

Rechnertechnik

Prof. Dr. rer. nat. Peter Gerwinski

21. Juni 2021

Rechnertechnik

- 1 Einführung**
- 2 Vom Schaltkreis zum Computer**
- 3 Architekturmerkmale von Prozessoren**
- 4 Der CPU-Stack**
- 5 Anwender-Software**
- 6 Pipelining**
 - 6.1 Konzept
 - 6.2 Arithmetik-Pipelines
 - 6.3 Instruktions-Pipelines
- 7 Bus-Systeme**
 - 7.1 Was sind Bus-Systeme?
 - 7.2 Chipsatz
 - 7.3 RS-232
 - 7.4 I²C (TWI)
 - 7.5 SPI

...

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“

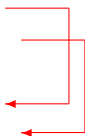
Pseudo-Code: Berechnung von

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

mit einer Pipeline der Länge 3

push $a_1 \cdot b_1$

push $a_2 \cdot b_2$



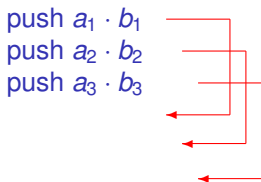
6.2 Arithmetik-Pipelines

„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

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

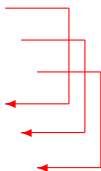
mit einer Pipeline der Länge 3

push $a_1 \cdot b_1$

push $a_2 \cdot b_2$

push $a_3 \cdot b_3$

$s_1 = \text{pop}$



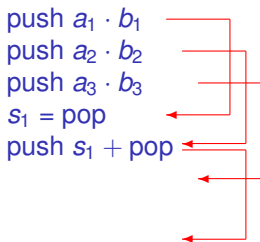
6.2 Arithmetik-Pipelines

„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



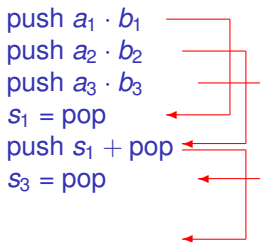
6.2 Arithmetik-Pipelines

„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



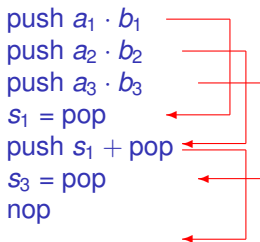
6.2 Arithmetik-Pipelines

„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



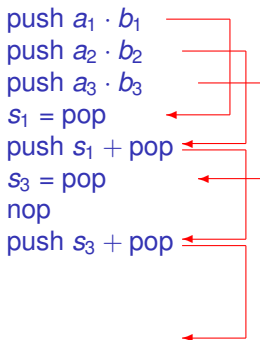
6.2 Arithmetik-Pipelines

„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



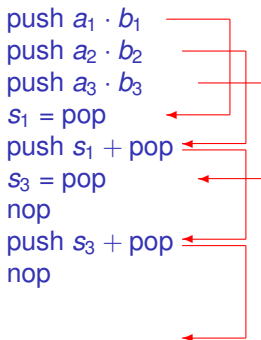
6.2 Arithmetik-Pipelines

„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



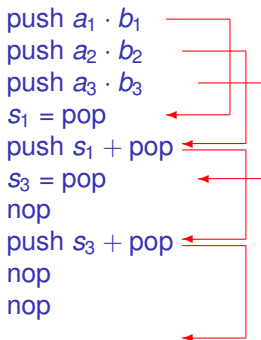
6.2 Arithmetik-Pipelines

„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
    I1[push a1 · b1] --> I2[push a2 · b2]
    I2 --> I3[push a3 · b3]
    I3 --> I4[s1 = pop]
    I4 --> I5[push s1 + pop]
    I5 --> I6[s3 = pop]
    I6 --> I7[nop]
    I7 --> I8[push s3 + pop]
    I8 --> I9[nop]
    I9 --> I10[nop]
    I10 --> I11[S = pop]
```

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
