Technische Informatik

3 - Assembler

© Lothar Thiele

Computer Engineering and Networks Laboratory

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Assemblerprogrammierung

- In diesem Kapitel werden nur einige wenige ***Grundprinzipien*** erläutert:
  - Bedingte Verzweigungen
  - Funktionsaufrufe
  - Unterbrechungen
- Eine detaillierte Beschreibung findet sich im Buch zur Vorlesung, der beigefügten CD (mit Simulator SPIM) sowie den Übungsunterlagen.
- Weiterhin wird ein ***anderer Instruktionssatz*** kurz beschrieben, der IA-32 Instruktionssatz.

Assemblerprogrammierung

Übersetzung (Wiederholung)

- Das Kapitel 2 der Vorlesung setzt sich mit der Maschinensprache und der Assemblersprache auseinander.
- Die Instruktionen und die zugehörige Maschinensprache definieren die Hardware-Software Schnittstelle einer Rechnerarchitektur.
- Die «Instruction Set Architecture» ist demzufolge der Teil einer Rechnerarchitektur, der mit ihrer Programmierung zusammenhängt:
  - Datenformate,
  - Instruktionen und Adressierungsarten,
  - Register und Speicherarchitektur,
  - Unterbrechungs- und Ausnahmebehandlung,
  - Ein- und Ausgabe.

High-level language program (in C)

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

Compiler

Assembly language program (for MIPS)

```
swap:
      muli $2, $5,4
      add  $2, $4,$2
      lw   $15, 0($2)
      lw   $16, 4($2)
      sw   $16, 0($2)
      sw   $15, 4($2)
      jr   $31
```

Assembler

Binary machine language program (for MIPS)

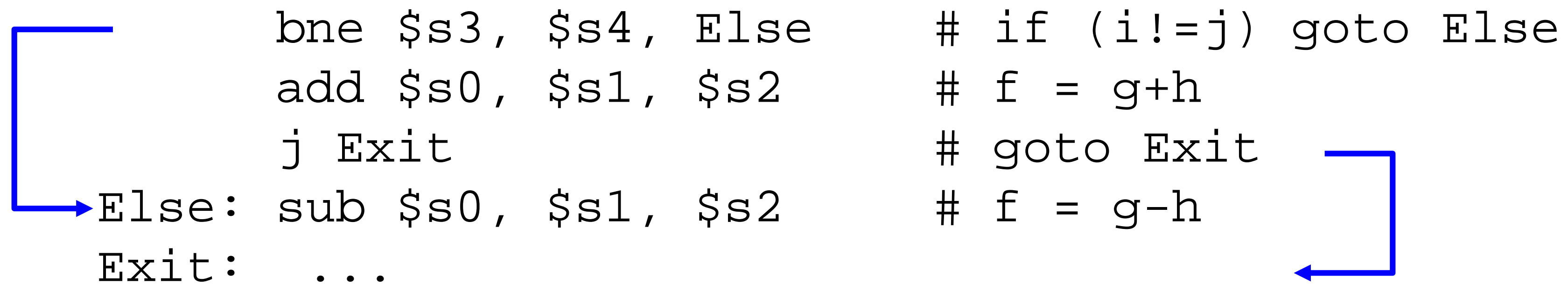
```
00000000101000010000000000011000
00000000000110000001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

Bedingte Verzweigungen

Beispiel:

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

- *Annahmen*: `f,g,h,i,j` sind in `$s0,$s1,$s2,$s3,$s4.`
- *Assemblerprogramm*:



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

Bedingte Verzweigungen

Beispiel:

```
while (save[i] == k) i = i + 1;
```

- *Annahmen*: `i` und `k` sind in `$s3` und `$s5`. Feld `save` startet bei Adresse `$s6`.
- *Assemblerprogramm*:

```
Loop: sll $t1, $s3, 2      # $t1 = 4 * i
      add $t1, $t1, $s6
      lw $t0, 0($t1)
      bne $t0, $s5, Exit
      addi $s3, $s3, 1
      j Loop
Exit: …
```

Array gegenüber Pointer

Pseudoinstruktion (siehe Seite 3-36)

C	MIPS Assembler

in \$a0 … in \$a1
`clear1(int array[], int size)`
`{`
`    int i;`
`    for (i = 0; i < size; i += 1)`
`        array[i] = 0;`
`}`
	`       move $t0,$zero        # i=0`
	`loop1:sll $t1,$t0,2        # $t1=i*4`
	`       add $t2,$a0,$t1       # $t2=&array[i]`
	`       sw $zero,0($t2)       # array[i]=0`
	`       addi $t0,$t0,1        # i=i+1`
	`       slt $t3,$t0,$a1       # $t3=(i<size)`
	`       bne $t3,$zero,loop1 # if() go to loop1`

in \$a0 … in \$a1
`clear2(int *array, int size)`
`{`
`    int *p;`
`    for (p = &array[0]; p < &array[size]; p = p + 1)`
`        *p = 0;`
`}`
	`       move $t0,$a0          # p=&array[0]`
	`       sll $t1,$a1,2         # $t1=size*4`
	`       add $t2,$a0,$t1       # $t2=&array[size]`
	`loop2:sw $zero,0($t0)       # Memory[p]=0`
	`       addi $t0,$t0,4        # p=p+4`
	`       slt $t3,$t0,$t2       # $t3=(p<&array[size])`
	`       bne $t3,$zero,loop2 # if() go to loop2`

Assembler Syntax

Zusätzlich zu Instruktionen werden in einem Assemblerprogramm weitere Informationen benötigt, zum Beispiel ***lokale und globale Marken*** (label), ***Daten***, ***Makrodefinitionen*** sowie Anweisungen zum ***Speicherlayout***. Beispiele:

`.data` nachfolgende Einträge werden im Datensegment gespeichert

`.align n` nachfolgende Daten werden im Speicher auf 2^n Byte Grenzen ausgerichtet; align 2 richtet also das nächste Datum auf eine Wortgrenze aus

`.byte b1,…,bn / .half h1,…,hn / .word w1,….,wn / .ascii str`
Daten im Byte, Halbwort, Word oder Textformat

`.text` nachfolgende Einträge werden im (Programm)-Textsegment gespeichert

`.globl sym` die Marke `sym` ist global und kann von anderen Files referenziert werden

`# text` Kommentar

Beispiel Sortieralgorithmus

Pseudocode

```
for (i=0, i<11, i=i+1) {
  for (j=i+1, j<12, j=j+1) {
    if (nums[j] < nums[i]) {
      nums[j] <-> nums[i];
    }
  }
}
```

MIPS Assembler

