MIXED SIGNAL
MSC
CIRCUIT DESIGN

Technische Universität Berlin

Befehle

Einstein-Prof. Dr.-Ing. Friedel Gerfers

Kapitel 4:
Befehle – Die Sprache des Rechners

Ziele

Nach diesem Kapitel sind Sie in der Lage:

- C-Anweisungen in MIPS-Assemblersprache umzusetzen
- komplexere Programmierkonstrukte zu realisieren
  - if-Anweisungen
  - while- und for-Schleifen
  - switch-Anweisungen
  - (rekursive) Funktionen

- MIPS-Registerkonventionen zu befolgen
- zu erklären was ein Stack (Stapel) ist und wofür er gebraucht wird
- ein Assemblerprogramm in Maschinensprache umzusetzen und umgekehrt
- und vieles mehr…

Inhalte

- Arithmetische Befehle
- Lade- und Speicherbefehle
- Darstellung von Befehlen im Rechner
- Logische Operationen
- Befehle zum Treffen von Entscheidungen
- Unterstützung von Funktionen
- Zeichen und Zeichenfolgen
- MIPS-Adressierung
- Ein komplettes Beispiel

Befehle (Instructions)

- Befehle sind die „Sprache“ des Rechners

- Viel primitiver als höhere Programmiersprachen
  - z. B. keine while-Schleifen, if-Anweisungen

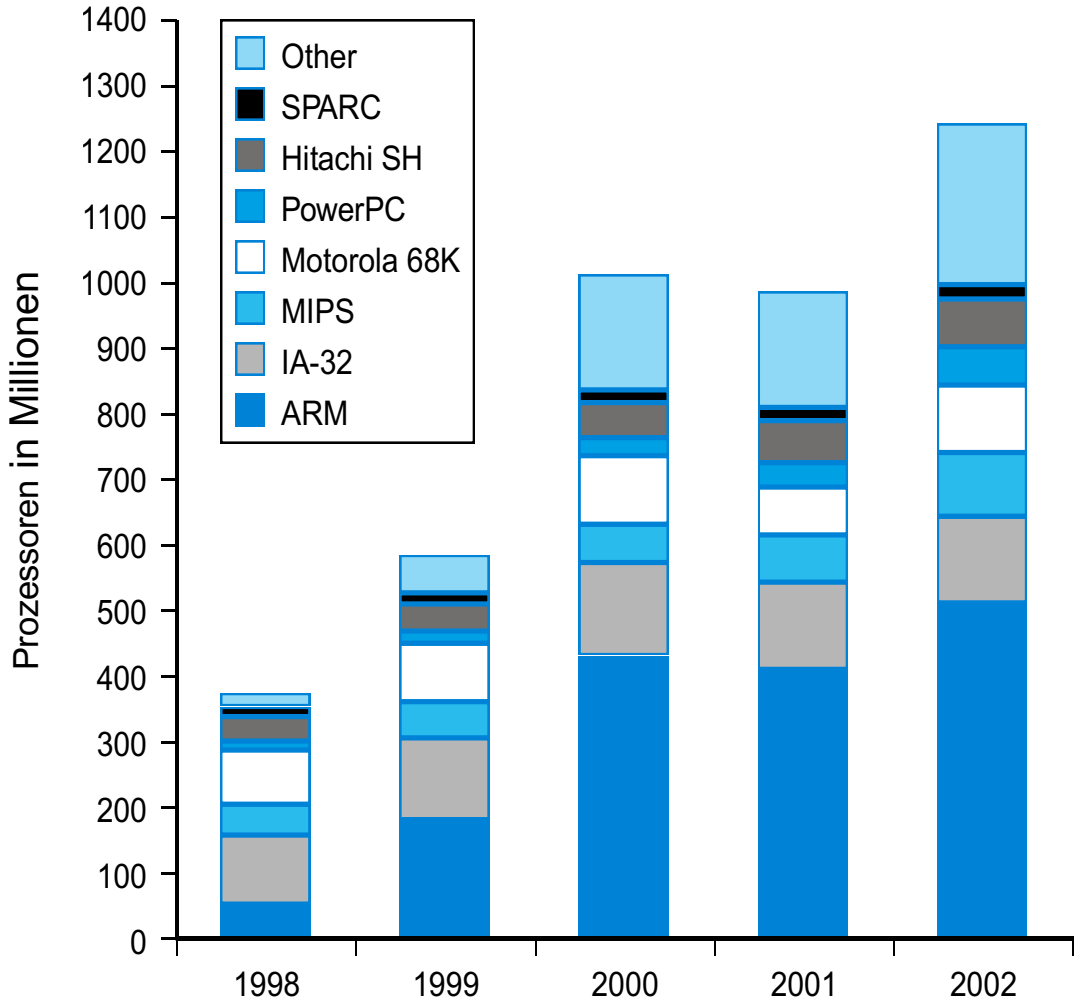
- Sehr restriktiv
  - z. B. viele MIPS-Befehle haben genau 3 Operanden

- Rechnerarchitekten verfolgen ein gemeinsames Ziel:

- Eine Sprache zu finden, die das Konstruieren der Hardware und des Compilers erleichtert und dabei die Leistung maximiert und die Kosten/Aufwand minimiert.

Microprocessor without Interlocked Pipeline Stages [=MIPS]

- Als Beispiel nehmen wird den MIPS-Befehlssatz
  - ähnlich zu anderen Befehlssätzen entwickelt seit den 80er-Jahren
  - wird/wurde gebraucht von NEC, Nintendo, Silicon Graphics, Sony...



Befehlskategorien

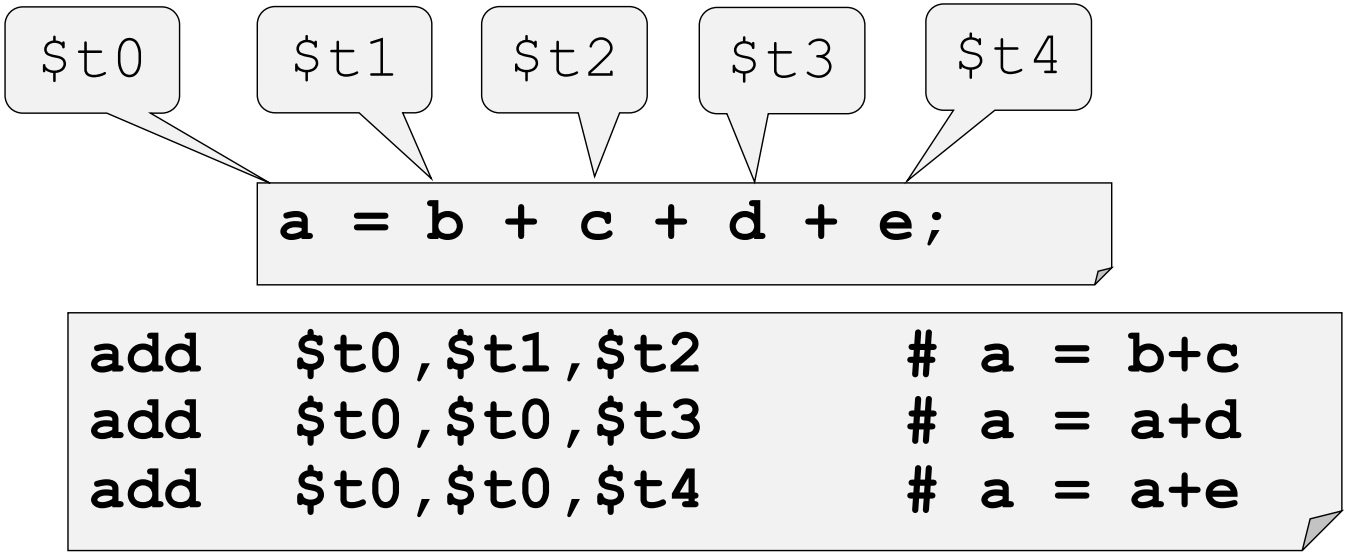
- Arithmetische Befehle
  - Integer
  - Gleitpunkt (Floating Point)
- Datentransfer-Befehle
  - Laden und speichern (Load & Store)
- Kontrollbefehle
  - Sprung (Jump)
  - bedingte Verzweigungen (Conditional Branch)
  - zur Unterstützung von Prozeduren (Call & Return)

Arithmetische Befehle / 1

- Die meisten arithmetischen Befehle haben 3 Operanden
- Folge der Operanden steht fest (Ziel vorne)
- Operanden sind Register
  - Speicherbereiche, die innerhalb eines Prozessors direkt mit der eigentlichen Recheneinheit verbunden sind und die unmittelbaren Operanden und Ergebnisse aller Berechnungen aufnehmen (Wikipedia)
- 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. Deren Bedeutung wird später klar.

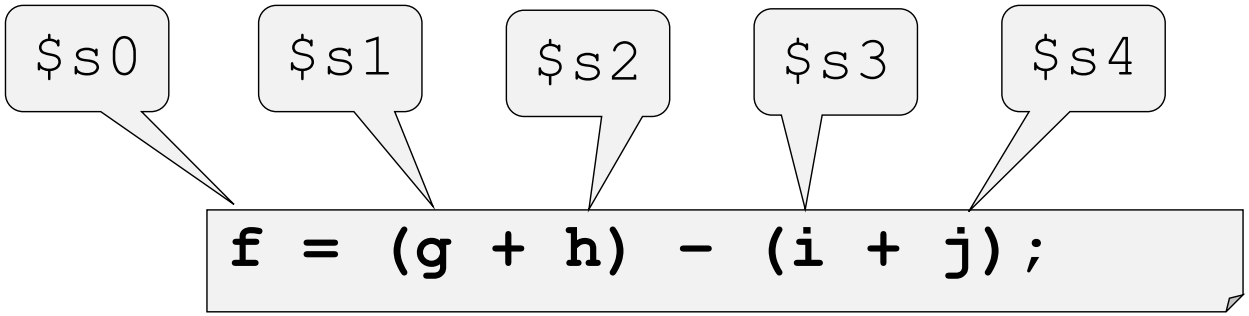
Arithmetische Befehle /2
Zuweisungen mit >3 Operanden

- Genau 3 Operanden entspricht Philosophie, die HW einfach zu halten
  - HW für eine variable Anzahl an Operanden ist komplexer.
- Entwurfsprinzip 1: Simplicity favors regularity (Einfachheit begünstigt Regelmäßigkeit)



Arithmetische Befehle / 3

- Etwas komplexeres Beispiel:




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

Operanden

- Operand:
  - Objekt, auf das eine mathematische Funktion oder ein Operator angewendet wird

- MIPS kennt folgende Typen:
  - Registeroperanden
  - Speicheroperanden
  - Konstante oder Direktoperanden

MIPS Register 0..31

- MIPS verfügt über 32 integer- (fixed-point-) Register
- In Assembler angedeutet als $0, $1, …, $31 oder mit den symbolischen Namen $zero, $at, …,$ra.
  - Bedeutung dieser symbolischen Namen wird später erklärt
  - Z.B. $ra - return address
- Register 0 ($0 oder $zero) ist immer Null
- Register sind 32 Bit breit (= ein Wort (word) in MIPS)

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

MIPS Register 0..31

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

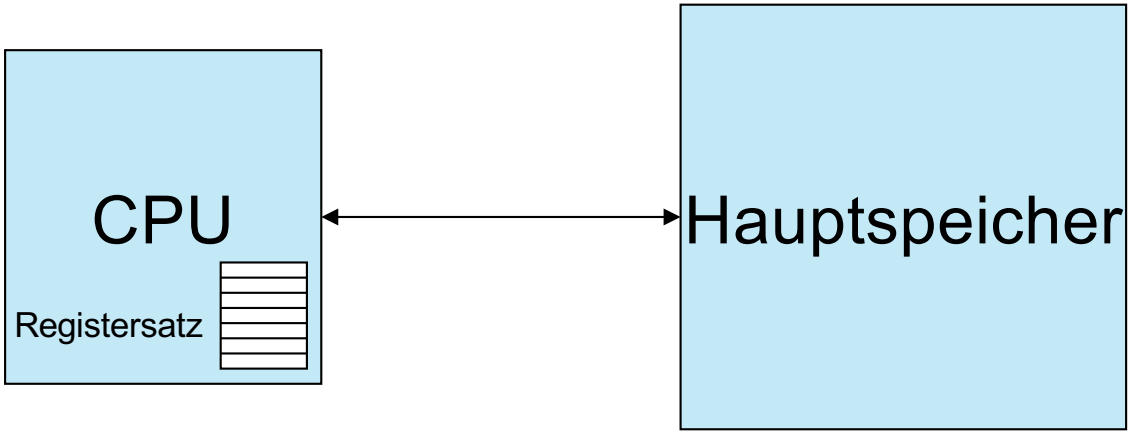
Wert
0

32 MIPS Register

- Wieso nur 32 int Register?
  - mehr passen nicht im Befehlsformat (später mehr)
  - Entwurfsprinzip 2: Smaller is faster (Kleiner ist schneller)
  - Eine große Anzahl von Registern kann zu einer längeren Taktzykluszeit führen.