 $S = \text{pop}$

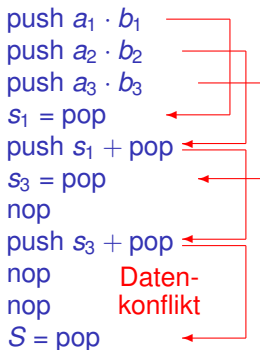
6.2 Arithmetik-Pipelines

„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



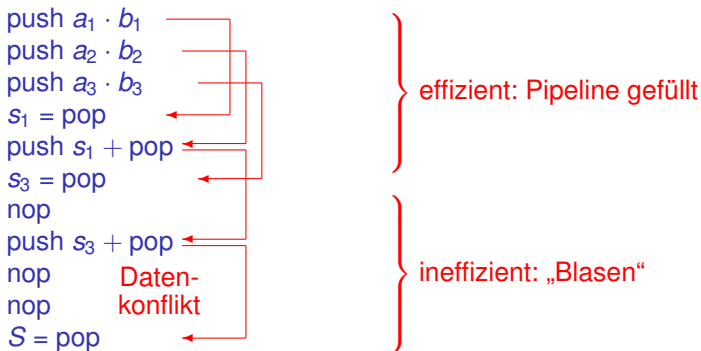
6.2 Arithmetik-Pipelines

„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



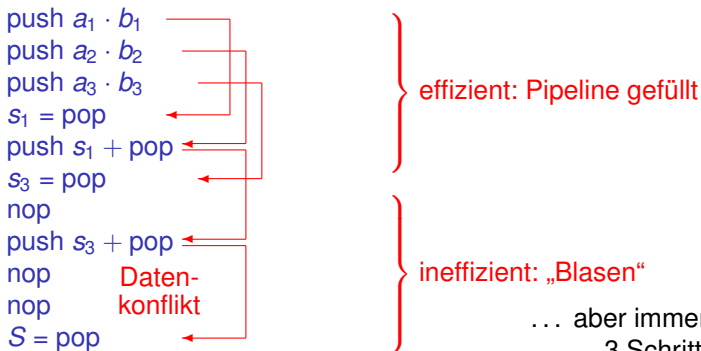
6.2 Arithmetik-Pipelines

„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



... 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
```

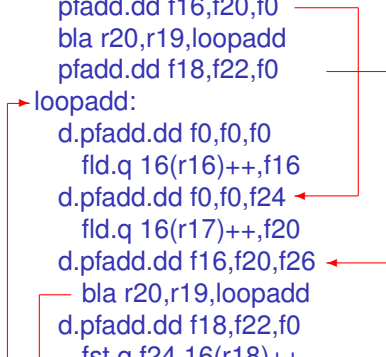
```
fld.q 16(r16)++,f16
fld.q 16(r17)++,f20
pfadd.dd f16,f20,f0
bla r20,r19,loopadd
pfadd.dd f18,f22,f0
loopadd:
    d.pfadd.dd f0,f0,f0
    fld.q 16(r16)++,f16
    d.pfadd.dd f0,f0,f24
    fld.q 16(r17)++,f20
    d.pfadd.dd f16,f20,f26
    bla r20,r19,loopadd
    d.pfadd.dd f18,f22,f0
    fst.q f24,16(r18)++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

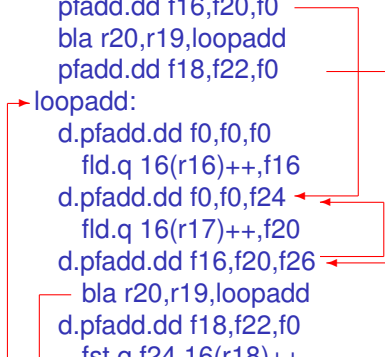
```
fld.q 16(r16)++,f16
fld.q 16(r17)++,f20
pfadd.dd f16,f20,f0
bla r20,r19,loopadd
pfadd.dd f18,f22,f0
loopadd:
    d.pfadd.dd f0,f0,f0
    fld.q 16(r16)++,f16
    d.pfadd.dd f0,f0,f24
    fld.q 16(r17)++,f20
    d.pfadd.dd f16,f20,f26
    bla r20,r19,loopadd
    d.pfadd.dd f18,f22,f0
    fst.q f24,16(r18)++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

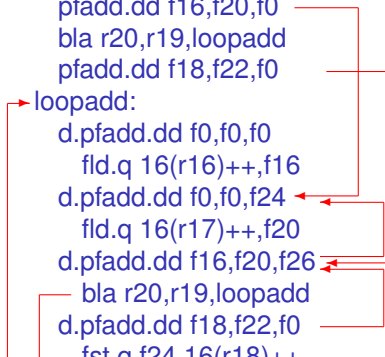
```
fld.q 16(r16)++,f16
fld.q 16(r17)++,f20
pfadd.dd f16,f20,f0
bla r20,r19,loopadd
pfadd.dd f18,f22,f0
loopadd:
    d.pfadd.dd f0,f0,f0
    fld.q 16(r16)++,f16
    d.pfadd.dd f0,f0,f24
    fld.q 16(r17)++,f20
    d.pfadd.dd f16,f20,f26
    bla r20,r19,loopadd
    d.pfadd.dd f18,f22,f0
    fst.q f24,16(r18)++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

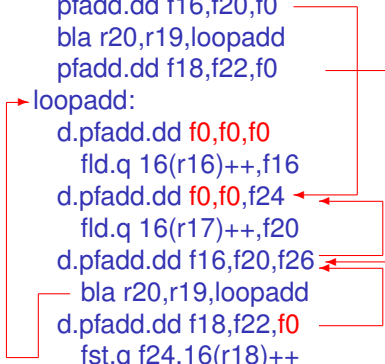
```
fld.q 16(r16)++,f16
fld.q 16(r17)++,f20
pfadd.dd f16,f20,f0
bla r20,r19,loopadd
pfadd.dd f18,f22,f0
loopadd:
    d.pfadd.dd f0,f0,f0
    fld.q 16(r16)++,f16
    d.pfadd.dd f0,f0,f24
    fld.q 16(r17)++,f20
    d.pfadd.dd f16,f20,f26
    bla r20,r19,loopadd
    d.pfadd.dd f18,f22,f0
    fst.q f24,16(r18)++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

