

Assembler programmierung

RISC V

( MIPS )

# Assemblyprogrammierung

## Weiterungen zur Programmierung

Von der Hochsprache zur Maschine

- Entscheidungen ( IF - CASE )

- Schleifen ( FOR - WHILE )

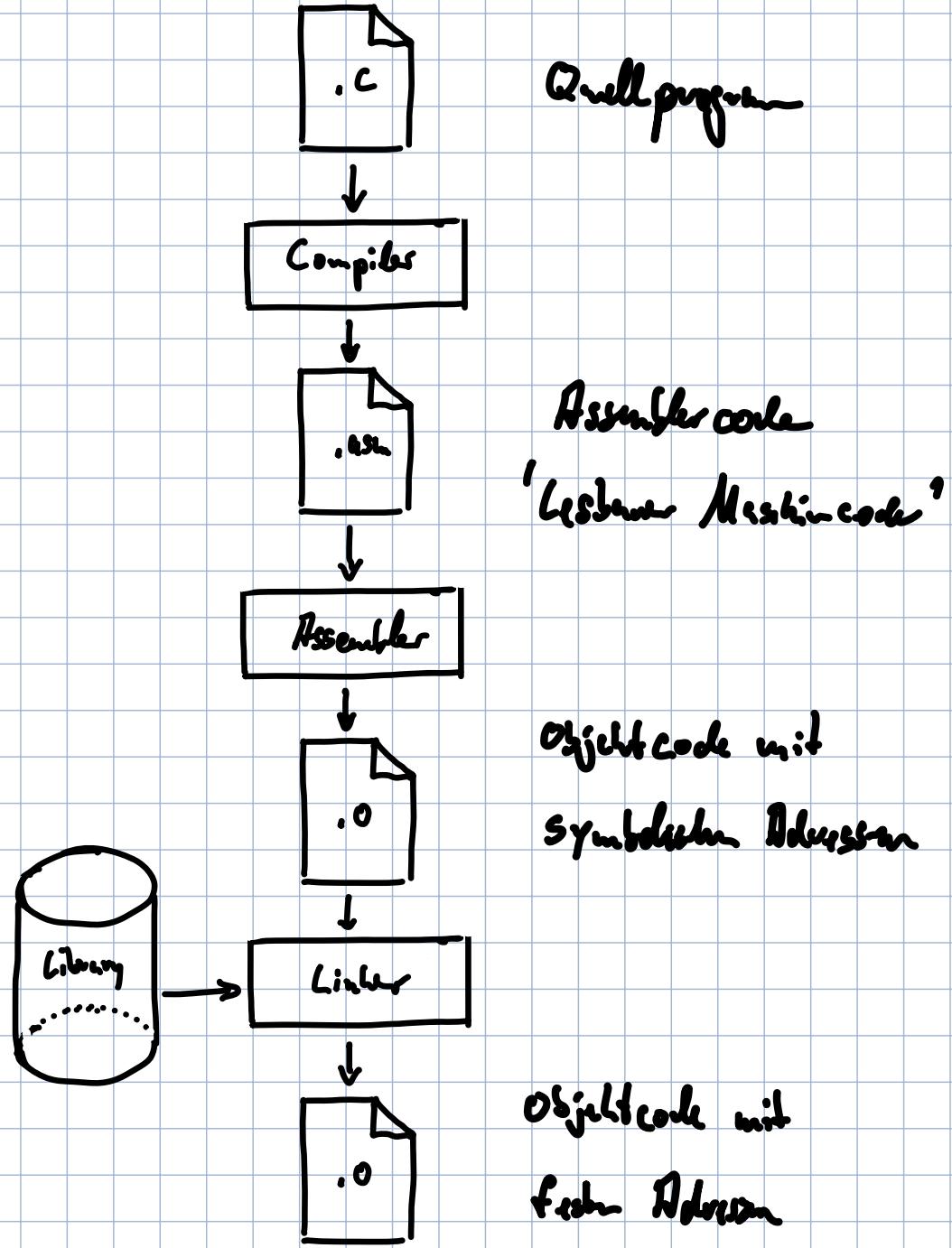
- Listen ( ARR ) und Strukturen ( STRUCT )

- Unterprogramme ( Prozeduren und Funktionen )

- Rekursionen und : Die Fibonacci-Zahlen

- Werkzeuge und transkribierte Funktionen

## Der Übersetzungsprozess:



## Macros in MARS :

.macro myMacro( %n, %m, ... )

label : some code

more code

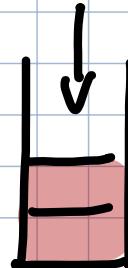
j labelb

.endmacro

f = forward

b = backward

push  
( eingehen, schließen )



pop  
( öffnen, herausholen )



Schablone: Verzweigung

CB = Kontrollblock

RISCV

BB = Basisblock

if ( CB ) {

BB<sub>1</sub> ;

3 else {

BB<sub>2</sub> ;

3

cb: beq t<sub>1</sub>, t<sub>2</sub>, if

bne \$t<sub>1</sub>, t<sub>2</sub>, else

if : BB<sub>1</sub>

j fi

else: BB

fi:

Schablone: for-Schleife

RISCV

for ( $u \geq 0; u < m; u++$ ) {

BB<sub>1</sub>

3

Annahm:  $u = \$t_1$      $m = \$t_2$

for: addiu  $t_1, t_1, zero$

#  $u = 0$

sltu  $t_3, t_1, t_2$

# Wenn  $u < m$  setze Register  $t_3 = 1$

addiu  $t_1, t_1, 1$

#  $u = u + 1$

BB<sub>1</sub>

bne  $t_3, zero, for$

# Beobach der Schleife

# Wenn  $t_3 \neq 0$  springe zu for

Schleife : While - Schleife

RISCV

while ( $n++ < m$ ) {

BB<sub>1</sub>

3

Annahm:  $n = \$t_1 \quad m = t_2$

while :  $sltu \quad t_3, \quad t_1, \quad t_2 \quad \# \text{ Wenn } n < m \text{ setze Register } t_3 = 1$   
 $addiu \quad t_1, \quad t_1, \quad 1 \quad \# n = n + 1$   
BB<sub>1</sub>  $\# \text{ Basisblock der Schleife}$   
bne  $t_3, \quad zero, \quad \text{while} \quad \# \text{ Wenn } t_3 \neq 0 \text{ springe zu While}$

Schaltanweisungen: Do - Schleife

RISCV

do §

BB<sub>1</sub>

3 while ( $n < m$ )

Anmerkung:  $n = t_1$   $m = t_2$

do:

BB<sub>1</sub>

sltu  $t_3, t_1, t_2$

addiu  $t_1, t_1, 1$

beq  $t_3, zero, do$

# Basisblock der Schleife

# Wenn  $n < m$  setze Register  $t_3 = 1$

#  $n = n + 1$

# Wenn Register  $t_3 \neq 0$  springe zu do

Zugriffe auf Felder:

lui  $t_0$

sw  $t_1, 5(t_0)$

lui  $t_0$

lw  $t_1, 5(t_0)$

RISCV

Oder als Makro:

.macro swArray (%w, %i, %a)

lui  $t_0, a$

sw  $w, i(t_0)$

.end

.macro lwArray (%w, %i, %a)

lui  $t_0, a$

lw  $w, i(t_0)$

.end

Beispiel von oben:

swArray (\$t1, \$s1, \$f1)

lwArray (\$t1, \$s1, \$f1)

## Felder in C

int Array[10]

/\* Ein Feld für 10 ganze Zahlen \*/

int UArray[]

/\* Ein Feld unbekannter Länge \*/

int \*pArray := Array[] /\* Ein Zeiger (Referenz) auf ein Feld \*/

## Datumsstrukturen in C

struct myStruct {

int a;

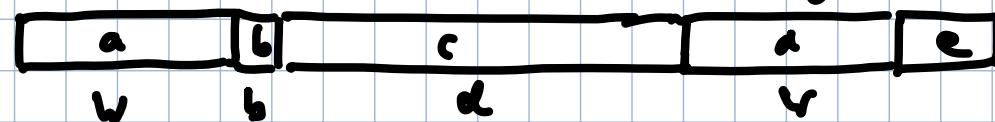
char b;

double c;

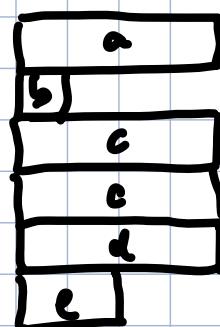
int d;

short e;

/\* Structs können auf Felder abgesetzt werden \*/



Layout := Spalten



.data

myStruct.a  
align 0  
myStruct.b  
myStruct.c

.word  
#aligned off  
:byte  
:double

myStruct.d  
align 2  
myStruct.e

.vark  
#align do v  
.half

## Indirekter Zugriff auf Felder:

MIPS

.data

wFeld: .word 0:3

# definire cldx (%i,%ba)

# Berechne die Adresse des Index

# %i = Index %a = Basisadresse des Feldes

la \$t0,ba

# Lade Adresse in \$t0

sll \$t1,i,2

# \$t1 = 4 \* (%i) , Wird abgerundet des Index

add \$t0,\$t0,\$t1 # Adresse des Feldes steht in \$t0

# definire lwldx (%i,%w)

# Laden des Feldwertes i

cldx (i,wFeld)

# Ermittle Adresse des Index i des Feldes

lw w,0(\$t0)

# definire swldx (%i,%w)

# Speichern des Feldwertes i

cldx (i,wFeld)

# Ermittle Adresse des Index des Feldes

sw w,0(\$t0)

## Indirekte Zugriff auf Felder:

.data

wFeld: .word 0:3

# definire cldx (%i,%ba)

# Berechne die Adresse des Index

# %i = Index %a = Basisadresse des Feldes

Defizit  
RISCV?

la t<sub>0</sub>,ba

# Lade Adresse in \$t<sub>0</sub>

sll t<sub>1</sub>,i,2

# t<sub>1</sub> = 4.(%i), Wird aligniert des Index

adrx t<sub>0</sub>, t<sub>1</sub>, t<sub>n</sub>

# Adresse des Feldes steht in \$t<sub>0</sub>

# definire lwdx (%i,%w)

# Laden des Feldwertes i

cldx (i,wFeld)

# Ermittle Adresse des Index i des Feldes

lw w,o(%t<sub>0</sub>)

# definire swlwdx (%i,%w)

# Speichern des Feldwertes i

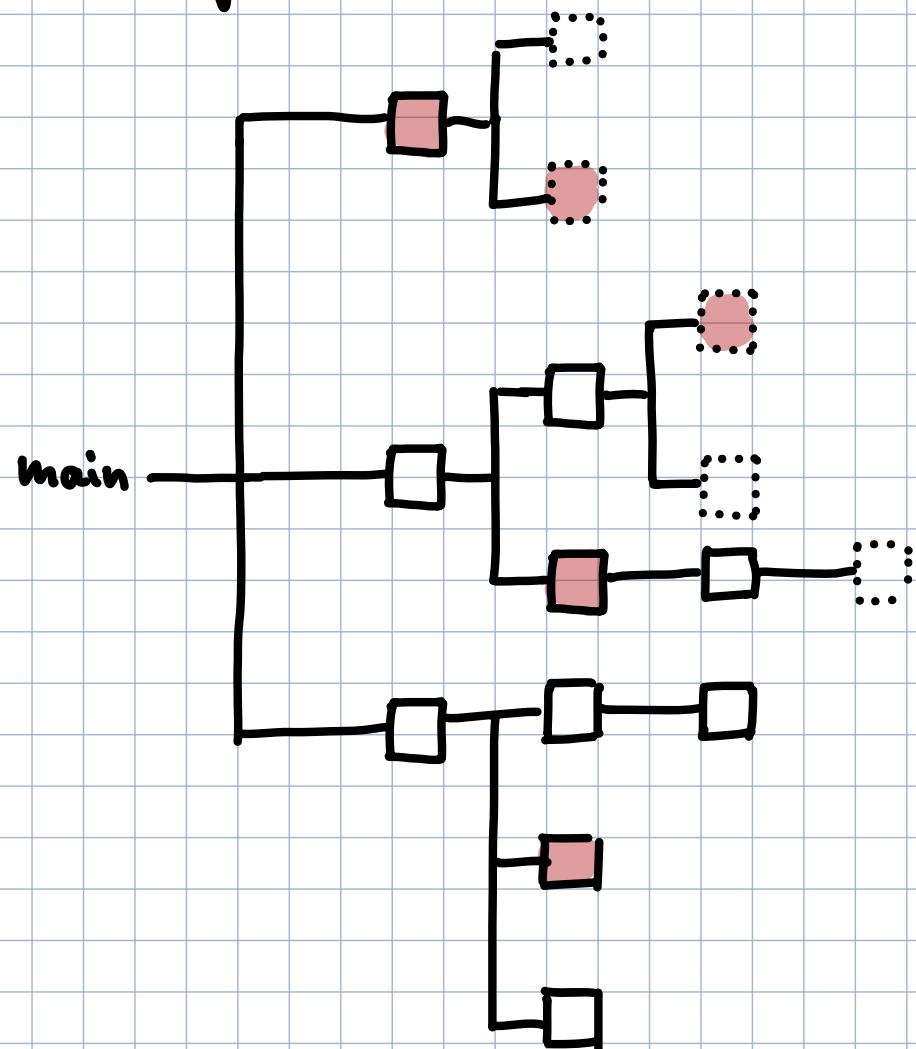
cldx (i,wFeld)

