SJC – Small Java Compiler

Ein Überblick

Folien für Vortrag am
17. März 2008

Stefan Frenz

© Stefan Frenz, Weitergabe und Kopie nur mit Genehmigung

Inhalt

- Überblick über den Small Java Compiler
  - Umfeld und Abgrenzung von SJC
  - Konzeption und Module
- Beispiele
  - Übersetzung
  - Intel ia32 (x86)
  - Atmel ATmega16
- Zusammenfassung
- Fragen, Diskussion, Anregungen

Historie

- Plurix-Java-Compiler PJC der Universität Ulm
  - Compiler für Cluster-Betriebssystem und Lehre
  - Fähigkeit zur Selbstübersetzung unter Plurix
  - Erzeugung von 32 Bit Code für ia32

Motivation

- Modularer Compiler
  - Möglichkeit für Cross-Compilation
  - freie CPU-Wahl (8-64 Bit, Harvard / von Neumann)

Sprach-Alternativen

- C
  - hardwarenahe Programmierung
  - Namensraumflutung und Defines kritisch
- C++
  - unvorteilhafte Copy/Assign-Semantik
  - Mehrfach-Vererbung schwer vermittelbar
- Pascal, ObjectPascal, Oberon
  - etablierte Compiler-Techniken, einfache Syntax
  - stark rekursiv (Debugging), dünne Tool-Landschaft

Abwägung der Sprach-Alternativen

- Vorteile von Java(-Syntax)
  - typsicher, eindeutige Vererbung
  - Code maschinenunabhängig eindeutig
  - Vielzahl an Entwicklungstools vorhanden
  - Compiler und Laufzeitumgebung vorhanden
- Nachteile von Java(-Syntax)
  - Objektorientierung für manche Probleme Overkill
  - Laufzeitverhalten bei exzessiver new-Verwendung
- Hardware-Zugriff bei SunJava über native-Calls

Abgrenzung zu SunJava

- SunJava als Dreieinigkeit
  - Sprache (Syntax, Compiler)
  - Bibliotheken (Umfang, Schnittstellen)
  - Virtuelle Maschine (Bytecode, Laufzeitsystem)
- Vollständig OS-gekapselte virtuelle Umgebung
  - SunJVM benötigt unterstützendes Betriebssystem
  - Programme sind Betriebssystem-unabhängig
  - JIT-Compiler seit SunJDK 1.2 (1998)
- SunJava seit 2007 unter GPL

Abgrenzung im Embedded-Bereich

- SunJava Card
  - Subset der Java-Sprache und -VM
  - Applets auf Chipkarte (z.B. für Kryptographie)
- SunJava für Embedded Systems (J2SE, JRTS)
  - Grundlage für JRTS ist Solaris 10 oder RT-Posix
  - „for today's more powerful embedded systems“
- Alternativen zu SunJDK und SunJVM
  - NanoVM als VM für Atmel ATmega
  - gcj für diverse Plattformen

Zielsetzung I

- Beibehaltung der Möglichkeiten von PJC
  - Java-Syntax als Grundlage, Selbstübersetzung
  - Erzeugung eines bootfähigen Speicherabbilds
  - bidirektionale Objekte, Zeigerverwaltung
- Modularer Aufbau
  - Austauschbarkeit des Backends
- Unabhängigkeit vom Wirtssystem
  - keine Verwendung vorhandener Bibliotheken
  - Cross-Compilation auch für fremde Zielsysteme

