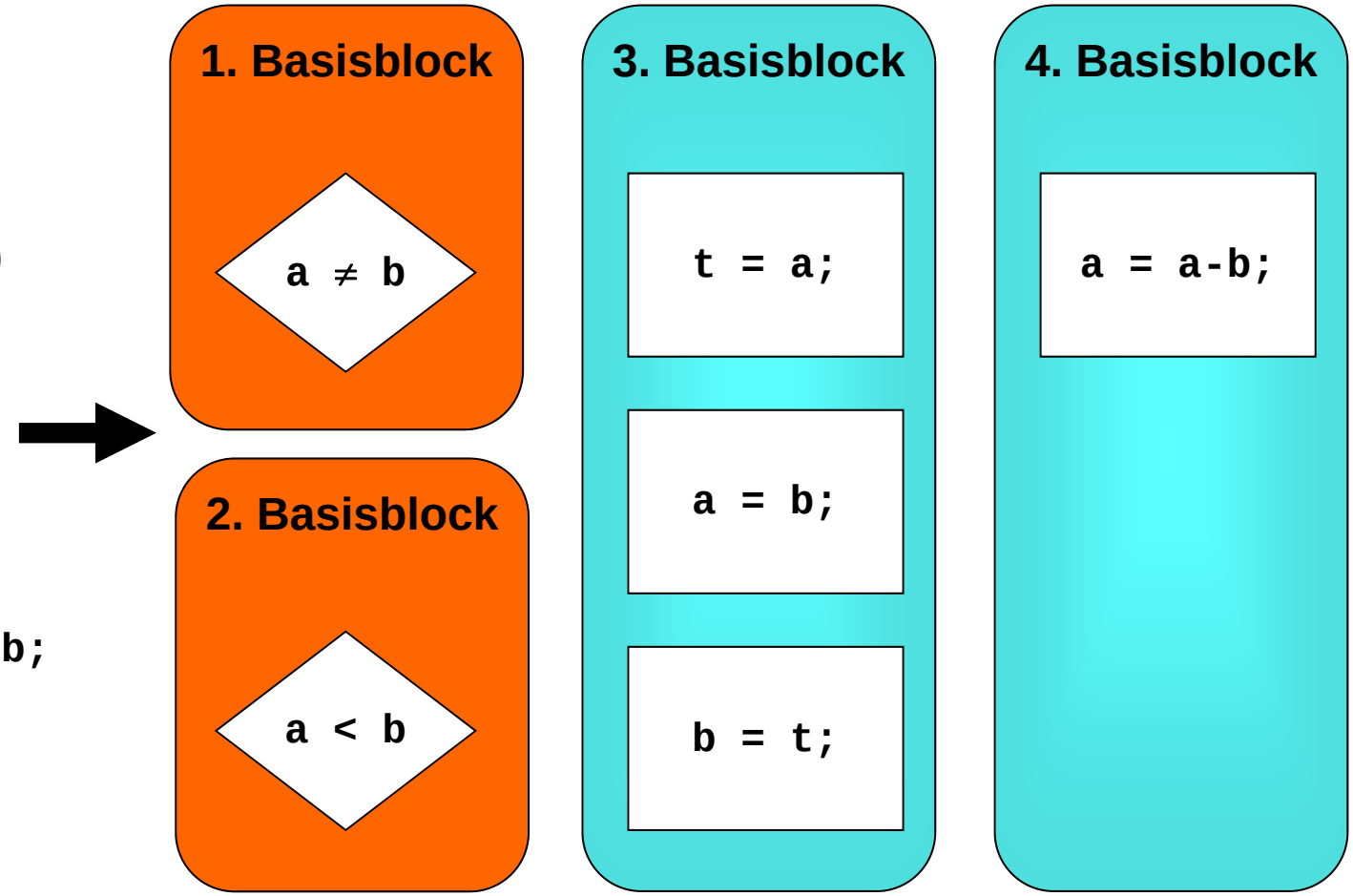
3.6.2 Software-Performanz: Profiling & Tracing mit WARTS (III)

- **Definition: Ausführung (Execution) EX eines CFG**
  - Eine Ausführung EX eines CFG ist ein gerichteter Pfad, der mit dem Wurzelknoten beginnt und mit dem EXIT- Knoten endet, wobei der EXIT-Knoten nur einmal auftritt.
  - Die Frequenz eines Knotens v oder einer Kante e in einer Ausführung EX ist die Anzahl/Häufigkeit, in der v oder e in EX auftauchen.
    (EXIT, ROOT = 1)
  - Die Kantenhäufigkeiten für jegliche Ausführung eines CFG ergeben die Gewichtung des CFG.