```
# Sort some numbers.
        .data
nums:   .word 1, 6, 9, 8, 2, 3, 8, 8, 1, 3, 7, 4
        .text
        .globl main
main:   addi $t0 $zero 0       # init i=0
        addi $t6 $zero 11      # init loop bound outer
        addi $t7 $zero 12      # init loop bound inner
outer:  sll $t4 $t0 2          # outer: convert i
        addi $t1 $t0 1         # init counter j=i+1
inner:  sll $t5 $t1 2          # inner: convert j
        lw $t2 nums($t4)       # load nums[i]
        lw $t3 nums($t5)       # load nums[j]
        bge $t3 $t2 noswap     # compare them
        sw $t2 nums($t5)       # store nums[j] in nums[i]
        sw $t3 nums($t4)       # store nums[i] in nums[j]
noswap: addi $t1 $t1 1         # noswap: increment j=j+1
        bne $t1 $t7 inner      # loop to inner if j =! 12
        addi $t0 $t0 1         # increment i=i+1
        bne $t0 $t6 outer      # loop to outer if i =! 11
        …
```

Funktionen

Funktionen

- Die folgenden ***Konventionen*** beschreiben die Verwendung von Registern und Kellerspeichern in Zusammenhang mit ***Unterprogrammaufrufen*** (Funktionsaufrufen). Sie werden üblicherweise befolgt, um Programmteile getrennt übersetzen zu können.
- Der ***Kontext*** eines Unterprogramms kann
  - *Argumente* enthalten, die dem Unterprogramm mitgeteilt wurden,
  - kann *Registerinhalte* enthalten, die vom Unterprogramm nicht geändert werden dürfen und
  - kann *lokale Variablen* des Unterprogramms enthalten.

Funktionen

Der ***Kontext*** einer Funktion wird im Hauptspeicher abgelegt.

- Der Kontext wird im Hauptspeicher als Aktivierungsrahmen (*‚activation record‘* oder *‚procedure frame‘)* gespeichert.
- Bei *verschachtelten Funktionsaufrufen* werden die Kontexte in Form eines Stacks (Kellerspeicher) verwaltet.
- Die Verwaltung muss im Assemblerprogramm implementiert werden.
- Beginn und Ende eines Aktivierungsrahmens werden durch einen *Stackpointer* (Register `$sp`) und einen *Framepointer* (Register `$fp`) markiert.

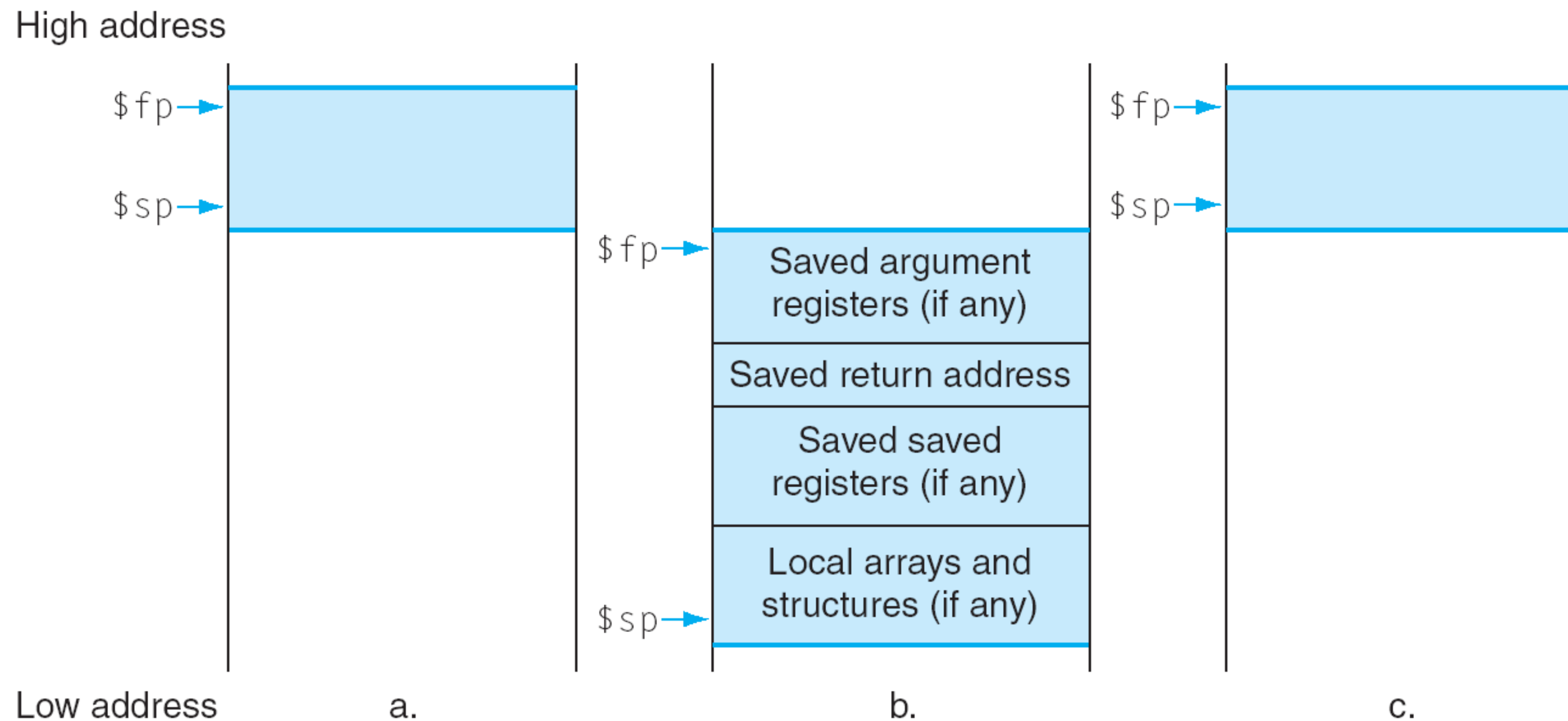
Sichere und temporäre Register

- ***Ziel 1:*** Der Funktionsaufruf soll für das aufrufende Programm möglichst transparent sein: möglichst viele Registerwerte bleiben über den Funktionsaufruf erhalten.
- ***Ziel 2:*** Möglichst wenige Registerwerte sollten im Hautspeicher abgelegt werden müssen.
- ***Vorgehensweise:*** Teile die Register in sichere (Werte bleiben erhalten) und temporäre Register auf.

Preserved	Not preserved
Saved registers: `$s0–$s7`	Temporary registers: `$t0–$t9`
Stack pointer register: `$sp`, `$fp`	Argument registers: `$a0–$a3`
Return address register: `$ra`	Return value registers: `$v0–$v1`
Stack above the stack pointer	Stack below the stack pointer
Global memory pointer: $gp	

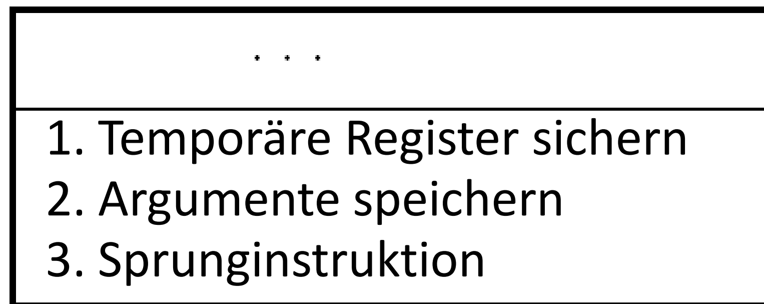
Ablauf eines Funktionsaufrufes

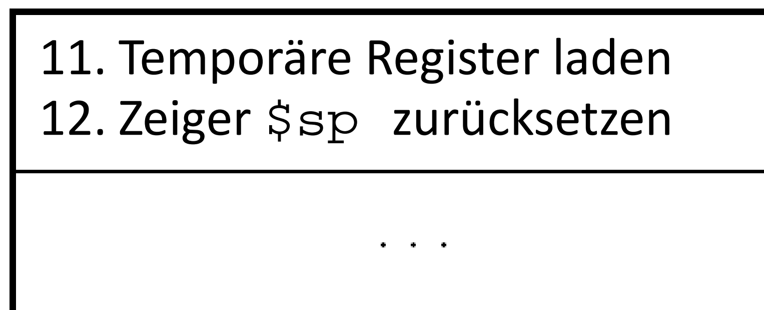
- ***Speicheranordnung***

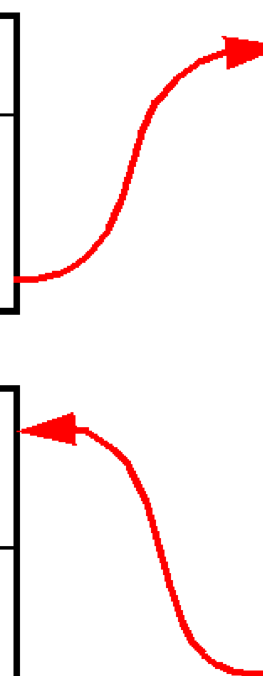


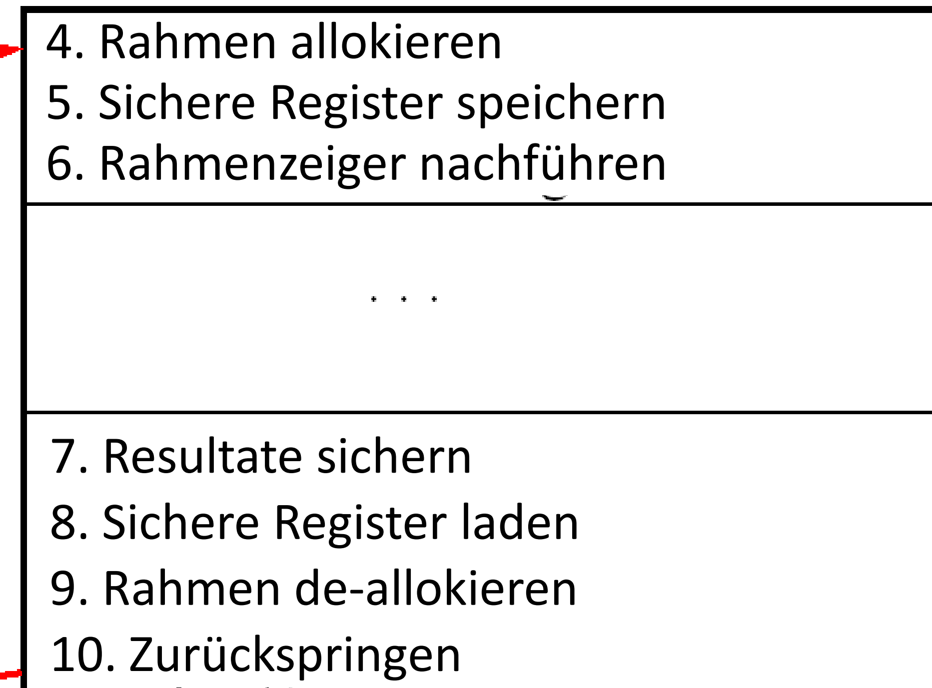
Ablauf eines Funktionsaufrufes

- ***Übersicht Ablauf***









Ablauf eines Funktionsaufrufes

Aufrufendes Programm:

1. *Temporäre Register sichern*: Falls das aufrufende Programm Register aus `$a0–$a3, $t0–$t9, $v0–$v1` benutzt, muss es die Inhalte vor dem Aufruf im eigenen Rahmen sichern.
2. *Argumente speichern*: Die ersten 4 Argumente werden in `$a0` bis `$a3` gespeichert. Alle restlichen Argumente werden am eigenen Rahmen angehängt. Der Zeiger `$sp` wird in diesem Fall angepasst, d.h. wenn mehr als 4 Argumente übergeben werden.
3. *Sprunginstruktion*: Führe die Instruktion `jal` aus, die zur ersten Instruktion des Unterprogramms springt und die Rücksprungadresse in Register `$ra` speichert.

Beginn Unterprogramm:

4. *Rahmen allokieren*: Allokiere Speicher, in dem vom Zeiger `$sp` die notwendige Grösse für den eigenen Rahmen abgezogen wird.
5. *Sichere Register speichern*: Das gerufene Programm muss den Inhalt der Register `$fp, $ra, $s0–$s7` im eigenen Rahmen speichern, falls sie im Unterprogramm verändert werden.
6. *Rahmenzeiger nachführen*: Der Rahmenzeiger `$fp` wird auf den Beginn des eigenen Rahmens gesetzt.

Ablauf eines Funktionsaufrufes

Ende Unterprogramm:

7. *Resultate sichern*: Eventuelle Resultate werden in `$v0` und `$v1` gespeichert.
8. *Sichere Register laden*: Die in Schritt 5 gespeicherten Inhalte der sicheren Register werden in die Register zurückgespeichert.
9. *Rahmen de-allokieren*: Zum Zeiger `$sp` wird die Grösse des eigenen Rahmens addiert. Er wird also auf den Wert gesetzt, den er zum Ende des 3. Schrittes hatte.
10. *Zurückspringen*: Kehre zum aufrufenden Programm durch Sprung zur Adresse in `$ra` zurück.

Aufrufendes Programm:

11. *Temporäre Register laden*: Die in Schritt 1 gespeicherten Inhalte der temporären Register werden in die Register zurückgespeichert.
12. *Zeiger* `$sp` *zurücksetzen*: Die in Schritt 2 vorgenommene Veränderung des Zeigers `$sp` (Speichern zusätzlicher Funktionsargumente) wird rückgängig gemacht.

Beispiel Funktionsaufruf

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

