Adv. Radio Sci., 7, 213–218, 2009
www.adv-radio-sci.net/7/213/2009/
© Author(s) 2009. This work is distributed under
the Creative Commons Attribution 3.0 License.

Advances in Radio Science

Hardwarearchitektur für einen universellen LDPC Decoder

C. Beuschel and H.-J. Pfleiderer

Institut für Allgemeine Elektrotechnik und Mikroelektronik, Universität Ulm, Albert-Einstein-Allee 43, 89081 Ulm, Germany

Abstract. Im vorliegenden Beitrag wird eine universelle Decoderarchitektur für einen Low-Density Parity-Check (LDPC) Code Decoder vorgestellt. Anders als bei den in der Literatur häufig beschriebenen Architekturen für strukturierte Codes ist die hier vorgestellte Architektur frei programmierbar, so dass jeder beliebige LDPC Code durch eine Änderung der Initialisierung des Speichers für die Prüfmatrix mit derselben Hardware decodiert werden kann. Die größte Herausforderung beim Entwurf von teilparallelen LDPC Decoder Architekturen liegt im konfliktfreien Datenaustausch zwischen mehreren parallelen Speichern und Berechnungseinheiten, wozu ein Mapping und Scheduling Algorithmus benötigt wird. Der hier vorgestellte Algorithmus stützt sich auf Graphentheorie und findet für jeden beliebigen LDPC Code eine für die Architektur optimale Lösung. Damit sind keine Wartezyklen notwendig und die Parallelität der Architektur wird zu jedem Zeitpunkt voll ausgenutzt.

1 Einleitung

Low-Density Parity-Check (LDPC) Codes wurden bereits in den 60er Jahren (Gallager, 1963) eingeführt, gerieten aber weitgehend in Vergessenheit und wurden erst in den 90er Jahren (MacKay and Neal, 1997) wiederentdeckt. Wegen ihrer exzellenten Fehlerkorrekturfähigkeit werden sie in vielen Datenübertragungsstandards eingesetzt, wie z.B. in DVB-S2, WiMax oder WiFi.

Die Decodierung von LDPC Codes basiert auf dem Aktualisieren und Versenden von Nachrichten zwischen den sogenannten Prüf- und Variablenknoten eines Bipartiten Graphen. Zur Decodierung der Codes werden üblicherweise teilparallele Architekturen verwendet, da der Datendurchsatz rein sequentieller Architekturen zu niedrig ist während vollständig parallele Architekturen eine zu hohe Verdrahtungskomplexität aufweisen (Blanksby and Howland, 2002). Die Daten, die während der Decodierung mehrfach aktualisiert werden müssen, werden in parallelen Speicherbänken gespeichert. Da die Reihenfolge der Datenzugriffe während der Prüf- und Variablenknotenberechnungen unterschiedlich ist, ist eine der größten Herausforderungen beim Decoder Design der konfliktfreie Zugriff auf die gerade benötigten Daten.

Um konfliktfreien Zugriff zu gewährleisten wird häufig Decoder-Aware Codedesign eingesetzt (Boutillon et al., 2000). Das Codedesign wird dabei jedoch stark eingeschränkt, da die Struktur der Prüfmatrix vielen Randbedingungen genügen muss. Ein Beispiele für strukturierte Codes sind Quasi-Cyclic (QC) Codes mit Decodern wie in (Brack et al., 2007) oder Protograph Codes (Andrews et al., 2007). Bei strukturierten Codes ist nach der Definition eines Codes die Wahl der Parallelität des Decoders weitgehend vorgegeben (Dielissen et al., 2006). Des weiteren lässt sich mit einem entworfenen Decoder im Allgemeinen nur ein ganz bestimmer LDPC Code oder aber spezielle Klasse strukturierter Codes decodieren.

Der im folgenden vogestellte Ansatz für einen Decoder erlaubt konfliktfreie Speicherzugriffe für jeden beliebigen LDPC Code auf derselben Decoderarchitektur. Des weiteren ist die Parallelität der Architektur unabhängig vom Code und beliebig skalierbar. Mit nur einem Decoder Chip lässt sich somit jeder beliebige LDPC Code decodieren.

