

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

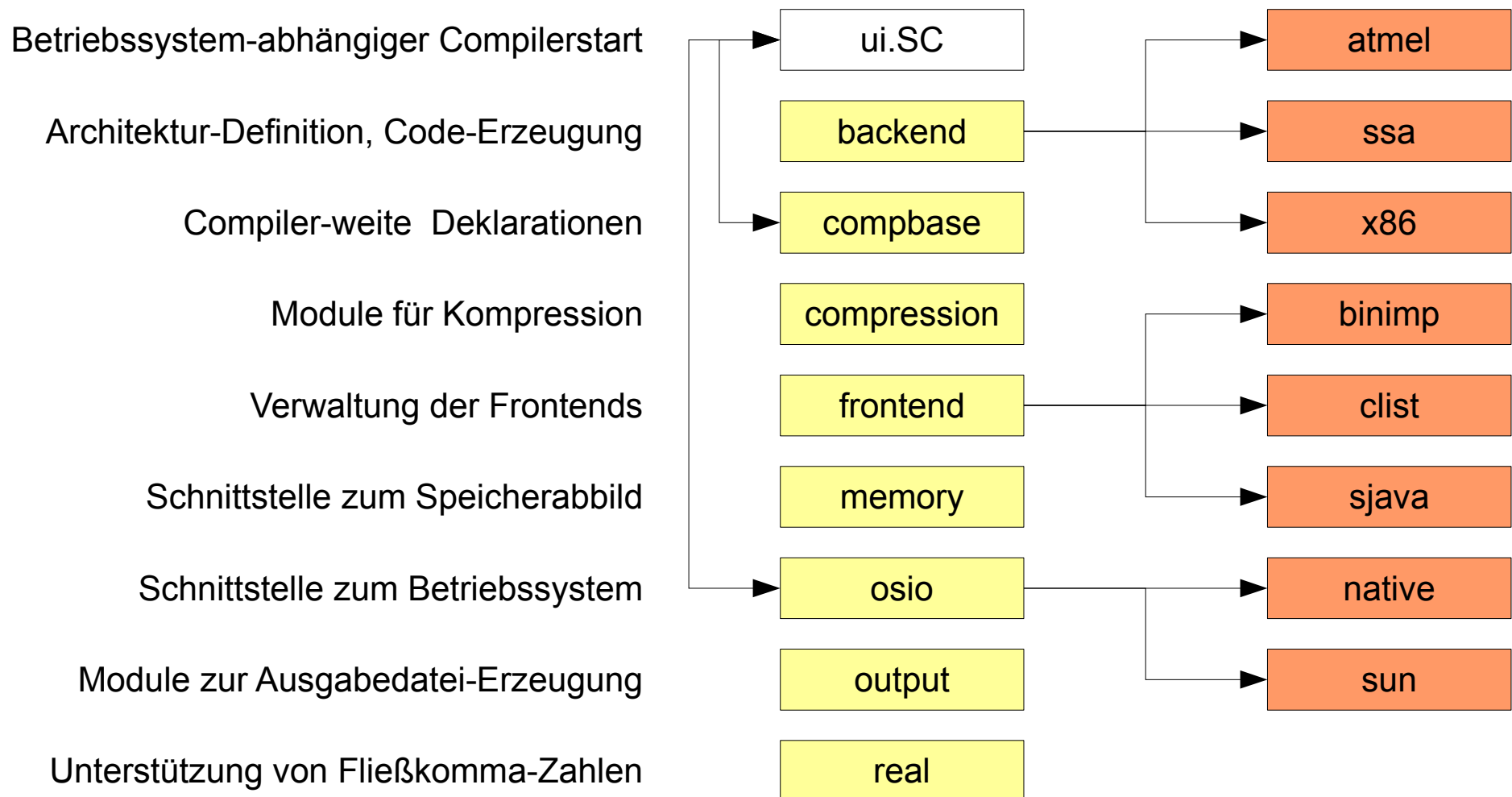
Zielsetzung II

- 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

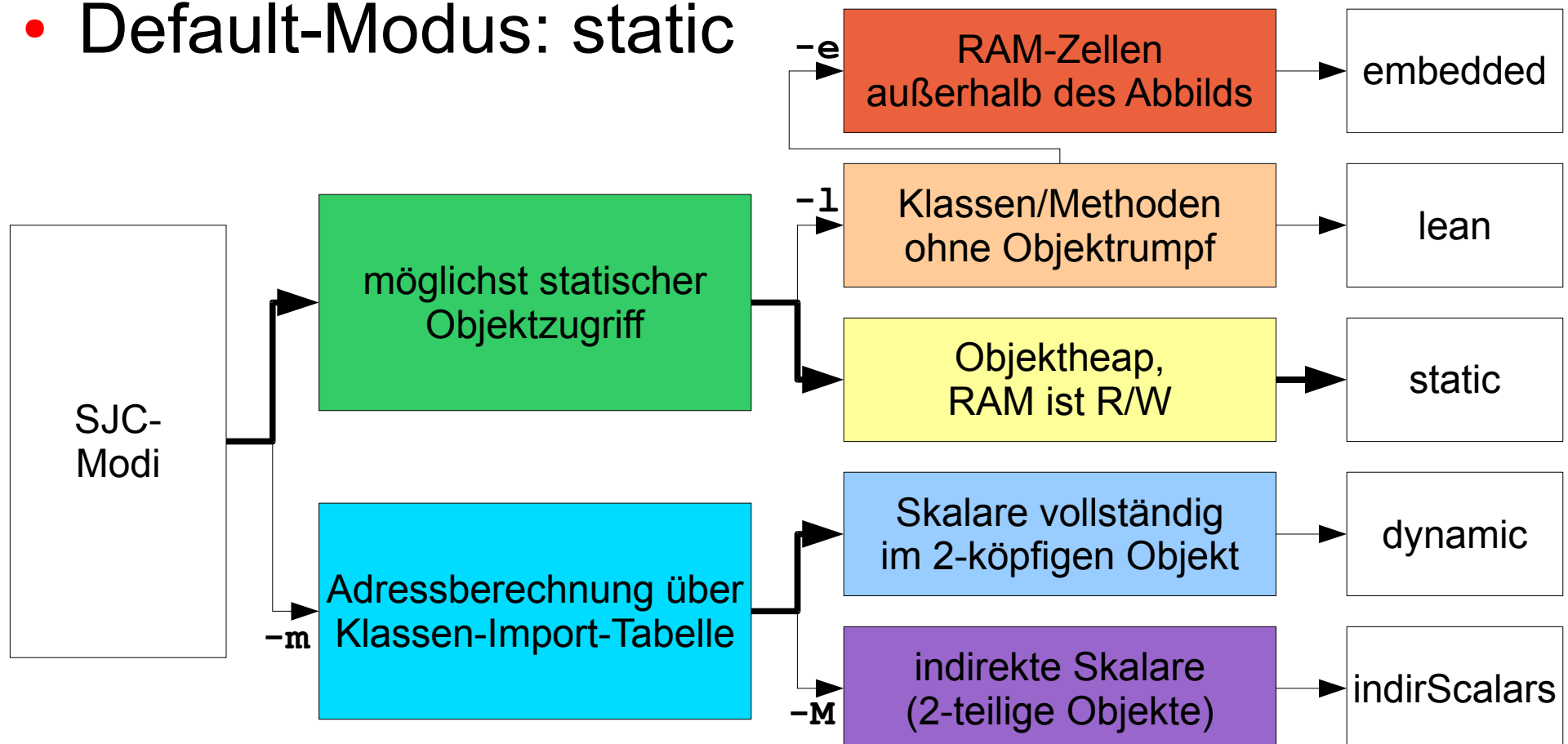
- SunJava 1.4 als Basis
- Modifikation weniger Elemente
 - keine impliziten Basistyp-Konvertierungen
 - keine Exceptions oder Errors (try, catch, finally)
- Erweiterung für Systemprogrammierung
 - Klasse MAGIC: direkter Systemzugriff
 - Klasse MARKER: Compiler-Direktive
 - Klasse STRUCT: Speichermapping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

