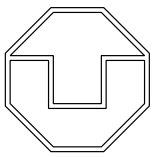
Technische Universität Dresden

Fakultät Elektrotechnik

Institut für Grundlagen der Elektrotechnik und Elektronik



Studienarbeit

Thema: FPGA-Implementierung der Quantisierung und Codierung des JPEG-Kompressionsalgorithmus

Vorgelegt von: Ralf Hildebrandt

Betreuer:	Dipl.-Ing. Jens-Uwe Schlüßler
Verantwortlicher Hochschullehrer:	Prof. Dr.-Ing. habil. René Schüffny
Tag der Einreichung:	9.11.2001

Zusammenfassung

Im folgenden wird eine Realisierung der Quantisierung und Huffman-Codierung des JPEG-Kompressionsalgorithmus präsentiert, welche synthetisiert und auf eine FPGA von Xilinx implementiert wurde. Es handelt sich um baseline JPEG und damit um eine verlustbehaftete Komrepssion.

Diese Realisierung des JPEG-Algorithmus ist angepasst an das spezifische Ausgabeformat und die Auflösung des Bildsensors VISP2000 [1].

Als Entwicklungswerkzeug wurde VHDL mit den Entwicklungstools von Synopsys und Xilinx eingesetzt.

Inhaltsverzeichnis

1 Aufgabenstellung

Im Zuge dieser Studienarbeit war eine Realisierung der Quantisierung und Codierung des JPEG[1]-Bildkompressionsalgorithmus [6] zu entwickeln. Diese Realisierung sollte synthesefähig sein und auf eine FPGA[2] von Xilinx [11] implementiert werden. Im speziellen handelt es sich um das Modell Virtex V400BG432-5. Der Algorithmus war in der Hardwarebeschreibungssprache VHDL zu beschreiben.

Des weiteren musste die Realisierung an den Bildsensor VISP2000 und dessen spezielles Ausgabeformat angepasst werden.

Die Basis dafür bietet ein Demonstrator-Board, welches in der Studienarbeit von Mirko Pügner [3] beschrieben ist. Diese Studienarbeit selbst ist auch als Ausgangsbasis verwendet worden.

2 Problemüberblick

Im folgenden Überblick wird das Problem in Teilprobleme zerlegt und grundlegende Ideen zur Lösung werden vorgestellt. Eine detailliertere Beschreibung und die Vorstellung der Realisierung folgt später im Abschnitt 5.

2.1 Das Demonstrator-Board

Mirko Pügner hat in seiner Studienarbeit [3] das Demonstrator-Board beschrieben. Deshalb soll hier nur ein grober Überblick wiedergegeben werden: Das Demonstrator-Board besteht aus einer konfigurierbaren Schnittstelle zu einem (an diese Schnittstelle angepassten) Bildsensor, einer FPGA zur Verarbeitung und zur Weitergabe der Bilddaten und einer FireWire-Schnittstelle (IEEE 1394 / iLink), welche die Aufgabe hat, die verarbeiteten Daten an einen PC weiter zu leiten. Die FireWire-Schnittstelle wird gesteuert durch einen DSP von Texas Intruments [4].

Das Ergebnis der Studienarbeit von Mirko Pügner war die Realisierung der Datenausgabe von der FPGA zu dem FireWire-steuernden DSP. Innerhalb der FPGA wird eine Schnittstelle bereit gestellt, die es erlaubt, Daten an die Ausgabestufe in Richtung FireWire weiterzugeben. Diese Schnittstelle sollte in meiner Studienarbeit zur Ausgabe des JPEG-Datenstromes verwendet werden.

Das Demonstrator-Board bietet zwar noch einige weitere Möglichkeiten, wie z.B. einen Dual-Port-RAM, auf den die FPGA Zugriff hat und auch RS232-Schnittstellen (COM-Port) für Steuerungs- und Debugging-Aufgaben, aber diese Dinge sind für die hier vorgestellte Bildverarbeitung und -ausgabe über FireWire nicht von Bedeutung. Der Dual-Port-RAM würde Bedeutung erlangen, wenn sehr große Teile eines Bildes zwischenzeitlich abgelegt werden müssten, um z.B. großflächige Bildoperationen anwenden zu können.

2.2 JPEG

Der JPEG-Standard, wie er in [6] abgedruckt ist, sieht mehrere Möglichkeiten zur Kompression von Bilddaten vor. Es soll hier nur die einfachste Möglichkeit

[1] Joint Photographic Experts Group
[2] field programmable grid array

der verlustbehafteten Kompression realisiert werden, welche des Namen „baseline JPEG“ im Englischen trägt.

Baseline JPEG beinhaltet die Anwendung der diskreten Cosinus - Transformation auf die Bilddaten, eine Quantisierung und eine Huffman-Codierung. Details folgen in Abschnitt 3.

Alle anderen Möglichkeiten von JPEG, wie hirarchisches Codieren, arithmetisches Codierung oder verlustlose Kompression werden nicht betrachtet, da sie meist einen erheblichen Mehr-Aufwand an Hardware mit sich bringen würden. Für den Bildsensor VISP2000 und dessen begrenzte Auflösung von 64×128 Bildpunkten ist baseline JPEG völlig ausreichend. Auch ist baseline JPEG das Grundformat, welches jeder Bildebtrachter verstehen sollte, der JPEG darstellen kann.

Baseline JPEG bietet einen guten Kompromiss zwischen hoher Kompression und geringem Codierungsaufwand.

2.3 Die Fähigkeiten des VISP2000

Der Bildsensor VISP2000 hat die Fähigkeit, bestimmte Berechnungen in einer mixed-signal - Vorverarbeitungsstufe direkt hinter den Pixel-Zellen auszuführen, wie es in [1] beschrieben ist. Zu den ausführbaren mathematischen Operationen zählen bestimmte Filter, wie z.B. Gauß-Filter, aber auch die Möglichkeit der Berechnung der DCT. Damit kann ein wesentlicher Schritt des JPEG-Algorithmus sehr verlustleistungsarm im VISP2000 ausgeführt werden. Ein spezieller DSP für die DCT ist nicht mehr nötig.

Allgemein ist folgender Kernel im Kamerasensor realisierbar:

$$I(u,v) = \sum_{x=0}^{7} \sum_{y=0}^{7} \underbrace{X(u+x, v+y)}_{Pixelwerte} \quad \underbrace{c(x,y)}_{Kernel(switches)} \tag{1}$$

Die Variablen x und y stellen dabei die Koordinaten der Pixel dar. Größere als 8×8-Kernel sind durch geschickte Wahl der Kernelkoeffizienten realisierbar. Für Die DCT bei JPEG wird aber ein 8×8-Kernel verwendet, der direkt ohne zusätzliche Umstrukturierung bearbeitet werden kann.

2.4 Die Quantisierung

Durch die DCT wurde das Bild in den Frequenzbereich transformiert. Das stellt an sich noch keine Informationsreduktion dar, sondern nur eine Umordnung der Bildinformation gemäß ihrer spektralen Anteile.

Baseline JPEG sieht nun eine Quantisierung der DCT-transformierten Bilddaten vor. Da das menschliche Auge weniger empfindlich auf hochfrequente Bildanteile ist, können diese stärker quantisiert werden, was zu einer Informationsreduktion durch Irrelevanzelimination und damit zu einem niedrigerem Datenstrom führt.

Die Quantisierung entspricht in den Grundzügen einer Division. Beschränkt man sich aber auf eine nicht variable Quantisierung, indem man einmalig die Quantisierungskoeffizienten festlegt, so kann man diese Division als Multiplikation mit den bekannten (da nicht variablen) Inversen der Divisoren beschreiben. Damit spart man die aufwändige Realisierung einer Division in Hardware.

2.5 Die Codierung

Baseline JPEG verlangt eine Huffman-Codierung.

Das Prinzip dieser Codierung ist folgendes: Man ermittle das Datenwort mit der höchsten Wahrscheinlichkeit des Auftretens im Datenstrom. Diesem Datenwort weise man ein möglichst kurzes Codewort zu. Weniger häufig auftretende Datenwörter erhalten ein entsprechend längeres Codewort. Prinzipiell ähnelt dieses Verfahren dem Morse-Code, aber der Huffman-Algorithmus produziert einen optimalen Code, der ein kompakter[3] Code ist. Details findet man in der Vorlesung von Professor Adolf Finger [12] und in vielen anderen Publikationen, sowie im Internet. Zwei anschauliche Beispiele für die Arbeitsweise des Algorithmus findet man z.B. in [13] und in [14].

Eine Huffman-Codierung allein reicht aber für den gewünschten hohen Kompressionsgrad bei JPEG nicht aus, sodass zusätzlich eine Lauflängencodierung eingesetzt wird. Bei einer Lauflängencodierung werden mehrere aufeinander folgende Datenwörter zu einem Codewort zusammengefasst.

Motiviert wird diese Lauflängencodierung durch die vorangegangene Quantisierung. Durch die Quantisierung werden viele Werte zu Null werden. Es bietet sich damit an, diese Nullen zusammen zu fassen.

Da eine möglichst einfache und platzsparende Realisierung favorisiert wurde, wäre eine Echtzeit-Huffman-Codierung zu aufwändig geworden. Eine solche Codierung müsste ein gesamtes Bild analysieren und die Auftrittswahrscheinlichkeiten der einzelnen Datenworte errechnen, um ein geeignetes Codealphabet aufzubauen. Stattdessen wurde ein vorgegebenes Huffman-Codewort-Alphabet verwendet, dass in [6] beispielhaft angegeben wurde. Dieses Alphabet wurde aus den Auftrittswahrscheinlichkeiten der Lauflängen-codierten Wörter vieler durchschnittlicher Bilder aufgebaut. Damit passt es zwar nie optimal zu einem Bild, bietet aber vergleichsweise gute Kompressionsresultate bei sehr geringem Codierungsaufwand.

2.6 Direkte Datenausgabe

Will man eine andere Berechnung als die DCT im Bildsensor VISP2000 ausführen lassen (z.B. einen Filterkernel), so muss die JPEG-Quantisierung und -Codierung abschaltbar oder umgehbar sein. Somit werden die Bilddaten direkt ausgegeben, ohne dass eine JPEG-Codierung angewendet wird. Man kann dann die Implementierung der Schnittstelle VISP2000 $\Leftrightarrow$ FPGA nutzen, um diese Daten direkt auszugeben.

Diese direkte Datenausgabe stellt eine kleine zusätzliche Aufgabe dar.

3 Baseline JPEG

In diesem Kapitel werden die grundsätzlichen Prinzipien von baseline JPEG erläutert, die bei dieser Studienarbeit von Bedeutung waren. Tiefergehende Erläuterungen sind zu finden in [7], [8] und vor allem in [6], wo auch der JPEG-Standard abgedruckt ist. Der Standard umfasst weit mehr Möglichkeiten, als hier bei der Realisierung verwendet wurden.

[3] Die mittlere Codewortlänge eines kompakten Codes ist gleich der minimalen mittleren Codewortlänge bzw. weist der Codebaum keine freien Enden auf.

3.1 Die DCT

Die Diskrete Cosinus Transformation (2) wird in baseline JPEG angewendet auf ein Pixelfeld der Größe 8×8.

$$F(u,v) = \frac{1}{4}C(u)C(v)\sum_{x=0}^{7}\sum_{y=0}^{7} f(x,y)\cos\frac{(2x+1)u\pi}{16}\cos\frac{(2y+1)v\pi}{16} \qquad (2)$$

$$u, v \in [0,7]$$

$$C(u) = C(v) = \begin{cases} 1/\sqrt{2}, & \text{wenn } u,v = 0 \\ 1, & \text{sonst} \end{cases} \qquad (3)$$

Dabei bezeichnet $f(x,y)$ den Wert eines Pixels an der Stelle (x,y). Dieser Wert kann eine Farbinformation, oder wie beim VSIP2000 eine Helligkeitsinformation sein. (Im folgenden ist nur noch von einer Helligkeitsinformation die Rede.)

Mit Hilfe der DCT berechnet man aus den 64 Pixelwerten 64 Koeffizienten $F(u,v)$ mit $u,v \in [0,7]$.

$F(0,0)$ wird als Gleichanteil bezeichnet und entspricht einem Mittelwert der Helligkeit über alle 64 Pixel. Dieser Gleichanteil (DC-coefficient) wird vom menschlichen Auge besonders stark wahrgenommen und ist deshalb in der späteren Bearbeitung besonders wichtig. Alle anderen 63 Koeffizienten (AC coefficients) entsprechen einer mehr oder weniger starken „Welligkeit“ in x- bzw. y-Richtung.

Damit befindet sich die wesentliche Information, die in dem Bild vorhanden ist in der linken oberen Ecke des DCT-transformierten 8×8 -Feldes, wenn man die Koeffizienten wie in Abbildung 1, S. 6 anordnet.

Reiht man die Koeffizienten in der Zig-Zag-Ordnung wie in Abbildung 1 illustriert auf, so entsteht eine Reihenfolge, die etwa der Wichtigkeit der Frequenzanteile bei der Wahnehmung im menschlichen Auge entspricht. Höherfrequente Anteile sind zudem oft weniger stark als niederfrequente Anteile in einem typischen Bild zu finden.

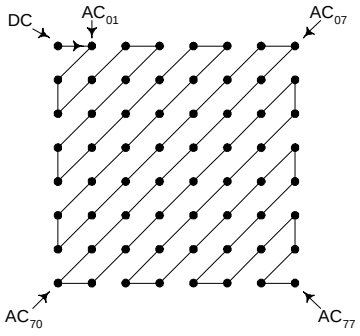


Abbildung 1: Zig-Zag-Ordnung (Quelle: [8])

Die DCT hat damit eine Basis geschaffen, die für eine effektive Redundanz- und Irrelevanzreduktion nötig ist, indem sie die Information, die in dem Bildes enthalten ist, zusammen gefasst hat.

Damit der Wertebereich der DCT verringert wird, sieht JPEG eine Verschiebung aller Pixelwerte um -128 vor. Damit erhält man aus (2) folgende Formel:

$$F^*(u,v) = \frac{1}{4}C(u)C(v)\sum_{x=0}^{7}\sum_{y=0}^{7}(f(x,y)-128)\cos\frac{(2x+1)u\pi}{16}\cos\frac{(2y+1)v\pi}{16} \tag{4}$$

und es gilt:

$$Wb(F^*(u,v)) = [-1023, 1024] \tag{5}$$

Da der Wertebereich des VISP2000 auch $[-1023, 1024]$ umfasst, könnten alle Koeffizienten theoretisch im VISP2000 berechnet werden. Eine solche Verschiebung um -128 ist aber in der Form nicht auf dem VISP2000 zu berechnen, wie Gleichung (1) zeigt. (Negative Pixelströme sind nicht möglich.) Also muss man die Verschiebung um -128 extern realisieren, da sie ein wichtiger Teil des JPEG-Algorithmus ist. Formt man (4) um, so erhält man:

$$F^*(u,v) = F(u,v) - \frac{1}{4}C(u)C(v)\sum_{x=0}^{7}\sum_{y=0}^{7}128\cos\frac{(2x+1)u\pi}{16}\cos\frac{(2y+1)v\pi}{16} \tag{6}$$

$$F^*(u,v) = \begin{cases} F(u,v) - 1024, & \text{wenn } u = v = 0 \\ F(u,v), & \text{sonst} \end{cases} \tag{7}$$

Gleichung (7) wurde berechnet mit Hilfe von Kapitel 9.1.

Hat man nun die Verschiebung um -128 nach aussen verlagert, kann man im VISP2000 nicht mehr von der Begrenzung des Wertebereichs profitieren. Da der Wertebereich $[0, 2048]$ der DC-Koeffizienten von Gleichung (2) auf dem VISP2000 nicht erreichbar ist, muss also folgerichtig bei DC-Koeffizienten die Aussteuerung des VISP2000 halbiert werden. Dies geschieht durch geeignete Wahl der Ansteuerungskoeffizienten des VISP2000. Die berechneten Daten müssen dann extern verdoppelt werden.

Aus diesem Grund besitzt der DC-Koeffizient leider nur die Hälfte der Auflösung der AC-Koeffizienten, aber das sollte sich auf den visuellen Bildeindruck nicht so schwerwiegend auswirken.

3.2 Die Quantisierung

Um eine Irrelevanzreduktion und damit eine Kompression des Datenstromes zu erreichen, werden die DCT-transformierten Pixelwerte quantisiert.

$$F^Q(u,v,) = integer\ round\left(\frac{F^*(u,v)}{Q(u,v)}\right) \tag{8}$$

Dabei ist $Q(u,v)$ ein 8×8-Array von festen Quantisierungskoeffizienten. Wichtigen Bildanteilen, wie dem Gleichanteil und den niederwertigen Wechselanteilen werden kleine Quantisierungskoeffizienten, weniger wichtigen Anteilen große zugeordnet. Wie stark man jeden Koeffizienten quantisieren kann, hängt vom Bild und von der menschlichen Wahrnehmung ab. Hat man ein spezielles Anwendungsgebiet mit sehr ähnlichen Bildern, bei denen auf bestimmte Merkmale Wert gelegt wird, so müsste man in einer Studie untersuchen, wie stark welcher Koeffizient quantisiert werden darf, um akzeptable Resultate zu erzielen. Für verschiedene Bilder aus vielen möglichen Anwendungsgebieten wurde eine

solche Studie mit mehreren Testpersonen durchgeführt. Das Ergebnis ist publiziert in [6] und wird in dieser Studienarbeit verwendet, da bei einer Kamera ein ebenso breites Spektrum an Bildern zu erwarten ist.

Ist man mit den Ergebnissen der Quantisierung nicht zufrieden, so kann man z.B. alle Quantisierungskoeffizienten halbieren und erhält ein deutlich weniger „blockiges“ Bild.

Damit eine Bildbetrachtungssoftware weiß, welcher Koeffizient wie stark quantisiert wurde, müssen die Quantisierungskoeffizienten dem Bild als zusätzliche Information beigelegt werden. Dies geschieht im JPEG-Header, der später in Kapitel 3.4 vorgestellt wird.

3.3 Die Codierung

3.3.1 Die Codierung des Gleichanteils

Da der Gleichanteil z.T. deutlich anderes statistisches Verhalten aufweist als die Wechselanteile und zudem die wichtigste Bildinformation ist, wird er gesondert von den Wechselkoeffizienten codiert. Weil die Gleichanteile von benachbarten 8×8-Blöcken eine starke Korrelation besitzen, d.h. häufig sehr ähnlich sind, da ein Bild oft größere homogene Bereiche besitzt, bietet sich eine Differenz-Codierung an, wie in Abbildung 2, S. 8 illustriert. Beim Start eines jeden Bildes

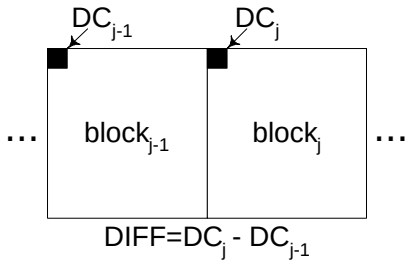


Abbildung 2: differenzielles Codieren des Gleichanteils (Quelle: [8])

wird der erste Gleichanteil DC_0 bezüglich des Wertes 0 codiert.

Durch die vorangegangene Quantisierung und die Differenzbildung sind oft nur noch recht kleine Zahlenwerte zu übertragen. Es wäre Verschwendung, jedes Mal ein Wort mit der vollen Bitbreite des worst-case-Falles zu übermitteln. Man versucht, nur die wirklich nötigen Bits zu übertragen und unterteilt deshalb den Wertebereich der differenzcodierten Gleichanteile wie folgt:

Kategorie	Wertebereich
0	0
1	-1 ; 1
2	-3,-2 ; 2,3
3	-7,. . . ,-4 ; 4,. . . ,7
4	-15,. . . ,-8 ; 8,. . . ,15
5	-31,. . . ,-16 ; 16,. . . ,31
6	-63,. . . ,-32 ; 32,. . . ,63
7	-127,. . . ,-64 ; 64,. . . ,127
8	-255,. . . ,-128 ; 128,. . . ,255
9	-511,. . . ,-256 ; 256,. . . ,511
10	-1023,. . . ,-512 ; 512,. . . ,1023
11	-2047,. . . ,-1024 ; 1024,. . . ,2047

Jede dieser Kategorien wird später ein eigenes Codewort bebekommen. Doch damit ist (ausser bei der Kategorie 0) der tatsächliche Wert der DC-Differenz nicht voll beschrieben. Deswegen werden direkt nach dem Codewort für die Kategorie genau so viele zusätzliche Bits an den Datenstrom angehängt, wie die Kategorie gross ist. Die Regel für die zusätzlichen Bits ist folgende: Ist die DC-Differenz positiv, so werden die niederwertigsten Bits der DC-Differenz angehängt. Ist die DC-Differenz dagegen negativ, wird 1 von der DC-Differenz subtrahiert und dann die niederwertigsten Bits anhängt.

Das Ergebnis ist dann „ein Codewort für die Kategorie c“ und c weitere Bits der DC-Differenz.

3.3.2 Die Codierung der Wechselanteile

Generell werden die Wechselanteile gemäß der Zig-Zag-Reihenfolge codiert.

Der Codierung der Wechselanteile liegt eine komplett andere Idee als bei den Gleichanteilen zugrunde, die durch ihre speziellen statistischen Eigenschaften begründet ist.

Durch die Quantisierung sind viele AC-Koeffizienten zu Null geworden - ganz besonders die höherfrequenten Wechselanteile. Es bietet sich also eine Lauflängencodierung an, die die Anzahl von Nullen vor einem nicht-Null-Wert mit in das Codewort hinein bringt (den sogenannten „run“). Bei JPEG wurde vereinbart, maximal 15 Nullen plus einem folgenden nicht-Null-Wert so zu einem Codewort zusammenzufassen. Treten mehr als 15 Nullen hintereinander auf, so wird es codiert als „Folge von 15 Nullen, gefolgt von dem Wert Null“, was effektiv 16 aufeinander folgenden Nullen entspricht. Dieses seltene Ereignis nennt man Null-Reihe (zero-line (ZRL)).

Da, wie gesagt oft die höherfrequenten Anteile fast alle Null sind, braucht man diese nicht zu codieren, wenn keine nicht-Null-Koeffizieten in dem 8×8-Block mehr folgen. Folgen also nach dem letzten nicht-Null-Wert ausschliesslich Nullen, so wird der 8×8-Block abgeschlossen mit „end of block“ (EOB).

Ein (nicht-Null-)AC-Wert, der codiert wird, wird ebenfalls dem Kategorie-Schema aus Tabelle 3.3.1 unterworfen. Er wird dann wie die DC-Differenz auch codiert als Codewort für die Kategorie plus je nach Grösse der Kategorie ein paar zusätzliche Bits des AC-Wertes. (Auch hier gilt die selbe Regel für positive bzw. negative AC-Werte, wie bei der DC-Differenz.)

Das endgültige Codewort-Konstrukt sieht dann folgendermaßen aus: „ein run der Länge r und die Kategorie der Größe c“ - gefolgt von den zusätzlichen c Bits.

3.4 Der Header

Das JPEG-Bildformat sieht vor, im Header des Bildes mehrere Informationen zum Bild einzubetten. Unter anderem werden die Quantisierungstabelle sowie das Huffman-Codewortalphabet in den Header genommen, damit stets garantiert ist, das Bild exakt wieder rekonstruieren zu können.

Anmerkung: Es existiert ein JPEG-Format, welches bei einer Folge von JPEG-komrpimierten Bildern nur einmalig den Header beim ersten Bild übertragt, sofern der Header bei allen Bildern gleich bleibt. Dieses Format wird aber eher selten von den üblichen Bildbetrachtern unterstützt. Daher wird hier zu jedem Bild ein eigener Header ausgegeben.

Basierned auf den Vorschlägen in [6] wurde folgender Header in dieser Studienarbeit verwendet (Angabe hexadezimal):

```
ffd8
ffdB
00 43 00 10 0b 0c 0e 0c 0a 10 0e 0d 0e 12 11 10
13 18 28 1a 18 16 16 18 31 23 25 1d 28 3a 33 3d
3c 39 33 38 37 40 48 5c 4e 40 44 57 45 37 38 50
6d 51 57 5f 62 67 68 67 3e 4d 71 79 70 64 78 5c
65 67 63
ffc0
00 0b 08 00 80 00 40 01 01 11 00
ffc4
00 d2 00 00 01 05 01 01 01 01 01 01 00 00 00 00
00 00 00 00 01 02 03 04 05 06 07 08 09 0a 0b 10
00 02 01 03 03 02 04 03 05 05 04 04 00 00 01 7d
01 02 03 00 04 11 05 12 21 31 41 06 13 51 61 07
22 71 14 32 81 91 a1 08 23 42 b1 c1 15 52 d1 f0
24 33 62 72 82 09 0a 16 17 18 19 1a 25 26 27 28
29 2a 34 35 36 37 38 39 3a 43 44 45 46 47 48 49
4a 53 54 55 56 57 58 59 5a 63 64 65 66 67 68 69
6a 73 74 75 76 77 78 79 7a 83 84 85 86 87 88 89
8a 92 93 94 95 96 97 98 99 9a a2 a3 a4 a5 a6 a7
a8 a9 aa b2 b3 b4 b5 b6 b7 b8 b9 ba c2 c3 c4 c5
c6 c7 c8 c9 ca d2 d3 d4 d5 d6 d7 d8 d9 da e1 e2
e3 e4 e5 e6 e7 e8 e9 ea f1 f2 f3 f4 f5 f6 f7 f8
f9 fa
ffda
00 08 01 01 00 00 3f 00
```

Mit der Bitfolge „FF“ wird immer ein sogenannter Marker angekündigt. Aus diesem Grund muss im späteren Datenstrom gewährleistet werden, dass nicht irrtümlich eine Bitfolge als Marker interpretiert wird. Falls also im Datenstrom die Bitfolge „FF“ auftreten sollte, muss das folgende Worf „00“ heißen. Diese Bitfolge veranlasst eine Bildbetrachtungssoftware, „FF“ als Daten und nicht als Marker zu interpretieren.

Es folgt nun der Header im Detail. Hier sollen nur stichpunktartige Erklärungen genügen. Die exakten Definitionen der einzelnen Header-Segmente befinden sich in [6].

Marker	Parameter/wert	Kommentar
FFD8		(SOI) start of image
FFDB		(DQT) define quantisazion table
	0043	Länge der Quantisierungstabelle in Byte
		(einschl. dieser Längenangabe, ausschl. Marker)
	0	Präzision (8 Bit)
	0	Qunatisierungstabbellenidentifikation
		(laufende Nummer der Quantisierungstabelle)
	Rest	64 Qunatisierungskoeffizienten
FFC0		(SOF) start of frame (baseline JPEG)
	000B	Länge des frame header
	08	8 Bit sample precision
	0080	Auflösung in y-Richtung
	0040	Auflösung in x-Richtung
	01	Anzahl Komponenten im Frame
	01	Komponentenidentifikation
	1	horizontaler Sampling-Faktor
	1	vertikaler Sampling-Faktor
	00	Wahl der Quantisierungstabelle
FFC4		(DHT) define huffman table
	00D2	Länge der Huffman-Tabelle
	0	Wahl der Codetabelle für DC-Werte
	0	Huffmantabellenidentifikation
	16 Bytes	Anzahl L_i der Codewörter der Länge i
		$i = 1, \ldots, 16$
	12 Bytes	Wert, zugeordnet zu jedem Huffman-Code
		$i = 1, \ldots, 16; j = 1, \ldots, L_i$ hier: Kategorie
	1	Codetabelle für AC-Werte
	0	Wahl der Huffmantabelle
	16 Bytes	Anzahl L_i der Codewörter der Länge i
		$i = 1, \ldots, 16$
	Rest	Wert, zugeordnet zu jedem Huffman-Code
		$i = 1, \ldots, 16; j = 1, \ldots, L_i$
		hier: Lauflänge / Kategorie
FFDA		(SOS) start of scan
	0008	Scan-Header Länge
	01	Anzahl Komponenten im Scan
	01	Name des Scans
	0	Wahl der DC-Codetabelle
	0	Wahl der AC-Codetabelle
	003F00	feste Werte für baseline JPEG

4 Die Schnittstelle zur Datenausgabe

Mirko Pügner hat in seiner Studienarbeit [3] eine Schnittstelle in der FPGA realisiert, mit deren Hilfe man Daten über die FireWire-Schnittstelle ausgeben kann. Dadurch ist es nur nötig, die Daten an diese Schnittstelle mit dem geforderten Protokoll zu übergeben. Jede weitere erforderliche Bearbeitung für

die Ausgabe zu dem DSP auf dem Domonstrator-Board, der die FireWire-Schnittstelle steuert, wird durch die Arbeit von Mirko Pügner realisiert.

Das Datenprotokoll von Mirko Pügner ist in Abbildung 3, S. 12 abgebildet.

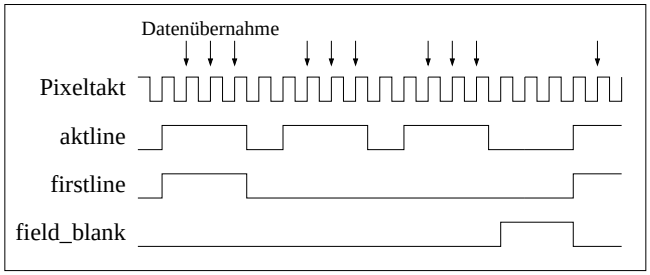


Abbildung 3: Datenprotokoll für Pügners Schnittstelle (Quelle: [3])

Dabei sei darauf hingewiesen, dass es von hoher Bedeutung ist, das Signal „aktline“ einen Taktzyklus vor der dem Datenwort zur steigenden Flanke des Taktsignals high und zur fallenden Flanke des selben Taktes des letzten Datenworts low zu setzen.

Diese Schnittstelle wurde entwickelt, um Daten von Bildsensoren direkt auszugeben. Da viele Bildsensoren das Video-Bildformat unterstützen, ist diese Schnittstelle darauf angepasst. Das bedeuet im speziellen, dass eine Pixelzeile am aktiven „aktline“-Signal und ein Zeilensprung (vertikale Austastlücke) am inaktiven „aktline“-Signal erkannt wird. Das Signal „firstline“ markiert die erste Zeile eines Bildes, „field_blank“ die horizontale Austastlücke. Details befinden sich in Mirko Pügners Arbeit [3].

Die letztgenannten beiden Signale lösen in der Ausgabestufe verschiedene Zustandsübergänge von state machines aus. Auf so erreichte Zustände kann man sich von außen, von einem PC aus über die FireWire-Schnittstelle triggern, um Bildanfang- und Ende am PC zu synchronisieren. Diese Triggerung wurde durch Jens-Uwe Schlüßler realisiert [5].

Da der JPEG-Algorithmus keine Daten produziert, die vergleichbar wären mit dem Video-Bildformat, sondern sogar noch variable Bildlängen (je nach Komprimierbarkeit des Bildrohmaterials) ausgibt, müssen die speziellen Eigenheiten der Schnittstelle bzgl. des Video-Formates umgangen werden. Um aber die Schnittstelle möglichst unverändert nutzen zu können wurden folgende Festlegungen getroffen:

- Wird der erste JPEG-Header ausgegeben, ist „firstline“ aktiv.
- Vor der Ausgabedes ersten Headers ist „field_blank“ aktiv.
- Die Signale „firstline“ und „field_blank“ werden später nicht mehr gesetzt.
- Daten werden ausgegeben, wenn der Ausgangspuffer ausreichend gefüllt ist. Damit wird „aktline“ variabel lang sein.
- Die Austastlücken werden variable Länge haben.

Die Erkennung von Bildanfang und -ende sollte also bei JPEG-komprimierten Daten ausschliesslich durch die speziell dafür vorgesehenen Marker im Header

geschehen. Das setzen der Signale „firstline“ und „field_blank“ dient nur dem korrekten Start der state machines in Mirko Pügners Ausgabefunktionsblock.

Etwas anders sieht es aus, wenn die Bilddaten ohne die JPEG-Quantisierung und -Codierung ausgegeben werden sollen, also wenn ein anderer Filterkernel auf das Bild gelegt wird. Dann existiert ein statischer Datenstrom, der gut mit dem Datenprotokoll der Schnittstelle synchronisiert werden kann.

5 Die Realisierung im Detail

5.1 Die DCT auf dem VISP2000

Wie in Kapitel 3.1 gezeigt wurde, kann die DCT auf dem VISP2000 realisiert werden. Da die DCT-transformierten Bilddaten später in Zig-Zag-Ordnung vorliegen müsen, kann man die Ansteuerung des VISP2000 gleich so gestalten, dass genau diese Reihenfolge bei der Ausgabe der Daten eingehalten wird. Parallel zu dieser Studienarbeit lief die Studienarbeit von Matthias Steidl [9], in der eine allgemeine Ansteuerung des VISP2000 realisiert wurde. Um im speziellen die DCT auf dem VISP auszuführen, sind Ansteuerungskoeffizienten nötig, wie sie in Kapitel 9.2 mit Hilfe eines C++ Programmes berechnet wurden.

