(19) **Bundesrepublik Deutschland**
Deutsches Patent- und Markenamt

(10) **DE 103 08 531 B4** 2007.03.29

(12) **Patentschrift**

(21) Aktenzeichen: **103 08 531.9**
(22) Anmeldetag: **27.02.2003**
(43) Offenlegungstag: **18.09.2003**
(45) Veröffentlichungstag der Patenterteilung: **29.03.2007**

(51) Int Cl.[8]: ***H04N 1/41*** (2006.01)

(30) Unionspriorität:

60/360,835	**27.02.2002**	**US**
10/341,821	**14.01.2003**	**US**

(73) Patentinhaber:
Hewlett-Packard Development Co., L.P., Houston, Tex., US

(74) Vertreter:
Schoppe, Zimmermann, Stöckeler & Zinkler, 82049 Pullach

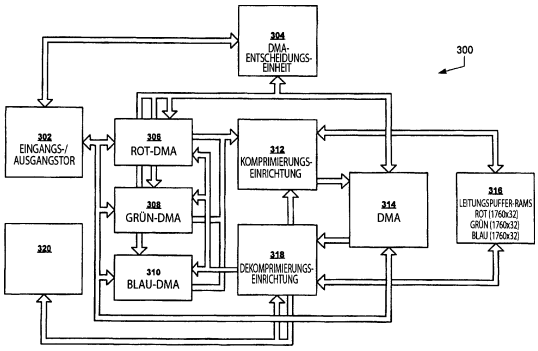
(72) Erfinder:
Clouthier, Scott C., Boise, Id., US; Harris, John H., Boise, Id., US

(56) Für die Beurteilung der Patentfähigkeit in Betracht gezogene Druckschriften:

US	**59 40 585 A**
US	**63 30 363 B1**
US	**63 04 339**
EP	**07 03 549 A2**

(54) Bezeichnung: **Hardwareimplementierte verlustfreie Seitendatenkomprimierungseinrichtung/-dekomprimierungseinrichtung**

(57) Hauptanspruch: Anwendungsspezifische integrierte Schaltung (ASIC) (202), die folgende Merkmale aufweist:
eine Direktspeicherzugriffs-(DMA-)Entscheidungseinheit (304), um Farbebenendaten syntaktisch zu analysieren, wobei die Daten jeder Farbebene an einen jeder Farbebene zugeordneten DMA (306–310) gesendet werden;
eine Komprimierungseinrichtung (312), um Farbebenendaten von dem jeder Farbebene zugeordneten DMA zu empfangen und um Farbdaten in Läufen, Keimreihenkopien und Literalen zu komprimieren, wodurch komprimierte Daten gebildet werden;
eine Dekomprimierungseinrichtung (318), um die komprimierten Daten zu dekomprimieren, wobei die Dekomprimierungseinrichtung folgende Merkmale aufweist:
einen Dekomprimierungseinrichtungskern (504), um eine Datendekomprimierung zu leiten; und
eine Zustandsmaschine (600), die in dem Dekomprimierungseinrichtungskern (504) arbeitet, um Läufe, Keimreihenkopien und Literale zu erkennen und zu dekomprimieren.



Beschreibung

[0001] Diese Offenbarung bezieht sich auf eine Hardwareimplementierung für eine verlustfreie Seitendatenkomprimierungseinrichtung und -dekomprimierungseinrichtung.

Stand der Technik

[0002] Systeme, die verlustfreie Komprimierungstechniken verwenden, erbringen allgemein keine gute Leistung in bezug auf Bilddaten, beispielsweise Photographien und andere Graphiken mit hoher Auflösung. Während komplexe verlustfreie Komprimierungsalgorithmen bei Daten, die einem Text und Geschäftsgraphiken zugeordnet sind (Strichvorlage, Balkendiagramme und dergleichen), eine Komprimierung von 100:1 erreichen, erreichen sie üblicherweise eine weniger als 2:1 betragende Komprimierung von Bilddaten. Umgekehrt gilt, daß, während Daten, die Photographien und anderen Bildern zugeordnet sind, mit einem „verlustbehafteten" Algorithmus effektiv komprimiert werden können, ohne die wahrnehmbare Bildqualität wesentlich zu beeinträchtigen, verlustbehaftete Algorithmen Textdaten auch bei relativ niedrigen Komprimierungspegeln sichtbar verschlechtern (beispielsweise indem sie visuelle Artefakte hinterlassen). Ferner erzielen Techniken einer verlustbehafteten Komprimierung keine so hohen Komprimierungsverhältnisse wie diejenigen, die allgemein erwartet werden, wenn Daten auf Textbildern basieren. Ferner sind die Vorteile einer JPEG-ähnlichen Komprimierung gegenüber anderen Techniken verringert, wenn Bilddaten komprimiert werden, die unter Verwendung eines Pixelnachbildungsskalierungsalgorithmus, der für rasterisierte zusammengesetzte Dokumente üblich ist, skaliert wurden (z.B. Bilddaten von 150 Punkten pro Zoll („dpi"), die bis zu einer Auflösung von 300 dpi oder 600 dpi hochskaliert wurden). Während verlustbehaftete Algorithmen bei Bilddaten eine gute Leistung erbringen, liefern somit verlustfreie Algorithmen bessere Ergebnisse bei textbasierten Daten.

