

# Rechnertechnik

Prof. Dr. rer. nat. Peter Gerwinski

14. Juni 2021

# Rechnertechnik

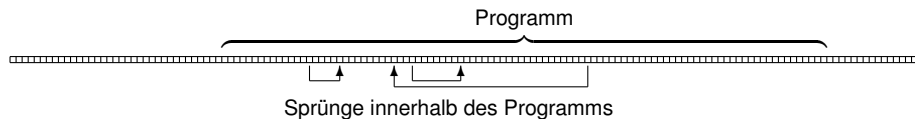
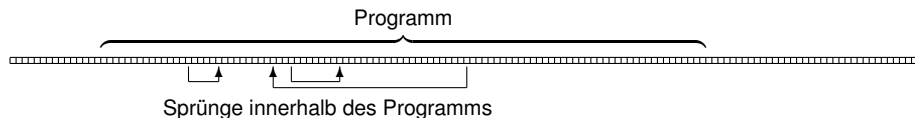
- 1 Einführung**
- 2 Vom Schaltkreis zum Computer**
- 3 Architekturmerkmale von Prozessoren**
- 4 Der CPU-Stack**
  - 4.1 Implementation
  - 4.2 Unterprogramme
  - 4.3 Register sichern
  - 4.4 Stack-Überläufe
  - 4.5 Puffer-Überläufe
- 5 Anwender-Software**
  - 5.1 Relokation und Linken
  - 5.2 Dateiformate
  - 5.3 Die Toolchain
  - 5.4 Besonderheiten von Mikrocontrollern
- 6 Pipelining**
  - 6.1 Konzept
  - 6.2 Arithmetik-Pipelines
  - 6.3 Instruktions-Pipelines

...

# 5 Anwender-Software

## 5.1 Relokation und Linken

Software im Speicher

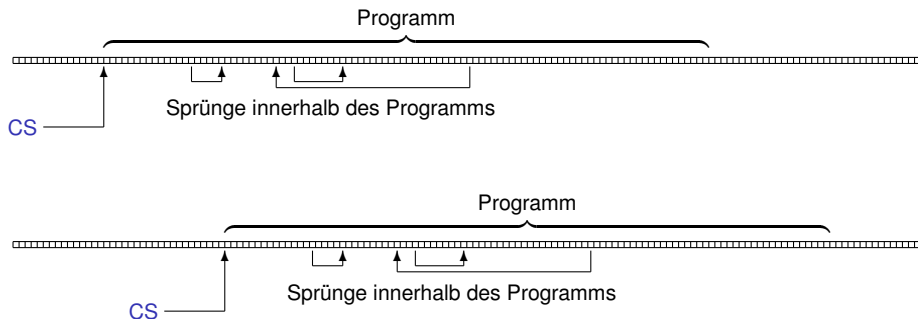


Sprünge anpassen: Relokation

# 5 Anwender-Software

## 5.1 Relokation und Linken

### Software im Speicher

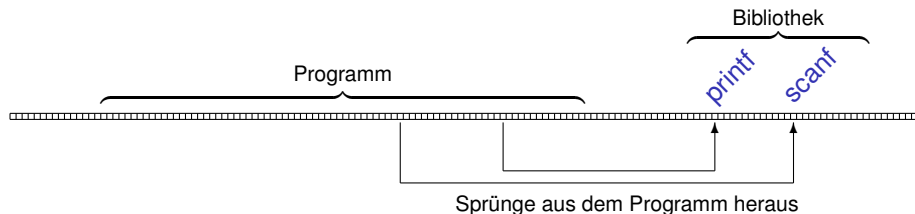
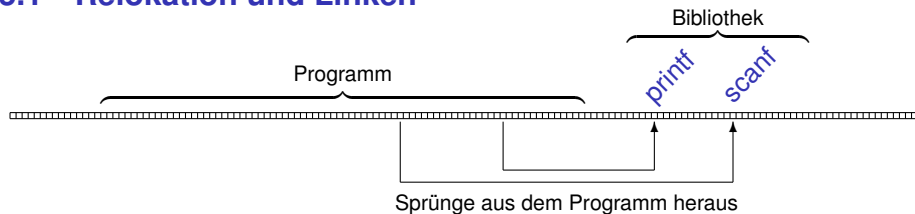


Sprünge anpassen: Relokation

Hardware-Unterstützung (z. B. Intel): Speichersegmentierung

CS = Code-Segment: Segment-Register oder Selektor

## 5.1 Relokation und Linken



Sprünge anpassen: Linken

Beim Erzeugen der Datei: statisches Linken

Beim Laden: dynamisches Linken

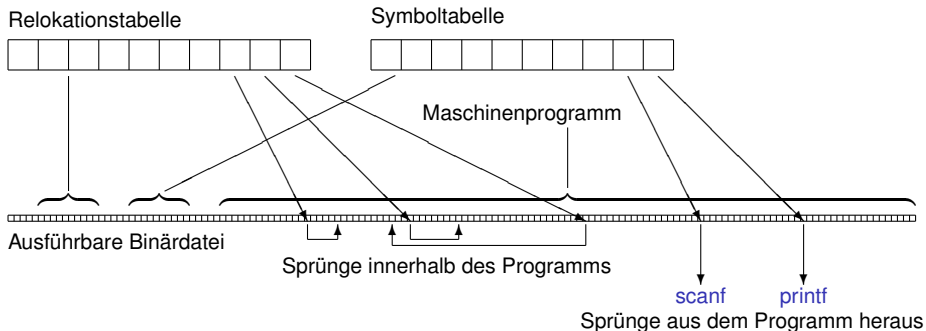
## 5.2 Dateiformate

Man kann Maschinenprogramme nicht „einfach so“ in den Speicher laden.

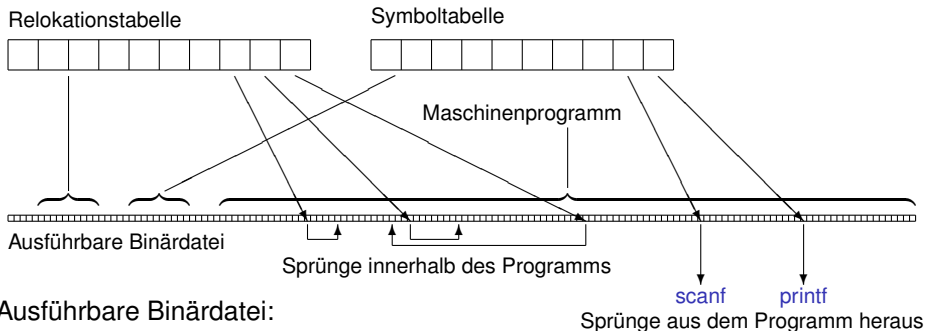
Sprünge anpassen

- Relokation: Relokationstabelle
- Linken: Symboltabelle

## 5.2 Dateiformate



## 5.2 Dateiformate



Ausführbare Binärdatei:

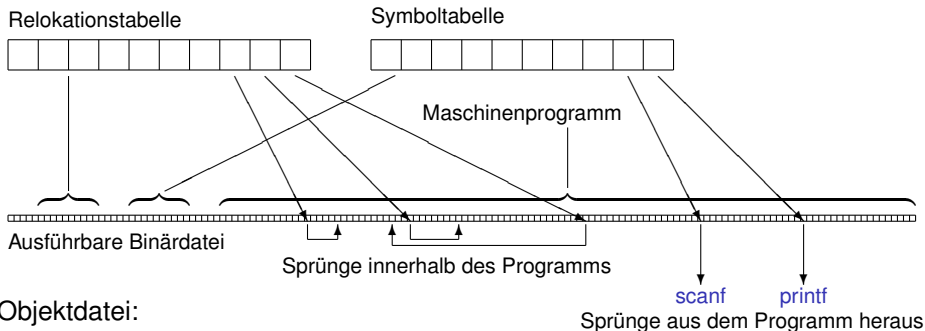
Relokationstabelle,  
Symboltabelle für dynamischen Linker

Formate: a.out, COFF, ELF, ...

Dateiendungen: (keine), .elf, .com, .exe, .scr



## 5.2 Dateiformate



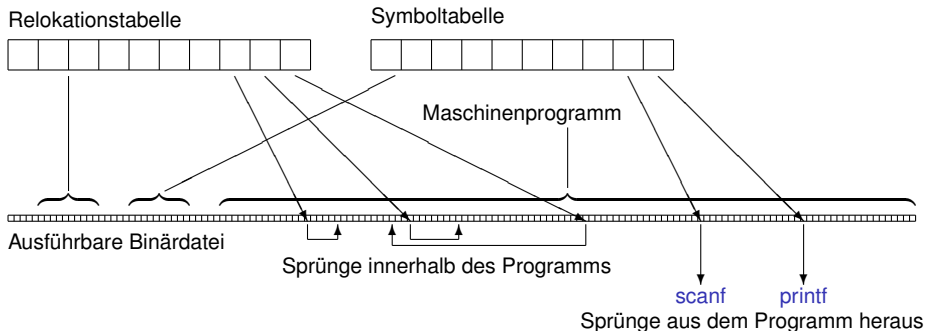
Objektdatei:

Relokationstabelle,  
Symboltabellen für statischen und dynamischen Linker

Formate: a.out, COFF, ELF, ...

Dateiendungen: .o, .obj

## 5.2 Dateiformate



Bibliothek:

Zusammenfassung mehrerer Objekt-Dateien

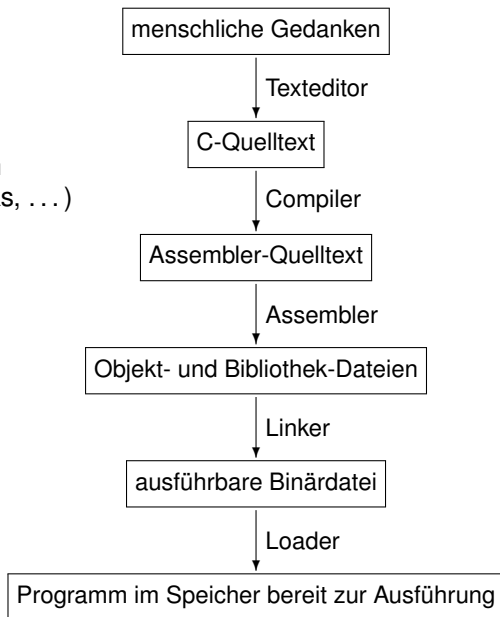
Statische Bibliotheken: .a, .lib

Dynamische Bibliotheken: .so, .dll

## 5.3 Die Toolchain

Automatischer Aufruf:

- Entwicklungsumgebungen  
(z. B. Eclipse, Code::Blocks, ...)
- `gcc` = Compiler  
+ Assembler  
+ Linker  
+ ...
- `make` kann *alles* aufrufen



## 5.4 Besonderheiten von Mikro-Controllern

Kein Betriebssystem

→ kein Relocator, kein dynamischer Linker

→ Wir müssen dem Mikro-Controller alles „mundgerecht“ servieren.

- fertiges ROM: Hersteller
- Flash-Speicher und In-System Programmer (ISP)
- Flash-Speicher und Boot-Loader

In jedem Fall: statisch linken, Relokation vorher

→ ELF-Datei in HEX-Datei umwandeln

Format: Intel-Hex-Format

Dateiendung: .hex

# 4 Der CPU-Stack

## 4.1 Implementation

Speicher, in dem Werte „gestapelt“ werden: *Stack*

- Speicherbereich (ein array) reservieren
- Variable (typischerweise: Prozessorregister) als *Stack Pointer* reservieren  
→ *SP*
- Assembler-Befehl *push foo*:  $*SP++ = foo;$
- Assembler-Befehl *pop bar*:  $bar = *--SP;$

Speziell: Unterprogramme

## 4 Der CPU-Stack

### 4.2 Unterprogramme

Parameter:

- Prozessorregister
- CPU-Stack

Rückgabewert:

- Prozessorregister

Aufruf:

- push IP  
  jmp foo ← mov #foo IP  
          → call foo

Rücksprung:

- pop IP  
  → ret

## 4 Der CPU-Stack

### 4.3 Register sichern

Ein Unterprogramm verändert Registerinhalte.

- im Hauptprogramm nötigenfalls vor Aufruf sichern
- im Unterprogramm vor Benutzung sichern
- Kombinationen (manche Register so, manche so)

## 4 Der CPU-Stack

### 4.4 Stack-Überläufe

Unendliche Rekursion

```
#include <stdio.h>
```

```
int fak (int n)
```

```
{
```

```
    if (n <= 1)
```

```
        return 1;
```

```
    else
```

```
        return n * fak (n); ← Fehler!
```

```
}
```

```
int main (void)
```

```
{
```

```
    printf ("%d! = %d\n", 6, fak (6));
```

```
    return 0;
```

```
}
```

Bei jedem Aufruf wird die Rücksprungadresse auf den Stack gelegt und die Variable `n` auf dem Stack gesichert.



## 4 Der CPU-Stack

### 4.5 Puffer-Überläufe

```
#include <stdio.h>
```

```
int main (void)
```

```
{
```

```
    int ID;
```

```
    char buffer[20];
```

```
    printf ("Your_ID,_please:_");
```

```
    gets (buffer);
```

```
    sscanf (buffer, "%d", &ID);
```

```
    printf ("Your_name,_please:_");
```

```
    gets (buffer);
```

```
    printf ("Hello,_%s!\nYour_ID_is_%d.\n", buffer, ID);
```

```
    return 0;
```

```
}
```

Die Funktion `gets()` prüft nicht, ob `buffer[]` für den eingegebenen Namen ausreicht, und überschreibt ggf. die Variable `ID` sowie die Rücksprungadresse des Funktionsaufrufs von `main()`.

