technische universität
dortmund

fakultät für informatik
informatik 12

Mikroarchitekturen

Peter Marwedel
Informatik 12
TU Dortmund

2012/05/04

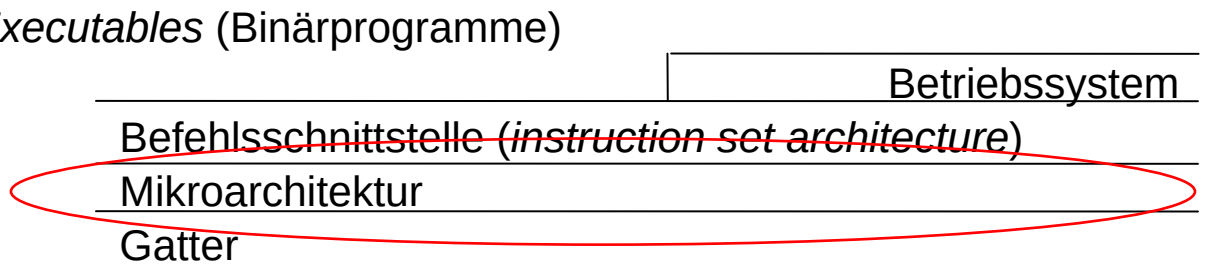
Diese Folien enthalten Graphiken mit Nutzungseinschränkungen. Das Kopieren der Graphiken ist im Allgemeinen nicht erlaubt.

Gegenüberstellung der Definitionen

Programmierschnittstelle	Interner Aufbau
Externe Rechnerarchitektur	Interne Rechnerarchitektur
Architektur	Mikroarchitektur
Rechnerarchitektur	Rechnerorganisation

Die externe Rechnerarchitektur definiert

- Programmier- oder Befehlssatzschnittstelle
- engl. *instruction set architecture, (ISA)*
- eine (reale) Rechenmaschine bzw.
- ein *application program interface (API)*.



3.2 Realisierung elementarer Datentypen
3.2.3 Ganze Zahlen

Interpretation der Bitvektoren

Annahme:
Darstellung im **Zweierkomplement**, Interpretation mittels **`int`**:

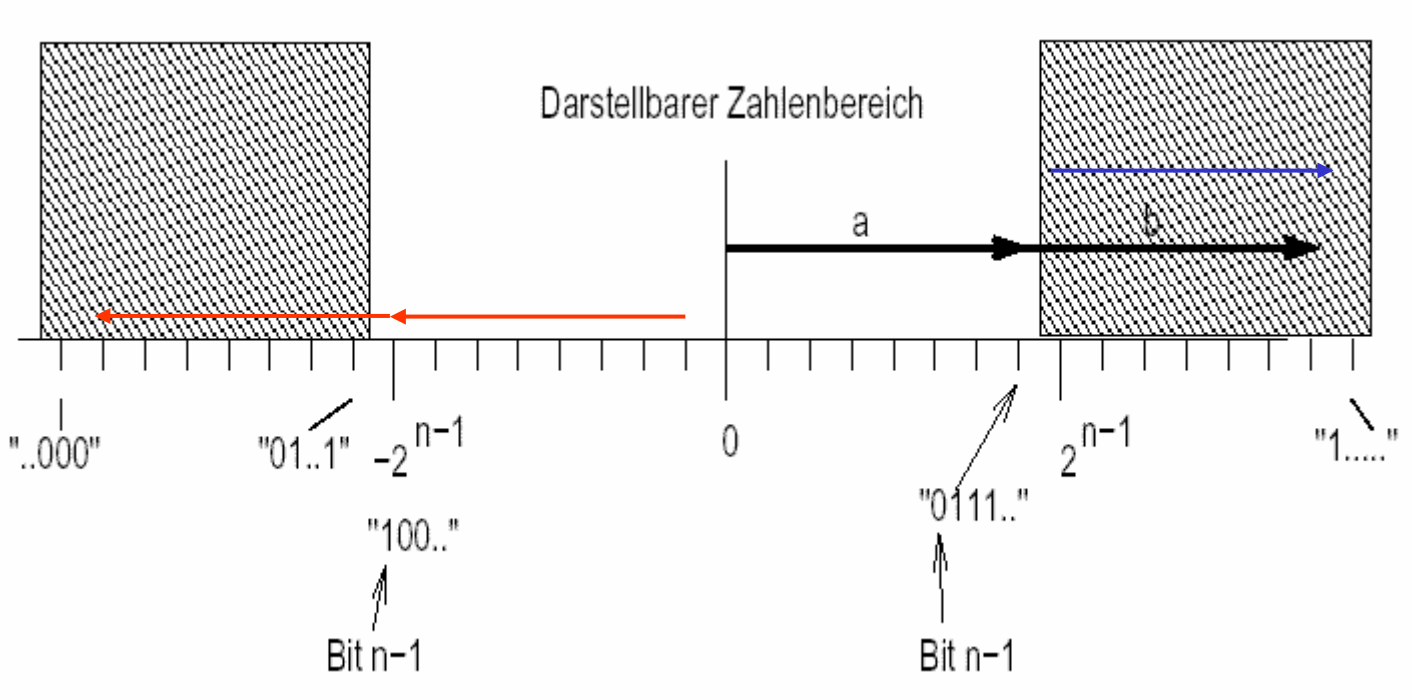
$$\texttt{int(a)} = -2^{a'left} \cdot a_{a'left} + \sum_{i=0}^{a'left-1} a_i \cdot 2^i$$

