

# Rechnerorganisation – Kapitel 6



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Prof. Sarah Harris, Ph.D.  
Fachgebiet Eingebettete Systeme und ihre Anwendungen (ESA)  
Fachbereich Informatik

SS 16



# Professorin Sarah Harris



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

- Technische Universität Darmstadt  
2015 – 2016
- **Fachgebiet:** Eingebettete Systeme, Rechnerarchitekturen
- **Kontakt Informationen:**  
Sprechstunden: Mittwochs, 13:30 Uhr – 14:30 Uhr  
Email: [harris@esa.informatik.tu-darmstadt.de](mailto:harris@esa.informatik.tu-darmstadt.de)  
Büro: S2/02 (Piloty Gebäude), Raum E102

# Wo ich herkomme

- Stanford University  
Ph.D. (≈Doktor)  
Electrical and Computer Engineering  
(2005)
- Harvey Mudd College  
Assistant/Associate Professorin  
(2004-2014)
- University of Nevada, Las Vegas  
Associate Professorin  
(2014 - )

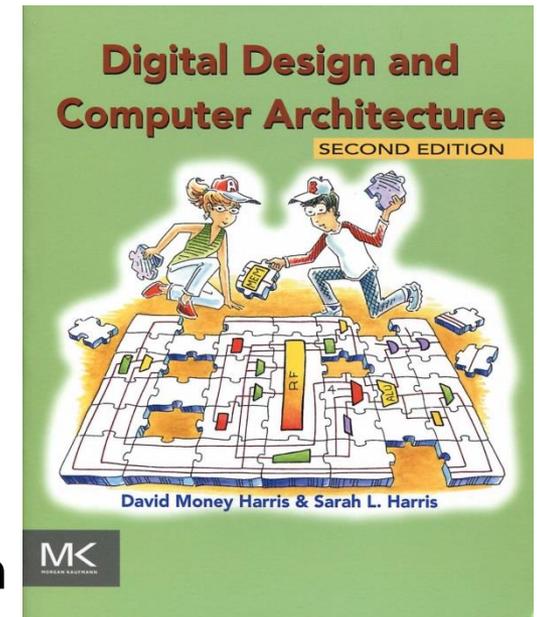


# Lehr- und Anschauungsmaterial

- Aus dem Lehrbuch

## *Digital Design and Computer Architecture, Zweite Auflage*

- Diese Folien nach englischen Originalvorlagen erstellt (Originale sind © 2007 Elsevier)
- Buch wird an Studierende **subventioniert** abgegeben
  - Organisiert durch Fachschaft Informatik
- Mehr **Hintergrundmaterial** auf Web-Seite zu Buch
- Dieses Semester: Kapitel 6 bis 8



# Überblick

Themen zu diesem Semester:

- **Architektur:** Programmierersicht auf Computer (Kapitel 6)
- **Mikroarchitektur:** Hardware-Implementierung der Architektur (Kapitel 7)
- **Speichersysteme:** Caches, Virtueller Speicher, I/O (Kapitel 8)

# Empfehlungen

- Rechtzeitig die Vorlesungen anschauen und Übungen mitmachen
  - die aktuelle Vorlesung baut auf die vorgehenden Vorlesungen auf
- Wenn Sie **Fragen** haben, stellen Sie die mal während der Vorlesung, nach der Vorlesung, während der Sprechstunden, in den Übungsgruppen

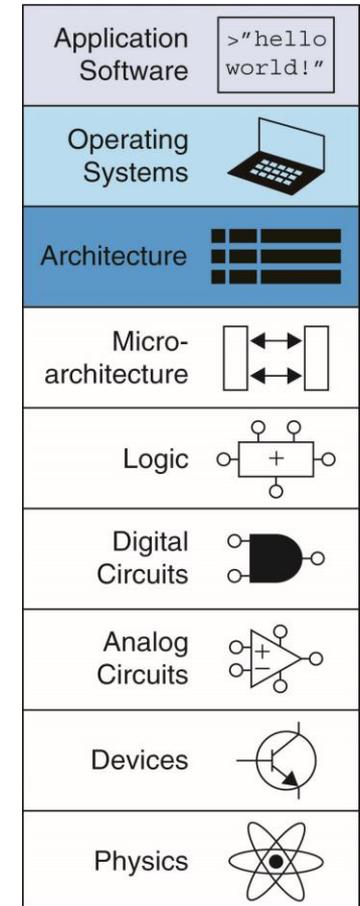
Ich werde Ihnen während der Vorlesung auch Fragen stellen

# Organisatorisches

- **Übungen:** Organisatorisches von dem Assistent jetzt erklärt...
- **Webseite:** [www.esa.informatik.tu-darmstadt.de](http://www.esa.informatik.tu-darmstadt.de)
  - unter Lehre → SS 2016 → Rechnerorganisation
  - Vorlesungsfolien, Aufzeichnungen der Vorlesungen, kommentierte Vorlesungsfolien
- **Prüfung:** 18. Juli 2016

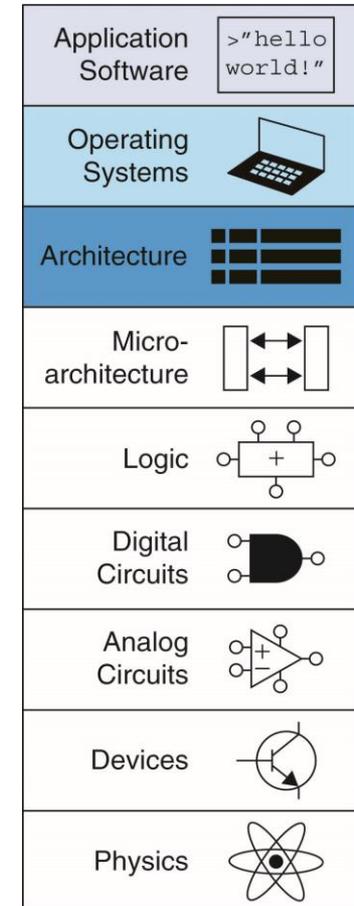
# Kapitel 6: Themen

- **Einleitung**
- **Assembler-Sprache**
- **Maschinensprache**
- **Programmierung**
- **Adressierungsmodi**
- **Compilieren, Assemblieren und Linken**
- **Dies und Das**



# Einleitung

- Nun Sprung auf höhere Abstraktionsebene
  - Erstmal ...
- **Architektur:** Programmierersicht auf Computer
  - Definiert durch Instruktionen (Operationen) und Operanden
- **Mikroarchitektur:** Hardware-Implementierung der Architektur
  - Kommt im Detail in Kapitel 7



# Assemblersprache



## Programmieren in Sprache des Computers

- **Instruktionen / Befehle:** Einzelne Worte
- **Befehlssatz:** Gesamtes Vokabular

Befehle geben Art der Operation und ihre Operanden an

## Zwei Darstellungen

- **Assemblersprache:** für Menschen lesbare Schreibweise für Instruktionen
- **Maschinsprache:** maschinenlesbares Format (1'en und 0'en)

# MIPS Architektur

- Von John Hennessy und Kollegen in Stanford in den 1980ern entwickelt
- In vielen Computern und eingebetteten Systemen verwendet:
  - Silicon Graphics, Nintendo, Sony, Cisco, Cavium, NetLogic, ...
- Gut zur Darstellung von allgemeinen Konzepten
  - Vieles auch auf andere Architekturen übertragbar
- Imagination Technologies hat MIPS im Feb 2013 gekauft

# John Hennessy



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

- Präsident der Universität Stanford
- Professor in Elektrotechnik und Informationstechnik in Stanford seit 1977
- Miterfinder des Reduced Instruction Set Computers (RISC)
- Entwickelte MIPS-Architektur in Stanford in 1984 und war Mitgründer von MIPS Computer Systems
- Bis 2004: Über 300 Millionen MIPS Prozessoren verkauft



# Entwurfsprinzipien für Architekturen



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

John Hennessy (Stanford) und David Patterson (Berkeley):

1. Regularität vereinfacht Entwurf
2. Mach den häufigsten Fall schnell
3. Kleiner ist schneller
4. Ein guter Entwurf verlangt gute Kompromisse

# Befehle: Addition

## Hochsprache

```
a = b + c;
```

## MIPS Assemblersprache

```
add a, b, c
```

- **add:** Befehlsname (*mnemonic*) gibt die Art der auszuführenden Operation an
- **b, c:** Quelloperanden auf denen die Operation ausgeführt wird
- **a:** Zieloperand in den das Ergebnis eingetragen wird

# Befehl: Subtraktion

Subtraktion ist ähnlich zur Addition. Nur der Befehlsname ändert sich.

## Hochsprache

$a = b - c;$

## MIPS Assemblersprache

sub a, b, c

- **sub:** mnemonic
- **b, c:** Quelloperanden
- **a:** Zieloperand

## Regularität vereinfacht Entwurf

- Konsistentes Befehlsformat
- Gleiche Anzahl von Operanden
  - Zwei Quellen, ein Ziel
- Leichter zu kodieren und in Hardware zu bearbeiten

# Befehle: Komplexere Abläufe

Komplexere Abläufe werden durch Folgen von einfachen Befehlen realisiert

## Hochsprache

```
a = b + c - d;
```

```
// Kommentare bis Zeilenende  
/* mehrzeiliger  
Kommentar */
```

## MIPS Assemblersprache

```
add t, b, c # t=b+c  
sub a, t, d # a=t-d
```

```
# Kommentare bis Zeilende
```

## Mach den häufigen Fall schnell

- MIPS enthält nur **einfache, häufig verwendete** Befehle
- Hardware zur Dekodierung und Ausführung der Befehle kann einfach, klein und schnell sein
- Komplexe Anweisungen (die nur seltener auftreten) können durch Folgen von einfachen Befehlen realisiert werden
- MIPS ist ein Computer mit reduziertem Befehlssatz (***reduced instruction set computer, RISC***)
- Alternative: Computer mit komplexem Befehlssatz (***complex instruction set computer, CISC***)
  - Beispiel: Intel IA-32 / x86 (weit verbreitet in PCs)

# Operanden

Ein Prozessor hat physikalische Speicherorte für die Operanden von Befehlen

## Mögliche Speicherorte

- Register
- Speicher
- Konstante Werte (*immediates*) - stehen häufig direkt im Befehl

# Operanden: Register

- Speicher ist langsam
- Viele Architekturen haben deshalb kleine Anzahl von schnellen Registern
- MIPS hat 32 Register, jedes **32 Bit breit**  
Wird deshalb auch “**32b Architektur**” genannt
- Es gibt auch eine 64b-Version von MIPS  
... wird hier aber nicht weiter behandelt

# Entwurfsprinzip 3

## Kleiner ist schneller

- MIPS stellt nur eine kleine Anzahl von Registern bereit
- Kann in schnellerer Hardware realisiert werden als größeres Registerfeld

# MIPS Registerfeld

Name	Registernummer	Verwendungszweck
\$0	0	Konstante Null
\$at	1	Temporäre Variable für Assembler
\$v0-\$v1	2-3	Rückgabe von Werten aus Prozedur
\$a0-\$a3	4-7	Aufrufparameter in Prozedur
\$t0-\$t7	8-15	Temporäre Variablen
\$s0-\$s7	16-23	Gesicherte Variablen
\$t8-\$t9	24-25	Mehr temporäre Variablen
\$k0-\$k1	26-27	Temporäre Variablen für Betriebssystem
\$gp	28	Zeiger auf globale Variablen im Speicher
\$sp	29	Stapelzeiger im Speicher
\$fp	30	Zeiger auf aktuellen Aufruf-Frame im Speicher
\$ra	31	Rücksprungadresse aus Prozedur

# Operanden: Register

## Register:

- Kennzeichnend gemacht durch dem Namen vorangestelltes Dollar-Zeichen
- **Beispiel:** Register 0 wird geschrieben als “\$0”  
Gelesen als: “Register Null” oder “Dollar Null”.

## Bestimmte Register für bestimmte Verwendungszwecke, zum Beispiel:

- \$0 enthält immer den konstanten Wert 0.
- Gesicherte Register (\$s0 - \$s7) für das Speichern von Variablen
- Temporäre Register (\$t0 - \$t9) für das Speichern von Zwischenergebnissen während einer komplizierteren Rechnung

# Operanden: Register

## Zunächst benutzen wir nur

- Temporäre Register ( $\$t0$  -  $\$t9$ )
- Gesicherte Register ( $\$s0$  -  $\$s7$ )

Später mehr ...