3.6.2 Software-Performanz: Profiling & Tracing mit WARTS (IV)

- Partitionierung einer Befehlssequenz in Basisblöcke

```
while (a ≠ b)
{
    if (a < b)
    {
        t = a;
        a = b;
        b = t;
    }
    else
    {
        a = a-b;
    }
}
```



3.6.2 Software-Performanz: Profiling & Tracing mit WARTS (V)

- Zur Bestimmung des Profilings eines Programms durch QPT sind in diesem Programm zusätzliche Zähler-Programmstücke einzufügen.
  - Platzierung kann in jedem Basisblock (= CFG- Knoten) erfolgen.
  - Jeder Zähler wird bei Ausführung des entsprechenden Basisblocks inkrementiert.

- **Nachteile dieser direkten Methode:**
  - Es werden mehr Zähler verwendet als wirklich benötigt.
  - Die Gesamtzahl der Inkrementoperationen während der Programmausführung ist größer als notwendig.

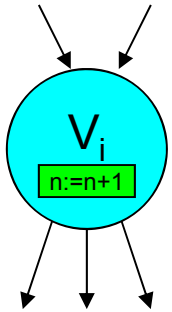
- Durch geeignete Plazierung der Zähler kann man ein Profiling ausführen, ohne dabei **alle** Basisblöcke berücksichtigen zu müssen.

- Zusätzlich sollte man die Zähler in Programmbereichen mit niedriger Ausführungsfrequenz platzieren $\Rightarrow$ weitere Einsparungen.
  - Erfordert Vorwissen über die Ausführungshäufigkeiten der Basisblöcke z.B. anhand des Programmcodes (feste Schleifen etc.).

- Führt auf Knoten-Frequenzproblem und Kanten-Frequenzproblem

3.6.2 Anwendungsbeispiel: Profiling & Tracing mit WARTS (VI)

- **Definition: Knoten-Frequenzproblem VF (G, pl)**

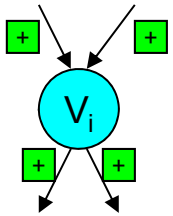


- Das Knoten- bzw. Vertex-Frequenzproblem VF (G, pl) dient der Bestimmung einer geeigneten Plazierung pl von Zählern im CFG G, wobei pl eine Menge von Knoten und/oder Kanten ist, die mit Zählern assoziiert sind, so dass die Häufigkeit eines jeden Knoten in einer Ausführung von G allein aus den Zählerwerten und dem CFG G abgeleitet werden kann.

- pl sollte VF(G, pl) lösen und cost(G, pl, W) für eine gegebene Gewichtung W minimieren. Ein solches pl wird als optimale Lösung betrachtet.

