Compiler für Eingebettete Systeme (CfES)

Sommersemester 2009

Dr. Heiko Falk

Technische Universität Dortmund

Lehrstuhl Informatik 12

Entwurfsautomatisierung für Eingebettete Systeme

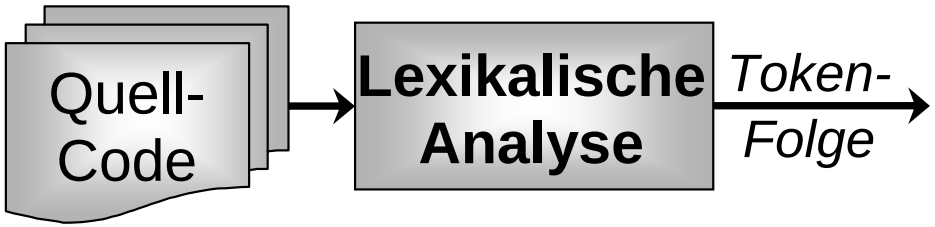
Kapitel 2

Interner Aufbau von Compilern

Gliederung der Vorlesung

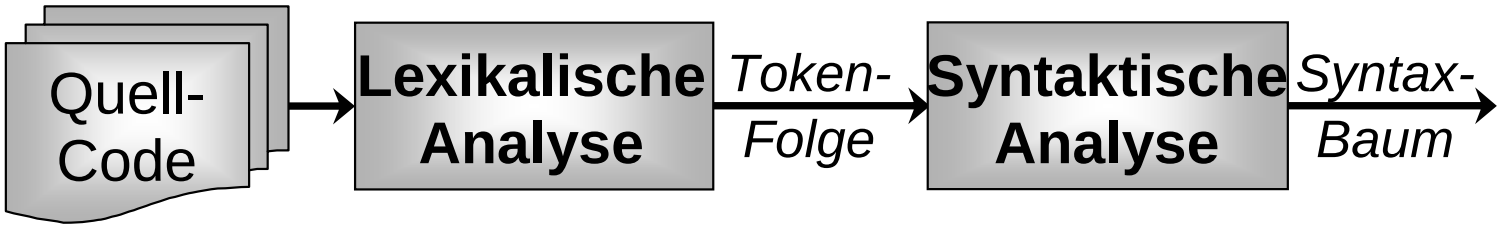
- Kapitel 1: Compiler für Eingebettete Systeme
- **Kapitel 2: Interner Aufbau von Compilern**
  - Compilerphasen
  - Interne Zwischendarstellungen
  - Optimierungen & Zielfunktionen
- Kapitel 3: Prepass-Optimierungen
- Kapitel 4: HIR Optimierungen und Transformationen
- Kapitel 5: Instruktionsauswahl
- Kapitel 6: LIR Optimierungen und Transformationen
- Kapitel 7: Register-Allokation
- Kapitel 8: Compiler zur $WCET_{EST}$-Minimierung
- Kapitel 9: Ausblick

Das Frontend



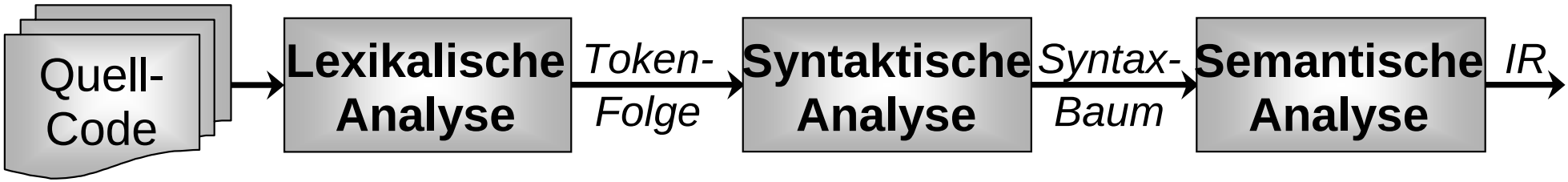
- **Lexikalische Analyse *(Scanner)*:**
  - Zerlegung des Quellprogramms in lexikalische Einheiten *(Token)*
  - Erkennung von Token (reguläre Ausdrücke, endliche Automaten)
  - Token: Zeichenfolge von Bedeutung in Grammatik der Quellsprache (z.B. Bezeichner, Konstanten, Schlüsselworte)

Das Frontend



- **Syntaktische Analyse *(Parser)*:**
  - Sei *G* Grammatik der Quellsprache
  - Entscheidung, ob Tokenfolge aus *G* ableitbar ist.
  - Syntaxbaum: Baumförmige Darstellung des Codes anhand während Ableitung benutzter Regeln aus *G*
  - Fehlerbehandlung

Das Frontend

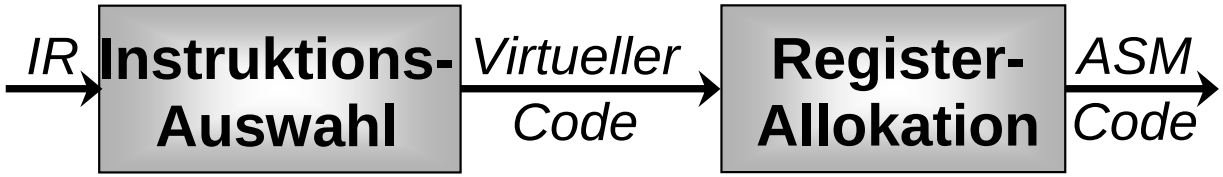


- **Semantische Analyse *(IR Generator)*:**
  - Namensanalyse (z.B. Gültigkeitsbereiche von Symbolen)
  - Prüfung, daß jeder Ausdruck korrekten Typs ist *(Typanalyse)*
  - Aufbau von Symboltabellen (Abbildung von Bezeichnern zu deren Typen und Positionen)
  - Erzeugung einer Internen Zwischendarstellung *(Intermediate Representation, IR)* zur weiteren Verarbeitung

Das Backend

- **Instruktionsauswahl *(Code Selector)*:**
  - Auswahl von Maschinenbefehlen zur Implementierung einer IR
  - *Oft – Generierung von Virtuellem Code:* Nicht lauffähiger Assemblercode; Annahme unendlich vieler Virtueller Register, anstatt begrenzt vieler Physikalischer Register
  - *Alternativ – Generierung von Code mit Stack-Zugriffen:* Lauffähiger Assemblercode; sehr eingeschränkte Nutzung von Registern; Variablen werden im Speicher gehalten (Bsp.: GCC)

Das Backend

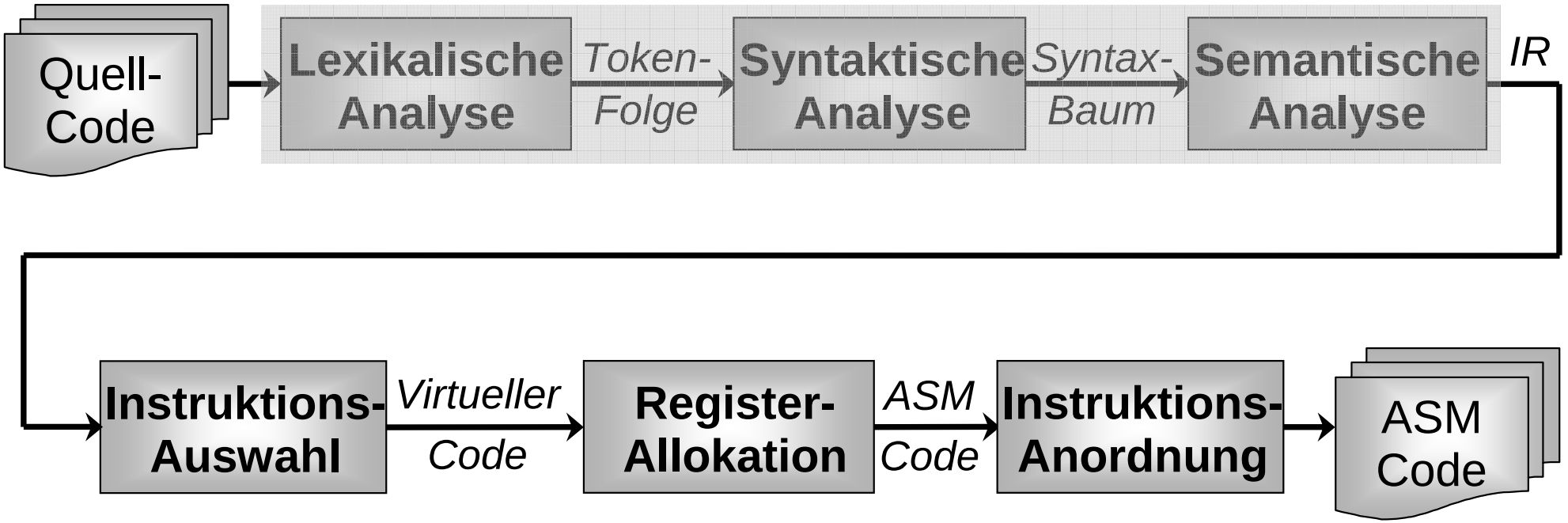


- **Registerallokation:**
  - *Entweder:* Abbildung Virtueller auf Physikalische Register
  - *Oder:* Ersetzen von Stack-Zugriffen durch Speicherung von Daten in Registern
  - Einfügen von Speicher-Transfers *(Aus-/Einlagern, Spilling)* falls zu wenig physikalische Register vorhanden

