

# Technische Informatik 1 – Übung 2

## Assembler (Rechenübung)

*Sarah Hoffmann*  
*20.-21. Oktober 2011*



# Ziele der Übung

## Aufgabe 1

- Aufbau und Aufruf von Funktionen in Assembler
- Codeanalyse

## Aufgabe 2

- Aufbau und Manipulation des Stacks
- Schreiben einer vollständigen Assembler-Funktion

# Voraussetzungen

- grundlegende Assembler-Kenntnisse  
(Vorlesung Kapitel 2)
- Vorlesung Kapitel 3: Funktionen und Stack  
(v.a. Folien 8-18)

# Aufgabe 1.1: Wurzelverfahren

- Implementierung von Heron

- Eingabe:             $a$  ... Zahl, von der Wurzel ermittelt werden soll  
                          $n$  ... Anzahl Iterationen

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

- Ausgabe             $x$  ... Ergebnis

- keine Stackverwaltung

- Sicherung der Register vereinfacht

# Aufgabe 1.1: Zu Implementierung 2

heron:

###

sw \$f20, 24(\$sp)

move \$f20, \$f12

sw \$ra, 16(\$sp)

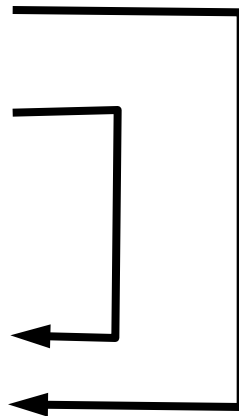
...

lw \$ra, 16(\$sp)

lw \$f20, 24(\$sp)

###

jr \$ra



Sichern der Register

Wiederherstellen der Register

- Register werden **vor** Verwendung gesichert, um sie am Ende der Funktion wieder in den Originalzustand zu versetzen

# Lösung 1.1: Implementierung 1

heron:

###

```

move    $f2, $f12      # $f2 := $f12
move    $t1, $0        # $t1 := 0
ble     $a1, $0, $Label3 # if ($a1 <= $0) then goto $Label3
li      $f4, 5.0e-1    # $f4 := 0.5

```

} Initialisierung

\$Label 5:

```

div     $f0, $f12, $f2 # $f0 := $f12 / $f2
add     $f0, $f2, $f0  # $f0 := $f0 + $f2
mul     $f0, $f0, $f4  # $f0 := f0 * $f4

```

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

```
addi $t1, $t1, 1
```

oder: *addi \$a1, \$a1, -1*

Schleifenzähler

```

slt     $t0, $t1, $a1  # if ($t1 < $a1) then $t0 := 1
                        # else $t0 := 0
move    $f2, $f0       # $f2 := $f0
bne     $t0, $0, $Label5 # if ($t0 <> $0) then goto $Label5

```

} while (\$t1 < n) ...

\$Label 3:

```

move    $f0, $f2      # $f0 := $f2
###
jr      $ra            # Jump and return ($pc := $ra)

```

} Finalisierung

# Lösung 1.1: Implementierung 2 (main)

```
void main()  
{  
    float sqrt2 = heron(2, 10);  
}
```

```
main:  
    ###  
    li    $f12, 2.0e0  
    li    $a1, 10  
    jal   heron  
    ###
```

# Lösung 1.1: Implementierung 2 (heron)

```
float heron(float a, int n) {
```

```
    // Sichern der Register
```

```
heron:
```

```
    ###
```

```
    sw    $f20, 24($sp)
```

```
    move $f20, $f12
```

```
    sw    $ra, 16($sp)
```

```
    bgt   $a1, $0, $Label 2
```

```
    move  $f0, $f20
```

```
    j     $Label 4
```

```
$Label 2:
```

```
    move  $f12, $f20
```

```
    addi  $a1, $a1, -1
```

```
    jal   heron
```

```
    di v  $f2, $f20, $f0
```

```
    add   $f2, $f2, $f0
```

```
    li    $f0, 5.0e-1
```

```
    mul   $f2, $f2, $f0
```

```
    move  $f0, $f2
```

```
$Label 4:
```

```
    lw    $ra, 16($sp)
```

```
    lw    $f20, 24($sp)
```

```
    ###
```

```
    jr   $ra
```

```
    // Wiederherstellen der Register
```

```
}
```

Reg.	Variable
a1	n
f0	result
f2	
f12	a
f20	<b>a</b>

# Lösung 1.1: Implementierung 2 (heron)

```
float heron(float a, int n) {
```

```
    // Sichern der Register
```

```
    if (n <= 0) {
```

```
    } else {
```

```
    }  
    // Wiederherstellen der Register
```

```
}
```

```
heron:
```

```
###
```

```
sw    $f20, 24($sp)
```

```
move  $f20, $f12
```

```
sw    $ra, 16($sp)
```

```
bgt  $a1, $0, $Label 2
```

```
move  $f0, $f20
```

```
j    $Label 4
```

```
$Label 2:
```

```
move  $f12, $f20
```

```
addi  $a1, $a1, -1
```

```
jal   heron
```

```
div   $f2, $f20, $f0
```

```
add   $f2, $f2, $f0
```

```
li    $f0, 5.0e-1
```

```
mul   $f2, $f2, $f0
```

```
move  $f0, $f2
```

```
$Label 4:
```

```
lw    $ra, 16($sp)
```

```
lw    $f20, 24($sp)
```

```
###
```

```
jr    $ra
```

