```
FUN:    sw $ra,-4($sp)
        sw $s0,-8($sp) # $s0 für C
        sw $s1,-12($sp) # $s1 für j
        sw $s2,-16($sp) # $s2 für i
        sw $s3,-20($sp) # $s3 für m
        addi $sp,$sp,-20
        move $s0,$a0
        move $s1,$a1
        move $s2,$a3
        addi $s3,$a2,9
        move $a0,$s2
        move $a1,$s1
        jal V
        add $s3,$s3,$v0
        move $a0,$s1
        move $a1,$s2
        jal X
        add $s3,$s3,$v0
        sll $t0,$s3,2
        add $t0,$t0,$s0
        lw $v0,0($t0)
        addi $sp,$sp,20
        lw $ra,-4($sp)
        lw $s0,-8($sp) # $s0 für C
        lw $s1,-12($sp) # $s1 für j
        lw $s2,-16($sp) # $s2 für i
        lw $s3,-20($sp) # $s3 für m
        jr $ra
```

Rechnerarchitektur (AIN 2)

SoSe 2021

Kapitel 2

Befehle: Die Sprache des Rechners

Prof. Dr.-Ing. Michael Blaich
mblaich@htwg-konstanz.de

Vorlesungsinhalt

Kapitel 1: Grundlegende Ideen, Technologien, Komponenten

Kapitel 2: Befehle: Die Sprache des Rechners

…

2.5 Kontrollstrukturen

2.6 MIPS Assembler und MARS Simulator

2.7 Weitere MIPS-Befehle

2.8 Compiler, Assembler, Linker, Loader

2.8.1 Schritte vor Ausführung eines C Programms

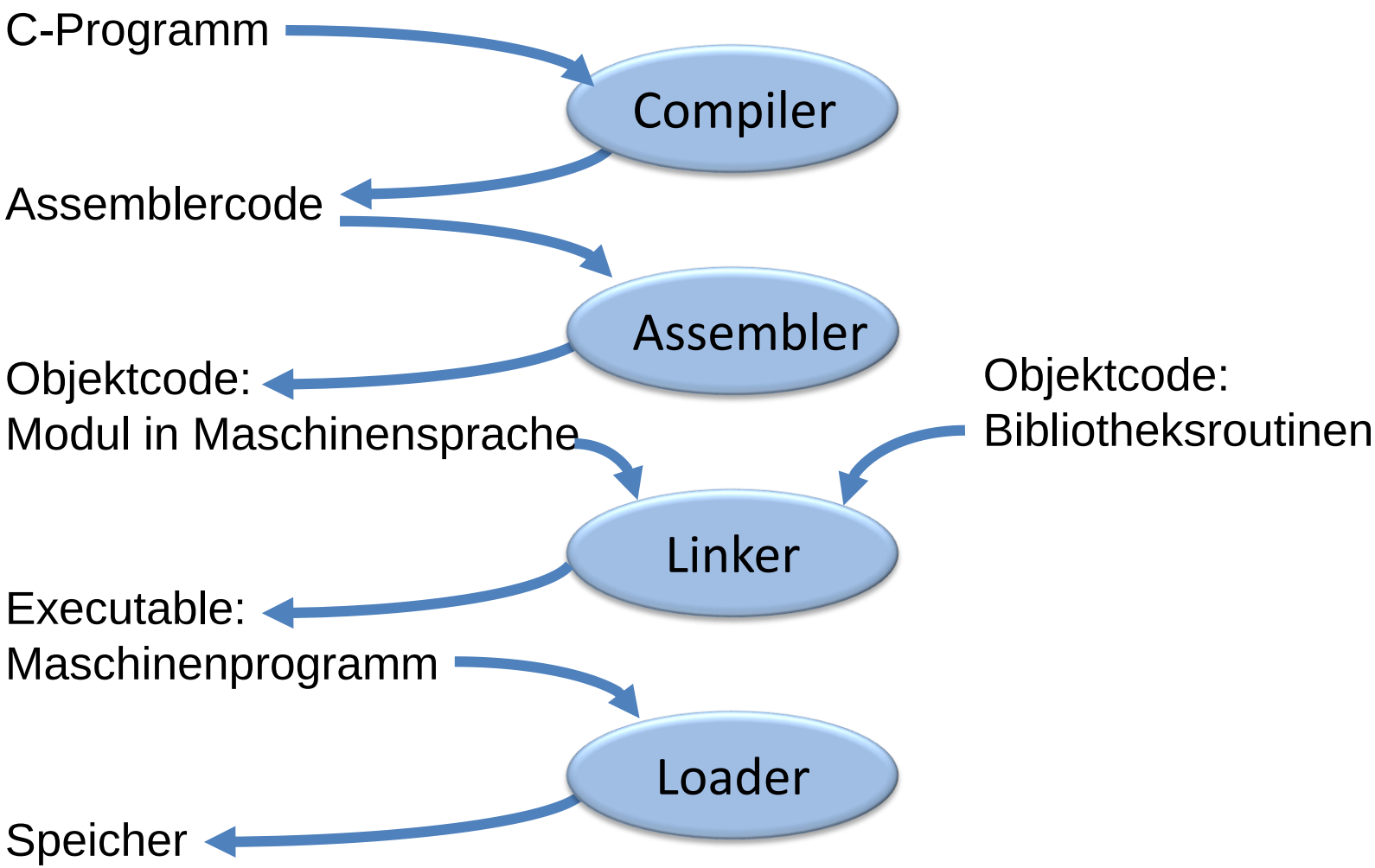
2.8.2 Assembler

2.8.3 Linker und Loader

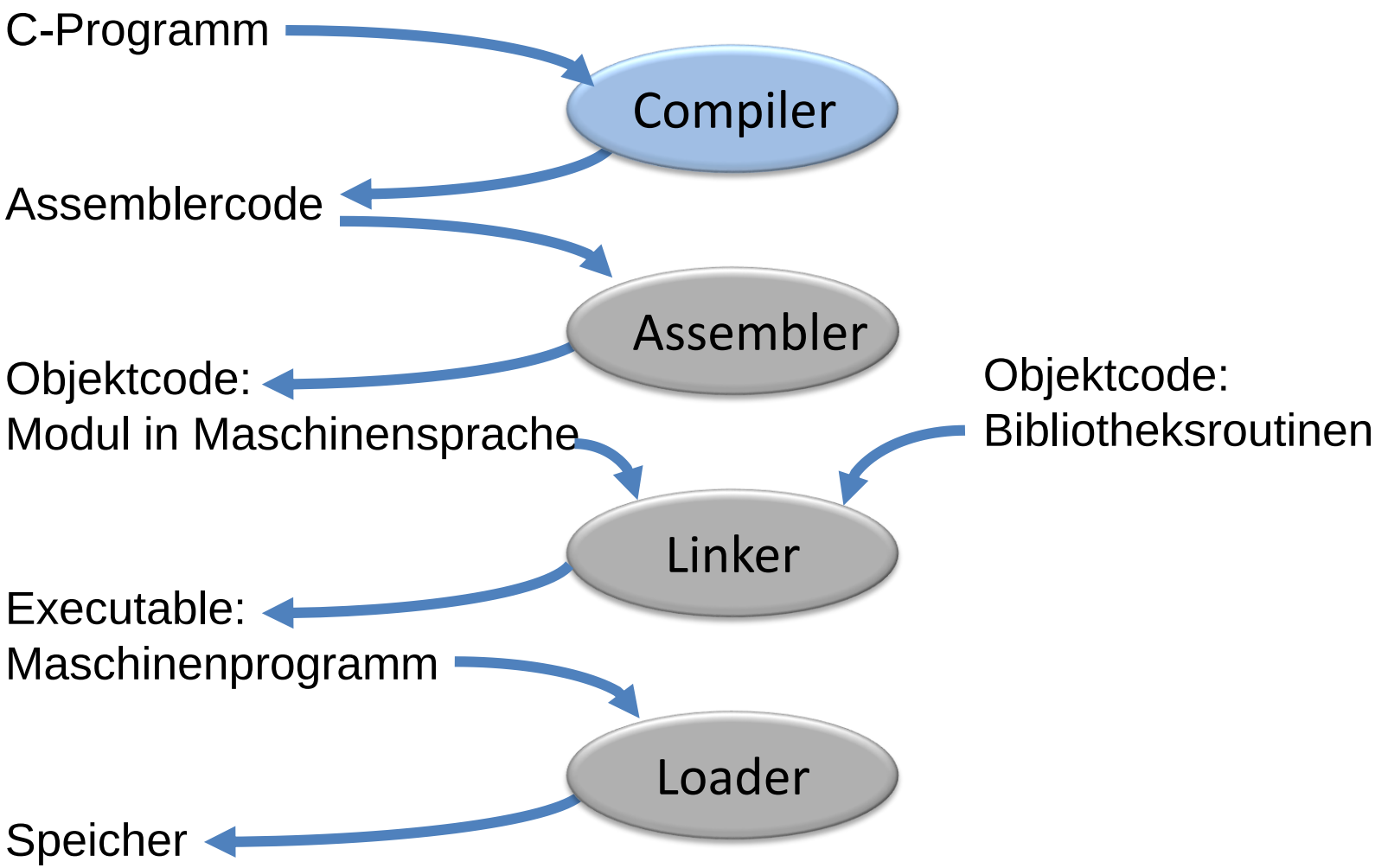
2.8.4 Dynamic Linked Libraries (DLL)

2.8.5 Laufzeitvergleich

Wie kommt ein C-Programm in den Speicher?



Wie kommt ein C-Programm in den Speicher?