Das Backend

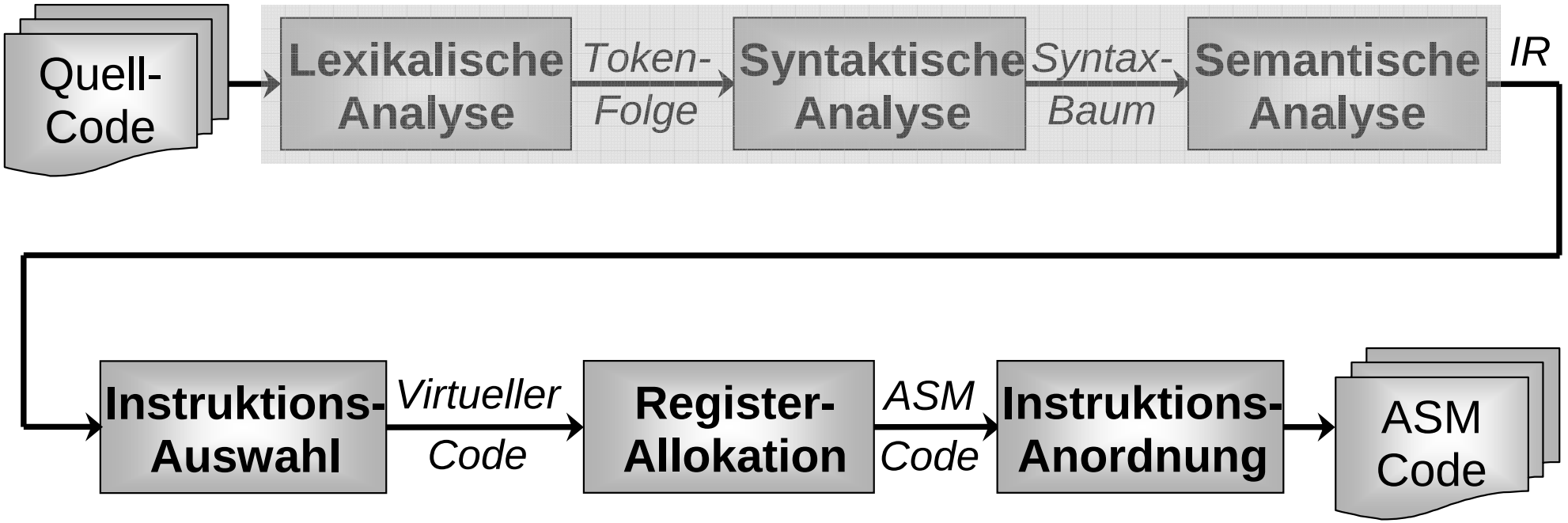
- **Instruktionsanordnung *(Scheduler)*:**
  - Umordnen von Maschinenbefehlen zur Erhöhung der Parallelität
  - Abhängigkeitsanalyse zwischen Maschinenbefehlen (Daten- & Kontroll-Abhängigkeiten)

Frontend & Backend

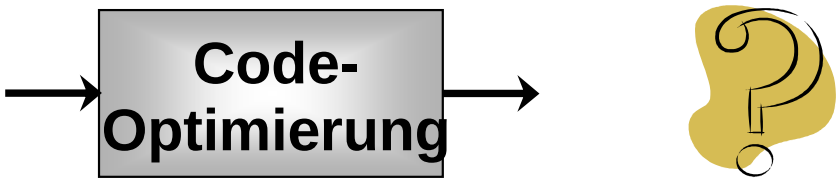


- **Vorlesung “Compiler für Eingebettete Systeme”:**
  - ✓ Frontend nicht weiter betrachtet (☞ Vorlesung “Übersetzerbau”)
  - ✓ *Schwerpunkt: Backend & Compiler-Optimierungen*

Preisfrage



Wo sollten Compiler-Optimierungen angesiedelt sein?



Begriff “Code-Optimierung”

- **Definition *(Optimierung)*:**
  - Compilerphase, die Code einliest, ändert und ausgibt.
  - Code-Änderung erfolgt mit Ziel der *Verbesserung* des Codes.

- **Bemerkungen:**
  - Optimierungen erzeugen i.d.R. keinen *optimalen Code* (oft unentscheidbar), sondern (hoffentlich) *besseren Code*.
  - Code-Verbesserung erfolgt bzgl. einer *Zielfunktion*.

Begriff “Code-Optimierung”

- **Vorhandensein formaler Code-Analysen:**
  - Code-Änderungen müssen wieder zu korrektem Code führen.
  - Optimierung muss entscheiden, wann Änderungen am Code vorgenommen werden dürfen, und wann nicht.
  - *Formale Code-Analysen* helfen bei dieser Entscheidung.
  - Beispiele: Kontroll- & Datenflußanalyse, Abhängigkeitsanalyse, ...

Voraussetzung zur Code-Optimierung

- **Benötigte Infrastruktur zur Optimierung:**
  - Effektive interne Darstellung von Code,
    - Code-Manipulation leicht ermöglicht
    - Notwendige Analysen für Optimierungen bereitstellt

  ☞ Interne Zwischendarstellung *(IR)*

Wo sollten Compiler-Optimierungen angesiedelt sein?

- **Optimierungen finden (i.d.R.) auf IR-Ebene im Compiler statt**

Gliederung der Vorlesung

- Kapitel 1: Compiler für Eingebettete Systeme
- **Kapitel 2: Interner Aufbau von Compilern**
  - Compilerphasen
  - Interne Zwischendarstellungen
  - Optimierungen & Zielfunktionen
- Kapitel 3: Prepass-Optimierungen
- Kapitel 4: HIR Optimierungen und Transformationen
- Kapitel 5: Instruktionsauswahl
- Kapitel 6: LIR Optimierungen und Transformationen
- Kapitel 7: Register-Allokation
- Kapitel 8: Compiler zur $WCET_{EST}$-Minimierung
- Kapitel 9: Ausblick

Abstraktionsniveaus von IRs

```
float a[20][10];
... a[i][j+2] ...;
```

- **High-Level**

  `t1 ← a[i,j+2]`

- **Medium-Level**

  $t_1 \leftarrow j+2$

  $t_2 \leftarrow i*20$

  $t_3 \leftarrow t_1+t_2$

  $t_4 \leftarrow 4*t_3$

  $t_5 \leftarrow \text{addr } a$

  $t_6 \leftarrow t_5+t_4$

  $t_7 \leftarrow *t_6$

- **Low-Level**

  `r1 ← [fp-4]`

  `r2 ← r1+2`

  `r3 ← [fp-8]`

  `r4 ← r3*20`

  `r5 ← r4+r2`

  `r6 ← 4*r5`

  `r7 ← fp-216`

  `f1 ← [r7+r6]`

Abstraktionsniveaus von IRs

- **High-Level IRs:**
  - Repräsentation sehr nah am Quellcode
  - Oft: Abstrakte Syntaxbäume
  - Variablen & Typen zur Speicherung von Werten
  - Erhaltung komplexer Kontroll- & Datenflussoperationen (insbes. Schleifen, if-then / if-else Ausdrücke, Array-Zugriffe [])
  - Rücktransformation der High-Level IR in Quellcode leicht

*[**S. S. Muchnick,** Advanced Compiler Design & Implementation, Morgan Kaufmann, 1997]*