# Befehle mit Registerangaben

## Rückblick auf add-Befehl

### Hochsprache

```
a = b + c
```

### MIPS Assemblersprache

```
# $s0 = a, $s1 = b, $s2 = c  
add $s0, $s1, $s2
```

# Operanden: Speicher

- Daten passen nicht alle in 32 Register
- Lege Daten im Hauptspeicher ab
- Hauptspeicher ist **groß** (GB...TB) und kann viele Daten halten ... ist aber auch **langsam**
- Speichere **häufig verwendete Daten in Registern**
- Kombiniere Register und Speicher zum Halten von Daten
  - **Ziel:** Greife schnell auf große Mengen von Daten zu

# Wort- und Byte-Adressierung von Daten im Speicher

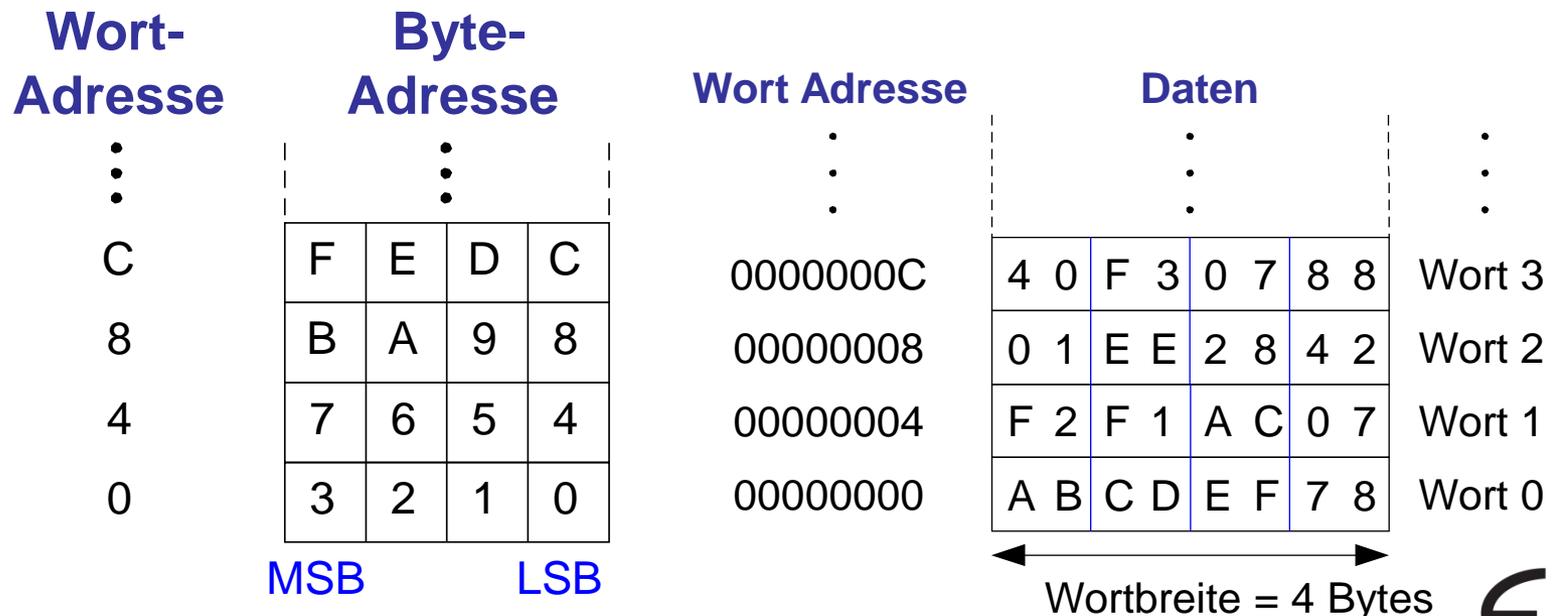
- MIPS ist **byte-adressiert**, das heißt...  
**jedes Byte hat eine eindeutige Adresse**
- 32-bit Wort = 4 Bytes...  
Adressen von Worten sind also Vielfache von 4

Wort- Adresse	Byte- Adresse			
⋮	⋮			
C	F	E	D	C
8	B	A	9	8
4	7	6	5	4
0	3	2	1	0
	MSB			LSB

# Wort- und Byte-Adressierung von Daten im Speicher

- MIPS ist **byte-adressiert**, das heißt...  
**jedes Byte hat eine eindeutige Adresse**
- 32-bit Wort = 4 Bytes...

Adressen von Worten sind also Vielfache von 4



# Lesen aus byte-adressiertem Speicher



- Lesen geschieht durch Ladebefehle (*load*)
- Befehlsname: *load word* (`lw`)
- **Format:** `lw $t0, 8($s2)`

# Lesen aus byte-adressiertem Speicher



- Lesen geschieht durch Ladebefehle (*load*)
- Befehlsname: *load word* ( $lw$ )
- **Format:**  $lw \ \$t0, \ 8 (\$s2)$

Lese ein Datenwort von der Speicheradresse ( $\$s2 + 8$ ) nach  $\$t0$

**Adressarithmetik:** Adressen werden relativ zu einem Register angegeben

- Basisadresse ( $\$s2$ ) plus Distanz (*offset*) (8)
- Adresse = ( $\$s2 + 8$ )

**Ergebnis:**  $\$t0$  enthält das Datawort von Speicheradresse ( $\$s2 + 8$ )

**Jedes Register darf als Basisadresse verwendet werden**

# Lesen aus byte-adressiertem Speicher

**Beispiel:** Lese Datenwort 3 (Speicheradresse 0xC)  
nach  $\$s7$

```
lw $s7, 0xC($0)
```

## Adressarithmetik:

- Basisadresse ( $\$0$ ) plus Distanz (*offset*) (0xC)
- Adresse = ( $\$0 + 12$ ) = 12

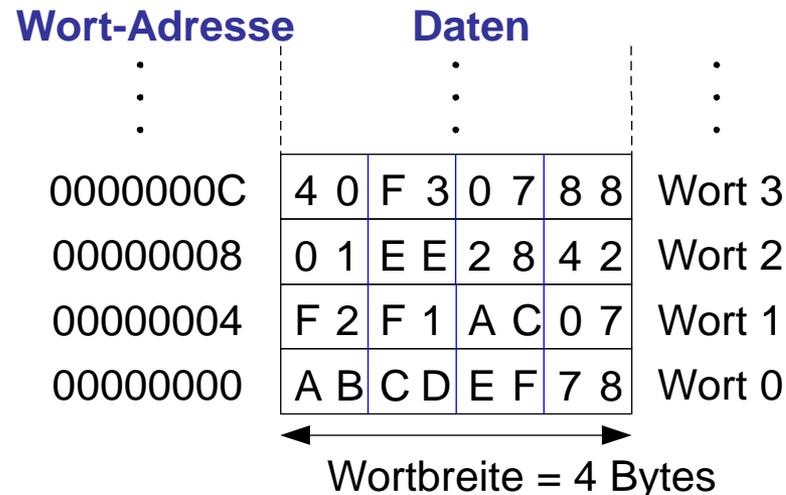
Nach Abarbeiten des Befehls hat  $\$s7$  den Wert 0x40F30788

Wort-Adresse	Daten	
⋮	⋮	⋮
0000000C	4 0 F 3 0 7 8 8	Wort 3
00000008	0 1 E E 2 8 4 2	Wort 2
00000004	F 2 F 1 A C 0 7	Wort 1
00000000	A B C D E F 7 8	Wort 0

← Wortbreite = 4 Bytes →

# Lesen aus byte-adressiertem Speicher

**Beispiel:** Lese Datenwort 1 nach  $\$s5$



# Schreiben in byte-adressiertem Speicher

- Schreiben geschieht durch Speicherbefehle (*store*)
- Befehlsname: *store word* ( $sw$ )
- **Format:**  $sw \ \$t4, \ 0x1c (\$0)$

**Beispiel:** Schreibe (speichere) den Wert aus  $\$t4$  in Speicherwort 7 (bzw. Speicheradresse  $7 \times 4 = 28 = 0x1C$ )

## Adressarithmetik:

- Basisadresse ( $\$0$ ) plus Distanz (*offset*) ( $0x1c$ )
- Adresse = ( $\$0 + 28$ )

**Ergebnis:** Nach Abarbeiten des Befehls enthält Speicheradresse ( $\$0 + 28$ ) das Datawort von  $\$t4$

# Schreiben in byte-adressiertem Speicher



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

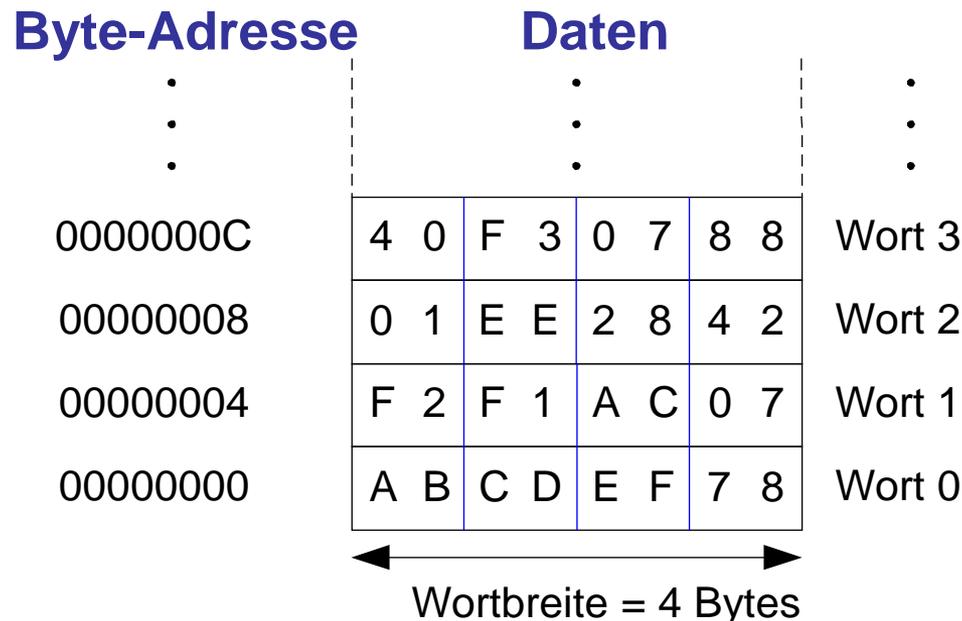
**Beispiel:** Schreiben Sie den Wert 79 in Speicherwort 15

# Byte-adressierbarer Speicher

Speicherbefehle können auf Worten oder Bytes arbeiten

**Worte:**  $lw / sw$

**Bytes:**  $lb / sb$



# Speicherorganisation: Big-Endian und Little-Endian

Schemata für Nummerierung von Bytes in einem Wort

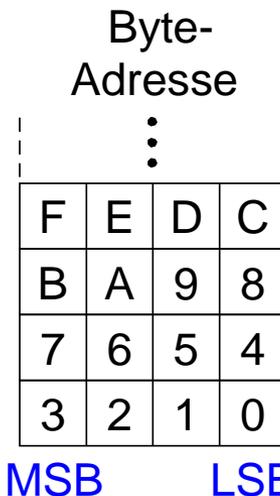
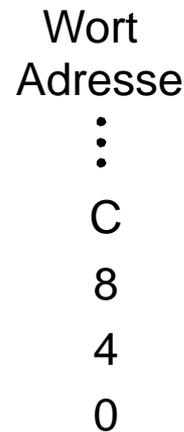
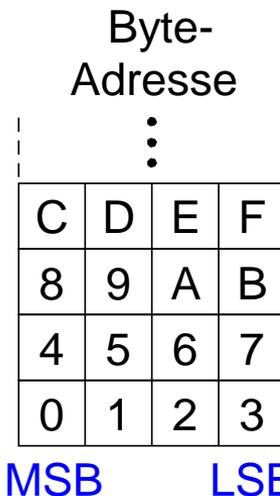
**Wort-Adresse** ist bei beiden **gleich**

**Little-endian:** Bytes werden vom niederstwertigen Ende an gezählt

**Big-endian:** Bytes werden vom höchstwertigen Ende an gezählt

Big-Endian

Little-Endian



# Speicherorganisation: Big-Endian und Little-Endian



Aus Jonathan Swift's *Gullivers Reisen*

- **Little-Endians** schlagen Eier an der schmalen Seite auf
- **Big-Endians** schlagen Eier an der breiten Seite auf

Welche Organisation benutzt wird ist im Prinzip egal ...

... außer wenn unterschiedliche Systeme Daten austauschen müssen

## Big-Endian

Byte-Adresse			
⋮			
C	D	E	F
8	9	A	B
4	5	6	7
0	1	2	3
<b>MSB</b>			<b>LSB</b>

## Little-Endian

Byte-Adresse			
⋮			
F	E	D	C
B	A	9	8
7	6	5	4
3	2	1	0
<b>MSB</b>			<b>LSB</b>

# Beispiel: Big-Endian und Little-Endian



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

**Annahme:**  $\$t0$  enthält den Wert  $0x23456789$

**Programm:**

```
sw $t0, 0($0)
```

```
lb $s0, 1($0)
```

**Fragen:** Welchen Wert hat  $\$s0$  nach Ausführung auf einem...

- ... Big-Endian Prozessor?
- ... Little-Endian Prozessor?

