



## ■ **Hybrid-Vorlesung mit Aufnahme**

- Die Aufnahme ist anschließend in Stud.IP verfügbar
- Nutzen Sie die Gelegenheit zur Live-Veranstaltung!

## ■ Wir nehmen auf

- Folien, Dozent, Live-Audio sowie BBB-Audio
- **Ihre Stimme** beim Fragen und Sprechen
- **Durch aktive Teilnahme erklären Sie sich einverstanden!**

## ■ Fragen: Live, im Chat, Sprechen in der BBB-Sitzung



Technische  
Universität  
Braunschweig



Institute of Operating Systems  
and Computer Networks  
Reliable System Software



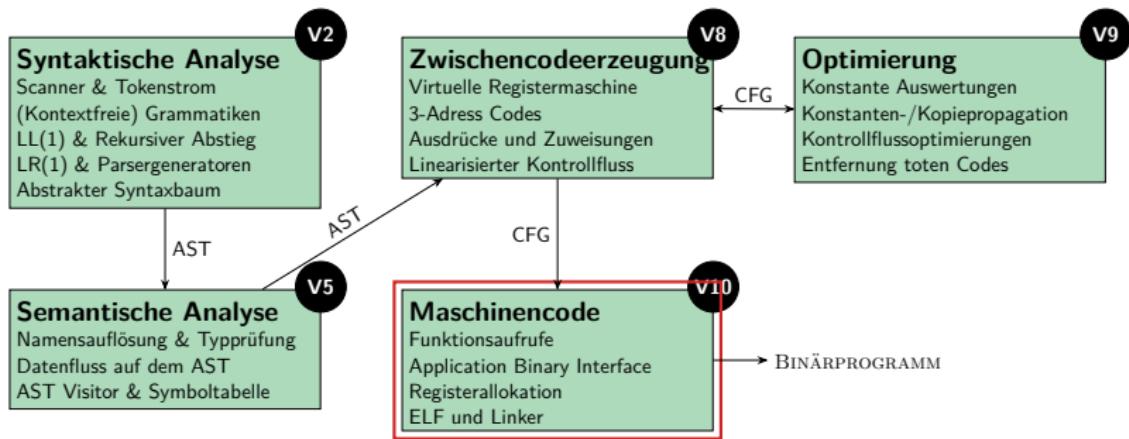
# Programmiersprachen und Übersetzer

10 - Maschinencode

Christian Dietrich

Sommersemester 2024

# Einordnung in die Vorlesung: Maschinencodeerzeugung



- Zwischencode der virtuellen Maschine auf eine **reale Maschine** abbilden
  - Verbleibende **semantische Lücken**: Funktionsaufrufe, lokale Variablen, Befehle
  - Application Binary Interface (ABI), Registervergabe, Binärformat (ELF)

## Zwischencode-Maschine

```
func foo(a: int) : int
```

```
t0 := Add a, 3
t1 := Mul t0, 4
t2 := Call bar, a, t1, 19
Return t2
```

BB0

```
parameters = [a]
variables = [t0, t1, t2]
basic_blocks = {BB0}
entry_block = BB0
```

- Speicherabstraktion
  - unendlich viele Parameter
  - unendlich viele Variablen
- Befehle
  - Meist 3-Adress-Befehle
  - `Call`: Beliebige Argumentanzahl
- Speicherung des Programms
  - Objekte im Übersetzerzustand
  - Basisblöcke sind ungeordnet

## Reale Maschine (z.B. IA-32)

**foo:**

```
c8 0c 00 00 8b 45 08 bb 03 00 00 00 01 c3 89 5d
fc b8 04 00 00 00 0f af d8 89 5d f8 b8 13 00 00
00 50 53 8b 4d 08 51 e8 bb ff ff ff 83 c4 0c 89
45 f4 c9 c3
```

(Keine Sorge, wir gehen nur bis Assembler)

- Speicherabstraktion
  - 6 Ganzzahl-Register (32-Bit)
  - Endlicher Speicher (ausreichend)
- Befehle
  - 2 Operanden, zeitgleich Quelle+Ziel
  - `call` transportiert keine Argumente
  - Verschiedene Adressierungsmodi
  - Komplexe Befehle (CISC)
- Speicherung des Programms
  - Binärformat des Betriebssystems
  - Lineare Sequenz von Befehlen

## Zwischencode-Maschine

```
func foo(a: int) : int
```

```
t0 := Add a, 3
t1 := Mul t0, 4
t2 := Call bar, a, t1, 19
Return t2
```

BB0

```
parameters
variables
basic_blocks
entry_block = BB0
```

Wir verwenden IA-32  
als Beispiel

- Speicherabstraktion
  - unendlich viele Parameter
  - unendlich viele Variablen
- Befehle
  - Meist 3-Adress-Befehle
  - `Call`: Beliebige Argumentanzahl
- Speicherung des Programms
  - Objekte im Übersetzerzustand
  - Basisblöcke sind ungeordnet

## Reale Maschine (z.B. IA-32)

**foo:**

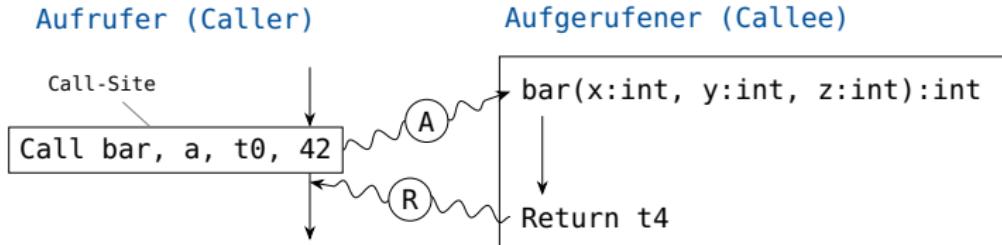
```
c8 0c 00 00 8b 45 08 bb 03 00 00 00 01 c3 89 5d
fc b8 04 00 00 00 0f af d8 89 5d f8 b8 13 00 00
00 50 53 8b 4d 08 51 e8 bb ff ff ff 83 c4 0c 89
45 f4 c9 c3
```

(Keine Sorge, wir gehen nur bis Assembler)

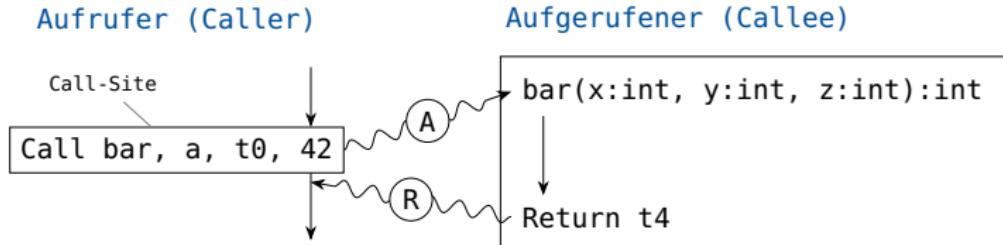
- Speicherabstraktion
  - 6 Ganzzahl-Register (32-Bit)
  - Endlicher Speicher (ausreichend)
- Befehle
  - 2 Operanden, zeitgleich Quelle+Ziel
  - `call` transportiert keine Argumente
  - Verschiedene Adressierungsmodi
  - Komplexe Befehle (CISC)
- Speicherung des Programms
  - Binärformat des Betriebssystems
  - Lineare Sequenz von Befehlen



# Speicherabstraktion: Call-Frames



- **Funktionsaufruf:** Argumente werden als Parameter übertragen
  - IR-Ebene: Beliebig viele Parameter kommen „magisch“ beim Callee an
  - **Erinnerung:** Aufrufe erzeugen **Funktionsinstanz mit eigenem Call-Frame**
  - Call-Frame enthält Parameter, Rücksprungadresse und lokale Variablen



- **Funktionsaufruf:** Argumente werden als Parameter übertragen
    - IR-Ebene: Beliebig viele Parameter kommen „magisch“ beim Callee an
    - **Erinnerung:** Aufrufe erzeugen **Funktionsinstanz mit eigenem Call-Frame**
    - Call-Frame enthält Parameter, Rücksprungadresse und lokale Variablen
  - Für die reale Maschine müssen wir einige **Entscheidungen** treffen:
    - **Datenlayout:** Welches Datum liegt an welcher Stelle?
    - **Verantwortung:** Wer legt den Call-Frame an und füllt ihn mit Daten?
    - **Invarianten:** Welche Teile des Maschinenzustandes bleiben über einen Funktionsaufruf hinweg erhalten?
- ⇒ Die Antworten hierauf ergeben die **Aufrufkonvention**.

Definition: Die Aufruf**konvention** bestimmt,...

...wie **Argumente** an Funktionen übergeben werden und  
...welche Teile des **Maschinenzustands** der Callee erhalten muss und  
...wo der Caller den **Rückgabewert** auslesen kann.

- Das Betriebssystem und Übersetzer bestimmen die Aufrufkonvention
  - Funktionen können unabhängig voneinander übersetzt werden.
  - Interoperabilität zwischen verschiedenen Übersetzern und Sprachen

Definition: Die Aufruf**konvention** bestimmt,...

- ...wie **Argumente** an Funktionen übergeben werden und
- ...welche Teile des **Maschinenzustands** der Callee erhalten muss und
- ...wo der Caller den **Rückgabewert** auslesen kann.

- Das Betriebssystem und Übersetzer bestimmen die Aufrufkonvention
  - Funktionen können unabhängig voneinander übersetzt werden.
  - Interoperabilität zwischen verschiedenen Übersetzern und Sprachen
- Beispiel: Sys-V Calling Convention für C auf IA-32/GNU Linux
  - Callee legt Call-Frame auf dem Stack an und räumt ihn wieder weg
  - Argumente werden von **rechts-nach-links** auf den Stack gelegt
  - Rückgabewerte: `%eax` für **int**, `%st0` für **double**  
Größere Argumente/Rückgabewerte wie in Vorlesung 8
  - **Callee-saved Register**: `%esp`, `%ebp`, `%ebx`, `%esi`, `%edi`

Definition: Die Aufruf**konvention** bestimmt,...

