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

Diplomarbeit Nr. 3092

Effiziente FPGA Implementierung des JPEG-LS Encoders mit Xilinx System Generator

Constantin Sibianu

Studiengang:	Informatik
Prüfer:	Prof. Dr. -Ing. Sven Simon
Betreuer:	M. Sc. Zhe Wang
begonnen am:	15. Oktober 2010
beendet am:	16. April 2011
CR-Klassifikation:	B.5.2, B.7.1, E.4, I.4.2

Vorwort

Die vorliegende Arbeit markiert den Höhepunkt meines Studiums und ist das Ende dieses Lebensabschnittes. Ich stellte mich den Herausforderungen, die während meiner Ausbildung als Diplominformatiker auftauchten, und ich werde mich stets mit Freude an diese Zeit erinnern. Ich habe mir während dieses Studiums nicht nur sehr viel Fachwissen angeeignet, sondern auch sehr viele Erfahrungen gesammelt, die mich in meinem Leben weiter gebracht haben und die für meinen späteren Werdegang sehr von Nutzen sein werden.

An dieser Stelle möchte ich meiner Frau besonderen Dank schenken, für die Geduld und die moralische Unterstützung, die sie mir entgegenbrachte.

Prof. Dr. -Ing. Sven Simon danke ich für das Vertrauen und die Möglichkeit diese Diplomarbeit innerhalb seiner Abteilung zu schreiben.

Vielen Dank auch an meinen Betreuer M. Sc. Zhe Wang für seine tatkräftige Unterstützung und sehr gut strukturierte Planung dieser Arbeit, sowie für seine wertvolle Zeit, die er für mich opfern musste. Weiterhin bedanke ich mich noch bei seinem Zimmerkollegen Dipl.-Inf. Simeon Wahl und Dipl.-Math. techn. Philipp Werner für ihre wertvollen Tipps, die mir sehr geholfen haben.

Zum Schluss möchte ich mich noch bei allen bedanken, die an mich geglaubt haben, und mir damit sehr viel Kraft gegeben haben um mein Studium zu beenden.

Inhaltsverzeichnis

Abbildungsverzeichnis

Verzeichnis der Algorithmen

1. Einleitung

In dieser Diplomarbeit beschäftige ich mich explizit mit der FPGA-Implementierung des JPEG-LS-Algorithmus mit dem Designer-Tool System Generator von der Firma Xilinx. Dabei soll untersucht werden wie weit und im besten Fall wie einfach man aus diesem Algorithmus ein synthetisierbares System-Generator-Modell erzeugen kann, ohne über ein fundiertes Wissen im Bereich des digitalen Hardwaredesigns zu verfügen. Dafür eignet sich System Generator besonders gut, da man damit ein simulink-ähnliches Modell aufbauen kann und dieses später theoretisch in VHDL- oder Verilog-Code umwandeln kann. Wichtig dabei sind die MCode-Blöcke, mit deren Hilfe sich MATLAB-Funktionen in System Generator einbinden lassen können. Diese Blöcke sollen die Basis für einen schnellen und einfachen Modellaufbau sein, mit denen die meisten Teile des JPEG-LS-Algorithmus implementieren werden sollen.

1.1. FPGAs

1984 wurde die Firma Xilinx gegründet und brachte 1985 die ersten FPGAs (Field-Programmable Gate Array) auf den Markt [Xil], die im Prinzip nur einen einfachen Schnittstellenchip darstellten [Ins]. Die ständige Weiterentwicklung dieser Technologie führte dazu, dass die FPGAs die benutzerdefinierten ASICs (Application-Specific Integrated Circuits) und Prozessoren sowohl in der Signalverarbeitung als auch für Steuer- und Regelanwendungen immer mehr verdrängten.

Der Grund dieser Beliebtheit ist wie so oft die Leistungsfähigkeit der FPGAs, die grob gesagt, die Vorteile der ASICs und der Prozessoren vereint. FPGAs sind durch die Wiederprogrammierbarkeit genauso flexibel wie Programme, die auf einem Prozessor laufen, allerdings wird die Leistung nicht durch Anzahl der Prozessorkerne eingeschränkt. Dieser Vorteil kommt durch die parallelisierbare Ausführung, die die FPGAs besitzen. Im Gegensatz zu ASICs sind die wiederprogrammierbaren Siliziumchips hardwaregetaktet und trotz gleicher Kosteneffizienz verfügen sie über eine höhere Zuverlässigkeit.

Die Funktionalität eines FPGAs wird durch eine Konfigurationsdatei oder einen Bitstream beschrieben, die von einem Programmierer erstellt und am Ende kompiliert wird. Somit kann ein FPGA jederzeit rekonfiguriert werden indem er eine neue Schaltungskonfiguration implementiert bekommt.

Dadurch dass beim Programmieren eines FPGAs zum großen Teil mit Logikblöcken gearbeitet wurde, stand diese Technologie nur Anwendern zur Verfügung, die sich in die Materie lang genug eingearbeitet haben. Dieser Aufwand ist heutzutage nicht mehr unbedingt notwendig, da viele sehr komplexe Designwerkzeuge in der Lage sind Blockdiagrame, oder C- und MATLAB-Code, zu digitalisieren und in Hardwareschaltungen umzuwandeln.

1.2. MATLAB

Bei MATLAB handelt es sich um eine kommerzielle Software der Firma The Mathworks (`http://www.mathworks.de`), die sich in erster Linie für Kalkulationen mit Hilfe von Matrizen eignet, sowie für die grafische Darstellung der Ergebnisse. Daher auch der Name der Software: MATrix LABoratory.

MATLAB wurde Ende der 70er Jahre von Cleve Moller an der Universität New Mexiko entwickelt. Die Software sollte den Studenten helfen, indem sie die Fortran-Bibliotheken LINPACK und EISPACK der linearen Algebra nach Außen zugänglich machten, ohne dass man irgendwelche Fortran-Kenntnisse besitzen muss.

1984 gründete Cleve Moller zusammen mit Jack Little und Steve Bangert die Firma „The Mathworks“ und machten aus MATLAB ein kommerzielles Produkt, das am Anfang nur eine Funktionssammlung „Control System Toolbox“ besaß.

Außer bei Studenten, die bis heute noch über günstige Lizenzen verfügen können, wurde MATLAB vor allem in der Regelungstechnik immer beliebter.

Unter anderem ist MATLAB eine Programmiersprache. Wie bei den meisten Programmiersprachen lassen sich in MATLAB auch Funktionen definieren. Diese werden dann in Dateien mit der Endung .m gespeichert und können z.B. aus der Kommandozeile aufgerufen werden, wenn sich diese im aktuellen Suchpfad befinden. Diese M-Files können, wie wir später sehen werden, auch für die Definition von MCode-Blöcke in System Generator benutzt werden. Ich werde in dieser Arbeit versuchen, die wichtigsten Komponenten mit solchen MCode-Blöcken zu implementieren, also in erster Linie nur mit MATLAB-Programmierung.

1.3. Simulink

Simulink, ein weiteres Produkt von der Firma The Mathwoks, dient zur Modellierung von Systemen, egal ob technische, mathematische oder physikalische. Für die Implementierung eines solchen Systems werden sogenannte Blöcke benötigt, die zum Teil direkt in den

Simulink-Bibliotheken zu finden sind, aber auch von anderen Herstellern bezogen werden können. Die Daten fließen von Block zu Block über sogenannte Verbindungslinien. So entsteht für den Datenfluss ein gerichteter Graph, der für die spätere Simulation des Systems notwendig ist. Mit Hilfe von Toolboxen, wie z.B. Real-Time Workshop oder HDL-Coder, lässt sich aus MATLAB/Simulink ein fertiger Code generieren (z.B. C oder VHDL), der später beispielsweise auf Mikroprozessoren oder FPGAs ausgeführt werden kann.

1.4. Xilinx System Generator

System Generator für DSP (digital signal processor = Prozessor für digitale Signalverarbeitung) ist ein Tool für die Herstellung von DSP-Systemen für FPGAs. Dabei wird die Modellierung und Codegenerierung direkt aus MATLAB und Simulink unterstützt. Für die Herstellung solcher Systeme stellt Xilinx einen recht großen Blockset zur Verfügung, der mit vielen Funktionen, wie z.B. für Signalverarbeitung, arithmetische Berechnungen, logische Schaltungen oder Speicher-Blöcke, bestückt ist. Abbildung 1.1 zeigt die drei Standard-Xilinx-Libraries in Simulink (Xilinx Blockset, Xilinx Reference Blockset und Xilinx XtremeDSP Kit).

System Generator bietet eine automatische Codegenerierung in VHDL und Verilog direkt aus Simulink.

Ein anderes Feature von Simulink ist die Hardware-Co-Simulation. Dabei kann eine laufende Hardware mit MATLAB und Simulink unterstützt werden. Für die Kommunikation kann entweder Ethernet (10/100/Gigabit) oder JTAG in Anspruch genommen werden.

1.5. Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Hauptelemente von JPEG-LS: Kurze Erklärung des JPEG-LS-Algorithmus und seiner einzelnen Komponenten: Prädiktion, Gradientendetektion, Qunatisierung, Berechnung des Prädiktionsfehlers und die anschließende Golomb/Rice-Codierung.

Kapitel 3 – Modellaufbau: Implementierung des Algorithmus mit System Generator. Schwerpunkt dieses Kapitel ist der Umgang mit System Generator und die bei der Implementierung aufgetauchten Probleme.

Kapitel 5 – Fazit und Ausblick Fazit zu der Benutzung von System Generator, sowie mögliche Erweiterungen und Verbesserungen dieser Arbeit.

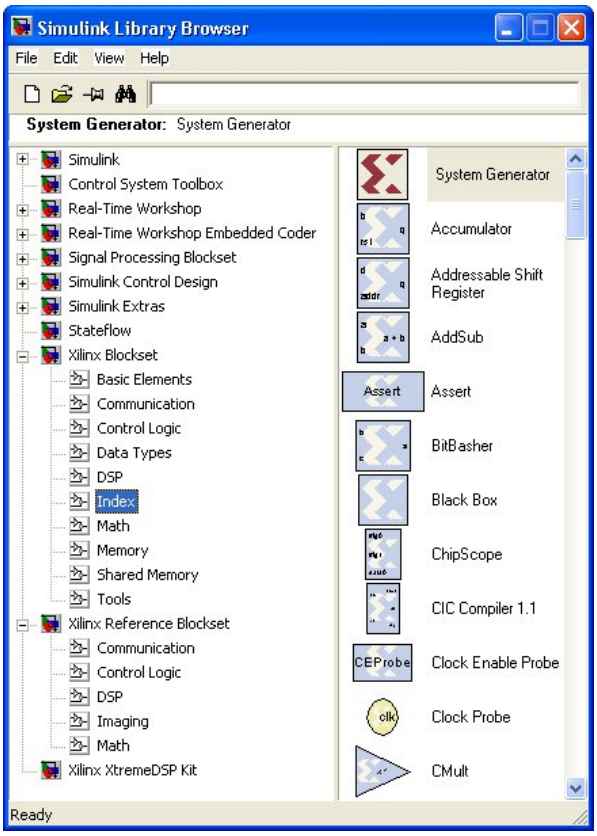


Abbildung 1.1.: Xilinx Library

2. Hauptelemente von JPEG-LS

JPEG-LS ist ein Standard aus dem JPEG-Komitee für verlustlose und fast verlustlose Standbilder. Dieser Standard wird charakterisiert durch eine sehr einfacher Prädiktion und adaptiven Kompressionsalgorithmus, sowie die Entropie-Codierung, die mit Hilfe des Golomb-Rice-Codes erfolgt. Hinter dem JPEG-LS steckt der LOCO-I-Algorithmus (LOw COmplexity LOssless COmpression for Images) von Hewlett -Packard. [MJW96] Das Besondere daran ist unter Anderem das Finden der glatten Bereiche im Bild und dessen Codierung. In diesem sogenannten Lauflängenmodus wird nicht jedes Pixel einzeln codiert, sondern es wird eine größere Anzahl an konsekutiven Pixeln in einem Codewort geschrieben. An dieser Stelle möchte ich darauf hinweisen, dass dieser Lauflängenmodus, der im unteren Teil der Abbildung 2.1 zu sehen ist, in dieser Arbeit nicht implementiert wird.

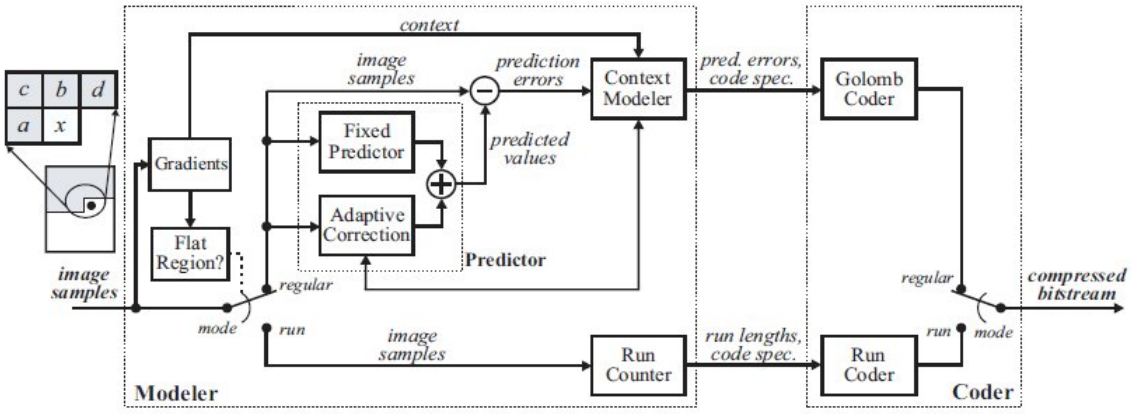


Abbildung 2.1.: JPEG-LS Block-Diagram [MJW00]

2.1. LOCO-I

Bei LOCO-I handelt es sich um einen verlustfreien Kompressionsalgorithmus, der in erster Linie auf einer Kontextmodellierung basiert. Trotz der anschließenden, sehr einfachen Golomb-Rice-Codierung, erreicht der Algorithmus in einem Schritt vergleichbare und sogar bessere Kompressionsraten als die meisten state-of-the-art Algorithmen und das mit viel

weniger Komplexität. Deswegen wurde eine vereinfachte Version des LOCO-I-Algorithmus von dem ISO-Komitee als Standard für die verlustlose Kompression für Anwendungen mit geringer Komplexität vorgeschlagen.

Eine der wichtigsten Komponente des Algorithmus, die Kontextmodellierung, geht Pixel für Pixel durch und versucht Folgerungen für den nächsten Pixel anhand einer bedingten Wahrscheinlichkeitsverteilung zu schließen. Durch die sequentielle Bearbeitung wird die Verteilung aus den vorherigen Werten berechnet und für die Codierung zur Verfügung gestellt. Eine andere Möglichkeit ein Bild zu komprimieren wäre eine Berechnung in zwei Schritten. Diese, auch sehr verbreitete Alternative, liest im ersten Schritt das gesamte Bild und berechnet dabei die dazugehörige Verteilung, die danach im zweiten Schritt als Informationsheader zum Decoder geschickt wird.

In den meisten state-of-the-art Algorithmen wird die Berechnung der Wahrscheinlichkeitsverteilung wiederum in drei Schritte gesplittet.

1. Als erstes wird eine Prädiktion durchgeführt, indem anhand der bisher gelesenen Pixel ein deterministischer Wert für den folgenden Pixel erraten wird.

2. Im zweiten Schritt wird dann durch die Erkenntnisse der gelesenen Pixel der Kontext bestimmt.

3. Im letzten Schritt wird dann ein Modell oder ein Signal mit den Werten der Prädiktionsabweichung gebildet.

Bei dem Prädiktionsschritt wird ein adaptiv optimierter, kontextabhängiger Prädiktor benutzt und für die Modellierung ein quantisierter Kontext mit variabler Länge. Die Abweichung der Prädiktion wird mit Hilfe des Golomb-Rice-Code codiert um eine bessere Codelänge zu erreichen. [MJW00]

2.2. Wichtige Variablen für JPEG-LS

Bevor man mit dem Algorithmus anfängt müssen zunächst ein paar Parameter initialisiert werden.

Für die spätere Modellierung brauchen wir folgende 4 Arrays, die im regulären Modus nur eine Länge von 365 benötigen. Die letzten zwei Bits werden in dieser Arbeit trotzdem implementiert, um eine spätere Erweiterung des Lauflängenmodus zu vereinfachen.

- N[0 .. 366] - in diesem Array wird die Anzahl der pro Kontext ausgeführten Operationen gespeichert.
- A[0 .. 366] - hier wird die Summe aller Prädiktionsfehler im Betrag gespeichert.
- B[0 .. 366] - sind die Werte für die Biaskompensation.
- C[0 .. 366] - beinhaltet die Korrekturwerte für die spätere Berechnung des Prädiktionswertes.

Während B und C mit Nullen und N mit Einsen initialisiert werden, wird A mit dem folgenden Wert gefüllt:

$$A[\,] = max(2, \lfloor (RANGE + 32)/64 \rfloor$$

In unserem Fall ist die Variable $RANGE$, die im Kapitel 2.6 erklärt wird, gleich 256, so dass unser Array A mit dem Wert 4 initialisiert wird.

Nach jeden Schritt des Algorithmus wird der aktuelle Wert von $N[]$ mit einer Konstante RESET verglichen, die einen Default-Wert von 64 hat. Ist dieser Wert erreicht, werden die aktuellen Werte für $A[]$, $B[]$ und $N[]$ halbiert. Dieser RESET-Wert ist also verantwortlich für die Geschwindigkeit der Adaption und für die Anfälligkeit gegenüber von Ausreißern in der Statistik. [Str09]

2.3. Prädiktion

Typischerweise wird bei Bildsignalen eine nichtlineare, vom aktuellen Kontext abhängige Prädiktion, benutzt. Links in der Abbildung 2.1 sieht man die Matrix des Bildes, das gerade gelesen wird. Sei x der Pixel, der gerade verarbeitet wird und a, b, c und d seine umgebenden Nachbarn, die den Kontext des Pixels x definieren. Für die nichtlineare Prädiktion in JPEG-LS reichen lediglich a, b und c um aus den Grauwerten eine vertikale Kante, eine horizontale Kante oder einen flächigen Farbverlauf zu erkennen. Eine vertikale Kante hat man meistens, wenn c größer als a und als b ist und b der kleinste Wert ist. Der vorauszusagende Punkt befindet sich dann höchstwahrscheinlich auf der gleichen Seite wie b und wird deswegen als Prädiktionswert genommen. Ebenfalls eine vertikale Struktur, aber mit umgedrehten Vorzeichen, ist zu vermuten wenn c der kleinste Wert ist und b der größte. Man nimmt dann dafür auch den Wert b als Prädiktionswert. Die gleiche Prozedur, nur mit a und b vertauscht, benutzt man um eine horizontale Kante zu erkennen. Bei der dritten Möglichkeit, wenn c also zwischen a und b liegt, wird mit Hilfe einer Flächenapproximation ein Schätzwert gebildet. Algorithmus 2.1 zeigt uns eine mögliche Berechnung. [JB00]

Algorithmus 2.1 Prädiktionsalgorithmus [JBoo]

$$\hat{x}(n) = \begin{cases} min(a,b), & \text{if } c \geq max(a,b) \\ max(a,b), & \text{if } c \leq min(a,b) \\ a+b-c, & \text{sonst.} \end{cases}$$

2.4. Gradientendetektion und Quantisierung

Zuerst werden drei Schwellwerte definiert, die für die spätere Quantisierung der Gradienten notwendig sind. Als Default sind diese drei Werte auf $T_1 = 3$, $T_2 = 7$ und $T_3 = 21$ gesetzt, können aber jederzeit über die definierten Modelleingänge geändert werden. Die Änderungen sind vor allem bei Bildern mit mehr als 8 Bits pro Pixel sinnvoll.