# Ermittle Adresse des Index i des Feldes

sw w,o(%t<sub>0</sub>)

# RISCV

## Unterprogramme



Call - Graph

non-leaf :



ohne lokale Daten



mit lokalen Daten



einzelne Blattroutine



Blattroutine mit Data

# Sprachergebnis für Unterprogramme (Funktionsaufrufe) Funktion des Registers

main

↳ func 1

↳ func 2

↳ func 3

0

.txt

.data

Heap

func

FFFF

Vereinigung des

Stacks in der  
aktuellen Funktion

Rahmen des  
Rückgragswerte

Heap

\$sp

\$fp

stackframe

# Registersatz des RISC-V : Belegungskonventionen

RISC-V

Register:

Name:

Description:

x 0	zero	
x 1	ra	Return address
x 2	sp	Stack pointer
x 3	gp	Global pointer
x 4	tp	Thread pointer
x 5-7	t0 - t2	Temporary registers
x 8	so / fp	Saved register / frame pointer
x 9	s1	Saved register
x 10-11	a0 - a1	Function arguments / return values
x 12-17	a2 - a7	Function arguments
x 18-27	s2 - s11	Saved registers
x 28-31	t3 - t6	Temporary registers

## Unterprogrammaufrufe

Was doen mit den Registern?

Übergabeparameter

$a_7 - a_1$

Temporäre Werte

$t_6 - t_0$

Rückgabewerte

$a_n - a_0$

}

Caller:

Diese Register werden von Aufrüfer auf den Stapel gesenkt!

Rückprungadresse

ra

Rahmenziger

fp / tp / gp

Sicherungsregister

$s_m - s_0$

}

Callee:

Diese Register werden vom Aufrufen gerichtet.

# RISC V

# Unterprogrammaufrufe ( Stapelrufen - stack frame )

RISC V

Caller:

subia sp, sp, 64 # Platz reservieren für 12 Variablen

sv a7, 60

:

sv a0, 28

# Argumente und Rückgabewerte sammeln

sw t6, 24

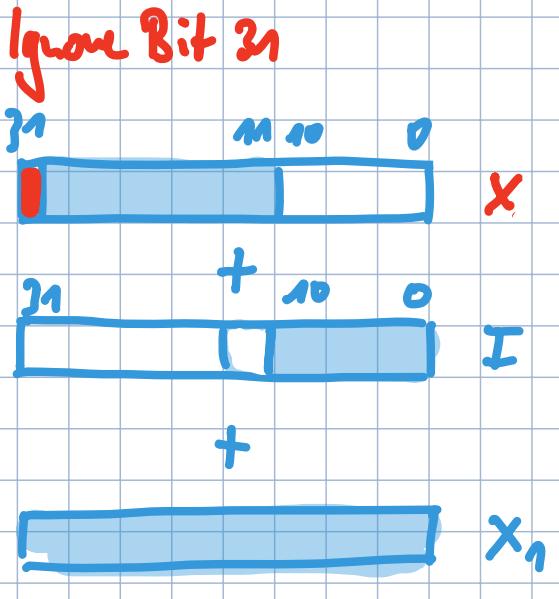
# Temporäre Variablen sammeln

:

sw t0, 0

jalr zero, 0(ar)

jal i(xn)



# Unterprogrammaufrufe (Stackframe - Stack frame)

Callee:

Aufruf:

func: Subiu

sp, sp, 64

sw ra, \$6 (\$sp)

sw fp, \$2 (\$sp)

sv gp, 48 (\$sp)

sw tp, 44 (\$sp)

sw \$u, 40 (\$sp)

⋮ ⋮

sw \$o, 0 (\$sp)

Subiu sp, \$sp, 24

sw zero, 20 (\$sp)

⋮ ⋮

sv zero, 0 (\$sp)

addiu f, s, zero

Lokale Variablen



Rücksprung

return: addiu sp, fp, zero

lw ra, \$6 (\$sp)

lw fp, \$2 (\$sp)

lw gp, 48 (\$sp)

lw tp, 44 (\$sp)

lw \$u, 40 (\$sp)

⋮ ⋮

lw \$o, 0 (\$sp)

addiu sp, sp, 24

jahr zero, 0(\$ar)

