Institut für Parallele und Verteilte Systeme
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3070

Hardware Implementierung eines parallelen Entropie Koders

Maximiliano Keller

Studiengang:	Informatik
Prüfer:	Prof. Dr.-Ing. Sven Simon
Betreuer:	Dipl.-Inf. Simeon Wahl
begonnen am:	26. Juli 2010
beendet am:	25. Januar 2011
CR-Klassifikation:	E4, B.2.1, F.2.0

Inhaltsverzeichnis

Abbildungsverzeichnis

Verzeichnis der Listings

Abstract

Das Arithmetische Kodieren hat den Vorteil, nahe der Entropie komprimieren zu können. Das Verfahren besteht aus zwei Differenzengleichungen, deren Berechnung sequenziell ist. Teilrechnungen können jedoch zusammengefasst und parallel berechnet werden.

In dieser Diplomarbeit wurde die Parallelisierung des Arithmetischen Kodierens in Hinblick auf eine Hardware-Implementierung untersucht. Lösungsvorschläge wurden analysiert und gegenübergestellt sowie ihre Gemeinsamkeiten herausgearbeitet.

Ein wesentlich einfacherer Algorithmus für die Skalierung wurde entwickelt und eine Formel für die mindestens zu wählende Bit Breite erarbeitet. Abschließend wurde eine VHDL Lösung implementiert.

Abstract

The advantage of Arithmetic Coding is a compression rate close to the Entropy. The method consists of two difference equations. Their calculation is of a sequential nature. Parts of them can be combined and calculated in parallel.

In this diploma theses the parallelization of Arithmetic Coding has been examined with focus on a hardware implementation. Different approaches for such parallelization were analyzed and compared. The similarities have been worked out.

A simpler scaling algorithm has been presented and a formula for the minimum bit width was developed. A VHDL solution has been implemented.

1 Einleitung

In der Informationsverarbeitung müssen große Datenmengen gespeichert und verwaltet werden. Suchmaschinenhersteller wie beispielsweise Google™, die Inhalte aus Internet-Seiten für ihre Suchanfragen speichern müssen, erreichen Index-Größen von einer Trillion URLs (1.000.000.000.000)[1]. Soziale Netzwerke wie Facebook™ zählen inzwischen 500 Millionen registrierte Nutzer mit über 50 Billionen[2] Uploads, deren Daten bewältigt und verwaltet werden müssen. Es ist klar, dass man bei diesen Mengen nach möglichst effizienten Speichermethoden sucht. Hier spielt die Komprimierung eine große Rolle. Nicht nur der Platzbedarf würde sinken und damit Kosten sparen, sondern selbst geringe Komprimierungsraten würden ausreichen, um erhebliche Geschwindigkeitssteigerungen zu erreichen.

Wann immer Daten übertragen werden, kann man den Durchsatz steigern, indem man für den selben Informtionsgehalt weniger Bytes überträgt. Das gilt nicht nur für Datenübertragung von der Festplatte zum Computer, sondern ist allgemein gültig. Es findet unter anderem Anwendung in der Telekommunikationsbranche. Hier werden große Anstrengungen unternommen, die Gespräche möglichst stark zu komprimieren, um die Bandbreite besser zu nutzen.

Bei Anwendungen wie beispielsweise einer Hochgeschwindigkeitskamera, die sehr große Mengen an Daten produziert, ist man zusätzlich mit dem Problem konfrontiert, ob die Daten in der geforderten Zeit überhaupt komprimiert werden können. Man kann davon ausgehen, dass eine stärkere Komprimierung mehr Rechenzeit in Anspruch nehmen wird als eine geringe.

Das Arithmetische Kodieren verspricht Komprimierungsraten nahe der Entropie. Die streng sequenzielle Abarbeitung der Daten macht den Algorithmus jedoch sehr langsam. In diesem Zusammenhang wäre es interessant zu wissen, ob man dieses Verfahren durch eine Parallelisierung in Hardware beschleunigen kann, um es zum Beispiel in Mobiltelefonen oder Hochgeschwindigkeitskameras einsetzen zu können. Dieser Frage werden wir in dieser Arbeit nachgehen.

Gliederung

Es erschien mir sinnvoll, erst das Prinzip des Arithmetischen Kodierens zu erklären und dann dieses mit der Parallelisierung zu vertiefen. Zuerst wird die allgemeine Idee erklärt.

[1] http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html
[2] http://www.usatoday.com/tech/news/ 2010-07-21-facebook-hits-500-million-users_N.htm

Sind diese Grundlagen geschafft, kann man sich weiter Gedanken über die Parallelisierung machen. Schließlich fließen die gewonnenen Erkenntnisse dann in die konkrete Umsetzung ein.

Kapitel 2 – Arithmetisches Kodieren: Hier wird das Prinzip des sequenziellen Arithmetischen Kodierens beschrieben und ein Algorithmus entwickelt.

Kapitel 3 – Parallelisierung: Ansätze zur Parallelisierung des Algorithmus werden hier vorgestellt. Diese werden im Hinblick auf eine konkrete Hardware Implementierung hin analysiert und gegenübergestellt. Die daraus gewonnenen Erkenntnisse bilden den Kern dieser Diplomarbeit.

Kapitel 4 – Implementierung: Das Ergebnis der Analyse endet mit einer konkreten Implementierung in VHDL. In diesem Kapitel werden die einzelnen Module des Kodes im Detail erklärt.

2 Arithmetisches Kodieren

2.1 Grundlagen

2.1.1 Motivation

Die Geschichte des Arithmetischen Kodierens ist eigentlich ziemlich interessant. Shannon hat schon 1948 [Sha48] in einem Paper ein Verfahren erwähnt, das die Verteilungsfunktion für eine Kodierung vorschlägt. Später wurde es unter dem Namen Shannon-Fano Kode bekannt. Peter Elias, der auch Fanos Vorlesung über Informationstheorie hörte, entwickelte eine rekursive Implementierung des Problems. Diese wurde jedoch nie veröffentlicht. Jelinek entwickelte diese Idee in einem Anhang in seinem Buch, das 1968 erschien, weiter.

Der wirkliche Durchbruch gelang aber erst 1976 durch unabhängige Erkenntnisse von Pasco [Pas76] und Rissanen [Ris76]. Hier wurde erstmalig das Problem der endlichen Genauigkeit gelöst. Schließlich erschienen aufgrund dieser Erkenntnisse Paper mit praktischen Implementierungen. Eine der bekanntesten davon ist die Referenzimplementierung von Witten, Neal und Cleary [WNC87]. Wie wir sehen werden, vergingen wieder einige Jahre bis eine parallele Implementierung erstmals von Jiang und Jones [JJ94] 1994 in einem Paper vorgeschlagen wurde.

Was diese Geschichte so interessant macht, ist die Tatsache, dass das Prinzip zwar relativ früh bekannt wurde, es jedoch sehr lange gedauert hat, bis man die Probleme, die dieses Verfahren mit sich bringt, in den Griff bekommen hat. Eine andere Betrachtungsweise wäre, dass dieses Verfahren eher uninteressant ist und von keiner praktischen Bedeutung. Dann würden sich auch weniger Menschen damit befassen und sich so die lange Geschichte erklären. Das ist nicht der Fall. Ganz im Gegenteil! Das Arithmetische Kodieren hat einige Vorteile gegenüber dem sehr umfassend behandelten Huffman Coding.

Welche Vorteile hat das Arithmetische Kodieren gegenüber dem Huffman Coding?

Dazu benötigen wir erstmal ein paar Definitionen. Ich halte mich dabei an die Nomenklatur und die Beispiele von K. Sayood [Say96].

Sei $P(A)$ die Wahrscheinlichkeit, dass ein Ereignis A eintritt, dann ist die Selbstinformation des Ereignisses A

$$i(A) = \log_b \frac{1}{P(A)} = -\log_b P(A)$$

Diese Definition kann man sich so vorstellen, dass ein Ereignis welches selten auftritt mehr Informationsgehalt hat als eines welches häufig vorkommt. Das entspricht auch der intuitiven Auffassung von Informationsgehalt.

Beispiel 1. Seien K und Z die möglichen Ergebnisse eines Münzwurfs. Dann gilt

$$P(K) = P(Z) = \frac{1}{2}$$

und damit auch

$$i(K) = i(Z) = -\log_b(\frac{1}{2}) = 1 \text{ bit}$$

Würde man die Münze unfair gestalten, so dass

$$P(K) = \frac{1}{8} \text{ und } P(Z) = \frac{7}{8}$$

dann ergibt sich daraus

$$i(K) = 3 \text{ bits, und } i(Z) = 0.193 \text{ bits}$$

$\Diamond$

Den Informationsgehalt eines Textes nennt man die Entropie. Sie ist die Summe der einzelnen Informationsgehalte multipliziert mit der Wahrscheinlichkeit, mit der sie auftreten:

$$E = \sum P(A_i) i(A_i) = -\sum P(A_i) \log_b P(A_i)$$

Die Entropie stellt die untere Grenze einer Kompaktierung dar. In der Regel wird man diese Grenze nur in Ausnahmefällen erreichen.

Das genügt an dieser Stelle, um in einem Beispiel die Probleme des Huffman Codings zu zeigen.

Beispiel 2. Gegeben sei ein Alphabet $A = \{a_1, a_2, a_3\}$ mit den Wahrscheinlichkeiten $P(a_1) = 0.95$, $P(a_2) = 0.02$ und $P(a_1) = 0.03$. Die Entropie ist also

$$-(0.95 \log_b(0.95) + 0.03 \log_b(0.03) + 0.02 \log_b(0.02) = 0.3349$$

Einen Huffman Code für diese Quelle zeigt Tabelle 2.1

Die durchschnittliche Kodelänge für diesen Kode ist $1 \cdot 0.95 + 2 \cdot 0.02 + 2 \cdot 0.03 = 1.05$ Bits/Symbol und damit 213% der Länge der Entropie. Das bedeutet, dass es mehr als doppelt soviel Bits braucht wie die Entropie. Selbst wenn man die Symbole in Zweiergruppen aufteilen würde, erhält man noch eine durchschnittliche Kodelänge, die 72% über der Entropie liegt.

Um noch bessere Kompaktierung zu erreichen, kann man noch größere Gruppen zusammenfassen. Bei einer Gruppe von 8 Symbolen erreicht man akzeptable Werte. Das dazugehörige Alphabet steigt aber zu einer Anzahl von $3^8 = 6561$ Symbolen an. Hier sieht man, dass in diesem Fall das Huffman Coding nicht mehr effizient ist.

$\Diamond$

Symbol	Kodewort
a_1	0
a_2	11
a_3	10

Tabelle 2.1: Huffman Code für einstellige Sybole

Symbol	Wahrscheinlichkeit	Kodewort
a_1a_1	0.9025	0
a_1a_2	0.0190	111
a_1a_3	0.0285	100
a_2a_1	0.0190	1101
a_2a_2	0.0004	110011
a_2a_3	0.0006	110001
a_3a_1	0.0285	101
a_3a_2	0.0006	110010
a_3a_3	0.0009	110000

Tabelle 2.2: Huffman Code für zweistellige Symbole

An diesem Beispiel wird klar, wo die Probleme beim Huffman Code liegen: Man erreicht bessere Kompaktierung, wenn man Kodewörter für Gruppen von Symbolen generiert. Doch für lange Symbolgruppen muss man dann sehr viele Kodes erzeugen. Die zu erzeugenden Kodes steigen exponentiell an.

Man kann das umgehen, indem man ein Kodewort für eine bestimmte Symbolsequenz generiert. Genau das ist die Idee hinter dem Arithmetischen Kodieren.

Beim Arithmetischen Kodieren wird einer Symbolsequenz $S = x_1x_2x_3\ldots$ genau eine Zahl (oder Kode) zugewiesen. Die Länge der Sequenz ist $|S| = n$.

2.1.2 Prinzip

Um einer Symbolsequenz oder auch einer Buchstabensequenz eine eindeutige Zahl zuzuweisen bedienen wir uns der Verteilungsfunktion einer diskreten Zufallsvariablen. Sei also ein Ereignis $a_i \in A$ einer Zufallsvariablen

$$X(a_i) = i$$

zugewiesen, wobei $A = a_1, a_2, a_3 \ldots a_m$ ein Alphabet ist. Damit gilt die Wahrscheinlichkeit, dass X den Wert i annimmt:

$$P(X = i) = p(a_i)$$

Das Zuweisen einer Wahrscheinlichkeit ist das zugrundeliegende Modell P für die Quelle. Wir werden später sehen, dass diese Zuweisung großen Einfluss auf die Kompressionsrate haben wird.

Die Verteilungsfunktion ist dann als

$$F_x(i) = \sum_{k=1}^{i} P(X = k) = \sum_{k=1}^{i} p(a_i) \tag{2.1}$$

definiert. Es gelte außerdem, dass sämtliche Ereignisse a_i disjunkt sind und

$$\cap a_i = A$$

Nun aber endlich zu der grundlegenden Idee: Wir wollen erreichen, einer Sequenz einen eindeutigen Wert zuzuweisen. Dafür geht man folgendermaßen vor:

Mit $|A| = m$ ist $F_x(m) = 1$. Deshalb betrachten wir ein Intervall von $[0, 1)$. Wir nutzen die Tatsache, dass in diesem Intervall unendlich viele irrationale Zahlen existieren und teilen dieses Intervall in Teilintervalle aus der diskreten Verteilungsfunktion $F_x(i)$. Wir weisen jedem a_i das Teilintervall:

$$[F_x(i-1), F_x(i)) \quad \text{für} \quad i = 1 \ldots m \tag{2.2}$$

zu.

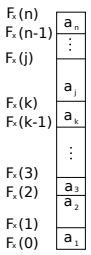

Abbildung 2.1: Intervalleinteilung (obere Grenzen $= F_x(i)$)

Wird jetzt zum Beispiel als erstes das Symbol a_k gelesen, weisen wir das Intervall $[F_x(k-1), F_x(k))$ unserem Kode zu. Wir merken uns hierzu die untere Grenze $F_x(k-1)$ und die obere, offene Grenze $F_x(k)$. Dann teilen wir dieses Intervall wieder in m Teilintervalle ein.

Damit das Verhältnis erhalten bleibt, müssen die neuen Grenzen durch die Länge des neuen Intervalls geteilt werden. Lesen wir beispielsweise als nächstes das Symbol a_j ein, wird dem Kode das Intervall

$$\left[\frac{F_x(k-1)+F_x(j-1)}{F_x(k)-F_x(k-1)}, \frac{F_x(k-1)+F_x(j)}{F_x(k)-F_x(k-1)}\right) \tag{2.3}$$

zugewiesen.

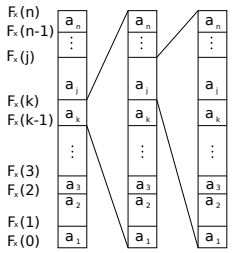

Abbildung 2.2: Teilintervalleinteilung (obere Grenzen $= F_x(i)$)

Beispiel 3. Gegeben sei ein Alphabet $A = \{a_1, a_2, a_3\}$ mit $P(a_1) = 0.7$, $P(a_2) = 0.1$ und $P(a_3) = 0.2$. Die Gleichung (2.1) liefert $F_x(1) = 0.7$, $F_x(2) = 0.8$ und $F_x(3) = 1.0$. Falls nun das Symbol a_1 kodiert werden soll, liegt das Intervall zwischen $[0.0, 0.7)$, für a_2 zwischen $[0.7, 0.8)$ und entsprechend für a_3 zwischen $[0.8, 1.0)$.

Angenommen wir lesen erst das Symbol a_1. Dann liegt unser Kode zwischen den Werten $[0.0, 0.7)$. Dieses Intervall wird gemäß der Gleichung (2.3) in die Teilintervalle $[0.0, 0.49)$, $[0.49, 0.56)$ und $[0.56, 0.7)$ zerlegt. Würde als nächstes wieder das Symbol a_1 gelesen, dann läge der Kode im Intervall $[0.0, 0.49)$, oder bei a_2 im Intervall $[0.49, 0.56)$ und bei a_3 $[0.56, 0.7)$.

Und wieder würde man das Intervall aufteilen und für das nächste Symbol das entsprechende Teilintervall wählen. $\Diamond$

Das Zerlegen in Teilintervalle und Auswählen des Teilintervalls wiederholt man so lange, bis die Quelle gelesen ist. Das letzte (Teil-)Intervall[1] bildet das Ergebnis der Kodierung. Man muss sich aber nicht die untere und obere Grenze merken. Es reicht, eine beliebige Zahl aus dem Intervall zu nehmen. Das ergibt sich aus der Tatsache, dass die Teilintervalle voneinander disjunkt sind und sobald man irgendeine Zahl aus dem Intervall gewählt hat, es kein anderes Intervall gibt, welches diese Zahl beinhaltet. Damit ist eindeutig festgelegt, um welches Intervall es sich handelt. Aus praktischen Gründen wählt man gerne die untere Grenze, wie wir später sehen werden.

[1]Das erste Intervall wird in Teilintervalle zerlegt. Das dann ausgesuchte Teilintervall wird nun zum Intervall

Um den mathematischen Weg der Kodegenerierung zu erläutern, wählen wir hier die Mitte des Intervalls und definieren dafür folgende Rechenregel

$$\begin{aligned}\bar{T}_x(a_i) &= \sum_{k=1}^{i-1} P(X=k) + \frac{1}{2}P(x=i) \\ \bar{T}_x(a_i) &= F_x(i-1) + \frac{1}{2}P(x=i) \end{aligned} \tag{2.4}$$

Das $\bar{T}$ steht hier für das englische Wort «tag». Jetzt können wir jedem Symbol a_i einen Kode zuweisen.

Beispiel 4. Sei die Augenzahl eines sechsseitigen, fairen Würfels den Ereignissen $\{1,2,3,4,5,6\}$ zugewiesen. Dann gilt

$$P(X=i) = \frac{1}{6} \quad \text{für} \quad i = 1,2,\dots 6$$

Mit der Gleichung (2.4) erhält man

$$\begin{aligned}\bar{T}_x(1) &= P(X=0) + \frac{1}{2}P(X=1) = 0 + \frac{1}{2}\cdot\frac{1}{6} = \frac{1}{12} = 0.08\overline{33} \\ \bar{T}_x(2) &= P(X=1) + \frac{1}{2}P(X=2) = \frac{1}{6} + \frac{1}{2}\cdot\frac{1}{6} = 0.25 \\ \bar{T}_x(3) &= \sum_{k=1}^{2} P(X=k) + \frac{1}{2}P(X=3) = 0.41\overline{66} \\ &\vdots\end{aligned}$$

Und so kann man auf einfache Weise für alle Ereignisse einen Kode zuweisen. $\Diamond$

Das Prinzip des Kodezuweisens kann man auf eine Sequenz von Ereignissen erweitern. Hierzu wird zunächst eine feste Ordnung der Symbole oder Ereignisse definiert:

$$\bar{T}_x^{(m)}(x_i) = \sum_{y<x_i} P(y) + \frac{1}{2}P(x_i) \tag{2.5}$$

Dabei bedeutet $y < x$, dass das Ereignis y dem Symbol x vorangeht und das hochgestellte m ist die Länge des Alphabets.

Beispiel 5. Wir benutzen wieder einen fairen Würfel und berechnen den Kode, der sich für das Ereignis 1 3, also dass man zuerst eine 1, und im Anschluss eine 3 würfelt, ergibt.

$$\begin{aligned}\bar{T}_x(13) &= P(X=11) + P(X=12) + \frac{1}{2}P(X=13) \\ &= 1/36 + 136 + 1/2(1/36) \\ &= 5/72\end{aligned}$$

In diesem Beispiel ist zu sehen, dass zur Berechnung der Sequenz der Länge n die Kodes aller vorangehenden Sequenzen der Länge $n-1$ benötigt werden. Das wäre aber genauso aufwendig als wenn man für das Huffman Coding die Wahrscheinlichkeiten aller möglichen Symbole der Länge n berechnen würde. Im nächsten Beispiel werden wir aber sehen, dass dieses nicht notwendig ist.

Im Folgenden werden wir die mathematische Vorgehensweise zum Erstellen eines Kodes für eine Sequenz kennenlernen.

Beispiel 6. Wir benutzen weiterhin den fairen, sechsseitigen Würfel und wollen jetzt eine obere und untere Grenze des Intervalls für einen Kode für das Ereignis 3 2 2 berechnen. Sind die Grenzen bekannt, kann man wie schon beschrieben eine beliebige Zahl aus diesem Intervall benutzen, um daraus einen Kode zu generieren.

Das Ereigniss besteht darin, dass zuerst eine 3 dann eine 2 und schließlich wieder eine 2 gewürfelt wird. Wir bezeichnen die obere Grenze mit der Sequenzlänge n mit H_n und die untere Grenze mit L_n.

Wir betrachten das Ereignis 3, gehen gemäß Gleichung (2.2) vor und setzen $i = 3$. Die Länge der Sequenz ist $n = 1$. Die Ereignisse sind laut Gleichung (2.5) in der Reihenfolge $\{1, 2, 3, 4, 5, 6\}$ geordnet. Damit ist das Ereignis $i - 1 = 2$. Wir setzen ein:

$$H_1 = F_x(3), \quad L_1 = F_x(2)$$

Als nächstes lesen wir die 2. Die Sequenz ist $x = 32$. Die neuen Grenzen sind

$$H_2 = F_x(32), \quad L_2 = F_x(31)$$

Die Berechnung dieser Werte geht folgendermaßen:

$$\begin{aligned} F_x(32) =& P(X = 11) + P(X = 12) + P(X = 13) \dots P(X = 16) + \\ & P(X = 21) + P(X = 22) + P(X = 23) \dots P(X = 26) + \\ & P(X = 31) + P(X = 32) \end{aligned}$$

Wenn i sämtliche Ereignisse m beinhaltet — also in unserem Beispiel 1 bis 6 — dann gilt

$$\sum_{i=1}^{m} P(X = ki) = \sum_{i=1}^{m} P(X_1 = k, X_2 = i) = P(X_1 = k)$$

wobei $X = X_1 X_2$. Wir können also schreiben

$$\begin{aligned} F_x(32) &= P(X_1 = 1) + P(X_2 = 2) + P(X = 31) + P(X = 32) \\ &= F_x(2) + P(X = 31) + P(X = 32) \end{aligned}$$

Weil die einzelnen Würfe unabhängige Ereignisse sind, gilt

$$P(X = 31) = P(X = 3)P(X = 1)$$

und

$$P(X = 32) = P(X = 3)P(X = 2)$$

Damit ergibt sich

$$\begin{aligned} P(X = 31) + P(X = 32) &= P(X_1 = 3)(P(X_2 = 1) + P(X_2 = 2)) \\ &= P(X_1 = 3)F_x(2) \end{aligned}$$

Weil

$$P(X_1 = i) = F_x(i) - F_x(i - 1)$$

also in unserem Beispiel

$$P(X_1 = 3) = F_x(3) - F_x(2)$$

können wir schreiben

$$P(X = 31) + P(X = 32) = (F_x(3) - F_x(2)) \cdot F_x(2)$$

oder

$$F_x(32) = F_x(2) + (F_x(3) - F_x(2)) \cdot F_x(2)$$

oder auch

$$H_2 = L_1 + (H_1 - L_1) \cdot F_x(2)$$

Auf gleiche Weise können wir zeigen, dass für die untere Grenze L_2 gilt

$$F_x(31) = F_x(2) + (F_x(3) - F_x(2)) \cdot F_x(1)$$

oder

$$L_2 = L_1 + (H_1 - L_1)F_x(1)$$

Das dritte Symbol in diesem Beispiel ist wieder eine 2 und damit

$$H_3 = F_x(322), \quad L_3 = F_x(321)$$

Die gleiche Rechnung ergibt damit

$$\begin{aligned} F_x(321) &= F_x(31) + (F_x(32) - F_x(31)) \cdot F_x(1) \\ F_x(322) &= F_x(31) + (F_x(32) - F_x(31)) \cdot F_x(2) \end{aligned}$$

oder

$$\begin{aligned} L_3 &= L_2 + (H_2 - L_2) \cdot F_x(1) \\ H_3 &= L_2 + (H_2 - L_2) \cdot F_x(2) \end{aligned}$$

$\Diamond$

Die allgemeine Vorschrift für die Intervallbildung der Sequenz $X = x_1x_2x_3 \ldots x_n$ sind die Differenzengleichungen

$$\boxed{\begin{aligned} L_i &= L_{i-1} + (H_{i-1} - L_{i-1}) \cdot F_x(i-1) \\ H_i &= L_{i-1} + (H_{i-1} - L_{i-1}) \cdot F_x(i) \end{aligned}} \tag{2.6}$$
$$\tag{2.7}$$

Für die Berechnung der Grenzen mussten keine vereinigten Wahrscheinlichkeiten berechnet werden. Es reicht also, die Verteilungsfunktion über das Alphabet zu kennen. Diese wird wie schon erwähnt durch das Modell gegeben. Für den Kode nimmt man nun eine Zahl aus dem Intervall. Wir wählen wieder die Hälfte

$$\bar{T}_x(X) = \frac{H_n - L_n}{2}$$

falls $X = x_1x_2x_3 \ldots x_n$ ist.

2.2 Kodierung als reelle Zahl

Wir gehen jetzt etwas konkreter vor und wollen die Zeichenfolge `SWISS␣MISS` kodieren. Dieses Beispiel ist aus [Salo8] entnommen. Die fünf vorkommenden Symbole können in einer beliebigen Reihenfolge in eine Tabelle für die Wahrscheinlichkeiten gespeichert werden. Die Funktion CUM_COUNT stehend für «cummulative count» wird mit

$$\texttt{cum\_count}_{a_i} = \sum_{k=1}^{i} |a_i|$$

definiert.

Wie schon eingangs erwähnt, entspricht diese Zuweisung dem Modell des Koders. Das Modell für unser Beispiel ist in Tabelle 2.3 zusammengefasst.

Symbol	Häufigkeit	Wahrscheinlichkeit p_i	$F_x(i-1)$	$F_x(i)$	Intervall	`cum_count`
$a_1 = \sqcup$	1	1/10=0.1	0.0	0.1	$[0.0, 0.1)$	1
$a_2 = \texttt{M}$	1	1/10=0.1	0.1	0.2	$[0.1, 0.2)$	2
$a_3 = \texttt{I}$	2	2/10=0.2	0.2	0.4	$[0.2, 0.4)$	4
$a_4 = \texttt{W}$	1	1/10=0.1	0.4	0.5	$[0.4, 0.5)$	5
$a_5 = \texttt{S}$	5	5/10=0.5	0.5	1.0	$[0.5, 1.0)$	10

Tabelle 2.3: Modell für Zeichenfolge `SWISS␣MISS`

Wir benutzen die Gleichungen (2.6) und (2.7), um sequentiell H und L zu errechnen. Am Anfang muss H mit 1 und L mit 0 initialisiert werden. Das bedeutet, am Anfang geht das

Intervall über die volle Breite. Erst nach Einlesen des ersten Symbols wird dieses Intervall eingeschränkt.

