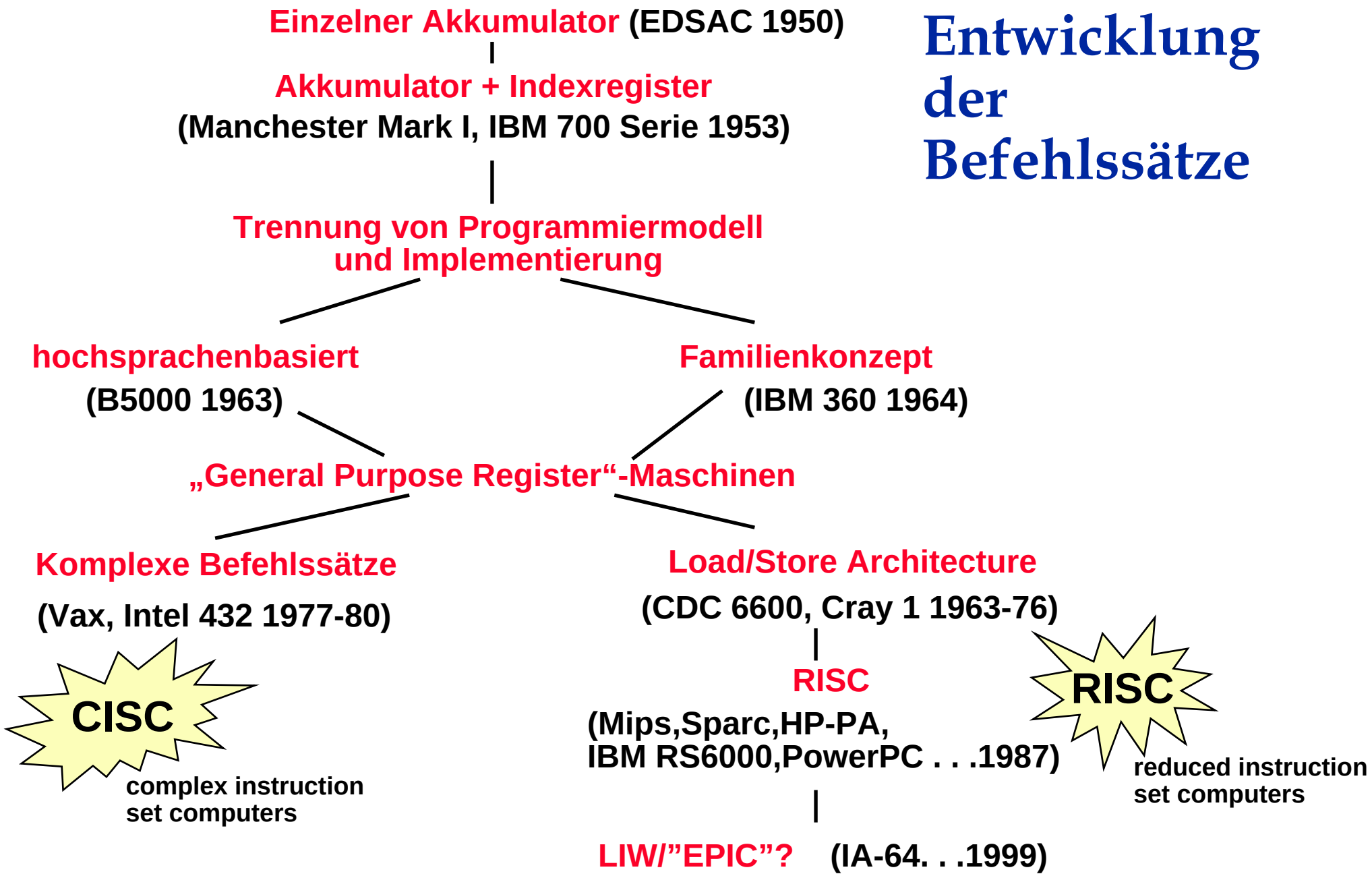
8. Assembler-programmierung

```
Loop:   addi $s4,$s4,1          # j = j + 1?
        add $t1,$s3,$s3         # $t1 = 2 * i
        add $t1,$t1,$t1         # $t1 = 4 * i
        add $t1,$t1,$s5         # $t1 = @ A[i]
        lw $t0,0($t1)           # $t0 = A[i]
        addi $s3,$s3,4          # i = i + 1?
        slti $t1,$t0,10         # $t1 = $t0 < 10?
        beq $t0,$0, Loop        # goto Loop if >=
        slti $t1,$t0, 0         # $t1 = $t0 < 0?
        bne $t0,$0, Loop        # goto Loop if <
```

Assemblerbefehle
Ausführungsmodelle
Speicherorganisation
Ein-/Ausgabe
Prozeduren

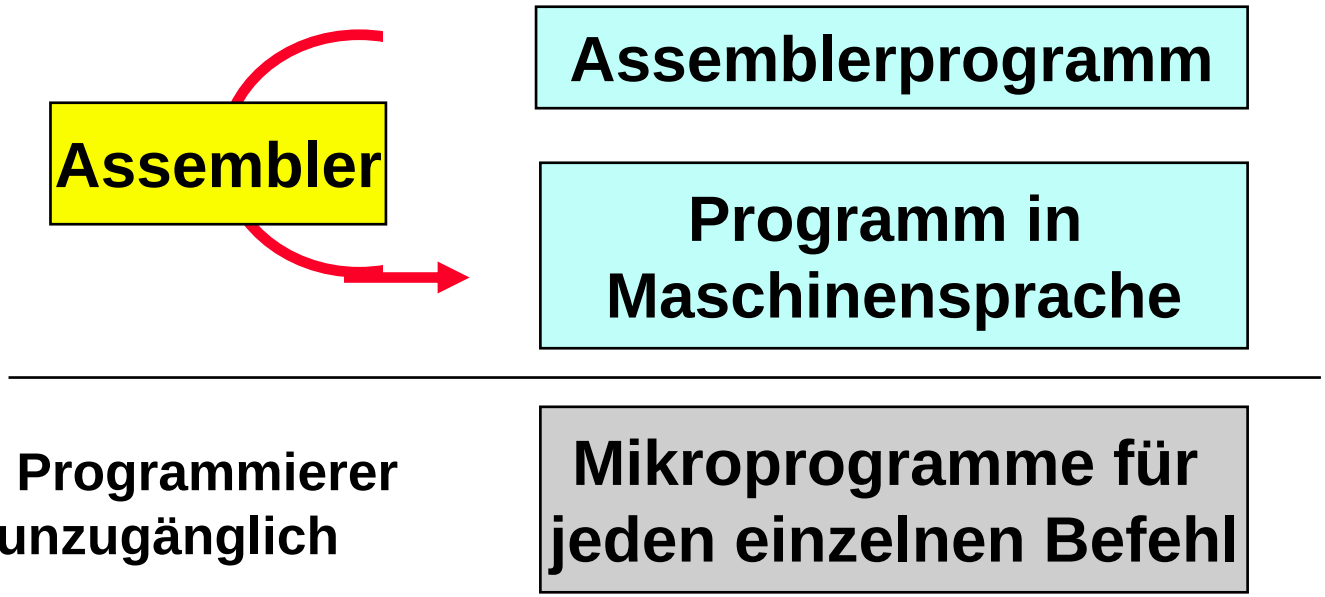
Entwicklung der Befehlssätze



Maschinensprache, Assembler

Unter der Maschinensprache versteht man die Sammlung der Befehle, die einem Programmierer zur Verfügung stehen. Jeder Maschinenbefehl ist durch ein Mikroprogramm implementiert oder fest verdrahtet.

Eine Assemblersprache ist eine lesbare Art, Programme in Maschinensprache zu formulieren. Ein Assembler ist ein Programm, das Befehlsfolgen von der Assemblersprache in die Maschinensprache übersetzt.



Nachteile der Maschinensprache

Programme in Maschinensprache sind schwer lesbar.

Es können keine Variablennamen, keine Registernamen, keine symbolischen Sprungziele verwendet werden und kein Platz für Daten reserviert werden.

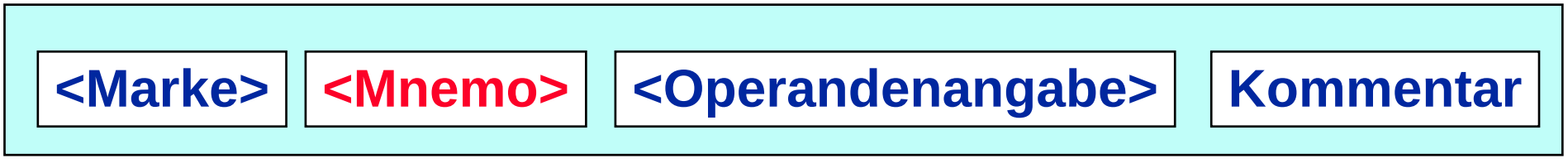
```
add   $t1,$t1,$s5
lw    $t0,0($t1)
addi  $s3,$s3,4
```

Assemblerprogramm

```
0010101111000011
1000101111001000
10111010000000000000000
```

Durch Assembler daraus erzeugtes Maschinenprogramm

Aufbau von Assemblerinstruktionen



<Marke>: symbolische Bezeichnung einer Speicheradresse (optional)

<Mnemo>: symbolische Bezeichnung einer Operation

<Operandenangabe>: Notwendigkeit und Art hängt vom jeweiligen <Mnemo> ab

<Kommentar>: beliebige Zeichenfolge zur Erläuterung der Instruktion

Ausführungsmodelle

Art und Weise der Operandenübergabe und der Speicherung der Resultate

Modelle:

- **Register:**
  - » explizite Angabe von Operanden- und Resultatadressen
  - » Operanden befinden sich in Registern oder im Speicher
- **Akkumulator:**
  - » spezielles Akkumulatorregister enthält jeweils einen Operanden und ist außerdem Zieladresse jeder Operation
  - » nur eine explizite Operandenangabe erforderlich
- **Keller (Stack):**
  - » implizite Angabe von Operanden und Ergebnisadressen
  - » alle Operationen werden auf einem Kellerspeicher ausgeführt