# Entwurfsprinzip 4

## Ein guter Entwurf verlangt gute Kompromisse

- Mehrere **Befehlsformate** erlauben Flexibilität ...
  - add, sub: verwenden drei Register als Operanden
  - lw, sw: verwendet zwei Register und eine Konstante als Operanden
- ... aber **Anzahl** von Befehlsformaten sollte klein sein
  - Entwurfsprinzip 1: Regularität vereinfacht Entwurf
  - Entwurfsprinzip 3: Kleiner ist schneller

# Operanden: Konstante Werte in Befehl (*immediates*)

- $l_w$  und  $s_w$  zeigen die Verwendung von **konstanten Werten** (*immediates*)
  - Direkt im Befehl untergebracht, deshalb auch **Direktwerte** genannt
  - Brauchen kein eigenes Register oder Speicherzugriff
- Befehl “add immediate” (`addi`) addiert Direktwert auf Register
- Direktwert ist 16b Zweierkomplementzahl

## Hochsprache

```
a = a + 4;  
b = a - 12;
```

## MIPS Assemblersprache

```
# $s0 = a, $s1 = b  
addi $s0, $s0, 4  
addi $s1, $s0, -12
```

# Maschinensprache



- Computer verstehen nur 0'en und 1'en
- **Maschinensprache:** Binärdarstellung von Befehlen
- 32b Befehle
  - **Regularität vereinfacht Entwurf:** Daten und Befehle sind beides 32b Worte

## Drei Befehlsformate

- **R-Typ:** Operanden sind nur Register
- **I-Typ:** Register und ein Direktwert
- **J-Typ:** für Programmsprünge (kommt noch)

# Befehlsformat R-Typ (Register Typ)

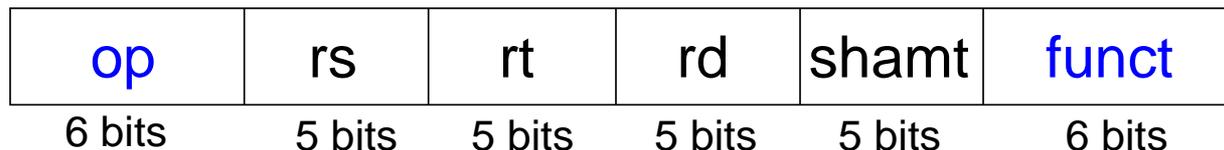
## 3 Registeroperanden

**rs:** Quellregister  
**rt:** Quell/Zielregister  
**rd:** Zielregister

## Andere Angaben in binärkodiertem Befehl:

**op:** *Operations-Code* oder *Opcode* (ist 0 für Befehle vom R-Typ)  
**funct:** Auswahl der genauen *Funktion*  
Opcode und Funktion zusammen bestimmen die auszuführende Operation  
**shamt:** Schiebeweite für Shift-Befehle, sonst 0

## R-Typ



# Beispiele für Befehle vom R-Typ

## Assemblersprache

```
add $s0, $s1, $s2
```

```
sub $t0, $t3, $t5
```

## Felder in Befehlswort

op	rs	rt	rd	shamt	funct
0	17	18	16	0	32
0	11	13	8	0	34

6 bits      5 bits      5 bits      5 bits      5 bits      6 bits

## Maschinsprache

op	rs	rt	rd	shamt	funct	
000000	10001	10010	10000	00000	100000	(0x02328020)
000000	01011	01101	01000	00000	100010	(0x016D4022)

6 bits      5 bits      5 bits      5 bits      5 bits      6 bits

**Beachte** andere Reihenfolge der Register in Assembler-Sprache:

```
add rd, rs, rt
```

# Befehlsformat I-Typ (Immediate Typ)

- 3 Operanden:
    - rs**: Quellregister
    - rt**: Zielregister
    - imm**: 16b Direktwert im Zweierkomplement
  - Andere Angaben:
    - op**: Opcode
- Regularität vereinfacht Entwurf: **Alle** Befehle haben einen Opcode  
Operation wird bei I-Typ **nur** durch Opcode bestimmt
- Keine Angabe über Funktion nötig (oder vorhanden!)

## I-Typ



# Beispiel für Befehle vom I-Typ

## Assemblersprache

```
addi $s0, $s1, 5
addi $t0, $s3, -12
lw   $t2, 32($0)
sw   $s1, 4($t1)
```

## Felder im Befehlswort

op	rs	rt	imm
8	17	16	5
8	19	8	-12
35	0	10	32
43	9	17	4

6 bits      5 bits      5 bits      16 bits

**Beachte** unterschiedliche Reihenfolge von Registern in Assembler- und Maschinensprache

```
addi rt, rs, imm
lw   rt, imm(rs)
sw   rt, imm(rs)
```

## Maschinensprache

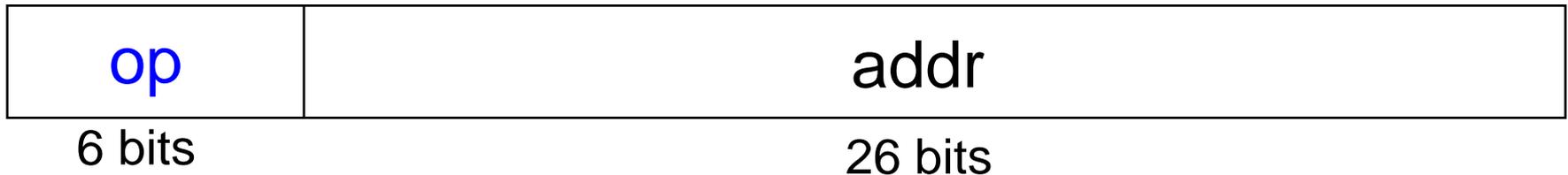
op	rs	rt	imm	
001000	10001	10000	0000 0000 0000 0101	(0x22300005)
001000	10011	01000	1111 1111 1111 0100	(0x2268FFF4)
100011	00000	01010	0000 0000 0010 0000	(0x8C0A0020)
101011	01001	10001	0000 0000 0000 0100	(0xAD310004)

6 bits      5 bits      5 bits      16 bits

# Befehlsformat J-Typ (Jump Typ)

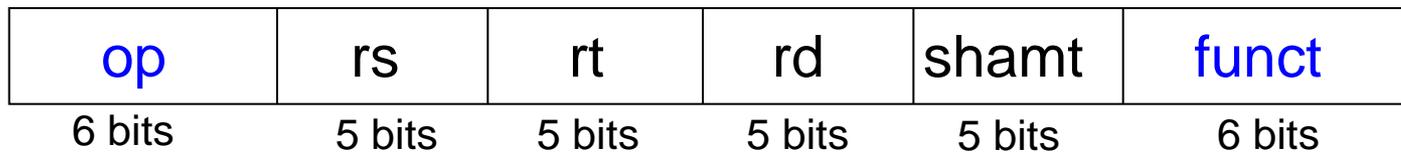
- 26b Adressoperand (`addr`)
- Verwendet für Sprungbefehle (`j`)

## J-Typ



# Übersicht über Befehlsformate

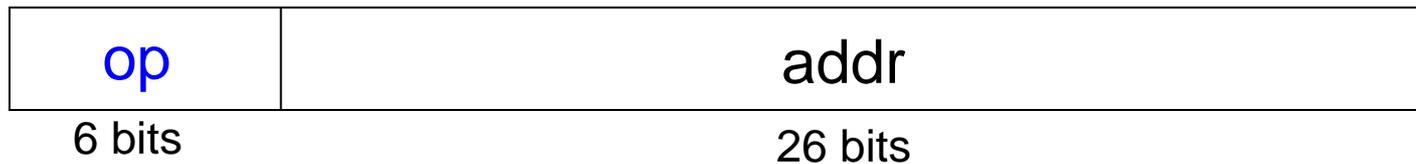
## R-Typ



## I-Typ



## J-Typ



# Flexibilität durch gespeicherte Programme



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

- 32b Befehle und Daten im **Speicher**
- Folgen von Befehlen bestimmen **Verhalten**
  - Einziger Unterschied zwischen zwei Anwendungen
- Ausführen von **unterschiedlichen** Programmen
  - Ohne Neuverdrahten oder Neuaufbau von Hardware
  - Nur neues Programm als **Maschinensprache** im Speicher ablegen

# Flexibilität durch gespeicherte Programme



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Die Hardware des Prozessors führt Programm **schrittweise** aus:

- **Holt neue Befehle** aus dem Speicher (*fetch*) in richtiger Reihenfolge
- **Führt die** im Befehl verlangte Operation aus

**Programmzähler** (*program counter, PC*)

- Zeigt Adresse des auszuführenden Befehls an

Bei MIPS: Programmausführung **beginnt** auf Adresse 0x00400000

# Im Speicher abgelegtes Programm

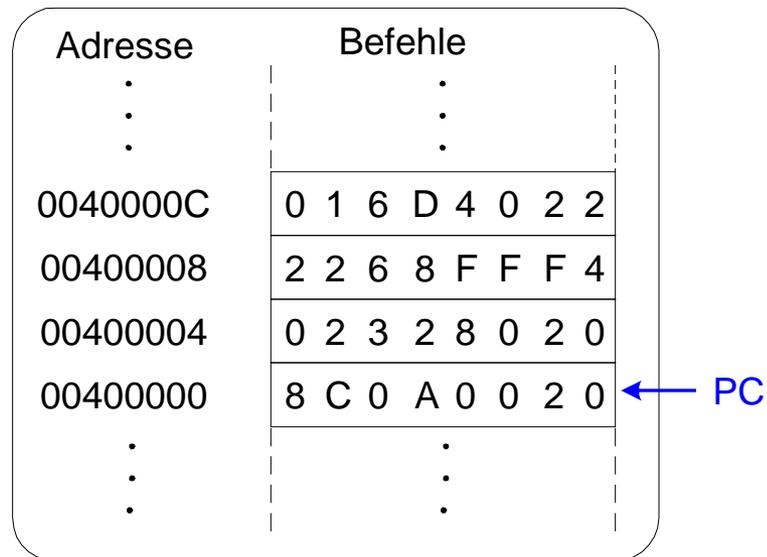
## Assemblersprache

```
lw    $t2, 32($0)
add   $s0, $s1, $s2
addi  $t0, $s3, -12
sub   $t0, $t3, $t5
```

## Maschinensprache

```
0x8C0A0020
0x02328020
0x2268FFF4
0x016D4022
```

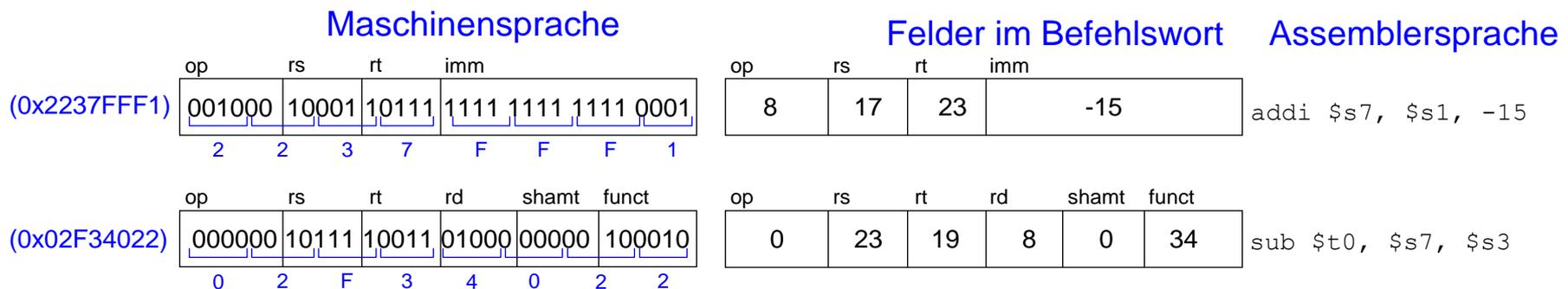
## Programm im Speicher



Hauptspeicher

# Maschinensprache verstehen

- Beginn mit **Entschlüsseln** des Opcodes
- Opcode bestimmt **Bedeutung** der anderen Bits
- Wenn Opcode **Null** ist
  - ... liegt ein Befehl im **R-Format** vor
  - Die Operation wird durch das Funktionsfeld bestimmt
- **Sonst**
  - Bestimmt Opcode alleine die Operation, siehe Anhang B im Buch