```
swap: addi $sp,$sp,-8 # 4: zwei Register werden gesichert
      sw $fp,4($sp)   # 5: $fp wird in swap geändert
      sw $s0,0($sp)   # 5: $s0 wird in swap geändert
      addi $fp,$sp,4  # 6: $fp zeigt nun auf Rahmenanfang
      sll $t6,$a1,2   #    $t6 = k * 4
      add $t6,$a0,$t6 #    $t6 = v + (k * 4)
      lw $t7,0($t6)   #    temp = v[k]
      lw $s0,4($t6)   #    $s0 = v[k+1]
      sw $t7,4($t6)   #    v[k+1] = temp
      sw $s0,0($t6)   #    v[k] = $s0
      addi $sp,$fp,-4 # 8: Vorbereitung für rückspeichern
      lw $fp,4($sp)   # 8: Rückspeichern von $fp
      lw $s0,0($sp)   # 8: Rückspeichern von $s0
      addi $sp,$sp,8  # 9: Rahmen de-allozieren
      jr $ra          # 10: Zurückspringen
```

Funktionsaufruf

- *Variante:* Zuweilen wird auf die Verwendung des Framepointers `$fp` verzichtet und lediglich der Stackpointer `$sp` verwendet. Der Stackpointer muss dann beim Verlassen des Unterprogramms wieder auf den ursprünglichen Wert gesetzt werden.
- *Beispiel* ohne Framepointer:

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

Funktionsaufruf

Saving registers

```
sort:   addi    $sp,$sp, -20       # make room on stack for 5 registers
        sw      $ra, 16($sp)       # save $ra on stack
        sw      $s3,12($sp)        # save $s3 on stack
        sw      $s2, 8($sp)        # save $s2 on stack
        sw      $s1, 4($sp)        # save $s1 on stack
        sw      $s0, 0($sp)        # save $s0 on stack
```

Procedure Body (auf der nächsten Seite …)

Restoring registers

```
exit1:  lw      $s0, 0($sp)        # restore $s0 from stack
        lw      $s1, 4($sp)        # restore $s1 from stack
        lw      $s2, 8($sp)        # restore $s2 from stack
        lw      $s3,12($sp)        # restore $s3 from stack
        lw      $ra,16($sp)        # restore $ra from stack
        addi    $sp,$sp, 20        # restore stack pointer
```

Procedure return

