Compiler für Eingebettete Systeme (CfES)
- Vorschau -

Sommersemester 2009

Dr. Heiko Falk

Technische Universität Dortmund

Lehrstuhl Informatik 12

Entwurfsautomatisierung für Eingebettete Systeme

Organisatorisches (1)

- **Vorlesung (3V)**
  - Montags, 14.15 – 15.45 Uhr
    OH14 / E23 (Hörsaal)
    Freitags, 09.15 – 10.00 Uhr
    OH14 / E23 (Hörsaal)
- **Übungen (1Ü)**
  - Blockübung, 3 Termine à 3:45 Zeitstunden
  - Praktische Implementierung eines (einfachen) ANSI-C Compilers am Rechner
  - ☞ *Bitte zum 1. Vorlesungstermin (Freitag 17. April) erscheinen!*
- **Materialien**
  - Foliensätze online:
    `http://ls12-www.cs.tu-dortmund.de/teaching/lectures/ss09/cfes`
  - Bücher & Originalartikel

Organisatorisches (2)

- **Zuhörer**
  - Diplom-Studierende Informatik / Angewandte Informatik im Hauptstudium, Schwerpunktgebiet 2 (“Rechnerarchitektur, eingebettete Systeme und Simulation”)
  - Master-Studierende Informatik / Angewandte Informatik
- **Wünschenswerte Voraussetzungen**
  - Eingebettete Systeme
  - Übersetzerbau
- **Prüfungen**
  - Mündliche Prüfung (30 min, 6 Leistungspunkte)
  - Erfolgreiche Übungsteilnahme für unbenoteten Schein

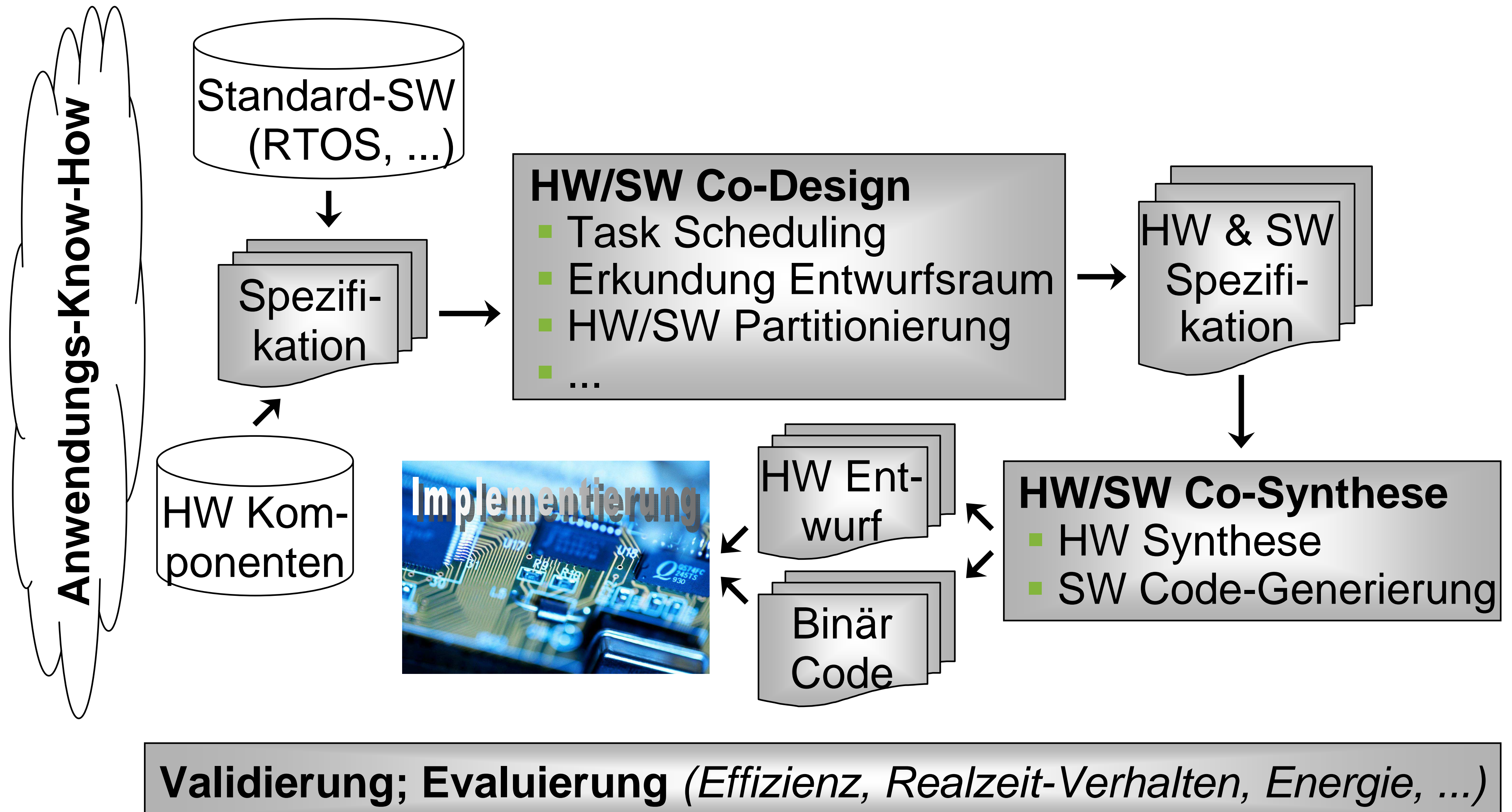
Marketing

- **CfES WS07/08**
  - Gewinner des Lehrer-Lämpel-Pokals
  - 2. Platz der offiziellen Lehre-Evaluation der Fakultät