5.2 Die Ausgabe des VISP2000

Der Bildsensor VISP2000 hat nicht die Fähigkeit seine Ausgabedaten direkt als Binärzahl auszugeben. Stattdessen werden die Ausgänge seines AD-Converters direkt nach außen geführt. Das hat den Vorteil, dass man sehr leicht testen kann, ob z.B. ein Komparator des AD-Converters nicht funktioniert. Die Wandlung der Komparatorergebnisse zu Binärzahlen muss außerhalb des VISP2000 erfolgen.

Um den Fehler der Bildwerte des VISP2000 niedrig zu halten, wurde beim Design vorgesehen, an den Ausgängen der AD-Converter - Komparatoren eine (externe) RSD-Korektur nachzuschalten. Das Prinzip dieser RSD-Korrektur ist beschrieben in [10]. Diese RSD-Korrektur musste nun auf der FPGA in Hardware realisiert werden.

Weiterhin ist es wichtig, das Schema der Ausgabe des VISP2000 zu kennen und die folgenden Stufen darauf anzupassen.

Die Synchronisation wird durch ein Pixeltakt-Signal gewährleistet, welches von Matthias Steidls Ansteuerung gleichzeitig den AD-Converter im VISP2000 und meine Eingangsstufe (mit RSD-Korrektur) ansteuert. Mit jedem dieser Pixeltakt-Signale kann ein gültiges Komparatorenergebnis vom VISP2000 zur weiteren Verarbeitung übernommen werden. Die ersten 5 so übernommenen Werte nach einem Neustart oder Reset werden verworfen, da der VISP2000 zu diesem Zeitpunkt noch keine sinnvollen Daten liefert. Der Grund ist in der Pipeline-Architektur des VISP2000 zu suchen: Die Pixelströme, welche über einen analogen FIFO und in den gepipelineten AD-Converter geleitet werden, sind noch nicht (digitalisiert) am Ausgang angekommen. Die Pipeline ist noch nicht eingeschwungen.

Der VISP2000 hat eine weitere Besonderheit: Dieser Chip gibt von seinen acht 8×8-Blöcken in einer Zeile jeweils alle ersten (‚zweiten, dritten...) Pixel aus. Abbildung 4 auf Seite 14 deutet dieses Schema an. Welches Pixel allerdings das „erste“ („zweite“, „dritte“...) in einem 8×8-Block ist, bestimmen die von

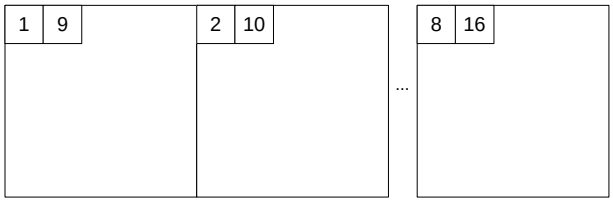


Abbildung 4: Ausgabeschema des VISP2000

der Ansteuerung des VISP2000 angelegten Koeffizienten. Hier in der Abbildung entspricht die Reihenfolge einem Zeilenscan oder Zig-Zag-Verfahren.

5.2.1 Der ADC des VISP2000

Der AD-Converter des VISP2000 arbeitet nach dem Pipelineprinzip. Es sind 10 ADC-Stufen vorhanden, wie in Bild 5 illustriert. Jede dieser 10 Stufen besitzt 4

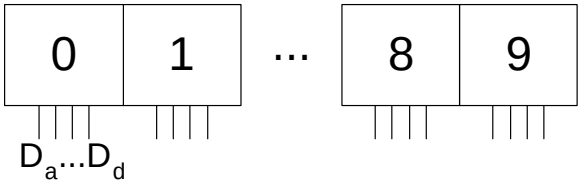


Abbildung 5: ADC des VISP

Ausgänge $D_a \ldots D_d$. Diese 4 Ausgänge repräsentieren 3 Entscheidungsschwellen innerhalb einer ADC-Stufe und sind mit einer Art „Temperaturcode“ codiert. Es sind jeweils immer nur 2 Stufen des ADC aktiv: die erste und die fünfte, die zweite und die sechste usw.. Damit stehen am Ausgang des ViSP2000 mit jedem Takt 2×4 Komparatorergebnisse bereit.

5.2.2 Das Arbeitsprinzip der RSD-Korrektur

Die RSD-Korrektur wird ebenfalls in 10 Stufen berechnet. Es sind immer die selben 2 Stufen aktiv, die auch bei den ADC-Stufen aktiv sind. Dabei addiert jede Stufe den um eine Bitposition nach links geshifteten Digitalwert (also den doppelten Wert) der vorherigen Stufe zu ihrem direkten Input hinzu. Die erste Stufe gibt nur ihren Input weiter. Siehe dazu Grafik 6 auf S. 14.

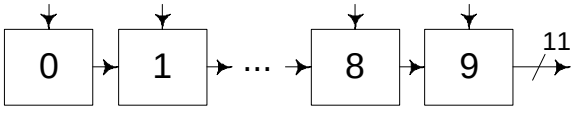


Abbildung 6: Arbeitsprinzip der RSD-Korrektur

Die Inputs der Korrektur-Stufen sind die Outputs der ADC-Stufen, welche zu einer Binärzahl gewandelt worden sind. Dabei wurde aus dem „Temperaturcode“ ein Binärwert mit dem Wertebereich $[-2; 2]$ erzeugt.

5.2.3 Die Realisierung der RSD-Korrektur

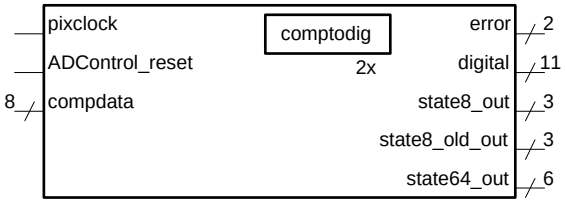


Abbildung 7: rsd.vhdl

Basierend auf der Arbeit und mit Hilfe eine Programmes in C von Jörg Schreiter wurde die Realisierung der RSD-Korrektur entwickelt. Die Realisierung stellt eine direkte Umsetzung von [10] dar.

Zur Wandlung der Comparatorergebnisse in Binärzahlen wie in Kapitel 5.2.2 angekündigt, existiert ein eigenes Modul - comptodig.vhdl, welches in dem RSD-Modul instanziiert wird. Siehe dazu Kapitel 9.3

In der Realisierung der RSD-Korrektur werden dann zusätzliche Informationen für die weiter verarbeitenden Stufen mit erzeugt. Es handelt sich um die Signale „state8“, welches anzeigt, zu welchem der 8 Blöcke in einer Reihe der aktuelle Bildwert zuzuordnen ist und um „state64“, was die Pixelposition innerhalb eines 8×8-Blockes repräsentiert. Siehe dazu auch Abbildung 4 auf Seite 14. Wie man erkennen kann, durchläuft also „state8“ einen kompletten Zyklus innerhalb eines steps von „state64“. Hat „state64“ einen kompletten Zyklus durchlaufen, so ist eine Zeile aus acht 8×8-Blöcken komplett. Der VISP2000 besitzt 16 dieser Zeilen.

Um spätere Verarbeitungsschirtte einfacher zu gestalten, wurde zudem das Signal „state8“ um einen Pixeltakt verzögert und als „state8_old“ ausgegeben.

Die Realisierung der RSD-Korrektur ist im Listung rsd.vhdl nachzulesen, welches in Kapitel 9.4 zu finden ist.

5.3 Die Anpassung des Wertebereichs

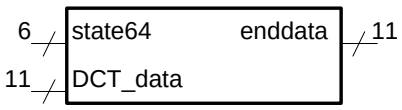


Abbildung 8: minus128.vhdl

In Kapitel 3.1 wurde erläutert, dass für JPEG eine Verschiebung der Pixelwerte definiert ist, die aber vom VISP2000 nicht berechnet werden kann. Diese Verschiebung hat nur Auswirkungen auf den Gleichanteil, also können sämtliche Wechselanteile im folgenden unbeachtet bleiben.

Der Gleichanteil beim Output des VISP2000 wird stets positiv sein ($[0, 1024]$). Er muss verdoppelt werden, um seinem eigentlichen Wertebereich $[0, 2048]$ zu entsprechen. Danach ist der Wert 1024 von ihm zu subtrahieren (siehe Gleichung (7)). Durch diese Operation überstreicht der Wertebereich des Ergebnisses

den vollen gewünschte Wertebereich $[-1023, 1024]$, allerdings mit dem Nachteil, nur die halbe Auflösung zu besitzen. Die Abbildung 9 auf S. 16 illustriert die Problematik noch einmal. Das Listing minus128.vhdl (siehe Kapitel 9.5) stellt

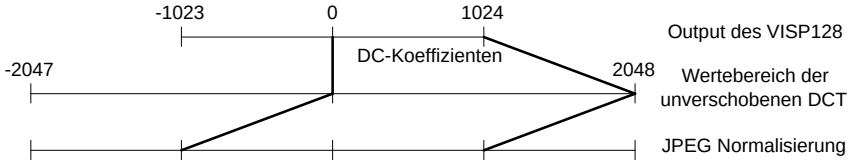


Abbildung 9: Wertebereichsanpassung

eine Realisierung dieser Anpassung des Wertebereichs dar.

Zusätzlich zu der eigentlichen Aufgabe dieses Funktionsblockes existiert noch eine Absicherung gegen negative DC-Koeffizienten, wie sie theoretisch nicht auftreten dürften. Falls dennoch aufgrund von Messfehlern z.B. durch Rauschen negative DC-Koeffizienten entstehen, so werden diese zu Null gesetzt.

5.4 Quantisierung

Die Quantisierung entspricht, wie in Kapitel 3.2 erläutert einer Division. Da eine Division aber in Hardware sehr aufwändig zu realisieren gewesen wäre, wurde die Entscheidung getroffen, die Qunatisierungskoeffizienten einmalig festzulegen und statt zu dividieren mit ihren Inversen zu multiplizieren. Die Koeffizienten wurde aus [6] übernommen. Ihre Inversen wurden per Hand berechnet.

Die nun im folgenden vorgestellte Multiplikation als Ersatz für die Division basiert auf der einfachen schriftlichen Multiplikation aus der Grundschule in Verbindung mit einem Wallace-Tree-Multiplizierer. Die Multiplikation ist nicht getaktet, sondern rein kombinatorisch. Damit sind zwar keine so großen Taktgeschwindigkeiten möglich, wie nach einer Partitionierung und somit einem Aufbau einer Pipeline, aber es genügt den Ansprüchen vollauf und ist weniger komplex zu realisieren.

Alle Koeffizienten in [6] sind kleiner als 128. Um jede Inverse eines Koeffizienten von den anderen in binärer Darstellung noch unterscheiden zu können ist eine Genauigkeit der Inversen bis 2^{-14} nötig. Das begründet sich wie folgt:

$$\frac{1}{127} - \frac{1}{128} = 6,1516 * 10^{-5} \tag{9}$$

Um diese Differenz aus (9) darstellen zu können, ist mindestens Bitposition 2^{-14} nötig. In Anbetracht möglicher späterer Änderungen wurde hier bis 2^{-15} die Binärdarstellung der Inversen berechnet.

Anmerkung: Die Binärdarstellung der inversen Divisoren im Listing entspricht keinem genormtem Zahlenformat. Das MSB des Divisor-Vektors stellt 2^0 dar, das LSB 2^{-15}.

Das Prinzip der Multiplikation veranschaulicht sich am besten duch ein (völlig frei gewähltes) Beispiel. Siehe dazu Abbildung 10 auf Seite 17. Dadurch ergeben sich folgende Erkenntnisse:

- Das letzte Bit des Ergenisses ist unnötig, da bei seiner Berechnung kein Carry erzeugt wurde.

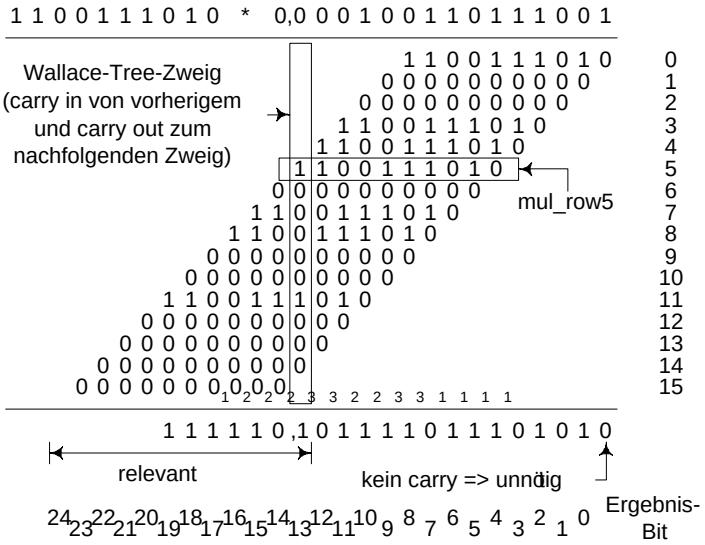


Abbildung 10: Prinzip der Multiplikation

- Alle anderen Bits des Ergebnisses haben entweder direkten Einfluss auf das Ergebnis oder einen indirekten Einfluss durch die Erzeugung von Carrys bei ihrer Berechnung.
- Die Ergebnis-Bits 13 bis 24 repräsentieren das gewünschte Ergebnis. Bit 13 wird für eine Rundung des ganzzahligen Resultates der Multiplikation eingesetzt.
- Die 4 höchstwertigsten Bits des Divisors sind bei den Koeffizienten aus [6] stets Null. Sie werden hier trotzdem berücksichtigt, um diesen Funktionsblock flexibler später einmal weiterverwenden zu können. Dadurch kann der inverse Divisor ≤ 1 sein.
- Das MSB, was das Gewicht 2^0 besitzt, darf nur 1 sein, wenn alle anderen Bits 0 sind. Somit ist der Divisor gleich 1. Kleinere Divisoren würden zu einer Vergrößereung des Wertebereiches der Daten führen, was nicht zulässig ist.

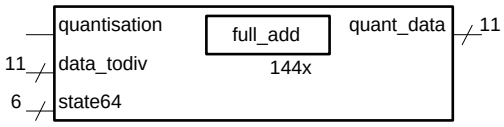


Abbildung 11: division.vhdl

Die konkrete Realisierung des Wallace-Tree-Multiplizierers läßt sich am besten mit Hilfe von Grafiken veranschaulichen. Siehe dazu Abbildung 12, Seite 18 und Abbildung 13, Seite 19.

Jeder Zweig entspricht einer Spalte aus Abbildung 10, S.17. Die Inputs für einen Zweig sind demnach diejenigen Werte, welche durch die Bezeichner

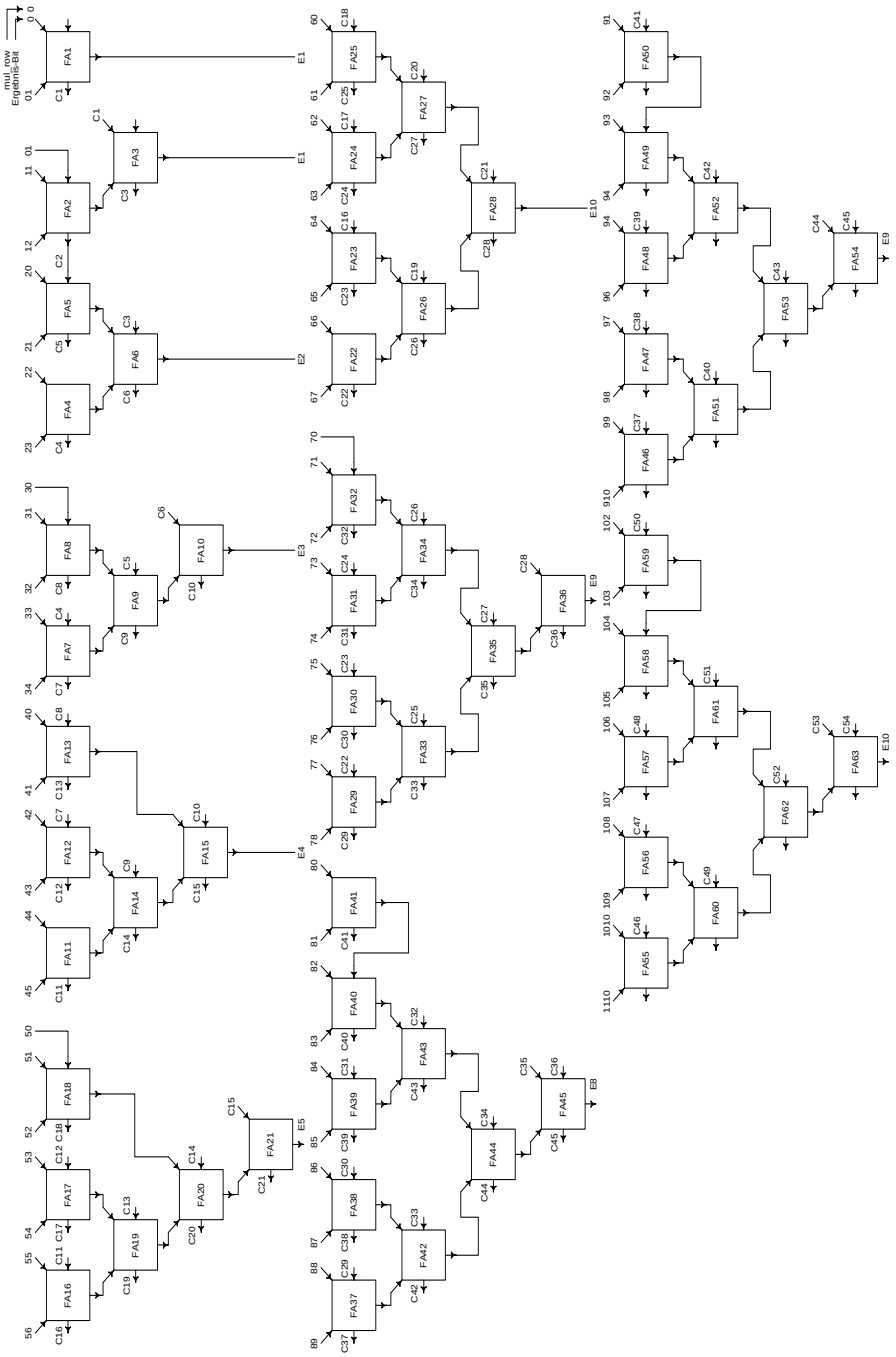


Abbildung 12: Wallace-Tree-Multiplizierer, Teil 1

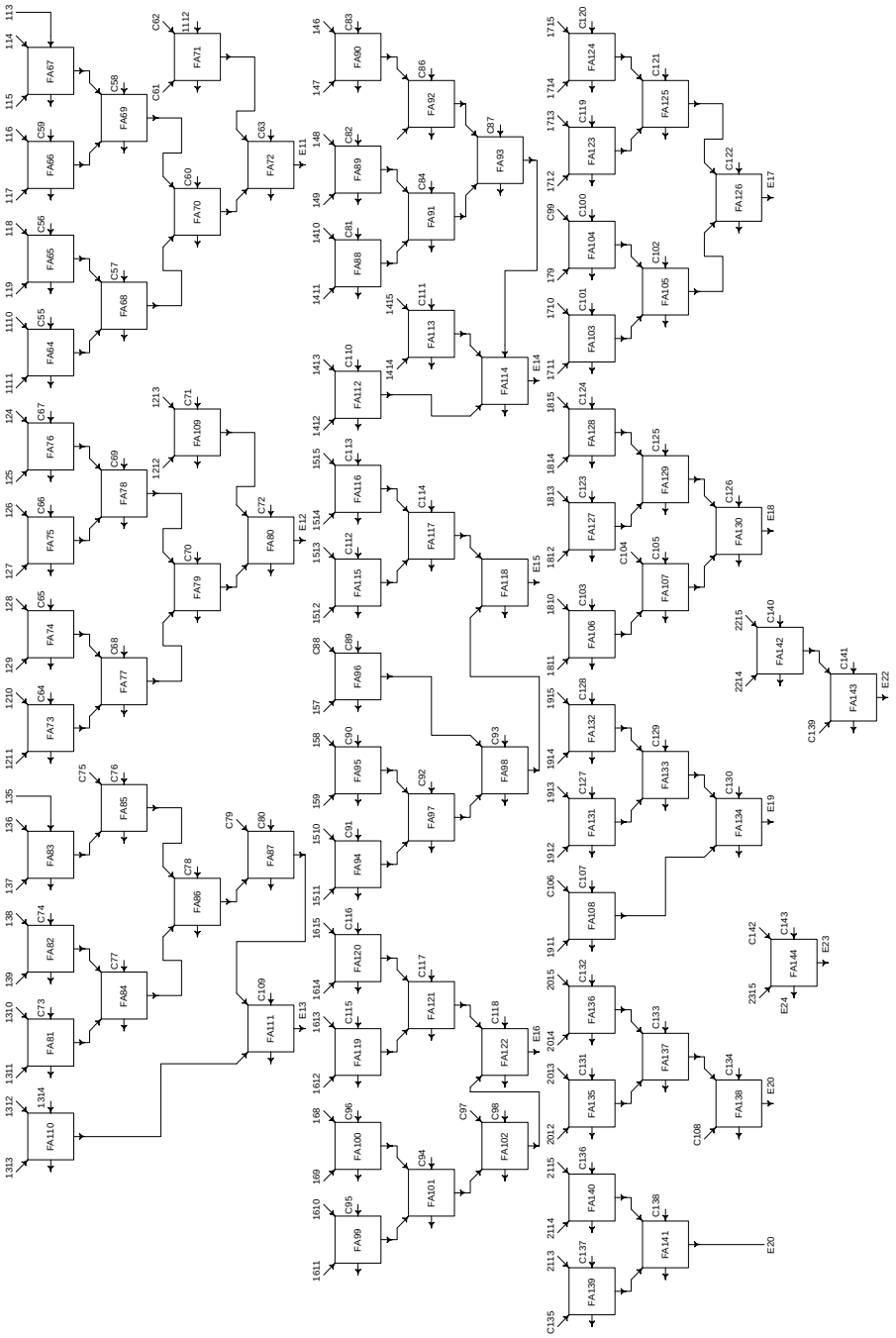


Abbildung 13: Wallace-Tree-Multiplizierer, Teil 2

mul_row-Nummer und Ergebnis-Bit-Nummer deklariert werden. Diese beiden Werte stellen eine Art Koordinatensystem über die einzelnen Werte in den mul_row-Reihen dar.

Bei genauerem Studium des Wallace-Tree-Multiplizierers wird man feststellen, dass alle Volladder nach dem Volladder mit der laufenden Nummer FA108 ein wenig die Symmetrie der vorherigen Aufbaustruktur verletzen. Das resultiert daraus, dass diese Volladder erst nachträglich hinzugefügt wurden, als ich mich entschlossen hatte, die 4 höchstwertigsten Bits des Divisors, welche in dem Fall der Koeffizienten aus [6] immer Null sind, für spätere Erweiterungen doch mit einzuarbeiten.

Das zugehörige Listing division.vhdl befindet sich in Kapitel 9.6.

5.5 Zusammenfassung der ersten 3 Stufen

Um eine bessere Übersicht zu erreichen und um potentielle Fehlerquellen ausschließen zu können, wurden die ersten 3 hier vorgestellten Funktionsblöcke (Kapitel 5.2 bis 5.4) zu einem Baublock zusammengefasst.

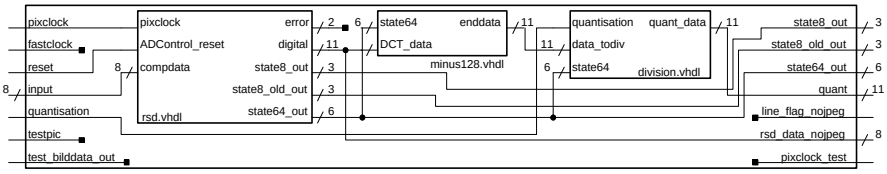


Abbildung 14: comb3.vhdl

Des weiteren hat dieser Funktionsblock die Aufgabe, Testdaten oder die Rohdaten der RSD-Korrektur direkt auszugeben, wenn dies gewünscht ist.

Als Testdaten existiert das Konstanten-Array „bildarray“. Es repräsentiert die 64 Werte eines DCT-transformierten 8×8-Bildblockes. Will man die Arbeitsweise der nachfolgenden Huffman-Codierung testen, so kann man diese 64 Testwerte statt der real durch die RSD-Korrektur berechneten weitergeben. Dazu muss das Signal „testpic“=1 sein. Dieses Signal ist nach außen an den Jumper AG31 auf dem Demonstrator-Board herausgeführt. (Der geschlossene Jumper führt zu einem high-Signal.)

Alternativ kann man die Testdaten auch ohne die Codierung ausgeben, um z.B. die Funktion des FireWire-Interface zu testen. Für die Auswahl steht das Signal „test_bilddata_out“ zur Verfügung, welches durch Jumper AF31 nach außen geführt wird. Dadurch werden zwar nur die reinen Bitwerte des DCT-transformierten Testbildes ausgegeben, aber diese sind wie jedes andere Bitmuster zum Test der Ausgabe ebenso geeignet.

Will man dagegen komplett auf die JPEG-Kompression verzichten, um z.B. mit Hilfe des VISP2000 einen Filterkern auf das Bild zu anzuwenden, so kann man die RSD-korrigierten Ergebnisse auch direkt zur FireWire-Schnittstelle senden. Jumper AF30, repräsentiert durch das Signal „nojpeg“ (in dem später noch näher betrachtetem Listung jpeg.vhdl (Details folgen in Kapitel 5.7.)) in Verbindung mit Jumper AF31 („test_bilddata_out“) führen zu diesem Ergebnis.

Die Aufgabe, gewisse Daten direkt ohne die Huffman-Codierung auszugeben, zwingt dazu, das Protokoll der Schnittstelle von Mirko Pügner (siehe Abbildung 3 auf S. 12) auch in diesem Funktionsblock zu implementieren. Dabei

ergibt sich folgendes Problem: Die RSD-korrigierten Daten liegen mit 11 Bit Genauigkeit vor, die Schnittstelle zu Pügners Ausgabefunktionsblock ist aber nur 8 Bit breit. Da die RSD-korrigierten Daten aber nur zu jedem 5. Pixel-Takt-Zyklus anliegen, stellt es kein Problem dar, die 11 Bit aufzuspalten und in 2 Schritten auszugeben. Beim 2. Ausgabeschritt werden einfach die höherwertigsten Bits mit Nullen aufgefüllt. (Diese Festlegung muss natürlich eine PC-Software, die die Daten verarbeiten will, berücksichtigen.)

Um die Aufspaltung zu erreichen, muss man aber einen modifizierten Pixeltakt (pixclock_test) bereit stellen. Zusätzlich müssen dann noch die Signalisierungssignale, die Pügners Schnittstelle benötigt, erzeugt werden. Dies alles wird in der konkreten Realisierung, die sich als Listing in Kapitel 9.7 befindet mit den Prozessen rsd_data_to_out realisiert. Das Taktschema, dass diesen Prozessen zugrunde liegt ist folgendes:

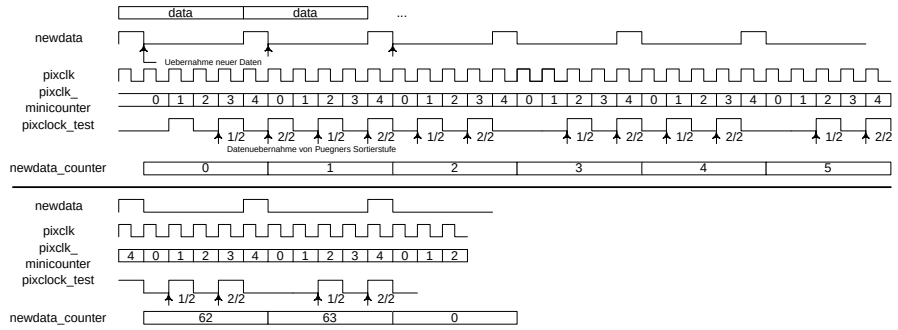


Abbildung 15: Taktschema für die direkte Ausgabe der RSD-korrigierten Daten

Durch das hier verwendete Anhalten des Signals pixclock_test ist es möglich, das von Mirko Pügner geforderte Signal „aktline“ über alle 64 Bildwerte auf high zu belassen.

Zählt man die Flanken von „aktline“, so kann man die geforderten Signale „firstline“ und „field_blank“ ebenfalls zu den richtigen Zeiten setzen. Hier geschieht das Setzen nach 128 Datenblöcken, also einem kompletten Bild des VISP2000, dessen Auflösung ja bekanntermaßen 64×128 beträgt. Diese Zählung wird im Listung jpeg.vhdl vorgenommen, welches in Kapitel 5.7 näher vorgestellt wird. Dort werden auch „firstline“ und „field_blank“ gesetzt.

5.6 Huffman-Codierung

Die Codierung wurde unterteilt in zwei Baugruppen. Die erste Baugruppe hat die Aufgabe aus dem eigehenden Datenstrom wichtige Informationen zu extrahieren, um die entsprechenden Codeworte zu erstellen. Es handelt sich um die Informationen „run“ und „category“, deren Bedeutung in Kapitel 3.3 erläutert wurde. Für diese Extraktion benötigt die Baugruppe einen Datenpuffer.

Mit Hilfe der extrahierten Informationen kann man danach das exakte Codewort zusammenstellen. Da die erzeugten Codewörter immer unterschiedlich lang sind und außerdem erst eine genügende Anzahl von Codewörtern zusammen gekommen sein muss, bis eine Datenausgabe in Richtung Pügners Schnittstelle sinnvoll wird, wurde in der zweiten Baugruppe zusätlich zu der Funktion der

Erstellung der Codewörter ein Ausgangspuffer realisiert, der diese geforderten Eigenschaften unterstützt. Weiterhin wird der gewünschte Header erstellt.

5.6.1 Erste Pufferung und Vorcodierung

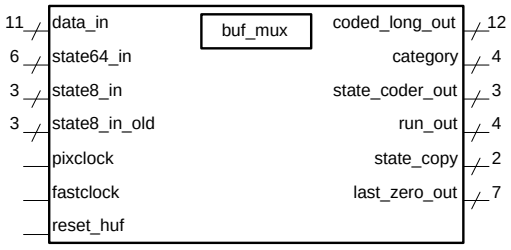


Abbildung 16: data_buf.vhdl

Durch die spezifischen Eigenheiten bei der Ausgabe des VISP2000 (siehe 5.2 und im speziellen Abbildung 4, S. 14) ist es nötig, mindestens eine ganze Zeile von 8×8-Blöcken eines Bildes zu puffern, um danach jeden 8×8-Block gesondert betrachten und codieren zu können. Da aber gleich nachdem ein 8×8-Block-Puffer gefüllt ist, schon der nächste Block aus der nächsten Zeile übertragen wird, muss ein zweiter Puffer von der selben Größe zusätzlich verwendet werden. Damit muss der Puffer also Platz für 2 Bildreihen, bestehend aus jeweils acht 8×8-Blöcken bieten. Bei einer Bitbreite von 11 Bit ergibt das $11 \times (8 \times (8 \times 8)) \times 2 = 11264$ Bit.

Abbildung 17, S. 22 illustriert das Puffer-Prinzip. In die eine Pufferreihe wer-

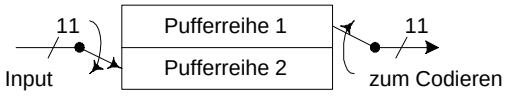


Abbildung 17: Prinzip der Pufferung in data_buf.vhdl

den immer die Input-Daten eingeschrieben, aus der anderen werden die Daten ausgelesen und codiert. Die Codierung einer Bildreihe muss also abgeschlossen sein, wenn die Daten einer 3. Reihe am Eingang anliegen. Ist dies der Fall, so werden die Pufferreihen getauscht und die alten inzwischen fertig codierten Daten überschrieben.

Eine spezielle Sortierung der Daten muss nicht vorgenommen werden, da durch die sinnvoll gewählten Ansteuerungskoeffizienten (siehe Kapitel 5.1) die Reihenfolge der Daten schon der gewünschten Zig-Zag-Ordnung entspricht.

Im Zuge des Einschreibens bietet es sich an, von einem 8×8-Block die Postition des jeweils letzten nicht-Null-Wertes zu speichern, damit später ein end of block (EOB) einfacher codiert werden kann.

Die Regeln der Codierung wurden in Kapitel 3.3 vorgestellt. Durch die Differenzcodierung des Gleichanteils muss dieser extra gepuffert werden, da sonst beim Reihenwechsel der jeweils vorherige Gleichanteil möglicherweise schon mit dem Gleichanteil aus der nächsten Reihe überschrieben werden könnte.