Strukturmerkmale verschiedener Rechner

	IBM 370/168	VAX 11/780	XEROX Dorado	Intel IAPX 432	SPARC	MIPS
Jahr	1973	1978	1978	1982	1987	1986
Anzahl der Befehle	208	303	270	222	64	64
Mikroprog.-speicher (Kbit)	420	480	136	64		
Befehls-format (Bits)	16 - 48	16 - 456	8 - 24	6 - 321	32	32
Technologie	ECL MSI	TTL MSI	ECL MSI	NMOS VLSI	CMOS VLSI	CMOS VLSI
Ausführungs-modelle	Reg-Sp Sp-Sp Reg-Reg	Reg-Sp Sp-Sp Reg-Reg	Keller	Keller Sp-Sp	Reg-Reg	Reg-Reg

Maschineninstruktionen

- **Maschineninstruktionen (Maschinenbefehle):**

  Angaben, die den Assembler veranlassen, Bitmuster zu erzeugen, die vom Prozessor als Befehle ausgeführt werden können (Maschinencode)

- **Maschinenbefehlsklassen:**
  - **Arithmetische und logische Operationen**
    - » **z.B. ADD, SUB . . . NOR**
    - » **Vergleichsbefehle**
    - » **Shift- und Rotationsbefehle**
  - **Datentransfer**
    - » **Lade- und Speicherbefehle**
    - » **Speicher <-> Register, Register <-> Register**
  - **Steuerbefehle**
    - » **Sprungbefehle: unbedingt, bedingt, relativ**
    - » **Kontrollbefehle: STOP, Unterprogrammsprung**
  - **Ein-/Ausgabebefehle**

Assemblerinstruktionen

- **Assemblerinstruktionen (Pseudobefehle):**

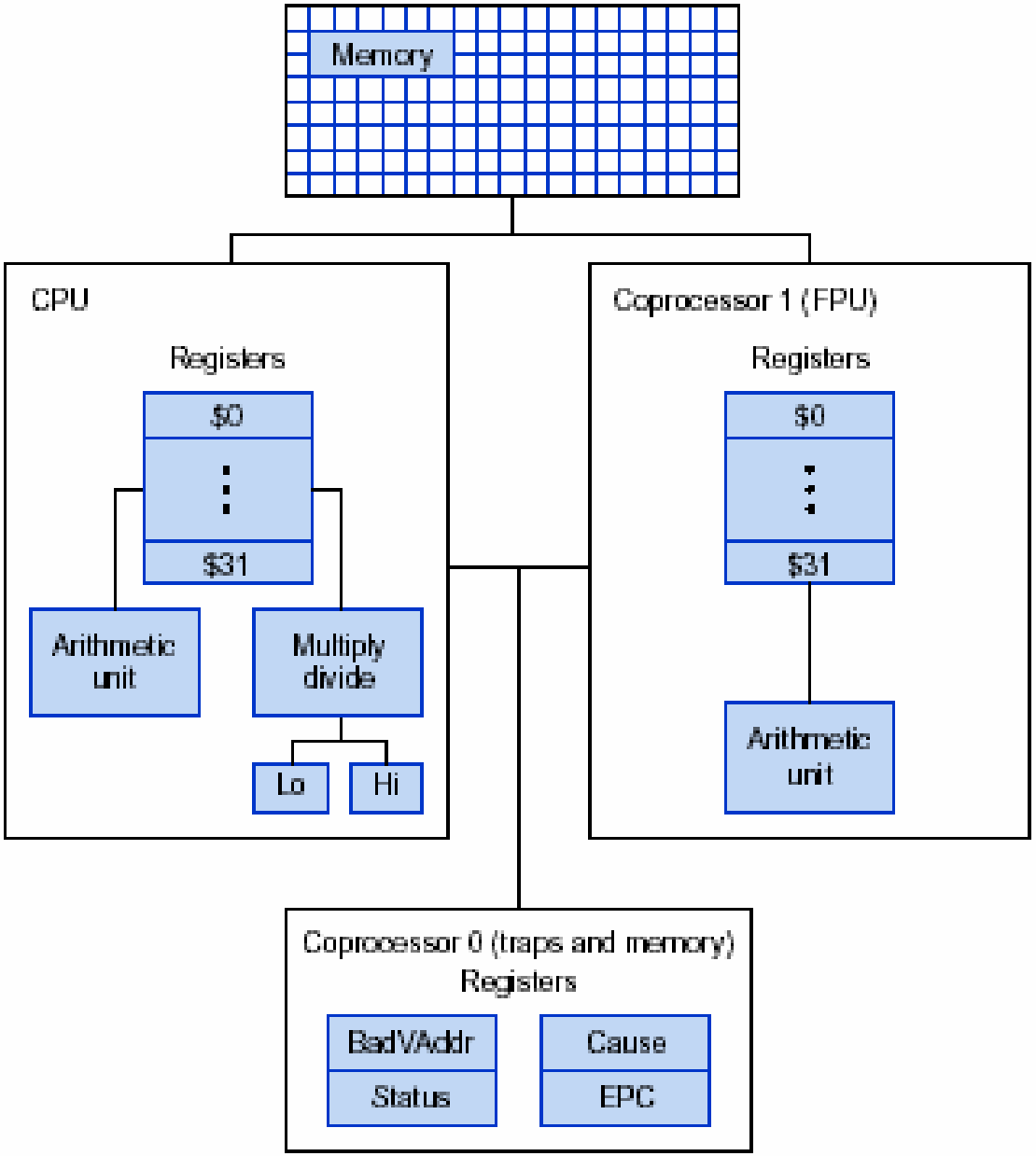
  Anweisungen an den Assembler, die den Übersetzungsvorgang selbst steuern. Sie erzeugen in der Regel keinen Maschinencode, haben aber Auswirkungen auf die Erzeugung der Maschineninstruktionen.

- **Pseudobefehlsarten:**
  - **Programmorganisation**
    - » **Steuerung der Anordnung der Befehle und Daten im Adressraum des Programms**
  - **Gleichsetzungen und symbolische Adressen**
    - » **Verwendung symbolischer Bezeichnungen statt expliziter Angaben**
  - **Definition von Konstanten und Speicherbereichen**
    - » **Deklaration von Datentypen und Reservierung von Speicherbereichen**
  - **Adressierung**
    - » **Steuerung der Art der Adressbildung, z.B. Basisadressierung**

MIPS R2000 Assembler

- ALU Instruktionen
- Datentransfer
- Kontrollinstruktionen
- Ein-/Ausgabe
- Ausnahmen und Unterbrechungen
- Prozeduren
- …

- **SPIM Simulator**

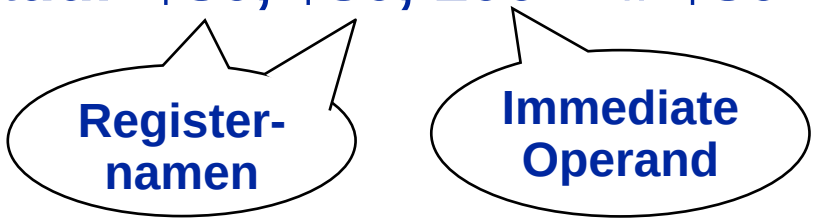


Elementare MIPS-Instruktionen

- **MIPS ist load-store-Architektur, d.h.**
  - **Speicherzugriffe erfolgen nur in Lade-/Speicherbefehlen und**
  - **ALU-Instruktionen operieren nur auf Registern.**

- **ALU-Instruktionen haben ein Drei-Adress-Format:**

```
add  $s0, $s1, $s2     # $s0 <- $s1 + $s2
addi  $s0, $s0, 100    # $s0 <- $s0 + 100
```



- **Geladen und gespeichert werden Bytes (8 Bits), Halbworte (16 Bits) und Worte (32 Bits):**

```
lw     $s5, <speicheradresse>
sb     $s3, <speicheradresse>
```

MIPS Register

32 Register		Benutzungskonvention
0	$zero	Konstante 0
1	$at	reserviert für Assembler
2-3	$v0 - $v1	Ausdrucksauswertung, Funktionswert
4-7	$a0 - $a3	Argumentregister
8-15	$t0 - $t7	temporäre Register
16-23	$s0 - $s7	sichere Register
24-25	$t8 - $t9	temporäre Register
26-27	$k0 - $k1	reserviert für Betriebssystem
28	$gp	Zeiger auf Globale Daten
29	$sp	Stack Pointer
30	$fp	Frame Pointer
31	$ra	Rücksprungadresse

Adressiermodi für Lade-/Speicherbefehle

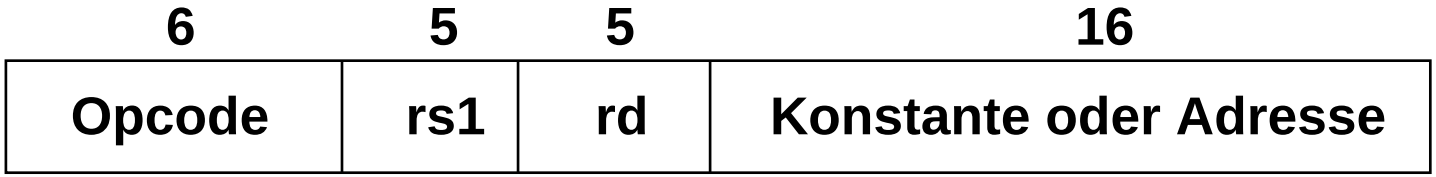
Format	Adressberechnung
(Register)	Basisregisterinhalt
imm	immediate
imm(Register)	immediate + Basisregisterinhalt
Marke	Adresse der Marke
Marke +/- imm	Adresse der Marke +/- immediate
Marke +/- imm(Register)	Adresse der Marke +/- (immediate+Basisregisterinhalt)

Beispiele:

```
lw $s5, 512($t3)      # $s5 <- mem[$t3+512]
sh $s4, 100($s3)      # mem[$s3+100] <- $s4(15:0)
sb $t5, feld + 13($s4)
                      # mem[adr(feld)+$s4+13] <- $t5(7:0)
```

