Studienarbeit

Wavelet-Bildkompression
auf dem adaptiven Rechner ACE-V

von cand. inform. Hagen Gädke

Abteilung Entwurf Integrierter Schaltungen (E.I.S.)
Prof. Dr. U. Golze

Betreuer: Dr. Andreas Koch

November 2003

Erklärung

Ich erkläre, die vorliegende Arbeit selbständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

Braunschweig, den 06.11.2003

Inhaltsverzeichnis

1	**Einleitung**	**6**
2	**Anwendung und Entwurfsaspekte**	**7**
2.1	Die diskrete Wavelet-Transformation	7
2.1.1	Phase 1: mehrstufige Wavelet-Transformation	8
2.1.2	Phase 2: Quantisierung	11
2.1.3	Phase 3: Lauflängencodierung	12
2.1.4	Phase 4: Huffman-Codierung	13
2.2	Die Software-Implementierung	13
2.2.1	Beschreibung der Software-Implementierung	13
2.2.2	Effizienzsteigernde Implementierungs-Änderungen	15
2.3	Definition der Anforderungen für die Studienarbeit	16
2.4	Partitionierung	16
3	**Der Software-Teil der Anwendung**	**18**
4	**Die Hardware-Software-Schnittstelle**	**21**
4.1	Slave-Mode-Zugriffe	21
4.2	Master-Mode-Zugriffe	21
5	**Der Hardware-Teil der Anwendung**	**23**
5.1	Pipelining	24
5.1.1	Wavelet-Pipeline	24
5.1.2	QZH-Pipeline	25
5.2	Speicherorganisation	28
5.2.1	Speicherzugriffe während der Transformationen	28
5.2.2	Aufteilung des Speichers für MARC	35
5.3	Stream-Flusskontrolle	37
5.4	Der Hardware-Teil im Überblick	39
5.5	Detaillierte Modulbeschreibungen	42
5.5.1	wavelet_8_4_l	42
5.5.2	wavelet_16_2_hl	46
5.5.3	wavelet_controller	51
5.5.4	quantization	55
5.5.5	zle	57
5.5.6	huffman	60
5.5.7	huffman_lookup	63
5.5.8	qzh	64
5.5.9	qzh_controller	66
5.5.10	min_max	69
5.5.11	compare_16b_neg	70
5.5.12	parameter_storage	71
5.5.13	result_params_mux	72
5.5.14	result_controller	73
5.5.15	read_flow_control	75
5.5.16	write_flow_control	79
5.5.17	user	84

6 Testumgebung 89
6.1 Funktionen des Testrahmens stimulus 89
6.2 Kombinierter HW/SW-Test 91

7 Benchmark-Ergebnisse 92
7.1 Signal- zu Rauschverhältnis und Bits pro Pixel 92
7.2 Bildvergleiche für unterschiedliche Komprimierungsstärken 95
7.3 Laufzeit und Fläche 98

8 Zusammenfassung und Ausblick 100

9 Abkürzungsverzeichnis und Begriffserklärungen 102

10 Literaturverzeichnis 104

Anhänge auf CD:
A1 Anhang 1: optimierter Algorithmus zur Wavelet-Bildkompression
A2 Anhang 2: angepasster Algorithmus zur Dekompression
A3 Anhang 3: Software- und Hardware-Teil der Anwendung für die ACE-V

1 Einleitung

Adaptive Rechner bieten Geschwindigkeitsvorteile gegenüber Standard-CPUs. Die Hardware eines adaptiven Rechners hat zusätzlich zu einem herkömmlichen Prozessor einen Bereich, in dem sich eine rekonfigurierbare Komponente (engl.: „reconfigurable component“ oder kurz „RC“) befindet. Dieser Teil der Hardware ist nicht fest bestimmt, sondern kann im laufenden Betrieb geändert werden. So ist es möglich, zeitkritische Berechnungen in Hardware auszulagern, die auf die aktuelle Anwendung maßgeschneidert ist. Anwendungen, die weniger Rechenleistung benötigen (z. B. administrative Aufgaben), können jedoch weiterhin in Software ausgeführt werden. So büßt ein adaptiver Rechner nichts an Flexibilität ein.

Da das Konzept adaptiver Rechner recht neu ist und sich noch in intensiver Forschung befindet, sind bislang vergleichsweise wenige anspruchsvolle Anwendungen auf diesem Gebiet verfügbar. Entsprechend schwierig ist es, aussagekräftige Laufzeit-Vergleichswerte bereitzustellen.

Die Anwendung, die in dieser Studienarbeit auf dem adaptiven Rechner ACE-V entworfen wurde, ist ein Algorithmus zur verlustbehafteten Wavelet-Komprimierung von Graustufenbildern fester Größe. Im Bereich der Wavelet-Komprimierung existiert für adaptive Rechner bislang keine dokumentierte Implementierung, was diese Anwendung sehr interessant macht. Anspruchsvoll ist sie in dem Sinne, dass jeder Eingabedatenstrom in zwei Ausgabeströme resultiert, die in ihrer vollen Länge zwischengespeichert werden, um anschließend erneut als Eingabe zu dienen. Mehrere Berechnungsphasen müssen durchlaufen werden, bevor das Ergebnis vorliegt.

Als Benchmark für adaptive Rechner soll diese Studienarbeit Vergleichswerte für das verwendete System liefern. Dadurch ist es nicht nur möglich, Standardrechner mit adaptiven Rechnern zu vergleichen, sondern auch verschiedene adaptive Rechner untereinander zu messen. Da der C-Quellcode der verwendeten Anwendung frei zugänglich ist, kann dieser Benchmark auch Aussagen treffen über die Effizienz von automatisch generierten Hardwarelösungen. Als Beispiel sei der Compiler NIMBLE [3], [8] genannt, der aus beliebigem C-Code automatisch Soft- und Hardwareteil der Anwendung für einen adaptiven Rechner generiert.

Da für diese Anwendung sehr unterschiedliche und sehr schnelle Speicherzugriffe notwendig sind, wird als Speicherzugriffssystem das in der Abteilung E.I.S. entwickelte System MARC (Memory Architecture for Reconfigurable Computers) [2] eingesetzt.

Nach den in Kapitel 2 beschriebenen Entwurfsvoraussetzungen wird auf den Software-Teil der Anwendung eingegangen (Kapitel 3), danach wird in Kapitel 4 die Hardware-Software-Schnittstelle beschrieben und in Kapitel 5 der Hardware-Teil erläutert. Kapitel 6 fasst wichtige Bemerkungen zur Testumgebung zusammen, bevor Kapitel 7 einige Benchmark-Ergebnisse und Laufzeitvergleiche vorstellt. Eine Zusammenfassung findet sich schließlich in Kapitel 8.

2 Anwendung und Entwurfsaspekte

Kapitel 2 enthält einige grundlegende Informationen zur Komprimierung von Bilddaten mittels der diskreten Wavelet-Transformation. Ferner wird der in Software gegebene Algorithmus vorgestellt, der die Grundlage für den Entwurf bildet. Danach werden die Anforderungen an die Studienarbeit definiert, und der letzte Punkt geht auf die Partitionierung der Anwendung in Hardware und Software ein.

2.1 Die diskrete Wavelet-Transformation

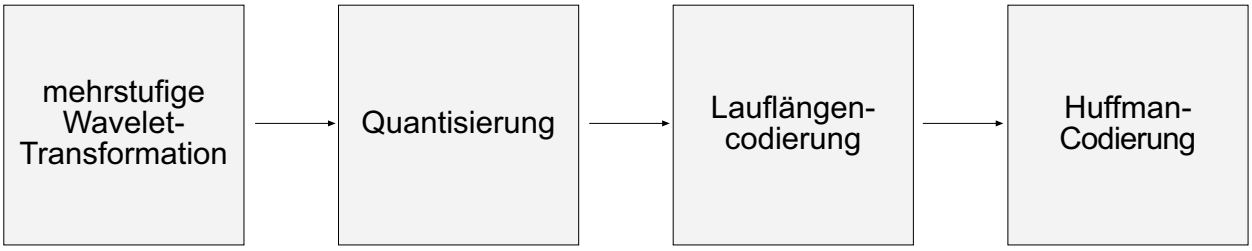


Bild 2.1:
Typische Verarbeitungsreihenfolge bei einer Bilddatenkomprimierung mittels Wavelet-Transformation.

Typischerweise geschieht die Wavelet-Komprimierung eines Bildes in vier Phasen (siehe Bild 2.1). Zuerst erfolgt die mehrstufige Wavelet-Transformation der Bilddaten. Hier wird das Ursprungsbild aufgeteilt in einen Datenblock mit etwas vergröberten Bildinformationen und einen anderen Block, in dem die Feinheiten stehen, die bei der Vergröberung im ersten Block verloren gegangen sind. „Mehrstufig“ bedeutet, dass dieses Verfahren auf die beiden entstandenen Datenblöcke erneut angewendet wird und so fort.
Die so entstandenen Datenblöcke werden nun der Quantisierung zugeführt. Die Quantisierung hat zwei Aufgaben: einerseits ordnet sie jeweils mehreren Eingabewerten den gleichen Ausgabewert zu (was zu einer kompakteren Zahlendarstellung führt). Andererseits markiert sie Eingaben, die einen besdtimmten Mindestbetrag nicht übersteigen. Interessant ist, dass in dem hier verwendeten Verfahren der Benutzer des Programms die Höhe dieses Mindestbetrages über die Kommandozeile mit der Eingabe der Komprimierungsstärke regeln kann. Je höher die Komprimierungsstärke, um so höher der Mindestbetrag, und damit steigt die Effektivität der Komprimierung; leider aber auch der Qualitätsverlust.
Die nun folgende Lauflängencodierung nutzt Korrelationen zwischen Eingabedaten aus, um verlustfrei zu effizienteren Darstellungen zu gelangen. Das simpelste Beispiel ist, mehrfach hintereinander auftretende Werte zu zählen und als Ergebnis nur die Anzahl dieser gezählten Werte auszugeben. Wie stark ein Datenstrom dadurch komprimiert wird, hängt von der Beschaffenheit der Eingangsdaten ab.
Die so codierten Daten werden zuletzt einer Huffman-Codierung unterzogen. Der Sinn dieser ebenfalls verlustfreien Codierung ist, häufig auftretende Datenworte mit kürzeren Bitfolgen zu codieren als seltener auftretende. So bringt man die gleiche Menge an Informationen in einem kürzeren Ausgabedatenstrom unter.

Die Funktionsweise der vier Phasen soll in den Abschnitten 2.1.1 bis 2.1.4 an Beispielen erläutert werden.

2.1.1 Phase 1: mehrstufige Wavelet-Transformation

Es wurde bereits angedeutet, dass bei der Wavelet-Transformation ein eingehendes Signal in einen Datenblock mit groben Bildinformationen und in einen zweiten mit den dabei verlorengegangenen Feinheiten aufgeteilt wird. Wir betrachten dazu das Beispiel in Bild 2.2.

Eingangssignal:	13	13	5	5	9	13	21	17
Ausgang 1 (Grobinformationen):	13		5		11		19	
Ausgang 2 (Detailinformationen):	0		0		-2		2	

Bild 2.2:
Beispiel für die Aufteilung des Eingangssignals in Grob- und Detailinformationen.

Die Grobinformationen (Ausgang 1) des Eingangssignals entsprechen in diesem Beispiel schlicht dem Mittelwert von zwei paarweise aufeinanderfolgenden Eingangsdaten. Ausgang 2 stellt die notwendigen Daten zur Verfügung, um aus den Grob-Informationen das Eingangssignal wieder zurückgewinnen zu können (z. B. ist 11 + (-2) = 9 und 11 - (-2) = 13). Wir bemerken, dass für n Eingangspixel die Ausgänge 1 und 2 je n / 2 Ausgangswerte produzieren. Insgesamt bleibt die Datenmenge also gleich.

Man bezeichnet den Datenblock mit Grob-Informationen auch als *Mittelwert-Teil* oder *Low-Pass-Teil* („*LP*“), die Detailinformationen nennt man auch *Differenz-Teil* oder *High-Pass-Teil* („*HP*“). In Bild 2.2 wurde das einfachste mögliche Beispiel angegeben; die konkrete Berechnungsvorschrift für die Ausgänge 1 und 2 hängt jedoch von der verwendeten Wavelet-Basis ab (siehe z. B. [4]).

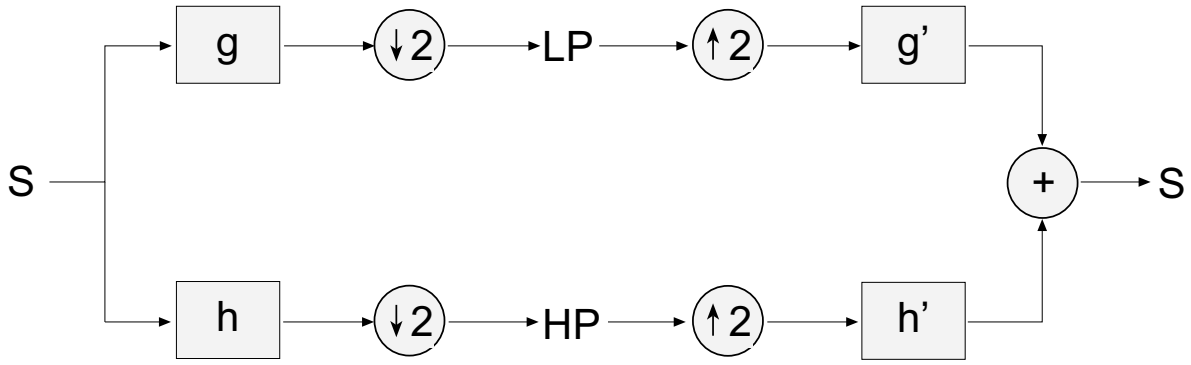


Bild 2.3:
Aufteilung des Eingangssignals in Grob- und Detailinformationen.

Allgemeiner entstehen LP und HP durch Anwendung eines LP-Filters g bzw. HP-Filters h auf ein Eingangssignal S, mit nachgeschaltetem Downsampling (die „2“ mit dem abwärts zeigenden Pfeil davor), siehe Bild 2.3. Das Downsampling entspricht der oben genannten „paarweisen“ Verarbeitung und sorgt dafür, dass LP und HP je halb so lang sind wie das Eingangssignal. Die Rückgewinnung von S geschieht durch Upsampling mit anschließender Anwendung der inversen Filter g' bzw. h', wonach S aus den beiden Teilkomponenten wieder zusammengesetzt werden kann.

Bisher haben wir die Wavelet-Transformation von eindimensionalen Daten beschrieben, ein Bild ist aber zweidimensional. Wie die Transformation eines Bildes gehandhabt wird, zeigt Bild 2.4. In einem ersten Schritt wird jede Bildzeile des Bildes S einzeln einer eindimensionalen Transformation unterworfen, wobei für jede Zeile ein LP-Teil g(S) und ein HP-Teil h(S) anfallen, die jeweils halb so lang wie die Ursprungszeile sind. Unter Beibehaltung der Zeilenreihenfolge werden diese Datenblöcke wie im Bild gezeigt angeordnet (g(S) links, h(S) rechts). Das daraus resultierende, zusammengesetzte Bild wird nun erneut transformiert,

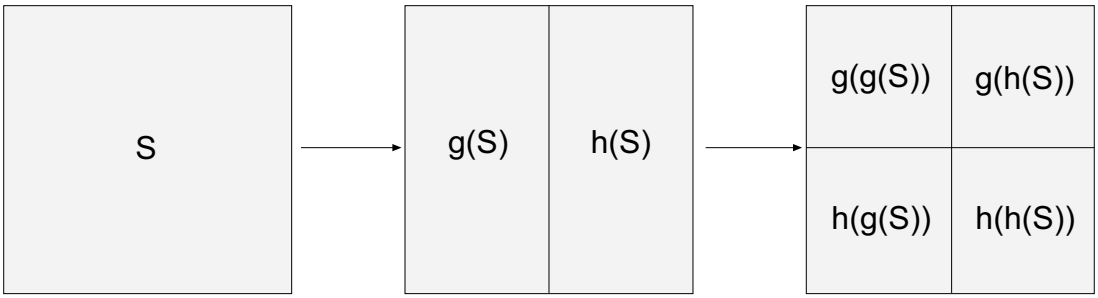


Bild 2.4:
Wavelet-Transformation auf zweidimensionalen Daten.

diesmal jedoch spaltenweise. Der LP-Teil (g(g(S)) und g(h(S))) wird oben, der HP-Teil (h(g(S)) und h(g(S))) unten angeordnet. Die so entstandenen vier Datenblöcke sind das Ergebnis einer (zweidimensionalen) Transformationsstufe.

Ziel der Transformation ist es, einen möglichst kleinen Datenblock mit den wichtigsten Bildinformationen zu erhalten neben anderen, größeren Blöcken, die unwichtiger sind und stärker komprimiert werden können. Die „wichtigen“ Bildinformationen stecken jeweils in dem LP-Teil der Transformationen. In der zweidimensionalen Betrachtungsweise entspricht dies dem Teil g(g(S)). Man wendet also eine weitere Transformationsstufe auf den Bildausschnitt g(g(S)) an. Man kann nach diesem Schema theoretisch solange fortfahren, bis der finale LP-Teil aus nur noch einem Pixel besteht. In der Praxis erzielt man aber mit vier und mehr Stufen keine weiteren Komprimierungsvorteile mehr, weshalb man meist bei drei Transformationsstufen bleibt. Bild 2.5 zeigt die Verschachtelung der drei Stufen.

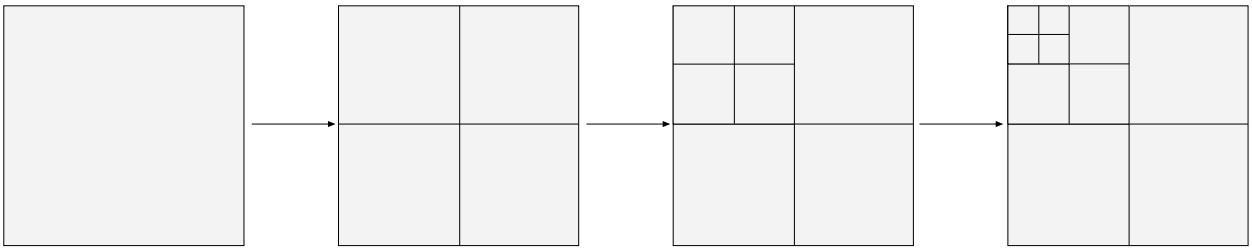


Bild 2.5:
Verschachtelungs-Schema für drei Wavelet-Transformationsstufen.

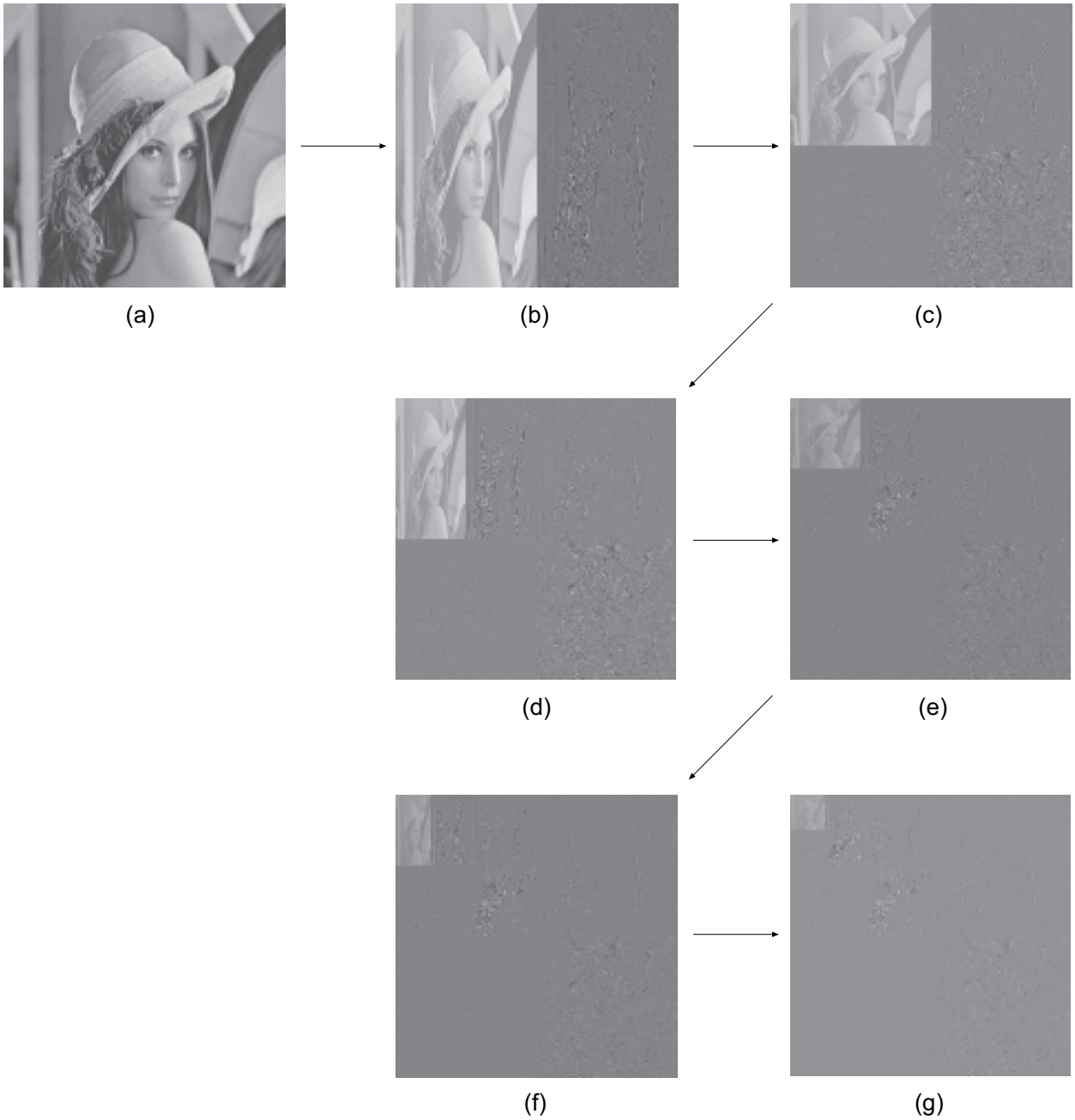


Bild 2.6:
Dreistufige Wavelet-Transformation an einem Beispielbild. (a) Originalbild, (b) und (c) Stufe 1, (d) und (e) Stufe 2, (f) und (g) Stufe 3. (g) ist das Ausgabebild der kompletten Wavelet-Transformation.

Um einen Eindruck von den Inhalten der 10 entstandenen Datenblöcke zu erhalten, werfen wir einen Blick auf Bild 2.6. Man erkennt im LP-Teil des Bildes stets das Originalbild wieder; nach der dritten Transformationsstufe ist es nur noch ganz klein oben links zu sehen.
Diese 10 Datenblöcke sind die Eingabe für die nächste globale Bearbeitungsphase: die Quantisierung.

2.1.2 Phase 2: Quantisierung

In der Quantisierung werden jeweils mehrere unterschiedliche Eingabedaten auf das gleiche Ausgabedatum abgebildet. Dadurch ergeben sich weniger mögliche Ausgabewerte, was die Codierung der einzelnen Ausgaben mit weniger Bits gegenüber den Eingaben ermöglicht. Wir betrachten als Beispiel die 6-Bit-Eingabedaten

0 1 5 6 7 35 57 61 62 63

Angenommen, die Quantisierung produziert aus diesen Eingabedaten Ausgabewerte der Länge 4 Bit. Die Werte 0..3 würden dann dem Ausgabewert 0 zugeordnet, 4..7 würden zu 1 codiert usw. Obige Beispielwerte würden zu

0 0 1 1 1 8 14 15 15 15

quantisiert. Es wird unmittelbar klar, dass hier ein Datenverlust auftritt, denn eine Dequantisierung würde

1 1 5 5 5 33 57 61 61 61

ergeben, also lediglich eine Annäherung an die ursprünglichen Daten. Dieser Verlust wird in Kauf genommen.

Für eine effiziente Quantisierung müssen das absolute Minimum und Maximum der zu bearbeitenden Daten bekannt sein. Wegen der unterschiedlich gearteten Ausgabeblöcke der Wavelet-Transformation macht es Sinn, jeden Block einzeln zu quantisieren. Die Eingabe für diese Bearbeitungsphase besteht also aus 10 verschieden großen Wavelet-Blöcken mit zugehörigen Block-Minima und -maxmima. Jene Block-Extremwerte werden später zur Dequantisierung gebraucht, und sind daher ein wichtiger Teil der Ergebnisparameter der gesamten Komprimierung.

Die in dieser Studienarbeit verwendete Bearbeitungsreihenfolge der Datenblöcke wird in Bild 2.7 gegeben.

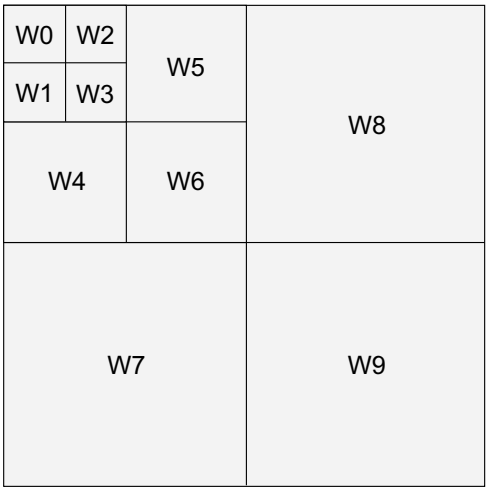


Bild 2.7:
Bearbeitungsreihenfolge der Wavelet-Datenblöcke in der Quantisierung.

Eine Besonderheit in der hier verwendeten Methode ist, dass die drei größten Detailblöcke, W7, W8 und W9 (entstanden in Wavelet-Transformationsstufe 1) vernachlässigt, d. h. zu Null gesetzt, werden. Dies macht die Komprimierung sehr viel effektiver, zu einem vergleichsweise geringen Preis in punkto Qualitätsverlust.

Jedem der 7 tatsächlich bearbeiteten Datenblöcke wird ein Block-Grenzwert zugeordnet. Werte, die betragsmäßig kleiner sind als dieser Blockgrenzwert, werden von der Quantisierung besonders markiert

(ein spezieller Ausgabewert dient als Markierung). Dies hat den Zweck, „kleine Details“ des Bildes fortzulassen und ermöglicht in der weiteren Verarbeitung eine sehr effiziente Lauflängencodierung. Der Benutzer kann über einen Kommandozeilenparameter die Höhe der Blockgrenzwerte festlegen und so direkt Einfluss nehmen auf die Stärke der Komprimierung und den damit verbundenen Qualitätsverlust.

2.1.3 Phase 3: Lauflängencodierung

In der Quantisierung wurden einige Ausgabedaten speziell markiert. Diese markierten Daten wollen wir im folgenden mit „ZERO“ bezeichnen (nicht zu verwechseln mit dem tatsächlichen Wert „0“!). Die Idee der Lauflängencodierung ist, Folgen von gleichen Werten zu erkennen und ihre Länge zu messen. Ausgegeben wird anstatt jedes einzelnen Wertes die Anzahl der direkt aufeinander folgenden Werte. Konzeptionell sollte die Eingabe

4 0 0 0 0 0 12 15 7 8

also die Ausgabe

4 „5 mal die 0“ 12 15 7 8

erzeugen. Wir bezeichnen hier die „0“ als Grundwert und die „5“ als Lauflänge. Die Frage ist nun, wie man den Wert „5 mal die 0“ im Ausgabestrom codiert. Sicherlich kann man nicht einfach die Ausgaben „5“ und „0“ schreiben, denn diese Symbole sind bereits für die tatsächlichen Werte 5 und 0 vergeben.

Eine mögliche Codierung besteht darin, einen Teil des Ausgabewortes zur Darstellung des Grundwertes und den anderen Teil zur Lauflängendarstellung zu verwenden. Da die Quantisierung Ausgaben der Breite 4 Bit produziert, würde man dann für eine angenommene Länge der Lauflängen-Ausgabeworte von 8 Bit die ersten 4 Bit zur Grundwertcodierung benutzen und die übrigen 4 Bit zur Lauflängencodierung. Darstellen könnte man damit Läufe bis zur Länge 16.

Um längere Läufe codieren zu können, müsste man weitere Bits spendieren. Man könnte dafür das erste Bit jedes Ausgabewortes als Switch benutzen, um für die aktuelle Ausgabewortlänge von 8 auf 16 Bit umzuschalten oder nicht.

In einem Test zur Evaluierung eines passenden Verfahrens wurden die maximal auftretenden Lauflängen - gemittelt über die drei Testbilder aus Kapitel 7.2 - gemessen. Legt man die Standard-Komprimierungsstärke 128 zugrunde, ergibt sich für die Eingabewerte 0 bis 15 eine mittlere maximale Lauflänge von 3,25; für die ZERO-Werte jedoch ergab der Test eine mittlere Maximallänge von 762.

Da ZERO-Läufe (bedingt durch die Quantisierung) anscheinend überproportional häufiger auftreten als andere Läufe, ist es insgesamt effizienter, ZERO-Läufe bevorzugt effizient zu codieren. Die in dieser Studienarbeit verwendete Methode ist, nur ZERO-Läufe zu codieren; dadurch spart man nämlich die Codierung der Grundwerte ein. Das Verfahren wird deswegen ab jetzt nicht mehr „Lauflängencodierung“, sondern „ZERO-Längencodierung“ genannt. Die Tatsächlich verwendete Ausgabewortgröße beträgt 8 Bit, in denen zusätzlich zu den Werten 0 bis 15 die Codes 16 bis 255 für die Codierung der ZERO-Lauflänge verwendet werden.

Als Abkürzung für den Begriff „ZERO-Längencodierung“ wird im folgenden häufig „ZLE“ verwendet (engl.: „zero length encoding“).

2.1.4 Phase 4: Huffman-Codierung

Die Idee dieser letzten, verlustfreien Komprimierungsphase ist, häufig auftretende Werte mit kurzen Bitfolgen zu codieren, während man für seltener auftretende Werte lange Bitfolgen einsetzt. Angenommen, wir hätten folgenden Datenstrom aus 4 Worten zu je 8 Bit:

11110000 11110000 11110000 10100110

Mit der Huffman-Tabelle in Bild 2.8 ergäbe das die Ausgabe

0010 0010 0010 01001100111

Aus 36 Bits wurden auf diese Art und Weise 23 Bits.

Die Implementierung einer Huffman-Codierung wird hardwarefreundlicher, wenn die Huffman-Tabelle fest vorgegeben ist, da dann nur ein kompletter Durchlauf durch die Eingabedaten notwendig ist. So wird in dieser Studienarbeit eine feste Tabelle verwendet, die sich für dieses Verfahren bereits bewährt hat.

Eingabe	Ausgabe
⋮	⋮
11110000	0010
⋮	⋮
10100110	01001100111
⋮	⋮

Bild 2.8:
Ausschnitt einer Huffman-Tabelle.

Beliebig geartete Daten mit einer festen Tabelle zu codieren würde natürlich keinen Sinn machen. Wir haben hier aber bereits eine Quantisierung und eine Zerolängencodierung vorgeschaltet, wodurch wir glücklicherweise Aussagen über die Auftrittswahrscheinlichkeit bestimmter Wertebereiche machen können. Die Blockgrenzwerte in der Quantisierung setzen typischerweise entweder sehr wenige aufeinanderfolgende oder alle Werte des aktuellen Blocks zu ZERO. Daher treten für die Huffman-Codierung neben den ohnehin häufigen quantisierten Pixelwerten 0 bis 15 (die von ZLE unangetastet bleiben) noch Eingabewerte von 16 und knapp darüber relativ häufig auf. Je größer die Eingabewerte nun werden, umso unwahrscheinlicher treten sie auf, denn es ist sicher wahrscheinlicher, dass ein Wert zufällig n-mal in Folge als m-mal in Folge auftritt, mit m > n. Eine Ausnahme bildet hier jedoch - wie zuvor angedeutet - der Wert 255. Wurde in der Quantisierung ein ganzer Datenblock mit ZERO markiert, so zählt ZLE entsprechend viele ZERO-Folgen der Länge 255 (längere Folgen werden bei 255 „abgeschnitten“, da keine weiteren Bits zur Codierung von größeren Längen zur Verfügung stehen). Diese Überlegungen zu den Auftrittswahrscheinlichkeiten lassen sich anhand der tatsächlich in dieser Arbeit verwendeten Huffman-Tabelle anschaulich nachvollziehen (siehe Anhang 1).

2.2 Die Software-Implementierung

Der auf dem adaptiven Rechner ACE-V ([5]) zu entwerfende Algorithmus liegt bereits in Software vor, geschrieben in der Programmiersprache C (siehe [1]). Im Abschnitt 2.2.1 wird die Implementierung des Algorithmus zum besseren Verständnis erläutert. In 2.2.2 sind die Änderungen beschrieben, die an der Software vorgenommen wurden, um sie effizienter und besser hardwarerealisierbar zu machen.

2.2.1 Beschreibung der Software-Implementierung

Die Implementierung orientiert sich sehr stark an den in 2.1.1 bis 2.1.4 beschriebenen Bearbeitungsphasen.
Zunächst findet eine Initialisierung (initialize(argc, argv)) statt. Hier werden die Kommandozeilenparameter eingelesen und interpretiert, bestehend aus dem Eingabebild im PGM-Format (siehe z. B. [6]) und der Komprimierungsstärke. Die Komprimierungsstärke ist eine Ganzzahl zwischen 0 und 255. Je größer dieser

Wert ist, umso kleiner wird die komprimierte Datei, umso größer wird aber auch der damit verbundene Qualitätsverlust. Die Komprimierungsstärke legt nämlich die Höhe der Blockgrenzwerte (siehe 2.1.2) fest. Für eine Stärke von 128 (Default-Wert) werden die Blockgrenzwerte {0, 39, 27, 104, 79, 50,191, 99999, 99999, 99999} verwendet (geordnet nach Blocknummer). Der finale LP-Teil der mehrstufigen Transformation (Block W0) erhält den Grenzwert 0, was dazu führt, dass für beliebige Kompressionsstärken dieser Block immer den Wert 0 zugeteilt bekommt. Die Komprimierungsstärke (gespeichert in der Variablen cratio) skaliert die für den Default-Wert gegebenen Grenzwerte blockthresh nämlich linear:

```
for (i = 0; i < 9; i++)
    blockthresh[i] = (blockthresh[i] * cratio) >> 7;
```

Dies ist sehr sinnvoll, da auf diese Weise der LP-Teil (welcher sehr wichtige Informationen hält) keinem zusätzlichen Datenverlust unterworfen wird.

Als nächstes werden in der Methode read_image(in_name) die Bilddaten byteweise aus der PGM-Datei eingelesen und in ein Integer-Array kopiert. Die Verwendung des Datentyps Integer ist essentiell, da in der folgenden Wavelet-Transformation die transformierten Werte ein größeres Spektrum einnehmen als die Argumente. 8 Bit pro Wert reichen dafür nicht aus.

forward_wavelet() führt nun die Wavelet-Transformation durch. Die Schleife

```
for (nt=512;nt>=BLOCK_SIZE;nt>>=1) {
    for (i=0;i<nt*512;i+=512) fcdf22(&int_data[i],nt,ROW);
    for (i=0;i<nt;i++) fcdf22(&int_data[i],nt,COL);
}
```

ist zwar im Original für die Verarbeitung von 512x512-Pixel-Bildern geschrieben, kann aber leicht verallgemeinert werden für Bilder der Größe $2^n x 2^n$. Durchlaufen wird die Schleife dann dreimal, was der dreistufigen Transformation entspricht; in jeder Stufe erfolgen zunächst die Zeilen- und anschließend die Spaltentransformationen, jeweils ausgelagert in die Methode fcdf22. In jener Methode halten die Integer-Arrays s[] und d[] die transformierten Werte. s nimmt die Low-Pass-Werte auf, d die High-Pass-Werte.

quantization() ist für die Quantisierung verantwortlich. Die Wavelet-Blöcke werden hier nacheinander durchlaufen. Für jeden Block findet zunächst die Berechnung der Blockminima und -maxima statt, dann folgt die Bestimmung der Quantisierungs-Grenzwerte thresh1 bis thresh16. Falls ein Argument betragsmäßig kleiner als der zu diesem Block gehörige Blockgrenzwert ist (siehe 2.1.2), erfolgt die Markierung mit „ZERO_MARK“ (hier wird der Wert 16 für diese Markierung benutzt), ansonsten wird das Argument seinem Quantisierungsbereich zugeordnet (Methode classify(int val)). Die Blockminima und -maxima werden zur späteren Ausgabe nach der Quantisierung des aktuellen Blocks gesichert.

Die Zerolängencodierung RLE_encode() arbeitet ebenfalls auf jedem Block einzeln. Die einzige hier zunächst nicht sehr anschauliche Quelltextzeile lautet:

```
if ((count == 256 - ZERO_MARK) || (i == img_size)) break;
```

Sie bedeutet, dass die aktuelle Lauflängenzählung beendet wird, wenn der aktuelle Lauf bereits seine maximal erlaubte Länge (siehe 2.1.3) erreicht hat (count == 256 - ZERO_MARK) oder das Bildende erreicht wurde (i == img_size).

Die Huffman-Codierung findet in der Methode entropy_encode() statt; das wesentliche hieran ist die Untermethode hufenc(unsigned char ich, int *nb). Mit

```
nbits   = HufSize[ich];
val     = HufVal[ich];
```

wird sowohl die codierte Bitfolge val des Arguments ich in der Huffman-Tabelle HufVal nachgeschlagen

als auch die Länge nbits dieser Bitfolge in der Tabelle HufSize. Wir bemerken, dass hier zwei Tabellen à 256 Einträge nötig sind, die im Programmkopf der Quelltextdatei compress.c definiert wurden. Die Bitfolgen werden dann Bit für Bit dem Ausgabestrom hinzugefügt, wobei dessen aktuelle Bitlänge in nb gesichert wird.

write_compressed_file(in_name) bildet den Abschluss des Verfahrens. Der codierte Bitstrom wird in eine Ausgabedatei geschrieben, mit einem vorangehendem, binärem Header, der wichtige Informationen zur Decodierung beinhaltet. Am Anfang des Headers stehen die Blockminima und -maxima (7 + 7 Integer-Werte), danach folgen die Größen der ZLE-codierten Blöcke (7 Integer-Werte) und in einem letzten Integer-Wert die Länge des codierten Bitstroms in Bytes. Die Headergröße beträgt demnach (7 + 7 + 7 + 1) * sizeof(int) = 88 Bytes.

