

# Einführung in den Compilerbau

Andreas Koch

Wintersemester 2018-2019

## Inhaltsverzeichnis

<b>1 Organisatorisches</b>	<b>2</b>
1.1 Grundlage der Vorlesung . . . . .	2
1.2 Übersichtswerk . . . . .	2
1.3 Aufbau der Veranstaltung . . . . .	2
<b>2 Einleitung</b>	<b>2</b>
2.1 Compiler . . . . .	2
2.2 Auswirkung von Compilern . . . . .	3
2.3 Programmiersprachen . . . . .	3
2.4 Abstraktionsebenen . . . . .	4
<b>3 Zielmaschine</b>	<b>4</b>
3.1 Auswirkungen der Zielmaschine . . . . .	4
3.2 Anforderungen an CPUs . . . . .	6
3.3 Paralleles Rechnen . . . . .	6
3.4 Compilerbau in der Lehre . . . . .	6
<b>4 Aufbau</b>	<b>7</b>
4.1 Vorgehen . . . . .	7
4.2 Syntaxanalyse . . . . .	7
4.3 Kontextanalyse . . . . .	7

# 1 Organisatorisches

## 1.1 Grundlage der Vorlesung

Die Vorlesung basiert **fast vollständig** auf *Programming Language Processors in Java*<sup>1</sup>. Auszugsweise noch weiteres Material, z. B. zum ANTLR-Parsergenerator.

## 1.2 Übersichtswerk

Einen guten allgemeinen Überblick, aber im Detail mit anderen Schwerpunkten als diese Vorlesung, bietet *Compilers, 2. Auflage*<sup>2</sup>.

## 1.3 Aufbau der Veranstaltung

Diese Veranstaltung ist logisch in mehrere Teile gegliedert.

**Front-End**<sup>3</sup> Übersicht, ca. 3 Wochen.

- Lexing und Parsing,
- Zwischendarstellungen.

**Middle-End** Übersicht, ca. 2 Wochen.

- Semantische- und Kontextanalyse.

**Back-End** Übersicht, ca. 4 Wochen.

- Laufzeitorganisation,
- Code-Erzeugung.

**Front-End-Generatoren** Verwendung, ca. 2–3 Wochen.

**Java Virtuelle Maschine** ca. 1–2 Wochen.

# 2 Einleitung

## 2.1 Compiler

Ein Compiler ist eine Schnittstelle zwischen *Mensch* und *Maschine*. Er übersetzt von einer Programmiersprache (Menschenlesbar) in eine Maschinensprache (maschinenlesbar).

---

<sup>1</sup>von David Watt und Deryck Brown, Prentice-Hall 2000

<sup>2</sup>Von Aho, Sethi, Ullmann, Lam, Addison-Wesley 2006. Auch auf Deutsch verfügbar.

<sup>3</sup>Die ersten drei Teile der Veranstaltung richten sich an die Veranstaltungen *IMT3052* von Ivar Farup, Universität Grøvik, Norwegen; und *Vertalerbouw* von Theu Ruys, Universität Twente, Niederlande.

**Programmiersprache** Gut für Menschen lesbar. Beispiele für Programmiersprachen sind:

- Smalltalk,
- Java,
- C++.

**Maschinensprache** Getrimmt auf

- Ausführungsgeschwindigkeit,
- Preis pro Chip, Fläche,
- Energieverbrauch,
- (nur selten) leichte Programmierbarkeit.

## 2.2 Auswirkung von Compilern

Mit den Eigenschaften entscheidet ein Compiler über die dem Nutzer zugängliche Rechenleistung. Durch gewisse Abwägungen kann ein Compiler ein Programm so kompilieren, dass es am schnellsten läuft oder dass der Resultierende Code an kleinsten ist.

Compiler	Ausführungszeit	Programmgröße
GCC 3.3.6	7,5 ms	13 KB
ICC 9.0	6,5 ms	511 KB

Tabelle 1: Bildkompression auf Dothan CPU, 2 GHz

## 2.3 Programmiersprachen

**Hohe Ebene** Smalltalk, Java, C++. Beispiel:

```
let
  var i : Integer;
in
  i := i + 1;
```

**Mittlere Ebene** Assembly

```
LOAD  R1, (i)
LOADI R2, 1
ADD   R1, R1, R2
STORE R1, (i)
```

## Niedrige Ebene Maschinensprache

```
0110000100000110  
0111001001000001  
1011000100010010  
1001000100000110
```

## 2.4 Abstraktionsebenen

Auf den unteren Ebenen werden die Beschreibungen immer feiner, da man näher an der Zielmaschine (Hardware) arbeitet.

