

# Rechnertechnik

Prof. Dr. rer. nat. Peter Gerwinski

14. Juni 2022

# Rechnertechnik

- 1 Einführung**
- 2 Vom Schaltkreis zum Computer**
- 3 Architekturmerkmale von Prozessoren**
  - 3.1 Speicherarchitekturen
  - 3.2 Registerarchitekturen
  - 3.3 Befehlssätze
- 4 Der CPU-Stack**
  - 4.0 Was ist ein Stack?
  - 4.1 Implementation
  - 4.2 Unterprogramm
  - 4.3 Register sichern
  - 4.4 Puffer-Überläufe
  - 4.5 Stack-Überläufe
- 5 Anwender-Software**
- 6 Pipelining**

# 3 Architekturmerkmale von Prozessoren

## 3.1 Speicherarchitekturen

### Bezeichnungen

- *Bit* = 0 oder 1 – kleinste Einheit an Information
- *Byte* = Zusammenfassung mehrerer *Bits*  
zu einer Binärzahl, die ein Zeichen (*Character*) darstellen kann,  
häufig 8 Bits (*Oktett*)
- *Speicherwort* = Zusammenfassung mehrerer Bits  
zu der kleinsten adressierbaren Einheit, häufig 1 Byte
- *RAM* = *Random Access Memory* = Hauptspeicher
- *ROM* = *Read Only Memory* = nur lesbarer Speicher

## 3.1 Speicherarchitekturen

Verschiedene Arten von Speicher

- *Prozessor-Register*  
können direkt mit ALU verbunden werden,  
besonders schnell (Flipflops),  
überschaubare Anzahl von Registern
- *Hauptspeicher*  
kann direkt adressiert und mit Prozessor-Registern abgeglichen werden,  
heute i. d. R. dynamischer Speicher (Kondensatoren)
- *I/O-Ports*  
sind spezielle Speicheradressen, über die  
mit externen Geräten kommuniziert wird
- *Massenspeicher*  
liegt auf externem Gerät, wird über I/O-Ports angesprochen,  
Festplatte, Flash-Speicher, ...

## 3.1 Speicherarchitekturen

- *Von-Neumann-Architektur*

Es gibt nur 1 Hauptspeicher, in dem sich sowohl die Befehle als auch die Daten befinden.

Vorteil: Flexibilität in der Speichernutzung

Nachteil: Befehle können überschrieben werden.

—→ Abstürze und Malware möglich

## 3.1 Speicherarchitekturen

- *Von-Neumann-Architektur*

Es gibt nur 1 Hauptspeicher, in dem sich sowohl die Befehle als auch die Daten befinden.

- *Harvard-Architektur*

Es gibt 2 Hauptspeicher. In einem befinden sich die Befehle, im anderen die Daten.

Vorteil: Befehle können nicht überschrieben werden

—> sicherer als Von-Neumann-Architektur

Nachteile: Leitungen zum Speicher (Bus) müssen doppelt vorhanden sein, freier Befehlsspeicher kann nicht für Daten genutzt werden.

## 3.1 Speicherarchitekturen

- *Von-Neumann-Architektur*

Es gibt nur 1 Hauptspeicher, in dem sich sowohl die Befehle als auch die Daten befinden.

- *Harvard-Architektur*

Es gibt 2 Hauptspeicher. In einem befinden sich die Befehle, im anderen die Daten.

- Weitere Kombinationen

Hauptspeicher und I/O-Ports gemeinsam oder getrennt,  
Hauptspeicher und Prozessorregister gemeinsam oder getrennt

## 3.1 Speicherarchitekturen

Beispiele:

- Intel IA-32 (i386, Nachfolger und Kompatible):  
Von-Neumann-Architektur (plus Speicherschutzmechanismen),  
Prozessorregister und I/O-Ports vom Hauptspeicher getrennt
- Atmel AVR (z. B. ATmega):  
Harvard-Architektur (Befehlsspeicher als Flash-Speicher grundsätzlich auch  
schreibbar),  
Prozessorregister und I/O-Ports in gemeinsamem Adressbereich mit  
Hauptspeicher
- 6502, Renesas-Mikro-Controller:  
Von-Neumann-Architektur,  
I/O-Ports in gemeinsamem Adressbereich mit Hauptspeicher,  
Prozessorregister und Hauptspeicher getrennt



## 3.2 Registerarchitekturen

- Mehrere Register, einzeln ansprechbar
- *Akkumulator*: Nur 1 Register kann rechnen.
- *Stack-Architektur*: Stapel, „umgekehrte Polnische Notation“

## 3.2 Registerarchitekturen

- Mehrere Register, einzeln ansprechbar
- *Akkumulator*: Nur 1 Register kann rechnen.
- *Stack-Architektur*: Stapel, „umgekehrte Polnische Notation“

Operationen: typischerweise nur  $=$ ,  $+=$ ,  $-=$ ,  $*=$ ,  $/=$ , ...

## 3.2 Registerarchitekturen

- Mehrere Register, einzeln ansprechbar
- *Akkumulator*: Nur 1 Register kann rechnen.
- *Stack-Architektur*: Stapel, „umgekehrte Polnische Notation“

Operationen: typischerweise nur  $=$ ,  $+=$ ,  $-=$ ,  $*=$ ,  $/=$ , ...

Beispiel:  $c = a + 2 * b;$

## 3.2 Registerarchitekturen

- Mehrere Register, einzeln ansprechbar
- *Akkumulator*: Nur 1 Register kann rechnen.
- *Stack-Architektur*: Stapel, „umgekehrte Polnische Notation“

Operationen: typischerweise nur  $=$ ,  $+=$ ,  $-=$ ,  $*=$ ,  $/=$ , ...

