



C++ Hiwi gesucht



Aufgabe

Erstellung eines Prototypen zur klinischen Evaluation von Algorithmen zur Perfusionsuntersuchung des Herzmuskels



Gesucht

wird ein motivierter Student oder eine motivierte Studentin mit fundierten C++ Kenntnissen und Interesse an einer Arbeit im medizinischen Umfeld. Medizinische Vorkenntnisse sind nicht erforderlich.



Geboten

wird eine interessante und abwechslungsreiche Arbeit im Grenzbereich von Informatik und Medizin in enger Zusammenarbeit mit einem industriellen Partner.

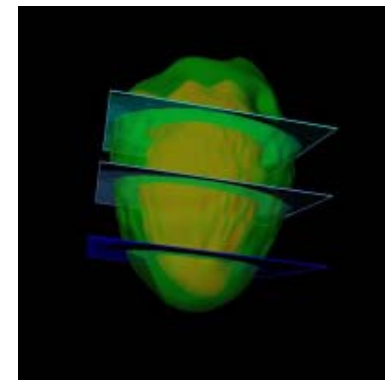
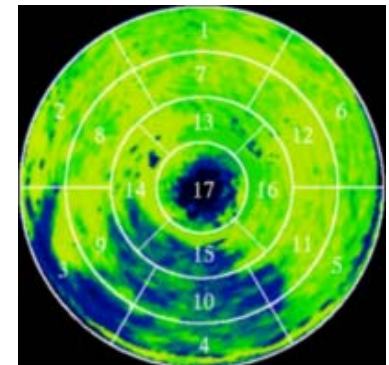
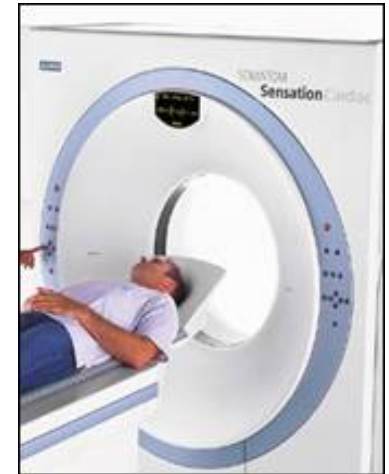


Weitere Details

Bei Dominik Fritz:

Tel.: 0721 608 4261, Email: dfritz@ira.uka.de

oder am besten persönlich im Raum 311.2,
Technologiefabrik, rechter Flügel, 2. Stock



Übung 4

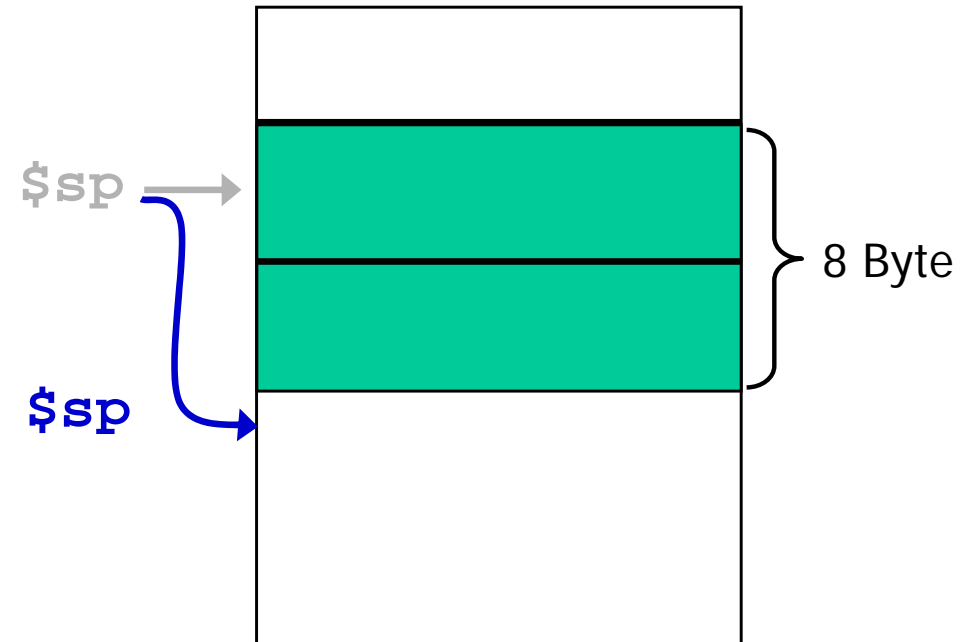
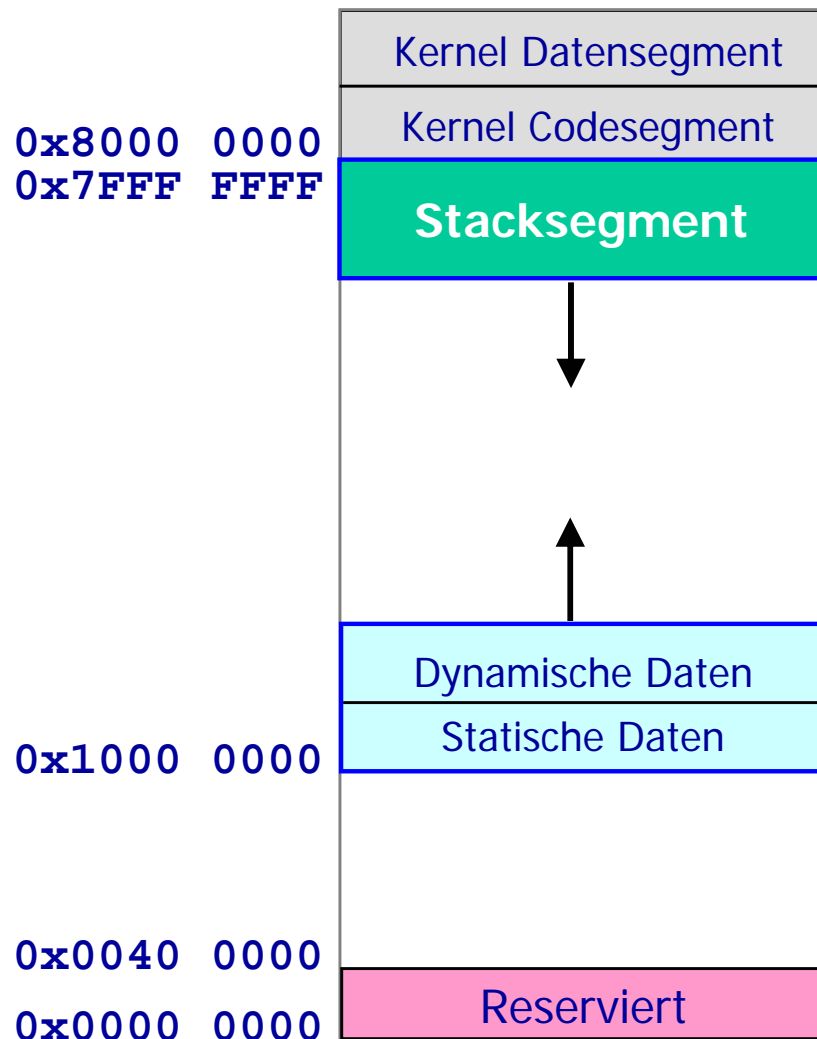
□ MIPS-Assembler

