# Rechnerorganisation – Kapitel 6

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

TECHNISCHE UNIVERSITÄT DARMSTADT

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
  Büro: S2/02 (Piloty Gebäude), Raum E102



Embedded Systems & Applications

# Wo ich herkomme

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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 - )



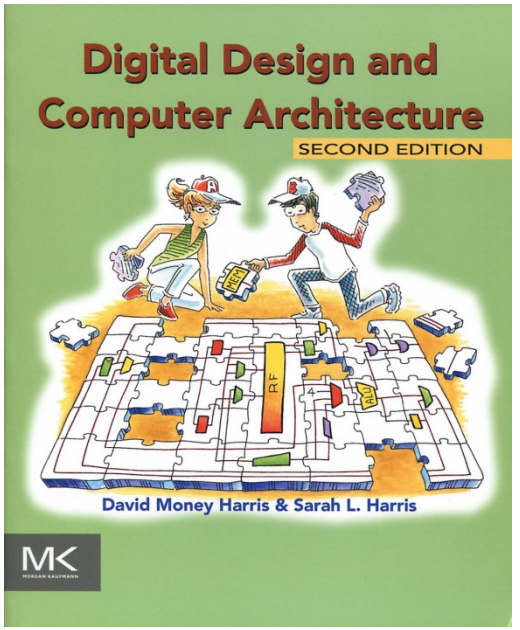

Embedded Systems & Applications

# Lehr- und Anschauungsmaterial

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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



Embedded Systems & Applications

# Überblick

TECHNISCHE UNIVERSITÄT DARMSTADT

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

TECHNISCHE UNIVERSITÄT DARMSTADT

- Rechtzeitig die Vorlesungen anschauen und Übungen mitmachen

  die aktuelle Vorlesung baut auf die vorgehenden Vorlesungen/Stoff 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

# Vorlesung Ablauf

TECHNISCHE UNIVERSITÄT DARMSTADT

- Wir machen eine kurze (2 Minuten) Pause in ungefähr der Mitte der Vorlesung.
- Bitte sich nach vorne Platz nehmen.
- Falls Sie während der Vorlesung ein Gespräch mit Ihren Kommilitonen ausführen möchten, bitte das außerhalb des Hörsaals machen.

# Organisatorisches

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Übungen:** Organisatorisches von dem Assistent jetzt erklärt…

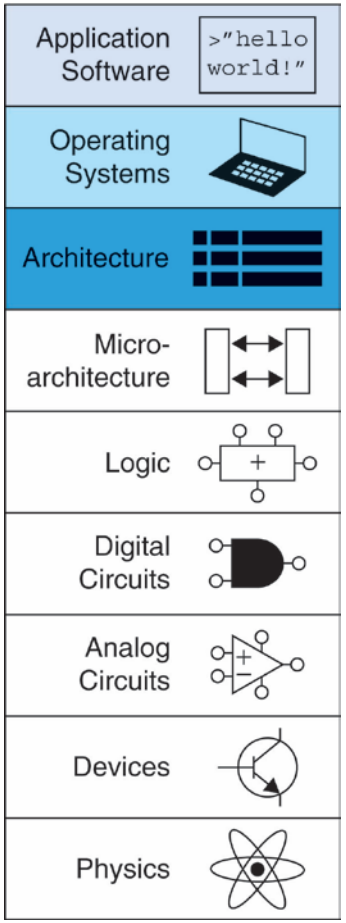
- **Webseite:** www.esa.informatik.tu-darmstadt.de
  - unter Lehre → SS 2016 → Rechnerorganisation
  - Vorlesungsfolien, Aufzeichnungen der Vorlesungen, kommentierte Vorlesungsfolien

- **Prüfung:** 18. Juli 2016 (vorläufig)

# Kapitel 6: Themen

TECHNISCHE UNIVERSITÄT DARMSTADT

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



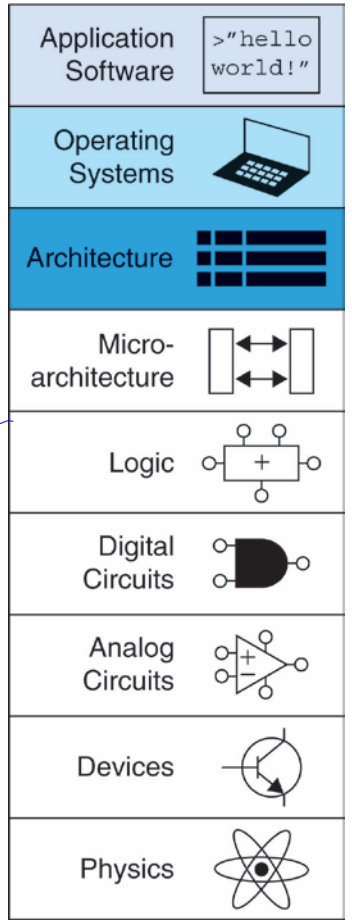



Embedded Systems & Applications

# Einleitung

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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






Embedded Systems & Applications

# Assemblersprache

TECHNISCHE UNIVERSITÄT DARMSTADT

**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
- **Maschinensprache:** maschinenlesbares Format (1'en und 0'en)

# MIPS Architektur

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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



Embedded Systems & Applications

# 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



Embedded Systems & Applications

# Befehle: Addition

TECHNISCHE UNIVERSITÄT DARMSTADT

**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



Embedded Systems & Applications

# Befehl: Subtraktion

TECHNISCHE UNIVERSITÄT DARMSTADT

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

| Hochsprache | MIPS Assemblersprache |
| --- | --- |
| `a = b - c;` | `sub a, b, c` |

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



Embedded Systems & Applications

# Entwurfsprinzip 1

TECHNISCHE UNIVERSITÄT DARMSTADT

## Regularität vereinfacht Entwurf

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



Embedded Systems & Applications

# Befehle: Komplexere Abläufe

TECHNISCHE UNIVERSITÄT DARMSTADT

Komplexere Abläufe werden durch Folgen von einfachen Befehlen realisiert

**Hochsprache**

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

**MIPS Assemblersprache**

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

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

```
# Kommentare bis Zeilende
```



Embedded Systems & Applications

# Entwurfsprinzip 2

TECHNISCHE UNIVERSITÄT DARMSTADT

## 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



Embedded Systems & Applications

# Entwurfsprinzip 2

TECHNISCHE UNIVERSITÄT DARMSTADT

## 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)



Embedded Systems & Applications

# Operanden

TECHNISCHE UNIVERSITÄT DARMSTADT

Ein Prozessor hat physikalische Speicherorte für die Operanden von Befehlen

**Mögliche Speicherorte**



Embedded Systems & Applications

# Operanden

TECHNISCHE UNIVERSITÄT DARMSTADT

Ein Prozessor hat physikalische Speicherorte für die Operanden von Befehlen

## Mögliche Speicherorte

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



Embedded Systems & Applications

# Operanden: Register

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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



Embedded Systems & Applications

# Entwurfsprinzip 3

TECHNISCHE UNIVERSITÄT DARMSTADT

## Kleiner ist schneller

- MIPS stellt nur eine kleine Anzahl von Registern bereit

- Kann in schnellerer Hardware realisiert werden als größeres Registerfeld



Embedded Systems & Applications

# MIPS Registerfeld

TECHNISCHE UNIVERSITÄT DARMSTADT

| 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

TECHNISCHE UNIVERSITÄT DARMSTADT

**Register:**

- Kenntlich 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

Saved



Embedded Systems & Applications

# Operanden: Register

TECHNISCHE UNIVERSITÄT DARMSTADT

## Zunächst benutzen wir nur

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

Später mehr …

# Befehle mit Registerangaben

TECHNISCHE UNIVERSITÄT DARMSTADT

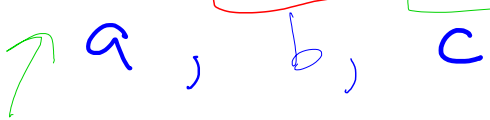
Rückblick auf add-Befehl

**Hochsprache**

```
a = b + c
```

**MIPS Assemblersprache**

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






Embedded Systems & Applications

# Operanden: Speicher

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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



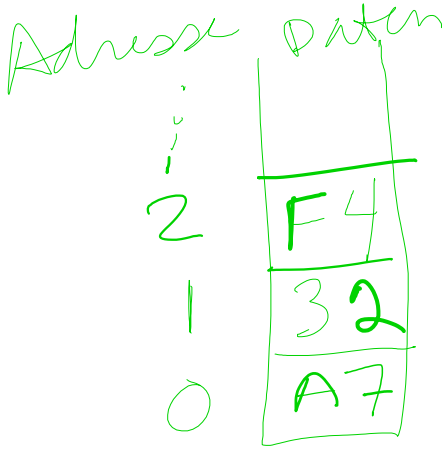
Embedded Systems & Applications

# Wort- und Byte-Adressierung von Daten im Speicher

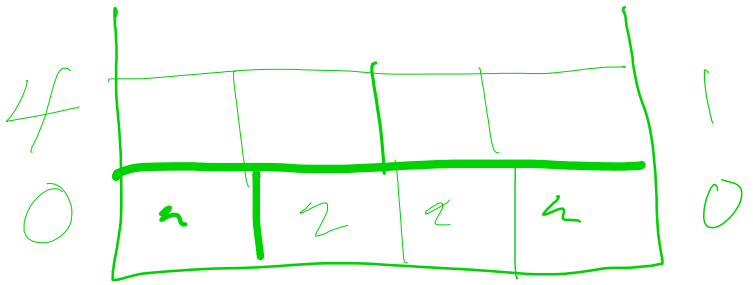
TECHNISCHE UNIVERSITÄT DARMSTADT

- MIPS ist **byte-adressiert**, das heißt...

  **jedes Byte hat eine eindeutige Adresse**










Embedded Systems & Applications

# Wort- und Byte-Adressierung von Daten im Speicher

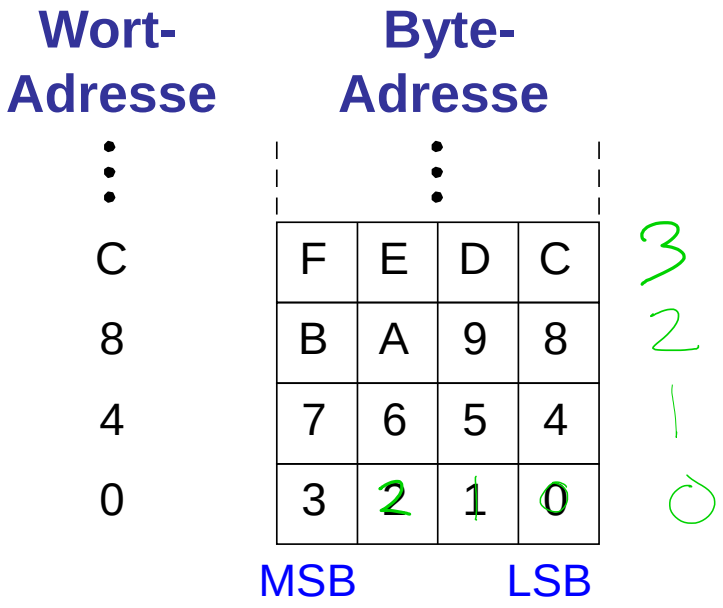
TECHNISCHE UNIVERSITÄT DARMSTADT

- 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






Embedded Systems & Applications

# Wort- und Byte-Adressierung von Daten im Speicher

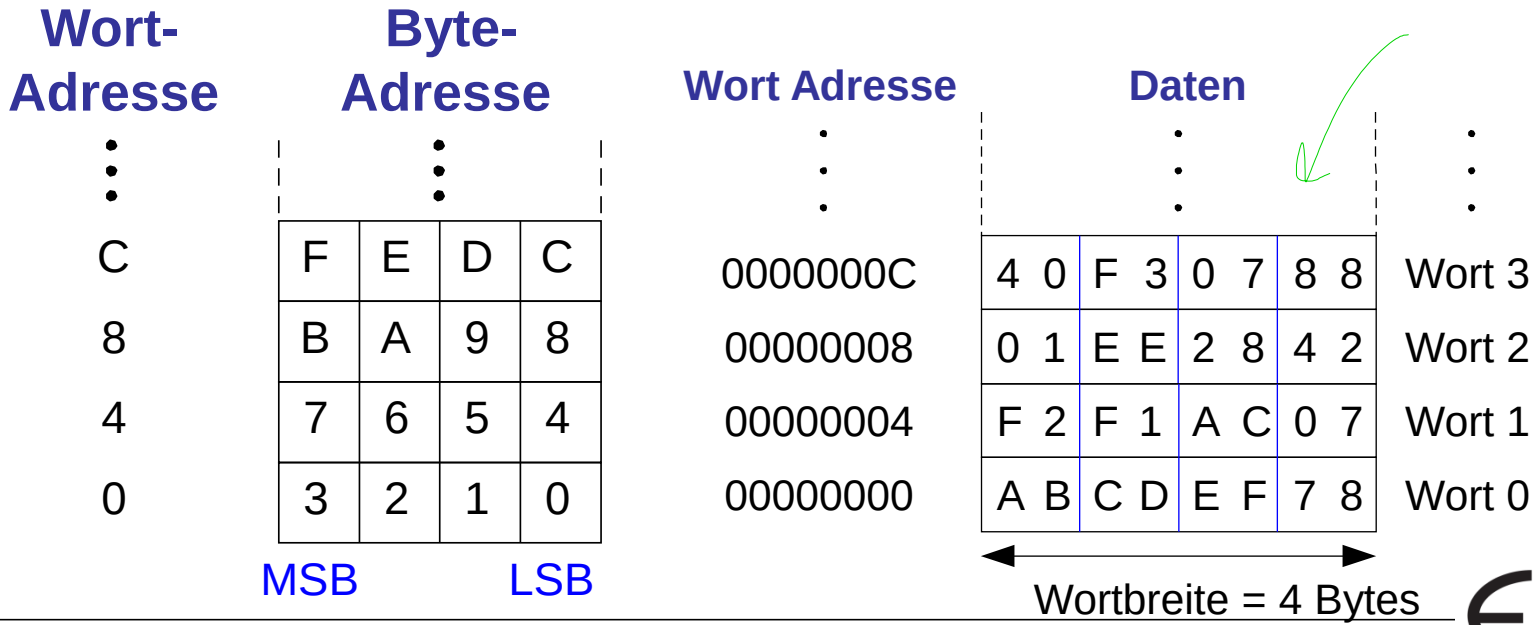
TECHNISCHE UNIVERSITÄT DARMSTADT

- 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



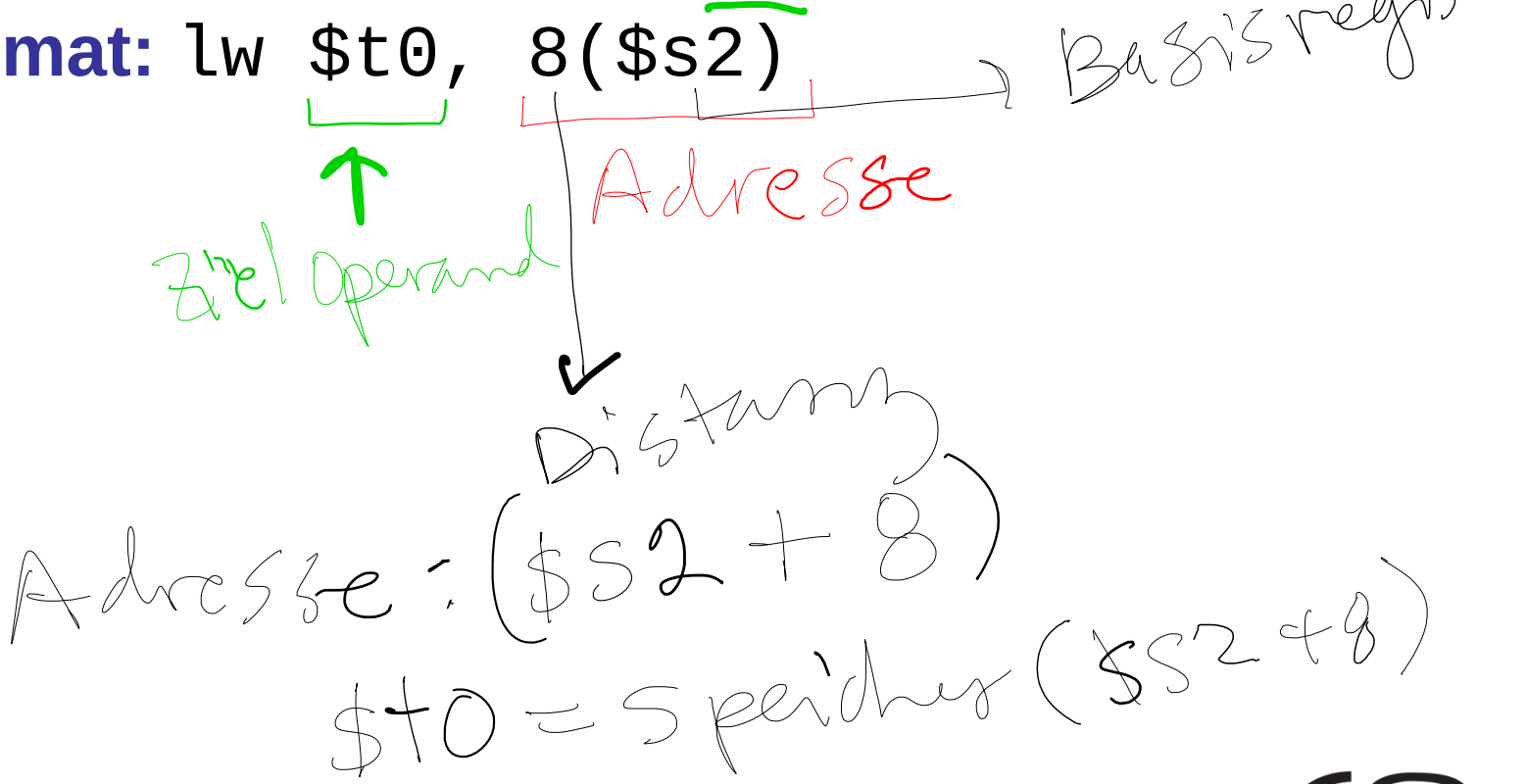



Embedded Systems & Applications

# Lesen aus byte-adressiertem Speicher

TECHNISCHE UNIVERSITÄT DARMSTADT

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






Embedded Systems & Applications

# Lesen aus byte-adressiertem Speicher

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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**

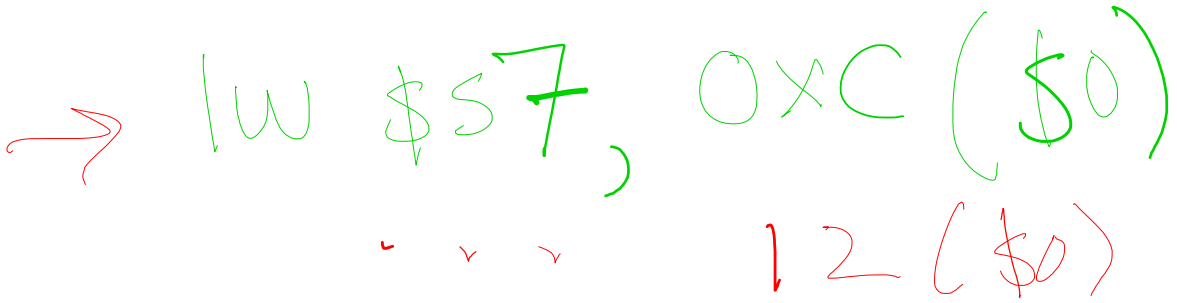


Embedded Systems & Applications

# Lesen aus byte-adressiertem Speicher

TECHNISCHE UNIVERSITÄT DARMSTADT

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




| 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



Embedded Systems & Applications

# Lesen aus byte-adressiertem Speicher

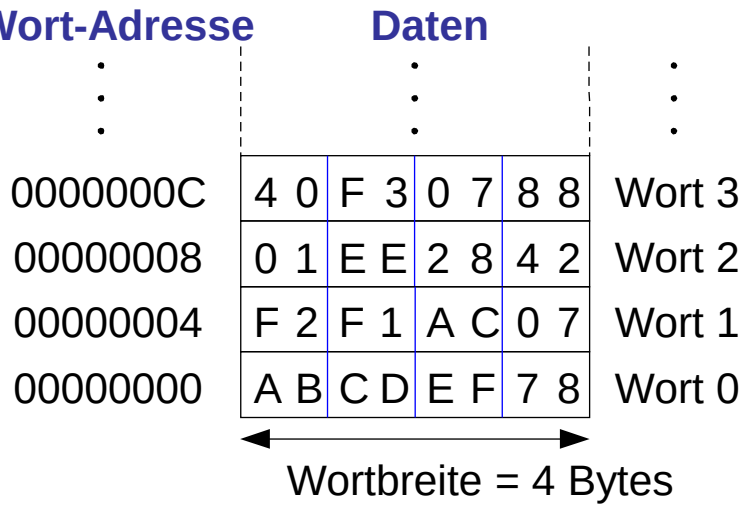
**Beispiel:** Lese Datenwort 3 (Speicheradresse 0xC) nach $s7

