Universität Bamberg

GRABS: Grundlagen der Rechnerarchitektur und Betriebssysteme

Literatur:
Arpaci-Dusseau [1]
Kap. 1 und 13

Vorlesung 9:
Vom Programm zum Prozess

Michael Engel (michael.engel@uni-bamberg.de)
Lehrstuhl für Praktische Informatik, insbes. Systemnahe Programmierung
https://www.uni-bamberg.de/sysnap

Licensed under CC BY-SA 4.0
unless noted otherwise

Idee: Abstraktion durch Interpretation

Universität Bamberg

Software

Programm in einer Hochsprache (z.B. C)

Compiler

Programm in Assembler-Sprache (z.B. RISC-V)

Assembler

Maschinensprache-Programm (z.B. RISC-V)

```
temp = v[k];
v[k] = v[k]+1;
v[k+1] = temp;
```

```
lw      a5,0(sp)
slli    a5,a5,2
addi    a5,a5,48
add     a5,a5,sp
```

```
0000 0000 0000 0001 0010 0111 1000 0011
0000 0000 0010 0111 1001 0111 1001 0011
0000 0011 0000 0111 1000 0111 1001 0011
0000 0000 0010 0111 1000 0111 1011 0011
```

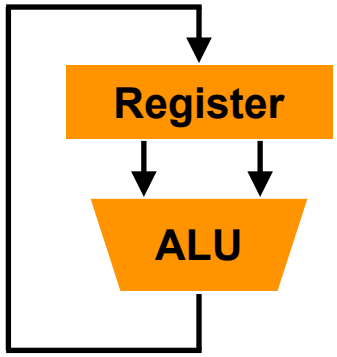
Hardware

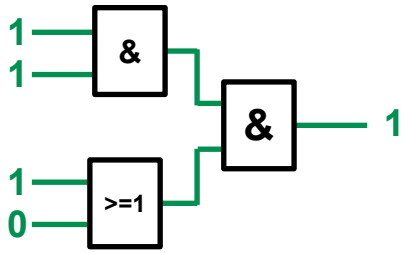
Maschinen-interpretation

Hardwarearchitektur-Diagramm (z.B. Blockschaltbilder)

Architektur-implementierung

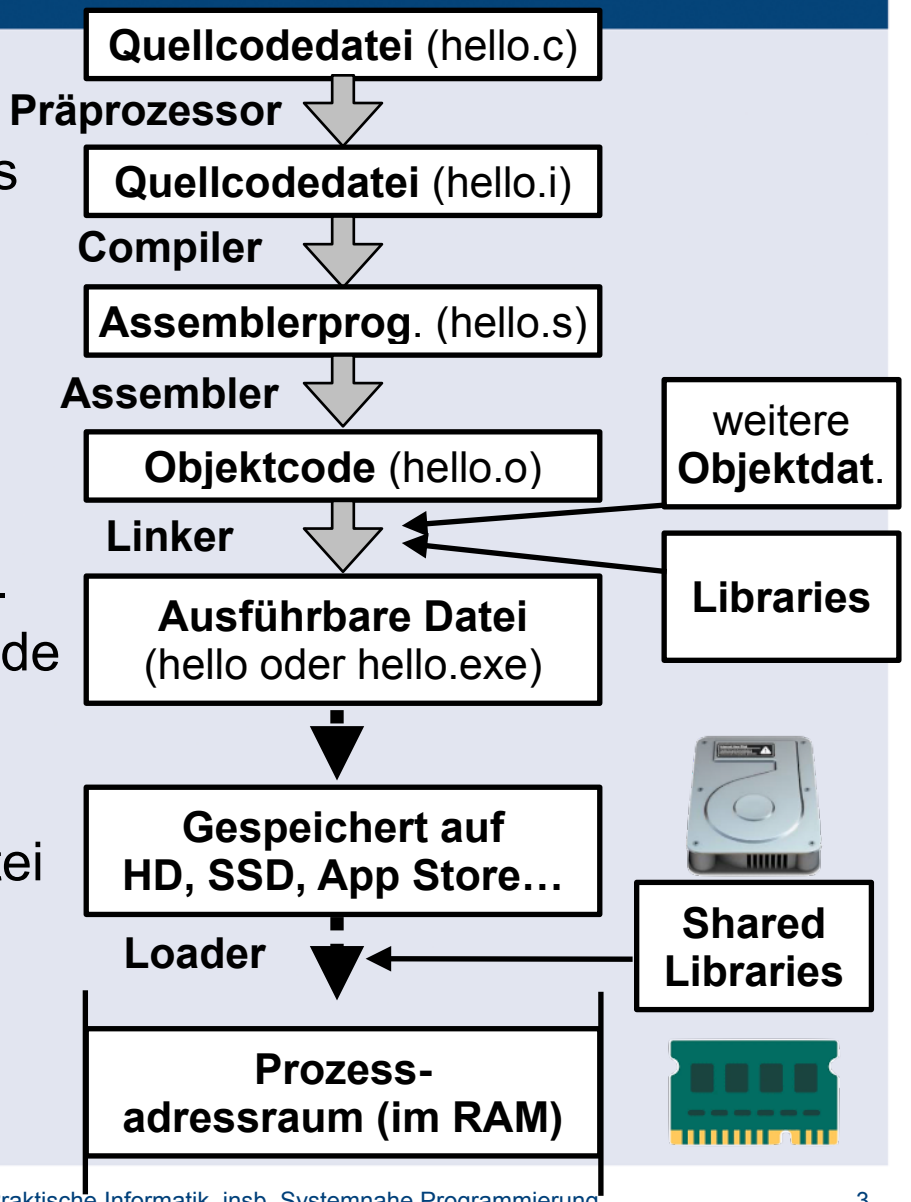
Logische Schaltungsbeschreibung (Schaltplan)





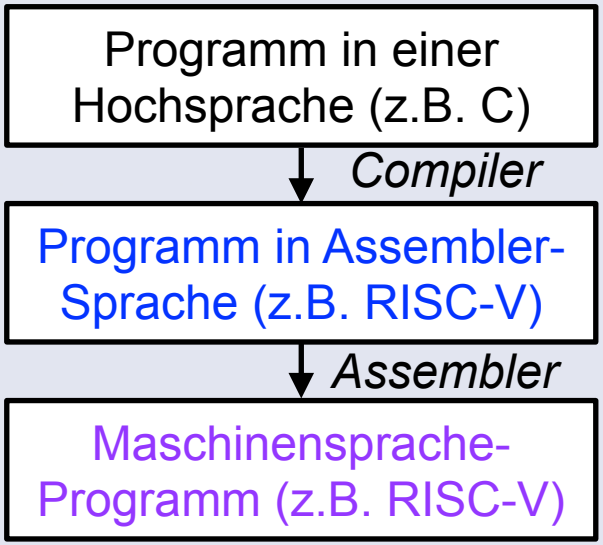
Übersetzung von Quellcode

- **Präprozessor**
  - Expandiert #includes und Makros
- **Compiler** (Übersetzer)
  - Erzeugt Assembler-Quellcode aus Hochsprachen-Quellcode (C, Rust, …)
- **Assembler**
  - Erzeugt Objektcode (Maschinen-sprache) aus Assembler-Quellcode
- **Linker** (Binder)
  - Kombiniert Objektcode-Dateien (+ Libraries) zu ausführbarer Datei
- **Loader** (Lader)
  - Lädt ausführbare Datei in den Hauptspeicher