# Programmierung

## Hochsprachen:

- z.B. C, Java, Python, Scheme
- Auf einer **abstrakteren** Ebene programmieren

## Häufige Konstrukte in Hochsprachen:

- if/else-Anweisungen
- for-Schleifen
- while-Schleifen
- Feld (Array) zugriffe
- Prozeduraufrufe\

## Andere **nützliche** Anweisungen:

- Arithmetische/logische Ausdrücke
- Verzweigungen

# Ada Lovelace, 1815 - 1852



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

- Schrieb das erste Computerprogramm
- Sollte die Bernoulli-Zahlen auf der Analytischen Maschine von Charles Babbage berechnen
- Ein Kind des Dichters Lord Byron



# Logische Befehle

## and, or, xor, nor

- `and`: nützlich zum **Maskieren** von Bits  
**Beispiel:** Ausmaskieren aller Bits außer dem LSB:  
 $0xF234012F \text{ AND } 0x000000FF = 0x0000002F$
- `or`: Nützlich zum **Vereinigen** von Bitfeldern  
**Beispiel:** Vereinige  $0xF2340000$  mit  $0x000012BC$ :  
 $0xF2340000 \text{ OR } 0x000012BC = 0xF23412BC$
- `nor`: nützlich zur **Invertierung** von Bits:  
**Beispiel:**  $A \text{ NOR } \$0 = \text{NOT } A$

# Logische Befehle mit Konstanten



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

## `andi`, `ori`, `xori`

- 16-bit Direktwert wird erweitert mit führenden **Nullbits** (*nicht vorzeichenerweitert*)
- `nori` wird nicht benötigt

# Beispiele: Logische Befehle

## Quellregister

<b>\$s1</b>	1111	1111	1111	1111	0000	0000	0000	0000
<b>\$s2</b>	0100	0110	1010	0001	1111	0000	1011	0111

## Assemblersprache

and \$s3, \$s1, \$s2  
or \$s4, \$s1, \$s2  
xor \$s5, \$s1, \$s2  
nor \$s6, \$s1, \$s2

## Ergebnisse

<b>\$s3</b>								
<b>\$s4</b>								
<b>\$s5</b>								
<b>\$s6</b>								

# Beispiele: Logische Befehle

## Operanden

\$s1	0000	0000	0000	0000	0000	0000	1111	1111
imm	0000	0000	0000	0000	1111	1010	0011	0100

Null-erweitert

## Assemblersprache

```
andi $s2, $s1, 0xFA34 $s2
ori  $s3, $s1, 0xFA34 $s3
xori $s4, $s1, 0xFA34 $s4
```

## Ergebnisse

\$s2								
\$s3								
\$s4								

# Schiebebefehle

**sll:** shift left logical

- **Beispiel:** `sll $t0, $t1, 5`  
# `$t0 <= $t1 << 5`

**srl:** shift right logical

- **Beispiel :** `srl $t0, $t1, 5`  
# `$t0 <= $t1 >> 5`

**sra:** shift right arithmetic

- **Beispiel :** `sra $t0, $t1, 5`  
# `$t0 <= $t1 >>> 5`

## Schieben mit variabler Distanz:

**sllv:** shift left logical variable

- **Beispiel:** `sllv $t0, $t1, $t2`  
# `$t0 <= $t1 << $t2`

**srlv:** shift right logical variable

- **Beispiel:** `srlv $t0, $t1, $t2`  
# `$t0 <= $t1 >> $t2`

**srav:** shift right arithmetic variable

- **Beispiel:** `srav $t0, $t1, $t2`  
# `$t0 <= $t1 >>> $t2`

# Schiebebefehle

## Assemblersprache

## Felder in Instruktion

	op	rs	rt	rd	shamt	funct
<code>sll \$t0, \$s1, 2</code>	0	0	17	8	2	0
<code>srl \$s2, \$s1, 2</code>	0	0	17	18	2	2
<code>sra \$s3, \$s1, 2</code>	0	0	17	19	2	3
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

## Maschinsprache

op	rs	rt	rd	shamt	funct	
000000	00000	10001	01000	00010	000000	(0x00114080)
000000	00000	10001	10010	00010	000010	(0x00119082)
000000	00000	10001	10011	00010	000011	(0x00119883)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

# Handhabung von Konstanten

## 16-Bit Konstante mit `addi`:

### Hochsprache

```
// int ist ein  
// vorzeichenbehaftetes  
// 32b Wort
```

```
int a = 0x4f3c;
```

### MIPS Assemblersprache

```
# $s0 = a
```

```
addi $s0, $0, 0x4f3c
```

**32-Bit Konstante** mit Load Upper Immediate (`lui`) und `ori`:  
(`lui` lädt den 16-Bit Direktwert in obere Registerhälfte und setzt untere Hälfte auf 0.)

### Hochsprache

```
int a = 0xFEDC8765;
```

### MIPS Assemblersprache

```
# $s0 = a
```

```
lui $s0, 0xFEDC
```

```
ori $s0, $s0, 0x8765
```

# Multiplikation und Division

**Spezialregister:** `lo`, `hi`

- `32b × 32b` Multiplikation, **64b** Produkt

```
mult $s0, $s1
```

Ergebnis in `{hi, lo}`

- `32b` Division, `32b` **Quotient**, `32b` **Rest**

```
div $s0, $s1
```

Quotient in `lo`

Rest in `hi`

- **Lesen** von Daten aus Spezialregistern („*move from ...*“)

```
mflo $s2
```

```
mfhi $s3
```

# Verzweigungen und Sprünge



- Ändern der Ausführungsreihenfolge von Befehlen
- Arten von Verzweigungen: Beispiele
  - **Bedingte**
    - branch if equal (`beq`): Verzweige, wenn gleich
    - branch if not equal (`bne`): Verzweige, wenn ungleich
  - **Unbedingte Verzweigungen**
    - jump (`j`): Springe
    - jump register (`jr`): Springe auf Adresse aus Register
    - jump and link (`jal`): Springe und merke Adresse des nächsten Befehls

# Wiederholung: Programm im Speicher

## Assemblersprache

## Maschinensprache

lw	\$t2, 32(\$0)	0x8C0A0020
add	\$s0, \$s1, \$s2	0x02328020
addi	\$t0, \$s3, -12	0x2268FFF4
sub	\$t0, \$t3, \$t5	0x016D4022

## Abgespeichertes Programm

Adresse	Befehle
⋮	⋮
0040000C	0 1 6 D 4 0 2 2
00400008	2 2 6 8 F F F 4
00400004	0 2 3 2 8 0 2 0
00400000	8 C 0 A 0 0 2 0 ← PC
⋮	⋮

Hauptspeicher

# Bedingte Verzweigungen (beq)



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

## # MIPS Assemblersprache

```
addi $s0, $0, 4           # $s0 = 0 + 4 = 4
addi $s1, $0, 1           # $s1 = 0 + 1 = 1
sll  $s1, $s1, 2          # $s1 = 1 << 2 = 4
beq $s0, $s1, target    # Verzweigung wird genommen
addi $s1, $s1, 1          # nicht ausgeführt
sub  $s1, $s1, $s0        # nicht ausgeführt

target:                   # Positionsmarkierung (label)
add  $s1, $s1, $s0        # $s1 = 4 + 4 = 8
```

**Label** sind Namen für Stellen (Adressen) im Programm. Sie müssen anders als Mnemonics heißen und haben einen Doppelpunkt am Ende.

# Nicht genommene Sprünge (bne)



## # MIPS Assemblersprache

```
addi $s0, $0, 4           # $s0 = 0 + 4 = 4
addi $s1, $0, 1           # $s1 = 0 + 1 = 1
sll  $s1, $s1, 2          # $s1 = 1 << 2 = 4
bne $s0, $s1, target    # Verzweigung nicht genommen
addi $s1, $s1, 1          # $s1 = 4 + 1 = 5
sub  $s1, $s1, $s0        # $s1 = 5 - 4 = 1

target:
add  $s1, $s1, $s0        # $s1 = 1 + 4 = 5
```

# Unbedingte Verzweigungen / Springen (j)



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

## # MIPS Assemblersprache

```
addi $s0, $0, 4           # $s0 = 4
addi $s1, $0, 1           # $s1 = 1
j     target              # Springe zu target
sra  $s1, $s1, 2          # nicht ausgeführt
addi $s1, $s1, 1          # nicht ausgeführt
sub  $s1, $s1, $s0        # nicht ausgeführt
```

target:

```
add  $s1, $s1, $s0        # $s1 = 1 + 4 = 5
```

# Unbedingte Verzweigungen (jr)



## # MIPS Assemblersprache

```
0x00002000    addi $s0, $0, 0x2010
0x00002004    jr   $s0
0x00002008    addi $s1, $0, 1
0x0000200C    sra  $s1, $s1, 2
0x00002010    lw   $s3, 44($s1)
```

# Konstrukte in Hochsprachen



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

- `if`-Anweisungen
- `if/else`-Anweisungen
- `while`-Schleifen
- `for`-Schleifen

# If-Anweisung

## Hochsprache

```
if (i == j)
    f = g + h;

f = f - i;
```

## MIPS Assemblersprache

```
# $s0=f, $s1=g, $s2=h
# $s3=i, $s4=j
```

# If / Else-Anweisung

## Hochsprache

```
if (i == j)
    f = g + h;
else
    f = f - i;
```

## MIPS Assemblersprache

```
# $s0=f, $s1=g, $s2=h
# $s3=i, $s4=j
```

# While-Schleife



## Hochsprache

```
// berechnet  
// x = log2(128)
```

```
int pow = 1;  
int x   = 0;
```

```
while (pow != 128) {  
    pow = pow * 2;  
    x = x + 1;  
}
```

## MIPS Assemblersprache

```
# $s0 = pow, $s1 = x
```

Auch hier: Assemblersprache prüft auf **entgegengesetzte**  
Bedingung (`pow == 128`) als Hochsprache (`pow != 128`).

## Allgemeiner Aufbau:

**for** (*Initialisierung; Bedingung; Schleifenanweisung*)  
*Schleifenrumpf*

- *Initialisierung* : wird **einmal** vor Ausführung der Schleife ausgeführt
- *Bedingung* : wird vor **Beginn** jedes Schleifendurchlaufs geprüft
- *Schleifenanweisung* : wird am **Ende** jedes Schleifendurchlaufs ausgeführt
- *Schleifenrumpf* : wird einmal ausgeführt, wenn Bedingung **wahr** ist

# For-Schleifen

## Hochsprache

```
// addiere Zahlen  
// von 0 to 9 auf  
int sum = 0;  
int i;  
  
for (i = 0; i != 10;  
      i = i+1) {  
    sum = sum + i;  
}
```

## MIPS Assemblersprache

```
# $s0 = i, $s1 = sum
```

# Kleiner-als Vergleiche



## Hochsprache

```
// addiere Zweierpotenzen  
// kleiner gleich 100
```

```
int sum = 0;  
int i;
```

```
for (i=1; i<101; i=i*2) {  
    sum = sum + i;  
}
```

## MIPS Assemblersprache

```
# $s0 = i, $s1 = sum
```

$\$t1 = 1$  if  $i < 101$ .

# Datenfelder (*arrays*)



- Nützlich um auf eine große Zahl von Daten **gleichen Typs** zuzugreifen
- Zugriff auf einzelne Elemente über **Index**
- **Größe** eine Arrays: Anzahl von Elementen im Array

# Verwendung von Arrays

- Array mit 5 Elementen
- **Basisadresse**, hier 0x12348000
  - Adresse des **ersten** Array-Elements
  - Index **0**, geschrieben als `array[0]`
- Erster Schritt für Zugriff auf Element: Lade Basisadresse des Arrays in Register

0x12340010	array[4]
0x1234800C	array[3]
0x12348008	array[2]
0x12348004	array[1]
0x12348000	array[0]

# Verwendung von Arrays



## // Hochsprache

```
int array[5];  
array[0] = array[0] * 2;  
array[1] = array[1] * 2;
```

## # MIPS Assemblersprache

```
# Basisadresse von array = $s0
```

# Bearbeite Array in for-Schleife



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

## // Hochsprache

```
int array[1000];  
int i;
```

```
for (i=0; i < 1000; i = i + 1)  
    array[i] = array[i] * 8;
```

## # MIPS Assemblersprache

```
# $s0 = Basisadresse von Array, $s1 = i
```

# Zeichendarstellung im ASCII-Code

## *American Standard Code for Information Interchange*

- Definiert für gängige Textzeichen einen 7b breiten Code
- Einfach, aber schon älter
- Heute Unicode: breitere Darstellung für *alle* Textzeichen

**Beispiel:** “S” = 0x53, “a” = 0x61, “A” = 0x41

Klein- und Großbuchstaben liegen auseinander um 0x20 (32).