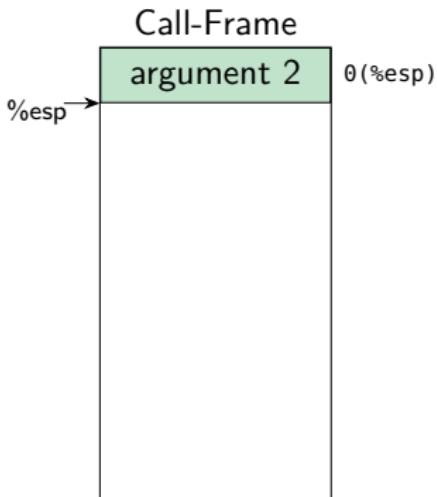
- ...wie **Argumente** an Funktionen übergeben werden und
- ...welche Teile des **Maschinenzustands** der Callee erhalten muss und
- ...wo der Caller den **Rückgabewert** auslesen kann.

- Das Betriebssystem und Übersetzer bestimmen die Aufrufkonvention
  - Funktionen können unabhängig voneinander übersetzt werden.
  - Interoperabilität zwischen verschiedenen Übersetzern und Sprachen
- Beispiel: Sys-V Calling Convention für C auf IA-32/GNU Linux
  - Callee legt Call-Frame auf dem Stack an und räumt ihn wieder weg
  - Argumente werden von **rechts-nach-links** auf den Stack gelegt
  - Rückgabewerte: `%eax` für **int**, `%st0` für **double**  
Größere Argumente/Rückgabewerte wie in Vorlesung 8
  - **Callee-saved Register**: `%esp`, `%ebp`, `%ebx`, `%esi`, `%edi`

Andere Konvention (auf IA-32): `syscall`, `optlink`, `pascal`, `stdcall`, `fastcall`,  
`vectorcall`, `safecall`, `thiscall`, ...

- Auf der IA-32 Plattform wächst der Stack von den hohen zu den niedrigen Adressen
- Der **Stackpointer** ist in `%esp`

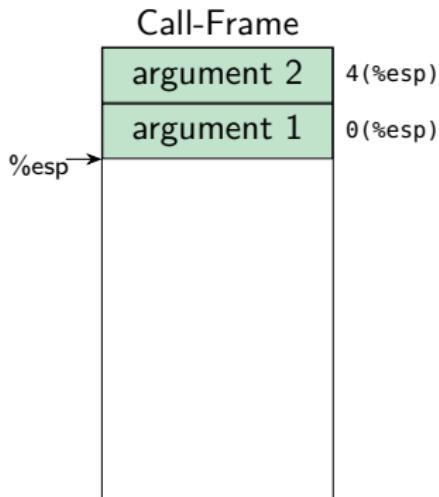




- Auf der IA-32 Plattform wächst der Stack von den hohen zu den niedrigen Adressen
- Der **Stackpointer** ist in **%esp**

## Aufrufsequenz

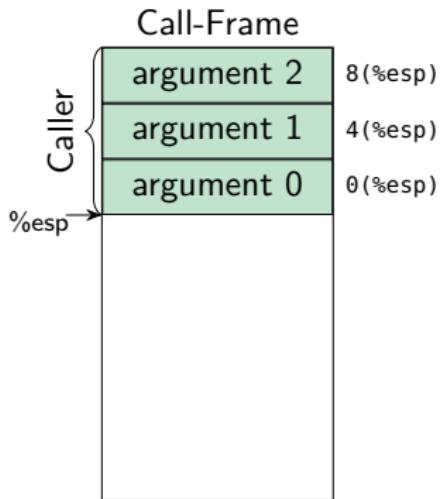
1. Argumente von **rechts nach links** ermöglicht Funktionen mit variabler Argumentanzahl
  - `push arg2;`



- Auf der IA-32 Plattform wächst der Stack von den hohen zu den niedrigen Adressen
- Der **Stackpointer** ist in **%esp**

## Aufrufsequenz

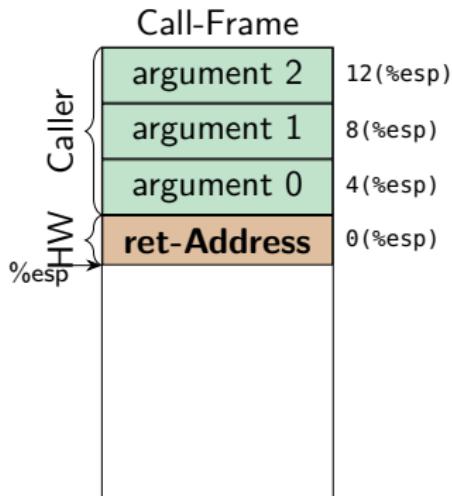
1. Argumente von **rechts nach links** ermöglicht Funktionen mit variabler Argumentanzahl
  - `push arg2; push arg1;`



- Auf der IA-32 Plattform wächst der Stack von den hohen zu den niedrigen Adressen
- Der **Stackpointer** ist in **%esp**

### Aufrufsequenz

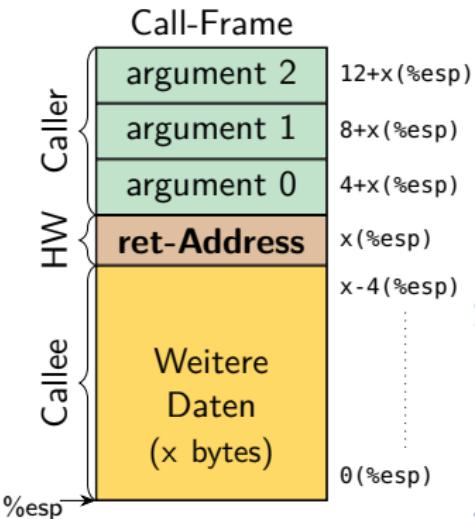
1. Argumente von **rechts nach links** ermöglicht Funktionen mit variabler Argumentanzahl
  - `push arg2; push arg1; push arg0;`



- Auf der IA-32 Plattform wächst der Stack von den hohen zu den niedrigen Adressen
- Der **Stackpointer** ist in **%esp**

### Aufrufsequenz

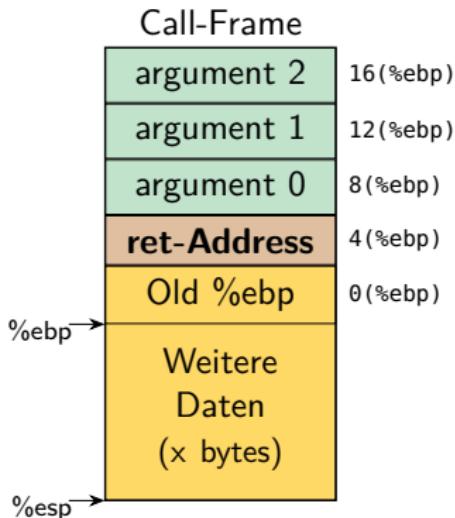
1. Argumente von **rechts nach links** ermöglicht Funktionen mit variabler Argumentanzahl
  - `push arg2; push arg1; push arg0;`
2. CPU legt Rücksprungadresse auf den Stack
  - `call bar`



- Auf der IA-32 Plattform wächst der Stack von den hohen zu den niedrigen Adressen
  - Der [Stackpointer](#) ist in `%esp`

## Aufrufsequenz

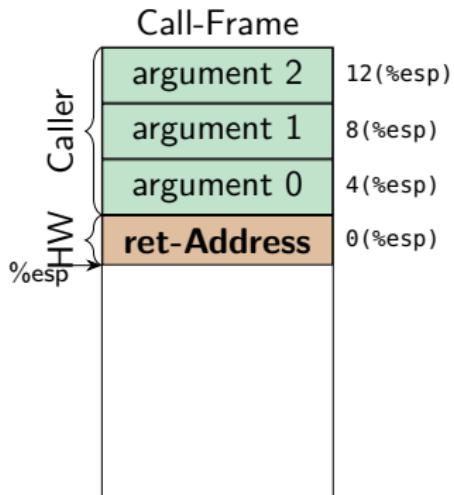
1. Argumente von **rechts nach links** ermöglicht Funktionen mit variabler Argumentanzahl
    - push arg2; push arg1; push arg0;
  2. CPU legt Rücksprungadresse auf den Stack
    - call bar
  4. Callee kann Raum für lokale Daten anlegen
    - sub x, %esp



- Auf der IA-32 Plattform wächst der Stack von den hohen zu den niedrigen Adressen
- Der **Stackpointer** ist in `%esp`

## Aufrufsequenz

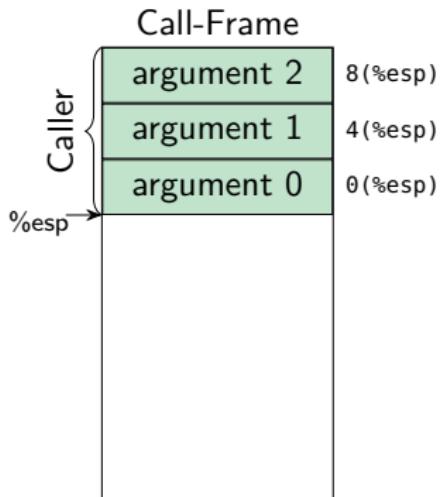
1. Argumente von **rechts nach links** ermöglicht Funktionen mit variabler Argumentanzahl
  - `push arg2; push arg1; push arg0;`
2. CPU legt Rücksprungadresse auf den Stack
  - `call bar`
3. Konstante Offsets durch Basiszeiger
  - `push %ebp; mov %esp, %ebp;`
4. Callee kann Raum für lokale Daten anlegen
  - `sub x, %esp`



- Auf der IA-32 Plattform wächst der Stack von den hohen zu den niedrigen Adressen
- Der **Stackpointer** ist in **%esp**

