

# 64-040 Modul IP7: Rechnerstrukturen

## 5. Boole'sche Algebra, Logische Operationen

Norman Hendrich

Universität Hamburg  
MIN Fakultät, Department Informatik  
Vogt-Kölln-Str. 30, D-22527 Hamburg  
hendrich@informatik.uni-hamburg.de

WS 2013/2014

# Inhalt

## 1. Boole'sche Algebra

Grundbegriffe der Algebra

Boole'sche Algebra

## 2. Logische Operationen

Bitweise logische Operationen

Schiebeoperationen

Anwendungsbeispiele

Literatur



## Wiederholung: Grundbegriffe der Algebra

- ▶ Mengen
- ▶ Relationen, Verknüpfungen
- ▶ Gruppe, Abel'sche Gruppe
- ▶ Körper, Ring
- ▶ Vektorraum
- ▶ usw.

## Nutzen einer (abstrakten) Algebra?!

Analyse und Beschreibung von

- ▶ gemeinsamen, wichtigen Eigenschaften
- ▶ mathematischer Operationen
- ▶ mit vielfältigen Anwendungen
  
- ▶ die Art der Elemente (z.B. ganze Zahlen, Aussagen, usw.)
- ▶ die Verknüpfungen (z.B. Addition, Multiplikation)
- ▶ zentrale Elemente (z.B. Null-, Eins-, inverse Elemente)
  
- ▶ Anwendungen: z.B. fehlerkorrigierende Codes auf CD/DVD

# Boole'sche Algebra

- ▶ George Boole, 1850: Untersuchung von logischen Aussagen mit den Werten *true* (wahr) und *false* (falsch)
- ▶ Definition einer Algebra mit diesen Werten
- ▶ Vier grundlegende Funktionen:
  - ▶ NEGATION (NOT)
  - ▶ UND
  - ▶ ODER
  - ▶ XOR
  
- ▶ Claude Shannon, 1937: Realisierung der Boole'schen Algebra mit Schaltfunktionen (binäre digitale Logik)

# Grundverknüpfungen

- ▶ zwei Werte: *wahr* (*true*, 1) und *falsch* (*false*, 0)
- ▶ vier grundlegende Verknüpfungen:

NOT(x)

x	
0	1
1	0

AND(x, y)

y	x	0	1
0	0	0	0
1	0	0	1

OR(x, y)

y	x	0	1
0	0	0	1
1	1	1	1

XOR(x,y)

y	x	0	1
0	0	0	1
1	1	1	0

- ▶ alle logischen Operationen lassen sich mit diesen Funktionen darstellen (*vollständige Basismenge*)

# Grundverknüpfungen

- ▶ zwei Werte,  $\{0, 1\}$
- ▶ insgesamt 4 Funktionen mit einer Variable  
 $f_0(x) = 0$ ,  $f_1(x) = 1$ ,  $f_2(x) = x$ ,  $f_3(x) = \neg x$
- ▶ insgesamt 16 Funktionen zweier Variablen
- ▶ allgemein  $2^{2^n}$  Funktionen von  $n$  Variablen
- ▶ später noch viele Beispiele

# Alle Funktionen von zwei Variablen

$x = 0\ 1\ 0\ 1$ $y = 0\ 0\ 1\ 1$	Bezeichnung	Notation	Alternativnotation	Java/C-Notation
0 0 0 0	Nullfunktion	0		0
0 0 0 1	AND	$x \cap y$		$x \& \& y$
0 0 1 0	Inhibition	$y > x$		$y > x$
0 0 1 1	Identität y	$y$		$y$
0 1 0 0	Inhibition	$x > y$		$x > y$
0 1 0 1	Identität x	$x$		$x$
0 1 1 0	XOR	$x \oplus y$	$x \neq y$	$x != y$
0 1 1 1	OR	$x \cup y$		$x    y$
1 0 0 0	NOR	$\neg(x \cup y)$		$!(x    y)$
1 0 0 1	Äquivalenz	$\neg(x \oplus y)$	$x = y$	$x == y$
1 0 1 0	NICHT x	$\neg x$	$x'$	$!x$
1 0 1 1	Implikation	$x \leq y$	$x \rightarrow y$	$y >= x$
1 1 0 0	NICHT y	$\neg y$	$y'$	$!y$
1 1 0 1	Implikation	$x \geq y$	$x \leftarrow y$	$x >= y$
1 1 1 0	NAND	$\neg(x \cap y)$		$!(x \& \& y)$
1 1 1 1	Einsfunktion	1		1