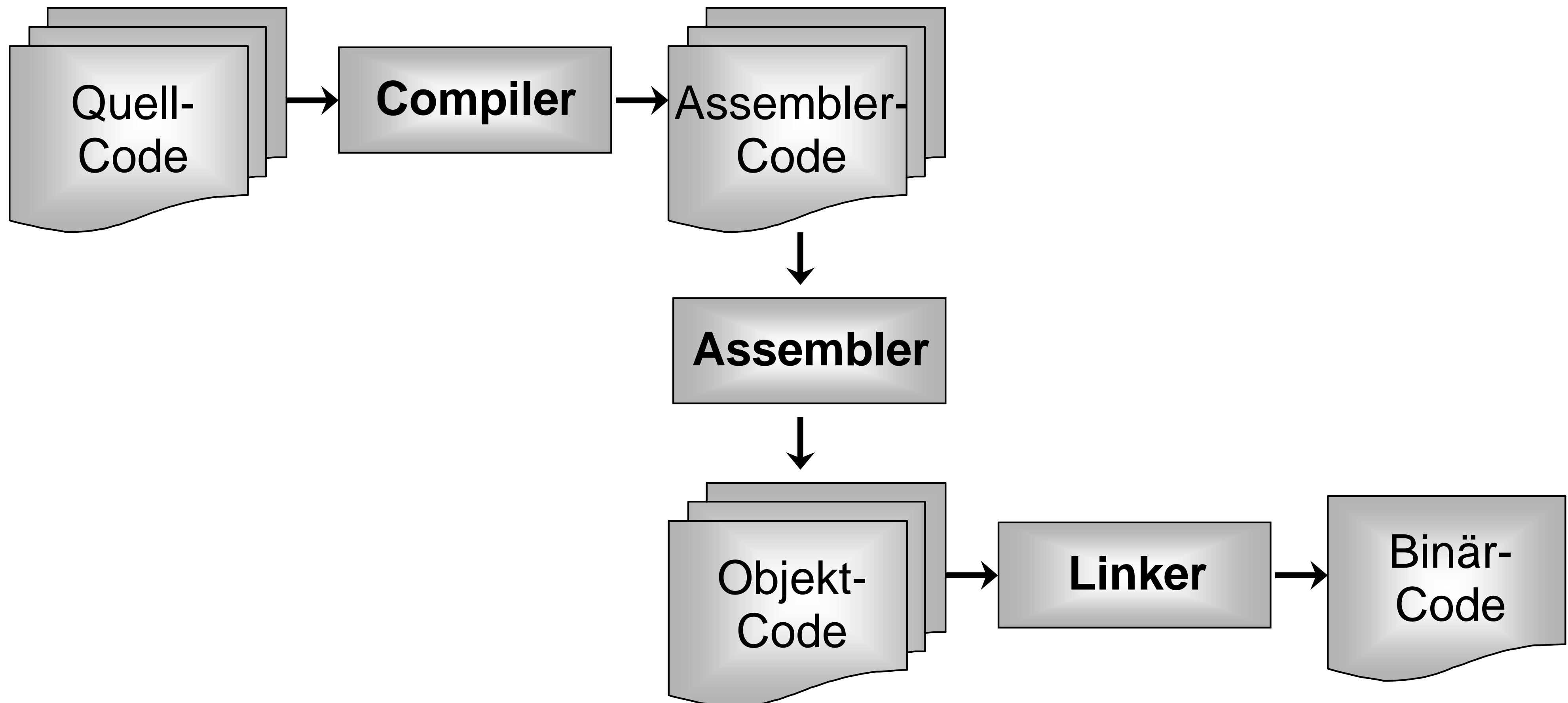
Gliederung der Vorlesung

- Kapitel 1: Compiler für Eingebettete Systeme
- Kapitel 2: Interner Aufbau von Compilern
- Kapitel 3: Prepass-Optimierungen
- Kapitel 4: HIR Optimierungen und Transformationen
- Kapitel 5: Code-Selektion
- Kapitel 6: LIR Optimierungen und Transformationen
- Kapitel 7: Register-Allokation
- Kapitel 8: Ausblick

Entwicklungsprozeß Eingebetteter Systeme



Werkzeuge zur Code-Generierung



Quellsprache ANSI-C (1)

- **Rein imperativ**
  - Keine Objektorientierung: keine Klassen, keine Objekte
  - C-Programm: Menge von Funktionen
  - Funktion `main`: Ausgezeichnete Startfunktion
  - Funktionen: Folge von Befehlen, sequenzielle Abarbeitung

```
int filtep( int rlt1, int al1, intrlt2, int al2 )
{
  long pl, pl2;
  pl = 2 * rlt1;
  pl = (long) al1 * pl;
  pl2 = 2 * rlt2;
  pl += (long) al2 * pl2;
  return( (int)(pl >> 15) );
}
```

