Einführung in die Programmierung

Tutorium 7

Wintersemester 2025/2026

Arbeitsgruppe Systemsoftware
Angewandte Informatik 12

technische universität
dortmund

Arbeitsgruppe
Systemsoftware

Implementieren Sie die IEEE-754-konforme Addition zweier Fließkommazahlen (`ieee32_add_u`) rein mittels Ganzzahlarithmetik und Bitoperationen, inklusive aller Teilschritte wie

- Exponentenausrichtung,
- Addition der Mantissen,
- Normalisierung nach Subtraktion
- Rundung

Beispiel

34 + (-7)

Umwandlung von 34 in IEEE 754

1. Vorzeichen dokumentieren

Falls positiv: S = 0

Falls negativ: S = 1

2. Umwandlung in eine Binärzahl

$(34)_D = (100010,0)_B$

3. Normalisierung der Binärzahl

$(100010,0)_B = (1,000100)_B * 2^5$

Beispiel **Umwandlung von 34 in IEEE 754**

34 + (-7)

3. Normalisierung der Binärzahl

$(100010,0)_B = (1,000100)_B * 2^5$

4. Exponenten berechnen

$(5)_D + (127)_D$ (Bias) $= (132)_D = (10000100)_B$

5. Mantisse erzeugen

Nehme von $(1,000100)_B$ Nachkommastellen, also 000100.

Beispiel **Umwandlung von 34 in IEEE 754**

34 + (-7)

6. Alles zusammensetzen

0	1000 0100	000 1000 0000 0000 0000 0000

Beispiel

34 + (-7)

Umwandlung von -7 in IEEE 754

1. Vorzeichen dokumentieren

Falls positiv: S = 0

Falls negativ: S = 1

2. Umwandlung in eine Binärzahl

$(7)_D = (111,0)_B$

3. Normalisierung der Binärzahl

$(111,0)_B = (1,11)_B * 2^2$

Beispiel **Umwandlung von -7 in IEEE 754**

34 + (-7)

3. Normalisierung der Binärzahl

$(111,0)_B = (1,11)_B * 2^2$

4. Exponenten berechnen

$(2)_D + (127)_D$ (Bias) $= (129)_D = (10000001)_B$

5. Mantisse erzeugen

Nehme von $(1,11)_B$ Nachkommastellen,
also 11.

Beispiel **Umwandlung von -7 in IEEE 754**

34 + (-7)

6. Alles zusammensetzen

1	1000 0001	110 0000 0000 0000 0000 0000

Beispiel

34 + (-7)

Addition von 34 und -7 in IEEE 754

0	1000 0100	000 1000 0000 0000 0000 0000	$(34)_D$
1	1000 0001	110 0000 0000 0000 0000 0000	$(-7)_D$

1. Extraktion der Komponenten
2. Exponentenausrichtung
3. Addition der Mantissen
4. Normalisierung
5. Rundung

Beispiel

34 + (-7)

Addition von 34 und -7 in IEEE 754

0	1000 0100	000 1000 0000 0000 0000 0000	$(34)_D$
1	1000 0001	110 0000 0000 0000 0000 0000	$(-7)_D$

1. Extraktion der Komponenten

0	1000 0100	1,000 1000 0000 0000 0000 0000
1	1000 0001	1,110 0000 0000 0000 0000 0000

Mantissen mit Hidden-Bit!

Beispiel

34 + (-7)

Addition von 34 und -7 in IEEE 754

0	1000 0100	000 1000 0000 0000 0000 0000	$(34)_D$
1	1000 0001	110 0000 0000 0000 0000 0000	$(-7)_D$

2. Exponentenausrichtung

Exponenten müssen immer identisch sein. Allerdings…

1000 0100 $= (132)_D \neq (129)_D =$ 1000 0001

Immer kleineren Exponenten an größeren anpassen.

Beispiel

34 + (-7)

Addition von 34 und -7 in IEEE 754

0	1000 0100	000 1000 0000 0000 0000 0000	$(34)_D$
1	1000 0001	110 0000 0000 0000 0000 0000	$(-7)_D$

2. Exponentenausrichtung

$(34)_D = (100010,0)_B = (1,000100)_B * 2^5$

$(7)_D = (111,0)_B = (1,11)_B * 2^2$

(echte) Exponenten unterscheiden sich um 3

Immer kleineren Exponenten an größeren anpassen.

Beispiel

34 + (-7)

Addition von 34 und -7 in IEEE 754

0	1000 0100	000 1000 0000 0000 0000 0000	$(34)_D$
1	1000 0001	110 0000 0000 0000 0000 0000	$(-7)_D$

2. Exponentenausrichtung

$(34)_D = (100010,0)_B = (1,000100)_B * 2^5$

$(7)_D = (111,0)_B = (0,00111)_B * 2^5$

(echte) Exponenten unterscheiden sich um 0

Immer kleineren Exponenten an größeren anpassen.

Beispiel

34 + (-7)

Addition von 34 und -7 in IEEE 754

0	1000 0100	000 1000 0000 0000 0000 0000	$(34)_D$
1	1000 0001	110 0000 0000 0000 0000 0000	$(-7)_D$

3. Addition der Mantissen

	1	,	0	0	0	1	0
-	0	,	0	0	1	1	1
Ü	1		1	1	1	1	
	0	,	1	1	0	1	1

Beispiel

34 + (-7)

Addition von 34 und -7 in IEEE 754

0	1000 0100	000 1000 0000 0000 0000 0000	$(34)_D$
1	1000 0001	110 0000 0000 0000 0000 0000	$(-7)_D$

4. Normalisierung

Das Ergebnis muss in der Form 1, xxx... vorliegen.

In unserem Fall trifft das nicht zu, deshalb:

$0,\ 1\ 1\ 0\ 1\ 1 * 2^5 = 1,\ 1\ 0\ 1\ 1 * 2^4$

Beispiel

34 + (-7)

Addition von 34 und -7 in IEEE 754

0	1000 0100	000 1000 0000 0000 0000 0000	$(34)_D$
1	1000 0001	110 0000 0000 0000 0000 0000	$(-7)_D$

5. Rundung

Es muss gerundet werden, wenn Summe der Mantissen 23-Bit-Bereich verlässt.

In unserem Fall passiert das nicht.

Beispiel

34 + (-7)

Addition von 34 und -7 in IEEE 754

0	1000 0100	000 1000 0000 0000 0000 0000	$(34)_D$
1	1000 0001	110 0000 0000 0000 0000 0000	$(-7)_D$

6. Zusammensetzen

0	1000 0011	101 1000 0000 0000 0000 0000	$(27)_D$

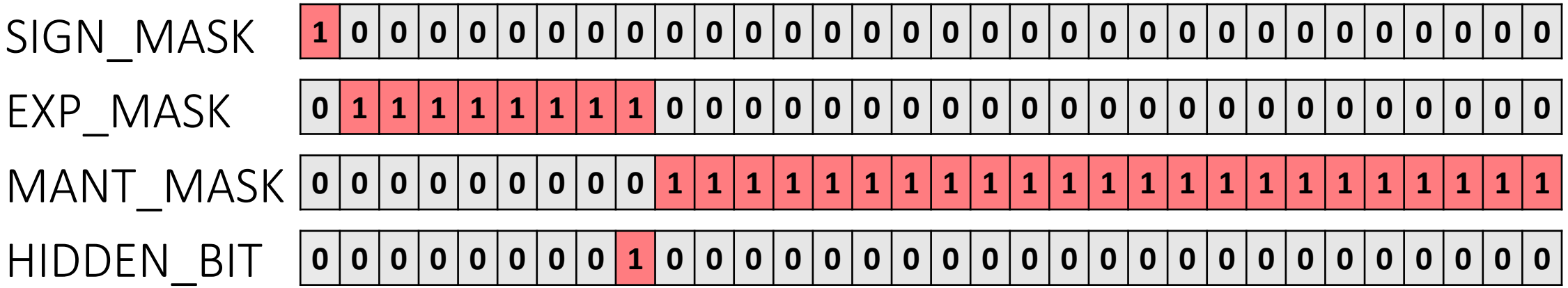
Addition zweier Fließkommazahlen - Code

Code

```
138     // a) Extraktion von Vorzeichen und Exponenten
139     sign_a = a & SIGN_MASK;
140     sign_b = b & SIGN_MASK;
141     exp_a = (a & EXP_MASK) >> 23;
142     exp_b = (b & EXP_MASK) >> 23;
143     mant_a = ((a & MANT_MASK) | HIDDEN_BIT) << 3;
144     mant_b = ((b & MANT_MASK) | HIDDEN_BIT) << 3;
```

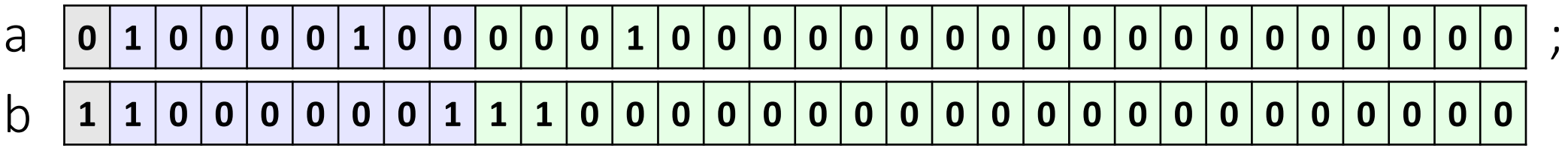
Code

```
138     // a) Extraktion von Vorzeichen und Exponenten
139     sign_a = a & SIGN_MASK;
140     sign_b = b & SIGN_MASK;
141     exp_a = (a & EXP_MASK) >> 23;
142     exp_b = (b & EXP_MASK) >> 23;
143     mant_a = ((a & MANT_MASK) | HIDDEN_BIT) << 3;
144     mant_b = ((b & MANT_MASK) | HIDDEN_BIT) << 3;
```

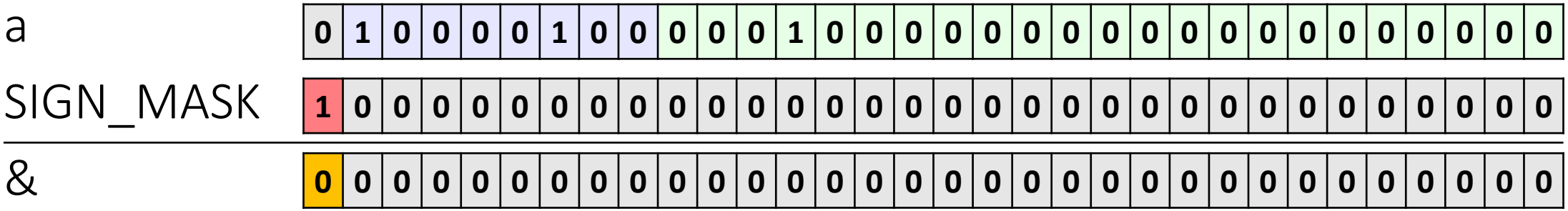


Code

```
138     // a) Extraktion von Vorzeichen und Exponenten
139     sign_a = a & SIGN_MASK;
140     sign_b = b & SIGN_MASK;
141     exp_a = (a & EXP_MASK) >> 23;
142     exp_b = (b & EXP_MASK) >> 23;
143     mant_a = ((a & MANT_MASK) | HIDDEN_BIT) << 3;
144     mant_b = ((b & MANT_MASK) | HIDDEN_BIT) << 3;
```

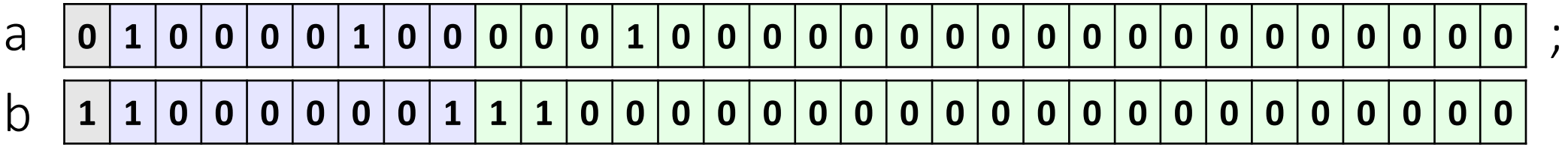


Durchlauf beispielhaft für a

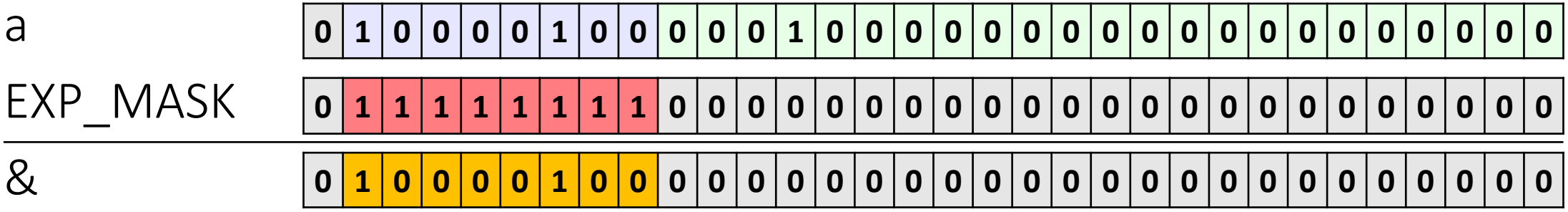


Code

```
138     // a) Extraktion von Vorzeichen und Exponenten
139     sign_a = a & SIGN_MASK;
140     sign_b = b & SIGN_MASK;
141     exp_a = (a & EXP_MASK) >> 23;
142     exp_b = (b & EXP_MASK) >> 23;
143     mant_a = ((a & MANT_MASK) | HIDDEN_BIT) << 3;
144     mant_b = ((b & MANT_MASK) | HIDDEN_BIT) << 3;
```