Der Beitrag gliedert sich wie folgt: Zuerst werden LDPC Codes definiert und ihre Decodierung beschrieben, dann wird eine Mapping Funktion für konfliktfreien Speicherzugriff eingeführt und beispielhaft auf den Bipartiten Graphen eines LDPC Codes angewendet. Danach wird eine voll programmierbare LDPC Decoder Architektur vorgestellt sowie die Reihenfolge der Abarbeitung der Berechnungen anhand der zuvor definierten Mapping Funktion beschrieben und zuletzt wird die vorgeschlagene Architektur mit anderen LDPC Decodern aus der Literatur verglichen.

Correspondence to: C. Beuschel
(christiane.beuschel@uni-ulm.de)

Published by Copernicus Publications on behalf of the URSI Landesausschuss in der Bundesrepublik Deutschland e.V.

2 LDPC Codes: Definition und Decodierung

Ein binärer (N, K) LDPC Code lässt sich über eine sehr dünn besetzte M x N Prüfmatrix $\mathbf{H}$ oder über einen bipartiten Graphen mit M Prüfknoten und N Variablenknoten definieren. Dabei entspricht jeder Prüfknoten eine Zeile und jeder Variablenknoten einer Spalte in $\mathbf{H}$. Prüfknoten m ist genau dann mit dem Variablenknoten n verbunden, wenn für den Eintrag in der Prüfmatrix $h_{mn}=1$ gilt. $M(n)$ ist die Menge der Prüfknoten, die mit Variablenknoten n verbunden ist und $N(m)$ die Menge der Variablenknoten, die mit Prüfknoten m verbunden ist. E ist die Anzahl der Einsen in der Prüfmatrix oder die Anzahl der Kanten im bipartiten Graphen. Im folgenden sei c_n ein gesendetes Codesymbol und r_n ein nach dem Kanal empfangener Wert. Der iterative Belief Propagation (BP) Decodieralgorithmus lässt sich folgendermaßen beschreiben:

1. Initialisierung, $l=0$
   - Intrinsische Werte für alle n zuweisen:
   $$I_n=\ln\frac{P(c_n=0|r_n)}{P(c_n=1|r_n)},\quad S_n^{(0)}=I_n \tag{1}$$
   - Für alle $(m,n)\epsilon\left\{(i,j)|h_{ij}=1\right\}$:
   $$R_{m\to n}^{(l)}=0 \tag{2}$$
2. Iteration $l=l+1$
   - Aktualisierung der Prüfknoten für alle $(m,n)\epsilon\left\{(i,j)|h_{ij}=1\right\}$:
   $$R_{m\to n}^{(l)}=2\cdot\tanh^{-1}\prod_{j\epsilon N(m),j\neq n}\tanh\left(\frac{S_j^{(l-1)}-R_{m\to j}^{(l-1)}}{2}\right) \tag{3}$$
   - Aktualisierung der Variablenknoten für alle $n=0\ldots N-1$:
   $$S_n^{(l)}=I_n+\sum_{i\epsilon M(n)}R_{i\to n}^{(l)} \tag{4}$$
3. Entscheidung:
   - Falls die maximale Anzahl an Iterationen I erreicht wurde wird für alle n decodiert:
   $$q_n=\text{sign}\left(S_n^{(l)}\right) \tag{5}$$
   - Andernfalls: zurück zu 2)

3 Mapping Funktion

In diesem Abschnitt wird eine Mapping Funktion definiert und auf den Bipartiten Graphen von LDPC Codes angewendet. Die Ergebnisse werden in Abschnitt 4 benutzt um eine frei programmierbare LDPC Decoder Architektur herzuleiten, bei der alle Speicherzugriffskonflikte vermieden werden.

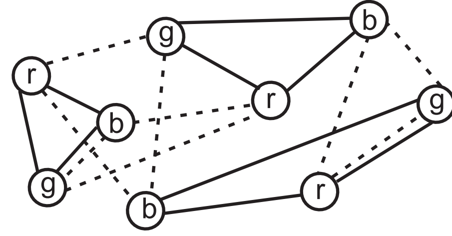

Fig. 1. Beispiel für Mapping mit $E=9$, $p=3$.