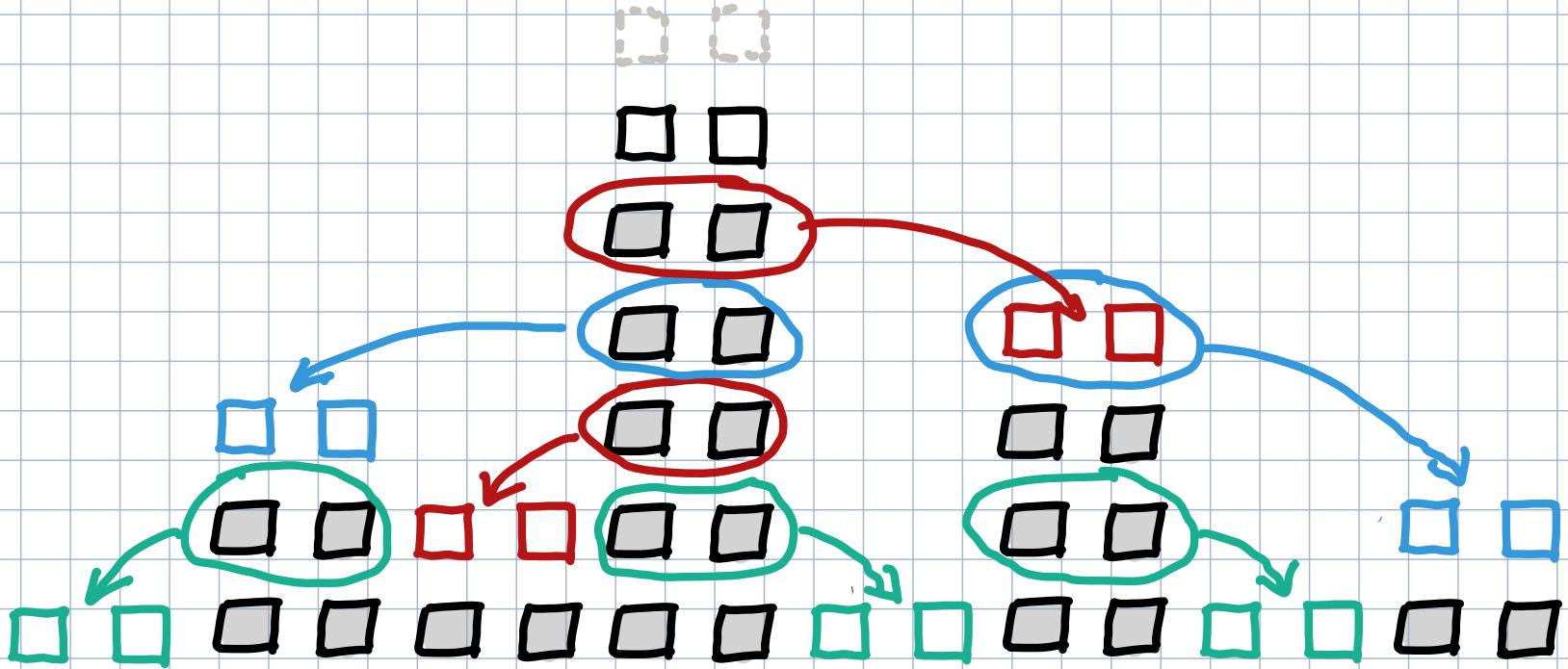
RISCV

## Beispiel: Fibonacci-Zahlen

□ : Kaninchen

■ : gesohlachtstüfes Kaninchen

- 0
- 1 Paar
- 1 Paar
- 2 Paare
- 3 Paare
- 5 Paare
- 8 Paare



$$f(n) = \begin{cases} 0 & n=0 \\ 1 & n=1 \\ f(n-1) + f(n-2) & \text{sonst} \end{cases}$$

$n=0$

$n=1$

sonst

## Rekursive Berechnung der n-ten Fibonacci-Zahl

```
int rFib ( int n ) {  
    switch ( n ) {  
        case 0: return 0;  
        case 1: return 1;  
        default : return rFib ( n - 1 ) + rFib ( n - 2 );  
    } /* switch */  
} /* rFib */
```

$$f(n) = \begin{cases} 0 & n=0 \\ 1 & n=1 \\ f(n-1) + f(n-2) & \text{sonst} \end{cases}$$

## Iterative Berechnung der n-ten Fibonacci-Zahl

```
int iFib (int n) {  
    int i=0;  
    int F[3];  
    if (n<1) return 0;  
    if (n<2) return 1;  
    F[i]=0  
    F[i++]=1  
  
    do {  
        F[2] = F[1] + F[0];  
        F[0] = F[1];  
        F[1] = F[2];  
    } while (i++ < n);  
    return F[2];  
} /* iFib */
```

$$f(n) = \begin{cases} 0 & n=0 \\ 1 & n=1 \\ f(n-1) + f(n-2) & \text{sonst} \end{cases}$$

## Berechting van n Fibonacci-tallen:

```
void nFib [int n, int F[]] {  
    i = 0;  
    F[i+] = 0;  
    if (n<1) return;  
    F[i+] = 1;  
    if (n<2) return;  
  
    do {  
        F[i+] = F[i-1] + F[i-2];  
    } while (i<n)  
}; /* nFib */
```

## Rekursive Berechnung der n-ten Fibonacci-Zahl (1)

addi ra zero done

addi a0 zero  $N \leftarrow N$  für N-te Fibonacci-Zahl } Startparameter initialisieren

fib:

beq a0 zero done

addi t2 zero 1

beq a0 t2 done

} Abdecken der Basisfälle

recd:

addi sp sp -12

sw ra 0(sp)

bne a0 zero return

---

sw a0 4(sp)

sw t2 8(sp)

jr rd

---

} Rekursionsabstieg  
für genug Speicher auf dem Stack

} Rekursionsaufstieg  
das eigentliche Ergebnis wird berechnet

## Rekursive Berechnung der n-ten Fibonacci-Zahl (2)

return:

addi a0 a0 -1

jal reca

lw t0 4(sp)

lw t1 8(sp)

add: sp sp 72

lw ra 0(sp)

sw t0 8(sp)

add a0 t0 t1

sw a0 4(sp)

jr ra

done:

} Rekursionsabstieg

für genug Speicher auf dem Stack

} Rekursionsaufstieg

das eigentliche Ergebnis wird berechnet

Die Wurzel einer Zahl berechnen: Mit der Brechstange RISC V

$$\sum_{i=1}^n (2i-1) = n^2$$

Beispiel:

i = 1 2 3 4 5

$$1+3+5+7+9=25$$

addi t<sub>2</sub>, zero, 2

#  $n^2 = n^2$

addiu t<sub>1</sub>, zero, zero

# i=0

for: addi t<sub>1</sub>, t<sub>1</sub>, 1

mul t<sub>3</sub>, t<sub>1</sub>, t<sub>2</sub>

subi t<sub>4</sub>, t<sub>3</sub>, 1

addu a<sub>0</sub>, a<sub>0</sub>, t<sub>4</sub>

bne a<sub>0</sub>, a<sub>2</sub>, for

Newton - Raphson - Verfahren:

$$t(h+x_n) = f(x_n) + f'(x_n) h$$

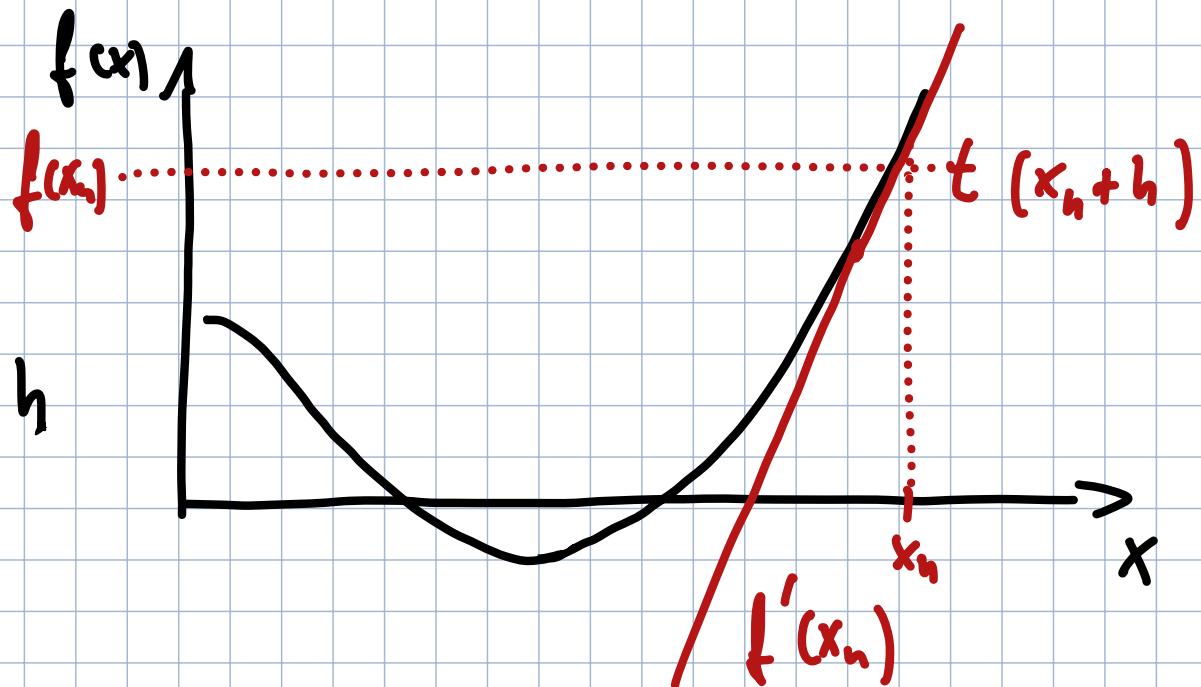
mit  $h = x - x_n$

$$\exists x_{n+1} : f(x_{n+1}) = 0 :$$

$$f(x_n) + f'(x_n)(x_{n+1} - x_n) = 0$$

$$f'(x_n)x_{n+1} - f'(x_n)x_n + f(x_n) = 0$$

$$f'(x_n)x_{n+1} = f'(x_n)x_n - f(x_n)$$



$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

# Berechnung des Quadratwurzel $\sqrt{a}$ : Newton-Raphson Verfahren

$$x^2 = a \quad x = \sqrt{a}$$

$$x^2 - a = 0 \Leftrightarrow f(x) = x^2 - a$$

$$f'(x) = 2x$$

$$x_{n+1} = x_n - \frac{x_n^2 - a}{2x_n}$$

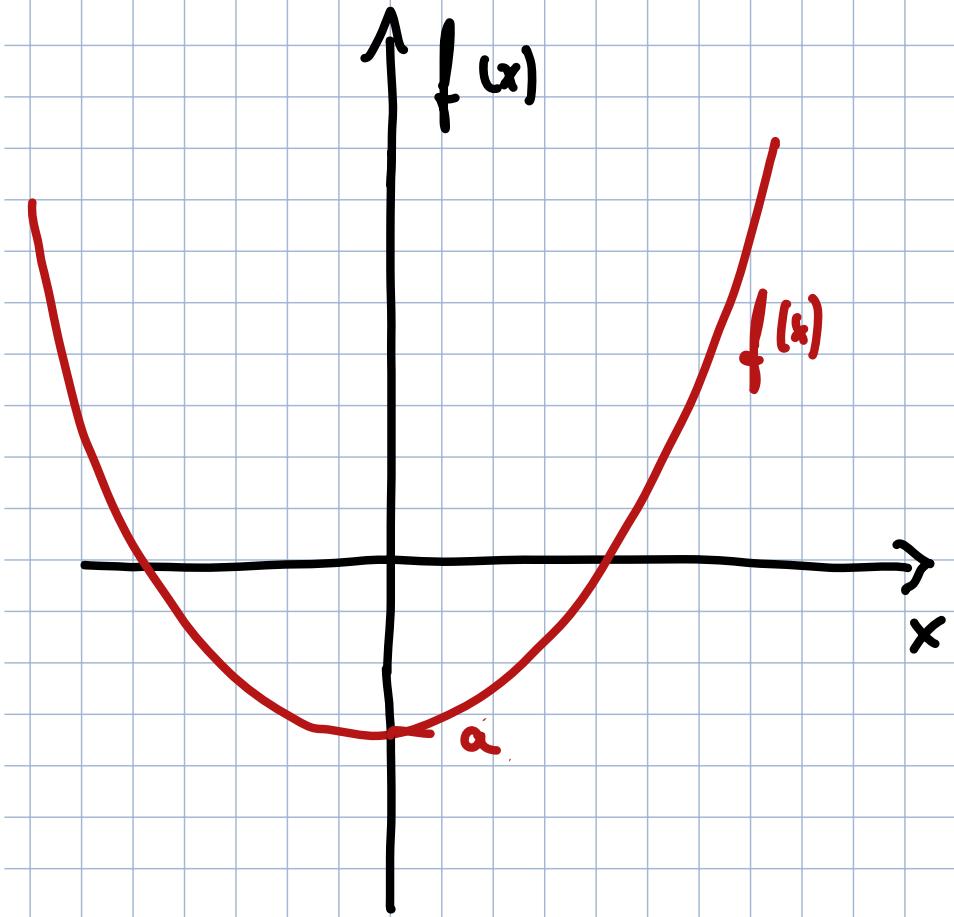
$$x_{n+1} = \frac{2x_n^2 - x_n^2 + a}{2x_n}$$

$$x_{n+1} = \frac{x_n^2 + a}{2x_n}$$

$$x_{n+1} = \frac{1}{2} \left( x_n + \frac{a}{x_n} \right)$$

$$x_0 \neq 0$$

Newton-Verfahren



Berechnung des Quadratwurzel  $\sqrt{a}$ :

$$x_{n+1} = \frac{1}{2} \left( x_n + \frac{a}{x_n} \right)$$

double sqrt (double a) {

    double x = a;

    double x1;

    do {

$x1 = x$

$x = 0.5 (x + a/x);$

    } while ( $x1 != x$ );

    return x;

} /\* double sqrt (double a) \*/

$$\sqrt{16} \quad a = 16$$

## Berechnung des Quadratwurzel $\sqrt{a^t}$ :

Beispiel:

2

1,5

1,416667

1,414216

1,414214

16

8,5

5,191176

4,136665

4,002258

4,000001

4,000000

25

13,000000

7,461538

5,406027

5,014298

5,000023

5,000000

## 32 Floating point extension registers

RISCV

f <sub>0</sub> -f <sub>7</sub>	f <sub>60-7</sub>	floating point temporaries registers	caller
f <sub>8</sub> -f <sub>15</sub>	f <sub>50-1</sub>	floating point Saved registers	callee
f <sub>16</sub> -f <sub>23</sub>	f <sub>40-1</sub>	floating point arguments / return values	caller
f <sub>24</sub> -f <sub>31</sub>	f <sub>30-7</sub>	floating point arguments registers	caller
f <sub>32</sub> -f <sub>39</sub>	f <sub>20-11</sub>	floating point Saved registers	caller
f <sub>40</sub> -f <sub>47</sub>	f <sub>10-11</sub>	floating point temporaries registers	caller

F-Extension : 32-Bit floating point numbers

D-Extension : 64-Bit floating point numbers

# Unterprogramm Wurzel : Stack frame und Initialisierung

RISCV

Sqvt:

Subiu

sp, sp, 8

# Stack frame

sv

ra, 4 (sp)

su

fp, 0 (sp)

fcvt.vu.d

ft<sub>0</sub>, zero

# Initialisierung

addiu

t<sub>0</sub>, t<sub>0</sub>, 2

# ft<sub>0</sub>=0 ft<sub>2</sub>=2

fcvt.vu.d

ft<sub>2</sub>, t<sub>0</sub>

# Unterprogramm Wurzel : Hauptteil

RISCV

```
#double sqrt (double a) {
    double x = a;
    double x1;
```

```
do: fadd.d ft11, fa2, ft0
    fdiv.d ft3, fa2, fa0
    fadd.d ft4, ft3, fa0
    fdiv.d fa0, ft4, ft2
    feq.d fa0, ft1, t0
    beqz t0, do
```

# do {  
# #  
# } while (x<sub>1</sub> != x);  
# return x;  
# /\*double sqrt (double a) \*/

$x_1 = x \cdot \frac{ft_2}{ft_3}$   
 $x = 0.5 \left( \underbrace{x + a/x}_{ft_4} \right);$

## Unterprogramm Wurzel : Stack frame

RISC V

lv

ra, 4 (sp)

# Stack frame

lu

fP, 0 (sp)

subiu

sp, sp, 8

jalr

zero, 0 (ra)

Allgemeine :

fsqrt.l a0,



## Schablone: Verzweigung

if ( CB ) {

BB<sub>1</sub>;

} else {

BB<sub>2</sub>;

}

cb: beq \$t<sub>1</sub>, \$t<sub>2</sub>, if

CB = Kontrollblock

BB = Basisblock

MIPS

if :

BB<sub>1</sub>

j fi

else : BB

fi :

bne \$t<sub>1</sub>, \$t<sub>2</sub>, else

Schablone : for-Schleife

for ( $u \geq 0; u < m; u++$ ) {

BB<sub>1</sub>

3

MIPS

Annahm:  $u = \$t_1$      $m = \$t_2$

for: addiu \$t<sub>1</sub>, \$t<sub>1</sub>, \$zero

#  $u = 0$

sltu \$t<sub>3</sub>, \$t<sub>1</sub>, \$t<sub>2</sub>

# Wenn  $u < m$  setze Register  $\$t_3 = 1$

addiu \$t<sub>1</sub>, \$t<sub>1</sub>, 1

#  $u = u + 1$

BB<sub>1</sub>

bne \$t<sub>3</sub>, \$zero, for

# Beendeblock der Schleife

# Wenn  $\$t_3 \neq 0$  springe zu for

## Schachbrett : While - Schleife

MIPS

while ( $n++ < m$ ) {

BB<sub>1</sub>

3

Annahme:  $n = \$t_1 \quad m = \$t_2$

while :  $sltu \$t_3, \$t_1, \$t_2$  # Wenn  $n < m$  setze Register  $\$t_3 = 1$   
 $addiu \$t_1, \$t_1, 1$  #  $n = n + 1$   
BB<sub>1</sub> # Basisblock der Schleife  
bne  $\$t_3, \$zero$ , while # Wenn  $\$t_3 \neq 0$  springe zu While

Schaltan : Do - Schleife

MIPS

do {

BB<sub>1</sub>

3 while ( $n < m$ )

Annehmen:  $n = \$t_1$     $m = \$t_2$

do:

BB<sub>1</sub>

sltu \$t<sub>3</sub>, \$t<sub>1</sub>, \$t<sub>2</sub>

addiu \$t<sub>1</sub>, \$t<sub>1</sub>, 1

beq \$t<sub>3</sub>, \$zero, do

# Basisblock der Schleife

# Wenn  $n < m$  setze Register  $\$t_3 = 1$

#  $n = n + 1$

# Wenn Register  $\$t_3 \neq 0$  springe zu do

## Felder in Assembly:

MIPS

.data

# Feld aus 4·32 Bit (4 Wörter)

wArray: .word 0:3

# Feld von Bytes (VINT8) in Big-Endian Reihenfolge

bArrayBE: .byte 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

# Feld von Bytes (VINT8) in Little-Endian Reihenfolge

bArrayLE: 15,14,13,12,11,10,9,8,7,6,5,4,3,2,1,0

# Feld einer Zeichenfolge (String)

str: .ascii "Hello world"

str0: .ascii3 "Hello world" # String mit 0 am Ende 10

Zugriffe auf Felder:

lui \$t<sub>0</sub>

sw \$t<sub>1</sub>, 5(\$t<sub>0</sub>)

lui \$t<sub>0</sub>

lw \$t<sub>1</sub>, 5(\$t<sub>0</sub>)

MIPS

Oder als Makro:

.macro swArray (%w, %i, %a)

lui \$at,a

sw w, i(\$at)

.endmacro

.macro lwArray (%w, %i, %a)

lui \$at,a

lw w, i(\$at)

.endmacro

Beispiel von oben:

swArray (\$t<sub>1</sub>, \$t<sub>0</sub>, F)

lw Array (\$t<sub>1</sub>, \$t<sub>0</sub>, F)

# Registersatz des MIPS : Belegungskonventionen

MIPS

Register:

Name:

Description:

0

\$zero

1

\$at

assembler temporary

2 - 3

\$v<sub>0</sub> - \$v<sub>1</sub>

values for function results

4 - 7

\$a<sub>0</sub> - \$a<sub>3</sub>

arguments

8 - 15

\$t<sub>0</sub> - \$t<sub>7</sub>

temporaries

16 - 23

\$s<sub>0</sub> - \$s<sub>7</sub>

saved temporaries

24 - 25

\$t<sub>8</sub> - \$t<sub>9</sub>

temporaries

26 - 27

\$k<sub>0</sub> - \$k<sub>1</sub>

reserved for OS kernel

28

\$gp

global pointer

29

\$sp

stack pointer

30

\$fp

frame pointer

31

\$ra

return address

## Unterprogrammaufrufe

MIPS

Was doen mit den Registern?

Übergabeparameter

\$a<sub>3</sub> - \$a<sub>0</sub>

Temporäre Werte

\$t<sub>7</sub> - \$t<sub>0</sub>

Rückgabewerte

\$v<sub>1</sub> - \$v<sub>0</sub>



Caller:

Diese Register werden von  
Aufrüfer auf den Stapel  
gesetzt!

Rückprungadresse

\$ra

Rahmenregister

\$fp

Silbenregister

\$s<sub>1</sub> - \$s<sub>0</sub>



Called:

Diese Register werden von  
Aufrufen gesetzt.

