

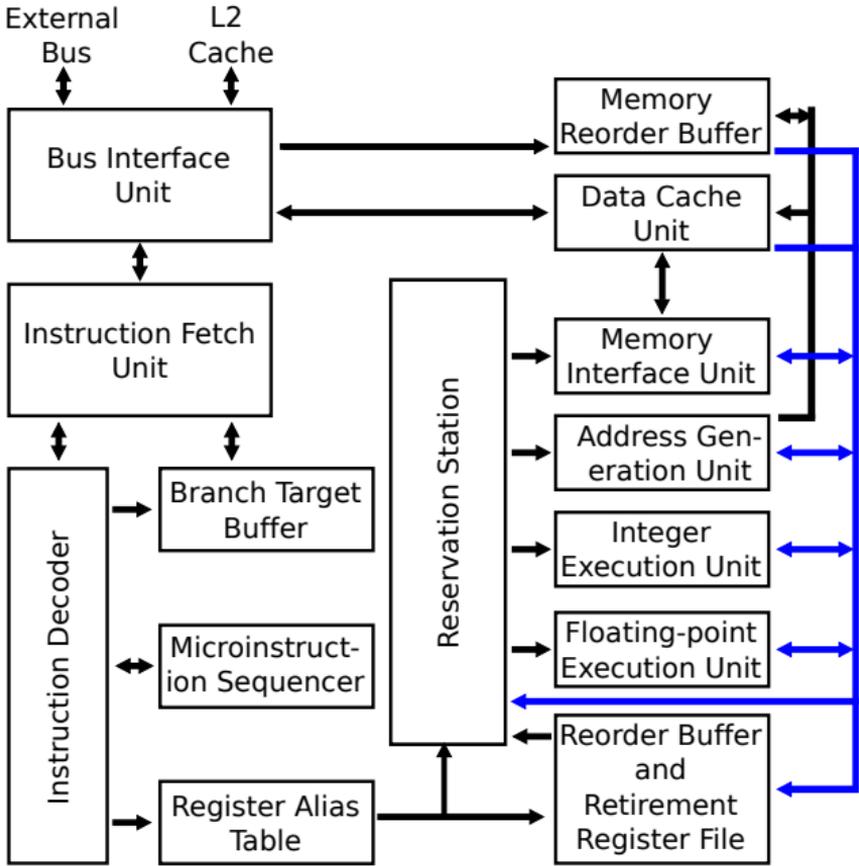
# Die Mikroarchitektur moderner Prozessoren: Ein Crashkurs