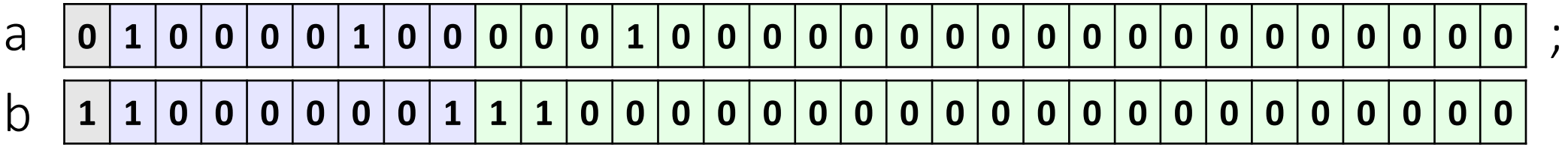


Durchlauf beispielhaft für a

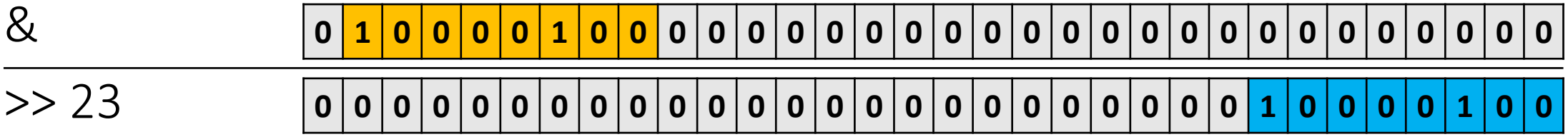


Code

```
138     // a) Extraktion von Vorzeichen und Exponenten
139     sign_a = a & SIGN_MASK;
140     sign_b = b & SIGN_MASK;
141     exp_a = (a & EXP_MASK) >> 23;
142     exp_b = (b & EXP_MASK) >> 23;
143     mant_a = ((a & MANT_MASK) | HIDDEN_BIT) << 3;
144     mant_b = ((b & MANT_MASK) | HIDDEN_BIT) << 3;
```

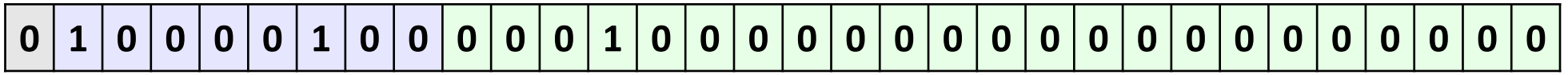


Durchlauf beispielhaft für a

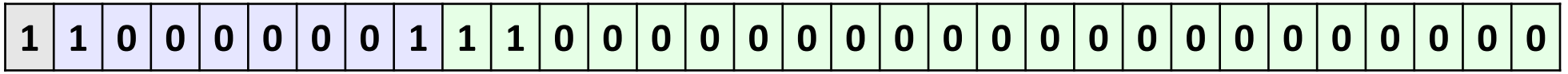


Code

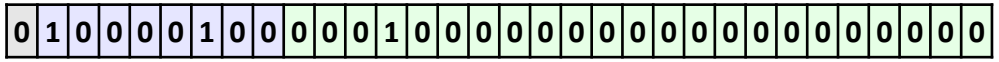
```
138     // a) Extraktion von Vorzeichen und Exponenten
139     sign_a = a & SIGN_MASK;
140     sign_b = b & SIGN_MASK;
141     exp_a = (a & EXP_MASK) >> 23;
142     exp_b = (b & EXP_MASK) >> 23;
143     mant_a = ((a & MANT_MASK) | HIDDEN_BIT) << 3;
144     mant_b = ((b & MANT_MASK) | HIDDEN_BIT) << 3;
```

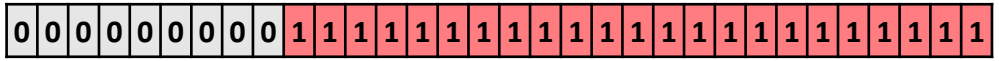
a 

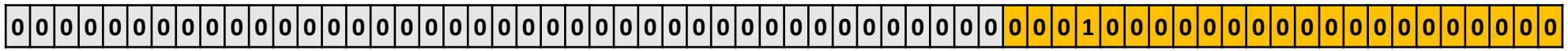
;

b 

Durchlauf beispielhaft für a

a 

MANT_MASK 

& 

Code

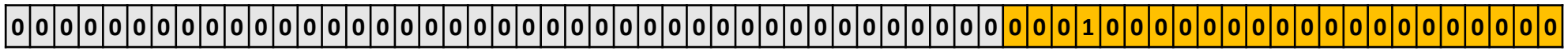
```
138     // a) Extraktion von Vorzeichen und Exponenten
139     sign_a = a & SIGN_MASK;
140     sign_b = b & SIGN_MASK;
141     exp_a = (a & EXP_MASK) >> 23;
142     exp_b = (b & EXP_MASK) >> 23;
143     mant_a = ((a & MANT_MASK) | HIDDEN_BIT) << 3;
144     mant_b = ((b & MANT_MASK) | HIDDEN_BIT) << 3;
```

a 0 1 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ;

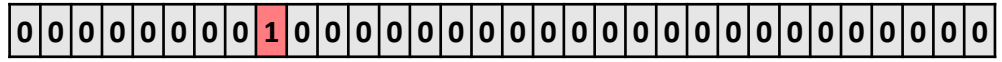
b 1 1 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Durchlauf beispielhaft für a

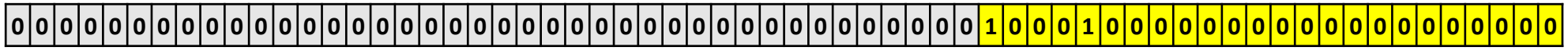
&



HIDDEN_BIT



|



Code

```
138     // a) Extraktion von Vorzeichen und Exponenten
139     sign_a = a & SIGN_MASK;
140     sign_b = b & SIGN_MASK;
141     exp_a = (a & EXP_MASK) >> 23;
142     exp_b = (b & EXP_MASK) >> 23;
143     mant_a = ((a & MANT_MASK) | HIDDEN_BIT) << 3;
144     mant_b = ((b & MANT_MASK) | HIDDEN_BIT) << 3;
```

a 0 1 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ;

b 1 1 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Durchlauf beispielhaft für a

| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

<< 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Code

```
146     // Sortieren: A soll der Betragsgrossere sein
147     if (exp_b > exp_a || (exp_b == exp_a && mant_b > mant_a)) {
148         uint32_t ts = sign_a; sign_a = sign_b; sign_b = ts;
149         int32_t te = exp_a; exp_a = exp_b; exp_b = te;
150         uint64_t tm = mant_a; mant_a = mant_b; mant_b = tm;
151     }
```

Der Code tauscht die Operanden a und b, falls |B| > |A|,
um sicherzustellen, dass a stets der betragsmäßig Größere ist.

Warum ist das wichtig?

Indem wir sicherstellen, dass wir immer "Große Zahl minus Kleine Zahl" rechnen, sparen wir uns komplizierte Fallunterscheidungen für negative Zwischenergebnisse und wechselnde Vorzeichen.

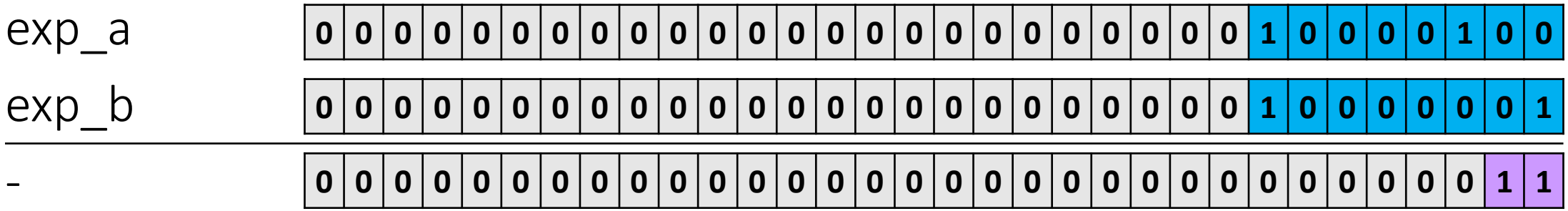
Addition zweier Fließkommazahlen - Code

Code

```
153     // b) Angleichen
154     int shift = exp_a - exp_b;
155     if (shift > 60) {
156         mant_b = 0; // Zu klein
157     } else if (shift > 0) {
158         remainder = mant_b & ((1ULL << shift) - 1);
159         mant_b >>= shift;
160         if (remainder != 0) mant_b |= 1; // Sticky bit setzen
161     }
162     res_exp = exp_a;
163     res_sign = sign_a;
```

Code

```
153     // b) Angleichen
154     int shift = exp_a - exp_b;
155     if (shift > 60) {
156         mant_b = 0; // Zu klein
157     } else if (shift > 0) {
158         remainder = mant_b & ((1ULL << shift) - 1);
159         mant_b >>= shift;
160         if (remainder != 0) mant_b |= 1; // Sticky bit setzen
161     }
162     res_exp = exp_a;
163     res_sign = sign_a;
```



Berechnet Abstand zwischen Exponenten der beiden Zahlen.

Code

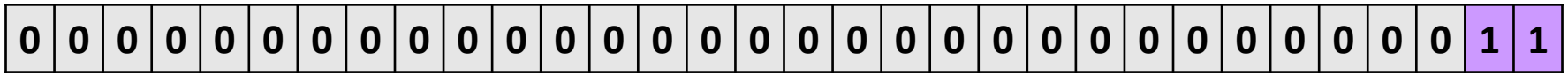
```
153     // b) Angleichen
154     int shift = exp_a - exp_b;
155     if (shift > 60) {
156         mant_b = 0; // Zu klein
157     } else if (shift > 0) {
158         remainder = mant_b & ((1ULL << shift) - 1);
159         mant_b >>= shift;
160         if (remainder != 0) mant_b |= 1; // Sticky bit setzen
161     }
162     res_exp = exp_a;
163     res_sign = sign_a;
```

Wenn der Unterschied zwischen den Exponenten extrem groß ist, ist die Zahl B so winzig im Vergleich zu A, dass sie nach dem Verschieben komplett verschwindet.
Statt die Bits zu schieben wird Rechenzeit gespart, indem mant_b = 0.

Code

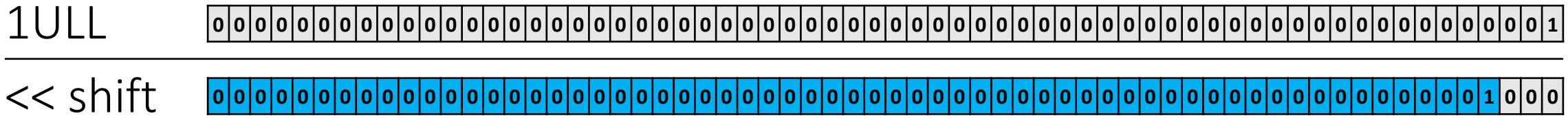
```
153     // b) Angleichen
154     int shift = exp_a - exp_b;
155     if (shift > 60) {
156         mant_b = 0; // Zu klein
157     } else if (shift > 0) {
158         remainder = mant_b & ((1ULL << shift) - 1);
159         mant_b >>= shift;
160         if (remainder != 0) mant_b |= 1; // Sticky bit setzen
161     }
162     res_exp = exp_a;
163     res_sign = sign_a;
```

shift



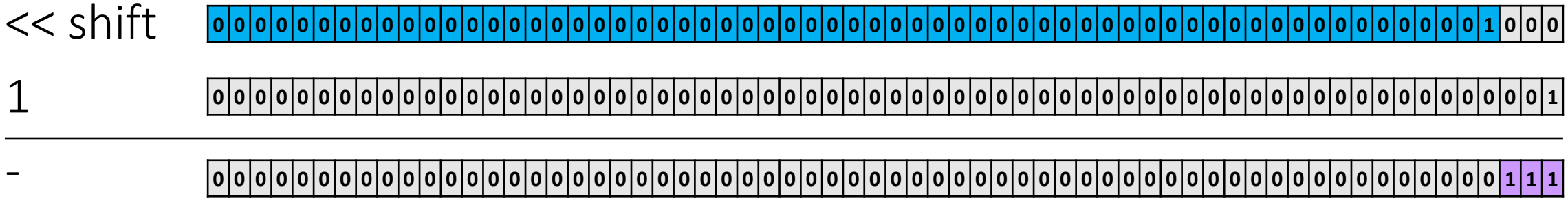
Code

```
153     // b) Angleichen
154     int shift = exp_a - exp_b;
155     if (shift > 60) {
156         mant_b = 0; // Zu klein
157     } else if (shift > 0) {
158         remainder = mant_b & ((1ULL << shift) - 1);
159         mant_b >>= shift;
160         if (remainder != 0) mant_b |= 1; // Sticky bit setzen
161     }
162     res_exp = exp_a;
163     res_sign = sign_a;
```



Code

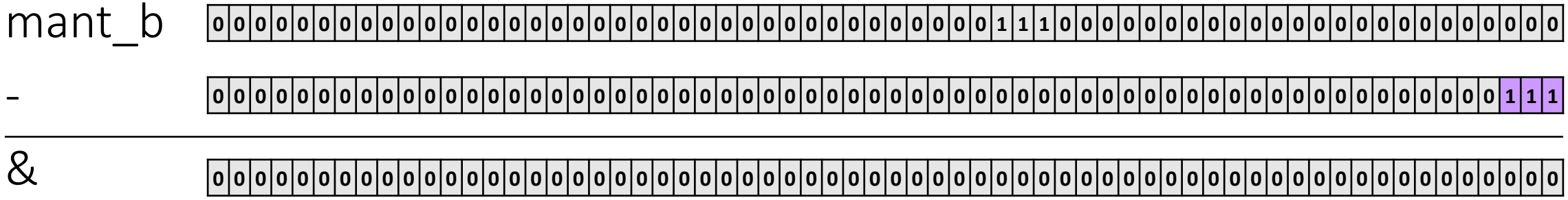
```
153     // b) Angleichen
154     int shift = exp_a - exp_b;
155     if (shift > 60) {
156         mant_b = 0; // Zu klein
157     } else if (shift > 0) {
158         remainder = mant_b & ((1ULL << shift) - 1);
159         mant_b >>= shift;
160         if (remainder != 0) mant_b |= 1; // Sticky bit setzen
161     }
162     res_exp = exp_a;
163     res_sign = sign_a;
```



Isoliert Bits, die beim folgenden Rechts-Shift verloren gehen würden.