The screenshot shows two windows. The left window is a terminal titled 'Terminal - smf@smf: ~/compiler/test'. It displays the output of the compilation command `./compile hello.java rte.java -t ssa32`. The output includes a welcome message, compilation options, and progress indicators. The right window is titled 'smfEmulator' and shows the execution of the program. The main display area shows 'Hello World'. The 'Disassembler' panel on the right shows assembly code, with the instruction `100179+ jump always 0x00100179` highlighted in red. A status bar at the top right of the emulator window indicates 'Endless loop detected'. The 'Log' panel at the bottom shows 'File loaded' and 'Endless loop detected'.

```
Terminal - smf@smf: ~/compiler/test
File Edit View Terminal Go Help
smf@smf:~/compiler/test$ ./compile hello.java rte.java -t ssa32
Welcome to the smfCompiler SJC
Compiling static fullsized objects for 32 bit, assign pointers directly
  bound exc is native, null exc is native
  maxInlineLevels==3, maxStmtAutoInline==0
Parse file "hello.java"...
Parse file "rte.java"...
  progress: .
Checking environment...
Resolving unit-interfaces...
Resolving methods and updating imports...
Assigning offsets and preparing descriptors...
Check environment-structure of languages...
Building constant objects and strings...
Generating descriptors and code...
Entering startup-information...
Writing output to target...
Everything done.
Created 20 objects for 10 units, 12 methods and 1 strings.
Included 3 kb code and 11 string-chars (1 kb) in an 3 kb image.
smf@smf:~/compiler/test$

smfEmulator
quit load start single over single into stop statistics options breakpoints export
Hello World
Disassembler
100114 marker 0
10011a enter 0 0
100124 nfreq 8
10012a alloc ptr r6
100131 alloc int r7
100138 loadi int r7,1048768
100143 conv ptr r6 from int r7
100152 kill r7
100158 push ptr r6
10015e kill r6
100165 callc 0x001001A0,parSize==4
10016f regrange r0,r0
100179+ jump always 0x00100179
100180 leave 0 0
method end, pointer to code points to 10010c
Log
File loaded
Endless loop detected
```