All das führt zu der Idee, den Puffer mittels eines Dual-Port-RAM zu beschreiben. Solcherart RAM ist auf der FPGA in variabler Konfiguration als Block-RAM vorhanden. Damit stellt das Einschreiben der Daten kein Problem dar. Lediglich die korrekte Adresse muss stets übergeben werden. Etwas komplizierter ist ein sinnvolles Auslesen, um möglichst effizient codieren zu könnnen.

Das Lesen von einem RAM in der FPGA wird prinzipiell nach einem Schema ablaufen, wie es in Abbildung 18 illustriert ist. Dabei wurde festgelegt, dass der RAM ebenso wie die ihn ansteuernde Logik mit der positiven Taktflanke aktiv wird. Eine Realisierung mittels zweiphasiger Ansteuerung (Logik mit der

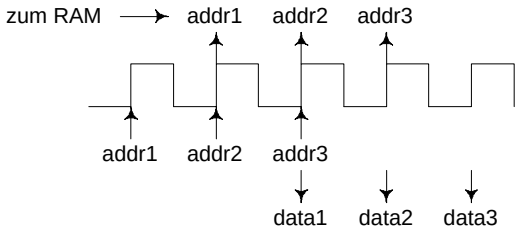


Abbildung 18: generelles Taktschema für Block-RAM

steigenden und RAM mit der fallenden Taktflanke aktiv) ist angünstig, da die Gatterlaufzeiten zu lang sind, um hohe Taktfrequenzen zu erreichen. (Eine solche zweiphasige Ansteuerung würde effektiv dem doppelten Takt entsprechen.)

Auf jeden Fall wird ein schnellerer Takt, als der langsame Pixeltakt benötigt, um die Codierung in der erforderlichen Zeit zu schaffen. Als Vorgabe wurde genommen, dass dieser Takt „fastclock“ um den Faktor 10 höher sein soll, als der Pixeltakt („pixclock“). Ein höherer Faktor würde zwar die Realisierung vereinfachen, aber bei der Synthese später Probleme bereiten, da Takte über 50 MHz nur sehr aufwändig auf der FPGA zu erreichen sind.

Hat man einen geeigneten schnellen Takt, so kann man mit jedem Taktschritt ein Datum auslesen (nach dem Schema aus Abbildung 18). Die Wechselanteile werden bezüglich ihrer Ordnung in der Zig-Zag-Reihenfolge nacheinander ausgegeben. Dabei kann man die auftretenden Nullen zählen („run“) und nach dem Auftreten des nächsten nicht-Null-Wertes die Information zusammenstellen, die für die engültige Erstellung des codewortes nötig sind („category“).

Die Realisierung befindet sich als Listing in Kapitel 9.8. Im folgenden soll jetzt auf einige Details eingegangen werden:

Beim Einschreiben in den RAM ist es nötig, die Daten immer an die richtige Position zu schreiben. Da immer die 8 korrespondierenden Werte einer Reihe nacheinander am Eingang anliegen, müssen diese auch an die korrekte Position geschrieben werden. Dazu werden die Informationen „state8“ und „state64“ verwendet, die die Zugehörigkeit der Daten exakt beschreiben. Diese Informationen steuern Adress-Offsets (siehe Prozess „data_into_block“ in data_buf.vhdl).

Für das Auslesen aus dem RAM und die Codierung wurde eine state-machine entworfen, die in Abbildung 19 illustriert ist. Aus dem Zustand „NOP“ wird herausgesprungen, wenn ein 8×8-Block codiert werden kann. Im Folgestate „init“ wird die DC-Differenz (zum vorherigen DC-Wert) berechnet und die korrekte

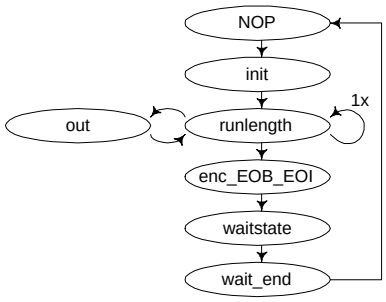


Abbildung 19: state machine für die Codierung

erste Adresse für den ersten AC-Wert an den RAM angelegt, damit beim Start des nächsten states schon ein gültiges Datum vorliegt. Danach, im state „runlength“ angekommen, bleibt die state machine dort, wenn der 8×8-Block noch nicht vollständig codiert ist.

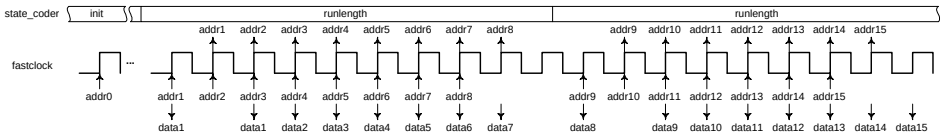


Abbildung 20: Taktschema beim RAM-Lesezugriff in data_buf.vhdl

Im state „runlength“ wird nun mit jedem fastclock-Zyklus ein Datenwort vom RAM angefordert und/oder ein erhaltenes Datenwort überprüft, ob es nicht Null ist (siehe Taktschema in Abbildung 20, S. 24). Ist das Datenwort nicht Null, so konnte man aus den nötigen Zyklen bestimmen, wie lang der „run“ der Nullen vor dem nicht-Null-Wert war. Zur Codierung des nächsten Codewortes sind also maximal 16 aufeinanderfolgende Datenworte nötig (nämlich dann, wenn eine Folge aus 15 Nullen auftritt). Da bei einem 10-fach schnellerem fastclock gegenüber dem pixclock keine 16 Zyklen abgearbeitet werden können, ein Statewechsel aber an pixclock gekoppelt ist, muss sich „runlength“ über zwei pixclock-Zyklen erstrecken.

In Abbildung 20 werden aber nur 9 Taktzyklen genutzt, um Daten aus dem RAM zu lesen und zu verarbeiten. Das hat folgende Bewandnis: Prinzipiell ist der Takt „fastclock“ der primäre Takt auf der FPGA. Matthias Steidl erzeugt in seiner VISP2000-Ansteuerung [9] unter anderem daraus den langsamen Takt „pixclock“. Jede Taktänderung des erzeugten Taktes geschieht mit der steigenden Flanke von fastclock. Nun werden Laufzeiteffekte dazu beitragen, dass der Statewechsel der state machine der Codierung nicht mehr zur steigenden Taktflanke von fastclock geschieht. Dem wurde in Abbildung 20 Rechnung getragen und der 10. fastclock-Zyklus aus Sicherheitsgründen nicht benutzt.

Nach dem (doppelten) Durchlaufen von runlength wird in den Zustand „out“ übergegangen, der die Kombination aus „run“ (der Anzahl der Nullen) und „value“ (dem nächsten nicht-Null-Datenwort) in Richtung Ausgang weiter gibt.

Ist der letzte Wert im 8×8-Block erreicht, oder existieren nur noch Nullen bei den höherweriten Koeffizienten, so wird von der state machine von „runlength“ in „enc_EOB_EOI“ übergegangen.

Das Wertepaar „run“ / „value“ stellt prinzipiell schon den Kern eines Huffman-Codewortes dar. Es muss jetzt nur noch die jeweilige Kombination durch das exakte Codewort ersetzt werden. Die genaue Ersetzung erfolgt in dem Listing huffman.vhdl, welches in Kapitel 5.6.2 beschrieben wird.

Wesentlich bei dieser Ersetzung ist die tatsächliche Größe von „value“, da nur so viele Bits übertragen werden, wie unbedingt nötig sind. (Siehe dazu Kapitel 3.3.) Die sogennate Kategorie wird durch eine untergeordnete Hilfskomponente (buf_mux.vhdl) bestimmt. Diese Hilfskomponente wird ebenfalls in Kapitel 9.8 als Listing vorgestellt.

Die Zustände der state machine für die Codierung können in der nächsten Verabreitungsstufe (huffman.vhdl - Listing befindet sich in Kapitel 9.9) genutzt werden, um zu wissen, wann gültige vorcodierte Daten am Ausgang von data_buf.vhdl anliegen.

Der state „wait_end“ sorgt dafür, dass der Codierungsprozess für den selben 8×8-Block nicht noch einmal beginnt.

5.6.2 Komplettierung der Codierung und Ausgangspufferung

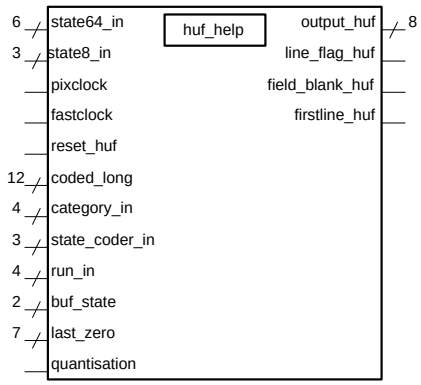


Abbildung 21: huffman.vhdl

Die Daten, die data_buf.vhdl bereit stellt, bestehen aus 3 Teilen: aus dem „run“ der Nullen, dem nächsten nicht-Null-Wert (hier benannt mit „coded_long“ (=„value“ in data_buf.vhdl)) und der „category“ dieses Wertes. Daraus muss ein Codewort abgeleitet werden.

Dies geschieht hier in huffman.vhdl, indem in einer Tabelle nachgesehen wird. Für den Gleichanteil existiert das Konstanten-Array „DC_code_table“, für die Wechelanteile „code_table“. Da auch die Codewörter eine variable Bitlänge haben, in einem Konstanten-Array aber nur Konstanten der selben Bitlänge gespeichert werden können, ist die Anzahl der nötigen Bits in den Arrays „DC_code_table_length1“ sowie „code_table_length“ gespeichert.

Das endgültige codierte Datenwort ergibt sich, wie in Kapitel 3.3 beschreiben aus dem Codewort für das Paar „run“/„category“ plus so viele Bits des Wortes „coded_long“, wie die Kategorie groß ist.

Im folgenden wird nun eine Möglichkeit vorgestellt, die codierten Daten in einem Ausgangspuffer abzulegen und diesen Puffer bei ausreichender Füllung über den Ausgang zu leeren.

Als Grundgedanke bietet sich die Realisierung eines FIFO-Buffers an. In einer Art Stapelspeicher werden Daten abgelegt und die Daten, die am längsten im Puffer verweilen (im Stapel unten liegen), werden entnommen. Den Füllstand zeigt ein Pointer an. Doch diese Idee hat den entscheidenden Nachteil, beim Leeren des Puffers stets alle Daten „nach unten durchrutschen“ zu lassen. Dieses Verschieben der Daten ist bei einem großen Puffer nur relativ kompliziert zu lösen.

Eine Verbesserung dieses Prinzips stellt ein Puffer mit einer Ring-Architecktur dar, wie in Abbildung 22, S. 26 illustriert. Auch hier werden die Daten „ober-

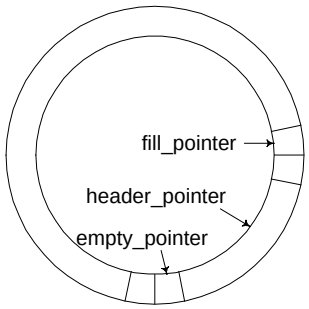


Abbildung 22: Prinzip eines Ring-Puffers

halb“ der vorherigen Daten im Puffer abgelegt, doch bei der Entnahme von Daten wird lediglich ein Pointer, der auf das „unterste“ Datum zeigt, verschoben, welcher zusätzlich zum „oberen“ Pointer existiert. Gefüllt wird der Puffer kontinuierlich „im Kreis“, d.h.: Wird die höchste Puffer-Adresse erreicht, so wird anschliessend wieder bei der niedrigsten Adresse weiter gefüllt.

Einen Pufferüberlauf erkennt man bei einer Ring-Architektur daran, dass der Pointer, der anzeigt, wo die neuesten Daten hingeschrieben wurden (eine Art Füllstandspointer - „fill_pointer“) den anderen Pointer, der anzeigt, wo die ältesten Daten liegen („empty_pointer“) eingeholt hat.

Dieser Ring-Puffer wurde so modelliert, dass jedes Ring-Segment ein Datenwort bestimmter Länge aufnehmen kann. Anschaulich könnte man es sich als Schublade vorstellen. Die feste Länge der Schublade korreliert mit der festen Länge des Wertes aus der Codewort-Tabelle und der festen Länge des Datums „coded_long“. Mann kann das „run“/„category“-Codewort und das Datum „coded_long“ in jeweils ein eigenes Ring-Segment abspeichern. Da „coded_long“ nur maximal 12 Bit, aber das „run“/„category“-Codewort maximal 16 Bit breit ist, werden die unnötigen Bits in einem Segment, was ein Datum „coded_long“ aufnimmt, ignoriert.

Will man zwei Ring-Segmente (Schubladen) gleichzeitig füllen, wie in der hier vorgestellten Realisierung vorgesehen, so muss man zwangsläufig wieder einen Dual-Port-RAM verwenden.

Damit die wichtige Information, wie viele Bits von den abgelegten Werten gültig sind, nicht verfällt, muss man diese Information ebenfalls abspeichern.

Es bietet sich an, dies analog zu dem Ring-Puffer in einem eigenen Puffer zu realisieren. Diese Idee ist illustriert in Abbildung 23, S. 27. Da der Daten-Puffer 16 Bit breit ist, muss der Puffer für die Anzahl der gültigen Bits 4 Bit breit sein.

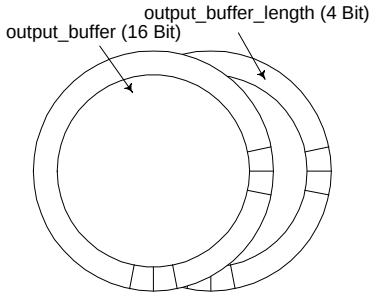


Abbildung 23: Die beiden Ausgangspuffer

Mit Hilfe des Signals „state_coder“, welches den Zustand der state machine für die Codierung in data_buf.vhdl anzeigt, werden die am Eingang anliegenden Daten verarbeitet. Es wird in der entsprechenden Tabelle nachgesehen, welchem endgültigen Codewort die Datenkombination am Eingang entspricht und dieses Codewort auf dem Ring-Puffer abgelegt. Hinzu kommt die Prüfung ob das Datenwort „coded_long“ negativ ist oder nicht. Ist es negativ, wir es dekrementiert und auf dem Puffer abgelegt, ist es positiv wird es direkt auf dem Puffer abgelegt. Bei jeder Ablage von Daten im Puffer werden gleichzeitig deren jeweils gültige Längen in „output_buf_length“ gespeichert.

Weiterhin wird nach dem Ende eines 8×8-Blockes ggf. ein end of block (EOB) gesetzt. (Es wird nicht gesetzt, wenn der 63. AC-Koeffizient nicht Null gewesen ist.) Nach dem Ende eines ganzen Bildes wird der Pointer „herader_pointer“ auf die nächstfolgende „Schublade“ des Ring-Puffers gesetzt. Mit ihm wird angezeigt, dass vor der Ausgabe dieses Datenwortes das Bild mit einem end of imgae (EOI) abgeschlossen und ein neues Bild mit einem neuen Header begonnen werden muss. Es werden dann keine weiteren Daten ausgegeben, sondern nur der Header, bis er komlett ist.

Hier in der Realisierung wurde das EOI-Kommando mit in das Konstanten-Array hinein gezogen, welches den Header symbolisiert. Dadurch wird immer vor dem Beginn eines neuen Headers ein EOI ausgegeben. So wird zwar auch beim allerersten Bild der Datensrom mit einem EOI enden, bevor er mit dem ersten korrekten Header beginnt, aber es sollte für eine Software kein Problem darstellen, dieses erste EOI zu ignorieren, da eine Triggerung auf start of image (SOI) sowieso wichtig erscheint.

Durch die Header-Ausgabe bestimmt sich auch in etwa, wie groß der Ring-Puffer sein muss. Die Erfahrung zeigt, dass ein Header innerhalb der Zeit ausgegeben wird, in der drei 8×8-Blöcke codiert und im Puffer abgelegt werden. Nimmt man den worst case, also den maximalen Datenstrom an, so tritt dieser auf, wenn in den 8×8-Blöcken keine Null enthalten ist. Damit sind für die 64 Werte der Blöcke auch 64 Codeworte nötig. Es sind also 128 Schubladen im Puffer nötig, die gefüllt werden bei einem 8×8-Block (eine Schublade für das Codewort und eine für die zusätzlichen Bits). Der Puffer wurde so in der

Realisierung bemessen, dass er 512 Schubladen (also Platz für vier 8×8-Blöcke im worst case) besitzt. Diese recht knappe Bemessung hat den zwei Gründe: Erstens existieren 20 Blöcke RAM auf der FPGA. 2 Blöcke sind durch Mirko Pügners Schaltung [3] verbraucht, 12 Blöcke durch die Ansteuerung von Matthias Steidl [9] und weitere 2 Blöcke durch data_buf.vhdl. Die beiden Puffer hier in huffman.vhdl erstrecken sich über 4 RAM-Blöcke. Zweitens wurde für die Berechnung des Puffers der worst-case heran genommen, so dass mehr als 512 Schubladen nie nötig sein sollten.

Immer dann, wenn der Block data_buf.vhdl seine Codierungs-Arbeit für einen 8×8-Block beendet hat und somit keine Daten in die Ring-Puffer geschrieben werden müssen, kann eine Ausgabe von Daten an Mirko Pügners Sortierstufe [3] beginnen. Dazu muss der Puffer mindestens ein 8 Bit - Datenwort beinhalten. Weil in einer Schublade nicht immer alle Bits gültig sind und die Anzahl der gültigen Bits in data_buf_length vermerkt ist, muss als erstes geprüft werden, wann garantiert 8 Bit in den Schubladen zusammen kommen. Ein Auslesen der Längeninformation aus data_buf_length würde viel Zeit in Anspruch nehmen, weil erst auf den RAM zugegriffen werden müsste. Es muss also überschlagsmäßig berechnet werde, wann garantiert 8 Bit im Puffer liegen. Dabei tritt der worst case ein, wenn im Puffer eine lange Folge nur von den kürzesten Codewörtern im Alphabet auftritt. Das kürzeste DC-Codewort ist 2 Bit lang. In diesem Fall ist auch die Kategorie 0 und keine weiteren Bits werden angehangen. Bei den AC-Werten ist das kürzeste Codewort ebenfalls 2 Bit lang, aber die Kategorie ist 1, was bedeutet, dass noch ein weiteres Bit hinzu kommt. Folgerichtig ist eine Folge von diesen AC-Codewörtern der worst case, da sowohl für Codewort als auch für das zusätzliche Bit jeweils eine Schublade reserviert ist. Wie die Abbildung 24, S. 28 verdeutlicht, sind also mindestens

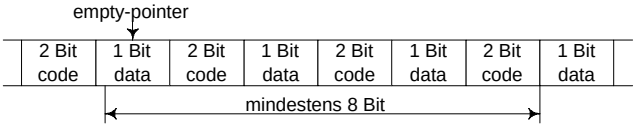


Abbildung 24: worst case im Ring Puffer

6 gefüllte Puffer-Schubladen nötig, um eine Füllung von 8 Bit zu garantieren. Die Differenz von fill_pointer und empty_pointer gibt Auskunft darüber, ob mindestens 6 Schubladen gefüllt sind.

Die Ausgabe von Daten wird dennoch nicht gestartet, wenn 6 Schubladen gefüllt sind, sondern erst, wenn mindestens 5 Datenwörter zu 8 Bit im Puffer liegen (also die Differenz von fill_pointer und empty_pointer 30 beträgt). Das hat den Grund, dass Pügners Sortierstufe bei nur einem einzigen Datenwort Ausgabe fehlerhaft reagiert. Die Wahl von 5 Datenwörtern ist Ergebnis der Schätzung, dass damit weder zu große Puffer-Füllungen noch zu kurze Ausgabe-Bursts auftreten. Im Prinzip ist es aber Willkür und kann in gewissen Maßen frei verändert werden.

Es wird die Ausgabe nun also erst gestartet, wenn ausreichend Datenwörter bereit liegen (oder ein Header ausgegeben werden muss). Die Unterbrechungsbedingung für die Datenausgabe (also der Fall, dass der Puffer nicht genügend voll ist) entspricht der Bedinung für den worst case.

Sämtliche Dinge für die Datenausgabe sind in der Realisierung in Kapitel 9.9 in eine state machine eingearbeitet, die durch das (einzige) case-statement in dem Listing beschrieben ist. Diese state machine ist realativ einfach aufgebaut. Sie startet mit der Datenausgabe (data_to_out), wenn die oben beschriebenen

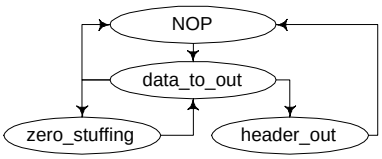


Abbildung 25: state machine der Datenausgabe

Bedinungen eingetreten sind. Dort verweilt sie, bis nicht mehr gewährleistet werden kann, dass 8 Bit im Puffer liegen. Dieser Zustand der Datenausgabe wird unterbrochen, wenn durch die Code- und Datenwörter die 8 Bit - Kombination FF (hex) entstanden ist. Da FF ein Marker ist, ein solcher aber nicht mitten im Datensegment auftauchen darf, muss diesem FF einmal 00 folgen. Dies wird im state „zero_stuffing“ erledigt. Auch unterbrochen werden kann der Zustand, wenn empty_pointer=header_pointer. Dann wird in den state „header_out“ übergegangen und der Header ausgegeben.

Mit Hilfe dieser state machine werden zusätzlich die Signale gesetzt, die Mirko Pügner mit seinem Datenübernahmeprotokoll erwartet.

Zur Ausgabe der Daten ist wieder ein etwas komplizierteres Taktschema notwendig, um nacheinander die Daten der einzelnen Schubladen zu lesen und die gültigen Bits zu einem Datenwort zusammenzufassen. Dieses Schema ist in Abbildung 26, S. 29 illustriert. Es werden also 6 Daten und Längenangaben aus den

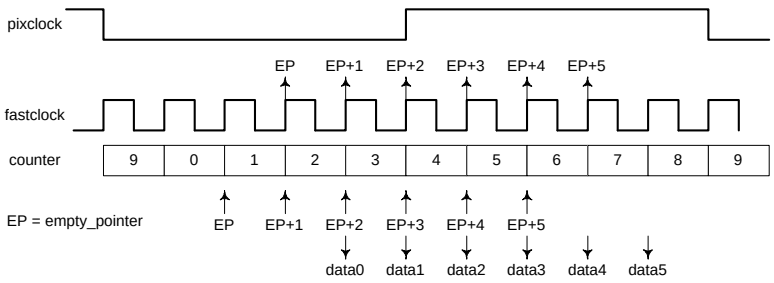


Abbildung 26: Taktschema der Datenausgabe

Schubladen gelesen. In der gewünschten Reihenfolge wird daraus ein 8 Bit Wort erstellt, welches dann zum Ausgang geleitet wird. Dadurch kann es vorkommen, dass eine Schublade nur teilweise geleert wird. Um bei der nächsten Ausgabe eines 8 Bit Wortes genau an der richtigen Stelle weiter zu machen, exisitert ein Pointer, der den Füllstand einer Schublade anzeigt: bufblock_pointer. Dieses Prinzip ist illustreiert in Abbildung 27, S. 30.

Bei der Ausgabe wird der korrekte neue Wert von empty_pointer gesetzt. Sollte es vorkommen, dass empty_pointer=header_pointer, aber das letzte 8 Bit Wort nicht gefüllt ist, so werden die verbleibenden Bits mit Nullen aufgefüllt, bevor mit der Ausgabe des Headers im nächsten Zyklus begonnen wird.

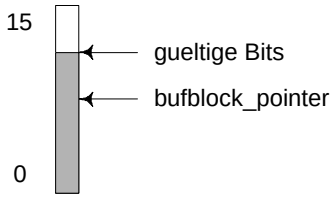


Abbildung 27: Prinzip von bufblock_pointer

5.6.3 Die Zusammenfassung der Huffman-Komponenten

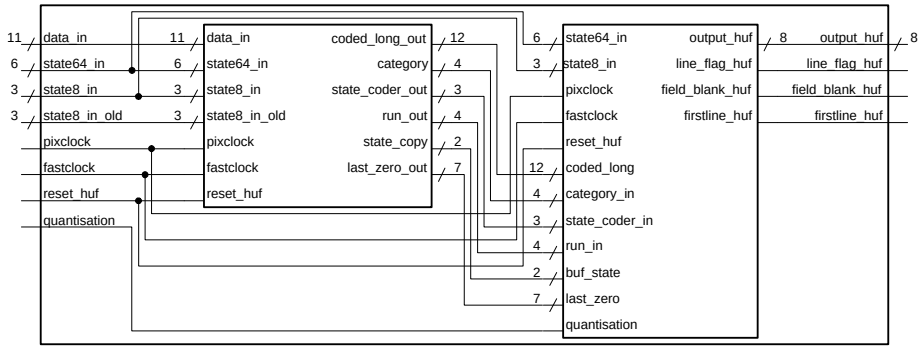


Abbildung 28: huf_comb.vhdl

Um die Baublöcke der Huffman-Codierung zusammen synthetisieren und testen zu können, wurden sie zusammengefasst in huf_comb.vhdl. (Das Listing ist in Kapitel 9.10 abgedruckt).

5.7 Die Zusammenführung aller Baublöcke

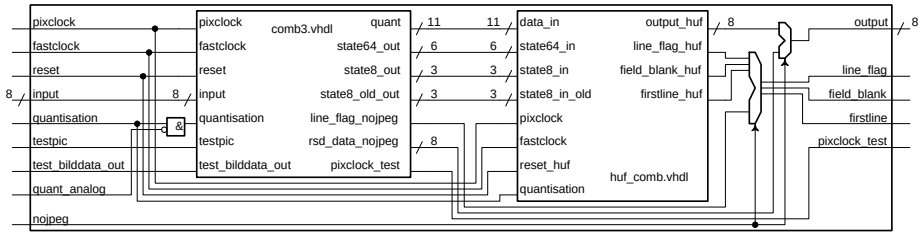


Abbildung 29: jpeg.vhdl

Die nächste Aufgabe besteht darin, alle nun vorhandenen Baublöcke miteinander zu verbinden. Dabei handelt es sich um die RSD-Korrektur, die Wertebereichsanpassung und die Quantisierung, welche mit comb3.vhdl zusammengefasst wurden und die beiden Huffman-Komponenten, die durch huf_comb.vhdl miteinander verbunden sind.

Es exisitiert mit den Signal „nojpeg“ ein Schalter, der es erlaubt, endweder den direkten Output der RSD-Korrektur oder die fertig codierten Bilddaten

auszugeben.

Die Quantisierung wurde mit zweierlei Möglichkeiten zur Abschaltung implementiert: Einerseits existiert das Signal „quantisation“ (Jumper AG30), welches die Quantisierung generell abschaltet. Es wird dann ein Header generiert (in huffman.vhdl), der eine Quantisierungstabelle voller Einsen enthält. Andererseits kann man auch die Quantisierung (in division.vhdl) durch „quant_analog“ (Jumper AF29) deaktivieren. Ein Header mit der Quantisierungstabelle wird dann dennoch generiert (vorausgesetzt „quantisation“=1). Der Grund dafür ist die Absicht, später einmal in die Ansteuerungskoeffizienten des VISP2000 die Quantisierung mit einfließen zu lassen, um auch noch den Multiplizierer einsparen zu können.

6 Das Gesamtprojekt

Der JPEG-Kompressionsalgorithmus ist nur ein Teil des Gesamtprojektes, den Kamerasensor VISP2000 ansteuern und nutzen zu können. Die anderen Teile sind die Ansteuerung des VISP2000 [9] und die Ausgabe von Daten an die FireWire-Schnittstelle [3] zur weiteren Verarbeitung im Computer.

Diese 3 Komponenten müssen zusammen synthetisiert werden und auf der FPGA arbeiten.

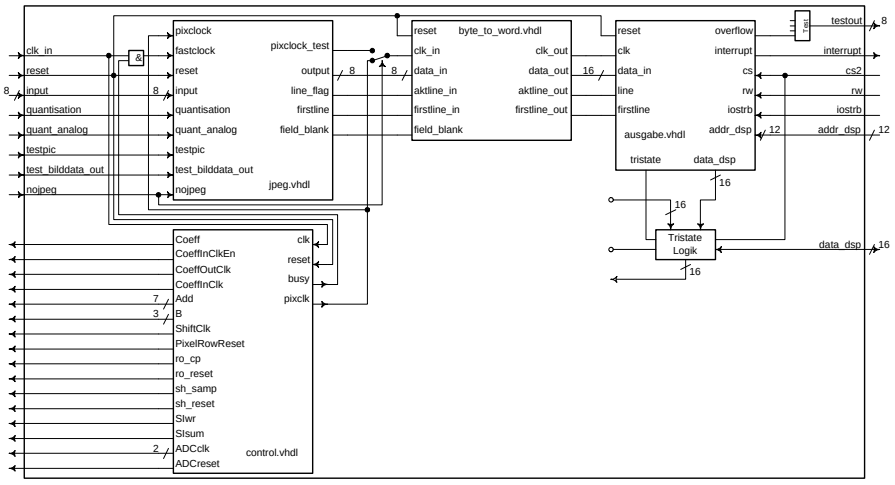


Abbildung 30: top.vhdl

Da die Ansteuerung das Taktverhalten und die Ausgabe des VISP2000 regelt, kommt dieser Komponente zentrale Bedeutung bei der Erzeugung der Takte zu. Primär existiert ein externer Takt (in dieser Studienarbeit immer wieder mit „fastclock“ bezeichnet), der durch einen Schwingquarz erzeugt wird. Aus diesem Takt werden in der Ansteuerung diverse Untertakte abgeleitet, die der VISP2000 benötigt. Unter anderem existiert damit ein Takt für den AD-Converter. Dieser AD-Converter-Takt ist bestimmend für die Datenausgabe und somit auch für die gesamten nachfolgenden Verarbeitungsstufen. Er ist aber ein Takt, dessen low/high-Verhältnis nicht 1:1 ist. Für die JPEG-Komponente wird aber ein solcher Takt benötigt. Um dies zu gewährleisten, wird durch die Ansteue-

rungskomponente zusätzlich ein Takt „pixclock“ bereit gestellt, dessen steigende Flanke synchron mit dem AD-Converter-Takt, aber dessen fallende Flanke nach der Hälfte der Taktperiode auftritt.

Weil der AD-Converter nicht ständig arbeitet, da während der Pixelauslesephase keine gültigen Daten anliegen, wird der Pixeltakt angehalten. Somit ist es noch notwengig den globalen Takt „fastclock“ für die JPEG-Komponente anzuhalten, da nur so gewährleistet werden kann, dass die gesamte JPEG-Komponente „eingefroren“ ist.

Die Ausgabestufe wird durch (teilweise modifizierte) Pixeltaktsignale durch die JPEG-Komponente angesteuert. Dadurch sind keine weiteren Modifikationen nötig, da die Ausgabestufe so durch die angehaltene JPEG-Komponente mit gestoppt wird. Es handelt sich hier zwar prinzipiell um gated clocks, aber da der Pixeltakt ein sehr langsamer Takt ist und die Signale, die mit diesem Takt übergeben werden, die selbe Verzögerung haben, hat dies keine weitere Bedeutung.

Zur besseren Testbarkeit wurden von Jens-Uwe Schlüßler einige Signale zusammengefasst und mit dem Signalvector „testout“ auf Testpins gemappt.

Die Tristate-Logik wurde von Mirko Pügner entwickelt, um Daten einerseits ausgeben und andererseits auch einlesen zu können. Damit ist es möglich, Bilddaten wie auch Statusinformationen von einer Kamerasteuerung auszugeben. Diese Statusinformationsausgabe ist durch Matthias Steidl z.Z. noch nicht implementiert. Ebenso ist die Möglichkeit, Daten von aussen in die Steuerung einzuspeisen noch ungenutzt. Um für die Zukunft diese Möglichkeit aber dennoch offen zu halten, wurde die Tristate-Logik nicht entfernt, sondern permanent auf Bilddatenausgabe gestellt.

7 Simulation und Synthese

Da das Gesamtprojekt noch nicht wie gewünscht arbeitete, soll hier nur die JPEG-Komponente betrachtet werden, um deren Funktion zu verifizieren.

7.1 Simulation

7.1.1 Das reale Bild

Beim Funktionstest des Kamerasensors VISP2000 wurde mit Hilfe eines Testers von Hewlett-Packard ein Bild aufgenommen. Leider lagen diese Bilddaten nicht mehr als Rohdaten, d.h. als Datenstrom des Output des VISP2000, sondern als RSD-korrigierte Daten vor. (Die RSD-Korrektur wurde durch [10] ermöglicht.)