**gets() nicht verwenden!**

## 7 Pipelining

# 7 Pipelining

## 7.1 Konzept

- Aufgabe in Teilaufgaben zerlegen
- Teilaufgaben parallel ausführen

# 7 Pipelining

## 7.1 Konzept

- Aufgabe in Teilaufgaben zerlegen
- Teilaufgaben parallel ausführen




Beispiel: Wäsche waschen

# 7 Pipelining

## 7.1 Konzept

- Aufgabe in Teilaufgaben zerlegen
- Teilaufgaben parallel ausführen

Beispiel: Wäsche waschen




- Teilaufgaben:  ,  , 

# 7 Pipelining

## 7.1 Konzept

- Aufgabe in Teilaufgaben zerlegen
- Teilaufgaben parallel ausführen

Beispiel: Wäsche waschen

- Teilaufgaben:  ,  , 
- müssen nacheinander ausgeführt werden

# 7 Pipelining

## 7.1 Konzept

- Aufgabe in Teilaufgaben zerlegen
- Teilaufgaben parallel ausführen

Beispiel: Wäsche waschen




- Teilaufgaben: , , 
- müssen nacheinander ausgeführt werden: Datenfluß

# 7 Pipelining

## 7.1 Konzept

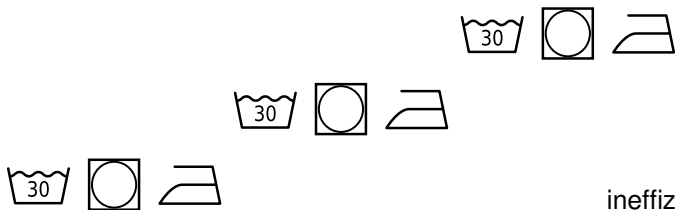
- Aufgabe in Teilaufgaben zerlegen
- Teilaufgaben parallel ausführen

Beispiel: Wäsche waschen

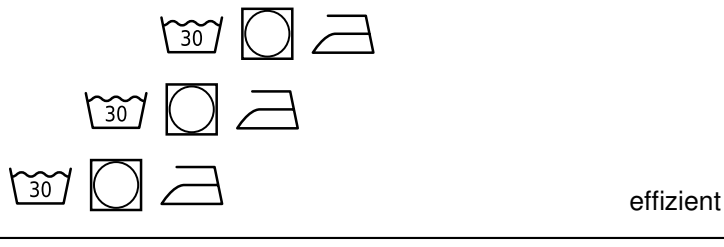
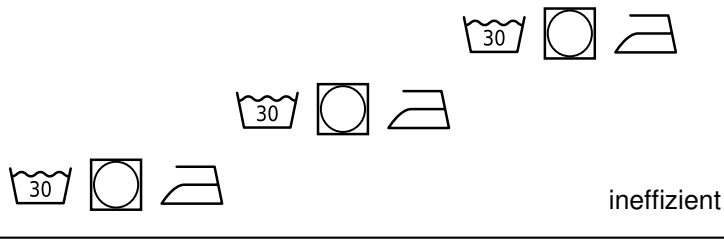
- Teilaufgaben: , , 
- müssen nacheinander ausgeführt werden: Datenfluß
- belegen jeweils 1 Ressource



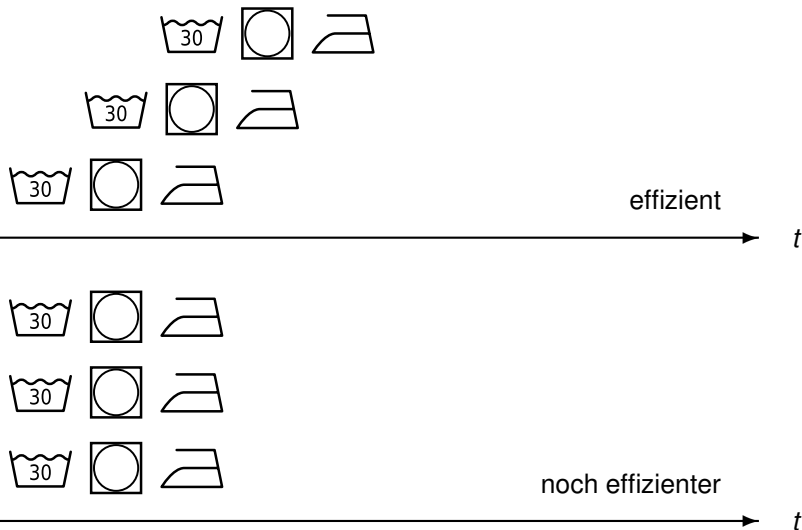
### 3 Ladungen Wäsche



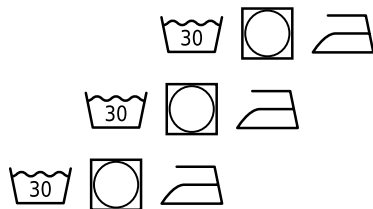
## 3 Ladungen Wäsche



### 3 Ladungen Wäsche

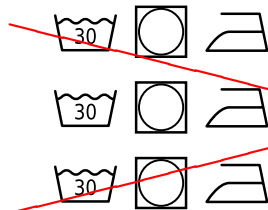


### 3 Ladungen Wäsche



effizient

$t$



Ressourcen-  
konflikt

noch effizienter

$t$

### 3 Ladungen Wäsche



Ressourcen-  
konflikt

noch effizienter

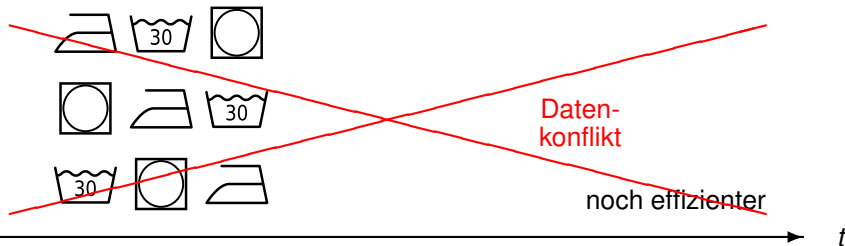
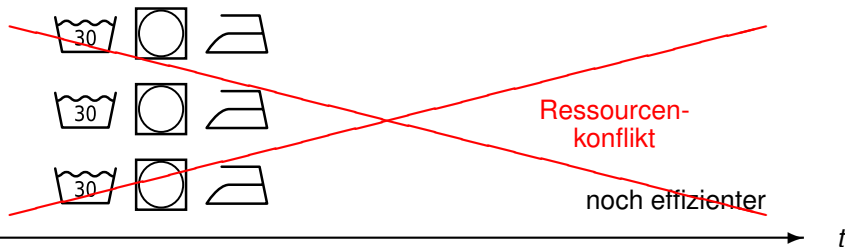
$t$



