Computerarithmetik

Kutil, 2007

Unter Computerarithmetik versteht man üblicherweise elektronische Schaltkreise, die Rechenoperationen auf binären Zahlen durchführen. Die Strategien, die beim Entwurf solcher Schaltungen angewendet werden, haben aber dennoch starken Algorithmus-Charakter. Diese Dualität kommt in der hier verwendeten Sprache *Alluvion* zur Geltung. In der Art von Algorithmen können in dieser Sprache die Algorithmen der Computerarithmetik formal beschrieben werden. Diese "Programme" können dann in üblicher Weise ausgeführt und getestet werden. Das Wichtige ist aber, dass jedes Programm eindeutig einer Schaltung entspricht, deren Eigenschaften wie Gatecount und Gatedelays abgeleitet und errechnet werden können. Auf diese Weise ersparen wir uns das mühsame Zeichnen von Schaltdiagrammen und die so formulierten Algorithmen sind auch leichter verständlich.

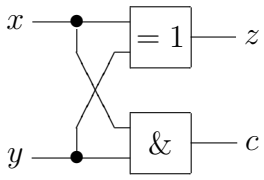
1 Alluvion

Alluvion soll computerarithmetische Algorithmen kurz und prägnant darstellen können. Es lehnt sich syntaktisch an bekannte Programmiersprachen an (Java, C). Außerdem soll aus einem Alluvion-Programm der entsprechende Schaltkreis ableitbar sein.

1.1 Funktionen

Funktionen sind der Hauptbestandteil von Alluvion. Sie werden in ähnlicher Weise definiert, wie sie benutzt werden.

```
1 (c, z) = HalfAdd (x, y)
2 {
3   z = xor (x, y);
4   c = and (x, y);
5 }
```



In diesem Beispiel wird die Funktion `HalfAdd` definiert. Sie hat zwei In-Argumente, nämlich `x` und `y`, und zwei Out-Argumente, nämlich `c` und `z`. Diese Argumente, sowie alle anderen Datenvariablen, sind *Bits*, dh sie können die Werte 1 oder 0 annehmen.

Das Schema, nach dem Funktionen deklariert werden, ist immer "Out-Args = Funktionsname (In-Args)". Nach dem Funktionsnamen können dann noch Kontrollargumente dazu kommen. Dazu später.

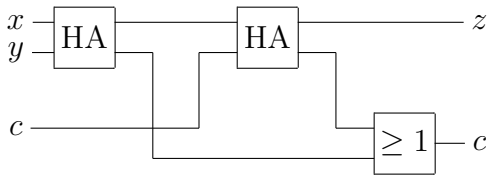
Nach der Deklaration kommt der Funktionskörper, eingeleitet mit `{` und beendet mit `}`. Dazwischen befinden sich Anweisungen. In diesem Beispiel sind das zwei Zuweisungen inklusive Funktionsaufruf. Es werden zwei interne Funktionen aufgerufen, nämlich `xor` und `and`. Es gibt folgende interne Funktionen:

Funktion	berechnet
`not (x)`	$\bar{x}$
`and (x, y)`	$x \wedge y$
`or (x, y)`	$x \vee y$
`xor (x, y)`	$x \dot{\vee} y = (x \wedge \bar{y}) \vee (\bar{x} \wedge y)$

Obige Funktion berechnet also $z = x \dot{\vee} y$ und $c = x \wedge y$. Das entspricht einem Halbaddierer, wie er in der Schaltung rechts dargestellt ist.

Neue Variablen müssen nicht deklariert werden. Es reicht, wenn sie in den Out-Argumenten einer Zuweisung vorkommen. Sie dürfen allerdings erst verwendet werden, wenn sie vorher beschrieben wurden. Siehe folgendes Beispiel.

```
1 (cout, z) = FullAdd (x, y, cin)
2 {
3   (c1, s1) = HalfAdd (x, y);
4   (c2, z) = HalfAdd (s1, cin);
5   cout = or (c1, c2);
6 }
```



Hier werden drei Variablen erzeugt, nämlich `c1`, `s1` und `c2`. Als Variablennamen können beliebige alphanumerische Zeichenketten verwendet werden. Sie müssen aber mit einem Buchstaben beginnen. Für Argumente und Funktionsnamen gilt übrigens das selbe.

Außerdem wird hier die Funktion `HalfAdd` aufgerufen, die wir oben definiert haben. Es ergibt sich dann ein Volladdierer, der in der Schaltung rechts dargestellt ist, wobei mit HA die Halbaddierer-Schaltung von vorher als Block benutzt wird.

Man sieht, dass Verbindungen in der Schaltung immer dann existieren, wenn eine aufgerufene Funktion mit einer Variable versorgt wird, die vorher von einer anderen beschrieben wurde. Um genau zu sein, entspricht jeder Funktionsaufruf, dh jede Funktionsinstanz einem Block in der Schaltung. Jedes Mal, wenn eine Variable beschrieben wird, wird ein neuer Verbindungsknoten erzeugt und mit dem Ausgang des Funktionsblocks verbunden. Wenn die Variable als In-Argument einer anderen Funktionsinstanz verwendet wird, wird der entsprechende Eingang des entsprechenden Funktionsblocks mit dem Verbindungsknoten verbunden. Im Beispiel wird beim ersten Aufruf von `HalfAdd` u.a. ein Verbindungsknoten für `s1` erzeugt und mit dem ersten Halbaddierer verbunden. Beim zweiten Aufruf von `HalfAdd` mit `s1` als In-Argument wird der Knoten mit dem zweiten Halbaddierer verbunden.

1.2 Aufruf

Wenn man die gewünschten Funktionen in eine Datei, zB `add.alv`, geschrieben hat, ruft man die Funktion `FullAdd` auf mit

```
1 alluvion add.alv "FullAdd ('1', '0', '1')"
```

und erhält folgende Ausgabe (Auszug):

```
1 cout    1       (3, 3)
2 z       0       (2, 2)
```

Das bedeutet, dass `FullAdd` $1 + 0 + 1 = 10_2$ berechnet und die zwei Ergebnisbits ausgibt (`cout`$= 1$, `z`$= 0$). Zu den Werten in Klammer später.

1.3 Arrays

Es ist möglich, mehrere Bits in eine Variable zu packen. Dabei muss ein Indexbereich angegeben werden, das sind ein bis drei ganzzahlige Werte in eckigen Klammern, durch Doppelpunkt getrennt. Der erste Werte ist der Beginn des Bereichs, der zweite das Ende. Ist der zweite Wert

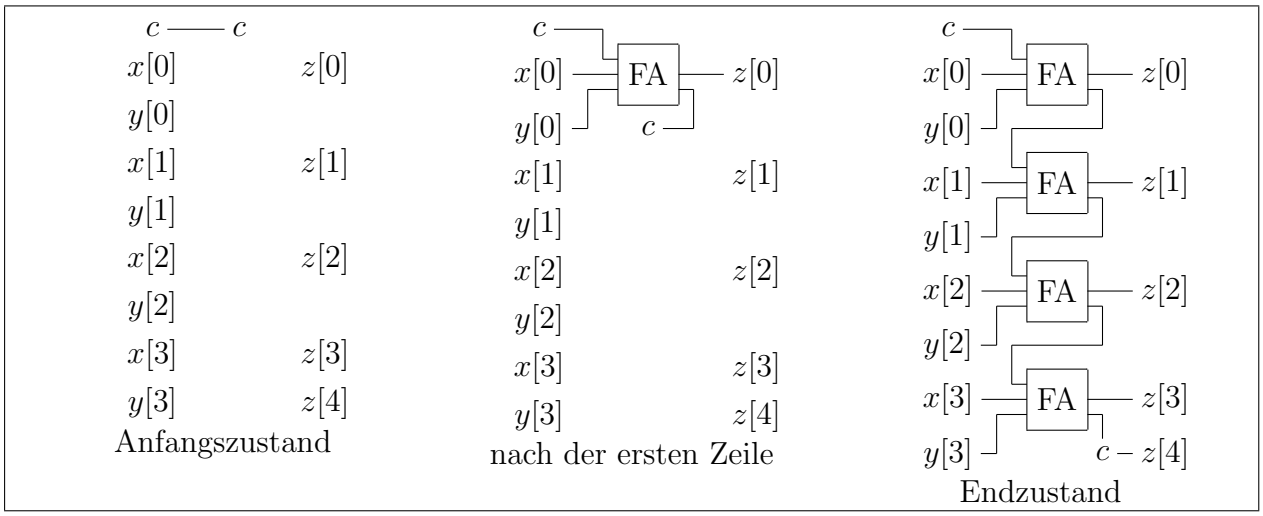
nicht angegeben, wird er gleich dem ersten gesetzt. Der Bereich umfasst dann nur ein Element. Im dritten Wert kann dann noch ein Schrittwert angegeben werden.

Folgendes Beispiel implementiert einen 4-Bit-Addierer.

```
1 z[0:4] = Add4 (x[0:3], y[0:3], c)
2 {
3   (c, z[0]) = FullAdd (x[0], y[0], c);
4   (c, z[1]) = FullAdd (x[1], y[1], c);
5   (c, z[2]) = FullAdd (x[2], y[2], c);
6   (c, z[3]) = FullAdd (x[3], y[3], c);
7   z[4] = c;
8 }
```

Man sieht, dass auch Teilbereiche eines bestehenden Arrays beschrieben werden können, wenn zB `z[2]` als Out-Argument auftaucht.

Ein weiteres wichtiges Detail ist die Verwendung von `c`, um das Carry-Bit von einem Volladdierer zum nächsten weiterzugeben. Es wird nach jedem Aufruf von `FullAdd` überschrieben und erzeugt damit jedes Mal einen neuen Verbindungsknoten im zugehörigen Schaltbild, wie im folgenden Bild gezeigt.



Die Funktion kann zB mittels `Add4('1100','0101','0')` aufgerufen werden. Man sieht, dass mehrere Bits in einem Argument übergeben werden. Das linksstehende Bit ist immer das mit dem größeren Index. Es wird also `x[3]='1'`, `x[2]='1'`, `x[1]='0'` und `x[0]='0'` gesetzt.

Es ist nicht notwendig, dass jedem deklarierten Argument genau ein Argument beim Aufruf entspricht. Es muss nur die gesamte Anzahl an Bits, die an die Funktion übergeben werden, mit der von der Funktion angenommenen übereinstimmen. Dabei ist wichtig, dass die Argumente immer von links nach rechts und vom größten Index bis zum kleinsten abgearbeitet werden. So würde die Anweisung `z[0:1]=HalfAdd('11')` bewirken, dass `z[0]='0'` und `z[1]='1'` gesetzt wird. Und die Anweisung `z[0:4]=Add4(x[0:8])` würde `x[5:8]` und `x[1:4]` addieren und `x[0]` als incoming Carry dazurechnen.

Es können auch negative Indizes angegeben werden (zB `x[-3]`), was das Programmieren von Fixpunktzahlen erleichtert. Wenn ein Element eines Arrays beschrieben wird, das noch nicht existiert, wird es einfach erzeugt. Wenn versucht wird, ein nichtexistentes Element mit einer internen Funktion zu lesen, tritt ein Fehler auf. Man kann jedoch nichtexistente Elemente an definierte Funktionen übergeben.

Es gibt auch mehrdimensionale Arrays. Die Indizes bzw. Indexbereiche werden dann durch Beistrich getrennt. So ist zB `a[0:5,-2:3]` ein zweidimensionales Array der Größe 6×6 und `a[1,2]` ein Element davon.

1.4 Kontrollvariablen

Eine zweite Art von Variablen beinhaltet nicht Bits sondern ganzzahlige Werte. Damit wird es möglich, Indizes nicht nur als unmittelbare Werte anzugeben. Hier die `Add4`-Funktion mit Kontrollvariablen:

```
z[0:4] = Add4 (x[0:3], y[0:3], c)
{
  i = 0;
  (c, z[i]) = FullAdd (x[i], y[i], c); i = i + 1;
  (c, z[i]) = FullAdd (x[i], y[i], c); i = i + 1;
  (c, z[i]) = FullAdd (x[i], y[i], c); i = i + 1;
  (c, z[i]) = FullAdd (x[i], y[i], c); i = i + 1;
  z[i] = c;
}
```

Dieses Beispiel macht zugegebenermaßen nicht viel Sinn, demonstriert aber die Funktionsweise von Kontrollvariablen. Mit Kontrollvariablen kann in üblicher Weise gerechnet werden. Es stehen die Operatoren `+-*/%^` zur Verfügung, wobei `%` die Modulo-Operation ist und `^` das Potenzieren. Daten- und Kontrollvariablen dürfen einander nicht zugewiesen werden. Die Anweisungsfolge `x='1';x=1;` führt zB zu einem Fehler, weil die erste Zuweisung `x` als Datenvariable mit dem Wert "wahr" definiert und die zweite einen ganzzahligen Wert zuweisen würde.

Kontrollvariablen können auch an Funktionen übergeben werden. Und zwar nicht in der normalen Argumentliste sondern in `<>`-Klammern direkt nach dem Funktionsnamen, ähnlich wie generische Parameter in C++. Damit könnte man eine Funktion `z[0:n] = Add<n> (x[0:n-1], z[0:n-1], c)` deklarieren, die binäre Zahlen mit beliebig vielen Bits addiert. Allerdings können wir das jetzt noch nicht programmieren. Dazu benötigt man Schleifen.

Es sind übrigens auch Arrays von Kontrollvariablen möglich.

1.5 Schleifen

Schleifen funktionieren wie üblich in Java oder C. Es gibt eine `for`- und eine `while`-Schleife. Hier die vollständig parametrisierte `Add`-Funktion:

```
z[0:n] = Add<n> (x[0:n-1], y[0:n-1], c)
{
  for i = [0:n-1]
  {
    (c, z[i]) = FullAdd (x[i], y[i], c);
  }
  z[n] = c;
}
```

Der Bereich, den die Schleifenvariable (`i`) durchläuft, wird gleich angegeben wie der Indexbereich von Arrays. Man beachte, dass das Carry-Bit in der Variable `c` von Iteration zu Iteration

weitergegeben wird. Bei Aufruf von `Add<4>('1100','0101','0')` wird die selbe Schaltung wie bei `Add4` erzeugt.

Um auch die `while`-Schleife zu demonstrieren, hier noch die gleiche Funktion etwas komplizierter implementiert:

```
z[0:n] = Add<n> (x[0:n-1], y[0:n-1], c)
{
  i = 0;
  while i < n
  {
    (c, z[i]) = FullAdd (x[i], y[i], c);
    i = i + 1;
  }
  z[n] = c;
}
```

Hier wird auch schon ein logischer Ausdruck (`i<n`) gebraucht. Mehr dazu im nächsten Abschnitt.