- Stackprogrammierung
- Unterprogramm-Aufruf

□ Pipelining

- DLX-Pipeline
- Abhängigkeiten
- Konflikte und deren Lösungen

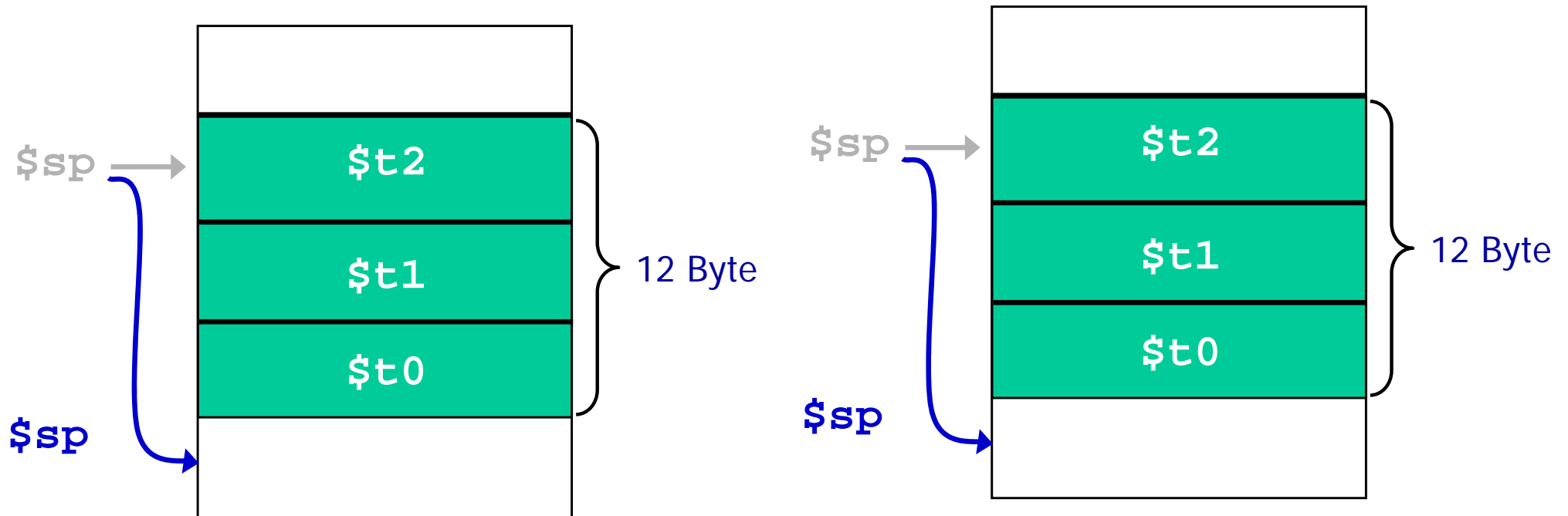
Stackprogrammierung



```
subu    $sp, $sp, 8
sw      $t0, 4($sp)
sw      $t1, 8($sp)
```

Stackprogrammierung

□ Lesen vom Stack:



```
subu    $sp, $sp, 12
sw      $t0, 4($sp)
sw      $t1, 8($sp)
sw      $t2, 12($sp)
```

```
lw      $t0, 4($sp)
lw      $t1, 8($sp)
lw      $t0, 12($sp)
addi    $sp, $sp, 12
```

Unterprogrammaufrufe

- ❑ Regeln oder Konventionen zur Verwendung von **Registern 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) x 4**