noch effizienter

$t$

### 3 Ladungen Wäsche



## 7.2 Arithmetik-Pipelines

„Register-FIFO“

## 7.2 Arithmetik-Pipelines

„Register-FIFO“

Pseudo-Code: Berechnung von

$$S = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3$$

mit einer Pipeline der Länge 3



## 7.2 Arithmetik-Pipelines

„Register-FIFO“

Pseudo-Code: Berechnung von

$$S = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3$$

mit einer Pipeline der Länge 3

push  $a_1 \cdot b_1$

## 7.2 Arithmetik-Pipelines

„Register-FIFO“

Pseudo-Code: Berechnung von

$$S = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3$$

mit einer Pipeline der Länge 3

push  $a_1 \cdot b_1$



## 7.2 Arithmetik-Pipelines

„Register-FIFO“

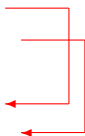
Pseudo-Code: Berechnung von

$$S = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3$$

mit einer Pipeline der Länge 3

push  $a_1 \cdot b_1$

push  $a_2 \cdot b_2$



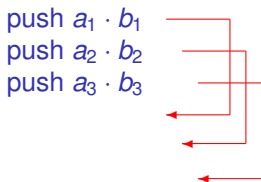
## 7.2 Arithmetik-Pipelines

„Register-FIFO“

Pseudo-Code: Berechnung von

$$S = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3$$

mit einer Pipeline der Länge 3



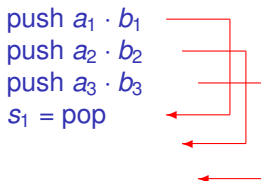
## 7.2 Arithmetik-Pipelines

„Register-FIFO“

Pseudo-Code: Berechnung von

$$S = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3$$

mit einer Pipeline der Länge 3



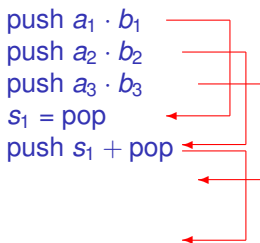
## 7.2 Arithmetik-Pipelines

„Register-FIFO“

Pseudo-Code: Berechnung von

$$S = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3$$

mit einer Pipeline der Länge 3



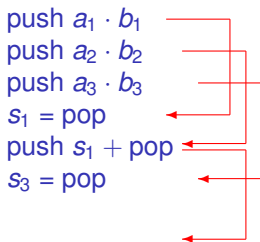
## 7.2 Arithmetik-Pipelines

„Register-FIFO“

Pseudo-Code: Berechnung von

$$S = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3$$

mit einer Pipeline der Länge 3



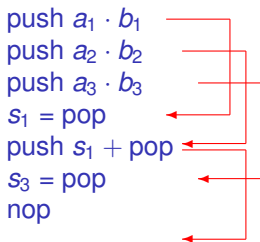
## 7.2 Arithmetik-Pipelines

„Register-FIFO“

Pseudo-Code: Berechnung von

$$S = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3$$

mit einer Pipeline der Länge 3





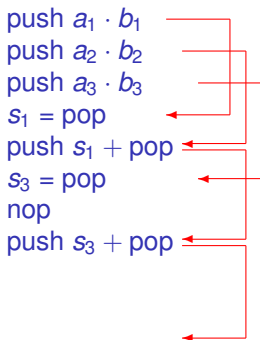
## 7.2 Arithmetik-Pipelines

„Register-FIFO“

Pseudo-Code: Berechnung von

$$S = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3$$

mit einer Pipeline der Länge 3



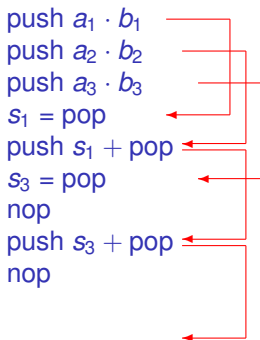
## 7.2 Arithmetik-Pipelines

„Register-FIFO“

Pseudo-Code: Berechnung von

$$S = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3$$

mit einer Pipeline der Länge 3





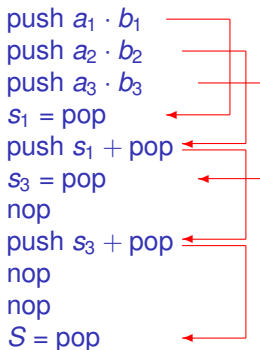
## 7.2 Arithmetik-Pipelines

„Register-FIFO“

Pseudo-Code: Berechnung von

$$S = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3$$

mit einer Pipeline der Länge 3



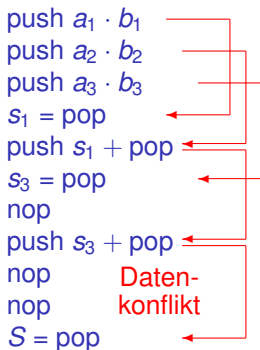
## 7.2 Arithmetik-Pipelines

„Register-FIFO“

Pseudo-Code: Berechnung von

$$S = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3$$

mit einer Pipeline der Länge 3



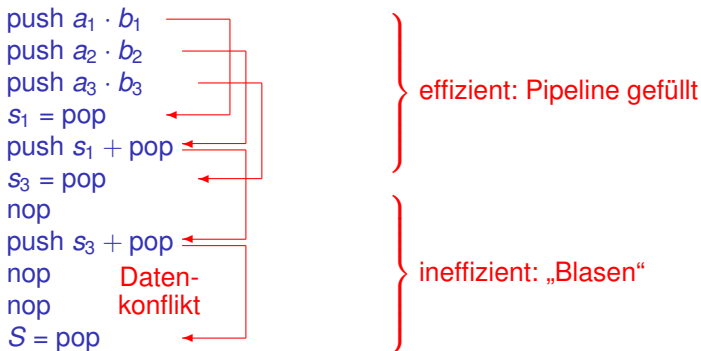
## 7.2 Arithmetik-Pipelines

„Register-FIFO“

Pseudo-Code: Berechnung von

$$S = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3$$

mit einer Pipeline der Länge 3



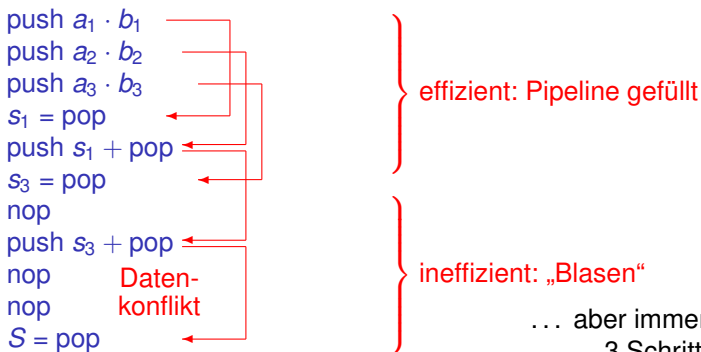
## 7.2 Arithmetik-Pipelines

„Register-FIFO“

Pseudo-Code: Berechnung von

$$S = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3$$

mit einer Pipeline der Länge 3



... aber immer noch effizienter als  
3 Schritte für jede Operation

## Reales Beispiel: Vektor-Addition auf i860

```
.align 8
.globl _vadd
_vadd:
    shr 1,r19,r19
    bte r19,r0,exitadd
    addu 0x000F,r16,r16
    andnot 0x000F,r16,r16
    adds -16,r16,r16