Code

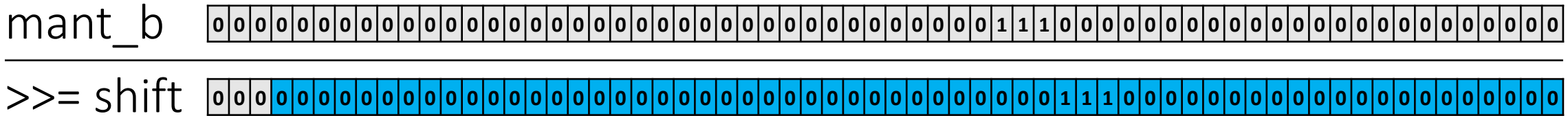
```
153     // b) Angleichen
154     int shift = exp_a - exp_b;
155     if (shift > 60) {
156         mant_b = 0; // Zu klein
157     } else if (shift > 0) {
158         remainder = mant_b & ((1ULL << shift) - 1);
159         mant_b >>= shift;
160         if (remainder != 0) mant_b |= 1; // Sticky bit setzen
161     }
162     res_exp = exp_a;
163     res_sign = sign_a;
```



Isoliert Bits, die beim folgenden Rechts-Shift verloren gehen würden.

Code

```
153     // b) Angleichen
154     int shift = exp_a - exp_b;
155     if (shift > 60) {
156         mant_b = 0; // Zu klein
157     } else if (shift > 0) {
158         remainder = mant_b & ((1ULL << shift) - 1);
159         mant_b >>= shift;
160         if (remainder != 0) mant_b |= 1; // Sticky bit setzen
161     }
162     res_exp = exp_a;
163     res_sign = sign_a;
```



Da der Exponent von b kleiner ist, verschieben wir seine Mantisse nach rechts, um sie an die Basis von a anzupassen.

Code

```
153     // b) Angleichen
154     int shift = exp_a - exp_b;
155     if (shift > 60) {
156         mant_b = 0; // Zu klein
157     } else if (shift > 0) {
158         remainder = mant_b & ((1ULL << shift) - 1);
159         mant_b >>= shift;
160         if (remainder != 0) mant_b |= 1; // Sticky bit setzen
161     }
162     res_exp = exp_a;
163     res_sign = sign_a;
```

remaind. 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Setzt das "Sticky-Bit" am Ende der Mantisse, falls beim Verschieben wertvolle Bits verloren gegangen sind, damit später korrekt gerundet werden kann.

Code

```
153     // b) Angleichen
154     int shift = exp_a - exp_b;
155     if (shift > 60) {
156         mant_b = 0; // Zu klein
157     } else if (shift > 0) {
158         remainder = mant_b & ((1ULL << shift) - 1);
159         mant_b >>= shift;
160         if (remainder != 0) mant_b |= 1; // Sticky bit setzen
161     }
162     res_exp = exp_a;
163     res_sign = sign_a;
```

Das vorläufige Ergebnis bekommt Vorzeichen und Exponent des betragsmäßig größeren Operanden A.

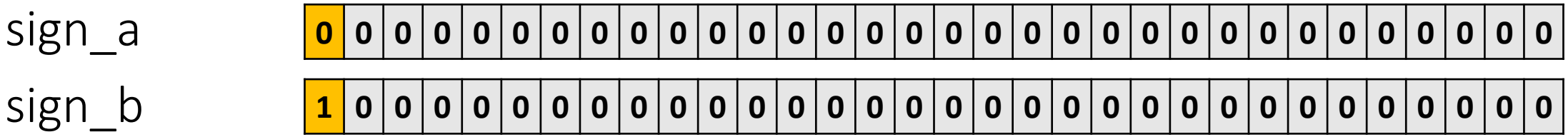
Code

```
166     if (sign_a == sign_b) {
167         // c) Addition durchfuehren
168         res_mant = mant_a + mant_b;
169         // Ueberlauf addieren? (Bit 27 gesetzt, da wir um 3 erweitert haben 23+1+3 = 27)
170         if (res_mant & (1ULL << 27)) {
171             // Sticky Bit bewahren beim Rechts-Shift
172             if (res_mant & 1) res_mant |= 2; // Sticky weitergeben
173              res_mant >>= 1;
174              res_exp++;
175          }
176     ...
```

Der if-Block implementiert die Addition der Mantissen bei identischen Vorzeichen der Operanden.
Falls durch die Addition ein Überlauf in Bit 27 entsteht, wird die Mantisse erneut normalisiert und der Exponent inkrementiert.

Code

```
166     if (sign_a == sign_b) {
167         // c) Addition durchfuehren
168         res_mant = mant_a + mant_b;
169         // Ueberlauf addieren? (Bit 27 gesetzt, da wir um 3 erweitert haben 23+1+3 = 27)
170         if (res_mant & (1ULL << 27)) {
171             // Sticky Bit bewahren beim Rechts-Shift
172             if (res_mant & 1) res_mant |= 2; // Sticky weitergeben
173              res_mant >>= 1;
174              res_exp++;
175          }
176     ...
```



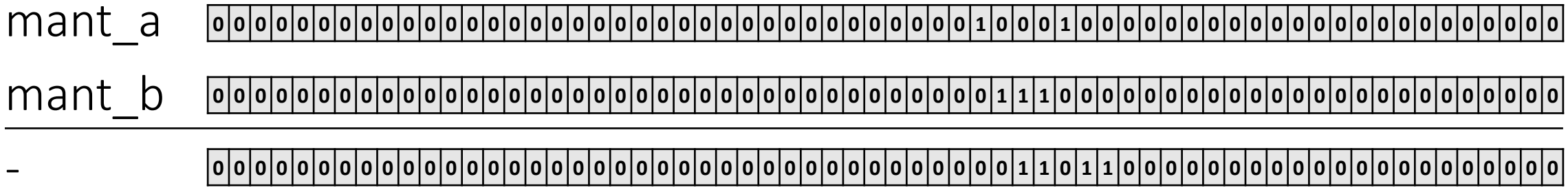
Addition zweier Fließkommazahlen - Code

Code

```
176     } else {
177         // d) Subtraktion (mant_a >= mant_b garantiert durch Sortierung)
178         res_mant = mant_a - mant_b;
179         if (res_mant == 0) return 0;
180     ...
```

Code

```
176     } else {
177         // d) Subtraktion (mant_a >= mant_b garantiert durch Sortierung)
178         res_mant = mant_a - mant_b;
179         if (res_mant == 0) return 0;
180     ...
```



Da die Operanden unterschiedliche Vorzeichen besitzen, berechnen wir die Differenz ihrer Beträge, wobei das Ergebnis aufgrund der vorherigen Sortierung |A| >= |B| stets nicht-negativ ist.

Code

```
176     } else {
177         // d) Subtraktion (mant_a >= mant_b garantiert durch Sortierung)
178         res_mant = mant_a - mant_b;
179         if (res_mant == 0) return 0;
180     ...
```

Falls das Ergebnis der Subtraktion exakt 0 ist, beenden wir die Funktion sofort und geben 0 zurück, da keine Normalisierung oder Rundung mehr nötig ist.

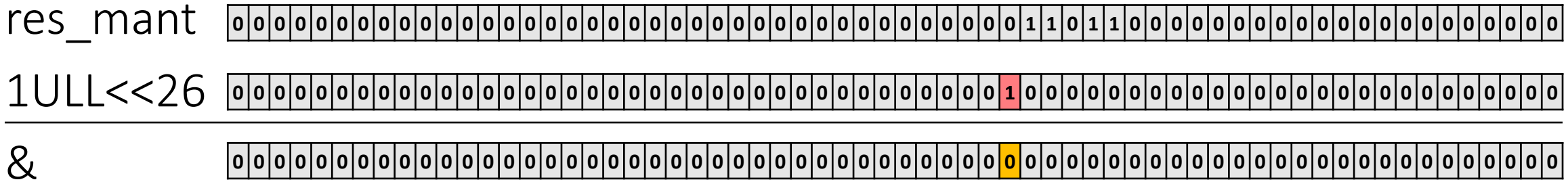
Code

```
181         // e) Normalisieren (Links Shift bis Bit 26 gesetzt ist)
182         while (!(res_mant & (1ULL << 26))) {
183             res_mant <<= 1;
184             res_exp--;
185             if (res_exp <= 0) return 0; // Underflow
186         }
187     }
```

Solange an der höchsten Position (Bit 26, unser "Start-Bit") noch eine 0 steht, ist die Zahl zu weit rechts.
Wir müssen sie also so lange nach links schieben, bis die erste 1 vorne anstößt. Erst dann ist sie gültig normalisiert.

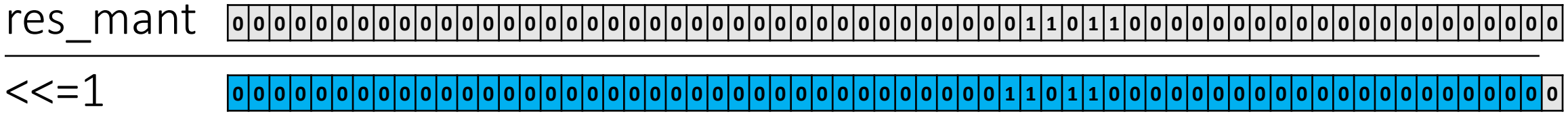
Code

```
181         // e) Normalisieren (Links Shift bis Bit 26 gesetzt ist)
182         while (!(res_mant & (1ULL << 26))) {
183             res_mant <<= 1;
184             res_exp--;
185             if (res_exp <= 0) return 0; // Underflow
186         }
187     }
```



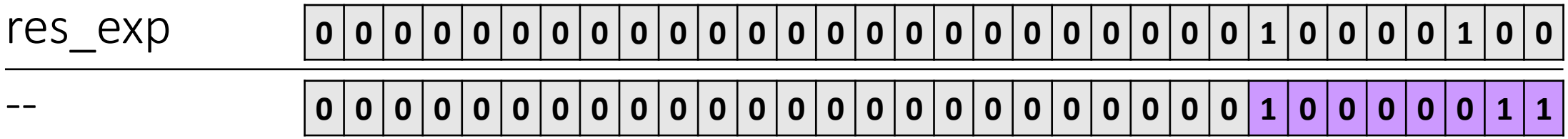
Code

```
181         // e) Normalisieren (Links Shift bis Bit 26 gesetzt ist)
182         while (!(res_mant & (1ULL << 26))) {
183             res_mant <<= 1;
184             res_exp--;
185             if (res_exp <= 0) return 0; // Underflow
186         }
187     }
```



Code

```
181         // e) Normalisieren (Links Shift bis Bit 26 gesetzt ist)
182         while (!(res_mant & (1ULL << 26))) {
183             res_mant <<= 1;
184             res_exp--;
185             if (res_exp <= 0) return 0; // Underflow
186         }
187     }
```



Code

```
181         // e) Normalisieren (Links Shift bis Bit 26 gesetzt ist)
182         while (!(res_mant & (1ULL << 26))) {
183             res_mant <<= 1;
184             res_exp--;
185             if (res_exp <= 0) return 0; // Underflow
186         }
187     }
```

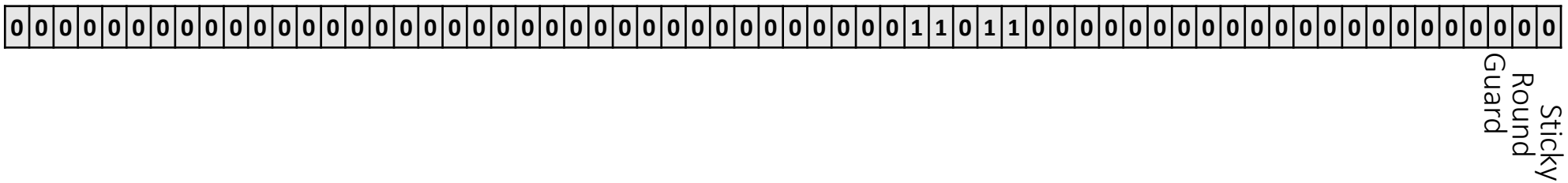
Arithmetischen Unterlauf (Underflow) abfangen, indem das Ergebnis auf Null setzt, sobald der Exponent durch die für die Normalisierung erforderliche Verringerung den zulässigen Wertebereich unterschreitet.

