

Kapitel 06 - Boolesche Logik und Bitoperationen

Peter Ulbrich

AG Systemsoftware

Veranstaltungswebseite

Rückblick

- Bisher kennengelernt:
 - **Binärformat**
 - Wertigkeit der Bits im Binärsystem
 - Verteilung der Bits auf mehrere Bytes für Zahlen ≥ 255
 - Verschiedene Stellenwertsysteme (Dezimal, Hexadezimal, Binär, Oktal)
 - **Boolesche Operatoren**
 - *AND, OR, NOT*
- Jetzt:
 - **Boolesche Logik** (Interpretation/Bedeutung von Binärwerten)
 - **Bitweise Operatoren/Operationen**

Rückblick - Binärformat

- Computer speichert Informationen im Binärformat:
1 oder 0 (= genannt Bit)
- Kombination von mehreren Ziffern bildet Binärzahl: 1001
- Jede Stelle der Binärzahl steht für eine Zweierpotenz:
 $2^0 = 1, 2^1 = 2, 2^2 = 4, 2^3 = 8, \dots$
- Beispiel
 - Darstellung der Zahl 79 = 64 + 8 + 4 + 2 + 1

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

Rückblick - Binärformat

- Zahlen > 255 werden genauso dargestellt, bestehen aber aus mehreren Bytes
- Beispiel: $591 = 512 + 79$

512	256	128	64	32	16	8	4	2	1
1	0	0	1	0	0	1	1	1	1

- $591 = 0000\ 0010 \times 256 + 0100\ 1111$

Rückblick - Binärformat

- Byte mit dem höchsten/niedrigsten Wert heißt *Most/Least Significant Byte* (*MSB* bzw. *LSB*)
- Reihenfolge der Bytes heißt *Byte Order*: Entweder von MSB zu LSB (*Big Endian*) oder von LSB zu MSB (*Little Endian*)
- Analog für Bits: *Most/Least Significant Bit* (*MSBit* bzw. *LSBit*)

128	64	32	16	8	4	2	1
0 (MSBit)	1	0	0	1	1	1	1 (LSBit)

Rückblick - Binärformat

-  **Achtung:** Zahlen können gleich aussehen, aber in unterschiedlichen Zahlensystemen stehen
- Beispiel: **110** (Binär) $\neq$ 110 (Dezimal)
- Unterscheidung bei gemischten Zahlensystemen über **Index**: $110_B = 6_D$
- Gibt mehrere gängige Zahlensysteme:
 - Dezimal $\rightarrow$ **D** oder dec
 - Binär $\rightarrow$ **B** oder bin
 - Hexadezimal $\rightarrow$ **H** oder hex
 - Oktal $\rightarrow$ **O** oder oct

Rückblick - Wahrheitswerte

- Ein Ausdruck, der zu `true` oder `false` evaluieren kann, heißt *boolescher Ausdruck*

Logischer Datentyp (`bool`)

- Zum Speichern von Wahrheitswerten „wahr“ und „falsch“
- Wertevorrat: `true` und `false`
- Datendefinition: `bool b;`
- Zuweisung: `b = true;`

Logische Operatoren

Den Umgang mit boolescher Logik lernen wir jetzt kennen!

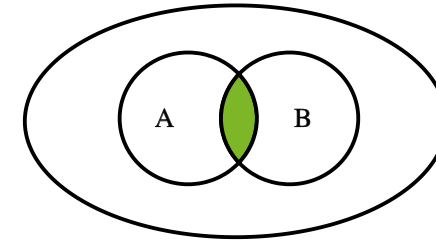
Name	Symbol	Beispiel
AND	<code>&&</code>	<code>b && (x < 7)</code>
OR	<code> </code>	<code>b x > 8</code>
Negation (NOT)	<code>!</code>	<code>!b</code>

Operatoren der booleschen Algebra

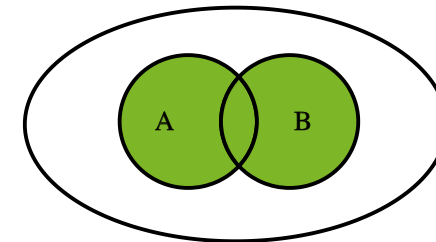
Notationen der Logikoperatoren

Operation	Bool. Algebra	C
UND	$A \wedge B$	<code>A && B</code>
ODER	$A \vee B$	<code>A B</code>
NEGATION	$\overline{A}$	<code>!A</code>