3.6.2 Anwendungsbeispiel: Profiling & Tracing mit WARTS (VII)

- **Definition: Kanten-Frequenzproblem EF (G, pl)**

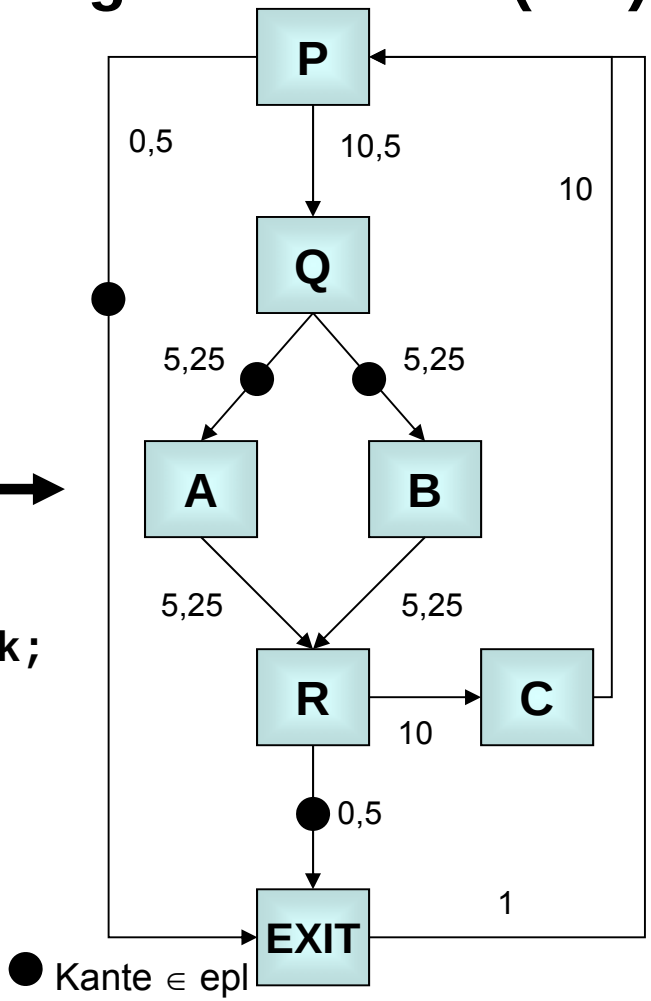


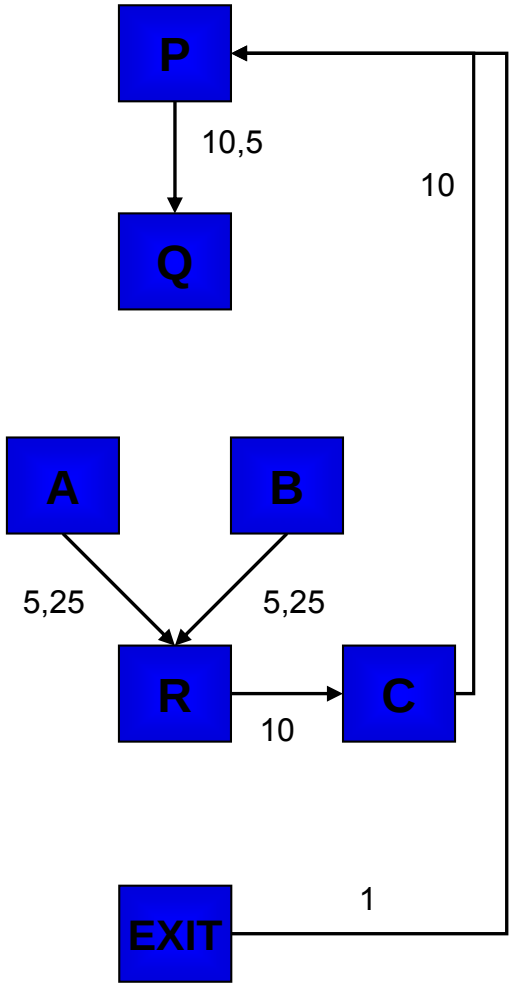
  - Das dem VF ähnliche Kanten-Frequenzproblem EF (G, pl) dient der Bestimmung einer geeigneten Plazierung pl von Zählern auf den Kanten im CFG G, wobei pl eine Menge von Übergangskanten des CFG ist, die mit Zählern assoziiert sind, so dass die Häufigkeit jedes Übergangs von einem Knoten zum anderen in einer Ausführung von G allein aus den Zählerwerten und dem CFG G abgeleitet werden kann.

  - Eine Lösung für das Kanten-Frequenzproblem löst auch offensichtlich das Knoten- Frequenzproblem. (Summe der eingehenden bzw. ausgehenden Kantenhäufigkeiten eines Knotens).
    Zur Beschränkung der kombinatorischen Anzahl an Lösungen wird pl nur auf **Knoten vpl** oder nur auf **Kanten epl** beschränkt.

3.6.2 Software-Performanz: Profiling & Tracing mit WARTS (VIII)

```
program
  while P do
  begin
    if Q then
      A;
    else
      B;
    if R then break;
    C;
  end;
end
```





Programmbeispiel

CFG mit Gewichten W

Maximal-aufspannender Baum (zusammenhängend + zyklenfrei !)

- epl liefert eine minimale Lösung bzgl. EF(G, epl) ⇔ (E- epl) bildet einen maximal aufspannenden Baum G´ von G

3.6.2 Software-Performanz: Profiling & Tracing mit WARTS (IX)

- Der **Profiling-Algorithmus** in QPT berechnet nun VF(G, epl) bzw. EF(G, epl) getrennt für jede Prozedur durch Anwendung des o.a. Verfahrens mittels der Bestimmung **Maximal-aufspannender-Bäume** und nachfolgende Bestimmung der Kantenfrequenzen.

