

# Technische Grundlagen der Informatik – Kapitel 6



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Prof. Dr.-Ing. Andreas Koch  
Fachgebiet Eingebettete Systeme und ihre Anwendungen (ESA)  
Fachbereich Informatik

WS 12/13



# Kapitel 6: Themen



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

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

- 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

Anwendungs-Software	Programme
Betriebs-systeme	Gerätetreiber
Architektur	Instruktionen Register
Mikro-architektur	Datenpfade Steuerwerke
Logik	Addierer Speicher
Digital-schaltungen	AND Gatter NOT Gatter
Analog-schaltungen	Verstärker Filter
Bauelemente	Transistoren Dioden
Physik	Elektronen



- 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:
  - 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

# John Hennessy



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

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



# Entwurfsprinzipien für Architekturen



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

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

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

# Befehle: Addition



## Hochsprache

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

## MIPS Assemblersprache

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

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

# Befehl: Subtraktion



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

## Hochsprache

$a = b - c;$

## MIPS Assemblersprache

sub a, b, c

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





## Regularität vereinfacht Entwurf

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

# Befehle: Komplexere Abläufe



- Komplexere Abläufe werden durch Folgen von einfachen Befehlen realisiert

## Hochsprache

```
a = b + c - d;  
// Kommentare bis Zeilenende  
/* mehrzeiliger Kommentar */
```

## MIPS Assemblersprache

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



## 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)
  - Befehl: Kopiere Zeichenfolge im Speicher umher

# Operanden

- Ein Prozessor hat physikalische Speicherorte für die Operanden von Befehlen
- Mögliche Speicherorte
  - Register
  - Speicher
  - Konstante Werte (*immediates*)
    - Stehen häufig direkt im Befehl

# Operanden: Register

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

## Kleiner ist schneller

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

# MIPS Registerfeld

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

# Operanden: Register



- Register:
  - Kennzeichnend gemacht durch dem Namen vorangestelltes Dollar-Zeichen
  - Beispiel: Register 0 wird geschrieben als “\$0”
    - Gelesen als: “Register Null” oder “Dollar Null”.
- Bestimmte Register für bestimmte Verwendungszwecke:
  - Beispiele
    - \$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
- Zunächst benutzen wir nur
  - Temporäre Register (\$t0 - \$t9)
  - Gesicherte Register (\$s0 - \$s7)
- Später mehr ...



# Befehle mit Registerangaben



- Rückblick auf add-Befehl

## Hochsprache

```
a = b + c
```

## MIPS Assemblersprache

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

# Operanden: Speicher



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

# Wort-Adressierung von Daten im Speicher



- Jedes 32-bit Datenwort hat eine eindeutige Adresse

Wortadresse	Daten	
⋮	⋮	⋮
00000003	4 0 F 3 0 7 8 8	Wort 3
00000002	0 1 E E 2 8 4 2	Wort 2
00000001	F 2 F 1 A C 0 7	Wort 1
00000000	A B C D E F 7 8	Wort 0

# Lesen aus wort-adressiertem Speicher



- Lesen geschieht durch Ladebefehle (*load*)
- Befehlsname: *load word* (`lw`)
- **Beispiel:** Lese ein Datenwort von der Speicheradresse 1 nach `$s3`
  - Adressarithmetik: Adressen werden relativ zu einem Register angegeben
  - Basisadresse (`$0`) plus Distanz (*offset*) (1)
  - Adresse = (`$0 + 1`) = 1
  - Jedes Register darf als Basisadresse verwendet werden
  - Nach Abarbeiten des Befehls hat `$s3` den Wert `0xF2F1AC07`

## Assemblersprache (nicht MIPS!)

```
lw $s3, 1($0) # lese Wort 1 aus Speicher in $s3
```

Wortadresse	Daten	
⋮	⋮	⋮
00000003	4 0 F 3 0 7 8 8	Wort 3
00000002	0 1 E E 2 8 4 2	Wort 2
00000001	F 2 F 1 A C 0 7	Wort 1
00000000	A B C D E F 7 8	Wort 0

# Schreiben in wort-adressiertem Speicher



- Schreiben geschieht durch Speicherbefehle (*store*)
- Befehlsname: *store word* (*sw*)
- **Beispiel:** Schreibe (speichere) den Wert aus  $\$t4$  in Speicherwort 7
- Offset kann dezimal (Standard) oder hexadezimal angegeben werden
- Adressarithmetik:
  - Basisadresse ( $\$0$ ) plus Offset ( $0x7$ )
  - Adresse: ( $\$0 + 0x7$ ) = 7
  - Jedes Register darf als Basisadresse verwendet werden

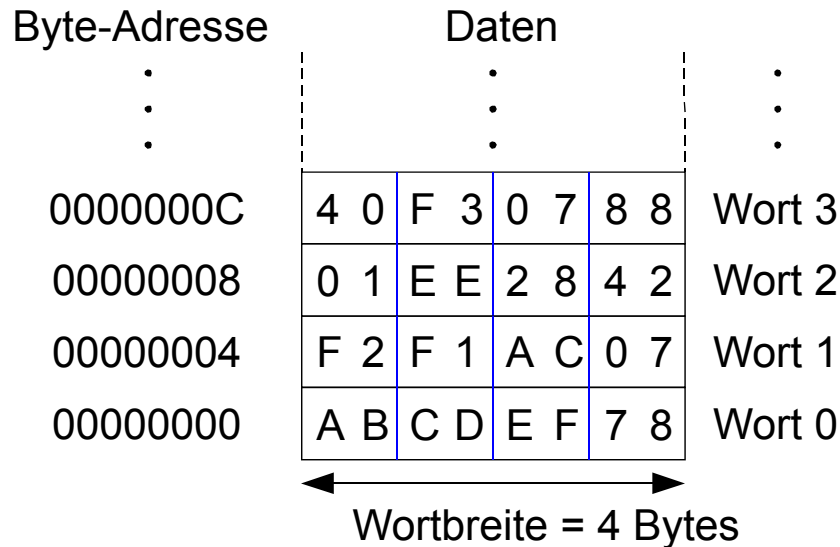
## Assemblersprache (nicht MIPS!)

```
sw $t4, 0x7($0) # schreibe Wert aus $t4 in Speicherwort 7
```

Wortadresse	Daten	
⋮	⋮	⋮
00000003	4 0 F 3 0 7 8 8	Wort 3
00000002	0 1 E E 2 8 4 2	Wort 2
00000001	F 2 F 1 A C 0 7	Wort 1
00000000	A B C D E F 7 8	Wort 0

# Byte-adressierbarer Speicher

- Jedes **Byte** hat eine individuelle Adresse
- Speicherbefehle können auf Worten oder Bytes arbeiten
  - Worte:  $1w / sw$  Bytes:  $1b / sb$
- Jedes Wort enthält vier Bytes
  - Adressen von Worten sind also Vielfache von 4



