

- ❑ Spim-Simulator
- ❑ Besonderheiten: globaler Zeiger, lb & lbu, ... usw.
- ❑ Statische und dynamische Speicherallokierung
- ❑ Programmiertechniken in Assembler
- ❑ Unterprogrammaufrufe

Der SPIM-Simulator

Die MIPS-Architektur ist schwer zu programmieren

Gründe:

- verzögertes Verzweigen (delayed branch)
Verzweigung benötigt zwei Zyklen → Befehl unmittelbar nach einem Verzweigungsbefehl wird noch ausgeführt
- verzögertes Laden (delayed load)
Speicherzugriff benötigt zwei Zyklen → Befehl unmittelbar nach einem Ladebefehl kann Speicherinformation noch nicht nutzen
- nur eine Adressierungsart (R2000)

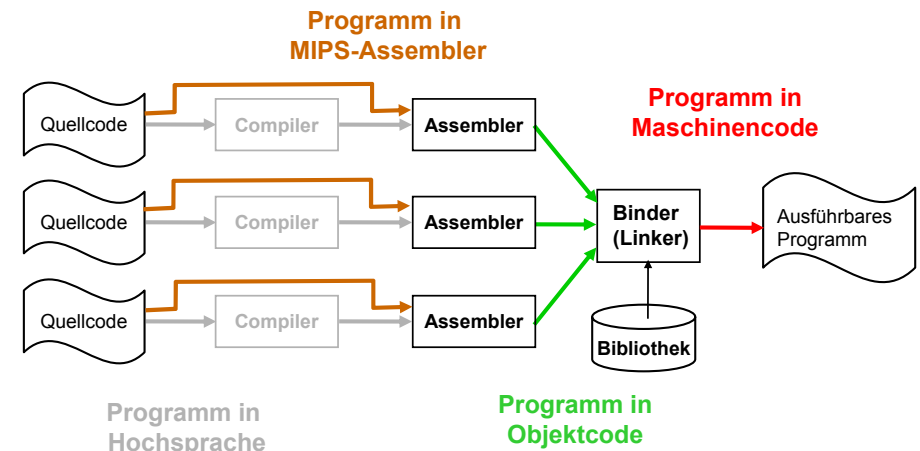
Der SPIM-Simulator

Lösungsmöglichkeit:

- Assembler generiert Code für eine virtuelle Maschine, d. h.
 - automatisches Vertauschen von Instruktionen, um Verzögerungslücken (delay slots) zu füllen
 - Bereitstellen von Pseudobefehlen, die durch eine kurze Folge von MIPS-Befehlen ersetzt werden
 - Bereitstellen von zusätzliche Adressierungsarten, die auf die einzige MIPS-Adressierungsart abgebildet werden

Der SPIM-Simulator

Der SPIM-Simulator ist Assembler, Linker und Debugger in einem Programm.



Installation und Benutzung

- Software abrufbar unter:

<http://i61www.ira.uka.de/users/asfour/TI/TI-2/Spim>

- SPIM ist Public-Domain. Die neueste Version ist erhältlich unter:

<ftp://ftp.cs.wisc.edu/pub/spim>



Ersetzung von Pseudoinstruktionen

Pseudoinstruktion	wird ersetzt durch
<code>move Rd, Rs</code>	<code>addu Rd, \$0, Rs</code>
<code>neg Rd, Rs</code>	<code>sub Rd, \$0, Rs</code>
<code>b sym</code>	<code>bgez \$0, sym</code>
<code>li Rd, Imm</code>	<code>ori Rd, \$0, Imm</code>
<code>la Rd, sym</code>	<code>lui \$at, [sym / 10000₁₆] ori Rd, \$at, sym & FFFF₁₆</code>
<code>l.d Fd, sym</code>	<code>lui \$at, [sym / 10000₁₆] lwcl Fd, sym & FFFF₁₆(\$at) lui \$at, [sym / 10000₁₆] lwcl Fd + 1, sym & FFFF₁₆(\$at)</code>