Compiler

Compiler - Übersetzung von Hochsprachen in Assembler

- Hochsprachenprogrammierung wird verwendet,
  - da die Programmiereffizienz hier ungleich höher ist als bei der Verwendung der Maschinensprache,
  - da Programme in Hochsprachen (mehr oder weniger) maschinenunabhängig sind.
- Moderne Compiler erzeugen Assemblercode, der so gut optimiert ist, dass er manuell kaum besser erzeugt werden könnte.
- Während höhere Programmiersprachen (weitestgehend) maschinenunabhängig sind, muss für jede Zielarchitektur ein spezieller Compiler bereit gestellt werden.
- Cross-Compiler laufen auf einer Host- Plattform und compilieren Code für eine andere Ziel-Plattform
  - z.B. kann auf einer Windows Host-Plattform mit x86-Architektur Code für eine MIPS Ziel-Architektur erzeugt werden
- Beispiel für einen Online C-to-MIPS-Compiler: https://godbolt.org/

Manuelle Assembler-Programmierung

- zur Optimierung spezieller Codesequenzen
- z.B. um Reaktionszeit einer Bremse zur garantieren
- wenn es keinen (guten) Compiler gibt

Vorlesungsinhalt

Kapitel 1: Grundlegende Ideen, Technologien, Komponenten

Kapitel 2: Befehle: Die Sprache des Rechners

…

2.5 Kontrollstrukturen

2.6 MIPS Assembler und MARS Simulator

2.7 Weitere MIPS-Befehle

2.8 Compiler, Assembler, Linker, Loader

2.8.1 Schritte vor Ausführung eines C Programms

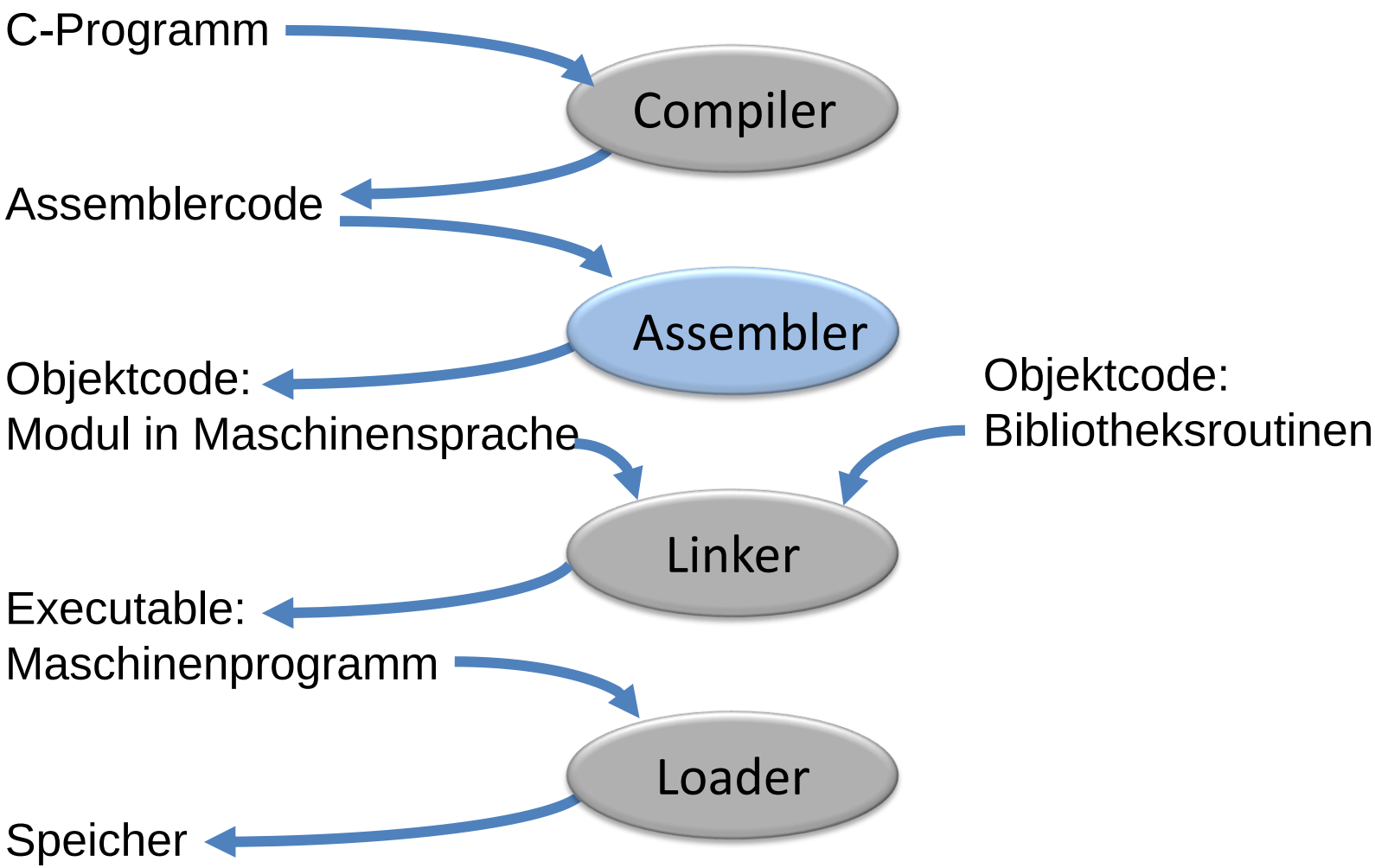
2.8.2 Assembler

2.8.3 Linker und Loader

2.8.4 Dynamic Linked Libraries (DLL)

2.8.5 Laufzeitvergleich

Wie kommt ein C-Programm in den Speicher?



Arbeitsweise eines Assembler

Aufgaben des Assemblers:

- Auflösen von Pseudo-Instruktionen und Markos
- Auflösen von symbolischen Adressen (Labels)

Globale und lokale Labels

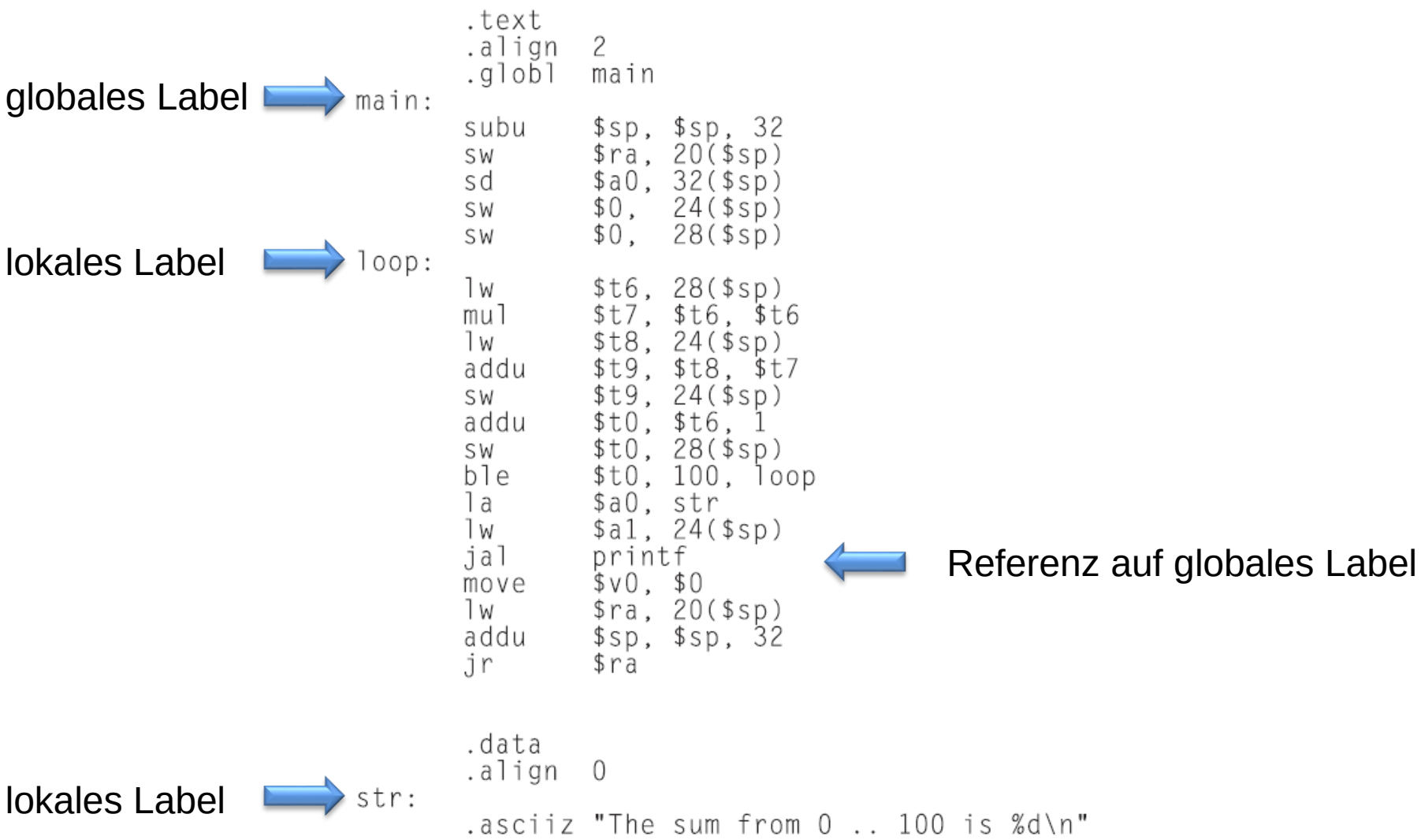
- lokal: Label in gleicher Datei/Code-Segment
- global, extern: Label in referenzierter Datei/Code-Segment

Forward Reference Problem:

- Sprünge zu einem Label, das sich „weiter vorne“ im Code befindet
- Assembler kann das Label nicht auflösen, ohne die „Nummer“ der Code-Zeile des Labels zu kennen

→ aufgrund von Vorwärtssprüngen benötigt der Assembler zwei Durchgänge durch den Assembler-Code

Lokale und globale Labels



Assembler: Zwei Durchgänge

Erster Durchlauf

- Auflösung von Pseudo-Instruktionen und Makros
  - wie viele Zeilen Maschinen-Code braucht eine Zeile Assembler-Code
- produziert Symboltabelle
  - Zuordnung von Labels zu Speicheradressen/Code-Zeilen
- Speicheradressen von „statischen“ Daten werden bestimmt

Zweiter Durchlauf

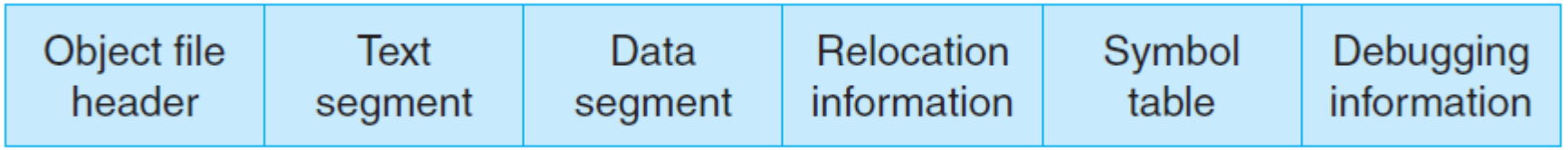
- produziert Code in Maschinensprache (object file)
- ersetzt lokale Label durch Speicheradressen

Alternative: Backpatching

- erster Durchgang produziert lückenhaften Code in Maschinensprache
- im zweiten Durchgang werden die Lücken anhand einer Tabelle gefüllt
- Vorteil: schneller
- Nachteil: benötigt mehr Speicher (vollständiger Code in Maschinensprache)

Assembler produziert Object-Code

Objektfiles enthalten alle Informationen, die für die weitere Bearbeitung erforderlich sind.