Diese Daten mussten somit hinter der hier vorgestellten RSD-Korrektur (rsd.vhdl) und vor der Wertebereichsanpassung (minus128.vhdl) eingeschleust werden. Somit ist es ausgeschlossen, dieses Testbild nach einer Synthese der gesamten JPEG-Schaltung zu verwenden.

Das Einschleusen der Bilddaten wurde durch eine modifizierte Version von comb3.vhdl realisiert. Immer dann, wenn RSD-korrigierte Daten aus der RSD-Korrektur kamen, wurden sie einfach durch die realen Testbilddaten ersetzt.

Abbildung 31 zeigt das JPEG-komprimierte Bild. Als Quantisierungskoeffizienten wurden die in [6] vorgestellten Koeffizienten verwendet. Die Größe des Bildes beträgt 861 Bytes.

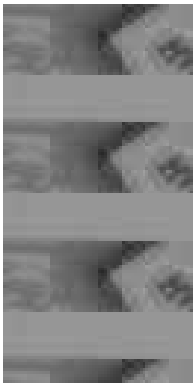


Abbildung 31: Das reale Testbild (quantisiert)

Anmerkung: Das Bild ist horizontal und vertikal gespiegelt, da bei der Aufnahme nur eine Linse verwendet wurde. Es ist also ein direktes Resultat des Linsenabbildungsgesetzes. Die sich weiderholenden Bildbereiche entstanden durch Vervielfachung des ursprünglich nur fünf 8×8-Reihen umfassenden Bildes.

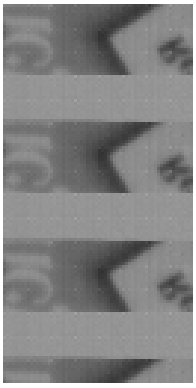


Abbildung 32: Das reale Testbild (unquantisiert)

Abbildung 32 zeigt das Bild ohne jegliche Quantisierung. Die Dateigröße beträgt 4984 Bytes. Ohne vorherige Quantisierung läßt sich der Datenstrom bedeutend schlechter komprimieren. Allein durch die Codierung wurde jetzt eine Kompression erreicht. (Ein Bitmap (BMP) wäre fast doppelt so groß.) Das so erhaltene Bild gleicht exakt dem Original, da ohne Quantisierung keinerlei verlustbehaftete Kompression eingesetzt wurde.

Vergleicht man beide Bilder, so werden selbstverständlich die typischen JPEG-Artefakte im quantisierten Bild auffallen. Dennoch ist das Motiv fast ebenso gut zu erkennen, wie bei keiner Quantisierung. Dabei ist das unquantisierte Bild 5,8 mal so groß, wie das quantisierte.

7.1.2 Das zufällige Bild

Um alle Baublöcke gemeinsam testen zu können, wurde zufällig ein Datenstrom erzeugt, der dem Datenstrom des Output des VISP2000 entspricht. Durch die

RSD-Korrektur ist es aber nicht möglich den Datenstrom so anzupassen, dass bestimmte Effekte erzielt werden können. Es handelt sich also um völlig willkürliche Bitkombinationen. Die Daten werden durch die Testbench von einem System-File aus eingelesen.

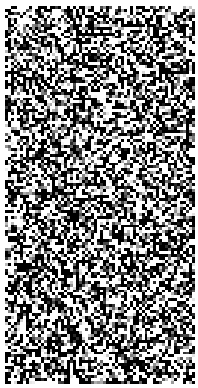


Abbildung 33: Das zufällige Bild (7235 Bytes)

Abbildung 33 zeigt ein Bild mit extrem vielen hochfrequenten Anteilen. Diese lassen sich so schlecht codieren, dass der hier vorgestellte JPEG-Algorithmus es nicht schafft, diese in Echtzeit zu verarbeiten. Für Abbildung 33 wurde deshalb die Quantisierung eingeschaltet, um den Datenstrom überhaupt codieren zu können. Da aber in der Realität niemals ein solches Bild auftreten wird, kann man darauf verzichten, diesen Extremfall unquantisiert codieren zu können.

Dieses (zugegebenermaßen unrealistische) Bild bietet dennoch einige interessante Aspekte in Bezug auf den Test der JPEG-Komponente. Es enthält unter anderem viele hochfrequente Anteile, die eine Codierung schwer machen und den Ausgangsdatenpuffer schnell füllen, zum Teil starke Unterschiede zwischen den einzelnen 8×8-Blöcken, was hohe DC-Differenzen erzeugt und einige hochgewichtete Wechselanteile, die codiert werden mit Codewörtern, die bei natürlichen Bildern eher selten auftreten. Damit bietet dieses Bild eine Testmöglichkeit für die JPEG-Komponente, die sehr unterschiedliche Anforderungen an die Teilkomponenten stellt. Mangels weiterer natürlicher Testbilder konnte so die Funktionsweise trotzdem mit einem großen Spektrum an Daten getestet werden.

Ein anderes Extrem wären großflächig homogene Bildteile. Dieser Fall ist am ehesten mit dem realen Testbild aus Kapitel 7.1.1 und mit den Möglichkeiten, die in Kapitel 7.1.3 beleuchtet werden, nachzubilden.

7.1.3 Das Testbild aus dem Testbildgenerator

In Kapitel 5.5 wurde mit der Kombination der ersten 3 Verarbeitungsstufen auch gleichzeitig die Möglichkeit implementiert, einige Testdaten synthesefähig in der JPEG-Komponente zu erzeugen. Dabei handelt es sich zwar nur um 64 DCT-transformierte Bildwerte (siehe Kapitel 9.7), aber das genügt, um prinzipiell die Funktion zu überprüfen.

Der Grund, warum gerade diese 64 Werte für das Testbild gewählt wurden, ist der, dass für den Test des FireWire-Übertragung Zahlenwerte zur Verfügung stehen sollten, die eine „ansteigende Rampe“ beschreiben. Legt man keinen Wert

auf eine solch spezielle Funktion, so sind beliebige andere 64 Werte möglich.

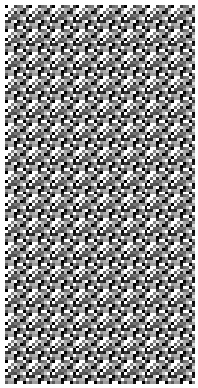


Abbildung 34: Das Testbild (quantisiert, 4485 Bytes)

Abbildung 34 zeigt das quantisierte Testbild, was durch den synthesefähigen Testbildgenerator erzeugt wurde. Deutlich zu sehen ist die ständige Wiederholung der 8×8-Blöcke und der hohe Anteil von hochfrequenten Bildanteilen.

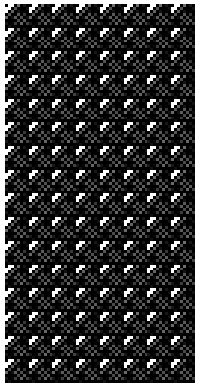


Abbildung 35: Das Testbild (unquantisiert, 15056 Bytes)

Abbildung 35 zeigt das selbe Testbild, aber dieses Mal ohne Quantisierung. Auffällig ist der gravierende Unterschied zum quantisierten Testbild. Dieser kommt allein durch die Quantisierung zustande. Wie im Listing comb3.vhdl in Kapitel 9.7 zu sehen ist, existieren sehr viele hochfrequente Anteile im Bild. Diese werden nur sehr grob durch die zu ihnen gehörenden hohen Quantisierungskoeffizienten aufgelöst. Zusätzlich sind die Quantisierungskoeffizienten nicht symmetrisch bezüglich der x- und y-Achse. Diese Unsymmetrie ist deutlich daran zu erkennen, dass in Abbildung 34 die x-/y-Symmetrie von Abbildung 35 nur noch zu erahnen ist. Dass es sich dennoch um das selbe Bild handeln muss, sieht man erst, wenn man sich jeweils die linken oberen Ecken der 8×8-Blöcke ansieht und diese bei beiden Bildern vergleicht.

In der Tat ist eine solche Verfälschung des Bildes durch die Quantisierung erschreckend, aber dieses Beispiel repräsentiert einen sehr unrealistischen und in

der Realität eigentlich nicht vorkommenden Fall. Nichtsdestotrotz ist auch dieser Fall für einen einfachen Funktionstest der synthetisierten Schaltung ausreichend.

Erwähnt sei noch die ungewöhnliche Größe des Bildes. Aufgrund des festen Huffman-Codewortalphabetes, was auf natürlich vorkommende Bilder optimiert ist, entsteht mit dem unquantisierten Testbild ein aussergewöhnlich starker Datenstrom. Eine Echtzeit-Huffman-Codierung würde hier weitaus bessere Werte liefern, aber eben auch deutlich mehr Ressourcen verbrauchen.

7.2 Synthese

Eine Synthese der JPEG-Komponente wurde durchgeführt. Als Optimierungskriterium wurde eine Optimierung nach Chipfläche gewählt. Es wurde ein Mapping auf die FPGA durchgeführt, aber da die JPEG-Komponente nie allein arbeiten wird, wurden keinerlei Vorgaben gemacht, welche Signale auf welche Pins gemappt werden sollen. Einzig der Takteingang für das Signal „fastclock“ wurde mit einer Taktvorgabe belegt. Mit dieser Strategie wird natürlich keine optimale Realisierung auf der FPGA entstehen. Das hat zwei hauptsächliche Gründe: Einerseits kann das Synthese- und Mapping-Tool nicht wissen, dass das Signal „pixclock“ ausgesprochen langsam ist und andererseits entstehen sehr lange Signalwege auf der FPGA, da jetzt alle Inputs und Outputs der JPEG-Komponente an externe Pins geführt werden. Im Verbund mit der Kamerasteuerung und der Ausgabeeinheit werden kürzere Wege und damit kürzere Signallaufzeiten erwartet. In dieser jetzt synthetisierten Form mit Optimierung auf Chipfläche ist ein Takt von 38 MHz erreichbar.

Damit wird auch die maximale Bildwiederholrate bestimmt. Laut der Aussage von [9] benötigt eine Zeile aus 8×8-Blöcken 33646 Taktzyklen. Das Bild besitzt 16 dieser Zeilen. Damit berechnet sich die Bildwiederholrate wie folgt:

$$fps = \frac{1}{33646 \times 16 \times T_{clk}} \tag{10}$$

Mit einer Taktperiode $T_{clk} = 1/38\,MHz = 26,3\,ns$ ergibt dies eine Bildwiederholrate $fps \approx 70\frac{1}{s}$.

Die nötigen Ressourcen auf der FPGA veranschaulicht folgende Tabelle:

	belegt	Auslastung
Slices	1,570	32%
Slice-FlipFlops	827	8%
Look-up-tables	2,603	27%
BlockRAM	6	30%
GCLK	2	50%
äquivalente Gate-Anzahl	121,858	

In der Tabelle wurden bewusst alle Angaben bezüglich der I/O-Pinbelegungen nicht erwähnt. In dem Gesamtprojekt wird die JPEG-Komponente nämlich gerade einmal 8 Pins für die Übernahme der Daten des VISP2000 und ein Pin für den Takt „fastclock“ benötigen. Alle anderen Signale, die in die JPEG-Komponente hinein oder aus ihr hinaus gehen, werden von der Steuerung bereit gestellt oder gehen an die Ausgabestufe.

Die synthetisierte Version mit Timing-Informationen wurde in das Gesamtprojekt eingebettet und mit den unsynthetisierten anderen Teilkomponenten zusammen getestet. Es wurde mit dem synthesefähigen Testbildgenerator (siehe Kapitel 7.1.3) und dem zufälligen Bild (siehe Kapitel 7.1.2) bei einem Takt von knapp 28 MHz getestet. Die Simulationsergebnisse der Timing-Simulationen stimmen mit dem Ergebnissen der theoretischen Simulationen exakt überein.

8 Zusammenfassung und Wertung

Es wurde eine synthesefähige Realisierung der Quantisierung und Codierung des JPEG-Kompressionsalgorithmus, eingebettet in das Konzept des Demonstrator-Boards vorgestellt. Diese Realisierung zeichnet sich durch einen geringen Ressourcenbedarf aus. Es ist möglich, bis zu einer Bildwiederholrate von etwa 70 Bildern pro Sekunde eine Echtzeit-JPEG-Codierung durchzuführen. Dabei stehen die Möglichkeiten zur Wahl, verlustbehaftet zu quantisieren, oder das Bild ohne Qualitätsverlust codieren zu lassen.

Zusätzlich ist es möglich, Kameradaten direkt, also ohne JPEG-Codierung auszugeben, wenn dies gewünscht ist.

Möglichkeiten zum Test der Übertragungsstrecke und der JPEG-Komponente unabhängig von den Bilddaten wurden implementiert.

Die hier vorgestellte Realisierung stellt eine sehr speziell auf den Kamerasensor VISP2000 angepasste Version dar. Es wäre nur mit teilweise erheblichem Mehraufwand möglich, sie bei höheren Auflösungen oder gar mit anderen Kamerasensoren einzusetzen. Prinzipbedingt sind die Ergebnisse der JPEG-Codierung je nach Bild nicht immer optimal. Das liegt einerseits an der starren Quantisierungstabelle und andererseits an dem fest vorgegebenem Code-Alphabet. Bei natürlich vorkommenden Bildern sollten allerdings die Größenunterschiede zu optimal JPEG-codierten Bildern vernachlässigbar klein sein.

Ein kleiners Problem soll nicht verschwiegen werden: Bei abgeschalteter Quantisierung und einem Bild mit einem sehr großen Spektrum an Bildanteilen (im Frequenzbereich) kann es zum Ausgangspuffer-Überlauf kommen. Dieser Fall ist aber so synthetisch und unrealistisch, dass er ignoriert werden kann. In der Natur wird kein Bild auftreten, was über seine gesamte Bildfläche ein solch breites Spektrum an Bildanteilen liefert. Ausserdem stellt das Abschalten der Quantisierung eher ein zusätzliches Feature dar, was im Normalfall sowieso nicht eingesetzt werden soll.

Der hier vorgestellte JPEG-Algorithmus generiert Einzelbild-JPEG. Dadurch ist es leicht, einzelne Bilder aus dem Datenstrom heraus zu greifen, aber zeitliche Korrelationen zwischen den Bildern werden nicht ausgenutzt.

Literatur

[1] J. Schreiter, A. Graupner, S. Getzlaff, R. Srowik, and R. Schüffny. A CMOS Image Sensor with Parallel Analog Processing Unit for Transformation and Spatial Convolutions. http://www.iee.et.tu-dresden.de/iee/hpsn/ Weitere Details in [2]

[2] Stefan Getzlaff, Achim Graupner, Jörg Schreiter. VISP-Teststruktur. Unveröffentlichte Technische Dokumentation, Technische Universität Dresden, 2000

[3] Mirko Pügner: VHDL-Implementierung einer universellen Schnittstelle für CMOS-Bildsensoren auf FPGA. Technischer Report, Sonderforschungsbereich 358 - A7, Technische Universität Dresden, Germany, September 2000

[4] Texas Instruments: TMS320C3x Users Guide, 1997

[5] Jens-Uwe Schlüßler. Modifikation von [3], Entwicklung eines Treibers und einer Video-Ausgabe für PC für das Demonstrator-Board. Noch undokumentiert, Technische Universität Dresden, 2001

[6] Pennebaker, William B.; Mitchell, Joan L.: „JPEG: still image data compresion standard"; New York: Van Nostrand Reynold; ISBN 0-442-01272-1

[7] Vasudev Bashkaran, Konstantinos Konstantinides: Image and Video compression standards

[8] http://ei.cs.vt.edu/ mm/pdfs/wallace.pdf

[9] Matthias Steidl. Entwicklung einer Demonstrationsumgebung für die digitale Kamera mit Vorverarbeitung VISP2000, TU-Dresden, 2001

[10] Jörg Schreiter. Quellcode zur RSD-Korrektur. Unveröffentlichte Software, Technische Universität Dresden, 2000

[11] http://www.xilinx.com/

[12] Vorlesung „Codierungstechnik", Professor Adolf Finger, Technische Universität Dresden; http://www.ifn.et.tu-dresden.de/tnt/index.htm

[13] http://swww.ee.uwa.edu.au/~plsd210/ds/huff-op.html

[14] http://ls4-www.cs.uni-dortmund.de/~Misch/applets/AAP/huffman.html

9 Anhang (Quellcodes)

9.1 Die Verschiebung um -128 bei der DCT

Zur Berechnung von Gleichung (7) wurde folgendes Listing in Microsoft Visual C++ 6.0 geschrieben:

```
#include "stdafx.h"
#include <math.h>
#include <iostream.h>

int main()
{
  const double C0 = 1/(sqrt(2));
  const double Cx = 1;
  const double pi = 3.14159265358979323846264338327795;

  double value = 0;
  double value_big = -1023;
  double value_small = 1023;
  double pixvalue = -32;
  int u=0;
  int v=0;

  for (u=0; u<8; u++)
  {
    for (v=0; v<8; v++)
    {
      value=0;
      for (int x=0; x<8; x++)
      {
        for (int y=0; y<8; y++)
        {
          value = value + cos((2*x+1)*u*pi/16)*cos((2*y+1)*v*pi/16);
        }
      }

      if (u==0)
      {
        if (v==1)
          value = pixvalue*C0*Cx*value;
        else
          value = pixvalue*C0*C0*value;
      }
      else
      {
        if (v==1)
          value = pixvalue*Cx*Cx*value;
        else
          value = pixvalue*Cx*C0*value;
      }
      if (value>value_big) value_big=value;
      if (value<value_small) value_small=value;

      cout << "u=" << u << " v=" << v << " Koeffizient=" <<value << endl;
    }
  }
  cout << "Kleinster Wert: " << value_small << " Groesster Wert: " << value_big << endl;
  return 0;
}
```

9.2 Die DCT auf dem VISP2000

Folgendes Listing in MS Visual C++ 6.0 berechnet die für die Ausführung der DCT auf dem VISP2000 notwendigen Ansteuerungskoeffizienten in der gewünschten Reihenfolge (zig-zag) und dem Datenformat, welches Matthias Steidl in seiner Studienarbeit [9] benötigt.

```
#include "stdafx.h"
#include <iostream.h>
#include <math.h>
```

```
#include <fstream.h>

int main()
{
int u;
int v;
double wurzel = 1/sqrt(2);
const double pi = 3.14159265358979323846264338327795;
double coeff;
double coeff_biggest=0;
double coeff_smallest=0;
ofstream outfile("coeff.txt", ios::out );
ofstream outfile_real("coeff_real.txt", ios::out );
ofstream out_sign("coeff_sign.txt", ios::out );
ofstream out_nocom("coeff_nocom.txt", ios::out );
//int setmode( int nMode = filebuf::text );
int negative;
bool coeff_bin[10];
bool actual_bit;
bool carry;
double coeff_dummy;
// zig-zag-order (coordinates):
int u_zig[65];
u_zig[1]=1; u_zig[2]=2; u_zig[3]=1; u_zig[4]=1; u_zig[5]=2; u_zig[6]=3; u_zig[7]=4;
u_zig[8]=3; u_zig[9]=2; u_zig[10]=1; u_zig[11]=1; u_zig[12]=2; u_zig[13]=3; u_zig[14]=4;
u_zig[15]=5; u_zig[16]=6; u_zig[17]=5; u_zig[18]=4; u_zig[19]=3; u_zig[20]=2; u_zig[21]=1;
u_zig[22]=1; u_zig[23]=2; u_zig[24]=3; u_zig[25]=4; u_zig[26]=5; u_zig[27]=6; u_zig[28]=7;
u_zig[29]=8; u_zig[30]=7; u_zig[31]=6; u_zig[32]=5; u_zig[33]=4; u_zig[34]=3; u_zig[35]=2;
u_zig[36]=1; u_zig[37]=2; u_zig[38]=3; u_zig[39]=4; u_zig[40]=5; u_zig[41]=6; u_zig[42]=7;
u_zig[43]=8; u_zig[44]=8; u_zig[45]=7; u_zig[46]=6; u_zig[47]=5; u_zig[48]=4; u_zig[49]=3;
u_zig[50]=4; u_zig[51]=5; u_zig[52]=6; u_zig[53]=7; u_zig[54]=8; u_zig[55]=8; u_zig[56]=7;
u_zig[57]=6; u_zig[58]=5; u_zig[59]=6; u_zig[60]=7; u_zig[61]=8; u_zig[62]=8; u_zig[63]=7;
u_zig[64]=8;
int v_zig[65];
v_zig[1]=1; v_zig[2]=1; v_zig[3]=2; v_zig[4]=3; v_zig[5]=2; v_zig[6]=1; v_zig[7]=1;
v_zig[8]=2; v_zig[9]=3; v_zig[10]=4; v_zig[11]=5; v_zig[12]=4; v_zig[13]=3; v_zig[14]=2;
v_zig[15]=1; v_zig[16]=1; v_zig[17]=2; v_zig[18]=3; v_zig[19]=4; v_zig[20]=5; v_zig[21]=6;
v_zig[22]=7; v_zig[23]=6; v_zig[24]=5; v_zig[25]=4; v_zig[26]=3; v_zig[27]=2; v_zig[28]=1;
v_zig[29]=1; v_zig[30]=2; v_zig[31]=3; v_zig[32]=4; v_zig[33]=5; v_zig[34]=6; v_zig[35]=7;
v_zig[36]=8; v_zig[37]=8; v_zig[38]=7; v_zig[39]=6; v_zig[40]=5; v_zig[41]=4; v_zig[42]=3;
v_zig[43]=2; v_zig[44]=3; v_zig[45]=4; v_zig[46]=5; v_zig[47]=6; v_zig[48]=7; v_zig[49]=8;
v_zig[50]=8; v_zig[51]=7; v_zig[52]=6; v_zig[53]=5; v_zig[54]=4; v_zig[55]=5; v_zig[56]=6;
v_zig[57]=7; v_zig[58]=8; v_zig[59]=8; v_zig[60]=7; v_zig[61]=6; v_zig[62]=7; v_zig[63]=8;
v_zig[64]=8;
for (int i=1; i<=64; i++)
{
    u=u_zig[i]-1;
    v=v_zig[i]-1;
    for (int bitposition=9; bitposition>=0; bitposition--)
    {
      outfile << "Teil der DCT-Koeffizienten für u=" << u << " v=" << v << " Bitposition="
              << bitposition<< endl;
      if (bitposition==0) outfile_real << "64 DCT-Koeffizienten für u=" << u
              << " v=" << v << endl; // only once
      if (bitposition==0) out_sign << "64 Vorzeichen für u=" << u
              << " v=" << v << endl; // only once
      for (int y=0 ; y<8 ; y++)
      {
        if (bitposition==0) outfile_real << endl; // only once
        for (int x=0 ; x<8 ; x++)
        {
          coeff = cos((2*x+1)*u*pi/16) * cos((2*y+1)*v*pi/16);
          if (u==0) coeff = coeff * wurzel;
          if (v==0) coeff = coeff * wurzel;
          if (u==0)
          {
            if (v==0)
            {
              coeff = coeff/2;
            }
          }
          coeff = coeff/4;
          // set of values of VISP2000 is only [-1023,1024], but DCT will produce
          //coeff = coeff/2;
```

```
          if (coeff>coeff_biggest) coeff_biggest=coeff;
          if (coeff<coeff_smallest) coeff_smallest=coeff;
          // up to here - coefficients are finished as real numbers
          coeff_dummy = coeff;
          coeff = coeff * 1024;
          if (coeff<0) negative=1;
          else negative=0;
          coeff = sqrt(coeff*coeff);
          coeff = coeff + 0.005;
          coeff_bin[9]=0;
          for (int n=8; n>=0; n--)
          {
            if (pow(2,n)<=coeff)
            {
              coeff_bin[n]=1;
              coeff = coeff - pow(2,n);
            }
            else coeff_bin[n]=0;
          }
          if (coeff>=0.5)
          {
            carry=1;
            for (int m=0; m<=9; m++)
            {
              actual_bit=coeff_bin[m];
              coeff_bin[m] = coeff_bin[m] ^ carry;
              carry = actual_bit && carry;
            }
          }
          if (bitposition==0) out_sign << negative;
          if (coeff_bin[bitposition]==true)
            outfile << "1";
          else outfile << "0";
          if (coeff_bin[bitposition]==true)
            out_nocom << "1";
          else out_nocom << "0";
          if (bitposition==0) outfile_real << coeff_dummy << " "; // only once
        }
      }
      outfile << endl << endl;
      out_nocom << endl;
      if (bitposition==0) outfile_real << endl << endl; // only once
      if (bitposition==0) out_sign << endl << endl; // only once
    }
}
outfile_real << "Der grösste Koeffizient ist " << coeff_biggest << endl;
outfile_real << "Der kleinste Koeffizient ist " << coeff_smallest << endl;

outfile.close();
outfile_real.close();
out_sign.close();
out_nocom.close();
return 0;
}
```

9.3 Die Realisierung der RSD-Korrektur

comptodig.vhdl

```
-- Conversion of the 4 bit comparator-result of the VISP chip
-- to a digital 3 Bit value plus an error-recognisation.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL, IEEE.NUMERIC_STD.all;
use WORK.ALL;

entity comptodig is
port (comp    : in  std_logic_vector(3 downto 0);
      dig_out : out std_logic_vector(2 downto 0);
      err     : out std_logic    );
end entity comptodig;

architecture behavior_comptodig of comptodig is
```

```
signal    dig    : signed(2 downto 0);
begin

process (comp)
begin
case comp is
-- comp is mirrored related to the variable in the C-programm
  when "0000" =>  dig<=to_signed(0,3);
                  err<='0';
  when "0001" =>  dig<=to_signed(-1,3);
                  err<='0';
  when "0010" =>  dig<=to_signed(-1,3);
                  err<='1';
  when "0011" =>  dig<=to_signed(-2,3);
                  err<='0';

  when "0100" =>  dig<=to_signed(1,3);
                  err<='0';
  when "0101" =>  dig<=to_signed(0,3);
                  err<='1';
  when "0110" =>  dig<=to_signed(0,3);
                  err<='1';
  when "0111" =>  dig<=to_signed(-1,3);
                  err<='1';

  when "1000" =>  dig<=to_signed(1,3);
                  err<='1';
  when "1001" =>  dig<=to_signed(0,3);
                  err<='1';
  when "1010" =>  dig<=to_signed(0,3);
                  err<='1';
  when "1011" =>  dig<=to_signed(-1,3);
                  err<='1';

  when "1100" =>  dig<=to_signed(2,3);
                  err<='0';
  when "1101" =>  dig<=to_signed(1,3);
                  err<='1';
  when "1110" =>  dig<=to_signed(1,3);
                  err<='1';
  when "1111" =>  dig<=to_signed(0,3);
                  err<='1';

  when others  =>  null;

end case;
end process;

process (dig)
begin
for N in 0 to 2 loop
  dig_out(N)<=dig(N);
end loop;
end process;

end architecture behavior_comptodig;

configuration comptodig_cfg of comptodig is
for behavior_comptodig
end for;
end comptodig_cfg;
```

9.4 Die Realisierung der RSD-Korrektur

rsd.vhdl

```
-- This listing is based on a C-program by Joerg Schreiter.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL, IEEE.NUMERIC_STD.all;
use WORK.ALL;

entity rsd is
```

```
port (pixclock        : in  std_logic;
      ADControl_reset : in  std_logic;
      compdata        : in  std_logic_vector(7 downto 0);
      error           : out std_logic_vector(1 downto 0);
      digital         : out std_logic_vector(10 downto 0);
      state8          : out std_logic_vector(2 downto 0);
      state64         : out std_logic_vector(5 downto 0);
      state8_old      : out std_logic_vector(2 downto 0)  );
end entity rsd;

-- state8 and state64 are pointers correlated with the special
-- output stream of the VISP. (See documtentation.)
-- These signals are triggers for the next stages. They show from
-- witch pixel a data word has come.

architecture behavior_rsd of rsd is

component comptodig
port (comp    : in  std_logic_vector(3 downto 0);
      dig_out : out std_logic_vector(2 downto 0);
      err     : out std_logic    );
end component;

signal    digital_sig    : signed(10 downto 0);
signal    state64_unsig  : unsigned(5 downto 0);

signal    comp1          : std_logic_vector(3 downto 0);
signal    dig1           : signed(2 downto 0);
signal    dig1_slv       : std_logic_vector(2 downto 0);
signal    err1           : std_logic;
signal    comp2          : std_logic_vector(3 downto 0);
signal    dig2           : signed(2 downto 0);
signal    dig2_slv       : std_logic_vector(2 downto 0);
signal    err2           : std_logic;

signal    i              : unsigned(2 downto 0);
signal    i_next         : unsigned(2 downto 0);
signal    step           : integer range 0 to 64;
signal    eight          : unsigned(2 downto 0);
signal    eight_old      : unsigned(2 downto 0);
signal    latch0         : signed(10 downto 0);
signal    latch1         : signed(10 downto 0);
signal    latch2         : signed(10 downto 0);
signal    latch3         : signed(10 downto 0);
signal    latch4         : signed(10 downto 0);
signal    latch5         : signed(10 downto 0);
signal    latch6         : signed(10 downto 0);
signal    latch7         : signed(10 downto 0);
signal    latch8         : signed(10 downto 0);
signal    latch9         : signed(10 downto 0);
signal    latchinit      : std_logic;
signal    errori         : std_logic_vector(1 downto 0);

signal    state64_int    : unsigned(5 downto 0);

begin

-- Convert a 4Bit comparator result to a 3 bit digital word.
Comp1_to_Dig : comptodig
port map(
  comp=>comp1,
  dig_out=>dig1_slv,
  err=>err1  );

Comp2_to_dig : comptodig
port map(
  comp=>comp2,
  dig_out=>dig2_slv,
  err=>err2  );

-- I/O-conversion
process (dig1_slv,dig2_slv)
begin
for N in 0 to 2 loop
```

```
  dig1(N)<=dig1_slv(N);
  dig2(N)<=dig2_slv(N);
end loop;
end process;


process (digital_sig,eight,eight_old,state64_unsig)
begin
for N in 0 to 10 loop
  digital(N)<=digital_sig(N);
end loop;
for M in 0 to 2 loop
  state8(M)<=eight(M);
  state8_old(M)<=eight_old(M);
end loop;
for T in 0 to 5 loop
  state64(T)<=state64_unsig(T);
end loop;
end process;


data_in : process (compdata)
begin
comp1<=compdata(7 downto 4);
comp2<=compdata(3 downto 0);
end process data_in;


process (pixclock,ADControl_reset,i,i_next,step,eight)
begin

-- asynchronous reset + init
if (ADControl_reset='0') then
  i<="100"; --3+1
  i_next<="011";
  step<=0;
  errori<="00";
  -- avoid logic U
  state64_int<="111111";
  eight<="111";
  eight_old<="111";
  latchinit<='1';
elsif rising_edge(pixclock) then
  i<=i_next;
  latchinit<='0';
  if (step<44) then
    step<=step+1;
    -- step runs 1 clock cycle too fast
    -- > o.k.
    if (step=43) then
      state64_int<="111111";
      eight<="111";
    end if;
  else  errori(1)<=err1 or err2;
  end if;
  case i_next is
  when "000" =>  i_next<="100"; --4
                 if (step>=44) then errori(0)<='0';
                 end if;
  when "100" =>  i_next<=i_next-1;
                 if (step>=44) then
                   errori(0)<='1';
                   eight<=eight+1;
                   if (eight="111") then state64_int<=state64_int+1;
                   end if;
                 end if;
  when "011" =>  eight_old<=eight;
                 i_next<=i_next-1;
                 if (step>=44) then errori(0)<='0';
                 end if;
  when others =>  i_next<=i_next-1;
                 if (step>=44) then errori(0)<='0';
                 end if;
  end case;
```

```
end if;
end process;

process (state64_int)
begin
state64_unsig<=state64_int;
end process;

-- RSD-kernel.
latchfunction : process (i,latchinit,dig1,dig2,latch0,latch1,latch2,latch3,
                         latch4,latch5,latch6,latch7,latch8,pixclock)
begin
if rising_edge(pixclock) then
if (latchinit='1') then
-- The registers where called "latch" in the C-programm code,
-- but they are here described as FFs.
  latch0<=(others=>'0');
  latch1<=(others=>'0');
  latch2<=(others=>'0');
  latch3<=(others=>'0');
  latch4<=(others=>'0');
  latch5<=(others=>'0');
  latch6<=(others=>'0');
  latch7<=(others=>'0');
  latch8<=(others=>'0');
  latch9<=(others=>'0');
else  case i is
  when "000" =>  latch0<=resize(dig1,11);
                 latch5<=shift_left(latch4,1) + resize(dig2,11);
  when "001" =>  latch1<=shift_left(latch0,1) + resize(dig1,11);
                 latch6<=shift_left(latch5,1) + resize(dig2,11);
  when "010" =>  latch2<=shift_left(latch1,1) + resize(dig1,11);
                 latch7<=shift_left(latch6,1) + resize(dig2,11);
  when "011" =>  latch3<=shift_left(latch2,1) + resize(dig1,11);
                 latch8<=shift_left(latch7,1) + resize(dig2,11);
  when "100" =>  latch4<=shift_left(latch3,1) + resize(dig1,11);
                 latch9<=shift_left(latch8,1) + resize(dig2,11);
  --redundant - appears never
  when others =>  latch0<=resize(dig1,11);
                 latch5<=shift_left(latch4,1) + resize(dig2,11);
  end case;
end if;
end if;
end process latchfunction;
-- RSD-kernel. End.

-- Output-driver.
process (errori,latch9)
begin
-- error-code:
-- 0=nothing, 1=there is a new values in latch9
-- 2=error, 3=error and there is a new values in latch9
error<=errori;
digital_sig<=latch9;
end process;
-- Output-driver.

end architecture behavior_rsd;

configuration rsd_cfg of rsd is
for behavior_rsd
end for;
end rsd_cfg;
```