Robert Clausecker

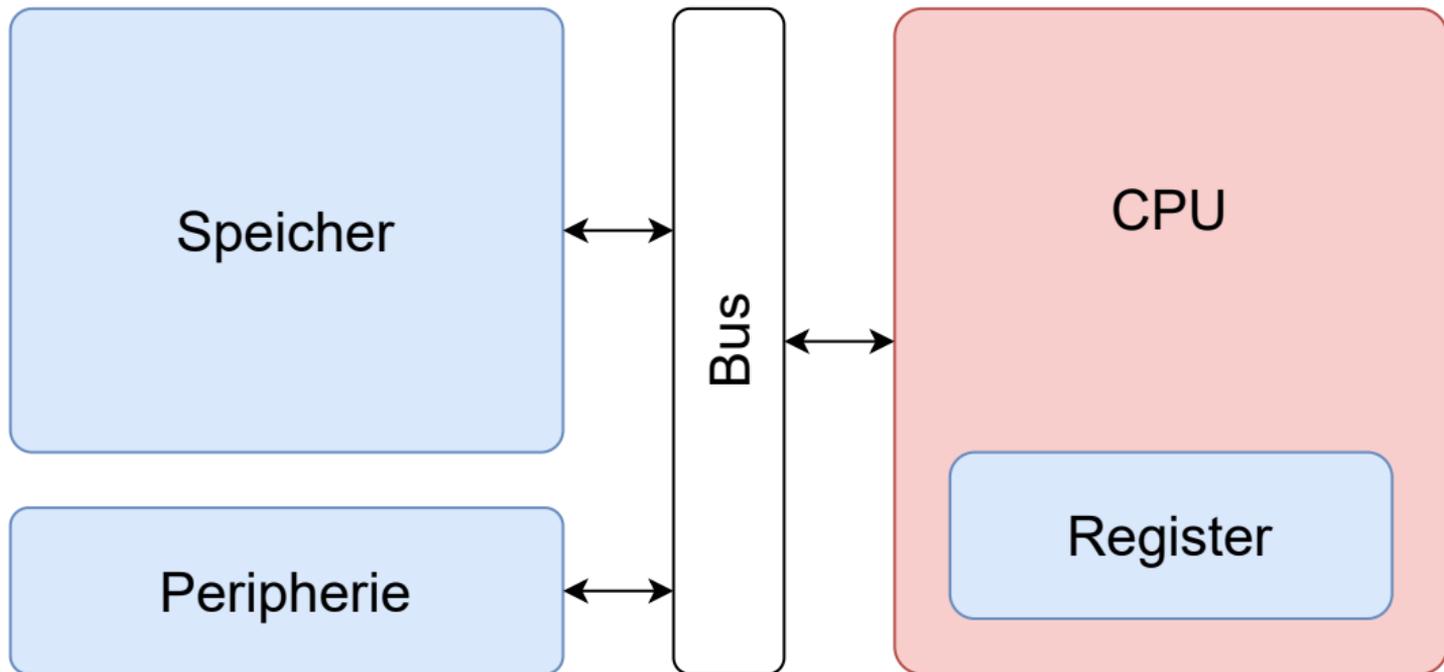
29. Dezember 2022

# Was ist eigentlich ein Computer?

# Was ist eigentlich ein Computer?



# Was ist eigentlich ein Computer?



# Was ist eigentlich ein Computer?

- Speicher** Enthält Programm und Daten, an jeder Adresse ein Byte
- Peripherie** Wird wie Speicher behandelt
  - CPU** Führt den Maschinencode aus
  - Register** Spezielle Speicherstellen direkt in der CPU
  - Bus** Ermöglicht den Systemteilen, miteinander zu reden

# Runter zum Assembler

Was ist ein Maschinenbefehl?

# Runter zum Assembler

Was ist ein Maschinenbefehl?

```
48 03 08    addq (%rax), %rcx
```

48 03 08 Befehlscode in hexadezimal

# Runter zum Assembler

Was ist ein Maschinenbefehl?

```
48 03 08    addq (%rax), %rcx
```

48 03 08 Befehlscode in hexadezimal

add Mnemonic (Name) des Befehls

# Runter zum Assembler

Was ist ein Maschinenbefehl?

```
48 03 08    addq (%rax), %rcx
```

48 03 08 Befehlscode in hexadezimal

`add` Mnemonic (Name) des Befehls

`mov a, b` Daten bewegen ( $b = a$ )

`add a, b` Addition ( $b = a + b$ )

`cmp a, b` Vergleich zwischen  $a$  und  $b$

`jmp label` Sprung zu *label*

...viele, viele mehr

# Runter zum Assembler

Was ist ein Maschinenbefehl?

```
48 03 08    addq (%rax), %rcx
```

48 03 08 Befehlscode in hexadezimal

**add** Mnemonic (Name) des Befehls

**mov a, b** Daten bewegen ( $b = a$ )

**add a, b** Addition ( $b = a + b$ )

**cmp a, b** Vergleich zwischen  $a$  und  $b$

**jmp label** Sprung zu  $label$

...viele, viele mehr

**q** Datengröße: **b** – 8-Bit Byte, **w** – 16-Bit Word,  
**l** – 32-Bit Longword, **q** – 64-Bit Quadword

# Runter zum Assembler

Was ist ein Maschinenbefehl?

```
48 03 08    addq (%rax), %rcx
```

48 03 08 Befehlscode in hexadezimal

`add` Mnemonic (Name) des Befehls

`mov a, b` Daten bewegen ( $b = a$ )

`add a, b` Addition ( $b = a + b$ )

`cmp a, b` Vergleich zwischen  $a$  und  $b$

`jmp label` Sprung zu  $label$

...viele, viele mehr

`q` Datengröße: `b` – 8-Bit Byte, `w` – 16-Bit Word,  
`l` – 32-Bit Longword, `q` – 64-Bit Quadword

