Bit Twiddling

Martin Hilscher

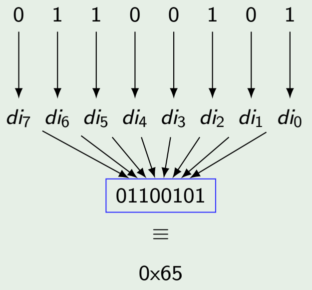
Gegeben

- 64 digitale In-Ports
- Jeder Port liefert ein Bit eines *long*-Werts

Gesucht

- Funktion *bitscan*(*long bitset*)
- Führe für jedes 1-Bit *process*(*i*) aus
- Mit *i* Index des gesetzte 1-Bits
- Reihenfolge ist unwichtig

Beispiel (8 Bit)



```
void bitscan(long bitset) {
 for (int i=0; i < 64; ++i){
    if ((bitset >>> i) &  1 == 1) {
      process(i);
    }
  }
}
```

Wieviel kann man da schon falsch machen?

```
void bitscan(long bitset) {
 for (int i=0; i < 64; ++i){
    if ((bitset >>> i) & 1 == 1) {
      process(i);
    }
  }
}
```

Fast alles !!!111!!

```
void bitscan(long bitset) {
 for (int i=0; i < 64; ++i){
    if ((bitset >>> i) & 1 == 1) {
      process(i);
    }
  }
}
```

Wird für jedes Bit aufgerufen

Fast alles !!!111!!

```
void bitscan(long bitset) {
 for (int i=0; i < 64; ++i){
    if ((bitset >>> i) & 1 == 1) {
      process(i);
    }
  }
}
```

Wird für jedes Bit aufgerufen

Im Schnitt nur für jedes 2te Bit

Fast alles !!!111!!

$64 \cdot 4$ cycle

```
void bitscan(long bitset) {
 for (int i=0; i < 64; ++i){
    if ((bitset >>> i) & 1 == 1) {
      process(i);
    }
  }
}
```

Fast alles !!!111!!

64 · 4 cycle

64 · 1 cycle

```
void bitscan(long bitset) {
 for (int i=0; i < 64; ++i){
    if ((bitset >>> i) & 1 == 1) {
      process(i);
    }
  }
}
```

Fast alles !!!111!!

$64 \cdot 4$ cycle

$64 \cdot 1$ cycle

```
void bitscan(long bitset) {
 for (int i=0; i < 64; ++i){
    if ((bitset >>> i) & 1 == 1) {
      process(i);
    }
  }
}
```

$64 \cdot 1$ cycle

Fast alles !!!111!!

$64 \cdot 4$ cycle

$64 \cdot 1$ cycle

```
void bitscan(long bitset) {
 for (int i=0; i < 64; ++i){
    if ((bitset >>> i) & 1 == 1) {
      process(i);
    }
  }
}
```

$64 \cdot 1$ cycle

$64 \cdot 1$ cycle

Fast alles !!!111!!

$64 \cdot 4$ cycle $64 \cdot 4$ cycle $64 \cdot 1$ cycle

```
void bitscan(long bitset) {
 for (int i=0; i < 64; ++i){
    if ((bitset >>> i) & 1 == 1) {
      process(i);
    }
  }
}
```

$64 \cdot 1$ cycle $64 \cdot 1$ cycle

	Plain
Ohne Loop-Unrolling:	704 cycle

Fast alles !!!111!!

64 · 4 cycle

```
void bitscan(long bitset) {
 for (int i=0; i < 64; ++i){
    if ((bitset >>> i) & 1 == 1) {
      process(i);
    }
  }
}
```

64 · 1 cycle

64 · 1 cycle

	Plain
Ohne Loop-Unrolling:	704 cycle
Mit Loop-Unrolling:	384 cycle

Fast alles !!!111!!

```
void bitscan(long bitset) {
 for (int i=0; i < 64; ++i){
    if ((bitset >>> i) & 1 == 1) {
      process(i);
    }
  }
}
```

	Plain	Branch Prediction
Ohne Loop-Unrolling:	704 cycle	+ ?
Mit Loop-Unrolling:	384 cycle	+ ?

Fast alles !!!111!!

```
void bitscan(long bitset) {
 for (int i=0; i < 64; ++i){
    if ((bitset >>> i) & 1 == 1) {
      process(i);
    }
  }
}
```

Für diesen Code
extrem kontraproduktiv

	Plain	Branch Prediction
Ohne Loop-Unrolling:	704 cycle	+ ?
Mit Loop-Unrolling:	384 cycle	+ ?

Zwei Ideen

1. Wenig (/ Keine) bedingte Sprünge
   - ⇒ Vermeidet teure Operationen
   - ⇒ Branch-Prediction ausgeschaltet
2. Weniger Rechnen
   - Berechne schnell Position des nächsten 1-Bits
   - Überspringe die 0-Bits
     - ⇒ I can haz speedup !!!111!!

Finde das nächste 1-bit

Erster Schritt

0010 0001 0100 0100	x		
1101 1110 1011 1011	˜x		
1101 1110 1011 1100	˜x + 1	$\equiv$	-x
0000 0000 0000 0100	x & (˜x + 1)	$\equiv$	x & -x

Finde das nächste 1-bit

Erster Schritt

0010 0001 0100 0100		x		
1101 1110 1011 1011		˜x		
1101 1110 1011 1100		˜x + 1	≡	-x
0000 0000 0000 0100	x &	(˜x + 1)	≡	x & -x

Problem

Jetzt haben wir 2^i, aber wir brauchen i.