Abstraktionsniveaus von IRs

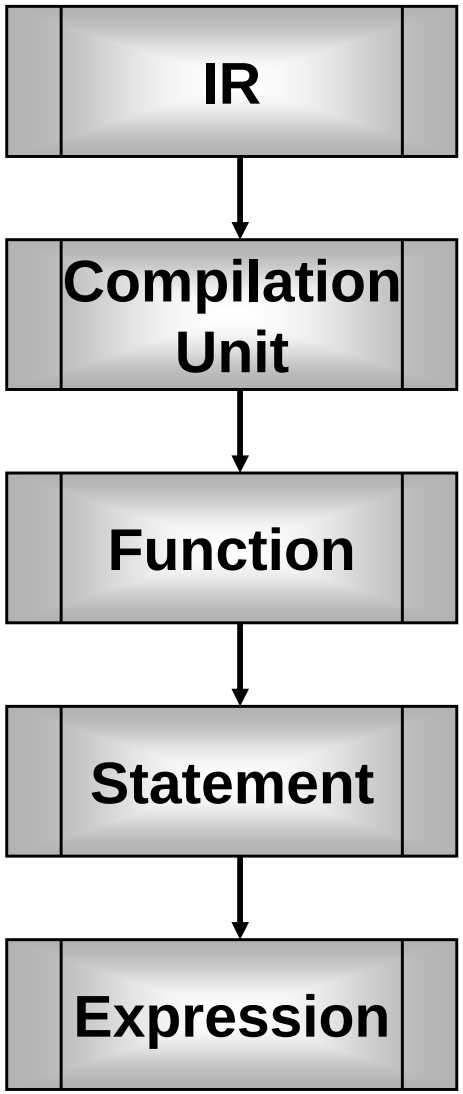
- **Medium-Level IRs:**
  - Drei-Adreß-Code: $a_1 \leftarrow a_2$ *op* a_3;
  - IR-Code unabhängig von Quell-Sprache & Ziel-Prozessor
  - Temporäre Variablen zur Speicherung von Werten
  - Komplexe Kontroll- & Datenflußoperationen vereinfacht (Labels & Sprünge, Zeiger-Arithmetik)
  - Kontrollfluß in Form von *Basisblöcken*

- **Definition:** Ein *Basisblock* $B=(I_1, ..., I_n)$ ist eine Befehlssequenz maximaler Länge, so daß
  - B nur durch die erste Instruktion I_1 betreten wird, und
  - B nur durch die letzte Instruktion I_n verlassen wird.

Abstraktionsniveaus von IRs

- **Low-Level IRs:**
  - Repräsentation von Maschinen-Code
  - Operationen entsprechen Maschinenbefehlen
  - Register zur Speicherung von Werten
  - Transformation der Low-Level IR in Assemblercode leicht

High-Level IR: ICD-C

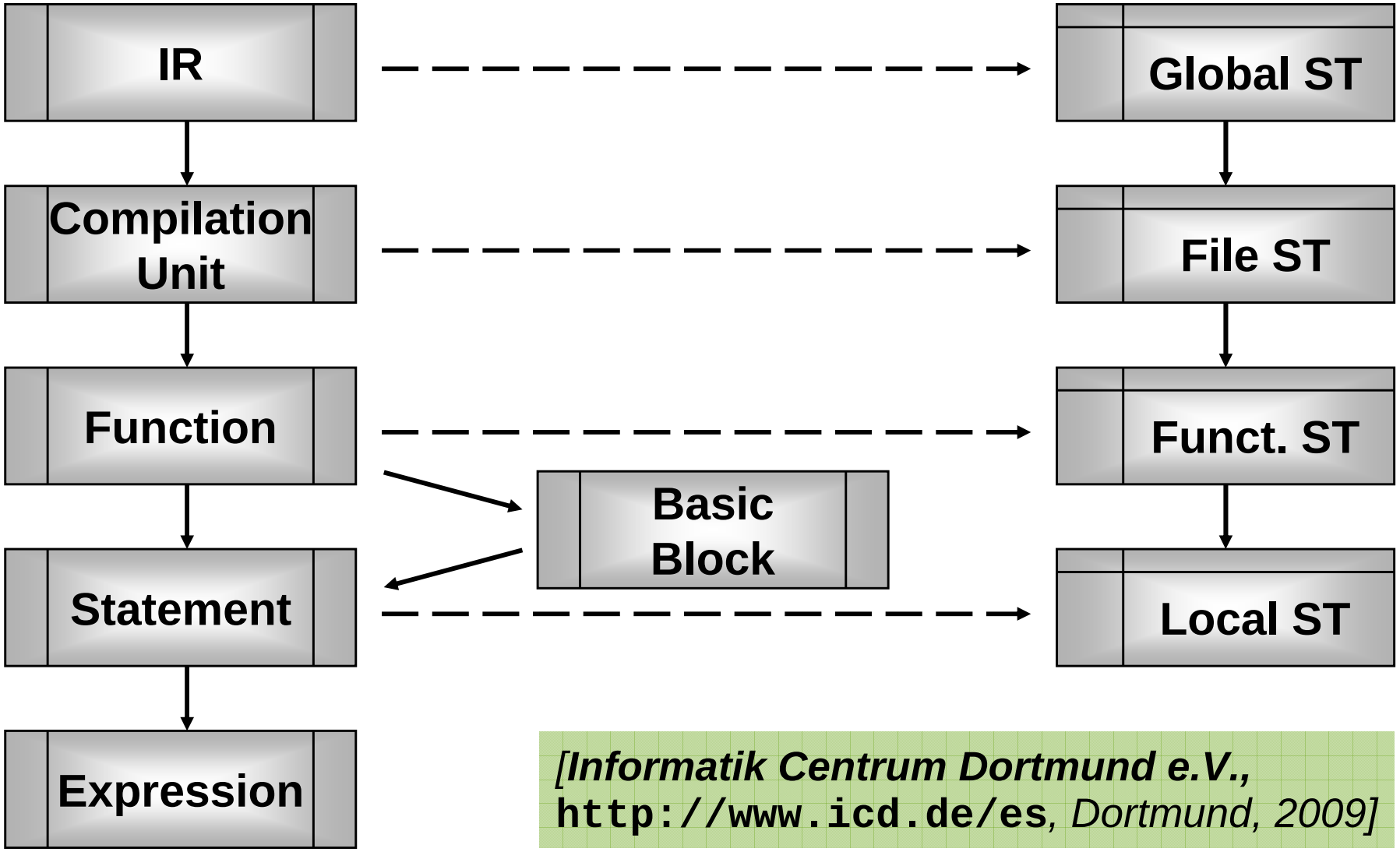


- 1 C-File bei gleichzeitiger Übersetzung mehrerer Quell-Dateien

- Loop Statements (for, do-while, while-do)
- Selection Statements (*if*, *if-else*, *switch*)
- Jump Statements (*return*, *break*, *continue*, ...)
- ...

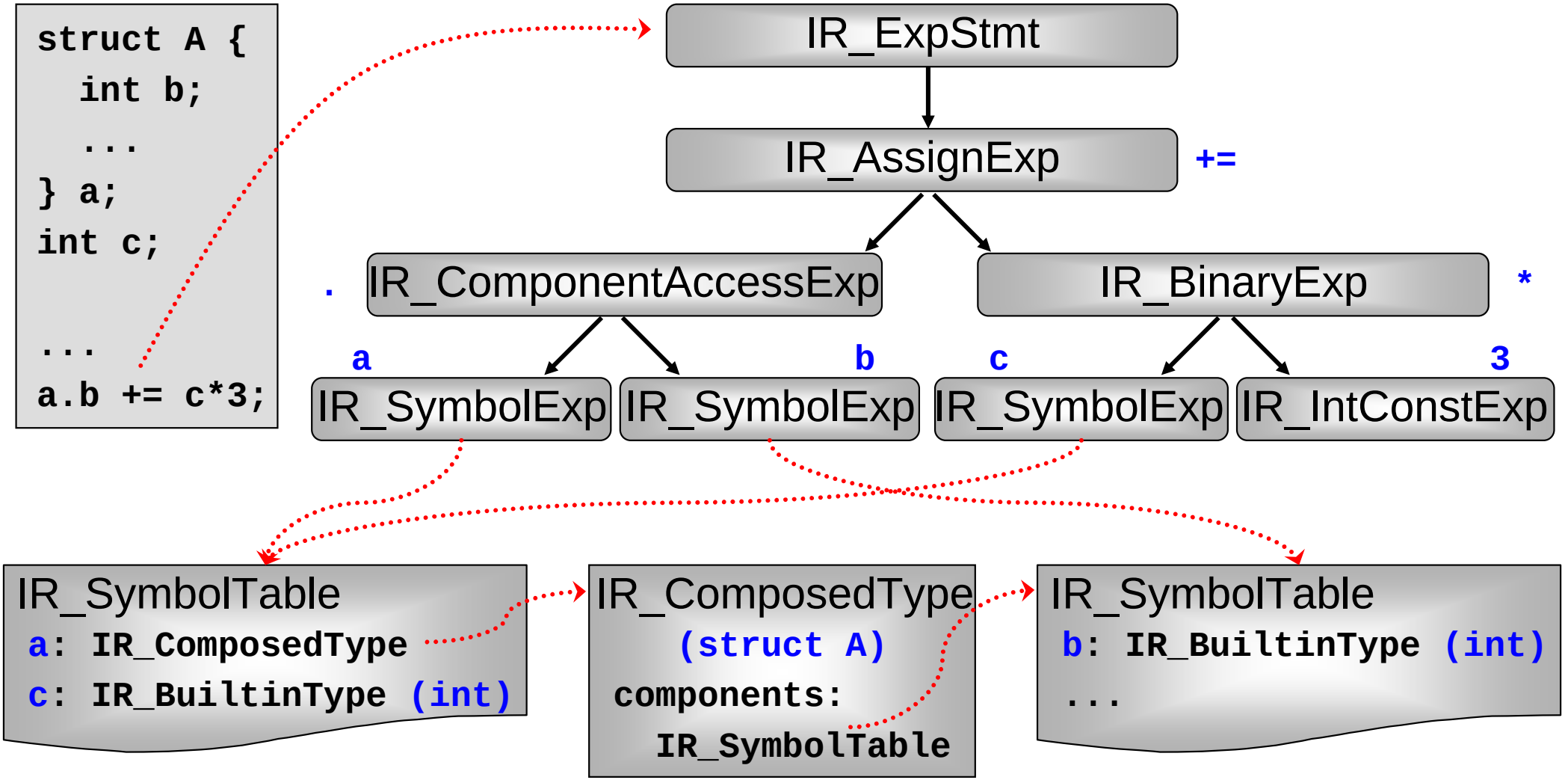
- Binary & Unary Expressions (+, -, *, /, ...)
- Zuweisungsoperatoren (=, +=, -=, ...)
- Index & Komponentenzugriff (*a[x]*, *a.x*, ...)
- ...