Dann gibt es noch den Parameter $NEAR$, der bei einer verlustlosen Kompression gleich 0 ist oder ungleich 0 für eine Kompression mit einer gewissen Toleranz. In dieser Arbeit behandle ich nur die verlustlose Version, deswegen wird $NEAR$ immer als 0 angenommen. Nichtsdestotrotz wird bei der späteren Implementierung auch diese Variable als Input des Modells deklariert um eine evtl. spätere Erweiterung des Lauflängenmodus zu erleichtern.

Mit Hilfe der Werte a, b, c und d werden zuerst die Gradienten des aktuellen Bildpunktes berechnet [Salo4]:

$$D1 = d - b,\ D2 = b - c,\ D3 = c - a.$$

Wenn alle drei D-Variablen kleiner gleich $NEAR$ sind dann wird der Lauflängenmodus für den Coder angestoßen, der aber in dieser Arbeit nicht implementiert wird. Ich werde mich hier nur mit dem regulären Modus beschäftigen.

Das Bild 2.2 beschreibt den Verlauf der Quantisierung und die Abbildung der Werte $D1$, $D2$ und $D3$ auf einen Vektor ($Q1$, $Q2$, $Q3$).

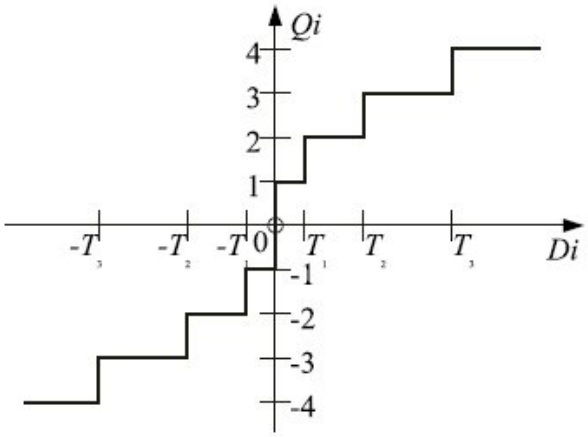


Abbildung 2.2.: Quantisierung der Gradienten [Str09]

Der Algorithmus für diese Quantisierung ist recht einfach und könnte wie in dem Beispielalgorithmus 2.2 aussehen.

Algorithmus 2.2 Quantisierungs-Algorithmus

```
procedure QUANT( )
    if D_i ⩽ −T_3 then
        Q_i ← −4
    else if D_i ⩽ −T_2 then
        Q_i ← −3
    else if D_i ⩽ −T_1 then
        Q_i ← −2
    else if D_i < −NEAR then
        Q_i ← −1
    else if D_i = NEAR then
        Q_i ← 0
    else if D_i < T_1 then
        Q_i ← 1
    else if D_i < T_2 then
        Q_i ← 2
    else if D_i < T_3 then
        Q_i ← 3
    else
        Q_i ← 4
    end if
end procedure
```

Da alle 3 Werte gleich behandelt werden, wurden die Indizes durch i ersetzt. Bei der späteren Implementierung wird dieser Algorithmus also dreimal durchgeführt um alle Q's berechnen zu können. Dies geschieht im einfachsten Fall mit Hilfe einer for-Schleife die von 1 bis 3 verläuft. Mit Hilfe dieser Quantisierung verringern wir die Anzahl der verschiedenen Kontexte auf $9 \cdot 9 \cdot 9 = 729$.

Nun können wir mit Hilfe dieser Q-Werte die sogenannte Kontextnummer berechnen. Eine Möglichkeit für diese Berechnung wäre:

$$cx = 9 \cdot (9 \cdot Q_1 + Q_2) + Q_3.$$

Da es bei den Strukturen von Bildern keine bestimmte Richtung gibt, kann man mit dem absoluten Betrag von cx weiterarbeiten, was wiederum eine Halbierung dieser Werte bedeutet. Diese werden dann auf einem Intervall von 0 bis 364 abgebildet. Falls es sich bei cx um einen negativen Wert handelt wird die Variable $SIGN$ auf -1 gesetzt, bei positiven Werten auf 1.

2.5. Korrektur des Prädiktionswertes

Für die Berechnung des Prädiktionswertes müssen wir zuerst die Variable $MAXVAL$ definieren. $MAXVAL$ gibt den größten Wert innerhalb des Scans an und lässt sich folgendermaßen berechnen: $2^p - 1$ wobei p die Anzahl der Bits pro Bildpunkt ist (in unseren Fall 8, da hier nur 8-Bit-Bilder betrachtet werden). Wir bekommen also für $MAXVAL$ einen Wert von 255.

Mit Hilfe der im vorherigen Kapitel berechneten Kontextnummer wird aus dem C-Array der richtige Wert für die Korrektur der Prädiktionswertes genommen. Je nachdem ob die Variable $SIGN$ 1 oder -1 ist, wird $C[cx]$ vom Prädiktionswert subtrahiert oder hinzuaddiert. Das Resultat wird danach auf einen gültigen Bereich zwischen 0 und $MAXVAL$ limitiert.

2.6. Berechnung des Prädiktionsfehlers

Die Konstante $NEARm2p1 = NEAR \cdot 2 + 1$, die in unserer verlustlosen Version immer gleich 1 ist, wird für den Lauflängenmodus benötigt und kann für unsere Implementierung ganz weggelassen werden.

Aus den oben genannten Parameter können wir den $RANGE$ berechnen, den wir später für das Zentrieren des Error-Wertes benötigen:

$$RANGE = \left\lfloor \frac{MAXVAL + 2 \cdot NEAR}{NEARm2p1} \right\rfloor + 1$$

Da wir die in der Funktion benutzten Variablen als Konstanten annehmen können, bekommen wir für $RANGE$ einen Wert von 256.

Auch die Berechnung des Prädiktionsfehlers ist abhängig von der Variable $SIGN$ (Alg. 2.3).

Algorithmus 2.3 Prädiktionsfehler [Salo4]

if $SIGN < 0$ **then**
  $e \leftarrow \hat{x} - x$;
else
  $e \leftarrow x - \hat{x}$;
end if

Der bis jetzt berechnete Error-Wert liegt theoretisch zwischen $-MAXVAL$ und $+MAXVAL$. Diesen Wert können wir aber nochmal auf das Intervall $[-\hat{x}, RANGE - \hat{x})$ reduzieren, da der Schätzwert $\hat{x}$ sowohl dem Encoder als auch dem Decoder bekannt ist. Mit der folgenden Modulo-Operation (Alg. 2.4) lässt sich der Error-Wert auf einen Bereich von $(-RANGE/2)$ bis $(RANGE/2 - 1)$ zentrieren:

Algorithmus 2.4 Zentrieren des Intervals [Stro9]

if $e < -RANGE/2$ **then**
  $e \leftarrow e + RANGE$;
else if $e > (RANGE - 1)/2$ **then**
  $e \leftarrow e - RANGE$;
end if

Dieser neuberechnete Wert wird für die Kalkulation des Golomb/Rice-Codes auf einen positiven Wert e_M abgebildet. Für den Rest des Algorithmus wird aber trotzdem der vorzeichenbehaftete Prädiktionsfehlers benutzt. Die Berechnung von e_M wird nach dem Algorithmus 2.5 berechnet:

Algorithmus 2.5 Berechnung von e_M [MJWoo]

if $((k = 0) \wedge (NEAR = 0) \wedge (2 \cdot B[cx] \leqslant N[cx]))$ **then**

  if $e < 0$ **then**
    $e_M \leftarrow -2 \cdot (e + 1)$
  else
    $e_M \leftarrow 2 \cdot e + 1$
  end if
else

  if $e < 0$ **then**
    $e_M \leftarrow -2 \cdot e - 1$
  else
    $e_M \leftarrow 2 \cdot e$
  end if
end if

2.7. Golomb/Rice-Codierung

Neben dem im Kapitel davor berechneten positiven Fehler brauchen wir für die Codierung noch einen Parameter k, der in dem Golomb/Rice-Algorithmus für die Auswahl einer geeigneten Codetabelle zuständig ist. Bei der Berechnung von k geht man davon aus, dass man für die Berechnung des mittleren absoluten Prädiktionsfehler $A[cx]/N[cx]$ genau k Bits zum Abspeichern braucht, denn wenn

$$2^k \geqslant A[cx]/N[cx]$$

oder

$$N[cx] \cdot 2^k \geqslant A[cx]$$

folgt daraus automatisch

$(N[cx] \ll k) \geqslant A[cx]$. [Str09]

Um diesen Wert k ganz einfach zu berechnen, lässt man eine for-Schleife (Alg. 2.6) solange laufen bis diese Bedingung erfüllt ist.

Algorithmus 2.6 Berechnung von k [SDR01]

$$for(k = 0; (N[cx] << k) < A[cx]; k++);$$

Nach diesen Vorbereitungen kann man mit der eigentlichen Codierung anfangen. Zuerst berechnet man den Rest indem man e_M um k Stellen nach rechts shiftet. Das entspricht dem Abschneiden der untersten k Bits. Dieser Rest wird dann unär ausgegeben, das heißt e_M Nullen gefolgt von einer Eins. Anschließend wird der k-stellige Quotient berechnet und an den davor berechneten unären Wert angehängt. Dies entspricht erwartungsgemäß der letzten k-Stellen von e_M.

Wie wir am Ende des Kapitels sehen werden, kann das Codewort im schlimmsten Fall sehr lang werden. Für eine Ausnahmebehandlung solch langer Codewörter wird der Golomb/Rice-Code mit limitierter Länge benutzt, für den folgende Variablen gebraucht werden:

$$qbpp = \lceil \log_2 (RANGE) \rceil \text{ (Bittiefe von RANGE)}$$

und

$$LIMIT = (2 \cdot (bpp + max(8, bpp) - qbpp - 1))$$

mit

$$bpp = max(2, \lceil log_2(MAXVAL + 1) \rceil)$$

wobei bpp die Anzahl der Bits, die für die Darstellung von $MAXVAL$ benötigt wird, angibt.

Nachdem wir den Error-Wert auf einem positiven Wert abgebildet haben, könnte e_M einen Wert von 255 aufweisen. Wenn k auch noch gleich 0 ist heißt es für uns, dass wir ein Codewort der Länge 256 haben. Dies entspricht der 32-fachen Länge des eigentlichen Wertes, was sehr unperformant ist. In diesem Fall wird der Restwert auf die Variable $LIMIT$ begrenzt und der eigentliche Wert - 1 mit einer Länge von $qbpp$ Bits binär übertragen. [MJW00]

3. Modellaufbau

In diesem Kapitel werden die zwei Phasen der Implementierung beschrieben. In der ersten Phase, Kapitel 3.1, wird der erste Versuch einer Implementierung vorgestellt, sowie die zum Schluss aufgetretenen Probleme. Da diese Implementierung sogar in der Simulation funktioniert hat, sollte diese nicht außer Acht gelassen werden. Außerdem sind die an dieser Stelle aufgetretenen Fehler wichtig um die Funktionsweise des System Generators zu verstehen. Kapitel 3.2 beschreibt dann die Lösung der aufgetretenen Probleme und endgültige synthetisierbare Implementierung.

Bild 3.1 zeigt die Implementierung des Algorithmus aus der zweiten Phase, die sich vom Aussehen kaum von dem ersten Versuch unterscheidet. Deswegen kann dieses Bild als Überblick der kompletten Arbeit genutzt werden.

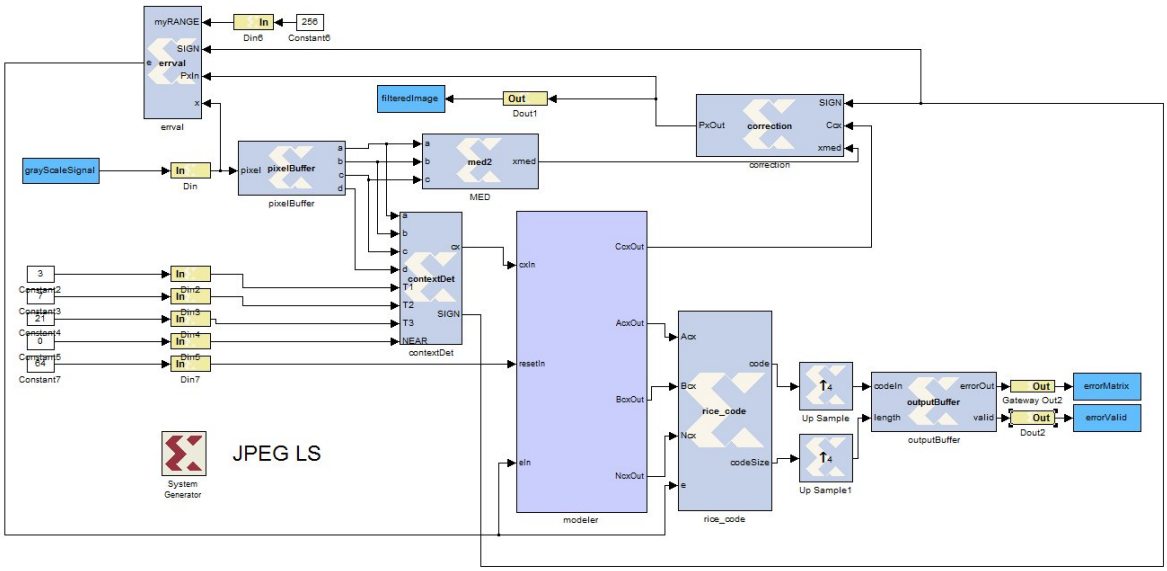


Abbildung 3.1.: Finale Implementierung des Algorithmus

Für eine bessere Übersicht befindet das Bild nochmal im Landscape-Modus im Kapitel Anhang (Bild A.1).

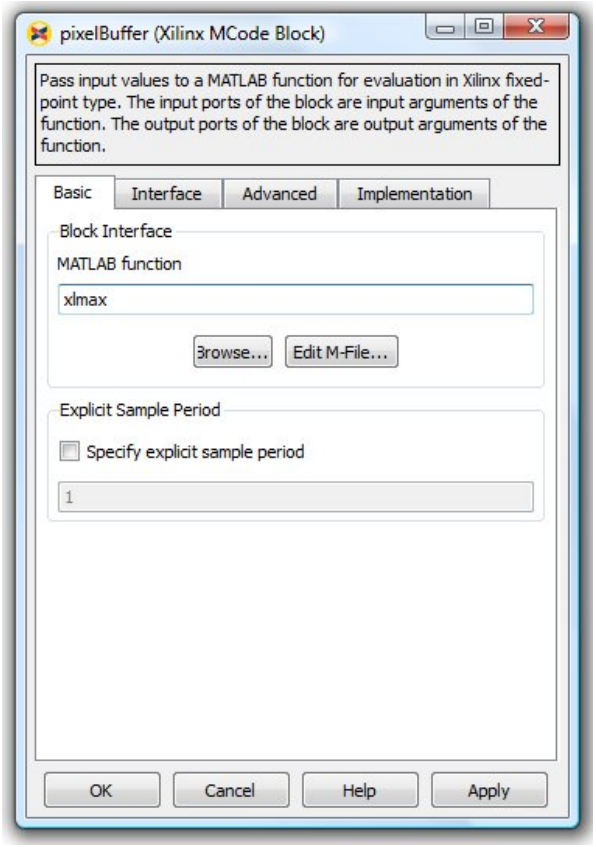


Abbildung 3.2.: Einstellungen MCode

3.1. Erster Implementierungsversuch

Wie im Kapitel 1 schon erwähnt werde ich versuchen den größten Teil des Algorithmus mit Hilfe von MCode-Blöcken zu implementieren. Der Vorteil dieser Blöcke liegt in der Einfachheit der Programmiersprache MATLAB, in der sich unser JPEG-LS-Algorithmus in kürzester Zeit schreiben lässt. Um so einen Block zu erstellen, wird zuerst eine Funktion in MATLAB geschrieben und in einer Datei mit der Endung .m abgespeichert. Diese Funktion beschreibt später die Funktionalität eines MCode-Blockes. Danach sucht man unter den Simulink-Libraries nach dem Xilinx-Blockset und zieht einen Xilinx-MCode-Block in ein Model, vorausgesetzt man hat MATLAB/Simulink installiert und für System Generator konfiguriert. Als Standard für jeden neuen MCode-Block befindet sich unter den Einstellungen die Funktion „xlmax“(siehe Bild 3.2). Diese kann dann mit der vorher von uns abgespeicherten Funktion ersetzt werden. Nach dem Drücken des Buttons OK unter Einstellungen ändern sich die Ein- und Ausgänge unseres Blockes, je nachdem wie die Parameter und die Rückgabewerte der Funktion definiert sind.

Z.B. folgender Funktionskopf

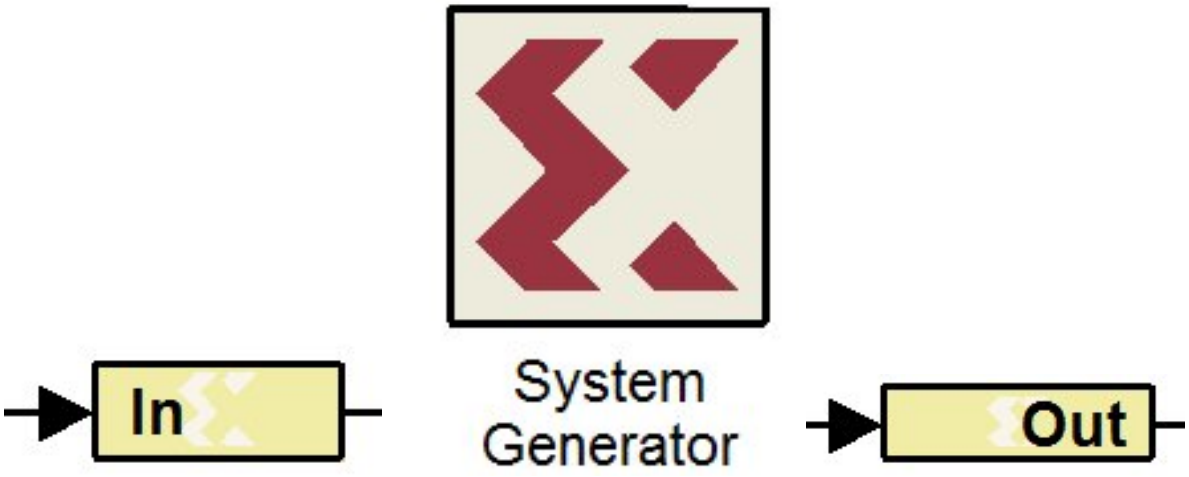
$function\ result = summe(a, b)$

definiert die Funktion „summe“ mit den Eingangsparametern a und b und mit dem Ergebnis $result$. Wenn man mehrere Ausgänge haben will, muss man alle Ausgänge, mit Komma getrennt und zwischen eckigen Klammern, aufzählen.