9.5 Die Anpassung des Wertebereichs

minus128.vhdl

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL, IEEE.NUMERIC_STD.all;
use WORK.ALL;
```

```
entity minus128 is
port (DCT_data     : in  std_logic_vector(10 downto 0);
      state64      : in  std_logic_vector(5 downto 0);
      enddata      : out std_logic_vector(10 downto 0)  );
end entity minus128;

architecture behavior128 of minus128 is
constant  subtrahend         : signed(11 downto 0):=to_signed(-1024,12);
signal    DCT_data_sig       : signed(10 downto 0);
signal    enddata_sig        : signed(10 downto 0);

signal    oversize_data      : signed(11 downto 0);
signal    oversize_data_slv  : std_logic_vector(11 downto 0);
signal    oversize_enddata   : signed(11 downto 0);
begin

convert_to : process (DCT_data)
begin
--multiplication with 2
oversize_data_slv(11 downto 1)<=DCT_data(10 downto 0);
oversize_data_slv(0)<='0';
end process convert_to;

process (oversize_data_slv,enddata_sig,DCT_data)
begin
for N in 0 to 11 loop
  oversize_data(N)<=oversize_data_slv(N);
end loop;
for N in 0 to 10 loop
  DCT_data_sig(N)<=DCT_data(N);
  enddata(N)<=enddata_sig(N);
end loop;
end process;

subtraction : process(oversize_data)
begin
if (oversize_data(11)='1') then
  oversize_enddata<=subtrahend; -- If a DC-coeff. is <0 (because of noise).
else  oversize_enddata<=oversize_data+subtrahend;
end if;
end process subtraction;

convert_back : process (oversize_enddata,state64,DCT_data_sig)
begin
if (state64="000000") then
  -- Only DC-coefficients are subtracted by 128.
  enddata_sig<=resize(oversize_enddata,11);
else  enddata_sig<=DCT_data_sig;
end if;
end process convert_back;

end architecture behavior128;

configuration cfg128 of minus128 is
for behavior128
end for;
end cfg128;
```

9.6 Quantisierung

division.vhdl

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL, IEEE.NUMERIC_STD.all;
use WORK.ALL;

entity division is
port (data_todiv   : in  std_logic_vector(10 downto 0);
      state64      : in  std_logic_vector(5 downto 0);
      quant_data   : out std_logic_vector(10 downto 0);
      quantisation : in  std_logic    );
end entity division;
```

```
architecture div_behavior of division is

component full_add
port (in1       : in  std_logic;
      in2       : in  std_logic;
      carry_in  : in  std_logic;
      carry_out : out std_logic;
      output    : out std_logic  );
end component;

type divisor_array_type is array (natural range<>) of std_logic_vector(15 downto 0);
constant  divisor_array  : divisor_array_type(0 to 63):=(
  "0000100000000000","0000101110100010","0000110011001100","0000100000000000",
  "0000010101010101","0000001100110011","0000001010000010","0000010000011001",
  "0000101010101010","0000101010101010","0000100100100100","0000011010111100",
  "0000010011101100","0000001000110100","0000001000100010","0000001001010011",
  "0000100100100100","0000100111011000","0000100000000000","0000010101010101",
  "0000001100110011","0000001000111110","0000000111011010","0000001001001001",
  "0000100100100100","0000011010111100","0000010111010001","0000010001101001",
  "0000001010000010","0000000101111000","0000000110011001","0000001000010000",
  "0000011100011100","0000010111010001","0000001101110101","0000001001001001",
  "0000000111100001","0000000100101100","0000000100111110","0000000110101001",
  "0000010101010101","0000001110101000","0000001001010011","0000001000000000",
  "0000000110010100","0000000100111011","0000000100100001","0000000101100100",
  "0000001010011100","0000001000000000","0000000110100100","0000000101111000",
  "0000000100111110","0000000100001110","0000000100010001","0000000101000100",
  "0000000111000111","0000000101100100","0000000101011000","0000000101001110",
  "0000000100100100","0000000101000111","0000000100111110","0000000101001010"    );
-- Luminance quantisation table (example from ISO DIS 10918-1 Annex K)
--Divisor<=1 (!) if you want to change this table.

signal    data_todiv_sig  : signed(10 downto 0);
signal    out_data  : signed(10 downto 0);
signal    state64_unsig  : unsigned(5 downto 0);
signal    quant_data_sig  : signed(10 downto 0);

signal    divisor    : std_logic_vector(15 downto 0);
signal    divident  : signed(10 downto 0);
signal    sign_divident  : std_logic;

signal    zero    : std_logic;

-- Signals of the sums of the FAs.
signal    s1,s2,s3,s4,s5,s6,s7,s8,s9,s10,s11,s12,s13,s14,s15,s16,s17,s18,s19
    ,s20,s21,s22,s23,s24,s25,s26,s27,s28,s29,s30,s31,s32,s33,s34,s35,s36,s37,s38,s39
    ,s40,s41,s42,s43,s44,s45,s46,s47,s48,s49,s50,s51,s52,s53,s54,s55,s56,s57,s58,s59
    ,s60,s61,s62,s63,s64,s65,s66,s67,s68,s69,s70,s71,s72,s73,s74,s75,s76,s77,s78,s79
    ,s80,s81,s82,s83,s84,s85,s86,s87,s88,s89,s90,s91,s92,s93,s94,s95,s96,s97,s98,s99
    ,s100,s101,s102,s103,s104,s105,s106,s107,s108,s109,s110,s111,s112,s113,s114,s115
    ,s116,s117,s118,s119,s120,s121,s122,s123,s124,s125,s126,s127,s128,s129,s130,s131
    ,s132,s133,s134,s135,s136,s137,s138,s139,s140,s141,s142,s143,s144  : std_logic;

-- Signals of the carrys of the FAs.
signal    c1,c2,c3,c4,c5,c6,c7,c8,c9,c10,c11,c12,c13,c14,c15,c16,c17,c18,c19
    ,c20,c21,c22,c23,c24,c25,c26,c27,c28,c29,c30,c31,c32,c33,c34,c35,c36,c37,c38,c39
    ,c40,c41,c42,c43,c44,c45,c46,c47,c48,c49,c50,c51,c52,c53,c54,c55,c56,c57,c58,c59
    ,c60,c61,c62,c63,c64,c65,c66,c67,c68,c69,c70,c71,c72,c73,c74,c75,c76,c77,c78,c79
    ,c80,c81,c82,c83,c84,c85,c86,c87,c88,c89,c90,c91,c92,c93,c94,c95,c96,c97,c98,c99
    ,c100,c101,c102,c103,c104,c105,c106,c107,c108,c109,c110,c111,c112,c113,c114,c115
    ,c116,c117,c118,c119,c120,c121,c122,c123,c124,c125,c126,c127,c128,c129,c130,c131
    ,c132,c133,c134,c135,c136,c137,c138,c139,c140,c141,c142,c143,c144  : std_logic;

signal    mul_row0  : std_logic_vector(8 downto 0);
signal    mul_row1  : std_logic_vector(9 downto 0);
signal    mul_row2  : std_logic_vector(10 downto 1);
signal    mul_row3  : std_logic_vector(11 downto 2);
signal    mul_row4  : std_logic_vector(12 downto 3);
signal    mul_row5  : std_logic_vector(13 downto 4);
signal    mul_row6  : std_logic_vector(14 downto 5);
signal    mul_row7  : std_logic_vector(15 downto 6);
signal    mul_row8  : std_logic_vector(16 downto 7);
signal    mul_row9  : std_logic_vector(17 downto 8);
signal    mul_row10  : std_logic_vector(18 downto 9);
```

```
signal    mul_row11  : std_logic_vector(19 downto 10);
signal    mul_row12  : std_logic_vector(20 downto 11);
signal    mul_row13  : std_logic_vector(21 downto 12);
signal    mul_row14  : std_logic_vector(22 downto 13);
signal    mul_row15  : std_logic_vector(23 downto 14);
signal    result    : unsigned(10 downto 0);
signal    round_result  : unsigned(10 downto 0);
signal    quant_data_abs  : signed(10 downto 0);

begin

--I/O-conversion
process (data_todiv,out_data)
begin
for N in 0 to 10 loop
  data_todiv_sig(N)<=data_todiv(N);
  quant_data(N)<=out_data(N);
end loop;
end process;

process(quantisation,quant_data_sig,data_todiv_sig)
begin
if (quantisation='1') then
  out_data<=quant_data_sig;
else  out_data<=data_todiv_sig; -- input=output (no quantisation)
end if;
end process;

process (state64)
begin
for N in 0 to 5 loop
  state64_unsig(N)<=state64(N);
end loop;
end process;

process (data_todiv_sig)
begin
sign_divident<=data_todiv_sig(10);
divident<=abs(data_todiv_sig);
zero<='0';
end process;


-- Array of divisors
-- search for the right divisor
process (state64_unsig)
begin
divisor<=divisor_array(to_integer(state64_unsig));
end process;
-- Array of divisors End

-- The mul_rows are inspired by the manual multiplication with a sheet of paper.
process (divident,divisor)
begin
for N in 0 to 8 loop
  mul_row0(N)<=(divident(N+1) and divisor(0));
end loop;
for N in 0 to 9 loop
  mul_row1(N)<=(divident(N) and divisor(1));
end loop;
for N in 1 to 10 loop
  mul_row2(N)<=(divident(N-1) and divisor(2));
end loop;
for N in 2 to 11 loop
  mul_row3(N)<=(divident(N-2) and divisor(3));
end loop;
for N in 3 to 12 loop
  mul_row4(N)<=(divident(N-3) and divisor(4));
end loop;
for N in 4 to 13 loop
  mul_row5(N)<=(divident(N-4) and divisor(5));
end loop;
for N in 5 to 14 loop
  mul_row6(N)<=(divident(N-5) and divisor(6));
```

```
end loop;
for N in 6 to 15 loop
  mul_row7(N)<=(divident(N-6) and divisor(7));
end loop;
for N in 7 to 16 loop
  mul_row8(N)<=(divident(N-7) and divisor(8));
end loop;
for N in 8 to 17 loop
  mul_row9(N)<=(divident(N-8) and divisor(9));
end loop;
for N in 9 to 18 loop
  mul_row10(N)<=(divident(N-9) and divisor(10));
end loop;
for N in 10 to 19 loop
  mul_row11(N)<=(divident(N-10) and divisor(11));
end loop;
for N in 11 to 20 loop
  mul_row12(N)<=divident(N-11) and divisor(12);
end loop;
for N in 12 to 21 loop
  mul_row13(N)<=divident(N-12) and divisor(13);
end loop;
for N in 13 to 22 loop
  mul_row14(N)<=divident(N-13) and divisor(14);
end loop;
for N in 14 to 23 loop
  mul_row15(N)<=divident(N-14) and divisor(15);
end loop;

end process;

FA1 : full_add
port map(
  in1=>mul_row0(0),
  in2=>mul_row1(0),
  carry_in=>zero,
  output=>s1,
  carry_out=>c1  );
```

Alle anderen Vollander werden analog dem Beispiel von FA1 instanziiert. Den Quelltext wiederzugeben, wird hier verzichtet, da Abbildung 12 und 13 auf den Seiten 18 und 19 genügend Aufschluss geben.

Es folgt noch das Ende des Listungs:

```
process (s114,s118,s122,s126,s130,s134,s138,s141,s143,s144,c144)
begin
result(0)<=s114;
result(1)<=s118;
result(2)<=s122;
result(3)<=s126;
result(4)<=s130;
result(5)<=s134;
result(6)<=s138;
result(7)<=s141;
result(8)<=s143;
result(9)<=s144;
result(10)<=c144;
end process;

process (result,s111)
begin
if (s111='1') then
  round_result<=result+1;
else  round_result<=result;
end if;
end process;

process (round_result)
begin
-- result is not allowed to be bigger than 1023!
-- -> divident<=1023 (everytime given) and Divisor<=1
for N in 0 to 10 loop
  quant_data_abs(N)<=round_result(N);
```

```
end loop;
end process;

process(quant_data_abs,sign_divident)
begin
if (sign_divident='1') then
  quant_data_sig<=-quant_data_abs;
else  quant_data_sig<= quant_data_abs;
end if;
end process;

end architecture div_behavior;

configuration div_cfg of division is
for div_behavior
end for;
end div_cfg;
```

9.7 Zusammenfassung der ersten 3 Stufen

comb3.vhdl

```
-- While writing the code, the first three stages of the JPEG-algorithm
-- where combined with this listing because of testing it as an big block.
-- Now an other task is computed here: the direct output of the
-- RSD-corrected results (test_bilddata_out).
library IEEE;
use IEEE.STD_LOGIC_1164.ALL, IEEE.NUMERIC_STD.all, STD.TEXTIO.all;
use WORK.ALL;

entity comb3 is
port (pixclock          : in  std_logic;
      fastclock         : in  std_logic;
      reset             : in  std_logic;
      input             : in  std_logic_vector(7 downto 0);
      state8_out        : out std_logic_vector(2 downto 0);
      state8_old_out    : out std_logic_vector(2 downto 0);
      state64_out       : out std_logic_vector(5 downto 0);
      quantisation      : in  std_logic;
      quant             : out std_logic_vector(10 downto 0);
      testpic           : in  std_logic;
      line_flag_nojpg   : out std_logic;
      rsd_data_nojpg    : out std_logic_vector(7 downto 0);
      test_bilddata_out : in  std_logic;
      pixclock_test     : out std_logic    );
end entity comb3;

architecture comb3_behavior of comb3 is

component rsd
port (  pixclock      : in  std_logic;
      ADControl_reset : in  std_logic;
      compdata        : in  std_logic_vector(7 downto 0);
      error           : out std_logic_vector(1 downto 0);
      digital         : out std_logic_vector(10 downto 0);
      state8          : out std_logic_vector(2 downto 0);
      state64         : out std_logic_vector(5 downto 0);
      state8_old      : out std_logic_vector(2 downto 0)  );
end component;

component minus128
port (DCT_data : in  std_logic_vector(10 downto 0);
      state64  : in  std_logic_vector(5 downto 0);
      enddata  : out std_logic_vector(10 downto 0)  );
end component;


component division
port (data_todiv   : in  std_logic_vector(10 downto 0);
      state64      : in  std_logic_vector(5 downto 0);
      quant_data   : out std_logic_vector(10 downto 0);
      quantisation : in  std_logic    );
```

```
end component;

-- Wires bitween blocks.
-- output of RSD
signal    raw_data    : std_logic_vector(10 downto 0);
--signal  raw_data_bv    : bit_vector(10 downto 0);
signal    data_to_minus128  : std_logic_vector(10 downto 0);
signal    DCT_enddata    : std_logic_vector(10 downto 0);
signal    RSD_error    : std_logic_vector(1 downto 0);
signal    new_data    : std_logic;
signal    state64      : std_logic_vector(5 downto 0);
signal    state8      : std_logic_vector(2 downto 0);
signal    state8_old    : std_logic_vector(2 downto 0);

signal    raw_data_buf    : std_logic_vector(10 downto 0);
signal    pixclock_minicounter  : unsigned(2 downto 0);
signal    firststart    : std_logic;
signal    pixclock_test_i    : std_logic;

signal    new_data_counter  : unsigned(5 downto 0);
-- output of RSD

signal    bilddata    : std_logic_vector(10 downto 0);

-- This is a test picture. Note: All values are the DCT-transformed values of 8x8 pixels!
type bildarray_type is array(natural range<>) of unsigned(10 downto 0);
constant  bildarray    : bildarray_type(0 to 63):=(
  "00000000000","00000000100","00000001000","00000001100","00000010000","00000010100",
  "00000011000","00000011100","00000100000","00000100100","00000101000","00000101100",
  "00000110000","00000110100","00000111000","00000111100","00001000000","00001000100",
  "00001001000","00001001100","00001010000","00001010100","00001011000","00001011100",
  "00001100000","00001100100","00001101000","00001101100","00001110000","00001110100",
  "00001111000","00001111100","00010000000","00010000100","00010001000","00010001100",
  "00010010000","00010010100","00010011000","00010011100","00010100000","00010100100",
  "00010101000","00010101100","00010110000","00010110100","00010111000","00010111100",
  "00011000000","00011000100","00011001000","00011001100","00011010000","00011010100",
  "00011011000","00011011100","00011100000","00011100100","00011101000","00011101100",
  "00011110000","00011110100","00011111000","00011111100"    );
signal    counter64    : unsigned(5 downto 0);
signal    counter8     : unsigned(2 downto 0);

begin

RSD_Input : rsd
port map(
  pixclock=>pixclock,
  ADControl_reset=>reset,
  compdata=>input,
  error=>RSD_error,
  digital=>raw_data,
  state8=>state8,
  state64=>state64,
  state8_old=>state8_old  );

sub128 : minus128
port map(
  DCT_data=>data_to_minus128,
  state64=>state64,
  enddata=>DCT_enddata  );


mul_div : division
port map(
  data_todiv=>DCT_enddata,
  state64=>state64,
  quant_data=>quant,
  quantisation=>quantisation  );

-- Output of the global state-values (needed by the 2 Huffman stages).
process (state64,state8,state8_old)
begin
state64_out<=state64;
state8_out<=state8;
state8_old_out<=state8_old;
```

```
end process;

-- New data signal for division.vhdl.
process (RSD_error)
begin
case RSD_error is
when "01" =>  new_data<='1';
when "11" =>  new_data<='1';
when others =>  new_data<='0';
end case;
end process;


-- Input data - simulation with the Test-Picture
-- (see the definition of the constant bildarray).
testpicture : process (new_data,reset,counter8,counter64)
variable    bilddata_dummy  : unsigned(10 downto 0);
begin
if (reset='0') then
  counter64<="000000";
  counter8<="000";
elsif rising_edge(new_data) then
  counter8<=counter8+"001";
  if (counter8="111") then
    counter64<=counter64+"000001";
  end if;
  bilddata_dummy:=bildarray(to_integer(counter64));
  for N in 0 to 10 loop
    bilddata(N)<=bilddata_dummy(N);
  end loop;
end if;
end process;

-- Eighter the RSD-corrected results or the testpic will tunnel to the next stage
-- in the JPEG-algorithm.
process (testpic,bilddata,raw_data)
begin
if (testpic='1') then
  data_to_minus128<=bilddata;
else  data_to_minus128<=raw_data;
end if;
end process;


-- These 4 processes provide the data for the direct output without the JPEG-algorithm.
-- It is possible to tunnel eighter the RSD-corrected data or some well-known bits
-- (the bits of the testpic) to output.
rsd_data_to_out1 : process (reset,new_data,bilddata,test_bilddata_out
                            ,raw_data,new_data_counter)
begin
if (reset='0') then
  new_data_counter<="111111";
  raw_data_buf<="00000000000";
elsif (falling_edge(new_data)) then
  if (test_bilddata_out='1') then
    raw_data_buf<=bilddata; -- must be: nojpg='1' to take effect
  else  raw_data_buf<=raw_data; -- normal case
  end if;
  new_data_counter<=new_data_counter+"000001";
end if;
end process rsd_data_to_out1;

rsd_data_to_out2 : process (pixclock_minicounter,pixclock_test_i,pixclock,firststart
                            ,reset,new_data_counter)
begin
if (reset='0') then
  pixclock_minicounter<="000";
  firststart<='1';
  line_flag_nojpg<='0';
  pixclock_test_i<='0';
elsif (rising_edge(pixclock)) then
  if (new_data='1') then
    pixclock_minicounter<="000";
    firststart<='0';
```

```
  elsif (firststart='0') then
    pixclock_minicounter<=pixclock_minicounter+"001";
  end if;

  -- See the documentation for details of this clock-scheme
  -- set lineflag
  if (pixclock_minicounter="000" and new_data_counter="000000") then
    line_flag_nojpg<='1';
  elsif (pixclock_minicounter="100" and new_data_counter="111111") then
    line_flag_nojpg<='0';
  end if;

  -- See the documentation for details of this clock-scheme.
  -- set clock for Puegners byte_to_word-converter
  if (pixclock_minicounter="000"
      and (new_data_counter(0)='0' or new_data_counter="000001")) then
    pixclock_test_i<=not(pixclock_test_i);
  elsif (pixclock_minicounter="001"
      or pixclock_minicounter="010" or pixclock_minicounter="011") then
    pixclock_test_i<=not(pixclock_test_i);
  elsif (pixclock_minicounter="100"
      and (new_data_counter(0)='1' or new_data_counter="000000")) then
    pixclock_test_i<=not(pixclock_test_i);
  end if;

end if;
end process rsd_data_to_out2;

rsd_data_to_out3 : process(pixclock_minicounter,pixclock,reset
                           ,new_data_counter,raw_data_buf)
begin
if reset='0' then
  rsd_data_nojpg<="00000000"; -- not nessesary
elsif (falling_edge(pixclock)) then
  -- See the documentation for details of this clock-scheme..
  -- apply data to output
  if (pixclock_minicounter="100" and new_data_counter="000000") then
    rsd_data_nojpg(2 downto 0)<=raw_data_buf(10 downto 8);
    rsd_data_nojpg(7 downto 3)<="00000";
  elsif (pixclock_minicounter="000") then
    rsd_data_nojpg(7 downto 0)<=raw_data_buf(7 downto 0);
  elsif (pixclock_minicounter="010") then
    rsd_data_nojpg(2 downto 0)<=raw_data_buf(10 downto 8);
    rsd_data_nojpg(7 downto 3)<="00000";
  end if;
end if;
end process rsd_data_to_out3;

rsd_data_to_out4 : process(pixclock_test_i)
begin
pixclock_test<=pixclock_test_i;
end process rsd_data_to_out4;

end architecture comb3_behavior;

configuration comb3_cfg of comb3 is
for comb3_behavior
end for;
end comb3_cfg;
```

9.8 Erste Pufferung und Vorcodierung

buf_mux.vhdl

```
-- This component geneates the long version ofthe final code words.
-- It takes eighter the signal of the DC difference (DC_diff_mux_in)
-- or the value of an AC coefficient (value_mux_in).
-- Furthermore this component calculates the the the category
-- ("SSSS" called in "JPEG: still image data compression standard") of the input.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL, IEEE.NUMERIC_STD.all;
```

```
use WORK.ALL;

entity buf_mux is
port (state_coder_mux_in : in  std_logic_vector(2 downto 0);
      DC_diff_mux_in     : in  std_logic_vector(11 downto 0);
      value_mux_in       : in  std_logic_vector(10 downto 0);
      reset_mux          : in  std_logic;
      category_mux       : out std_logic_vector(3 downto 0);
      coded_long_mux     : out std_logic_vector(11 downto 0);
      fastclock          : in  std_logic  );
end entity buf_mux;


architecture buf_mux_behavior of buf_mux is
signal    coded_long    : signed(11 downto 0);
signal    state_coder_mux    : unsigned(2 downto 0);
signal    DC_diff_mux    : signed(11 downto 0);
signal    value_mux    : signed(10 downto 0);

begin

-- I/O-conversion
process (state_coder_mux_in)
begin
for N in 0 to 2 loop
  state_coder_mux(N)<=state_coder_mux_in(N);
end loop;
end process;
process (DC_diff_mux_in)
begin
for N in 0 to 11 loop
  DC_diff_mux(N)<=DC_diff_mux_in(N);
end loop;
end process;
process (value_mux_in)
begin
for N in 0 to 10 loop
  value_mux(N)<=value_mux_in(N);
end loop;
end process;
--


-- output mux
out_mux : process (state_coder_mux,DC_diff_mux,value_mux,reset_mux,coded_long,fastclock)
-- state_coder_mux: 000=nop, 001=init, 010=abort, 011=runlength, 100=wait_end
variable  coded_long_dummy  : signed(11 downto 0);
variable  coded_long_inv    : signed(11 downto 0);
variable  coded_long_dummy_pos  : signed(11 downto 0);
begin
if (reset_mux='0') then
  coded_long<="000000000000";
  coded_long_dummy:="000000000000";
elsif rising_edge(fastclock) then
  if (state_coder_mux="001") then
    coded_long<=DC_diff_mux; -- take the DC difference as input
    coded_long_dummy:=DC_diff_mux;
  elsif (state_coder_mux="011" or state_coder_mux="010") then
    -- AC-coefficients are only allowed in range [-1023, 1023]
    -- -> -1024 has to be "changed" to -1023
    if (value_mux/="10000000000") then
      -- /= -1024
      coded_long<=resize(value_mux,12);
      -- take an AC coefficient as input
      coded_long_dummy:=resize(value_mux,12);
    else  coded_long<="110000000001";
      -- -1024 leads not to a valid codeword, but -1024 should never appear
      coded_long_dummy:="110000000001";
      -- This change to -1023 is only for safety and for help for the category-calculation.
    end if;
  end if;

  -- get absolute value of coded_long_dummy
  if (coded_long_dummy(11)='1') then
```

```
    for N in 0 to 11 loop
      coded_long_inv(N):=not(coded_long_dummy(N));
    end loop;
    coded_long_dummy_pos:=coded_long_inv+"000000000001";
  else  coded_long_dummy_pos:=coded_long_dummy;
  end if;
  -- "100000000000" = -2048 would  be converted to -2048, but it does not occour

  -- get category (elsif-statements better? (maybe slower))
  if (coded_long_dummy_pos(11 downto 0)="000000000000") then
    -- coded_long=0
    category_mux<="0000";
  end if;
  if (coded_long_dummy_pos(11 downto 0)="000000000001") then
    -- -1 , 1
    category_mux<="0001";
  end if;
  if (coded_long_dummy_pos(11 downto 1)="00000000001") then
    -- -3,-2 , 2,3
    category_mux<="0010";
  end if;
  if (coded_long_dummy_pos(11 downto 2)="0000000001") then
    -- -7 ... -4 , 4 ... 7
    category_mux<="0011";
  end if;
  if (coded_long_dummy_pos(11 downto 3)="000000001") then
    -- -15 ... -8 , 8 ... 15
    category_mux<="0100";
  end if;
  if (coded_long_dummy_pos(11 downto 4)="00000001") then
    -- -31 ... -16 , 16 ... 31
    category_mux<="0101";
  end if;
  if (coded_long_dummy_pos(11 downto 5)="0000001") then
    -- -63 ... -32 , 32 ... 63
    category_mux<="0110";
  end if;
  if (coded_long_dummy_pos(11 downto 6)="000001") then
    -- -127 ... -64 , 64 ... 127
    category_mux<="0111";
  end if;
  if (coded_long_dummy_pos(11 downto 7)="00001") then
    -- -255 ... -128 , 128 ... 255
    category_mux<="1000";
  end if;
  if (coded_long_dummy_pos(11 downto 8)="0001") then
    -- -511 ... -256 , 256 ... 511
    category_mux<="1001";
  end if;
  if (coded_long_dummy_pos(11 downto 9)="001") then
    -- -1023 ... -512 , 512 ... 1023
    category_mux<="1010";
  end if;
  if (coded_long_dummy_pos(11 downto 10)="01") then
    -- -2047 ... -1024 , 1024 ... 2047
    category_mux<="1011";
  end if;
  if (coded_long_dummy_pos(11 downto 0)="100000000000") then
    -- -2048 - will never occour
    category_mux<="1011";
  end if;
end if;
end process out_mux;
-- output mux - end

process(coded_long)
begin
for N in 0 to 11 loop
  coded_long_mux(N)<=coded_long(N);
end loop;
end process;


end architecture buf_mux_behavior;
```

```
configuration buf_mux_cfg of buf_mux is
for buf_mux_behavior
end for;
end buf_mux_cfg;
```