1.6 Bedingte Ausführung

Teile des Codes können mit der `if`-Anweisung bedingt ausgeführt werden. Blöcke, die nicht ausgeführt werden, führen auch zu keinen zusätzlichen Komponenten in der Schaltung. Folgendes Programm demonstriert sowohl die `if`-Anweisung als auch, dass in Alluvion auch rekursiv programmiert werden kann:

```
z[0:n] = Add<n> (x[0:n-1], y[0:n-1], c)
{
  if n = 1
  {
    z[0:1] = FullAdd (x[0], y[0], c);
  }
  else
  {
    m = n / 2;
    z[0:m] = Add<m> (x[0:m-1], y[0:m-1], c);
    z[m:n] = Add<n-m> (x[m:n-1], y[m:n-1], z[m]);
  }
}
```

Obwohl diese Funktion die Wörter hierarchisch in Hälften, Viertel, usw. unterteilt, wird die gleiche Schaltung wie oben erzeugt. Der Leser möge das nachvollziehen.

Kontrollvariablen können mit den Operationen `=`, `!=`, `>`, `<`, `>=`, `<=` verglichen werden, um Wahrheitswerte zu erzeugen. Wahrheitswerte können dann mit den Operationen `&` (und), `|` (oder) und `!` (nicht) verknüpft werden.

1.7 Delays

Jedes Gatter (`and`, `or`, `xor`, `not`) benötigt eine gewisse Zeit (ein Delay), um aus den Eingangssignalen ein stabiles Ausgangssignal zu erzeugen. Werden mehrere Gatter hintereinandergeschaltet, summiert sich dieses Delay. Da zB in Prozessoren der ganze arithmetische Algorithmus

während eines Taktzyklus ablaufen muss, bestimmt das Gesamtdelay die Taktfrequenz und daher die Leistungsfähigkeit des Prozessors.

Alluvion berechnet das Delay für jedes Ausgangsbit und gibt es (als ersten Wert in Klammer) aus. Bei Aufruf von `Add<8>('11110000','00101100','0')` ergibt sich:

```
z[0:n]:
  [8]   1       (17, 6)
  [7]   0       (16, 5)
  [6]   0       (14, 3)
  [5]   0       (12, 10)
  [4]   1       (10, 8)
  [3]   1       (8, 6)
  [2]   1       (6, 4)
  [1]   0       (4, 3)
  [0]   0       (2, 2)
```

Oft ist der Ausgang eines Gatters schon stabil, bevor der letzte Eingang stabil ist. So hat zB der Ausdruck `or('1',and('0','1'))` ein Delay von 2, weil ein `or`- und ein `and`-Gatter hintereinandergeschaltet werden. Der Ausgang des `or` ändert sich aber nach einem Delay von 1 nicht mehr, da der zweite Eingang (der Ausgang des `and`-Gatters) keinen Einfluss mehr auf den Ausgang des `or`-Gatters hat. Das "effektive Delay" ist daher 1. Dieser Wert ist natürlich datenabhängig. Das größte effektive Delay ist meist (aber nicht immer) gleich dem normalen Delay. Alluvion berechnet auch das effektive Delay und gibt es als zweiten Wert in der Klammer aus.

1.8 Weitere Statistiken

Des weiteren berechnet Alluvion die Anzahl der Gates sowie den maximalen Fanout:

```
Statistics:
  and: 16 instances, max. fanout = 1
  or: 8 instances, max. fanout = 2
  xor: 16 instances, max. fanout = 2
  HalfAdd: 16 instances, 32 gates (32 internal, 0 external)
  FullAdd: 8 instances, 40 gates (8 internal, 32 external)
  Add: 1 instances, 40 gates (0 internal, 40 external)
```

Für jede Funktion wird gezählt, wie oft sie in der Abarbeitung des Programms aufgrufen wird (instances). Das entspricht bei eingebauten Funktionen (and, or, xor, not) dem Gate-Count. Für andere Funktionen werden die direkten Aufrufe von eingebauten Funktionen gezählt (internal) und die Summe der Gatter gebildet, die indirekt durch Aufruf anderer Funktionen erzeugt werden (external).

Der Fanout ist die Anzahl der Gatter-Eingänge, die von einem Ausgang versorgt werden müssen. In gewissen Fertigungstechniken kann das problematisch werden. Ab einer bestimmten Fanout-Zahl muss dann ein sogenannter "Buffer" zwischengeschaltet werden, was zusätzliche Delays verursacht.

1.9 Debugging

Die beliebteste Form des Debugging ist das "printf-Debugging". Daher wird dieses von Alluvion unterstützt. Das heißt nichts anderes als, dass ein `print`-Befehl existiert. Er hat die Form

"`print var`", wobei `var` eine Daten- oder Kontrollvariable ist, bzw. "`print(v1,v2,...)`" zur Ausgabe von mehreren Variablen. Die Variablen werden ähnlich wie bei der Ergebnisausgabe angezeigt.

2 Addition

2.1 Ripple-Carry-Addition

Diese einfachste Art der Addition kennen wir schon aus Abschnitt 1. Sie heißt deshalb Ripple-Carry-Addition, weil das Carry-Bit der k-ten Stufe (k-tes Bit) alle höherwertigen Bits noch verändern kann, wodurch ein n-stufiger Durchreichprozess entsteht (ripple), wobei n die Wortbreite ist.

Die Komplexität des Algorithmus' ist linear sowohl bezüglich Delay als auch bezüglich der Gatterzahl:

n	Delay	Gatter
4	8	20
8	16	40
16	32	80
32	64	160
64	128	320

Das Delay wird hier und auch für alle weiteren Untersuchungen an `z[n-1]` gemessen, weil dieses Ausgabebit durchwegs das mit dem größten Delay ist.

2.2 Carry-Lookahead-Addition

Das Problem der Ripple-Carry-Addition ist, dass jede Stufe erst berechnet werden kann, wenn das hereinkommende Carry-Bit bekannt ist. Daher summieren sich die vollen Delays jeder Stufe. Wenn man es schafft, die Carry-Bits schneller zu berechnen, wird die gesamte Addition schneller ablaufen (dh mit kleinerem Gesamtdelay).

Betrachten wir einen Block von Stufen, zB `x[k:l-1]`. Am Ende dieses Blocks (dh nach Stufe $l-1$) kann ein Carry-Bit aus zwei Gründen entstehen. Einerseits kann der Block selbst ein Carry-Bit erzeugen, indem zB `x[l-1]=1` und `y[l-1]=1` ist. Andererseits kann ein hereinkommendes Carry-Bit weitergegeben werden. Das ist dann der Fall, wenn an allen Stellen `x[i]` und `y[i]` verschieden sind.

Diese beiden Fälle können auf zwei Bits abgebildet werden, das Generate-Bit G und das Propagate-Bit P. Bei einem Block der Länge 1 gilt: $G = x \wedge y, P = x \dot{\vee} y$.

```
1 (g, p) = GP (x, y)
2 {
3   g = and (x, y);
4   p = xor (x, y);
5 }
```

Wenn die G- und P-Bits von zwei angrenzenden Blöcken `[k:l-1]` und `[l:m-1]` bekannt sind (G_0, P_0, G_1, P_1), dann können diese Bits für den vereinigten Block `[k:m-1]` berechnet werden mittels $G = G_1 \vee (G_0 \wedge P_1), P = P_1 \wedge P_0$.

```
(g, p) = GP_Op (g0, p0, g1, p1)
{
  g = or (g1, and (g0, p1));
  p = and (p1, p0);
}
```

Mit diesen beiden Funktionen können das G- und P-Bit eines Blocks berechnet werden. Und zwar am besten durch hierarchisches Zweiteilen mittels Rekursion. Die untere und die obere Hälfte können getrennt unabhängig berechnet werden, weshalb hier kein Ripple-Effekt entsteht. Dann werden die Ergebnisse mit `GP_Op` kombiniert.

```
(g, p) = GP_Group<n> (x[0:n-1], y[0:n-1])
{
  if n > 1
  {
    m = n/2;
    (g, p) = GP_Op ( GP_Group<m> (x[0:m-1], y[0:m-1])
                   , GP_Group<n-m> (x[m:n-1], y[m:n-1])
                   );
  }
  else
  { (g, p) = GP (x[0], y[0]); }
}
```

Jetzt können wir ganz einfach einen Carry-Lookahead-Addierer konstruieren, indem wir die Wörter in gleich große Blöcke (der Größe k) zerlegen und auf diesen die normale Ripple-Carry-Addition durchführen. Die Input-Carries für jeden Block werden aber aus den G- und P-Bits berechnet. Dadurch "überspringt" das Carry-Bit den Ripple-Effekt in den Blöcken. Die Carry-Bits, die aus den Block-Additionen entstehen, werden ignoriert.

```
z[0:n] = CLA_Add<n,k> (x[0:n-1], y[0:n-1], c)
{
  for l = [0:n-k:k]
  {
    (cdead, z[l:l+k-1]) = Add<k> (x[l:l+k-1], y[l:l+k-1], c);
    (g, p) = GP_Group<k> (x[l:l+k-1], y[l:l+k-1]);
    c = or (g, and (c, p));
  }
  z[n] = c;
}
```

Die Komplexität der geblockten Carry-Lookahead-Addition ist:

	$k=2$		$k=4$		$k=8$		$k=16$	
n	Delay	Gates	Delay	Gates	Delay	Gates	Delay	Gates
8	**11**	76	13	78	16	79		
16	19	152	**17**	156	23	158	32	159
32	35	304	**25**	312	27	316	41	318
64	67	608	41	624	**35**	632	45	636

Man sieht, dass die beste Wahl für k so ist, dass $k^2 = n$ gilt. Die Berechnung der Carry-Bits hat dann eine Delay-Komplexität von $O(\log \frac{n}{k}) = O(\log k)$. Dazu kommt aber noch die Komplexität der Ripple-Carry-Addition der Blöcke, also $O(k)$. Insgesamt ergibt sich also $O(\log k + k) = O(k) = O(\sqrt{n})$.

Es stellt sich nun die Frage, ob es nicht geschickt wäre, innerhalb der Blöcke nicht die Ripple-Carry-Addition zu verwenden, sondern wiederum die Carry-Lookahead-Addition und auf dessen Blöcken wiederum und so weiter. Das führt zu folgendem rekursiven Aufbau:

```
1  (g, p, z[0:n-1]) = RCLA_Add<n> (x[0:n-1], y[0:n-1], c)
2  {
3    if n = 1
4    {
5      z[0] = xor (c, xor (x[0], y[0]));
6      (g, p) = GP (x[0], y[0]);
7    }
8    else
9    {
10     m = n/2;
11     (g0, p0, z[0:m-1]) = RCLA_Add<m> (x[0:m-1], y[0:m-1], c);
12     (g1, p1, z[m:n-1]) = RCLA_Add<n-m> (x[m:n-1], y[m:n-1], or (g0, and (c, p0)));
13     (g, p) = GP_Op (g0, p0, g1, p1);
14   }
15 }
```

Die G- und P-Bits werden gemeinsam mit dem Ergebnis berechnet, nach der gleichen Rekursion wie schon zuvor. Das Input-Carry-Bit für die obere Hälfte wird mit Hilfe der G- und P-Bits der unteren Hälfte berechnet (`or(g0,and(c,p0))`), um den Ripple-Effekt zu verhindern. Man beachte, dass weder `FullAdd` noch `HalfAdd` verwendet werden. Die eigentliche Addition passiert alleinig in Zeile 5. Es muss kein Output-Carry-Bit berechnet werden.

Die Komplexität der rekursiven Carry-Lookahead-Addition ist jetzt wirklich logarithmisch $O(\log n)$:

n	Delay	Gates
8	11	67
16	15	139
32	19	283
64	23	571

2.3 Conditional Sum

Eine “Brechstangenmethode” zur Vermeidung der Abhängigkeit vom hereinkommenden Carry-Bit ist, einmal das Ergebnis für $c = 0$ zu rechnen und einmal für $c = 1$ und dann das richtige Ergebnis auszuwählen, wenn das hereinkommende Carry-Bit verfügbar wird. “Auswählen” passiert in einer Schaltung mittels eines Multiplexers:

```
1  z[0:n-1] = Mux1<n> (x[0:n-1], y[0:n-1], s)
2  {
3    for i = [0:n-1]
4    { z[i] = or (and (x[i], not(s)), and (y[i], s)); }
5  }
```

Bemerkung: Leider produziert der Multiplexer ein Delay von 3, was sich auf die Komplexität des Algorithmus negativ auswirkt. Auch der Fanout-Wert für das Select-Bit `s` ist enorm. Bei gewissen Fertigungsmethoden können Multiplexer aber besser implementiert werden.

Man könnte das Prinzip jetzt natürlich auf Blöcke anwenden, innerhalb der Blöcke den normalen Ripple-Carry-Addierer verwenden und die (gemultiplexten) Carry-Bits der Blöcke auch in Ripple-Carry-Manier weitergeben. Wir wollen aber gleich ins Volle greifen und das Prinzip durch fortgesetztes Halbieren rekursiv anwenden. Das sieht dann so aus:

```
1  z[0:n] = CondSum_AddMux<n> (x[0:n-1], y[0:n-1], c)
2  {
3    if n > 1
4    {
5      m = n/2;
6      z0[0:m] = CondSum_Add<m>      (x[0:m-1], y[0:m-1], '0');
7      z0[m:n] = CondSum_AddMux<n-m> (x[m:n-1], y[m:n-1], z0[m]);
8      z1[0:m] = CondSum_Add<m>      (x[0:m-1], y[0:m-1], '1');
9      z1[m:n] = CondSum_AddMux<n-m> (x[m:n-1], y[m:n-1], z1[m]);
10     z[0:n] = Mux1<n+1> (z0[0:n], z1[0:n], c);
11   }
12   else
13   {
14     z0[0] = xor (x[0], y[0]);
15     z0[1] = and (x[0], y[0]);
16     z1[0] = not (z0[0]);
17     z1[1] = or (x[0], y[0]);
18     z[0:1] = Mux1<2> (z0[0:1], z1[0:1], c);
19   }
20 }
21
22 z[0:n] = CondSum_Add<n> (x[0:n-1], y[0:n-1], c)
23 {
24   if n > 1
25   {
26     m = n/2;
27     z[0:m] = CondSum_Add<m>      (x[0:m-1], y[0:m-1], c);
28     z[m:n] = CondSum_AddMux<n-m> (x[m:n-1], y[m:n-1], z[m]);
29   }
30   else
31   {
32     z[0:1] = FullAdd (x[0], y[0], c);
33   }
34 }
```

Wir brauchen eine Funktion (`CondSum_Add`) für den Fall, dass das hereinkommende Carry-Bit frühzeitig vorhanden ist, dh beim ursprünglichen Aufruf oder bei Angabe eines Versuchs-Carry-Bits, und eine Funktion (`CondSum_AddMux`), die – wie beschrieben – zwei Summen berechnet und dann erst die richtige auswählt.

In Zeile 14 bis 17 sehen wir das Prinzip auf einen Block der Länge 1 angewendet. `z0[0:1]` ist die Summe für $c = 0$ und `z1[0:1]` die Summe für $c = 1$. In Zeile 18 wird das richtige Ergebnis aufgrund des echten Carry-Bits selektiert.