[0003] Lösungen, die eine Mischung aus einer verlustbehafteten und einer verlustfreien Datenkomprimierung verwenden, sind oft langsam und komplex. Beispielsweise sind manchmal Text- und Bilddaten in verschiedene Kanäle getrennt, wobei einer die Bilder enthält und dabei eine verlustbehaftete Komprimierungstechnik wie beispielsweise JPEG verwendet und der andere eine verlustfreie Komprimierungstechnik verwendet, die besser für Text und einfache Geschäftsgraphiken geeignet ist. Diese Trennung von Daten in einzelne Kanäle kann langsam sein, und die Ergebnisse hängen von der Architektur der Rasterisierungsmaschine ab, die das zusammengesetzte Dokument anfänglich rasterisierte. Überdies verringert die Verwendung eines verlustbehafteten Algorithmus Datenverarbeitungsgeschwindigkeiten, was die Leistungsfähigkeit von Vorrichtungen, die konfiguriert sind, um „Druckkopien" auf Druckmedien auszugeben, verlangsamt. Wiederum sind die Vorteile eines Algorithmus vom JPEG-Typ für Bilder, die skaliert wurden, verringert. Überdies ist die relative Langsamkeit von JPEG auch dann nicht verbessert, wenn mit Hochauflösungspixeln nachgebildete Bilddaten komprimiert werden.

[0004] Somit liefern derzeit bekannte Komprimierungs- und Dekomprimierungstechniken bei vielen Anwendungen keine zufriedenstellenden Ergebnisse.

Aufgabenstellung

[0005] Es ist die Aufgabe der vorliegenden Erfindung, anwendungsspezifische integrierte Schaltungen (ASICs), Verfahren und Drucker zu schaffen, die eine verbesserte verlustfreie Komprimierung von Bilddaten ermöglichen.

[0006] Diese Aufgabe wird durch ASICs gemäß den Ansprüchen 1, 7, 16 oder 27, durch Verfahren gemäß den Ansprüchen 13 oder 22 sowie durch Drucker gemäß den Ansprüchen 17, 28 oder 29 gelöst.

[0007] Ein Ausführungsbeispiel der Erfindung betrifft einen Drucker, der konfiguriert ist, um Farbebenendaten zu verschachteln. Die verschachtelten Daten werden komprimiert, wodurch komprimierte Daten gebildet werden, die Läufe, Keimreihenkopien und Literale aufweisen. Bei dem Komprimierungsvorgang werden Befehle gepuffert, um einen unabhängigen und gleichzeitigen Betrieb eines Laufmoduls, eines Keimreihenkopiemoduls und eines Literalmoduls, die beim Bilden der komprimierten Daten verwendet werden, zu ermöglichen.

Ausführungsbeispiel

[0008] Bevorzugte Ausführungsbeispiele der vorliegenden Erfindung werden nachfolgend Bezug nehmend auf die beiliegenden Zeichnungen, bei denen gleiche Merkmale und Komponenten durchwegs mit denselben Bezugszeichen bezeichnet sind, näher erläutert. Es zeigen:

[0009] Fig. 1 eine Umgebung, bei der eine Version der hardwareimplementierten verlustfreien Seitendatenkomprimierungseinrichtung/-dekomprimierungseinrichtung implementiert sein kann;

[0010] Fig. 2 eine Veranschaulichung einer exemplarischen Druckvorrichtung, bei der eine Version der hardwareimplementierten verlustfreien Seitendatenkomprimierungseinrichtung/-dekomprimierungseinrichtung verkörpert ist;

[0011] Fig. 3 ein Blockdiagramm, das ein Beispiel

von Komponenten auf hoher Ebene bei einer Hardwareimplementierung einer Vorrichtung zur Komprimierung und Dekomprimierung von Farbebenendaten veranschaulicht;

[0012] Fig. 4 ein Blockdiagramm, das ein Beispiel einer möglichen Implementierung der in Fig. 3 zu sehenden Komprimierungseinrichtung veranschaulicht;

[0013] Fig. 5 ein Blockdiagramm, das ein Beispiel einer möglichen Implementierung der in Fig. 3 zu sehenden Dekomprimierungseinrichtung veranschaulicht;

[0014] Fig. 6 ein Blockdiagramm, das ein Beispiel einer möglichen Implementierung einer bei der Dekomprimierungseinrichtung der Fig. 5 implementierten Zustandsmaschine veranschaulicht;

[0015] Fig. 7 eine schematische Darstellung einer exemplarischen Implementierung einer Datenstruktur, die komprimierten Daten zugeordnet ist;

[0016] Fig. 8 ein Flußdiagramm, das ein Beispiel eines Verfahrens zum Komprimieren von Daten veranschaulicht; und

[0017] Fig. 9 ein Flußdiagramm, das ein Beispiel eines Verfahrens zum Dekomprimieren von Daten veranschaulicht.

[0018] Das US-Patent 6,304,339 von Miller et al., das am 16. Oktober 2001 erteilt und an die Firma Hewlett-Packard übertragen wurde, ist durch Bezugnahme in dieses Dokument aufgenommen. Die Referenz Miller'339 beschreibt eine Softwareimplementierung eines nahezu verlustfreien Komprimierungsalgorithmus, der einen gewissen Verlust in der blauen Farbebene erfährt. Das US-Patent 6,373,583 von Wood et al., das am 16. April 2002 erteilt und an die Firma Hewlett-Packard übertragen wurde, ist durch Bezugnahme in dieses Dokument aufgenommen. Die Referenz Wood'583 beschreibt eine Datenkomprimierung für zusammengesetzte Dokumente. Die vorläufige Anmeldung 60/360,835 derselben Erfinder, die am 27.02.2002 erteilt und an die Firma Hewlett-Packard übertragen wurde, ist ebenfalls durch Bezugnahme in dieses Dokument aufgenommen.