$function[result_1, ..., result_n] = myFunction(a, b, ...)$

Alle Ein- und Ausgänge verschiedener Blöcke werden genau wie in Simulink mit Linien verbunden. Diese entsprechen dann dem Datenfluss zwischen den Blöcken. Die Breiten der späteren Busse werden automatisch von System Generator auf den maximalen Wert der durch diesen Bus fließen könnte gesetzt. Man kann es also beeinflussen indem man die Werte, die durch diesen Bus transportiert werden sollen, auf eine bestimmte Größe castet (umwandelt). Die dafür verwendete Funktion xl_force, die vom Xilinx zur Verfügung gestellt wurde, wird in einem späteren Kapitel erklärt.

Um ein System-Generator-Modell zu bilden sind grundsätzlich zwei Aufgaben zu erledigen. Zuerst ist in jedem Modell und Teilmodell ein System-Generator-Block zu platzieren. [Xil10b] Dadurch erkennt Simulink später, dass es sich um ein System-Generator-Modell handelt und bestimmte Aktionen, wie z.B. Kompilieren und Synthetisieren, werden direkt von dem System-Generator-Tool übernommen. Desweiteren muss System Generator wissen wo ein System anfängt bzw. aufhört. Dies kann man mit Hilfe der Gateway In und Gateway Out Blöcke, die sich ebenfalls in der Xilinx-Blockset-Library befinden, definieren. Alles was sich vor dem Gateway In und nach dem Gateway Out befindet gehört nicht mehr zum Modell, kann aber dank der Co-Simulation-Funktionalität des System Generators, für die Simulation benutzt werden. Dies können z.B. Signale, bestimmte Hilfsfunktionen oder sogar Simulink-Blöcke sein. Sehr praktisch sind z.B. die To Workspace Blöcke oder die sogenannten Displays, die man in den Standard-Libraries von Simulink findet. Damit kann man die Werte als Variablen in den MATLAB-Workspace speichern oder gleich auf einem Display anzeigen lassen.



(a) Gateway In Block (b) System Generator Block (c) Gateway Out Block

Abbildung 3.3.: Wichtige Xilinx Blöcke

Für die Simulation des Modells wird die Simulink-Umgebung benutzt. In dieser Umgebung kann man ein System-Generator-Modell mit Standardblöcken aus Simulink verknüpfen. Dadurch lässt sich ein Modell vor der Synthetisierung ausführlich testen.

Eine weitere Hilfe für die Simulation sind die Modellfunktionen PreLoadFcn und StopFcn. Diese Funktionen werden vor bzw. nach der Ausführung einer Simulation aufgerufen. Mit Hilfe dieser MATLAB-Funktionen lassen sich z.B. verschiedene Variablen oder Signale, die für die Simulation notwendig sind, vorbereiten. In unserem Fall wird in der PreLoadFcn ein Testbild aus einer JPEG-Datei geladen und in ein für die Simulation passendes Signal umgewandelt. In der StopFcn wird unter anderem das Ausgangssignal des Modells in einer Variablen in dem MATLAB-Workspace gespeichert, wo man dann später, außerhalb des Modells, die Ergebnisse besser auswerten kann. Außerdem werden Eingangs- und Ausgangsbild in einem Fenster abgebildet um einen groben visuellen Vergleich zu haben (Bild 3.4). Mit diesem Vergleich konnte man schon im Anfangsstadium des Projekts anhand der Unterschiede zwischen den zwei Bildern grobe Fehler lokalisieren.

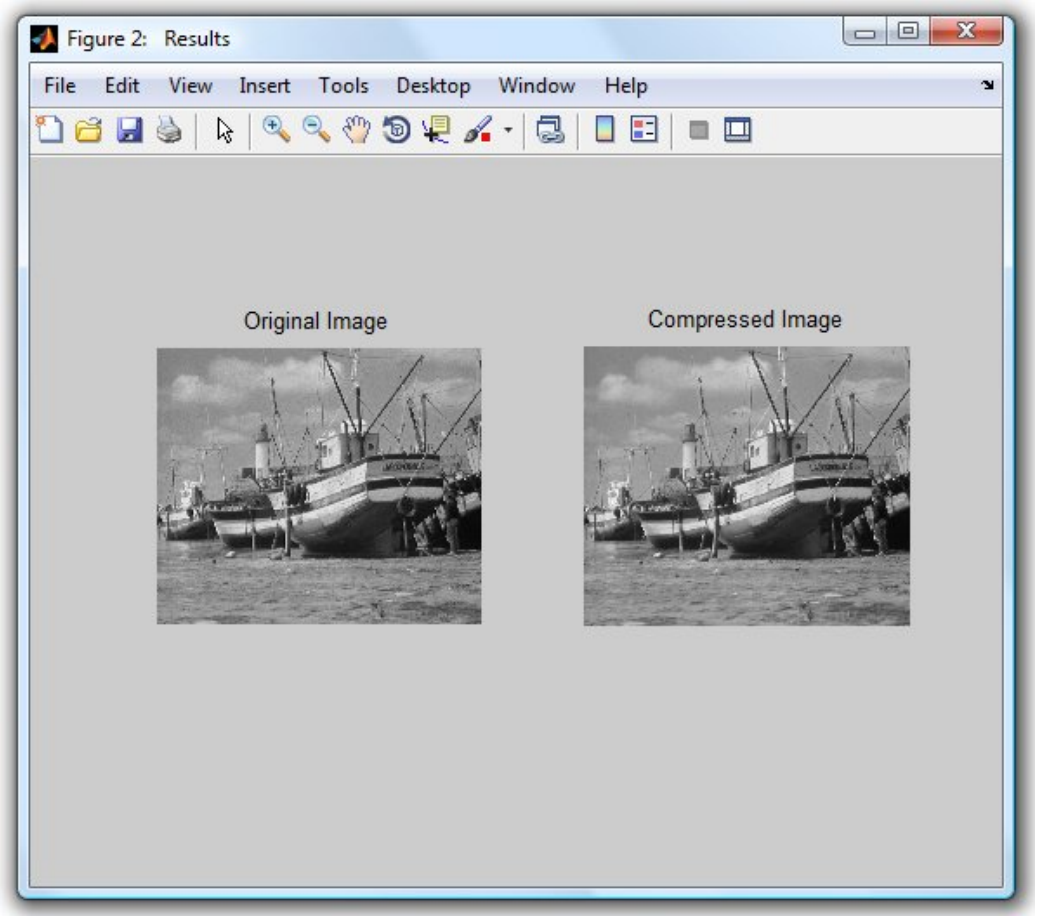


Abbildung 3.4.: Vergleich Originalbild mit dem Ergebnis

In den nächsten Kapiteln werden die für dieses Modell programmierten Blöcke einzeln erklärt und auf ihre Besonderheiten eingegangen.

3.1.1. InputBuffer

Der InputBuffer ist für das Puffern der eingehenden Pixel zuständig. Die wichtigste Komponente des InputBuffers ist eine persistente Variable. Mit Hilfe einer persistenten Variable lassen sich Variablen über mehrere Zyklen benutzen, ohne dass sie den Wert aus dem letzten Zyklus verlieren.

Mit der Funktion push_back_pop_front werden die Pixel in den Inputbuffer hinten eingefügt, und die vorderen Werte, die nicht mehr benötigt werden, rausgeschmissen. Die Länge des Buffers wird anhand der Breite des Bildes gewählt. Zu der Breite wird 1 hinzu addiert, so dass man genug Platz für alle nötigen Werte hat (siehe Bild 3.5). Der erste Platz im Buffer

entspricht dem Wert b, gefolgt von c und d und der letzte Platz wird von dem Wert a besetzt.

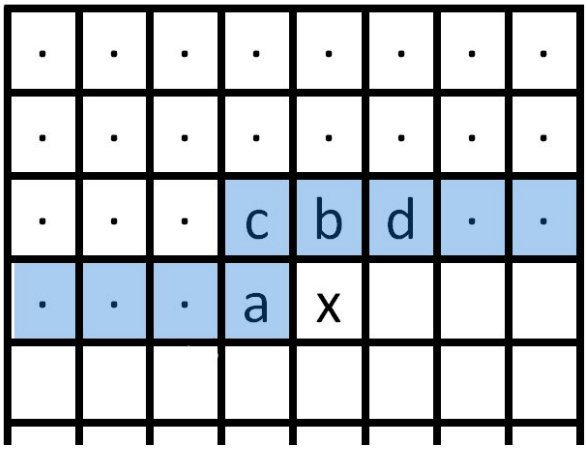


Abbildung 3.5.: Länge des InputBuffers

In unserem Fall handelt es sich bei der persistenten Variablen um ein Register, aus dem man die Werte über einen Index ansprechen kann. So kann man in der Funktion InputBuffer als Beispiel statt b buffer(0) benutzen.

Dieser Block wird also nur benötigt, um die schon gelesenen Pixel, die im späteren Algorithmus noch gebraucht werden, zu speichern und zum richtigen Zeitpunkt zur Verfügung zu stellen. Dafür werden die Werte an den oben genannten Stellen über die Ausgänge des Blockes an die med- und contextDet-Funktion weitergegeben (siehe Bild 3.1).

Das einzige vorauf man achten muss bei persistenten Variablen in einem MCode-Block ist, dass das Aktualisieren der Variablen erst am Ende der Funktion geschehen soll. Wenn man also nach der Aktualisierung der Variable noch einmal auf diese zugreifen will, bekommt man einen Fehler beim Kompilieren. Das Problem liegt darin, dass trotz der Zuweisung eines neuen Wertes, die Variable nicht sofort aktualisiert wird, sondern erst am Ende des Zyklus. Man würde also bei einer späteren Benutzung den alten Wert bekommen.

3.1.2. MED-Funktion

Die med-Funktion ist die vom aktuellen Kontext abhängige Prädiktion. Dafür benötigt man die Werte a, b und c, die direkt vom InputBuffer weitergeleitet werden. Wie in Kapitel 2.3 schon beschrieben, versuche ich die darin erklärte Formel zu implementieren.

Bei der Implementierung stellte sich heraus, das „min“und „max“zwei der Funktionen darstellen, die vom System Generator nicht unterstützt werden. Dies lässt sich aber ganz

einfach mit Hilfe von if-Statements umschreiben.

if $c \geqslant max(a, b)$ **then**
    ...
end if

kann man beispielsweise folgendermaßen umschreiben:

if $((c \geqslant a) \wedge (c \geqslant b))$ **then**
    ...
end if

da $c \geqslant max(a, b)$ nichts anderes bedeutet, als dass c mindestens genauso groß sein muss wie der größere der beiden Werte. Dies impliziert natürlich, dass c auch größer als der kleinere Wert sein muss. Den vorderen Teil der Funktion, der besagt, dass

$\hat{x} = min(a, b)$

kann man ebenfalls mit einem if-Statement implementieren:

if $a \leqslant b$ **then**
    $xmed = a$;
else
    $xmed = b$;
end if

Wenn $a \leqslant b$ ist, bedeutet das, dass a entweder der kleinere Wert oder gleich b ist. Wenn $a \leqslant b$ nicht zutrifft, dann ist b trivialerweise der kleinere Wert.

Die Implementierung von $\hat{x} = max(a, b)$ erfolgt analog zu der „min“-Funktion.

Bis auf diese kleinen Umschreibungen, lässt sich die Funktion eins zu eins in MATLAB kopieren (siehe Anhang A.1.2).

3.1.3. ContextDet

Im ContextDet-Block wird die Kontextnummer für den aktuellen eingehenden Pixel berechnet. Dabei werden die Werte a, b, c und d benutzt, die genauso wie für die MED-Funktion von dem InputBuffer zur Verfügung gestellt werden.

Für diese Funktion braucht man in erster Linie zwei Vektoren D und Q. Hier gibt es wieder einen kleinen Unterschied zu der Standard-MATLAB-Programmierung. Vektoren und Arrays müssen immer als persistente Variablen definiert werden. Wenn man das nicht macht bekommt man gleich bei dem ersten Versuch diese Variable anzusprechen, folgenden Fehler:

„assigning to an element of non-persistent variable is not allowed “.

Eine Alternative zu dieser persistenten Deklarierung ist einzelne Variablen für jede Vektorkomponente zu deklarieren. Also anstatt

persistent D, D = xl_state(zeros(1, 3), xlSigned, 10, 0);

könnte man drei Variablen $D1$, $D2$ und $D3$ deklarieren und damit im weiteren Verlauf der Funktion arbeiten. Allerdings würde dies zu einer sehr unübersichtlichen Funktion führen wie man gleich sehen wird.

Die Variante mit Vektoren wurde wie im folgenden Algorithmus 3.1 sehr schön übersichtlich programmiert:

Algorithmus 3.1 Quantisierung der Quotienten und Kontextbestimmung

```
1   for i=0:2
2       if (D(i) <= -T3)
3           Q(i) = -4;
4       elseif (D(i) <= -T2)
5           Q(i) = -3;
6       elseif (D(i) <= -T1)
7           Q(i) = -2;
8       elseif (D(i) < -NEAR)
9           Q(i) = -1;
10      elseif (D(i) <= NEAR)
11          Q(i) = 0;
12      elseif (D(i) < T1)
13          Q(i) = 1;
14      elseif (D(i) < T2)
15          Q(i) = 2;
16      elseif (D(i) < T3)
17          Q(i) = 3;
18      else
19         Q(i) = 4;
20      end
21  end
22
23  cxhelp = 9 * (9 * Q(0) + Q(1)) + Q(2);
```

Dabei sind $T1$, $T2$, $T3$ und $NEAR$ Variablen, die die Berechnung des JPEG-Algorithmus beeinflussen können und die als Inputs im Modell implementiert wurden. So lassen sie sich bei einer evtl. Ausführung auf einem FPGA von Außen über bestimmte Eingänge einstellen.

Algorithmus 3.2 zeigt wie die Schleife ohne Vektoren aussehen könnte.

Algorithmus 3.2 Quantisierung der Quotienten ohne Vektoren

```
1   for i=1:3
2       if i == 1
3           if (D1 <= -T3)
4               Q1 = -4;
5           elseif (D1 <= -T2)
6               Q1 = -3;
7           .
8           .
9           end
10      elseif i == 2
11          if (D2 <= -T3)
12              Q2 = -4;
13          elseif (D2 <= -T2)
14              Q2 = -3;
15                .
16                .
17          end
18      else
19          if (D3 <= -T3)
20              Q3 = -4;
21          elseif (D3 <= -T2)
22              Q3 = -3;
23                .
24                .
25          end
26                .
27                .
28      end
29  end
```

Wie man also leicht sehen kann, vergrößert sich die Länge dieser for-Schleife um mehr als das Dreifache. Die Schleife wird dadurch eigentlich unnötig, so dass man sie gleich weglassen könnte. Für die spätere Performance wäre wahrscheinlich die zweite Variante ohne Vektoren minimal besser als die erste, da man sich die Speicherung der persistenten Variablen spart.Da es sich um zwei sehr kleine Vektoren handelt, die kaum Ressourcen im Speicher beanspruchen, macht sich dieser Unterschied später nicht bemerkbar.

An dieser Stelle wird auch unsere Variable $SIGN$ gesetzt, je nachdem ob unsere Kontextnummer positiv oder negativ ausfällt. Durch Nutzung der Variable $SIGN$ können wir im weiteren Verlauf des Algorithmus mit dem Betrag der Kontextnummer rechnen und sparen

dadurch wieder an Busbreite beim Weiterleiten des Wertes.

Da uns die Funktion *abs* aus MATLAB, die den Absolut-Wert (den Betrag) liefert, auch nicht in System Generator zur Verfügung steht, müssen wir wieder ein if-Statement benutzen um abzufragen, ob der berechnete Wert negativ ist. Wenn dies der Fall ist,multipliziert man ihn mit -1. Da wir dieses if-Statement auch für die Setzung der Variable $SIGN$ benutzen können, ist an dieser Stelle die Nichtverfügbarkeit der Funktion „abs“nicht vom Nachteil.

Algorithmus 3.3 Absolutwert und Setzen von $SIGN$

```
1 if cx < 0
2   cx = -cx;
3   SIGN = -1;
4 else
5   SIGN = 1;
6 end
```

3.1.4. Modeler

Für die A, B, C und N-Arrays werden genauso wie bei dem InputBuffer persistente Variablen benutzt. Um mehrere Zugriffe auf diese Arrays zu vermeiden (z.B. mehrere Wertzuweisungen, die bei persistenten Variablen nicht erlaubt sind) werden am Anfang der Funktion die zu aktualisierenden Werte in lokalen Variablen gespeichert, mit denen weiter gerechnet. Am Ende der Funktion werden die lokalen Variablen wieder an die richtige Stelle im Array abgespeichert.

Vor diesem Block befinden sich zwei Delay-Blöcke, und zwar bei den Eingängen cx und $errVal$. Diese sind notwendig um einen Kreis im Modell zu vermeiden, einen sogenannten algebraischen Loop. Solche kombinatorischen Feedbacks sind im System Generator grundsätzlich nicht erlaubt, da die Periode im Modell nicht klar genug definiert werden kann. Dies war einer der größten Knackpunkte dieser Arbeit. An dieser Stelle differieren die Meinungen, welche die beste Lösung sei, sehr stark. Nach mehreren Wochen Recherche und sehr vielen Versuchen habe ich meiner Meinung nach eine optimale Lösung für dieses Problem gefunden, die ich im nächsten Abschnitt versuchen werde zu erklären.