Von C zum ausführbaren Programm

Ein **Compiler** übersetzt die Konstrukte der Hochsprache (Kontrollanweisungen wie if/else und Schleifen, Funktionen, Variablen usw.) in die Assemblersprache des jeweiligen Prozessors



Struktur und Ablauf eines Compilers:

Quellcode

Zeichenstrom

Lexikalische Analyse → Syntax-analyse → Semantische Analyse → Code-erzeugung → Code-optimierung

Objektcode

Compiler: Lexikalische Analyse

Quellcode

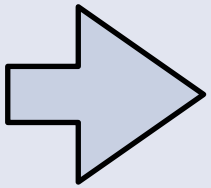
Zeichenstrom

Lexikalische Analyse (*scanning*):

- Teilt Quellcode in lexikalische Einheiten auf
- Erkennt ***Tokens*** (mit Automaten / regulären Ausdrücken)
- Token: Zeichenfolge, die in der ***Grammatik*** der zu übersetzenden Sprache relevant ist

```
int a = 42;
int b = 23;

int foo(void) {
  return a+b*2;
}
```
C



```
T(int); T(ident,a); T('=');
T(number, 42); T(';'); T(int);
T(ident,b); T('='); T(number, 23);
T(int); T(ident,foo); T('(');
T(void); T(')'); T('{');
T(return); T(ident,a); T('+'); …
```

Compiler: Syntaxanalyse

Quellcode

Zeichenstrom

Lexikalische Analyse → **Syntax-analyse** → Semantische Analyse → Code-erzeugung → Code-optimierung → ***Objektcode***

Syntaxanalyse (*parsing*):

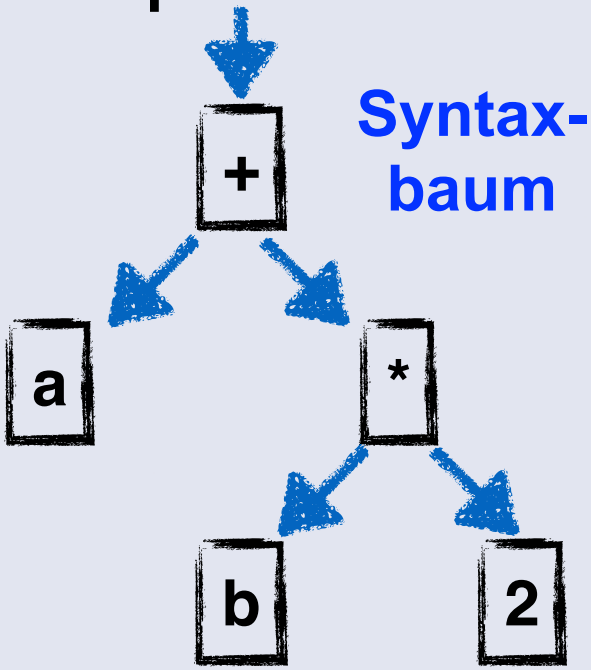
- Verwendet ***Grammatik*** der Quellsprache
- Entscheidet, ob Eingabe-Tokenfolge aus der Grammatik abgeleitet werden kann

```
T(ident,a); T('+'); T(ident,b);
T('*'); T(number, 2); T(';'); …
```

?=

```
expression → term { (+|-) term }
term →       factor { (*|/) factor }
factor →     '(' expression ')'
           | id | number
```

Expression



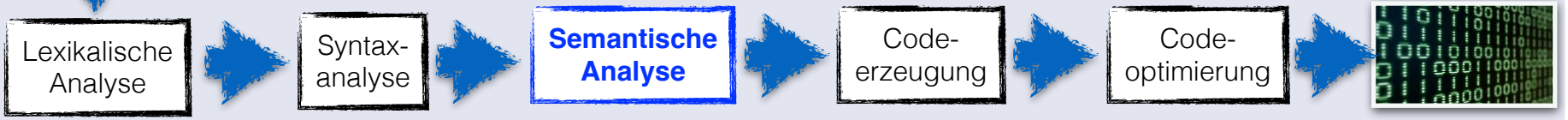
Syntax-baum

Compiler: Semantische Analyse

Universität Bamberg

Quellcode

Zeichenstrom



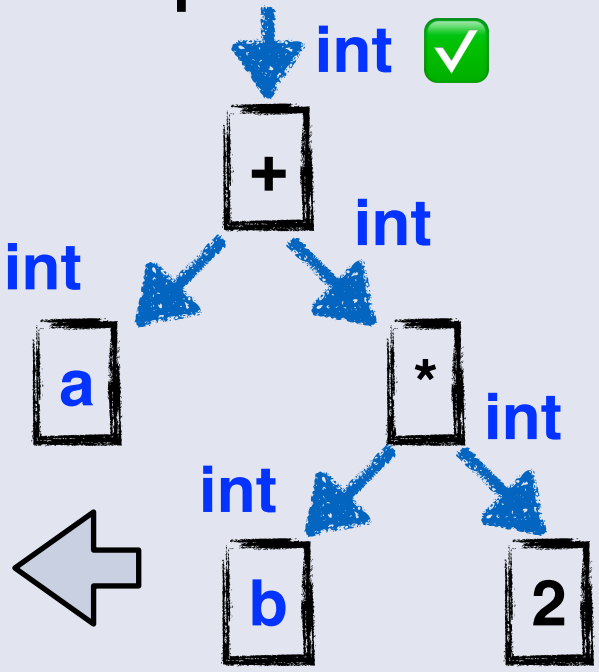
Objektcode

Semantische Analyse:

- **Namensanalyse**:
  Definition & Scope von Symbolen
- **Typanalyse:**
  Korrekter Typ von Ausdrücken
- Erzeugung von **Symboltabellen:**
  Identifier ihren Typen und der Position im Quellcode zuordnen

Name	Typ	Posit.
a	int	23
b	int	42

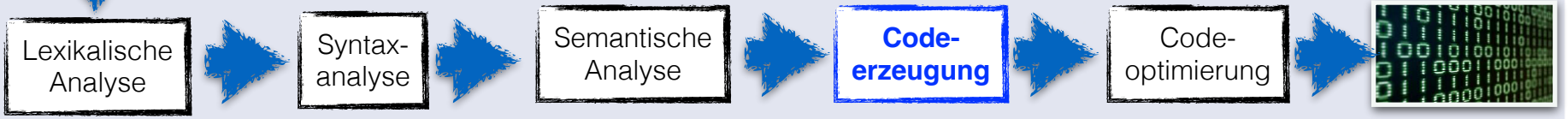
Expression



Compiler: Codeerzeugung

Quellcode

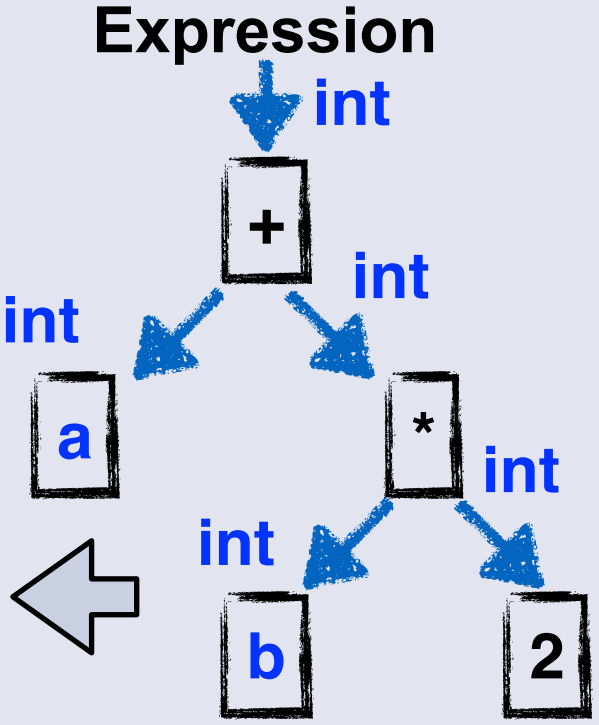
Zeichenstrom



Codeerzeugung:

- Erzeugt den Knoten des Syntaxbaums entsprechende Assemblerinstruktionen
- Oft direkte Übersetzung der Baumstruktur
- Weist Variablen "passende" Speicherzellen (und Register) zu

```
la   x10, b
lw   x10, 0(x10)
li   x11, 2
mul  x12, x10, x11
la   x10, a
lw   x10, 0(x10)
add  x13, x10, x12
```



Compiler: Codeoptimierung

Quellcode

Zeichenstrom

Objektcode

Codeoptimierung:

- Analysiert Redundanzen und versucht, diese zu eliminieren und/oder Code zu vereinfachen
  - z.B. Speichern einer Variablen direkt gefolgt von Laden
  - z.B. Multiplikation mit Zweierpotenz durch Shift ersetzen
- Komplexere Optimierungen sind oft schwierig oder aufwändig

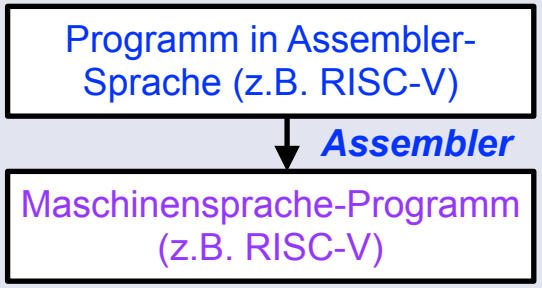
```
la   x10, b
lw   x10, 0(x10)
li   x11, 2
mul  x12, x10, x11
la   x10, a
lw   x10, 0(x10)
add  x13, x10, x12
```

```
la   x10, b
lw   x10, 0(x10)
slli x12, x10, 1
la   x10, a
lw   x10, 0(x10)
add  x13, x10, x12
```

Beispiel für Maschinensprache

Universität Bamberg

Der Assembler übersetzt den Assembler-Quellcode und erzeugt eine **Binärdatei**



Dies sind die Instruktionen in Maschinensprache

RISC-V **Assembler**

```
    li   x3, 10
while:
    bge  x1, x3, end
    add  x1, x1, 1
    add  x2, x2, 2
    j    while
end:
    …
```

RISC-V **Maschinensprache**

```
00000000 <while-0x4>:
   0:  00a00193    li   x3,10

00000004 <while>:
   4:  0030d863    bge  x1,x1,14 <end>
   8:  00108093    addi x1,x1,1
   c:  00210113    addi x2,x2,2
  10:  ff5ff06f    j    4 <while>
00000014 <end>:
```

Adresse

```
00000000  93 01 a0 00 63 d8 30 00
00000008  93 80 10 00 13 01 21 00
00000010  6f f0 5f ff
```

Little Endian Bytereihenfolge!

Dies ist die Ausgabe eines *Disassemblers*, der aus den binären (in Hex-Darstellung) Operationen wieder menschenlesbaren Assemblercode erzeugt

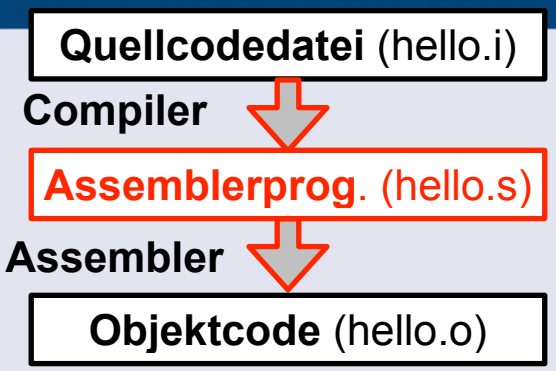
Moment mal…

Wo ist unser Assembler-Quellcode?

Wird üblicherweise nur intern als temporäre Datei erzeugt!

Die Compileroption "-S" generiert eine .s Assembler-Quellcodedatei:

```
$ gcc -c foo.c
$ ls -1
foo.c
foo.o
$ gcc -S foo.c
$ ls -1
foo.c
foo.o
foo.s
```




```
        .file   "foo.c"
        .option nopic
        .attribute arch, "rv32i2p0_m2p0"
        .attribute unaligned_access, 0
        .attribute stack_align, 16
        .text
        .align  2
        .globl  add
        .type   add, @function
add:
        add     a0,a0,a1
        ret
        .size   add, .-add
        .align  2
        .globl  foo
        .type   foo, @function
foo:
        slli    a1,a1,1
        add     a0,a1,a0
        ret
```

Speichersektionen von Programmen

Universität Bamberg

- Wir kennen schon die **Direktiven** für den Assembler (aus Vorlesung 5)

```
    .data
a:  .dc.w 42
b:  .dc.w 23

    .text
foo:
    la  x1, a
    lw  x3, 0(x1)
    la  x2, b
    lw  x4, 0(x2)
    add x10, x3, x4
    ret
```

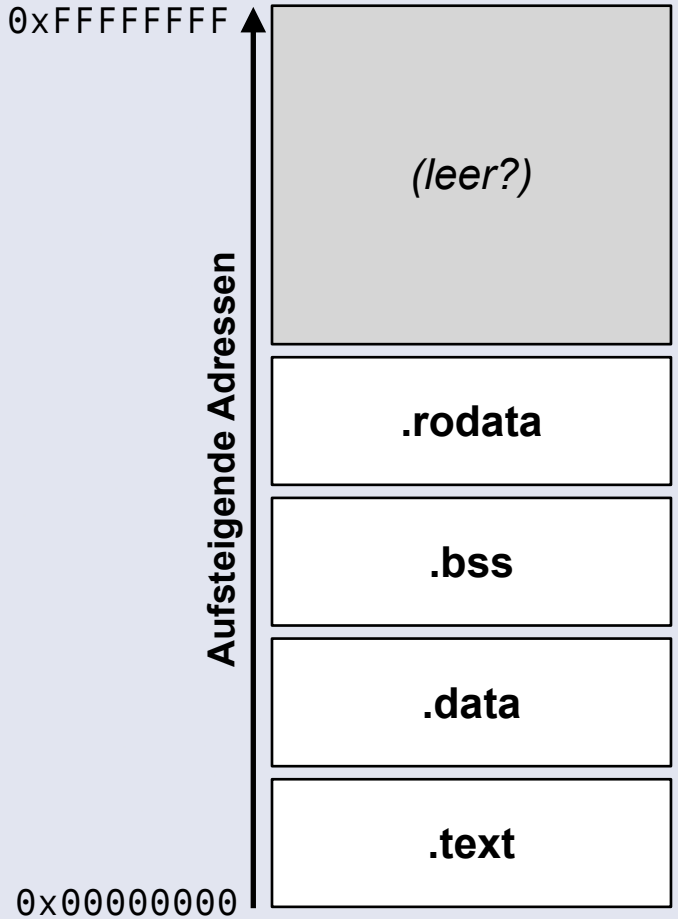
`.data` und `.text` sind **Direktiven**, diese geben an, zu welcher **Sektion** des Programms die folgenden Zeilen gehören

Was bedeuten diese?