[0019] Fig. 1 zeigt eine Netzwerkumgebung **100**, bei der ein Drucker oder eine andere Vorrichtung konfiguriert sein kann, um eine hardwareimplementierte verlustfreie Seitendatenkomprimierung und -dekomprimierung zu verwenden. Ein Druckserver oder Dateiserver **102** ist konfiguriert, um von irgendeiner aus einer Mehrzahl von Arbeitsstationen **104** über ein Netzwerk **114** einen Druckauftrag zu empfangen. Der Druckauftrag kann an eine Ausgabevorrichtung gesendet werden, die konfiguriert ist, um eine hardwareimplementierte verlustfreie Datenkomprimierung und -dekomprimierung einzusetzen. Eine derartige Ausgabevorrichtung kann in einer beliebigen Form vorliegen, beispielsweise in der Form eines Druckers **106**, eines multifunktionalen Peripheriegeräts **108**, eines Faxgeräts **110**, eines Netzwerkkopierers **112** oder einer anderen Vorrichtung.

[0020] Fig. 2 zeigt einen exemplarischen Drucker **106**, der eine anwendungsspezifische integrierte Schaltung (ASIC) **202** auf, die konfiguriert ist, um eine verlustfreie Datenkomprimierung und -dekomprimierung zu implementieren. Eine Steuerung **204** und eine Druckmaschine **206** kommunizieren mit der ASIC **202**, um eine Druckausgabe zu bewirken. Während Fig. 2 die in dem Drucker **106** implementierte ASIC **202** zeigt, könnte alternativ statt dessen eine beliebige Ausgabevorrichtung, beispielsweise die in Fig. 1 gezeigte, verwendet werden. Desgleichen gilt, daß, während die Implementierung in einer ASIC aus Gründen der Geschwindigkeit und Wirtschaftlichkeit in der Regel bevorzugt wird, bei manchen Anwendungen statt dessen ein Mehrzweckprozessor oder eine Mehrzwecksteuerung verwendet werden könnte.

[0021] Fig. 3 zeigt ein Blockdiagramm, das Komponenten einer hohen Ebene in einer Implementierung **300** einer Vorrichtung **202** zur Komprimierung und Dekomprimierung von Farbebenendaten veranschaulicht. Die Vorrichtung kann als eine anwendungsspezifische integrierte Schaltung (ASIC) in Hardware konfiguriert sein, oder sie kann konfiguriert sein, um an einer alternativen Hardwarevorrichtung, die Anweisungen ausführt, zu arbeiten.

[0022] Ein Eingangs-/Ausgangstor **302** ermöglicht, daß Daten in der Komprimierungs- und Dekomprimierungseinrichtung empfangen und von derselben gesendet werden. Beispielsweise können Farbebenendaten durch das Tor **302** von einer Quelle empfangen werden, beispielsweise einem Interpretierer, der eine Seitenbeschreibungssprache in Vorrichtung-Bereit-Bits übersetzt. Nach einer Komprimierung können die komprimierten Daten durch das Tor **302** passieren, wobei sie in der Regel für eine Speicherstelle bestimmt sind. Die komprimierten Daten können wiederum durch das Tor **302** passieren, um eine Dekomprimierung zu ermöglichen. Nach einer Dekomprimierung können die dekomprimierten Daten auf dem Weg zu einer Druckmaschine ein letztes Mal das Tor **302** passieren. Dieser Datenverkehr wird teilweise durch eine DMA-Entscheidungseinheit (DMA = direkter Speicherzugriff) **304** geregelt, die Farbebenendaten syntaktisch analysiert (parse) und die DMAs **306**, **308**, **310** und DMA **314** und ihren Zugriff auf das Eingangs-/Ausgangstor **302** koordiniert, wodurch ermöglicht wird, daß Daten auf eine geordnete Weise bewegt werden.

[0023] Eine Komprimierungseinrichtung **312** ist kon-

figuriert, um Daten von den DMA-Einheiten **306**, **308**, **310** zu empfangen, und komprimiert die Daten, bei manchen Vorgängen unter Verwendung von Zeilenpuffern **316**. Eine Ausgabe aus der Komprimierungseinrichtung **312** wird durch die DMA-Einheiten und das Tor **302** gesendet, in der Regel an eine Speicherstelle.

[0024] Eine Dekomprimierungseinrichtung **318** ist konfiguriert, um komprimierte Daten zu empfangen, die durch das Tor **302** und den DMA **314** rückgerufen werden. Die Dekomprimierungseinrichtung **318** dekomprimiert die Daten in 3 Ausgabeebenen – bei einer Implementierung mit einem Pixel pro Taktzyklus – und die 3 dekomprimierten Ebenen werden anschließend durch die DMAs **306**–**310** aus dem Tor **302** hinaustransferiert.

[0025] Eine Registerschnittstelle **320** ermöglicht ein Programmieren von Variablen und/oder Registern in der Komprimierungseinrichtung **312** und der Dekomprimierungseinrichtung **318**, wodurch Faktoren wie Pixel pro Zeile und andere Parameter gesteuert werden.

[0026] Fig. 4 zeigt ein Blockdiagramm, das Einzelheiten eines Ausführungsbeispiels der in Fig. 3 zu sehenden Komprimierungseinrichtung **312** veranschaulicht. Ein Konfigurationsregister **402** enthält Steuer- und Datenregister, die in der Komprimierungseinrichtung **312** verwendet werden. Die Bildgröße (Bytes pro Zeile und Zeilen pro Streifen), Anfangskeimreihenpixel und Komprimierungsmodus sind programmierbare Parameter, die in dem Konfigurationsregister **402** konfiguriert sind. Eine Adreßdecodierung für DMA-Register wird ebenfalls in dem Konfigurationsregister **402** erzeugt. Ein Unterbrechungsaktivierungsregister und ein Statusregister sind ebenfalls in dem Konfigurationsregister **402** enthalten.