Beispiele:

```
-int("1000") = -8.
-int("1001") = -7.
```

Überläufe bei der Addition

Mittels $a_{n-1}..a_0$ bzw. $b_{n-1}..b_0$ darstellbar:



Überläufe: wenn beide Operanden das gleiche und das Ergebnis das entgegen gesetzte Vorzeichen haben:

Überlauf

a_{n-1}	b_{n-1}	Überlauf unter der Bedingung
0	0	$f_{n-1} = 1$
0	1	nicht möglich
1	0	nicht möglich
1	1	$f_{n-1} = 0$

Auch für **`int(a)`**= -2^{n-1} bzw. **`int(a)`** = -2^{n-1}. Es gilt also:

$$overflow\_add(a,b) = (a_{n-1} \equiv b_{n-1}) \wedge (a_{n-1} \textbf{ xor } f_{n-1})$$

overflow_add in vielen Rechnern in weiterem Condition-Code-Register (overflow_flag, ov) gespeichert .

Reaktion mittels *branch if_overflow*-Befehls.

Keine Unterscheidung zwischen **integer** und **natural** beim *add*-Befehl, beim *branch*-Befehl richtigen Datentyp wählen!

Sättigungsarithmetik: Datentypen beim *add*-Befehl bekannt.

Subtraktion

a_{n-1}	b_{n-1}	Überlauf unter der Bedingung
0	0	nicht möglich
0	1	$f_{n-1} = 1$
1	0	$f_{n-1} = 0$
1	1	nicht möglich

Überlauf, wenn beide Operanden entgegengesetztes Vorzeichen haben und das Ergebnis f_{n-1} ... f_0 ein anderes Vorzeichen als der erste Operand hat:

$$overflow\_sub(a,b) = (a_{n-1} \textbf{ xor } b_{n-1}) \wedge (a_{n-1} \textbf{ xor } f_{n-1})$$

Größenvergleich

Für Zahlen in Zweierkomplementdarstellung:

wegen: $overflow\_sub(a,b) = (a_{n-1} \texttt{ xor } b_{n-1}) \wedge (a_{n-1} \texttt{ xor } sf)$
$= (\overline{a_{n-1}} \wedge b_{n-1} \wedge sf) \vee (a_{n-1} \wedge \overline{b_{n-1}} \wedge \overline{sf})$

folgt:

$sf = \texttt{'1'} \rightarrow overflow\_sub = \overline{a_{n-1}} \wedge b_{n-1}$
$sf = \texttt{'0'} \rightarrow overflow\_sub = a_{n-1} \wedge \overline{b_{n-1}}$

ARM instruction set tests [ARM]	
Signed greater Than or Equal	*N set and V set, or N clear and V clear (N=V)*
Signed Less Than	*N set and V clear, or N clear and V set (N != V)*
Signed Greater Than	*Z Clear , and either N set and V set, or N clear and V clear (Z=0, N=V)*
Signed Less Than or Equal	*Z set, or N set and V clear, or N clear and V set (Z=1, N != V)*

☞:

overflow_sub	sf	Kommentar	Ergebnis
0	0	Kein Überlauf, F positiv, $0 \leq a\text{-}b \leq 2^{n-1}\text{-}1$	$a \geq b$
0	1	Kein Überlauf, F negativ, $-2^{n-1} \leq a\text{-}b < 0$	$a < b$
1	0	$(a_{n-1}=1) \wedge (b_{n-1}=0)$: a negativ, b positiv	$a < b$
1	1	$(a_{n-1}=0) \wedge (b_{n-1}=1)$: a positiv, b negativ	$a > b$

Daraus ergibt sich:

$a<b \quad \Leftrightarrow \quad (overflow\_sub \texttt{ xor } sf)$
$a \leq b \quad \Leftrightarrow \quad (a<b) \vee (a=b) \quad \Leftrightarrow \quad (overflow\_sub \texttt{ xor } sf) \vee zf$
$a>b \quad \Leftrightarrow \quad \neg(a \leq b) \quad \Leftrightarrow \quad \neg((overflow\_sub \texttt{ xor } sf) \vee zf)$
$a \geq b \quad \Leftrightarrow \quad \neg(a<b) \quad \Leftrightarrow \quad overflow\_sub \equiv sf$

Anwendung

Codeerzeugung für Vergleiche:

z.B. Intel-Prozessoren (im Prinzip):

```
sub a,b   # Setzen von Flag-Registern
bgt ziel  # Prüfen der Flag-Register
```

Größenvergleiche ≠ Subtraktion und Vorzeichentest:

$a<b \Leftrightarrow (overflow\_sub$ **xor** $sf)$

Beispiel: *Predicated execution* @ ARM-Prozessor

Opcode (31:28)	*Mnemonic Extension*	*Meaning*	*Status flag state*
0001	NE	*Not Equal*	…
0010	CS/HS	*Carry Set/Unsigned Higher or Same*	…
0011	CC/LO	*Carry Clear/Unsigned Lower*	…
0100	MI	*Minus/Negative*	…
0101	PL	*Plus/Positive or Zero*	…
0110	VS	*Overflow*	V set
1000	HI	*Unsigned Higher*	…
1001	LS	*Unsigned Lower or Same*	…
…	…	…	…

Multiplikation

Vorgehen analog zu natürlichen Zahlen erfordert Sonderbehandlung des Vorzeichens. Wird beim Algorithmus von Booth vermieden.

Grundidee: für eine Kette von Einsen nur 2 Additionen erforderlich:

`i`: Position der am weitesten rechts stehenden 1

`j`: die Position der am weitesten links stehenden 1:

```
              j  i
A * int("0001111000")
```

$$\underset{\text{int("0001111111")}}{\overset{j\ \ i}{\ }} = 2^{j+1} - 1.$$

$$\text{int("0000000111")} = 2^{i} - 1.$$

$$\text{int("0001111000")} = (2^{j+1} - 1 - 2^{i} + 1)$$

Also:

$$\underset{A * \text{int("0001111000")} = A * (2^{j+1} - 2^{i})}{\overset{j\ \ i}{\ }}$$

Beispiel:

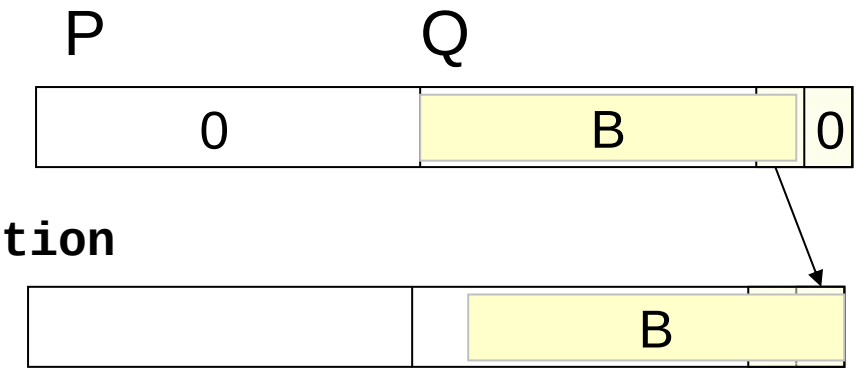
```
A = int("0011") = 3; B = int("0110") = 6 ;
n = 4; A * 6 = A * int("0110") = A * (2^3 − 2^1)
```

`-A` für die volle Bitvektorlänge des Ergebnisses: `-A =int("11111101")`
Schritte des Rechenvorgangs:

B	Aktion		Ergebnis
			"00000000"
"0110"	keine	+	"00000000"
			"00000000"
"0110"	−2*A	+	"11111010"
			"11111010"
"0110"	keine	+	"00000000"
			"11111010"
"0110"	+8*A	+	"00011000"
			"00010010"

Booth-Algorithmus

```
FUNCTION Booth(A,B: IN bit_vector) RETURN bit_vector IS
       CONSTANT n : natural := A'LENGTH;
       VARIABLE P : bit_vector(n-1 DOWNTO 0):=(OTHERS => '0');
       VARIABLE Q : bit_vector(n DOWNTO 0) :=(OTHERS => '0');
BEGIN
 Q(n DOWNTO 1) := B;
 FOR i IN 0 TO n-1 LOOP
   CASE Q(1 DOWNTO 0) IS
       WHEN "10" => P := P - A;
       WHEN "01" => P := P + A;
       WHEN OTHERS => -- keine Aktion
   END CASE;
   P & Q := sra (P & Q);
  END LOOP;
  RETURN P(n-2 DOWNTO 0) & Q(n DOWNTO 1);
END Booth;
```



Einzelschritte der Prozedur

Opr.	P	Q	Kommentar
	0000	00000	
	0000	01100	Q(1 DOWNTO 0) = "00"
sra	0000	00110	Q(1 DOWNTO 0) = "10"
-A	1101	00110	- 1 Bit vom LSB entfernt =-2*A
sra	1110	10011	Q(1 DOWNTO 0) = "11"
sra	1111	01001	Q(1 DOWNTO 0) = “01"
+A	0010	01001	"1111"+"0011"="0010"; +8*A
sra	0001	00100	

LSB des Ergebnisses

MSB des Ergebnisses

```
FOR i IN 0 TO n-1 LOOP
 CASE Q(1 DOWNTO 0) IS
  WHEN "10" => P := P - A;
  WHEN "01" => P := P + A;
  WHEN OTHERS => --
 END CASE;
 P & Q := sra (P & Q);
```

Ergebnis: *int*(“0010010“) =18.
Wegen ’-A’: P & Q in 2k-Darstellung u. arithmethisches Schieben.