    addu 0x000F,r17,r17
    andnot 0x000F,r17,r17
    adds -16,r17,r17
    addu 0x000F,r18,r18
    andnot 0x000F,r18,r18
    adds -16,r18,r18
    mov -1,r20
```

```
fld.q 16(r16)++,f16
fld.q 16(r17)++,f20
pfadd.dd f16,f20,f0
bla r20,r19,loopadd
pfadd.dd f18,f22,f0
loopadd:
    d.pfadd.dd f0,f0,f0
    fld.q 16(r16)++,f16
    d.pfadd.dd f0,f0,f24
    fld.q 16(r17)++,f20
    d.pfadd.dd f16,f20,f26
    bla r20,r19,loopadd
    d.pfadd.dd f18,f22,f0
    fst.q f24,16(r18)++
    nop
    nop
    nop
exitadd:
    bri r1
    nop
```



## Reales Beispiel: Vektor-Addition auf i860


```
.align 8
.globl _vadd
_vadd:
    shr 1,r19,r19
    bte r19,r0,exitadd
    addu 0x000F,r16,r16
    andnot 0x000F,r16,r16
    adds -16,r16,r16
    addu 0x000F,r17,r17
    andnot 0x000F,r17,r17
    adds -16,r17,r17
    addu 0x000F,r18,r18
    andnot 0x000F,r18,r18
    adds -16,r18,r18
    mov -1,r20
```

```
fld.q 16(r16)++,f16
fld.q 16(r17)++,f20
pfadd.dd f16,f20,f0
bla r20,r19,loopadd
pfadd.dd f18,f22,f0
loopadd:
    d.pfadd.dd f0,f0,f0
    fld.q 16(r16)++,f16
    d.pfadd.dd f0,f0,f24
    fld.q 16(r17)++,f20
    d.pfadd.dd f16,f20,f26
    bla r20,r19,loopadd
    d.pfadd.dd f18,f22,f0
    fst.q f24,16(r18)++
    nop
    nop
    nop
exitadd:
    bri r1
    nop
```

## Reales Beispiel: Vektor-Addition auf i860

```
.align 8
.globl _vadd
_vadd:
    shr 1,r19,r19
    bte r19,r0,exitadd
    addu 0x000F,r16,r16
    andnot 0x000F,r16,r16
    adds -16,r16,r16
    addu 0x000F,r17,r17
    andnot 0x000F,r17,r17
    adds -16,r17,r17
    addu 0x000F,r18,r18
    andnot 0x000F,r18,r18
    adds -16,r18,r18
    mov -1,r20
```


```
fld.q 16(r16)++,f16
fld.q 16(r17)++,f20
pfadd.dd f16,f20,f0
bla r20,r19,loopadd
pfadd.dd f18,f22,f0
loopadd:
    d.pfadd.dd f0,f0,f0
    fld.q 16(r16)++,f16
    d.pfadd.dd f0,f0,f24
    fld.q 16(r17)++,f20
    d.pfadd.dd f16,f20,f26
    bla r20,r19,loopadd
    d.pfadd.dd f18,f22,f0
    fst.q f24,16(r18)++
    nop
    nop
    nop
exitadd:
    bri r1
    nop
```



## Reales Beispiel: Vektor-Addition auf i860

```
.align 8
.globl _vadd
_vadd:
    shr 1,r19,r19
    bte r19,r0,exitadd
    addu 0x000F,r16,r16
    andnot 0x000F,r16,r16
    adds -16,r16,r16
    addu 0x000F,r17,r17
    andnot 0x000F,r17,r17
    adds -16,r17,r17
    addu 0x000F,r18,r18
    andnot 0x000F,r18,r18
    adds -16,r18,r18
    mov -1,r20
```

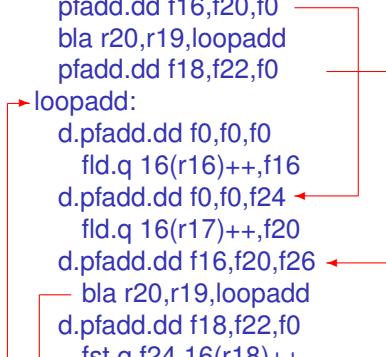
```
fld.q 16(r16)++,f16
fld.q 16(r17)++,f20
pfadd.dd f16,f20,f0
bla r20,r19,loopadd
pfadd.dd f18,f22,f0
loopadd:
    d.pfadd.dd f0,f0,f0
    fld.q 16(r16)++,f16
    d.pfadd.dd f0,f0,f24
    fld.q 16(r17)++,f20
    d.pfadd.dd f16,f20,f26
    bla r20,r19,loopadd
    d.pfadd.dd f18,f22,f0
    fst.q f24,16(r18)++
    nop
    nop
    nop
exitadd:
    bri r1
    nop
```



## Reales Beispiel: Vektor-Addition auf i860

```
.align 8
.globl _vadd
_vadd:
    shr 1,r19,r19
    bte r19,r0,exitadd
    addu 0x000F,r16,r16
    andnot 0x000F,r16,r16
    adds -16,r16,r16
    addu 0x000F,r17,r17
    andnot 0x000F,r17,r17
    adds -16,r17,r17
    addu 0x000F,r18,r18
    andnot 0x000F,r18,r18
    adds -16,r18,r18
    mov -1,r20
```

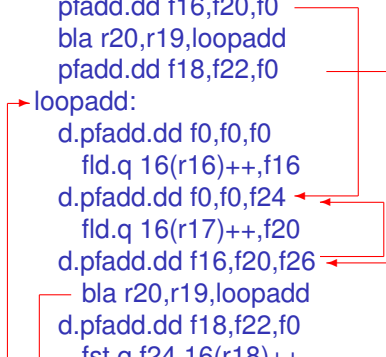
```
fld.q 16(r16)++,f16
fld.q 16(r17)++,f20
pfadd.dd f16,f20,f0
bla r20,r19,loopadd
pfadd.dd f18,f22,f0
loopadd:
    d.pfadd.dd f0,f0,f0
    fld.q 16(r16)++,f16
    d.pfadd.dd f0,f0,f24
    fld.q 16(r17)++,f20
    d.pfadd.dd f16,f20,f26
    bla r20,r19,loopadd
    d.pfadd.dd f18,f22,f0
    fst.q f24,16(r18)++
    nop
    nop
    nop
exitadd:
    bri r1
    nop
```