3.1 Definition

Gegeben sei die Menge

$$V=\{v_0, ..., v_{E-1}\} \tag{6}$$

mit E Elementen v_i, wobei E faktorisierbar sei zu $E=L\cdot p$. Gegeben seien weiter zwei beliebige Partitionen P und P' auf der Menge V mit

$$P=\{V_0, ..., V_{L-1}\}\quad\text{und}\quad P'=\{V'_0, ..., V'_{L-1}\} \tag{7}$$

wobei jede Untermenge V_i (bzw. V'_j) genau p Elemente aus V enthält. Nach (Tarable et al., 2004) ist es möglich eine Mapping Funktion T zu definieren mit

$$T:\{v_0, ..., v_{E-1}\}\mapsto\{0, ..., p-1\} \tag{8}$$

so dass die folgenden beiden Bedingungen für alle $k=0, ..., L-1$ und alle $i,\ j=0, ..., E-1$ mit $i\neq j$ erfüllt sind:

$$v_i, v_j\in V_k\Rightarrow T(v_i)\neq T(v_j) \tag{9}$$
$$v_i, v_j\in V'_k\Rightarrow T(v_i)\neq T(v_j). \tag{10}$$

Dies bedeutet, dass zwei beliebigen Elemente v_i und v_j, die beide in derselben Untermenge sind, immer auf verschiedenen Werte abgebildet werden können, so dass $T(v_i)\neq T(v_j)$. In (Tarable et al., 2004) wird zum einen bewiesen, dass solch eine Mapping Funktion immer existiert und zum anderen, dass es einen Algorithmus gibt, der eine Mapping Funktion findet.

Obiges Problem lässt sich auch mit dem Einfärbens eines Graphen beschreiben wie in Abb. 1 dargestellt: Der Graph besteht aus einer Menge von $E=9$ Knoten v_i, auf der wie oben beschrieben zwei Partitionen mit $p=3$ definiert sind. Eine Kante verbindet zwei Knoten genau dann, wenn sie Elemente der gleichen Untermenge V_k oder V'_k sind. In der Abbildung sind die Untermengen der Partition P mit durchgezogenen Linien und die Untermengen der Partition P' mit gestrichelten Linien gekennzeichnet. Eine Färbung der Knoten mit p Farben (dargestellt durch die Buchstaben r, g, b) entspricht dem Finden einer geeigneten Mapping Funktion.