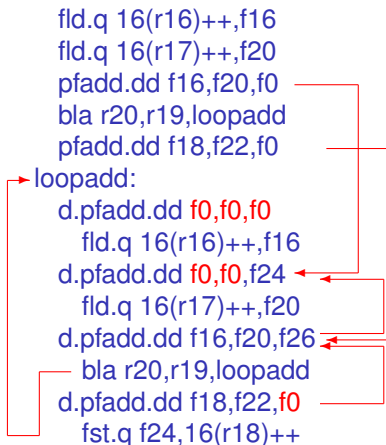


6mal f0 = 2 Blasen

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)++
```



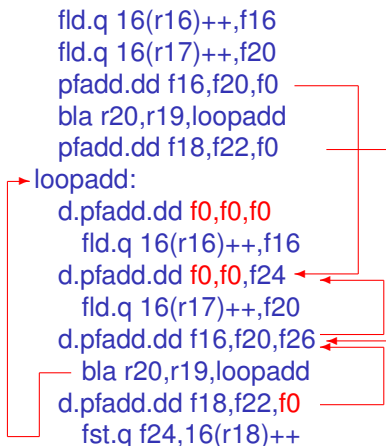
6mal f0 = 2 Blasen

Immerhin: 2 Additionen in 4 Taktzyklen

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:



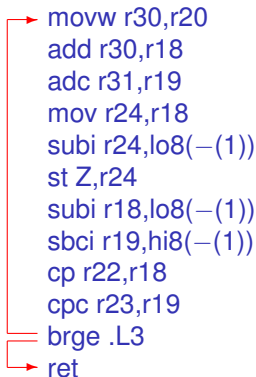
```
→ movw r30,r20
   add r30,r18
   adc r31,r19
   mov r24,r18
   subi r24,lo8(-(1))
   st Z,r24
   subi r18,lo8(-(1))
   sbci r19,hi8(-(1))
   cp r22,r18
   cpc r23,r19
   brge .L3
   ret
```

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
```

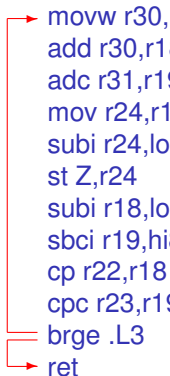
The diagram illustrates a loop structure. A red arrow points from the start of the code block to the first instruction, `movw r30,r20`. Another red arrow points from the `brge .L3` instruction back to the start of the code block, indicating a loop. A third red arrow points from the `ret` instruction to the right, indicating the end of the function.

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
```

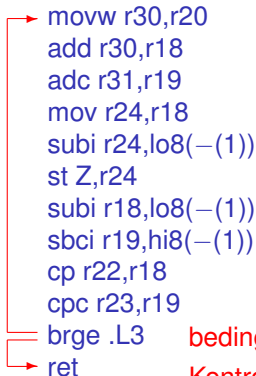
bedingter Sprung: Welche Befehle vorauslesen?

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

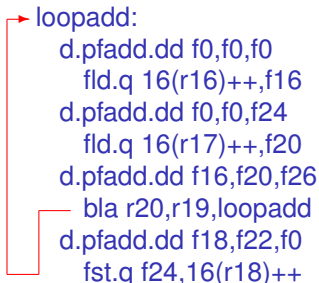
Zweigvorhersage – Branch Prediction

- 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

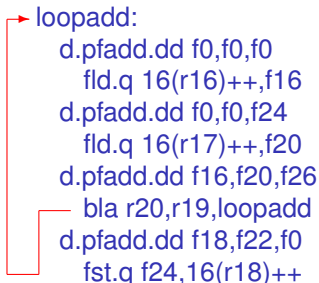