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

**Adressarithmetik:**

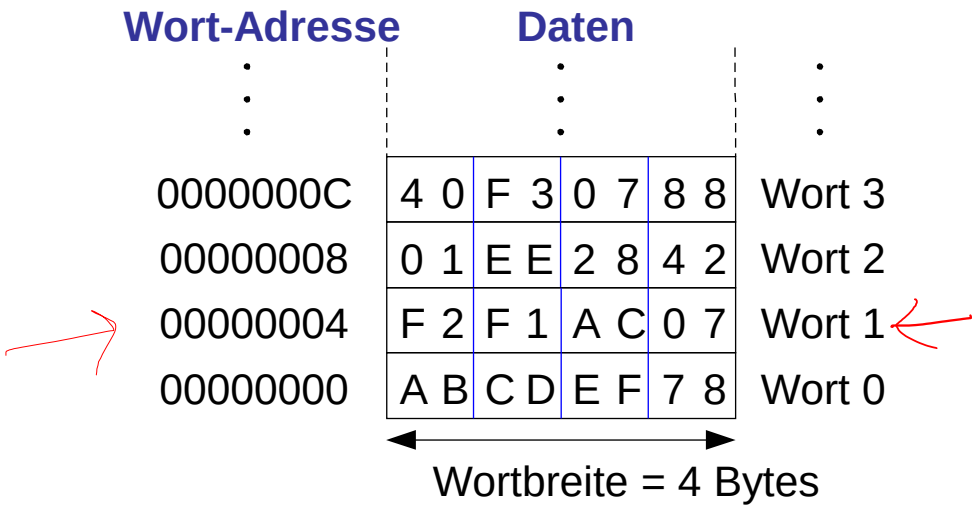
- Basisadresse ($0) plus Distanz (*offset*) (0xC)
- Adresse = ($0 + 12) = 12

Nach Abarbeiten des Befehls hat $s7 den Wert 0x40F30788



# Lesen aus byte-adressiertem Speicher

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



# Lesen aus byte-adressiertem Speicher

TECHNISCHE UNIVERSITÄT DARMSTADT

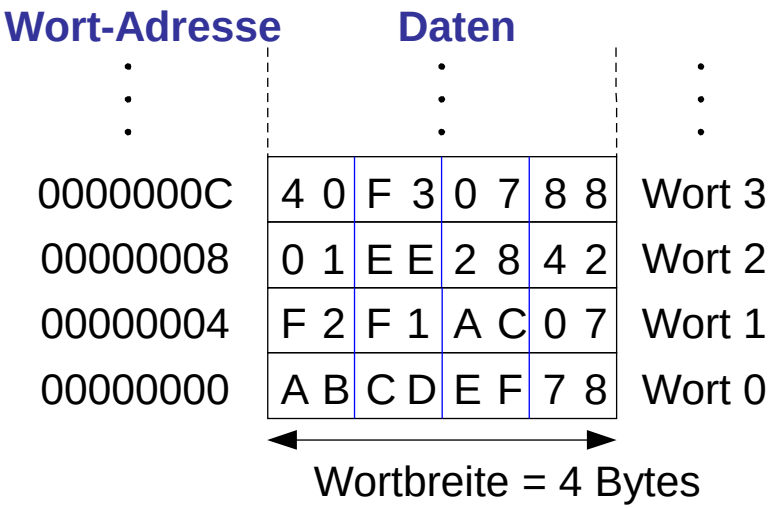
**Beispiel:** Lese Datenwort 1 nach `$s5`

```
lw $s5, 4($0)
```

**Adressarithmetik:**

- Basisadresse (`$0`) plus Distanz (*offset*) (4)
- Adresse = ($0 + 4) = 4

Nach Abarbeiten des Befehls hat `$s5` den Wert 0xF2F1AC07






Embedded Systems & Applications

# Schreiben in byte-adressiertem Speicher

TECHNISCHE UNIVERSITÄT DARMSTADT

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

Adresse

Adresse: ($0 + 0x1C)



Embedded Systems & Applications

# Schreiben in byte-adressiertem Speicher

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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. Speicheradrese 7x4=28 = 0x1C)



Embedded Systems & Applications

# Schreiben in byte-adressiertem Speicher

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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. Speicheradrese 7x4=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`



Embedded Systems & Applications

# Schreiben in byte-adressiertem Speicher

TECHNISCHE UNIVERSITÄT DARMSTADT

**Beispiel:** Schreiben Sie den Wert in $t2 in Speicherwort 15

15 × 4 = 60

sw $t2, 60($0)



Embedded Systems & Applications

# Schreiben in byte-adressiertem Speicher

TECHNISCHE UNIVERSITÄT DARMSTADT

**Beispiel:** Schreiben Sie den in $t2 in Speicherwort 15

```
addi $t2, $0, 79
sw $t2, 60($0)
```

**Adressarithmetik:**

- Basisadresse (`$0`) plus Distanz (*offset*) (60)
- Adresse = (`$0` + 60) = 60

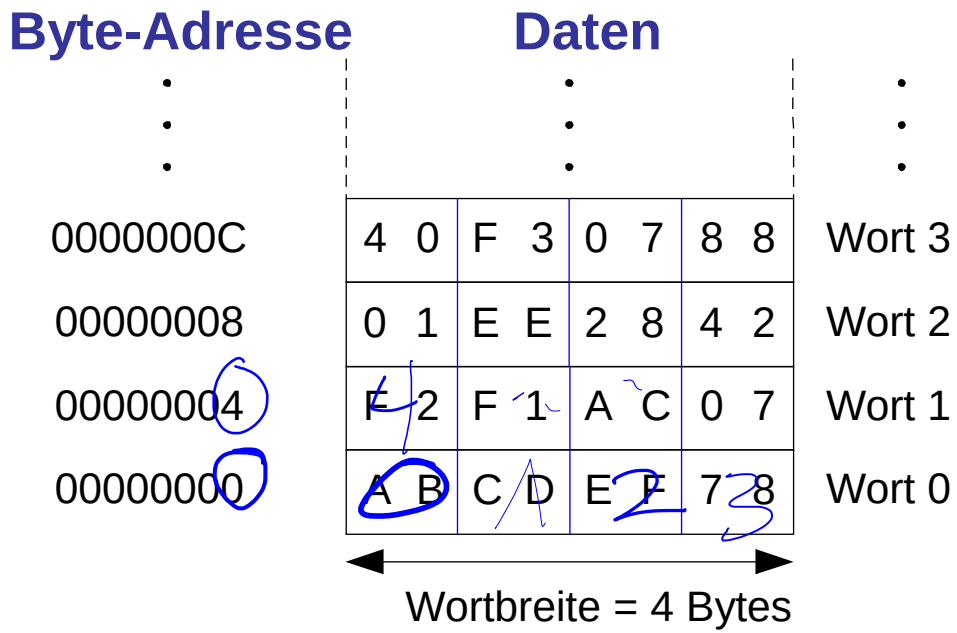
Nach Abarbeiten des Befehls hat Speicheradresse 60 den Wert 79.

# Byte-addressierbarer Speicher

TECHNISCHE UNIVERSITÄT DARMSTADT

Speicherbefehle können auf Worten oder Bytes arbeiten

**Worte:** `lw` / `sw` **Bytes:** `lb` / `sb`






Embedded Systems & Applications

# Speicherorganisation: Big-Endian und Little-Endian

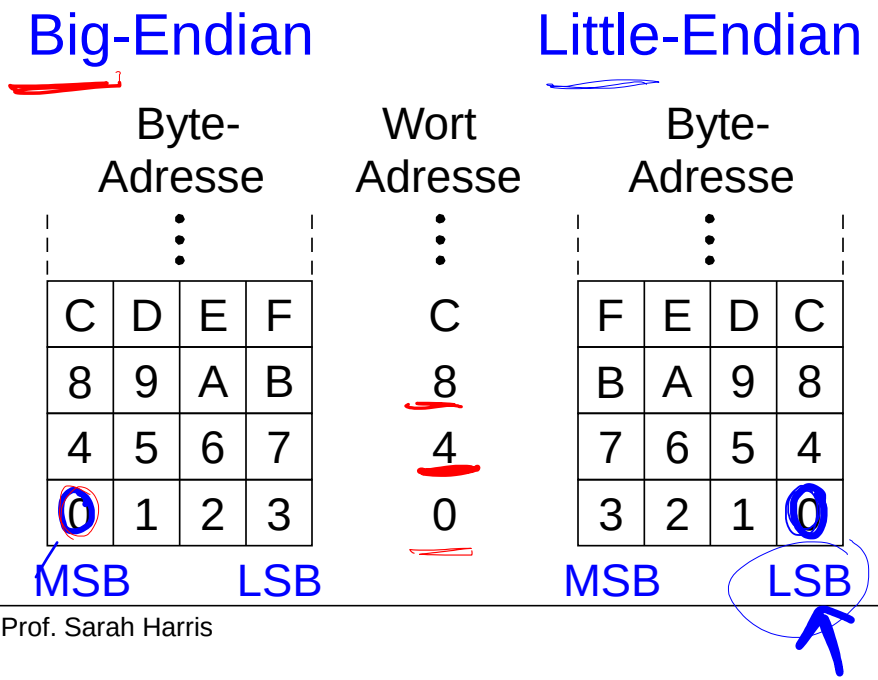
TECHNISCHE UNIVERSITÄT DARMSTADT

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






Embedded Systems & Applications

# Speicherorganisation: Big-Endian und Little-Endian

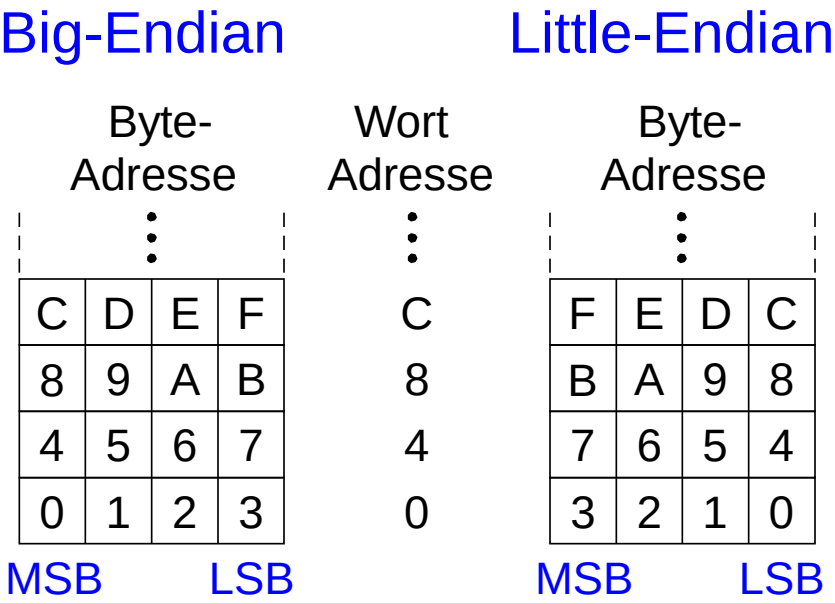
TECHNISCHE UNIVERSITÄT DARMSTADT

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






Embedded Systems & Applications

# 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?



Embedded Systems & Applications

# Beispiel: Big-Endian und Little-Endian

TECHNISCHE UNIVERSITÄT DARMSTADT

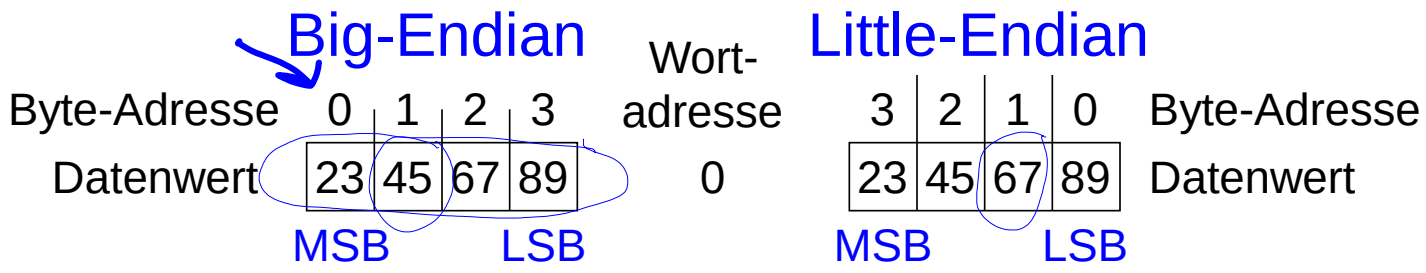
**Annahme:** `$t0` enthält den Wert 0x23456789

**Programm:**

```
sw $t0, 0($0)
lb $s0, 1($0)
```

**Fragen:** Welchen Wert hat `$s0` nach Ausführung auf einem…

- … Big-Endian Prozessor? 0x00000045
- … Little-Endian Prozessor? 0x00000067






Embedded Systems & Applications

# Entwurfsprinzip 4

TECHNISCHE UNIVERSITÄT DARMSTADT

## 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



Embedded Systems & Applications

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

TECHNISCHE UNIVERSITÄT DARMSTADT

- `lw` und `sw` 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
```



Embedded Systems & Applications

# Maschinensprache

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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)



Embedded Systems & Applications

# Befehlsformat R-Typ (Register Typ)

TECHNISCHE UNIVERSITÄT DARMSTADT

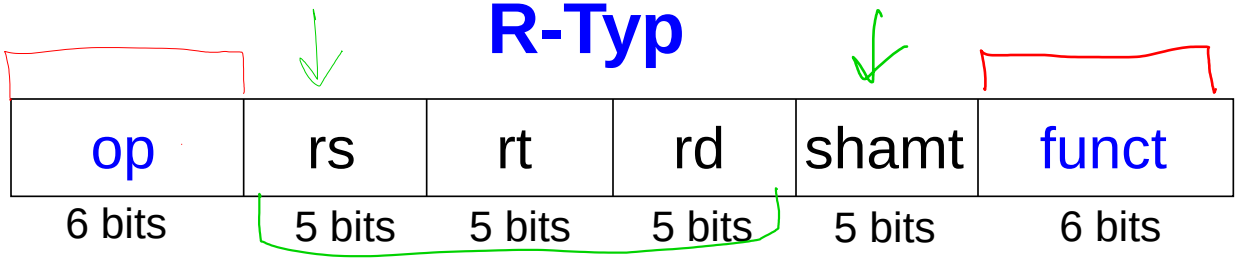
## 3 Registeroperanden

- **rs:** Quellregister
- **rt:** Quellregister
- **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






Embedded Systems & Applications

# Beispiele für Befehle vom R-Typ

TECHNISCHE UNIVERSITÄT DARMSTADT

## 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 |

## Maschinensprache

| 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
```



Embedded Systems & Applications

# Zusammenfassung von der letzten Vorlesung

TECHNISCHE UNIVERSITÄT DARMSTADT

## MIPS Architektur

**Assemblersprache:**

- **Befehle:** `add`, `sub`, `lw`, `sw`, `addi`
- **Operanden:** Register, Speicher, Direktwert (immediate)
  - 32 32b Register
  - Byte-adressierbarer Speicher
  - 16b Zweierkomplement Direktwert im Befehl

**Machinensprache:**

- **32b breit:** Assemblerbefehl in Machinenbefehl umsetzen
- **R-Typ, I-Typ, J-Typ**