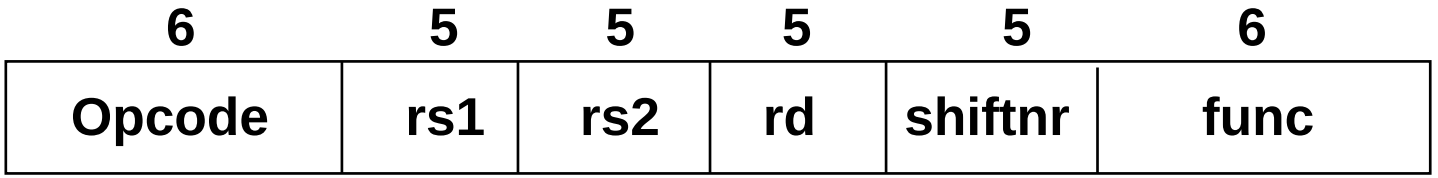
Befehlslayout

- **I-Typ-Befehle**



  - Laden und Speichern von Bytes, Worten und Halbworten
  - ALU-Operationen mit Immediate-Operand (rd <- rs1 op Konstante (immediate))
  - bedingte Sprungbefehle

- **R-Typ-Befehle**



  - Register-Register-ALU-Operationen: rd <- rs1 op rs2

- **J-Typ-Befehle**
  - Sprungbefehle

Befehlskodierungen

Assemblerbefehle werden vom Programmierer mit leicht merkbaren Abkürzungen angegeben, die vom Assembler in Bitfolgen übertragen werden.

Instruktion	Layout	Opcode	rs	rt	rd	shiftnr	fct	/ Adresse
add	R	0	reg	reg	reg	0	20h	
sub	R	0	reg	reg	reg	0	22h	
addi	I	8h	reg	reg				Konstante
lw	I	23h	reg	reg				Adresse
sw	I	2Bh	reg	reg				Adresse

Bsp:

		op	rs	rd	address/ rd shnr fct
lw $t0, 1200($t1)	->	100011	01001	01000	00000 10010 110000
add $t0, $s2, $t0	->	000000	10010	01000	01000 00000 100000
sw $t0, 1200($t1)	->	101011	01001	01000	00000 10010 110000

Logische Operationen

Operationen auf Bits oder Teilen von Worten

- **Shiftbefehle sll (shift left logical), srl (shift right logical)**

```
z.B.    sll $t2, $s0, 4          #  $t2  <-  $s0 ++ 0000  (Linksshift um 4 Bits)
        Befehlslayout:   op      rs      rt      rd     shiftnr   fct
                         000000 00000 10000 01010  00100  000000
```

- **Bitweise logische Verknüpfungen:**
  and, or, nor, andi, ori (and/or immediate)

```
z.B.    and $t0, $t1, $t2        # $t0 <- $t1 & $t2 (bitweise Konjunktion)
        nor $t0, $zero, $t1      # $t0 <- not ($t1)
                                          (Negation durch nor-Verknüpfung mit 0)
        andi $t0, $s0, 1         # Test, ob letztes Bit in Register $s0 gesetzt ist
```

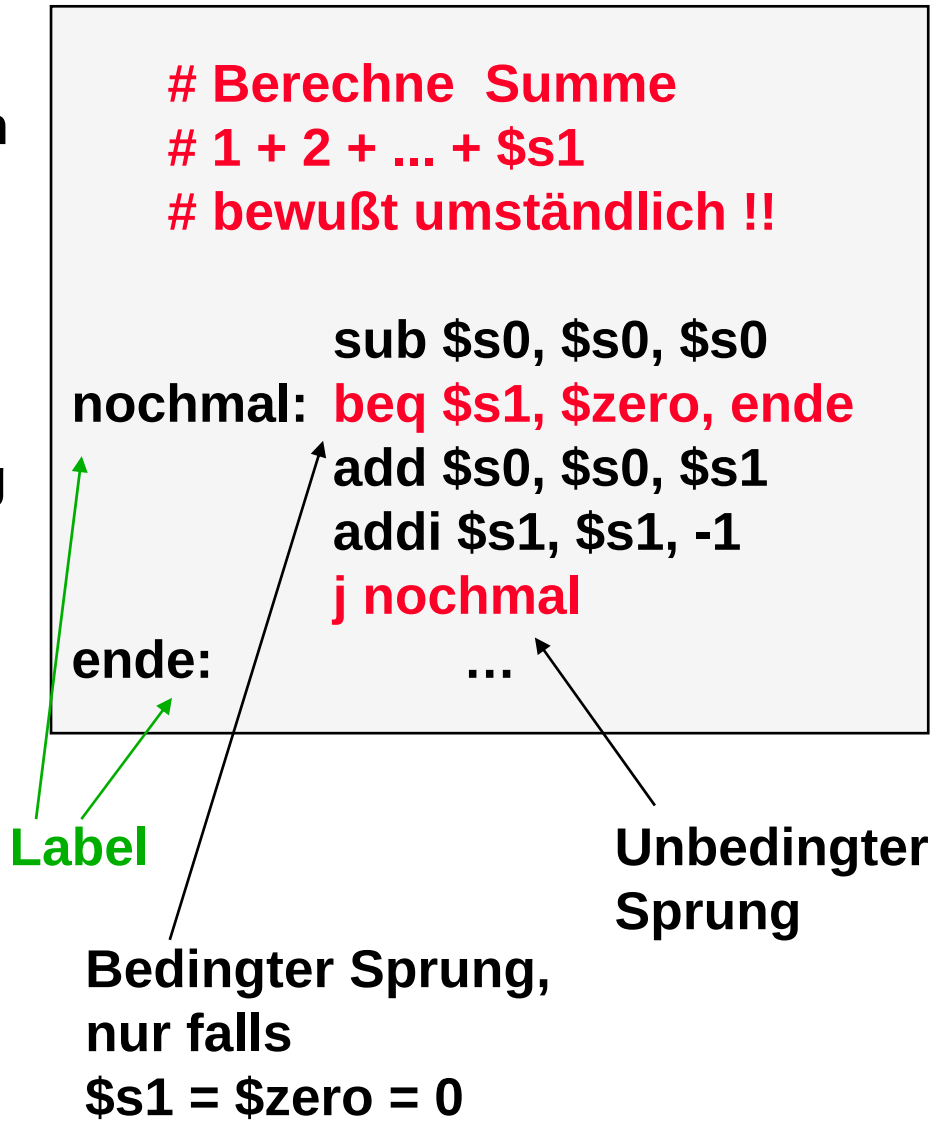
Sprünge

Maschinenbefehle werden in der Reihenfolge ausgeführt, in der sie im Text erscheinen, es sei denn, es wird ein Sprungbefehl bearbeitet.
Das Sprungziel ist die Adresse eines bestimmten Befehls oder eine Sprungmarke (label) .

Es gibt unbedingte Sprünge (der Sprung findet auf jeden Fall statt) und bedingte Sprünge (der Sprung findet nur statt, falls eine bestimmte Bedingung erfüllt ist).

bedingte MIPS-Sprungbefehle:

- beq reg1, reg2, label
  (branch if equal)
- bne reg1, reg2, label
  (branch if not equal)



Bedingte Sprünge

RISC-Prinzip:
wenige und
einfache
Maschinenbefehle

- **MIPS-Maschinenbefehle:**
  - Test auf Gleichheit, Ungleichheit von Registerinhalten: beq, bne
  - Größenvergleiche:
    - » **set on less than: slt reg1, reg2, reg3**
      Bedeutung: if (reg2 < reg3) then reg1=1 else reg1=0
    - » **set on less than immediate: slt reg1, reg2, const**
      Bedeutung: if (reg2 < const) then reg1=1 else reg1=0
- **MIPS-Pseudobefehle:**
  - Test auf Kleiner bzw. Kleinergleich
    - » blt reg1, reg2, label
    - » ble reg1, reg2, label


```
slt $at, reg1, reg2
bne $at, $zero, label
```

```
slt $at, reg2, reg1
beq $at, $zero, label
```

Condition Codes zur Bedingungsentscheidung

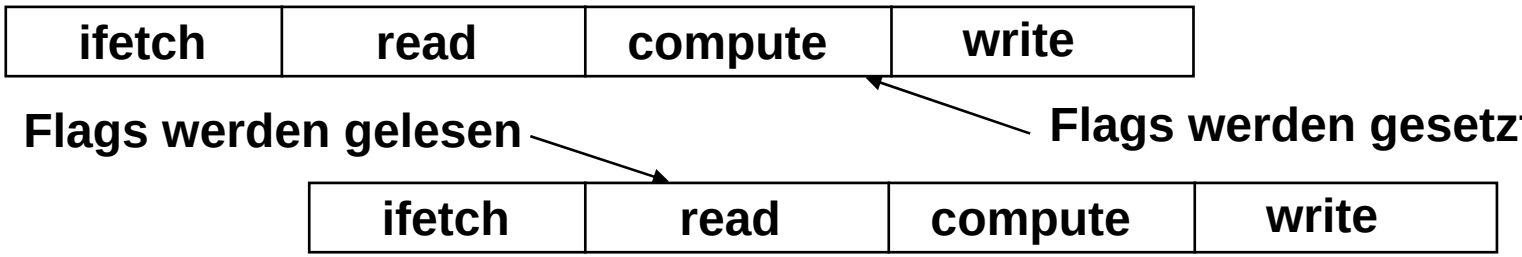
In CISC werden häufig Condition Codes, also ein Flagregister, für die Steuerung bedingter Sprünge verwendet. Der Prozessor setzt als Seiteneffekt einer Operation oder Datenbewegung oder explizit durch Compare- oder Test-Instruktionen Statusbits.

Nachteile:

- **Nicht alle Instruktionen verändern das Flagregister. Das kann zu Konfusionen führen.**

  Beispiel: shift Instruktion setzt das Carry Bit

- **Abhängigkeit zwischen der Instruktion, die das Flagregister setzt, und der, die es testet.**
  Eine überlappende Ausführung, etwa durch Fließbandverarbeitung, könnte das Zwischenschieben einer Instruktion bewirken, die die Flags nicht beeinflusst.