```
        jr      $ra                # return to calling routine
```

Funktionsaufruf

Procedure body				
Move parameters		move	$s2, $a0	# copy parameter $a0 into $s2 (save $a0)
		move	$s3, $a1	# copy parameter $a1 into $s3 (save $a1)
Outer loop		move	$s0, $zero	# i = 0
	for1tst:	slt	$t0, $s0, $s3	# reg $t0 = 0 if $s0 ≥ $s3 (i ≥ n)
		beq	$t0, $zero, exit1	# go to exit1 if $s0 ≥ $s3 (i ≥ n)
Inner loop		addi	$s1, $s0, -1	# j = i - 1
	for2tst:	slti	$t0, $s1, 0	# reg $t0 = 1 if $s1 < 0 (j < 0)
		bne	$t0, $zero, exit2	# go to exit2 if $s1 < 0 (j < 0)
		sll	$t1, $s1, 2	# reg $t1 = j * 4
		add	$t2, $s2, $t1	# reg $t2 = v + (j * 4)
		lw	$t3, 0($t2)	# reg $t3 = v[j]
		lw	$t4, 4($t2)	# reg $t4 = v[j + 1]
		slt	$t0, $t4, $t3	# reg $t0 = 0 if $t4 ≥ $t3
		beq	$t0, $zero, exit2	# go to exit2 if $t4 ≥ $t3
Pass parameters and call		move	$a0, $s2	# 1st parameter of swap is v (old $a0)
		move	$a1, $s1	# 2nd parameter of swap is j
		jal	swap	# swap code shown in Figure 2.34
Inner loop		addi	$s1, $s1, -1	# j -= 1
		j	for2tst	# jump to test of inner loop
Outer loop	exit2:	addi	$s0, $s0, 1	# i += 1
		j	for1tst	# jump to test of outer loop

Typisches Speicher Layout

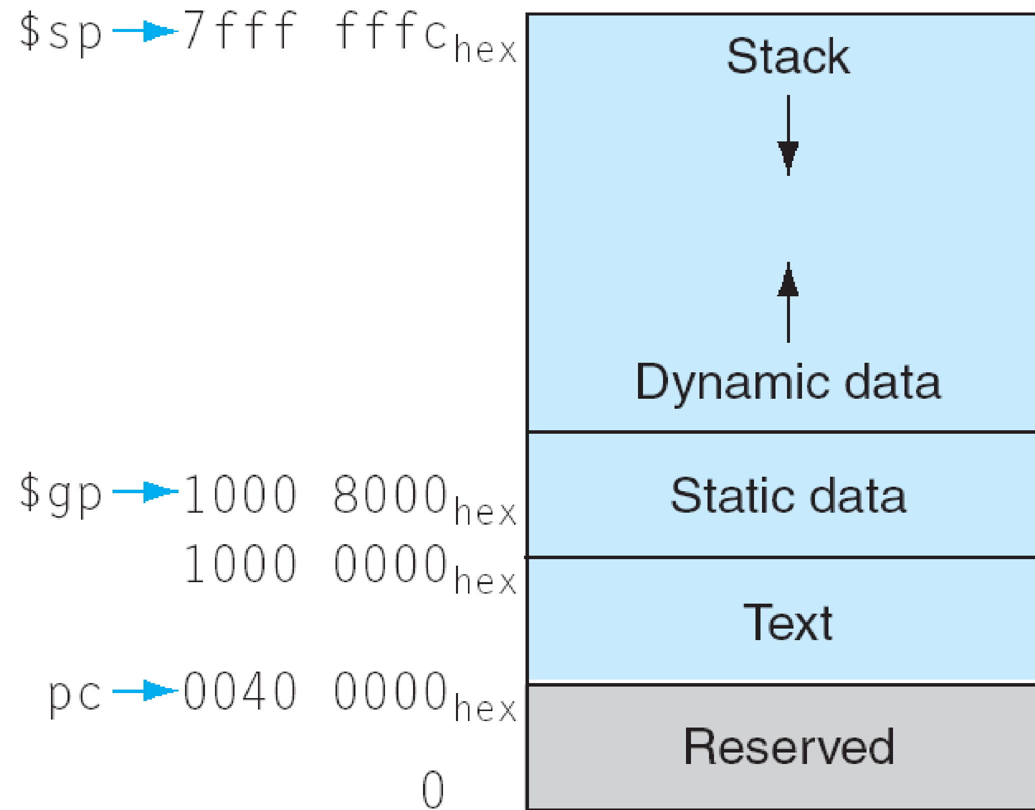


FIGURE 2.17 The MIPS memory allocation for program and data. These addresses are only a software convention, and not part of the MIPS architecture. Starting top down, the stack pointer is initialized to 7fff $fffc_{hex}$ and grows down toward the data segment. At the other end, the program code ("text") starts at 0040 0000_{hex}. The static data starts at 1000 0000_{hex}. Dynamic data, allocated by malloc in C and via new in Java, is next and grows up toward the stack in an area called the heap. The global pointer, $gp, is set to an address to make it easy to access data. It is initialized to 1000 8000_{hex} so that it can access from 1000 0000_{hex} to 1000 $ffff_{hex}$ using the positive and negative 16-bit offsets from $gp

Typisches Speicherlayout

- ***Text:*** Bereich für das auszuführende Maschinenprogramm. Im Assemblerprogramm wird das Programm mit `.text` markiert.
- ***Static Data:*** Bereich für Daten, die mit dem Programm in den Speicher geladen werden. Im Assemblerprogramm werden diese Daten mit `.data` markiert. Das Register `$gp` wird so initialisiert, dass es auf die Mitte des Bereiches zeigt, in dem statische Daten gespeichert werden können.
- ***Dynamic Data:*** Zum Beispiel in C werden Speicherbereiche während der Laufzeit allokiert und de-allokiert. Der Aufruf `malloc()`allokiert Speicherplatz und liefert einen Zeiger darauf zurück, der Aufruf `free()`gibt den Speicherbereich wieder frei, auf den der übergebene Zeiger zeigt.
- ***Stack:*** Kontexte der Funktionen, die als Aktivierungsrahmen in einem Stack (Kellerspeicher) verwaltet werden.

Unterbrechungen und Ausnahmen

Unterbrechungen

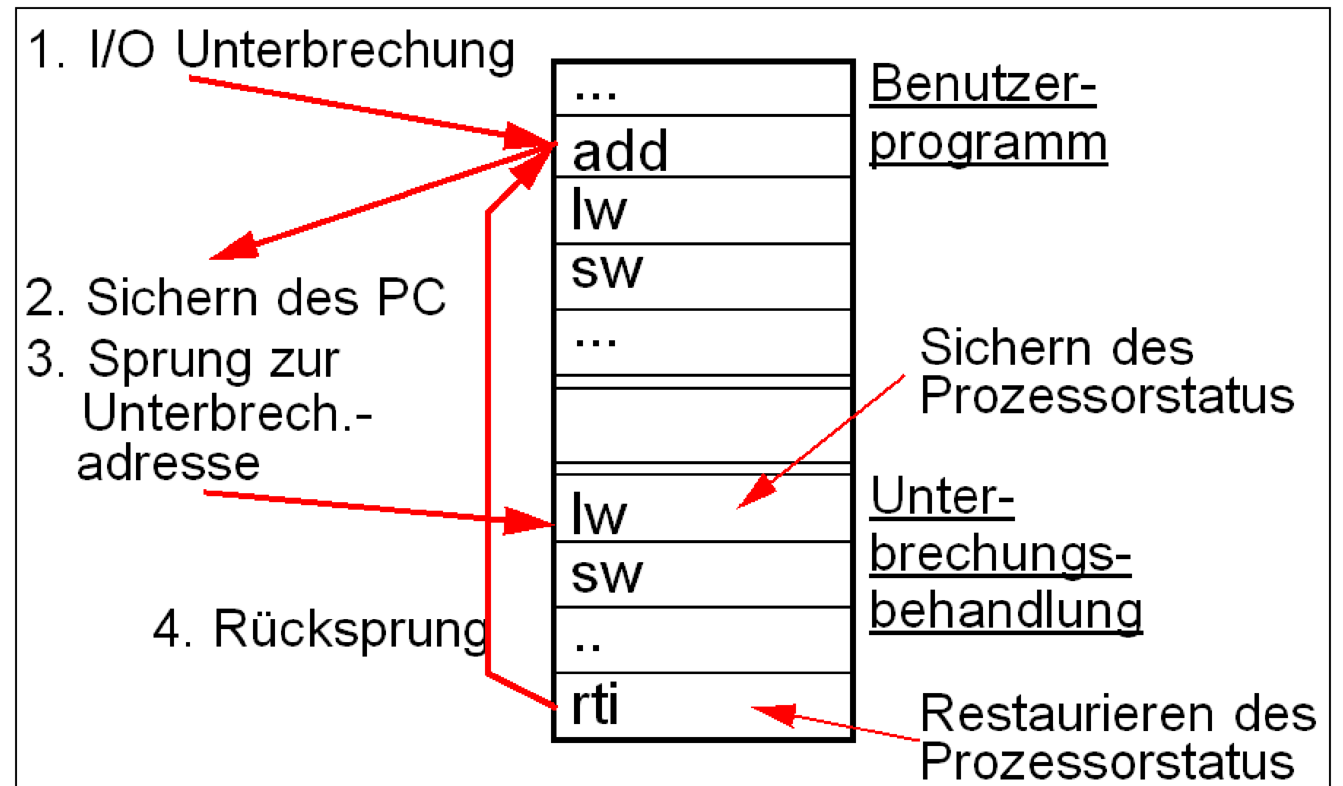
Unterbrechungen (Interrupt) und Ausnahme (Exception):

- Im Gegensatz zu einem Unterprogrammaufruf ist eine Unterbrechung “*nicht geplant*“, d.h. das Programm wird nach der laufenden Instruktion verlassen und eine Unterbrechungsprozedur wird ausgeführt.
- *Gründe* für eine Unterbrechung oder Ausnahme können z.B. sein:
  - Hardwareunterbrechung (spezielle Interruptsignale am Prozessor),
  - arithmetische Ausnahme (z.B. teilen durch 0),
  - falsche Adressierung,
  - Fehler beim Buszugriff,
  - Softwareunterbrechung (Instruktionen `break`, `syscall`),
  - Anhalten zum Zweck des Debuggens, und andere.