Stellengerechte Addition/Subtraktion im *Booth*-Algorithmus:

```
FUNCTION Booth(A,B: IN bit_vector) RETURN bit_vector IS
CONSTANT n : natural := A'LENGTH;
  -- Vor.: A'LENGTH = B'LENGTH
  VARIABLE P:bit_vector(n-1 DOWNTO 0):=(OTHERS=>'0');
  VARIABLE Q:bit_vector(n DOWNTO 0) := (OTHERS=>'0');
BEGIN
 Q(n DOWNTO 1) := B;
 FOR i IN 0 TO n-1 LOOP
   CASE Q(1 DOWNTO 0) IS
             WHEN "10" => P := P - A;
             WHEN "01" => P := P + A;
             WHEN OTHERS => -- keine Aktion
   END CASE;
       P & Q := sra (P & Q); -- >>, arithm., n. rechts
 END LOOP;
 RETURN P(n-2 DOWNTO 0) & Q(n DOWNTO 1);
END Booth;
```

Booth-Algorithmus
Sonderfall: A = kleinste darstellbare Zahl

Falls **A** die kleinste darstellbare Zahl ist,

dann ist **-A** nicht in n Bit darstellbar.

Beispiel:

(-8)*(-8) = 64 = int("0100 0000")

Dieser Wert kann vom obigen Booth-Algorithmus nicht geliefert werden, da aufgrund des abschließenden Schiebeschritts **P(n-1)=P(n-2)** ist.

Ausweg: Verlängern von **P** um ein Bit.

Korrekte Interpretation von B ?

Nachweise für B´*length*=4; Seien b_0 bis b_3 die Elemente von B.

Jeder einzelne Schritt: stellengerechte Multiplikation von A mit (b_{i-1} -b_i);

$$\begin{aligned} \text{Booth}(A, B) = \; & A * (b_{-1} - b_0) * 2^0 + \\ & A * (b_0 - b_1) * 2^1 + \\ & A * (b_1 - b_2) * 2^2 + \\ & A * (b_2 - b_3) * 2^3 \end{aligned}$$

Es gilt: $-b_i \times 2^i + b_i \times 2^{i+1} = b_i \times 2^i$

➔:*Booth*(A, B)= $A \times (b_0 \times 2^0 + b_1 \times 2^1 + b_2 \times 2^2 - b_3 \times 2^3) = A \times int(B)$

Trick der korrekten Behandlung von B als Integer-Zahl:
”vergessen” , bei einer ’1’ im Vorzeichen von B die eigentlich übliche Behandlung am linken Rand einer Folge von Einsen vorzunehmen.

Verbesserungen des Booth-Algorithmus:

Einzelne Nullen bzw. Einsen sollen wie beim Standard-verfahren nur eine Operation erzeugen.

Bei isolierter 1 im Bit i wird $2^{i+1} - 2^i = 2^i$ addiert.

Bei isolierter 0 im Bit i wird $2^i - 2^{i+1} = -2^i$ addiert.

Übergang auf die Betrachtung eines Fensters von 3 Bit und Verschiebung um jeweils ± 2 Bit; abhängig vom Muster im Fenster Addition von $\pm 2 * A$, A (siehe Hayes)

Ignorieren von Folgen gleicher Ziffern

Multiplikation

Ergebnis: Produkt ganzer Zahlen, soweit bei fester Datenwortlänge möglich.

Doppelt-langes Ergebnis?

Integer in VHDL

Interpretation von Bitvektoren als **integer**
(falls der Zahlenbereich ausreicht):

```
function int (a: bit_vector) return
  integer is -- Annahme: a'left > 0
constant t : natural := (2 ** (a'left));
  begin
    if a(a'left) = '0' - - positive Zahl
    then return nat (a)
    else return(nat(a(a'left-1 downto 0))-t)
    end if;
  end int;
```

2^n im Datentyp **integer** evtl. nicht darstellbar.

Es kann helfen, $-(2^{n-1}) + nat(a(a'left\text{-}1 \text{ downto } 0)) - (2^{n-1})$ zu berechnen.

3.2.4 Gleitkomma-Operationen System-Aspekte

Verwendung von Gleitkomma-Arithmetik in höheren Programmiersprachen?

Keinerlei Aussagen im IEEE-Standard.

Grund: Im IEEE-Gremium kein Compilerbauer.

Zwischenrechnungen mit welcher Genauigkeit?