2.2.2 Effizienzsteigernde Implementierungs-Änderungen

Da die vorgestellte Implementierung sehr verschwenderisch mit Ressourcen umgeht (für die Ausgabe der vier Berechnungsphasen wird jeweils eigener Speicher reserviert), wurde sie vor dem Hardware-Entwurf optimiert (diese Optimierungen sind nicht Teil der Studienarbeit). So wurden die vier Speicherfelder int_data, quant_buf, rle_buf und codep auf die beiden Felder int_data und quant_buf reduziert (als ZLE- und Huffman-Ausgaben werden anstatt eigener Speicherbereiche rle_buf und codep die beiden bereits existierenden Bereiche int_data und quant_buf wiederverwendet. Außerdem werden in der optimierten Variante für jeden Zeilen- und Spaltendurchlauf der Wavelet-Transformation nicht zu Beginn die kompletten Daten der Zeile bzw. Spalte einem lokalen Speicher zugewiesen wie dies in der unoptimierten Variante der Fall ist (siehe [1], Methode fcdf22, erste for-Schleife). Stattdessen geschehen diese Zuweisungen direkt während der Berechnungen zur jeweiligen Transformation, so dass hier insgesamt gesehen ein kompletter Bilddurchlauf eingespart wird. Eine weitere Ersparnis besteht in der Verarbeitung der Blökke in Quantisierung und ZLE: da nämlich nur 7 der 10 Wavelet-Blöcke gespeichert werden (die anderen werden zu Null gesetzt), müssen auch nur diese 7 Blöcke anstatt der 10 Blöcke in den beiden Phasen bearbeitet werden.

Eine weitere Optimierung (ebenfalls nicht zu dieser Arbeit gehörig) ermöglicht es der Anwendung, falls sie auf der ACE-V ausgeführt wird, den schnellen SRAM dieser Plattform als Zwischenspeicher für Wavelet-Transformation und Quantisierung zu benutzen.

Zwei kleinere Änderungen wurden im Zuge dieser Arbeit am Algorithmus vorgenommen. In der Quantisierung werden die einzelnen Blöcke im Original spaltenweise durchlaufen:

```
for (i = x; i < x + size; i++) {
        for (j = y; j < y + size; j++) {
                ...
        }
}
```

Für eine hardwareseitige Bearbeitung sind aus Speicherzugriffsgründen zeilenweise Durchläufe effizienter. Deswegen wurden die beiden obigen for-Schleifen vertauscht:

```
for (j = y; j < y + size; j++) {
        for (i = x; i < x + size; i++) {
                ...
        }
}
```

Die nächste und letzte Änderung betrifft die Byte-Länge des Ausgabedatenstroms nach der Huffman-Codierung. Sie wird im Original wie folgt berechnet:

```
nbytes = (nb / 8);
```

Dabei ist nbytes die Bytelänge des Ausgabestroms und nb die Anzahl dessen Bits. Da nbytes und nb beides ganzzahlige Werte sind, werden durch diese Zuweisung eventuelle Rest-Bits (maximal 7) vernachlässigt. Diese Vernachlässigung ist aber nicht sinnvoll; also ignoriert die optimierte Software-Fassung jene Bits nicht:

```
nbytes = (nb >> 3);
if (nb % 8 != 0) nbytes++;
```

2.3 Definition der Anforderungen für die Studienarbeit

Aufgabe der Studienarbeit ist es, den gegebenen C-Algorithmus [1] zur wavelet-basierten Kompression von Graustufenbildern in eine kombinierte HW/SW-Lösung für die ACE-V [5] umzusetzen, mit den folgenden Einschränkungen:

- es wird nur die Codierung entworfen, nicht die Decodierung, und
- die Anwendung verarbeitet Bilder der festen Größe 256x256 Pixel, mit 8 Bit Graustufeninformation pro Pixel, im PGM-Format [6].

Dabei sind weitere Anforderungen zu beachten:

- die Komprimierungsstärke soll wie in der Softwarelösung variabel einstellbar sein,
- bei der Ausarbeitung des Entwurfs soll auf möglichst kurze Laufzeiten der Anwendung hingearbeitet werden, und
- wenn die Softwarelösung aus Gründen der effizienteren Hardwarenutzung abgeändert wird, ist zu zeigen, dass die Qualität der erzielten Komprimierung die Versatility-Stressmark-Eigenschaften aufweist (siehe [1]).

Da die ACE-V als Zielplattform ein adaptiver Rechner ist, ist unmittelbar klar, dass die fertige Anwendung aus einem Software- und einem Hardware-Teil besteht. Der Software-Tteil wird in der Programmiersprache C verfasst, der Hardware-Teil in der Hardware-Beschreibungssprache Verilog.

Um das Gros der für die Bearbeitung der Studienarbeit verwendeten Zeit nicht mit der Implementierung von PCI-Kommunikationprotokollen zu verbringen, wird als Speicherzugriffsystem MARC [2] verwendet. MARC bietet sowohl wahlfreie Zugriffe auf einzelne Speicherpositionen als auch Burst-Transfers für die schnelle Übertragung von zusammenhängenden Datenblöcken.

Die Entwicklungsumgebung besteht aus den Programmen „Synplify 7.1.3“ (Hardware-Optimierung und Erzeugung der Gatternetzlisten), den Xilinx-Tools in der Version 5.2 (Platzierung, Verdrahtung und Bitstromgenerierung) sowie dem Programm „VCS“ von der Firma Synopsys in der Version 6.2 (Simulation).

2.4 Partitionierung

Wir wollen nun festlegen, welche Teile des Algorithmus in Hardware ausgelagert werden und welche softwareseitig gerechnet werden.

Wie wir in Abschnitt 2.1 bereits festgestellt haben, besteht die Anwendung im wesentlichen aus den vier

Bearbeitungsphasen der mehrstufigen Wavelet-Transformation, der Quantisierung, Lauflängen- und Huffman-Codierung. Die mehrstufige Wavelet-Transformation beinhaltet sehr viel Arithmetik und sollte deswegen auf jeden Fall in Hardware umgesetzt werden. Die anderen drei Phasen sind effizient zu einer Verarbeitungspipeline hintereinanderschaltbar. Es bietet sich daher an, alle vier Phasen in Hardware auszulagern. Dabei sollte man darauf achten, dass die entworfene Hardware nicht zu kompliziert wird, denn schließlich muss sie in den FPGA der ACE-V passen. Ein Ausweg bei einer zu komplexen Realisierung wäre, die Hardware in mehrere FPGA-Beschreibungen aufzuteilen und den FPGA der ACE-V für jede oder jede zweite Bearbeitungsphase neu zu programmieren. Dies hätte allerdings den Nachteil einer erheblich längeren Laufzeit, da jede Neuprogrammierung und Initialisierung der ACE-V größenordnungsmäßig etwa 2 Sekunden benötigt (das ist um mehrere Größenordnungen mehr, als eine Ausführung des Algorithmus in Software auf einem Standard-PC benötigen würde). Angenehm ist jedoch, dass auf der ACE-V ein sehr großer FPGA steckt (Xilinx Virtex 1000), so dass an Hardware-Komplexität nicht allzu sehr gespart werden muss.

Die Aufgaben des Bilddatenein- und Auslesens von bzw. auf den Festspeicher übernimmt zweckmäßigerweise der Software-Teil der Anwendung.

Übrig bleibt die Berechnung der Blockgrenzwerte:

```
for (i = 1; i < 7; i++)
    blockthresh[i] = (blockthresh[i] * quant) >> 7;
```

Diese Berechnung könnte in Hardware ausgeführt werden. Die Vorteile wären, dass die Zeit für die Berechnung in Software (dies sind ca. 5 µs @ 100 MHz CPU-Takt; die auf der ACE-V verwendete CPU ist eine microSPARC IIep) wegfallen würde. Außerdem bräuchte anstatt der sechs Datenworte `blockthresh[1]`, ..., `blockthresh[6]` nur der Wert `quant` von der Software in die Hardware übertragen werden. Die Dauer von 6 schreibenden Slave-Mode-Zugriffen liegt derzeit bei ca. 18 µs (bei 30 MHz FPGA-Takt). Das ergäbe eine Ersparnis von maximal 23 µs. Diese Zeit steht in keinem Verhältnis zu der Laufzeit von über 6000 µs, die die Hardware ohnehin für die Abarbeitung der vier Hauptbearbeitungsphasen benötigt (ebenfalls bei 30 MHz FPGA-Takt). Daher wurde die Entscheidung getroffen, die Blockgrenzwertberechnung ebenfalls in Software durchzuführen.

3 Der Software-Teil der Anwendung

Die Bilder 3.1 und 3.2 geben einen Überblick über den Programmablauf des Software-Teils der Anwendung.

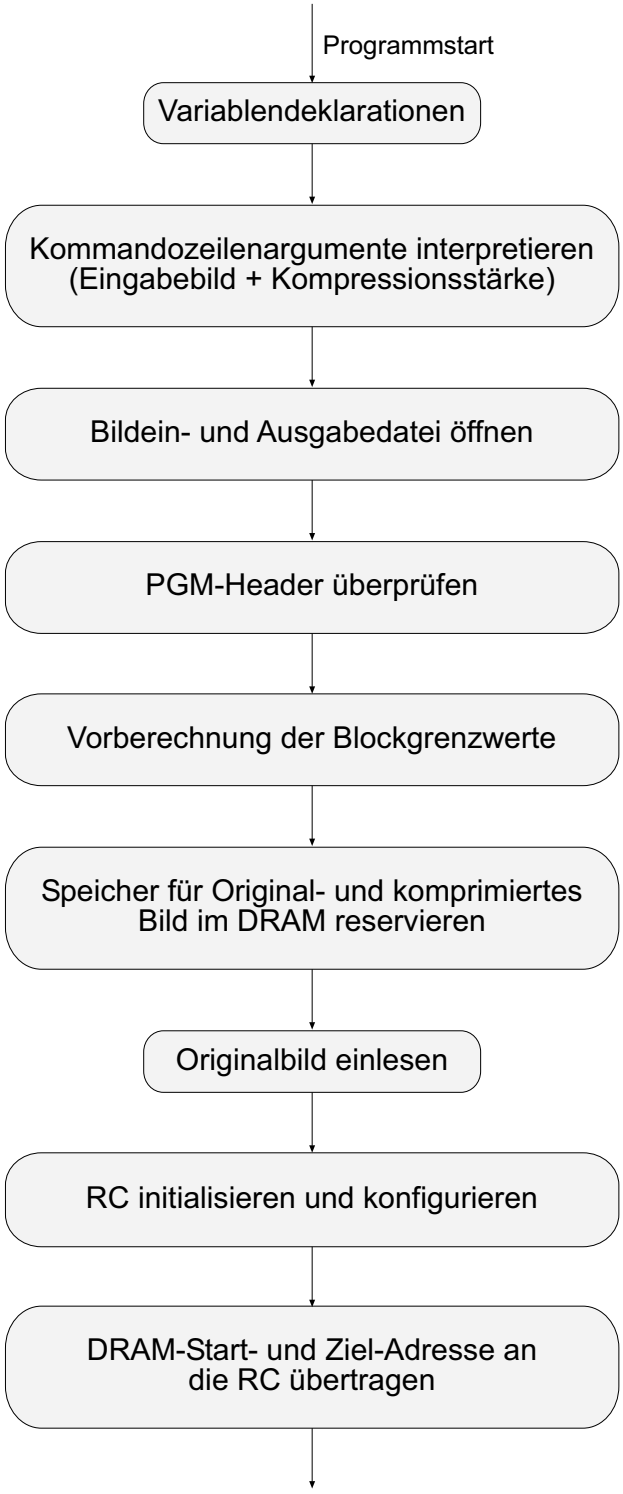


Bild 3.1:
Software-Programmablauf, Teil 1 von 2.

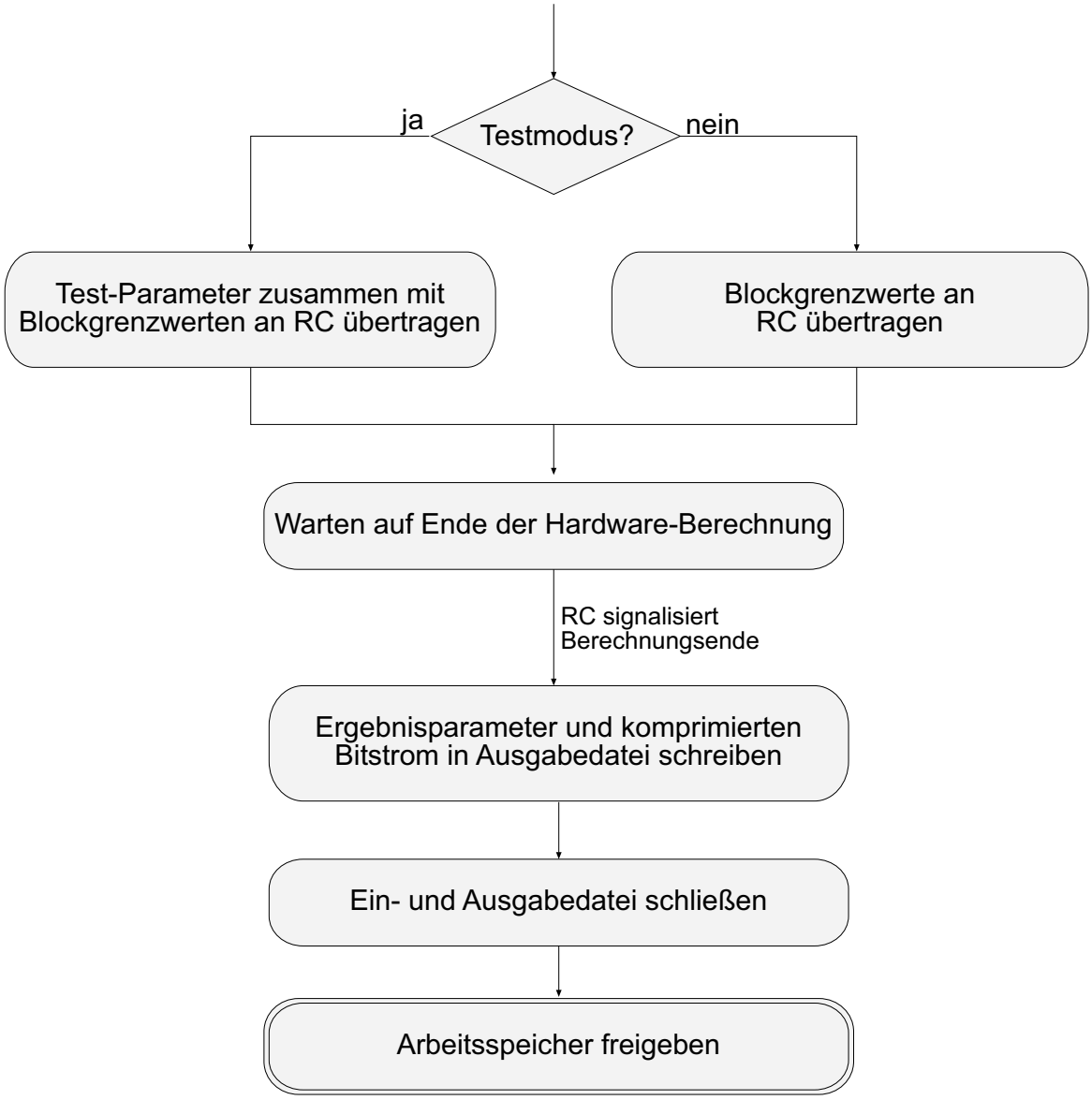


Bild 3.2:
Software-Programmablauf, Teil 2 von 2.

Die Eingabeparameter (PGM-Bild und Komprimierungsstärke) sind mit denen der reinen Softwarelösung (siehe Abschnitt 2.2) identisch. Dass der Quelltext des Softwareteils der Anwendung hier viel kürzer ausfällt als die reine Softwarelösung ist klar, da ja die meiste Arbeit von der Hardware erledigt wird. Die Software liest lediglich die Bilddaten ein, berechnet die 6 Blockgrenzwerte (der Blockgrenzwert für Wavelet-Block Nr. W0 ist immer 0, daher muss dieser nicht berechnet werden), überträgt einige Parameter an die RC (für eine genauere Beschreibung der Schnittstelle siehe Kapitel 4) und schreibt den komprimierten Bitstrom samt Header in die Ausgabedatei.

Für Laufzeittests ist es wichtig, im Programmkopf die Zeile

```
#define VERBOSE 1
```

zu ändern in

```
#define VERBOSE 0
```

damit die zu messenden Zeiten nicht durch die zeitintensiven `printf()`-Aufrufe verfälscht werden.

Über das Define

```
#define TICKS_PER_USEC 30
```

kann die Taktung der Hardware (in MHz) eingestellt werden. In diesem Beispiel beträgt die Taktung 30 MHz.

Zum Testen einzelner Hardwarekomponenten sowie auch der Hardware als ganzes werden - über Zuweisungen an

```
rc[REG_TESTMODE]
```

auswählbar - vier verschiedene Testmodi angeboten. Zur Anwendung der Testmodi sei auf Kapitel 6 verwiesen.

4 Die Hardware-Software-Schnittstelle

Dank des verwendeten Speicherzugriffsystems MARC [2] konnte der Datenaustausch zwischen Hard- und Software-Teil der Anwendung relativ einfach realisiert werden.

4.1 Slave-Mode-Zugriffe

Die RC wird in den CPU-Adressraum eingebettet, was lesende und schreibende „memory-mapped"-Speicherzugriffe für den sogenannten Slave-Modus ermöglicht. Dieser Modus wird benutzt, um wie im folgenden Beispiel Parameter von der Software in die Hardware zu übertragen:

```
rc[REG_START] = 1;
```

Andersherum kann durch diese simplen Zuweisungen die Software auch Daten von der Hardware lesen, doch lesende Slave-Zugriffe sind sehr zeitintensiv und werden deswegen in dieser Anwendung vermieden.

Der Datentransfer wird für Slave-Mode-Zugriffe von der Software initiiert; die Hardware bleibt dabei also passiv. Den Pointer rc auf den eingebetteten Speicherbereich liefert die in der Datei acevapi.h definierte Methode acev_get_s0(NULL). Bei der Definition des Speicher-Pointers rc ist auf die Verwendung des Schlüsselwortes volatile zu achten; nur so wird garantiert, dass der Compiler Zugriffe der Art

```
rc[0]    = temp[0];
temp[0]  = rc[0];
```

nicht wegoptimiert.

In der Anwendung treten Slave-Mode-Zugriffe lediglich auf, um die Start-Parameter (DRAM-Start- und Ziel-Adresse, Test-Modus-Konfiguration, Hardware-Startsignal) an die RC zu übertragen. Der Rücktransfer der Ergebnisparameter (Blockminima und -maxima, ZLE-Blockgrößen, Huffman-Bytelänge) nach Beendigung der eigentlichen Bilddaten-Transformation geschieht - wie oben bereits erwähnt - aus Effizienzgründen nicht im Slave-Mode.

4.2 Master-Mode-Zugriffe

Die zweite mögliche Speicherzugriffsart sind Master-Mode-Zugriffe; die Kontrolle über den Datenaustausch liegt in diesem Fall bei der Hardware. Benutzt werden diese Zugriffe hier zum Lesen und Schreiben der originalen und der transformierten Bilddaten. Die Initialisierung und Übertragung über den ACE-V-internen PCI-Bus steuert MARC, nachdem für jeden Transfer die Lese- und Schreibströme entsprechend von der Hardware programmiert wurden. Um einen MARC-Strom zu programmieren, müssen die Parameter

- Startadresse,
- Anzahl Datensätze,
- Schrittweite,
- Wortbreite der Zugriffe sowie
- Zugriffsart (lesen oder schreiben)

eingestellt werden. (Als Beispiel siehe Quelltext z. B. der Datei wavelet_controller.v.)

Während einer Hardwareberechnung werden in dieser Anwendung typischerweise zwei oder drei MARC-Ströme gleichzeitig benutzt: ein Strom liest Daten ein, ein weiterer schreibt die berechneten Werte an die Zielposition im Speicher. Im Fall der Wavelet-Verarbeitung werden für die Wavelet-Phasen 3 bis 6 (ent-

spricht den Wavelet-Stufen 2 und 3) sogar zwei Schreibströme benötigt: einer für die Low-Daten und einer für die High-Daten.

Abgesehen von gleichzeitig arbeitenden Strömen können MARC-Ströme auch immer wieder neu programmiert werden, um so in verschiedenen Phasen auf unterschiedliche Speicherpositionen zuzugreifen.

Nachdem die Hardware-Berechnung beendet ist, meldet sich die RC bei der Software durch das Setzen eines Interrupts („IRQ“). Das veranlasst die Software zur Ausführung der Funktion irq_handler(). Hierin findet ein Slave-Mode-Zugriff auf die RC statt, was den Interrupt (zur weiteren Software-Berechnung) rücksetzt. Anschließend fährt die Software durch den Funktionsaufruf acev_mark_done() nach der Stelle acev_wait() im Hauptprogramm, an der auf das Hardware-Berechnungsende gewartet wurde, fort.

5 Der Hardware-Teil der Anwendung

Dieses Kapitel beschreibt die Konfiguration des FPGAs auf der ACE-V für diese Anwendung. Die Hardware muss drei wesentliche Funktionen implementieren: die Entgegennahme von Parametern im Slave-Mode (1), das eigenverantwortliche Transformieren der Bilddaten im Master-Mode (2) sowie die Rückübermittlung der Ergebnisparameter im Master-Mode (3). Für Punkt (1) sei auf Abschnitt 5.5.17 (detaillierte Beschreibung des Moduls user) verwiesen, da diese Implementierungen sehr kurz und einfach sind. Punkt (3) wird in Abschnitt 5.5.14 (detaillierte Beschreibung des Moduls result_controller) behandelt. Die Hauptaufgabe der Hardware ist Punkt (2): die Verarbeitung der eigentlichen Bilddaten.

Nach den Vorüberlegungen aus Kapitel 2 lässt sich leicht ein Verarbeitungsablauf definieren; wir werfen dazu einen Blick auf Bild 5.1.

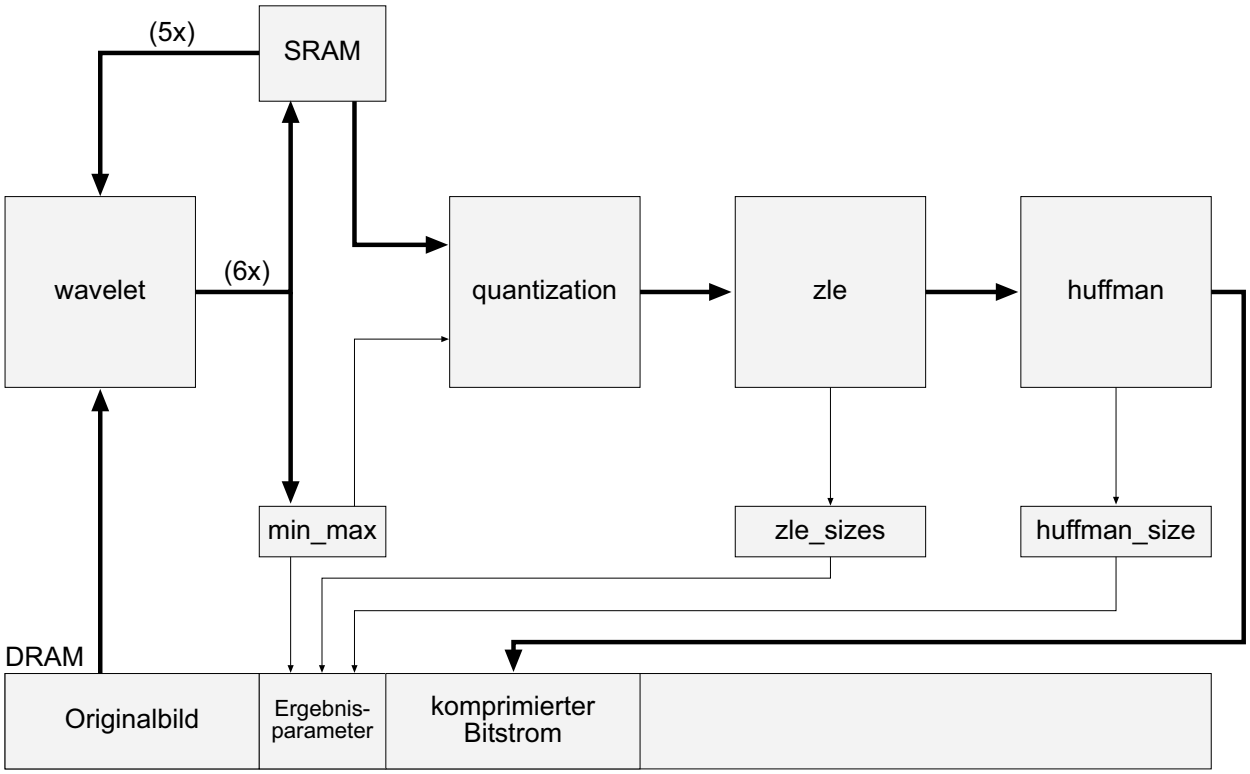


Bild 5.1:
Hardware-Verarbeitungsablauf. Die fett gedruckten Pfeile deuten den Weg der Bilddaten an.

Das Originalbild wird aus dem DRAM gelesen, wavelet-transformiert und in den SRAM geschrieben. Da die Transformation dreistufig aufgebaut ist, erfolgen drei Wavelet-Stufen hintereinander, wobei jede Stufe aus zwei Transformationsdurchläufen durch das gesamte Bild besteht: je einmal zeilenweise und einmal spaltenweise. Insgesamt macht das sechs Transformationen, was durch die Beschriftung „6x“ im obigen Bild angedeutet wird. Aus dem SRAM müssen die Daten daher fünfmal („5x“) wieder gelesen und der Wavelet-Transformation zugeführt werden (nur für den Zeilendurchlauf der ersten Wavelet-Stufe werden die Daten direkt dem DRAM entnommen; dieser Bereich im DRAM ist mit „Originalbild“ gekennzeichnet). Der SRAM ist - ohne Umweg über den internen PCI-Bus der ACE-V - direkt am FPGA angeschlossen und wird daher als schneller Zwischenspeicher in den Wavelet-Stufen gegenüber dem DRAM bevorzugt.

Da die Quantisierung die 7 Blockminima und -maxima benötigt, werden diese in min_max gespeichert und jeweils während der Wavelet-Transformationen aktualisiert. Die im SRAM abgelegten Wavelet-Blöcke wandern nun der Reihe nach durch die Quantisierung (quantization), die Zerolängencodierung (zle) und

die Huffman-Codierung (huffman) und landen komprimiert im DRAM. Die Ergebnisparameter der ZLE-Blockgrößen sowie die Huffman-Bytelänge werden in zle_sizes bzw. huffman_size gespeichert und nach Ende der Bildverarbeitung zusammen mit den Blockminima und -maxima im Master-Mode in den DRAM übertragen.

5.1 Pipelining

Da für die Studienarbeit eine möglichst schnelle Datenverarbeitung eins der Hauptziele ist, wird hier natürlich das bewährte Pipeline-Prinzip eingesetzt. Optimal wäre eine einzige Pipeline, in der alle Transformations- und Komprimierungsschritte enthalten sind. Die Bilddaten würden dann vom DRAM gelesen werden, und am Ende der Pipeline würde der Ergebnisdatenstrom abgegriffen und wieder zurück in den DRAM geschrieben.

Leider kann eine solche Vorgehensweise in dieser Anwendung aber nicht funktionieren. Der Grund ist einfach: die Wavelet-Transformation erfolgt abwechselnd zeilen- und spaltenweise. Bevor die erste Spalte transformiert werden kann, müssen alle ersten Pixel jeder Zeile bereits zeilen-transformiert worden sein. Dies ließe sich durch 256 parallel arbeitende Zeilen-Transformationsmodule erreichen (mit den entsprechenden Speicherzugriffen), was eher illusorisch als praktikabel ist. Praktisch muss ein Zeilen-Transformationsschritt also abgeschlossen sein, bevor die Spaltentransformationen beginnen können. Und genau diese Tatsache verhindert die oben beschriebene Wunsch-Pipeline.

Wir werden Pipelines in der Anwendung also eine Ebene tiefer entwerfen. Ein Zeilen- oder Spaltendurchlauf einer Wavelet-Transformation lässt sich gut pipelinen (Abschnitt 5.1.1). Die drei übrigen Phasen - Quantisierung, Zerolängencodierung, Huffman-Codierung, kurz: „QZH“ - lassen sich sogar zu einer größeren Pipeline zusammenfassen (Abschnitt 5.1.2).

5.1.1 Wavelet-Pipeline

Da die Wavelet-Transformation Zeile für Zeile bzw. Spalte für Spalte geschieht, ist eine Zeile bzw. Spalte der größte „am Stück“ in einer Pipeline bearbeitbare Bereich. Die Tatsache, ob das Bild gerade zeilen- oder spaltenweise durchlaufen wird, macht für den Pipeline-Betrieb keinen Unterschied. Anders ist in beiden Fällen jedoch der Gesamtablauf. Bei einer zeilenweisen Bearbeitung können die Daten ohne „Pause“ zwischen den Zeilen eingelesen, verarbeitet und ausgegeben werden, im Spaltenfall jedoch muss der Lesestrom nach dem Einlesen einer Spalte erneut programmiert (hier: auf die passende Anfangsposition der nächsten Spalte gesetzt) werden. Eine Spaltenverarbeitungspipeline muss daher während dieser Neuprogrammier-Zeit angehalten werden.

Kurz soll hier noch auf die Wortbreite der verarbeiteten Daten eingegangen werden. Die Original-Bilddaten aus dem DRAM besitzen 8 Bit pro Pixel. MARC kann pro Datenwort maximal 32 Bit liefern, daher ist es für eine schnelle Verarbeitung günstig, jeweils 4 Pixel gleichzeitig einzulesen. Dies geschieht in der Realisierung in der Tat (siehe Modul wavelet_8_4_l, Abschnitt 5.5.1). Ausgegeben werden die transformierten Daten jedoch bereits mit 16 Bit pro Pixel, da pro Wavelet-Transformation - bedingt durch Shifts und Additionen - die Ausgabedaten gegenüber den Eingaben um 1 Bit breiter werden. Ein Bit kommt noch durch die Darstellung im 2er-Komplement für negative Zahlen hinzu (man könnte nach der ersten Transformation 10 Bit, nach der zweiten 11 Bit usw. verwenden, wegen der einfacheren praktischen Umsetzung werden aber auch in diesen Fällen schon 16-bittige Werte benutzt). Ausgegeben werden müssten vom Modul wavelet_8_4_l nun eigentlich 64 Bit (2x 16 Bit Low-Daten + 2x 16 Bit High-Daten), aber glücklicherweise setzt die Quantisierung die drei größten Detailblöcke sowieso zu Null, wodurch hier die

Ausgabe der High-Daten einfach weggelassen werden kann! Die Ausgabe von wavelet_8_4_l besteht daher lediglich aus 2x 16 Bit Low-Daten.

Die übrigen 5 Wavelet-Transformationen (Spaltendurchlauf von Stufe 1 plus Zeilen- und Spaltendurchläufe der Stufen 2 und 3) werden hingegen von wavelet_16_2_hl bewältigt, welches die nun 16 Bit breiten Eingangsdaten mit 2 Pixeln pro Datenwort einliest und entsprechend auch 2x 16 Bit ausgibt; ein Ausgabedatenwort besteht dabei aus einem Low- und einem High-Datum.

Wie die genaue Berechnung in der Wavelet-Transformation lautet und wie diese in einen Pipelinebetrieb umgesetzt wurde, kann man in den detaillierten Modulbeschreibungen der beiden Wavelet-Module wavelet_8_4_l und wavelet_16_2_hl (Abschnitte 5.5.1 und 5.5.2) nachlesen.

5.1.2 QZH-Pipeline

Anders als in der Wavelet-Pipeline erstreckt sich diese Pipeline einerseits über drei verschiedene Module und andererseits über den gesamten Ausgabedatenstrom, bestehend aus 7 Wavelet-Blöcken, die zusammenhängend in einen komprimierten Bitstrom überführt werden.

Im Unterschied zum Wavelet-Fall handelt es sich hier um eine Pipeline mit variabler Datenrate. Das Modul quantization gibt genausoviele Daten aus, wie es einliest. Die beiden Module zle und huffman jedoch machen ihre Ausgabe aber von den eingelesenen Daten abhängig. Wenn zle beispielsweise gerade die Länge eines ZERO-Laufs misst, liest das Modul zwar zu jedem Takt Daten ein, produziert aber keine Ausgabe. Bei huffman kommt noch ein weiterer Punkt hinzu: ist der interne Bit-Puffer zu voll, werden weiterhin Daten ausgegeben, aber im Moment keine eingelesen. Die Module müssen daher selbst kontrollieren können, wann sie Daten lesen, und wann schreiben möchten.

In der Implementierung wurde hier auf das Stream-Konzept gesetzt: Neben einem Datenein- und Ausgangsport (STREAM_READ und STREAM_WRITE) hat jedes der drei Module in der QZH-Pipeline einen 2-bittigen „STREAM_STALL“-Eingang (STREAM_STALL[0] für den Lesestrom, STREAM_STALL[1] für den Schreibstrom) sowie entsprechend einen 2-bittigen „STREAM_ENABLE“-Ausgang. Wenn STREAM_STALL[0] gesetzt ist, ist das zur aktuellen Taktflanke (dies ist die gerade vergangene Flanke) gelesene Datenwort ungültig und wird nicht verarbeitet. Bei STREAM_STALL[1] ist das zur aktuellen Taktflanke geschriebene Datenwort ungültig und muss (damit das Datum nicht verlorengeht) erneut an den Ausgang angelegt werden. STREAM_ENABLE[0] bedeutet, dass das Modul zur aktuellen Taktflanke Daten lesen möchte, STREAM_ENABLE[1] meint analog, dass zur aktuellen Taktflanke Daten geschrieben werden sollen.

Um das gewünschte Steuerverhalten zu erzielen, werden die Enable-Ausgänge eines Moduls über je einen Inverter mit dem entsprechenden Stall-Eingang der Nachbarmodule verbunden, wie wir es in Bild 5.2 sehen. Mit eingezeichnet sind in diesem Bild auch alle anderen Modul-Ein- und Ausgänge. Die Eingänge CLK, RESET und ENABLE verstehen sich von selbst. Ebenfalls bei allen drei Modulen vorhanden ist der Ausgang FINISHED. Ist FINSHED = 1, so liegt am STREAM_WRITE-Ausgang des Moduls das letzte gültige Datenwort an.

quantization hat die zusätzlichen Eingänge BLOCK_THRESH_9 (Blockgrenzwert des aktuellen Blocks), BLOCK_MIN und BLOCK_MAX (aktuelle Block-Minima und -maxima), die zur Quantisierung der Eingangsdaten unerlässlich sind. Zusätzliche Ausgänge sind BLOCK_NUM und BLOCK_FINISHED: BLOCK_NUM gibt die Nummer des Blocks an, aus dem die gelesenen Daten stammen (diese Information ist für die Bereitstellung der korrekten Blockminima, -maxima und -grenzwerte wichtig). BLOCK_FINISHED hat genau dann den Wert 1, wenn im aktuellen Takt das letzte zu lesende Datum am Dateneingang anliegt (bei BLOCK_FINISHED = 1 in einem gültigen Lesetakt weiß so der

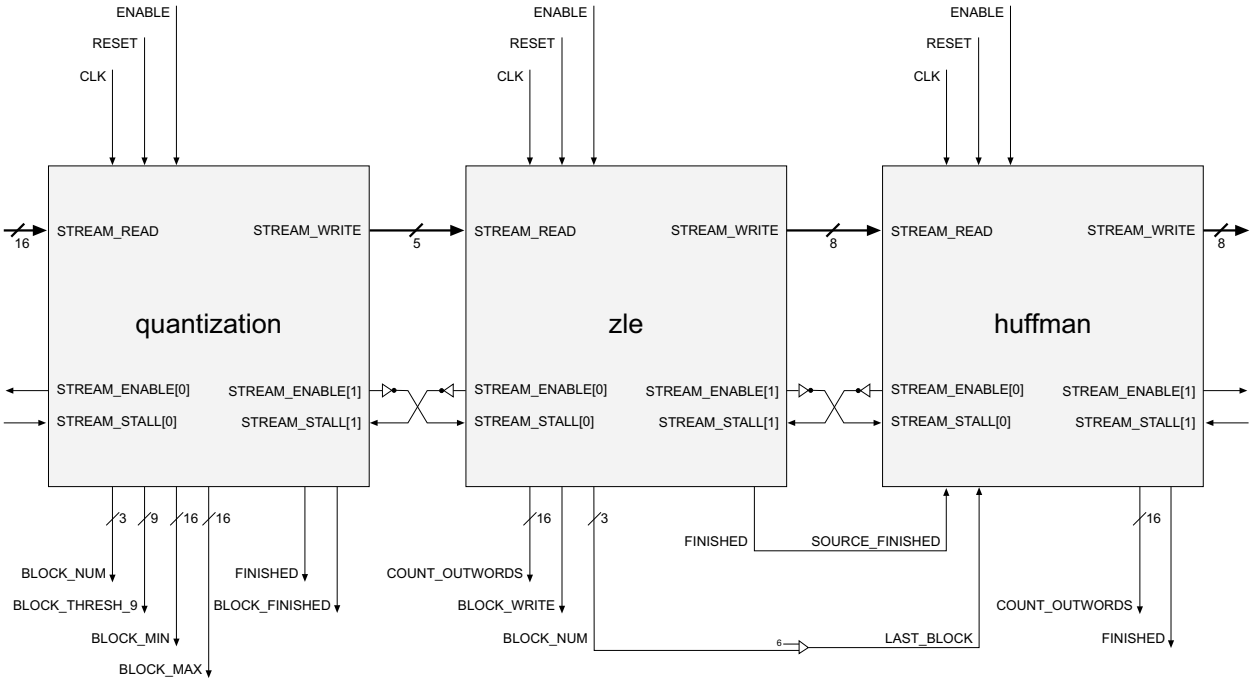


Bild 5.2:
Verkettung der Module quantization, zle, huffman zur QZH-Pipeline. Die fett gedruckten Pfeile deuten den Pfad der komprimierten Bilddaten an.

qzh_controller, dass er den Lesestrom stoppen kann).

zle hat auch ein BLOCK_NUM-Signal als Ausgang, das jedoch im Unterschied zum gleichnamigen Quantisierungs-Signal die Blockzugehörigkeit des momentan zu schreibenden Datums angibt, nicht die des zu lesenden Datums. Ist BLOCK_WRITE = 1, wird die in COUNT_OUTWORDS gezählte ZLE-Blockgröße des Blocks BLOCK_NUM von einem außenstehenden Modul zur späteren Abfrage gesichert. Ferner gibt BLOCK_NUM dem Huffman-Modul die Auskunft, ob es sich momentan um den letzten zu bearbeitenden Block handelt oder nicht. Dazu wird BLOCK_NUM mit der Zahl 6 verglichen (Block Nr. 6 ist der letzte Block) und - falls die Signale identisch sind - der Eingang LAST_BLOCK von huffman auf 1 gesetzt.