Beispiel:  $c = a + 2 * b;$

C:

```
R = b;  
R *= 2;  
R += a;  
c = R;
```

## 3.2 Registerarchitekturen

- Mehrere Register, einzeln ansprechbar
- *Akkumulator*: Nur 1 Register kann rechnen.
- *Stack-Architektur*: Stapel, „umgekehrte Polnische Notation“

Operationen: typischerweise nur  $=$ ,  $+=$ ,  $-=$ ,  $*=$ ,  $/=$ , ...

Beispiel:  $c = a + 2 * b$ ;

C:            Mehrere Register:

```
R = b;      movl (b), %eax
R *= 2;     imull $2, %eax, %eax
R += a;     addl (a), %eax
c = R;      movl %eax, (c)
            (IA-32-Assembler)
```

## 3.2 Registerarchitekturen

- Mehrere Register, einzeln ansprechbar
- *Akkumulator*: Nur 1 Register kann rechnen.
- *Stack-Architektur*: Stapel, „umgekehrte Polnische Notation“

Operationen: typischerweise nur  $=$ ,  $+=$ ,  $-=$ ,  $*=$ ,  $/=$ , ...

Beispiel:  $c = a + 2 * b$ ;

C:	Mehrere Register:	Akkumulator:
$R = b$ ;	<code>movl (b), %eax</code>	<code>load (b)</code>
$R *= 2$ ;	<code>imull \$2, %eax, %eax</code>	<code>mul \$2</code>
$R += a$ ;	<code>addl (a), %eax</code>	<code>add (a)</code>
$c = R$ ;	<code>movl %eax, (c)</code>	<code>store (c)</code>
	(IA-32-Assembler)	(Pseudo-Assembler)

## 3.2 Registerarchitekturen

- Mehrere Register, einzeln ansprechbar
- *Akkumulator*: Nur 1 Register kann rechnen.
- *Stack-Architektur*: Stapel, „umgekehrte Polnische Notation“

Operationen: typischerweise nur  $=$ ,  $+=$ ,  $-=$ ,  $*=$ ,  $/=$ , ...

Beispiel:  $c = a + 2 * b$ ;

C:	Mehrere Register:	Akkumulator:	Register-Stack:
$R = b$ ;	<code>movl (b), %eax</code>	<code>load (b)</code>	<code>push (a)</code>
$R *= 2$ ;	<code>imull \$2, %eax, %eax</code>	<code>mul \$2</code>	<code>push (b)</code>
$R += a$ ;	<code>addl (a), %eax</code>	<code>add (a)</code>	<code>push \$2</code>
$c = R$ ;	<code>movl %eax, (c)</code>	<code>store (c)</code>	<code>mul</code>
	(IA-32-Assembler)	(Pseudo-Assembler)	<code>add</code>
			<code>pop (c)</code>

## 3.2 Registerarchitekturen

Beispiele:

- Intel IA-32 (i386, Nachfolger und Kompatible):  
Mehrere Register, für verschiedene Zwecke spezialisiert (unübersichtlich),  
Fließkommaregister: Stack-Architektur
- Atmel AVR (z. B. ATmega):  
32 Register
- 6502, Renesas-Mikro-Controller:  
3 Register: A, X, Y. Nur A kann rechnen → Akkumulator
- Java Virtual Machine (JVM):  
Stack-Architektur
- Redcode:  
Jede Speicherzelle fungiert als Register



## 3.3 Befehlssätze

- *Complex Instruction Set Computer (CISC)*

Umfangreiche Befehlssätze, mächtige Befehle

→ komfortable manuelle Programmierung in Assembler

→ längere Abarbeitungszeit der einzelnen Befehle

Realisierung: „Prozessor im Prozessor“ – *Mikroprogramme*

Beispiele: IA-32, AMD-64

## 3.3 Befehlssätze

- *Complex Instruction Set Computer (CISC)*

Umfangreiche Befehlssätze, mächtige Befehle

- *Reduced Instruction Set Computer (RISC)*

wenige, wenig mächtige Befehle

→ Programmierung in Assembler für Menschen unkomfortabel

→ schnelle Abarbeitung der Befehle

Beispiele: Atmel AVR, Redcode

## 3.3 Befehlssätze

- *Complex Instruction Set Computer (CISC)*  
Umfangreiche Befehlssätze, mächtige Befehle
- *Reduced Instruction Set Computer (RISC)*  
wenige, wenig mächtige Befehle
- *Very Long Instruction Word (VLIW)* und  
*Explicitly Parallel Instruction Computing (EPIC)*  
mehrere Befehle gleichzeitig ausführbar  
→ mehr Rechenleistung möglich  
→ Programmierung sehr aufwendig  
Beispiel: IA-64

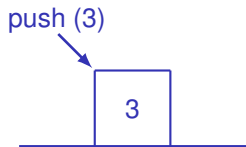
## 3.3 Befehlssätze

- *Complex Instruction Set Computer (CISC)*  
Umfangreiche Befehlssätze, mächtige Befehle
- *Reduced Instruction Set Computer (RISC)*  
wenige, wenig mächtige Befehle
- *Very Long Instruction Word (VLIW)* und  
*Explicitly Parallel Instruction Computing (EPIC)*  
mehrere Befehle gleichzeitig ausführbar
- *Orthogonaler Befehlssatz*  
jeder Befehl mit jeder Adressierungsart kombinierbar

## 4 Der CPU-Stack

### 4.0 Was ist ein Stack?

„Last In – First Out“

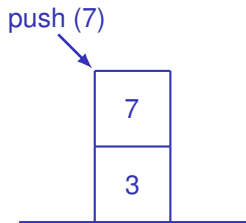


LIFO = Stack = Stapel

## 4 Der CPU-Stack

### 4.0 Was ist ein Stack?

„Last In – First Out“



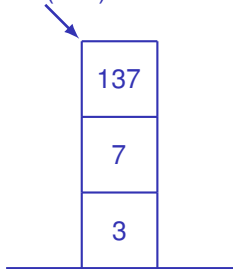
LIFO = Stack = Stapel

## 4 Der CPU-Stack

### 4.0 Was ist ein Stack?

„Last In – First Out“

push (137)