`%rax, %rcx` Register

# Runter zum Assembler

Was ist ein Maschinenbefehl?

```
48 03 08    addq (%rax), %rcx
```

48 03 08 Befehlscode in hexadezimal

`add` Mnemonic (Name) des Befehls

`mov a, b` Daten bewegen ( $b = a$ )

`add a, b` Addition ( $b = a + b$ )

`cmp a, b` Vergleich zwischen  $a$  und  $b$

`jmp label` Sprung zu  $label$

...viele, viele mehr

`q` Datengröße: `b` – 8-Bit Byte, `w` – 16-Bit Word,  
`l` – 32-Bit Longword, `q` – 64-Bit Quadword

`%rax, %rcx` Register

(...) Indirekter Operand (Speicherzugriff)

`$...` Immediat-Operand (Konstante)

# Runter zum Assembler

Was ist ein Maschinenbefehl?

```
48 03 08    addq (%rax), %rcx
```

48 03 08 Befehlscode in hexadezimal

`add` Mnemonic (Name) des Befehls

`mov a, b` Daten bewegen ( $b = a$ )

`add a, b` Addition ( $b = a + b$ )

`cmp a, b` Vergleich zwischen  $a$  und  $b$

`jmp label` Sprung zu  $label$

...viele, viele mehr

`q` Datengröße: `b` – 8-Bit Byte, `w` – 16-Bit Word,  
`l` – 32-Bit Longword, `q` – 64-Bit Quadword

`%rax, %rcx` Register

(...) Indirekter Operand (Speicherzugriff)

`$...` Immediat-Operand (Konstante)

# Sequentielle Ausführung

%rax: 0x00401234

...

%rcx: 0xc0fec0fe

%rip: 0x0020dead

# Sequentielle Ausführung

%rax: 0x00401234

%rcx: 0xc0fec0fe

...

%rip: 0x0020dead

## 1. Laden des Befehls aus dem Arbeitsspeicher (*instruction fetch*)

Adresse des Befehls steht im *Befehlszeiger* %rip

0x0020dead: 03 08

Befehlszähler wird weitergezählt:  $\%rip \leftarrow \%rip + 2$

# Sequentielle Ausführung

%rax: 0x00401234

%rcx: 0xc0fec0fe

...

%rip: 0x0020dead

## 1. Laden des Befehls aus dem Arbeitsspeicher (*instruction fetch*)

Adresse des Befehls steht im *Befehlszeiger* %rip

0x0020dead: 03 08

Befehlszähler wird weitergezählt:  $\%rip \leftarrow \%rip + 2$

## 2. Dekodierung des Befehles (*instruction decode*)

0x0020dead: 03 08 addl (%rax), %ecx

# Sequentielle Ausführung

%rax: 0x00401234

%rcx: 0xc0fec0fe

...

%rip: 0x0020dead

1. Laden des Befehls aus dem Arbeitsspeicher (*instruction fetch*)

Adresse des Befehls steht im *Befehlszeiger* %rip

0x0020dead: 03 08

Befehlszähler wird weitergezählt:  $\%rip \leftarrow \%rip + 2$

2. Dekodierung des Befehles (*instruction decode*)

0x0020dead: 03 08 addl (%rax), %ecx

3. Laden der indirekten Operanden (*memory*)

(%rax) hält 0x00002342

# Sequentielle Ausführung

%rax: 0x00401234

%rcx: 0xc0fec0fe

...

