Hauptseminarvortrag
XASM – ein generisches Assemblerframework

Marco Kaufmann

s9186072@mail.inf.tu-dresden.de

Gliederung

1. Motivation
2. Grundgedanken & Entwurfsentscheidungen
3. Opcode Maps
4. Assemblerprogramme
5. Sprachelemente
6. Aktueller Stand & Ausblick

1. Motivation

1. Motivation
2. Grundgedanken & Entwurfsentscheidungen
3. Opcode Maps
4. Assemblerprogramme
5. Sprachelemente
6. Aktueller Stand & Ausblick

- Viele verschiedene Architekturen und Befehlssätze
  - Zahl steigend, bedingt durch technische Entwicklung
  - Vielzahl kleiner Architekturen, die weniger verbreitet sind
- Aufwändig, für alle einen Assembler von Grund auf neu zu entwickeln

Was bedeutet „generisch“?

- Lösung: ein Assemblerframework, dass sich für verschiedene Architekturen und Befehlssätze einfach anpassen lässt
  - Bereitstellung von bestimmten Standardfunktionalitäten
  - Anpassung an individuelle Bedürfnisse
- Kriterien für ein generisches Assemblerframework
  - Je weniger Aufwand für Anpassung an Architekturen erforderlich, desto besser.
  - Dabei muss das Framework trotzdem so flexibel sein, dass es sich für möglichst alle Architekturen anpassen lässt.

GNU Assembler

- Ein Open Source – Assembler unter GNU Public License
  - Quelltext kann von jedem modifiziert werden
  - Stellt ein Framework bereit, in dem der Programmierer nach Belieben Änderungen vornehmen und den Assembler an eigene Bedürfnisse anpassen kann.
- Ist jedoch nicht flexibel genug, lässt sich für bestimmte Sachverhalte nicht oder nur mit erheblichem Aufwand anpassen.
- Beispiel: SHAP
  - 9 Bit Wortbreite im Codesegment, verschiedene Wortbreiten in Code – und Datensegment
  - Jump Table, automatische Konstantentabelle

2. Grundgedanken & Entwurfsentscheidungen

1. Motivation
2. Grundgedanken & Entwurfsentscheidungen
3. Opcode Maps
4. Assemblerprogramme
5. Sprachelemente
6. Aktueller Stand & Ausblick

XASM „Vorläufer“: CodeX Assembler

- Als BELL im Jahr 2002 (Klasse 11/12) entstanden
  - Motivation: Assembler, der selbst nicht modifiziert werden muss wenn der x86 Befehlssatz erweitert wird
- Idee: Befehlssatz nicht fest im Assembler integriert, sondern durch externe Textdatei (Opcode Map) beschrieben.
  - Deklaration der Registersätze, Befehlsmnemoniken und Opcodes durch eigene, einfache Beschreibungssprache
  - CodeX Assembler kann diese Textdateien selbst übersetzen
  - Erweiterung des Befehlssatzes dadurch sehr einfach und sogar vom Anwender selbst durchführbar. Assembler muss dazu nicht modifiziert und neucompiliert werden.

XASM „Vorläufer“: CodeX Assembler

- Eigenschaften:
  - Implementierung in FreePascal, Assembler lief unter DOS
  - Konzept ging auf: Unterstützung aller x86 Befehlssätze bis Penitum IV und AMD 3D Now
  - Ausgabeformate: 16 / 32 Bit Flat Model Binaries, Real Mode – und 16 / 32 Bit Protected Mode DOS Anwendungen
- ABER: nur begrenzte Flexibilität!
  - Assembler nur für x86 ausgelegt, nicht für andere Architekturen verwendbar (z.B. MCS8)
  - Architekturspezifische Merkmale fest implementiert (Speicheroperanden, Mod R/M -, SIB Byte, ...)
  - X86-64 bereits ebenfalls nicht mehr durch Opcode Maps umsetzbar, da 64 Bit Operanden durch zusätzliche Befehlspräfixe codiert.
  - Erweiterung des Assemblers selbst quasi unmöglich: keine strenge Strukturierung, keine Kapselung
  - War nie als Framework vorgesehen, Benutzer sollte lediglich eigene Opcode Maps hinzufügen

- Opcode Maps beibehalten!
  - Da einfachste Art neue Befehlssätze zu implementieren.
- Framework soll diesmal *wirklich* generisch sein
  - ohne größere Probleme selbst an „exotische“ Architekturen anpassbar.

  ➔ Opcode Maps müssen flexibler gestaltet werden.