```
loopadd:
    d.pfadd.dd f0,f0,f0
    fld.q 16(r16)++,f16
    d.pfadd.dd f0,f0,f24
    fld.q 16(r17)++,f20
    d.pfadd.dd f16,f20,f26
    bla r20,r19,loopadd
    d.pfadd.dd f18,f22,f0
    fst.q f24,16(r18)++
```

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



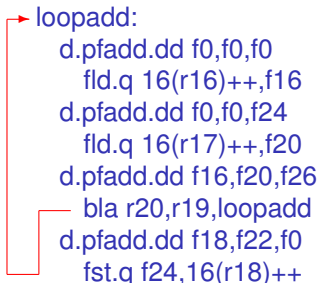
```
loopadd:
    d.pfadd.dd f0,f0,f0
    fld.q 16(r16)++,f16
    d.pfadd.dd f0,f0,f24
    fld.q 16(r17)++,f20
    d.pfadd.dd f16,f20,f26
    bla r20,r19,loopadd
    d.pfadd.dd f18,f22,f0
    fst.q f24,16(r18)++
```

- 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

7 Bus-Systeme

7.1 Was sind Bus-Systeme?

Ein Bus ist ein System zur Datenübertragung zwischen mehreren Teilnehmern über einen gemeinsamen Übertragungsweg.

Findet eine Datenübertragung zwischen zwei Teilnehmern statt, so müssen die übrigen Teilnehmer schweigen, da sie sich sonst gegenseitig stören würden. Umgangssprachlich werden mitunter – oft aus historischen Gründen – auch Datenübertragungssysteme als „Bus“ bezeichnet, die technisch eigentlich eine andere Topologie besitzen.

[https://de.wikipedia.org/wiki/Bus_\(Datenverarbeitung\)](https://de.wikipedia.org/wiki/Bus_(Datenverarbeitung))

Beispiele:

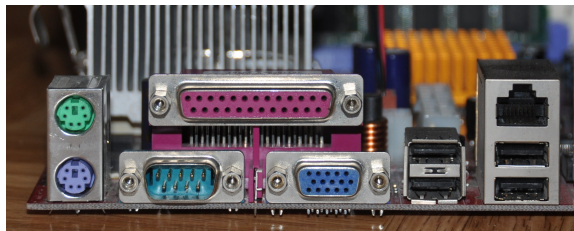
- Computer kommuniziert mit Peripherie
- Computer kommunizieren (direkt) miteinander
- Prozessor kommuniziert mit externem Speicher
- Teile eines Prozessors kommunizieren miteinander

7 Bus-Systeme

7.1 Was sind Bus-Systeme?

Standard-Computer:

- Einsteckkarten: PCI (und Vorgänger)
- Festplatten: SATA (und Vorgänger)
- USB, FireWire, ...
- Ethernet, CAN-Bus, ...
- WLAN, BlueTooth, IR, ...
- PS/2, RS-232, IEEE 1284



7 Bus-Systeme