Reg.	Variable
a1	n
f0	result
f2	
f12	a
f20	a

# Lösung 1.1: Implementierung 2 (heron)

```
float heron(float a, int n) {
```

```
    // Sichern der Register
```

```
    if (n <= 0) {
        return a;
```

```
    } else {
```

```
heron:
```

```
    ###
```

```
    sw    $f20, 24($sp)
```

```
    move  $f20, $f12
```

```
    sw    $ra, 16($sp)
```

```
    bgt   $a1, $0, $Label 2
```

```
    move  $f0, $f20
```

```
    j     $Label 4
```

```
$Label 2:
```

```
    move  $f12, $f20
```

```
    addi  $a1, $a1, -1
```

```
    jal   heron
```

```
    div   $f2, $f20, $f0
```

```
    add   $f2, $f2, $f0
```

```
    li    $f0, 5.0e-1
```

```
    mul   $f2, $f2, $f0
```

```
    move  $f0, $f2
```

```
$Label 4:
```

```
    lw    $ra, 16($sp)
```

```
    lw    $f20, 24($sp)
```

```
    ###
```

```
    jr    $ra
```

```
}
```

```
    // Wiederherstellen der Register
```

Reg.	Variable
a1	n
f0	result
f2	
f12	a
f20	a

# Lösung 1.1: Implementierung 2 (heron)

```
float heron(float a, int n) {
    // Sichern der Register

    if (n <= 0) {
        return a;
    } else {
        float prev_res = heron(a, n-1);

    }
    // Wiederherstellen der Register
}
```

```
heron:
    ###
    sw    $f20, 24($sp)
    move  $f20, $f12
    sw    $ra, 16($sp)
    bgt   $a1, $0, $Label 2
    move  $f0, $f20
    j     $Label 4
$Label 2:
    move  $f12, $f20
    addi  $a1, $a1, -1
    jal   heron
    div   $f2, $f20, $f0
    add   $f2, $f2, $f0
    li    $f0, 5.0e-1
    mul   $f2, $f2, $f0
    move  $f0, $f2
$Label 4:
    lw    $ra, 16($sp)
    lw    $f20, 24($sp)
    ###
    jr    $ra
```

Reg.	Variable
a1	n
f0	result/ prev_res
f2	
f12	a
f20	a

# Lösung 1.1: Implementierung 2 (heron)

```
float heron(float a, int n) {
    // Sichern der Register

    if (n <= 0) {
        return a;
    } else {
        float prev_res = heron(a, n-1);

        return (0.5*(prev_res
                    + a/prev_res));
    }
    // Wiederherstellen der Register
}
```

```
heron:
    ###
    sw    $f20, 24($sp)
    move  $f20, $f12
    sw    $ra, 16($sp)
    bgt   $a1, $0, $Label 2
    move  $f0, $f20
    j     $Label 4
$Label 2:
    move  $f12, $f20
    addi  $a1, $a1, -1
    jal   heron
    div   $f2, $f20, $f0
    add   $f2, $f2, $f0
    li    $f0, 5.0e-1
    mul   $f2, $f2, $f0
    move  $f0, $f2
$Label 4:
    lw    $ra, 16($sp)
    lw    $f20, 24($sp)
    ###
    jr    $ra
```

Reg.	Variable
a1	n
f0	result/ prev_res
f2	<i>temp</i>
f12	a
f20	a

# Lösung 1.1: Effizienz

**Implementierung 1: Iteration (Schleifen)**

**Implementierung 2: Rekursion**

Schleifen sind schneller:

- Aufbau des Stacks weniger aufwändig
- weniger Instruktionen (Register effektiver ausgenutzt)

## Aufgabe 1.2: Fibonacci-Zahlen

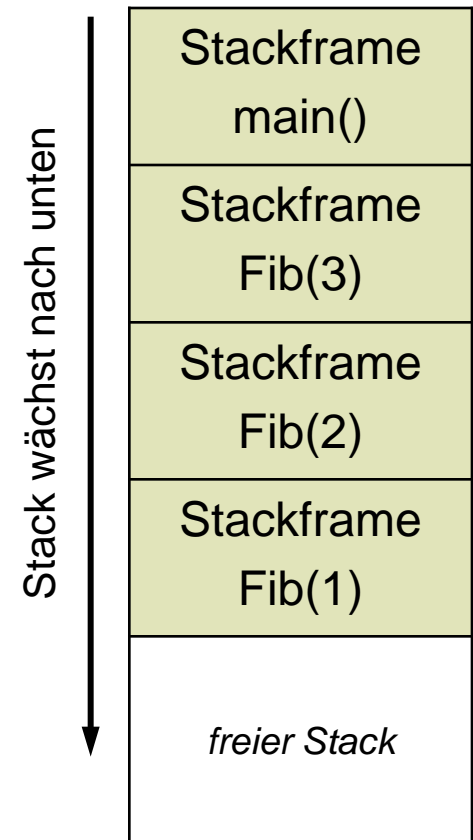
$$Fib(n) = Fib(n-1) + Fib(n-2)$$

$$Fib(1) = Fib(2) = 1$$

- Funktion mit Parameterübergabe via Stack

# Aufgabe 1.2: Der Stack

- Sichern von Registerinhalten und funktionslokalen Daten im Speicher
- wegen verschachtelter Funktionsaufrufe keine absolute Adresse im Speicher