Ersetzung von Pseudoinstruktionen

Pseudoinstruktion	wird ersetzt durch
<code>bge Ra, Rb, sym</code>	<code>slt \$at, Ra, Rb beq \$at, \$0, sym</code>
<code>abs Rd, Rs</code>	<code>addu Rd, \$0, Rs bgez Rs, lbl sub Rd, \$0, Rs lbl:</code>
<code>rol Rd, Rs, dist</code>	<code>srl \$at, Rs, 32 - dist sll \$at, Rd, Rs, dist or Rd, Rd, \$at</code>
<code>rem Rd, Ra, Rb</code>	<code>bne Rb, \$0, lbl break 0 lbl: div Ra, Rb mfhi Rd</code>
<code>nop</code>	<code>or \$0, \$0, \$0</code>



MIPS R2000: Adressierungsarten

Der MIPS-Prozessor (R2000) besitzt nur die Adressierungsart **imm(reg)**, wobei die Adresse aus der Summe des Registerinhaltes von **reg** und des unmittelbaren Wertes **imm** gebildet wird.

Die virtuelle Maschine unterstützt jedoch auch folgende Formate:



Adressierungsarten in SPIM

Format	Beispiel	Adressberechnung
(register)	(\$s0)	Inhalt des Registers
imm	0x10003248	unmittelbarer Wert
imm(register)	0x23 (\$s4)	Inhalt des Registers + unmittelbarer Wert
symbol	label1	Adresse des Symbols
symbol ± imm	marke+0x45	Adresse des Symbols ± unmittelbarer Wert
symbol ± imm(register)	label + 0x13 (\$s1)	Adresse des Symbols ± (unmittelbarer Wert + Inhalt des Registers)

Systemaufrufe

Die Nummer des Systemaufrufes wird in Register **\$v0** übergeben:

```
li $v0, Nummer
syscall
```

Bildschirmausgabe

Dienst	Nummer	Eingabe
print_int	1	Integer in \$a0
print_float	2	float in \$f12
print_double	3	double in \$f12
print_string	4	Adresse des String in \$a0

Systemaufrufe

Tastatureingabe

Dienst	Nummer	(Ein-)Ausgabe
read_int	5	Integer in \$v0
read_float	6	float in \$f0
read_double	7	double in \$f0 (\$f1)
read_string	8	Adresse der Zeichenkette in \$a0 und maximale Länge in \$a1 übergeben Zeichenkette ist nullterminiert. Bei weniger eingegebenen Zeichen wird Zeichenkette zusätzlich mit "\n" abgeschlossen

Systemaufrufe

Sonstiges Systemaufrufe:

Dienst	Nummer	Eingabe	Ausgabe
sbrk	9	Byte-Anzahl in \$a0	Anfangsadresse in \$v0
exit	10		

Beispiel

```
.data
string: .asciiz "Hallo MIPS-World"

.text
la $a0, string    # Adresse von string in $a0
li $v0, 4         # print_string
syscall
```

```
li $v0, Nummer
syscall
```

print_string 4 Adresse des String in \$a0

Ausgabe einer Integerzahl mit CR

```
.data
cr_string: .asciiz "\n"

.text
pr_str:   li $v0, 1          # print_int
          syscall
          la $a0, cr_string  # print_string
          li $v0, 4
          syscall
          jr $ra
```

Struktur eines MIPS-Programms

```
.data
# globale Daten
.text
# Unterprogramme

.globl main

main: subu $sp, $sp, 8      # Stack Frame ist 8 Bytes
      sw $ra, 0($sp)       # Sicherung der Ruecksprungadresse
      sw $fp, 4($sp)       # Sicherung des alten Frame-Pointers
      addu $fp, $sp, 8     # neuen Frame-Pointer definieren

# Hauptprogramm

lw $ra, 0($sp)            # Ruecksprungadresse wiederherstellen
lw $fp, 4($sp)            # Frame-Pointer wiederherstellen
addu $sp, $sp, 8         # Stack-Frame loeschen
jr $ra
```