# Zuordnung von Zeichen zu Codes



#	Char	#	Char	#	Char	#	Char	#	Char	#	Char
20	space	30	0	40	@	50	P	60	'	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	§	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	'	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	:	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	
2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	0	5F	_	6F	o		



## Definitionen

- **Aufrufer:** Ursprung des Prozeduraufrufs (hier `main`)
- **Aufgerufener:** aufgerufene Prozedur (hier `sum`)

## Hochsprache

```
void main()
{
    int y;
    y = sum (42, 7);
    ...
}

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



## Aufrufer:

- Übergibt Argumente (aktuelle Parameter) an Aufgerufenen
- Springt Aufgerufenen an

## Aufgerufener:

- Führt Prozedur/Funktion aus
- Gibt Ergebnis (Rückgabewert) an Aufrufer zurück (für Funktion)
- Springt hinter Aufrufstelle zurück
- Darf keine Register oder Speicherstellen überschreiben, die im Aufrufer genutzt werden

# Prozedur- und Funktionsaufruf Konventionen für MIPS



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

- Prozeduraufruf: “jump and link (`jal`)”
- Rücksprung: “jump register (`jr`)”
- Register für Argumente: `$a0` – `$a3`
- Register für Ergebnis: `$v0`

# Prozedur- und Funktionsaufruf



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

## Hochsprache

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

## MIPS Assemblersprache

```
0x00400200 main: jal simple  
0x00400204          add $s0, $s1, $s2  
...
```

```
void simple () {  
    return;  
}
```

```
0x00401020 simple: jr $ra
```

`void` bedeutet, dass `simple` keinen Rückgabewert hat.  
- Also eine Prozedur und keine Funktion ist

# Prozedur- und Funktionsaufruf



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

## Hochsprache

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

```
void simple() {  
    return;  
}
```

## MIPS Assemblersprache

```
0x00400200 main: jal    simple  
0x00400204         add    $s0, $s1, $s2  
...  
0x00401020 simple: jr    $ra
```

**jal:** springt zu simple  
speichert PC+4 im Spezialregister \$ra "return address register"  
Hier: \$ra = 0x00400204 nach Ausführung von jal

**jr \$ra:** springt zur Adresse in \$ra, hier also 0x00400204.

# Aufrufargumente und Rückgabewert



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

## MIPS Konventionen:

- Argumentwerte (aktuelle Parameter):  $\$a0 - \$a3$
- Rückgabewert (Funktionswert, Ergebnis):  $\$v0$

# Aufrufargumente und Rückgabewert



## Hochsprache

```
int main()
{
    int y;
    ...
    y = diffosums (2, 3, 4, 5); // 4 Argumente
    ...
}
```

```
int diffosums (int f, int g, int h, int i)
// 4 formale Parameter
{
    int result;
    result = (f + g) - (h + i);
    return result; // Rückgabewert
}
```

# Aufrufargumente und Rückgabewert



## MIPS Assemblersprache

```
# $s0 = y
```

```
main:
```

```
...
```

```
addi $a0, $0, 2    # Argument 0 = 2
addi $a1, $0, 3    # Argument 1 = 3
addi $a2, $0, 4    # Argument 2 = 4
addi $a3, $0, 5    # Argument 3 = 5
jal  diffofsums    # Prozeduraufruf
add  $s0, $v0, $0  # y = Rückgabewert
```

```
...
```

```
# $s0 = Rückgabewert
```

```
diffofsums:
```

```
add $t0, $a0, $a1  # $t0 = f + g
add $t1, $a2, $a3  # $t1 = h + i
sub $s0, $t0, $t1  # result = (f + g) - (h + i)
add $v0, $s0, $0   # Lege Rückgabewert in $v0 ab
jr  $ra            # Rücksprung zum Aufrufer
```

# Aufrufargumente und Rückgabewert

## MIPS Assemblersprache

```
# $s0 = result
diffofsums:
    add $t0, $a0, $a1    # $t0 = f + g
    add $t1, $a2, $a3    # $t1 = h + i
    sub $s0, $t0, $t1    # result = (f + g) - (h + i)
    add $v0, $s0, $0     # Lege Rückgabewert in $v0 ab
    jr  $ra              # Rücksprung zum Aufrufer
```

- `diffofsums` überschreibt drei Register: `$t0`, `$t1` und `$s0`
- `diffofsums` kann benötigte Register temporär auf **Stack** sichern

# Stack (auch Stapel- oder Kellerspeicher)

- Speicher für **temporäres Zwischenspeichern** von Werte
- Agiert wie ein **Stapel** (Beispiel: Teller)
  - Zuletzt aufgelegter Teller wird zuerst heruntergenommen
  - “last in, first out” (LIFO)

**Dehnt sich aus:** Belegt mehr Speicher, wenn mehr Daten unterzubringen sind

**Zieht sich zusammen:** Belegt weniger Speicher, wenn zwischengespeicherte Daten nicht mehr gebraucht werden



# Stack

Wächst bei MIPS nach **unten** (von hohen zu niedrigeren Speicheradressen)

- Übliche Realisierung (deshalb auch **Kellerspeicher** genannt)

Stapelzeiger (“stack pointer”):  $\$sp$

- zeigt auf zuletzt auf dem Stack abgelegtes Datenelement

Adresse	Daten		Adresse	Daten	
7FFFFFFC	12345678	← $\$sp$	7FFFFFFC	12345678	
7FFFFFF8			7FFFFFF8	AABBCCDD	
7FFFFFF4			7FFFFFF4	11223344	← $\$sp$
7FFFFFF0			7FFFFFF0		
⋮	⋮		⋮	⋮	
⋮	⋮		⋮	⋮	
⋮	⋮		⋮	⋮	

# Verwendung des Stacks in Prozeduren

- Aufgerufene Prozeduren dürfen keine unbeabsichtigten Nebenwirkungen (“Seiteneffekte”) haben
- **Problem:** `diffofsums` überschreibt die drei Register `$t0`, `$t1`, `$s0`

## MIPS Assemblersprache

```
# $s0 = result
```

```
diffofsums:
```

```
    add $t0, $a0, $a1    # $t0 = f + g
```

```
    add $t1, $a2, $a3    # $t1 = h + i
```

```
    sub $s0, $t0, $t1    # result = (f + g) - (h + i)
```

```
    add $v0, $s0, $0      # Lege Rückgabewert in $v0 ab
```

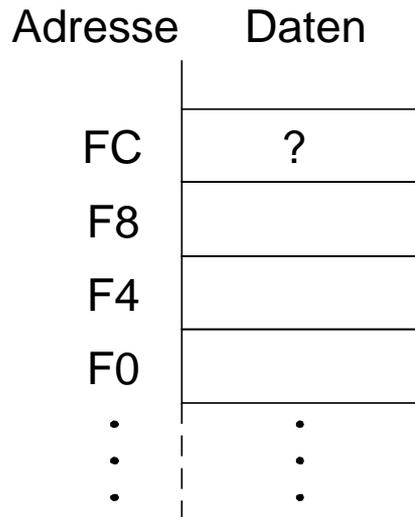
```
    jr  $ra              # Rücksprung zum Aufrufer
```

# Register auf Stack zwischenspeichern

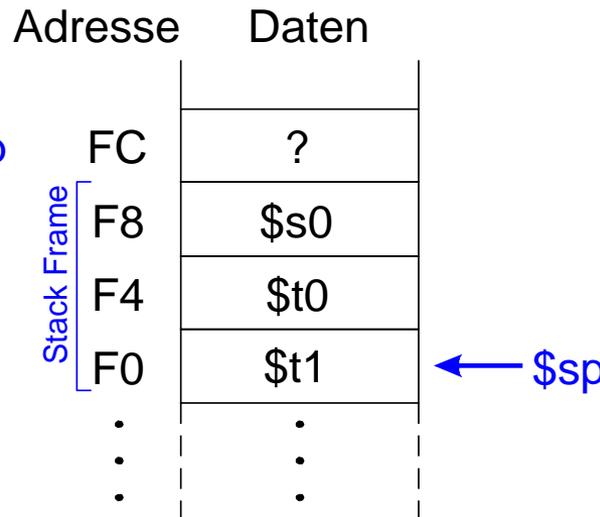


```
# $s0 = result
diffofsums:
    addi $sp, $sp, -12    # 3*4 Bytes auf Stack anfordern
                        # um drei 32b Register zu sichern
    sw    $s0, 8($sp)    # speichere $s0 auf Stack
    sw    $t0, 4($sp)    # speichere $t0 auf Stack
    sw    $t1, 0($sp)    # speichere $t1 auf Stack
    add   $t0, $a0, $a1   # $t0 = f + g
    add   $t1, $a2, $a3   # $t1 = h + i
    sub   $s0, $t0, $t1   # result = (f + g) - (h + i)
    add   $v0, $s0, $0    # Lege Rückgabewert in $v0 ab
    lw    $t1, 0($sp)    # stelle $t1 wieder vom Stack her
    lw    $t0, 4($sp)    # stelle $t0 wieder vom Stack her
    lw    $s0, 8($sp)    # stelle $s0 wieder vom Stack her
    addi  $sp, $sp, 12    # Platz auf Stack wird nicht mehr benötigt,
                        # wieder freigeben
    jr    $ra            # Rücksprung zum Aufrufer
```

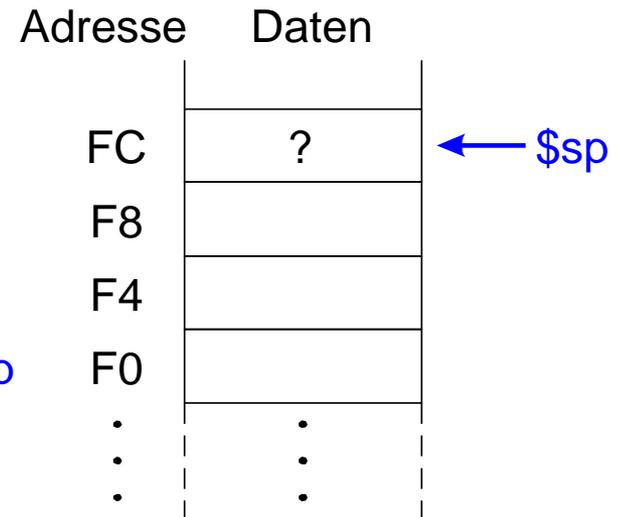
# Veränderung des Stacks während diffosums



(a)



(b)



(c)

# Sicherungskonventionen für Register



Erhalten <i>Gesichert vom Aufgerufenen</i>	Nicht erhalten <i>Gesichert vom Aufrufer</i>
$\$s0 - \$s7$	$\$t0 - \$t9$
$\$ra$	$\$a0 - \$a3$
$\$sp$	$\$v0 - \$v1$
Stack oberhalb von $\$sp$	Stack unterhalb von $\$sp$

# Mehrfache Prozeduraufrufe: Sichern von \$ra



```
proc1:
    addi $sp, $sp, -4    # Platz auf Stack anlegen
    sw   $ra, 0($sp)    # sichere $ra auf Stack
    jal  proc2
    ...
    lw   $ra, 0($sp)    # stelle $ra vom Stack wieder her
    addi $sp, $sp, 4    # Stapelspeicher wieder freigeben
    jr   $ra            # Rückkehr zum Aufrufer von proc1

proc2:
    ...
```

# Erhalten von Registern mittels Stack



```
# $s0 = result
diffofsums:
    addi $sp, $sp, -4    # Platz auf Stack für 4 Bytes anlegen
                        # reicht zum Sichern eines Registers
    sw   $s0, 0($sp)    # sichere $s0 auf Stack
                        # $t0 und $t1 brauchen nicht erhalten zu werden!
    add  $t0, $a0, $a1   # $t0 = f + g
    add  $t1, $a2, $a3   # $t1 = h + i
    sub  $s0, $t0, $t1   # result = (f + g) - (h + i)
    add  $v0, $s0, $0    # Lege Rückgabewert in $v0 ab
    lw   $s0, 0($sp)    # stelle $s0 vom Stack wieder her
    addi $sp, $sp, 4    # Gebe nicht mehr benötigten Speicher auf Stack frei
    jr   $ra            # Rücksprung zum Aufrufer
```

# Rekursive Prozeduraufrufe



## Hochsprache