Der Compiler ist dafür zuständig, Details hinzuzufügen. In den obenstehenden Codebeispielen musste der Compiler unter anderem Register wählen, in denen die Werte während dem Programmablauf zwischengespeichert werden. Außerdem musste die Adresse der Variable `i`, hinzugefügt werden, wobei hier `00000110` verwendet wurde.

Diese Details werden mithilfe von verschiedensten Algorithmen ergänzt, welche die **Programmeigenschaften Analysieren** und durch die **Synthese von Details** die Beschreibung Verfeinern.

## 3 Zielmaschine

### 3.1 Auswirkungen der Zielmaschine

Die Zielmaschine hat einen Einfluss auf die Architektur des Compilers. So basiert zum Beispiel die DLX Architektur von John Hennessy und David Patterson auf der MIPS Architektur, und ist nur leicht verändert um diese zu modernisieren. Damit ist es sehr einfach, einen Compiler zu bauen, der für diese Architektur Code generiert.

Etwas komplizierter wird es mit der *TigerSHARC* Architektur von Analog Devices. Dies ist ein Beispiel für ein DSP, also *Digital Signalling Processor*. Es gibt hier zwei *Computational Blocks*, damit parallel Rechnungen ausgeführt werden können. Diese können aber nicht kommunizieren, also muss der Compiler drauf achten, dass auf die Register nur von dem jeweiligen Block aus zugegriffen werden können. Außerdem besitzt diese Architektur separate Rechenblocks, um Adressen zu bestimmen. Ein Compiler für diese Architektur muss also wissen, wie die Architektur aufgebaut ist, um sie effizient zu nutzen.

Am problematischsten wird das aber erst bei extremen Architekturen wie der des IBM/Sony *Cell* Prozessors. Hierbei handelt es sich um eine sehr gewagte, und wie sich leider herausgestellt hat, zu komplexe Architektur. Es gibt ein PowerPC-basierten Hauptprozessor, der aber nicht sehr leistungsstark ist. Die eigentliche Arbeit kann von *Synergistic Processing Elements*, kurz SPE, ausgeführt werden. Diese sind so konstruiert, dass ein sorgfältig produzierter Instruktionsstream die Hardware