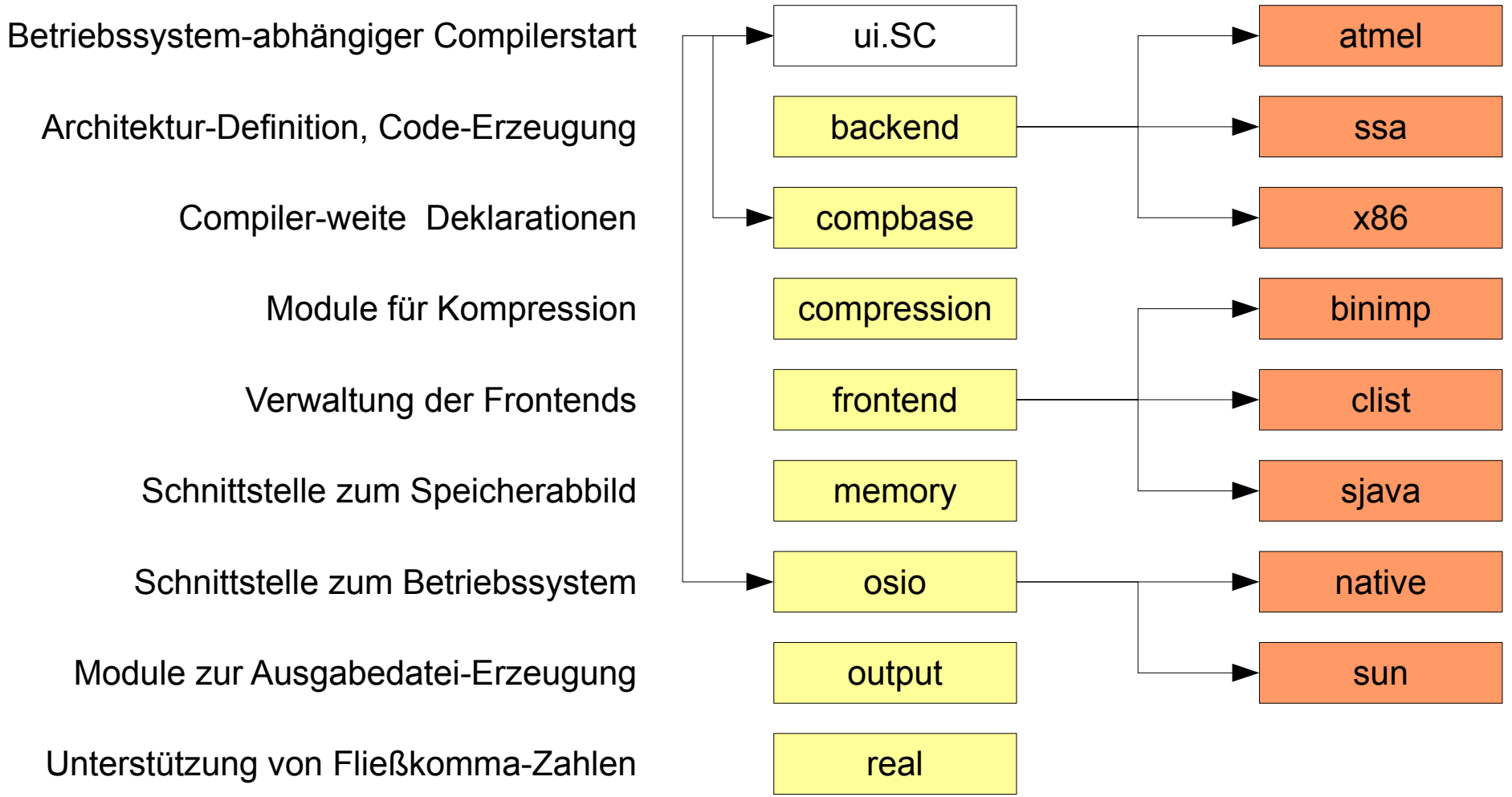
Zielsetzung II

- Schlank, vermittelbar, erweiterbar
  - Entfernung nicht benötigter Sprachelemente
  - nachvollziehbare Phasen bei der Übersetzung
  - schlanke Schnittstellen zwischen Modulen
- Möglichkeit zur Code-Optimierung
  - Backend unabhängig vom Parse-Tree
  - Backend-Schnittstelle als Zwischendarstellung
- Theoretische Maschine zur Code-Überprüfung
  - Zwischendarstellung direkt in Emulator lauffähig

