Technische Grundlagen der Informatik 2

Rechnerorganisation

Kapitel 3: Rechnerarithmetik

Prof. Dr. Ben Juurlink

Fachgebiet: Architektur eingebetteter System
Institut für Technische Informatik und Mikroelektronik
Fak. IV – Elektrotechnik und Informatik

SS 2013

Ziele

Nach dieser Vorlesung sollten Sie in der Lage sein:

- Binär-/Octal-/Hexadezimalzahlen zu Dezimalzahlen zu konvertieren und umgekehrt.
- 2-Komplement-Zahlen zu negieren.
- m-Bit 2-Komplement-Zahlen zu n-Bit zu konvertieren.
- die Unterschiede zwischen `addi` und `addiu`, `slt` und `sltu`, `lb` und `lbu`, etc. zu benennen.
- Überlauf (overflow) zu erklären. Erläutern wann er auftritt und wie er behandelt wird.
- Einen einfachen Addierer zu designen und zu erweitern.
- Einen „Carry-Look-Ahead“-Addierer zu beschreiben.
- Eine Fraktalzahl in „IEEE 754“ FP-Standard zu konvertieren und umgekehrt.
- MIPS-FP (Floating Point)-Befehlssatz zu nutzen.

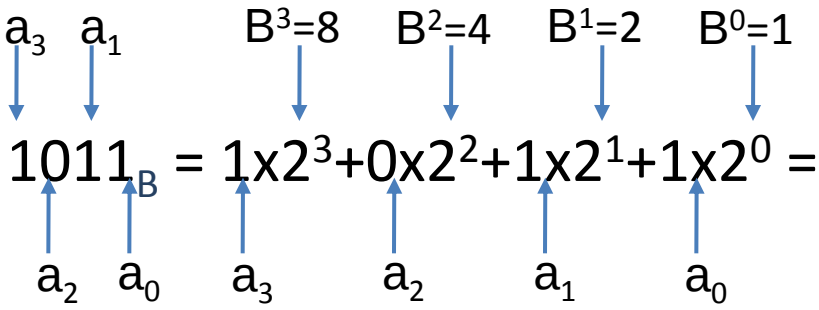
Inhalte

- Vorzeichenbehaftete (signed) und vorzeichenlose (unsigned) Zahlen
- Addition und Subtraktion
- Multiplikation
- Division
- Gleitkommazahlen (Floating Point)
- Gleitkommazahlen in MIPS
- „Pentium bug“

Natürliche Zahlen können in jeder Basis repräsentiert werden:

$$a_{n-1}a_{n-2}\ldots a_1 a_{o(\text{Basis B})} = a_{n-1}B^{n-1} + \ldots + a_1 B^1 + a_o B^o = \sum_{i=0}^{n-1} a_i B^i$$

- B: Basis (Radix), z.B. 10 (dezimal), 2 (binär), 8 (octal), 16 (hexadezimal)
- B^i: Gewicht (weight) der i-ten Ziffer (digit)
- a_i: i-te Ziffer aus der Menge {0, 1, …, B-1}



- Zerlegung nach **Horner-Schema**:

$$\sum_{i=0}^{n-1} a_i B^i = ((...(a_{n-1}B + a_{n-2})B + ... + a_2)B + a_1)B + a_0$$

- Dezimal nach dual / binär:

167_D ->	167 / 2 = 83	Rest 1
	83 / 2 = 41	Rest 1
	41 / 2 = 20	Rest 1
	20 / 2 = 10	Rest 0
	10 / 2 = 5	Rest 0
	5 / 2 = 2	Rest 1
	2 / 2 = 1	Rest 0
	1 / 2 = 0	Rest 1

Niederwertigstes Bit ***(least significant bit* (LSB))**

Höchstwertigstes Bit ***(most significant bit* (MSB))**

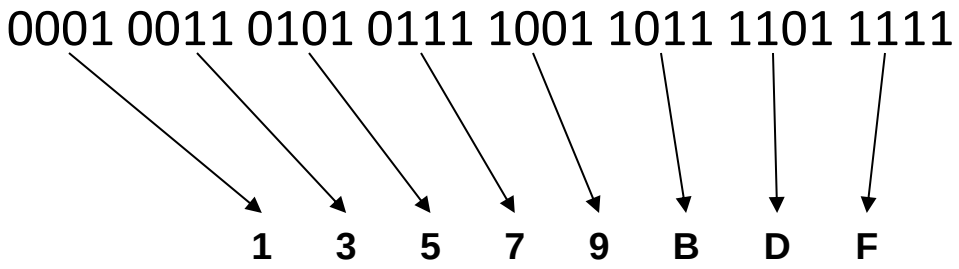
- $167_D = 10100111_B$

- Um lange Folgen mit Binärzahlen zu vermeiden, werden oft Hexadezimalzahlen (Basis 16) verwendet.
- Ziffernmenge: { 0, 1, 2, … , 8, 9, A, B, … ,F }
- Dezimal nach hexadezimal:

  167_D -> 167 / 16 = 10 Rest 7

  10 / 16 = 0 Rest A

  - $167_D = A7_H$
  - In C/Java: 0xa7
- Binär nach hexadezimal:



Einige Möglichkeiten negative Zahlen zu repräsentieren:

- *Vorzeichen-/Betrags-Zahlen* (Sign-magnitude numbers)
  - MSB zeigt Vorzeichen (sign) an (0: positiv, 1: negativ).
  - Die übrigbleibenden (n-1) Bits bilden den Betrag (magnitude).
- *1-Komplement-Zahlen* (One's complement numbers)
  - Zahl wird durch die Invertierung aller Bits negiert.
  - MSB *impliziert* das Vorzeichen.
- *2-Komplement-Zahlen* (Two's complement numbers)
  - MBS hat ein negatives Gewicht (-2^{n-1}).
  - $b_{n-1}b_{n-2}\ldots b_1b_0$ (binär) = $-b_{n-1}2^{n-1}+b_{n-2}2^{n-2}+\ldots+b_12^1+b_02^0$ (dezimal)
  - MSB *impliziert* das Vorzeichen.

Beispiel (4-Bit)

Dezimal	Vorzeichen-/Betrags-Zahlen (Sign-Magnitude)	1-Komplement-Zahlen (One's complement)	2-Komplement-Zahlen (Two's complement)
-0	1000	1111	
-1	1001	1110	1111
-2	1010	1101	1110
-3	1011	1100	1101
-4	1100	1011	1100
-5	1101	1010	1011
-6	1110	1001	1010
-7	1111	1000	1001
-8			1000

Und der Gewinner ist …

- 2-Komplement-Zahlen
  - Arithmetik ist einfacher (identisch zu vorzeichenlos).
  - Es gibt nur eine Möglichkeit die 0 zu repräsentieren.

- MIPS benutzt 32-Bit 2-Komplement-Zahlen
- $b_{31}b_{30}…b_1b_0$ (binär) = $-b_{31}2^{31}+b_{30}2^{30}+…+b_12^1+b_02^0$ (dezimal)
  - Was ist die kleinste von MIPS repräsentierbare Integerzahl?
  - Was ist die größte von MIPS repräsentierbare Integerzahl?
  - Was ist die binäre Repräsentation der 42? Hexadezimal?