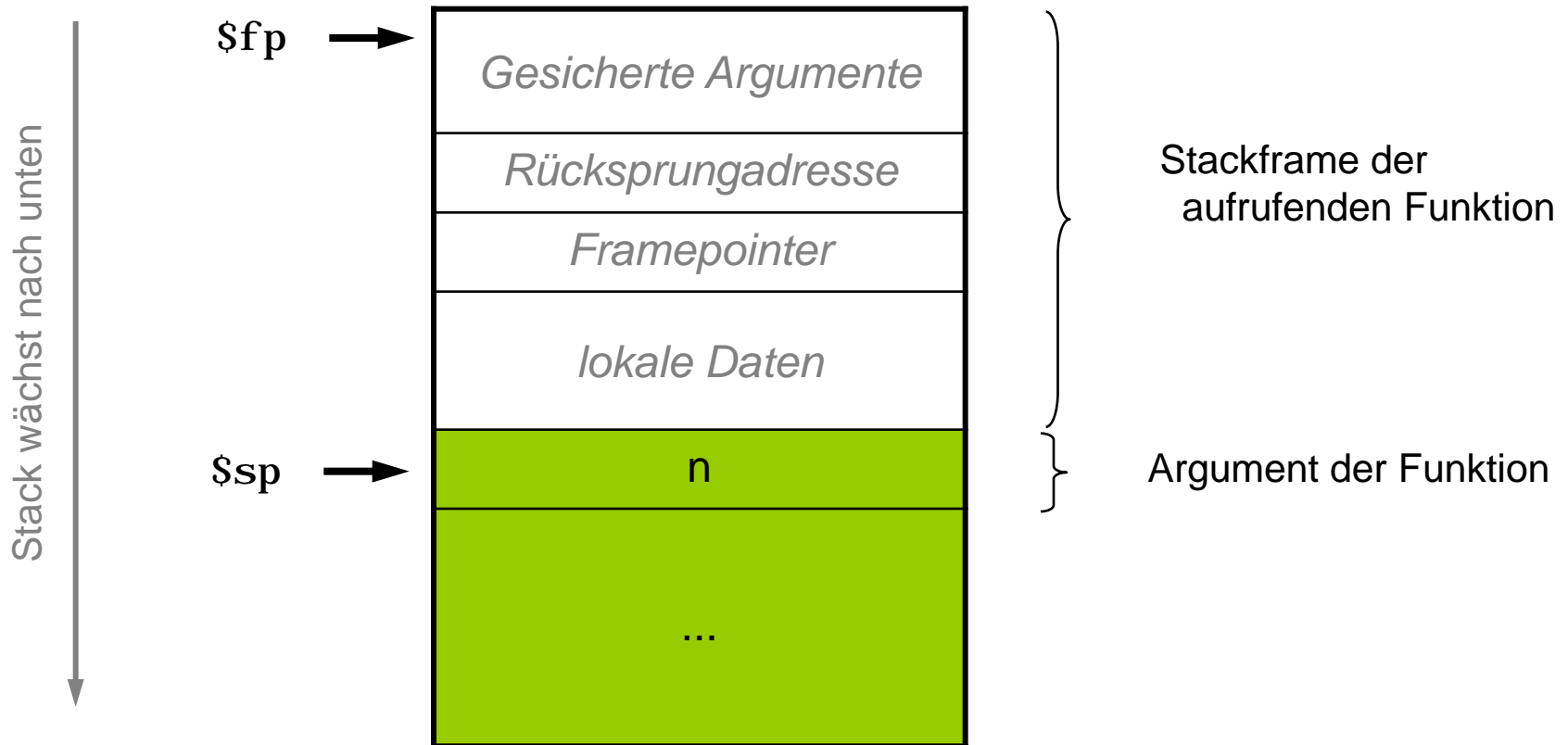
Adresse 0 →

## Aufgabe 1.2: Sichern der Register

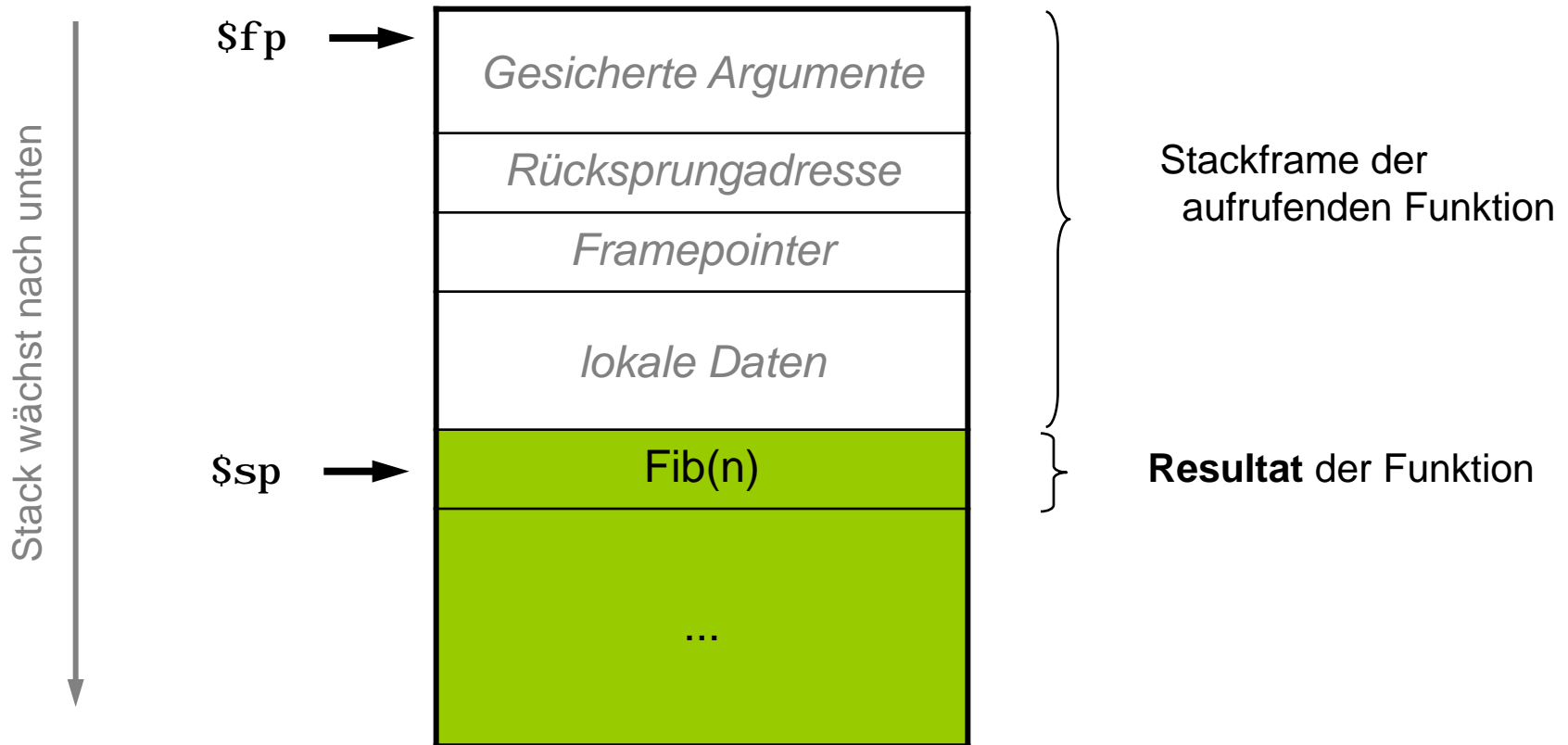
- am Anfang der Funktion sichern:
  - \$s0- \$s7, \$gp, \$sp, \$fp, \$ra  
(wenn in Funktion geändert)
- vor Funktionsaufruf sichern:
  - \$t0- \$t19
  - \$a0- \$a3, \$v0, \$v1  
(wenn nicht als Parameter genutzt)
- Werte auf dem Stack wiederverwenden