# Boole'sche Algebra

- ▶ Tupel  $\langle \{0, 1\}, |, \&, \neg, 0, 1 \rangle$  bildet eine Algebra
- ▶  $|$  ist die „Addition“
- ▶  $\&$  ist die „Multiplikation“
- ▶  $\neg$  ist das „Komplement“ (nicht das Inverse!)
- ▶ 0 (false) ist das Nullelement der Addition
- ▶ 1 (true) ist das Einselement der Multiplikation

# Rechenregeln: Ring / Algebra

Eigenschaft	Ring der ganzen Zahlen	Boole'sche Algebra
Kommutativgesetz	$a + b = b + a$ $a \times b = b \times a$	$a b = b a$ $a\&b = b\&a$
Assoziativgesetz	$(a + b) + c = a + (b + c)$ $(a \times b) \times c = a \times (b \times c)$	$(a b) c = a (b c)$ $(a\&b)\&c = a\&(b\&c)$
Distributivgesetz	$a \times (b + c) = (a \times b) + (a \times c)$	$a\&(b c) = (a\&b) (a\&c)$
Identitäten	$a + 0 = a$ $a \times 1 = a$	$a 0 = a$ $a\&1 = a$
Vernichtung	$a \times 0 = 0$	$a\&0 = 0$
Auslöschung	$-(-a) = a$	$\neg(\neg a) = a$
Inverses	$a + (-a) = 0$	—

# Rechenregeln: Ring / Algebra

Eigenschaft	Ring der ganzen Zahlen	Boole'sche Algebra
Distributivgesetz	—	$a (b\&c) = (a b)\&(a c)$
Komplement	—	$a   \neg a = 1$
	—	$a \& \neg a = 0$
Idempotenz	—	$a \& a = a$
	—	$a   a = a$
Absorption	—	$a (a\&b) = a$
	—	$a\&(a b) = a$
De-Morgan Regeln	—	$\neg(a \& b) = \neg a   \neg b$
	—	$\neg(a   b) = \neg a \& \neg b$

# De-Morgan Regeln

- ▶ Ersetzen von UND durch ODER und umgekehrt
- ▶ Austausch der Funktion und gleichzeitig Invertieren aller Ein- und Ausgänge
- ▶ alternative Schreibweise:  $\sim x$  für  $\neg x$
  
- ▶  $\sim(a \ \& \ b) = \sim a \ | \ \sim b$  NAND: NOT(AND(a,b))
- ▶  $\sim(a \ | \ b) = \sim a \ \& \ \sim b$  NOR: NOT(OR(a,b))
  
- ▶ wird beim Entwurf von Schaltungen häufig verwendet

## XOR: Exklusiv-Oder

- ▶ XOR-Funktion: entweder a oder b, a ungleich b
- ▶  $a \hat{=} b = (\sim a \ \& \ b) \mid (a \ \& \ \sim b)$   
genau einer von den Termen a und b ist wahr
- ▶  $a \hat{=} b = (a \mid b) \ \& \ \sim(a \ \& \ b)$   
entweder a ist wahr, oder b ist wahr, aber nicht beide gleichzeitig
- ▶  $a \hat{=} a = 0$

## Logische Operationen in Java und C

- ▶ eigener Datentyp?
  - ▶ Java: Datentyp boolean
  - ▶ C: implizit für alle Integertypen
  
- ▶ Vergleichsoperationen
- ▶ logische Grundoperationen
- ▶ Auswertungs-Reihenfolge/-prioritäten
  
- ▶ logische Operationen auch bitweise parallel möglich (s.u.)