Struktur

- Header: Beschreibt die Größe und Position der übrigen Teile des Objektfiles

Code und Daten

- Textsegment: Code in Maschinensprache, kann nicht aufgelöste Referenzen (externe Labels) enthalten
- Datensegment:
  - statische Daten: werden während Programmlaufzeit allokiert
  - dynamische Daten: können Größe während Programmlaufzeit verändern

Informationen für den Linker

- Relokationsinformation: Identifikation von Instruktionen und Daten, die von absoluten Speicheradressen abhängig sind
- Symboltabelle: noch undefinierte Labels (z.B. externe Sprungziele)

Debugging Information

- Informationen, die es dem Debugger ermöglichen, Maschinenbefehle mit C-Sourcecode in Verbindung zu bringen

Vorlesungsinhalt

Kapitel 1: Grundlegende Ideen, Technologien, Komponenten

Kapitel 2: Befehle: Die Sprache des Rechners

…

2.5 Kontrollstrukturen

2.6 MIPS Assembler und MARS Simulator

2.7 Weitere MIPS-Befehle

2.8 Compiler, Assembler, Linker, Loader

2.8.1 Schritte vor Ausführung eines C Programms

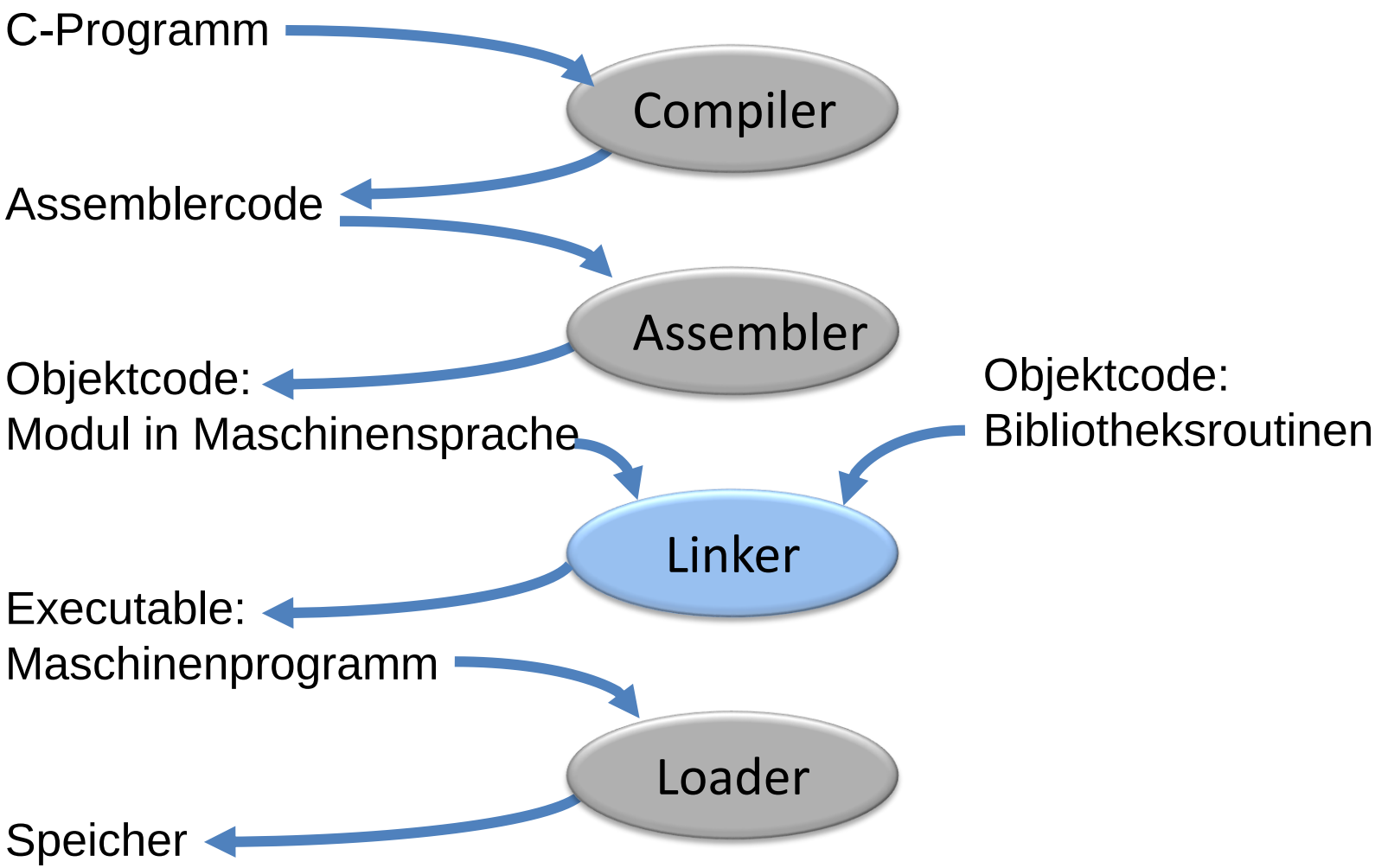
2.8.2 Assembler

2.8.3 Linker und Loader

2.8.4 Dynamic Linked Libraries (DLL)

2.8.5 Laufzeitvergleich

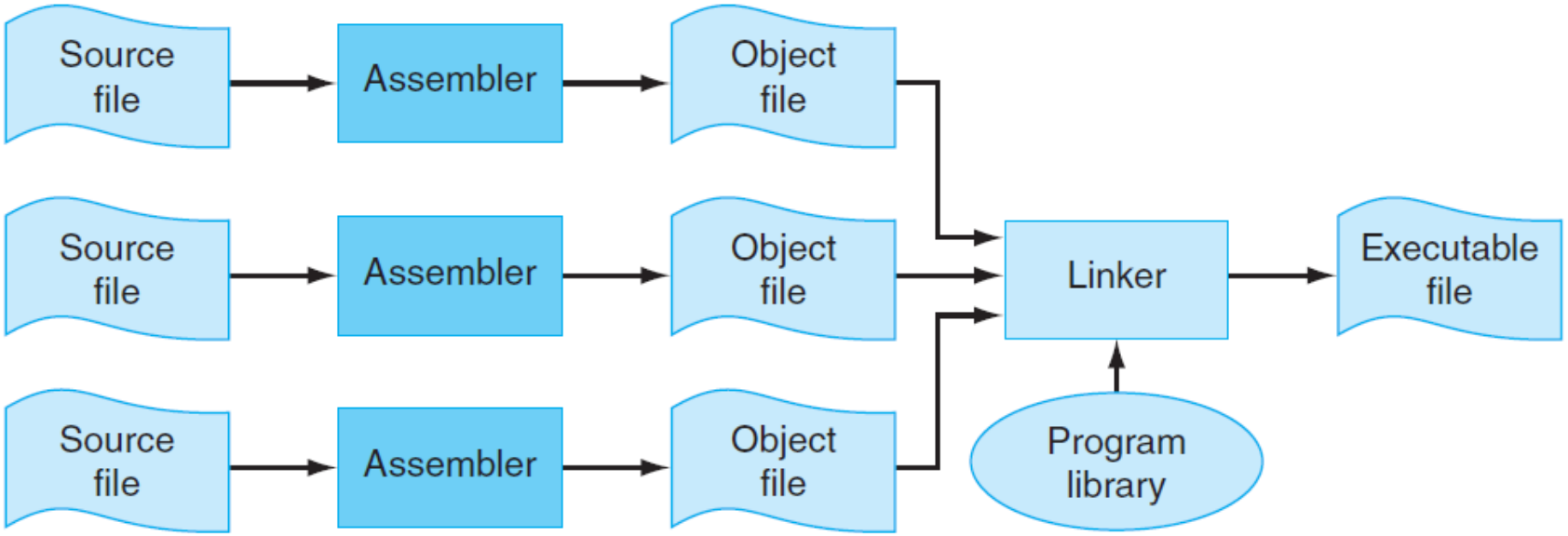
Wie kommt ein C-Programm in den Speicher?