Als nächstes haben wir das Symbol $x_1 = \texttt{S}$ gelesen. Die neuen Grenzen errechnen sich mit

$$L_1 = 0 + (1 - 0) \cdot F_x(4) = 0 + 0.5 = 0.5$$
$$H_1 = 0 + (1 - 0) \cdot F_x(5) = 0 + 1.0 = 1.0$$

Das neue Intervall nach Einlesen von S ist $[0.5, 1.0)$. Wir fahren fort und lesen $x_2 = \texttt{W}$

$$L_2 = 0.5 + (1.0 - 0.5) \cdot F_x(3) = 0.5 + 0.5 \cdot 0.4 = 0.70$$
$$H_2 = 0.5 + (1.0 - 0.5) \cdot F_x(4) = 0.5 + 0.5 \cdot 0.5 = 0.75$$

Das Intervall verkleinert sich weiter auf $[0.70, 0.75)$. Wir lesen $x_3 = \texttt{I}$

$$L_3 = 0.70 + (0.75 - 0.70) \cdot F_x(2) = 0.70 + 0.05 \cdot 0.2 = 0.71$$
$$H_3 = 0.70 + (0.75 - 0.70) \cdot F_x(3) = 0.70 + 0.05 \cdot 0.4 = 0.72$$

und so weiter bis das letzte Symbol x_n gelesen ist. Wir können gemäß Gleichung (2.4) für den Kode die Mitte des Intervalls nehmen, oder einfach die untere Grenze. Der Kode ist in dem Fall 0.71753375.

2.3 Dekodierung als reelle Zahl

Die Dekodierung läuft analog zur Kodierung. Wir starten damit, dass wir L den Wert 0 und H den Wert 1 zuweisen. Nach der Dekodierung des ersten Symbols erhalten wir

$$L_1 = 0 + (1 - 0) \cdot F_x(x_1 - 1) = F_x(x_1 - 1)$$
$$H_1 = 0 + (1 - 0) \cdot F_x(x_1) = F_x(x_1)$$

Mit anderen Worten heißt das, dass wir dasjenige x_1 suchen, welches im Intervall $[F_x(x_1 - 1), F_x(x_1))$ liegt. Der Kode von 0.71753375 liegt im Intervall von $\texttt{S} = [0.5, 1.0)$. Also ist das gesuchte $x_1 = \texttt{S}$.

Wir wiederholen die Vorgehensweise und suchen x_2.

$$L_2 = 0.5 + (1 - 0.5) \cdot F_x(x_2 - 1) = 0.5 + 0.5 \cdot F_x(x_2 - 1)$$
$$H_2 = 0.5 + (1 - 0.5) \cdot F_x(x_2) = 0.5 + 0.5 \cdot F_x(x_2)$$

Jetzt wählen wir $x_2 = \texttt{W}$, denn

$$L_2 = 0.5 + 0.5 \cdot F_x(3) = 0.7 \quad \text{und} \quad H_2 = 0.5 + 0.5 \cdot F_x(4) = 0.75$$

Und das Intervall beinhaltet als einziges 0.71753375. Doch die Rechenregel lässt sich vereinfachen. Wir haben beim Kodieren zu L immer ein Teilintervall hinzuaddiert. Beim Dekodieren können wir entsprechend dieses wieder abziehen, die Werte anpassen und erhalten so etwas direkter das gesuchte x_i. Wir ziehen L_1 vom Kode ab $0.71753375 - 0.5 = 0.21753375$. Jetzt wird dieser Kode durch die Intervalllänge von S geteilt, um es auf den ursprünglichen Wert zu bringen $0.21753375/0.5 = 0.4350675$. Jetzt sieht man direkt, dass dieser Wert im Intervall $\texttt{W} = [0.4, 0.5)$ liegt. Diesen errechneten Wert (0.4350675) bezeichnen wir mit RANGE . Die weitere Dekodierung ist in Tabelle 2.5 zusammengefasst.

Symbol		Berechnung von H und L
S	L	$0.0 + (1.0 - 0.0) \cdot 0.5 = 0.5$
	H	$0.0 + (1.0 - 0.0) \cdot 1.0 = 1.0$
W	L	$0.5 + (1.0 - 0.5) \cdot 0.4 = 0.70$
	H	$0.5 + (1.0 - 0.5) \cdot 0.5 = 0.75$
I	L	$0.7 + (0.75 - 0.7) \cdot 0.2 = 0.71$
	H	$0.7 + (0.75 - 0.7) \cdot 0.4 = 0.72$
S	L	$0.71 + (0.72 - 0.71) \cdot 0.5 = 0.715$
	H	$0.71 + (0.72 - 0.71) \cdot 1.0 = 0.72$
S	L	$0.715 + (0.72 - 0.715) \cdot 0.5 = 0.7175$
	H	$0.715 + (0.72 - 0.715) \cdot 1.0 = 0.72$
␣	L	$0.7175 + (0.72 - 0.7175) \cdot 0.0 = 0.7175$
	H	$0.7175 + (0.72 - 0.7175) \cdot 0.1 = 0.71775$
M	L	$0.7175 + (0.71775 - 0.7175) \cdot 0.1 = 0.717525$
	H	$0.7175 + (0.71775 - 0.7175) \cdot 0.2 = 0.717550$
I	L	$0.717525 + (0.71755 - 0.717525) \cdot 0.2 = 0.717530$
	H	$0.717525 + (0.71755 - 0.717525) \cdot 0.4 = 0.717535$
S	L	$0.71753 + (0.717535 - 0.71753) \cdot 0.5 = 0.7175325$
	H	$0.71753 + (0.717535 - 0.71753) \cdot 1.0 = 0.717535$
S	L	$0.7175325 + (0.717535 - 0.7175325) \cdot 0.5 = 0.71753375$
	H	$0.7175325 + (0.717535 - 0.7175325) \cdot 1.0 = 0.717535$

Tabelle 2.4: Kodierung von SWISS␣MISS

2.4 Beweis der Eindeutigkeit

Sei $\bar{T}_x(x)$ eine Nummer im Intervall $[0, 1)$. Einen binären Kode für diese Nummer können wir erhalten, indem wir die binäre Darstellung nehmen und auf $l(x) = \lceil \log_b(\frac{1}{p(x)}) \rceil + 1$ Bits beschränken.

Wir erinnern uns, dass $\bar{T}_x(x)$ ein Kode für die Sequenz S ist. Um zu beweisen, dass $\lfloor \bar{T}_x(x) \rfloor_{l(x)}$ eindeutig ist, müssen wir lediglich zeigen, dass es im Intervall $[F_x(x-1), F_x(x))$ liegt.

Durch das Runden von $\lfloor \bar{T}_x(x) \rfloor_{l(x)}$ gilt $\lfloor \bar{T}_x(x) \rfloor_{l(x)} \leq \bar{T}_x(x)$. Damit ist

$$0 \leq \bar{T}_x(x) - \lfloor \bar{T}_x(x) \rfloor_{l(x)} < \frac{1}{2^{l(x)}} \tag{2.8}$$

i	Symbol	Kode - L_i = /($H_i - L_i$)	= range
1	S	0.71753375-0.5 = 0.21753375	/0.5 = 0.4350675
2	W	0.4350675-0.4 = 0.0350675	/0.4 = 0.350675
3	I	0.350675-0.2 = 0.150675	/0.2 = 0.753375
4	S	0.753375-0.5 = 0.253375	/0.5 = 0.50675
5	S	0.50675-0.5 = 0.00675	/0.5 = 0.0135
6	␣	0.0135-0.0 = 0.0135	/0.1 = 0.135
7	M	0.135-0.1 = 0.035	/0.1 = 0.35
8	I	0.35-0.2 = 0.15	/0.2 = 0.75
9	S	0.75-0.5 = 0.25	/0.5 = 0.5
10	S	0.5-0.5 = 0.0	/0.5 = 0.0

Tabelle 2.5: Dekodierung von SWISS␣MISS

Weil $\bar{T}_x(x)$ die Hälfte vom Intervall $[F_x(x-1), F_x(x))$ ist, gilt

$$\begin{aligned} \bar{T}_x(x) &< F_x(x) \\ \lfloor \bar{T}_x(x) \rfloor_{l(x)} &< \bar{T}_x(x) < F_x(x) \\ \lfloor \bar{T}_x(x) \rfloor_{l(x)} &< F_x(x) \end{aligned}$$

Jetzt müssen wir noch zeigen, dass $\lfloor \bar{T}_x(x) \rfloor_{l(x)} \geq F_x(x-1)$. Mit

$$\begin{aligned} \frac{1}{2^{l(x)}} &= \frac{1}{2^{\lceil \log_b(\frac{1}{p(x)}) \rceil + 1}} \\ &< \frac{1}{2^{\log_b(\frac{1}{p(x)}) + 1}} \\ &= \frac{1}{2\frac{1}{p(x)}} \\ &= \frac{p(x)}{2} \end{aligned}$$

Aus Gleichung (2.4) wissen wir

$$\frac{p(x)}{2} = \bar{T}_x(x) - F_x(x-1)$$

und damit auch

$$\bar{T}_x(x) - F_x(x-1) > \frac{1}{2^{l(x)}} \tag{2.9}$$

Kombinieren wir Gleichung (2.8) und (2.9) erhalten wir

$$\lfloor \bar{T}_x(x) \rfloor_{l(x)} > F_x(x-1)$$

Damit ist $\lfloor \bar{T}_x(x) \rfloor_{l(x)}$ eine eindeutige Repräsentierung von $\bar{T}_x(x)$.

Um jetzt zu zeigen, dass dieser Kode auch eindeutig dekodierbar ist, zeigen wir dass es ein Präfix Kode ist. Das heisst, kein Kode ist Präfix eines anderen Kodes. Weil ein Präfix Kode immer eindeutig dekodierbar ist, reicht es zu zeigen, dass $\lfloor \bar{T}_x(x) \rfloor_{l(x)}$ ein Präfix Kode ist.

Falls x und y zwei verschiedene Sequenzen sind, wissen wir, dass $\lfloor \bar{T}_x(x) \rfloor_{l(x)}$ und $\lfloor \bar{T}_x(y) \rfloor_{l(y)}$ in zwei unterschiedlichen Intervallen $[F_x(x-1), F_x(x))$ und $[F_x(y-1), F_x(y))$ liegen. Falls wir also zeigen können, dass für jede Sequenz x das Intervall $[\lfloor \bar{T}_x(x) \rfloor_{l(x)}, \lfloor \bar{T}_x(x) \rfloor_{l(x)} + 2^{l(x)})$ ganz in $[F_x(x-1), F_x(x))$ liegt, kann x kein Präfix für einen anderen Kode sein.

Wir wissen bereits, dass $\lfloor \bar{T}_x(x) \rfloor_{l(x)} > F_x(x-1)$. Was wir noch zeigen müssen ist

$$F_x(x) - \lfloor \bar{T}_x(x) \rfloor_{l(x)} > 2^{l(x)}$$

und das gilt, weil:

$$\begin{aligned} F_x(x) - \lfloor \bar{T}_x(x) \rfloor_{l(x)} &> F_x(x) - \bar{T}_x(x)_{l(x)} \\ &= \frac{p(x)}{2} \\ &> \frac{1}{2^{l(x)}} \end{aligned}$$

2.5 Effizienz

Wir haben gezeigt, dass die Anzahl $l(x)$ der benötigten Bits um $F_x(x)$ eindeutig zu dekodieren

$$l(x) = \lceil \log_b(\frac{1}{p(x)}) \rceil + 1$$

ist. $l(x)$ ist die Zahl der benötigten Bits für die gesamte Sequenz x. Für die Sequenz der Länge $|x| = m$ gilt demnach

$$\begin{aligned} l_{A^m} &= \sum p(x) l(x) \\ &= \sum p(x) \left[\lceil \log_b(\frac{1}{p(x)}) \rceil + 1 \right] \\ &< \sum p(x) \left[\log_b(\frac{1}{p(x)}) + 1 + 1 \right] \\ &= -\sum p(x) \log_b p(x) + 2 \sum p(x) \\ &= E(x^m) + 2 \end{aligned}$$

Weil die durchschnittliche Länge immer größer als die Entropie ist, können wir für $l(x)$ die Grenzen

$$E(x^m) \leq l_{A^m} < E(x^m) + 2$$

festlegen. Die durchschnittliche Länge pro Symbol l_A ist $\frac{l_{A^m}}{m}$. Durch Einsetzen gilt damit

$$\frac{E(x^m)}{m} \leq l_A < \frac{E(x^m)}{m} + \frac{2}{m}$$

und mit $E(x^m) = mE(x)$ erhalten wir

$$E(x) \leq l_A < E(x) + \frac{2}{m}$$

Desto länger also die Länge m der Sequenz, desto näher kommen wir an die Entropie.

2.6 Kodierung als begrenzte Festkommazahl

Je länger die Quelle, desto länger der Kode und damit umso höher die Präzision. Die derzeitigen Rechner sind sehr ineffizient, wenn es um Rechnungen beliebiger Genauigkeit geht. Je länger die Quelle, desto länger der Kode und damit umso höher die Präzision. Man stelle sich eine Quelle von 1 MByte vor. Selbst bei einer Komprimierung auf die Hälfte wäre das Teilen einer 500 kByte großen Zahl sehr komplex und aufwendig. Das war Jahrzehnte lang das Hindernis für eine praktische Verwendung des Arithmetischen Kodierens. Pasco und Rissanen haben unabhängig voneinander das Problem gelöst, indem sie einen Algorithmus entwarfen, mit dem man mit einer begrenzten Festkommazahl dennoch einen Kode generieren kann. Die Vorgehensweise, die hier vorgestellt wird, lehnt sich an die von Witten [WNC87] an.

2.6.1 Abbildung auf Ganze Zahlen

Für das Arithmetische Kodieren verwendet man am besten Ganzzahl-Variablen (integer Variable) statt Fließkomma-Variablen (floating point Variable), weil in der Fließkommaarithmetik Genauigkeit verloren geht[2]. Um möglichst den ganzen Bereich der Ganzzahl-Variablen zu nutzen, bilden wir die 0 auf die 0 ab und die 1 auf die $0.\overline{9}$, denn $0.\overline{9} = 1$. Diesen Bereich kann man so erweitern, dass keine Dezimalstelle mehr notwendig wird. Das ergibt für eine Ganzzahl-Variable die höchstmögliche darstellbare Zahl.

Beispiel 7. Eine integer Variable der Länge 4 kann im Dezimalsystem die Werte $0000 \ldots 9999$ annehmen. 9999 ist der höchste Wert. In diesem Fall würde man also $L = 0000$ und $H = 9999$ setzen, um alle Möglichkeiten dieser Variablen zu nutzen.

Eine Variable mit der Bitbreite 4 kann im Dualsystem die Werte $0000_\text{b} \ldots 1111_\text{b}$ annehmen. Wobei hier 1111_b der höchste Wert ist. Analog wäre hier $L = 0000_\text{b}$ und $H = 1111_\text{b}$. $\Diamond$

[2]Eine Ganzzahl-Variable kann mehr Werte annehmen als eine Fließkomma-Variable derselben Bit Breite

Das tiefgestellte «b» bedeutet, dass die Zahl im Dualsystem geschrieben ist. In den Fällen, an denen es nicht eindeutig ist in welchem System wir uns befinden, benutzen wir «d» für dezimal oder «h» für hexadezimal.

Daraus ergibt sich die interessante Frage nach der Auswirkung, die das Abbilden von unendlich vielen Zahlen auf eine endlich viele Menge auf den Algorithmus hat.

Wir definieren

$$r_i = H_i - L_i \tag{2.10}$$

als die Intervalllänge `range` .

Wenn wir wie im Beispiel 7 $H_0 = 1111$ setzen und damit das Intervall $[L_0, H_0)$ berechnen, kommt als Ergebnis

$$r_0 = H_0 - L_0 = 1111 - 0000 = 1111$$

heraus. Das entspricht aber nicht dem vollen Intervall. Dieser ist nämlich 1.0_d und H_0 wurde auf $0.\overline{9}_\mathrm{d}$ abgebildet. Um dieser Tatsache nachzukommen, muss, falls mittels H die `range` berechnet wird, noch 1_b hinzuaddiert werden.

Das hat aber eine Auswirkung auf die zu wählende Bit Breite. Für dasselbe Beispiel würde bei einer Bit Breite von 4 Bits $H_0 = 1111$ bei Addition von 1 ein Überlauf stattfinden. Deshalb muss man bei der Wahl der Bit Breite 1 Bit Puffer hinzufügen.

Für die Berechnung von `range` schreiben wir

$$\texttt{range}_i = H_i - L_i + 1$$

Wir wissen, dass $F_x(i)$ die kumulative Verteilungsfunktion für a_i ist. Also folgt daraus

$$F_x(i) = \frac{\texttt{high\_count}_i}{\texttt{total}} \tag{2.11}$$

Weiter gilt

$$F_x(i-1) = \frac{\texttt{high\_count}_{i-1}}{\texttt{total}} = \frac{\texttt{low\_count}_i}{\texttt{total}}$$

Wir setzen

$$\texttt{high\_count}_i = \texttt{cum\_count}_i \tag{2.12}$$

und

$$\texttt{step}_i = \texttt{range}_i / \texttt{total}$$

Setzen wir die Gleichungen (2.10), (2.11), und (2.12) in Gleichung (2.7) ein, ergibt das

$$\begin{aligned} L_i &= L_{i-1} + \texttt{range}_{i-1} \cdot \frac{\texttt{low\_count}_i}{\texttt{total}} \\ &= L_{i-1} + \frac{\texttt{range}_{i-1} \cdot \texttt{low\_count}_i}{\texttt{total}} \\ &= L_{i-1} + \frac{\texttt{range}_{i-1} \cdot \texttt{cum\_count}_{i-1}}{\texttt{total}} \\ &= L_{i-1} + \texttt{step}_{i-1} \cdot \texttt{cum\_count}_{i-1} \end{aligned} \tag{2.13}$$

Und für H_i

$$H_i = L_{i-1} + \texttt{step}_{i-1} \cdot \texttt{cum\_count}_i - 1 \tag{2.14}$$

Man beachte hier die -1 bei H_i. Wie schon erwähnt wird H um 1 verringert, um die nach oben offene Intervallgrenze darzustellen.

Weiterhin werden wir später sehen, dass es eine wichtige Rolle spielt, wann man durch `total` teilt. Nun können wir uns dem Problem der endlichen Präzision widmen.

2.6.2 Skalierung bei Überlauf

Wenn man sich die Tabelle 2.4 anschaut, dann fällt auf, dass sobald eine der höherwertigen Ziffern bei L und H gleich sind, sich diese nicht mehr ändern. Die Erklärung ist, dass sich ein Intervall, sobald er einen Wertebereich eingenommen hat, diesen nicht mehr verlassen kann. Bei jedem Schritt wird ein Intervall in Teilintervalle geteilt, die alle innerhalb des (Ursprungs-)Intervalls liegen. Wenn also eine Stelle von links bei H und L gleich sind, braucht man diese für die weitere Berechnung nicht und kann diese Ziffer als Kode herausschieben. Wir nennen das hier – eher willkürlich – den Überlauf [3]. Im Dezimalsystem ist das dann der Fall, wenn das Intervall um eine Zehnerpotenz kleiner geworden ist als der ursprüngliche Intervall; hier also ein Zehntel so groß wie vorher ($10^{-1} = 0.1$). Sei B die Basis des verwendeten Systems, dann ist die Skalierung bei Erreichen von sc durchführbar. Die Gleichung dafür ist

$$sc = B^{-1}$$

Im Dualen System entspricht eine Stelle einer Zweierpotenz. Hier findet der Überlauf bei der Hälfte ($2^{-1} = 0.5$) des Ursprungs-Intervalls statt. Ist das höchstwertigste Bit von L und H gleich, dann ist das Intervall nur noch halb so groß wie ursprünglich. Das höchstwertigste Bit kann als Kode nach links heraus geschoben werden. Für L wird von rechts eine 0 und für H eine 1 nach geschoben. Das höchstwertige Bit ist dann bei L und H gleich, wenn beide Grenzen entweder in der unteren Hälfte oder beide in der oberen Hälfte des Intervalls liegen.

[3] vielleicht angelehnt an die Tatsache, dass der Wert nach links, also höherwertig, überläuft

Jetzt können wir konkreter werden und zu einem Beispiel übergehen, welches später auch in Hardware realisiert werden wird. Dieses Beispiel soll den Anspruch haben, möglichst klein zu sein und dennoch sämtliche Fälle abzudecken. Dieses Beispiel ist aus [BCK02] entnommen.

Beispiel 8. Ein Alphabet mit $A = \{a, b, c, d, e\}$ sei gegeben. Wir wollen die Sequenz $S = abccedac$ kodieren. Dann ist $|S| = \texttt{total} = 8$.

Das Modell ergibt dann Tabelle 2.6

Symbol	`tot_count`	`low_count`	`high_count`
$a_1 = \mathrm{a}$	2	0	2
$a_2 = \mathrm{b}$	1	2	3
$a_3 = \mathrm{c}$	3	3	6
$a_4 = \mathrm{d}$	1	6	7
$a_5 = \mathrm{e}$	1	7	8

Tabelle 2.6: Modell für Zeichenfolge «abccedac»

Als Bit Breite wählen wir 8 Bits und können somit die unteren 7 Bits für die Kodierung nutzen. Das achte Bit ist nur für den Überlauf. Für uns gilt also das siebte Bit als das oberste. Zur besseren Lesbarkeit verwenden wir das Hexadezimalsystem und geben es andernfalls explizit an. L und H werden nun mit

$$L_0 = 00_h = 00000000_b$$
$$H_0 = 7F_h = 01111111_b$$

initialisiert. Wir lesen das erste Symbol a ein und erhalten mit den Formeln (2.13) und (2.14)

$$\texttt{range}_0 = 7F - 00 + 1 = 80$$
$$\texttt{step}_0 = 80/8 = 10$$
$$L_1 = 00 + \texttt{step}_0 \cdot \texttt{cum\_count}_0 = 00 + 10 \cdot 0 = 00$$
$$H_1 = 00 + \texttt{step}_0 \cdot \texttt{cum\_count}_1 - 1 = 00 + 10 \cdot 2 - 1 = 20 - 1 = 1F$$

Wir sehen, dass die oberen zwei Bits von H und L gleich sind $(00)_b$. Das sind also die ersten zwei Kode Bits. Wir schieben diese raus und erhalten für L und H

$$L_1 = 00$$
$$H_1 = 3F$$

Wir nennen diesen Vorgang die SKALIERUNG oder auch NORMALISIERUNG von L und H. Der Algorithmus in C dafür ist in Listing 2.1 präsentiert. Wir schieben L und H und zwei bits nach links und fügen jeweils 0 bzw. 1 nach.

$$L_1 = 00$$
$$H_1 = 7F$$

Jetzt lesen wir b ein und erhalten mit den Formeln (2.13) und (2.14)

$$\texttt{range}_1 = 7F - 00 + 1 = 80$$
$$\texttt{step}_1 = 80/8 = 10$$
$$L_2 = 00 + \texttt{step}_1 \cdot \texttt{cum\_count}_1 = 00 + 10 \cdot 2 = 20$$
$$H_2 = 00 + \texttt{step}_1 \cdot \texttt{cum\_count}_2 - 1 = 00 + 10 \cdot 3 - 1 = 30 - 1 = 2F$$

Wir sehen, dass die oberen drei Bits von H und L gleich sind $(010)_b$. Wir schieben diese raus und erhalten für L und H

$$L_2 = 00$$
$$H_2 = 7F$$

Und so fahren wir fort. Der Kode für diese zwei Buchstaben ist 00010. $\Diamond$

2.6.3 Skalierung bei Unterlauf

Es kann passieren, dass sich das Intervall immer um die Mitte des Intervalls (`range`) verkleinert. Dann bleibt L in der unteren und H in der oberen Hälfte des Intervalls. In diesem Fall bleiben die oberen Bits gleich und eine Skalierung ist nicht möglich. Das führt irgend wann dazu, dass die Präzision für das Kodieren nicht mehr reicht. Wir benennen das — auch wieder willkürlich — UNTERLAUF. [4] Dieses Problem des Unterlaufs löst man folgendermaßen:

Der Unterlauf entsteht nur, wenn L in der unteren Hälfte und H in der ober Hälfte des Intervalls ist. Dann ist das höchste Bit von L gleich 0 und von H gleich 1. Sobald das zweithöchste Bit von L Eins ist, und von H Null, ist das Intervall innerhalb der Grenzen des zweiten und dritten Quadranten. Das heißt es könnte wieder skaliert werden, denn dann ist es kleiner als die Hälfte des ursprünglichen Intervalls. Wir erweitern das Intervall, indem von L und H ein Viertel des Gesamt-Intervalls abziehen und dann verdoppeln. Allerdings wissen wir noch nicht, in welche Hälfte des Intervalls das Teil-Intervall fallen wird.

[4] im Gegensatz zum Überlauf, läuft der Wert hier nach rechts runter

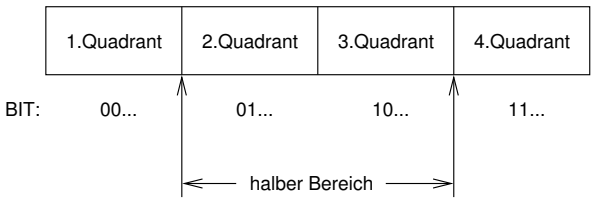

Abbildung 2.3: Intervalleinteilung in Quadranten

Deshalb wird skaliert, und gleichzeitig ein Zähler inkrementiert, der solange zählt, bis wieder eine Skalierung wegen eines Überlaufs stattfindet. Erst hier wird dann wieder Kode generiert. Falls die höchsten Bits Null waren wird hier eine Eins kodiert und dann werden so viele Nullen wie der Zähler gezählt hat hinzugefügt. Falls die höchsten Bits beide Eins waren, wird eine Null und entsprechend dem Zähler so viele Eisen kodiert. Einen Beweis dafür, dass das in der Folge geht liefert E. Bodden M. Clasen und J. Kneiss in [BCK02]. Wir wollen diesen Vorgang zum besseren Verständnis hier erklären.

Wir benennen die einzelnen Skalierungen wiefolgt

$E1 =$ Überlauf Skalierung in der unteren Hälfte des Intervalls
$E2 =$ Überlauf Skalierung in der oberen Hälfte des Intervalls
$E3 =$ Skalierung bei Unterlauf

Weiter soll $g \circ f$ die hintereinander Ausführung von f und danach g bedeuten.

Im Falle, dass sich das Intervall um die Mitte des Ursprungs Intervalls verkleinert entsteht die Folge $(E1)^n \circ (E2)$ oder $(E2)^n \circ (E1)$ von Skalierungen. Aus der oben genannten Problematik würden wir aber nicht Skalieren können, weil sich die oberen Bits nie gleichen. Mit der $E3$ Skalierung erreichen wir folgende Gleichheiten:

$$E1 \circ (E3)^n = (E2)^n \circ (E1)$$
$$E2 \circ (E3)^n = (E1)^n \circ (E2)$$

und können n mal $E3$ skalieren, bis eine Überlauf Skalierung eintritt, bei der dann der Kode ausgegeben wird.