# Vergleichsoperationen

- ▶  $a == b$      wahr, wenn  $a$  gleich  $b$
  - ▶  $a != b$      wahr, wenn  $a$  ungleich  $b$
  - ▶  $a >= b$     wahr, wenn  $a$  größer oder gleich  $b$
  - ▶  $a > b$       wahr, wenn  $a$  größer  $b$
  - ▶  $a < b$       wahr, wenn  $a$  kleiner  $b$
  - ▶  $a <= b$     wahr, wenn  $a$  kleiner oder gleich  $b$
- 
- ▶ Vergleich zweier Zahlen, Ergebnis ist logischer Wert
  - ▶ Java: Integerwerte alle im Zweierkomplement
  - ▶ C: Auswertung berücksichtigt signed/unsigned-Typen
  - ▶ Auswertung von links nach rechts, Klammerung berücksichtigt



# Logische Operationen in C

- ▶ drei **logische** Operatoren
- ▶ zusätzlich zu den Vergleichsoperatoren  $<$ ,  $<=$ ,  $==$ ,  $!=$ ,  $>$ ,  $>=$
  
- ▶  $!$  logische Negation
- ▶  $\&\&$  logisches UND
- ▶  $||$  logisches ODER
  
- ▶ der Zahlenwert 0 gilt als logische 0 (false)
- ▶ alle anderen Werte interpretiert als logische 1 (true)  
 $(a > b) || ((b != c) \&\& (b <= d))$
- ▶ völlig andere Funktion als die bitweisen Operationen (s.u.)



## Logische Operationen in C

- ▶ Integer mit Zahlenwert 0 gilt auch als logische 0 (false)
- ▶ alle anderen Werte interpretiert als logische 1 (true)
- ▶ völlig andere Semantik als in der Mathematik (!)
- ▶ *shortcut*-Auswertung: von links nach rechts

Beispiel	Wert
<code>!0x41</code>	<code>0x00</code>
<code>!0x00</code>	<code>0x01</code>
<code>!!0x00</code>	<code>0x00</code>
<code>0x69 &amp;&amp; 0x55</code>	<code>0x01</code>
<code>0x69    0x55</code>	<code>0x01</code>

## Logische Operationen in C: Beispiel

- ▶ nur Zahlenwert 0 gilt als logische 0 (false)
- ▶ alle anderen Werte interpretiert als logische 1 (true)

Beispiel:  $x = 0x66$  und  $y = 0x93$

Ausdruck (bitweise)	Wert	Ausdruck (logisch)	Wert
<code>x</code>	0110 0110	<code>x</code>	0000 0001
<code>y</code>	1001 0011	<code>y</code>	0000 0001
<code>x &amp; y</code>	0000 0010	<code>x &amp;&amp; y</code>	0000 0001
<code>x   y</code>	1111 0111	<code>x    y</code>	0000 0001
<code>~x   ~y</code>	1111 1101	<code>!x    !y</code>	0000 0000

## Logische Operationen in C: „Shortcut“-Auswertung

- ▶ logische Ausdrücke werden von links nach rechts ausgewertet
- ▶ Klammern werden natürlich berücksichtigt
- ▶ Abbruch, sobald der Wert eindeutig feststeht
  
- ▶  $(a \ \&\& \ 5/a)$  niemals Division durch Null. Der Quotient wird nur berechnet, wenn der linke Term ungleich Null ist.
- ▶  $(p \ \&\& \ *p++)$  niemals Nullpointer-Zugriff. Der Pointer wird nur verwendet, wenn  $p$  nicht Null ist.
  
- ▶ ternäre Abfrage:  
`condition ? true-expression : false-expression`
- ▶ Beispiel Absolutwert von  $x$ :  $(x < 0) ? -x : x$

# Logische Operationen in Java

- ▶ Java definiert eigenen Datentyp `boolean`
- ▶ elementare Werte `false` und `true`
- ▶ alternativ `Boolean.FALSE` und `Boolean.TRUE`
- ▶ **keine** Mischung mit Integer-Werten wie in C
  
- ▶ Vergleichsoperatoren `<`, `<=`, `==`, `!=`, `>`, `>=`
- ▶ Shortcut-Auswertung von links nach rechts
  
- ▶ ternäre Abfrage:  
`condition ? true-expression : false-expression`
- ▶ Beispiel Absolutwert von `x`: `(x < 0) ? -x : x`

# Bitweise logische Operationen

Integer-Datentypen doppelt genutzt:

- ▶ Zahlenwerte (Ganzzahl, Zweierkomplement, Gleitkomma)  
arithmetische Operationen (Addition, Subtraktion, usw.)
- ▶ Binärwerte mit  $w$  einzelnen Bits (Wortbreite  $w$ )
- ▶ Boole'sche Verknüpfungen bitweise auf allen  $w$  Bits  
 Grundoperationen: Negation, UND, ODER, XOR  
 Schiebe-Operationen (shift-left, rotate-right, usw.)