- Festlegung, zu welcher **Sektion** des zu erzeugenden Maschinenprogramms die folgenden Zeilen gehören:
  - **.text** – ausführbarer Code (Maschineninstruktionen), nur lesen
  - **.data** – globale und `static`-Variablen, lesen und schreiben
  - **.bss** – nicht initialisierte Daten, lesen und schreiben
  - **.rodata** – globale Konstanten, nur lesen

Speicherlayout von Programmen

- Die Sektionen **.text, .data, .bss** und **.rodata** werden beim Programmstart in separate Speicherbereiche geladen
  - Üblich: in aufsteigende Bereiche in der angegebenen Reihenfolge



- Festlegung der Adressbereiche erfolgt durch **Linkerskript**: [2]

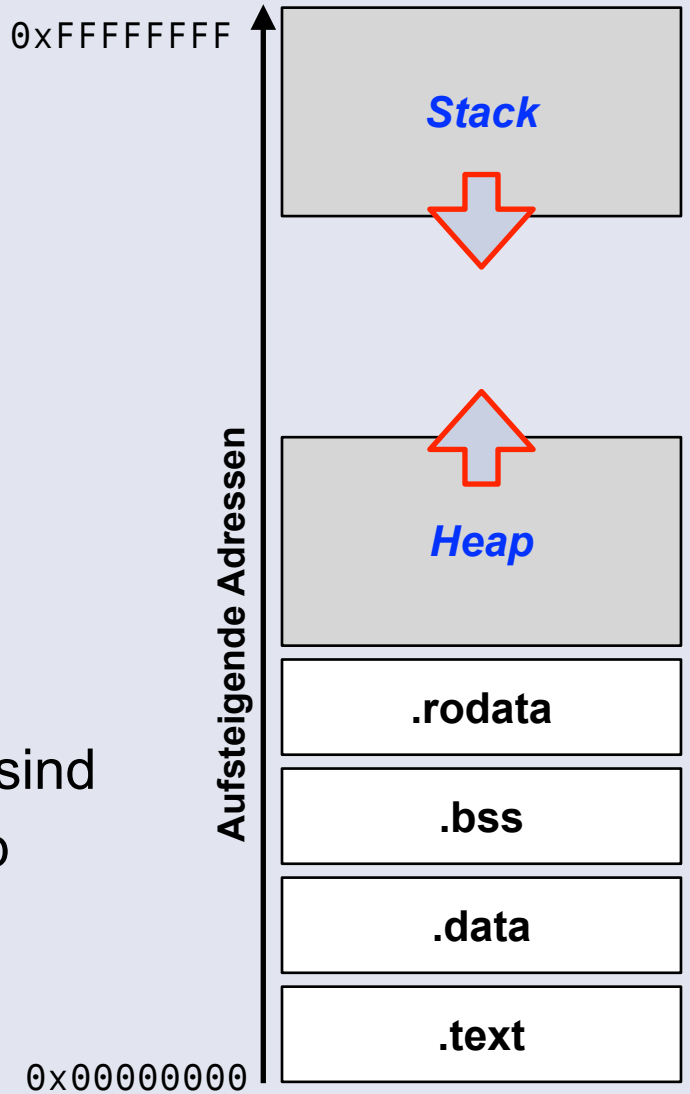
```
OUTPUT_ARCH( "riscv" )
ENTRY( _start )

SECTIONS
{
  . = 0x00000000;
  .text : {
    *(.text .text.*)
    . = ALIGN(0x1000);
    PROVIDE(etext = .);
  }
  /* …weiter… ☞ */
```

```
  /* … */
  .data : {
    . = ALIGN(16);
     *(.data .data.*)
  }
  .bss : {
    . = ALIGN(16);
     *(.bss .bss.*)
  }
  /* … */
  PROVIDE(end = .);
}
```

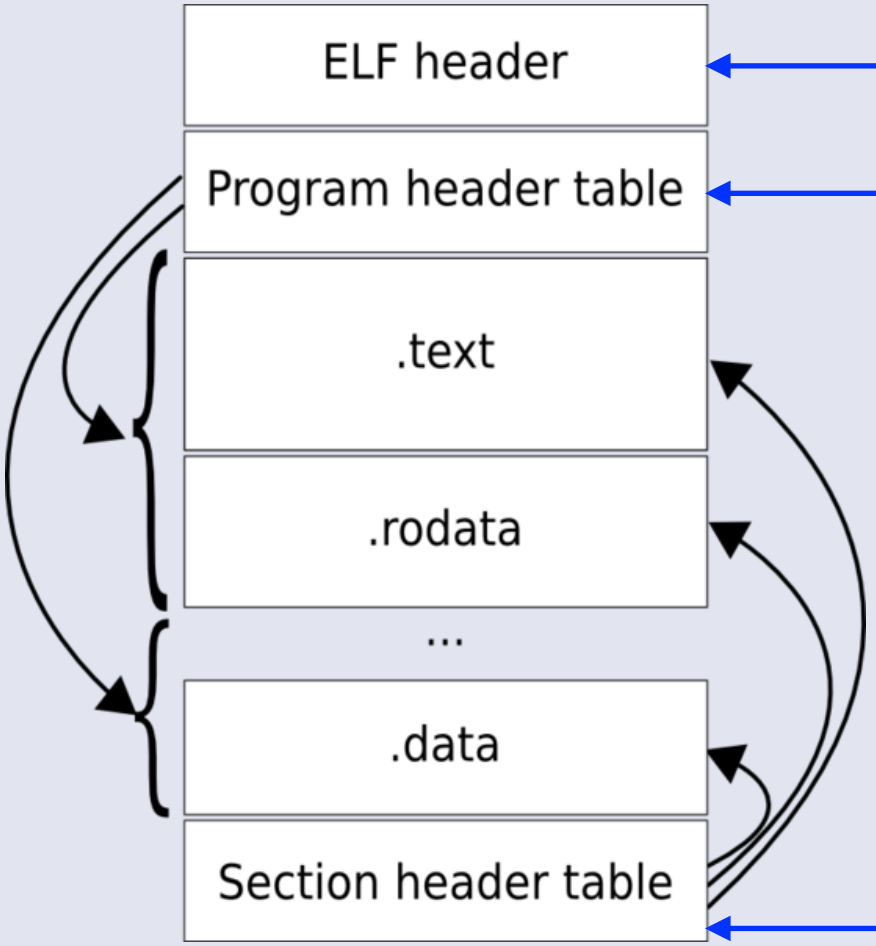
Speicherlayout von Programmen

- **.text, .data, .bss** und **.rodata** sind **statisch allozierte** Speicherbereiche
- Was ist im "leeren" Bereich?
  - **Dynamisch allozierte Daten**
- ***Heap*** ("Halde"):
  - beliebig anforderbare (`malloc`) und freigebbare (`free`) Speicherbereiche von wahlfreier Größe
- ***Stack*** ("Stapel" oder "Kellerspeicher"):
  - enthält Werte, die zur Ausführung von Instanzen von Funktionen notwendig sind
- Die **Adressbereiche** von Stack und Heap **laufen** i.d.R. **aufeinander zu**
  - Stack wächst von "oben", Heap von "unten"



Struktur ausführbarer Dateien: ELF

Die Struktur, die im Speicher realisiert werden soll, wird vom Linker in der ausführbaren Datei mit vermerkt:



Header mit allgemeinen Informationen (see previous slides)

Informationen über die ELF-Dateisegmente

- "**ELF**" steht für "Executable and Linkable Format" und wird von Linux und den meisten anderen Unix-Systemen verwendet [3]
- MacOS verwendet **Mach-O**
- Windows verwendet **COFF/PE**

Tabelle der Sektions-Header mit detaillierten Informationen zu jedem Segment

ELF-Sektionen

readelf ist ein Unix-Programm, das wichtige Informationen aus ELF-Dateien decodiert

```
$ riscv32-unknown-readelf -S foo
There are 13 section headers, starting at offset 0x1148:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00010074 000074 000594 00  AX  0   0  4
  [ 2] .eh_frame         PROGBITS        00011608 000608 000004 00  WA  0   0  4
  [ 3] .init_array       INIT_ARRAY      0001160c 00060c 000008 04  WA  0   0  4
  [ 4] .fini_array       FINI_ARRAY      00011614 000614 000004 04  WA  0   0  4
  [ 5] .data             PROGBITS        00011618 000618 000428 00  WA  0   0  8
  [ 6] .sdata            PROGBITS        00011a40 000a40 000014 00  WA  0   0  4
  [ 7] .bss              NOBITS          00011a54 000a54 00001c 00  WA  0   0  4
  [ 8] .comment          PROGBITS        00000000 000a54 000012 01  MS  0   0  1
  [ 9] .riscv.attributes RISCV_ATTRIBUTE 00000000 000a66 000021 00      0   0  1
  [10] .symtab           SYMTAB          00000000 000a88 000400 10     11  39  4
  [11] .strtab           STRTAB          00000000 000e88 00024f 00      0   0  1
  [12] .shstrtab         STRTAB          00000000 0010d7 000070 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  D (mbind), p (processor specific)
```

ELF-Sektionen: Was ist wo?

Beispielprogramm foo.c:

```
const int a = 42;
int b = 23;
int c;

int main(void) {
  c = a + b;
  return c;
}
```

Was ist wo in der Objektdatei?

$ **gcc –c foo.c**
$ **readelf –S foo.o**

```
  [Nr] Name   Type      Address  Offset   Size     EntSize Flags Link Info Align
…
[14] .text    PROGBITS  00001040 00001040 0000010c 00000000 AX     0    0    16
…
[16] .rodata  PROGBITS  00002000 00002000 00000008 00000000 A      0    0    4
…
[23] .data    PROGBITS  00004000 00003000 00000014 00000000 WA     0    0    8
[24] .bss     NOBITS    00004014 00003014 0000000c 00000000 WA     0    0    4
```

Sektion	Inhalt
text (.text)	Maschinencode (Instruktionen) und Einsprungpunkt (Adresse)
read only data (.rodata)	Initialisierte Konstanten
read/write data (.data)	Initialisierte Variablen
Base Storage Segment (.bss)	Uninitialisierte Variablen

Funktionen in RISC-V

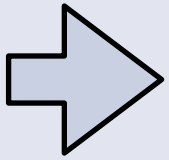
C

```
int x = 42;
int y = 23;

int add(int a, int b) {
  return a+b;
}

int foo(int a, int b) {
  int c = 2;
  return add(a, b*c);
}

int main(int argc, char **argv) {
  return foo(x,y);
}
```



Assembler

```
foo:
        addi    sp,sp,-48
        sw      ra,44(sp)
        sw      s0,40(sp)
        addi    s0,sp,48
        sw      a0,-36(s0)
        sw      a1,-40(s0)
        li      a5,2
        sw      a5,-20(s0)
        lw      a4,-40(s0)
        lw      a5,-20(s0)
        mul     a5,a4,a5
        mv      a1,a5
        lw      a0,-36(s0)
        call    add
        mv      a5,a0
        mv      a0,a5
        lw      ra,44(sp)
        lw      s0,40(sp)
        addi    sp,sp,48
        jr      ra
```

Übersetzt mit:

```
riscv64-unknown-elf-gcc -march=rv32i \
     -mabi=ilp32 -O0 -S foo.c
```

Funktionen bei RISC-V

- In der Hörsaalübung schon gesehen: Für die Register `x0`–`x31` existieren **alternative Registernamen**

- Diese beschreiben einen Teil der **Konvention**, wie Compiler, Programme und Betriebssystem die Register nutzen sollen:

  Application Binary Interface (ABI) [4]

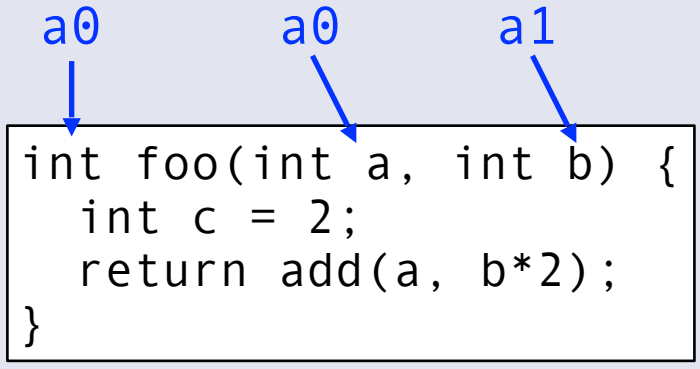
  - legt Registerkonventionen, Stackaufbau, Systemaufrufe u.v.m. fest

x_i	ABI	Bedeutung	Gesichert von
x0	zero	immer = 0	
x1	ra	Returnadresse	Aufrufer
x2	sp	Stackzeiger	Funktion
x3	gp	Globaler Zeiger	
x4	tp	Thread-Zeiger	
x5	t0	Temporäre Daten	Aufrufer
x6-7	t1-t2	Temporäre Daten	Aufrufer
x8	s0/fp	Gesicherte Register/ Frame Pointer	Funktion
x9	s1	Gesichertes Register	Funktion
x10-11	a0-a1	Funktionsparameter/ Rückgabewert	Aufrufer
x12-17	a2-a7	Funktionsparameter	Aufrufer
x18-27	s2-s11	Gesichertes Register	Funktion
x28-31	t3-t6	Temporäre Daten	Aufrufer

Registerkonventionen im RISC-V-ABI

- **Parameterübergabe** an Funktionen in Registern `a0`...`a7`
- **Rückgabe** von Werten aus Funktionen in `a0` (und `a1` bei Bedarf)
- **Rücksprungadresse** aus Funktion in `ra`
- **Stackzeiger** in `sp` (*stack pointer*)

- Einige Register müssen laut Konvention von der aufgerufenen Funktion ("Funktion"), andere von der aufrufenden Funktion ("Aufrufer") gesichert werden, wenn der Wert erhalten werden soll