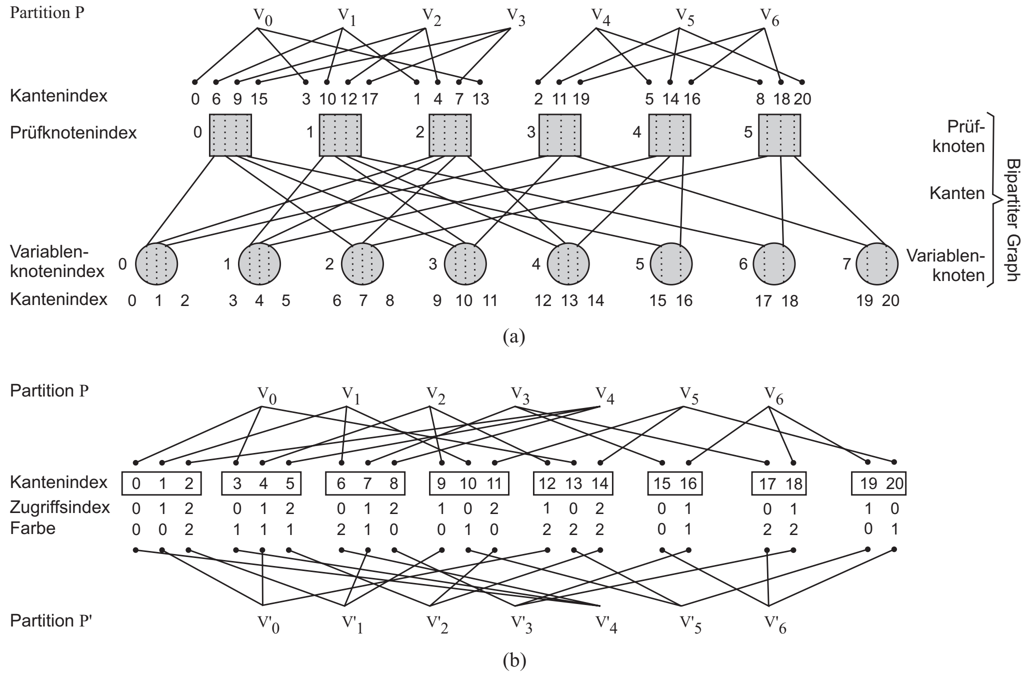

Fig. 2. Beispiel **(a)** für einen bipartiten Graphen **(b)** und die Anwendung der Mapping Funktion

3.2 Anwendung auf den Bipartiten Graphen

Im folgenden wird ein LDPC Code mit N=8, M=6 und $E = 21$ als Beispiel verwendet, dessen bipartiter Graph in Abb. 2a dargestellt ist. Unterhalb der Variablenknoten sind die Kanten in linear aufsteigender Reihenfolge nummeriert und oberhalb der Prüfknoten in permutierter Reihenfolge, die der Permutation der Kanten im bipartiten Graphen entspricht ("Kantenindex").

Nun wird die Mapping Funktion aus Abschnitt 3.1 angewendet: Die Menge V in Gleichung (6) ist die Menge der Kanten im Graphen und für das Beispiel wird p=3, L=7 gewählt. Partition P in Gleichung (7) wird durch die erste Zeile in Abb. 2a beschrieben, d.h. Untermenge V_0 besteht beispielsweise aus den Kanten 0, 3 and 13. Die Partition wird so definiert, dass alle Kanten in einer Untermenge V_k mit verschiedenen Prüfknoten verbunden sind. Die mit einem Prüfknoten verbundenen Kanten sind in aufeinanderfolgenden Untermengen zu finden und zwei beliebige Kanten aus der gleichen Untermenge sind mit unterschiedlichen Variablenknoten verbunden.

In Abb. 2b wurden nur die erste und letzte Zeile aus Abb. 2a beibehalten. Um Kantenindizes, die mit dem gleichen Variablenknoten verbunden sind, wurde ein Rechteck gezeichnet. Nun wird angenommen, dass auf immer p Kanten parallel zugegriffen wird und die Untermengen in P mit aufsteigendem Index durchlaufen werden. Somit lässt sich für die Kanten, die mit demselben Variablenknoten verbunden sind, eine linear aufsteigende Zugriffsreihenfolge von 0 bis $d_V - 1$ festlegen, die in der Zeile "Zugriffsindex" angegeben ist. Die Untermenge V'_k der zweiten Partition P' wird definiert als die Menge der Kanten, die mit den gleichen Variablenknoten wie die Kanten in Untermenge V_k verbunden sind, aber jeweils einen um eins (modulo dem Variablenknotengrad) erhöhten Zugriffsindex haben. Damit besteht die Untermenge V'_0 beispielsweise aus den Kanten 1, 4 und 12. Warum die Definition der Partitionen genau so zu erfolgen hat wird in Abschnitt 4.1 näher erläutert.

Mit den obigen Definitionen wurde eine Menge an Elementen, zwei Partitionen und Werte für p, L definiert. Damit kann nun die Mapping Funktion aus Gleichung (8) angewendet werden. Jede Kante wird mit einer von p Farben (entsprechend einer Ziffer zwischen 0 und $p - 1$) eingefärbt, so dass für alle $i \in \{0, L-1\}$ alle Kanten, die in der gleichen Untermenge V_i oder V'_i sind, unterschiedliche Farben (bzw. Ziffern) bekommen. Eine mögliche Einfärbung ist in der Zeile "Farbe" in Abb. 2b dargestellt.