Struktur eines MIPS-Programms

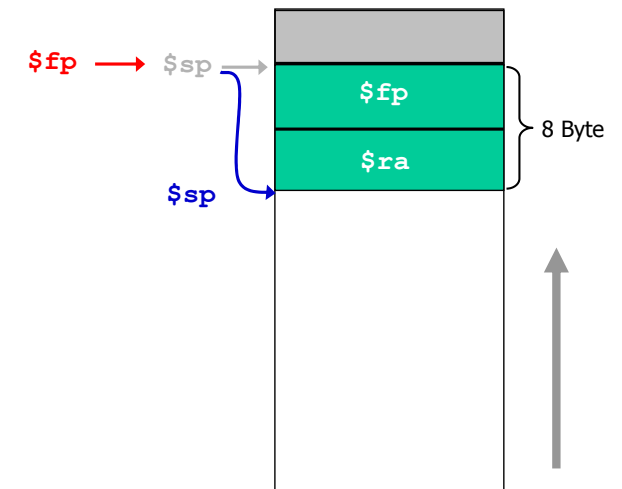
```
.data
# globale Daten
.text
# Unterprogramme

.globl main

main: subu $sp, $sp, 8
      sw $ra, 0($sp)
      sw $fp, 4($sp)
      addu $fp, $sp, 8

# Hauptprogramm

lw $ra, 0($sp)
lw $fp, 4($sp)
addu $sp, $sp, 8
jr $ra
```



Einfacher Unterprogrammaufruf

```
main:          jal subroutine

subroutine:    # keine weiteren
               # Unterprogrammaufrufe

               # für lokale Variablen
               # nur $t0 bis $t9

               jr $ra
```

Beispiel: Arithmetische Befehle

- Alle Befehle haben 3 Operanden
- Operand-Reihenfolge ist fest (Zielregister zuerst)
- Operanden einer arithmetischen Operation **müssen** in Registern stehen. Alle Register sind 32 Bit breit

Beispiele:

C-Code:	$A = B + C$
MIPS-Code:	add \$s0, \$s1, \$s2

C-Code:	$A = B + C + D;$ $E = F - A;$
MIPS-Code:	add \$t0, \$s1, \$s2 add \$s0, \$t0, \$s3 sub \$s4, \$s5, \$s0

Beispiel: Integer-Arithmetik

```
.data
cr_string: .asciiz "\n"          # Sonderzeichen "neue Zeile"
eingabeA:  .asciiz "Integer-Zahl A: "
eingabeB:  .asciiz "Integer-Zahl B: "
result_sum: .asciiz "A + B = "
result_dif: .asciiz "A - B = "
result_mul: .asciiz "A * B = "
result_div: .asciiz "A mod B = "
result_rst: .asciiz "Rest = "
error_str: .asciiz "Division durch Null nicht definiert!\n"
```

```
.text

# Prozedur: Ausgabe eine Integer-Zahl mit CR
print_int: li $v0, 1
           syscall
           la $a0, cr_string
           li $v0, 4
           syscall
           jr $ra
```

Beispiel: Integer-Arithmetik

```
# Prozedur: Ausgabe eines Strings
print_str: li $v0, 4
           syscall
           jr $ra

.globl main
main:      subu $sp, $sp, 8      # Stack Frame ist 8 Bytes
           sw $ra, 0($sp)       # Sichern der Ruecksprungsadresse
           sw $fp, 4($sp)       # Sichern des alten Frame-Pointers
           addu $fp, $sp, 8     # neuen Frame-Pointer definieren

           la $a0, eingabeA     # Integer-Zahl A holen
           jal print_str
           li $v0, 5
           syscall
           move $s0, $v0        # A in $s0 sichern

           la $a0, eingabeB     # Integer-Zahl B holen
           jal print_str
           li $v0, 5
           syscall
           move $s1, $v0        # B in $s1 sichern
```