%rip: 0x0020dead

1. Laden des Befehls aus dem Arbeitsspeicher (*instruction fetch*)

Adresse des Befehls steht im *Befehlszeiger* %rip

0x0020dead: 03 08

Befehlszähler wird weitergezählt:  $\%rip \leftarrow \%rip + 2$

2. Dekodierung des Befehles (*instruction decode*)

0x0020dead: 03 08 addl (%rax), %ecx

3. Laden der indirekten Operanden (*memory*)

(%rax) hält 0x00002342

4. Durchführung der Berechnung (*execute*)

$0x00002342 + 0xc0fec0fe = 0xc0fee440$

# Sequentielle Ausführung

%rax: 0x00401234

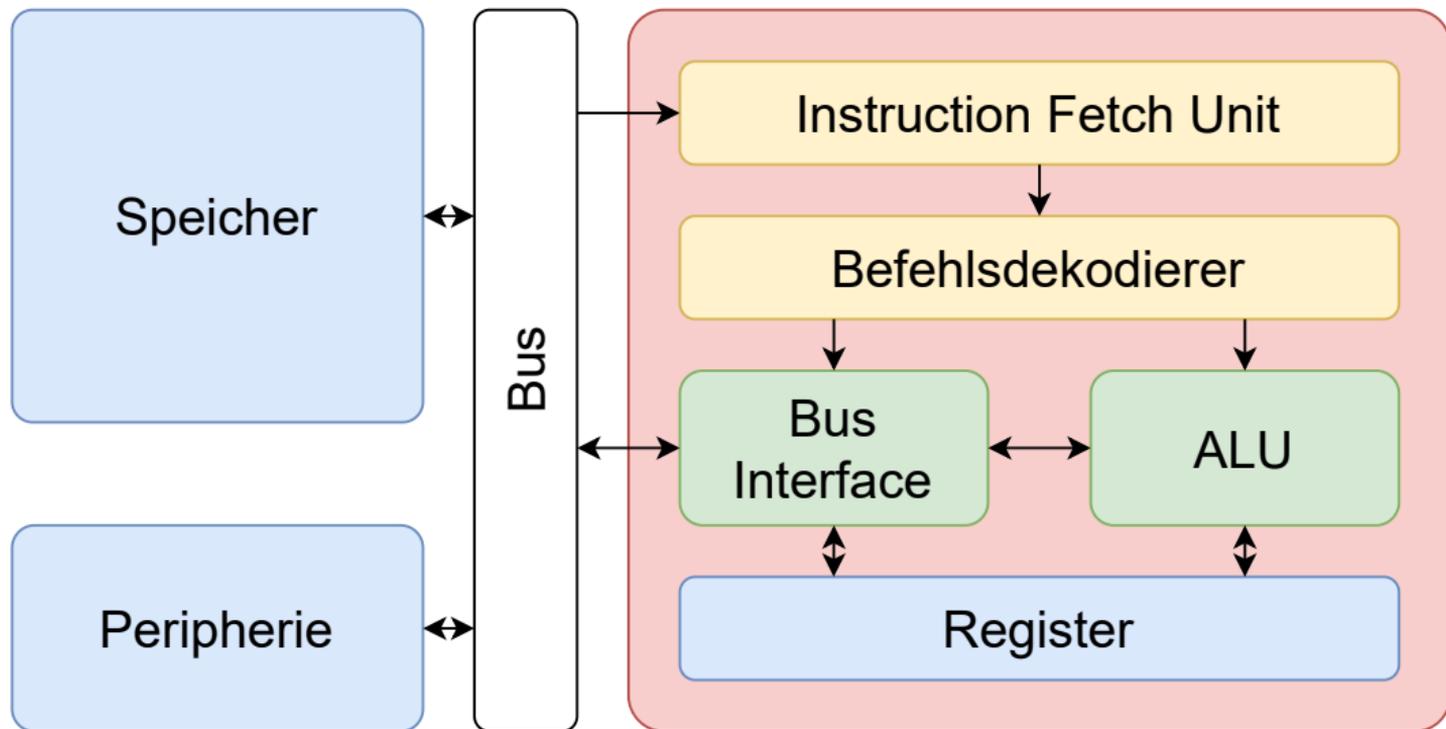
%rcx: 0xc0fec0fe

...