# Lesen aus byte-adressiertem Speicher



- Adresse eines Speicherwortes muss nun mit 4 multipliziert werden
  - Byte-Adresse von Wort 2 ist  $2 \times 4 = 8$
  - Byte-Adresse von Wort 10 is  $10 \times 4 = 40$  (0x28)
- Beispiel: Lade Wort 1 aus Byte-Adresse 4 nach  $\$s3$
- Nach dem Befehl enthält  $\$s3$  den Wert 0xF2F1AC07
- **MIPS ist byte-adressiert, nicht wort-adressiert!**

## Assemblersprache

```
lw $s3, 4($0) # Lese Wort 1 an Byte-Adresse 4 nach $s3
```

Byte-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

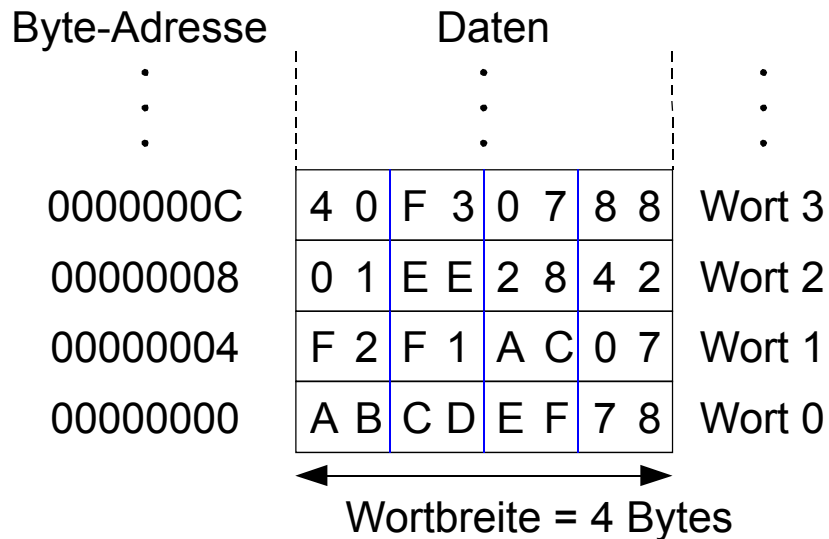
← Wortbreite = 4 Bytes →

# Schreiben in byte-adressiertem Speicher

- **Beispiel:** Schreibe Wert aus `$t7` in Speicheradresse `0x2C (44)`

## MIPS Assemblersprache

```
sw $t7, 44($0) # schreibe $t7 nach Byte-Adresse 44
```

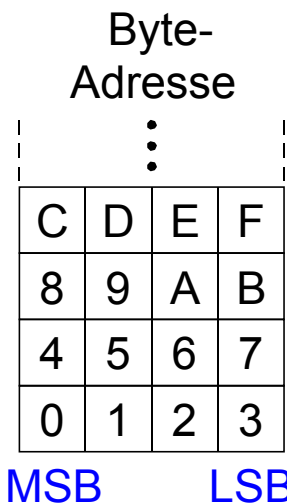




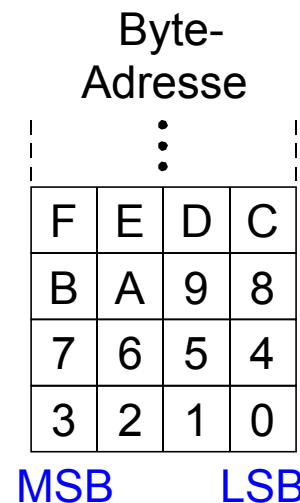
# Speicherorganisation: Big-Endian und Little-Endian

- Schemata für Nummerierung von Bytes in einem Wort
  - Wort-Adresse ist bei beiden gleich
- Little-endian: Bytes werden vom niederstwertigen Ende an gezählt
- Big-endian: Bytes werden vom höchstwertigen Ende an gezählt

## Big-Endian



## Little-Endian



# Speicherorganisation: Big-Endian und Little-Endian

- Aus Jonathan Swift's *Gullivers Reisen*
  - Little-Endians schlagen Eier an der schmalen Seite auf
  - Big-Endians schlagen Eier an der breiten Seite auf
- Welche Organisation benutzt wird ist im Prinzip egal ...
- ... außer wenn unterschiedliche Systeme Daten austauschen müssen

## Big-Endian

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

MSB                  LSB

## Little-Endian

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

MSB                  LSB

# Beispiel: Big-Endian und Little-Endian



- Annahme:  $\$t0$  enthält den Wert  $0x23456789$
- Programm:

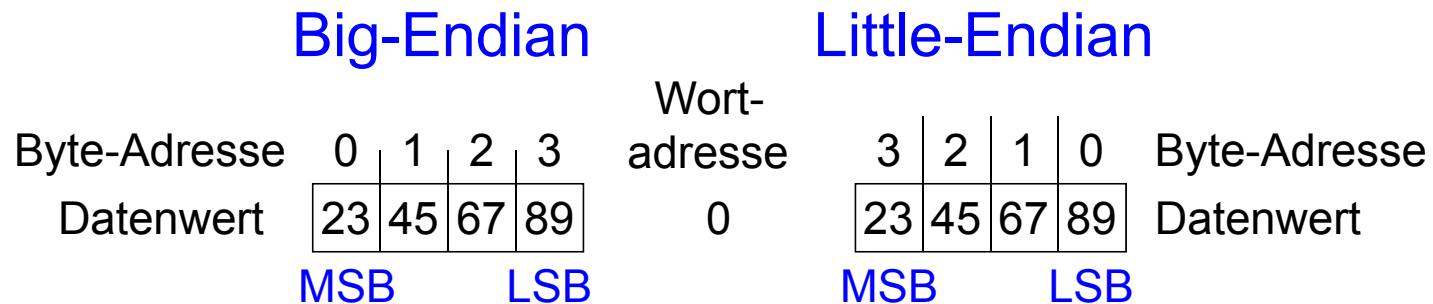
```
sw $t0, 0($0)
lb $s0, 1($0)
```
- Fragen: Welchen Wert hat  $\$s0$  nach Ausführung auf einem ...
  - ... Big-Endian Prozessor?
  - ... Little-Endian Prozessor?