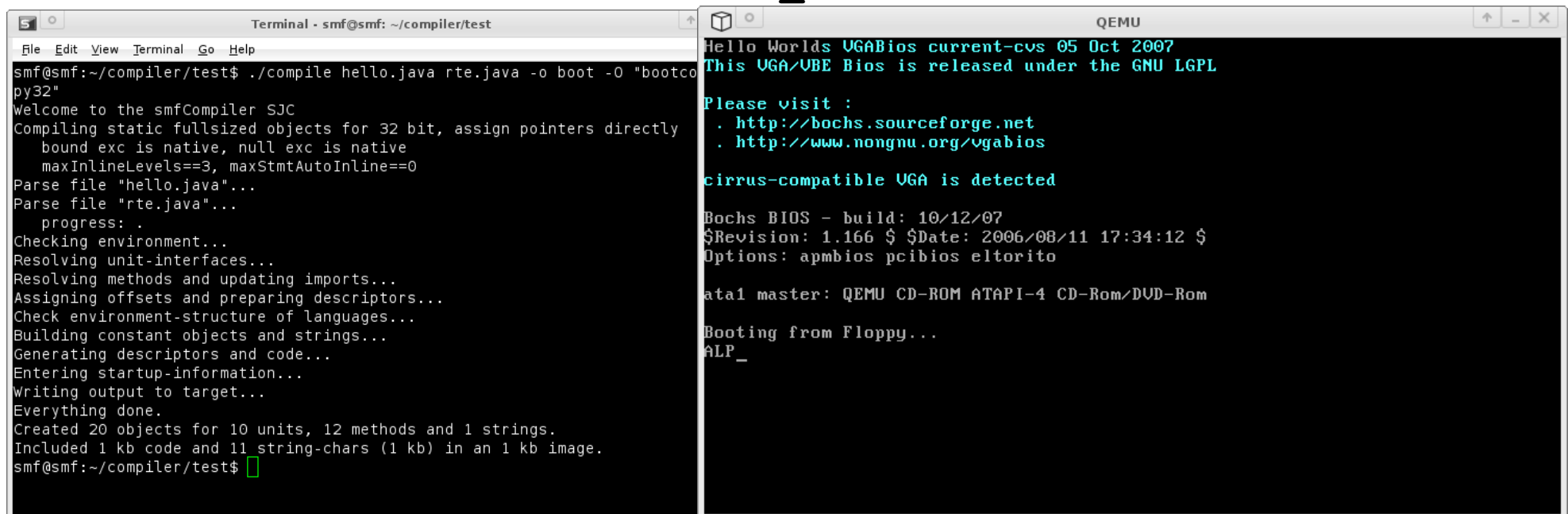
Übersetzungsbeispiel: ia32 Bootdisk

- Kommandozeile mit nativem Compiler für ia32

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

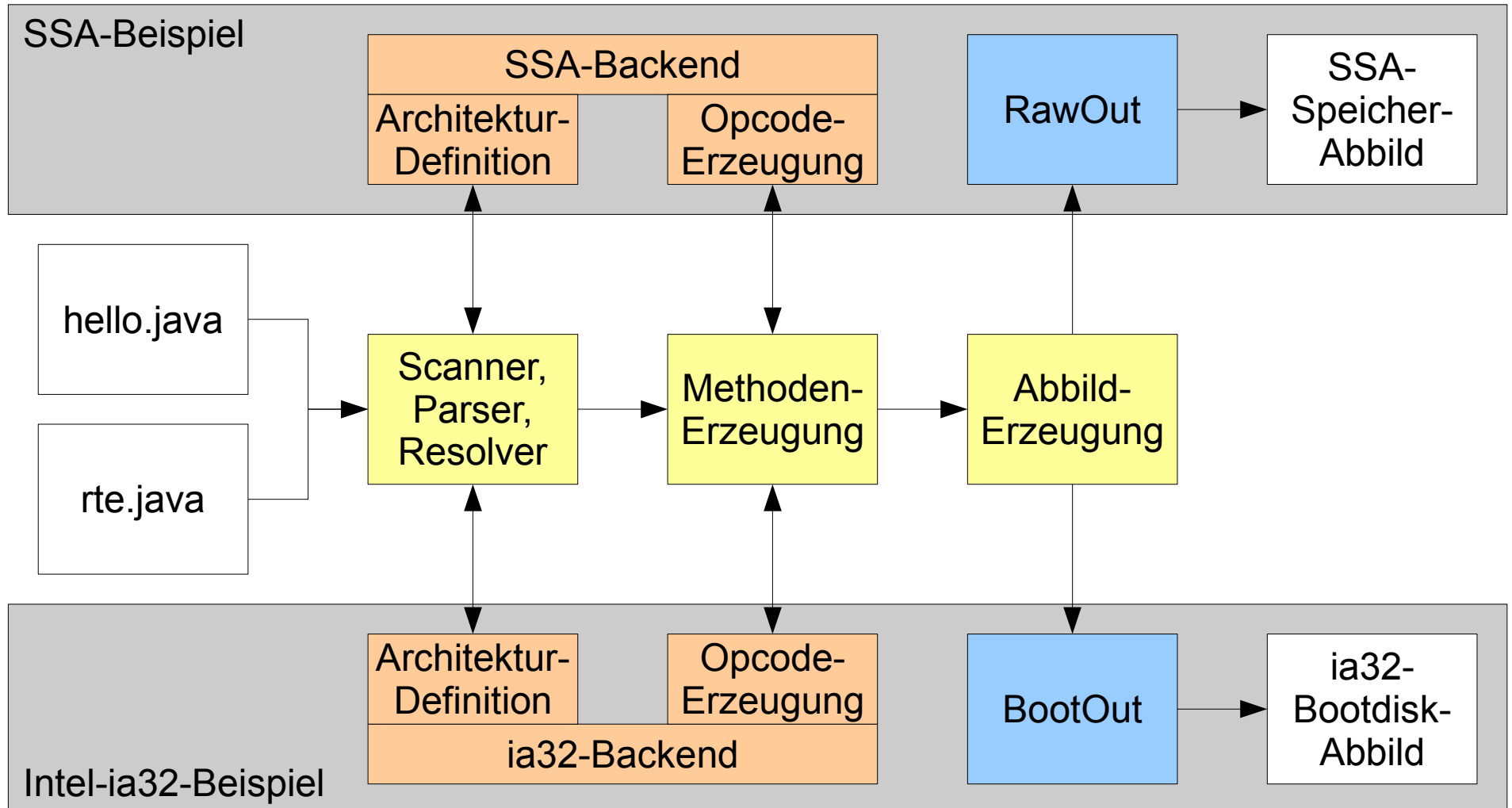
- Aufruf von QEmu als Emulator

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



The image shows two terminal windows side-by-side. The left window, titled 'Terminal - smf@smf: ~/compiler/test', displays the output of the compilation command. It shows the compiler's progress, including parsing files, checking the environment, and generating code. The output ends with 'smf@smf:~/compiler/test\$' and a cursor. The right window, titled 'QEMU', shows the output of the emulator. It displays the BIOS boot sequence, including the 'Hello Worlds UGABios' message, the GNU LGPL license notice, and the detection of a 'cirrus-compatible UGA'. The boot sequence continues with 'Bochs BIOS - build: 10/12/07', '\$Revision: 1.166 \$ \$Date: 2006/08/11 17:34:12 \$', and 'Options: apmbios pcibios eltorito'. The boot sequence ends with 'ata1 master: QEMU CD-ROM ATAPI-4 CD-Rom/DVD-Rom' and 'Booting from Floppy... ALP_'. The cursor is at the end of the output.