Negation

- Negation von 2-Komplement-Zahlen:
  - Invertiere alle Bits und addiere 1
- 8-Bit-Beispiel: $0110\ 1001_B = +105_D$

invertieren: 1001 0110

1 addieren: 1001 0111 = -128 + 16 + 7 = -105

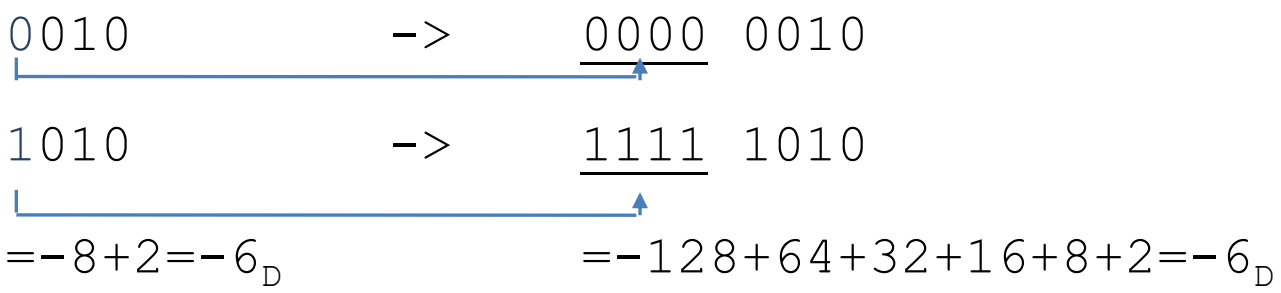
Rückwärts:

invertieren: 0110 1000

1 addieren: 0110 1001

Vorzeichenerweiterung

- Konvertierung einer n-Bit 2-Komplenent-Zahl in eine m-Bit 2-Komplenent-Zahl (m>n):
  - MIPS' 16-Bit Immediats werden für die Arithmetik auf 32 Bit erweitert.
  - Kopiere das MBS (sign bit) in die anderen Bitpositionen.
  - Beispiel (4-Bit -> 8-Bit):



  - Dies wird Vorzeichenerweiterung (sign extension) genannt.

Vorzeichenbehaftete und vorzeichenlose Zahlen

- Einige Sprachen (z.B. *C*) können mit vorzeichenbehafteten (signed) und vorzeichenlosen (unsigned) Zahlen arbeiten.

C Datentyp	MIPS Datentyp	MIPS Ladebefehl
`int`	32-Bit Wort	`lw`
`unsigned int`	32-Bit wort	`lw`
`short`	16-Bit Halbwort	`lh`
`unsigned short`	16-Bit Halbwort (unsigned)	`lhu`
`char`	Byte	`lb`
`unsigned char`	Byte (unsigned)	`lbu`

- *Java*s primitive Datentypen (`byte`, `short`, `int`, `long`) sind signed.
- `lh` und `lb` machen für das zu ladene Byte/Halfwort eine Vorzeichenerweiterung (`lhu` und `lbu` nicht).

Vorzeichbehaftete Vs. vorzeichenlose Vergleiche

- MIPS bietet 2 Versionen für einen *set-on-less-than*–Befehl an:
  - `slt` und `slti` arbeiten mit vorzeichenbehafteten Integers.
  - `sltu` und `sltiu` arbeiten mit vorzeichenlosen Integers.
- Beispiel:

```
$s0 = 1111 1111 1111 1111 1111 1111 1111 1111
```

= -1 (signed), 2^{32}-1 (unsigned)

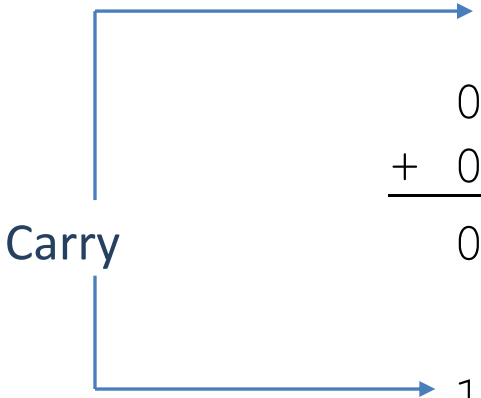
```
$s1 = 0000 0000 0000 0000 0000 0000 0000 0001
```

= 1 (signed und unsigned)

```
slt  $t0,$s0,$s1      # $t0 = 1 (wahr[true])
sltu $t1,$s0,$s1      # $t1 = 0 (falsch[false])
```

Binäre Addition

- Addition von rechts nach links mit Übertrag (carry) wie in der Grundschule
- Beispiele (4-Bit):




```
  1                    
  0010          2      
+ 0011        + 3      
------        ---      
  0101          5      
```

```
  1                    
  0101          5      
+ 0110        + 6      
------        ---      
  1011         -5      
```

Rechenregeln:

- 0 + 0 = 0
- 0 + 1 = 1
- 1 + 0 = 1
- 1 + 1 = $1_{Carry}0$

??? Überlauf (overflow)

Binäre Subtraktion

- Beispiele (4-Bit):

```
  0010        2
- 0011      - 3
  ---
  111
  ---
  1111       -1

  1101       -3
- 0110      - 6
  ---
  11
  ---
  0111        7
```

Borrows

wieder Überlauf

Rechenregeln:

- 0 - 0 = 0
- 0 - 1 = 1 (Borrow)
- 1 - 0 = 1
- 1 - 1 = 0

- Alternative:
  Den 2. Operanden negieren und dann beide addieren.(Wir werden diesen Trick beim Erstellen unserer Arithmetic Logic Unit (ALU) nutzen.)

Überlauf / 1

- **Überlauf(overflow)**:
  - Das Ergebnis ist zu groß für ein endliches Computer-Wort.
  - z.B. Addition von zwei n-Bit-Zahlen muss keine n-Bit-Zahl ergeben.
- Kein Überlauf, wenn …
  - Addition von 2 Zahlen mit entgegengesetztem Vorzeichen
    - Betrag der Summe ist immer ≤ den Beträgen der beiden Operanden
    - Beispiel: -10 + 6 = -4
  - Subtraktion von 2 Zahlen mit dem selben Vorzeichen
- Overflow tritt auf, wenn…

Operation	A	B	Ergebnis
A+B	≥ 0	≥ 0	< 0
A+B	< 0	< 0	≥ 0
A-B	≥ 0	< 0	< 0
A-B	< 0	≥ 0	≥ 0