Falls im Modul huffman die Eingänge SOURCE_FINISHED und LAST_BLOCK auf 1 gesetzt sind, weiß das Modul, dass es gerade das letzte gültige Datum liest. Nun muss nur noch der eigene Puffer geleert werden, bevor auch hier FINISHED auf 1 gesetzt werden kann, was dem qzh_controller das Ende der Berechnung signalisiert.

Der Leser fragt sich nun vielleicht, woher denn quantization und zle die Information über die Blocknummer und die FINISHED-Signale haben, die sie nach außen führen. Da die Anzahl der einzulesenden Datenworte bei beiden Modulen nur von der Größe des Originalbildes abhängt (diese Größe beträgt konstant 256x256 Pixel), zählen beide Module die gültigen Eingabeworte von Anfang an. Es gibt also je ein Zählregister: in quantization heißt es COUNT_PIXEL, in zle COUNT_INWORDS. Diese Zählregister werden ständig mit den fest codierten Blockgrößen (Blöcke W0, W1, W2, W3: 1024 Werte; Blöcke W4, W5, W6: 4096 Werte) verglichen, um den Übergang zum nächsten Block festzustellen.

Hinzugefügt sei noch eine Bemerkung zum Modul quantization. Beim Übergang von einem Block zum nächsten müssen die 15 Quantisierungs-Grenzwerte aus dem aktuellen Blockminimum und -maximum berechnet werden, bevor mit der Verarbeitung fortgefahren werden kann. Das Schreib-Stall-Signal muss während dieser Zeit gesetzt sein, um keine ungültigen Daten weiterzugeben.

Wir wollen uns nun anhand eines Wave-Diagramms den Weg der wavelet-transformierten Bilddaten durch die QZH-Pipeline anschauen (Bild 5.3).

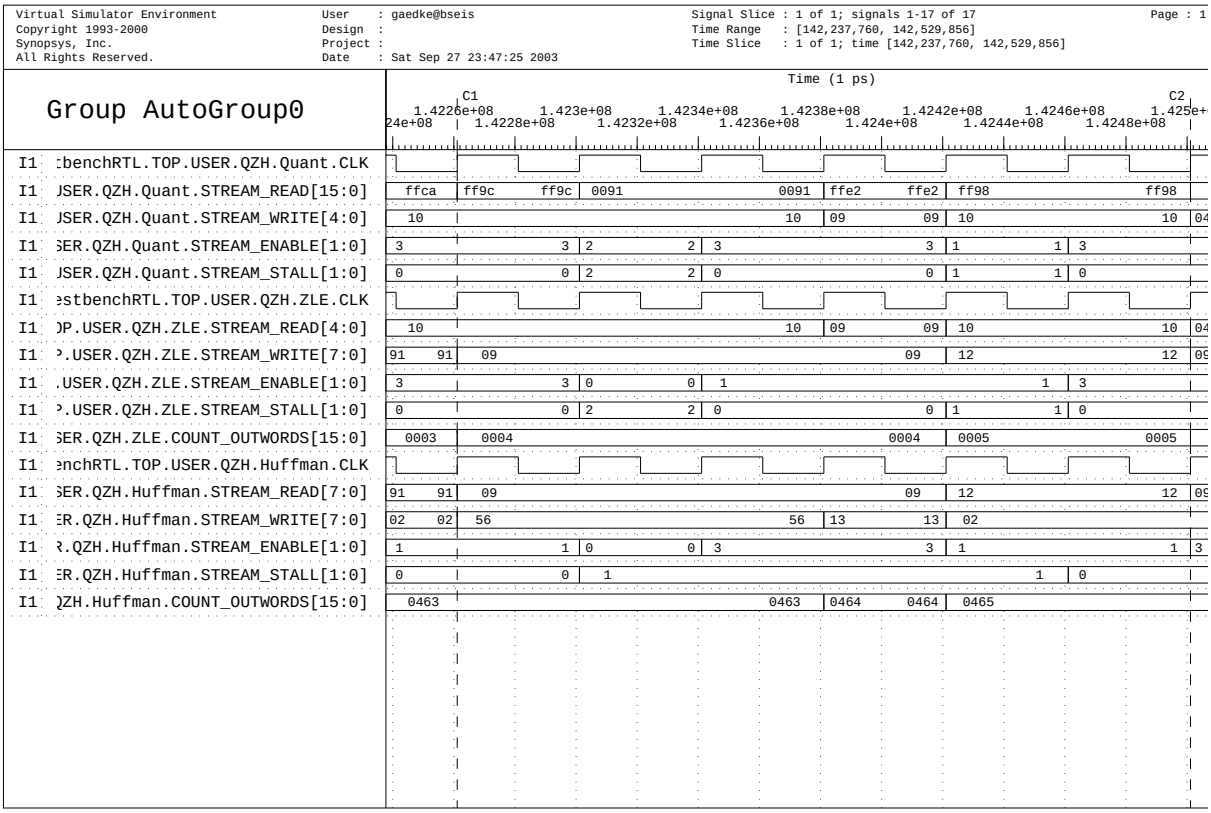


Bild 5.3:
Waveform zur Verarbeitung der wavelet-transformierten Bilddaten in der QZH-Pipeline: oben Quantisierung, mittig Zerolängencodierung, unten Huffman-Codierung.

Der Takt direkt rechts neben der vertikalen gestrichelten Linie „C1“ sei Takt Nr. 1, der Takt links neben der Linie „C2“ entsprechend Takt Nr. 6. In Takt 1 sehen wir im Signal Quant.STREAM_READ das Datum „ff9c“ (dies ist die Zahl „-100“ in hexadezimaler Zweierkomplementdarstellung). Das Wort „ff9c“ wird in diesem Takt gelesen, da Quant.STREAM_ENABLE[0] = 1 ist (Quant.STREAM_ENABLE = 3, d. h. Quant.STREAM_ENABLE[0] = 1 und Quant.STREAM_ENABLE[0] = 1) und Quant.STREAM_STALL[0] = 0. Die „0091“, die in Takt 2 am Leseeingang der Quantisierung anliegt, wird jedoch nicht gelesen, da das Lese-Enable (Quant.STREAM_ENABLE[0]) hier nicht gesetzt ist. Die „ff9c“ aus Takt 1 würde in Takt 2 von quantization geschrieben und von zle gelesen, wenn hier nicht Quant.STREAM_STALL[1] gesetzt wäre. So wird das Ausgabewort „10“, in das die „ff9c“ quantisiert wurde, zunächst am Quantisierungs-Ausgang gehalten und erst in Takt 3 von zle gelesen. Das in Takt 3 von der Quantisierung gelesene Wort „0091“ wird in den Wert „09“ quantisiert, der in Takt 4 von zle aufgenommen wird. zle beendet daraufhin den aktuellen ZERO-Run („10“ = „ZERO“), der bereits drei ZEROs gezählt hat: zwei ZEROs stammen aus der Zeit vor Takt 1, und der dritte ZERO-Wert entstand aus der in Takt 1 von quantization gelesenen Zahl „ff9c“. Ein Run, der drei ZEROs gezählt hat, wird mit der Zahl „12“ codiert, die tatsächlich in Takt 5 am Ausgang von zle anliegt. Da ZLE.STREAM_ENABLE[1] jedoch in Takt 5 nicht gesetzt ist, wird die „12“ erst in Takt 6 von zle geschrieben und von huffman gelesen. In Bild 5.3 ist noch ansatzweise zu sehen, dass huffman in Takt 7 ein Datum schreibt (Huffman.STREAM_ENABLE = 3, also lesen und schreiben). Anscheinend hat also der in Takt 6 gelesene Wert „12“ den huffman-internen Puffer soweit aufgefüllt, dass ein weiteres Byte geschrieben werden kann. In den Takten 3 und 4 war huffman ebenfalls schreibend aktiv, der Ursprung für jene Ausgaben liegt aber noch vor Takt 1.

Wir bemerken, dass der Byte-Zähler des Huffman-Moduls nach den Takten 3 und 4 jeweils um 1 inkrementiert wird. Das ZLE-Modul dagegen inkrementiert seinen Ausgabewort-Zähler schon zum Schreib-

Takt und nicht erst im Schreib-Takt. Dadurch konnten bereits vorhandene if-Zweige in der Hardware-Beschreibung des Moduls für den Zähler mitverwendet werden, was den Quelltext etwas einfacher macht. Das hat zwar zur Folge, dass z. B. in Takt 5 ZLE.COUNT_OUTWORDS schon den Ausgabewert 12 gezählt hat, obwohl dieser erst in Takt 6 tatsächlich geschrieben wird. Aber dies hat keine negativen Konsequenzen, da Takt 5 aus Sicht des zle-Moduls wegen des fehlenden STREAM_ENABLE[1] sowieso kein gültiger Schreibtakt ist.

5.2 Speicherorganisation

Die Aufteilung des DRAM und SRAM in Speicherbereiche ist ein zentraler Aspekt für die Implementierung. Das Speicherzugriffssystem MARC bietet maximal 4 Speicherbereiche an, auf die gleichzeitig zugegriffen werden kann, jeweils entweder lesend oder schreibend. Zunächst sollen die nötigen Speicherblöcke identifiziert werden (5.2.1), danach erfolgt die Aufteilung in die 4 Bereiche für MARC (5.2.2).

5.2.1 Speicherzugriffe während der Transformationen

Im QZH-Fall werden die Daten aus den Wavelet-Blöcken nacheinander durch die QZH-Pipeline geschickt und in einen großen Ziel-Bereich im DRAM geschrieben, was relativ unspektakulär ist; daher wird dieser Fall hier nicht extra beleuchtet. Die 6 verschiedenen Wavelet-Phasen (2 Phasen pro Wavelet-Stufe) verlangen aus Sicht der Speicherverwendung aber eine etwas genauere Untersuchung.

Angegeben werden pro Phase die Speicher- und Datenstrom-Konfiguration. Die identifizierten Speicherbereiche werden mit den Namen W0, W1, W2, W3, W4, W5, W6, T0, T1 und T2 bezeichnet (die allesamt im SRAM liegen). W0 bis W6 stellen die Wavelet-Blöcke dar, die später in der QZH-Pipeline weiterbearbeitet werden, während T0, T1 und T2 nur Zwischenspeicher sind. Die gestrichelten Linien in den folgenden sechs Bildern dienen der Kennzeichnung fortgelassener, nicht gespeicherter Daten. Die Schalter in den Bildern 5.7 und 5.9 werden umgelegt, nachdem die Hälfte der Eingabedaten gelesen bzw. die Hälfte der Ausgabedaten geschrieben wurde. Außerdem wird unterschieden zwischen gleichzeitig geschriebenen Daten („+“ im Text) und sequentiellen Abfolgen (Angabe „danach“ im Text; Sequenzen entstehen durch das Umlegen von Schaltern).

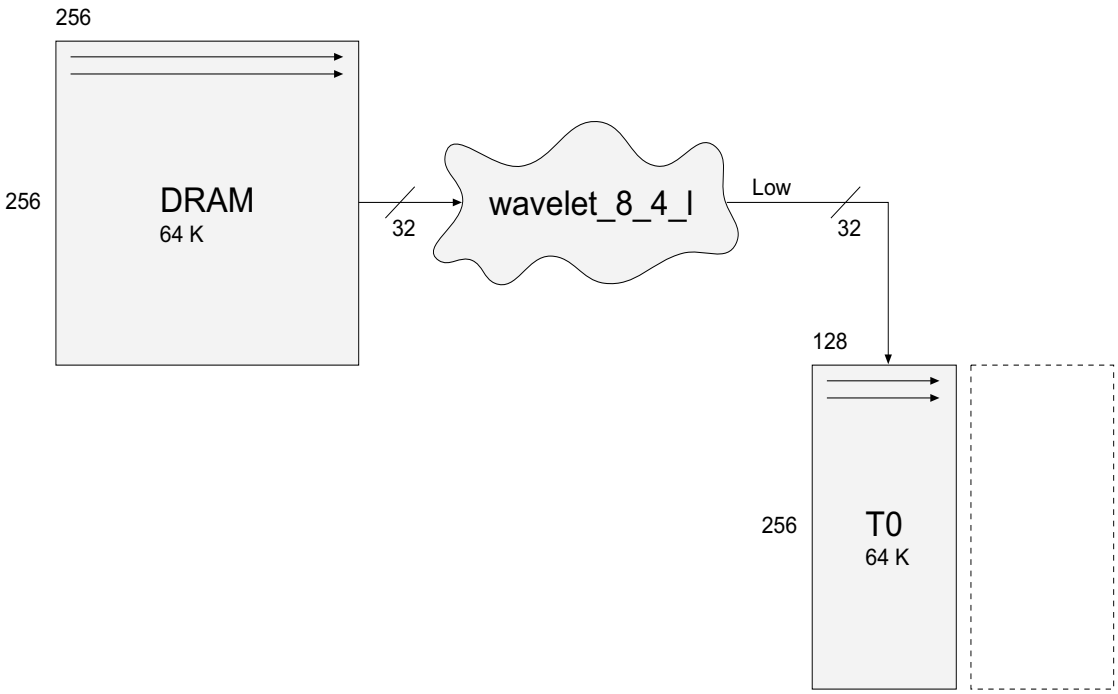


Bild 5.4:
Speicher- und Datenstrom-Konfiguration, Wavelet-Phase 1.

Phase 1 / 6:

Pixelfolge:	zeilenweise	
Speicherquelle:	DRAM (64 K)	
Speicherziel:	T0 (64 K)	
Pixelanordnung Lesen:	256x256	
Pixelanordnung Schreiben:	128x256	
Wortbreite Lesen:	32 Bit	
Wortbreite Schreiben:	32 Bit	
Datenformat Lesen:	8 Bit / Pixel,	4 Pixel / Datenwort
Datenformat Schreiben:	16 Bit / Pixel,	2 Pixel / Datenwort (Low-Daten)

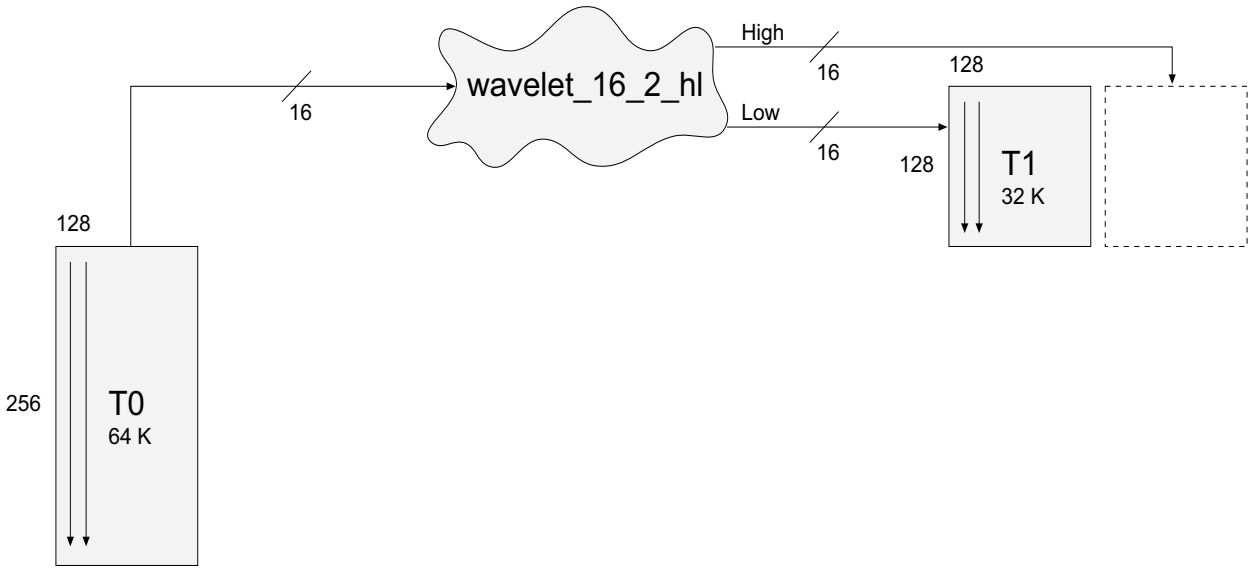


Bild 5.5:
Speicher- und Datenstrom-Konfiguration, Wavelet-Phase 2.

Phase 2 / 6:

Pixelfolge:	spaltenweise	
Speicherquelle:	T0 (64 K)	
Speicherziel:	T1 (32 K)	
Pixelanordnung Lesen:	128x256	
Pixelanordnung Schreiben:	128x128	
Wortbreite Lesen:	16 Bit	
Wortbreite Schreiben:	16 Bit	
Datenformat Lesen:	16 Bit / Pixel,	1 Pixel / Datenwort
Datenformat Schreiben:	16 Bit / Pixel,	1 Pixel / Datenwort (Low-Daten)

In den Phasen 2, 4 und 6 wird pro Takt nur ein Datum eingelesen und wieder geschrieben. In der Spaltenverarbeitung können nämlich mit dem verwendeten Speicherzugriffssystem nicht zwei Pixel gleichzeitig gelesen werden. Man könnte MARC zwar anweisen, 32-Bit-Worte zu lesen anstatt 16-Bit-Worte, würde dann als zweites eingelesenes Datum aber nicht wie gewünscht das zweite Datum der ersten Spalte, sondern das zweite Datum der ersten Zeile erhalten.

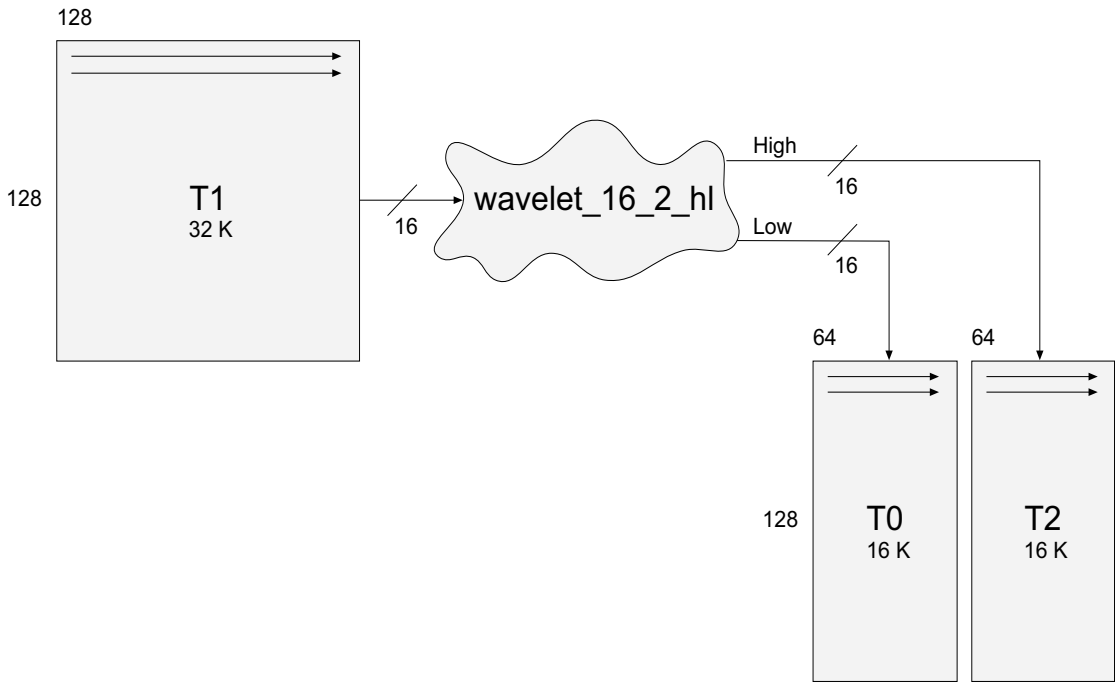


Bild 5.6:
Speicher- und Datenstrom-Konfiguration, Wavelet-Phase 3.

Phase 3 / 6:

Pixelfolge:	zeilenweise		
Speicherquelle:	T1 (32 K)		
Speicherziel:	T0 (16 K) + T2 (16 K)		
Pixelanordnung Lesen:	128x128		
Pixelanordnung Schreiben:	64x128 + 64x128		
Wortbreite Lesen:	32 Bit		
Wortbreite Schreiben:	16 Bit (Low) + 16 Bit (High)		
Datenformat Lesen:	16 Bit / Pixel,	2 Pixel / Datenwort	1 Datenwort / Takt
Datenformat Schreiben:	16 Bit / Pixel,	1 Pixel / Datenwort,	2 Datenworte / Takt

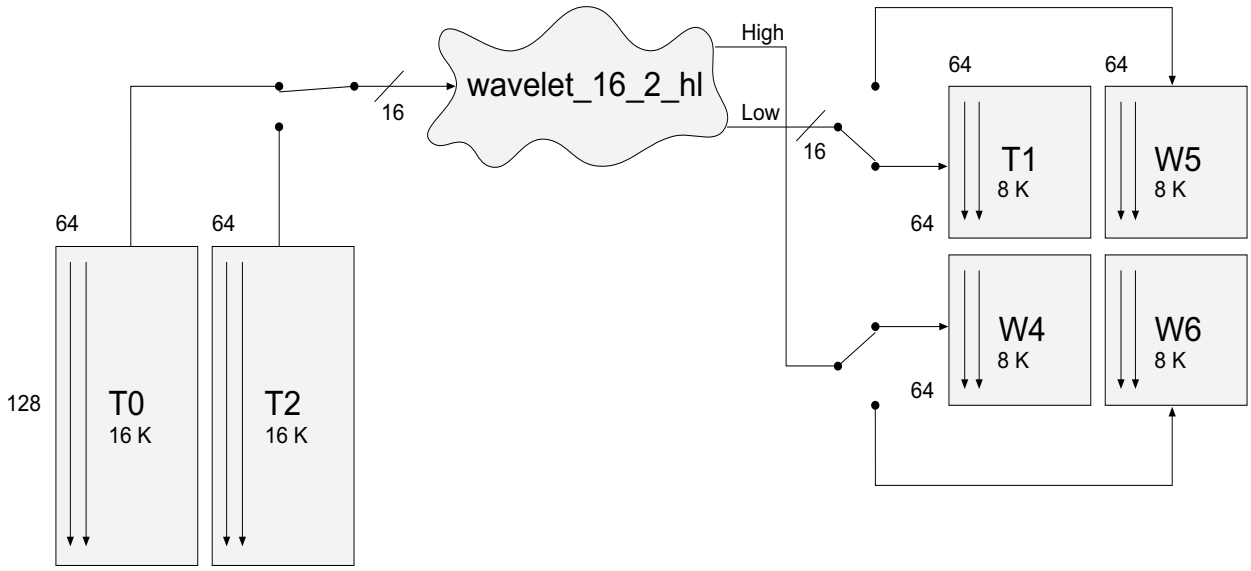


Bild 5.7:
Speicher- und Datenstrom-Konfiguration, Wavelet-Phase 4.

Phase 4 / 6:

Pixelfolge:	spaltenweise
Speicherquelle:	T0 (16 K), danach T2 (16 K)
Speicherziel:	T1 (8 K) + W4 (8 K), danach W5 (8 K) + 6 (8 K)
Pixelanordnung Lesen:	64x128
Pixelanordnung Schreiben:	64x64 + 64x64
Wortbreite Lesen:	16 Bit
Wortbreite Schreiben:	16 Bit
Datenformat Lesen:	16 Bit / Pixel, 1 Pixel / Datenwort
Datenformat Schreiben:	16 Bit / Pixel, 1 Pixel / Datenwort (alternierend 1 Low / 1 High)

Erklärung des Zusatzes „alternierend 1 Low / 1 High“ unter „Datenformat Schreiben“:
Wie schon in Phase 2 beschrieben, kann in der Spaltenverarbeitung nur ein Datum pro Takt gelesen werden. Dies hat zur Folge, dass auch nur eines pro Takt geschrieben werden kann. So werden alternierend Low- und High-Daten geschrieben: im ersten Schreib-Takt wird ein Low-Datum nach T1, im zweiten Takt ein High-Datum nach W4 geschrieben, im dritten Takt das nächste Low-Datum nach T1 usw., bis der Datenblock T0 abgearbeitet ist. Anschließend wird der Inhalt von T2 transformiert und analog nach W5 und W6 geschrieben.

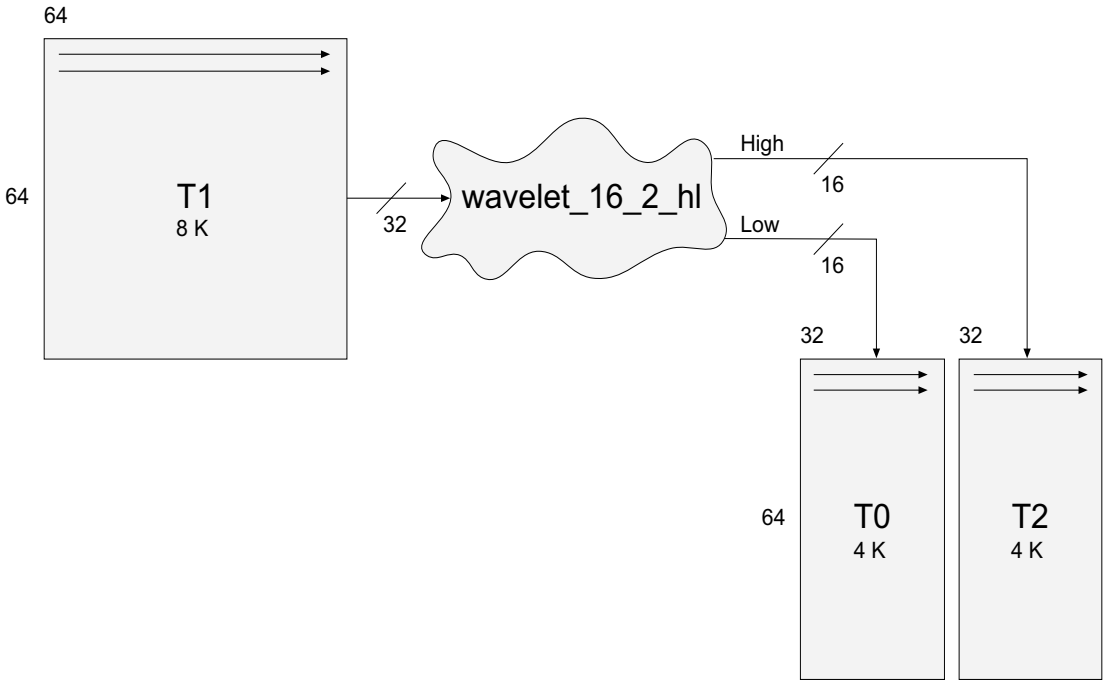


Bild 5.8:
Speicher- und Datenstrom-Konfiguration, Wavelet-Phase 5.

Phase 5 / 6:

Pixelfolge:	zeilenweise		
Speicherquelle:	T1 (8 K)		
Speicherziel:	T0 (4 K) + T2 (4 K)		
Pixelanordnung Lesen:	64x64		
Pixelanordnung Schreiben:	32x64 + 32x64		
Wortbreite Lesen:	32 Bit		
Wortbreite Schreiben:	16 Bit + 16 Bit		
Datenformat Lesen:	16 Bit / Pixel,	2 Pixel / Datenwort,	1 Datenwort / Takt
Datenformat Schreiben:	16 Bit / Pixel,	2 Pixel / Datenwort,	2 Datenworte / Takt (1 Low + 1 High)

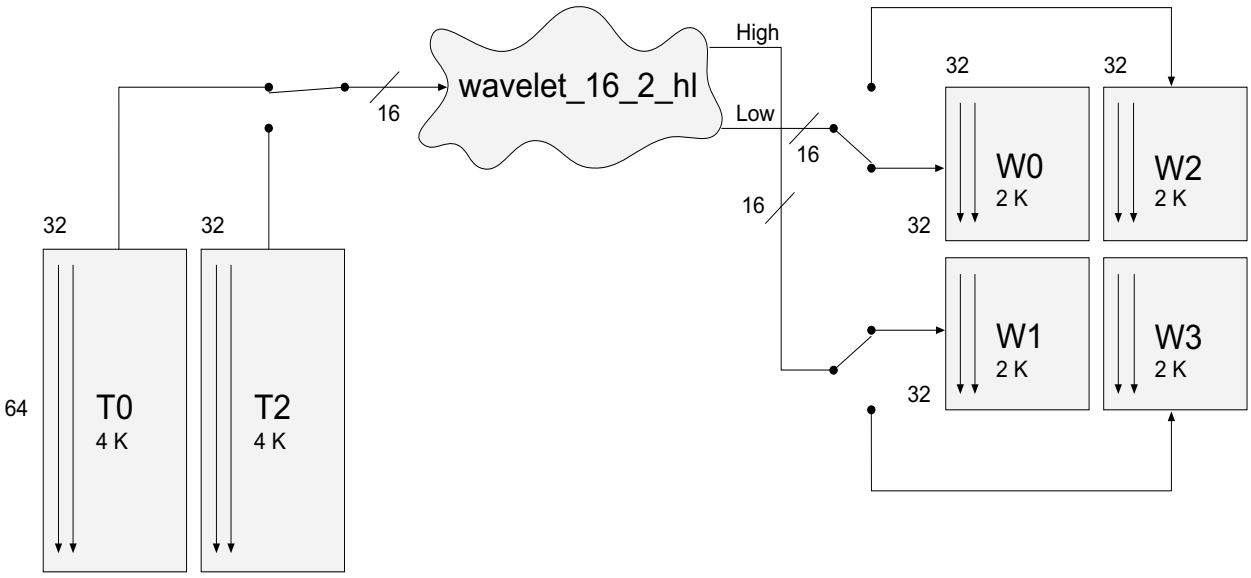


Bild 5.9:
Speicher- und Datenstrom-Konfiguration, Wavelet-Phase 6.

Phase 6 / 6:

Pixelfolge:	spaltenweise
Speicherquelle:	T0 (4 K), danach T2 (4 K)
Speicherziel:	W0 (2 K) + W1 (2 K), danach W2 (2 K) + W3 (2 K)
Pixelanordnung Lesen:	32x64
Pixelanordnung Schreiben:	32x32 + 32x32
Wortbreite Lesen:	16 Bit
Wortbreite Schreiben:	16 Bit
Datenformat Lesen:	16 Bit / Pixel, 1 Pixel / Datenwort
Datenformat Schreiben:	16 Bit / Pixel, 1 Pixel / Datenwort (alternierend 1 Low / 1 High)

Die Zwischenspeicher T0, T1 und T2 wurden zur effizienteren Speichernutzung während der sechs Phasen mehrfach verwendet. Wir bemerken, dass nicht immer die gleiche Speichermenge für diese Zwischenspeicher verwendet wird; die Dimensionierung muss aber natürlich für den „worst case“ ausreichen. Wir erhalten folgende Speichergrößen:

T0:	64 K
T1:	32 K
T2:	16 K
W0:	2 K
W1:	2 K
W2:	2 K
W3:	2 K
W4:	8 K
W5:	8 K
W6:	8 K

Die Speicherplatznamen W0 bis W6 entsprechen den in Kapitel 2 (Abschnitt 2.1.2) eingeführten Wavelet-Block-Bezeichnungen W0 bis W6.

5.2.2 Aufteilung des Speichers für MARC

Optimalerweise würde man die Blöcke W0 bis W6 so im SRAM anordnen, dass die Daten in der QZH-Verarbeitung in einem Rutsch eingelesen werden können, d. h. ohne nach jedem Block die Startadresse des Lesestroms neu zu programmieren. Wir haben wahrgenommen, dass während der Wavelet-Phasen 3 bis 6 auf jeweils drei Speicherblöcke gleichzeitig zugegriffen wird (1x lesend, 2x schreibend). Die MARC-Philosophie besagt nun, dass solche gleichzeitig bearbeiteten SRAM-Blöcke in unterschiedlichen Speicherbereichen liegen müssen. Insgesamt sind aber maximal vier solcher Bereiche erlaubt. Es stellt sich also die Frage, ob die 10 Blöcke so in 4 Bereiche einsortiert werden können, dass einerseits keine zwei gleichzeitig bearbeiteten Blöcke in ein und demselben Bereich liegen und andererseits die optimale Speicheranordnung „W0 bis W6“ eingehalten werden kann.

Die Blöcke, auf die gleichzeitig zugegriffen wird, lauten:

(T0, T1, T2), (T0, T1, W4), (T2, W5, W6), (T0, W0, W1), (T2, W2, W3).

Blöcke, die in einer Gruppe liegen, dürfen nicht im selben MARC-Speicherbereich angesiedelt sein. Dieses Problem lässt sich als graphentheoretische Fragestellung formulieren:

Gegeben sei ein ungerichteter Graph G mit der Knotenmenge V = {W0, W1, W2, W3, W4, W5, W6, T0, T1, T2} und der Kantenmenge K = {{W0, W1}, {W0, T0}, {W1, T0}, {W2, W3}, {W2, T2}, {W3, T2}, {W4, T0}, {W4, T1}, {W5, W6}, {W5, T2}, {W6, T2}, {T0, T1}, {T0, T2}, {T1, T2}}.

Die Knoten entsprechen dabei den 10 Datenblöcken. Zwischen zwei Knoten existiert eine Kante genau dann, wenn MARC auf die entsprechenden Datenblöcke gleichzeitig zugreift.

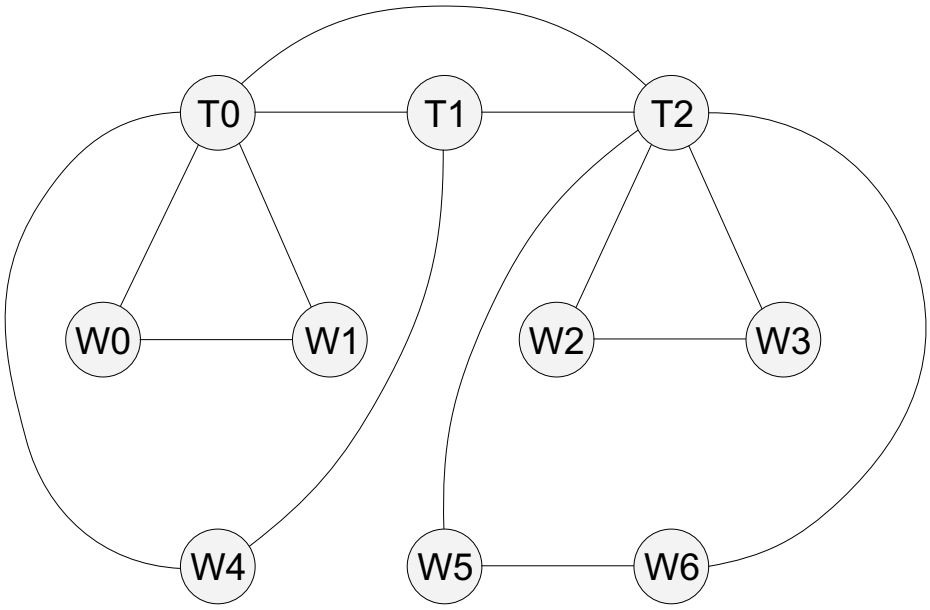


Bild 5.10:
Graph G; Knoten = Speicherblöcke, Kanten = gleichzeitiger Zugriff.

Wir suchen eine Aufteilung der Knoten in 4 Teilmengen, wobei zwischen Knoten aus unterschiedlichen Teilmengen keine Kante verlaufen darf.

Unter Einhaltung der Reihenfolge „W0 bis W6“ ergäben sich unter Beachtung der Restriktionen durch den Graphen die Speicherbereiche (W0), (W1, W2), (W3, W4, W5), (W6). Die drei Blöcke T0, T1, T2 können nur an zwei Stellen in die Sortierung eingefügt werden (Anfang oder Ende), da sonst die Reihenfolge W0, ..., W6 unterbrochen werden würde. Nach dem Schubfachprinzip werden zwei der drei Blöcke am Anfang oder am Ende eingefügt. Dies ist aber verboten, da die drei Blöcke im Graphen eine Clique bilden und deswegen zwingend drei unterschiedlichen Speicherbereichen zugewiesen werden müssen.

Man muss also eine andere Aufteilung wählen. Tatsächlich verwendet wird in der Implementierung die folgende (Bezeichnungen A, B, C, D für die vier MARC-Speicherbereiche):
Es fällt auf, dass die Bereiche stets etwas größer gewählt wurden als nötig. Dies geschah lediglich, um die

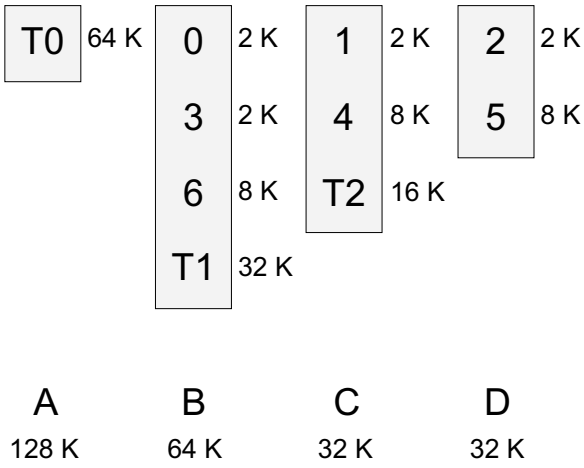


Bild 5.11:
Tatsächliche Einteilung der 10 Blöcke in die 4 möglichen MARC-SRAM-Speicherbereiche.