Embedded Systems & Applications

# Befehlsformat R-Typ (Register Typ)

TECHNISCHE UNIVERSITÄT DARMSTADT

## 3 Registeroperanden

- **rs:** Quellregister
- **rt:** Quellregister
- **rd:** Zielregister

add rd, rs, rt

add $sv, $t1, $s2

## 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**




Embedded Systems & Applications

# Beispiele für Befehle vom R-Typ

TECHNISCHE UNIVERSITÄT DARMSTADT

## 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 |

## Maschinensprache

| 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
```



Embedded Systems & Applications

# Befehlsformat I-Typ (Immediate Typ)

TECHNISCHE UNIVERSITÄT DARMSTADT

addi $s0, $s2, -4

## 3 Operanden:

- **rs:** Quellregister
- **rt:** Quell/Zielregister
- **imm:** 16b Direktwert im Zweierkomplement

## Andere Angaben:

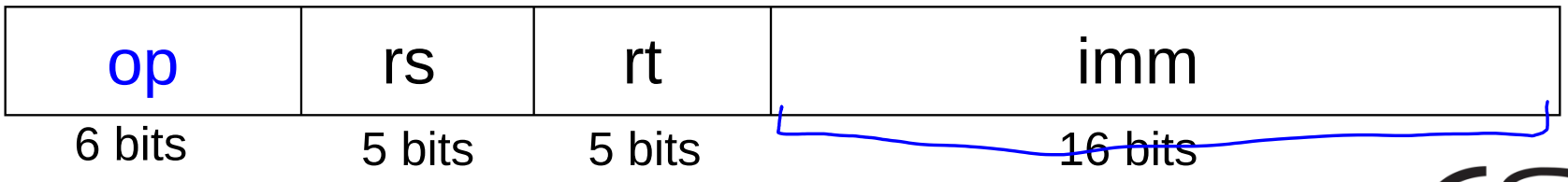
- **op:** Opcode

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**






Embedded Systems & Applications

# Wiederholung: Zahlen

TECHNISCHE UNIVERSITÄT DARMSTADT

**Beispiele:** 4b Zahlen:

**Vorzeichenlose** Zahlen: $0000_2$ bis $1111_2$ (Dezimal: 0 bis 15)

**Vorzeichenbehaftete** Zahlen: 1000 bis 0111 (Dezimal: -8 bis 7)

(genannt: zweierkomplemente Zahlen)



Embedded Systems & Applications

# Beispiele

TECHNISCHE UNIVERSITÄT DARMSTADT

## Vorzeichenlosezahl

**1 1 0 0**

$2^3$ $2^2$ $2^1$ $2^0$

8 4 2 1

$= 8 + 4 = 12$

## Zweierkomplementezahl

**1 0 0 1**

$-2^3$ $2^2$ $2^1$ $2^0$

$-8$ 4 2 1

$= -8 + 1 = -7$



Embedded Systems & Applications

# Beispiele

TECHNISCHE UNIVERSITÄT DARMSTADT

## Vorzeichenlosezahl

$$\begin{array}{cccc} 1 & 1 & 0 & 0 \\ \hline 2^3 & 2^2 & 2^1 & 2^0 \end{array} \quad = 8 + 4 = 12$$

## Zweierkomplementezahl

$$\begin{array}{cccc} 1 & 0 & 0 & 1 \\ \hline -2^3 & 2^2 & 2^1 & 2^0 \end{array} \quad = -8 + 1 = -7$$



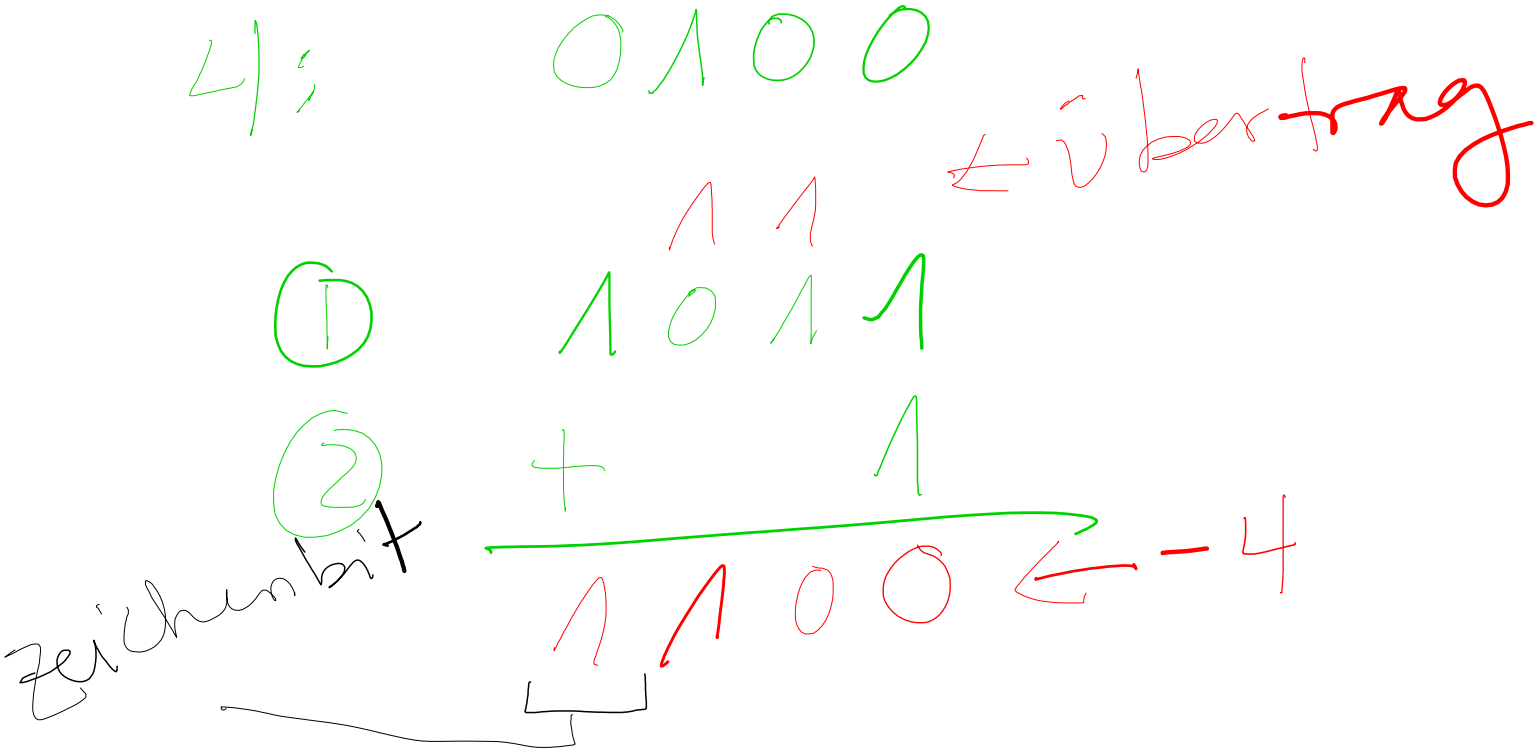
Embedded Systems & Applications

# Wiederholung: Zweierkomplementzahlen

TECHNISCHE UNIVERSITÄT DARMSTADT

## Beispiel:

Stellen Sie -4 als ein 4b Zweierkomplementzahl dar:






Embedded Systems & Applications

# Wiederholung: Zweierkomplementzahlen

TECHNISCHE UNIVERSITÄT DARMSTADT

## Beispiel:

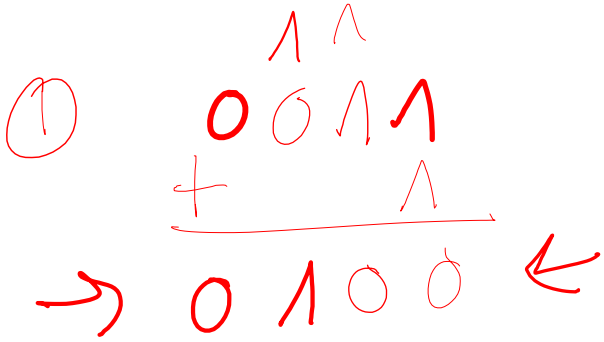
Stellen Sie -4 als ein 4b Zweierkomplementzahl dar:

4 = 0100

**Lösung:**

1. Bits invertieren: 1011
2. 1 darauf addieren: + 1

**Ergebnis:** **1100**






Embedded Systems & Applications

# Zweierkomplement arithmetisch bilden

TECHNISCHE UNIVERSITÄT DARMSTADT

In **beide** Richtungen anwendbar

- Vorzeichenwechsel: *k* ➔ *-k*

**Algorithmus**

1. Alle Bits invertieren (0➔1, 1➔0)
2. Dann 1 addieren

**Beispiel:** Vorzeichenwechsel von $3_{10} = 00011_2$

**1. $11100_2$**

**2. $11101_2 = -3_{10}$**

**Beispiel:** Vorzeichenwechsel von $-3_{10} = 11101_2$

**1. $00010_2$**

**2. $00011_2 = 3_{10}$**



Embedded Systems & Applications

# Weitere Beispiele Zweierkomplement

TECHNISCHE UNIVERSITÄT DARMSTADT

- Bestimme Zweierkomplement von $6_{10} = 0110_2$

  1. 1001
  2. \+ 1

  $1010_2 = -6_{10}$

- Was ist der Dezimalwert der Zweierkomplementzahl $1001_2$?

  1. 0110
  2. \+ 1

  $0111_2 = 7_{10}$, msb war vorher 1 also negativ: $1001_2 = -7_{10}$



Embedded Systems & Applications

# Addition im Zweierkomplement: es funktioniert

- Addiere 6 + (-6)

```
  111
   0110
+  1010
------
  10000
```

- Addiere -2 + 3

```
  111
   1110
+  0011
------
  10001
```

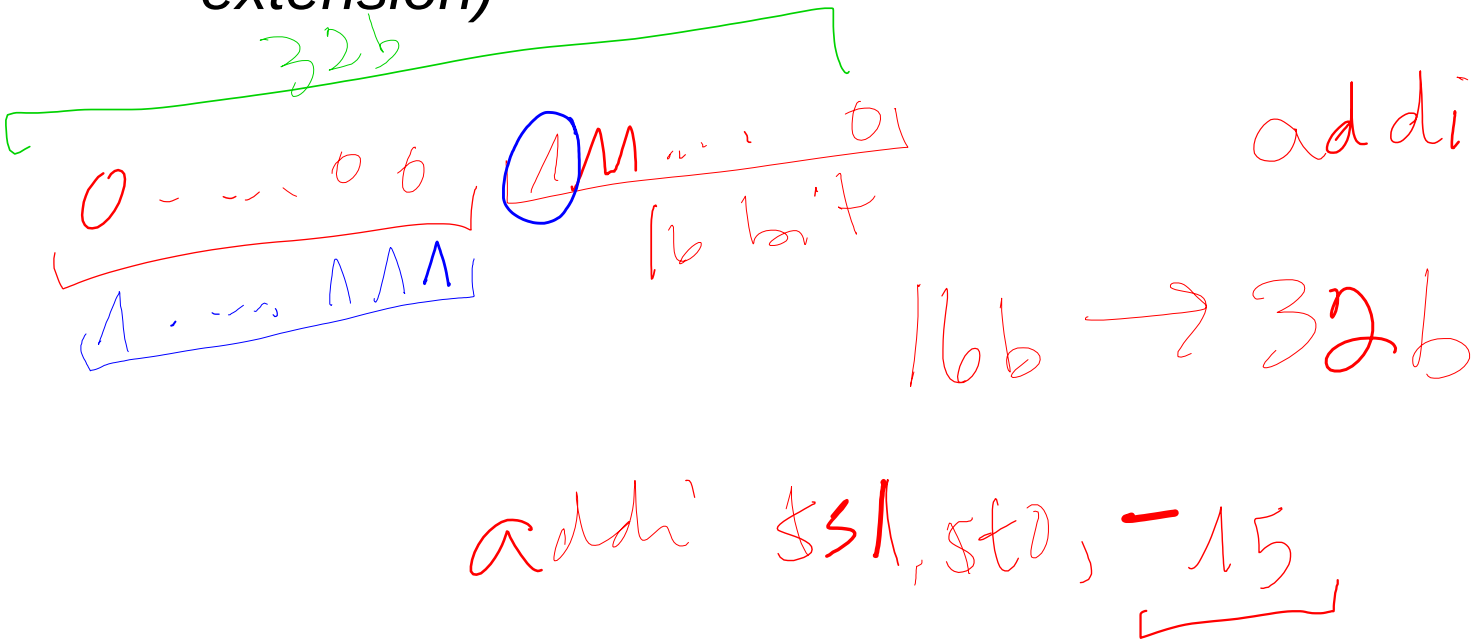
Übertrag: Ignorieren, wenn Positive und negative Zahlen gleicher Bitbreite addiert werden

# Erweitern von Zahlen auf höhere Bitbreite

TECHNISCHE UNIVERSITÄT DARMSTADT

## Zwei Möglichkeiten

- Auffüllen mit führenden Nullen (*zero extension)*
- Auffüllen mit dem bisherigen Vorzeichen (*sign extension)*






Embedded Systems & Applications

# Erweitern durch Auffüllen mit Vorzeichenbit

TECHNISCHE UNIVERSITÄT DARMSTADT

Vorzeichenbit nach links kopieren bis gewünschte Breite erreicht

- Zahlenwert bleibt unverändert - auch bei negativen Zahlen!

**Beispiel 1:**

- 4-bit Darstellung von 3 = **0**011
- 8-bit aufgefüllt durch Vorzeichen: **0000**0011

**Beispiel 2:**

- 4-bit Darstellung von -5 = **1**011
- 8-bit aufgefüllt durch Vorzeichen : **1111**1011



Embedded Systems & Applications

# Erweitern durch Auffüllen mit Nullbits

TECHNISCHE UNIVERSITÄT DARMSTADT

Nullen nach links anhängen bis gewünschte Breite erreicht

- Zerstört Wert von negativen Zahlen - positive Zahlen bleiben unverändert

**Beispiel 1:**

- 4-bit Wert = $0011 = 3_{10}$
- 8-bit durch Auffüllen mit Nullbits: 00000011 $= 3_{10}$

**Beispiel 2:**

- 4-bit Wert = $1011 = -5_{10}$
- 8-bit durch Auffüllen mit Nullbits : 00001011 = $11_{10}$, falsch!

# Befehlsformat I-Typ (Immediate Typ)

TECHNISCHE UNIVERSITÄT DARMSTADT

## 3 Operanden:

- `rs:` Quellregister
- `rt:` Quell/Zielregister
- `imm:` 16b Direktwert im Zweierkomplement

## Andere Angaben:

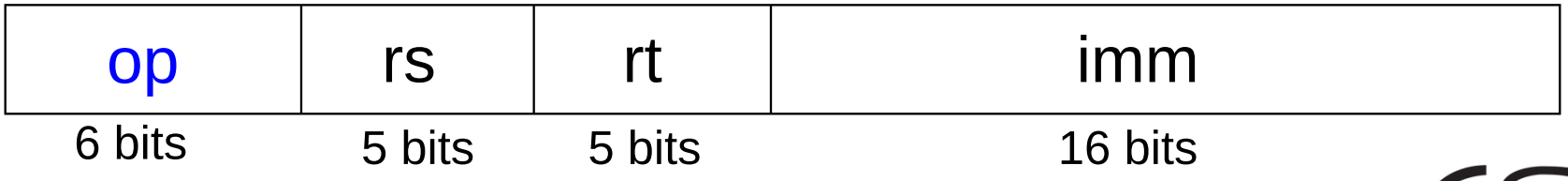
- `op:` Opcode

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






Embedded Systems & Applications

# Beispiel für Befehle vom I-Typ

TECHNISCHE UNIVERSITÄT DARMSTADT

## 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 |

## 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 | |



Embedded Systems & Applications

# Beispiel für Befehle vom I-Typ

TECHNISCHE UNIVERSITÄT DARMSTADT

## 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 | |



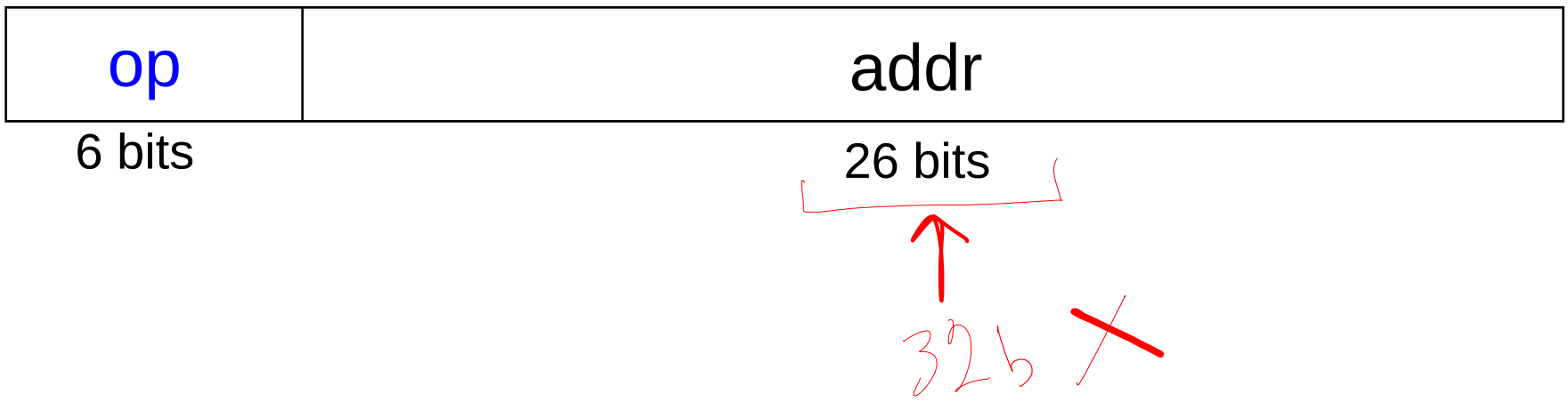
Embedded Systems & Applications

# Befehlsformat J-Typ (Jump Typ)

TECHNISCHE UNIVERSITÄT DARMSTADT

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

## J-Typ



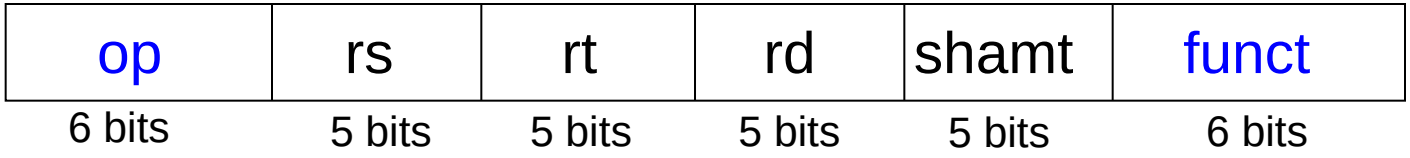



Embedded Systems & Applications

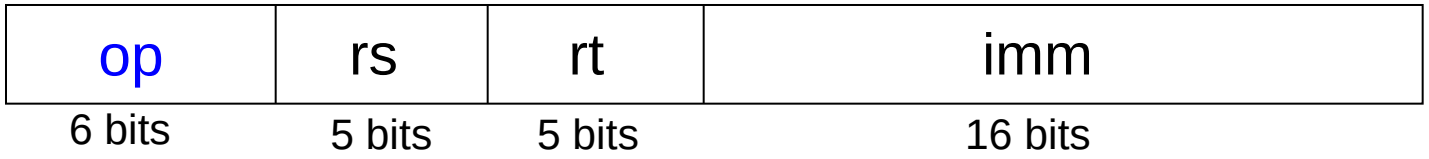
# Übersicht über Befehlsformate

TECHNISCHE UNIVERSITÄT DARMSTADT

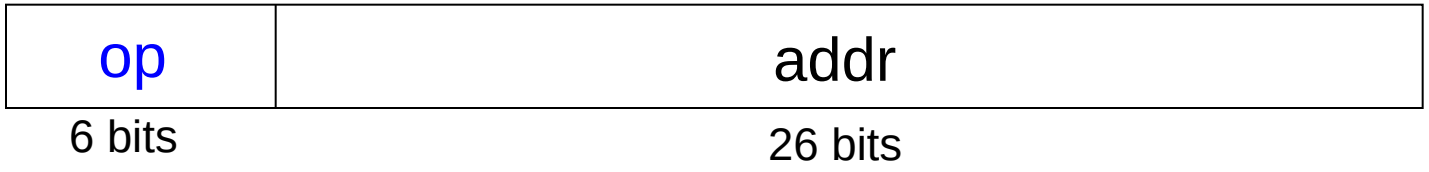
## R-Typ




## I-Typ




## J-Typ






Embedded Systems & Applications

# 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



Embedded Systems & Applications

# 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

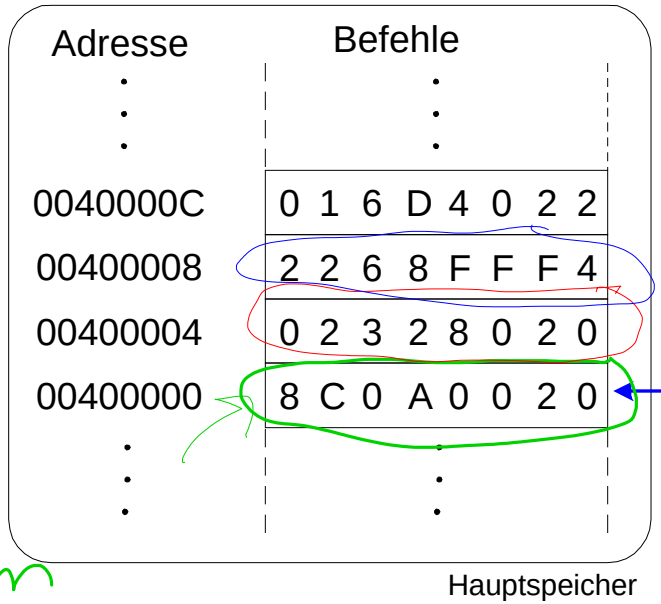


Embedded Systems & Applications

# Im Speicher abgelegtes Programm

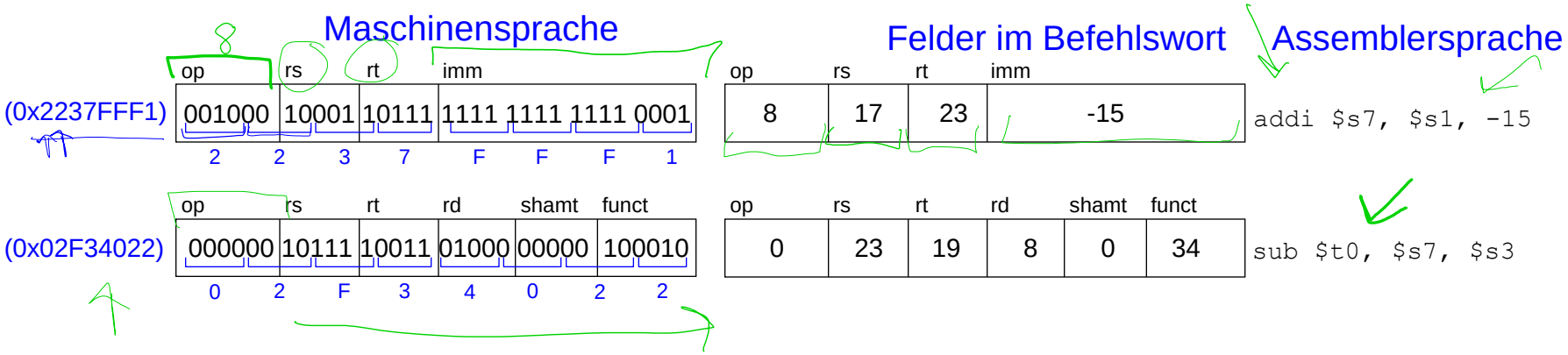
| Assemblersprache | Maschinensprache |
|---|---|
| `lw $t2, 32($0)` | `0x8C0A0020` |
| `add $s0, $s1, $s2` | `0x02328020` |
| `addi $t0, $s3, -12` | `0x2268FFF4` |
| `sub $t0, $t3, $t5` | `0x016D4022` |

Programm im Speicher



# 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

TECHNISCHE UNIVERSITÄT DARMSTADT

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



Embedded Systems & Applications

# 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



Embedded Systems & Applications

# Programmierung

TECHNISCHE UNIVERSITÄT DARMSTADT

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**

# Logische Befehle

TECHNISCHE UNIVERSITÄT DARMSTADT

## and, or, xor, nor

- `and`: nützlich zum Maskieren von Bits

  **Beispiel:** Ausmaskieren aller Bits außer dem LSB:

  0xF234012F AND 0x000000FF = 0x0000002F

- `or`: Nützlich zum Vereinigen von Bitfeldern

  **Beispiel:** Vereinige 0xF2340000 mit 0x000012BC:

  0xF2340000 OR 0x000012BC = 0xF23412BC

- `nor`: nützlich zur Invertierung von Bits:

  **Beispiel:** A NOR $0 = NOT A



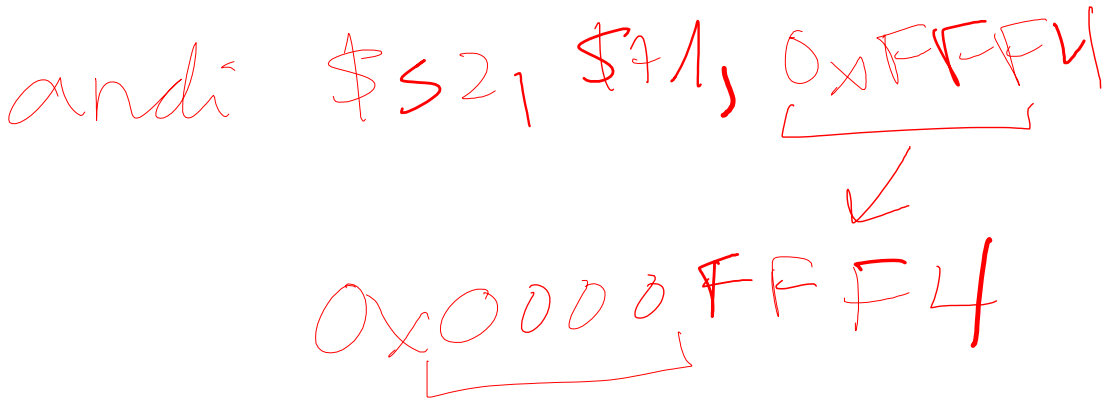
Embedded Systems & Applications

# Logische Befehle mit Konstanten

TECHNISCHE UNIVERSITÄT DARMSTADT

**`andi`, `ori`, `xori`**

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






Embedded Systems & Applications

# Beispiele: Logische Befehle

TECHNISCHE UNIVERSITÄT DARMSTADT

**Assemblersprache**

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

**Quellregister**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $s1 | 1111 | 1111 | 1111 | 1111 | 0000 | 0000 | 0000 | 0000 |
| $s2 | 0100 | 0110 | 1010 | 0001 | 1111 | 0000 | 1011 | 0111 |

**Ergebnisse**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $s3 | | | | | | | | |
| $s4 | | | | | | | | |
| $s5 | | | | | | | | |
| $s6 | | | | | | | | |



Embedded Systems & Applications

# Beispiele: Logische Befehle

TECHNISCHE UNIVERSITÄT DARMSTADT

Quellregister

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $s1 | 1111 | 1111 | 1111 | 1111 | 0000 | 0000 | 0000 | 0000 |
| $s2 | 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
```