- Die Eigenschaften des MAB (Zyklenfreiheit und zusammenhängend) erlauben die **eindeutige Rekonstruktion** der restlichen Ausführungs-häufigkeiten der Kanten im MAB anhand der ausgewählten Zähler.

- Die Ausführungshäufigkeiten jedes Basisblocks sind das Endergebnis des Algorithmus, so daß weitere Transformationen auf den sich ergebenden Daten eine Identifikation der Performanz-Engpässe des Programms ermöglichen.

3.6.2 Software-Performanz: Profiling & Tracing mit WARTS (X)

- Die zweite von QPT verwendete Methode ist der bereits erwähnte **Programm-Tracing Algorithmus.**
  - Damit ist es möglich, die Abfolge der Basisblöcke in einem Programmablauf aufzuzeichnen.
  - Die Ermittlung der Ausführungsabfolge einer Programmsequenz liefert neben der Möglichkeit, Ausführungshäufigkeiten zu bestimmen, auch die wichtige Möglichkeit, die Performanz-Flaschenhälse zu identifizieren.
    - Beispiel: Reduktion des Kommunikationsoverheads zwischen einem Programmteil, der in Hardware und einem der in Software ausgeführt wird. Das ständige Wechseln des Ausführungskontextes zwischen beiden Teilen verursacht zu viel Kommunikation.
  - Das sich daraus ergebende **Tracing-Problem** behandelt die Frage, wie mit minimalen Aufwand genug Information gesammelt werden kann, um daraus vollständige Ausführungssequencen (-> Traces) ableiten zu können.

3.6.2 Software-Performanz: Profiling & Tracing mit WARTS (XI)

- Eine Möglichkeit, um das **Tracing-Problem** zu lösen, besteht z.B. darin, jeden Basisblock so zu modifizieren, dass dieser bei seiner Ausführung eindeutige Spuren/Zeugen (= **witness**) hinterlässt, deren späteres Auslesen aus einer Tracedatei eine **vollständige Rekonstruktion** des Programmablaufs zulässt.

- **Effiziente Methode:** nur bei Basisblöcken einen witness zu schreiben, die Ziel einer Kontrollflussverzeigung (=Prädikat) sind.

- Unter Annahme der Existenz einer Standardrepräsentation für die witness-Marken (z.B. Byte, Halbworte oder Worte je witness), kann das Tracing-Problem mit weniger Zeit und Speicheraufwand gelöst werden als jede andere Lösung, indem nur Spuren hinterlassen werden, wenn Kanten traversiert werden (nicht Knoten ausgeführt werden) und geeignete Kanten dafür zuvor ausgewählt wurden. Dies führt für das Tracing-Problem auf folgende Definitionen:

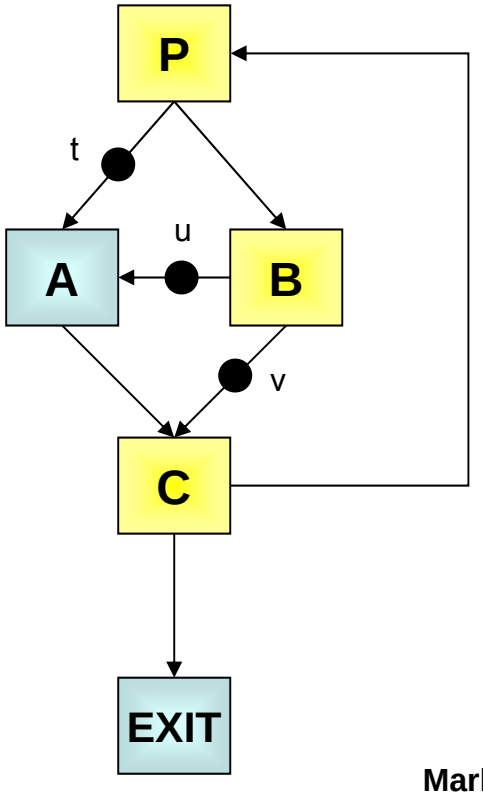
3.6.2 Anwendungsbeispiel: Profiling & Tracing mit WARTS (XII)