- *Anwendungsbeispiele*: Dateneingabe und Datenausgabe, reagieren auf Sensorsignale, Zeitmessungen (timer), ...

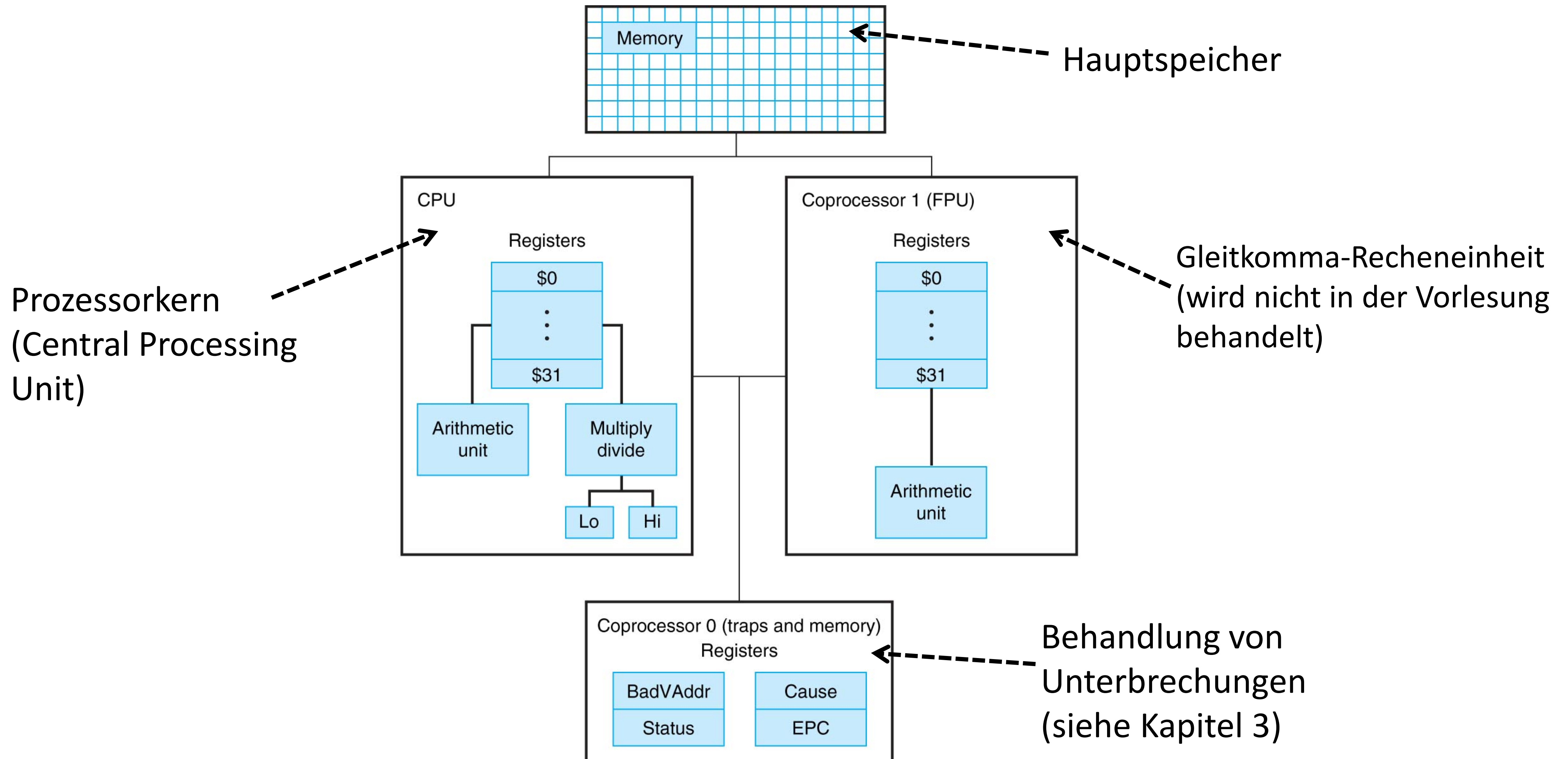
Unterbrechungen

Das Programm an der speziellen “Unterbrechungsadresse“ ist *vergleichbar mit einem Unterprogramm*. Spezielle Eigenschaften:

- Das verlassene Programm hat *keine Register gesichert*, dies erfolgt also im Unterbrechungsprogramm.
- Weitere Unterbrechungen können zugelassen werden: **Vorsicht!**
- Unterbrechungsprogramm kehrt zur *Adresse im EPC-Register* zurück.

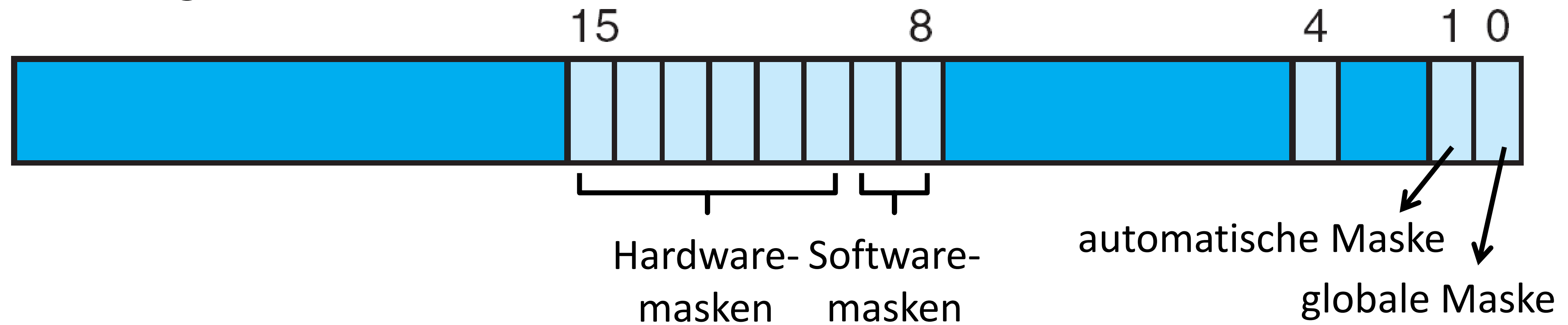


Logische Struktur eines MIPS Prozessors (Wiederholung)

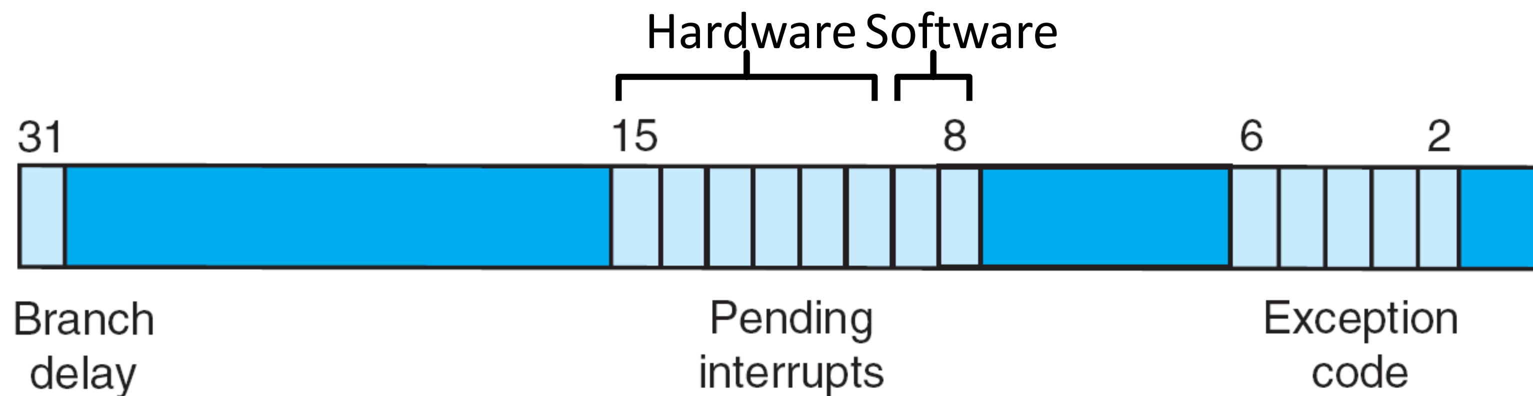


Unterbrechungen

- Status-Register `($12)`:

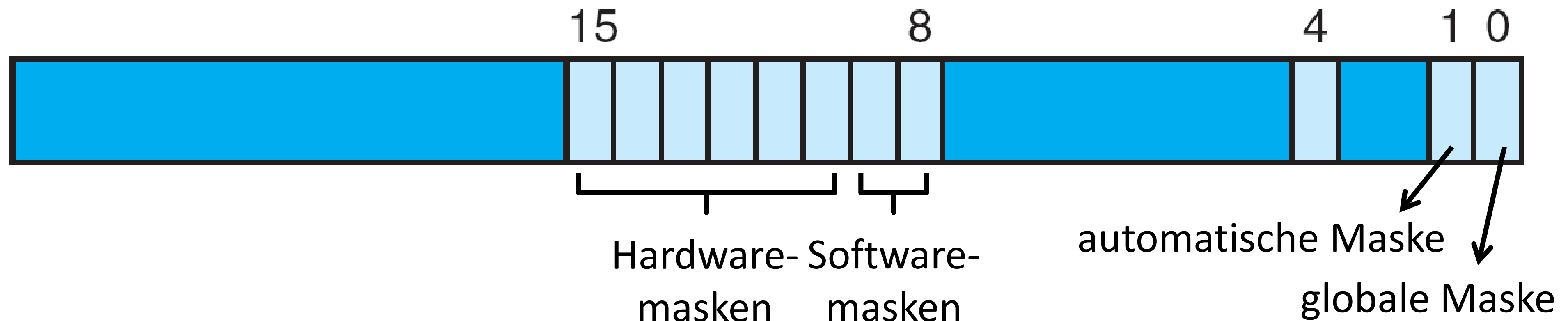


- Cause-Register `($13)`:



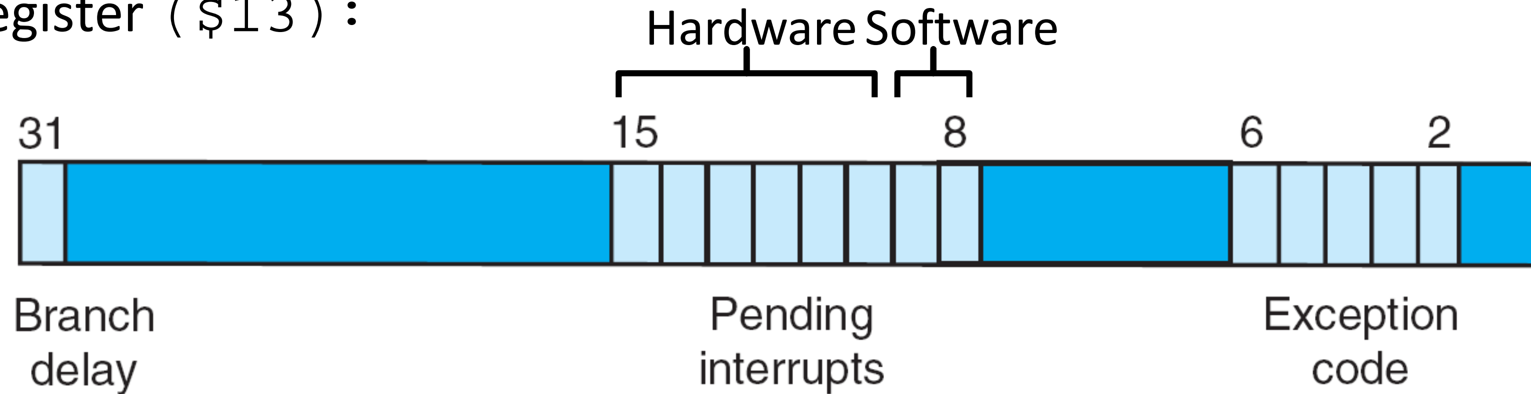
Unterbrechungen

- Es gibt zahlreiche *Hardware- und Softwaremechanismen* zur Behandlung von Unterbrechungen. Es wird beispielhaft nur der Mechanismus im MIPS-Rechner erläutert.
- Unterbrechungen werden vom *Coprozessor 0* erkannt und verarbeitet. Einige der wichtigsten Register hierzu sind `EPC($14)`, `Status ($12)` und `Cause ($13)`. Mit den Instruktionen `mtc0` (store) und `mfc0` (load) können die Inhalte dieser Register in CPU-Register geladen/gespeichert werden.
- Status-Register `($12)`:



Unterbrechungen

- Cause-Register `($13)`:



- Bei Erkennung einer Unterbrechung, die
  - nicht durch eine 0 in der entsprechende Hardware- (Bits 10-15 des *Status Registers*) oder Softwaremaske (Bits 8-9 des *Status Registers*) oder
  - nicht durch eine 0 im globalen Maskierungsbit (Bit 0 des *Status Registers*) oder
  - nicht durch eine 0 im automatischen Maskierungsbit (Bit 1 des *Status Registers*) maskiert ist,

  wird folgendes ausgeführt:

Unterbrechungen

a. Das laufende Programm wird unterbrochen, die Adresse der zu beginnenden oder derzeit laufenden Instruktion im *EPC-Register* gespeichert und der Programmzähler auf eine spezielle Adresse (0x80000180) gesetzt. Hier steht ein benutzerdefiniertes "*Unterbrechungsprogramm*".
b. Weitere Unterbrechungen werden durch Setzen des automatischen Maskierungsbits (Bit 1 des *Status-Register*s) auf 0 *maskiert*.
c. Im *Cause-Register* wird das der Unterbrechung entsprechende Bit im Feld von Bit 8-15 auf 1 gesetzt und die Nummer der *Ursache* im Feld von Bit 2-6 eingetragen.
d. Das "*Unterbrechungsprogramm*" wird ausgeführt.

- Je nach Ursache der Unterbrechung oder Ausnahme kann auch auf eine andere Adresse gesprungen werden, siehe Kapitel 6 und 7. Ansonsten bleibt der Ablauf aber vergleichbar.

Unterbrechungsursachen

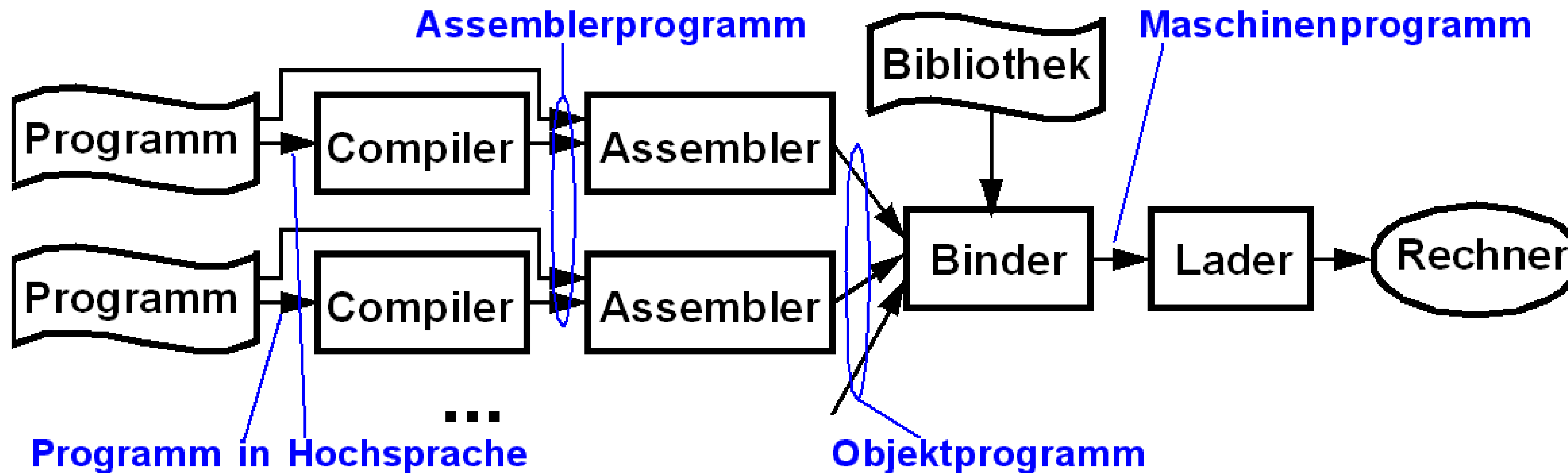
Einige Beispiele von Unterbrechungsursachen:

Number	Name	Cause of exception
0	Int	interrupt (hardware)
4	AdEL	address error exception (load or instruction fetch)
5	AdES	address error exception (store)
6	IBE	bus error on instruction fetch
7	DBE	bus error on data load or store
8	Sys	syscall exception
9	Bp	breakpoint exception
10	RI	reserved instruction exception
11	CpU	coprocessor unimplemented
12	Ov	arithmetic overflow exception
13	Tr	trap
15	FPE	floating point

Vom Programm zur Ausführung

Übersetzungsvorgang

Ein Programm in einer ***Hochsprache*** wird vom ***Compiler*** (Übersetzer) in ein Programm in *Assemblersprache*, anschliessend vom ***Assembler*** in ein ***Objektprogramm*** und dann vom ***Linker*** (Binder) in ein ***Maschinenprogramm*** übersetzt. Der ***Loader*** (Lader) plaziert das Maschinenprogramm in den Speicher:



Hochsprache - Assemblerprogramm

C-Programm (Hochsprache)