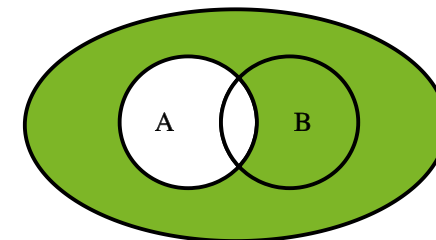
$A \wedge B$



$A \vee B$



$\overline{A}$

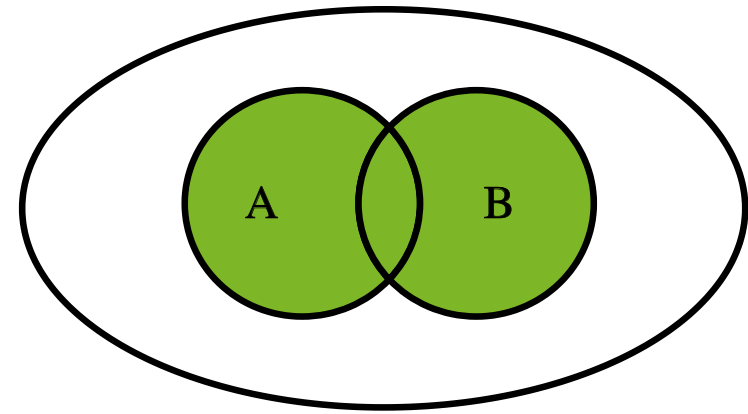


Wahrheitstabelle Oder

- OR: true, sobald **mindestens einer** der beiden Teilausdrücke wahr ist

A	B	$A \vee B$
0	0	0
1	0	1
0	1	1
1	1	1

$$\mathbf{A \vee B}$$

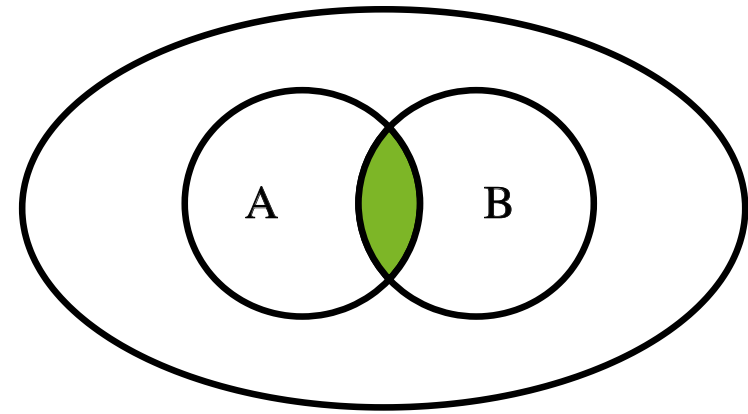


Wahrheitstabellen Und

- *AND*: true , sobald *beide* Teilausdrücke wahr sind

A	B	$A \wedge B$
0	0	0
1	0	0
0	1	0
1	1	1

$A \wedge B$

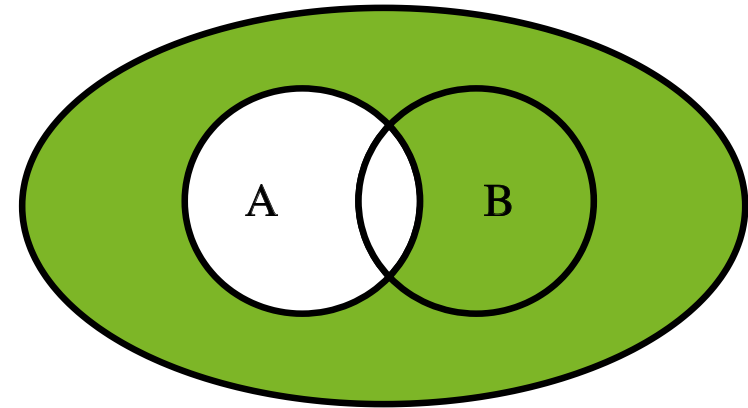


Wahrheitstabellen Negation

- NOT: *Gegenteil* des ursprünglichen Werts

A	$\overline{A}$
0	1
1	0

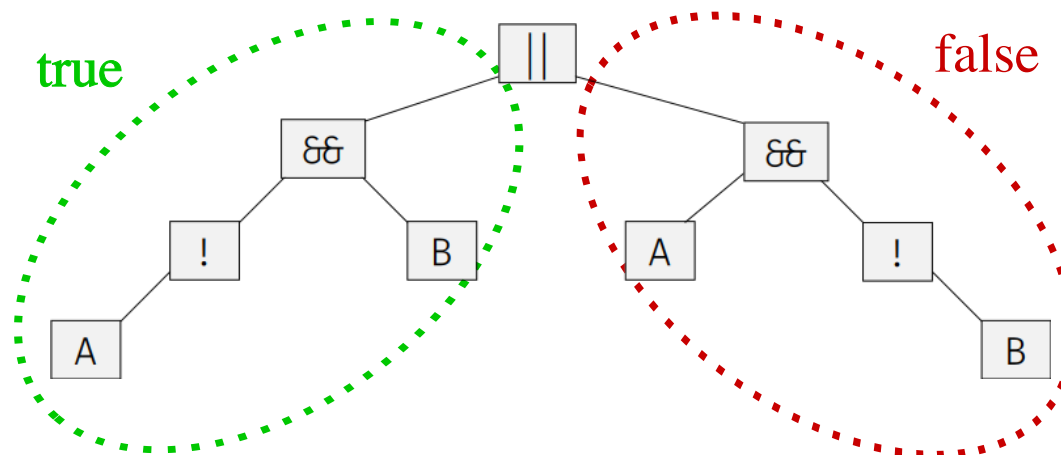
$\overline{A}$



Boolesche Algebra

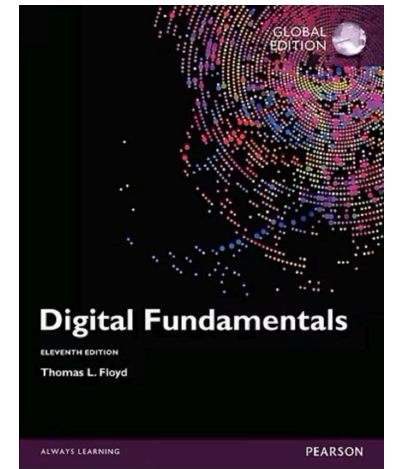
- Boolesche Ausdrücke können natürlich **mehr als zwei Variablen** beinhalten
→ **Schrittweise Evaluierung** von links nach rechts
- **Setzt Klammern**, um eindeutige Auswertungsreihenfolge zu erzwingen → **Lesbarkeit**
- **Beispiel:** $(!A \ \&\& \ B) \ || \ (A \ \&\& \ !B)$

`bool A = false; bool B = true;`



Regeln der booleschen Algebra