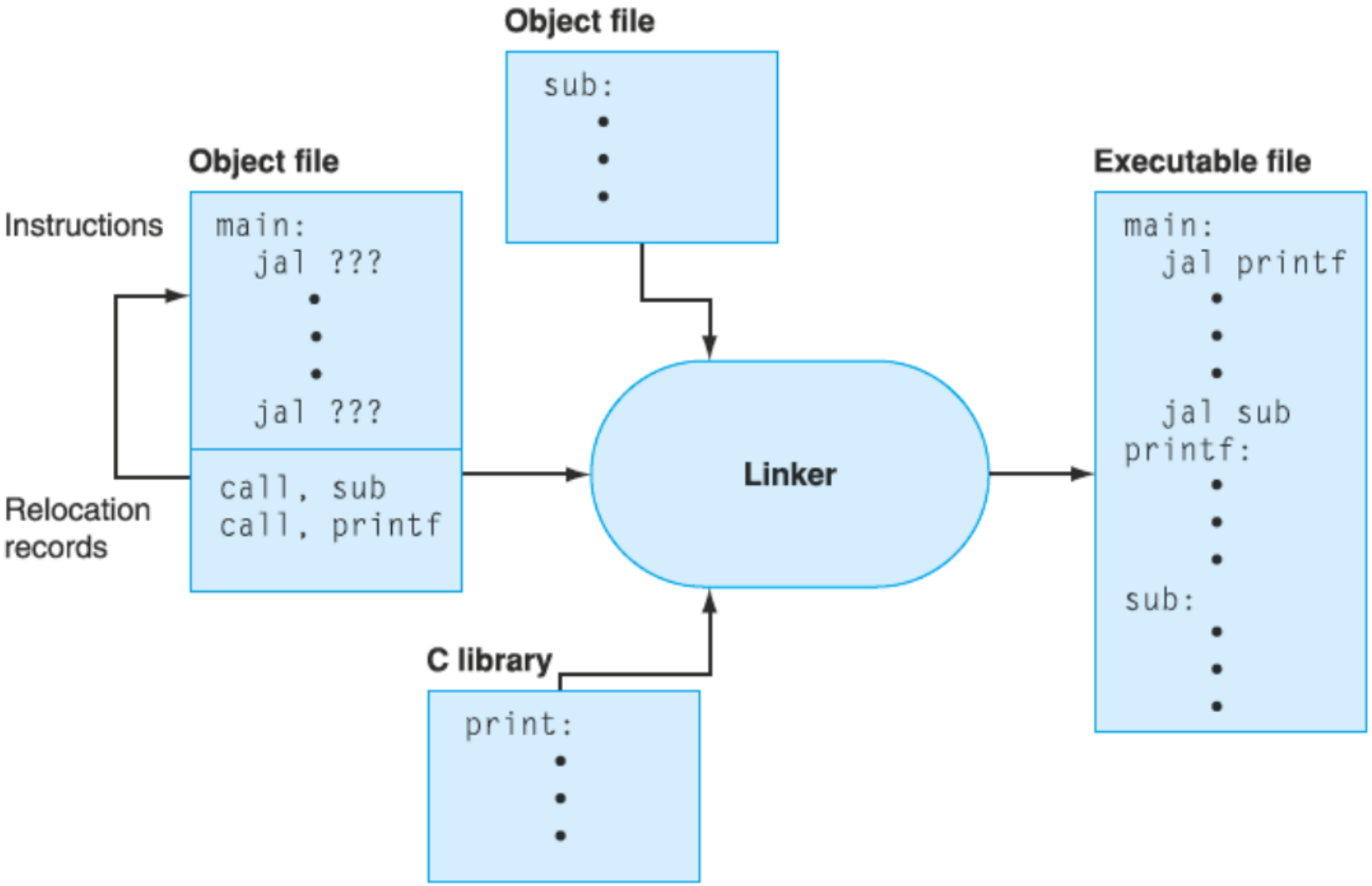
Linker

- Um eine weitgehende Wiederverwendung von Programmteilen zu ermöglichen, werden Programme Prozedur für Prozedur übersetzt
- Für jedes einzelne der so entstehenden Programmstücke müssen die relativen Positionen innerhalb des Speichers bestimmt werden
- Aufgaben des Linkers:
  - Symbolische Platzierung von Code und Daten im Speicher
  - Bestimmung der Adressen von Daten- und Instruktionslabels
  - „Patchen“ von internen und externen Referenzen, d.h. Einsetzen der ermittelten Sprungweiten bzw. Sprungziele
  - Das Executable hat dasselbe Format wie das Objektfile, nur mit aufgelösten Referenzen
- Im Executable müssen alle Referenzen aufgelöst sein
  - Ausnahme: DLL (Dynamic Linked Libraries)

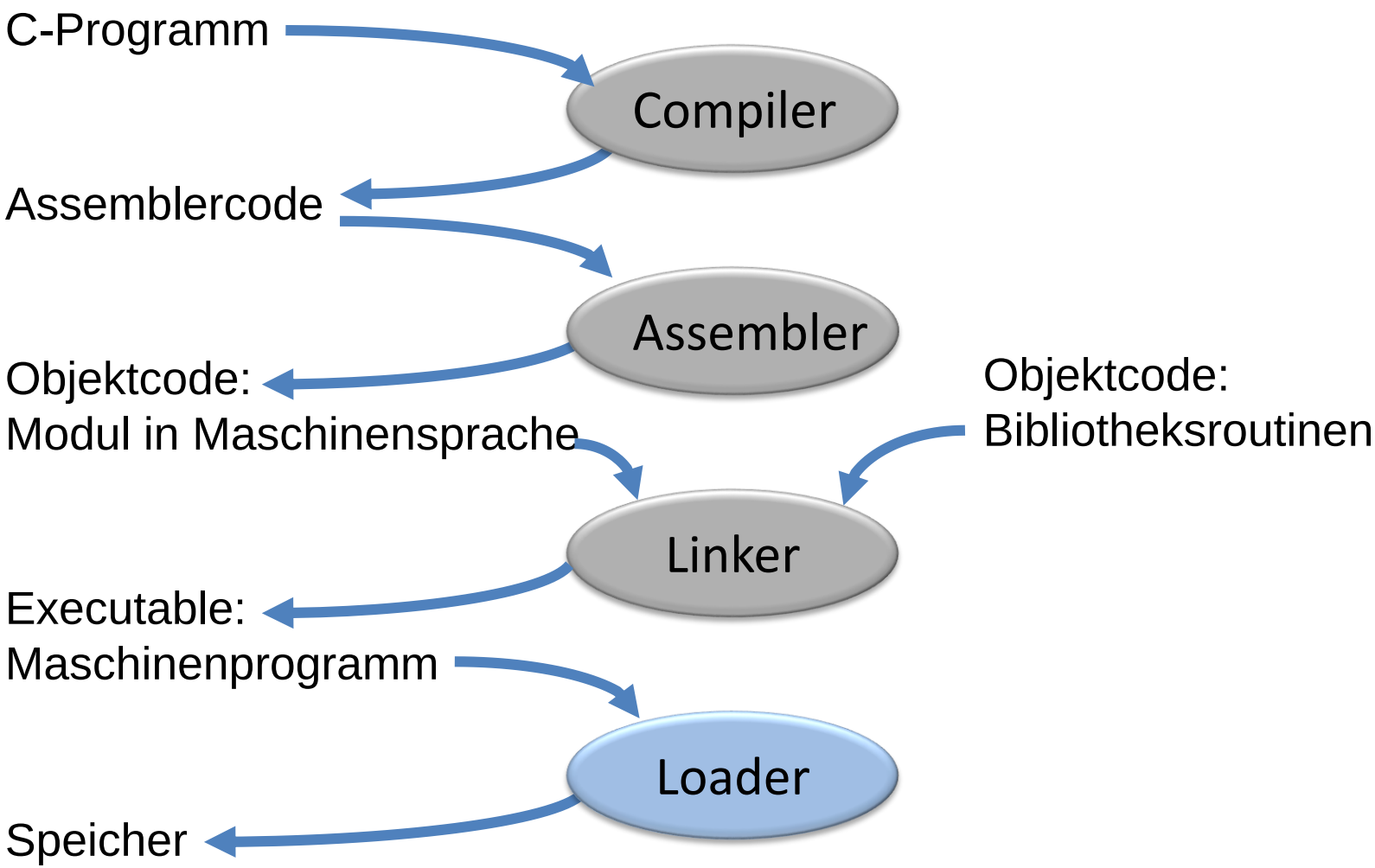
Linken mehrerer Module (Source-Files)



Arbeitsweise Linker



Wie kommt ein C-Programm in den Speicher?



Loader

Betriebssystemroutine, die ein Programm zur Ausführung auf dem Rechner bringt.

Dabei werden in Unix die folgenden Schritte ausgeführt:

- **Lesen des Programms** und Bestimmen der Programm- und Datensegmentgrößen
- Bereitstellen eines hinreichend großen **Adressraums** für Programm und Daten
- **Kopieren** der Instruktionen und Daten aus dem Programm **in den Speicher**
- **Initialisierung** der Maschinenregister; Setzen des Stackpointers auf die erste freie Position
- Sprung zu einer **Start-up-Routine**, die Parameter in die Argumentregister schreibt und zur main-Prozedur des Programms verzweigt

Vorlesungsinhalt

Kapitel 1: Grundlegende Ideen, Technologien, Komponenten

Kapitel 2: Befehle: Die Sprache des Rechners

…

2.5 Kontrollstrukturen

2.6 MIPS Assembler und MARS Simulator

2.7 Weitere MIPS-Befehle

2.8 Compiler, Assembler, Linker, Loader

2.8.1 Schritte vor Ausführung eines C Programms

2.8.2 Assembler

2.8.3 Linker und Loader

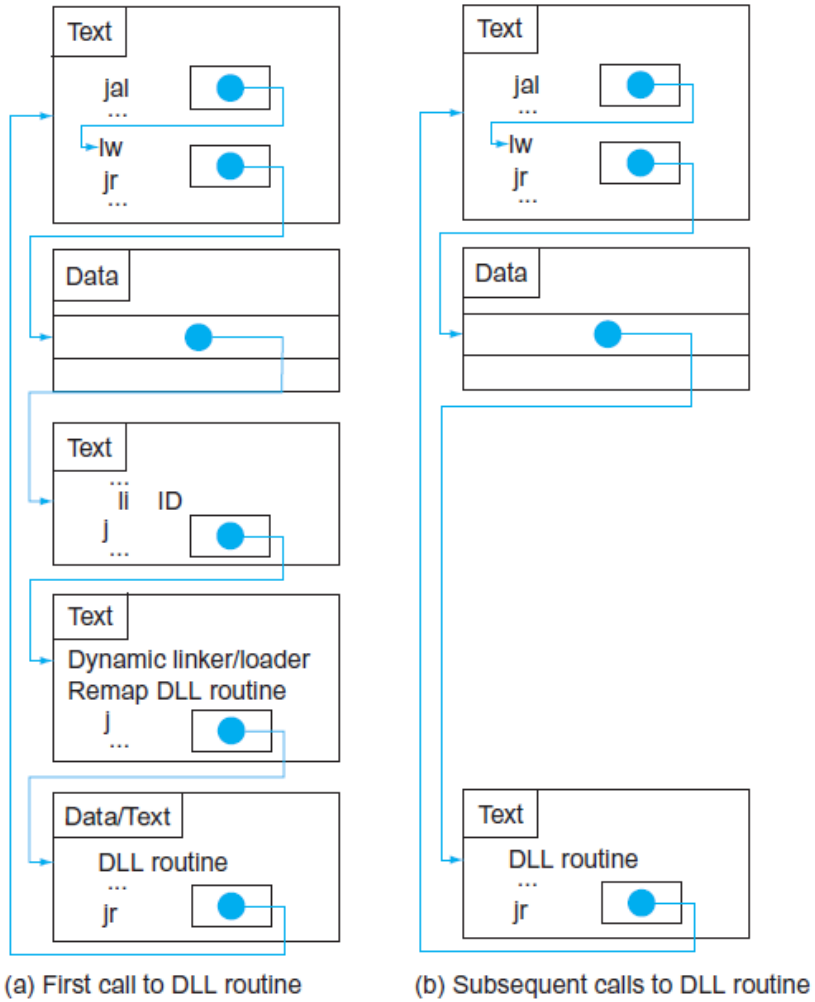
2.8.4 Dynamic Linked Libraries (DLL)

2.8.5 Laufzeitvergleich

Dynamic Linked Libraries (DLL)

- Library-Routinen werden nicht vom Loader eingebunden sondern während der Laufzeit nachgeladen
  - nicht immer werden alle Routinen einer Library benötigt
  - reduziert Programm-Code auf benutzte Routinen
  - Bug Fixes für Libraries können vorgenommen werden, ohne Programme zu ändern

- Prinzip:
  - Routine wird beim ersten Aufruf dem Programm-Code hinzugefügt und die Code-Adresse an entsprechende Stelle im Speicher eingefügt



(a) First call to DLL routine

(b) Subsequent calls to DLL routine

Vorlesungsinhalt

Kapitel 1: Grundlegende Ideen, Technologien, Komponenten

Kapitel 2: Befehle: Die Sprache des Rechners

…

2.5 Kontrollstrukturen

2.6 MIPS Assembler und MARS Simulator

2.7 Weitere MIPS-Befehle

2.8 Compiler, Assembler, Linker, Loader

2.8.1 Schritte vor Ausführung eines C Programms

2.8.2 Assembler

2.8.3 Linker und Loader

2.8.4 Dynamic Linked Libraries (DLL)

2.8.5 Laufzeitvergleich

Wie messe ich die Leistung eines Computers?

