UA RUHR

Algorithmen auf Sequenzen

Bitsequenzen

Sven Rahmann

Genominformatik
Universitätsklinikum Essen
Universität Duisburg-Essen
Universitätsallianz Ruhr

Bitsequenzen

- Einfachste Form von Sequenzen.
- Alphabet besteht nur aus zwei Buchstaben, nämlich $\{0, 1\}$ oder $\{\texttt{false}, \texttt{true}\}$.
- Die Leserichtung geht vom *least significant bit* (LSB) zum *most significant bit* (MSB).

$$\leftarrow$$

(MSB)							(LSB)
1	0	1	0	0	0	1	0

Darstellung im Rechner

- 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.

Bitoperationen

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	1	1
$=$	0	0	0	1

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

	0	1	0	1
$\mid$	0	0	1	1
$=$	0	1	1	1

Bitoperationen

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

Bitoperationen

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.

Bitoperationen

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

Bitoperationen

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

Bitoperationen

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“

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

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

Bitoperationen

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		
	$=$		

Bitoperationen

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:

$$\begin{array}{rrlcr} & s & 01001100 & & 76_{10} \\ \& & m & 00011111 & \widehat{=}\ (1 \ll 5) - 1 & 31_{10} \\ \hline & = & 00001100 & & 12_{10} \end{array}$$

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 der Index j des Wortes bestimmt werden, in dem es gespeichert ist.
Sei dazu B die Sequenz der Maschinenwörter.

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

Zu gegebenem i ist $j = i // W = i \gg w$ mit $W = 2^w$.
Die Bitnummer innerhalb des Wortes ist $k = i \% W$ (modulo-Operation).

Manipulieren von Bits

UA RUHR

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)`

Popcount

Naive Implementierung, alle Bits aufaddieren:

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

Alternative: Look-Up-Tabelle:

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

Popcount

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

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

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

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

Geht es effizienter?

Popcount

Effiziente Berechnung in $\Theta(\log W)$ Schritten:

Bit-Parallelisierung ausnutzbar, Wort wird in Blöcke aufgeteilt:
anfangs 1-Bit-Blöcke, dann 2-Bit-, dann 4-Bit-, dann 8-Bit-, etc.

- Zähler von zwei benachbarten Blöcken werden addiert.
- Ergebnis hat im doppelt so großen Block Platz
- Additionen benachbarter Blöcke unabhängig, kein carry-over
- Shifts und &-Masken werden zur Alignierung der Blöcke verwendet.

Popcount

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

$$
\begin{array}{r l}
 & {\scriptstyle 15} \hspace{8em} {\scriptstyle 0} \\
x & 0101111001001101 \\
M_1 & 0101010101010101 \\
\hline
x \,\&\, M_1 & 0101010001000101 \\
+(x \gg 1) \,\&\, M_1 & 0000010100000100 \\
\hline
x = & 0101100101001001
\end{array}
$$

Popcount

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

	15 … 0
x	0101100101001001
M_2	0011001100110011
$x \mathbin{\&} M_2$	0001000100000001
$+(x \gg 2) \mathbin{\&} M_2$	0001001000010010
$x =$	0010001100010011

Popcount

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

$$
\begin{array}{rl}
 & {\scriptstyle 15}\qquad\qquad\qquad\qquad {\scriptstyle 0} \\
x & 0010001100010011 \\
M_3 & 0000111100001111 \\
\hline
x \mathbin{\&} M_3 & 0000001100000011 \\
+(x \gg 4) \mathbin{\&} M_3 & 0000001000000001 \\
\hline
x = & 0000010100000100
\end{array}
$$

Popcount

4) Die letzten beiden Blöcke addieren, dazu Maske M_4 nutzen.

$$
\begin{array}{rl}
 & {\scriptstyle 15} \hspace{13em} {\scriptstyle 0} \\
x & 0000010100000100 \\
M_4 & 0000000011111111 \\
\hline
x \mathbin{\&} M_4 & 0000000000000100 \\
+(x \gg 8) & 0000000000000101 \\
\hline
x = & 0000000000001001
\end{array}
$$

Popcount

Popcount hat Laufzeit $\mathcal{O}(\log W)$. Man kann dies in C-Code mit einigen Tricks noch effizienter codieren[1]; hier für $W = 64$.

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

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