

IT Infrastrukturen – Rechnerstrukturen

Thema 3: Befehle – die Sprache des Rechners

Prof. Dr.-Ing. Sebastian Schlesinger
Professur für Wirtschaftsinformatik

Dieser Foliensatz basiert auf den
Folien zu „Rechnerorganisation“ von
Prof. Dr. Ben Juurlink (TU Berlin) und
Prof. Dr. Paula Herber, WWU Münster

Nach diesem Kapitel sind Sie in der Lage:

- Die Entwurfsprinzipien eines **Befehlssatzes** (*Instruction Set Architecture*) am Beispiel des MIPS Befehlssatzes zu erläutern
- C-Programme in **MIPS-Assemblersprache** umzusetzen
- komplexere Programmierkonstrukte in Assembler zu realisieren
 - Kontrollfluss (**if**, **while**)
 - (rekursive) Funktionen
- **MIPS-Registerkonventionen** zu befolgen
- zu erklären was ein **Stack** ist und wofür er gebraucht wird
- ein Assemblerprogramm in **Maschinsprache** umzusetzen und umgekehrt
- verschiedene **Adressierungsarten** zu erklären



1. Einführung: MIPS Befehlssatz
2. Assemblersprache
 - Arithmetische Befehle
 - Lade- und Speicherbefehle
 - Logische Operationen
 - Kontrollbefehle
 - Multiplikation und Division
 - Zeichen und Zeichenfolgen
3. Funktionen und Prozeduren
 - Funktionsaufrufe und Stack
 - Registerkonventionen
 - Rekursion
4. Ein größeres Beispiel: Bubble Sort
5. Systemfunktionen
6. Überlauf-Behandlung
7. Floating Point Befehle
8. Maschinensprache
 - Darstellung von Befehlen im Rechner
 - Von-Neumann-Konzept
 - Adressierungsarten
9. Zusammenfassung



1. Einführung: MIPS Befehlssatz

2. Assemblersprache

- Arithmetische Befehle
- Lade- und Speicherbefehle
- Logische Operationen
- Kontrollbefehle
- Multiplikation und Division
- Zeichen und Zeichenfolgen

3. Funktionen und Prozeduren

- Funktionsaufrufe und Stack
- Registerkonventionen
- Rekursion

4. Ein größeres Beispiel: Bubble Sort

5. Systemfunktionen

6. Überlauf-Behandlung

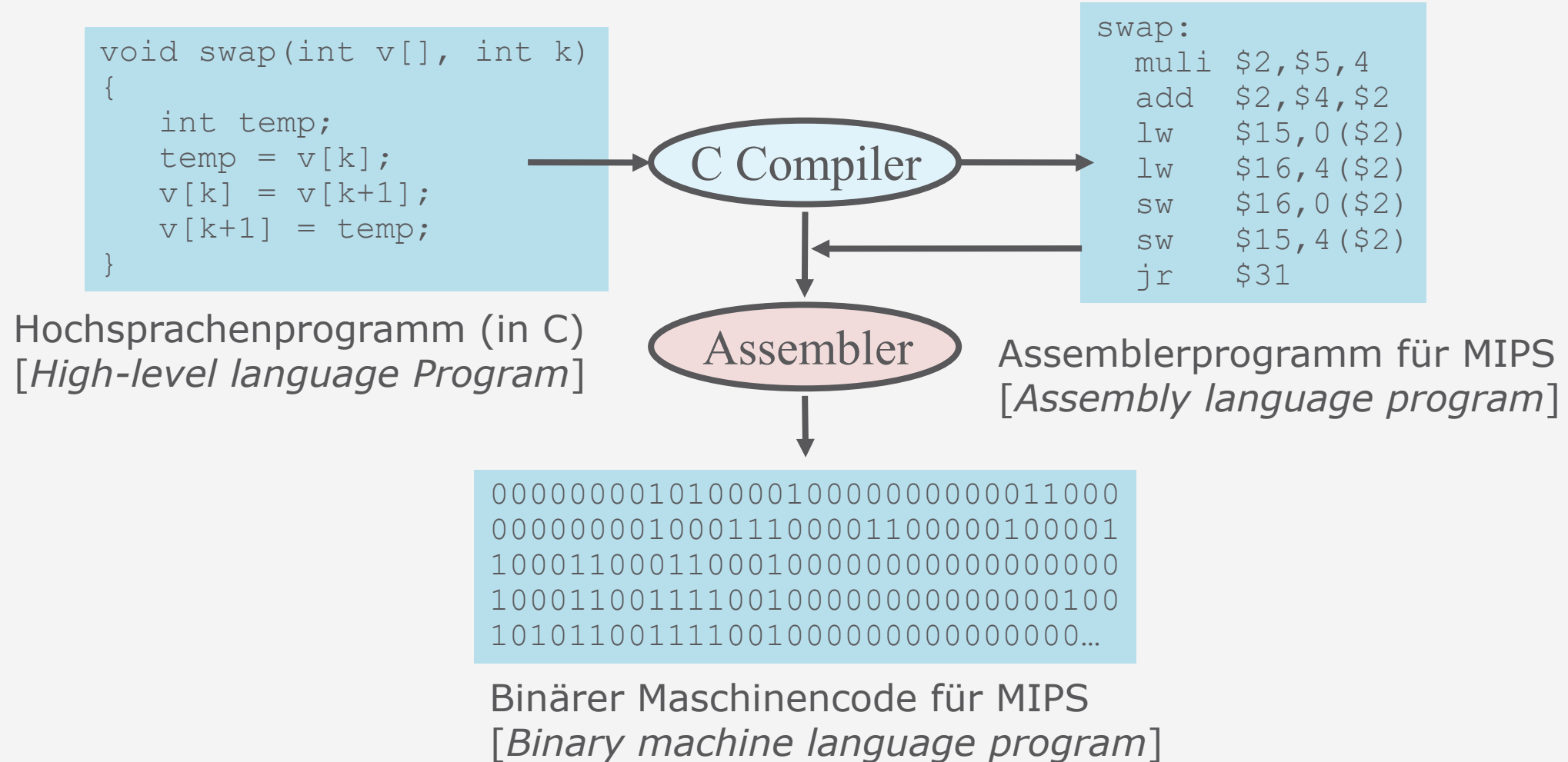
7. Floating Point Befehle

8. Maschinensprache

- Darstellung von Befehlen im Rechner
- Von-Neumann-Konzept
- Adressierungsarten

9. Zusammenfassung

Von Hochsprache zu Maschinensprache



- Eine sehr wichtige Abstraktion!
- **Schnittstelle** zwischen **Hardware** und **Software**: Die „Sprache“ des Rechners
- Standardisierung von Befehlen, Bitfolgen, u.s.w.
- verschiedene Implementierungen einer ISA möglich
- Viel **primitiver als höhere Programmiersprachen** (z.B.: kein **while**, kein **if**)
- Sehr **restriktiv** (z. B.: viele MIPS-Befehle haben genau 3 Operanden)

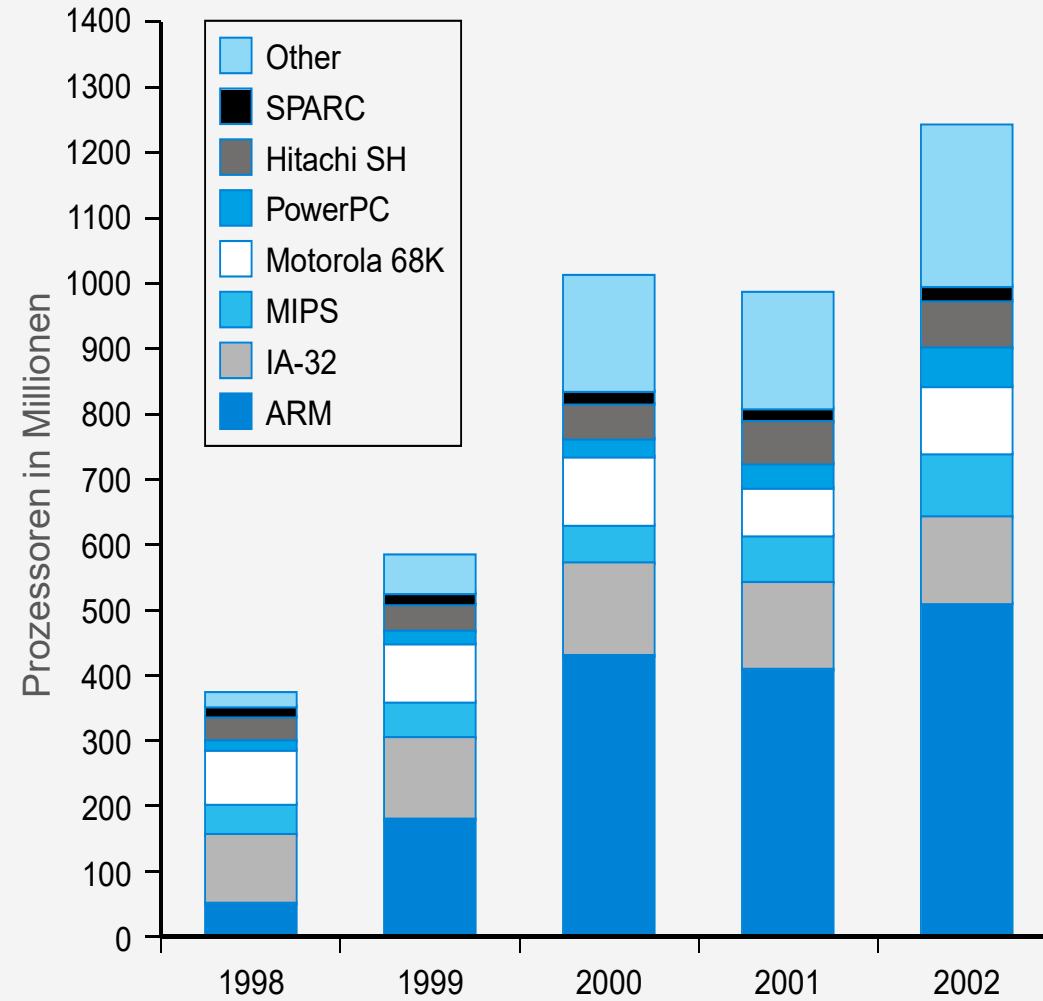
Ziel der **Rechnerarchitektur**: Eine **Sprache** zu finden, die

- das **Konstruieren** der **Hardware** und des **Compilers erleichtert** und dabei
- die **Leistung maximiert** und
- die **Kosten minimiert**.

MIPS



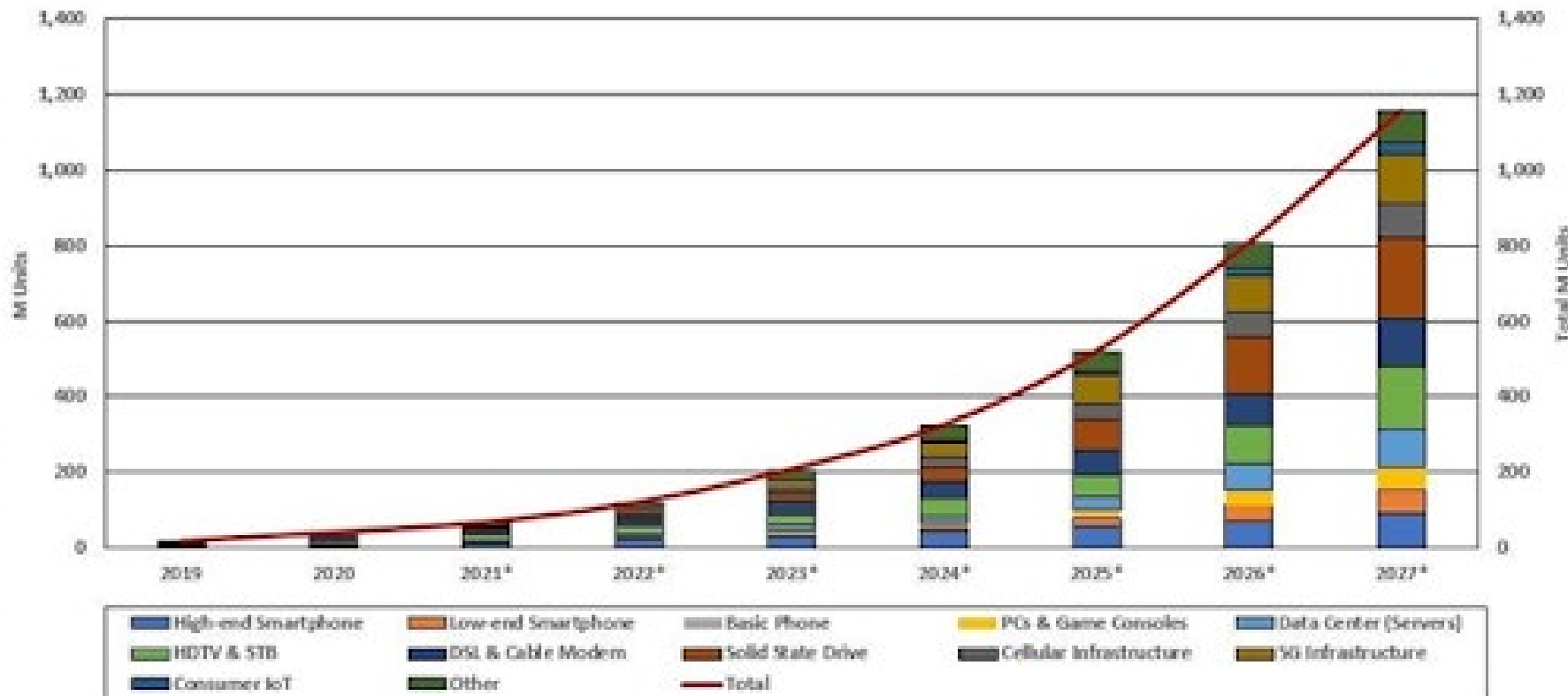
- MIPS = Microprocessor without Interlocked Pipeline Stages
- **MIPS-Befehlssatz**
 - entwickelt seit den 80er-Jahren (Stanford University)
 - wurde verwendet von NEC, Nintendo, Silicon Graphics, Sony,
 - 2013 übernommen von Imagination Technologies
- **Entwurfsprinzipien relevant für viele moderne Architekturen!**



MIPS vs. RISC-V



- RISC-V (= Reduced Instruction Set Computer "five") **basiert auf MIPS**
- offener Standard, seit 2010 an der UC Berkeley entwickelt (Patterson, Asanovic)





- **Arithmetische und Logische Befehle**
 - Integer
 - Gleitpunkt (*Floating Point*)
- **Datentransfer-Befehle**
 - Laden und speichern (*Load & Store*)
- **Kontrollbefehle**
 - Sprung (*Jump*)
 - bedingte Verzweigungen (*Conditional Branch*)
 - zur Unterstützung von Prozeduren (*Call & Return*)



1. Einführung: MIPS Befehlssatz
2. Assemblersprache
 - Arithmetische Befehle
 - Lade- und Speicherbefehle
 - Logische Operationen
 - Kontrollbefehle
 - Multiplikation und Division
 - Zeichen und Zeichenfolgen
3. Funktionen und Prozeduren
 - Funktionsaufrufe und Stack
 - Registerkonventionen
 - Rekursion
4. Ein größeres Beispiel: Bubble Sort
5. Systemfunktionen
6. Überlauf-Behandlung
7. Floating Point Befehle
8. Maschinensprache
 - Darstellung von Befehlen im Rechner
 - Von-Neumann-Konzept
 - Adressierungsarten
9. Zusammenfassung



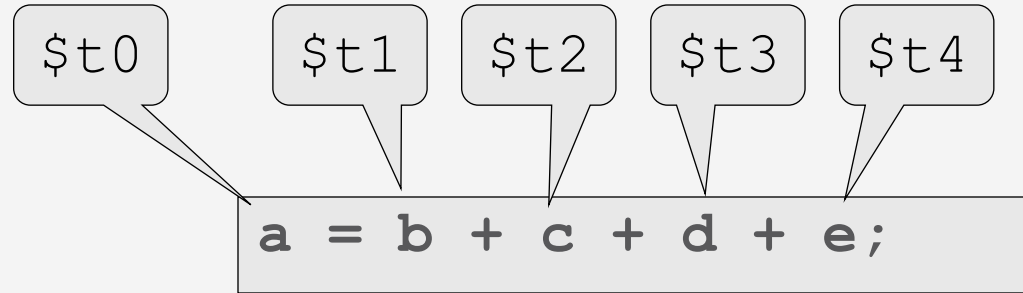
- Die meisten arithmetischen Befehle haben **drei Operanden**
- Folge der Operanden steht fest (**Ziel als erster Operand**)
- **Operanden sind Register**
- Beispiel: C/Java Anweisung: **$A = B + C;$**

MIPS-Code:

add \$s0, \$s1, \$s2

- **\$s0**, **\$s1** und **\$s2** sind Register, die der Compiler (oder Assembler-Programmierer) mit Variablen assoziiert
- Dieser Befehl **addiert den Inhalt von \$s1 und \$s2** und schreibt das **Ergebnis nach \$s0**.
- analog: **sub \$s0, \$s1, \$s2**

Arithmetische Befehle mit >3 Operanden



```
add    $t0, $t1, $t2    # a = b+c
add    $t0, $t0, $t3    # a = a+d
add    $t0, $t0, $t4    # a = a+e
```

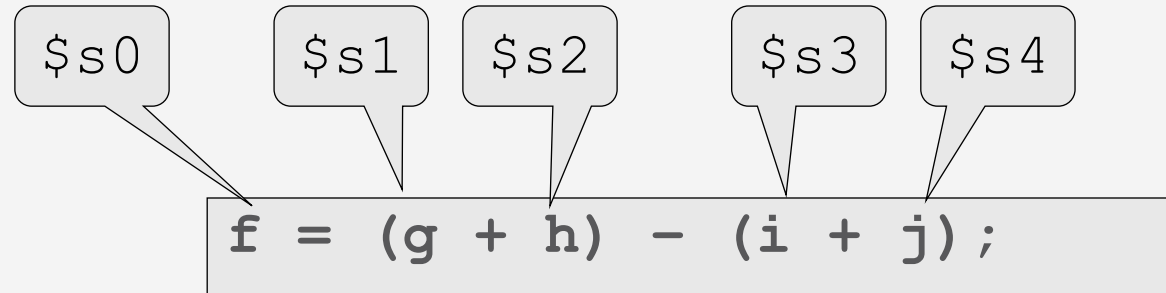
- Genau 3 Operanden entspricht Philosophie, die Hardware einfach zu halten
- Hardware für eine variable Anzahl an Operanden ist komplexer.

Entwurfsprinzip 1: ***Simplicity favors regularity***
(Regelmäßigkeit vereinfacht den Entwurf)

Komplexere Arithmetische Befehle



- Etwas komplexeres Beispiel:

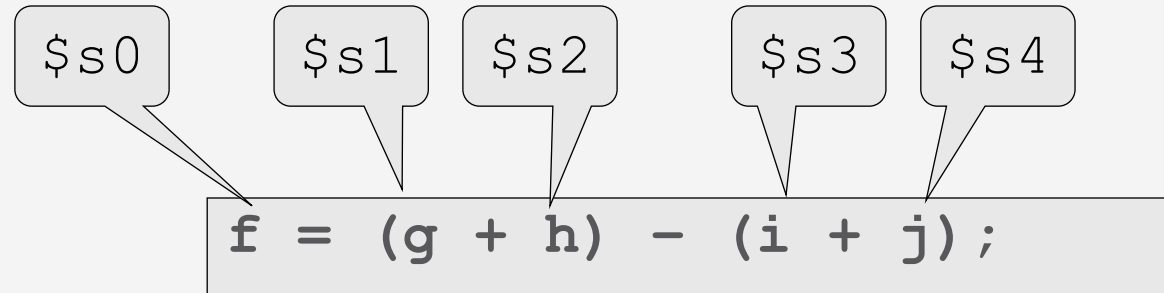


Wie lauten die Assemblerbefehle?