## Reales Beispiel: Vektor-Addition auf i860

```
.align 8
.globl _vadd
_vadd:
    shr 1,r19,r19
    bte r19,r0,exitadd
    addu 0x000F,r16,r16
    andnot 0x000F,r16,r16
    adds -16,r16,r16
    addu 0x000F,r17,r17
    andnot 0x000F,r17,r17
    adds -16,r17,r17
    addu 0x000F,r18,r18
    andnot 0x000F,r18,r18
    adds -16,r18,r18
    mov -1,r20
```

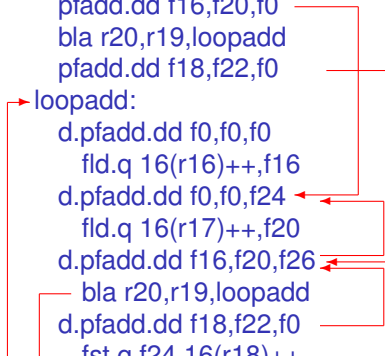
```
fld.q 16(r16)++,f16
fld.q 16(r17)++,f20
pfadd.dd f16,f20,f0
bla r20,r19,loopadd
pfadd.dd f18,f22,f0
loopadd:
    d.pfadd.dd f0,f0,f0
    fld.q 16(r16)++,f16
    d.pfadd.dd f0,f0,f24
    fld.q 16(r17)++,f20
    d.pfadd.dd f16,f20,f26
    bla r20,r19,loopadd
    d.pfadd.dd f18,f22,f0
    fst.q f24,16(r18)++
    nop
    nop
    nop
exitadd:
    bri r1
    nop
```



## Reales Beispiel: Vektor-Addition auf i860

```
.align 8
.globl _vadd
_vadd:
    shr 1,r19,r19
    bte r19,r0,exitadd
    addu 0x000F,r16,r16
    andnot 0x000F,r16,r16
    adds -16,r16,r16
    addu 0x000F,r17,r17
    andnot 0x000F,r17,r17
    adds -16,r17,r17
    addu 0x000F,r18,r18
    andnot 0x000F,r18,r18
    adds -16,r18,r18
    mov -1,r20
```

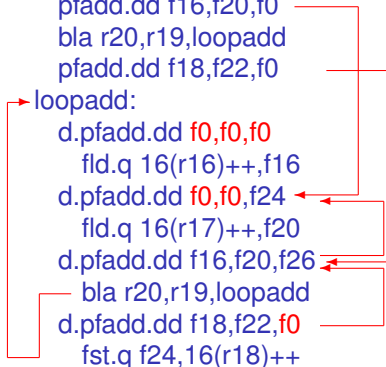
```
fld.q 16(r16)++,f16
fld.q 16(r17)++,f20
pfadd.dd f16,f20,f0
bla r20,r19,loopadd
pfadd.dd f18,f22,f0
loopadd:
    d.pfadd.dd f0,f0,f0
    fld.q 16(r16)++,f16
    d.pfadd.dd f0,f0,f24
    fld.q 16(r17)++,f20
    d.pfadd.dd f16,f20,f26
    bla r20,r19,loopadd
    d.pfadd.dd f18,f22,f0
    fst.q f24,16(r18)++
    nop
    nop
    nop
exitadd:
    bri r1
    nop
```



## Reales Beispiel: Vektor-Addition auf i860

```
.align 8
.globl _vadd
_vadd:
    shr 1,r19,r19
    bte r19,r0,exitadd
    addu 0x000F,r16,r16
    andnot 0x000F,r16,r16
    adds -16,r16,r16
    addu 0x000F,r17,r17
    andnot 0x000F,r17,r17
    adds -16,r17,r17
    addu 0x000F,r18,r18
    andnot 0x000F,r18,r18
    adds -16,r18,r18
    mov -1,r20
```

```
fld.q 16(r16)++,f16
fld.q 16(r17)++,f20
pfadd.dd f16,f20,f0
bla r20,r19,loopadd
pfadd.dd f18,f22,f0
loopadd:
    d.pfadd.dd f0,f0,f0
    fld.q 16(r16)++,f16
    d.pfadd.dd f0,f0,f24
    fld.q 16(r17)++,f20
    d.pfadd.dd f16,f20,f26
    bla r20,r19,loopadd
    d.pfadd.dd f18,f22,f0
    fst.q f24,16(r18)++
```



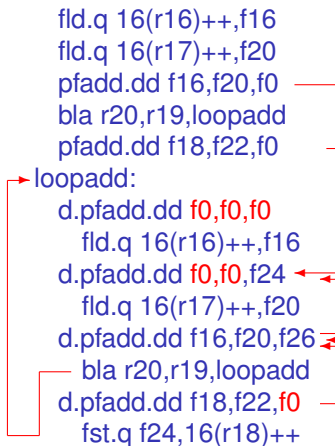
```
    nop
    nop
    nop
exitadd:
    bri r1
    nop
```

6mal f0 = 2 Blasen

## Reales Beispiel: Vektor-Addition auf i860

```
.align 8
.globl _vadd
_vadd:
    shr 1,r19,r19
    bte r19,r0,exitadd
    addu 0x000F,r16,r16
    andnot 0x000F,r16,r16
    adds -16,r16,r16
    addu 0x000F,r17,r17
    andnot 0x000F,r17,r17
    adds -16,r17,r17
    addu 0x000F,r18,r18
    andnot 0x000F,r18,r18
    adds -16,r18,r18
    mov -1,r20
```

```
fld.q 16(r16)++,f16
fld.q 16(r17)++,f20
pfadd.dd f16,f20,f0
bla r20,r19,loopadd
pfadd.dd f18,f22,f0
loopadd:
    d.pfadd.dd f0,f0,f0
    fld.q 16(r16)++,f16
    d.pfadd.dd f0,f0,f24
    fld.q 16(r17)++,f20
    d.pfadd.dd f16,f20,f26
    bla r20,r19,loopadd
    d.pfadd.dd f18,f22,f0
    fst.q f24,16(r18)++
```



6mal f0 = 2 Blasen

Immerhin: 2 Additionen in 4 Taktzyklen

```
nop
nop
nop
exitadd:
    bri r1
    nop
```