%rip: 0x0020dead

1. Laden des Befehls aus dem Arbeitsspeicher (*instruction fetch*)  
Adresse des Befehls steht im *Befehlszeiger* %rip  
0x0020dead: 03 08  
Befehlszähler wird weitergezählt:  $\%rip \leftarrow \%rip + 2$
2. Dekodierung des Befehles (*instruction decode*)  
0x0020dead: 03 08 addl (%rax), %ecx
3. Laden der indirekten Operanden (*memory*)  
(%rax) hält 0x00002342
4. Durchführung der Berechnung (*execute*)  
 $0x00002342 + 0xc0fec0fe = 0xc0fee440$
5. Rückschreiben der Ergebnisse (*writeback*)  
 $\%ecx \leftarrow 0xc0fee440$

## Verfeinertes Blockschaltbild



# Out-of-Order-Ausführung

5 oder mehr Takte pro Befehl ist zu langsam. Was können wir tun?

# Out-of-Order-Ausführung

5 oder mehr Takte pro Befehl ist zu langsam. Was können wir tun?

- ▶ Pipelining: Jeden Takt wird ein neuer Befehl nachgeschoben
  - ▶ Instr. Fetch → Decode → Execute → Memory → Writeback
  - ▶ Jede Pipelinestufe behandelt je Takt einen Befehl
  - ▶ Erlaubt Durchsatz von bis zu einem Befehl pro Takt (1 IPC)
  - ▶ *Hazards*, wenn Daten nicht rechtzeitig bereit sind.

# Out-of-Order-Ausführung

5 oder mehr Takte pro Befehl ist zu langsam. Was können wir tun?

- ▶ Pipelining: Jeden Takt wird ein neuer Befehl nachgeschoben
  - ▶ Instr. Fetch → Decode → Execute → Memory → Writeback
  - ▶ Jede Pipelinestufe behandelt je Takt einen Befehl
  - ▶ Erlaubt Durchsatz von bis zu einem Befehl pro Takt (1 IPC)
  - ▶ *Hazards*, wenn Daten nicht rechtzeitig bereit sind.
- ▶ Darüber hinaus: mehrere Befehle pro Takt ausführen.
  - ▶ Mehrere Befehle pro Takt laden und dekodieren
  - ▶ Mehrere ALUs für gleichzeitige Ausführung mehrerer Befehle
  - ▶ Die CPU muss rausfinden, ab wann ein Befehl ausführbar ist
  - ▶ Schnelle Befehle müssen nicht auf langsame Befehle warten
  - ▶ Realistisch etwa 4 Befehle pro Takt

# Out-of-Order-Ausführung: Komponenten

**IFU** Lädt Befehlsstrom schnell genug nach

**Dekodierer** Dekodiert mehrere Befehle parallel

- ▶ Auf x86: bis zu 4 Befehle durch schwarze Magie
- ▶ Komplizierte Befehle werden in Mikrobefehle gespalten