SJC-Java

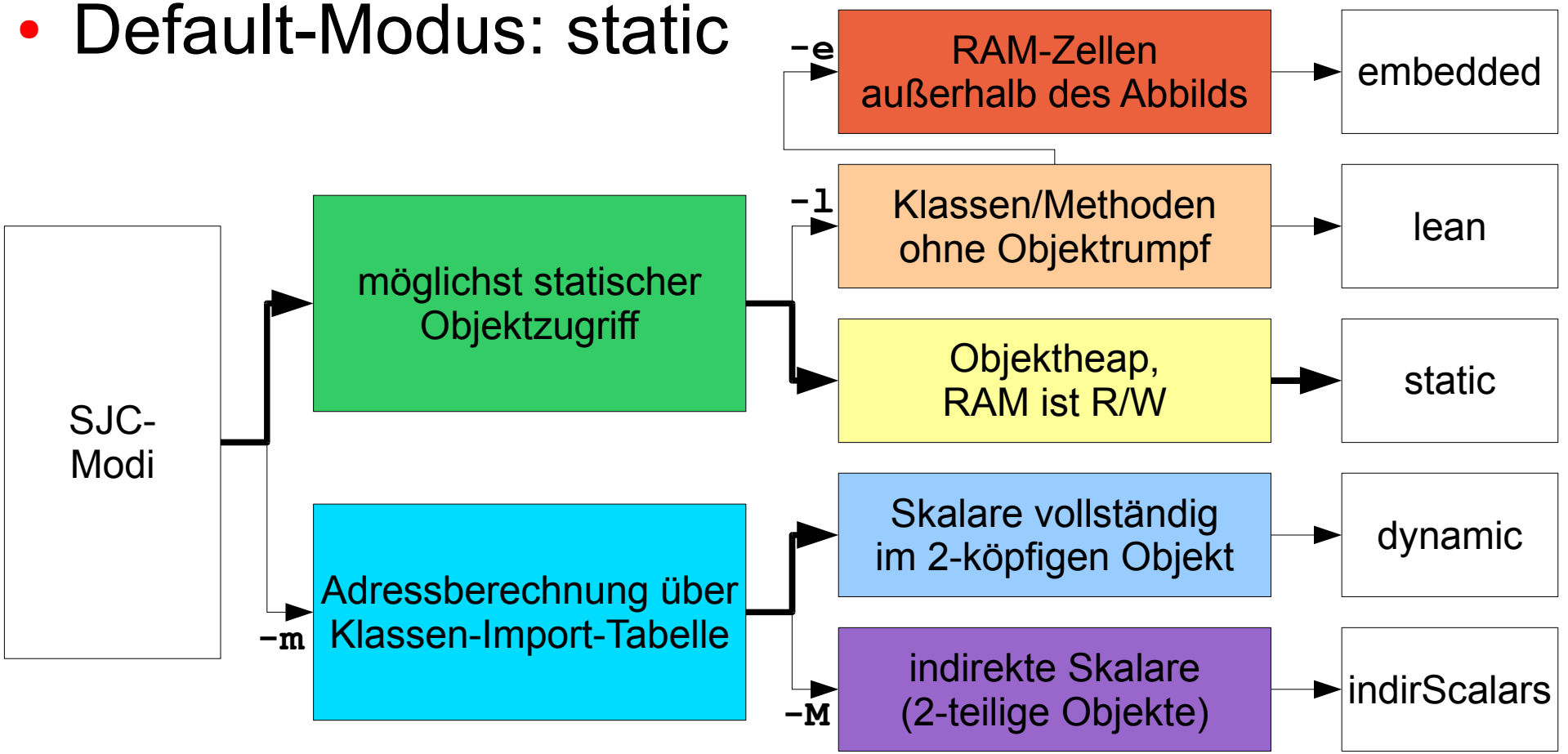
- SunJava 1.4 als Basis
- Modifikation weniger Elemente
  - keine impliziten Basistyp-Konvertierungen
  - keine Exceptions oder Errors (try, catch, finally)
- Erweiterung für Systemprogrammierung
  - Klasse MAGIC: direkter Systemzugriff
  - Klasse MARKER: Compiler-Direktive
  - Klasse STRUCT: Speichermapping
- Flexibles Laufzeitsystem

SJC-Modul-System



Übersetzungsmodi

- Optionen zum Umschalten: -e, -l, -m, -M
- Default-Modus: static



Übersetzungsbeispiel: Quellen

- Hello-World

```
package kernel;
public class Kernel {
  private static int vidMem=0xB8000;
  public static void main() {
    print("Hello World"); while(true);
  }
  public static void print(String s) {
    for (int i=0; i<s.length(); i++) print(s.charAt(i));
  }
  public static void print(char c) {
    MAGIC.wMem8(vidMem++, (byte)c);
    MAGIC.wMem8(vidMem++, (byte)0x07);
  }
}
```

- Minimales Laufzeitsystem (~60 Zeilen)