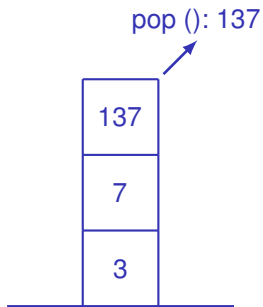


LIFO = Stack = Stapel

## 4 Der CPU-Stack

### 4.0 Was ist ein Stack?

„Last In – First Out“



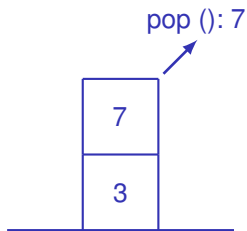
LIFO = Stack = Stapel



## 4 Der CPU-Stack

### 4.0 Was ist ein Stack?

„Last In – First Out“

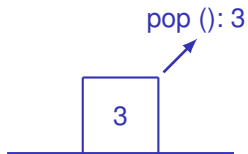


LIFO = Stack = Stapel

## 4 Der CPU-Stack

### 4.0 Was ist ein Stack?

„Last In – First Out“



LIFO = Stack = Stapel

# 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*:  $bar = *--SP;$

## 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 Puffer-Überläufe

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

```
int main (void)
```

```
{
```

```
    char name[10];
```

```
    printf ("Ihr_Name:_");
```

```
    gets (name);
```

```
    printf ("Hallo,_%s!:_-)\n", name);
```

```
    return 0;
```

```
}
```