[0027] Eine Datenverschachtelungseinrichtung **404** ist dafür verantwortlich, die Rot-, Grün- und Blaudaten von den drei DMAs **306**, **308**, **310** zu einem einzigen Strom zu verschachteln. Durch ein Verschachteln der Farbdaten kombiniert die Datenverschachtelungseinrichtung **404** drei Ströme von planaren Daten zu einem einzigen Strom, der aus verschachtelten Daten besteht, beispielsweise in einer Form RGB, RGB, RGB ..., wobei R, G und B Daten darstellen, die Rot-, Grün- und Blaudaten von den drei DMAs **306**, **308** bzw. **310** zugeordnet sind. Man beachte, daß ersatzweise auch alternative Farben verwendet werden könnten, beispielsweise CMY (Cyan, Magenta, Gelb) oder diejenigen, die in anderen dreidimensionalen Farbräumen zu finden sind. Jeder der drei Eingangskanäle weist zwei Puffer in der Datenverschachtelungseinrichtung **404** auf, um Eingangsdaten zu speichern. Transfers von jedem der DMA-Kanäle bestehen aus vier Bytes von Daten aus derselben Farbebene. Transfers von der Datenverschachtelungseinrichtung **404** bestehen aus vier Bytes von verschachtelten Daten. Da ein Pixel aus drei Bytes zusammengesetzt ist (RGB), muß das Ausgabewort mit einer der drei folgenden Kombinationen gepackt sein: (RGBR, GBRG, BRGB). Man beachte, daß statt RGB auch alternative Farben eingesetzt werden könnten, beispielsweise CMY (Cyan, Magenta, Gelb), und daß RGB drei unterschiedliche Farben angeben soll, kein Erfordernis spezifischer Farben. Eine Steuerung in der Datenverschachtelungseinrichtung **404** ist konfiguriert, um die drei Kombinationen zyklisch zu durchlaufen, um die richtige Reihenfolge aufrechtzuerhalten. Die Ende-Des-Bildes-Markierung (EOI-Markierung, EOI = end of image), die von jedem der drei DMA-Kanäle kommt, kann in Farbdaten gespeichert sein, beispielsweise einschließlich der Markierung in einem Byte von Blaudaten.

[0028] Ein Eingangsdateninterpretierer **406** interpretiert verschachtelte Farbdaten, die von der Datenverschachtelungseinrichtung **404** empfangen werden. Auf einen Empfang hin werden verschachtelte Daten in einem von vier Eingangspuffern gespeichert. Der Interpretierer **406** ist konfiguriert, um die gepufferten Daten mit einem in einem Zeilenpuffer **416**–**420** gespeicherten Keimreihenpixel zu vergleichen. Signale, die eine Keimreihen-Übereinstimmung des aktuellen Pixels, des nächsten Pixels und des um zwei vorausgehenden Pixels anzeigen, werden je nach Bedarf erzeugt. Der Eingangsdateninterpretierer **406** ist ferner konfiguriert, um einen Lauf des aktuellen Pixels mit dem nächsten Pixel und des nächsten Pixels mit dem um zwei vorausgehenden Pixel zu erfassen. Laufbytes werden in einem von zwei Ping-Pong-Puffern gespeichert und herausgemultiplext und an die Laufbe- fehlseinheit **426** gesendet. Die Pixel-Cachespeicherung-Vergleiche (nordöstlich, westlich, Cache, neu) werden auch in dem Eingangsdateninterpretierer **406** durchgeführt, und die codierte Ausgabe wird an die Literaleinheit **430** gesendet.

[0029] Ein Entropiemodul **408**, **410**, **412** für jeden der drei Pixelströme ist ebenfalls in dem Eingangsdateninterpretierer **406** enthalten. Die Entropiemodule **408**–**412** führen eine Messung der Effektivität der Komprimierung durch, was die Option eines Abbrechens des Komprimierungsvorgangs, wenn er weniger effektiv ist, unterstützt. Das Entropiemodul zählt Veränderungen zwischen sequentiellen Pixeln, um ein Maß einer Entropie auf einer Zeilenbasis zu ermitteln. Mehrere Zeilen können miteinander gemittelt werden, um ein breiteres Maß der Entropie zu erhalten. Der Eingangsdateninterpretierer **406** kann konfiguriert sein, um eine Auswahl der maximalen Anzahl von bei der Berechnung verwendeten Zeilen zu ermöglichen. Das Entropiemodul **408**–**412** wird in dem Eingangsdateninterpretierer **406** dreimal instanziert,

einmal für jede Farbebene.

[0030] Eine Zeilenpuffersteuerung **414** ist konfiguriert, um einen Zugriff auf die und von den Zeilenpuffer-RAM-Puffern **416**–**420**, die verwendet werden, um Daten zu speichern, die die Keimreihe darstellen, zu steuern. Keimreihenvergleiche werden während eines Prozesses durchgeführt, der ein gleichzeitig mit einem Lesen der vorherigen Zeile von Pixeldaten erfolgendes Schreiben einer aktuellen Zeile von Pixeldaten in die Puffer **416**–**420** umfaßt. Ein logischer RAM-Speicherbereich für jede Farbebene liegt vor, d.h. drei RAM-Bereiche, wobei jeder seine eigene Steuerung aufweist. Ein Puffern ist für Daten vorgesehen, die in den RAM geschrieben werden, um jeden Bytestrom von Daten in 32-Bit-Daten umzuwandeln. Dort, wo die RAM-Datenbreite 32 Bits (d.h. 4 Bytes) beträgt, ist die Anzahl benötigter Zugriffe nicht untragbar. Ein Puffern ist auch für Daten vorgesehen, die von dem RAM geschrieben werden, da der Eingangsdateninterpretierer **406** Daten für das aktuelle Pixel, das dem aktuellen Pixel um eines vorausgehende Pixel und das dem aktuellen Pixel um zwei vorausgehende Pixel benötigt. Diese Daten werden benötigt, um zu ermöglichen, daß Keimreihenvergleiche durchgeführt werden.