- **Definition: Zeugen- (witness)- freier Pfad im CFG G**
  - Ein Pfad im CFG G wird als **Zeugen/witness- frei** im Bezug auf eine Kantenmenge epl bezeichnet, genau dann wenn es keine Kante im Pfad gibt, die zu epl gehört.
- **Definition: Zeugen/Witness Menge witness(G, epl, p ,q)**
  - Gegeben sei ein CFG G und eine zuvor bestimmte Kantenmenge epl sowie eine Kante $p \rightarrow q$, wobei p ein Prädikatenknoten ist. Dann ist die witness- Menge (bezgl. Knoten q) für den Prädikatenknoten p **witness(G, epl, p, q) =**

$$\{w \mid p \rightarrow q \in \text{epl (und schreibt auf witness- Kante w)}\}$$
$$\cup \quad \{w \mid x \rightarrow y \in \text{epl (und schreibt auf witness- Kante w) und es existiert ein witness- freier Pfad } p \rightarrow q \rightarrow \ldots \rightarrow x\}$$
$$\cup \quad \{\text{EOF} \mid \text{es existiert ein witness- freier Pfad } p \rightarrow q \rightarrow \ldots \rightarrow \text{EXIT}\}$$

3.6.2 Software-Performanz: Profiling & Tracing mit WARTS (XIII)

- Beispiel für ein Tracing einer Funktion: Tracesequenz der Marken {t, u, v, EOF}



Witnesses-Mengen (zur Rekonstruktion):

***witness*(P, A) = {t}** ***witness*(B, C) = {v}**

***witness*(P, B) = {u, v}** ***witness*(C, P) = {t, u, v}**

***witness*(B, A) = {u}** ***witness*(C, EXIT) = {EOF}**

Prädikatenknoten: P, B, C

Instruktionen aus Prozessorhandbuch erlauben Hochrechnung der Gesamtausführungszeit.

Ausführung: **P A C P B A C P B C EXIT**

Marker hier abgearbeitet: **↑t ↑u ↑v ↑ EOF**

Trace-Kanten

witnesses aus epl

Prädikatenknoten

3.6.2 Software-Performanz: Profiling & Tracing mit WARTS (XIV)

- Wenn ein **witness w** in zwei Mengen **witness(G, epl, p, a)** und **witness(G, epl, p, b)** mit a ≠ b ist, dann sind zwei verschiedene Ausführungen von G möglich, die gleiche Trace-Protokolle erzeugen.

- Dies macht die Rekonstruktion der Programmausführung nur basierend auf dem Kontrollfluß und den Trace-Protokollen unmöglich.
  - Beispiel aus der vorhergehenden Grafik:
  - Wenn man annimmt, daß die **Kante P→ A keinen witness erzeugt**, dann liefern witness(P, A)={u, v, EOF} und witness(P, B)={u, v}.
  - Die Ausführungen (P, A, C, P, B, C, EXIT) und (P, B, C, EXIT) erzeugen den gleichen Trace (V, EOF).

- Dies motiviert eine Definition des Tracing-Problems.

3.6.2 Software-Performanz: Profiling & Tracing mit WARTS (XV)

- **Definition: Tracing- Problem TP(G, epl)**
  - Eine Menge von Kanten epl, die witness-Einträge schreibt, löst das Tracing- Problem TP(G, epl) für einen gegebenen CFG G $\Leftrightarrow$ für jeden Prädikatenknoten (=Kontrollflußverzweigungsknoten) p in G mit Nachfolgern q_1, ..., q_m für alle Paare (q_i, q_j) und für $i \neq j$ gilt:

$$\mathbf{witness(G, epl, p, q_i) \cap witness(G, epl, p, q_j) = \varnothing}$$

3.6.2 Software-Performanz: Profiling & Tracing mit WARTS (XVI)

- Für eine witness-Platzierung epl, die das Tracing- Problem TP(G, epl) löst, haben Larus und Ball einen Algorithmus (einschließlich Korrektheitsnachweis) angegeben, welcher die Ausführung EX von G für jeden gemessenen Trace rekonstruieren kann:

```
Program
pc := root-vertex(G);
output(pc);
while (pc ≠ EXIT) do
        begin
        if not IsPredicate(pc)      then pc := successor(G, pc);
                                    else pc := read(trace)
        output(pc);
        end;
end.
```

- Dieser Algorithmus ist ebenso im QPT Programm-Profiler und Tracing-System implementiert worden.
- Instruktions-Traces sind die Grundlage für Simulation und Analyse und werden ebenso in Debugging- und Code-Optimierungen verwendet.