Größere Blöcke werden in zwei Hälften zerteilt. In Zeile 6 wird die Summe der unteren Hälfte durch rekursiven Aufruf berechnet und zwar für $c = 0$. In Zeile 7 wird die obere Hälfte

berechnet und dabei das Carry-Bit aus der unteren Hälfte eingesetzt. Das Gleiche passiert in Zeile 8 und 9 für $c = 1$. In Zeile 9 wird das richtige Ergebnis aufgrund des echten Carry-Bits selektiert.

Das heißt, zwischen dem Ergebnis und dem hereinkommenden Carry-Bit liegt nur das Delay des Multiplexers. Das gilt aber auch für die rekursiven Aufrufe. Daher liegt zwischen dem Ergebnis der oberen Hälfte und dem Carry-Bit aus der unteren Hälfte *auch* nur das Multiplexer-Delay. Konsequent weitergedacht muss das Gesamtdelay also logarithmisch sein ($O(\log n)$).

n	Delay	Gates
8	12	209
16	15	663
32	18	2057
64	21	6303

Die Gatteranzahlen sind natürlich exorbitant. Sie werden hauptsächlich von den Multiplexern verursacht. Bei bestimmten Fertigungsmethoden kann dieses Problem aber reduziert werden.

2.4 Carry-Skip-Addition

Das Ziel, dass das Carry-Bit Blöcke "überspringen" soll, wird im folgenden Ansatz wörtlich genommen. Wird in einem Block ein Carry-Bit *erzeugt*, dann wird es ganz normal wie im Ripple-Carry-Addierer an den nächsten Block weitergegeben. Wird ein Carry-Bit aber *durchgereicht*, dann wird es direkt vom vorhergehenden an den nächsten Block weitergegeben.

Dazu muss zuerst für den Block das Propagate-Bit P (wie bekannt) errechnet werden. Dann wird das Ausgangs-Carry berechnet mit $c'_{\text{out}} = P \wedge c_{\text{in}} \vee c_{\text{out}}$, wobei c_{out} das Carry-Bit ist, das die Ripple-Carry-Addition auf dem Block ausgibt. Falls also $P = 1$ und $c_{\text{in}} = 1$, dann wird das Ausgangs-Carry gleich nach Verfügbarwerden von P und c_{in} auf 1 gesetzt und bleibt dort. Soll heißen: Es kommt nur das *effektive Delay* zum Tragen. Was aber, wenn $c_{\text{in}} = 0$ ist? In diesem Falle sollte das Ausgangs-Carry besser mittels $c'_{\text{out}} = (\bar{P} \vee c_{\text{in}}) \wedge c_{\text{out}}$ berechnet werden, dann gilt hier das gleiche Delay. Man muss also beide Fälle zusammenführen. Das führt zur Carry-Skip-Funktion $c'_{\text{out}} = (P \wedge c_{\text{in}} \vee c_{\text{out}}) \wedge (\bar{P} \vee c_{\text{in}})$. In beiden Fällen ändert das (zu spät kommende) c_{out} jetzt nichts mehr am (schon richtigen) Ergebnis. Ist aber $P = 0$, dann ist das Ergebnis einfach c_{out}.

P	c_{in}	c_{out}	c'_{out}	$\Delta(c'_{\text{out}})$ (eff. Delay)
1	1	1	1	$\Delta(c_{\text{in}}) + 3$
1	0	0	0	$\Delta(c_{\text{in}}) + 2$
0	$\star$	$\star$	c_{out}	$\Delta(c_{\text{out}}) + 2 \leq 2 \cdot \text{Blocksize} + 2$

Hier kommt noch der Code. Zuerst die Berechnung von P für einen Block analog zu `GP_Group`.

```
1 p = P<n> (x[0:n-1], y[0:n-1])
2 { if n > 1
3   { m = n/2;
4     p = and (P<m> (x[0:m-1], y[0:m-1]), P<n-m> (x[m:n-1], y[m:n-1]));
5   }
6   else
7   { p = xor (x[0], y[0]); }
8 }
```

Für einen Block werden nun folgende Berechnungen ausgeführt.

```
1 (cout, z[0:n-1]) = CarrySkip_Group<n> (x[0:n-1], y[0:n-1], cin)
2 {
3   (cg, z[0:n-1]) = Add<n> (x[0:n-1], y[0:n-1], cin);
4   p = P<n> (x[0:n-1], y[0:n-1]);
5   cout = and (or (cg, and (p, cin)), or (not (p), cin));
6 }
```

In Zeile 3 wird die normale Ripple-Carry-Addition auf dem Block ausgeführt. Das dabei berechnete c_{out} wird in `cg` gespeichert. In Zeile 4 wird das Propagate-Bit P berechnet. In Zeile 5 wird nun das neue Ausgangs-Carry nach obiger Formel berechnet.

Die zu addierenden Wörter müssen nun nur noch in Blöcke zerlegt werden und diese durch obige Funktion gejagt werden. Hier ein Beispiel für 16-Bit-Wörter.

```
1 z[0:16] = CarrySkip_Add_16 (x[0:15], y[0:15], c)
2 {
3   (c, z[0:1]) = CarrySkip_Group<2> (x[0:1], y[0:1], c);
4   (c, z[2:4]) = CarrySkip_Group<3> (x[2:4], y[2:4], c);
5   (c, z[5:9]) = CarrySkip_Group<5> (x[5:9], y[5:9], c);
6   (c, z[10:13]) = CarrySkip_Group<4> (x[10:13], y[10:13], c);
7   (c, z[14:15]) = CarrySkip_Group<2> (x[14:15], y[14:15], c);
8   z[16] = c;
9 }
```

Wir könnten zwar auch gleich große Blöcke nehmen (die optimale Größe wäre dabei übrigens $\sqrt{n/2}$), es stellt sich aber heraus, dass es besser ist, wenn die mittleren Blöcke größer sind als die äußeren. Die optimale Größe der Blöcke hängt vom Delay der verwendeten Addition (hier Ripple-Carry) und vom Delay der Carry-Skip-Funktion ab. Die Blockgrößen zu ermitteln ist nicht trivial und am besten mittels ganzzahliger Optimierungsverfahren zu bewerkstelligen.

Bei der Auswertung der Gesamtdelays dieses Algorithmus' ist zu beachten, dass die *effektiven Delays* verwendet werden müssen. Da diese datenabhängig sind, müssten eigentlich alle Input-Wörter durchprobiert werden und das maximale Delay ermittelt werden. Bei diesem Algorithmus kann es sogar sein, dass das maximale Delay nicht in `z[n-1]` sondern in einem anderen Bit auftritt. Auf folgende Ergebnisse gibt es also keine Gewähr. Auch die Komplexität $O(?)$ lässt sich analytisch nicht leicht ermitteln.

n	Delay	Gates
8	13	68
16	19	132
32	27	256
64	38	492

Es ist zu bemerken, dass die Delays durch die Carry-Skip-Funktion dominiert werden. Je nach Fertigungstechnik kann diese optimiert implementiert werden, zB mittels Switches in Form von Pass-Transistoren (Manchester-Adder). Dadurch werden die Delays noch signifikant verbessert. (Die optimalen Blockgrößen müssen dann auch neu berechnet werden.) Dadurch wird der Algorithmus konkurrenzfähig, vor allem weil er weniger Gatter/Bauteile benötigt als der Carry-Lookahead-Addierer.

2.5 Addition mehrerer Summanden

Wenn man mehrere Summanden gegeben hat, ist es natürlich nicht besonders geschickt, zuerst die ersten zwei Summanden zu addieren, dann den dritten und die Zwischensumme usw. Die Komplexität würde mit der Anzahl der Summanden linear wachsen.

2.5.1 Straight Forward

Ein etwas besserer Ansatz ist, die Summanden in zwei Gruppen zu teilen, dann die Summe innerhalb jeder Gruppe zu berechnen und die zwei Summen dann zu addieren. Innerhalb der Gruppe kann man natürlich den gleichen Ansatz verfolgen, was zu einem rekursiven Schema führt.

```
1  z[0:n+m-1] = MultiAdd<n,m> (x[0:2^m-1,0:n-1])
2  {
3    if m = 0
4    { z[0:n-1] = x[0,0:n-1]; }
5    else
6    { (g, p, z[0:n+m-2]) = RCLA_Add<n+m-1>
7        ( MultiAdd<n,m-1> (x[0:2^(m-1)-1,0:n-1])
8        , MultiAdd<n,m-1> (x[2^(m-1):2^m-1,0:n-1])
9        , '0'
10       );
11     z[n+m-1] = g;
12   }
13 }
```

Hier werden 2^m Summanden der Länge n addiert. Das Ergebnis hat dann $n+m$ Bits, da bei jeder Verdoppelung der Summandenanzahl der Wertebereich des Ergebnisses verdoppelt wird und daher ein Bit mehr zur Darstellung benötigt wird. In Zeile 7 werden die ersten $\frac{2^m}{2} = 2^{m-1}$ Summanden rekursiv summiert, in Zeile 8 die zweite Gruppe. Die zwei Summen werden dann mit dem Carry-Lookahead-Addierer `RCLA_Add` addiert. In Zeile 4 erfolgt der Rekursionsabbruch für den Fall eines einzelnen "Summanden", der dann einfach in die Ausgabebits kopiert wird.

Nachdem die Berechnung der niederwertigsten Summenbits auf diese Weise in logarithmischer Komplexität vonstatten geht und die restlichen Bits des finalen Ergebnisses danach auch in logarithmischer Komplexität berechnet werden (wegen `RCLA_Add`), ergibt sich ein Gesamtdelay von $O(\log(2^m) + \log n) = O(m + \log n)$.

	$2^m = 2^1 = 2$		$2^m = 2^2 = 4$		$2^m = 2^3 = 8$		$2^m = 2^4 = 16$	
n	Delay	Gates	Delay	Gates	Delay	Gates	Delay	Gates
8	11	67	19	210	26	505	32	1104
16	15	139	26	426	34	1009	43	2184
32	19	283	33	858	44	2017	55	4344
64	23	571	40	1722	54	4033	68	8664

2.5.2 Carry-Save-Addition

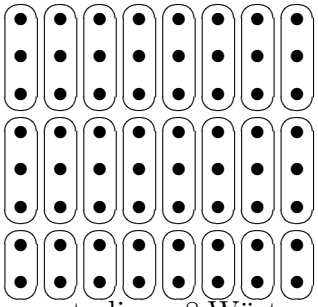
Eine ausgefeiltere Methode ist die Carry-Save-Addition. Sie beruht auf der Idee, dass man zuerst alle Bits mit gleicher Wertigkeit addieren und sich das Weitergeben der Carry-Bits bis zum Schluss aufheben sollte (daher Carry-Save).

Die wichtigste Erkenntnis dabei ist, dass man den Volladdierer (`FullAdd`) als 3-Bit-Zähler betrachten kann. Er zählt die 1-en in den 3 Eingabebits und gibt die Anzahl als 2-Bit-Binärzahl aus. Er wird daher korrekt als $(3, 2)$-Zähler bezeichnet. Der Halbaddierer ist demnach ein $(2, 2)$-Zähler.

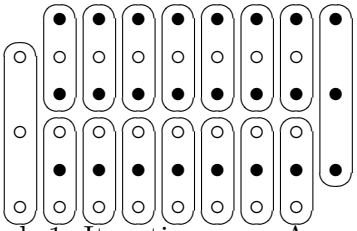
Wenn man jetzt also 3 Bits mit gleicher Wertigkeit 2^k in einen $(3, 2)$-Zähler steckt, ersetzt man die 3 Bits durch ein Bit mit gleicher Wertigkeit 2^k und ein Bit mit Wertigkeit 2^{k+1}. Man geht daher folgendermaßen vor: Man fasst möglichst viele gleichwertige Bits in Dreiergruppen zusammen, die übrigen in Zweiergruppen. Einzelne bleiben stehen. Diese Gruppen schickt man durch die Zähler und erhält an jeder Stelle für jede Gruppe ein Ergebnisbit und zusätzlich die Carry-Bits der vorhergehenden Stelle. Dadurch reduziert sich die Anzahl der Bit an jeder Stelle um den Faktor $\frac{2}{3}$.

Diesen Vorgang wiederholt man so oft, bis die Anzahl der Bits an jeder Stelle ≤ 2 ist. Für 8 Summanden ergeben sich also nacheinander folgende Bitanzahlen an jeder Stelle: $8 \rightarrow \lceil 8 \cdot \frac{2}{3} \rceil = 6 \rightarrow 4 \rightarrow 3 \rightarrow 2$. Danach wird die Bitanzahl nicht mehr verringert und der Prozess wird zu einem suboptimalen Ripple-Carry-Prozess mit Halbaddierern. Daher wird an dieser Stelle ein herkömmlicher Addierer (am besten mit logarithmischer Komplexität) eingesetzt, um das Endergebnis zu berechnen.

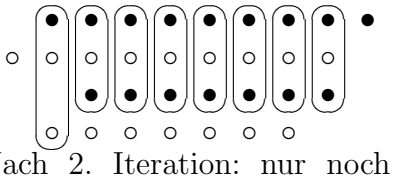
Folgendes Beispiel demonstriert das Prinzip für 8 Summanden mit je 8 Bit.



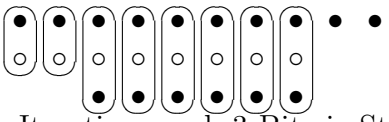
Anfangsstadium: 8 Wörter mit je 8 Bit, 3 Gruppen an jeder Stelle.



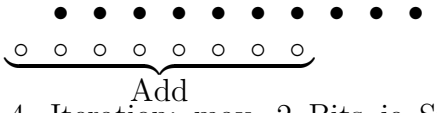
Nach 1. Iteration: • = Ausgabebit eines Zählers, ∘ = Carry-Bit eines Zählers. 8 Bits wurden auf 6 reduziert.



Nach 2. Iteration: nur noch max. 4 Bits pro Stelle. Je 1 Bit passt nicht in eine Gruppe.



Nach 3. Iteration: noch 3 Bits je Stelle inklusive der einzelnen von vorher.



Nach 4. Iteration: max. 2 Bits je Stelle. Diese werden mit herkömmlichem Addierer addiert.