Überlauf / 2

- Vergleich der Operantionen A + B und A – B
  - Kann ein Überlauf auftreten, wenn B = 0 ist?
  - Kann ein Überlauf auftreten, wenn A = 0 ist?
- Überlauf tritt nur auf, wenn das Übertragsbit (carry in) für das MSB ≠ dem entstehenden Übertragsbit (carry out) aus der Operation der beiden MSBs ist.
- Beispiele (4-Bit):

carry out → 0 1 ← carry in

```
  0101
+ 0110
------
  1011
```

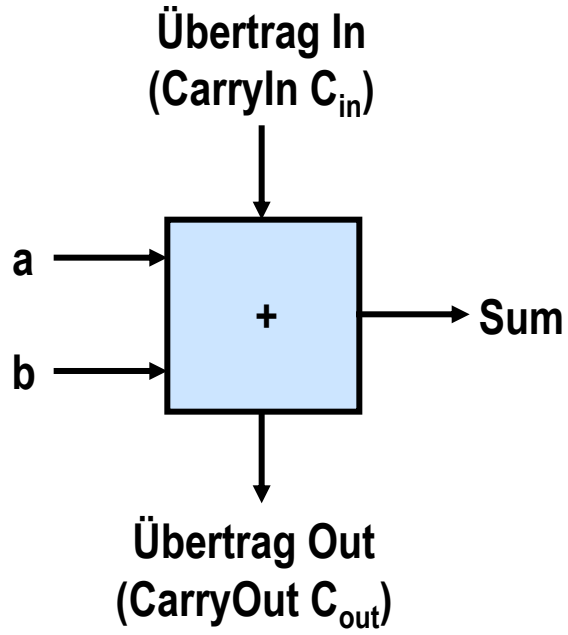
Überlaufbehandlung

- Wie werden Überläufe behandelt?
  - Eine Exception (Interrupt) wird ausgelöst.
  - Sprung zu einer vordefinierten Adresse, um die Ausnahme (Exception) zu behandeln.
    - Interrupt Handler (Teil des Betriebssystems (Operating System [OS]))
    - Register **`$k0`** (26) und **`$k1`** (27) sind fürs OS reserviert.
  - Unterbrochene Adresse wurde für einen möglichen Rücksprung im exception program counter (`$epc`) gespeichert.
    - Neuer MIPS-Befehl: *move from coprocessor* 0 (`mfc0 $k0,$epc`)
- Details basieren auf dem Software-System / der Software-Sprache
  - C/Java ignorieren Überlauf, Fortran nicht
- Nicht immer soll ein Überlauf angezeigt werden.
  - neue MIPS-Befehle: `addu, addiu, sltu, sltiu, subu`
  - **`addiu` und `sltiu` "sign extend" ihre 16-Bit Immediats bereits!**

1-Bit Volladdierer

- Wir werden jetzt ein Addierer basteln.
- Dazu brauchen wir einen 1-Bit Volladdierer (*1-bit Full Adder*).

Blockbild:



Wahrheitstabelle (Truth table):

a	b	C_{in}	C_{out}	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Boolsche Gleichungen (Summe von Produkten):

$$Sum = \bar{a}\bar{b}c_{in} + \bar{a}b\bar{c}_{in} + a\bar{b}\bar{c}_{in} + abc_{in}$$

$$c_{out} = \bar{a}bc_{in} + a\bar{b}c_{in} + ab\bar{c}_{in} + abc_{in}$$

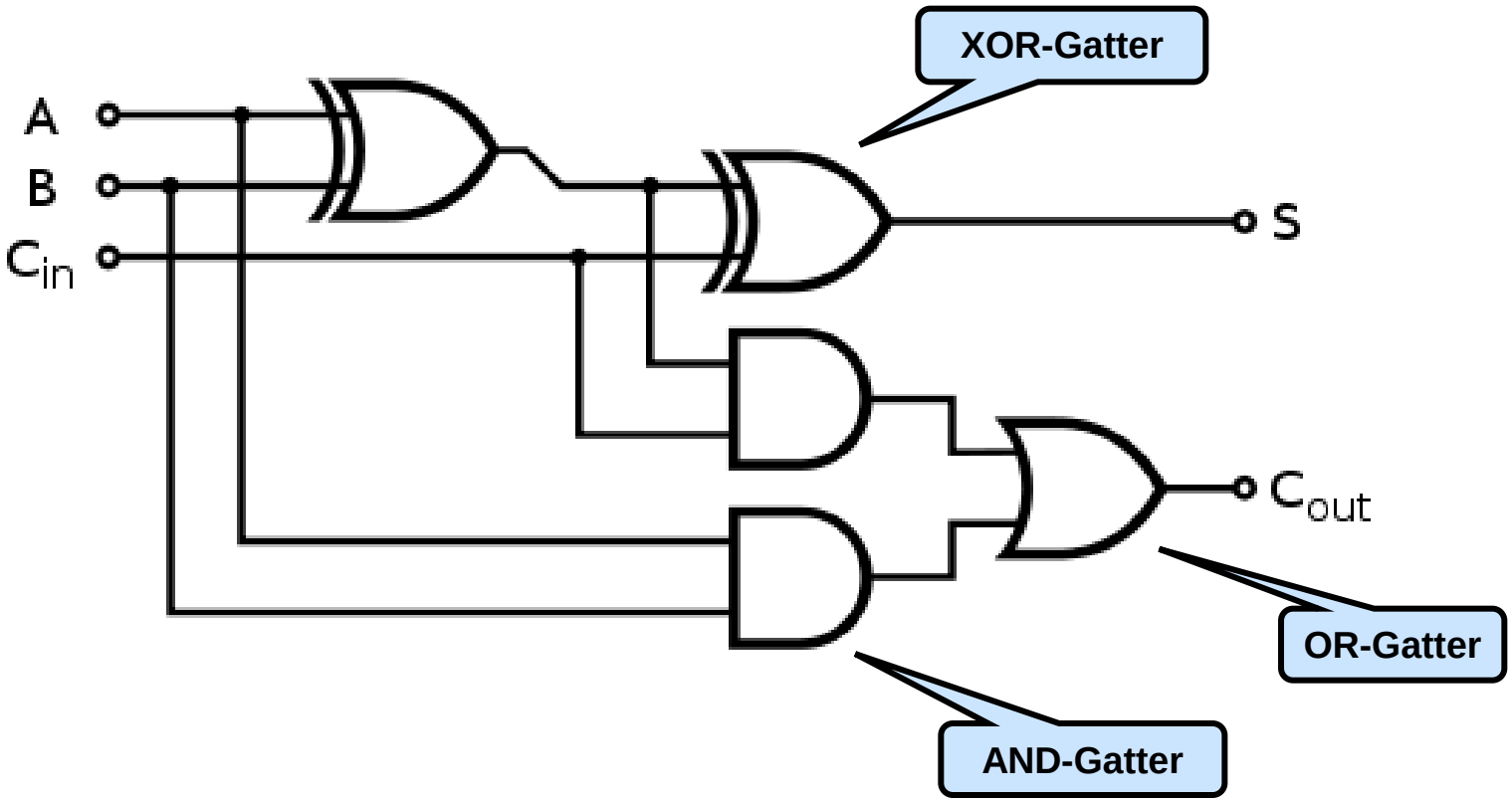
Vereinfachung:

$$Sum = a \text{ XOR } b \text{ XOR } c_{in}$$

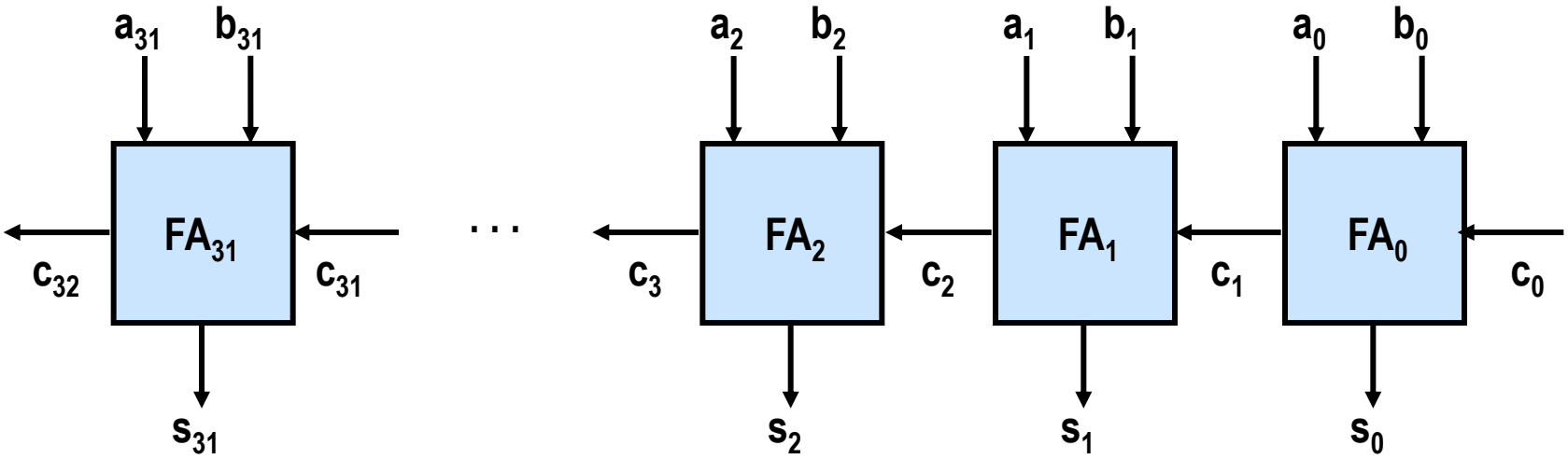
$$c_{out} = ab + ac_{in} + bc_{in}$$

a	b	C_{in}	C_{out}	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	**1**	0
1	0	0	0	1
1	0	1	**1**	0
1	1	0	**1**	0
1	1	1	**1**	1

1-Bit Volladdierer (Schaltung)



Ripple-Carry-Addierer:



FA = Full Adder

- Problem: Lange Durchlaufzeit für den Übertrag bei großer Stellenzahl, z.B. bei 32-Bit- oder 64-Bit-Addierer.
- Lösung: Parallelisierung der Übertragsbildung:

  →Carry-Look-Ahead-Addierer (CLA-Addierer)

Carry-Look-Ahead Addierer (Motivation)

- Gleichungen (Summe von Produkten):

  $c_1 = b_0c_0 + a_0c_0 + a_0b_0$

  $c_2 = b_1c_1 + a_1c_1 + a_1b_1 = b_1b_0c_0 + b_1a_0c_0 + b_1a_0b_0 + a_1b_0c_0 + a_1a_0c_0 + a_1a_0b_0 + a_1b_1$

  $c_3 = b_2c_2 + a_2c_2 + a_2b_2 = \ldots$

  $c_4 = b_3c_3 + a_3c_3 + a_3b_3 = \ldots$
- Nicht umsetzbar! Warum?
- VA_i wird immer ein Übertrag generieren, wenn $g_i = a_i\, b_i\, (= 1)$
- VA_i propagiert ein Übertrag, wenn $p_i = a_i + b_i\, (= 1)$

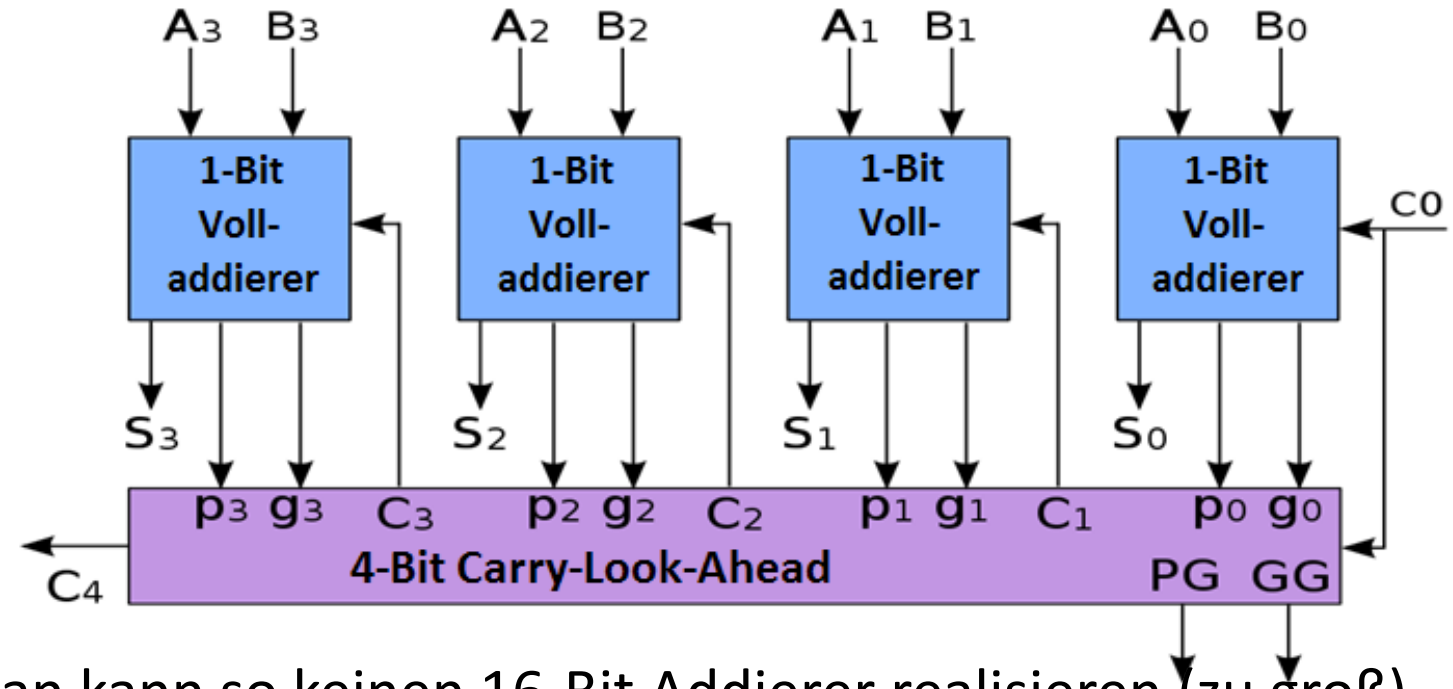
  $c_1 = g_0 + p_0c_0$

  $c_2 = g_1 + p_1c_1 = g_1 + p_1g_0 + p_1p_0c_0$

  $c_3 = g_2 + p_2c_2 = g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0c_0$

  $c_4 = g_3 + p_3c_3 = \ldots$
- Umsetzbar! Warum?

Carry-Look-Ahead Addierer (4-Bit)



- Man kann so keinen 16-Bit Addierer realisieren (zu groß).
- Man könnte den Übertrag durch ein Hintereinanderschalten von 4-Bit CLA-Addierern realisieren (Ripple-Carry-Prinzip).
- Besser: erneutes Anwenden des CLA -Prinzips!

