Technische Grundlagen der Informatik 2
Rechnerorganisation

Kapitel 2: Befehle - Die Sprache des Rechners

Prof. Dr. Ben Juurlink

Fachgebiet: Architektur eingebetteter System
Institut für Technische Informatik und Mikroelektronik
Fak. IV – Elektrotechnik und Informatik

SS 2014

Ziele

- Nach diesem Kapitel sollten Sie in der Lage sein:
  - C-Anweisungen in MIPS-Assemblersprache umzusetzen
  - komplexere Befehlssequenzen zu realisieren
    - **`while`**-Schleifen
    - **`for`**-Schleifen
    - **`switch`**-Anweisungen
    - (rekursive) Funktionen
  - MIPS-Registerkonventionen zu befolgen
  - zu erklären was ein Keller (*stack*) ist und wofür er gebraucht wird
  - ein Assemblerprogramm in Maschinensprache umzusetzen und umgekehrt
  - und vieles mehr…

Inhalte

- Computer-Operationen
- Computer-Operanden
- Darstellung von Befehlen im Rechner
- Logische Operationen
- Befehle zum Treffen von Entscheidungen
- Unterstützung von Prozeduren
- Charakters und Zeichenfolgen
- MIPS-Adressierung
- Ein komplettes Beispiel

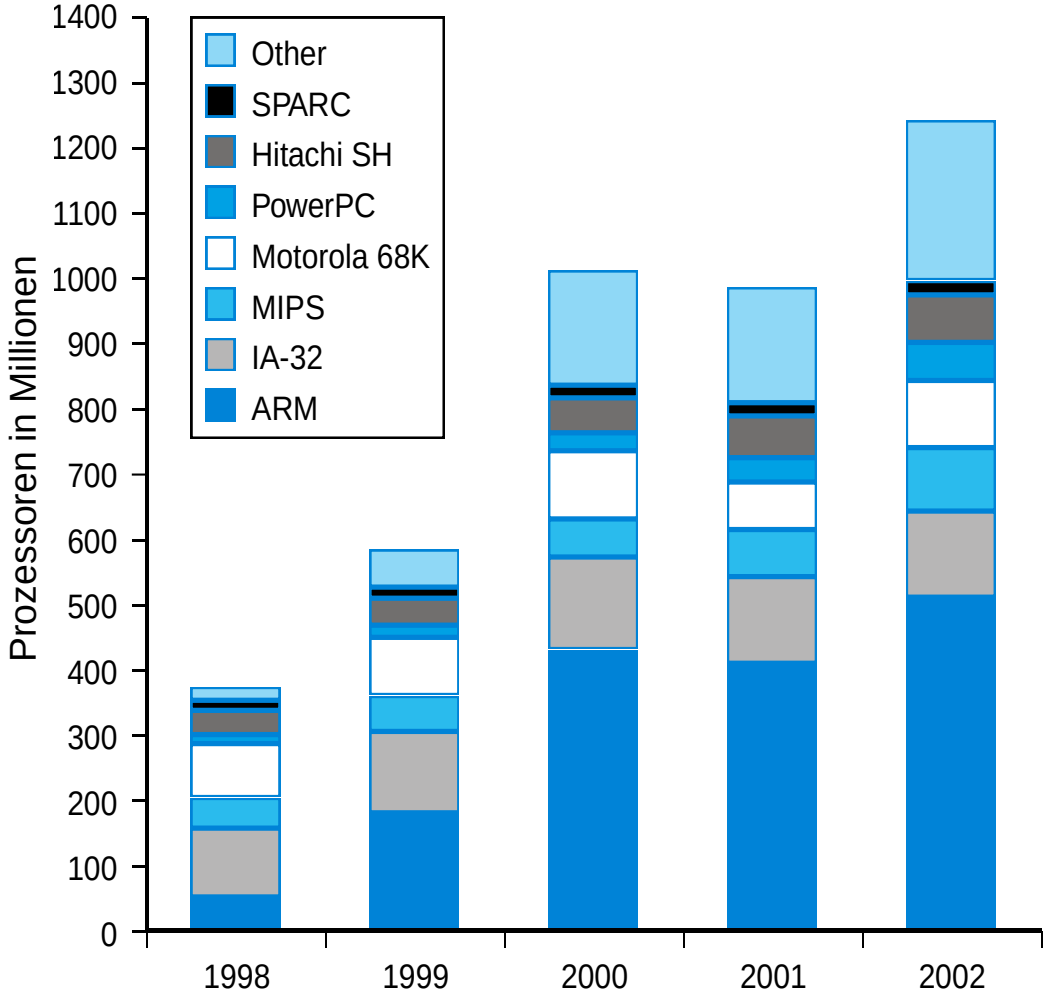
Befehle (*Instructions*)

- Die „Sprache“ des Rechners
- Viel primitiver als höhere Programmiersprachen
  - z. B. keine **`while`**-Schleifen, **`case/switch`**-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 minimiert.

- 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 voran)
- 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

- Wir möchten die Summe der Variablen b ($t1), c ($t2), d ($t3) und e ($t4) in Variable a ($t0) speichern
- C/Java:

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

- MIPS:

```
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 der 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

- Ein etwas komplexeres Beispiel:
- C/Java:

```
f = (g + h) - (i + j);  // f..j in $s0..$s4
```

- MIPS:

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

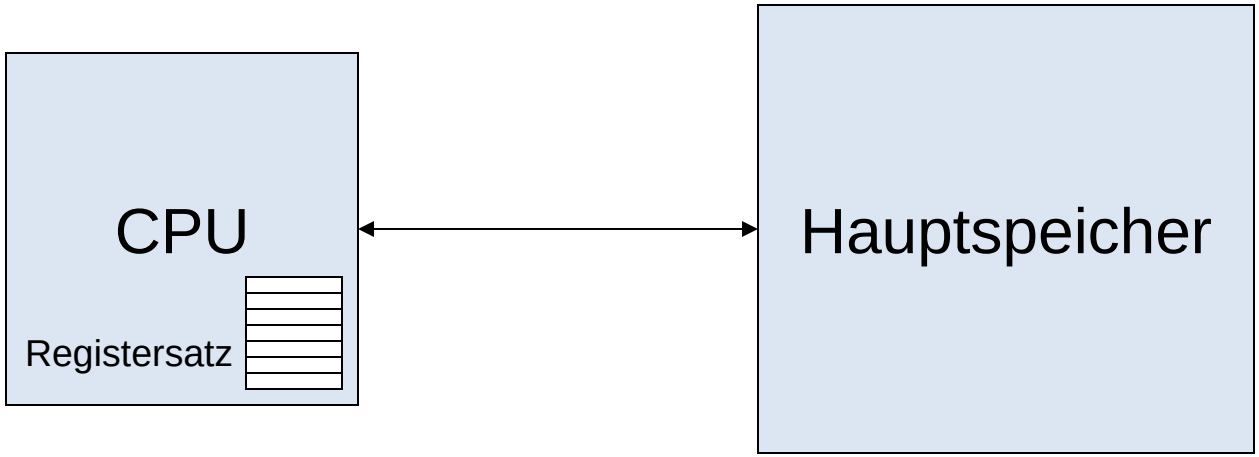
Operanden

- Operand: Quantität, worauf eine Operation ausgeführt wird
- MIPS kennt folgende Typen:
  - Registeroperanden
  - Speicheroperanden
  - Konstante oder Direktoperanden

MIPS Register

- MIPS verfügt über 32 integer- (fixed-point-) Register.
- In Assembler angedeutet als **`$0, $1, …, $31`** oder mit den symbolischen Namen **`$zero, $v0, …,$ra`**.
  - Bedeutung dieser symbolischen Namen wird später erklärt.
- Register 0 ($0 oder $zero) ist immer Null.
- Register sind 32 Bit breit (= ein Wort (*word*) in MIPS)
- Nur 32 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.

- 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 der Index 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
. . .	. . .

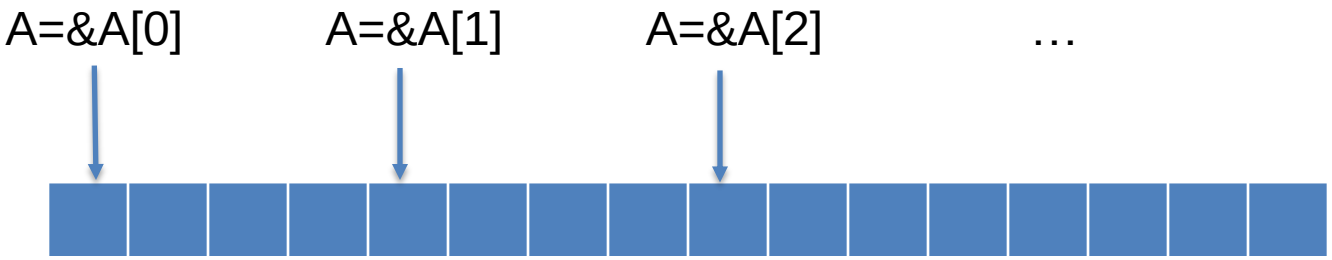
Speicherorganisation / 2

- Meisten Programme benutzen Wörter (words) 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, ...

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

- 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*

Datentransfer-Befehle

- Ladebefehl: *load word*

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

  - Konstante 12 wird *Offset* genannt.

- Speicherbefehl: *store word*

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

  - Bemerke: Ziel steht am Ende

Beispiel Datentransfer-Befehle

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

- *MIPS:*

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

Ausrichtung an Wortgrenzen

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

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

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

  - Bei Little-Endian („Klein-Ender“) 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:

```
int temp, k, v[100];
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

- MIPS (Annahmen: Basisadresse `v` in `$a0`, `k` in `$a1`):

```
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 einen kürzeren Weg lernen um `4*k` zu berechnen.
- `&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 eins 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 kein **`subi`** Befehl. Wieso nicht?

3. Entwurfsprinzip

- Entwurfsprinzip 3: ***Make the common case fast*** (Optimiere den häufig vorkommenden Fall)
  - 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-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 Datenstrukturen, 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

- Jetzt: Darstellung Befehlen im Rechner
  - dazu müssen wir Dualzahlen „lesen und schreiben“ können

- Menschen benutzen Dezimalzahlen

  $2435 = 2 \cdot 10^3 + 4 \cdot 10^2 + 3 \cdot 10^1 + 5 \cdot 10^0$

- Rechner benutzen Binärzahlen

  $1011_B = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 11_D$

- Zwei einfach darzustellen
  - ein / aus
  - hohes /niedriges Potenzial
  - schwarz / weiß
- i. A.: $b_{n-1}b_{n-2}. . .b_1b_0 = b_{n-1} \cdot 2^{n-1} + b_{n-2} \cdot 2^{n-2} + . . . + b_1 \cdot 2^1 + b_0 \cdot 2^0$

Konvertierung

- Dezimal nach dual / binär:

167_D →	167 / 2 = 83	Rest 1
	83 / 2 = 41	Rest 1
	41 / 2 = 20	Rest 1
	20 / 2 = 10	Rest 0
	10 / 2 = 5	Rest 0
	5 / 2 = 2	Rest 1
	2 / 2 = 1	Rest 0
	1 / 2 = 0	Rest 1

***least significant bit* (LSb)**

***most significant bit* (MSb)**

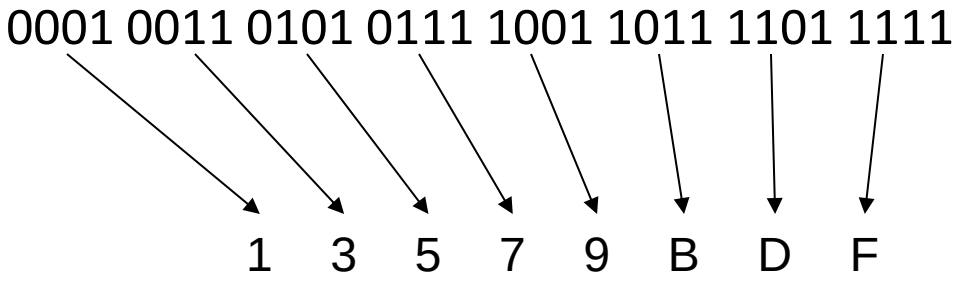
- $167_D = 10100111_B$

- Um lange Folgen mit Binärzahlen zu vermeiden, werden oft Hexadezimalzahlen (Basis 16) verwendet.
- Ziffernmenge: { 0, 1, 2, … , 8, 9, A, B, … ,F }
- Dezimal nach hexadezimal:

  167_D -> 167 / 16 = 10 Rest 7

  10 / 16 = 0 Rest A

  - $167_D = A7_H$
  - In C/Java: 0xa7
- Binär nach hexadezimal:



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*)
- Register haben Zahlen
  - **`$s0`** bis **`$s7`** sind Register 16 bis 23
  - **`$t0`** bis **`$t7`** sind Register 8 bis 15
  - Beispiel: **`add $t0, $s1, $s2`**

Maschinensprache / 2

- Befehlsformat:

	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?*
- Bemerke: In Assemblersprache Zielregister vorne, in Maschinensprache hinten.

- Felder im MIPS-Befehlsformat
  - *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 dazu)
  - *funct*: Funktionscode (*function code*). Für viele Befehle bestimmen *op* und *funct* zusammen die Operation.

Maschinensprache / 4

- Was ist mit den `lw`- und `sw`-Befehlen?
  - **`lw $t0,1000($t1)` und `sw $t4,12($t1)`**
  - Nach dem Regelmäßigkeitsprinzip nur 5 Bits für den konstanten Offset (Offset auf $2^5 = 32$ begrenzt)
  - oder verschiede Befehlslängen
- Entwurfsprinzip 4: ***Good design demands compromises*** (Ein guter Entwurf fordert Kompromisse)
- Neues Befehlsformat:
  - *I-Typ* oder *I-Format* (I für *immediate* = direkt)
  - Vorige Format ist *R-Typ* oder *R-Format* (R für Register)

- 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

- Im `lw`-Befehl gibt das `rt`-Feld das Zielregister an (*target register*).
- 16-Bit-Adresse bedeutet, dass ein beliebiges Wort im Bereich von $\pm 2^{15}$ oder 65.536 Byte ab Adresse im Basisregister `rs` geladen werden kann.
- Auch Konstanten im `addi`-Befehl sind auf $\pm 2^{15}$ beschränkt.

- Beispiel: MIPS-Assemblersprache in Maschinensprache übersetzen
- C/ Java: `A[300] = h + A[300];`
- MIPS:

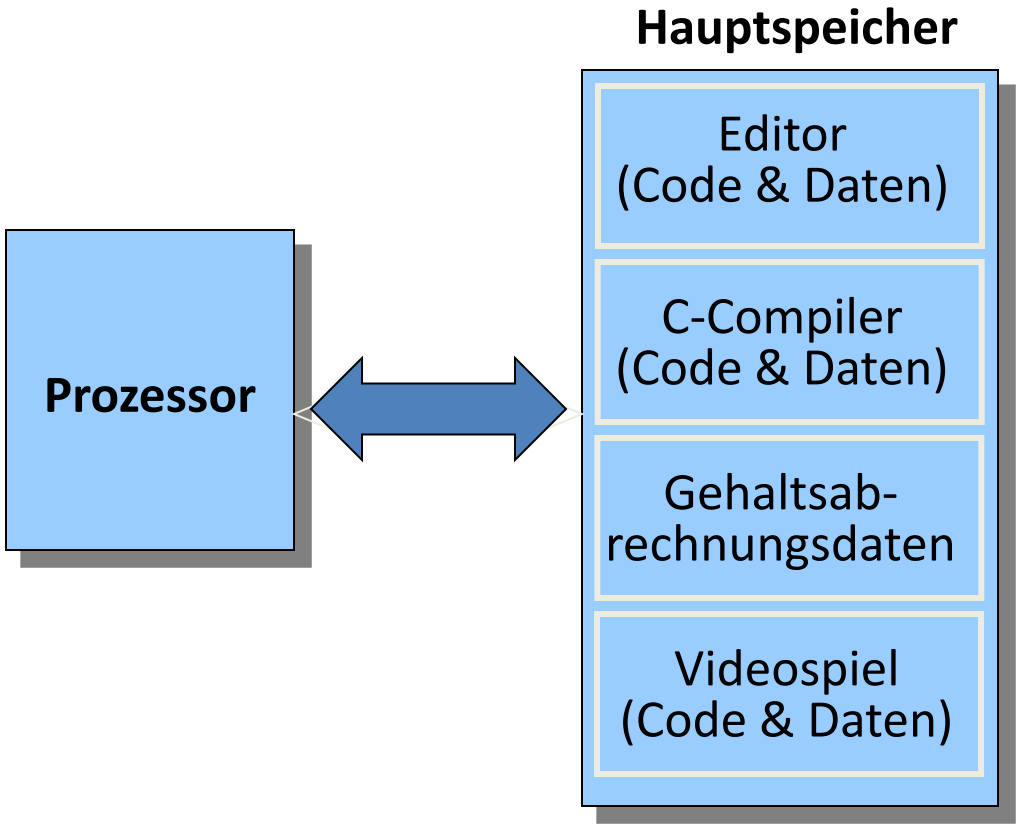
```
lw   $t0,1200($t1)   # $t0 = A[300]
add  $t0,$s2,$t0     # $t0 = h+$t0
sw   $t0,1200($t1)   # A[300] = $t0
```

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

- 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.
- 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.

John von Neumann

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 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];
    perform instr;
    PC = PC+4;
}
```