Konfiguration des MARC-Kerns zu vereinfachen. Der DRAM liegt als zusätzlicher Bereich unterhalb der eingebetteten SRAM-Bereiche. Die genaue Adressierung sieht wie folgt aus:

```
24'h000000 ... 24'hFBFFFF  DRAM via PCI (16MB-256KB)
24'hFC0000 ... 24'hFDFFFF  SRAM A (128 KB)
24'hFE0000 ... 24'hFEFFFF  SRAM B (64 KB)
24'hFF0000 ... 24'hFF7FFF  SRAM C (32 KB)
24'hFF8000 ... 24'hFFFFFF  SRAM D (32 KB)
```

Dadurch lauten die Adressbereiche der einzelnen Blöcke:

```
W0 = 24'hFE0000 ... 24'hFE07FF
W1 = 24'hFF0000 ... 24'hFF07FF
W2 = 24'hFF8000 ... 24'hFF87FF
W3 = 24'hFE0800 ... 24'hFE0FFF
W4 = 24'hFF0800 ... 24'hFF27FF
W5 = 24'hFF8800 ... 24'hFFA7FF
W6 = 24'hFE1000 ... 24'hFE2FFF
T0 = 24'hFC0000 ... 24'hFDFFFF
T1 = 24'hFE3000 ... 24'hFEAFFF
T2 = 24'hFF2800 ... 24'hFF67FF
```

5.3 Stream-Flusskontrolle

Im Abschnitt 5.1.2 über das QZH-Pipelining haben wir bereits eine Idee davon bekommen, wie eine Flusskontrolle funktioniert. Nach demselben Prinzip wird auch der Datenfluss an der MARC-Schnittstelle im Master-Mode gesteuert.

Für den Fall zweier gleichzeitig arbeitender Ströme, ein Lese- und ein Schreibstrom, betrachten wir die MARC-Steuer- und Datensignale und ihre Bedeutung.

Lesestrom:

- *STREAM_ENABLE[0]:*

  Ist STREAM_ENABLE[0] gesetzt, liegen zur nächsten Flanke an STREAM_READ gültige Daten zum Lesen an, aber nur, falls momentan (seit der vergangenen Flanke) STREAM_STALL[0] nicht gesetzt ist. Ist STREAM_ENABLE[0] nicht gesetzt, liegen zur nächsten Flanke an STREAM_READ keine gültigen Daten an.

- *STREAM_STALL[0]:*

  Ist STREAM_STALL[0] gesetzt, liegen zur nächsten Flanke keine gültigen Daten an.

- *STREAM_READ:*

  Lese-Datenport.

Schreibstrom:

- *STREAM_ENABLE[1]:*

  Ist STREAM_ENABLE[1] gesetzt, liegen momentan (seit der vergangenen Flanke) an STREAM_WRITE_PROG gültige Daten zum Schreiben an, aber nur falls momentan (seit der vergangenen Flanke) STREAM_STALL[1] nicht gesetzt ist. Ist STREAM_ENABLE[1] nicht gesetzt, liegen momentan (seit der vergangenen Flanke) an STREAM_WRITE_PROG keine gültigen Daten zum Schreiben an.

- *STREAM_STALL[1]:*

  Ist STREAM_STALL[1] gesetzt, liegen momentan (seit der letzten Flanke) keine gültigen Daten zum Schreiben an.

- *STREAM_WRITE_PROG:*

  Schreib-Datenport.

Die Schaltung dieser Studienarbeit lässt sich intuitiver programmieren und einfacher debuggen, wenn sich die Signale Stall und Enable des Lesestroms - genau wie beim Schreibstrom auch - auf den aktuellen Takt (die vergangene Flanke) beziehen. Das Modul read_flow_control ist gerade für diese Übersetzungen entworfen worden. Es dient aber auch zur Verzögerung der Signale, die die Grenze zwischen MARC und der Schaltung passieren, um kritische Pfade zu verkürzen. Analog befindet sich zwischen je einem der beiden Schreibports der Schaltung und dem zugehörigen MARC-Port eine Instanz des Moduls write_flow_control.

read_flow_control und write_flow_control betreiben demnach keine Datenverarbeitung im eigentlichen Sinn. Sie sind lediglich für die Weiterleitung von Signalen zu den jeweils korrekten Zeitpunkten zuständig. Um dies deutlich zu machen, unterscheiden sich die Stream-Schnittstellen der Flusskontroll-Module von denen der QZH-Module. Z. B. hat quantization die Ausgänge STREAM_ENABLE[0] für den Lesestrom und STREAM_ENABLE[1] für den Schreibstrom sowie die Eingänge STREAM_STALL[0] des Lesestroms und STREAM_STALL[1] des Schreibstroms. read_flow_control hat hingegen auf der Leseseite (MARC) den Eingang MARC_STALL und den

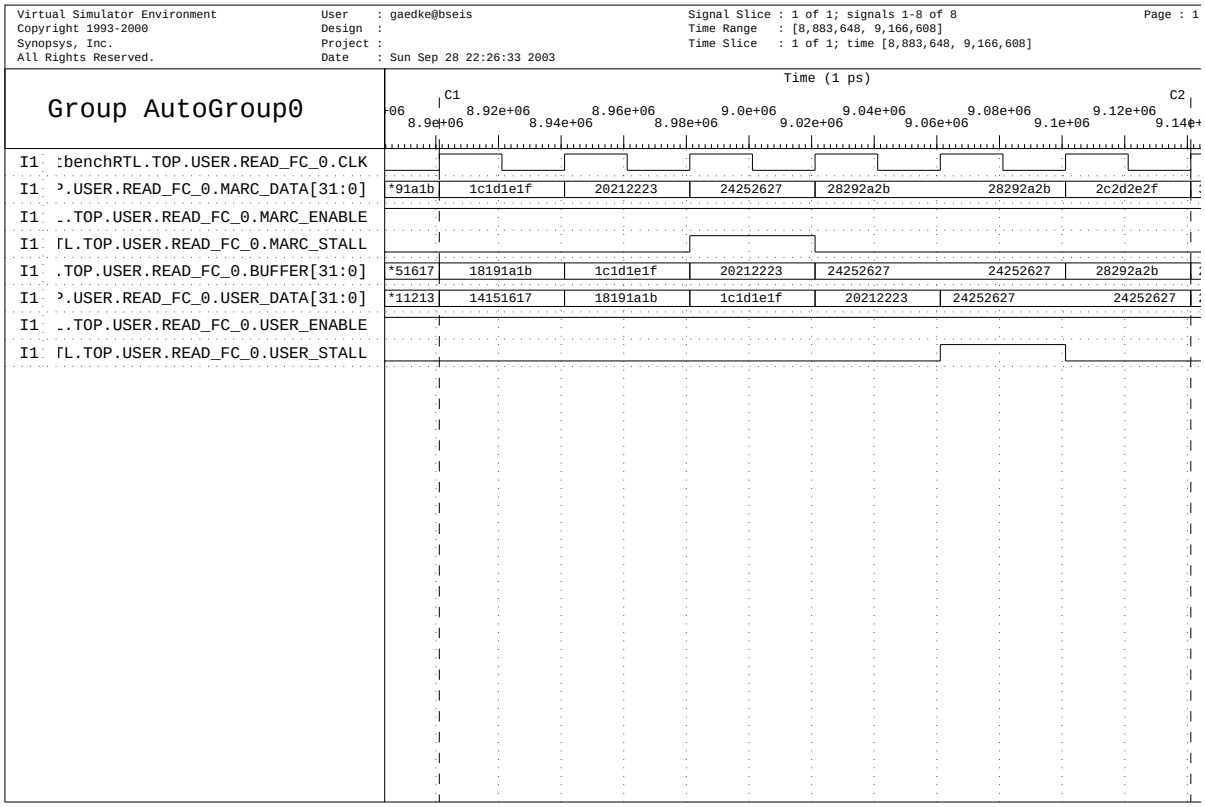


Bild 5.12:
Eine RTL-Simulationsausgabe für das Modul read_flow_control.

Ausgang MARC_ENABLE, und auf der Schreibseite (user) den Eingang USER_ENABLE und den Ausgang USER_STALL.

In Bild 5.12 sehen wir eine Simulationsausgabe des Moduls read_flow_control als Waveform. Der oberste Signalverlauf gibt den Takt wieder. Die drei direkt darunter liegenden Signale MARC_DATA, MARC_ENABLE und MARC_STALL sind Signale der MARC-Schnittstelle, der fünfte Signalverlauf stellt den Inhalt des Puffers innerhalb read_flow_control dar, und die untersten drei Signale sind die entsprechenden Daten- Enable- und Stall-Signale der user-Schaltung. Wir nummerieren in Gedanken die Takte zwischen den vertikalen gestrichelten Begrenzungslinien mit den Nummern 1 bis 6 durch. Der Stall des MARC-Stroms in Takt 3 bedeutet, dass das Datum „28292a2b“ in Takt 4 ungültig ist. Es wird daher zum Takt 5 nicht in den modulinternen Puffer geschrieben. In Takt 4 schreibt read_flow_control das noch im Puffer gehaltene Datum „20212223“ und meldet der user-Schaltung erst in Takt 5 einen Stall, um den Puffer zum Takt 6 hin wieder mit einem neuen Datum zu füllen.

Hervorgehoben sei anhand des Bildes die unterschiedliche Bedeutung der Stall-Signale: MARC meldet in Takt 3 einen Stall und sagt damit aus, dass das in Takt 4 anliegende Datum „28292a2b“ ungültig ist. read_flow_control gibt der user-Schaltung in Takt 5 einen Stall und meint damit, dass das in Takt 5 anliegende Datum „24252627“ ungültig ist.

Um die Schaltung für eine höhere Taktung zu entwerfen, war es nötig, einige kritische Pfade zu verkürzen. Die längsten Pfade verliefen von der Anwendung über die Flusskontrolle zu MARC und wieder zurück in die Anwendung. Ein idealer Punkt für eine Pfadaufspaltung war daher die Flusskontrolle selbst, und zwar im Lese- wie im Schreibfall. In read_flow_control und write_flow_control wurde also eine Verzögerungsstufe zur Pfadverkürzung eingearbeitet.

5.4 Der Hardware-Teil im Überblick

Um nicht nur Detaileinblicke wie in Abschnitt 5.5 zu geben, wird die Hardware hier global beschrieben. So bekommt der Leser einen Überblick über die verschiedenen Modulinstanzen und deren Zusammenwirken.

Zunächst nennen wir alle zur Anwendung gehörenden Module bzw. Modulinstanzen (Module ohne eigenen Instanznamen wurden je nur einmal instanziert):

- wavelet_8_4_l
- wavelet_16_2_hl
- wavelet_controller
- quantization
- zle
- huffman
- huffman_lookup
- qzh_controller
- qzh
- min_max
- compare_16b_neg
- parameter_storage, Instanz Blockthresh)
- parameter_storage, Instanz ZLE_sizes)
- result_params_mux
- result_controller
- read_flow_control
- write_flow_control, Instanz WRITE_FC_1
- write_flow_control, Instanz WRITE_FC_2
- user

Alle diese Module sind in Bild 5.13 aufgeführt, mit Ausnahme des Moduls result_controller (mehr dazu weiter unten). Das user-Modul ist der globale Rahmen der Anwendung; im Bild wurde er der Übersichtlichkeit halber weggelassen.
Oben in Bild 5.13 ist der Wavelet-Bereich angesiedelt. Je nach Phase wird wavelet_8_4_l oder wavelet_16_2_hl zur Berechnung verwendet. wavelet_controller ist für die Programmierung der Ströme (Streams) zuständig und darf die Flusskontroll-Module read_flow_control und write_flow_control rücksetzen. Zusätzlich steuert dieser Controller die Aktualisierung der Blockminima und -maxima in min_max.

Nach Abschluss der Wavelet-Phasen wird qzh_controller aktiv, der die Programmierung der Streams übernimmt. Der zweite Schreibstrom (Zugriff über write_flow_control, Instanz WRITE_FC_2) wird hierbei im Gegensatz zur Wavelet-Verarbeitung nicht verwendet.

qzh beinhaltet die zur QZH-Pipeline nötigen Modulinstanzen quantization, zle und huffman, die nun die Eingangsdaten in den Ausgabe-Bitstrom umwandeln. Da die QZH-Module für sich selbst feststellen, welcher Block gerade verarbeitet wird und wann die Berechnung beendet ist, werden diese Informationen an den Controller übermittelt, der so vorhandene Ressourcen nutzen kann. Trotzdem behält der Controller insofern die Kontrolle über den QZH-Ablauf, als dass er die gesamte Pipeline anhalten kann, um den Lesestrom zu Anfang jedes Blockes auf die korrekte Startadresse einzustellen.

Wie schon angedeutet bietet read_flow_control zusammen mit den beiden write_flow_control-Instanzen

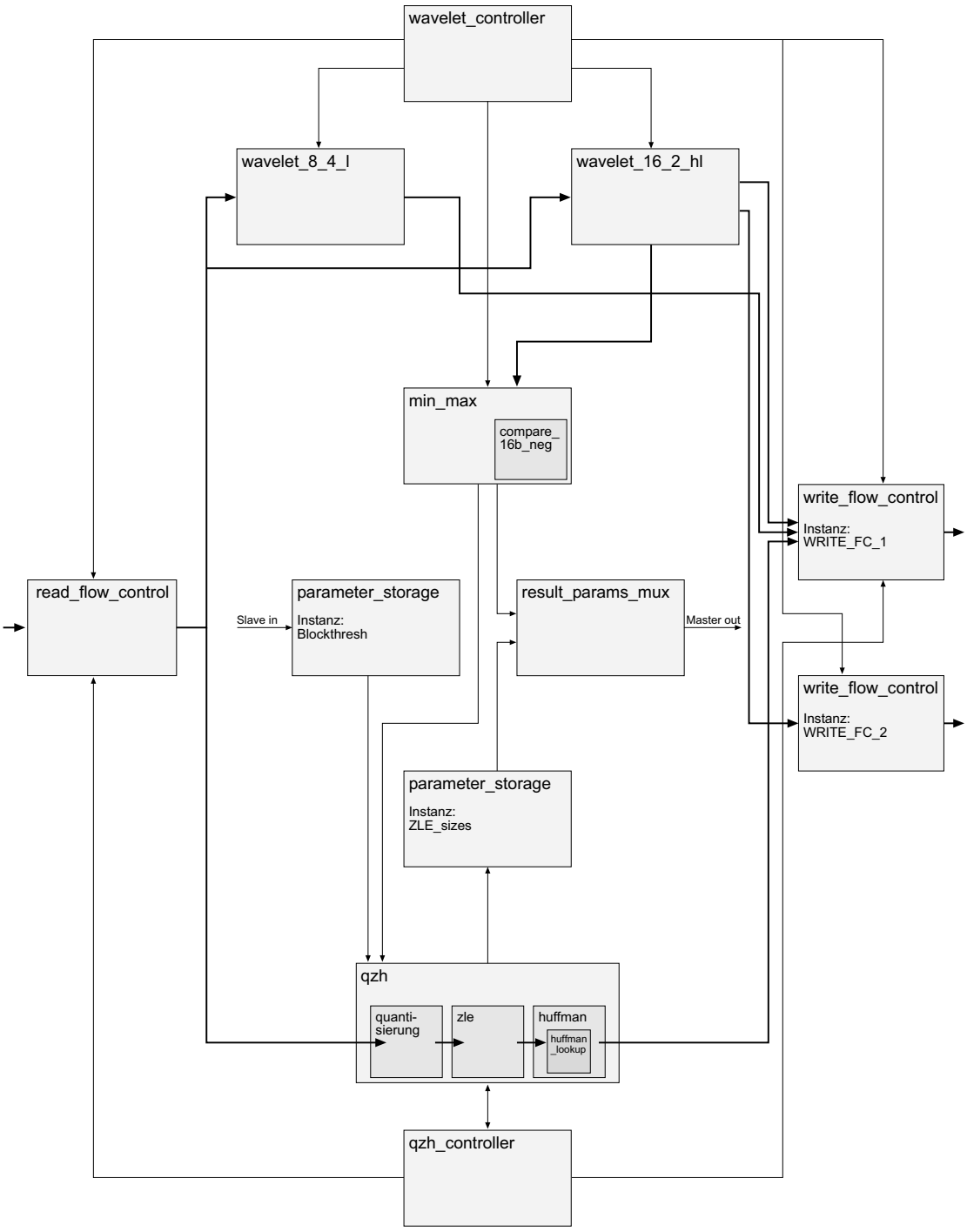


Bild 5.13:
Überblick über den gesamten Hardware-Teil der Anwendung. Fette Linien deuten die Wege der Bilddaten an. Aus Gründen der Übersichtlichkeit wurde das Modul result_controller hier ausgelassen.

die Schnittstelle zu den drei MARC-Strömen an. MARC-Streams werden traditionell durchnummeriert; Stream 0 wird hier nur als Lesestrom genutzt, die Streams 1 und 2 stellen die beiden über die write_flow-control-Instanzen WRITE_FC_1 und WRITE_FC_2 ansprechbaren Schreibströme dar. Der zweite Schreibstrom wird lediglich während den Wavelet-Phasen verwendet, und selbst dann nur in den Phasen 3 bis 6, wenn tatsächlich High-Daten geschrieben werden.

Die vier in der Mitte des Bildes befindlichen Modulinstanzen dienen der Speicherung und Ausgabe der Ergebnisparameter. parameter_storage wird zweimal instanziert: Blockthresh speichert die im Slave-Mode von der Software übertragenen Blockgrenzwerte, ZLE_sizes nimmt die 7 ZLE-Blockgrößen und

auch die Huffman-Bytelänge auf. min_max hält die während der Wavelet-Verarbeitung aktualisierten Blockminima und -maxima, die zur QZH-Phase dem Modul qzh zugänglich gemacht werden. Für eine geordnete Ausgabe der Ergebnisparameter sorgt das Modul result_params_mux. Nachdem die QZH-Phase beendet ist, steuert das Modul result_controller das Kopieren der Ergebnisparameter in den DRAM im Master-Mode (benutzt wird Schreibstrom 1). Da diese Übertragung kein wesentlicher Teil der eigentlichen Datentransformationen ist, wurde result_controller im Schaubild 5.13 fortgelassen; so ist das Bild auch übersichtlicher.

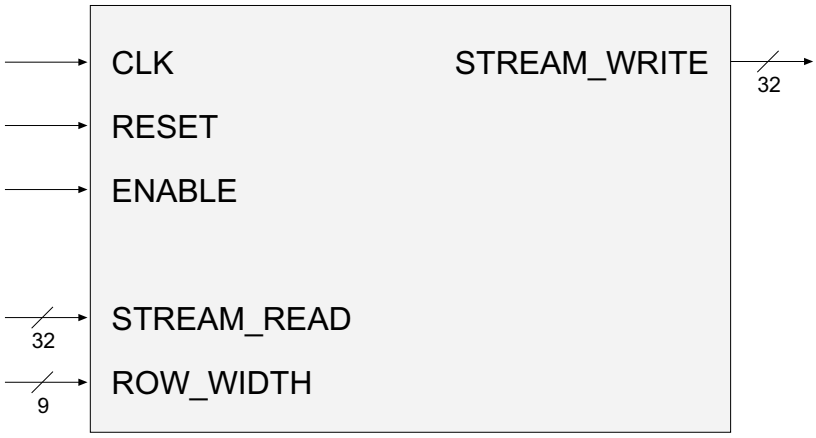
5.5 Detaillierte Modulbeschreibungen

5.5.1 wavelet_8_4_l

Funktionsbeschreibung:

wavelet_8_4_l ist ein Modul zur Durchführung einer Wavelet-Transformation nach dem (2, 2)-biorthogonal-Cohen-Daubechies-Fouveau-Verfahren [7]. Es werden dabei lediglich ganzzahlige Werte verwendet. Pro Takt wird ein 32-Bit-Datenwort (4 Bildpixel zu je 8 Bit) gelesen; geschrieben werden nur die Low-Daten der Transformation, daher nur 2 transformierte Werte pro Takt, jedoch zu je 16 Bit. So haben Dateneingangs- und Ausgangsport (STREAM_READ und STREAM_WRITE) je 32 Bit.
Der Name „wavelet_8_4_l“ bedeutet, dass jedes eingelesene Bildpixel 8 Bit breit ist, und dass 4 Pixel pro Takt (zusammengefasst in einem 32-Bit-Datenwort) eingelesen werden. Das „l“ steht für „low“: es werden nur die Low-Daten der Berechnung ausgegeben.

Schnittstelle:



CLK	Globale Clock.
RESET	Reset-Signal, getrieben von wavelet_controller.
ENABLE	Enable-Signal, getrieben von wavelet_controller.
STREAM_READ	Dateneingangssignal, von user getrieben. Das erste Datenwort besteht aus den Bits 31:24, das vierte aus den Bits 7:0.
ROW_WIDTH	Gibt die Länge der Zeilen des zu bearbeitenden Bildes an; getrieben von wavelet_controller.
STREAM_WRITE	Datenausgangssignal. Das erste Datenwort besteht aus den Bits 31:16, das zweite aus den Bits 15:0.

Implementierung:

Um eine möglichst effiziente Datenverarbeitung zu gewährleisten, wendet dieses Modul das Pipelining-Prinzip an. Die Daten werden von STREAM_READ in einen Puffer kopiert, dort transformiert und nach STREAM_WRITE geschrieben. Als Puffer wären theoretisch 4 mal 8 Bit (entsprechend einem Takt) ausreichend; jedoch wird hier ein weiterer Takt gepuffert, um einen kritischen Pfad in der Schaltung zu verkürzen. Schematisch lässt sich die Pipeline wie folgt darstellen:

STREAM_WRITE <= {A0, B0, C0, D0} <= {A1, B1, C1, D1} <= STREAM_READ

Takt 1: Kopieren der Eingangsdaten von STREAM_READ nach {A1, B1, C1, D1}.
Takt 2: Kopieren der Daten von {A1, B1, C1, D1} nach {A0, B0, C0, D0}.
Takt 3: Transformation (verwendet zur Berechnung A0, B0, C0, D0, A1 und einen weiteren Wert XB, der weiter unten erläutert wird) sowie Schreiben der Ergebnisse nach STREAM_WRITE.

Um die Wavelet-Transformation, die das Modul durchführt, nachvollziehen zu können, wirft man zunächst einen Blick auf folgendes Datenflussdiagramm:

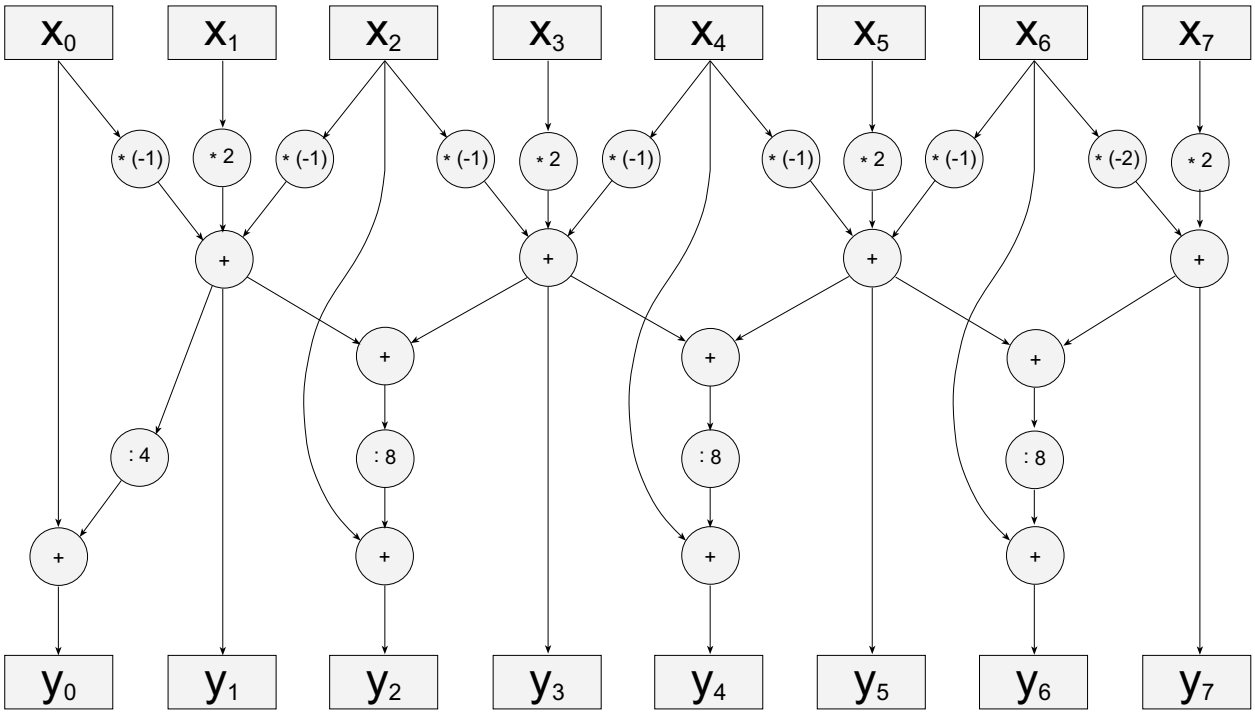


Bild 5.14:
Datenflussdiagramm für die Transformation einer 8 Pixel breiten Bildzeile [x_0, ..., x_7] in Low-Pass-Daten [y_0, y_2, y_4, y_6] und High-Pass-Daten [y_1, y_3, y_5, y_7]. Als Basisfunktionen werden (2, 2)-biorthogonale Cohen-Daubechies-Fouveau-Wavelets verwendet.

Dieses Diagramm erklärt die Berechnungen, die für die verlangte Wavelet-Transformation durchgeführt werden müssen, wobei hier eine Zeilenlänge von 8 Pixeln beispielhaft zugrundegelegt wird. (Die Berechnung lässt sich aber sehr leicht auf n Pixel erweitern, mit n gerade.) Die Eingangsdaten sind hier x_0, x_1, ..., x_7, die entsprechenden Ausgangsdaten sind y_0, y_1, ..., y_7. Der „Low“-Bereich der komprimierten Bildzeile wäre in diesem Fall {y_0, y_2, y_4, y_6}, der „High“-Bereich entsprechend {y_1, y_3, y_5, y_7}. Man sieht deutlich, dass am Zeilenanfang (x_0) und -ende (x_7) die Berechnung anders ist als im restlichen Teil der Zeile. Daher benötigt die berechnende Einheit als Parameter stets die Zeilenlänge.

Wie sieht nun die Realisierung in Hardware aus? Wir erinnern uns daran, dass die Werte {x_0, x_1, x_2, x_3} nach dem zweiten Takt der Pipeline in dem Puffer {A0, B0, C0, D0} liegen. Genauso sind {x_4, x_5, x_6, x_7} in {A1, B1, C1, D1} gespeichert. Die nun im dritten Takt von der Pipeline zu berechnenden Werte sind {y_0, y_1, y_2, y_3}. Die Pipeline-Taktung wird in Bild 5.15 dargestellt: die obere Zeile gibt die Pufferinhalte im Takt n an, die untere zeigt den Zustand im Takt n+1. Wir bemerken hier auch, dass nur die Low-Daten tatsächlich an STREAM_WRITE ausgegeben werden. Die High-Daten werden hier lediglich zur Berechnung der Low-Daten benötigt.

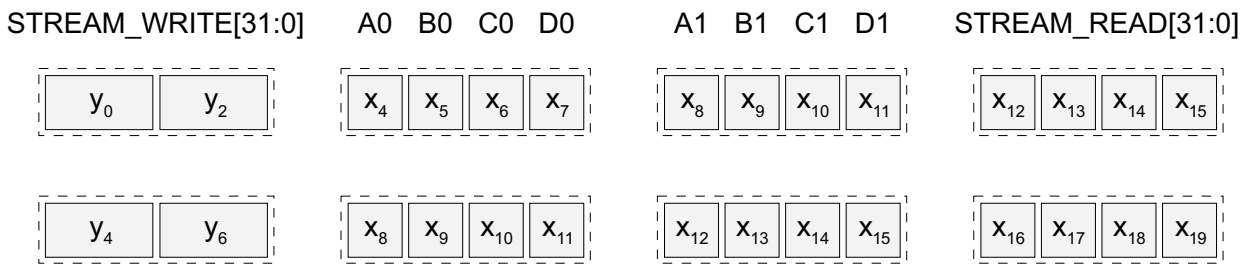


Bild 5.15:
Pipeline-Taktung des Moduls wavelet_8_4_l. Die Registerinhalte einer Zeile gehören zeitlich zusammen.

Die Ausgänge der Logik, die in Hardware die y-Werte berechnet, heißen {A_NEW, B_NEW, C_NEW, D_NEW}; dies gilt fast immer, nur nicht am Zeilenanfang und am Zeilenende. Dort werden die berechneten y-Werte durch {A_NEW_BEGINNING, B_NEW, C_NEW, D_NEW} (Zeilenanfang) bzw. {A_NEW, B_NEW, C_NEW_END, D_NEW_END} (Zeilenende) gegeben.
Die Berechnungsformeln für A_NEW, A_NEW_BEGINNING, B_NEW, C_NEW, C_NEW_END, D_NEW, D_NEW_END lassen sich nun recht schnell dem obigen Datenflussgraphen entnehmen (zur Verwendung des Registers XB: siehe unten):

```
A_NEW            = A0 + ((XB + B_NEW) >> 3);
A_NEW_BEGINNING  = A0 + (B_NEW >> 2);
B_NEW            = (B0 << 1) - A0 - C0;
C_NEW            = ((B_NEW + D_NEW) >> 3) + C0;
C_NEW_END        = ((B_NEW + D_NEW_END) >> 3) + C0;
D_NEW            = (D0 << 1) - C0 - A1;
D_NEW_END        = (D0 - C0) << 1;
```

Wenn wir uns nicht am Zeilenanfang befinden, haben wir bei der Berechnung von A_NEW ein Problem: In der Berechnung von y_0 bzw. A_NEW fehlt für den Operator „+“, dessen Ausgang in den Eingang des Operators „:8“ weist, der erste Summand. An dieser Stelle werden Daten aus dem vorherigen Takt benötigt!

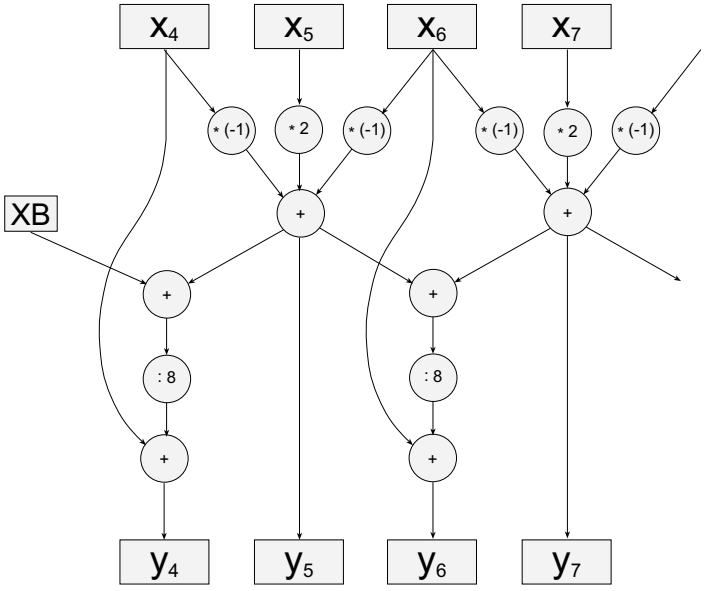


Bild 5.16:
Zugriffe auf bereits abgearbeitete Daten erzwingen die Einführung eines weiteren Registers „XB“.

Als Ausweg dient das Register „XB“ („Extra-Buffer“), das die fehlenden Daten enthält (siehe Bild 5.16). Natürlich muss XB in jedem Takt der korrekte Wert zugewiesen werden. Wir erkennen, dass genau D_NEW der gesuchte Wert ist.

Die oben aufgeführten ständigen Zuweisungen an die Wires A_NEW bis D_NEW enthalten Shifts. Das ist insofern problematisch, als dass Verilog in der hier verwendeten Version alle Zahlen als positive Ganzzahlen betrachtet, in der Wavelet-Transformation aber auch negative Zahlen auftreten. Deswegen müssen „arithmetische“ Shifts selbst implementiert werden. Der von diesem Workaround verursachte, sehr undurchsichtige Verilog-Code, soll an einem Beispiel erklärt werden. Wir betrachten die Zeile

```
A_NEW               =  A0 + ((XB + B_NEW) >> 3);
```

Das Ergebnis der Addition XB + B_NEW wird zunächst in dem Wire XB_PLUS_B_NEW festgehalten:

```
XB_PLUS_B_NEW   =  XB + B_NEW;
```

Um diesen 16-bittigen Wert um 3 Positionen nach rechts zu shiften, werden die Bits 14:3 von (XB_PLUS_B_NEW) den Bits 11:0 des Ergebnisses (XB_PLUS_B_NEW_SHIFTED) zugeordnet. Das Vorzeichenbit bleibt erhalten, und die 3 freien Positionen des Ergebnisses werden mit dem Vorzeichenbit von XB_PLUS_B_NEW aufgefüllt. Auf diese Methode lassen sich positive wie negative Zahlen korrekt rechtsshiften. Der folgende Verilog-Quellcode shiftet XB_PLUS_B_NEW um 3 Positionen nach rechts und speichert das Ergebnis in XB_PLUS_B_NEW_SHIFTED:

```
XB_PLUS_B_NEW_SHIFTED[15]    =  XB_PLUS_B_NEW[15];

XB_PLUS_B_NEW_SHIFTED[14:0]  =  {XB_PLUS_B_NEW[15],
                                 XB_PLUS_B_NEW[15],
                                 XB_PLUS_B_NEW[15],
                                 XB_PLUS_B_NEW[14:3]};
```

Die neue Zuweisung für A_NEW sieht nun so aus:

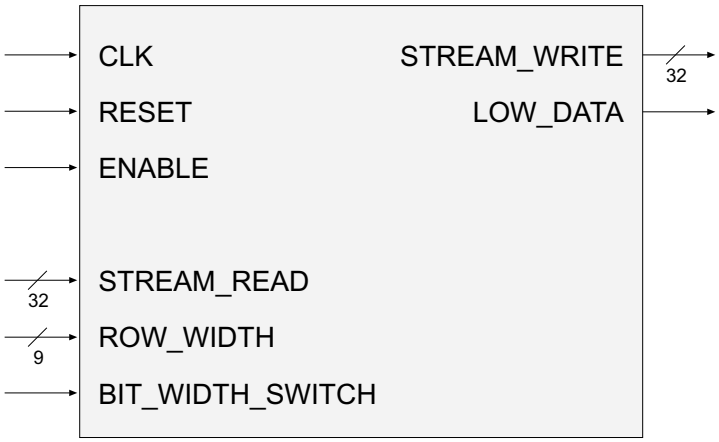
```
A_NEW               =  A0 + XB_PLUS_B_NEW_SHIFTED;
```

5.5.2 wavelet_16_2_hl

Funktionsbeschreibung:

wavelet_16_2_hl führt eine Wavelet-Transformation nach dem (2, 2)-biorthogonal-Cohen-Daubechies-Fouveau-Verfahren [7] durch, ähnlich dem Modul wavelet_8_4_l. Der wesentliche Unterschied ist, dass wavelet_16_2_hl 16-Bit-breite Pixelwerte verarbeitet anstatt 8-Bit-breite Werte.
Das Modul kann in zwei verschiedenen Modi arbeiten, nämlich im 32-Bit-Modus (auch „Zeilenmodus“) oder im 16-Bit-Modus (auch „Spaltenmodus“).
Im Zeilenmodus verarbeitet das Modul pro Takt 2 Pixelwerte zu je 16 Bit, im Spaltenmodus hingegen beschränkt sich wavelet_16_2_hl auf 1 Pixelwert. Low- und High-Daten werden im Zeilenmodus gleichzeitig ausgegeben (1x 16 Bit Low und 1x 16 Bit High), im Spaltenmodus geschieht dies abwechselnd (im Takt i: 1x 16 Bit Low, im Takt i+1: 1x 16 Bit High, im Takt i+2: 1x 16 Bit Low usw.).
Im Spaltenmodus werden stets die unteren 16 Bit der 32-Bit-Eingabe gelesen. Die Ausgabe erfolgt auf die obere und untere Hälfte des Ausgangssignals (die Bits 15:0 werden auf die Bits 31:16 kopiert).

Schnittstelle:



CLK	Globale Clock.
RESET	Reset-Signal, getrieben von wavelet_controller.
ENABLE	Enable-Signal, getrieben von wavelet_controller.
STREAM_READ	Dateneingangssignal, von user getrieben. Das erste Datum besteht aus den Bits 31:16, das vierte aus den Bits 15:0.
ROW_WIDTH	Gibt die Länge der Zeilen des zu bearbeitenden Bildes an; getrieben von wavelet_controller.
BIT_WIDTH_SWITCH	Legt fest, ob das Modul im 32-Bit-Modus / Zeilenmodus (`ROWS, 0) oder im 16-Bit-Modus / Spaltenmodus (`COLUMNS, 1) arbeiten soll.
STREAM_WRITE	Datenausgangssignal. Das erste Datum besteht aus den Bits 31:16, das zweite aus den Bits 15:0. Im Spaltenmodus sind diese beiden 16-Bit-Werte identisch, da dann nur ein 16-Bit-Datum pro Takt ausgegeben wird.
LOW_DATA	Falls sich das Modul im Spaltenmodus befindet, gibt LOW_DATA an, ob gerade Low-Daten (`LOW_DATA, 1) oder High-Daten (`HIGH_DATA, 0) an STREAM_WRITE liegen.

Implementierung:

Um eine möglichst effiziente Datenverarbeitung zu gewährleisten, wendet dieses Modul das Pipelining-Prinzip an. Die Daten werden von STREAM_READ in einen Puffer kopiert, dort transformiert und nach STREAM_WRITE geschrieben.
Zur Bestimmung der optimalen Puffergröße werfen wir einen Blick auf Bild 5.17. Im Gegensatz zu wavelet_8_4_l schreibt wavelet_16_2_hl pro Takt maximal 2 (anstatt 4) Werte, daher genügt ein Puffer für 4 Werte (also 4 mal 16 Bit) plus zwei weitere Werte für noch benötigte Daten aus dem vorhergehenden Takt.
Je nach Betriebsmodus (Zeilen oder Spalten) ändert sich das Schema der Pipelineverarbeitung.
Im Zeilenmodus (32 Bit pro Takt lesen) sieht es wie folgt aus:

STREAM_WRITE <= {A, B} <= {C, D} <= STREAM_READ

Takt 1: Kopieren der Eingangsdaten von STREAM_READ nach {C, D}
Takt 2: Kopieren der Daten von {C, D} nach {A, B}
Takt 3: Transformation (verwendet zur Berechnung A, B, C, D und ein weiteres Register XB („Extra-Buffer“), das weiter unten erläutert wird) sowie Schreiben der Ergebnisse nach STREAM_WRITE.