# Beispiel: Big-Endian und Little-Endian



- 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?  $0x00000067$





## Ein guter Entwurf verlangt gute Kompromisse

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

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

- `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
```



- Computer verstehen nur 0'en und 1'en
- Maschinensprache: Binärdarstellung von Befehlen
- 32b Befehle
  - Regularität vereinfacht Entwurf: Daten und Befehle sind beides 32b Worte
- Drei Befehlsformate
  - R-Typ: Operanden sind nur Register
  - I-Typ: Register und ein Direktwert
  - J-Typ: für Programmsprünge (kommt noch)

# Befehlsformat R-Typ



- *Register Typ*
- 3 Registeroperanden
  - `rs, 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





# Beispiele für Befehle vom R-Typ

## Assemblersprache

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

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

## Felder in Befehlswort

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

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

## Maschinsprache

op	rs	rt	rd	shamt	funct	
000000	10001	10010	10000	00000	100000	(0x02328020)
000000	01011	011010	01000	00000	100010	(0x016D4022)

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

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

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

# Befehlsformat I-Typ



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

## I-Typ



# Beispiel für Befehle vom I-Typ

## Assemblersprache

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

## Felder im Befehlswort

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

6 bits      5 bits      5 bits      16 bits

## 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

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

addi rt, rs, imm

lw rt, imm(rs)

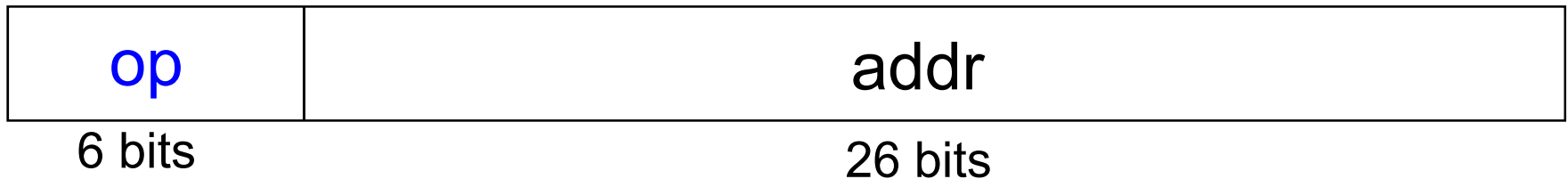
sw rt, imm(rs)

# Befehlsformat J-Typ



- *Jump-Typ*
- 26b Adressoperand (*addr*)
- Verwendet für Sprungbefehle (*j*)

## J-Typ



# Übersicht über Befehlsformate

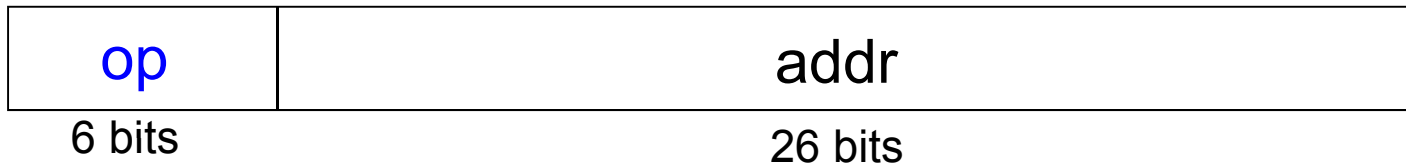
## R-Typ



## I-Typ



## J-Typ



# Flexibilität durch gespeicherte Programme



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

- 32b Befehle und Daten im **Speicher**
- Folgen von Befehlen bestimmen **Verhalten**
  - Einziger Unterschied zwischen zwei Anwendungen
- Ausführen von **unterschiedlichen** Programmen
  - Ohne Neuverdrahten oder Neuaufbau von Hardware
  - Nur neues Programm als **Maschinensprache** im Speicher ablegen
- Die Hardware des Prozessors führt Programm **schrittweise** aus:
  - Holt neue Befehle aus dem Speicher (*fetch*) in richtiger Reihenfolge
  - Führt die im Befehl verlangte Operation aus
- **Programmzähler** (*program counter, PC*)
  - Zeigt Adresse des auszuführenden Befehls an
- Bei MIPS: Programmausführung **beginnt** auf Adresse 0x00400000

# Im Speicher abgelegtes Programm



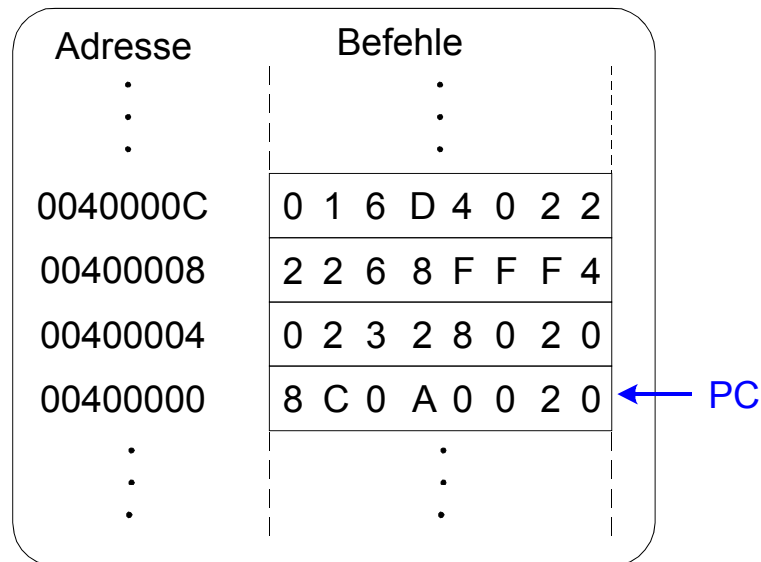
## Assemblersprache

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

## Maschinensprache

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

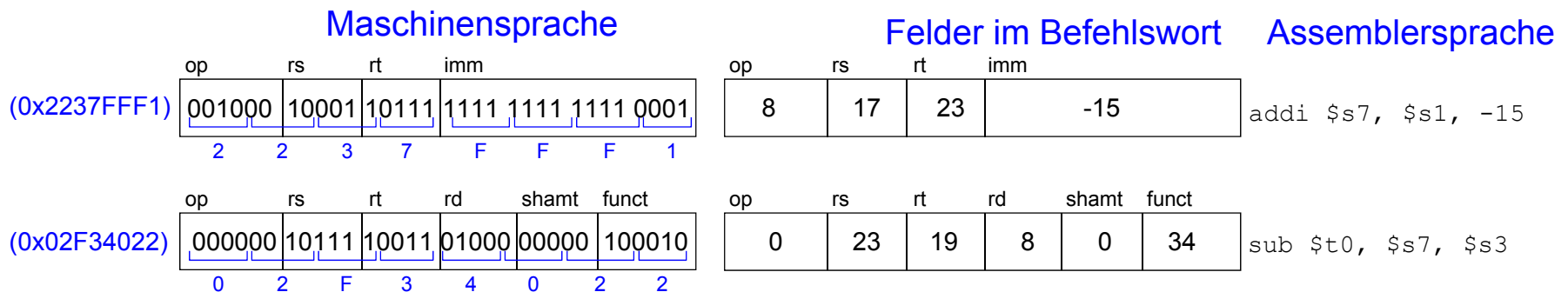
## Programm im Speicher



Hauptspeicher

# Maschinensprache verstehen

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







- Hochsprachen:
  - z.B. C, Java, Python, Scheme
  - Auf einer **abstrakteren** Ebene programmieren
  
- **Häufige** Konstrukte in Hochsprachen:
  - if/else-Anweisungen
  - for-Schleifen
  - while-Schleifen
  - Feld (Array) zugriffe
  - Prozeduraufrufe
  
- Andere **nützliche** Anweisungen:
  - Arithmetische/logische Ausdrücke
  - Verzweigungen

# Ada Lovelace, 1815 - 1852



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

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



# Logische Befehle



- `and, or, xor, nor`
  - `and`: nützlich zum **Maskieren** von Bits
    - Ausmaskieren aller Bits außer dem LSB:  
 $0xF234012F \text{ AND } 0x000000FF = 0x0000002F$
  - `or`: Nützlich zum **Vereinigen** von Bitfeldern
    - Vereinige `0xF2340000` mit `0x000012BC`:  
 $0xF2340000 \text{ OR } 0x000012BC = 0xF23412BC$
  - `nor`: nützlich zur **Invertierung** von Bits:
    - $A \text{ NOR } 0 = \text{NOT } A$
- `andi, ori, xori`
  - 16-bit Direktwert wird erweitert mit führenden Nullbits (**nicht vorzeichenerweitert**)
  - `nori` wird nicht benötigt

# Beispiele: Logische Befehle



## Quellregister

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

## Assemblersprache

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

## Ergebnisse

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

# Beispiele: Logische Befehle

## Quellregister

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

## Assemblersprache

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

## Ergebnisse

<b>\$s3</b>	0100	0110	1010	0001	0000	0000	0000	0000
<b>\$s4</b>	1111	1111	1111	1111	1111	0000	1011	0111
<b>\$s5</b>	1011	1001	0101	1110	1111	0000	1011	0111
<b>\$s6</b>	0000	0000	0000	0000	0000	1111	0100	1000

# Beispiele: Logische Befehle



## Operanden

<b>\$s1</b>	0000	0000	0000	0000	0000	0000	1111	1111
<b>imm</b>	0000	0000	0000	0000	1111	1010	0011	0100

◀ Null-erweitert ▶

## Assemblersprache

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

## Ergebnisse

<b>\$s2</b>								
<b>\$s3</b>								
<b>\$s4</b>								

# Beispiele: Logische Befehle

## Operanden

<b>\$s1</b>	0000	0000	0000	0000	0000	0000	1111	1111
<b>imm</b>	0000	0000	0000	0000	1111	1010	0011	0100
					← Null-erweitert →			

## Assemblersprache

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

## Ergebnisse

<b>\$s2</b>	0000	0000	0000	0000	0000	0000	0011	0100
<b>\$s3</b>	0000	0000	0000	0000	1111	1010	1111	1111
<b>\$s4</b>	0000	0000	0000	0000	1111	1010	1100	1011

- `sll`: shift left logical
  - **Beispiel:** `sll $t0, $t1, 5 # $t0 <= $t1 << 5`
- `srl`: shift right logical
  - **Beispiel :** `srl $t0, $t1, 5 # $t0 <= $t1 >> 5`
- `sra`: shift right arithmetic
  - **Beispiel :** `sra $t0, $t1, 5 # $t0 <= $t1 >>> 5`

## Schieben mit variabler Distanz:

- `sllv`: shift left logical variable
  - **Beispiel :** `sllv $t0, $t1, $t2 # $t0 <= $t1 << $t2`
- `srlv`: shift right logical variable
  - **Beispiel :** `srlv $t0, $t1, $t2 # $t0 <= $t1 >> $t2`
- `srav`: shift right arithmetic variable
  - **Beispiel :** `srav $t0, $t1, $t2 # $t0 <= $t1 >>> $t2`



# Schiebebefehle



## Assemblersprache

## Felder in Instruktion

	op	rs	rt	rd	shamt	funct
sll \$t0, \$s1, 2	0	0	17	8	2	0
srl \$s2, \$s1, 2	0	0	17	18	2	2
sra \$s3, \$s1, 2	0	0	17	19	2	3

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

## Maschinsprache

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

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

# Handhabung von Konstanten



- **16-Bit Konstante** mit `addi`:

## Hochsprache