- Erlauben **Umformen** von booleschen Ausdrücken
- Ermöglichen **Vereinfachung** von **langen und komplexen** booleschen **Ausdrücken**
- Gesetze sind (meist) einfach zu verstehen
- Ergeben sich intuitiv aus der Mengenlehre
- **Anmerkung:** Anschauliche, vertiefende Literatur mit vielen Übungsaufgaben ist
 - *Thomas L. Floyd - Digital Fundamentals* [1]
 - Gedacht für Elektrotechniker, auf Englisch



Regeln der booleschen Algebra

Kommutativgesetz

- $A \vee B = B \vee A$
- $A \wedge B = B \wedge A$

Assoziativgesetz

- $(A \vee B) \vee C = A \vee (B \vee C)$
- $(A \wedge B) \wedge C = A \wedge (B \wedge C)$

Distributivgesetz

- $A \wedge (B \vee C) = (A \wedge B) \vee (A \wedge C)$
- $A \vee (B \wedge C) = (A \vee B) \wedge (A \vee C)$

- OR kann wie eine **Addition** gesehen werden
→ Alt. Schreibweise: $A \vee B = A + B$
- AND wie eine **Multiplikation**
→ Alt. Schreibweise: $A \wedge B = AB$
- Es gilt: „**Punkt vor Strich**“
- Erleichtert das „Rechnen“ mit booleschen Ausdrücken

Regeln der booleschen Algebra

- Darüber hinaus gelten auch noch weitere Regeln:

	Bool. Algebra	C
1.	$A \wedge 1 = A$	$(A \ \&\& \ \text{true}) \rightarrow A$
2.	$A \wedge 0 = 0$	$(A \ \&\& \ \text{false}) \rightarrow \text{false}$
3.	$A \vee 0 = A$	$(A \ \ \text{false}) \rightarrow A$
4.	$A \vee 1 = 1$	$(A \ \ \text{true}) \rightarrow \text{true}$
5.	$A \vee A = A$	$(A \ \ A) \rightarrow A$
6.	$A \vee \overline{A} = 1$	$(A \ \ !A) \rightarrow \text{true}$

Regeln der booleschen Algebra

	Bool. Algebra	C
7.	$A \wedge A = A$	$(A \ \&\& \ A) \rightarrow A$
8.	$A \wedge \bar{A} = 0$	$(A \ \&\& \ !A) \rightarrow \text{false}$
9.	$\overline{\bar{A}} = A$	
10.	$A \vee (A \wedge B) = A$ $\Rightarrow A(1 \vee B)$ $= A \wedge 1$ $= A$	Regel 4: $1 \vee B = 1$
11.	$A \vee \bar{A}B = A \vee B$	$(A \ \ (!A \ \&\& \ B)) \rightarrow (A \ \ B)$

Die Herleitung von Regel 11
schauen wir uns einmal als
Beispiel an

Beispiel - Herleitung einer Regeln zur booleschen Algebra

Herleitung: $A \vee \overline{A}B = A \vee B$

$$A + \overline{A}B$$

$$= (A + AB) + \overline{A}B$$

Regel 10: $A = A + AB$

$$= (AA + AB) + \overline{A}B$$

Regel 7: $A = AA$

$$= AA + AB + A\overline{A} + \overline{A}B$$

Hinzufügen von: $A\overline{A} = 0$

$$= (A + \overline{A})(A + B)$$

Faktorisierung

$$= 1(A + B)$$

Regel 6: $A + \overline{A} = 1$

$$= A + B$$

Regeln der booleschen Algebra

- Ein zweites Beispiel: $(A \vee B)(A \vee C) = A \vee BC$

$$(A + B)(A + C)$$

$$= AA + AC + AB + BC$$

Distributivgesetz

$$= A + AC + AB + BC$$

Regel 7: $A = AA$

$$= A(1 + C) + AB + BC$$

Faktorisierung (Distributivgesetz)

$$= A + AB + BC$$

Regel 4: $1 + C = 1$

$$= A(1 + B) + BC$$

Faktorisierung (Distributivgesetz)

$$= A + BC$$

Regel 4: $1 + B = 1$

De-Morganschen Regeln

- Abschließend gibt es noch zwei wichtige Regeln nach De-Morgan:

	Bool. Algebra	C
1.	$\overline{AB} = \overline{A} \vee \overline{B}$	<code>!(A && B) → !A !B</code>
2.		<code>!(A B) → !A && !B</code>

Das war jetzt genug Theorie.
Zurück zu C!

Die De-Morganschen

$$\overline{ABC} = \overline{A} \vee \overline{B} \vee \overline{C}$$

Short-Circuiting

- In C gibt es einen Mechanismus namens *Short-Circuiting* für *AND* und *OR*
- Boolesche Teilausdrücke werden *von links nach rechts* evaluiert
- Steht Gesamtergebnis schon *nach ersten Teilausdrucks* fest → *keine weitere Auswertung*

Beispiel - *Short-Circuiting*

- *Short-Circuiting* für OR:

Linker Ausdruck ist **true**

```
if (true || ((A && B) || !C)) {  
    run_critical_function();  
} else {  
    // ...
```

- Short-Circuiting für AND:

Linker Ausdruck ist **false**

```
if (false && (A || B)) {  
    run_critical_function();  
} else {  
    // ...
```

Vorteile von *Short-Circuiting*

- Short-Circuiting bietet **Geschwindigkeitsvorteile**:
Lohnenswert, unkomplizierte Ausdrücke als erstes zu evaluieren
- **Beispiel:**

```
bool a = true;  
if (a || long_complicated_expression())  
    do_stuff();  
} else {  
    // ...  
}
```

Und nun betrachten wir noch die
bitweisen Operationen

Bitweise Operationen

- **Gleichen Operationen** wie zuvor für die **Manipulation von Bits**
- Wichtig: **Separate** Evaluierung für **jedes Bit**
- Verkettung von Operationen auch hier möglich

(Klammern für Leserlichkeit und Operator-Präzedenzen beachten!)