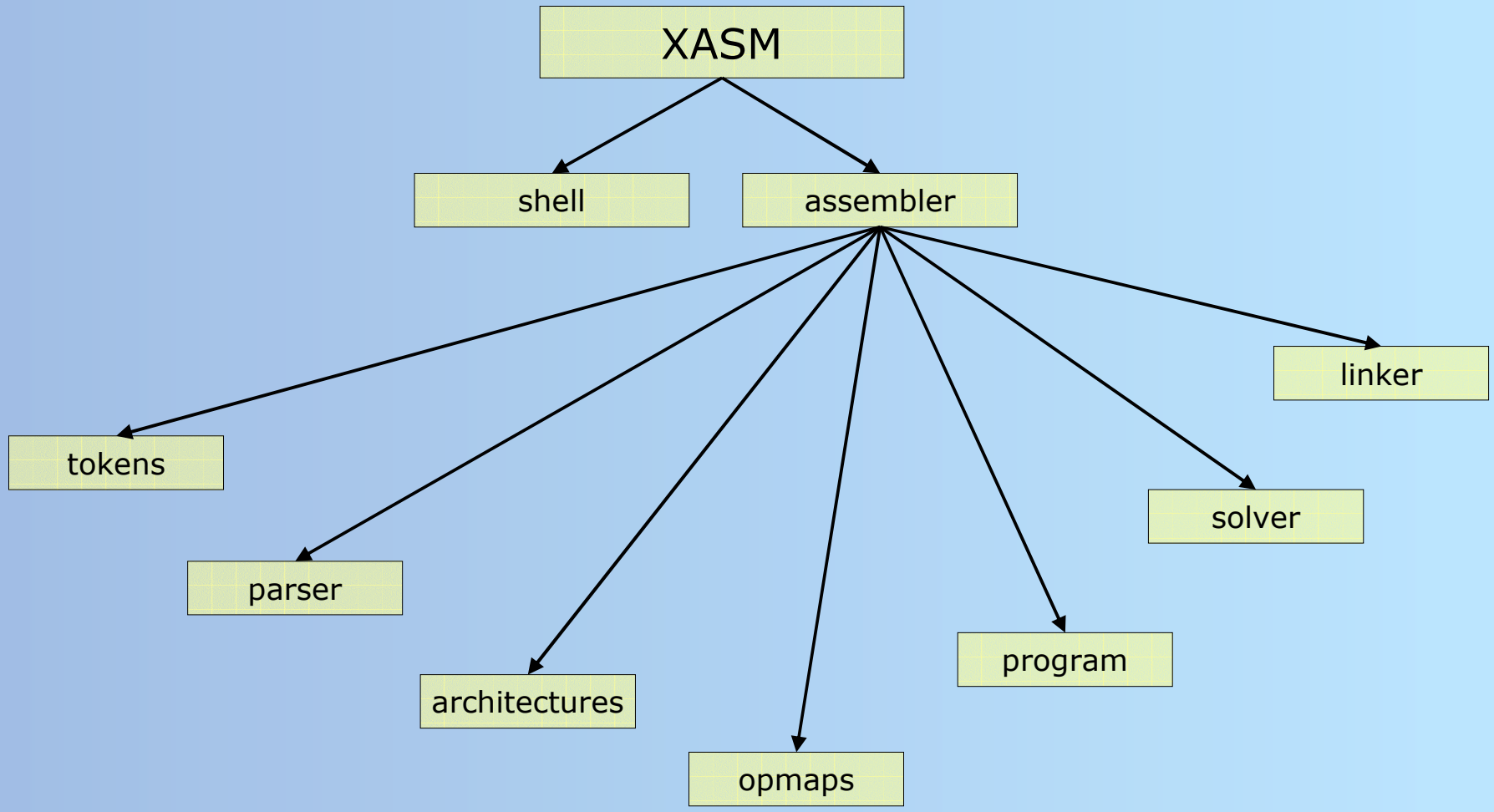
  ➔ Generik durch Opcode Maps allein außerdem nicht ausreichend
- Bei Bedarf kann in jeden Arbeitsschritt des Assemblers eingegriffen werden
  - Bau des Befehlsworts, Adressberechnung, Parsen einer Assemblerbefehlszeile, ...

  ➔ Arbeitsschritte und Teilaufgaben müssen gekapselt, Methoden gut dokumentiert sein. Welche architekturspezifischen Klassen und Methoden bewirken was? Wie überschreiben, um neue architekturspezifische Features zu integrieren?
- Ideales Zusammenspiel von Opcode Maps und architekturspezifischen Klassen
  - Weitestgehende Realisierung durch Opcode Maps („alles was geht“)
  - Spezialfälle werden durch architekturspezifische Klassen implementiert

Implementierung

- Implementierung in Java
  - Streng objektorientiert
  - Macht es dem Programmierer „einfach“ (→ Garbage Collector, Generics, keine Pointer), erleichtert so das Schreiben architekturspezifischer Klassen
  - Plattformübergreifend; Assembler läuft auf jeder Plattform mit JRE
- Opcode Map Parser mit WISENT implementiert
  - LALR1 Parser Generator, YACC kompatibel, Backends für Java und C++
  - Einfacher, als Parser per Hand zu schreiben.
  - Opcode Map Parser muss nicht erweiterbar sein.
- Parser für Assemblerprogramme selbst implementiert
  - Parser muss erweiterbar sein, u.a. architekturspezifische Assemblerdirektiven, alternative Syntax für Befehlszeilen
  - Modularer Aufbau, Kapselung von Teilaufgaben
  - Soll nicht nach erster Fehlermeldung abbrechen, Fehlerbehandlung mit Parsergeneratoren aber schwierig.