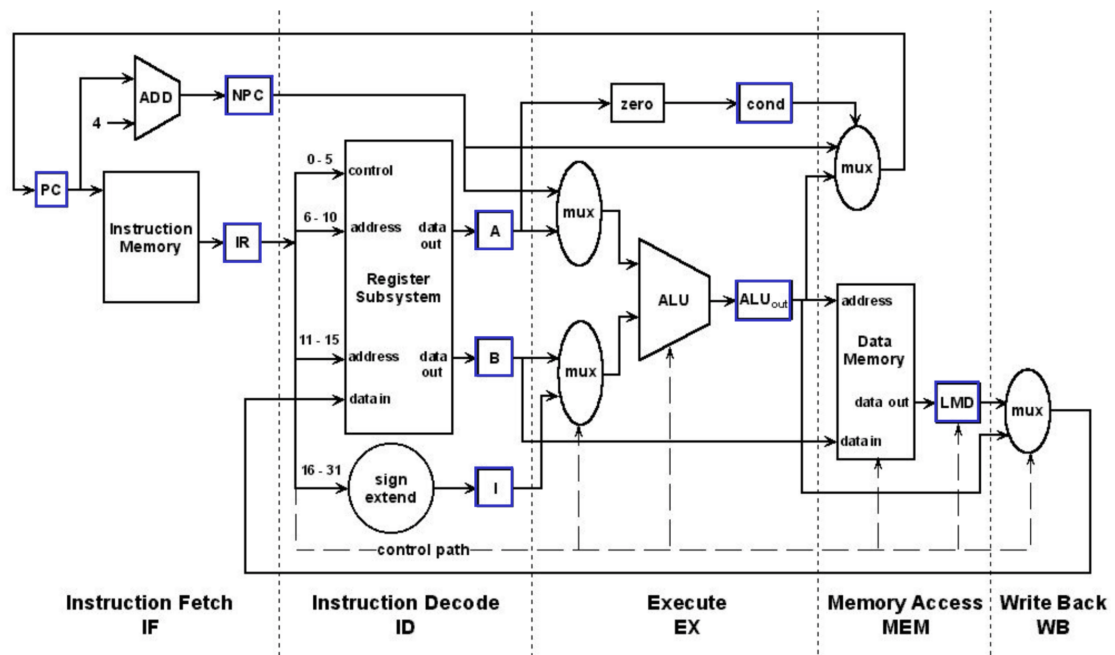


Abbildung 1: Die DLX RISC Prozessorarchitektur

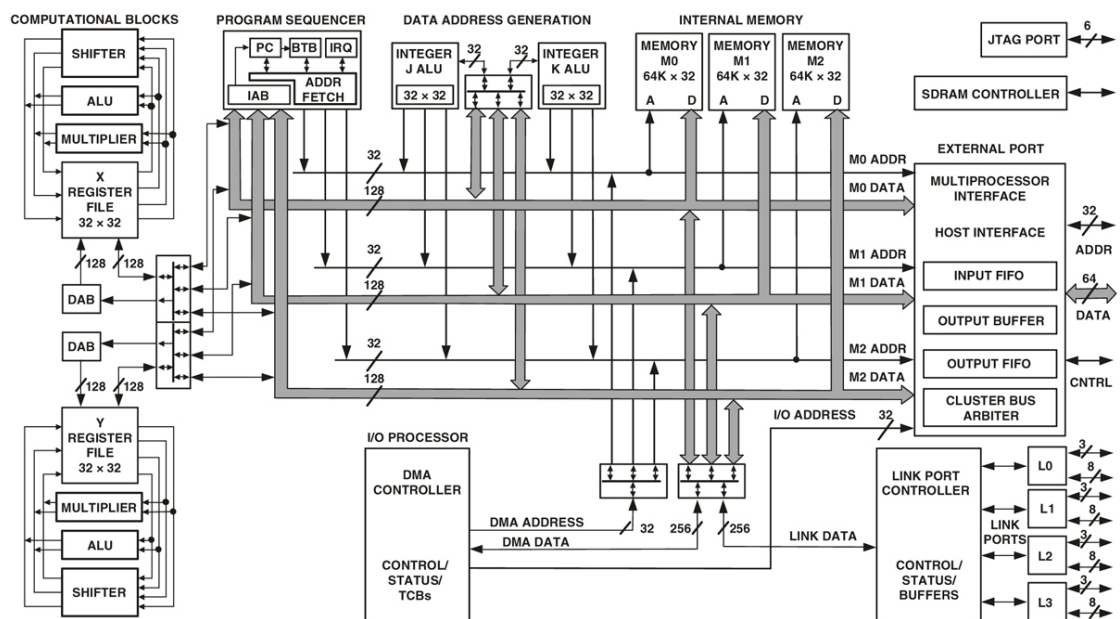


Abbildung 2: Analog Devices TigerSHARC

Abbildung 3: Ein Synergistic Processing Element eines Cell Prozessors.

maximal ausnutzen kann, mit parallelen Arithmetischen- und Speichereinheiten (siehe Abbildung 3).

### 3.2 Anforderungen an CPUs

Je nach Anwendungsgebiet sind die Anforderungen mehr oder weniger wichtig.

- Rechenleistung
- Datentype (Gleitkomma, ganzzahlig, Vektoren)
- Operationen (Multiplikationen, MAC<sup>4</sup>)
- Speicherbandbreite (parallele Speicherzugriffe)
- Energieeffizienz
- Platzbedarf (für den Prozessorchip)

Oft können manche Anforderungen nur durch spezialisierte Prozessoren erfüllt werden.