Logische Operationen / 1

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

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

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`
  - $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 (Bit „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 / 4

- Letzte logische Basisoperation ist die Negation: NICHT (*NOT*)
- MIPS hat keine NICHT-Operation.
- Stattdessen NOR (NICHT ODER)
  - nur 1 wenn beide Operanden 0
- Entspricht NICHT wenn ein Operand 0 ist:
  - A NOR 0 = NOT(A OR 0) = 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

Kontrollbefehle / 1

- 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
```

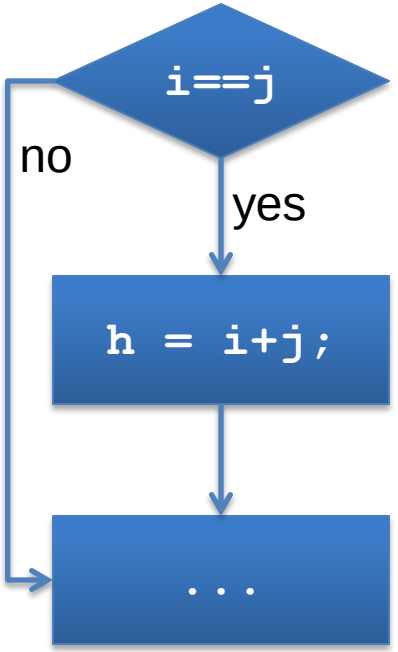
Übersetzung einer *If-then*-Anweisung:

- C/Java:

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

- MIPS:

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



- Effizienter um zu überprüfen, ob gegenteilige Bedingung erfüllt ist.
- Marker (*label*) **`endif`** wird vom Assembler übersetzt zu einer Adresse.

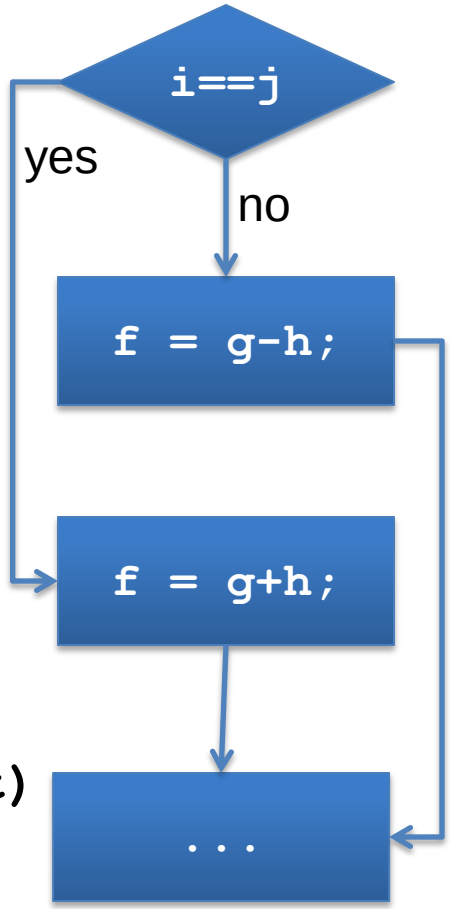
Übersetzung einer *If-then-else*-Anweisung:

- C/Java:

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

- MIPS:

```
  # $s0,…,$s4 beinhalten i, j, f, g und h
  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: ...
```



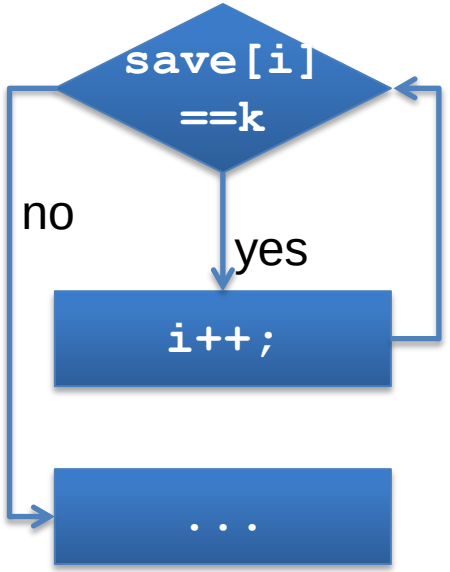
Übersetzung einer *While*-Schleife:

- C/Java:

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

- MIPS:

```
# $s0, $s1 beinhalten i und k
# $s2 beinhaltet save[] (Basisadresse)
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 ($t0!=k) goto endwhile
    addi $s0,$s0,1          # i++
    j    while              # goto while
endwhile:
    ...
```



Kleiner oder größer? / 1

- Wie steht's mit z. B.

```
if (a<0) a = -a;
```

  aus?

- 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)
```

- 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   $s1,$s2,label   # if ($s1<=$s2) goto label
```

- Echte MIPS-Befehle:

```
slt   $t0,$s2,$s1     # $t0 = ($s2 < $s1)
beq   $t0,$zero,label # if (!$t0) [$s2 >= $s1]
                      # goto label
```

- **`blt`**, **`ble`**, **`bgt`**, **`bge`**, … werden vom Assembler akzeptiert (Pseudo-befehle (*pseudo-instructions*)) und übersetzt zu echten MIPS-Befehlen.
- Assembler braucht einen Register dazu:
  - Register **`$at`** (*assembler temporary*)

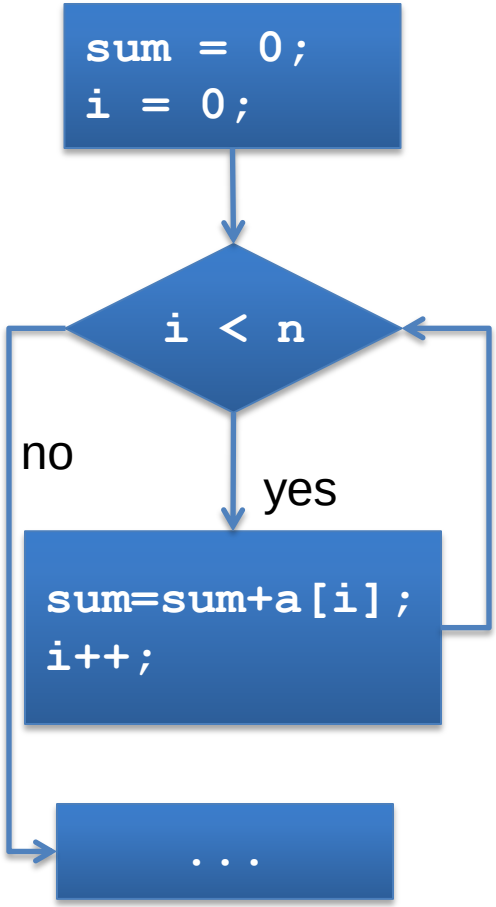
For-Schleifen

- 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++;
}
```



For-Schleifen

- C/Java:

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

- MIPS:

```
      # $a1 beinhaltet n und sum in $v0
      # $a0 beinhaltet a[] (Basisadresse des Arrays)
      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 Register `$a0` befindet und `n` in `$a1`.
- `min` soll in `$v0` abgelegt werden.

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