Um das Prinzip in Alluvion zu formulieren, überlegen wir uns eine verallgemeinerte Addition, wo für jede Stelle i eine individuelle Anzahl d_i von Bits übergeben wird. Die Prozedur soll versuchen, möglichst viele Dreiergruppen pro Stelle i mit `FullAdd` zu reduzieren, restliche Zweiergruppen mit `HalfAdd`. Die dabei entstehenden Carry-Bits werden in der nächsten Iteration an der Stelle $i + 1$ hinzugefügt. Das Ganze soll so lange wiederholt werden, bis an keiner Stelle mehr als zwei Bits vorhanden sind.

```
1 z[0:n+m-1] = CS_Add_Gen<n,m,d[0:n-1],maxd> (x[0:maxd-1,0:n-1])
2 {
3   for i = [n:n+m-1] { d[i] = 0; }
4
5   while (maxd > 2)
6   {
7     maxd = 0; oci = 0; oc[0] = '0';
```

```
8     for i = [0:n+m-1]
9     {
10      ci = 0; di = 0;
11      for j = [0:d[i]-3:3]
12      { (c[ci], x[di,i]) = FullAdd (x[j:j+2,i]);
13        ci = ci + 1; di = di + 1;
14      }
15      if d[i] % 3 = 2
16      { (c[ci], x[di,i]) = HalfAdd (x[d[i]-2:d[i]-1,i]);
17        ci = ci + 1; di = di + 1;
18      }
19      if d[i] % 3 = 1
20      { x[di,i] = x[d[i]-1,i]; di = di + 1; }
21      x[di:di+oci-1,i] = oc[0:oci-1]; di = di + oci;
22      d[i] = di;
23      if di > maxd { maxd = di; }
24      oc[0:ci-1] = c[0:ci-1]; oci = ci;
25    }
26  }
27
28  u = 0; while d[u] = 1 { u = u + 1; }
29  o = u; while d[o] = 2 & o < n+m { o = o + 1; }
30  z[0:u-1] = x[0,0:u-1];
31  for i = [o:n+m-1] { x[1,i] = '0'; if d[i] = 0 { x[0,i] = '0'; } }
32  (dum1, dum2, z[u:n+m-1]) = RCLA_Add<n+m-u> (x[0,u:n+m-1],x[1,u:n+m-1],'0');
33 }
```

Die Parameter von `CS_Add_Gen` sind: `n` – die Wortlänge der Summanden, `m` – die Anzahl der (Carry-)Bits, die über `n` hinaus im Ergebniswort stehen soll (dh das Ergebniswort hat `n+m` Bit), `d[i]` – die Anzahl der Bits an der Stelle `i`, `maxd` – das Maximum aller `d[i]`, das ist die maximale Anzahl der Bits pro Stelle, `x[0:maxd-1,0:n-1]` – die zu addierenden Bits als zweidimensionales Array. Wenn ein `d[i]<maxd` ist, können Teile des Arrays uninitialisiert sein und werden ignoriert.

In Zeile 11-14 werden Dreiergruppen mit `FullAdd` reduziert. Die Ergebnisbits werden gleich in `x` zurückgeschrieben. `di` iteriert die Position dieser Ergebnisbits. Die Carry-Bits werden in `c` gesammelt. `ci` iteriert die Position der Carry-Bits in `c`. Die Carry-Bits werden in Zeile 21 und 24 in die `x`-Bits der nächsten Stelle übernommen. Durch Kopieren in `oc` werden sie in die nächste `i`-Iteration übernommen. In Zeile 15-17 werden übrigbleibende Zweiergruppen reduziert, sofern vorhanden. In Zeile 19-20 werden einzelne übrigbleibende Bits durch Kopieren in die nächste Iteration übernommen.

Schlussendlich werden in Zeile 32 jene Stellen mit `RCLA_Add` addiert, in denen $d_i = 2$ ist. Dafür muss zuvor der Beginn u und das Ende o dieser Stellen ermittelt werden. Zum Verständnis des Algorithmus' sei noch empfohlen, zwischen Zeile 25 und 26 mittels `print d[0:n+m-1]` die sich reduzierenden Bitanzahlen zu beobachten.

Um die Carry-Save-Addition noch mit der Straight-Forward-Addition vergleichen zu können, brauchen wir noch folgende Funktion, die das gleiche Interface wie `MultiAdd` besitzt und `CS_Add_Gen` benutzt, um 2^m gleich große Summanden zu summieren.

```
z[0:n+m-1] = CS_Add<n,m> (x[0:2^m-1,0:n-1])
{
  for i = [0:n-1] { d[i] = 2^m; }
```

```
4   z[0:n+m-1] = CS_Add_Gen<n,m,d[0:n-1],2^m> (x[0:2^m-1,0:n-1]);
5 }
```

Die Anzahl der Iterationen, die für die Reduktion benötigt wird, hängt logarithmisch von der Anzahl der Summanden ab, da in jeder Iteration die Anzahl der Bits um einen Faktor kleiner wird. Wird für die finale Addition ein Addierer mit logarithmischer Komplexität verwendet, so ist auch das Gesamtdelay von logarithmischer Komplexität.

	$2^m = 2^1 = 2$		$2^m = 2^2 = 4$		$2^m = 2^3 = 8$		$2^m = 2^4 = 16$	
n	Delay	Gates	Delay	Gates	Delay	Gates	Delay	Gates
8	13	76	19	153	22	317	27	649
16	17	148	23	305	26	645	31	1313
32	21	292	27	609	30	1301	35	2641
64	25	580	31	1217	34	2613	39	5297

Die Delays sind besser als im Straight-Forward-Ansatz und die Gatterzahlen sind auch kleiner.

3 Multiplikation

3.1 Naive Multiplikation

Die Multiplikation zweier Zahlen $x = \sum_{i=0}^{n-1} 2^i x_i$ und $y = \sum_{j=0}^{n-1} 2^j y_j$ ist

$$x \cdot y = x \cdot \sum_{j=0}^{n-1} 2^j y_j = \sum_{j=0}^{n-1} x y_j 2^j \,,$$

wobei der Ausdruck xy_j2^j für $y_j = 1$ einfach gleich x um j Stellen nach links geschoben ist, bzw. gleich 0 für $y_j = 0$. Die Multiplikation entspricht also $n-1$ kumulativen Additionen mit x oder 0 als Summanden. Die höchstwertige Stelle, die summiert wird, ist $x_{n-1}2^{n-1}y_{n-1}2^{n-1} = x_{n-1}y_{n-1}2^{2n-2}$. Weil noch eine Carry-Bit entstehen kann, ist die höchstwertige Stelle im Produkt 2^{2n-1}. Das Ergebnis hat also $2n$ Bit.

```
1 z[0:2*n-1] = Mul<n> (x[0:n-1], y[0:n-1])
2 {
3   z[0:n-1] = And<n> (x[0:n-1], y[0]);
4   z[n] = '0';
5
6   for j = [1:n-1]
7   { z[j:j+n] = Add<n> (z[j:j+n-1], And<n> (x[0:n-1], y[j]), '0'); }
8 }
```

In Zeile 3 wird das vorläufige Ergebnis auf y_0x gesetzt. In den Zeilen 6 und 7 werden alle y_jx2^j aufaddiert und zwar an den Stellen $[j, j+n-1]$, was einer Multiplikation mit 2^j entspricht. Man beachte, dass jedes Mal auch ein Carry-Bit entsteht, das an die Stelle $j+n$ geschrieben wird und so bei der jeweils nächsten Addition wieder mitaddiert wird.

Die Multiplikation mit y_j entspricht einer And-Operation von jedem x-Bit mit y_j. Dafür wird folgende Array-And-Funktion verwendet.

```
z[0:n-1] = And<n> (x[0:n-1], y)
{
  for i = [0:n-1]
  { z[i] = and (x[i], y); }
}
```

Obwohl n Additionen hintereinander ausgeführt werden müssen, addiert sich das Delay der Additionen nicht, da mit der Addition der unteren Bits schon begonnen werden kann, während die oberen Bits noch berechnet werden. Daher ist die Komplexität der naiven Addition bezüglich des Delays linear $O(n)$. Die Komplexität bezüglich der Gatteranzahl ist aber natürlich quadratisch $O(n^2)$, da man n Addierer braucht mit jeweils $O(n)$ Gatter.

n	Delay	Gates
4	18	76
8	42	344
16	90	1456
32	186	5984
64	378	24256

3.2 Multiplikation mit Vorzeichen

Während alle Additionsalgorithmen, die wir kennengelernt haben, automatisch auch für negative Zahlen funktionieren, zumindest für Zahlen im 2-er-Komplement, ist das für die Multiplikation nicht mehr so selbstverständlich.

n-Bit-Zahlen im 2-er-Komplement liegen im Bereich $[-2^{n-1}, 2^{n-1}-1]$. Das Produkt zweier solcher Zahlen liegt daher im Bereich $[-2^{n-1}(2^{n-1}-1), -2^{n-1}\cdot -2^{n-1}] = [-2^{2n-2}+2^{n-1}, 2^{2n-2}]$. Es sind daher ebenfalls $2n$ Bit notwendig, um das Produkt darzustellen.

Es müssten eigentlich alle x-Summanden auf $2n-1-j$ Bit aufgefüllt werden (durch Kopieren des Vorzeichenbits). Die Zwischenergebnisse hätten dann immer $2n-1$ Bits und es wäre bei Addition des j-ten Summanden ein $(2n-1-j)$-Bit-Addierer notwendig. Wir greifen stattdessen zu folgendem Trick. An Stelle von negativen Summanden der Form $11111xxxxx$ addieren wir $0xxxxx$. Das entspricht einer zusätzlichen Addition von 2^{n-1}. An Stelle von positiven Summanden der Form $0xxxxx$ addieren wir $1xxxxx$. Auch das entspricht einer zusätzlichen Addition von 2^{n-1}. Wir invertieren also bei allen Summanden einfach das Vorzeichenbit. Am Ende haben wir also $2^{n-1}+2^n+\ldots+2^{2n-2} = 2^{2n-1}-2^{n-1}$ zusätzlich addiert. Um das zu korrigieren, müssen wir $2^{2n-1}-2^{n-1}$ subtrahieren. Dazu addieren wir 2^{n-1} zum ersten Summanden, indem wir statt $\bar{s}xxxxx$ $\bar{s}sxxxxx$ setzen. Schließlich subtrahieren wir 2^{2n-1}, indem wir das Vorzeichenbit des Endergebnisses invertieren.

Außerdem ist darauf zu achten, dass y_{n-1} als Vorzeichenbit eine Wertigkeit von -2^{n-1} besitzt. Daher muss statt der letzten Addition eine Subtraktion stattfinden. Um nicht extra eine Subtraktion programmieren zu müssen, addieren wir das 2-er-Komplement von x. Geschickterweise bilden wir dazu nur das 1-er-Komplement durch Invertieren aller Bits und addieren den fehlenden 1er während der letzten Addition, indem wir ihn einfach als Carry-Bit einschmuggeln.

```
z[0:2*n-1] = MulSigned<n> (x[0:n-1], y[0:n-1])
{
  z[0:n-1] = And<n> (x[0:n-1], y[0]);
  z[n] = not (z[n-1]);
```

```
5
6   for i = [1:n-2]
7   { a[0:n-1] = And<n> (x[0:n-1], y[i]);  a[n-1] = not (a[n-1]);
8     z[i:i+n] = Add<n> (z[i:i+n-1], a[0:n-1], '0');
9   }
10
11  for i = [0:n-1] { ix[i] = not(x[i]); }  // x invertieren wegen Komplement
12  a[0:n-1] = And<n> (ix[0:n-1], y[n-1]);  a[n-1] = not (a[n-1]);
13  // Subtraktion: Addition von x invertiert plus Carry=1, falls y[n-1]=1
14  z[n-1:2*n-1] = Add<n> (z[n-1:2*n-2], a[0:n-1], y[n-1]);
15  z[2*n-1] = not (z[2*n-1]);
16 }
```

In Zeile 3-4 wird der erste Summand präpariert, indem das invertierte Vorzeichenbit vorne drangehängt wird (Vorgangsweise siehe oben). In Zeile 7 und 12 werden die anderen Summanden präpariert, indem das Vorzeichenbit invertiert wird. In Zeile 11 wird das 1er-Komplement von x gebildet, das in Zeile 14 als 2er-Komplement addiert wird, indem ein künstliches Carry-Bit in die Addition hineingeschoben wird. Sowohl der Summand als auch das künstlichen Carry-Bit müssen aber auf 0 gesetzt werden, wenn der Multiplikator y *nicht* negativ ist, dh wenn $y_{n-1} = 0$ ist. Daher muss das künstliche Carry-Bit gleich y_{n-1} sein.

3.3 Verwendung höherer Basen

Die Idee hier ist, die Zahlen zB zur Basis 4 (statt 2) zu betrachten. Dadurch reduziert sich die Anzahl der benötigten Additionen um die Hälfte. Praktisch heißt das, die Produkte $0x$, $1x$, $2x$ und $3x$ im vorhinein auszurechnen und dann aufgrund von jeweils 2 y-Bits eines dieser Produkte auszuwählen und zu addieren. Die Position dieser Additionen verschiebt sich natürlich um jeweils 2 Bits.

Zum Auswählen der Teilprodukte brauchen wir vorerst einen Multiplexer mit mehreren Select-Eingängen.

```
1  z[0:n-1] = Mux<n,m> (x[0:2^m-1,0:n-1], s[0:m-1])
2  {
3    for i = [0:m-1]
4    {
5      snot = not (s[i]);
6      for j = [0:2^(m-i-1)-1]
7      {
8        a = 2^(i+1)*j;
9        b = a+2^i;
10       for k = [0:n-1]
11       { x[a,k] = or (and (x[a,k], snot), and (x[b,k], s[i])); }
12     }
13   }
14   z[0:n-1] = x[0,0:n-1];
15 }
```

Die Funktion `Mux<n,m>` wählt aus 2^m Wörtern der Länge n das s-te aus. Hier ist nun die Multiplikation zur Basis 4.

```
z[0:2*n-1] = MulRadix4<n> (x[0:n-1], y[0:n-1])
{
  for i = [0:n+1] { p0[i] = '0'; }
  p1[0:n-1] = x[0:n-1]; p1[n:n+1] = '00';
  p2[1:n] = x[0:n-1]; p2[0] = '0'; p2[n+1] = '0';
  p3[0:n+1] = Add<n+1> ('0',x[0:n-1], x[0:n-1],'0', '0');
  p[0:4*(n+2)-1] = (p3[0:n+1], p2[0:n+1], p1[0:n+1], p0[0:n+1]);

  z[0:n+1] = Mux<n+2,2> (p[0:4*(n+2)-1], y[0:1]);
  z[n+2:n+3] = '00';

  for j = [2:n-2:2]
  {
    z[j:j+n+2] = Add<n+2> (z[j:j+n+1], Mux<n+2,2> (p[0:4*(n+2)-1], y[j:j+1]), '0');
    z[j+n+3] = '0';
  }
}
```

In `p` werden die Teilprodukte $0x$ bis $3x$ berechnet. Der Rest ist analog zur Multiplikation zur Basis 2, außer dass als Summand eines der Teilprodukte mittels Mux aufgrund der Bits (y_j, y_{j+1}) ausgewählt wird, wobei j in 2-er-Schritten y durchläuft.