Komplexere Arithmetische Befehle



- Etwas komplexeres Beispiel:



Wie lauten die Assemblerbefehle?

```
add    $t0, $s1, $s2    # $t0 = g+h
add    $t1, $s3, $s4    # $t1 = i+j
sub    $s0, $t0, $t1    # f = $t0-$t1
```

- **Operand**: Objekt, auf das eine mathematische Funktion oder ein Operator angewendet wird
- MIPS kennt folgende Typen von Operanden
 - **Registeroperanden**
 - **Speicheroperanden**
 - Konstante oder **Direktoperanden**

MIPS Register



- MIPS verfügt über 32 Integer- (Fixpunkt-) Register.
- Register sind **32 Bit breit** (= ein **Wort** (*word*) in MIPS)
- In Assembler adressiert als **\$0, \$1, ..., \$31** oder mit den symbolischen Namen **\$zero, \$at, ..., \$ra**.
- Register 0 (**\$0** oder **\$zero**) ist immer Null.
- **\$at** (= assembler temporary) ist für den Assembler reserviert
- **\$v0, \$v1** : *values* (Ergebnisse)
- **\$a0, \$a1, ...** : *arguments*
- **\$t0, \$t1, ...** : *temporaries*
- **\$s0, \$s1, ...** : *saved registers*
- **\$ra** : *return address*

Reg. #	Name	Wert
0	\$zero	0
1	\$at	
2	\$v0	
3	\$v1	
4	\$a0	
...		
31	\$ra	

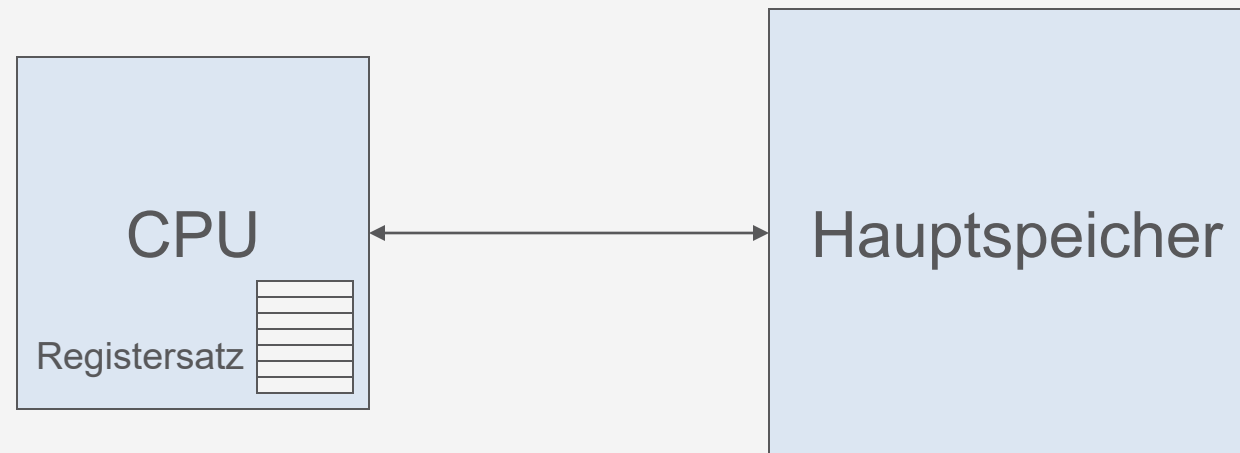
32 MIPS Register



- Wieso nur 32 Integer Register?
 - mehr passen nicht in das Befehlsformat (nur 5 Bit für Register-Adressen)
 - Eine große Anzahl von Registern kann zu einer längeren Taktzykluszeit führen.

Entwurfsprinzip 2: ***Smaller is faster*** (Kleiner ist schneller)

- Variablen und große Datenstrukturen wie Arrays (Felder) befinden sich im **Hauptspeicher**.



- Zum Transport von Daten zwischen Hauptspeicher und Register gibt es **Datentransfer-Befehle** (*data transfer instructions*)

Speicherorganisation: Byte-Adressierung



- Hauptspeicher kann als ein großes Array von Bytes betrachtet werden.
- Eine Speicheradresse ist ein Index in diesem Array.
- „Byte-Adressierung“ bedeutet, dass mit jeder Adresse genau ein Byte adressiert wird.

Adresse	Inhalt
0	8 Datenbits
1	8 Datenbits
2	8 Datenbits
3	8 Datenbits
4	8 Datenbits
5	8 Datenbits
6	8 Datenbits
7	8 Datenbits
.

Speicherorganisation: Wörter



- Die meisten Programme benutzen **Wörter** (words) statt Bytes.
- Für MIPS entspricht ein Wort 32 Bits oder 4 Bytes.
- Der Hauptspeicher kann betrachtet werden als
 - ein **Array von Bytes mit Adressen 0, 1, 2, 3, ...**
 - ein **Array von Wörtern mit Adressen 0, 4, 8, 12, ...**

Adresse	Inhalt
0	32 Datenbits
4	32 Datenbits
8	32 Datenbits
12	32 Datenbits
...	...

- Die **Byte-Reihenfolge** (*Byte-Order* oder *Endianness*) bezeichnet, welches Byte zuerst gespeichert wird.
- **Big-Endian**: das Byte mit den höchstwertigen Bits (d. h. die signifikantesten Stellen) wird zuerst gespeichert, d. h. an der kleinsten Speicheradresse (MIPS)
- Beispiel: 0xaabbccdd

Speicheradresse	992	993	994	995
Inhalt	0xaa	0xbb	0xcc	0xdd

- **Little-Endian**: das Byte mit den niederwertigsten Bits wird an der kleinsten Speicheradresse gespeichert (x86/IA32)

Speicheradresse	992	993	994	995
Inhalt	0xdd	0xcc	0xbb	0xaa



Alignment

- In MIPS müssen Wörter bei Adressen beginnen, die ein Vielfaches von 4 sind.
- Wird als **Ausrichtung an Wortgrenzen** (*alignment restriction*) bezeichnet.
- **Was sind die letzten (niederwertigsten) 2 Bits einer Wortadresse?**

Adresskalkulation

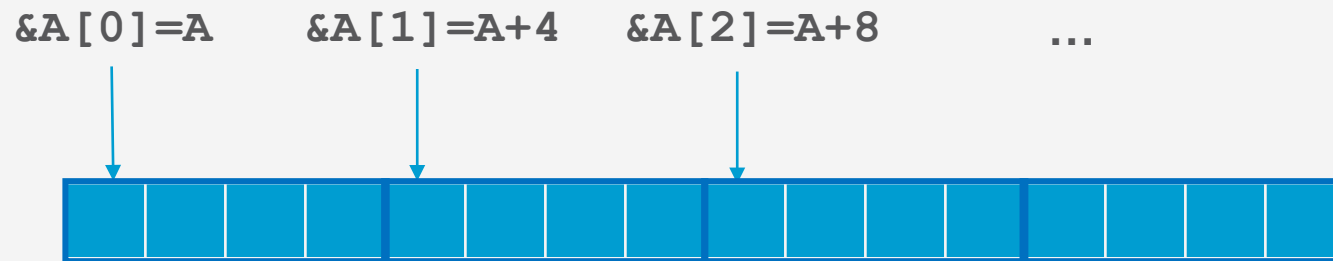
- **A** ist die **Basisadresse** oder **Startadresse** eines Arrays.
- **Was ist die Adresse von A[2]?**

Alignment und Adresskalkulation für Arrays



Hochschule für
Wirtschaft und Recht Berlin
Berlin School of Economics and Law

- Die letzten 2 Bits einer Wortadresse sind immer 00.
(... 0000, ... 0100, ... 1000, ... 1100, ...)
- Was ist die Adresse von $A[2]$?**
- A ist die Basisadresse oder Startadresse des Arrays.
- Wenn A ein Array von Bytes ist: $A + 2$
- Wenn A ein Array von Wörtern ist: $A + 2 * 4 = A + 8$



Adresse	Inhalt
0	32 Datenbits
4	32 Datenbits
8	32 Datenbits
12	32 Datenbits
...	...

$\&A[0]$ ist (C-)Abkürzung
für „Adresse von“ $A[0]$

- Adresse von $A[i] = A + i * \text{Elementgröße}$



1. Einführung: MIPS Befehlssatz
2. Assemblersprache
 - Arithmetische Befehle
 - Lade- und Speicherbefehle
 - Logische Operationen
 - Kontrollbefehle
 - Multiplikation und Division
 - Zeichen und Zeichenfolgen
3. Funktionen und Prozeduren
 - Funktionsaufrufe und Stack
 - Registerkonventionen
 - Rekursion
4. Ein größeres Beispiel: Bubble Sort
5. Systemfunktionen
6. Überlauf-Behandlung
7. Floating Point Befehle
8. Maschinensprache
 - Darstellung von Befehlen im Rechner
 - Von-Neumann-Konzept
 - Adressierungsarten
9. Zusammenfassung



- **Ladebefehl:** *load word*

```
lw $t0, 12($s1) # $t0 = Mem[$s1+12]
```

- Lädt den Inhalt des Speichers von **Mem[\$s1+12]** in das Register **\$t0**
- Konstante 12 wird *Offset* genannt.

- **Speicherbefehl:** *store word*

```
sw $t0, 12($s1) # Mem[$s1+12] = $t0
```

- Speichert den Inhalt des Registers **\$t0** nach **Mem[\$s1+12]**
- Bemerke: Ziel steht am Ende

Datentransfer-Befehle: Beispiel



- *C/Java*: `A[12] = A[8] + c;`
 - Basisadresse `A` in `$s2`
 - `c` in `$s3`
 - `A` ist ein Array von Wörtern
 - `$t0` zum Zwischenspeichern (`$t0 ... $t7`: temporaries)

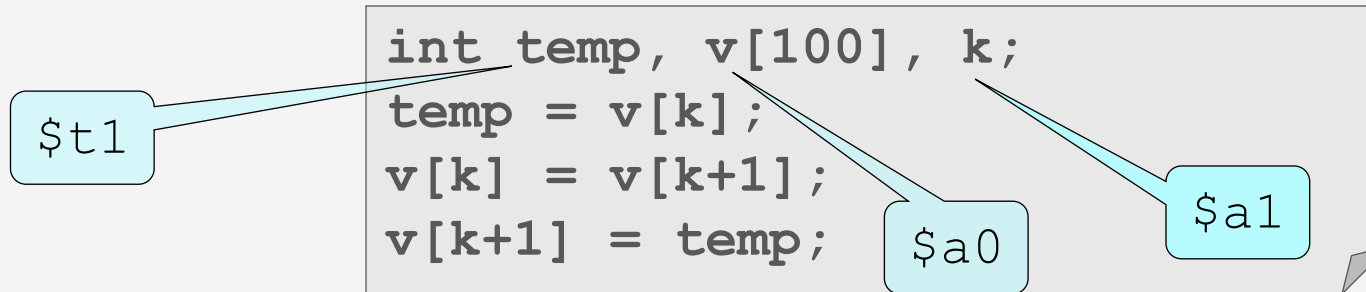
- *MIPS*:

```
lw    $t0, 32($s2)    # $t0 = A[8], &A[8] = A + 4*8 = A + 32
add   $t0, $t0, $s3    # $t0 += c
sw    $t0, 48($s2)    # a[12] = $t0, &A[12] = A + 4*12 = A + 48
```

Größeres Beispiel



- C/Java:



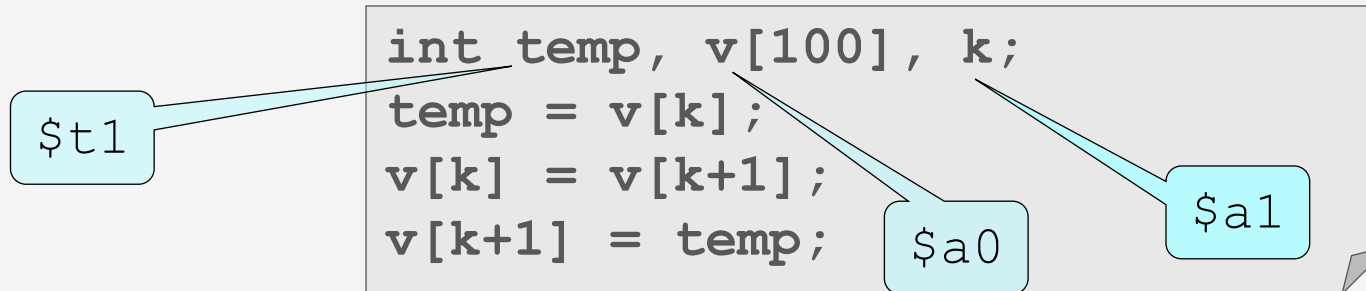
Übersetzen Sie in MIPS-Assembler!

Hinweis: Verwenden Sie mehrfache Addition statt Multiplikation.

Größeres Beispiel



- C/Java:



Übersetzen Sie in MIPS-Assembler!

- Adresse von $A[i] = A + i * \text{Elementgröße} \Rightarrow \&v[k] = v + k * 4$
- MIPS:

```
add    $t0,$a1,$a1 # $t0 = 2*k
add    $t0,$t0,$t0 # $t0 = 4*k
add    $t0,$a0,$t0 # $t0 = $a0 + 4*k = &v[k]
lw     $t1,0($t0)  # $temp = v[k]
lw     $t2,4($t0)  # $t2 = v[k+1]
sw     $t2,0($t0)  # v[k] = $t2 = v[k+1]
sw     $t1,4($t0)  # v[k+1] = $temp
```

Konstanten oder Direktoperanden



- In Programmen werden häufig (kleine) Konstanten verwendet.

- Beispiele:

```
a = a+5;
```

```
i++;
```

```
i < 10
```

- Lösungen:

- Konstanten aus **Hauptspeicher** laden
- **hardwired Register** (wie `$zero`) für Konstanten wie 1 erstellen
- **spezielle Befehle** für Addition kleiner Konstanten (Java bytecode)
- Versionen der Befehle bereitstellen, bei denen ein Operand eine Konstante ist (**Direktooperand**).

- MIPS-Befehl: *add immediate* („addiere direkt“): **addi \$s1,\$s2,4**



Direktooperand



Entwurfsprinzip 3: *Make the common case fast* (Mach den häufig vorkommenden Fall schnell)

- Konstanten werden häufig als Operanden verwendet
- Durch Verwendung von Konstanten in Befehlen können diese schneller ausgeführt werden, als wenn sie erst aus dem Hauptspeicher geladen werden müssen.

MIPS-Architektur bis hier / 1



MIPS-Operanden		
Name	Beispiel	Anmerkungen
32 Register	$\$s0, \$s1, \dots$ $\$t0, \$t1, \dots$	<ul style="list-style-type: none">• Speicherort für schnellen Zugriff• Bei MIPS müssen Daten für arithmetische Operationen in Registern stehen.
2^{30} Speicherwörter	Mem[0], Mem[4], ..., Mem[$2^{32}-4$]	<ul style="list-style-type: none">• Zugriff: Datentransfer-Befehle.• MIPS verwendet Byte-Adressen.• Im Hauptspeicher werden Datenstrukturen, Arrays und ausgelagerte Register gespeichert.

MIPS-Architektur bis hier / 2



MIPS-Assemblersprache				
Kategorie	Befehl	Beispiel	Bedeutung	Anmerkungen
Arithmetische Befehle	add	add \$s1,\$s2,\$s3	\$s1 = \$s2+\$s3	Drei Operanden; Daten in Registern
	subtract	sub \$s1,\$s2,\$s3	\$s1 = \$s2-\$s3	
	add immediate	addi \$s1,\$s2,100	\$s1 = \$s2+100	Addieren von Konstanten
Daten-transfer-Befehle	load word	lw \$s1,100(\$s2)	\$s1 = Mem[\$s2+100]	Lade Daten vom Hauptspeicher in ein Register
	store word	sw \$s1,100(\$s2)	Mem[\$s2+100] = \$s1	Speichere Daten von einem Register in den Hauptspeicher



1. Einführung: MIPS Befehlssatz
2. Assemblersprache
 - Arithmetische Befehle
 - Lade- und Speicherbefehle
 - Logische Operationen
 - Kontrollbefehle
 - Multiplikation und Division
 - Zeichen und Zeichenfolgen
3. Funktionen und Prozeduren
 - Funktionsaufrufe und Stack
 - Registerkonventionen
 - Rekursion
4. Ein größeres Beispiel: Bubble Sort
5. Systemfunktionen
6. Überlauf-Behandlung
7. Floating Point Befehle
8. Maschinensprache
 - Darstellung von Befehlen im Rechner
 - Von-Neumann-Konzept
 - Adressierungsarten
9. Zusammenfassung



Logische Operation	C/Java Operator	MIPS-Befehl
Linksschieben (<i>shift left logical</i>)	<<	sll
Rechtsschieben (<i>shift right logical</i>)	>>	srl
Bitweise UND	&	and, andi
Bitweise ODER		or, ori
Bitweise NICHT	~	nor mit \$0

Logische Operationen: Beispiel



- Beispiel: Linksschieben (*shift left logical*)
 - MIPS-Befehl: `sll $t2,$s0,4` # $\$t2 = \$s0 \ll 4$
 - $\$s0 = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001_B = 9_D$

Was steht nach der Operation in \$t2?

Logische Operationen: Beispiel



- Beispiel: Linksschieben (*shift left logical*)

- MIPS-Befehl: `sll $t2,$s0,4` # $\$t2 = \$s0 \ll 4$

- $\$s0 = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001_B = 9_D$

- Was steht nach der Operation in \$t2?

- $\$t2 = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001\ 0000_B = 144_D = 9 \cdot 16$

- Linksschieben wird verwendet um mit Zweierpotenz zu multiplizieren

- Schieben um i Bits nach links = multiplizieren mit 2^i
 - Schiebeoperation kostet i. A. weniger Zeit als Multiplikation
 - wird oft verwendet um Adressen von Arrayelementen zu berechnen

Linksschieben statt Multiplizieren



- C/Java:

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

\$a0

\$a1

- Vorhin:

```
add    $t0,$a1,$a1 # $t0 = 2*k
add    $t0,$t0,$t0 # $t0 = 4*k
add    $t0,$a0,$t0 # $t0 = $a0+4*k = &v[k]
lw     $t1,0($t0)  # $t1 = v[k]
```

- Mit Linksschieben

```
sll    $t0,$a1,2   # $t0 = 4*k
add    $t0,$a0,$t0 # $t0 = $a0+4*k = &v[k]
lw     $t1,0($t0)  # $t1 = v[k]
```



- Eine UND-Verknüpfung erzwingt an den Stellen eine 0, an denen sich im Bitmuster eine 0 befindet (Maske):

- MIPS-Befehl: `andi $t2,$s0,15` # $\$t2 = \$s0 \ \& \ 15$
 $\$s0 = 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 1001 \ 1001_B = 153_D$
 $15_D = 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 1111_B$
 $\$t2 = 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 1001_B = 9_D$

- Eine ODER-Verknüpfung ergibt eine 1, wenn *einer* der Operandenbits eine 1 aufweist (Bits „setzen“):

- MIPS-Befehl: `ori $t2,$s0,15` # $\$t2 = \$s0 \ | \ 15$
 $\$s0 = 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 1001 \ 1001_B = 153_D$
 $15_D = 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 1111_B$
 $\$t2 = 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 1001 \ 1111_B = 159_D$

Logische Operationen: NOT



- MIPS hat **keine NICHT-Operation**
- Stattdessen **NOR** (NICHT ODER)
 - nur 1 wenn beide Operanden 0
- Entspricht NICHT wenn ein Operand 0 ist:
 - $A \text{ NOR } 0 = \text{NOT}(A \text{ OR } 0) = \text{NOT } A$
 - Register `$0 = $zero` ist immer Null.

A	B	A nor B
0	0	1
0	1	0
1	0	0
1	1	0

32-Bit Direktoperanden



- MIPS Direktoperanden sind 16 Bit breit.
 - in der Regel sind Konstanten kurz
- Größere Konstanten (32-Bit):
 - *load upper immediate* (**lui**) zum Laden der oberen Hälfte
 - **ori** zum Setzen der unteren Hälfte
- Beispiel: lade $16.711.935 = 255 \cdot 2^{16} + 255$

lui \$t0,255

\$t0 =

0000 0000 1111 1111	0000 0000 0000 0000
---------------------	---------------------

ori \$t0,\$t0,255

\$t0 =

0000 0000 1111 1111	0000 0000 1111 1111
---------------------	---------------------

- 16-Bit Direktoperanden (*Immediates*) werden für Arithmetik auf 32 Bit erweitert.
- Dazu muss das MSB (**sign bit**) in die anderen Bitpositionen kopiert werden
- Beispiel (4-Bit → 8-Bit):

0010 -> 0000 0010

1010 -> 1111 1010

$$=-8+2=-6_D \qquad =-128+64+32+16+8+2=-6_D$$

- Dies wird **Vorzeichenerweiterung** (*sign extension*) genannt.
- **addi** macht Vorzeichenerweiterung
- **andi** und **ori** **nicht**



1. Einführung: MIPS Befehlssatz
2. Assemblersprache
 - Arithmetische Befehle
 - Lade- und Speicherbefehle
 - Logische Operationen
 - Kontrollbefehle
 - Multiplikation und Division
 - Zeichen und Zeichenfolgen
3. Funktionen und Prozeduren
 - Funktionsaufrufe und Stack
 - Registerkonventionen
 - Rekursion
4. Ein größeres Beispiel: Bubble Sort
5. Systemfunktionen
6. Überlauf-Behandlung
7. Floating Point Befehle
8. Maschinensprache
 - Darstellung von Befehlen im Rechner
 - Von-Neumann-Konzept
 - Adressierungsarten
9. Zusammenfassung



1. Bedingte Verzweigungen (*conditional branches*)

- *Branch if equal* („verzweige, wenn gleich“):
`beq $t0,$t1,label # if ($t0==$t1) goto label`
- *Branch if not equal* („verzweige, wenn nicht gleich“):
`bne $t0,$t1,label # if ($t0!=$t1) goto label`

2. Unbedingte Verzweigungen (*unconditional branches*)

- *jump* („Sprung“)
`j label # goto label`

Übersetzung einer *If-then*-Anweisung



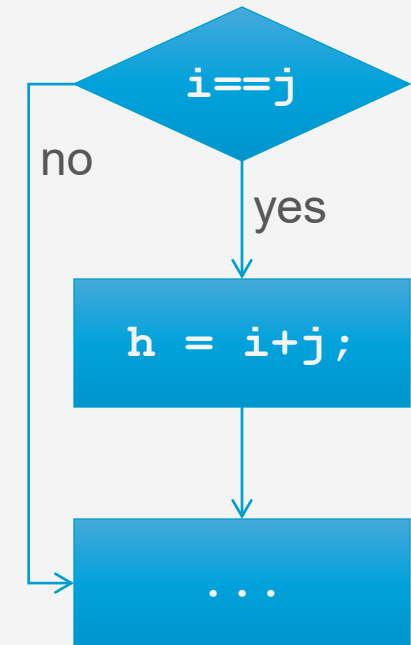
- C/Java:

```
if (i==j) h = i + j;  
...
```

Registers: \$s0, \$s1, \$s2

- MIPS: (1. Versuch)

```
beq $s0,$s1,then    # if (i==j) goto then  
j endif  
then:               add $s2,$s0,$s1    # h = i+j  
endif:              ...
```



- Labels **then** und **endif** werden vom Assembler in Adressen übersetzt.