4 Voll programmierbare LDPC Decoder Architektur

Die vorgeschlagene voll programmierbare LDPC Decoder Architektur ist in Abb. 3 für Parallelität p=3 dargestellt. Die Parallelität der Architektur ist frei skalierbar, so dass Datendurchsatz und Hardwareaufwand gegeneinander ausgetauscht werden können. Anstatt einer abwechselnden Aktualisierung von Prüf und Variablenknoten findet die Aktualisierung der Summen S_n aus Gleichung (4) bereits während der Prüfknotenberechnungen statt. Diese vorgezogene Aktualisierung ist auch aus dem Decoderdesign von QC

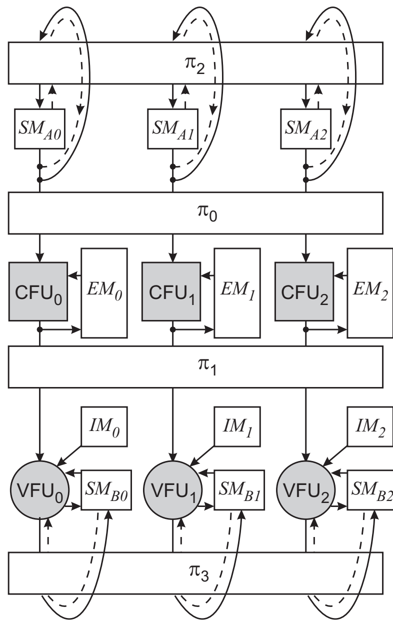

Fig. 3. Vollständig programmierbare LDPC Decoder Architektur, hier für p=3

LDPC Codes bekannt und erlaubt dort bei etwas mehr Hardwareaufwand den doppelten Datendurchsatz.

Die vorgeschlagene Architektur besteht aus p intrinsischen Speicherbänken IM_i für die Werte I_n in Gleichung (1), p extrinsischen Speicherbänken EM_i für die extrinsischen Werte $R_{m\rightarrow n}$ in Gleichung (3), p Summenspeicherbänken SM_{Ai} und p Summenspeicherbänken SM_{Bi} um die vorangegangenen und aktuellen Summenwerte S_n in Gleichung (3) und (4) zu speichern. Alle Speicher werden im dual-port Modus betrieben, so dass pro Takt zwei unabhängige Datenwerte gelesen oder geschrieben werden können. Weiter besteht der Decoder aus p Prüf- und p Variablenknotenberechnungseinheiten (Check / Variable Functional Units, CFU$_i$ und VFU$_i$), die sequentielle Knotenberechnungen ausführen. Pro Takt werden in jedem CFU und in jedem VFU also genau ein Datenwert aktualisiert, so dass insgesamt $2p$ Datenwerte aktualisiert werden. Einige der Datenpfade müssen in geraden Iterationen umgekehrt werden, wie durch die gestrichelten Linien in Abb. 3 angedeutet wird (s. auch Abschnitt 4.1). Neben den dargestellten Komponenten ist für den Decoder noch ein Kontrollmodul notwendig sowie ein Speicher, in dem die aktuelle Prüfmatrix abgespeichert wird. Ein neuer Code kann mit demselben Decoder decodiert werden sobald eine neue Prüfmatrix in den Speicher eingelesen wurde.

4.1 Mapping und scheduling

Die Decodierung eines empfangenen fehlerbehafteten Codeworts beginnt mit der Initialisierung aller Speicher in Abb. 3. Die intrinsischen Speicher IM_i und Summenspeicher SM_{Ai} werden nach Gleichung (1) mit den Werten I_n initialisiert. Wie bereits erwähnt wurden in Abb. 2b alle in den gleichen Variablenknoten mündende Kanten mit einer rechteckigen Box zusammengefasst. Von links nach rechts entsprechen die Boxen den Variablenknoten in aufsteigender Reihenfolge. Die zwei Zeilen tiefer notierte "Farbe" entspricht dem Index der Speicherbank. Der zu Variablenknoten n gehörende intrinsische Wert I_n wird in die Speicherbank geschrieben, die dem Zugriffsindex 0 zugeordnet ist. Demnach wird z.B. I_0 in SM_{A0} und IM_0 abgespeichert und I_4 in SM_{A2} und IM_2. Sowohl die extrinsischen Speicher EM_i als auch die Summenspeicher für die aktuellen Werte SM_{Bi} werden mit Nullen initialisiert.