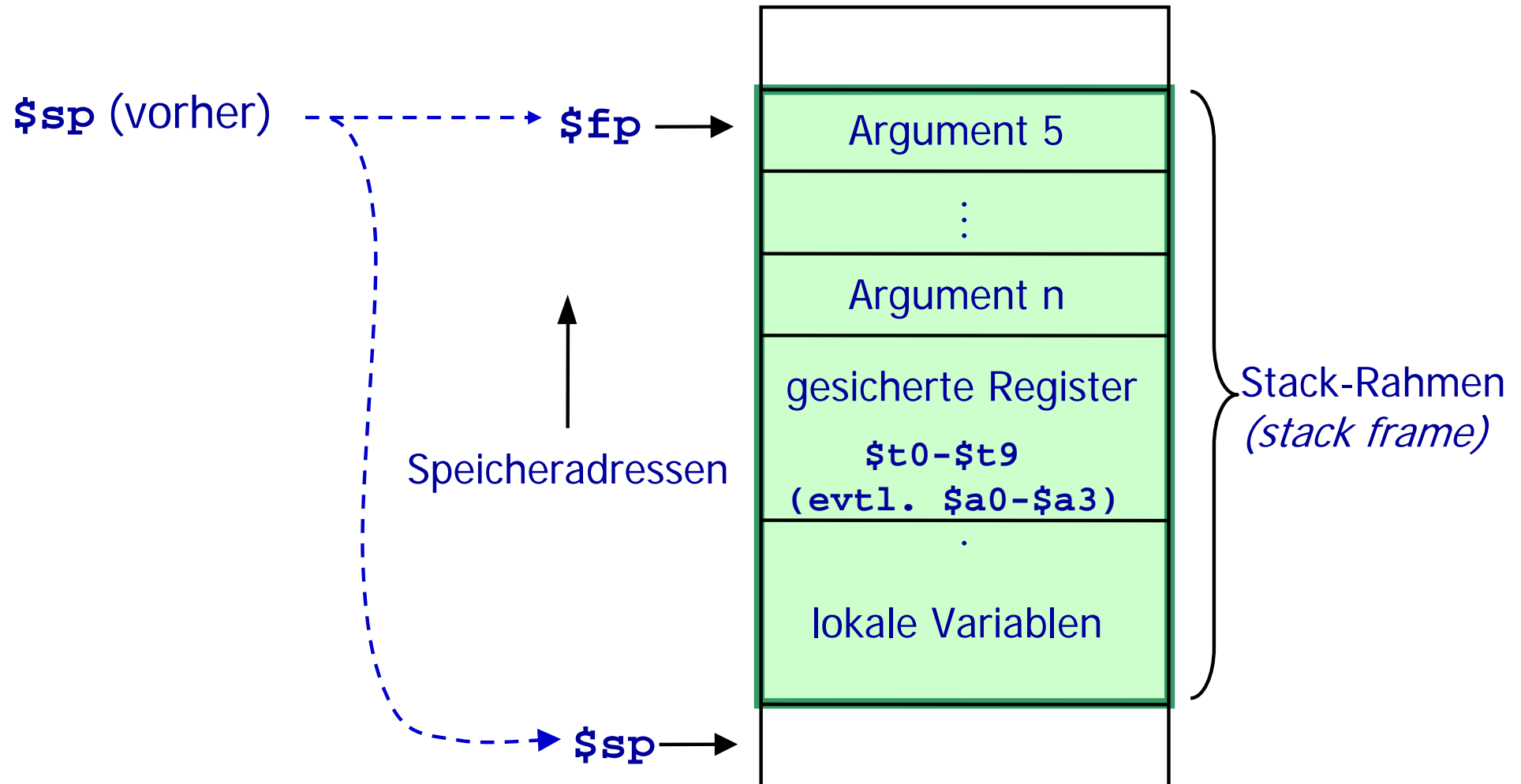
(Zu sichernde Fließkommazahlen mit doppelter Genauigkeit benötigen
acht Bytes Speicherplatz)

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

Stack-Rahmen (*stack frame*)



Einfacher Unterprogrammaufruf

```
main:          jal subroutine
```

```
subroutine:    # keine weiteren  
              # Unterprogrammaufrufe  
  
              # für lokale Variablen  
              # nur $t0 bis $t9  
  
              jr $ra
```