Beweis für den Algorithmus der Unterlauf Skalierung 1. Die Skalierungsfunktionen sehen wiefolgt aus

$$E1\begin{pmatrix} L \\ H \end{pmatrix} = \begin{pmatrix} 2L \\ 2H \end{pmatrix}$$
$$E2\begin{pmatrix} L \\ H \end{pmatrix} = \begin{pmatrix} 2L-1 \\ 2H-1 \end{pmatrix}$$
$$E3\begin{pmatrix} L \\ H \end{pmatrix} = \begin{pmatrix} 2L-(1/2) \\ 2H-(1/2) \end{pmatrix}$$

Die erste Iteration ergibt

$$(E1 \circ E1)\begin{pmatrix} L \\ H \end{pmatrix} = \begin{pmatrix} 2 \cdot 2L \\ 2 \cdot 2H \end{pmatrix} = \begin{pmatrix} 2^2L \\ 2^2H \end{pmatrix}$$
$$(E2 \circ E2)\begin{pmatrix} L \\ H \end{pmatrix} = \begin{pmatrix} 2 \cdot (2L-1)-1 \\ 2 \cdot (2H-1)-1 \end{pmatrix} = \begin{pmatrix} 2^2L-3 \\ 2^2H-3 \end{pmatrix} = \begin{pmatrix} 2^2L-2^2-1 \\ 2^2H-2^2-1 \end{pmatrix}$$
$$(E3 \circ E3)\begin{pmatrix} L \\ H \end{pmatrix} = \begin{pmatrix} 2 \cdot (2L-0.5)-0.5 \\ 2 \cdot (2H-0.5)-0.5 \end{pmatrix} = \begin{pmatrix} 2^2L-1.5 \\ 2^2H-1.5 \end{pmatrix} = \begin{pmatrix} 2^2L-2^1+0.5 \\ 2^2H-2^1+0.5 \end{pmatrix}$$

und die n-te

$$E1^n\begin{pmatrix} L \\ H \end{pmatrix} = \begin{pmatrix} 2^nL \\ 2^nH \end{pmatrix}$$
$$E2^n\begin{pmatrix} L \\ H \end{pmatrix} = \begin{pmatrix} 2^nL-2^n+1 \\ 2^nH-2^n+1 \end{pmatrix}$$
$$E3^n\begin{pmatrix} L \\ H \end{pmatrix} = \begin{pmatrix} 2^nL-2^{n-1}+0.5 \\ 2^nH-2^{n-1}+0.5 \end{pmatrix}$$

Einen Beweis für diese Folgerung liefert eine vollständige Induktion. Mit

$$(E1 \circ (E3)^n)\begin{pmatrix} L \\ H \end{pmatrix} = E1\begin{pmatrix} 2^nL-2^{n-1}+0.5 \\ 2^nH-2^{n-1}+0.5 \end{pmatrix} = \begin{pmatrix} 2^{n+1}L-2^n+1 \\ 2^{n+1}H-2^n+1 \end{pmatrix} \tag{2.15}$$
$$((E2)^n \circ E1)\begin{pmatrix} L \\ H \end{pmatrix} = (E2)^n\begin{pmatrix} 2L \\ 2H \end{pmatrix} = \begin{pmatrix} 2^{n+1}L-2^n+1 \\ 2^{n+1}H-2^n+1 \end{pmatrix} \tag{2.16}$$

Damit gilt mit den Gleichungen (2.15) und (2.16)

$$E1 \circ (E3)^n = (E2)^n \circ E1$$

□

Beispiel 9. Wir nehmen Beispiel 8 wieder auf und lesen als nächstes den Buchstaben c ein.

$$\texttt{range}_2 = 7F - 00 + 1 = 80$$
$$\texttt{step}_2 = 80/8 = 10$$
$$L_3 = 00 + \texttt{step}_2 \cdot \texttt{cum\_count}_2 = 00 + 10 \cdot 3 = 30 = 0011\_0000_b$$
$$H_3 = 00 + \texttt{step}_2 \cdot \texttt{cum\_count}_3 - 1 = 00 + 10 \cdot 6 - 1 = 60 - 1 = 5F = 0101\_1111_b$$

Die Bits 7 sind ungleich, aber Bit 6 ist bei L Eins und bei H Null. Das Intervall liegt im zweiten und dritten Quadranten und kann skaliert werden. Wir merken uns, dass einmal für den Unterlauf skaliert wurde.

$$L_3 = 30 - 20 = 10 = 0001\_0000_b$$
$$H_3 = (5F - 20) \cdot 2 = 7E = 1101\_1110_b$$

Bit 8 wird maskiert, und zu H noch 1 addiert:

$$L_3 = 20 = 0010\_0000_b$$
$$H_3 = 7F = 0101\_1111_b$$

Erst wenn eine Skalierung aufgrund eines Überlaufs entsteht, wird der Kode generiert und ausgegeben. $\Diamond$

Die Details kann man aus dem Algorithmus in C vom Listing 2.1 entnehmen.

Die vollständige Kodierung ist in Tabelle 2.7 zusammengefasst. Der so erhaltene Kode ist 00010101001101111_b.

Listing 2.1 Algorithmus für Kodierung (Bit Breite 8)

```
...
   for (;;)
   {
      t1 = *low & 0x40; t2 = *high & 0x40;
      if ( t1 == t2 ) // check high bit
      { // overflow
         if ( t1 >= 1 ) // high bit is 1
         {
            setbit (output, output_counter, 1);
            while ( *udcount > 0 )
            {
               setbit (output, output_counter, 0);
               (*udcount)--;
            }
         }
         else
         {
            setbit (output, output_counter, 0);
            while ( *udcount > 0 )
            {
               setbit (output, output_counter, 1);
               (*udcount)--;
            }
         }
         shift ( low, high );
      }
      else
      { // check for underflow
         t1 = *low & 0x20; t2 = *high & 0x20;// second highest bit
         if ( t1 > t2 )
         {
            do { // shift low and high but no output
               (*udcount)++; // underflow counter
               *low = *low & 0x1f; // same as subtracting 20 as bit must be 1
               *high = *high | 0x20; // same as adding 20 as bit must be 0
               shift ( low, high );
               t1 = *low & 0x20;
               t2 = *high & 0x20;
            } while ( t1 > t2 );
         }
         else
         { // no more shifting
            break;
         }
      }
   }
...
void shift ( unsigned short *low, unsigned short *high )
{
   *low <<= 1;
   *high <<= 1;
   *high = *high | 0x01; // same as adding one
   /* mask high bit as it's not used for calculation */
   *low = *low & 0x7f;
   *high = *high & 0x7f;
}
```

Sym	lc	hc	rng	L_i	H_i	KdBts	ÜL-Low	ÜL-High	cnt	UNL-Low	UNL-High
a	0	2	10	0000000 [0]	0011111 [1F]	00	0000000 [0]	1111111 [7F]	0		
b	2	3	10	0100000 [20]	0101111 [2F]	010	0000000 [0]	1111111 [7F]	0		
c	3	6	C	0110000 [30]	1011111 [57]				1	0100000 [20]	1111111 [7F]
c	3	6	9	1000100 [44]	1100111 [67]	10			0	0001000 [8]	1001111 [4F]
e	7	8	9	1000111 [47]	1001111 [4F]	100	0111000 [38]	1111111 [7F]	0		
d	6	7	9	1101111 [6E]	1110110 [76]	11	0111000 [38]	1011011 [5B]	1	0110000 [30]	1110111 [77]
a	0	2	9	0110000 [30]	1000001 [41]				3	0000000 [00]	1000111 [47]
c	2	3	9	0011011 [1B]	0110101 [35]	0111	0110110 [36]	1101011 [3B]	0		
rest						1					

Tabelle 2.7: Kodierung von «abccedac»

Sym Symbol

lc low_count

hc high_count

L_i untere Grenze des Intervalls beim Buchstaben i

H_i obere Grenze des Intervalls beim Buchstaben i

KdBits Kode Bits

ÜL-Low untere Grenze des Intervalls nach einer Überlauf Skalierung

ÜL-High obere Grenze des Intervalls nach einer Überlauf Skalierung

cnt Zähler für Unterlauf Skalierung

UNL-Low untere Grenze des Intervalls nach einer Unterlauf Skalierung

UNL-High obere Grenze des Intervalls nach einer Unterlauf Skalierung

2.7 Dekodierung als begrenzte Festkommazahl

Das Dekodieren als begrenzte Festkommazahl unterscheidet sich nicht wesentlich vom dem im Kapitel 2.3 vorgestellten Verfahren. Hinzu kommt nur, dass man hier die Überlauf- und Unterlauf-Skalierungen berücksichtigen muss.

Beim Puffer `buf`, der den Kode enthält, muss man beim links Schieben die nächsten Bits des Kodes nachschieben. Also anders als die festen Werte die Bei L und H nachgeschoben werden.

Die Subtraktion von einem Quadranten bei der Unterlauf Skalierung kann man durch einfache Bit Manipulation sowohl für L, H als auch `buf` ersetzen. Nur die oberen Bits sind von Bedeutung. Wir lassen die unteren Bits weg und schreiben dafür $XXX\cdots$. Betrachten wir als ersten den Fall L. Das Puffer-Bit ist eingeklammert, weil es für die Rechnung nicht relevant ist.

$$(0)010\cdots_b \leq L \leq (0)011\cdots_b$$

oder mit anderen Worten: L liegt im zweiten Quadranten. (siehe auch Bild 2.3 Damit ist das zweithöchste Bit immer gesetzt. Das entspricht einem Viertel des Intervalls. Setzt man dieses Bit auf 0, entspricht es dem Abzug von einem Viertel.

Für H gilt

$$(0)100\cdots_b \leq H \leq (0)101\cdots_b$$

oder mit anderen Worten: H liegt im dritten Quadranten. Zwar manipuliert eine Subtraktion in der zweithöchsten Stelle auch die höchste Stelle, diese wird aber bei der anschließenden Multiplikation mit 2 raus geschoben und hat damit keine Relevanz. Wir können also einfach das zweithöchste Bit von 0 auf 1 setzen.

Für `buf` ist die Betrachtung ähnlich. Der Wertebereich ist jedoch größer:

$$(0)010\cdots_b \leq \texttt{buf}_b \leq (0)101\cdots_b$$

Wir haben aber gesehen, dass sowohl für L als auch für H das zweithöchste Bit einfach negiert wird. Diese Regel gilt also für den gesamten Bereich von `buf`. Wir können damit die Subtraktion von einem Viertel so formulieren:

$$\texttt{buf}_b = \texttt{buf}_b \quad \text{xor} \quad (0)010\cdots_b$$

Wir verzichten an dieser Stelle auf ein Beispiel und zeigen den Algorithmus in Listing 2.2. Das Ergebnis der Dekodierung ist in Tabelle 2.8 dargestellt.

Listing 2.2 Algorithmus für Dekodierung (Bit Breite 8)

```
   ...
   for (;;)
   {
      t1 = *low & 0x4000;
      t2 = *high & 0x4000;
      if ( t1 == t2 )
      {
         // overflow
         while ( *udcount > 0 )
         {
            (*udcount)--;
         }
         shift_d ( low, high, buf, output, output_counter );
      }
      else
      {
         // check for underflow
         t1 = *low & 0x20;
         t2 = *high & 0x20;
         if ( t1 > t2 )
         {
            *low = *low & 0x1f;
            *high = *high | 0x20;
            *buf = *buf ^ 0x20;
            shift_d ( low, high, buf, output, output_counter );
            (*udcount)++;
         }
         else
         {
            // no more shifting
            break;
         }
      }
   }
   ...
void shift_d ( unsigned short *low, unsigned short *high,
   unsigned short *buf, int *output, int *output_counter )
{
   *low <<= 1;
   *high <<= 1;
   *high = *high | 0x01;
   *buf <<= 1;
   *buf = *buf | output[ *output_counter ];
   (*output_counter)++;
   *low = *low & 0x7f;
   *high = *high & 0x7f;
   *buf = *buf & 0x7f;
}
```

S	lc	hc	buf	rg	L_i	H_i	KB	ÜL-Low	ÜL-High	cn	UNL-Low	UNL-High
a	0	2	0001010 [0A]	10	0000000 [0]	0011111 [1F]	00	0000000 [0]	0011111 [1F]	0		
b	2	3	0101010 [2A]	10	0100000 [20]	0101111 [2F]	010	0100000 [20]	0101111 [2F]	0		
c	3	6	1010011 [53]	10	0110000 [30]	1011111 [57]				1	0110000 [30]	1011111 [5F]
c	3	6	1100110 [66]	C	1000100 [44]	1100111 [67]	10	0001000 [8]	1001111 [4F]	0		
e	7	8	1001101 [4D]	E	1000111 [47]	1001111 [4F]	100	0111000 [38]	1111111 [7F]	0		
d	6	7	1101111 [6F]	D	1101111 [6E]	1110110 [76]	11	0111000 [38]	1011011 [5B]	1	0110000 [30]	1110111 [77]
a	0	2	0111000 [38]	A	0110000 [30]	1000001 [41]				3	0000000 [00]	1000111 [47]
c	2	3	0100000 [20]	C	0011011 [1B]	0110101 [35]	0111			0		

Tabelle 2.8: Dekodierung von «abccedac»

S Symbol

lc low_count

hc high_count

buf Buffer mit dem Kode

L_i untere Grenze des Intervalls beim Buchstaben i

H_i obere Grenze des Intervalls beim Buchstaben i

KB Kode Bits

ÜL-Low untere Grenze des Intervalls nach einer Überlauf Skalierung

ÜL-High obere Grenze des Intervalls nach einer Überlauf Skalierung

cn Zähler für Unterlauf Skalierung

UNL-Low untere Grenze des Intervalls nach einer Unterlauf Skalierung

UNL-High obere Grenze des Intervalls nach einer Unterlauf Skalierung

3 Parallelisierung

Es wird zunehmend schwieriger, die Taktraten der Integrierten Schaltungen zu steigern. Die immer dichter werdenden Schaltungen erreichen die theoretisch machbaren Grenzen. Leckströme werden zunehmend zum Problem. Die Rechenleistung der heutigen Rechner lässt sich nicht mehr allein durch höhere Taktraten steigern.

Eine Möglichkeit, dennoch die Leistung zu steigern, ist das Parallelisieren. In Großrechenanlagen hat dieses Vorgehen schon längst Einzug gehalten. Inzwischen findet dieser Trend verstärkt Anwendung auch im PC. Moderne CPUs haben inzwischen acht Kerne und Intel hat kürzlich ein «clowd on chip» Prozessor mit 48 Kernen vorgestellt[1]. Grafikkarten werden mittlerweile nicht nur zur Darstellung und Berechnung von Polygonen benutzt, sondern vermehrt für allgemeine Berechnungen. Diese sogenannten GPGPU General-purpose computing on Graphics Processing Unit mit hunderten von Kernen erreichen beeindruckende Geschwindigkeitssteigerungen. Diese GPGPUs verarbeiten die Daten doppelt bis dreißig Mal schneller als eine herkömmliche CPU[2].

Doch der wesentliche Punkt ist, dass diese Geschwindigkeitssteigerungen nur erreicht werden können, wenn der zugrundeliegende Algorithmus auf die gegebene Architektur parallelisierbar ist. Es ist also entscheidend, ob der Algorithmus parallelisierbar ist und wenn ja, wie stark.

Dieser Frage wollen wir für das Arithmetische Kodieren nachgehen. Wie schon in Kapitel 2 (Arithmetisches Kodieren) erwähnt, hat es sehr lange gedauert, bis hierzu etwas veröffentlicht wurde. J. Jiang und S. Jones haben 1994 erstmals einen parallelen Algorithmus vorgeschlagen [JJ94]. Nur wenige Papers sind danach zu diesem Thema veröffentlicht worden. Wir wollen diese im Einzelnen vorstellen und hinsichtlich einer konkreten Umsetzung in Hardware analysieren.

[1] http://www.pcper.com/article.php?aid=825
[2] http://code.google.com/p/pyrit/

3.1 Parallelisierung nach J. Jiang und S. Jones

3.1.1 Einleitung

Dieses Paper ist das erste, das zum Thema Parallelisierung des Arithmetischen Kodierens geschrieben wurde. In diesem Paper verwenden Jiang und Jones eine Vorwärtskonvention im Gegensatz zu einer Rückwärtskonvention. Damit ist die Bezeichnung des Teilintervalls gemeint. Bei unserer bisherigen Rückwärtskonvention gilt

$$p(x_i) = F_x(i) - F_x(i-1)$$

und bei einer Vorwärtskonvention

$$p(x_i) = F_x(i+1) - F_x(i)$$

Der Konsistenz wegen bleiben wir bei der Rückwärtskonvention.

Wir nehmen die Differenzengleichungen 2.6 und 2.7

$$\begin{aligned} L_i &= L_{i-1} + (H_{i-1} - L_{i-1}) \cdot F_x(i-1) \\ H_i &= L_{i-1} + (H_{i-1} - L_{i-1}) \cdot F_x(i) \end{aligned}$$

Weil

$$\begin{aligned} r_i &= H_i - L_i \\ &= [L_{i-1} + (H_{i-1} - L_{i-1}) \cdot F_x(i)] - [L_{i-1} + (H_{i-1} - L_{i-1}) \cdot F_x(i-1)] \\ &= (H_{i-1} - L_{i-1}) \cdot (F_x(i) - F_x(i-1)) \\ &= r_{i-1} \cdot (F_x(i) - F_x(i-1)) \\ &= r_{i-1} \cdot p(x_i) \end{aligned} \tag{3.1}$$

und

$$\begin{aligned} L_i &= L_{i-1} + (H_{i-1} - L_{i-1}) \cdot F_x(i-1) \\ &= L_{i-1} + r_{i-1} \cdot F_x(i-1) \end{aligned}$$

kann man die Gleichungen (2.6) und (2.7) etwas anders schreiben. Wir rechnen H nicht mehr explizit aus, sondern nur noch L und die dazugehörige Länge des Teilintervalls r_i.

$$\begin{aligned} L_i &= L_{i-1} + r_{i-1} \cdot F_x(i-1) \\ r_i &= r_{i-1} \cdot p(x_i) \end{aligned}$$

3.1.2 Prinzip

Wir nehmen zur Vereinfachung an, dass die Sequenz aus einer geraden Anzahl von Symbolen besteht $|S| = 2n$. Eine ungerade Länge kann man leicht auf eine gerade erweitern.

Um den Algorithmus zu parallelisieren, verfolgen wir, die Ergebnisse der Berechnungen, wenn man diese ausschreibt.

$$
\begin{aligned}
r_1 &= r_0 \cdot p(x_0) \\
r_2 &= r_1 \cdot p(x_1) = (r_0 \cdot p(x_0)) \cdot p(x_1) \\
r_3 &= r_2 \cdot p(x_2) = ((r_0 \cdot p(x_0)) \cdot p(x_1)) \cdot p(x_2) \\
r_4 &= r_3 \cdot p(x_3) = (((r_0 \cdot p(x_0)) \cdot p(x_1)) \cdot p(x_2)) \cdot p(x_3) \\
&\vdots \\
r_{2n} &= r_0 \cdot p(x_0) \cdot p(x_1) \cdot p(x_2) \cdot p(x_3) \dots p(x_{2n-1})
\end{aligned} \tag{3.2}
$$

Und für L

$$
\begin{aligned}
L_1 &= L_0 + r_0 \cdot F_x(0) \\
L_2 &= L_1 + r_1 \cdot F_x(1) \\
&= (L_0 + r_0 \cdot F_x(0)) + r_1 \cdot F_x(1) \\
&= (L_0 + r_0 \cdot F_x(0)) + r_0 \cdot p(x_0) \cdot F_x(1) \\
&= L_0 + r_0 \cdot F_x(0) + r_0 \cdot p(x_0) \cdot F_x(1) \\
L_3 &= L_2 + r_2 \cdot F_x(2) \\
&= (L_0 + r_0 \cdot F_x(0) + r_0 \cdot p(x_0) \cdot F_x(1)) + r_2 \cdot F_x(2) \\
&= (L_0 + r_0 \cdot F_x(0) + r_0 \cdot p(x_0) \cdot F_x(1)) + (r_0 \cdot p(x_0)) \cdot p(x_1) \cdot F_x(2) \\
&= L_0 + r_0 \cdot F_x(0) + r_0 \cdot p(x_0) \cdot F_x(1) + r_0 \cdot p(x_0) \cdot p(x_1) \cdot F_x(2) \\
&\vdots \\
L_{2n} &= L_0 + r_0 \cdot F_x(0) + r_0 \cdot p(x_0) \cdot F_x(1) + r_0 \cdot p(x_0) \cdot p(x_1) \cdot F_x(2) + \\
&\quad \dots + r_0 \cdot p(x_0) \cdot p(x_1) \dots p(x_{2n-2}) \cdot F_x(2n-1)
\end{aligned} \tag{3.3}
$$

Wir teilen jetzt die Berechnungen (3.2) und (3.3) in zwei gleich große Abschnitte und erhalten

$$
\begin{aligned}
r_{1..n} &= r_0 \cdot p(x_0) \cdot p(x_1) \cdot p(x_2) \cdot p(x_3) \dots p(x_{n-1}) \\
L_{1..n} &= L_0 + r_0 \cdot F_x(0) + r_0 \cdot p(x_0) \cdot F_x(1) + r_0 \cdot p(x_0) \cdot p(x_1) \cdot F_x(2) + \\
&\quad \dots + r_0 \cdot p(x_0) \cdot p(x_1) \dots p(x_{n-1}) \cdot F_x(n-1)
\end{aligned}
$$

und

$$
\begin{aligned}
r_{(n+1)..2n} &= r_0 \cdot p(x_{n+1}) \cdot p(x_{n+2}) \cdot p(x_{n+3}) \cdot p(x_{n+4}) \dots p(x_{2n-1}) \\
L_{(n+1)..2n} &= L_0 + r_0 \cdot F_x(n+1) + \\
&\qquad r_0 \cdot p(x_{n+1}) \cdot F_x(n+2) + \\
&\qquad r_0 \cdot p(x_{n+1}) \cdot p(x_{n+2}) \cdot F_x(n+3) + \dots \\
&\qquad + r_0 \cdot p(x_{n+1}) \cdot p(x_{n+2}) \dots p(x_{2n-2}) \cdot F_x(2n-1)
\end{aligned}
$$

Somit ist r_{2n} nichts anderes als

$$r_{2n} = \frac{r_{1..n} \cdot r_{(n+1)..2n}}{r_0} \tag{3.4}$$

Wir schreiben L_{2n} nochmal komplett aus und fügen zusammen

$$\begin{aligned} L_{2n} =& L_0 + r_0 \cdot F_x(0) + r_0 \cdot p(x_0) \cdot F_x(1) + r_0 \cdot p(x_0) \cdot p(x_1) \cdot F_x(2) + \dots \\ &+ r_0 \cdot p(x_0) \cdot p(x_1) \dots \cdot p(x_{n-1}) \cdot F_x(n) + \dots \\ &+ r_0 \cdot p(x_0) \cdot p(x_1) \dots \cdot p(x_n) \cdot F_x(n+1) + \dots \\ &+ r_0 \cdot p(x_0) \cdot p(x_1) \dots \cdot p(x_{2n-2}) \cdot F_x(2n-1) \\ =& L_{1..n} + r_{1..n} \cdot \\ & (F_x(n+1) + \dots + r_0 \cdot p(x_0) \cdot p(x_1) \dots p(x_{2n-2}) \cdot F_x(2n-1)) \end{aligned}$$

Wir erweitern die abgesetzte Zeile mit r_0

$$\begin{aligned} L_{2n} =& L_{1..n} + r_{1..n} \\ & \left(\frac{r_0 \cdot [F_x(n+1) + \dots + r_0 \cdot p(x_0) \cdot p(x_1) \dots p(x_{2n-2}) \cdot F_x(2n-1)]}{r_0} \right) \end{aligned}$$

und erweitern noch mit L_0

$$\begin{aligned} L_{2n} =& L_{1..n} + r_{1..n} \\ & \left(\frac{L_0 + [r_0 \cdot F_x(n+1) + \dots + r_0 \cdot p(x_0) \cdot p(x_1) \dots p(x_{2n-2}) \cdot F_x(2n-1)] - L_0}{r_0} \right) \end{aligned}$$

und erhalten

$$\begin{aligned} L_{2n} =& L_{1..n} + r_{1..n} \left(\frac{L_{(n+1)..2n} - L_0}{r_0} \right) \\ =& L_{1..n} + \frac{r_{1..n}(L_{(n+1)..2n} - L_0)}{r_0} \end{aligned} \tag{3.5}$$

Die Differenzengleichungen (3.5) und (3.4) ergeben die neue Rechenvorschrift für das Kodieren. Setzen wir $L_0 = 0$, erhalten wir

$$L_{2n} = L_{1..n} + \frac{r_{1..n} \cdot L_{(n+1)..2n}}{r_0} \tag{3.6}$$

$$r_{2n} = \frac{r_{1..n} \cdot r_{(n+1)..2n}}{r_0} \tag{3.7}$$

Wir können Gleichung (3.2) und (3.3) immer weiter nach dieser Regel aufteilen, bis zum Schluss nur noch ein Symbol kodiert werden muss. Die Formel für diese Kodierung ist dann

$$\begin{aligned} L_1 =& L_0 + r_0 \cdot F_x(0) \\ r_1 =& r_0 \cdot p(x_0) \end{aligned}$$

Weil $L_0 = 0$ gilt

$$L_1 = r_0 \cdot F_x(0) \tag{3.8}$$
$$r_1 = r_0 \cdot p(x_0) \tag{3.9}$$

Wir können also zwei verschiedene Rechenvorschriften definieren. Eine zum Berechnen von L und r, und eine um zwei Ls und rs zusammenzufügen. In diesem Paper sprechen die Autoren von einer `BPE` und einer `GPE` . Die `BPE`s berechnen ein L_{x_i} und r_{x_i} zum Symbol x_i und die `GPE` konkateniert die Berechnungen zu den Symbolen x_i und x_{i+1}. Das Ergebnis dieser Zusammenführung wird mit der nächsten Berechnung in einer weiteren Stufe zusammengefügt. Auf diese Weise kann man beliebig viele Symbole durch `BPE`s einlesen und in

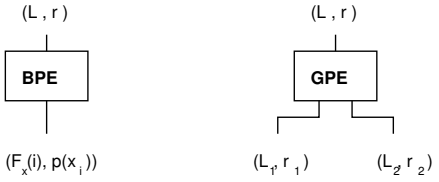

Abbildung 3.1: Basisformeln für Parallelisierung

einer baumartigen Struktur von `GPE`s zusammenfügen. Abbildung 3.2 zeigt ein Beispiel mit 8 Symbolen. In der obersten `GPE` erhält man dann das neue L und r. Es stellt sich nun die

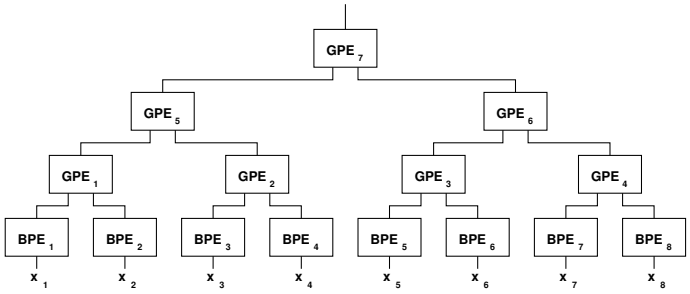

Abbildung 3.2: Berechnungs-Baum für 8 Symbole

Frage, wie die Berechnung fortgeführt werden kann. Eine Möglichkeit wäre, so viele `BPE`s und `GPE`s zur Verfügung zu stellen, wie es Symbole gibt. Dann wäre die Länge des Textes begrenzt und außerdem würde die dafür benötigte Präzision zu groß werden, wie wir später erfahren werden.