**Umbenener** Logischen Registern (wenige) werden physische Register (viele) zugewiesen

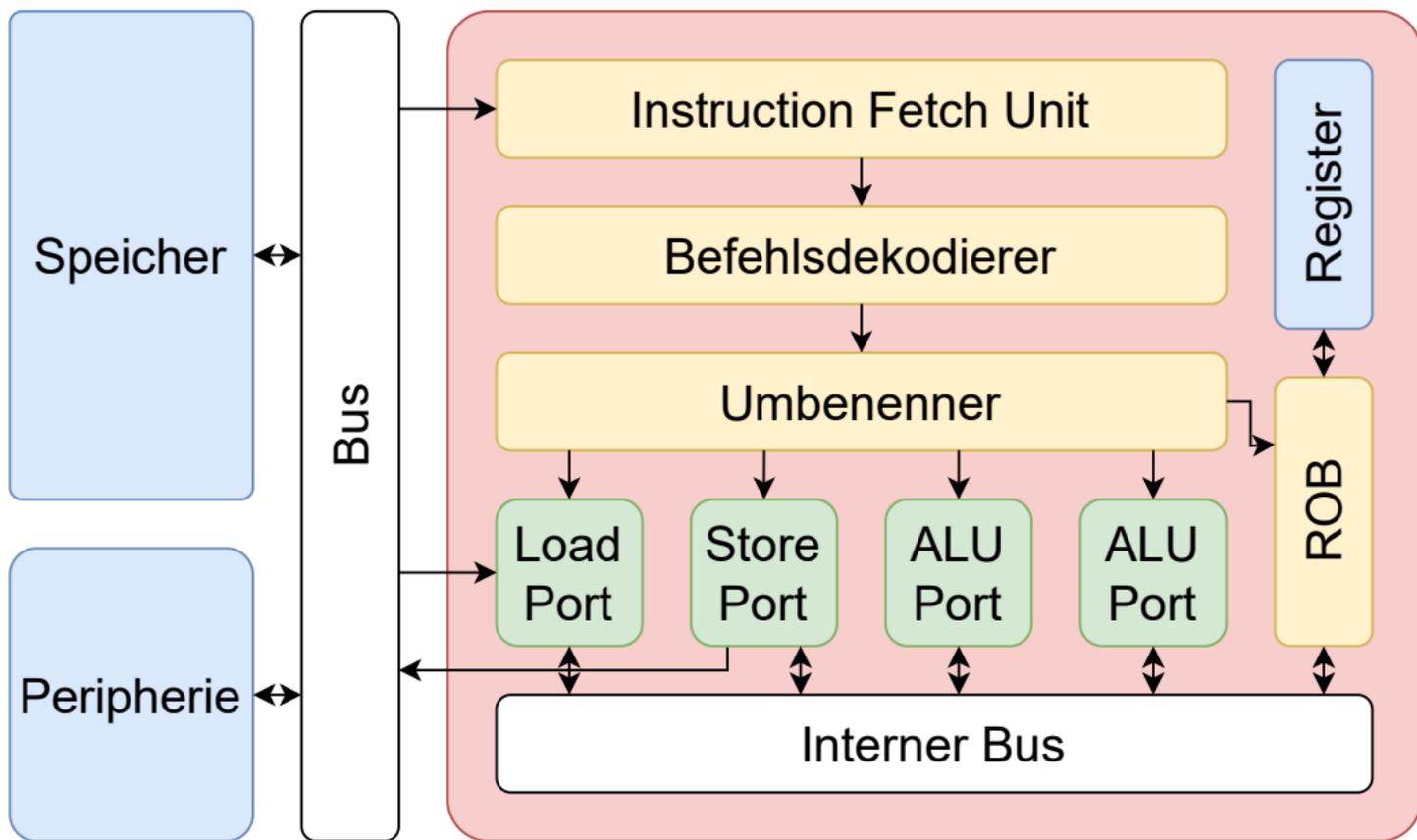
- ▶ Umbenennung löst Hazards auf
- ▶ Befehle werden in Port-Warteschlangen sortiert

**Ports** Führen Befehle aus

- ▶ verschiedene Befehlstypen je nach Port
- ▶ Befehle warten in *Warteschlangen* auf Ausführung
- ▶ Ausführung möglich, sobald Operanden bereit sind
- ▶ Ältestmöglicher Befehl wird ausgeführt

**Reorder-Buffer** Sammelt fertige Befehle ein und stellt Reihenfolge her

# Out-of-Order-Ausführung: Blockschaltbild



## Out-of-Order-Ausführung: Beispiel

```
movl    $42, %eax
movq    $bar, %rcx

imull   $23, (foo), %ebx

addl    %ebx, (%rcx)

subl    %eax, %ebx
```

## Out-of-Order-Ausführung: Beispiel

```
movl    $42, %eax
movq    $bar, %rcx
movl    (foo), %tmp
imull   $23, %tmp, %ebx
movl    (%rcx), %tmp
addl    %ebx, %tmp
movl    %tmp, (%rcx)
subl    %eax, %ebx
```

## Out-of-Order-Ausführung: Beispiel

```
movl    $42,    %0
movq    $bar,   %1
movl    (foo),  %2
imull   $23,    %2,    %3
movl    ( %1),  %4
addl    %3,    %4,    %5
movl    %5,    ( %2)
subl    %0,    %3,    %6
```

## Out-of-Order-Ausführung: Beispiel

Load Port	Store Port	ALU Port I	ALU Port II
<code>movl (foo), %2</code> <code>movl (%1), %4</code>	<code>movl %5, (%2)</code>	<code>movl \$42, %0</code> <code>imull \$23, %2, %3</code> <code>subl %0, %3, %6</code>	<code>movq \$bar, %1</code> <code>addl %3, %4, %5</code>