---

data_buf.vhdl

```
-- This stage will will buffer the input data stream and encode 8x8-Huffman-blocks.
-- Xilinx CoreGen
-- synopsys translate_off
Library XilinxCoreLib;
-- synopsys translate_on
-- Xilinx CoreGen

library IEEE;
use IEEE.STD_LOGIC_1164.ALL, IEEE.NUMERIC_STD.all, IEEE.std_logic_unsigned.all;
use WORK.ALL;

entity data_buf is
port (data_in             : in  std_logic_vector(10 downto 0);
      state64_in          : in  std_logic_vector(5 downto 0);
      state8_in           : in  std_logic_vector(2 downto 0);
      state8_in_old       : in  std_logic_vector(2 downto 0);
      pixclock            : in  std_logic;
      fastclock           : in  std_logic;
      reset_huf           : in  std_logic;
      coded_long_out      : out std_logic_vector(11 downto 0);
      category            : out std_logic_vector(3 downto 0);
      state_coder_out     : out std_logic_vector(2 downto 0);
      nextstate_coder_out : out std_logic_vector(2 downto 0);
      run_out             : out std_logic_vector(3 downto 0);
      state_copy          : out std_logic_vector(1 downto 0);
      last_zero_out       : out std_logic_vector(6 downto 0)  );
end entity data_buf;


architecture data_buf_behavior of data_buf is
--I/O-Signals
signal    fastclock_int    : std_logic;
signal    reset_int    : std_logic;
signal    pixclock_int    : std_logic;
signal    state64      : unsigned(5 downto 0);
signal    state8      : unsigned(2 downto 0);
signal    state8_old    : unsigned(2 downto 0);
signal    state_coder    : unsigned(2 downto 0);
signal    nextstate_coder    : unsigned(2 downto 0);
signal    run      : integer range 0 to 15;
signal    run_unsig    : unsigned(3 downto 0);
signal    run_dummy    : integer range 0 to 15;

--type statetype is (nop,first_write1,write1, write2);
-- nop=00, first_write1=01, write1=10, write2=11
signal    state      : std_logic_vector(1 downto 0);
signal    nextstate    : std_logic_vector(1 downto 0);
signal    state_changed    : std_logic;

-- end pix-cache

-- Huffman coder
signal    blocks_in_frame    : unsigned(3 downto 0);
signal    blocks_in_frame_old  : unsigned(3 downto 0);
signal    last_zero    : integer range 0 to 65;
 -- last_zeros of block 1
signal    last_zero1_0    : integer range 0 to 65;
signal    last_zero1_1    : integer range 0 to 65;
signal    last_zero1_2    : integer range 0 to 65;
signal    last_zero1_3    : integer range 0 to 65;
signal    last_zero1_4    : integer range 0 to 65;
signal    last_zero1_5    : integer range 0 to 65;
signal    last_zero1_6    : integer range 0 to 65;
signal    last_zero1_7    : integer range 0 to 65;
```

```
 -- last_zeros of block 2
signal    last_zero2_0    : integer range 0 to 65;
signal    last_zero2_1    : integer range 0 to 65;
signal    last_zero2_2    : integer range 0 to 65;
signal    last_zero2_3    : integer range 0 to 65;
signal    last_zero2_4    : integer range 0 to 65;
signal    last_zero2_5    : integer range 0 to 65;
signal    last_zero2_6    : integer range 0 to 65;
signal    last_zero2_7    : integer range 0 to 65;
signal    last_zero_unsig    : unsigned(6 downto 0);
signal    last_DC      : std_logic_vector(10 downto 0);
signal    DC1_0,DC1_1,DC1_2,DC1_3  : std_logic_vector(10 downto 0);
signal    DC1_4,DC1_5,DC1_6,DC1_7 : std_logic_vector(10 downto 0);
signal    DC2_0,DC2_1,DC2_2,DC2_3  : std_logic_vector(10 downto 0);
signal    DC2_4,DC2_5,DC2_6,DC2_7  : std_logic_vector(10 downto 0);
signal    DC_diff      : signed(11 downto 0);
signal    init_state    : std_logic;
signal    code_state    : integer range 0 to 31;
signal    position    : integer range 1 to 64;
signal    found_next    : std_logic;
signal    value      : std_logic_vector(10 downto 0);
signal    value_dummy    : std_logic_vector(10 downto 0);
signal    address_offset    : std_logic_vector(9 downto 6);
signal    counter0,counter1  : integer range 0 to 65;
signal    counter2,counter3  : integer range 0 to 65;
signal    counter4,counter5  : integer range 0 to 65;
signal    counter6,counter7  : integer range 0 to 65;
signal    counter_reset    : std_logic;

-- conversion-signals
signal    state_coder_slv    : std_logic_vector(2 downto 0);
signal    DC_diff_slv    : std_logic_vector(11 downto 0);

-- Xilinx RAM-module
signal    address_a    : std_logic_vector(9 downto 0);
signal    data_in_a    : std_logic_VECTOR(10 downto 0);
signal    write_en_a    : std_logic;
signal    data_out_a    : std_logic_VECTOR(10 downto 0);
signal    address_b    : std_logic_vector(9 downto 0);
signal    data_in_b    : std_logic_VECTOR(10 downto 0);
signal    write_en_b    : std_logic;
signal    data_out_b    : std_logic_VECTOR(10 downto 0);


-- This component chooses eigther the DC difference or an AC coefficient
-- and results the category of the value.
component buf_mux
port (state_coder_mux_in : in  std_logic_vector(2 downto 0);
      DC_diff_mux_in     : in  std_logic_vector(11 downto 0);
      value_mux_in       : in  std_logic_vector(10 downto 0);
      reset_mux          : in  std_logic;
      category_mux       : out std_logic_vector(3 downto 0);
      coded_long_mux     : out std_logic_vector(11 downto 0);
      fastclock          : in  std_logic   );
end component;

-- Memory: 8x8 block (11 bit depth)
--         8 blocks = row
--         2 rows
-- 1 row is used for buffering the input data stream
-- the other row provides the data for encoding.
-- Xilinx CoreGen
component BlockMem_dual
  port (
  addra: IN std_logic_VECTOR(9 downto 0);
  clka: IN std_logic;
  addrb: IN std_logic_VECTOR(9 downto 0);
  clkb: IN std_logic;
  dia: IN std_logic_VECTOR(10 downto 0);
  wea: IN std_logic;
  dib: IN std_logic_VECTOR(10 downto 0);
  web: IN std_logic;
  doa: OUT std_logic_VECTOR(10 downto 0);
  dob: OUT std_logic_VECTOR(10 downto 0));
```

```
end component;

-- FPGA Express Black Box declaration
attribute fpga_dont_touch: string;
attribute fpga_dont_touch of BlockMem_dual: component is "true";
-- Xilinx CoreGen


begin

reset_int<=reset_huf;
fastclock_int<=fastclock;
pixclock_int<=pixclock;

data_buf_mux : buf_mux
port map(
  state_coder_mux_in=>state_coder_slv,
  DC_diff_mux_in=>DC_diff_slv,
  value_mux_in=>value,
  reset_mux=>reset_int,
  category_mux=>category,
  coded_long_mux=>coded_long_out,
  fastclock=>fastclock_int  );

-- Xilinx CoreGen
in_buffer : BlockMem_dual
    port map (
      addra => address_a,
      clka => fastclock_int,
      addrb => address_b,
      clkb => fastclock_int,
      dia => data_in_a,
      wea => write_en_a,
      dib => data_in_b,
      web => write_en_b,
      doa => data_out_a,
      dob => data_out_b);
-- Xilinx CoreGen

-- I/O-Conversion
-- for lower component data_buf_mux
process (state_coder)
begin
for N in 0 to 2 loop
  state_coder_slv(N)<=state_coder(N);
end loop;
end process;
process (DC_diff)
begin
for N in 0 to 11 loop
  DC_diff_slv(N)<=DC_diff(N);
end loop;
end process;
--
IO_convert1 : process (state64_in,state8_in,state8_in_old,state_coder,nextstate_coder,run)
begin
for N in 0 to 5 loop
  state64(N)<=state64_in(N);
end loop;
for O in 0 to 2 loop
  state8(O)<=state8_in(O);
  state8_old(O)<=state8_in_old(O);
  state_coder_out(O)<=state_coder(O);
  nextstate_coder_out(O)<=nextstate_coder(O);
end loop;
run_unsig<=to_unsigned(run,4);
end process IO_convert1;

IO_convert2 : process(run_unsig)
begin
for M in 0 to 3 loop
  run_out(M)<=run_unsig(M);
end loop;
end process IO_convert2;
```

```
process(last_zero)
begin
last_zero_unsig<=to_unsigned(last_zero,7);
end process;

process (last_zero_unsig)
begin
for N in 0 to 6 loop
  last_zero_out(N)<=last_zero_unsig(N);
end loop;
end process;
-- I/O-Conversion


-- state machine of the pix-cache
change_state : process(reset_int,state64,nextstate,state_changed,fastclock_int)
begin
if (reset_int='0') then
  state<="00";
  state_changed<='0';
elsif rising_edge(fastclock_int) then
  if (state64="000000" and state_changed='0') then
    state<=nextstate;
    state_changed<='1';
  elsif (state64/="000000") then
    state_changed<='0';
  end if;
end if;
end process change_state;

data_to_block : process (reset_int,state,state64,last_zero1_0,last_zero1_1,Last_zero1_2,
                         last_zero1_3,last_zero1_4,last_zero1_5,last_zero1_6,last_zero1_7,
                         last_zero2_0,last_zero2_1,last_zero2_2,last_zero2_3,last_zero2_4,
                         last_zero2_5,last_zero2_6,last_zero2_7,DC1_0,DC1_1,DC1_2,DC1_3,
                         DC1_4,DC1_5,DC1_6,DC1_7,DC2_0,DC2_1,DC2_2,DC2_3,DC2_4,DC2_5,DC2_6,
                         DC2_7,blocks_in_frame,fastclock_int)
-- state: nop=00, first_write1=01, write1=10, write2=11
-- The input data is written alternating to the 1st or to the 2nd row.
begin
if (reset_int='0') then
  last_DC<="00000000000";
  last_zero<=0;
  address_offset<="0000";
  -- address_offset points into the memory block to all 2x8x(8x8) - blocks.
elsif rising_edge(fastclock_int) then
--else
  case state is
  when "00" =>  nextstate<="01";
  when "01" =>  nextstate<="11";
  when "11" =>  nextstate<="10";
      -- Provide everytime the actual last_zero- and Last_DC- value
      case state64(5 downto 3) is
      when "000" =>  last_zero<=last_zero1_0;
                       if (blocks_in_frame="0001" and state64="000000") then
                         -- reset for every new frame
                         last_DC<="00000000000";
                       else  last_DC<=DC2_7;
                       end if;
                       address_offset<="0000";
      when "001" =>  last_zero<=last_zero1_1;
                     last_DC<=DC1_0;
                     address_offset<="0001"; --64
      when "010" =>  last_zero<=last_zero1_2;
                     last_DC<=DC1_1;
                     address_offset<="0010"; --128
      when "011" =>  last_zero<=last_zero1_3;
                     last_DC<=DC1_2;
                     address_offset<="0011"; --192
      when "100" =>  last_zero<=last_zero1_4;
                     last_DC<=DC1_3;
                     address_offset<="0100"; --256
      when "101" =>  last_zero<=last_zero1_5;
                     last_DC<=DC1_4;
```

```
                    address_offset<="0101"; --320
      when "110" =>  last_zero<=last_zero1_6;
                     last_DC<=DC1_5;
                     address_offset<="0110"; --384
      when "111" =>  last_zero<=last_zero1_7;
                     last_DC<=DC1_6;
                     address_offset<="0111"; --448
      when others =>  null;
      end case;
  when "10" =>  nextstate<="11";
      case state64(5 downto 3) is
      when "000" =>  last_zero<=last_zero2_0;
                     if (blocks_in_frame="0001" and state64="000000") then
                       -- reset for every new frame
                       last_DC<="00000000000";
                     else  last_DC<=DC1_7;
                     end if;
                     address_offset<="1000"; --512
      when "001" =>  last_zero<=last_zero2_1;
                     last_DC<=DC2_0;
                     address_offset<="1001"; --576
      when "010" =>  last_zero<=last_zero2_2;
                     last_DC<=DC2_1;
                     address_offset<="1010"; --640
      when "011" =>  last_zero<=last_zero2_3;
                     last_DC<=DC2_2;
                     address_offset<="1011"; --704
      when "100" =>  last_zero<=last_zero2_4;
                     last_DC<=DC2_3;
                     address_offset<="1100"; --768
      when "101" =>  last_zero<=last_zero2_5;
                     last_DC<=DC2_4;
                     address_offset<="1101"; --832
      when "110" =>  last_zero<=last_zero2_6;
                     last_DC<=DC2_5;
                     address_offset<="1110"; --896
      when "111" =>  last_zero<=last_zero2_7;
                     last_DC<=DC2_6;
                     address_offset<="1111"; --960
      when others =>  null;
      end case;
      -- 8 states time for every 8x8-block to encode
  when others =>  nextstate<="00";
  end case;
end if;
end process data_to_block;

counter_last_zero_reset : process(reset_int,pixclock_int,blocks_in_frame_old
                                  ,blocks_in_frame)
begin
if (reset_int='0') then
  blocks_in_frame_old<="0000";
  counter_reset<='0';
elsif falling_edge(pixclock_int) then
    if (blocks_in_frame/=blocks_in_frame_old) then
      counter_reset<='1';
      blocks_in_frame_old<=blocks_in_frame;
    else  counter_reset<='0';
    end if;
end if;
end process counter_last_zero_reset;

data_into_block : process (pixclock_int,reset_int,state,data_in,state8,state8_old,state64,
                           counter_reset,counter0,counter1,counter2,counter3,counter4,
                           counter5,counter6,counter7)
-- state: nop=00, first_write1=01, write1=10, write2=11
variable    address_unsig    : unsigned(5 downto 0);
begin
if (reset_int='0') then
  write_en_a<='1';
  address_a<="0000000000";
  write_en_b<='0';
  DC1_0<="00000000000";
  DC1_1<="00000000000";
```

```
  DC1_2<="00000000000";
  DC1_3<="00000000000";
  DC1_4<="00000000000";
  DC1_5<="00000000000";
  DC1_6<="00000000000";
  DC1_7<="00000000000";
  DC2_0<="00000000000";
  DC2_1<="00000000000";
  DC2_2<="00000000000";
  DC2_3<="00000000000";
  DC2_4<="00000000000";
  DC2_5<="00000000000";
  DC2_6<="00000000000";
  DC2_7<="00000000000";
elsif rising_edge(pixclock_int) then
  write_en_a<='1';
  write_en_b<='0';
  if (counter_reset='1') then
    --counter0<=0; -- counter_reset is too late for counter0
    counter1<=0;
    counter2<=0;
    counter3<=0;
    counter4<=0;
    counter5<=0;
    counter6<=0;
    counter7<=0;
    if (state="11") then
      last_zero2_0<=65;
      last_zero2_1<=65;
      last_zero2_2<=65;
      last_zero2_3<=65;
      last_zero2_4<=65;
      last_zero2_5<=65;
      last_zero2_6<=65;
      last_zero2_7<=65;
    else  last_zero1_0<=65;
      last_zero1_1<=65;
      last_zero1_2<=65;
      last_zero1_3<=65;
      last_zero1_4<=65;
      last_zero1_5<=65;
      last_zero1_6<=65;
      last_zero1_7<=65;
    end if;
  end if;
  -- Sort the input data to the correct position.
  -- Save the last_zero-value. (last_zero points to the index number of the last
  -- zero-value within an 8x8-block, that is only followed by zeros.)
  -- Save the actual DC-coefficient for calculation the DC-difference.
  -- The counter-signals provide information of the index within an 8x8 block.
  if (state8_old/=state8) then
    case state8 is
    when "000" =>  if (state="01" or state="10") then
                     if (counter_reset='1') then
                       address_a<="0000000000";
                       counter0<=1;
                       DC1_0<=data_in;
                     else  address_unsig:=to_unsigned(counter0,6);
                       for N in 0 to 5 loop
                         address_a(N)<=address_unsig(N);
                       end loop;
                       address_a(9 downto 6)<="0000";
                       counter0<=counter0+1;
                     end if;
                     data_in_a<=data_in;
                     if (data_in/="00000000000") then
                       if (counter_reset='1') then
                         last_zero1_0<=1;
                       else  last_zero1_0<=counter0+1;
                       end if;
                     end if;
                   elsif (state="11") then
                     if (counter_reset='1') then
                       address_a<="1000000000"; -- 512, 2nd row
```

```
                    counter0<=1;
                    DC2_0<=data_in;
                  else  address_unsig:=to_unsigned(counter0,6);
                    for N in 0 to 5 loop
                      address_a(N)<=address_unsig(N);
                    end loop;
                    address_a(9 downto 6)<="1000"; -- +512
                    counter0<=counter0+1;
                  end if;
                  data_in_a<=data_in;
                  if (data_in/="00000000000") then
                    if (counter_reset='1') then
                      last_zero2_0<=1;
                    else  last_zero2_0<=counter0+1;
                    end if;
                  end if;
                end if;
when "001" =>  if (state="01" or state="10") then
                  address_unsig:=to_unsigned(counter1,6);
                  for N in 0 to 5 loop
                    address_a(N)<=address_unsig(N);
                  end loop;
                  address_a(9 downto 6)<="0001"; -- +64, 1st row, 2nd block
                  data_in_a<=data_in;
                  if (counter1=0) then
                    DC1_1<=data_in;
                  end if;
                  if (data_in/="00000000000") then
                    last_zero1_1<=counter1+1;
                  end if;
                  counter1<=counter1+1;
                elsif (state="11") then
                  address_unsig:=to_unsigned(counter1,6);
                  for N in 0 to 5 loop
                    address_a(N)<=address_unsig(N);
                  end loop;
                  address_a(9 downto 6)<="1001"; -- +64 +512, 2nd row
                  data_in_a<=data_in;
                  if (counter1=0) then
        DC2_1<=data_in;
                  end if;
                  if (data_in/="00000000000") then
                    last_zero2_1<=counter1+1;
                  end if;
                  counter1<=counter1+1;
                end if;
when "010" =>  if (state="01" or state="10") then
                  address_unsig:=to_unsigned(counter2,6);
                  for N in 0 to 5 loop
                    address_a(N)<=address_unsig(N);
                  end loop;
                  address_a(9 downto 6)<="0010"; -- +128, 1st row, 3rd block
                  data_in_a<=data_in;
                  if (counter2=0) then
                    DC1_2<=data_in;
                  end if;
                  if (data_in/="00000000000") then
                    last_zero1_2<=counter2+1;
                  end if;
                  counter2<=counter2+1;
                elsif (state="11") then
                  address_unsig:=to_unsigned(counter2,6);
                  for N in 0 to 5 loop
                    address_a(N)<=address_unsig(N);
                  end loop;
                  address_a(9 downto 6)<="1010"; -- +128 +512
                  data_in_a<=data_in;
                  if (counter2=0) then
                    DC2_2<=data_in;
                  end if;
                  if (data_in/="00000000000") then
                    last_zero2_2<=counter2+1;
                  end if;
                  counter2<=counter2+1;
```

```
                end if;
when "011" =>   if (state="01" or state="10") then
                  address_unsig:=to_unsigned(counter3,6);
                  for N in 0 to 5 loop
                    address_a(N)<=address_unsig(N);
                  end loop;
                  address_a(9 downto 6)<="0011"; -- +192
                  data_in_a<=data_in;
                  if (counter3=0) then
                    DC1_3<=data_in;
                  end if;
                  if (data_in/="00000000000") then
                    last_zero1_3<=counter3+1;
                  end if;
                  counter3<=counter3+1;
                elsif (state="11") then
                  address_unsig:=to_unsigned(counter3,6);
                  for N in 0 to 5 loop
                    address_a(N)<=address_unsig(N);
                  end loop;
                  address_a(9 downto 6)<="1011"; -- +192 +512
                  data_in_a<=data_in;
                  if (counter3=0) then
                    DC2_3<=data_in;
                  end if;
                  if (data_in/="00000000000") then
                    last_zero2_3<=counter3+1;
                  end if;
                  counter3<=counter3+1;
                end if;
when "100" =>   if (state="01" or state="10") then
                  address_unsig:=to_unsigned(counter4,6);
                  for N in 0 to 5 loop
                    address_a(N)<=address_unsig(N);
                  end loop;
                  address_a(9 downto 6)<="0100"; -- +256
                  data_in_a<=data_in;
                  if (counter4=0) then
                    DC1_4<=data_in;
                  end if;
                  if (data_in/="00000000000") then
                    last_zero1_4<=counter4+1;
                  end if;
                  counter4<=counter4+1;
                elsif (state="11") then
                  address_unsig:=to_unsigned(counter4,6);
                  for N in 0 to 5 loop
                    address_a(N)<=address_unsig(N);
                  end loop;
                  address_a(9 downto 6)<="1100"; -- +256 +512
                  data_in_a<=data_in;
                  if (counter4=0) then
                    DC2_4<=data_in;
                  end if;
                  if (data_in/="00000000000") then
                    last_zero2_4<=counter4+1;
                  end if;
                  counter4<=counter4+1;
                end if;
when "101" =>   if (state="01" or state="10") then
                  address_unsig:=to_unsigned(counter5,6);
                  for N in 0 to 5 loop
                    address_a(N)<=address_unsig(N);
                  end loop;
                  address_a(9 downto 6)<="0101"; -- +320
                  data_in_a<=data_in;
                  if (counter5=0) then
                    DC1_5<=data_in;
                  end if;
                  if (data_in/="00000000000") then
                    last_zero1_5<=counter5+1;
                  end if;
                  counter5<=counter5+1;
                elsif (state="11") then
```

```
                address_unsig:=to_unsigned(counter5,6);
                for N in 0 to 5 loop
                  address_a(N)<=address_unsig(N);
                end loop;
                address_a(9 downto 6)<="1101"; -- +320 +512
                data_in_a<=data_in;
                if (counter5=0) then
                  DC2_5<=data_in;
                end if;
                if (data_in/="00000000000") then
                  last_zero2_5<=counter5+1;
                end if;
                counter5<=counter5+1;
              end if;
when "110" =>  if (state="01" or state="10") then
                address_unsig:=to_unsigned(counter6,6);
                for N in 0 to 5 loop
                  address_a(N)<=address_unsig(N);
                end loop;
                address_a(9 downto 6)<="0110"; -- +384
                data_in_a<=data_in;
                if (counter6=0) then
                  DC1_6<=data_in;
                end if;
                if (data_in/="00000000000") then
                  last_zero1_6<=counter6+1;
                end if;
                counter6<=counter6+1;
              elsif (state="11") then
                address_unsig:=to_unsigned(counter6,6);
                for N in 0 to 5 loop
                  address_a(N)<=address_unsig(N);
                end loop;
                address_a(9 downto 6)<="1110"; -- +384 +512
                data_in_a<=data_in;
                if (counter6=0) then
                  DC2_6<=data_in;
                end if;
                if (data_in/="00000000000") then
                  last_zero2_6<=counter6+1;
                end if;
                counter6<=counter6+1;
              end if;
when "111" =>  if (state="01" or state="10") then
                address_unsig:=to_unsigned(counter7,6);
                for N in 0 to 5 loop
                  address_a(N)<=address_unsig(N);
                end loop;
                address_a(9 downto 6)<="0111"; -- +448
                data_in_a<=data_in;
                if (counter7=0) then
                  DC1_7<=data_in;
                end if;
                if (data_in/="00000000000") then
                  last_zero1_7<=counter7+1;
                end if;
                counter7<=counter7+1;
              elsif (state="11") then
                address_unsig:=to_unsigned(counter7,6);
                for N in 0 to 5 loop
                  address_a(N)<=address_unsig(N);
                end loop;
                address_a(9 downto 6)<="1111"; -- -- +448 +512
                data_in_a<=data_in;
                if (counter7=0) then
                  DC2_7<=data_in;
                end if;
                if (data_in/="00000000000") then
                  last_zero2_7<=counter7+1;
                end if;
                counter7<=counter7+1;
              end if;
when others => null;
end case;
```

```
  end if;
--end if;
end if;
end process data_into_block;

-- end of saving data into the memory block


-- Find markers for begin and end of frame and rows.
markers : process (state64,reset_int,blocks_in_frame)
begin
if (reset_int='0') then
  blocks_in_frame<="1111";
elsif falling_edge(state64(5)) then
  if (blocks_in_frame="1111") then --blocks_in_frame=128;
    blocks_in_frame<="0000"; --8;
  else blocks_in_frame<=blocks_in_frame+"0001"; --+8;
  end if;
end if;
end process markers;
-- find markers for begin and end of frame and rows - end
```

Bis hierhin sind alle benötigten Informationen gesammelt. Die DC-Werte sind separat gespeichert, die AC-Werte befinden sich an der korrekten Position im RAM und der letzte nicht-Null-Wert eines jeden 8×8-Blockes wurde bestimmt. Die folgenden Prozesse beschräftigen sich mit dem Auslesen aus dem RAM und der Codierung.

```
-- FSM coder
FSM_coder : process (pixclock_int,reset_int)
begin
if (reset_int='0') then
  state_coder<="000";
elsif falling_edge(pixclock_int) then
    state_coder<=nextstate_coder;
end if;
end process FSM_coder;


coder : process (state_coder,state64,state,position,last_zero,last_DC,
                 found_next,reset_int,fastclock_int,init_state,code_state,
                 address_offset,counter_reset)
-- state_coder: 000=nop, 001=init, 010=runlength, 011=out,
--              100=wait_end, 101=enc_EOB_EOI, 110=waitstate
-- state: nop=00, first_write1=01, write1=10, write2=11
variable    address_dummy    : unsigned(6 downto 0);
-- 2^6 is dummy because position is larger than 63 (theoretically)
variable    dummy_data    : signed(10 downto 0);
variable    last_DC_sig    : signed(10 downto 0);
begin
if (reset_int='0') then
  DC_diff<=to_signed(0,12);
  found_next<='0';
  address_b<="1111111111"; -- no collision with address_a after reset
elsif rising_edge(fastclock_int) then
  case state_coder is
  when "000" =>   nextstate_coder<="000";
                  if (state64(2 downto 0)="000" and (state="10" or state="11")) then
                    -- every 8th state
                    nextstate_coder<="001";
                  else  nextstate_coder<="000";
                  end if;
                  value<="00000000000"; -- init
                  run<=0;
                  init_state<='0';
                  position<=1;
                  -- Position is an index in the 8x8 block.
                  -- It shows at whitch point the coding algorithm is working.
                  found_next<='0';
                  code_state<=0;
  when "001" =>  if (init_state='0') then
```

```
                nextstate_coder<="001";
                address_b(9 downto 6)<=address_offset(9 downto 6);
                address_b(5 downto 0)<="000000";
              else  nextstate_coder<="010";
                for N in 0 to 10 loop
                  dummy_data(N):=data_out_b(N);
                  last_DC_sig(N):=last_DC(N);
                end loop;
                DC_diff<=resize((dummy_data - last_DC_sig),12);
                -- Calculate the actual DC-difference.
              end if;
              init_state<='1';
when "010" =>  -- See clock scheme in the documentation for details.
              -- The runlength of zeros between non-zero values will be calculated.
              -- Every runlength-calculation needs a single cycle
              -- in the state_coder="010". (-> max 63 cycles)
              case code_state is
              -- code_state is a counter, that split a clock-cycle of the pixclock
              -- into several parts with the help of fastclock.
              when 0 =>  if (position=last_zero or last_zero=65) then
                           nextstate_coder<="101";
                         else  address_dummy:=to_unsigned(position,7);
                           for N in 0 to 5 loop
                             address_b(N)<=address_dummy(N);
                           end loop;
                           address_b(9 downto 6)<=address_offset(9 downto 6);
                           -- +address_offset
                           code_state<=code_state+1;
                         end if;
              when 1 =>  if (position+1<65) then
                           address_dummy:=to_unsigned(position+1,7);
                           for N in 0 to 5 loop
                             address_b(N)<=address_dummy(N);
                           end loop;
                           code_state<=code_state+1;
                           address_b(9 downto 6)<=address_offset(9 downto 6);
                           -- +address_offset
                         end if;
              when 2 =>  if (data_out_b/="00000000000" and found_next='0') then
                           run_dummy<=0;
                           value_dummy<=data_out_b;
                           position<=position+1;
                           found_next<='1';
                         end if;
                         code_state<=code_state+1;
                         if (position+2<65) then
                           address_dummy:=to_unsigned(position+2,7);
                           for N in 0 to 5 loop
                             address_b(N)<=address_dummy(N);
                           end loop;
                           address_b(9 downto 6)<=address_offset(9 downto 6);
                           -- +address_offset
                         end if;
              when 3 =>  if (data_out_b/="00000000000" and found_next='0') then
                           run_dummy<=1;
                           value_dummy<=data_out_b;
                           position<=position+2;
                           found_next<='1';
                         end if;
                         code_state<=code_state+1;
                         if (position+3<65) then
                           address_dummy:=to_unsigned(position+3,7);
                           for N in 0 to 5 loop
                             address_b(N)<=address_dummy(N);
                           end loop;
                           address_b(9 downto 6)<=address_offset(9 downto 6);
                           -- +address_offset
                         end if;
              when 4 =>  if (data_out_b/="00000000000" and found_next='0') then
                           run_dummy<=2;
                           value_dummy<=data_out_b;
                           position<=position+3;
                           found_next<='1';
                         end if;
```

```
            code_state<=code_state+1;
            if (position+4<65) then
              address_dummy:=to_unsigned(position+4,7);
              for N in 0 to 5 loop
                address_b(N)<=address_dummy(N);
              end loop;
              address_b(9 downto 6)<=address_offset(9 downto 6);
              -- +address_offset
            end if;
 when 5 =>  if (data_out_b/="00000000000" and found_next='0') then
              run_dummy<=3;
              value_dummy<=data_out_b;
              position<=position+4;
              found_next<='1';
            end if;
            code_state<=code_state+1;
            if (position+5<65) then
              address_dummy:=to_unsigned(position+5,7);
              for N in 0 to 5 loop
                address_b(N)<=address_dummy(N);
              end loop;
              address_b(9 downto 6)<=address_offset(9 downto 6);
              -- +address_offset
            end if;
 when 6 =>  if (data_out_b/="00000000000" and found_next='0') then
             run_dummy<=4;
             value_dummy<=data_out_b;
             position<=position+5;
             found_next<='1';
            end if;
            code_state<=code_state+1;
            if (position+6<65) then
              address_dummy:=to_unsigned(position+6,7);
              for N in 0 to 5 loop
                address_b(N)<=address_dummy(N);
              end loop;
              address_b(9 downto 6)<=address_offset(9 downto 6);
               -- +address_offset
            end if;
when 7 =>  if (data_out_b/="00000000000" and found_next='0') then
              run_dummy<=5;
              value_dummy<=data_out_b;
              position<=position+6;
              found_next<='1';
            end if;
            code_state<=code_state+1;
            if (position+7<65) then
              address_dummy:=to_unsigned(position+7,7);
              for N in 0 to 5 loop
                address_b(N)<=address_dummy(N);
              end loop;
              address_b(9 downto 6)<=address_offset(9 downto 6);
               -- +address_offset
            end if;
when 8 =>  if (data_out_b/="00000000000" and found_next='0') then
             run_dummy<=6;
             value_dummy<=data_out_b;
             position<=position+7;
             found_next<='1';
           end if;
           code_state<=code_state+1;
           if (position+8<65) then
             address_dummy:=to_unsigned(position+8,7);
             for N in 0 to 5 loop
               address_b(N)<=address_dummy(N);
             end loop;
             address_b(9 downto 6)<=address_offset(9 downto 6);
               -- +address_offset
           end if;
when 9 =>  if (data_out_b/="00000000000" and found_next='0') then
             run_dummy<=7;
             value_dummy<=data_out_b;
             position<=position+8;
             found_next<='1';
```

```
            end if;
            code_state<=code_state+1;
            if (position+9<65) then
              address_dummy:=to_unsigned(position+9,7);
              for N in 0 to 5 loop
                address_b(N)<=address_dummy(N);
              end loop;
              address_b(9 downto 6)<=address_offset(9 downto 6);
                -- +address_offset
            end if;
when  10 =>  if (data_out_b/="00000000000" and found_next='0') then
              run_dummy<=8;
              value_dummy<=data_out_b;
              position<=position+9;
              found_next<='1';
            end if;
            code_state<=code_state+1;
            if (position+10<65) then
              address_dummy:=to_unsigned(position+10,7);
              for N in 0 to 5 loop
                address_b(N)<=address_dummy(N);
              end loop;
              address_b(9 downto 6)<=address_offset(9 downto 6);
                -- +address_offset
            end if;
when  11 =>  if (data_out_b/="00000000000" and found_next='0') then
              run_dummy<=9;
              value_dummy<=data_out_b;
              position<=position+10;
              found_next<='1';
            end if;
            code_state<=code_state+1;
            if (position+11<65) then
              address_dummy:=to_unsigned(position+11,7);
              for N in 0 to 5 loop
                address_b(N)<=address_dummy(N);
              end loop;
              address_b(9 downto 6)<=address_offset(9 downto 6);
                -- +address_offset
            end if;
when  12 =>  if (data_out_b/="00000000000" and found_next='0') then
              run_dummy<=10;
              value_dummy<=data_out_b;
              position<=position+11;
              found_next<='1';
            end if;
            code_state<=code_state+1;
            if (position+12<65) then
              address_dummy:=to_unsigned(position+12,7);
              for N in 0 to 5 loop
                address_b(N)<=address_dummy(N);
              end loop;
              address_b(9 downto 6)<=address_offset(9 downto 6);
                -- +address_offset
            end if;
when  13 =>  if (data_out_b/="00000000000" and found_next='0') then
              run_dummy<=11;
              value_dummy<=data_out_b;
              position<=position+12;
              found_next<='1';
            end if;
            code_state<=code_state+1;
            if (position+13<65) then
              address_dummy:=to_unsigned(position+13,7);
              for N in 0 to 5 loop
                address_b(N)<=address_dummy(N);
              end loop;
              address_b(9 downto 6)<=address_offset(9 downto 6);
                -- +address_offset
            end if;
when  14 =>  if (data_out_b/="00000000000" and found_next='0') then
              run_dummy<=12;
              value_dummy<=data_out_b;
              position<=position+13;
```

```
                        found_next<='1';
                      end if;
                      code_state<=code_state+1;
                      if (position+14<65) then
                        address_dummy:=to_unsigned(position+14,7);
                        for N in 0 to 5 loop
                          address_b(N)<=address_dummy(N);
                        end loop;
                        address_b(9 downto 6)<=address_offset(9 downto 6);
                          -- +address_offset
                      end if;
           when  15 =>  if (data_out_b/="00000000000" and found_next='0') then
                        run_dummy<=13;
                        value_dummy<=data_out_b;
                        position<=position+14;
                        found_next<='1';
                      end if;
                      code_state<=code_state+1;
                      if (position+15<65) then
                        address_dummy:=to_unsigned(position+15,7);
                        for N in 0 to 5 loop
                          address_b(N)<=address_dummy(N);
                        end loop;
                        address_b(9 downto 6)<=address_offset(9 downto 6);
                          -- +address_offset
                      end if;
           when  16 =>  if (data_out_b/="00000000000" and found_next='0') then
                        run_dummy<=14;
                        value_dummy<=data_out_b;
                        position<=position+15;
                        found_next<='1';
                      end if;
                      code_state<=code_state+1;
                      nextstate_coder<="011";
         when others => nextstate_coder<="011";
         end case;
  when "011" =>  -- Finish the runlength-calculation an move the correct data to output.
                 case code_state is
                 when  17 =>  if (data_out_b/="00000000000" and found_next='0') then
                                run<=15;
                                value<=data_out_b;
                                position<=position+16;
                                --found_next<='1';
                              elsif (found_next='0') then
                                --ZRL
                                run<=15;
                                value<="00000000000";
                                position<=position+16;
                              else  run<=run_dummy;
                                value<=value_dummy;
                              end if;
                              code_state<=0;
                              found_next<='0';
                              nextstate_coder<="010";
              when others => nextstate_coder<="010";
              end case;
  when "101" =>  -- a state reserved for EOB + EOI - generation in huffman.vhdl
                 nextstate_coder<="110";
  when "110" =>  nextstate_coder<="100";
                 -- simple wait-state
  when "100" =>  if (state64(0)='1') then
                   nextstate_coder<="000";
                 else  nextstate_coder<="100";
                 end if;
                 -- wait_end - state for running encode-process only once per block
                 -- (protection against strating from nop while state64 did not change)
  when others =>  nextstate_coder<="000";
  end case;
end if;
end process coder;
-- FSM coder - end

-- Following information is needed by huf_coder (the next stage).
process (state)
```

```
begin
state_copy<=state;
end process;
-- testoutput


end architecture data_buf_behavior;

configuration data_buf_cfg of data_buf is
for data_buf_behavior
-- XilinxCoreGen
-- synopsys translate_off

  for all : BlockMem_dual use entity XilinxCoreLib.C_MEM_DP_BLOCK_V1_0(behavioral)
    generic map(
      c_depth_b => 1024,
      c_depth_a => 1024,
      c_has_web => 1,
      c_has_wea => 1,
      c_has_dib => 1,
      c_has_dia => 1,
      c_clka_polarity => 1,
      c_web_polarity => 1,
      c_address_width_b => 10,
      c_address_width_a => 10,
      c_width_b => 11,
      c_width_a => 11,
      c_clkb_polarity => 1,
      c_ena_polarity => 1,
      c_rsta_polarity => 1,
      c_has_rstb => 0,
      c_has_rsta => 0,
      c_read_mif => 0,
      c_enb_polarity => 1,
      c_pipe_stages => 0,
      c_rstb_polarity => 1,
      c_has_enb => 0,
      c_has_ena => 0,
      c_mem_init_radix => 16,
      c_default_data => "0",
      c_mem_init_file => "/home/eeebs2/hilde/synopsys/jpeg/SRC/data_buf/BlockMem_dual.mif",
      c_has_dob => 1,
      c_generate_mif => 1,
      c_has_doa => 1,
      c_wea_polarity => 1);
  end for;

-- synopsys translate_on
-- XilinxCoreGen
end for;
end data_buf_cfg;
```