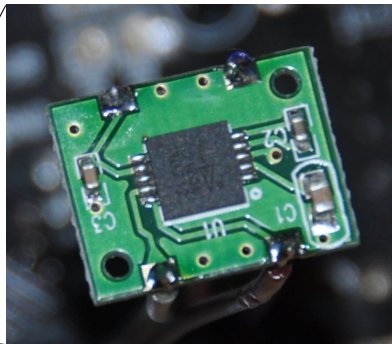
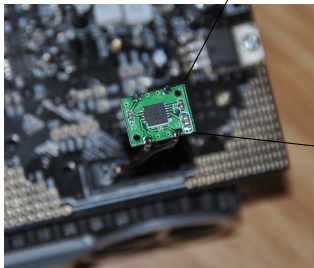
7.1 Was sind Bus-Systeme?

Standard-Computer:

- Einsteckkarten: PCI (und Vorgänger)
- Festplatten: SATA (und Vorgänger)
- USB, FireWire, ...
- Ethernet, CAN-Bus, ...
- WLAN, BlueTooth, IR, ...
- PS/2, RS-232, IEEE 1284

Minimal-Hardware:

- RS-232
- I²C (TWI)
- SPI



7 Bus-Systeme

7.1 Was sind Bus-Systeme?

<i>seriell</i>	jedes Bit einzeln übertragen	
<i>parallel</i>	mehrere Bits gleichzeitig	
<i>synchron</i>	Ableich über Steuerleitung: <i>Takt</i>	
<i>asynchron</i>	Ableich über Zeitvereinbarungen	
<i>Punkt-zu-Punkt</i>	genau zwei Teilnehmer	
<i>busfähig</i>	mehrere Teilnehmer, mit <i>Adressierung</i>	moderne Definition ←

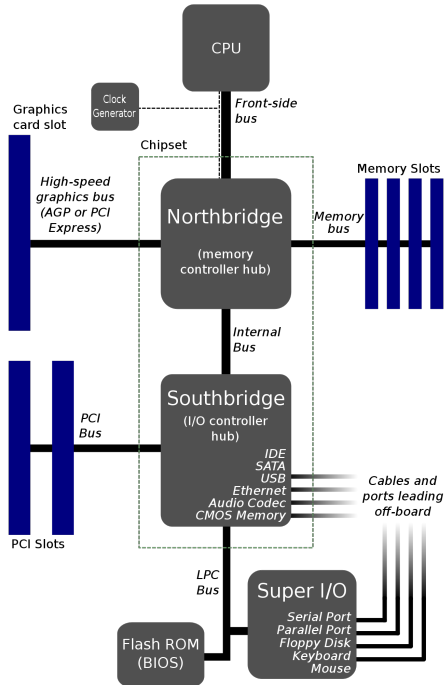
- I²C: seriell, synchron, mit Adressierung
- RS-232: seriell, asynchron, Punkt-zu-Punkt
- RS-485, USB, CAN: seriell, asynchron, mit Adressierung
- SPI: seriell, synchron, Punkt-zu-Punkt oder mit Adressierung
- PCI Express: seriell, asynchron, Punkt-zu-Punkt
- SATA: seriell, asynchron(?), Punkt-zu-Punkt
- PATA/ATAPI/IDE: parallel, asynchron(?), mit Adressierung

7 Bus-Systeme

7.2 Chipsatz

Kommunikation des Prozessors mit

- Arbeitsspeicher
- Peripherie

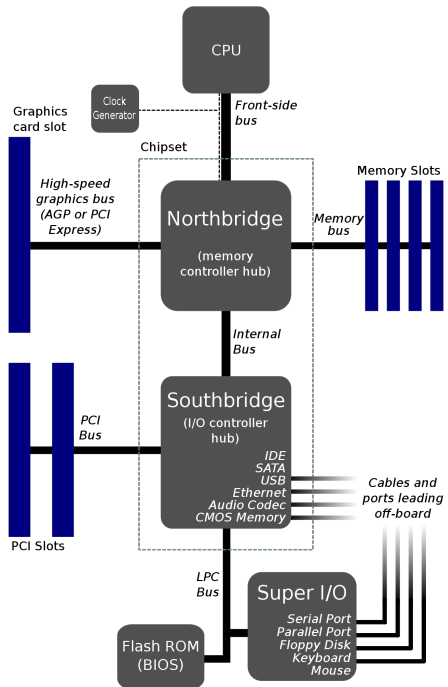


7 Bus-Systeme

7.2 Chipsatz

Kommunikation des Prozessors mit

- Arbeitsspeicher
- Hochgeschwindigkeits-Peripherie
- „normale“ Peripherie

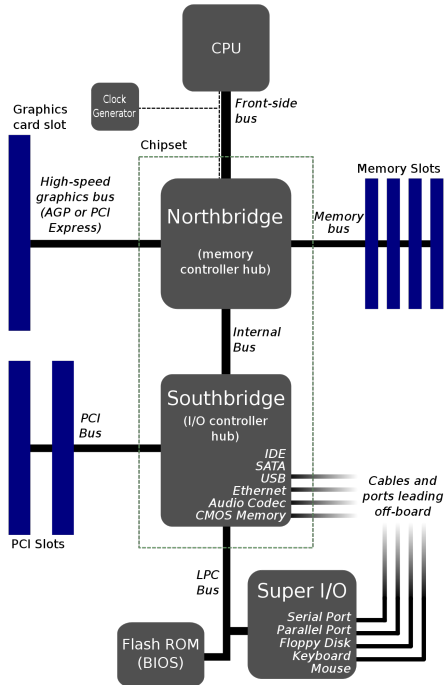


7 Bus-Systeme

7.2 Chipsatz

Kommunikation des Prozessors mit

- Arbeitsspeicher
- Hochgeschwindigkeits-Peripherie:
Northbridge
- „normale“ Peripherie:
Southbridge
- Northbridge + Southbridge
= *Chipsatz*