Übersetzung einer *If-then*-Anweisung

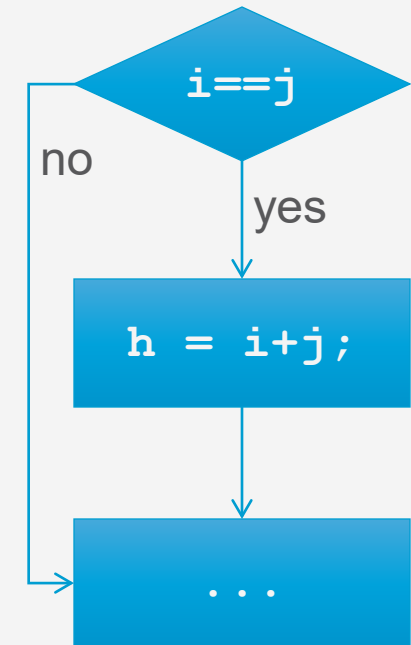


- C/Java:

```
if (i==j) h = i + j;  
...
```

- MIPS: (besser)

```
        bne $s0,$s1,endif    # if (i!=j) goto endif  
        add $s2,$s0,$s1      # h = i+j  
endif:  ...
```

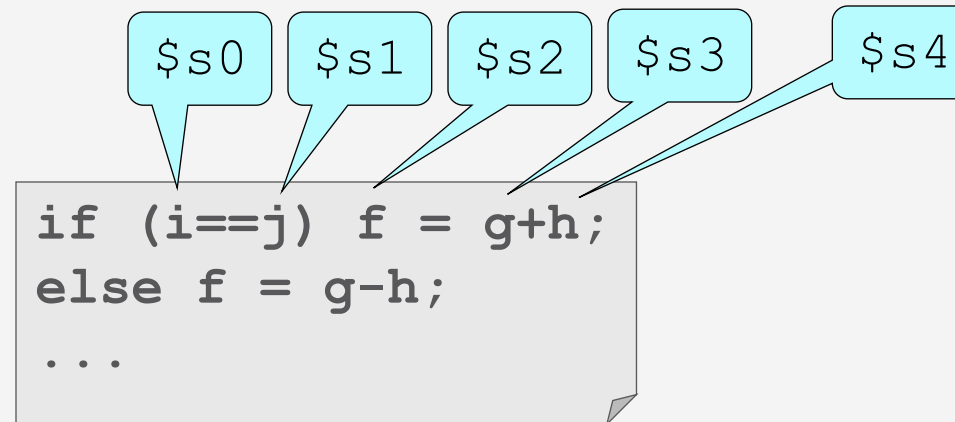


- Effizienter zu prüfen, ob gegenteilige Bedingung erfüllt ist!

Übersetzung einer *If-then-else*-Anweisung

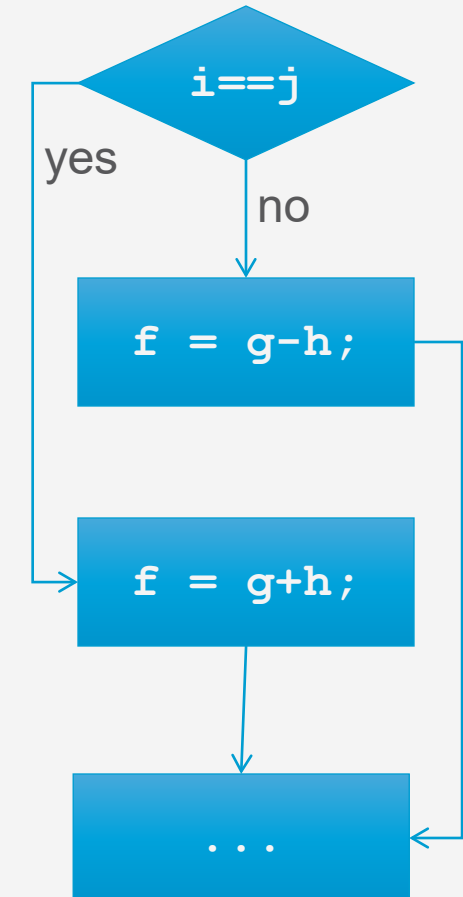


■ C/Java:



■ MIPS:

```
        bne    $s0,$s1,else    # if (i!=j) goto else
        add    $s2,$s3,$s4     # f = g+h (if-part)
        j      endif          # goto endif
else:    sub    $s2,$s3,$s4     # f = g-h (else-part)
endif:  ...
```



Übersetzung einer *While*-Schleife

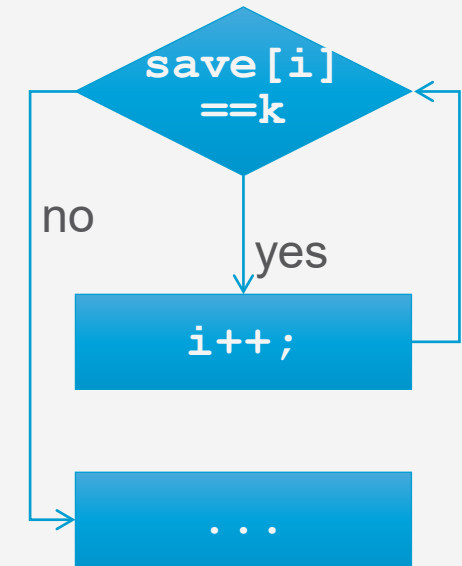


- C/Java:

```
while (save[i]==k) i++;  
...
```

\$s2 \$s0 \$s1

- Übersetzen Sie in MIPS Assembler!



Übersetzung einer *While*-Schleife



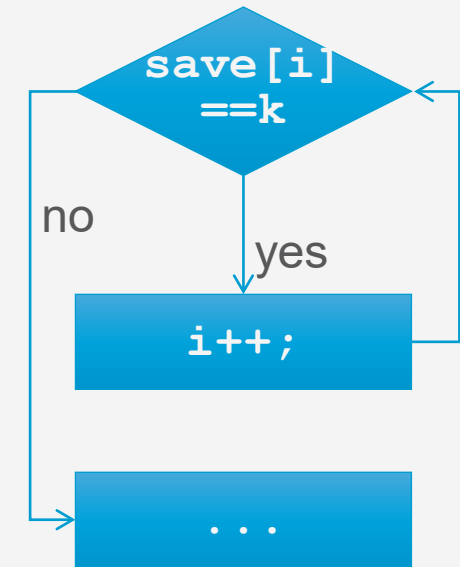
- C/Java:

```
while (save[i]==k) i++;  
...
```

Registers: $\$s2$ (points to `save`), $\$s0$ (points to `i`), $\$s1$ (points to `k`)

- Übersetzen Sie in MIPS Assembler!

```
while:    sll    $t0,$s0,2           # $t0 = 4*i  
          add    $t0,$s2,$t0        # $t0 = &save[i]  
          lw     $t0,0($t0)         # $t0 = save[i]  
          bne    $t0,$s1,endwhile   # if (save[i]!=k)  
                                              # goto endwhile  
          addi   $s0,$s0,1          # i++  
          j      while             # goto while  
endwhile: ...
```



Kleiner oder größer?



- Wie steht's mit z. B.
`if (a<0) a = -a;`
- MIPS-Befehlssatz enthält *kein branch-on-less-than*, da es die Taktzykluszeit verlängern würde (später mehr dazu).
- Verwende „Setze auf 1, wenn kleiner“ (*set on less than*):
`slt $t0,$s0,$s1 # $t0 = ($s0<$s1)`
`# $t0 = 1 wenn $s0<$s1, sonst $t0 = 0`
`slti $t0,$s0,10 # $t0 = ($s0<10)`
- `slt` und `slti` sind *keine* Verzweigungen

- Mithilfe **slt**, **beq**, **bne** und Register 0 können alle relativen Bedingungen ($=$, \neq , $<$, \leq , $>$, \geq) gebildet werden.
- Beispiel (*branch on less or equal*):
`ble $s1,$s2,label # if ($s1<=$s2) goto label`
- Echte MIPS-Befehle:
`slt $at,$s2,$s1 # $at = ($s2 < $s1)`
`beq $at,$zero,label # if ($at == 0) [$s2 >= $s1]`
`# goto label`
- **blt**, **ble**, **bgt**, **bge**, ... werden vom Assembler als **Pseudo-Befehle** (*pseudo-instructions*) akzeptiert und zu echten MIPS-Befehlen übersetzt.
- Assembler braucht ein Register dazu: Register **\$at** (*assembler temporary*)

Vorzeichenbehaftete vs. vorzeichenlose Vergleiche



- MIPS bietet 2 Versionen für den *set-on-less-than*-Befehl an:
 - **slt** und **slti** arbeiten mit vorzeichenbehafteten Integers.
 - **sltu** und **sltiu** arbeiten mit vorzeichenlosen Integers.

- Beispiel:

```
$s0 = 1111 1111 1111 1111 1111 1111 1111 1111  
     = -1 (signed),  $2^{32}-1$  (unsigned)
```

```
$s1 = 0000 0000 0000 0000 0000 0000 0000 0001  
     = 1 (signed und unsigned)
```

```
slt  $t0,$s0,$s1  # $t0 = 1 (wahr[true])  
sltu $t1,$s0,$s1  # $t1 = 0 (falsch[false])
```

For-Schleifen

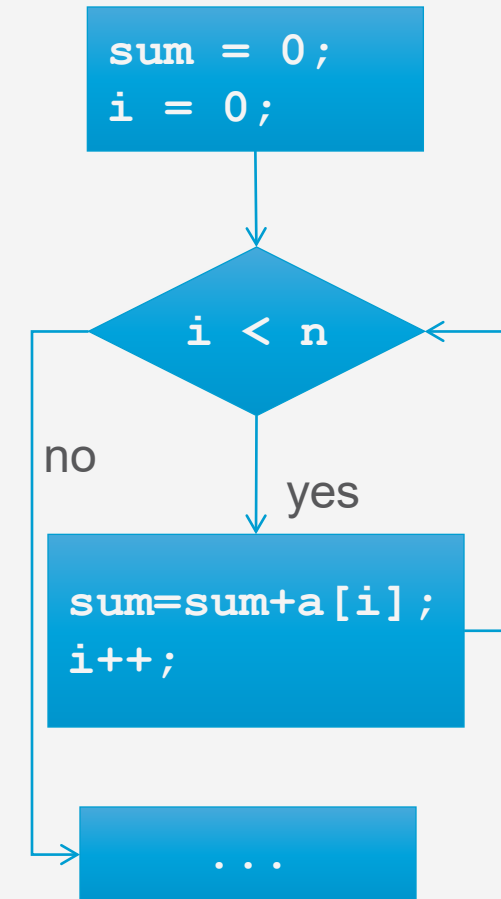


- C/Java:

```
sum = 0;  
for (i=0; i<n; i++)  
    sum = sum + a[i];
```

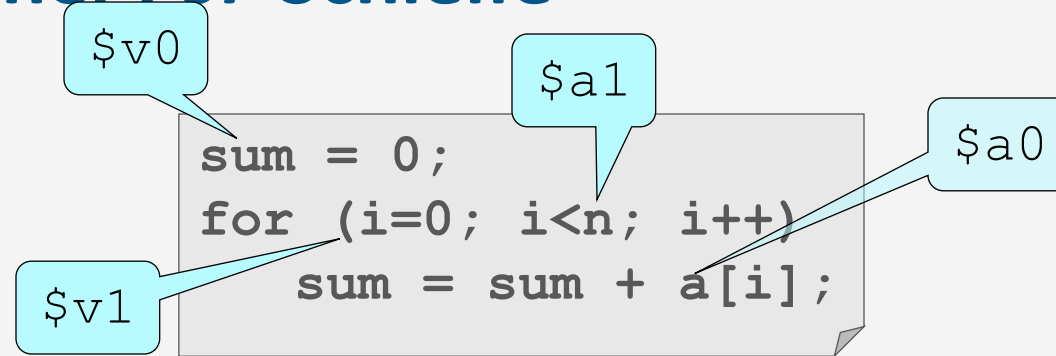
- Ist äquivalent zu:

```
sum = 0;  
i = 0;  
while (i<n) {  
    sum = sum + a[i];  
    i++;  
}
```



Übersetzung einer *For*-Schleife

- C/Java:



- MIPS:



**Übersetzen Sie in
MIPS-Assembler!**

Übersetzung einer *For*-Schleife



■ C/Java:

```
sum = 0;  
for (i=0; i<n; i++)  
    sum = sum + a[i];
```

Annotations:
- `sum` is associated with register `$v0`.
- `i` is associated with register `$v1`.
- `n` is associated with register `$a1`.
- `a` is associated with register `$a0`.

**Übersetzen Sie in
MIPS-Assembler!**

■ MIPS:

```
for:      add    $v0,$zero,$zero      # sum = 0  
          add    $v1,$zero,$zero      # i = 0  
          bge    $v1,$a1,endfor        # if (i>=n) goto endfor  
          sll    $t0,$v1,2             # $t0 = 4*i  
          add    $t0,$a0,$t0           # $t0 = a+4*i = &a[i]  
          lw     $t0,0($t0)            # $t0 = a[i]  
          add    $v0,$v0,$t0           # sum = sum+a[i]  
          addi   $v1,$v1,1             # i++  
          j      for                  # goto for  
endfor:   ...
```



1. Einführung: MIPS Befehlssatz
2. Assemblersprache
 - Arithmetische Befehle
 - Lade- und Speicherbefehle
 - Logische Operationen
 - Kontrollbefehle
 - Multiplikation und Division
 - Zeichen und Zeichenfolgen
3. Funktionen und Prozeduren
 - Funktionsaufrufe und Stack
 - Registerkonventionen
 - Rekursion
4. Ein größeres Beispiel: Bubble Sort
5. Systemfunktionen
6. Überlauf-Behandlung
7. Floating Point Befehle
8. Maschinensprache
 - Darstellung von Befehlen im Rechner
 - Von-Neumann-Konzept
 - Adressierungsarten
9. Zusammenfassung



- Das Multiplizieren von zwei 32-Bit Zahlen kann ein 64-Bit Produkt ergeben.

- Neue Register: **Hi** und **Lo**

- **Hi** beinhaltet die 32 höchstwertigsten Bits des 64-Bit Produkts.

- **Lo** beinhaltet die 32 niederwertigsten Bits.

- MIPS-Befehle:

```
mult $s3,$s4                # Hi#Lo = $s3·$s4
```

- *move from lo*: `mflo $s1` # \$s1 = Lo

- *move from hi*: `mfhi $s2` # \$s2 = Hi

- Pseudo-Instruktion: `mul $s1,$s3,$s4` # \$s1 = (\$s3·\$s4) & 0xFFFFFFFF

- entspricht

```
mult $s3,$s4
mflo $s1
```


Division in MIPS



- Division benutzt auch die Register **Hi** und **Lo**
- nach Division: **Quotient** befindet sich in **Lo**, **Rest** in **Hi**.
- MIPS-Befehl:

`div $s2,$s3` # $Lo = \$s2 / \$s3$, $Hi = \$s2 \% \$s3$

- Pseudo-Instruktionen

<code>div \$s1,\$s2,\$s3</code>	entspricht	<code>div \$s2,\$s3</code> <code>mflo \$s1</code>
<code>rem \$s1,\$s2,\$s3</code>	entspricht	<code>div \$s2,\$s3</code> <code>mfhi \$s1</code>



1. Einführung: MIPS Befehlssatz
2. Assemblersprache
 - Arithmetische Befehle
 - Lade- und Speicherbefehle
 - Logische Operationen
 - Kontrollbefehle
 - Multiplikation und Division
 - Zeichen und Zeichenfolgen
3. Funktionen und Prozeduren
 - Funktionsaufrufe und Stack
 - Registerkonventionen
 - Rekursion
4. Ein größeres Beispiel: Bubble Sort
5. Systemfunktionen
6. Überlauf-Behandlung
7. Floating Point Befehle
8. Maschinensprache
 - Darstellung von Befehlen im Rechner
 - Von-Neumann-Konzept
 - Adressierungsarten
9. Zusammenfassung



- *ASCII (American Standard Code for Information Interchange)* ist ein Standard für Zeichendarstellung.
 - 8 Bit pro Byte, nur 7 Bits werden gebraucht

ASCII	Zeichen	ASCII	Zeichen	ASCII	Zeichen
32	Leerz.	64	@	96	'
33	!	65	A	97	a
34	"	66	B	98	b

- $\text{ASCII-Wert}(a) - \text{ASCII-Wert}(A) = \dots = \text{ASCII-Wert}(z) - \text{ASCII-Wert}(Z) = 32$
- Java verwendet (auch) Unicode
 - 16-Bit
 - z. Z. > 107.000 Zeichen

Befehle zum Transfer von Bytes



- *load byte* lädt ein Byte aus Hauptspeicher in Register
- *store byte* nimmt die rechtsbündigen 8 Bits eines Registers und schreibt sie in den Hauptspeicher

```
lb $t0, 0($gp)    # $t0 =8 Mem[$gp]
```

```
sb $t0, 0($gp)    # Mem[$gp] =8 $t0
```

- In C wird jede Zeichenfolge mit Byte 0 abgeschlossen

„Abba“ =

65	98	98	97	0
----	----	----	----	---

- Wenn man ein Element eines Byte-Arrays laden will, muss man den Index *i* **nicht** mit 4 multiplizieren

Kopieren einer Zeichenfolge



■ C: `int i = 0;`
`while ((d[i] = s[i]) != '\0') i++;`

\$a0

\$a1

`char d[], char s[]`

**Übersetzen Sie in
MIPS-Assembler!**

Kopieren einer Zeichenfolge



■ C: `int i = 0;`
`while ((d[i] = s[i]) != '\0') i++;`

\$a0

\$a1

char d[], char s[]

**Übersetzen Sie in
MIPS-Assembler!**

■ MIPS:

while:

```
add    $t0,$zero,$zero
add    $t1,$a1,$t0
lb     $t1,0($t1)
add    $t2,$a0,$t0
sb     $t1,0($t2)
beq    $t1,$zero,endwhile
addi   $t0,$t0,1
j      while
```

endwhile:

...

```
# i=0
# $t1 = &s[i]
# $t1 = s[i]
# $t2 = &d[i]
# d[i] = s[i]
# if(s[i]==0) goto endwhile
# i++
# goto while
```

Vorzeichenbehaftete und vorzeichenlose Zahlen



- Einige Sprachen (z.B. C) können mit vorzeichenbehafteten (*signed*) und vorzeichenlosen (*unsigned*) Zahlen arbeiten.

C Datentyp	MIPS Datentyp	MIPS Ladebefehl
<code>int</code>	32-Bit Wort	<code>lw</code>
<code>unsigned int</code>	32-Bit Wort	<code>lw</code>
<code>short</code>	16-Bit Halbwort	<code>lh</code>
<code>unsigned short</code>	16-Bit Halbwort (unsigned)	<code>lhu</code>
<code>char</code>	Byte	<code>lb</code>
<code>unsigned char</code>	Byte (unsigned)	<code>lbu</code>

- Javas primitive Datentypen (`byte`, `short`, `int`, `long`) sind signed.
- `lh` und `lb` machen für das zu ladende Byte/Halbword eine **Vorzeichenerweiterung** (`lhu` und `lbu` **nicht**).