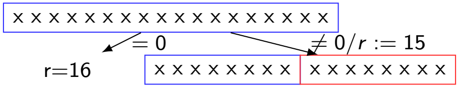
$2^i \longrightarrow i$ zähle rechtsstehende 0 bits

- Wir zählen einfach die 0-Bits rechts vom 1-Bit
- Können nicht alle Bits einzeln Zählen
  - $\Rightarrow$ Für jedes gesetzte Bit der Algorithmus von Folie 2
- Benutze binäre Suche

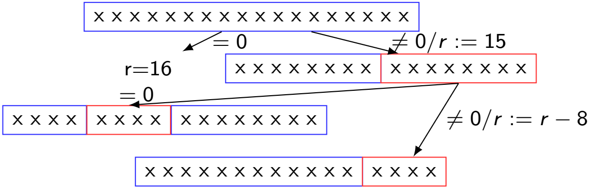
Hacker's Delight 00

$\boxed{x\ x\ x\ x\ x\ x\ x\ x\ x\ x\ x\ x\ x\ x\ x\ x\ x\ x}$

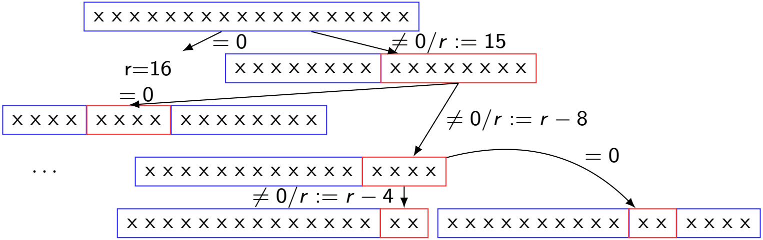
Hacker's Delight 00



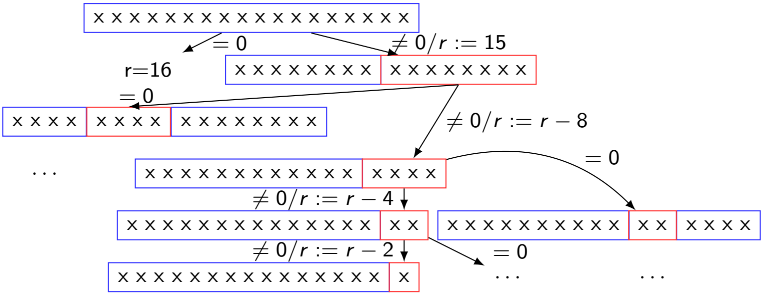
Hacker's Delight 00



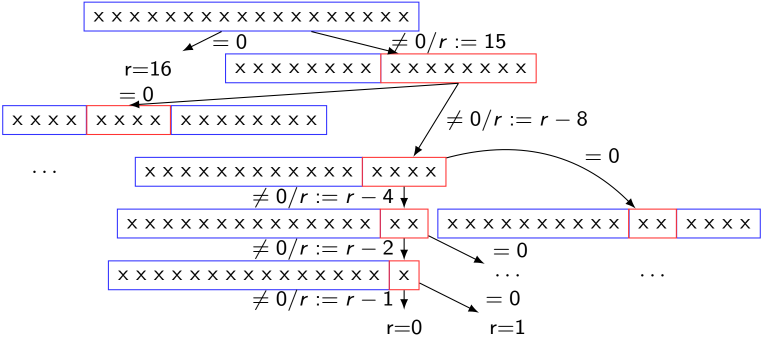
Hacker's Delight 00



Hacker's Delight 00



Hacker's Delight 00



Hacker's Delight 00

```
int numberOfTrailingZeros(long i) {
  int x, y;
  if (i == 0) return 64;
  int n = 63;
  y = (int)i;
    if (y != 0) { n = n - 32; x = y; }
    else x = (int)(i >>> 32);
  y = x << 16; if (y != 0) { n = n - 16; x = y; }
  y = x <<  8; if (y != 0) { n = n -  8; x = y; }
  y = x <<  4; if (y != 0) { n = n -  4; x = y; }
  y = x <<  2; if (y != 0) { n = n -  2; x = y; }
  return n - ((x << 1) >>> 31);
}

void bitscan(long bitset) {
  while (bitset) {
    int t = bitset & -bitset;
    process(Long.numberOfTrailingZeros(t));
    bitset ^= t;
  }
}
```

Probleme

- Immer noch zu viele bedingte Sprünge :-(
- 33 Operationen / 1-Bit :-C

Probleme

- Immer noch zu viele bedingte Sprünge :-(
- 33 Operationen / 1-Bit :-C

Aber ...