Beispiel für einen Unterprogrammaufruf

```
# Hauptprogramm
        .globl main
main:    subu $sp, $sp, 8
        sw $ra, 4($sp)
        sw $fp, 8($sp)
        addu $fp, $sp, 8

        jal subroutine

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

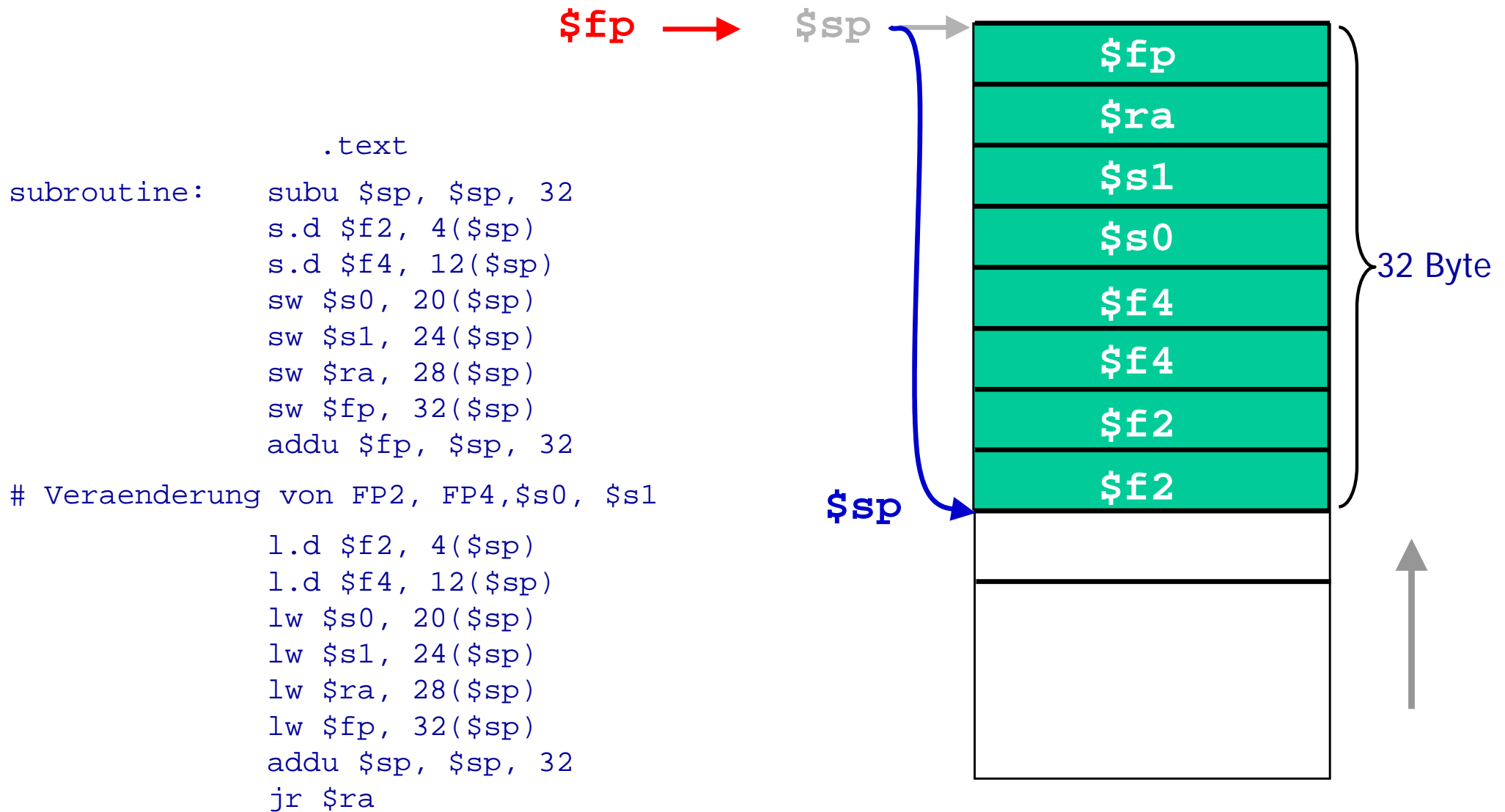
**Unterprogrammaufrufe
mit jal-Befehl !**

```
        .text
subroutine: subu $sp, $sp, 32
          s.d $f2, 4($sp)
          s.d $f4, 12($sp)
          sw $s0, 20($sp)
          sw $s1, 24($sp)
          sw $ra, 28($sp)
          sw $fp, 32($sp)
          addu $fp, $sp, 32

# Veraenderung von $fp2,$fp4,$s0,$s1

          l.d $f2, 4($sp)
          l.d $f4, 12($sp)
          lw $s0, 20($sp)
          lw $s1, 24($sp)
          lw $ra, 28($sp)
          lw $fp, 32($sp)
          addu $sp, $sp, 32
          jr $ra
```

Beispiel für einen Unterprogrammaufruf



Unterprogrammaufrufe

Beispiel (c-Code)

```
int min(int a, int b)
{
    if (a < b) return a;
    else return b;
}

main()
{
    int x = 123; int y = 456;
    int z = min (x,y);
}
```

Unterprogrammaufrufe

MIPS-Assembler:

```
j min          # Sprung zu min
...           # Befehl nach Berechnung von min

min: ...       # code fuer min(a,b)
...
j ??          # zurueck zum Hauptprogramm
```

Probleme:

- Rücksprungadresse sichern und wiederherstellen
- Parameterübergabe

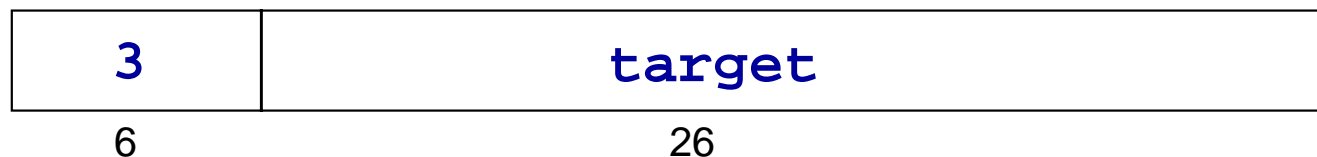


Unterprogrammaufrufe

- **Rücksprungadresse:** MIPS unterstützt den Funktionsaufruf durch speziellen Befehl

jal ("jump and link")

jal target



Springe zu target. Speichere die Adresse des nächsten Befehls in \$ra

Unterprogrammaufrufe

□ Parameterübergabe:

Konventionen zur Verwendung der Register, um Fehler zu vermeiden

- \$a0, ..., \$a3 Register für Argumente des Unterprogramms
- \$v0, \$v1 Register für Funktionswert bzw. für Ausgabeparameter des Unterprogramms
- \$t0-\$t9: Register können im Unterprogramm überschrieben werden
- \$s0-\$s7: Register dürfen nicht überschrieben werden



Unterprogrammaufrufe