Vorzeichenbehaftete vs. vorzeichenlose Instruktionen

- **Laden von Halbwörtern und Bytes**
  - lh vs lhu (load half word unsigned)
  - lb vs lbu (load byte unsigned)
- **Größenvergleich**
  - slt vs sltu (set less than unsigned)
  - slti vs sltiu (set less than immediate unsigned)
- **Arithmetik**
  - add vs addu (add unsigned)
  - addi vs addiu (add immediate unsigned)
  - sub vs subu (subtract unsigned)

Auffüllen mit Vorzeichen vs. Auffüllen mit Nullen

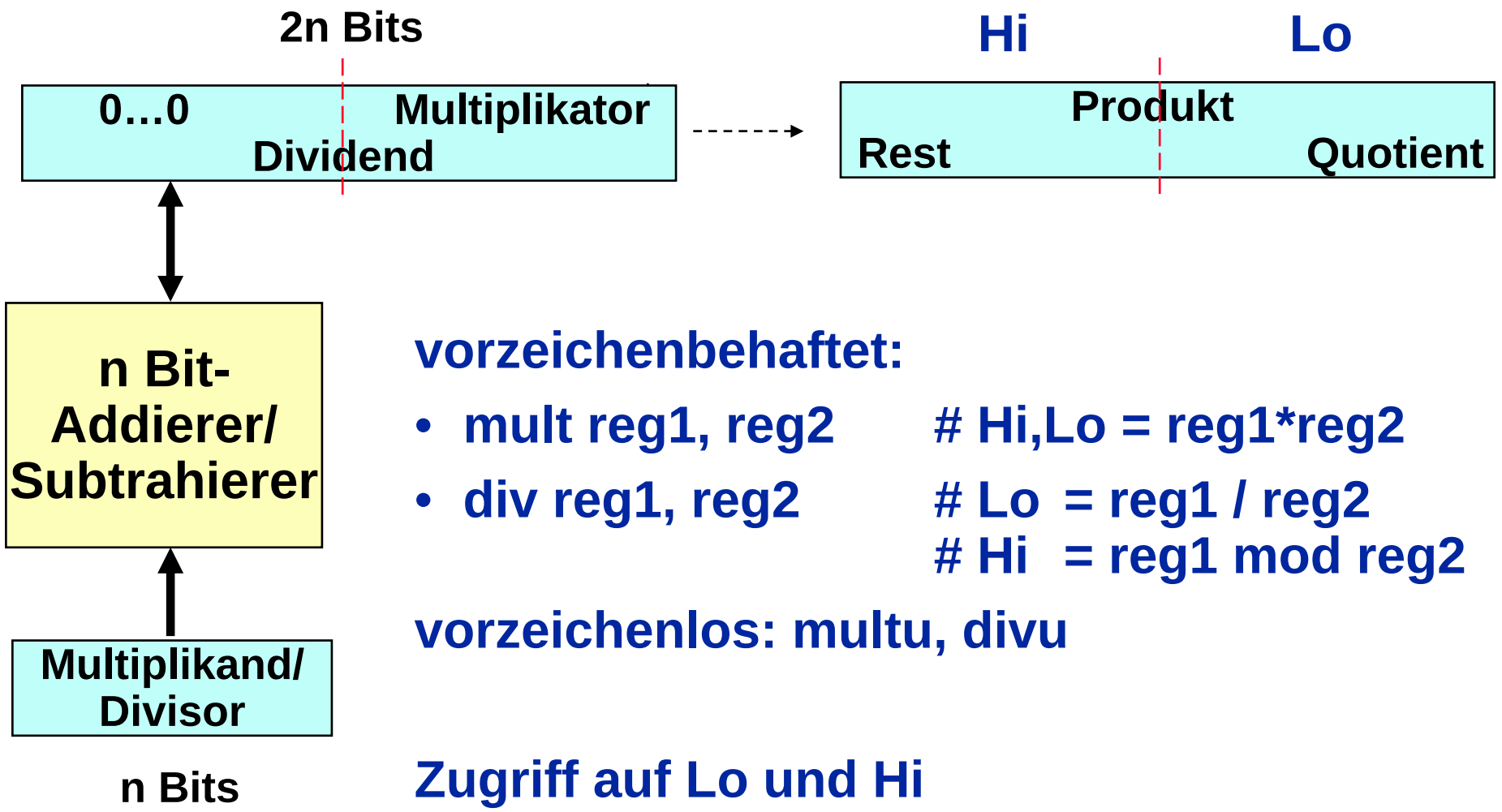
1... < 0… vs. 0… < 1…

Überlauferkennung (mittels Exception) vs. Überlaufignoranz

Simulation eines „Branch on Overflow“-Befehls

```
addu  $t0, $t1, $t2        # $t0 = Summe, ignoriere Überlauf
xor   $t3, $t1, $t2        # Teste, ob Vorzeichen verschieden
slt   $t3, $t3, $zero      # Falls $t3 = 1 sind die Vorzeichen verschieden

bne   $t3, $zero, no_of    # Vorzeichen verschieden -> kein Überlauf

xor   $t3, $t0, $t1        # Vergleiche Summenvorzeichen mit
                           # Summandenvorzeichen
slt   $t3, $t3, $zero      # Falls $t3 = 1 sind die Vorzeichen verschieden

bne   $t3, $zero, overflow # Überlauf, da Summenvorzeichen verschieden
                           # von Summandenvorzeichen
no_of: …
```

Multiplikation und Division



vorzeichenbehaftet:

- **mult reg1, reg2** **# Hi,Lo = reg1*reg2**
- **div reg1, reg2** **# Lo = reg1 / reg2**
  # Hi = reg1 mod reg2

vorzeichenlos: multu, divu

Zugriff auf Lo und Hi

- **mflo reg** **(move from Lo) # reg = Lo**
- **mfhi reg** **(move from Hi) # reg = Hi**

MIPS-Unterprogramme

- strukturierte Programmierung durch Abstraktion und Wiederverwendung
- auf Assemblerebene unterstützt durch
  - spezielle Kontrollbefehle:

» **jal label (jump and link)**
Sprung nach label und Sicherung der Adresse des nachfolgenden Befehls in Register $ra (Rücksprungadresse)

Prozeduraufruf

```
jal  Meine_Prozedur
```

Prozedurdeklaration

```
Meine_Prozedur:
                Befehl1
                  ...
                Befehlk
                jr $ra
```

» **jr reg (jump register)**
unbedingter Sprung zur Befehlsadresse im Register reg
Anwendung: jr $ra für Rücksprung aus Unterprogramm

MIPS-Unterprogramme (2)

– **Konventionen zur Parameterübergabe und zur Wertrückgabe**

» **$a0 - $a3:** 4 Argumentregister zur Parameterübergabe

» **$v0 – $v1:** 2 Rückgaberegister

» **bei geschachtelten oder rekursiven Prozeduren zusätzlich Sicherungen in einem Stacksegment im Speicher**

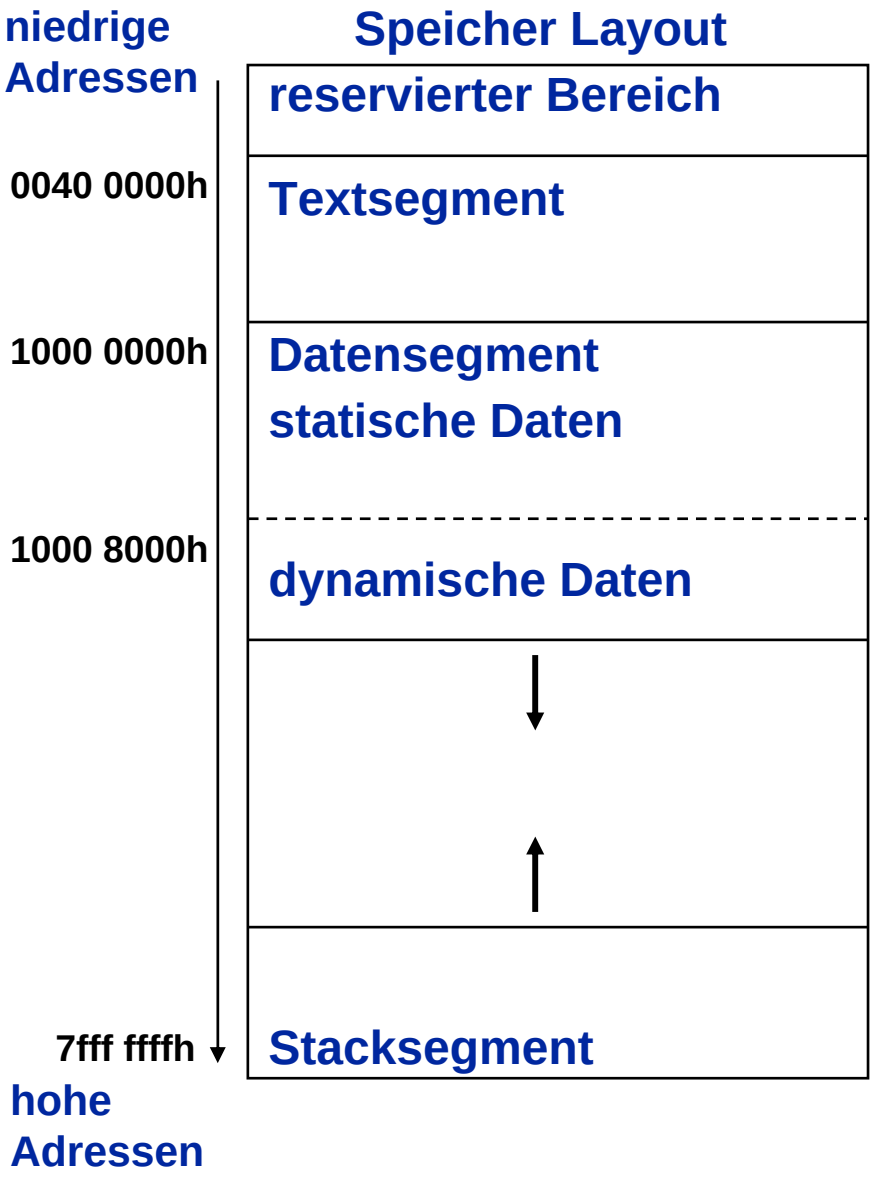
0	$zero	Konstante 0
1	$at	reserviert für Assembler
2-3	$v0 - $v1	Ausdrucksauswertung, Funktionswert
4-7	$a0 - $a3	Argumentregister
8-15	$t0 - $t7	temporäre Register (Sicherung vor Prozeduraufrufen)
16-23	$s0 - $s7	sichere Register (Sicherung in Prozeduren)
24-25	$t8 - $t9	temporäre Register (Sicherung vor Prozeduraufrufen)
26-27	$k0 - $k1	reserviert für Betriebssystem
28	$gp	Zeiger auf Globale Daten
29	$sp	Stack Pointer
30	$fp	Frame Pointer
31	$ra	Rücksprungadresse