Beispiel: Integer-Arithmetik

```

la $a0, result_sum      # Ausgabe A + B
jal print_str
add $a0, $s0, $s1
jal print_int

la $a0, result_dif      # Ausgabe A - B
jal print_str
sub $a0, $s0, $s1
jal print_int

la $a0, result_mul      # Ausgabe A * B
jal print_str
mul $a0, $s0, $s1
jal print_int

beqz $s1, error
la $a0, result_div      # Ausgabe A mod B
jal print_str
div $s0, $s1
mflo $a0
jal print_int

```

Beispiel: Integer-Arithmetik

```

la $a0, result_rst # Ausgabe des Restes
jal print_str
mfhi $a0
jal print_int
b fertig

error:    la $a0, error_str # Division durch Null
jal print_str

fertig:   lw $ra, 0($sp)    # Ruecksprungadresse wiederherstellen
lw $fp, 4($sp)            # Frame-Pointer wiederherstellen
addu $sp, $sp, 8          # Stack-Frame loeschen
jr $ra

```

Beispiele

Befehl	Bedeutung
add \$s1, \$s2, \$s3	# \$s1 = \$s2 + \$s3
sub \$s1, \$s2, \$s3	# \$s1 = \$s2 - \$s3
lw \$s1, 100(\$s2)	# \$s1 = Memory[\$s2+100]
sw \$s1, 100(\$s2)	# Memory[\$s2+100] = \$s1
bne \$s4, \$s5, Label	# Nächster Befehl bei Label, # wenn \$s4 ≠ \$s5
beq \$s4, \$s5, Label	# Nächster Befehl bei Label, # wenn \$s4 = \$s5
slt \$s1, \$s2, \$s3	# \$s1 = 1, wenn \$s2 < \$s3 # sonst \$s1 = 0
j Label	# Nächster Befehl bei Label

Lade- und Speicherbefehle

MIPS: Wort mit 32 Bit oder 4 Bytes

12	32 bits
8	32 bits
4	32 bits
0	32 bits
...	

Es werden Wörter
geladen, aber Bytes
adressiert.

- 2^{32} Bytes mit Byte-Adressen von 0 bis $2^{32}-1$
- 2^{30} Wörter mit Byte-Adressen 0, 4, 8, ... $2^{32}-4$
- Wörter sind ausgerichtet.

Welche Werte haben die 2 niedrigstwertigen Bits einer Wort-Adresse?

Lade- und Speicherbefehle

- Speicher: Eindimensionales Array aus Speicherzellen mit Adressen.
- Speicheradresse: Index im Array
- "Byte-Adressierung" heisst, dass der Index auf ein Byte im Speicher zeigt.

...	
7	8 bits
6	8 bits
5	8 bits
4	8 bits
3	8 bits
2	8 bits
1	8 bits
0	8 bits

Beispiel

Lade- und Speicher-Befehle :

C-Code: **A[8] = h + A[8];**

MIPS code: **lw \$t0, 32(\$s3)**
 add \$t0, \$s2, \$t0
 sw \$t0, 32(\$s3)

Unterscheid zwischen **lb** und **lbu**

```
.data
result: .word 0x89abcdef 0x79abcdef

.text

# Start des Hauptprogrammes

.globl main
main:
...
lbu $a0, result
# $a0 enthaelt 0x0000089 0x0000079
...
lb $a0, result
# $a0 enthaelt 0xffffffff89 0x0000079
...
jr $ra
```

MIPS-Speichermodell "Big-Endian"

Big-Endian:

Höchstwertigstes Byte liegt an kleinster Adresse

```
result: .word 0x89abcdef

lbu $a0, result
lbu $a1, result+1
lbu $a2, result+2
lbu $a3, result+3
# $a0 enthaelt 0x89
# $a1 enthaelt 0xab
# $a2 enthaelt 0xcd
# $a3 enthaelt 0xef
```

In SPIM wird die Konvention des unterliegenden Rechners verwendet.