```
// int ist ein vorzeichenbehaftetes 32b Wort  
int a = 0x4f3c;
```

## MIPS Assemblersprache

```
# $s0 = a  
addi $s0, $0, 0x4f3c
```

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

## Hochsprache

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

## MIPS Assemblersprache

```
# $s0 = a  
lui $s0, 0xFEDC  
ori $s0, $s0, 0x8765
```

# Multiplikation und Division



- **Spezialregister:** `lo`, `hi`
- **32b × 32b Multiplikation, 64b Produkt**
  - `mult $s0, $s1`
  - Ergebnis in `{hi, lo}`
- **32b Division, 32b Quotient, 32b Rest**
  - `div $s0, $s1`
  - Quotient in `lo`
  - Rest in `hi`
- **Lesen** von Daten aus Spezialregistern („*move from ...*“)
  - `mflo $s2`
  - `mfhi $s3`

# Verzweigungen und Sprünge



- Ändern der Ausführungsreihenfolge von Befehlen
- Arten von Verzweigungen: Beispiele
  - **Bedingte**
    - branch if equal ( $b_{eq}$ ): Verzweige, wenn gleich
    - branch if not equal ( $b_{ne}$ ): Verzweige, wenn ungleich
  - **Unbedingte Verzweigungen**
    - jump ( $j$ ): Sprünge
    - jump register ( $j_r$ ): Sprünge auf Adresse aus Register
    - jump and link ( $j_{al}$ ): Sprünge und merke Adresse des nächsten Befehls

# Wiederholung: Programm im Speicher



## Assemblersprache

## Maschinensprache

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

## Abgespeichertes Programm

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

Hauptspeicher

# Bedingte Verzweigungen (beq)



## # MIPS Assemblersprache

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

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

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

# Nicht genommene Sprünge (bne)



## # MIPS Assemblersprache

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

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

# Unbedingte Verzweigungen / Springen (j)



## # MIPS Assemblersprache

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

target:

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



# Unbedingte Verzweigungen (jr)



## # MIPS Assemblersprache

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

# Konstrukte in Hochsprachen



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

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

# If-Anweisung



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
```



## 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 ( $i \neq j$ ) als in der Hochsprache ( $i == j$ ).

# If / Else -Anweisung



## Hochsprache

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

## MIPS Assemblersprache

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

# If / Else-Anweisung



## Hochsprache

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

## MIPS Assemblersprache

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

# While-Schleife



## Hochsprache

```
// berechnet x = ld 128

int pow = 1;
int x   = 0;

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

## MIPS Assemblersprache

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

# While-Schleife



## Hochsprache

```
// berechnet x = 1d 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`).



## Allgemeiner Aufbau:

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

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

# For-Schleifen



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
```

# For-Schleifen



## Hochsprache

```
// addiere Zahlen von 0 to 9 auf
int sum = 0;
int i;

for (i = 0; i != 10; i = i+1) {
    sum = sum + i;
}
```

## MIPS Assemblersprache

```
# $s0 = i, $s1 = sum
    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 ( $i != 10$ ).