Packages


XASM
shell
assembler
linker
tokens
parser
solver
architectures
program
opmaps

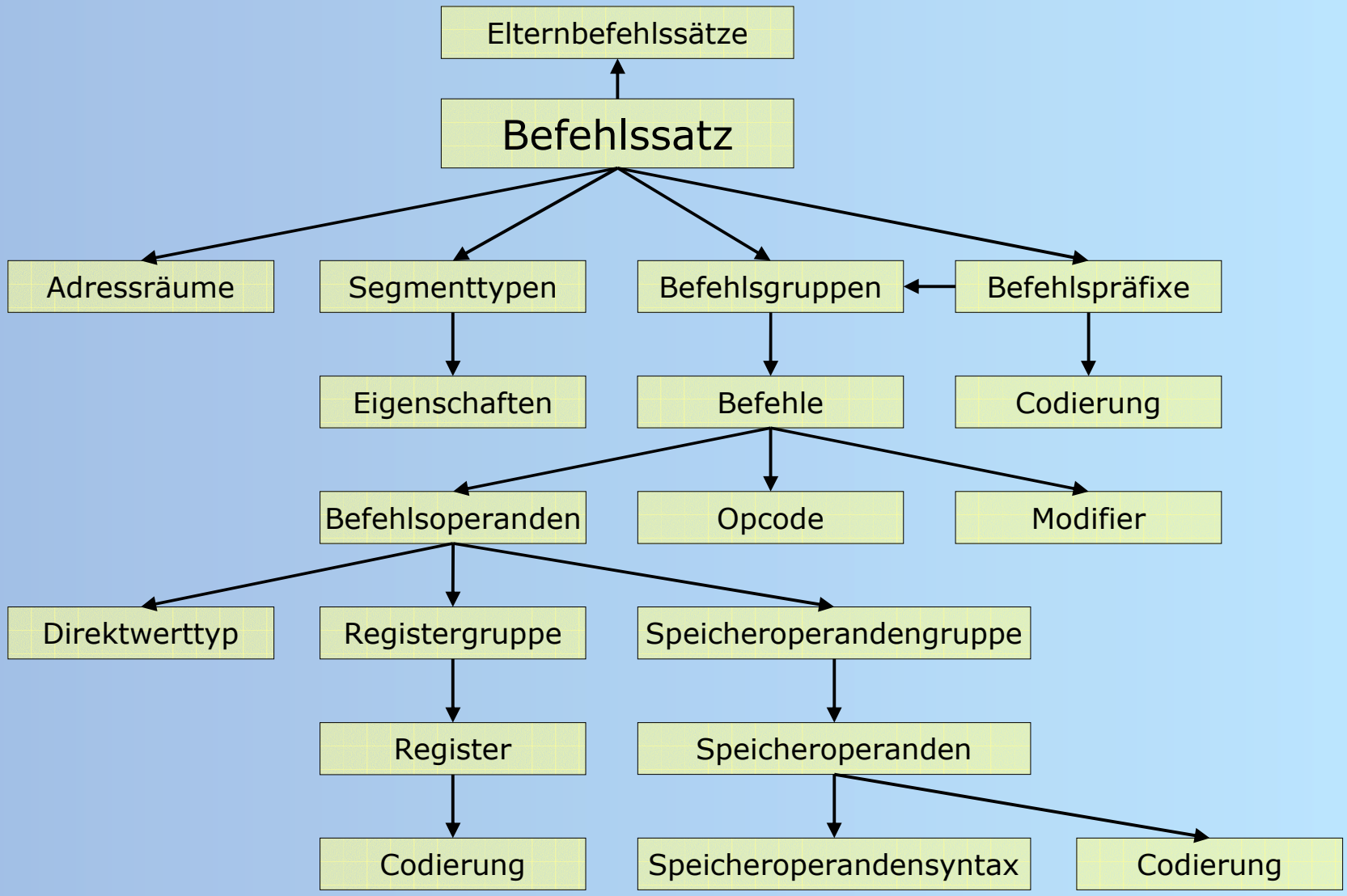
3. Opcode Maps

1. Motivation
2. Grundgedanken & Entwurfsentscheidungen
3. Opcode Maps
4. Assemblerprogramme
5. Sprachelemente
6. Aktueller Stand & Ausblick