[0031] Eine Hauptkomprimierungssteuerung **422** ist konfiguriert, um zu bestimmen, ob ein Pixel als Teil von beliebigen von drei Typen von komprimierten Daten codiert werden sollte: eines Laufs, einer Keimreihenkopie oder eines Literals. Eine Keimreihenkopie bedeutet, daß das aktuelle Byte dasselbe ist wie das decodierte Byte in der vorhergehenden Reihe. Ein Lauf beschreibt ein dekomprimiertes Byte, das dasselbe ist wie das vorhergehende Byte. Ein Literal ist ein dekomprimiertes Byte, das keine Keimreihenkopie und das nicht Teil eines Laufs ist. Eine Keimreihenkopie ist immer ein Teil eines Laufs oder eines Literals, es sei denn, die Keimreihenkopie erstreckt sich bis zum Ende der Zeile. In diesem Fall spezifiziert der Befehl, der die Keimreihenkopie festlegt, ausschließlich eine Keimreihenkopie.

[0032] Ein Neusortierungspuffer **424** ist konfiguriert, um bis zu acht Befehle zu speichern; ein derartiges Puffern von Befehlen ermöglicht, daß die Lauf-, Keimreihenkopie- und Literaleinheiten **426**, **428**, **430** gleichzeitig arbeiten. Der Neusortierungspuffer **424** verfolgt, welcher Einheit **426**–**430** ermöglicht werden sollte, Daten in das Komprimierungsausgangsmodul **434** zu schreiben. Die Keimreihenkopie- und Laufzählwerte werden ebenfalls in dem Neusortierungspuffer **424** berechnet. Bei dem Ausführungsbeispiel der Fig. 4 liegen vier Keimreihenzählwertpuffer und zwei Laufzählwertpuffer vor, obwohl statt dessen auch alternative Konfigurationen eingesetzt werden könnten. Ein kreisförmiger Zeiger ist konfiguriert, um auf den Puffer zu zeigen, der derzeit in Gebrauch ist. Der Keimreihenkopiezählwert wird durch die Lauf-, Keimreihenkopie- und Literaleinheiten **426**, **428** und **430** verwendet. Der Laufzählwert wird lediglich durch die Laufeinheit **426** verwendet.

[0033] Die Laufbefehlseinheit **426** ist konfiguriert, um Laufbefehle zu codieren und um Laufzählwertbytes oder Endnullen auszugeben. Diese Funktionalität kann durch eine Verwendung einer Laufbefehlzustandsmaschine erfüllt werden, die auch Datenübertragungen und einen Quittungsaustausch mit dem Komprimierungsausgangsmodul **434** durchführen kann. Die Laufbefehlseinheit **426** gibt höchstens ein Datenpixel pro Laufbefehl aus, gibt jedoch eventuell kein Datenpixel aus, wenn es als ein westliches, nordöstliches oder Cache-Pixel codiert ist. Falls Keimreihenkopiepixel als Teil des Laufbefehls codiert werden sollen, verwendet die Laufbefehlzustandsmaschine den Gesamtkeimreihenkopiezählwert von der Komprimierungssteuerung **422** und gewährleistet, daß der Zählwert korrekt codiert wird. Wie die Literaleinheit **424** können zwei aufeinanderfolgende Laufvorgänge durchgeführt werden, bevor die Laufbefehlseinheit **426** anhalten und darauf warten muß, daß der nächste ausstehende Befehl ausgegeben wird. Die Ende-Des-Bildes-Markierung (EOI-Markierung) wird ebenfalls durch die Laufbefehlseinheit **426** erzeugt, wenn das bzw. die letzte(n) Pixel, das bzw. die codiert werden soll(en), Teil eines Laufs ist bzw. sind.

[0034] Die Keimreihenkopieeinheit **428** ist konfiguriert, um Keimreihenkopiebefehle zu codieren und um Keimreihenzählwertbytes oder Endnullen auszugeben. Diese Funktionalität wird durch eine Keimreihenkopiezustandsmaschine erfüllt, die ebenfalls Datenübertragungen mit der Ausgabeeinheit **428** unter Verwendung von Quittungsaustausch (handshake) durchführt. Im Gegensatz zu der Lauf- oder Literaleinheit **426**, **430** werden von der Keimreihenkopieeinheit **428** keine tatsächlichen Datenpixel ausgegeben. Statt dessen wird ein Code ausgegeben, der das Erfordernis, die Keimreihe zu kopieren, angibt. Wie bei der Literalbefehlseinheit **430** können zwei aufeinanderfolgende Keimreihenkopievorgänge durchgeführt werden, bevor die Einheit anhalten und darauf warten muß, daß der erste ausstehende Befehl ausgegeben wird, obwohl auch eine alternative Konfiguration instanziert werden könnte. Die Ende-Des-Bildes-Markierung (EOI-Markierung) könnte auch durch die Keimreihenkopieeinheit **428** erzeugt werden, wenn das bzw. die letzte(n) zu codierende(n) Pixel Teil einer Keimreihenkopie ist bzw. sind.