# Kleiner-Als Vergleiche



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

## Hochsprache

```
// addiere Zweierpotenzen
// kleiner als 100

int sum = 0;
int i;

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

## MIPS Assemblersprache

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

# Kleiner-als Vergleiche



## Hochsprache

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

int sum = 0;
int i;

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

## MIPS Assemblersprache

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

        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:
```

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

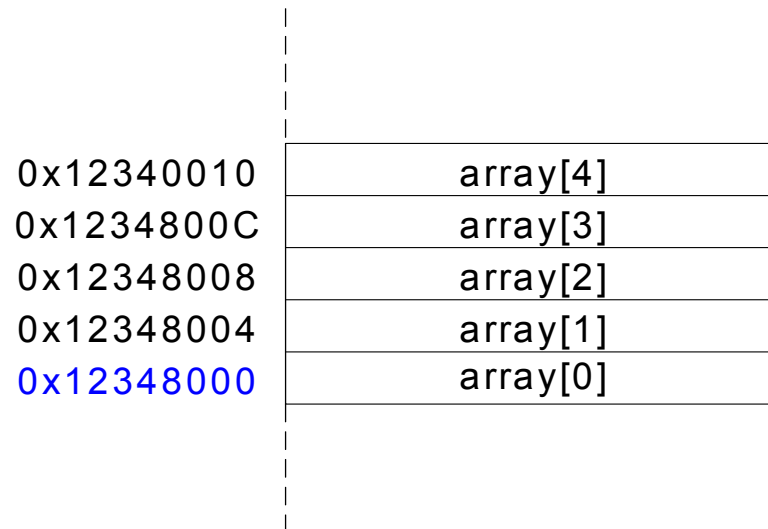
# Datenfelder (*arrays*)



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

# Verwendung von Arrays

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



# 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
```



# Verwendung von Arrays



## // 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
```

# Bearbeite Array in for-Schleife



## // 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
```

# Bearbeite Array in for-Schleife



## // 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:
```

# Zeichendarstellung im ASCII-Code



- *American Standard Code for Information Interchange*
  - Definiert für gängige Textzeichen einen 7b breiten Code
  - Einfach, aber schon älter
  - Heute Unicode: breitere Darstellung für *alle* Textzeichen
- Beispiel: “S” = 0x53, “a” = 0x61, “A” = 0x41
- Klein- und Großbuchstaben liegen auseinander um 0x20 (32).

# Zuordnung von Zeichen zu Codes



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

# Prozedur- und Funktionsaufruf



## 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



## Aufrufkonventionen:

- 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

## Konventionen für MIPS:

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

# Prozedur- und Funktionsaufruf



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

## Hochsprache

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

## MIPS Assemblersprache

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

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

- Also eine Prozedur und keine Funktion ist



# Prozedur- und Funktionsaufruf



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

## Hochsprache

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

void simple() {
    return;
}
```

## MIPS Assemblersprache

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

0x00401020 simple: jr  $ra
```

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

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

# Aufrufargumente und Rückgabewert



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

## MIPS Konventionen:

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

# Aufrufargumente und Rückgabewert



## Hochsprache

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

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

# Aufrufargumente und Rückgabewert



## MIPS Assemblersprache

```
# $s0 = y
```

```
main:
```

```
...
```

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

```
# $s0 = Rückgabewert
```

```
diffofsums:
```

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



## MIPS Assemblersprache

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

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

# Stack (auch Stapel- oder Kellerspeicher)

- Speicher für temporäres Zwischenspeichern von Werte
- Agiert wie ein Stapel (Beispiel: Teller)
  - Zuletzt aufgelegter Teller wird zuerst heruntergenommen
  - “last in, first out” (LIFO)
- *Dehnt sich aus*: Belegt mehr Speicher, wenn mehr Daten unterzubringen sind
- *Zieht sich zusammen*: Belegt weniger Speicher, wenn zwischengespeicherte Daten nicht mehr gebraucht werden



# Stack

- Wächst bei MIPS nach **unten** (von hohen zu niedrigeren Speicheradressen)
  - Übliche Realisierung (deshalb auch **Kellerspeicher** genannt)
- Stapelzeiger (“stack pointer”):  $\$sp$ 
  - zeigt auf zuletzt auf dem Stack abgelegtes Datenelement