Ein anderer Ansatz ist, für die Berechnungen die Differenzengleichungen (3.4) und (3.5) statt (3.6) und (3.7) für die `GPE`s zu nehmen. Dann setzt man für L_0 das zuvor errechnete L ein und erhält bei `GPE7` das neue L und r, welches dann für den nächsten Schritt verwendet wird. So kann man immer 8 Symbole gleichzeitig einlesen und den Vorgang durch eine parallele Implementierung der Bausteine in

$$m = (\log_b k) + 1$$

Schritten berechnen. Dabei ist k die Anzahl der gleichzeitig eingelesenen Symbole. In unserem Beispiel ist $k = 8$ und damit $m = 4$.

Eine weitere Möglichkeit ist, die vorhergehende Berechnung und die aktuelle aus der höchsten BPE mit einer weiteren GPE zu konkatenieren. Wir würden also für die GPEs weiterhin die Differenzengleichungen (3.6) und (3.7) verwenden, aber nach der höchsten GPE eine weitere Stufe anbringen, die das vorangehende L und r mit dem der höchsten GPE zusammmenfügt. Das ist nichts anderes als das vorhergehende Ergebnis in einem weiteren Durchlauf in einer nächsten GPE wieder einzufügen. Bild 3.3 zeigt das in unserem Beispiel.

Für den ersten Durchlauf muss dann für diese — nun höchste GPE — $L = 0$ und $r = \text{max\_val}$ gesetzt werden. In unserem Beispiel GPE8. Die Berechnung von L und r stünden somit nach

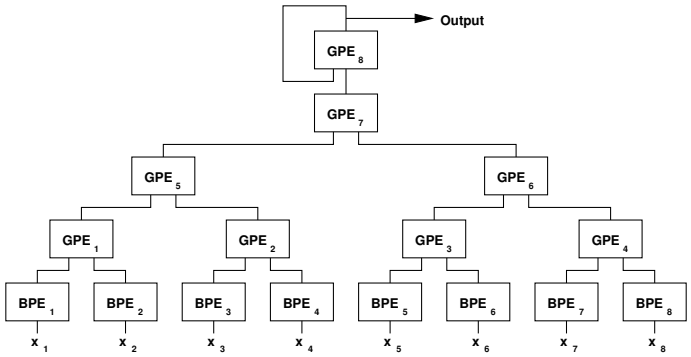

Abbildung 3.3: Berechnungs-Baum für 8 Symbole + Konkatenation

$$m = (\log_b k) + 2$$

Schritten fest.

3.1.3 Normalisierung

Nachdem nun die Berechnung schrittweise parallel durchgeführt werden kann, stellt sich die Frage nach der erforderlichen Präzision. Im sequenziellen Fall wurde nach jedem Berechnungsschritt die Normalisierung durchgeführt. Durch diese Maßnahme wurde effektiv ein Über- und Unterlauf verhindert.

J. Jiang und S. Jones schlagen dieses auch in ihrem Paper vor. Hiernach soll bei jeder GPE eine lokale Skalierung stattfinden. Die auf der linken Seite anliegenden Werte bei einer GPE entsprechen dem alten L und r. Von der rechten Seite wird mit L_2 und r_2 die Berechnung fortgeführt. Das heißt, dass für die Werte an der rechten Seite die volle Präzision vorhanden sein muss. Die Autoren schlagen hierfür einen Puffer vor, der mitgeführt wird und die Bits puffert, die bei einer Skalierung herausgeschoben werden. Dieser Puffer wird der darüber liegenden GPE weitergegeben und für die Berechnung von L und r herangezogen. Somit muss an der obersten GPE alle herausgeschobenen Bits berücksichtigt werden. Man hat hier

also innerhalb der Baumstruktur — oder des parallelen Blocks — keine reale Skalierung durchgeführt.

Wir könnten also, anstatt nach jedem Schritt zu skalieren, die Berechnungen bis zur obersten `GPE` durchrechnen und erst dann skalieren. Bei genügender Präzision ist es egal, wann man skaliert, solange Koder und Dekoder nach der gleichen Rechenregel vorgehen. Es ist klar, dass eine höhere Präzision erforderlich sein wird, wenn man länger rechnet bis man skaliert.

Diese Erkenntnis ermöglicht somit eine vollständige Berechnung des parallelen Blocks und im Anschluss die Normalisierung. Wir müssen hierfür lediglich die Präzision erhöhen und können somit, ohne nach jedem Schritt auf die Normalisierung zu warten und ohne zusätzliche Maßnahmen, zu Ende rechnen. Um wieviel die Präzision erhöht werden muss, ist Gegenstand von Kapitel 3.4.2.

Die Normalisierung gestaltet sich hier wesentlich einfacher als die der Sequentiellen Vorgehensweise mit der Berechnung von L und H. Wir berechnen hier r direkt und können dieses auch direkt kontrollieren. Im Paper [JJ94] ist kein spezieller Algorithmus dafür beschrieben. Allein die Idee ist hier skizziert. Wir werden im Kapitel 3.5.2 noch einen Algorithmus vorstellen und analysieren.

3.1.4 Stark unterschiedliche Häufigkeiten

Ein weiteres Problem könnte auftreten, wenn die Häufigkeiten der einzelnen Symbole sehr unterschiedlich sind. Wenn also

$$\exists \quad i \neq j \quad i,j \in A \quad | \quad p(x_i) \gg p(x_j)$$

Dann kann es passieren, dass sehr lange nicht skaliert wird. Vor allem dann nicht, wenn zusätzlich $p(x_i)$ sehr nahe bei r_0 (dem Ursprungs-Intervall) liegt. Das kann zur Folge haben, dass nach einiger Zeit auch hier die Präzision nicht mehr ausreicht.

In diesem Paper wird als Lösung ein Tausch der Reihenfolge der Symbole vorgeschlagen. Das «end of file» Symbol wird als Puffer zur oberen Grenze des Intervalls gesetzt. Wir werden diese Problematik in Kapitel 3.4.2 wieder aufnehmen.

3.2 Parallelisierung nach J. Šupol und B. Melichar

3.2.1 Einleitung

Zehn Jahre nach der Veröffentlichung von [JJ94] haben Jan Šupol und Bořivoj Melichar in einem Paper [SM05] ein weiteres Verfahren zur Parallelisierung vorgeschlagen. Bei diesem Verfahren gehen die Autoren davon aus, dass beliebig viele Prozessoren zur Verfügung stehen. Sie verwenden hierzu das Modell eines Exclusive Read Exclusive Write Parallel RAM (EREW PRAM).

Bei Ausnutzung voller Parallelität ergibt das eine Kodierung eines Textes S in $\log k$ Schritten, wobei $k = |S|$ bei Annahme, dass k Prozessoren zur Verfügung stehen.

3.2.2 Prinzip

Wieder nehmen wir die Differenzen Gleichungen (2.6) und (2.7).

$$L_i = L_{i-1} + (H_{i-1} - L_{i-1}) \cdot F_x(i-1)$$
$$H_i = L_{i-1} + (H_{i-1} - L_{i-1}) \cdot F_x(i)$$

Wir setzen

$$LR_i = (H_{i-1} - L_{i-1}) \cdot F_x(i-1) \tag{3.10}$$
$$HR_i = (H_{i-1} - L_{i-1}) \cdot F_x(i)$$

und schreiben die Gleichungen (2.6) und (2.7) als

$$L_i = L_{i-1} + LR_i \tag{3.11}$$
$$H_i = L_{i-1} + HR_i \tag{3.12}$$

Wenn wir nun (3.11) ausschreiben, erhalten wir

$$\begin{aligned} L_i &= L_{i-1} + LR_i = L_{i-2} + LR_i + LR_{i-1} \\ &= L_0 + LR_i + LR_{i-1} + LR_{i-2} + \ldots + LR_0 \\ &= \sum_{k=0}^{i} LR_k + L_0 \end{aligned} \tag{3.13}$$

und weil $L_0 = 0$

$$L_i = \sum_{k=0}^{i} LR_k$$

Entsprechend kann man das auch für Gleichung (3.12) berechnen und erhält

$$H_i = \sum_{k=0}^{i} HR_k + H_0$$

wobei H_0 die oberste Grenze des Ursprungs Intervalls ist. Wir setzen nun

$$r_i = H_i - L_i$$

und erhalten aus Gleichung (3.10)

$$LR_i = r_{i-1} \cdot F_x(i-1)$$

Wir erinnern uns an Gleichung (3.1) wonach

$$r_i = r_{i-1} \cdot p(x_i)$$

und schreiben

$$\begin{aligned} LR_i &= r_{i-1} \cdot F_x(i-1) \\ &= r_{i-2} \cdot p(x_{i-1}) \cdot F_x(i-1) \\ &= r_{i-3} \cdot p(x_{i-2}) \cdot p(x_{i-1}) \cdot F_x(i-1) \\ &= r_{i-4} \cdot p(x_{i-3}) \cdot p(x_{i-2}) \cdot p(x_{i-1}) \cdot F_x(i-1) \\ &\vdots \\ &= r_0 \cdot (\prod_{k=0}^{i-1} p(x_k)) \cdot F_x(i-1) \end{aligned} \tag{3.14}$$

wobei

$$\prod_{j=0}^{i} p(x_j) = 1 \quad i < 0 \tag{3.15}$$

Wir setzen Gleichung (3.14) in (3.13) ein

$$\begin{aligned} L_i &= \sum_{k=1}^{i} LR_k + L_0 \\ &= \sum_{k=1}^{i} [r_0 \cdot (\prod_{j=0}^{k-2} p(x_j)) \cdot F_x(k-1)] + L_0 \end{aligned} \tag{3.16}$$

beachte dass das innere Produkt jetzt mit j läuft, und weil die äußere Summe bei $k = 1$ beginnt, darf das Produkt nur bis $k-2$ laufen. Falls $r_0 = 1$ und $L_0 = 0$ gilt

$$L_i = \sum_{k=1}^{i} [(\prod_{j=0}^{k-2} p(x_j)) \cdot F_x(k-1)] \tag{3.17}$$

und für H

$$H_i = \sum_{k=1}^{i} [(\prod_{j=0}^{k-2} p(x_j)) \cdot F_x(k)] \tag{3.18}$$

wobei gelten soll

$$p(x_i) = F_x(i) := 0 \quad \text{für} \quad i < 0$$

Die Idee ist nun, die Produkte und Summen aus Gleichungen (3.17) und (3.18) parallel in einem Baum auszurechnen.

3.2.3 Beispiel

In diesem Paper wird das Beispiel aus Kapitel 2.2 berechnet. $A = \{\texttt{S},\texttt{W},\texttt{I},\texttt{M},\sqcup\}$ und $S = \{\texttt{SWISS}\sqcup\texttt{MISS}\}$

$$
\begin{aligned}
LR_{-1} &= 0 \\
HR_{-1} &= 1
\end{aligned}
$$

$$
\begin{aligned}
LR_0 &= (HR_{-1} - LR_{-1}) \cdot F_x(x_0 - 1 = a_5 - 1 = a_{4=\texttt{W}}) \\
&= 1 \cdot F_x(4) \\
&= 1.0 \cdot 0.5 = 0.5 \\
HR_0 &= (HR_{-1} - LR_{-1}) \cdot F_x(x_0 = a_5 = a_{5=\texttt{S}}) \\
&= 1 \cdot F_x(5) \\
&= 1.0 \cdot 1.0 = 1.0
\end{aligned}
$$

$$
\begin{aligned}
LR_1 &= (HR_0 - LR_0) \cdot F_x(x_1 - 1 = a_4 - 1 = a_{3=\texttt{I}}) \\
&= (H_5 - L_5) \cdot F_x(3) \\
&= p(a_5) \cdot F_x(3) \\
&= 0.5 \cdot 0.4 = 0.2 \\
HR_1 &= (HR_0 - LR_0) \cdot F_x(x_1 = a_4 = a_{4=\texttt{W}}) \\
&= (H_5 - L_5) \cdot F_x(4) \\
&= p(a_5) \cdot F_x(4) \\
&= 0.5 \cdot 0.5 = 0.25
\end{aligned}
$$

$$
\begin{aligned}
LR_2 &= (HR_1 - LR_1) \cdot F_x(x_2 - 1 = a_3 - 1 = a_{2=\texttt{M}}) \\
&= (p(a_5) \cdot H_4 - p(a_5) \cdot L_4) \cdot F_x(2) \\
&= p(a_5) \cdot p(x_4) \cdot F_x(2) \\
&= 0.5 \cdot 0.1 \cdot 0.2 = 0.01 \\
HR_2 &= (HR_1 - LR_1) \cdot F_x(x_2 = a_{3=\texttt{I}}) \\
&= (p(a_5) \cdot H_4 - p(x_5) \cdot L_4) \cdot F_x(2) \\
&= p(a_5) \cdot p(x_4) \cdot F_x(2) \\
&= 0.5 \cdot 0.1 \cdot 0.4 = 0.02
\end{aligned}
$$

$$
\begin{aligned}
LR_3 &= (HR_2 - LR_2) \cdot F_x(x_3 - 1 = a_5 - 1 = a_{4=\mathtt{W}}) \\
&= (p(a_5) \cdot p(a_4) \cdot H_3 - p(a_5) \cdot p(a_4) \cdot L_3) \cdot F_x(4) \\
&= p(a_5) \cdot p(a_4) \cdot p(a_3) \cdot F_x(4) \\
&= 0.5 \cdot 0.1 \cdot 0.2 \cdot 0.5 = 0.005 \\
HR_3 &= (HR_2 - LR_2) \cdot F_x(x_3 = a_{5=\mathtt{S}}) \\
&= (p(a_5) \cdot p(a_4) \cdot H_3 - p(x_5) \cdot p(a_4) \cdot L_3) \cdot F_x(5) \\
&= p(a_5) \cdot p(a_4) \cdot p(a_3) \cdot F_x(5) \\
&= 0.5 \cdot 0.1 \cdot 0.2 \cdot 1.0 = 0.01 \\
&\vdots
\end{aligned}
$$

Addieren wir jetzt die LRs zusammen, erhalten wir die zugehörigen Ls.

$$
\begin{aligned}
L_{-1} &= 0 \\
L_0 &= 0 + 0.5 = 0.5 \\
L_1 &= 0 + 0.5 + 0.2 = 0.7 \\
L_2 &= 0 + 0.5 + 0.2 + 0.01 = 0.71 \\
L_3 &= 0 + 0.5 + 0.2 + 0.01 + 0.005 = 0.715 \\
&\vdots
\end{aligned}
$$

Diese Ergebnisse können wir mit Tabelle 2.4 vergleichen.

3.2.4 Parallelisierung

Die Parallelisierung geschieht dadurch, dass man die Multiplikation und die Addition der LRs und Ls, beziehungsweise die der HRs und Hs in einer Baumstruktur anordnet.

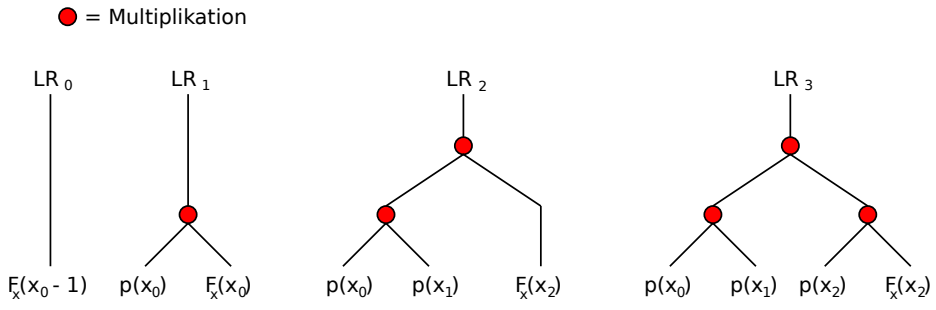

Abbildung 3.4: Berechnungs-Baum LR für die ersten 4 Symbole

Dadurch ergibt sich eine rein rechnerische Abarbeitung in

$$m = (\log_b k) + 1 \tag{3.19}$$

Schritten. Auch hier ist k die Anzahl gleichzeitig eingelesener Buchstaben.

Anders als bei [JJ94] ist hier mehr Hardware notwendig um den Kode zu berechnen. Bei dieser Struktur müssen sämtliche Folgen von $LR_{-1}, LR_0, LR_1 \ldots LR_k$ als Bäume implementiert werden um den Kode zu berechnen. Es sind also hier k Bäume notwendig. Der höchste von ihnen — der für LR_k — hat dann die Tiefe von $\log_b k$.

Parallel zur Berechnung von LR kann die Addition zu L geschehen. Nach $\log_b k$ Schritten steht LR_k fest und kann zu L_{k-1} addiert werden. Deshalb der zusätzliche Schritt in (3.19).

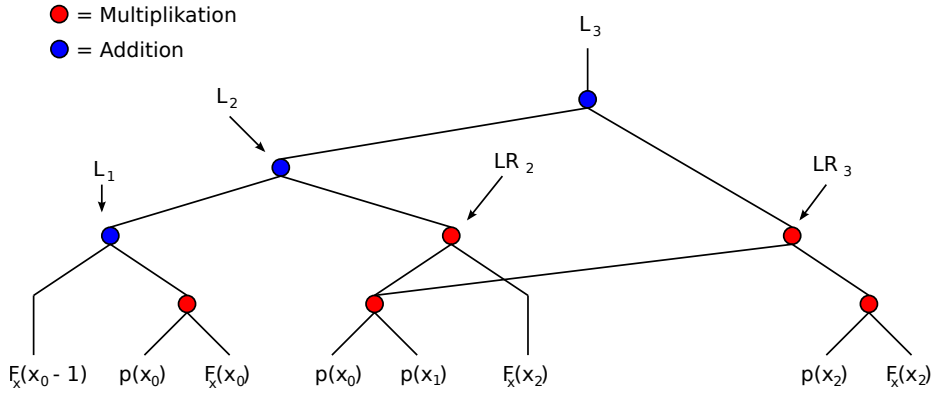

Abbildung 3.5: Berechnungs-Baum L für die ersten 4 Symbole

Allerdings wird nicht der vollständige Baum für jedes L benötigt, wie man aus Abbildung 3.5 sehen kann. Zu dem höchsten Baum braucht man zusätzlich jeweils nur die Seiten mit $F_x(i)$.

Die Autoren sind in diesem Paper nicht auf Problematik eingegangen, wie man vorgehen soll, wenn nicht unendlich viele Prozessoren zur Verfügung stehen. Es ist in den seltensten Fällen möglich einen Text komplett parallel auszurechnen. Deshalb muss man sich überlegen, wie man schrittweise vorgeht.

Ein Vorschlag wäre, einen Parallelen Block für die Kodierung von k Symbolen zu entwerfen und diesen dann Stück für Stück vorgehen zu lassen. Als Ergebnis stehen am obersten Knoten dann L_k und, als Intervall, LR_k bereit. L_k wird bei der folgenden Iteration als $F_x(x_0 - 1)$, und LR_k als $F_x(x_0)$ eingegeben. Bei den anderen Blättern werden die nächsten Symbole, beziehungsweise $p(x_i)$ eingegeben. Nach jedem Durchlauf werden also $k - 1$ Buchstaben angehängt.

Für die Normalisierung berufen wir uns wieder auf die Tatsache, dass es keine Rolle spielt, wann diese stattfindet, solange die Präzision hoch genug ist. Damit wird wie schon in Kapitel 3.1.3 besprochen die Skalierung erst nach dem Parallelen Block durchgeführt.

3.3 Parallelisierung nach Horg-Yeong Lee, Leu-Shing Lan, Ming-Hwa Sheu und Chien-Hsing Wu

3.3.1 Einleitung

Im Paper [LLSW96] aus dem Jahr 1996 zeigen die Autoren eine Variante von der Implementierung von [JJ94]. Im Wesentlichen sind hier keine großen Veränderungen oder Neuerungen hinzugekommen. Vielmehr ist es eine andere Darstellung der bekannten Formel.

3.3.2 Prinzip

Wir nehmen Gleichung (3.17) und schreiben diese bis zu L_7, also 8 Symbole, aus.

$$
\begin{aligned}
L_7 = &\sum_{k=1}^{7}(\prod_{j=0}^{k-2} p(x_j)) \cdot F_x(k-1) \\
=&F_x(0) + p(x_0) \cdot F_x(1) + p(x_0) \cdot p(x_1) \cdot F_x(2) + p(x_0) \cdot p(x_1) \cdot p(x_2) \cdot F_x(3) \\
&+ p(x_0) \cdot \ldots \cdot p(x_3) \cdot F_x(4) + p(x_0) \cdot \ldots \cdot p(x_4) \cdot F_x(5) \\
&+ p(x_0) \cdot \ldots \cdot p(x_5) \cdot F_x(6) + p(x_0) \cdot \ldots \cdot p(x_6) \cdot F_x(7) \\
=&[\underbrace{(F_x(0) + p(x_0) \cdot F_x(1))}_{\text{mul+add}} + p(x_0) \cdot p(x_1) \cdot \underbrace{(F_x(2) + p(x_2) \cdot F_x(3))}_{\text{mul+add}}] \\
&+ p(x_0) \cdot p(x_1) \cdot p(x_2) \cdot p(x_3) \\
&\cdot [\underbrace{(F_x(4) + p(x_4) \cdot F_x(5))}_{\text{mul+add}} + p(x_4) \cdot p(x_5) \cdot \underbrace{(F_x(6) + p(x_6) \cdot F_x(7))}_{\text{mul+add}}]
\end{aligned}
\tag{3.20}
$$

Die Idee ist, jetzt zwei verschiedene Einheiten zu konstruieren, die eine multipliziert zwei Zahlen, die andere, in Gleichung (3.20) als «add+mul» bezeichnet, multipliziert erst $p(x_i)$ mit $F_x(i+1)$ und addiert dann $F_x(i-1)$. Der zugehörige Baum ist in Abbildung 3.6 dargestellt.

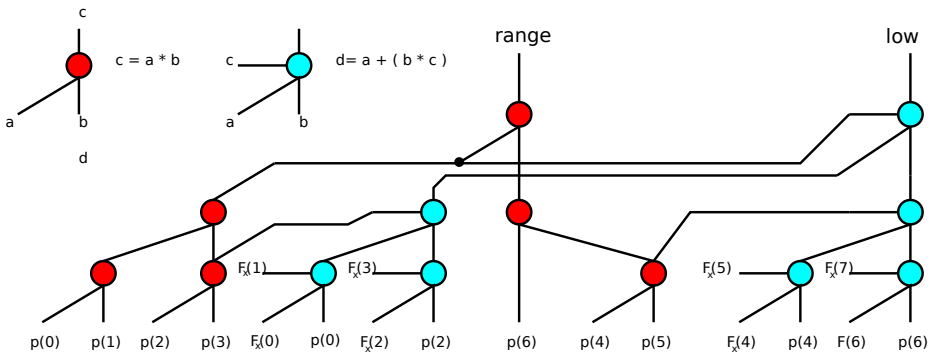

Abbildung 3.6: Berechnungs-Baum

3.4 Konklusion

Jetzt haben wir die Grundlagen für das sequenzielle Arithmetische Kodieren und die Vorschläge für eine parallele Implementierung kennengelernt. Wir sind nun in der Lage eine tiefere Analyse der vorgestellten Lösungen vorzunehmen.

Bei dieser Analyse sind mir drei Probleme besonders aufgefallen. Diese sollen im Folgenden besprochen werden und falls möglich Lösungen vorgestellt werden, wie man damit umgehen kann.

3.4.1 Gleichungen für Parallelisierung

Wir betrachten die Gleichung (3.2) und schreiben diese abkürzt als

$$r_k = r_0 \cdot \prod_{j=0}^{k} p(x_j)$$

Diese Gleichung setzen wir in Gleichung (3.3) ein

$$\begin{aligned}
L_i =& L_0 \\
&+ r_0 \cdot (\prod_{j=0}^{-1} p(x_j)) \cdot F_x(0)^3 \\
&+ r_0 \cdot (\prod_{j=0}^{0} p(x_j)) \cdot F_x(1) \\
&+ r_0 \cdot (\prod_{j=0}^{1} p(x_j)) \cdot F_x(2) \\
&+ r_0 \cdot (\prod_{j=0}^{2} p(x_j)) \cdot F_x(3) \\
&\vdots \\
&+ r_0 \cdot (\prod_{j=0}^{i-1} p(x_j)) \cdot F_x(n)
\end{aligned}$$

$$= L_0 + \sum_{k=1}^{i} [r_0 \cdot (\prod_{j=0}^{k-2} p(x_j)) \cdot F_x(k-1)]$$

Das ist aber genau die Gleichung (3.16). Mit anderen Worten J. Šupol und B. Melichar benutzen die gleiche Gleichung die J. Jiang und S. Jones in der BPE verwenden. Allerdings teilen sie die Gleichung nicht in weitere Teile. Auch [LLSW96] verwenden Gleichung (3.16). Der Unterschied liegt allein in der Kombination von Gleichung (3.16), die der BPE entspricht und (3.6) für die GPE.

Dabei haben [JJ94] und [SM05] die Extremwerte der möglichen Kombinationen gezeigt. Bei [JJ94] wurden die Blätter — also die BPEs — so weit geteilt, bis nur noch ein Symbol zu kodieren war. [SM05] ging den Weg, am Blatt selbst, das entspricht der BPE, die Parallelität durch eine Baumstruktur zu erreichen.

Man kann jetzt einen Schritt weiter gehen und die beiden Ansätze kombinieren. Es wäre zum Beispiel möglich, BPEs nach Gleichung (3.16) zu implementieren und dann die Baumstruktur in Abbildung 3.3 weiter zu nutzen .

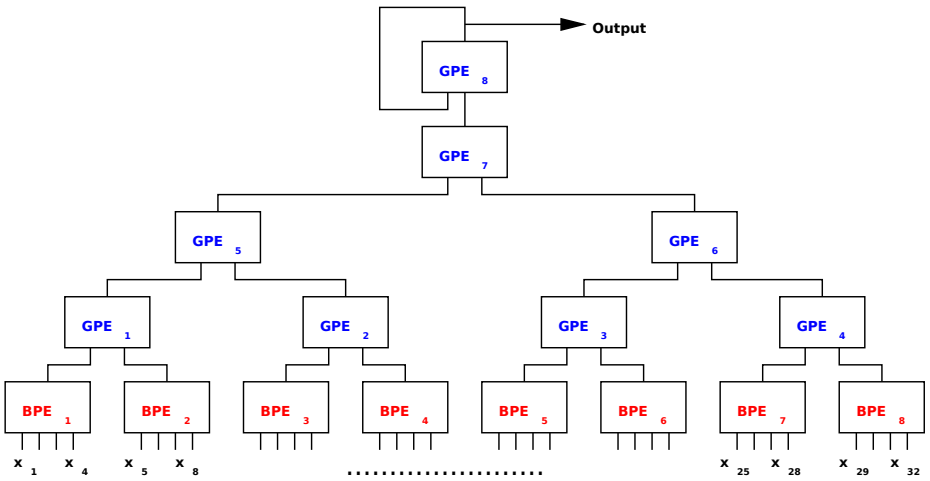

Abbildung 3.7: Berechnungs-Baum für 32 Symbole

Abbildung 3.7 zeigt ein Beispiel bei dem die Blätter nach Gleichung (3.16) jeweils vierfach parallel gerechnet werden. Die Ergebnisse werden dann mit Gleichung (3.6) zum Gesamtergebnis zusammengefügt.