Leistungsbewertungen und -vergleiche benötigen exakte Definition, was gemessen wird

- CPU-Ausführungszeit (kurz CPU-Zeit):
  - die Zeit, während der die CPU für die Bewältigung einer Aufgabe aktiv ist (z.B. ohne Zeit für I/O oder andere Programme)
  - besteht aus

- Benutzer-CPU-Zeit:
  - Anteil der CPU-Zeit für die Abarbeitung innerhalb eines Programms

- System-CPU-Zeit:
  - Anteil der CPU-Zeit für die Abarbeitung von Betriebssystemaufrufen für ein Programm

Performanz eines Programms?

Programmperformanz: Taktgeschwindigkeit und Anzahl der Taktzyklen bestimmen die CPU-Zeit eines Programms

- Prozessortakt:
  - bestimmt, wann Ereignisse innerhalb der Hardware ausgeführt werden
- Taktzyklus:
  - Zeit für ein Taktintervall
  - auch Tick, Taktintervall, Takt
- Taktgeschwindigkeit:
  - Anzahl Taktintervalle pro Zeiteinheit
  - Beispiel: Taktgeschwindigkeit=4GHz⇔ Taktzyklus=250ps
- Programmperformanz:

$$\text{CPU Zeit eines Programms} = \frac{\text{Anzahl CPU Taktzyklen für das Programm}}{\text{Taktgeschwindigkeit der CPU}}$$

Beispiel

Verringerung der Ausführungszeit um 40%

- bei Erhöhung der Anzahl Taktzyklen um 20%
- erfordert Erhöhung der Taktgeschwindigkeit um 100%

- Computer A:
  - Taktgeschwindigkeit: 2GHz, Ausführungszeit für Programm A: 10s
- Design für Computer B:
  - Zielausführungszeit: 6s
  - Taktgeschwindigkeit kann auf Kosten einer 1.2-fach größeren Anzahl von Taktzyklen erhöht werden
- Welche Taktgeschwindigkeit benötigt Computer B?

$$\text{Taktgeschwindigkeit B} = \frac{\#\text{Taktzyklen B}}{\text{Ausführungszeit B}} = \frac{1.2\times\#\text{Taktzyklen A}}{6s}$$

$$\#\text{Taktzyklen A} = \text{Ausführungszeit A} \times \text{Taktgeschwindigkeit A} = 10\text{s} \times 2\text{GHz} = 2\times10^{10}$$

$$\text{Taktgeschwindigkeit B} = \frac{1.2\times\#\text{Taktzyklen A}}{6s} = \frac{1.2\times2\times10^{10}}{6s} = 4GHz$$

Performanz auf Befehlsebene

- Ausführungszeit von unterschiedlichen Befehle kann verschieden sein
- Schnellere Befehle können ein Programm schneller machen, besonders im Falle von paralleler Abarbeitung

- Anzahl Instruktionen für ein Programm
  - bestimmt durch Programm, ISA und Compiler
- Mittlere Anzahl von Taktzyklen pro Instruktion
  - CPI: Clock Cycles per Instruction
  - bestimmt von der CPU Hardware
  - bestimmt von der **Art von Befehlen im Programm**
- Leistungsgrößen eines Programm (CPU-Taktzyklen und Zeit):

$$Anzahl\ CPU\ Taktzyklen = Instruktionsanzahl \times CPI$$

$$CPU\ Zeit = \frac{Instruktionsanzahl \times CPI}{Taktgeschwindigkeit}$$

Ein abstraktes Beispiel

- Die Hardware benötigt für die Instruktionen A, B und C die folgende Anzahl Taktzyklen:

Instruktion	A	B	C
Taktzyklen	1	2	3

- Der Compilerdesigner hat folgende Alternativen, um eine Codesequenz mit der folgenden Anzahl von Instruktionen A, B und C umzusetzen:

Instruktion	A	B	C
Variante 1	2	1	2
Variante 2	4	1	1

- Welche Variante ist schneller? Was ist der CPI der beiden Varianten?

Ein abstraktes Beispiel

- Anzahl Instruktionen

Instruktion	A	B	C	Gesamt
Variante 1	2	1	2	5
Variante 2	4	1	1	6

- Anzahl Taktzyklen

Instruktion	A	B	C	Gesamt
Variante 1	2x1=2	1x2=2	2x3=6	10
Variante 2	4x1=4	1x2=2	1x3=3	9

- CPI

Instruktion	Taktzyklen	Instruktionen	CPI
Variante 1	10	5	2
Variante 2	9	6	1,5

Noch ein Beispiel

- Eine Java-Anwendung läuft in 15 Sekunden auf einem Desktop-Prozessor. Ein neuer Java-Compiler kommt heraus, der nur noch 60% der Anzahl Instruktionen benötigt. Allerdings steigt dafür der CPI um 10%. Wie schnell läuft die Anwendung mit dem neuen Compiler?

a. $\frac{15 \times 0.6}{1.1} = 8.2 \text{ sec}$

b. $15 \times 0.6 \times 1.1 = 9.9 \text{ sec}$ ✓

c. $\frac{15 \times 1.1}{0.6} = 27.5 \text{ sec}$

➡ Warum „x 0.6“? Weniger Instruktion bedeutet kleinere Laufzeit

➡ Warum „x 1.1“? Mehr Taktzyklen pro Instruktion bedeutet größere Laufzeit

Leistungsmessung zusammengefasst

- CPU-Ausführungszeit ist die wichtigste Leistungskomponente
- Programmierung, Übersetzung in Maschinensprache und Hardwareschnittstelle beeinflussen Ausführungszeit eines Programms

Leistungskomponenten	Maßeinheit
CPU-(Ausführungs-)Zeit für ein Programm	Sekunden für das Programm
Anzahl Instruktionen für ein Programm	Anzahl
CPI	Mittlere Anzahl Taktzyklen pro Instruktion
Taktzykluszeit, Taktintervall	Sekunden pro Taktzyklus

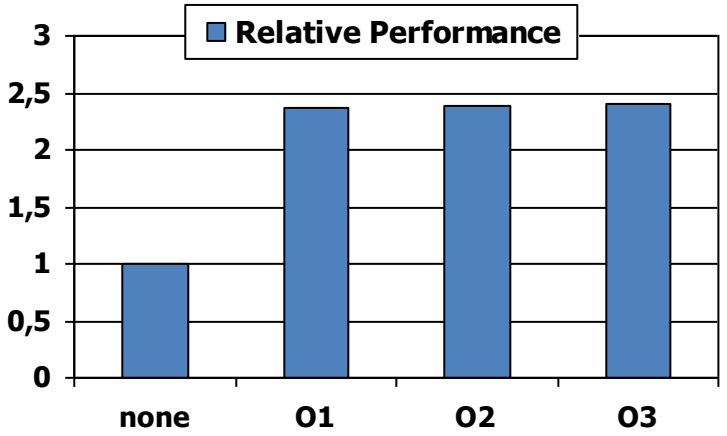
Einflussfaktoren	Beeinflusste Leistungskomponente
Algorithmus	Anzahl Instruktionen, CPI
Programmiersprache	Anzahl Instruktionen, CPI
Compiler	Anzahl Instruktionen, CPI
ISA (Instruction Set Architecture)	Anzahl Instruktionen, CPI, Taktgeschwindigkeit

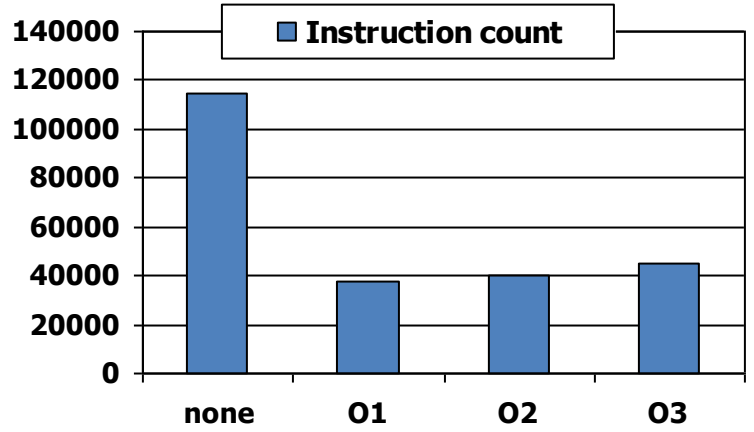
Compiler Optimierung – Beispiel Bubblesort

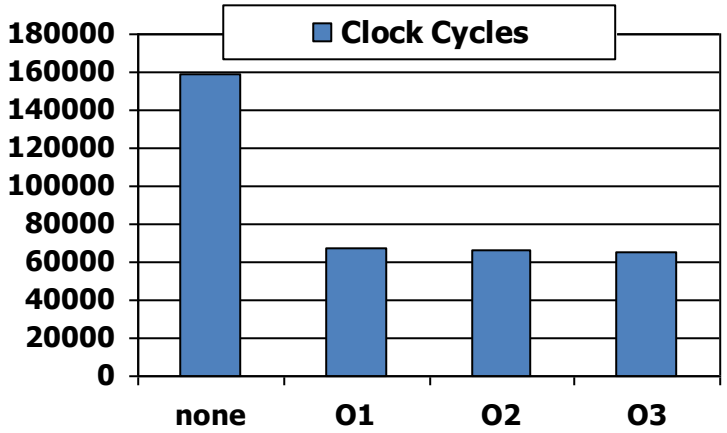
Laufzeitvergleich von Bubble-Sort kompiliert mit gcc für einen Pentium 4 unter Linux