- Wir betrachtent nur einen Spezialfall
- Es gibt noch andere Lösungen

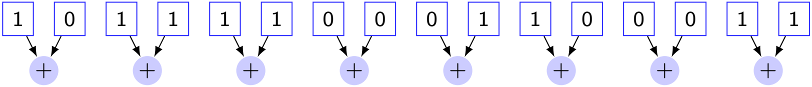
$2^i \longrightarrow i$ nutze $2^i - 1 = 2^{i-1} + 2^{i-2} + \ldots + 2^0$

- Wir betrachten 2^i und suchen i
- Es gilt: $2^i - 1 = 2^{i-1} + 2^{i-2} + \ldots + 2^0$
- Oder binär z.B.: $2^8 - 1 = 11111111_b$
- Also: $\#_1(2^i - 1) = i$
- Wir müssen also nur die 1-Bits zählen

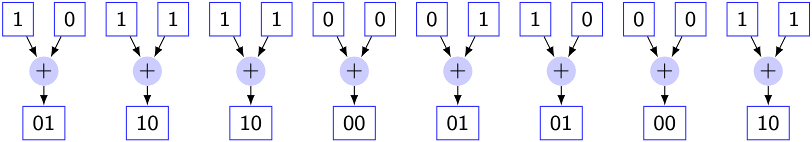
Hacker's Delight 01

1	0	1	1	1	1	0	0	0	1	1	0	0	0	1	1

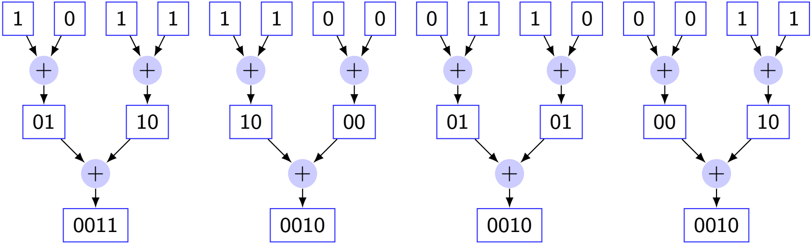
Hacker's Delight 01



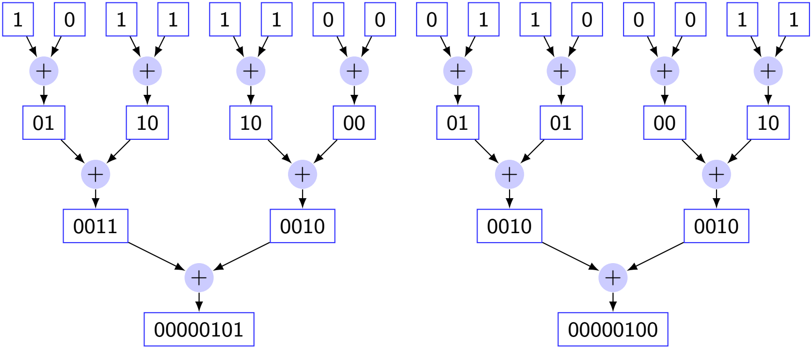
Hacker's Delight 01



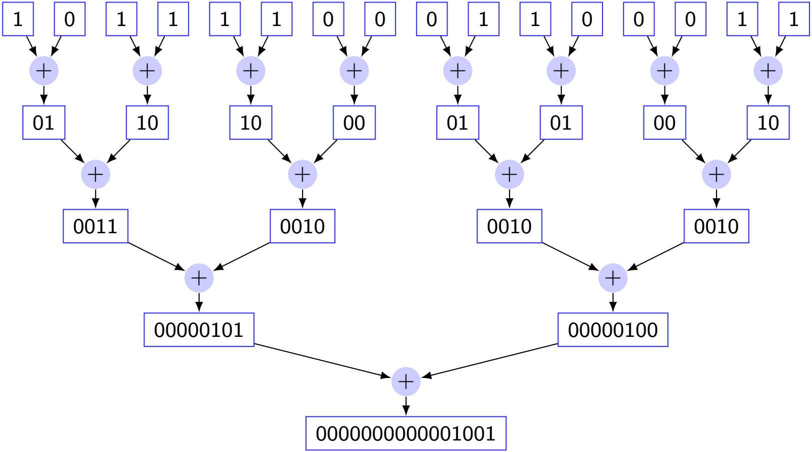
Hacker's Delight 01



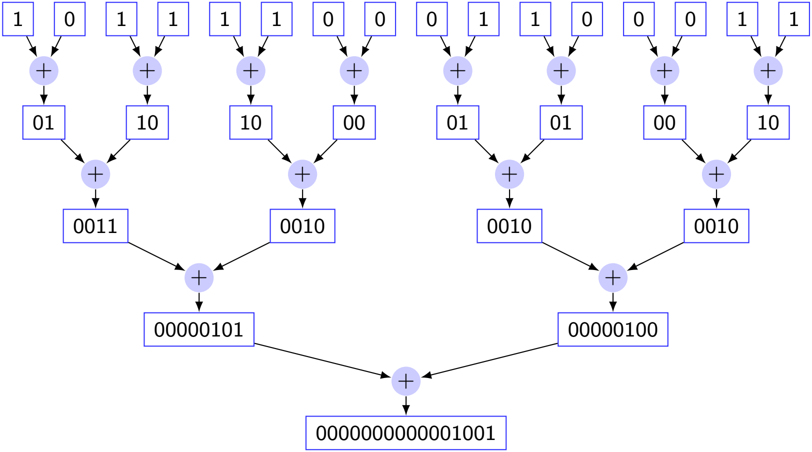
Hacker's Delight 01



Hacker's Delight 01



Wieviele Operationen sind das?