Der Vorteil dabei ist, dass für Gleichung (3.16) eine andere Architektur verwendet werden kann als für Gleichung (3.6). Man könnte für die niedrigere Präzision (BPE) vorhandene Hardware, zum Beispiel CUDA, und für die hohe Präzision (GPE) dedizierte Hardware implementieren.

[3] siehe Gleichung (3.15)

3.4.2 Präzision

Das größte Problem bei der Arithmetischen Kodierung ist die notwendige Präzision. Wir haben gesehen, dass bisher noch keine wirkliche Lösung vorgeschlagen worden ist. Die in [JJ94] vorgeschlagene Skalierung nach jedem Schritt hilft nicht weiter. Trotz der Skalierung muss die Präzision bei der nächsten Stufe in vollem Umfang eingerechnet werden. Die anderen Papers sagen gar nichts zu der Problematik.

In Kapitel 3.1.3 wurde schon eine Variante vorgestellt, möglichst ohne viel zusätzlichen Hardware Aufwand das Problem in den Griff zu bekommen. Wir konzentrieren uns deshalb an dieser Stelle auf die Frage der Präzision, die notwendig ist, um diesem Vorschlag nachzugehen.

Für eine praktische Implementierung wird das Intervall gemäß Kapitel 2.6.1 auf eine Ganze Zahl abgebildet. Das Intervall von 0 bis zur höchstmöglichen Zahl nennen wir r_0. Die kleinste Einteilung ist damit $1/r_0$ und ist zugleich die höchstmögliche Präzision. Es soll nun im Folgenden darum gehen, die Bit Breite für r_0 zu bestimmen.

Wir nennen nun $\texttt{total} = \texttt{tot}$ und $\texttt{high\_count}_{a_i} = \texttt{f}_i$. Dann ist $p(a_i) = \frac{\texttt{f}_i}{\texttt{tot}}$. Bemerkenswert ist an dieser Stelle, dass tot nicht unbedingt die Länge des Textes ist, sondern der gekürzte Bruch von $\sum \texttt{f}_i/\texttt{Textlänge}$.

Beispiel 10. Sei $A = \{a, b, c, d\}$ und $S = aaaabbcd$. Dann gilt $\texttt{f}_1 = 4$, $\texttt{f}_2 = 2$, $\texttt{f}_3 = 1$ und $\texttt{f}_4 = 4$. Weiter gilt $\texttt{tot} = 8$. Dann ist hier $p(a_1) = 4/8 = 0.5$.

Das würde aber auch gelten, wenn $S = aaaabbcdaaaabbcd$. Die Reihenfolge spielt dabei keine Rolle. Lediglich das Verhältnis. $\Diamond$

Dann ist $1/\texttt{tot}$ die kleinste vorkommende Wahrscheinlichkeit. Um eindeutig Kodieren und Dekodieren zu können muss also

$$r_i \geq \texttt{tot} \quad \forall i \in n$$

Als nächstes nehmen wir die Formeln für die GPE und BPE.

Für die Gleichung (3.9) gilt

$$\begin{aligned} r_1 &= r_0 \cdot p(x_0) \\ &= r_0 \cdot \frac{\texttt{f}_i}{\texttt{tot}} \end{aligned}$$

Falls aber im schlechtesten Fall $\mathtt{f}_i = 1$ und damit die kleinste Wahrscheinlichkeit übergeben wird, dann muss $r_1 \geq \mathtt{tot}$. Also können wir zeigen:

$$\begin{aligned} r_1 &\geq \mathtt{tot} \\ r_0 \cdot \frac{\mathtt{f}_i}{\mathtt{tot}} &\geq \mathtt{tot} \\ r_0 \cdot \frac{1}{\mathtt{tot}} &\geq \mathtt{tot} \\ \frac{r_0}{\mathtt{tot}} &\geq \mathtt{tot} \\ r_0 &\geq \mathtt{tot}^2 \end{aligned} \tag{3.21}$$

Für Gleichung (3.8) ergibt sich für L keine direkte Einschränkung, weil für L mindestens L_0 als untere Schranke behalten wird. Trotzdem gilt

$$\begin{aligned} \log_b L &< \log_b r_0 \\ \log_b L &\leq (\log_b r_0) - 1 \end{aligned}$$

Im besten Fall ist L ein Bit kürzer als r. Hat man die Bit Breite von r kann man diese als obere Grenze für L nehmen und als untere Grenze $(\log_b r) - 1$. In der Praxis wird man Register der selben Breite implementieren und damit $\log_b L = \log_b r$ setzen.

Betrachten wir nun die Gleichung (3.7)

$$\begin{aligned} r_{i+1} = \frac{r_1 \cdot r_2}{r_0} &\geq \mathtt{tot} \\ \frac{r_1 \cdot r_2}{\mathtt{tot}^2} &\geq \mathtt{tot} \quad | \quad \text{wegen Gleichung (3.21)} \\ \frac{r_1 \cdot 1}{\mathtt{tot}^2} &\geq \mathtt{tot} \quad | \quad \text{schlechtester Fall } \mathtt{f}_i = 1 \\ r_1 &\geq \mathtt{tot}^3 \quad | \quad \mathtt{tot} > 0 \end{aligned}$$

Wir sehen also, dass nach jeder Iteration die Präzision von r um $1/\mathtt{tot}$ steigt. Sei p die Anzahl paralleler Stufen, dann gilt für ein Binäres System

$$r_i \geq \mathtt{tot}^p$$

und damit für die Bit Breite von r_i

$$\begin{aligned} \log_b r_i &\geq \log_b \mathtt{tot}^p \\ &\geq p \cdot \log_b \mathtt{tot} \end{aligned}$$

Bit Breite für r 1. Sei p die Anzahl der parallelen Stufen in einem Baum, dann ist die mindest erforderliche Bit Breite für alle r

$$\boxed{\log_b r_i \geq p \cdot \log_b \mathtt{tot} \quad \forall i \in n} \tag{3.22}$$

Analog kann das auch für die Gleichung (3.2) gezeigt werden.

Wir haben in Kapitel 3.1.4 schon die Problematik der extrem hohen Wahrscheinlichkeiten angesprochen. Ich will an dieser Stelle ein paar Überlegungen dazu skizzierenÖ

Sei $1/r_0 = p_{min}$, $p_{max} = 1 - p_{min}$ und σ die Anzahl der Schritte, dann wird spätestens nach

$$\begin{aligned} 0.5 &= p_{max}^{\sigma} \\ \log 0.5 &= \log p_{max}^{\sigma} \\ \log 0.5 &= \sigma \cdot \log p_{max} \\ \sigma &= \lfloor \log 0.5 / \log p_{max} \rfloor \end{aligned}$$

Schritten skaliert. In einer parallelen Abarbeitung also erst nach

$$\sigma_{par} = \lfloor \sigma / p \rfloor$$

Beispiel 11. Gegeben sei $r_0 = 8$, $p_{min} = 1/8 \Rightarrow p_max = 7/8$ und $p = 4$. Dann ist

$$\begin{aligned} \sigma &= \lfloor \log 0.5 / \log 7/8 \rfloor \\ &= \lfloor 5.19 \rfloor \\ &= 6 \end{aligned}$$

und

$$\begin{aligned} \sigma_{par} &= \lfloor 6/p \rfloor \\ &= \lfloor 1.5 \rfloor = 2 \end{aligned}$$

$\Diamond$

Im Beispiel 11 sehen wir, dass der parallele Block zweimal durchläuft, ehe skaliert wird. Das könnte bei zu niedriger Bit Breite von r zu einem Unterlauf führen. Eine genauere Analyse wäre hier interessant, um herauszufinden ab welcher Rundung das Dekodieren fehl schlägt. Mit größerem `tot` steigt auch die erforderliche Bit Breite. Es ist denkbar, dass dadurch die Rundungsfehler so klein gehalten werden, dass ein Dekodieren immer möglich bleibt.

Ich überlasse diese Problematik an dieser Stelle dem interessierten Leser und breche hier ab.

3.4.3 Algorithmus für Skalierung

Eine weitere Erkenntnis können wir bei näherer Betrachtung des Skalierens gewinnen. In den Kapiteln 2.6.2 und 2.6.3 wurden die Grundlagen und Notwendigkeit der Skalierung beschrieben. Hierzu wurden die Grenzen L und H benutzt. Die Verwendung von r statt H vereinfacht die Skalierung drastisch. Die Einteilung in verschiedene Quadranten kann gänzlich wegfallen. Wir betrachten allein die Größe von r. Ist $r \leq 1/2 \cdot r_0$, dann kann skaliert werden.

Das Skalieren verläuft analog des beschriebenen Verfahrens. L und r werden nach links geschoben und das höchstwertigste, rauslaufende Bit von L als Kode gespeichert. Allerdings muss man einen Sonderfall betrachten: Falls $L + r > r_0$ nach einer Skalierung gelten würde, dann darf nicht skaliert werden. Der Algorithmus hierfür ist in Algorithmus 3.1 beschrieben.

Algorithmus 3.1 Skalierung bei Verwendung von r

```
while r ≤ (r0/2) do
    r_temp ← r · 2
    L_temp ← L · 2
    if L_temp > r0 then
        L_temp ← L_temp − r0
        if (r_temp + L_temp) < r0 then
            Kode ← 1
            L ← L_temp
            r ← r_temp
        else
            exit
        end if
    else
        if (r_temp + L_temp) < r0 then
            Kode ← 0
            L ← L_temp
            r ← r_temp
        else
            exit
        end if
    end if
    r ← r_temp
    L ← L_temp
end while
```

Den Kode kann man auch direkter bestimmen. Man sucht die signifikanteste «1» (rot) in r. Diese Position bezeichnen wir mit j. Von dieser Bit-Position sucht man die nächst höherwertigere Bit-Position i (gelb) in L, die erstmals den Wert «0» hat, wobei $i > j$ sein muß. Die MSB-Bits in $L_{15\ldots(i+1)}$, unabhängig vom Wert, entsprechen dem Kode.

Beispiel 12. r_0 und L haben eine Bit Breite von 16 Bits. Wir erhalten nach einem parallelen Durchlauf die Werte $L = \texttt{0A10}_\text{h}$ und $r = \texttt{0090}_\text{h}$. Dann ist der herauszugebende Kode $\texttt{000101}_\text{b}$. Siehe Abbildung 3.8

Falls wir die Werte $L = \texttt{7836}_\text{h}$ und $r = \texttt{0016}_\text{h}$ annehmen, ergibt sich der Kode $\texttt{11110000}_\text{b}$. Siehe hierzu Abbildung 3.9. ◊

Das Finden der Position i entspricht der Bedingung, dass sowohl $L < r_0$ als auch $r + L \leq r_0$ gelten muss. Wie bei der E3 Skalierung muss man warten, bis die zu kodierende Stelle bei

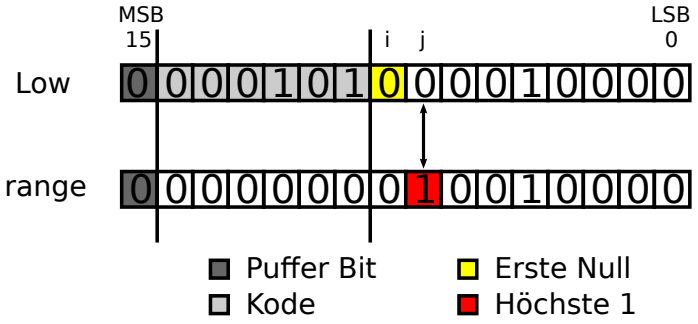

Abbildung 3.8: direkte Bestimmung des Kodes für $L = 0A10$ und $r = 0090$

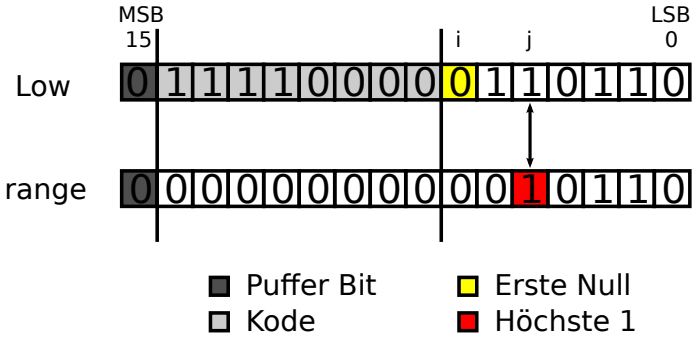

Abbildung 3.9: direkte Bestimmung des Kodes für $L = 7836$ und $r = 0016$

L stabil anliegt. Beim sequentiellen Vorgehen geschieht das, indem man erst dann Kode Bits herausgibt, wenn man sicher in der unteren oder oberen Hälfte des Intervalls liegt. Im Algorithmus 3.1 werden als Kode die Bits herausgegeben, die durch die Addition von r, L nicht mehr beeinflussen können.

Das ist genau dann der Fall, wenn man solange bei L in Richtung most significant Bit geht, bis die erste Null auftritt. An dieser Stelle steht sowohl bei L als auch bei r eine Null. Bei einer Addition wird ab hier L nicht mehr beeinflusst, weil in r in Richtung most significant Bit, nur noch Nullen folgen.

Man kann also nach dieser Methode die maximale Anzahl an Schleifendurchläufe l auf $2 \cdot \log_b l$ bei geeigneter Implementierung reduzieren.

Beispiel 13. Gegeben sei die Bit Breite $\log_b r = 8$. Wir bilden erst einen Prioritätsenkoder mit einer Maske. Wenn also $x_m = 1$ dann gilt $x_i = 1, i < m$. Der Baum aus `OR` Gattern ist bis zum vierten Bit in Abbildung 3.10 dargestellt. Nachdem die Maske in y fertig ist, können wir mit `XOR`s das höchste Bit bestimmen. Falls der Übergang von 0 auf 1 an der Stelle (y_4, y_3) ist, dann ist z_3 gesetzt. $\Diamond$

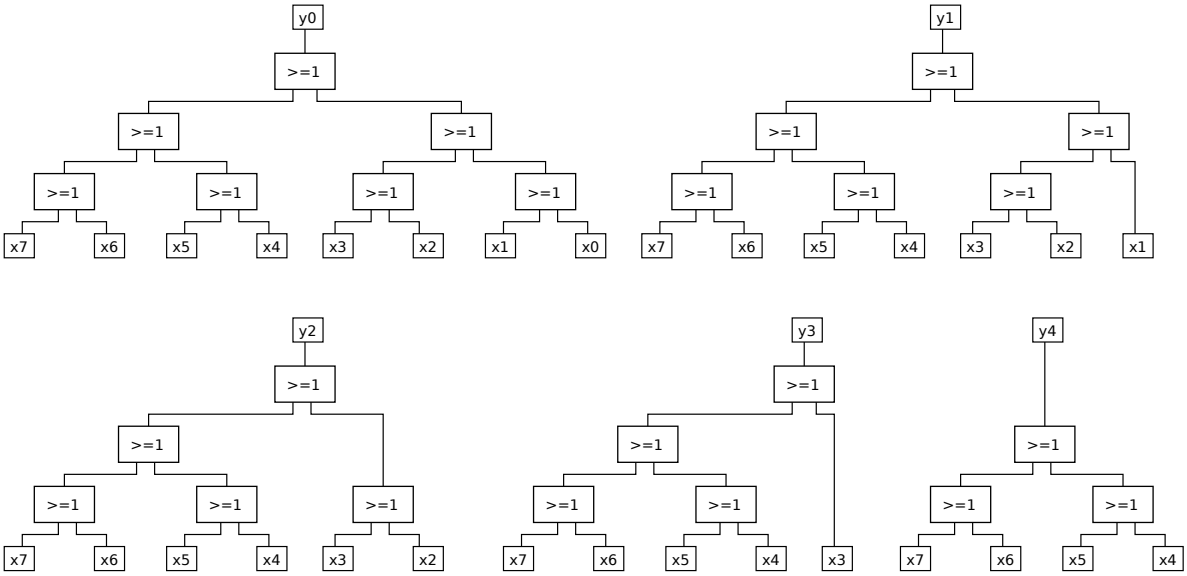

Abbildung 3.10: Prioritätsenkoder `OR`-Baum

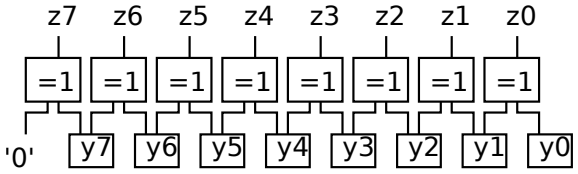

Abbildung 3.11: Bitbestimmung mit `XOR` Gattern

3.5 Parallelisierung in Hardware

Nach der Analyse soll nun als weiterer Bestandteil dieser Arbeit eine Realisierung in Hardware erfolgen. Die Architektur ist nicht vorgegeben. Grundsätzlich besteht jetzt die Möglichkeit, vorhandene Hardware zu verwenden, oder eine eigens für den Zweck zu entwerfen. Bei der vorhandenen Hardware sind besonders die Grafikkarten interessant. Diese bieten viel mehr parallele Prozessoren als eine herkömmliche CPU. Wir beschränken uns deshalb auf die Architektur einer Grafikkarte. Es wäre natürlich denkbar, auch Großrechner mit vielen CPUs anzuschauen, aber wie wir sehen werden, treten hier dieselben Probleme wie bei der Implementierung auf einer GPU. Für die dedizierte Hardware schauen wir uns eine Implementierung in VHDL an.

3.5.1 CUDA

Für eine Implementierung auf vorhandener Hardware nehmen wir eine GPU. Als Beispiel benutzen wir die von Nvidia™ 2007 eingeführte Compute Unified Device Architecture kurz: CUDA.

Architektur

Die CUDA Architektur teilt sich auf in Symetric Multiprocessors (SMs), Blöcke und Threads. Jede SM kann maximal aus 1024 Threads [4] bestehen, und aus maximal 8 Blöcken. Die Zuweisung der Threads zu den Blöcken geschieht flexibel: Zum Beispiel 8 Blöcke zu je 128 Threads, oder 4 Blöcken je 256 Threads. Für eine GT200 die 30 SMs zu je 1024 Threads hat, besteht so die Möglichkeit $30 \cdot 1024 = 30720$ Threads gleichzeitig abzuarbeiten.

Jeder Thread bearbeitet eine Berechnung. Die Threads werden durch einen Aufruf eines sogenannten Kernel gestartet. Davor müssen die Daten in den Kartenspeicher gebracht werden. Sind die Threads zu groß, können sie nicht in den Kartenspeicher gebracht werden und damit nicht parallel gerechnet werden. Das geschieht zwar für den Nutzer transparent, aber ohne Beachtung dieser Beschränkung geht man Geschwindigkeitsverluste ein. Threads können intern synchronisiert werden. Auch hierdurch entstehen Geschwindigkeitsverluste. Bei `IF THEN ELSE` Anweisungen geht der Kernel immer den `IF`, dann den `ELSE` Zweig durch und entscheidet anschließend welcher ausgeführt wird. Das heißt, bei einer Abfrage von acht verschiedenen Abzweigungen wird der Thread acht mal durchlaufen, obwohl nur ein Zweig real berechnet werden müsste. Die Threads werden in Gruppen von 32 geteilt. Diese Gruppen nennen sich Warps. Findet in einem solchen Warp eine `IF THEN ELSE` Abfrage statt, warten alle Threads in diesem Warp bis beide Zweige durchlaufen sind.

Die Bit Breite der Fließkomma Operationen ist bis zur G200 32 Bit. Bei der Ganzzahl-Arithmetik sind es 64 Bit. Ab der compute capability 1.3 sind 64 Bit Berechnungen auch für Fließkomma Arithmetik möglich.

Implementierung

Durch die vielen Einschränkungen ist eine Verwendung solcher Architekturen relativ schwierig. Die zwei Hauptprobleme sind die Geschwindigkeitsverluste beim Kernel Aufruf und vor allem die Bit Breite der Register.

Vor jedem Kernel Aufruf müssen die Daten in den Grafikspeicher kopiert werden. Die Ergebnisse werden in den Grafikspeicher geschrieben und müssen nach der Berechnung wieder in den Hauptspeicher kopiert werden. Dieser Prozess ist sehr langsam. Mit Einführung der Compute capability 2.0 kann die GPU vereinfacht auf den Hauptspeicher zugreifen. Die schnellste Bearbeitung bleibt aber die auf dem eigenen Speicher. Nach jeder Parallelen

[4] compute capability 1.2

Stufe müssten die Ergebnisse hin- und herkopiert werden um dann die nächste Stufe mit den vorher berechneten Ergebnissen zu starten. Mit etwas mehr Aufwand könnte man das Problem umgehen, indem man maximal so viel Parallelität implementiert, so dass sie komplett in den Grafikspeicher hinein passt.

Selbst dann würde die vorhandene Bit Breite die Parallelität einschränken. In einem deutschen Text tritt der Buchstabe x mit einer Wahrscheinlichkeit von 0.0003 auf [5]. Um diese darzustellen braucht man $\texttt{tot} = 1/0.0003 = 3333.\overline{3}$. Im Binärsystem ergibt das

$$\begin{aligned} 2^x &= 3333.\overline{3} \\ \log(2^x) &= \log 3333.\overline{3} \\ x &= \lfloor \log 3333.\overline{3} / \log 2 \rfloor \\ x &= 12 = \log_b \texttt{tot} \end{aligned}$$

Bits. Und gemäß Gleichung (3.22) können bei einer Bit Breite von $r_i = 64$ Bits nur

$$\begin{aligned} \log_b r_i &\geq p \cdot \log_b \texttt{tot} \\ 64 &\geq p \cdot \log_b \texttt{tot} \\ 64 &\geq p \cdot 12 \\ \lceil 64/12 \rceil &\geq p \\ 5 &\geq p \end{aligned}$$

Parallele Stufen ohne fehlerhafte Dekodierung realisiert werden. Das heißt, trotz der vielen Threads können nur $2^5 = 32$ gleichzeitig berechnet werden. Diese starke Einschränkung macht diese Technologie ungeeignet für massive parallele Kodierungen.

Wie man diese Architektur trotzdem zum Vorteil nutzen kann wurde in Kapitel 3.4.1 vorgestellt.

3.5.2 VHDL

Der Implementierung in dedizierter Hardware sind weit weniger Beschränkungen unterlegen als zum Beispiel in CUDA. Die Bit Breiten sind hier frei wählbar und somit steigt auch der Grad der möglichen Parallelität. Trotz der geringen Beschränkungen von architektonischer Seite ergeben sich aus Sicht der Algorithmen gewisse Einschränkungen die nicht unerheblich sind.

Mit jeder parallelen Stufe kommen $\log_b \texttt{tot}$ Bits zu der Bit Breite hinzu. Bei einer anfänglichen Bit Breite von $\log_b r_i = 16$ ergeben sich nach 8 Stufen eine Bit Breite von $8 * 16 = 256$ Bit. Das Ergebnis einer Solchen Multiplikation nimmt 512 Bits in Anspruch. Die Berechnung von solch großer Bit Breiten sind sehr komplex und langsam. Der Aufwand steigt bei

[5] aus [Beu93]

Multiplikationen quadratisch an. Es ist also fraglich, ob der Gewinn an paralleler Abarbeitung die Geschwindigkeit des Multiplizierers kompensieren kann. Wenn nicht, wäre auch hier eine Analyse interessant, ab welcher Latenz des Multiplizierers eine zusätzliche Parallele Stufe keine Geschwindigkeitssteigerung bringt. Bei exakter Berechnung von $p(i)$ ist sogar noch ein Dividierer notwendig. Die Hardware dazu ist komplexer als die eines Multiplizierers und erschwert einen Geschwindigkeits-gewinn zusätzlich.

Es ist aus diesen Gründen ratsam, im zugrundeliegenden Modell, für `tot` Zweierpotenzen zu wählen und dabei eine etwas schlechtere Komprimierung in Kauf zu nehmen. Auf diese Weise spart man sich einen Dividierer weil man leicht durch Zweierpotenzen mit Verschiebe-Operationen teilen kann. Um wieviel schlechter man durch solche Rundungen wird, wäre Teil einer weiteren Analyse.

Algorithmus

Als nächstes widmen wir uns der Frage, welche vor und Nachteile die besprochenen Vorschläge haben.

J. Šupol und B. Melichar Der Algorithmus von [SM05] hat die geringste Tiefe. Anders als bei [JJ94], oder [LLSW96] müssen die Knoten innerhalb des Baumes lediglich eine Operation durchführen. Bei den anderen Vorschlägen sind mindestens zwei Operationen notwendig. Dadurch entstehen weitere Verzögerungen, weil zuerst auf das Ergebnis der vorangegangenen Operation gewartet werden muss. Auf Register Transfer Level (RTL) ist deshalb diese Lösung die schnellste. Sie lässt sich aber nicht so gut erweitern, weil immer ein Teil-Baum der höchsten parallelen Stufe mehrfach verwendet wird. Nur mit dieser Mehrfach-Verwendung sind große Einsparungen von Hardware möglich.

J. Jiang und S. Jones Bei dieser Lösung steht die Modularisierung im Vordergrund. Sie beinhaltet beide Differenzengleichungspaare für L und r. Die Gleichung (3.6) hat zwei Operationen. Zuerst die Multiplikation, anschließend die Addition. Wie schon oben beschrieben entsteht damit eine Verzögerung auf RTL Ebene. Die starke Modularisierung erlaubt durch einfaches Hinzufügen von `BPE`s und `GPE`s die Parallelität zu erhöhen.

Horg-Yeong Lee, Leu-Shing Lan, Ming-Hwa Sheu und Chien-Hsing Wu Diese Variante bringt zu den oben genannten keine wesentliche Verbesserung. Die Knoten im Baum haben wie in [JJ94] zwei Operationen. Das führt auf RTL Ebene zu zusätzlichen Takten. Die Erweiterbarkeit ist ähnlich komplex wie in [SM05] aber nicht so einfach wie in [JJ94]

Die Entscheidung fiel auf die Variante von [JJ94]. Durch die klar gegliederte Modularisierung lässt sich der Entwurf ohne großen Aufwand erweitern. Anders als bei [SM05], kommen beide herausgearbeiteten Gleichungspaare (3.6), (3.7) und (3.13), (3.14) zur Anwendung. Schließlich kann diese Architektur als einzige mit vorhandener Hardware kombiniert werden, um die in Kapitel 3.4.1 vorgestellte Lösung zu realisieren.

4 Implementierung

Wir haben die Grundlagen des Arithmetischen Kodierens im Kapitel 2 kennengelernt. Diese wurden im Kapitel 3 durch eine Analyse von Lösungsvorschlägen zur Parallelisierung in Hardware vertieft. Mit diesem Wissen sind wir in der Lage, eine reale Implementierung anzugehen.