Code

```
189         // Die Mantisse ist aktuell um 3 Bits nach links verschoben.
190         // Bit 2: Guard, Bit 1: Round, Bit 0: Sticky.
191         round_bits = res_mant & 7; // Die unteren 3 Bits
192         res_mant >>= 3; // Zurück auf normale Position
193
194         if (round_bits > 4) { // > 100 (binaer) -> Aufrunden
195             res_mant++;
196         } else if (round_bits == 4) { // == 100 (binaer) -> Tie to even
197             if (res_mant & 1) res_mant++;
198         }
```

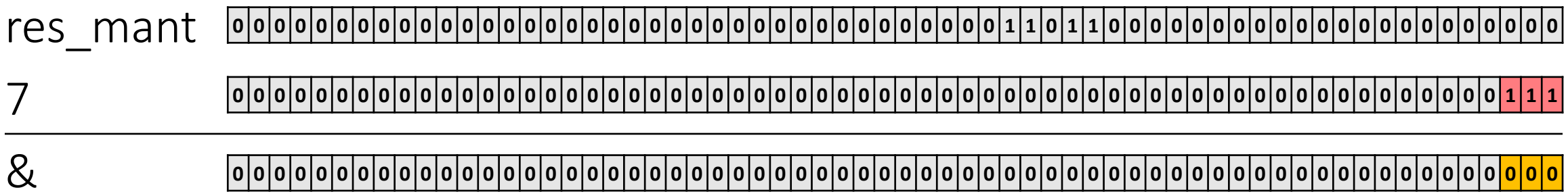
Wir entfernen die für die Rechengenauigkeit genutzten drei Zusatzbits wieder und verwenden sie zum Runden.

res_mant



```
189         // Die Mantisse ist aktuell um 3 Bits nach links verschoben.
190         // Bit 2: Guard, Bit 1: Round, Bit 0: Sticky.
191         round_bits = res_mant & 7; // Die unteren 3 Bits
192         res_mant >>= 3; // Zurück auf normale Position
193
194         if (round_bits > 4) { // > 100 (binaer) -> Aufrunden
195             res_mant++;
196         } else if (round_bits == 4) { // == 100 (binaer) -> Tie to even
197             if (res_mant & 1) res_mant++;
198         }
```

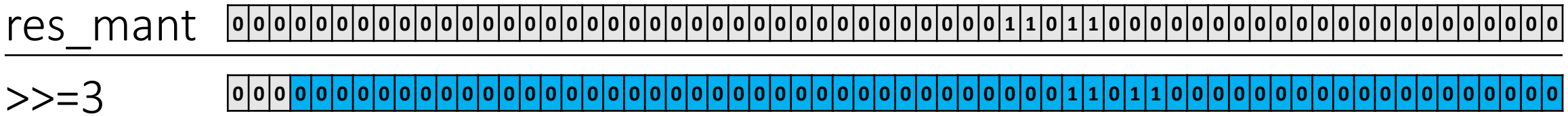
Wir entfernen die für die Rechengenauigkeit genutzten drei Zusatzbits wieder und verwenden sie zum Runden.



Code

```
189         // Die Mantisse ist aktuell um 3 Bits nach links verschoben.
190         // Bit 2: Guard, Bit 1: Round, Bit 0: Sticky.
191         round_bits = res_mant & 7; // Die unteren 3 Bits
192         res_mant >>= 3; // Zurück auf normale Position
193
194         if (round_bits > 4) { // > 100 (binaer) -> Aufrunden
195             res_mant++;
196         } else if (round_bits == 4) { // == 100 (binaer) -> Tie to even
197             if (res_mant & 1) res_mant++;
198         }
```

Wir entfernen die für die Rechengenauigkeit genutzten drei Zusatzbits wieder und verwenden sie zum Runden.



Code

```
189         // Die Mantisse ist aktuell um 3 Bits nach links verschoben.
190         // Bit 2: Guard, Bit 1: Round, Bit 0: Sticky.
191         round_bits = res_mant & 7; // Die unteren 3 Bits
192         res_mant >>= 3; // Zurück auf normale Position
193
194         if (round_bits > 4) { // > 100 (binaer) -> Aufrunden
195             res_mant++;
196         } else if (round_bits == 4) { // == 100 (binaer) -> Tie to even
197             if (res_mant & 1) res_mant++;
198         }
```

Entscheidet anhand der abgeschnittenen Nachkommastellen, ob die Mantisse aufgerundet werden muss, wobei exakt halbe Werte zur nächsten geraden Zahl gerundet werden.

Hier: Aufgerundet (Mantisse + 1), da der Rest größer als 0,5 ist.

Code

```
189         // Die Mantisse ist aktuell um 3 Bits nach links verschoben.
190         // Bit 2: Guard, Bit 1: Round, Bit 0: Sticky.
191         round_bits = res_mant & 7; // Die unteren 3 Bits
192         res_mant >>= 3; // Zurück auf normale Position
193
194         if (round_bits > 4) { // > 100 (binaer) -> Aufrunden
195             res_mant++;
196         } else if (round_bits == 4) { // == 100 (binaer) -> Tie to even
197             if (res_mant & 1) res_mant++;
198         }
```

Entscheidet anhand der abgeschnittenen Nachkommastellen, ob die Mantisse aufgerundet werden muss, wobei exakt halbe Werte zur nächsten geraden Zahl gerundet werden.

Hier: Aufgerundet, wenn die Mantisse aktuell ungerade ist.

- *Endet die Mantisse auf ...0: Zahl ist gerade.*
- *Endet die Mantisse auf ...1: Zahl ist ungerade.*

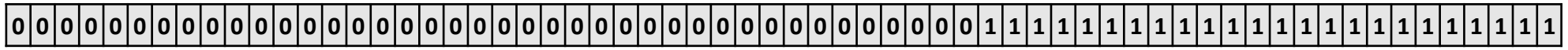
Code

```
200         // Nach Rundung kann die Mantisse ueberlaufen (hidden bit pruefen)
201         if (res_mant & (1ULL << 24)) {
202             res_mant >>= 1;
203             res_exp++;
204         }
```

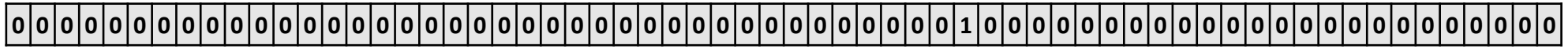
Korrigiert einen Überlauf, der durch das Aufrunden entstanden sein kann.

Beispiel:

Vorher



Nach Rundung



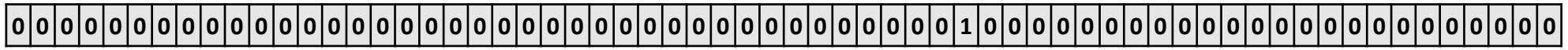
Code

```
200         // Nach Rundung kann die Mantisse ueberlaufen (hidden bit pruefen)
201         if (res_mant & (1ULL << 24)) {
202             res_mant >>= 1;
203             res_exp++;
204         }
```

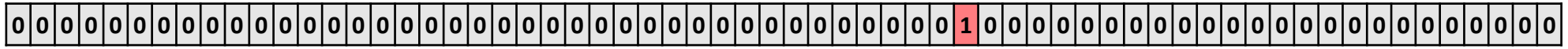
Korrigiert einen Überlauf, der durch das Aufrunden entstanden sein kann.

Beispiel:

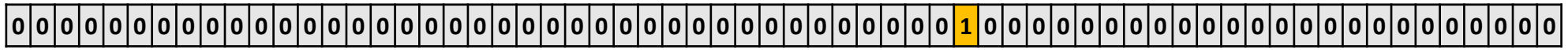
Nach Rundung



1ULL<<24



&



Implementieren Sie in kahan.c die aufsteigende, absteigende und Kahan-Summation, um numerische Stabilität zu demonstrieren.

Analysieren Sie anschließend in task2.txt die auftretenden Rundungsfehler bei wachsendem N sowie das historische Patriot-Raketen-Beispiel.

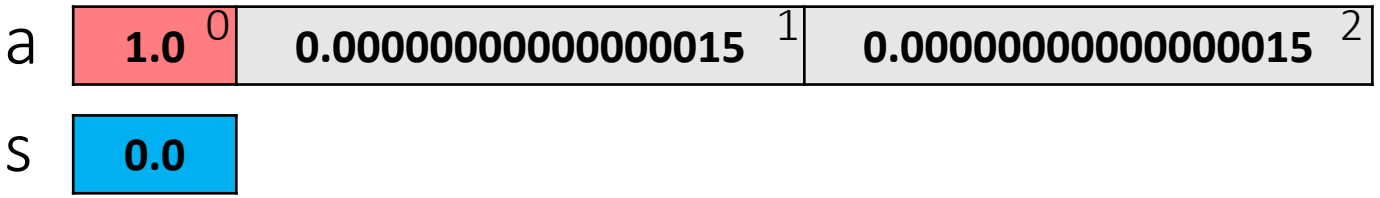
Kahan-Summe und Stabilität - Code

Code

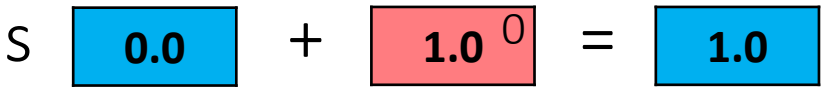
```
19      static double sum_ascending(double* a, size_t N) {
20          double s = 0.0;
21          for (size_t i = 0; i < N; i++) s += a[i];
22          return s;
23      }
```

Code

```
19      static double sum_ascending(double* a, size_t N) {
20          double s = 0.0;
21          for (size_t i = 0; i < N; i++) s += a[i];
22          return s;
23      }
```

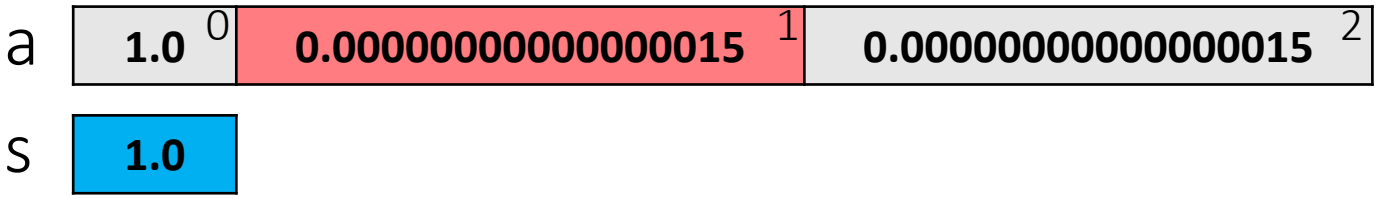


Iteration i = 0

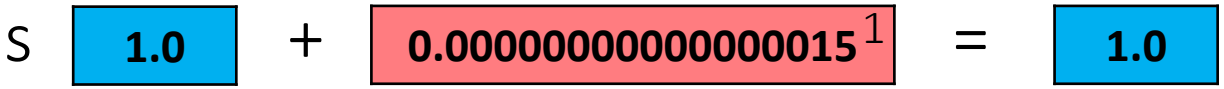


Code

```
19      static double sum_ascending(double* a, size_t N) {
20          double s = 0.0;
21          for (size_t i = 0; i < N; i++) s += a[i];
22          return s;
23      }
```



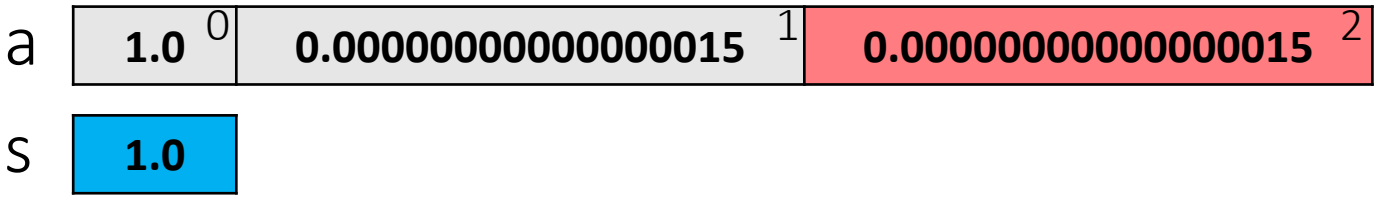
Iteration i = 1



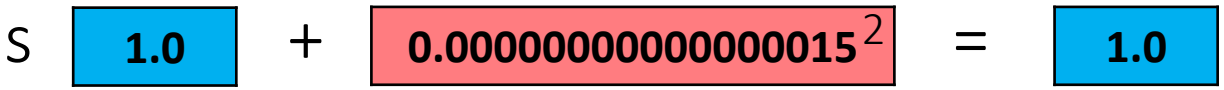
Beim Angleichen der Exponenten werden die signifikanten Bits der kleineren Zahl rechts aus der Mantisse hinausgeschoben.