Beispiel: Prozedur complexAdd

```
main:   ...
        lw $a0, XReal
        lw $a1, XComp
        lw $a2, YReal
        lw $a3, YComp
        jal complexAdd
        sw   $v0, sumReal
        sw   $v1, sumComp
        ...
```

```
        ...
complexAdd:
        add $v0, $a0, $a2
        add $v1, $a1, $a3
        jr $ra
        ...
```

Konvention für die Speicheraufteilung in MIPS



- **reservierter Bereich**
  - Betriebssystemcode
- **Textsegment**
  - auszuführende Befehlsfolge
  - adressiert über Program Counter (pc) (= Befehlszeiger)
- **Datensegment**
  - statische Daten
    adressiert über „global pointer“ Register $gp
  - dynamische Daten
- **Stacksegment**
  - Sicherung von Registerinhalten bei Prozeduraufrufen
    adressiert über „stack pointer“ Register $sp

Prozeduraufrufe auf Assemblerebene

explizite Sicherung von Daten in Stacksegment

– **Push:**

```
addi $sp, $sp, -12      # Schaffe Platz für drei Registerwerte auf dem Stack
sw $t0, 8($sp)
sw $t1, 4($sp)          # Sichere Register $t0-$t2 auf dem Stack
sw $t2, 0($sp)
```

– **Pop:**

```
lw $t2, 0($sp)
lw $t1, 4($sp)          # Restauriere Register $t0-$t2 vom Stack
lw $t1, 8($sp)
addi $sp, $sp, 12       # Lösche Platz von drei Registerwerten auf dem Stack
```

Prozedurblöcke (Frames)

Vor einem Prozeduraufruf in MIPS:

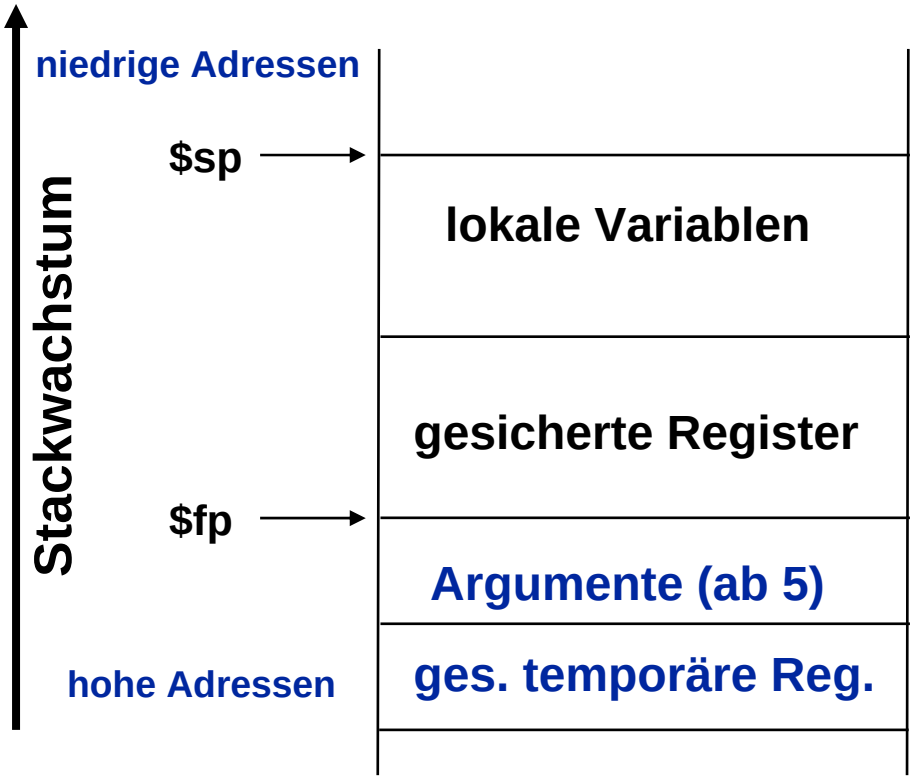
1. Sicherung temporärer Register, die später noch benötigt werden:
   - $t0 - $t9 - $a0 -$a3 - $v0 -$v1
2. Argumentübergabe:
   - die ersten 4 Argumente über Register $a0 -$a3
   - weitere Argumente über den Stack
3. Prozeduraufrufs mittels jal-Instruktion

Aktionen am Beginn einer Prozedur:

1. Stackpointer erniedrigen, um Platz zu reservieren
2. Sicherung aller Register, die vor Rücksprung restauriert werden müssen
   - $s0 - $s7 - $fp - $ra
3. Einrichten des Frame Pointers $fp

Aktionen am Ende einer Prozedur:

1. Rückgabewerte in $v0 - $v1 speichern
2. Restaurierung aller Register, die zu Beginn gesichert wurden
3. Anpassen des Stack Pointers $sp
4. Rücksprung mittels jr $ra



Nach einem Prozeduraufruf:

1. Rückspeicherung vorher gesicherter Register
2. Entfernen der Argumente, die über den Stack übergeben wurden
3. Anpassen des Stackpointers $sp

Beispiel: Rekursive Prozedur

```
fact:   addi   $sp, $sp, -8     # Reserviere Stackplatz für zwei Einträge
        sw     $ra, 4($sp)      # Sichere Rücksprungadresse
        sw     $a0, 0($sp)      # Sichere Argumentregister
        slti   $t0, $a0, 1      # Teste, ob arg < 1
        beq    $t0, $zero, L1   # Falls arg >= 1, goto L1
        addi   $v0, $zero, 1    # return 1
        addi   $sp, $sp, 8      # Entferne Stackelemente
        jr     $ra              # Rücksprung
L1:     addi   $a0, $a0, -1     # arg >= 1, dekrementiere um 1
        jal    fact             # rekursiver Aufruf von fact
        lw     $a0, 0($sp)      # restauriere $a0 von Stack
        lw     $ra, 4($sp)      # restauriere $ra
        addi   $sp, $sp, 8      # Entferne Stackelemente
        mul    $v0, $a0, $v0    # return arg * fact (arg-1)
        jr     $ra              # Rücksprung
```

Struktur eines Assemblerprogramms

```
.text
    # Instruktionssequenz
    .globl  main                # globale Label können aus anderen Dateien
                                # referenziert werden
    main :  ……

    loop : …….

.data
    # Datendeklarationen

    str:  .asciiz “The sum from 0 .. 100 is %d\n”
          .byte 84, 104, 101, 32, 115, 117, 109, 32
```

Assemblerdirektiven haben die Form .directive.

Sie geben Anweisungen an den Assembler.

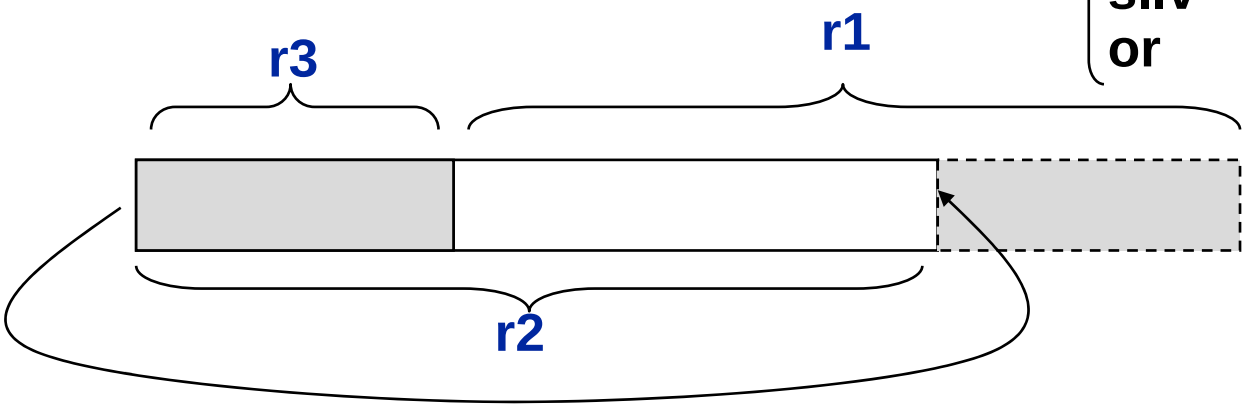
Assembler-Direktiven

- **.align n** Ausrichten des nächsten Datums auf eine Adresse der Form 2^n ^ Zum Beispiel wird mit .align 2 eine Wortgrenze festgelegt. .align 0 schaltet die automatische Ausrichtung von .half, .word, .float, and .double Direktiven bis zur nächsten .data Direktive aus.
- **.ascii str** Der String str wird im Speicher abgelegt, ohne Nullterminierung.
- **.asciiz str** wie .ascii mit Nullterminierung
- **.byte** b1,..., bn Speichere die *n* Werte in aufeinander folgende Speicherzellen, analog **.half, .word, .float, and .double**
- **.data <addr>** Nachfolgende Einträge werden im Datensegment abgelegt. Die Startadresse <addr> ist optional.
- **.extern sym size** Deklaration, dass das Datum mit Label *sym* aus *size* Bytes besteht und global ist. Diese Direktive ermöglicht dem Assembler, das Datum in einem Bereich zu speichern, der effizient durch den globalen Datenzeiger in Register $gp adressierbar ist.
- **.globl sym** Deklariere *sym* als globales Label, das von anderen Dateien aus referenziert werden kann.
- **.space n** Allokiere *n* Leer-Bytes im Datensegment
- **.text <addr>** Beginn des Textsegmentes, in dem in SPIM nur Instruktionen oder Worte stehen können. Die Startadresse <addr> ist optional.