Operation	Operator
<i>OR</i>	
<i>AND</i>	&
<i>NOT</i>	~
<i>XOR</i>	^
Links-Shift	<<
Rechts-Shift	>>

(Bitweise Operatoren)

Bitweises Oder

- Wende OR-Operation separat auf **jedes Bit** zweier Zahlen an
 - **Mindestens** ein Bit 1 → Ergebnis ist auch 1
 - Nur 0, falls beide Bits Wert 0 haben
- **Beispiele:**

A	B	A B
0	0	0
1	0	1
0	1	1
1	1	1
0011	1100	1111
1010	1001	1011

$$\begin{array}{r} 0011 \text{ A} \\ + 1100 \text{ B} \\ \hline 1111 \\ \hline \end{array}$$



Bitweises Und

- Wende *AND*-Operation separat **jedes Bit** zweier Zahlen an
- Sind **beide** Bits 1
→ Ergebnis ist auch 1

- **Beispiele:**

A	B	A & B
0	0	0
1	0	0
0	1	0
1	1	1
0011	1100	0000
1010	1001	1000

$$\begin{array}{r} A \ 1010 \\ B \ 1001 \\ \hline 1000 \end{array}$$


Bitweise Negation

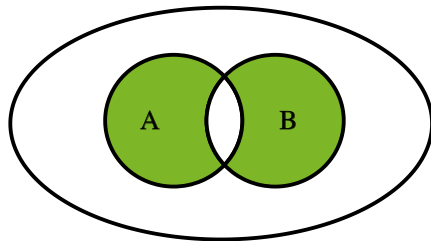
- Invertiere den Wert jedes Bits einer Zahl

- Beispiele:

A	$\sim A$
0	1
1	0
0011	1100
1010	0101

Bitweises exklusives Oder

- Ist **genau ein** Bit 1 → Ergebnis ist 1
- **Anders ausgedrückt:**
Bei **Vereinigung zweier Mengen** darf ein Element **nur in einer** Menge existieren
- **Häufige Notation:** $A \oplus B$
- **XOR ist reversibel:** $(A \oplus B) \oplus B = A$



A 1010
B 1001

0011

- **Beispiele:**

A	B	$A \wedge B$
0	0	0
1	0	1
0	1	1
1	1	0
0011	1100	1111
1010	1001	0011
0011	1001	1010



Anwendung des exklusiven Oders

- Exklusives Oder ist **sehr nützliche** und **viel benutzte** Operation

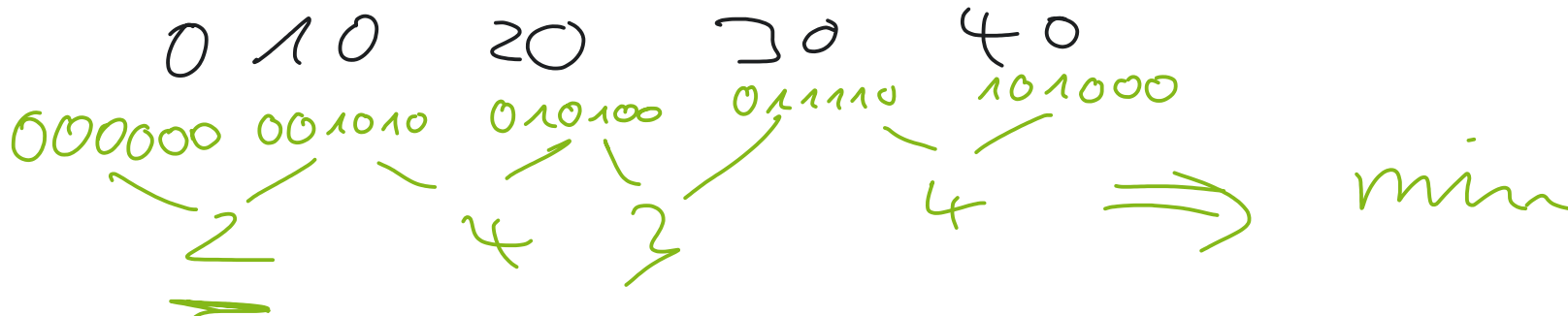
- Beispiel Kryptographie**

- Passwort P in Bitmuster übersetzen und abschnittsweise auf Text T anwenden
→ Nicht lesbares Ergebnis $T \oplus P$
- Entschlüsselung: $(T \oplus P) \oplus P = T$

key werte

3	102
17	98
8	35

- Hamming-Distanz:** Bestimmung der Anzahl verschiedener Bits
- Hashing:** Erzeugen einer eindeutigen Signatur (Zwischenschritt, z.B. SHA-3)