Code

```
19      static double sum_ascending(double* a, size_t N) {
20          double s = 0.0;
21          for (size_t i = 0; i < N; i++) s += a[i];
22          return s;
23      }
```



Iteration i = 2



Beim Angleichen der Exponenten werden die signifikanten Bits der kleineren Zahl rechts aus der Mantisse hinausgeschoben.

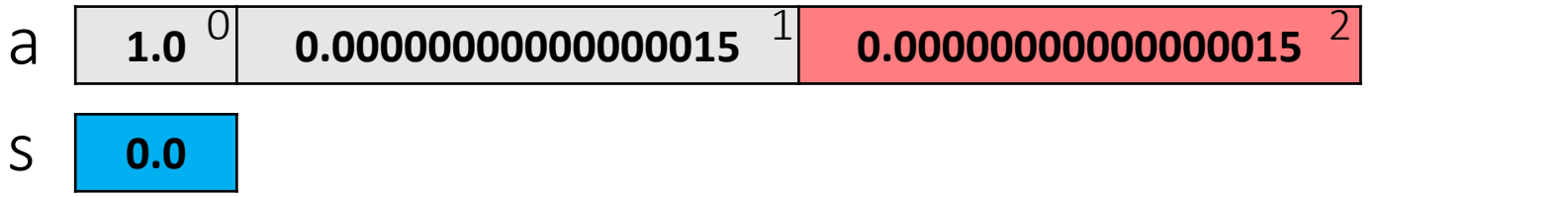
Kahan-Summe und Stabilität – Code (Absteigende Summation)

Code

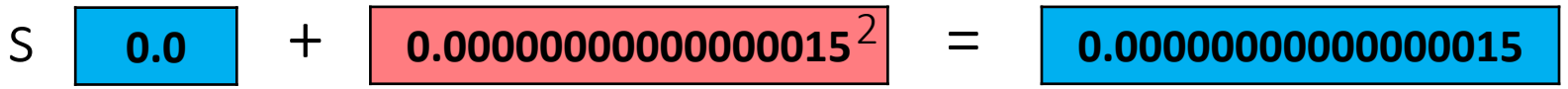
```
25      static double sum_descending(double* a, size_t N) {
26          double s = 0.0;
27          for (size_t i = N; i-- > 0; ) s += a[i];
28          return s;
29      }
```

Code

```
25      static double sum_descending(double* a, size_t N) {
26          double s = 0.0;
27          for (size_t i = N; i-- > 0; ) s += a[i];
28          return s;
29      }
```

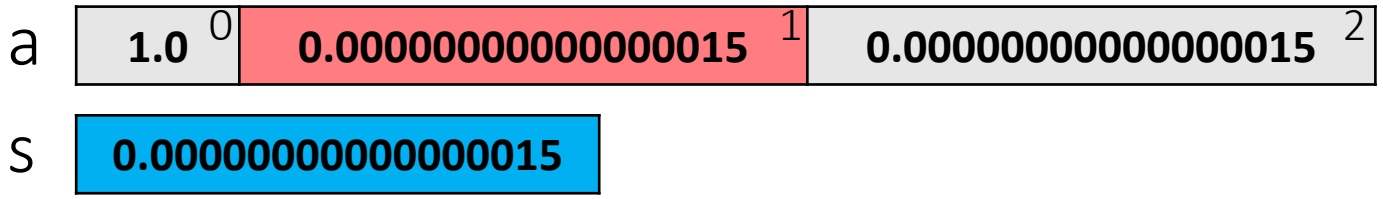


Iteration i = 2

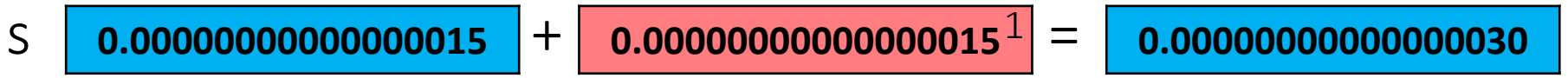


Code

```
25      static double sum_descending(double* a, size_t N) {
26          double s = 0.0;
27          for (size_t i = N; i-- > 0; ) s += a[i];
28          return s;
29      }
```



Iteration i = 1



Beide Zahlen haben denselben Exponenten.
Es muss nichts geschoben werden.

Code

```
25      static double sum_descending(double* a, size_t N) {
26          double s = 0.0;
27          for (size_t i = N; i-- > 0; ) s += a[i];
28          return s;
29      }
```

a | **1.0** 0 | **0.0000000000000015** 1 | **0.0000000000000015** 2

s | **0.0000000000000030**

Iteration i = 0

s **0.0000000000000030** + **1.0** 0 = **1.0000000000000030**

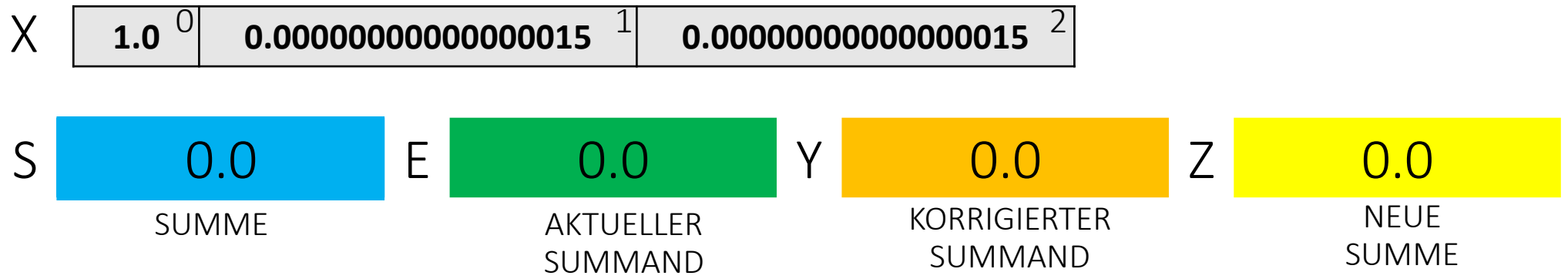
Beim Angleichen der Exponenten fallen die letzten Bits dieses Mal nicht raus!

Code

```
31      static double kahan(double* X, size_t N) {
32          double S = 0.0, E = 0.0, Y = 0.0, Z = 0.0;
33          size_t i;
34          for (i = 0; i < N; i++) {
35              Y = X[i] - E;
36              Z = S + Y;
37              E = (Z - S) - Y;
38              S = Z;
39          }
40          return S;
41      }
```

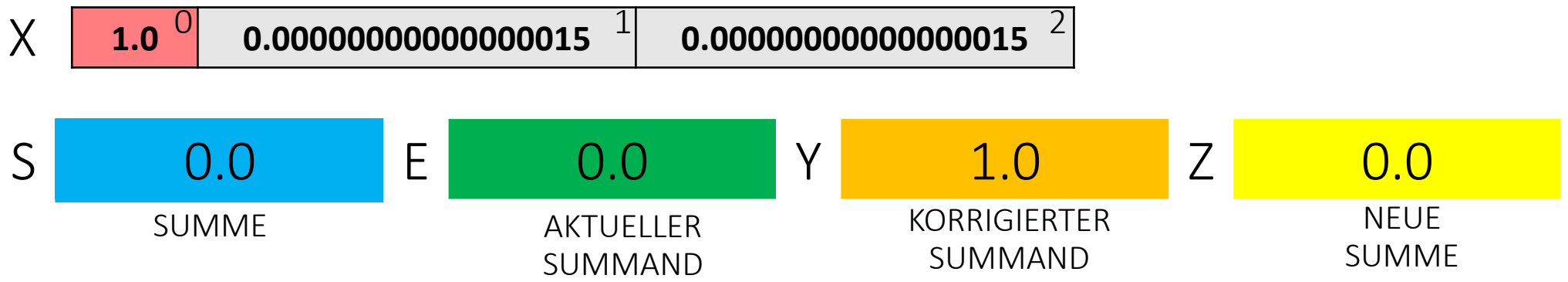
Code

```
31      static double kahan(double* X, size_t N) {
32          double S = 0.0, E = 0.0, Y = 0.0, Z = 0.0;
33          size_t i;
34          for (i = 0; i < N; i++) {
35              Y = X[i] - E;
36              Z = S + Y;
37              E = (Z - S) - Y;
38              S = Z;
39          }
40          return S;
41      }
```



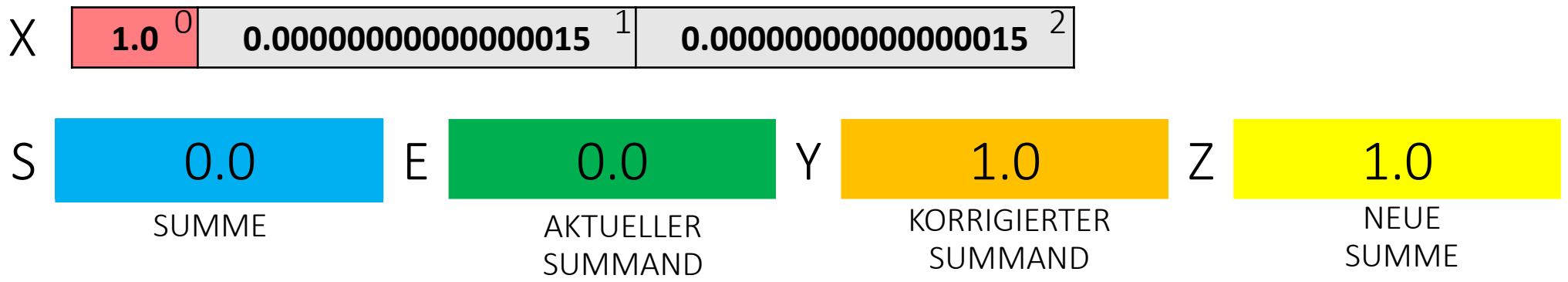
Code

```
31      static double kahan(double* X, size_t N) {
        ...
34          for (i = 0; i < N; i++) {
35              Y = X[i] - E;
36              Z = S + Y;
37              E = (Z - S) - Y;
38              S = Z;
39          }
```



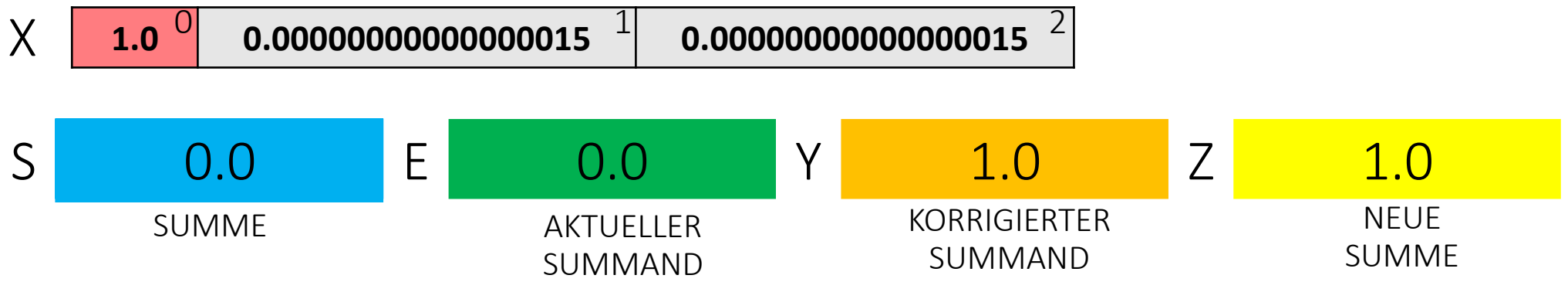
Code

```
31      static double kahan(double* X, size_t N) {
        ...
34          for (i = 0; i < N; i++) {
35              Y = X[i] - E;
36              Z = S + Y;
37              E = (Z - S) - Y;
38              S = Z;
39          }
```



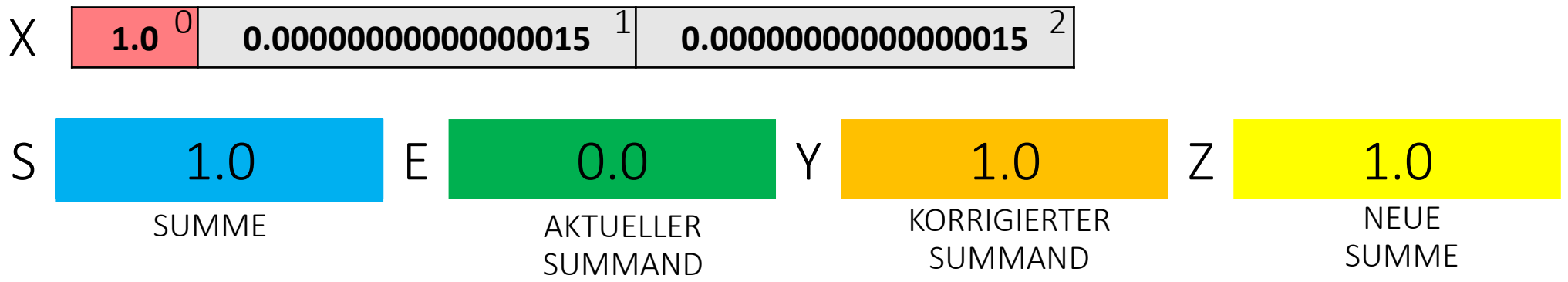
Code

```
31      static double kahan(double* X, size_t N) {
        ...
34          for (i = 0; i < N; i++) {
35              Y = X[i] - E;
36              Z = S + Y;
37              E = (Z - S) - Y;
38              S = Z;
39          }
```