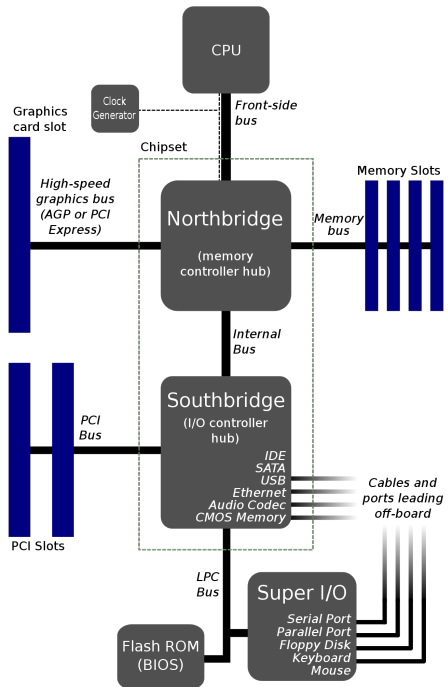


7 Bus-Systeme

7.2 Chipsatz

Kommunikation des Prozessors mit

- Arbeitsspeicher
- Hochgeschwindigkeits-Peripherie:
Northbridge
- „normale“ Peripherie:
Southbridge
- Northbridge + Southbridge
= *Chipsatz*
- Problem: unterschiedliche
Bus-Geschwindigkeiten

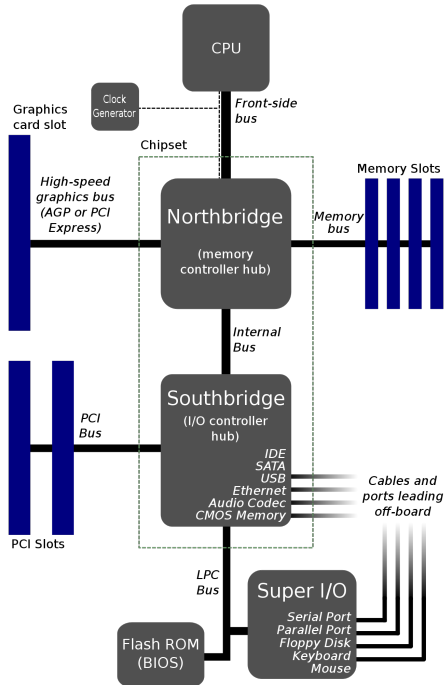


7 Bus-Systeme

7.2 Chipsatz

Kommunikation des Prozessors mit

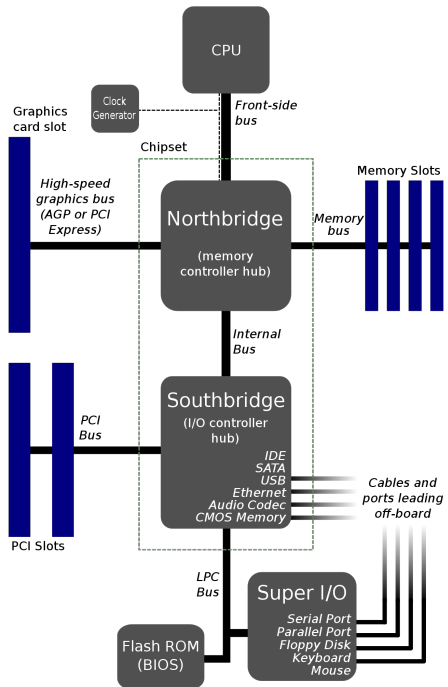
- Arbeitsspeicher
- Hochgeschwindigkeits-Peripherie: *Northbridge*
- „normale“ Peripherie: *Southbridge*
- Northbridge + Southbridge = *Chipsatz*
- Problem: unterschiedliche Bus-Geschwindigkeiten
- Northbridge: heute meistens in Prozessor integriert.
Damit besteht der Chipsatz nur noch aus der Southbridge.



7 Bus-Systeme

7.2 Chipsatz

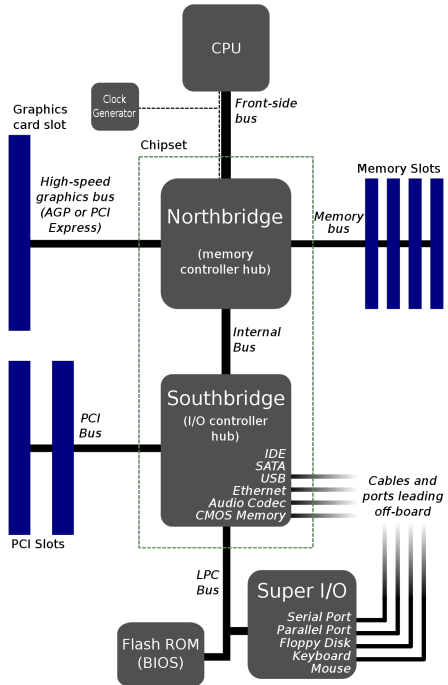
- Problem: Arbeitsspeicher nur über Bus erreichbar
→ langsam



7 Bus-Systeme

7.2 Chipsatz

- Problem: Arbeitsspeicher nur über Bus erreichbar
→ langsam
- Lösung: Cache
= Arbeitsspeicher innerhalb des Prozessors

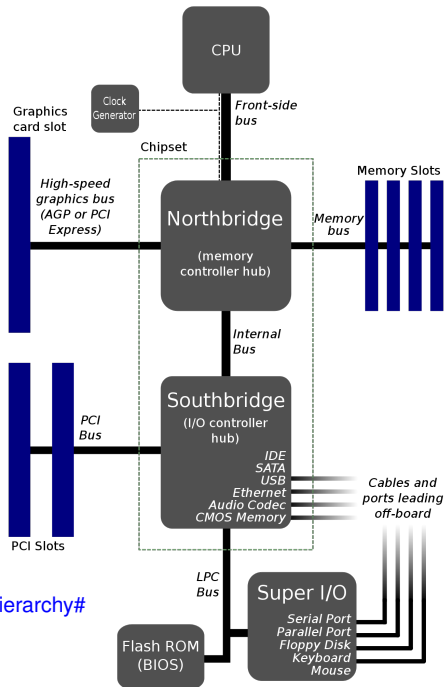


7 Bus-Systeme

7.2 Chipsatz

- Problem: Arbeitsspeicher nur über Bus erreichbar
→ langsam
- Lösung: Cache
= Arbeitsspeicher innerhalb des Prozessors
- mehrstufiger Cache
Beispiel: AMD Zen 2 (2019)
L1: 32 kB Daten
+ 32 kB Instruktionen pro Kern
L2: 512 kB pro Kern
L3: 16–256 MB pro 4 Kerne

Quelle: https://en.wikipedia.org/wiki/Cache_hierarchy#Recent_implementation_models



7 Bus-Systeme

7.3 RS-232

seriell

- *TX*: 1 Leitung für Daten
- *RX*: ggf. 1 Leitung für Daten in der anderen Richtung
- *GND*: gemeinsame *Masse*
- evtl. zusätzliche Steuerleitungen

asynchron

- *keine* Taktleitung für Abgleich, wann Daten anliegen