ANSI-C: Diskussion

- **Vorteile**

  - Standardisierte Hochsprache, weite Verbreitung
  - Viele existierende Tools zur Code-Generierung
  - Viel bereits existierender Quellcode (open source & proprietär)
  - Trotz Hochsprache: Low-level Programmierung möglich
  - Maschinenähe
  - Aufwand für Compilerentwurf noch akzeptabel
- **Nachteile**

  - Maschinennähe, Mangelnde Portabilität von Quellcodes
  - Programmierer-verantwortliche Speicherverwaltung fehleranfällig
  - Keinerlei Objektorientierung

Anforderungen an Compiler für ES

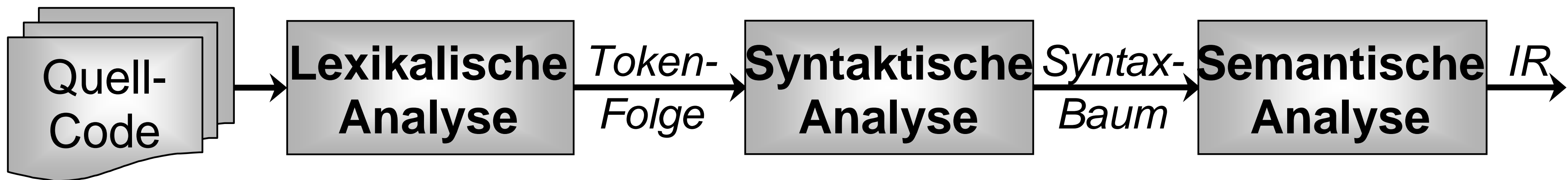
- **Maximale Code-Qualität**
  - Laufzeit-Effizienz
  - Geringer Energieverbrauch
  - Geringe Codegröße
  - Maximale Parallelisierung
  - Echtzeitfähigkeit
  - ...

- **Sinnvolle Maßnahmen**
  - Bestmögliche Abbildung der Quell- auf die Zielsprache
  - Präsenz starker Compiler-Optimierungen
  - Wiederverwendung von Code-Fragmenten
  - Maximale Nutzung schneller und kleiner Speicher
  - Einbeziehung der WCET *(Worst-Case Execution Time)*
  - ...

Nebensächliche Anforderung

- **Geschwindigkeit des Compilers**
  - Situation bei Desktop-Rechnern:
    - ✓ Großer Umfang verfügbarer Ressourcen
    - ✓ Code-Qualität von geringerem Interesse
    - ✓ Compiler sollen schnell korrekten Code generieren
  - Situation bei Eingebetteten Systemen:
    - ❑ Code-Qualität von maximalem Interesse
    - ❑ Compiler sollen hoch-optimierten Code generieren
    - ❑ Compiler werden im ES-Entwicklungsprozeß seltener aufgerufen als bei Desktop-Rechnern

Aufbau von Compilern: Das Frontend



- **Lexikalische Analyse:** Zerlegung des Quellcodes in Token der Quellsprache (z.B. Bezeichner, Konstanten, Schlüsselworte, ...)
- **Syntaktische Analyse:** Entscheidung, ob gegebene Tokenfolge aus Grammatik der Quellsprache ableitbar ist
- **Semantische Analyse:** Typanalyse, Gültigkeit von Symbolen (Scopes), Aufbau von Symboltabellen, Erzeugung einer *Internen Zwischendarstellung (IR)* zur weiteren Verarbeitung

Aufbau von Compilern: Das Backend

- **Instruktionsauswahl:** Auswahl von Maschinenbefehlen zur Implementierung der IR; Generierung von nicht ausführbarem Code, der Pseudoregister *(virtuelle Register)* statt physikalischer Prozessor-Register nutzt
- **Registerallokation:** Abbildung virtueller auf physikalische Register, Einfügen von Speicher-Transfers *(Spill-Code)*, falls zu wenige physikalische Register vorhanden
- **Instruktionsanordnung:** Umordnen von Maschinenbefehlen zur Erhöhung der Parallelität des Codes

Struktur eines hoch-optimierenden Compilers

Quell-Code → **Lexikalische Analyse** —*Token-Folge*→ **Syntaktische Analyse** —*Syntax-Baum*→ **Semantische Analyse** —*High-Level IR*→ **Code-Optimierung** —*High-Level IR*→ **Instruktions-Auswahl** —*Low-Level IR*→ **Code-Optimierung** —*Low-Level IR*→ **Register-Allokation** —*Low-Level IR*→ **Instruktions-Anordnung** → ASM Code

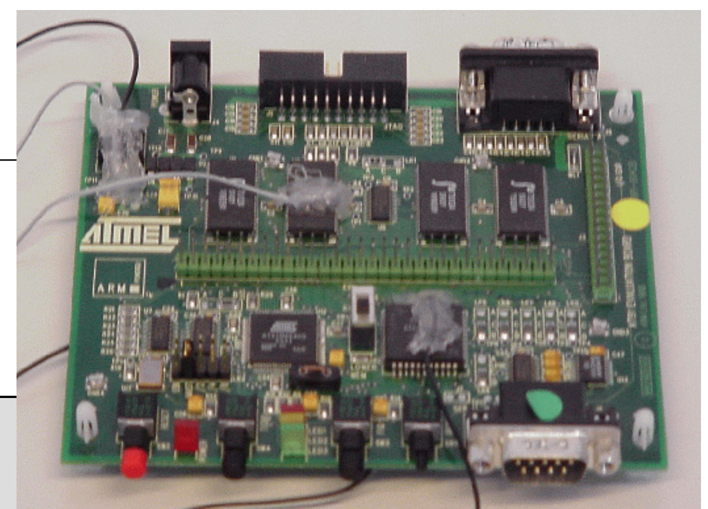
- **Vorlesung "Compiler für Eingebettete Systeme":**
  - ✓ Frontend nicht weiter betrachtet (☞ Vorlesung "Übersetzerbau")
  - ✓ *Schwerpunkt: Backend & Compiler-Optimierungen*