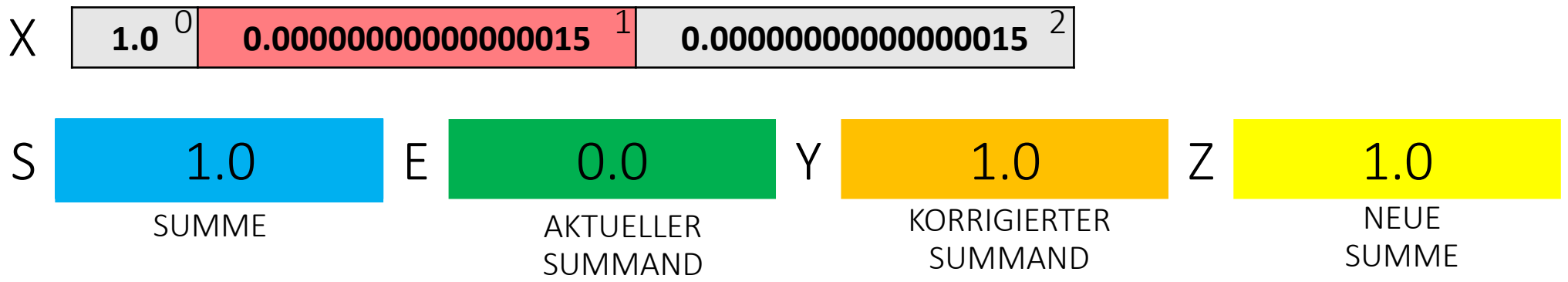
Code

```
31      static double kahan(double* X, size_t N) {
        ...
34          for (i = 0; i < N; i++) {
35              Y = X[i] - E;
36              Z = S + Y;
37              E = (Z - S) - Y;
38              S = Z;
39          }
```



Code

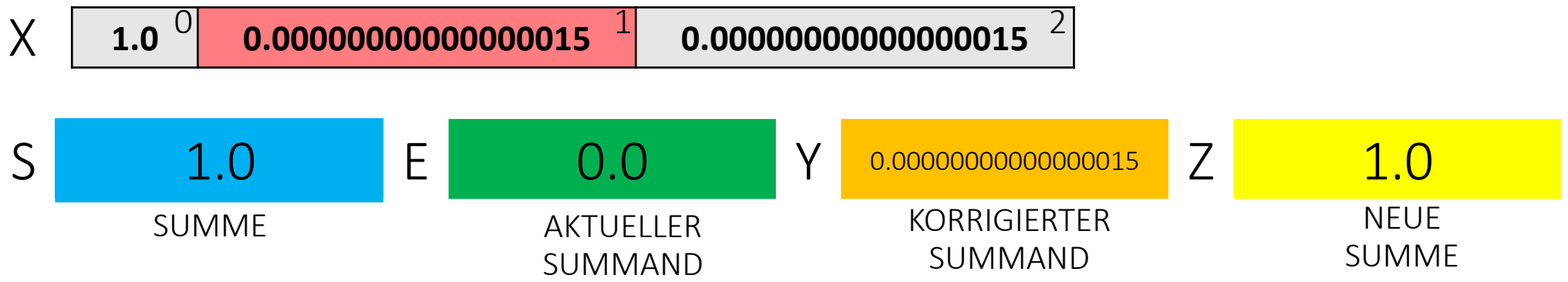
```
31      static double kahan(double* X, size_t N) {
        ...
34          for (i = 0; i < N; i++) {
35              Y = X[i] - E;
36              Z = S + Y;
37              E = (Z - S) - Y;
38              S = Z;
39          }
```



Aufgabe 2

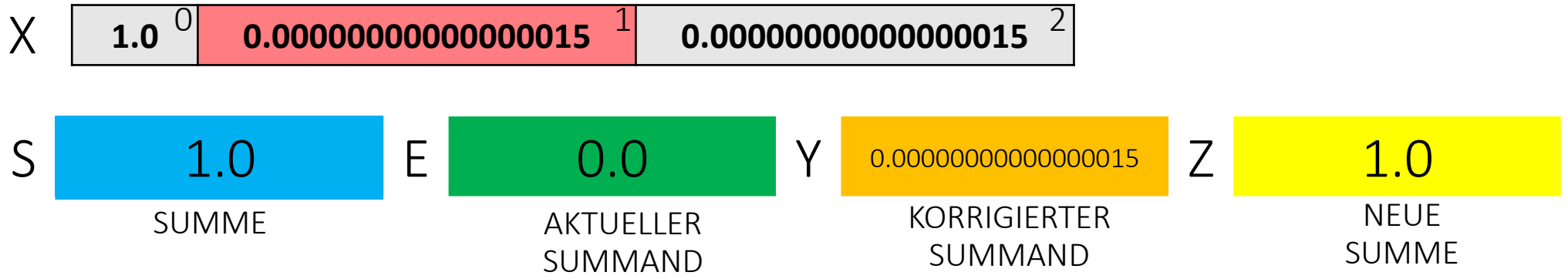
Code

```
31      static double kahan(double* X, size_t N) {
        ...
34          for (i = 0; i < N; i++) {
35              Y = X[i] - E;
36              Z = S + Y;
37              E = (Z - S) - Y;
38              S = Z;
39          }
```



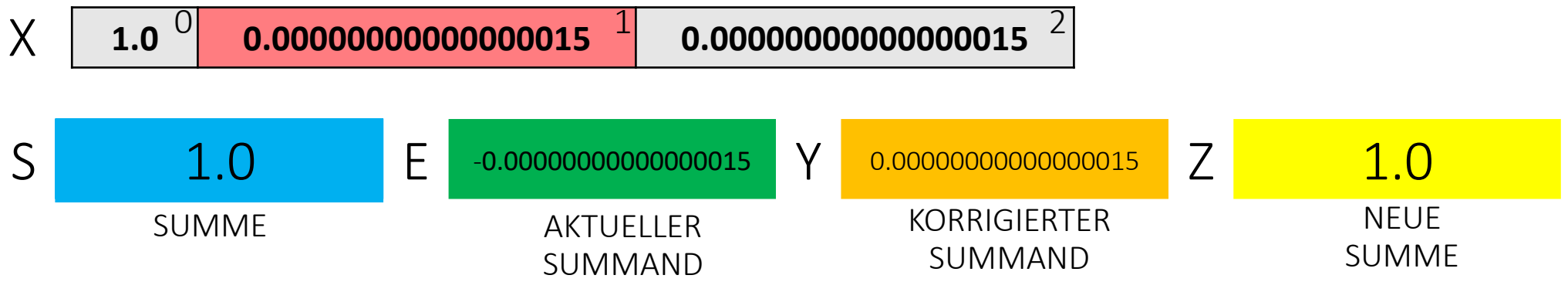
Code

```
31      static double kahan(double* X, size_t N) {
        ...
34          for (i = 0; i < N; i++) {
35              Y = X[i] - E;
36              Z = S + Y;
37              E = (Z - S) - Y;
38              S = Z;
39          }
```



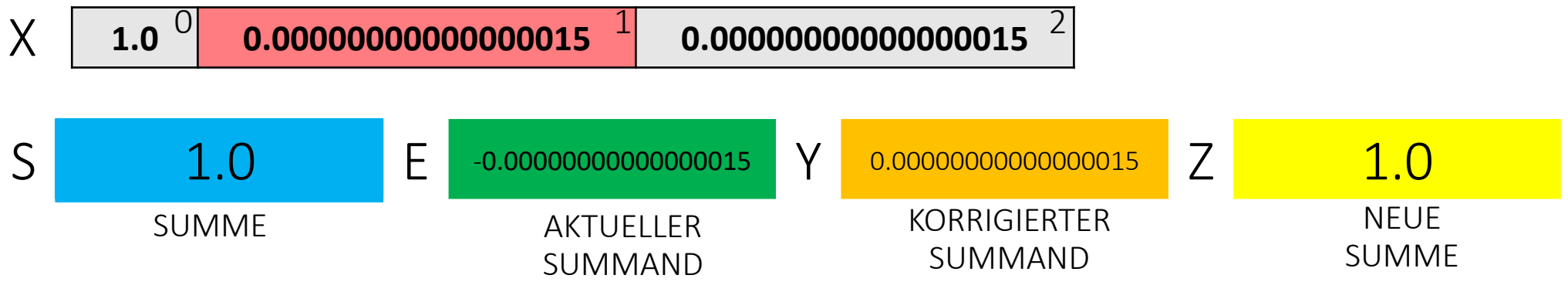
Code

```
31      static double kahan(double* X, size_t N) {
        ...
34          for (i = 0; i < N; i++) {
35              Y = X[i] - E;
36              Z = S + Y;
37              E = (Z - S) - Y;
38              S = Z;
39          }
```



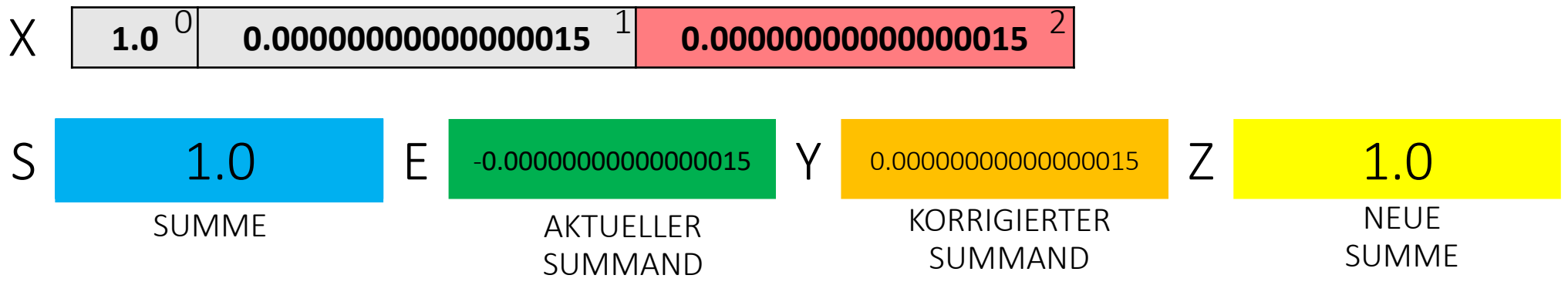
Code

```
31      static double kahan(double* X, size_t N) {
        ...
34          for (i = 0; i < N; i++) {
35              Y = X[i] - E;
36              Z = S + Y;
37              E = (Z - S) - Y;
38              S = Z;
39          }
```



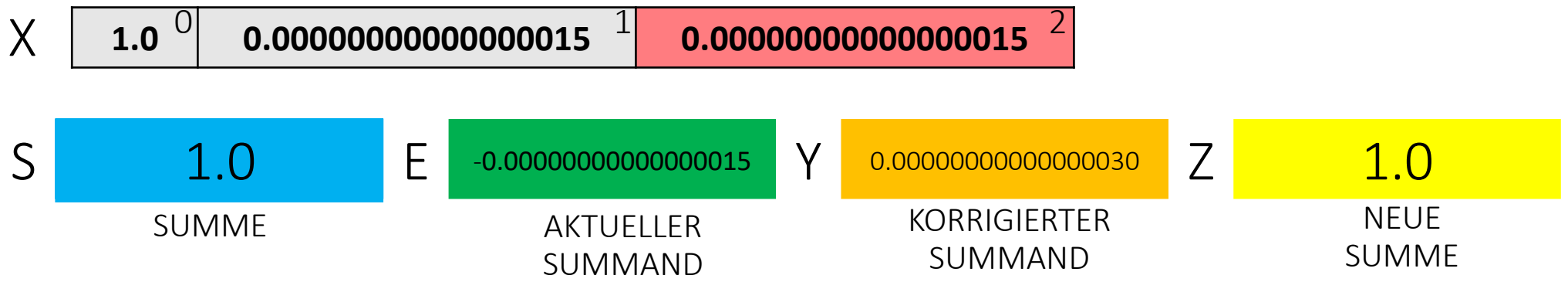
Code

```
31      static double kahan(double* X, size_t N) {
        ...
34          for (i = 0; i < N; i++) {
35              Y = X[i] - E;
36              Z = S + Y;
37              E = (Z - S) - Y;
38              S = Z;
39          }
```