High-Level IR: ICD-C



[Informatik Centrum Dortmund e.V., **`http://www.icd.de/es`***, Dortmund, 2009]*

ICD-C: Code-Beispiel



ICD-C: Features

- **ANSI-C Compiler Frontend:** C89 + C99 Standards
  GNU Inline-Assembler
- **Enthaltene Analysen:** Datenflußanalysen
  Kontrollflußanalysen
  Schleifenanalysen
  Zeigeranalyse
- **Schnittstellen:**
  - ANSI-C Dump der IR als Schnittstelle zu externen Tools
  - Schnittstelle zur Code-Selektion in Compiler-Backends
- **Interne Strukturen:**
  - Objektorientiertes Design (C++)

Medium-Level IR: MIR

- **MIR Program:** 1 – *N* Program Units *(d.h. Funktionen)*
- **Program Unit:** **`begin`** MIRInst* **`end`**

- **MIR-Instruktionen:**
  - Quadrupel: 1 Operator, 3 Operanden *(d.h. 3-Adreß-Code)*
  - Instruktionstypen:
    Zuweisungen, Sprünge (**`goto`**), Bedingungen (**`if`**),
    Funktionsaufruf & -rücksprung (**`call`**, **`return`**),
    Parameterübergabe (**`receive`**)
  - Können MIR Ausdrücke *(Expressions)* enthalten

Medium-Level IR: MIR

- **MIR Ausdrücke:**
  - Binäre Operatoren: **`+`**, **`-`**, **`*`**, **`/`**, **`mod`**, **`min`**, **`max`**
  - Relationale Operatoren: **`=`**, **`!=`**, **`<`**, **`<=`**, **`>`**, **`>=`**
  - Schiebe- & Logische Operatoren: **`shl`**, **`shr`**, **`shra`**, **`and`**, **`or`**, **`xor`**
  - Unäre Operatoren: **`-`**, **`!`**, **`addr`**, **`cast`**, **`*`**

- **Symboltabelle:**
  - Enthält Variablen und symbolische Register
  - Einträge haben Typen: **`integer`**, **`float`**, **`boolean`**

*[**S. S. Muchnick,** Advanced Compiler Design & Implementation, Morgan Kaufmann, 1997]*

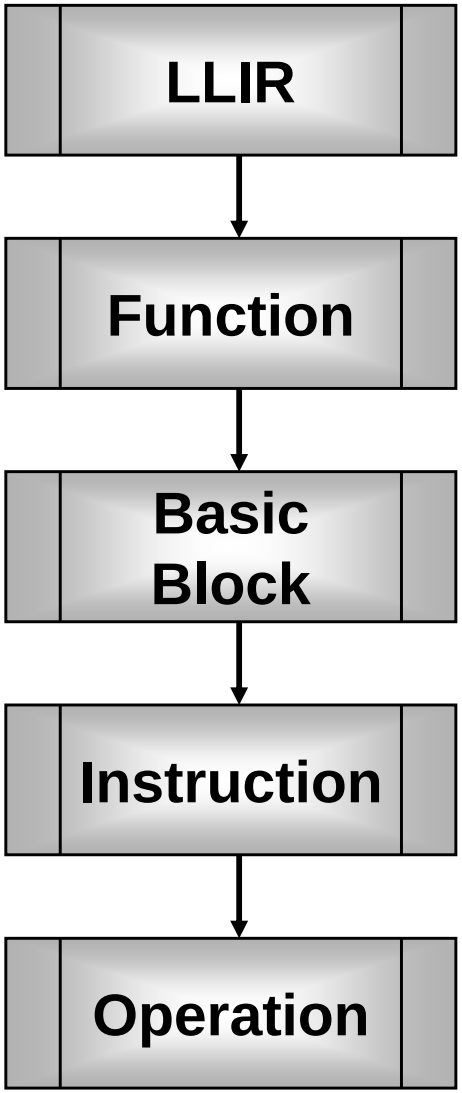
MIR: Eigenschaften

- **MIR ist keine High-Level IR:**
  - Nähe zur Quellsprache fehlt
  - High-Level Konstrukte fehlen: Schleifen, Array-Zugriffe, ...
  - Nur wenige, meist simple Operatoren präsent

- **MIR ist keine Low-Level IR:**
  - Nähe zur Zielarchitektur fehlt: Verhalten von Operatoren ist maschinenunabhängig definiert
  - Konzept von Symboltabellen, Variablen & Typen nicht low-level
  - Abstrakte Mechanismen zum Funktionsaufruf, Rücksprung und Parameterübergabe