Im Spaltenmodus können die gleichen Puffer-Register verwendet werden, der Ablauf unterscheidet sich aber vom Zeilenmodus:

STREAM_WRITE[15:0] <= A <= B <= C <= D <= STREAM_READ[15:0]

Takt 1: Kopieren der Eingangsdaten von STREAM_READ[15:0] nach D
Takt 2: Kopieren von D nach C
Takt 3: Kopieren von C nach B
Takt 4: Kopieren von B nach A
Takt 5: Transformation (verwendet zur Berechnung A, B, C, D sowie zwei weitere Werte XB und XB_OLD, die weiter unten erläutert werden) sowie Schreiben der Ergebnisse nach STREAM_WRITE[15:0].

Schauen wir uns nun an, wie die von diesem Modul ausgeführten Berechnungen aussehen; zunächst für den Zeilen-, dann für den Spaltenmodus.

Zeilenmodus:

Im nächsten Bild sieht man Eingangs- und Ausgangsdaten sowie den Pufferinhalt zu zwei aufeinanderfolgenden Takten.

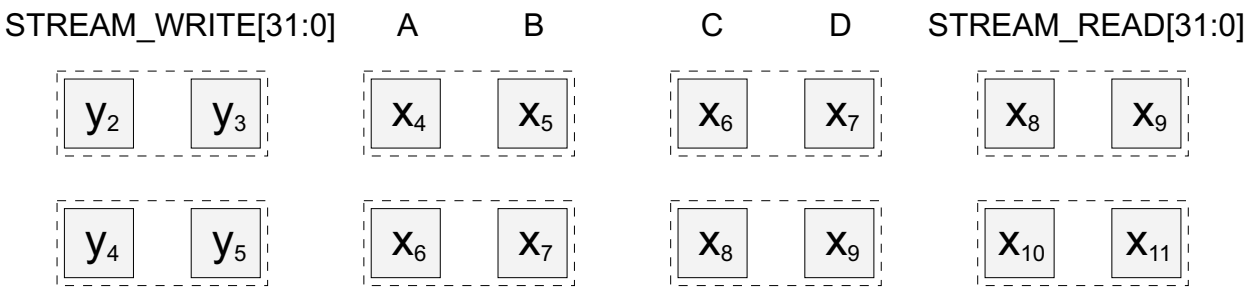


Bild 5.17:
Pipeline-Taktung des Moduls wavelet_16_2_hl im Zeilenmodus. Die Registerinhalte einer Zeile gehören zeitlich zusammen.

Die oberen Kästchen symbolisieren die Inhalte der verschiedenen Puffer zum Takt n (genau genommen wäre es aus Sicht der Pipeline der vierte Takt), die unteren stehen für die Inhalte im Takt n+1. Die gestrichelten Kästchen sollen die Zusammengehörigkeit der enthaltenen Puffer verdeutlichen (von einem Takt auf den nächsten rücken die Daten um ein gestricheltes Kästchen nach links, z. B. $\{x_6, x_7\}$ von {C, D} nach {A, B}).

Unter der Annahme, dass wir uns nicht am Zeilenanfang oder -ende befinden (dies soll durch die Indizierung deutlich werden), lauten die Formeln zur Berechnung von y_4 und y_5 folgendermaßen (siehe auch Bild 5.14):

(1) $y_4 = ((2 * x_3 - x_2 - x_4) + (2 * x_5 - x_4 - x_6)) : 8 + x_4$

(2) $y_5 = (2 * x_5 - x_4 - x_6)$

Wir wollen die Ausgänge der kombinatorischen Logiken, die sich hinter diesen beiden Berechnungen verbergen, mit A_NEW (für y_4) und B_NEW (für y_5) bezeichnen. Man sieht sofort, dass der Term $(2 * x_5 - x_4 - x_6)$ in beiden Berechnungen auftritt und daher tatsächlich nur einmal berechnet werden muss. Der in der Formel für y_4 links stehende Term, $(2 * x_3 - x_2 - x_4)$, benötigt die Werte x_2 und x_3, die vor einem Takt noch in A und B gespeichert, jetzt aber bereits durch neue Werte überschrieben wurden. Hier ist daher eine weitere Puffervariable nötig. Sie heißt XB („Extra-Buffer“) und enthält schon das Ergebnis des kompletten Terms $(2 * x_3 - x_2 - x_4)$. Dies würde durch die Zuweisung

```
XB      <=   2 * B - A - C
```

erreicht werden, aber es geht noch geschickter: B_NEW berechnet bereits den gewünschten Wert! So kommen wir zu folgender Lösung:

```
A_NEW   =    ((XB + B_NEW) >> 3) + A;          // man vergleiche mit (1)
B_NEW   =    (B << 1) - A - C;                 // man vergleiche mit (2)
XB      <=   B_NEW;
```

Durch die Zuweisung

```
STREAM_WRITE[31:0]    <=  {A_NEW, B_NEW};
```

gelangen die gewünschten Resultate zu jedem Takt an den Datenausgang des Moduls. Dennoch sieht die Berechnung am Zeilenanfang und -ende etwas anders aus. Die passenden kombinatorischen Logiken lassen sich analog aus den Bildern 5.14 und 5.17 ableiten und entsprechend STREAM_WRITE zuweisen:

```
STREAM_WRITE[31:0]    <=  {A_NEW_BEGINNING, B_NEW};     // Zeilenanfang
STREAM_WRITE[31:0]    <=  {A_NEW_END, B_NEW_END};       // Zeilenende
```

Die genauen Definitionen von A_NEW_BEGINNING, A_NEW_END und B_NEW_END kann man dem Verilog-Code dieses Moduls entnehmen.

Analog zu wavelet_8_4_l wird hier mit negativen Zahlen gerechnet, daher müssen Shifts eigenhändig implementiert werden; siehe Kommentare in Abschnitt 5.5.1.

Spaltenmodus:

Wir betrachten das nächste Bild, um uns mit den Eigentümlichkeiten dieses Modus vertraut zu machen.

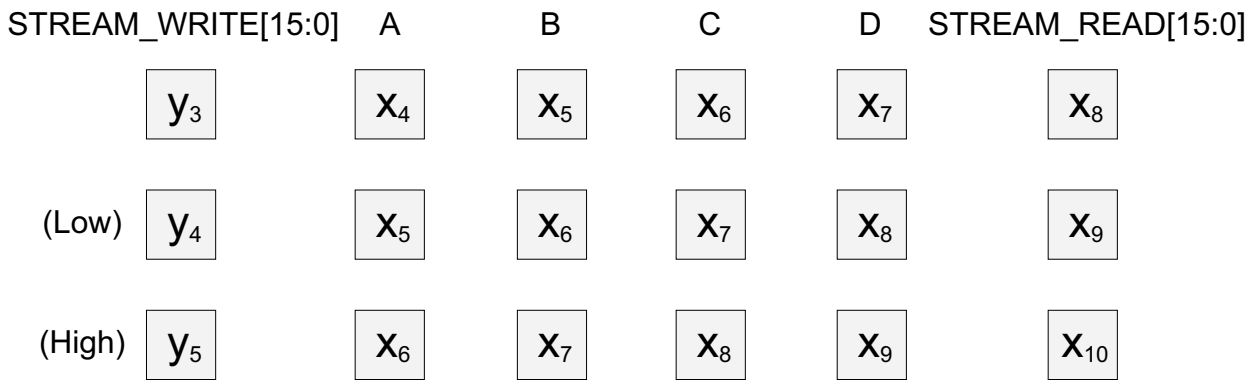


Bild 5.18:
Pipeline-Taktung des Moduls wavelet_16_2_hl im Spaltenmodus. Registerinhalte einer Zeile gehören zeitlich zusammen.

Dargestellt sind wieder die Pufferinhalte zum Takt n und n+1, aber auch zum Takt n+2. Da im Spaltenmodus nur 1 Datum pro Takt gelesen wird, muss der Ausgang zwischen Low- und High-Daten umschalten. Hier werden im Takt n+1 Low-Daten, im Takt n sowie n+2 High-Daten nach STREAM_WRITE geschrieben. Entsprechend muss STREAM_WRITE in „Low-Takten" der Ausgang einer anderen kombinatorischen Logik zugewiesen werden als in „High-Takten". Die Berechnungsvorschriften für y_4 und y_5 ändern sich gegenüber dem Zeilenmodus nicht:

(1) $y_4 = ((2 * x_3 - x_2 - x_4) + (2 * x_5 - x_4 - x_6)) : 8 + x_4$

(2) $y_5 = (2 * x_5 - x_4 - x_6)$

Sehr wohl unterscheidet sich aber die Hardware-Realisierung der Gleichung (2), da y_5 nun einen Takt später berechnet wird als y_4. Die kombinatorische Logik für die Low-Daten-Berechnung soll dem Wire A_NEW_16 zugewiesen werden, die für die High-Daten dem Wire B_NEW_16. A_NEW_16 ist identisch mit A_NEW aus dem Zeilenmodus, bis auf das Register PRE, das hier an Stelle von B_NEW auftritt:

```
A_NEW_16  =   ((XB + PRE) >> 3) + A;          // 16 Bit, low
```

Der Sinn von PRE ist eine Vorberechnung zur Verkürzung eines kritischen Pfades. PRE ist daher mit B_NEW identisch, mit dem Unterschied, dass die benötigten Daten etwas früher verfügbar sind; PRE erhält also zu jedem Takt die Zuweisung

```
PRE         <=  (C << 1) - B - D;             // wie B_NEW, jedoch um 1 Takt nach
                                              // vorne verschoben
```

Einen Takt in die andere Richtung verschoben ist B_NEW_16 gegenüber B_NEW, da diese Berechnung ja gerade einen Takt später ausgeführt wird:

```
B_NEW_16  =   (A << 1) - XB_OLD - B;          // 16 Bit, high
```

Erläuterungsbedarf besteht hier für das Register XB_OLD, welches bisher noch gar nicht auftrat. Es dient dazu, den schon vergessenen Wert von A des vorhergehenden Taktes zu liefern, da dieser für die Rechnung zwingend erforderlich ist (entspricht x_4 in der obigen Formel (2) für y_5). XB_OLD erhält daher zu jedem Takt den Wert von A:

```
XB_OLD     <=  A;
```

Analog zum Zeilenmodus ändern sich die Zuweisungen an STREAM_WRITE[15:0] je nachdem, ob gerade der Zeilenanfang, die Zeilenmitte oder das Zeilenende transformiert wird. Hinzu kommen noch die Fallunterscheidungen für Low- und High-Takte; eine Übersicht über die jeweils passenden Zuweisungen folgt.

```
STREAM_WRITE[15:0]   <=   A_NEW_16_BEGINNING;   // Zeilenanfang, Low
STREAM_WRITE[15:0]   <=   B_NEW_16;             // Zeilenanfang, High
STREAM_WRITE[15:0]   <=   A_NEW_16;             // Zeilenmitte, Low
STREAM_WRITE[15:0]   <=   B_NEW_16;             // Zeilenmitte, High
STREAM_WRITE[15:0]   <=   A_NEW_END;            // Zeilenende, Low
STREAM_WRITE[15:0]   <=   B_NEW_END_16;         // Zeilenende, High
```

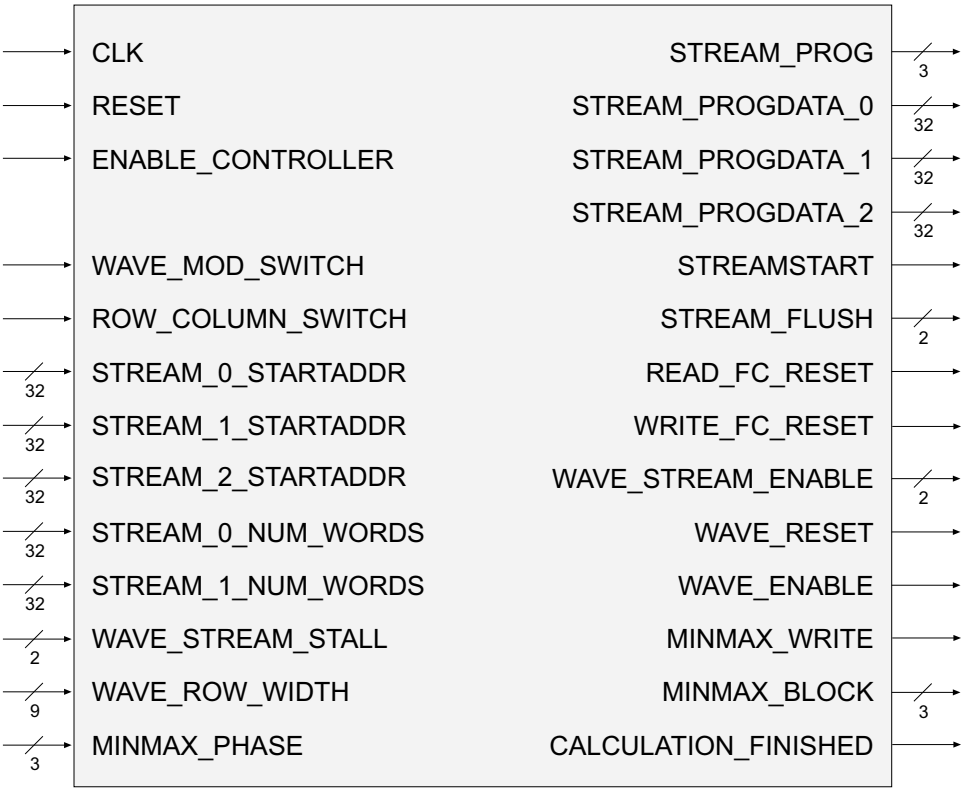
Man beachte, dass einerseits wieder neue kombinatorische Logiken anfallen (A_NEW_16_BEGINNING und B_NEW_END_16) und andererseits A_NEW_END aus dem Zeilenmodus für den Low-Takt am Zeilenende übernommen werden kann.

5.5.3 wavelet_controller

Funktionsbeschreibung:

wavelet_controller programmiert die Ein- und Ausgabedatenströme für die auszuführenden Wavelet-Transformationen und stellt sicher, dass wavelet_8_4_l bzw. wavelet_16_2_hl stets auf gültigen Daten arbeiten (Setzung des Enable-Signals der beiden Wavelet-Module nur dann, falls kein Stall auftritt). Zusätzlich gibt dieses Modul an, ob und für welche Bild-Blöcke während der Verarbeitung die Blockminima und -maxima upgedatet werden.

Schnittstelle:



CLK	Globale Clock.
RESET	Reset-Signal, getrieben von user.
ENABLE_CONTROLLER	Enable-Signal, getrieben von user.
WAVE_MOD_SWITCH	Gibt an, ob wavelet_8_4_l (`WAVE_8) oder wavelet_16_2_hl (`WAVE_16) als Berechnungsmodul verwendet werden soll; getrieben von user.
ROW_COLUMN_SWITCH	Gibt an, ob eine Berechnung im Zeilenmodus (`ROWS) oder im Spaltenmodus (`COLUMNS) durchgeführt werden soll; getrieben von user.
STREAM_0_STARTADDR	Startadresse für den Lesestrom; getrieben von user.
STREAM_1_STARTADDR	Startadresse für den ersten Schreibstrom; getrieben von user.
STREAM_2_STARTADDR	Startadresse für den zweiten Schreibstrom; getrieben von user.

STREAM_0_NUM_WORDS	Anzahl der zu lesenden Datenworte; getrieben von user.
STREAM_1_NUM_WORDS	Anzahl der insgesamt zu schreibenden Datenworte; getrieben von user.
WAVE_STREAM_STALL	Stall-Signale des Lese- und Schreibstroms (WAVE_STREAM_STALL[0] ist gesetzt, wenn im Lesestrom ein Stall auftritt; WAVE_STREAM_STALL[1] ist gesetzt, wenn in Schreibstrom 1 oder in Schreibstrom 2 ein Stall auftritt). Getrieben von read_flow_control bzw. write_flow_control.
WAVE_ROW_WIDTH	Breite der zu transformierenden Zeilen bzw. Höhe der Spalten; getrieben von user.
MINMAX_PHASE	Legt fest, ob und welche Blockextremwerte während der nächsten Transformation aktualisiert werden sollen; getrieben von user. Wenn das höchste Bit gesetzt ist, findet keine Aktualisierung statt; die beiden unteren Bits codieren die im Fall einer Aktualisierung betroffenen Blöcke.
STREAM_PROG	Programmiermodusschalter für die Ströme.
STREAM_PROGDATA_0	Programmier- und Datenleitung für den Lesestrom.
STREAM_PROGDATA_1	Programmier- und Datenleitung für den ersten Schreibstrom.
STREAM_PROGDATA_2	Programmier- und Datenleitung für den zweiten Schreibstrom.
STREAMSTART	Kontrollsignal zum Starten bzw. Anhalten der Ströme.
STREAM_FLUSH	Kontrollsignal zum Leeren der internen Puffer der beiden Schreibströme.
READ_FC_RESET	Reset für Flusskontrolle des Lesestroms.
WRITE_FC_RESET	Reset für Flusskontrolle der beiden Schreibströme.
WAVE_STREAM_ENABLE	Je ein Enable-Signal für Lese- und Schreibstrom; das Schreib-Enable steuert beide Schreibströme gleichzeitig.
WAVE_RESET	Reset für wavelet_8_4_l und wavelet_16_2_hl.
WAVE_ENABLE	Enable für wavelet_8_4_l und wavelet_16_2_hl.
MINMAX_WRITE	Gibt an, ob im aktuellen Takt die Blockminima und -maxima aktualisiert werden dürfen.
MINMAX_BLOCK	Gibt an, für welchen Block gerade Maxima und Minima aktualisiert werden. Die Nummerierung der Blöcke beginnt bei 0.
CALCULATION_FINISHED	Zur Benachrichtigung des user-Moduls, dass die aktuelle Berechnung beendet ist.

Implementierung:

Das wesentliche an diesem Modul ist der Zustandsautomat, zu sehen in Bild 5.19.

Nach dem anfänglichen Reset befindet sich der Automat im Zustand STATE_PROG_PREPARE. Hier werden die Stream-Startadressen und die Register für die Datenzählung auf die korrekten Startwerte gesetzt. Außerdem werden die Register MINMAX_BLOCK_0 und MINMAX_BLOCK_1 mit den Blocknummern beschrieben, deren Minima und Maxima während der aktuellen Wavelet-Phase aktualisiert werden sollen (die jeweiligen Blocknummern entnimmt die Schaltung dem Eingangssignal MINMAX_PHASE).

Es folgen die 5 Standardzustände, die zum Programmieren von MARC-Strömen benötigt werden: STATE_PROG_START (Programmieren der Startadressen), STATE_PROG_COUNT (Angabe der Anzahl der zu verarbeitenden Datenworte), STATE_PROG_STEP (Schrittweite programmieren), STATE_PROG_WIDTH (Bitbreite der Datenworte festlegen) und STATE_PROG_MODE (lesenden oder schreibenden Zugriff für Streams einstellen).

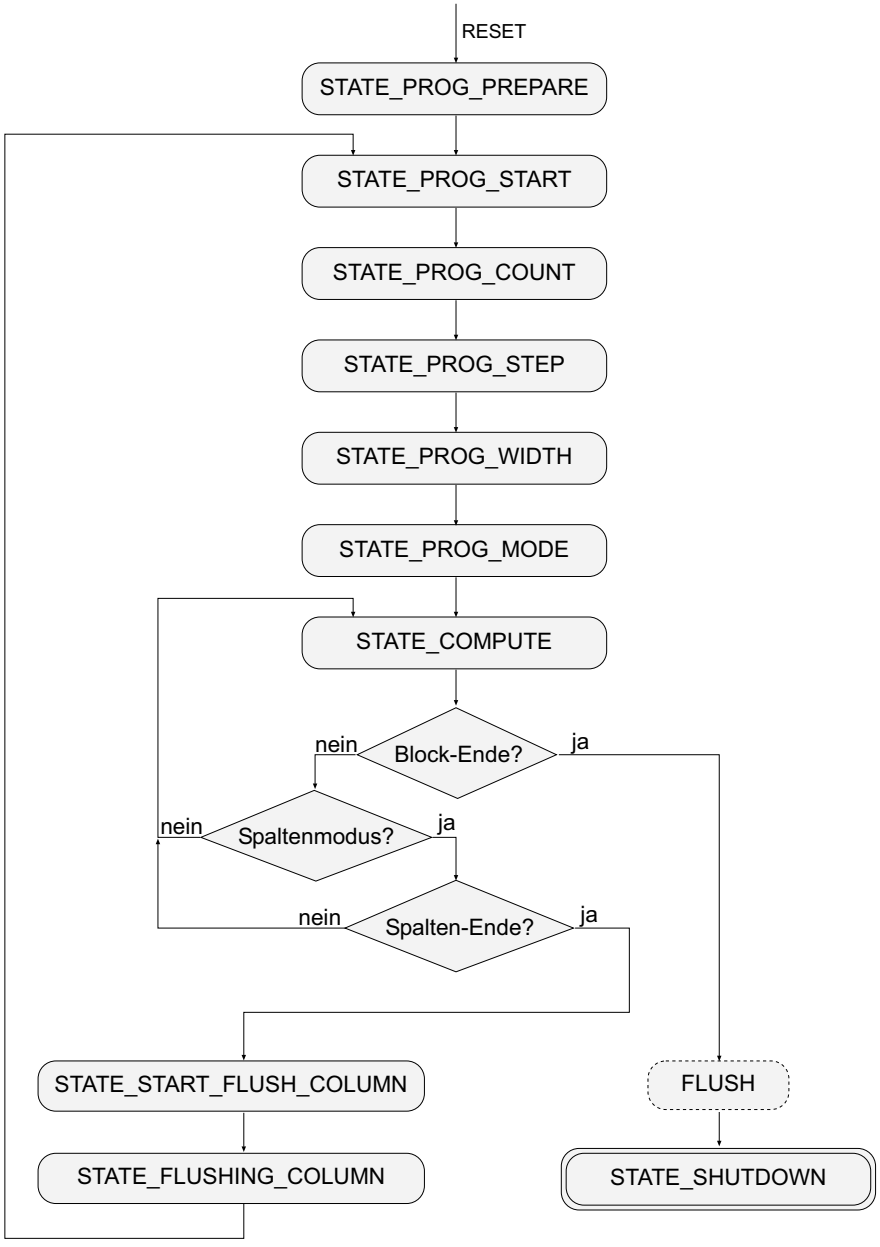


Bild 5.19:
Zustandautomat des Moduls wavelet_controller.

Danach gelangt man zum umfangreichsten Teil: STATE_COMPUTE; hier werden die Streams nach dem Programmieren gestartet. Bei dieser Anwendung müssen die Schreibströme etwas zeitversetzt zu dem Lesestrom gestartet werden, da die Pipeline der Waveletmodule erst gefüllt werden muss, bevor die bearbeiteten Daten dann am Ausgang anliegen. Eine weitere Aufgabe der Schaltung im Compute-Zustand ist die Kontrolle, ob der aktuelle Datenblock zuende ist, d. h. ob alle zu schreibenden Daten wirklich an den MARC-Stream geschrieben wurden. In diesem Fall müssen die Streams nämlich angehalten werden, und es wird nach dem Flushen der Schreibströme (geschieht praktisch noch in STATE_COMPUTE, daher die gestrichelte Umkreisung im Diagramm) in den End-Zustand STATE_SHUTDOWN gewechselt.

Im Spaltenmodus kommt für STATE_COMPUTE noch eine kleine Erweiterung hinzu: da ein Spaltendurchlauf praktisch einen gesamten Blockdurchlauf (mit Auslassung der ungewollten Daten) darstellt, müssen Lese- und Schreibströme für jede Spalte neu programmiert werden. D. h., während STATE_COMPUTE wird im Spaltenmodus überprüft, ob gerade eine Spalte zuende ist. Falls ja, werden die innerhalb von MARC noch gepufferten Daten in den beiden nächsten Zuständen (STATE_START_FLUSH_COLUMN und STATE_FLUSHING_COLUMN) geschrieben. Dort findet auch das Inkrementieren der Startadressen für den nächsten Spaltendurchlauf statt, bevor dann erneut das Programmieren der Streams einsetzt (Wechsel nach STATE_PROG_START).

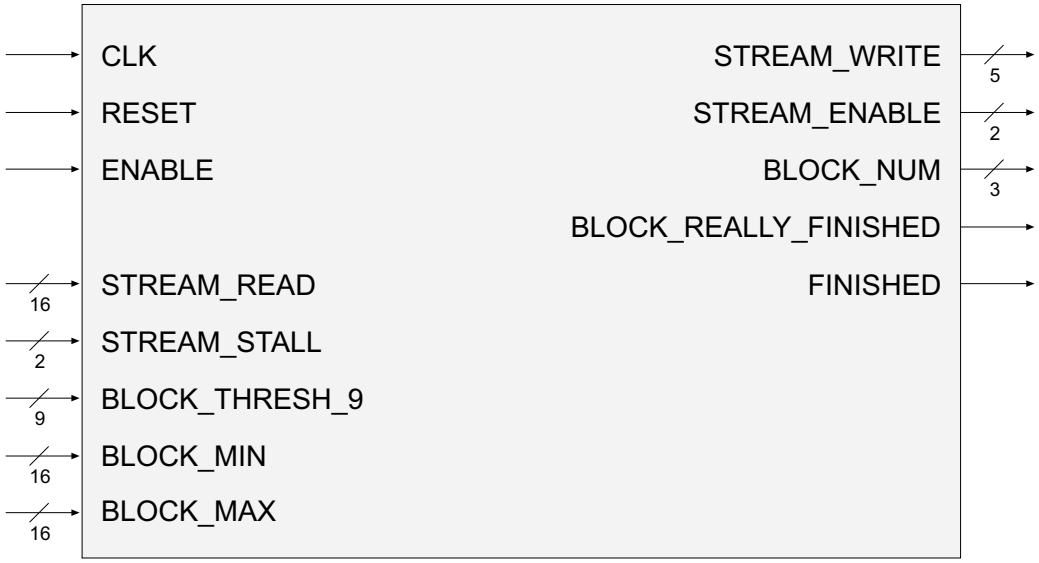
Abseits vom Zustandsautomaten wird durch ständige Zuweisungen einerseits eine lokale Flusskontrolle durchgeführt (Setzen von WAVE_STREAM_ENABLE nur dann, wenn der jeweilige Strom auch arbeiten soll, sowie Reaktion auf Stream-Stalls). Andererseits werden die Module wavelet_8_4_l und wavelet_16_2_hl während eines Stalls über das Rücksetzen des Kontrollsignals WAVE_ENABLE angehalten. Interessant ist, dass die beiden genannten Wavelet-Module zu Beginn eines Datenstroms „immun“ gegen Schreib-Stalls sind, da sie ja erst einmal ihre Pipeline füllen, bevor sie überhaupt schreiben möchten. Analog gilt dies auch für Lese-Stalls während des Berechnungsendes.

5.5.4 quantization

Funktionsbeschreibung:

quantization quantisiert 16-Bit-Daten in 4-Bit-Daten. Die Datenblöcke W0 bis W6 werden einzeln quantisiert, d. h. es gibt für jeden Block ein Minimum und ein Maximum, welche die Quantisierungsgrenzen bestimmen (die Maxima und Minima wurden während der Wavelet-Transformation gespeichert und können nun abgefragt werden). Ein Zusatzfeature dieser Quantisierung bilden die Blockgrenzwerte (BLOCK_THRESH): Eingangsdaten, die betragsmäßig diese vorgegebenen Grenzen nicht übersteigen, werden besonders gekennzeichnet, indem ihnen der Quantisierungswert ZERO_MARK zugeordnet wird. Zusätzlich zu dem Qualitätsverlust durch die eigentliche Quantisierung leidet die Bildqualität durch höhere Blockgrenzwerte also noch stärker; dafür lässt sich so aber auch eine deutlich effektivere Komprimierung erreichen, denn die entstandenen ZERO_MARKs können in der Lauflängencodierung sehr kompakt codiert werden.

Schnittstelle:



CLK	Globale Clock.
RESET	Reset-Signal, getrieben von user.
ENABLE	Enable-Signal, getrieben von user.
STREAM_READ	Dateneingang, getrieben von read_flow_control.
STREAM_STALL	Stall-Signale für Lese- und Schreibstrom, getrieben von read_flow_control bzw. zle.
BLOCK_THRESH_9	9-bittiger Blockgrenzwert des aktuellen Blocks. 9 Bits, weil die Blockgrenzwerte selbst für die höchste einstellbare Komprimierungsstärke nicht größer werden als 380, das ist in binärer Schreibweise die 9-bittige Zahl 101111100. Getrieben wird dieses Signal von der 9-bittigen Instanz Blockthresh des Moduls parameter_storage.
BLOCK_MIN	Minimum des aktuellen Blocks, getrieben von min_max.
BLOCK_MAX	Maximum des aktuellen Blocks, getrieben von min_max.

STREAM_WRITE	Datenausgang; die eigentlichen Daten befinden sich in den unteren 4 Bits, das fünfte Bit dient lediglich der ZERO_MARK-Kennzeichnung.
STREAM_ENABLE	Enable-Signale für Lese- und Schreibstrom.
BLOCK_NUM	Nummer des aktuell zu bearbeitenden Blocks; gültig sind die Blocknummern 0 bis 6 (für die Wavelet-Blöcke W0 bis W6).
BLOCK_REALLY_FINISHED	Ein Kontrollsignal, das gesetzt ist, wenn das letzte gültige Datum des aktuellen Blockes am Ausgang liegt.
FINISHED	Ein Kontrollsignal, das gesetzt ist, wenn das letzte gültige Datum des letzten Blockes am Ausgang liegt.

Implementierung:

Die Pipeline dieses Moduls ist mit einem Puffer nur zwei Takte lang, trotzdem muss zur optimalen Flusskontrolle zwischen den Zuständen „Pipeline wird gefüllt", „Pipeline liest und schreibt" und „Pipeline wird geleert" unterschieden werden. Daher treten die drei entsprechenden Zustände QUANT_PIPE_STATE_FILL, QUANT_PIPE_STATE_WORK und QUANT_PIPE_STATE_FLUSH im Zustandsautomaten dieses Moduls auf. Zusammen bilden sie die Hauptbearbeitungsphase, in der nach dem Blockgrenzwert-Test („Ist das gerade gelesene Datum betragsmäßig größer als der aktuelle Blockgrenzwert?") die Quantisierung mittels einer baumartig strukturierten if-Bedingung erschlagen wird. Hier wird auch überprüft, ob ein Datenblock beendet ist, um in diesem Fall zum nächsten zu wechseln bzw. in den Endzustand überzugehen, falls der letzte Datenblock beendet wurde.

Beim Übergang zu einem neuen Block müssen - wie auch schon vor dem ersten Block - die Quantisierungsgrenzen (THRESH_1 bis THRESH_15) anhand der aktuellen Blockminima und -maxima berechnet werden. Der Zustandsautomat geht dazu in den Zustand QUANT_PIPE_STATE_CALC_THRESH über. Dort wird die folgende Schleife berechnet:

```
for (i = 1; i < 16; i++)
    thresh[i] = minval + ((i * (maxval - minval) + 8) >> 4);
```

Naiv betrachtet wären dafür 15 Multiplizierer oder 1 Multiplizierer, der 15 mal hintereinander benutzt wird, nötig, mit der entsprechenden Berechnungszeit.
Tatsächlich geht es aber wesentlich einfacher und schneller. Bezeichnet man die auftretenden Produkte im obigen Term mit

```
prod[i] = i * (maxval - minval)
```

so fällt sofort die Abhängigkeit

```
prod[i+1] = prod[i] + (maxval - minval)
```

auf. Wenn man zusätzlich bedenkt, dass sich prod[2], prod[4] und prod[8] mittels Shifts in einem Takt berechnen lassen (was die additive Zusammensetzung der übrigen prod-Werte deutlich erleichtert), erkennt man, dass für THRESH_1 bis THRESH_15 eine Berechnungszeit von nur 3 Takten notwendig ist, plus 2 weitere Takte für das parallele Addieren von 8, Rechtsshiften um 4 und Addieren des Minimums. (Die 3 Schritte wurden auf 2 Takte verteilt, um keine zu langen Pfadverzögerungen zu produzieren.)

Ein Detail der Quantisierung soll hier nicht verschwiegen werden. In der Software-Vorlage werden Größenvergleiche der Art

(1) if (STREAM_READ > THRESH_1)

vorgenommen. In der Hardware-Realisierung geht das nicht so einfach, da STREAM_READ negativ sein kann (negative Werte würden von der verwendeten Verilog-Version als sehr große positive Werte interpretiert werden). Der Vergleich wird daher umgeformt:

```
        STREAM_READ > THRESH_1
<=>     STREAM_READ - THRESH_1 > 0
<=>     STREAM_READ - THRESH_1 - 1 >= 0
<=>     STREAM_READ - THRESH_1 - 1
<=>     THRESH_1_DIFF[15] = 0
```

wobei THRESH_1_DIFF = STREAM_READ - THRESH_1 - 1 ist. So wird aus (1) der Vergleich

(2) if (THRESH_1_DIFF[15] == 0)

welcher sich sehr schön in der Hardware realisieren lässt.

5.5.5 zle

Funktionsbeschreibung:

In dieser speziellen Form der Lauflängencodierung (englisch: „run length encoding“, RLE) werden nur diejenigen Eingangsdaten in einem „Run“ gezählt, die von der Quantisierung mit dem Wert ZERO_MARK gekennzeichnet wurden. Daher spricht man auch von einem „Zero-Length-Encoding“-Verfahren, ZLE. Non-ZERO_MARK-Eingangsdaten werden schlicht zum Datenausgang durchgereicht. Gelesen werden 5-Bit-Worte (4 Bit Daten plus 1 Bit zur ZERO_MARK-Markierung), geschrieben werden 8-Bit-Worte, wobei pro Takt ein neues Wort eingelesen wird. Ob auch ein Ausgabewort zu schreiben ist, hängt vom Inhalt der gelesenen Daten ab: wird ein ZERO_MARK gelesen, so schreibt zle im nächsten Takt nicht, es wird lediglich der ZERO_MARK-Zähler für den aktuellen Run um 1 erhöht. Das nächste zu schreibende Datum wird erst nach dem Ende des Runs anfallen, und beinhaltet die Run-Länge + 15 (die Ausgaben 0 bis 15 sind für die übrigen Daten reserviert, die keine Runs produzieren, aber auch keine Run-Längen darstellen sollen).

Schnittstelle:

CLK	Globale Clock.
RESET	Reset-Signal, getrieben von user.
ENABLE	Enable-Signal, getrieben von user.
STREAM_READ	Dateneingang, getrieben von quantization.
STREAM_STALL	Stall-Signale für Lese- und Schreibstrom, getrieben von quantization bzw. huffman.
STREAM_WRITE	Ausgangsdaten.
STREAM_ENABLE	Enables für Lese- und Schreibstrom.
BLOCK_NUM	Nummer (0 bis 6) des aktuellen Blocks.
BLOCK_WRITE	Dieses Signal wird jeweils am Blockende gesetzt, wenn COUNT_OUTWORDS die Zahl der Ausgabeworte des aktuellen Blocks angibt. (Dies hat zur Folge, dass der aktuelle Wert von COUNT_OUTWORDS in der 16-Bit-breiten Instanz ZLE_sizes des Moduls parameter_storage als "ZLE-Größe" des aktuellen Blocks gespeichert wird.)
FINISHED	Ist gesetzt, sobald das letzte gültige Datum am Ausgang anliegt.

Implementierung:

Zum Zählen der Run-Längen wird ein 8-bittiges Register BUFFER verwendet. Falls gerade kein Run vorliegt, puffert es die Lesedaten, um sie dann an STREAM_WRITE weiterzugeben. Ob die in STREAM_WRITE enthaltenen Daten geschrieben werden, bestimmt das Register WRITE_SWITCH. Es geht daher direkt in das Enable des Schreibstroms mit ein. Hier der passende Ausschnitt aus dem Quellcode:

```
assign STREAM_ENABLE[1] =
   (PIPE_STATE == `ZLE_PIPE_STATE_FILL)
               ? 0
   : (PIPE_STATE == `ZLE_PIPE_STATE_FILL_AND_WRITE)
               ? ~STREAM_STALL[0]
   : (PIPE_STATE == `ZLE_PIPE_STATE_WORK)
               ? (~STREAM_STALL[0] & WRITE_SWITCH)
   : (PIPE_STATE == `ZLE_PIPE_STATE_NEXT_BLOCK)
               ? WRITE_SWITCH
   : (PIPE_STATE == `ZLE_PIPE_STATE_FLUSH)
               ? WRITE_SWITCH
   : (PIPE_STATE == `ZLE_PIPE_STATE_END)
               ? 1
   : 1;
```

WRITE_SWITCH ist hier nur für den Zustand ZLE_PIPE_STATE_WORK relevant, da in allen anderen Zuständen von vornherein klar ist, ob geschrieben werden soll oder nicht. An dem Code-Stück sieht man auch die definierten Zustände des Automaten für dieses Modul. Da die Namensgebung analog zu den anderen Modulen (z. B. quantization) ist, wird die Bedeutung dieser Zustände unmittelbar klar, bis auf

die beiden Fälle ZLE_PIPE_STATE_FILL_AND_WRITE und ZLE_PIPE_STATE_NEXT_BLOCK. Während ZLE_PIPE_STATE_NEXT_BLOCK wird das letzte noch im BUFFER befindliche Datum STREAM_WRITE zugewiesen und anschließend während ZLE_PIPE_STATE_FILL_AND_WRITE von STREAM_WRITE aus dem Modul heraustransportiert (Enable des Schreibstroms). In letzterem Zustand wird aber auch schon ein Datum vom Lesestrom gelesen, um den BUFFER wieder für die „normale“ Bearbeitungsphase (ZLE_PIPE_STATE_WORK) vorzubereiten.