Generell gibt es zwei Möglichkeiten so ein kombinatorisches Feedback zu lösen: Man fügt irgendwo innerhalb dieser Rückkopplung einen Delay- oder einen Memory-Block ein. Dadurch weißSystem Generator, dass an dieser Stelle eine Periode des Modells zu Ende ist. Da ein Speicher-Block mehr Ressourcen verbraucht als ein Delay und man sowieso nichts hat was man unbedingt speichern sollte, habe ich mich für die erste Lösung entschieden. Die Anfangsidee war, diesen Loop mit nur einem einzigen Delay-Block zu lösen. Dies gestaltet

sich für dieses Model sehr schwierig, wenn nicht sogar unmöglich. Das erste Problem dabei ist, dass eigentlich fast alle Blöcke in diesem Loop mehrere Ein- und Ausgänge haben. Das heißt, dass man mehrere Datenflüsse gleichzeitig mit einem Delay versehen muss damit die Datenströme für die verschiedenen Perioden nicht durcheinander kommen. Es gibt nur eine einzige Stelle in diesem Modell, wo man nur einen Datenstrom zwischen zwei Blöcken hat, und zwar zwischen dem errval-Block, der im nächsten Kapitel erklärt wird, und dem Modeler-Block, über den wir in diesem Kapitel sprechen. Man kann also dieses kombinatorische Feedback mit nur einem Delay unterbrechen, damit bringt man aber die Perioden durcheinander, so dass die Aktualisierung der A, B, C und N-Arrays nicht mehr korrekt durchgeführt werden kann. Das Problem dabei ist wenn man den Error-Value mit einem Delay bestückt, der Datenfluss dieses Wertes sozusagen einen Takt hinterherhinkt. Man bekommt also die Kontextnummer aus dem contextDet-Block für den aktuellen Pixel, man hat aber den Error-Value von dem vorherigen Pixel mit dem man die zu aktualisierenden Werte berechnet. Dies führt natürlich dazu, dass man die Werte an der falschen Stelle in den A, B, C und N-Array speichert. Meine Lösung dafür ist, dass man die Kontextnummer ebenfalls mit einem Delay bestückt um das ganze wieder zeitlich anzupassen. Nichtsdestotrotz benutze ich auch die aktuelle Kontextnummer um die aktuellen Werte aus den vier Arrays weitergeben zu können. Die alte Kontextnummer wird also am Anfang der modeler-Funktion für die Aktualisierung der Arrays benutzt. Dadurch dass die Aktualisierung am Anfang der Funktion geschieht, hat man beim Auslesen der Arrays im späteren Verlauf der Funktion immer den aktuellen Wert.

Ein weiteres Problem in diesem Block gab es beim Runden von Werten. Nach dem Shiften eines Wertes können Kommazahlen entstehen. Diese Kommazahlen wurden richtig auf- oder abgerundet, mit Ausnahme vom Zahlen, die eine fünf nach dem Komma hatten. Diese wurden einfach so unverändert gelassen. Dieses Phänomen wurde mit sehr viel Mühe, nachdem der ganze Algorithmus mehrmals im Debugger-Modus untersucht wurde. Leider war das die letzte Stelle an der gesucht wurde, da die Zahlen an der Stelle immer gerundet werden sollten. Um diesen Fehler zu beheben musste ein kleiner Trick verwenden werden: mit Hilfe der Funktion xl_binpt(x) kann man im System Generator die Position des Kommas bestimmen. Jedes Mal nach dem Shiften wurde das Ergebnis untersucht ob die Stelle des Kommas größer 0 ist. Wenn das der Fall war, dann hätte man ab oder aufrunden müssen, je nachdem ob der Wert positiv oder negativ ist. Da es keine Funktion zum Auf- oder Abrunden in System Generator gibt, vor allem weil dies automatisch geschehen sollte, musste man überlegen wie man dieses Erzwingen kann. Es wurde eine Konstante 0,5 eingeführt, die man dann einfach hinzuaddiert oder subtrahiert wenn das Ergebnis der obengenannten Funktion größer 0 ist. Damit erhält man die nächstnähere natürliche Zahl.

Da man in der Funktion mit lokalen Variablen arbeitet und erst am Ende der Funktion die Arrays aktualisiert, muss man beim Ausgeben fragen ob die aktuelle Kontextnummer gleich der alten Kontextnummer ist. Ist dies der Fall, wird nicht der Wert aus dem Array genommen, sondern gleich die Variable die mit Hilfe der alten Kontextnummer berechnet wurde. Mit diesem letzten Schritt ist auf jeden Fall gewährleistet, dass man immer den aktuellsten Wert übernimmt.

3.1.5. Correction

Im Correction-Block wird zuerst der Korrektionswert, der aus dem C-Array geholt wird, dem xmed hinzuaddiert oder abgezogen, je nachdem ob der Wert $SIGN$ positiv oder negativ ist. Da es sich bei dem Ergebnis eigentlich um Pixel handelt, muss man noch überprüfen ob sie sich nach der Korrektur noch in dem gültigen Bereich befinden. In unserem Fall müssen die Werte zwischen 0 und 255 liegen. Falls sie sich nicht in diesem Intervall befinden setzt man sie einfach auf eine dieser Intervallgrenzen, je nachdem auf welcher Seite des Intervalls sie sich befinden.

3.1.6. ErrVal

Bei der Berechnung des Errors wird einfach nur die Differenz zwischen dem aktuellen Pixel und dem berechneten Pixel aus dem Correction-Block genommen. Um den Fehler mit weniger Bit zu speichern mappt man als erstes diesen Wert auf einen Bereich zwischen -128 und 127. Dies geschieht analog zu dem in Kapitel 2.6 vorgestellten Algorithmus 2.4. Dieser Wert wird dann sowohl für den Modeler als auch für die Golomb/Rice-Codierung zur Verfügung gestellt.

3.1.7. GrEncoder

Dieser Block ist für die Codierung des Error-Wertes zuständig. Leider muss man bei diesem Block ein paar minimale Leistungseinbußen im Kauf nehmen wenn man das mit einem MCode-Block implementieren möchte. Ein Problem ist, dass man in einer for-Schleife keine Variable für die Anzahl der Schleifen benutzen darf. Das heißt man muss immer die maximale Anzahl der Schleifen durchgehen und mit einem if-Statement abfragen ob man die gewünschte Anzahl an Schleifen erreicht hat. Leider kann man in den for-Schleifen auch kein Break-Statement benutzen um die for-Schleifen zu beenden. Da es sich aber bei dem worst-case um keine großen Werte handelt und die for-Schleifen ohnehin nichts mehr machen wenn der gewünschte Wert erreicht ist, sind die Leistungseinbußen zu vernachlässigen.

Ein weiteres Problem in diesem Block ist, dass man nicht explizit mit binären Werten arbeiten kann. In System Generator gibt es nur drei Typen die zur Verfügung stehen: xl_Boolean, xl_Signed und xl_Unsigned. Das zwingt uns eine weitere for-Schleife zu programmieren, die den Quotienten Bit für Bit in dem Buffer schiebt.

Die Alternative zu dem MCode-Block wäre eine BlackBox. Durch einer BlackBox kann man die Codierung in VHDL schreiben. Mit Hilfe von Registern und Shift-Operationen lässt sich dieser Algorithmus sehr einfach implementieren. Da es sich aber in dieser Diplomarbeit darum handelt, herauszufinden ob man ein System ohne große Programmierkenntnisse aufzubauen kann, wurde die erste Variante implementiert. Dennoch werde ich ein paar

Worte zu der VHDL-Implementierung sagen. In VHDL gibt es die sogenannten Register. Mit deren Hilfe kann man ganz einfach einen Buffer implementieren und diesen auf eine sehr schöne und effiziente Art und Weise befüllen. Ein Register definiert man zum Beispiel folgendermaßen:

variable outputbuffer : std_logic_vector(38 downto 0) := (others => '0');

In diesem Fall handelt es sich um eine Variable namens „outputbuffer "vom Typ std_logic_vector.
(38 downto 0) heißt, dass dieses Register 39 Stellen hat, von 0 bis 38. (other => '0') initialisiert diesen Vektor komplett mit Nullen. Man könnte an dieser Stelle auch einen beliebigen String von Nullen und Einsen für die Initialisierung benutzen. Das schöne an VHDL und Registern ist z.B., dass man komplette Teilbereiche ansprechen kann und nicht nur einzelne Werte, wie in einem MCode-Block. In System Generator habe ich leider keine Möglichkeit gefunden wie man nur einen Bereich aus einem Array auslesen und woanders benutzen kann. Ich war gezwungen einzelne Stellen auszulesen und diese dann später wieder zusammenzufügen. Als Beispiel in VHDL würde ich gerne einen Ausschnitt aus dem limited Golomb/Rice Encoder zeigen. Bei den limited Golomb/Rice Encoder wird die Länge des Codes limitiert. Diese Limitierung ist bei uns bei einer Länge von 32. Wenn diese Länge überschritten wird, dann wird in unserem Fall 23 unär codiert (23 Bit + 1 Bit = 24 Bit) ausgegeben, gefolgt von dem eigentlichen Fehlerwert auf 8 Bit limitiert (24 Bit + 8 Bit = 32 Bit Gesammtlänge). Das Schreiben des unären Wertes in dem Buffer könnte folgendermaßen aussehen:

outputbuffer := outputbuffer(14 downto 0) & "00000000000000000000001";

Damit hat man die letzten 15 Stellen des Buffers, gefolgt von dem Wert 23 unär codiert, wofür wir 24 Stellen brauchen. Indem man nur die letzten Stellen aus dem alten Buffer übernimmt und diese nach vorne verschoben werden, verhält sich das ganze als würde man die Bits in dem hinteren Bereich des Buffers pushen.

Für genau die gleiche Stelle brauchen wir in dem MCode-Block eine for-Schleife von 0 bis 22, in der wir die oer in den Buffer schieben, gefolgt von einer einzelnen Operation, in der wir die letzte 1 hinzufügen. Das wird in MATLAB wie in dem folgenden Beispiel implementiert:

Algorithmus 3.4 Shiften im Buffer

```
1 for l = 0 : 22
2
3 outputBuffer.push_front_pop_back('0');
4
5 end
6
7 outputBuffer.push_front_pop_back('1');
```

Wie man sieht ist die MATLAB-Implementierung an der Stelle nicht so performant wie in VHDL. Genau dasselbe Problem hat man auch wenn man mehr als acht Bit im Buffer hat und man möchte diese ausgeben. In VHDL würde man z.B. folgende Implementierung benutzen:

result := outputbuffer(x downto (x - 7));

wobei x die Stelle ist bis wohin der Buffer gefüllt ist. In System Generator muss man dafür wieder eine for-Schleife von 0 bis 7 programmieren und die Bits einzeln aus dem outputbuffer zum Ergebnis hinzu kopieren.

Ein weiterer Vorteil von VHDL ist, dass man einen variablen Wert für die Endbedingung einer for-Schleife benutzen kann. Beispielhaft lässt sich das folgendermaßen realisieren:

```
1 for i in 0 to (19 - to_integer(unsigned(kIn))) loop
2         .
3         .
4 end loop
```

Dabei ist *kIn* eine beliebige Variable, die aber natürlich kleiner gleich 19 sein muss. Eine solche Implementierung ist in System Generator leider nicht erlaubt, obwohl das an vielen Stellen von Vorteil sein könnte.

Nichtsdestotrotz gibt es bei der Benutzung von VHDL auch ein gravierenden Nachteil. Anscheinend hat MATLAB bei der Benutzung von BlackBoxen ein Problem. Wenn man Modelle mit BlackBoxen benutzt stürzt MATLAB ohne Vorwarnung oder das Erscheinen einer Fehlermeldung einfach ab. Der erste Absturz trat bei mir nach circa einer Stunde auf. Nach dem ersten Programmcrash stürzte MATLAB in immer kleineren Abständen ab, bis es sich schließlich unmittelbar nach dem Start selbst beendete. Dies machte eine Arbeit mit den BlackBoxen fast unmöglich. Meiner Meinung nach und meinem Erfahrungen aus der

Java-Entwicklung folgend, sieht es so aus, als hätte man an dieser Stelle ein Problem mit der Garbage-Collection (Freigabe von nicht mehr benutzten Variablen). Der Java-Heap wird anscheinend überschritten, was zum Absturz von MATLAB führt. Leider konnte mir von der Mathworks Seite aus auch nicht weiter geholfen werden, da bei dem Absturz kein Crash-Dump erstellt wird. Dies könnte ein weiterer Hinweis auf Speicherüberlastung sein. Diese Problematik war für mich der endgültige Auslöser den Golomb/Rice-Coder mit MATLAB zu implementieren, vor allem weil ich mehrere Tage gebraucht habe, um herauszufinden, dass die Abstürze in Zusammenhang mit der BlackBox standen. Das Problem mit dem Absturz besteht übrigens bei allen von mir getesteten MATLAB-Versionen. Man könnte das evtl. mit anderen System-Generator-Versionen testen, die mir aber nicht zur Verfügung standen.

3.1.8. Up-Sample

Die Up-Sample-Blocks sind für die Erhöhung der Geschwindigkeit in einem System Generator Modell zuständig. Die Sample-Rate wird für alle folgenden Blöcke um das n-fache erhöht. Durch diese Frequenzerhöhung werden die Samples n-fach ausgewertet. Wenn man diese Erhöhung nur für einen Block oder nur einen Teil der folgenden Blöcke haben will, muss man die Frequenz mit Hilfe eines Down-Sample-Blocks wieder verringern.

Die Erhöhung der Sample-Rate ist für den Output-Buffer nötig, der mit einer viermal höheren Taktung als der Rest des Models laufen muss. Warum das sein muss wird im nächsten Kapitel erklärt.

In unserem Fall kommt nach dem OutputBuffer nichts mehr, das mit einer anderen Geschwindigkeit laufen muss. Allerdings müssen noch einige Parameter eingestellt werden um bei der Simulation die richtige Ausgabe zu bekommen. Wenn man die Variablen wie in unserem Modell mit Hilfe des To-Workspace-Blockes in der MATLAB-Umgebung speichern will, dann muss man diese Blöcke parametrisieren. Ein To-Workspace-Block stellt sich nicht automatisch auf die Frequenz des System-Generator-Modells ein. Das heißt speziell für unserem Fall, dass wir die Sample-Rate dieser Blöcke auf ein Viertel einstellen müssen (siehe Bild 3.6). Dadurch laufen auch diese Blöcke viermal schneller als normal und können alle unsere Ausgaben in dem MATLAB-Workspace herausschreiben.

3.1.9. OutputBuffer

Der Output-Buffer basiert auf der gleichen Technik wie der Input-Buffer. Man benutzt eine persistente Variable, in der die Codewörter geshiftet werden.

Die Codewörter, die in dem Output-Buffer hineingeschoben werden sollen, haben verschiedene Längen. Um die Ausgabe besser auszunutzen, habe ich mich entschieden den Code in 8-Bit-Einheiten auszugeben. Ein Codewort kann maximal 32 Bit lang sein. Im schlimmsten

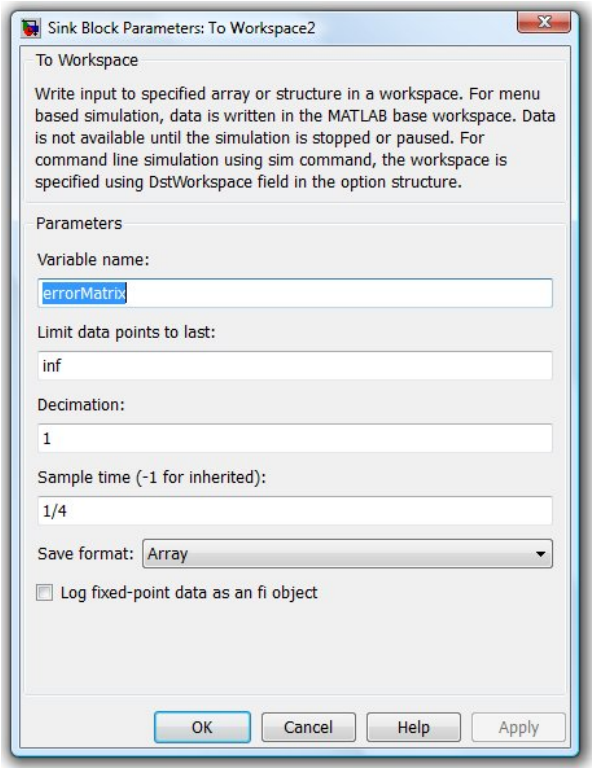


Abbildung 3.6.: Einstellungen To Workspace

Fall kann der Buffer schon bei Eingang eines 32-Bit-Codewortes bereits mit 7 Bit gefüllt sein, die noch nicht ausgegeben wurden, weil die 8 Bit noch nicht voll sind. Das heißt man braucht höchstens 39 Bit. Wenn man mehr als 32 Bit in dem Buffer hat muss man also 4x8 Bit ausgeben um den Buffer wieder zu leeren. Das ist der Grund warum der Output-Buffer viermal schneller als der Rest laufen muss.

Für den Fall, dass der Buffer nicht voll ist, muss man deutlich machen, dass die ausgegebenen Werte ungültig sind. Dies geschieht mit Hilfe eines weiteren Outputs „validOut“, der nur dann auf 1 gesetzt wird wenn es sich um eine gültige Ausgabe handelt. Sonst ist der Ausgang „validOut“immer auf 0 zu setzen. Falls es sich um eine ungültige Ausgabe handelt ist im Prinzip egal was man über den Output „errorOut“ausgibt, da man diesen Wert sowieso nicht betrachtet. Einfachheitshalber habe ich mich trotzdem dafür entschieden, nur eine 0 in diesem Fall auszugeben.

Bei der Ausgabe muss man wissen wie voll der Buffer zu diesem Zeitpunkt ist. Für diese Aufgabe wurde eine persistente Variable *counter* eingesetzt, die bei jedem Shiften in den Buffer sich um 1 erhöht und bei jeder Ausgabe sich um 8 verringert. Diese Variable wird auch dafür benutzt um zu überprüfen, ob man genug Bit im Buffer hat für eine gültige

Ausgabe. Eine Ausgabe geschieht also nur dann, wenn der Counter mindestens den Wert 8 hat.

3.1.10. Probleme

Nach der Implementierung des Algorithmus wie in den vorherigen Kapiteln beschrieben, tauchten leider ein paar Probleme auf. Trotz erfolgreicher Simulation des Modells ließ sich das nicht synthetisieren. Das größte Problem dabei waren die persistenten Variablen. Auch wenn die Implementierung laut User-Guide korrekt war, konnte keine erfolgreiche Synthetisierung durchgeführt werden. Da die Fehlermeldungen auch nicht sehr aufschlussreich waren dauerte die Suche nach dem Problem sehr viel länger als geplant.

Auf der Suche nach einer Lösung kam ich auf eine gänzlich andere Idee wie man das Ganze implementieren könnte. Die Idee basierte auf den Single-Port-RAM-Blöcken, die von Xilinx in der Standard-Library zur Verfügung gestellt werden. Mit Hilfe dieser Blöcke kann man dann sogar auf den Gebrauch von Registern verzichten, da man mit RAM-Blöcken auch ein kombinatorisches Feedback unterbrechen kann. Wie man diese RAM-Blöcke benutzt wird dann im Kapitel „Finale Implementierung“ erklärt.

Eine etwas ineffiziente Implementierung stellte auch der OutputBuffer dar. Das Problem an dieser Stelle war, dass man im System Generator keine Arrays oder Teile davon zwischen Variablen hin und her kopieren kann. Man kann nur eine Stelle eines Arrays ansprechen und kopieren. Dies führte dazu, dass ich die einzelnen Codewörter mit Hilfe einer for-Schleife Bit für Bit in den Buffer hineinschieben musste. Eine schönere Implementierung wird ebenfalls im nächsten Kapitel „Finale Implementierung“ vorgestellt.