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.**

Zielfunktion: Energieverbrauch



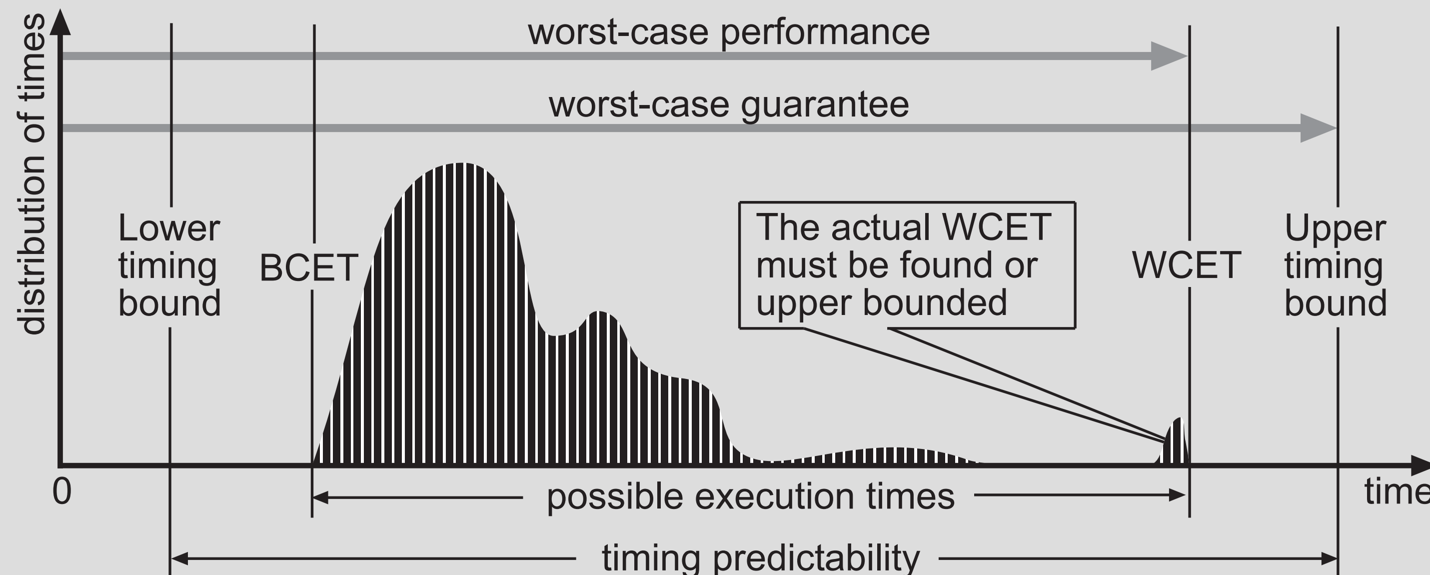
- **Generierung von Code, der zur Laufzeit minimale Energie verbraucht**
- **Einfaches Energiemodell für Prozessoren:**
  - *Basiskosten* eines Befehls: Energieverbrauch des Prozessors bei Ausführung nur dieses einen Befehls
  - Ermittlung der Basiskosten *(z.B. für* **ADD***-Befehl)*:

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

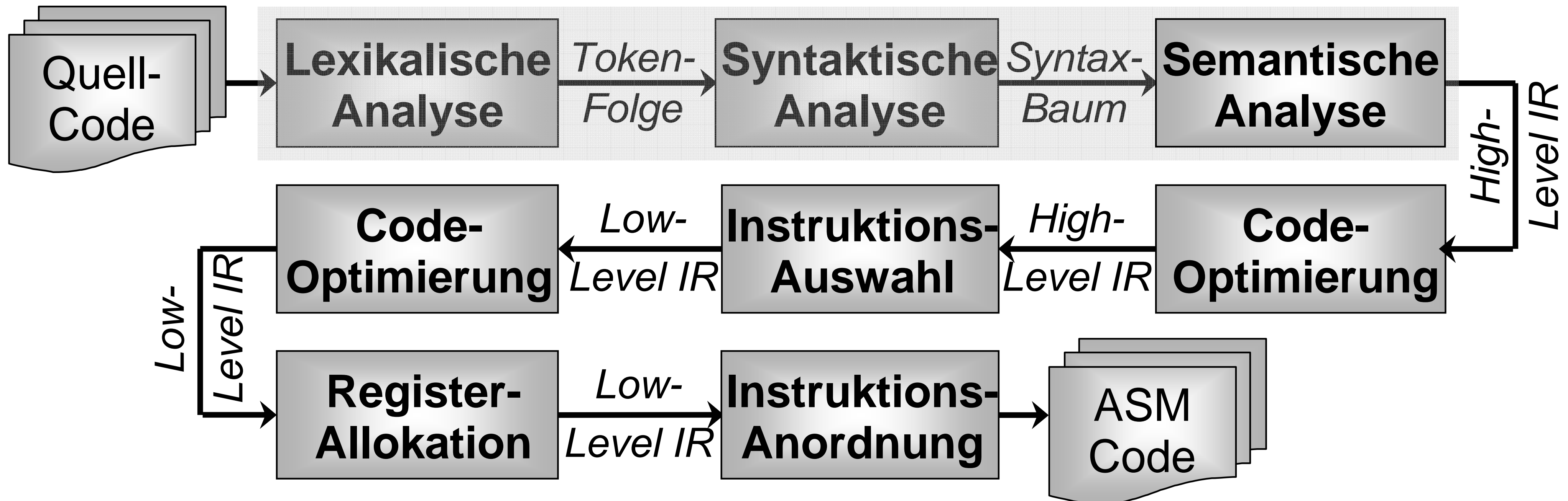
- Schleife, die zu untersuchenden Befehl ***sehr oft*** enthält.
- Ausführung auf realer Hardware
- Energiemessung: Ampèremeter
- Ergebnis herunterrechnen auf einmal **ADD**
- Wiederholen für kompletten Befehlssatz

