Compilerbau

Wintersemester 2009 / 2010

Dr. Heiko Falk

Technische Universität Dortmund

Lehrstuhl Informatik 12

Entwurfsautomatisierung für Eingebettete Systeme

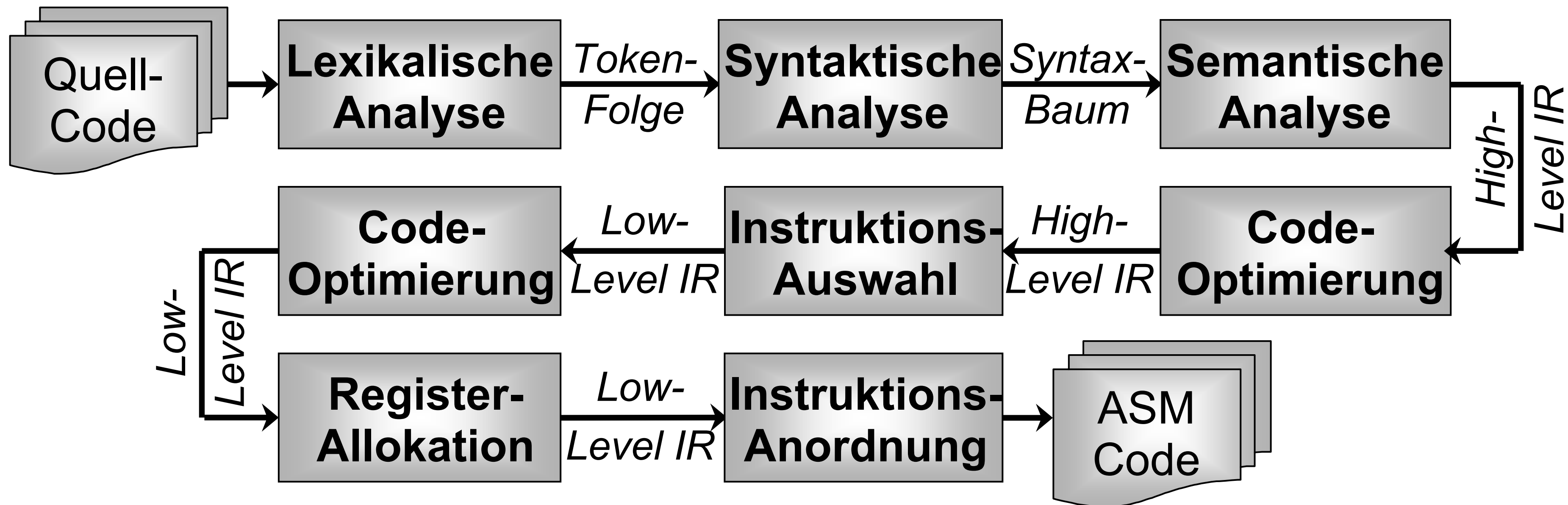
Kapitel 9

Ausblick

Gliederung der Vorlesung

- Kapitel 1: Compiler – Abhängigkeiten und Anforderungen
- Kapitel 2: Interner Aufbau von Compilern
- Kapitel 3: Lexikalische Analyse (Scanner)
- Kapitel 4: Syntaktische Analyse (Parser)
- Kapitel 5: Semantische Analyse
- Kapitel 6: Instruktionsauswahl
- Kapitel 7: Register-Allokation
- Kapitel 8: Code-Optimierung
- **Kapitel 9: Ausblick**
  - Instruktionsanordnung
  - Retargierbarkeit

Motivation

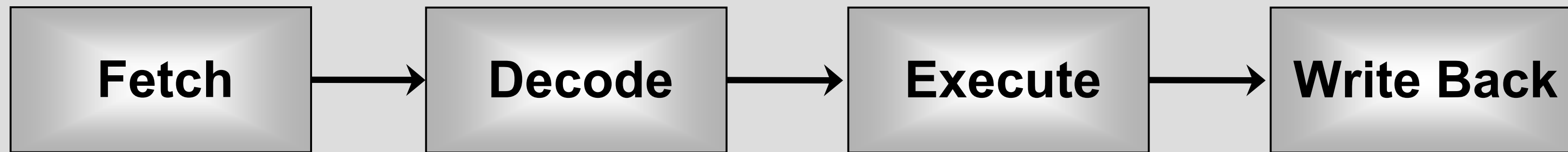


Instruktionsanordnung *(Scheduling)*:

- Umordnen von Maschinenbefehlen zur Erhöhung der Parallelität
- Einfügen von NOP-Befehlen zur Erhaltung der Code-Korrektheit
- Wegen Kürze der Zeit in Vorlesung nicht weiter behandelt

Pipeline-Verarbeitung (1)

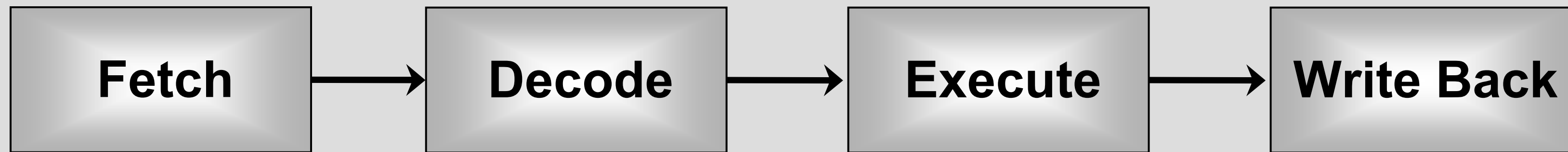
Eine triviale Prozessor-Pipeline:



- ***Fetch:*** Holt den nächsten auszuführenden Befehl aus dem Speicher.
- ***Decode:*** Dekodiert den Befehl und zerlegt diesen in Opcodes, Operanden, Adressierungsmodi, ...
- ***Execute:*** Führt einen Befehl gemäß Opcodes und Operanden aus. Das Ergebnis der Ausführungsphase wird in einem internen Zwischenregister gepuffert.
- ***Write Back:*** Schreibt den Inhalt des Puffer-Registers in das Registerfile des Prozessors zurück.

Pipeline-Verarbeitung (2)

Eine triviale Prozessor-Pipeline:



- Jede Stufe der Pipeline arbeitet i.d.R. in einem Taktzyklus
- Zu einem Zeitpunkt *t* kann die Pipeline parallel vier verschiedene Befehle enthalten:
  Befehl i_1 in Fetch-Phase, i_2: Decode, i_3: Execute, i_4: Write Back
- Im Idealfall beendet eine volle Pipeline mit jedem Taktzyklus die Abarbeitung eines Befehls
- Dieser Idealzustand kann jedoch sehr leicht gestört werden...

Pipeline-Verarbeitung (3)

Mögliche Störungen des Ablaufs einer Pipeline:

- ***Datenabhängigkeiten bei Speicherzugriffen:***
  Ein Lade-Befehl i_1 sei in Execute-Phase. Ein anderer Befehl i_2, der das Register benutzt, das i_1 aus dem Speicher lädt, sei in Decode-Phase.
  Da der Speicherzugriff von i_1 i.d.R. mehrere Takte dauert, muss die Pipeline für diese Dauer angehalten werden.
- ***Sprünge:***
  Ein Sprung wird in der Execute-Phase ausgeführt, so dass die dem Sprung folgenden Befehle bereits in Fetch und Decode enthalten sind.
  Wird der Sprung genommen, sind Fetch- und Decode-Stufen zu leeren und ab Adresse des Sprungziels erst neu zu füllen.

Anordnung von Instruktionen

Grundsätzliche Idee:

- Maschinenbefehle können in einem Programm in beliebiger Reihenfolge umgeordnet werden, solange zwischen je zwei Befehlen i_1 und i_2 ...
  - ...Datenabhängigkeiten beachtet werden.
    Wenn z.B. i_1 ein Register benutzt, das i_2 definiert, darf i_1 im Code nicht hinter i_2 angeordnet werden.
  - ...Kontrollabhängigkeiten beachtet werden.
    Wenn z.B. aufgrund des Kontrollflusses gilt, dass eine Ausführung von i_1 stets eine Ausführung von i_2 nach sich zieht, so dürfen i_1 und i_2 nicht so umgeordnet werden, dass diese Beziehung später nicht mehr gilt.