[0035] Die Literaleinheit **430** verwendet eine interne RAM-Vorrichtung (136 × 53 Bits) **432**, um Literalpixel als Stapel zu verarbeiten. Die RAM-Vorrichtung **432** speichert bis zu 255 Pixel, die in den komprimierten Strom geschrieben werden, wenn sie nicht durch die Laufbefehlseinheit **426** (weil sie kein Lauf sind) oder die Keimreihenkopieeinheit **428** (weil sie nicht Teil ei-

ner Keimreihenkopie sind) codiert werden können. Die Literaleinheit **430** ist dafür verantwortlich, Daten einzurichten, die in den RAM **432** geschrieben werden, und dafür, Daten aus dem RAM **432** auszulesen. Jedes zweite Pixel wird von dem Eingangsinterpretierermodul **406** registriert, das erlaubt, daß 2 Pixel (48 Bits) gleichzeitig in den RAM **432** geschrieben werden. Da manche Literalpixel codiert sein können, wird ein Flag verwendet, um jedes Pixel in dem RAM als ein gültiges Pixel zu markieren, das ausgegeben werden wird. Ein Puffern ist vorgesehen, um das Codieren zweier aufeinanderfolgender Literalvorgänge zu erlauben. Eine Literalbefehlzustandsmaschine kann verwendet werden, um das Codieren des Literalbefehls und das Schreiben von Datenpixeln in die Ausgabeeinheit **434** zu steuern. Falls Keimreihenkopiepixel als Teil eines Literalbefehls codiert werden sollen, verwendet die Literbefehlzustandsmaschine den Gesamtkeimreihenkopiezählwert von der Komprimierungssteuerung **422** und gewährleistet, daß der Zählwert korrekt codiert wird. Die Ende-Des-Bildes-Markierung (EOI-Markierung) wird ferner durch die Literaleinheit **430** erzeugt, wenn das bzw. die letzte(n) zu codierende(n) Pixel (ein) Literalpixel ist bzw. sind.

[0036] Das Komprimierungsausgangsmodul **434** ist für ein Puffern der komprimierten Ausgangsdaten von den Lauf-, Keimreihen- und Literalkopieeinheiten **426**, **428**, **430** und für ein Sequenzieren derselben zum Zweck einer Übertragung an den abgehenden DMA **314** in der richtigen Pixelreihenfolge verantwortlich. Das Komprimierungsausgangsmodul **434** enthält einen 16-Byte-FIFO-Puffer, in dem Lauf-, Keimreihenkopie- oder Literaldaten gespeichert werden, bis der DMA **314** die Daten anfordert. Bei einer Implementierung ist das Komprimierungsausgangsmodul **434** konfiguriert, um 1, 3 oder 6 Datenbytes zu empfangen. Eine Übertragung eines Bytes kann von jeglicher der drei Einheiten (der Laufeinheit **226**, der Keimreihenkopieeinheit **428** oder der Literaleinheit **430**) erfolgen. Eine Übertragung von 3 Bytes (1 Pixel) kann von der Laufeinheit **426** oder der Literaleinheit **430** erfolgen. Eine Übertragung von 6 Bytes (2 Pixeln) kann lediglich von der Literaleinheit **430** erfolgen. Diese Übertragungsmengen könnten geändert werden, um einer gewünschten Anwendung zu entsprechen.

[0037] Fig. 5 zeigt ein Blockdiagramm, das Einzelheiten der in Fig. 1 zu sehenden Dekomprimierungseinrichtung **318** veranschaulicht. Eine Konfigurationseinheit **502** enthält in der Regel Steuer- und Datenregister für die Dekomprimierungseinrichtung **318**. Die Bildgröße (Bytes pro Zeile und Zeilen pro Streifen), das Anfangskeimreihenpixel und der Komprimierungsmodus können programmierbare Parameter sein. In diesem Modul oder in dem Komprimierungsmodul **312** könnte ein Adressendecodieren für die DMA-Register erzeugt werden. Das Unterbrechungsaktivierungsregister und das Statusregister können ebenfalls in der Konfigurationseinheit **502** enthalten sein.

[0038] Ein Dekomprimierungseinrichtungskern **504** liefert eine Funktionalität, die für den Betrieb der Dekomprimierungseinrichtung **318** wesentlich ist. Der Dekomprimierungseinrichtungskern **504** nimmt einen komprimierten Strom von 32-Bit-Wörtern von dem Eingangs-DMA **508** entgegen und gibt einen Strom von 32-Bit-Wörtern von dekomprimierten Pixeln an RAM-Steuerungen **510**, **512**, **514**, die jeder Farbebene (Rot, Grün und Blau) zugeordnet sind, aus. Der Strom von dekomprimierten Pixeln wird in die Zeilenpuffer-RAMs **416**, **418**, **420**, die den Steuerungen **510**–**514** zugeordnet sind, geschrieben, um eine Identifizierung von Keimreihen- oder nordöstlich codierten Übereinstimmungen von Pixeln von der nächsten Zeile zu ermöglichen.

[0039] Ein exemplarischer Entwurf des Dekomprimierungseinrichtungskerns **504** kann durch eine Bezugnahme auf vier Teile verstanden werden. Der erste Teil besteht aus Ladegleichungen, die konfiguriert sind, um parallel zu laufen und dadurch alle möglichen Fälle für Läufe, Keimreihenkopien und Literale zu erfassen. Die Ladegleichungen können asynchron arbeiten, und die Ladegleichungen können Puffer umfassen, um ein Vorgreifen (looking ahead) zu ermöglichen, d.h. eine Prüfung von Pixeldaten, die noch nicht dekomprimierten Pixeln zugeordnet sind, während des Dekomprimierens von Daten, die einem aktuellen Pixel (d.h. einem Pixel, das Daten aufweist, die derzeit dekomprimiert werden) zugeordnet sind. Der zweite Teil umfaßt kombinatorische Gleichungen, die konfiguriert sind, um den Datenfluß in den und aus dem Block zu/von den Zeilenpuffern **316**, zu und von dem DMA **508** und zu und von in Flußrichtung nachgelagerten Vorrichtungen, die die in Fig. 3 gezeigten Rot-, Grün- und Blau-DMAs **306**, **308**, **310** umfassen, zu steuern. Der dritte Abschnitt besteht aus Registern, die konfiguriert sind, um auf der Basis der Ladegleichungen geladen zu werden.