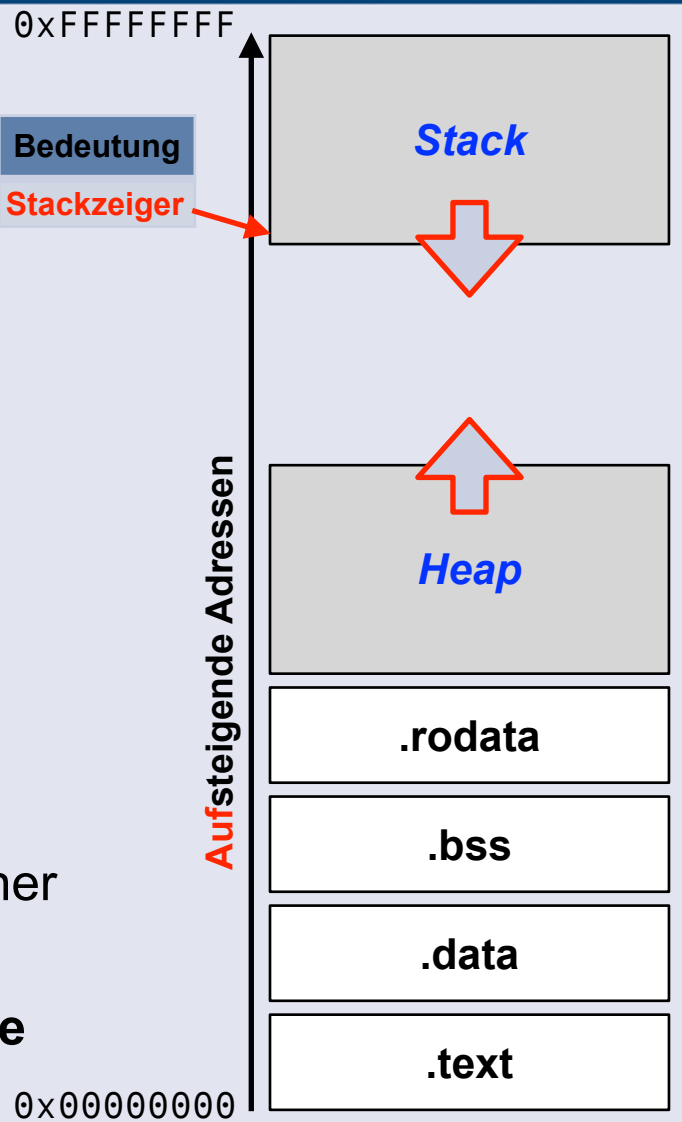


x_i	ABI	Bedeutung	Gesichert von
x0	zero	immer = 0	
x1	ra	Returnadresse	Aufrufer
x2	sp	Stackzeiger	Funktion
x3	gp	Globaler Zeiger	
x4	tp	Thread-Zeiger	
x5	t0	Temporäre Daten	Aufrufer
x6-7	t1-t2	Temporäre Daten	Aufrufer
x8	s0/fp	Gesicherte Register/ Frame Pointer	Funktion
x9	s1	Gesichertes Register	Funktion
x10-11	a0-a1	Funktionsparameter/ Rückgabewert	Aufrufer
x12-17	a2-a7	Funktionsparameter	Aufrufer
x18-27	s2-s11	Gesichertes Register	Funktion
x28-31	t3-t6	Temporäre Daten	Aufrufer

Der Stack bei RISC-V

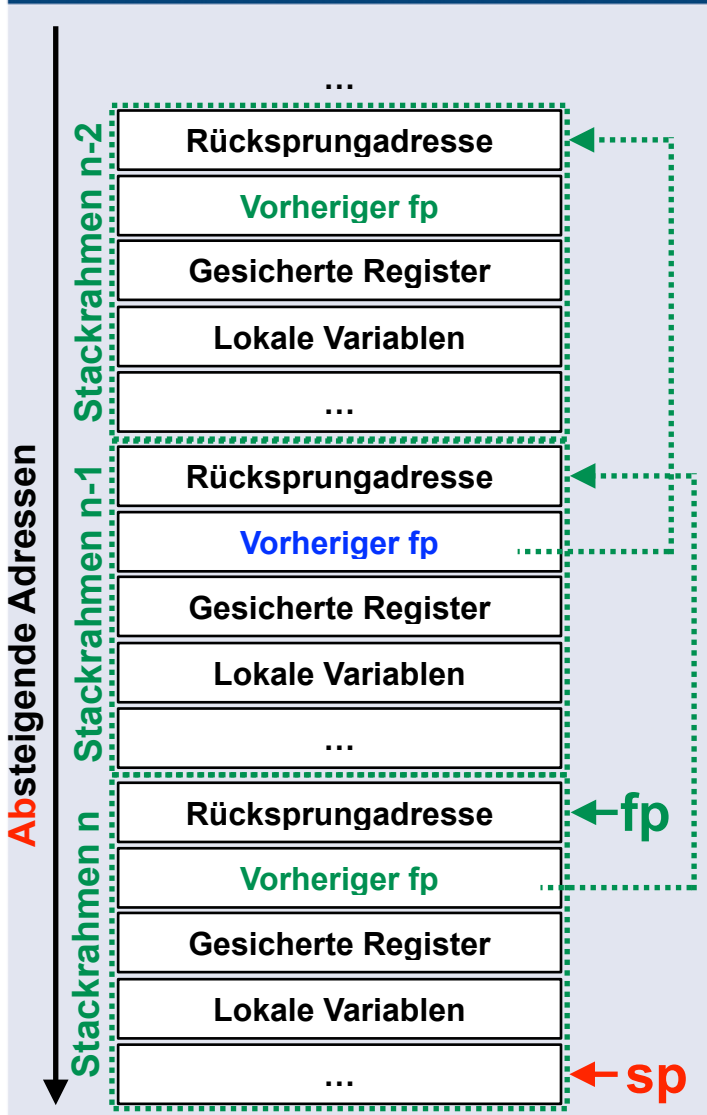
- ***Stack*** ("Stapel" oder "Kellerspeicher"):
  - enthält Werte, die zur Ausführung von Instanzen von Funktionen notwendig sind
- Der Stack ist eine **dynamische Datenstruktur**
  - wächst (nach "unten") beim Aufruf einer Funktion
  - schrumpft wieder, wenn aufgerufene Funktion verlassen wird
- **Inhalte** des Stacks:
  - Rücksprungadresse
  - Funktionsparameter
  - Lokale Variablen und lokal allozierter Speicher
  - Zeiger auf Stack-Rahmen (*stack* frame)
- Der **Stackzeiger sp** zeigt auf das **letzte belegte Element** im Stack

x_i	ABI	Bedeutung
x2	sp	Stackzeiger



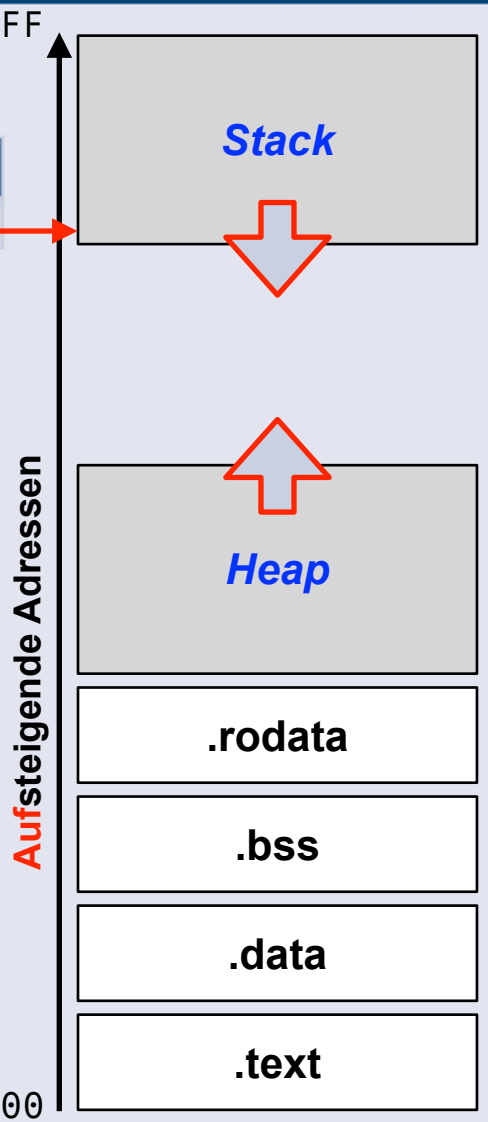
Stack-Struktur im RISC-V-ABI

Universität Bamberg



x_i	ABI	Bedeutung
x2	sp	Stackzeiger

- Beim **Aufruf einer Funktion** wird ein **neuer Stack-Rahmen** (*stack* frame) **unterhalb** des aktuellen Rahmens angelegt
- Der **Stackzeiger sp** und der **Rahmenzeiger fp** werden entsprechend angepasst