## 4 Der CPU-Stack

### 4.5 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;
```

```
}
```

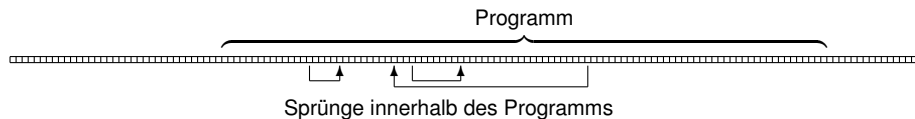
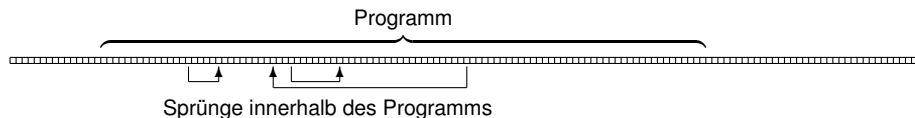
Bei jedem Aufruf wird die Rücksprungadresse auf den Stack gelegt und die Variable `n` auf dem Stack gesichert.



# 5 Anwender-Software

## 5.1 Relokation und Linken

Software im Speicher

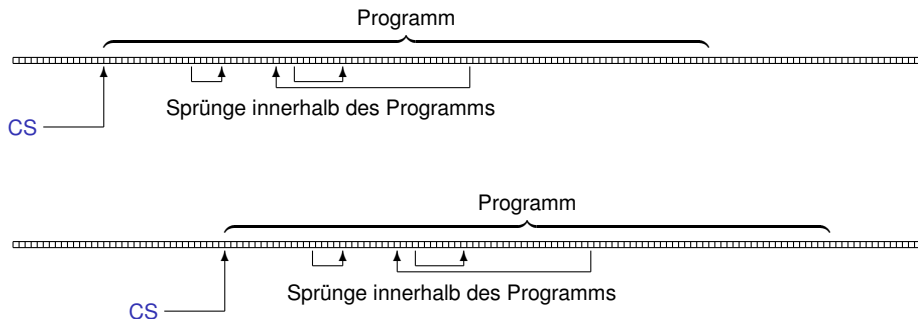


Sprünge anpassen: Relokation

# 5 Anwender-Software

## 5.1 Relokation und Linken

### Software im Speicher

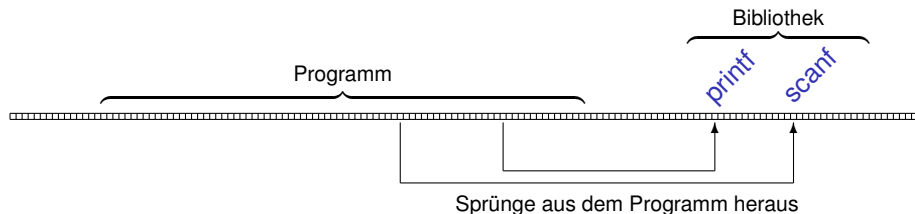
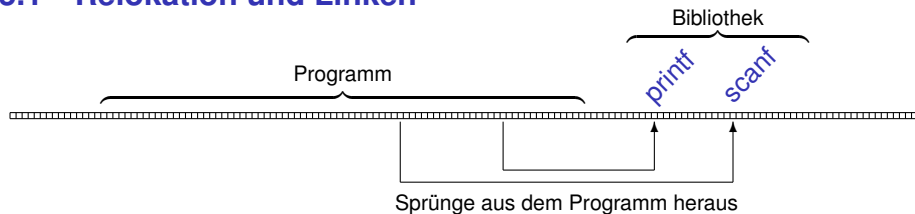


Sprünge anpassen: Relokation

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

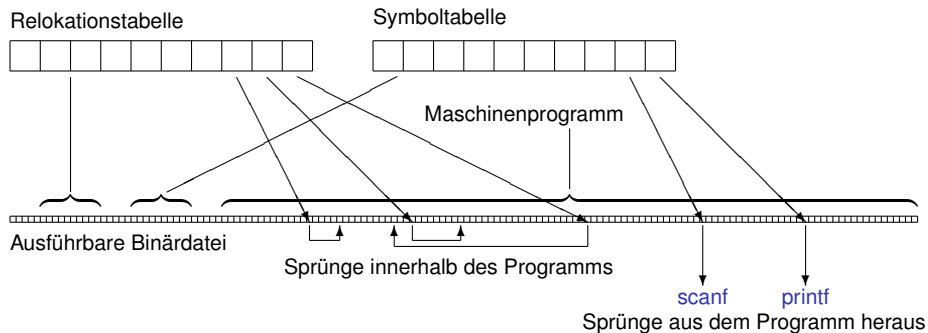
## 5.2 Dateiformate

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

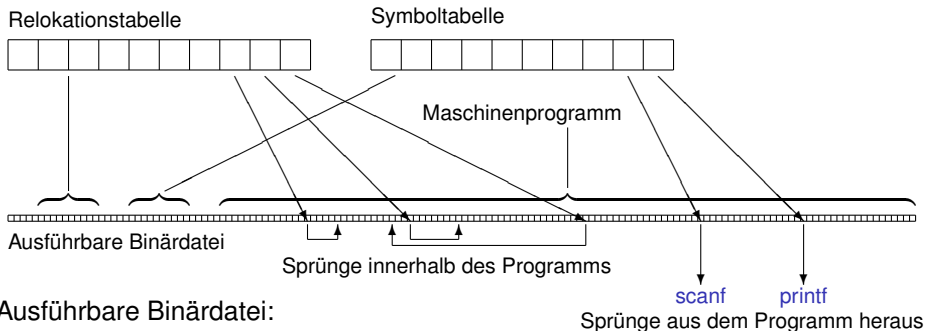
Sprünge anpassen

- Relokation: Relokationstabelle
- Linken: Symboltabelle

## 5.2 Dateiformate



## 5.2 Dateiformate



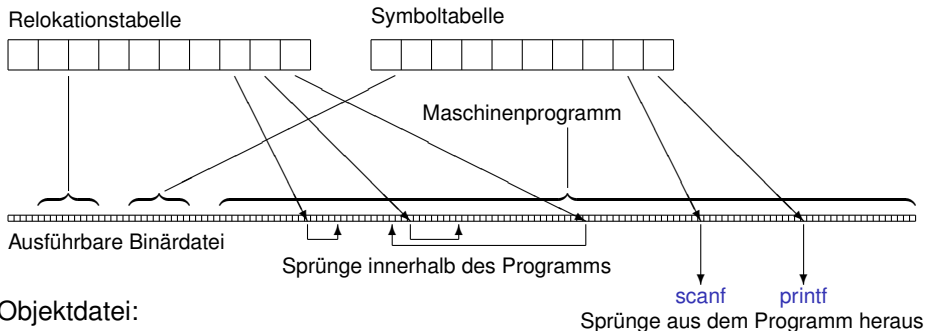
Ausführbare Binärdatei:

Relokationstabelle,  
Symboltabelle für dynamischen Linker

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

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

## 5.2 Dateiformate



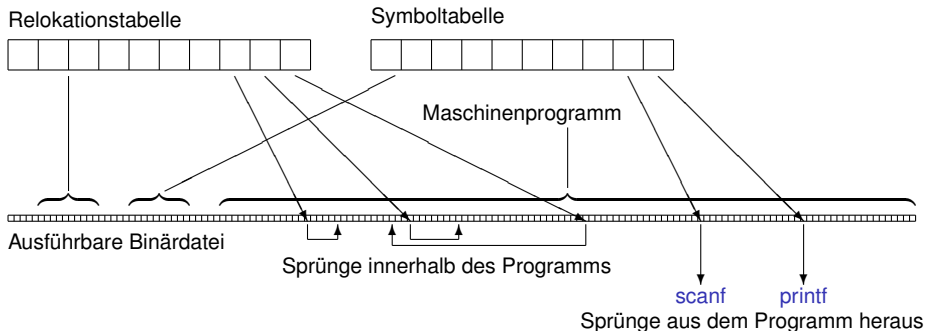
Objektdatei:

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

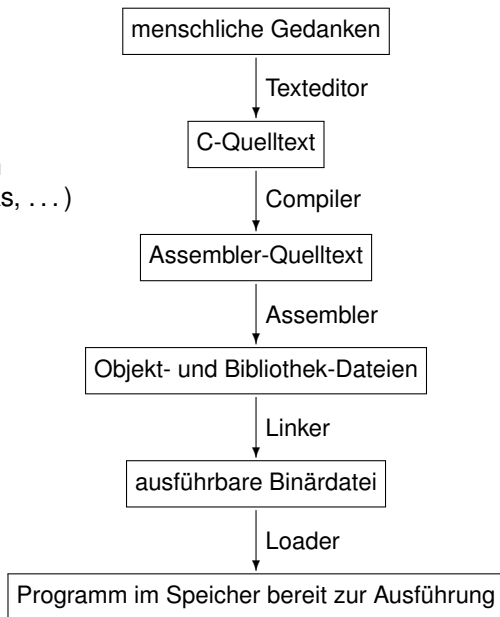
Dynamische Bibliotheken: .so, .dll



## 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

## 6 Pipelining

# 6 Pipelining

## 6.1 Konzept

- Aufgabe in Teilaufgaben zerlegen
- Teilaufgaben parallel ausführen

# 6 Pipelining

## 6.1 Konzept

- Aufgabe in Teilaufgaben zerlegen
- Teilaufgaben parallel ausführen




Beispiel: Wäsche waschen

# 6 Pipelining

## 6.1 Konzept

- Aufgabe in Teilaufgaben zerlegen
- Teilaufgaben parallel ausführen

Beispiel: Wäsche waschen




- Teilaufgaben:  ,  , 

# 6 Pipelining

## 6.1 Konzept

- Aufgabe in Teilaufgaben zerlegen
- Teilaufgaben parallel ausführen

Beispiel: Wäsche waschen

- Teilaufgaben:  ,  , 
- müssen nacheinander ausgeführt werden

# 6 Pipelining

## 6.1 Konzept

- Aufgabe in Teilaufgaben zerlegen
- Teilaufgaben parallel ausführen

Beispiel: Wäsche waschen

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






# 6 Pipelining

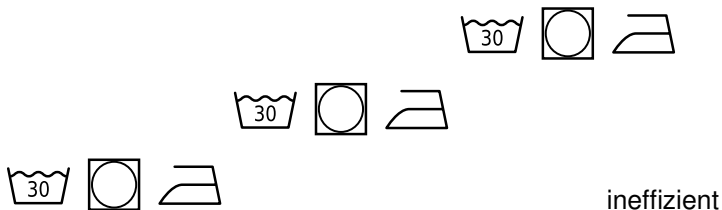
## 6.1 Konzept

- Aufgabe in Teilaufgaben zerlegen
- Teilaufgaben parallel ausführen

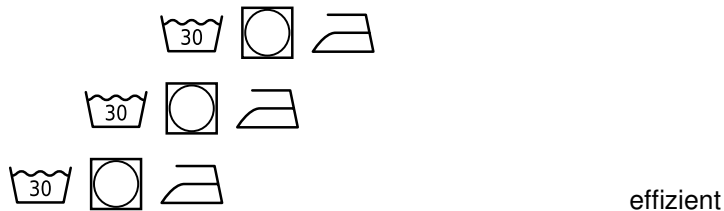
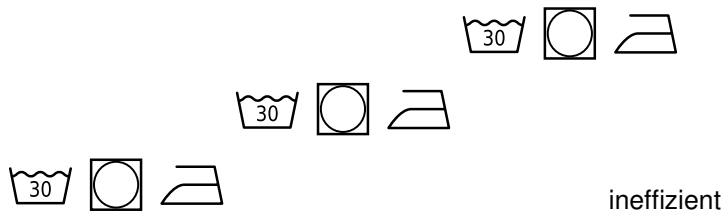
Beispiel: Wäsche waschen

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

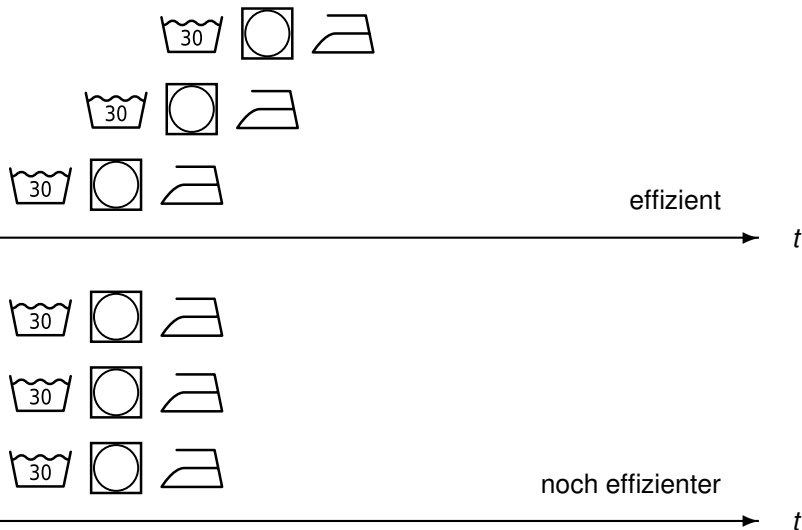
### 3 Ladungen Wäsche



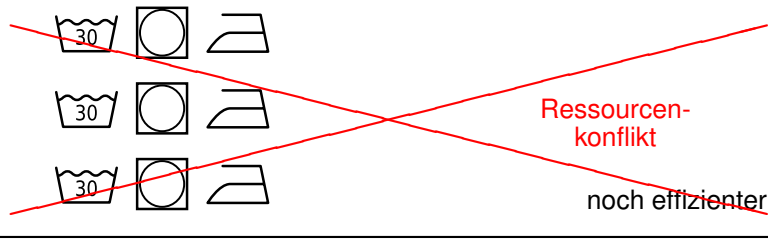
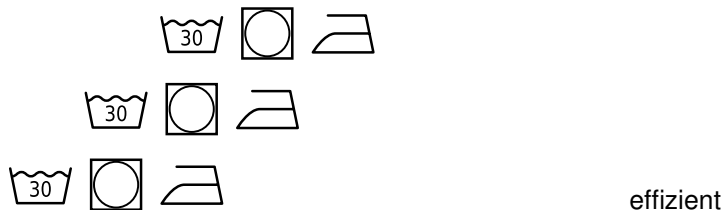
## 3 Ladungen Wäsche



### 3 Ladungen Wäsche



### 3 Ladungen Wäsche



### 3 Ladungen Wäsche



Ressourcen-  
konflikt

noch effizienter

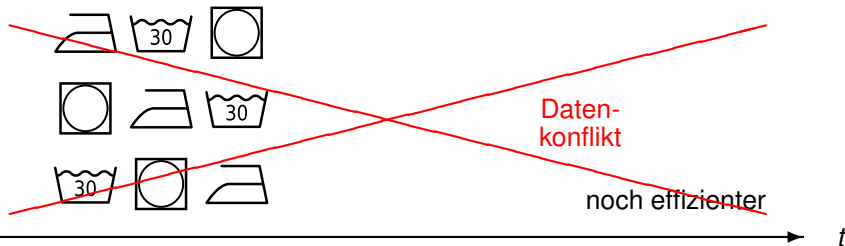
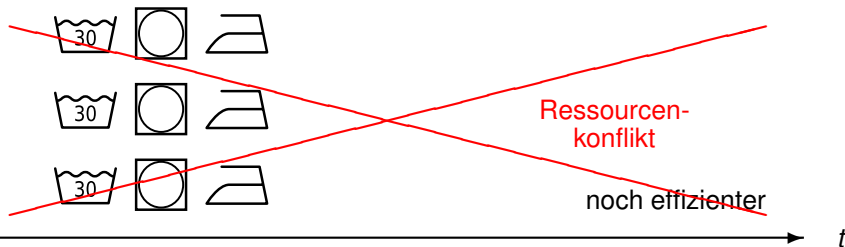
$t$



noch effizienter

$t$

### 3 Ladungen Wäsche



## 6.2 Arithmetik-Pipelines

„Register-FIFO“



## 6.2 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

## 6.2 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  $a_1 \cdot b_1$

## 6.2 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  $a_1 \cdot b_1$



## 6.2 Arithmetik-Pipelines

„Register-FIFO“

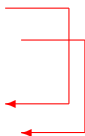
Pseudo-Code: Berechnung von

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

mit einer Pipeline der Länge 3

push  $a_1 \cdot b_1$

push  $a_2 \cdot b_2$



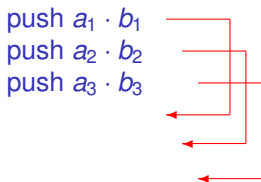
## 6.2 Arithmetik-Pipelines

„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



## 6.2 Arithmetik-Pipelines

„Register-FIFO“

Pseudo-Code: Berechnung von

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

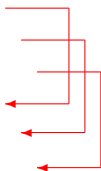
mit einer Pipeline der Länge 3

push  $a_1 \cdot b_1$

push  $a_2 \cdot b_2$

push  $a_3 \cdot b_3$

$s_1 = \text{pop}$



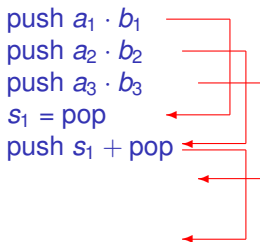
## 6.2 Arithmetik-Pipelines

„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



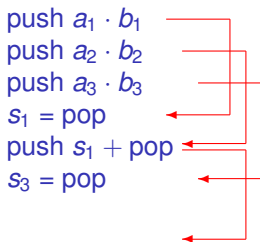
## 6.2 Arithmetik-Pipelines

„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





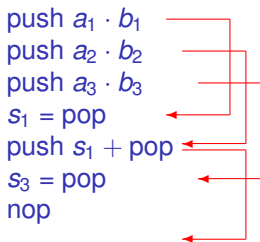
## 6.2 Arithmetik-Pipelines

„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



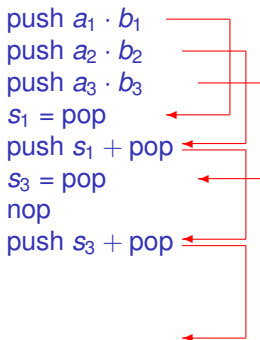
## 6.2 Arithmetik-Pipelines

„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



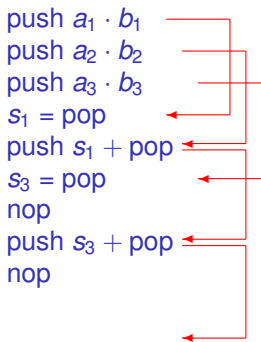
## 6.2 Arithmetik-Pipelines

„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



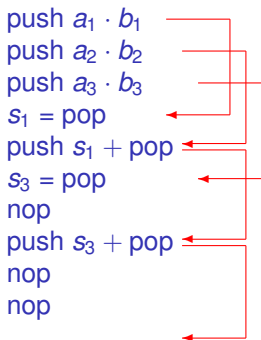
## 6.2 Arithmetik-Pipelines

„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



## 6.2 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