Recap: MIPS-Befehlssatz bisher



Kategorie	Befehl	Beispiel	Bedeutung	Anmerkungen
Arithme- tisch	add	<code>add \$s1,\$s2,\$s3</code>	$\$s1 = \$s2 + \$s3$	3 Registeroperanden
	subtract	<code>sub \$s1,\$s2,\$s3</code>	$\$s1 = \$s2 - \$s3$	3 Registeroperanden
	add immediate	<code>addi \$s1,\$s2,100</code>	$\$s1 = \$s2 + 100$	Konstante addieren
	mult	<code>mult \$s3,\$s4</code>	$Hi\#Lo = \$s3 \cdot \$s4$	Multiplikation
	div	<code>div \$s2,\$s3</code>	$Lo = \$s2 / \$s3,$ $Hi = \$s2 \% \$s3$	Division mit Rest
Daten - transfer	load word	<code>lw \$s1,100(\$s2)</code>	$\$s1 = Mem[\$s2 + 100]$	Wort von Hauptspeicher in Register
	store word	<code>sw \$s1,100(\$s2)</code>	$Mem[\$s2 + 100] = \$s1$	Wort von Register in Hauptspeicher
	load byte	<code>lb \$s1,100(\$s2)</code>	$\$s1 = Mem[\$s2 + 100]$	Byte vom Hauptspeicher in Register
	store byte	<code>sb \$s1,100(\$s2)</code>	$Mem[\$s2 + 100] = \$s1$	Byte von Register in Hauptspeicher
	load upper immediate	<code>lui \$s0,100</code>	$\$s1 = 100 \times 2^{16}$	Konstante in obere 16 Bit

blau: Kategorie nicht unbedingt passend

Recap: MIPS-Befehlssatz bisher



Kategorie	Befehl	Beispiel	Bedeutung	Anmerkungen
Logisch	and	<code>and \$s1,\$s2,\$s3</code>	$\$s1 = \$s2 \ \& \ \$s3$	Bitweise UND
	or	<code>or \$s1,\$s2,\$s3</code>	$\$s1 = \$s2 \ \ \$s3$	Bitweise ODER
	nor	<code>nor \$s1,\$s2,\$s3</code>	$\$s1 = \sim (\$s2 \ \ \$s3)$	Bitweise NOR
	and imm.	<code>andi \$s1,\$s2,7</code>	$\$s1 = \$s2 \ \& \ 7$	Bitweise UND mit Konst.
	or imm.	<code>ori \$s1,\$s2,7</code>	$\$s1 = \$s2 \ \ 7$	Bitweise ODER mit Konst.
	shift left logical	<code>sll \$s1,\$s2,10</code>	$\$s1 = \$s2 \ \ll \ 10$	Linksschieben
	shift right logical	<code>srl \$s1,\$s2,10</code>	$\$s1 = \$s2 \ \gg \ 10$	Rechtschieben
Verzweigung	branch on equal	<code>beq \$s1,\$s2,label</code>	<code>if (\$s1==\$s2) goto label</code>	Bedingte Verzweigung
	branch on not equal	<code>bne \$s1,\$s2,label</code>	<code>if (\$s1!=\$s2) goto label</code>	Bedingte Verzweigung
	set on less than	<code>slt \$s0,\$s1,\$s2</code>	$\$s0 = (\$s1 < \$s2)$	Vergleich, kleiner als
	set less than imm.	<code>slti \$s0,\$s1,10</code>	$\$s0 = (\$s1 < 10)$	Kleiner als Konstante
Sprung	jump	<code>j label</code>	<code>goto label</code>	Unbedingter Sprung

blau: Kategorie nicht unbedingt passend

MIPS Reference Sheet



MIPS Reference Data

①



CORE INSTRUCTION SET

NAME, MNEMONIC	FOR-MAT	OPERATION (in Verilog)	OPCODE / FUNCT (Hex)
Add	add	R R[rd] = R[rs] + R[rt]	(1) 0 / 20 _{hex}
Add Immediate	addi	I R[rt] = R[rs] + SignExtImm	(1,2) 8 _{hex}
Add Imm. Unsigned	addiu	I R[rt] = R[rs] + SignExtImm	(2) 9 _{hex}
Add Unsigned	addu	R R[rd] = R[rs] + R[rt]	0 / 21 _{hex}
And	and	R R[rd] = R[rs] & R[rt]	0 / 24 _{hex}
And Immediate	andi	I R[rt] = R[rs] & ZeroExtImm	(3) c _{hex}
Branch On Equal	beq	I if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4) 4 _{hex}
Branch On Not Equal	bne	I if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4) 5 _{hex}
Jump	j	J PC=JumpAddr	(5) 2 _{hex}
Jump And Link	jal	J R[31]=PC+8; PC=JumpAddr	(5) 3 _{hex}
Jump Register	jrr	R PC=R[rs]	0 / 08 _{hex}
Load Byte Unsigned	lbu	I R[rt]={24'b0,M[R[rs] +SignExtImm](7:0)}	(2) 24 _{hex}
Load Halfword Unsigned	lhu	I R[rt]={16'b0,M[R[rs] +SignExtImm](15:0)}	(2) 25 _{hex}
Load Linked	ll	I R[rt] = M[R[rs]+SignExtImm]	(2,7) 30 _{hex}
Load Upper Imm.	lui	I R[rt] = {imm, 16'b0}	f _{hex}
Load Word	lw	I R[rt] = M[R[rs]+SignExtImm]	(2) 23 _{hex}
Nor	nor	R R[rd] = ~(R[rs] R[rt])	0 / 27 _{hex}
Or	or	R R[rd] = R[rs] R[rt]	0 / 25 _{hex}
Or Immediate	ori	I R[rt] = R[rs] ZeroExtImm	(3) d _{hex}
Set Less Than	slt	R R[rd] = (R[rs] < R[rt]) ? 1 : 0	0 / 2a _{hex}

ARITHMETIC CORE INSTRUCTION SET

②

NAME, MNEMONIC	FOR-MAT	OPERATION	OPCODE / FMT / FT / FUNCT (Hex)
Branch On FP True	bc1t	FI if(FPcond)PC=PC+4+BranchAddr	(4) 11/8/1/--
Branch On FP False	bc1f	FI if(!FPcond)PC=PC+4+BranchAddr	(4) 11/8/0/--
Divide	div	R Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]	0/--/1a
Divide Unsigned	divu	R Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]	(6) 0/--/1b
FP Add Single	add.s	FR F[fd] = F[fs] + F[ft]	11/10/--/0
FP Add Double	add.d	FR {F[fd],F[fd+1]} = {F[fs],F[fs+1]} + {F[ft],F[ft+1]}	11/11/--/0
FP Compare Single	c.x.s*	FR FPcond = (F[fs] op F[ft]) ? 1 : 0	11/10/--/y
FP Compare Double	c.x.d*	FR FPcond = ({F[fs],F[fs+1]} op {F[ft],F[ft+1]}) ? 1 : 0	11/11/--/y
* (x is eq, lt, or le) (op is ==, <, or <=) (y is 32, 3c, or 3e)			
FP Divide Single	div.s	FR F[fd] = F[fs] / F[ft]	11/10/--/3
FP Divide Double	div.d	FR {F[fd],F[fd+1]} = {F[fs],F[fs+1]} / {F[ft],F[ft+1]}	11/11/--/3
FP Multiply Single	mul.s	FR F[fd] = F[fs] * F[ft]	11/10/--/2
FP Multiply Double	mul.d	FR {F[fd],F[fd+1]} = {F[fs],F[fs+1]} * {F[ft],F[ft+1]}	11/11/--/2
FP Subtract Single	sub.s	FR F[fd]=F[fs] - F[ft]	11/10/--/1
FP Subtract Double	sub.d	FR {F[fd],F[fd+1]} = {F[fs],F[fs+1]} - {F[ft],F[ft+1]}	11/11/--/1
Load FP Single	lwc1	I F[rt]=M[R[rs]+SignExtImm]	(2) 31/--/--
Load FP Double	ldc1	I F[rt]=M[R[rs]+SignExtImm]; F[rt+1]=M[R[rs]+SignExtImm+4]	(2) 35/--/--
Move From Hi	mfhi	R R[rd] = Hi	0 /--/--/10
Move From Lo	mflo	R R[rd] = Lo	0 /--/--/12
Move From Control	mfc0	R R[rd] = CR[rs]	10 /0/--/0
Multiply	mult	R {Hi,Lo} = R[rs] * R[rt]	0/--/--/18
Multiply Unsigned	multu	R {Hi,Lo} = R[rs] * R[rt]	(6) 0/--/--/19
Shift Right Arith.	sra	R R[rd] = R[rt] >>> shamt	0/--/--/3
Store FP Single	swc1	I M[R[rs]+SignExtImm] = F[rt]	(2) 39/--/--
Store FP Double	sdc1	I M[R[rs]+SignExtImm] = F[rt]; M[R[rs]+SignExtImm+4] = F[rt+1]	(2) 3d/--/--



1. Einführung: MIPS Befehlssatz
2. Assemblersprache
 - Arithmetische Befehle
 - Lade- und Speicherbefehle
 - Logische Operationen
 - Kontrollbefehle
 - Multiplikation und Division
 - Zeichen und Zeichenfolgen
3. Funktionen und Prozeduren
 - Funktionsaufrufe und Stack
 - Registerkonventionen
 - Rekursion
4. Ein größeres Beispiel: Bubble Sort
5. Systemfunktionen
6. Überlauf-Behandlung
7. Floating Point Befehle
8. Maschinensprache
 - Darstellung von Befehlen im Rechner
 - Von-Neumann-Konzept
 - Adressierungsarten
9. Zusammenfassung



- Äußerst wichtig in höheren Programmiersprachen
 - Strukturierung des Programms
 - Abstraktion!
- In diesem Abschnitt lernen wir:
 - „Blatt“-Funktionen zu übersetzen.
 - Nicht-Blatt-Funktionen zu übersetzen.
 - MIPS Registerkonventionen zu beachten.
 - Rekursive Funktionen zu übersetzen.

Funktionsaufruf



- Diskutieren Sie den Ablauf der Funktionsaufrufe in diesem Beispiel!

```
main()  
{  
    fun1(a, b);  
  
    c = fun2(a+b);  
  
    fun1(c, d);  
}
```

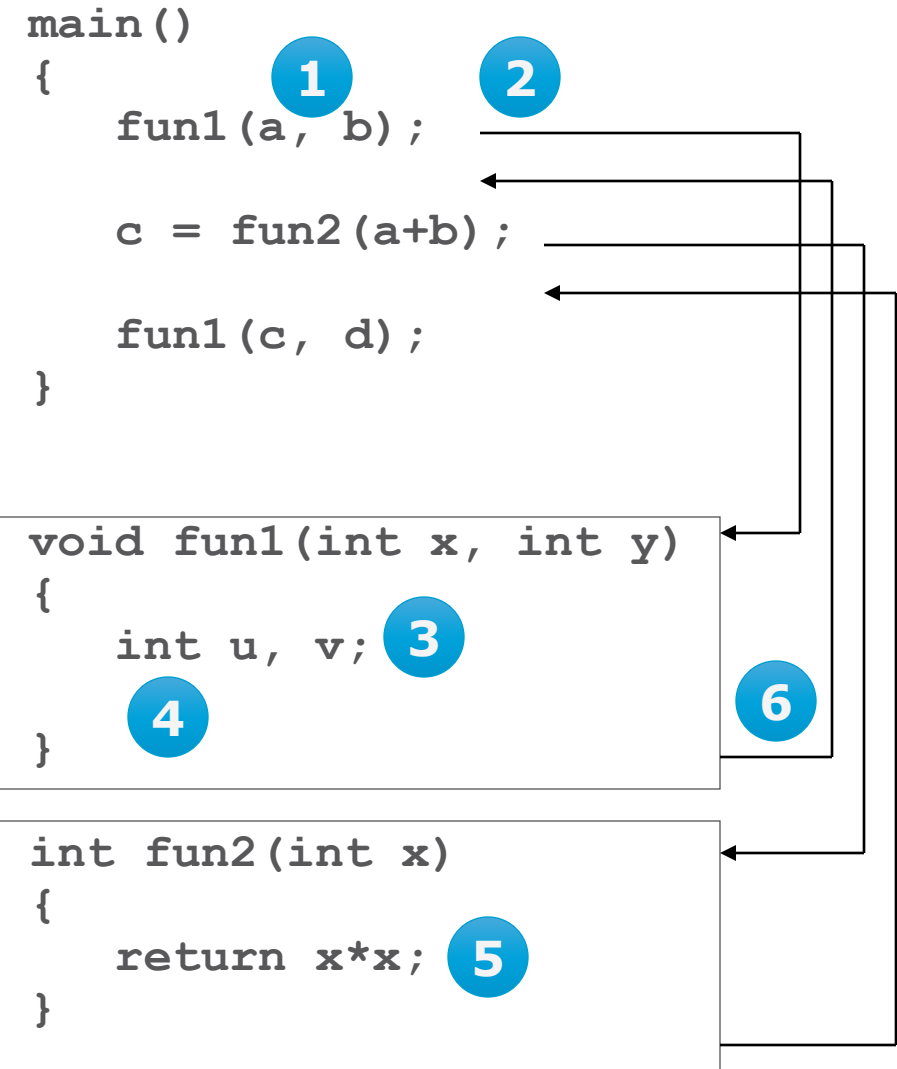
```
void fun1(int x, int y)  
{  
    int u, v;  
  
}
```

```
int fun2(int x)  
{  
    return x*x;  
}
```

Funktionsaufruf



- 6 Schritte beim Ausführen einer Funktion:
 1. **Parameter** werden an einer Stelle abgelegt, auf die die aufgerufene Funktion zugreifen kann.
 2. **Kontrollfluss** wird an die Funktion **übergeben**.
 3. Die für die Funktion benötigten **Speicherressourcen** werden **bereitgestellt**.
 4. **Prozedur** wird **ausgeführt**.
 5. **Ergebnis** wird an einer Stelle abgelegt, auf die die aufrufende Funktion zugreifen kann.
 6. **Rücksprung** an die Stelle, an der die Funktion aufgerufen wurde
- **Caller**: aufrufende Funktion
- **Callee**: aufgerufene Funktion



Funktionsaufrufe in MIPS



- Registerkonventionen für Funktionsaufrufe:
 - `$a0–$a3`: 4 Argumentregister
 - `$v0–$v1`: 2 Register für Rückgabewerte
 - `$ra`: Rücksprungadresse (*return address*)
- Befehl, um zu einer Funktion zu springen: *Jump-and-Link* (**jal**)
`jal FunAddress # speichere Rücksprungadresse in $ra`
`# und springe zu Label FunAddress`
- Befehlszeiger (*Program Counter*, PC) enthält Adresse des aktuellen Befehls.
Wie gelangen wir zur Rücksprungadresse?
 $\Rightarrow \$ra = PC + 4$
- Befehl zur Rückkehr in aufrufende Funktion: *Jump-Register* (**jr**)
`jr $ra # PC = $ra`

Beispiel Parameterübergabe und Funktionsaufruf



Hochschule für
Wirtschaft und Recht Berlin
Berlin School of Economics and Law

Pseudo-C:

```
main()  
{  
    fun1(a, b);  
  
    c = fun2(a+b);  
  
    fun1(c, d);  
}  
  
void fun1(int x, int y)  
{  
    int u, v;  
  
}  
  
int fun2(int x)  
{  
    return x*x;  
}
```

Pseudo-MIPS:


```
main:  
    move $a0,a  
    move $a1,b  
    jal  fun1  
  
    add  $a0,a,b  
    jal  fun2  
  
    move $a0,$v0  
    move $a1,d  
    jal  fun1  
  
    jr   $ra  
  
fun1:  
    # x in $a0, y in $a1  
    jr   $ra  
  
fun2:  
    mul  $v0,$a0,$a0  
    jr   $ra
```


Übersetzung einer Blattfunktion



- Blattfunktion = Funktion, die keine andere Funktion aufruft

■ Beispiel: `void swap(int v[], int k){`
 `int tmp;`
 `tmp = v[k];`
 `v[k] = v[k+1];`
 `v[k+1] = tmp;`
}



**Übersetzen Sie in
MIPS-Assembler!**

Übersetzung einer Blattfunktion



- Blattfunktion = Funktion, die keine andere Funktion aufruft

- Beispiel: `void swap(int v[], int k){`

```
    int tmp;  
    tmp = v[k];  
    v[k] = v[k+1];  
    v[k+1] = tmp;  
}
```

\$a0

\$a1

**Übersetzen Sie in
MIPS-Assembler!**

- MIPS:

```
swap: sll $t0,$a1,2    # $t0 = 4*k  
      add $t0,$a0,$t0  # $t0 = &v[k]  
      lw  $t1,0($t0)   # temp = v[k]  
      lw  $t2,4($t0)   # $t2 = v[k+1]  
      sw  $t2,0($t0)   # v[k] = v[k+1]  
      sw  $t1,4($t0)   # v[k+1] = temp  
      jr  $ra          # Rücksprung (return)
```

- Caller: `jal swap`

Funktionsaufrufe: Probleme



- Probleme, Probleme, Probleme, ...
 - Was, wenn eine Funktion eine andere aufruft? (**\$ra?**)
 - Was, wenn eine Funktion **mehr als 4 Parameter** hat?
 - Was, wenn eine Funktion **mehr als 2 Rückgabewerte** hat?
 - Was, wenn eine Funktion **andere Register** des Aufrufers **überschreibt**?

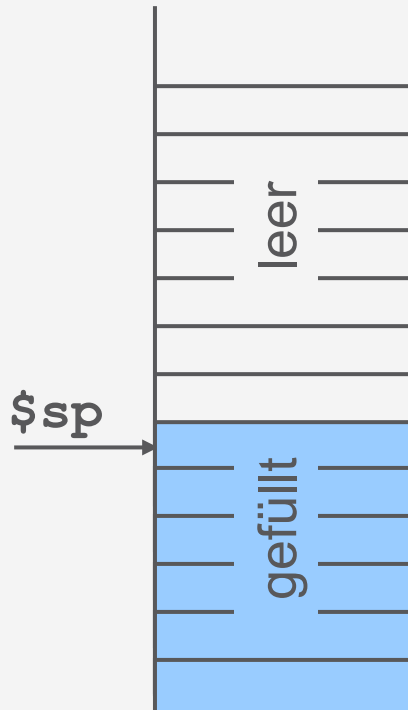
- Probleme, Probleme, Probleme, ...
 - Was, wenn eine Funktion eine andere aufruft? (**\$ra?**)
 - Was, wenn eine Funktion mehr als 4 Parameter hat?
 - Was, wenn ... ?
- Wichtige Datenstruktur:
 - **Stack (=Stapel)**: *last-in-first-out* (LIFO)
- 2 Basisoperationen:
 - **push**: etwas auf dem Stack ablegen
 - **pop**: etwas vom Stack entfernen
- In MIPS:
 - Stack wächst **von höherwertigen zu niederwertigen Adressen**
 - **Stack pointer (\$sp)** zeigt auf das „*oberste*“ Element des Stacks



Werte auf dem Stack ablegen



niedrige Adresse

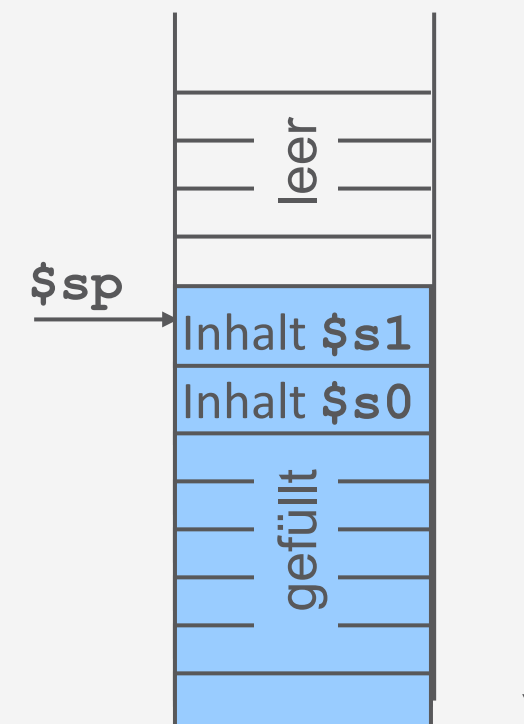


hohe Adresse

Ablegen von `$s0` und `$s1`:

```
addi $sp, $sp, -8  
sw   $s0, 4($sp)  
sw   $s1, 0($sp)
```

niedrige Adresse

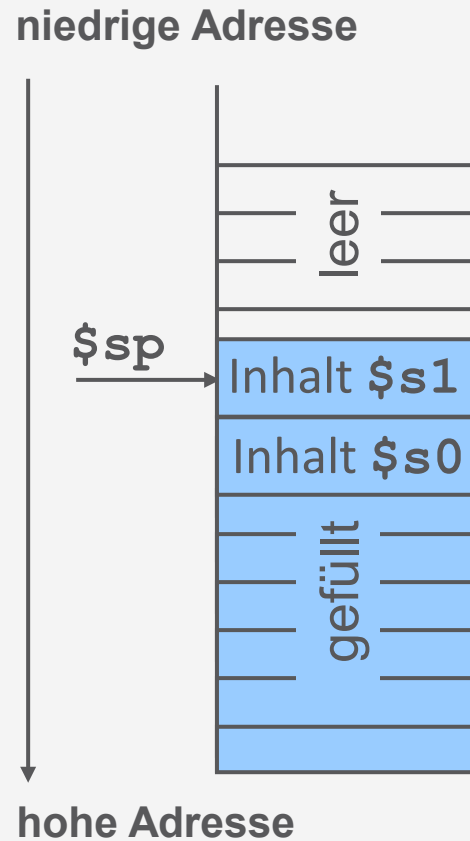


hohe Adresse

Werte vom Stack entfernen



Wie lauten die Befehle um die Werte wieder aus dem Stack zu entfernen und in `$s0` und `$s1` zu speichern?



