Kapitel 07 - Darstellung von Informationen

Peter Ulbrich

AG Systemsoftware

Veranstaltungswebseite

Einleitung

- Bisher:
  - Elementare Datentypen, wie z. B. `int`, `bool`, `char`, `float`
  - Bitoperationen und Byte-Order sind ebenfalls bekannt
- Was noch fehlt:
  - Interner Aufbau der Datentypen
    - Zweierkomplement
    - IEEE-754
  - Konvertierung zwischen den Datentypen

Darstellung von Gleitkommazahlen

- Zur Erinnerung: Gleitkommazahlen im Standard `IEEE 754` definiert
- In C Unterscheidung zwischen
  - einfacher Genauigkeit (32 Bit → `float`) und
  - doppelter Genauigk
- Außerdem noch (n
  - *Halbe* Genauigkeit (
  - *Vierfache* Genauigkeit (128 Bits)
  - *Achtfache* Genauigkeit (256 Bits)

Schauen wir uns die Kodierung einmal im Detail an

Darstellung von Gleitkommazahlen

- Wie kann Gleitkommazahl dargestellt werden?

- Vorkomma- und Nachkommaanteil

  4711.1174_D

- Nachkommadarstellung mit Exponent

  $0.47111174 \cdot 10^4_D$

- Die zweite Darstellu … erwendet

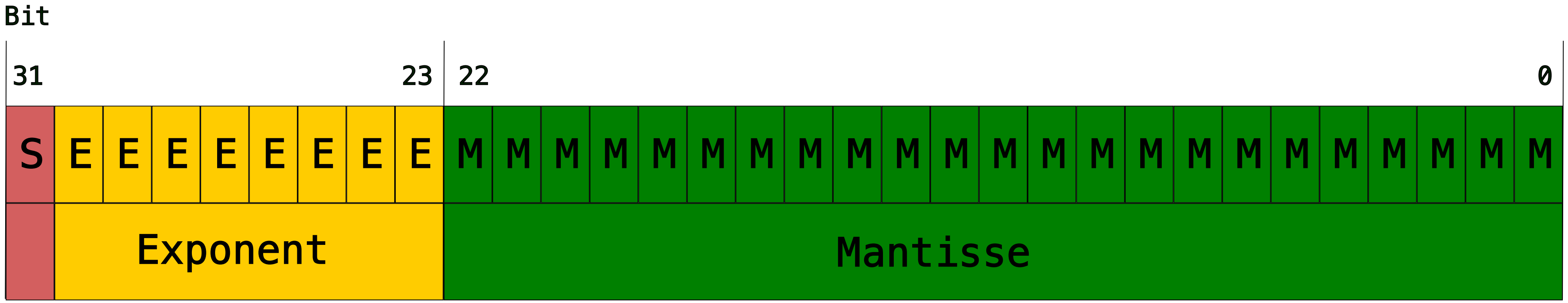
  → Float $F = (-1)$ …

Was bedeutet das jetzt alles? 🤯

Aufbau einer Gleitkommazahl

Bestandteil	Bits (FP-32)
Vorzeichen S (*Sign*)	Bit 31
Exponent E	Bit 30-23
Mantisse M (Nachkommateil)	Bit 22-0

- Aufbau einer Gleitkommazahl mit einfacher Genauigkeit (32 Bit)



Gleitkommazahl - Vorzeichen S

- Höchstwertiges Bit
- Mögliche Werte:
  - `0` → positive Zahl
  - `1` → negative Zahl
- Deswegen ist das Vorzeichen in der Formel auch als $(-1)^S$ dargestellt

Gleitkommazahl - Exponent E

- Darstellung mit **Bias**:

  Auf den Exponent wird zusätzlich immer eine Konstante addiert

- Konstante abhängig von der Genauigkeit der Gleitkommazahl
  - Ist für jede Genauigkeit fest
  - Einfache Genauigkeit → Bias $B = 127$

- Beispiel: $E = 16\,(\rightarrow 2^{16}) \rightarrow$ `0001 0000`
  - Bias wird addiert: : $E = 16 + B = 143 \rightarrow$ `1000 1111`
  - *Biased Exponent* wird in das Exponentfeld eingetragen