Speicheroperanden

- 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)



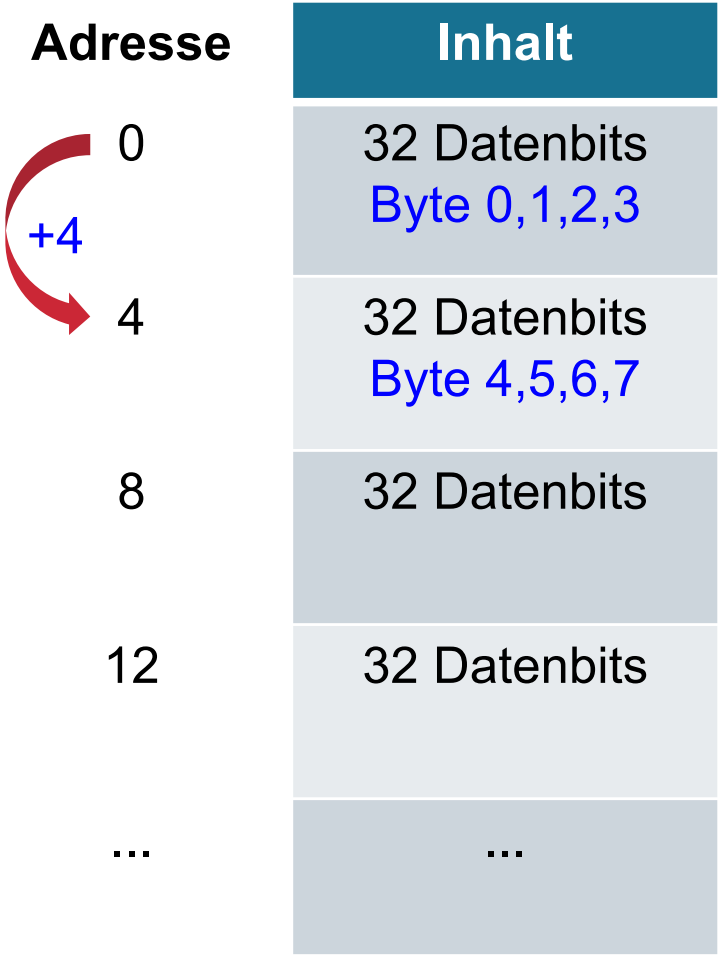
Speicherorganisation / 1

- Hauptspeicher kann als ein großes Array von Bytes betrachtet werden.
- Eine Speicheradresse ist ein Index in diesem Array.
- „Byte-Adressierung“ bedeutet, dass die Adresse auf einem Byte zeigt.

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
. . .	. . .

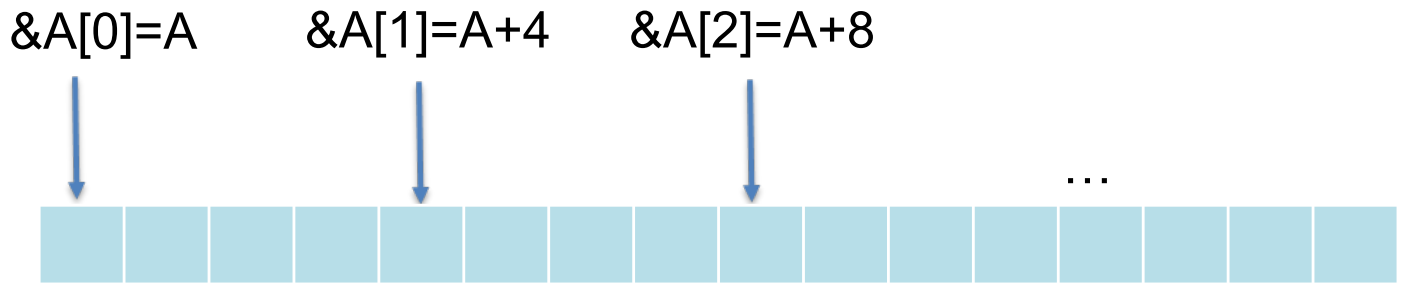
Speicherorganisation / 2

- Meisten Programme benutzen Wörter (words = 32-bits) statt Bytes
- Für MIPS ist ein Wort gleich 32 Bits oder 4 Bytes.
- Der Hauptspeicher kann also betrachtet werden als
  - ein Array von Bytes mit Adressen 0, 1, 2, 3, …
  - ein Array von Wörtern mit Adressen 0, 4, 8, 12, ...



Adresskalkulation

- Was ist die Adresse von z. B. A[2]?
- A ist die Basisadresse oder Startadresse des Feldes / Arrays.
  - Wenn A ein Array von Bytes ist: A + 2
  - Wenn A ein Array von Wörtern ist: A + 2*4 = A + 8



- I. A.: Adresse von A[i] = &A + i*Elementgröße

Hinweis: &-Zeichen bedeutet in der Programmiersprache C “Adresse von”

Datentransfer-Befehle

- Ladebefehl: load word (copy from memory to register)
  - `lw $t0, <Speicheradresse>`
  - `lw $t0, 12($s1)` # $t0 = Mem[$s1+12]
  - Konstante 12 wird Offset genannt.

- Speicherbefehl: store word (copy from register to memory)
  - `sw $t0, <Speicheradresse>`
  - `sw $t0, 12($s1)` # Mem[$s1+12] = $t0
  - Hinweis: Ziel(adresse) steht am Ende

Beispiel Datentransfer-Befehle

- C/Java: A[12]=A[8]+h;
  - A ein Array von Wörtern
  - Basisadresse A in $s2
  - h liegt in $s3

- MIPS:

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

Word→Inkrement 4*8

Basisadresse + 4*12

Ausrichtung an Wortgrenzen

- In MIPS müssen Wörter bei Adressen beginnen, die ein Vielfaches von 4 sind

- Konzept wird als Ausrichtung an Wortgrenzen (alignment restriction) bezeichnet
  - Was sind die letzten (niederwertigsten) 2 Bits einer Wortadresse?
  - Können sie sich einen Grund für diese Forderung denken?

- Bei Bytes unterscheiden wir 2 Fälle ....

Byte-Reihenfolge /1

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

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

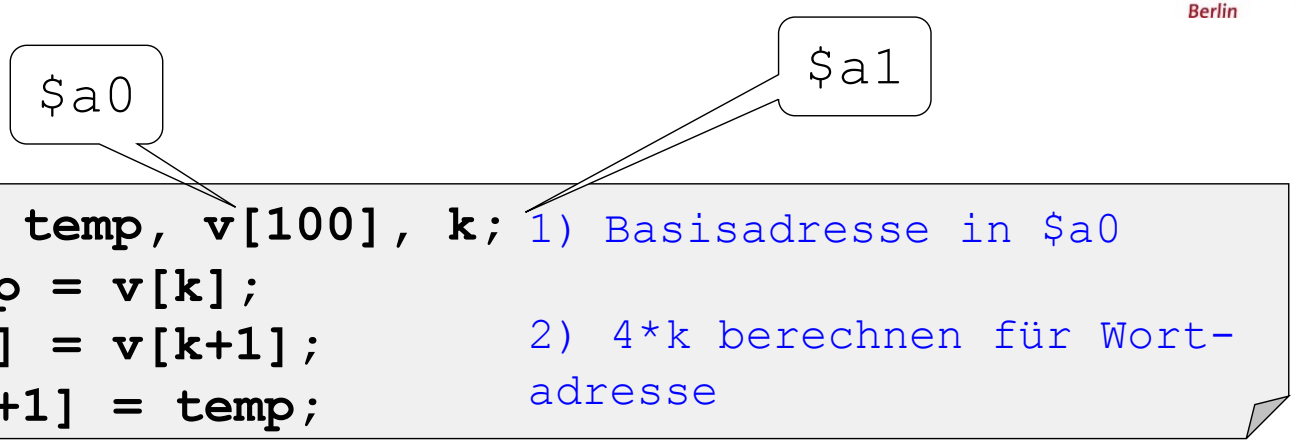
Byte-Reihenfolge /2

- Bei Little-Endian („Klein-Ender“, Kleine Endung) wird das Byte mit den niederwertigsten Bits an der kleinsten Speicheradresse gespeichert (**x86/IA32**)

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

Größeres Beispiel

- C/Java:



- MIPS:

```
add    $t0,$a1,$a1   # $t0 = 2*k
add    $t0,$t0,$t0   # $t0 = 4*k
```

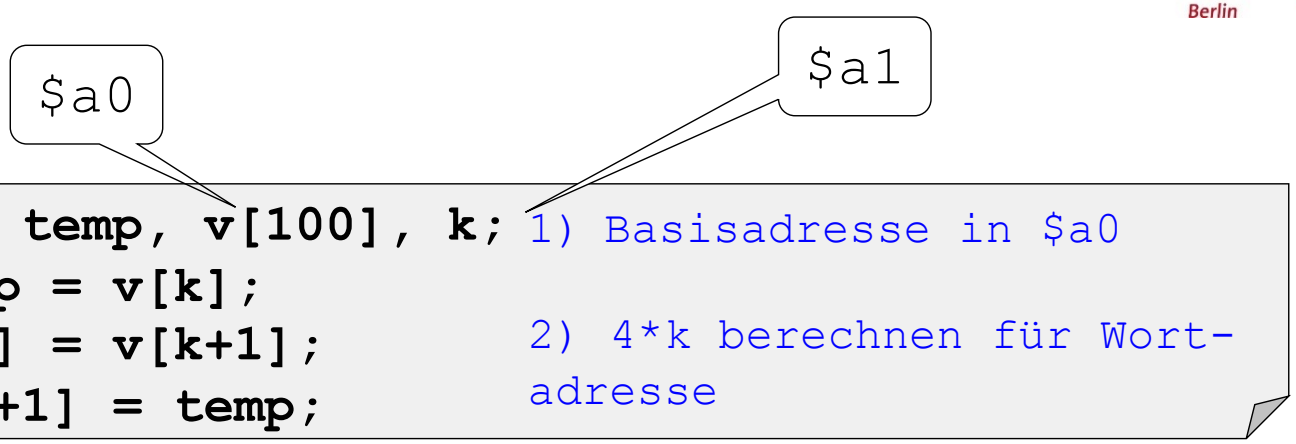
Als erstes die Adresse von v[k] berechnen. Dazu muessen wir k mit 4 multiplizieren. Wir kennen noch keine Multiplikationsinstruktion. Deshalb werden wir k 2 mal mit sich selbst addieren und bekommen somit auch 4*k

- Bald werden wir kürzeren Weg lernen um 4*k zu berechnen.

Hinweis: &v[k] ist (C-)Abkürzung für „Adresse von“ v[k]

Größeres Beispiel

- C/Java:



- 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)    # $t1 = 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] = $t1
```

- Bald werden wir kürzeren Weg lernen um 4*k zu berechnen.

Hinweis: &v[k] ist (C-)Abkürzung für „Adresse von“ v[k]

Konstanten oder Direktoperanden

- In Programmen werden häufig (kleine) Konstanten verwendet.
- Beispiele:

```
a = a+5;
i++;
i<10
```

- Mögliche Lösungen:
  - Konstanten aus Hauptspeicher laden
  - „hardwired“-Register (wie $zero) für Konstanten wie 1 erstellen
  - Versionen der Befehle bereitstellen, bei denen ein Operand eine Konstante ist (MIPS).
  - Spezielle Befehle für Addition kleiner Konstanten (Java bytecode)
- MIPS-Befehl: `add immediate` („addiere direkt“):
  `addi $s1,$s2,4`
- Es gibt keinen `subi` Befehl. Wieso nicht?

3. Entwurfsprinzip

- 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.

Bisher bearbeitete MIPS-Architektur / 1

MIPS-Operanden		
Name	Beispiel	Anmerkungen
32 Register	$s0, $s1, . . . $t0, $t1, . . .	- Speicherort für schnellen Zugriff - Bei MIPS müssen Daten für arithmetische Operation in Registern stehen
2^{30} Speicherwörter	Mem[0], Mem[4], . . ., Mem[2^{32}-4]	- Zugriff bei MIPS durch Datentransport-Befehle - MIPS verwendet Byte-Adressen. - Im Hauptspeicher werden Daten-strukturen, Arrays und ausgelagerte Register gespeichert

Bisher bearbeitete MIPS-Architektur / 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
Datentransport	load word	`lw $s1,100($s2)`	`$s1 = Mem[$s2+100]`	Daten vom Hauptspeicher in ein Register
	store word	`sw $s1,100($s2)`	`Mem[$s2+100] = $s1`	Daten von einem Register in den Hauptspeicher

Maschinensprache / 1

- Wie werden Befehlen im Rechner dargestellt?

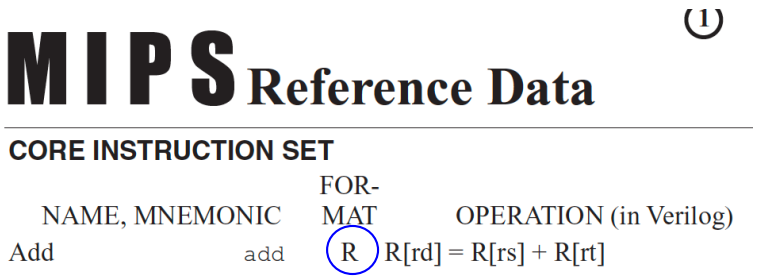
- Befehle sind Bitfolgen
  - Alles in einem digitalen Rechner ist binär

- MIPS-Befehle sehr einfach und regulär
  - alle Befehle sind 32 Bits lang
  - Operanden immer an der gleichen Stelle
  - nur 3 Befehlsformate (instruction formats: R, I, J)

- Register haben Nummer
  - $s0 bis $s7 entsprechen Register 16 bis 23
  - $t0 bis $t7 entsprechen Register 8 bis 15

Maschinensprache / 2

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



MIPS Reference Data

CORE INSTRUCTION SET

NAME, MNEMONIC	FOR-MAT	OPERATION (in Verilog)
Add add	R	R[rd] = R[rs] + R[rt]

	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

- Können Sie erraten, was die Felder bedeuten?
- Merke: In Assemblersprache Zielregister vorne, in Maschinensprache hinten.

Maschinensprache / 3

	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

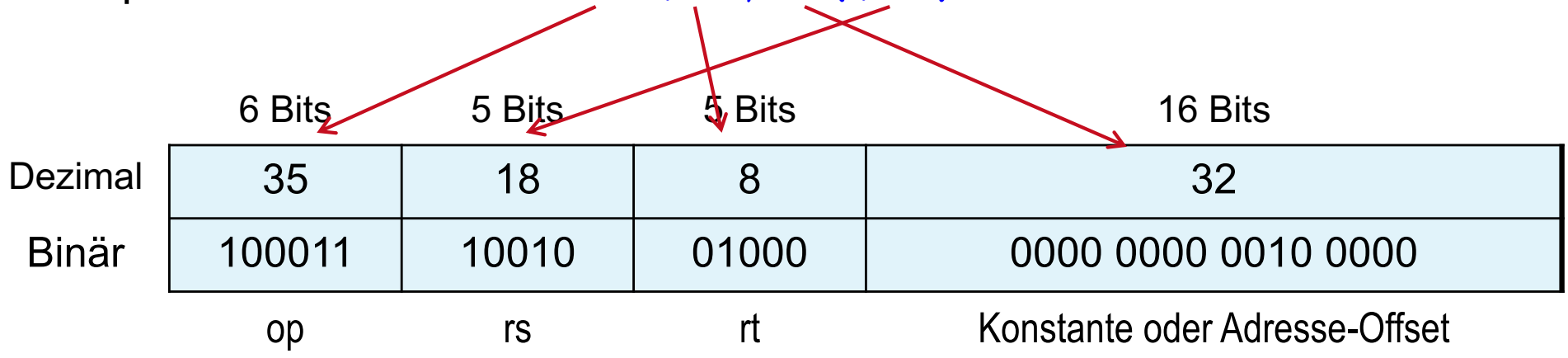
- Felder in diesem MIPS-Befehlsformat (R-Format, R für Register Operationen)
  - *op*: Opcode oder Operationscode; Basisoperation des Befehls
  - *rs*: erstes Quellregister (***s****ource register*)
  - *rt*: zweites Quellregister oder Zielregister (***t****arget register*)
  - *rd*: Zielregister (***d****estination register*)
  - *shamt*: *shift amount*, nur für Schiebe-Befehle (später mehr)
  - *funct*: Funktionscode (*function code*). *op* und *funct* zusammen bestimmen die Operation (meistens)

Maschinensprache / 4

- 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 auf $2^5 = 32$ begrenzt
  - oder verschiedene Befehlslängen?
- Entwurfsprinzip 4: Good design demands compromises (Guter Entwurf erfordert Kompromisse)
- Neues Befehlsformat:
  - I-Typ oder I-Format (I für immediate = direkt)
  - Voriges Format ist R-Typ oder R-Format (R für Register)

Maschinensprache – I-Format

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



- Im lw-Befehl gibt das rt-Feld das Zielregister an (*target register*).
- 16-Bit-Adresse-Offset → beliebiges Wort im Bereich von $-2^{15}..2^{15}-1$ ab Adresse im Basisregister rs kann geladen werden
- Auch Konstanten im `addi`-Befehl sind auf $-2^{15}..2^{15}-1$ beschränkt.

Maschinensprache – Größeres Beispiel

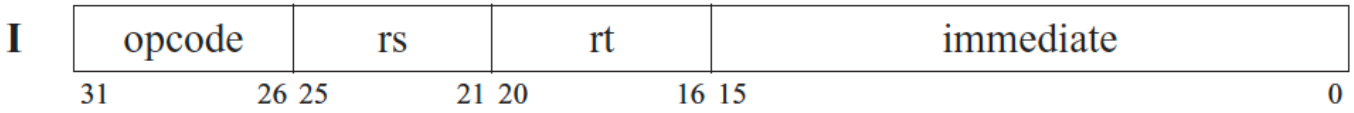
- C/ Java: A[300] = h + A[300];
- MIPS:

```
lw    $t0,1200($t1)    # $t0 = A[300]
add   $t0,$s2,$t0      # $t0 = h+$t0
sw    $t0,1200($t1)    # A[300] = $t0
```

Instr	Opcode	Funct
add	0	32
lw	35	
sw	43	

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

Load Word `lw` I R[rt] = M[R[rs]+SignExtImm]



③

OPCODES, BASE CONVERSION, ASCII SYMBOLS

MIPS opcode (31:26)	(1) MIPS funct (5:0)	(2) MIPS funct (5:0)	Binary	Deci-mal	Hexa-deci-mal	ASCII Char-acter	Deci-mal	Hexa-deci-mal	ASCII Char-acter
(1)	sll	add.*f*	00 0000	0	0	NUL	64	40	@
		sub.*f*	00 0001	1	1	SOH	65	41	A
j	srl	mul.*f*	00 0010	2	2	STX	66	42	B
jal	sra	div.*f*	00 0011	3	3	ETX	67	43	C
lb	add	cvt.s.*f*	10 0000	32	20	Space	96	60	‘
lh	addu	cvt.d.*f*	10 0001	33	21	!	97	61	a
lwl	sub		10 0010	34	22	"	98	62	b
lw	subu		10 0011	35	23	#	99	63	c

Maschinensprache – Größeres Beispiel

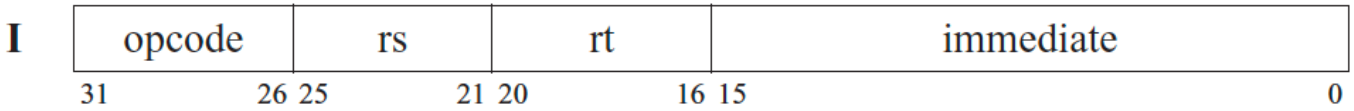
- C/ Java: A[300] = h + A[300];
- MIPS:

```
lw    $t0,1200($t1)     # $t0 = A[300]
add   $t0,$s2,$t0       # $t0 = h+$t0
sw    $t0,1200($t1)     # A[300] = $t0
```

Instr	Opcode	Funct
add	0	32
lw	35	
sw	43	

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

Load Word `lw` I R[rt] = M[R[rs]+SignExtImm]



lb	add	cvt.s.*f*	10 0000	32	20	Space	96	60	‘
lh	addu	cvt.d.*f*	10 0001	33	21	!	97	61	a
lwl	sub		10 0010	34	22	"	98	62	b
lw	subu		10 0011	35	23	#	99	63	c

op	rs	rt	Adresse-Offset		
			rd	shamt	funct
35	9 ($t1)	8 ($t0)	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
```

Instr	Opcode	Funct
add	0	32
lw	35	
sw	43	

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

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

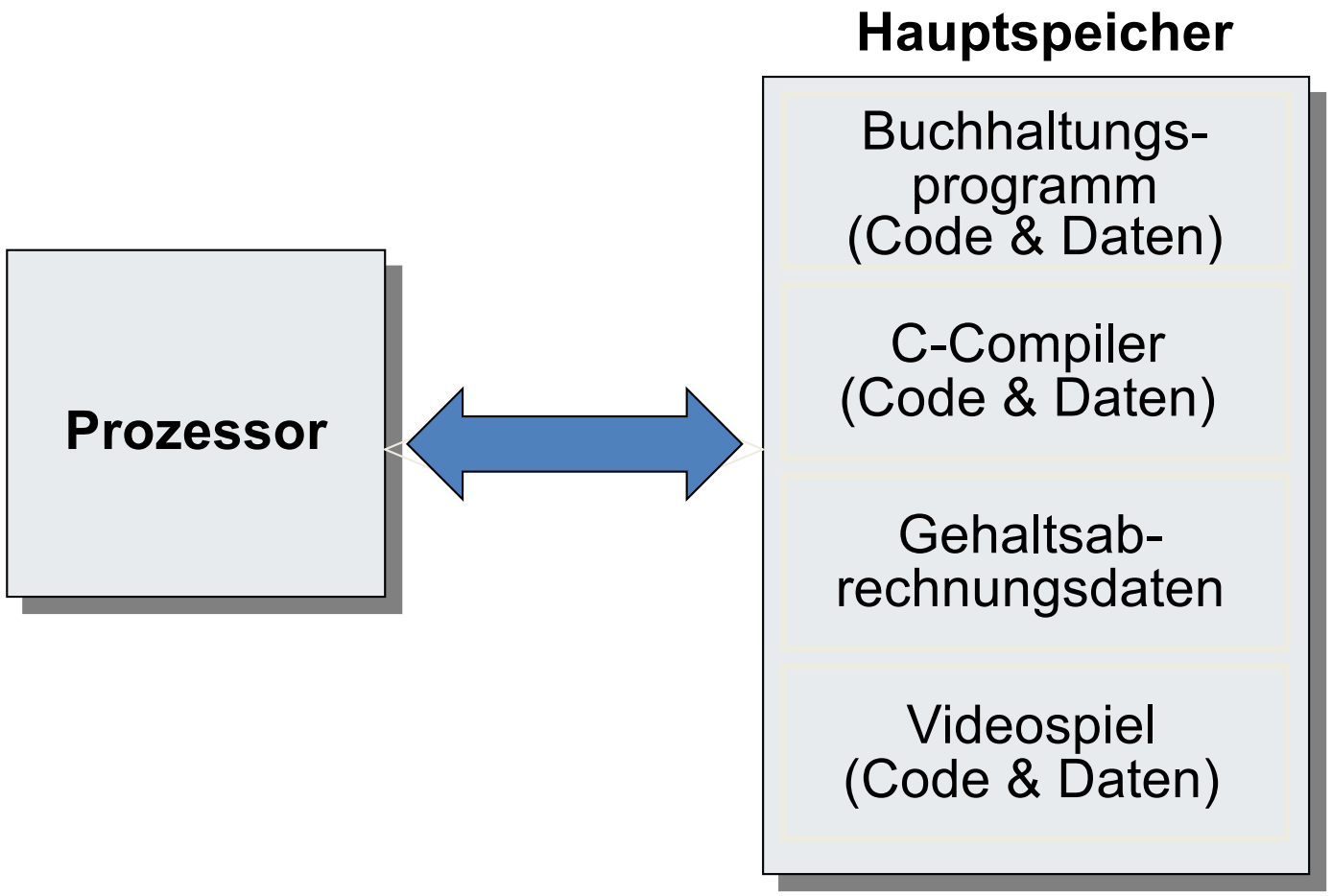
Von-Neumann-Konzept oder Stored-Program Concept

- 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.

John von Neumann

- Erfindung öffnete dem Geist die Flasche
- Rechner ist eine „Metamaschine“
  - Durch Programmwechsel ändert sich die Maschine
  - Die ersten Rechner waren an ein festes Programm gebunden

Von-Neumann-Konzept oder Stored-Program Konzept



Fetch-and-Execute-Zyklus

- 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
  - u. s. w.
- Befehlszähler (program counter, PC) enthält die Adresse des nächsten Befehls.

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

Logische Operationen / 1

- Manchmal notwendig, um auf Bitfelder in einem Wort oder auf einzelne Bits zugreifen zu können.