Schnelle parallele Addition von Bits

10 11 11 00 01 10 00 11	x
01 01 01 01 01 01 01 01	0x5555
00 01 01 00 01 00 00 01	x & 0x5555

10 11 11 00 01 10 00 11	x
01 01 11 10 00 11 00 01	$(x \gg 1)$
01 01 01 01 01 01 01 01	0x5555
01 01 01 00 00 01 00 01	$(x \gg 1)$ & 0x5555

00 01 01 00 01 00 00 01	x & 0x5555
01 01 01 00 00 01 00 01	$(x \gg 1)$ & 0x5555
01 10 10 00 01 01 00 10	(x & 0x5555) + $(x \gg 1)$ & 0x5555

Schnelle parallele Addition von Bits

10 11 11 00 01 10 00 11	x
01 01 01 01 01 01 01 01	0x5555
00 01 01 00 01 00 00 01	x & 0x5555

10 11 11 00 01 10 00 11	x
01 01 11 10 00 11 00 01	$(x \gg 1)$
01 01 01 01 01 01 01 01	0x5555
01 01 01 00 00 01 00 01	$(x \gg 1)$ & 0x5555

10 11 11 00 01 10 00 11	x
01 01 01 00 00 01 00 01	$(x \gg 1)$ & 0x5555
01 10 10 00 01 01 00 10	$(x - (x \gg 1)$ & 0x5555

Hacker's Delight 01

```
int bitCount(long i) {
  i = i - ((i >> 1) & 0x5555555555555555L);
  i =      (i         & 0x3333333333333333L)
      + ((i >> 2)     & 0x3333333333333333L);
  i = (i + (i >> 4)) & 0x0f0f0f0f0f0f0f0fL;
  i = i + (i >> 8);
  i = i + (i >> 16);
  i = i + (i >> 32);
  return (int)i & 0x7f;
}

void bitscan(long bitset) {
  while (bitset) {
    int t = bitset & -bitset;
    process(Long.bitCount(t-1));
    bitset ^= t;
  }
}
```

Problem

- Keine bedingten Sprünge :-)
- Immer noch 17 Operationen / 1-Bit :-(
- Kann nur noch wenig optimiert werden

Problem

- Keine bedingten Sprünge :-)
- Immer noch 17 Operationen / 1-Bit :-(
- Kann nur noch wenig optimiert werden

Aber . . .

- Es gibt eine völlig andere Lösung

Achtung Mathe!!!!111!!!!

Einige Definitionen

Kombinatorik

Kombinatorik ist das Teilgebiet der Mathematik, dass sich mit **endlichen** und abzählbar unendlichen Strukturen beschäftigt.

Alphabet

Eine Alphabet ist eine endliche Menge von Buchstaben z.B. $\{0, 1\}$ oder $\{a, b, c\}$.

Wort / Sequenz

Ein Wort oder Sequenz über einem Alphabet A ist eine Abfolge von Buchstaben aus A z.B. 10110101.

De Bruijen Sequenz

De Bruijen Sequenz

Eine De Bruijen Sequenz $B(n, k)$ ist eine zyklische Sequenz der Länge n^k, über einem Alphabet mit n Buchstaben, in der jede Sequenz der Länge k genau einmal vorkommt.

De Bruijen Sequenz

De Bruijen Sequenz

Eine De Bruijen Sequenz $B(n, k)$ ist eine zyklische Sequenz der Länge n^k, über einem Alphabet mit n Buchstaben, in der jede Sequenz der Länge k genau einmal vorkommt.

Beispiel

0011 ist ein De Bruijen Sequence $B(2, 2)$. Denn: $2^2 = 4$ und 0011, 0011, 0011, 0011 sind alle Sequenzen der Länge 2 über $\{0, 1\}$

De Bruijen Sequenz

De Bruijen Sequenz

Eine De Bruijen Sequenz $B(n, k)$ ist eine zyklische Sequenz der Länge n^k, über einem Alphabet mit n Buchstaben, in der jede Sequenz der Länge k genau einmal vorkommt.

Beispiel

0011 ist ein De Bruijen Sequence $B(2, 2)$. Denn: $2^2 = 4$ und 0011, 0011, 0011, 0011 sind alle Sequenzen der Länge 2 über $\{0, 1\}$

Beispiel

00010111 ist ein De Bruijen Sequence $B(2, 3)$. Denn: $2^3 = 8$ und 00010111, 00010111, 00010111, 00010111, 00010111, 00010111, 00010111, 00010111 sind alle Sequenzen der Länge 3 über $\{0, 1\}$

Na und?

Na und?

- Was wäre, wenn wir $B(2,6)$ finden könnten?

Na und?

- Was wäre, wenn wir $B(2,6)$ finden könnten?
- $B(2,6)$ wäre 64 Zeichen $\equiv$ 64 Bit lang
- $B(2,6)$ enthält jede Teilsequenz der Länge 6 genau einmal
- Also jede Zahl zwischen 0 und 63 genau einmal

Dämmerts?