MIPS-Pseudo-Instruktionen

Instruktionen, die der Assembler akzeptiert und in Sequenzen elementarer MIPS-Instruktionen umsetzt, z.B.:

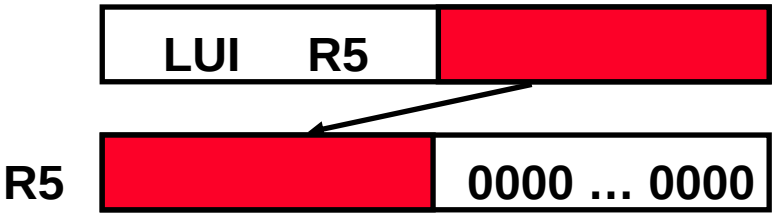
Pseudo-Instruktion	Simulation
– move reg1, reg2	add reg1, $zero, reg2
– not reg1, reg2	nor reg1, $zero, reg2
– li reg, imm (load immediate)	addi reg, $zero, imm
– la reg, addr (load address)	siehe später
– rol r1, r2, r3 (rotate left)	subu $at, $0, r3 srlv $at, r2, $at (shift right logical variable) sllv r1, r2, r3 or r1, r1, $at



32-Bit Konstanten und Adressen

- **Zum Laden von 32-Bit Konstanten steht die Instruktion**

  lui reg, imm (load upper immediate)



  zur Verfügung, mit der die oberen 16 Bits eines Registers gesetzt werden können.

  Z.B. wird die Konstante 0x003d0900 in das Register $s0 durch die folgenden beiden Instruktionen geladen:

```
lui $s0, 61          #  61 = 0x003d
ori $s0, $s0, 2304   # 2304 = 0x0900
```

- **Die Pseudoinstruktion la reg, addr wird entsprechend simuliert.**

Sprungzieladressierung und weite Sprünge

- **bedingte Sprünge beq oder bne**
  -> PC-relative Sprungzieladressierung
  -> Zielbereich: PC+4 +/- 2^{15}

- **unbedingte Sprünge j und jal**
  -> (pseudo-)direkte Adressierung über 26 Bit Wortadresse (= 28 Bit Byteadresse)
  -> Zielbereich: PC[31..28]*2^{28}+[0..2^{28}-1]

- **weite Sprünge durch Kombination von bedingten und unbedingten Sprungbefehlen, z.B.**

```
        bne $s0, $s1, L2          statt          beq $s0, $s1, L1
        j L1
L2:
```

Makros

Ein Makro ist eine Abkürzung eines Textteiles durch ein Schlüsselwort. Jedes spätere Erscheinen dieses Schlüsselwortes wird von dem Assembler vor der Übersetzung automatisch "expandiert", d.h. durch den abgekürzten Text ersetzt. Ein Makro kann auch Parameter haben.

Makrodefinitionen :

```
.text
.macro print_int($arg)
      move $a0, $arg
            # Load macro's parameter
            # ($arg) into $a0
      li $v0, 1
      syscall
            # Operating system call
.end_macro
```

Aufrufe :

```
...

print_int($10)
print_int($7)
print_int ($s0)
```

Expansion:

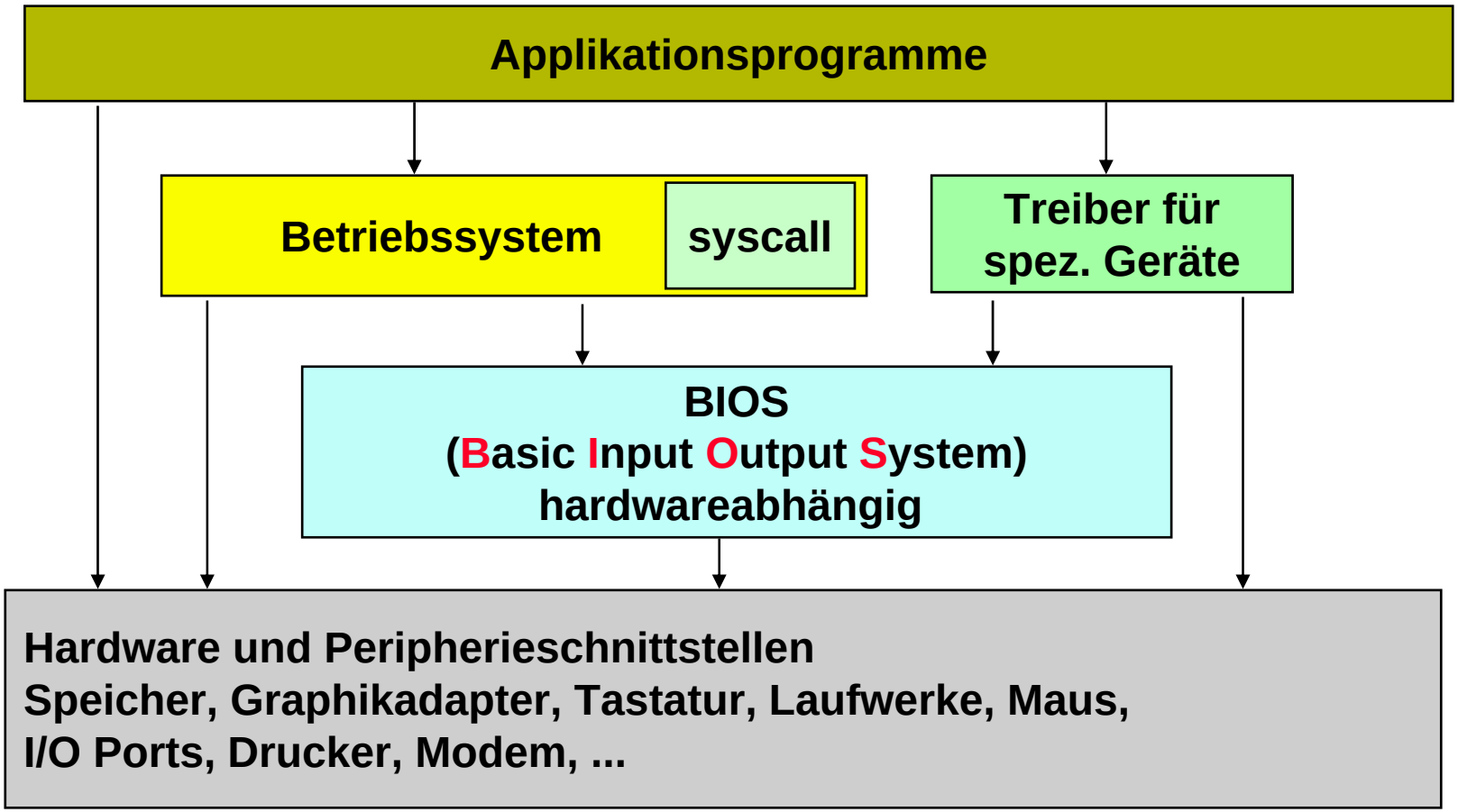
```
move $a0, $7
li $v0, 1
syscall
```

ein Parameter

Achtung: Makros stehen in MIPS nicht zur Verfügung.

Ein-/Ausgabeorganisation

Applikationsprogramme, die direkt auf die Hardware zugreifen, wären viel zu kompliziert (ein Ändern des Graphikmodus erfordert z.B. ca. 30 Ausgabe-anweisungen) und nicht portabel. Zur Hardwareansteuerung werden über den Betriebssystemaufruf syscall viele Funktionen bequem bereitgestellt.



Durch syscall bereitstehende Dienste

Service	System call code	Arguments	Result
print_int	1	$a0 = integer	
print_float	2	$f12 = float	
print_double	3	$f12 = double	
print_string	4	$a0 = string	
read_int	5		integer (in $v0)
read_float	6		float (in $f0)
read_double	7		double (in $f0)
read_string	8	$a0 = buffer, $a1 = length	
sbrk	9	$a0 = amount	address (in $v0)
exit	10		
print_char	11	$a0 = char	
read_char	12		char (in $v0)
open	13	$a0 = filename (string), $a1 = flags, $a2 = mode	file descriptor (in $a0)
read	14	$a0 = file descriptor, $a1 = buffer, $a2 = length	num chars read (in $a0)
write	15	$a0 = file descriptor, $a1 = buffer, $a2 = length	num chars written (in $a0)
close	16	$a0 = file descriptor	
exit2	17	$a0 = result	

Ein Beispielprogramm mit syscall

```
        .text
        .globl  main
main:   li      $t0, 10             # ASCII-Zeichen
                                    # für Returntaste
loop:   li      $v0, 12             # Tastatur-
        syscall                     # eingabe

        beq   $v0, $t0, Fertig      # Returntaste?
        move $a0, $v0
        li      $v0, 11             # Bildschirm-
        syscall                     # ausgabe
        j       loop                # das Ganze
                                    # nochmal
Fertig:
        li $v0, 10                  # Programm
        syscall                     #         beenden
```

Stringverarbeitung

- C Strings sind Bytefolgen von ASCII-Zeichen, die nullterminiert sind, d.h. durch das Nullzeichen abgeschlossen werden.