Multiplikation

- Für den Moment betrachten wir nur positive Zahlen.
- Schulmathematik:

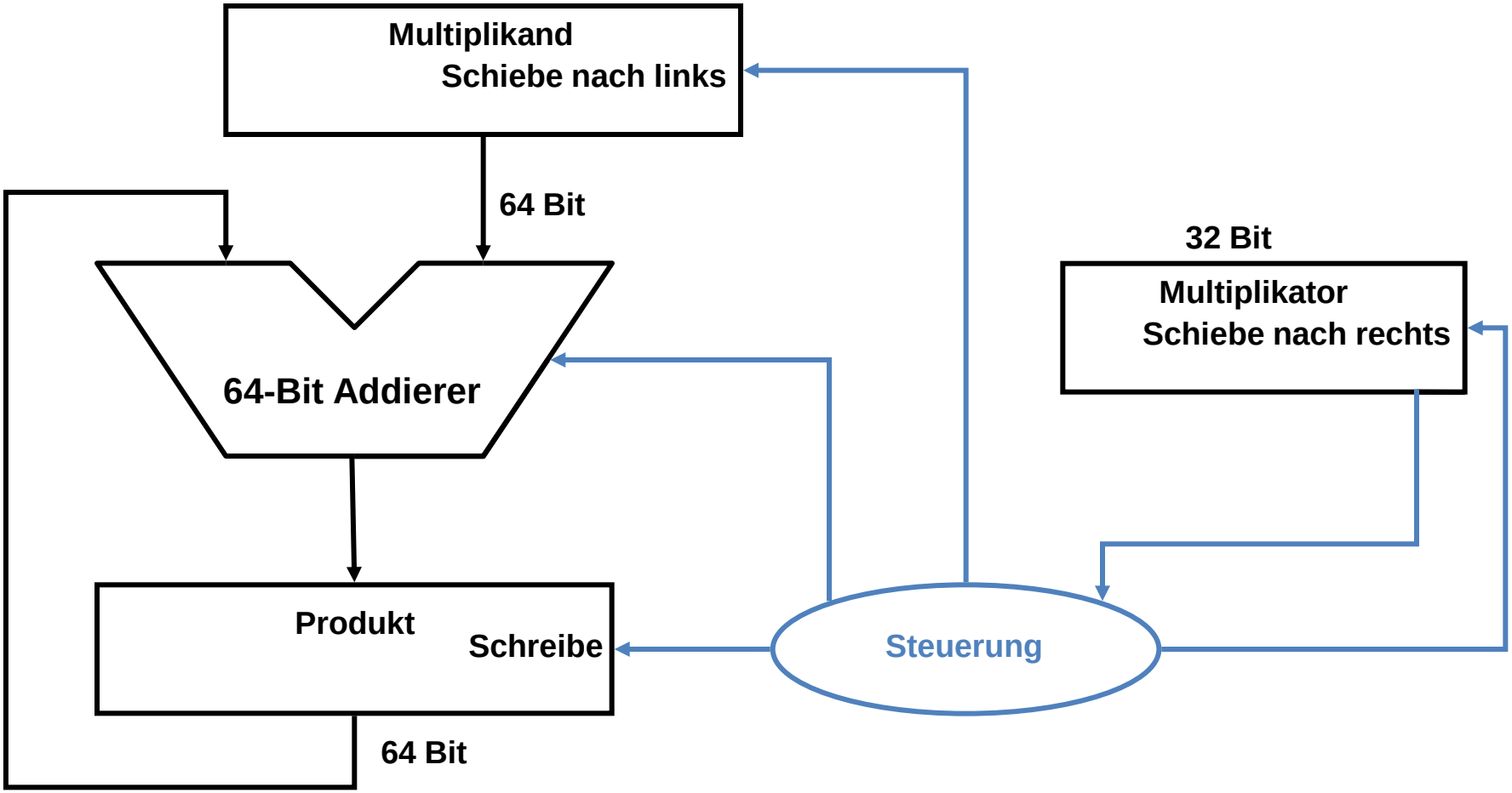
Multiplikand		1101	13
Multiplikator	x	1011	x 11
		1101	
		1101	
		0000	
		1101	
Produkt		10001111	143

- Produkt erfordert doppelte Stellenanzahl.

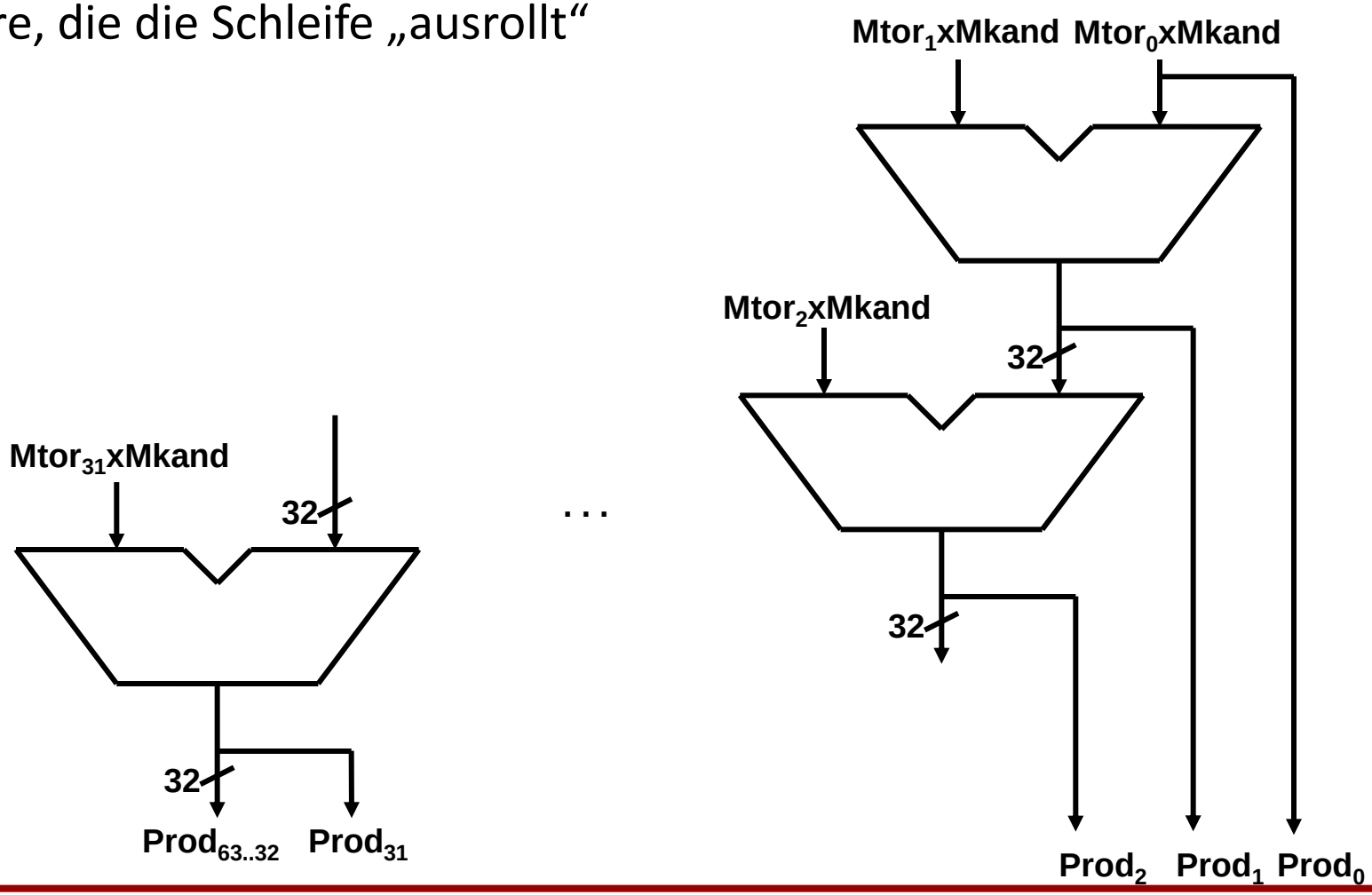
C-Algorithmus für einen sequentiellen Multiplizierer