(siehe auch Vorlesung Folie 2-15)

# Aufgabe 1.2: Stack am Funktionsanfang



# Aufgabe 1.2: Stack am Funktionsende



# Lösung 1.2: Pseudocode

```

int Fib(int n)
{

    int result;
    if (n <= 2) {
        result = 1;
    }
    else {

        int n1 = Fib(n-1);

        int n2 = Fib(n-2);
        result = n1 + n2;
    }
    return result;
}

```

```

Fib:
# Vorbereitungsphase:

...
sw    $ra, 0($sp)
...
bgt   ..., ..., rekursion
...
j     abbauphase
rekursion:
...
jal   Fib
...
jal   Fib
...
abbauphase:
...
jr    $ra

```

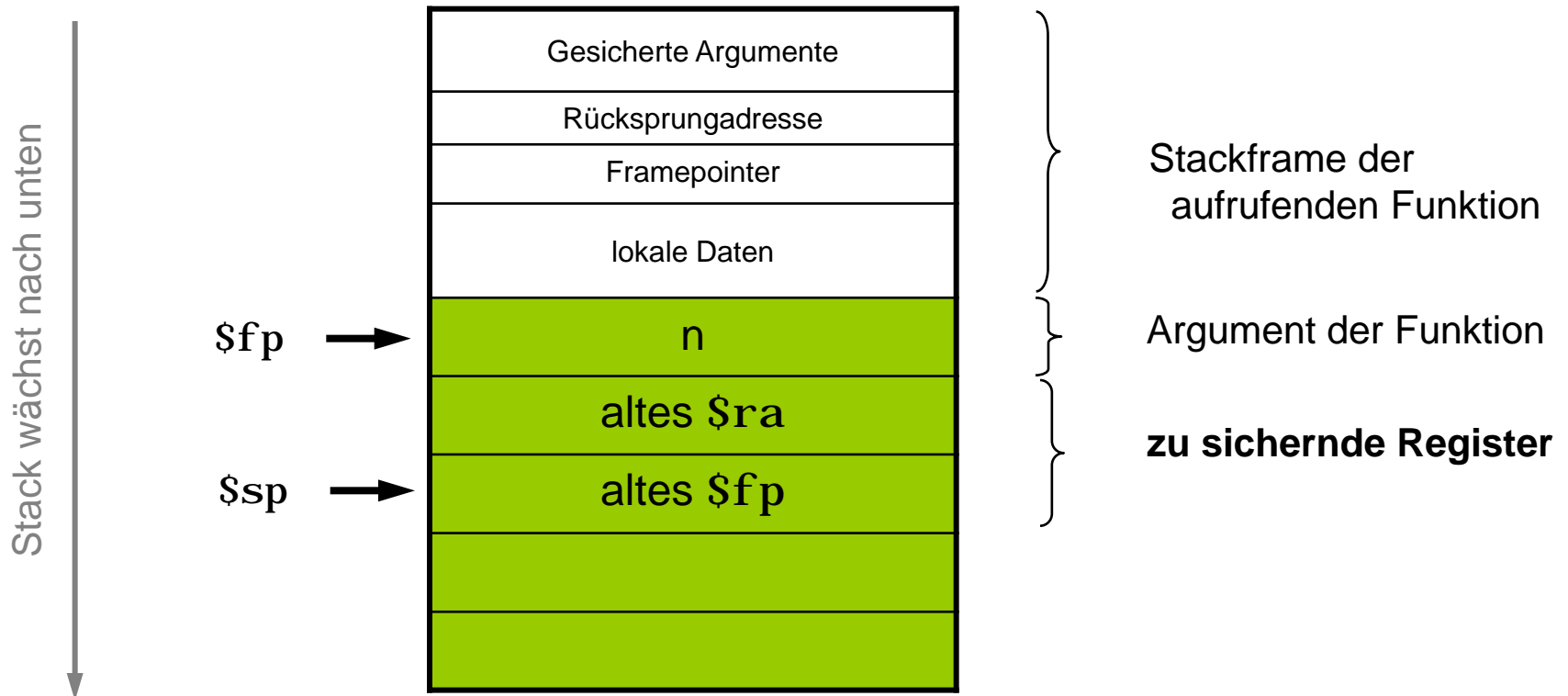
# Lösung 1.2: Registerzuordnung

```
int Fib(int n)
{
    int result;
    if (n <= 2) {
        result = 1;
    }
    else {
        int n1 = Fib(n-1);
        int n2 = Fib(n-2);
        result = n1 + n2;
    }
    return result;
}
```