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

Logische Operationen / 2

- Beispiel: Linksschieben
  - MIPS-Befehl: `sll $t2,$s0,4  # $t2 = $s0 << 4`
  - $s0 = 0000 0000 0000 0000 0000 0000 0000 $1001_B = 9_D$
  - $t2 = 0000 0000 0000 0000 0000 0000 1001 $0000_B = 144_D = 9x16$
- Maschinensprache:

op	rs	rt	rd	shamt	funct
0	0 (NV)	16	10	4	0

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

Logische Operationen / 3

- 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
  - Annahme Inhalt von $s0 ist 153D
  - $s0 = 0000 0000 0000 0000 0000 0000 1001 1001B = 153D
  - 15D = 0000 0000 0000 0000 0000 0000 0000 1111B
- `andi` == AND Immediate Funktion

Logische Operationen / 3

- 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
  - Annahme Inhalt von $s0 ist 153D
  - $s0 = 0000 0000 0000 0000 0000 0000 1001 1001B = 153D
  - 15D = 0000 0000 0000 0000 0000 0000 0000 1111B
- `andi` == AND Immediate Funktion
  - $t2 = 0000 0000 0000 0000 0000 0000 0000 1001B = 9D

Logische Operationen / 4

- Eine ODER-Verknüpfung ergibt eine 1, wenn einer der Operandenbits eine 1 aufweist (Bit „setzen“):
  - MIPS-Befehl: `ori $t2,$s0,15` # $t2 = $s0 ¦ 15
  - $s0 = 0000 0000 0000 0000 0000 0000 1001 1001B = 153D
  - 15D = 0000 0000 0000 0000 0000 0000 0000 1111B
  - $t2 = 0000 0000 0000 0000 0000 0000 1001 1111B = 159D

Logische Operationen / 5

- Letzte logische Basisoperation ist die Negation: NICHT (NOT)
- MIPS hat keine NICHT-Operation.
- Stattdessen NOR (NOT OR)
  - nur 1 wenn beide Operanden 0
- Entspricht NICHT wenn ein Operand 0 ist:
  - A NOR 0 = NOT(A OR 0) = NOT A
  - ¬(A + 0) = (¬A) * (¬0) = ¬A * 1 = ¬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 groß.
  - in der Regel sind Konstanten kurz und passen
- Manchmal 32-Bit Konstanten
  - load upper immediate (`lui`)-Befehl
  - zusammen mit OR immediate `ori` verwenden
- Beispiel: lade $255 \times 2^{16} + 255 = 16.711.935$
- `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

Vorzeichenerweiterung

- MIPS' 16-Bit Immediates werden für die Arithmetik auf 32 Bit erweitert.
- Dazu muss das MSB (sign bit) in die anderen Bitpositionen kopiert werden
- Beispiel (4-Bit → 8-Bit):
  - + 2_D
- 0010 -> 0000 0010

Vorzeichenerweiterung

- MIPS' 16-Bit Immediates werden für die Arithmetik auf 32 Bit erweitert.
- Dazu muss das MSB (sign bit) in die anderen Bitpositionen kopiert werden
- Beispiel (4-Bit → 8-Bit):
  - + 2_D
- 0010 -> 0000 0010

  - - 6_D
- 1010 -> 1111 1010

- = -8 + 2= -6_D = -128+64+32+16+8+2=-6_D

Vorzeichenerweiterung

- MIPS' 16-Bit Immediates werden für die 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 =-128+64+32+16+8+2=-6_D

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

Kontrollbefehle

- Befehle zum Treffen von Entscheidungen:
  - ändern die normale Reihenfolge der Befehle
  - wichtigster Unterschied zwischen Computer und Taschenrechner

- 1.Typ: 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

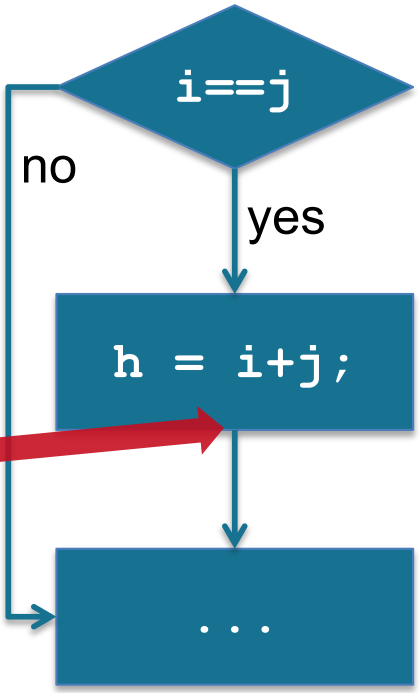
- 2.Typ: Unbedingte Verzweigungen (unconditional branches)
  - jump („Sprung“)
  - `j label` # goto label

Übersetzung einer If-then-Anweisung

- C/Java:

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

$s0 $s1 $s2



- MIPS:

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

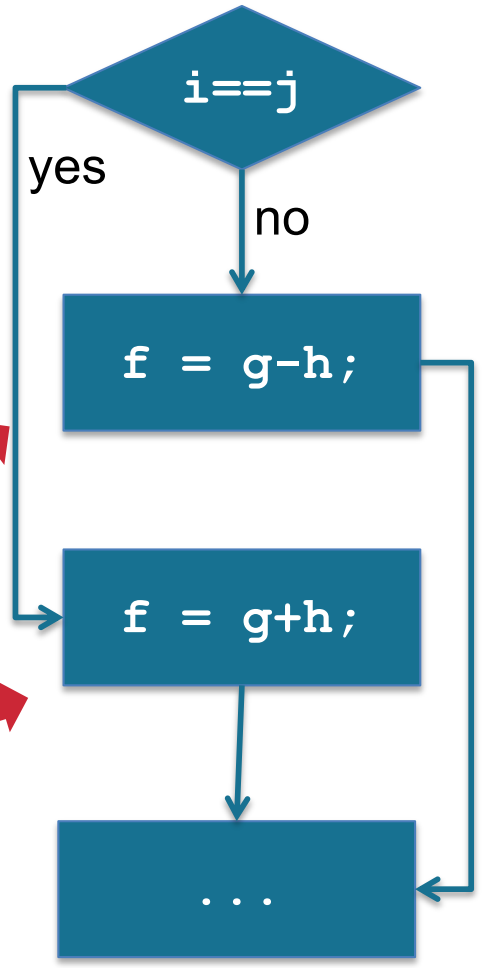
- Ungleichheit effizienter zu überprüfen, d.h. ob gegenteilige Bedingung erfüllt ist
- Marker (label) endif wird vom Assembler übersetzt zu einer Adresse

Übersetzung einer If-then-else-Anweisung

- C/Java:

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

$s0 $s1 $s2 $s3 $s4



- MIPS

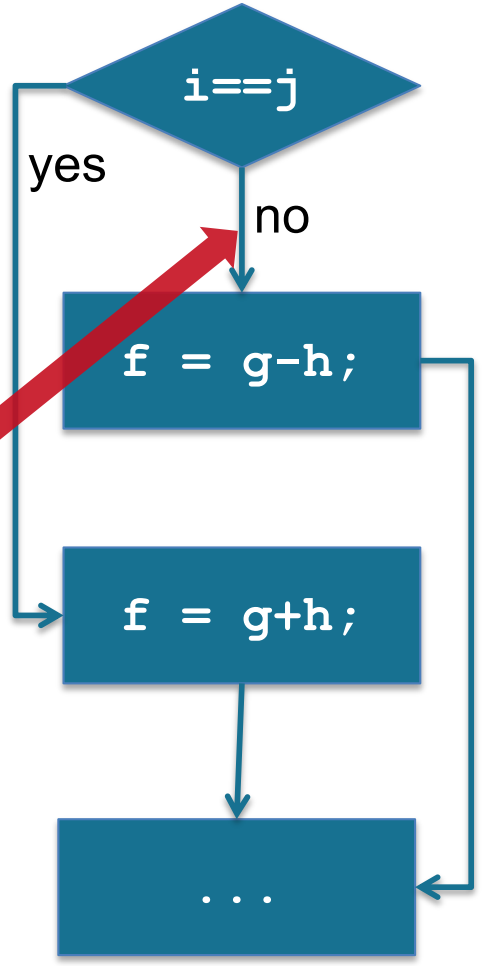
```
  beq   $s0,$s1,if    # if (i==j) goto if
  sub   $s2,$s3,$s4   # f = g-h (else-part)
  j     endif         # goto endif
if:
  add   $s2,$s3,$s4   # f = g+h (if-part)
endif:
    ...
```

Übersetzung einer If-then-else-Anweisung

- C/Java:

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

($s0, $s1, $s2, $s3, $s4)

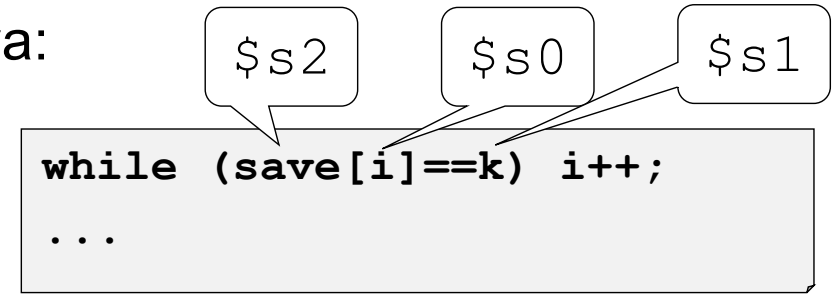


- MIPS

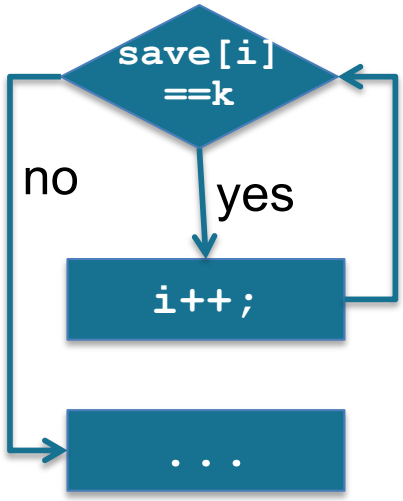
```
  beq   $s0,$s1,if    # if (i==j) goto if
  sub   $s2,$s3,$s4   # f = g-h (else-part)
  j     endif         # goto endif
if:
  add   $s2,$s3,$s4   # f = g+h (if-part)
endif:
    ...
```

Übersetzung einer While-Schleife

- C/Java:



base of array save[ ] is in `$s2`



- MIPS:

```
while:
   sll   $t0,$s0,2          # $t0 = 4*i
   add   $t0,$s2,$t0        # $t0 = &save[i], pointer to 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? / 1

- Wie sieht's mit folgendem Beispiel aus?

  if (a<0) a = -a;

- Verwende „Setze auf 1, wenn kleiner (set on less than)“:

```
slt  $t0,$s0,$s1        # $t0 = ($s0<$s1)
                        # $t0 = 1, wenn $s0<$s1
slti $t0,$s0,10         # $t0 = ($s0<10)
                        # $t0 = 1, wenn $s0<10
```

- MIPS-I Befehlssatz enthielt keinen Branch-on-less-than, da es die Taktzykluszeit verlängern würde (später mehr dazu)
- `slt` und `slti` sind keine Verzweigungen

Kleiner oder größer? / 2

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

Vorzeichbehaftete 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` # $t0 = 1 (**`$s0=`**-1 < **`$s1=`**1 → wahr[true])
  - `sltu $t1,$s0,$s1` # $t1 = 0 (**`$s1=`**$2^{32}$-1 > **`$s1=`**1 → falsch[false])

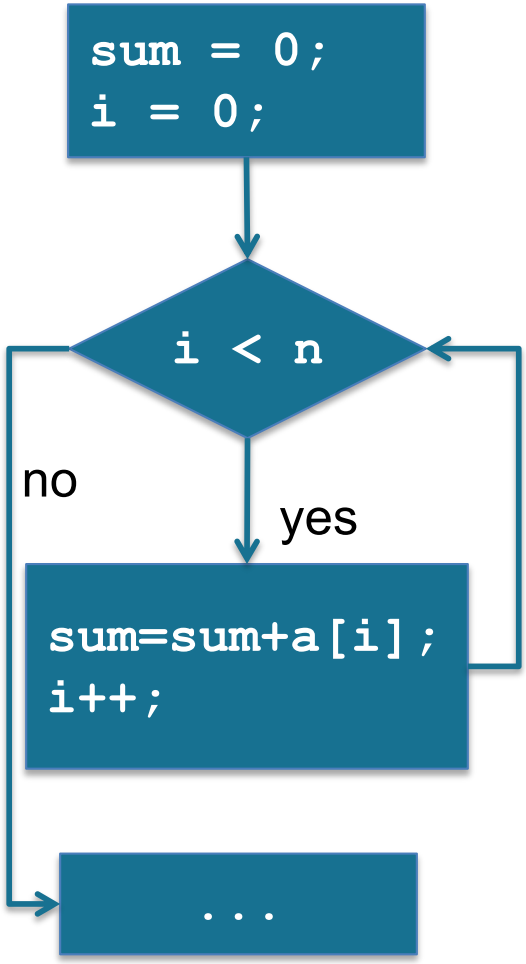
For-Schleifen

- C/Java:

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

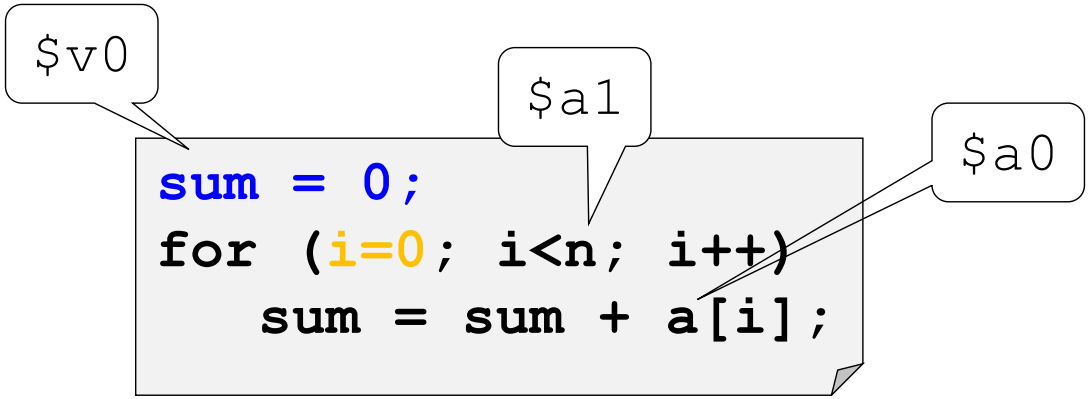
- Ist äquivalent zu:

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



Übersetzung einer For-Schleife

- C/Java: :



- MIPS:

```
      add  $v0,$zero,$zero # sum = 0
      add  $t0,$zero,$zero # i = 0
for:  bge  $t0,$a1,endfor  # if (i>=n) goto endfor
      sll  $t1,$t0,2       # $t1 = 4*i
      add  $t1,$a0,$t1     # $t1 = a+4*i = &a[i]
      lw   $t1,0($t1)      # $t1 = a[i]
      add  $v0,$v0,$t1     # sum = sum+a[i]
      addi $t0,$t0,1       # i++
      j    for             # goto for
endfor: ...
```

Übung

- Übersetzen Sie folgenden Programmabschnitt zu MIPS Assemblersprache
- a (Basisadresse des Arrays von Wörtern) befindet sich in $a0 und n in $a1
- min in $v0 ablegen

```
int i;
int min = a[0];
for (i=1; i<n; i++)
   if (a[i] < min)
      min = a[i];
```

```
# Raum für Lösung
   lw   $v0,0($a0)    # $v0=a(0)
   addi $t0,$zero,1   # i=1
for:
   bge  $t0,$a1,endfor
        # if (i>=n) goto endfor
   sll  $t1,$t0,2     # $t1=4*i
   add  $t1,$a0,$t1   # $t1=&a(i)
   lw   $t1,0($t1)    # $t1=a(i)
   bge  $t1,$v0,endif
        # if (a(i)>=min) goto endif
   add  $v0,$t1,$zero # min=a(i)
endif:
   addi $t0,$t0,1     # i++
   j    for           # goto for
endfor:
```

Adressbildung bei Verzweigungen (1)

③

OPCODES, BASE CONVERSION, ASCII SYMBOLS

MIPS opcode (31:26)	(1) MIPS funct (5:0)	(2) MIPS funct (5:0)	Binary	Deci-mal	Hexa-deci-mal	ASCII Char-acter	Deci-mal	Hexa-deci-mal	ASCII Char-acter
(1)	sll	add.*f*	00 0000	0	0	NUL	64	40	@
		sub.*f*	00 0001	1	1	SOH	65	41	A
j	srl	mul.*f*	00 0010	2	2	STX	66	42	B
jal	sra	div.*f*	00 0011	3	3	ETX	67	43	C
beq	sllv	sqrt.*f*	00 0100	4	4	EOT	68	44	D
bne		abs.*f*	00 0101	5	5	ENQ	69	45	E
blez	srlv	mov.*f*	00 0110	6	6	ACK	70	46	F
bgtz	srav	neg.*f*	00 0111	7	7	BEL	71	47	G

- Befehle:

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

- Format `bne`:

	6 Bits	5 Bits	5 Bits	16 Bits
I-Format	opcode	rs	rt	16-Bit Adress-Offset
I-Format	5	16	17	16-Bit Adress-Offset

Adressbildung bei Verzweigungen (2)

- Befehlszähler (program counter, PC) enthält die Adresse des Befehls, der gerade ausgeführt wird.

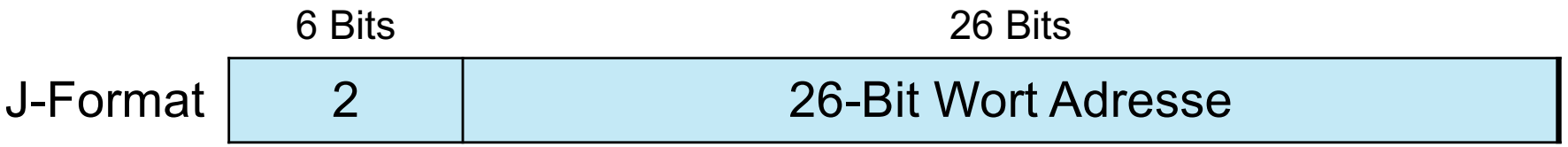
- 16-bit Adress-Offset in Verzweigungen ist relativ zu PC+4
  - Bedingte Verzweigungen springen oft zu nah gelegenen Befehlen.
  - Offset ist außerdem Wortoffset (Zieladresse ist PC + 4 + 4*Offset).

- Befehlszählerrelative Adressierung (PC-relative Adressing)

Adressbildung bei Sprüngen

- Jump Befehl:

```
j label   # goto label
```



- 26-Bit Wortadresse wird erneut mit 4 multipliziert, oberen 4 Bits werden vom PC übernommen
  - Neuer PC = PC_{31-28} # 26-Bit Wort Adresse # 00
- Beispiel:

```
j 1000          # PC31-0 = PC31-28 #4000      (# == Konkatenation)
```

- Adressgrenze von 256 MB == 2^28 (64 Millionen Befehle)

Übung MIPS Maschinensprache

- Übersetze folgenden MIPS-Assemblercode zu MIPS-Maschinencode
  - Nehme an, die Schleife beginnt an Adresse 80000 im Hauptspeicher
  - zeige dezimalen Wert aller 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:
```

MIPS-Maschinensprache

OPCODES, BASE CONVERSION, ASCII SYMBOLS ③

MIPS opcode (31:26)	(1) MIPS funct (5:0)	(2) MIPS funct (5:0)	Binary	Deci-mal	Hexa-deci-mal	ASCII Char-acter	Deci-mal	Hexa-deci-mal	ASCII Char-acter
(1)	sll	add.*f*	00 0000	0	0	NUL	64	40	@
		sub.*f*	00 0001	1	1	SOH	65	41	A
j	srl	mul.*f*	00 0010	2	2	STX	66	42	B
jal	sra	div.*f*	00 0011	3	3	ETX	67	43	C
bne		abs.*f*	00 0101	5	5	ENQ	69	45	E
lb	add	cvt.s.*f*	10 0000	32	20	Space	96	60	‘
lh	addu	cvt.d.*f*	10 0001	33	21	!	97	61	a
lwl	sub		10 0010	34	22	"	98	62	b
lw	subu		10 0011	35	23	#	99	63	c

PC=JumpAddr

PC=PC+4+BranchAddr

- Bestimmung des MIPS Maschinencodes mittels MISP Green-Card
  - Maschinenbefehle verwenden alle 32 bit, aber je nach Befehl eine unterschiedliche Formatierung, entweder das I-Format (z.B. add immediate), J-Format (z.B. jump) oder R-Format (z.B. shift left logic)
  - Jump Befehl: OP code ist 10_B somit eine 2. Der jump Befehl verwendet eine direkte Adressierung, die wir im Folgenden noch berechnen

`j loop`	2	JumpAddr

MIPS-Maschinensprache

OPCODES, BASE CONVERSION, ASCII SYMBOLS ③

MIPS opcode (31:26)	(1) MIPS funct (5:0)	(2) MIPS funct (5:0)	Binary	Deci-mal	Hexa-deci-mal	ASCII Char-acter	Deci-mal	Hexa-deci-mal	ASCII Char-acter
(1)	sll	add.*f*	00 0000	0	0	NUL	64	40	@
		sub.*f*	00 0001	1	1	SOH	65	41	A
j	srl	mul.*f*	00 0010	2	2	STX	66	42	B
jal	sra	div.*f*	00 0011	3	3	ETX	67	43	C
bne		abs.*f*	00 0101	5	5	ENQ	69	45	E
lb	add	cvt.s.*f*	10 0000	32	20	Space	96	60	‘
lh	addu	cvt.d.*f*	10 0001	33	21	!	97	61	a
lwl	sub		10 0010	34	22	"	98	62	b
lw	subu		10 0011	35	23	#	99	63	c

PC=JumpAddr

PC=PC+4+BranchAddr

- Bestimmung des MIPS Maschinencodes mittels MISP Green-Card
  - Der Opcode des branch not equal entspricht 101_B also einer 5_D. Die Verzeigungsadresse ist eine relative Adresse zum PC und muss in diesem Fall auf die Exit-Adresse zeigen

`bne $t0,$s5,exit`	5	rs	rt	BranchAddr

  - Der load word Befehl, hat einen Opcode von $10\ 00\ 11_B \rightarrow 35_D$ Adressoffset ist 0, wie wir sofort aus dem MIPS Befehl erkennen

`lw $t0,0($t1)`	35	rs	rt	offs

MIPS-Maschinensprache

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

	`loop: sll $t1,$s3,2`	0	0	rt	rd	shamt	0	R
	`add $t1,$t1,$s6`	0	rs	rt	rd	0	32	R
➡	`lw $t0,0($t1)`	35	rs	rt	offs			I
➡	`bne $t0,$s5,exit`	5	rs	rt	BranchAddr			I
	`addi $s3,$s3,1`	8	rs	rt	imm			I
➡	`j loop`	2	JumpAddr					J
	`exit:`							

MIPS-Register

Name	Number	Verwendung
`$zero`	0	Der Wert 0
`$v0-$v1`	2-3	$1 wird vom Assembler verwendet (nicht im Programm verwenden) (Werte) aus algebra. Ausdrücken und Funktionsergebnissen
`$a0-$a3`	4-7	(Argumente) erste 4 Parameter für das Unterprogramm
`$t0-$t7`	8-15	Temporäre Variablen (nicht geschützt)
`$s0-$s7`	16-23	Gespeicherte Werte (endgültige berechnete Ergebnisse - geschützt)
`$t8-$t9`	24-25	Temporäre Variablen (nicht geschützt)
`$gp`	28	$26-27 reserviert für das Betriebssystem Global pointer
`$sp`	29	Stack pointer
`$fp`	30	Frame pointer
`$ra`	31	Return address

Lösung /1

- **Hauptspeicheradressen**: Schleife beginnt bei Adresse 80000
  - Erhöht sich jeweils um 4, d.h. der neue PC-Wert ist PC+4

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

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

Lösung /2

Shift Left Logical	sll	R	R[rd] = R[rt] << shamt

```
loop: sll   $t1,$s3,2
```

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

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

Lösung /3

- $t1 == reg. 9
- $s6 == reg. 22

```
add   $t1,$t1,$s6
```

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

`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	offs			80008
`bne $t0,$s5,exit`	5	rs	rt	BranchAddr			80012
`addi $s3,$s3,1`	8	rs	rt	imm			80016
`j loop`	2	JumpAddr					80020
`exit:`							80024

Lösung /4

- $t0 == reg. 8
- $t1 == reg. 9

lb	add	cvt.s.*f*	10 0000	32	20	Space	96	60	‘
lh	addu	cvt.d.*f*	10 0001	33	21	!	97	61	a
lwl	sub		10 0010	34	22	"	98	62	b
lw	subu		10 0011	35	23	#	99	63	c

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

`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	BranchAddr			80012
`addi $s3,$s3,1`	8	rs	rt	imm			80016
`j loop`	2	JumpAddr					80020
`exit:`							80024

Lösung /5

- $t0 == reg. 8
- $s5 == reg. 21

| bne | abs.*f* | 00 0101 | 5 | 5 | ENQ | 69 | 45 | E |

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

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	BranchAddr				80012
addi $s3,$s3,1	8	rs	rt	imm				80016
j loop	2	JumpAddr						80020
exit:								80024

Lösung /6

- bne: relative Zieladressierung PC + 4 + 4 * BranchAddr

  → 4 * BranchAddr = 80024 – 80016 = 8 (nächster MIPS Befehl)

  → BranchAddr = 8/4 = 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

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

Lösung /7

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

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	JumpAddr					80020
exit:							80024

Lösung /8

- Pseudo-direkte PC Adressierung: Multiplikation mit 4
  - → Jump Adresse JumpAddr = 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

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

Multiplikation in MIPS

- 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 $s2,$s3** # Hi#Lo = $s2x$s3
    - Move from lo: **mflo $s1** # $s1 = Lo
    - Move from hi: **mfhi $s1** # $s1 = Hi

- Pseudo-Instruktion: `mul $s1,$s2,$s3`

- Reale Umsetzung: `mult $s2,$s3`
- `mflo $s1`

Division in MIPS

- Division benutzt auch die Register Hi und Lo
  - Quotient befindet sich im Lo, Rest im Hi
  - Rest wird als Nebeneffekt der Division mitproduziert
- MIPS-Befehl:
  - `div $s2,$s3` # Lo = $s2/$s3, Hi = $s2%$s3
- Pseudo-Befehle:
  - `div $s1,$s2,$s3`
  - `rem $s1,$s2,$s3`

Überlauf

- Was passiert wenn Überlauf auftritt?