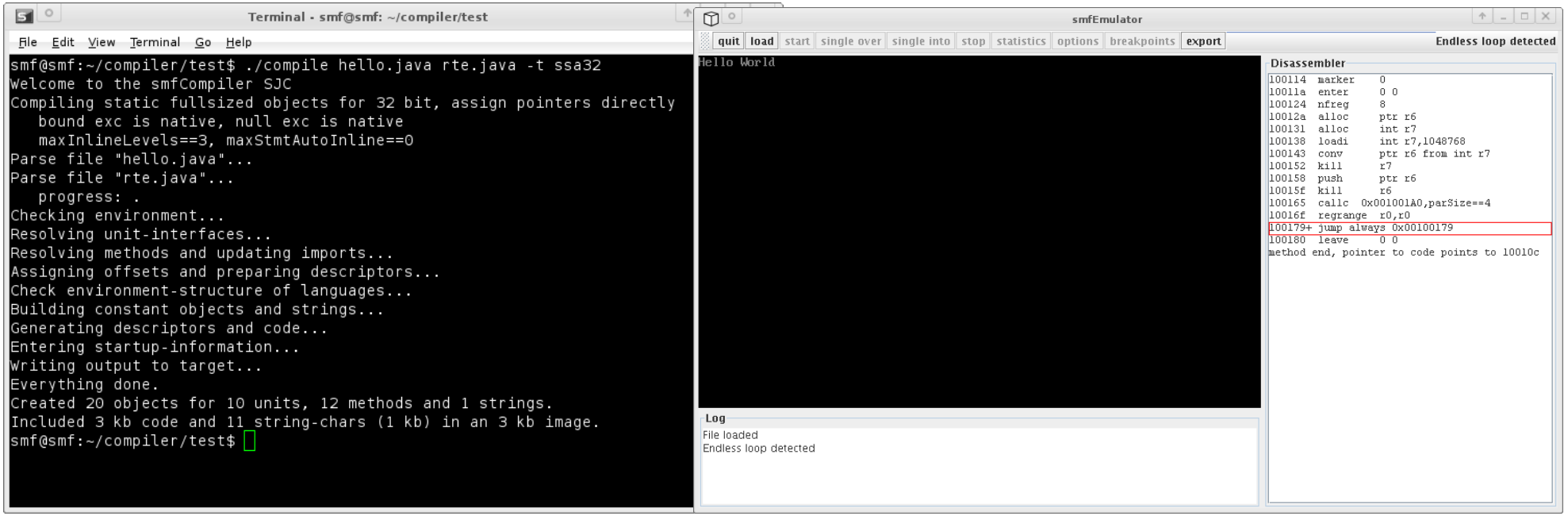
Übersetzungsbeispiel: SSA

- Kommandozeile mit nativem Compiler für SSA

```
./compile hello.java rte.java -t ssa32
```