Funktionen in RISC-V

```
int foo(int a, int b) {
  int c = 2;
  return add(a, b*2);
}
```

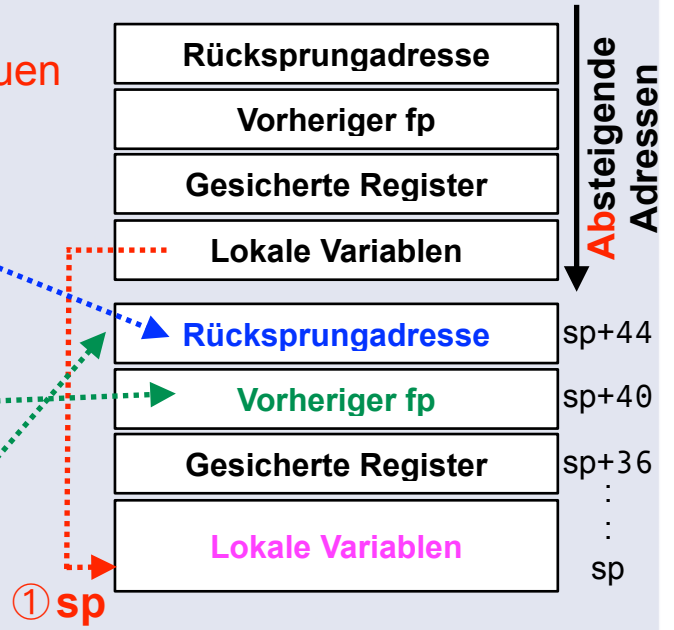
```
foo:
①   addi   sp,sp,-48
②   sw     ra,44(sp)
③   sw     s0,40(sp)
④   addi   s0,sp,48
⑤   sw     a0,-36(s0)
       sw     a1,-40(s0)
⑥   li     a5,2
       sw     a5,-20(s0)
       lw     a4,-40(s0)
       lw     a5,-20(s0)
       mul    a5,a4,a5
       mv     a1,a5
       lw     a0,-36(s0)
       call   add
       mv     a5,a0
⑦   mv     a0,a5
⑧   lw     ra,44(sp)
       lw     s0,40(sp)
       addi   sp,sp,48
⑨   jr     ra
```

① ①①atz auf Stack für neuen Stackrahmen schaffen

② Rücksprungadresse aus `ra` auf dem Stack sichern

③ **Alten** Rahmenzeiger (`s0=fp=x8`) auf dem Stack sichern

④ Rahmenzeiger auf den **neuen Rahmen** zeigen lassen



⑤ Parameter a (Register `a0`) und b (Register `a1`) sichern

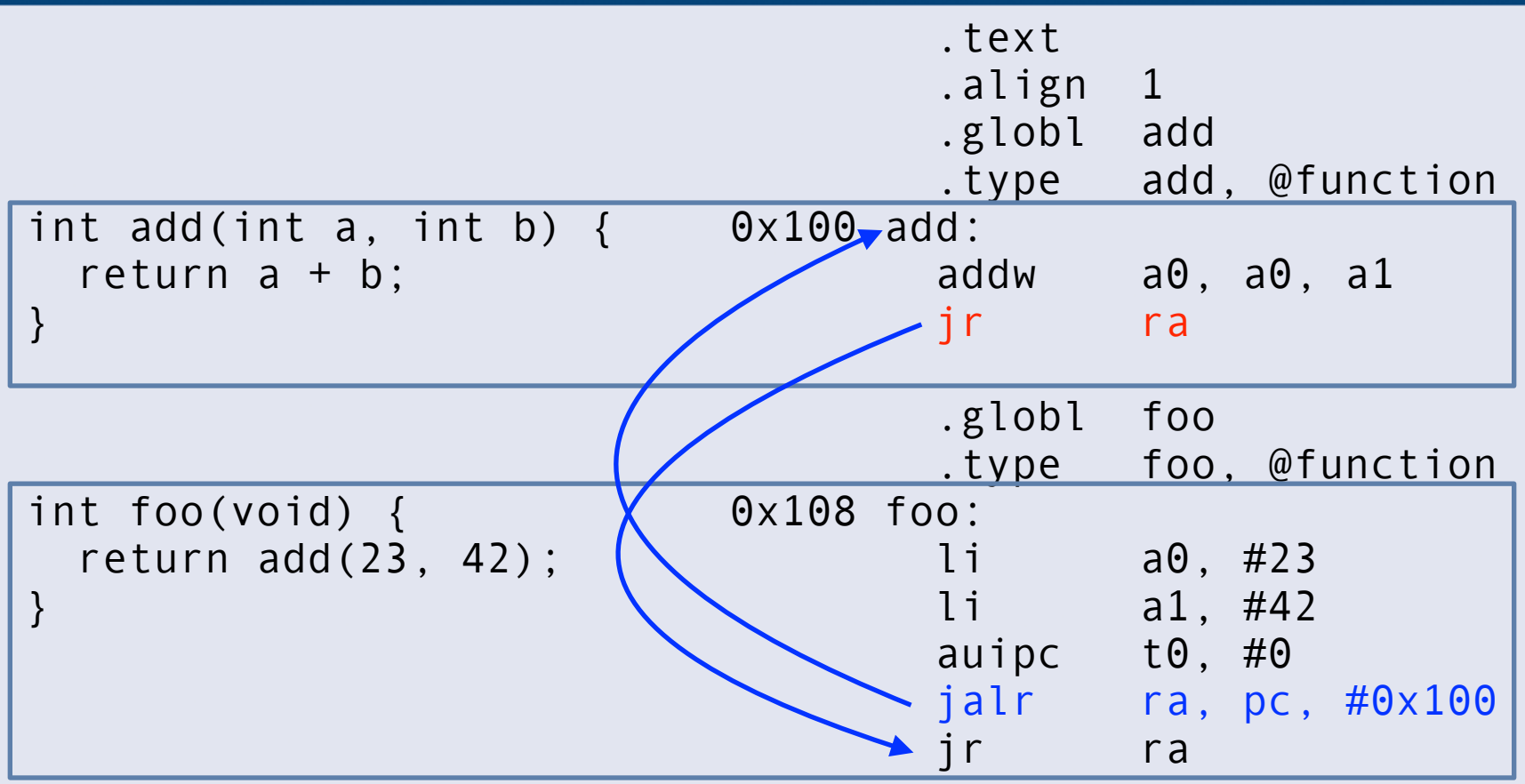
⑥ Hier beginnt der eigentliche Code der Funktion
Variable c wird initialisiert und auf dem Stack gesichert

⑦ Der berechnete Rückgabewert wird in das vom ABI festgelegte Register `a0` geschrieben

⑧ Gesicherte Werte zurückladen und Stack wieder "aufräumen"

⑨ Zur aufrufenden Funktion zurückspringen

Funktionsaufrufe bei RISC-V



`jalr`: "Universeller" Sprungbefehl bei RISC-V:

`JALR rd, rs, immediate`

- Writes `PC+4` to `rd` (return address)
- Sets `PC` = `rs` + `immediate` (12 bit, sign extended)
- Uses same immediates as arithmetic and loads

Funktionsaufrufe bei RISC-V – *Oops?!?*


```
int add(int a, int b) {
  return a + b;
}
```

```
0x100 add:
        addw    a0, a0, a1
        jr      ra  ; Fußnote[1]
```

```
int foo(void) {
  return add(23, 42);
}
```

```
0x108 foo:
        li      a0, #23
        li      a1, #42
        auipc   t0, #0
        jalr    ra, t0, #0x100
        jr      ra
```

Register `ra` wird hier ***überschrieben*** →

`ra` enthält nicht mehr die korrekte Rücksprungadresse!