Variable	zu sichern?	im Stack?
n	ja	ja
result	nein	nein
n1	ja	ja
n2	nein	ja

- Non-preserved Register für Zwischenergebnisse verwenden

# Lösung 1.2: Stack nach Vorbereitungsphase



# Lösung 1.2: Vorbereitungsphase

Fib:

```
addi    $sp, $sp, - 8           # Stack Pointer herabsetzen

sw      $ra, 4($sp)            # Rücksprungadresse retten
sw      $fp, 0($sp)            # Alten Frame Pointer auf Stack retten

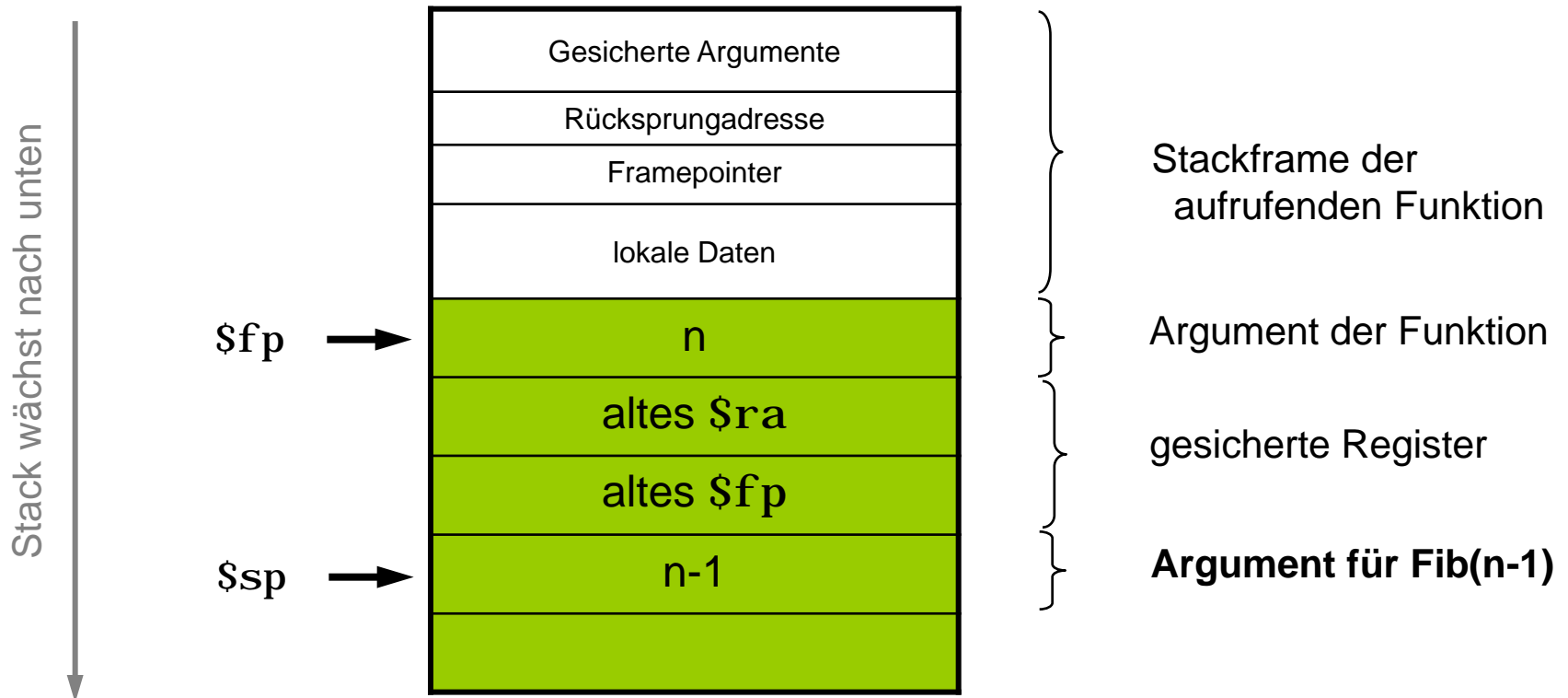
addi    $fp, $sp, 8            # Neuen Frame Pointer setzen

lw      $t0, 0($fp)            # Das Argument von Fib in t0 laden
li      $t1, 2
bgt     $t0, $t1, rekursion    # if n > 2 then goto $rekursion
li      $t0, 1                  # else fib(2) = fib(1) = 1 (result:= $t0)
j       abbauphase              # Zur Abbauphase springen
```

rekursion:

...

# Lösung 1.2: Stack vor Aufruf von Fib(n-1)



# Lösung 1.2: Aufruf von Fib(n-1)

...

lw \$t0, 0(\$fp) # Das Argument von Fib in t0 laden

...