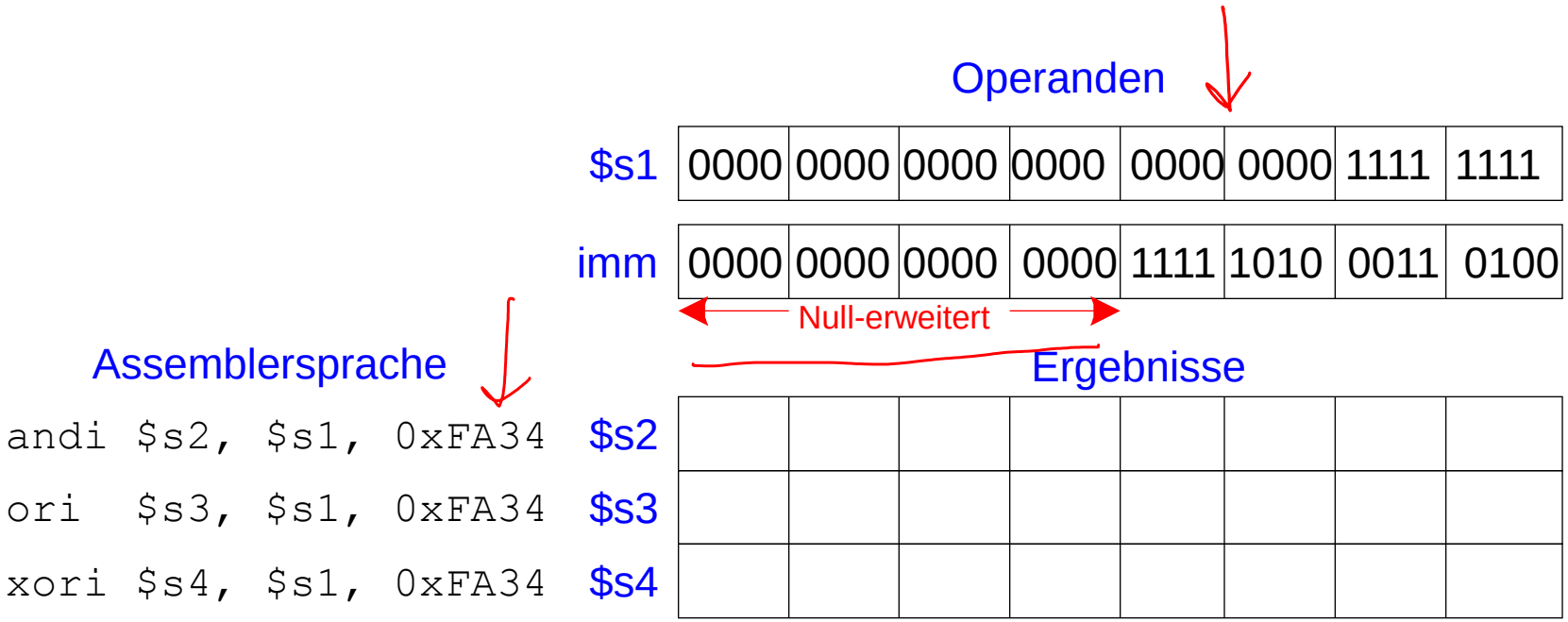
Ergebnisse

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $s3 | 0100 | 0110 | 1010 | 0001 | 0000 | 0000 | 0000 | 0000 |
| $s4 | 1111 | 1111 | 1111 | 1111 | 1111 | 0000 | 1011 | 0111 |
| $s5 | 1011 | 1001 | 0101 | 1110 | 1111 | 0000 | 1011 | 0111 |
| $s6 | 0000 | 0000 | 0000 | 0000 | 0000 | 1111 | 0100 | 1000 |



Embedded Systems & Applications

# Beispiele: Logische Befehle



# 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 | | Ergebnisse | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| andi $s2, $s1, 0xFA34 | $s2 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0011 | 0100 |
| ori $s3, $s1, 0xFA34 | $s3 | 0000 | 0000 | 0000 | 0000 | 1111 | 1010 | 1111 | 1111 |
| xori $s4, $s1, 0xFA34 | $s4 | 0000 | 0000 | 0000 | 0000 | 1111 | 1010 | 1100 | 1011 |

# Schiebebefehle

TECHNISCHE UNIVERSITÄT DARMSTADT

**sll:** shift left logical

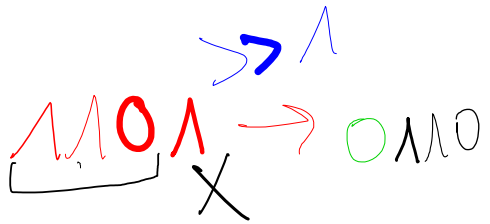
- **Beispiel:** `sll $t0, $t1, 5`

```
# $t0 <= $t1 << 5
```

**srl:** shift right logical

- **Beispiel :** `srl $t0, $t1, 5`

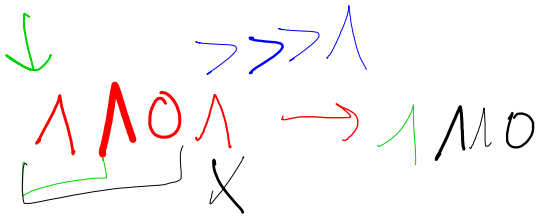
```
# $t0 <= $t1 >> 5
```



**sra:** shift right arithmetic

- **Beispiel :** `sra $t0, $t1, 5`

```
# $t0 <= $t1 >>> 5
```





Embedded Systems & Applications

# Schiebebefehle

TECHNISCHE UNIVERSITÄT DARMSTADT

**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
```



Embedded Systems & Applications

# Schiebebefehle

## Assemblersprache

```
sll $t0, $s1, 2
srl $s2, $s1, 2
sra $s3, $s1, 2
```

## Felder in Instruktion

| op | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|
| 0 | 0 | 17 | 8 | 2 | 0 |
| 0 | 0 | 17 | 18 | 2 | 2 |
| 0 | 0 | 17 | 19 | 2 | 3 |
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

## Maschinensprache

| 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

TECHNISCHE UNIVERSITÄT DARMSTADT

**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
```



Embedded Systems & Applications

# Multiplikation und Division

TECHNISCHE UNIVERSITÄT DARMSTADT

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`



Embedded Systems & Applications

# Verzweigungen und Sprünge

TECHNISCHE UNIVERSITÄT DARMSTADT

- Ändern der Ausführungsreihenfolge von Befehlen
- Arten von Verzweigungen:
  - **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

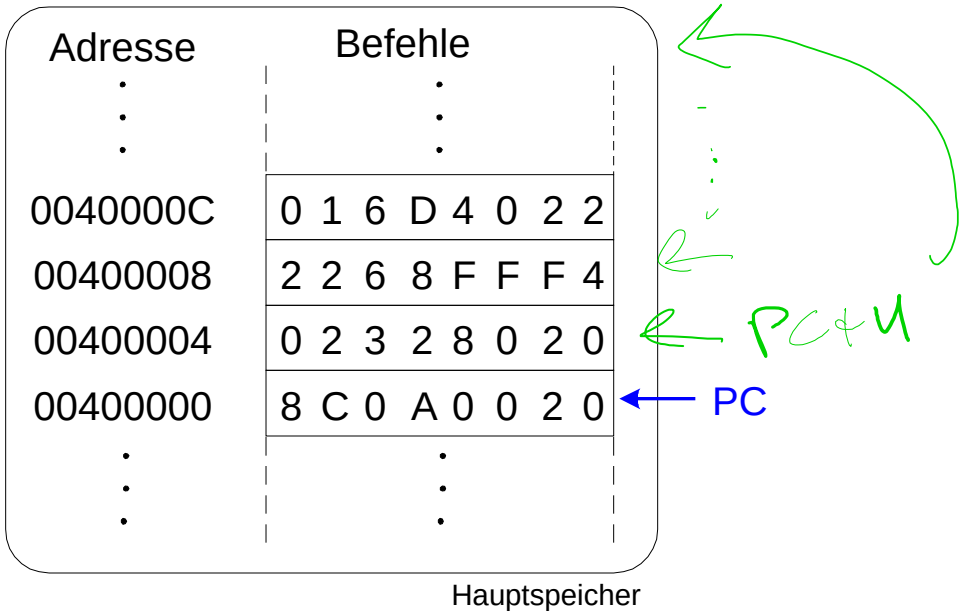


Embedded Systems & Applications

# 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



# 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.



Embedded Systems & Applications

# Nicht genommene Sprünge

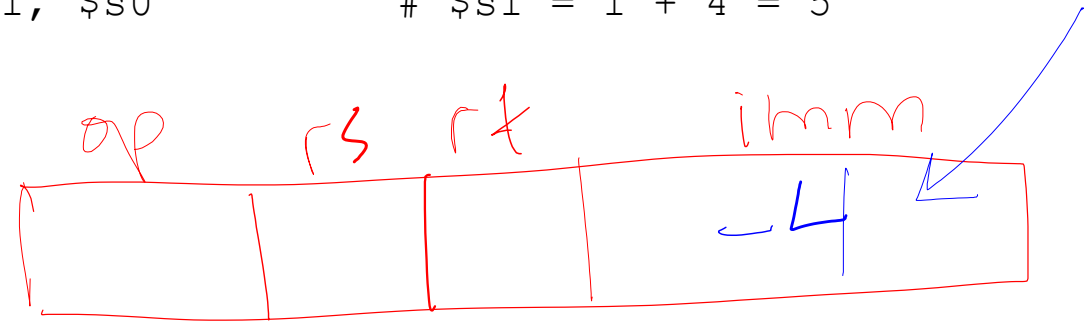
TECHNISCHE UNIVERSITÄT DARMSTADT

## # 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
    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
```

bne und beq sind **I-Typ**






Embedded Systems & Applications

# 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
```



Embedded Systems & Applications

# Unbedingte Verzweigungen (`jr`)

TECHNISCHE UNIVERSITÄT DARMSTADT

## # 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)
```

$s0 = 0x00002010



Embedded Systems & Applications

# Programmierung

TECHNISCHE UNIVERSITÄT DARMSTADT

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



Embedded Systems & Applications

# Konstrukte in Hochsprachen

TECHNISCHE UNIVERSITÄT DARMSTADT

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

# If-Anweisung

TECHNISCHE UNIVERSITÄT DARMSTADT

## Hochsprache

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

f = f - i;
```

## MIPS Assemblersprache

```
# $s0=f, $s1=g, $s2=h
# $s3=i, $s4=j
beq $s3, $s4, L1
sub $s0, $s0, $s3
j   L2
L1: add $s0, $s1, $s2
L2:
```

**Nicht** so in Assemblersprache umsetzen!



Embedded Systems & Applications

# If-Anweisung

TECHNISCHE UNIVERSITÄT DARMSTADT

## Hochsprache

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

f = f - i;
```

## MIPS Assemblersprache

```
# $s0=f, $s1=g, $s2=h
# $s3=i, $s4=j
  bne $s3, $s4, L1
  add $s0, $s1, $s2

L1: sub $s0, $s0, $s3
```

Beachte: Im Assembler wird auf **entgegengesetzte** Bedingung geprüft (`i != j`) als in der Hochsprache (`i == j`).



Embedded Systems & Applications

# If-Anweisung

TECHNISCHE UNIVERSITÄT DARMSTADT

## Hochsprache

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

else
  f = f - i;
```

## MIPS Assemblersprache

```
# $s0=f, $s1=g, $s2=h
# $s3=i, $s4=j
     bne $s3, $s4, L1
     add $s0, $s1, $s2
     j   L2
L1:  sub $s0, $s0, $s3
L2:
```

Beachte: Im Assembler wird auf **entgegengesetzte** Bedingung geprüft (`i != j`) als in der Hochsprache (`i == j`).



Embedded Systems & Applications

# If / Else-Anweisung

TECHNISCHE UNIVERSITÄT DARMSTADT

## Hochsprache

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

## MIPS Assemblersprache

```
# $s0=f, $s1=g, $s2=h
# $s3=i, $s4=j
        bne $s3, $s4, L1
        add $s0, $s1, $s2
        j   done
L1:     sub $s0, $s0, $s3
done:
```



Embedded Systems & Applications

# While-Schleife

TECHNISCHE UNIVERSITÄT DARMSTADT

## 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`).



Embedded Systems & Applications

# While-Schleife

TECHNISCHE UNIVERSITÄT DARMSTADT

## 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

      addi $s0, $0, 1
      add  $s1, $0, $0
      addi $t0, $0, 128
while: beq  $s0, $t0, done
      sll  $s0, $s0, 1
      addi $s1, $s1, 1
      j    while
done:
```


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



Embedded Systems & Applications

# For-Schleife

TECHNISCHE UNIVERSITÄT DARMSTADT

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



Embedded Systems & Applications

# For-Schleifen

TECHNISCHE UNIVERSITÄT DARMSTADT

## 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
      addi $s1, $0, 0
      add  $s0, $0, $0
      addi $t0, $0, 10
for:  beq  $s0, $t0, done
      add  $s1, $s1, $s0
      addi $s0, $s0, 1
      j    for
done:
```

Auch hier: Prüfen auf **entgegengesetzte** Bedingung in Assemblersprache (`i == 10`) als in Hochsprache (`i != 10`).



Embedded Systems & Applications

# For-Schleifen

TECHNISCHE UNIVERSITÄT DARMSTADT

## 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
    addi $s1, $0, 0
    add  $s0, $0, $0
    addi $t0, $0, 10
for: beq  $s0, $t0, done
    add  $s1, $s1, $s0
    addi $s0, $s0, 1
    j    for
done:
```

init.

Auch hier: Prüfen auf **entgegengesetzte** Bedingung in Assemblersprache (`i == 10`) als in Hochsprache (`i != 10`).



Embedded Systems & Applications

# slt: Set if Less Than

TECHNISCHE UNIVERSITÄT DARMSTADT

## Befehl `slt`: das Ergebnis (rd) auf 1 setzen, wenn rs < rt

### MIPS Assemblersprache

```
addi $t0, $0, 12   # $t0 = 12
addi $t1, $0, 15   # $t1 = 15
slt $t2, $t0, $t1  # wenn ($t0 < $t1), $t2 = 1
                   # sonst, $t2 = 0
                   # also, $t2 = 1


slt $t2, $t1, $t0  # wenn ($t1 < $t0), $t2 = 1
                   # sonst, $t2 = 0
                   # also, $t2 = 0
```



Embedded Systems & Applications

# Kleiner-als Vergleiche

TECHNISCHE UNIVERSITÄT DARMSTADT

## 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

        addi $s1, $0, 0
        addi $s0, $0, 1
        addi $t0, $0, 101
loop:   slt  $t1, $s0, $t0
        beq  $t1, $0, done
        add  $s1, $s1, $s0
        sll  $s0, $s0, 1
        j    loop
done:
```

$t1 = 1 wenn i < 101.



Embedded Systems & Applications

# Kleiner-als Vergleiche

TECHNISCHE UNIVERSITÄT DARMSTADT

## 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

        addi $s1, $0, 0
        addi $s0, $0, 1
        addi $t0, $0, 101
loop:   slt  $t1, $s0, $t0
        beq  $t1, $0, done
        add  $s1, $s1, $s0
        sll  $s0, $s0, 1
        j    loop
done:
```

$t1 = 1 wenn i < 101.



Embedded Systems & Applications

# Datenfelder (*arrays*)

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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

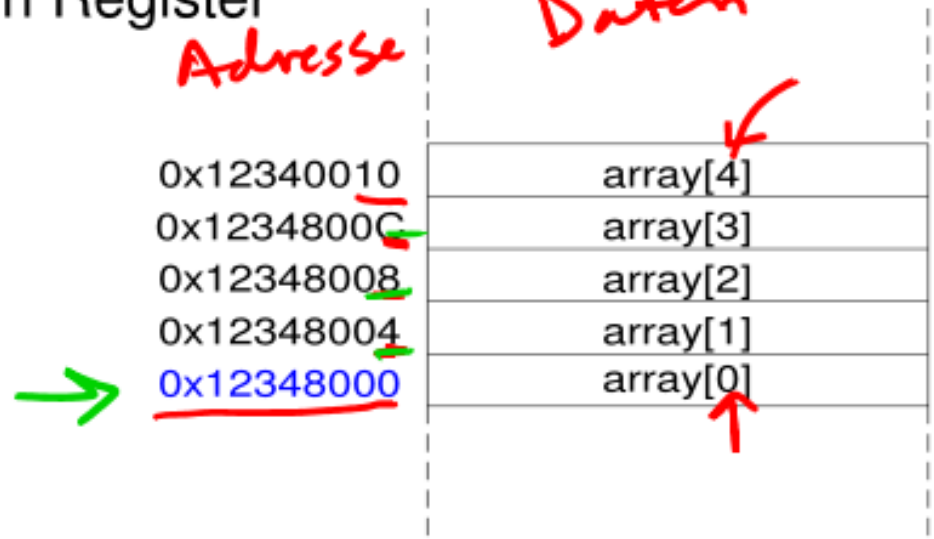


Embedded Systems & Applications

# Verwendung von Arrays

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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






Embedded Systems & Applications

# Verwendung von Arrays

TECHNISCHE UNIVERSITÄT DARMSTADT

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

# MIPS Assemblersprache
# Basisadresse von array = $s0

  lui  $s0, 0x1234          # lade 0x1234 in obere Hälfte von $S0
  ori  $s0, $s0, 0x8000     # lade 0x8000 in untere Hälfte von $s0

  lw   $t1, 0($s0)          # $t1 = array[0]
  sll  $t1, $t1, 1          # $t1 = $t1 * 2
  sw   $t1, 0($s0)          # array[0] = $t1
```



Embedded Systems & Applications

# Verwendung von Arrays

TECHNISCHE UNIVERSITÄT DARMSTADT

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

# MIPS Assemblersprache
# Basisadresse von array = $s0

   lui  $s0, 0x1234          # lade 0x1234 in obere Hälfte von $S0
   ori  $s0, $s0, 0x8000     # lade 0x8000 in untere Hälfte von $s0

   lw   $t1, 0($s0)          # $t1 = array[0]
   sll  $t1, $t1, 1          # $t1 = $t1 * 2
   sw   $t1, 0($s0)          # array[0] = $t1

   lw   $t1, 4($s0)          # $t1 = array[1]
   sll  $t1, $t1, 1          # $t1 = $t1 * 2
   sw   $t1, 4($s0)          # array[1] = $t1
```



Embedded Systems & Applications

# 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

# Initialisierung
  lui  $s0, 0x23B8        # $s0 = 0x23B80000
  ori  $s0, $s0, 0xF000   # $s0 = 0x23B8F000
  addi $s1, $0, 0         # i = 0
  addi $t2, $0, 1000      # $t2 = 1000
loop:
  slt  $t0, $s1, $t2      # i < 1000?
  beq  $t0, $0, done      # if not then done
  sll  $t0, $s1, 2        # $t0 = i * 4 (byte offset)
  add  $t0, $t0, $s0      # address of array[i]
  lw   $t1, 0($t0)        # $t1 = array[i]
  ...


done:
```