```
# Loesung:
addi $t0,$zero,1     # i
lw   $v0,0($a0)      # m
for:
bge  $t0,$a1,endfor  #
sll  $t1,$t0,2       #
add  $t1,$a0,$t1     #
lw   $t1,0($t1)      #
bge  $t1,$v0,endif   #
move $v0,$t1         #
endif:
addi $t0,$t0,1
j    for
endfor:
```

Adressbildung bei Verzweigungen

- 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	5	16	17	16-Bit Adress-Offset

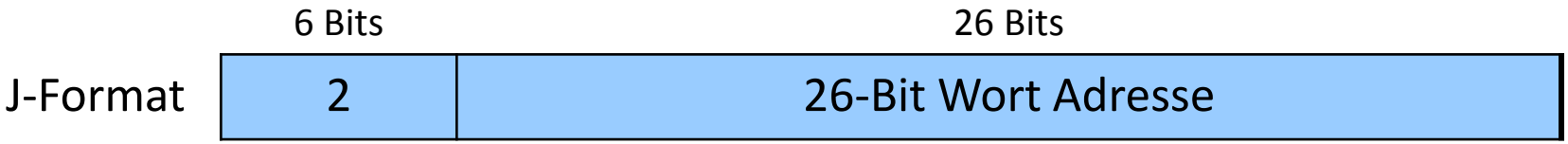
- Befehlszähler (*program counter, PC*) enthält die Adresse des Befehls, der gerade ausgeführt wird.
- 16-bit Adresse in Verzweigungen ist *relativ* zu *PC**+4*** (*Offset*)
  - Auf diese Weise können Programme Größen bis zu 2^{32} Bytes annehmen.
  - Bedingte Sprünge verweisen oft auf nahe gelegene Befehle.
  - Offset ist außerdem Wortoffset (Zieladresse ist PC + 4 + **4***Offset).
  - Befehlszählerrelative Adressierung (*PC-relative adressing*)

Adressbildung bei Sprüngen

- Befehl:

```
j label     # goto label
```

- Neues Format:



- Wortadresse wird erneut mit 4 multipliziert.
- Restlichen 4 Bits werden vom PC genommen.
- Beispiel:
  - `j 1000` # $PC_{31-0} = PC_{31-28}$ #4000
- Adressgrenze von 256 MB (64 Millionen Befehle)

Übung

- Übersetze folgenden MIPS-Assemblercode zu MIPS-Maschinencode.
  - Nehme an, die Schleife beginnt an Adresse 80000 im Hauptspeicher.
  - zeige dezimalen Wert aller Befehlsfelder
  - benutze nächste 2 Folien

```
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

`sll rd,rt,shamt`	0	0	rt	rd	shamt	0
`add rd,rs,rt`	0	rs	rt	rd	0	32
`lw rt,offs(rs)`	35	rs	rt	offs		
`bne rs,rt,addr`	5	rs	rt	addr		
`addi rt,rs,imm`	8	rs	rt	imm		
`j addr`	2	addr				

MIPS-Register

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

Lösung

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

Lösung

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

Fazit

- 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`.
- 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				

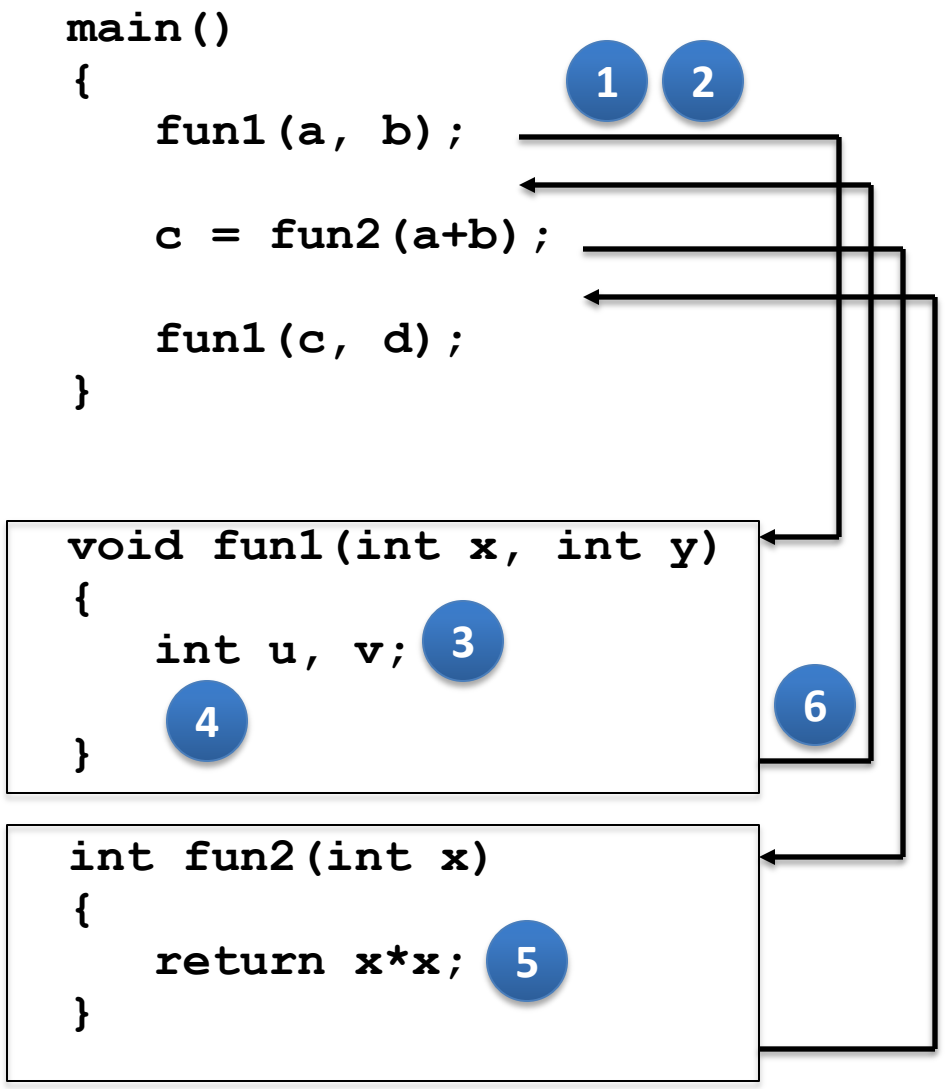
Prozeduren und Funktionen / 1

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

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

Beispiel

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
- C :

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

- MIPS:

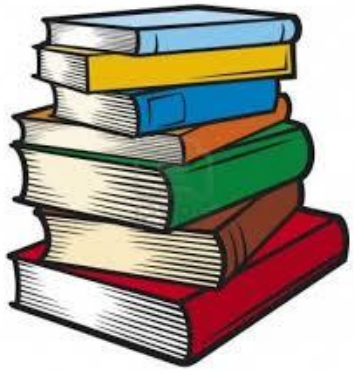
```
swap:  sll $t0,$a1,2    # $t0 = 4*k
       add $t0,$a0,$t0  # $t0 = &v[k]
       lw  $t1,0($t0)   # temp = v[k]
       lw  $t2,4($t0)   # $t2 = v[k+1]
       sw  $t2,0($t0)   # v[k] = v[k+1]
       sw  $t1,4($t0)   # v[k+1] = temp
       jr  $ra          # Rücksprung (return)
```

- Caller:

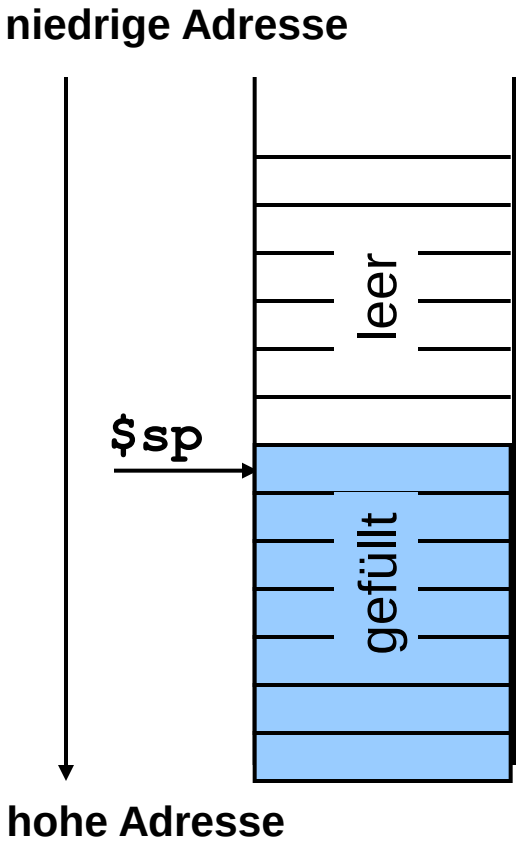
```
jal swap
```

Keller (*Stack*) / 1

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

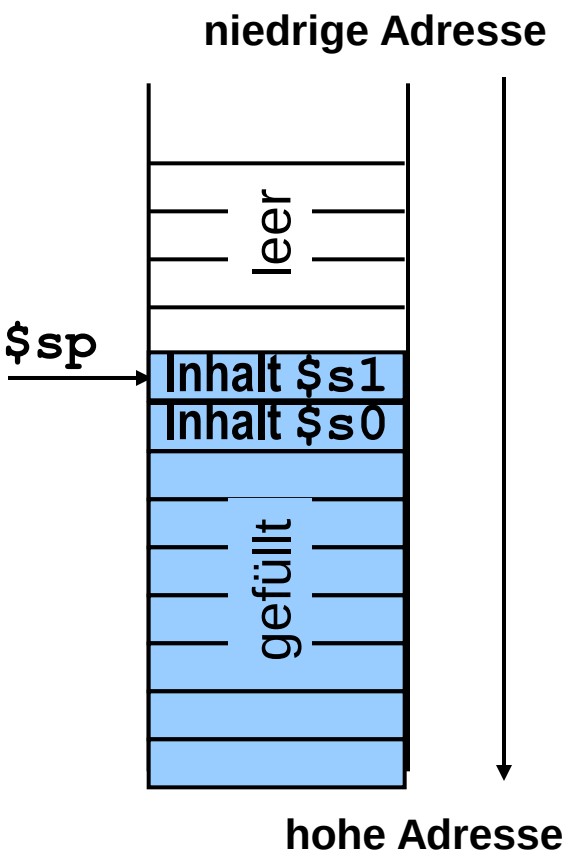


Keller (*Stack*) / 2

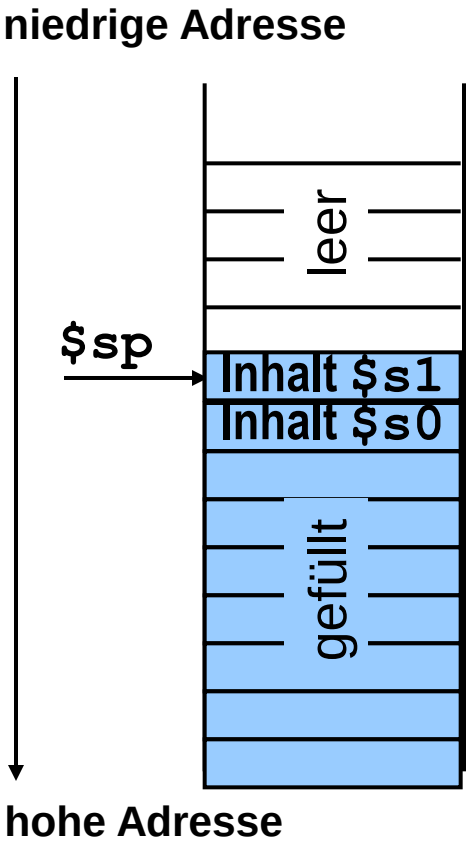


Ablegen von `$s0` und `$s1`:

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

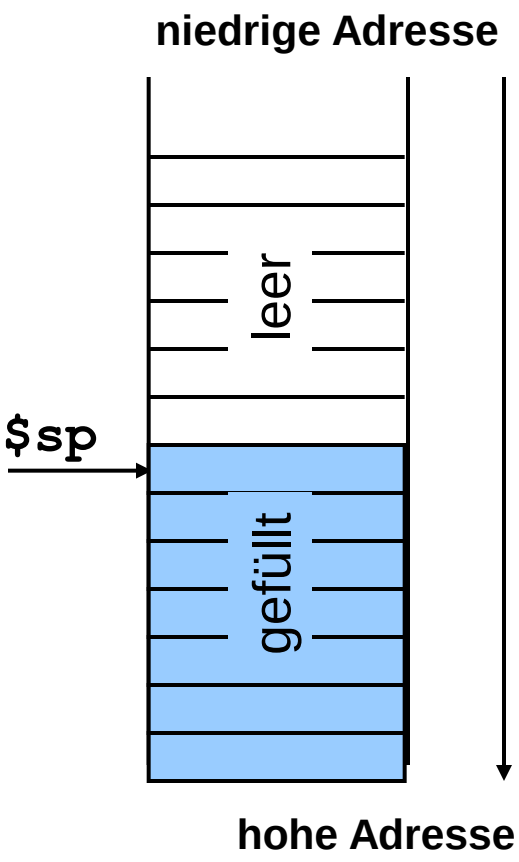


Keller (*Stack*) / 3



Entfernen von $s0 und $s1:

```
lw   $s0,4($sp)
lw   $s1,0($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-$s7`	16-23	gespeicherte Variablen	ja
`$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

MIPS-Registerkonventionen / 2

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

- C:
```
int poly(int x){
        return square(x)+x+1;
}
```
- MIPS:
```
poly: addi $sp,$sp,-8    # Stack-Reservierung für 2 Variable
      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
```

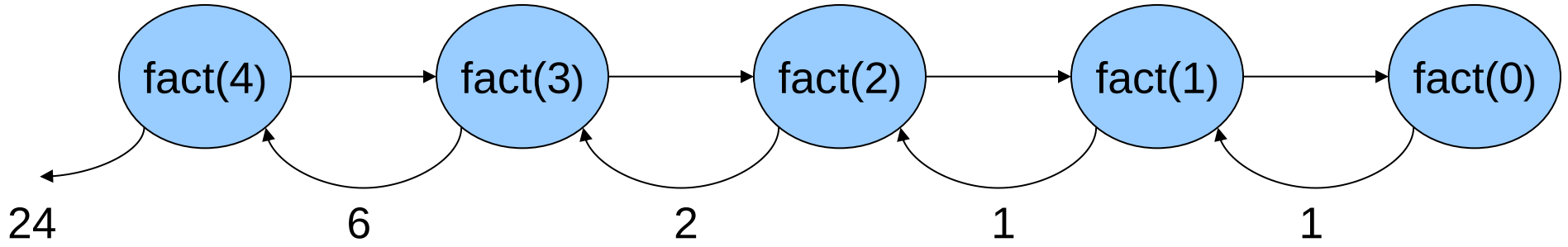
Rekursive Funktionen

- 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

Illustration




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

MIPS-Assemblercode / 1

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

```
addi $sp,$sp,-8    # Stack-Reservierung für 2 Variable
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:

```
sw   $ra,4($sp)    # speichere $ra auf Stack
sw   $a0,0($sp)    # speichere $a0 auf Stack
addi $sp,$sp,8     # wiederherstellen des Stackpointers
```