**rekursion:**

addi \$t0, \$t0, -1 # \$t0 = n-1

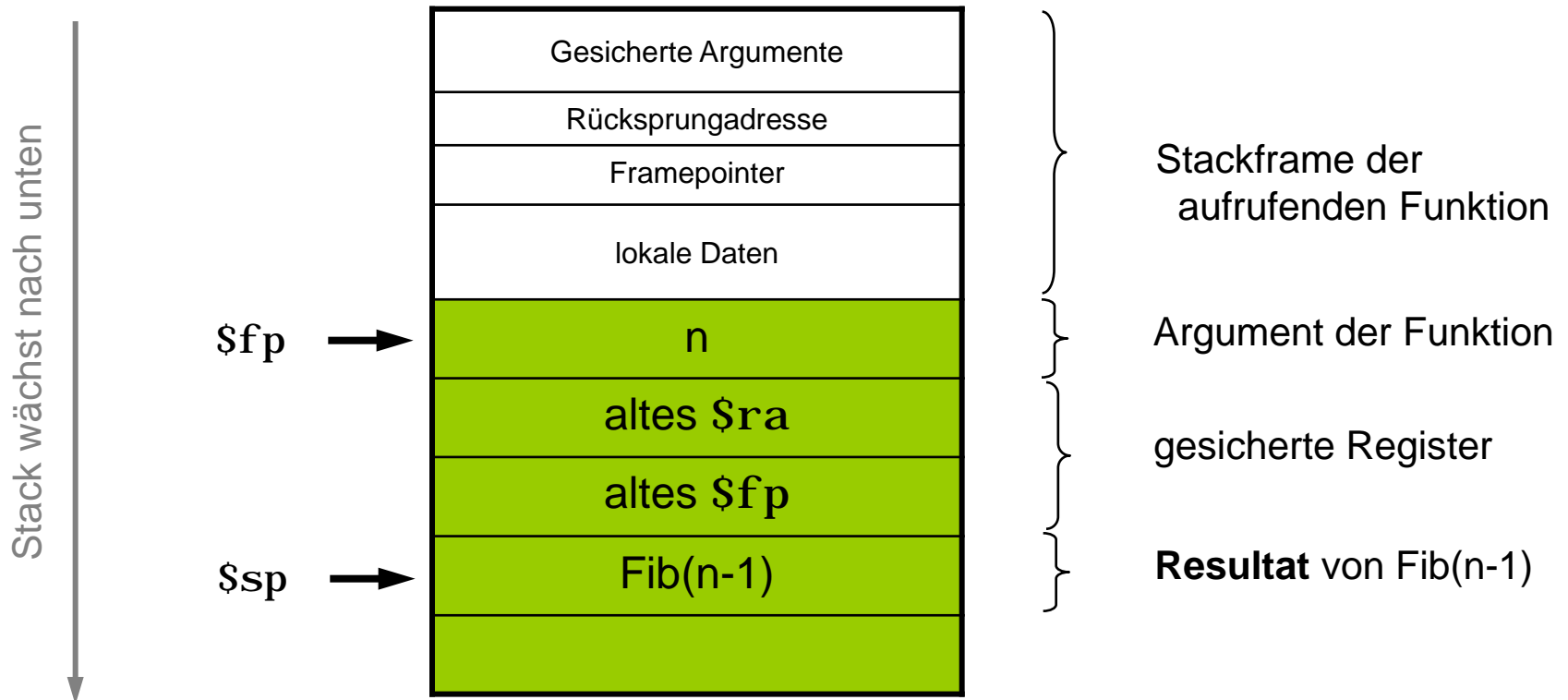
addi \$sp, \$sp, -4 # Stack Pointer herabsetzen

sw \$t0, 0(\$sp) # Argument n-1 auf dem Stack übergeben

jal Fib # Fib(n-1)

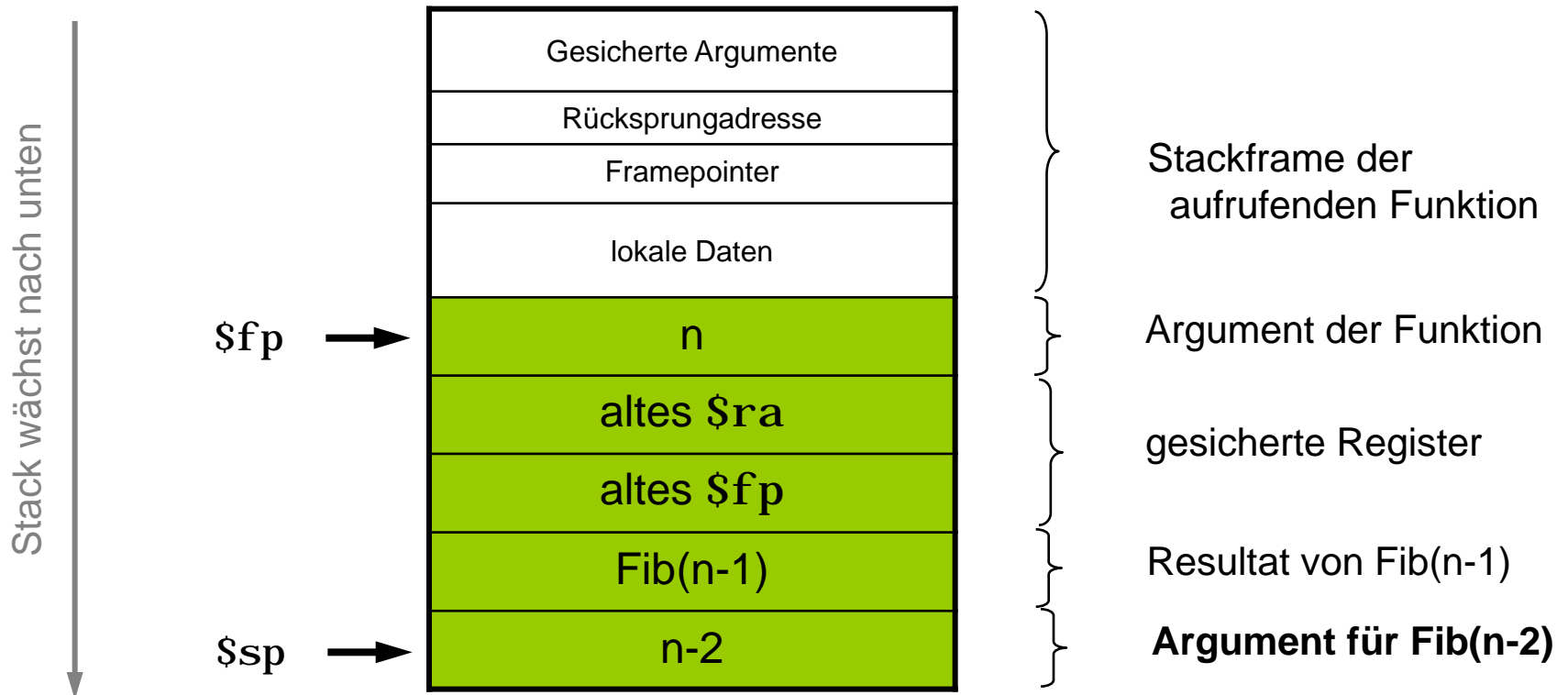
...