- Aufruf des SSA-Emulators

```
java ui.Emulate raw_out.bin
```



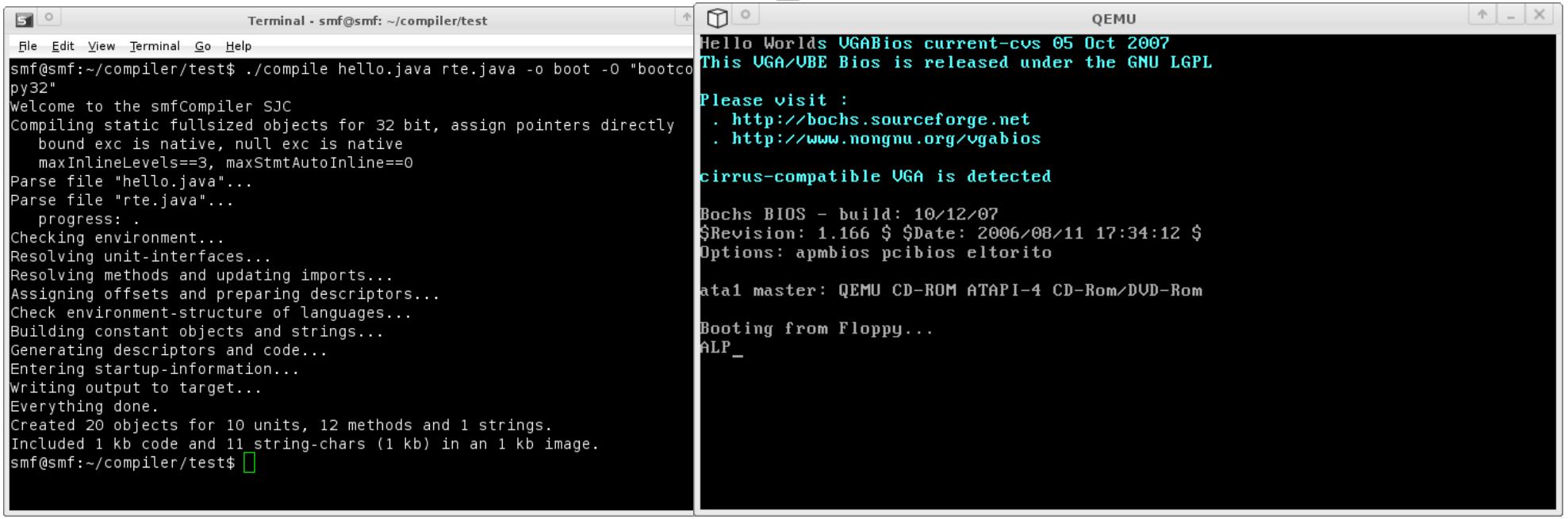
Übersetzungsbeispiel: ia32 Bootdisk

- Kommandozeile mit nativem Compiler für ia32

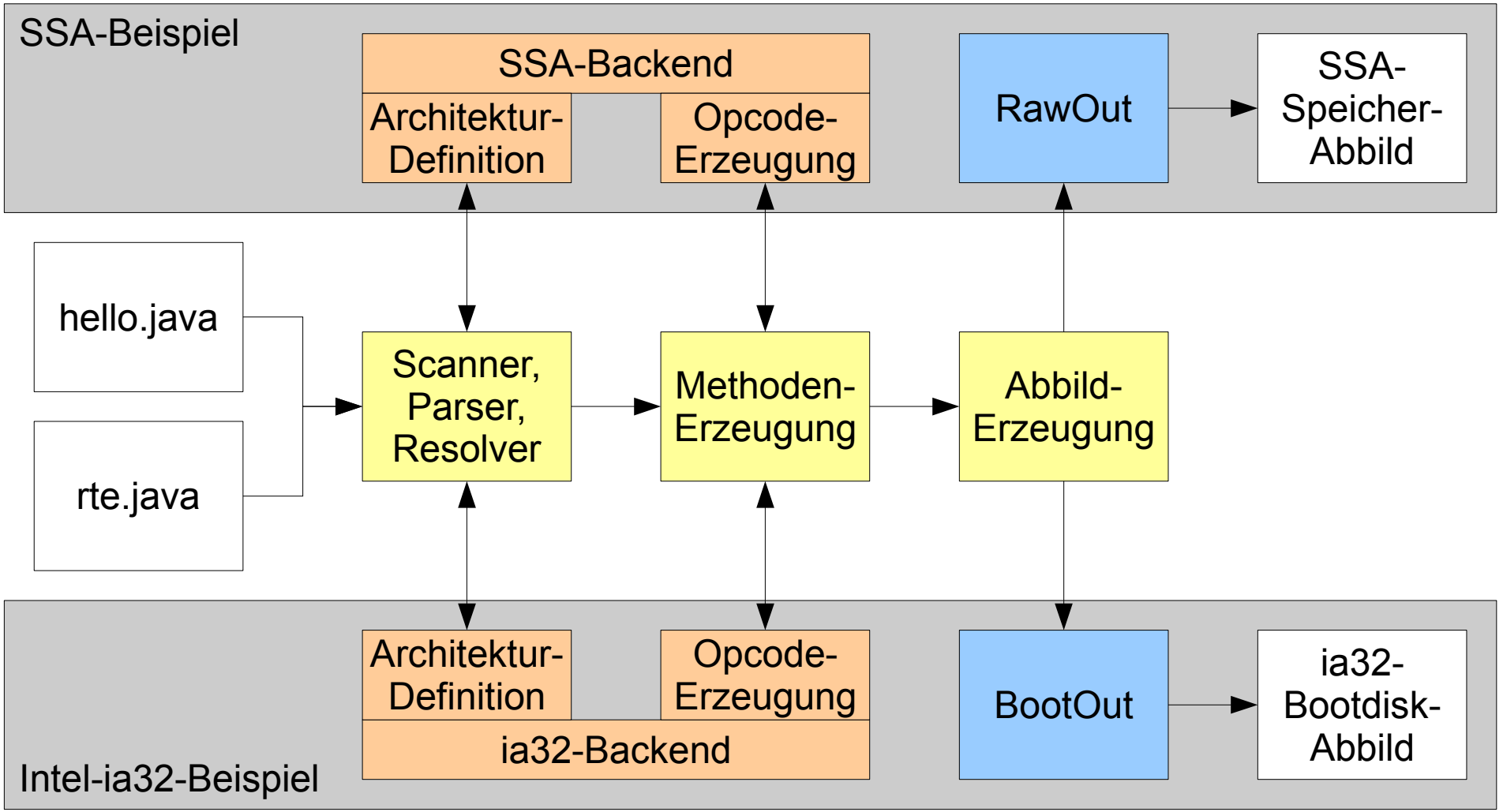
```
./compile hello.java rte.java -o boot -O "bootconf#floppy32"
```