```
int fakultaet (int n) {  
    if (n <= 1)  
        return 1;  
    else  
        return (n * fakultaet (n-1));  
}
```

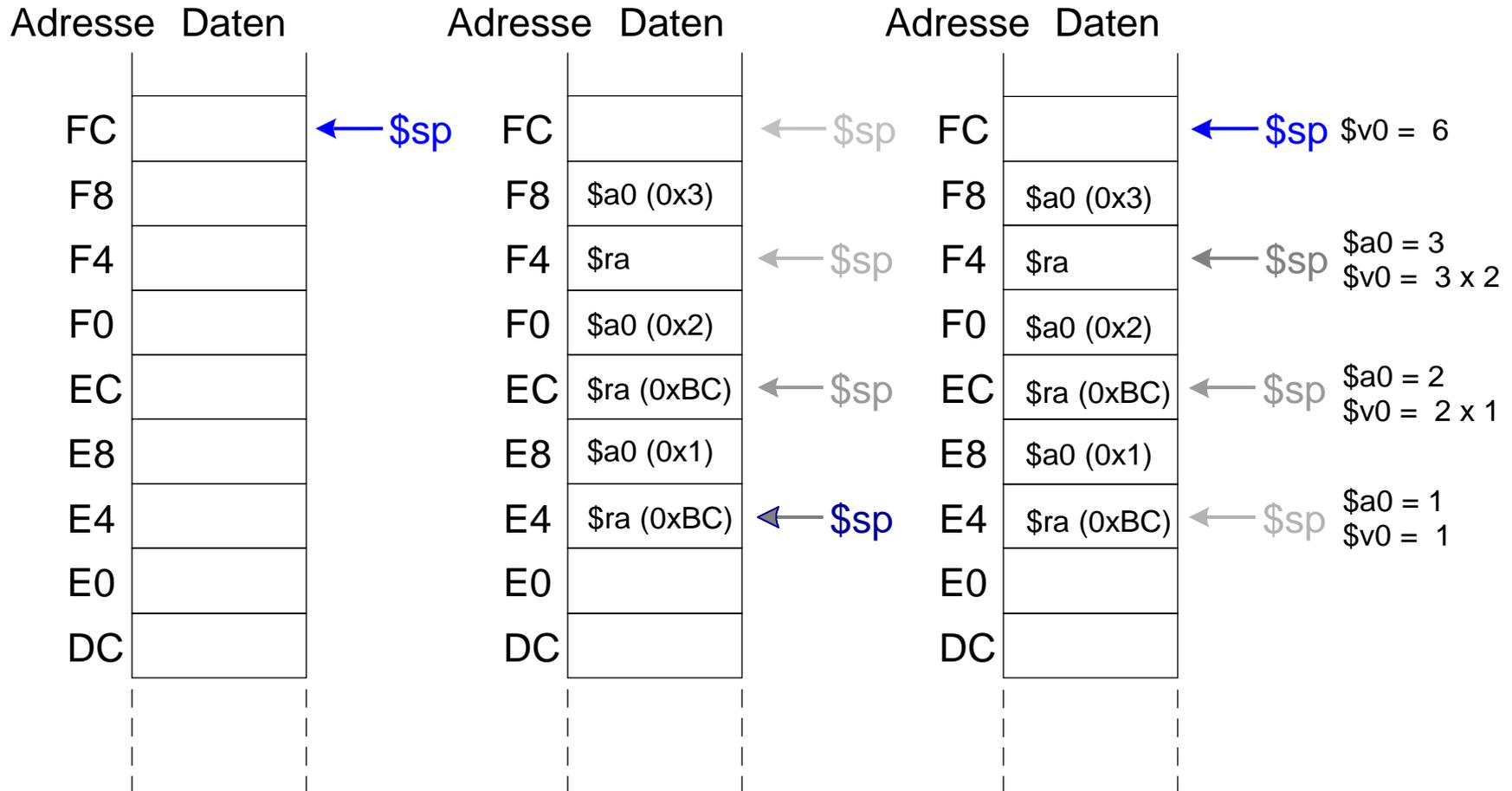
# Rekursive Prozeduraufrufe



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

## MIPS Assemblersprache

# Veränderung des Stacks bei rekursivem Aufruf



# Zusammenfassung: Prozeduraufruf

## Aufrufer

- Lege Aufrufparameter (aktuelle Parameter) in  $\$a0-\$a3$  ab
- Sichere zusätzlich benötigte Register auf Stack ( $\$ra$ , manchmal auch  $\$t0-t9$ ) - entsprechend Konvention über Erhaltung von Registern
- `jal aufrufener`
- Stelle gesicherte Register wieder her
- Hole evtl. Rückgabewert aus  $\$v0$  (bei Funktionen)

## Aufgerufener

- Sichere zu erhaltende verwendete Register auf Stack (üblicherweise  $\$s0-\$s7$ )
- Führe Berechnungen der Prozedur aus
- Lege Rückgabewert in ab  $\$v0$  (bei Funktionen)
- Stelle gesicherte Register wieder her
- `jr $ra`

# Adressierungsarten

## Wo kommen Operanden für Befehle her?

- Aus einem Register
- Direktwert aus Instruktion
- Relativ zu einer Basisadresse
  - Sonderfall: Relativ zum Programmzähler
- Pseudodirekt

# Adressierungsarten

## Aus Register (*register operands*)

- **Beispiel:** `add $s0, $t2, $t3`
- **Beispiel:** `sub $t8, $s1, $0`

## Direktwert aus Instruktion (*immediate*)

- 16b Direktwert als Operand verwenden
  - **Beispiel:** `addi $s4, $t5, -73`
  - **Beispiel::** `ori $t3, $t7, 0xFF`

# Adressierungsarten

## Relativ zu einer Basisadresse

Adresse eines Operanden im Speicher ist:

Basisadresse + Vorzeichenerweiterter Direktwert

▪ **Beispiel:** `lw $s4, 72($0)`

Adresse = `$0 + 72`

▪ **Beispiel:** `sw $t2, -25($t1)`

Adresse = `$t1 - 25`

# Adressierungsarten

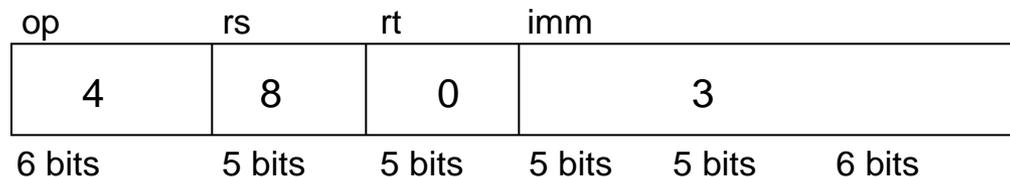
## Relativ zur nächsten Adresse im Programmzähler

```
0x10          beq    $t0, $0, else
0x14          addi   $v0, $0, 1
0x18          addi   $sp, $sp, i
0x1C          jr     $ra
0x20          else:  addi   $a0, $a0, -1
0x24          jal   fakultaet
```

### Assemblersprache

```
beq $t0, $0, else
(beq $t0, $0, 3)
```

### Bitfelder in Instruktion



# Adressierungsarten

## Pseudodirekte Operanden

### Auffüllen von entfallenen Bits (mit Nullen und (PC+4)[31:28])

**0x0040005C**            jal    sum

...

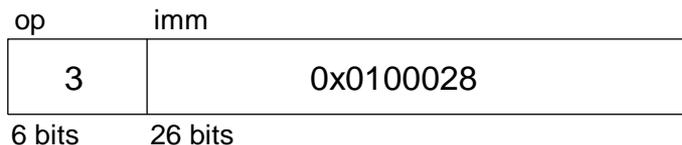
**0x004000A0**    sum:    add    \$v0, \$a0, \$a1

**32b Sprungzieladresse**    0000 0000 0100 0000 0000 0000 1010 0000    (0x004000A0)

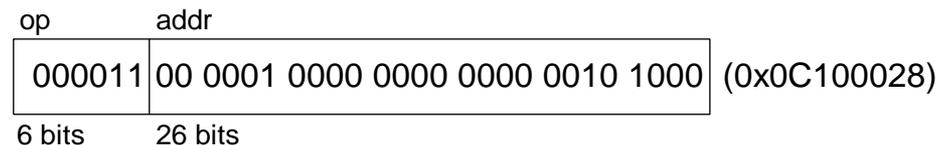
**26b Feld in J-Instruktion**    0000 0000 0100 0000 0000 0000 1010 0000    (0x0100028)

0    1    0    0    0    2    8

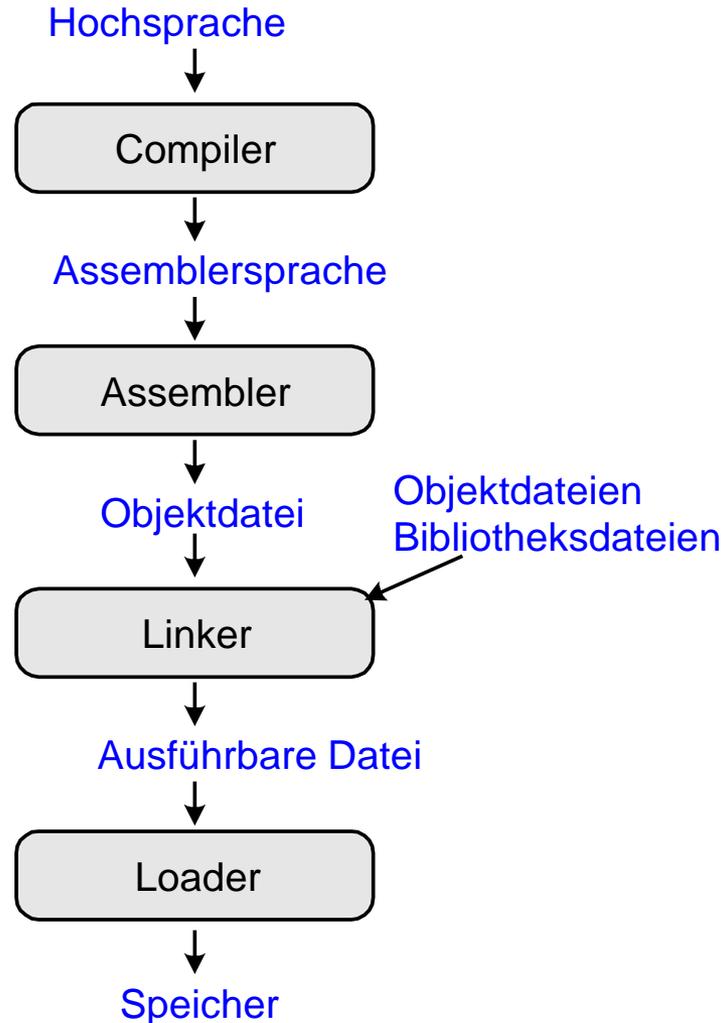
#### Bitfelder in Instruktion



#### Maschinencode



# Compilieren und Ausführen einer Anwendung



# Grace Hopper, 1906 - 1992



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

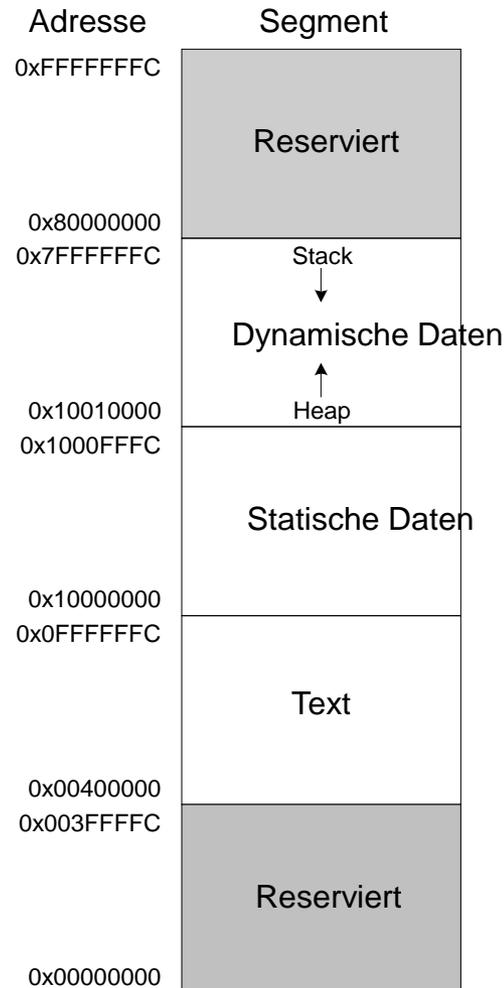
- Promovierte zum Dr. der Mathematik in Yale
- Entwickelte den ersten Compiler
- Half bei der Entwicklung von COBOL
- Prägte den Begriff „Debugging“
  - Elektromechanischer Harvard Mark-I Computer
- Hochdekorierte Marineoffizierin



# Was muss im Speicher abgelegt werden?

- Instruktionen (historisch auch genannt *Text*)
- Daten
  - Globale und statische: angelegt vor Beginn der Programmausführung
  - Dynamisch: während der Programmausführung angelegt
- Speicherobergrenze bei MIPS (32)?
  - Maximal  $2^{32} = 4$  Gigabytes (4 GB)
  - Von Adresse 0x00000000 bis 0xFFFFFFFF

# MIPS Speicherorganisation (*memory map*)



# Beispielprogramm in “C”



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
int f, g, y; // globale Variablen
```

```
int main(void)
{
    f = 2;
    g = 3;
    y = sum(f, g);

    return y;
}
```