# Bitweise logische Operationen

- ▶ Integer-Datentypen interpretiert als Menge von Bits
- ▶ bitweise logische Operationen möglich
- ▶ es gibt insgesamt  $2^{2^n}$  Operationen mit  $n$  Operanden
  
- ▶ in Java und C: vier Operationen definiert:
  - ▶ Negation       $\sim x$     Invertieren aller einzelnen Bits
  - ▶ UND             $x \& y$     Logisches UND aller einzelnen Bits
  - ▶ OR              $x | y$      Logisches ODER aller einzelnen Bits
  - ▶ XOR             $x \wedge y$     Logisches XOR aller einzelnen Bits
  
- ▶ alle anderen Funktionen können damit dargestellt werden

## Bitweise logische Operationen: Beispiel

 $x = 0010 \ 1110$ 
 $y = 1011 \ 0011$ 
 $\sim x = 1101 \ 0001$  alle Bits invertiert

 $\sim y = 0100 \ 1100$  alle Bits invertiert

 $x \ \& \ y = 0010 \ 0010$  bitweises UND

 $x \ | \ y = 1011 \ 1111$  bitweises ODER

 $x \ ^ \ y = 1001 \ 1101$  bitweises XOR

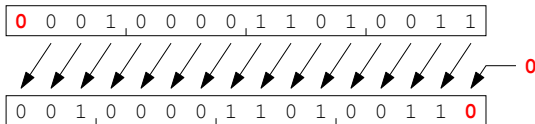
# Schiebeoperationen

- ▶ als Ergänzung der bitweisen logischen Operationen
- ▶ für alle Integer-Datentypen verfügbar
- ▶ fünf Varianten:
  - ▶ shift-left
  - ▶ logical shift-right
  - ▶ arithmetic shift-right
  - ▶ rotate-left
  - ▶ rotate-right
  
- ▶ Schiebeoperationen in Hardware leicht zu realisieren
- ▶ auf fast allen Prozessoren im Befehlssatz



## Shift-Left (shl)

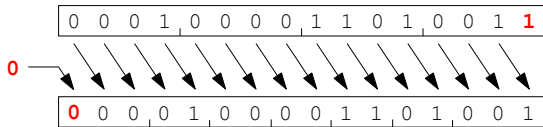
- ▶ Verschieben der Binärdarstellung von  $x$  um  $n$  bits nach links
- ▶ links herausgeschobene  $n$  bits gehen verloren
- ▶ von rechts werden  $n$  Nullen eingefügt



- ▶ in Java und C direkt als Operator  $x \ll n$
- ▶ shl um  $n$  bits entspricht Multiplikation mit  $2^n$

## Logical Shift Right (sr1)

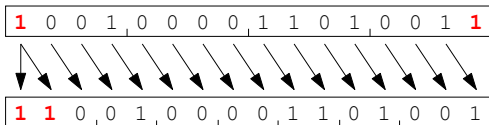
- ▶ Verschieben der Binärdarstellung von  $x$  um  $n$  bits nach rechts
- ▶ rechts herausgeschobene  $n$  bits gehen verloren
- ▶ von links werden  $n$  Nullen eingefügt



- ▶ in Java direkt als Operator verfügbar:  $x \ggg n$
- ▶ in C nur für unsigned-Typen als Operator:  $x \gg n$
- ▶ in C für signed-Typen nicht als Operator

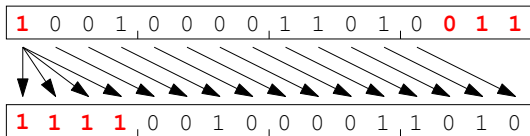
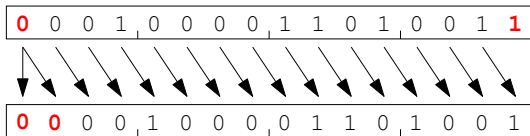
## Arithmetic Shift Right (sra)

- ▶ Verschieben der Binärdarstellung von  $x$  um  $n$  bits nach rechts
- ▶ rechts herausgeschobene  $n$  bits gehen verloren
- ▶ von links wird  $n$ -mal das MSB (Vorzeichenbit) eingefügt
- ▶ Vorzeichen bleibt dabei erhalten (auch Zweierkomplement)



- ▶ in Java direkt als Operator verfügbar:  $x \gg n$
- ▶ in C nur für signed-Typen:  $x \gg n$
- ▶ sra um  $n$  bits ähnlich der Division durch  $2^n$