# Lösung 1.2: Stack nach Aufruf von Fib(n-1)



Resultat vorläufig auf Stack belassen

# Lösung 1.2: Stack vor Aufruf von Fib(n-2)



## Lösung 1.2: Aufruf von Fib(n-2)

...

**rekursion:**

```
addi    $t0, $t0, - 1           # $t0 = n-1
addi    $sp, $sp, - 4           # Stack Pointer herabsetzen
sw      $t0, 0($sp)             # Argument n-1 auf dem Stack übergeben
jal     Fib                     # Fib(n-1)

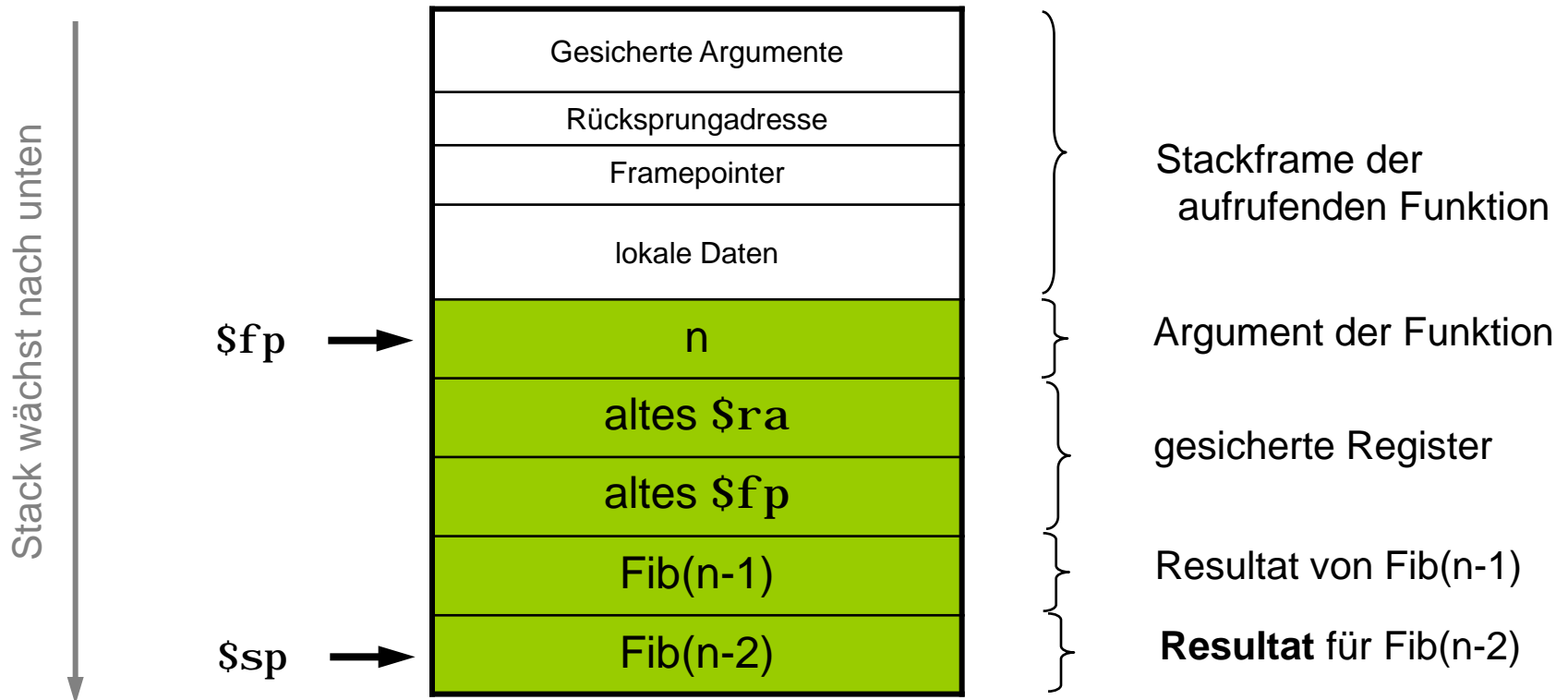
lw      $t0, 0($fp)             # Argument von Fib wieder holen (t0 = n)
addi    $t0, $t0, - 2           # t0 = n-2

addi    $sp, $sp, - 4           # Stack Pointer herabsetzen
sw      $t0, 0($sp)             # push argument n-2 on stack

jal     Fib                     # Fib(n-2)
```

...