```
graph LR
    subgraph Pipeline
        direction TB
        S1[ ]
        S2[ ]
        S3[ ]
    end
    I1[push a1 · b1] --> S1
    I2[push a2 · b2] --> S1
    I3[push a3 · b3] --> S1
    I4[s1 = pop] --> S1
    I5[push s1 + pop] --> S1
    I6[s3 = pop] --> S1
    I7[nop] --> S1
    I8[push s3 + pop] --> S1
    I9[nop] --> S1
    I10[nop] --> S1
    I11[S = pop] --> S1
    I1 --> S2
    I2 --> S2
    I3 --> S2
    I4 --> S2
    I5 --> S2
    I6 --> S2
    I7 --> S2
    I8 --> S2
    I9 --> S2
    I10 --> S2
    I11 --> S2
    I1 --> S3
    I2 --> S3
    I3 --> S3
    I4 --> S3
    I5 --> S3
    I6 --> S3
    I7 --> S3
    I8 --> S3
    I9 --> S3
    I10 --> S3
    I11 --> S3
```

push  $a_1 \cdot b_1$   
push  $a_2 \cdot b_2$   
push  $a_3 \cdot b_3$   
 $s_1 = \text{pop}$   
push  $s_1 + \text{pop}$   
 $s_3 = \text{pop}$   
nop  
push  $s_3 + \text{pop}$   
nop  
nop  
 $S = \text{pop}$

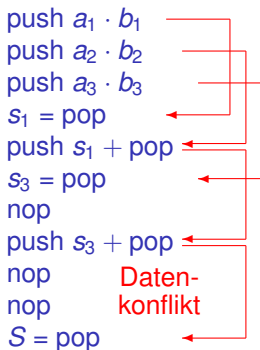
## 6.2 Arithmetik-Pipelines

„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



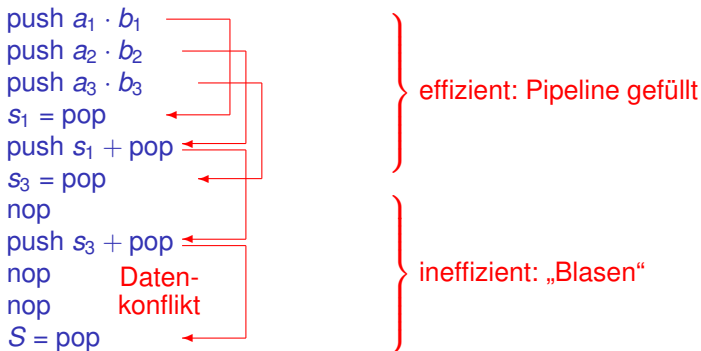
## 6.2 Arithmetik-Pipelines

„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



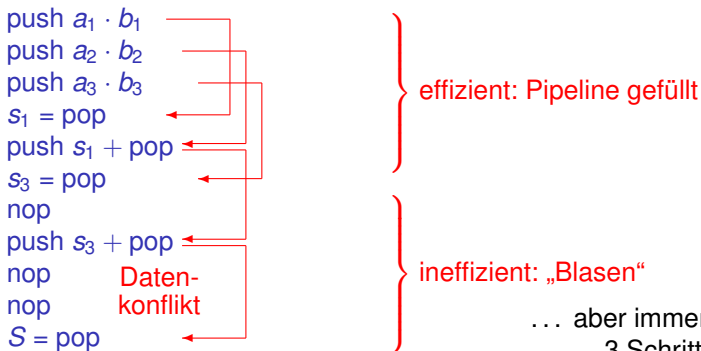
## 6.2 Arithmetik-Pipelines

„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





## 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
```