Zielfunktion: Worst-Case Laufzeit

- **Worst-Case Execution Time *(WCET)*:** Obere Schranke für die Laufzeit eines Programms über alle denkbaren Eingaben hinweg.
- **Problem:** WCET eines Programmes nicht berechenbar! (Reduzierbar auf Halte-Problem)
- **Lösung:** Schätzung oberer Grenzen für die echte (unbekannte) WCET

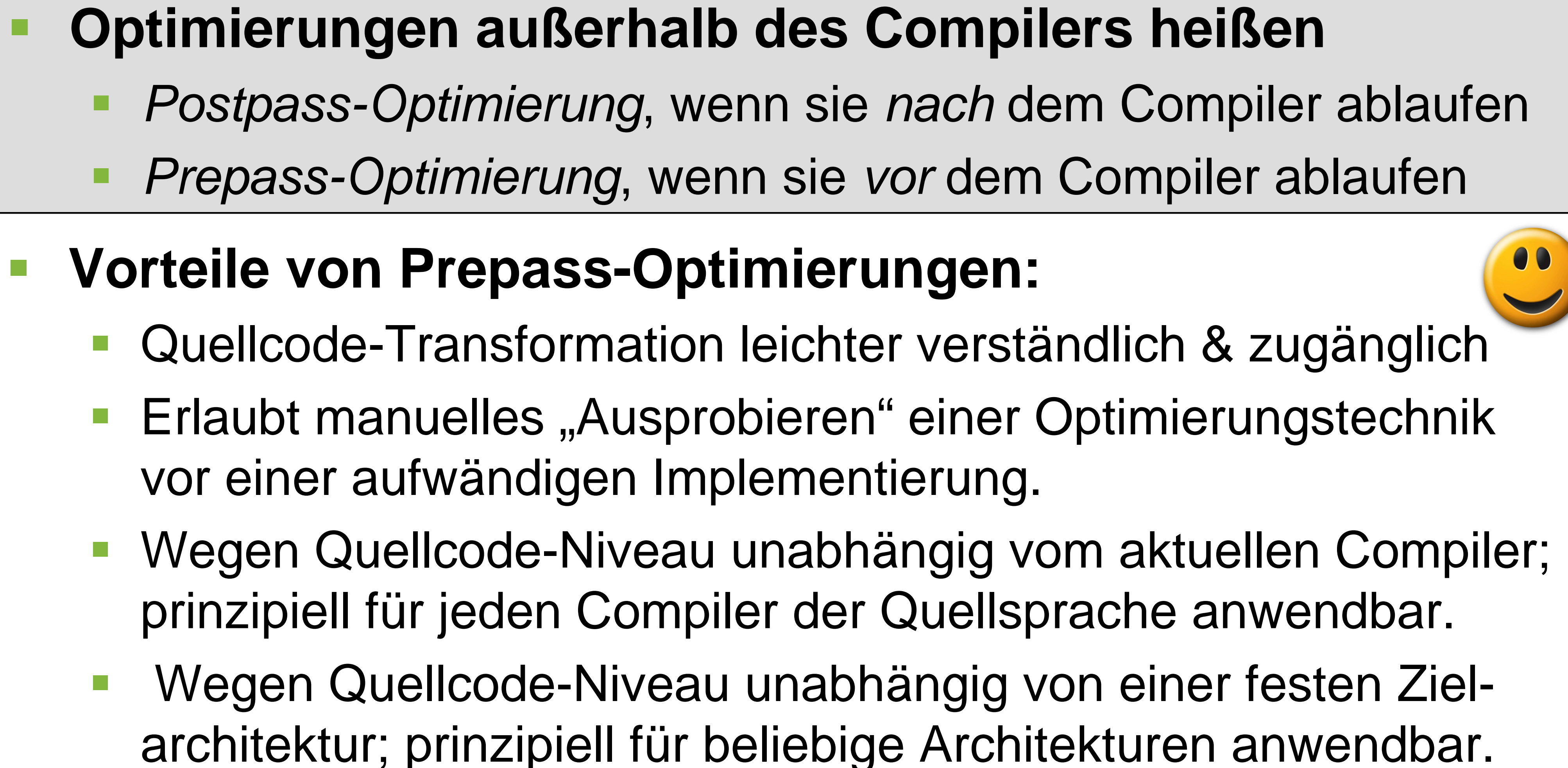


Motivation von Prepass-Optimierungen



Frage: Darf nur der Compiler Code optimieren?