### Aufrufsequenz

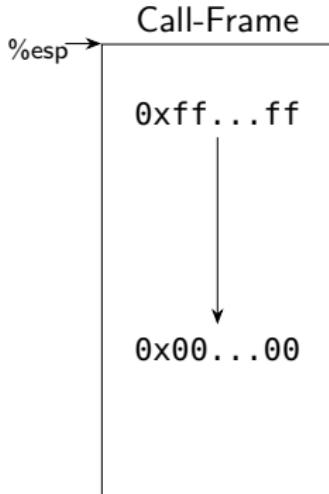
1. Argumente von **rechts nach links** ermöglicht Funktionen mit variabler Argumentanzahl
  - `push arg2; push arg1; push arg0;`
2. CPU legt Rücksprungadresse auf den Stack
  - `call bar`
3. Konstante Offsets durch Basiszeiger
  - `push %ebp; mov %esp, %ebp;`
4. Callee kann Raum für lokale Daten anlegen
  - `sub x, %esp`
5. Return in umgekehrter Reihenfolge



- Auf der IA-32 Plattform wächst der Stack von den hohen zu den niedrigen Adressen
- Der **Stackpointer** ist in **%esp**

## Aufrufsequenz

1. Argumente von **rechts nach links** ermöglicht Funktionen mit variabler Argumentanzahl
  - `push arg2; push arg1; push arg0;`
2. CPU legt Rücksprungadresse auf den Stack
  - `call bar`
3. Konstante Offsets durch Basiszeiger
  - `push %ebp; mov %esp, %ebp;`
4. Callee kann Raum für lokale Daten anlegen
  - `sub x, %esp`
5. Return in umgekehrter Reihenfolge



- Auf der IA-32 Plattform wächst der Stack von den hohen zu den niedrigen Adressen
- Der **Stackpointer** ist in **%esp**

## Aufruferfolge

1. Argumente von **rechts nach links** ermöglicht Funktionen mit variabler Argumentanzahl
  - `push arg2; push arg1; push arg0;`
2. CPU legt Rücksprungadresse auf den Stack
  - `call bar`
3. Konstante Offsets durch Basiszeiger
  - `push %ebp; mov %esp, %ebp;`
4. Callee kann Raum für lokale Daten anlegen
  - `sub x, %esp`
5. Return in umgekehrter Reihenfolge

- Übergabe von **Argumenten in Registern**
  - Die ersten N Argumente können in Registern übergeben werden
  - Spart Speicherzugriffe beim Funktionsaufruf

- Übergabe von **Argumenten in Registern**
  - Die ersten N Argumente können in Registern übergeben werden
  - Spart Speicherzugriffe beim Funktionsaufruf
- **Trade-Off** zwischen Caller-save/Callee-save Registermengen
  - Wenige Caller-save Register: wenig unnötige Registersicherungen
  - Wenige Callee-save Register: Blattfunktionen können in Registern arbeiten

- Übergabe von **Argumenten in Registern**
  - Die ersten N Argumente können in Registern übergeben werden
  - Spart Speicherzugriffe beim Funktionsaufruf
- **Trade-Off** zwischen Caller-save/Callee-save Registermengen
  - Wenige Caller-save Register: wenig unnötige Registersicherungen
  - Wenige Callee-save Register: Blattfunktionen können in Registern arbeiten
- Zusätzliche Parameter liefern Aufrufkontext
  - **thiscall**: In C++ wird der this-Zeiger in `%ecx` übergeben
  - Zieladresse für Rückgabeobjekte im 0. Parameter

## Die Welt am Beginn einer Funktion

Nach der Sicherung des **alten Basiszeigers**, sind die Parameter mit einem **konstanten, positiven Offset** zum **neuen Basiszeiger** adressierbar.

Parameter: ✓

bar:

```
push %ebp  
mov %esp, %ebp  
mov 8(%ebp), %eax
```

## Die Welt am Beginn einer Funktion

Nach der Sicherung des **alten Basiszeigers**, sind die Parameter mit einem **konstanten, positiven Offset** zum **neuen Basiszeiger** adressierbar.

Parameter: ✓

bar:

```
push %ebp
mov %esp, %ebp
mov 8(%ebp), %eax
```

- Unendlicher virtueller Registersatz (IR) vs. endlicher realer Registersatz
  - IR-Variablen müssen auf Speicherstellen im Call-Frame abgebildet werden
  - Einfachste Variante: Jede Variable → ein **Slot** im Call-Frame

```
for idx, var in enumerate(function.variables):
    var.slot      = idx
    var.ebp_offset = -4 + (-4 * idx)                                Python
```

- Variablenslots werden relativ zum Basiszeiger adressiert

## Die Welt am Beginn einer Funktion

Nach der Sicherung des **alten Basiszeigers**, sind die Parameter mit einem **konstanten, positiven Offset** zum **neuen Basiszeiger** adressierbar.

Parameter: ✓

bar:

```
push %ebp
mov %esp, %ebp
mov 8(%ebp), %eax
```

- Unendlicher virtueller Registersatz (IR) vs. endlicher realer Registersatz
  - IR-Variablen müssen auf Speicherstellen im Call-Frame abgebildet werden
  - Einfachste Variante: Jede Variable → ein **Slot** im Call-Frame

```
for idx, var in enumerate(function.variables):
    var.slot      = idx
    var.ebp_offset = -4 + (-4 * idx)
```

Python

- Variablenslots werden relativ zum Basiszeiger adressiert
- Komplexer: Colocation von Variablen in Slots + kluge **Registerallokation**



# Befehlsauswahl und Registerallokation

## Befehlsauswahl

Wähle für jeden IR-Befehl eine einzelne oder eine Sequenz aus Maschinenbefehlen aus.

## Registerallokation

Bestimme, in welchen Abschnitten eine IR-Variable in ihrem Slot oder in einem CPU-Register lebt.

## Befehlsauswahl

Wähle für jeden IR-Befehl eine einzelne oder eine Sequenz aus Maschinenbefehlen aus.

## Registerallokation

Bestimme, in welchen Abschnitten eine IR-Variable in ihrem Slot oder in einem CPU-Register lebt.

- Abbildung meist nicht eindeutig
    - „AMD64 kennt 36 `mov`-Varianten“
      - Unterschiedliche Programmgröße
      - Unterschiedliche Laufzeit
      - Unterschiedlicher Energieverbrauch
  - Betrachtung mehrerer IR-Befehle verbessert die Befehlsauswahl
- ⇒ Für sich: **NP-vollständig**

## Befehlsauswahl

Wähle für jeden IR-Befehl eine einzelne oder eine Sequenz aus Maschinenbefehlen aus.

- Abbildung meist nicht eindeutig
    - „AMD64 kennt 36 `mov`-Varianten“
      - Unterschiedliche Programmgröße
      - Unterschiedliche Laufzeit
      - Unterschiedlicher Energieverbrauch
  - Betrachtung mehrerer IR-Befehle verbessert die Befehlsauswahl
- ⇒ Für sich: **NP-vollständig**

## Registerallokation

Bestimme, in welchen Abschnitten eine IR-Variable in ihrem Slot oder in einem CPU-Register lebt.

- Problem bei  $\# \text{Variablen} > \# \text{Register}$
- Konsistent für alle Kontrollflüsse
- Massive Performance-Auswirkung
  - CPU-Register: 1 Zyklus
  - L1-Cache (Hit): 4 Zyklen
  - L2-Cache (Hit): 10 Zyklen
  - Speicher: 60-100 Zyklen

⇒ Für sich: **NP-vollständig**

## Befehlsauswahl

Interaktion

## Registerallokation

Wähle für jeden IR-Befehl eine einzelne oder eine Sequenz aus Maschinenbefehlen aus.

Bestimme, in welchen Abschnitten eine IR-Variable in ihrem Slot oder in einem CPU-Register lebt.

- Abbildung meist nicht eindeutig
    - „AMD64 kennt 36 `mov`-Varianten“
      - Unterschiedliche Programmgröße
      - Unterschiedliche Laufzeit
      - Unterschiedlicher Energieverbrauch
  - Betrachtung mehrerer IR-Befehle verbessert die Befehlsauswahl
- ⇒ Für sich: **NP-vollständig**

- Problem bei  $\# \text{Variablen} > \# \text{Register}$
- Konsistent für alle Kontrollflüsse
- Massive Performance-Auswirkung
  - CPU-Register: 1 Zyklus
  - L1-Cache (Hit): 4 Zyklen
  - L2-Cache (Hit): 10 Zyklen
  - Speicher: 60-100 Zyklen

⇒ Für sich: **NP-vollständig**

**Alles noch schlimmer:** Befehlsauswahl und Registerallokation beeinflussen sich gegenseitig und sind Abhängig von der CPU-Mikroarchitektur.

Wenn alles zu kompliziert ist, überlegt man sich den einfachsten Basisfall.

- Befehlsauswahl: Makroexpansion

  $x := \text{Add}$     $y, z$



- Registerallokation: Spilling

Wenn alles zu kompliziert ist, überlegt man sich den einfachsten Basisfall.

## ■ Befehlsauswahl: Makroexpansion

- Jeder IR-Befehl wird zu einer festen Assembler-Sequenz
- Platzhalter für Registeroperanden

  $x := \text{Add}$     $y, z$



## ■ Registerallokation: Spilling

add , 

Wenn alles zu kompliziert ist, überlegt man sich den einfachsten Basisfall.