[1] Die Instruktion **jr ra** ist eine Pseudoinstruktion, die expandiert wird zu **jalr x0, ra, 0**

Sichern der Rücksprungadresse


```
int foo(void) {
  return add(23, 42);
}
```

```
                .globl  foo
                .type   foo, @function
0x108 foo:
                addi    sp,sp,-8
                sd      ra,8(sp)
                li      a0, #23
                li      a1, #42
                auipc   t0, #0
                jalr    ra, t0, #0x100
                ld      ra,8(sp)
                addi    sp,sp,8
                jr      ra
```

Reserviere 8 Bytes im stack und speichere "alten" Wert von `ra`

Hier wird `ra` durch den Aufruf der add-Funktion überschrieben

Hole gesicherten "alten" Wert von `ra` und korrigiere Stackzeiger

`ra` enthält wieder korrekte Rücksprungadr.

Sichern und Wiederherstellen der Rücksprungadresse in ra ist für ***Blattfunktionen*** (*leaf functions*) nicht erforderlich

- Funktionen, die nie eine andere Funktion aufrufen
- Optimierung besonders für Rekursion in funktionalen Sprachen

Start eines Programms

- Wie wird die **allererste Funktion** eines Programms **aufgerufen**?
  - Bei ELF-Dateien existiert ein Eintrag für die **Einsprungadresse** (*entry point address*) im Header
  - Die Einsprungadresse zeigt auf eine Funktion im .text-Segment
- …und **welche Funktion** ist es?
  - bei C-Code: `main()`?… **Nein!**
  - Linker fügt Funktion `_start` hinzu, die main aufruft und…
    - Den Stack initialisiert und `.bss` auf 0 setzt
    - Die Parameter für main setzt (`argc` und `argv`)
    - `main()` aufruft, Programm nach return von main beendet und Rückgabewert an das **Betriebssystem** zurückgibt

Die verwendete Startfunktion wird im Linkerskript festgelegt!

```
$ riscv32-unknown-readelf -h foo
ELF Header:
  …
 Entry point address: 0x1008c
```

```
$ riscv32-unknown-nm foo
# Zeigt Adressen von Code&Vars an
0001008c T _start
000101bc T main
```

Aufgaben eines Betriebssystems

Warum benötigen wir ein Betriebssystem?

- **Starten** von Programmen und **Isolation** verschiedener Programme

- Bereitstellen einer nützlichen **Menge von Diensten** für Programme
  - Verwalten und **abstrahieren** der "low-level"-Hardware
    - z.B. muss eine Textverarbeitung sich nicht damit befassen, welche Art von Festplatte ein Computer verwendet, um Dateien zu speichern

- **Teilen** der Hardware unter mehreren Programmen, damit diese **gleichzeitig** ablaufen können (oder es so erscheint…)

- Bereitstellung **sicherer Wege**, damit Programme untereinander und mit dem Betriebssystem **interagieren** können
  - Daten teilen und kooperative Berechnungen ermöglichen und gleichzeitig die Konsistenz und Vertraulichkeit der Daten des jeweiligen Programms sicherstellen

Betriebssystem als Bibliothek häufig verwendeter Funktionen

- **Wie(so) wurden die ersten Betriebssysteme entwickelt?**
  - Frühe Computer (50er) führten nur ein Programm gleichzeitig aus
  - Programm laden → ausführen → Ergebnisse drucken → nächstes…
- Einige Funktionalität wurde von jedem Programm neu implementiert
  - z.B. Zugriff auf Festplatten oder Drucker
  - Aufwändig und fehleranfällig
- ***Bibliotheken*** *(libraries)* ***wiederverwendbarer gemeinsamer Funktionen*** wurden entwickelt
- Nutzer wollen Bibliothek nicht bei jedem Programmwechsel neu laden (dies dauerte jeweils einige Minuten!)
  - Bibliothek von Programmen getrennt geschützt im Speicher abgelegt
  - ***Privilegienmodi*** wurden eingeführt, um den Zugriff auf den Speicher und die Funktionen der Bibliothek von "normalen" Programmen zu beschränken

Vom Programm zum Prozess

- Das Betriebssystem startet unser Programm an Adresse `_start`
  - …die Start-Funktion ruft danach `main` auf
  - Ein **Programm in Ausführung** bezeichnen wir als ***Prozess***
- Das Programm läuft jetzt und führt aus:
  - Variablenzuweisungen, Kontrollstrukturen, Funktionsaufrufe…
  - Programm **ändert** seinen **Zustand**
    - Wir kennen schon: **Prozessorregister**
  - Außerdem:
    - Hauptspeicher (RAM) – `.data`, `.bss`, Stack und Heap
  - **Mehr?**
    - Aufruf von **Funktionen des Betriebssystems**!
    - Sind dies normale Funktionsaufrufe (wie gerade gesehen)?

Übrigens: RISC-V Code optimiert

Universität Bamberg

Quellcode

Zeichenstrom

Lexikalische Analyse → Syntax-analyse → Semantische Analyse → Code-erzeugung → **Code-optimierung** → ***Objektcode***

C

```
int x = 42;
int y = 23;

int add(int a, int b) {
  return a+b;
}

int foo(int a, int b) {
  int c = 2;
  return add(a, b*2);
}

int main(int argc, char **argv) {
  return foo(x,y);
}
```

Assembler

```
foo:
    slli  a1,a1,1
    add   a0,a1,a0
    ret
```

Experimentieren mit dem Compiler Explorer von Matt Godbolt:
https://godbolt.org/

Übersetzt mit:

```
riscv64-unknown-elf-gcc -march=rv32i \
    -mabi=ilp32 -O3 -S foo.c
```

Referenzen

[1] Remzi Arpaci-Dusseau, Andrea Arpaci-Dusseau
Operating Systems: Three Easy Pieces
https://pages.cs.wisc.edu/~remzi/OSTEP/

[2] John R. Levine, **Linkers and Loaders**, MKP 1999, ISBN 1-55860-496-0
https://www.iecc.com/linker/

[3] ELF TIS Committee, Tool Interface Standard (TIS)
Executable and Linking Format (ELF) Specification, Version 1.2
https://www.uclibc.org/docs/elf.pdf

[4] **RISC-V ELF psABI** Document
https://github.com/riscv-non-isa/riscv-elf-psabi-doc/

[5] Russ Cox, Frans Kaashoek, Robert Morris
xv6: a simple, Unix-like teaching operating system
https://pdos.csail.mit.edu/6.828/2023/xv6/book-riscv-rev3.pdf