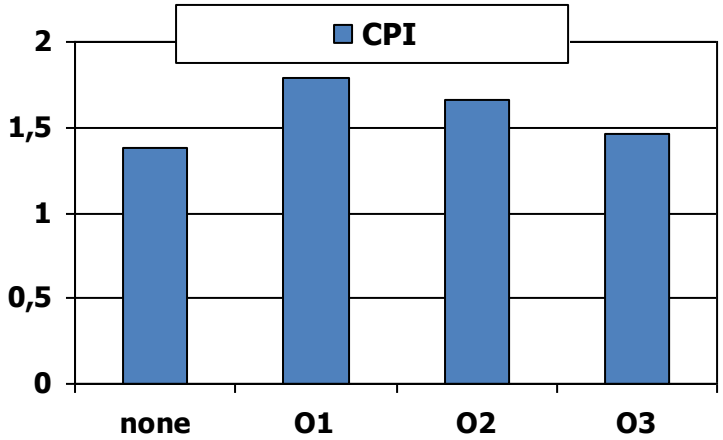
- Optimierungsgrad des Compilers: none, Q1, Q2, Q3

- Aussagen:
  - Optimierungsoptionen des Compilers führen zu einer wesentlich verbesserten Laufzeit
  - Compiler optimiert zunächst (Q1) die Anzahl der Instruktionen und dann (Q2, Q3) die Komplexität, den CPI, der Instruktionen





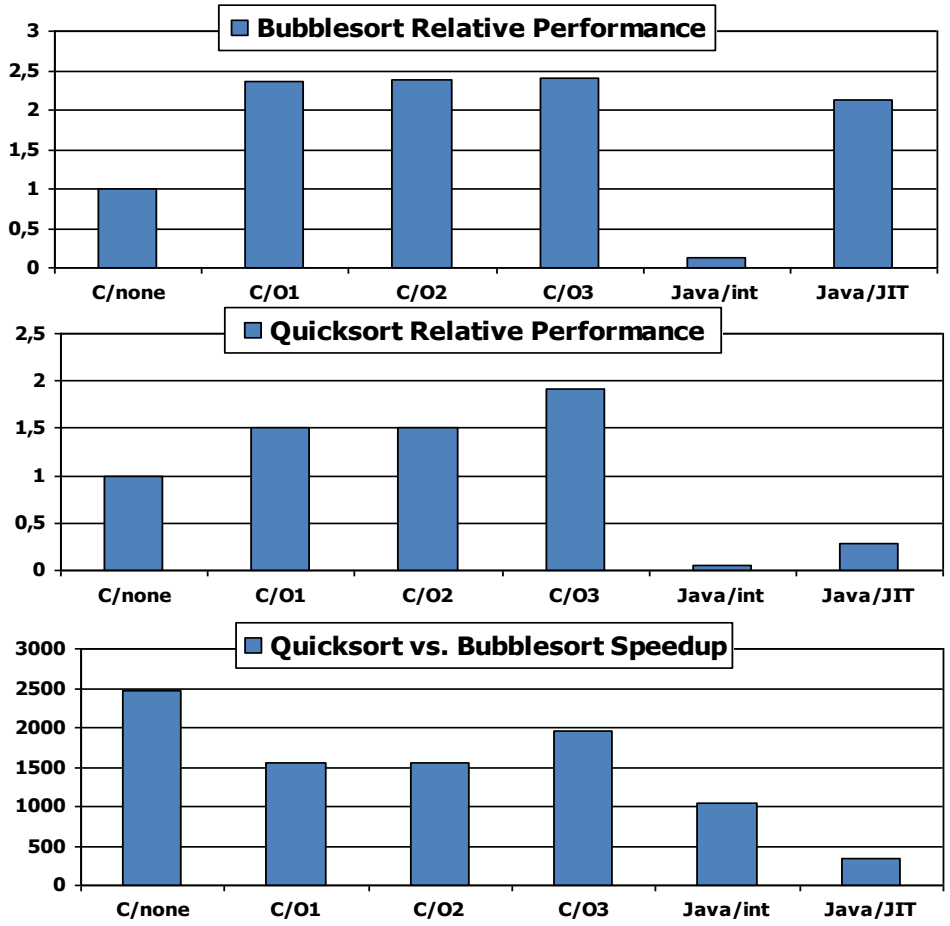




Compiler Optimierung – Beispiel Bubblesort

Vergleich der Sortier-Algorithmen Bubblesort und Quicksort in den Programmiersprachen C (unterschiedliche Optimierungsgrade) und Java (Interpreter und Just-in-Time Compiler)

- Die Auswirkung der Compilierungsgrades ist stark vom Algorithmus abhängig
  - bei Bubblesort Hauptgewinn bei Q1
  - bei Quicksort Gewinn bei Q1 und Q3
- Java Interpreter immer deutlich langsamer als C (selbst unoptimiert)
- Verbesserung durch Java Just-in-Time Compiler stark vom Algorithmus abhängig
  - bei Bubblesort erreicht Java mit JIT fast die Performance von optimiertem C
  - bei Quicksort ist Java auch mit JIT Compiler erheblich schlechter als unoptimiertes C
- Der Speedup von Quicksort zu Bubblesort ist immer vorhanden und erheblich
- Der Speedup von Quicksort zu Bubblesort hängt stark vom Compiler ab und kann sich bis zu einem Faktor 6 unterscheiden.



Vorlesungsinhalt

Kapitel 1: Grundlegende Ideen, Technologien, Komponenten

Kapitel 2: Befehle: Die Sprache des Rechners

…

2.5 Kontrollstrukturen

2.6 MIPS Assembler und MARS Simulator

2.7 Weitere MIPS-Befehle

2.8 Compiler, Assembler, Linker, Loader

2.9 Andere Befehlssätze

2.9.1 ARM ISA

2.9.2 Intel x86 ISA

2.10 Zusammenfassung

ISA Klassifikation

- CISC und RISC
  - CISC: viele komplexe Instruktionen
  - RISC: wenige regelmäßige Instruktionen
- Zwei- und Drei-Registermaschinen
  - Zwei-Registermaschine: ein Operandenregister ist auch Zielregister
  - Drei-Registermaschine: zwei Operanden- und ein zusätzlichen Zielregister
- Anzahl Register zwischen 16 und 256
  - Trend geht zur leicht größerer Zahl von Registern
  - moderne Compiler sind in der Lage viele Register auszunutzen
- Speicherzugriffe
  - nur über explizite Befehle wie in MIPS
  - Speicheradresse als Operanden oder Zielregister
- Branches
  - über explizite Registervergleiche
  - über Flags und Conditions als Ergebnis arithmetischer Instruktionen
- unterstützte Addressierungsarten

Vorlesungsinhalt

Kapitel 1: Grundlegende Ideen, Technologien, Komponenten

Kapitel 2: Befehle: Die Sprache des Rechners

…

2.5 Kontrollstrukturen

2.6 MIPS Assembler und MARS Simulator

2.7 Weitere MIPS-Befehle

2.8 Compiler, Assembler, Linker, Loader

2.9 Andere Befehlssätze

2.9.1 ARM ISA

2.9.2 Intel x86 ISA

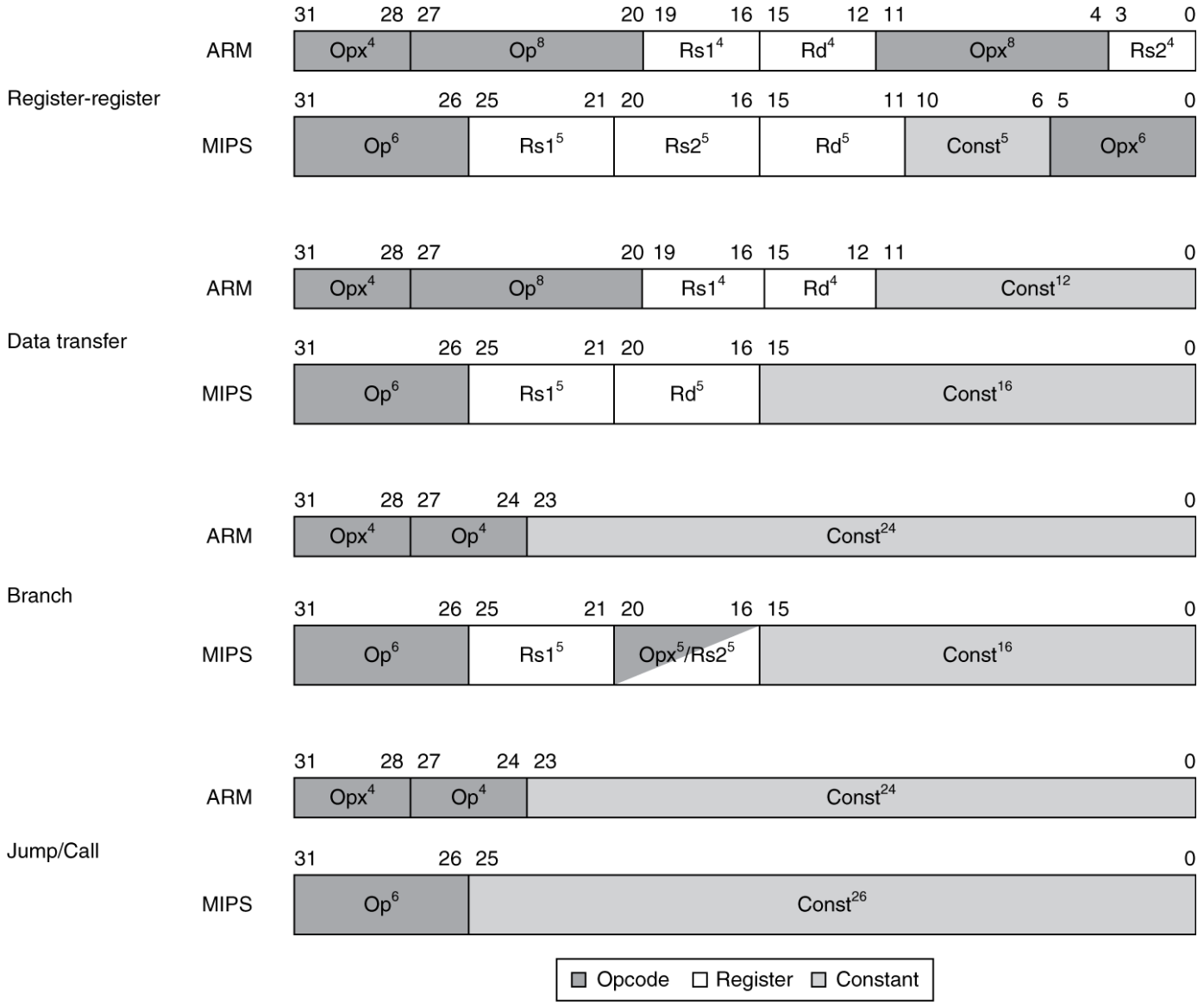
2.10 Zusammenfassung

Gemeinsamkeiten von ARM & MIPS

- ARM: populärste Befehlssatzarchitektur für eingebettete Systeme
- Befehlssatz sehr ähnlich zu MIPS Befehlssatz