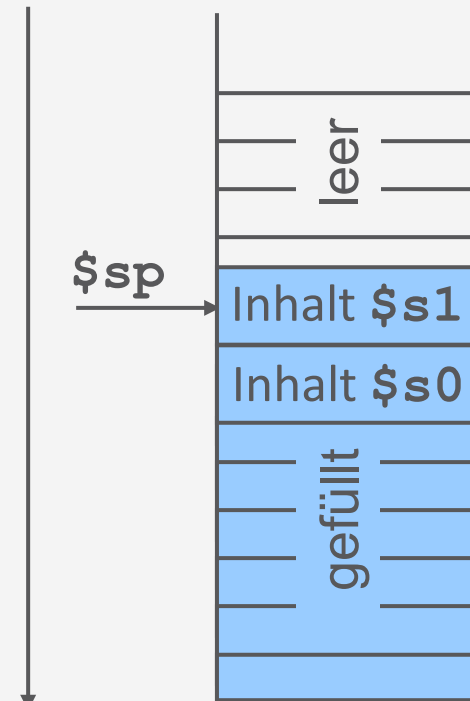
Werte vom Stack entfernen



Wie lauten die Befehle um die Werte wieder aus dem Stack zu entfernen und in `$s0` und

`$s1` zu speichern?

niedrige Adresse

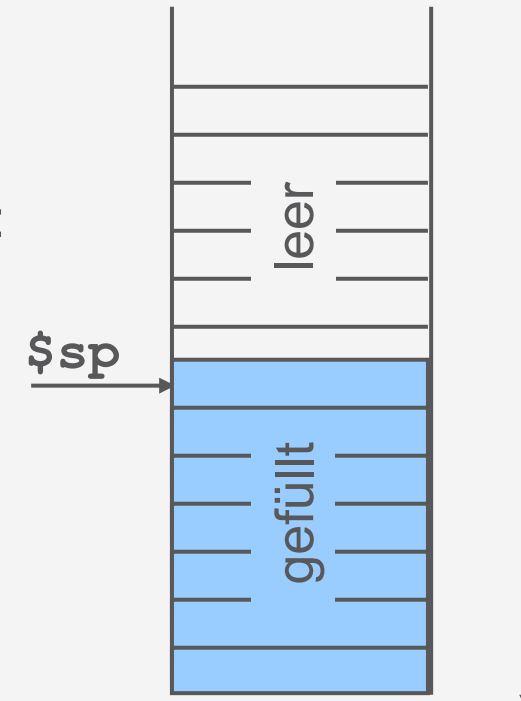


hohe Adresse

Entfernen von `$s0` und `$s1`:

```
lw    $s0, 4($sp)
lw    $s1, 0($sp)
addi  $sp, $sp, 8
```

niedrige Adresse

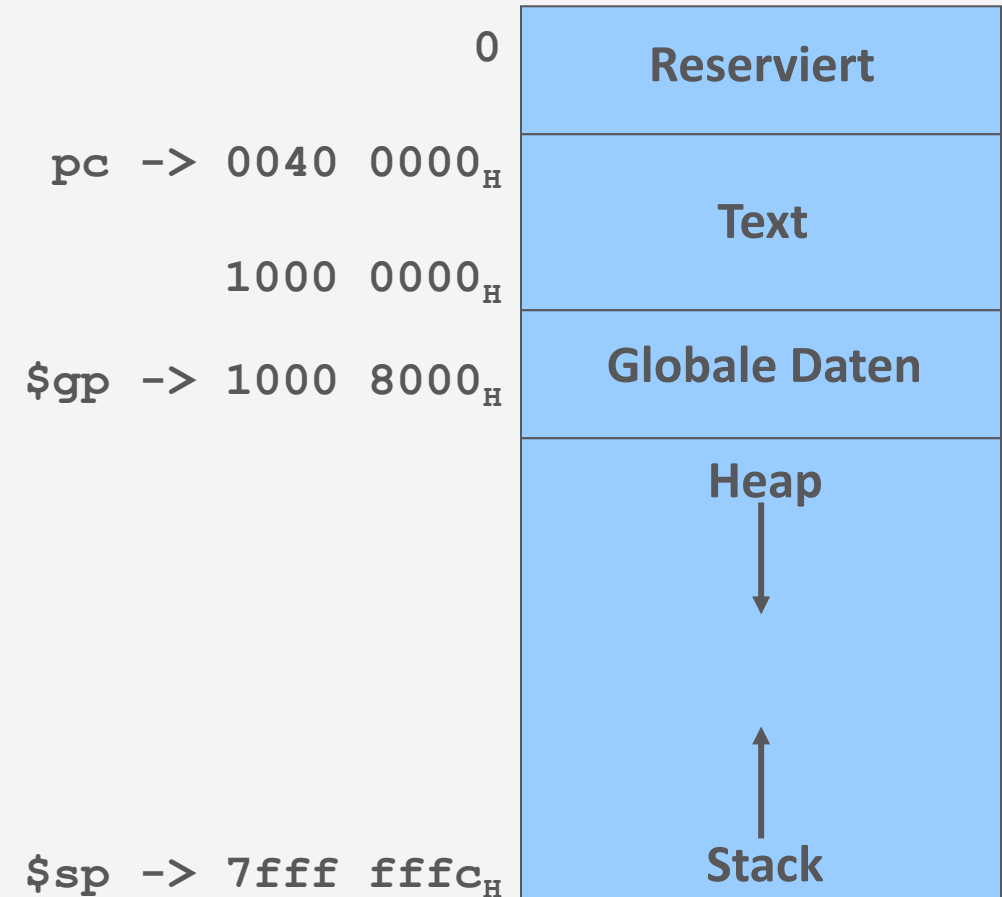


hohe Adresse

MIPS-Speicheraufteilung



- Kleinster Adress-Bereich ist reserviert (für das Betriebssystem)
- **Programcode** (Text) beginnt bei 0040 0000_H
- Statische Daten (z. B. globale Variablen)
 - **\$gp** zeigt etwa in die Mitte
- Bereich für **dynamische Daten** (Halde = *heap*) wächst hin zu größeren Adressen
 - C: **malloc**, Java: **new**
- **Stack** pointer wird mit 7fff ffff_H initialisiert und wächst hin zu kleineren Adressen
- Stack und Heap wachsen in entgegengesetzte Richtungen





1. Einführung: MIPS Befehlssatz
2. Assemblersprache
 - Arithmetische Befehle
 - Lade- und Speicherbefehle
 - Logische Operationen
 - Kontrollbefehle
 - Multiplikation und Division
 - Zeichen und Zeichenfolgen
3. Funktionen und Prozeduren
 - Funktionsaufrufe und Stack
 - Registerkonventionen
 - Rekursion
4. Ein größeres Beispiel: Bubble Sort
5. Systemfunktionen
6. Überlauf-Behandlung
7. Floating Point Befehle
8. Maschinensprache
 - Darstellung von Befehlen im Rechner
 - Von-Neumann-Konzept
 - Adressierungsarten
9. Zusammenfassung

MIPS-Registerkonventionen / 1



Name	Register- nummer	Verwendung	Bei Aufruf beibehalten?
\$zero	0	Konstante 0	-
\$v0-\$v1	2-3	Werte für Ergebnisse und für die Auswertung von Ausdrücken	nein
\$a0-\$a3	4-7	Argumente	nein
\$t0-\$t7	8-15	temporäre Variablen	nein
\$s0-\$s7	16-23	gesicherte Variablen	ja
\$t8-\$t9	24-25	weitere temporäre Variablen	nein
\$gp	28	globaler Zeiger (<i>Global pointer</i>)	ja
\$sp	29	Kellerzeiger (<i>Stack pointer</i>)	ja
\$fp	30	Rahmenzeiger (<i>Frame pointer</i>)	ja
\$ra	31	Rücksprungadresse	ja

MIPS-Registerkonventionen / 2



- $\$t0-\$t9$: 10 **temporäre** Register, die vom *Callee* nicht gerettet werden müssen.
- $\$s0-\$s7$: 8 zu sichernde Register (***saved registers***), die vom *Callee* bei Verwendung gerettet werden müssen.
 - „**Vertrag**“ zwischen *Caller* und *Callee*
- Regeln bei Übersetzung einer nicht-Blatt-Funktion:
 - sichere **$\$ra$** auf dem Stack
 - weise Variablen, die nach einem Aufruf benötigt werden, an ein **$\$si$** Register zu und sichere zuvor **$\$si$** auf dem Stack
 - weise Variablen, die nach einem Aufruf nicht länger benötigt werden, an ein **$\$ti$** Register zu
 - kopiere Argumente (**$\$ai$**), die nach einem Aufruf benötigt werden, in ein **$\$si$** -Register und sichere zuvor **$\$si$** auf dem Stack



Übersetzung einer Nicht-Blattfunktion



■ C:

```
int poly(int x) {  
    return square(x)+x+1;  
}
```

x (\$a0) wird nach dem Aufruf
von `square` wieder benötigt

Implementieren
Sie `poly` in
Assembler!

Übersetzung einer Nicht-Blattfunktion



■ C:

```
int poly(int x){  
    return square(x)+x+1;  
}
```

x (\$a0) wird nach dem Aufruf
von square wieder benötigt

Implementieren
Sie poly in
Assembler!

■ MIPS:

```
poly:  addi  $sp,$sp,-8  # Stack-Reservierung für 2 Variablen  
       sw   $ra,4($sp)  # speichere $ra auf Stack  
       sw   $s0,0($sp)  # speichere $s0 auf Stack  
       addi $s0,$a0,0   # $s0 = $a0 (=x)  
       jal  square     # $v0 = square(x)  
       add  $v0,$v0,$s0 # $v0 = $v0+x  
       addi $v0,$v0,1   # $v0 = $v0+1  
       lw   $ra,4($sp)  # wiederherstellen der Rücksprungsadresse  
       lw   $s0,0($sp)  # wiederherstellen von $s0  
       addi $sp,$sp,8   # wiederherstellen des $sp  
       jr   $ra         # Rücksprung
```



1. Einführung: MIPS Befehlssatz
2. Assemblersprache
 - Arithmetische Befehle
 - Lade- und Speicherbefehle
 - Logische Operationen
 - Kontrollbefehle
 - Multiplikation und Division
 - Zeichen und Zeichenfolgen
3. Funktionen und Prozeduren
 - Funktionsaufrufe und Stack
 - Registerkonventionen
 - Rekursion
4. Ein größeres Beispiel: Bubble Sort
5. Systemfunktionen
6. Überlauf-Behandlung
7. Floating Point Befehle
8. Maschinensprache
 - Darstellung von Befehlen im Rechner
 - Von-Neumann-Konzept
 - Adressierungsarten
9. Zusammenfassung



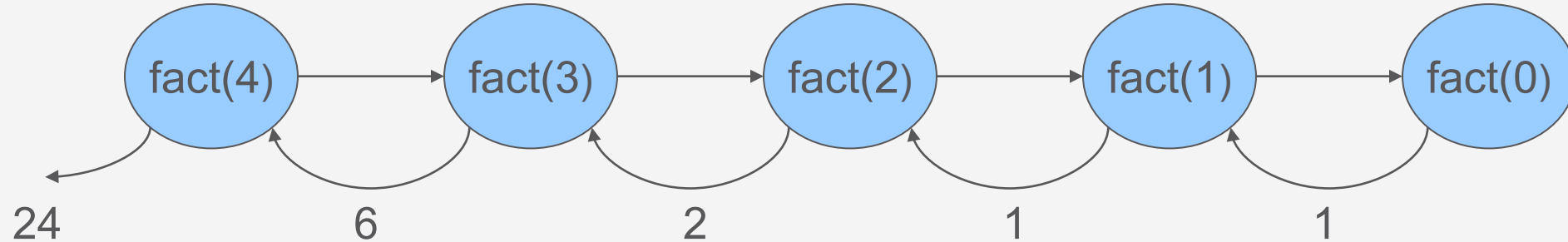
- Rekursive Funktion zur Berechnung der Fakultät:

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

- Mathematisch:

- $n! = n \cdot (n-1)!$ für $n \geq 1$
- $0! = 1$

Aufrufkette: Fakultät



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




- Vor dem rekursiven Aufruf sichert man die Rücksprungadresse (**\$ra**) und das Register **\$s0** (wird für das Argument **n** bzw. **\$a0** verwendet) auf dem Stack:

```
addi $sp,$sp,-8 # Stack-Reservierung für 2 Variablen
sw    $ra,4($sp) # speichere $ra auf Stack
sw    $s0,0($sp) # speichere $s0 auf Stack
```

- Vor dem Verlassen der Funktion stellt man **\$s0** und **\$ra** wieder her:

```
lw    $ra,4($sp) # stelle $ra wieder her
lw    $s0,0($sp) # stelle $s0 wieder her
addi $sp,$sp,8  # wiederherstellen des Stackpointers
```

Assemblercode für fact

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

Befehlsadresse (Instruction address)

1000	fact: slti \$t0,\$a0,1	# \$t0 = n<1
1004	beq \$t0,\$zero,else	# if (n>=1) goto else
1008	addi \$v0,\$zero,1	# \$v0 = 1
1012	jr \$ra	# Rücksprung (return)
1016	else: addi \$sp,\$sp,-8	# reserviere 2 Plätze auf Stack
1020	sw \$ra,4(\$sp)	# speichere \$ra auf Stack
1024	sw \$s0,0(\$sp)	# speichere \$s0 auf Stack
1028	addi \$s0,\$a0,0	# \$s0 = n
1032	addi \$a0,\$a0,-1	# \$a0 = n-1
1036	jal fact	# \$v0 = fact(n-1)
1040	mult \$s0,\$v0	# Hi#Lo = n*fact(n-1)
1044	mflo \$v0	# \$v0 = Lo
1048	lw \$ra,4(\$sp)	# wiederherstellen (\$ra)
1052	lw \$s0,0(\$sp)	# wiederherstellen (\$s0)
1056	addi \$sp,\$sp,8	# Stack freigeben
1060	jr \$ra	# Rücksprung

Fakultätsfunktion: Ablauf



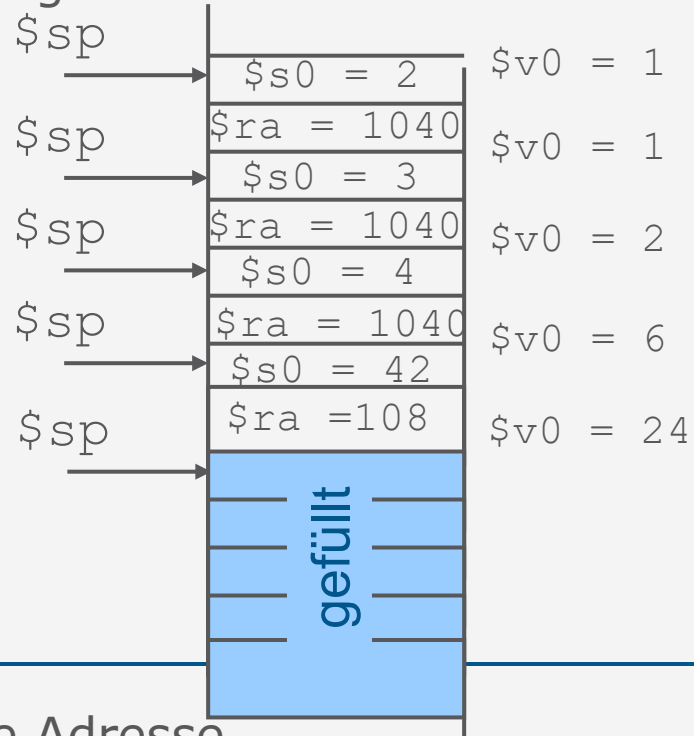
Caller

```
100 addi $a0,$zero,4
```

```
104 jal fact
```

```
108 ...
```

niedrige Adresse



hohe Adresse

```
1000 fact: slti $t0,$a0,1
1004        beq  $t0,$zero,else
1008        addi $v0,$zero,1
1012        jr   $ra
1016 else:  addi $sp,$sp,-8
1020        sw   $ra,4($sp)
1024        sw   $s0,0($sp)
1028        addi $s0,$a0,0
1032        addi $a0,$a0,-1
1036        jal  fact
1040        mult $s0,$v0
1044        mflo $v0
1048        lw   $ra,4($sp)
1052        lw   $s0,0($sp)
1056        addi $sp,$sp,8
1060        jr   $ra
```



1. Einführung: MIPS Befehlssatz
2. Assemblersprache
 - Arithmetische Befehle
 - Lade- und Speicherbefehle
 - Logische Operationen
 - Kontrollbefehle
 - Multiplikation und Division
 - Zeichen und Zeichenfolgen
3. Funktionen und Prozeduren
 - Funktionsaufrufe und Stack
 - Registerkonventionen
 - Rekursion
4. Ein größeres Beispiel: Bubble Sort
5. Systemfunktionen
6. Überlauf-Behandlung
7. Floating Point Befehle
8. Maschinensprache
 - Darstellung von Befehlen im Rechner
 - Von-Neumann-Konzept
 - Adressierungsarten
9. Zusammenfassung

Beispiel: Bubble Sort



- Bubble Sort: sortieren durch Aufsteigen
- „Benötigt“ Prozedur **swap**
- Bei manueller Übersetzung von C in Assemblerspache wie folgt vorgehen:
 - Register an Programmvariablen zuteilen
 - Code für den Rumpf der Prozedur generieren
 - Register über Prozeduraufruf hinweg erhalten

Bubble Sort – C Code



```
void sort(int v[], int n){
    int i, j;
    for (i=0; i < n-1; i++)
        for (j=0; j < n-i-1; j++)
            if (v[j] > v[j+1])
                swap(v, j);
}

void swap(int v[], int k){
    int temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

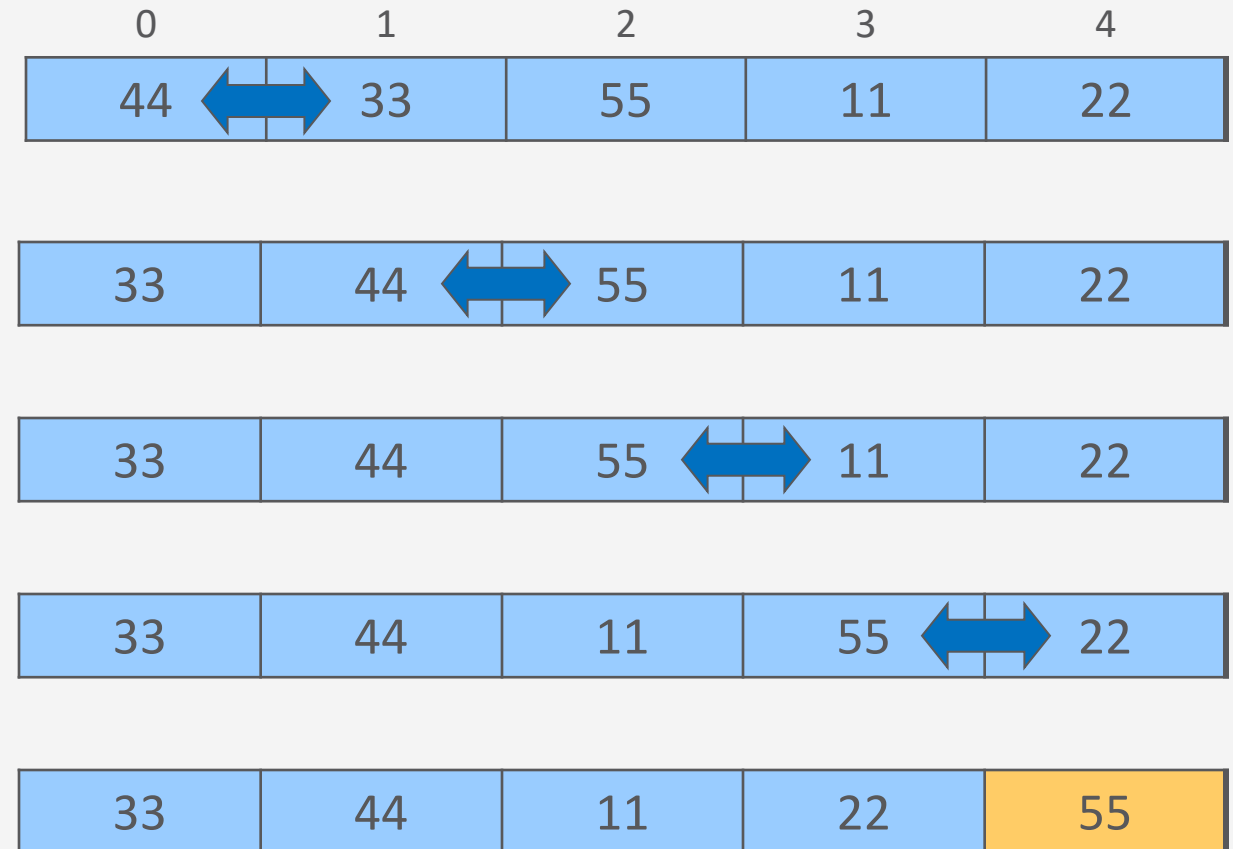
Bubble Sort / 1



Bubble Sort, 1. Durchlauf der äußeren Schleife ($i = 0, j = 0, \dots, 3$)