# Lösung 1.2: Stack nach Aufruf von Fib(n-2)



# Lösung 1.2: Rekursion

## rekursion:

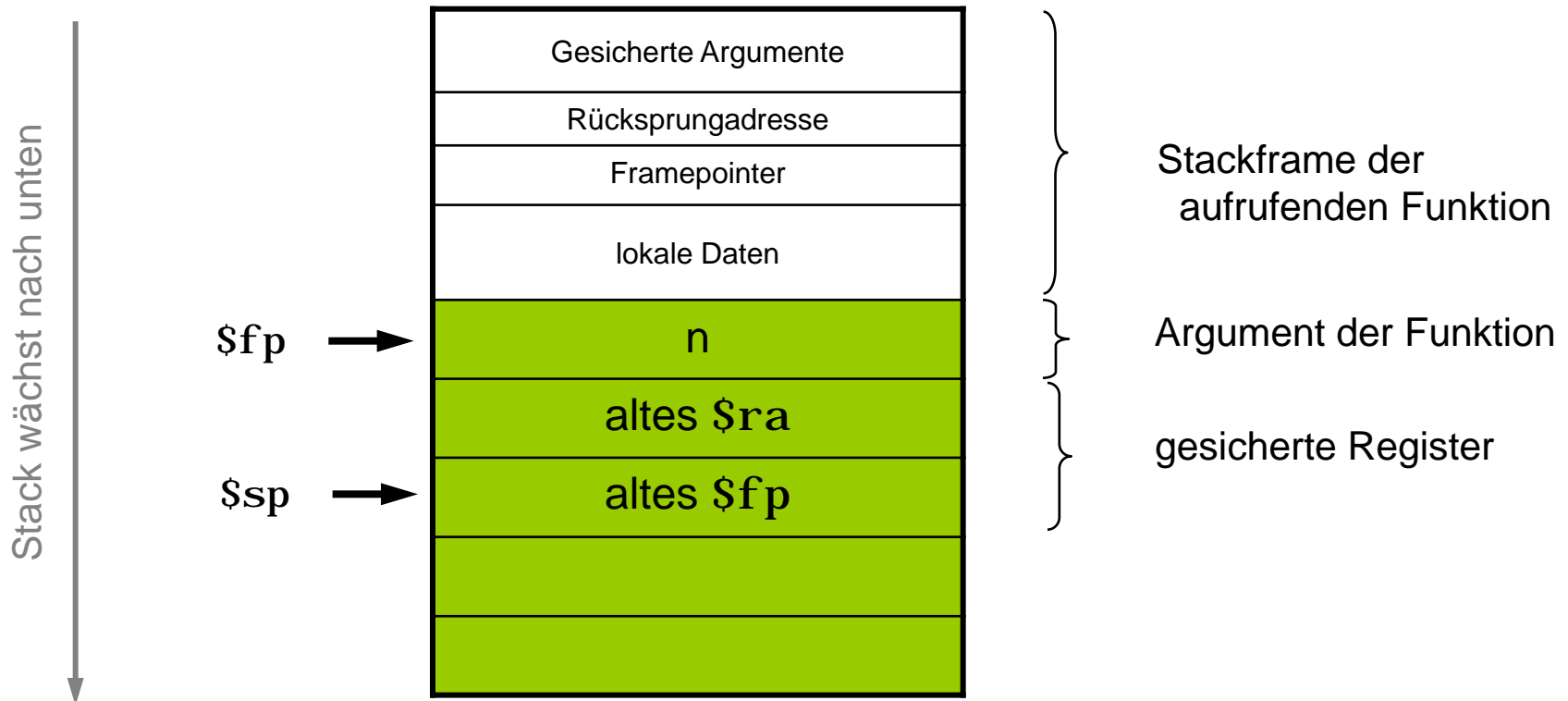
```
addi    $t0, $t0, -1           # $t0 = n-1
addi    $sp, $sp, -4           # Stack Pointer herabsetzen
sw      $t0, 0($sp)           # Argument n-1 auf dem Stack übergeben
jal     Fib                   # Fib(n-1)

lw      $t0, 0($fp)           # Argument von Fib wieder holen (t0 = n)
addi    $t0, $t0, -2           # t0 = n-2
addi    $sp, $sp, -4           # Stack Pointer herabsetzen
sw      $t0, 0($sp)           # push argument n-2 on stack
jal     Fib                   # Fib(n-2)

lw      $t0, 0($sp)           # Resultat von Fib(n-2) vom Stack in t0 laden
lw      $t1, 4($sp)           # Resultat von Fib(n-1) vom Stack in t1 laden
addi    $sp, $sp, 8           # Stack Pointer heraufsetzen

add     $t0, $t0, $t1         # t0 = Fib(n-2) + Fib(n-1)
```

# Lösung 1.2: Stack vor Abbauphase



# Lösung 1.2: Abbauphase

**abbauphase:**

			# Im Register t0 liegt das Resultat
lw	\$fp,	0(\$sp)	# Alter Frame Pointer holen und in fp laden
lw	\$ra,	4(\$sp)	# Return Adresse vom Stack holen
addi	\$sp,	\$sp, 8	# Stack Pointer heraufsetzen
sw	\$t0,	0(\$sp)	# Resultat Fib(n) auf dem Stack übergeben
jr	\$ra		# Rücksprung