- Stattdessen: Abgleich über Zeitvereinbarungen

→ Jeder Teilnehmer braucht eine eigene Zeitbasis.

Punkt-zu-Punkt

- nur 2 Teilnehmer vorgesehen

7.3 RS-232

Synchronisation

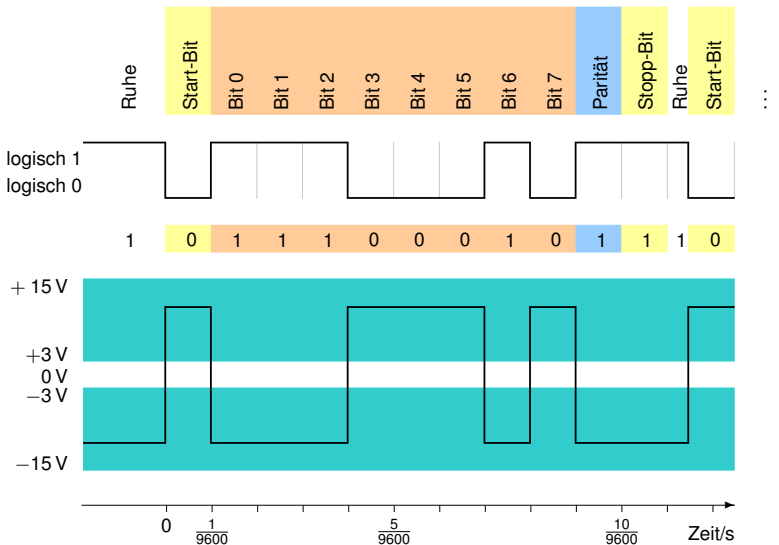
Daten

Check

9600 Baud, 8 Daten-Bits, ungerade Parität, 1 Stopp-Bit

Beispiel-Daten: ASCII „G“ = 71 = 0100 0111 binär

Übertragung der Daten von rechts (Bit 0) nach links (Bit 7)



7 Bus-Systeme

7.4 I²C (TWI)

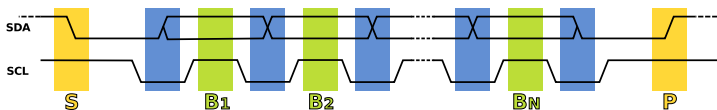
I²C = Inter-IC; TWI = Two-Wire-Interface

seriell

- *SDA*: 1 Leitung für Daten (in beiden Richtungen)
- *SCL*: Taktleitung (Clock)
- *GND*: gemeinsame Masse
- evtl. *VCC*: Stromversorgung für Peripheriegerät

synchron

- Abgleich über Taktleitung



busfähig

- *Master* initiiert Kommunikation und steuert Taktleitung
- erstes gesendetes Byte: *Adresse* des Teilnehmers
- 2 Adressen pro Teilnehmer: Lesen/Schreiben

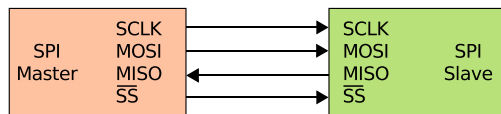
7 Bus-Systeme

7.5 SPI

Serial Peripheral Interface

seriell

- *MOSI*: Master Out, Slave In
- *MISO*: Master In, Slave Out
- *SCLK*: Taktleitung (Clock)
- \overline{SS} : Slave Select (invertiert)
- *GND*: gemeinsame Masse
- evtl. *VCC*: Stromversorgung für Peripheriegerät



synchron

- Abgleich über Taktleitung

busfähig

- *Master* initiiert Kommunikation und steuert Taktleitung
- *Slave* wird über *Slave Select* ausgewählt

7 Bus-Systeme

7.5 SPI

Serial Peripheral Interface

seriell

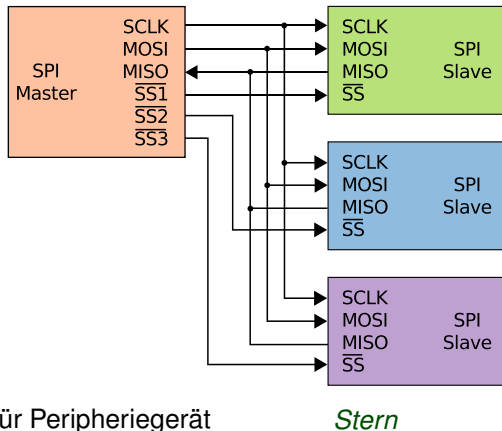
- *MOSI*: Master Out, Slave In
- *MISO*: Master In, Slave Out
- *SCLK*: Taktleitung (Clock)
- \overline{SS} : Slave Select (invertiert)
- *GND*: gemeinsame Masse