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

# Verwendung des Stacks in Prozeduren



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

## # MIPS Assemblersprache

```
# $s0 = result
```

```
diffofsums:
```

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

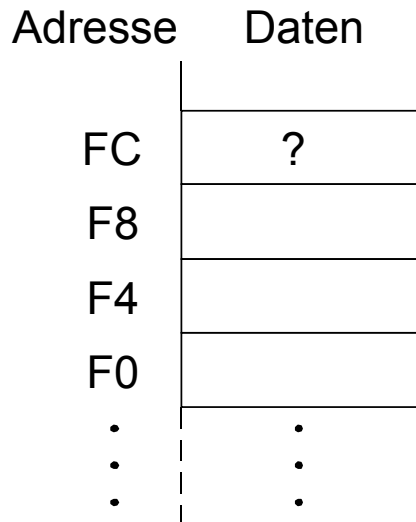


# Register auf Stack zwischenspeichern



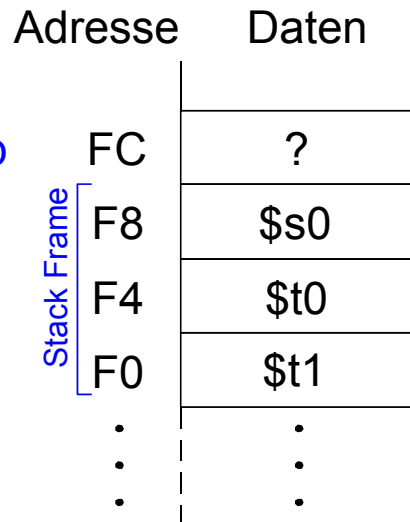
```
# $s0 = result
diffofsums:
    addi $sp, $sp, -12    # 3*4 Bytes auf Stack anfordern
                        # um drei 32b Register zu sichern
    sw   $s0, 8($sp)     # speichere $s0 auf Stack
    sw   $t0, 4($sp)     # speichere $t0 auf Stack
    sw   $t1, 0($sp)     # speichere $t1 auf Stack
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

# Veränderung des Stacks während `diffofsums`



(a)

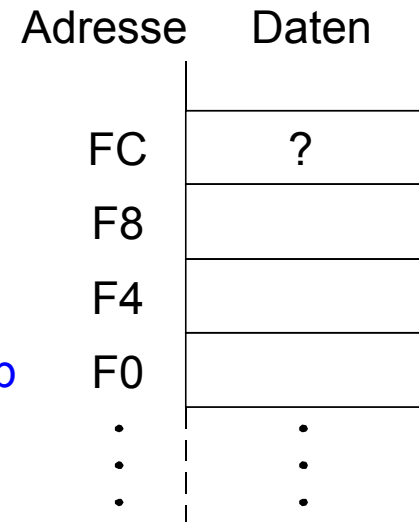
←  $\$sp$



(b)

Stack Frame

←  $\$sp$



(c)

←  $\$sp$

# Sicherungskonventionen für Register



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

# Mehrfache Prozeduraufrufe: Sichern von \$ra



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

# Erhalten von Registern mittels Stack



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



## Hochsprache

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

# Rekursive Prozeduraufrufe



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

## MIPS Assemblersprache



## MIPS Assemblersprache

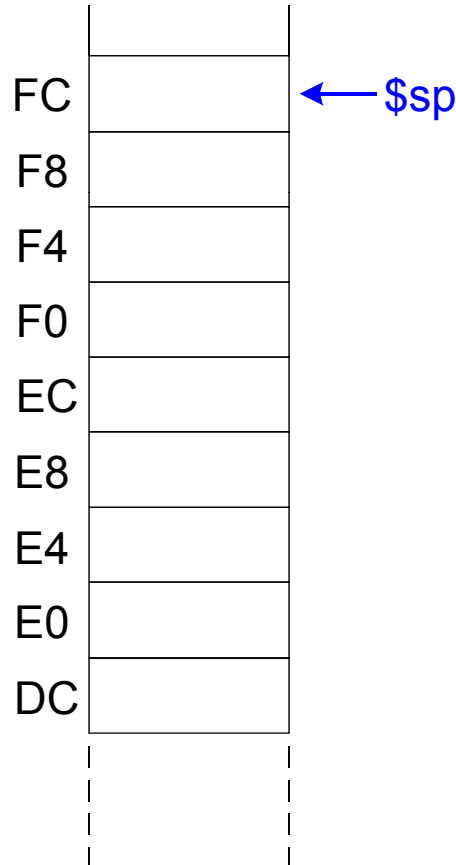
```
0x90 fakultaet: addi $sp, $sp, -8 # Platz für zwei Register
0x94          sw  $a0, 4($sp) # sichere $a0
0x98          sw  $ra, 0($sp) # sichere $ra
0x9C          addi $t0, $0, 2
0xA0          slt  $t0, $a0, $t0 # a <= 1 ?
0xA4          beq  $t0, $0, else # nein: weiter bei else
0xA8          addi $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          addi $sp, $sp, 8 # Platz wieder freigeben
0xC8          mul  $v0, $a0, $v0 # n * fakultaet(n-1)
0xCC          jr   $ra # Rücksprung
```



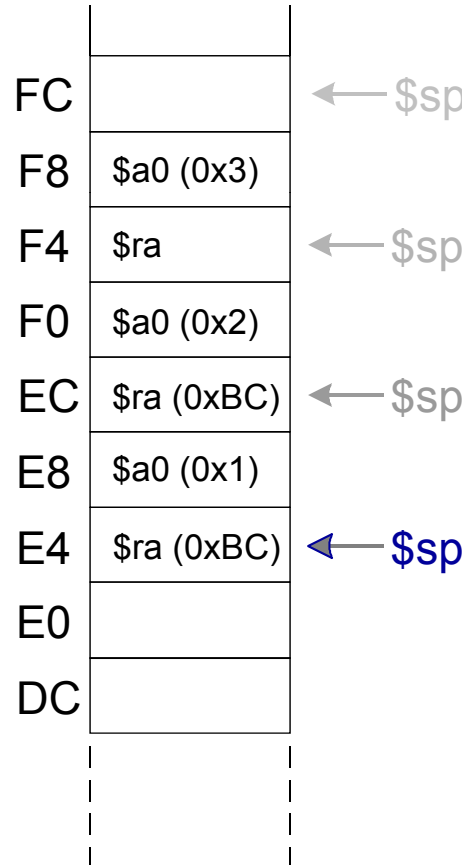
# Veränderung des Stacks bei rekursivem Aufruf



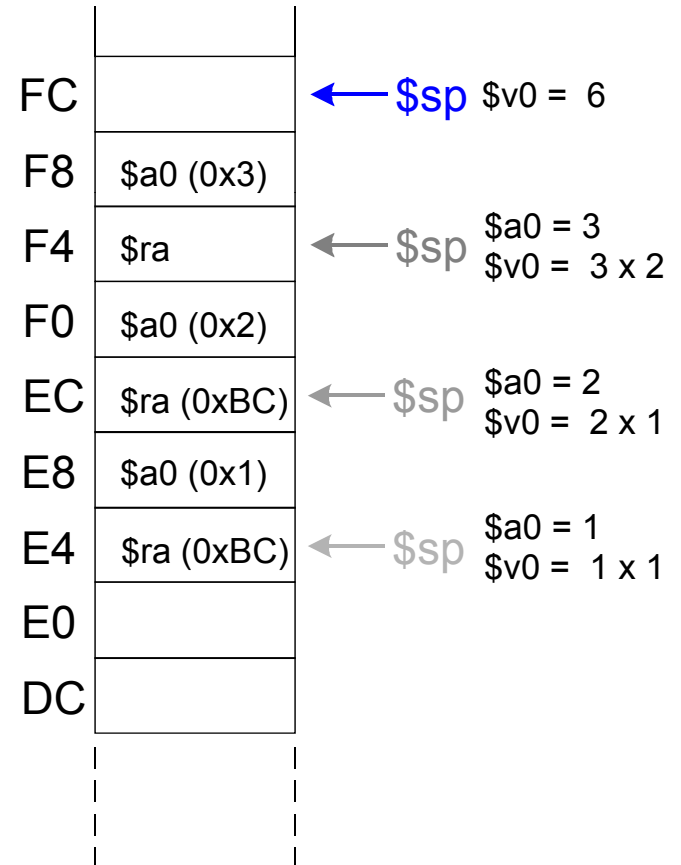
Adresse Daten



Adresse Daten



Adresse Daten



# Zusammenfassung: Prozeduraufruf



- Aufrufer
  - Lege Aufrufparameter (aktuelle Parameter) in  $\$a0-\$a3$  ab
  - Sichere zusätzlich benötigte Register auf Stack ( $\$ra$ , manchmal auch  $\$t0-t9$ )
    - Entsprechend Konvention über Erhaltung von Registern
  - `jal aufrufer`
  - 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`



## Wo kommen Operanden für Befehle her?

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



# Adressierungsarten

## Aus Register (*register operands*)

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

## Direktwert aus Instruktion (*immediate*)

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

## 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 =  $\$t1 - 25$

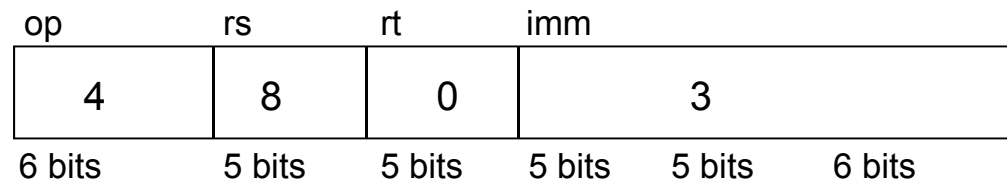
## Relativ zur nächsten Adresse im Programmzähler

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

### Assemblersprache

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

### Bitfelder in Instruktion



## Pseudodirekte Operanden

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

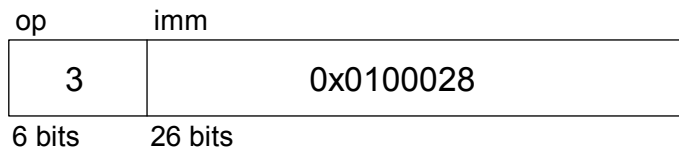
```
0x0040005C      jal      sum
...
0x004000A0  sum:  add      $v0, $a0, $a1
```

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

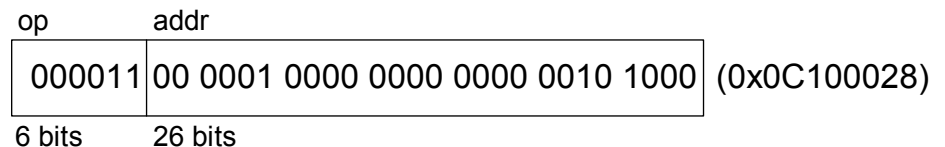
26b Feld in J-Instruktion 0000 0000 0100 0000 0000 0000 1010 0000 (0x0100028)

0 1 0 0 0 2 8

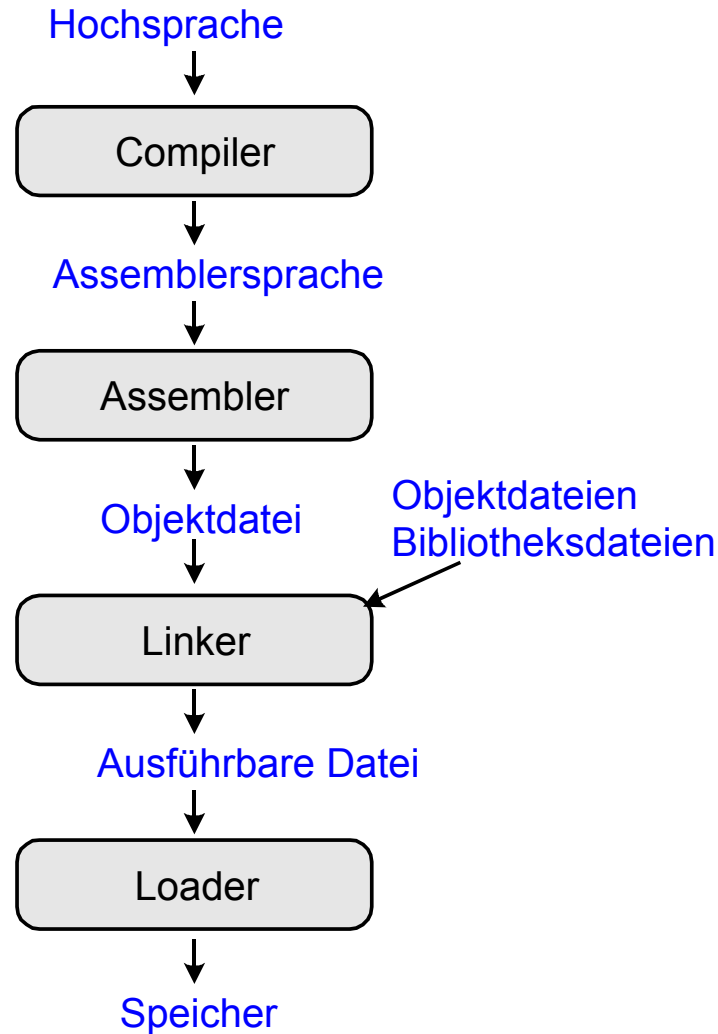
### Bitfelder in Instruktion



### Maschinencode



# Compilieren und Ausführen einer Anwendung





# Grace Hopper, 1906 - 1992



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

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

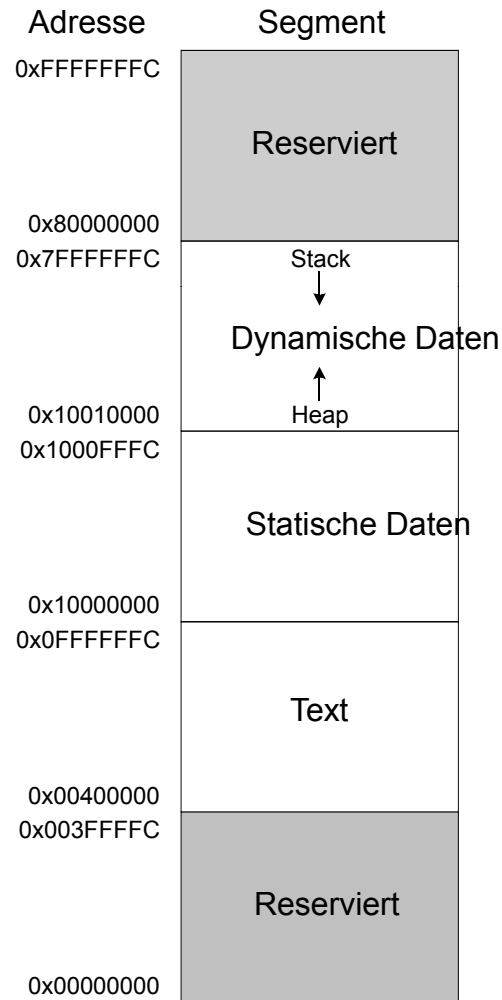


# Was muss im Speicher abgelegt werden?



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

# MIPS Speicherorganisation (*memory map*)



# Beispielprogramm in “C”



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

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

    return y;
}
```