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

# Beispielprogramm: MIPS Assemblersprache



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
int f, g, y; // globale Variablen

int main(void)
{
    f = 2;
    g = 3;

    y = sum(f, g);
    return y;
}

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

.data
f: .space 4           # Direktiven für Assembler
g: .space 4           # jeweils ein Wort, initialisiert
y: .space 4           # auf den Wert 0

.text
main:
    addi $sp, $sp, -4 # Stack Frame anlegen
    sw   $ra, 0($sp)  # sichere $ra
    addi $a0, $0, 2   # $a0 = 2
    sw   $a0, f       # f = 2
    addi $a1, $0, 3   # $a1 = 3
    sw   $a1, g       # g = 3
    jal  sum          # Aufruf von sum
    sw   $v0, y       # y = sum()
    lw   $ra, 0($sp)  # stelle $ra wieder her
    addi $sp, $sp, 4  # stelle $sp wieder her
    jr   $ra          # Rückkehr ins Betriebssystem

sum:
    add  $v0, $a0, $a1 # $v0 = a + b
    jr   $ra          # return
```

# Beispielprogramm: Symboltabelle



Symbol	Adresse

# Beispielprogramm: Ausführbare Datei

Dateikopf	Text Größe	Daten Größe
	0x34 (52 bytes)	0xC (12 bytes)
Textsegment	Adresse	Instruktion
	0x00400000	0x23BDFFFC
	0x00400004	0xAFBF0000
	0x00400008	0x20040002
	0x0040000C	0xAF848000
	0x00400010	0x20050003
	0x00400014	0xAF858004
	0x00400018	0x0C10000B
	0x0040001C	0xAF828008
	0x00400020	0x8FBF0000
	0x00400024	0x23BD0004
	0x00400028	0x03E00008
	0x0040002C	0x00851020
	0x00400030	0x03E00008
Datensegment	Adresse	Datum
	0x10000000	0
	0x10000004	0
	0x10000008	0

```

addi $sp, $sp, -4
sw  $ra, 0 ($sp)
addi $a0, $0, 2
sw  $a0, 0x8000 ($gp)
addi $a1, $0, 3
sw  $a1, 0x8004 ($gp)
jal  0x0040002C
sw  $v0, 0x8008 ($gp)
lw  $ra, 0 ($sp)
addi $sp, $sp, -4
jr  $ra
add $v0, $a0, $a1
jr  $ra

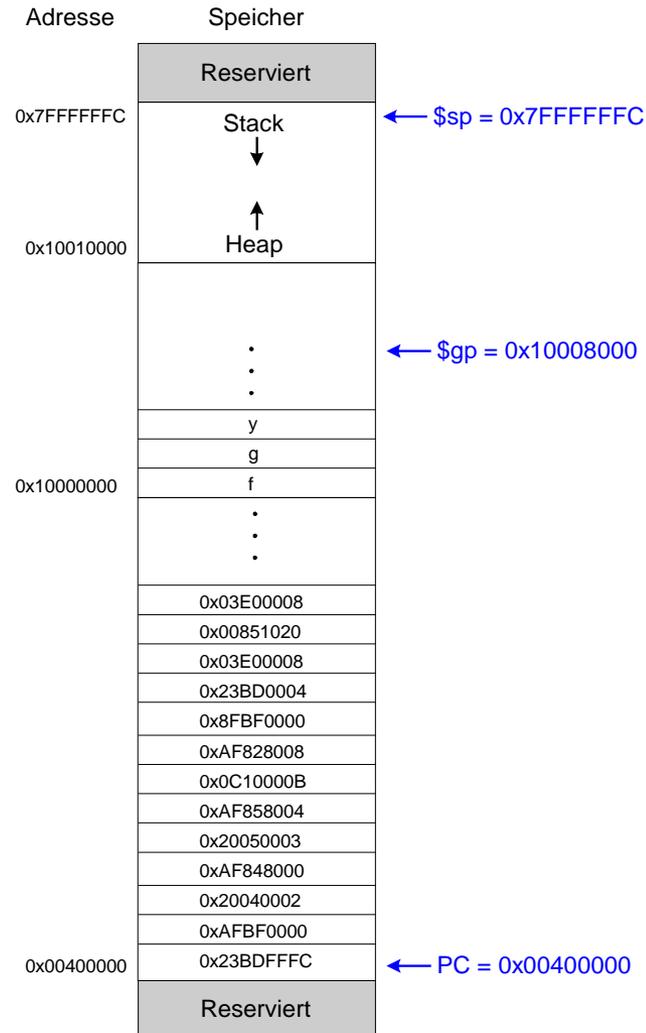
```

```

f
g
y

```

# Beispielprogramm im Speicher



# Dies und Das



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

- Pseudobefehle
- Ausnahmebehandlung (*exceptions*)
- Befehle für vorzeichenbehaftete und vorzeichenlose Zahlen
- Gleitkommabefehle

# Beispiele für Pseudobefehle



Pseudobefehle	MIPS Befehle
<pre>li \$s0, 0x1234AA77</pre>	
<pre>move \$s1, \$s2</pre>	
<pre>nop</pre>	
<pre>not \$s1, \$s2</pre>	

# Ausnahmebehandlung (*exceptions*)

- Abweichen von der normalen **Ausführungsreihenfolge** von Befehlen
  - Beim Auftreten **außergewöhnlicher** Umstände (*exception*)
  - Automatischer Aufruf spezieller Prozedur: Ausnahmebehandlung (*exception handler*)
- **Auslösung** der Ausnahmebehandlung z.B. durch
  - Hardware, dann genannt **Interrupt** (z.B. Tippen einer Taste auf Tastatur)
  - Software, dann genannt **Trap** (z.B. Versuch der Ausführung einer unbekanntem Instruktion)

# Ausnahmebehandlung (*exceptions*)

Beim Auftreten der Ausnahme

- **Grund** der Ausnahme wird gespeichert
- **Sprung** zur Ausnahmebehandlung auf Adresse 0x80000180
- Dann **Wiederaufnahme** der normalen Programmausführung

# Spezialregister für Ausnahmebehandlung

- **Außerhalb** des regulären Registerfeldes
  - **Cause**
    - Enthält den Grund für Ausnahme
  - **EPC** (Exception PC)
    - Enthält den regulären PC an dem die Aufnahme auftrat
- **EPC und Cause**: Nicht Bestandteil des “**eigentlichen**” MIPS-Prozessors
  - Ausgelagert in **Coprozessor** (unterstützt Hauptprozessor)
  - Genauer: **Coprozessor 0**

# Spezialregister für Ausnahmebehandlung

- Datenaustausch mit Coprozessor (hier nur lesen)
  - “Move from Coprocessor 0”
    - `mfc0 $t0, EPC`
  - Lädt Inhalt des Spezialregisters `EPC` in reguläres Register `$t0`
    - Analog auch für `Cause`

# Auslöser für Ausnahmen



Ausnahme	Cause
Hardware Interrupt	0x00000000
Systemaufruf	0x00000020
Breakpoint / Division durch 0	0x00000024
Unbekannte Instruktion	0x00000028
Arithmetischer Überlauf	0x00000030

# Ausnahmen

## Wenn eine Ausnahme passiert:

1. Prozessor speichert Grund und Auftritts-PC in `Cause` und `EPC`
2. Prozessor springt Ausnahmebehandlung an (`0x80000180`)

## Ausnahmebehandlung:

1. Speichere Register auf Stack
2. Lese `Cause` Spezialregister: `mfc0 $t0, Cause`
3. Bearbeite Ausnahme
4. Stelle alle Register wieder her
5. Springe zurück ins eigentlich laufende Programm