☞ ***MIR ist eine Medium-Level IR.***

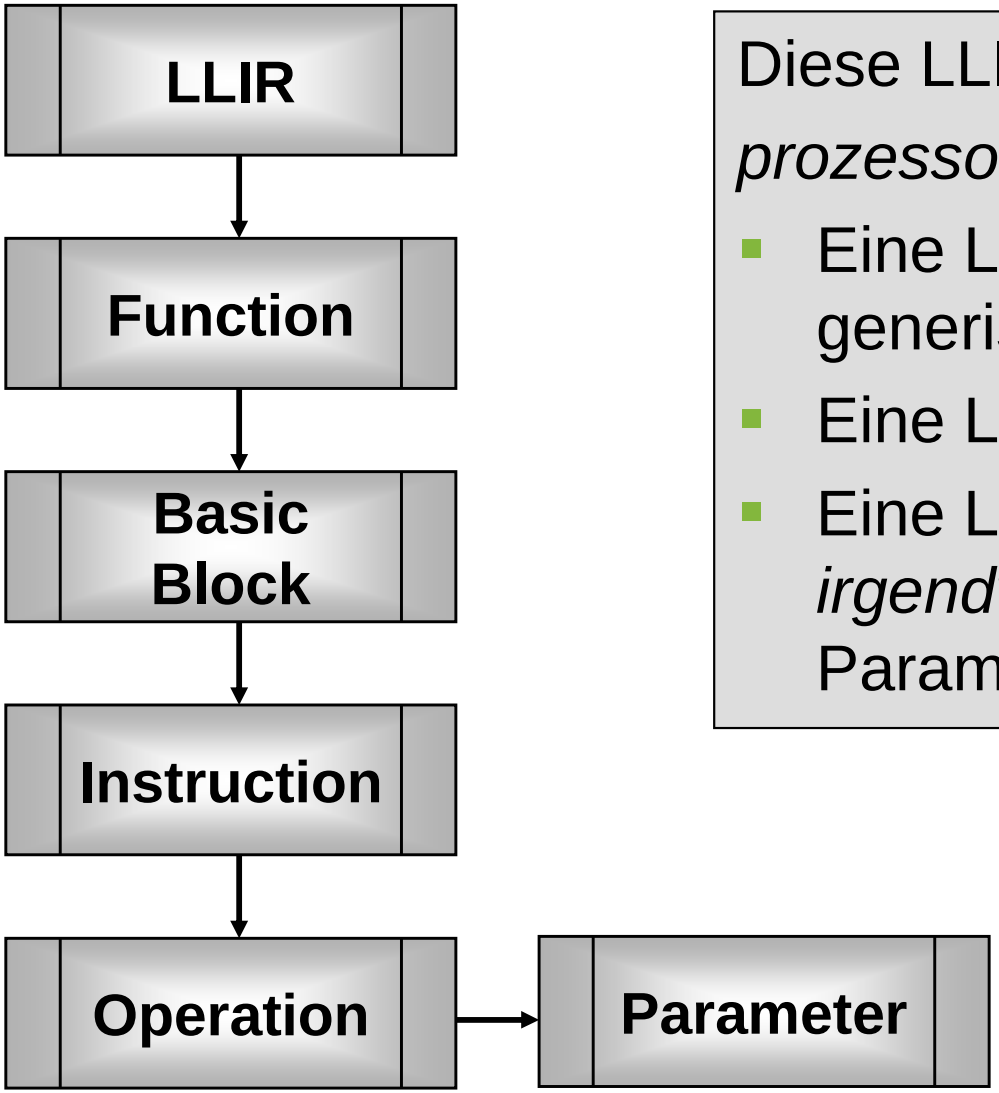
Low-Level IR: LLIR



- Maschinen-Instruktion
- Enthält 1-*N* Maschinen-Operationen
- Operationen werden parallel ausgeführt *(☞ VLIW)*

- Maschinen-Operation
- Enthält Assembler-Opcode (z.B. **`ADD`**, **`MUL`**, ...)
- Enthält 0-*M* Parameter

Low-Level IR: LLIR

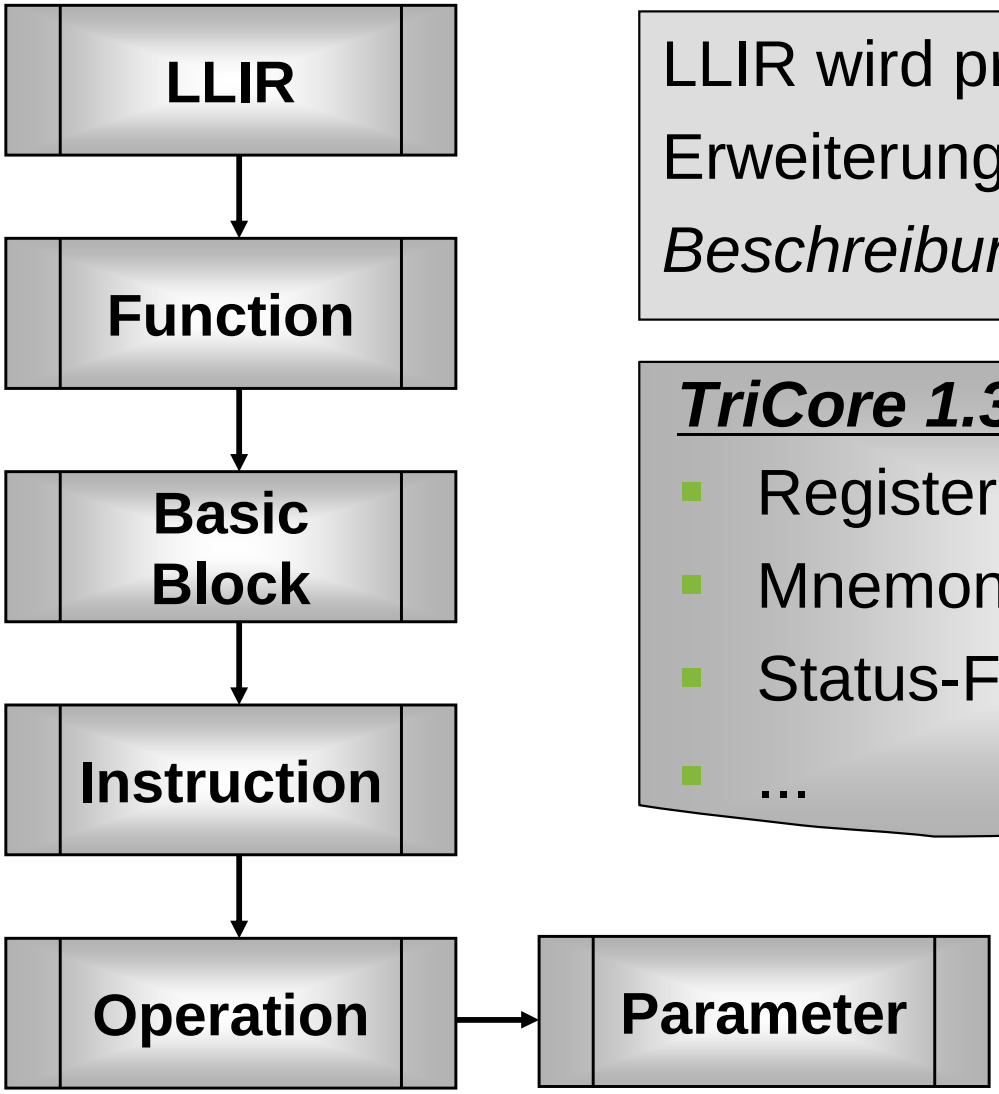


Diese LLIR-Struktur ist vollkommen *prozessor-unabhängig*:

- Eine LLIR besteht aus *irgendwelchen* generischen Funktionen
- Eine LLIR-Funktion besteht aus...
- Eine LLIR-Operation besteht aus *irgendwelchen* generischen Parametern

- Register
- Integer-Konstanten & Labels
- Adressierungsmodi
- ...

Low-Level IR: LLIR



LLIR wird prozessor-spezifisch durch Erweiterung um eine *Prozessor-Beschreibung*:

TriCore 1.3:

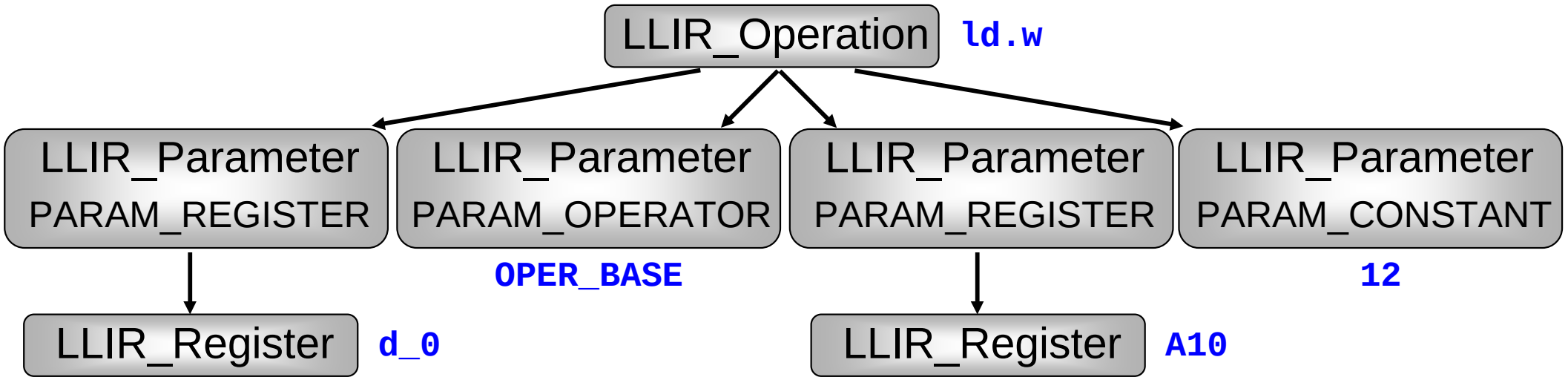
- Register = {**D0**, ..., **D15**, **A0**, ..., **A15**}
- Mnemonics = {**ABS**, **ABS.B**, ..., **XOR.T**}
- Status-Flags = {**C**, **V**, ..., **SAV**}
- ...