```
void sort(int v[], int n){
    int i, j;
    for (i=0; i < n-1; i++)
        for (j=0; j < n-i-1; j++)
            if (v[j] > v[j+1])
                swap(v, j);
}

void swap(int v[], int k){
    int temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```



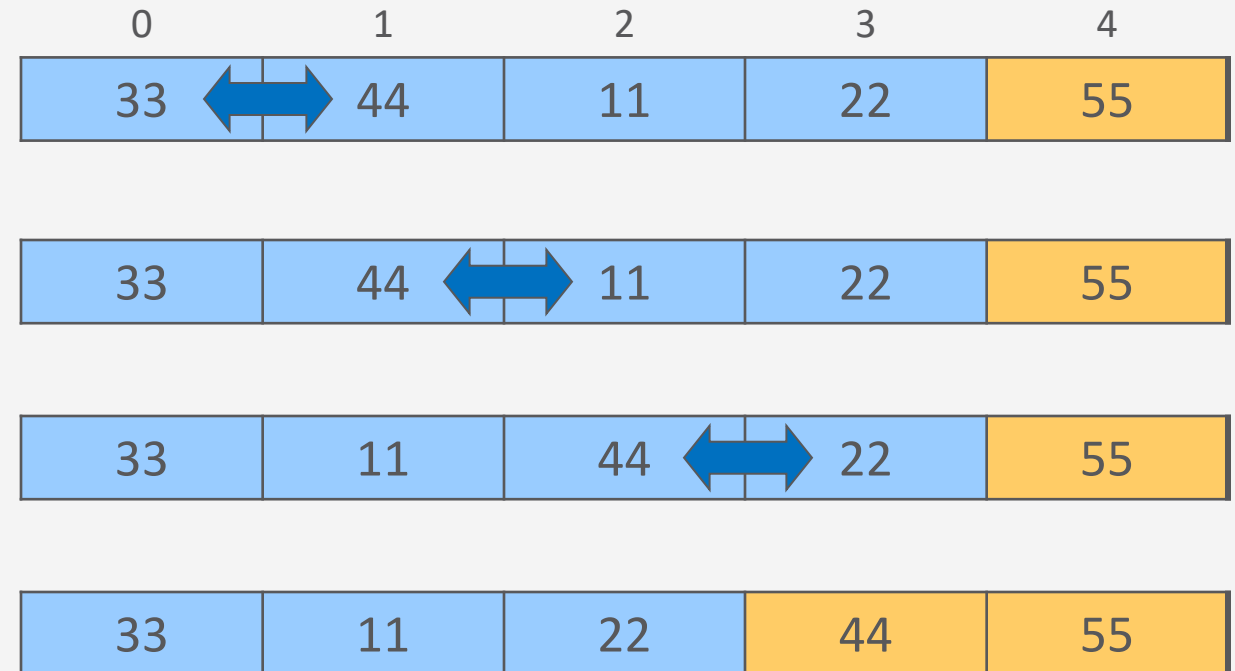
Bubble Sort / 2



Bubble Sort, 2. Durchlauf der äußeren Schleife ($i = 1, j = 0, 1, 2$)

```
void sort(int v[], int n){
    int i, j;
    for (i=0; i < n-1; i++)
        for (j=0; j < n-i-1; j++)
            if (v[j] > v[j+1])
                swap(v, j);
}

void swap(int v[], int k){
    int temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```



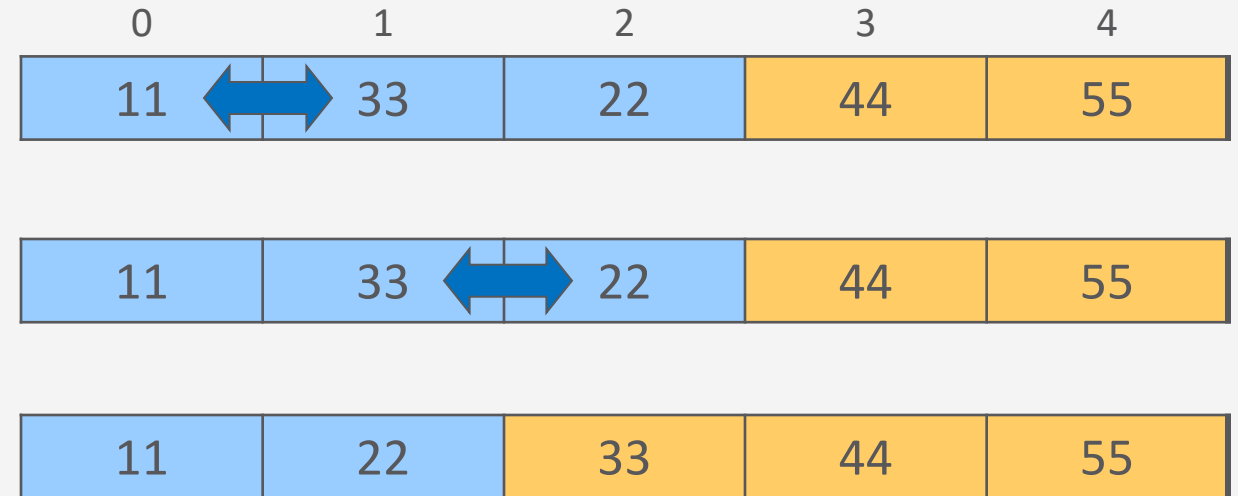
Bubble Sort / 3



Bubble Sort, 3. Durchlauf der äußeren Schleife ($i = 2, j = 0, 1$)

```
void sort(int v[], int n){
    int i, j;
    for (i=0; i < n-1; i++)
        for (j=0; j < n-i-1; j++)
            if (v[j] > v[j+1])
                swap(v, j);
}

void swap(int v[], int k){
    int temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```



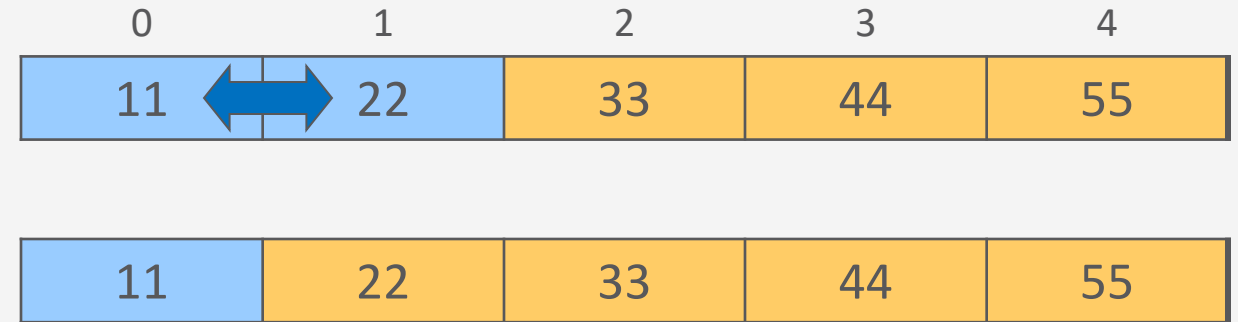
Bubble Sort / 4



Bubble Sort, 4. Durchlauf der äußeren Schleife ($i = 3, j = 0$)

```
void sort(int v[], int n){
    int i, j;
    for (i=0; i < n-1; i++)
        for (j=0; j < n-i-1; j++)
            if (v[j] > v[j+1])
                swap(v, j);
}

void swap(int v[], int k){
    int temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```



Assemblercode für swap



- Registerkonvention:
 - `v[]` in `$a0`, `k` in `$a1`
 - brauchen nicht gesichert zu werden
- Blattfunktion
 - verwende (wenn möglich) sonst nur temporäre (`$ti`) Register

```
void swap(int v[], int k){  
    int temp = v[k];  
    v[k] = v[k+1];  
    v[k+1] = temp;  
}
```

swap:

```
    sll    $t0,$a1,2      # $t0 = 4*k  
    add    $t0,$a0,$t0    # $t0 = &v[k]  
    lw     $t1,0($t0)     # $t1 = v[k]  
    lw     $t2,4($t0)     # $t2 = v[k+1]  
    sw     $t2,0($t0)     # v[k] = v[k+1]  
    sw     $t1,4($t0)     # v[k+1] = $t1  
    jr     $ra            # Rücksprung(return)
```

Assemblercode für sort

- Registerkonvention
 - `v[]` in `$a0`, `n` in `$a1`
- Nicht-Blattfunktion
 - sichere `$ra` auf dem Stack
 - verwende für Variablen, die nach Aufruf benötigt werden, *saved* (`$si`)-Register
 - `i`, `j`, `v[]` (`$a0`), `n-1`, `n-i-1`
 - sichere saved-Register vor der Nutzung auf dem Stack und stelle sie nach der Nutzung wieder her

```
void sort(int v[], int n){
    int i, j;
    for (i=0; i < n-1; i++)
        for (j=0; j < n-i-1; j++)
            if (v[j] > v[j+1])
                swap(v, j);
}
```

<code>\$s0</code>	<code>i</code>
<code>\$s1</code>	<code>n-1</code>
<code>\$s2</code>	<code>j</code>
<code>\$s3</code>	<code>n-i-1</code>
<code>\$s4</code>	<code>v[]</code>

Assemblercode für sort

- 1. Register retten

Hochschule für

```
void sort(int v[], int n){
    int i, j;
    for (i=0; i < n-1; i++)
        for (j=0; j < n-i-1; j++)
            if (v[j] > v[j+1])
                swap(v, j);
}
```

```
sort:
    addi    $sp,$sp,-24      # Stack-Reservierung
                                # für 6 Register
    sw      $ra,20($sp)      # speichere $ra
    sw      $s4,16($sp)      # speichere $s4
    sw      $s3,12($sp)      # speichere $s3
    sw      $s2,8($sp)       # speichere $s2
    sw      $s1,4($sp)       # speichere $s1
    sw      $s0,0($sp)       # speichere $s0
    # weiter auf nächste Folie
```

\$s0	i
\$s1	n-1
\$s2	j
\$s3	n-i-1
\$s4	v[]

Assemblercode für sort

■ 2. Prozedurrumpf

```
void sort(int v[], int n){
    int i, j;
    for (i=0; i < n-1; i++)
        for (j=0; j < n-i-1; j++)
            if (v[j] > v[j+1])
                swap(v, j);
}
```

```
        move    $s0,$zero        # i = 0
        move    $s4,$a0          # $s4 = v[] (rette $a0)
        addi    $s1,$a1,-1       # $s1 = n-1
for1:    bge     $s0,$s1,endifor1  # if (i>=n-1) goto endifor1
        move    $s2,$zero        # j = 0
        sub     $s3,$s1,$s0      # $s3 = n-1-i
for2:    bge     $s2,$s3,endifor2  # if (j>=n-i-1) goto endifor2
        sll     $t0,$s2,2        # $t0 = 4*j
        add     $t0,$s4,$t0      # $t0 = v+4*j = &v[j]
        lw      $t1,0($t0)       # $t1 = v[j]
        lw      $t2,4($t0)       # $t2 = v[j+1]
        ble     $t1,$t2,endif    # if (v[j]<=v[j+1]) goto endif
# weiter auf nächster Folie
```

\$s0	i
\$s1	n-1
\$s2	j
\$s3	n-i-1
\$s4	v[]

Assemblercode für sort

■ 3. Prozeduraufruf

Hochschule für

```
void sort(int v[], int n){
    int i, j;
    for (i=0; i < n-1; i++)
        for (j=0; j < n-i-1; j++)
            if (v[j] > v[j+1])
                swap(v, j);
}
```

```
        ble     $t1,$t2,endif      # if (v[j]<=v[j+1]) goto endif
        move    $a0,$s4            # $a0 = v[]
        move    $a1,$s2            # $a1 = j
        jal     swap               # swap(v, j)
endif:
        addi    $s2,$s2,1          # j++
        j       for2              # goto for2
endfor2:
        addi    $s0,$s0,1          # i++
        j       for1              # goto for1
endfor1:
# weiter auf nächste Folie
```

\$s0	i
\$s1	n-1
\$s2	j
\$s3	n-i-1
\$s4	v[]

Assemblercode für sort

■ 4. Register wieder herstellen

Hochschule für

```
void sort(int v[], int n){
    int i, j;
    for (i=0; i < n-1; i++)
        for (j=0; j < n-i-1; j++)
            if (v[j] > v[j+1])
                swap(v, j);
}
```

```
endfor1:
    lw    $ra, 20($sp)    # wiederherstellen von $ra
    lw    $s4, 16($sp)    # wiederherstellen von $s4
    lw    $s3, 12($sp)    # wiederherstellen von $s3
    lw    $s2, 8($sp)     # wiederherstellen von $s2
    lw    $s1, 4($sp)     # wiederherstellen von $s1
    lw    $s0, 0($sp)     # wiederherstellen von $s0
    addi   $sp, $sp, 24    # wiederherstellen von $sp
    jr     $ra            # Rücksprung (return)
```

\$s0	i
\$s1	n-1
\$s2	j
\$s3	n-i-1
\$s4	v[]



1. Einführung: MIPS Befehlssatz
2. Assemblersprache
 - Arithmetische Befehle
 - Lade- und Speicherbefehle
 - Logische Operationen
 - Kontrollbefehle
 - Multiplikation und Division
 - Zeichen und Zeichenfolgen
3. Funktionen und Prozeduren
 - Funktionsaufrufe und Stack
 - Registerkonventionen
 - Rekursion
4. Ein größeres Beispiel: Bubble Sort
5. Systemfunktionen
6. Überlauf-Behandlung
7. Floating Point Befehle
8. Maschinensprache
 - Darstellung von Befehlen im Rechner
 - Von-Neumann-Konzept
 - Adressierungsarten
9. Zusammenfassung

Systemfunktionen (*system services*)



- Um Systemfunktionen (*system services*) nutzen zu können, steht in MIPS der Spezialbefehl **syscall** zur Verfügung
- = „Software-Interrupt“, der vom Betriebssystem gehandhabt wird
- Vorgehen zum Aufruf von Systemfunktionen:
 1. Service-Nummer in Register **\$v0** laden
 2. Argumente für den Service (falls nötig) in **\$a0** , **\$a1** , **\$a2** , **\$f12** laden
 3. **syscall** aufrufen
 4. Rückgabewerte des Services (falls vorhanden) sichern

Systemfunktionen in MARS (1)



Service	Code in \$v0	Arguments	Result
print integer	1	\$a0 = integer to print	
print float	2	\$f12 = float to print	
print double	3	\$f12 = double to print	
print string	4	\$a0 = address of null-terminated string to print	
read integer	5		\$v0 contains integer read
read float	6		\$f0 contains float read
read double	7		\$f0 contains double read
read string	8	\$a0 = address of input buffer \$a1 = maximum number of characters to read	<i>analog zu UNIX fgets</i>

Systemfunktionen in MARS (2)



Service	Code in \$v0	Arguments	Result
sbrk (allocate heap memory)	9	\$a0 = number of bytes to allocate	\$v0 contains address of allocated memory
exit (terminate execution)	10		
print character	11	\$a0 = character to print	
read character	12		\$v0 contains character read
open file	13	\$a0 = address of null-terminated string containing filename \$a1 = flags \$a2 = mode	\$v0 contains file descriptor (negative if error).

Systemfunktionen in MARS (3)



Service	Code in \$v0	Arguments	Result
read from file	14	\$a0 = file descriptor \$a1 = address of input buffer \$a2 = maximum number of characters to read	\$v0 contains number of characters read (0 if end-of-file, negative if error).
write to file	15	\$a0 = file descriptor \$a1 = address of output buffer \$a2 = number of characters to write	\$v0 contains number of characters written (negative if error).
close file	16	\$a0 = file descriptor	
exit2 (terminate with value)	17	\$a0 = termination result	

- Weitere Systemfunktionen finden Sie in der Hilfe von MARS



1. Einführung: MIPS Befehlssatz
2. Assemblersprache
 - Arithmetische Befehle
 - Lade- und Speicherbefehle
 - Logische Operationen
 - Kontrollbefehle
 - Multiplikation und Division
 - Zeichen und Zeichenfolgen
3. Funktionen und Prozeduren
 - Funktionsaufrufe und Stack
 - Registerkonventionen
 - Rekursion
4. Ein größeres Beispiel: Bubble Sort
5. Systemfunktionen
6. Überlauf-Behandlung
7. Floating Point Befehle
8. Maschinensprache
 - Darstellung von Befehlen im Rechner
 - Von-Neumann-Konzept
 - Adressierungsarten
9. Zusammenfassung

- Was passiert wenn ein Überlauf auftritt?
- Hängt ab vom Software-System / von der Programmiersprache
 - C und Java ignorieren Überläufe
 - Fortran nicht
- Nicht immer soll ein Überlauf angezeigt werden.
 - neue MIPS-Befehle: **addu, addiu, subu, sltu, sltiu, mulu, divu**
 - ⇒ alle „unsigned“ Befehle ignorieren Überläufe
 - ⇒ **Achtung: addiu und sltiu** erweitern das Vorzeichen ihrer 16-Bit Immediates
- „normale“ arithmetische Befehle (**add, addi, sub, slt, slti, mul, div**) lösen bei Überlauf eine *Exception* aus

Wie werden Überläufe behandelt?



- Eine *Exception* (Ausnahme) wird ausgelöst
= unerwartete **Unterbrechung** des Programmflusses (**von innen**)
- Wird in MIPS genau so gehandhabt wie ein *Interrupt*
= **Unterbrechung von außen**, z.B. durch Eingabegeräte
- *Exception Handler* wird gestartet
(Teil des Betriebssystems / *Operating System* (OS))
- **Sprung zu einer vordefinierten Adresse** (in MIPS: 0x800000180), um die Ausnahme (*Exception*) zu behandeln.



- Damit der *Exception Handler* unabhängig vom Programm arbeiten kann, sind Register **\$k0** (26) und **\$k1** (27) für das Betriebssystem reserviert.

- Spezielle MIPS-Register für Exceptions:

\$epc Adresse an der unterbrochen wurde (*exception program counter*)

\$cr Ursache der Ausnahme (*cause register*)

\$sr Ausnahme-Status (*status register*)

In den 80ern gab es für die Ausnahmebehandlung noch einen eigenen Coprozessor

- Spezielle MIPS-Befehle:

mfc0 **\$k0** , **\$epc** # **\$k0** = **\$epc** (*move from coprocessor 0*)

mtc0 **\$epc** , **\$k0** # **\$epc** = **\$k0** (*move to coprocessor 0*)

eret # PC = **\$epc** , (*error return*)

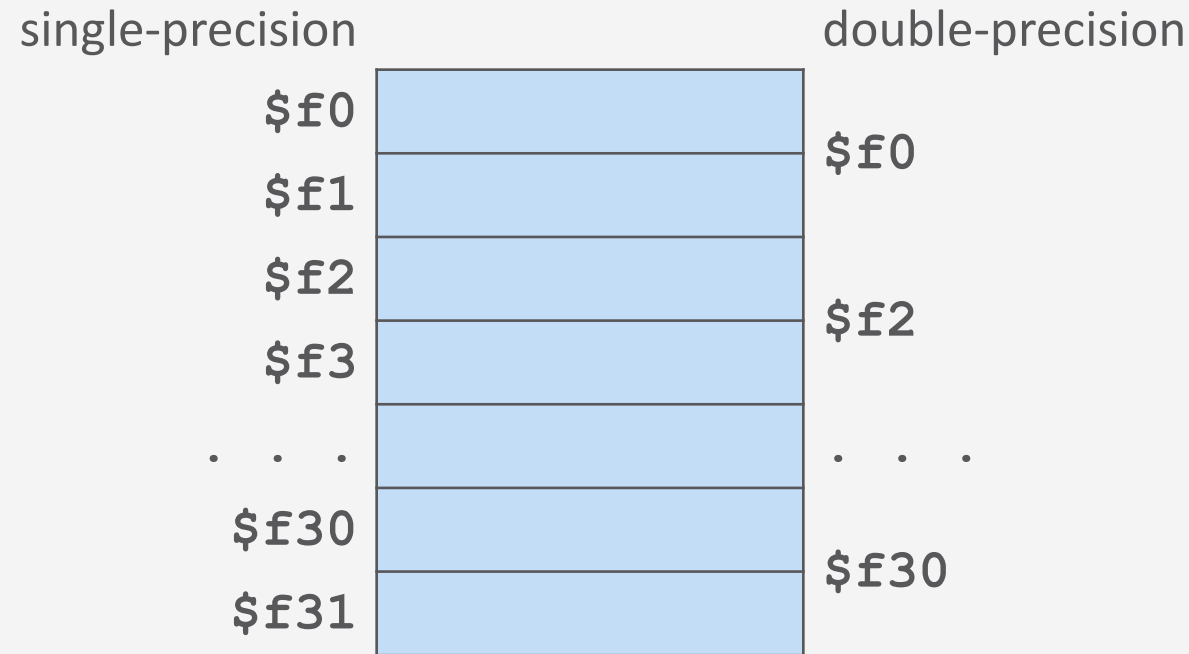


1. Einführung: MIPS Befehlssatz
2. Assemblersprache
 - Arithmetische Befehle
 - Lade- und Speicherbefehle
 - Logische Operationen
 - Kontrollbefehle
 - Multiplikation und Division
 - Zeichen und Zeichenfolgen
3. Funktionen und Prozeduren
 - Funktionsaufrufe und Stack
 - Registerkonventionen
 - Rekursion
4. Ein größeres Beispiel: Bubble Sort
5. Systemfunktionen
6. Überlauf-Behandlung
7. Floating Point Befehle
8. Maschinensprache
 - Darstellung von Befehlen im Rechner
 - Von-Neumann-Konzept
 - Adressierungsarten
9. Zusammenfassung

MIPS Floating Point: Register



- MIPS hat
 - 32 Floating-Point-Register mit einfacher Genauigkeit (*single-precision*) [\$f0, \$f1, ..., \$f31] **oder**
 - 16 Register mit doppelter Genauigkeit (*double-precision*) [\$f0, \$f2, ..., \$f30]



MIPS Floating Point: Befehle

In den 80ern gab es für die Floating-Point Unit (FPU) noch einen eigenen Coprozessor

FP add single	<code>add.s \$f0,\$f1,\$f2</code>	<code>\$f0=\$f1+\$f2</code>
FP sub. single	<code>sub.s \$f0,\$f1,\$f2</code>	<code>\$f0=\$f1-\$f2</code>
FP mult. single	<code>mul.s \$f0,\$f1,\$f2</code>	<code>\$f0=\$f1*\$f2</code>
FP div. single	<code>div.s \$f0,\$f1,\$f2</code>	<code>\$f0=\$f1/\$f2</code>
FP add double	<code>add.d \$f0,\$f2,\$f4</code>	<code>\$f0,\$f1 = \$f2,\$f3 + \$f4,\$f5</code>
FP sub. double	<code>sub.d \$f0,\$f2,\$f4</code>	<code>\$f0,\$f1 = \$f2,\$f3 - \$f4,\$f5</code>
FP mult. double	<code>mul.d \$f0,\$f2,\$f4</code>	<code>\$f0,\$f1 = \$f2,\$f3 * \$f4,\$f5</code>
FP div. double	<code>div.d \$f0,\$f2,\$f4</code>	<code>\$f0,\$f1 = \$f2,\$f3 / \$f4,\$f5</code>
load word coproc. 1	<code>lwc1 \$f0,100(\$s0)</code>	<code>\$f0=Mem[\$s0+100]</code>
store word copr. 1	<code>swc1 \$f0,100(\$s0)</code>	<code>Mem[\$s0+100]=\$f0</code>
branch on coproc.1 true <code>bc1t</code>	<code>25</code>	<code>if (cond) goto PC+4+100</code>
branch on coproc.1 false <code>bc1f</code>	<code>25</code>	<code>if (!cond) goto PC+4+100</code>
FP compare single	<code>c.lt.s \$f0,\$f1</code>	<code>cond = (\$f0 < \$f1)</code>
FP compare double	<code>c.ge.d \$f0,\$f2</code>	<code>cond = (\$f0,\$f1 >= \$f2,\$f3)</code>

MIPS Floating Point: Condition Code



- Vergleichsbefehle setzen den **condition code (cc)**
- Sukzessive Branch-Befehle testen, ob cc erfüllt ist (*true*) oder nicht (*false*)
- Beispiel: Suche nach kleinstem n , so dass gilt $0.5^n \leq 1.0 \cdot 10^{-9}$