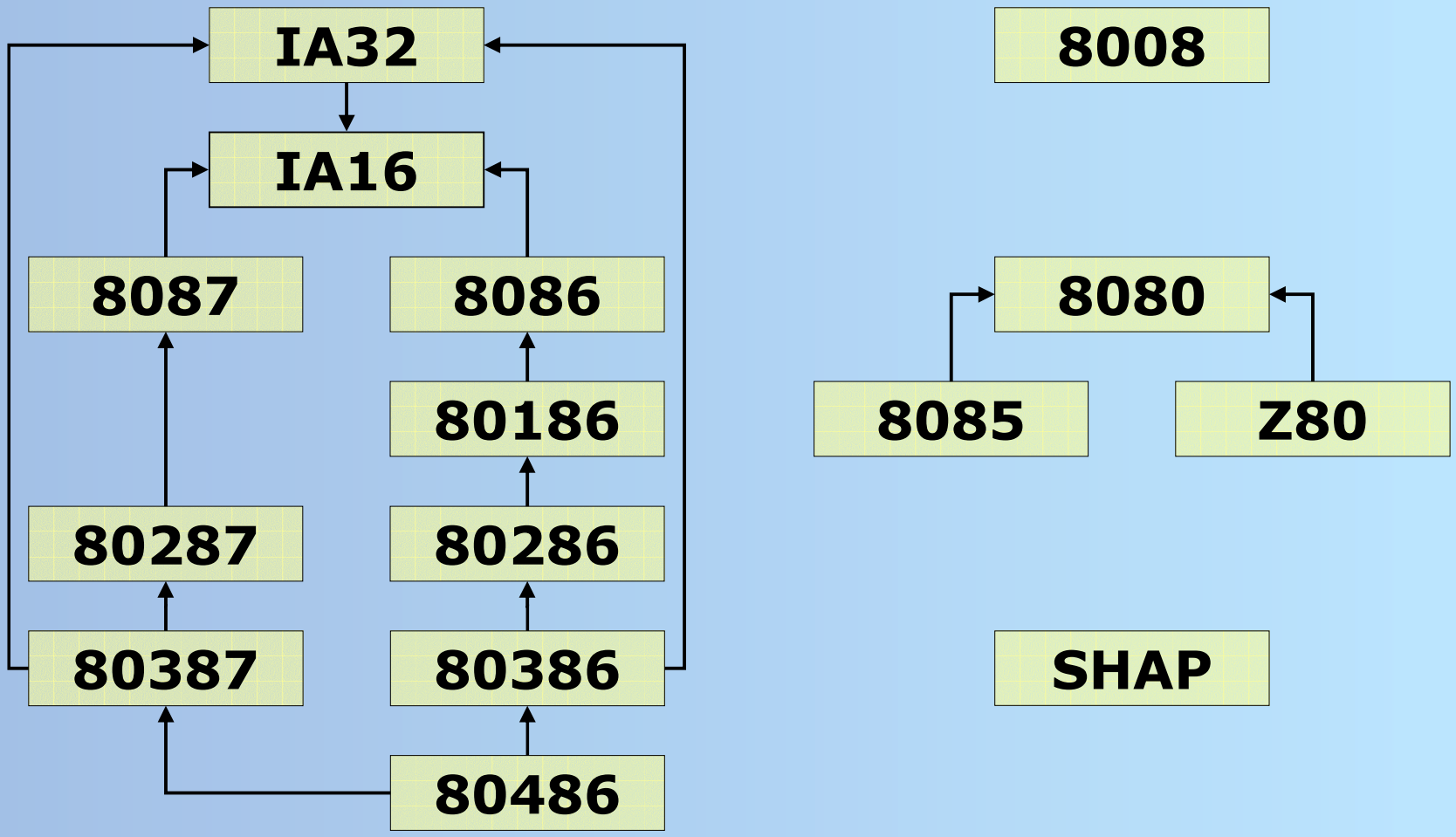
Eigenschaften

- Beschreibung aller Befehlssatzelemente: Assemblerbefehle und Befehlsmnemonic, Opcodes, Registersätze, Segmenttypen, Adressräume u.a
- Nicht aber weitere Architekturmerkmale und Details: Befehlswortformat, Besonderheiten bei der Adressberechnung, architekturspezifische Assemblerdirektiven etc.
- Vererbung und Mehrfachvererbung von Befehlssätzen möglich
- Automatisch generierte Befehlssatzdokumentation in HTML