[0040] Der vierte Teil ist eine Dekomprimierungseinrichtungszustandsmaschine **600**, die die aktuelle Operation bzw. den aktuellen Betrieb der Dekomprimierungseinrichtung **318** verfolgt und leitet und unter Bezugnahme auf Fig. 6 nachstehend ausführlicher erörtert wird. Die Dekomprimierungseinrichtungszustandsmaschine **600** ist konfiguriert, um Pixel in einer Sequenz zu dekomprimieren und dabei ausreichend vorzugreifen (d.h. während des Dekomprimierens einer Sequenz von Pixeldaten noch nicht dekomprimierte Pixeldaten zu betrachten), um zu bestimmen, wie aktuelle komprimierte Pixeldaten dekomprimiert werden sollten. Bei der Implementierung des Dekomprimierungseinrichtungskerns **504** der Fig. 5 umfaßt ein Vorgriffspuffer (look-ahead buffer) **506** zwei 4-Byte-Puffer, die eine Speicherung von Daten er-

möglichen, die Pixeln zugeordnet sind, die noch nicht dekomprimiert sind, um die Dekomprimierung aktueller Pixeldaten zu unterstützen. Bei der offenbarten Implementierung ist die Dekomprimierungseinrichtungszustandsmaschine **600** konfiguriert, um dekomprimierte Daten, die einem Pixel zugeordnet sind, bei jedem Taktzyklus der Hardware, beispielsweise einer ASIC, in der die Dekomprimierungseinrichtung **318** instanziert ist, auszugeben.

[0041] Die RAM-Steuerungen **510**, **512**, **514** steuern Zugriffe auf die und von den Zeilenpuffer-RAMs **416**, **418**, **420**, die verwendet werden, um Keimreihendaten zu speichern. Die aktuelle Zeile von dekomprimierten Pixeldaten wird zur selben Zeit in den RAM **510**–**514** geschrieben, wie die vorhergehende Zeile von Pixeldaten gelesen wird. Ein Pixel von dem Zeilenpuffer wird verwendet, falls eine Keimreihenkopie oder eine NE-Übereinstimmung (NE = Nordost) erfaßt wird. Da für jede Farbebene ein RAM **416**, **418**, **420** vorliegt, gibt es drei RAMs – jeder mit seiner eigenen Steuerung **510**–**514**. Bei der betrachteten Implementierung ist ein Puffern für Daten vorgesehen, die in den RAM geschrieben werden, um jeden Bytedatenstrom in 32-Bit-Daten umzuwandeln. Eine Verwendung von RAM-Daten, die eine Breite von 32 Bits (4 Bytes) aufweisen, ermöglicht, daß ein Zugreifen auf einen Speicher bei einer akzeptablen Rate durchgeführt wird. Ein Puffern ist auch für Daten vorgesehen, die aus dem RAM geschrieben werden, weil der Dekomprimierungseinrichtungskern **504** Daten benötigt, die dem aktuellen Pixel, dem um eins vorausgehenden Pixel und dem um zwei vorausgehenden Pixel zugeordnet sind, so daß die Ladegleichungen verwendet werden können.

[0042] Ein Schnüffeltor (snoop port) **516** ermöglicht, daß dekomprimierte Pixel außerhalb der Hardwareimplementierung **300** der Komprimierungseinrichtung/Dekomprimierungseinrichtung beobachtet werden, und kann für Fehlersuchzwecke verwendet werden.

[0043] Fig. 6 zeigt ein Blockdiagramm, das Einzelheiten einer Zustandsmaschine **600** veranschaulicht, die in dem Dekomprimierungseinrichtungskern **504** der Fig. 5 implementiert ist. Es werden sieben Zustände offenbart, einschließlich eines IDLE-Zustands (Ruhezustands) **602**, eines DECODE_COMMAND-Zustands (Decodiere_Befehl-Zustands) **604**, eines WRITE_SRC_INI-Zustands (Schreibe-Keimreihenkopier-Anfangszustands) **606**, eines WRITE_SRC_SUB-Zustands (Schreibe-Keimreihenkopie-Folgezustands) **608**, eines WAIT_FOR_FP-Zustands (Warte-Auf-Erstes-Pixel-Zustands) **610**, eines WRITE_REP_INI-Zustands (Schreibe-Ersetzung-Anfangszustands) **612** und eines WRITE_REP_SUB-Zustands (Schreibe-Ersetzung-Folgezustands) **614**.

[0044] Ein Transfer zwischen Zuständen in der Zustandsmaschine wird durch die Auswertung von Booleschen Gleichungen geregelt, die aus Ausdrücken bestehen können, die Ladegleichungen, Register, Flags, Signale, Ausdrücke usw. umfassen. Jeder Zustand in der Zustandsmaschine ist zumindest einem Booleschen Ausdruck zugeordnet, und jeder dieser Booleschen Ausdrücke ist einem Zustand zugeordnet, zu dem die Zustandsmaschine transferieren könnte. Beispielsweise kann ein Zustand „A" einen ersten und einen zweiten Booleschen Ausdruck „A1" und „A2", die einem Transfer von dem Zustand „A" zu dem Zustand „B" zugeordnet sind, und einen dritten Booleschen Ausdruck „A3", der einem Transfer von dem Zustand „A" zu dem Zustand „C" zugeordnet ist, aufweisen. Die jedem Zustand zugeordneten Booleschen Ausdrücke werden priorisiert. Die einem aktuellen Zustand (d.h. einer aktuellen Position der Zustandsmaschine) zugeordneten Booleschen Ausdrücke werden ausgewertet. Auf eine Auswertung hin wird die aktuelle Position der Zustandsmaschine an einen Zustand transferiert, der dem Booleschen Ausdruck bzw. den Booleschen Ausdrücken mit der höchsten Priorität, die als wahr ermittelt wird, zugeordnet ist.