Instruktions-Scheduling im Compiler

Grundsätzliche Idee:

- Für jede Instruktion *i* kann man ein Intervall [*v*, *h*] bestimmen: um wie viele Instruktionen kann man *i* nach vorne / hinten im Code verschieben, ohne die Korrektheit des Programms zu verletzen.
- Ein Scheduler im Compiler kann diese Intervalle nun ausnutzen, um den Code so umzuordnen, dass die Prozessor-Pipeline(s) möglichst wenig gestört werden.
- ***Bsp. Datenabhängigkeiten:*** Schiebe einen weiteren Befehl i_3 zwischen Lade-Befehl i_1 und i_2.
- ***Bsp. Sprünge:*** Schiebe 1 oder 2 Befehle, von denen ein Sprung *i* nicht datenabhängig ist, hinter *i*, um die sog. *Delay-Slots* zu füllen.
- ***Bsp. TriCore:*** Ordne Befehle so an, dass alle 3 Pipelines stets laufen.

Gliederung der Vorlesung

- Kapitel 1: Compiler – Abhängigkeiten und Anforderungen
- Kapitel 2: Interner Aufbau von Compilern
- Kapitel 3: Lexikalische Analyse (Scanner)
- Kapitel 4: Syntaktische Analyse (Parser)
- Kapitel 5: Semantische Analyse
- Kapitel 6: Instruktionsauswahl
- Kapitel 7: Register-Allokation
- Kapitel 8: Code-Optimierung
- **Kapitel 9: Ausblick**
  - Instruktionsanordnung
  - Retargierbarkeit

Motivation

Bisherige Sichtweise auf einen Compiler:

- Ein Ziel-Prozessor *P* sei fest vorgegeben. Dann besteht die Aufgabe darin, einen Übersetzer von einer Quell- in die Maschinensprache von *P* zu konstruieren, mit möglichst hoher Codequalität.
- Die meisten Komponenten des Compilers brauchen Detailwissen über die Architektur von *P*. Dieses Wissen ist fest in die einzelnen Phasen / Optimierungen des Compilers einprogrammiert.
- Was, wenn man statt eines Compilers für *P* einen für *P'* braucht?
  - ☞ Alle Compiler-Komponenten für *P'* neu programmieren?

Retargierbarkeit

Retargierbarkeit:

Ist die Fähigkeit, einen Compiler an eine andere Ziel-Architektur *(target architecture)* anzupassen.

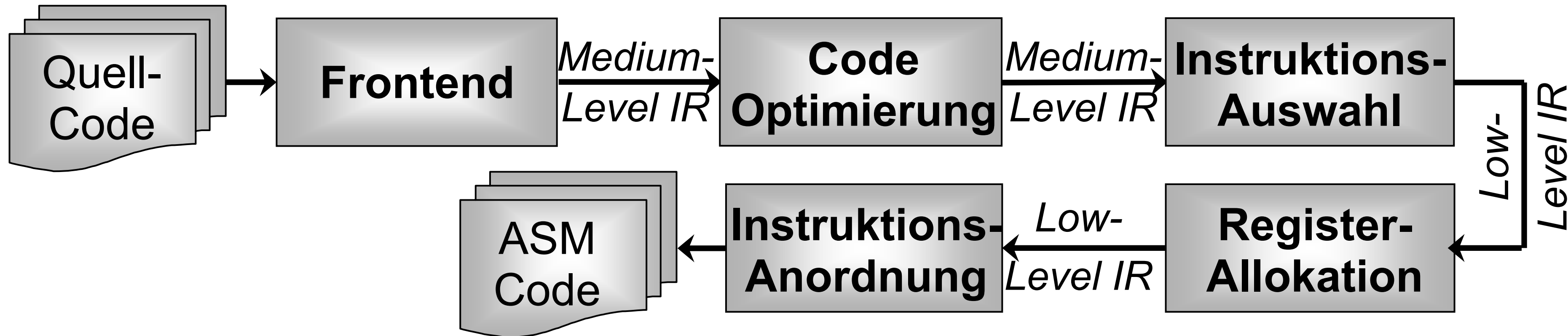
Struktur hochoptimierender Compiler:

Die bisher in dieser Vorlesung betrachtete Struktur von Compilern *(☞ siehe Folie 4)* ist schlecht retargierbar. Zum Retargieren müssen

- Code-Selektion,
- sämtliche LIR-Optimierungen,
- Register-Allokator und
- Scheduler

zu großen Teilen neu implementiert werden.