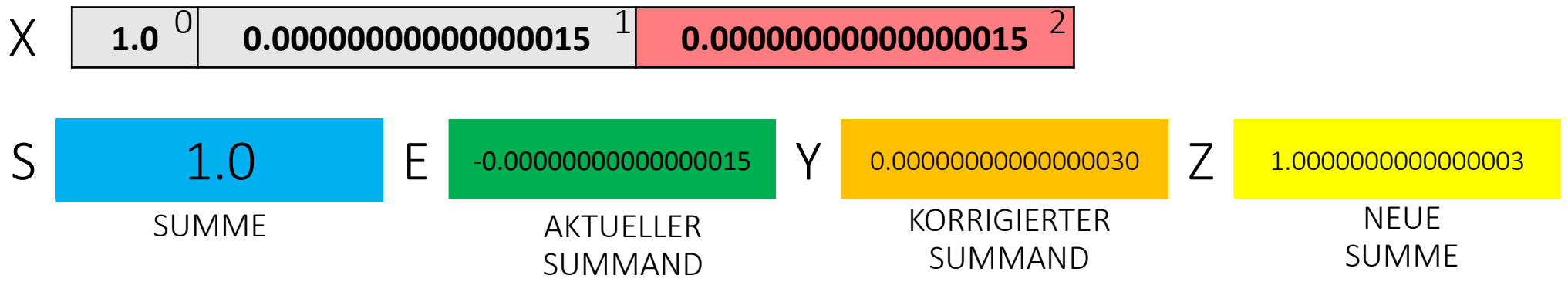
Code

```
31      static double kahan(double* X, size_t N) {
        ...
34          for (i = 0; i < N; i++) {
35              Y = X[i] - E;
36              Z = S + Y;
37              E = (Z - S) - Y;
38              S = Z;
39          }
```



Code

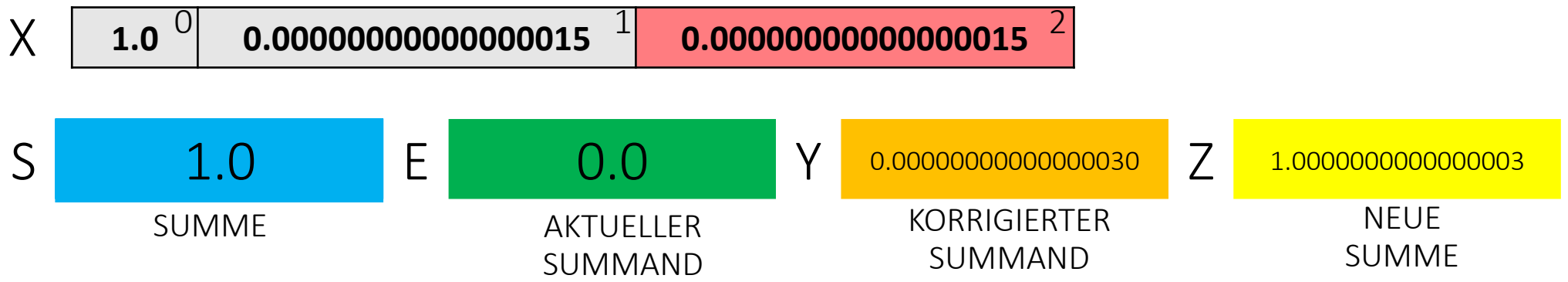
```
31      static double kahan(double* X, size_t N) {
        ...
34          for (i = 0; i < N; i++) {
35              Y = X[i] - E;
36              Z = S + Y;
37              E = (Z - S) - Y;
38              S = Z;
39          }
```



Aufgabe 2

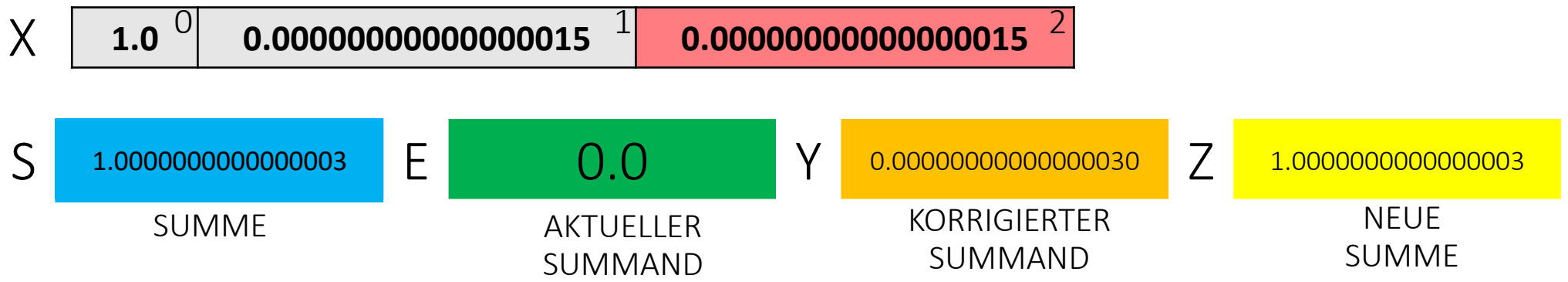
Code

```
31      static double kahan(double* X, size_t N) {
        ...
34          for (i = 0; i < N; i++) {
35              Y = X[i] - E;
36              Z = S + Y;
37              E = (Z - S) - Y;
38              S = Z;
39          }
```



Code

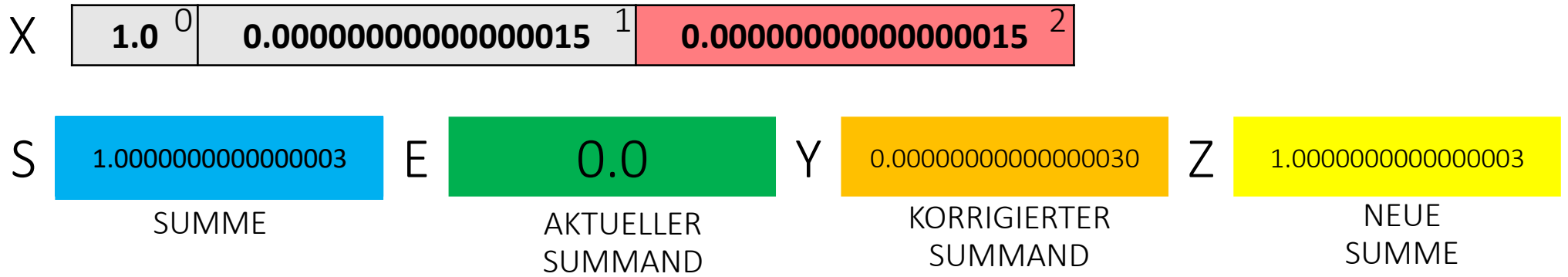
```
31      static double kahan(double* X, size_t N) {
        ...
34          for (i = 0; i < N; i++) {
35              Y = X[i] - E;
36              Z = S + Y;
37              E = (Z - S) - Y;
38              S = Z;
39          }
```



Aufgabe 2

Code

```
31      static double kahan(double* X, size_t N) {
32          double S = 0.0, E = 0.0, Y = 0.0, Z = 0.0;
33          size_t i;
34          for (i = 0; i < N; i++) {
35              Y = X[i] - E;
36              Z = S + Y;
37              E = (Z - S) - Y;
38              S = Z;
39          }
40          return S;
41      }
```



	Testen für unterschiedliche N
Aufsteigende Summe	Sobald die Summe groß ist, werden nachfolgende kleine Summanden wirkungslos (zu klein für die Mantisse) und komplett ignoriert.
Absteigende Summe	Kleine Zahlen werden zuerst addiert und bilden eine relevante Größe, bevor sie zur Gesamtsumme hinzugefügt werden.

	Testen für unterschiedliche N
Kahan-Summe	Rundungsfehler werden in einer Hilfsvariable (E) gespeichert und im nächsten Schritt korrigiert, wodurch der Informationsverlust verhindert wird.

Der Faktor 0,1 ist im Binärsystem eine periodische Zahl und kann daher nicht exakt gespeichert werden.

Durch das Abschneiden der Bits entstand ein winziger Rundungsfehler, der sich über die Betriebsdauer auf ca. 0,34 Sekunden aufsummierte.

Aufgrund der hohen Geschwindigkeit der Scud-Rakete entsprach diese Zeitabweichung einer Fehlpositionierung von über 500 Metern, wodurch das Ziel außerhalb des Erfassungsbereichs (range gate) lag.

Implementieren Sie die Funktionen ieee32_to_ll und ll_to_ieee32 in fpu.c, um die bidirektionale Konvertierung zwischen IEEE-754-Bitmustern und long long-Ganzzahlen durchzuführen.

Nutzen Sie hierfür ausschließlich Bit-Operationen und Ganzzahl-Arithmetik, ohne auf native Fließkomma-Datentypen zurückzugreifen.

Addition zweier Fließkommazahlen - Code (ieee32_to_ll)

Code

```
277     // a) Zerlegung der Eingabe
278     exp = (int32_t)((u & EXP_MASK) >> 23) - EXP_BIAS;
279     mant = (u & MANT_MASK) | HIDDEN_BIT;
280     sign = u & SIGN_MASK;
```

Vom Prinzip her dasselbe wie bei den Aufgaben vorhin!

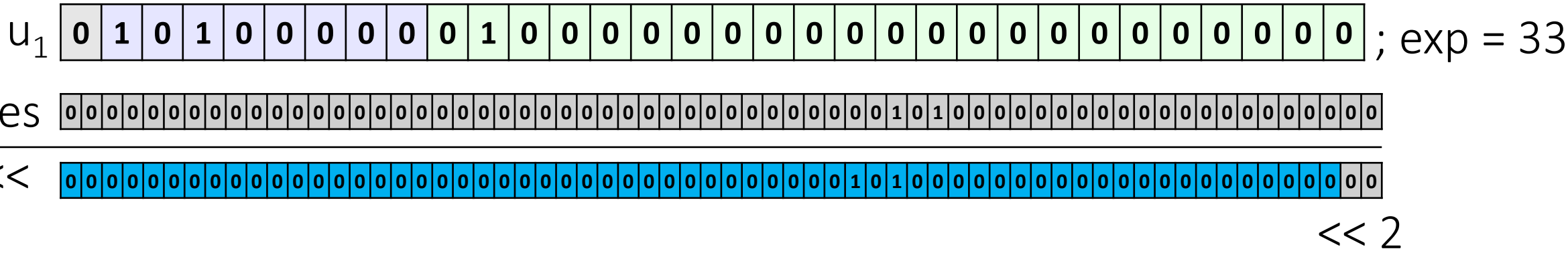
Addition zweier Fließkommazahlen - Code (ieee32_to_ll)

Code

```
288     // b) Mantisse ist 1.M * 2^exp. Bitmuster in 'mant' entspricht dem Wert * 2^23.
289     if (exp > 23) {
290         res <<= (exp - 23);
291     } else {
292         res >>= (23 - exp);
293     }
```

Code

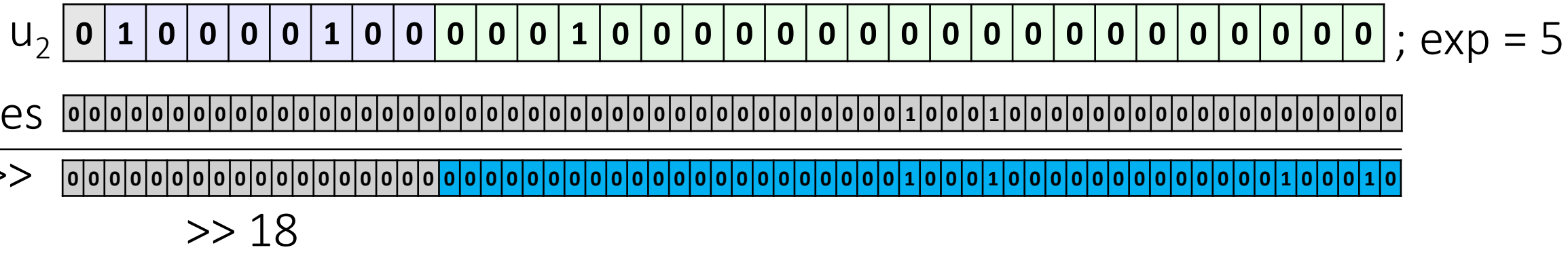
```
288     // b) Mantisse ist 1.M * 2^exp. Bitmuster in 'mant' entspricht dem Wert * 2^23.
289     if (exp > 23) {
290         res <<= (exp - 23);
291     } else {
292         res >>= (23 - exp);
293     }
```



Verschiebt die Mantisse basierend auf dem Exponenten so nach links oder rechts, dass das ursprüngliche Komma rechnerisch an die korrekte Ganzzahl-Position wandert.

Code

```
288     // b) Mantisse ist 1.M * 2^exp. Bitmuster in 'mant' entspricht dem Wert * 2^23.
289     if (exp > 23) {
290         res <<= (exp - 23);
291     } else {
292         res >>= (23 - exp);
293     }
```



Verschiebt die Mantisse basierend auf dem Exponenten so nach links oder rechts, dass das ursprüngliche Komma rechnerisch an die korrekte Ganzzahl-Position wandert.

Addition zweier Fließkommazahlen - Code (ll_to_ieee32)

Code

```
308     // a) Vorzeichen behandeln und Betrag bilden
309     if (i < 0) {
310         sign = SIGN_MASK;
311         uval = 0ULL - (unsigned long long)i;
312     } else {
313         uval = (unsigned long long)i;
314     }
```

Code

```
308     // a) Vorzeichen behandeln und Betrag bilden
309     if (i < 0) {
310         sign = SIGN_MASK;
311         uval = 0ULL - (unsigned long long)i;
312     } else {
313         uval = (unsigned long long)i;
314     }
```

i $(33\ 554\ 435)_D$ / $(10000000000000000000000011)_B$

Prüft, ob die Eingabezahl negativ ist, setzt entsprechend das IEEE-Vorzeichenbit und speichert den absoluten Betrag der Zahl in der vorzeichenlosen Variable uval, damit die nachfolgende Berechnung unabhängig vom Vorzeichen erfolgen kann.