Während ZLE_PIPE_STATE_WORK müssen zwei wesentliche Fallunterscheidungen gemacht werden. Erstens: wird gerade ein ZERO_MARK gelesen oder nicht? Und zweitens: wird bereits an einem Run gezählt oder nicht? Entsprechend wird dann mit dem Zählen begonnen bzw. fortgefahren oder nicht bzw. das Zählen wird beendet. Hinzu kommt die Schwierigkeit, dass ein Run nicht nur durch einen eingelesenen Wert ungleich ZERO_MARK abgebrochen wird, sondern auch durch das Ende des aktuellen Blocks sowie durch eine zu lange Folge von ZERO_MARKs (ein Run darf maximal 256 - 16 = 240 Positionen lang sein). Die dazu notwendigen Abfragen wurden in der vorliegenden Realisierung eingebaut.

Für die Ausgabe der Signale BLOCK_NUM, BLOCK_WRITE und FINISHED muss zle wissen, wann genau ein Eingabe-Datenblock zuende ist. Dazu wurde das Zählregister COUNT_INWORDS eingerichtet. Die Anzahl der Eingabeworte pro Block ist im Modul fest codiert und kann stets mit COUNT_INWORDS auf Übereinstimmung verglichen werden.

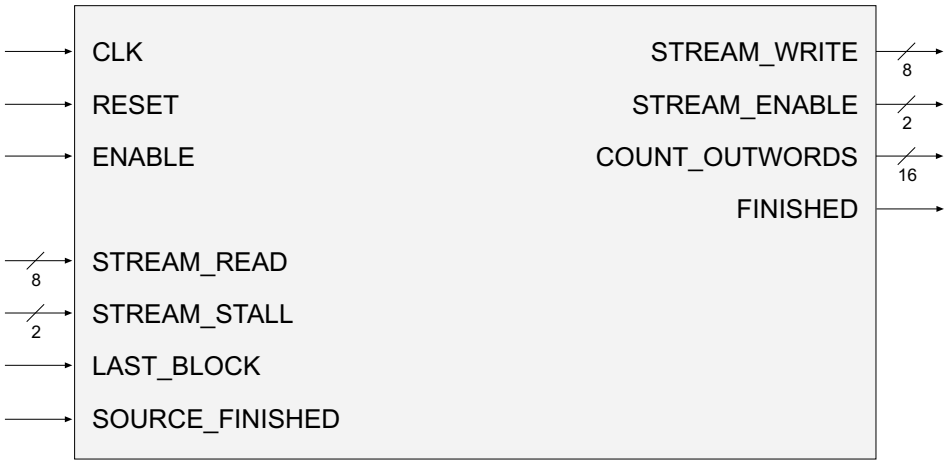
5.5.6 huffman

Funktionsbeschreibung:

huffman implementiert die Huffman-Codierung von 8-bittigen Eingangsdaten. Die Huffman-Tabelle ist dabei fest vorgegeben, so dass man diese Codierung auch als einfache Look-Up-Codierung beschreiben könnte. Die codierten Werte sind zwischen 3 und 18 Bit breit. Warum gerade diese spezielle Tabelle verwendet wird, lässt sich theoretisch nicht vollständig beantworten. Es gibt dazu nur die empirische Aussage: „Sie hat sich bisher in der Bildverarbeitung bewährt." Auffällig ist in jedem Fall, dass sehr wahrscheinlich auftretende Fälle von Eingangsdaten effizienter codiert werden als andere. Wahrscheinliche Eingangswerte liegen in erster Instanz im Bereich [0..15] (ungleich ZERO_MARK), ferner im Bereich von 16 bis ca. 40 (wenige aufeinanderfolgende ZERO_MARKs). Schließlich tritt der Fall 255 noch recht häufig auf, nämlich dann, wenn die Kompressionsstärke so hoch eingestellt ist, dass in der Quantisierung die hochfrequenten Blöcke komplett mit ZERO_MARKs gekennzeichnet werden.

Der Datenausgang von huffman ist 8 Bit breit, schreibt aber nicht zu jedem Takt ein neues Datum. Das liegt daran, dass für einige Argumente weniger als 8 Bit ausgegeben werden, so dass nicht immer sofort 8 Bit zum Schreiben vorliegen.

Schnittstelle:



CLK	Globale Clock.
RESET	Reset-Signal, getrieben von user.
ENABLE	Enable-Signal, getrieben von user.
STREAM_READ	Eingangsdaten, getrieben von zle.
STREAM_STALL	Stream-Stall-Signale für Lese- und Schreibstrom, getrieben von zle bzw. write_flow_control.
LAST_BLOCK	Kontrollsignal, das gesetzt ist, falls der aktuelle Block der letzte zu bearbeitende Block ist; getrieben von zle.
SOURCE_FINISHED	Kontrollsignal, das gesetzt ist, sobald am Eingang das letzte gültige Datum anliegt; getrieben von zle.
STREAM_WRITE	Datenausgang.
STREAM_ENABLE	Enables für Lese- und Schreibstrom.

BLOCK_REALLY_FINISHED	Kontrollsignal, das gesetzt ist, wenn das letzte gültige Datum des aktuellen Blockes am Ausgang liegt.
FINISHED	Kontrollsignal, das gesetzt ist, wenn das letzte gültige Datum des letzten Blockes am Ausgang liegt.

Implementierung:

Wir wollen uns das Verhalten dieses Moduls anhand eines kurzen Beispiels veranschaulichen.

Bild 5.20 zeigt einen Auszug aus der verwendeten Huffman-Lookup-Tabelle. Angenommen, die vom Modul gelesenen Worte wären $x_0 = 6$, $x_1 = 0$, $x_2 = 2$, $x_3 = 9$. huffman würde darauf antworten mit den binären Worten $y_0 = 10111001$ und danach $y_1 = 01110100$, wie man auch Bild 5.21 entnehmen kann. Dabei werden die einzelnen Worte f(x) von rechts nach links aneinandergereiht und in 8-Bit-Gruppen geschrieben.

x	f(x)
0	010010111
1	1100110
2	000111
3	10110
4	0000
5	1011
6	001
7	1111
8	010
9	100
⋮	⋮

Bild 5.20:
Auszug aus der Huffman-Tabelle. Hier ist x dezimal, f(x) binär dargestellt.

Wie die hier verwendete Hardware-Realisierung funktioniert, soll Bild 5.22 illustrieren. Das Register BUFFER dient der Akkumulierung der einzelnen Huffman-Werte zu 8-Bit-Gruppen. Eine Pufferlänge von 36 Bit hat sich als sinnvoll herausgestellt, um größere Mengen an Stalls zu vermeiden. Da in Verilog keine variablen Zugriffe der Art

```
BUFFER[i:j]     <=   f(x)
```

möglich sind, werden solche Zuweisungen mittels Veroderungen und Shifts gelöst:

```
BUFFER          <=   BUFFER | (f(x) << k)
```

wobei k von der nächsten, noch freien Bitposition im Puffer abhängt. Im Bild 5.22 wird die nächste freie Bitposition durch den kleinen senkrechten Pfeil gekennzeichnet. Im Modul-Quelltext entspricht jene Bitposition dem Register FREE_BIT.

Den Puffer kann man von links nach rechts oder von rechts nach links füllen; in dieser Implementierung wurde aus Konsistenzgründen die erste Variante gewählt. Das hat aber zur Folge, dass die einzelnen Huffman-Werte $f(x_i)$ gespiegelt akkumuliert und die gebildeten 8er-Bitgruppen gespiegelt ausgegeben werden müssen. Ersteres wird erreicht, indem die Huffman-Werte bereits gespiegelt in der Tabelle gespeichert werden; die gespiegelte Ausgabe hingegen geschieht über eine entsprechende ständige Zuweisung der oberen 8 Puffer-Bits an STREAM_WRITE.

Ist der Puffer zu voll, darf muss ein Lesestop erfolgen, ist er zu leer, ein Schreibstop. Daher ist huffman eines der beiden eigenwilligen Module, die selbst bestimmen, wann sie lesen und wann sie schreiben möchten. Die beiden steuernden Register, die zu diesem Zweck in die lokale Flusskontrolle mit eingehen, heißen READ_SWITCH und WRITE_SWITCH.

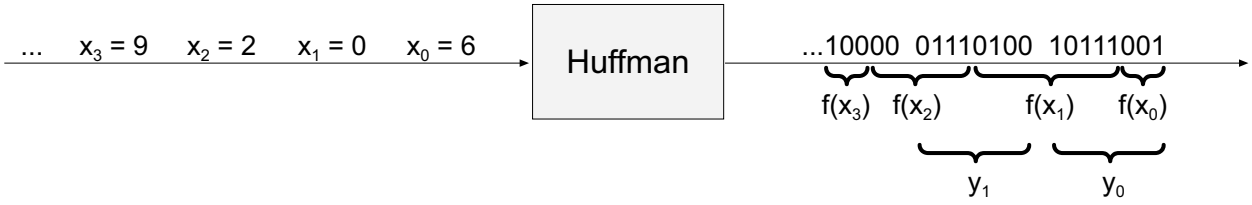


Bild 5.21:
Die Huffman-Codierung als Black-Box. Argumente x_i, Ausgabeworte y_i.

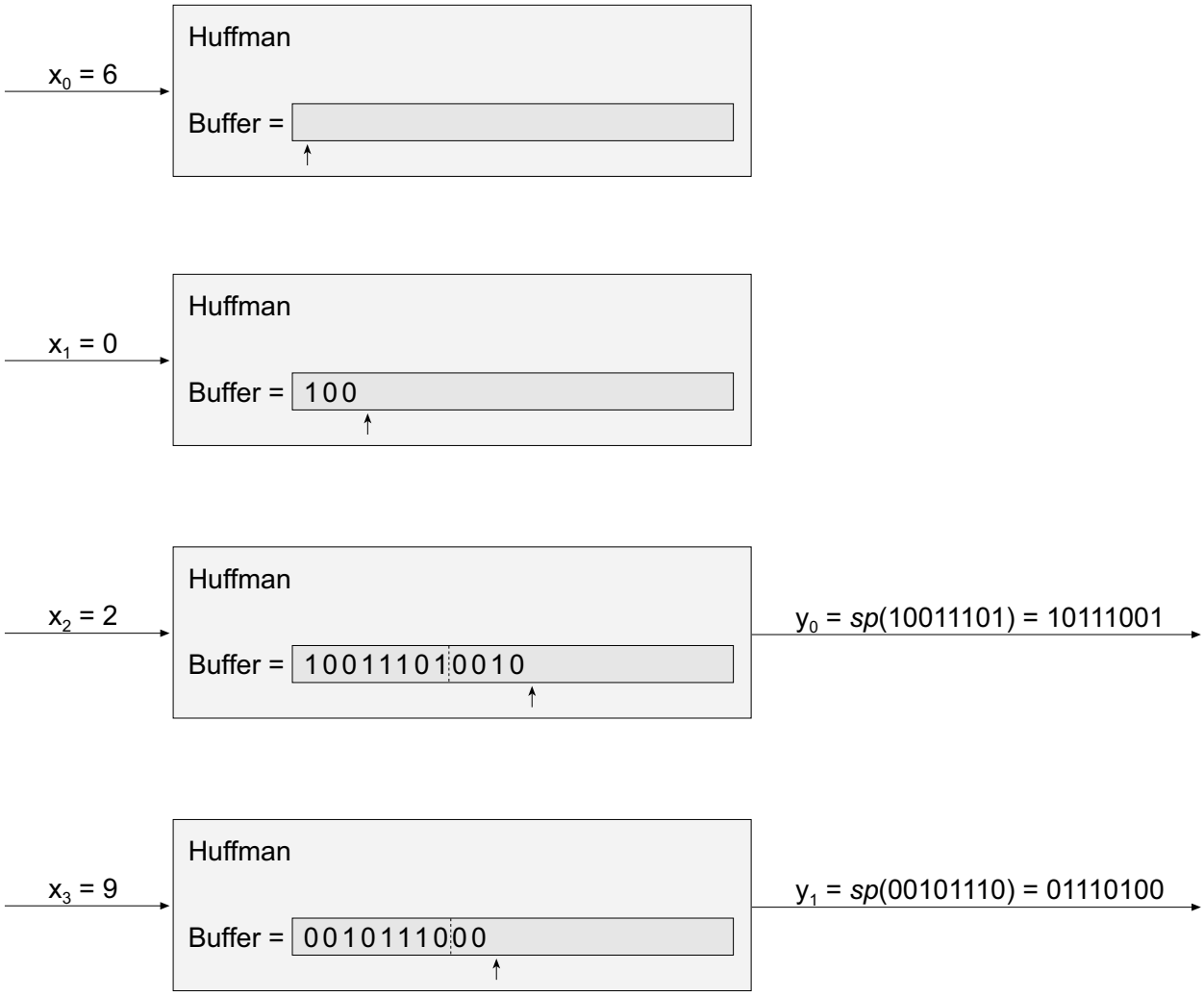


Bild 5.22:
Pufferung und Akkumulierung der Huffman-Werte zu 8-Bit-Worten, die gespiegelt ausgegeben werden. Argumente x_i, Ausgabeworte y_i.

huffman enthält eine Instanz des Moduls huffman_lookup, welches die Huffman-Tabelle speichert und lesende Zugriffe darauf erlaubt.

5.5.7 huffman_lookup

Funktionsbeschreibung:

Hier werden die Huffman-Werte für das Modul huffman gespeichert und zur Verfügung gestellt (Achtung: verglichen mit der Software-Version des Algorithmus werden die Werte hier gespiegelt in der Lookup-Tabelle gespeichert; siehe Erklärungen in Abschnitt 5.5.6, huffman). Die anliegenden 9-Bit-Adressen veranlassen huffman_lookup, am Ausgang OUT_DATA den entsprechenden Huffman-Wert zur nächsten positiven Taktflanke anzulegen. Um eine einfache Adressierung zu ermöglichen, wird jeder Huffman-Wert in der Breite 18 Bit dargestellt; dabei stehen die Huffman-Werte jeweils in den most significant bits, während die übrigen Bitstellen werden mit Nullen aufgefüllt werden. Um die Huffman-Werte extrahieren zu können, ist daher für jeden Wert die zusätzliche Angabe der Bitlänge erforderlich. Jene Bitlängen stehen als 5-Bit-Zahl am Ausgang OUT_SIZE zur Verfügung.

Schnittstelle:

CLK	Globale Clock.
ENABLE	Enable, getrieben von huffman.
IN_DATA	Zu codierendes Datenwort, getrieben von huffman.
OUT_DATA	Huffman-Wert zu dem Argument IN_DATA, least significant bits mit Nullen aufgefüllt.
OUT_SIZE	Bitlänge des Huffman-Wertes OUT_DATA.

Implementierung:

Für jeden der 256 Huffman-Werte existiert eine 23-Bit-Zahl {OUT_DATA[17:0], OUT_SIZE[4:0]}. Diese Zahlen könnten in ROMs (Read Only Memory) gespeichert werden, da sie während der gesamten Berechnung konstant bleiben und lediglich abgefragt werden. ROMs würden aber eine zu krause Verdrahtung verursachen, deswegen werden hier Block-RAMs für die Speicherung verwendet, obwohl auf die Daten nicht schreibend zugegriffen wird. Das kleinste Block-RAM-Primitiv für den Xilinx Virtex FPGA, „RAMB4_Sn“, kann 4096 Bit speichern. Diese Anwendung verwendet 3 Instanzen des Block-RAMs „RAMB4_S8“ (Adressen 9 Bit, Datenworte 8 Bit): bram1, bram2 und bram3. bram1 speichert {OUT_DATA[17:10]}, bram2 speichert {OUT_DATA[9:2]}, bram3 speichert {OUT_DATA[1:0], OUT_SIZE[4:0], 1'b0}. (Das LSB in bram3 bleibt ungenutzt.) Um schnell und unkompliziert arbeiten zu können, wird hier recht verschwenderisch mit Ressourcen umgegangen, da von jedem der 3 Block-RAMs tatsächlich nur die Hälfte des verwendbaren Speichers genutzt wird.

Die Eingänge Reset und Write-Enable der Block-RAM-Instanzen werden konstant auf 0 gehalten, da diese Funktionen hier keine sinnvolle Anwendung finden.

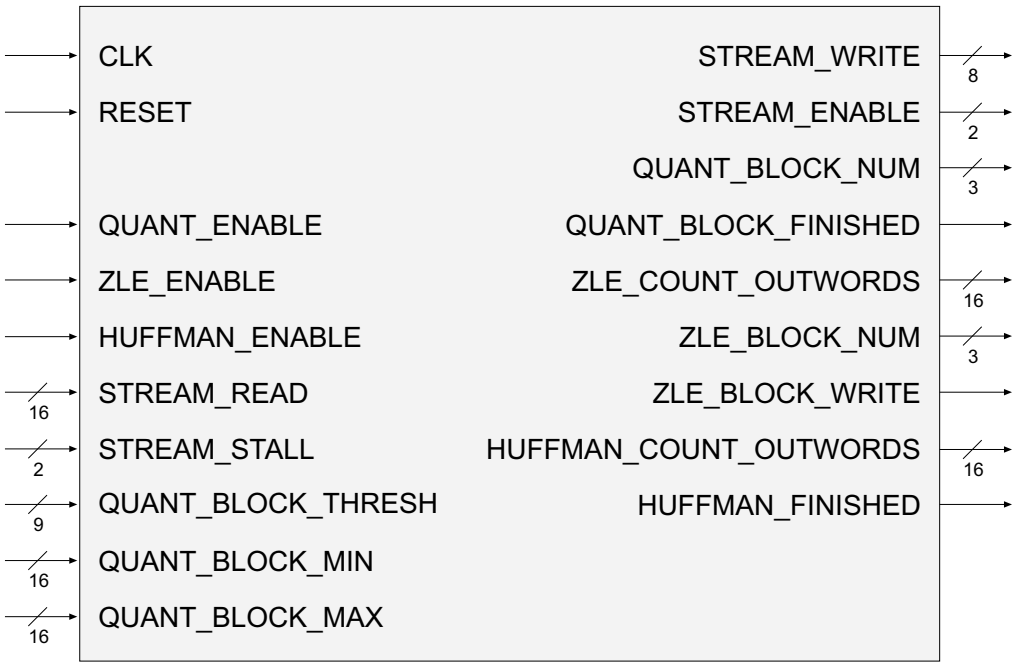
Für jeden Block-RAM findet man im Quelltext des Moduls huffman_lookup zwei Initialisierungsphrasen; die eine in der Form „synthesis xc_props“, die andere per „defparam“-Anweisung. Beide sind notwendig, um die gewünschten Ergebnisse nicht nur im tatsächlichen Betrieb (synthesis xc_props), sondern auch in der Simulation (defparam) zu liefern.

5.5.8 qzh

Funktionsbeschreibung:

qzh fasst die Module quantization, zle und huffman zu einer Verarbeitungspipeline zusammen.

Schnittstelle:



CLK	Globale Clock.
RESET	Reset-Signal für alle Untermodule, getrieben von user.
QUANT_ENABLE	Enable für quantization, getrieben von user.
ZLE_ENABLE	Enable für zle, getrieben von user.
HUFFMAN_ENABLE	Enable für huffman, getrieben von user.
STREAM_READ	Dateneingang, getrieben von read_flow_control.
STREAM_STALL	Stall-Signale für Lese- und Schreibstrom, getrieben von read_flow_control bzw. write_flow_control.
QUANT_BLOCK_THRESH	Blockgrenzwert für den aktuell von quantization bearbeiteten Block.

QUANT_BLOCK_MIN	Blockminimum für den aktuell von quantization bearbeiteten Block.
QUANT_BLOCK_MAX	Blockmaximum für den aktuell von quantization bearbeiteten Block.
STREAM_WRITE	Datenausgang.
STREAM_ENABLE	Enable für Lese- und Schreibstrom.
QUANT_BLOCK_NUM	Nummer des von quantization im Moment bearbeiteten Blocks.
QUANT_BLOCK_FINISHED	Kontrollsignal, das gesetzt ist, sobald das letzte gültige Datum des aktuellen Blocks am Datenausgang von quantization anliegt (wird benötigt von qzh_controller).
ZLE_COUNT_OUTWORDS	Aktuelle Anzahl der von zle geschriebenen Datenworte.
ZLE_BLOCK_NUM	Nummer des im Moment von zle bearbeiteten Blocks.
ZLE_BLOCK_WRITE	Kontrollsignal, das gesetzt ist, falls jetzt ein schreibender Zugriff auf das Speichermodul der ZLE-Blockgrößen (ZLE_sizes) erfolgen soll.
HUFFMAN_COUNT_OUTWORDS	Aktuelle Anzahl der von huffman geschriebenen Datenworte.
HUFFMAN_FINISHED	Kontrollsignal, das gesetzt ist, sobald das insgesamt letzte gültige Datum am Datenausgang von huffman anliegt.

Implementierung:

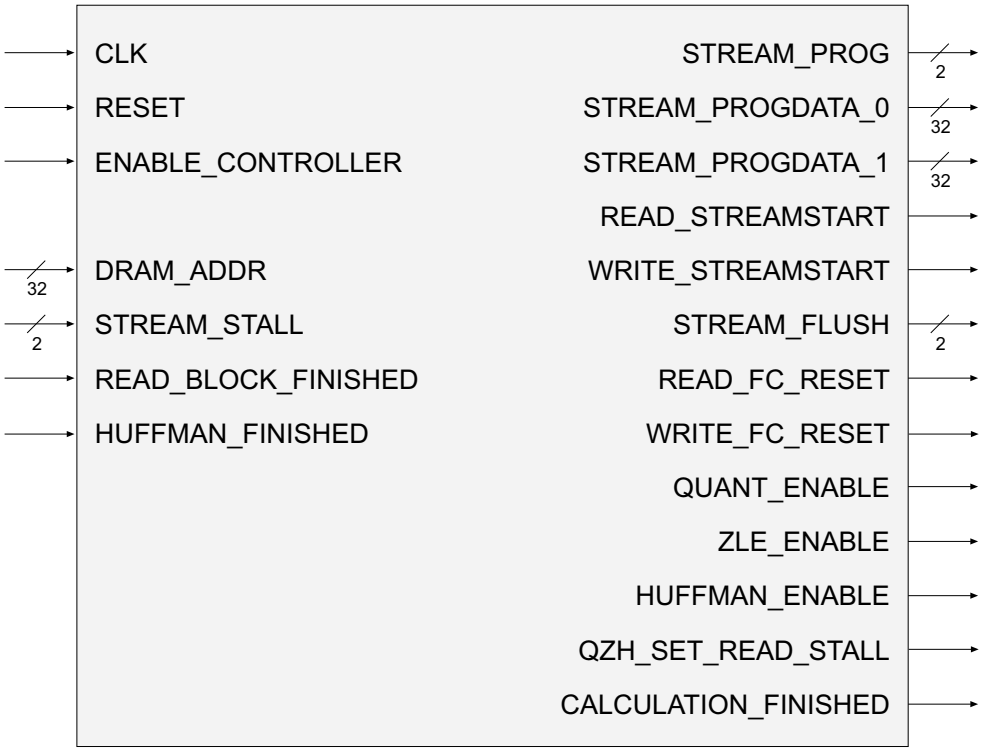
Die Module quantization, zle und huffman werden hier instanziert und verdrahtet; die Verdrahtung ist selbsterklärend und muss daher nicht weiter erläutert werden.

5.5.9 qzh_controller

Funktionsbeschreibung:

qzh_controller dient der Stream-Programmierung für die Datenverarbeitung in der QZH-Pipeline und der Aktivierung der Pipeline-Berechnungsmodule. Die Stream-Kontrolle (Enables, Stalls) fällt nicht in den qzh_controller-Aufgabenbereich, da quantization, zle und huffman dies selbständig erledigen. Trotzdem benötigt qzh_controller die Stall-Signale, um die Gültigkeit des letzten Ausgabewortes zu garantieren und das Leeren des Schreibstroms korrekt zu bewältigen.

Schnittstelle:



CLK	Globale Clock.
RESET	Reset-Signal, getrieben von user.
ENABLE_CONTROLLER	Enable-Signal, getrieben von user.
DRAM_ADDR	Zieladresse für Ausgabedaten im DRAM, getrieben von user.
STREAM_STALL	Stallsignale für Lese- und Schreibstrom, getrieben von read_flow_control bzw. write_flow_control.
READ_BLOCK_FINISHED	Kontrollsignal, das gesetzt ist, falls das letzte Datum des aktuellen Blocks gelesen wurde. Getrieben von quantization.
HUFFMAN_FINISHED	Kontrollsignal, das gesetzt ist, sobald das insgesamt letzte Datum am Ausgang von huffman anliegt. Das dort anliegende Datum ist nur dann gültig, wenn gerade kein Schreib-Stall auftritt. Getrieben von huffman.

STREAM_PROG	Programmier-/Datenmodus-Schalter für Lese- und Schreibstrom.
STREAM_PROGDATA_0	Programmier- und Datenleitung für den Lesestrom.
STREAM_PROGDATA_1	Programmier- und Datenleitung für den Schreibstrom.
READ_STREAMSTART	Startsignal für den Lesestrom.
WRITE_STREAMSTART	Startsignal für den Schreibstrom.
STREAM_FLUSH	Flush-Signale für die MARC-Ströme.
READ_FC_RESET	Reset für read_flow_control.
WRITE_FC_RESET	Reset für write_flow_control.
QUANT_ENABLE	Enable für quantization.
ZLE_ENABLE	Enable für zle.
HUFFMAN_ENABLE	Enable für huffman.
QZH_SET_READ_STALL	Kontrollsignal für die künstliche Erzeugung eines Readstalls, um die Datenverarbeitung während der Lesestrom-Neuprogrammierungen kurzzeitig anzuhalten.
CALCULATION_FINISHED	Kontrollsignal für user: QZH-Berechnung ist beendet.

Implementierung:

Bild 5.23 stellt den Zustandsautomaten dieses Controllers dar. Nach dem einmaligen Programmieren des Schreibstroms wird Block für Block der Lesestrom neu programmiert, da die einzelnen Datenblöcke nach der Wavelet-Transformation nicht zusammenhängend im Speicher liegen.

Die QZH-Berechnungsmodule (quantization, zle, huffman) werden nach dem Programmieren des Schreibstroms gestartet. Damit während der Lesestrom-Neuprogrammierungen keine ungültigen Daten geschrieben werden, signalisiert qzh_controller über den Kanal QZH_SET_READ_STALL, dass die momentane Berechnung unterbrochen werden soll.

Nach der Abarbeitung der Datenblöcke muss im Zustand STATE_WAIT_FOR_HUFFMAN noch gewartet werden, bis die QZH-Pipeline geleert ist. Die Ströme werden angehalten, der Schreibstrom wird geleert, und qzh_controller geht in den Endzustand STATE_SHUTDOWN über, in dem es dem übergeordneten Modul user mitteilt, dass die QZH-Berechnung beendet ist.

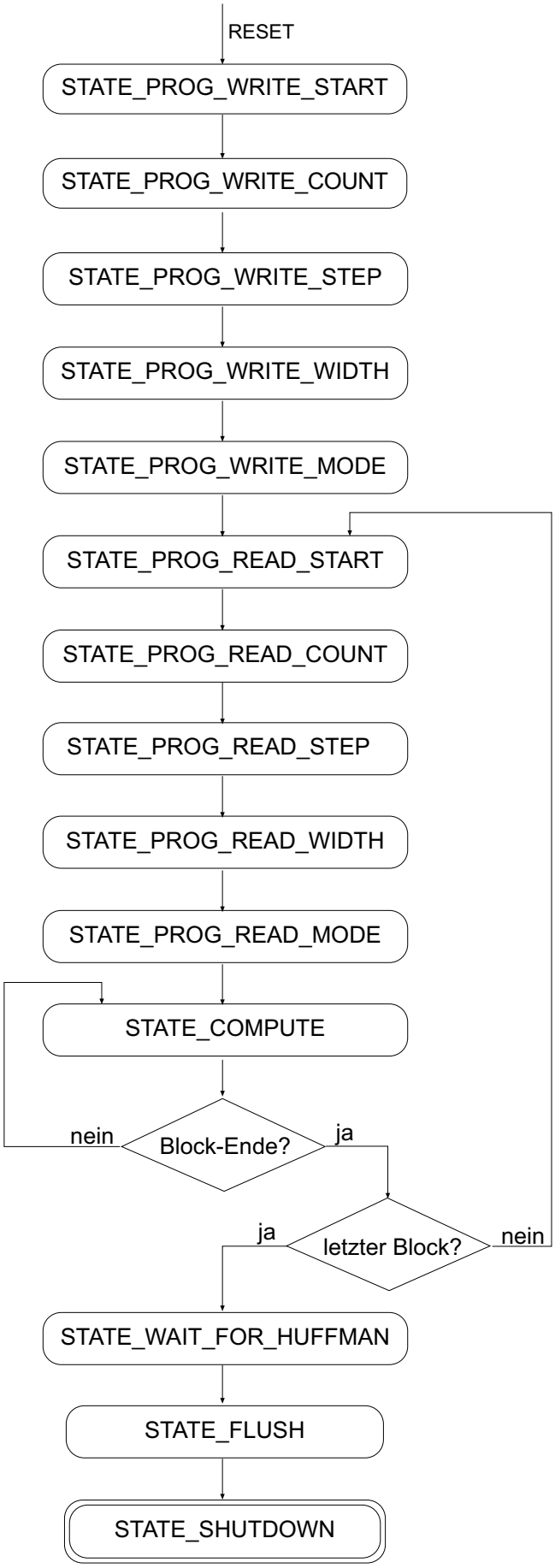


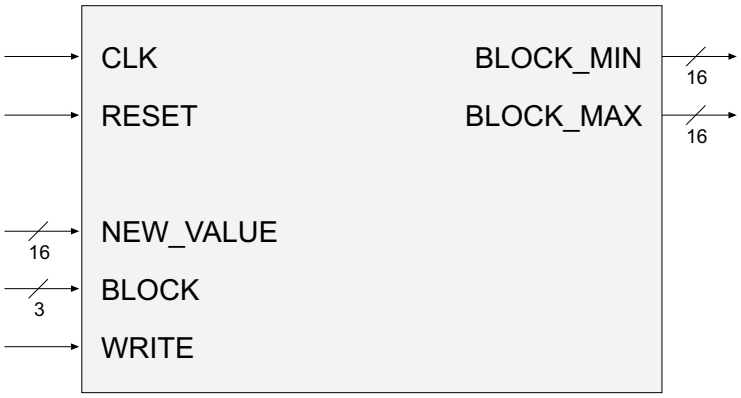
Bild 5.23:
Zustandsautomat des Moduls qzh_controller.

5.5.10 min_max

Funktionsbeschreibung:

Dies ist ein Modul zur Speicherung, Aktualisierung und Ausgabe der Blockminima und -maxima.

Schnittstelle:



CLK	Globale Clock.
RESET	Reset-Signal, getrieben von user.
NEW_VALUE	Potentiell neuer Extremwert, getrieben von wavelet_16_2_hl.
BLOCK	Blocknummer des aktuellen bearbeiteten Blocks, getrieben von wavelet_controller, qzh oder result_params_mux, je nach aktueller Bearbeitungsphase.
WRITE	Wenn dieses Signal gesetzt ist, wird NEW_VALUE für den Block BLOCK als potentiell neuer Extremwert interpretiert. Getrieben von wavelet_16_2_hl.
BLOCK_MIN	Bisheriges Block-Minimum des Blockes BLOCK.
BLOCK_MAX	Bisheriges Block-Maximum des Blockes BLOCK.

Implementierung:

Die Register BLOCK_MIN_0 bis BLOCK_MIN_6 enthalten die bisherigen Minima der Blöcke W0 bis W6. Jeweils eins von ihnen - die Auswahl erfolgt durch das Signal BLOCK - wird an den Ausgang BLOCK_MIN angeschlossen. Im Falle eines schreibenden Zugriffs (WRITE = 1) wird dem gewählten Blockminimum zur nächsten steigenden Taktflanke das neue Minimum zugewiesen, welches schlicht die kleinere der beiden Zahlen BLOCK_MIN und NEW_VALUE ist. Analoges gilt für die Block-Maxima.

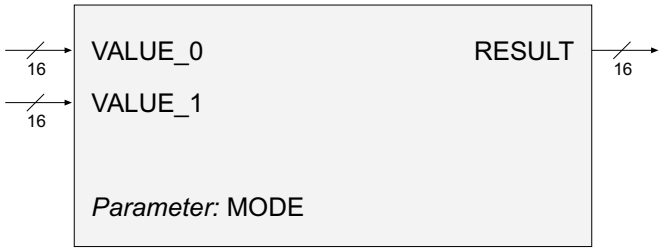
Achtgeben muss man beim Vergleichen der Zahlen, da hier auch wieder negative Werte vorkommen können. Die Vergleiche werden von dem Modul compare_16b_neg erschlagen, das je nach Instanzierungsparameter entweder das Maximum oder das Minimum zweier Zahlen berechnen kann.

5.5.11 compare_16b_neg

Funktionsbeschreibung:

compare_16b_neg vergleicht zwei vorzeichenbehaftete 16-Bit-Zahlen und leitet die größere (für MODE = 1) bzw. kleinere Zahl (für MODE = 0) direkt an den Ausgang weiter.

Schnittstelle:



MODE	Konfiguriert das Modul bei der Instanzierung auf Maximumsberechnung (MODE = 1) bzw. Minimumsberechnung (MODE = 0). Der Default-Wert ist 0.
VALUE_0	Erstes Datum.
VALUE_1	Zweites Datum.
RESULT	Größerer (MODE = 1) bzw. kleinerer Wert (MODE = 0) der beiden angelegten Argumente.

Implementierung:

Zur Fallunterscheidung wird der 1-Bit-Wire IS_BIGGER definiert, der den Wert 1 genau dann annimmt, wenn

```
VALUE_0 > VALUE_1
```

gilt (arithmetischer Vergleich, d. h. vorzeichenbehaftet).

Haben VALUE_0 und VALUE_1 das gleiche Vorzeichen, d. h. es gilt

```
~(VALUE_0[15] ^ VALUE_1[15]),
```

so ist VALUE_0 genau dann größer als VALUE_1, wenn VALUE_0[14:0] > VALUE_1[14:0] (Reduktion des Problems auf den Vergleich positiver Zahlen). Anderenfalls ist das Ergebnis bereits durch die Vorzeichen von VALUE_0 und VALUE_1 vorbestimmt.

Über die trickreiche Zuweisung

```
assign RESULT = (MODE ^ IS_BIGGER)
                    ? VALUE_1
                    : VALUE_0;
```

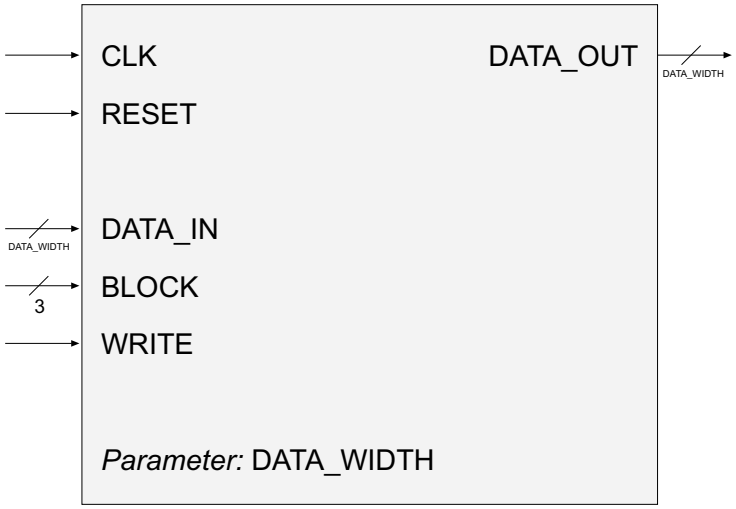
erhält RESULT schließlich genau den gewünschten Wert, wie sich leicht nachvollziehen lässt.

5.5.12 parameter_storage

Funktionsbeschreibung:

Dies ist ein Modul zum Speichern und Ausgeben von acht 16-Bit-Zahlen; es wird für die ZLE-Werte und die Huffman-Bytelänge benutzt.

Schnittstelle:



DATA_WITH	Legt die Bitbreite der zu speichernden Daten bei der Instanzierung fest. Der Default-Wert ist 16 Bit.
CLK	Globale Clock.
RESET	Reset-Signal, getrieben von user.
DATA_IN	Zu speicherndes Datum, das bei WRITE = 1 zur nächsten steigenden Taktflanke in den durch BLOCK angegebenen Speicherplatz geschrieben wird, getrieben von zle, huffman oder user, je nach Instanz und Bearbeitungsphase.
BLOCK	Gibt das Datum an, auf das zugegriffen werden soll (gültige Werte: 0 bis 7), getrieben von zle, huffman, result_params_mux oder user, je nach Instanz und Bearbeitungsphase.
WRITE	Signal, das gesetzt ist, wenn ein schreibender Zugriff auf die gespeicherten Werte erfolgen soll, getrieben von zle, huffman oder user, je nach Instanz und Bearbeitungsphase.
DATA_OUT	Gibt das gespeicherte Datum der Nummer BLOCK aus.

Implementierung:

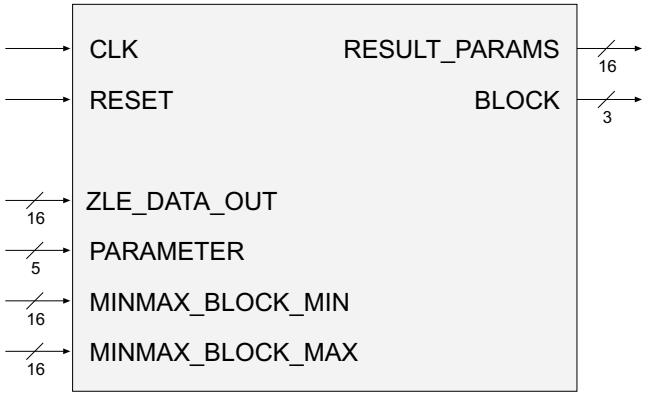
Die Ausgabe erfolgt über eine gemultiplexte, ständige Zuweisung an DATA_OUT. Das Speichern neuer Werte geschieht in einem getakteten always-Block.

5.5.13 result_params_mux

Funktionsbeschreibung:

Dieses Modul dient der Auswahl und Ausgabe der während der Komprimierung gespeicherten Parameter: 7 Blockminima und -maxima, 7 ZLE-Blockgrößen sowie der Bytelänge des von huffman produzierten Bitstroms.