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

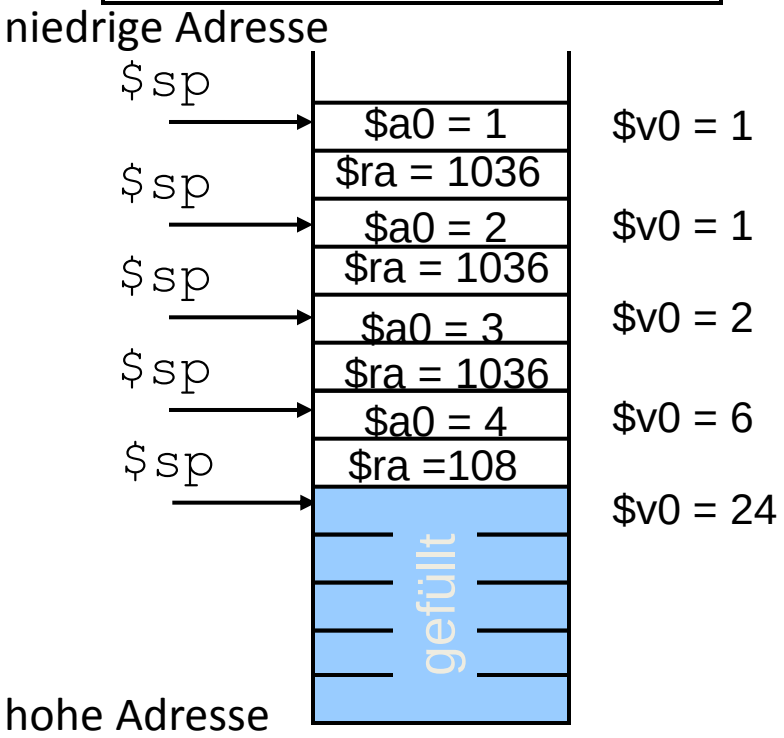
Befehlsadresse (Instruction address)

```
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?!

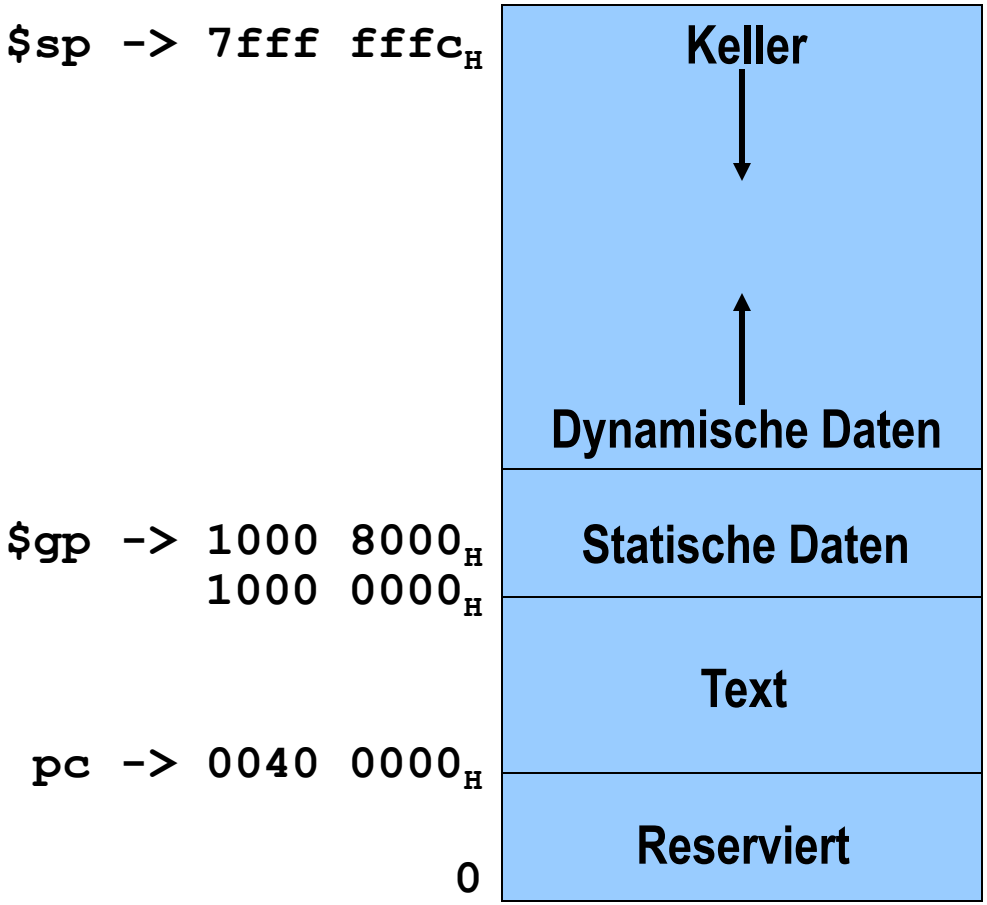
Caller

```
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)

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

- *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

„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:

```
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)
```

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 `ori` verwenden
- Beispiel: lade $255 \times 2^{16} + 255 = 16.711.935$

```
lui $t0,255
```

$t0 =	0000 0000 1111 1111	0000 0000 0000 0000

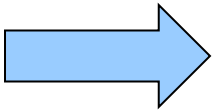
```
ori $t0,$t0,255
```

$t0 =	0000 0000 1111 1111	0000 0000 1111 1111

Weite Verzweigung

- Mit `beq` und `bne` kann man 2^{15} Befehle vorwärts und 2^{15}-1 Befehle rückwärts springen.
  - meiste Sprünge innerhalb beschränkten Adressbereichs
- Selten muss weiter verzweigt werden.
- Trick:

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



```
        bne   $s0,$s1,L2
        j L1
L2:     ...
```

MIPS-Adressierungsarten

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

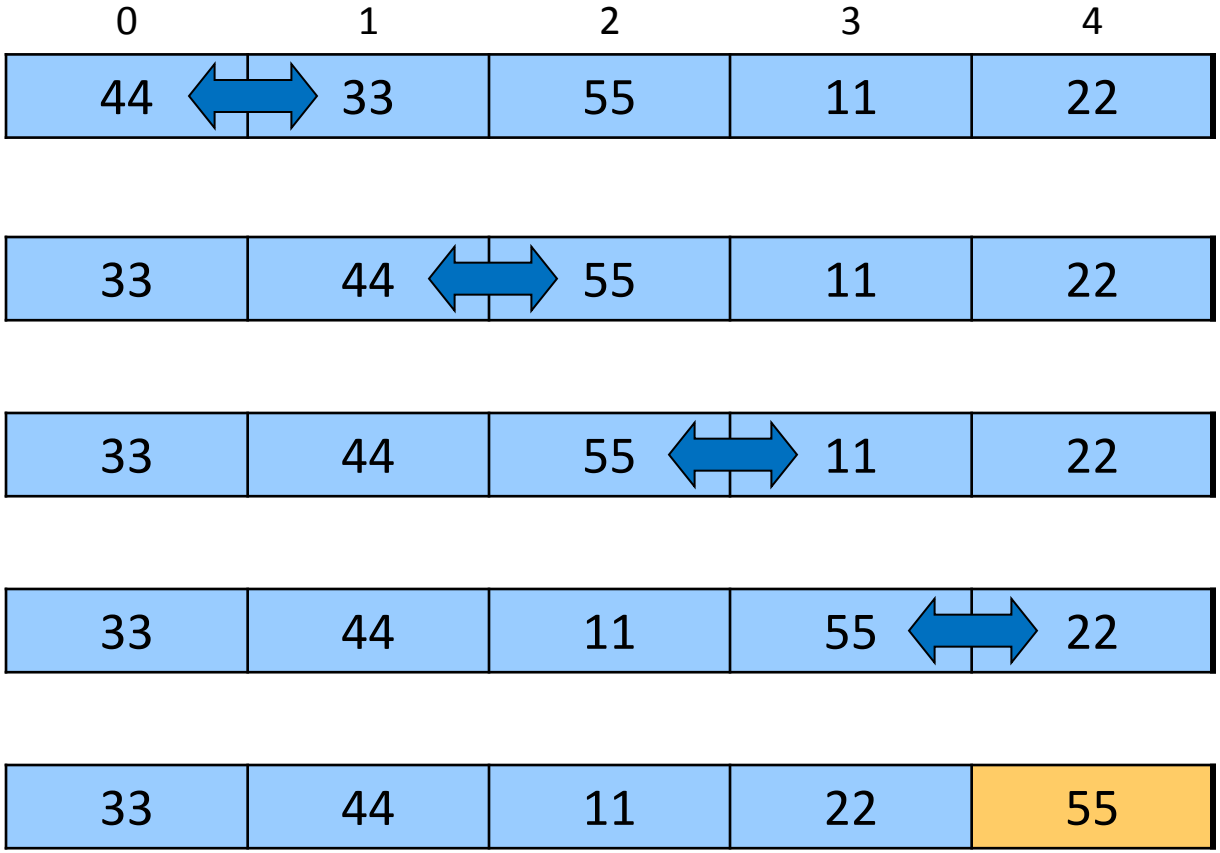
MIPS-Adressierungsarten:

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)`

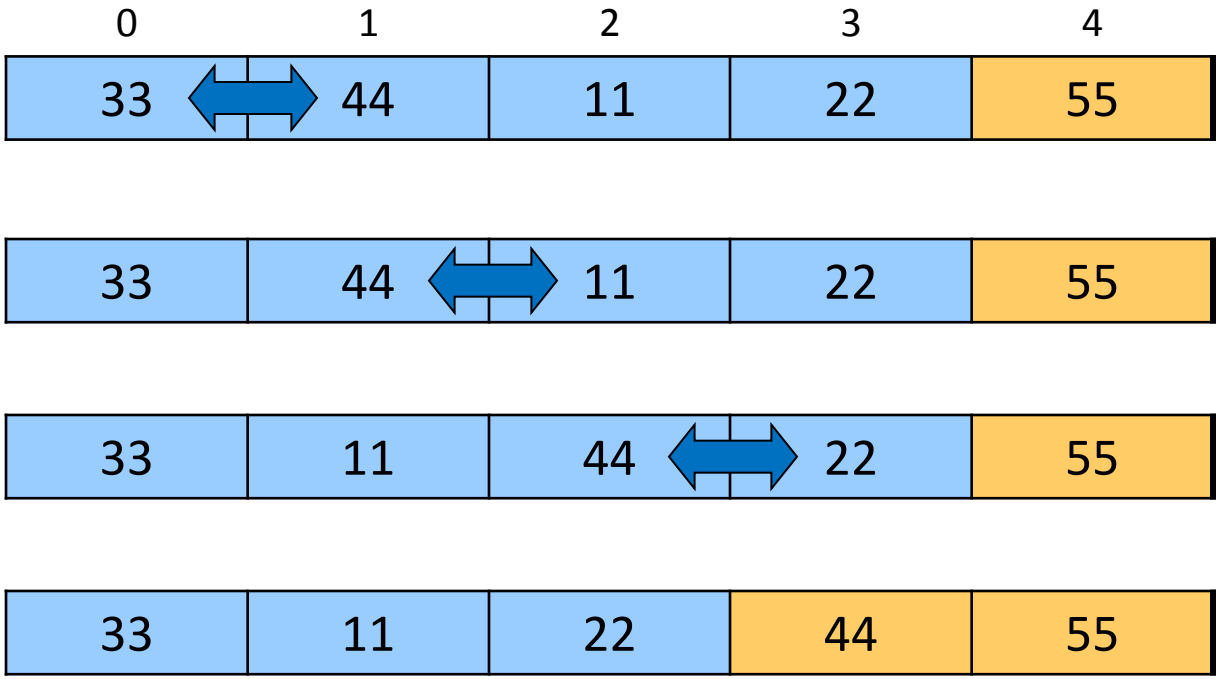
Gesamtes Programm in Assemblersprache

- 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

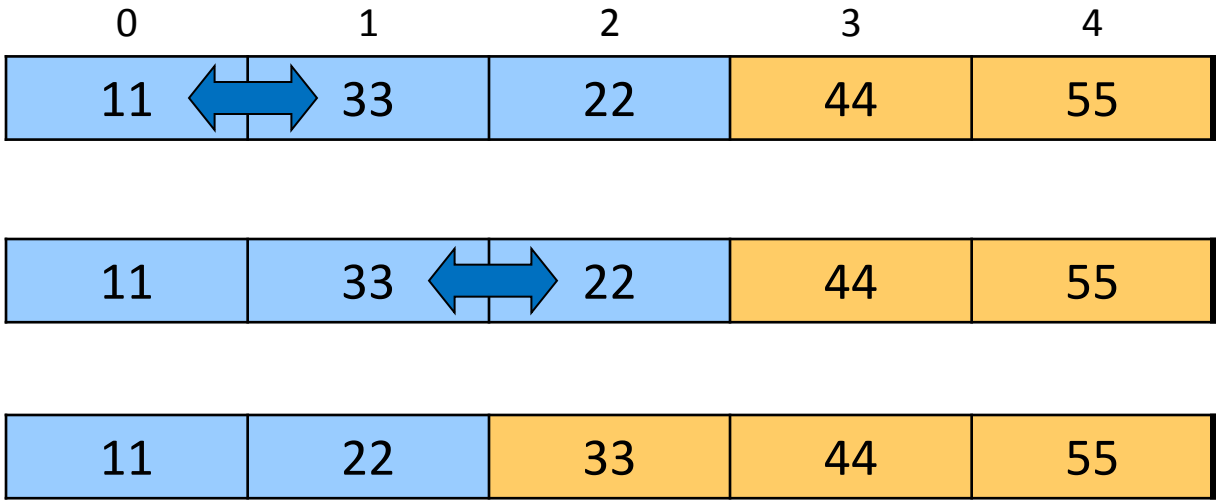
Bubble Sort, äußere Schleife 1



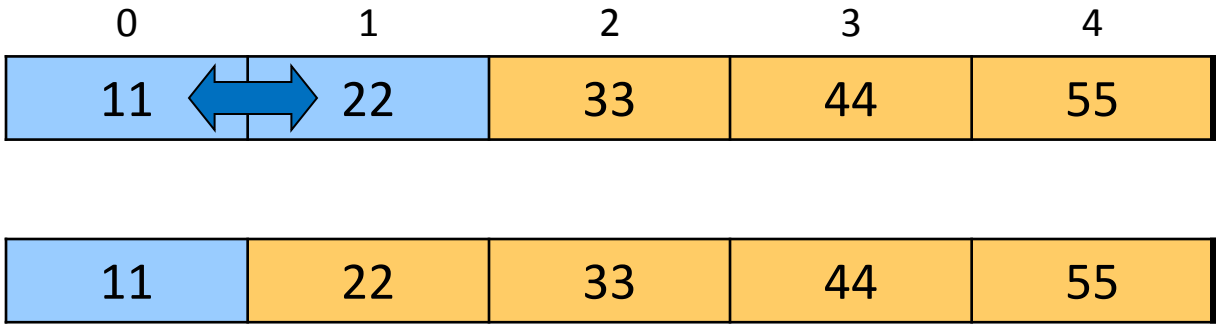
Bubble Sort, äußere Schleife 2



Bubble Sort, äußere Schleife 3



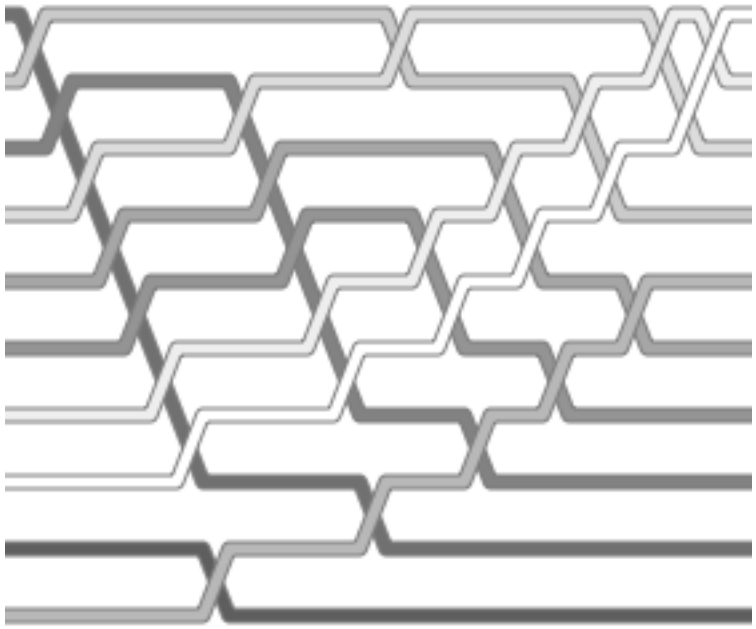
Bubble Sort, äußere Schleife 4



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;
}
```



Assemblercode für swap

- Registerkonvention:
  - `v[]` in `$a0`, `k` in `$a1`
  - brauchen nicht gesichert zu werden

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

- Blattfunktion
  - verwende (wenn möglich) sonst nur temporäre (`$t`i) Register

```
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`

```
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);
}
```

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

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);
}
```

```
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
```

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);
}
```

```
        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]) go endif
```

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);
}
```

```
        ble     $t1,$t2,endif       # if (v[j]<=v[j+1]) go 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:
```

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);
}
```

```
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)
```

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

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 = 100×2^{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 + 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

rot = richtige Kategorie?