```
mfc0 $k0, EPC
```

```
jr $k0
```

# Vorzeichenbehaftete und –lose Befehle



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

- Addition und Subtraktion
- Multiplikation und Division
- Set-less-than

# Addition und Subtraktion

**Vorzeichenbehaftet:** `add`, `addi`, `sub`

- Gleiche Operation wie vorzeichenlose Versionen
- Aber: Prozessor löst Ausnahme bei arithmetischem Überlauf aus

**Vorzeichenlos:** `addu`, `addiu`, `subu`

- Prüft nicht auf Überlauf
- **Hinweis:** `addiu` vorzeichenerweitert den Direktwert

# Multiplikation und Division



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

**Vorzeichenbehaftet:** `mult, div`

**Vorzeichenlos:** `multu, divu`

# Set Less Than

**Vorzeichenbehaftet:** `slt, slti`

**Vorzeichenlos:** `sltu, sltiu`

- **Hinweis:** `sltiu` vorzeichenerweitert den Direktwert vor dem Vergleich mit dem Register

# Laden von 8b und 16b breiten Daten



## Vorzeichenbehaftet:

- Vorzeichenerweiterung schmale Daten auf volle 32b Registerbreite
- Load halfword: `lh`
- Load byte: `lb`

## Vorzeichenlos:

- Fülle schmale Daten mit Nullen auf volle 32b Registerbreite auf
- Load halfword unsigned: `lhu`
- Load byte: `lbu`

## Im Digitaltechnik kennengelernt:

- Positive Zahlen: Vorzeichenlose Binärdarstellung
- Negative Zahlen
  - Zweierkomplement
  - Darstellung als Vorzeichen/Betrag

## Wo bleiben Brüche?

- Rationale Zahlen?
  
- Reelle Zahlen?

# Zahlen mit Bruchanteilen

## Zwei gängige Darstellungen:

- Festkomma (*fixed-point*)
- Gleitkomma (*floating-point*)

# Zahlen mit Bruchanteilen

## **Festkomma (*fixed-point*):**

Position des Kommas bleibt konstant

## **Gleitkomma (*floating-point*):**

Position des Kommas kann wandern, ist stets **rechts der höchstwertigen** Stelle. Angabe der Position des Kommas in Exponentenschreibweise

# Zahlen mit Bruchanteilen

## Festkomma (*fixed-point*):

Position des Kommas bleibt konstant

**Beispiel:** Dezimalsystem, **2** Vorkomma-, **3** Nachkommastellen

2,000 99,999 0,000 -2,718

**nicht:** 3,1415 365,250

## Gleitkomma (*floating-point*)

Position des Kommas ist stets **rechts der höchstwertigen Stelle**.

Angabe der Position des Kommas in Exponentenschreibweise

**Beispiel:** Dezimalsystem, insgesamt **5** Stellen

$2 \cdot 10^0$                        $9,9999 \cdot 10^1$                        $0 \cdot 10^0$                        $-2,718 \cdot 10^0$

$3,1415 \cdot 10^0$                        $3,6525 \cdot 10^2$                        $5 \cdot 10^6$

**nicht:**  $3,14159 \cdot 10^0$

Auch: Obergrenze für Exponenten, keine beliebig großen Zahlen darstellbar

# Binäre Festkommazahlen

Darstellung von 6,75 mit 4b für ganzen Anteil und 4b für Binärbruch:

01101100

0110 , 1100

$$2^2 + 2^1 + 2^{-1} + 2^{-2} = 6,75$$

- Binärkomma wird nicht explizit dargestellt: Position wird durch Format impliziert (hier: 4,4)
- Alle Leser und Schreiber von Festkommdatensystemen müssen dasselbe Format verwenden

# Binäre Festkommazahlen

## Beispiel:

Stelle  $7,5_{10}$  in 8b im 4,4-Festkommaformat dar

# Vorzeichenbehaftete Festkommazahlen



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Wie bei ganzen Zahlen: Zwei Darstellungen möglich

- Vorzeichen/Betrag
- Zweierkomplement
- Stelle  $-7,5_{10}$  in 8b als 4,4-Festkommazahl dar
  - **Vorzeichen/Betrag:**
  - **Zweierkomplement:**

# Binäre Gleitkommazahlen

- Binärkomma liegt immer genau rechts von höchstwertiger 1
- Ähnlich zur wissenschaftlichen Darstellung von Dezimalbrüchen
- Beispiel: 4.387.263 in wissenschaftlicher Darstellung

$$4,387263 \times 10^6$$

# Binäre Gleitkommazahlen

- Binärkomma liegt immer genau rechts von höchstwertiger 1
- Ähnlich zur wissenschaftlichen Darstellung von Dezimalbrüchen
- Beispiel: 4.387.263 in wissenschaftlicher Darstellung

$$4,387263 \times 10^6$$

- Allgemeine Schreibweise:

$$\pm M \times B^E$$

wobei

- **M** = Mantisse
- **B** = Basis
- **E** = Exponent
- Im Beispiel:  $M = 4,387263$  ,  $B = 10$  , and  $E = 6$

# Binäre Gleitkommazahlen



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

$$\pm M \times B^E$$

1 Bit

8 Bits

23 Bits



**Vorzeichen**

**Exponent**

**Mantisse**

# Binäre Gleitkommazahlen



**Vorzeichen**

**Exponent**

**Mantisse**

- **Beispiel:** Stelle den Wert  $228_{10}$  als 32b-Gleitkommazahl dar

Im folgenden drei Versionen, nur die **letzte** davon ist eine Standarddarstellung!

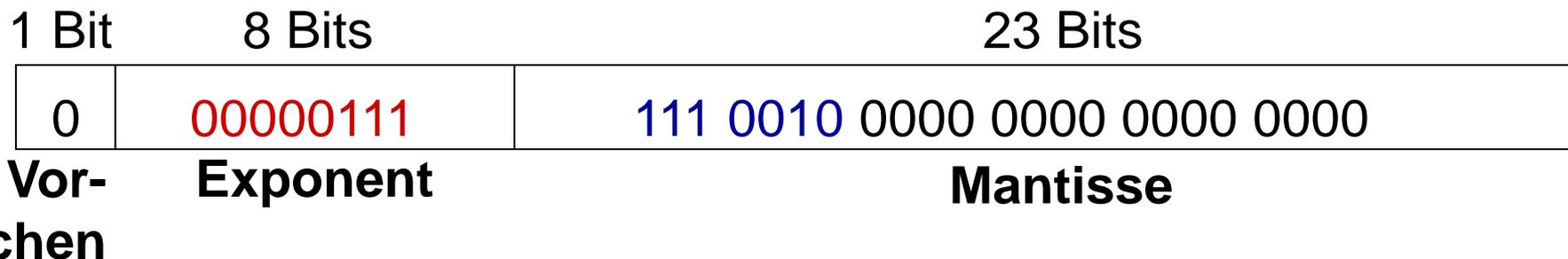
*IEEE 754, single precision format*

# Binäre Gleitkommadarstellung:

## 1. Versuch



- Wandele Dezimalzahl in Binärdarstellung um:
  - $228_{10} = 11100100_2 = 1,11001 \times 2^7$
- Trage nun Daten in die Felder des 32b Wortes ein:
  - Vorzeichenbit ist positiv (0)
  - Die 8b des Exponenten stellen den Wert 7 dar
  - Die verbliebenen 23 Bit stellen die Mantisse dar



# Binäre Gleitkommadarstellung:

## 2. Versuch



- **Beobachtung:** Das erste Bit der Mantisse ist so immer 1
  - $228_{10} = 11100100_2 = 1,11001 \times 2^7$
- Man kann sich das explizite Abspeichern der führenden 1 sparen
  - Die führende 1 wird implizit immer als präsent angenommen
- Stattdessen: **Speichere nur den Bruchanteil** (die “Nachkommastellen”) explizit ab

1 Bit

8 Bits

23 Bits

0	00000111	110 0100 0000 0000 0000 0000
---	----------	------------------------------

**Vor-**

**Exponent**

**Bruchanteil**

**zeichen**

# Binäre Gleitkommadarstellung:

## 3. Versuch



- **Exponent kann auch negativ sein**
  - Idee: Zweierkomplement. Wäre möglich, hat aber praktische Nachteile
  - Besser: Exponent relativ zu konstantem Grundwert (Exzess, Biaswert) angeben
- Hier: **Biaswert = 127** ( $01111111_2$ )
  - Exponent mit Bias = Biaswert + Exponent
  - Exponent 7 wird also gespeichert als:  
 $127 + 7 = 134 = 0x10000110_2$
- Damit **IEEE 754 32-bit Gleitkommadarstellung** von  $228_{10}$

1 Bit

8 Bits

23 Bits

0	10000110	110 0100 0000 0000 0000 0000
---	----------	------------------------------

**Vorz.**

**Exponent  
mit Bias**

**Bruchanteil**

# Beispiel IEEE 754

## Gleitkommadarstellung

Stelle  $-58,25_{10}$  gemäß dem IEEE 754 32-bit Gleitkommastandard dar



# IEEE 754 Gleitkommadarstellung: Sonderfälle



Nicht alle benötigten Werte nach dem Schema darstellbar

- **Beispiel:** 0, hat keine führende 1

Wert	Vorz.	Exponent	Bruchanteil
0	X	00000000	000000000000000000000000000000
$\infty$	0	11111111	000000000000000000000000000000
$-\infty$	1	11111111	000000000000000000000000000000
NaN	X	11111111	Ein Wert $\neq 0$

NaN steht für “Not a Number” und stellt häufig Rechenfehler dar

Beispiele:  $\sqrt{-1}$  oder  $\log(-5)$  oder eine Zahl durch 0.

# Genauigkeit der Gleitkommadarstellungen

- Einfache Genauigkeit (***single-precision***):
  - 32-bit Darstellung
  - 1 Vorzeichenbit, 8 Exponentenbits, 23 Bits für Bruchanteil
  - Exponentenbias = 127
- Doppelte Genauigkeit (***double-precision***):
  - 64-bit Darstellung
  - 1 Vorzeichenbit, 11 Exponentenbits, 52 Bits für Bruchanteil
  - Exponentenbias = 1023

# Rundungsmodi für Gleitkommazahlen

- **Overflow:** Betrag der Zahl ist zu groß, um korrekt dargestellt zu werden
- **Underflow:** Zahl ist zu nahe bei 0, um korrekt dargestellt zu werden
- **Rundungsmodi:**
  - Abrunden zu minus Unendlich
  - Aufrunden zu plus Unendlich
  - Hin zu Null
  - Hin zu nächster darstellbarer Zahl
- **Beispiel:** Runde  $1,100101$  ( $1,578125_{10}$ ) auf 3 Bits Bruchanteil
  - Ab:  $1,100$
  - Auf:  $1,101$
  - Zu Null:  $1,100$
  - Zu nächster:  $1,101$  ( $1,625$  liegt näher an  $1,578125$  als an  $1,5$ )

# Addition von Gleitkommazahlen mit gleichem Vorzeichen



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

1. Exponenten- und Bruchanteile aus Gleitkommawort extrahieren
2. Bruchanteil um führende 1 erweitern, um Mantisse zu bilden
3. Vergleiche Exponenten
4. Schiebe Mantisse von Zahl mit kleinerem Exponenten nach rechts (bis Exponenten gleich sind)
5. Addiere Mantissen
6. Normalisiere Mantisse und passe Exponent an, falls nötig
7. Runde Ergebnis entsprechend dem gewählten Rundungsmodus
8. Baue Gleitkommawort aus Exponenten und Bruchanteil des Ergebnisses

# Beispiel: Addition von Gleitkommazahlen



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

**Addiere die beiden Gleitkommazahlen:**

0x3FC00000

0x40500000

# Beispiel: Addition von Gleitkommazahlen

## 1. Extrahiere Exponenten und Bruchanteile aus 32b Worten

1 Bit	8 Bits	23 Bits
0	01111111	100 0000 0000 0000 0000 0000
<b>Vorz.</b>	<b>Exponent</b>	<b>Bruchanteil</b>

1 Bit	8 Bits	23 Bits
0	10000000	101 0000 0000 0000 0000 0000
<b>Vorz.</b>	<b>Exponent</b>	<b>Bruchanteil</b>

**S**      **E**                      **F**

1. Zahl (N1):                       $S1 = 0, E1 = 127 (= \times 2^0), F1 = ,1$

2. Zahl (N2):                       $S2 = 0, E2 = 128 (= \times 2^1), F2 = ,101$

## 2. Erweitere Bruchanteile um führende 1, um Mantissen zu bilden

M1:                      1,1

M2:                      1,101

# Beispiel: Addition von Gleitkommazahlen

## 3. Vergleiche Exponenten

$E2 - E1 = 128 - 127 = 1$ , N1 muss also um ein Bit geschoben werden

## 4. Mantisse von Zahl mit kleinerem Exponenten entsprechend nach rechts schieben

schiebe M1:  $1,1 \gg 1 = 0,11 \quad (\times 2^1)$

## 5. Mantissen addieren (haben jetzt den gleichen Exponenten)

$$\begin{array}{r} + \quad 0,11 \times 2^1 \\ \hline \quad 1,101 \times 2^1 \\ \quad 10,011 \times 2^1 \end{array}$$

# Beispiel: Addition von Gleitkommazahlen

## 6. Normalisiere Mantisse und passe Exponenten an, falls nötig

$$10,011 \times 2^1 = 1,0011 \times 2^2$$

## 7. Runde Ergebnis entsprechend Rundungsmodus

Hier nicht nötig (passt in 23b)

## 8. Baue neues Gleitkommawort für Ergebnis aus Exponent und Mantisse

$$S = 0, E = 2 + 127 = 129 = 10000001_2, F = 001100..0$$

1 Bit	8 Bits	23 Bits
0	10000001	001 1000 0000 0000 0000 0000
<b>Vorz.</b>	<b>Exponent</b>	<b>Bruchanteil</b>

# Gleitkommabefehle

- Nicht Bestandteil des “**eigentlichen**” MIPS-Prozessors
- **Gleitkommakoprozessor** (Coprocessor 1)
- 32 32-bit **Gleitkommaregister** ( $\$f0 - \$f31$ )
  - Single precision
- Werte mit **doppelter** Genauigkeit benötigen je zwei **aufeinanderfolgende** Register
  - z.B.  $\$f0$  und  $\$f1$ ,  $\$f2$  und  $\$f3$ , etc.
  - Double precision-Register sind also:  $\$f0$ ,  $\$f2$ ,  $\$f4$ , etc.

# Gleitkommabefehle

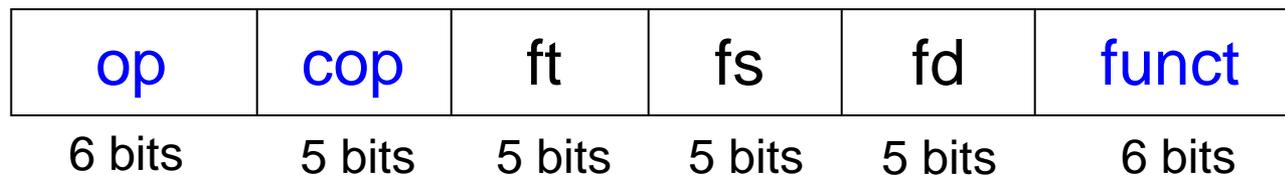
Namen	Registernummern	Zweck
$\$fv0 - \$fv1$	0, 2	Rückgabewerte
$\$ft0 - \$ft3$	4, 6, 8, 10	Temporäre Variablen
$\$fa0 - \$fa1$	12, 14	Prozedurargumente
$\$ft4 - \$ft8$	16, 18	Temporäre Variablen
$\$fs0 - \$fs5$	20, 22, 24, 26, 28, 30	Erhaltene Variablen

# Format für F-Typ Instruktionen



- Opcode = 17 (010001<sub>2</sub>)
- Drei Registeroperanden:
  - fs, ft: Quelloperanden
  - fd: Zieloperanden
- **Single-precision:**
  - cop = 16 (010000<sub>2</sub>)
  - add.s, sub.s, div.s, neg.s, abs.s, etc.
- **Double-precision:**
  - cop = 17 (010001<sub>2</sub>)
  - add.d, sub.d, div.d, neg.d, abs.d, etc.

## F-Typ



# Weitere Gleitkommabefehle



Setzt boole'sches **Spezialregister** bei Vergleichen : `fpcond`

- Gleichheit: `c.seq.s`, `c.seq.d`
- Kleiner-als: `c.lt.s`, `c.lt.d`
- Kleiner-als-oder-gleich: `c.le.s`, `c.le.d`
- Beispiel: `c.lt.s $fs1, $fs2`

Bedingte **Verzweigung** abhängig von Spezialregister

- `bc1f`: springt falls `fpcond = FALSE`
- `bc1t`: springt falls `fpcond = TRUE`
- Beispiel: `bc1f toosmall`

Loads und Stores: jeweils **Single precision**

- `lwc1: lwc1 $ft1, 42($s1)`
- `swc1: swc1 $fs2, 17($sp)`
- Double precision braucht je zwei Anweisungen

Bisher **Architektur**

- Programmierersicht

Nun **Mikroarchitektur**

- Aufbau der zugrundeliegenden **Hardware**