*[**Informatik Centrum Dortmund e.V.**, **http://www.icd.de/es**, Dortmund, 2009]*

LLIR: Code-Beispiel *(Infineon TriCore 1.3)*

```
ld.w %d_0, [%A10] 12;
```

- Lade Speicherinhalt von Adresse **`[%A10] 12`** in Register **`d_0`**
- *Erinnerung:* Register **`A10`** = Stack-Pointer ☞ Phys. Register
- Adresse **`[%A10] 12`** = Stack-Pointer + 12 Bytes
  (sog. Base + Offset-Adressierung)
- TriCore hat kein Register **`d_0`** ☞ Virtuelles Datenregister



LLIR: Features

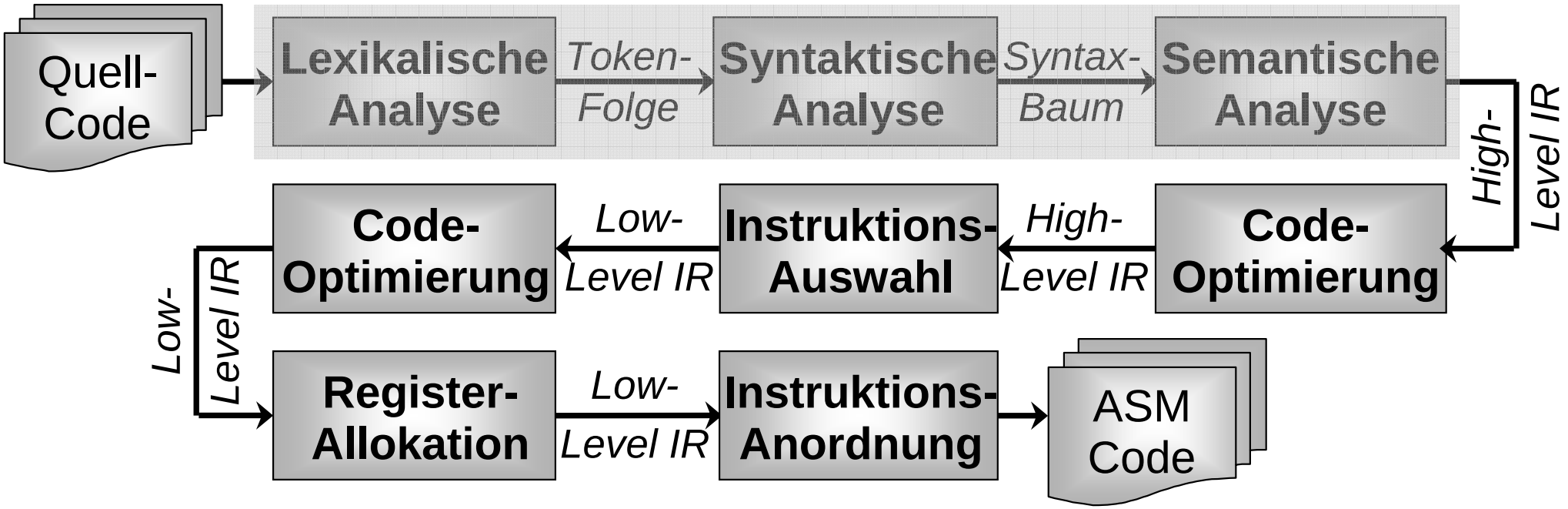
- **Retargierbarkeit:**
  - Anpaßbarkeit auf verschiedenste Prozessoren (z.B. DSPs, VLIWs, NPUs, ...)
  - ☞ Modellierung verschiedenster Befehlssätze
  - ☞ Modellierung verschiedenster Registersätze
- **Enthaltene Analysen:**
  - Datenflußanalysen
  - Kontrollflußanalysen
- **Schnittstellen:**
  - Einlesen und Ausgabe von Assembler-Dateien
  - Schnittstelle zur Code-Selektion

Zurück zur Preisfrage...

Wo sollten Compiler-Optimierungen angesiedelt sein?

- **Optimierungen finden (i.d.R.) auf IR-Ebene im Compiler statt**

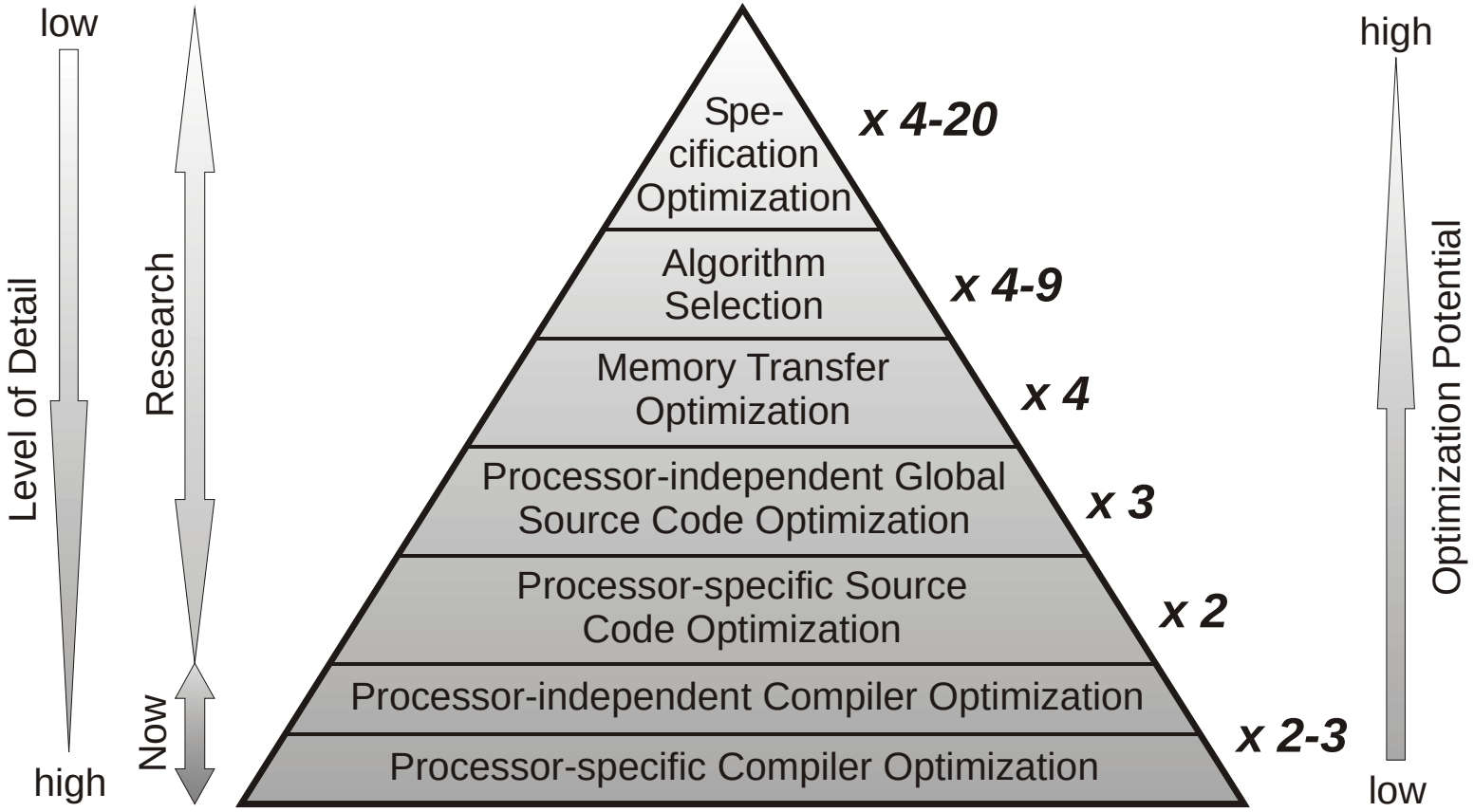
☞ ***Struktur eines Optimierenden Compilers mit 2 IRs:***



Gliederung der Vorlesung

- Kapitel 1: Compiler für Eingebettete Systeme
- **Kapitel 2: Interner Aufbau von Compilern**
  - Compilerphasen
  - Interne Zwischendarstellungen
  - Optimierungen & Zielfunktionen
- Kapitel 3: Prepass-Optimierungen
- Kapitel 4: HIR Optimierungen und Transformationen
- Kapitel 5: Instruktionsauswahl
- Kapitel 6: LIR Optimierungen und Transformationen
- Kapitel 7: Register-Allokation
- Kapitel 8: Compiler zur $WCET_{EST}$-Minimierung
- Kapitel 9: Ausblick

Abstraktionsebenen v. Optimierungen



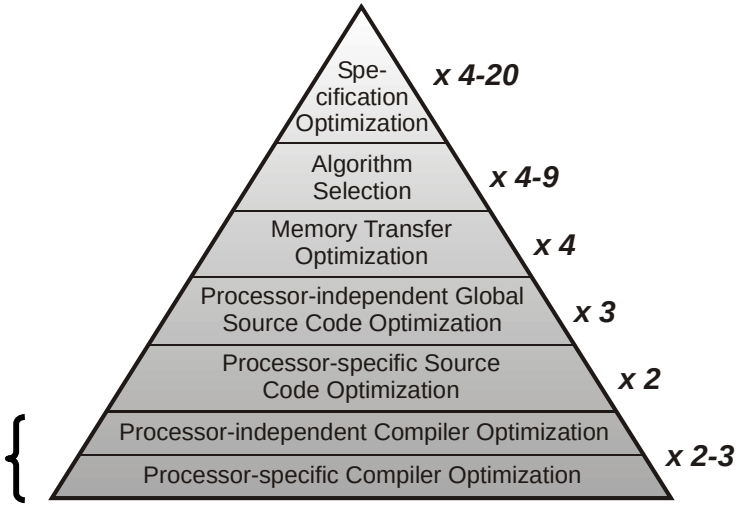
*[**H. Falk,** Source Code Optimization Techniques for Data Flow Dominated Embedded Software, Kluwer, 2004]*