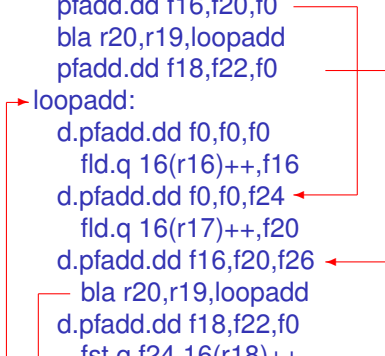
```
fld.q 16(r16)++,f16
fld.q 16(r17)++,f20
pfadd.dd f16,f20,f0
bla r20,r19,loopadd
pfadd.dd f18,f22,f0
loopadd:
    d.pfadd.dd f0,f0,f0
    fld.q 16(r16)++,f16
    d.pfadd.dd f0,f0,f24
    fld.q 16(r17)++,f20
    d.pfadd.dd f16,f20,f26
    bla r20,r19,loopadd
    d.pfadd.dd f18,f22,f0
    fst.q f24,16(r18)++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

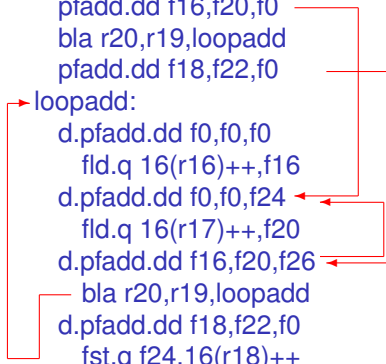
```
fld.q 16(r16)++,f16
fld.q 16(r17)++,f20
pfadd.dd f16,f20,f0
bla r20,r19,loopadd
pfadd.dd f18,f22,f0
loopadd:
    d.pfadd.dd f0,f0,f0
    fld.q 16(r16)++,f16
    d.pfadd.dd f0,f0,f24
    fld.q 16(r17)++,f20
    d.pfadd.dd f16,f20,f26
    bla r20,r19,loopadd
    d.pfadd.dd f18,f22,f0
    fst.q f24,16(r18)++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