	ARM	MIPS
Datum der Veröffentlichung	1985	1985
Instruktionsgröße	32 Bits	32 Bits
Adressraum	32 Bit (flach)	32 Bit (flach)
Datenausrichtung	ausgerichtet	ausgerichtet
Datenadressierungsmodi	9	3
Register	15 x 32 Bit	31 x 32 Bit
Ein-/Ausgabe	über den Speicher	über den Speicher

Übersicht der ARM Instruktionsformate



Vergleichen und Springen in ARM

MIPS: Ergebnis von Vergleich in Register, bedingte Sprünge abhängig von Registerinhalt

ARM: Ergebnis von Vergleich in Condition Codes, bedingte Sprünge abhängig von Condition Code

- Realisiert bedingte Verzweigungen über Condition Codes
- Condition Codes sind Flags in der ALU, die nach logischen und arithmetischen Operationen gesetzt werden
  - negative (N), zero (Z), carry (C), overflow (O)
- Vergleiche setzen NUR diese Condition Codes
  - das Ergebnis des Vergleichs wird nicht in Register geschrieben
  - Beispiel: CMP r2, r9
    - berechnet r2-r9 und setzt Flags (N,Z,C,O)
    - N und Z zur Unterscheidung von kleiner, gleich und größer
- Die Ausführung jeder Instruktion kann von den Condition Codes abhängig gemacht werden
  - Top 4 Bits der Instruktion (OPX) legen fest, ob Instruktion ausgeführt wird
  - dadurch können Sprünge vermieden werden
    - z.B. IF-Anweisung mit nur einer Instruktion im THEN Block

Vorlesungsinhalt

Kapitel 1: Grundlegende Ideen, Technologien, Komponenten

Kapitel 2: Befehle: Die Sprache des Rechners

…

2.5 Kontrollstrukturen

2.6 MIPS Assembler und MARS Simulator

2.7 Weitere MIPS-Befehle

2.8 Compiler, Assembler, Linker, Loader

2.9 Andere Befehlssätze

2.9.1 ARM ISA

2.9.2 Intel x86 ISA

2.10 Zusammenfassung

Intel x86 ISA

Intel x86 Architektur geprägt von langer Entwicklung und Komptabilität zu Vorgängerarchitekturen

- Während MIPS und auch PowerPC bei ihrer Einführung neue Architekturen waren, hat INTEL bei der Prozessorentwicklung über 20 Jahre lang auf Kompatibilität geachtet.
  - Einsatz von PowerPC unter anderem bei Spielekonsolen

- Auf diese Weise mussten die Leistungsparameter neuer Architekturkonzepte immer unter Berücksichtigung der Möglichkeiten der alten Prozessoren eingeführt werden.

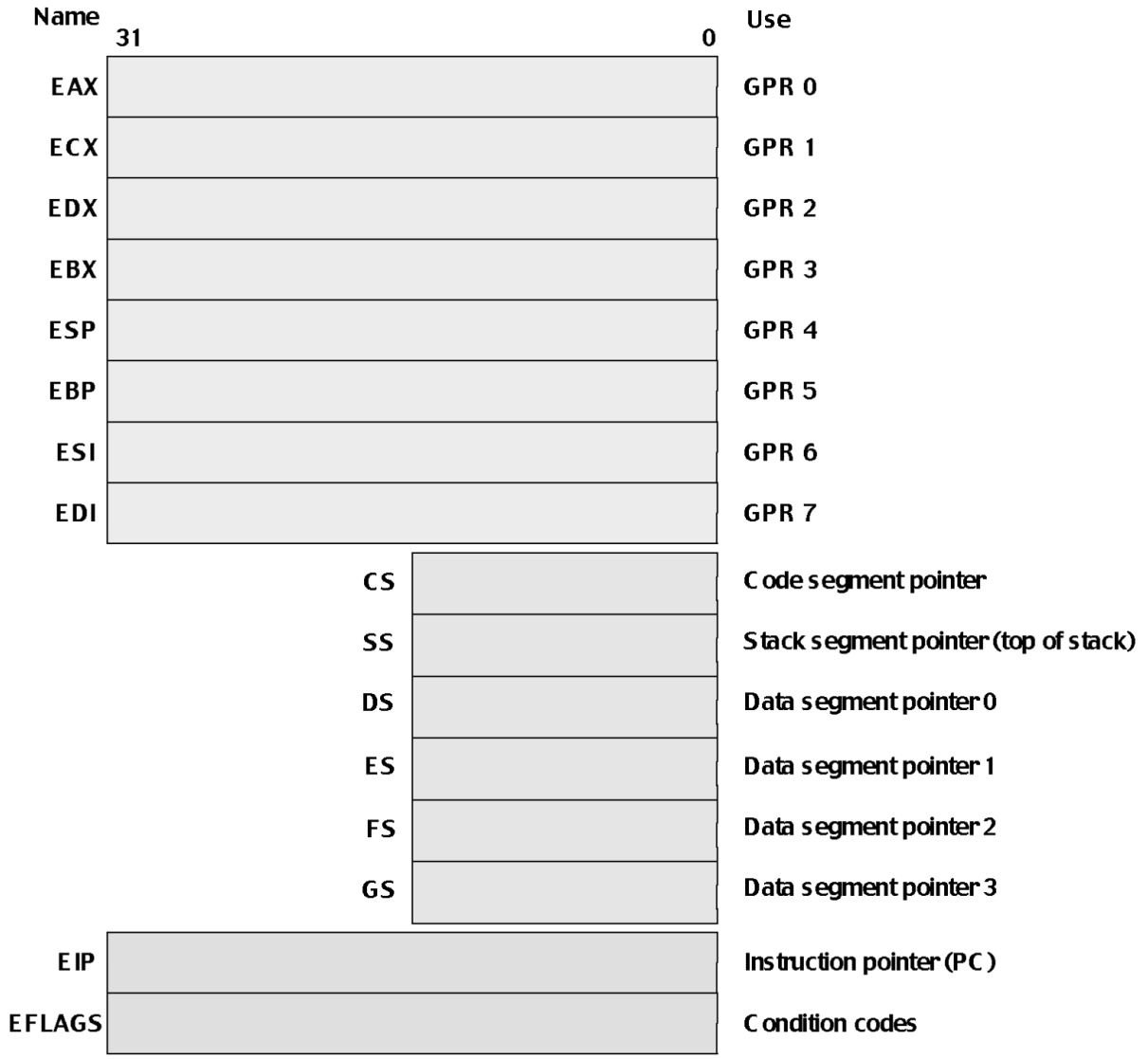
- Das Ergebnis sind leistungsfähige Prozessoren mit sehr komplexen und sehr irregulären Instruktionssatzarchitekturen.

Entwicklung der x86 ISA

- 1978: 8086
  - Erster 16 Bit-Prozessor als kompatible Erweiterung des erfolgreichen 8080. Register mit spezifischen Aufgaben.
- 1982: 80286
  - Erweiterter Adressraum (24 bit). Zusätzliche Instruktionen zur Abrundung des Befehlssatzes, speziell für Protection.
- 1985: 80386
  - Sowohl Adressraum, als auch Register auf 32 bit erweitert. Zusätzliche Operationen schaffen fast eine General-purpose-Register-Maschine.
- 1989: 80486; 1993: Pentium; 1995 Pentium Pro
  - Erhöhte Performanz durch interne Maßnahmen. Nur 4 neue Instruktionen.
- 1997: Pentium MMX
  - 57 zusätzliche Instruktionen für Multimedia-Anwendungen.

x86 Register (ab 386)

- 8 der ursprünglichen 16 Bit-Register wurden auf 32 Bit erweitert und dienen als „General-Purpose-Register“
- Für einzelne Instruktionen dienen einzelne Register speziellen Zwecken



x86 Instruktionen

- 2-Adressbefehle
  - Ein Register dient sowohl als Quell- als auch als Zielregister
  - Operationen spezifizieren also höchstens zwei Register/Speicheradressen

    ➔2-Adress-Maschine

Source/dest operand	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

- Alle Operationen mit Speicherzugriff
  - Keine „Load/Store"-Architektur.
  - Jede Instruktion kann Operanden aus dem Speicher nutzen

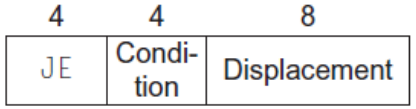
x86 Instruktionen

- x86 enthält mehr Spezialfunktionen als MIPS, z.B. pop und push für den Stack. Bedingte Sprünge sind wie bei ARM über Condition Code realisiert.