- Hängt ab vom Software-System / von der Programmiersprache
  - C/Java ignorieren Überlauf
  - Fortran nicht

- Nicht immer soll ein Überlauf angezeigt werden.
  - neue MIPS-Befehle: **`addu, addiu, subu, sltu, sltiu, mulu, divu`**
  - **`addiu`** und **`sltiu`** erweitern das Vorzeichen ihrer 16-Bit Immediates!

Überlaufabhandlung

- Wie werden Überläufe abgehandelt?
  - Eine Exception (Interrupt) wird ausgelöst
  - Sprung zu einer vordefinierten Adresse, um die Ausnahme (Exception) zu behandeln
    - Interrupt Handler (Teil des Betriebssystems (Operating System [OS]))
    - Register $k0 (26) und $k1 (27) sind fürs OS reserviert.
  - Unterbrochene Adresse wurde für einen möglichen Rücksprung im exception program counter ($epc) gespeichert.
    - Neuer MIPS-Befehl: move from coprocessor 0
    - `mfc0 rt, rd` → `mfc0 $k0, $epc`

Fazit /1

- Befehle sind Zahlen.
  - Assemblersprache bieten „komfortable“ symbolische Darstellungen.
  - Maschinensprache ist jedoch die Wirklichkeit.
- Assembler kann „Pseudobefehle“ anbieten.
  - z. B. move $t0,$t1 gibt‘s nur in Assemblersprache und wird übersetzt zu add $t0,$t1,$zero.

Fazit /2

- MIPS:
  - einfache Befehle (alle 32 Bits lang)
  - sehr strukturiert, keine unnötige Bagage
  - nur 3 Befehlsformate:

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

Prozeduren und Funktionen / 1

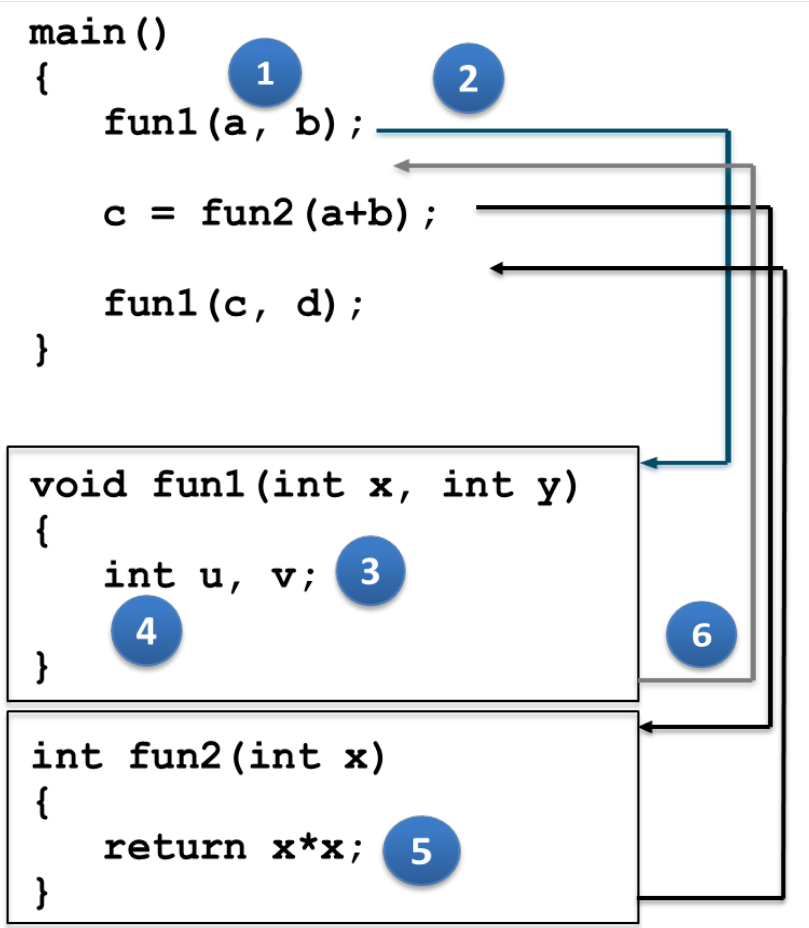
- Äußerst wichtig in höheren Programmiersprachen
  - Strukturierung des Programms
  - Abstraktion!

- In diesem Abschnitt lernen wir:
  - „Blatt“-Funktionen zu übersetzen.
  - Nicht-Blatt-Funktionen zu übersetzen.
  - MIPS Registerkonventionen zu benutzen.
  - Rekursive Funktionen zu übersetzen.

Prozeduren und Funktionen / 2

- 6 Schritte beim Ausführen einer Prozedur:
  1. Parameter werden an einer Stelle abgelegt, auf die die aufgerufene Prozedur zugreifen kann
  2. Programmsteuerung wird an die Prozedur übergeben
  3. Die für die Prozedur benötigten Speicherressourcen bereitstellen
  4. Prozedur führt Aufgabe aus
  5. Ergebnis wird an einer Stelle abgelegt, worauf die aufrufende Prozedur zugreifen kann
  6. Rücksprung an die Stelle, an der die Prozedur aufgerufen wurde

Caller: aufrufende Prozedur
Callee: aufgerufene Prozedur



Prozeduren und Funktionen / 3

- Registerkonvention für Prozeduraufrufe:
  - $a0-$a3: 4 Argumentregister
  - $v0-$v1: 2 Register für Rückgabewerte
  - $ra: Rücksprungadresse (return address)
- Neuer Befehl: Jump-and-Link (`jal`)

```
jal FunAddress    # speichere Rücksprungadresse
                  # in $ra und springe zur FunAddress
```

  - Befehlszeiger (PC) enthält Adresse des aktuellen Befehls
    Was ist also die Rücksprungadresse?

- Neuer Befehl: Jump-Register (jr)

```
jr $ra            # PC ← $ra
```

  - Die `jr`-Anweisung **gibt die Kontrolle an den Aufrufer zurück.** Es kopiert den Inhalt von $ra in den PC

Beispiel Parameterübergabe und Funktionsaufruf

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:

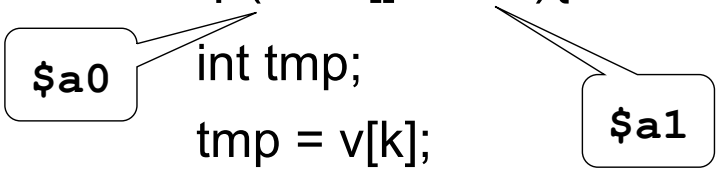
```
main:
     move $a0,a
     move $a1,b
     jal  fun1

     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
```

`jal` # Rücksprungadresse in $ra, jump to fun1

`jr` # PC ← $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;      }
```

- MIPS:

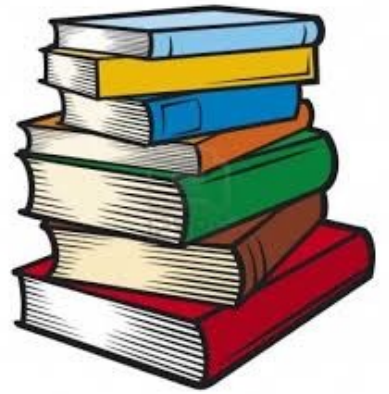
```
swap: sll $t0,$a1,2    # $t0 = 4*k
      add $t0,$a0,$t0  # $t0 = &v[k]
      lw  $t1,0($t0)   # temp = $t1 = v[k]
      lw  $t2,4($t0)   # $t2 = v[k+1]
      sw  $t2,0($t0)   # v[k] = v[k+1]
      sw  $t1,4($t0)   # v[k+1] = temp
      jr  $ra          # Rücksprung (return)
```

- **Caller:**

```
jal swap
```

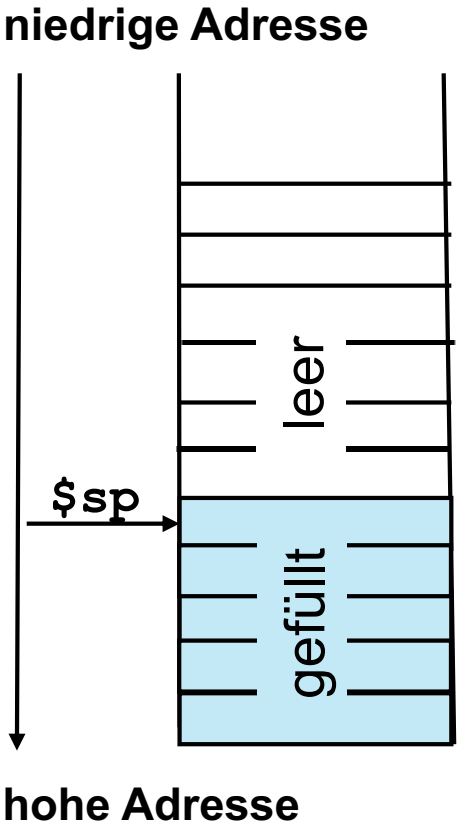
Keller (Stack)

- Probleme, Probleme, Probleme, ...
  - Was, wenn eine Funktion eine andere aufruft? ($ra?)
  - Was, wenn eine Funktion mehr als 4 Parameter hat?
  - Was, wenn . . . ?
- Wichtige Datenstruktur:
  - Keller (stack (=Stapel)): last-in-first-out (LIFO)
- 2 Basisoperationen:
  - push: etwas auf den Keller/Stapel ablegen
  - pop: etwas vom Keller/Stapel entfernen



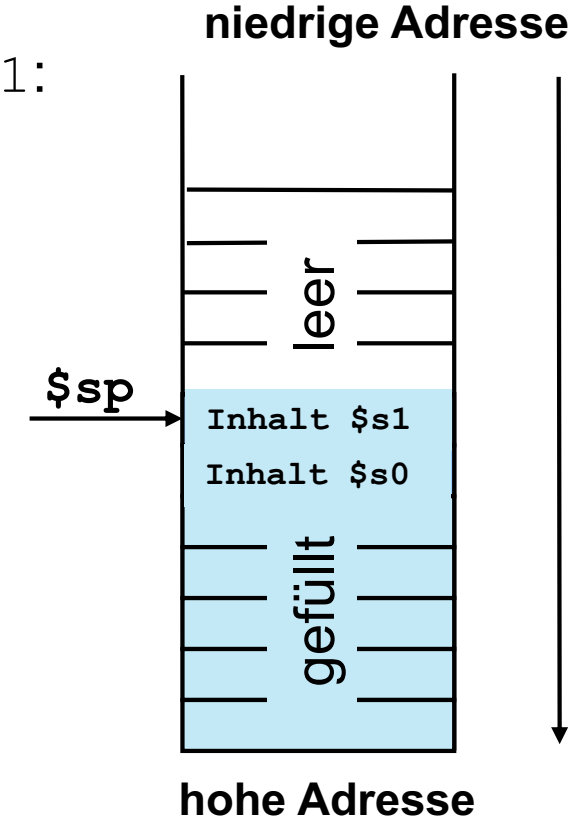
- In MIPS:
  - **Keller/Stack** wächst von höherwertigen zu niederwertigen Adressen
  - **Kellerzeiger** (**stack pointer**) ($sp) zeigt auf das „oberste" Element des Kellers

Register auf dem Stack ablegen



Ablegen von `$s0 und $s1`:

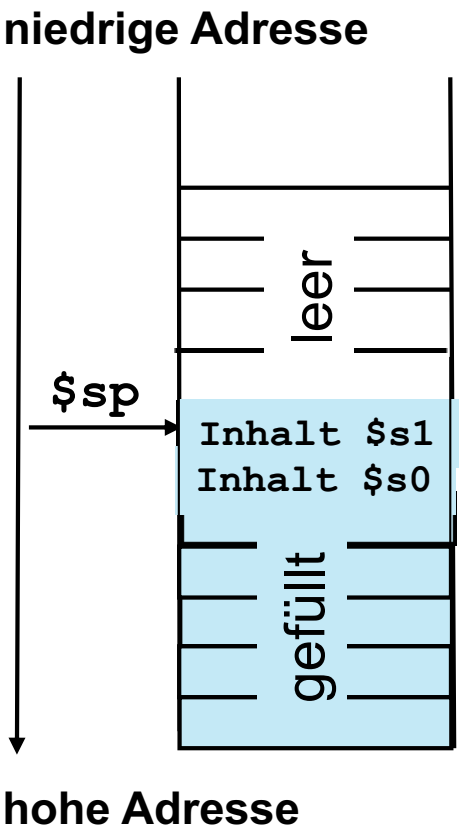
```
addi $sp,$sp,-8
sw   $s0,4($sp)
sw   $s1,0($sp)
```

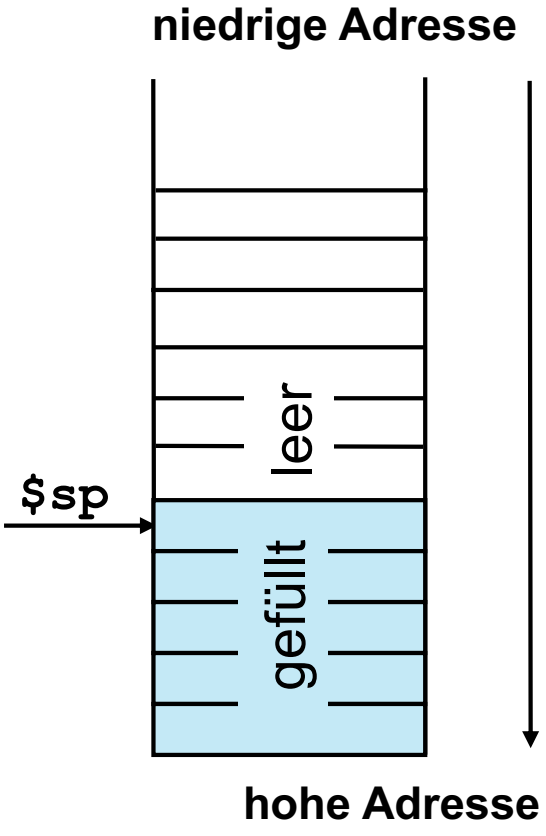


Register von Stack entfernen

Entfernen von `$s0 und $s1`:

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





MIPS-Registerkonventionen / 1

Name	Register-nummer	Verwendung	Bei Aufruf beibehalten?
$zero	0	Kontante 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 (*Global pointer*)	ja
$sp	29	Kellerzeiger (*Stack pointer*)	ja
$fp	30	Rahmenzeiger (*Frame pointer*)	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 von der Callee bei Verwendung gerettet werden müssen
  - „Vertrag“ d.h. Konvention zwischen Caller und Callee

- Regeln, bei Übersetzung einer nicht-Blatt-Funktion:
  1. sichere $ra auf dem Stack
  2. weise Variablen, die nach einem Aufruf benötigt werden, an einen $si Register zu und **sichere zuvor** $si auf dem Stack
  3. weise Variablen, die nach einem Aufruf nicht länger benötigt werden, an einen $ti Register zu
  4. kopiere Argumente ($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; }
- MIPS:

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