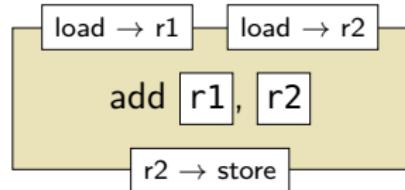
## ■ Befehlsauswahl: Makroexpansion

- Jeder IR-Befehl wird zu einer festen Assembler-Sequenz
- Platzhalter für Registeroperanden
- Zusätzliche Anweisungen wo und wie die Operanden vorliegen müssen

  $x := \text{Add}$     $y, z$



## ■ Registerallokation: Spilling

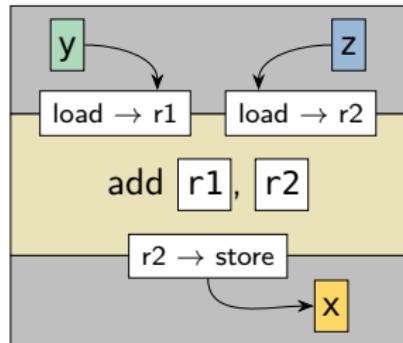


Wenn alles zu kompliziert ist, überlegt man sich den einfachsten Basisfall.

## ■ Befehlsauswahl: Makroexpansion

- Jeder IR-Befehl wird zu einer festen Assembler-Sequenz
- Platzhalter für Registeroperanden
- Zusätzliche Anweisungen wo und wie die Operanden vorliegen müssen
- Muster für jede Instruktion instantiiieren

$x := \text{Add } y, z$



## ■ Registerallokation: Spilling

Wenn alles zu kompliziert ist, überlegt man sich den einfachsten Basisfall.

## ■ Befehlsauswahl: Makroexpansion

- Jeder IR-Befehl wird zu einer festen Assembler-Sequenz
- Platzhalter für Registeroperanden
- Zusätzliche Anweisungen wo und wie die Operanden vorliegen müssen
- Muster für jede Instruktion instantiiieren

 := Add  



```
mov  , %eax  
mov  , %ebx
```

```
add %eax, %ebx
```

```
mov %ebx, 
```

## ■ Registerallokation: Spilling

- **Spilling:** Variablenwert aus einem Register in den Speicher schreiben.
- Variablen immer neuladen
- Ergebnisse direkt spilen

Wenn alles zu kompliziert ist, überlegt man sich den einfachsten Basisfall.

## ■ Befehlsauswahl: Makroexpansion

- Jeder IR-Befehl wird zu einer festen Assembler-Sequenz
- Platzhalter für Registeroperanden
- Zusätzliche Anweisungen wo und wie die Operanden vorliegen müssen
- Muster für jede Instruktion instantiiieren

  $x := \text{Add}$     $y, z$



```
mov 8(%ebp), %eax  
mov -8(%ebp), %ebx
```

```
add %eax, %ebx
```

```
mov %ebx, -4(%ebp)
```

## ■ Registerallokation: Spilling

- **Spilling:** Variablenwert aus einem Register in den Speicher schreiben.
- Variablen immer neuladen
- Ergebnisse direkt spilen
- Jede Variable hat einen Speicherslot

### 3 Probleme:

## 3 Probleme: Ineffizient

## 3 Probleme: Ineffizient, Ineffizient

### 3 Probleme: Ineffizient, Ineffizient, Ineffizient.

- Ständig werden Register, völlig ohne Not, gesichert und geladen
  - In Registern vorgeladene Variablen werden verworfen
  - Hauptteil des Programms macht nur noch Spilling
  - Speicherzugriffe sind, trotz Cache, langsamer

```
...  
mov %ebx, -8(%ebp)  
mov -8(%ebp), %eax  
...
```

### 3 Probleme: Ineffizient, Ineffizient, Ineffizient.

- Ständig werden Register, völlig ohne Not, gesichert und geladen

- In Registern vorgeladene Variablen werden verworfen
- Hauptteil des Programms macht nur noch Spilling
- Speicherzugriffe sind, trotz Cache, langsamer

```
...  
mov %ebx, -8(%ebp)  
mov -8(%ebp), %eax  
...
```

- Starre Ersetzungsmuster nutzen komplexe CPU Befehle nicht

- Besonders bei CISC sind Befehle oft sehr mächtig
- IR-Ops sind absichtlich einfach und HW-unabhängig
- Ein Befehl überdeckt mehrere IR-Befehle

```
imul %ebx, 8  
add %eax, %ebx  
mov (%ebx), %eax
```

```
mov (%eax,%ebx,8), %eax
```

### 3 Probleme: Ineffizient, Ineffizient, Ineffizient.

- Ständig werden Register, völlig ohne Not, gesichert und geladen

- In Registern vorgeladene Variablen werden verworfen
- Hauptteil des Programms macht nur noch Spilling
- Speicherzugriffe sind, trotz Cache, langsamer

```
...  
mov %ebx, -8(%ebp)  
mov -8(%ebp), %eax  
...
```

- Starre Ersetzungsmuster nutzen komplexe CPU Befehle nicht

- Besonders bei CISC sind Befehle oft sehr mächtig
- IR-Ops sind absichtlich einfach und HW-unabhängig
- Ein Befehl überdeckt mehrere IR-Befehle

```
imul %ebx, 8  
add %eax, %ebx  
mov (%ebx), %eax
```

```
mov (%eax,%ebx,8), %eax
```

- Starre Befehlsreihenfolgen ignorieren moderne Mikroarchitekturen

- Moderne Prozessoren arbeiten Pipelined, Out-of-Order und Superskalar
- Befehle können im „Windschatten“ anderer Befehle ausgeführt werden
- Reihenfolge der Befehle hat massiven Einfluss auf die Ausführungszeit

## 3 Probleme: Ineffizient, Ineffizient, Ineffizient.

- Ständig werden Register, völlig ohne Not, gesichert und geladen

- In Registern v...
- Hauptteil des

## Globale(re) Registerallokation

- Speicherzugriffe sind, trotz Cache, langsamer

, -8(%ebp)

mov -8(%ebp), %eax  
...

- Starre Ersetzungsmuster nutzen komplexe CPU Befehle nicht

- Besonders bei...  
■ IR-Ops sind a...

## Peephole-Optimizer

- Ein Befehl überdeckt mehrere IR-Befehle

imul %ebx, %eax  
bx %eax

mov (%eax,%ebx,8), %eax

- Starre Befehlsreihenfolgen ignorieren moderne Mikroarchitekturen

- Moderne P...  
■ Befehle kö...  
■ Reihenfolge der Befehle hat massiven Einfluss auf die Ausführungszeit

## Instruktions-Scheduling

# → Ein Registerallokator mit Gedächtnis

Die Minimallösung vergisst nach jeder Instruktion alles.

- **Idee:** Der Registersatz ist **Cache** für die Speicher-Variablenslots
  - Bereits in Register geladene Variablen sollen wiederverwendet werden
  - Veränderte Variablen werden erst verzögert zurückgeschrieben
  - Variablen **leben** in ihrem Slot **oder** in einem Register

Die Minimallösung vergisst nach jeder Instruktion alles.

- **Idee:** Der Registersatz ist **Cache** für die Speicher-Variablenslots
  - Bereits in Register geladene Variablen sollen wiederverwendet werden
  - Veränderte Variablen werden erst verzögert zurückgeschrieben
  - Variablen **leben** in ihrem Slot **oder** in einem Register
- Registerallokation auf Granularität eines Basisblocks
  - Verschränkung von Registerallokation (RA) und Makroexpansion (ME)
  - ME weist RA an Variablen zu laden bzw. Register zurückzuschreiben
  - RA führt währenddessen Buch über den Zustand des Registersatzes

```
def emit_Add(self, instr): # Makroexpansion für dst := Add lhs, rhs
    reg_lhs = self.RA.load(instr.lhs)
    reg_rhs = self.RA.load(instr.rhs, modify=True)
    self.emit_instr("add", reg_lhs, reg_rhs)
    self.RA.write(reg_rhs, instr.dst)
```

- LLVM bietet mit **RegAllocFast.cpp** eine ähnliche Heuristik

- Synchronisation zwischen Allokator und Expansion mittels **Hooks**
  - `def before_Function(func)` – Variablenslots festlegen
  - `def before_BasisBlock(bb)` – Zurücksetzen des Allokatorzustandes
  - `def before_Instruction(instr)` – Sonderbehandlung von Call, Goto...
- Allokieren eines leeren Registers: `def alloc_register(reg=None)`
  - Wenn kein Register vorgegeben wird, wählt der Allokator eines aus
  - Noch ungesicherte Ergebnisse werden ggf. in den Variablenslot gespeichert
- Lade Variable nach Register: `def load(src, dst_reg=None, modify=False)`
  - Sorgt dafür, dass `src` in einem Register vorliegt
  - Mit `dst_reg` können wir ein spezifisches Register verlangen
  - Mit `modify` zeigen wir an, ob wir das Register verändern werden
- Schreibe Register nach Variable: `def write(src_reg, dst_var)`
  - `dst_var` wird zukünftig den Wert `src_reg` haben
  - Das Schreiben in den Speicher kann verzögert erfolgen

	eax	ebx	ecx	edx
free				
value				
dirty				

- Während seiner Arbeit trackt der Allokator den Registerzustand
  - **free**: Wurde das Register für die aktuelle Instruktion schon benutzt?
  - **value**: Welche Variable lebt aktuell in diesem Register?
  - **dirty**: Muss der Wert noch in den Slot zurückgeschrieben werden?

	eax	ebx	ecx	edx
free	yes	no	yes	no
value	—	a	b	c
dirty	—	no	yes	no

- Während seiner Arbeit trackt der Allokator den Registerzustand
  - **free**: Wurde das Register für die aktuelle Instruktion schon benutzt?
  - **value**: Welche Variable lebt aktuell in diesem Register?
  - **dirty**: Muss der Wert noch in den Slot zurückgeschrieben werden?