```
lw $s0, x           # initialisiere $s0 mit x=123
lw $s1, y           # initialisiere $s1 mit y=456
add $a0, $s0, $zero # kopiere $s0 in Register $a0
                    # fuer Parameteruebergabe
add $a1, $s1, $zero # entsprechend $s1 in $a1
jal min             # springe zur Funktion min
add $s2, $v0, $zero # kopiere Funktionswert in
                    # Ergebnisregister
sw $s2, z           # speichere Ergebnis ab
...

```

Unterprogrammaufrufe

```
min:    add $t0, $zero, $a0          # initialisiere $t0 mit
                                           # Parameter a

        add $t1, $zero, $a1          # initialisiere $t1 mit
                                           # Parameter b

        slt $t2, $t1, $t0            # ($t1 < $t0)?
        beq $t2, $zero, else          # wenn die Bedingung nicht
                                           # erfuehlt ist, gehe zu else

        add $v0, $t1, $zero           # es gilt: ($t1 < $t0)
        j  ende                       #

else:    add $v0, $t0, $zero           # es gilt ($t1 >= $t0)
                                           # move $t0 to $v0 (result)

ende:    j  $ra                       # Ruecksprung
```

Rekursive Unterprogrammaufrufe

□ Problem:

- Rücksprungadresse im Register **\$ra** wird bei wiederholtem Unterprogrammaufruf immer wieder überschrieben
- Entsprechendes gilt für lokale Daten in Registern

□ Lösung:

- Sichern von Rücksprungadressen und lokalen Daten auf einem Stack

Rekursive Unterprogrammaufrufe

c-Code:

```
main ()
{
    printf („Die Fakultät von 10 ist: %d\\n“, fakultaet(10));
}

int fakultaet (int n)
{
    if (n<1)
        return (1);
    else
        return (n*fakultaet(n-1));
}
```

Rekursive Unterprogrammaufrufe

MIPS-Code:

```
fakultaet: subu $sp, $sp, 8          # Stack-Frame von 8 Bytes
           sw $ra, 4($sp)           # Ruecksprungadresse
           sw $a0, 0($sp)           # Argument(n) sichern

           slt $t0, $a0, 1          # (n < 1)? lade das Argument
           beqz $t0, $zero, L1      # falls n >=1 gilt, gehe zu
L1

           add $v0, $zero, 1        # return 1

           lw $ra, 4($sp)           # Ruecksprungadresse
wiederherstellen
           addu $sp, $sp, 8         # Frame-Stack loeschen
           jr $ra                  # gehe zur
Ruecksprungadresse
```

Rekursiver Unterprogrammaufruf

```
L1:      sub $a0, $a0, 1      # verwende n-1 als Argument
        jal fakultaet       # rufe mit n-1 auf

        lw $a0, 0($sp)      # hole Argument n wieder vom Stack
        lw $ra, 4($sp)      # hole Ruecksprungadresse vom Stack
        add $sp, $sp, 8     # Stack-Frame loeschen

        mul $v0, $a0, $v0   # return n*fak(n-1)