Struktur eines Befehlssatzes



Vererbung - Beispiel



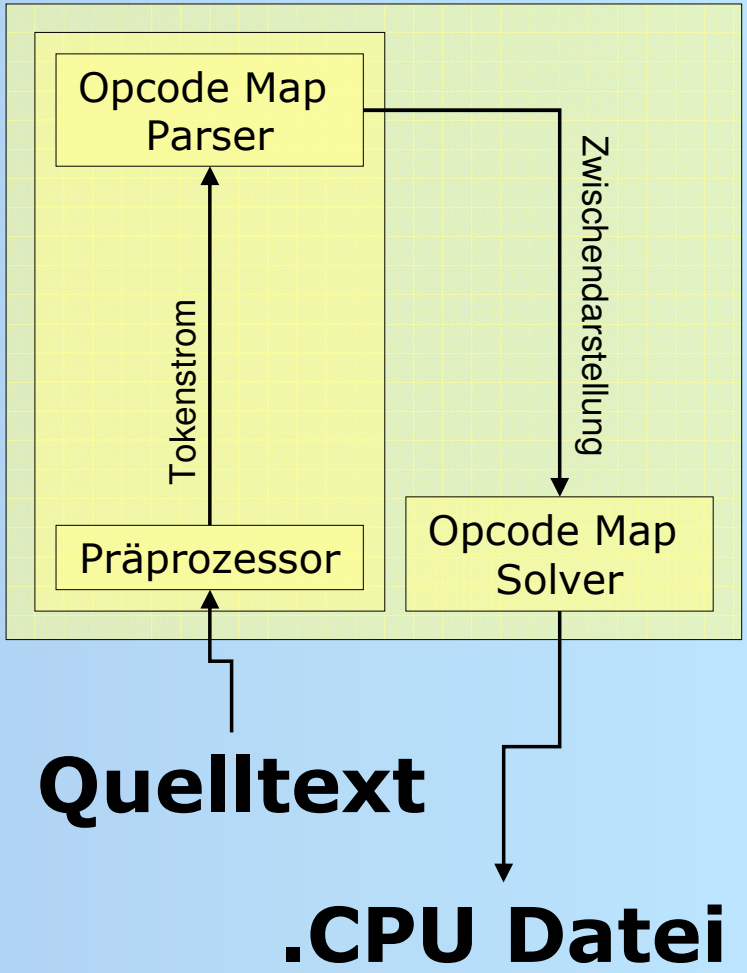
Opcode Map Quelltextdateien

```
architecture default

public set z80 "Zilog Z80" extends 8080
{
  registers
  {
    group pp "Register pairs and IX"
    {
      bc = 0x00; de = 0x10; ix = 0x20; sp = 0x30
    }

    group rx "Index Registers"
    {
      ix                                    = 0xDD pre
      iy                                    = 0xFD pre
    }
    ...
  }

  memory
  {
    group mx "Memory indexed"
    {
      [rx+imm8] = mem:2
    }
    ...
  }

  instructions
  {
    group transfer "Data Transfer"
    {
      ld,mov d,mx                           = 0x46
      ld,mov mx,s                           = 0x70
      ld,mov mx,imm8                        = 0x36
      ...
    }
    ...
  }
}
```

```
public set shap "Shap CPU"
{
  segments // segment types
  {
    group text "Code Segment (Type)"
    {
      byte = 9
      word = 1
      endian = big
    }

    group data "Data Segment (Type)"
    {
      byte = 32
      word = 1
      max = 64
      endian = big
    }
  }

  spaces // address spaces of CPU
  {
    text text "Code Memory"
    data data "Data Memory"
  }

  immediates
  {
    addimm = 10 add "Immediate to add to the opcode."
  }

  ...
```

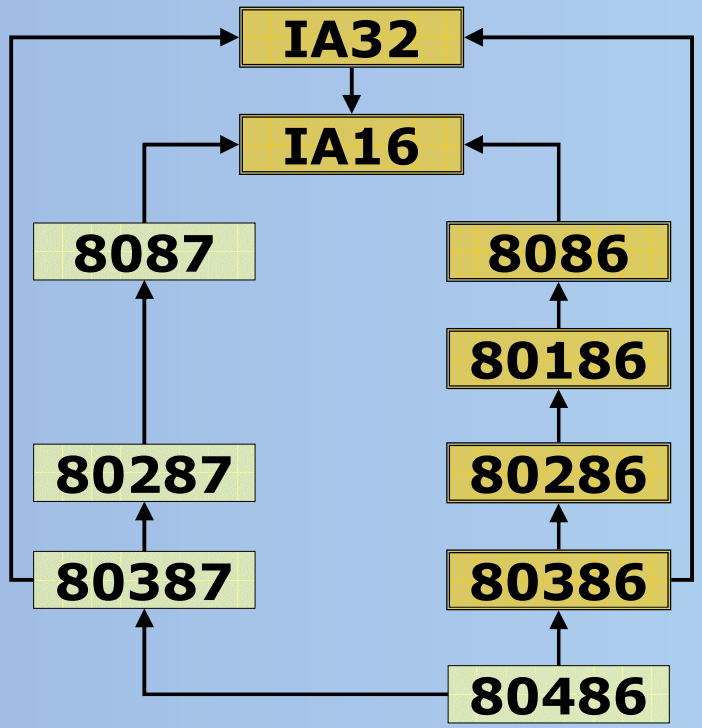
Arbeitsschritte beim Übersetzen

- Scannen
  - Verarbeitung von Makros und anderen Präprozessoranweisungen
  - Liefert Tokenstrom für Parser
- Parsen
  - Syntaktische Analyse
  - Überführung in interne Zwischendarstellung
- Solven
  - Soweit möglich, Auflösen von Symbolen
  - Zieldatei schreiben (serialisiertes InstructionSet – Objekt)