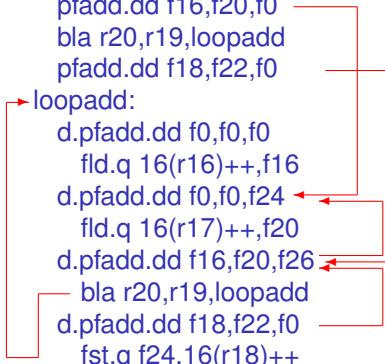
```
fld.q 16(r16)++,f16
fld.q 16(r17)++,f20
pfadd.dd f16,f20,f0
bla r20,r19,loopadd
pfadd.dd f18,f22,f0
loopadd:
    d.pfadd.dd f0,f0,f0
    fld.q 16(r16)++,f16
    d.pfadd.dd f0,f0,f24
    fld.q 16(r17)++,f20
    d.pfadd.dd f16,f20,f26
    bla r20,r19,loopadd
    d.pfadd.dd f18,f22,f0
    fst.q f24,16(r18)++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

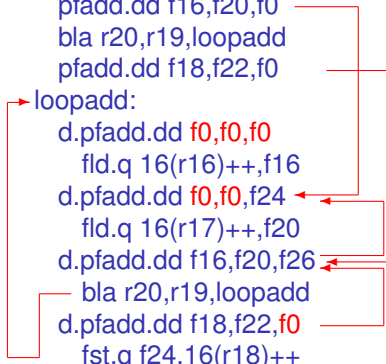
```
fld.q 16(r16)++,f16
fld.q 16(r17)++,f20
pfadd.dd f16,f20,f0
bla r20,r19,loopadd
pfadd.dd f18,f22,f0
loopadd:
    d.pfadd.dd f0,f0,f0
    fld.q 16(r16)++,f16
    d.pfadd.dd f0,f0,f24
    fld.q 16(r17)++,f20
    d.pfadd.dd f16,f20,f26
    bla r20,r19,loopadd
    d.pfadd.dd f18,f22,f0
    fst.q f24,16(r18)++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