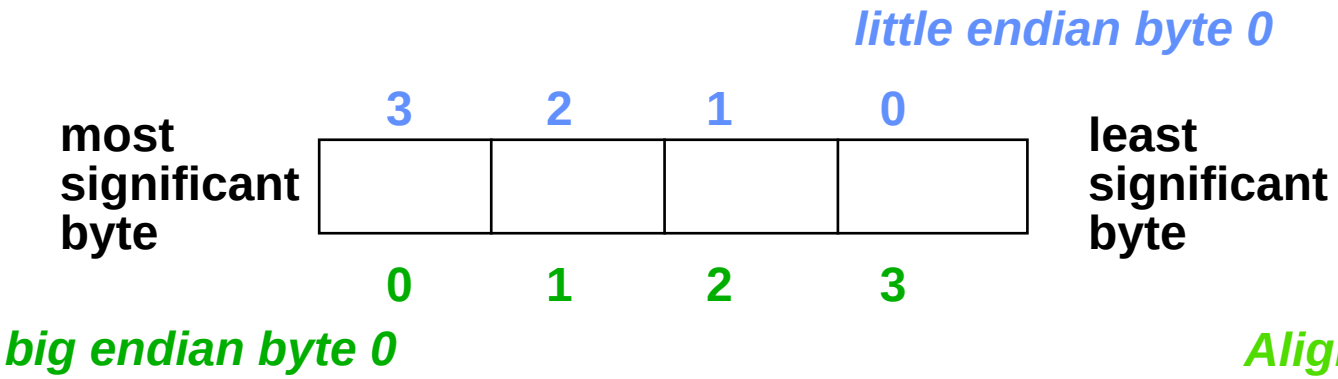
- Wie kann eine Assemblerprozedur entwickelt werden, die einen nullterminierten ASCII-String von einer Adresse source an eine Adresse sink kopiert?

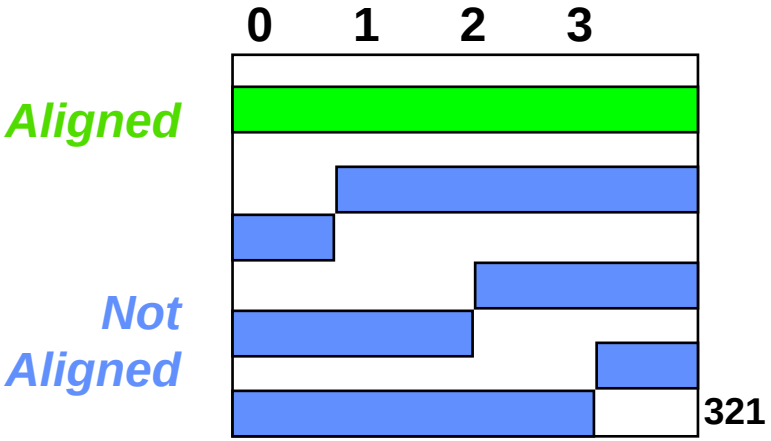
```
            .text
            .globl main

main:       la   $a1, source    # Initialisiere Argumentregister
            la   $a0, sink
            jal  strcpy         # Aufruf der Kopierprozedur
            li   $v0, 10        # Terminiere Programm
            syscall

strcpy:     .....
            jr   $ra            # Rücksprung

            .data
source:     .asciiz  "This is an example string."
sink:       .space 30
```

Speicheradressierung

Zuordnung zwischen Byte und Wortadressen

- **Big Endian:** Adresse des signifikantesten Bytes = Wortadresse (xx00 = Big End of word)
  - IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- **Little Endian:** Adresse des niedrigwertigsten Bytes = Wortadresse (xx00 = Little End of word)
  - Intel 80x86, DEC Vax, DEC Alpha (Windows NT)





Ausrichtung (alignment):

- Objekte werden nur an Adressen gespeichert, die Vielfache ihrer Größe sind.

Adressierungstechniken

- **direkte Adressierung (absolute Adressierung)**
  - **Angabe der absoluten Adresse**
    - » **Vorteil: einfach handhabbar**
    - » **Nachteil: lange Adressen oder kleine Adreßräume**
- **Basisregister-Adressierung**
  - **der Inhalt eines ausgezeichneten Basisregisters wird zu der Adressangabe in einer Operation addiert**
    - » **Vorteil: der adressierbare Speicher wird größer**
- **Indexregister-Adressierung**
  - **technisch wie Basisregister-Adressierung**
  - **Anwendung bei Bearbeitung linearer Listen, Schleifen etc.**
- **indirekte Adressierung**
  - **Speicheradresse wird als Inhalt einer anderen Speicheradresse ermittelt**

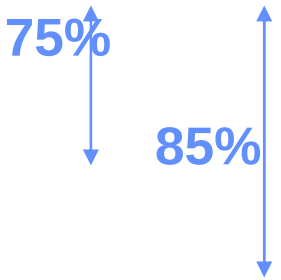
Beispiel: Adressiermodi der VAX

Adressiermodus	Beispiel	Bedeutung
Register	Add R4,R3	R4 ← R4+R3
Immediate	Add R4,#3	R4 ← R4+3
Displacement	Add R4,100(R1)	R4 ← R4+Mem[100+R1]
Register indirect	Add R4,(R1)	R4 ← R4+Mem[R1]
Indexed / Base	Add R3,(R1+R2)	R3 ← R3+Mem[R1+R2]
Direct or absolute	Add R1,(1001)	R1 ← R1+Mem[1001]
Memory indirect	Add R1,@(R3)	R1 ← R1+Mem[Mem[R3]]
Auto-increment	Add R1,(R2)+	R1 ← R1+Mem[R2]; R2 ← R2+d
Auto-decrement	Add R1,–(R2)	R2 ← R2–d; R1 ← R1+Mem[R2]
Scaled	Add R1,100(R2)[R3]	R1 ← R1+Mem[100+R2+R3*d]

Verwendung der Adressiermodi (ohne Register)

Analyse von 3 Programmen auf einer VAX

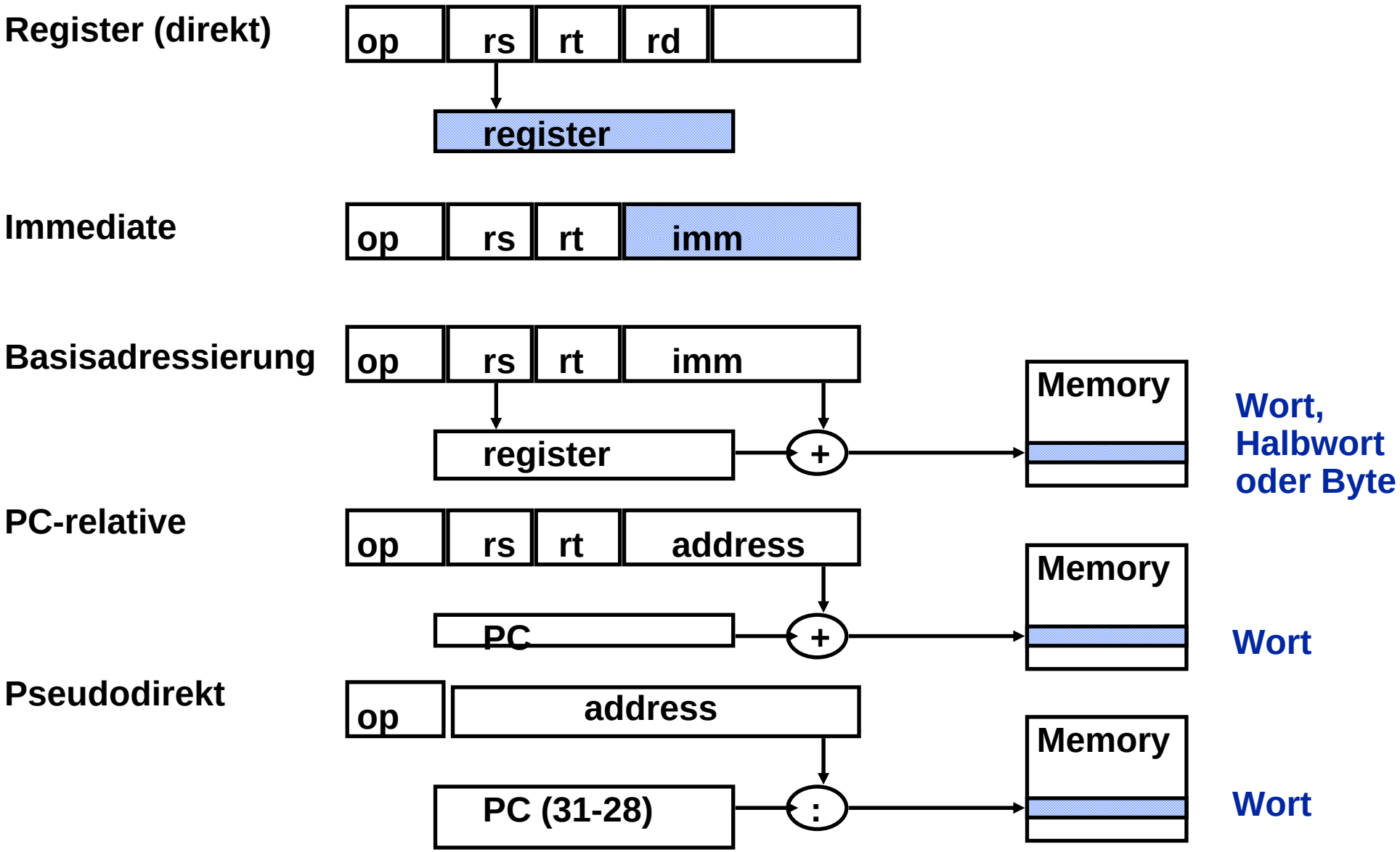
---	**Displacement:**	**42% avg, 32% to 55%**
---	**Immediate:**	**33% avg, 17% to 43%**
---	**Register indirect:**	**13% avg, 3% to 24%**
---	**Scaled:**	**7% avg, 0% to 16%**
---	**Memory indirect:**	**3% avg, 1% to 6%**
---	**Misc:**	**2% avg, 0% to 3%**