```
int n = 1;  
float exp = 0.5;  
while (exp > 1e-9) {  
    exp = exp*0.5;  
    n++;  
}
```

$1.0 \cdot 10^{-9}$
in C Syntax

MIPS Floating Point: Beispiel



```
int n = 1;
float exp = 0.5;
while (exp > 1e-9) {
    exp = exp*0.5;
    n++;
}
```

Übersetzen Sie in MIPS-Assembler!

Nehmen Sie dabei an, dass die Konstanten 0.5 und 10^{-9} an den Adressen **fphalf(\$gp)** und **fptiny(\$gp)** gespeichert sind.

MIPS Floating Point: Beispiel



```
int n = 1;
float exp = 0.5;
while (exp > 1e-9) {
    exp = exp*0.5;
    n++;
}
```

**Übersetzen Sie in
MIPS-Assembler!**

```
addi    $t0,$zero,1        # n = 1
lwc1    $f0,fphalf($gp)    # exp = 0.5
lwc1    $f1,fptiny($gp)    # $f1 = 1e-9
lwc1    $f2,fphalf($gp)    # $f2 = 0.5
while:
    c.gt.s    $f0,$f1        # cc = exp>1e-9
    bclf     endwhile       # if (!cc) goto endwhile
    mul.s    $f0,$f0,$f2     # exp = exp*0.5
    addi    $t0,$t0,1        # n++
    j        while          # goto while
endwhile:    ...
```



1. Einführung: MIPS Befehlssatz
2. Assemblersprache
 - Arithmetische Befehle
 - Lade- und Speicherbefehle
 - Logische Operationen
 - Kontrollbefehle
 - Multiplikation und Division
 - Zeichen und Zeichenfolgen
3. Funktionen und Prozeduren
 - Funktionsaufrufe und Stack
 - Registerkonventionen
 - Rekursion
4. Ein größeres Beispiel: Bubble Sort
5. Systemfunktionen
6. Überlauf-Behandlung
7. Floating Point Befehle
8. Maschinensprache
 - Darstellung von Befehlen im Rechner
 - Von-Neumann-Konzept
 - Adressierungsarten
9. Zusammenfassung



- Wie werden Befehle im Rechner dargestellt?
- Befehle sind Bitfolgen.
 - Alles in einem digitalen Rechner ist binär.
- MIPS-Befehle: sehr einfach, reguläre Struktur
 - alle Befehle sind 32 Bits lang
 - Operanden immer an der gleichen Stelle
 - nur drei Befehlsformate (*instruction formats*)
- Register haben Nummern
 - \$s0 bis \$s7 entsprechen Register 16 bis 23
 - \$t0 bis \$t7 entsprechen Register 8 bis 15

Maschinensprache: Beispiel



- `$s0` bis `$s7` entsprechen Register 16 bis 23
- `$t0` bis `$t7` entsprechen Register 8 bis 15
- Beispiel: `add $t0, $s1, $s2`

Opcodes, Register, Function codes:
im MIPS reference sheet (green card)


Befehlsformat:

	op	rs	rt	rd	shamt	funct
Dezimal	0	17	18	8	0	32
Binär	000000	10001	10010	01000	00000	100000
	6 Bit	5 Bit	5 Bit	5 Bit	5 Bit	6 Bit

Achtung: In Assemblersprache Zielregister vorne, in Maschinensprache hinten!

MIPS Reference Sheet (Green Card)

①



MIPS Reference Data

CORE INSTRUCTION SET

NAME, MNEMONIC	FOR-MAT	OPERATION (in Verilog)	OPCODE / FUNCT (Hex)
Add	add	R R[rd] = R[rs] + R[rt]	(1) 0 / 20 _{hex} = 32 ₁₀
Add Immediate	addi	I R[rt] = R[rs] + SignExtImm	(1,2) 8 _{hex}
Add Imm. Unsigned	addiu	I R[rt] = R[rs] + SignExtImm	(2) 9 _{hex}
Add Unsigned	addu	R R[rd] = R[rs] + R[rt]	0 / 21 _{hex}
And	and	R R[rd] = R[rs] & R[rt]	0 / 24 _{hex}
And Immediate	andi	I R[rt] = R[rs] & ZeroExtImm	(3) c _{hex}
Branch On Equal	beq	I if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4) 4 _{hex}
Branch On Not Equal	bne	I if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4) 5 _{hex}
Jump	j	J PC=JumpAddr	(5) 2 _{hex}
Jump And Link	jal	J R[31]=PC+8;PC=JumpAddr	(5) 3 _{hex}

REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
\$zero	0	The Constant Value 0	N.A.
\$at	1	Assembler Temporary	No
\$v0-\$v1	2-3	Values for Function Results and Expression Evaluation	No
\$a0-\$a3	4-7	Arguments	No
\$t0-\$t7	8-15	Temporaries	No
\$s0-\$s7	16-23	Saved Temporaries	Yes
\$t8-\$t9	24-25	Temporaries	No
\$k0-\$k1	26-27	Reserved for OS Kernel	No
\$gp	28	Global Pointer	Yes
\$sp	29	Stack Pointer	Yes
\$fp	30	Frame Pointer	Yes
\$ra	31	Return Address	Yes

Maschinensprache: R-Format



	6 Bits	5 Bits	5 Bits	5 Bits	5 Bits	6 Bits
Dezimal	0	17	18	8	0	32
Binär	000000	10001	10010	01000	00000	100000
	op	rs	rt	rd	shamt	funct

- gleiches Format für alle **Befehle mit drei Registern** (log. + arithm. Befehle)
- **Felder** in diesem MIPS-Befehlsformat (R-Format, R für Register)
 - *op*: *opcode* oder Operationscode; Basisoperation des Befehls
 - *rs*: erstes Quellregister (*source register*)
 - *rt*: zweites Quellregister oder Zielregister (*target register*)
 - *rd*: Zielregister (*destination register*)
 - *shamt*: *shift amount*, nur für Schiebe-Befehle
 - *funct*: **Funktionscode** (*function code*). *op* und *funct* bestimmen die Operation (meistens)



- Was ist mit den **lw**- und **sw**-Befehlen?
 - **lw** \$t0, **1000** (\$t1) und **sw** \$t4, **12** (\$t1)
 - Nach Regelmäßigkeitsprinzip nur 5 Bits für den konstanten Offset
 - Offset auf $2^5 = 32$ begrenzt
 - oder verschiedene Befehlslängen?
- MIPS: Unterscheidung von zwei **Befehlsformaten**:
 - **R-Typ** oder **R-Format** (R für Register)
 - **I-Typ** oder **I-Format** (I für *Immediate* = Direktoperand)

Entwurfsprinzip 4: ***Good design demands compromises***
(Guter Entwurf erfordert Kompromisse)

- Beispiel I-Format-Befehl: **lw \$t0, 32(\$s2)**

	6 Bits	5 Bits	5 Bits	16 Bits
Dezimal	35	18	8	32
Binär	100011	10010	01000	0000 0000 0010 0000
	op	rs	rt	Konstante oder Adresse-Offset

- Im **lw**-Befehl gibt das **rt**-Feld das Zielregister an (*target register*).
- 16-Bit-Adress-Offset** → beliebiges Wort im Bereich von $-2^{15}..2^{15}-1$ ab Adresse im Basisregister **rs** (*source register*) kann geladen werden
- gleiches Format für **arithmetisch-logische Befehle mit Direktoperanden** (auch Konstanten im **addi**-Befehl sind auf $-2^{15}..2^{15}-1$ beschränkt)

Maschinensprache: Größeres Beispiel



- C/ Java: `A[300] = h + A[300];`
- MIPS:
`lw $t0,1200($t1)` # `$t0 = A[300]`
`add $t0,$s2,$t0` # `$t0 = h+$t0`
`sw $t0,1200($t1)` # `A[300] = $t0`

**Übersetzen Sie in
Maschinensprache! (als
Dezimalzahlen)**

Instr	Opcode	Funct
add	0	32
lw	35	
sw	43	

Reg. Name	Nummer
<code>\$t0-\$t7</code>	8-15
<code>\$s0-\$s7</code>	16-23

Maschinensprache: Größeres Beispiel



- C/ Java: `A[300] = h + A[300];`
- MIPS:
`lw $t0,1200($t1)` # `$t0 = A[300]`
`add $t0,$s2,$t0` # `$t0 = h+$t0`
`sw $t0,1200($t1)` # `A[300] = $t0`

**Übersetzen Sie in
Maschinensprache! (als
Dezimalzahlen)**

Instr	Opcode	Funct
add	0	32
lw	35	
sw	43	

Reg. Name	Nummer
<code>\$t0-\$t7</code>	8-15
<code>\$s0-\$s7</code>	16-23

op	rs	rt	Adress-Offset		
			rd	shamt	funct
35	9	8	1200		

Maschinensprache: Größeres Beispiel



- C/ Java: `A[300] = h + A[300];`
- MIPS:
`lw $t0,1200($t1)` # `$t0 = A[300]`
`add $t0,$s2,$t0` # `$t0 = h+$t0`
`sw $t0,1200($t1)` # `A[300] = $t0`

**Übersetzen Sie in
Maschinensprache! (als
Dezimalzahlen)**

Instr	Opcode	Funct
add	0	32
lw	35	
sw	43	

Reg. Name	Nummer
<code>\$t0-\$t7</code>	8-15
<code>\$s0-\$s7</code>	16-23

op	rs	rt	Adress-Offset		
			rd	shamt	funct
35	9	8	1200		
0	18	8	8	0	32

Maschinensprache: Größeres Beispiel



- C/ Java: `A[300] = h + A[300];`
- MIPS:

<code>lw \$t0,1200(\$t1)</code>	<code># \$t0 = A[300]</code>
<code>add \$t0,\$s2,\$t0</code>	<code># \$t0 = h+\$t0</code>
<code>sw \$t0,1200(\$t1)</code>	<code># A[300] = \$t0</code>

**Übersetzen Sie in
Maschinensprache! (als
Dezimalzahlen)**

Instr	Opcode	Funct
add	0	32
lw	35	
sw	43	

Reg. Name	Nummer
\$t0-\$t7	8-15
\$s0-\$s7	16-23

op	rs	rt	Adress-Offset		
			rd	shamt	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		



1. Einführung: MIPS Befehlssatz
2. Assemblersprache
 - Arithmetische Befehle
 - Lade- und Speicherbefehle
 - Logische Operationen
 - Kontrollbefehle
 - Multiplikation und Division
 - Zeichen und Zeichenfolgen
3. Funktionen und Prozeduren
 - Funktionsaufrufe und Stack
 - Registerkonventionen
 - Rekursion
4. Ein größeres Beispiel: Bubble Sort
5. Systemfunktionen
6. Überlauf-Behandlung
7. Floating Point Befehle
8. Maschinensprache
 - Darstellung von Befehlen im Rechner
 - Von-Neumann-Konzept
 - Adressierungsarten
9. Zusammenfassung

Von-Neumann- oder *Stored-Program-Konzept*



Hochschule für
Wirtschaft und Recht Berlin
Berlin School of Economics and Law

- Die ersten Rechner waren an ein festes Programm gebunden.
- John von Neumann 1945: *Stored-Program-Konzept*
- Befehle werden wie Zahlen dargestellt.
- Programme werden im Hauptspeicher gespeichert, um wie Daten gelesen oder geschrieben werden zu können.
 - Wenn der Rechner eine Zahl als Befehl interpretiert, dann ist es ein Befehl.
 - Wenn der Rechner eine Zahl als Daten interpretiert, dann sind es Daten.
- Rechner ist eine „Metamaschine“
 - Durch Programmwechsel ändert sich die Maschine.

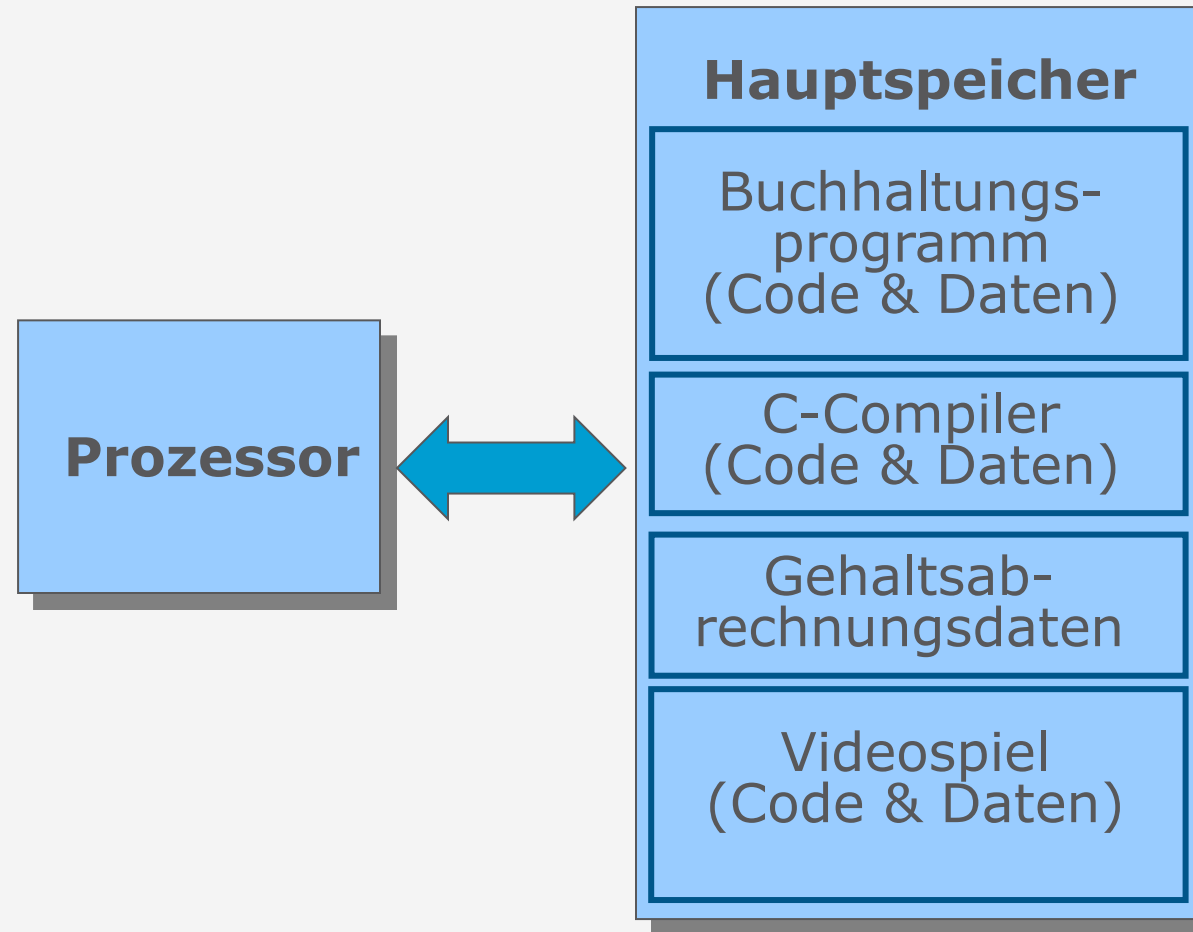


John von Neumann

Von-Neumann- oder *Stored-Program-Konzept*



Hochschule für
Wirtschaft und Recht Berlin
Berlin School of Economics and Law



- **Fetch-and-Execute-Zyklus:**
 - Lese den nächsten Befehl aus dem Hauptspeicher
 - Führe Operation aus
 - Bits im Befehl geben an welche Operation auf welchen Operanden ausgeführt werden muss
 - Lese den nächsten Befehl
 - usw.
- **Befehlszähler** (*program counter*, Register **PC**) enthält die Adresse des nächsten Befehls.