- Multiplikand: 2n-Bit Register
- Multiplikator: n-Bit Register
- Produkt: 2n-Bit Register
- Verfügen über 2n-Bit Addierer

```
for (i=0; i<n; i++)
{
   if (Multiplikator&1)          // Multiplikator0==1
      Produkt += Multiplikand;
   Multiplikator >>= 1;          // Schiebe rechts 1 Bit
   Multiplikand <<= 1;           // Schiebe links 1 Bit
}
```


Multiplikand
Schiebe nach links
64 Bit
32 Bit
Multiplikator
Schiebe nach rechts
64-Bit Addierer
Produkt
Schreibe
Steuerung
64 Bit

Hardware, die die Schleife „ausrollt“



Multiplikation in MIPS

- Das Multiplizieren von zwei 32-Bit Zahlen kann ein 64-Bit Produkt ergeben.
  - Neue Register: **Hi** und **Lo**
    - **Hi** beinhaltet die 32 MSBs des 64-Bit Produkts.
    - **Lo** beinhaltet die 32 LSBs.
  - MIPS-Befehle:
    - `mult $s2,$s3` `# Hi#Lo = $s2x$s3`
    - `multu $s2,$s3` `# Hi#Lo = $s2x$s3 (unsigned)`
    - *Move from lo*: `mflo $s1` `# $s1 = Lo`
    - *Move from hi*: `mfhi $s1` `# $s1 = Hi`
- Pseudo-Instruktion: `mul $s1,$s2,$s3`
- Reale Umsetzung:

```
mult $s2,$s3
mflo $s1
```

Division / 1

- Schulmathematik:

Dividend		Divisor		Quotient
1001010	/	1000	=	1
1000				
1				

Division / 2

- Schulmathematik:

Dividend		Divisor		Quotient
1001010	/	1000	=	10

```
1001010
1000
```

```
   10
```

Division / 3

- Schulmathematik:

Dividend Divisor Quotient

```
1001010  /  1000    =   100
1000
 101
```

- Schulmathematik:

Dividend		Divisor		Quotient
1001010	/ 1000	=	1001	74/8= 9 Rest 2

```
1001010
1000
   1010
   1000
     10   → Rest
```

=9

=2

Divisionsalgorithmus

C-Algorithmus für Division

- Quotient:
  n-Bit Register
- Divisor:
  2n-Bit Register, Divisor wird zu Beginn in der linken Hälfte abgelegt
- Rest:
  2n-Bit Register, wird mit Dividenden initialisiert
- Verfügen über 2n-Bit Substrahierer

```
Quotient = 0;
Rest = Dividend;
Divisor <<= n;
for (i=0; i<=n; i++){
   Quotient <<= 1;
   if (Rest>=Divisor){
      Rest -= Divisor;
      Quotient |= 1;
   }
   Divisor >>= 1;
}
```

4-Bit Beispiel

- 7_D durch 2_D, d. h. 0111_B durch 0010_B

Iteration	Quot.	Divisor	Rest	Anm.
Anfangsw.	0000	0010 0000	0000 0111	R < D
0	0000	0001 0000	0000 0111	R < D
1	0000	0000 1000	0000 0111	R < D
2	0000	0000 0100	0000 0111	R ≥ D
3	0001	0000 0010	0000 0011	R ≥ D
4	0011	0000 0001	0000 0001	

```
Quotient = 0;
Rest = Dividend;
Divisor <<= n;
for (i=0; i<=n; i++){
    Quotient <<= 1;
    if (Rest>=Divisor){
        Rest -= Divisor;
        Quotient |= 1;
    }
    Divisor >>= 1;
}
```

Division in MIPS

- Dividieren in MIPS:
  - Benutzt auch die Register **`Hi`** und **`Lo`**
    - Quotient befindet sich im **`Lo`**, Rest im **`Hi`.**
    - Rest wird als Nebeneffekt der Division mitproduziert.
- MIPS-Befehle:
  - `div $s2,$s3 # Lo = $s2/$s3, Hi = $s2%$s3`
  - `divu $s2,$s3 # idem unsigned`
- Pseudo-Befehle:
  - `div $s1,$s2,$s3`
  - `rem $s1,$s2,$s3`

- Darstellung gebrochener Zahlen:

$$a_{n-1}a_{n-2}\ldots a_0, a_{-1}a_{-2}\ldots a_{-m}$$
$$= a_{n-1}B^{n-1} + \ldots + a_0B^0 + a_{-1}B^{-1} + a_{-2}B^{-2} + \ldots + a_{-m}B^{-m}$$
$$= \sum_{i=0}^{n-1} a_i B^i + \sum_{i=-m}^{-1} a_i B^i$$

- Beispiel (binär):

$11.1010_B = 3 + 2^{-1} + 2^{-3}$

$= 3 + 0{,}5 + 0{,}125 = 3{,}625_D$

- Dezimal nach Dual
  - Beispiel: $0{,}24_D$

$0{,}24_D$ ->

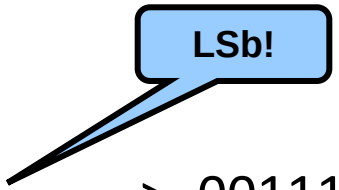
$0.24 \cdot 2 = 0.48 + 0$ 

$0.48 \cdot 2 = 0.96 + 0$

$0.96 \cdot 2 = 0.92 + 1$

$0.92 \cdot 2 = 0.84 + 1$

$0.84 \cdot 2 = 0.68 + 1$

$0.68 \cdot 2 = 0.36 + 1$ 

$0.36 \cdot 2 = 0.72 + 0$

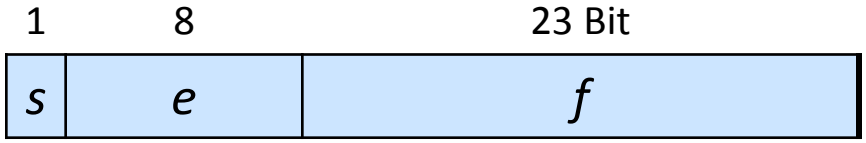
$0.72 \cdot 2 = 0.44 + 1$ -> $.00111101_B$

  - Abbruch nach 8 Stellen (Näherung mit 0,238…)

Gleitkommazahlen (*floating point*)