Embedded Systems & Applications

# 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

# Initialisierung
  lui  $s0, 0x23B8        # $s0 = 0x23B80000
  ori  $s0, $s0, 0xF000   # $s0 = 0x23B8F000
  addi $s1, $0, 0         # i = 0
  addi $t2, $0, 1000      # $t2 = 1000
loop:
  slt  $t0, $s1, $t2      # i < 1000?
  beq  $t0, $0, done      # if not then done
  sll  $t0, $s1, 2        # $t0 = i * 4 (byte offset)
  add  $t0, $t0, $s0      # address of array[i]
  lw   $t1, 0($t0)        # $t1 = array[i]
  sll  $t1, $t1, 3        # $t1 = array[i] * 8
  sw   $t1, 0($t0)        # array[i] = array[i] * 8
  addi $s1, $s1, 1        # i = i + 1
  j    loop               # repeat
done:
```



Embedded Systems & Applications

# 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

# Initialisierung
   lui  $s0, 0x23B8          # $s0 = 0x23B80000
   ori  $s0, $s0, 0xF000     # $s0 = 0x23B8F000
   addi $s1, $0, 0           # i = 0
   addi $t2, $0, 1000        # $t2 = 1000
loop:
   slt  $t0, $s1, $t2        # i < 1000?
   beq  $t0, $0, done        # if not then done
   sll  $t0, $s1, 2          # $t0 = i * 4 (byte offset)
   add  $t0, $t0, $s0        # address of array[i]
   lw   $t1, 0($t0)          # $t1 = array[i]
   sll  $t1, $t1, 3          # $t1 = array[i] * 8
   sw   $t1, 0($t0)          # array[i] = array[i] * 8
   addi $s1, $s1, 1          # i = i + 1
   j    loop                 # repeat
done:
```



Embedded Systems & Applications

# Zeichendarstellung im ASCII-Code

TECHNISCHE UNIVERSITÄT DARMSTADT

***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

0x 20

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

# Zuordnung von Zeichen zu Codes

TECHNISCHE UNIVERSITÄT DARMSTADT

| # | 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 | O | 5F | _ | 6F | o | | |



Embedded Systems & Applications

# Programmierung

TECHNISCHE UNIVERSITÄT DARMSTADT

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

# Prozedur- und Funktionsaufruf

TECHNISCHE UNIVERSITÄT DARMSTADT

## 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);
}
```

# Prozedur- und Funktionsaufruf

TECHNISCHE UNIVERSITÄT DARMSTADT

**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



Embedded Systems & Applications

# 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`

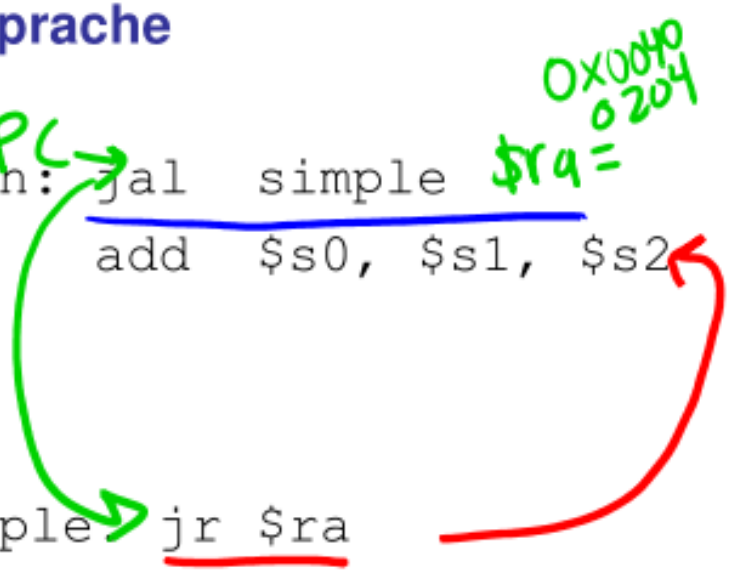


Embedded Systems & Applications

# Prozedur- und Funktionsaufruf

TECHNISCHE UNIVERSITÄT DARMSTADT

| Hochsprache | MIPS Assemblersprache |
|---|---|
| `int main() {`<br>`  simple();`<br>`  a = b + c;`<br>`}` | **0x00400200** main: `jal simple`<br>**0x00400204** `add $s0, $s1, $s2`<br>`...` |
| `void simple() {`<br>`  return;`<br>`}` | **0x00401020** simple: `jr $ra` |




`void` bedeutet, dass `simple` keinen Rückgabewert hat.

- Also eine Prozedur und keine Funktion ist



Embedded Systems & Applications

# 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.



Embedded Systems & Applications

# Aufrufargumente und Rückgabewert

TECHNISCHE UNIVERSITÄT DARMSTADT

## MIPS Konventionen:

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



Embedded Systems & Applications

# Aufrufargumente und Rückgabewert



## Hochsprache

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

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

$a0 $a1 $a2 $a3

# Aufrufargumente und Rückgabewert

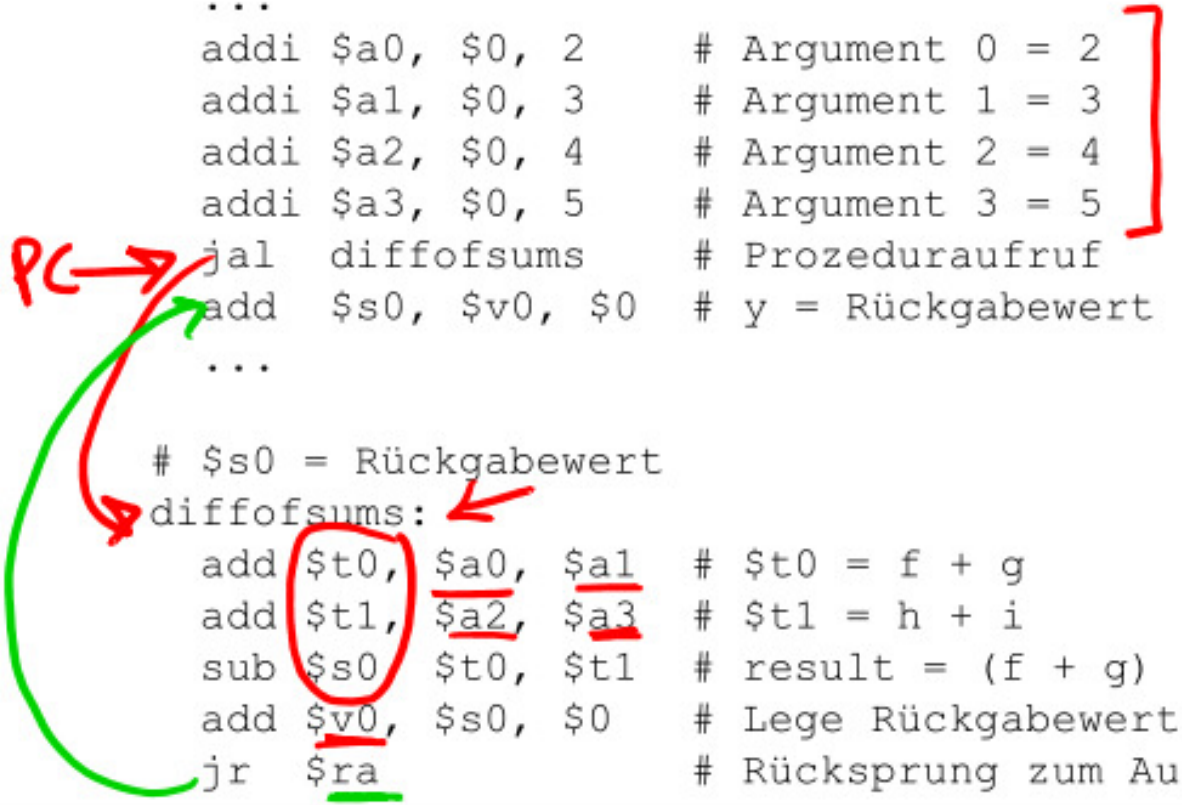
TECHNISCHE UNIVERSITÄT DARMSTADT

**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
```






Embedded Systems & Applications

# Aufrufargumente und Rückgabewert

TECHNISCHE UNIVERSITÄT DARMSTADT

## 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



Embedded Systems & Applications

# Stack (auch Stapel- oder Kellerspeicher)

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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



Embedded Systems & Applications

# Stack

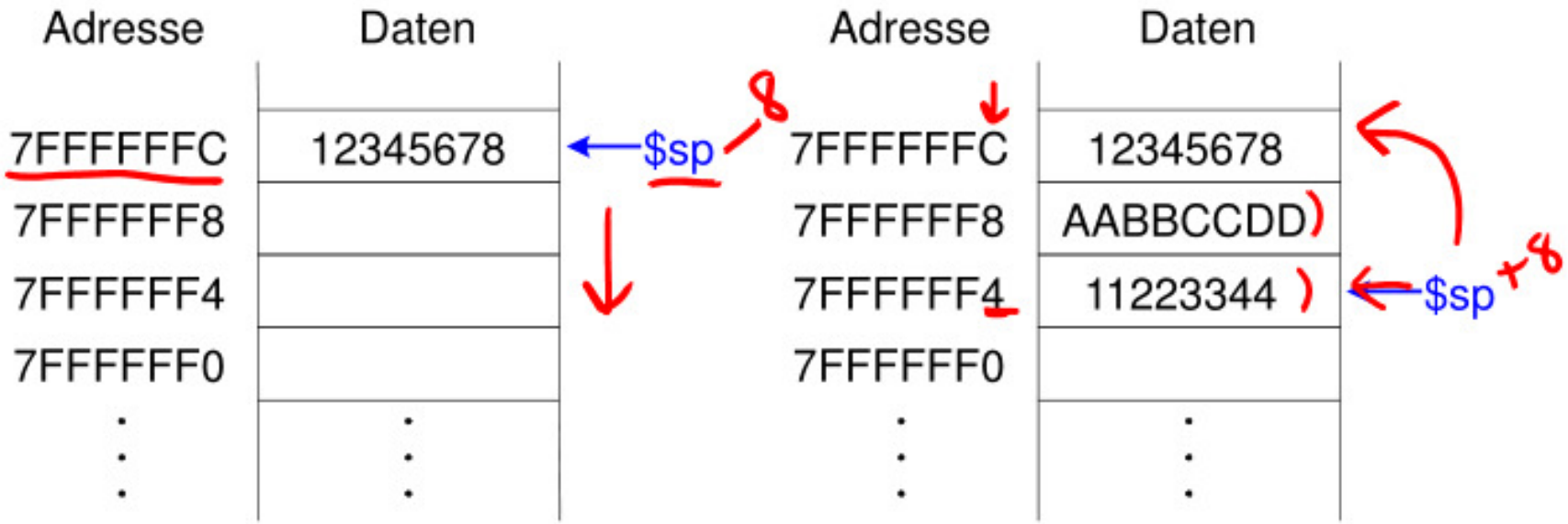
TECHNISCHE UNIVERSITÄT DARMSTADT

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






Embedded Systems & Applications

# Verwendung des Stacks in Prozeduren

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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
```

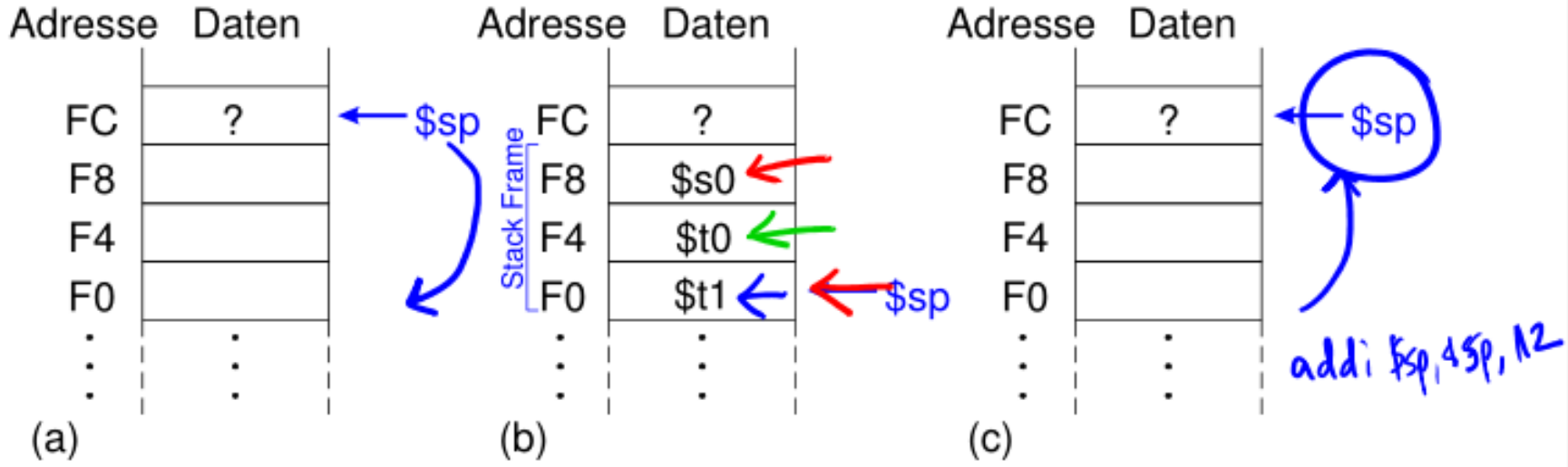


Embedded Systems & Applications

# Register auf Stack zwischenspeichern

TECHNISCHE UNIVERSITÄT DARMSTADT

```
# $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
```






Embedded Systems & Applications

# Register auf Stack zwischenspeichern

TECHNISCHE UNIVERSITÄT DARMSTADT

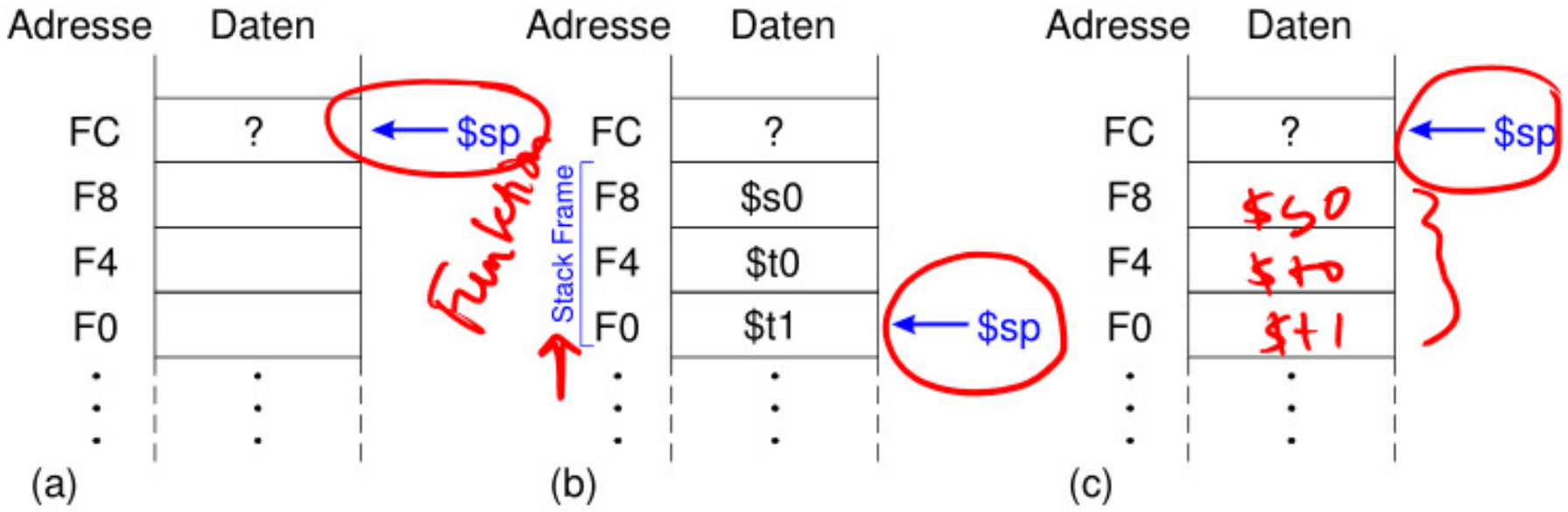
```
# $s0 = result
diffofsums:
  addi $sp, $sp, -12  # 3*4 Bytes auf Stack anfordern
                      # um drei 32b Register zu sichern
  sw   $s0, 8($sp)    # speichere $s0 auf Stack
  sw   $t0, 4($sp)    # speichere $t0 auf Stack
  sw   $t1, 0($sp)    # speichere $t1 auf Stack
  add  $t0, $a0, $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
```



Embedded Systems & Applications

# Veränderung des Stacks während

`diffofsums`



# Sicherungskonventionen für Register

TECHNISCHE UNIVERSITÄT DARMSTADT

| Erhalten<br>*Gesichert vom Aufgerufenen* | Nicht erhalten<br>*Gesichert vom Aufrufer* |
|---|---|
| `$s0 - $s7` | `$t0 - $t9` |
| `$ra` | `$a0 - $a3` |
| `$sp` | `$v0 - $v1` |
| Stack oberhalb von `$sp` | Stack unterhalb von `$sp` |



Embedded Systems & Applications

# Register auf Stack zwischenspeichern

TECHNISCHE UNIVERSITÄT DARMSTADT

```
# $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
```



Embedded Systems & Applications

# Erhalten von Registern mittels Stack

TECHNISCHE UNIVERSITÄT DARMSTADT

```
# $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
```



Embedded Systems & Applications

# Sicherungskonventionen für Register

TECHNISCHE UNIVERSITÄT DARMSTADT

| Erhalten<br>*Gesichert vom Aufgerufenen* | Nicht erhalten<br>*Gesichert vom Aufrufer* |
|---|---|
| `$s0 – $s7` | `$t0 – $t9` |
| `$ra` | `$a0 – $a3` |
| `$sp` | `$v0 – $v1` |
| Stack oberhalb von `$sp` | Stack unterhalb von `$sp` |



Embedded Systems & Applications

# Mehrfache Prozeduraufrufe: Sichern von $ra

TECHNISCHE UNIVERSITÄT DARMSTADT


```
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:
   ...
   jr   $ra
```

PC+4

Blattprozedur



Embedded Systems & Applications

# Rekursive Prozeduraufrufe

TECHNISCHE UNIVERSITÄT DARMSTADT

## Hochsprache

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

## MIPS Assemblersprache

```
fakultaet:
  addi $t0, $0, 2
  slt  $t0, $a0, $t0
  beq  $t0, $0,else
  addi $v0, $0, 1
  jr   $ra

else:
  addi $a0, $a0, -1  # n = n-1
  jal  fakultaet
  mul $v0, $a0, $v0
  jr   $ra
```

# Rekursive Prozeduraufrufe

TECHNISCHE UNIVERSITÄT DARMSTADT

## Hochsprache

```
int fakultaet (int n) {


  if (n <= 1)
    return 1;



  else

    return (n * fakultaet(n-1));
}
```

(3 & 2)

## MIPS Assemblersprache

```
fakultaet:


  addi $t0, $0, 2
  slt  $t0, $a0, $t0 # n <= 1 ?
  beq  $t0, $0, else # nein: weiter bei else
  addi $v0, $0, 1    # ja: gebe 1 zurück

  jr   $ra           # Rücksprung
else:
  addi $a0, $a0, -1  # n = n - 1
  jal  fakultaet     # rekursiver Aufruf


  mul  $v0, $a0, $v0 # n * fakultaet(n-1)

  jr   $ra           # Rücksprung
```



Embedded Systems & Applications

# Rekursive Prozeduraufrufe

TECHNISCHE UNIVERSITÄT DARMSTADT

## Hochsprache