Die Delay-Komplexität ist nach wie vor linear. Dadurch, dass die Anzahl der Additionen um die Hälfte reduziert wurde, ist zwar eine Verbesserung zu erwarten, die Multiplexer fressen aber viel davon wieder auf.

n	Delay	Gates
4	22	167
8	42	563
16	82	2027
32	162	7643
64	322	29627

3.4 Verwendung der Carry-Save-Addition

Im Prinzip ist die Multiplikation nur eine Addition von n Summanden mit je n Bits, die jeweils um 1 Bit nach links versetzt sind. Schaut man sich nun an, wie viele Bits an der Stelle i summiert werden müssen, erhält man $1, 2, 3, \ldots, n, \ldots, 3, 2, 1$. Das ist eindeutig ein Fall für den verallgemeinerten Carry-Save-Addierer.

```
z[0:2*n-1] = CS_Mul<n> (x[0:n-1], y[0:n-1])
{
  for i = [0:2*n-2] { d[i] = 0; }
  for i = [0:n-1]
  { for j = [0:n-1]
    { xy[d[i+j],i+j] = and (x[i], y[j]);
      d[i+j] = d[i+j] + 1;
    }
  }

```

```
11   z[0:2*n-1] = CS_Add_Gen<2*n-1,1,d[0:2*n-2],n> (xy[0:n-1,0:2*n-2]);
12 }
```

In Zeile 4-7 werden die Summanden-Bits für die Stellen $0 \ldots 2n-1$ in `xy` eingetragen. In Zeile 11 wird der verallgemeinterte Carry-Save-Addierer aufgerufen. Dabei kann sich ein zusätzliches Carry-Bit ergeben, wodurch das Ergebniswort $2n$ Bits lang ist.

Da die Carry-Save-Addition logarithmisch sowohl bezüglich der Wortlänge (hier $2n-1$) als auch bezüglich der Anzahl der Summanden ist (hier n), ergibt sich ein Gesamt-Delay von $O(\log(2n-1)+\log n) = O(\log n)$.

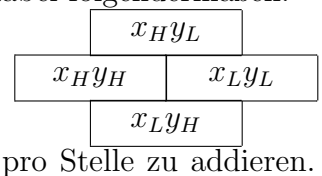
n	Delay	Gates
4	16	88
8	26	388
16	34	1612
32	44	6446
64	52	25468

3.5 Zurückführung auf kleinere Multiplizierer

Teilt man die zu multiplizierenden Wörter in zwei Hälften in der Form $x = x_H 2^m + x_L$, $y = y_H 2^m + y_L$, wobei $m = \frac{n}{2}$, dann ergibt sich

$$x \cdot y = x_H y_H 2^{2m} + (x_H y_L + x_L y_H) 2^m + x_L y_L \,.$$

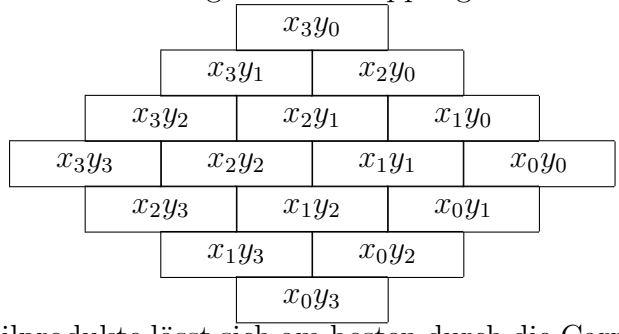
Die Teilprodukte überlappen sich dabei folgendermaßen.



Es sind also noch max. 3 Bits pro Stelle zu addieren. Dieses Prinzip lässt sich noch verallgemeinern. Zerlegt man die Wörter in Teile der Länge m, so dass $x = \sum_{i=0}^{n/m-1} x_i 2^{im}$, dann bekommt man

$$x \cdot y = \sum_{i=0}^{n/m-1} \sum_{j=0}^{n/m-1} x_i y_j 2^{(i+j)m} \,.$$

Das ergibt zB für $n = 16$ und $m = 4$ folgende Überlappung:



Die Summe dieser Teilprodukte lässt sich am besten durch die Carry-Save-Addition berechnen. Das sieht dann so aus:

```
1  z[0:2*n-1] = Partial_Mul<n,m> (x[0:n-1], y[0:n-1])
2  {
3    for i = [0:2*n-1] { d[i] = 0; }
```

```
4   for i = [0:n-m:m]
5   { for j = [0:n-m:m]
6     { zz[0:2*m-1] = CS_Mul<m> (x[i:i+m-1], y[j:j+m-1]);
7       for k = [0:2*m-1]
8       { xy[d[i+j+k],i+j+k] = zz[k];
9         d[i+j+k] = d[i+j+k] + 1;
10      }
11    }
12  }
13  z[0:2*n-1] = CS_Add_Gen<2*n,0,d[0:2*n-1],2*n/m-1> (xy[0:2*n/m-2,0:2*n-1]);
14 }
```

In Zeile 6 werden die Teilprodukte gebildet und in Zeile 8 in das Array `xy` eingetragen. In `d[i]` wird die Anzahl der Bits für jede Stelle `i` mitgezählt. In Zeile 13 wird `xy` an den Carry-Save-Addierer übergeben.

Von der Komplexität her bringt diese Methode leider nichts.

	$m = 4$		$m = 8$		$m = 16$		$m = 32$	
n	Delay	Gates	Delay	Gates	Delay	Gates	Delay	Gates
8	31	486						
16	43	2075	45	1834				
32	53	8375	57	7583	56	7026		
64	61	33459	67	30381	68	28583	70	26954

Manchmal sprechen aber Implementierungsgründe dafür, dieses Prinzip einzusetzen. ZB können kleinere Multiplizierer schon vorhanden oder leichter implementierbar sein.

4 Division

4.1 Normale Division

Wir gehen davon aus, dass der Dividend so groß ist, wie das Produkt bei der Multiplikation, also doppelt so groß ($2n$) wie Divisor und Quotient (n). Wie beim händischen Dividieren ziehen wir den Divisor von den oberen Stellen des Dividenden ab, falls diese größer sind als der Divisor. In diesem Fall wird das oberste Quotientenbit auf 1 gesetzt. Andernfalls bleibt der Rest unverändert und das Quotientenbit wird auf 0 gesetzt. Danach fährt man mit dem Rest plus dem nächsten Dividendenbit auf gleiche Weise fort. Falls der Zwischenrest mehr als n Bit beansprucht, wird auch der gesamte Divisionsrest zu groß sein und es tritt ein Overflow auf.

```
1  (q[0:n-1], r[0:n-1], o) = Div<n> (x[0:2*n-1], d[0:n-1])
2  {
3    for i = [0:n-1] { nd[i] = not (d[i]); }
4    r[0:2*n-1] = x[0:2*n-1];
5    for i = [n-1:0:-1]
6    {
7      (g, p, rr[i:i+n-1]) = RCLA_Add<n> (r[i:i+n-1], nd[0:n-1], '1');
8      c = or (g, p);
9      rr[i+n] = xor (not (r[i+n]), c);
10     q[i] = or (r[i+n], c);
11     r[i:i+n-1] = Mux1<n> (r[i:i+n-1], rr[i:i+n-1], q[i]);
```

```
12      oo = and (q[i], rr[i+n]);
13      if i = n-1 { o = oo; } else { o = or (o, oo); }
14    }
15  }
```

Dieser Code implementiert die Division. `x` ist der Dividend mit Länge $2n$, `d` der Divisor. `q` ist der Quotient und `r` der Divisionsrest. Die Division operiert auf `r`, das am Anfang (Zeile 4) auf `x` gesetzt wird. Die Schleife `i` iteriert die Quotientenbits von links nach rechts. In Zeile 7 wird der Divisor vom Rest subtrahiert und es ergibt sich ein potentiell neuer Rest `rr`. Die Subtraktion passiert wieder durch Addition der invertierten Divisorbits (`nd`, Zeile 3) mit auf 1 gesetztem Carry-In. Falls der Rest ≤ 0 ist, soll das i-te Quotientenbit 0 sein (Zeile 10) und der alte Rest beibehalten werden (Zeile 11).

In der Subtraktion wird das n-te Bit `r[i+n]` extra behandelt, da dieses Bit in `d` immer 0 und daher in `nd` immer 1 ist. Das Bit `rr[i+n]` der Differenz ergibt sich daher so wie in Zeile 9 und das Vorzeichen so wie in Zeile 10 (nachprüfen).

Ein Overflow ensteht, wenn ein Rest $n+1$ Bits beansprucht. Das kann nur dann passieren, wenn eine Subtraktion stattfindet und das oberste Bit `rr[i+n]` der Differenz 1 ist (Zeile 12). Die Overflow-Bits `oo` jeder Iteration müssen zusammengeodert werden (Zeile 13).

Weil der ursprügliche Rest im Falle einer negativen Differenz "wiederhergestellt" wird, heißt das Prinzip auch "Restoring Division".

Jeder Schritt hat eine Komplexität von $O(\log n)$ und es sind n solche Schritte notwendig. Da ein Quotientenbit potentiell von jedem Bit des Restes aus der vorhergehenden Iteration abhängt, summieren sich die Delays der Subtraktionen auf, weshalb die Komplexität $O(n \log n)$ ist. Aus diesem Grund wird eine Division meist auf mehrere Taktzyklen verteilt.

n	Delay	Gates
4	41	215
8	98	847
16	228	3359
32	533	13375
64	1253	53375

4.2 Dividend einfacher Länge

Meistens ist der Dividend nicht in doppelter Länge gegeben sondern hat ebenfalls nur n Bit. In diesem Fall ist kein Overflow möglich (außer bei Division durch 0 natürlich). Man könnte natürlich die obige Division nehmen und die oberen Dividendenbits auf 0 setzen. Durch geschicktere Implementation kann man aber einiges an Delay sparen.

Wenn der Divisor d an der Stelle i subtrahiert wird, dann setzt sich der aktuelle Rest aus `r[i:n-1]` und i 0en an führender Stelle zusammen. Die Differenz ist also genau dann positiv, wenn die Bits `d[n-i:n-1]` gleich 0 sind und die Differenz von `r[i:n-1]` und `d[0:n-i-1]` positiv ist.

Wir benötigen also nacheinander die Information, ob die führenden i Bits von d gleich 0 sind. Dazu verwenden wir folgende Funktion, die die Bits eines Arrays kumulativ zusammenodert, dh es soll $y_k = x_0 \vee x_1 \vee \ldots \vee x_k$ sein. Das ist in logarithmischer Komplexität implementierbar.

```
1  y[0:n-1] = OrKumul<n> (x[0:n-1])
2  {
3    if n = 1
```

```
4   { y[0] = x[0]; }
5   else
6   { m = n/2;
7     y[0:m-1] = OrKumul<m> (x[0:m-1]);
8     y[m:n-1] = OrKumul<n-m> (x[m:n-1]);
9     for i = [m:n-1] { y[i] = or (y[i], y[m-1]); }
10  }
11 }
```

Die Funktion teilt das Array in eine obere und untere Hälfte. Für beide wird das kumulative Oder-Array berechnet. Die Bits des oberen Arrays müssen dann noch jeweils mit dem obersten Bit des unteren Arrays verknüpft werden, was für jedes Bit nur ein Delay von 1 hinzufügt.

Wenn wir nun `o[n-1:0:-1] = OrKumul<n> (d[n-1:0:-1])` berechnen, dann liefert `not(o[k])` die Information, ob die Bits `d[k:n-1]` alle 0 sind. Damit können wir die Division implementieren.

```
1  (q[0:n-1], r[0:n-1]) = Single_Div<n> (x[0:n-1], d[0:n-1])
2  {
3    for i = [0:n-1] { nd[i] = not (d[i]); }
4    r[0:n-1] = x[0:n-1];
5    o[n-1:0:-1] = OrKumul<n> (d[n-1:0:-1]);  o[n] = '0';
6    for i = [n-1:0:-1]
7    {
8      (g, p, rr[i:n-1]) = RCLA_Add<n-i> (r[i:n-1], nd[0:n-i-1], '1');
9      q[i] = and (not (o[n-i]), or (g, p));
10     r[i:n-1] = Mux1<n-i> (r[i:n-1], rr[i:n-1], q[i]);
11   }
12 }
```

Der Algorithmus entspricht der Restoring Division. Die Unterschiede sind, dass in Zeile 8 nur $n - i$ statt n Bits subtrahiert werden und in Zeile 9 `q[i]` unter Verwendung von `o[n-i]` ermittelt wird. Außerdem kann auf die Berechnung des Overflow-Bits verzichtet werden.

Die Komplexität im k-ten Schritt ist $O(\log k)$. Die Gesamtkomplexität ist daher $O(\log 1 + \log 2 + \ldots + \log n)$ und das ist leider ebenfalls $O(n \log n)$.

n	Delay	Gates
4	34	130
8	80	472
16	189	1784
32	444	6912
64	1032	27168

4.3 Non-Restoring Division

Jetzt wollen wir uns die Multiplexer-Operation nach der Differenzbildung ersparen. Daher muss das Quotientenbit bestimmt werden ohne die Differenz zwischen aktuellem Rest und Divisor zu kennen. Wir setzen das Quotientenbit daher einfach auf 1. Dadurch kann der nächste Rest aber negativ werden. Wir lassen also negative Reste zu und setzen das Quotientenbit auf -1, falls der Rest negativ ist. Da wir es also sowieso mit negativen Zahlen zu tun haben, entwickeln wir das Prinzip gleich für Zahlen im 2-er-Komplement.

Falls also der aktuelle Rest im Bereich $[-2d, 2d-1]$ liegt, wobei d der Divisor ist, wird im negativen Fall d addiert und im positiven Fall (≥ 0) d subtrahiert. Daher liegt die Differenz im Bereich $[-d, d-1]$. Nach Hinzunahme des nächsten Dividendenbits an der rechten Seite liegt der neue Rest wiederum im Bereich $2 \cdot [-d, d-1] + \{0, 1\} = [-2d, 2d-1]$.

Man erhält am Schluss einen Divisionsrest im Bereich $[-d, d-1]$ und einen Quotienten der Form $\hat{q}_0 + \hat{q}_1 2^1 + \ldots + \hat{q}_{n-1} 2^{n-1}$ mit $\hat{q}_i \in \{+1, -1\}$. Um einen Quotienten der Form $q_0 + q_1 2^1 + \ldots - q_n 2^n$ (also im 2-er-Komplement) zu erhalten, formen wir folgendermaßen um:

$$\begin{aligned} & \hat{q}_0 + \hat{q}_1 2^1 + \ldots + \hat{q}_{n-1} 2^{n-1} \\ = \; & (\hat{q}_0 + 1) + (\hat{q}_1 + 1)2 + \ldots + (\hat{q_{n-1}} + 1)2^{n-1} - \underbrace{(1 + 2 + \ldots + 2^{n-1})}_{=2^n - 1} \\ = \; & 1 + \frac{\hat{q}_0 + 1}{2} 2^1 + \frac{\hat{q}_1 + 1}{2} 2^2 + \ldots - \left(1 - \frac{\hat{q}_{n-1} + 1}{2}\right) 2^n \,. \end{aligned}$$

Da $\frac{\hat{q}_i + 1}{2} \in \{0, 1\}$ ist, setzen wir $q_0 = 1$ (es gibt tatsächlich nur ungerade Quotienten) und $q_i = \frac{\hat{q}_{i-1} + 1}{2}$ für $i \geq 1$, wobei q_n noch invertiert werden muss. Das heißt, wir schreiben statt -1 einfach 0, schieben das Ganze um eine Stelle nach links, setzen die unterste Stelle auf 1 und invertieren die oberste.