## ■ Beispielbelegung

eax Nicht Herausgegeben; Nichts geladen

ebx Herausgegeben; Variable a geladen; Synchronisiert mit Speicher

ecx Nicht Herausgegeben; Variable b geladen; Rückschreiben erforderlich

edx Herausgegeben; Variable c geladen; Synchronisiert mit Speicher

	eax	ebx	ecx	edx
free	yes	no	yes	no
value	—	a	b	c
dirty	—	no	yes	no

- **Vorbereitung:** Vor jeder Instruktion setzen wir `free` zurück
  - Jedes Register kann in jeder Instruktion verwendet werden
  - Herausgabe eines Registers erzeugt **Kosten** für Spilling und Neuladen

	eax	ebx	ecx	edx
free	yes	yes	yes	yes
value	—	a	b	c
dirty	—	no	yes	no

- **Vorbereitung:** Vor jeder Instruktion setzen wir `free` zurück
  - Jedes Register kann in jeder Instruktion verwendet werden
  - Herausgabe eines Registers erzeugt **Kosten** für Spilling und Neuladen

	eax	ebx	ecx	edx
free	yes	yes	yes	yes
value	—	a	b	c
dirty	—	no	yes	no

- **Vorbereitung:** Vor jeder Instruktion setzen wir `free` zurück
  - Jedes Register kann in jeder Instruktion verwendet werden
  - Herausgabe eines Registers erzeugt **Kosten** für Spilling und Neuladen
- **Priorisierte Allokation** von Registern für die Befehlsauswahl

	eax	ebx	ecx	edx
free	no	yes	yes	yes
value	—	a	b	c
dirty	—	no	yes	no

- **Vorbereitung:** Vor jeder Instruktion setzen wir `free` zurück
  - Jedes Register kann in jeder Instruktion verwendet werden
  - Herausgabe eines Registers erzeugt **Kosten** für Spilling und Neuladen
- **Priorisierte Allokation** von Registern für die Befehlsauswahl
  - Leere Register erzeugen keine Folgekosten (Kosten: 0 `mov`)

	eax	ebx	ecx	edx
free	no	no	yes	yes
value	–	–	b	c
dirty	–	–	yes	no

- **Vorbereitung:** Vor jeder Instruktion setzen wir `free` zurück
  - Jedes Register kann in jeder Instruktion verwendet werden
  - Herausgabe eines Registers erzeugt **Kosten** für Spilling und Neuladen
- **Priorisierte Allokation** von Registern für die Befehlsauswahl
  - Leere Register erzeugen keine Folgekosten (Kosten: 0 `mov`)
  - Wert-Neuladen für saubere, aber belegte Register (Kosten: 1 `mov`)
    - Wissen über aktuellen Wert wird gelöscht

	eax	ebx	ecx	edx
free	no	no	yes	no
value	–	–	b	–
dirty	–	–	yes	–

- **Vorbereitung:** Vor jeder Instruktion setzen wir `free` zurück
  - Jedes Register kann in jeder Instruktion verwendet werden
  - Herausgabe eines Registers erzeugt **Kosten** für Spilling und Neuladen
- **Priorisierte Allokation** von Registern für die Befehlsauswahl
  - Leere Register erzeugen keine Folgekosten (Kosten: 0 `mov`)
  - Wert-Neuladen für saubere, aber belegte Register (Kosten: 1 `mov`)
    - Wissen über aktuellen Wert wird gelöscht

	eax	ebx	ecx	edx
free	no	no	no	no
value	—	—	—	—
dirty	—	—	—	—

- **Vorbereitung:** Vor jeder Instruktion setzen wir `free` zurück
  - Jedes Register kann in jeder Instruktion verwendet werden
  - Herausgabe eines Registers erzeugt **Kosten** für Spilling und Neuladen
- **Priorisierte Allokation** von Registern für die Befehlsauswahl
  - Leere Register erzeugen keine Folgekosten (Kosten: 0 `mov`)
  - Wert-Neuladen für saubere, aber belegte Register (Kosten: 1 `mov`)  
Wissen über aktuellen Wert wird gelöscht
  - Spilling und Wert-Neuladen für dreckige Register (Kosten: 2 `mov`)  
Der Allokator emittiert direkt einen Spill-Befehl für das Register

- Befehlsauswahl hat Befehle emittiert, die **src\_reg** beschrieben haben
  - Registerinhalt soll in Zukunft als **dst**-Variable verfügbar sein
  - Verzögertes Herausschreiben der Variable in ihren Speicherslot

	eax	ebx	ecx	edx
free	yes	yes	yes	yes
value	—	—	—	—
dirty	—	—	—	—

- Befehlsauswahl hat Befehle emittiert, die `src_reg` beschrieben haben
  - Registerinhalt soll in Zukunft als `dst`-Variable verfügbar sein
  - Verzögertes Herausschreiben der Variable in ihren Speicherslot
- **Beispiel:** Zuweisung zwischen zwei Variablen

`b := Assign a`

```
def emit_Assign(instr):  
    src = self.RA.load(instr.src)  
    dst = self.RA.alloc_register()  
    self.emit_instr("mov", src, dst)  
    self.RA.write(src, instr.dst)
```

	eax	ebx	ecx	edx
free	no	yes	yes	yes
value	a	—	—	—
dirty	no	—	—	—

- Befehlsauswahl hat Befehle emittiert, die `src_reg` beschrieben haben
  - Registerinhalt soll in Zukunft als `dst`-Variable verfügbar sein
  - Verzögertes Herausschreiben der Variable in ihren Speicherslot
- **Beispiel:** Zuweisung zwischen zwei Variablen `b := Assign a`

```
def emit_Assign(instr):
    src = self.RA.load(instr.src)
    dst = self.RA.alloc_register()
    self.emit_instr("mov", src, dst)
    self.RA.write(src, instr.dst)
```

```
mov -8(%ebp), %eax
```

	eax	ebx	ecx	edx
free	no	no	yes	yes
value	a	—	—	—
dirty	no	—	—	—

- Befehlsauswahl hat Befehle emittiert, die `src_reg` beschrieben haben
  - Registerinhalt soll in Zukunft als `dst`-Variable verfügbar sein
  - Verzögertes Herausschreiben der Variable in ihren Speicherslot
- **Beispiel:** Zuweisung zwischen zwei Variablen

`b := Assign a`

```
def emit_Assign(instr):
    src = self.RA.load(instr.src)
    dst = self.RA.alloc_register()
    self.emit_instr("mov", src, dst)
    self.RA.write(src, instr.dst)
```

`mov -8(%ebp), %eax`

	eax	ebx	ecx	edx
free	no	no	yes	yes
value	a	-	-	-
dirty	no	-	-	-

- Befehlsauswahl hat Befehle emittiert, die `src_reg` beschrieben haben
  - Registerinhalt soll in Zukunft als `dst`-Variable verfügbar sein
  - Verzögertes Herausschreiben der Variable in ihren Speicherslot
- **Beispiel:** Zuweisung zwischen zwei Variablen `b := Assign a`

```
def emit_Assign(instr):
    src = self.RA.load(instr.src)
    dst = self.RA.alloc_register()
    self.emit_instr("mov", src, dst)
    self.RA.write(src, instr.dst)
```

```
mov -8(%ebp), %eax
mov %eax, %ebx
```

	eax	ebx	ecx	edx
free	no	no	yes	yes
value	a	b	—	—
dirty	no	yes	—	—

- Befehlsauswahl hat Befehle emittiert, die `src_reg` beschrieben haben
  - Registerinhalt soll in Zukunft als `dst`-Variable verfügbar sein
  - Verzögertes Herausschreiben der Variable in ihren Speicherslot
- **Beispiel:** Zuweisung zwischen zwei Variablen `b := Assign a`

```
def emit_Assign(instr):
    src = self.RA.load(instr.src)
    dst = self.RA.alloc_register()
    self.emit_instr("mov", src, dst)
    self.RA.write(src, instr.dst)
```

```
mov -8(%ebp), %eax
mov %eax, %ebx
■ write() setzt nur value und dirty
■ mov %ebx, -12(%ebx) geschieht später
```

	eax	ebx	ecx	edx
free	yes	yes	yes	yes
value	—	a	c	b
dirty	—	no	yes	yes

- Laden einer Variable in ein beliebiges Register

	eax	ebx	ecx	edx
free	no	yes	yes	yes
value	z	a	c	b
dirty	no	no	yes	yes

- Laden einer Variable in ein beliebiges Register
  - **Basisfall:** Variable ist nicht geladen → `alloc_register()` + Ladebefehl

	eax	ebx	ecx	edx
free	no	no	yes	yes
value	z	a	c	b
dirty	no	no	yes	yes

- Laden einer Variable in ein beliebiges Register
  - **Basisfall:** Variable ist nicht geladen → `alloc_register()` + Ladebefehl
  - Bereits geladen und sauber → Register allokiert markieren und herausgeben

	eax	ebx	ecx	edx
free	no	no	no	yes
value	z	a	c	b
dirty	no	no	yes	yes

- Laden einer Variable in ein beliebiges Register
  - **Basisfall:** Variable ist nicht geladen → `alloc_register()` + Ladebefehl
  - Bereits geladen und sauber → Register allokiert markieren und herausgeben
  - Variable bereits geladen, aber dreckig
    - Der Aufrufer will die Variable nicht verändern → wie geladen und sauber



	eax	ebx	ecx	edx
free	no	no	no	no
value	z	a	c	b
dirty	no	no	yes	no

- Laden einer Variable in ein beliebiges Register
  - **Basisfall:** Variable ist nicht geladen → `alloc_register()` + Ladebefehl
  - Bereits geladen und sauber → Register allokiert markieren und herausgeben
  - Variable bereits geladen, aber dreckig
    - Der Aufrufer will die Variable nicht verändern → wie geladen und sauber
    - `load(src, modify=True)` → Register vorher spinnen

	eax	ebx	ecx	edx
free	no	no	no	no
value	z	a	c	b
dirty	no	no	yes	no

- Laden einer Variable in ein beliebiges Register
  - **Basisfall:** Variable ist nicht geladen → `alloc_register() + Ladebefehl`
  - Bereits geladen und sauber → Register allokiert markieren und herausgeben
  - Variable bereits geladen, aber dreckig
    - Der Aufrufer will die Variable nicht verändern → wie geladen und sauber
    - `load(src, modify=True)` → Register vorher spinnen
  