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

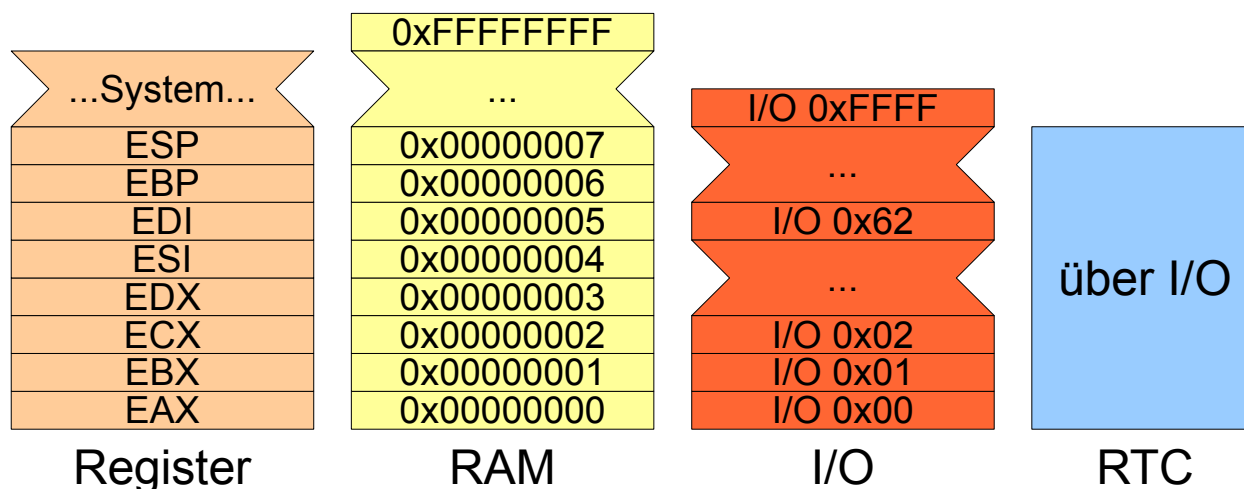


Cons

- 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

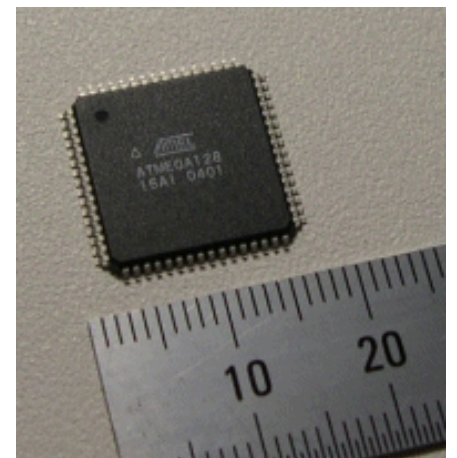
- Code und Daten im gemeinsamen RAM
- RAM linear 1-, 2- und 4-Byte adressierbar
- RAM-Mapping für BIOS und Erweiterungen
- I/O-Raum für Systemchips und Erweiterungen
- I/O langsam



Konzept für Protected Mode mit Flat Segments ohne Paging

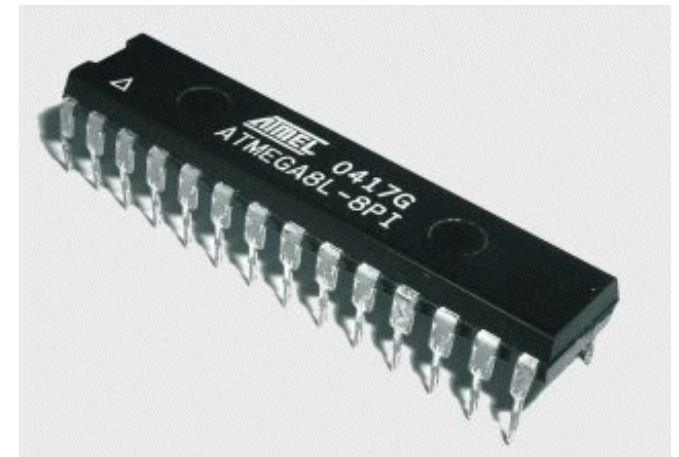
Beispiel: ATmega

- 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

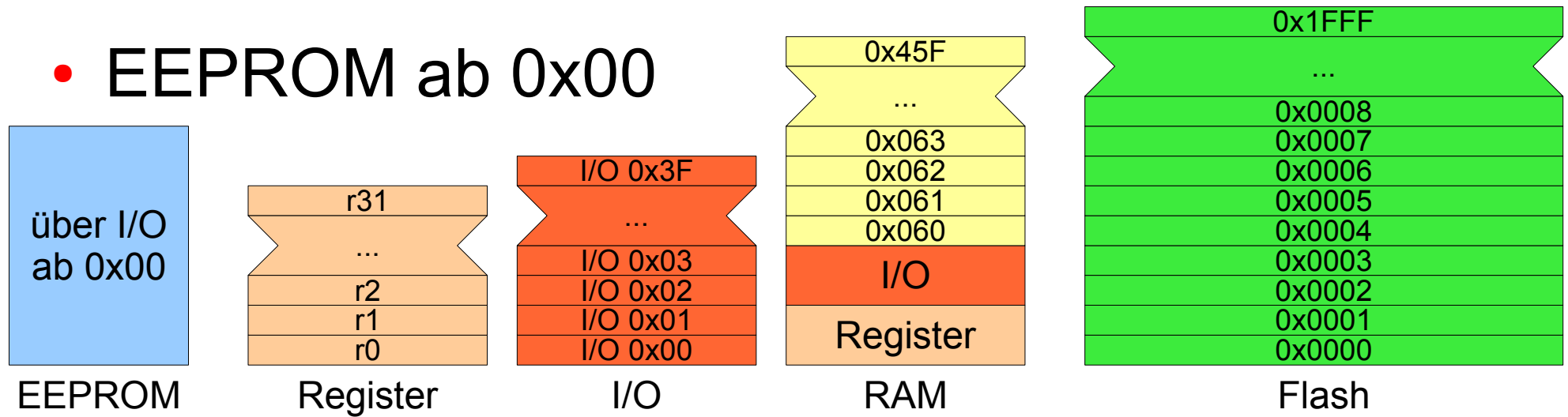


Cons

- 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 symmetrisch
- 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