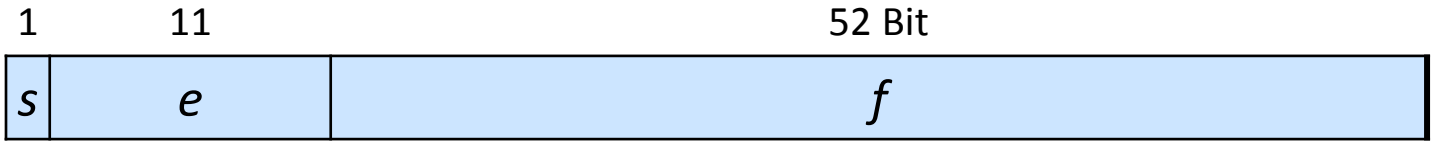
- Näherung für reelle Zahlen: $(-1)^s \times 1.f \times 2^E$
  - *s*: Vorzeichen (*sign*): 0 → positiv, 1 → negativ
  - 1.*f*: Mantisse (Betrag, *significand*) als normalisierte Zahl
    - Zahl wird so lange geschoben, bis sie führende 1 aufweist
    - Binärpunkt wird rechts von dieser 1 festgelegt ($1.0 \leq 1.f < 2.0$)
  - *f*: nur der Bruch *f* (*fraction*) wird gespeichert, führende 1 ist implizit (wird von Recheneinheit ergänzt)
  - *E*: vorzeichenbehafteter Exponent, wird als transformierter Exponent *e* gespeichert
  - *e*: $e = E + \text{bias}$
    - bias wird so gewählt, dass 2-Komplement-Zahl *E* zur vorzeichenlosen Dualzahl *e* wird

- Einfache Genauigkeit (*single precision*, 32 Bit)



  - bias = 127
  - C/Java: `float`

- Doppelte Genauigkeit (*double precision*, 64 Bit)



  - bias = 1023
  - C/Java: `double`

- $-0{,}75_D$ mit einfacher Genauigkeit
  - $s = 1$
  - $0{,}75_D$ als gebrochene Dualzahl ist 0.11_B
  - Normalisiere: $0.11 = 1.1 \times 2^{-1}$
    - führende 1 ist implizit → Bruch = 10000....
  - transformierter Exponent e

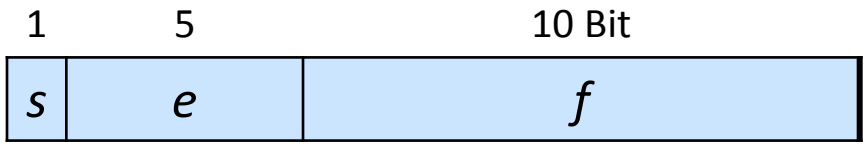
$$e = E + \text{bias} = -1 + 127 = 126 = 0111\ 1110_B$$

1	8	23 Bit
1	0111 1110	100 0000 0000 0000 0000 0000

Sonderfälle

- Neben normalisierten Zahlen sind außerhalb des Zahlenraums definiert:
  - ± Null
  - ± Unendlich: z. B. Division durch Null
  - ± unnormalisierte (*unnormalized*) Zahlen: winzige (*tiny*) Zahlen
  - Nichtzahlen (Not a Number, NaN): Ergebnis ungültiger Operation wie 0/0
- Codiert durch den größten und kleinsten Exponentwert e und f:
  - normal.: $(-1)^s \times 1.f \times 2^{e-127/-1023}$ $1 \leq e \leq 254/2046$
  - Null: $(-1)^s \times 0$ $e = 0, f = 0$
  - Unendlich: $(-1)^s \times \infty$ $e = 255/2047, f = 0$
  - unnorm.: $(-1)^s \times 0.f \times 2^{-126/-1022}$ $e = 0, f \neq 0$; interpretiert mit $e = 1$
  - Nichtzahl: NaN $e = 255/2047, f \neq 0$

- Beispiel basiert auf 16-Bit Minifloat Format:



bias = 15
Exponentenbereich: $-14 \leq E \leq 15$

- $Z = X + Y$ mit
  - $X = 2{,}35_D =$ $10.0101\ 1001\ 1001\ 1001\ \ldots_B$
  - $Y = 10{,}17_D =$ $1010.0010\ 1011\ 1000\ 0101\ \ldots_B$

- **1. Schritt:**
  - **Normalisieren** und **Anpassung** an 16-Bit-Format:
  - $X = 1.0010\ 1100\ 11 \cdot 2^1$
  - $Y = 1.0100\ 0101\ 01 \cdot 2^3$

    → Verlust von signifikanten Stellen!

- **2. Schritt:**
  - **Vergleichen der beiden Exponenten** E.
  - Bei Ungleichheit kleinere Exponent an den größeren anpassen
  - $X = 0.0100\ 1011\ 00\ 11 \cdot 2^3$

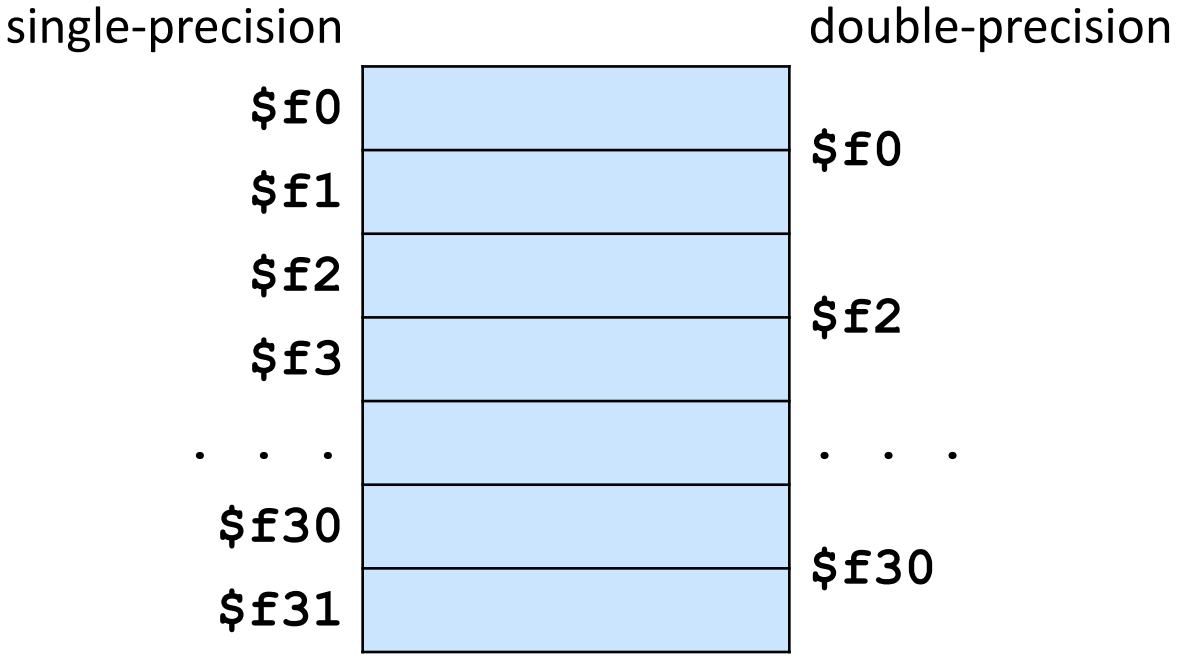
    → rot dargestellten Stellen gehen verloren
- **3. Schritt:**
  - **Addieren der Mantissen**:

$$\begin{array}{r} 0.0100\ 1011\ 00\ (X) \\ \underline{+\ 1.0100\ 0101\ 01\ (Y)} \\ 1.1001\ 0000\ 01\ (Z) \end{array}$$

- **Ergebnis**
  - muss ggf. noch normalisiert werden (hier nicht)
  - $Z = 1.1001\ 0000\ 01 \cdot 2^3 = 12{,}500\ 975\ 656\ 2510$ (korrekt wäre: 12,52).