Motivation von Prepass-Optimierungen



- **Optimierungen außerhalb des Compilers heißen**
  - *Postpass-Optimierung*, wenn sie *nach* dem Compiler ablaufen
  - *Prepass-Optimierung*, wenn sie *vor* dem Compiler ablaufen
- **Vorteile von Prepass-Optimierungen:**
  - Quellcode-Transformation leichter verständlich & zugänglich
  - Erlaubt manuelles „Ausprobieren" einer Optimierungstechnik vor einer aufwändigen Implementierung.
  - Wegen Quellcode-Niveau unabhängig vom aktuellen Compiler; prinzipiell für jeden Compiler der Quellsprache anwendbar.
  - Wegen Quellcode-Niveau unabhängig von einer festen Ziel-architektur; prinzipiell für beliebige Architekturen anwendbar.

☞ **Vorlesung:** Prepass-Optimierung „Loop Nest Splitting"

Motivation von HIR-Optimierungen

- **High-Level IRs *(HIR)*:**
  - Sehr eng an Quellsprache angelehnt
  - High-Level Sprachkonstrukte (insbes. Schleifen, Funktionsaufrufe mit Parameterübergabe, Array-Zugriffe) bleiben erhalten.

- **High-Level Optimierungen:**
  - Nutzen Features der HIR stark aus.
  - Konzentrieren sich auf starkes Umstrukturieren von Schleifen und Funktionsstruktur.
  - Sind auf niedriger Ebene nur schwer zu realisieren, da high-level Informationen erst wieder rekonstruiert werden müssten.

HIR-Optimierungen in CfES

- **Function Specialization (alias *Procedure Cloning*):**
  - Erzeugen von Kopien von Funktionen mit weniger Parametern
  - Aufwand f. Parameterübergabe reduziert, Ermöglichen weiterer Standard-Optimierungen
  - Einfluß von Cloning auf ACET, WCET & Codegrösse

- **Parallelisierung für Homogene Multi-DSPs:**
  - Parallel arbeitende DSPs hoch-performant, aber de facto keine parallelisierenden Compiler für Multi-DSPs verfügbar
  - Systematische Umstrukturierung von Schleifen, Arrays und Array-Zugriffen, um Parallelisierung zu erreichen
  - *Verblüffendes Ergebnis:* Parallelisierung für 4 DSPs liefert Speed-Up von mehr als Faktor 4!

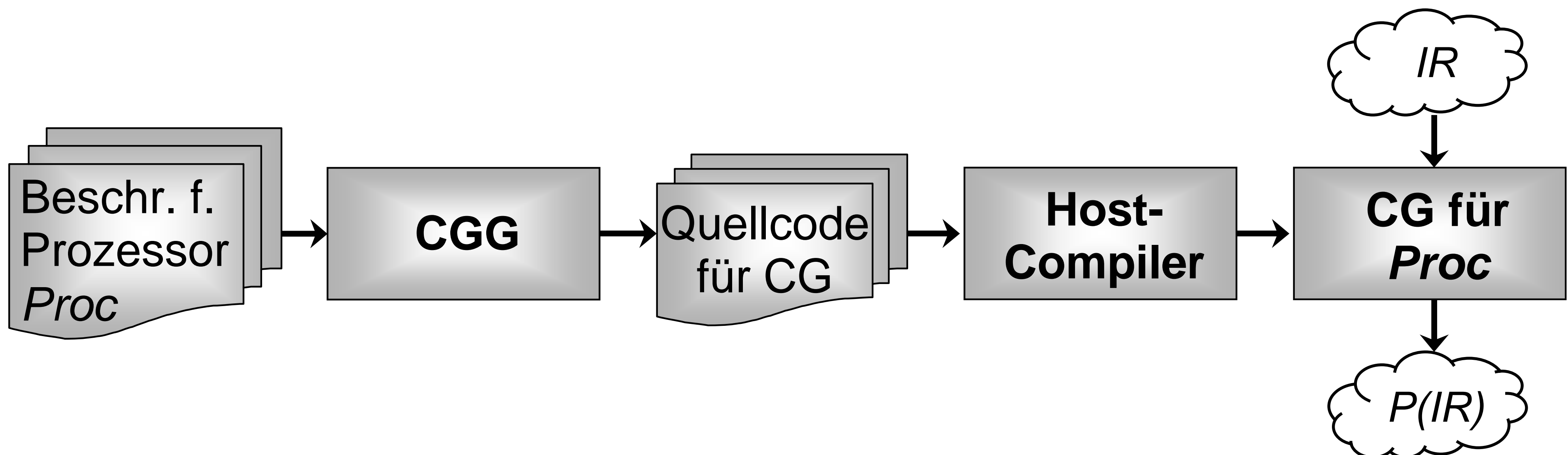
Instruktionsauswahl

- **Ziel und Aufgabe:**
  - Auch *Code-Selektion* oder *Code-Generierung* genannt.
  - “Herz” des Compilers, das eigentliche Übersetzung von Quell- in Zielsprache vornimmt.
  - Überdeckung der Knoten aller Datenflussgraphen (DFGs) der IR mit semantisch äquivalenten Operationen der Zielsprache.