9.9 Komplettierung der Codierung und Ausgangspufferung

huf_help.vhdl

```
-- This component was written because of finding an error during coding.
-- Some code, that could be cut out of huf_coder was put into this
-- listing to minimize the error sources.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL, IEEE.NUMERIC_STD.all;
use WORK.ALL;

entity huf_help is
port (coded_long_hlp         : in  std_logic_vector(11 downto 0);
      coded_long_dec_slv_hlp : out std_logic_vector(11 downto 0)  );
end entity huf_help;

architecture huf_help_behavior of huf_help is
signal    coded_long_dec    : signed(11 downto 0);
```

```
signal    state_coder_hlp    : unsigned(2 downto 0);
signal    coded_long_hlp_sig  : signed(11 downto 0);

begin
-- I/O conversion
process (state_coder_hlp,coded_long_hlp)
begin
  for N in 0 to 11 loop
    coded_long_hlp_sig(N)<=coded_long_hlp(N);
  end loop;
end process;

process (coded_long_hlp_sig)
begin
coded_long_dec<=coded_long_hlp_sig-1; -- a simple decrementer
end process;

-- I/O conversion
process (coded_long_dec)
begin
for N in 0 to 11 loop
  coded_long_dec_slv_hlp(N)<=coded_long_dec(N);
end loop;
end process;

end architecture huf_help_behavior;

configuration huf_help_cfg of huf_help is
for huf_help_behavior
end for;
end huf_help_cfg;
```

---

hufman.vhdl

```
-- Final stage of the huffman coding process. The Huffman codewords
-- (coded_long) will be saved in a memory buffer (out_buf).
-- coded_long has fixed bitwidth.
-- Information about the needed length of the codewords are written
-- to a 2nd memory buffer (out_buf_length).
-- After all the output behavoir is described. (Exaktly 8 bits must
-- be given out, but the memory fill is variable.)

-- The memory is described as a ring buffer. Two pointers are nessecary:
-- fill_pointer and empty_pointer.
-- The first one shows the position of the last filled buffer word, the
-- second one shows the position up to with the buffer is clered by the output.

-- synopsys translate_off
Library XilinxCoreLib;
-- synopsys translate_on

library IEEE;
use IEEE.STD_LOGIC_1164.ALL, IEEE.NUMERIC_STD.all,IEEE.std_logic_unsigned.ALL;
use WORK.ALL;

entity huffman is
port (state64_in     : in  std_logic_vector(5 downto 0);
      state8_in      : in  std_logic_vector(2 downto 0);
      pixclock       : in  std_logic;
      fastclock      : in  std_logic;
      reset_huf      : in  std_logic;
      coded_long     : in  std_logic_vector(11 downto 0);
      category_in    : in  std_logic_vector(3 downto 0);
      output_huf     : out std_logic_vector(7 downto 0);
      line_flag_huf  : out std_logic;
      field_blank_huf: out std_logic;
      firstline_huf  : out std_logic;
      state_coder_in : in  std_logic_vector(2 downto 0);
      run_in         : in  std_logic_vector(3 downto 0);
      buf_state      : in  std_logic_vector(1 downto 0);
      last_zero      : in  std_logic_vector(6 downto 0);
      quantisation   : in  std_logic    );
```

```
end entity huffman;

architecture huffman_behavior of huffman is

signal    state64      : unsigned(5 downto 0);
signal    state8      : unsigned(2 downto 0);
signal    state_coder    : unsigned(2 downto 0);
signal    nextstate_coder    : unsigned(2 downto 0);

signal    coded_long_dec    : std_Logic_vector(11 downto 0);

constant  buf_bit      : integer:=8;
constant  buf_size    : integer:=511;  --(should be worst case)

signal    fill_pointer    : unsigned(buf_bit downto 0);
signal    fill_pointer_inc  : unsigned(buf_bit downto 0);
signal    fill_pointer_inc2  : unsigned(buf_bit downto 0);
signal    empty_pointer    : unsigned(buf_bit downto 0);
signal    empty_pointer_new  : unsigned(buf_bit downto 0);
signal    bufblock_pointer  : integer range 0 to 15;
signal    header_pointer    : unsigned(buf_bit downto 0);
signal    header_set    : std_logic;
signal    header_generated  : std_logic;
signal    header_part    : integer range 0 to 307;
signal    block_counter    : integer range 0 to 127;
signal    block_counted    : std_logic;
signal    eoi_generated    : std_logic;

signal    field_blank_was_set  : std_logic;
signal    firstrun_after_reset  : std_logic;

signal    line_flag_huf_i    : std_logic;
signal    line_flag_huf_i2  : std_logic;

-- The chosen JPEG-Header (plain text, hex). Its the example of the JPEG DIS Annex K.
--  ffd9 -- (EOI - last image)
--  ffd8
--  ffdB  00 43 00 10 0b 0c 0e 0c 0a 10 0e 0d 0e 12 11 10 -- quantisation table
--    13 18 28 1a 18 16 16 18 31 23 25 1d 28 3a 33 3d
--    3c 39 33 38 37 40 48 5c 4e 40 44 57 45 37 38 50
--    6d 51 57 5f 62 67 68 67 3e 4d 71 79 70 64 78 5c
--    65 67 63
--  ffc0  00 0b 08 00 80 00 40 01 01 11 00
--  ffc4  00 d2 00 00 01 05 01 01 01 01 01 01 00 00 00 00 -- Huffman codeword table
--    00 00 00 00 01 02 03 04 05 06 07 08 09 0a 0b 10
--    00 02 01 03 03 02 04 03 05 05 04 04 00 00 01 7d
--    01 02 03 00 04 11 05 12 21 31 41 06 13 51 61 07
--    22 71 14 32 81 91 a1 08 23 42 b1 c1 15 52 d1 f0
--    24 33 62 72 82 09 0a 16 17 18 19 1a 25 26 27 28
--    29 2a 34 35 36 37 38 39 3a 43 44 45 46 47 48 49
--    4a 53 54 55 56 57 58 59 5a 63 64 65 66 67 68 69
--    6a 73 74 75 76 77 78 79 7a 83 84 85 86 87 88 89
--    8a 92 93 94 95 96 97 98 99 9a a2 a3 a4 a5 a6 a7
--    a8 a9 aa b2 b3 b4 b5 b6 b7 b8 b9 ba c2 c3 c4 c5
--    c6 c7 c8 c9 ca d2 d3 d4 d5 d6 d7 d8 d9 da e1 e2
--    e3 e4 e5 e6 e7 e8 e9 ea f1 f2 f3 f4 f5 f6 f7 f8
--    f9 fa
--  ffda  00 08 01 01 00 00 3f 00
-- 2448 bit, 306 byte  + 2 Byte EOI (16 Bit)

type header_part_type is array(natural range<>) of integer range 0 to 255;
constant    header_intro    : header_part_type(0 to 8):=(
16#ff#,16#d9#,16#ff#,16#d8#,16#ff#,16#dB#,16#00#,16#43#,16#00# );

-- quantisation like in JPEG DIS Annex K
constant  header_quant    : header_part_type(9 to 72):=(
16#10#,16#0b#,16#0c#,16#0e#,16#0c#,16#0a#,16#10#,16#0e#,16#0d#,16#0e#,16#12#,16#11#,16#10#,
16#13#,16#18#,16#28#,16#1a#,16#18#,16#16#,16#16#,16#18#,16#31#,16#23#,16#25#,16#1d#,16#28#,
16#3a#,16#33#,16#3d#,16#3c#,16#39#,16#33#,16#38#,16#37#,16#40#,16#48#,16#5c#,16#4e#,16#40#,
16#44#,16#57#,16#45#,16#37#,16#38#,16#50#,16#6d#,16#51#,16#57#,16#5f#,16#62#,16#67#,16#68#,
16#67#,16#3e#,16#4d#,16#71#,16#79#,16#70#,16#64#,16#78#,16#5c#,16#65#,16#67#,16#63# );

-- no quantisation (division.vhdl must be "deactivated")
constant  header_unquant    : header_part_type(9 to 72):=(
```

```
16#01#,16#01#,16#01#,16#01#,16#01#,16#01#,16#01#,16#01#,16#01#,16#01#,16#01#,16#01#,16#01#,
16#01#,16#01#,16#01#,16#01#,16#01#,16#01#,16#01#,16#01#,16#01#,16#01#,16#01#,16#01#,16#01#,
16#01#,16#01#,16#01#,16#01#,16#01#,16#01#,16#01#,16#01#,16#01#,16#01#,16#01#,16#01#,16#01#,
16#01#,16#01#,16#01#,16#01#,16#01#,16#01#,16#01#,16#01#,16#01#,16#01#,16#01#,16#01#,16#01#,
16#01#,16#01#,16#01#,16#01#,16#01#,16#01#,16#01#,16#01#,16#01#,16#01#,16#01#,16#01# );

constant  header_huff    : header_part_type(73 to 307):=(
16#ff#,16#c0#,16#00#,16#0b#,16#08#,16#00#,16#80#,16#00#,16#40#,16#01#,16#01#,16#11#,16#00#,
16#ff#,16#c4#,16#00#,16#d2#,16#00#,16#00#,16#01#,16#05#,16#01#,16#01#,16#01#,16#01#,16#01#,
16#01#,16#00#,16#00#,16#00#,16#00#,16#00#,16#00#,16#00#,16#00#,16#01#,16#02#,16#03#,16#04#,
16#05#,16#06#,16#07#,16#08#,16#09#,16#0a#,16#0b#,16#10#,16#00#,16#02#,16#01#,16#03#,16#03#,
16#02#,16#04#,16#03#,16#05#,16#05#,16#04#,16#04#,16#00#,16#00#,16#01#,16#7d#,16#01#,16#02#,
16#03#,16#00#,16#04#,16#11#,16#05#,16#12#,16#21#,16#31#,16#41#,16#06#,16#13#,16#51#,16#61#,
16#07#,16#22#,16#71#,16#14#,16#32#,16#81#,16#91#,16#a1#,16#08#,16#23#,16#42#,16#b1#,16#c1#,
16#15#,16#52#,16#d1#,16#f0#,16#24#,16#33#,16#62#,16#72#,16#82#,16#09#,16#0a#,16#16#,16#17#,
16#18#,16#19#,16#1a#,16#25#,16#26#,16#27#,16#28#,16#29#,16#2a#,16#34#,16#35#,16#36#,16#37#,
16#38#,16#39#,16#3a#,16#43#,16#44#,16#45#,16#46#,16#47#,16#48#,16#49#,16#4a#,16#53#,16#54#,
16#55#,16#56#,16#57#,16#58#,16#59#,16#5a#,16#63#,16#64#,16#65#,16#66#,16#67#,16#68#,16#69#,
16#6a#,16#73#,16#74#,16#75#,16#76#,16#77#,16#78#,16#79#,16#7a#,16#83#,16#84#,16#85#,16#86#,
16#87#,16#88#,16#89#,16#8a#,16#92#,16#93#,16#94#,16#95#,16#96#,16#97#,16#98#,16#99#,16#9a#,
16#a2#,16#a3#,16#a4#,16#a5#,16#a6#,16#a7#,16#a8#,16#a9#,16#aa#,16#b2#,16#b3#,16#b4#,16#b5#,
16#b6#,16#b7#,16#b8#,16#b9#,16#ba#,16#c2#,16#c3#,16#c4#,16#c5#,16#c6#,16#c7#,16#c8#,16#c9#,
16#ca#,16#d2#,16#d3#,16#d4#,16#d5#,16#d6#,16#d7#,16#d8#,16#d9#,16#da#,16#e1#,16#e2#,16#e3#,
16#e4#,16#e5#,16#e6#,16#e7#,16#e8#,16#e9#,16#ea#,16#f1#,16#f2#,16#f3#,16#f4#,16#f5#,16#f6#,
16#f7#,16#f8#,16#f9#,16#fa#,16#ff#,16#da#,16#00#,16#08#,16#01#,16#01#,16#00#,16#00#,16#3f#,
16#00#  );

-- The code table for the huffman codewords for the run/length code combination.
type code_table_row_type is array (natural range<>) of std_logic_vector(15 downto 0);
type code_table_type is array (natural range<>) of code_table_row_type(0 to 10);
constant  code_table    : code_table_type(0 to 15):=(
("0000000000001010","0000000000000000","0000000000000001","0000000000000100",
"0000000000001011","0000000000011010","0000000001111000","0000000011111000",
"0000001111110110","1111111110000010","1111111110000011"),
("0000000000000000","0000000000001100","0000000000011011","0000000001111001",
"0000000111110110","0000011111110110","1111111110000100","1111111110000101",
"1111111110000110","1111111110000111","1111111110001000"),
("0000000000000000","0000000000011100","0000000011111001","0000001111110111",
"0000111111110100","1111111110001001","1111111110001010","1111111110001011",
"1111111110001100","1111111110001101","1111111110001110"),
("0000000000000000","0000000000111010","0000000111110111","0000111111110101",
"1111111110001111","1111111110010000","1111111110010001","1111111110010010",
"1111111110010011","1111111110010100","1111111110010101"),
("0000000000000000","0000000000111011","0000001111111000","1111111110010110",
"1111111110010111","1111111110011000","1111111110011001","1111111110011010",
"1111111110011011","1111111110011100","1111111110011101"),
("0000000000000000","0000000001111010","0000011111110111","1111111110011110",
"1111111110011111","1111111110100000","1111111110100001","1111111110100010",
"1111111110100011","1111111110100100","1111111110100101"),
("0000000000000000","0000000001111011","0000111111110110","1111111110100110",
"1111111110100111","1111111110101000","1111111110101001","1111111110101010",
"1111111110101011","1111111110101100","1111111110101101"),
("0000000000000000","0000000011111010","0000111111110111","1111111110101110",
"1111111110101111","1111111110110000","1111111110110001","1111111110110010",
"1111111110110011","1111111110110100","1111111110110101"),
("0000000000000000","0000000111111000","0111111111000000","1111111110110110",
"1111111110110111","1111111110111000","1111111110111001","1111111110111010",
"1111111110111011","1111111110111100","1111111110111101"),
("0000000000000000","0000000111111001","1111111110111110","1111111110111111",
"1111111111000000","1111111111000001","1111111111000010","1111111111000011",
"1111111111000100","1111111111000101","1111111111000110"),
("0000000000000000","0000000111111010","1111111111000111","1111111111001000",
"1111111111001001","1111111111001010","1111111111001011","1111111111001100",
"1111111111001101","1111111111001110","1111111111001111"),
("0000000000000000","0000001111111001","1111111111010000","1111111111010001",
"1111111111010010","1111111111010011","1111111111010100","1111111111010101",
"1111111111010110","1111111111010111","1111111111011000"),
("0000000000000000","0000001111111010","1111111111011001","1111111111011010",
"1111111111011011","1111111111011100","1111111111011101","1111111111011110",
"1111111111011111","1111111111100000","1111111111100001"),
("0000000000000000","0000011111111000","1111111111100010","1111111111100011",
"1111111111100100","1111111111100101","1111111111100110","1111111111100111",
"1111111111101000","1111111111101001","1111111111101010"),
("0000000000000000","1111111111101011","1111111111101100","1111111111101101",
```

```
"1111111111101110","1111111111101111","1111111111110000","1111111111110001",
"1111111111110010","1111111111110011","1111111111110100"),
("0000011111111001","1111111111110101","1111111111110110","1111111111110111",
"1111111111111000","1111111111111001","1111111111111010","1111111111111011",
"1111111111111100","1111111111111101","1111111111111110")  );
-- table is filled from LSM

-- The valid length of the codewords in the code table.
type code_length_table_row_type is array (natural range<>) of integer range 0 to 15;
type code_length_table_type is array (natural range<>) of code_length_table_row_type(0 to 10);
constant  code_table_length  : code_length_table_type(0 to 15):=(
  ( 3, 1, 1, 2, 3, 4, 6, 7, 9,15,15),
  ( 0, 3, 4, 6, 8,10,15,15,15,15,15),
  ( 0, 4, 7, 9,11,15,15,15,15,15,15),
  ( 0, 5, 8,11,15,15,15,15,15,15,15),
  ( 0, 5, 9,15,15,15,15,15,15,15,15),
  ( 0, 6,10,15,15,15,15,15,15,15,15),
  ( 0, 6,11,15,15,15,15,15,15,15,15),
  ( 0, 7,11,15,15,15,15,15,15,15,15),
  ( 0, 8,14,15,15,15,15,15,15,15,15),
  ( 0, 8,15,15,15,15,15,15,15,15,15),
  ( 0, 8,15,15,15,15,15,15,15,15,15),
  ( 0, 9,15,15,15,15,15,15,15,15,15),
  ( 0, 9,15,15,15,15,15,15,15,15,15),
  ( 0,10,15,15,15,15,15,15,15,15,15),
  ( 0,15,15,15,15,15,15,15,15,15,15),
  (10,15,15,15,15,15,15,15,15,15,15)  );
-- increase code length by 1 to get the amount of bits

-- The code table for the DC codewords.
type DC_code_table_type is array (natural range<>) of std_logic_vector(8 downto 0);
constant  DC_code_table    : DC_code_table_type(0 to 11):=(
  "000000000","000000010","000000011","000000100","000000101","000000110",
  "000001110","000011110","000111110","001111110","011111110","111111110"  );

-- The valid length of the DC-codewords in the DC code table.
type DC_code_table_length_type is array (natural range <>) of integer range 1 to 9;
constant  DC_code_table_length1  : DC_code_table_length_type(0 to 11):=(
  1,2,2,2,2,2,3,4,5,6,7,8  );
-- Again, but incremented (easier coding and hopefully a faster synthesized version).
constant  DC_code_table_length2  : DC_code_table_length_type(0 to 11):=(
  2,3,3,3,3,3,4,5,6,7,8,9  );

-- The codeword for EOB.
constant  eob        : std_logic_vector(3 downto 0):="1010";

signal    category_unsig      : unsigned(3 downto 0);
signal    run_unsig      : unsigned(3 downto 0);
signal    category      : integer range 0 to 11;
signal    run        : integer range 0 to 15;

type statetype_output is (nop,data_to_out,header_out,zero_stuffing,set_flag);
signal    state_output      : statetype_output;
signal    state_output_dummy    : statetype_output;
signal    nextstate_output    : statetype_output;

-- conversion signals
signal    state_coder_slv      : std_logic_vector(2 downto 0);

-- signals for the buffer - components
signal    write,write2      : std_logic;
signal    address,address2    : std_logic_VECTOR(buf_bit downto 0);
signal    address_unsig,address2_unsig  : unsigned(buf_bit downto 0);
signal    data_in,data_in2    : std_logic_vector(15 downto 0);
signal    data_out,data_out2    : std_logic_vector(15 downto 0);
signal    length_in,length_in2    : std_logic_vector(3 downto 0);
signal    length_unsig,length2_unsig  : unsigned(3 downto 0);
signal    length_out,length_out2    : std_logic_vector(3 downto 0);
signal    length_out_int,length_out2_int  : integer range 0 to 15;

signal    firstoutput      : std_logic;
signal    pixclock_old      : std_logic;
signal    counter        : integer range 0 to 15;
-- fastclock=10*pixclock -> counter up to 10
```

```
constant  counter_const      : integer:=8; -- 9 possible
signal    go_counter      : std_logic;

signal    output_huf_dummy    : std_logic_vector(95 downto 0);
signal    not_filled      : std_logic;
signal    step1        : integer range 0 to 95;
signal    step2        : integer range 0 to 95;
--signal    coding_finished    : std_logic;

component huf_help
port (coded_long_hlp         : in  std_logic_vector(11 downto 0);
      coded_long_dec_slv_hlp : out std_logic_vector(11 downto 0)  );
end component;

component out_buf
  port (  addra: IN std_logic_VECTOR(8 downto 0);
    clka: IN std_logic;
    addrb: IN std_logic_VECTOR(8 downto 0);
    clkb: IN std_logic;
    dia: IN std_logic_VECTOR(15 downto 0);
    wea: IN std_logic;
    dib: IN std_logic_VECTOR(15 downto 0);
    web: IN std_logic;
    doa: OUT std_logic_VECTOR(15 downto 0);
    dob: OUT std_logic_VECTOR(15 downto 0)  );
end component;

attribute fpga_dont_touch: string;
attribute fpga_dont_touch of out_buf: component is "true";

component out_buf_length
  port (  addra: IN std_logic_VECTOR(8 downto 0);
    clka: IN std_logic;
    addrb: IN std_logic_VECTOR(8 downto 0);
    clkb: IN std_logic;
    dia: IN std_logic_VECTOR(3 downto 0);
    wea: IN std_logic;
    dib: IN std_logic_VECTOR(3 downto 0);
    web: IN std_logic;
    doa: OUT std_logic_VECTOR(3 downto 0);
    dob: OUT std_logic_VECTOR(3 downto 0)  );
end component;

--attribute fpga_dont_touch: string;
attribute fpga_dont_touch of out_buf_length: component is "true";

begin

helping_variables : huf_help
port map(
  coded_long_hlp=>coded_long,
  coded_long_dec_slv_hlp=>coded_long_dec  );

output_buffer :  out_buf
port map (
  addra => address,
  clka => fastclock,
  addrb => address2,
  clkb => fastclock,
  dia => data_in,
  wea => write,
  dib => data_in2,
  web => write2,
  doa => data_out,
  dob => data_out2  );

output_buffer_length : out_buf_length
port map (
  addra => address,
  clka => fastclock,
  addrb => address2,
  clkb => fastclock,
  dia => length_in,
  wea => write,
```

```
  dib => length_in2,
  web => write2,
  doa => length_out,
  dob => length_out2  );


-- input conversion
process (state_coder)
begin
for N in 0 to 2 loop
  state_coder_slv(N)<=state_coder(N);
end loop;
end process;

process(category_in,run_in)
begin
category<=conv_integer(category_in);
run<=conv_integer(run_in);
end process;

process (state64_in)
begin
for N in 0 to 5 loop
  state64(N)<=state64_in(N);
end loop;
end process;

process (state8_in,state_coder_in)
begin
for N in 0 to 2 loop
  state8(N)<=state8_in(N);
  state_coder(N)<=state_coder_in(N);
end loop;
end process;
-- input conversion - end


-- Some helping variables (will produce a code that is easier to read for the programmer).
process(fill_pointer)
variable  fill_pointer_add1  : unsigned(buf_bit downto 0);
variable  fill_pointer_add2  : unsigned(buf_bit downto 0);
begin
fill_pointer_add1:=to_unsigned(1,buf_bit+1);
fill_pointer_inc<=fill_pointer+fill_pointer_add1;
fill_pointer_add2:=to_unsigned(2,buf_bit+1);
fill_pointer_inc2<=fill_pointer+fill_pointer_add2;
end process;


-- output stack buffer + code table
process (reset_huf,state_coder,category,fastclock,nextstate_coder,state64,state8,fill_pointer,
         empty_pointer,header_pointer,bufblock_pointer,state_output,nextstate_output,coded_long,
         coded_long_dec,buf_state,length_out,length_out2,data_out,block_counter,header_set,
         block_counted,field_blank_was_set,firstrun_after_reset,firstoutput,last_zero,pixclock,
         pixclock_old,counter,eoi_generated,go_counter,step1,step2,header_generated)
variable    header_dummy    : unsigned(7 downto 0);
variable    bufblock_pointer_dummy  : integer range 0 to 15;
variable    step2_v      : integer range 0 to 95;

-- state_coder: 000=nop, 001=init, 010=runlength, 011=out, 100=wait_end,
                101=enc_EOB_EOI, 110=waitstate

begin
if (reset_huf='0') then
  fill_pointer<=to_unsigned(0,buf_bit+1);
  empty_pointer<=to_unsigned(0,buf_bit+1); -- points to the 1st free place in out_buffer
  empty_pointer_new<=to_unsigned(0,buf_bit+1);
  header_pointer<=to_unsigned(0,buf_bit+1); -- points to the position, where a new header must be applied
  eoi_generated<='1';
  state_output<=nop;
  nextstate_output<=nop;
  block_counter<=0; -- counts the 8x8 blocks (128)
  header_set<='1';
  header_generated<='1';
```

```
  block_counted<='1';
  field_blank_huf<='1';
  field_blank_was_set<='1';
  firstrun_after_reset<='0';
  firstline_huf<='0';
  line_flag_huf_i<='0';
  write<='0';
  write2<='0';
  length_unsig<="0000";
  length2_unsig<="0000";
  -- no address collision
  for N in buf_bit downto 1 loop
    address_unsig(N)<='0';
    address2_unsig(N)<='0';
  end loop;
    address_unsig(0)<='0';
    address2_unsig(0)<='1';
  data_in<="0000000000000000";
  data_in2<="0000000000000000";
  firstoutput<='1';
  pixclock_old<=pixclock;
  go_counter<='0';
elsif rising_edge(fastclock) then
  data_in2(15 downto 12)<="0000"; -- flipflop instead of latch
  if (pixclock_old='1' and pixclock='0') then -- nearly a falling_edge(pixclock)
    counter<=0;
    -- Counter divides the period of a pixclock into several parts
    -- (with the help of fastclock)
    go_counter<='1';
  elsif (go_counter='1') then
    counter<=counter+1;
  end if;
  if (pixclock_old/=pixclock) then
    pixclock_old<=pixclock; -- delay pixclock
  end if;
  if (state_coder="001") then
    header_generated<='0';
    block_counted<='0';
    if (counter=counter_const) then -- short before next pseudo-falling_edge(pixclock)
      -- at the end of this state_coder-cycle
      -- but one clock-cycle faster

      --apply correct address
      address_unsig<=fill_pointer; -- old fill_pointer
      write<='1';
      -- apply DC-codeword
      data_in(8 downto 0)<=DC_code_table(category);
      length_unsig<=to_unsigned(DC_code_table_length1(category),4);
      -- apply DC-value
      address2_unsig<=fill_pointer_inc;
      if (category/=0) then
        write2<='1';
        if (coded_long(11)='1') then -- negative
          data_in2(11 downto 0)<=coded_long_dec;
        else  data_in2(11 downto 0)<=coded_long;
        end if;
        length2_unsig<=to_unsigned(category-1,4);
        fill_pointer<=fill_pointer_inc2;
      else  fill_pointer<=fill_pointer_inc;
        -- No data word needs to be applied when category=0.
        write2<='0';
      end if;
      -- break while giving header or data out (new encoding starts -> output has to wait)
      if (state_output=header_out) then
        firstline_huf<='0';
        line_flag_huf_i<='0';
        state_output<=set_flag;
        nextstate_output<=header_out;
      end if;
      if (state_output=data_to_out) then
        firstline_huf<='0';
        line_flag_huf_i<='0';
        state_output<=set_flag;
        nextstate_output<=data_to_out;
```

```
    end if;
    -- break while giving header out
  else  write<='0';
    write2<='0';
  end if;
elsif (state_coder="011") then
  if (counter=counter_const) then
    --apply correct address
    address_unsig<=fill_pointer; -- old fill_pointer
    -- apply AC-codeword
    write<='1';
    data_in<=code_table(run)(category);
    length_unsig<=to_unsigned(code_table_length(run)(category),4);
    -- apply AC-value
    address2_unsig<=fill_pointer_inc;
    if (category/=0) then
      write2<='1';
      if coded_long(11)='1' then
        data_in2(11 downto 0)<=coded_long_dec;
      else  data_in2(11 downto 0)<=coded_long;
      end if;
      length2_unsig<=to_unsigned(category-1,4);
      fill_pointer<=fill_pointer_inc2;
    else  fill_pointer<=fill_pointer_inc;
      write2<='0';
    end if;
  else  write<='0';
    write2<='0';
  end if;
elsif (state_coder="101") then
  if (counter=counter_const) then
    -- EOB - generation
    if (last_zero/="1000000") then
      -- EOB is not coded, if the 63rd coefficient is not zero
      write<='1';
      data_in(3 downto 0)<=eob;
      length_unsig<=to_unsigned(3,4);
      fill_pointer<=fill_pointer_inc;
    else  fill_pointer<=fill_pointer;
      write<='0';
    end if;
    --apply correct address
    address_unsig<=fill_pointer; -- old fill_pointer
    -- EOI and header - generation
    if (block_counter=127 and header_generated='0' and block_counted='0') then
      -- generate EOI
      if (eoi_generated='0') then
        if (last_zero/="1000000") then
          header_pointer<=fill_pointer_inc;
        else  eoi_generated<='1';
          header_pointer<=fill_pointer;
        end if;
      end if;
      --generate header
      header_generated<='1';
      header_set<='1';
      block_counter<=0;
      block_counted<='1'; -- safety: This block will be not counted again.
    elsif (block_counted='0') then
      block_counter<=block_counter+1;
      -- block_counter counts the number of 8x8-blocks (128)
      -- it is a marker for the header-generation
      block_counted<='1';
    end if;
  else  write<='0';
    write2<='0';
  end if;
elsif (state_coder="100" or state_coder="000") then
  case state_output is
  -- note: state_output runs one too fast (o.k.)
  when nop => if (counter=counter_const and go_counter='1') then
                line_flag_huf_i<='0';
                firstline_huf<='0';
                if (to_integer(fill_pointer-empty_pointer)<30) then
```

```
                --<7 is correct, but Puegner's machines ignore a single 8Bit output
                -- at least 8 bit available for output
                state_output<=nop;
                if (header_set='1') then
                  state_output<=data_to_out;
                  line_flag_huf_i<='1';
                  if (field_blank_was_set='1') then
                    firstline_huf<='1';
                  end if;
                  field_blank_huf<='0';
                end if;
              else  state_output<=data_to_out;
                    line_flag_huf_i<='1';
                    if (field_blank_was_set='1') then
                      firstline_huf<='1';
                    end if;
                    field_blank_huf<='0';
              end if;
              if ((block_counter=127 and state64(5 downto 0)="111111"
                   and state8="000") or firstrun_after_reset='0') then
              -- should be everytime after last output in block 127
                field_blank_huf<='1';
                field_blank_was_set<='1';
              end if;
              header_part<=0;
            end if;
  when data_to_out => write<='0';
            write2<='0';
            if (counter=0) then
              not_filled<='1';
              step1<=95; -- init
            end if;
            if (counter<=5) then
              -- See clock scheme in the documentation for details.
              address_unsig<=empty_pointer+to_unsigned(counter,buf_bit+1);
              address2_unsig<=empty_pointer+to_unsigned(counter+1,buf_bit+1);
            end if;
            if (counter=2) then
              -- Unique: step2 hast to be calculated within this counter-cycle and not
              -- one cycle before like in the counter-cycles 3 to 7.
              -- The same idea leads to the use of bufblock_pointer_dummy.
              -- All other parts in the algorithm are equal.

              -- buf_state: nop=00, first_write1=01, write1=10, write2=11
              if ((buf_state="11" or buf_state="10") and firstoutput='1') then
                -- 1st output after reset
                bufblock_pointer<=length_out_int; -- Get the 1st bufblock_pointer.
                bufblock_pointer_dummy:=length_out_int;
                firstoutput<='0';
                step2_v:=95-length_out_int;
                --step2<=95-length_out_int;
              else  step2_v:=95-bufblock_pointer;
                --step2<=95-bufblock_pointer;
                bufblock_pointer_dummy:=bufblock_pointer;
              end if;

              if ((header_set='1') and (header_pointer=empty_pointer)) then
                for N in 95 downto 0 loop
                  if (N<=step1) then
                    output_huf_dummy(N)<='1';
                    -- Fill last bits with ones and jump to header_out.
                  end if;
                end loop;
                state_output_dummy<=header_out;
                header_set<='0';
                not_filled<='0';
              else  for N in 95 downto 0 loop
                  if ((N<=step1) and (N>=step2_v)) then
                    -- Add next data word to the correct position in output_huf_dummy.
                    output_huf_dummy(N)<=data_out(bufblock_pointer_dummy-(step1-N));
                  end if;
                end loop;
                if (step2_v>88) then
                  -- output_huf_dummy not filled with at least 8 bits
```

```
      bufblock_pointer<=length_out2_int;
      step1<=step2_v-1;
      step2<=step2_v-1-length_out2_int;
      --not_filled<='1';  -- redundant
      empty_pointer_new<=empty_pointer+1;
    elsif (step2_v=88) then
      -- output_huf_dummy exactly filled with 8 bits
      not_filled<='0';
      bufblock_pointer<=length_out2_int;
      empty_pointer_new<=empty_pointer+1;
    else  not_filled<='0';
      bufblock_pointer<=bufblock_pointer-(step1-87);
    end if;
    state_output_dummy<=data_to_out;
  end if;
end if;
if (counter>=3 and counter<=7 and not_filled='1') then
  if ((header_set='1') and (header_pointer=empty_pointer_new)) then
    for N in 95 downto 0 loop
      if (N<=step1) then
        output_huf_dummy(N)<='1';
      end if;
    end loop;
    state_output_dummy<=header_out;
    header_set<='0';
    not_filled<='0';
  else  for N in 95 downto 0 loop
      if ((N<=step1) and (N>=step2)) then
        output_huf_dummy(N)<=data_out(bufblock_pointer-(step1-N));
      end if;
    end loop;
    if (step2>88) then
      -- output_huf_dummy not filled with at least 8 bits
      bufblock_pointer<=length_out2_int;
      step1<=step2-1;
      step2<=step2-1-length_out2_int;
      --not_filled<='1';  -- redundant
      empty_pointer_new<=empty_pointer_new+1;
    elsif (step2=88) then
      -- output_huf_dummy exactly filled with 8 bits
      not_filled<='0';
      bufblock_pointer<=length_out2_int;
      empty_pointer_new<=empty_pointer_new+1;
    else  not_filled<='0';
      bufblock_pointer<=bufblock_pointer-(step1-87);
    end if;
    state_output_dummy<=data_to_out;
  end if;
end if;
if (counter=counter_const) then
  -- 10*pixclk needed (output_huf must be valid at next rising_edge(pixclock))
  line_flag_huf_i<='1';
  if (field_blank_was_set='1') then
    firstline_huf<='1';
    field_blank_was_set<='0';
  end if;
  field_blank_huf<='0';
  firstrun_after_reset<='1';

  if (state_output_dummy=header_out) then
    state_output<=header_out;
  elsif  (output_huf_dummy(95 downto 88)="11111111") then
    state_output<=zero_stuffing;
    -- After an output of FF(hex) that is not part of the header
    -- or a marker 00(hex) must be given out.
  elsif (to_integer(fill_pointer-empty_pointer_new)<7) then
    -- see comment at nop
    state_output<=nop;
    if (header_set='1') then
      state_output<=data_to_out;
    end if;
  else  state_output<=data_to_out;
  end if;
```

```
                empty_pointer<=empty_pointer_new;
                output_huf<=output_huf_dummy(95 downto 88);
              end if;
    when zero_stuffing => if (counter=counter_const) then
                output_huf<="00000000";
                if (to_integer(fill_pointer-empty_pointer)<7) then
                  -- see comment at nop (continue output or stop it)
                  state_output<=nop;
                  if (header_set='1') then
                    state_output<=data_to_out;
                  end if;
                else  state_output<=data_to_out;
                end if;
              end if;
    when header_out => if (counter=counter_const) then
                line_flag_huf_i<='1';
                if (header_part<9) then
                  header_dummy:=to_unsigned(header_intro(header_part),8);
                elsif (header_part<73) then
                  if (quantisation='1') then
                    header_dummy:=to_unsigned(header_quant(header_part),8);
                  else  header_dummy:=to_unsigned(header_unquant(header_part),8);
                  end if;
                else  header_dummy:=to_unsigned(header_huff(header_part),8);
                end if;
                for M in 7 downto 0 loop
                  output_huf(M)<=header_dummy(M);
                end loop;
                if (header_part=307) then
                  state_output<=nop;
                else  header_part<=header_part+1;
                  state_output<=header_out;
                end if;

                firstoutput<='1';
              end if;
    when set_flag => state_output<=nextstate_output;
              line_flag_huf_i<='1';
    when others => null;
    end case;
  end if;
  if (state_coder="100") then
    eoi_generated<='0'; --reset while state_coder="wait_end"
  end if;
end if;
end process;


-- output conversion for the buffer -  components
process (address_unsig,address2_unsig)
begin
for N in 0 to buf_bit loop
  address(N)<=address_unsig(N);
  address2(N)<=address2_unsig(N);
end loop;
end process;

process (length_unsig, length2_unsig)
begin
for N in 0 to 3 loop
  length_in(N)<=length_unsig(N);
  length_in2(N)<=length2_unsig(N);
end loop;
end process;

process (length_out, length_out2)
variable  length_out_dummy,length_out2_dummy  : unsigned(3 downto 0);
begin
for N in 0 to 3 loop
  length_out_dummy(N):=length_out(N);
  length_out2_dummy(N):=length_out2(N);
end loop;
length_out_int<=to_integer(length_out_dummy);
length_out2_int<=to_integer(length_out2_dummy);
```

```
end process;
--output conversion for the buffer -  components

-- Because of bugs in the byte_to_word.vhd (Puegner) the falling edge of the
-- signal lineflag has to be delayed a little bit.
-- line_flag has to change to 0 _after_ a falling_edge(pixclock)
lineflag_delay : process (line_flag_huf_i,line_flag_huf_i2,fastclock)
begin
if (falling_edge(fastclock)) then
  line_flag_huf_i2<=line_flag_huf_i;
end if;

if (line_flag_huf_i2='1') then
  line_flag_huf<=line_flag_huf_i2;
else  line_flag_huf<=line_flag_huf_i;
end if;
end process lineflag_delay;


end architecture huffman_behavior;

configuration huffman_cfg of huffman is
for huffman_behavior
-- synopsys translate_off
  for all : out_buf use entity XilinxCoreLib.C_MEM_DP_BLOCK_V1_0(behavioral)
    generic map(
      c_depth_b => 512,
      c_depth_a => 512,
      c_has_web => 1,
      c_has_wea => 1,
      c_has_dib => 1,
      c_has_dia => 1,
      c_clka_polarity => 1,
      c_web_polarity => 1,
      c_address_width_b => 9,
      c_address_width_a => 9,
      c_width_b => 16,
      c_width_a => 16,
      c_clkb_polarity => 1,
      c_ena_polarity => 1,
      c_rsta_polarity => 1,
      c_has_rstb => 0,
      c_has_rsta => 0,
      c_read_mif => 0,
      c_enb_polarity => 1,
      c_pipe_stages => 0,
      c_rstb_polarity => 1,
      c_has_enb => 0,
      c_has_ena => 0,
      c_mem_init_radix => 16,
      c_default_data => "0",
      c_mem_init_file => "/home/eeebs2/hilde/synopsys/zusammen/SRC/cores/out_buf.mif",
      c_has_dob => 1,
      c_generate_mif => 1,
      c_has_doa => 1,
      c_wea_polarity => 1);
  end for;
-- synopsys translate_on
-- synopsys translate_off
  for all : out_buf_length use entity XilinxCoreLib.C_MEM_DP_BLOCK_V1_0(behavioral)
    generic map(
      c_depth_b => 512,
      c_depth_a => 512,
      c_has_web => 1,
      c_has_wea => 1,
      c_has_dib => 1,
      c_has_dia => 1,
      c_clka_polarity => 1,
      c_web_polarity => 1,
      c_address_width_b => 9,
      c_address_width_a => 9,
      c_width_b => 4,
      c_width_a => 4,
      c_clkb_polarity => 1,
```

```
      c_ena_polarity => 1,
      c_rsta_polarity => 1,
      c_has_rstb => 0,
      c_has_rsta => 0,
      c_read_mif => 0,
      c_enb_polarity => 1,
      c_pipe_stages => 0,
      c_rstb_polarity => 1,
      c_has_enb => 0,
      c_has_ena => 0,
      c_mem_init_radix => 16,
      c_default_data => "0",
      c_mem_init_file => "/home/eeebs2/hilde/synopsys/zusammen/SRC/cores/out_buf_length.mif",
      c_has_dob => 1,
      c_generate_mif => 1,
      c_has_doa => 1,
      c_wea_polarity => 1);
  end for;
-- synopsys translate_on
end for;
end huffman_cfg;
```