Mir der Umsetzung in Hardware sollen die gewonnenen Kenntnisse auf Korrektheit überprüft und gleichzeitig die Machbarkeit gezeigt werden. Im Vordergrund steht, die Theorie in einem leicht verständlichen Beispiel umzusetzen. Ressourcen- und Laufzeitoptimierung sind hier nachrangig. Dennoch ist der Kode so gestaltet, dass eine Erweiterung ohne größeren Aufwand möglich ist.

4.1 Referenz Beispiel

Für die Implementierung verwenden wir das in Kapitel 2.6.2 vorgestellte Beispiel. Der Vorteil dieses Beispiels ist die geringe Komplexität und die Abdeckung aller möglichen Pfade. Damit können wir sämtliche Teile der Hardware auf Korrektheit überprüfen.

Wir schauen uns Beispiel 8 nochmal an. Hier wurde beispielhaft eine Bit Breite von 8 genommen. Das ist für ein sequenzielles Vorgehen ausreichend. Für eine parallele Abarbeitung müssen wir die Register verbreitern.

Das Alphabet des Beispiels besteht aus acht Buchstaben $|A| = 8$. Die kleinste auftretende Wahrscheinlichkeit ist $p_{min} = 1/\texttt{tot}$. Mit $\texttt{tot} = 8$ ist $p_{min} = 1/8$. Um `tot` im Dualen System darzustellen sind $\log_b \texttt{tot} = 3$ Bits notwendig. Weil sich 4 Bits leichter in Hexadezimal darstellen lassen, verwenden wir diese für unsere Implementierung, obwohl wir nur 3 brauchen. Dabei generieren wir hohe Redundanzen, aber wie schon eingangs erwähnt, soll es hier nicht um eine optimale Realisierung gehen.

Um die Struktur nicht unnötig komplex werden zu lassen, wählen wir eine Parallelität von 4. Das heißt, 4 Buchstaben sollen gleichzeitig eingelesen und anschließend kodiert werden. Wir setzen $p = 4$ und erhalten nach Gleichung (3.22)

$$\begin{aligned}
\log_b r_i &\geq p \cdot \log_b \texttt{tot} \\
&\geq 4 \cdot \log_b \texttt{tot} \\
&\geq 4 \cdot 4 \quad \text{wir nehmen 4 statt 3 (siehe oben)} \\
&\geq 16
\end{aligned}$$

Wir brauchen also für r und L eine Bit Breite von 16 Bits.

Die zu kodierende Sequenz ist $S = abccedac$. Die Baum-Struktur für eine vierfache Parallelisierung ist in Abbildung 4.1 dargestellt. Wir benutzen fortan für das Beispiel das hexadezimale System, ansonsten geben wir die Basis explizit an.

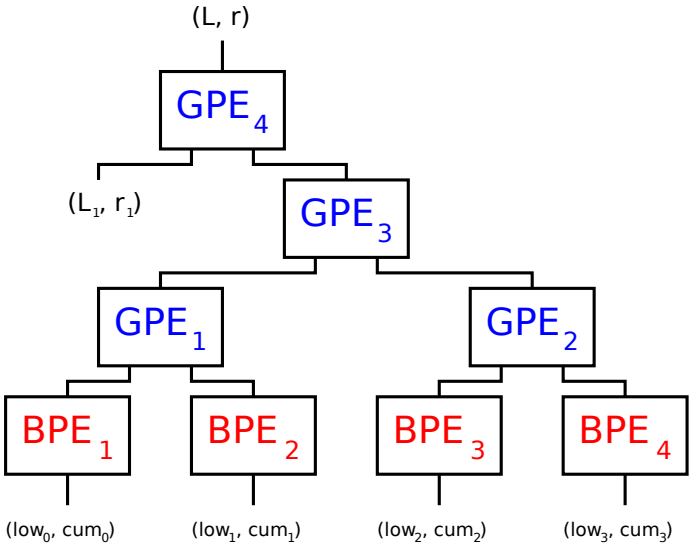

Abbildung 4.1: Berechnungs-Baum für 4 Symbole + Konkatenation

Um besser rechnen zu können, nutzen wir nicht den vollen Bereich von $r = FFFF$ aus und schränken r auf 8000 ein. De facto stehen uns nur 15 Bits zur Verfügung. Das höchstwertigste 16. Bit wird nur dann gesetzt, wenn der volle Bereich benutzt wird. Es dient auch als Puffer für Berechnungen, die über den Bereich von r_0 hinausgehen. Dies tritt bei der Berechnung des Kodes im Algorithmus 3.1 auf. In diesem Algorithmus kann L den Wert $L = 2(r_0 - 1)$ annehmen. L und r werden also mit $L = 0$ und $r = 8000$ initialisiert. Das gilt im Übrigen auch für die $\texttt{GPE}_4$. Hier muss auf der linken Seite $L_1 = 0$ und $r_1 = 8000$ initialisiert werden.

Als Ergebniss der BPEs erhalten wir

$\texttt{BPE}_1$

$$\begin{aligned} L &= r_0 \cdot F_x(x_0 - 1 = a_1 - 1 = 0) \\ &= 8000 \cdot \texttt{low\_count}_a/\texttt{tot} \\ &= 8000 \cdot 0/8 \\ &= 0000 \end{aligned}$$

$$\begin{aligned} r &= r_0 \cdot p(x_0 = a_1 = 0)/\texttt{tot} \\ &= 8000 \cdot \texttt{tot\_count}_a/\texttt{tot} \\ &= 8000 \cdot 2/8 \\ &= 2000 \end{aligned}$$

$\texttt{BPE}_2$

$$\begin{aligned} L &= r_0 \cdot F_x(x_1 - 1 = a_2 - 1 = 1) \\ &= 8000 \cdot \texttt{low count}_b/\texttt{tot} \\ &= 8000 \cdot 2/8 \\ &= 2000 \end{aligned}$$

$$\begin{aligned} r &= r_0 \cdot p(x_1 = a_2 = 1)/\texttt{tot} \\ &= 8000 \cdot \texttt{tot count}_b/\texttt{tot} \\ &= 8000 \cdot 1/8 \\ &= 1000 \end{aligned}$$

$\texttt{BPE}_3$

$$\begin{aligned} L &= r_0 \cdot F_x(x_2 - 1 = a_3 - 1 = 2) \\ &= 8000 \cdot \texttt{low count}_c/\texttt{tot} \\ &= 8000 \cdot 3/8 \\ &= 3000 \end{aligned}$$

$$\begin{aligned} r &= r_0 \cdot p(x_2 = a_3 = 3)/\texttt{tot} \\ &= 8000 \cdot \texttt{tot count}_c/\texttt{tot} \\ &= 8000 \cdot 3/8 \\ &= 3000 \end{aligned}$$

$\texttt{BPE}_4$

$$\begin{aligned} L &= r_0 \cdot F_x(x_2 - 1 = a_3 - 1 = 2) \\ &= 8000 \cdot \texttt{low count}_c/\texttt{tot} \\ &= 8000 \cdot 3/8 \\ &= 3000 \end{aligned}$$

$$\begin{aligned} r &= r_0 \cdot p(x_2 = a_3 = 3)/\texttt{tot} \\ &= 8000 \cdot \texttt{tot count}_c/\texttt{tot} \\ &= 8000 \cdot 3/8 \\ &= 3000 \end{aligned}$$

Und für die GPEs entsprechend

$\texttt{GPE}_1$

$$\begin{aligned} L &= L_1 + r_1 \cdot L_2 / r_0 \\ &= 0 + 2000 \cdot 2000 / 8000 \\ &= 0 + 4000000 / 8000 \\ &= 800 \end{aligned}$$

$$\begin{aligned} r &= r_1 \cdot r_2 / r_0 \\ &= 2000 \cdot 1000 / 8000 \\ &= 400 \end{aligned}$$

$\texttt{GPE}_2$

$$\begin{aligned} L &= L_1 + r_1 \cdot L_2 / r_0 \\ &= 3000 + 3000 \cdot 3000 / 8000 \\ &= 3000 + 9000000 / 8000 \\ &= 4200 \end{aligned}$$

$$\begin{aligned} r &= r_1 \cdot r_2 / r_0 \\ &= 3000 \cdot 3000 / 8000 \\ &= 1200 \end{aligned}$$

$\texttt{GPE}_3$

$$\begin{aligned} L &= L_1 + r_1 \cdot L_2 / r_0 \\ &= 800 + 400 \cdot 4200 / 8000 \\ &= 3000 + 1080000 / 8000 \\ &= A10 \end{aligned}$$

$$\begin{aligned} r &= r_1 \cdot r_2 / r_0 \\ &= 400 \cdot 1200 / 8000 \\ &= 90 \end{aligned}$$

$\texttt{GPE}_4$

$$\begin{aligned} L &= L_1 + r_1 \cdot L_2 / r_0 \\ &= 0 + 8000 \cdot A10 / 8000 \\ &= A10 \end{aligned}$$

$$\begin{aligned} r &= r_1 \cdot r_2 / r_0 \\ &= 8000 \cdot 90 / 8000 \\ &= 90 \end{aligned}$$

L	r	$L \cdot 2$	$r \cdot 2$	L_{next}	r_{next}	output
$0A10$	0090	1420	0120	1420	0120	0
1420	0120	2840	0240	2840	0240	0
2840	0240	5080	0480	5080	0480	0
5080	0480	$A100$	0900	2100	0900	1
2100	0900	4200	1200	4200	1200	0
4200	1200	8400	2400	0400	2400	1
0400	2400	0800	4800	0800	4800	0
			> 4000			

Tabelle 4.1: Normalisierung

Die Werte $L = A10$ und $r = 90$ werden jetzt mit Algorithmus 3.1 normalisiert.

Für r_{temp} erhalten wir $r_{temp} = r \cdot 90 = 120$ und für $L_{temp} = A10 \cdot 2 = 1420$. L_{temp} und $r_{temp} + L_{temp}$ sind kleiner als r_0. Als Kode wird somit eine 0 ausgegeben und die Werte von L_{temp} beziehungsweise von r_{temp} in L und r übernommen. Die zweite und dritte Iteration verlaufen analog.

Bei der dritten Iteration ist $L_{temp} = 5080 \cdot 2 = A100$ und damit größer als r_0. L_{temp} wird angepasst, indem wir $r_0 = 8000$ davon abziehen. Als Kode ergibt sich eine 1. Alles andere verläuft wieder gleich. Hier sehen wir, warum nicht $FFFF$ als r_0 gewählt wurde. L_{temp} kann hier nur den Wert $8000 - 1$ annehmen und passt deshalb bei Verdoppelung noch in $FFFF$ hinein ($7FFF \cdot 2 = FFFE$). Würden wir den Bereich noch um 1 erweitern, wären für die Berechnung der Normalisierung größere Register notwendig.

Die vollständige Normalisierung von $L = 0A10$ und $r = 0090$ ist in Tabelle 4.1 dargestellt. Nach den ersten vier Buchstaben *abcc* ist also ein Kode von 0001010 ausgegeben, in L steht 0800 und in r steht 4800.

4.2 Struktur

Der Kodierer wird in 6 Module unterteilt: Incrementer, RAM, Parallel Loader, Parallel Koder, Normalizer und Control Unit. Dadurch ist es einfacher, den Entwurf zu erweitern und zu validieren. Sämtliche Module sind als Moore-Zustandsautomaten implementiert.

Die Kodierung beginnt damit, dass Incrementer und RAM vier Symbole bereitstellen. Der Parallel Loader wandelt diese anhand des implementierten Modells in Wahrscheinlichkeiten für den Koder um. Der Parallel Koder kodiert diese und gibt das Ergebnis als L und r an den Normalizer weiter. Dieser normalisiert L und r. Der daraus entstandene Datenstrom wird in einen im Normalizer implementierten RAM geschrieben. Damit ist ein Durchlauf beendet.

Dieser Prozess läuft so lange, bis das `terminate` Signal vom Incrementer gesetzt wird.

Die Abbildung 4.2 auf Seite 79 zeigt die angeschlossenen Module als Block-Schaltbild im Gesamtentwurf.

4.3 Incrementer

Der Incrementer hat zwei Funktionen: zum Einen die Aktuelle Adresse für das RAM zu liefern, zum Anderen das `terminate` Signal zu geben. Die Adresse wird entweder bei Setzen des `inc` Signals hochgezählt oder aber mit Setzen von `load_addr` mit der Adresse `addr_in` geladen.

Anders aber als im Synchron Zähler SN74161 darf nicht bei jeder steigenden Taktflanke gezählt werden. Er soll die Adresse einmal steigern, dann erst wieder auf erneutem Kommando. Das wird mit dem `en` Signal gegeben. Sind `inc` und `en` gesetzt, wird die Adresse um 1 inkrementiert. Um erneut zu inkrementieren muss das `en` Signal erst auf 0, dann wieder auf 1 gesetzt werden.

Das `terminate` Signal gibt an, dass die letzte Adresse anliegt und das Modul nach dem Kodieren dieser Symbole fertig ist. Es wäre auch denkbar, statt einer festgelegten Länge des Textes, ein `eof`[1] Symbol einzuführen. Das Terminate Signal würde dann nicht vom Incrementer geliefert werden, sondern vom Parallel Loader, der nach Lesen von `eof` das Terminate Signal setzt.

Die Funktionsweise kann man in Abbildung 4.3 sehen.

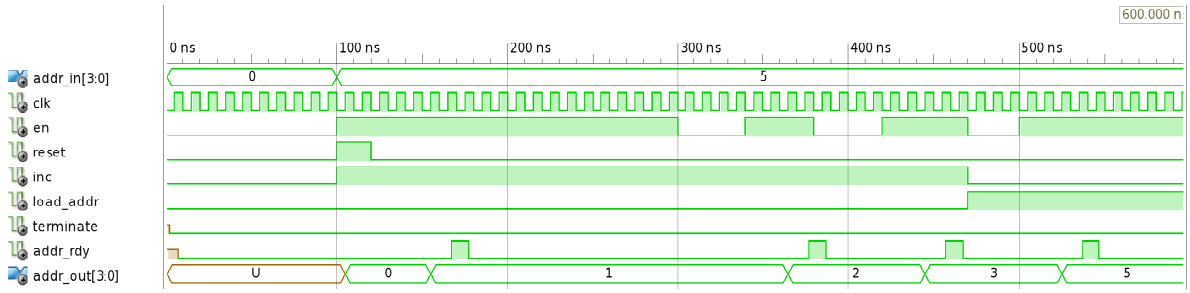

Abbildung 4.3: Incrementer Test Bench

[1] end of file

Name	Typ	Wert	Beschreibung
en	in	0 1	Wartezustand Einheit aktiviert
reset	in	1	`addr_out='000'` `terminate=0`
inc	in	1	Adresse wird um 1 erhöht
load_addr	in	1	Addresse `addr_in` wird geladen
addr_in	in	3:0	zu ladende Adresse
addr_rdy	out	1	Adresse liegt an
addr_out	out	3:0	Adresse
terminate	out	1	Endadresse erreicht

Tabelle 4.2: Incrementer Pins

4.4 RAM

Das RAM beinhaltet den zu kodierenden Text. Der Koder liest vier Symbole gleichzeitig ein. Wir hatten für ein Symbol vier Bits reserviert. Die gesamte Datenbreite des RAM Moduls muss also $4 \cdot 4 = 16$ Bits breit sein, um alle Daten parallel ausgeben zu können.

Um eine Konvertierung zu sparen haben wir die Symbole als Zahlen in den Speicher geschrieben. Für a=0, b=1, c=2, d=3 und e=4. Bei Adresse 0 ist also die Zahl 0122_h gespeichert und bei Adresse 1 4302_h, um die Folge *abcc edac* abzubilden. In unserem Beispiel werden im ersten Durchlauf die Buchstaben *abcc* = 0122 von der Adresse 0 gelesen.

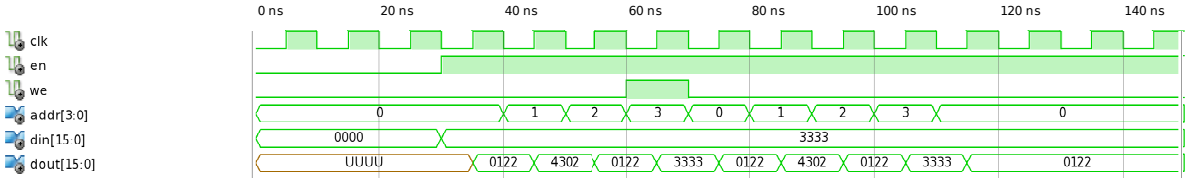

Abbildung 4.4: RAM Testbench

Die Testbench zum RAM Modul ist in Abbildung 4.4 abgebildet. Bei 60 ns ist das `we` Signal gesetzt. In dem Fall liegt am Ausgang der Wert `din` an, der gleichzeitig in Speicherzelle 3 geschrieben wird.

4.5 Parallel Loader

Stehen die Symbole bereit, werden diese Daten für den Koder passend gewandelt. Das geschieht im Modul Parallel Loader. Dieses Wandeln entspricht dem Anwenden eines

Name	Typ	Wert	Beschreibung
en	in	0 1	Wartezustand Einheit aktiviert
we	in	1	`din` wird in `addr` geschrieben und an `dout` weitergeleitet
addr_in	in	3:0	Adresse
din	in	15:0	Daten Eingang
dout	out	15:0	Daten Ausgang

Tabelle 4.3: RAM Pins

Modells auf den Text. Ein großer Vorteil des Arithmetischen Kodierens ist die einfache Änderung des Modells im Algorithmus. In unserem Beispiel braucht man hier nur dieses Modul ändern, um ein anderes Modell zu wählen. Man könnte hier beispielsweise ein adaptives Modell benutzen.

Wir benutzen ein Markov Order 0 Modell. Das heisst, dass die Wahrscheinlichkeiten vorher bekannt sein müssen. Das haben wir in unserem Fall schon in Tabelle 2.6 dargestellt.

Die Zuordnung von `tot_count` und `low_count` zu den am Eingang liegenden Symbolen wird über einen direkten Offset generiert. Die Symbole werden als die zu lesende Adresse interpretiert. Die Werte von `tot_count` und `low_count` sind in zwei separaten ROM Baustein Gruppen kodiert. Insgesamt sind also acht ROM Module notwendig. Das erscheint etwas viel, ist jedoch für eine echte Parallelisierung unumgänglich. Durch die Verwendung von Offsets werden für den Entwurf keine zusätzlichen Komparatoren benötigt wie in [SMJ99].

Beispiel 14. Das dritte Symbol sei ein c. Am Eingang `din` liegt an den Leitungen `din`$(8..11)$ der Wert $2 \equiv c$ an. Der Wert 2 wird als Adresse (Offset) an die jeweiligen ROM Bausteine für `low` und `tot` angelegt. Die Werte 3 für `tot` und 3 für `low` werden als Ergebnisse an `low3` und `tot3` angelegt. $\Diamond$

Den vollständigen Quellcode kann man in Listing 5.1 auf Seite 83 im Appendix einsehen.

Name	Typ	Wert	Beschreibung
en	in	0 1	Wartezustand Einheit aktiviert
reset	in	1	`low1 ... low4='0000'`
din	in	15:0	Daten Eingang
rdy	out	1	Daten liegen an
low 1...4	out	15:0	low_count 1...4
tot 1...4	out	15:0	total_count 1...4

Tabelle 4.4: Parallel Loader Pins

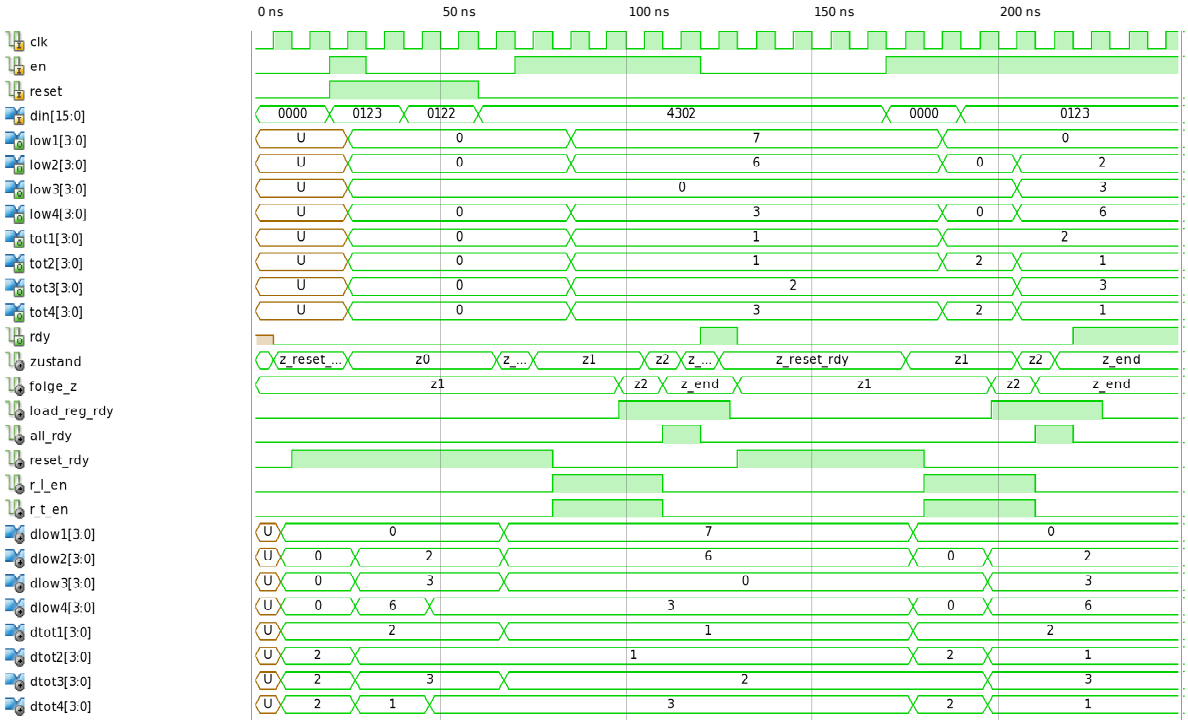

Abbildung 4.5: Parallel Loader Testbench

4.6 Parallel Koder

Der Parallel Koder bildet den in Abbildung 4.1 dargestellten Baum nach. Dieser besteht ausschließlich aus `BPE`s und `GPE`s.

4.6.1 BPE

Die Berechnungsgrundlage für die `BPE` sind die Gleichungen (3.8) und (3.9). Wir setzen $r_0 = \texttt{range}_0$, $F_x(x_0) = \frac{\texttt{cum\_count}}{\texttt{tot}}$ und $p(x_0) = \frac{\texttt{tot\_count}}{\texttt{tot}}$. Daraus ergibt sich

$$\begin{aligned}\texttt{low} &= \texttt{range}_0 \cdot F_x(x_0) \\ &= \texttt{range}_0 \cdot \frac{\texttt{cum\_count}_x}{\texttt{tot}}\end{aligned}$$

und

$$\begin{aligned}\texttt{range} &= \texttt{range}_0 \cdot p(x_0) \\ &= \texttt{range}_0 \cdot \frac{\texttt{tot\_count}_x}{\texttt{tot}}\end{aligned}$$

Weil $\texttt{range}_0 = 8000$ und $\texttt{tot} = 8$ können wir die Gleichungen schreiben als

$$\begin{aligned}\texttt{low} &= \texttt{range}_0 \cdot \frac{\texttt{cum\_count}_x}{\texttt{tot}} \\ &= \frac{\texttt{range}_0}{\texttt{tot}} \cdot \texttt{cum\_count}_x \\ &= \frac{8000}{8} \cdot \texttt{cum\_count}_x \\ &= 1000 \cdot \texttt{cum\_count}_x\end{aligned}$$

und entsprechend

$$\begin{aligned}\texttt{range} &= \texttt{range}_0 \cdot \frac{\texttt{tot\_count}_x}{\texttt{tot}} \\ &= \frac{\texttt{range}_0}{\texttt{tot}} \cdot \texttt{tot\_count}_x \\ &= \frac{8000}{8} \cdot \texttt{tot\_count}_x \\ &= 1000 \cdot \texttt{tot\_count}_x\end{aligned}$$

Die Bit Breite von `cum_count` und `tot_count` sind 4 Bits. Die Multiplikation mit 16 Bits ergeben ein 20 Bit Ergebnis. Wir wissen, dass das Ergebnis hier immer innerhalb der von uns gewählten Bit Breite von 16 bleiben wird, weil sowohl $\texttt{cum\_count} \leq \texttt{tot}$ als auch $\texttt{tot\_count} \leq \texttt{tot}$ gilt. Beide Gleichungen können in einer Stufe parallel berechnet werden. Abbildung 4.6 zeigt den internen Aufbau der `BPE`.

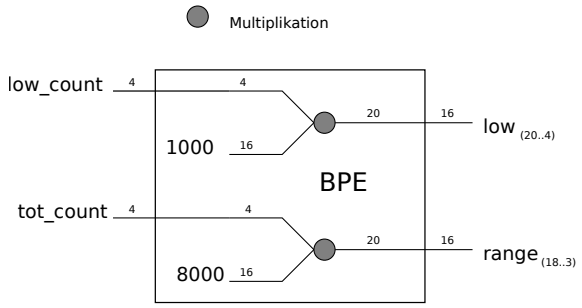

Abbildung 4.6: Schematische BPE

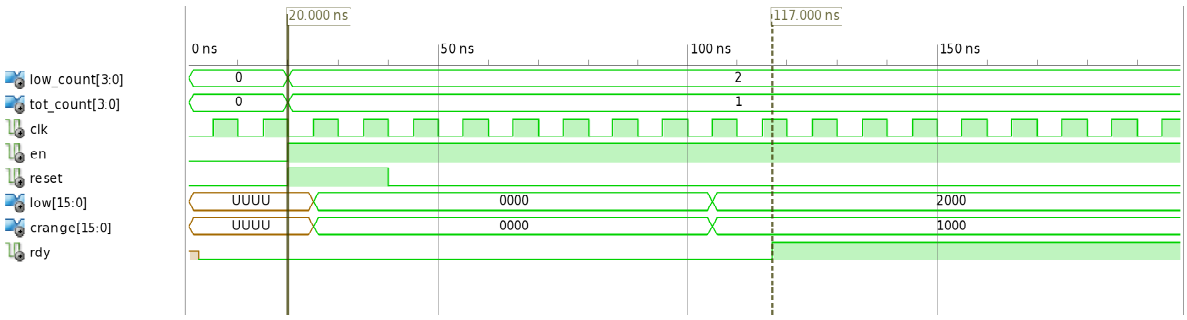

Abbildung 4.7: Testbench für BPE

4.6.2 GPE

Die `GPE` ist etwas komplexer als die `BPE`. Die Gleichungen hierfür sind (3.6) und (3.7). Diese Formeln bringen zwei Ketten von L und r zusammen. Für das Modul gehen jeweils $\texttt{range}_1$ und $\texttt{range}_2$, beziehungsweise $\texttt{low}_1$ und $\texttt{low}_2$ in die Rechnung ein. Zusammen mit den Annahmen aus Kapitel 4.6.1 BPE erhalten wir

$$\begin{aligned}\texttt{low} &= \texttt{low}_1 + \frac{\texttt{range}_1 \cdot \texttt{low}_2}{\texttt{range}_0} \\ &= \texttt{low}_1 + \frac{\texttt{range}_1 \cdot \texttt{low}_2}{8000}\end{aligned}$$

und für r

$$\begin{aligned}\texttt{range} &= \frac{\texttt{range}_1 \cdot \texttt{range}_2}{\texttt{range}_0} \\ &= \frac{\texttt{range}_1 \cdot \texttt{range}_2}{8000}\end{aligned}$$

Wir sehen hier, warum es von Vorteil ist, für r_0 eine Zweierpotenz zu wählen. So kann die Division durch einfache Bitverschiebung umgesetzt werden, andernfalls wäre hier ein Ganzzahl-Dividierer notwendig.