Schnittstelle:



CLK	Globale Clock.
RESET	Reset-Signal, getrieben von user.
ZLE_DATA_OUT	ZLE-Größe des durch PARAMETER gewählten Blockes bzw. Huffman-Bytelänge (die Huffman-Bytelänge wird in dem 8. Speicherplatz des Speichermoduls für die ZLE-Größen gespeichert), getrieben von parameter_storage, Instanz ZLE_sizes.
PARAMETER	Gibt an, welcher Parameter ausgegeben werden soll. Codierung: 0, 1, 2, 3, 4, 5, 6: Blockminima 0..6 8, 9, 10, 11, 12, 13, 14: Blockmaxima 0..6 16, 17, 18, 19, 20, 21, 22: ZLE-Größen 0..6 23: Huffman-Bytelänge. Getrieben wird dieses Signal von user.
MINMAX_BLOCK_MIN	Minimum des durch PARAMETER gewählten Blockes, getrieben von min_max.
MINMAX_BLOCK_MAX	Maximum des des durch PARAMETER gewählten Blockes, getrieben von min_max.
RESULT_PARAMS	Dies ist die gewünschte Auswahl aus den 3 Signalen MINMAX_BLOCK_MAX, MINMAX_BLOCK_MIN und ZLE_DATA_OUT.
BLOCK	Gewählter Datenblock (Umcodierung von PARAMETER zur direkten Ansteuerung der Module min_max und parameter_storage).

Implementierung:

PARAMETER wurde so codiert, dass die beiden most significant bits dieses Signals bereits angeben, ob das jeweilige Blockminimum (PARAMETER[4:3] == 2'b00), Blockmaximum (PARAMETER[4:3] == 2'b01) oder die ZLE-Größe bzw. Huffman-Bytelänge (PARAMETER[4:3] == 2'b1x) gewünscht ist. Entsprechend simpel fällt die ständige Zuweisung für RESULT_PARAMS aus.

Nun muss nur noch garantiert werden, dass die beiden Speichermodule min_max und parameter_storage (Instanz ZLE_sizes) auch die korrekten Daten liefern. Dazu wird PARAMETER in das Signal BLOCK umcodiert, das dann die Ausgabe der Speichermodule bestimmt. Genialerweise gilt

```
BLOCK = PARAMETER mod 8,
```

was die Umcodierung von PARAMETER zu BLOCK auf die einfache Zuweisung

```
BLOCK <= PARAMETER[2:0];
```

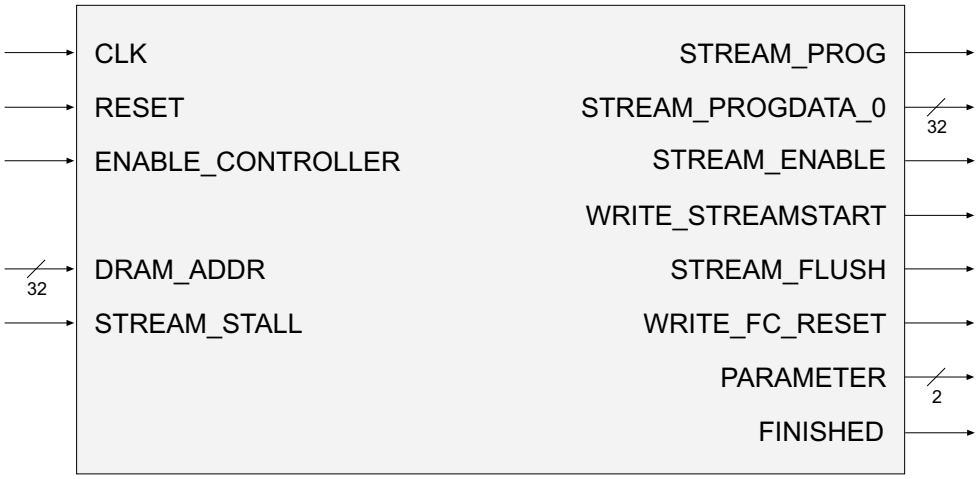
reduziert.

5.5.14 result_controller

Funktionsbeschreibung:

result_controller steuert die Master-Mode-Übertragung der Ergebnisparameter (Blockminima, Blockmaxima, ZLE-Blockgrößen, Huffman-Bytelänge) in den DRAM. Benötigt wird nur ein MARC-Strom, der als Schreibstrom genutzt wird. Die Übertragung findet nach der QZH-Berechnung statt. Man könnte die Übertragung im Slave-Mode programmiertechnisch viel einfacher realisieren als im Master-Mode, doch benötigt jenes Verfahren durch die lesenden RC-Zugriffe eine deutlich höhere Laufzeit.

Schnittstelle:



CLK	Globale Clock.
RESET	Reset-Signal, getrieben von user.
ENABLE_CONTROLLER	Enable-Signal, getrieben von user.
DRAM_ADDR	Zieladresse für Ausgabedaten im DRAM, getrieben von user.

STREAM_STALL	Stallsignal für den Schreibstrom, getrieben von write_flow_control.
STREAM_PROG	Programmier-/Datenmodus-Schalter für den Schreibstrom.
STREAM_PROGDATA	Programmier- und Datenleitung für den Schreibstrom.
STREAM_ENABLE	Enable für den Schreibstrom.
WRITE_STREAMSTART	Startsignal für den Schreibstrom.
STREAM_FLUSH	Flush-Signal für den Schreibstrom.
WRITE_FC_RESET	Reset für write_flow_control.
PARAMETER	Code für den auszugebenden Parameter.
FINISHED	Kontrollsignal für user: Übertragung ist beendet.

Implementierung:

Wird der Controller aktiviert (RESET = 0, ENABLE_CONTROLLER = 1), setzt sogleich die Programmierung des Ausgabedatenstroms ein. DRAM_ADDR dient als Zieladresse der 22 zu schreibenden Ergebnisparameter (7 Blockminima, 7 Blockmaxima, 7 ZLE-Größen, 1 Huffman-Bytelänge). Eigentlich würden 16 Bit als Datenwortbreite für die Übertragung ausreichen, da keiner der 22 Parameter größer als 16 Bit werden kann. Das von der Software verwendete Format zur Speicherung des Bitstroms verlangt jedoch 32 Bit für jeden Parameter. Durch die daher eingesetzte 32-Bit-Übertragung landen die Daten direkt an der gewünschten Speicherposition und bedürfen keiner Überarbeitung.

Nach dem Programmieren des Schreibstroms setzt result_controller den Ausgang PARAMETER für jedes zu übertragende Datum auf den Code für dieses Datum. Die Codes sind so gewählt, dass sie von result_params_mux direkt verstanden werden. result_params_mux sorgt dann für das Anlegen des gewünschten Ausgabedatums an den Datenport des Schreibstroms.

Pro zu schreibendes Datum durchläuft der Automat dieses Moduls die beiden Zustände STATE_PREPARE_WRITE (setzt das Enable für den Schreibstrom zum Schreiben des nächsten Parameters) und STATE_WRITE (bereitet das Schreiben des nächsten Parameters vor, falls kein Schreibstall auftrat).

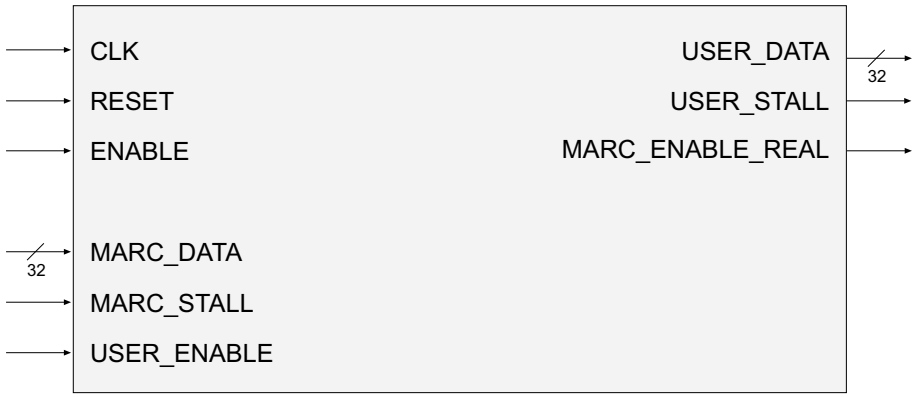
Wurden alle 22 Parameter geschrieben, sorgt der Controller mit dem STREAM_FLUSH-Signal für ein Leeren des Schreibstroms und gibt dem user-Modul über FINISHED = 1 anschließend das Ende der Übertragung bekannt.

5.5.15 read_flow_control

Funktionsbeschreibung:

read_flow_control ist ein Modul zur Flusssteuerung des Lesestroms. Seine Aufgaben bestehen darin, einerseits die Kontrollsignale Enable und Stall des MARC-Lesestroms in entsprechende Signale für die Anwendungsmodule zu übersetzen (siehe auch Abschnitt 5.3) und andererseits eine Verzögerungsstufe zu bilden, um kritische Pfade abzukürzen.

Schnittstelle:



CLK	Globale Clock.
RESET	Reset-Signal, getrieben von wavelet_controller bzw. qzh_controller, je nach Bearbeitungsphase.
ENABLE	Nur für ENABLE = 1 wird das Enable des Lesestroms gesetzt; getrieben von user.
MARC_DATA	Lese-Daten vom MARC-Streamport. Wenn MARC_ENABLE_REAL = 1 und MARC_STALL = 0 gelten, ist das im folgenden Takt an diesem Datenport anliegende Datum gültig und wird gelesen, ansonsten ist es ungültig.
MARC_STALL	Lese-Stallsignal vom MARC-Streamport.
USER_ENABLE	Lese-Enable von der Anwendung, getrieben von wavelet_controller bzw. qzh, je nach Bearbeitungsphase.
USER_DATA	Für die Anwendung zu lesende Daten. Wenn USER_ENABLE = 1 und USER_STALL = 0 gelten, ist das aktuell an diesem Datenport anliegende Datum gültig und muss von der Anwendung gelesen werden; für die drei übrigen Kombinationen von USER_ENABLE und USER_STALL ist das anliegende Datum ungültig.
USER_STALL	Stallsignal für die Anwendung.
MARC_ENABLE_REAL	Enable für den Lesestrom.

Implementierung:

Als Implementierungskonzept wurde auch hier der endliche Automat zugrundegelegt. Je nach Zustand und Stream-Kontrollsignalen werden die an MARC_DATA anliegenden Daten über das Pufferregister BUFFER an den Datenausgang USER_DATA weitergeschoben oder auch nicht.

Zu Beginn befindet sich das Modul im Zustand STATE_START, in dem es auf den „Startschuss“ (d. h. USER_ENABLE = 1) der user-Schaltung wartet.

In den darauf folgenden Zuständen STATE_FILL_PREPARE, STATE_FILL_0 und STATE_FILL_1 wird zunächst der Puffer gefüllt, bevor in den Standard-Bearbeitungszustand STATE_NORMAL übergegangen werden kann. Dort wird zu jedem Takt ein Datum vom MARC-Strom gelesen und entsprechend eines an die Anwendung weitergereicht.

Tritt ein Stall auf der user-Seite auf, werden zwar noch Daten vom MARC-Strom eingelesen, aber es können wegen des Stalls keine an die Anwendung weitergegeben werden. Die überschüssigen Daten werden in den Zuständen STATE_BUFFER_1_TOO_FULL und STATE_BUFFER_2_TOO_FULL in den Pufferregistern XBUFFER und XXBUFFER gehalten. Der zu volle Puffer wird geleert, um die Anwendung dann wieder über STATE_PREPARE_NORMAL in den Normalzustand bringen zu können.

Falls ein Stall auf der MARC-Seite auftritt, wird ein Datum aus dem Puffer an die Anwendung abgegeben, ohne dass der Puffer wieder aufgefüllt wird. Dies wird durch das Übergehen in den Zustand STATE_BUFFER_1_TOO_EMPTY registriert. Es wird daher auf das Einlesen des fehlenden Datums gewartet, bis erneut der Normalzustand erreicht wird.

MARC_ENABLE_REAL ist das tatsächlich an den MARC-Lesestrom übertragene Enable-Signal. Der Zusatz „_REAL“ wurde angefügt, um Verwechslungen mit dem modulinternen Signal MARC_ENABLE zu vermeiden. MARC_ENABLE ist das aus dem aktuellen Modulzustand hergeleitete Enable-Signal, das mit dem globalen Enable verundet wird, um so als MARC_ENABLE_REAL ein immer gültiges Lesestrom-Kontrollsignal zu liefern. Für die korrekte Funktionalität von MARC muss nämlich sichergestellt sein, dass das Enable-Signal nur dann gesetzt ist, wenn der Stream auch wirklich aktiv ist (und nicht gerade programmiert wird).

In Abschnitt 5.3, Bild 5.12, wurde bereits dargestellt, wie sich die Leseflusskontrolle bei einem Stall des MARC-Stroms (Lese-Stall) verhält. In Bild 5.25 wird ein Szenario durchgespielt, in dem die user-Schaltung ihr Enable für einige Zeit auf 0 setzt. Dabei ist auf der rechten Seite stets der Zustand des Automaten zu sehen, während links die Stream-Schnittstelle des Moduls und die aktuellen Pufferinhalte dargestellt sind. Die zusätzlichen Pufferregister XBUFFER und XXBUFFER werden im Normalbetrieb nicht benutzt und nur eingezeichnet, wenn sie relevante Daten enthalten. Als Legende für Bild 2.25 dient das folgende Bild 2.24. In Bild 2.25 umkreiste Datensignale sind gültige Signale, d. h. tatsächlich gelesene bzw. geschriebene Signale.

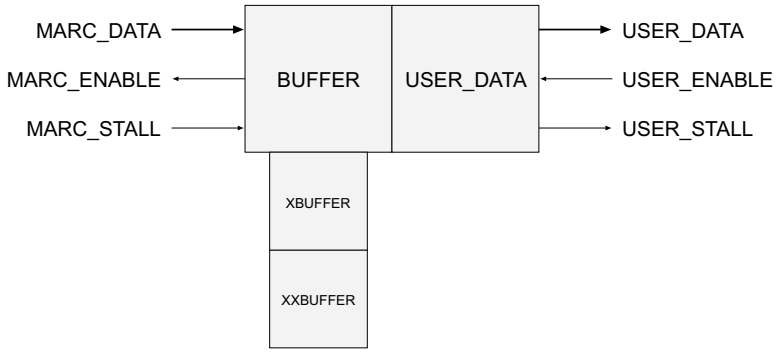
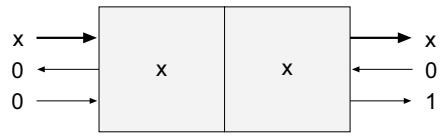
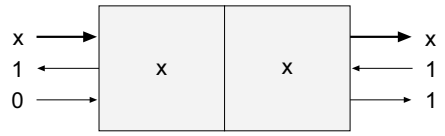


Bild 5.24:
Legende für Bild 2.25. Die Inhalte der Register XBUFFER und XXBUFFER werden nur angezeigt, wenn sie relevante Daten enthalten. Die fett gedruckten Pfeile deuten den Pfad der Bilddaten an.



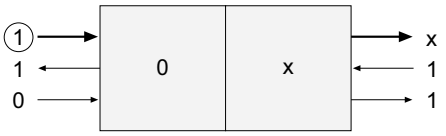
STATE_START

STATE_START

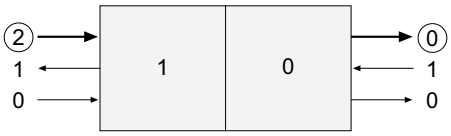


STATE_FILL_PREPARE

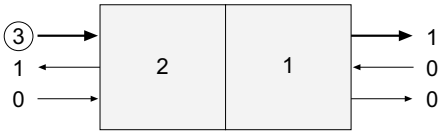
STATE_FILL_0



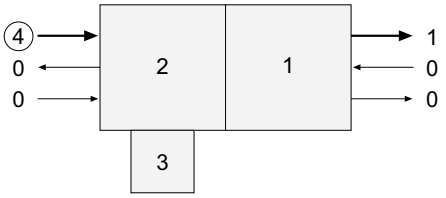
STATE_FILL_1



STATE_NORMAL



STATE_NORMAL



STATE_BUFFER_1_TOO_FULL

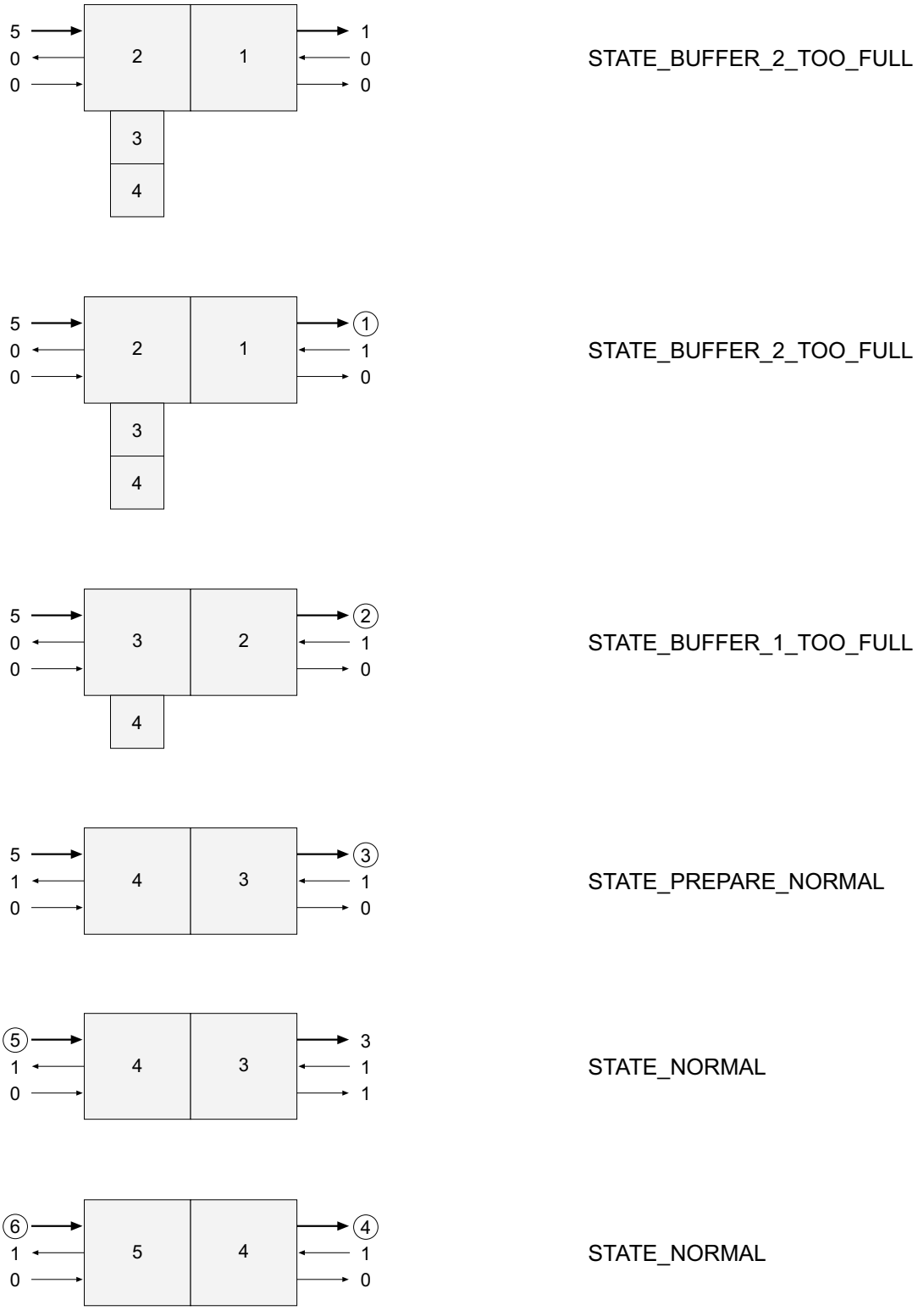


Bild 5.25:
Ein mögliches Szenario für read_flow_control. Die Daten 0, 1, 2, 3, 4 sollen gelesen werden. user setzt nach dem Lesen der 0 das Enable auf 0. Dies hätte einen Pufferüberlauf in der Flusskontrolle zur Folge, wenn nicht die zusätzlichen Puffer XBUFFER und XXBUFFER die überschüssigen Daten auffangen würden.
Legende: siehe Bild 2.24. Umkreiste Datensignale (3) werden momentan gelesen bzw. geschrieben.

5.5.16 write_flow_control

Funktionsbeschreibung:

write_flow_control ist ein Modul zur Flusssteuerung eines Schreibstroms. Da die Schreib-Enable- und Schreib-Stall-Semantiken der Anwendung mit denen von MARC übereinstimmen, muss hier keine Übersetzung der Kontrollsignale zwischen Anwendung und MARC erfolgen (im Gegensatz zu read_flow_control). Die Aufgabe dieses Moduls ist es lediglich, die langen Signalpfade aufzuspalten. So wird keines der angelegten Signale zum nächsten Modul durchgeschleift; stattdessen findet eine Pufferung statt.

Schnittstelle:

CLK	Globale Clock.
RESET	Reset-Signal, getrieben von wavelet_controller, qzh_controller oder result_controller, je nach Bearbeitungsphase.
USER_DATA	Daten, die die Anwendung schreiben möchte, getrieben von wavelet_8_4_l, wavelet_16_2_hl, huffman oder result_params_mux, je nach Bearbeitungsphase.
USER_ENABLE	Schreibstrom-Enable von der Anwendung, getrieben von wavelet_controller, qzh_controller oder result_controller, je nach Bearbeitungsphase.
USER_FLUSH	Flush-Signal der Anwendung.
MARC_STALL	Schreib-Stallsignal vom MARC-Streamport.
MARC_DATA	Vom MARC-Strom zu schreibende Daten.
MARC_ENABLE	Enable für den Schreibstrom.
MARC_FLUSH	Flush-Signal für den Schreibstrom.
USER_STALL	Stallsignal für die Anwendung.

Implementierung:

Die Verzögerung der Daten wird über eine Pufferung im Register BUFFER erreicht. Zur Handhabung der verschiedenen auftretenden Enable- und Stall-Konfigurationen wird auch in diesem Modul wieder auf das bewährte Zustandsautomaten-Konzept gesetzt.

Ähnlich wie in der Leseflusskontrolle read_flow_control ist nach dem Starten im Zustand STATE_STARTING eine Pufferfüllung (STATE_FILL) nötig, um im Normalzustand STATE_NORMAL in jedem Takt ein Datum lesen und auch schreiben zu können.

Im Fall eines MARC-Stalls werden überschüssige (zuviel von user gelesene) Daten im Register XBUFFER gesichert, und der Automat wartet im Zustand STATE_WAIT_FOR_MARC auf das Stall-Ende, was eine Leerung des zusätzlichen Puffers XBUFFER und ein Rückkehren in den Normalzustand ermöglicht.

Bei einem Stall der Anwendung (USER_STALL) wartet der Automat im Zustand STATE_WAIT_FOR_USER auf das Stall-Ende und damit die Rückkehr in den Normalzustand.

Setzt die Anwendung das Flush-Signal, so müssen zunächst die noch im Puffer enthaltenen Daten in den MARC-Strom geschrieben werden (Zustände STATE_FLUSH_MINUS_5 bis STATE_FLUSH_MINUS_1), bis anschließend in STATE_FLUSHING tatsächlich MARC_FLUSH gesetzt und so der MARC-Strom geleert werden kann.

Bild 2.27 gibt ein Szenario zum Schreiben von sechs Datenworten (0, ..., 5) an. Zunächst wird der Puffer gefüllt; nach dem Schreiben des ersten Datums tritt - ganz unerwartet - ein Stall im MARC-Strom auf, und das zuviel gelesene Datum „3“ wird im XBUFFER gehalten. Nach einer Pufferleerung setzt plötzlich die Anwendung ihr Enable auf 0, weshalb der Schreibstrom angehalten werden muss. Schließlich wird noch eine typische Flush-Sequenz präsentiert. Bild 2.26 dient als Legende für die in Bild 2.27 angegebenen Signale.

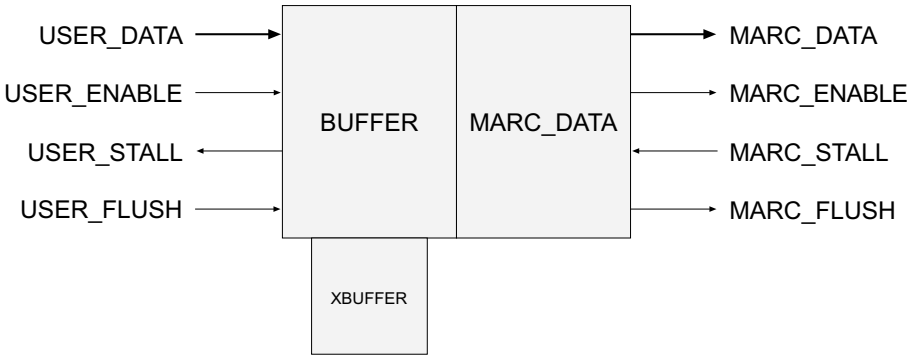
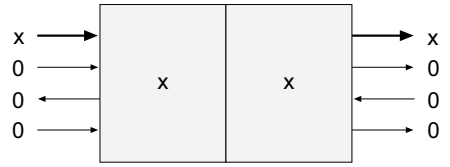
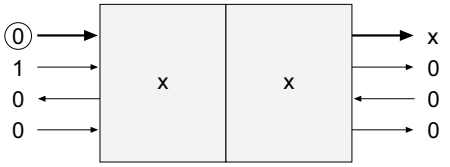


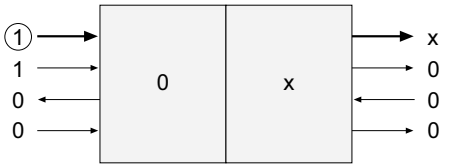
Bild 5.26:
Legende für Bild 2.27. Der Inhalt des Registers XBUFFER wird nur angezeigt, wenn es relevante Daten enthält. Die fett gedruckten Pfeile deuten den Pfad der Bilddaten an.



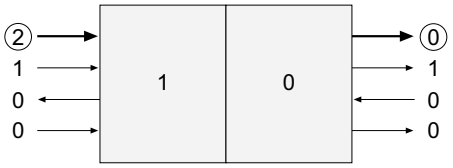
STATE_STARTING



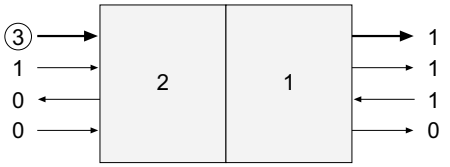
STATE_STARTING



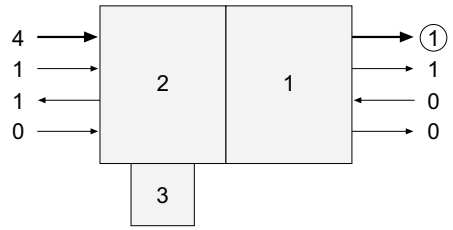
STATE_FILL



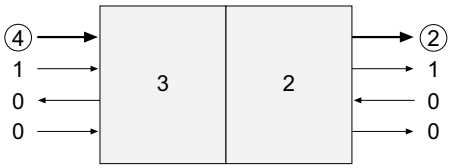
STATE_NORMAL



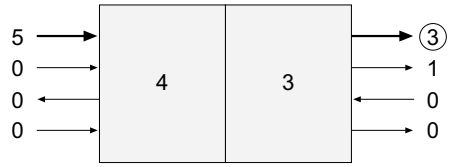
STATE_NORMAL



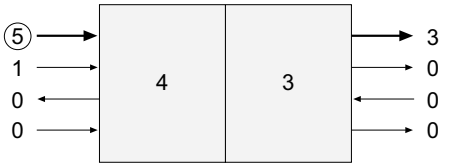
STATE_WAIT_FOR_MARC



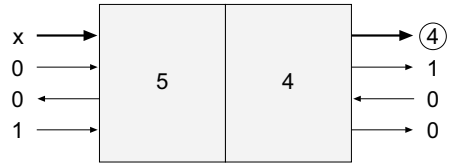
STATE_NORMAL



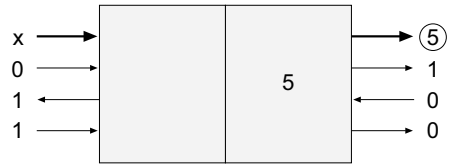
STATE_NORMAL



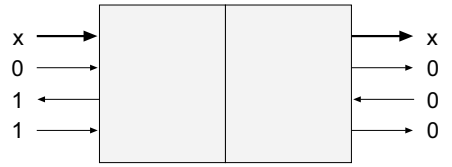
STATE_WAIT_FOR_USER



STATE_NORMAL



STATE_FLUSH_MINUS_3



STATE_FLUSH_MINUS_2

STATE_FLUSH_MINUS_1

STATE_FLUSHING

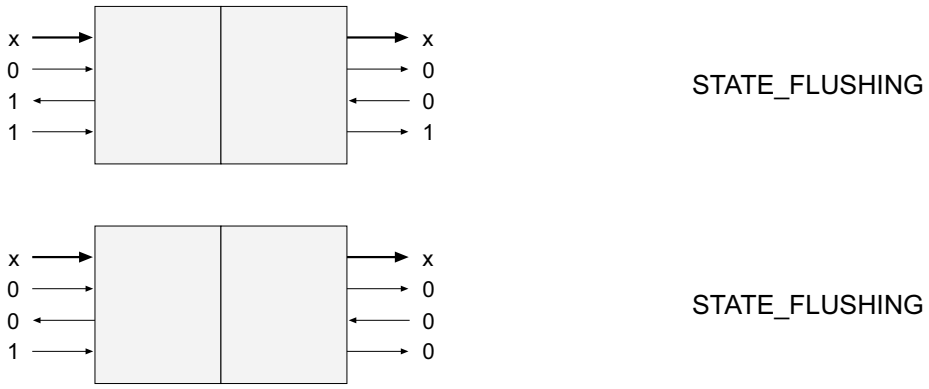


Bild 5.27:
Ein mögliches Szenario für write_flow_control. Die Daten 0, 1, 2, 3, 4, 5 sollen geschrieben werden. Nach dem Schreiben der 0 meldet MARC einen Stall, und so muss das Register XBUFFER das von user zuviel gelesene Datum aufnehmen. Als zweites Stör-Beispiel setzt die Anwendung USER_ENABLE auf 0, weswegen der Schreibstrom kurzzeitig angehalten werden muss. Schließlich wird noch eine typische Flush-Sequenz dargestellt. Legende: siehe Bild 2.26. Umkreiste Datensignale (3) werden momentan gelesen bzw. geschrieben.

5.5.17 user

Funktionsbeschreibung:

Als Top-Level-Modul ist user mit seinem globalen Zustandsautomaten für den Gesamtablauf der Anwendung verantwortlich. Es bildet über die Flusskontroll-Module read_flow_control und write_flow_control die Schnittstelle zwischen MARC und der Anwendung. Während die Master-Mode-Zugriffe von Untermodulen gesteuert werden, fallen Zugriffe im Slave-Mode komplett in den Verantwortungsbereich von user. Durch seine Top-Level-Funktionalität ist user das Modul mit den meisten Modul-Instanzen und Verdrahtungen.

Schnittstelle:

CLK	Globale Clock.
RESET	Chip-Reset.
ADDRESSED	Wird gesetzt, um die RC im Slave-Mode anzusprechen.
WRITE	Wird bei Schreibzugriffen auf die RC im Slave-Mode gesetzt.
DATAIN	Dateneingang im Slave-Mode.
ADDRESS	Adresseingang im Slave-Mode.
STREAM_READ	Leseports für die 3 verwendeten Ströme im Master-Mode (32 Bit pro Port).
STREAM_STALL	Stall-Signale für die 3 verwendeten Ströme im Master-Mode.
DATAOUT	Datenausgang im Slave-Mode.
STREAM_WRITE_PROG	Programmier- und Datenausgänge für die 3 verwendeten Ströme im Master-Mode (32 Bit pro Port).
STREAM_ENABLE	Enables für die 3 verwendeten Ströme im Master-Mode.
STREAM_FLUSH	Flush-Signale für die 3 verwendeten Ströme im Master-Mode. Ein Flush wird gesetzt, um nach der Beendigung der Verarbeitung die noch im MARC-Puffer befindlichen Daten zu schreiben.

STREAM_PROG	Kontrollsignale für die 3 verwendeten Ströme im Master-Mode. Ein gesetztes STREAM_PROG bedeutet für den angesprochenen Strom, dass er momentan programmiert wird und keine Daten verarbeitet werden. Das zugehörige STREAM_ENABLE darf während der Programmierphasen nicht gesetzt sein.
IRQ	Löst im gesetzten Zustand einen Interrupt in der CPU aus.

Implementierung:

Da in der Schaltung drei Controller für ihre Bearbeitungsabschnitte die Kontrolle über die Datenströme benötigen, treten in user viele Multiplexer auf, die entsprechende Signale je nach gerade aktivem Controller zuordnen. Schauen wir uns als Beispiel den Multiplexer für das Schreibstrom-Resetsignal an:

```
assign WRITE_FC_RESET = (CONTROLLER_SWITCH == `WAVELET_CONTROLLER)
                                    ? WAVELET_WRITE_FC_RESET
                                    : (CONTROLLER_SWITCH == `QZH_CONTROLLER)
                                        ? QZH_WRITE_FC_RESET
                                        : RESULT_WRITE_FC_RESET;
```

Falls wavelet_controller momentan die Kontrolle besitzt, wird sein Kontrollsignal WAVELET_WRITE_FC_RESET an den Reseteingang der Schreibflusskontrolle gelegt; anderenfalls übernehmen entweder qzh_controller oder result_controller die Steuerung.

Etwas komplizierter ist das Multiplexen der verschiedenen Enable-Signale für die Datenströme. Wir werfen als Beispiel einen Blick auf das Enable für Strom Nr. 1 (erster der beiden Schreibströme):

```
assign MODULE_STREAM_ENABLE[1]
    = (CONTROLLER_SWITCH == `WAVELET_CONTROLLER)
        ? (
            (ROW_COLUMN_SWITCH == `COLUMNS)
                ? (WAVE_STREAM_ENABLE[1] & ~WAVE_STREAM_STALL[1]
                    & (LOW_DATA == `LOW_DATA))
                : WAVE_STREAM_ENABLE[1] & ~WAVE_STREAM_STALL[1]
          )
        : (CONTROLLER_SWITCH == `QZH_CONTROLLER)
            ? QZH_STREAM_ENABLE[1]
            : RESULT_STREAM_ENABLE;
```

Das Enable-Signal, welches hier das Ziel der Zuweisung ist, wird nicht direkt an den MARC-Strom weitergegeben, sondern zunächst noch von der Flusskontrolle bearbeitet. Daher wurde es hier „MODULE_STREAM_ENABLE“ anstatt „STREAM_ENABLE“ genannt. QZH_STREAM_ENABLE sowie RESULT_STREAM_ENABLE werden - je nach aktivem Controller - an MODULE_STREAM_ENABLE durchgeleitet. Im Wavelet-Fall muss jedoch eine Fallunterscheidung stattfinden:

In der Spaltenverarbeitung (ROW_COLUMN_SWITCH == `COLUMNS) wechseln sich die beiden Schreibströme mit dem Schreiben ab, je nachdem, ob gerade Low- oder High-Daten geschrieben werden sollen. Daher muss, damit Strom Nr. 1 ein Datum schreiben darf, nicht nur das Wavelet-Modul ein Datum schreiben wollen (WAVE_STREAM_ENABLE[1]) und dürfen (~WAVE_STREAM_STALL[1]), sondern es müssen auch gerade zu schreibende Low-Daten vorliegen: (LOW_DATA == `LOW_DATA). Für Strom Nummer 2 steht an dieser Stelle analog (LOW_DATA == `HIGH_DATA).

Im Zeilenmodus wird nicht abwechselnd, sondern gleichzeitig geschrieben, daher fällt hier die LOW_DATA-Bedingung weg.

Eine weitere interessante Implementierung wird zur Realisierung der schreibenden Zugriffe im Slave-Mode gegeben. Im Slave-Mode werden wichtige Konfigurationsdaten von der CPU an die RC übermittelt: DRAM-Speicheradressen für die Bilddaten, die Blockgrenzwerte für die Quantisierung, im Testmodus die verschiedenen Testparameter, sowie natürlich das Signal zum Starten des aktiven RC-Betriebs. Für jeden dieser zu übermittelnden Parameter wird eine eigene Adresse benötigt. Mehr Adressen bedeuten einen höheren Verarbeitungsaufwand in der RC (größerer Multiplexer). Hier kommt die Anwendung mit einem 4-Bit-Multiplexer aus, obwohl theoretisch mehr als 16 Parameter übermittelt werden können. Dies wird erreicht, indem die Breite der Zugriffe besser ausgenutzt wird. Eventuelle Testparameter werden gleichzeitig mit den Blockgrenzwerten übertragen: die oberen 16 Bit (eigentlich nur 9 Bit) werden für die Blockgrenzwerte verwendet, die unteren für die Testparameter. Hier ein Beispiel aus dem Quelltext (NUM_INWORDS ist dabei ein Testparameter, BTHRESH_DATA_IN ist der erste übertragene Blockgrenzwert):

```
case (REGNUM)
        ...
        5:    begin
                    NUM_INWORDS             <= {16'h0, DATAIN[15:0]};
                    BTHRESH_WRITE           <= 1;
                    BTHRESH_BLOCK_REG       <= 1;
                    BTHRESH_DATA_IN         <= DATAIN[24:16];
              end
```

Weitere Informationen zur Benutzung der Testmodi findet man in Kapitel 6.

Als letztes soll hier auf den globalen Zustandsautomaten eingegangen werden, der in Bild 5.28 dargestellt ist. Im Startzustand PHASE_WAIT wartet die RC auf die Beendigung der hier stattfindenden Slave-Mode-Zugriffe.

Sobald das Startsignal von der Software gegeben wird, wechselt der Automat nach PHASE_BEGIN, wo die im Bild gezeigten Fallunterscheidungen gemacht werden. Hier sieht man ebenfalls die in der Schaltung integrierten Testmöglichkeiten: in PHASE_TEST_WAVE erfolgt ein einziger Wavelet-Transformations-Durchlauf mit den Parametern (Zeilen- / Spaltenmodus, Anzahl Ein- / Ausgabeworte, Start- / Zieladressen, Zeilenlänge, 8 oder 16 Bit pro Pixel), die zuvor im Slave-Mode übermittelt wurden. Im QZH-Test-Fall werden sämtliche Wavelet-Phasen übersprungen und nur die QZH-Verarbeitung mit anschließendem Transfer der Ergebnisparameter durchgeführt.