Intuitive Ansätze:

1. Für alle Zwischenrechnungen maximal verfügbare Genauigkeit:

Unerwartete Ergebnisse, Beispiel:

Sei **q** Variable einfacher Genauigkeit (32 Bit).

```
q = 3.0/7.0; print(q==(3.0/7.0))
```

führt zum Ausdruck von **false**,

Bei Zuweisung von **3.0/7.0** zu **q** gehen Mantissenstellen verloren.

Wird der Vergleich (als “Zwischenrechnung”) mit doppelter Genauigkeit ausgeführt, so müssen Mantissenstellen von **q** mit Nullen oder Einsen aufgefüllt werden.

Zwischenrechnungen mit welcher Genauigkeit?

2. Für alle Zwischenrechnungen das Maximum der Genauigkeiten der Argumente.

Unnötiger Verlust bekannter Information.

Beispiel: Seien **`x,y`**: Variablen einfacher Genauigkeit;

sei **`dx`** Variable doppelter Genauigkeit.

Betrachte: `dx = x - y`

Subtraktion: einfache Genauigkeit.

`dx` würden im Prinzip bekannte Mantissenstellen nicht zugewiesen.

Lösung des Problems

Im Compiler zwei Durchläufe durch Ausdrucksbaum:

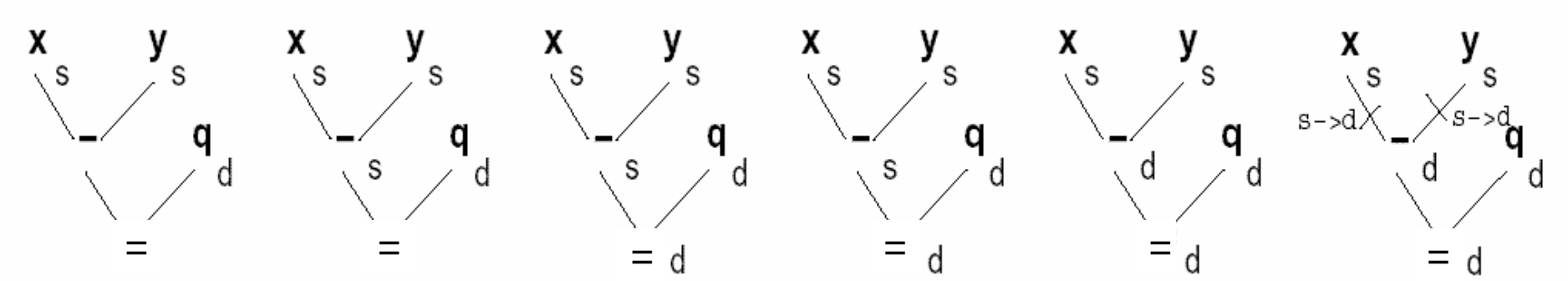
1. **Durchlauf von den Blättern zur Wurzel**
   Für jede arithmetische Operation wird das Maximum der Genauigkeiten der Argumente gebildet.
   Genauigkeit einer Zuweisung = Genauigkeit der Zielvariablen.
   Genauigkeit von Vergleichen: in der Regel das Minimum der Genauigkeit der Argumente.
   Ausdrucksbaum kann jetzt inkonsistent sein.

2. **Durchlauf von der Wurzel zu den Blättern**
   Genauigkeit reduzieren, wenn Operationsergebnis nicht in der bisherigen Genauigkeit benötigt wird.
   Genauigkeit erhöhen, wenn Operationsergebnis in größerer Genauigkeit benötigt wird.

2 Durchläufe

1. **Durchlauf von den Blättern zur Wurzel**
   ∀ arithmetischen Operationen: Max. der Genauigkeiten der Argumente!
   Genauigkeit einer Zuweisung = Genauigkeit der Zielvariablen.
   Genauigkeit von Vergleichen: i.d. Regel Min. der Genauigkeit der Argumente.
   Ausdrucksbaum kann jetzt inkonsistent sein.
2. **Durchlauf von der Wurzel zu den Blättern**
   Genauigkeit ↓, wenn Ergebnis nicht in der bisherigen Genauigkeit benötigt.
   Genauigkeit ↑, wenn Ergebnis in größerer Genauigkeit benötigt wird.