3.2. Finale Implementierung

3.2.1. Problemanalyse

Die meisten Probleme beim ersten Implementierungsversuch gab es bei der Benutzung der persistenten Variablen. Deswegen sind die Änderungen, die in diesem Kapitel vorgestellt werden, fast nur in dem Modeler sowie in dem Output-Buffer, wo am meisten mit den persistenten Variablen gearbeitet wird. Ich werde versuchen, die Nutzung dieser Variablen soweit wie möglich zu reduzieren oder sogar komplett darauf zu verzichten, indem wir Alternativen suchen.

Auf der Suche nach alternativen Implementierungen habe ich versucht zu verstehen was überhaupt eine persistente Variable ist und was System Generator damit macht. Nach langer Suche und noch mehr Experimenten habe ich herausgefunden, dass System Generator aus einer persistenten Variable drei verschiedene Objekte synthetisieren kann. Je nachdem

wie man diese Variablen benutzt wird daraus ein Register, ein Single-Port-RAM oder ein ROM-Block erzeugt. [Xil10a] Dies ist im ersten Moment etwas unverständlich, wenn man sich aber die Funktionalität genauer anschaut ist das eigentlich logisch. Wenn man eine persistente Variable folgendermaßen benutzt

```
outputBuffer.push_front_pop_back('0');
```

wird daraus ein Register erzeugt. Mehr ist an dieser Stelle nicht nötig, da man die neuen Werte immer von einer Seite in das Register hineingeschoben werden.

Beim Auslesen einer solchen Variablen wird es schon komplizierter. Wenn man immer von der gleichen Stelle liest reicht ebenfalls ein Register, wenn man aber von verschieden Stellen die gespeicherten Werte nutzen will, wird daraus ein Single-Port-RAM erzeugt. Die Stelle, von der man lesen will, entspricht dann der Adressen des Speichers. Und das ist der Grund warum unsere Implementierung für den Modeler nicht funktionieren kann. Wenn aus unserer persistenten Variable ein Single-Port-RAM erzeugt wird, können wir in jedem Schritt nur von einer einzigen Adresse lesen bzw. Schreiben. Unser Encoder versucht aber im gleichen Schritt von einer Stelle zu lesen und an einer anderen Stelle zu schreiben. Das ist mit einem Single-Port-RAM nicht möglich. Das wäre höchstens nur mit einem Dual-Port-RAM zu realisieren.

Die dritte Variante wäre die Benutzung von ROMs, die aber FPGA spezifisch ist, da man vor der Implementierung schon wissen muss um welchen ROM es sich handelt.

Ich werde in diesem Kapitel versuchen die persistenten Variablen aus dem Modeler mit Dual-Port-RAM-Blöcke zu ersetzen und in dem Output-Buffer komplett wegzulassen.

3.2.2. Neuer outputBuffer

Trotz fehlerfreier Simulation ist die Funktion push_back_pop_front beim Synthetisieren nicht erlaubt, push_front_pop_back dagegen schon. Das heißt für uns, dass ich den Buffer einfach umdrehen muss um das Problem zu lösen. Wenn man die benötigten Stellen einfach um die Breite des Buffers spiegelt, ändert sich nichts an dem Ergebnis. Die Ergebnisse dieser einfachen Änderungen kann man im Anhang im Kapitel A.1.1 betrachten.

3.2.3. Neuer Modeler

Die Dual-Port-RAM-Blöcke lassen sich in unserem Fall auf ähnliche Weise implementieren wie die persistenten Variablen. Man muss von einer bestimmten Adresse lesen und an die vorherige Stelle den berechneten Wert schreiben. Für die Adresse können wir, genau wie in

dem alten Modeler, die Werte cx und $oldCx$ nehmen, die zwischen 1 und 367 liegen können. Diese Breite wird dann auch als Eigenschaft für den RAM-Block eingestellt. Eine weitere Eigenschaft, die für unseren Algorithmus sehr wichtig ist, heißt „Read before write“, diese muss für den schreibenden Port eingestellt werden. Das heißt, wie der Name schon sagt, dass von der eingestellten Adresse zuerst gelesen wird und dann mit einem neuen Wert aktualisiert wird. Dies ist wichtig um Kollisionen zu vermeiden.

Mögliche Kollisionen beim Dual-Port-RAM sind die write-write-Kollision und die write-read-Kollision. Eine write-write-Kollision kann in unserem Fall nicht auftreten, da wir den zweiten Port nur zum Schreiben benutzen. Eine write-read-Kollision tritt immer dann ein wenn beim Schreib-Port „Read after write“ eingestellt ist. Bild 3.7 zeigt die Unterschiede der zwei Optionen. Dabei ist $DOUTB_{ARF}$ der Ausgang mit der eingestellten Option „Read before write“ (read first) und $DOUTB_{AWF}$ für „Read after write“ (write first).

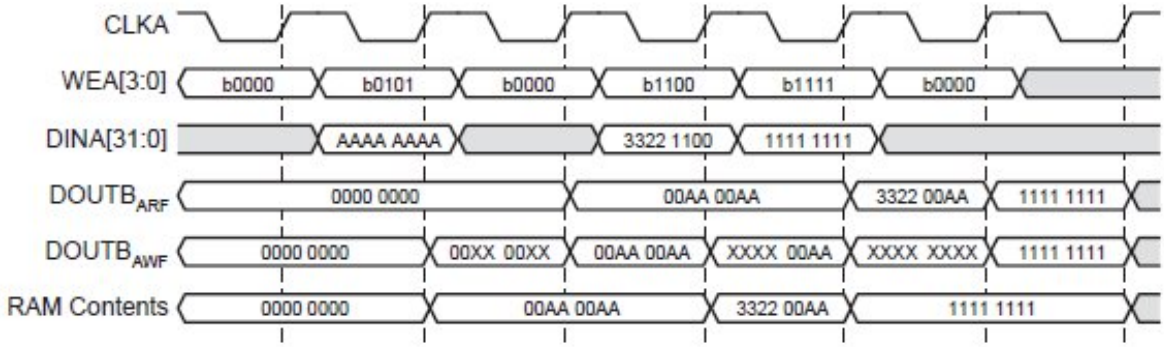


Abbildung 3.7.: Write-Read-Kollision [Xil11]

Man sieht, wenn man „Read after write“ benutzt bekommt man, wenn man an der gleichen Adresse lesen und schreiben will einen ungültigen Wert als Ausgang für den Lese-Port. Auf der anderen Seite bekommt man bei der Option „Read before write“ einen veralteten Wert wenn man die gleiche Adresse benutzt. Dies können wir vermeiden indem wir einen Bypass implementieren, der es uns bei Eintritt dieser Situation erlaubt den aktuellen Wert direkt von dem Eingang des Speichers nehmen. Die Entscheidung welchen Wert man nimmt ist also nur von der neuen und der alten Adresse abhängig und könnte auf zwei verschiedene Weisen implementiert werden. Die erste Möglichkeit ist einen MCode-Block mit einer einfachen if-Abfrage zu implementieren. Die Implementierung könnte wie im Alg. 3.5 aussehen.

Algorithmus 3.5 Wahl des richtigen Wertes aus dem Speicher

```
function [newValue] = selectNewValue(cx, oldCx, beforeRAM, afterRAM)

if (cx == oldCx)
    newValue = beforeRAM;
else
    newValue = afterRAM;
end

end
```

Eine alternative Implementierung wäre die Benutzung eines Komparators und eines Multiplexers. Der Komparator soll im Prinzip den neuen *cx* mit dem alten *cx* vergleichen und sein Ausgang soll mit dem Select-Eingang des Multiplexers verknüpft werden. Die anderen zwei Werte des Multiplexers sollen einmal mit dem Ausgang und einmal mit dem Eingang des RAM-Blockes verbunden werden (siehe Bild 3.8). Ich habe mich für die zweite Variante entschieden in der Hoffnung, dass der System Generator die Standardblöcke besser synthetisiert als die per Hand erstellten MCode-Blocks.

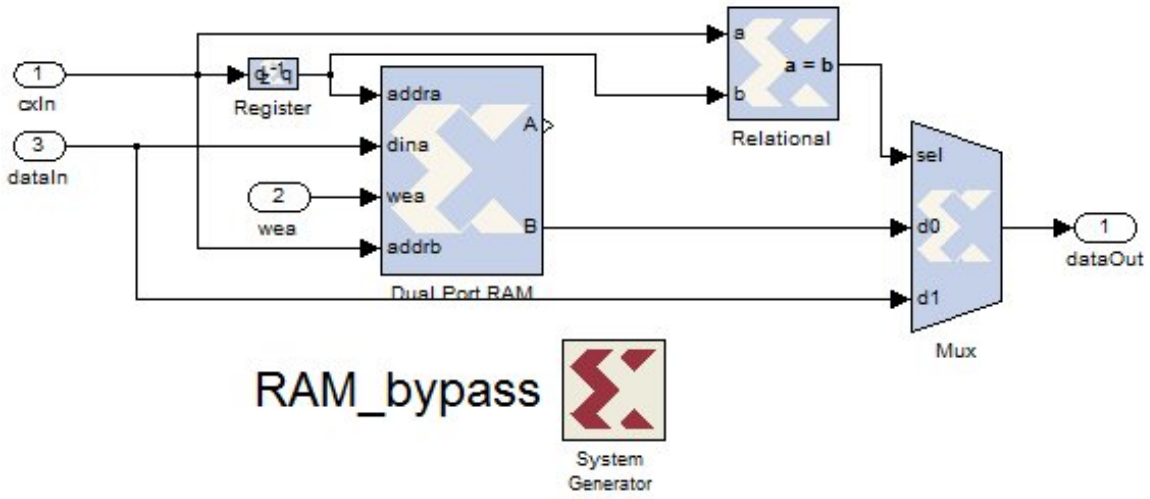


Abbildung 3.8.: RAM_ bypass Subsystem

Ein weiteres Problem haben wir bei einem echten Dual-Port-RAM mit der Einstellung der Latenz. Wenn wir die Latenz wie bei dem voreingestellten Wert auf 1 lassen, bekommen wir unsere Daten immer eine Periode später am Ausgang des Speichers. Das heißt für den

Modeler, dass er die Werte zu spät bekommt und dadurch die falschen Ergebnisse an den Speicher zurückliefert. Die Lösung dafür, die leider auch nirgends zu lesen ist, liefert uns die Option "Distributed Memory", die man unter den Einstellungen des Dual-Port-RAMs findet. Durch das Auswählen dieser Option transformieren wir den echten Dual-Port-RAM in einen einfachen Dual-Port-RAM. Ein einfacher Dual-Port-RAM wird dadurch gekennzeichnet, dass der zweite Port nur zum Lesen benutzt werden darf. Durch diese Umstellung darf man die Latenz auf 0 stellen und wir bekommen die Werte aus dem Speicher immer zum richtigen Zeitpunkt (siehe Bild 3.9).

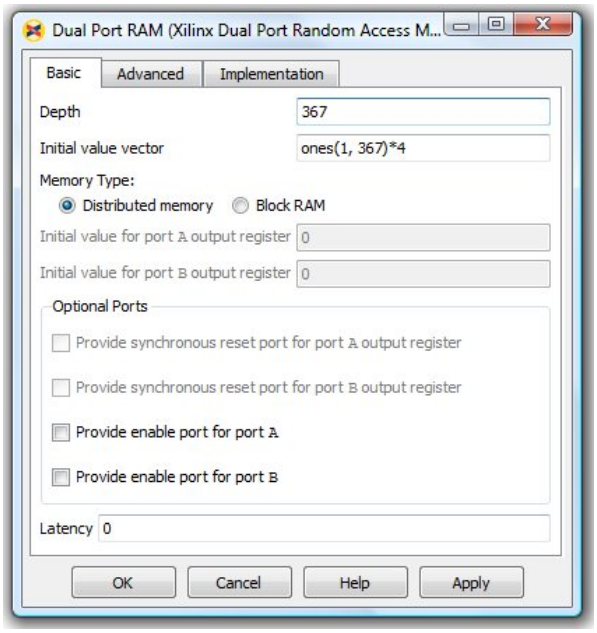


Abbildung 3.9.: RAM-Einstellungen

Um unser Modell übersichtlicher und kleiner zu halten werden wir an manchen Stellen Untersysteme bilden. Dadurch werden vor allem die Verbindungen zwischen Modeler und den Dual-Port-RAM-Blöcken etwas aufgeräumt und lassen sich besser verfolgen. Die Erzeugung eines Subsystems ist sehr einfach. Man markiert alle Komponenten, die in das Subsystem verschoben werden sollen, und dann wählt man mit Hilfe der rechten Maustaste die Option „build Subsystem“. Aus den ausgewählten Komponenten entsteht nun nur noch ein einziger Block mit den nötigen Ein- und Ausgängen. Der neue Modeler, als Subsystem implementiert, ist im Bild 3.10 zu sehen. Die RAM_ bypass-Blöcke entsprechen dem Subsystem aus dem Bild 3.8.

Bei dieser neuen Implementierung müssen wir vor allem unseren Modeler-Block etwas verändern. Da die Speicherung von $A[cx]$, $B[cx]$, $C[cx]$ und $N[cx]$ jetzt außerhalb des Blockes geschieht, muss man diese Werte zuerst über Eingänge in diesen Block hineinbekommen.

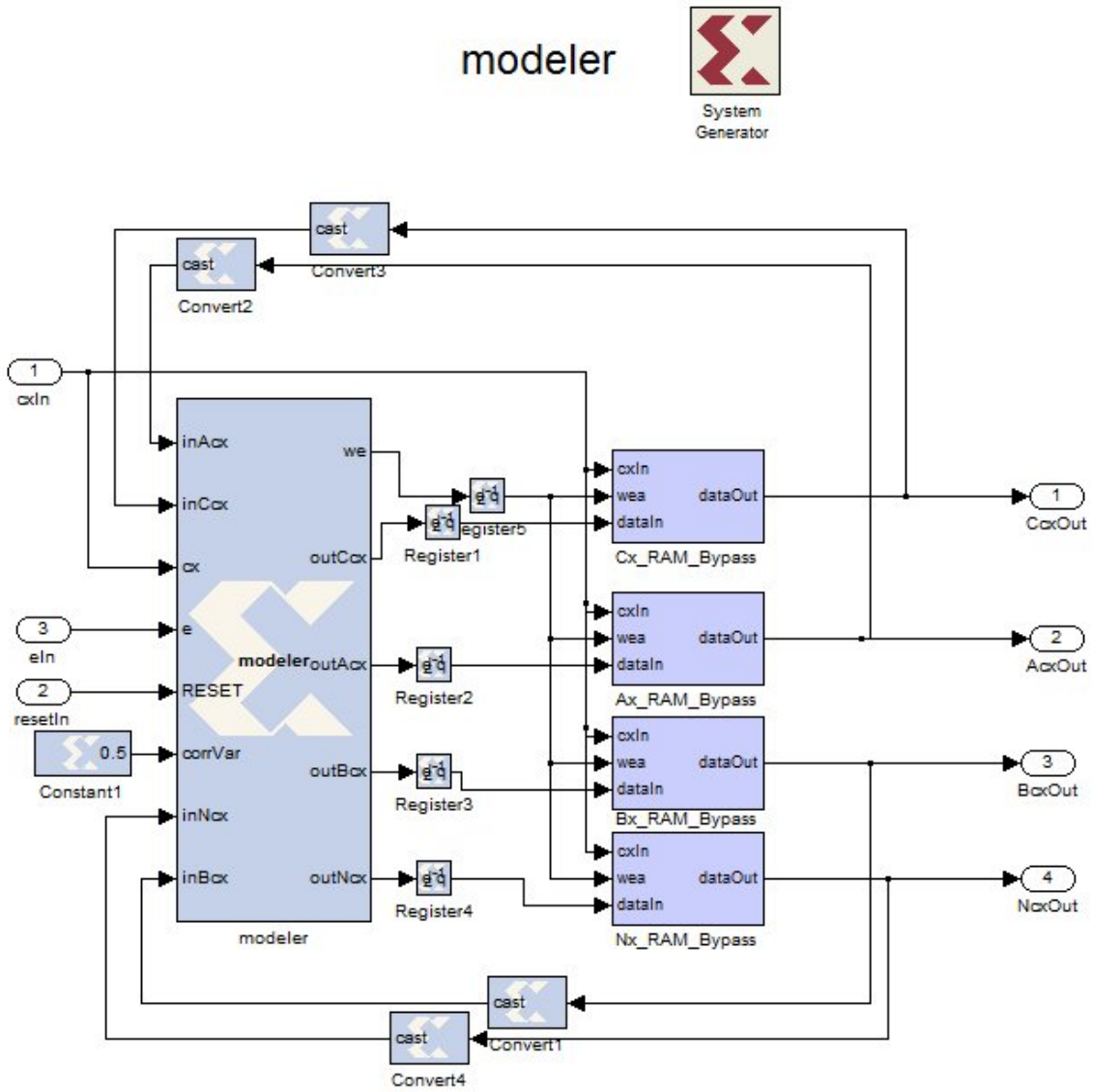


Abbildung 3.10.: Modeler Subsystem

Dafür werden zuerst die Eingänge *acxIn*, *bcxIn*, *ccxIn* und *ncxIn* definiert, die dann mit den Eingängen der RAM-Blöcke verbunden werden. Die aktualisierten Werte werden dann über die Ausgänge des Blockes zu den Eingängen der RAM-Blöcke geleitet. Die cx Werte werden innerhalb des Modeler-Blocks gar nicht mehr benötigt, so dass dieser Eingang ganz wegfällt. Dafür wird cx jetzt für alle der vier Speicherblöcke als Input für die Adresse benötigt. Da das Schreiben in einem RAM-Block nur einmal pro Periode geschehen soll, sind Konflikte beim Schreiben ausgeschlossen und wir können den Eingang „we“(Write enable) immer mit dem Wert 1 belegen. Dadurch ist ein Schreiben immer erlaubt und wir müssen uns keine Gedanken über einer besonderen Taktung machen.

Wie in Kapitel 3.1.10 schon erwähnt, unterbrechen auch die RAM-Blöcke ein kombinatorisches Feedback, so dass unser Register zwischen dem errVal-Block und den Modeler-Block wegfallen kann. Durch diese Änderung wird auch das zweite Register nach dem contextDet-Block unnötig und kann somit auch entfernt werden.

Da man nur noch mit einer Stelle der vier Arrays arbeitet, fallen auch ein paar der temporären Variablen weg, die früher für die internen Berechnungen innerhalb des Blockes benutzt wurden.