Abstraktionsebenen v. Optimierungen

Compiler-Optimierungen:

- Alles, was in heutigen Compilern enthalten ist
- Prozessor-spezifisch: low-level
- Prozessor-unabhängig: high-level
- Typische Speed-Ups: insgesamt um Faktor 2 bis 3

☞ *Mehr dazu in Kapiteln 4 - 8*

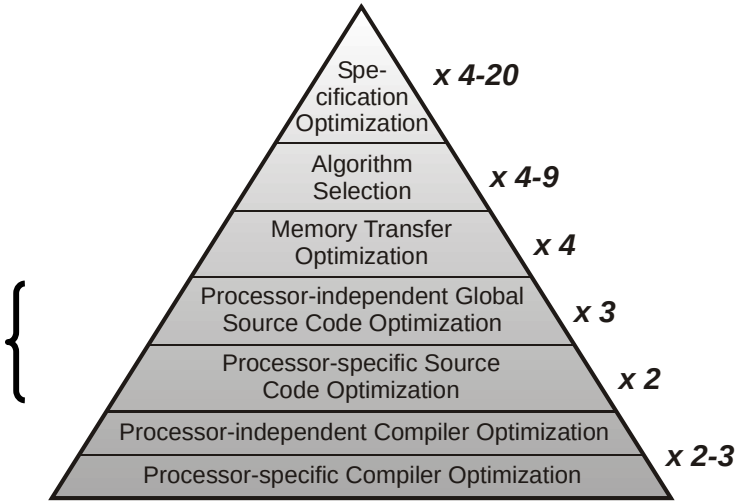


Abstraktionsebenen v. Optimierungen

Quellcode-Optimierung:

- Quellcode-Umschreiben, so daß Compiler effizienteren Code erzeugt
- Prozessor-spezifisch: Unterstützung des Compilers bei Abbildung von Quell- auf Maschinensprache
- Prozessor-unabhängig: Maschinen-unabhängiges Verbessern der Quellcode-Struktur
- Teils automatisch, teils manuell
- Typische Speed-Ups: je x2 oder x3

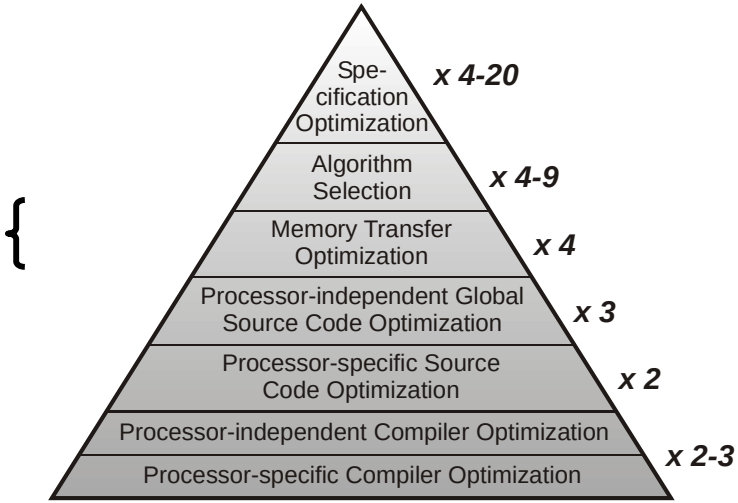
☞ *Mehr dazu in Kapitel 3*



Abstraktionsebenen v. Optimierungen

Speichertransfer-Minimierung:

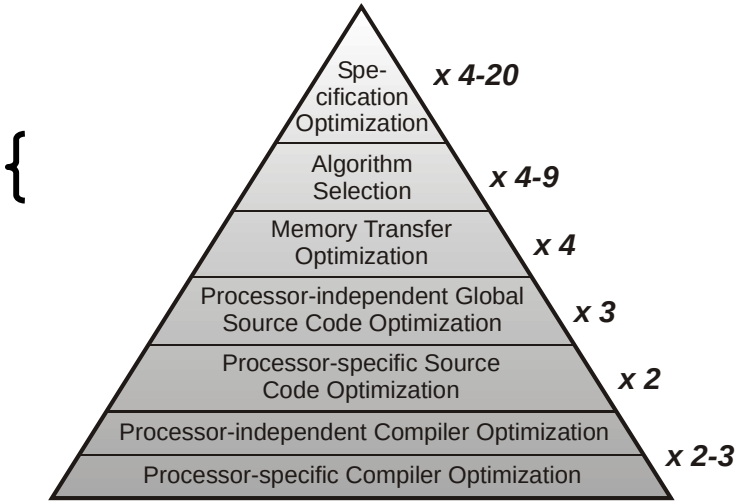
- Reduktion von Daten- & Code-Transfers vom Speicher zum Prozessor auf sehr abstraktem Niveau
- z.B. Umordnen der Datenstrukturen einer Applikation, Umordnung von (mehrdimensionalen) Feldinhalten im Speicher, Zusammenfassen & Teilen von Feldern
- Ausschließlich manuell
- Typische Speed-Ups: ca. Faktor 4



Abstraktionsebenen v. Optimierungen

Algorithmen-Auswahl:

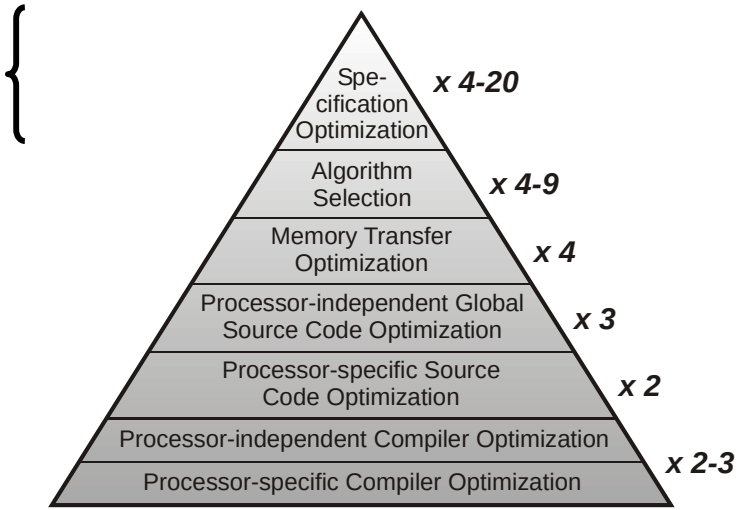
- Ersetzung ganzer Algorithmen einer Applikation durch andere, effizientere Implementierungen
- z.B. Bubblesort → Quicksort
- Ersetzung muß funktionales Verhalten der Applikation beibehalten
- Ausschließlich manuell
- Typische Speed-Ups: Faktor 4 – 9



Abstraktionsebenen v. Optimierungen

Optimierung der Spezifikation:

- Ersetzung von Algorithmen wie bei “Algorithm Selection”
- *Aber:* Ersetzung darf funktionales Verhalten der Applikation ändern
- z.B. Ersetzung von **`double`** Gleitkommazahlen durch einfache Genauigkeit oder **`integer`**; Ersetzung komplexer Formeln durch einfachere Approximationen (**`sin`**, **`cos`**)
- Ausschließlich manuell
- Typische Speed-Ups: Faktor 4 – 20



Zielfunktion: (Typische) Laufzeit

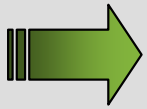
- **Durchschnittliche Laufzeit, Average-Case Execution Time (ACET)**
  Ein ACET-optimiertes Programm soll bei einer “typischen” Ausführung (d.h. mit “typischen” Eingabedaten) schneller ablaufen.
- ***Die* Zielfunktion optimierender Compiler schlechthin.**
  Strategie: “Greedy”, d.h. wo die Ausführung von Code zur Laufzeit eingespart werden kann, wird dies i.d.R. auch getan.
- **ACET-optimierende Compiler haben meist kein präzises Modell der ACET.**
  Exakte Auswirkung von Optimierungen auf die effektive Laufzeit ist dem Compiler unbekannt.