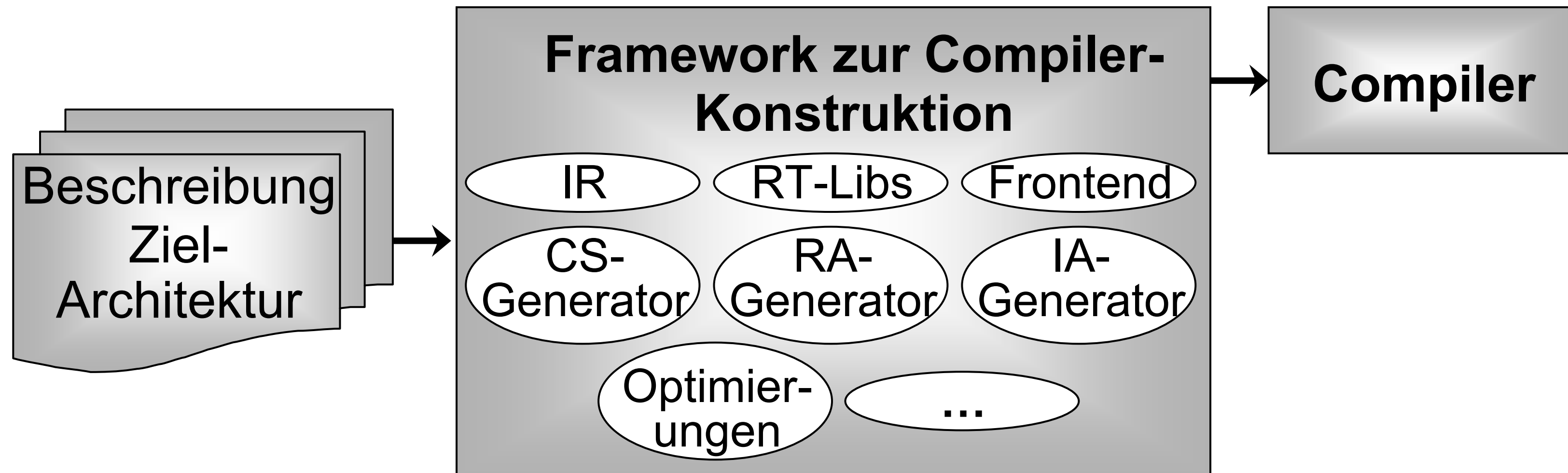
Ein besser retargierbarer Compiler



Eigenschaften:

- Statt einer HIR enthält dieser Compiler eine MIR und LIR.
- Sämtliche Code-Optimierungen finden auf MIR-Ebene statt, sind daher prozessorunabhängig und brauchen nicht retargiert werden.
- Hochgradig spezielle Optimierungen, die komplexe Prozessor-Features ausnutzen, sind nur schwer bis gar nicht zu integrieren.
- Aufwand zur Retargierung des Backends bleibt bestehen.