Unterprogrammaufrufe

( Stapelrahmen - Stack frame )

MIPS

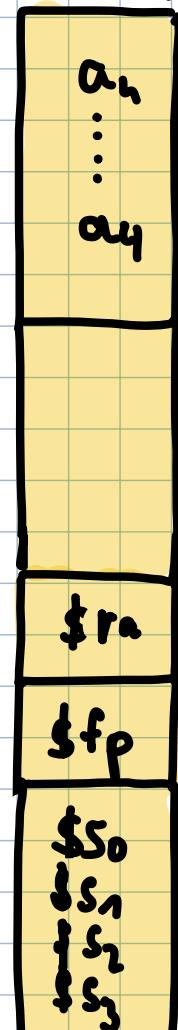
Caller closure

Registers

\$a<sub>3</sub>  
\$a<sub>2</sub>  
\$a<sub>1</sub>  
\$a<sub>0</sub>

Caller data

\$fp  
\$sp



reserved

\$ra

\$fp

\$s<sub>0</sub>

\$s<sub>1</sub>

\$s<sub>2</sub>

\$s<sub>3</sub>

local  
data

} Local use of stack

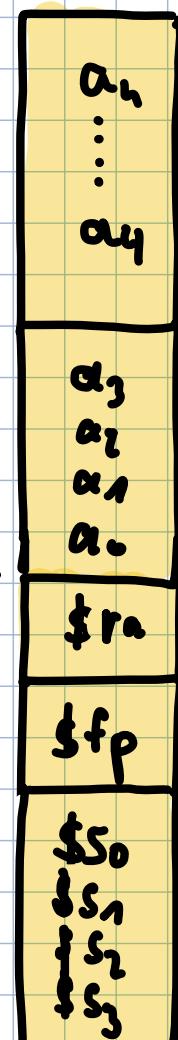
Arguments

Caller save

on the stack

Caller save

Save registers



\$fp  
\$sp

## Unterprogrammaufrufe ( Stapelrufen - stack frame )

MIPS

Caller:

sub	\$sp, \$sp, 56	# Platz reservieren 4.12 Variablen
sw	\$a <sub>3</sub> , 52(\$sp)	
:	:	# Argumente siedeln
sw	\$a <sub>0</sub> , 44(\$sp)	
sw	\$t <sub>7</sub> , 40(\$sp)	# Temporäre Variablen siedeln
:	:	
sw	\$t <sub>0</sub> , 8(\$sp)	
sw	\$v <sub>0</sub> , 4(\$sp)	# Rückgabewerte siedeln
sw	\$v <sub>1</sub> , 0(\$sp)	
jal	funcf	

# Unterprogrammaufrufe (Stackframe - Stack frame)

Callee:

Aufruf:

func: Subiu \$sp, \$sp, 24  
 sw \$ra, 20(\$sp)  
 sw \$fp, 16(\$sp)  
 sw \$s3, 12(\$sp)  
 :  
 sw \$s0, 0(\$sp)  
 Subiu \$sp, \$sp, 24  
 sw \$zero, 20(\$sp)  
 :  
 sw \$zero, 0(\$sp)  
 addiu \$fp, \$sp, \$zero

(lokale  
Variablen)

Rücksprung

return: addiu \$sp, \$fp, \$zero  
 lw \$ra, 20(\$sp)  
 lw \$fp, 16(\$sp)  
 lw \$s3, 12(\$sp)  
 :  
 lw \$s0, 0(\$sp)  
 addiu \$sp, \$sp, 24  
 jr \$ra

MIPS

Berechnung von n- Fibonacci-Zahlen spezifisch in Liste (Aufgabe)

.data

n .word

F .word 0:h

MIPS

Ausgabe: 1. Register ist h

2. Register ist F

.txt

# Programm

nFib:

subi \$sp, \$sp, 24

# Stapelrahmen

sw \$ra, 20(\$sp)

sw \$fp, 16(\$sp)

sw \$s0, 12(\$sp)

sw \$s1, 8(\$sp)

sw \$s2, 4(\$sp)

sw \$s3, 0(\$sp)

# Berechnung von n-Fibonacci-Zahlen speziell in Liste (Bewerfung)

MIPS

lw \$a0, n

addiu \$s0, \$zero, 0

addiu \$t1, \$zero, 1

sublx (0, \$s0, F)

beq \$a0, \$s0, returnb

addiu \$s0, \$s0, 1

sublx (1, \$s0, F)

beq \$a0, \$s0, returnb

addiu \$s0, \$s0, 1

# \$a0 = n

# i=0

# F[0] = 0

# n = i

# i = 0 + 1 = 1

# F[1] = 1

# i = 1 + 1 = 2

MIPS

Berechnung von n-Fibonacci-Zahlen speziell in Liste  
(Brenten-Tortkay)

nFib-do:  
subiu \$t<sub>5</sub>, \$S<sub>0</sub>, 1  
lwldx(\$S<sub>2</sub>, \$t<sub>5</sub>, F)  
subiu \$t<sub>5</sub>, \$S<sub>0</sub>, 2  
lwldx(\$S<sub>3</sub>, \$t<sub>5</sub>, F)  
add \$t<sub>4</sub>, \$S<sub>2</sub>, \$<sub>3</sub>  
srldx (\$t<sub>4</sub>, \$S<sub>0</sub>, F)  
sltu \$t<sub>3</sub>, \$S<sub>0</sub>, \$ad  
addiu \$S<sub>0</sub>, \$S<sub>0</sub>, 1  
bne \$t<sub>3</sub>, \$zero, nFib-do

# \$t<sub>5</sub> = i - 1  
# \$S<sub>2</sub> = F[i - 1]  
# \$t<sub>5</sub> = i - 2  
# \$S<sub>3</sub> = F[i - 2]  
# \$t<sub>4</sub> = F[i - 1] + F[i - 2]  
# F[i] = \$t<sub>4</sub>  
# t<sub>3</sub> = 1 wenn i < n

Berechnung von n- Fibonacci-Zahlen spezifisch im Linker (Rückspur)

MIPS

return: lw \$ra, 20(\$sp)  
lw \$fp, 16(\$sp)  
lw \$s0, 12(\$sp)  
lw \$s1, 8(\$sp)  
lw \$s2, 4(\$sp)  
lw \$s3, 0(\$sp)  
addi \$sp, \$sp, 24  
jr \$ra

# Assembler-Programm Fibonacci-Zahlen Iterativ (Auftrag)

MIPS

/\* push StackFrame \*/

# Caller

/\* function prologue: fib does not call any function, fp not used \*/

hFib: subiu \$sp, \$sp, 16

sw \$ra, 12(\$sp)

sw \$s2, 8(\$sp)

sw \$s1, 4(\$sp)

sw \$s0, 0(\$sp)