Arbeitsschritte beim Laden

```
.cpu 80386

.code

xor eax,eax
...
```

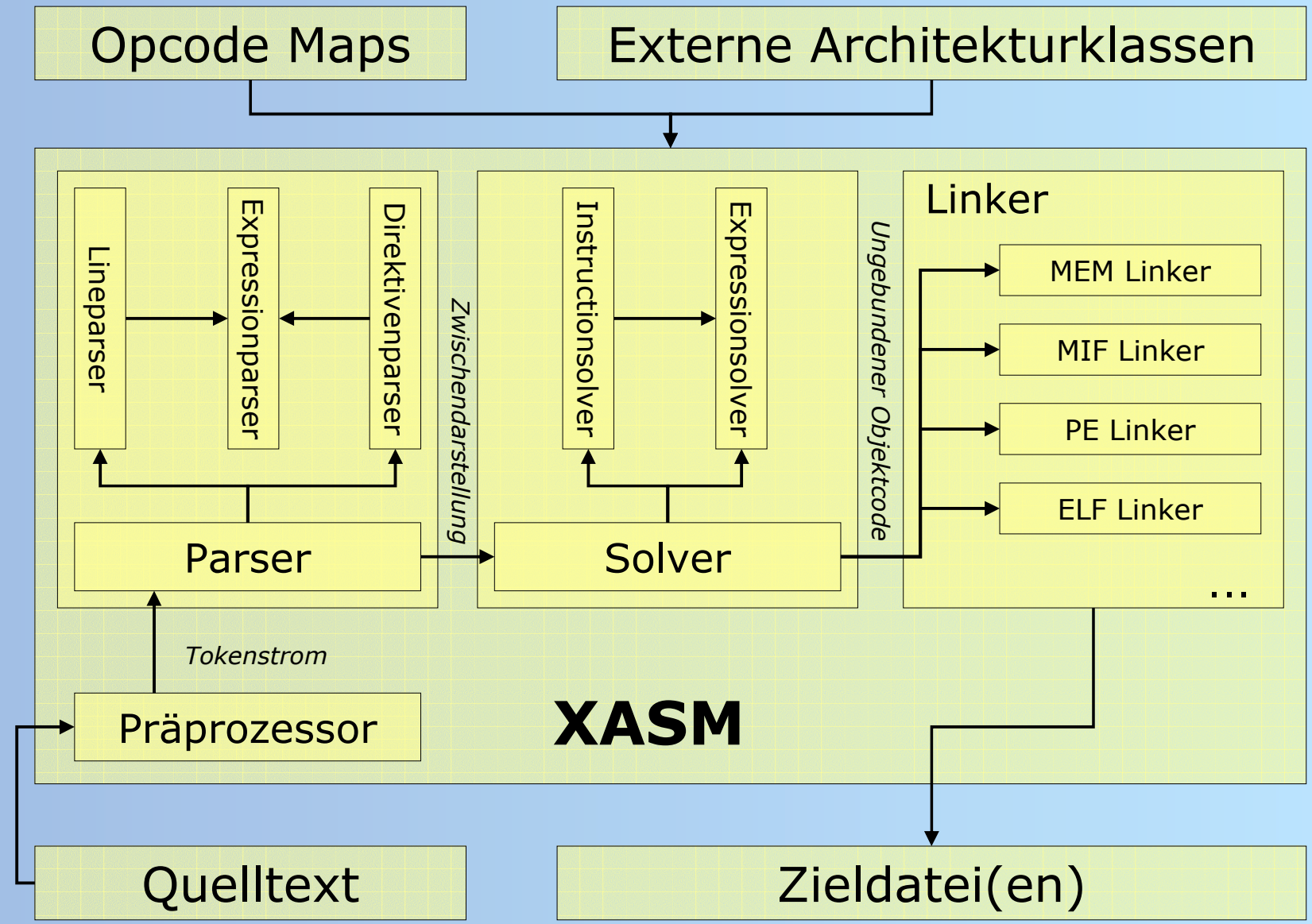


- Datei lesen
  - Serialisiertes InstructionSet - Objekt laden
  - Elterndateien ebenfalls lesen
- Vererbung
  - Entlang Vererbungspfad: Eltern – mit Kindbefehlssatz vereinigen
- Solven
  - Auflösen befehlssatzübergreifender Symbole