## Arithmetic Shift Right: Beispiel



$x \gg 1$  aus  $0x10D3$  (4307) wird  $0x0869$  (2153)

$x \ggg 3$  aus  $0x90D3$  (-28460) wird  $0xF21A$  (-3558)

## Arithmetic Shift Right: Division durch Zweierpotenzen?

- ▶ positive Werte:  $x \gg n$  entspricht Division durch  $2^n$
- ▶ negative Werte:  $x \gg n$  Ergebnis zu klein (gerundet in Richtung negativer Werte statt in Richtung Null)

1111 1011 (-5)

1111 1101 (-3)

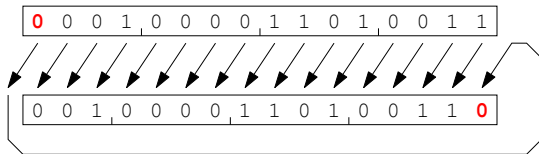
1111 1110 (-2)

1111 1111 (-1)

- ▶ C: Kompensation durch Berechnung von  $(x + (1 \ll k) - 1) \gg k$
- ▶ Details: Bryant & O'Hallaron

## Rotate Left (rol)

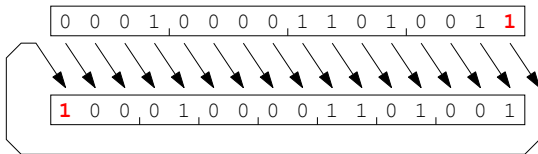
- ▶ Rotation der Binärdarstellung von  $x$  um  $n$  bits nach links
- ▶ herausgeschobene Bits werden von rechts wieder eingefügt



- ▶ in Java und C nicht als Operator verfügbar
- ▶ Java: `Integer.rotateLeft( int x, int distance)`

## Rotate Right (ror)

- ▶ Rotation der Binärdarstellung von  $x$  um  $n$  bits nach rechts
- ▶ herausgeschobene Bits werden von links wieder eingefügt



- ▶ in Java und C nicht als Operator verfügbar
- ▶ Java: `Integer.rotateRight( int x, int distance)`







## Beispiel: bit-set, bit-clear

Bits an Position  $p$  in einem Integer setzen oder löschen?

- ▶ Maske erstellen, die genau eine 1 gesetzt hat
- ▶ dies leistet  $(1 \ll p)$ , mit  $0 \leq p \leq w$  bei Wortbreite  $w$

```
public int bit_set( int x, int pos ) {  
    return x | (1 << pos); // mask = 0...010...0  
}
```

```
public int bit_clear( int x, int pos ) {  
    return x & ~(1 << pos); // mask = 1...101...1  
}
```

## Beispiel: ntohl/htonl Byte-Swapping

Linux: /usr/include/bits/byteswap.h

```
#  if __BYTE_ORDER == __LITTLE_ENDIAN
#  define ntohl(x)  __bswap_32 (x)
#  define ntohs(x) __bswap_16 (x)
#  define htonl(x) __bswap_32 (x)
#  define htons(x) __bswap_16 (x)
#  endif

...

/* Swap bytes in 32 bit value.  */
#define __bswap_32(x) \
    (((x) & 0xff000000) >> 24) \
    | (((x) & 0x00ff0000) >> 8) \
    | (((x) & 0x0000ff00) << 8) \
    | (((x) & 0x000000ff) << 24))
```

## Beispiel: RGB-Format für Farbbilder

Farbdarstellung am Monitor / Bildverarbeitung?

- ▶ Matrix aus  $w \times h$  Bildpunkten
- ▶ additive Farbmischung aus Rot, Grün, Blau
- ▶ pro Farbkanal typischerweise 8-bit, Wertebereich 0..255
- ▶ Abstufungen ausreichend für (untrainiertes) Auge
  
- ▶ je ein 32-bit Integer pro Bildpunkt
- ▶ typisch: 0x00RRGGBB oder 0xAARRGGBB
- ▶ je 8-bit für Alpha/Transparenz, rot, grün, blau
  