## Reales Beispiel: Vektor-Addition auf i860

```
.align 8
.globl _vadd
_vadd:
    shr 1,r19,r19
    bte r19,r0,exitadd
    addu 0x000F,r16,r16
    andnot 0x000F,r16,r16
    adds -16,r16,r16
    addu 0x000F,r17,r17
    andnot 0x000F,r17,r17
    adds -16,r17,r17
    addu 0x000F,r18,r18
    andnot 0x000F,r18,r18
    adds -16,r18,r18
    mov -1,r20
```

```
fld.q 16(r16)++,f16
fld.q 16(r17)++,f20
pfadd.dd f16,f20,f0
bla r20,r19,loopadd
pfadd.dd f18,f22,f0
loopadd:
    d.pfadd.dd f0,f0,f0
    fld.q 16(r16)++,f16
    d.pfadd.dd f0,f0,f24
    fld.q 16(r17)++,f20
    d.pfadd.dd f16,f20,f26
    bla r20,r19,loopadd
    d.pfadd.dd f18,f22,f0
    fst.q f24,16(r18)++
nop
nop
nop
exitadd:
    bri r1
nop
```

6mal f0 = 2 Blasen

Immerhin: 2 Additionen in 4 Taktzyklen

Dies ist ein *einfaches* Beispiel.

## 7.3 Instruktions-Pipelines

Ein Prozessor benötigt Zeit, um einen Befehl zu verstehen.

→ Während Befehlsausführung nächste Befehle vorauslesen

.L3:

```
movw r30,r20
add r30,r18
adc r31,r19
mov r24,r18
subi r24,lo8(-(1))
st Z,r24
subi r18,lo8(-(1))
sbci r19,hi8(-(1))
cp r22,r18
cpc r23,r19
brge .L3
ret
```

## 7.3 Instruktions-Pipelines

Ein Prozessor benötigt Zeit, um einen Befehl zu verstehen.

→ Während Befehlsausführung nächste Befehle vorauslesen

.L3:

movw r30,r20

add r30,r18

adc r31,r19

mov r24,r18

subi r24,lo8(-(1))

st Z,r24

subi r18,lo8(-(1))

sbc r19,hi8(-(1))

cp r22,r18

cpc r23,r19

brge .L3

ret

## 7.3 Instruktions-Pipelines

Ein Prozessor benötigt Zeit, um einen Befehl zu verstehen.

→ Während Befehlsausführung nächste Befehle vorauslesen

.L3:

movw r30,r20

add r30,r18

adc r31,r19

mov r24,r18

subi r24,lo8(-(1))

st Z,r24

subi r18,lo8(-(1))

sbc r19,hi8(-(1))

cp r22,r18

cpc r23,r19

brge .L3

ret

## 7.3 Instruktions-Pipelines

Ein Prozessor benötigt Zeit, um einen Befehl zu verstehen.

→ Während Befehlsausführung nächste Befehle vorauslesen

.L3:

movw r30,r20

add r30,r18

adc r31,r19

mov r24,r18

subi r24,lo8(-(1))

st Z,r24

subi r18,lo8(-(1))

sbc r19,hi8(-(1))

cp r22,r18

cpc r23,r19

brge .L3

ret

## 7.3 Instruktions-Pipelines

Ein Prozessor benötigt Zeit, um einen Befehl zu verstehen.

→ Während Befehlsausführung nächste Befehle vorauslesen

.L3:

```
movw r30,r20
add r30,r18
adc r31,r19
mov r24,r18
subi r24,lo8(-(1))
st Z,r24
subi r18,lo8(-(1))
sbci r19,hi8(-(1))
cp r22,r18
cpc r23,r19
brge .L3
ret
```

## 7.3 Instruktions-Pipelines

Ein Prozessor benötigt Zeit, um einen Befehl zu verstehen.

→ Während Befehlsausführung nächste Befehle vorauslesen

.L3:

```
movw r30,r20
add r30,r18
adc r31,r19
mov r24,r18
subi r24,lo8(-(1))
st Z,r24
subi r18,lo8(-(1))
sbci r19,hi8(-(1))
cp r22,r18
cpc r23,r19
brge .L3
ret
```

## 7.3 Instruktions-Pipelines

Ein Prozessor benötigt Zeit, um einen Befehl zu verstehen.

→ Während Befehlsausführung nächste Befehle vorauslesen

.L3:

```
movw r30,r20
add r30,r18
adc r31,r19
mov r24,r18
subi r24,lo8(-(1))
st Z,r24
subi r18,lo8(-(1))
sbci r19,hi8(-(1))
cp r22,r18
cpc r23,r19
brge .L3
ret
```




## 7.3 Instruktions-Pipelines

Ein Prozessor benötigt Zeit, um einen Befehl zu verstehen.

→ Während Befehlsausführung nächste Befehle vorauslesen

.L3:



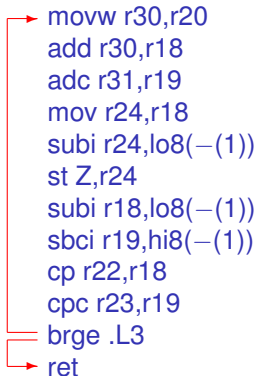
```
→ movw r30,r20
   add r30,r18
   adc r31,r19
   mov r24,r18
   subi r24,lo8(-(1))
   st Z,r24
   subi r18,lo8(-(1))
   sbci r19,hi8(-(1))
   cp r22,r18
   cpc r23,r19
   brge .L3
   ret
```

## 7.3 Instruktions-Pipelines

Ein Prozessor benötigt Zeit, um einen Befehl zu verstehen.

→ Während Befehlsausführung nächste Befehle vorauslesen

.L3:



```
→ movw r30,r20
  add r30,r18
  adc r31,r19
  mov r24,r18
  subi r24,lo8(-(1))
  st Z,r24
  subi r18,lo8(-(1))
  sbci r19,hi8(-(1))
  cp r22,r18
  cpc r23,r19
  brge .L3
→ ret
```

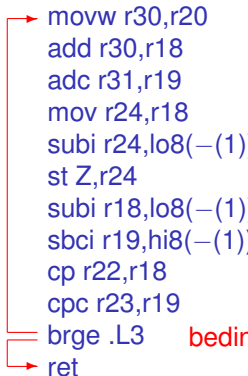
The diagram shows a sequence of assembly instructions. A red arrow on the left side of the code block starts at the first instruction, 'movw r30,r20', and extends downwards. It then turns to the right, pointing to the 'brge .L3' instruction, which is a branch back to the label '.L3:'. This illustrates the loop structure of the code.

## 7.3 Instruktions-Pipelines

Ein Prozessor benötigt Zeit, um einen Befehl zu verstehen.