4. Assemblerprogramme

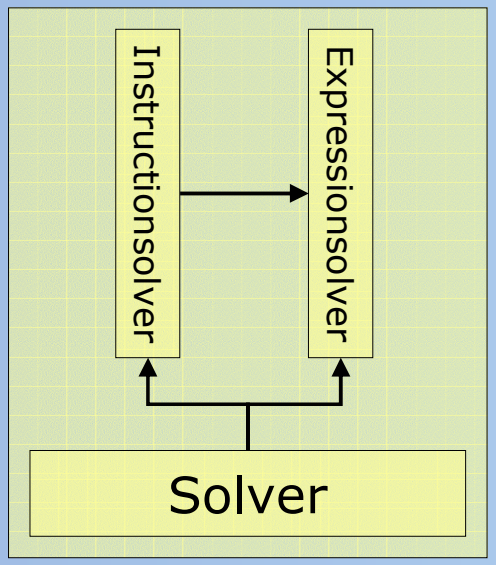
1. Motivation
2. Grundgedanken & Entwurfsentscheidungen
3. Opcode Maps
4. Assemblerprogramme
5. Sprachelemente
6. Aktueller Stand & Ausblick

Arbeitsschema



Arbeitsschema: Adressberechnung

```
.cpu 80386

c1 equ 3*sizeof(myvar)

.code
.subsect s1
.subsect s2

  mov ax,myvar
  cmp ax,1+2+3+4
  jz m1

.subsect s2
m1:
  jmp m1

.data
myvar dw sizeof s2
```



- Segmente in Reihenfolge der Deklaration lösen
- Subsections, Prozeduren, Variablen, Anweisungen in Reihenfolge der Deklaration lösen
- Elemente (Auswahl):
  - Anweisung: Operanden auflösen, Befehlswort bestimmen, Adresse aktualisieren.
  - Variable: mit aktueller Adresse markieren, Größe bestimmen, Adresse aktualisieren.
  - Prozedur oder Subsection: Mit aktueller Adresse markieren, Elemente aus Prozedur oder Subsection lösen
  - Label: mit aktueller Adresse markieren

Adressberechnung – Problem I

- Adresse wird als Operand verwendet, obwohl noch unbestimmt

```
.cpu 80386

...

mov eax,offset m1

...

m1:
```

- Adresse beeinflusst sich selbst

```
.absolute 20h
buffer resb offset m1-20h

...

m1:
```

- **Lösung:** Ausdrücke, die Adressen enthalten erst auswerten und einsetzen, nachdem Adressen berechnet
- Bereich im Zielcode wird vorher entsprechend Operandengröße reserviert
- D.h. zusätzlicher Arbeitsschritt: nach Auflösen und Adressberechnung noch immer ungelöste Ausdrücke auswerten und einsetzen

- **Lösung:** nicht erlaubt!

Adressberechnung – Problem II

- Mehrere Befehle möglich, Beispiel: jnz rel8; jnz rel16

```
.cpu 80386

...

jnz m1

...

m1:
```

- Verwendung des Befehls mit kürzestem Befehlswort (optimal)
  - Operandenbreite muss hinreichend groß sein
- ABER: Entfernung von m1 zur aktuellen Adresse noch gar nicht bekannt!
- **Lösung:** „Abschätzen“, wie groß Entfernung maximal werden kann
  - Für alle Assemblerbefehle zwischen aktueller und angegebener Adresse maximale Befehlswortgröße bestimmen
- Entsprechend maximaler Entfernung Befehl auswählen
- Besser, als stets Befehl mit den breitesten Operandenfeldern zu verwenden
- exakte Bestimmung des optimalen Befehls wäre dagegen zu aufwändig

Weitere Aspekte

- Modifier & Standardvorgaben für Befehlswortbildung
- Solven von Ausdrücken
- Segmentadressen
  - Zum Übersetzungszeitpunkt meist unbekannt
- Compilerdirektiven, z.B.
  - .ORG, .ABSOLUTE, .ALIGN, ...
- Automatische Codeerzeugung, z.B.
  - Automatisch generierter Prozedur Ein – und Austrittscode
  - Ersetzen nicht existenter Makro - Maschinenbefehle durch Sequenz existierender Befehle (beispielsweise 80286 shr ax,2 = 8086 shr ax,1 shr ax,1)
- Einfache Codeoptimierung?
  - Beispiel: mov ax,0 = xor ax,ax

Eingriff in Arbeitsschritte

- Überschreiben architekturspezifischer Klassen
- Beispiel: Befehlswortbildung weicht vom Standard dadurch ab, dass jedem Befehlswort das Byte 0xFF vorangestellt werden soll

```
class NewSet extends InstructionSet
{
  public Solver createProgSolver()
  {
    return new NewSolver();
  }
}
```

```
class NewSolver extends ProgSolver
{
  public Inst createInst()
  {
    return new NewInst();
  }
}
```

```
class NewInst extends Inst
{
  public int getInstWordSize(Instruction i)
  {
    return super.getInstWordSize(i) + config.bytesize;
  }

  public Code getInstWord()
  {
    Code c = new Code(0xFF,config.bytesize);
    c.add(super.getInstWord());
    return c;
  }
}
```

```
public void main(String[] args)
{
  ConsoleShell s = new ConsoleShell
  (
    new AdvancedConsole(),args
  );

  if (s.prepare()
  {
    s.asm().registerArchitecture
    (
      "newarch", new NewSet("newarch",
      false,null)
    );
    s.run();
  }
}
```

```
architecture newarch

public set myinstructionset
{
  ...
}
```