Code

```
308     // a) Vorzeichen behandeln und Betrag bilden
309     if (i < 0) {
310         sign = SIGN_MASK;
311         uval = 0ULL - (unsigned long long)i;
312     } else {
313         uval = (unsigned long long)i;
314     }
```

i $(33\ 554\ 435)_D$ / $(10000000000000000000000011)_B$

Übernimmt positive Zahlen in uval, wobei das Vorzeichenbit auf 0 (für positiv) bleibt.

Addition zweier Fließkommazahlen - Code (ll_to_ieee32)

Code

```
316     // b) Höchstes gesetztes Bit finden (MSB)
317     for (int k = 63; k >= 0; k--) {
318         if (uval & (1ULL << k)) {
319             msb = k;
320             break;
321         }
322     } exp = msb + EXP_BIAS;
```

Code

```
316     // b) Höchstes gesetztes Bit finden (MSB)
317     for (int k = 63; k >= 0; k--) {
318         if (uval & (1ULL << k)) {
319             msb = k;
320             break;
321         }
322     } exp = msb + EXP_BIAS;
```

i (33 554 435)$_D$ / (10000000000000000000000011)$_B$; k 63

Da Integer keine expliziten Exponenten haben, sucht der Code in einer Schleife von Bit 63 rückwärts bis 0 nach dem ersten Bit, das eine 1 ist (das sogenannte "Most Significant Bit" oder MSB).

Code

```
316     // b) Höchstes gesetztes Bit finden (MSB)
317     for (int k = 63; k >= 0; k--) {
318         if (uval & (1ULL << k)) {
319             msb = k;
320             break;
321         }
322     } exp = msb + EXP_BIAS;
```

i $(33\ 554\ 435)_D$ / $(10000000000000000000000011)_B$; k 63

uval 0000000000000000000000000000000000000000000000000000000000000011

1ULL 1000000000000000000000000000000000000000000000000000000000000000

&

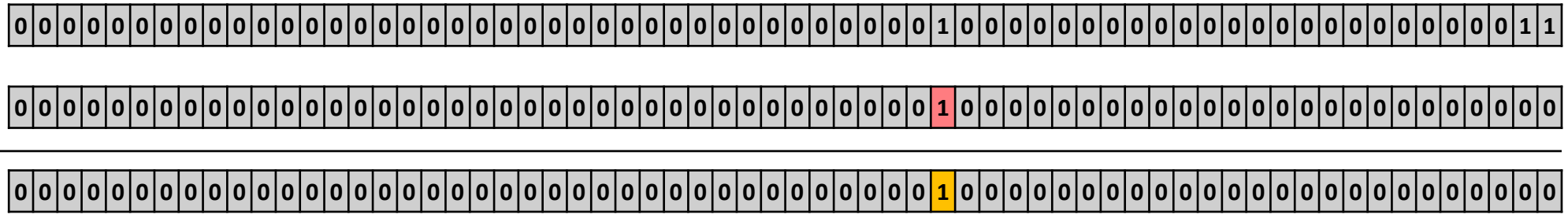
 0000000000000000000000000000000000000000000000000000000000000000

Da Integer keine expliziten Exponenten haben, sucht der Code in einer Schleife von Bit 63 rückwärts bis 0 nach dem ersten Bit, das eine 1 ist (das sogenannte "Most Significant Bit" oder MSB).

Code

```
316     // b) Höchstes gesetztes Bit finden (MSB)
317     for (int k = 63; k >= 0; k--) {
318         if (uval & (1ULL << k)) {
319             msb = k;
320             break;
321         }
322     } exp = msb + EXP_BIAS;
```

i $(33\ 554\ 435)_D$ / $(10000000000000000000000011)_B$; k 25

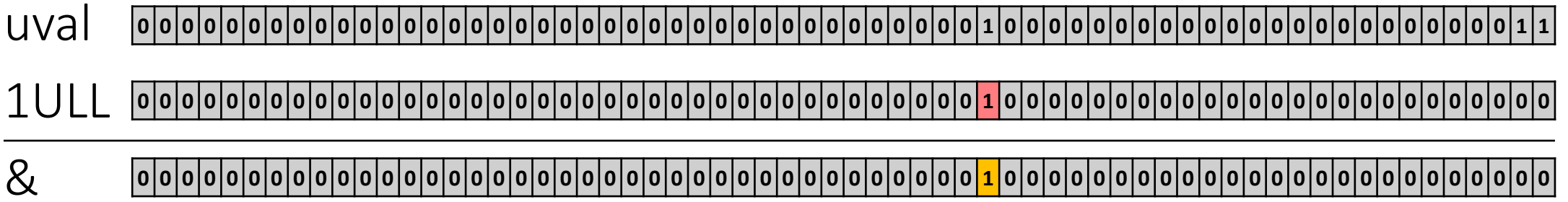


Da Integer keine expliziten Exponenten haben, sucht der Code in einer Schleife von Bit 63 rückwärts bis 0 nach dem ersten Bit, das eine 1 ist (das sogenannte "Most Significant Bit" oder MSB).

Code

```
316     // b) Höchstes gesetztes Bit finden (MSB)
317     for (int k = 63; k >= 0; k--) {
318         if (uval & (1ULL << k)) {
319             msb = k;
320             break;
321         }
322     } exp = msb + EXP_BIAS;
```

i $(33\,554\,435)_D$ / $(10000000000000000000000011)_B$; k 25 ; msb 25



Da Integer keine expliziten Exponenten haben, sucht der Code in einer Schleife von Bit 63 rückwärts bis 0 nach dem ersten Bit, das eine 1 ist (das sogenannte "Most Significant Bit" oder MSB).

Code

```
316     // b) Höchstes gesetztes Bit finden (MSB)
317     for (int k = 63; k >= 0; k--) {
318         if (uval & (1ULL << k)) {
319             msb = k;
320             break;
321         }
322     } exp = msb + EXP_BIAS;
```

i $(33\ 554\ 435)_D$ / $(1000000000000000000000011)_B$; k 25 ; msb 25

Da Integer keine expliziten Exponenten haben, sucht der Code in einer Schleife von Bit 63 rückwärts bis 0 nach dem ersten Bit, das eine 1 ist (das sogenannte "Most Significant Bit" oder MSB).

Code

```
316     // b) Höchstes gesetztes Bit finden (MSB)
317     for (int k = 63; k >= 0; k--) {
318         if (uval & (1ULL << k)) {
319             msb = k;
320             break;
321         }
322     } exp = msb + EXP_BIAS;
```

i (33 554 435)$_D$ / (10000000000000000000000011)$_B$; k 25 ; msb 25

exp 25 + 127

Da Integer keine expliziten Exponenten haben, sucht der Code in einer Schleife von Bit 63 rückwärts bis 0 nach dem ersten Bit, das eine 1 ist (das sogenannte "Most Significant Bit" oder MSB).

Code

```
324     // c) Mantisse extrahieren
325     if (msb <= 23) {
326         // Wenn Wert in 23 Bits passt: Links shiften, bis MSB auf Hidden-Bit Position liegt
327         mant = (uint32_t)((uval << (23 - msb)) & MANT_MASK);
328     } else {
329         // Wenn Wert größer ist: Rechts shiften und Runden
330         shift = msb - 23;
331         remainder = uval & ((1ULL << shift) - 1);
332         half = 1ULL << (shift - 1);
333
334         mant = (uint32_t)(uval >> shift);
335
336         ...
```

Bitmuster der vorzeichenlosen Ganzzahl (uval) so verschoben, dass ihr höchstes Bit genau auf der Position 23 landet, wobei im Fall einer Rechtsverschiebung (bei großen Zahlen) zusätzlich korrekt gerundet und ein eventueller Überlauf durch Neuanpassung des Exponenten behandelt wird.

Code

```
336         // Round to nearest, ties to even
337         if (remainder > half || (remainder == half && (mant & 1))) {
338             mant++;
339             // Check auf Mantissen-Überlauf durch Rundung
340             if (mant & (1u << 23)) {
341                 mant = 0;
342                 exp++;
343             }
344         }
345         mant &= MANT_MASK;
346     }
```

Wird geprüft, ob gemäß der Regel aufgerundet werden muss.
Falls das Aufrunden dazu führt, dass die Mantisse überläuft, wird der Exponent erhöht und die Mantisse entsprechend korrigiert.

Besondere Belegungen von Charakteristik und Mantisse

Vorzeichen	Charakteristik	Mantisse	Bedeutung	Mathematische Darstellung
0	$000\cdots000$	M	Fließkommazahl ohne 1	$(0.M)_2 \cdot 2^{-127+1}$
1	$000\cdots000$	M	Fließkommazahl ohne 1	$-(0.M)_2 \cdot 2^{-127+1}$
0	$0 < E < 2^{c-1} - 1$	M	Fließkommazahl	$(1.M)_2 \cdot 2^{C-127}$
1	$0 < E < 2^{c-1} - 1$	M	Fließkommazahl	$-(1.M)_2 \cdot 2^{C-127}$
0	?		Unendlich (positiv)	inf bzw. ∞
1			Unendlich (negativ)	-inf bzw. $-\infty$
0	$111\cdots111$	$M \neq 0$	Keine (rationale) Zahl	N. a. N. z.B.: $\sqrt[2]{-1}_{+} \cdot (1.M)$
1	$111\cdots111$	$M \neq 0$	Keine (rationale) Zahl	N. a. N. z.B.: $\sqrt[2]{-1}_{-} \cdot (1.M)$

Besondere Belegungen von Charakteristik und Mantisse

Vorzeichen	Charakteristik	Mantisse	Bedeutung	Mathematische Darstellung
0	$000\cdots000$	M	Fließkommazahl ohne 1	$(0.M)_2 \cdot 2^{-127+1}$
1	$000\cdots000$	M	Fließkommazahl ohne 1	$-(0.M)_2 \cdot 2^{-127+1}$
0	$0 < E < 2^{c-1} - 1$	M	Fließkommazahl	$(1.M)_2 \cdot 2^{C-127}$
1	$0 < E < 2^{c-1} - 1$	M	Fließkommazahl	$-(1.M)_2 \cdot 2^{C-127}$
0	$111\cdots111$	$000\cdots000$	Unendlich (positiv)	inf bzw. ∞
1	$111\cdots111$	$000\cdots000$	Unendlich (negativ)	-inf bzw. $-\infty$
0	$111\cdots111$	$M \neq 0$	Keine (rationale) Zahl	N. a. N. z.B.: $\sqrt[2]{-1} \cdot (1.M)$
1	$111\cdots111$	$M \neq 0$	Keine (rationale) Zahl	N. a. N. z.B.: $\sqrt[2]{-1} \cdot (1.M)$

∞ als Fließkommazahl

0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Fließkommazahl aus Zeichenkette auslesen (Vorgabe)

```c
#define P2DIGIT(strptr) (*strptr - '0')

static uint32_t str_to_ieee32(char* str) {
    uint32_t res = 0;
    char* ptr = str;
    bool has_digits = false, negative = false;
    long long int frac_part = 0, frac_divisor = 1, int_part = 0;

    if (!str || !*str) return 0x7FC00000; // NaN

    while (*ptr == ' ') { ptr++; }

    if (strstr(ptr,  "NaN") == ptr || strstr(ptr,  "nan") == ptr) { return 0x7FC00000; }
    if (strstr(ptr,  "Inf") == ptr || strstr(ptr,  "inf") == ptr) { return 0x7F800000; }
    if (strstr(ptr, "-Inf") == ptr || strstr(ptr, "-inf") == ptr) { return 0xFF800000; }

    if (*ptr == '-') { negative = true; ptr++; } else if (*ptr == '+') { ptr++; }

    // ...
}
```

Fließkommazahl aus Zeichenkette auslesen (Vorkommateil)

```
while (*ptr >= '0' && *ptr <= '9') {
        int_part = int_part * 10 + P2DIGIT(ptr);
        ptr++;
        has_digits = true;
}
```

Fließkommazahl aus Zeichenkette auslesen (Nachkommateil)

```
if (*ptr == '.') {
    ptr++;
    while (*ptr >= '0' && *ptr <= '9') {
        frac_part = frac_part * 10 + P2DIGIT(ptr);
        frac_divisor *= 10;
        ptr++;
        has_digits = true;
    }
}
```

Fließkommazahl aus Zeichenkette auslesen (Vorgabe)

```c
static uint32_t str_to_ieee32(char* str) {

    // ...

    if (!has_digits) { return 0x7FC00000; } // Invalid input

    uint32_t u_int = ll_to_ieee32(int_part);
    uint32_t u_frac = ll_to_ieee32(frac_part);
    uint32_t u_div  = ll_to_ieee32(frac_divisor);
    uint32_t u_inv  = ieee32_inv_u(u_div);
    uint32_t u_frac_scaled = ieee32_mul_u(u_frac, u_inv);

    res = ieee32_add_u(u_int, u_frac_scaled);

    if (negative) { res = res | SIGN_MASK; }
    else res &= ~SIGN_MASK;

    return res;
}
```