Am Ende sollte noch der Rest r das gleiche Vorzeichen haben wie der Dividend. Daher muss eventuell noch d zum Rest addiert oder subtrahiert werden und der Quotient um eins verringert oder erhöht werden (respektive). Es kann aber auch passieren, dass $r = -d$ ist. Es sollte aber $|r| < |d|$ sein. Auch das muss noch korrigiert werden. Das macht die Sache sehr kompliziert. Auch die Overflow-Erkennung wird kompliziert. Daher verzichten wir hier einfach auf diese Dinge.

```
1  (q[0:n-1], r[0:n-1]) = NR_Div<n> (x[0:2*n-1], d[0:n-1])
2  {
3    r[0:2*n-1] = x[0:2*n-1];
4
5    for i = [n-1:0:-1]
6    {
7      q[i] = not (r[i+n]);
8      for j = [0:n-1] { xd[j] = xor (d[j], q[i]); }
9      (g, p, r[i:i+n-2]) = RCLA_Add<n-1> (r[i:i+n-2], xd[0:n-2], q[i]);
10     r[i+n-1] = xor (xor (r[i+n-1], xd[n-1]), or (g, and (p, q[i])));
11   }
12
13   for i = [n-1:1:-1] { q[i] = q[i-1]; }
14   q[0] = '1';
15 }
```

In Zeile 7 werden die Quotientenbits `q[i]` wie vereinbart gleich dem Vorzeichen des aktuellen Rests gesetzt. Danach muss je nach `q[i]` entweder `d` addiert oder `d` subtrahiert werden. Daher wird in Zeile 8 `d` mittels `xor` invertiert, falls `q[i]` gleich 1 ist. In Zeile 9–10 wird addiert/subtrahiert, wobei das Carry-In auf 1 gesetzt werden muss, wenn `q[i]` gleich 1 ist. Das höchste Bit wird hier gesondert behandelt, da es so schneller zur Verfügung steht. Das ist wichtig, da das Quotientenbit der nächsten Iteration nur von diesem Bit abhängt. In Zeile 13–14 wird die Transformation von $\hat{q}$ zu q vollzogen durch einen Shift nach links und Setzen von `q[0]` auf 1.

Die Komplexität ist hier natürlich ebenfalls $O(n \log n)$. Durch den Wegfall der Multiplexer sinken aber die Gatterzahl und die Delays ein wenig. Allerdings käme dann ja noch eine Korrekturoperation in der Komplexität einer Addition dazu.

n	Delay	Gates
4	32	124
8	82	568
16	196	2416
32	469	9952
64	1125	40384

4.4 SRT-Division

Die SRT-Division ist eine Verallgemeinerung der Non-Restoring Division. Sie ist nach Sweeney, Robertson und Tocher benannt, die sie Ende der 50er-Jahre unabhängig von einander erfunden haben. Die SRT-Division vereint folgende Prinzipien in sich:

- Der Divisor ist auf Werte beschränkt, die an führender Stelle (dh bei Wertigkeit 2^{n-1}) eine 1 haben, also auf Werte aus dem Bereich $2^{n-1} \leq d < 2^n$. Das ist zB bei Division von Gleitkommazahlen (zB IEEE) immer gegeben. Aber auch bei normalen ganzen Zahlen kann man in einem Vorbereitungsschritt sowohl Dividend als auch Divisor so weit nach links schieben, dass die Voraussetzung gegeben ist. Das Resultat ändert sich dadurch nicht.

- Als Quotientenziffern sind die Werte 1, 0 und -1 erlaubt.

- Die Auswahl der Quotientenziffern erfolgt rein aufgrund des aktuellen Restes. Ist dieser im Bereich $-2^{n-2} \leq d < 2^{n-2}$, wird 0 ausgewählt. Ist er darüber, $+1$ und der Divisor wird subtrahiert, ist er darunter, -1 und der Divisor wird addiert. Dadurch ist die Differenz im Bereich $-2^{n-2} \leq r' < 2^{n-2}$. Der aktuelle Rest im nächsten Schritt ist dann wie gewünscht im Bereich $-2^{n-1} \leq r < 2^{n-1}$. Zusammenfassend:

akt. Rest	Quotientenziffer $\hat{q}_i$	Aktion
01xxx	$+1$	Divisor subtrahieren
00xxx	0	–
11xxx	0	–
10xxx	-1	Divisor addieren

Daraus ergibt sich folgende Möglichkeit zur Verbesserung der Komplexität.

- Es wird ein Carry-Save-Addierer zur Differenzberechnung verwendet. Dh der aktuelle Rest besteht aus zwei Wörtern, eines für die Zähler-Summen und eines für die Zähler-Carrys. Der zu subtrahierende bzw. zu addierende Divisor kommt als drittes Wort hinzu und die drei Wörter werden in Carry-Save-Manier in konstanter Zeit wieder zu zwei Wörtern reduziert.

- Zur Auswahl der Quotientenziffer benötigt man nur ein paar führende Stellen der Summen- und Carry-Wörter. Da eine gewisse Unsicherheit besteht, da ein (unbekanntes) Carry-Bit bei exakter Addition der beiden Wörter die vorderen Bits noch verändern könnte, nehmen

wir ein Bit mehr, also 3 Bits. Durch eben diese Unsicherheit kann die Differenz aber etwas außerhalb von $-2^{n-2} \leq r' < 2^{n-2}$ liegen und daher auch der aktuelle Rest außerhalb von $-2^{n-1} \leq r < 2^{n-1}$. Dieser Fehler wird zwar durch die Redundanz der SRT-Division aufgefangen, aber wir brauchen trotzdem links noch ein Bit mehr, also jetzt 4 Bit. Wir nehmen also $3 + 1$ führende Bits der beiden Carry-Save-Wörter und berechnen daraus einen approximierten Rest. Daraus ergibt sich folgende Regel:

approx. Rest	Quotientenziffer $\hat{q}_i$	Aktion
0001xxx ... 0111xxx	$+1$	Divisor subtrahieren
0000xxx	0	–
1111xxx	0	–
1110xxx	0	–
1101xxx ... 1000xxx	-1	Divisor addieren

Die Ermittlung des approximierten Restes und die Ableitung der Quotientenziffern aus diesen Bits ist nun der aufwändigste Teil in jeder Iteration der SRT-Division.

```
(q[0:n-1], r[0:n]) = SRT_Div<n> (x[0:2*n-1], d[0:n-1])
{
  d[n] = '0';
  r[0:2*n-1] = x[0:2*n-1];  r[2*n] = x[2*n-1];
  for i = [0:2*n] { rc[i] = '0'; }

  for i = [n-1:0:-1]
  {
    (c, s[0]) = HalfAdd (r[i+n-2], rc[i+n-2]);
    (c, s[1]) = FullAdd (r[i+n-1], rc[i+n-1], c);
    (c, s[2]) = FullAdd (r[i+n], rc[i+n], c);
    s[3] = xor (xor (r[i+n+1], rc[i+n+1]), c);

    q0[i] = and (not (and (s[3], and (s[2], s[1]))),
                 or (or (s[3], s[2]), or (s[1], s[0])));
    q1[i] = and (s[3], q0[i]);

    for j = [0:n] { e[j] = and (xor (d[j], not (q1[i])), q0[i]); }
    rc[i] = and (not (q1[i]), q0[i]);
    for j = [i+n:i:-1]  { (rc[j+1], r[j]) = FullAdd (r[j], rc[j], e[j-i]); }
    rc[i] = '0';
  }

  r[1:n+1] = Add<n> (r[1:n], rc[1:n], '0');
  c = '0';
  for i = [0:n-1]
  { q[i] = xor (q0[i], c);
    c = or (q1[i], and (not (q0[i]), c));
  }
}
```

Der Algorithmus nimmt einen vorzeichenbehafteten Dividenden mit $2n$ Bit und einen positiven Divisor mit einer 1 an der höchsten Stelle (`d[n-1]`). Der Quotient kann negativ werden.

Auch der Rest ist vorzeichenbehaftet und braucht manchmal ein Bit mehr als üblich ($n + 1$ Bit).

In Zeile 9-12 wird aus den Carry-Save-Zwischensummen `r` und `rc` der approximierte Rest `s[0:3]` gebildet. In Zeile 14-16 wird daraus die Quotientenziffer gebildet. Dabei ist `(q1[i],q0[i])` als 2-stellige Zahl zu interpretieren, also 00 für 0, 01 für 1 und 11 für -1. In Zeile 18 wird der Summand ($-d$, 0, $+d$) erzeugt, der in Zeile 20 in der Carry-Save-Addition den neuen Rest in `r` und `rc` produziert. Für den negativen (invertierten) Summanden muss noch ein Carry-Bit eingeschoben werden (Zeile 19).

In Zeile 24 wird der letzte Schritt der Carry-Save-Addition durchgeführt, indem `r` und `rc` addiert werden, um den echten Rest zu berechnen. In Zeile 25-29 wird der Quotient q im binären 2er-Komplement aus $\hat{q}$ berechnet, dessen Ziffern ja die Werte $1, 0, -1$ annehmen können. Dabei kommt ein Carry-Bit mit der Wertigkeit -1 und folgende Wahrheitstabelle zum Einsatz:

$\hat{q}_i$	q1[i],q0[i]	c_{i-1}	c_i	q_i
1	01	0	0	1
1	01	1	0	0
0	00	0	0	0
0	00	1	1	1
-1	11	0	1	1
-1	11	1	1	0

Auch hier müsste am Ende noch eine Korrektur eingeschoben werden, damit der Rest das gleiche Vorzeichen hat wie der Dividend und betragsmäßig kleiner ist.

Sowohl die Berechnung der Quotientenziffern als auch die Carry-Save-Addition beanspruchen eine konstante Zeit pro Iteration (hängen nicht von n ab). Mit n Iterationen erhält man eine lineare Komplexität $O(n)$.

n	Delay	Gates
4	68	292
8	136	840
16	272	2704
32	544	9504
64	1088	35392

Dass die Delay-Werte dennoch nicht so berauschend sind, liegt an der suboptimalen Implementierung. Die Berechnung der Quotientenziffern (die die Iterationen dominiert) kann stark verbessert werden, indem man nicht zuerst die Summe der Carry-Save-Teilwörter (im Ripple-Carry-Verfahren!) bildet, sondern `q1` und `q0` direkt aus den Teilwörtern ableitet (Wahrheitstabelle, Schaltungsminimierung). Auch die Berechnung des endgültigen Restes und des Quotienten sollte nicht im Ripple-Carry-Verfahren geschehen.

4.5 Verwendung höherer Basen

Bei der Verwendung höherer Basen werden mehrere Quotientenbits in einem Schritt berechnet und dann Vielfache des Divisors $(0, d, 2d, 3d, \ldots)$ vom Rest subtrahiert. Bei der normalen Restoring Division läuft das auf mehrere Subtraktionen pro Iteration hinaus, wobei jene Differenz ausgewählt wird, die gerade noch ≥ 0 ist.

Bei der SRT-Division mit höherer Basis β können die Quotientenziffern die Werte $-\alpha, \ldots, +\alpha$ annehmen, wobei $\alpha \geq \frac{\beta}{2}$ gelten muss. Je größer α, desto größer die Redundanz im Ergebnis. Auch hier kann die Ermittlung der Quotientenziffern allein aufgrund von einigen führenden

Stellen des aktuellen Restes durchgeführt werden, allerdings unter Einbeziehung einiger Bits des Divisors.

Die große Wahlfreiheit für α, für die Quotientenziffern und nicht zuletzt für die Basis eröffnet viele Möglichkeiten zur Komplexitätsverbesserung. Es existieren hochoptimierte SRT-Dividierer mit Basis 4, 8 oder gar 16.

4.6 Division durch Multiplikation

Auch bei dieser Methode setzen wir voraus, dass der Divisor $\geq 2^{n-1}$ ist, also das oberste Bit 1 ist. Folgende Idee liegt der Methode zugrunde. Wir wollen den Quotienten $Q = \frac{X}{D}$ berechnen. Der Quotient bleibt unverändert, wenn wir zum Zähler und zum Nenner die gleiche Zahl R_0 multiplizieren. Das können wir sogar öfter machen und die Faktoren $R_0, \ldots, R_m$ multiplizieren. Wenn wir es nun schaffen, dass der Nenner gegen eins konvergiert, dann ist automatisch der Zähler gleich dem Quotienten.

$$Q = \frac{X}{D} = \frac{X \cdot R_0}{D \cdot R_0} = \frac{X \cdot R_0 \cdots R_m}{D \cdot R_0 \cdots R_m} \longrightarrow \frac{Q}{1}$$

Bei dieser Methode erhalten wir übrigens keinen Rest. Um diesen zu ermitteln wäre ein eigener Schritt notwendig ($X - Q \cdot D$).

Um den Zähler gegen 1 konvergieren zu lassen hilft uns folgende Überlegung. Der Divisor liegt ja zwischen $2^{n-1} \leq D < 2^n$. Es gibt also eine Zahl $0 < y \leq \frac{1}{2}$, so dass $D = 2^n(1-y)$ ist. Wir setzen nun $R_0 = 1 + y$. Dann ist

$$D \cdot R_0 = 2^n(1-y)(1+y) = 2^n(1-y^2)\,.$$

Der Zähler ist daher näher an 2^n. Wir wiederholen diesen Vorgang mit R_1 und lassen so den Nenner gegen 2^n gehen. Dementsprechend geht der Zähler gegen $Q \cdot 2^n$. Wir dividieren den Zähler also gleich am Anfang durch 2^n, dann kommt Q heraus.

Wie berechnen wir nun R am einfachsten aus D? Eingesetzt ergibt sich $2^n R = 2^{n+1} - D$. Das entspricht binär dem 2-er-Komplement mit einem vorangestellten Einser. Beispiel für $n = 4$: $32-10 = 100000_2 - 1010_2 = 10000_2 + 110110_2 = 10110_2 = 22$, wobei 0110_2 das 2-er-Komplement von 1010_2 ist. Wir brauchen also eine Funktion, die uns das 2-er-Komplement berechnet.

```
1 y[0:n-1] = TwosComplement<n> (x[0:n-1])
2 {
3   o[0:n-1] = OrKumul<n> (x[0:n-1]);
4   y[0] = x[0];
5   for i = [1:n-1] { y[i] = xor (x[i], o[i-1]); }
6 }
```

Beim 2-er-Komplement wird ein Bit an der Stelle i nur dann nicht invertiert, wenn alle Bits von 0 bis $i-1$ gleich 0 sind. Wir verwenden also die kumulierende Or-Funktion (Zeile 3) und invertieren jedes Bit mittels `xor`, wenn das kumulierte Or-Array and der Stelle $i-1$ gleich 1 ist. Die Funktion hat logarithmische Komplexität, da `OrKumul` logarithmische Komplexität hat.