- **Implementierung eines Code-Generators:**
  - Nicht-triviale, stark vom Ziel-Prozessor abhängige Aufgabe
  - Per-Hand-Implementierung eines Code-Generators bei Komplexität heutiger Prozessoren nicht mehr vertretbar
  - Statt dessen: Verwendung sog. *Code-Generator-Generatoren*

Code-Generator-Generatoren

- Sog. *Meta-Programme*, d.h. Programme, die Programme als Ausgabe erzeugen.
- Ein Code-Generator-Generator *(CGG)* erhält eine Prozessor-Beschreibung als Eingabe und erzeugt daraus einen Code-Generator *(CG)* für eben diesen Prozessor.



Instruktionsauswahl in CfES

- **Vorlesung:**
  - Baum-basierte Code-Selektion optimal & effizient lösbar
  - Standard-Verfahren: Tree Pattern Matching *(TPM)*
  - TPM-Algorithmus (dyn. Programmierung) & Komplexität
  - Prozessor-Beschreibung per Baum-Grammatik: Regeln, Cost- und Action-Teile
  - Code-Generator-Generator ***icd-cg***

- **Übung:**
  - Realisierung einer einfachen Baum-Grammatik zur Übersetzung von ANSI-C nach Assembler (Infineon TriCore)
  - Praktische Nutzung von ICD-C und ***icd-cg*** am Rechner

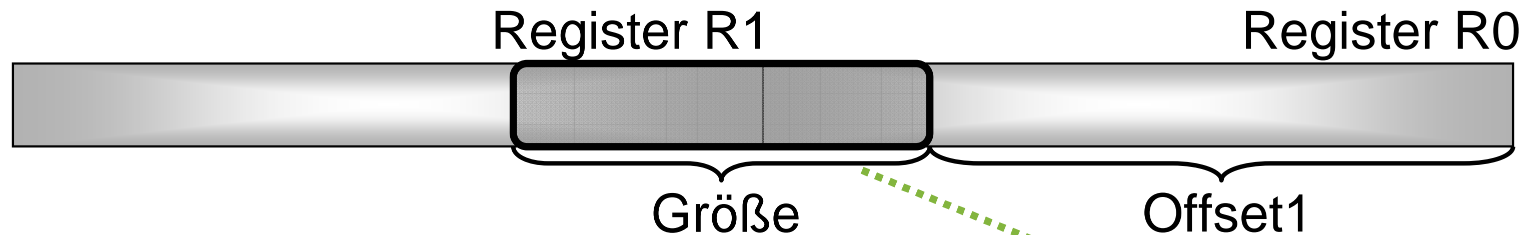
LIR-Optimierung: Erzeugung von Bit-Paketen

- **Bit-Pakete in Protokoll-Headern:**
  - Header zerfallen in Bereiche unterschiedlicher Bedeutung
  - Bit-Bereiche oft nicht nach Prozessor-Wortbreiten angeordnet
  - Bit-Paket:
    - Menge aufeinanderfolgender Bits beliebiger Länge
    - an beliebiger Position startend
    - u.U. Wortgrenzen überschreitend

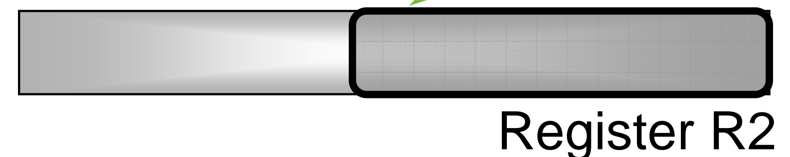
- **Netzwerk-Prozessoren *(NPUs)*:**
  - *Bit-Paket-Operationen* zum Extrahieren, Einfügen & Bearbeiten von Bit-Paketen und einzelner Bits

Operationen auf Bit-Paketen

- **Extrahieren von Bit-Paketen**

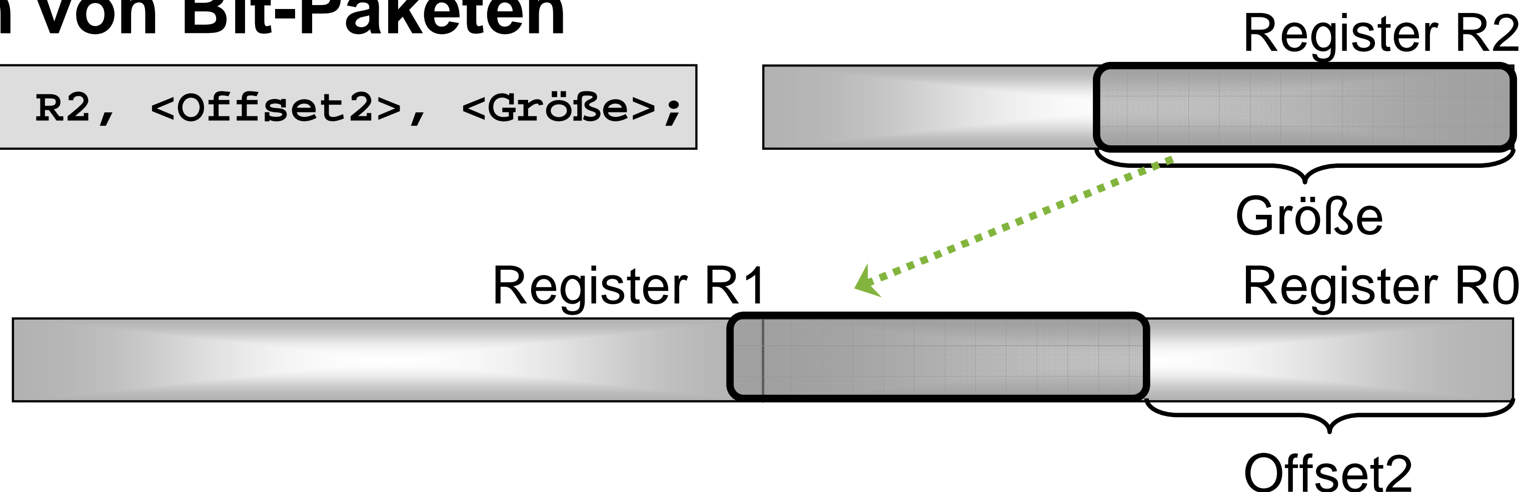



```
extr R2, R0, <Offset1>, <Größe>;
```