# Assembler - Programm Fibonacci - Zahlen Iterativ (initializing)

MIPS

```
/* $a0=1 , $a1=F[0] $S1=Fib1 */  
/* $S0=i . $S2=Fib2 */  
/* Initialising der Fibonacci - Routine */  
addiu $S0, $zero, 0          # i<0  
sw $S0, 0, F               # F[0]=0  
bgeq $a0, $S0, returnf  
addiu $S0, $S0, 1          # i=0+1=1  
sw $S0, 1, F               # F[1]=1  
bgeq $a0, $S0, returnf  
addiu $S0, $S0, 1          # i=1+1=2  
addiu $S1, $zero, 0          # $S1=0  
addiu $S2, $zero, 1          # $S2=1
```

## Assembler-Programm Fibonacci-Zahlen iterativ (Berechnung)

MIPS

$\$S_0$	$i$
$\$S_1$	$F[i-1]$
$\$S_2$	$F[i-2]$

/\* Berechnung der Fibonacci-Zahl \*/

do:	addu $\$t_0, \$S_0, \$S_1$	# Berechne nächste Fibonacci-Zahl
	swAmy ( $\$t_0, \$S_0, F$ )	# $F[i] = t_0$
	addu $\$S_2, \$zero, \$S_1$	# $\$S_2 = \$S_1$
	addu $\$S_1, \$zero, \$t_0$	# $\$S_1 = \$t_0 = F[i]$
	sltu $\$t_0, \$S_0, \$a0$	# $\$t_0 = 1$ wenn $i < n$
	addiu $\$S_0, \$S_0, 1$	# $i = i + 1$
	bne $\$t_0, \$zero, lab$	# do wenn $\$t_0 \neq 1$

# Assembler-Programm Fibonacci-Zahlen Iterativ ( Rücksprung )

MIPS

/\* popStackFrame und Rückkehr aus der Routine \*/

/\* Epilog of the function \*/

return: lw \$s<sub>0</sub>, 0(\$sp)

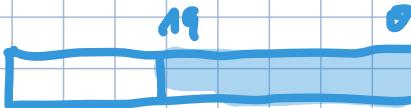
lw \$s<sub>0</sub>, 4(\$sp)

lw \$s<sub>2</sub>, 8(\$sp)

lw \$ra, 12(\$sp)

addiu \$sp, \$sp, 16

jr \$ra



## Mutterprogramm Aufruf (Stack frame)

MIPS

funktion:  
subu \$sp, 28  
sw \$ra, 24(\$sp)  
sw \$fp, 20(\$sp)  
sw \$so, 16(\$sp)  
sw \$s1, 12(\$sp)  
sw \$s2, 8(\$sp)  
sw \$s3, 4(\$sp)  
sw \$s4, 0(\$sp)  
adiu \$fp, \$sp, \$zero

# Reserve 7 Wörter auf dem Stack  
# Sichere die Rücksprungadresse des Callers  
# Sichere den Return Register  
#lege alle zu sicheren Register  
# des Callers auf den Stack  
# Set frame pointer

Unterprogramm aufmf : Rückkehr (stack free)

MIPS

return: addiu \$t,\$v,\$zero  
lw \$ra, 24(\$sp)  
lw \$fp, 20(\$sp)  
lw \$s0, 16(\$sp)  
lw \$s1, 12(\$sp)  
lw \$s2, 8(\$sp)  
lw \$s3, 4(\$sp)  
lw \$s4, 0(\$sp)  
addiu \$sp,\$sp, 28  
jr \$ra

Füllinsaufen: Inhalt der Temporären Register sind Argumente

MIPS

addiu \$a0,\$t0,\$zero	int a
addiu \$a1,\$t1,\$zero	int b
addiu \$a2,\$t2,\$zero	int c
addiu \$a3,\$t3,\$zero	int d

# Weitere Argumente kommen auf den Stapel:

subu \$sp,\$fp,4  
sw \$t5,0(\$sp)

# Assembler - Programm Fibonacci - Zahlen Rekursiv ( Aufruf )

MIPS

rFib:

- shllu \$sp, \$sp, 12
- sw \$ra, 8 (\$sp)
- sw \$fp, 4 (\$sp)
- sw \$so, 0 (\$sp)
- addu \$so, \$lo, \$zero
- addu \$fp, \$sp, \$zero

- # Rücken von Speicherplatz auf der Stack
- # Sichern des Rückgangs auf der Stack
- # Sichern des ehem. Src - Registers
- # n wird gesetzt
- # wird eigentlich nicht gebraucht

# Assembler-Programm Fibonacci-Zahlen Rekursiv (Berechnung)

MIPS

addiu \$V\_0, \$zero, 0

#  $V_0 = n$ ,  $V_1 = 0$

beq \$a\_0, \$two, return

# Verlasse 0, wenn  $n=0$

addiu \$V\_0, \$V\_0, 1

#  $V_1 = 1$

beq \$a\_0, \$to, return

# Verlasse 1, wenn  $n=1$

subiu \$a\_0, \$a\_0, 1

#  $a_0 - 1$  ist neues  $n$

call  
save

subiu \$sp, \$sp, 8

# lege  $n-1$

sv  $a_0, 4(\$sp)$

# auf den Stack

sv  $V_0, 0(\$sp)$

# ACHTUNG:  $V_0 = 1$

jal  $\nu Fib$

#  $\nu Fib$

# Assembler - Programm Fibonacci - Zahlen Rekursiv ( Berechnung - Fortsetzung )

MIPS

lw \$a<sub>0</sub>, 4(\$sp)

# n holen

sub \$a<sub>0</sub>, \$a<sub>0</sub>, 2

# n-2

sub \$sp, \$sp, 4

sv \$v<sub>0</sub>, 0(\$sp)

# Ergebnis von rFib(n-1) sfern

jal rFib

# rFib(n-2)

lw \$t<sub>0</sub>, 0(\$sp)

# Ergebnis rFib(n-1)

add \$v<sub>0</sub>, \$v<sub>0</sub>, \$t<sub>0</sub>

# rFib(n-1) + rFib(n-2)

add \$sp, \$fp, \$zero

# lokale Sichtung vergessen

# Assembler-Programm Fibonacci-Zahlen Rekursiv (Rückprung)

MIPS

return: addu \$sp, \$fp, \$zero

lw \$s0, 0(\$sp)

lw \$fp, 4(\$sp)

lw \$ra, 8(\$sp)

addu \$sp, \$fp, 8

jr \$ra

# Wird eigentlich nicht gebraucht

# Lokale Daten ignorieren

# Wiederherstellen des Stack-Registers

# Wiederherstellen des Frame-Registers

# und des Rücksprung-Adressen

# Stack freigeben

# Rückprung