Bevor wir nun die Division implementieren, stellen wir fest, dass wir dazu einen Multiplizierer brauchen, der verschieden lange Multiplikanden-Wörter akzeptiert. Auch brauchen wir oft nicht alle Produkt-Bits (nur die führenden sind wichtig). Hier ist so ein Multiplizierer:

```
1 z[0:nz-1] = PCS_Mul<nx,ny,nz> (x[0:nx-1], y[0:ny-1])
2 {
```

```
3   for i = [0:nz-2] { d[i] = 0; }
4
5   for iy = [0:ny-1]
6   { for ix = [0:nx-1]
7     { iz = ix + iy - nx - ny + nz;
8       if iz >= 0
9       { xy[d[iz],iz] = and (x[ix], y[iy]);
10        d[iz] = d[iz] + 1;
11      }
12    }
13  }
14
15  if (nx < ny) { nxy = nx; } else { nxy = ny; }
16  z[0:nz-1] = CS_Add_Gen<nz-1,1,d[0:nz-2],nxy> (xy[0:nxy-1,0:nz-2]);
17 }
```

Der Multiplizierer entspricht dem Carry-Save-Multiplizierer. Der Unterschied ist, dass für jedes Teilprodukt nur jene Bits generiert werden, die eine Wertigkeit der führenden `nz` Ergebnisbits haben. Dh die hinteren Bits werden schon vor dem Addieren abgeschnitten, was zu Fehlern führen kann.

Hier ist nun die straight-forward-Implementierung der obigen Idee.

```
1  q[0:n-1] = Msf_Div<n> (x[0:2*n-1], d[0:n-1])
2  {
3    q[-n:n-1] = x[0:2*n-1];
4    k = 1;
5    while k < n
6    { k = k * 2;
7      dc[0:n-1] = TwosComplement<n> (d[0:n-1]);  dc[n] = '1';
8      d[-1:n] = PCS_Mul<n,n+1,n+2> (d[0:n-1], dc[0:n]);
9      q[-n:n] = PCS_Mul<2*n,n+1,2*n+1> (q[-n:n-1], dc[0:n]);
10   }
11 }
```

In Zeile 3 wird der Zähler durch 2^n dividiert. Dann wird in Zeile 7 das 2-er-Komplement des Nenners gebildet und ein Einser vorangestellt. Danach werden der Nenner und der Zähler mit diesem Wert multipliziert.

Es stellt sich natürlich die Frage, wie oft der Vorgang wiederholt werden muss, bis das Ergebnis genügend genau ist. Glücklicherweise konvergiert die Folge $D \cdot R_0 \cdots R_m$ sehr schnell (quadratisch). Das heißt, dass sich die Anzahl der führenden Einser im Nenner in jedem Schritt verdoppeln. Da am Anfang ein führender Einser gegeben ist (Voraussetzung), sind $\log_2 n$ Schritte notwendig. Daher die Schleife mit `k`.

Man beachte, dass das Ergebnis im letzten Bit Rundungsfehler enthalten kann. Bei Gleitkommazahlen ist das aber nicht so schlimm. Es sind – wie gesagt – $O(\log n)$ Schritte notwendig, um die Konvergenz zu erreichen. Jeder Schritt hat eine Komplexität von $O(\log n)$ (Carry-Save-Addition und 2-er-Komplement-Bildung). Die gesamte Komplexität ist also $O(\log^2 n)$. Die Gatteranzahlen sind aber exorbitant. Daher wird dieses Prinzip meist iterativ unter Verwendung der Multiplizier-Einheit in mehreren Taktzyklen angewendet.

n	Delay	Gates
4	36	546
8	73	2766
16	139	14112
32	229	66620
64	323	311562

An diesem Algorithmus gibt es nun einiges zu verbessern. Erstens kann man den Multiplikator verkürzen. Um k führende Einser in D zu produzieren, muss der Multiplikator nur $k+1$ Bit haben. Dabei stellt sich dann heraus, dass D manchmal $> 2^n$ wird. Das macht aber nichts. D konvergiert dann eben von oben her gegen 2^n. Allerdings muss man bei der 2-er-Komplement-Bildung auch das führende Bit $d[n]$ mit einbeziehen, da es nicht immer 0 ist.

Als zweite Verbesserung verzichten wir auf die volle Präzision des Zählers. Für das Ergebnis brauchen wir ja nur n Bit. Um allzu arge Rundungsfehler zu vermeiden, verwenden wir hier $n+2$ Bit. Die dritte Verbesserung ist, dass im letzten Schritt der Nenner nicht mehr berechnet werden muss (er ist sowieso gleich $11\ldots11$).

```
1  q[0:n-1] = M_Div<n> (x[0:2*n-1], d[0:n-1])
2  {
3    d[n] = '0';
4    q[-n:n-1] = x[0:2*n-1];
5    k = 1;
6    while k < n
7    { k = k * 2;
8      dc[n-k:n] = TwosComplement<k+1> (d[n-k:n]);
9      if (k < n) { d[-1:n+1] = PCS_Mul<n+1,k+1,n+3> (d[0:n], dc[n-k:n]); }
10     q[-2:n] = PCS_Mul<n+2,k+1,n+3> (q[-2:n-1], dc[n-k:n]);
11   }
12 }
```

Die Komplexität ist prinzipiell noch immer $O(\log^2 n)$. Jedoch spart man sich etwas Delay und sehr viele Gatter.

n	Delay	Gates
4	30	304
8	61	1147
16	107	4194
32	168	15682
64	242	59315

Eine weitere Verbesserung der Methode ist, den ersten Multiplikator so zu wählen, dass mehr als 2 führende Bits 1 werden. Dafür reicht aber nicht das 2-er-Komplement. Dazu muss eine lookup-table (LUT), zB als ROM implementiert, verwendet werden, die aufgrund einiger führender Bits des Divisors den passenden Multiplikator liefert.

Des Weiteren kann man die Tatsache ausnutzen, dass der Nenner im i-ten Schritt 2^i führende Einsen hat und der Multiplikator daher 2^i Nullen. Der Multiplizierer muss diese Teilprodukte daher gar nicht erst erzeugen. In Kombination mit den verkürzten Multiplikatoren stellt sich das Problem, dass der Nenner auch die Form $10\ldots0x\ldots x$ haben kann, mit 2^i Nullen, und der Multiplikator daher führende Einsen haben kann. In diesem Fall muss $01\ldots1x\ldots x$ durch $10\ldots0x\ldots x - 00\ldots10\ldots0$ ersetzt werden, also eine Addition und eine Subtraktion.

5 Wurzel

Das Wurzelziehen ist der Division sehr ähnlich. Es gibt zwei Hauptunterschiede:

- Es werden in jedem Schritt *zwei* neue Radikandenziffern dazugenommen. (Radikand ist die Zahl, deren Wurzel berechnet werden soll.)
- Der "Divisor" ist nicht konstant, sondern setzt sich aus den bisher gefundenen Wurzelziffern mal 2 und der neuen Wurzelziffer zusammen, dh jene, die gerade gefunden werden soll.

Zuerst ein Beispiel im Dezimalsystem. Es soll die Wurzel der Zahl 123456 ermittelt werden.

$$
\begin{array}{rcl}
1\,2\,3\,4\,5\,6 & & \\
\hline
1\,2 & \approx & 3 \cdot 3 \\
\;\;9 & & \\
\hline
\;\;3\,3\,4 & \approx & 6\,5 \cdot 5 \\
\;\;3\,2\,5 & & \\
\hline
\quad 9\,5\,6 & \approx & 70\,1 \cdot 1 \\
\quad 7\,0\,1 & & \\
\hline
\quad 2\,5\,5 & &
\end{array}
$$

Zuerst ermittelt man (so wie beim 1×1) die Wurzel der ersten beiden Ziffern 12, also $r_2 = 3$. Der Rest ist dann $12 - 3 \cdot 3 = 3$. Dann nimmt man zwei Radikandenziffern dazu und erhält 334. Jetzt muss man die Ziffer r_1 finden, sodass $334 \approx 6r_1 \cdot r_1$ ist. Dabei ist 6 das Doppelte der bisherigen Wurzelziffern, also $2 \cdot 3$. Wir finden $r_1 = 5$, da $65 \cdot 5 = 325 \approx 334$. Der neue Rest ist daher $334 - 325 = 9$. Jetzt suchen wir die Ziffer r_0, sodass $70r_0 \cdot r_0 \approx 956$. Dabei ist 70 das Doppelte der bisherigen Wurzelziffern, also $2 \cdot 35$. Wir finden $r_0 = 1$ und erhalten den Rest 255. Die Wurzel ist daher $r_2 r_1 r_0 = 351$. Tatsächlich ist $351 \cdot 351 + 255 = 123456$.

Mit binären Zahlen sieht das Ganze sogar etwas einfacher aus, da die potentielle neue Wurzelziffer nur 1 oder 0 sein kann und daher entweder die bisherige Wurzel (mit 01 angehängt, 0 wegen der Verdopplung, 1 für die neue Ziffer) subtrahiert wird oder eben gar nichts subtrahiert wird, wenn die neue Ziffer 0 ist. Hier ein Beispiel. Es wird die Wurzel von 177 berechnet. Es ergibt sich $177 = 13 \cdot 13 + 8$.

$$
\begin{array}{rcl}
1\,0\,1\,1\,0\,0\,0\,1 & & \\
\hline
1\,0 & \approx & 1 \cdot 1 \\
\;\;1 & & \\
\hline
\;\;1\,1\,1 & \approx & 10\,1 \cdot 1 \\
\;\;1\,0\,1 & & \\
\hline
\quad 1\,0\,0\,0 & \approx & 110\,0 \cdot 0 \\
\hline
\quad 1\,0\,0\,0\,0\,1 & \approx & 110\,0\,1 \cdot 1 \\
\quad\;\; 1\,1\,0\,0\,1 & & \\
\hline
\quad\;\; 1\,0\,0\,0 & &
\end{array}
$$

Man beachte, dass der Rest immer länger werden kann. Im i-ten Schritt kann er $i+2$ Bit haben (vor Subtraktion). Hier die Implementierung in der Art der Restoring Division.

```
1 r[0:n/2-1] = Sqrt<n> (x[0:n-1])
2 {
3   x[n] = '0';
4   k = n - 2;  l = 3;  nr[0] = '0';
```

```
5   for i = [n/2-1:0:-1]
6   {
7     (g, p, xx[k:k+l-1]) = RCLA_Add<l> (x[k:k+l-1], '1',nr[i+1:n/2-1],'11', '0');
8     r[i] = g;  nr[i] = not (r[i]);
9     x[k:k+l-1] = Mux1<l> (x[k:k+l-1], xx[k:k+l-1], r[i]);
10    l = l + 1;  k = k - 2;
11  }
12 }
```

In Zeile 7 wird der potentielle neue Rest `xx` gebildet, indem von `x` die aktuellen Wurzelbits subtrahiert werden. Dabei wird das Array `nr` addiert, das die invertierten Wurzelbits enthält. Aus dem angehängten 01 wird hier wegen der 2-er-Komplement-Bildung 11. Falls die Differenz ≥ 0 ist, wird das neue Wurzelbit auf 1 gesetzt (Zeile 8). In Zeile 9 wird in Abhängigkeit davon der alte Rest beibehalten oder der neue Rest eingefügt.

Die Komplexität ist in jedem Schritt logarithmisch, die Anzahl der Schritte ist $\frac{n}{2}$, also $O(n)$. Genauer ist die Gesamtkomplexität $O(\log 3 + \log 4 + \ldots + \log(\frac{n}{2} + 2)) = O(n \log n)$.

n	Delay	Gates
4	11	83
8	28	218
16	67	644
32	163	2120
64	393	7568

Die große Ähnlichkeit mit der Division ist offensichtlich. Daher kann man natürlich alle Optimierungsmethoden der Division anwenden: Non-Restoring, SRT-Division mit Carry-Save-Addition, höhere Basen. Wir belassen es bei dieser Aussicht und verzichten auf eine Implementierung.

6 Elementare Funktionen

Bei der Division durch Multiplikation haben wir den Zähler und den Nenner eines Bruchs so verändert, dass der Quotient konstant geblieben ist. Wir haben den Nenner geschickt gegen 1 gehen lassen und haben so im Zähler den Quotienten bekommen. Dieses Prinzip lässt sich auf andere Funktionen verallgemeinern. Wenn wir zB $f(x)$ berechnen wollen und angenommen $f(0) = 1$ ist, dann müssen wir nur Folgen $(a_0 = 1, a_1, \ldots, a_m)$ und $(b_0 = x, b_1, \ldots, b_m \approx 0)$ finden, so dass gilt:

$$f(x) = 1 \cdot f(x) = a_0 f(b_0) = a_1 f(b_1) = \ldots = a_m f(b_m) \approx a_m f(0) = a_m \,.$$

So erhalten wir in a_m eine Approximation von $f(x)$. Im Folgenden suchen wir solche Folgen für einige elementaren Funktionen.

6.1 Exponentialfunktion

Hier ist $f(x) = e^x$ und es gilt: $f(0) = e^0 = 1$. Wir müssen nun also eine Folge von a_i und b_i finden, sodass $a_i e^{b_i} = a_{i+1} e^{b_{i+1}}$ gilt. Das ist erfüllt, wenn $a_{i+1} = a_i \cdot u_i$ und $b_{i+1} = b_i - \ln u_i$, denn

$$a_{i+1} e^{b_{i+1}} = a_i \cdot u_i \cdot e^{b_i - \ln u_i} = a_i \cdot u_i \cdot e^{b_i} \cdot u_i^{-1} = a_i e^{b_i} \,.$$

Weiters soll $b_i \longrightarrow 0$ und die Multiplikation mit u_i möglichst einfach sein. Wir nehmen an, dass $0 \leq x < 1$ ist. Eine gute Lösung ist dann

$$u_i := \begin{cases} 1 + 2^{-i} & b_i \geq \ln(1 + 2^{-i}) \\ 1 & b_i < \ln(1 + 2^{-i}) \,. \end{cases}$$

Dadurch wird b_i immer kleiner und die Multiplikation mit $1+2^{-i}$ reduziert sich auf eine Addition $x(1 + 2^{-i}) = x + x \cdot 2^{-i}$. Die Werte für $\ln(1 + 2^{-i})$ müssen (für jedes i) im vorhinein berechnet werden und können zB in einem ROM abgelegt werden. Hier ist eine Straight-Forward-Implementierung.

```
1  y[-n+1:1] = Expsf<n> (x[-n:-1])
2  {
3    lntab[6:10,0:9] = '0000000001_0000000010_0000000100_0000001000_0000010000';
4    lntab[1:5 ,0:9] = '0000100000_0000111110_0001111001_0011100100_0110011111';
5    lntab[0   ,0:9] = '1011000110';
6    for i = [0:10] { for j = [0:9] { lntabi[i,j] = not(lntab[i,j]); } }
7
8    x[0] = '0';
9    for i = [-n:n+1] { y[i] = '0'; }  y[0] = '1';
10
11   for i = [0:n]
12   {
13     (g, p, xx[-n:-1]) = RCLA_Add<n> (x[-n:-1], lntabi[i,10-n:9], '1');
14     s = or (g, p);
15     x[-n:-1] = Mux1<n> (x[-n:-1], xx[-n:-1], s);
16
17     for j = [-n:1] { a[j] = and (y[j+i], s); }
18     (g, p, y[-n:1]) = RCLA_Add<n+2> (y[-n:1], a[-n:1], '0');
19   }
20 }
```

Die Funktion akzeptiert Fixpunktzahlen mit 10-Bit-Präzision (es muss $n = 10$ sein) ohne Vorkommastellen. Dh in `x[-n:-1]` ist die Zahl $0.x_{-1}x_{-2}\ldots x_{-n}$ enthalten. Das Ergebnis hat zwei Vorkommastellen, da e^x maximal $e^1 \approx 2.718$ werden kann. In Zeile 3-5 werden die im vorhinein berechneten Werte für $\ln u_i$ definiert. So ist zB $\ln u_0 = \ln(1 + 2^{-0}) = \ln 2 \approx 0.101100010$, diese Bits werden in `lntab[0,0:9]` gespeichert. Die Bits von $\ln u_{10}$ werden in `lntab[10,0:9]` gespeichert.