☞ **ACET-Optimierungen sind meist vorteilhaft, manchmal aber auch bloß neutral oder sogar nachteilig.**

Beispiel: Function Inlining

```
main() {
  ...
  a = min( b, c );
  ...
  ...min( f, g )...
}
```

```
int min( int i,
         int j ) {
  return(
    i<j ? i : j );
}
```



```
main() {
  ...
  a = b<c ? b : c;
  ...
  ...f<g ? f : g;
}
```

Potentielle Laufzeit-Reduktion wegen:

- Wegfallenden Codes für Parameter- / Rückgabewert-Übergabe
- Wegfallenden Codes zum Sprung in die aufgerufene Funktion
- Evtl. wegfallender Speicherplatz-Allokation zu Beginn der aufgerufenen Funktion
- Evtl. Ermöglichung anderer Optimierungen, die sonst an Funktionsgrenzen scheitern

Zielfunktion: Codegröße

- **Erzeugung von minimal wenig Code, in Bytes gemessen**
- **Einfache Modellbildung:**
  Compiler weiß, welche Instruktionen er generiert, und wieviele Bytes jede einzelne Instruktion benötigt.

- **Oft Zielkonflikt mit Laufzeit-Minimierung:** *Bsp. Inlining*
  - Kopieren des Funktionsrumpfes an Stelle des Funktionsaufrufs
  - Bei großen Funktionen und/oder vielen Vorkommen von Aufrufen im Code: starke Erhöhung der Codegröße!
  - Codegrößen-minimierende optimierende Compiler:

  ☞ *Komplett deaktiviertes Function Inlining*

Zielfunktion: Energieverbrauch

- **Generierung von Code mit minimalem Energieverbrauch**
- **Modellbildung umfasst i.d.R. Prozessor *und* Speicher**

- **Einfaches Energiemodell für Prozessoren:**
  - *Basiskosten* eines Befehls: Energieverbrauch des Prozessors bei Ausführung nur dieses einen Befehls
  - Ermittlung der Basiskosten *(z.B. für* **ADD***-Befehl)*:

```
.L0:
  ...
  ADD d0, d1, d2;
  ADD d0, d1, d2;
  ADD d0, d1, d2;
  ...
  JMP .L0;
```

  - Schleife, die zu untersuchenden Befehl ***sehr oft*** enthält.
  - Ausführung auf realer Hardware
  - Energiemessung: Ampèremeter
  - Ergebnis herunterrechnen auf einmal **ADD**
  - Wiederholen für kompletten Befehlssatz

Zielfunktion: Energieverbrauch

- **Einfaches Energiemodell für Prozessoren:**
  - *Inter-Instruktionskosten* zwischen zwei nachfolgenden Befehlen: Modellieren Aktivierung und Deaktivierung Funktionaler Einheiten *(FUs)*
  - Beispiel: **`ADD`** wird in ALU ausgeführt, **`MUL`** in Multiplizierer

```
.L0:
  ADD d0, d1, d2;
  MUL d3, d4, d5;
  ADD d0, d1, d2;
  MUL d3, d4, d5;
  ...
  JMP .L0;
```

  - Schleife, die zu untersuchendes Befehlspaar ***sehr oft*** enthält.
  - Ausführung & Messung wie bei Basiskosten
  - Ergebnis herunterrechnen auf ein Befehlspaar **`ADD`** und **`MUL`**
  - Wiederholen für alle Kombinationen von FUs

Zielfunktion: Energieverbrauch

Berechnung der Prozessor-Energie durch Compiler:

- Summiere Basiskkosten aller generierten Instruktionen
- Summiere Inter-Instruktionskosten benachbarter Befehlspaare

*[**V. Tiwari et al.,** Power Analysis of Embedded Software: A First Step Towards Software Power Minimization, IEEE Transactions on VLSI, Dezember 1994]*

Berechnung der Speicher-Energie durch Compiler:

- Entweder anhand von Datenblättern oder durch Messungen
- Grundlage: Energieverbrauch pro Lade- und Speichervorgang
- Einfach für Statische RAMs *(SRAM)*, schwer für Caches und Dynamische RAMs *(DRAM)*

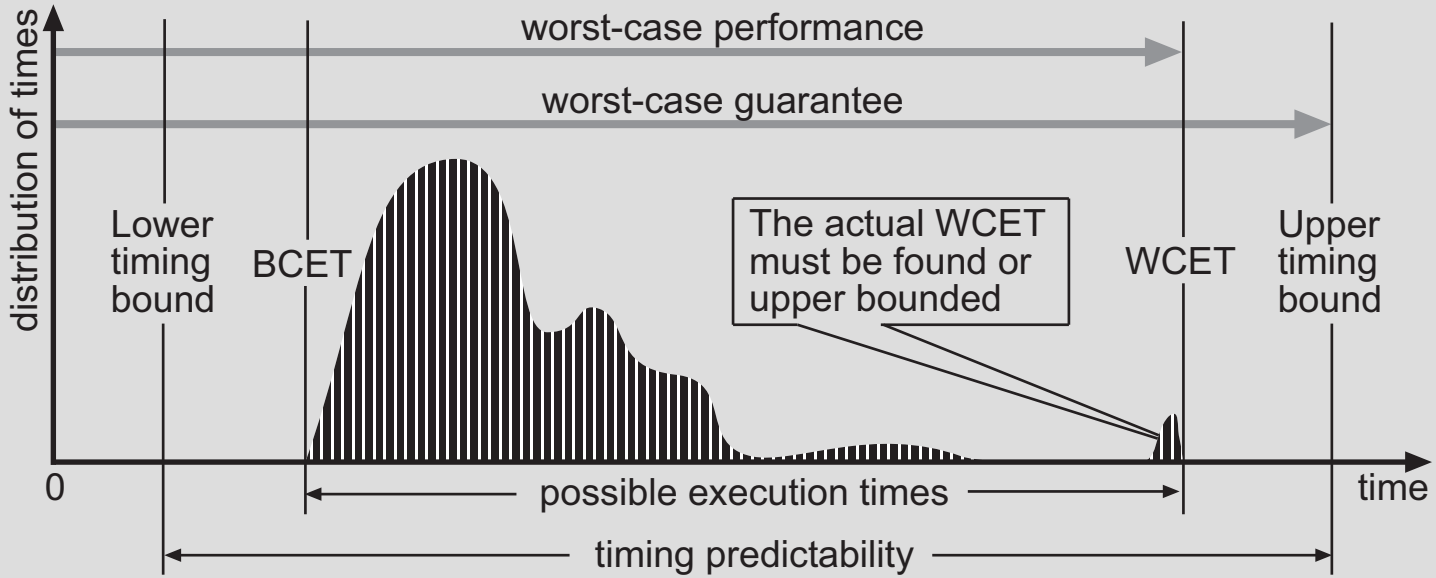
*[**S. Steinke et al.,** An Accurate and Fine Grain Instruction-Level Energy Model Supporting Software Optimizations, PATMOS Workshop, September 2001]*

Zielfunktion: Worst-Case Laufzeit

- **Worst-Case Execution Time *(WCET)*:**
  Obere Schranke für die Laufzeit eines Programmes über alle denkbaren Eingabedaten hinweg.
- **Problem:**
  Ermittlung der WCET eines Programmes nicht berechenbar!
  (Würde Lösen des Halte-Problems beinhalten)

Zielfunktion: Worst-Case Laufzeit

- **Lösung:** Schätzung oberer Grenzen für die echte (unbekannte) WCET



- **Anforderungen an WCET-Abschätzungen:**
  - Sicherheit *(safeness)*: $WCET \leq WCET_{EST}$!
  - Präzision *(tightness)*: $WCET_{EST} - WCET \rightarrow$ minimal

Literatur

- **Compilerphasen und IRs:**
  - Steven S. Muchnick, *Advanced Compiler Design & Implementation*, Morgan Kaufmann, 1997.
    ISBN 1-55860-320-4
  - Andrew W. Appel, *Modern compiler implementation in C*, Cambridge University Press, 1998.
    ISBN 0-521-58390-X
- **Abstraktionsebenen von Optimierungen**
  - H. Falk, *Source Code Optimization Techniques for Data Flow Dominated Embedded Software*, Kluwer Academic Publishers, 2004.
    ISBN 1-4020-2822-9

Zusammenfassung

- Bedeutung einzelner Phasen eines Compilers
- Anordnung von Optimierungen innerhalb des Compilers
- IRs verschiedener Abstraktionsniveaus, konkrete Beispiele
- Abstraktionsniveaus von Code-Optimierungen
- Zielfunktionen: ACET, Codegröße, Energieverbrauch, WCET