Die Berechnung von `low` erfolgt in zwei Schritten. Erst die Multiplikation, dann die Addition. Die `GPE`s haben dadurch eine höhere Latenz als die `BPE`s. Den internen Aufbau sehen wir in Abbildung 4.8.

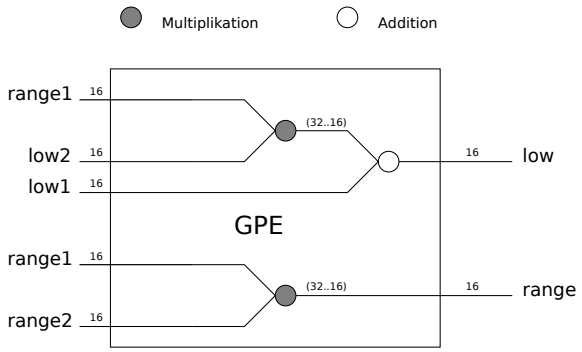

Abbildung 4.8: Schematische GPE

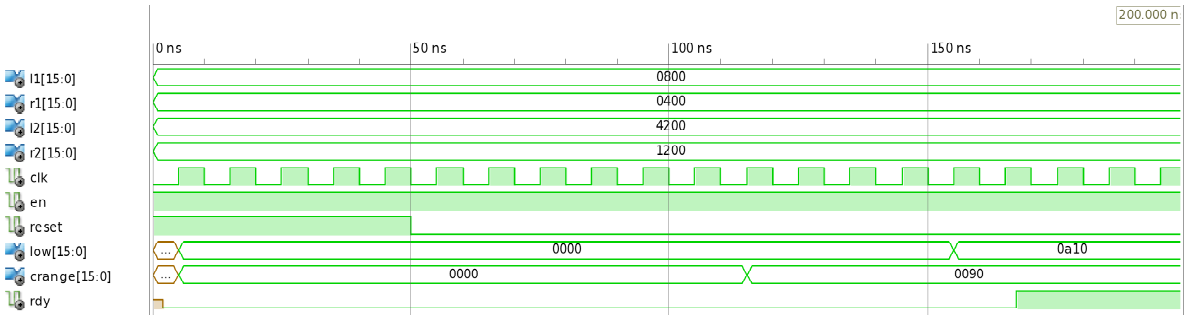

Abbildung 4.9: Testbench für GPE

4.6.3 Der Koder

Jetzt werden BPEs und GPEs gemäß Abbildung 4.1 miteinander verbunden. Ein Steuerwerk aktiviert nacheinander die einzelnen Stufen: zuerst BPE1-4, GPE1 und 2, GPE3 und schließlich GPE4.

Diese «Serialisierung» ist nicht notwendig. Die Schaltung ist ein rein kombinatorisches Netzwerk. Falls man an dieser Stelle den schnellstmöglichen Durchsatz erreichen will, kann man sich den längsten Pfad des Moduls ausrechnen lassen und diese Latenz warten.

Die genaue Verschaltung sehen wir in Abbildung 4.10 auf Seite 73. Das Technologieschaltnetz ist in Spalten eingeteilt. Jeder Block in derselben Spalte wird parallel abgearbeitet. Man kann hier gut erkennen, wie die BPEs in einer Stufe geschaltet sind. Es folgen die zwei parallelen Stufen der GPEs und am Ende erhält die oberste GPE (in der Abbildung ganz rechts) die Werte $r = \texttt{range}_0$ und $L = \texttt{low}_0$ aus der vorhergehenden Berechnung. Im «top_modul», der obersten Entwurfsebene muss der Ausgang low und range zu $L = \texttt{low}_0$ und $r = \texttt{range}_0$ zurückgeführt werden. Das entspricht der Verkettung der Berechnungen. Zu Beginn der Rechnung müssen $\texttt{low}_0$ mit 0 und $\texttt{range}_0$ mit 8000 initialisiert werden.

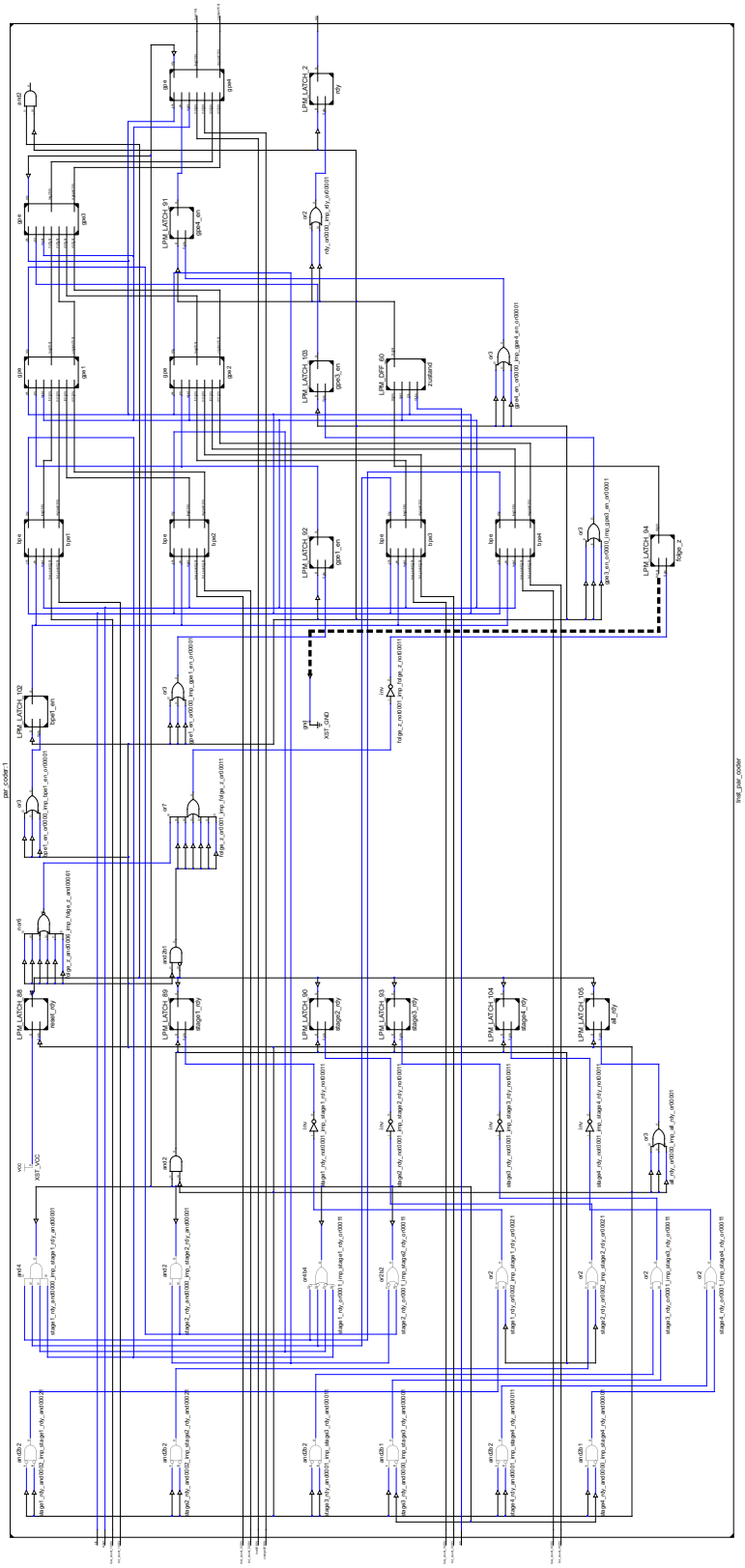

Abbildung 4.10: Technologie Block Parallel Koder

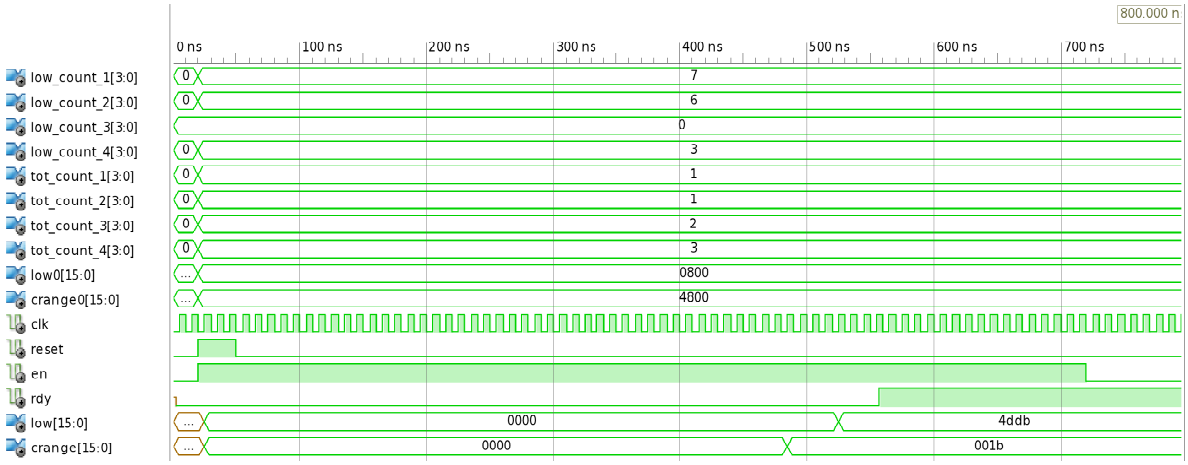

Abbildung 4.11: Testbench für Parallel Koder

Name	Typ	Wert	Beschreibung
en	in	0 1	Wartezustand Einheit aktiviert
reset	in	1	`crange` ='0000 0000 0000 0000' `low` ='0000 0000 0000 0000'
low count 1...4	in	15:0	low_count 1...4
tot count 1...4	in	15:0	total_count 1...4
rdy	out	1	Daten liegen an
crange	out	1	r
low	out	1	L
crangeo	out	1	r aus vorangehender Berechnung
lowo	out	1	L aus vorangehender Berechnung

Tabelle 4.5: Parallel Koder Pins

4.7 Normalizer

Das ist der komplexeste Block der Schaltung. Der Normalizer hat hier zwei Funktionen: Zum einen skaliert er L und r und generiert damit den Ausgangs-Strom, zum anderen hat er auch das RAM für die Speicherung desselben. Wie alle anderen ist auch dieses Modul als Moore Automat realisiert. Durch die Schleife ist das Modul kein kombinatorisches Netzwerk. Jedoch wurde in Kapitel 3.4.3 eine Möglichkeit vorgestellt, wie das realisierbar wäre.

Es gibt zwei verschiedene Abbruchbedingungen für die Schleife: eine falls ($\texttt{range} > \frac{r_0}{2} = 4000$) ist und die andere ist ($\texttt{low}_{temp} + \texttt{range}_{temp} > r_0 = 8000$). Im zuletzt genannten Fall

dürfen die Register `low` und `range` nicht mit den Werten von $\texttt{low}_{temp}$ und $\texttt{range}_{temp}$ geladen werden. Im Zustandsautomaten heißen diese Pfade `z_all_rdy_norm` beziehungsweise `z_exit_old_val_norm`.

Das interne RAM Modul speichert die kodierten Daten. Es ist als 1 Bit RAM implementiert, was die hohe Anzahl von Adress-Leitungen erklärt. Im Zustand `z_select_bit_norm` wird abgefragt, ob als Kode eine 0 oder 1 herausgegeben wird (Zeile 313 im Listing 5.2).

Tabelle 4.1 zeigt die Normalisierung von $L = 0A10$ und $r = 0090$. Das Ergebnis 0001010 wird seriell in den Speicher geschrieben. Nach dem ersten Durchlauf steht der RAM Zähler `mod_addr_out` auf 7.

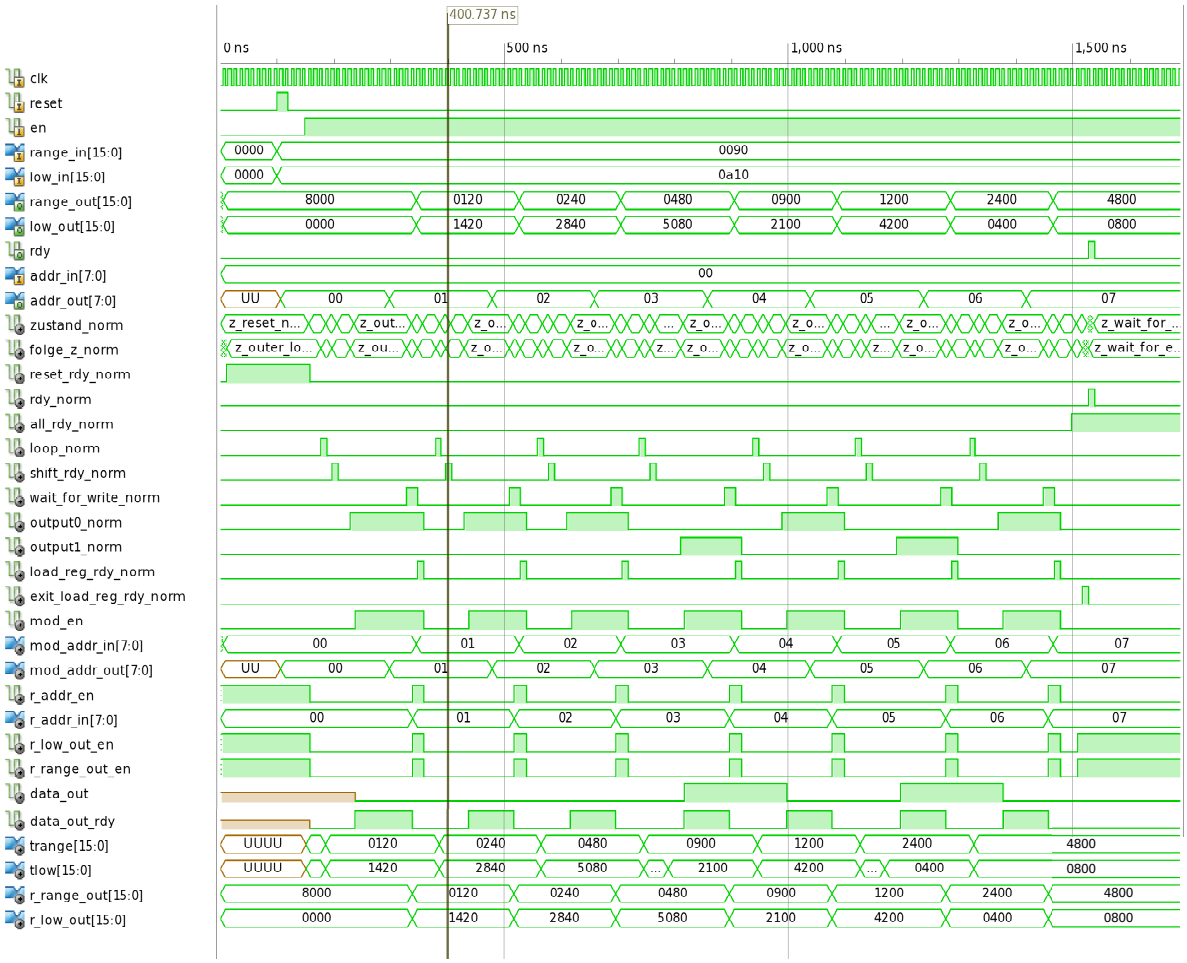

Abbildung 4.12: Testbench für Normalizer mit $L = 0A10$ und $r = 0090$

Name	Typ	Wert	Beschreibung
en	in	0 1	Wartezustand Einheit aktiviert
reset	in	1	`addr_out` ='0000 0000' `low_out` ='0000 0000 0000 0000' `range_out` ='0000 0000 0000 0000'
addr_in	in	7:0	erste Adresse zum Schreiben des Ausgangs-Stroms für internes RAM
low_in	in	15:0	low aus Koder Block
range_in	in	15:0	range aus Koder Block
rdy	out	1	Daten liegen an
range_out	out	1	normalisiertes r
low_out	out	1	normalisiertes L
addr_out	out	7:0	Adresse des internen RAMs nach dem Schreiben des Ausgangs-Stroms. Zeigt jetzt auf die nächste zu Schreibende Adresse.

Tabelle 4.6: Normalizer Pins

4.8 Control Unit

Das Steuerwerk hat die Aufgabe, die einzelnen Schritte in einer Abfolge zu steuern. Das geschieht durch das einzelne Setzen und Löschen der Aktivierungs-Bits. Es bildet ein großes Übergangsschaltnetz eines Moore Automaten. Ein Durchlauf geht vom Inkrementieren der RAM Adresse bis zur fertigen Kodierung der gelesenen Symbole. Der Durchlauf ist in Zyklen eingeteilt, die jeweils auf das `ready` Signal der einzelnen Module warten.

In dieser Form werden die Module eines nach dem anderen aktiviert und deaktiviert. Das heißt, dass jedes Modul pro Durchlauf nur einmal aktiviert ist. Man kann den Durchsatz erhöhen, indem man alle Module im gesamten Durchlauf aktiviert hält. Die Steuereinheit (Control Unit) wartet auf sämtliche `ready` Signale statt auf einzelne. Damit wird die Zykluslänge immer gleich lang. Pro Zyklus werden neue Daten eingelesen und ausgegeben. Somit ist es möglich, pro Zyklus 4 Symbole zu kodieren.

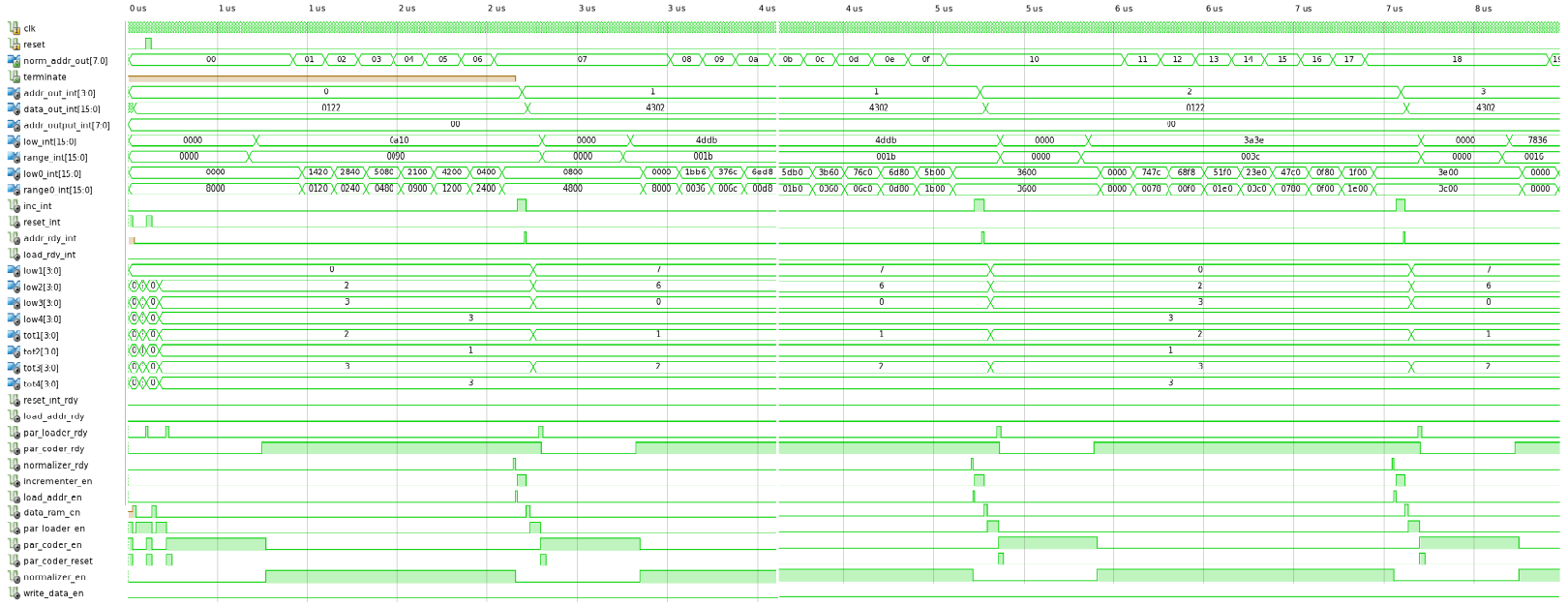

Abbildung 4.13: Gesamtentwurf Testbench

Name	Typ	Wert	Beschreibung
reset_in	in	1	`reset out ='1'`
addr_rdy	in	1	Adresse aus Incrementer liegt an
load_addr_in	in	1	lade Incrementer mit Adresse
normalizer_rdy	in	1	Normalizer fertig
par_coder_rdy	in	1	Parallel Koder fertig
terminate_in	in	1	letzte Adresse erreicht
data_ram_en	out	1	aktiviere RAM
incrementer_en	out	1	aktiviere Incrementer
inc_en	out	1	Adresse inkrementieren

Tabelle 4.7: Control Unit Pins

4.9 Validierung und Synthese Ergebnisse

Für jedes Modul wurde eine Testbench geschrieben und die Ergebnisse mit dem Referenzbeispiel verglichen. Das Timing der einzelnen Signale wurde durch die Gesamtenwurfs-Testbench überprüft.

Das Synthese Ergebnis auf ein Xilinx Virtex 5 XC5VSX50T ist in Tabelle 4.8 zusammengefasst.

Device Utilization Summary			
Slice Logic Utilization	Used	Available	Utilization
Number of Slice Registers	1,086	19,200	5%
Number used as Flip Flops	412		
Number used as Latches	674		
Number of Slice LUTs	507	19,200	2%
Number used as logic	479	19,200	2%
Number using O6 output only	478		
Number using O5 output only	1		
Number used as Memory	4	5,120	1%
Number used as Single Port RAM	4		
Number using O6 output only	4		
Number used as exclusive route-thru	24		
Number of DSP48Es	8	32	25%

Tabelle 4.8: Synthese Ergebnis Xilinx Virtex 5 XC5VSX50T -1 FFT1136

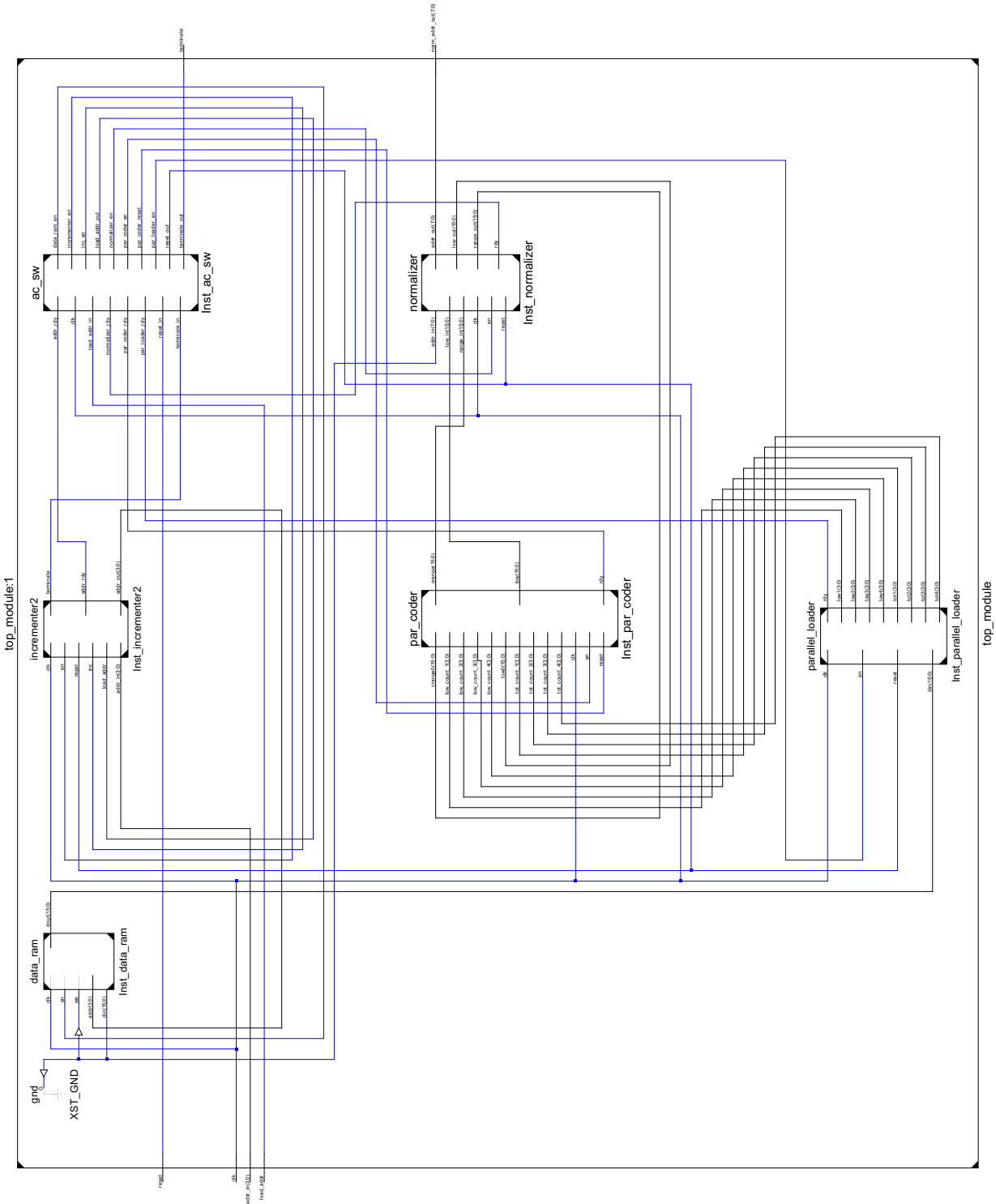

Abbildung 4.2: Gesamtentwurf

5 Zusammenfassung und Ausblick

Im Kapitel 2 – Arithmetisches Kodieren haben wir umfassend die Grundlagen des Arithmetischen Kodierens behandelt. Wir haben gesehen, dass die Rechenvorschrift aus zwei Differenzengleichungen ((2.6) und (2.7)) besteht. Die Lösungen von Differenzen- und Differenzialgleichungen lassen sich nicht parallelisieren — obwohl Teile, vor allem bei Partiellen Differenzialgleichungen, parallel berechnet werden können. Eine besondere Problematik des Algorithmus ist die dafür benötigte unendliche Präzision. Wir haben gesehen, dass dieses Problem mit einer Skalierung gelöst werden kann. Eine Implementierung in C des Algorithmus wurde vorgestellt.

Den Kern der Arbeit bildete Kapitel 3 – Parallelisierung. Hier wurden aufbauend auf das in Kapitel 2 eingeführte sequenzielle Vorgehen Ansätze für eine Parallelisierung gezeigt. Wie schon oben erwähnt wird nicht die Gleichung an sich parallelisiert, sondern Teile der Berechnungen.