MIPS Floating-Point-Register

- MIPS hat
  - 32 Floating-Point-Register mit einfacher Genauigkeit (single-precision) [`$f0,$f1,..., $f31`] **oder**
  - 16 Register mit doppelter Genauigkeit (double-precision) [`$f0,$f2,...,$f30`]



MIPS Floating Point Befehle

FP add single	`add.s $f0,$f1,$f2`	$f0=$f1+$f2
FP sub. single	`sub.s $f0,$f1,$f2`	$f0=$f1-$f2
FP mult. single	`mul.s $f0,$f1,$f2`	$f0=$f1*$f2
FP div. single	`div.s $f0,$f1,$f2`	$f0=$f1/$f2
FP add double	`add.d $f0,$f2,$f4`	$f0,$f1 = $f2,$f3 + $f4,$f5
FP sub. double	`sub.d $f0,$f2,$f4`	$f0,$f1 = $f2,$f3 - $f4,$f5
FP mult. double	`mul.d $f0,$f2,$f4`	$f0,$f1 = $f2,$f3 * $f4,$f5
FP div. double	`div.d $f0,$f2,$f4`	$f0,$f1 = $f2,$f3 / $f4,$f5
load word coproc. 1	`lwc1 $f0,100($s0)`	$f0=Mem[$s0+100]
store word copr. 1	`swc1 $f0,100($s0)`	Mem[$s0+100]=$f0
branch on coproc.1 true	`bc1t 25`	if (cond) goto PC+4+100
branch on coproc.1 false	`bc1f 25`	if (!cond) goto PC+4+100
FP compare single	`c.lt.s $f0,$f1`	cond = ($f0 < $f1)
FP compare double	`c.ge.d $f0,$f2`	cond = ($f0,$f1 >= $f2,$f3)

MIPS Floating Point Sprungbefehle / 1

- Vergleich-Befehle setzen den condition code (cc)
- Sukzessive Branch-Befehle testen, ob cc erfüllt ist (*true*) oder nicht (*false*)
- Beispiel: Suche nach kleinstem *n*, so dass gilt $0.5^n \leq 1.0 \cdot 10^{-9}$

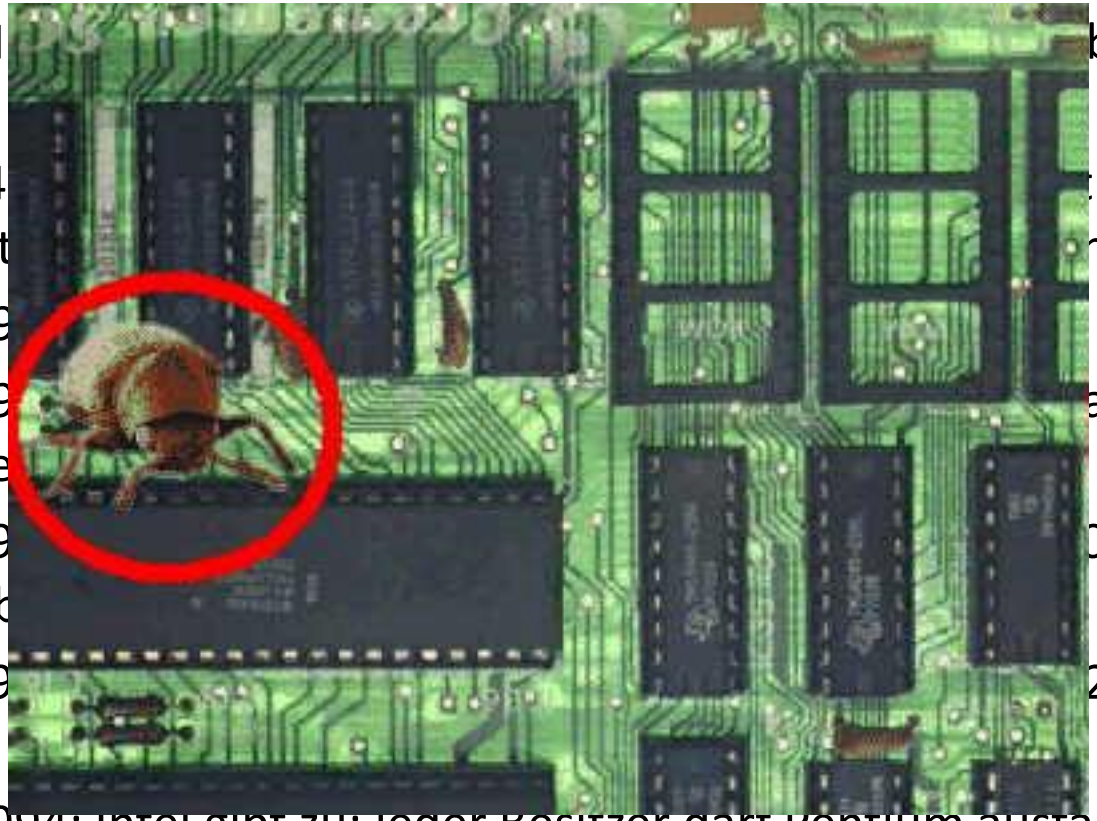
```
int n = 1;
float exp = 0.5;
while (exp > 1e-9){
   exp = exp*0.5;
   n++;
}
```

MIPS Floating Point Sprungbefehle / 2

```
int n = 1;
float exp = 0.5;
while (exp > 1e-9){
    exp = exp*0.5;
    n++;
}
```

```
   addi     $t0,$zero,1        # n = 1
   lwc1     $f0,fphalf($gp)    # exp = 0.5
   lwc1     $f1,fptiny($gp)    # $f1 = 1e-9
   lwc1     $f2,fphalf($gp)    # $f2 = 0.5
while:
   c.gt.s   $f0,$f1            # cc = exp>1e-9
   bc1f     endwhile           # if (!cc) goto enwhile
   mul.s    $f0,$f0,$f2        # exp = exp*0.5
   addi     $t0,$t0,1          # n++
   j        while              # goto while
endwhile:   ...
```

Fehler im Divisionsalgorithmus für Gleitpunktzahlen



- Juli 1994: I ben: einige 100K$
- Sept. 1994 keine offizielle St nternet
- 7. Nov. 199
- 22. Nov. 19 an 9. Stelle verursache
- 5. Dez. 199 000 Jahren auftreten k
- 12. Dez. 19 24 Tage Fehler
- 21. Dez. 1994: Intel gibt zu: Jeder Besitzer darf Pentium austauschen. Geschätzte Kosten: **500 M$**!

4195835.0/3145727.0 = 1.333 820 449 136 241 002 (korrekter Wert)
4195835.0/3145727.0 = 1.333 739 068 902 037 589 (fehlerhaften Pentium)

Zusammenfassung

- Computer-Arithmetik ist beschränkt durch limitierte Genauigkeit.
  - *Overflow* (Überlauf)
  - Underflow (floating point)
- Bitmuster haben keine inherente Bedeutung, es gibt jedoch Standards
  - 2-Komplement
  - IEEE 754 Floating Point
- Computer-Befehle bestimmen die “Bedeutung” der Bitmuster.
- Leistung und Genauigkeit sind wichtig. Daraus ergeben sich viele Komplexitäten für reale Maschinen.
  - z.B. Algorithmen und Implementierungen
- Was kommt als Nächstes?
  - Wir werden einen Prozessor implementieren.
  - Zuerst müssen wir jedoch “Leistung” verstehen.