```
int fakultaet (int n) {


  if (n <= 1)
    return 1;



  else

    return (n * fakultaet(n-1));
}
```

## MIPS Assemblersprache

```
fakultaet:
  addi $sp, $sp, -8   # Platz für zwei Register
  sw   $a0, 4($sp)    # sichere $a0
  sw   $ra, 0($sp)    # sichere $ra
  addi $t0, $0, 2
  slt  $t0, $a0, $t0 # n <= 1 ?
  beq  $t0, $0, else # nein: weiter bei else
  addi $v0, $0, 1     # ja: gebe 1 zurück
  addi $sp, $sp, 8    # Platz wieder freigeben
  jr   $ra            # Rücksprung
else:
  addi $a0, $a0, -1   # n = n - 1
  jal  fakultaet      # rekursiver Aufruf
  lw   $ra, 0($sp)    # wiederherstellen von $ra
  lw   $a0, 4($sp)    # wiederherstellen von $a0
  mul  $v0, $a0, $v0 # n * fakultaet(n-1)
  addi $sp, $sp, 8    # Platz wieder freigeben
  jr   $ra            # Rücksprung
```



Embedded Systems & Applications

# Rekursive Prozeduraufrufe

TECHNISCHE UNIVERSITÄT DARMSTADT

## MIPS Assemblersprache

```
0x90 fakultaet: addi $sp, $sp, -8   # Platz für zwei Register
0x94            sw   $a0, 4($sp)    # sichere $a0
0x98            sw   $ra, 0($sp)    # sichere $ra
0x9C            addi $t0, $0, 2
0xA0            slt  $t0, $a0, $t0  # n <= 1 ?
0xA4            beq  $t0, $0, else  # nein: weiter bei else
0xA8            addi $v0, $0, 1     # ja: gebe 1 zurück
0xAC            addi $sp, $sp, 8    # Platz wieder freigeben
0xB0            jr   $ra            # Rücksprung
0xB4       else: addi $a0, $a0, -1  # n = n - 1
0xB8            jal  fakultaet      # rekursiver Aufruf
0xBC            lw   $ra, 0($sp)    # wiederherstellen von $ra
0xC0            lw   $a0, 4($sp)    # wiederherstellen von $a0
0xC4            mul  $v0, $a0, $v0  # n * fakultaet(n-1)
0xC8            addi $sp, $sp, 8    # Platz wieder freigeben
0xCC            jr   $ra            # Rücksprung
```

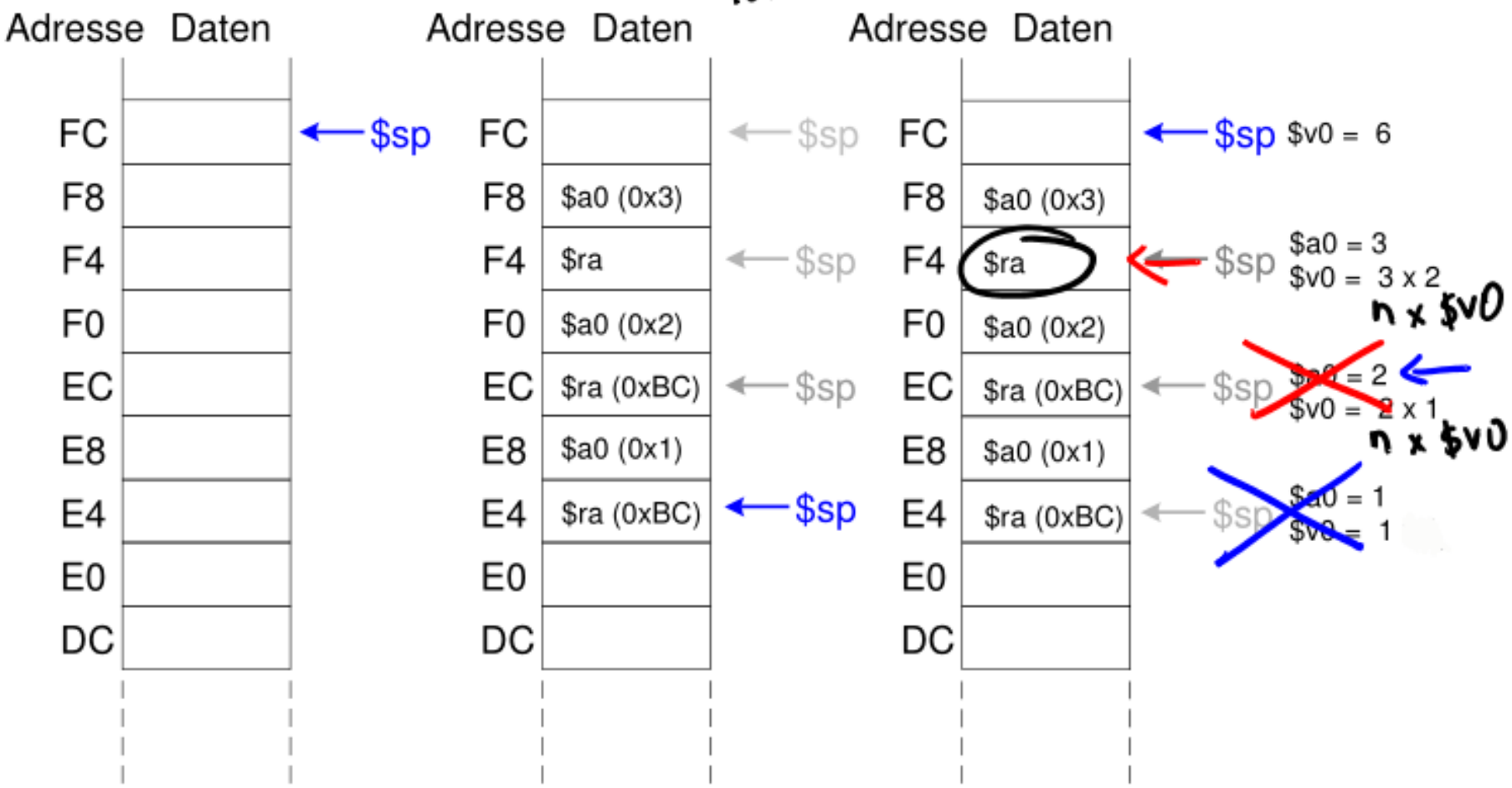


Embedded Systems & Applications

# Veränderung des Stacks bei rekursivem Aufruf

TECHNISCHE UNIVERSITÄT DARMSTADT

fakultaet(3)






Embedded Systems & Applications

# Zusammenfassung: Prozeduraufruf

TECHNISCHE UNIVERSITÄT DARMSTADT

## 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 aufgerufener`
- 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`



Embedded Systems & Applications

# Adressierungsarten

TECHNISCHE UNIVERSITÄT DARMSTADT

## Wo kommen Operanden für Befehle her?

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

lw, sw
beq



Embedded Systems & Applications

# Adressierungsarten

TECHNISCHE UNIVERSITÄT DARMSTADT

## 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`

2er komplement
null-erweitert



Embedded Systems & Applications

# Adressierungsarten

TECHNISCHE UNIVERSITÄT DARMSTADT

## 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`



Embedded Systems & Applications

# Adressierungsarten

TECHNISCHE UNIVERSITÄT DARMSTADT

## 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
```

Branch Target Address = BTA = PC+4 + (imm << 2)

Assemblersprache (Zieladresse) Bitfelder in Instruktion

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


0x14 + (3 x 4)

0x14 + 0xC

0x20 ✓



Embedded Systems & Applications

# Adressierungsarten

TECHNISCHE UNIVERSITÄT DARMSTADT

(Zieladresse)

Jump Target Address = JTA = {PC+4$_{31:28}$, addr, 2'b0}

## 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

| op | imm |
|---|---|
| 3 | 0x0100028 |
| 6 bits | 26 bits |

Maschinencode

| op | addr | |
|---|---|---|
| 000011 | 00 0001 0000 0000 0000 0010 1000 | (0x0C100028) |
| 6 bits | 26 bits | |



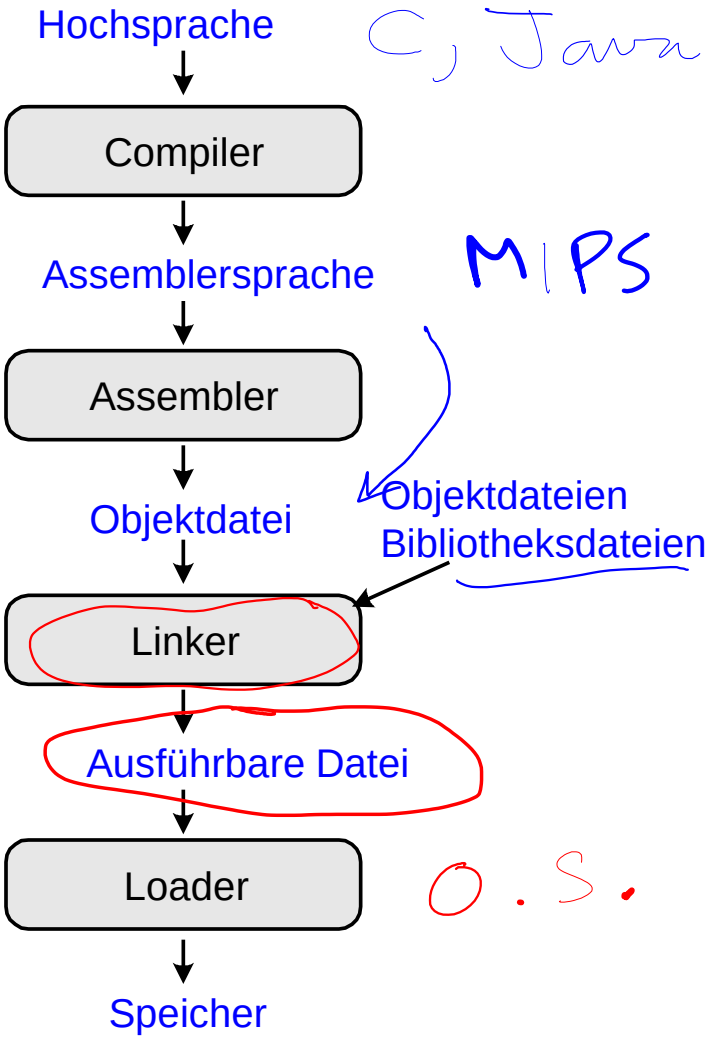
Embedded Systems & Applications

# Überblick der heutigen Themen

TECHNISCHE UNIVERSITÄT DARMSTADT

- von Programm bis zum Ausführen
- Dies und Das
  - Pseudobefehle
  - Ausnahmebehandlung (*exceptions*)
  - Befehle für vorzeichenbehaftete und vorzeichenlose Zahlen
  - Gleitkommabefehle

# 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

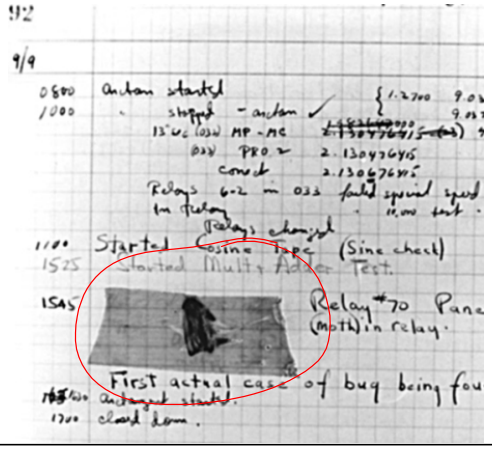


Embedded Systems & Applications

# 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






Embedded Systems & Applications

# Was muss im Speicher abgelegt werden?

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Instruktionen** (historisch auch genannt *Text*)
- **Daten**
  - Globale und statische: angelegt vor Beginn der Programmausführung
  - Dynamisch: während der Programmausführung angelegt

Stack

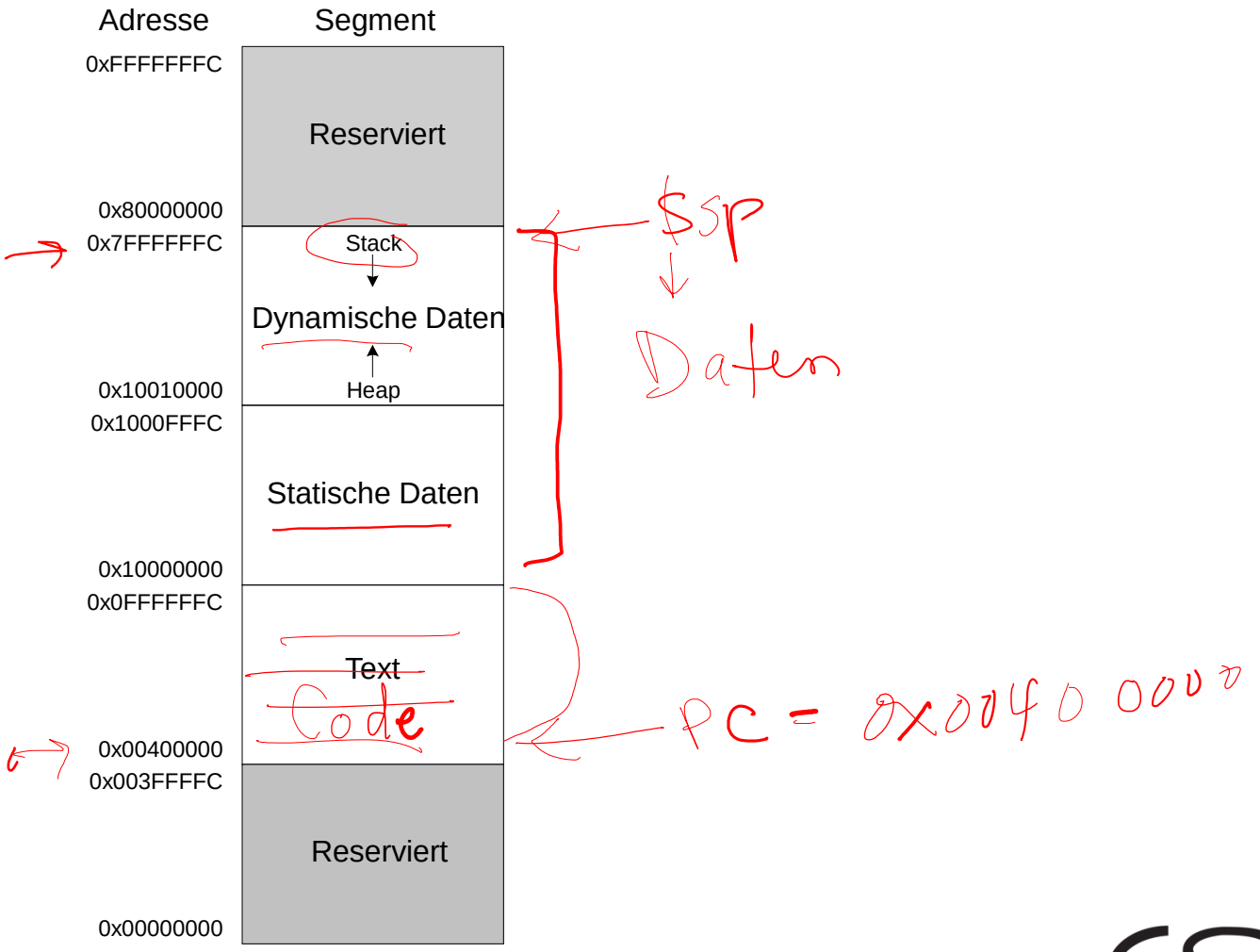
## Speicherobergrenze bei MIPS (32b Adressen)?

- Maximal $2^{32}$ = 4 Gigabytes (4 GB)
- Von Adresse 0x00000000 bis 0xFFFFFFFF



Embedded Systems & Applications

# 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);
}
```



Embedded Systems & Applications

# 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
```



Embedded Systems & Applications

# Beispielprogramm: Symboltabelle

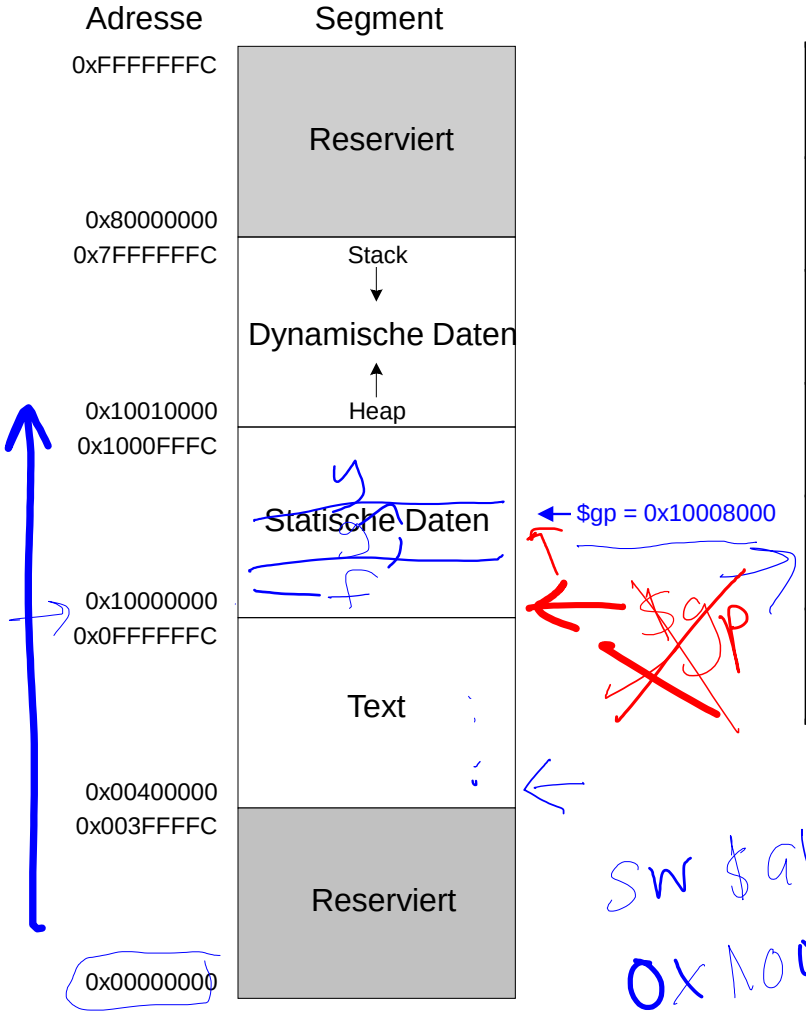
TECHNISCHE UNIVERSITÄT DARMSTADT

| Symbol | Adresse |
|---|---|
| f | |
| g | |
| y | |
| main | |
| sum | |



Embedded Systems & Applications

# Beispielprogramm: Symboltabelle



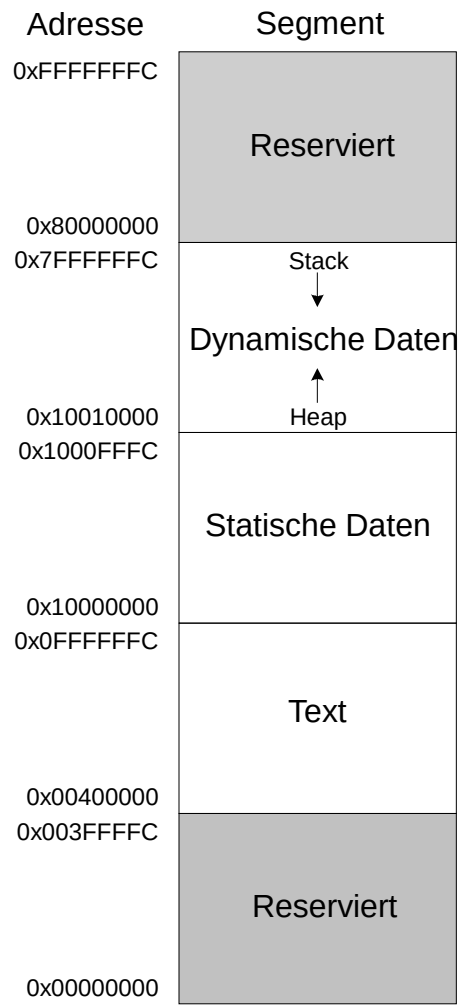

| Symbol | Adresse |
|---|---|
| f | 0x10000000 |
| g | 0x10000004 |
| y | 0x10000008 |
| main | 0x00400000 |
| sum | 0x0040002C |

# Beispielprogramm: Symboltabelle

TECHNISCHE UNIVERSITÄT DARMSTADT

| Symbol | Adresse |
|---|---|
| `f` | 0x10000000 |
| `g` | 0x10000004 |
| `y` | 0x10000008 |
| `main` | 0x00400000 |
| `sum` | 0x0040002C |

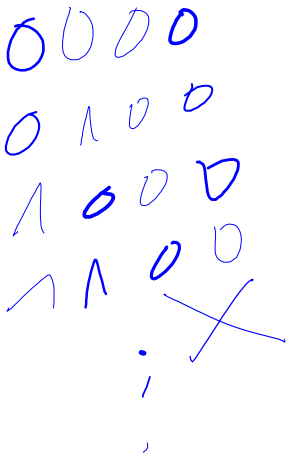
# Beispielprogramm: MIPS Assemblersprache