- **Einfügen von Bit-Paketen**

```
insert R0, R2, <Offset2>, <Größe>;
```



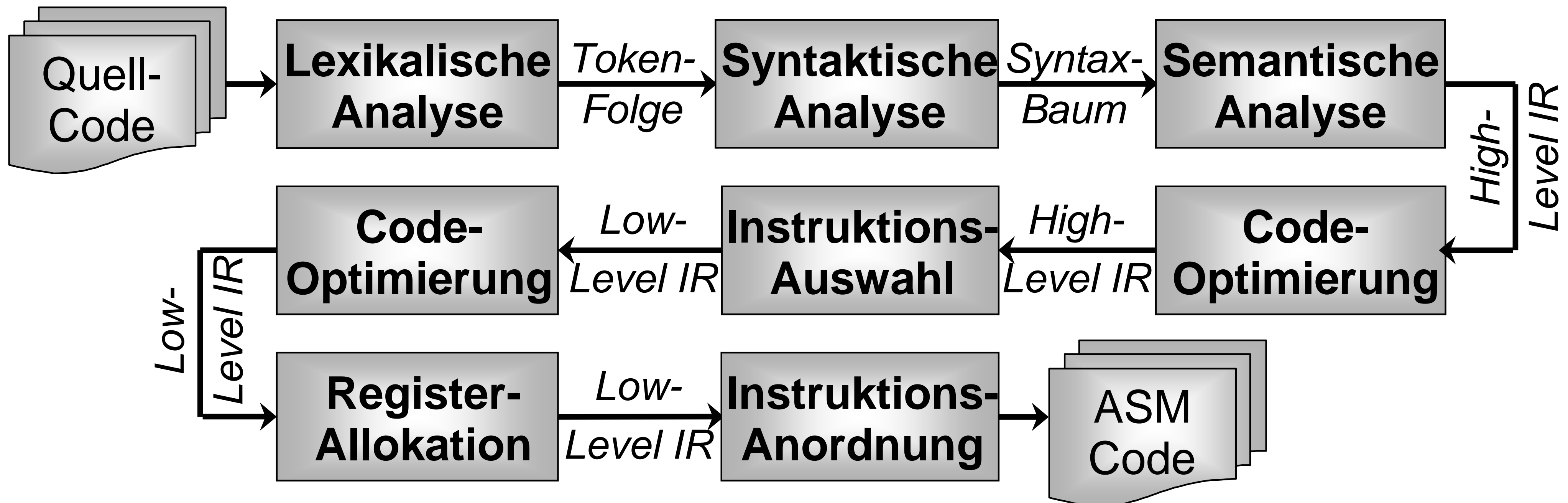
Generierung von Bit-Paket-Operationen

- **Software zur Protokoll-Verarbeitung:**
  - Hoher Code-Anteil zur Verarbeitung von Bit-Paketen
  - Typischer C-Code *(GSM-Kernel, TU Berlin)*:

```
xmc[0] = (*c >> 4) & 0x7;
xmc[1] = (*c >> 1) & 0x7;
xmc[2] = (*c++ & 0x1) << 2;
xmc[2] |= (*c >> 6) & 0x3;
xmc[3] = (*c >> 3) & 0x7;
```

- **Bitgenaue Daten- & Wertfluss-Analyse *(BDWFA)*:**
  - *Problem:* Wie kann solcher C-Code effizient auf die Bit-Paket-Operationen einer NPU abgebildet werden?
  - BDWFA erlaubt Aussagen über potenziellen Wert jedes einzelnen Bits eines Registers zu einem bestimmten Zeitpunkt
  - ☞ Erkennen von Bit-Paketen in Registern nach BDWFA möglich!

Register-Allokation



- Abbildung virtueller LIR-Register *(VREGs)* auf physikalische Prozessor-Register *(PHREGs)*
- Bestmögliche Ausnutzung der (knappen) Ressource von Prozessor-Registern

Register-Allokation durch Graph-Färbung

- **Idee:**
  - Erzeuge Graphen *G*, der für jedes VREG einen Knoten enthält
  - Färbe *G* mit *K* Farben, wobei Prozessor *K* PHREGs hat, so daß keine zwei benachbarten Knoten gleiche Farbe haben.

  ☞ Farbe k_v: welches PHREG belegt das zu *v* gehörende VREG?