Gleitkommazahl - Exponent E

- Warum so kompliziert?
- Wozu die Verwendung eines Bias?
- Darstellbarer Wertebereich: $2^{-126} - 2^{127}$ für einfache Genauigkeit
- Nach Bias-Addition: $1 - 254$
  - Darstellung mit 7 Bit möglich
  - **Keine** Verschwendung von einem Bit für Vorzeichen
- Exponentwerte mit spezieller Bedeutung
  - `0000 0000` $= 0 \rightarrow$ u. a. für die Null
  - `1111 1111` $= 255 \rightarrow \infty$ und *Not a Number* (=*NaN*)

Gleitkommazahl - Mantisse M

- Nachkommateil M einer Zahl in binärer Darstellung
- Bestimmung
  - Verschiebe das Komma bis eine 1 vor dem Komma steht
  - Das höchstwertige Bit ist Vorkommaanteil

    (implizite 1, wird **nicht** gespeichert)
  - Restlichen Bits entsprechen dem **Nachkommaanteil** M

  $2^{-1}\ 2^{-2}\ 2^{-3}$
- Beispiel: 8.125_D = `1000,001` $2^{-3} = 0{,}125$
  - Verschiebung des Kommas, bis **nur** das MSBit noch vor dem Komma steht

    → `1.000001` $\cdot 2^3$
  - Exponent entspricht Anzahl an verschobenen Stellen

    → $E = 3 + B = 3 + 127 = 130$

Genauigkeit der Gleitkommazahlen

- Beispiele bis hierher gelten nur für einfache Genauigkeit
- Berechnungsformel ist allerdings für alle gleich
  - Float $F = (-1)^S \cdot (1 + M) \cdot (2)^E$
- Unterschiede: Wertebereich, Bias und Bitbreite von Exponent und Mantisse

  → Ansonsten kein Unterschied in der Berechnung!

Name	**Bits**	E **(Bits)**	**Bias**	**Min.** E	**Max.** E	M **(Bits)**
FP-16 (*Half*)	16	5	15	-14	15	10
FP-32 (*Single*)	32	8	127	-126	127	23
FP-64 (*Double*)	64	11	1023	-1022	1023	52

Umwandlung Nachkommaanteil

- Nachkommastellen sind in ebenfalls gut darstellbar
- Beispiel: 16.625_D
- Vorkommaanteil: Ist einfach→ $2^4 =$ `0001 0000`
- Nachkommaanteil: Darstellung als Summe von Zweierpotenzen

  $\rightarrow 2^{-1} + 2^{-2} + 2^{-3} + 2^{-4} + \ldots$
- Beispiel: $0.625_D = 2^{-1} + 2^{-3} =$ `.101` B

Ungenauigkeit von Gleitkommazahlen

- Gleitkommazahlen **ungeeignet** für **exakte** Darstellung **beliebiger** Zahlen:
- 0.375_D = `0.011` → $2^{-2} + 2^{-3}$
- **Aber:** 0.1_D nicht genau darstellbar → Rundung auf nächste darstellbare Zahl
- Verschiedene Rundungsstrategien:
  - Immer **aufrunden** (zu $+\infty$)
  - Immer **abrunden** (zu $-\infty$)
  - Immer **Richtung Null** runden

Es gibt verschiedene Online-Tools für das Darstellen von IEEE-754-Zahlen, zum Beispiel auf weltz.de

Spezielle Gleitkommazahlen

- *Genau Null*
  - Exponent und Mantisse haben jeweils Wert `0`
  - Vorzeichen unterscheidet zwischen `+0` und `−0`
- *Infinity* (∞)
  - Alle Bits im Exponent gesetzt, Mantisse hat Wert `0`
  - Vorzeichenbit bestimmt $\pm\infty$
- *NaN (Not a Number)*
  - Kodiert mathematisch nicht erlaubte Operationen (z.B. Division durch `0`)
  - Exponentbits alle gesetzt, aber Mantisse hat Wert $\neq$ `0`

Beispiel - Umwandlung in Gleitkommazahl

- Beispiel: Aus 16.625 = $\texttt{10000.101}_B$ wird:

  1. Verschiebe Komma bis hinter die höchste Stelle → $\texttt{1.0000101} \cdot 2^4$
  2. Exponenten $E = 4 + 127 = 131 = \texttt{10000011}$
  3. Mantisse M entspricht Nachkommaanteil → `0000101`
  4. Vorzeichenbit S: Zahl positiv → `0`

- Insgesamt

`0100 0001 1000 0101 0000 0000 0000 0000`

Einschub: Floating-Point Unit (FPU)

- Berechnung von Gleitkommazahlen braucht auf der CPU viele Schritte
  → Verwendung von Gleitkommazahlen ist ein Flaschenhals
- Einsatz spezialisierter Hardware-Komponente nur für Gleitkommazahlen:
  → **Floating-Point Unit** (FPU)
- Auf allen modernen CPUs enthalten
- Einige alte CPUs und Mikrocontroller enthalten diese aber nicht
- Falls nicht vorhanden, dann Rückfall auf Software-Berechnung

Einschub: Fixed-Point

- Und wenn wir eine feste Anzahl an Nachkommastellen benötigen?
- Lösung: *Fixed-Point-Zahlen*
- Nur per Software möglich, keine übliche Hardware-Komponente
- Eigene Umsetzung notwendig, C bietet keine vorhandene Implementierung
- Beispiel: 16.375_D in Fixed-Point-Notation mit 2 Nachkommastellen

  `10000.011` → `010000.01` (Fixed-Point)

Zwischenstand

- Wir kennen jetzt fast alles Grundlegende, was elementare Datentypen angeht!
- Zeit für einen Blick **in** die CPU
  - Byte-Order wurde bereits vorgestellt (✅)
  - Es fehlt: Kodierung der Zahlen zur effizienten Verarbeitung

    → Jetzt!

Sign-Magnitude

- Vorgehen: „Ergänze eine positive Zahl in Binärdarstellung um Vorzeichenbit"
  - `0` → **positiv**
  - 1 → **negativ**
- Beispiel: -3 = 1 11
- Vorteil: Einfach zu lesen und zu konvertieren
- Aber: Es kostet Mehraufwand
  - Hälfte des Zahlenraums geht an das Vorzeichen verloren
  - `0` wird doppelt kodiert: $+0$ → `00` und -0 → `10`

Beispiele - Sign-Magnitude

- Bitbreite: 1 Byte = 8 Bit
- Beispiel: -79_D

Sign	64	32	16	8	4	2	1
1	1	0	0	1	1	1	1

- Beispiel: -255_D

…	Sign	128	64	32	16	8	4	2	1
…	1	1	1	1	1	1	1	1	1

→ Bei fester Byte-Größe Überlauf in das nächste Byte (7 Bits verschwendet)

Einerkomplement

- Positive Zahl: Keine Veränderung der Binärdarstellung – wie bisher
- Negative Zahl
  - Umwandlung in Binärdarstellung
  - Invertiere den Wert jedes Bits

  → Negative Zahlen haben ebenfalls 1 als MSBit
- Beispiel:
  - +79 → $0100\ 1111_B$
  - -79 → $1011\ 0000_1$
- Auch hier gilt: 0 wird doppelt kodiert mit $+0\ (00_1)$ und $-0\ (11_1)$

Umwandlung Einerkomplement zu Dezimal

- MSB ist 0: Umwandlung in Dezimal wie bisher
- MSB ist 1:
  1. Wandle die Binärzahl in Dezimalzahl um
  2. Addiere 1
- Beispiel: -79

-128	64	32	16	8	4	2	1
1	0	1	1	0	0	0	0

1. Summe der Bitwertigkeiten: -128 + 32 + 16 = -80
2. Addiere 1: -80 + 1 = -79

Zweierkomplement

- Positive Zahl: Keine Veränderung der Binärdarstellung
- Negative Zahl:
  1. Umwandlung in Binärdarstellung
  2. Invertiere jedes Bit
  3. Addiere 1
- Vorteile des Zweierkomplements:
  - **Keine** Überprüfung auf das Vorzeichen