- ▶ `java.awt.image.BufferedImage(TYPE_INT_ARGB)`

## Beispiel: RGB-Rotfilter

```

public BufferedImage redFilter( BufferedImage src ) {
    int    w = src.getWidth();
    int    h = src.getHeight();
    int type = BufferedImage.TYPE_INT_ARGB;
    BufferedImage dest = new BufferedImage( w, h, type );

    for( int y=0; y < h; y++ ) { // alle Zeilen
        for( int x=0; x < w; x++ ) { // von links nach rechts
            int  rgb = src.getRGB( x, y ); // Pixelwert bei (x,y)
                                                    // rgb = 0xAARRGGBB
            int  red = (rgb & 0x00FF0000); // Rotanteil maskiert
            dest.setRGB( x, y, red );
        }
    }
    return dest;
}
    
```

## Beispiel: RGB-Graufilter

```

public BufferedImage grayFilter( BufferedImage src ) {
    ...
    for( int y=0; y < h; y++ ) { // alle Zeilen
        for( int x=0; x < w; x++ ) { // von links nach rechts
            int  rgb = src.getRGB( x, y );           // Pixelwert
            int  red  = (rgb & 0x00FF0000) >>>16; // Rotanteil
            int  green = (rgb & 0x0000FF00) >>> 8; // Grünanteil
            int  blue  = (rgb & 0x000000FF);       // Blauanteil

            int  gray = (red + green + blue) / 3; // Mittelung

            dest.setRGB( x, y, (gray<<16)|(gray<<8)|gray );
        }
    }
    ...
}
    
```

## Beispiel: Bitcount (mit while-Schleife)

Anzahl der gesetzten Bits in einem Wort?

- ▶ Anwendung z.B. für Kryptalgorithmen (Hamming-Distanz)
- ▶ Anwendung für Medienverarbeitung

```
public static int bitcount( int x ) {
    int count = 0;

    while( x != 0 ) {
        count += (x & 0x00000001); // unterstes bit addieren
        x = x >>> 1;             // 1-bit rechts-schieben
    }

    return count;
}
```

## Bitcount: Tree

- ▶ Algorithmus mit Schleife ist einfach aber langsam
- ▶ schnellere parallele Berechnung ist möglich
- ▶ siehe Übungsaufgabe 4.4
  
- ▶ Lösungshinweise und weitere Varianten:
- ▶ siehe Knuth AoCP 4.1, Abschnitt 7.1.3
- ▶ siehe `java.lang.Integer.bitCount()`
  
- ▶ viele neuere Prozessoren/DSPs: eigener bitcount-Befehl



## Tipps & Tricks: Rightmost bits

Integer  $x$  schreiben als  $(\alpha 0 1^a 10^b)_2$

- ▶ beliebiger Bitstring  $\alpha$ , eine Null, dann  $a + 1$  Einsen und  $b$  Nullen, mit  $a \geq 0$  und  $b \geq 0$ .
- ▶ (Ausnahme:  $x = -2^b$  läßt sich nicht so schreiben)

$$\bar{x} = (\bar{\alpha} 1 0^a 0 1^b)_2$$

$$x - 1 = (\alpha 0 1^a 0 1^b)_2$$

$$-x = (\alpha 1 0^a 1 0^b)_2$$

$$\bar{x} + 1 = -x = \overline{x - 1}$$



## Tipps & Tricks

Integer  $x$  schreiben als  $(\alpha 0 1^a 1 0^b)_2$

$$x \& (x - 1) = (\alpha 0 1^a 0 0^b)_2$$

$$x \& -x = (0^\infty 0 0^a 1 0^b)_2$$

$$x | -x = (1^\infty 1 1^a 1 0^b)_2$$

$$x \oplus -x = (1^\infty 1 1^a 0 0^b)_2$$

$$x | (x - 1) = (\alpha 0 1^a 1 1^b)_2$$

$$\bar{x} \& (x - 1) = (0^\infty 0 0^a 0 1^b)_2$$

letzte 1 entfernt

letzte 1 extrahiert

letzte 1 nach links verschmiert

letzte 1 entfernt und verschmiert

letzte 1 nach rechts verschmiert

letzte 1 nach rechts verschmiert

$$((x | (x - 1)) + 1) \& x = (\alpha 0 0^a 0 0^b)_2$$

remove rightmost run of 1s

## Literatur: Vertiefung

- ▶ D.E.Knuth, *The Art of Computer Programming, Volume 4, Fascicle 1, Bitwise Tricks & Techniques*, Addison-Wesley 2009
- ▶ Klaus von der Heide, *Vorlesung Technische Informatik T1*, Universität Hamburg, FB Informatik, 2004