Man hat nun vier kleine Schleifen zwischen den Modeler und den Speicherblöcken. Da weder in dem Modeler-Block noch in dem RAM-Block definiert werden kann um was für einen Typ es sich bei den berechneten Werten handelt, muss man diese Werte mit Hilfe von cast-Blöcken konvertieren. Die cast-Blöcke werden dann zwischen dem Speicherausgang und dem Modelerausgang eingefügt, da dort die Werte das erste Mal von einem Block zum anderen übergeben werden. Alternativ dazu könnte man innerhalb des Modeler-Blocks den Typ dieser Variablen mit Hilfe der Funktion „xfix“festlegen. Dieser Variablentyp wird dann automatisch an die internen Variablen des Modelers übertragen und von dort aus wieder in den Speicher. Dadurch wird sichergestellt, dass überall der gleiche Datentyp benutzt wird und es können keine Konflikte entstehen.

3.2.4. Neuer outputBuffer

Eine neue Implementierung des OutputBuffers basiert auf Shiften und der die Or-Funktion. Die erste große Änderung ist der Typ der Variable die für den Buffer benutzt wird. Anstatt einen Vektor als persistente Variable zu benutzen, nehmen wir jetzt eine Variable vom Typ xlUnsigned mit einer Breite von 39 Bit. Der Vorteil dieser Variable ist, dass man jetzt im Gegensatz zum Vektor, den Wert shiften kann. Dadurch lässt sich das Shiften auf eine einfachere Weise implementieren.

Man bekommt ein Codewort und dessen Länge. Jetzt kann man die neue xlUnsigned-Variable um diese Länge nach links shiften um Platz für das neue Codewort zu schaffen. Nach dem Shiften hat man im Prinzip eine Reihe von Nullen am Ende des Buffers erzeugt,

deren Anzahl der Länge des neuen Codewortes entspricht. Mit Hilfe der Or-Operation zwischen dem Buffer und dem Codewort kopiert man das Codewort an das Ende des Buffers. Dadurch fällt die for-Schleife weg, da man das ganze Codewort in einem einzigen Schritt in den Buffer kopieren kann. Dadurch spart man sich auch die Probleme die man mit den for-Schleifen im System Generator hat.

Das Shiften funktioniert allerdings auch nicht so einfach wie man es in MATLAB gewöhnt ist. Das Problem an dieser Stelle sind die Funktionen xl_ rsh und xl_ lsh, die in System Generator für das Shiften nach rechts bzw. nach links verantwortlich sind. Diese zwei Funktionen erlauben keine Variablen als Parameter, was für uns bedeutet, dass wir nicht direkt um die Länge des Codewortes shiften können. Eine einfach Lösung, die zwar nicht schön aussieht, aber dafür perfekt funktioniert, ist die Benutzung von if-Anweisungen. Man fragt alle möglichen Längen ab und wenn man die Richtige gefunden hat, wird um diesen Wert geshiftet. Dadurch benutzt man die xl_ rsh-Funktionen nur noch mit konstanten Werten. Diese Aneinanderreihung von if-Anweisungen entspricht eigentlich einer case-Funktion, diese wird allerdings in einem MCode-Block nicht unterstützt.

Für die Ausgabe des 8-Bit-Strings aus dem Buffer müssen wir die ältesten acht Bits aus dem Buffer ausgeben. Die Stelle von der wir ausgeben müssen hängt natürlich von der bisherigen Befüllung des Buffers ab, sie variiert also vom Schritt zu Schritt. Da die Funktion xl_ slice, mit der wir einen String abschneiden können auch keine variablen Parameter unterstützt, benutzen wir den gleichen Trick wie bei dem Schieben in den Buffer. Wir kopieren zuerst unsere Variable „buffer“in eine temporäre Variable, die dann um die richtige Anzahl an Stellen nach links geshiftet wird, so dass sich unser benötigter String ganz am Anfang befindet. Nun können wir mit Hilfe der Funktion xl_ slice die obersten acht Bit abschneiden und ausgeben.

3.3. Tests und Überprüfungen

Um sicher zu gehen, dass wir eine korrekte Implementierung haben, mußte ich alle Ergebnisse und Teilergebnisse überprüfen. In diesem Kapitel werden einige der zahlreichen Möglichkeiten geschildert, die meiner Meinung nach am einfachsten sind.

Ein sehr wichtiger Test, den ich von Beginn an in diesem Modell eingebaut habe, ist die Korrektheit des Prädiktionswertes. Diese Werte werden in meinem Modell über einen To-Workspace-Block in den MATLAB-Workspace herausgeschrieben und nach der Simulation mit Hilfe der Funktion JPEG-LS_StopFcn in eine Matrix umgewandelt. Diese Matrix wird dann als bmp-Datei auf der Festplatte gespeichert. Dieses Bild wird dann mit einem 100%ig korrektem Bild, das mir von meinem Betreuer zu Verfügung gestellt wurde, verglichen. Wenn man bei den zwei verglichenen Bildern keinen Fehler mehr entdeckt kann man sich sicher sein, dass der Algorithmus richtig funktioniert. Ist das nicht der Fall, muss

man versuchen den Fehler in den einzelnen Blöcken zu lokalisieren.

Um die einzelnen Blöcke zu überpfüfen wurde zuerst der ganze Algorithmus in MATLAB geschrieben. Dabei wurde die gleiche Strukturierung wie in dem System-Generator-Modell benutzt um die späteren Vergleiche zu erleichtern. Die Funktionen der einzelnen MCode-Blöcke wurden also einfach hintereinander geschrieben um eine einzelne MATLAB-Funktion zu erzeugen. Um sicher zu gehen, dass diese Funktion korrekt ist, wurden die Ergebnisse dieser MATLAB-Funktion erstmal von meinem Betreuer anhand eines von ihm implementierten JPEG-Algorithmus untersucht. Erst nachdem sichergestellt wurde, dass meine MATLAB-Funktion die richtigen Ergebnisse liefert, durfte ich sie als Referenz für das System-Generator-Modell benutzen.

Um jetzt einzelne Werte aus dem Modell zu untersuchen, musste ich diese zuerst in den MATLAB-Workspace herausschreiben. Falls diese nicht außerhalb der Blöcke sichtbar waren, wurden zuerst neue Outputs für die jeweiligen Blöcke erzeugt. Das Schreiben in dem Workspace geschieht, genau wie bei allen anderen Ausgaben des Algorithmus, über „Gateway Out“-Blöcke, gefolgt von einem „To Workspace“-Block. Die gleichen Variablen schreiben wir auch aus unserer MATLAB-Funktion unter einem anderen Namen raus. Nun haben wir zwei Variablen, die den gleichen Inhalt haben sollten.

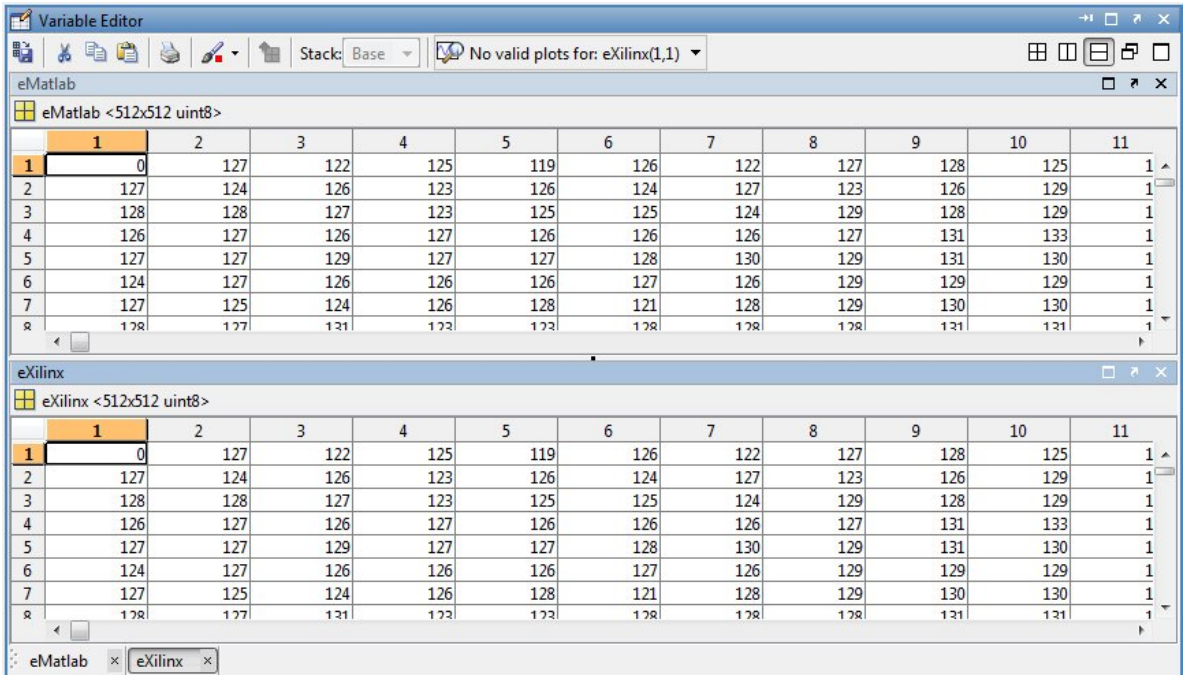


Abbildung 3.11.: Vergleich der Variablen

Angenommen unsere MATLAB-Variable heißt *eMatlab* und die aus dem System-Generator-Modell *eXilinx*. Man könnte nun beide Variablen jetzt mit Hilfe des „Variablen Editor “in MATLAB öffnen und jede Stelle einzeln überpfüfen ob sie gleich sind (siehe Bild 3.11). Dies wäre bei kleine Ausgaben kein Problem, bei sehr langen Simulationen aber viel zu aufwändig. Deswegen benutzten wir für den Vergleich die MATLAB-Funktion „find“. Mit dem Aufruf

e = find(eMatlab ∼= eXilinx);

schreibt man in einer neuen Variable *e* alle Stellen an denen sich die Variablen *eMatlab* und *eXilinx* unterscheiden. Ist *e* nach diesem Aufruf leer, dann sind unsere zwei Variablen identisch und unser Ergebnis aus System Generator richtig. Dadurch dass in der Variable *e* genau die Stellen geschrieben werden, wo sich die Werte unterscheiden, ist es etwas einfacher die Probleme zu lokalisieren.

Auf diese Art wurden alle Ergebnisse dieser Arbeit kontrolliert, so dass man sich sicher sein kann, dass der Algorithmus richtig implementiert wurde. Die MATLAB-Funktion, die während der gesamten Arbeit zum Testen genommen wurde, befindet sich als Anhang im Kapitel A.1.11.

4. Synthetisierung

Um die Vollständigkeit und Korrektheit der Implementierung zu überprüfen wird das System-Generator-Modell synthetisiert. In diesem Kapitel wird der Synthetisierungsvorgang beschrieben und seine Ergebnisse präsentiert.

Um das Modell synthetisieren zu können, muss es zuerst in VHDL umgewandelt werden. Dies geschieht mit Hilfe des System-Generator-Blocks (siehe Bild 3.3 b). Durch einen Doppelklick auf diesen Block öffnet sich das im Bild 4.1 gezeigte Fenster mit den Kompiliereinstellungen. Hier kann man verschiedene Optionen auswählen, wie z.B. ob man eine HDL-Netzliste oder einen Bitstream haben will, den Typ des FPGAs oder die Beschreibungssprache der Hardware (VHDL oder Verilog).

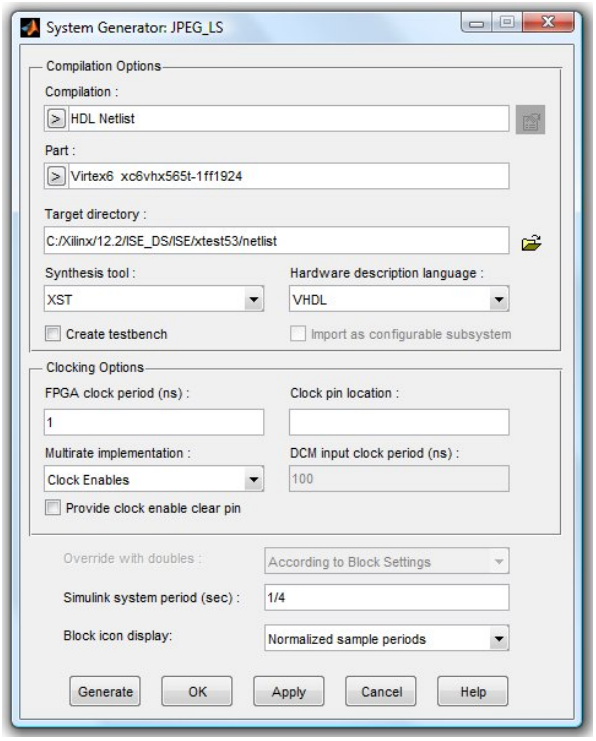


Abbildung 4.1.: Einstellungen System-Generator-Block

Durch das Drücken auf den Knopf „Generate“ wird nicht nur das ganze Modell in VHDL umgewandelt sondern auch ein Xilinx ISE Projekt erstellt. Mit Hilfe des Xilinx-ISE-Tools lassen sich dann nicht nur die Ergebnisse betrachten, sondern es lassen sich auch neue Synthetisierungen mit verschiedenen Parametern durchführen.

Die folgenden Ergebnisse wurden nicht mit den optimalen Einstellungen erzeugt, da dies nicht Teil der Diplomarbeit ist. Sie sollen lediglich zeigen, dass unser Modell synthetisierbar ist und um einen ersten Eindruck zu ermöglichen. Die nächsten Tabellen zeigen die wichtigsten Reportwerte der Synthetisierung. Als Target Device wurde ein Virtex-5 FPGA (XC5VSX50T-1FF1136) ausgewählt.

Slice Logic Utilization	Used	Available	Utilization
Number of Slice Registers	4.202	708.480	1%
Number of Slice LUTs	3.696	345.240	1%
Number used as logic	1.203	345.240	1%
Number used as Memory	592	101.920	1%
Number used exclusively as route-thrus	1.901		
Number of bonded IOBs	80	640	12%

Timing Summary	Value
Minimum period	39,829ns (Maximum Frequency: 25,107 MHz)
Minimum input arrival time before clock	38,573ns
Maximum output required time after clock	42,005ns
Maximum combinational path delay	40,749ns

Durch bessere Einstellung der Synthetisierung lassen sich mich hoher Wahrscheinlichkeit bessere Ergebnisse erzielen, die aber hier nicht untersucht wurden. Andere Verbesserungsmöglichkeiten werden im Kapitel Fazit und Ausblick vorgestellt.

5. Fazit und Ausblick

5.1. Fazit

Generell sind Implementierungen mit Hilfe von MCode-Blöcken möglich, oft ist aber sehr viel Vorstellungskraft gefragt. Das liegt daran, dass die Anzahl der MATLAB-Funktionen die man im MCode-Block benutzen darf, sehr klein gehalten ist oder sogar komplett durch eigene, von Xilinx selbst geschriebenen Funktionen, ersetzt wurden. Man muss also viele Sachen, die im MATLAB selbstverständlich sind, auf eine oft ungewohnte Art und Weise implementieren. Dies führt meistens zu sehr unverständlichen Algorithmen und kann evtl. auch zu Performanceeinbußen führen. An dieser Stelle ist der User-Guide von System Generator eine kleine Hilfe, in dem ein paar der wichtigsten Funktionen, die in Verbindung mit einem MCode-Block zur Verfügung stehen, erklärt werden. Vor allem die Benutzung von persistenten Variablen wird anhand von mehreren Beispielen etwas deutlicher. Leider ist dieser User-Guide sehr knapp gehalten, so dass man an vielen Stellen nur mit Ausprobieren und Experimentieren weiterkommt. Sachen, die evtl. selbstverständlich erscheinen, haben bei der Implementierung doch sehr viele Restriktionen, die aber nirgendwo dokumentiert sind. Andere Probleme kommen erst bei der Synthetisierung zum Vorschein, funktionieren aber bei der Simulation perfekt. Das war auch bei dieser Arbeit der Fall, so dass man erst zum Schluss bemerkt hat, dass die Implementierung doch nicht funktioniert wie gedacht.

Die Debugger-Option der MCode-Blöcke ist bei der Implementierung eine willkommene Funktion. Man kann damit theoretisch in jedem Block die Variablen und Funktionen überprüfen. Die Simulation wird dadurch sehr stark verlangsamt (in meiner Entwicklungsumgebung um das 250-fache), was vor allem bei Algorithmen mit großen Datenmengen von Nachteil ist. Der Debugger-Modus funktioniert leider auch nicht immer wie gewünscht. Bei manchen Blöcken kommt es zu unerklärlichen Abstürzen wenn man sie debuggen will. Die Fehlermeldung „Fatal Error“währen der Kompilierung hilft leider auch nicht weiter. Dies lässt sich aber umgehen indem man die zu überprüfenden Werte über einen Output in den Workspace von MATLAB herausschreibt und erst nach der Simulation untersucht.

Nichtsdestotrotz lassen sich auf diese Art auch schwierigere Algorithmen implementieren, und spätestens beim zweiten Projekt kennt man wahrscheinlich auch die Nachteile des System Generators etwas besser und weißt wie man sie umgehen kann. Da das Arbeiten mit Vektoren öfters in dieser Arbeit ein Problem war, vor allem wenn man die Werte aus diesen Vektoren an mehreren Stellen braucht, ist es vielleicht einfacher den Algorithmus nicht in zu viele Blöcken zu teilen. Dadurch kann man evtl. sehr viele Probleme vermeiden und die

Implementierung wird um Einiges einfacher. Auch für meine Arbeit wäre das an manchen Stellen von Vorteil gewesen. Ich habe mich aber trotzdem für eine feinere Unterteilung des Algorithmus entschieden, um dem Leser einen besseren Überblick zu verschaffen und um die Nähe zum Schema des JPEG-LS-Algorithmus (Abbildung 2.1) beizubehalten. Durch die Parallelen zu diesem Schema kann man die Implementierung in dieser Arbeit besser verfolgen und man entwickelt einen besseres Verständnis für die hier benutzten Tools.

Ein weiterer Vorteil von System Generator ist auch das Finden von Bugs. Mit Hilfe der Simulink-Umgebung lassen sich die System-Generator-Modelle mit den Simulink-Blöcken verbinden. Mit Hilfe dieser Simulink-Blöcke lässt sich dann das ganze Projekt simulieren und die Korrektheit des Modells überprüfen. So kann man gleich während der Implementierung Bugs finden und diese umgehend korrigieren, bevor man überhaupt mit der Hardware in Verbindung kommt. So verkürzt sich die Entwicklungszeit deutlich, da vor allem die Testphase und die damit verbundenen Folgeschritte deutlich reduziert werden und im besten Fall sogar ausfallen könnten.

Der Verlauf dieser Arbeit zeigte, dass diese Art der Implementierung ist vor allem für Softwareentwickler geeignet, die geringe Kenntnisse von Hardware haben, dafür aber in der Entwicklung von Algorithmen ihre Stärken aufweisen. Auch Algorithmen, die aus Sicht der Hardware-Seite sehr schwer vorstellbar sind, ließen sich mit System Generator einfacher implementieren. Algorithmen, die als Software zur Verfügung stehen, ließen sich auf dieser Art ebenfalls einfacher in ein System-Generator-Modell umsetzen.