```
#include <stdio.h>

int
main (int argc, char *argv[])
{
    int i;
    int sum = 0;

    for (i = 0; i <= 100; i = i + 1) sum = sum + i * i;
    printf ("The sum from 0 .. 100 is %d\n", sum);
}
```

MIPS Assembler

```
        .text
        .align  2
        .globl  main
main:
        subu    $sp, $sp, 32
        sw      $ra, 20($sp)
        sd      $a0, 32($sp)
        sw      $0,  24($sp)
        sw      $0,  28($sp)
loop:
        lw      $t6, 28($sp)
        mul     $t7, $t6, $t6
        lw      $t8, 24($sp)
        addu    $t9, $t8, $t7
        sw      $t9, 24($sp)
        addu    $t0, $t6, 1
        sw      $t0, 28($sp)
        ble     $t0, 100, loop
        la      $a0, str
        lw      $a1, 24($sp)
        jal     printf
        move    $v0, $0
        lw      $ra, 20($sp)
        addu    $sp, $sp, 32
        jr      $ra


        .data
        .align  0
str:
        .asciiz "The sum from 0 .. 100 is %d\n"
```

Assembler und Maschinenprogramm

- Der ***Assembler*** übersetzt ein Assemblerprogramm in ein Maschinenprogramm oder Objektprogramm.
- Ein Assemblerprogramm enthält:
  - *Kommentare*
  - *symbolische Operationscodes*
  - *symbolische Registernamen*
  - *symbolische Marken* (Kennzeichnung von Zeilen im Assemblerprogramm)
  - *Makros* (Ersetzung oft vorkommender Programmteile).
- Weiterhin werden üblicherweise
  - zusätzliche Instruktionen (*Pseudoinstruktionen*) definiert, die dann in Maschineninstruktionen übersetzt werden und
  - *Latenzen* im Maschinenprogramm berücksichtigt.

Assembler und Maschinenprogramm

- Beispiel ***Pseudoinstruktion***:

```
move $t0, $t1  # $t0 = $t1
```

wird vom Assembler zum Beispiel übersetzt in:

```
add $t0, $zero, $t1
```

- Beispiele ***Latenz***:
  - Die Ergebnisse aller *Ladeoperationen* (z.B. `lw`) sind erst in der zweiten Instruktion nach der Ladeoperation verfügbar (Latenz). Dies wird vom Assembler berücksichtigt, gilt also nicht im Assemblerprogramm.
  - Alle *Sprung- und Verzweigungsinstruktionen* haben eine Latenz: Die Instruktion nach der Sprung- oder Verzweigungsinstruktion wird in jedem Fall noch ausgeführt (“***Branch Delay Slot***”). Dies wird vom Assembler berücksichtigt, gilt also nicht im Assemblerprogramm, sondern nur im Maschinenprogramm.

Assembler und Maschinenprogramm

- Beispiele zur ***Latenz***:

AP: Assemblerprogramm
MP: Maschinenprogramm

```
        addi $t0,$zero,1
        j jump
        addi $t0, $t0, 1
        addi $t0, $t0, 1
jump:   .......
```

mit Delay (MP): t0 = 2
ohne Delay (AP): t0 = 1