Instruction	Function
je name	if equal(condition code) {EIP=name}; EIP-128 <= name < EIP+128
jmp name	EIP=name
call name	SP=SP-4; M[SP]=EIP+5; EIP=name;
movw EBX,[EDI+45]	EBX=M[EDI+45]
push ESI	SP=SP-4; M[SP]=ESI
pop EDI	EDI=M[SP]; SP=SP+4
add EAX,#6765	EAX= EAX+6765
test EDX,#42	Set condition code (flags) with EDX and 42
movsl	M[EDI]=M[ESI]; EDI=EDI+4; ESI=ESI+4

x86 Instruktionsformate

a. JE EIP + displacement

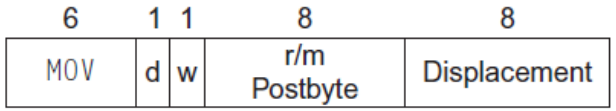


Sprungweite

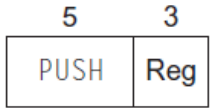
b. CALL

Bedingte Sprünge
über Condition Code

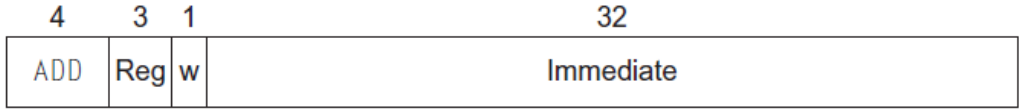
c. MOV EBX, [EDI + 45]



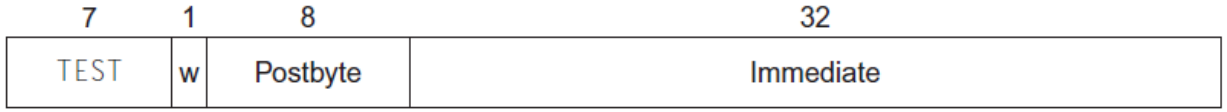
d. PUSH ESI



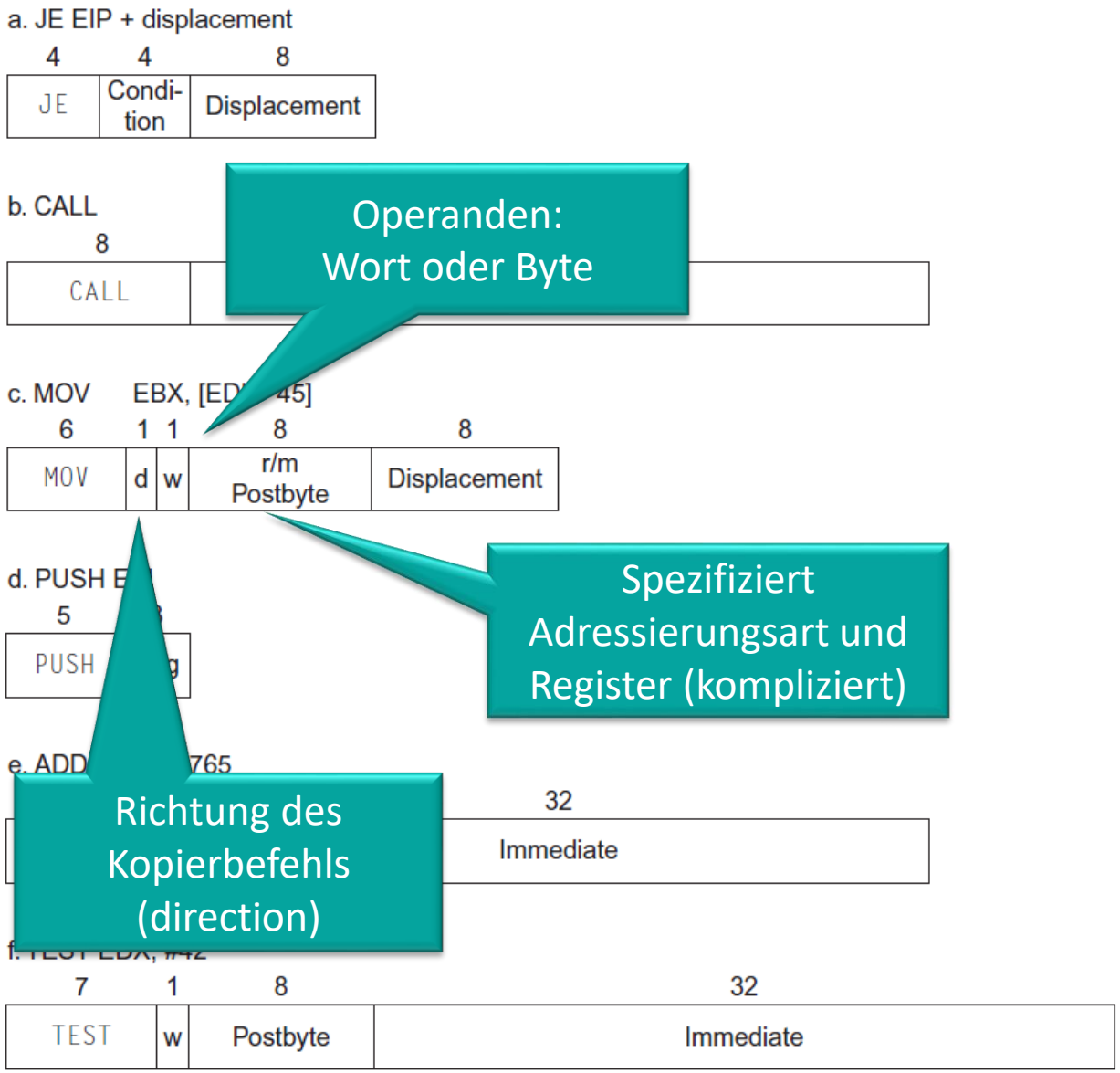
e. ADD EAX, #6765



f. TEST EDX, #42



x86 Instruktionsformate



Codierung in Maschinensprache-instruktionen variabler Länge

- Präfix-Bytes präzisieren die Operation
  - Länge der Operanden
  - Anzahl Wiederholungen
  - Lock
  - etc.
- Postfix-Bytes spezifizieren die Adressierungsart

Intel-Instruktionen variieren zwischen 1 und 17 Byte Länge bei äußerst verschiedenartigen Aufteilungen. Dies mach die HW-Implementierung ausgesprochen aufwendig.
Hardware übersetzt komplexe Befehle in einfache Micro-Instruktionen

Vorlesungsinhalt

Kapitel 1: Grundlegende Ideen, Technologien, Komponenten

Kapitel 2: Befehle: Die Sprache des Rechners

…

2.5 Kontrollstrukturen

2.6 MIPS Assembler und MARS Simulator

2.7 Weitere MIPS-Befehle

2.8 Compiler, Assembler, Linker, Loader

2.9 Andere Befehlssätze

2.9.1 ARM ISA

2.9.2 Intel x86 ISA

2.10 Zusammenfassung

Aussagen

Leistungsfähigere Befehle bedeuten höhere Leistung

- weniger Instruktionen benötigt
- Aber:
  - komplexe Instruktionen sind schwieriger in Hardware umzusetzen
    - langwierige Befehle können die Taktrate verringern und so auch die Ausführungszeit von einfachen Befehlen erhöhen
  - Compiler können auf Basis einfacher Instruktionen sehr effizienten Code generieren

Programmieren in Assemblersprache erzielt die größte Leistung

- Compiler sind heute so gut, dass sie besseren Code generieren als Menschen
- Grundregel des Programmierens: mehr Zeilen Code, mehr Fehler
  - Assembler-Code ist länger als Code in Hochsprache
- Sehr seltene Ausnahmen: Optimierung spezifischer Teile eines Programms, strikte Echtzeitanforderungen, kein guter Compiler vorhanden

Zusammenfassung

Soft-/Hardwareebenen zur Programmausführung

- Compiler, Assembler, Hardware

MIPS: ein typischer RISC Befehlssatz

- vergleiche die Einfachheit gegenüber x86 ISA

ISA Design-Prinzipien

- Simplicity favors regularity (Einfachheit begünstigt Regelmäßigkeit)
  - MIPS Befehlssatz sehr regelmäßig (32 Bit Instruktion, wenige Formate)
- Smaller is faster (kleiner ist schneller)
  - nur 32 Register, macht Instruktionen schnell (kurz Wege)
- Make the common case fast (optimiere den häufigen Fall)
  - 16 Bit Sprungweite bei bedingten Sprüngen
- Good design demands compromises (ein guten Entwurf erfordert Kompromisse)
  - Kompromiss zwischen langen Adressen/Konstanten und 32 Bit Instruktionen

Welche ISA ist am Besten?

- keine Aussage, hängt von mehrere Faktoren ab
  - Codedichte (insbesondere bei knappen Speicheressourcen)
  - Programmperformance

- ISA alleine kann nicht per se beurteilt werden sondern hängt von Qualität der zur Verfügung stehenden Hardware, Compiler und Debugger ab
  - Qualität der ISA beeinflusst Komplexität von Hardware-Design und Compilerbau

- Erfolgreiche ISAs
  - PC-Bereich: x86 (CISC), Power Architecture, SPARC, Itanium
  - Embedded-Bereich: ARM (RISC), Atmel AVR, TI MSP430, MIPS, etc.

Bisherige Instruktionen und Pseudo-Instruktionen

MIPS instructions	Name	Format	Pseudo MIPS	Name	Format
add	add	R	move	move	R
subtract	sub	R	multiply	mult	R
add immediate	addi	I	multiply immediate	multi	I
load word	lw	I	load immediate	li	I
store word	sw	I	branch less than	blt	I
load half	lh	I	branch less than or equal	ble	I
load half unsigned	lhu	I			
store half	sh	I	branch greater than	bgt	I
load byte	lb	I	branch greater than or equal	bge	I
load byte unsigned	lbu	I			
store byte	sb	I			
load linked	ll	I			
store conditional	sc	I			
load upper immediate	lui	I			
and	and	R			
or	or	R			
nor	nor	R			
and immediate	andi	I			
or immediate	ori	I			
shift left logical	sll	R			
shift right logical	srl	R			
branch on equal	beq	I			
branch on not equal	bne	I			
set less than	slt	R			
set less than immediate	slti	I			
set less than immediate unsigned	sltiu	I			
jump	j	J			
jump register	jr	R			
jump and link	jal	J			

- SPEC CPU 2006 (https://www.spec.org/cpu2006/)
- Integer: 70% durch Speicherzugriffe und bedingte Sprünge
  - wenig Arithmetik (16%)
- Floating-Point: 85% durch Speicherzugriff und Arithmetik
  - wenige bedingte Sprünge (8%)

Instruction class	MIPS examples	HLL correspondence	Frequency	
			Integer	Ft. pt.
Arithmetic	add, sub, addi	Operations in assignment statement s	16%	48%
Data transfer	lw, sw, lb, lbu, lh, lhu, sb, lui	References to data structures, such as arrays	35%	36%
Logical	and, or, nor, andi, ori, sll, srl	Operations in assignment statement s	12%	4%
Conditional branch	beq, bne, slt, slti, sltiu	*If* statements and loops	34%	8%
Jump	j, jr, jal	Procedure calls, returns, and *case/switch* statements	2%	0%