- Laden einer Variable in ein spezifisches Register
  - Manchmal brauchen wir etwas in einem spezifischen Register
  - Bereits geladenen Wert mit `xchg` austauschen

ret/%eax

- **Datenflüsse**, die die Grenzen der Basisblöcke überschreiten
    - Basisblock-Übergreifende Registervergabe ist **viel schwieriger**
    - Werte müssten in allen Vorgängern in den selben Registern sein
- ⇒ Wir starten jeden Basisblock mit einem leeren Zustand

- **Datenflüsse**, die die Grenzen der Basisblöcke überschreiten
  - Basisblock-Übergreifende Registervergabe ist **viel schwieriger**
  - Werte müssten in allen Vorgängern in den selben Registern sein

⇒ Wir starten jeden Basisblock mit einem leeren Zustand
- **Funktionsaufrufe** und **Sprünge**
  - Funktionen/andere Basisblöcke könnten die Variablen aus dem Speicher lesen
  - Funktionen können Speicher verändern

⇒ Alle Register sichern und Zustand zurücksetzen

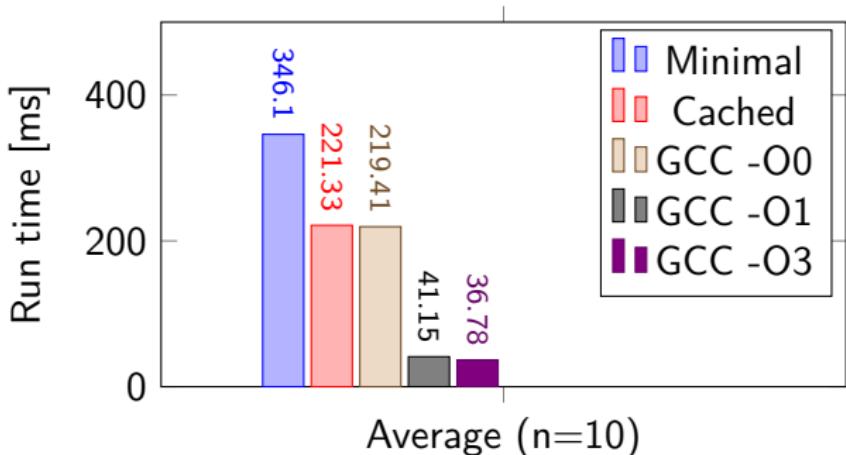
- **Datenflüsse**, die die Grenzen der Basisblöcke überschreiten
  - Basisblock-Übergreifende Registervergabe ist **viel schwieriger**
  - Werte müssten in allen Vorgängern in den selben Registern sein

⇒ Wir starten jeden Basisblock mit einem leeren Zustand
- **Funktionsaufrufe** und **Sprünge**
  - Funktionen/andere Basisblöcke könnten die Variablen aus dem Speicher lesen
  - Funktionen können Speicher verändern

⇒ Alle Register sichern und Zustand zurücksetzen
- **Speicheroperationen** haben wieder ein **Alias-Problem**
  - Wir wissen nicht, ob der gelesene/geschriebene Zeiger auf eine Variable zeigt
  - **Store** könnte Registerwerte invalidieren, **Load** könnte alte Werte lesen

⇒ Wir sichern/invalidieren Register, die jemals referenzierte Variablen enthalten

`ptr := Reference var`



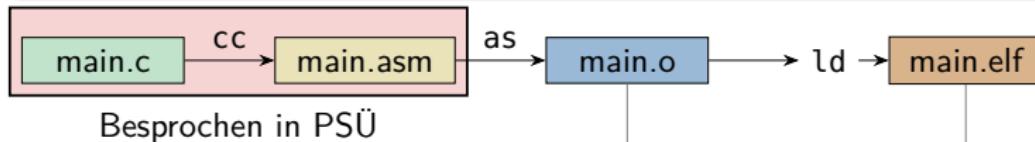
- Evaluation der beiden Allokatoren und Gegenüberstellung mit GCC
  - Benchmark: Iterativer Fibonacci, `fib_iter(1000000000)`
  - Evaluationssystem: i7 6600 @ 2.60 Ghz, 32-Bit Modus
  - Test-Setup: `perf stat -r 10 ./a.out`
- ⇒ PSÜ-Übersetzer mit Optimierungen ist Vergleichbar mit `gcc -O0`
- ⇒ GCC kann natürlich noch viel besseren Code erzeugen



# Programme und Prozesse

# Übersicht des restlichen Übersetzungsvorgang

In den letzten 10 Vorlesungen haben wir gelernt, aus Quellcode Assembler zu machen. Daraus muss noch ein **Programm** erzeugt werden, das als **Prozess** gestartet und ausgeführt werden kann.



Besprochen in PSÜ

**cc:** Übersetzer  
**as:** Assembler  
**ld:** Linker

Objekt-Datei

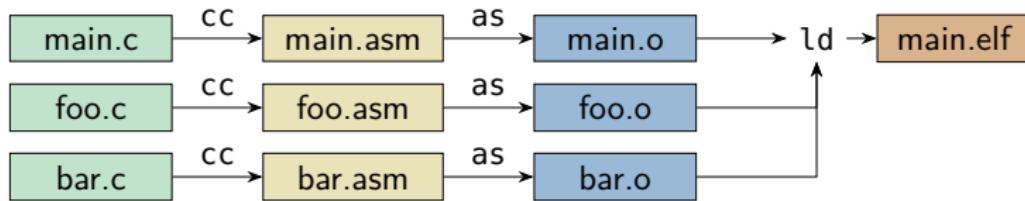
- Bereits Binärkode
- Symbolische Funktionsnamen

Binärdatei

- Aufgelöste Referenzen
- Speicherlayout ist festgelegt

# Übersicht des restlichen Übersetzungsvorgang

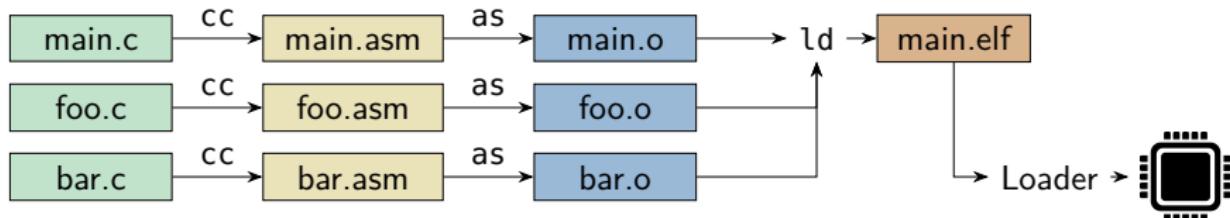
In den letzten 10 Vorlesungen haben wir gelernt, aus Quellcode Assembler zu machen. Daraus muss noch ein **Programm** erzeugt werden, das als **Prozess** gestartet und ausgeführt werden kann.



- Aber da ist noch mehr...
  - Separate Übersetzung ermöglicht inkrementelles Neuübersetzen

# Übersicht des restlichen Übersetzungsvorgang

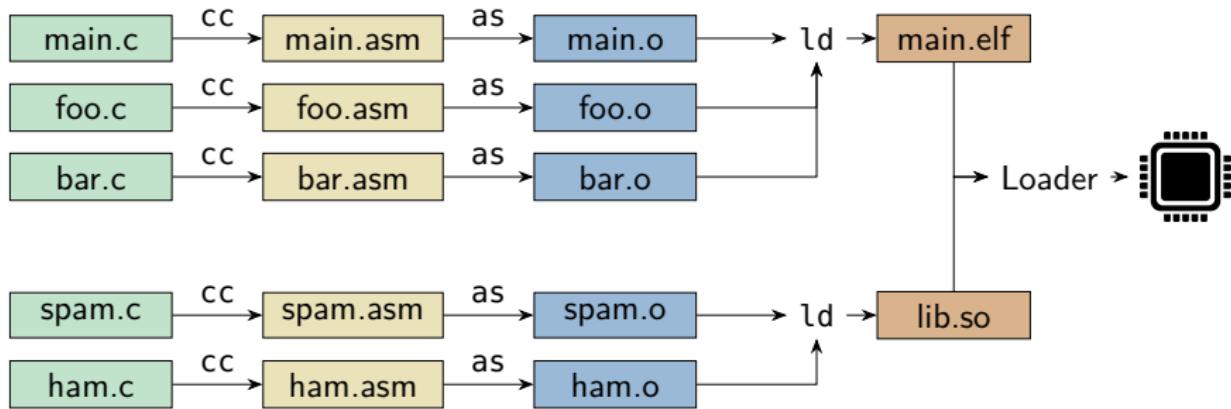
In den letzten 10 Vorlesungen haben wir gelernt, aus Quellcode Assembler zu machen. Daraus muss noch ein **Programm** erzeugt werden, das als **Prozess** gestartet und ausgeführt werden kann.



- Aber da ist noch mehr...
  - Separate Übersetzung ermöglicht inkrementelles Neuübersetzen
  - Der Loader bringt die Binärdatei in den Prozessspeicher

# Übersicht des restlichen Übersetzungsvorgang

In den letzten 10 Vorlesungen haben wir gelernt, aus Quellcode Assembler zu machen. Daraus muss noch ein **Programm** erzeugt werden, das als **Prozess** gestartet und ausgeführt werden kann.