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	

- *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

Optional

Case/Switch-Anweisung / 1

- Case/switch-Anweisung
  - könnte als Kette von If-then-else-Anweisungen implementiert werden.
  - Laufzeit proportional zu Anzahl der Fälle
  - geht schneller mithilfe Sprungadresstabelle (*jump address table*) und `jr`-Befehl
- Beispiel:

```
switch (k){
  case 0: f = i+j; break;
  case 1: f = g+h; break;
  case 2: f = g-h; break;
  case 3: f = i-j; break;
}
```

Sprungadresstabelle

Adresse case 0
Adresse case 1
Adresse case 2
Adresse case 3

Case/Switch-Anweisung / 2

```
switch (k){
  case 0: f = i+j; break;
  case 1: f = g+h; break;
  case 2: f = g-h; break;
  case 3: f = i-j; break;
}
```

Sprungadresstabelle

$a0	Adresse case 0
$a0+4	Adresse case 1
$a0+8	Adresse case 2
$a0+12	Adresse case 3

- Assemblercode
  - testet ob $0 \leq k \leq 3$
  - berechnet Adresse `jtable[k]`
  - lädt `jtable[k]` und springt
  - Code für alle Fälle
- Annahmen:
  - `k` in `$s5`, Basisadresse `jtable` in `$a0`, `f-j` in `$s0-$s4`

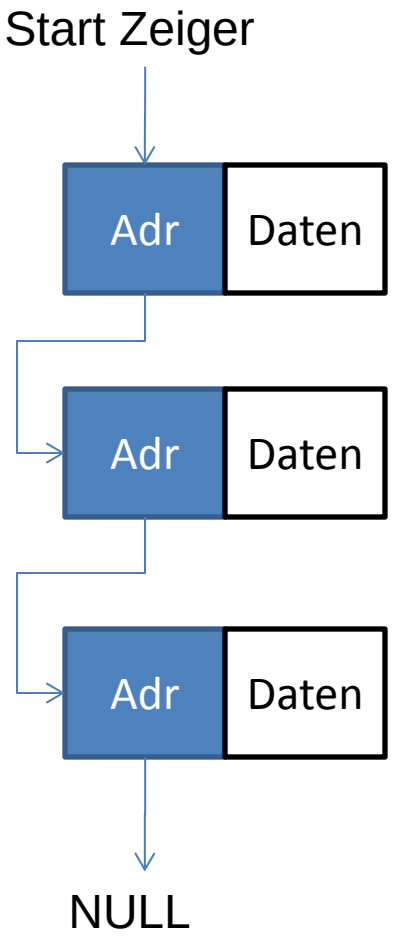
Case/Switch-Anweisung / 3

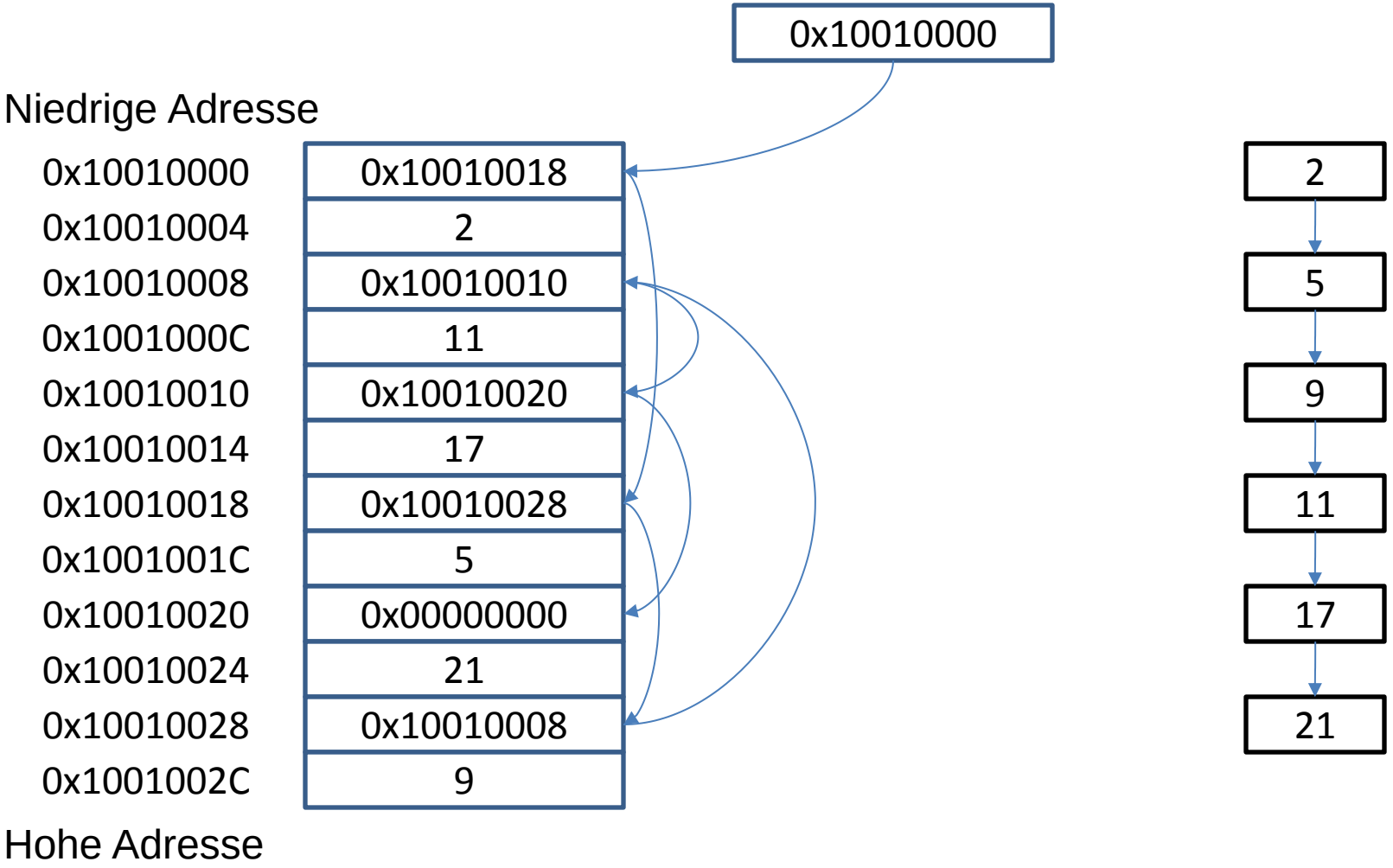
```
switch (k){
    case 0: f = i+j;
            break;
    case 1: f = g+h;
            break;
    case 2: f = g-h;
            break;
    case 3: f = i-j;
            break;
    }
```

```
        slt  $t3,$s5,$zero  # $t3 = (k<0)
        bne  $t3,$zero,exit # if ($t3) goto exit
        slti $t3,$s5,4      # $t3 = (k<4)
        beq  $t3,$zero,exit # if (!$t3) goto exit
        sll  $t1,$s5,2      # $t1 = 4*k
        add  $t1,$t1,$a0    # $t1 = &jtable[k]
        lw   $t0,0($t1)     # $t0 = jtable[k]
        jr   $t0            # goto $t0
L0:     add  $s0,$s3,$s4    # f = i+j
        j    exit           # goto exit
L1:     add  $s0,$s1,$s2    # f = g+h
        j    exit           # goto exit
L2:     sub  $s0,$s1,$s2    # f = g-h
        j    exit           # goto exit
L3:     sub  $s0,$s3,$s4    # f = i-j
exit:   ...
```

Einfach verkettete Liste

- Dynamische Datenstruktur
  - Anzahl der Elemente muss vorher nicht festgelegt werden
  - Müssen nicht nacheinander im Speicher abgelegt sein
- Zum Beispiel: Integer Liste
- Einzelne Elemente -> Knoten (engl. Node)
  - Daten (4 Byte)
  - Zeiger (4 Byte) (engl. Pointer)
- Einfach verkettete Liste:
  - „Start“ Zeiger
  - Knoten
  - Letztes Element zeigt auf NULL




0x10010000
Niedrige Adresse
0x10010000
0x10010004
0x10010008
0x1001000C
0x10010010
0x10010014
0x10010018
0x1001001C
0x10010020
0x10010024
0x10010028
0x1001002C
0x10010018
2
0x10010010
11
0x10010020
17
0x10010028
5
0x00000000
21
0x10010008
9
Hohe Adresse
2
5
9
11
17
21

Einfach verkettete Liste

- Addition aller Elemente einer Liste
- C:

```
struct LLnode {
     struct LLnode *next;
     int data;
};
-------------------------------------
int listsum (LLnode *node){
     int sum=0;
     while(node->next != NULL){
            sum = sum + node->data;
            node=node->next;
     }
     return sum;
}
```

- **MIPS:**

```
listsum:
add $v0,$zero,$zero      # Initalisiere v$0=0
add $t0, $a0,$zero       # Adresse node->next nach $t0

next:
beq $t0, $zero, end      # falls next==0 goto end
lw  $t1, 4($t0)          # Lade Daten in $0
add $v0, $v0, $t1        # addiere Inhalt zu $v0
lw  $t0, 0($t0)          # lade naechste Adresse
j   next                 # goto next

end:
jr  $ra                  # return
```