Schon ausgerechnet:

## Out-of-Order-Ausführung: Beispiel

Load Port	Store Port	ALU Port I	ALU Port II
<code>movl (%1), %4</code>	<code>movl %5, (%2)</code>	<code>imull \$23, %2, %3</code> <code>subl %0, %3, %6</code>	<code>addl %3, %4, %5</code>

Schon ausgerechnet: %0, %1, %2

## Out-of-Order-Ausführung: Beispiel

Load Port	Store Port	ALU Port I	ALU Port II
	<code>movl %5, (%2)</code>		
		<code>subl %0, %3, %6</code>	<code>addl %3, %4, %5</code>

Schon ausgerechnet: %0, %1, %2, %3, %4

## Out-of-Order-Ausführung: Beispiel

Load Port

Store Port

ALU Port I

ALU Port II

---

`movl %5, (%2)`

---

Schon ausgerechnet: %0, %1, %2, %3, %4, %5, %6

# Out-of-Order-Ausführung: Beispiel

**Load Port**

**Store Port**

**ALU Port I**

**ALU Port II**

---

Schon ausgerechnet: %0, %1, %2, %3, %4, %5, %6.

## Tipp: IPC erhöhen

Kann nur sequentiell ausgeführt werden:

```
int array[n], i, sum = 0;
```

```
for (i = 0; i < n; i++)  
    sum += array[i];
```

## Tipp: IPC erhöhen

Kann nur sequentiell ausgeführt werden:

```
int array[n], i, sum = 0;

for (i = 0; i < n; i++)
    sum += array[i];
```

Zwei Iterationen können parallel ausgeführt werden:

```
int array[n], i, sum, sum1 = 0, sum2 = 0;

for (i = 0; i < n; i += 2) {
    sum1 += array[i];
    sum2 += array[i+1];
}

sum = sum1 + sum2;
```

## Tipp: Sprungvorhersage

Bedingte und indirekte Sprünge sind schwierig:

- ▶ Sprungziel ist erst nach Ausführung bekannt
- ▶ Instruction Fetch müsste suspendiert werden, bis es soweit ist

## Tipp: Sprungvorhersage

Bedingte und indirekte Sprünge sind schwierig:

- ▶ Sprungziel ist erst nach Ausführung bekannt
- ▶ Instruction Fetch müsste suspendiert werden, bis es soweit ist
- ▶ Stattdessen: Sprungziel wird vorhergesehen (geraten)
- ▶ Richtig geraten: Sprung ist gratis
- ▶ Falsch geraten: Pipelineflush, teuer, langsam
- ▶ Moderne Sprungvorhersage ist ziemlich schlau

## Tipp: Sprungvorhersage

Bedingte und indirekte Sprünge sind schwierig:

- ▶ Sprungziel ist erst nach Ausführung bekannt
- ▶ Instruction Fetch müsste suspendiert werden, bis es soweit ist
- ▶ Stattdessen: Sprungziel wird vorhergesehen (geraten)
- ▶ Richtig geraten: Sprung ist gratis
- ▶ Falsch geraten: Pipelineflush, teuer, langsam
- ▶ Moderne Sprungvorhersage ist ziemlich schlau

### Performance-Tipp

Bedingte Sprünge sind toll, wenn sie einfach vorhersehbar sind. Sonst kann es sich lohnen, sie zu eliminieren. Im Zweifel messen.

## Tipp: Caching

Hauptspeicher ist langsam.

## Tipp: Caching

Hauptspeicher ist langsam.

Zugriff auf Register:	1 Takt
Zugriff auf L1 Cache:	4 Takte
Zugriff auf L2 Cache:	16 Takte
Zugriff auf Hauptspeicher:	150 Takte

## Tipp: Caching

Hauptspeicher ist langsam.

Zugriff auf Register:	1 Takt
Zugriff auf L1 Cache:	4 Takte
Zugriff auf L2 Cache:	16 Takte
Zugriff auf Hauptspeicher:	150 Takte

### Performance-Tipp

Versuche, zusammenhängende Daten nebeneinander im Speicher abzulegen. Große Arrays am besten sequentiell lesen und schreiben.