- evtl. *VCC*: Stromversorgung für Peripheriegerät

synchron

- Abgleich über Taktleitung

busfähig

- *Master* initiiert Kommunikation und steuert Taktleitung
- *Slave* wird über *Slave Select* ausgewählt



7 Bus-Systeme

7.5 SPI

Serial Peripheral Interface

seriell

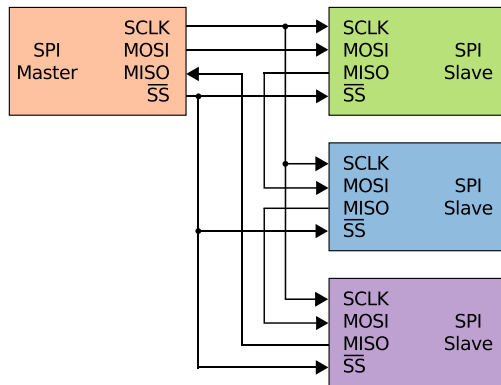
- *MOSI*: Master Out, Slave In
- *MISO*: Master In, Slave Out
- *SCLK*: Taktleitung (Clock)
- \overline{SS} : Slave Select (invertiert)
- *GND*: gemeinsame Masse
- evtl. *VCC*: Stromversorgung für Peripheriegerät

synchron

- Abgleich über Taktleitung

busfähig

- *Master* initiiert Kommunikation und steuert Taktleitung
- *Slave* wird über *Slave Select* ausgewählt



*Kaskade
Daisy Chain*

7 Bus-Systeme

7.5 SPI

Serial Peripheral Interface

seriell

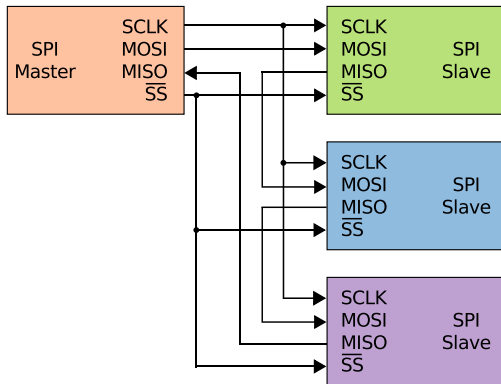
- *MOSI*: Master Out, Slave In
- *MISO*: Master In, Slave Out
- *SCLK*: Taktleitung (Clock)
- \overline{SS} : Slave Select (invertiert)
- *GND*: gemeinsame Masse
- evtl. *VCC*: Stromversorgung für Peripheriegerät

synchron

- Abgleich über Taktleitung

busfähig

- *Master* initiiert Kommunikation und steuert Taktleitung
- *Slave* wird über *Slave Select* ausgewählt



*Kaskade
Daisy Chain*

Slave gibt MOSI-Input um 1 Takt verzögert an MISO aus → Master setzt „im richtigen Moment“ \overline{SS}

Rechnertechnik

- 1 Einführung**
- 2 Vom Schaltkreis zum Computer**
- 3 Architekturmerkmale von Prozessoren**
- 4 Der CPU-Stack**
- 5 Anwender-Software**
- 6 Pipelining**
 - 6.1** Konzept
 - 6.2** Arithmetik-Pipelines
 - 6.3** Instruktions-Pipelines
- 7 Bus-Systeme**
 - 7.1** Was sind Bus-Systeme?
 - 7.2** Chipsatz
 - 7.3** RS-232
 - 7.4** I²C (TWI)
 - 7.5** SPI
- i Quantencomputer**