→ Während Befehlsausführung nächste Befehle vorauslesen

.L3:



A red arrow originates from the right side of the `brge .L3` instruction and points back to the `movw r30,r20` instruction, indicating a loop.

```
→ movw r30,r20
  add r30,r18
  adc r31,r19
  mov r24,r18
  subi r24,lo8(-(1))
  st Z,r24
  subi r18,lo8(-(1))
  sbci r19,hi8(-(1))
  cp r22,r18
  cpc r23,r19
  brge .L3
→ ret
```

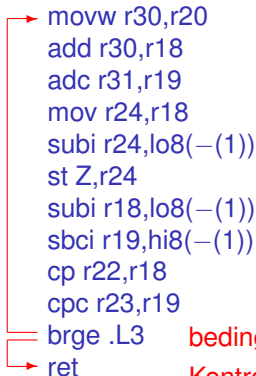
bedingter Sprung: Welche Befehle vorauslesen?

## 7.3 Instruktions-Pipelines

Ein Prozessor benötigt Zeit, um einen Befehl zu verstehen.

→ Während Befehlsausführung nächste Befehle vorauslesen

.L3:



```
→ movw r30,r20
  add r30,r18
  adc r31,r19
  mov r24,r18
  subi r24,lo8(-(1))
  st Z,r24
  subi r18,lo8(-(1))
  sbci r19,hi8(-(1))
  cp r22,r18
  cpc r23,r19
  brge .L3
→ ret
```

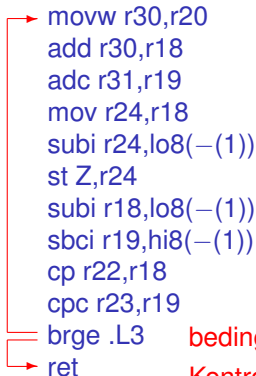
bedingter Sprung: Welche Befehle vorauslesen?  
Kontrollflußkonflikt

## 7.3 Instruktions-Pipelines

Ein Prozessor benötigt Zeit, um einen Befehl zu verstehen.

→ Während Befehlsausführung nächste Befehle vorauslesen

.L3:



```
→ movw r30,r20
  add r30,r18
  adc r31,r19
  mov r24,r18
  subi r24,lo8(-(1))
  st Z,r24
  subi r18,lo8(-(1))
  sbci r19,hi8(-(1))
  cp r22,r18
  cpc r23,r19
  brge .L3
→ ret
```

bedingter Sprung: Welche Befehle vorauslesen?  
Kontrollflußkonflikt

Lösungsansatz: Zweigvorhersage

## 7.3 Instruktions-Pipelines

### Zweigvorhersage – Branch Prediction

## 7.3 Instruktions-Pipelines

### Zweigvorhersage – Branch Prediction

- Sprünge nach oben sind Schleifen: „Ja“  
Sprünge nach unten sind Auswahl-Verzweigungen: „Nein“

## 7.3 Instruktions-Pipelines

### Zweigvorhersage – Branch Prediction

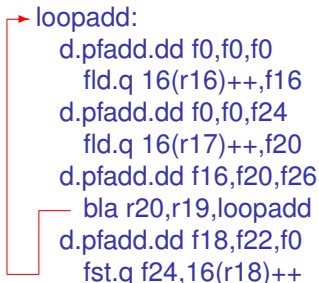
- Sprünge nach oben sind Schleifen: „Ja“  
Sprünge nach unten sind Auswahl-Verzweigungen: „Nein“
- Delayed Branches: Sprungbefehl verspätet ausführen  
→ Optimierung manuell oder durch Compiler



## 7.3 Instruktions-Pipelines

### Zweigvorhersage – Branch Prediction

- Sprünge nach oben sind Schleifen: „Ja“  
Sprünge nach unten sind Auswahl-Verzweigungen: „Nein“
- Delayed Branches: Sprungbefehl verspätet ausführen  
→ Optimierung manuell oder durch Compiler

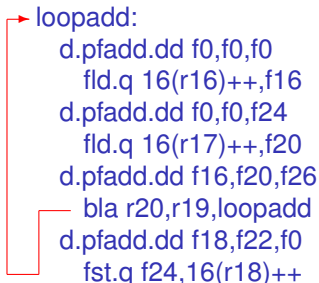


```
loopadd:
    d.pfadd.dd f0,f0,f0
    fld.q 16(r16)++,f16
    d.pfadd.dd f0,f0,f24
    fld.q 16(r17)++,f20
    d.pfadd.dd f16,f20,f26
    bla r20,r19,loopadd
    d.pfadd.dd f18,f22,f0
    fst.q f24,16(r18)++
```

## 7.3 Instruktions-Pipelines

### Zweigvorhersage – Branch Prediction

- Sprünge nach oben sind Schleifen: „Ja“  
Sprünge nach unten sind Auswahl-Verzweigungen: „Nein“
- Delayed Branches: Sprungbefehl verspätet ausführen  
→ Optimierung manuell oder durch Compiler



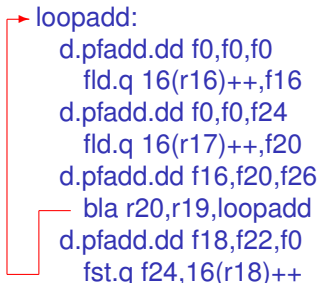
```
loopadd:
    d.pfadd.dd f0,f0,f0
    fld.q 16(r16)++,f16
    d.pfadd.dd f0,f0,f24
    fld.q 16(r17)++,f20
    d.pfadd.dd f16,f20,f26
    bla r20,r19,loopadd
    d.pfadd.dd f18,f22,f0
    fst.q f24,16(r18)++
```

- Branch History Table: Sprünge merken

## 7.3 Instruktions-Pipelines

### Zweigvorhersage – Branch Prediction

- Sprünge nach oben sind Schleifen: „Ja“  
Sprünge nach unten sind Auswahl-Verzweigungen: „Nein“
- Delayed Branches: Sprungbefehl verspätet ausführen  
→ Optimierung manuell oder durch Compiler



```
→ loopadd:
    d.pfadd.dd f0,f0,f0
    fld.q 16(r16)++,f16
    d.pfadd.dd f0,f0,f24
    fld.q 16(r17)++,f20
    d.pfadd.dd f16,f20,f26
    bla r20,r19,loopadd
    d.pfadd.dd f18,f22,f0
    fst.q f24,16(r18)++
```

- Branch History Table: Sprünge merken
- ...

# 7 Pipelining

## Zusammenfassung

- Teilaufgaben parallel ausführen
- Arithmetik-Pipelines führen Berechnungen parallel aus, Instruktions-Pipelines lesen Befehle voraus
- Ressourcen-, Daten- und Kontrollflußkonflikte führen zu „Blasen“
- Zweigvorhersage reduziert Kontrollflußkonflikte in Instruktions-Pipelines
  - nach oben / nach unten
  - Delayed Branches: manuell optimieren
  - Branch History Table: Sprünge merken