Laden von 32-Bit Operanden

10000010101000001010101010101010 → \$t0

lui \$t0, 1000001010100000 (load upper immediate)

1000001010100000	0000000000000000
------------------	------------------

mit Nullen ausfüllen

ori \$t0, \$t0, 1010101010101010

1000001010100000	0000000000000000
------------------	------------------

0000000000000000	1010101010101010
------------------	------------------

1000001010100000	1010101010101010
------------------	------------------

Der globale Zeiger \$gp

Lade-Befehl (Pseudoinstruktion):

lw rt, address lw \$v0, 0x10008000
Adresse liegt im Datensegment

Bisherige Lösung:

lui \$at, 0x1000 lui \$at, 0x1000
lw rt, Offset(\$at) lw \$v0, 0x8000(\$at)

Offset := vier niedrigstwertige Stellen von address

Bessere Lösung:

lw rt, Offset(\$gp) lw \$v0, 0(\$gp)

Der globale Zeiger \$gp enthält immer den Wert 1000 8000₁₆
→ Zugriff auf die Adressen 1000 0000₁₆ bis 1001 0000₁₆

Speicherallokierung

Statische Speicherallokierung

- erfolgt während der Assemblierung durch Assemblerdirektiven (.word, .space, ...)
- Speicherplatz während der gesamten Laufzeit belegt
- Veränderung der Größe nicht möglich

Dynamische Speicherallokierung:

- erfolgt während der Laufzeit durch Systemaufruf sbrk (Größe des angeforderten Speichers in Bytes in \$a0, Zeiger auf reservierten Speicherbereich in \$v0)
- Speicherplatz nur bei Bedarf belegt
- Aufwand während der Laufzeit

Programmiertechniken

Einstufige Verzweigung:

Eine einstufige Verzweigung wird durch einen bedingten Sprung realisiert

if-Anweisung

```
if ( register_s1 == 0)
{
    if-Anweisungen
}
next-Teil;
```

Assembler-Code

```
beqz    $s1, markel
j       marke2

markel:  { ... } if-Anweisungen
        { ... }
        { ... }

marke2:  .... next-Teil
```

Zweistellige Verzweigung:

if-else-Anweisung

```
if (register_s1 == 0)
{
    if-Anweisungen
}
else
{
    else-Anweisungen
}
next;
```

Assembler-Code

```
beqz    $s1, marke1
{
    ...
} else-Anweisungen
j       marke2
marke1: {
    ...
} if-Anweisungen
marke2: .... next-Teil
```

```
if $s1 < $s2 then
    $t0 = 1
else
    $t0 = 0
} slt $t0, $s1, $s2
```

Verzweigungsbefehle:

C:

```
if( ) {...}
else { if( ) {...}
      else { if( ) {...}
            else {...}
      }
}
```

C,C++:

```
switch (note)
{
    case1: 1-Anweisungen
            break;
    case2: 2-Anweisungen
            break;
    ...
    ...
    case6: 6-Anweisungen
            break;
    default: Fehlermeldung
            break;
}
```

Note steht in der Variablen note

```
lw $t0, note
li $t1, 1
li $t2, 2
li $t5, 5

beq $t0, $t1, marke1
beq $t0, $t2, marke2
....
....
....
beq $t0, $t5, marke5
.... # default
b weiter

marke1: .... # Note 1
        ....
        b weiter
marke2: .... # Note 2
        ....
        b weiter
marke5: .... # Note 5
        ....
        b weiter
weiter: .... # Mat.-Nr. ....
        .... # hat die Note:
```

Beispiel: Bedingte Verzweigung

```
bne $t0, $t1, Label
```

```
beq $t0, $t1, Label
```

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

```
bne $s0, $s1, Label
add $s3, $s0, $s1
```

```
Label:     ....
```

Beispiel: Unbedingte Verzweigung

```
j label
```

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

```
beq $s4, $s5, Lab1
add $s3, $s4, $s5
j Lab2
```

```
Lab1: sub $s3, $s4, $s5
Lab2:  ...
```

Unterprogrammaufrufe

- ❑ Regeln oder Konventionen zur Verwendung von **Registern** und **Kellerspeicher (Stack)** bei einem Unterprogrammaufruf
- ❑ Informationen bezüglich des Unterprogramms und des unterbrochenen Programms werden in einem **Rahmen (Frame)** auf dem Stack gespeichert. Diese können:
 - Argumente des Unterprogramms
 - Lokale Variablen des Unterprogramms
 - Registerinhalte, die vom Unterprogramm nicht verändert werden dürfen