Während den Iterationen der Decodierung werden die Speicher EM_i linear addressiert wohingegen die Summenspeicher SM eine "zufällige" Addressierung haben, die durch den aktuellen LDPC Code und die Mapping Funktion bestimmt ist. Die Untermengen werden in der Reihenfolge $V_0, V_0', V_1, V_1', \ldots, V_{L-1}'$ durchlaufen. Zum Zeitpunkt t=k werden p Summenwerte, die zu den Variablenknoten mit Kanten in V_k gehören, parallel aus den p Speicherbänken SM_{Ai} gelesen. Über das Permutationsnetzwerk π_2 werden die Werte sofort wieder an die gleichen Adressen aber in andere Speicherbänke zurückgeschrieben. Die Konfiguration des Permutationsnetzwerks π_2 bestimmt sich aus den Farben, die den Variablenknoten in den beiden Untermengen V_k und V_k' zugewiesen wurden. Für t=1 beispielsweise werden entsprechend V_1, V_1' die Werte (S_0, S_2, S_3) aus den Speicherbänken ($SM_{A0}, SM_{A2}, SM_{A1}$) gelesen und permutiert in die Bänke ($SM_{A2}, SM_{A1}, SM_{A0}$) zurückgeschrieben.

Die gelesenen Summenwerte werden außerdem an das Permutationsnetzwerk π_0 weitergegeben, das jedem CFU den passenden Summenwert zuweist. Nachdem in jedes CFU alle d_C zu einem Prüfknoten gehörende Summenwerte eingegangen sind erfolgt die Aktualisierung der Werte, die dann über π_1 permutiert und an die Variablenknoten VFUs weitergegeben werden. π_1 ist dabei genau mit der inversen Permutation zu π_0 konfiguriert.

In den Variablenknoten werden die Werte nach Gleichung (4) aktualisiert. Allerdings treffen die $R_{i\rightarrow n}$ Werte für einen Variablenknoten nicht direkt nacheinander ein sondern in einer "zufälligen" Reihenfolge über die ganze Iteration hinweg verteilt. Deswegen wird die Summe in den SM_{Bi} nur stückweise aktualisiert mit $S_{n,k+1}$=$S_{n,k}$+$R_{i\rightarrow n}$. Der intrinsische Wert wird bei der ersten Teil-Aktualisierung des Summenwertes mit aufaddiert. Lesen und Schreiben der teilaktualisierten Summenwerte in den SM_{Bi} erfolgt in derselben Reihenfolge wie Lesen und Schreiben der SM_{Ai}, lediglich zeitlich verzögert.

Die zweite Iteration beginnt mit einem Vertauschen der beiden Summenspeicher SM_A und SM_B. Dazu können beispielsweise Multiplexer an den Eingängen der Speicher verwendet werden. In der zweiten Iteration werden alle Operationen in umgekehrter Reihenfolge durchgeführt, d.h. die Untermengen werden in der

Table 1. Vergleich von Hardwareaufwand und Datendurchsatz verschiedener programmierbarer LDPC Decoderarchitekturen ([†]Zusätzlich wird eine signifikante Anzahl an Multiplexern benötigt, [‡]Durchschnittswert über eine Menge von Benchmark Codes)

	Masera 2007	Beuschel 2008	Tarable 2004	vorgeschl. Arch.
Perm. π_i	1[†]	2[†]	2	4
VFUs	p	p	p	p
CFUs	p	p	p	p
Speichergr.	$N{+}4E$	$N{+}E$	$N{+}E$	$3N{+}E$
$E=3.5N$	$=15N$	$=4.5N$	$=4.5N$	$=6.5N$
Updates pro Takt	$\approx 0.4\,p$[‡]	$\approx 0.8\,p$[‡]	p	$2p$