$gp = 0x10008000

```
.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   $a0, 0x8000($gp)  # f = 2
  addi $a1, $0, 3        # $a1 = 3
  sw   $a1, 0x8004($gp)  # g = 3
  jal  sum               # Aufruf von sum
  sw   $v0, 0x8008($gp)  # 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: MIPS Assemblersprache



```
                    .data
                    f: .space 4              # Direktiven für Assembler
                    g: .space 4              # jeweils ein Wort, initialisiert
                    y: .space 4              # auf den Wert 0
                    .text
                    main:
0x00400000            addi $sp, $sp, -4      # Stack Frame anlegen
0x00400004            sw   $ra, 0($sp)       # sichere $ra
0x00400008            addi $a0, $0, 2        # $a0 = 2
0x0040000c            sw   $a0, 0x8000($gp)  # f = 2
0x00400010            addi $a1, $0, 3        # $a1 = 3
0x00400014            sw   $a1, 0x8004($gp)  # g = 3
0x00400018            jal  sum               # Aufruf von sum
0x0040001c            sw   $v0, 0x8008($gp)  # y = sum()
0x00400020            lw   $ra, 0($sp)       # stelle $ra wieder her
0x00400024            addi $sp, $sp, 4       # stelle $sp wieder her
0x00400028            jr   $ra               # Rückkehr ins Betriebssystem
0x0040002c          sum:
0x00400030            add  $v0, $a0, $a1     # $v0 = a + b
0x00400034            jr   $ra               # return
```

32b Sprungzieladresse → **0000 0000 0100 0000 0000 0000 0010 1100**

26b Feld in jal-Instruktion → **0000 0000 0100 0000 0000 0000 0010 1100**

PC+4 → 0 1 0 0 0 0 B

# Beispielprogramm: Ausführbare Datei

TECHNISCHE UNIVERSITÄT DARMSTADT

| Dateikopf | Text Größe | Daten Größe |
|---|---|---|
| | 0x34 (52 bytes) | 0xC (12 bytes) |

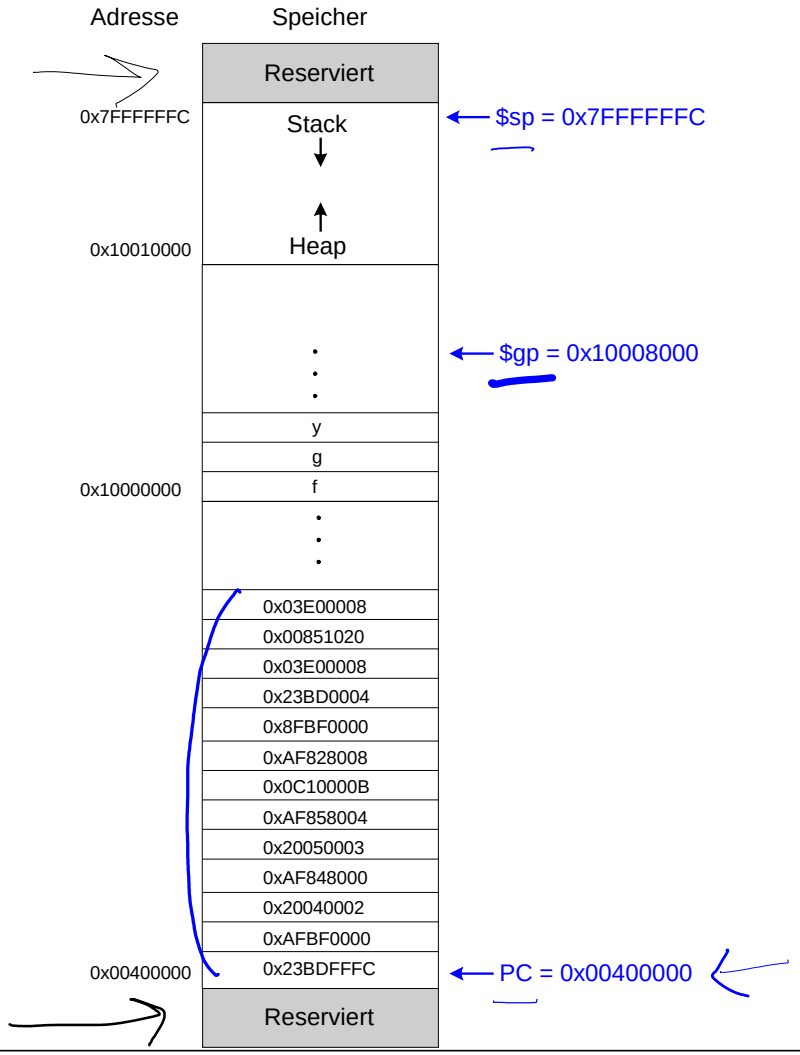
| Textsegment | Adresse | Instruktion | |
|---|---|---|---|
| | 0x00400000 | 0x23BDFFFC | addi $sp, $sp, -4 |
| | 0x00400004 | 0xAFBF0000 | sw $ra, 0 ($sp) |
| | 0x00400008 | 0x20040002 | addi $a0, $0, 2 |
| | 0x0040000C | 0xAF848000 | sw $a0, 0x8000 ($gp) |
| | 0x00400010 | 0x20050003 | addi $a1, $0, 3 |
| | 0x00400014 | 0xAF858004 | sw $a1, 0x8004 ($gp) |
| | 0x00400018 | 0x0C10000B | jal 0x0040002C |
| | 0x0040001C | 0xAF828008 | sw $v0, 0x8008 ($gp) |
| | 0x00400020 | 0x8FBF0000 | lw $ra, 0 ($sp) |
| | 0x00400024 | 0x23BD0004 | addi $sp, $sp, -4 |
| | 0x00400028 | 0x03E00008 | jr $ra |
| | 0x0040002C | 0x00851020 | add $v0, $a0, $a1 |
| | 0x00400030 | 0x03E00008 | jr $ra |

| Datensegment | Adresse | Datum | |
|---|---|---|---|
| | 0x10000000 | 0 | f |
| | 0x10000004 | 0 | g |
| | 0x10000008 | 0 | y |



Embedded Systems & Applications

# Beispielprogramm im Speicher



# Dies und Das

TECHNISCHE UNIVERSITÄT DARMSTADT

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

# Beispiele für Pseudobefehle

TECHNISCHE UNIVERSITÄT DARMSTADT

| Pseudobefehle | MIPS Befehle |
|---|---|
| `li $s0, 0x1234AA77` | `lui $s0, 0x1234`<br>`ori $s0, 0xAA77` |
| `move $s1, $s2` | `addu $s1, $0, $s2` |
| `nop` | `sll $0, $0, 0` |
| `not $t1, $s7` | `nor $t1, $s7, $0` |

`$at` ist dem Assembler zur Verfügung



Embedded Systems & Applications

# Ausnahmebehandlung (*exceptions*)

TECHNISCHE UNIVERSITÄT DARMSTADT

**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**

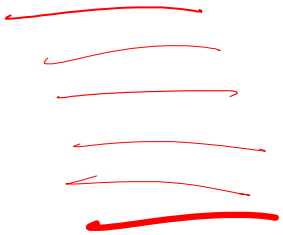
- Hardware, dann genannt Interrupt (z.B. Tippen einer Taste auf Tastatur)
- Software, dann genannt Trap (z.B. Versuch der Ausführung einer unbekannten Instruktion)

# Ausnahmebehandlung (*exceptions*)

TECHNISCHE UNIVERSITÄT DARMSTADT

Beim Auftreten der Ausnahme

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





Embedded Systems & Applications

# Spezialregister für Ausnahmebehandlung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Außerhalb des regulären Registerfeldes
  - **Cause**
    - Enthält den Grund für Ausnahme
  - **EPC** (Exception PC)
    - Enthält den regulären PC +4 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

TECHNISCHE UNIVERSITÄT DARMSTADT

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



Embedded Systems & Applications

# Ausnahmen

TECHNISCHE UNIVERSITÄT DARMSTADT

**Wenn eine Ausnahme passiert:**

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

**Ausnahmebehandlung**:

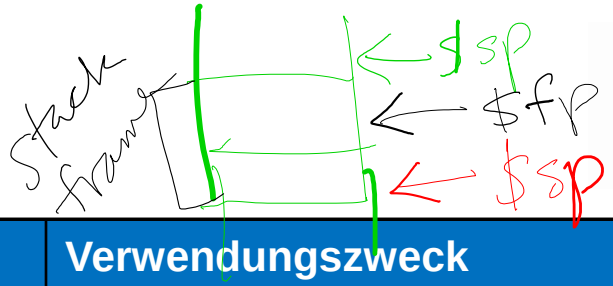
1. Speichere Register auf Stack
2. Lese `Cause` Spezialregister: `mfc0 $t0, Cause`
3. Bearbeite Ausnahme
4. Stelle alle Register wieder her
5. Springe zurück ins eigentlich laufende Programm