- Unsere Zahl x ist von der Form 2^i
- Wenn wir von $B(2,6) \cdot x$ die obersten 6 Bits betrachten ...

Dämmerts?

- Unsere Zahl x ist von der Form 2^i
- Wenn wir von $B(2,6) \cdot x$ die obersten 6 Bits betrachten ...
- ... haben wir eine eindeutige Zahl zwischen 0 und 63

Dämmerts?

- Unsere Zahl x ist von der Form 2^i
- Wenn wir von $B(2,6) \cdot x$ die obersten 6 Bits betrachten ...
- ... haben wir eine eindeutige Zahl zwischen 0 und 63
- Leider nicht in der richtigen Reihenfolge

Dämmerts?

- Unsere Zahl x ist von der Form 2^i
- Wenn wir von $B(2,6) \cdot x$ die obersten 6 Bits betrachten …
- … haben wir eine eindeutige Zahl zwischen 0 und 63
- Leider nicht in der richtigen Reihenfolge
- Aber dafür kann man in eine Tabelle gucken

Beispiel für 8 Bit

$B(2,3) \cdot 2^i \equiv B(2,3) \ll i$

$B(2,3)$		x & -x			Markierte Bits
00010111	$\cdot$	00000001	$=$	00010111	0
00010111	$\cdot$	00000010	$=$	00101110	1
00010111	$\cdot$	00000100	$=$	01011100	2
00010111	$\cdot$	00001000	$=$	10111000	5
00010111	$\cdot$	00010000	$=$	01110000	3
00010111	$\cdot$	00100000	$=$	11100000	7
00010111	$\cdot$	01000000	$=$	11000000	6
00010111	$\cdot$	10000000	$=$	10000000	4

Lookup-Table

table:

0	1	2	3	4	5	6	7
0	1	2	4	7	3	6	5

Überraschung

Es gibt Beweise die zeigen: für jede Wahl von n und k gibt es DeBruijen-Sequenzen der Form $B(n, k)$.

Binär

0000001000011000101000111001001011001101001111010101110110111111

Hexadecimal

0x0218a392cd3d5dbfL

Hexadecimal

0x0218a392cd3d5dbfL

ist eine DeBruijen Sequenz der Form $B(2,6)$

```
int table[64];
void bitscan(long bitset) {
  while (bitset) {
    int t = bitset & -bitset;
    process(table[(t * 0x0218a392cd3d5dbfL) >>> 58]);
    bitset ^= t;
  }
}
```

Muss nur einmal ausgerechnet werden

```
int table[64];
void bitscan(long bitset) {
  while (bitset) {
    int t = bitset & -bitset;
    process(table[(t * 0x0218a392cd3d5dbfL) >>> 58]);
    bitset ^= t;
  }
}
```

Problem

- Nur noch 7 Operationen / 1-Bit
- Aber eine davon potentiell seeeeehr teurer table lookup

Problem

- Nur noch 7 Operationen / 1-Bit
- Aber eine davon potentiell seeeeehr teurer table lookup

But ...

- ... we are still not done yet !!!!111oneeleven!!!!

Surprise: Hardwarhacking

Surprise: Hardwarhacking

- Die Reihenfolge der Abarbeitungen ist egal

Surprise: Hardwarhacking

- Die Reihenfolge der Abarbeitungen ist egal
- Ändere Verkabelung

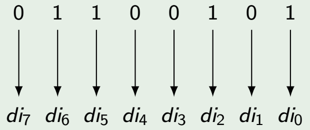
Surprise: Hardwarhacking

- Die Reihenfolge der Abarbeitungen ist egal
- Ändere Verkabelung
- Damit Reihenfolge der Bits doch wie aus DeBruijen-Sequenz

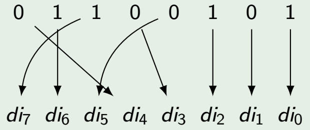
Surprise: Hardwarhacking

- Die Reihenfolge der Abarbeitungen ist egal
- Ändere Verkabelung
- Damit Reihenfolge der Bits doch wie aus DeBruijen-Sequenz
- Damit kein Table-Lookup mehr nötig

Beispiel (8 Bit)



Beispiel (8 Bit)



De Bruijen 01

```
void bitscan(long bitset) {
  while (bitset) {
    int t = bitset & -bitset;
    process((t * 0x0218a392cd3d5dbfL) >>> 58);
    bitset ^= t;
  }
}
```

Strike

Strike

- 6 Operationen / 1-Bit

Strike

- 6 Operationen / 1-Bit
- Kein bedingten Sprünge mehr $\rightarrow$ Keine Branch-Prediction

Strike

- 6 Operationen / 1-Bit
- Kein bedingten Sprünge mehr $\rightarrow$ Keine Branch-Prediction
- Falls alle Bits gesetzt sind $6 \cdot 64 = 384$ Operationen