- Aufruf von QEmu als Emulator

```
qemu -m 2 -boot a -fda BOOT_FLP.IMG
```



Übersetzungsbeispiel: Datenfluss



Aktuelle Implementierung: Module

- Version 164pre Stand 15.3.2008
- Backend
  - Atmel ATmega
  - (Pseudo-)SSA als abstrakte Maschine
  - Intel ia32 (x86-16, x86-32), AMD x64 (x86-64)
- Zielplattformen
  - ATmega (Hex-File)
  - PicOS 32, PicOS 64 (bootfähiges Image)
  - Windows 32, Linux 32, Linux 64 (ausführbare Datei)

Aktuelle Implementierung: Umfang

- Version 164pre Stand 15.3.2008
- <30k Zeilen für Compiler
  - 11391 Zeilen für alle Backends
  - 11762 Zeilen für alle Frontends
  - 6876 Zeilen für allgemeine Funktionen
- <10k Zeilen für Pseudo-SSA-Emulator
  - „Graphik“ als 80x25-EGA-Text
  - Oberfläche mit Disassembler
- 250 Zeilen für osio.sun

Aktuelle Implementierung: Natives

- Version 164pre Stand 15.3.2008
- 31879 Zeilen für Linux32-Executable
  - Laufzeitumgebung ohne Garbage-Collector
  - vollständige Compiler-Quellen ohne Emulator
  - Make in <2s auf P4 Mobile 1.9 GHz (von 2002)
  - Make in ~840ms auf C2Duo 3 GHz (SingleThread)
  - Ausgabedatei 409k Standard, 367k optimiert
  - ELF-Executable ohne Bibliotheksbedarf
  - Verwendung von Kernel-Aufrufen

Beispiel: Intel ia32 (x86)

- Von-Neumann-Architektur
  - Daten und Code gemeinsam, getrennt von I/O
  - verschiedene Befehle für Zugriff auf RAM und I/O
- 32 Bit Register und Speicherbus
- CISC-Rechner mit 4-8 „GP“-Register
  - EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP
- Memory Management Unit (Paging, Segmente)
- Erweiterungen bei Pentium / amd64 (x86-64)