- Aber da ist noch mehr...
  - **Separate Übersetzung** ermöglicht inkrementelles Neuübersetzen
  - Der **Loader** bringt die Binärdatei in den Prozessspeicher
  - **Gemeinsame Bibliotheken** erlauben es, Binärcode zu teilen

- ELF ist **das Dateiformat** für übersetzten Programmcode unter Linux
  - Objektdateien, Bibliotheken und Binärprogramme werden als ELF gespeichert
  - Andere Plattformen haben ähnliche Formate: Mach-O (Mac), PE (Windows)
  - Es gibt viele **Tools**, um ELF-Dateien zu inspizieren und zu verarbeiten
- ELFs speichern Code/Daten zusammen mit anderen Metadaten
  - `readelf -a ELF` liefert eine Übersicht über die Metadaten
  - Metadaten werden im Link-Prozess ergänzt und festgelegt
- Ein ELF beinhaltet zwei Sichten auf Programmdaten

### Link View

- Informationen für den Linker
- Welche Funktionen werden definiert?
- Welche Funktionen werden aufgerufen?
- Nebenbedingungen für das Arrangieren von Code/Daten

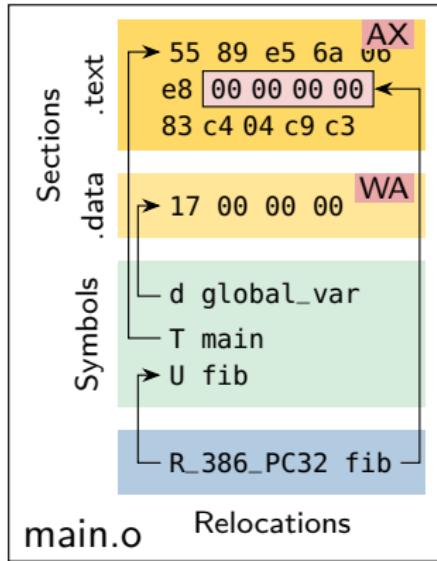
### Load View

- Informationen für den Loader
- Welcher Dateiabschnitt gehört an welche virtuelle Speicheradresse?
- Was wird ausführbar/schreibbar/lesbar?
- Müssen Bibliotheken geladen werden?

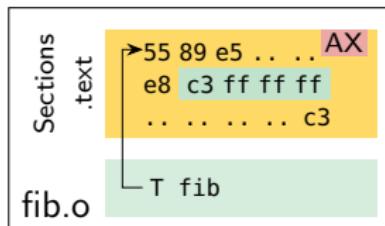
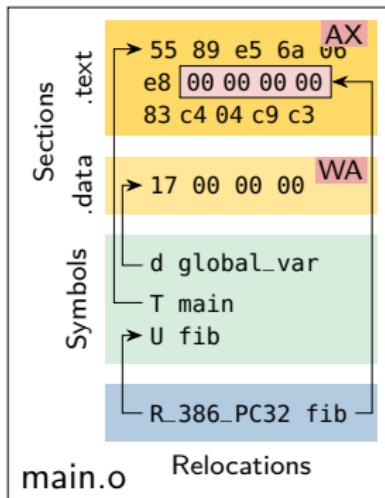
- Assembler erzeugt Objektdatei im ELF-Format
  - Assembler-Befehle werden zu Maschinencode
  - Pseudo-Instruktionen steuern die Codeerzeugung
  - Definition und Verwendung symbolischer Namen

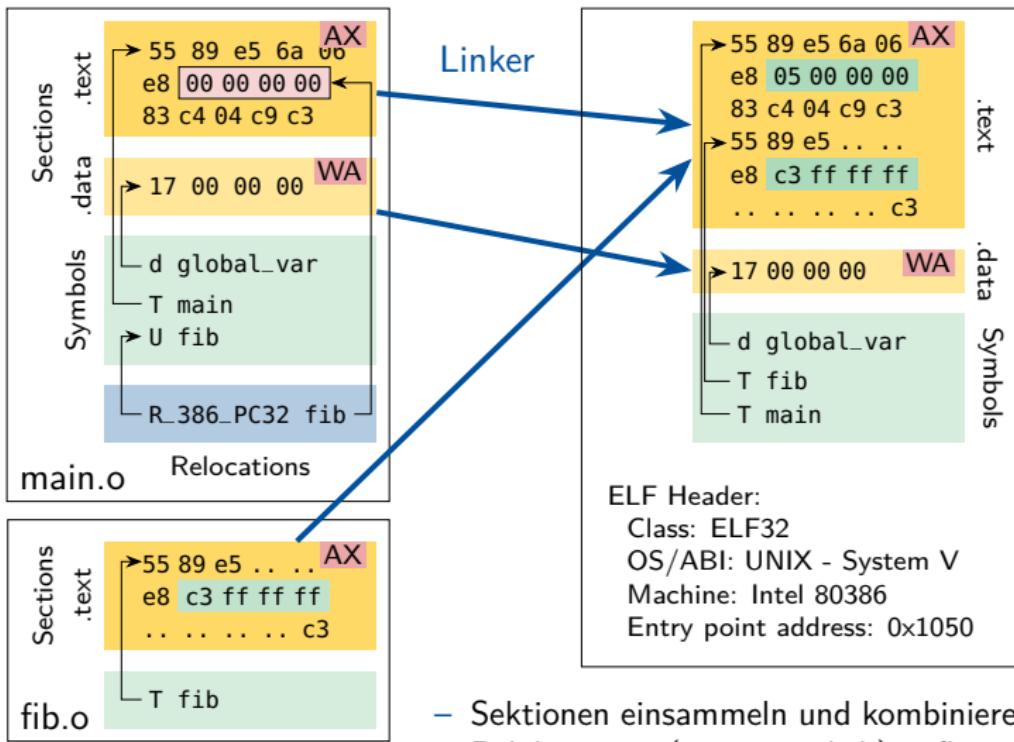
```
.text
main:
    push  %ebp
    mov   %esp, %ebp
    push  $6
    call   fib
    add   $4, %esp
    leave
    ret

.data
global_var:
    .long 23
```



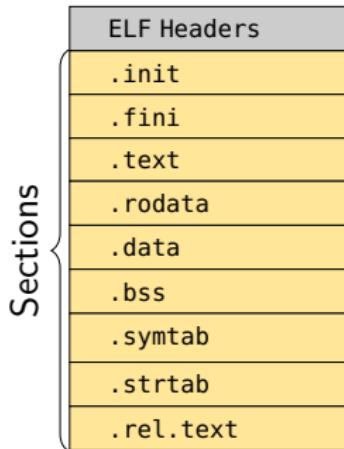
- Assembler erzeugt Objektdatei im ELF-Format
  - Assembler-Befehle werden zu Maschinencode
  - Pseudo-Instruktionen steuern die Codeerzeugung
  - Definition und Verwendung symbolischer Namen
- **Sektionen** enthalten Daten, Code und Metadaten
  - Flags: Allokier(A), Ausführbar(X), Schreibbar(W)
  - Relative Sprünge (**Goto**) bereits aufgelöst
  - Lücken für noch unbekannte Werte
  - Tool: `objdump -D ELF`
- **Symbole** geben einzelnen Bytes einen Namen
  - Symbole in diesem ELF definiert sein (z.B. **main**)
  - Undefinierte Symbole werden noch aufgelöst
  - Tool: `nm ELF`
- **Relokationen** sind Modifikationsregeln
  - Wo muss der Linker den Code modifizieren?
  - Bsp.: **&fib** wird als PC-relative Adresse eingefügt
  - Tool: `objdump -r ELF`





- Sektionen einsammeln und kombinieren
- Relocationen (wenn möglich) auflösen
- Startadresse wird festgelegt

- Linker und Loader haben unterschiedliche Sichten auf die Binärdatei
  - Der Loader arbeitet mit Segmenten und Bibliotheksreferenzen
  - Der Ladeprozess soll möglichst schnell und einfach sein
  - Sektionen (und Symbole) werden für den Load View nicht benötigt
- Das Laden geschieht auf Granularität von **Segmenten**
  - Im ELF: Dateioffset und Länge des Segments
  - Im Prozess: Ziel-Adresse, Länge der Allokation und Zugriffsrechte
- **Shared Libraries** sind ebenfalls ELF-Dateien
  - Symbole können für andere ELF-Dateien exportiert werden
  - ELF-Dateien können Bibliotheken und exportierte Symbole anfordern
  - Angeforderte Bibliotheken werden mitgeladen und die Importe aufgelöst



```
elf = ELF("main")                      # Ein minimaler Loader (ohne Libraries)
AS  = AddressSpace()
for seg in elf.program_headers:
    va_start = seg.VirtAddr or 0xf0000
    AS[va_start...seg.MemSize] = 0
    AS[va_start...seg.FileSize] = ELF[seg.Offset...seg.FileSize]
    AS[va_start...seg.MemSize].setFlags(seg.Flg)
os.StartProzess(as=AS, eip = ELF.EntryPoint, esp=0xf0fff)      Pseudo Code
```

# Was geschieht beim Laden?

Offset

	ELF Headers
0x1000	.init
0x1040	.fini
0x1140	.text
0x2000	.rodata
0x2150	.data
0x2250	.bss
0x2250	.symtab
	.strtab
	.rel.text

Segmente

Type	Offset	VirtAddr	FileSiz	MemSiz	Flg
LOAD	0x1000	0x20000	0x300	0x300	R E
LOAD	0x2000	0x21000	0x150	0x150	R
LOAD	0x2150	0x40150	0x100	0x200	RW
STACK	0x0000	0x00000	0x000	0xffff	RW

```
elf = ELF("main")                                # Ein minimaler Loader (ohne Libraries)
AS  = AddressSpace()
for seg in elf.program_headers:
    va_start = seg.VirtAddr or 0xf0000
    AS[va_start...seg.MemSize] = 0
    AS[va_start...seg.FileSize] = ELF[seg.Offset...seg.FileSize]
    AS[va_start...seg.MemSize].setFlags(seg.Flgs)
os.StartProzess(as=AS, eip = ELF.EntryPoint, esp=0xf0fff)
```

Pseudo Code

# Was geschieht beim Laden?

Offset

	ELF Headers
0x1000	.init
0x1040	.fini
0x1140	.text
0x2000	.rodata
0x2150	.data
0x2250	.bss
0x2250	.symtab
	.strtab
	.rel.text