    → Addition und Subtraktion sind identische Operationen
  - Keine doppelte Darstellung der 0
- Dies ist die in Computern übliche Darstellung

Beispiel - Zweierkomplement

- Beispiel

  $1 - 4 = -3 \Leftrightarrow 1 + (-4) = -3$

- Umwandlung 1: $\texttt{0001}_2$

- Umwandlung (-4)

  1. Invertiere $\texttt{0100}_B$ (+4) → $\texttt{1011}_1$
  2. Addiere 1 → $\texttt{1100}_2$

  1   −4

- Addition: $\texttt{0001}_2 + \texttt{1100}_2 = \texttt{1101}_2$ → -3

- Anmerkung: Bitweise Addition erfolgt genau wie bei der schriftlichen Addition

  → Übertrag (*Carry-Over*) wird bei der nächsthöheren Stelle zusätzlich addiert

	0	0	0	1
+	1	1	0	0
=	1	1	0	1

Zweikomplement im Computer

- Computer verwenden intern das Zweierkomplement
- Aufwand für Konvertierung < Aufwand für Rechenoperationen
- Anstatt zwei Operationen
  1. Prüfen des Vorzeichenbit
  2. anschließend Addieren oder Subtrahieren
- Nur noch eine Operation: Addition
- Achtung: Auf der Hardware kommt noch die Byte-Order hinzu!

  → Wichtig, wenn Speicher direkt ausgelesen wird

Typumwandlung

- Bisher: Darstellung von Datentypen in Hardware
- Es fehlt: Umwandlung von Datentypen
- In der CPU? Egal. Abfolge von Bits von Bytes
- Interpretation der Bitfolge geschieht auf höherer Ebene nach **fest definierten Regeln**

Typumwandlung

- Bereits bekannt: **Implizite** Umwandlung (automatisch) in einen Datentyp

  → Skalar zu booleschem Ausdruck bei if-Blöcken

- Alternativ: `float` zu `int` bei Zuweisung (und umgekehrt)

  ```
  float f = 3;
  ```

- Es fehlt: **explizite** Umwandlung (manuell) in einen Datentyp, z. B. `int` zu `double`

Typumwandlung

- In C gibt es Type Casting

  → Bezeichnung für **explizite** Umwandlung

- Syntax ist einfach
  - Zieltyp in Klammern davor schreiben
  - `(T) 3; // Umwandlung in Typ T`

- Beispiel: `(int) 3.5;`

- Besonders interessant bei komplexen Datentypen und bei Zuweisung von Speicher (→ nächstes Kapitel)

C Run ▶

```c
#include <stdio.h>

int main() {
    int i = 13;
    double d = (double) i;
    printf("%#.02f\n", d);
    return 0;
}
```

Typumwandlung

🚨 Achtung: Unerwartete Fehler bei Zwischenschritten durch Umwandlungen

C Run ▶

```c
#include <stdio.h>
int main() {
    int i = 13, j = 4;
    double d = i / j;
    printf("%#.02f\n", d);
    return 0;
}
```

Typumwandlung

Warum geschieht das?