In Zeile 13 werden diese Werte der Reihe nach (Schleife `i`) von `x` subtrahiert, indem sie vorher in Zeile 6 invertiert wurden und dann addiert werden (mit Carry 1). In Zeile 7 wird festgestellt, ob die Differenz negativ ist (`s='0'`) oder ≥ 0 (`s='1'`). Im letzteren Fall wird `x` auf die Differenz gesetzt, ansonsten in Restoring-Manier auf den alten Wert (Zeile 8). Das entspricht dem Übergang $b_i \mapsto b_{i+1}$. Danach muss noch y mit $1 + 2^{-i}$ multipliziert werden, was $a_i \mapsto a_{i+1}$ entspricht. Das ist einfach eine Addition von y mit einer geshifteten Version von sich selbst und wird in Zeile 11 erledigt.

Die Konvergenz des Systems ist linear, es werden genau n Iterationen benötigt. Die Additionen sind von logarithmischer Komplexität, daher hat der Algorithmus die Gesamtkomplexität von $O(n \log n)$. Da für jedes n die Werte für $\ln u_i$ neu eingegeben werden müssten (für eine korrekte Rundung reicht einfaches Abschneiden nicht), gibt es hier nur Ergebnisse für $n = 10$:

n	Delay	Gates
10	159	2761

Dieser Algorithmus kann nun auf zwei Arten verbessert werden.

1. Es lässt sich zeigen (und beobachten), dass in der Iteration i durch die Subtraktion von $\ln u_i$ das Bit x_{-i} nullgesetzt wird. Falls dieses Bit schon 0 ist, muss auch nicht subtrahiert werden. Daher kann man die Restoring-Differenzbildung fallen lassen und einfach `s=x[-i]` setzen. Für die erste Iteration $i = 0$ gilt dies allerdings nicht.

2. Ab $i = \frac{n}{2}$ besteht $\ln u_i$ nur noch aus einer Eins an der Stelle $-i$. Die Subtraktion hat daher keinen Einfluss mehr auf die Entscheidung für folgende Subtraktionen. Es stehen also alle weiteren `s` fest und sind in `x[n/2:n]` ablesbar. Die restlichen Multiplikationen von y mit $1 + 2^{-i}$ bzw. 1 reduzieren sich daher auf eine Multiplikation mit $1.0 \ldots 0 x_{n/2} \ldots x_n$. Das kann auch mit einem Carry-Save-Addierer erledigt werden.

```
1  y[-n+1:1] = Exp<n> (x[-n:-1])
2  {
3    lntab[0:4,0:9]  = '0000111110_0001111001_0011100100_0110011111_1011000110';
4    for i = [0:4] { for j = [0:9] { lntabi[i,j] = not(lntab[i,j]); } }
5
6    x[0] = '0';
7    for i = [-n:n+1] { y[i] = '0'; }  y[0] = '1';
8
9    for i = [0:n/2-1]
10   {
11     if i = 0
12     { (g, p, xx[-n:-1]) = RCLA_Add<n> (x[-n:-1], lntabi[0,10-n:9], '1');
13       s = or (g, p);
14       x[-n:-1] = Mux1<n> (x[-n:-1], xx[-n:-1], s);
15     }
16     else
17     { s = x[-i];
18       for j = [-n:-1] { b[j] = and (lntabi[i,10+j], s); }
19       (g, p, x[-n:-1]) = RCLA_Add<n> (x[-n:-1], b[-n:-1], s);
20     }
21
22     for j = [-n:1] { a[j] = and (y[j+i], s); }
23     (g, p, y[-n:1]) = RCLA_Add<n+2> (y[-n:1], a[-n:1], '0');
24   }
25
26   for i = [-n:1] { d[i] = 1; yy[0,i] = y[i]; }
27   for i = [n/2:n]
28   { for j = [1-i:-n:-1]
29     { yy[d[j],j] = and (y[j+i], x[-i]); d[j] = d[j] + 1; }
30   }
31
32   y[-n:1] = CS_Add_Gen<n+2,0,d[-n:1],n-n/2+2> (yy[0:n-n/2+1,-n:1]);
33 }
```

In Zeile 11-15 wird für die erste Iteration noch in Restoring-Manier verfahren. Für die Iterationen $i = 1 \ldots \frac{n}{2} - 1$ wird in Zeile 17-20 einfach `s` mittels `s=x[-i]` ermittelt und in Zeile 19 $\ln u_i$ subtrahiert, wenn `s=1` ist.

Die Iterationen $i = \frac{n}{2} \ldots n$ werden in einem Streich erledigt, indem in Zeile 26-30 die zu y zu addierenden Teilwörter $y \cdot 2^{-i} \cdot x_i$ im zweidimensionalen Array `yy` gesammelt und in Zeile 32 mit `CS_Add_Gen` addiert werden.

Die Komplexität ist zwar noch immer $O(n \log n)$, die Delays und Gatterzahlen werden aber deutlich reduziert.

n	Delay	Gates
10	91	1363

6.2 Logarithmus

Das Prinzip hier ist ähnlich. Wir starten mit $x_0 = x$ und $y_0 = 0$. Nun versuchen wir die Folgen x_i und y_i so hinzukriegen, dass gilt:

$$x = xe^0 = x_0 e^{y_0} = \ldots = x_m e^{y_m} \approx 1 \cdot e^y \,.$$

Um das zu erreichen, setzen wir $x_i \mapsto x_{i+1} = x_i \cdot u_i$ und $y_i \mapsto y_{i+1} = y_i - \ln u_i$. Daher bleibt der Ausdruck $x_i e^{y_i}$ konstant:

$$x_{i+1} e^{y_{i+1}} = x_i u_i e^{y_i - \ln u_i} = x_i u_i u_i^{-1} e^{y_i} = x_i e^{y_i} \,.$$

Für die u_i wählen wir:

$$u_i := \begin{cases} 1 + 2^{-i} & x_i(1 + 2^{-i}) < 1 \\ 1 & x_i(1 + 2^{-i}) \geq 1 \,. \end{cases}$$

Dadurch geht x_i immer näher gegen 1 und für die $\ln u_i$ können wir die selbe Tabelle wie bei der Exponentialfunktion verwenden. Als Wertebereich für x wählen wir $0.5 \leq x < 1$. Das ist bei Gleitkommazahlen immer erfüllt. Der Logarithmus bewegt sich daher im Bereich $-0.696 \leq y < 0$. Wir berechnen daher lieber $-\ln x$, dann haben wir positive Zahlen. Das ist kein Problem, da Gleitkommazahlen meist ohnehin nicht im 2-er-Komplement sonderen einfach mit einem Bit für das Vorzeichen gespeichert sind. Das Ergebnis müsste in diesem Fall natürlich noch um den Exponenten von x korrigiert werden.

```
1  y[-n:-1] = Logsf<n> (x[-n:-1])
2  {
3    lntab[6:10,0:9] = '0000000001_0000000010_0000000100_0000001000_0000010000';
4    lntab[1:5,0:9]  = '0000100000_0000111110_0001111001_0011100100_0110011111';
5    lntab[0,0:9]    = '1011000110';
6
7    for i = [0:n] { x[i] = '0'; }
8    for i = [-n:-1] { y[i] = '0'; }
9
10   for i = [1:n-1]
11   {
12     (g, p, xx[-n:-1]) = RCLA_Add<n> (x[-n:-1], x[i-n:i-1], '0');
13     s = not (g);
14     x[-n:-1] = Mux1<n> (x[-n:-1], xx[-n:-1], s);
15     (g, p, y[-n:-1]) = RCLA_Add<n> (y[-n:-1], And<n> (lntab[i,10-n:9], s), '0');
16   }
17 }
```

In Zeile 8 wird y mit 0 initialisiert. In Zeile 12 wird probeweise x mit $1 + 2^{-i}$ multipliziert. Ist das Ergebnis nicht ≥ 1 (Zeile 13), wird es nach x kopiert (Zeile 14). In diesem Fall wird auch noch $\ln(1 + 2^{-1})$ zu y addiert (Zeile 15).

Auch hier ist die Komplexität $O(n \log n)$, da im schlechtesten Fall n Schritte notwendig sind, um Konvergenz zu erreichen.

n	Delay	Gates
10	134	1989

Natürlich gibt es für diesen Algorithmus Verbesserungen analog zur Exponentialfunktion.

6.3 Trigonometrische Funktionen

Auch für Kosinus und Sinus existiert ein solches Schema. Im Prinzip ist es die Erweiterung auf die Exponentialfunktion mit imaginärem Argument.

$$e^{ix} = \cos x + i \sin x$$

Wir suchen nun einfach wieder u_j, so dass

$$1 \cdot e^{ix} = y_0 \cdot e^{ix_0} = \ldots = y_j \cdot e^{ix_j} = y_j u_j \cdot e^{i(x_j - \ln u_j)} = y_{j+1} \cdot e^{ix_{j+1}} = \ldots = y_m \cdot e^{ix_m} \approx y \cdot e^0$$

Die y_j sind dabei komplexe Werte und wir schreiben: $y_j =: c_j + is_j$. Wir setzen

$$u_j := \begin{cases} 1 + i2^{-j} & x_j \geq 0 \\ 1 - i2^{-j} & x_j < 0\,. \end{cases}$$

Damit soll x_j gegen 0 gehen. Dieses Kriterium ist ähnlich wie bei der Non-Restoring Division. Es ergibt sich:

$$c_{j+1} = c_j \mp 2^{-j} s_j, \qquad s_{j+1} = s_j \pm 2^{-j} c_j\,.$$

Wegen $a + ib = \sqrt{a^2 + b^2} e^{i \tan^{-1}(b/a)}$ gilt $(1 \pm i2^{-j}) = \sqrt{1 + 2^{-2j}} e^{i \tan^{-1}(\pm 2^{-j})}$ und wir setzen $x_{j+1} := x_j - \tan^{-1}(\pm 2^{-j})$. Dadurch ergibt sich:

$$y_{j+1} e^{x_{j+1}} = y_j (1 \pm i2^{-j}) e^{ix_j - i \tan^{-1}(\pm 2^{-j})} = \sqrt{1 + 2^{-2j}}\, y_j e^{ix_j}\,.$$

Das entspricht nur fast der oben geforderten Gleichheit des Ausdrucks nach dem Übergang $j \mapsto j + 1$. Wir erhalten am Ende also für y_m nicht $\approx e^{ix_0}$ sondern:

$$y_m e^{ix_m} = y_0 e^{ix_0} \prod_{j=0}^{m-1} \sqrt{1 + 2^{-2j}} =: y_0 e^{ix_0} K\,.$$

Das Produkt K ist ein konstanter Wert, den wir im voraus berechnen können. Setzen wir am Anfang $y_0 = \frac{1}{K}$, dann liefert der Prozess für $y_m = c_m + is_m$ die gewünschte Approximation von e^{ix}, wenn $x_0 = x$ und $x_m \approx 0$. Der Kosinus von x steht dann also in c_m und der Sinus in s_m.

Für den Wertebereich von x wählen wir $0 \leq x < \frac{\pi}{2} \approx 1.57$. Kosinus und Sinus sind dann positiv und liegen im Bereich $0 \leq c, s \leq 1$. Da $\tan^{-1}(-\alpha) = -\tan^{-1}(\alpha)$, brauchen wir nur die vorberechneten Werte von $\tan^{-1}(2^{-j})$ für $j = 0 \ldots n$. In jedem Schritt j müssen wir also folgende Zuweisungen machen:

$$x_{j+1} = x_k \mp \tan^{-1}(2^{-j}),\ c_{j+1} = c_j \mp 2^{-j} s_j,\ s_{j+1} = s_j \pm 2^{-j} c_j,$$

wobei $x_0 = 0$ und $c_0 + is_0 = \frac{1}{K} + i0$. Die Entscheidung, ob subtrahiert oder addiert wird, ergibt sich aus dem Vorzeichen von x_j, so dass x_j möglichst gegen 0 geht. Die Konvergenz ist linear.

```
1  (c[-n:-1], s[-n:-1]) = CosSin<n> (x[-n:0])
2  {
3    at[6:10,0:9] = '0000000001_0000000010_0000000100_0000001000_0000010000';
4    at[1:5,0:9]  = '0000100000_0001000000_0001111111_0011111011_0111011011';
5    at[0,0:9]    = '1100100100';
6
7    x[1] = '0';
8    c[-10:10] = '00000000000_1001101110';
9    s[-10:10] = '00000000000_0000000000';
10
11   for i = [0:n]
12   {
13     q = not (x[1]);
14     for j = [-n:-1] { a[j] = xor (at[i,10+j], q); }
15     (g, p, x[-n:1]) = RCLA_Add<n+2> (x[-n:1], q,q,a[-10:-1], q);
16     for j = [-n:-1] { a[j] = xor (s[j+i], q); }
17     (g, p, cc[-n:-1]) = RCLA_Add<n> (c[-n:-1], a[-n:-1], q);
18     for j = [-n:-1] { a[j] = xor (c[j+i], not (q)); }
19     (g, p, s[-n:-1]) = RCLA_Add<n> (s[-n:-1], a[-n:-1], not (q));
20     c[-n:-1] = cc[-n:-1];
21   }
22 }
```

Da $x > 1$ werden kann, brauchen wir eine Vorkommastelle, daher `x[-n:0]`. In `c` und `s` werden der Kosinus und der Sinus ausgegeben. In Zeile 3-5 wird die Tabelle für die $\tan^{-1}(2^{-j})$ definiert. In Zeile 8-9 wird `c` mit $\frac{1}{K}$ und `s` mit 0 initialisiert. In Zeile 13 wird das Vorzeichen von x_j ermittelt (`q`). Danach wird nach obigem Schema x_{j+1}, c_{j+1} und s_{j+1} in Abhängigkeit von `q` durch Subtraktion bzw. Addition berechnet.

Die Komplexität ist auch hier wieder $O(n \log n)$.

n	Delay	Gates
10	175	3465

Wie an der $\tan^{-1}$-Tabelle zu erkennen ist, kann auch hier der Algorithmus wieder auf ähnliche Weise wie bei `Exp` und `Log` verbessert werden.

Index

Inhaltsverzeichnis