clang:
der andere Compiler

Hans Buchmann FHNW/IME

30. März 2015

Ein anderer (als *gcc*): *neuer* Compiler

Warum

- gcc GNU Compiler Collection
  - relativ alt
  - etwas verrostet
  - schwierig anzupassen
  - GPL Lizenz für Firmen nicht immer optimal

Remark: GCC war (und ist) wichtig

Andere Compiler

- `http://en.wikipedia.org/wiki/List_of_compilers`

Remark: es gibt viele

Ein paar Begriffe

HLL **H**igh **L**evel programming *L*anguage
- C, C++, Fortran, Java

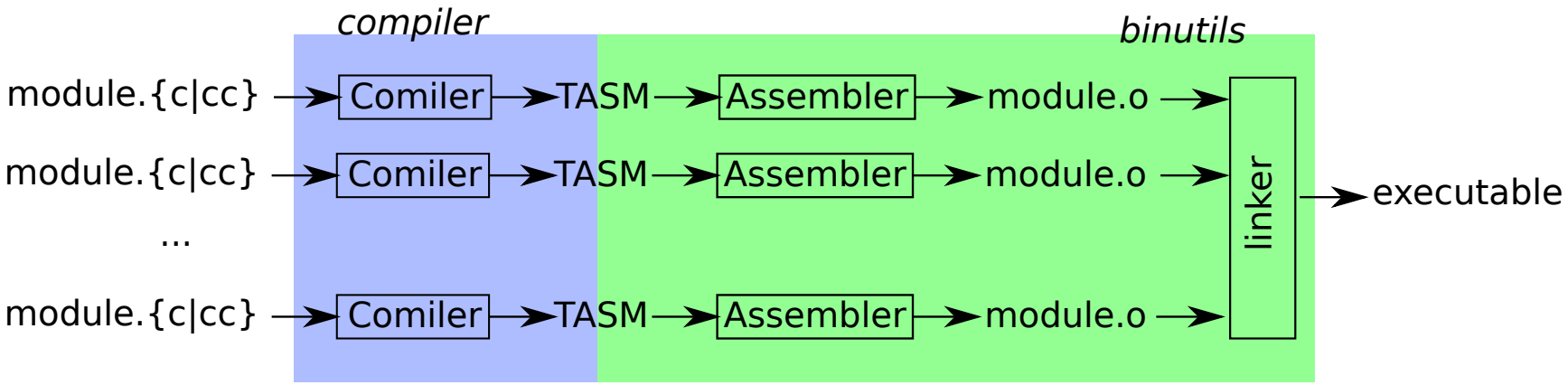
Target Rechnerarchitektur
- ARM, x86, x86_64

TASM **T**arget **As**S**e**M**bler**
- Maschineninstruktionen für ein *Target*

ObjectFile Der Maschinencode binär verpackt
- Ein Format ELF: **E**xecutable and **L**inkable **F**ormat

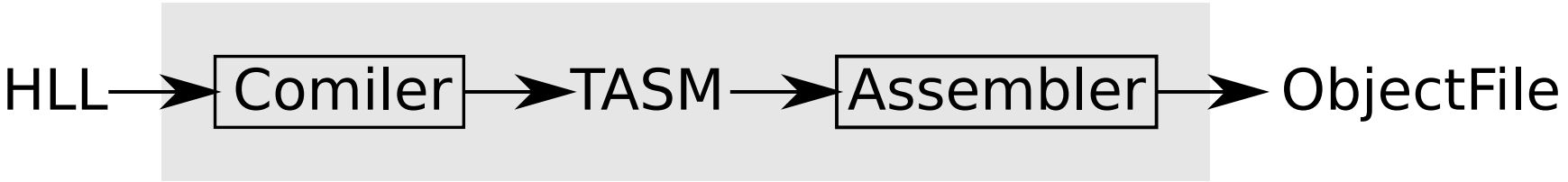
Module→Projekt

Viele Files bilden ein Projekt



- `gcc -v hello-world.c -o hello-world`

Die *CompileChain*



- `gcc -c hello-world.c -o hello-world.o`
  - der normale Fall
- `gcc -v -c hello-world.c -o hello-world.o`
  - **-v** was passiert genau

Remark(s):

