

Algorithmen auf Sequenzen

Bitsequenzen

Sven Rahmann

Genominformatik
Universitätsklinikum Essen
Universität Duisburg-Essen
Universitätsallianz Ruhr

- Abhängig von der Maschinenarchitektur besteht ein Wort aus $W = \{16, 32, 64, 128\}$ Bits.
- Um n Bits zu speichern, braucht man also $\lceil n/W \rceil$ Wörter.
- Verschiedene Bitoperationen sind hardwareseitig implementiert.

Verschiedene Operationen sind neben arithmetischen Operationen auf der Bitsequenz s möglich.

Bitweise Negation $\sim s$ ist definiert als $(\sim s)[i] := \sim s[i]$, wobei

\sim	0	1
=	1	0

Bitweise Verundung $s \& t$ ist definiert als $(s \& t)[i] := s[i] \& t[i]$, wobei

$\&$	0	1	0	1
=	0	0	0	1

Bitweise Veroderung $s | t$ ist definiert als $(s | t)[i] := s[i] | t[i]$, wobei

$ $	0	1	0	1
=	0	1	1	1

Weitere Operationen sind:

Bitweise exklusive

Veroderung $s \oplus t$ ist definiert als

$(s \oplus t)[i] := s[i] \oplus t[i]$, wobei

	0	1	0	1
\oplus	0	0	1	1
$=$	0	1	1	0

Shiftoperation:

Wir betrachten auf den W Bits von $s = (s[k])_{k=0}^{W-1}$ nun die Operationen **Linksverschiebung** \ll

$$(s \ll b)[i] := \begin{cases} s[i - b] & \text{wenn } 0 \leq i - b \\ 0 & \text{sonst} \end{cases}$$

und **Rechtsverschiebung** \gg

$$(s \gg b)[i] := \begin{cases} s[i + b] & \text{wenn } i + b < W \\ 0 & \text{sonst} \end{cases}$$

um jeweils $b \geq 0$ Bits.

Multiplikation und ganzzahlige Division mit Zweierpotenzen:

Linksshift um b Bits

entspricht

Multiplikation mit 2^b :

$(s)_{10}$	s	b	$(s \ll b)_{10}$	$s \ll b$
5	101	1	10	1010
12	1100	2	48	110000

Multiplikation und ganzzahlige Division mit Zweierpotenzen:

Linksshift um b Bits

entspricht

Multiplikation mit 2^b :

$(s)_{10}$	s	b	$(s \ll b)_{10}$	$s \ll b$
5	101	1	10	1010
12	1100	2	48	110000

Rechtsshift um b Bits

entspricht

ganzzahliger Division
durch 2^b :

$(s)_{10}$	s	b	$(s \gg b)_{10}$	$s \gg b$
17	10001	1	8	1000
163	10100011	3	20	10100

Multiplikation und ganzzahlige Division mit Zweierpotenzen:

Linksshift um b Bits

entspricht

Multiplikation mit 2^b :

$(s)_{10}$	s	b	$(s \ll b)_{10}$	$s \ll b$
5	101	1	10	1010
12	1100	2	48	110000

Rechtsshift um b Bits

entspricht

ganzzahliger Division
durch 2^b :

$(s)_{10}$	s	b	$(s \gg b)_{10}$	$s \gg b$
17	10001	1	8	1000
163	10100011	3	20	10100

Achtung: Bits können „verschwinden“

$$\boxed{01001110} \ll 2 \quad \boxed{00111000}$$

Mit $b \geq W$ ist das Verhalten in vielen CPUs nicht definiert.

Modulorechnung (ganzzahliger Rest bei Division):

Ist der Divisor 2^b , werden nur die niederwertigsten b Bits beibehalten, die höherwertigen Bits werden auf 0 gesetzt.

$$(s \% 2^b)[i] := \begin{cases} s[i] & \text{wenn } i < b \\ 0 & \text{sonst} \end{cases}$$

Operation durch Verundung mittels einer Maske m möglich,

Beispiel $01001100 \% 2^5$:

$$\begin{array}{r} s \quad 01001100 \qquad \qquad \qquad 76_{10} \\ \& \quad m \\ \hline = \end{array}$$

Modulorechnung (ganzzahliger Rest bei Division):

Ist der Divisor 2^b , werden nur die niederwertigsten b Bits beibehalten, die höherwertigen Bits werden auf 0 gesetzt.

$$(s \% 2^b)[i] := \begin{cases} s[i] & \text{wenn } i < b \\ 0 & \text{sonst} \end{cases}$$

Operation durch Verundung mittels einer Maske m möglich,

Beispiel $01001100 \% 2^5$:

s	01001100		76_{10}
$\& m$	00011111	$\hat{=} (1 \ll 5) - 1$	31_{10}
<hr/>			
$=$	00001100		12_{10}

Manipulieren von Bits

Da man nur auf die Worte und nicht auf die Bits im RAM zugreifen kann, muss für das Bit an Index i zuerst das j -te Wort bestimmt werden, in dem es gespeichert ist. Sei dazu die Sequenz der Maschinenwörter.

$$B := (B[j])_{j=0}^{\lceil n/W \rceil - 1}, B[j] = s[jW : (j+1)W]$$

Zu gegebenem i ist $j = i // W = i \gg w$ mit $w = \lceil \log_2 W \rceil$.
Die Bitnummer innerhalb des Wortes ist $k = i \% W$
(modulo-Operation).