        jr $ra              # Ruecksprung
```

Beispiel 2

Stack

\$ra (\$fp)	main
\$ra \$a0	fakultaet(10)
\$ra \$a0	fakultaet(9)
\$ra \$a0	fakultaet(8)
\$ra \$a0	fakultaet(7)



Unterprogrammaufruf

Aufrufendes Programm



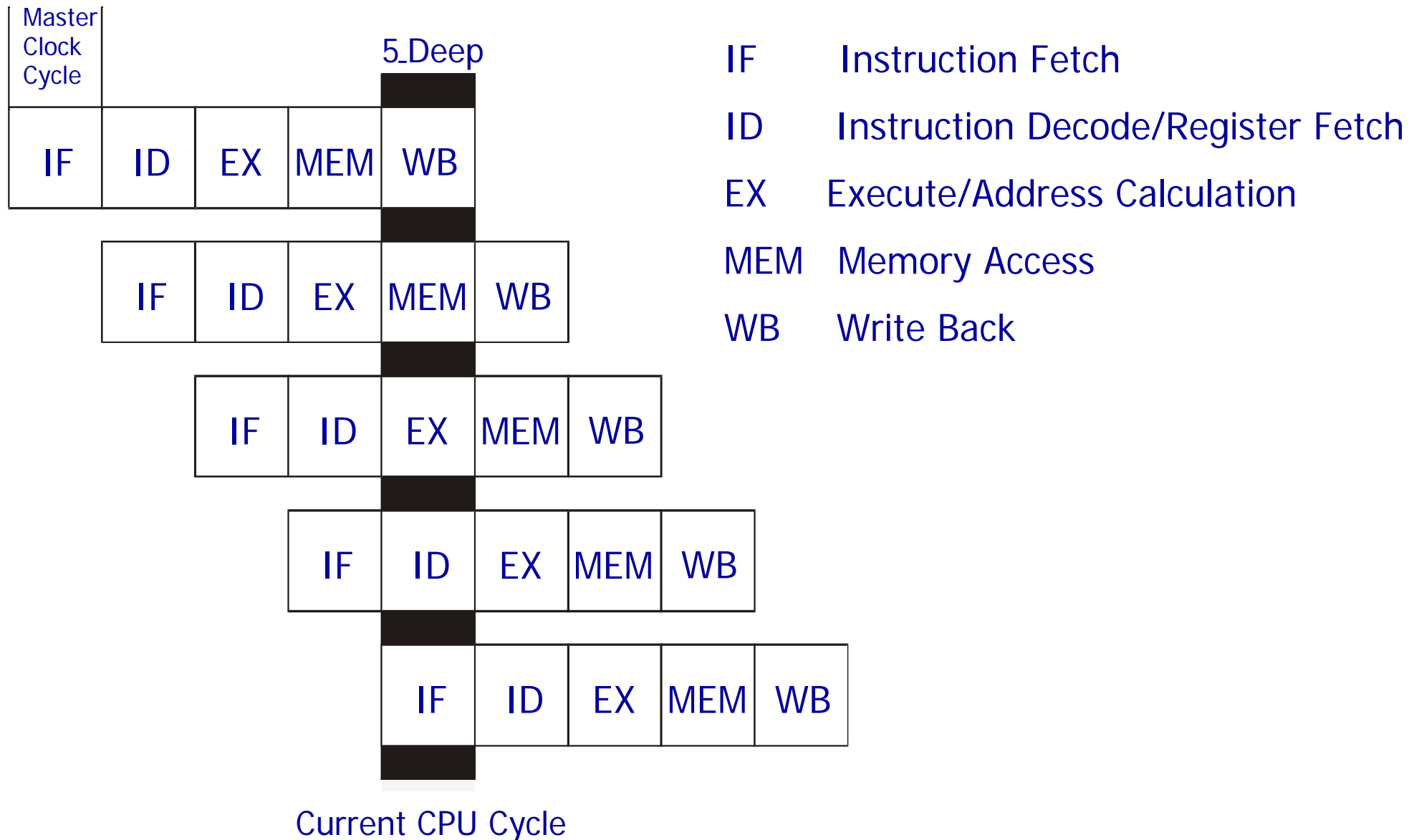
Unterprogramm



Pipelining

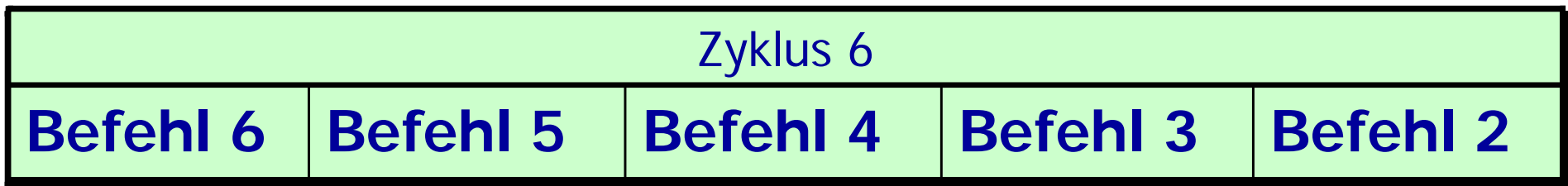
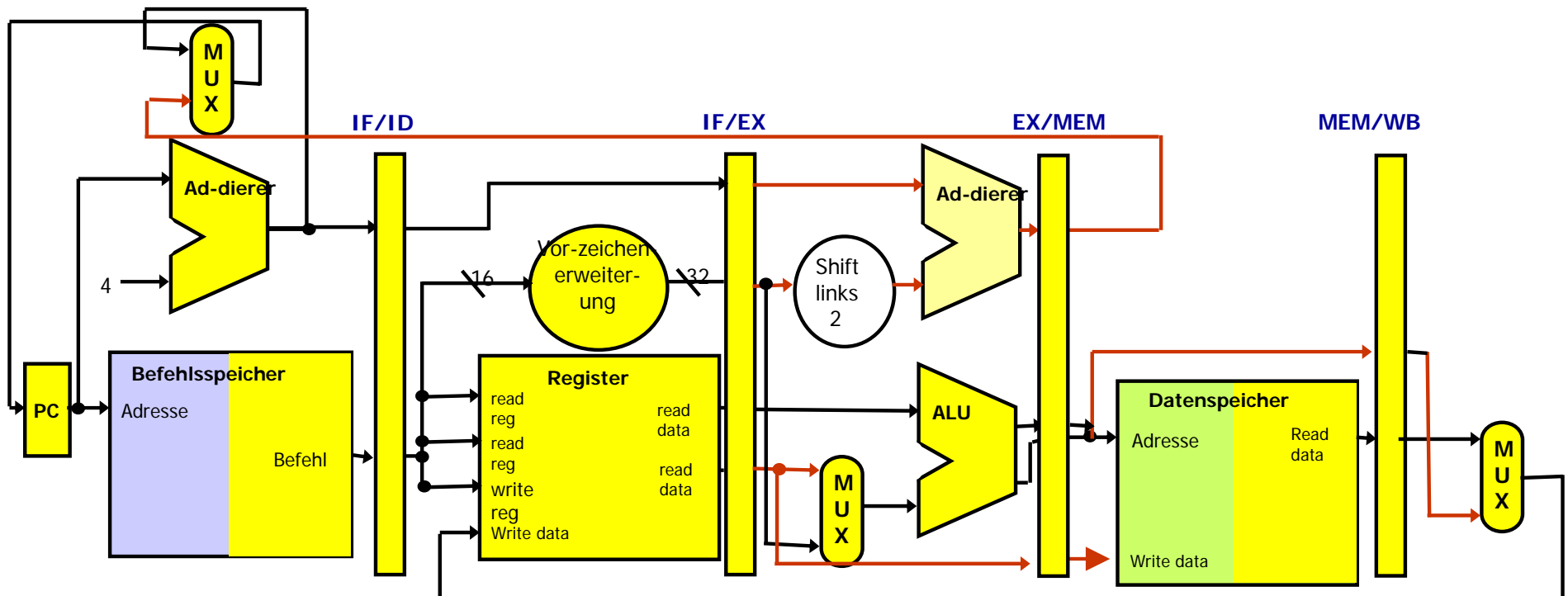
- **DLX-Pipeline**
- **Abhängigkeiten**
- **Konflikte und deren Lösungen**

DLX-Pipeline



Pipeline-Konflikte

- Bei einer gefüllten Pipeline wird pro Taktzyklus ein Befehl beendet.



Drei Arten von Pipeline-Konflikten

- ❑ **Datenkonflikte:** Treten auf, wenn ein Operand ist in der Pipeline (noch) nicht verfügbar.
 - Datenkonflikte werden durch Datenabhängigkeiten im Befehlsstrom erzeugt
- ❑ **Struktur- oder Ressourcenkonflikte:** Treten auf, wenn zwei Pipeline-Stufen dieselbe Ressource benötigen, auf diese aber nur einmal zugegriffen werden kann.