3.6.3 Software - Kosten

- Prozessor
- Größe des Programmspeichers
  - *Instr_Size(j)* ist der Speicherbedarf der generischen Instruktion *j*

$$\text{Progsize}_{B_i} = \sum_{j \in B_i} \text{Instr\_Size(j)}$$

- Größe des Datenspeichers
  - ein Programm hat eine Menge *D* von Deklarationen
  - *Data_Size(d)* ist der Speicherbedarf der Deklaration *d*

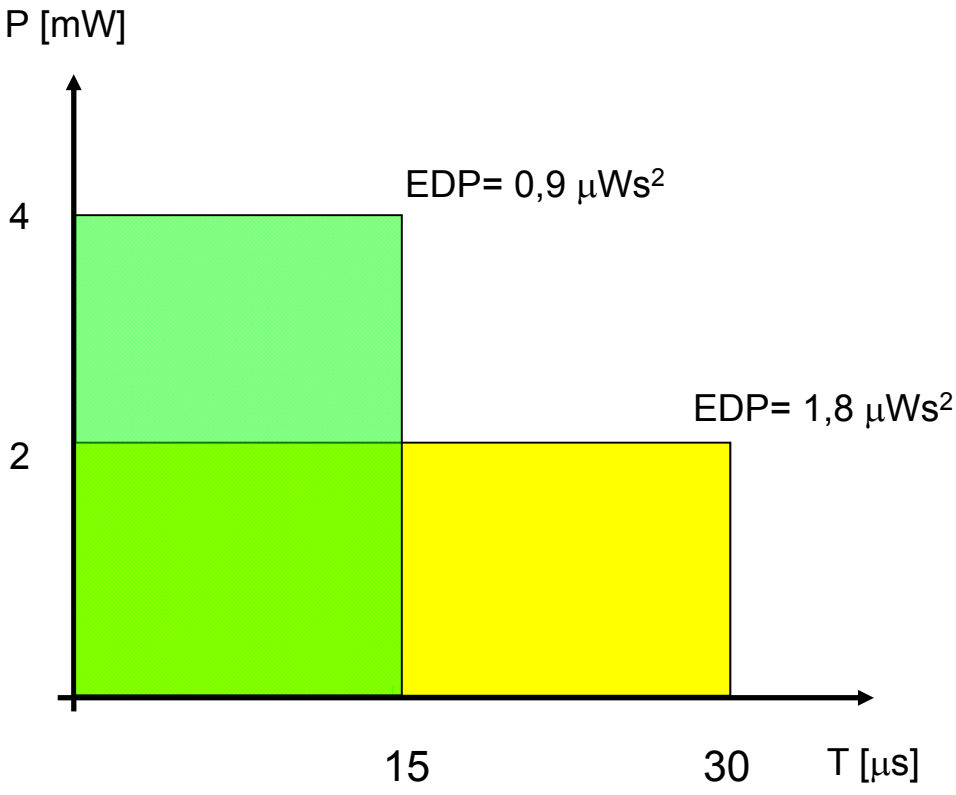
$$\text{Data\_Size} = \sum_{d \in D} \text{Data\_Size(d)}$$

3.6.3 Software: Verlustleistung und Energie (I)

- **Leistung** (power dissipation)
  - P [W]
  - wichtig für Dimensionierung von: packaging, power supply, cooling
- **Energie** (power-delay product)
  - $E = P_{avg} \bullet T_{exe}$ [Ws]
  - Wichtig für mobile Geräte (Batterie/Akku - Lebenszeit).
  - Sinnvolles Maß bei Systemen, die mit einer **festen Rate** betrieben werden.
  - Energie / Instruktion
  - Leistung auf einen Taktzyklus bezogen: [µW/MHz]
  - bei Prozessoren oft:
    - [µW/MIPS] bzw. [MIPS/µW]
    - [µW/SPEC] bzw. [SPEC/µW]

3.6.3 Software: Verlustleistung und Energie (II)

- **Energy-Delay Product**
  - EDP = E • T_{exe} [Ws^2]
  - Nützliche Metrik zur Bestimmung der vollst. Effizienz eines Entwurfs, während eine bestimmte Funktion ausgeführt wird.
  - Sinnvolles Maß bei Systemen, die mit **maximaler Rate** betrieben werden.