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

# Beispielprogramm: MIPS Assemblersprache



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

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

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

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

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

.data
f: .space 4           # Direktiven für Assembler
g: .space 4           # jeweils ein Wort, initialisiert
y: .space 4           # auf den Wert 0
.text
main:
    addi $sp, $sp, -4 # Stack Frame anlegen
    sw   $ra, 0($sp)  # sichere $ra
    addi $a0, $0, 2   # $a0 = 2
    sw   $a0, f        # f = 2
    addi $a1, $0, 3   # $a1 = 3
    sw   $a1, g        # g = 3
    jal  sum           # Aufruf von sum
    sw   $v0, y        # y = sum()
    lw   $ra, 0($sp)  # stelle $ra wieder her
    addi $sp, $sp, 4  # stelle $sp wieder her
    jr   $ra          # Rückkehr ins Betriebssystem

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

# Beispielprogramm: Symboltabelle



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Symbol	Adresse

# Beispielprogramm: Symboltabelle



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

# Beispielprogramm: Ausführbare Datei

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

```

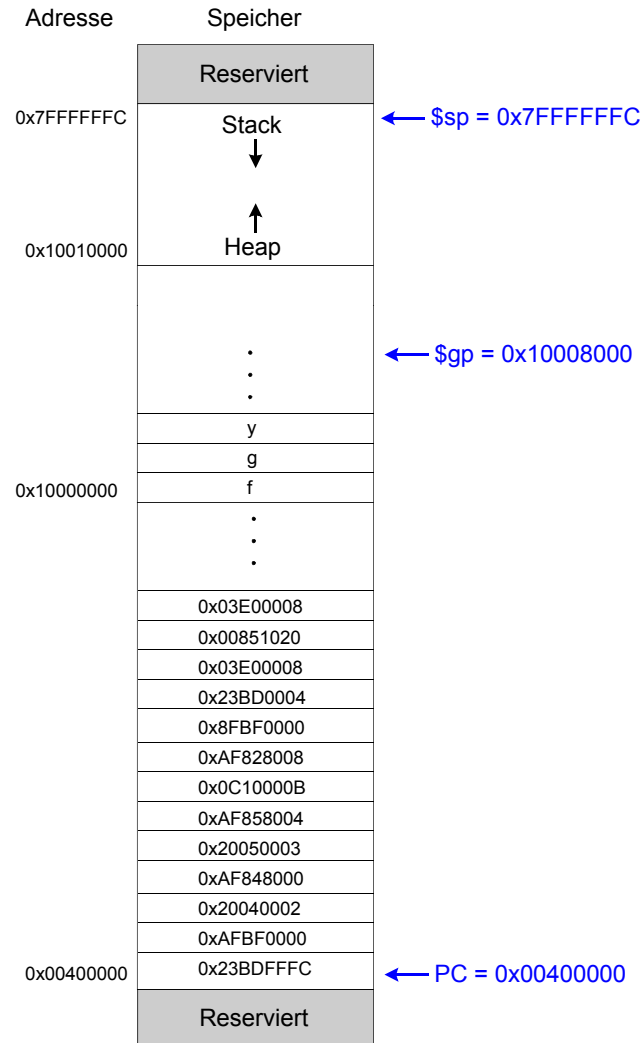
addi $sp, $sp, -4
sw  $ra, 0 ($sp)
addi $a0, $0, 2
sw  $a0, 0x8000 ($gp)
addi $a1, $0, 3
sw  $a1, 0x8004 ($gp)
jal  0x0040002C
sw  $v0, 0x8008 ($gp)
lw  $ra, 0 ($sp)
addi $sp, $sp, -4
jr  $ra
add $v0, $a0, $a1
jr  $ra
    
```

```

f
g
y
    
```