Strike

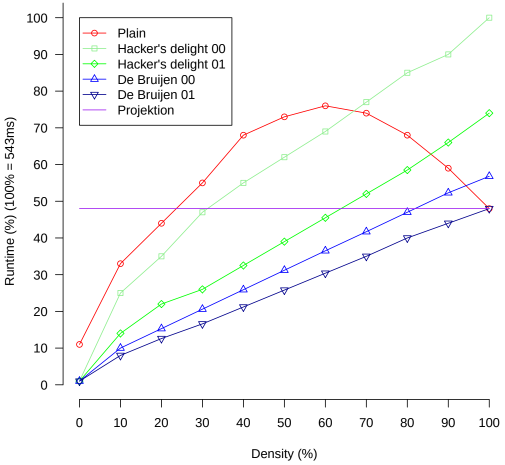
- 6 Operationen / 1-Bit
- Kein bedingten Sprünge mehr $\rightarrow$ Keine Branch-Prediction
- Falls alle Bits gesetzt sind $6 \cdot 64 = 384$ Operationen
- Damit:

  Worst Case Performance dieses Codes

  $\equiv$ Best case Performance des Codes mit dem wir angefangen haben

Glaub ich nicht!!!111einself!!

Gimme the numbers!!!111einself!!



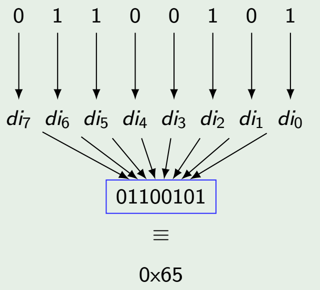
Gegeben

- 64 digitale In-Ports
- Jeder Port liefert ein Bit eines *long*-Werts

Gesucht

- Funktion *updateCounters*(*long bitset*)
- Zähle wie häufig das *i*-te Bit auf 1 gesetzt ist
- Reihenfolge ist unwichtig
- Counter werden häufig geschrieben, selten(er) gelesen

Beispiel (8 Bit)



Plain

```
void updateCounters(long bitset) {
  for (int i=0; i < 64; i++)
    if ((bitset >> i)&1)
      table[i]++;
}
```

De Bruijen

```
void updateCounters(long bitset) {
  for (int t; (t = bitset & -bitset); bitset ^= t)
    count[(i * DEBRUIJN_CONSTANT) >> 58]++;
}
```

Branchless

```
void updateCounters(long bitset) {
  for (int i=0;i<64;i++)
    count[i] += ((bitset[i] >> i) & 1);
}
```

Darstellung von Zahlen

Übliche Darstellung

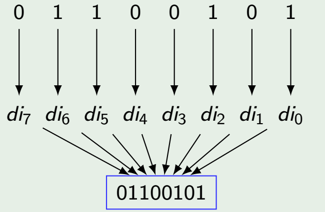
	msb ↔ lsb		
x[0]	00000010	=	2
x[1]	00000001	=	1
x[2]	00010101	=	21
x[3]	00000011	=	3
x[4]	00000110	=	6
x[5]	00000101	=	5
x[6]	00001001	=	9
x[7]	00001011	=	11

Und wo is das Problem

Übliche Darstellung

	msb ↔ lsb		
x[0]	00000010	=	2
x[1]	00000001	=	1
x[2]	00010101	=	21
x[3]	00000011	=	3
x[4]	00000110	=	6
x[5]	00000101	=	5
x[6]	00001001	=	9
x[7]	00001011	=	11

Beispiel (8 Bit)



Wieviele Operationen sind das pro gesetztem Bit?

Wir drehen unseren Kopf um 90°

übliche Darstellung

x[7]	x[6]	x[5]	x[4]	x[3]	x[2]	x[1]	x[0]	
0	0	0	0	0	0	0	0	msb
0	0	0	0	0	0	0	0	
0	0	0	0	0	0	0	0	
0	0	0	0	0	1	0	0	↑
1	1	0	0	0	0	0	0	↓
0	0	1	1	0	1	0	0	
1	0	0	1	1	0	0	1	
1	1	1	0	1	1	1	0	lsb
=	=	=	=	=	=	=	=	
11	9	5	6	3	21	1	2	

Wir drehen unseren Kopf um 90°

UN-übliche Darstellung

x[7]	0	0	0	0	0	0	0	0	msb
x[6]	0	0	0	0	0	0	0	0	
x[5]	0	0	0	0	0	0	0	0	
x[4]	0	0	0	0	0	1	0	0	↑
x[3]	1	1	0	0	0	0	0	0	↓
x[2]	0	0	1	1	0	1	0	0	
x[1]	1	0	0	1	1	0	0	1	
x[0]	1	1	1	0	1	1	1	0	lsb
	=	=	=	=	=	=	=	=	
	11	9	5	6	3	21	1	2	

Getauscht zum leichteren Lesen

UNübliche Darstellung

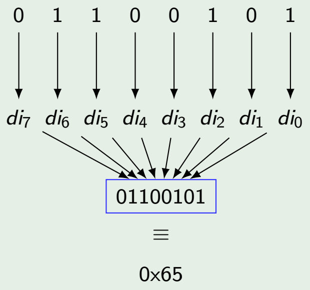
x[0]	1	1	1	0	1	1	1	0	lsb
x[1]	1	0	0	1	1	0	0	1	
x[2]	0	0	1	1	0	1	0	0	
x[3]	1	1	0	0	0	0	0	0	$\uparrow$
x[4]	0	0	0	0	0	1	0	0	$\downarrow$
x[5]	0	0	0	0	0	0	0	0	
x[6]	0	0	0	0	0	0	0	0	
x[7]	0	0	0	0	0	0	0	0	msb
	=	=	=	=	=	=	=	=	
	11	9	5	6	3	21	1	2	

Und nun?