Auf der anderen Seite ist es unter Umständen doch einfacher einen Algorithmus, oder nur Teile davon, mit VHDL zu implementieren. Das beste Beispiel dafür wäre der Golomb/Rice-Encoder in dieser Arbeit. In VHDL ließ sich dieser Encoder sehr einfach und schön programmieren, mit System Generator gab es doch an mehreren Stellen Schwierigkeiten und der Code wurde zum Schluss zunehmend unübersichtlich.

Am besten man verbindet diese zwei Möglichkeiten, also eine Mischung aus MATLAB- und VHDL-Code, was mit System Generator theoretisch kein Problem ist. Wie das funktioniert wurde schon im Kapitel 3.1.7 erklärt. Dadurch ist es möglich sich die Vorteile aus beiden Entwicklungsvarianten herauszupicken und so ein besseres Modell zu erstellen.

5.2. Ausblick

Diese Arbeit könnte an manchen Stellen vertieft und erweitert werden. Eine kleine, aber auch nützliche Erweiterung wäre zum Beispiel die Implementierung des Lauflängenmodus, der hier nicht Teil der Arbeit war. Teile für diesen Lauflängenmodus sind sogar schon vorhanden. Man hat die Gradienten D_1, D_2 und D_3, die man für diesen Modus abfragen muss ob sie gleich 0 sind. Die nötigen Kontexte [365] und [366] sind auch schon implementiert, werden

allerdings im regulären Modus nicht benutzt. Man müßte sich also nur um die Bestimmung der Lauflänge und um die Segmentierung kümmern. Mit Hilfe dieser Erweiterung wäre der JPEG-LS-Algorithmus komplett.

Eine effizientere Implementierung, wurde schon in Kap. 5.1 erklärt. Man könnte manche Blöcke zusammenführen um Variablenübergaben zu vermeiden. Wenn man mit lokalen Variablen arbeiten würde, müßte man nicht mehr so genau auf den Typ der Variablen und die dafür benötigte Bußbreite achten. An dieser Stelle funktioniert System Generator wirklich gut und konvertiert die Variablen immer in den an dieser Stelle benötigten Typ. Dagegen sollte man bei der Übergabe zwischen den Blöcken peinlich genau auf die Breite und den Typ dieser Variablen aufpassen, damit man keine Fehlermeldung bekommt. Oft sind cast-Blöcke nötig, damit System Generator versteht um welchen Typ es sich handelt. Diese cast-Blöcke würden dann bei einem Zusammenführen der Blöcke natürlich wegfallen.

Ob diese Verbesserungen auch Vorteile für die synthetisierten Ergebnisse bringen müsste zuerst anhand einer neuen Implementierung überprüft werden, aber der Code innerhalb der Blöcke würde sich etwas verkürzen und wäre dadurch auch verständlicher.

A. Anhang

In diesem Kapitel befinden sich weitere Bilder und Quellcodes des JPEG-LS-Models, die zum Verständnis dieser Arbeit dienen sollen.

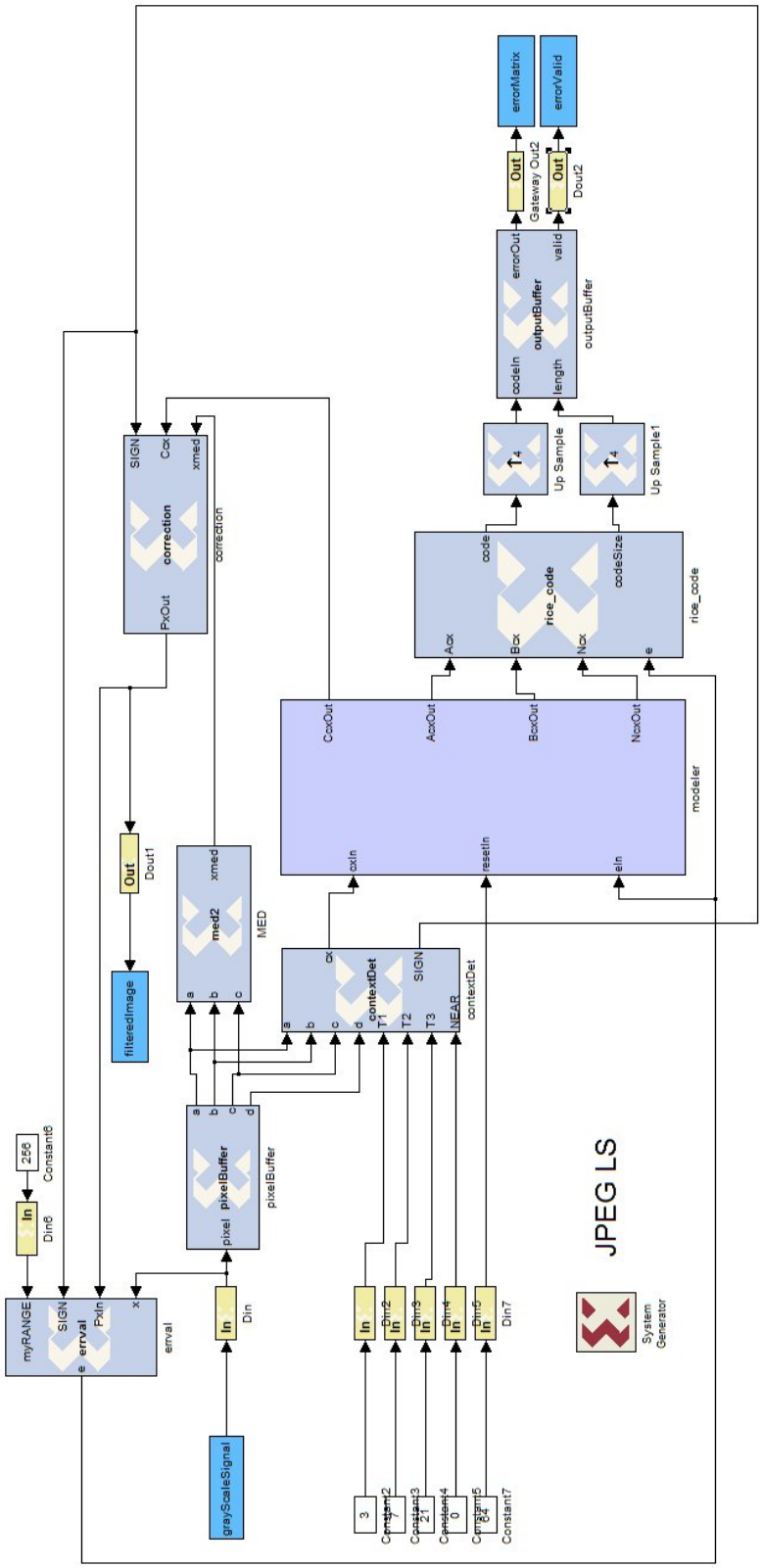


Abbildung A.1.: Finale Implementierung des Algorithmus - Landscape

A.1. Quellcodes

A.1.1. PixelBuffer

```
function [a, b, c, d] = pixelBuffer(pixel)

persistent buffer, buffer = xl_state(zeros(1, 513), pixel, 513);

persistent counter, counter = xl_state(0, 256);

persistent lastA, lastA = xl_state(0, 512);

counter = counter + 1;

b = buffer(511);

if counter == 1
    a = b;
    c = lastA;
    lastA = a;
else
    a = buffer(0);
    c = buffer(512);
end

if counter == 512
    d = b;
    counter = 0;
else
    d = buffer(510);
end

buffer.push_front_pop_back(pixel);

end
```

A.1.2. MED

```
function xmed = med2(a, b, c)

if c ≥ a && c ≥ b
    % min and max don't exists in System Generator, so we need the
    % following if
    if a ≤ b
        xmed = a;
    else
        xmed = b;
    end
elseif c ≤ a && c ≤ b
    % min and max don't exists in System Generator, so we need the
    % following if
    if a ≥ b
        xmed = a;
    else
        xmed = b;
    end
else
    xmed = a + b − c;
end


end
```

A.1.3. ContextDet

```
function [cx, SIGN] = contextDet(a, b, c, d, T1, T2, T3, NEAR)

  persistent Q, Q = xl_state(zeros(1, 3), {xlSigned, 10, 0});
  persistent D, D = xl_state(zeros(1, 3), {xlSigned, 10, 0});
  persistent counter, counter = xl_state(0, 262144);

  cxhelp = 0;

  %local gradients
  D(0) = d - b;
  D(1) = b - c;
  D(2) = c - a;

  %quantizing the local gradients
  for i=0:2
    if (D(i) <= -T3)
        Q(i) = -4;
    elseif (D(i) <= -T2)
        Q(i) = -3;
    elseif (D(i) <= -T1)
        Q(i) = -2;
    elseif (D(i) < -NEAR)
        Q(i) = -1;
    elseif (D(i) == NEAR)
        Q(i) = 0;
    elseif (D(i) < T1)
        Q(i) = 1;
    elseif (D(i) < T2)
        Q(i) = 2;
    elseif (D(i) < T3)
        Q(i) = 3;
    else
        Q(i) = 4;
    end
  end

  %calculation of the context number
  cxhelp = 9 * (9 * Q(0) + Q(1)) + Q(2);

  if cxhelp < 0
      cx = xfix({xlUnsigned, 9, 0}, (-cxhelp + 1));
      SIGN = -1;
  else
      cx = xfix({xlUnsigned, 9, 0}, (cxhelp + 1));
      SIGN = 1;
  end


end
```

A.1.4. Modeler

```
function [we, outCcx, outAcx, outBcx, outNcx] = modeler(inAcx, inCcx, cx, ...
    e, RESET, corrVar, inNcx, inBcx)

persistent isFirstStep, isFirstStep = xl_state(0, {xlUnsigned, 1, 0});
persistent weHelp, weHelp = xl_state(1, {xlBoolean});


if (cx == 1)
    isFirstStep = 1;
end
if (isFirstStep == 0)
   Acx = 4;
   Bcx = 0;
   Ccx = 0;
   Ncx = 1;
else
   Acx = inAcx;
   Bcx = inBcx;
   Ccx = inCcx;
   Ncx = inNcx;

    %Update of A[cx], B[cx], N[cx]
    if e < 0
        Acx = Acx - e;
    else
        Acx = Acx + e;
    end
    Bcx = Bcx + e;

    if Ncx == RESET
        Acx = xl_rsh(Acx, 1);
        Ncx = xl_rsh(Ncx, 1);
        if (xl_binpt(Acx) > 0)
            helpVar = xl_slice(Acx, xl_binpt(Acx) - 1, 0);
            if (helpVar > 0)
                if (Acx > 0)
                    Acx = Acx - corrVar;
                else
                    Acx = Acx + corrVar;
                end
            end
        end

        if (xl_binpt(Ncx) > 0)
            helpVar = xl_slice(Ncx, xl_binpt(Ncx) - 1, 0);
            if (helpVar > 0)
                if (Ncx < 0)
                    Ncx = Ncx - corrVar;
                else
                    Ncx = Ncx + corrVar;
                end
```

```
51              end
52          end
53
54          if Bcx ≥ 0
55              Bcx = xl_rsh(Bcx, 1);
56          else
57              Bcx = (xl_rsh(1 − Bcx, 1)) * (−1);
58          end
59
60          if (xl_binpt(Bcx) > 0)
61              helpVar = xl_slice(Bcx, xl_binpt(Bcx) − 1, 0);
62              if (helpVar > 0)
63                  if (Bcx > 0)
64                      Bcx = Bcx − corrVar;
65                  else
66                      Bcx = Bcx + corrVar;
67                  end
68              end
69          end
70
71      end
72      Ncx = Ncx + 1;
73
74      %calculate the correction value C(cx)
75      if Bcx ≤ −Ncx
76          if Ccx > −128
77              Ccx = Ccx − 1;
78          end
79          Bcx = Bcx + Ncx;
80          if Bcx ≤ −Ncx
81              Bcx = 1 − Ncx;
82          end
83      elseif Bcx > 0
84          if Ccx < 127
85              Ccx = Ccx + 1;
86          end
87          Bcx = Bcx − Ncx;
88          if Bcx > 0
89              Bcx = 0;
90          end
91      end
92  end
93
94  outCcx = Ccx;
95  outAcx = Acx;
96  outBcx = Bcx;
97  outNcx = Ncx;
98  we = weHelp;
99
100 if (isFirstStep == 0)
101     isFirstStep = 1;
102     weHelp = true;
103 end
104
105
```

```
106 end
```

A.1.5. Correction

```
function PxOut = correction( SIGN, Ccx, xmed)

if SIGN == -1
    PxOut = xmed - Ccx;
else
    PxOut = xmed + Ccx;
end

if PxOut > 255
    PxOut = 255;
elseif PxOut < 0
    PxOut = 0;
end

end
```

A.1.6. Errval

```
function e = errval(myRANGE, SIGN, PxIn, x)

  %calculate the prediction error
  if (SIGN < 0)
      e = PxIn - x;
  else
      e = x - PxIn;
  end

  %modulo reduction of the prediction error
  if e < 0
      e = e + myRANGE + 1;
  end
  if e ≥ ((myRANGE ) / 2)
      e = e - (myRANGE + 1);
  end

end
```

A.1.7. RiceCode

```
function [ code , codeSize] = rice_code(  Acx, Bcx, Ncx, e)

    one = 1;
    zero = 0;
    kTmp = 0;
    codeTmp = 0;
    codeSizeTmp = 0;
    for k=0:10

        if xl_lsh(Ncx, k) < Acx

            kTmp = kTmp + 1;
        else
            %              break
        end

    end

    if ((kTmp == 0) && ((2 * Bcx) ≤ -Ncx))
        if ( e < 0)
            positiveError = 2 * (e + 1) * (-1);
        else
            positiveError = 2 * e + 1;
        end
    else
        if ( e < 0)
            positiveError = 2 * e * (-1) - 1;
        else
            positiveError = 2 * e;
        end
    end



    lHelp = xl_nbits(positiveError) - 1;

    %slice the leading zeros to get the right number of bits for the error
    %value
    for l = 0 : lHelp
        lastBit = xl_slice(positiveError, xl_nbits(positiveError) - 1, ...
            xl_nbits(positiveError) - 1);
        if ((lastBit == 0) && (xl_nbits(positiveError) > 1))
            positiveError = xl_slice(positiveError, xl_nbits(positiveError) ...
                - 2, 0);
        end
    end
    firstBit = xl_nbits(positiveError) - kTmp;
    lastBit = xl_nbits(positiveError) - firstBit;


    qt = 0;
```

```
50      remainder = positiveError;
51
52      %calculate the quotient and the remainder
53      for m = 0 : 8
54          if ((m < firstBit) && (xl_nbits(remainder) > 1))
55              qt = xl_concat(qt, xl_slice(remainder, xl_nbits(remainder) - 1, ...
                    xl_nbits(remainder) - 1));
56              remainder = xl_slice(remainder, xl_nbits(remainder) - 2, 0);
57          else
58          end
59      end
60
61      qtmax = 23;
62      if qt < qtmax
63
64          %calculate the unary part and the code length
65          for l = 0 : 22
66              if (l < qt)
67                  codeTmp = xl_concat(codeTmp, zero);
68                  codeSizeTmp = codeSizeTmp + 1;
69              end
70          end
71          codeTmp = xl_concat(codeTmp, one);
72          codeSizeTmp = codeSizeTmp + 1;
73
74          %concat the remainder
75          for l = 0 : 7
76              if (l < xl_nbits(remainder))
77                  codeTmp = xl_concat(codeTmp, xl_slice(remainder, ...
                        xl_nbits(remainder) - 1, xl_nbits(remainder) - 1));
78                  codeSizeTmp = codeSizeTmp + 1;
79              end
80          end
81
82      else
83          %calculate the unary part and the code length
84          for l = 0 : 22
85              codeTmp = xl_concat(codeTmp, zero);
86              codeSizeTmp = codeSizeTmp + 1;
87          end
88          codeTmp = xl_concat(codeTmp, one);
89          codeSizeTmp = codeSizeTmp + 1;
90
91          %concat the remainder minus one
92          positiveError = positiveError - 1;
93          for l = 7 : -1 : 0
94              codeTmp = xl_concat(codeTmp, xl_slice(positiveError, l, l));
95              codeSizeTmp = codeSizeTmp + 1;
96          end
97      end
98
99      %slice the first zero
100     %this zero is necessary for the first concatenation
101     code = xl_slice(codeTmp, xl_nbits(codeTmp) - 2, 0);
102     codeSize = codeSizeTmp;
```

```
103
104 end
```