Eine Analyse vorhandener Lösungen wurde durchgeführt. Wir konnten herausfinden, dass alle hier vorgestellten Paper im Prinzip Varianten eines Algorithmus sind. Mit dieser Erkenntnis haben wir die Vor- und Nachteile, die sich daraus ergeben, gezeigt. Zum Beispiel bleibt die Präzision für große Parallelität ein Problem. Außerdem wurde ein Vorteil welches sich aus den unterschiedlichen Gleichungen ergibt, nämlich um verschiedene Architekturen zu kombinieren, erläutert. Im Zusammenhang mit der Skalierung haben wir einen Algorithmus entwickelt, der wesentlich einfacher ist als der in Kapitel 2. Es wurde skizziert, wie man diesen durch geeignete Implementierung eines Prioritätsencoder beschleunigen kann. Wir haben eine Gleichung zur Berechnung der nötigen Bit Breite herausgearbeitet. Mit dieser Formel sind wir jetzt in der Lage, eine mindestens erforderliche Bit Breite für die Register anhand gegebener Anforderungen zu berechnen.

Im Kapitel 4 – Implementierung wurde die Theorie anhand einer realen VHDL Implementierung verifiziert und eine Machbarkeit gezeigt. Details zu den Berechnungen, die Auswirkungen auf die Implementierung haben, wurden besprochen.

5.1 Ausblick

Unsere einleitende Frage — ob sich das Arithmetische Kodieren durch Parallelisierung beschleunigen lässt — können wir bejahen. Einige Fragen, die sich im Verlauf der Arbeit ergaben, blieben offen. Wir wollen ein paar davon an dieser Stelle wieder aufgreifen:

Mit steigender Anzahl an parallelen Stufen steigt die benötigte Bit Breite. Multiplizierer in großer Bit Breite sind langsam und komplex. Man könnte herausfinden, ab wann sich der Aufwand nicht mehr lohnt.

Falls ein Symbol a extrem oft vorkommt und damit $p(a)$ nahe bei r_0 liegt, wird lange nicht skaliert. In Kapitel 3.4.2 wurde angesprochen, dass dies zu einem Unterlauf führen könnte. Ob das tatsächlich stattfindet, wäre Gegenstand weiterer Untersuchungen.

Bei einem Pipelining, der nach jedem Takt neue Symbole einliest, müssen die einzelnen Pipeline-Stufen aus kombinatorischen Netzwerken bestehen. Die Sklalierung mit Schleifenkonstruktion ist nicht rein kombinatorisch. Eine Lösung, die Skalierung kombinatorisch zu gestalten, ist die oben genannte mit Prioritätsencoder. Man kann die vorgestellte Implementierung dahingehend erweitern.

Ein geeigneter Dekoder muss entworfen werden. Ob man hier einfach den Sequenziellen Dekoder verwenden kann oder aber durch die unterschiedliche Rechnung im parallelen Fall andere Rundungen entstehen und damit ein abgestimmter Dekoder notwendig wird, muss analysiert werden.

Wir stellen hiermit fest, dass diese Arbeit nicht nur unsere Frage beantwortet, sondern eine Ausgangsbasis und Grundlage für einige sehr interessante Nachforschungen bildet.

Appendix

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer: Maximiliano Keller
--
-- Create Date:  15:59:05 10/25/2010
-- Design Name:
-- Module Name:  parallel_loader - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity parallel_loader is
    Port ( clk : in STD_LOGIC;
           en : in STD_LOGIC;
           reset : in STD_LOGIC;
           din : in STD_LOGIC_VECTOR (15 downto 0);
```

```
37          low1 : out STD_LOGIC_VECTOR (3 downto 0);
            low2 : out STD_LOGIC_VECTOR (3 downto 0);
            low3 : out STD_LOGIC_VECTOR (3 downto 0);
            low4 : out STD_LOGIC_VECTOR (3 downto 0);
            tot1 : out STD_LOGIC_VECTOR (3 downto 0);
42          tot2 : out STD_LOGIC_VECTOR (3 downto 0);
            tot3 : out STD_LOGIC_VECTOR (3 downto 0);
            tot4 : out STD_LOGIC_VECTOR (3 downto 0);
            rdy : out STD_LOGIC);
    end parallel_loader;
47
    architecture Behavioral of parallel_loader is

    type zustaende is ( z0, z1, z2, z3, z4, z5, z6, z7,
       z_reset_rdy, z_end );
52  signal zustand, folge_z : zustaende := z0 ;

    constant rom_lat : time := 13 ns;
    constant reg_lat : time := 23 ns;
    constant sig_lat : time := 5 ns;
57
    signal load_reg_rdy : std_logic := '0';
    signal all_rdy : std_logic := '0';
    signal reset_rdy : std_logic := '0';

62  signal r_l_en : std_logic := '0';
    signal r_t_en : std_logic := '0';

    signal dlow1 : std_logic_vector (3 downto 0);
    signal dlow2 : std_logic_vector (3 downto 0);
67  signal dlow3 : std_logic_vector (3 downto 0);
    signal dlow4 : std_logic_vector (3 downto 0);

    signal dtot1 : std_logic_vector (3 downto 0);
    signal dtot2 : std_logic_vector (3 downto 0);
72  signal dtot3 : std_logic_vector (3 downto 0);
    signal dtot4 : std_logic_vector (3 downto 0);

    component tot_rom is
       Port ( clk : in STD_LOGIC;
77            addr : in STD_LOGIC_VECTOR (2 downto 0);
              dout : out STD_LOGIC_VECTOR (3 downto 0));
    end component;

    component low_rom is
82     Port ( clk : in STD_LOGIC;
              addr : in STD_LOGIC_VECTOR (2 downto 0);
              dout : out STD_LOGIC_VECTOR (3 downto 0));
    end component;

87  component dff_4_bit is
       Port ( d : in STD_LOGIC_VECTOR (3 downto 0);
              clk : in STD_LOGIC;
              reset : in STD_LOGIC;
              ce : in STD_LOGIC;
```

```
92            q : out STD_LOGIC_VECTOR (3 downto 0));
    end component;

    for all: low_rom use entity work.low_rom ( Behavioral );
    for all: tot_rom use entity work.tot_rom ( Behavioral );
97  for all: dff_4_bit use entity work.dff_4_bit ( Behavioral );

    begin

    r_l1: dff_4_bit port map (
102    clk => clk,
       ce => r_l_en,
       d => dlow1,
       q => low1,
       reset => reset
107    );

    r_l2: dff_4_bit port map (
       clk => clk,
       ce => r_l_en,
112    d => dlow2,
       q => low2,
       reset => reset
       );

117 r_l3: dff_4_bit port map (
       clk => clk,
       ce => r_l_en,
       d => dlow3,
       q => low3,
122    reset => reset
       );

    r_l4: dff_4_bit port map (
       clk => clk,
127    ce => r_l_en,
       d => dlow4,
       q => low4,
       reset => reset
       );
132
    r_t1: dff_4_bit port map (
       clk => clk,
       ce => r_t_en,
       d => dtot1,
137    q => tot1,
       reset => reset
       );

    r_t2: dff_4_bit port map (
142    clk => clk,
       ce => r_t_en,
       d => dtot2,
       q => tot2,
       reset => reset
```

```
147     );

        r_t3: dff_4_bit port map (
           clk => clk,
           ce => r_t_en,
152        d => dtot3,
           q => tot3,
           reset => reset
           );

157     r_t4: dff_4_bit port map (
           clk => clk,
           ce => r_t_en,
           d => dtot4,
           q => tot4,
162        reset => reset
           );

        low1_out: low_rom port map (
           clk => clk,
167        addr => din (2 downto 0),
           dout => dlow4
           );

        low2_out: low_rom port map (
172        clk => clk,
           addr => din (6 downto 4),
           dout => dlow3
           );

177     low3_out: low_rom port map (
           clk => clk,
           addr => din (10 downto 8),
           dout => dlow2
           );
182
        low4_out: low_rom port map (
           clk => clk,
           addr => din (14 downto 12),
           dout => dlow1
187        );

        tot1_out: tot_rom port map (
           clk => clk,
           addr => din (2 downto 0),
192        dout => dtot4
           );

        tot2_out: tot_rom port map (
           clk => clk,
197        addr => din (6 downto 4),
           dout => dtot3
           );

        tot3_out: tot_rom port map (
```

```
202     clk => clk,
        addr => din (10 downto 8),
        dout => dtot2
        );

207  tot4_out: tot_rom port map (
        clk => clk,
        addr => din (14 downto 12),
        dout => dtot1
        );
212
     z_speicher: process ( clk, en )
        begin
           if reset = '1' then zustand <= z0 after sig_lat;
           elsif (clk'event and clk = '1') then
217           if en ='1' then
                 zustand <= folge_z;
              else
                 zustand <= z_reset_rdy;
              end if;
222        end if;
     end process z_speicher;

     ue_sn: process ( zustand, load_reg_rdy, all_rdy,
        reset_rdy
227     )
        begin
           case zustand is
                           when z0 =>
                                  folge_z <= z1;
232           when z1 =>
                                  if load_reg_rdy = '1' then
                                         folge_z <= z2 ;
                                  end if;
              when z2 =>
237                               if all_rdy = '1' then
                                         folge_z <= z_end;
                                  end if;
              when z_reset_rdy =>
                                  if reset_rdy = '1' then
242                                      folge_z <= z1;
                                  end if;
              when others =>
                 folge_z <= folge_z;
           end case;
247  end process ue_sn;

     aus_sn: process ( zustand )
        begin
           case zustand is
252           when z0 => -- reset zustand, alles auf null
                                  all_rdy <= '0' after sig_lat;
                                  rdy <= '0' after sig_lat;
                                  load_reg_rdy <= '0' after sig_lat;
```

```
257                             r_l_en <= '0' after sig_lat;
                                r_t_en <= '0' after sig_lat;

                        when z1 => -- lade register
                reset_rdy <= '0' after sig_lat;
262                             r_l_en <= '1' after sig_lat;
                                r_t_en <= '1' after sig_lat;
                load_reg_rdy <= '1' after reg_lat;

                        when z2 => -- erste stage
267                             r_l_en <= '0' after sig_lat;
                                r_t_en <= '0' after sig_lat;

                load_reg_rdy <= '0' after reg_lat;
                all_rdy <= '1' after sig_lat;
272
                        when z_reset_rdy =>
                                all_rdy <= '0' after sig_lat;
                rdy <= '0' after sig_lat;
                reset_rdy <= '1' after sig_lat;
277
                        when z_end =>
                                all_rdy <= '0' after sig_lat;
                rdy <= '1' after sig_lat;

282         when others =>
        end case;
    end process aus_sn;

    end Behavioral;
```

Listing 5.1: Parallel Loader Code

```
    ----------------------------------------------------------------------------
    -- Company:
  3 -- Engineer: Maximiliano Keller
    --
    -- Create Date:  14:32:56 10/26/2010
    -- Design Name:
    -- Module Name:  normalizer - Behavioral
  8 -- Project Name:
    -- Target Devices:
    -- Tool versions:
    -- Description:
    --
 13 -- Dependencies:
    --
    -- Revision:
```

```
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_arith.ALL;
use IEEE.STD_logic_unsigned.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity normalizer is
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           en : in STD_LOGIC;
           range_in : in std_logic_vector ( 15 downto 0 );
           low_in : in std_logic_vector ( 15 downto 0 );
           range_out : out std_logic_vector ( 15 downto 0 );
           low_out : out std_logic_vector ( 15 downto 0 );
           rdy : out STD_LOGIC;
           addr_in : in STD_LOGIC_VECTOR (7 downto 0);
           addr_out : out STD_LOGIC_VECTOR (7 downto 0));
end normalizer;

architecture Behavioral of normalizer is

component output_module is
    Port ( clk : in STD_LOGIC;
           reset : in STD_LOGIC;
           en : in STD_LOGIC;
           data : in STD_LOGIC;
           data_rdy : in STD_LOGIC;
           rdy : out STD_LOGIC;
           addr_in : in STD_LOGIC_VECTOR (7 downto 0);
           addr_out : out STD_LOGIC_VECTOR (7 downto 0));
end component;

component dff_1_bit is
      Port ( clk : in STD_LOGIC;
             d : in STD_LOGIC;
             q : out STD_LOGIC;
             reset : in STD_LOGIC;
             ce : in STD_LOGIC
           );
end component;

component dff_8_bit is
      Port ( clk : in STD_LOGIC;
```

```
                d : in STD_LOGIC_VECTOR (7 downto 0);
                q : out STD_LOGIC_VECTOR (7 downto 0);
                reset : in STD_LOGIC;
                ce : in STD_LOGIC
            );
end component;

component dff_16_bit is
      Port ( clk : in STD_LOGIC;
                d : in STD_LOGIC_VECTOR (15 downto 0);
                q : out STD_LOGIC_VECTOR (15 downto 0);
                reset : in STD_LOGIC;
                ce : in STD_LOGIC
            );
end component;

type zustaende_norm is (
   z_reset_norm, z_outer_loop_norm, z_loop_start_norm,
   z_select_bit_norm, z_output0_norm, z_output1_norm,
   z_next_iteration_norm, z_all_rdy_norm,
   z_exit_old_val_norm, z_wait_for_en_norm
   );
signal zustand_norm, folge_z_norm : zustaende_norm := z_reset_norm ;

-- setting some arbitrary values for latencies
constant reset_lat : time := 13 ns;
constant add_lat : time := 23 ns;
constant sub_lat : time := 23 ns;
constant comp_lat : time := 23 ns;
constant sig_lat : time := 2 ns;
constant ram_lat : time := 2 ns;
constant load_lat : time := 12 ns;
constant inc_lat : time := 12 ns;
constant shift_lat : time := 12 ns;

signal reset_rdy_norm : std_logic := '0';
signal rdy_norm : std_logic := '0';
signal all_rdy_norm : std_logic := '0';
signal all_rdy_old_value_norm : std_logic := '0';
signal loop_norm : std_logic := '0';
signal shift_rdy_norm : std_logic := '0';
signal wait_for_write_norm : std_logic := '0';
signal output0_norm : std_logic := '0';
signal output1_norm : std_logic := '0';
signal load_reg_rdy_norm : std_logic := '0';
signal exit_load_reg_rdy_norm : std_logic := '0';

signal mod_en : std_logic := '0';
signal mod_addr_in : std_logic_vector ( 7 downto 0 );
signal mod_addr_out : std_logic_vector ( 7 downto 0 );

signal r_addr_en : std_logic := '0';
signal r_addr_in : std_logic_vector ( 7 downto 0 );

signal r_low_out_en : std_logic := '0';
```

```
    signal r_range_out_en : std_logic := '0';

128 signal gnd_sig : std_logic := '0'; -- standard low

    signal ram_en : std_logic := '0';
    signal ram_write_en : std_logic := '0';

133 signal data_out : std_logic;
    signal data_out_rdy : std_logic;
    signal trange : std_logic_vector ( 15 downto 0 );
    signal tlow : std_logic_vector ( 15 downto 0 );
    signal r_range_out : std_logic_vector ( 15 downto 0 );
138 signal r_low_out : std_logic_vector ( 15 downto 0 );
    signal const_one : std_logic_vector ( 7 downto 0 ) := x"01";

    -- initialising ffs
    for all : dff_16_bit use entity work.dff_16_bit ( Behavioral );
143 for all : dff_8_bit use entity work.dff_8_bit ( Behavioral );
    for all : dff_1_bit use entity work.dff_1_bit ( Behavioral );
    for all : output_module use entity work.output_module ( Behavioral );

    begin
148
    output_module_1 : output_module port map ( clk => clk,
         en => mod_en, reset => reset,
         addr_in => mod_addr_in,
         addr_out => mod_addr_out,
153      data => data_out, data_rdy => data_out_rdy,
         rdy => wait_for_write_norm );

    reg_addr : dff_8_bit port map ( d => r_addr_in, clk => clk,
         ce => r_addr_en, reset => reset, q => mod_addr_in );
158
    reg_low_out : dff_16_bit port map ( d => r_low_out, clk => clk,
         ce => r_low_out_en, reset => gnd_sig, q => low_out );
         -- no hard reset possible, load value

163 reg_range_out : dff_16_bit port map ( d => r_range_out, clk => clk,
         ce => r_range_out_en, reset => gnd_sig, q => range_out );
         -- no hard reset possible, load value

    z_speicher: process ( clk, en )
168   begin
        if reset = '1' then
          zustand_norm <= z_reset_norm;
        elsif (clk'event and clk = '1') then
          if en ='1' then
173         zustand_norm <= folge_z_norm;
          end if;
        end if;
    end process z_speicher;

178 ue_sn_norm: process (
      reset_rdy_norm, loop_norm, shift_rdy_norm,
      output0_norm, output1_norm, data_out_rdy,
```

```
   wait_for_write_norm, all_rdy_norm,
   load_reg_rdy_norm, all_rdy_old_value_norm,
183  exit_load_reg_rdy_norm, rdy_norm,
   zustand_norm, en )
   begin
      case zustand_norm is

188                  when z_reset_norm =>
         if reset_rdy_norm = '1' then
                              folge_z_norm <= z_outer_loop_norm;
         end if;

193                  when z_outer_loop_norm =>
         if loop_norm = '1' then
            if wait_for_write_norm = '0' then -- reset wait_for_write_norm
                                 folge_z_norm <= z_loop_start_norm;
            end if;
198      elsif all_rdy_norm = '1' then
            folge_z_norm <= z_all_rdy_norm;
         end if;

                  when z_loop_start_norm =>
203      if shift_rdy_norm = '1' then
                              folge_z_norm <= z_select_bit_norm;
         end if;

                  when z_select_bit_norm =>
208      if output1_norm = '1' then
                              folge_z_norm <= z_output1_norm;
         elsif output0_norm = '1' then
            folge_z_norm <= z_output0_norm;
         end if;
213
                  when z_output1_norm =>
         if wait_for_write_norm = '1' then
                              folge_z_norm <= z_next_iteration_norm;
         elsif all_rdy_old_value_norm ='1' then
218         folge_z_norm <= z_exit_old_val_norm;
         end if;

                  when z_output0_norm =>
         if wait_for_write_norm = '1' then
223                           folge_z_norm <= z_next_iteration_norm;
         elsif all_rdy_old_value_norm ='1' then
            folge_z_norm <= z_exit_old_val_norm;
         end if;

228                  when z_next_iteration_norm =>
         if load_reg_rdy_norm = '1' then
                              folge_z_norm <= z_outer_loop_norm;
         end if;

233                  when z_all_rdy_norm => -- new low and range will be loaded
         if exit_load_reg_rdy_norm = '1' then
                              folge_z_norm <= z_exit_old_val_norm;
```

```
            end if;

238                     when z_exit_old_val_norm =>
            if rdy_norm = '1' then
                                folge_z_norm <= z_wait_for_en_norm;
            end if;

243                     when z_wait_for_en_norm =>
            -- wait for en = 0 to reset rdy signal
            if en = '0' then
                                folge_z_norm <= z_reset_norm;
            end if;
248
          when others =>
        end case;

    end process ue_sn_norm;
253
    aus_sn_norm: process ( en, reset, zustand_norm )
      begin
        case zustand_norm is
          when z_reset_norm => -- entry point for en = 1
258          -- set low0 und range0 to default values
             --
             -- do not reset registers (0 values)
             r_low_out <= x"0000" after sig_lat;
             r_range_out <= x"8000" after sig_lat;
263          r_low_out_en <= '1' after sig_lat;
             r_range_out_en <= '1' after sig_lat;

             r_addr_in <= addr_in after sig_lat;
             r_addr_en <= '1' after sig_lat;
268          -- set the initial values if en = 1
             -- this will be done if in this state and
             -- the en signal changes
             if en = '1' then
                trange <= range_in after sig_lat;
273             tlow <= low_in after sig_lat;
             end if;
             rdy_norm <= '0' after sig_lat;
             all_rdy_norm <= '0' after sig_lat;
             all_rdy_old_value_norm <= '0' after sig_lat;
278          exit_load_reg_rdy_norm <= '0' after sig_lat;

             reset_rdy_norm <= '1' after load_lat; -- loading addr_ff

          when z_outer_loop_norm =>
283          mod_en <= '0' after sig_lat;
             r_low_out_en <= '0' after sig_lat;
             r_range_out_en <= '0' after sig_lat;
             output0_norm <= '0' after sig_lat;
             output1_norm <= '0' after sig_lat;
288          shift_rdy_norm <= '0' after shift_lat;
             reset_rdy_norm <= '0' after sig_lat;
             data_out_rdy <= '0' after sig_lat;
```

```
            load_reg_rdy_norm <= '0' after sig_lat;
            r_addr_en <= '0' after sig_lat;
293
            if trange <= x"4000" then
               loop_norm <= '1' after comp_lat;
               all_rdy_norm <= '0' after comp_lat;
            else
298            all_rdy_norm <= '1' after comp_lat;
               loop_norm <= '0' after comp_lat;
            end if;

         when z_loop_start_norm =>
303         loop_norm <= '0' after sig_lat;
            all_rdy_norm <= '0' after sig_lat;

            tlow <= tlow ( 14 downto 0 ) & '0';
            trange <= trange ( 14 downto 0 ) & '0';
308
            shift_rdy_norm <= '1' after shift_lat;

         when z_select_bit_norm =>
            shift_rdy_norm <= '0' after sig_lat;
313         if tlow (15) = '1' then -- would send 1
               tlow <= '0' & tlow (14 downto 0) after sub_lat;
               output1_norm <= '1' after
                 ( comp_lat + sub_lat );
            else -- would send 0
318            output0_norm <= '1' after comp_lat;
            end if;

         when z_output1_norm =>
               if ( tlow + trange ) <= x"8000" then -- send or exit?
323               mod_en <= '1' after sig_lat;
                  data_out_rdy <= '1' after sig_lat;
                  data_out <= '1' after sig_lat;

                  -- output_module will set the wait_for_write_rdy bit
328            else
                  all_rdy_old_value_norm <= '1' after sig_lat;
               end if;

         when z_output0_norm =>
333            if ( tlow + trange ) <= x"8000" then -- send or exit?
                  mod_en <= '1' after sig_lat;
                  data_out_rdy <= '1' after sig_lat;
                  data_out <= '0' after sig_lat;

338               -- output_module will set the wait_for_write_rdy bit
               else
                  all_rdy_old_value_norm <= '1' after sig_lat;
               end if;

343      when z_next_iteration_norm =>
            r_low_out <= tlow after sig_lat;
            r_range_out <= trange after sig_lat;
```

```
            r_low_out_en <= '1' after sig_lat;
            r_range_out_en <= '1' after sig_lat;

            r_addr_en <= '1' after sig_lat;
            r_addr_in <= mod_addr_out after sig_lat;

            -- reset the wait_for_write_rdy signal by setting
            -- data_rdy to '0'
            data_out_rdy <= '0' after sig_lat;

            load_reg_rdy_norm <= '1' after load_lat;

         when z_all_rdy_norm =>
            r_low_out <= tlow after sig_lat;
            r_range_out <= trange after sig_lat;
            r_low_out_en <= '1' after sig_lat;
            r_range_out_en <= '1' after sig_lat;

            exit_load_reg_rdy_norm <= '1' after load_lat;

         when z_exit_old_val_norm =>
            exit_load_reg_rdy_norm <= '0' after sig_lat;
            rdy_norm <= '1' after sig_lat;

         when z_wait_for_en_norm =>
            rdy_norm <= '0' after sig_lat;

         when others =>
      end case;
end process aus_sn_norm;

rdy <= rdy_norm;
addr_out <= mod_addr_out;

end Behavioral;
```

Listing 5.2: Normalizer Code

Literaturverzeichnis

[BCK02] E. Bodden, M. Clasen, J. Kneis. Arithmetische Kodierung. 2002. (Zitiert auf den Seiten 27 und 29)

[Beu93] A. Beutelspacher. *Kryptologie*. Vieweg, Braunschweig, 3 edition, 1993. (Zitiert auf Seite 59)

[JJ94] Jiang, Jones. Parallel Design of Arithmetic Coding. *IEEPCDT: IEE Proceedings on Computers and Digital Techniques*, 141, 1994. (Zitiert auf den Seiten 11, 37, 43, 48, 49, 51, 52 und 60)

[LLSW96] H.-Y. Lee, L.-S. Lan, M.-H. Sheu, C.-H. Wu. A parallel architecture for arithmetic coding and its VLSI implementation. In *Circuits and Systems (CAS)*, volume 3, pp. 1309–1312. IEEE, 1996. doi:10.1109/MWSCAS.1996.593169. (Zitiert auf den Seiten 49, 51 und 60)

[Pas76] R. Pasco. *Source coding algorithms for fast data compression*. Ph.D. thesis, Stanford University, Palo Alto, CA, 1976. (Zitiert auf Seite 11)

[Ris76] J. Rissanen. Generalized Kraft Inequality and Arithmetic Coding. *IBM Journal of Research and Development*, 20(3):198–203, 1976. (Zitiert auf Seite 11)

[Salo8] D. Salomon. *A Concise Introduction to Data Compression*. Undergraduate topics in computer science. Springer-Verlag, pub-SV:adr, 2008. (Zitiert auf Seite 19)

[Say96] K. Sayood. *Introduction to Data Compression*. Morgan Kaufmann, 1996. (Zitiert auf Seite 11)

[Sha48] C. E. Shannon. A mathematical theory of communication. *bstj*, 27:379–423 & 623–656, 1948. (Zitiert auf Seite 11)

[SMo5] Supol, Melichar. Arithmetic Coding in Parallel. *IJFCS: International Journal of Foundations of Computer Science*, 16, 2005. (Zitiert auf den Seiten 43, 51 und 60)

[SMJ99] C. F.-T. S. Mahapatra, J.L. Nunez, S. Jones. Parallel implementation of a multialphabet arithmetic coding algorithm. *Data Compression: Methods and Implementations (Ref. No. 1999/150), IEE Colloquium*, pp. 9/1 – 9/5, 1999. (Zitiert auf Seite 68)

[WNC87] I. H. Witten, R. M. Neal, J. G. Cleary. Arithmetic Coding for Data Compression. *Communications of the ACM*, 30(6):520–540, 1987. (Zitiert auf den Seiten 11 und 24)

Alle URLs wurden zuletzt am 01.01.2011 geprüft.

Zeichenliste

$\bar{T}_x$	zugewiesene Nummer (engl.: tag). Hier die Mitte des Intervalls $H - L$
$\circ$	Verkettung zweier Funktionen
b	binaer
d	dezimal
h	hexadezimal
$\diamondsuit$	Ende des Beispiels
A	Alphabet
P	Wahrscheinlichkeit
$F_x(i)$	Diskrete akkumulierte Verteilungsfunktion fur $X = i$
S	Sequenz
$\|S\|$	Laenge der Sequenz
p	Wahrscheinlichkeit
E	Entropie
$\log_b$	Logarithmus Dualis (Logarithmus zur Basis 2)
B	Basis des Zahlensystems
sc	Skalierungs Grenze
`tot`	Gesamthaeufiggkeit
$\mathtt{f}_i$	Frequenz von a_i
p	Anzahl paralleler Stufen
σ	Anzahl der Schritte ab der Skaliert wird
σ_{par}	Anzahl der Schritte ab der Skaliert wird in paralleler Abarbeitung

Stichwortverzeichnis

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

(Maximiliano Keller)