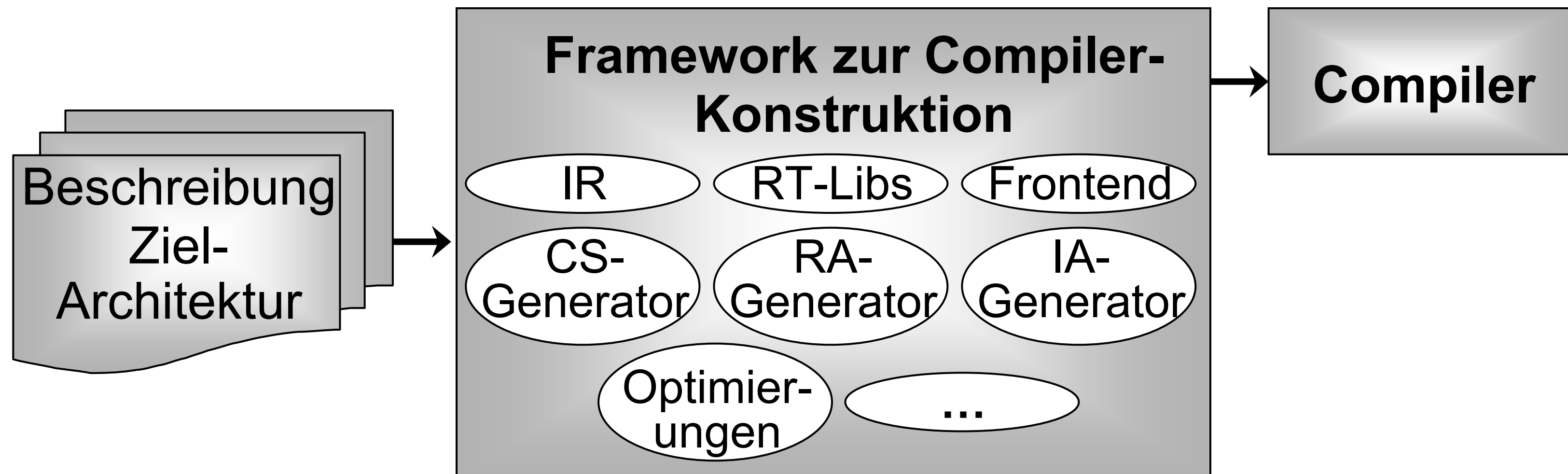
Automatische Compiler-Konstruktion



Compiler-Konstruktion:

- Ausgangspunkt ist eine Beschreibung des Ziel-Prozessors
  - entweder in HW-Beschreibungssprache (z.B. VHDL) oder
  - in System-Beschreibungssprache (z.B. SystemC oder Lisa)
- Aus dieser Beschreibung heraus wird automatisch ein Compiler-Backend erzeugt.

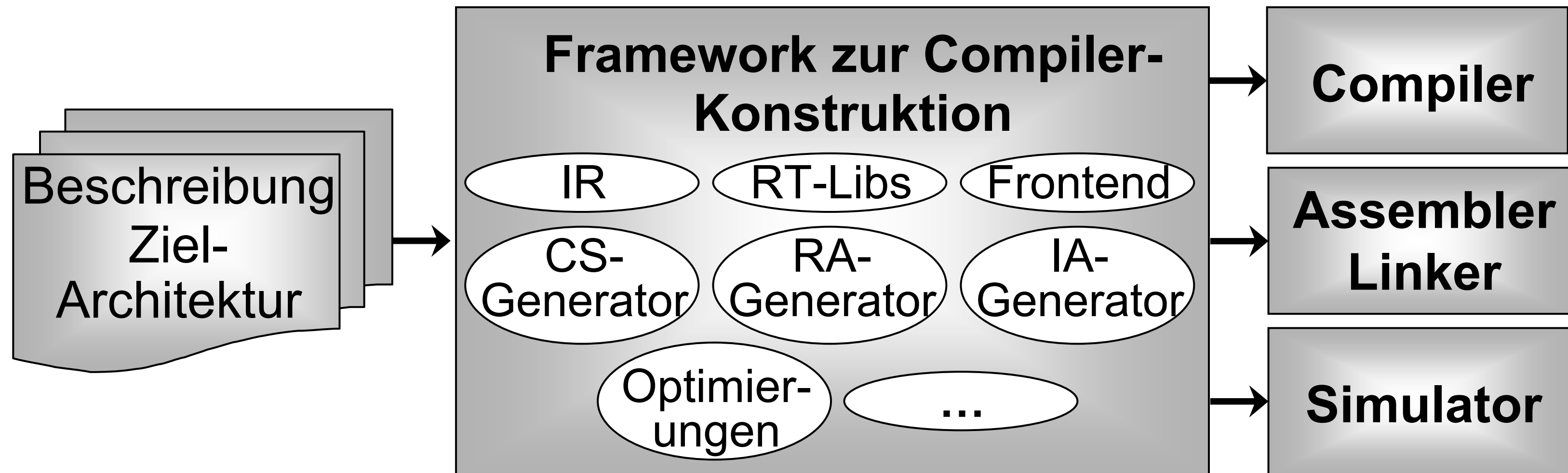
Automatische Compiler-Konstruktion



Compiler-Konstruktion *(Fortsetzung)*:

- Aus der Prozessor-Beschreibung wird der komplette Befehlssatz des Prozessors extrahiert. Mit diesem wird dann eine vollständige Baum-Grammatik zur Generierung des Code-Selektors erzeugt.
- Analog werden Features der Register-Files extrahiert, um einen Register-Allokator zu generieren. *(Analog für einen Scheduler)*

Automatische Compiler-Konstruktion



Compiler-Konstruktion *(Fortsetzung)*:

- Diese Backend-Komponenten werden mit „vorgefertigten" Standard-Komponenten gekoppelt, die die zentrale IR, das Frontend sowie IR-Optimierungen und Laufzeit-Bibliotheken bereitstellen.
- Neben einem Compiler kann ein solches Framework zusätzlich Assembler, Linker und Zyklus-genauen Simulator generieren.