- ❑ **Steuerflusskonflikte** treten bei Programmsteuerbefehlen auf:
 - wenn in der IF-Stufe die Zieladresse des als nächstes auszuführenden Befehls noch nicht berechnet ist bzw.
 - wenn im Falle eines bedingten Sprunges noch nicht klar ist, ob überhaupt gesprungen wird.

Pipelinekonflikte

- ❑ Ressourcenkonflikte: DLX-Pipeline ist entsprechend angepasst, so dass keine Ressourcenkonflikte auftreten
- ❑ Datenkonflikte:
 - RAW: Befehl $i+1$ liest einen Wert bevor Befehl i geschrieben hat
 - WAW: Kann nicht auftreten, da die DLX-Pipeline nur in WB schreibt
 - WAR: Alle Lesezugriffe erfolgen in der ID/RF und alle Schreibzugriffe in WB
- ❑ Steuerkonflikte:
 - Pipeline muss wegen branch geleert und neu gefüllt werden

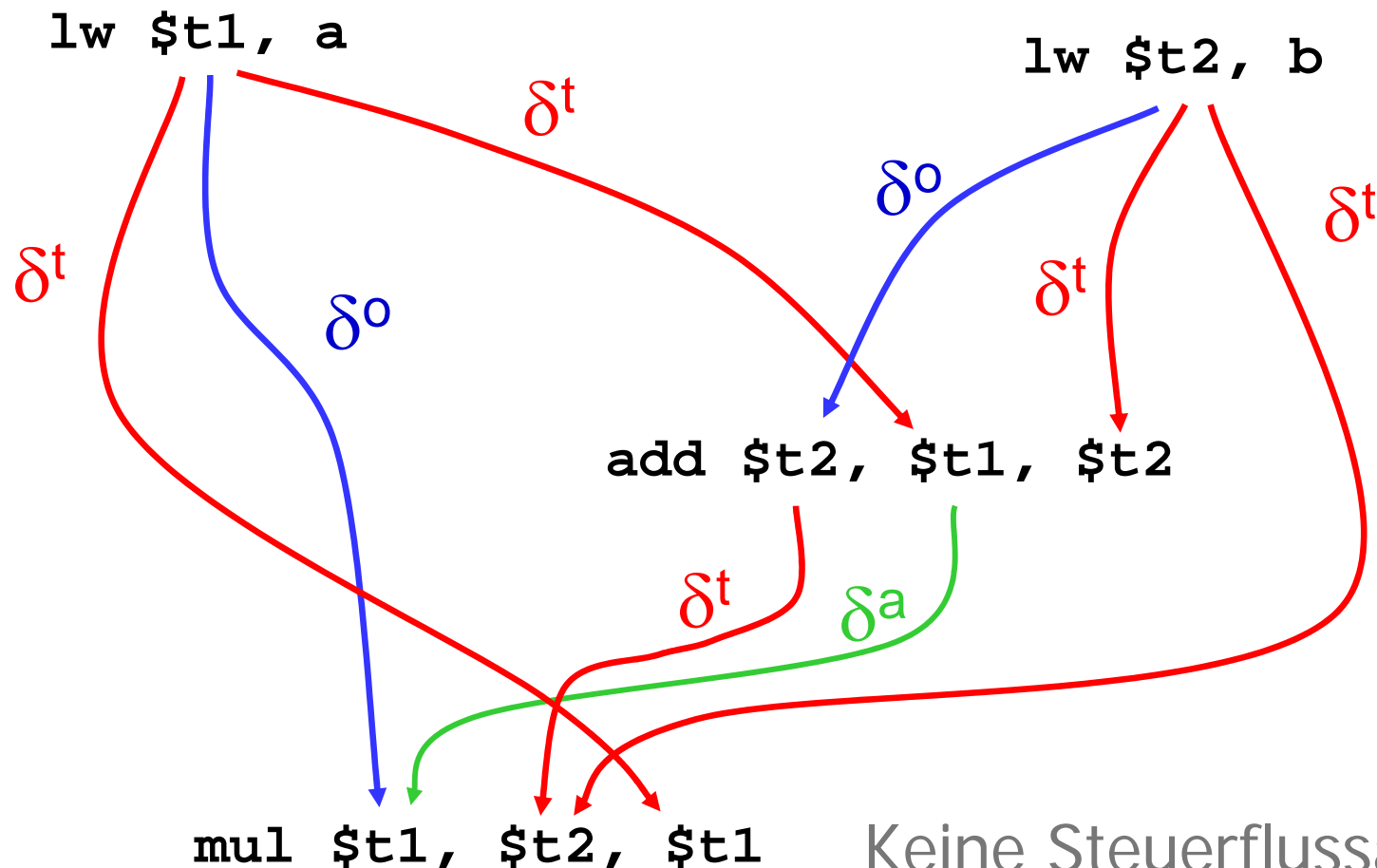
Aufgabe 1

Betrachten Sie das folgende sequentielle Programmstück, in dem die Konstanten **a** und **b** Speicheradressen darstellen:

```
s1:  lw    $t1, a           ; $t1 := [a]
s2:  lw    $t2, b           ; $t2 := [b]
s3:  add   $t2, $t1, $t2
s4:  mul   $t1, $t2, $t1
```

Aufgabe 1.1

- Bestimmen Sie alle Daten- und Steuerflussabhängigkeiten in diesem Programmstück



Aufgabe 1.2

2. Wieviele Pipelinekonflikte treten auf?

lw \$t1, a



lw \$t2, b



add \$t2, \$t1, \$t2



mul \$t1, \$t2, \$t1

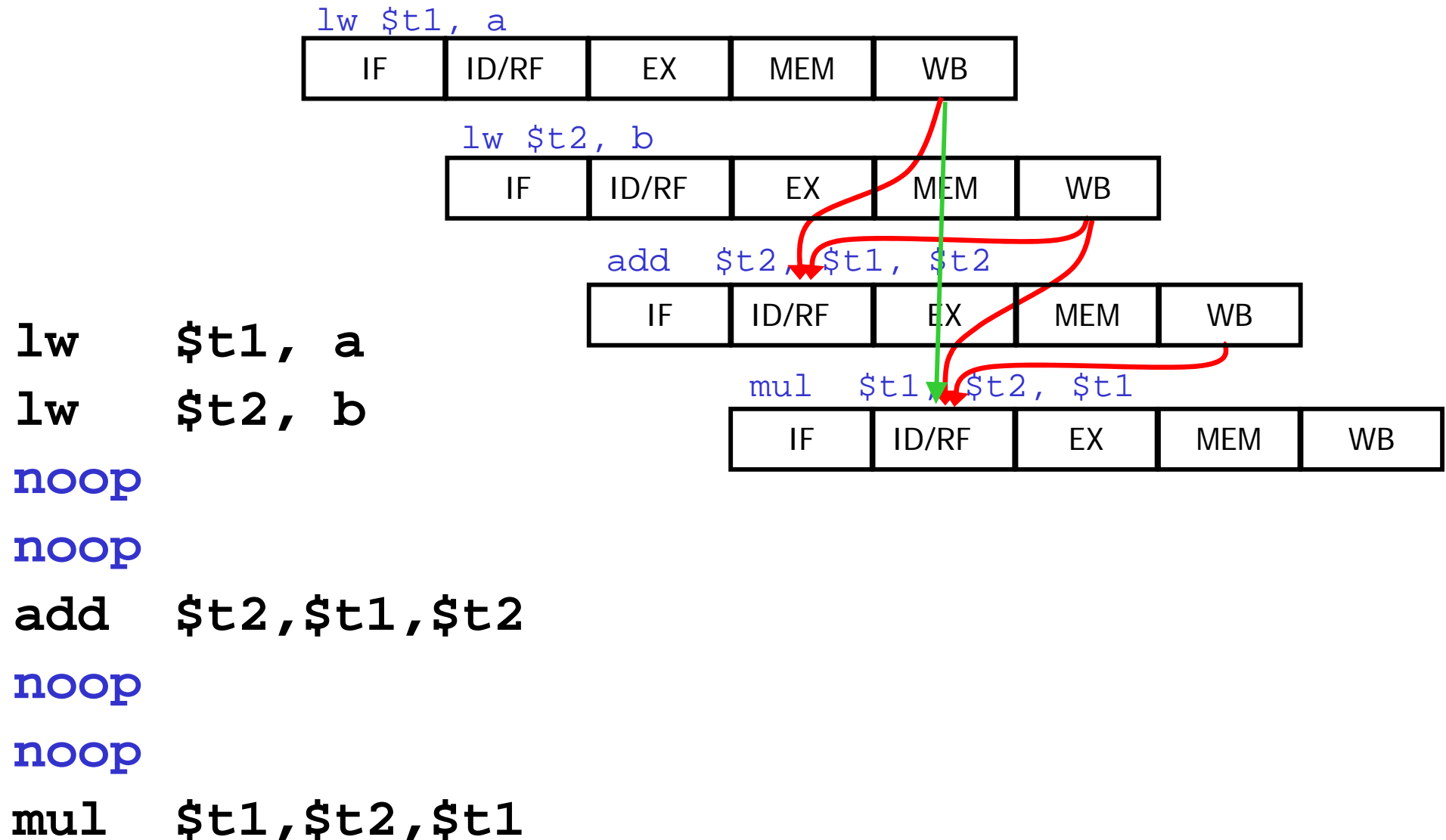


Aufgabe 1

- 3.** Die auftretenden Pipelinekonflikte werden von der Hardware nicht erkannt und müssen vom Compiler durch Einfügen von NOOP-Befehlen behandelt werden.

Ergänzen Sie das Programmstück so, dass auch die Pipelinekonflikte berücksichtigt werden

Aufgabe 1.3



Aufgabe 1

4. Welche der NOOP-Befehle sind noch notwendig, falls die auftretenden Pipelinekonflikte von der Hardware erkannt werden und durch *Load Forwarding* und *Result Forwarding* behandelt werden?



Aufgabe 1.4