UNübliche Darstellung

x[0]	11101110	lsb
x[1]	10011001	
x[2]	00110100	
x[3]	11000000	↑
x[4]	00000100	↓
x[5]	00000000	
x[6]	00000000	
x[7]	00000000	msb

Beispiel (8 Bit)

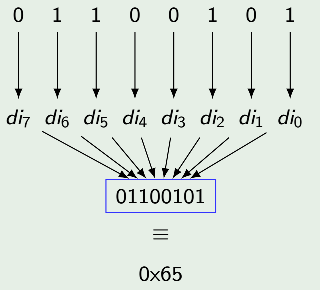


Und nun?

UNübliche Darstellung

x[0]	11101110	lsb
x[1]	10011001	
x[2]	00110100	
x[3]	11000000	$\uparrow$
x[4]	00000100	$\downarrow$
x[5]	00000000	
x[6]	00000000	
x[7]	00000000	msb

Beispiel (8 Bit)



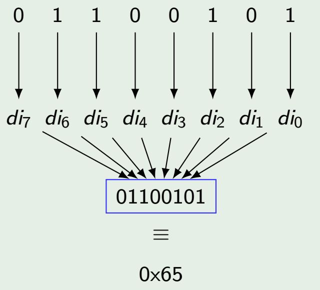
1. Flag-Vektor kann einfach addiert werden

Und nun?

UNübliche Darstellung

x[0]	11101110	lsb
x[1]	10011001	
x[2]	00110100	
x[3]	11000000	↑
x[4]	00000100	↓
x[5]	00000000	
x[6]	00000000	
x[7]	00000000	msb

Beispiel (8 Bit)



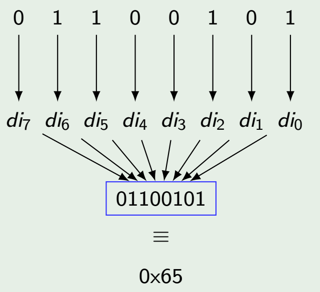
1. Flag-Vektor kann einfach addiert werden
2. Wir verlieren die CPU-Logik dafür

Und nun?

UNübliche Darstellung

x[0]	11101110	lsb
x[1]	10011001	
x[2]	00110100	
x[3]	11000000	↑
x[4]	00000100	↓
x[5]	00000000	
x[6]	00000000	
x[7]	00000000	msb

Beispiel (8 Bit)



1. Flag-Vektor kann einfach addiert werden
2. Wir verlieren die CPU-Logik dafür
3. Bauen wir uns halt unseren eigenen Addierer

Willkommen zu Addiererbauen leichtgemacht

Additionstabelle

A	B	Carry	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Willkommen zu Addiererbauen leichtgemacht

Additionstabelle

A	B	Carry	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Was sind die Formeln um Carry und Sum zu berechnen?

Willkommen zu Addiererbauen leichtgemacht

Additionstabelle

A	B	Carry	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Formeln für Carry und Sum

$$\begin{aligned} carry &= a \,\hat{}\, b \\ sum &= a \,\&\, b \end{aligned}$$

Was sind die Formeln um Carry und Sum zu berechnen?

Vertical

```
void updateCounters(long bitset) {
  long carry = bitset;
  long *p = arrayOfCounter;
  while (carry) {
    a = *p;
    b = carry;
    *p++ = a ^ b;
    carry = a & b;
  }
}
```

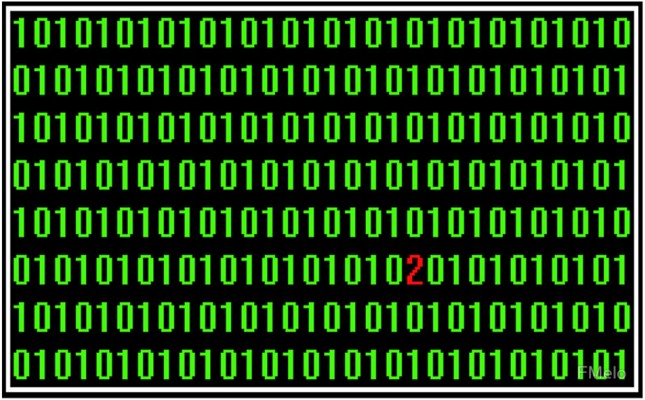
Ergebniss

- Wir sind nicht mehr abhängig von der Anzahl der gesetzten Bits
- Die Anzahl der Durchläufe hängt von der längsten Kette an Carries ab
- Läuft im Schnitt 6-7 mal (bei 64 Bit-Werten)

Noch ein Speziallfall

Wir benötigen weniger Zähler als Stellen im Bitfeld zur Verfügung stehen.