Reihenfolge $V'_{L-1}, V_{L-1}, V'_{L-2}, \ldots, V_0$ durchlaufen. Das hat zur Folge, dass einige Datenpfade in umgekehrter Richtung durchlaufen werden müssen wie durch die gestrichelten Linien in Abb. 3 angedeutet ist. Die intrinsischen Werte werden jeweils zusammen mit dem letzten Zweig eines Variablenknotens aufaddiert, so dass hierfür kein zusätzliches Permutationsnetzwerk notwendig ist. Nach der zweiten Iteration befinden sich alle Werte wieder an ihrer ursprünglichen Adresse in ihrer ursprünglichen Speicherbank. Alle weiteren ungeraden Iterationen werden wie Iteration 1 durchgeführt, die geraden Iterationen wie Iteration 2. Sobald ein gültiges Codewort erkannt wird kann die Decodierung beendet werden. Das Codewort wird den Vorzeichenbits der in den SM_{Bi} abgespeicherten Summenwerte entnommen.

Die Definition der zwei Partitionen, die Abbildung der Summenwerte auf die Speicher entsprechend der Mapping Funktion und eine Reihenfolge der Wertaktualisierungen wie oben beschrieben sorgen dafür, dass Speicherzugriffskonflikte vermieden werden. Somit können in jedem Takt für jeden beliebigen LDPC Code $2p$ Werte aktualisiert werden.

Der beschriebene Ansatz kann auch auf einen Layered Decoder erweitert werden. Hierbei lassen sich p Summenspeicher, p intrinsische Speicher sowie zwei Permutationsnetzwerke einsparen. Außerdem werden keine umgekehrten Datenpfade benötigt, stattdessen werden die beiden verbliebenen Permutationsnetzwerke in geraden und ungeraden Iterationen unterschiedlich konfiguriert.

5 Vergleich

In Tab. 1 wird der vorgeschlagene Ansatz mit anderen voll programmierbaren LDPC Decoderarchitekturen verglichen. Während die beiden ersten Ansätze in (Masera et al., 2007) und (Beuschel and Pfleiderer, 2008) einen heuristischen Ansatz für die Mapping Funktion verwenden nutzen die in (Tarable et al., 2004) beschriebene Architektur sowie der von uns vorgestellte Ansatz "optimales" Mapping. "Optimal" soll in diesem Zusammenhang heißen, dass die Parallelität der Architektur jederzeit voll ausgenutzt wird und keine Wartezyklen notwendig sind.

Bei den beiden ersten Designs wird zusätzlich zu den Permutationsnetzwerken eine hohe Anzahl an Multiplexern benötigt. Alle Designs benötigen die gleiche Anzahl an CFUs and VFUs. Die notwendige Größe zum Speichern der Werte $R_{m\mapsto n}$, S_n und I_n ist in Anzahl der zu speichernden Werte angegeben. Für einen durchschnittlichen LDPC Code mit $E = 3.5N$ benötigt der vorgeschlagene Ansatz zwar mehr Speicher als (Beuschel and Pfleiderer, 2008) und (Tarable et al., 2004) aber weniger als die Hälfte im Vergleich zu (Masera et al., 2007).

In (Masera et al., 2007) und (Beuschel and Pfleiderer, 2008) werden heuristische Mapping Algorithmen verwendet, weswegen die Anzahl der aktualisierten Datenwerte pro Takt stark vom betrachteten LDPC Code abhängt und im Allgemeinen deutlich unter p liegt. In (Tarable et al., 2004) werden Prüf- und Variablenknoten abwechselnd aktualisiert. Das Mapping ist optimal und somit können in jedem Takt p Werte aktualisiert werden. Im Gegensatz dazu werden bei der hier vorgeschlagenen Architektur die Variablenknoten bereits während den Prüfknotenberechnungen aktualisiert, so dass sich der Datendurchsatz für die vorgeschlagenen Architektur gegenüber (Tarable et al., 2004) verdoppeln lässt. Die parallele Aktualisierung ermöglicht außerdem die Erweiterung auf einen Layered Decoder, die in (Tarable et al., 2004) nicht möglich ist.