```
mfc0 $k0, EPC
jr $k0
```



Embedded Systems & Applications

# MIPS Registerfeld




TECHNISCHE UNIVERSITÄT DARMSTADT

| 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 |



Embedded Systems & Applications

# Vorzeichenbehaftete und –lose Befehle

TECHNISCHE UNIVERSITÄT DARMSTADT

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



Embedded Systems & Applications

# Addition und Subtraktion

TECHNISCHE UNIVERSITÄT DARMSTADT

**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




Embedded Systems & Applications

# Multiplikation und Division

TECHNISCHE UNIVERSITÄT DARMSTADT

**Vorzeichenbehaftet:** `mult`, `div`

**Vorzeichenlos:** `multu`, `divu`



Embedded Systems & Applications

# Set Less Than

TECHNISCHE UNIVERSITÄT DARMSTADT

**Vorzeichenbehaftet:** `slt`, `slti`

**Vorzeichenlos:** `sltu`, `sltiu`

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



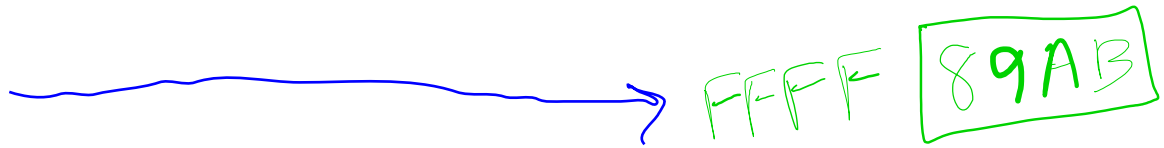
Embedded Systems & Applications

# Laden von 8b und 16b breiten Daten

TECHNISCHE UNIVERSITÄT DARMSTADT

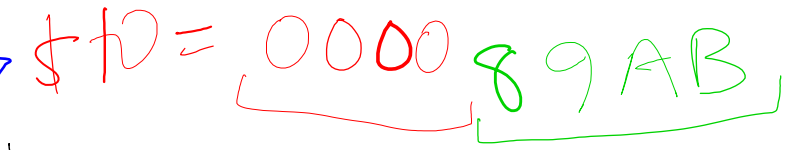
## Vorzeichenbehaftet:

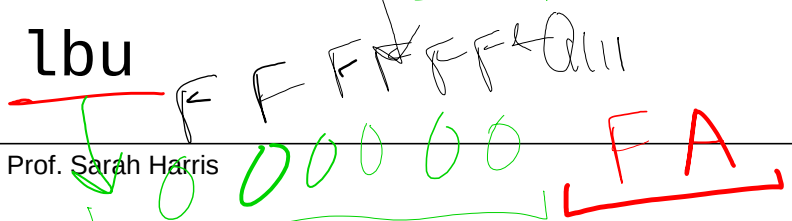
- Vorzeichenerweitere schmale Daten auf volle 32b Registerbreite
- Load halfword: `lh`
- Load byte: `lb`




## Vorzeichenlos:

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






Embedded Systems & Applications

# Zahlensysteme

TECHNISCHE UNIVERSITÄT DARMSTADT

## Im Digitaltechnik kennengelernt:

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

## Wo bleiben Brüche?

- Rationale Zahlen?
- Reelle Zahlen?

# Zahlen mit Bruchanteilen

TECHNISCHE UNIVERSITÄT DARMSTADT

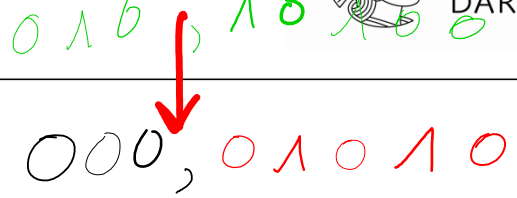
**Zwei gängige Darstellungen:**

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



Embedded Systems & Applications

# Zahlen mit Bruchanteilen

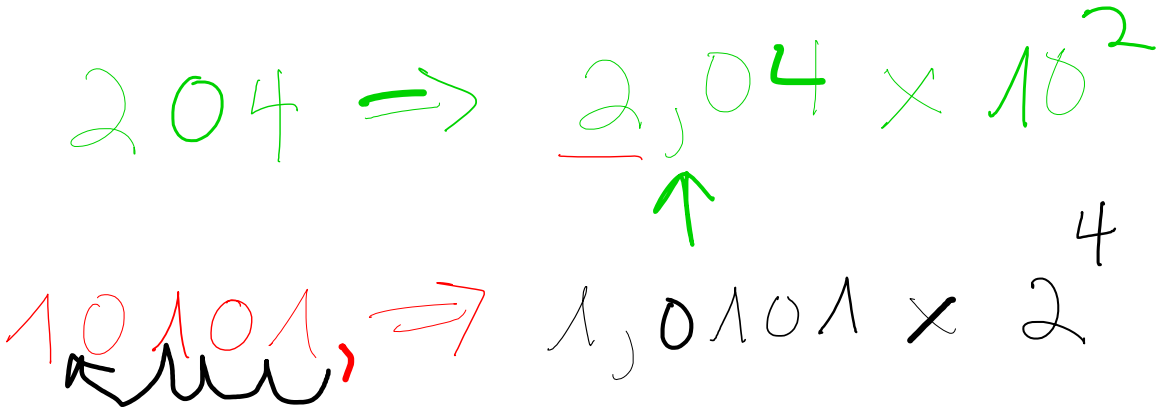


**Festkomma (*fixed-point*):**

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

TECHNISCHE UNIVERSITÄT DARMSTADT

## 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

$2*10^0$ $9,9999*10^1$ $0*10^0$ $-2,718*10^0$

$3,1415*10^0$ $3,6525*10^2$ $5*10^6$

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

Auch: Obergrenze für Exponenten, keine beliebig großen Zahlen darstellbar



Embedded Systems & Applications

# Binäre Festkommazahlen



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

01101100

0110 , 1100

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

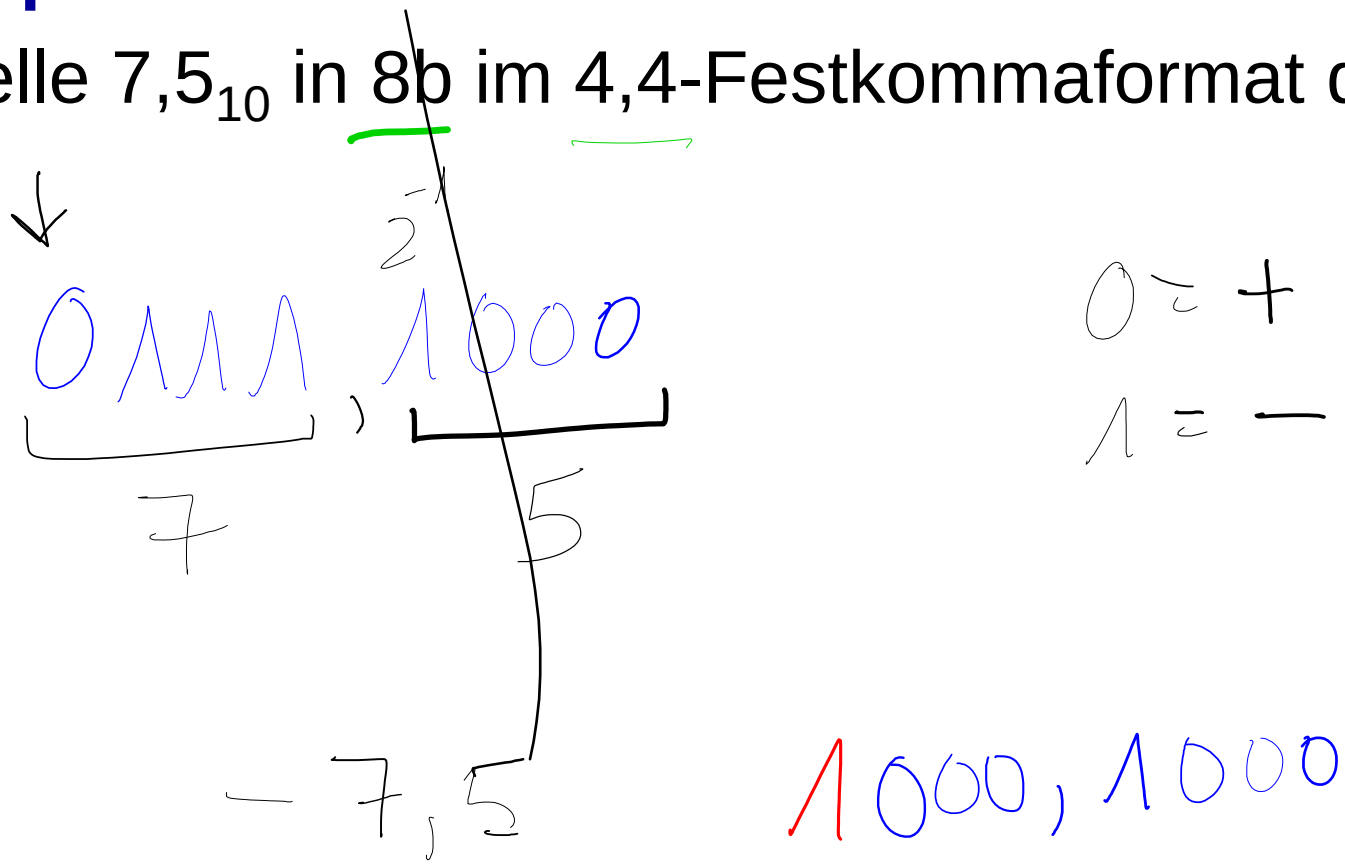
- Binärkomma wird nicht explizit dargestellt: Position wird durch Format impliziert (hier: 4,4)
- Alle Leser und Schreiber von Festkommadaten müssen dasselbe Format verwenden

# Binäre Festkommazahlen

TECHNISCHE UNIVERSITÄT DARMSTADT

## Beispiel:

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






Embedded Systems & Applications

# Binäre Festkommazahlen

TECHNISCHE UNIVERSITÄT DARMSTADT

**Beispiel:**

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

01111000



Embedded Systems & Applications

# Vorzeichenbehaftete Festkommazahlen

TECHNISCHE UNIVERSITÄT DARMSTADT

Wie bei ganzen Zahlen: Zwei Darstellungen möglich

- Vorzeichen/Betrag
- Zweierkomplement

# 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:**



Embedded Systems & Applications

# 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:**

    11111000

    7,5 = 0111 1000
  - **Zweierkomplement:**

    1 1000 0111

    \+ 1

    1000 1000



Embedded Systems & Applications

# 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:**

    11111000
  - **Zweierkomplement:**

| | |
|---|---|
| +7,5: | 01111000 |
| 1. Invertieren: | 10000111 |
| 2. Addiere 1 zu lsb: | + 1 |
| | 10001000 |



Embedded Systems & Applications

# Binäre Gleitkommazahlen

TECHNISCHE UNIVERSITÄT DARMSTADT

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

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



Embedded Systems & Applications

# Binäre Gleitkommazahlen

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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

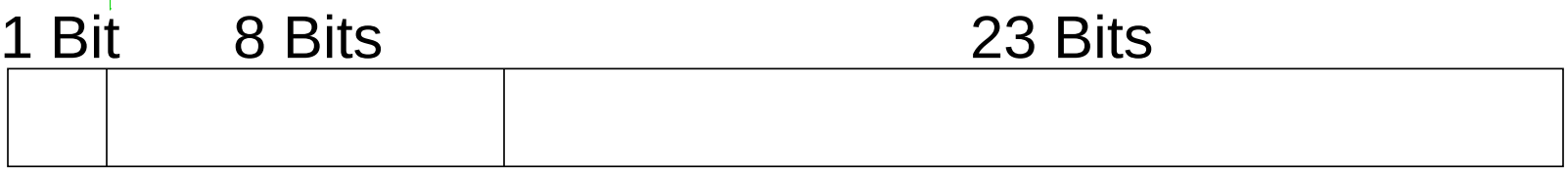
- **M** = Mantisse
- **B** = Basis
- **E** = Exponent
- Im Beispiel: M = 4,387263 , B = 10, and E = 6



Embedded Systems & Applications

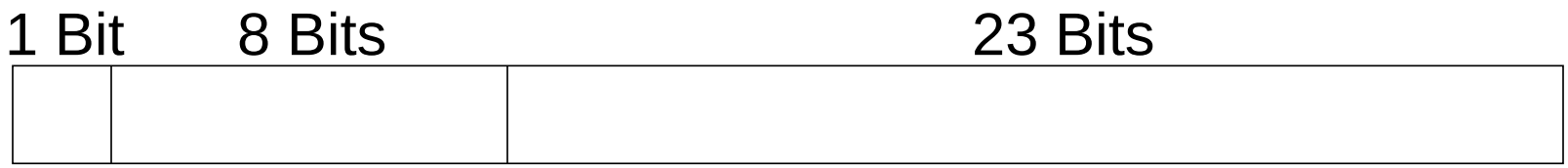
# Binäre Gleitkommazahlen

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




**Vorzeichen** **Exponent** **Mantisse**

# Binäre Gleitkommazahlen




- **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

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Wandele Dezimalzahl in Binärdarstellung um:**
  - $228_{10}$

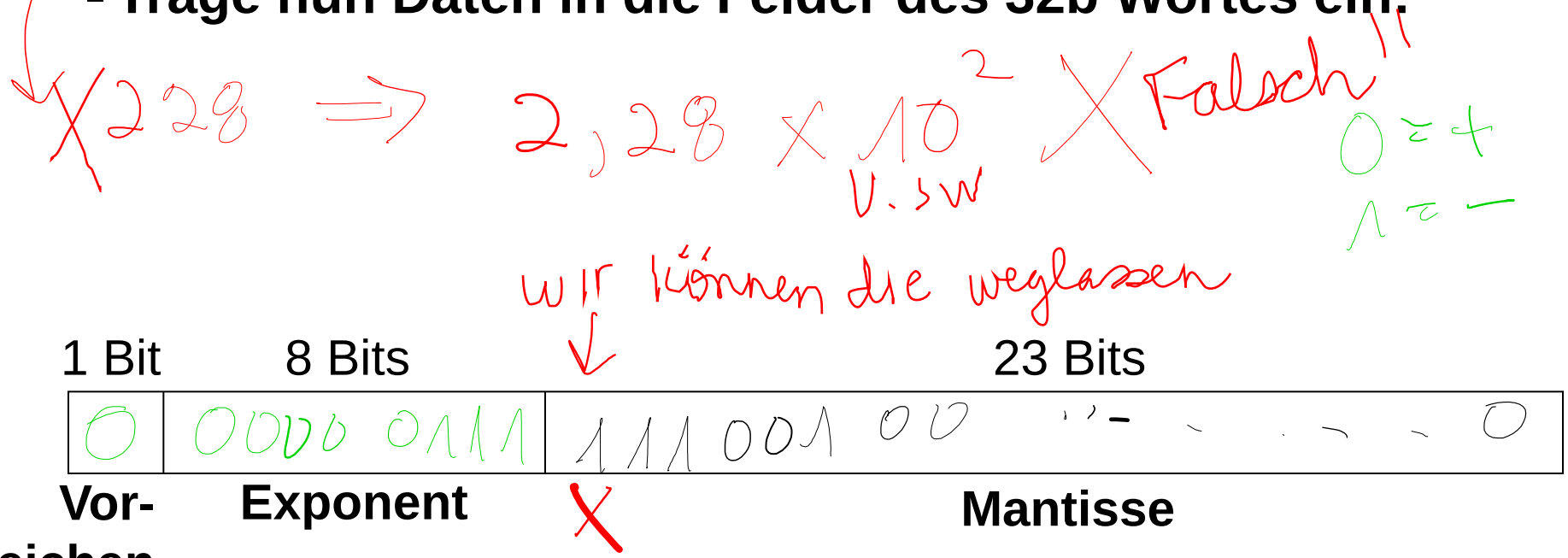


Embedded Systems & Applications

# Binäre Gleitkommadarstellung: 1. Versuch

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Wandele Dezimalzahl in Binärdarstellung um:**
  - $228_{10} = 11100100_2 = 1{,}11001 \times 2^7$
- **Trage nun Daten in die Felder des 32b Wortes ein:**






Embedded Systems & Applications

# Binäre Gleitkommadarstellung: 1. Versuch

TECHNISCHE UNIVERSITÄT DARMSTADT

- **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

| 1 Bit | 8 Bits | 23 Bits |
|---|---|---|
| 0 | 00000111 | 111 0010 0000 0000 0000 0000 |
| **Vor-zeichen** | **Exponent** | **Mantisse** |



Embedded Systems & Applications

# Binäre Gleitkommadarstellung: 2. Versuch

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Beobachtung:** Das erste Bit der Mantisse ist so immer 1
  - $228_{10} = 11100100_2 = \mathbf{1},\mathbf{11001} \times 2^7$
- 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-zeichen** | **Exponent** | **Bruchanteil** |



Embedded Systems & Applications

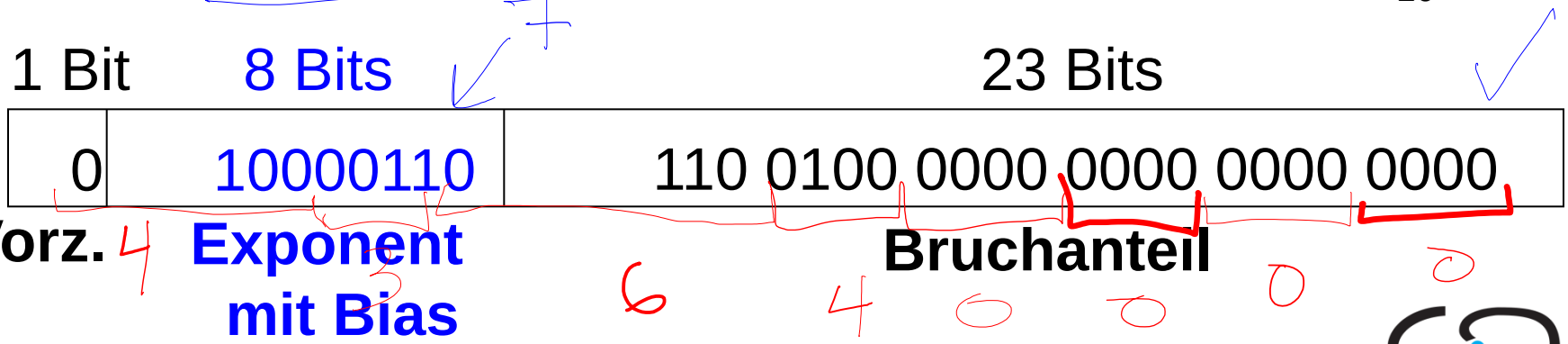
# Binäre Gleitkommadarstellung: 3. Versuch

TECHNISCHE UNIVERSITÄT DARMSTADT

- **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 $228_{10}$






Embedded Systems & Applications

# Beispiel IEEE 754 Gleitkommadarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

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

**1. Wandele in Binärdarstellung um:**

- $58{,}25_{10} = 111010{,}01_2$



Embedded Systems & Applications

# Beispiel IEEE 754 Gleitkommadarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

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

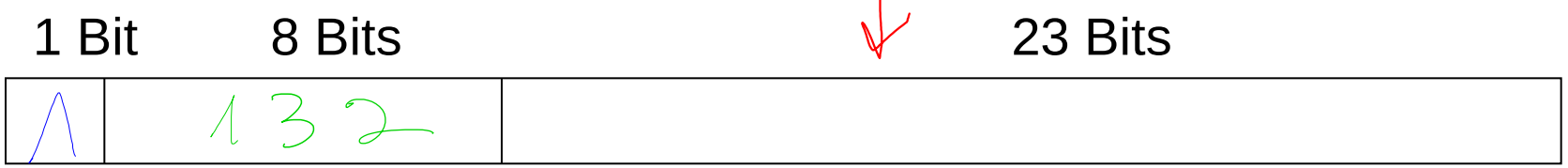
**1. Wandele in Binärdarstellung um:**

- $58{,}25_{10} = 111010{,}01_2 = 1{,}1101001 \times 2^5$

5 + 127 = 132

5 stellen

**2. Trage Felder des 32b Gleitkommawortes ein:**




**Vorz.** **Exponent** **Bruchanteil**



Embedded Systems & Applications

# Beispiel IEEE 754 Gleitkommadarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

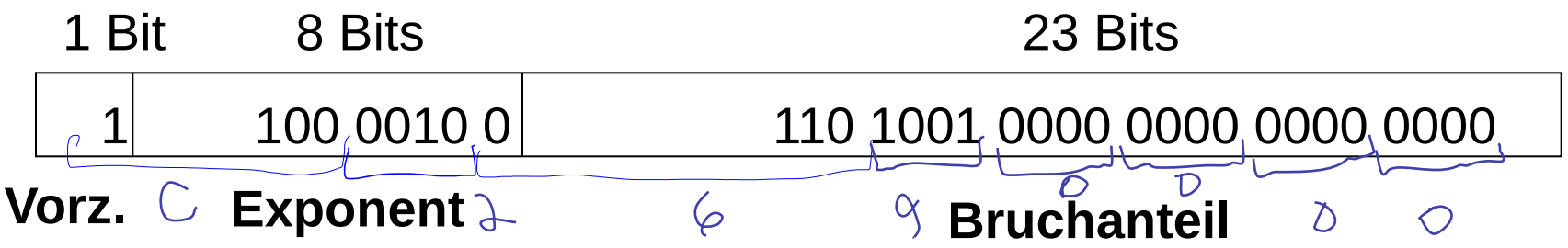
**Stelle -58,25$_{10}$ gemäß dem IEEE 754 32-bit Gleitkommastandard dar**

**1. Wandele in Binärdarstellung um:**

- $58{,}25_{10} = 111010{,}01_2 = 1{,}1101001 \times 2^5$

**2. Trage Felder des 32b Gleitkommawortes ein:**

- **Vorzeichen:** 1 (negativ)
- 8 Bits für **Exponent:** (127 + 5) = 132 = $10000100_2$
- 23 Bits für **Bruchanteil:** 110 1001 0000 0000 0000 0000




**in Hexadezimalschreibweise: 0xC2690000**



Embedded Systems & Applications

# Beispiel IEEE 754 Gleitkommadarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

**Stelle $0{,}125_{10}$ gemäß dem IEEE 754 32-bit Gleitkommastandard dar**

**1. Wandele in Binärdarstellung um:**

# Beispiel IEEE 754 Gleitkommadarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

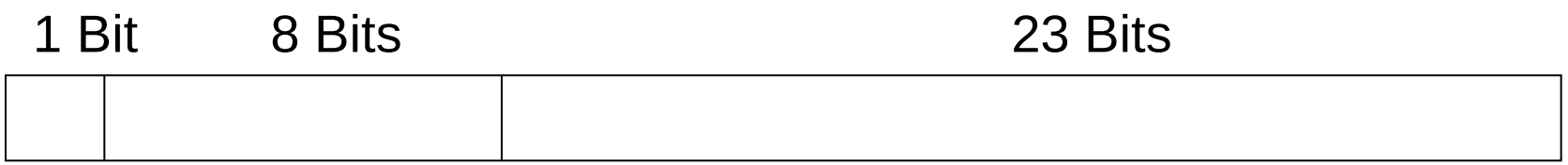
**Stelle $0{,}125_{10}$ gemäß dem IEEE 754 32-bit Gleitkommastandard dar**

**1. Wandele in Binärdarstellung um:**

- $0{,}125_{10} = 0{,}001_2 = 1{,}0 \times 2^{-3}$

$127 + (-3) = 124$

**2. Trage Felder des 32b Gleitkommawortes ein:**




**Vorz.** **Exponent** **Bruchanteil**



Embedded Systems & Applications

# Beispiel IEEE 754 Gleitkommadarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

**Stelle $0{,}125_{10}$ gemäß dem IEEE 754 32-bit Gleitkommastandard dar**

**1. Wandele in Binärdarstellung um:**

- $0{,}125_{10} = 0{,}001_2 = 1{,}0 \times 2^{-3}$

**2. Trage Felder des 32b Gleitkommawortes ein:**

- **Vorzeichen:** 0 (positiv)
- 8 Bits für **Exponent:** (127 - 3) = 124 = $01111100_2$
- 23 Bits für **Bruchanteil:** 000 0000 0000 0000 0000 0000

| 1 Bit | 8 Bits | 23 Bits |
|---|---|---|
| 0 | 011 1110 0 | 000 0000 0000 0000 0000 0000 |
| **Vorz.** | **Exponent** | **Bruchanteil** |

**in Hexadezimalschreibweise: 0x3E000000**



Embedded Systems & Applications

# 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 | 00000000000000000000000 |
| ∞ | 0 | 11111111 | 00000000000000000000000 |
| - ∞ | 1 | 11111111 | 00000000000000000000000 |
| NaN | X | 11111111 | Ein Wert <> 0 |

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

Beispiele: √-1 oder log(-5) oder eine Zahl durch 0.

# Genauigkeit der Gleitkommadarstellungen

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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

TECHNISCHE UNIVERSITÄT DARMSTADT

- **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 für Gleitkommazahlen

TECHNISCHE UNIVERSITÄT DARMSTADT

- **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



Embedded Systems & Applications

# Rundungsmodi für Gleitkommazahlen

TECHNISCHE UNIVERSITÄT DARMSTADT

- **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)



Embedded Systems & Applications

# Addition von Gleitkommazahlen mit gleichem Vorzeichen

TECHNISCHE UNIVERSITÄT DARMSTADT

1. Exponenten- und Bruchanteilfelder 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



Embedded Systems & Applications

# Beispiel: Addition von Gleitkommazahlen

TECHNISCHE UNIVERSITÄT DARMSTADT

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

| 1 Bit | 8 Bits | 23 Bits |
|---|---|---|
| 0 | 01111111 | 100 0000 0000 0000 0000 0000 |
| **Vorz.** | **Exponent** | **Bruchanteil** |

| 1 Bit | 8 Bits | 23 Bits |
|---|---|---|
| 0 | 10000000 | 101 0000 0000 0000 0000 0000 |
| **Vorz.** | **Exponent** | **Bruchanteil** |

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

1. Zahl (N1): S1 = 0, E1 = 127 (= × $2^{0}$), F1 = ,1

2. Zahl (N2): S2 = 0, E2 = 128 (= × $2^{1}$), F2 = ,101



Embedded Systems & Applications

# Beispiel: Addition von Gleitkommazahlen

TECHNISCHE UNIVERSITÄT DARMSTADT

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

| 1 Bit | 8 Bits | 23 Bits |
|---|---|---|
| 0 | 01111111 | 100 0000 0000 0000 0000 0000 |
| **Vorz.** | **Exponent** | **Bruchanteil** |

| 1 Bit | 8 Bits | 23 Bits |
|---|---|---|
| 0 | 10000000 | 101 0000 0000 0000 0000 0000 |
| **Vorz.** | **Exponent** | **Bruchanteil** |
| **S** | **E** | **F** |

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

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

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

M1: 1,1

M2: 1,101



Embedded Systems & Applications

# Beispiel: Addition von Gleitkommazahlen

TECHNISCHE UNIVERSITÄT DARMSTADT

## 3. Vergleiche Exponenten

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

N1: $1{,}1 \times 2^0$

N2: $1{,}101 \times 2^1$

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

schiebe M1: 1,1 >> 1 = 0,11 **(× $2^1$)**

N1: $0{,}11 \times 2^1$

N2: $1{,}101 \times 2^1$

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

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



Embedded Systems & Applications

# Beispiel: Addition von Gleitkommazahlen

TECHNISCHE UNIVERSITÄT DARMSTADT

## 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 |
| **Vorz.** | **Exponent** | **Bruchanteil** |



Embedded Systems & Applications

# Addition von Gleitkommazahlen mit gleichem Vorzeichen

TECHNISCHE UNIVERSITÄT DARMSTADT

1. Exponenten- und Bruchanteilfelder 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

# Organisatorisch

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Keine Vorlesung** am kommenden Montag (16.05.2016) – Feiertag (Pfingstmontag)

- Keine Übungen oder Testate zum Übungsblatt werden am 23.-27.05. stattfinden. (Diese Woche würde sich allerdings sehr gut für Testate zur Programmier-Aufgabe eigenen.)

# Wiederholung: Binäre Gleitkommadarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

## Felder:

- **Vorzeichen:** 0 (positiv), 1 (negativ)
- **Exponent mit bias:** Exponent + Biaswert (127)
- **Bruchanteil:** den Bruchanteil (die “Nachkommastellen”)

## Beispiel:

- $372{,}1875_{10} = 101110100{,}0011_2 = 1{,}011101000011 \times 2^8$
- **Vorzeichen:** 0
- **Exponent mit Bias:** 127+8 = 135 (10000111)
- **Bruchanteil:** 01110100001100000000000

| 1 Bit | 8 Bits | 23 Bits |
|---|---|---|
| 0 | 10000111 | 011 1010 0001 1000 0000 0000 |
| **Vorz.** | **Exponent mit Bias** | **Bruchanteil** |

**0x43BA1800**



Embedded Systems & Applications

# Gleitkommabefehle

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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.



Embedded Systems & Applications

# Gleitkommabefehle

TECHNISCHE UNIVERSITÄT DARMSTADT

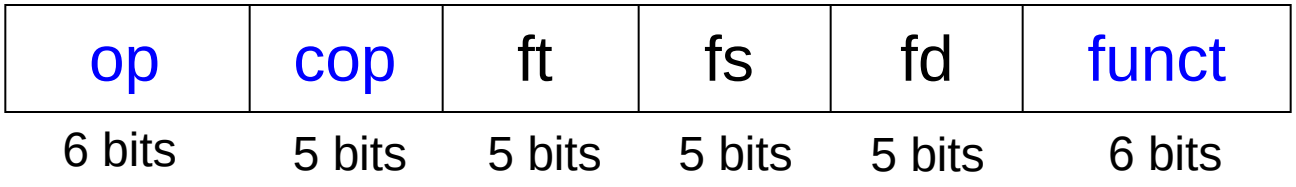
| Namen | Registernummern | Zweck |
|---|---|---|
| `$fv0-$fv1` | 0, 2 | Rückgabewerte |
| `$ft0-$ft3` | 4, 6, 8, 10 | Temporäre Variablen |
| `$fa0-$fa1` | 12, 14 | Prozedurargumente |
| `$ft4-$ft5` | 16, 18 | Temporäre Variablen |
| `$fs0-$fs5` | 20, 22, 24, 26, 28, 30 | Erhaltene Variablen |

# Format für F-Typ Instruktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

- `Opcode` = 17 ($010001_2$)
- Drei Registeroperanden:
  - `fs`, `ft`: Quelloperanden
  - `fd`: Zieloperanden
- **Single-precision:**
  - `cop` = 16 ($010000_2$)
  - `add.s`, `sub.s`, `div.s`, `neg.s`, `abs.s`, u.s.w.
- **Double-precision:**
  - `cop` = 17 ($010001_2$)
  - `add.d`, `sub.d`, `div.d`, `neg.d`, `abs.d`, u.s.w.

**F-Typ**






Embedded Systems & Applications

# Format für F-Typ Instruktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

- `Opcode` = 17 ($010001_2$)
- Drei Registeroperanden:
  - `fs`, `ft`: Quelloperanden
  - `fd`: Zieloperanden
- **Single-precision:**
  - `cop` = 16 ($010000_2$)
  - `add.s`, `sub.s`, `div.s`, `neg.s`, `abs.s`, u.s.w.
- **Double-precision:**
  - `cop` = 17 ($010001_2$)
  - `add.d`, `sub.d`, `div.d`, `neg.d`, `abs.d`, u.s.w.

**Beispiel:**

```
# $fs1 = $fs2 + $fs3
add.s $fs1, $fs2, $fs3
```

## F-Typ

| op | cop | ft | fs | fd | funct |
|---|---|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |



Embedded Systems & Applications

# Weitere Gleitkommabefehle

TECHNISCHE UNIVERSITÄT DARMSTADT

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` = FALSCH
- `bc1t`: springt falls `fpcond` = TRUE (wahr)
- Beispiel: `bc1f toosmall`

Loads und Stores: jeweils Single precision

- `lwc1`: `lwc1 $ft1, 42($s1)`
- `swc1`: `swc1 $fs2, 16($sp)`
- Double precision braucht je zwei Anweisungen



Embedded Systems & Applications

# Ausblick

TECHNISCHE UNIVERSITÄT DARMSTADT

Bisher **Architektur**

- Programmierersicht

Nun **Mikroarchitektur**

- Aufbau der zugrundeliegenden Hardware



Embedded Systems & Applications