9.10 Die Zusammenfassung der Huffman-Komponenten

huf_comb.vhdl

```
-- Simple combinatin of the 2 Huffman-stages.
-- They were combined because of testing this big block.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL, IEEE.NUMERIC_STD.all;
use WORK.ALL;

entity huf_comb is
port (data_in         : in  std_logic_vector(10 downto 0);
      state64_in      : in  std_logic_vector(5 downto 0);
      state8_in       : in  std_logic_vector(2 downto 0);
      state8_in_old   : in  std_logic_vector(2 downto 0);
      pixclock        : in  std_logic;
      fastclock       : in  std_logic;
      reset_huf       : in  std_logic;
      quantisation    : in  std_logic;
      output_huf      : out std_logic_vector(7 downto 0);
      line_flag_huf   : out std_logic;
      field_blank_huf : out std_logic;
      firstline_huf   : out std_logic    );
end entity huf_comb;

architecture huf_comb_behavior of huf_comb is

component data_buf
port (data_in         : in  std_logic_vector(10 downto 0);
      state64_in      : in  std_logic_vector(5 downto 0);
      state8_in       : in  std_logic_vector(2 downto 0);
      state8_in_old   : in  std_logic_vector(2 downto 0);
      pixclock        : in  std_logic;
      fastclock       : in  std_logic;
      reset_huf       : in  std_logic;
      coded_long_out  : out std_logic_vector(11 downto 0);
      category        : out std_logic_vector(3 downto 0);
      state_coder_out : out std_logic_vector(2 downto 0);
      run_out         : out std_logic_vector(3 downto 0);
      state_copy      : out std_logic_vector(1 downto 0);
      last_zero_out   : out std_logic_vector(6 downto 0)  );
end component;


component huffman
port(state64_in       : in  std_logic_vector(5 downto 0);
      state8_in       : in  std_logic_vector(2 downto 0);
      pixclock        : in  std_logic;
      fastclock       : in  std_logic;
```

```
      reset_huf       : in  std_logic;
      coded_long      : in  std_logic_vector(11 downto 0);
      category_in     : in  std_logic_vector(3 downto 0);
      output_huf      : out std_logic_vector(7 downto 0);
      line_flag_huf   : out std_logic;
      field_blank_huf : out std_logic;
      firstline_huf   : out std_logic;
      state_coder_in  : in  std_logic_vector(2 downto 0);
      run_in          : in  std_logic_vector(3 downto 0);
      buf_state       : in  std_logic_vector(1 downto 0);
      last_zero       : in  std_logic_vector(6 downto 0);
      quantisation    : in  std_logic  );
end component;

signal    hufdata      : std_logic_vector(11 downto 0);
signal    huf_category    : std_logic_vector(3 downto 0);
signal    huf_state    : std_logic_vector(2 downto 0);
signal    huf_run      : std_logic_vector(3 downto 0);
signal    last_zero    : std_logic_vector(6 downto 0);
signal    state_test_copy    : std_logic_vector(1 downto 0);

begin

huf_cache : data_buf
port map(
  data_in=>data_in,
  state64_in=>state64_in,
  state8_in=>state8_in,
  state8_in_old=>state8_in_old,
  pixclock=>pixclock,
  fastclock=>fastclock,
  reset_huf=>reset_huf,
  coded_long_out=>hufdata,
  category=>huf_category,
  state_coder_out=>huf_state,
  run_out=>huf_run,
  state_copy=>state_test_copy,
  last_zero_out=>last_zero  );

huf_coder : huffman
port map(
  state64_in=>state64_in,
  state8_in=>state8_in,
  pixclock=>pixclock,
  fastclock=>fastclock,
  reset_huf=>reset_huf,
  coded_long=>hufdata,
  category_in=>huf_category,
  output_huf=>output_huf,
  line_flag_huf=>line_flag_huf,
  field_blank_huf=>field_blank_huf,
  firstline_huf=>firstline_huf,
  state_coder_in=>huf_state,
  run_in=>huf_run,
  buf_state=>state_test_copy,
  last_zero=>last_zero,
  quantisation=>quantisation  );

end architecture huf_comb_behavior;

configuration huf_comb_cfg of huf_comb is
for huf_comb_behavior
-- synopsys translate_off
  for all : data_buf use configuration work.data_buf_cfg;
  end for;
  for all : huffman use configuration work.huffman_cfg;
  end for;
-- synopsys translate_on
end for;
end huf_comb_cfg;
```

9.11 Die Zusammenführung aller Baublöcke

jpeg.vhdl

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL, IEEE.NUMERIC_STD.all;
use WORK.ALL;

entity jpeg is
port (pixclock          : in  std_logic;
      fastclock         : in  std_logic;
      input             : in  std_logic_vector(7 downto 0);
      reset             : in  std_logic;
      output            : out std_logic_vector(7 downto 0);
      line_flag         : out std_logic;
      field_blank       : out std_logic;
      firstline         : out std_logic;
      quantisation      : in  std_logic;
      quant_analog      : in  std_logic;
      testpic           : in  std_logic;
      nojpg             : in  std_logic;
      test_bilddata_out : in  std_logic;
      pixclock_test     : out std_logic    );
end entity jpeg;

architecture jpeg_behavior of jpeg is

component comb3
port (pixclock          : in  std_logic;
      fastclock         : in  std_logic;
      reset             : in  std_logic;
      input             : in  std_logic_vector(7 downto 0);
      state8_out        : out std_logic_vector(2 downto 0);
      state8_old_out    : out std_logic_vector(2 downto 0);
      state64_out       : out std_logic_vector(5 downto 0);
      quantisation      : in  std_logic;
      quant             : out std_logic_vector(10 downto 0);
      testpic           : in std_logic;
      line_flag_nojpg   : out std_logic;
      rsd_data_nojpg    : out std_logic_vector(7 downto 0);
      test_bilddata_out : in  std_logic;
      pixclock_test     : out std_logic    );
end component;

component huf_comb
port (data_in         : in  std_logic_vector(10 downto 0);
      state64_in      : in  std_logic_vector(5 downto 0);
      state8_in       : in  std_logic_vector(2 downto 0);
      state8_in_old   : in  std_logic_vector(2 downto 0);
      pixclock        : in  std_logic;
      fastclock       : in  std_logic;
      reset_huf       : in  std_logic;
      quantisation    : in  std_logic;
      output_huf      : out std_logic_vector(7 downto 0);
      line_flag_huf   : out std_logic;
      field_blank_huf : out std_logic;
      firstline_huf   : out std_logic);
end component;

-- Wires bitween blocks.
signal    state64      : std_logic_vector(5 downto 0);
signal    state8      : std_logic_vector(2 downto 0);
signal    state8_old    : std_logic_vector(2 downto 0);
signal    quant      : std_logic_vector(10 downto 0);

signal    line_flag_nojpg    : std_logic;
signal    rsd_data_nojpg    : std_logic_vector(7 downto 0);

signal    quant_comb3  : std_logic;

signal    output_jpg    : std_logic_vector(7 downto 0);
signal    line_flag_jpg    : std_logic;
signal    field_blank_jpg    : std_logic;
signal    firstline_jpg    : std_logic;
```

```
signal    line_flag_counter  : unsigned(6 downto 0);
-- Wires bitween blocks.

begin
process(quant_analog, quantisation)
begin
quant_comb3<=quantisation and not(quant_analog);
end process;

combination3 : comb3
port map (
  pixclock=>pixclock,
  fastclock=>fastclock,
  reset=>reset,
  input=>input,
  state8_out=>state8,
  state8_old_out=>state8_old,
  state64_out=>state64,
  quantisation=>quant_comb3,
  quant=>quant,
  testpic=>testpic,
  line_flag_nojpg=>line_flag_nojpg,
  rsd_data_nojpg=>rsd_data_nojpg,
  test_bilddata_out=>test_bilddata_out,
  pixclock_test=>pixclock_test  );

huf_combi : huf_comb
port map(
  data_in=>quant,
  state64_in=>state64,
  state8_in=>state8,
  state8_in_old=>state8_old,
  pixclock=>pixclock,
  fastclock=>fastclock,
  reset_huf=>reset,
  quantisation=>quantisation,
  output_huf=>output_jpg,
  line_flag_huf=>line_flag_jpg,
  field_blank_huf=>field_blank_jpg,
  firstline_huf=>firstline_jpg  );

-- Take eighter the output of the JPEG-algorithm
-- or the output behind the RSD-stage.
process (nojpg,line_flag_nojpg,rsd_data_nojpg,output_jpg,line_flag_jpg,
         field_blank_jpg,firstline_jpg,reset,line_flag_counter)
begin
if (nojpg='1') then
  output<=rsd_data_nojpg;
  line_flag<=line_flag_nojpg;
  if (line_flag_counter="0000000") then -- After 128 pixel-rows (each with 64 pixels).
    field_blank<=not(line_flag_nojpg);
    firstline<=line_flag_nojpg;
  else  field_blank<='0';
    firstline<='0';
  end if;
else  line_flag<=line_flag_jpg;
  output<=output_jpg;
  field_blank<=field_blank_jpg;
  firstline<=firstline_jpg;
end if;
end process;

-- If the raw output of RSD or the bits of the testpic are given out,
-- it is nessecary to count the output-bursts to know when
-- to apply field_blank and firstline
process (reset,line_flag_nojpg)
begin
if (reset='0') then
  line_flag_counter<="0000000";
elsif falling_edge(line_flag_nojpg) then
  line_flag_counter<=line_flag_counter+"0000001";
end if;
end process;
```

```
end architecture jpeg_behavior;

configuration jpeg_cfg of jpeg is
for jpeg_behavior
-- synopsys translate_off
  for all : huf_comb use configuration work.huf_comb_cfg;
  end for;
-- synopsys translate_on
end for;
```

9.12 Das Gesamtprojekt

top.vhdl

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL, IEEE.NUMERIC_STD.all, STD.TEXTIO.all;
use WORK.ALL;

entity top is
-- modified copy of Puegners CMOSKAMERA.vhd
port (clk_in          : in  std_logic; -- fastclock
      reset           : in  std_logic;
      Coeff           :out std_logic_vector (7 downto 0); --inout
      CoeffInClkEn    :out std_logic; --inout
      CoeffOutClk     :out std_logic; --inout
      CoeffInClk      :out std_logic; --inout
      Add             :out std_logic_vector (6 downto 0); --inout
      B               :out std_logic_vector (2 downto 0); --inout
      ShiftClk        :out std_logic; --inout
      PixelRowReset   :out std_logic; --inout
      ro_cp           :out std_logic; --inout
      ro_reset        :out std_logic; --inout
      sh_samp         :out std_logic; --inout
      sh_reset        :out std_logic; --inout
      SIwr            :out std_logic; --inout
      SIsum           :out std_logic; --inout
      ADCclk          :out std_logic_vector (1 downto 0); --inout
      ADCreset        :out std_logic; --inout
      da1,db1,dc1,dd1,da2,db2,dc2,dd2 : in  std_logic;
      cs2, rw, iostrb : in  std_logic;  --, cs3, cs4 (not used yet)
      int3, rdy       : out std_logic;
      addr_dsp        : in  std_logic_vector(11 downto 0);
      data_dsp        : inout std_logic_vector(15 downto 0);
      quantisation    : in  std_logic;
      quant_analog    : in  std_logic;
      testpic         : in  std_logic;
      nojpg           : in  std_logic;
      testout         : out std_logic_vector (7 downto 0);
      test_bilddata_out : in  std_logic    );
-- modified copy of Puegners CMOSKAMERA.vhd
end entity top;

architecture top_behavior of top is

component jpeg is
port (pixclock          : in  std_logic;
      fastclock         : in  std_logic;
      input             : in  std_logic_vector(7 downto 0);
      reset             : in  std_logic;
      output            : out std_logic_vector(7 downto 0);
      line_flag         : out std_logic;
      field_blank       : out std_logic;
      firstline         : out std_logic;
      quantisation      : in  std_logic;
      quant_analog      : in  std_logic;
      testpic           : in  std_logic;
      nojpg             : in  std_logic;
      test_bilddata_out : in  std_logic;
      pixclock_test     : out std_logic    );
end component;
```

```
-- from M. Puegner
component BYTE_TO_WORD
port (clk_in, aktline_in, firstline_in, field_blank, reset : in std_logic;
      data_in  : in std_logic_vector (7 downto 0);
      clk_out, aktline_out, firstline_out : out std_logic;
      data_out : out std_logic_vector (15 downto 0)    );
end component;

-- Angabe der generischen Parameter der Ausgabestufe
constant  Blockgroesse    : integer := 2;
  -- (mit jeweils 64 Worten, d.h Grösse entspricht D*64)  D: 1..8
constant  Pufferanzahl    : integer := 2; -- von 1..8
-- es ist notwendig die benötigte Adressbreite selbst zu ermitteln, da
-- die generische Generierung bisher nicht synthesefähig beschrieben
-- werden konnte
-- 1 Pufferblock ist 256 Worte groß, wird also mit 8 Bit adressiert
-- --> Adressbreite= ld(Pufferanzahl*256)
constant   Adressbreite    : integer := 9;

-- Schluessler
-- 4LSB Version (1)
constant  FPGA_Ident    : integer := 16#92B1#;
-- Schluessler

component AUSGABE
generic (
       Blockgroesse: integer range 1 to 8;
       Pufferanzahl: integer range 1 to 8;
       Adressbreite: integer range 8 to 11;
       FPGA_Ident: integer range 1 to 16#FFFF#       ); --Schluessler
    port (reset, clk, line, firstline, cs, rw, iostrb: in std_logic;
    interrupt, overflow: out std_logic;
    addr_dsp: in std_logic_vector(11 downto 0);
    data_dsp: out std_logic_vector(15 downto 0);
    data_in: in std_logic_vector(15 downto 0);
    tristate: out std_logic;
    rdy: out std_logic    ); -- Schluessler
  end component;
-- from M. Puegner

-- Testoutput
--file outoutput:  text open write_mode is "out.txt";
--signal    firstoutput    : std_logic;
--signal    last_output    : std_logic_vector(7 downto 0);
-- Testoutput

signal    interrupt    : std_logic;
signal    tristate    : std_logic;
signal    input      : std_logic_vector(7 downto 0);
-- M. Puegner
signal    data_dsp_out_bild  : std_logic_vector(15 downto 0);
signal    data_dsp_out_steuer  : std_logic_vector(15 downto 0);
    -- no output data from controler yet, but possible
signal    data_dsp_out    : std_logic_vector(15 downto 0);
-- M. Puegner
-- Top level I/O.  End.

-- Wires bitween blocks.
signal    clk      : std_logic;
--output of jpeg
signal    output      : std_logic_vector(7 downto 0);
signal    line_flag    : std_logic;
signal    field_blank    : std_logic;
signal    firstline    : std_logic;
-- M. Puegner
signal    clk_ka      : std_logic;
signal    aktline_ka    : std_logic;
signal    aktline_ka2    : std_logic; -- for bugfixing by Hildebrandt
signal    aktline_ka_delay  : std_logic;
signal    firstline_ka    : std_logic;
signal    data_ka      : std_logic_vector(15 downto 0);
signal    tristate_bild    : std_logic;
signal    tristate_steuer    : std_logic;
-- M. Steidl
```

```
signal  busy_clock      :std_logic;
signal  pixclk_jpeg     :std_logic;
signal  Coeff_buf       :std_logic_vector (7 downto 0);
signal  CoeffInClkEn_buf:std_logic;
signal  CoeffOutClk_buf :std_logic;
signal  CoeffInClk_buf  :std_logic;
signal  ADCclk_buf      :std_logic_vector (1 downto 0);
signal  ADCreset_buf    :std_logic;
signal  Add_buf         :std_logic_vector (6 downto 0);
signal  B_buf           :std_logic_vector (2 downto 0);
signal  ShiftClk_buf    :std_logic;
signal  PixelRowReset_buf :std_logic;
signal  ro_cp_buf       :std_logic;
signal  ro_reset_buf    :std_logic;
signal  sh_samp_buf     :std_logic;
signal  sh_reset_buf    :std_logic;
signal  SIwr_buf        :std_logic;
signal  SIsum_buf       :std_logic;
signal  ADCreset_int  : std_logic;
-- Hildebrandt
signal    pixclk_jpeg_dummy    : std_logic; -- for BUFGS
signal    pixclk_jpeg_old      : std_logic;
signal    pixclk        : std_logic;
signal    pixclk_pueg      : std_logic;
signal    pixclock_test      : std_logic;
signal    pixclock_test_delay    : std_logic;
-- Schluessler
signal    rdy_steuerung,rdy_ausgabe,rdy_dsp  : std_logic;
signal    interrupt_out        : std_logic;
signal    overflow        : std_logic;
-- Signals for Steidl (not used)
signal    pixresetoffset         : std_logic_vector (6 downto 0);
signal    kernaddr        : std_logic_vector (9 downto 0);
-- Wires bitween blocks.

-- camera control M. Steidl
component control is
port (clk             :in std_logic;
      reset       :in std_logic;
      pixresetoffset  :in std_logic_vector (6 downto 0);
      kernaddr        :in std_logic_vector (9 downto 0);
      Coeff           :out std_logic_vector (7 downto 0);
      CoeffInClkEn    :out std_logic;
      CoeffOutClk     :out std_logic;
      CoeffInClk      :out std_logic;
      Add             :out std_logic_vector (6 downto 0);
      B               :out std_logic_vector (2 downto 0);
      ShiftClk        :out std_logic;
      PixelRowReset   :out std_logic;
      ro_cp           :out std_logic;
      ro_reset        :out std_logic;
      sh_samp         :out std_logic;
      sh_reset        :out std_logic;
      SIwr            :out std_logic;
      SIsum           :out std_logic;
      ADCclk          :out std_logic_vector (1 downto 0);
      ADCreset        :out std_logic;
      pixclk_ralf     :out std_logic;
      busy_ralf       :out std_logic       );
end component;

-- Schluessler
component PINOUT_OC
port (p_in  : in   std_logic;
      p_out : out  std_logic   );
end component;


begin
pixclk_jpeg_dummy<=pixclk_jpeg;

-- Schluessler
-- enable for synthesis
rdy_oc_OutPin: PINOUT_OC
```

```
port map(
  p_in => rdy_dsp,
  p_out => rdy   );

int3_oc_OutPin: PINOUT_OC
port map(
  p_in => interrupt_out,
  p_out => int3   );

-- enable for simulation
--rdy<=rdy_dsp;
--int3<=interrupt_out;


-- Schluessler
  testout(0)<= interrupt_out;
  testout(1)<= clk_ka;
  testout(2)<= aktline_ka;
  testout(3)<= firstline_ka;
  testout(4)<= rdy_dsp;
  testout(5)<= not(data_dsp(0));
  testout(6)<= overflow;
  testout(7)<= '0';

-- Modul gibt '1' aus, wenn es nicht selektiert ist (/CSn)
-- oder warten will
--      gibt '0' aus, wenn es fertig ist
-- >> '0' muss weitergegeben werden

  rdy_dsp <= rdy_ausgabe; --and rdy_steuerung
-- no rdy_steuerung needed, because no data
-- from controler have to be given out


-- Steidl
kamerasteuerung : control
port map(
  clk             => clk_in,
  reset           => reset,
  pixresetoffset  => pixresetoffset,
  kernaddr        => kernaddr,
  Coeff           => Coeff,
  CoeffInClkEn    => CoeffInClkEn,
  CoeffOutClk     => CoeffOutClk,
  CoeffInClk      => CoeffInClk,
  Add             => Add,
  B               => B,
  ShiftClk        => ShiftClk,
  PixelRowReset   => PixelRowReset,
  ro_cp           => ro_cp,
  ro_reset        => ro_reset,
  sh_samp         => sh_samp,
  sh_reset        => sh_reset,
  SIwr            => SIwr,
  SIsum           => SIsum,
  ADCclk          => ADCclk,
  ADCreset        => ADCreset,
  pixclk_ralf     => pixclk_jpeg,
  busy_ralf       => busy_clock     );

jpeg_algorithm : jpeg
port map(
  pixclock=>pixclk,
  fastclock=>clk,
  input=>input,
  reset=>reset,
  output=>output,
  line_flag=>line_flag,
  field_blank=>field_blank,
  firstline=>firstline,
  quantisation=>quantisation,
  quant_analog=>quant_analog,
  testpic=>testpic,
  nojpg=>nojpg,
```

```
  test_bilddata_out=>test_bilddata_out,
  pixclock_test=>pixclock_test    );


-- from M. Puegner
konverter: BYTE_TO_WORD
port map(
     clk_in => pixclk_pueg,
     aktline_in => line_flag,
     firstline_in => firstline,
     field_blank => field_blank,
     reset => reset ,
     data_in => output,
     clk_out => clk_ka,
     aktline_out => aktline_ka,
     firstline_out => firstline_ka,
     data_out => data_ka     );

out_modul: AUSGABE
generic map (
   Blockgroesse,
   Pufferanzahl,
   Adressbreite,
         FPGA_Ident   )
port map(
     reset => reset ,
     clk => clk_ka,
     line => aktline_ka_delay,
     firstline => firstline_ka,
     cs => cs2,
     rw => rw,
     iostrb => iostrb,
     interrupt => interrupt_out,
     overflow => overflow,
     addr_dsp => addr_dsp,
     data_dsp => data_dsp_out_bild,
     data_in => data_ka,
     tristate => tristate_bild,
     rdy => rdy_ausgabe     ); -- Schluessler
-- from M. Puegner

-- Hildebrandt
process (da1,db1,dc1,dd1,da2,db2,dc2,dd2)
begin
-- The input pins are mirrored because of a mistake while
-- writing the VHDL-code.
input(7)<=(dd1);
input(6)<=(dc1);
input(5)<=(db1);
input(4)<=(da1);
input(3)<=(dd2);
input(2)<=(dc2);
input(1)<=(db2);
input(0)<=(da2);
end process;

-- stop fastclock for Hildebrandt
clk<=clk_in and busy_clock;

process(clk,pixclk_jpeg_dummy)
begin
if (rising_edge(clk)) then
  pixclk_jpeg_old<=pixclk_jpeg_dummy;
end if;
end process;

process(pixclk_jpeg_dummy,busy_clock,pixclk_jpeg_old)
begin
if (busy_clock='1') then
  pixclk<=pixclk_jpeg_dummy;
else  pixclk<=pixclk_jpeg_old;
end if;
end process;
--pixclk<=pixclk_jpeg_dummy and busy_clock;
```

```
-- tristate logic from M. Puegner
-- modified
process (clk_in)
begin
if rising_edge(clk_in) then
  data_dsp_out_steuer<="0000000000000000";
  tristate_steuer<='1';-- control turned off
end if;
end process;
-- modified

process (cs2, data_dsp_out_bild, data_dsp_out_steuer)
begin
if (cs2='0') then
  data_dsp_out<=data_dsp_out_bild;
else  data_dsp_out<=data_dsp_out_steuer;
end if;
end process;

process (tristate_steuer, tristate_bild, data_dsp_out)
begin
if (tristate_steuer='0' or tristate_bild='0') then
  data_dsp<=data_dsp_out;
else  data_dsp<=(OTHERS => 'Z');
end if;
end process;
-- tristate logic from M. Puegner


-- for pure  (RSD-corrected) data to output (Hildebrandt)
process(pixclk,pixclock_test_delay,nojpg)
begin
if (nojpg='1') then
  pixclk_pueg<=pixclock_test_delay;
else  pixclk_pueg<=pixclk;
end if;
end process;

-- delay falling_edge(aktline_ka) for 1/2 fastclock
aktline_ka_delayer : process (aktline_ka,aktline_ka2,clk_in)
-- this process is written for byte_to_word.vhd (Puegner)
-- where aktline_ka has to change to 0 _after_ a falling_edge(pixclock)
begin
if (rising_edge(clk_in)) then
  aktline_ka2<=aktline_ka;
end if;

if (aktline_ka2='1') then
  aktline_ka_delay<=aktline_ka2;
else  aktline_ka_delay<=aktline_ka;
end if;
end process aktline_ka_delayer;

-- delay pixclock_test for 1/2 fastclock
pixclock_test_delayer : process (pixclock_test,clk_in)
begin
if (rising_edge(clk_in)) then
  pixclock_test_delay<=pixclock_test;
end if;
end process pixclock_test_delayer;
-- for pure  (RSD-corrected) data to output (Hildebrandt)


end architecture top_behavior;

-- synopsys translate_off
configuration top_cfg of top is
for top_behavior
  for all : jpeg use configuration work.jpeg_cfg;
  end for;
        for kamerasteuerung:control
          for controlarch
```

```
            for all : sicontrol use configuration work.sicontrol_cfg;
            end for;
          end for;
        end for;
end for;
end top_cfg;
-- synopsys translate_on
```