Beispiel: **s**: einfache, **d** doppelte Genauigkeit. 2 Konvertierungen.



Verbleibende Probleme

- Die Genauigkeit ist nur im Kontext zu ermitteln.
- Der mögliche Fehler eines Ergebnisses ist unbekannt.

Alternative (Kulisch): **Intervallarithmetik**

Für alle Berechnungen werden die Intervallgrenzen der möglichen Werte betrachtet.
In verschiedenen Paketen angeboten.
☞ sehr umfangreicher Artikel bei Wikipedia.

Unzulässige Optimierungen

Viele zunächst korrekt erscheinende Compiler-„Optimierungen” sind falsch.

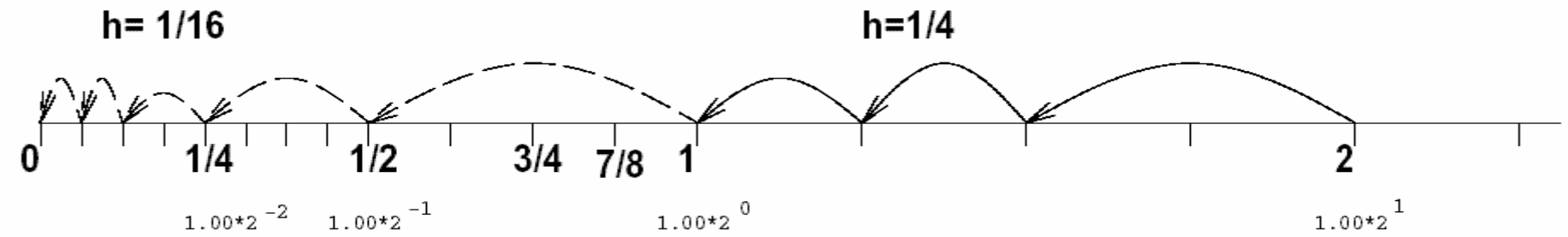
Beispiele:

Ausdruck	unzulässige Optim.	Problem
x / 10.0	0.1 * x	x nicht exakt darstellbar
$x*y-x*z$	$x*(y-z)$	Falls $y \approx z$
$x+(y+z)$	$(x+y)+z$	Rundungsfehler
konstanter Ausdruck	Result. Konstante	Flags werden nicht gesetzt
gemeinsamer Ausdruck	Ref. auf 1. Berechnung	Rundungsmodus geändert?

Mögliches Verhalten bei Subtraktion von konstanten Werten bei Vergleichsoperatoren

`eps = 1;`
`while (eps>0)`
`{h = eps;`
`eps = eps*0.5;`
`}`
	`eps = 1;`
	`while ((eps+1)>1)`
	`{h = eps;`
	`eps = eps*0.5;`
	`}`

Zwei Versionen der WHILE-Schleifen:



Die linke Schleife liefert bei 32 Bit-Gleitkommaarithmetik mit Realisierung nicht-normalisierter Zahlen den Wert $2^{-126}*2^{-23} \sim 1{,}4*10^{-45}$, bei Beschränkung auf normalisierte Zahlen den Wert $2^{-126} \sim 1{,}1*10^{-38}$.

Unproblematische Optimierungen

Lediglich einige sehr einfache Optimierungen sind unproblematisch:

Ausdruck	Optimierte Fassung
$x+y$	$y+x$
$2*x$	$x+x$
$1*x$	x
$x/2.0$	$x*0.5$

Zusammenfassung

- Vorzeichenbehaftete Zahlen (integer)
  - Überläufe
  - Multiplikation mit dem *Booth*-Algorithmus
- Gleitkomma-Zahlen
  - Bestimmung der Genauigkeit von Zwischenrechnungen
  - Zulässige und unzulässige Optimierungen