# Beispielprogramm im Speicher



# Dies und Das



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

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

# Beispiele für Pseudobefehle



Pseudobefehle	MIPS Befehle
<code>li \$s0, 0x1234AA77</code>	<code>lui \$s0, 0x1234</code> <code>ori \$s0, 0xAA77</code>
<code>mul \$s0, \$s1, \$s2</code>	<code>mult \$s1, \$s2</code> <code>mflo \$s0</code>
<code>clear \$t0</code>	<code>add \$t0, \$0, \$0</code>
<code>move \$s1, \$s2</code>	<code>add \$s2, \$s1, \$0</code>
<code>nop</code>	<code>sll \$0, \$0, 0</code>

# Ausnahmebehandlung (*exceptions*)

- Abweichen von der normalen **Ausführungsreihenfolge** von Befehlen
  - Beim Auftreten **außergewöhnlicher** Umstände (*exception*)
  - Automatischer Aufruf spezieller Prozedur: Ausnahmebehandlung (*exception handler*)
- **Auslösung** der Ausnahmebehandlung z.B. durch
  - Hardware, dann genannt **Interrupt** (z.B. Tippen einer Taste auf Tastatur)
  - Software, dann genannt **Trap** (z.B. Versuch der Ausführung einer unbekanntem Instruktion)
- Beim Auftreten der Ausnahme
  - **Grund** der Ausnahme wird gespeichert
  - **Sprung** zur Ausnahmebehandlung auf Adresse 0x80000180
  - Dann **Wiederaufnahme** der normalen Programmausführung



# Spezialregister für Ausnahmebehandlung

- **Außerhalb** des regulären Registerfeldes
  - **Cause**
    - Enthält den Grund für Ausnahme
  - **EPC** (Exception PC)
    - Enthält den regulären PC an dem die Aufnahme auftrat
- **EPC und Cause**: Nicht Bestandteil des “**eigentlichen**” MIPS-Prozessors
  - Ausgelagert in **Coprozessor** (unterstützt Hauptprozessor)
  - Genauer: **Coprozessor 0**
- **Datenaustausch** mit Coprozessor (hier nur lesen)
  - “**Move from Coprocessor 0**”
    - `mfc0 $t0, EPC`
  - Lädt Inhalt des Spezialregisters `EPC` in reguläres Register `$t0`
    - Analog auch für `Cause`

# Auslöser für Ausnahmen



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



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

- Ausnahmebehandlung:

- Speichere Register auf Stack
- Lese `Cause` Spezialregister

```
mfc0 $t0, Cause
```

- Bearbeite Ausnahme
- Stelle alle Register wieder her
- Springe zurück ins eigentlich laufende Programm

```
mfc0 $k0, EPC
```

```
jr $k0
```

# Vorzeichenbehaftete und –lose Befehle



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

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



# Addition und Subtraktion



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

# Multiplikation und Division



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

- **Vorzeichenbehaftet:** `mult`, `div`
- **Vorzeichenlos:** `multu`, `divu`

# Set Less Than

- **Vorzeichenbehaftet:** `slt`, `slti`
- **Vorzeichenlos:** `sltu`, `sltiu`
  - **Hinweis:** `sltiu` vorzeichenerweitert den Direktwert *vor* dem Vergleich mit dem Register

# Laden von 8b und 16b breiten Daten



- **Vorzeichenbehaftet:**
  - Vorzeichenerweiterung schmale Daten auf volle 32b Registerbreite
  - Load halfword: `lh`
  - Load byte: `lb`
- **Vorzeichenlos:**
  - Fülle schmale Daten mit Nullen auf volle 32b Registerbreite auf
  - Load halfword unsigned: `lhu`
  - Load byte: `lbu`



# Gleitkommabefehle

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

# Gleitkommabefehle



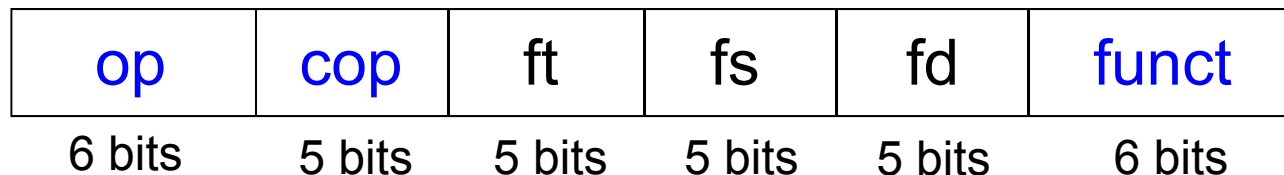
Namen	Registernummern	Zweck
\$fv0 - \$fv1	0, 2	Rückgabewerte
\$ft0 - \$ft3	4, 6, 8, 10	Temporäre Variablen
\$fa0 - \$fa1	12, 14	Prozedurargumente
\$ft4 - \$ft8	16, 18	Temporäre Variablen
\$fs0 - \$fs5	20, 22, 24, 26, 28, 30	Erhaltene Variablen

# Format für F-Typ Instruktionen



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

## F-Typ





# Weitere Gleitkommabefehle

- Setzt boole'sches **Spezialregister** bei Vergleichen : `fpcond`
  - Gleichheit: `c.seq.s`, `c.seq.d`
  - Kleiner-als: `c.lt.s`, `c.lt.d`
  - Kleiner-als-oder-gleich: `c.le.s`, `c.le.d`
  - Beispiel: `c.lt.s $fs1, $fs2`
- Bedingte **Verzweigung** abhängig von Spezialregister
  - `bc1f`: springt falls `fpcond = FALSE`
  - `bc1t`: springt falls `fpcond = TRUE`
  - Beispiel: `bc1f toosmall`
- Loads und Stores: jeweils **Single precision**
  - `lwc1: lwc1 $ft1, 42($s1)`
  - `swc1: swc1 $fs2, 17($sp)`
  - Double precision braucht je zwei Anweisungen



# Ausblick

- Bisher **Architektur**
  - Programmiersicht
- Nun **Mikroarchitektur**
  - Aufbau der zugrundeliegenden **Hardware**