- Input: 1 HLL File, Output 1 ObjectFile
- Der Assembler wird automatisch aufregrufen

→ unser Fokus: der Compiler

Der Compiler

Input *HLL* (praktisch) unabhängig vom *Target*

Output *TASM* praktisch unabhängig von der *HLL*

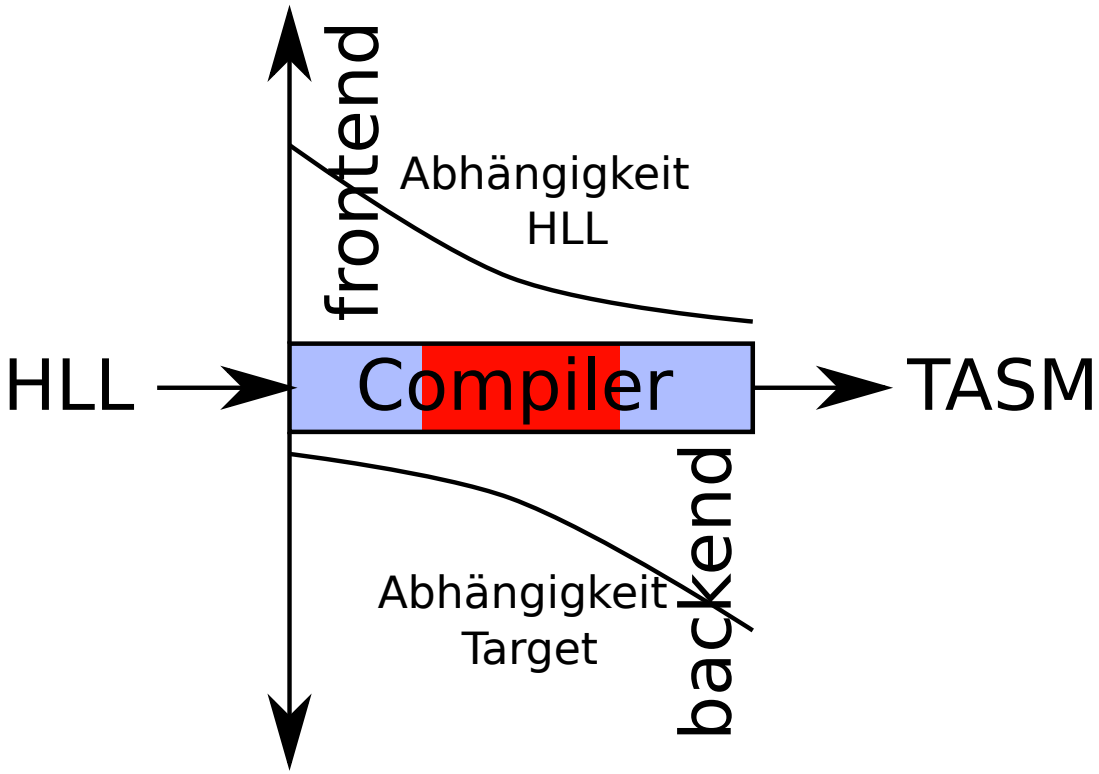
Anforderung *HLL* und *TASM* sollen **semantisch** gleich sein

Remark(s):