Verglichen mit einem festen Decoder, der nur einen einzigen Code decodieren kann, erfordert die hier vorgeschlagene voll programmierbare Architektur einen höheren Hardwareaufwand. Im Vergleich zu QC Decoderarchitekturen wie z.B. in (Brack et al., 2007) benötigt die vorgeschlagene programmierbare Architektur die gleiche Menge an Speicher. Allerdings werden statt der Shifternetzwerke volle Permutationsnetzwerke benötigt. Um eine zufälligen Prüfmatrix abzuspeichern wird ebenfalls mehr Speicher benötigt als für eine QC $\mathbf{H}$. Wird allerdings die gleiche Parallelität p verwendet, so kann die beschriebene voll programmierbare Decoderarchitektur unabhängig vom Code den gleichen Datendurchsatz wie QC Decoderarchitekturen erzielen.

6 Zusammenfassung

In diesem Beitrag wurde ein voll programmierbare und skalierbare LDPC Decoderarchitektur vorgestellt. Die Architektur ermöglicht die Decodierung eines beliebigen strukturierten oder nicht strukturierten LDPC Codes mit demselben Chip. Eine erneute Initalisierung des Speichers in dem die Prüfmatrix gespeichert wird genügt, um einen anderen LDPC Code decodieren zu können. Datendurchsatz und Hardwareaufwand lassen sich aufgrunden der freien Skalierbarkeit der Architektur beliebig gegeneinander

austauschen. Anhand des bipartiten Graphen wurde ein Mapping und Scheduling Algorithmus vorgestellt, der konfliktfreien Speicherzugriff und damit maximale Parallelität und Datendurchsatz garantiert. Es wurde gezeigt, dass eine Aktualisierung der Variablenknoten während der Prüfknotenberechnungen, die aus dem Design von QC LDPC Decodern bekannt ist, auch bei einer vollständig programmierbare Architektur möglich ist. Das erlaubt die Erweiterung zu einem voll programmierbaren Layered Decoder. Im Vergleich zu anderen programmierbaren Architekturen wird die Parallelität der Architektur optimal ausgenutzt. Somit können mit dem vorgeschlagenen Ansatz bei etwas höherem Hardwareaufwand doppelt so viele Werte pro Takt aktualisiert werden, so dass der doppelte Datendurchsatz erzielt wird.

References

Andrews, K. S., Divsalar, D., Dolinar, S., Hamkins, J., Jones, C. R., and Pollara, F.: The Development of Turbo and LDPC Codes for Deep-Space Applications, Proc. IEEE, 95(11), 2142–2156, 2007.

Beuschel, C. and Pfleiderer, H.-J.: FPGA implementation of a flexible decoder for long LDPC codes, IEEE International Conference on Field Programmable Logic and Applications, FPL, 185–190, 2008.

Blanksby, A. and Howland, C.: A 690-mW 1-Gb/s 1024-b, rate-1/2 low-density parity-check code decoder, IEEE J. Solid-St. Circ., 37(3), 404–412, 2002.

Boutillon, E., Castura, J., and Kschischang, F.: Decoder-first code design, in: 2nd International Symposium on Turbo Codes and Related Topics, 459–462, 2000.

Brack, T., Alles, M., Lehnigk-Emden, T., Kienle, F., Wehn, N., L'Insalata, N., Rossi, F., Rovini, M., and Fanucci, L.: Low Complexity LDPC Code Decoders for Next Generation Standards, Design, Automation and Test in Europe, DATE, 331–336, 2007.

Dielissen, J., Hekstra, A., and Berg, V.: Low Cost LDPC Decoder for DVB-S2, Design, Automation and Test in Europe, DATE, 2006.

Gallager, R.: Low-Density Parity-Check Codes, Cambridge, MA, MIT Press, 1963.

MacKay, D. and Neal, R.: Near Shannon limit performance of low-density parity-check codes, Elect. Lett., 33(6), 457–458, 1997.

Masera, G., Quaglio, F., and Vacca, F.: Implementation of a Flexible LDPC Decoder, IEEE T. Circ. Sys. II, Express Briefs, 54(6), 542–546, 2007.

Tarable, A., Benedetto, S., and Montorsi, G.: Mapping interleaving laws to parallel turbo and LDPC decoder architectures, IEEE T. Inform. Theory, 50(9), 2002–2009, 2004.