Adressraum des Prozesses

Segmente

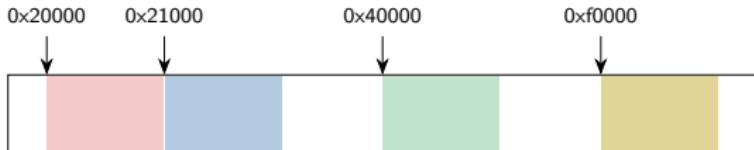
Type	Offset	VirtAddr	FileSiz	MemSiz	Flg
LOAD	0x1000	0x20000	0x300	0x300	R E
LOAD	0x2000	0x21000	0x150	0x150	R
LOAD	0x2150	0x40150	0x100	0x200	RW
STACK	0x0000	0x00000	0x000	0xffff	RW

```
elf = ELF("main")                                # Ein minimaler Loader (ohne Libraries)
AS  = AddressSpace()                               
for seg in elf.program_headers:
    va_start = seg.VirtAddr or 0xf0000
    AS[va_start...seg.MemSize] = 0
    AS[va_start...seg.FileSize] = ELF[seg.Offset...seg.FileSize]
    AS[va_start...seg.MemSize].setFlags(seg.Fl)
os.StartProzess(as=AS, eip = ELF.EntryPoint, esp=0xf0fff)    Pseudo Code
```

# Was geschieht beim Laden?

Offset

	ELF Headers
0x1000	.init
0x1040	.fini
0x1140	.text
0x2000	.rodata
0x2150	.data
0x2250	.bss
0x2250	.symtab
	.strtab
	.rel.text

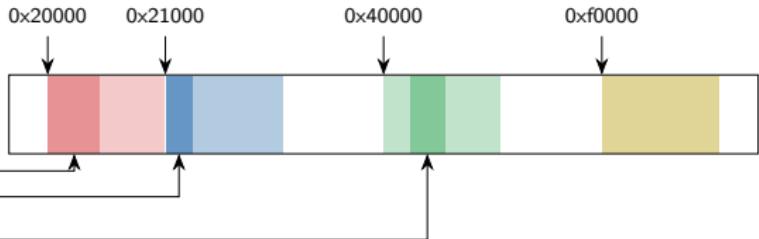
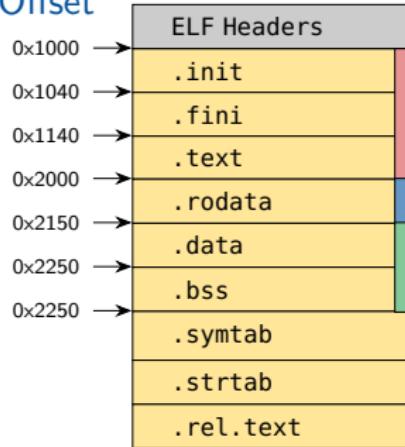


Type	Offset	VirtAddr	FileSiz	MemSiz	Flg
LOAD	0x1000	0x20000	0x300	0x300	R E
LOAD	0x2000	0x21000	0x150	0x150	R
LOAD	0x2150	0x40150	0x100	0x200	RW
STACK	0x0000	0x00000	0x000	0xffff	RW

```
elf = ELF("main")                                # Ein minimaler Loader (ohne Libraries)
AS  = AddressSpace()
for seg in elf.program_headers:
    va_start = seg.VirtAddr or 0xf0000
    AS[va_start...seg.MemSize] = 0
    AS[va_start...seg.FileSize] = ELF[seg.Offset...seg.FileSize]
    AS[va_start...seg.MemSize].setFlags(seg.Fl)
os.StartProzess(as=AS, eip = ELF.EntryPoint, esp=0xf0fff)    Pseudo Code
```

# Was geschieht beim Laden?

Offset

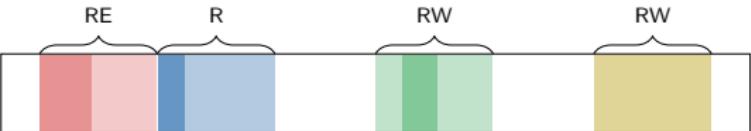


Type	Offset	VirtAddr	FileSiz	MemSiz	Flg
LOAD	0x1000	0x20000	0x300	0x300	R E
LOAD	0x2000	0x21000	0x150	0x150	R
LOAD	0x2150	0x40150	0x100	0x200	RW
STACK	0x0000	0x00000	0x000	0xffff	RW

```
elf = ELF("main")                                # Ein minimaler Loader (ohne Libraries)
AS  = AddressSpace()
for seg in elf.program_headers:
    va_start = seg.VirtAddr or 0xf0000
    AS[va_start...seg.MemSize] = 0
    AS[va_start...seg.FileSize] = ELF[seg.Offset...seg.FileSize]
    AS[va_start...seg.MemSize].setFlags(seg.Flgs)
os.StartProzess(as=AS, eip = ELF.EntryPoint, esp=0xf0fff)
```

Pseudo Code

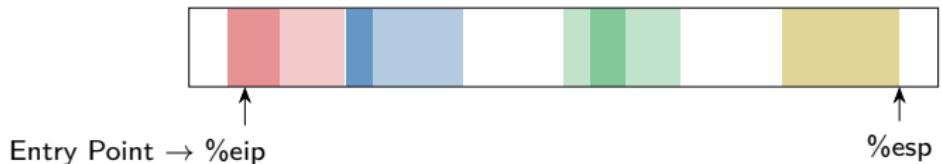
# Was geschieht beim Laden?



Type	Offset	VirtAddr	FileSize	MemSize	Flg
LOAD	0x1000	0x20000	0x300	0x300	R E
LOAD	0x2000	0x21000	0x150	0x150	R
LOAD	0x2150	0x40150	0x100	0x200	RW
STACK	0x0000	0x00000	0x000	0xffff	RW

```
elf = ELF("main")                      # Ein minimaler Loader (ohne Libraries)
AS  = AddressSpace()
for seg in elf.program_headers:
    va_start = seg.VirtAddr or 0xf0000
    AS[va_start...seg.MemSize] = 0
    AS[va_start...seg.FileSize] = ELF[seg.Offset...seg.FileSize]
    AS[va_start...seg.MemSize].setFlags(seg.Flgs)
os.StartProzess(as=AS, eip = ELF.EntryPoint, esp=0xf0fff)    Pseudo Code
```

# Was geschieht beim Laden?



```
elf = ELF("main")                      # Ein minimaler Loader (ohne Libraries)
AS  = AddressSpace()
for seg in elf.program_headers:
    va_start = seg.VirtAddr or 0xf0000
    AS[va_start...seg.MemSize] = 0
    AS[va_start...seg.FileSize] = ELF[seg.Offset...seg.FileSize]
    AS[va_start...seg.MemSize].setFlags(seg.Flg)
os.StartProzess(as=AS, eip = ELF.EntryPoint, esp=0xf0fff)  Pseudo Code
```

main
<b>Needed Libraries</b>
libblockfile.so.1
<b>Exports:</b>
–
<b>Imports:</b>
lockfile_create
lockfile_remove

libblockfile.so.1
<b>Needed Libraries</b>
libc.so.6
<b>Exports:</b>
lockfile_create
lockfile_check
lockfile_remove
<b>Imports:</b>
open
unlink

libc.so.6
<b>Needed Libraries</b>
–
<b>Exports:</b>
open
unlink
printf
<b>Imports:</b>
–

main
<b>Needed Libraries</b>
libblockfile.so.1
<b>Exports:</b>
–
<b>Imports:</b>
lockfile_create
lockfile_remove

libblockfile.so.1
<b>Needed Libraries</b>
libc.so.6
<b>Exports:</b>
lockfile_create
lockfile_check
lockfile_remove
<b>Imports:</b>
open
unlink

libc.so.6
<b>Needed Libraries</b>
–
<b>Exports:</b>
open
unlink
printf
<b>Imports:</b>
–

## ■ Vorteile von Shared Libraries

- Code kann zwischen Programmen geteilt werden
- Code kann zwischen Prozessen geteilt werden
- Update einer Bibliothek erfordert keine Neuübersetzung aller Programme

⇒ Weniger Festplatte  
⇒ Weniger RAM

main
<b>Needed Libraries</b>
libblockfile.so.1
<b>Exports:</b>
-
<b>Imports:</b>
lockfile_create
lockfile_remove

libblockfile.so.1
<b>Needed Libraries</b>
libc.so.6
<b>Exports:</b>
lockfile_create
lockfile_check
lockfile_remove
<b>Imports:</b>
open
unlink

libc.so.6
<b>Needed Libraries</b>
-
<b>Exports:</b>
open
unlink
printf
<b>Imports:</b>
-

## ■ Vorteile von Shared Libraries

- Code kann zwischen Programmen geteilt werden
- Code kann zwischen Prozessen geteilt werden
- Update einer Bibliothek erfordert keine Neuübersetzung aller Programme

⇒ Weniger Festplatte  
⇒ Weniger RAM

## ■ Besonderheiten von Shared Libraries beim Ladeprozess

- Das Auflösen der importierten Symbole (Linken) geschieht zur Ladezeit
  - Bibliothek kann in jedem Prozess woanders geladen sein
- ⇒ Bibliothekscode muss verschiebbar sein

- Maschinencodeerzeugung schließt die verbleibende **semantische Lücke**
  - Abbildung der IR-Maschine  $\Rightarrow$  reale Maschine mit **endlichen Ressourcen**
  - Komplexe Befehlssätze und keine Abstraktion vom Speicher
- **Call-Frame** enthält Argumente, Rücksprungadresse und Variablenslots
  - Jede Funktionsinstanz hat ihren eigenen Call-Frame
  - Caller und Callee halten sich an die **Aufrufkonvention**
  - Basiszeiger erlaubt Adressierung mit konstanten Offsets
- **Befehlsauswahl** und **Registerallokation** sind NP-vollständige Probleme
  - Die Minimallösung besteht aus Makroexpansion und **ständigem Spilling**
  - Verbesserte Registerallokation sieht Registersatz als Cache für Variablenslots
- ELF: Dateiformat für Objektdateien, Programme und Bibliotheken
  - **Link View**: Sektionen, Symbole und Relokationen
  - **Load View**: Segmente, importierte und exportierte Symbole
  - **Bibliotheken** erlauben Code-Sharing zwischen Programmen