```
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
                           # Rücksprungadresse
      lw   $s0,0($sp)      # wiederherstellen von $s0
      addi $sp,$sp,8       # wiederherstellen des $sp
      jr   $ra             # Rücksprung
```

Rekursive Funktionen

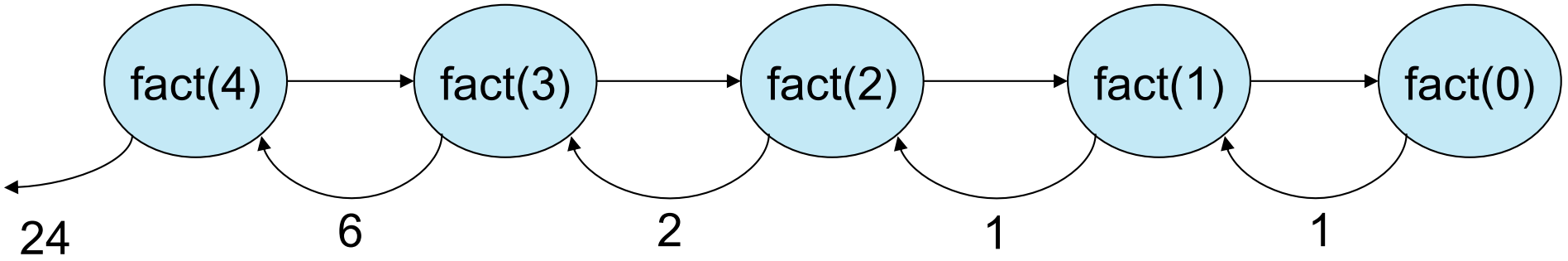
- Rekursive Funktion zur Berechnung der Fakultät:

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

- Mathematisch:
  - n! = n*(n-1)! für n > 1
  - 0! = 1

Aufrufkette

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



Assemblercode für fact

- Vor dem rekursiven Aufruf, sichert man die Rücksprungadresse ($ra) und das Argument n ($a0) auf dem Stack:

```
addi $sp,$sp,-8      # Stack-Reservierung für 2 Variablen
sw   $ra,4($sp)      # speichere $ra auf Stack
sw   $a0,0($sp)      # speichere $a0 auf Stack
```

- Nach dem rekursiven Aufruf, stellt man n und $ra wieder her:

```
lw   $ra,4($sp)      # stelle $ra wieder her
lw   $a0,0($sp)      # stelle $a0 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);
}
```

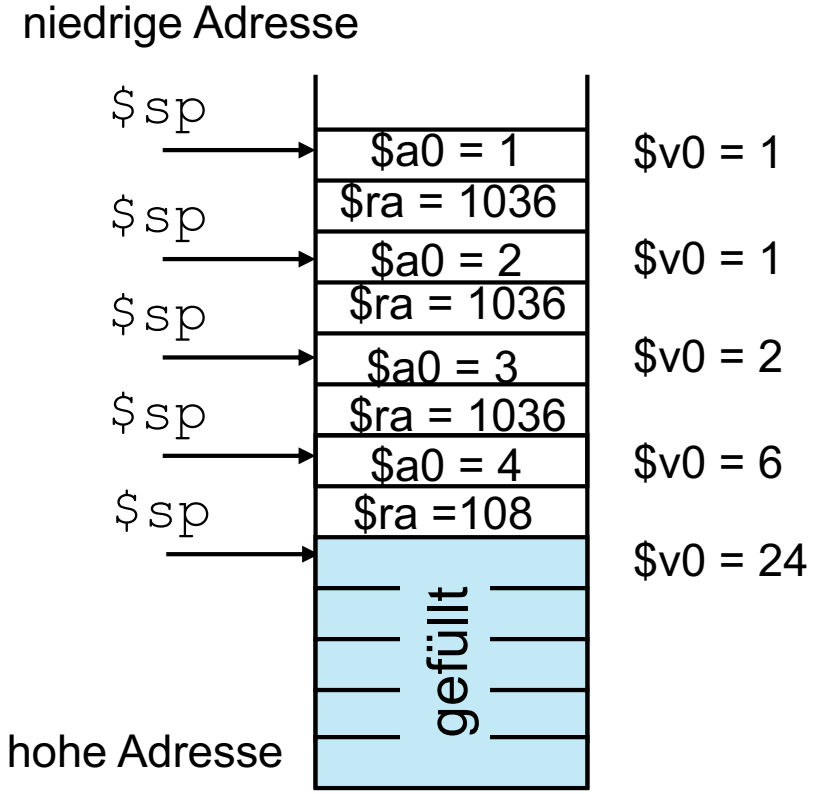
Befehlsadresse (Instruction address)

```
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
1020          sw   $ra,4($sp)    # speichere $ra auf Stack
1024          sw   $a0,0($sp)    # speichere $a0 auf Stack
1028          addi $a0,$a0,-1   # $a0 = n-1
1032          jal  fact          # $v0 = fact(n-1)
1036          lw   $a0,0($sp)   # wiederherstellen ($a0)
1040          lw   $ra,4($sp)   # wiederherstellen ($ra)
1044          addi $sp,$sp,8    # $sp += 8
1048          mul  $v0,$a0,$v0  # $v0 = n*fact(n-1)
1056          jr   $ra          # Rücksprung
```

Wie es funktioniert?!

Caller

```
100 addi $a0,$zero,4
104 jal fact
108 ...
```




```
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   $a0,0($sp)
1028           addi $a0,$a0,-1
1032           jal  fact
1036           lw   $a0,0($sp)
1040           lw   $ra,4($sp)
1044           addi $sp,$sp,8
1048           mul  $v0,$a0,$v0
1056           jr   $ra
```

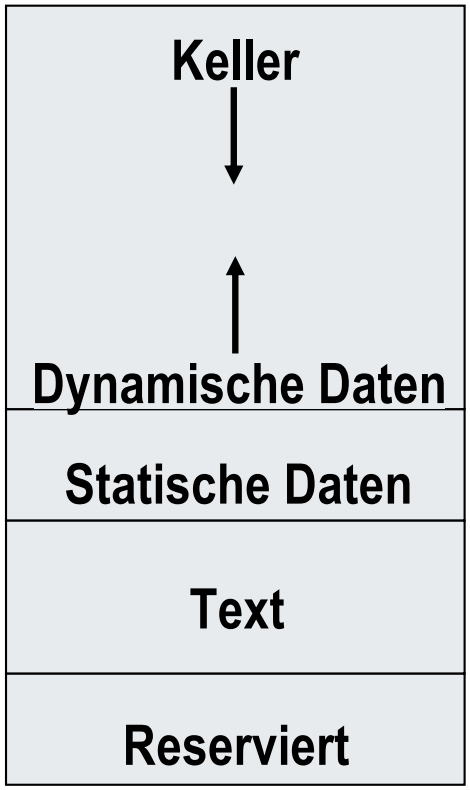
MIPS-Speicheraufteilung

- Kellerzeiger wird mit 7fff ffff$_H$ initialisiert und wächst nach unten
- Bereich für dynamischen Daten (Halde = heap) wächst nach oben
  - C: malloc, Java: new
  - Keller und Halde wachsen in gegengesetzte Richtung
- Statische Daten (z. B. globale Variablen) unter Halde und $gp zeigt etwa in die Mitte
- Programmcode (Text) beginnt bei 0040 0000^H
- Unterer Bereich ist reserviert (für das Betriebssystem)

`$sp -> 7fff fffc`$_H$

`$gp -> 1000 8000`$_H$
`1000 0000`$_H$

`pc -> 0040 0000`$_H$



Zeichen und Zeichenfolgen

- ASCII (American Standard Code for Information Interchange) ist eine Standard für Zeichendarstellung.
  - 8-Bit-Bytes, 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

- ASCII-Wert(a) – ASCII-Wert(A) =
  ASCII-Wert(b) – ASCII-Wert(B) = . . . =
  ASCII-Wert(z) – ASCII-Wert(Z) = 32
- Java verwendet (auch) Unicode
  - 16-Bit
  - z. Z. > 107,000 Zeichen

Befehle zum Transport von Bytes Zeichenfolgen

„Abba“ =	65	98	98	97	0

- *load-byte* lädt ein Byte aus Hauptspeicher in Register
- *store-byte* nimmt rechtsbündigen 8 Bits eines Registers und schreibt in Hauptspeicher
  - `lb $t0,0($gp) # $t0 =8 Mem[$gp]`
  - `sb $t0,0($gp) # Mem[$gp] =8 $t0`
- In C wird Zeichenfolge mit Byte 0 abgeschlossen

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

Kopieren einer Zeichenfolge

- C: void strcpy(char d[], char s[]){

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

- MIPS:

- strcpy: add $t0,$zero,$zero # i=0
- strcpy_while: add $t1,$a1,$t0 # $t1 = &s[i]

```
lb   $t1,0($t1)       # $t1 = s[i]
add  $t2,$a0,$t0      # $t2 = &d[i]
sb   $t1,0($t2)       # d[i] = s[i]
beq  $t1,$zero,strcpy_endwhile # if(s[i]==0)goto …
addi $t0,$t0,1        # i++
j    strcpy_while     # goto strcpy_while
```