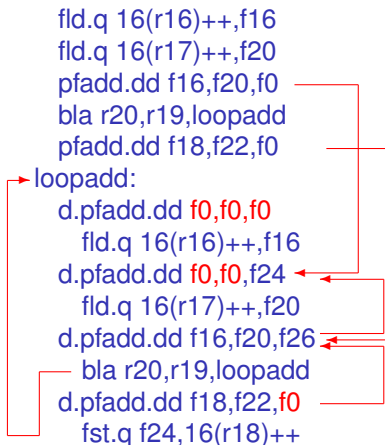
6mal f0 = 2 Blasen



## 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)++
```



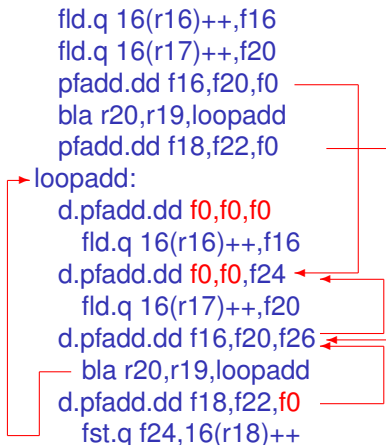
6mal f0 = 2 Blasen

Immerhin: 2 Additionen in 4 Taktzyklen

## 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

Dies ist ein *einfaches* Beispiel.

## 6.3 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
```

## 6.3 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

## 6.3 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

## 6.3 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

## 6.3 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
```

## 6.3 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
```



## 6.3 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
```

## 6.3 Instruktions-Pipelines

Ein Prozessor benötigt Zeit, um einen Befehl zu verstehen.

→ Während Befehlsausführung nächste Befehle vorauslesen

.L3:



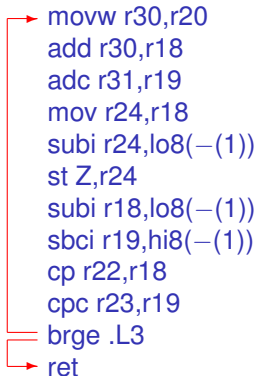
```
movw r30,r20
add r30,r18
adc r31,r19
mov r24,r18
subi r24,lo8(-(1))
st Z,r24
subi r18,lo8(-(1))
sbci r19,hi8(-(1))
cp r22,r18
cpc r23,r19
brge .L3
ret
```

## 6.3 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
```

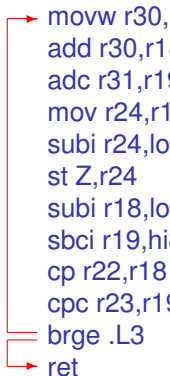
The diagram illustrates a loop structure. A red arrow points from the 'ret' instruction back to the 'movw r30,r20' instruction, indicating a branch back to the start of the loop body.

## 6.3 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
```

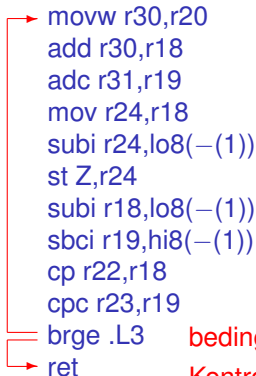
bedingter Sprung: Welche Befehle vorauslesen?

## 6.3 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
```

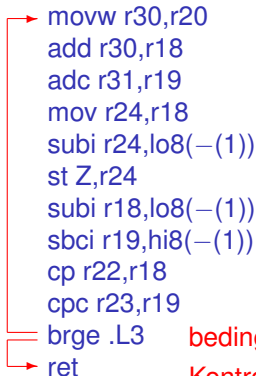
bedingter Sprung: Welche Befehle vorauslesen?  
Kontrollflußkonflikt

## 6.3 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

## 6.3 Instruktions-Pipelines

### Zweigvorhersage – Branch Prediction

## 6.3 Instruktions-Pipelines

### Zweigvorhersage – Branch Prediction

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



## 6.3 Instruktions-Pipelines

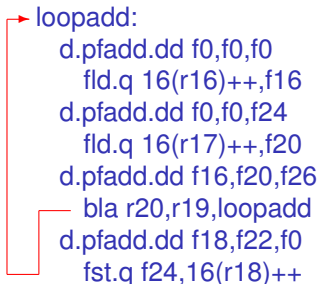
### Zweigvorhersage – Branch Prediction

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

## 6.3 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

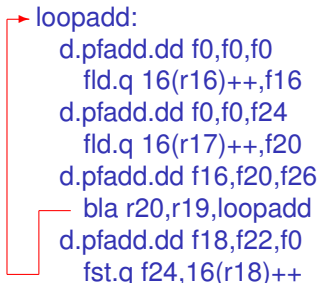


```
loopadd:
    d.pfadd.dd f0,f0,f0
    fld.q 16(r16)++,f16
    d.pfadd.dd f0,f0,f24
    fld.q 16(r17)++,f20
    d.pfadd.dd f16,f20,f26
    bla r20,r19,loopadd
    d.pfadd.dd f18,f22,f0
    fst.q f24,16(r18)++
```

## 6.3 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



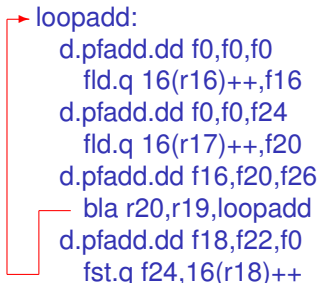
```
loopadd:
    d.pfadd.dd f0,f0,f0
    fld.q 16(r16)++,f16
    d.pfadd.dd f0,f0,f24
    fld.q 16(r17)++,f20
    d.pfadd.dd f16,f20,f26
    bla r20,r19,loopadd
    d.pfadd.dd f18,f22,f0
    fst.q f24,16(r18)++
```

- Branch History Table: Sprünge merken

## 6.3 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
- ...

# 6 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