Im „normalen Betrieb“ jedoch durchläuft der Zustandsautomat nach PHASE_BEGIN die 6 Wavelet-Phasen. Die ungeraden Wavelet-Phasen beziehen sich dabei auf Zeilendurchläufe, die geraden Phasen auf Spaltendurchläufe. Die Spaltendurchläufe der Phasen 4 und 6 sind jeweils zweigeteilt, da hier für jeden Ausgabestrom nach der Hälfte der zu schreibenden Daten ein neuer Unterblock entsteht (eigene Blockminima / -maxima sowie Speicheradressen). Auf die Darstellung dieser Zweiteilung wurde im Bild 5.28 verzichtet, da es sonst unnötig kompliziert aussehen würde.

Ebenso sei darauf hingewiesen, dass pro Wavelet-Phase 2 reale Zustände im Automaten implementiert wurden. Wir wollen das am Beispiel der Phase PHASE_WAVE_1 nachvollziehen. Es gibt für diese Phase die zwei Zustände PHASE_WAVE_1_PROG und PHASE_WAVE_1_COMPUTE.

In PHASE_WAVE_1_PROG (Programmierphase) wird wavelet_controller mit den zur Durchführung der aktuellen Transformation nötigen Parametern versorgt, z. B. der Zeilenlänge des Bildausschnitts und den Startadressen für die MARC-Ströme. Außerdem wird zum nächsten Takt der Controller aktiviert.

In PHASE_WAVE_1_COMPUTE beginnt der Controller mit seiner Arbeit, während user auf das

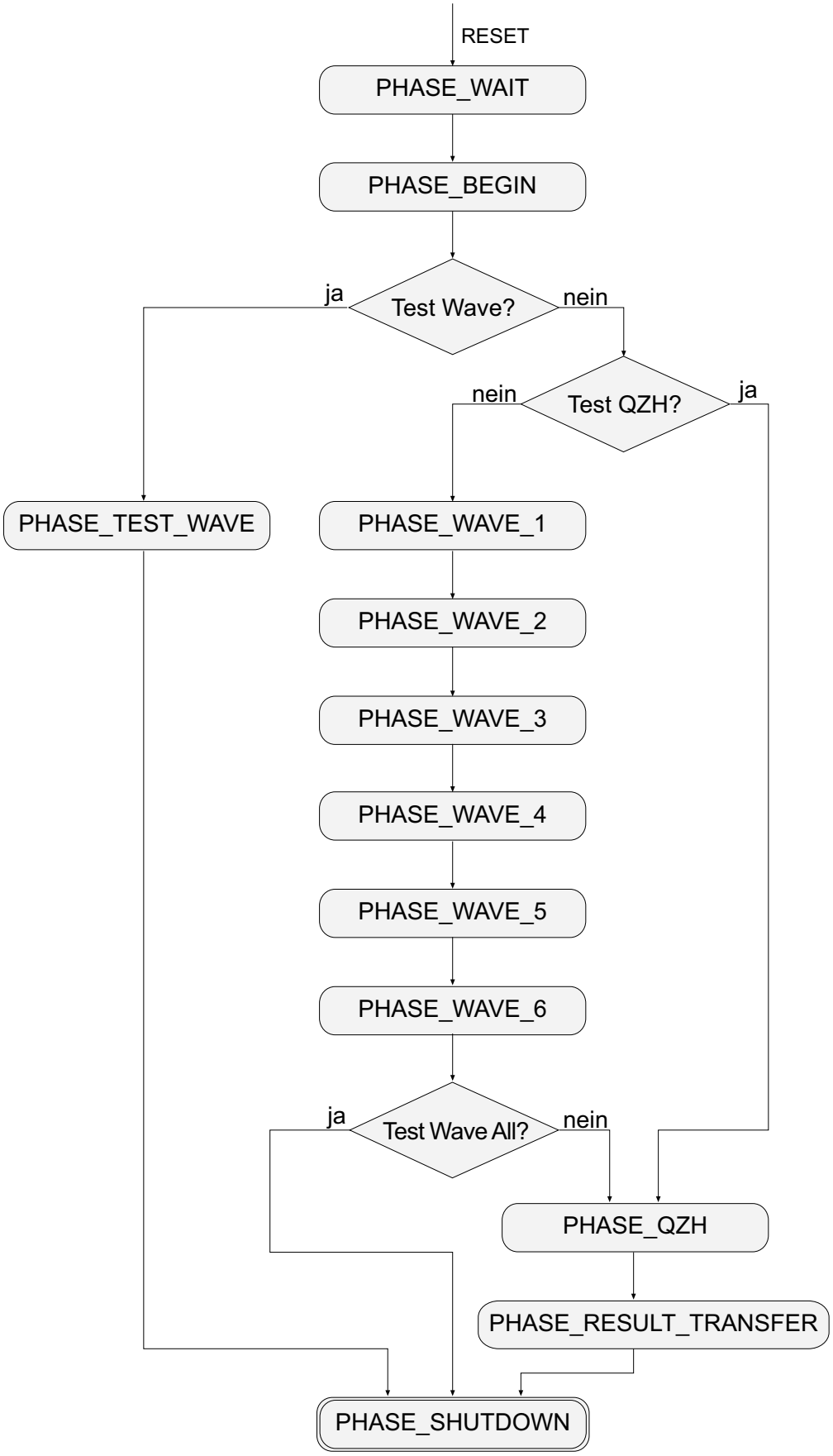


Bild 5.28:
Zustandsautomat des Moduls user. Die Aufschlüsselung in Programmier- und Compute-Phasen sowie die in zwei Abschnitte geteilten Spaltenbearbeitungsphasen sind hier der Übersichtlichkeit halber nicht aufgeführt.

Ende der aktuellen Transformation wartet.

Nach Abschluss von PHASE_WAVE_6 wechselt der Automat nach PHASE_SHUTDOWN, falls der Testmodus „TEST_WAVE_ALL“ aktiv ist (Durchlaufen aller Wavelet-Phasen). Im Normalbetrieb jedoch erfolgt zuvor noch die QZH-Bearbeitung (PHASE_QZH) mit anschließender Übertragung der 22 Ergebnisparameter im Zustand PHASE_RESULT_TRANSFER.

6 Testumgebung

Während der Entwicklung einer Hardware ist eine Testumgebung unabdingbar. Sie dient der Bestätigung des Entwurfs und hilft bei der Fehlersuche. Man unterscheidet zwischen Prä-Synthese-, Post-Synthese- und Post-Layout-Simulationen. In der Prä-Synthese-Simulation (hier auch *RTL-Simulation* genannt) wird direkt das unoptimierte RTL-Modell getestet. Gatter- und Leitungsverzögerungen werden hier vernachlässigt. Die Post-Synthese-Simulation testet die synthetisierte Schaltung inklusive Gatterverzögerungen, aber ohne Leitungsverzögerungen. Nach dem Platzieren und Verdrahten kann die Post-Layout-Simulation (hier auch *Layout-Simulation* genannt) erfolgen, die auch die Leitungsverzögerungen mit berücksichtigt.

Beim Entwurf von FPGA-basierter Hardware macht eine Post-Synthese-Simulation wenig Sinn, da beim Verdrahten deutlich spürbare Leitungsverzögerungen entstehen. Deswegen beschränkt man sich hier meist auf RTL- und Layout-Simulationen.

Die Simulation einer Anwendung auf einem adaptiven Rechner muss natürlich den Hardware- wie auch den Software-Teil berücksichtigen. Während die korrekte Funktionalität der Software mit herkömmlichen Mitteln erreicht werden kann, ist es in der vorliegenden Anwendung möglich, den Hardware-Teil separat zu testen. Dazu muss nicht nur die entworfene Schaltung selbst, sondern auch die Schnittstelle (z. B. für Slave-Mode-Zugriffe) und der Speicher simuliert werden.

Die Schnittstellen- und Speichersimulation kann für diese Anwendung von vorigen Arbeiten übernommen werden. Zu realisieren bleibt das Versorgen der Schaltung mit Testdaten sowie die Überwachung der Simulation und die Sicherung der Simulationsergebnisse.

Diese Aufgaben übernimmt der Testrahmen stimulus, dessen Funktionalität in 6.1 beschrieben wird.

Außer einem Hardware-Test mit emuliertem Speicher und emulierter Schnittstelle gibt es noch die Möglichkeit, die komplette Anwendung (Hardware- und Software-Teil gemeinsam) in einem Testmodus zu betreiben. Typischerweise wird nur - wie in 6.1 - der Hardware-Teil getestet. Der in 6.2 beschriebene, gemeinsame Test ist als kleines Zusatzfeature anzusehen.

6.1 Funktionen des Testrahmens stimulus

Das wichtigste Feature dieses Testrahmens ist die Auswahl des Test-Modus. Unterschieden werden vier Test-Modi:

- *normaler Betrieb* (testmode = 0): Simulation des gesamten Algorithmus mit den Bilddaten des Bildes lena256.pgm als Testmuster,
- *Wavelet-Test* (kurze Tests) (testmode = 1): ermöglicht kurze Wavelet-Transformationsdurchläufe, typischerweise nur die Bearbeitung von ein oder zwei Zeilen; verschiedene Konfigurationen und Testdaten einstellbar,
- *QZH-Test* (testmode = 2): Simulation der QZH-Berechnung auf dem wavelet-transformierten Lena-Bild mit anschließendem Ergebnisparameter-Transfer,
- *Wavelet-Test (komplett)* (testmode = 3): Simulation der sechs Wavelet-Phasen für das Lena-Bild, ohne anschließende QZH-Verarbeitung und ohne Ergebnisparameter-Transfer.

Hat man sich für einen Testmodus entschieden, wählt man als nächstes, ob der SRAM als Zwischenspeicher in der Wavelet-Transformation genutzt werden soll oder nicht. Da bei SRAM-Zugriffen keine Stalls auftreten, kann man so einerseits die Simulation gering vereinfachen und beschleunigen; der wesentliche Zweck dieses Modus ist aber, die Benutzung und Ansteuerung des SRAM zu verifizieren. Daten, die nach

Beendigung der Simulation noch im SRAM liegen, müssen zum Vergleichen ausgelesen werden. Ebenso müssen Stimuli zu Testbeginn evtl. in den SRAM kopiert werden. Um dies zu ermöglichen, wurde die vorhandene Datei „i960utils.v“ um die SRAM-Zugriffs-Tasks „WriteMemFileSRAM“ und „ReadMemFileSRAM“ erweitert.

Eine weitere Einstellmöglichkeit bietet die Variable quant. Ihr kann man die Komprimierungsstärke (0 bis 255) für den aktuellen Testdurchlauf zuweisen.

Im Testmodus 2 (QZH-Test) werden für die Berechnung die richtigen Blockminima und -maxima benötigt, darum muss das Modul min_max zuvor entsprechend befüllt werden. Hierfür wurde kein extra Mechanismus in der Hardware implementiert, daher wird direkt auf das Verilog-Modul min_max zugegriffen. Um z. B. für Wavelet-Block W0 das Maximum 239 zu setzen, benutzt man die folgende Zuweisung:

```
TOP.USER.Min_max.BLOCK_MAX_0 = 239;
```

Solche Zugriffe sind zwar praktisch, aber nur im RTL-Testmodus möglich, da nach der Synthese die Modulhierarchie plattgeklopft ist und z. B. die Instanz Min_max gar nicht mehr existiert. Dies hat zur Folge, dass der QZH-Modus mit den korrekten Argumenten zu einem Bild nur im RTL-Modus separat getestet werden kann! Möchte man ihn in der Layout-Simulation betreiben, benutzt man einfach Testmodus 0 (mit vorgeschalteter Wavelet-Transformations-Simulation).

Testmodus 1 (kurze Wavelet-Tests) ist sehr vielseitig. Man kann einzelne Wavelet-Phasen testen oder mehrere hintereinander im Zusammenhang, je nach Start- und End-Bedingung der for-Schleife

```
for (i = 0; i < 1; i = i + 1) begin
    case (i)
        0: phase = 1;
        1: phase = 2;
        2: phase = 3;
        3: phase = 3;
        4: phase = 4;
        5: phase = 4;
        6: phase = 5;
        7: phase = 5;
        8: phase = 6;
        9: phase = 6;
    endcase
    ...
end
```

Für i = 5 bis i < 7 würde man beispielsweise den zweiten Teil von Wavelet-Phase 4 und den ersten Teil von Phase 5 testen. Weiter unten im Quelltext lassen sich für j = 0 einzelne Zeilen- oder Spaltendurchläufe auf ausgewählten Testdaten simulieren, während j = 1 je eine komplette Phase (alle Zeilen bzw. Spalten) testet. Desweiteren lassen sich Eingabedaten (Quelle und Länge), Start- und Zieladressen, Zeilen-/Spaltenlängen, Wavelet-Berechnungsmodul (wavelet_8_4_l oder wavelet_16_2_hl) sowie Zieldatei für die Berechnungsergebnisse angeben.

Wie man welche Einstellungen vornehmen kann, lässt sich den Kommentaren im Quelltext entnehmen.

6.2 Kombinierter HW/SW-Test

Um die komplette Anwendung in einem Testmodus zu betreiben, ändert man in der Datei „main.c“ die Zuweisung

```
rc[REG_TESTMODE] = 0;
```

um in

```
rc[REG_TESTMODE] = 1;
```

Dies entspricht einem kurzen Wavelet-Test, wie in Abschnitt 6.1 für testmode = 1 angegeben. Die Beschränkung auf diesen einen Testmodus wurde vorgenommen, um die Menge an zu übertragenden Testparametern möglichst gering zu halten. Würde man in einem weiteren Modus die QZH-Verarbeitung separat testbar machen, sähe im Hardware-Teil der Multiplexer für die Slave-Mode-Eingangsdaten deutlich komplizierter aus. Es müssten dann nämlich zusätzlich zu den bisherigen Testparametern noch die Blockminima und -maxima übertragen werden (14 weitere Werte).

Abgesehen von der obigen Zuweisung müssen noch weitere Testparameter an die RC übertragen werden. Nehmen wir an, wir wollten die Wavelet-Phase 3 testen (d. h. 16 Bit pro Pixel, zeilenweise Bearbeitung, 128 Pixel pro Zeile). Als reservierter Speicher für die High-Daten der Transformation sei der C-Pointer dram_out_high gegeben. Getestet werden sollen zwei Zeilen, das entspricht 256 Pixeln, d. h. 128 Eingabeworte à 32 Bit (im Zeilenmodus sind die Worte 32 Bit lang, im Spaltenmodus 16 Bit). Angegeben werden stets Hexadezimalzahlen, also haben wir 128 = 0x80 Eingabeworte (und auch Ausgabeworte, da in Phase 3 Low- und High-Teil gesichert werden). Der zugehörige Code würde wie folgt lauten:

```
rc[REG_DRAM_WRITE_ADDR_TEMP] = dram_out_high;

rc[REG_INWORDS]              = 0x80
                             | blockthresh[1] << 16;

rc[REG_OUTWORDS]             = 0x80
                             | blockthresh[2] << 16;

rc[REG_ROW_WIDTH]            = 256
                             | blockthresh[3] << 16;

rc[REG_CONTROLLER_SWITCH]    = 0   // Wavelet-Testmodus
                             | blockthresh[4] << 16;

rc[REG_WAVE_MOD_SWITCH]      = 0   // 16 Bit/Pixel Eingabedaten
                             | blockthresh[5] << 16;

rc[REG_ROW_COLUMN_SWITCH     = 0   // zeilenweise Bearbeitung
                             | blockthresh[6] << 16;

rc[REG_WAVE_PHASE]           = 0;  // nicht Phase 2 testen
```

Wir bemerken, dass in den oberen 16 Bit (für den zweiten bis vorletzten Testparameter) jeweils ein Blockgrenzwert mitübermittelt wird. Dies geschieht, um Zeit zu sparen und die Adressierung in der Hardware zu vereinfachen (weniger ansprechbare Adressen verkleinern den Adressdecodierungs-Multiplexer). Da die Blockgrenzwerte nun bereits mit den Testparametern übertragen wurden, wird die im Quelltext folgende for-Schleife auskommentiert (wichtig, um die programmierten Testparameter nicht wieder zu überschreiben!).

7 Benchmark-Ergebnisse

Zunächst zeigen wir in diesem Kapitel, dass die (teils hardware-bedingten) Änderungen am originalen Versatility-Stressmark-Algorithmus [1] nicht zu einem stärkeren Qualitätsverlust nach der Komprimierung führen (Abschnitt 7.1). Danach sehen wir einige bildliche Vergleiche zwischen unterschiedlichen Komprimierungsstärken (Abschnitt 7.2). Das Kapitel schließt mit einem Laufzeitvergleich sowie einigen Angaben zur Chipfläche.

7.1 Signal- zu Rauschverhältnis und Bits pro Pixel

In [1] wird die Qualität eines Bildkomprimierverfahrens anhand zweier Werte gemessen:

- *PSNR*: „peak signal-to-noise ratio“, Signal-zu-Rauschverhältnis, Angabe in dB; gibt Auskunft über den Qualitätsverlust bei der Komprimierung. Ein höherer PSNR-Wert bedeutet einen niedrigeren Qualitätsverlust.
- *b*: Bitrate des komprimierten Bitstroms, Angabe in bpp (bits per pixel); gibt Auskunft über die Größe des Ausgabebitstroms relativ zur Größe des Originalbildes. Je kleiner b ist, umso kompakter konnte das Originalbild komprimiert werden.

Die Berechnungsformeln für PSNR und b lauten wie folgt (der Wert RMSE - „root mean square error“ wird zur Berechnung von PSNR benötigt):

$$PSNR = 20 \cdot log_{10}(\frac{255}{RMSE})$$

$$RMSE = \sqrt{\frac{1}{h^2} \cdot \sum_{i=1}^{h} \sum_{j=1}^{h} (D(i,j) - O(i,j))^2}$$

$$b = \frac{S \cdot 8}{h^2}$$

Dabei ist O das Originalbild, D das komprimierte und wieder dekomprimierte Bild, h die Bildhöhe (und gleichzeitig die Bildbreite) und S die Größe des komprimierten Bildes in Bytes.

Für die Autoren von [1] ist ein Algorithmus zur Wavelet-Bildkompression erst dann gut genug, wenn er für die drei Testbilder „Barbara“, „Goldhill“, „Lena“ die folgenden Ergebnisse für PSNR und b erzielt:

Bild	maximales b	minimales PSNR
Barbara	0.25 bpp	24.5 dB
Goldhill	0.25 bpp	28.0 dB
Lena	0.25 bpp	30.0 dB

Tabelle 7.1:
Vorgeschriebene Versatility-Stressmark-Werte für drei festgelegte Testbilder der Größe 512x512 Pixel.

Wie die tatsächlichen Werte lauten, hängt von der gewählten Komprimierungsstärke ab. Für die drei Stärken $q = 0$, $q = 128$ und $q = 255$ gibt Tabelle 7.2 die gemessenen Daten an.

Bild	b (bpp)			PSNR (dB)		
	q = 0	q = 128	q = 255	q = 0	q = 128	q = 255
Barbara	0.91	0.25	0.15	25.0	24.6	23.6
Goldhill	0.92	0.19	0.11	30.0	28.0	26.4
Lena	0.88	0.17	0.11	31.1	30.0	28.1

Tabelle 7.2:
Tatsächliche Kompressionswerte für die drei Testbilder, Größe 512x512, für je drei repräsentative Komprimierungsstärken (original Versatility-Algorithmus).

Wir bemerken, dass der Defaultwert für die Komprimierungsstärke $(q = 128)$ die verlangten Resultate in PSNR und b erfüllt.

Die betrachteten Werte gelten aber nur für Bilder der Größe 512x512 Pixel. Da die Hardware-Lösung für 256x256 Pixel ausgelegt ist, werden entsprechend angepasste Werte für diese Bildgröße benötigt. Theoretisch sind die angegebenen Werte zwar relativ und damit unabhängig von Bildgrößen. In der Praxis stimmt dies aber nicht, da größere Bilder besser komprimiert werden als kleinere Bilder, die das gleiche Motiv darstellen (je höher die Auflösung, um so mehr redundante Bereiche findet man in der Regel). Um Vergleichswerte für 256x256-Pixel-Bilder abzuleiten, wird der originale Versatility-Stressmark auf die drei herunterskalierten Testbilder angewendet. So ergeben sich die folgenden Daten:

Bild	b (bpp)			PSNR (dB)		
	q = 0	q = 128	q = 255	q = 0	q = 128	q = 255
Barbara	0.90	0.29	0.17	28.0	26.8	24.1
Goldhill	0.93	0.26	0.14	29.4	27.1	25.2
Lena	0.93	0.27	0.17	28.8	27.6	25.5

Tabelle 7.3:
Werte für die auf 256x256 Pixel skalierten Testbilder (original Versatility-Algorithmus).

Die Werte für $q = 128$ sollen von nun an als Richtwerte für Komprimierungen der drei Testbilder in der Größe 256x256 Pixel gelten.

Wir vergleichen diese Werte nun mit denen des laufzeitoptimierten Algorithmus (siehe Tabelle 7.4).

Bild	b (bpp)			PSNR (dB)		
	q = 0	q = 128	q = 255	q = 0	q = 128	q = 255
Barbara	0.90	0.29	0.17	28.1	26.8	24.8
Goldhill	0.93	0.26	0.14	29.4	27.6	25.6
Lena	0.93	0.27	0.17	28.8	27.6	25.5

Tabelle 7.4:
Werte für die auf 256x256 Pixel skalierten Testbilder (optimierter Algorithmus).

Beim Vergleich der 128er-Spalten zwischen Tabelle 7.3 und 7.4 erkennen wir, dass die Komprimierungsqualität durch die Optimierungen für die Hardware nicht gelitten hat. Im Fall des Bildes „Goldhill“ sehen wir sogar eine signifikante Verbesserung des Signal-zu-Rauschverhältnisses. Dies kann zurückgeführt werden auf den im Original-Algorithmus vorhandenen Fehler in der Huffman-Codierung: bis zu 7 Bits werden dort am Ende des Bitstroms verworfen (ganzzahlige Division bei der Umrechnung von Bits nach Bytes). In der hardwareoptimierten Version wurde dieser Fehler behoben, durch Anhängen eines weiteren Bytes, falls sonst ein Informationsverlust auftreten würde.

7.2 Bildvergleiche für unterschiedliche Komprimierungsstärken

Um einen subjektiven Eindruck von dem verwendeten Komprimierungsverfahren zu erhalten, werden die drei Testbilder „Barbara“, „Goldhill“ und „Lena“ der Größe 256x256 jeweils mit den drei komprimierten und wieder dekomprimierten Pendants verglichen (siehe Bilder 7.5 ff.).

Bild 7.5:
Rechts das Originalbild „Barbara“, links das komprimierte und wieder dekomprimierte Bild für die Komprimierungsstärken 0 (oben), 128 (Mitte) und 255 (unten).
Dateigrößen: Originalbild: 64 KB; Stärke 0: 7.23 KB; Stärke 128: 2.33 KB; Stärke 255: 1.38 KB.

Bild 7.6:
Rechts das Originalbild „Goldhill“, links das komprimierte und wieder dekomprimierte Bild für die Komprimierungsstärken 0 (oben), 128 (Mitte) und 255 (unten).
Dateigrößen: Originalbild: 64 KB; Stärke 0: 7.47 KB; Stärke 128: 2.05 KB; Stärke 255: 1.14 KB.

Bild 7.7:
Rechts das Originalbild „Lena“, links das komprimierte und wieder dekomprimierte Bild für die Komprimierungsstärken 0 (oben), 128 (Mitte) und 255 (unten).
Dateigrößen: Originalbild: 64 KB; Stärke 0: 7.40 KB; Stärke 128: 2.16 KB; Stärke 255: 1.34 KB.

7.3 Laufzeit und Fläche

Wir wollen die Laufzeit der Anwendung auf der ACE-V mit der Laufzeit der Softwarelösung auf einer Workstation sowie zwei herkömmlichen PCs vergleichen.

Die für das aus der Studienarbeit hervorgegangene Design maximale Taktrate der rekonfigurierbaren Einheit beträgt 30 MHz. In dieser Geschwindigkeit arbeitet also der Hardware-Teil der Anwendung. Der Software-Teil läuft auf der Micro-SPARC-CPU der ACE-V mit 100 MHz.

Als Vergleichssysteme dienen eine Workstation mit einer 900 MHz UltraSPARC III+ CPU, ein Standard-PC mit einer 1300 MHz AMD Athlon Thunderbird CPU sowie ein Standard-PC mit einer 1666 MHz AMD Athlon XP 2000+ CPU.

Als Testbild wird das Lena-Bild verwendet, mit der Komprimierungsstärke 0. (Die Tests ergaben, dass jedes andere Bild sowie jede andere Komprimierungsstärke die gleichen Laufzeiten verursachen.)

Folgende Ausführungszeiten wurden gemessen:

System	*Art der Messung*	*Laufzeit*
ACE-V, 30 MHz	Bildkomprimierung	6650 µs
ACE-V, 30 MHz	Laden der FPGA-Konfiguration und Bildkomprimierung	2205800 µs
UltraSPARC III+, 900 MHz	Bildkomprimierung	ca. 6700 µs
AMD Athlon Thunderbird, 1300 MHz	Bildkomprimierung	ca. 6000 µs (variiert je nach Auslastung)
AMD Athlon XP 2000+, 1666 MHz	Bildkomprimierung	ca. 3800 µs (variiert je nach Auslastung)

Tabelle 7.8:
Laufzeitvergleich zwischen der HW/SW-Anwendung auf der ACE-V und der Softwarelösung auf einer Workstation und einem Standard-PC.

Die zweite Zeile verwundert dabei kaum, da das Konfigurieren des FPGA auf der ACE-V im Sekundenbereich liegt. Man beachte aber, dass für die Berechnung auf n Bildern der FPGA nur einmal konfiguriert werden muss. Für das Komprimieren sehr vieler Bilder am Stück könnte die ACE-V daher schon sinnvoll eingesetzt werden.

Interessanter ist der Vergleich der Zeilen 1, 3, 4 und 5. Auf der ACE-V mit 30 MHz Taktung läuft die Hardware-Software-Anwendung geringfügig schneller als die Software-Lösung auf der UltraSPARC mit 900 MHz. Der 1300 MHz Standard-PC ist hier mit etwa 6000 µs etwas schneller als die ACE-V, die Ausführungszeiten variieren aber sehr stark. In den Tests lagen die Messungen zwischen 5500 und 7600 µs; der Mittelwert beträgt etwa die angegebenen 6000 µs. Der schnellste Testrechner war der in Zeile 5 angegebene Standard-PC mit einer 1666 MHz Athlon CPU.

Die platzierte und verdrahtete Hardware benötigt im Xilinx Virtex-1000-FPGA der ACE-V 41 % der vorhandenen Gatteräquivalente (5081 von 12288 Slices, eine Slice entspricht zwei Lookup-Tables und zwei Flip-Flops) und ist demnach eine verhältnismäßig große Schaltung. In den Bildern 7.9 bis 7.11 erhalten wir einen schematischen Überblick über das verdrahtete Design.

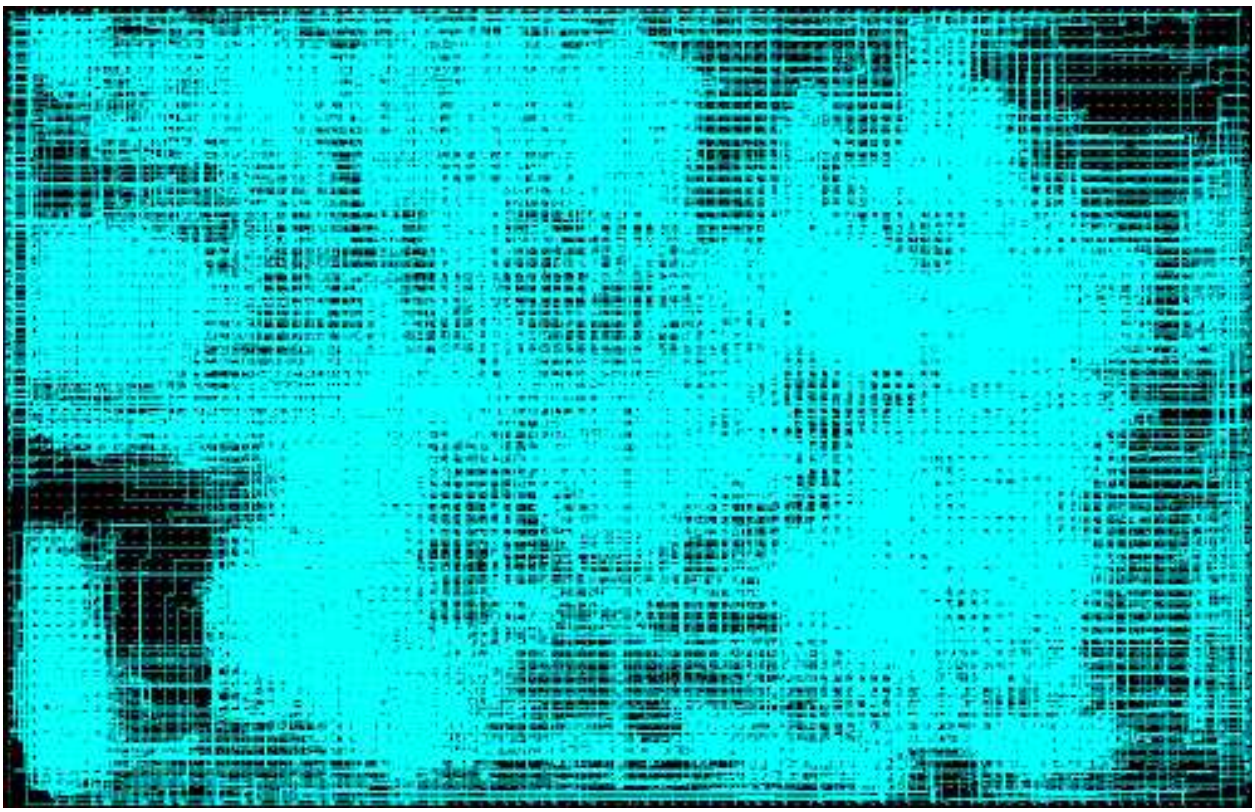


Bild 7.9:
Ansicht der verdrahteten Hardware im FPGA.

Bild 7.10:
Lage der Wavelet-Berechnungsmodule im FPGA.

Bild 7.11:
Lage der QZH-Berechnungsmodule im FPGA.

8 Zusammenfassung und Ausblick

Im Rahmen dieser Studienarbeit wurde eine Anwendung zur Bildkompression auf dem adaptiven Rechner ACE-V entworfen. Bearbeitet werden Graustufenbilder fester Größe, die durch eine Wavelet-Transformation in Grob- und Detailinformationen zerlegt werden. Dies macht eine effiziente, verlustbehaftete Komprimierung möglich. Der Benutzer der Anwendung kann die Komprimierungsstärke und damit den Qualitätsverlust durch Angabe eines Parameters beeinflussen. Bei geringem Qualitätsverlust kann ein Bild auf ca. 12 % der Originalgröße komprimiert werden, bei mittlerem Verlust auf ca. 2 %, bei starkem Verlust auf 1 %. Ein Hardware-Entwurf für die Dekomprimierung wurde in dieser Studienarbeit nicht erarbeitet.

Die Anwendung erfüllt die auf die Bildgröße 256x256 übertragenen Vorgaben des Versatility-Stressmark und kann als Benchmark für adaptive Rechner verwendet werden.

Beim Entwurf wurde auf eine schnelle Datenverarbeitung geachtet. Die Anwendung benötigt auf dem adaptiven Rechner ACE-V für die Komprimierung größenordnungsmäßig dieselbe Zeit wie eine UltraSPARC-Workstation oder ein Standard-PC. Genauer betrachtet ist die ACE-V hier etwas schneller als die UltraSPARC-Workstation, jedoch etwas langsamer als ein Standard-PC.

Ein Flaschenhals der Anwendung liegt im Speicherzugriff: da das verwendete Speicherzugriffssystem keinen „Spaltenverarbeitungsmodus“ anbietet, müssen nach der Transformation jeder einzelnen Spalte die Ein- und Ausgabeströme neu programmiert werden, was einen erheblichen Overhead verursacht.

Der zweite Flaschenhals ist die Wavelet-Transformation an sich: die sechs Phasen finden sequentiell statt; eine Parallelisierung „auf höchster Ebene“ ist so nicht möglich. Möglich wäre jedoch die parallele Transformation verschiedener Zeilen oder Spalten während derselben Phase, wenn man eine höhere Zahl unabhängiger Speicherblöcke und Datenströme hätte. Würde man die Reihenfolge der Phasen ändern, indem man zunächst alle Zeilen- und danach alle Spaltentransformationen durchführte, wäre es denkbar, eine Pipeline aus drei Zeilen- und eine aus drei Spaltentransformationsstufen zu entwerfen. So könnte man die bisherigen sechs sequentiellen Phasen auf zwei sequentielle reduzieren (erst alle Zeilen-, dann alle Spaltentransformationen), was aber auch wieder mehr unabhängige Speicherblöcke und Streams vorraussetzen würde.

Ähnlich verhält es sich mit der QZH-Pipeline. Die ersten beiden Pipeline-Stufen, Quantisierung und ZLE, finden bereits blockweise statt, die Huffman-Codierung aber arbeitet auf der Aneinanderreihung der zle-komprimierten Blöcke. Würde man das Format des komprimierten Bitstroms so ändern, dass jeder Block einzeln quantisiert, zle- und huffman-codiert würde, so könnte man anstatt einer einzigen QZH-Pipeline vier parallel arbeitende Instanzen einsetzen. Instanz 0 würde die Blöcke W0, W1, W2 und W3 komprimieren, Instanz 1 wäre für Block W4 verantwortlich, Instanz 2 für Block W5 und Instanz 3 für Block W6 (sinnvolle Aufteilung nach Blockgrößen). Für dieses Vorgehen bräuchte man lediglich vier Lese- und vier Schreibströme, aber entsprechend auch acht unabhängige Speicherbereiche, vier im SRAM (lesen) und vier im DRAM (schreiben).

Die genannten Erweiterungen können erst implementiert werden, wenn mehr unabhängige Speicherblöcke bzw. Datenströme auf der ACE-V möglich sind.
Ein anderes sinnvolles Zusatz-Feature würde jedoch mit den bestehenden Ressourcen auskommen: die Verarbeitung weiterer Bildgrößen, insbesondere der Größe 512x512 Pixel, um einen Benchmark auch für diese im Versatility-Stressmark vorgeschlagene Größe bereitzustellen. Für die Erweiterung auf 512x512 müsste man einerseits die fest codierten Anzahlen der Ein- bzw. Ausgabewörter in quantization und zle anpassen. Andererseits müssten die Block-Startadressen in user angepasst werden, und die Erzeugung eines neuen MARC-Cores wäre nötig, um die größere Datenmenge im SRAM zwischenzuspeichern.
Experimentieren könnte man außerdem mit anderen Wavelet-Basen, was allerdings ein komplettes Umprogrammieren der Module wavelet_8_4_l und wavelet_16_2_hl zur Folge hätte.
Zum Schluss sei noch die oben bereits angedeutete Variante hervorgehoben, die die Reihenfolge der

Wavelet-Phasen so ändert, dass erst alle Zeilen- und danach alle Spaltentransformationen stattfinden. Hierfür müssten allerdings die Default-Blockgrenzwerte evaluiert und ggf. angepasst werden.

9 Abkürzungsverzeichnis und Begriffserklärungen

High-Daten

Siehe „HP-Teil“.

High-Pass-Daten

Siehe „HP-Teil“.

HP-Teil

Differenz- oder Detail-Ausgabedaten einer Wavelet-Transformation. Wird auch als „hochfrequenter Teil“ der Daten bezeichnet.

HW/SW-Lösung

Hardware-Software-Lösung.

Low-Daten

Siehe „LP-Teil“.

Low-Pass-Daten

Siehe „LP-Teil“.

LP-Teil (auch: „Low-Daten“ oder „Low-Pass-Daten“)

Mittelwertdaten oder vergröberte Bildinformationen als Ausgabe einer Wavelet-Transformation. Wird auch als „niederfrequenter Teil“ der Daten bezeichnet.

QZH

Kurzform für die Verarbeitungspipeline „Quantisierung - Zerolängencodierung - Huffman-Codierung“.

RC

Rekonfigurierbare Einheit eines adaptiven Rechners (meist ein FPGA); engl.: „reconfigurable component“.

Stream

(Daten-)Strom; der Inhalt eines Speicherbereichs wird in fester Reihenfolge durch eine Schaltung geleitet.

Wavelet-Stufe

Zwei vollständige, aufeinander folgende Wavelet-Transformationsdurchläufe durch ein Eingabebild. Der erste Durchlauf erfolgt zeilenweise, der zweite spaltenweise.

ZLE

Zerolängencodierung (engl.: „zero length enconding“), Einschränkung einer Lauflängencodierung (engl.: „run length encoding“, RLE) auf die Codierung von Läufen (engl.: „runs“) eines einzigen speziellen Wertes (dieser Wert wird hier mit „ZERO“ bezeichnet).

10 Literaturverzeichnis

[1] Honeywell Technology Center, Minneapolis, „Benchmark Specification Document Versatility Stressmark (CDRL A001), Version 0.8“; November 1997.

[2] Lange, H., Koch, A., „Memory Access Schemes for Configurable Processors“, Tech. Univ. Braunschweig (E.I.S.), Germany.

[3] Li, Y., Callahan, T., Darnell, E., Harr, R., Kurkure, U., Stockwood, J., „Hardware-Software Co-Design of Embedded Reconfigurable Architectures“.

[4] Stollnitz, Eric J. , DeRose, Tony D., Salesin, David H., „Wavelets for Computergraphics: A Primer, Part 1“, University of Washington, 1994.

[5] Koch, A., „A comprehensive Prototyping-Platform for Hardware-Software Codesign“ Tech. Univ. Braunschweig (E.I.S.), Germany.

[6] Beschreibung des PGM-Bildformats: http://astronomy.swin.edu.au/~pbourke/dataformats/ppm/

[7] Uytterhoeven, G., Roose, D., Bultheel, A., „Wavelet Transforms Using the Lifting Scheme“, Report ITA-Wavelets-WP1.1 (Revised Version); 1997.

[8] Koch, A., „Architectures and Tools for Heterogeneous Reconfigurable Systems“, IEEE Workshop on Heteregeneous Reconfigurable Systems-on-Chip, Hamburg, April 2002.