Beispiel - Exklusives Oder

- Verschlüsselung mit fester Passphrase

C Run ▶

```
#include <stdio.h>

int main() {
    int msg = 0x07; // 0000 0111
    int passwd = 0xAA; // 1010 1010
    int enc = msg ^ passwd; // 1010 1101
    printf("%#X\n", enc);
    // Umkehren zu Klartext
    enc ^= passwd; // 0000 0111
    printf("%#X\n", enc);
    return 0;
}
```

Bitschieben - *Shifting*

- Verschieben des **gesamten Bitmusters** um **n Bits** nach links bzw. rechts

- **Bei fester Bitbreite**

- Links-Shift: n MSBits entfernt
- Rechts-Shift: n LSBits entfernt

- **Mathematisches Äquivalent**

- Links-Shift: **Multiplikation** um 2^n
- Rechts-Shift: **Division** durch 2^n mit Abrundung

- **Beispiele:**

A	n	A << n
0010	0	0010
0010	1	0100
0010	2	1000

A	n	A >> n
0101	0	0101
0101	1	0010
0101	2	0001

Beispiel - Bit Shifting

- Was ist das Ergebnis (GCC-Compiler)?

cpp Run ▶

```
#include <stdio.h>
int main() {
    int i = 0x1;                // 0001
    printf("%d, %x\n", i, i);
    printf("%d, %x\n", i << 31, i << 31);
    return 0;
}
```

A: undefiniertes Verhalten

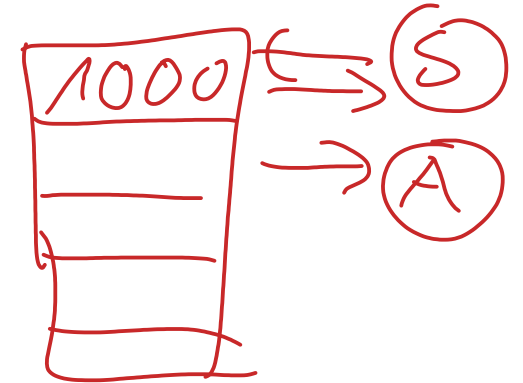
B: 0

C: -2 147 483 648 (INT_MIN)

D: -1

Bitmanipulationen

- **Häufiger Anwendungsfall:** Gezielte Manipulation eines oder mehrerer Bits
- **Beispiele:**
 - Hardwareprogrammierung
 - Einzelne Bits eines Speicherregisters stehen für verschiedene Funktionen (*Flags*)
 - Aktivierung über Setzen des Bits (*An-Aus-Schalter*)
 - Kommunikation
 - Nachricht mit fester Byte-Länge
 - Informationen sind an fester Stelle hinterlegt
- **Etablierte Muster:** Setzen (*Set*), Löschen (*Clear*), Flippen (*Toggle*) und Prüfen (*Check*)



Bits setzen

- **Setzen** eines Bits ist unkompliziert über die **OR-Operation** möglich
- In Kombination mit **Links-Shift**: Setzen eines **beliebigen Bits**

cpp Run ►

```
#include <stdio.h>
int main() {
    int bit_pattern = 0x07;           // 0111
    bit_pattern = bit_pattern | (1 << 3);
    printf("%#01x\n", bit_pattern);
    return 0;
}
```

1 1000
1111

1111

Achtung: Wegen Operator-Präzedenz Klammern beachten!

Löschen eines Bits

- Löschen eines Bits geschieht mithilfe von Links-Shift, *AND* und *NOT*
- Idee
 - Links-Shift einer 1 bestimmt das Bit
 - Negierung des Ergebnisses → nur ein Bit mit 0
 - Anwenden mit *AND*

Beispiel - Löschen eines Bits

Beispiel: Bit 2 soll gelöscht werden

1. **Links-Shift** um 2 mit $(1 \ll 2)$

Bitmuster: $\dots 0 \ 0100$

2. **Negierung:** $\sim(1 \ll 2)$

Bitmuster: $\dots 1 \ 1011$

3. **AND:** $\text{bit_pattern} \ \& \ \sim(1 \ll 2)$

Beispiel - Löschen eines Bits

cpp Run ▶

```
#include <stdio.h>
int main() {
    int bit_pattern = 0x07;           // 0111
    bit_pattern = bit_pattern & ~(1 << 2);
    printf("%#01x\n", bit_pattern);
    return 0;
}
```