Pros

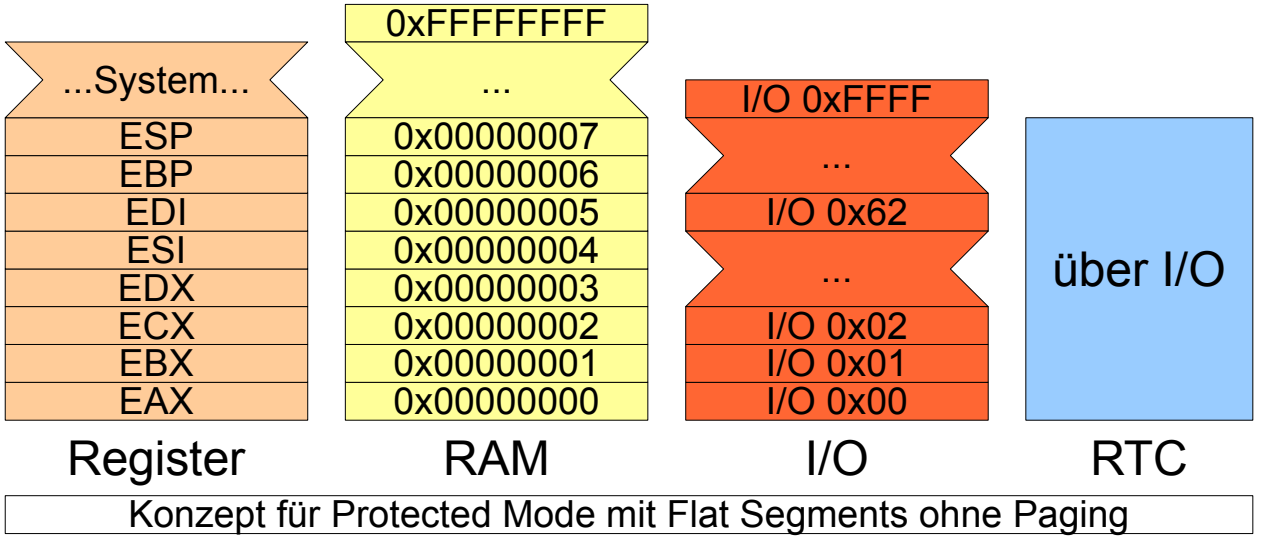
- Weite Verbreitung bei Desktop-Anwendungen
  - de Facto Standard (80386-Vorstellung 1985)
- Betriebssysteme mit bekannter API
  - Microsoft Windows, Linux
- Erweiterungsmöglichkeiten
  - ISA- und PCI-Bus, viele Anbieter
- Kostenlose Entwicklungsumgebungen
  - keine zusätzlichen Kosten, diverse Sprachen
- RAM und Festspeicher groß und sehr günstig

Cons

- Nur bedingte Eignung für Embedded Systeme
  - teilweise komplexe Peripherie erforderlich
  - hoher Stromverbrauch
  - hohe Kosten
- Unsymmetrischer CISC-Befehlssatz
  - Befehle mit Register-Einschränkung
  - wenige General Purpose Register
- Generationsinkompatibilitäten

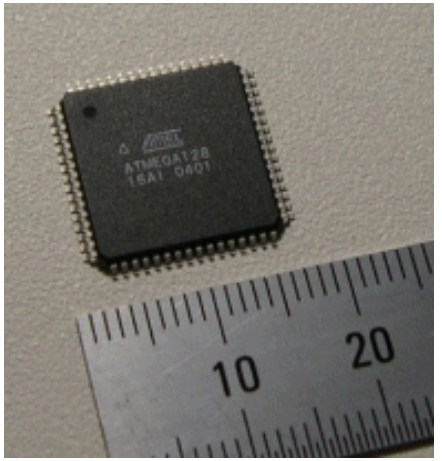
Speicherkonzept ia32

- Code und Daten im gemeinsamen RAM
- RAM linear 1-, 2- und 4-Byte adressierbar
- RAM-Mapping für BIOS und Erweiterungen
- I/O-Raum für Systemchips und Erweiterungen
- I/O langsam



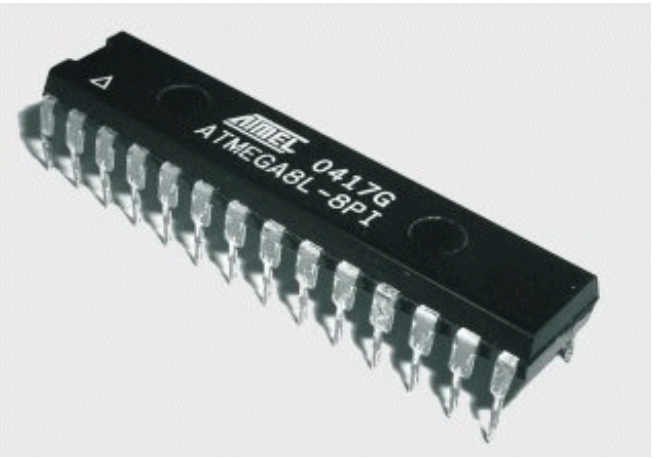
Beispiel: ATmega

- Harvard-Architektur
  - getrennte Adressräume für Daten, Code und I/O
  - verschiedene Befehle für Zugriff auf RAM und Flash
- Breites Toleranzband für Spannung und Takt
- 16 Bit interner Adressbus für Flash-Zugriff
  - durch Erweiterungsregister bis 24 Bit
- RISC-Rechner mit 32 GP-Register
  - 8 Bit Register 0-15 und 16-31
  - Zeiger in 26/27, 28/29, 30/31