5. Sprachelemente

1. Motivation
2. Grundgedanken & Entwurfsentscheidungen
3. Opcode Maps
4. Assemblerprogramme
5. Sprachelemente
6. Aktueller Stand & Ausblick

Grundlegendes

- Befehlszeile:

```
{Assemblerdirektive} {Befehlspräfix} [Assemblerbefehl [Operand{,Operand}]]
```

- Labels
  - Lokal, global

```
[@[@]] <Label>:
```

- Variablendeklaration
  - Lokal, global
  - Initialisiert, uninitialisiert
  - Stack, normal, absolut

```
[@[@]] <Variable> define | db | dw | dd | df | dq | dt | ddq
[@[@]] <Variable> res | resb | resw | resd | resf | resq | rest | resdq
```

- Konstantendeklaration

```
[@[@]] <Konstante> equ <Ausdruck>
```

Weitere Elemente

- Segmente und Subsections
  - Programm kann in verschiedene Segmente (Speichersektionen) und ein Segment in verschiedene Subsections unterteilt werden
  - Adresse pro Segment beginnt bei 0, Symbole im eigenen Sichtbarkeitsbereich
  - Anzahl und Reihenfolge frei wählbar, beliebige Segment– und Subsectionwechsel möglich
- Blöcke und Prozeduren
  - Prozeduren: Zusammenfassung einer Codesequenz, Symbole im eigenen Sichtbarkeitsbereich
  - Stackvariablen, Parameter und Procedure Overload
  - Automatischer Prozedur Ein- / Austrittscode
  - Blöcke: „Light“- Prozeduren
- Datentypen
  - Deklaration zusammengesetzter, komplexer Datentypen möglich
  - Records und Arrays
  - Beliebig tief verschachtelbar

Auswahl einiger Assemblerdirektiven

- .CPU
  - Wählt Ziel CPU aus
- .TARGET
  - Wählt Zielformat aus
- .STARTUP:
  - Programmeinsprungpunkt
- .SECTION, .SEGMENT, .CODE, .DATA
  - Segmentwechsel oder Segment erzeugen
- .SUBSECT, .SUBSEGMENT
  - Subsectionwechsel oder Subsection erzeugen
- .ABSOLUTE, .ALIGN
  - Ausrichtung des nachfolgenden Elements an einer Adresse
- .ORG:
  - Adresskorrektur für beliebige Segmente
- .ASSUME
  - Stellt Beziehungen zwischen Segmenten und Adressräumen her
- WARNINGS
  - Einstellung, wann Warnungen erzeugt werden sollen

Präprozessor

- Präprozessoranweisungen sind verschachtelbar
- Include Files
  - Einbinden weiterer Quelltextdateien

```
.include <Filename>, .require <Filename>, .incpath <Pfad>
```

- Makros
  - Ein– und mehrzeilig
  - Parameter + Overload möglich

```
.macro, .endmacro, %%<Makro>
```

- Bedingtes Einbinden
  - Bindet einen Teil des Quelltextes ein / ignoriert ihn, in Abhängigkeit davon ob bestimmte Makros / Umgebungsvariablen gesetzt sind

```
.ifdef, .ifndef, .else, .endif, .def, .undef
```

- Kommentare

```
/* ... */, // ..., ; ...
```

6. Aktueller Stand & Ausblick

1. Motivation
2. Grundgedanken & Entwurfsentscheidungen
3. Opcode Maps
4. Assemblerprogramme
5. Sprachelemente
6. Aktueller Stand & Ausblick

Was funktioniert bereits?

- Übersetzen & Verwenden von Opcode Maps: fertig
- Präprozessor: fertig
- Parsen von Assemblerprogrammen: fertig
  - außer einigen Assemblerdirektiven, die später hinzugefügt werden (.assume, .org, .pushprm, .popprm, .warnings)
- Solven von Assemblerprogrammen: zum Großteil implementiert
  - Berechnen von Ausdrücken, Auflösen von Bezeichnern
  - Adress- und Größenberechnung, Codierung von Befehlen, Variablen
- Linker: Ausgabeformate MEM und MIF implementiert
- Projekt derzeit: 71 Java Klassen, ca. 20000 Zeilen Quelltext, 95 Seiten Dokumentation

Was ist noch zu tun?

- Solver fertigstellen
  - Auswahl des optimalen Befehls in Arbeit
  - Einige Assemblerdirektiven werden noch nicht aufgelöst
  - Automatische Codeerzeugung & Codeoptimierung (später)
- Weitere Opcode Maps und Architekturklassen
  - z.B. x86, MCS8
- Linker fertigstellen, zusätzlich geplante Ausgabeformate:
  - Flat Model Binaries
  - PE, COFF, ELF (später)

Vielen Dank!