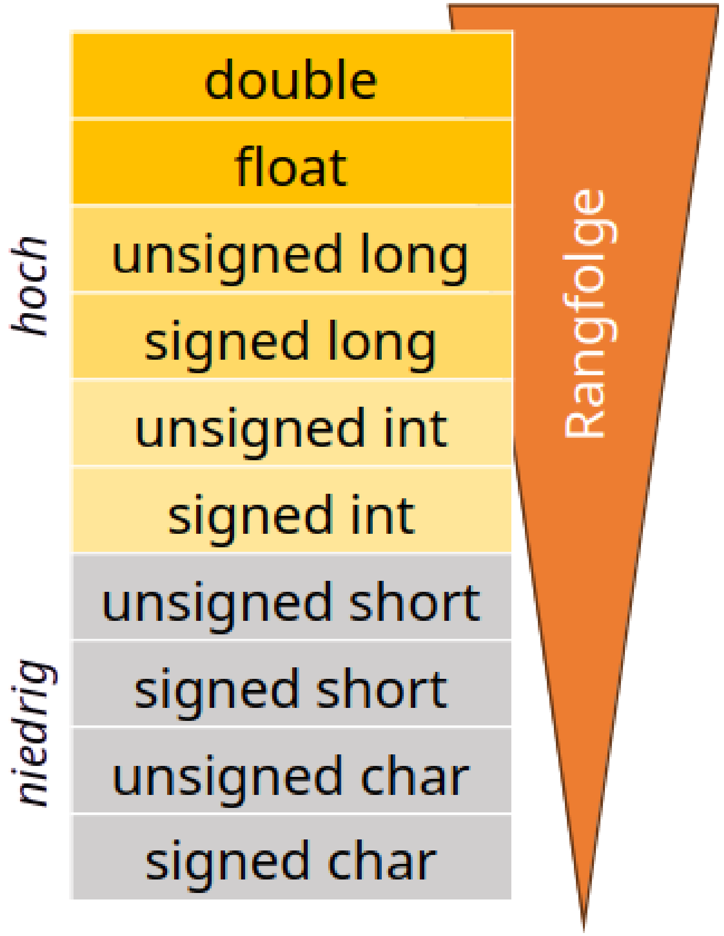
C Run ▶

```c
#include <stdio.h>
int main() {
    int i = 13, j = 4;
    double d = (double)i / j;
    printf("%#.02f\n", d);
    return 0;
}
```

- Implizite Umwandlung gilt auch für Zwischenschritte
- Division ist abgeschlossene Operation zweier Integers
- Keine Konvertierung **während** Division
  - Ergebnis Division ist ein `int`
  - `double` -Umwandlung erst bei der Zuweisung
- Lösung: **explizites** Type-Casting: `double d = (double) i / j;`

Weitere Typumwandlungen

- Upcasting
  - Abhängig von Parametern werden einzelne Variablen zu breiterem Datentyp umgewandelt
  - Festgelegte Hierarchie für Umwandlung
  - `int = char + int;` → `int = int + int;`
- Downcasting
  - Abhängig vom Ergebnistyp werden Variablen/das Ergebnis zu Typ mit weniger Bits konvertiert
  - `char = int + int;` → `char = (char)(int + int);`



Typumwandlung in C++

- In C++ prinzipiell ähnlich, **aber** einige wichtige Unterschiede
- Explizite Umwandlung in C++ nicht gern gesehen:
  - Geschieht unbedingt
  - Untergräbt das Typensystem des Compilers
- Compiler überprüft in C++ bereits mögliche Fehler oder Probleme
  - In C treten Fehler erst zur Laufzeit auf
  - C++ hat feiner unterteilte Möglichkeiten zur Typenumwandlung
  - Verteilt Funktionalität von C-Style-Casting auf verschiedene Funktionen

Typumwandlung in C++

- `static_cast`
  - Konvertierung unter Berücksichtigung impliziter Regeln
  - `double d = static_cast<double>(3);`
- `reinterpret_cast`:
  - Ähnlich zu C-Style-Cast, Uminterpretierung auf Bitebene (**gefährlich**)
  - `int i = reinterpret_cast<int>(3.5f);`
- (`const_cast`)
- (`dynamic_cast`)

Fixed-Size-Typen

- Erinnerung an Integer

  🚨 Breite in Bits ist abhängig von der Hardwarearchitektur! 🚨

- Unerwartete Effekte bei *Type Casting* möglich

  Besonders bei `int` auf unterschiedlichen CPU-Architekturen

- **Dringende** Empfehlung: Stattdessen Datentypen mit fester Breite verwenden

  → `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`

  → `int8_t`, `int16_t`, `int32_t`, `int64_t`

- Einbindung über den Header `<stdint.h>`

```
int32_t i32 = 0;
uint8_t ui8 = 0;
```

Zusammenfassung

- IEEE-754 zur Darstellung für Gleitkommazahlen
  - Probleme bei Gleitkommazahlen
  - *Fixed-Point*-Darstellung
- Zahlendarstellungen
  - Sign-Magnitude
  - Einerkomplement
  - **Zweierkomplement**
- Implizite Typumwandlung
  - *Down Casting*
  - *Up Casting*
- Explizite Typumwandlung
- *Fixed-Size*-Typen