### 3.3 Paralleles Rechnen

Paralleles Rechnen ist die Königsklasse der Forschung. Warum eigentlich? Das Wettrennen der Taktfrequenzen ist im großen und ganzen vorbei, mehr als ~4 GHz ist nicht realistisch. Also ist **der Trend weg von hochgetakteten Einzelprozessoren und hin zu vielen (2-8, teilweise 16) Prozessoren**, die aber weniger schnell getaktet werden. Dadurch erreicht man mehr Rechenleistung. Aber wie kann man für solche parallele Rechenkapazitäten programmieren?

Erste praktische Ansätze sind OpenMP, mit dem man für parallele CPUs programmieren kann. OpenCL kann mit heterogenen Systemen (GPUs, CPUs, experimentell auch FPGAs) arbeiten, und für Nvidia Grafikkarten gibt es CUDA. **Diese Lösungen erfordern aber aktuell noch, dass der Programmierer explizit für ein paralleles System programmiert. Es gibt noch keine automatische Parallelisierung.**

### 3.4 Compilerbau in der Lehre

Warum wird der Compilerbau so früh gelehrt? Es handelt sich hierbei um eine Kombination von verschiedenen Disziplinen. Zum Compilerbau benötigt man einen *Parser*, welcher zur theoretischen Informatik gehört. Man braucht auch das Hintergrundwissen der *Architektur der Zielmaschine*, welches zur technischen Informatik gehört. Außerdem braucht man zum Compilerbau Kenntnisse von Software-Engineering, was zur praktischen Informatik gehört. Damit vereint der Compilerbau drei wichtige Disziplinen.

---

<sup>4</sup>Mir als *fused multiply and add* bekannt, eine Operation, die vor allem für Matrizenmultiplikation signifikant ist.

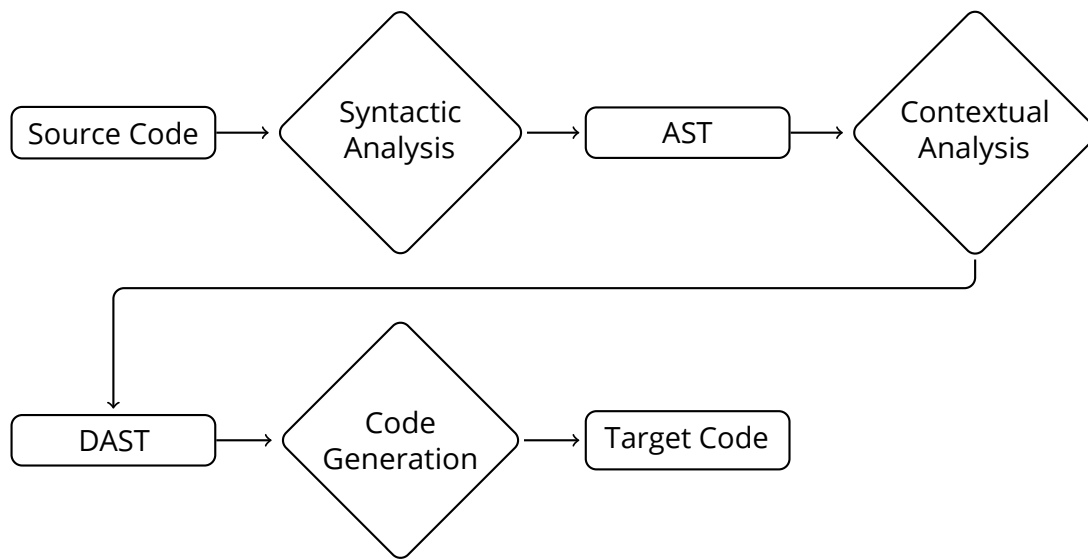


Abbildung 4: Zwischendarstellungen für den Informationsaustausch

## 4 Aufbau

### 4.1 Vorgehen

Compiler arbeiten generell in mehreren Phasen. In Abbildung 4 sind diese Phasen aufgezeigt. Außerdem sind die Zwischendarstellungen, die zum Informationsaustausch zwischen den Phasen genutzt werden, dargestellt.

### 4.2 Syntaxanalyse

Bei der Syntaxanalyse wird überprüft, ob das Programm Syntaxgerecht aufgebaut ist.

### 4.3 Kontextanalyse

Bei der Kontextanalyse werden Variablen ihren Deklarationen zugeordnet. Außerdem werden die Typen von Ausdrücken berechnet.