- *TASM* kann schon in binärer Form vorliegen

Aufbau

gilt für fast alle Compiler



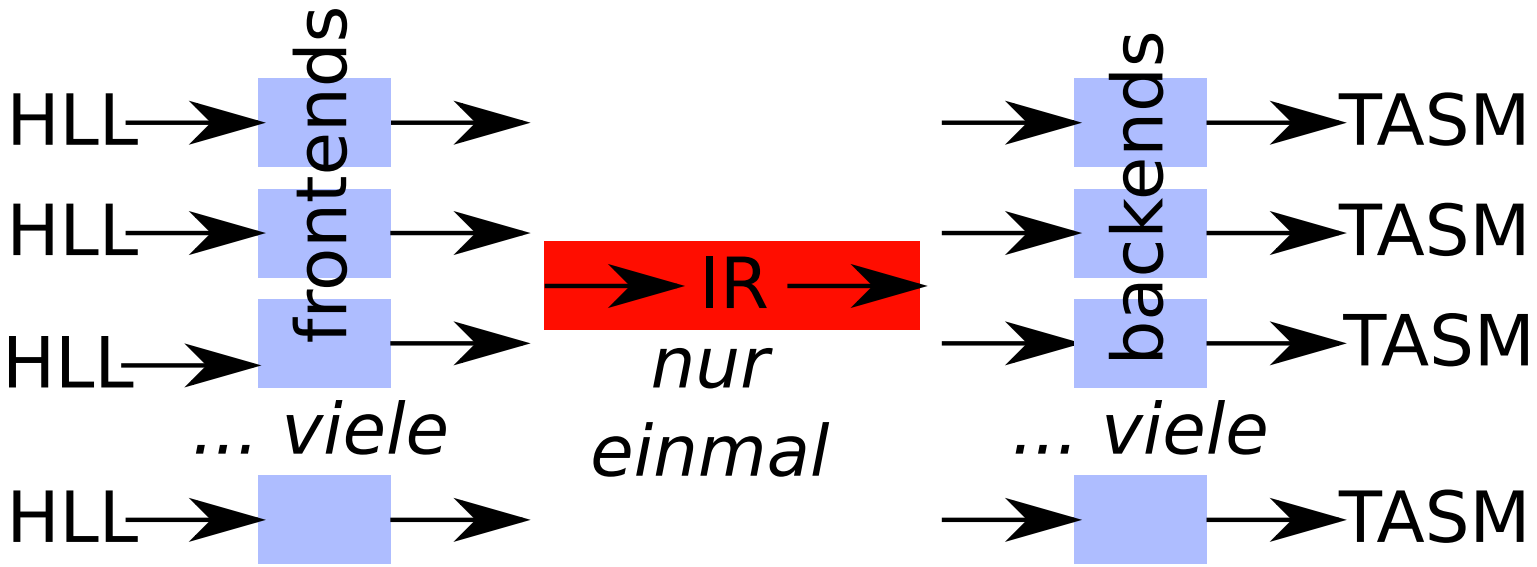
frontend unabhängig vom *Target*

backend unabhängig von der *HLL*

- Was liegt dazwischen ?

IR: Intermediate Representation

Vorteil



Was ist eine gute IR

IR

Ein paar Beispiele

ByteCode `www.oracle.com/technetwork/java/index.html`

HLL Java, Scala

TASM fast alle Platformen (Targets)

GENERIC `https://gcc.gnu.org`

HLL `https://gcc.gnu.org/frontends.html`

TASM `https://gcc.gnu.org/backends.html`

IR

LLVM `llvm.org`

HLL Ada, C, C++, D, Fortran,Objective-C [1]

TASM `llvm-config --targets-built`

[1] `en.wikipedia.org/wiki/LLVM`

Übersicht

- wie gcc
- Intermediate Representation
  - *HLL*→*IR*
  - *IR* →*TASM*
  - *IR* Virtual Machine

gcc like
die gleichen Optionen

direct `clang hello-world.c -o hello-world`

direct verbose `clang -v hello-world.c -o hello-world`

compile-link für Module

compile
```
clang -c hello-world.c\
    -o hello-world.o
```

link
```
gcc hello-world.o \
    -o hello-world
```

TASM Lesbare Maschineninstruktionen

- `clang -S hello-world.c -o hello-world.s`

Intermediate Representation

compile

```
clang -c -emit-llvm hello-world.c -o hello-world.bc
```

link

```
llvm-link hello-world.bc -o hello-world.llvm
```

interpret

```
lli c-hello-world.llvm
```

Intermediate Representation

Wie sieht er aus

disassemble `llvm-dis hello-world.bc -o -`

- nach `stdout`

clang
```
clang -S -emit-llvm hello-world.c \
-o hello-world.llvm
```

Warum `llvm/clang`

- Übersichtliche Aufteilung *front-/backend*
  - Eigene *front-/backends* sind möglich

  Remark: Auch mit `gcc` möglich aber viel unübersichtlicher
- Klar dokumentierte *Intermediate Representation*
- Modularer Aufbau der Software geschrieben in C++
- Grosse Firmen - Apple/Google - steht dahinter

Aufgaben

Installation als source oder als package

RaspberryPi `clang` als Compiler

Vergleich `gcc` vs. `clang` Beispiel `prime-number.cc`