1011
—
0011

Umkehren eines Bits

- Umkehren (*Flip*) eines Bits: Links-Shift in Kombination mit XOR

cpp Run ►

```
#include <stdio.h>
int main() {
    int bit_pattern = 0x07;           // 0111
    bit_pattern = bit_pattern ^ (1 << 1);
    printf("%#01x\n", bit_pattern);
    return 0;
}
```

0010
0101

Überprüfen eines Bits

- Überprüfen eines Bits: Links-Shift in Kombination mit **AND**

cpp Run ►

```
#include <stdio.h>
int main() {
    int bit_pattern = 0x07; // 0111
    printf("Is set: %#01x\n", bit_pattern & (1 << 0));
    return 0;
}
```

& 0001
0001

Überprüfen eines Bits

- Überprüfung darf auch als Bedingung verwendet werden

cpp Run ►

```
#include <stdio.h>
int main() {
    int bit_pattern = 0x07;                // 0111
    if (bit_pattern & (1 << 0)) {
        printf("Bit 1 is set\n");
    } else {
        printf("Bit 1 is not set\n");
    }
    return 0;
}
```

Typische Verwendung von Bitoperationen

- Szenario
 - Senden von Nachricht mit fester Länge
 - Informationen sind fester Stelle der Nachricht zu hinterlegen
- Beispiel
 - 64 Bit lange Nachricht
 - Bits 16-23 kodieren gemessene Temperatur

Typische Verwendung von Bitoperationen

cpp Run ▶

```
#include <stdint.h>
#include <stdio.h>

int main() {
    uint64_t message = 0x0;
    uint32_t temperature = 33; // 100001
    message |= (temperature << 16) & 0xFF0000;
    printf("temperature = %#01x, message = %#011x\n",
        temperature, message);
    return 0;
}
```

Byte-Reihenfolge (*Endianness*)

- Beschreibt **Reihenfolge**, wie **Bytes im Speicher** abgelegt werden
- **Ausgangssituation**: Eine Zahl mit **mehreren Bytes** → z. B. **int**, 4 Bytes
- **Big Endian**

32	16	8	4	2	1
0 (MSBit)	0	1	1	1	1 (LSBit)

- **Little Endian**

1	2	4	8	16	32
1 (LSBit)	1	1	1	0	1 (MSBit)

Byte-Reihenfolge (*Endianness*)

- Warum existiert diese Unterscheidung?
 - *Big Endian* ist *intuitiver* (Abbildung wie „klassische“ Zahlen)
 - *Little Endian* erlaubt eine *einfachere Umwandlung* in *Datentypen anderer Größe*
- *Beispiel Little Endian*

Byte 0	Byte 1	...
0x07	0x01	...

- Bei Interpretation bei Zahl als 1 Byte und als 2 Bytes:
 - Start bei gleicher Speicher-Adresse (Bit 0 in Byte 0 → „ganz links“)

Relevanz der Byte-Reihenfolge

- Verschiedene **Prozessorhersteller** legen für Produkte eine **Reihenfolge fest**
- Alternative Bezeichnung
 - **Motorola**-Reihenfolge (*Big Endian*)
 - **Intel**-Reihenfolge (*Little Endian*)
- Intel, AMD (CPUs für Desktop/Laptop) → Little Endian
- ARM (Mobile/Mikrocontroller) → Unterstützen beides
- **Keine Sorge:** Compiler übernimmt **automatisch** die Konvertierung
→ Wir müssen (erst mal) nichts tun! 😊

Zusammenfassung

- Komplexe Zustände mit Boolesche Algebra ausdrücken
- Boolesche Operationen für die praktische Umsetzung beim Programmieren
- *Short-Circuiting* vereinfachen Evaluation von booleschen Ausdrücken
- Bitweise Operationen erlauben Zugriff auf einzelne Bits Binärzahl
- Bitoperationen für das Bearbeiten von einzelnen Bits als unerlässliches Werkzeug
- Byte-Reihenfolge: Wichtig für verschiedene Anwendungszwecke

Referenzen

[1] T. L. Floyd, *Digital Fundamentals, Global Edition*, 11. Aufl. London, England: Pearson Education, 2014.