- bei Prozessoren:
  - [$MIPS^2/\mu W$]
  - [$SPEC^2/\mu W$]

3.6.3 Mobilkommunikation: Prozessoren

Processor (Vendor)	Techn.[µ]	VDD [V]	Clock [MHz]	Power [mW]	MIPS	MIPS/W	MIPS²/mW
StrongARM (Intel)	0.35	2	230	360	268	744	200
ARM710 (VLSI)	0.8	3.3	25	120	30	250	8
ARM940T (VLSI)	0.35	3.3	150	675	(a)160	(a)237	(a)38
MMC2001 (Motorola)	0.35	2	34	80	31	387	12
TR4102 (LSI)	0.25	1.8	80	40	(a)90	(a)2250	(a)203
SH7708 (Hitachi)	0.5	3.3	25	95	25	263	7
SH7750 (Hitachi)	0.25	1.8	200	1600	300	188	56

(a) = abgeschätzt

MIPS Ratings für einen Dhrystone Benchmark

Quelle: Jean-Michel Puiatti, *Instruction-Level Parallelism for Embedded Processors*, PhD Thesis, EPFL, 1999

3.6.3 Mobilkommunikation: VLIW-Prozessoren

Processor (Vendor)	Techn.[µ]	VDD [V]	Clock [MHz]	Power [mW]	MIPS	MIPS/W	MIPS²/mW
‘C6201 (Texas Instr.)	0.25	2.5	200	4600	1600	348	557
SC140 (Mot. /Lucent)	0.13	1.5	300	500	3000	6000	18000
TM1000 (Philips)	0.35	3.3	100	4000	2500	625	1563
Merced HP/Intel)	0.18	?	800	(a)70000	6400	(a)91	(a)585

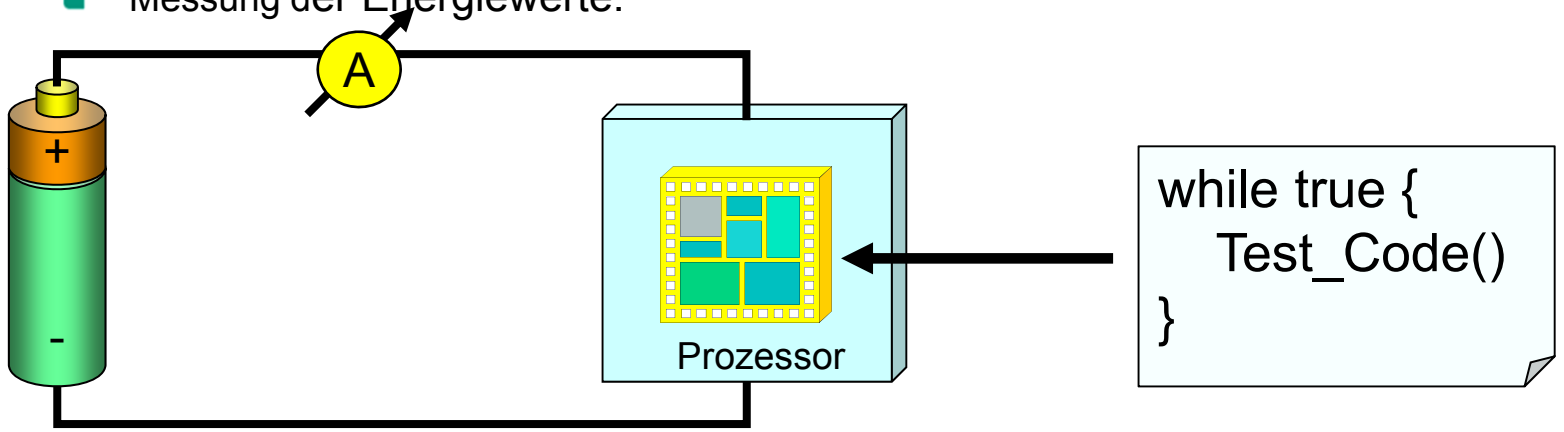
(a) = abgeschätzt

MIPS Ratings sind Spitzenwerte

Quelle: Jean-Michel Puiatti, *Instruction-Level Parallelism for Embedded Processors*, PhD Thesis, EPFL, 1999