Pros

- Weite Verbreitung im Embedded Bereich
  - Community-Unterstützung
- Niedriger Preis
  - 1.45 bis 14.30 €
- Sehr viele mögliche Optionen
  - ISP, SPI, TWI, CAN, USB, USART
- Kostenlose Entwicklungsumgebung
  - keine zusätzlichen Kosten, C und Assembler
- Fuse-Bits für Konfiguration und Datenschutz

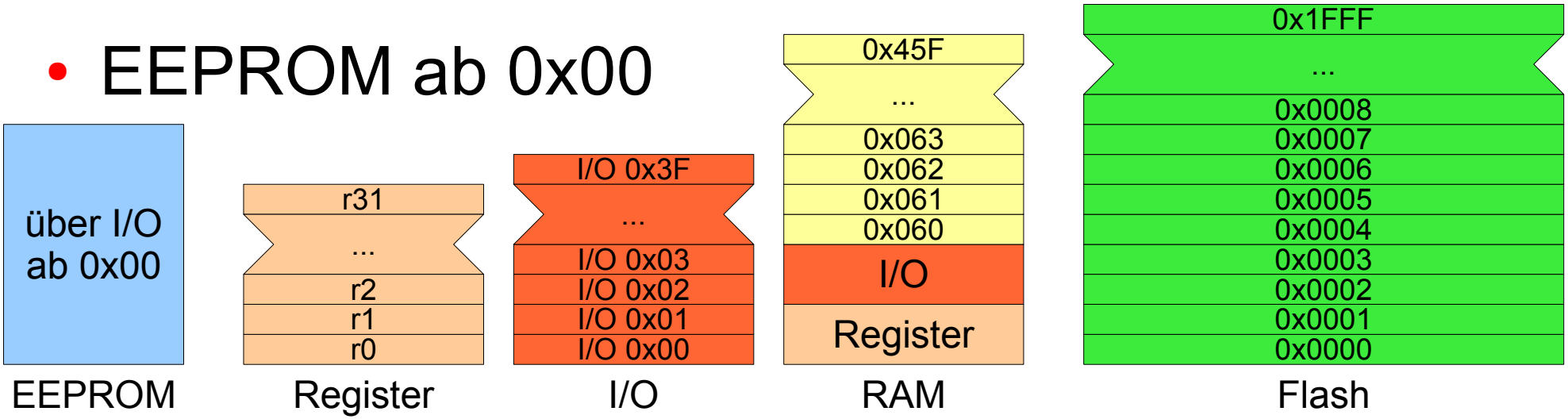


Cons

- Harvard-Architektur
  - Code-Reusage ggf. erschwert
  - Konstanten ggf. im RAM repliziert
- Eingeschränkter RISC-Befehlssatz
  - Sprungbefehle für „<=“ und „>“ fehlen
  - kurze Entfernung bei bedingten Sprüngen
  - keine Unterstützung für Division und Modulo
  - Registereinsatzmöglichkeiten nicht symetrisch
- Vergleichsweise erhöhter Codespeicherbedarf

Speicherkonzept ATmega

- CPU-Register im RAM gespiegelt
- I/O bei 0x00, in RAM ab 0x20 gespiegelt
- Tatsächliches SRAM startet nach Register+I/O
- Code im Flash startet bei 0x0000
- EEPROM ab 0x00



Konzept für alle ATmega, hier Maximalwerte I/O und Flash sowie Startwert und Maximalwert SRAM für ATmega16

Zusammenfassung

- Der Small Java Compiler
  - ist schlank und schnell
  - ist modular aufgebaut und erweiterbar
  - erzeugt Code für Plattformen von 8 bis 64 Bit
  - eignet sich zur Erstellung von Betriebssystemen
  - liefert Dateien zum Flashen, Booten und Aufrufen
- Aber SJC
  - enthält keine (SunJava-)Bibliotheken
  - erzeugt derzeit noch nicht voll optimierten Code

Vielen Dank für Ihre Aufmerksamkeit

Fragen, Diskussion, Anregungen

Bezugsquelle: http://www-vs.informatik.uni-ulm.de/dept/staff/frenz/private/
dort finden sich unter anderem Compiler, Quellcode, Handbuch, Beispiele