```
addi $t0,$zero,1
sw $t0 0($sp)
lw $t1 0($sp)
addi $t1, $t1, 1
.......
```

ohne Latenz (AP): t1 = 2

mit Latenz (MP): t1 = 1
da das Ergebnis der `addi`-Instruktion von der verspäteten Zuweisung der `lw`-Instruktion überschrieben wird

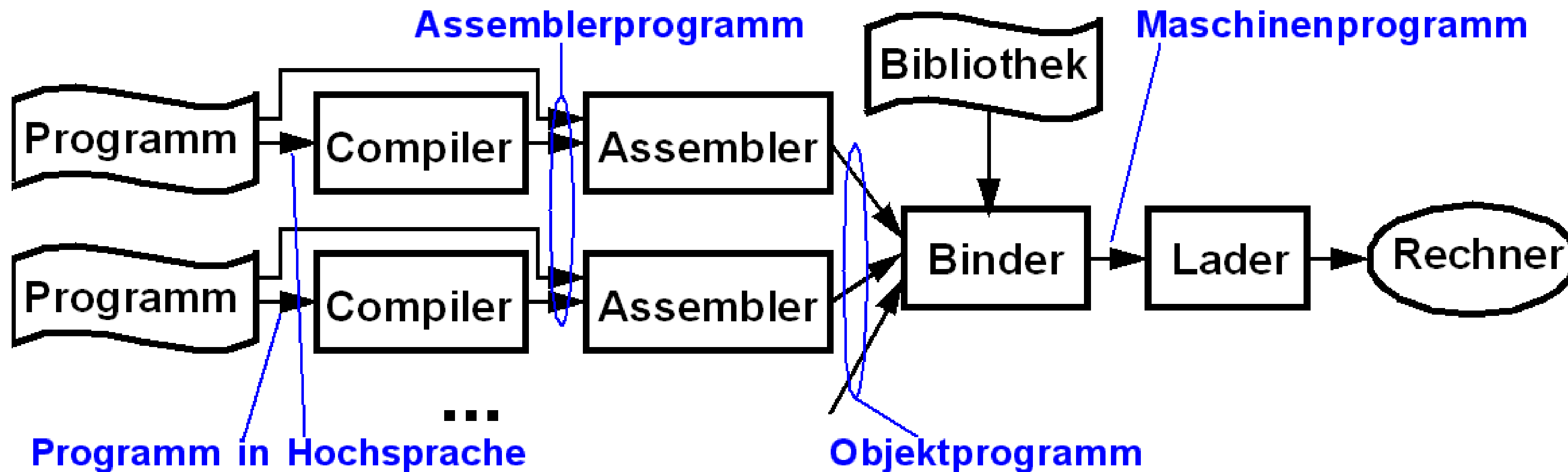
```
        addi $t0,$zero,1
        j jump
jump:   addi $t0, $t0, 1
        ....
```

ohne Delay (AP): t0 = 2

mit Delay (MP): t0 = 3
da die `addi`-Instruktion zweimal ausgeführt wird (einmal wegen der Latenz und ein weiteres Mal am Sprungziel)

Übersetzungsvorgang

Ein Programm in einer ***Hochsprache*** wird vom ***Compiler*** (Übersetzer) in ein Programm in *Assemblersprache*, anschliessend vom ***Assembler*** in ein ***Objektprogramm*** und dann vom ***Linker*** (Binder) in ein ***Maschinenprogramm*** übersetzt. Der ***Loader*** (Lader) plaziert das Maschinenprogramm in den Speicher:



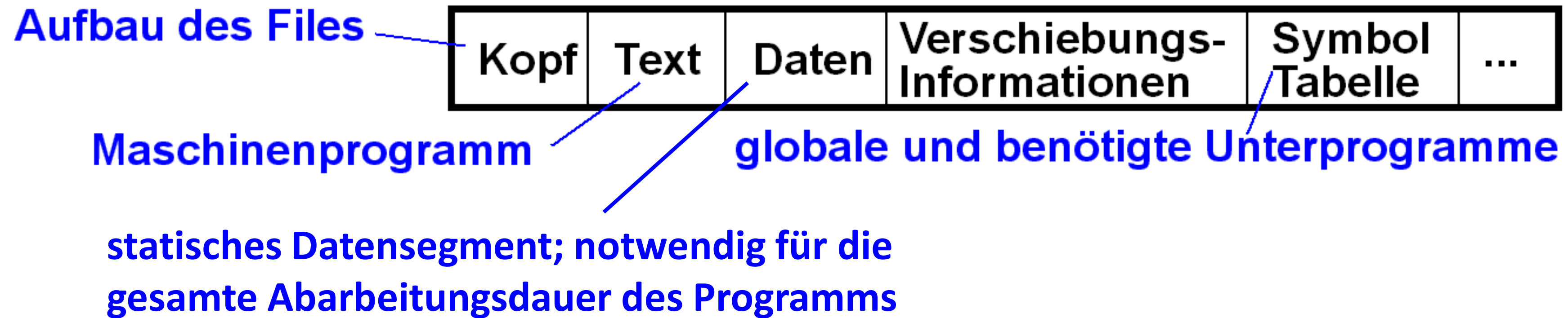
Objektsprache

***Aufbau eines Objektprogramms*:**

- Ein Objektprogramm enthält den Programmtext in Maschinensprache sowie die zugehörigen binären Daten.
- Es enthält zudem zusätzliche Informationen, die es erlauben, mehrere Objektprogramme zu verbinden, z.B.
  - Namen der global definierten Unterprogramme
  - Namen von benötigten Unterprogrammen (Querbezüge)
  - Verschiebungsinformationen (Instruktionen und Daten, die von absoluten Adressen abhängen)

Objektprogramm

Beispiel eines Objektformates (UNIX/LINUX):



Beispiel eines Maschinenprogramms (32-Bit breites binäres MIPS-Format):

00100111101111011111111111000000
10101111101111110000000000010100
10101111101001000000000000100000
...

Objektprogramm

Beispiel zweier Objektprogramme:

Object file header			
	Name	Procedure A	
	Text size	100_{hex}	
	Data size	20_{hex}	
Text segment	Address	Instruction	
	0	`lw $a0, 0($gp)`	
	4	`jal 0`	
	…	…	
Data segment	0	(X)	
	…	…	
Relocation information	Address	Instruction type	Dependency
	0	lw	X
	4	jal	B
Symbol table	Label	Address	
	X	—	
	B	—	
Object file header			
	Name	Procedure B	
	Text size	200_{hex}	
	Data size	30_{hex}	
Text segment	Address	Instruction	
	0	`sw $a1, 0($gp)`	
	4	`jal 0`	
	…	…	
Data segment	0	(Y)	
	…	…	
Relocation information	Address	Instruction type	Dependency
	0	sw	Y
	4	jal	A
Symbol table	Label	Address	
	Y	—	
	A	—	

Linker

Aufgabe und Arbeitsweise eines Binders (Linker):

- Fasst alle zu einem Programm gehörenden Teile zusammen und stellt somit sicher, dass das Programm keine undefinierten Marken enthält
- Sucht in Bibliotheken alle vordefinierten Teilprogramme, die im Programm benutzt werden (verwendet hierzu Symboltabelle)
- Bestimmt Speicherbereiche, die durch die einzelnen Programmteile belegt werden und verschiebt die Anweisungen (verwendet zum Anpassen der absoluten Adressen die Verschiebungsinformationen)
- Löst die Querbezüge zwischen den Programmteilen auf

Objektprogramm

Beispiel nach dem Linken der beiden Objektprogramme:

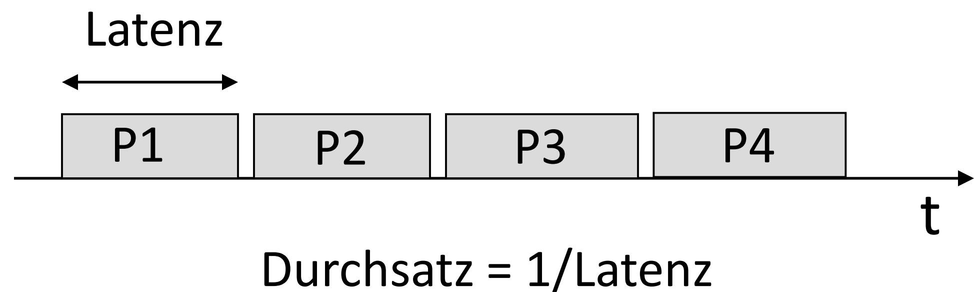
Executable file header		
	Text size	300_{hex}
	Data size	50_{hex}
Text segment	Address	Instruction
	$0040\ 0000_{hex}$	lw $a0, 8000_{hex}($gp)
	$0040\ 0004_{hex}$	jal $40\ 0100_{hex}$
	...	...
	$0040\ 0100_{hex}$	sw $a1, 8020_{hex}($gp)
	$0040\ 0104_{hex}$	jal $40\ 0000_{hex}$
	...	...
Data segment	Address	
	$1000\ 0000_{hex}$	(X)
	...	...
	$1000\ 0020_{hex}$	(Y)
	...	...

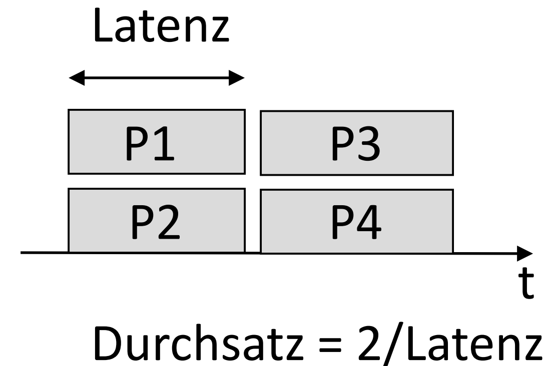
Rechenleistung

Bewertung eines Rechners

Bewertung eines Rechners (Beispiele):

- Kosten, Energieverbrauch
- Ausführungszeit von Programmen/Instruktionen/Aufgaben (***Latenz, Rechenzeit***)
- Zahl der Programme/Instruktionen/Aufgaben pro Zeiteinheit (***Durchsatz, Throughput***)
- ***Reaktionszeit*** auf Unterbrechungen (eingebettete Systeme)
- … und vieles mehr.





Rechenleistung

$$\text{CPUZeit} = \frac{\text{Rechenzeit}}{\text{Programm}} = \frac{\text{Instruktionen}}{\text{Programm}} \cdot \frac{\text{Takte}}{\text{Instruktion}} \cdot \frac{\text{Zeit}}{\text{Takt}}$$

$$\text{IPS} = \frac{\text{Instruktionen}}{\text{Zeit}} = \frac{\text{Instruktionen}}{\text{Programm}} \cdot \frac{1}{\text{CPUZeit}}$$

CPI

(Takt−)Zykluszeit

MIPS = Millionen Instruktionen pro Sekunde

CPI = Mittlere Zahl der Taktzyklen pro Instruktion

Components of performance	Units of measure
CPU execution time for a program	Seconds for the program
Instruction count	Instructions executed for the program
Clock cycles per instruction (CPI)	Average number of clock cycles per instruction
Clock cycle time	Seconds per clock cycle

Rechenleistung

n = Zahl der unterschiedlichen Instruktionsarten im Instruktionssatz

CPI_i = Zahl der Takte für Instruktionsart i

F_i = Anteil der Instruktionsart i an Gesamtzahl von Instruktionen

N_i = Zahl der Instruktionen der Instruktionart i im Programm

N = Gesamtzahl der Instruktionen im Programm

$$\mathrm{CPI} = \sum_{i=1}^{n} \mathrm{CPI}_i \cdot F_i \; ; \quad F_i = \frac{N_i}{N}$$

Einflussfaktoren	# Instruktionen	CPI	Taktdauer
Algorithmus/Programmiersprache	X	X	
Compiler	X	X	
Instruktionssatz	X	X	X
Rechnerarchitektur		X	X
Technologie			X

Benchmarks

SPEC: Weit verbreitete Benchmarks zum Vergleich von Rechnerarchitekturen: www.spec.org

- CINT2006 (Integer), CFP2006 (Gleitkomma): Geometrisches Mittel der relativen Ausführungszeiten für eine Menge von Programmen:

$$SPEC = \sqrt[n]{\prod_{i=1}^{n} \frac{\mathrm{CPUZeit}_i}{\mathrm{ReferenzZeit}_i}}$$

- CINT2006 und CFP2006 vernachlässigen die Daten Ein- und Ausgabe und fokussieren auf die Performanz des Prozessors.
- Andere Benchmarks existieren, die speziell die Ein- und Ausgabe, den Leistungsverbrauch, Multimedia-Anwendungen u.s.w. bewerten.

Benchmarks

Description	Name	Instruction Count x 10^9	CPI	Clock cycle time (seconds x 10^{-9})	Execution Time (seconds)	Reference Time (seconds)	SPECratio
Interpreted string processing	perl	2252	0.60	0.376	508	9770	19.2
Block-sorting compression	bzip2	2390	0.70	0.376	629	9650	15.4
GNU C compiler	gcc	794	1.20	0.376	358	8050	22.5
Combinatorial optimization	mcf	221	2.66	0.376	221	9120	41.2
Go game (AI)	go	1274	1.10	0.376	527	10490	19.9
Search gene sequence	hmmer	2616	0.60	0.376	590	9330	15.8
Chess game (AI)	sjeng	1948	0.80	0.376	586	12100	20.7
Quantum computer simulation	libquantum	659	0.44	0.376	109	20720	190.0
Video compression	h264avc	3793	0.50	0.376	713	22130	31.0
Discrete event simulation library	omnetpp	367	2.10	0.376	290	6250	21.5
Games/path finding	astar	1250	1.00	0.376	470	7020	14.9
XML parsing	xalancbmk	1045	0.70	0.376	275	6900	25.1
Geometric mean	–	–	–	–	–	–	25.7

SPECINTC2006 Benchmarks auf einem Intel Core i7 920 (4 cores).

Die grossen CPI-Werte für mcf und omnetpp beruhen auf einer hohen Cache-Missrate (siehe später …).

Andere Instruktionssätze

Rechenleistung

***Kriterien* für Instruktionssatz:**

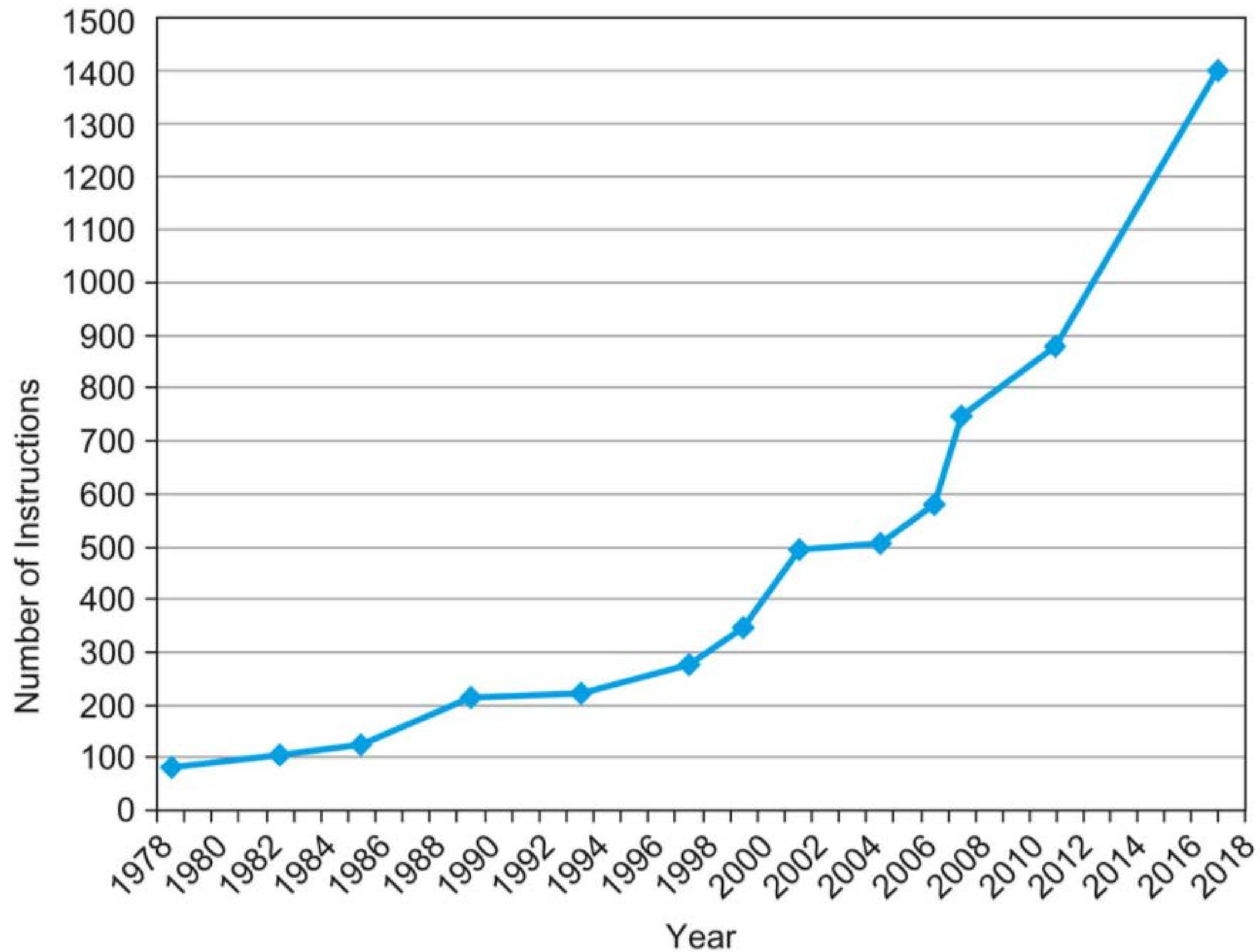
- Effiziente *Übersetzung* möglich (klar strukturierter Instruktionssatz)
- *Wenige Instruktionen* pro Programm (Einführung komplexer Instruktionen); *Vorsicht*: komplexe Instruktionen führen zu erhöhter CPI und erhöhter Taktdauer.
- Effiziente *Hardwareimplementierung* (einfache Instruktionen, wenige unterschiedliche Instruktionsarten, einfache Instruktionskodierung)
- Wenige *Takte pro Instruktion* (Parallelverarbeitung, Pipelining)
- *Ressourcen* dort einsetzen, wo wesentliche Zeit benötigt wird: CPI_i x F_i (spezielle Hardware für Gleitkommaoperationen, Multimediaoperationen, Graphik, ...)

Der IA-32 Instruktionssatz

Hat sich im Laufe von fast 30 Jahren entwickelt und ist der bei PCs ***am weitesten verbreitete Instruktionssatz***.

- Wesentliche Eigenschaft: *Rückwärtskompatibilität.*
- ***Historie*** (hier nur einige wenige Stationen erwähnt):
  - 1978: Intel 8086 Architektur (16 Bit Architektur)
  - 1980: Coprozessor für Gleitkommaoperationen
  - 1985: Erweiterung auf 32 Bit, weitere Instruktionen zugefügt
  - 1997: weitere 57 Multimedia-Instruktionen (MMX)
  - 1999: weitere 70 Instruktionen, 128 Bit Register
  - 2001: weitere 144 Instruktionen (Streaming SSE2)
  - 2004: weitere Instruktionen (128 Bit, video, graphics, threads, SSE3)
  - 2006-2011: weitere 352 Instruktionen …

Der IA-32 Instruktionssatz



Der IA-32 Instruktionssatz

- Der ***MIPS Instruktionssatz*** ist ein Vertreter der ***RISC*** (reduced instruction set) Familie:
  - Einfache Kodierung (alle Instruktionen gleich lang),
  - wenige orthogonale Instruktionsklassen (Speicher, Berechnung, Verzweigung),
  - viele (32 und mehr) allgemein verwendbare Register.
- Damit sind RISC Instruktionssätze auf *moderne Rechnerarchitekturen* (hohe ***Parallelität*** der Ausführung, ***hohe Taktrate***) abgestimmt.
- Der IA-32 Instruktionssatz (***CISC , complex instruction set***) ist in allen diesen Punkten unterschiedlich. Es folgen einige Beispiele zur Illustration.

Der IA-32 Instruktionssatz

Nur 8 interne
allgemein verwendbare
Register
(80386 Prozessor)

Name	31 ... 0	Use
EAX		GPR 0
ECX		GPR 1
EDX		GPR 2
EBX		GPR 3
ESP		GPR 4
EBP		GPR 5
ESI		GPR 6
EDI		GPR 7
CS		Code segment pointer
SS		Stack segment pointer (top of stack)
DS		Data segment pointer 0
ES		Data segment pointer 1
FS		Data segment pointer 2
GS		Data segment pointer 3
EIP		Instruction pointer (PC)
EFLAGS		Condition codes

Der IA-32 Instruktionssatz

Keine klare Trennung in *Speicher-Operationen* (nur load and store) und *arithmetisch-logische Instruktionen*:

Source/destination operand type	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

Komplexe Adressierungsarten:

Mode	Description	Register restrictions	MIPS equivalent
Register indirect	Address is in a register.	not ESP or EBP	`lw $s0,0($s1)`
Based mode with 8- or 32-bit displacement	Address is contents of base register plus displacement.	not ESP or EBP	`lw $s0,100($s1) # ≤16-bit # displacement`

Base plus scaled index	The address is Base + (2^{Scale} x Index) where Scale has the value 0, 1, 2, or 3.	Base: any GPR
		Index: not ESP
			`mul $t0,$s2,4`
			`add $t0,$t0,$s1`
			`lw $s0,0($t0)`

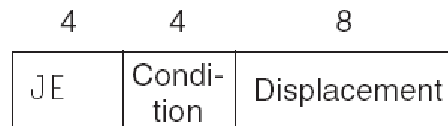
Base plus scaled index with 8- or 32-bit displacement	The address is Base + (2^{Scale} x Index) + displacement where Scale has the value 0, 1, 2, or 3.	Base: any GPR
		Index: not ESP
			`mul $t0,$s2,4`
			`add $t0,$t0,$s1`
			`lw $s0,100($t0) # ≤16-bit # displacement`

IA-32

Unterschiedliche Instruktionslängen zwischen 1 Byte und 17 Byte.

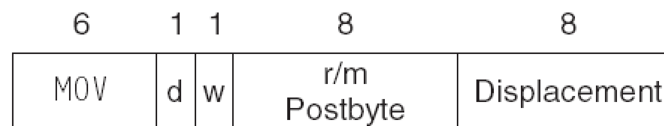
In der *Rechner-Architekturen* werden daher zunächst die IA-32 Instr. in RISC Instruktionen umgesetzt! Hoher Zusatzaufwand.

a. JE EIP + displacement

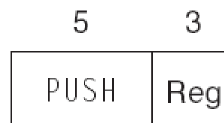


b. CALL

c. MOV EBX, [EDI + 45]



d. PUSH ESI



e. ADD EAX, #6765

f. TEST EDX, #42