Unterprogrammaufrufe

Ein **Argument** ist eine Integer- oder Fließkomma-Zahl oder ein Zeiger auf eine größere Datenstruktur (z. B. Zeichenkette)

Aufgaben des Aufrufers:

- *Temporäre* Register \$t0 bis \$t9 sichern, falls gültige Daten darin enthalten sind
- Übergabe der ersten vier Argumente in den Registern \$a0 bis \$a3 (ggf. als Zeiger auf das Argument)
- Register \$a0 bis \$a3 sichern, falls die Argumente später noch gebraucht werden
- Übergabe weiterer Argumente auf dem Stack

Unterprogrammaufrufe

Aufruf eines Unterprogramms an der Adresse <addr> mit dem Befehl

jal <addr>

Die Adresse des nachfolgenden Befehls wird im Register `$ra` gespeichert.

Informationen bezüglich des Unterprogramms werden in einem Rahmen auf dem Stapel gespeichert.

Rahmengröße := (Anzahl der Argumente + Anzahl der zu sichernden Register + Anzahl der lokalen Variablen) × 4

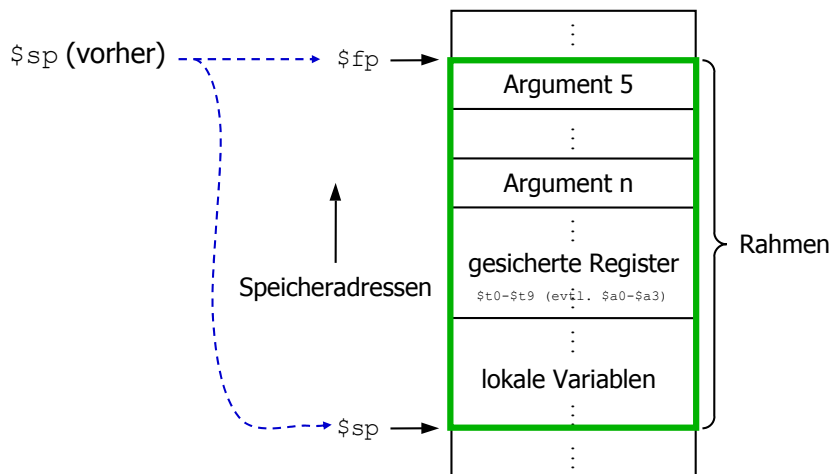
(Zu sichernde Fließkommazahlen mit doppelter Genauigkeit benötigen acht Bytes Speicherplatz)

Unterprogrammaufrufe

Aufgaben des Aufgerufenen:

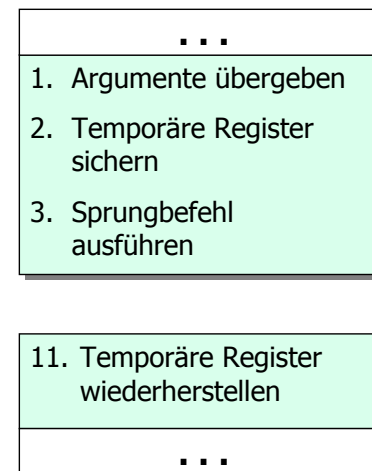
- Reservieren von Speicherplatz für einen neuen Rahmen durch Subtraktion der Rahmengröße von Stack-Pointer (`$sp`)
- Sichern der Register `$s0` bis `$s7`, falls diese im Unterprogramm verwendet werden
- Sichern des Rücksprungadressenregisters `$ra`, falls das Unterprogramm weitere Unterprogramme aufruft
- Sichern der Rahmenzeigers `$fp`
- Anlegen eines neuen Rahmenzeigers durch Addition der Rahmengröße zum Stapelzeiger

Unterprogrammaufruf



Unterprogrammaufruf

Aufrufendes Programm



Unterprogramm