```
strcpy_endwhile:  jr  $ra          # Rücksprung (return)
```

Vorzeichenbehaftete und vorzeichenlose Zahlen

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

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

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

MIPS-Adressierungsarten

- Verschiedene Formen der Adressberechnung werden als Adressierungsarten (addressing modes) bezeichnet.

MIPS-Adressierungsarten:

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 Befehlszähler und Konstante im Befehl.)
   - beq $t0,$a1,100 # PC = PC+4+4x100
5. Pseudo-direkte Adressierung ([Sprung]-Adresse Konkatenation 26 Bits im Befehl mit oberen Bits PC)
   - j 1000 # PC = $PC_{31..28}$#(4x1000)

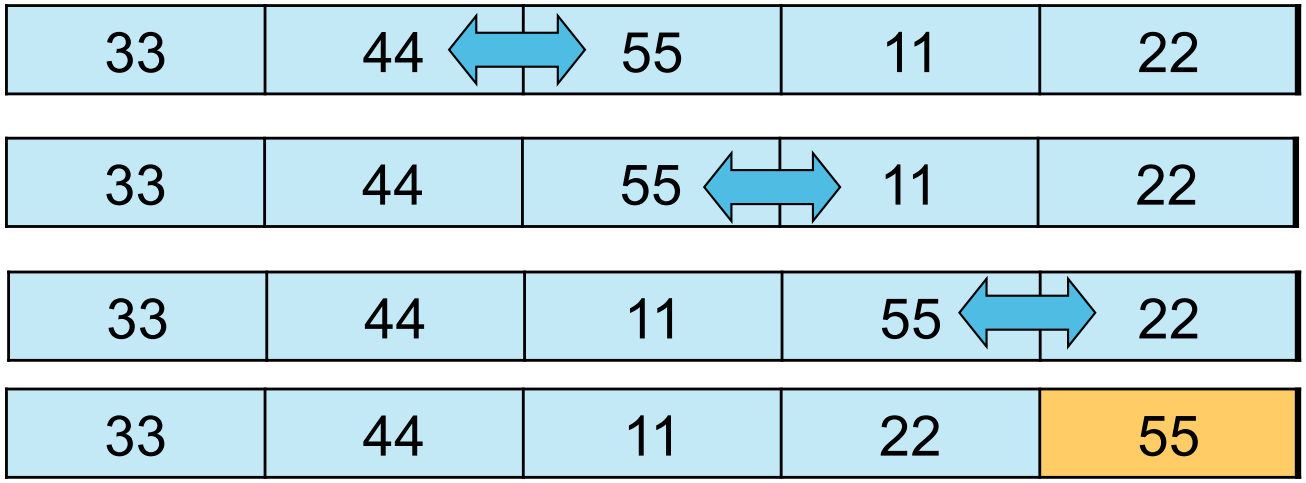
Gesamtes Programm in Assemblersprache

- Bubblesort: sortieren durch Aufsteigen
- „Benötigt“ Prozedur swap
- Bei manuellen Übersetzen von C in Assemblerspache wie folgt vorgehen
  - Register an Programmvariablen zuteilen
  - Code für den Rumpf der Prozedur generieren
  - Register über Prozeduraufruf hinweg beibehalten

Gesamtes Programm in Assemblersprache „Bubble Sort“ / 1

0	1	2	3	4
44	33	55	11	22

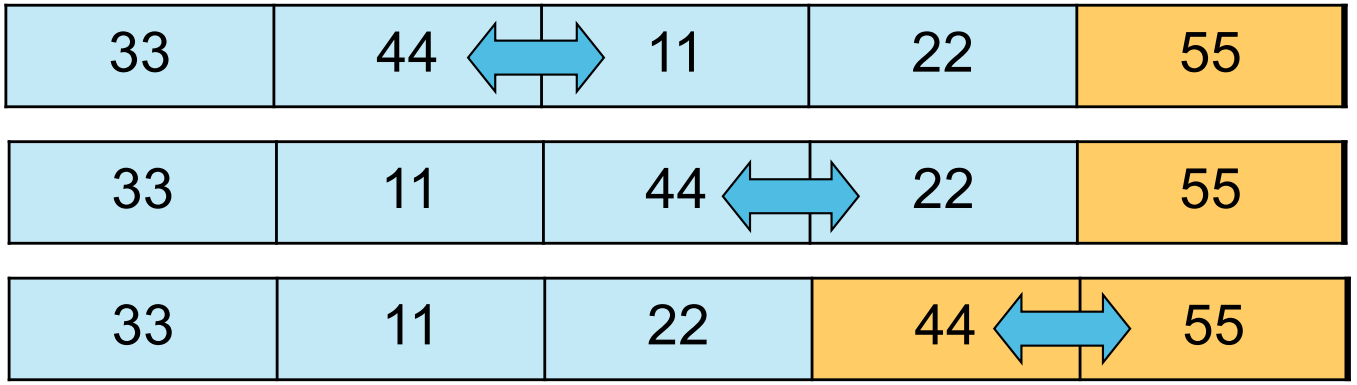
Bubblesort, äußere Schleife 1



Gesamtes Programm in Assemblersprache „Bubble Sort“ / 2

0	1	2	3	4
33	44	11	22	55

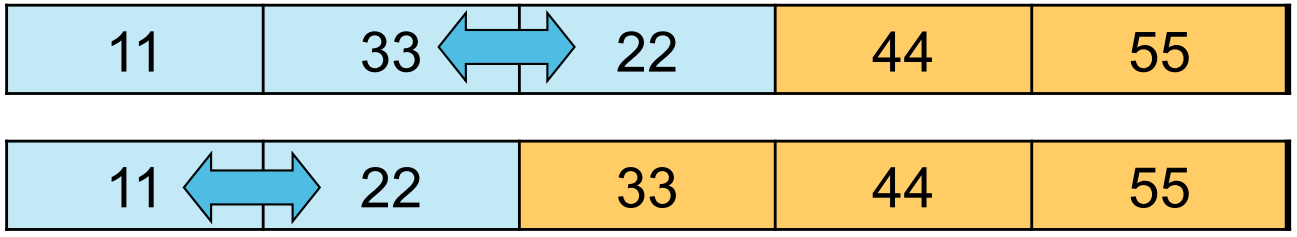
Bubblesort, äußere Schleife 2



Gesamtes Programm in Assemblersprache „Bubble Sort“ / 3

0	1	2	3	4
11	33	22	44	55

Bubblesort, äußere Schleife 3



Gesamtes Programm in Assemblersprache „Bubble Sort“ / 4

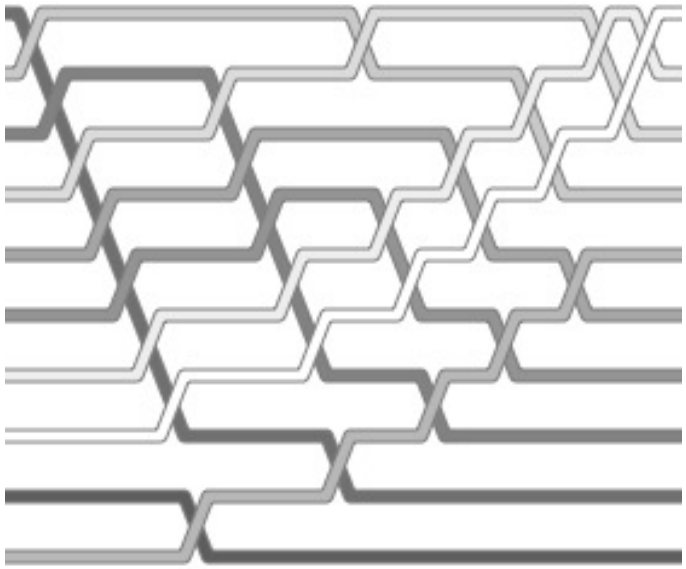
0	1	2	3	4
11	22	33	44	55

Bubblesort äußerere Schleife /4

11	22	33	44	55

Bubblesort – 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;
}
```



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 $ra auf Keller
  - verwende für Variablen, die nach Aufruf benötigt werden, saved ($si)-Register
    - i, j, v[] ($a0), n-1, n-i-1
    - sichere saved-Register auf Keller und stelle sie 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);
}
```

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

Assemblercode für sort – Register retten

```
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);
}
```

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

```
sort:
        addi    $sp,$sp,-24     # Kellerspeicher-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
```

Assemblercode für sort–Prozedurrumpf / 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);
```

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

```
        move    $s0,$zero       # i = 0
        move    $s4,$a0         # $s4 = v[] (rette $a0)
        addi    $s1,$a1,-1      # $s1 = n-1
for1:   bge     $s0,$s1,endfor1 # if (i>=n-1) goto endfor1
        move    $s2,$zero       # j = 0
        sub     $s3,$s1,$s0     # $s3 = n-1-i
for2:   bge     $s2,$s3,endfor2 # if (j>=n-i-1) goto endfor2
        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 nächste Folie
```

Assemblercode für sort–Prozedurrumpf / 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);
}
```

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

```
        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
```

Assemblercode für sort–Register wiederherstellen

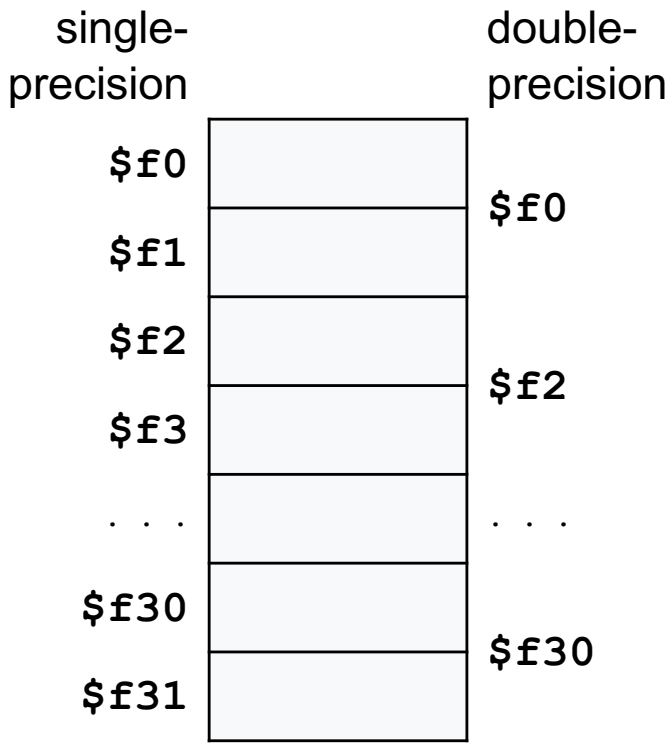
```
void sort(int v[], int n){
   int i, j;
   for (i=0; i < n-1; i++)
      for (j=0; j < n-i-1; j++)
         if (v[j] > v[j+1])
            swap(v, j);
}
```

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

```
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)
```

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

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

MIPS Floating Point Sprungbefehle / 1

- Vergleich-Befehle 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 0.5n < 1.0·10-9

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

MIPS Floating Point Sprungbefehle / 2

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

```
   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
   bc1f endwhile                 # if (!cc) goto enwhile
   mul.s        $f0,$f0,$f2      # exp = exp*0.5
   addi $t0,$t0,1                # n++
   j            while            # goto while
endwhile:   ...
```

Zusammenfassung / 1 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 Schiebebefehlen
- I-Format für Datentransfer, Immediate, Verzweigungen
- J-Format für Sprünge

Zusammenfassung / 2 MIPS-Register

Name	Register-nummer	Verwendung	Bei Aufruf beibehalten?
`$zero`	0	Kontante 0	-
`$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 / 3 MIPS-Assemblersprache / 1

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)	$s1=Mem[$s2+100]	Wort vom Hauptspeicher in Register
	store word	sw $s1,100($s2)	Mem[$s2+100] = $s1	Wort von Register in Hauptspeicher
	load byte	lb $s1,100($s2)	$s1=Mem[$s2+100]	Byte vom Hauptspeicher in Register
	store byte	sb $s1,100($s2)	Mem[$s2+100] = $s1	Byte von Register in Hauptspeicher
	load upper imm.	lui $s0,100	$s1 = 100x$2^{16}$	Konstante in obere 16 Bit

rot = richtige Kategorie?

Zusammenfassung / 4 MIPS-Assemblersprache / 2

Kategorie	Befehl	Beispiel	Bedeutung	Anmerkungen
Logisch	and	and $s1,$s2,$s3	$s1 = $s2 & $s3	Bitweise UND
	or	or $s1,$s2,$s3	$s1 = $s2 ¦ $s3	Bitweise ODER
	nor	nor $s1,$s2,$s3	$s1 = ~($s2 ¦ $s3)	Bitweise NOR
	and imm.	andi $s1,$s2,7	$s1 = $s2 & 7	Bitweise UND mit Konst.
	or imm.	ori $s1,$s2,7	$s1 = $s2 ¦ 7	Bitweise ODER mit Konst.
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Linksschieben
	shift right logical	srl $s1,$s2,10	$s1 = $s2 >> 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 ($s1!=$s2) PC = PC + 4 + 100	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

Zusammenfassung / 5 MIPS-Assemblersprache / 3

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

Zusammenfassung / 6
4 Entwurfsprinzipien

- 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)
  - oder: Sei nicht dogmatisch