A.1.8. OutputBuffer

```
function [ errorOut, valid ] = outputBuffer( codeIn, length )

persistent buffer, buffer = xl_state(0, {xlUnsigned, 39, 0});
persistent counter, counter = xl_state(0, {xlUnsigned, 8, 0});
persistent counter2, counter2 = xl_state(0, 2);

%push codeIn into the buffer
if (counter2 == 0)
    tmpbuffer = buffer;
    code_leftpadded = xfix({xlUnsigned, 39, 0}, codeIn);

    %shift the buffer at lengths positions
    if length == 1
        tmpbuffer = xl_lsh(tmpbuffer, 1);
    elseif length == 2
        tmpbuffer = xl_lsh(tmpbuffer, 2);
    elseif length == 3
        tmpbuffer = xl_lsh(tmpbuffer, 3);
    elseif length == 4
        tmpbuffer = xl_lsh(tmpbuffer, 4);
    elseif length == 5
        tmpbuffer = xl_lsh(tmpbuffer, 5);
    elseif length == 6
        tmpbuffer = xl_lsh(tmpbuffer, 6);
    elseif length == 7
        tmpbuffer = xl_lsh(tmpbuffer, 7);
    elseif length == 8
        tmpbuffer = xl_lsh(tmpbuffer, 8);
    elseif length == 9
        tmpbuffer = xl_lsh(tmpbuffer, 9);
    elseif length == 10
        tmpbuffer = xl_lsh(tmpbuffer, 10);
    elseif length == 11
        tmpbuffer = xl_lsh(tmpbuffer, 11);
    elseif length == 12
        tmpbuffer = xl_lsh(tmpbuffer, 12);
    elseif length == 13
        tmpbuffer = xl_lsh(tmpbuffer, 13);
    elseif length == 14
        tmpbuffer = xl_lsh(tmpbuffer, 14);
    elseif length == 15
        tmpbuffer = xl_lsh(tmpbuffer, 15);
    elseif length == 16
        tmpbuffer = xl_lsh(tmpbuffer, 16);
    elseif length == 17
        tmpbuffer = xl_lsh(tmpbuffer, 17);
    elseif length == 18
        tmpbuffer = xl_lsh(tmpbuffer, 18);
    elseif length == 19
        tmpbuffer = xl_lsh(tmpbuffer, 19);
    elseif length == 20
```

```
52          tmpbuffer = xl_lsh(tmpbuffer, 20);
53      elseif length == 21
54          tmpbuffer = xl_lsh(tmpbuffer, 21);
55      elseif length == 22
56          tmpbuffer = xl_lsh(tmpbuffer, 22);
57      elseif length == 23
58          tmpbuffer = xl_lsh(tmpbuffer, 23);
59      elseif length == 24
60          tmpbuffer = xl_lsh(tmpbuffer, 24);
61      elseif length == 25
62          tmpbuffer = xl_lsh(tmpbuffer, 25);
63      elseif length == 26
64          tmpbuffer = xl_lsh(tmpbuffer, 26);
65      elseif length == 27
66          tmpbuffer = xl_lsh(tmpbuffer, 27);
67      elseif length == 28
68          tmpbuffer = xl_lsh(tmpbuffer, 28);
69      elseif length == 29
70          tmpbuffer = xl_lsh(tmpbuffer, 29);
71      elseif length == 30
72          tmpbuffer = xl_lsh(tmpbuffer, 30);
73      elseif length == 31
74          tmpbuffer = xl_lsh(tmpbuffer, 31);
75      end
76
77      %copy the code at the end of the buffer
78      buffer = xl_or(tmpbuffer, code_leftpadded);
79      counter = counter + length;
80  end
81
82  if (counter > 7)
83
84      tmpbuffer = buffer;
85
86      %shift the buffer to have the code on the first position
87      if counter == 8
88          tmpoutput = xl_lsh(tmpbuffer, 31);
89      elseif counter == 9
90          tmpoutput = xl_lsh(tmpbuffer, 30);
91      elseif counter == 10
92          tmpoutput = xl_lsh(tmpbuffer, 29);
93      elseif counter == 11
94          tmpoutput = xl_lsh(tmpbuffer, 28);
95      elseif counter == 12
96          tmpoutput = xl_lsh(tmpbuffer, 27);
97      elseif counter == 13
98          tmpoutput = xl_lsh(tmpbuffer, 26);
99      elseif counter == 14
100         tmpoutput = xl_lsh(tmpbuffer, 25);
101     elseif counter == 15
102         tmpoutput = xl_lsh(tmpbuffer, 24);
103     elseif counter == 16
104         tmpoutput = xl_lsh(tmpbuffer, 23);
105     elseif counter == 17
106         tmpoutput = xl_lsh(tmpbuffer, 22);
```

```
107     elseif counter == 18
108         tmpoutput = xl_lsh(tmpbuffer, 21);
109     elseif counter == 19
110         tmpoutput = xl_lsh(tmpbuffer, 20);
111     elseif counter == 20
112         tmpoutput = xl_lsh(tmpbuffer, 19);
113     elseif counter == 21
114         tmpoutput = xl_lsh(tmpbuffer, 18);
115     elseif counter == 22
116         tmpoutput = xl_lsh(tmpbuffer, 17);
117     elseif counter == 23
118         tmpoutput = xl_lsh(tmpbuffer, 16);
119     elseif counter == 24
120         tmpoutput = xl_lsh(tmpbuffer, 15);
121     elseif counter == 25
122         tmpoutput = xl_lsh(tmpbuffer, 14);
123     elseif counter == 26
124         tmpoutput = xl_lsh(tmpbuffer, 13);
125     elseif counter == 27
126         tmpoutput = xl_lsh(tmpbuffer, 12);
127     elseif counter == 28
128         tmpoutput = xl_lsh(tmpbuffer, 11);
129     elseif counter == 29
130         tmpoutput = xl_lsh(tmpbuffer, 10);
131     elseif counter == 30
132         tmpoutput = xl_lsh(tmpbuffer, 9);
133     elseif counter == 31
134         tmpoutput = xl_lsh(tmpbuffer, 8);
135     elseif counter == 32
136         tmpoutput = xl_lsh(tmpbuffer, 7);
137     elseif counter == 33
138         tmpoutput = xl_lsh(tmpbuffer, 6);
139     elseif counter == 34
140         tmpoutput = xl_lsh(tmpbuffer, 5);
141     elseif counter == 35
142         tmpoutput = xl_lsh(tmpbuffer, 4);
143     elseif counter == 36
144         tmpoutput = xl_lsh(tmpbuffer, 3);
145     elseif counter == 37
146         tmpoutput = xl_lsh(tmpbuffer, 2);
147     elseif counter == 38
148         tmpoutput = xl_lsh(tmpbuffer, 1);
149     else
150         tmpoutput = tmpbuffer;
151     end
152 
153     %get the first eight positions
154     errorOut = xl_slice(tmpoutput, 38, 31);
155     valid = 1;
156     counter = counter - 8;
157 
158 else
159     errorOut = 1;
160     valid = 0;
161 end
```

```
162
163  counter2 = counter2 + 1;
164
165
166  end
```

A.1.9. JPEG_ LS_ PreLoadFcn

```
1
2  % sysgenConv5x5_imageData is a matrix which is created by reading
3  % the xsg_icon_64.jpg image
4  [sysgenConv5x5_imageData, map] = imread('xsg_icon_64.bmp');
5
6  lineSize = size(sysgenConv5x5_imageData,1);
7  NPixels = size(sysgenConv5x5_imageData,1) * size(sysgenConv5x5_imageData,2);
8
9  grayScaleImage = uint8(sysgenConv5x5_imageData);
10
11 grayScaleSignal = reshape(grayScaleImage',1,NPixels);
12
13 % insert a column of 'time values' in front -- the from workspace
14 % block expects time followed by data on every row of the input
15 grayScaleSignal = [ double(0:NPixels-1)'  double(grayScaleSignal)'];
16 lineSizeSignal = lineSize;
```

A.1.10. JPEG_ LS_ StopFcn

```
if (exist('filteredImage','var') & exist('lineSize','var') & ...
    exist('NPixels','var'))
    filteredImageSize=size(filteredImage);
    designLatency = 20+2*lineSize;

        rawImage = uint8(floor(reshape(filteredImage(1:NPixels), lineSize, ...
            lineSize)));

        rawImage = rot90(rawImage,-1);
        rawImage = flipdim(rawImage,2);

        % Plot Original and Filtered Images
        h = figure;
        clf;
        colormap(gray(256));

        set(h,'Name','  Results');
        subplot(1,2,1);
        image(grayScaleImage), ...
            axis equal, axis square, axis off, title 'Original Image';


        subplot(1,2,2);
        image(rawImage), axis equal, axis square, axis off;
        filterTitle = 'Compressed Image';
        title(filterTitle)
        colormap(gray(256));

        imwrite(rawImage, 'filteredimage.bmp', 'bmp');

end
```

A.1.11. JPEG_ LS_ Matlab

```
function [ eMatlab ] = JPEG_LS_Matlab( input_args )

%a variable to give out values to matlab workspace
global eMatlab;

%the buffer is needed for the pixels a, b, c and d
persistent buffer, buffer = zeros(1, 513);

% the counter is needed to detect the first column
persistent counter, counter = 0;

% c is a from the line before, so we have to save it
persistent lastA, lastA = 0;

persistent Q, Q = zeros(1, 3);
persistent D, D = zeros(1, 3);

persistent N, N = ones(1, 367);
persistent B, B = zeros(1, 367);
persistent C, C = zeros(1, 367);
persistent A, A = [4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, ...
    4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, ...
    4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, ...
    4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, ...
    4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, ...
    4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, ...
    4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, ...
    4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, ...
    4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, ...
    4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, ...
    4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, ...
    4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, ...
    4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, ...
    4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, ...
    4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, ...
    4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4];

int Acx;
int Bcx;
int Ccx;
int Ncx;
int k;

filteredImage = zeros (1, 262144);
medImage      = zeros (1, 262144);


oldCx = -1;
k = 0;


```

```
37 %-------------------------------------------------------------------------
38 %-----------------------------Preload-Function----------------------------
39 %-------------------------------------------------------------------------
40 [sysgenConv5x5_imageData, map] = imread('xsg_icon_64.bmp');
41
42 lineSize = size(sysgenConv5x5_imageData,1);
43 NPixels = size(sysgenConv5x5_imageData,1) * size(sysgenConv5x5_imageData,2);
44
45 grayScaleImage = 0;
46 lineSizeSignal = 0;
47 grayScaleImage = uint8(sysgenConv5x5_imageData);
48
49 % turn the array into a vector
50 grayScaleSignal = reshape(grayScaleImage,1,NPixels);
51
52 % insert a column of 'time values' in front -- the from workspace
53 % block expects time followed by data on every row of the input
54 grayScaleSignal = [ double(1:NPixels)'  double(grayScaleSignal)'];
55 lineSizeSignal = lineSize;
56
57
58 for i=1:NPixels
59
60     %-------------------------------------------------------------------------
61     %-------------------------------PixelBuffer-------------------------------
62     %-------------------------------------------------------------------------
63
64
65     counter = counter + 1;
66
67     b = buffer(2);
68
69     if counter == 1
70         a = b;
71         c = lastA;
72         lastA = a;
73     else
74         a = buffer(lineSize + 1);
75         c = buffer(1);
76     end
77
78     if counter == lineSize
79         d = b;
80         counter = 0;
81     else
82         d = buffer(3);
83     end
84
85     row = floor((i-1)/lineSize)+1;
86     if i == lineSize
87         column = i - ((row-1) * lineSize);
88     end
89     column = mod(i-1, lineSize) + 1;
90     pixel = grayScaleImage (row, column);
91
```

```
92      buffer = buffer(2:end);
93      buffer = [buffer pixel];
94
95      %-------------------------------------------------------------------------
96      %-----------------------------------MED-Function--------------------------
97      %-------------------------------------------------------------------------
98
99      if c ≥ a && c ≥ b
100         % min and max don't exists in System Generator, so we need the
101         % following if
102         if a ≤ b
103             xmed = a;
104         else
105             xmed = b;
106         end
107     elseif c ≤ a && c ≤ b
108         % min and max don't exists in System Generator, so we need the
109         % following if
110         if a ≥ b
111             xmed = a;
112         else
113             xmed = b;
114         end
115     else
116         xmed = double(a) + double(b) - double(c);
117     end
118
119     medImage(i) = xmed;
120
121     %-------------------------------------------------------------------------
122     %-----------------------------------Context-Det---------------------------
123     %-------------------------------------------------------------------------
124
125       %local gradients
126       D(1) = double(d) - double(b);
127       D(2) = double(b) - double(c);
128       D(3) = double(c) - double(a);
129
130       %quantizing the local gradients
131       for j=1:3
132         if (D(j) ≤ -21)
133             Q(j) = -4;
134         elseif (D(j) ≤ -7)
135             Q(j) = -3;
136         elseif (D(j) ≤ -3)
137             Q(j) = -2;
138         elseif (D(j) < 0)
139             Q(j) = -1;
140         elseif (D(j) == 0)
141             Q(j) = 0;
142         elseif (D(j) < 3)
143             Q(j) = 1;
144         elseif (D(j) < 7)
145             Q(j) = 2;
146         elseif (D(j) < 21)
```

```
147             Q(j) = 3;
148         else
149             Q(j) = 4;
150         end
151     end
152
153     cxhelp = double(9 * (9 * Q(1) + Q(2)) + Q(3));
154
155     if cxhelp < 0
156         cx = -cxhelp;
157         SIGN = -1;
158     else
159         cx = cxhelp;
160         SIGN = 1;
161     end
162
163     cx = cx + 1;
164
165   %-------------------------------------------------------------------------
166   %-------------------------------------Modeler-----------------------------
167   %-------------------------------------------------------------------------
168
169 if oldCx > -1
170
171     Acx = double(A(oldCx));
172     Bcx = double(B(oldCx));
173     Ccx = double(C(oldCx));
174     Ncx = double(N(oldCx));
175
176     %Update of A[cx], B[cx], N[cx]
177     if e < 0
178         Acx = double(Acx) - double(e);
179     else
180         Acx = double(Acx) + double(e);
181     end
182     Bcx = Bcx + e;
183     if Ncx == 64
184         Acx = bitshift(Acx, -1);
185         Ncx = bitshift(Ncx, -1);
186         if Bcx ≥ 0
187             Bcx = bitshift(Bcx, -1);
188         else
189             Bcx = (bitshift(double(1 - double(Bcx)), -1)) * (-1);
190         end
191     end
192     Ncx = Ncx + 1;
193
194     %calculate the correction value C(cx)
195     if Bcx ≤ -Ncx
196         if Ccx > -128
197             Ccx = double(Ccx) - 1;
198         end
199         Bcx = double(Bcx) + double(Ncx);
200         if Bcx ≤ -Ncx
201             Bcx = 1 - double(Ncx);
```

```
202             end
203         elseif Bcx > 0
204             if Ccx < 127
205                 Ccx = double(Ccx) + 1;
206             end
207             Bcx = double(Bcx) - double(Ncx);
208             if Bcx > 0
209                 Bcx = 0;
210             end
211         end
212
213         if (oldCx == cx)
214             outCcx = double(Ccx);
215         else
216             outCcx = double(C(cx));
217         end
218
219         A(oldCx) = Acx;
220         B(oldCx) = Bcx;
221         C(oldCx) = Ccx;
222         N(oldCx) = Ncx;
223
224     else
225
226         outCcx = C(cx);
227
228     end
229
230     oldCx = double(cx);
231
232      %-------------------------------------------------------------------------
233      %-------------------------------Correction--------------------------------
234      %-------------------------------------------------------------------------
235
236      Ccx = double(outCcx);
237
238      if SIGN == -1
239          Px = double(xmed) - double(Ccx);
240      else
241          Px = double(xmed) + double(Ccx);
242      end
243
244      if Px > 255
245          Px = 255;
246      elseif Px < 0
247          Px = 0;
248      end
249
250      %-------------------------------------------------------------------------
251      %-------------------------------Error-Value-------------------------------
252      %-------------------------------------------------------------------------
253         if (SIGN < 0)
254             e = double(Px) - double(pixel);
255         else
256             e = double(pixel) - double(Px);
```

```
        end

        %modulo reduction of the prediction error
        if e < 0
            e = double(e) + 256;
        end
        if e ≥ floor((256 + 1) / 2)
            e = double(e) − 256;
        end

        k = 0;

        while bitshift(N(cx), k) < A(cx)
            k = k + 1;
        end

            if ((k == 0) && ((2 * B(cx)) ≤ −N(cx)))
                if ( e < 0)
                    eM = 2 * (e + 1) * (−1);
                else
                    eM = 2 * e + 1;
                end
            else
                if ( e < 0)
                    eM = 2 * e * (−1) − 1;
                else
                    eM = 2 * e;
                end
            end

        qt = bitshift(eM, −k);
          qtmax = 23;
          if qt < qtmax
              % quotient in unary code: qt 0's followed by a 1
              % remainder: k LSBs of MErrval
              R = [dec2bin(0, qt), '1', dec2bin(eM − bitshift(qt, k), k)];
          else
              R = [dec2bin(0, qtmax), '1', dec2bin(eM − 1, 8)];
          end

        filteredImage(i) = double(Px);

end

%−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
%−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−Stop−Function−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
%−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−

rawImage = uint8(floor(reshape(filteredImage(1:NPixels), lineSize, lineSize)));
rawImage2 = uint8(floor(reshape(medImage(1:NPixels), lineSize, lineSize)));
rawImage = rot90(rawImage,−1);
rawImage = flipdim(rawImage,2);
rawImage2 = rot90(rawImage2,−1);
rawImage2 = flipdim(rawImage2,2);
```

```
312 % Plot Original and Compressed Images
313 h = figure;
314 clf;
315 colormap(gray(256));
316
317 set(h,'Name','  Results');
318 subplot(1,2,1);
319 image(grayScaleImage), ...
320     axis equal, axis square, axis off, title 'Original Image';
321
322 subplot(1,2,2);
323 image(rawImage), axis equal, axis square, axis off;
324 filterTitle = 'Compressed Image';
325 title(filterTitle)
326 colormap(gray(256));
327
328 imwrite(rawImage, 'filteredimage.bmp', 'bmp');
329 imwrite(rawImage2, 'medimage.bmp', 'bmp');
330
331 end
```

Literaturverzeichnis

[Ins] Instruments, National: Einführung in die FPGA-Technologie: Die 5 größten Vorteile / National Instruments. http://zone.ni.com/devzone/cda/tut/p/id/7195, . – Forschungsbericht (Zitiert auf Seite 9)

[JB00] Jacob Beutel, Yongmin K.: *Display and PACS*. SPIE Press, 2000 (Handbook of medical imaging / Jacob Beutel; Harold L. Kundel; Richard L. Van Metter, eds). – ISBN 9780819436238 (Zitiert auf den Seiten 7, 15 und 16)

[MJW96] Marcelo J. Weinberger, Guillermo S. Gadiel Seroussi S. Gadiel Seroussi: LOCO-I: A Low Complexity, Context-Based, Lossless Image Compression Algorithm / Institute of Electrical and Electronics Engineers. 1996. – Forschungsbericht (Zitiert auf Seite 13)

[MJW00] Marcelo J. Weinberger, Gadiel S.: The LOCO-I Lossless Image Compression Algorithm: Principles and Standardization into JPEG-LS / Hewlett-Packard Laboratories. 2000. – Forschungsbericht (Zitiert auf den Seiten 7, 13, 14, 20 und 21)

[Sal04] Salomon, David: *Data Compression*. Springer, 2004 (ISBN : 978-0-387-40697-8) (Zitiert auf den Seiten 7, 16 und 19)

[SDR01] Shantanu D. Rane, Guillermo S.: Evaluation of JPEG-LS, the New Lossless and Controlled-Lossy Still Image Compression Standard, for Compression of High-Resolution Elevation Data / IEEE TRANSACTIONS ON GEOSCIENCE AND REMOTE SENSING. 2001. – Forschungsbericht (Zitiert auf den Seiten 7 und 21)

[Str09] Strutz, Tilo: *Bilddatenkompression: Grundlagen, Codierung, Wavelets, JPEG, MPEG, H.264*. 3. Auflage. Vieweg Teubner, 2009 (ISBN: 3540331301) (Zitiert auf den Seiten 7, 15, 17, 19 und 20)

[Xil] Xilinx: *Our History*. `http://www.xilinx.com/company/history.htm` (Zitiert auf Seite 9)

[Xil10a] Xilinx ; Xilinx (Hrsg.): *Xilinx System Generator Refrence Guide fo DSP*. http://www.xilinx.com: Xilinx, Juli 2010 (Zitiert auf Seite 41)

[Xil10b] Xilinx ; Xilinx (Hrsg.): *Xilinx System Generator User Guide fo DSP*. http://www.xilinx.com: Xilinx, Juli 2010 (Zitiert auf Seite 25)

[Xil11] Xilinx ; Xilinx (Hrsg.): *LogiCORE IP Block Memory Generator v6.1*. Xilinx, März 2011 (Zitiert auf den Seiten 7 und 42)

Alle URLs wurden zuletzt am 12.04.2011 geprüft.

Erklärung

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

(Constantin Sibianu)