Das k -te Bit innerhalb eines Wortes kann gesetzt, gelöscht oder invertiert (toggle) werden.

- Setzen von Bits: $B[j] = B[j] \mid (1 \ll k)$
- Löschen von Bits: $B[j] = B[j] \& \sim(1 \ll k)$
- Invertieren von Bits: $B[j] = B[j] \oplus (1 \ll k)$

Population Count (popcount)

Die Funktion *popcount* zählt die Eins-Bits innerhalb eines Wortes.

- Auch: **population count**, **popcnt** Hamming-Gewicht.
- Popcount wird in verschiedenen Datenstrukturen und z.B. in Kryptographie-Algorithmen eingesetzt.
- Neue CPUs (Intel SSE4.2) bieten elementaren Popcount-Befehl.
- C-Funktion:

```
int _builtin_popcount(unsigned int x)
```

Naive Implementierung, alle Bits aufaddieren:

```
1 int popcount(uint64 x) {  
2     int count;  
3     for (count=0; x; x>>=1) count += x & 1;  
4     return count;  
5 }
```

Alternative: Look-Up-Tabelle:

```
1 int popcount_la[] = {0, 1, 1, 2, 1, 2, 2, 3, ...};
```

Naive Implementierung, alle Bits aufaddieren: Laufzeit $O(W)$.

```
1 int popcount(uint64 x) {  
2     int count;  
3     for (count=0; x; x>>=1) count += x & 1;  
4     return count;  
5 }
```

Alternative: Look-Up-Tabelle: Speicherplatz $O(2^W)$.

```
1 int popcount_la[] = {0, 1, 1, 2, 1, 2, 2, 3, ...};
```

Geht es effizienter?

Parallelisierung ausnutzbar, Wort wird in kleine Blöcke aufgeteilt.

- Werte innerhalb der Blöcke werden aufaddiert.
- Masken werden zur Separierung der Blöcke verwendet.
- Größe der Blöcke wird sukzessiv verdoppelt.

1) Bits innerhalb 2er Blöcke addieren, dazu 1er-Maske M_1 nutzen.

$$\begin{array}{r} \\ \\ \times \\ \hline \end{array}$$

1) Bits innerhalb 2er Blöcke addieren, dazu 1er-Maske M_1 nutzen.

	15		0													
x	0	1	0	1	1	1	1	0	0	1	0	0	1	1	0	1
M_1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
<hr/>																
$x \& M_1$	0	1	0	1	0	1	0	0	0	1	0	0	0	1	0	1
$+(x \gg 1) \& M_1$	0	0	0	0	0	1	0	1	0	0	0	0	0	0	1	0
<hr/>																

1) Bits innerhalb 2er Blöcke addieren, dazu 1er-Maske M_1 nutzen.

	15		0													
x	0	1	0	1	1	1	1	0	0	1	0	0	1	1	0	1
M_1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
<hr/>																
$x \& M_1$	0	1	0	1	0	1	0	0	0	1	0	0	0	1	0	1
$+(x \gg 1) \& M_1$	0	0	0	0	0	1	0	1	0	0	0	0	0	1	0	0
<hr/>																
$x =$	0	1	0	1	1	0	0	1	0	1	0	0	1	0	0	1

2) Bits innerhalb 4er Blöcke addieren, dazu 2er-Maske M_2 nutzen.

	15		0
x	01011001	01001001	
M_2	00110011	00110011	
<hr/>			
$x \& M_2$	00010001	00000001	
$+(x \gg 2) \& M_2$	00010010	00001001	
<hr/>			

2) Bits innerhalb 4er Blöcke addieren, dazu 2er-Maske M_2 nutzen.

	15		0
x	01011001	01001001	
M_2	00110011	00110011	
<hr/>			
$x \& M_2$	00010001	00000001	
$+(x \gg 2) \& M_2$	00010010	000010010	
<hr/>			
$x =$	00100011	000010011	

3) Bits innerhalb 8er Blöcke addieren, dazu 4er-Maske M_3 nutzen.

$$\begin{array}{r} \\ \\ \times \\ \hline \end{array}$$

3) Bits innerhalb 8er Blöcke addieren, dazu 4er-Maske M_3 nutzen.

	15	0
x	0010001100010011	
M_3	0000111100001111	
<hr/>		
$x \& M_3$	0000001100000011	
$+(x \gg 4) \& M_3$	0000001000000001	
<hr/>		

3) Bits innerhalb 8er Blöcke addieren, dazu 4er-Maske M_3 nutzen.

	15	0
x	0010001100010011	
M_3	0000111100001111	
<hr/>		
$x \& M_3$	0000001100000011	
$+(x \gg 4) \& M_3$	0000001000000001	
<hr/>		
$x =$	0000010100000100	

Popcount hat Laufzeit $\mathcal{O}(\log W)$. Man kann dies in C-Code mit einigen Tricks noch effizienter codieren¹; hier wird $W = 64$ angenommen.

```
1 const uint64 M1    = 0x5555555555555555;  
2 const uint64 M2    = 0x3333333333333333;  
3 const uint64 M3    = 0x0f0f0f0f0f0f0f0f;  
4 const uint64 H256  = 0x0101010101010101;  
5 int popcount(uint64 x) {  
6     x -= (x >> 1) & M1;  
7     x = (x & M2) + ((x >> 2) & M2);  
8     x = (x + (x >> 4)) & M3;  
9     return (x * H256) >> 56;  
10 }
```

¹Quelle: http://en.wikipedia.org/wiki/Hamming_weight