3.6.3 Software: Simulation von Leistung und Energie

- Simulation auf Instruktions-Ebene
  - jede Instruktion benötigt eine bestimmte Energie
  - jedes Paar von aufeinander-folgenden Instruktionen benötigt zusätzlich eine bestimmte Energie
  - einfaches Modell, benötigt nur Instruktionssatzsimulator
  - DSP-Genauigkeiten < 10% erreicht
  - Messung der Energiewerte:



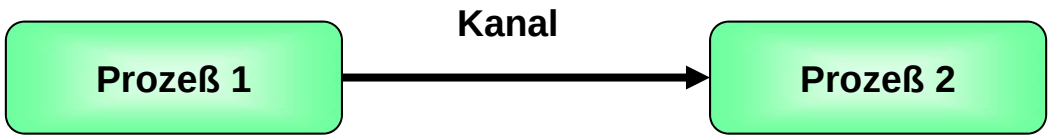
- Simulation auf Architektur-Ebene (genauer)
  - Kapazitätsmodelle für einzelne Blöcke: ALUs, Register, Controller, Cache, …
  - Aktivitäten der Blöcke simuliert
  - komplexes Modell, benötigt “cycle-accurate” Simulator
  - genauer als Leistungsabschätzung auf Instruktionsebene
  - Kapazitätsmodelle basieren auf konkreten Fertigungsprozessen
  - Orinocco Tool (Nebel), XPower (Xilinx)

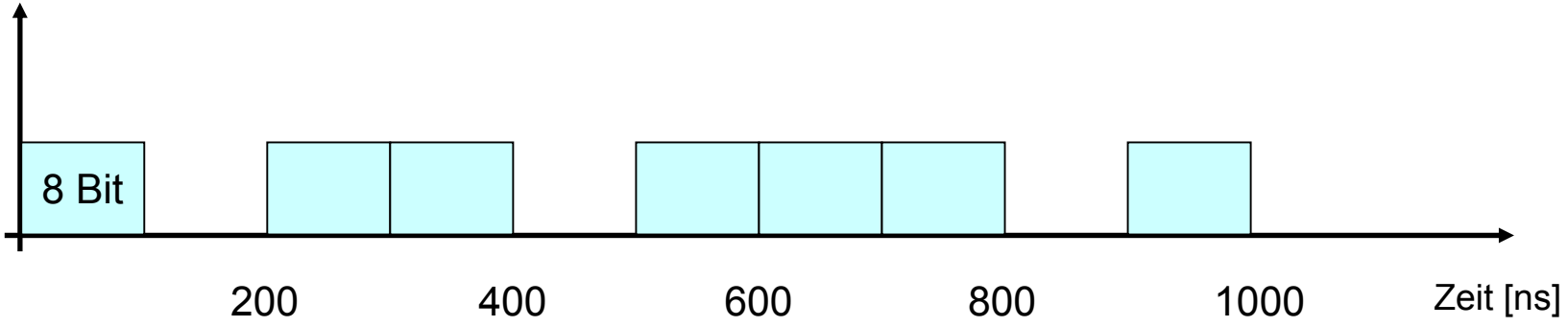
- Was ist Software-Performanz? Welche Metriken gibt es?
- Was ist die WCET? Auf was ist zu beachten?
- Welche Formulierung kann abgeleitet werden?
- Können Caches modelliert werden? Falls ja, mit welchen Einschränkungen?
- Was sind Profiling und Tracing?
- Welche Werkzeuge gibt es?
- Wieso wird zwischen Leistung und Energie unterschieden?

Inhalt

3.7 Hardware/Software: Kommunikation-Performanz (I)

- Prozesse schicken Botschaften über Kanäle





Spitzen-Rate = 8 Bit / 100 ns = 80 Mbit/s

Durchschnitts- Rate = 56 Bit / 1000 ns = 56 Mbit/s

3.7 Hardware/Software: Kommunikation-Performanz (II)

- **Modelle für Kommunikationszeit T_{com}**

$$T_{com} = T_{offset} + \frac{messagesize}{bitrate}$$

T_{offset} … Initialisierungszeit

$messagesize$ … in Bit

$bitrate$ … in Bit / sec

$$T_{com} = T_{tx} + T_{ch} + T_{rx}$$

T_{tx} … Transmitter-Verarbeitungszeit

T_{ch} … Kanal-Übertragungszeit

T_{rx} … Receiver-Verarbeitungszeit

- Wieso ist die Performanz der Kommunikation wichtig?
- Welche Auswirkungen kann die Kommunikations-Performanz haben?