Bender's nightmare



Redundant Positional System

Redundant Positional System

- Erlaubt mehrere Darstellungen für eine Zahl (ZehnUndZwanzig $\equiv$ Dreißig)
- Wir erlauben bei manchen Stellen die Ziffer 2 (Rote Ziffern)

Implementierung

- Am Anfang alle Ziffern normal (grün)
- Beim incrementieren einer roten Stelle behalten wir die Summe $\pmod 2$ als grüne, Rest als Carry
- Beim incrementieren einer grünen Stelle ist das Ergebnis höchstens 2, lassen wir als rote Stelle stehen
- Bei jedem Durchlauf werden einige rote Stellen grün und höchstens eine grüne rot (l.a.a.e.t.t.r)

Not shown here

Warning

Wir benötigen ein wenig Vorarbeit zur Aufbereitung der Daten, da wir zwischen den grünen Stellen immer eine Stelle Platz bei den Inputdaten lassen müssen. Diese Vorverarbeitung kann durch einfaches umstecken der drähte am Arduino geschehen. Ich nehme ab jetzt an, dass dies bereits passiert ist und das wir den dann resultierenden Wert auf Position 0 des Arrays `data` abelegt haben gelegt haben.

Additionstabelle

A	B	Carry	Sum
0	00	0	00
0	01	0	01
0	10	0	01
0	11	1	00
1	00	0	01
1	01	1	00
1	10	1	00
1	11	1	01

Additionstabelle

A	B	Carry	Sum
0	00	0	00
0	01	0	01
0	10	0	01
0	11	1	00
1	00	0	01
1	01	1	00
1	10	1	00
1	11	1	01

Was sind die Formeln um Carry und Sum zu berechnen?

Additionstabelle

A	B	Carry	Sum
0	00	0	00
0	01	0	01
0	10	0	01
0	11	1	00
1	00	0	01
1	01	1	00
1	10	1	00
1	11	1	01

Was sind die Formeln um Carry und Sum zu berechnen?
(ich empfehle eine Hilfsvariable zu benutzen)

Additionstabelle

A	B	Carry	Sum
0	00	0	00
0	01	0	01
0	10	0	01
0	11	1	00
1	00	0	01
1	01	1	00
1	10	1	00
1	11	1	01

Berechnung von Carry und Sum

$$\begin{aligned} t &= b[0] \mathbin{\hat{}} b[1] \\ carry &= (a \,\&\, b) \mathbin{\hat{}} (t \,\&\, carry) \\ sum[1] &= 0 \\ sum[0] &= a \mathbin{\hat{}} t \end{aligned}$$

Was sind die Formeln um Carry und Sum zu berechnen?
(ich empfehle eine Hilfsvariable zu benutzen)

Bender

```
long *p = data ;
long a, b, t;
while ((a = *p)) {
  b = p[1];
  t = a ^ b;
  *p++ = 0;
  *p++ = t ^ carry;
  carry = (a & b) ^ (t & carry);
}
*p = carry;
```

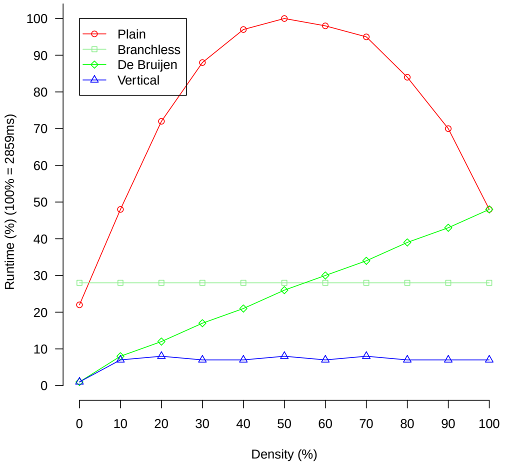
Eine Optimierung gibts noch

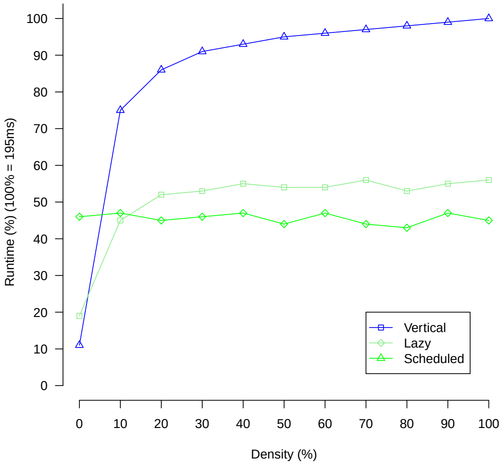
Es geht noch ein ganz wenig schneller, wenn man den Zustand in einem getrennten Register hält. Wie der Code funktioniert?

„left as an excercice to the reader“

Scheduled

```
int p = 0;
int t = ++time;
while ((t & 1) == 0) {
  long a = data[p];
  long b = data[p + 1];
  long s = a ^ b;
  data[p + 1] = s ^ c;
  c = (a & b) ^ (c & s);
  t >>= 1;
  p += 2;
}
data[p] = c;
```


Plain
Branchless
De Bruijen
Vertical
Runtime (%) (100% = 2859ms)
Density (%)
0
10
20
30
40
50
60
70
80
90
100



Runtime (%) (100% = 195ms)
100
90
80
70
60
50
40
30
20
10
0
0
10
20
30
40
50
60
70
80
90
100
Density (%)
Vertical
Lazy
Scheduled

Last remark

Mit einem „Carry-save adder“kann die Laufzeit von Verticalen Zählen auf amortisiert konstante Laufzeit gesenkt werden.