75% displacement & immediate

88% displacement, immediate & register indirect

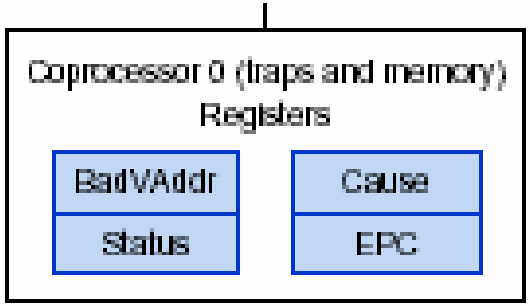
MIPS Adressiermodi/Instruktionsformate



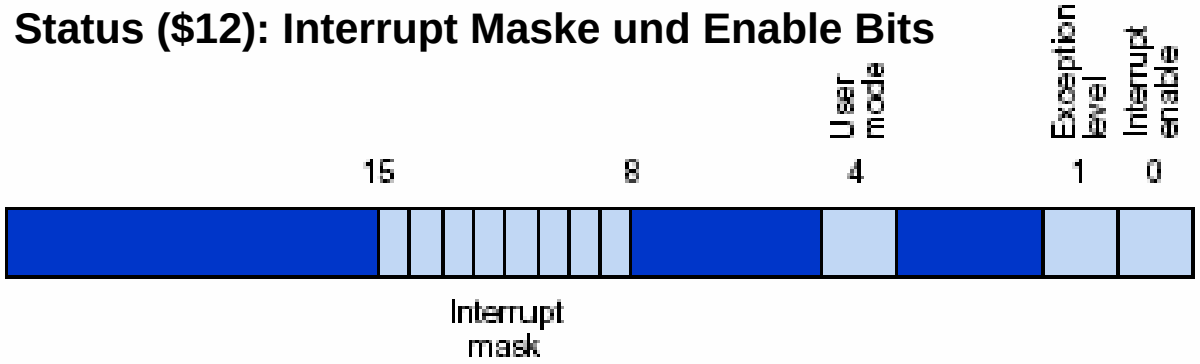
Exceptions und Interrupts

- … sind Ereignisse, die den Kontrollfluss ändern.
  Die Instruktionsverarbeitung wird unterbrochen und die Kontrolle an das Betriebssystem übergeben.
- Exceptions (Ausnahmen) werden vom Prozessor ausgelöst, z.B. bei einem Überlauf, einer unbekannten Operation oder einer Ein-/Ausgabeoperation (syscall).
- Interrupts (Unterbrechungen) kommen von außerhalb des Prozessors, z.B. von Ein-/Ausgabegeräten.
- Verarbeitung von Exceptions
  - Identifizieren des PC-Wertes, bei dem die Exception auftrat.
  - Erkennen der Ursache für die Exception.
  - Übertragen der Kontrolle an das Betriebssystem, das dann geeignete Maßnahmen ergreift.
- Implementierungsmöglichkeiten
  - Mit Statusregister: Es gibt ein spezielles Register, dessen Wert die Ursache einer Exception codiert (Cause Register).
  - Durch Vectored Interrupt: Es gibt für jede mögliche Exception eine festgelegte Speicheradresse, an die bei Eintreten verzweigt wird. Hier sollte dann die Exception-Handling-Routine (Interrupt Vector) stehen.

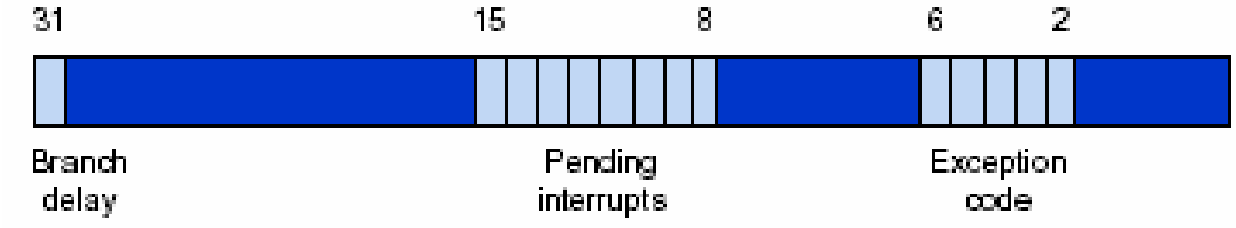
Exceptions in MIPS / SPIM



- **Exception Handler heißt Coprocessor 0.**
- **SPIM unterstützt u.a. die folgenden Register dieser Einheit:**
  - **EPC ($14): Instruktionsadresse, die Exception auslöste bzw. bei der Interrupt auftrat**
  - **BadVAddr ($8): Speicheradresse eines fehlschlagenden Speicherzugriffs**
  - **Status ($12): Interrupt Maske und Enable Bits**



  - **Cause ($13): Exception Typ und zurückgestellte Interrupts**



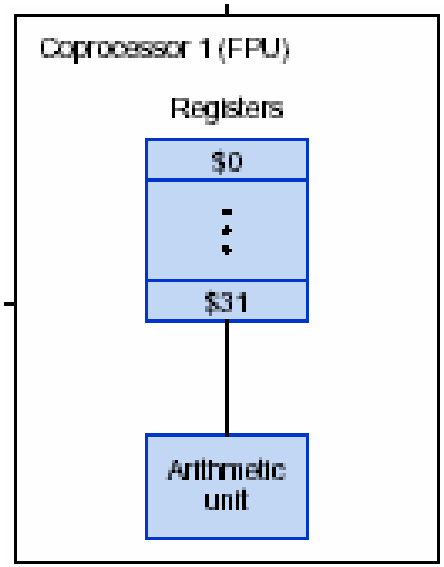
Einige Beispiele für Exception Codes

Nr	Name	Bedeutung
0	INT	Interrupt
4	ADDRL	Adressfehler beim Laden
6	ADDRS	Adressfehler beim Speichern
8	SYSCALL	Befehl syscall
12	OVF	Überlauf

Ein einfacher Exception Handler

```
.ktext 0x80000180
move $k1, $at     # Save $at register
sw   $a0, save0   # Handler not re-entrant
sw   $a1, save1   # and can't use stack to
                  # save $a0, $a1.

mfc0   $k0, $13   # Move Cause into $k0
srl    $a0, $k0, 2  # Extract ExcCode field
andi $a0, $a0, 0x1f
bgtz $a0, done    # Branch if ExcCode is
                  # Int (0)
move   $a0, $k0   # Move Cause into $a0
mfc0   $a1, $14   # Move EPC into $a1
jal print_excp    # Print exception error
                  # message

done:
    mfc0  $k0, $14    # Bump EPC
    addiu $k0, $k0, 4 # Do not reexecute
                      # faulting instruction
    mtc0 $k0, $14     # EPC
    ----------------------------------
    mtc0 $0,   $13    # Clear Cause reg.
    ----------------------------------
    mfc0 $k0, $12     # Fix Status register
    andi $k0, 0xfffd  # Clear EXL bit
    ori    $k0, 0x1   # Enable interrupts
    mtc0 $k0, $12

    lw $a0, save0     # Restore registers
    lw $a1, save1
    move $at, $k1
    eret # Return to EPC

.kdata
save0: .word 0
save1: .word 0
```

Gleitkommaberechnungen



- **Der MIPS-Prozessor verwendet einen mathematischen Koprozessor zur Durchführung von Berechnungen mit Gleitkommazahlen.**

- **Der Koprozessor 1 verfügt wie der Hauptprozessor über 32 32-Bit-Register mit den Namen $f0 bs $f31.**
  Diese können auch zu 16 Paaren $f0, $f2 bis $f30 von 64-Bit-Registern zusammengefasst werden (-> doppelte Genauigkeit).
  Betriebssystem-Eingaben über Register $f0, -Ausgaben über $f12

- **Datendirektiven .float und .double legen die nachfolgenden Werte als 32- oder 64-Bit-Gleitkommazahlen ab. Die Werte können in Exponentialschreibweise, z.B. 123.45e67 oder 0.0003e-9, mit kleinem e oder auch in Festkommaschreibweise mit einem Punkt, also 0.9999 oder 12345.67 notiert werden.**

Gleitkommainstruktionen

- **Lade- und Speicherbefehle**
  - lwc1, swc1 (load/store word coprocessor 1), Formatbeispiel: lwc1 $f1, 100($s2)
  - Pseudobefehle l.s, l.d (load single/double) bzw. s.s, s.d (store single/double) vereinfachen das Laden und Speichern von Gleitkommazahlen
- **Registertransferbefehle**
  - mfc1, mtc1 (move from/to coprocessor 1), Formatbeispiel: mfc1 $s0, $f0
- **arithmetische Befehle**
  - add.s, sub.s, mul.s div.s (single precision operations), Formatbeispiel: add.s $f0, $f1, $f2
  - add.d, sub.d, mul.d, div.d (double precision operations), Formatbeispiel: add.s $f0, $f2, $f4
- **Vergleichsbefehle**
  - c.cop.s, c.cop.d mit cop $\in$ {lt, le, eq}, Formatbeispiel: c.lt.d $f2, $f4
    -> Condition code flag wird gesetzt und mit bedingten Sprungbefehlen abgefragt
- **bedingte Sprungbefehle**
  - bclt, bclf PC-relativer Sprung, falls CC true/false, Formatbeispiel: bclf loop

Beispiel: Hornerschema

$$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x^1 + a_0 = \underbrace{(((((a_n)x + a_{n-1})x + a_{n-2})x + \ldots)x + a_1)x + a_0}_{\text{iterative Berechnung}}$$

```
.data
Koeffs:  .double 1., 5.,  -2.  # Testdaten
last:    .double -24.


.text
.globl main
main:    la $a0, Koeffs
         la $a1, last
         li $v0, 7             # read double
         syscall
         jal poly
         li $v0, 3             # print double
         syscall
         li $v0, 10            # exit
         syscall
```

```
.text
poly: addiu $sp, $sp, -8
                       # Platz für $f10/11
      s.d    $f10, 8($sp) # $f10/11 sichern
      sub.d  $f12. $f12, $f12  # $f12/13 <- 0
      slt    $t0, $a1, $a0
      bne    $t0, $0, exit       # Adressfehler
loop: l.d    $f10, 0($a0)        # $f10/11 <- ai
      mul.d  $f12, $f12,$f0      # $f12/13 * x
      add.d  $f12, $f12, $f10    # --- + ai
      addiu  $a0, $a0, 8         # i <- i+1
      ble    $a0, $a1, loop
exit: l.d    $f10, 8($sp)        # $f10/11
      addiu  $sp, $sp, 8         # restaurieren
      jr $ra                     # Rücksprung
```