```
while (true){  
    instr = Memory[PC];  
    execute instr;  
    PC = PC+4;  
}
```



1. Einführung: MIPS Befehlssatz
2. Assemblersprache
 - Arithmetische Befehle
 - Lade- und Speicherbefehle
 - Logische Operationen
 - Kontrollbefehle
 - Multiplikation und Division
 - Zeichen und Zeichenfolgen
3. Funktionen und Prozeduren
 - Funktionsaufrufe und Stack
 - Registerkonventionen
 - Rekursion
4. Ein größeres Beispiel: Bubble Sort
5. Systemfunktionen
6. Überlauf-Behandlung
7. Floating Point Befehle
8. Maschinensprache
 - Darstellung von Befehlen im Rechner
 - Von-Neumann-Konzept
 - Adressierungsarten
9. Zusammenfassung



- Verschiedene Formen der Operanden-Behandlung werden als **Adressierungsarten** (*addressing modes*) bezeichnet.
 1. **Registeradressierung**: Operand steht im Register
 - `add $t0, $a2, $s4`
 2. **Basis-Adressierung**:
Adresse als Summe von Register (= Basisadresse) und Konstante
 - `lw $s0, 4($t3) # $s0 = Mem[$t3+4]`
 3. **Direkte Adressierung**: Direktoperand als Konstante im Befehl
 - `ori $t0, $t0, 255`
 4. **Befehlszählerrelative Adressierung**: für bedingte Verzweigungen
 5. **Pseudo-direkte Adressierung**: für unbedingte Sprünge

Befehlszählerrelative Adressierung

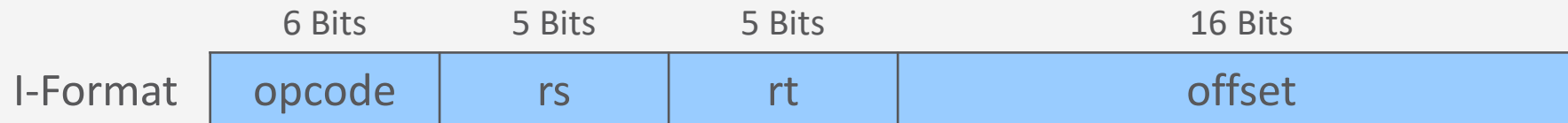


- Bedingte Sprung-Befehle:

`bne $s0,$s1,label# if ($s0!= $s1) goto label`

`beq $s0,$s1,label# if ($s0== $s1) goto label`

- Format:



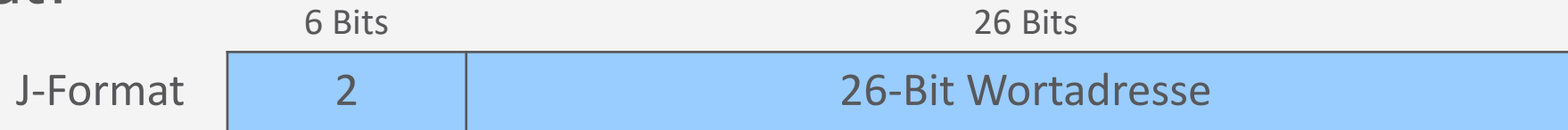
```
1000      bne  $t0,$s1,end
1004      addi $s0,$s0,1
1008      j    while
1012 end:  ...
```

- Befehlszähler (*program counter*, **PC**) enthält die Adresse des aktuellen Befehls
- 16-bit Adress-Offset in Verzweigungen ist **relativ zum Folgebefehl** ($PC + 4$)
- Offset ist außerdem **Wortoffset**: \Rightarrow Zieladresse: $PC + 4 + 4 * \text{Offset}$

Pseudo-direkte Adressierung



- Unbedingter Sprung-Befehl: `j label # goto label`
- J-Format:



- Gegebener Offset wird zur Berechnung der tatsächlichen Zieladresse erneut mit 4 multipliziert (zwei Nullen anhängen, 26 → 28 Bit).
- Fehlende 4 Bits werden vom PC genommen.
- Beispiel:

2	2000
---	------

`# PC31-0 = (PC+4)31-28 # 800027-0`

- Adressgrenze von 256 MB (64 Millionen Befehle)



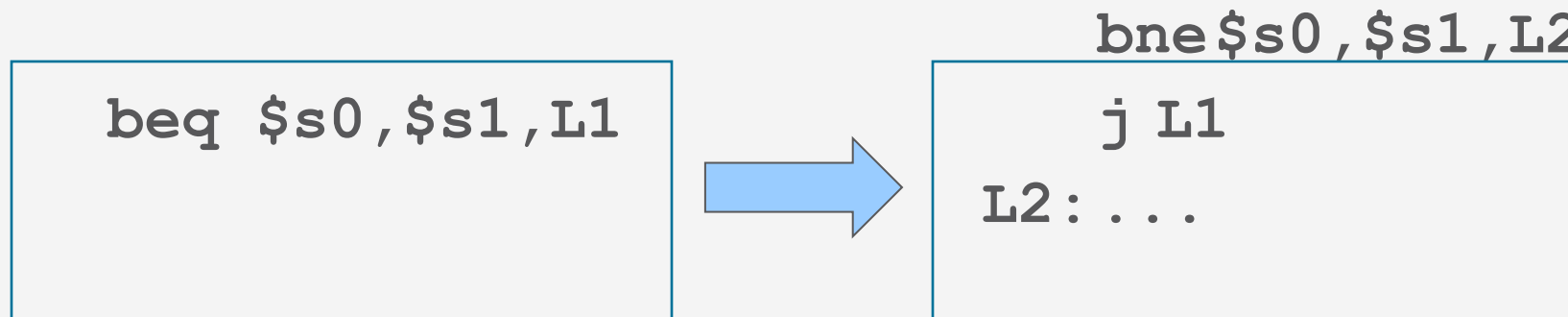
- Mit **beq** und **bne** kann man 2^{15} Befehle vorwärts und $2^{15}-1$ Befehle rückwärts springen.
 - Die meisten Sprünge finden innerhalb eines beschränkten Adressbereichs statt.
- Selten muss weiter verzweigt werden.
- Wie können wir weiter verzweigen als 2^{15} Befehle?

```
beq $s0, $s1, L1
```

Weite Verzweigung



- Mit **beq** und **bne** kann man 2^{15} Befehle vorwärts und $2^{15}-1$ Befehle rückwärts springen.
 - Die meisten Sprünge finden innerhalb eines beschränkten Adressbereichs statt.
- Selten muss weiter verzweigt werden.
- Wie können wir weiter verzweigen als 2^{15} Befehle?





1. Registeradressierung (Operand steht im Register.)

```
add $t0, $a2, $s4
```

2. Basis- oder Displacement-Adressierung

(Adresse ist Summe von Register und Konstante im Befehl.)

```
lw $s0, 4($t3) # $s0 = Mem[$t3+4]
```

3. Direkte Adressierung (Operand ist Konstante im Befehl.)

```
ori $t0, $t0, 255
```

4. Befehlszählerrelative Adressierung

([Sprung]-Adresse ist Summe von PC+4 und 4·Offset)

```
beq $t0, $a1, 100 # PC = PC+4+4x100
```

5. Pseudo-direkte Adressierung

([Sprung]-Adresse ist eine Konkatenation von den oberen 4 Bits von PC+4 und der 26-Bit Wortadresse um zwei Nullen erweitert (= 4·Offset))

```
j 1000 # PC = (PC+4)31..28 # (4·1000)
```



■ Übersetzen Sie folgenden MIPS-Assemblercode zu MIPS-Maschinencode.

- Nehmen Sie an, die Schleife beginnt an Adresse 80000 im Hauptspeicher.
- Verwenden Sie dezimale Werte für alle Befehlsfelder

```
loop: sll    $t1,$s3,2
      add    $t1,$t1,$s6
      lw     $t0,0($t1)
      bne    $t0,$s5,exit
      addi   $s3,$s3,1
      j      loop

exit:
```

Instr	Opcode	Funct
sll	0	0
add	0	32
lw	35	
bne	5	
addi	8	
j	2	

Name	Nummer
\$zero	0
\$v0-\$v1	2-3
\$a0-\$a3	4-7
\$t0-\$t7	8-15
\$s0-\$s7	16-23
\$t8-\$t9	24-25
\$gp	28
\$sp	29
\$fp	30
\$ra	31

MIPS-Maschinensprache: Lösung



loop: sll \$t1,\$s3,2	op	rs	rt	rd	shamt	func	80000
add \$t1,\$t1,\$s6	op	rs	rt	rd	shamt	func	80004
lw \$t0,0(\$t1)	op	rs	rt	offset			80008
bne \$t0,\$s5,exit	op	rs	rt	offset			80012
addi \$s3,\$s3,1	op	rs	rt	imm			80016
j loop	op	addr					80020
exit:							80024

Instr	Opcode	Funct
sll	0	0
add	0	32
lw	35	
bne	5	
addi	8	
j	2	

MIPS-Maschinensprache: Lösung



loop: sll \$t1,\$s3,2	0	rs	rt	rd	shamt	0	80000
add \$t1,\$t1,\$s6	0	rs	rt	rd	shamt	32	80004
lw \$t0,0(\$t1)	35	rs	rt	offset			80008
bne \$t0,\$s5,exit	5	rs	rt	offset			80012
addi \$s3,\$s3,1	8	rs	rt	imm			80016
j loop	2	addr					80020
exit:							80024

Instr	Opcode	Funct
sll	0	0
add	0	32
lw	35	
bne	5	
addi	8	
j	2	

MIPS-Maschinensprache: Lösung



loop: sll \$t1,\$s3,2	0	rs	rt	rd	shamt	0	80000
add \$t1,\$t1,\$s6		Shift Left Logical	sll	R	R[rd] = R[rt] << shamt		0 / 00 _{hex}
lw \$t0,0(\$t1)		Shift Right Logical	srl	R	R[rd] = R[rt] >> shamt		0 / 02 _{hex}
bne \$t0,\$s5,exit		Store Byte	sb	I	M[R[rs]+SignExtImm](7:0) = R[rt](7:0)		(2) 28 _{hex}
addi \$s3,\$s3,1	8	rs	rt	imm			80016
j loop	2	addr					80020
exit:							80024

Name	\$zero	\$v0-\$v1	\$a0-\$a3	\$t0-\$t7	\$s0-\$s7	\$t8-\$t9	\$gp	\$sp	\$fp	\$ra
Nummer	0	2-3	4-7	8-15	16-23	24-25	28	29	30	31

MIPS-Maschinensprache: Lösung



loop: sll \$t1,\$s3,2	0	0	19	9	2	0	80000
add \$t1,\$t1,\$s6		Shift Left Logical sll R R[rd] = R[rt] << shamt 0 / 00 _{hex}					
lw \$t0,0(\$t1)		Shift Right Logical srl R R[rd] = R[rt] >> shamt 0 / 02 _{hex}					
bne \$t0,\$s5,exit		Store Byte sb I M[R[rs]+SignExtImm](7:0) = R[rt](7:0) (2) 28 _{hex}					
	.						
addi \$s3,\$s3,1	8	rs	rt	imm		80016	
j loop	2	addr				80020	
exit:						80024	

Name	\$zero	\$v0- \$v1	\$a0- \$a3	\$t0- \$t7	\$s0- \$s7	\$t8- \$t9	\$gp	\$sp	\$fp	\$ra
Nummer	0	2-3	4-7	8-15	16-23	24-25	28	29	30	31

MIPS-Maschinensprache: Lösung



loop: sll \$t1,\$s3,2	0	0	19	9	2	0	80000
add \$t1,\$t1,\$s6	0	rs	rt	rd	shamt	32	80004
lw \$t0,0(\$t1)	35	rs	rt	offset			80008
bne \$t0,\$s5,exit	5	rs	rt	offset			80012
addi \$s3,\$s3,1	8	rs	rt	imm			80016
j loop	2	addr					80020
exit:							80024

Name	\$zero	\$v0- \$v1	\$a0- \$a3	\$t0- \$t7	\$s0- \$s7	\$t8- \$t9	\$gp	\$sp	\$fp	\$ra
Nummer	0	2-3	4-7	8-15	16-23	24-25	28	29	30	31

MIPS-Maschinensprache: Lösung



loop: sll \$t1,\$s3,2	0	0	19	9	2	0	80000
add \$t1,\$t1,\$s6	0	9	22	9	0	32	80004
lw \$t0,0(\$t1)	35	rs	rt	offset			80008
bne \$t0,\$s5,exit	5	rs	rt	offset			80012
addi \$s3,\$s3,1	8	rs	rt	imm			80016
j loop	2	addr					80020
exit:							80024

Name	\$zero	\$v0- \$v1	\$a0- \$a3	\$t0- \$t7	\$s0- \$s7	\$t8- \$t9	\$gp	\$sp	\$fp	\$ra
Nummer	0	2-3	4-7	8-15	16-23	24-25	28	29	30	31

MIPS-Maschinensprache: Lösung



bne: Befehlszählerrelative Adressierung, relativ zum Folgebefehl, als Wortadresse!

loop: sll \$t1,\$s3,2	0	0	19	9	2	0	80000
add \$t1,\$t1,\$s6	0	9	22	9	0	32	80004
lw \$t0,0(\$t1)	35	9	8	0			80008
bne \$t0,\$s5,exit	5	rs	rt	offset			80012
addi \$s3,\$s3,1	8	rs	rt	imm			80016
j loop	2	addr					80020
exit:							80024

+ 2

Name	\$zero	\$v0- \$v1	\$a0- \$a3	\$t0- \$t7	\$s0- \$s7	\$t8- \$t9	\$gp	\$sp	\$fp	\$ra
Nummer	0	2-3	4-7	8-15	16-23	24-25	28	29	30	31

MIPS-Maschinensprache: Lösung



bne: Befehlszählerrelative Adressierung, relativ zum Folgebefehl, als Wortadresse!

loop: sll \$t1,\$s3,2	0	0	19	9	2	0	80000
add \$t1,\$t1,\$s6	0	9	22	9	0	32	80004
lw \$t0,0(\$t1)	35	9	8	0			80008
bne \$t0,\$s5,exit	5	8	21	2			80012
addi \$s3,\$s3,1	8	rs	rt	imm			80016
j loop	2	addr					80020
exit:							80024

+ 2

Zieladresse = PC + 4 + 4 * offset

Name	\$zero	\$v0-\$v1	\$a0-\$a3	\$t0-\$t7	\$s0-\$s7	\$t8-\$t9	\$gp	\$sp	\$fp	\$ra
Nummer	0	2-3	4-7	8-15	16-23	24-25	28	29	30	31

MIPS-Maschinensprache: Lösung



j: Pseudo-direkte Adressierung!

loop:	sll \$t1,\$s3,2	0	0	19	9	2	0	80000
	add \$t1,\$t1,\$s6	0	9	22	9	0	32	80004
	lw \$t0,0(\$t1)	35	9	8		0		80008
	bne \$t0,\$s5,exit	5	8	21		2		80012
	addi \$s3,\$s3,1	8	19	19		1		80016
	j loop	2				addr		80020
exit:								80024

$$\text{Zieladresse} = (\text{PC}+4)_{31..28} \# (4 \cdot \text{addr}) \Rightarrow \text{addr} = \text{Zieladresse}_{27..0} / 4 = 80000 / 4 = 20000$$

Name	\$zero	\$v0- \$v1	\$a0- \$a3	\$t0- \$t7	\$s0- \$s7	\$t8- \$t9	\$gp	\$sp	\$fp	\$ra
Nummer	0	2-3	4-7	8-15	16-23	24-25	28	29	30	31

MIPS-Maschinensprache: Lösung



j: Pseudo-direkte Adressierung!

loop:	sll \$t1,\$s3,2	0	0	19	9	2	0	80000
	add \$t1,\$t1,\$s6	0	9	22	9	0	32	80004
	lw \$t0,0(\$t1)	35	9	8		0		80008
	bne \$t0,\$s5,exit	5	8	21		2		80012
	addi \$s3,\$s3,1	8	19	19		1		80016
	j loop	2			20000			80020
exit:								80024

Zieladresse = $(PC+4)_{31..28} \# (4 \cdot \text{addr}) \Rightarrow \text{addr} = \text{Zieladresse}_{27..0} / 4 = 80000 / 4 = 20000$

Name	\$zero	\$v0- \$v1	\$a0- \$a3	\$t0- \$t7	\$s0- \$s7	\$t8- \$t9	\$gp	\$sp	\$fp	\$ra
Nummer	0	2-3	4-7	8-15	16-23	24-25	28	29	30	31



1. Einführung: MIPS Befehlssatz
2. Assemblersprache
 - Arithmetische Befehle
 - Lade- und Speicherbefehle
 - Logische Operationen
 - Kontrollbefehle
 - Multiplikation und Division
 - Zeichen und Zeichenfolgen
3. Funktionen und Prozeduren
 - Funktionsaufrufe und Stack
 - Registerkonventionen
 - Rekursion
4. Ein größeres Beispiel: Bubble Sort
5. Systemfunktionen
6. Überlauf-Behandlung
7. Floating Point Befehle
8. Maschinensprache
 - Darstellung von Befehlen im Rechner
 - Von-Neumann-Konzept
 - Adressierungsarten
9. Zusammenfassung

Zusammenfassung: MIPS-Register



Name	Register- nummer	Verwendung	Bei Aufruf beibehalten?
\$zero	0	Konstante 0	-
\$v0-\$v1	2-3	Werte für Ergebnisse und für die Auswertung von Ausdrücken	nein
\$a0-\$a3	4-7	Argumente	nein
\$t0-\$t7	8-15	temporäre Variablen	nein
\$s0-\$s7	16-23	gespeicherte Variablen	ja
\$t8-\$t9	24-25	weitere temporäre Variablen	nein
\$gp	28	globaler Zeiger (Global pointer)	ja
\$sp	29	Kellerzeiger (Stack pointer)	ja
\$fp	30	Rahmenzeiger (Frame pointer)	ja
\$ra	31	Rücksprungadresse	ja

Zusammenfassung: MIPS-Befehlssatz



Kategorie	Befehl	Beispiel	Bedeutung	Anmerkungen
Arithme- tisch	add	add \$s1,\$s2,\$s3	$\$s1 = \$s2 + \$s3$	3 Registeroperanden
	subtract	sub \$s1,\$s2,\$s3	$\$s1 = \$s2 - \$s3$	3 Registeroperanden
	add immediate	addi \$s1,\$s2,100	$\$s1 = \$s2 + 100$	Konstante addieren
Daten - transfer	load word	lw \$s1,100(\$s2)	$\$s1 = \text{Mem}[\$s2 + 100]$	Wort vom Hauptspeicher in Register
	store word	sw \$s1,100(\$s2)	$\text{Mem}[\$s2 + 100] = \$s1$	Wort von Register in Hauptspeicher
	load byte	lb \$s1,100(\$s2)	$\$s1 = \text{Mem}[\$s2 + 100]$	Byte vom Hauptspeicher in Register
	store byte	sb \$s1,100(\$s2)	$\text{Mem}[\$s2 + 100] = \$s1$	Byte von Register in Hauptspeicher
	load upper imm.	lui \$s0,100	$\$s1 = 100 \times 2^{16}$	Konstante in obere 16 Bit

blau = richtige Kategorie?

Zusammenfassung: MIPS-Befehlssatz



Kategorie	Befehl	Beispiel	Bedeutung	Anmerkungen
Logisch	and	and \$s1,\$s2,\$s3	$\$s1 = \$s2 \& \$s3$	Bitweise UND
	or	or \$s1,\$s2,\$s3	$\$s1 = \$s2 \mid \$s3$	Bitweise ODER
	nor	nor \$s1,\$s2,\$s3	$\$s1 = \sim(\$s2 \mid \$s3)$	Bitweise NOR
	and imm.	andi \$s1,\$s2,7	$\$s1 = \$s2 \& 7$	Bitweise UND mit Konst.
	or imm.	ori \$s1,\$s2,7	$\$s1 = \$s2 \mid 7$	Bitweise ODER mit Konst.
	shift left logical	sll \$s1,\$s2,10	$\$s1 = \$s2 \ll 10$	Linksschieben
	shift right logical	srl \$s1,\$s2,10	$\$s1 = \$s2 \gg 10$	Rechtschieben
Verzwei- gung	branch on equal	beq \$s1,\$s2,25	if ($\$s1 == \$s2$) PC = PC + 4 + 100	Befehlszählerrelative Verzweigung
	branch on not equal	bne \$s1,\$s2,25	if ($\$s1 \neq \$s2$) PC = PC + 4 + 10	Befehlszählerrelative Verzweigung
	set on less than	slt \$s0,\$s1,\$s2	$\$s0 = (\$s1 < \$s2)$	Vergleich, kleiner als
	set less than imm.	slt \$s0,\$s1,10	$\$s0 = (\$s1 < 10)$	Kleiner als Konstante

blau = richtige Kategorie?

Zusammenfassung: MIPS-Befehlssatz



Hochschule für
Wirtschaft und Recht Berlin
Berlin School of Economics and Law

Kategorie	Befehl	Beispiel	Bedeutung	Anmerkungen
Sprung	jump	j 2500	PC = 10000	Unbedingter Sprung
	jump register	jr \$ra	PC = \$ra	Für Prozedurrücksprung, <i>Switch</i> -Anweisung
	jump and link	jal 2500	\$ra = PC+4; PC = 10000	

Zusammenfassung: Maschinensprache



- Befehle sind Zahlen.
 - Assemblersprache bieten „komfortable“ symbolische Darstellungen.
 - Maschinensprache entspricht der realen Darstellung in der Maschine.
- Assembler kann **Pseudobefehle** anbieten.
 - z. B. `move $t0,$t1` wird übersetzt zu `add $t0,$t1,$zero`.
- MIPS:
 - einfache Befehle (alle 32 Bits lang)
 - sehr strukturiert
 - nur drei Befehlsformate:

R-Format	op	rs	rt	rd	shamt	func
I-Format	op	rs	rt	16-Bit Konstante		
J-Format	op	26-Bit Konstante				

Zusammenfassung: MIPS Befehlsformate



	6 Bit	5 Bit	5 Bit	5 Bit	5 Bit	6 Bit
R-Format	op	rs	rt	rd	shamt	func
I-Format	op	rs	rt	16-bit address offset / immediate		
J-Format	op	26-bit word address				

- Alle MIPS-Befehle sind 32 Bits lang.
- R-Format für arithmetische/logische Befehle mit 3 Register-Operanden oder Schiebebefehle
- I-Format für Datentransfer, Immediate, bedingte Verzweigungen
- J-Format für Sprünge

Zusammenfassung: Entwurfsprinzipien



Hochschule für
Wirtschaft und Recht Berlin
Berlin School of Economics and Law

- *Simplicity favors regularity*
(Einfachheit begünstigt Regelmäßigkeit)
- *Smaller is faster*
(Kleiner ist schneller)
- *Make the common case fast*
(Optimiere den häufig vorkommenden Fall)
- *Good design demands compromises*
(Ein guter Entwurf fordert Kompromisse)