[0045] Jeder der Zustände **602**–**614** ist einer allgemeinen Funktion zugeordnet. Im IDLE-Zustand **602** wartet der Dekomprimierungseinrichtungskern **504** auf einen neuen Befehl. Der DECODE_COMMAND-Zustand **604** betrachtet den aktuellen Befehl. Der WRITE_SRC_INI-Zustand (Schreibe-Keimreihenkopier-Anfangszustand) **606** schreibt die Anzahl von Anfangskeimreihenkopiepixeln (bei einem möglichen Ausführungsbeispiel 1, 2 oder 3), die in den beiden Keimreihenzählwertbits des Befehls festgelegt sind, aus. Der WRITE_SRC_SUB-Zustand (Schreibe-Keimreihenkopie-Folgezustand) **608** schreibt Folgekeimreihenkopiepixel aus, die in Keimreihenkopiezählwertbytes, die auf einen Befehl folgen, festgelegt sind. Ein Zählwertbyte von 0xFF entspricht 255 Pixeln. Das letzte Zählwertbyte muß ein End-0x00- oder ein Nicht-0xFF-Zählwert sein. Der WAIT_FOR_FP-Zustand (Warte-Auf-Erstes-Pixel-Zustand) **610** wartet darauf, daß das erste Pixel eines Laufs oder Literals zur Verfügung steht. Der WRITE_REP_INI-Zustand (Schreibe-Ersetzung-Anfangszustand) **612** schreibt Literal- oder Laufpixel aus, die in den Ersetzungszählwertbits des Befehls festgelegt sind. Der WRITE_REP_SUB-Zustand (Schreibe-Ersetzung-Folgezustand) **614** schreibt Literal- oder Laufpixel aus, die in den Folge-Ersetzungszählwertbytes festgelegt sind. Wie das Keimreihenkopiezählwertbyte entspricht ein Ersetzungszählwertbyte von 0xFF 255 Pixeln. Das letzte Zählwertbyte muß ein End-0x00- oder ein Nicht-0xFF-Zählwert sein.

[0046] Der IDLE-Zustand **602** umfaßt eine Sequenz priorisierter Gleichungen, die zwei Boolesche Glei-

chungen umfassen. Wenn die erste Gleichung **602a**, input_available = 1 (verfügbarer_Eingang = 1), wahr ist, ändert sich der aktuelle Zustand der Zustandsmaschine **600** zu dem DECODE_COMMAND-Zustand **604**. Wenn die erste Gleichung nicht wahr ist, ist die zweite Gleichung **602B**, „sonst" oder „wahr" immer wahr, und der aktuelle Zustand verbleibt in dem IDLE-Zustand **602**.

[0047] Der DECODE_COMMAND-Zustand **604** umfaßt eine Sequenz priorisierter Gleichungen von drei Booleschen Gleichungen. Wenn die erste Gleichung **604A**, src_ini! = 0, wahr ist, ändert sich der aktuelle Zustand der Zustandsmaschine **600** zu dem WRITE_SRC_INI-Zustand **606**. Wenn die erste Gleichung nicht wahr ist und die zweite Gleichung **604B**, first_pixel_loaded = 1 (erstes_Pixel_geladen = 1) wahr ist, ändert sich der aktuelle Zustand der Zustandsmaschine **600** zu dem WRITE_REP_INI-Zustand **612**. Wenn höherrangige Gleichungen nicht wahr sind, ist die dritte Gleichung **604C**, „sonst" oder „wahr", immer wahr, und der aktuelle Zustand ändert sich zu dem WAIT_FOR_FP-Zustand **610**.

[0048] Der WRITE_SRC_INI-Zustand (Schreibe-Keimreihenkopier-Anfangszustand) **606** umfaßt eine Sequenz priorisierter Gleichungen, die zehn Booolesche Gleichungen umfassen. Wenn die erste Gleichung **606A**, decomp_eoi_b = 1 (dekomprimiere EndederZeile_b = 1) wahr ist, ändert sich der aktuelle Zustand der Zustandsmaschine **600** zu dem IDLE-Zustand **602**. Wenn die erste Gleichung nicht wahr ist und die zweite Gleichung **606B**, code_src_done = 1 (Code_Krk_erledigt = 1 [Krk = Keimreihenkopie]) & decomp_eol_b = 1 (Dekomp_EdZ_b = 1 [EdZ = Ende der Zeile]) und cmd_ld_src = 1 (Befehl_laden_Krk = 1), wahr ist, verbleibt der aktuelle Zustand der Zustandsmaschine **600** in dem WRITE_SRC_INI-Zustand **606**. Wenn höherrangige Gleichungen nicht wahr sind und die dritte Gleichung **606C**, code_src_done = 1 & decomp_eol_b = 1 & loading_command = 1 (Befehl_laden = 1) & cmd_ld_src = 0 & (first_pixel_loaded = 1 (erstes_Pixel_geladen = 1) oder loading_first_pixel = 1 (erstes_Pixel_laden = 1)) wahr ist, ändert sich der aktuelle Zustand der Zustandsmaschine **600** zu dem WRITE_REP_INI-Zustand **612**. Wenn höherrangige Gleichungen nicht wahr sind und die vierte Gleichung **606D**, code_src_done = 1 & decomp_eol_b = 1 & loading_command = 1 & cmd_ld_src = 0 & decomp_eoi_b = 0 (Dekomp_EdB_b = 1 [EdB = Ende des Bildes]), wahr ist, ändert sich der aktuelle Zustand der Zustandsmaschine **600** zu dem WAIT_FOR_FP-Zustand **610**. Wenn höherrangige Gleichungen nicht wahr sind und die fünfte Gleichung **606E**, code_src_done = 1 und decomp_eol_b = 1, wahr ist, ändert sich der aktuelle Zustand der Zustandsmaschine **600** zu dem IDLE-Zustand **602**. Wenn höherrangige Gleichungen nicht wahr sind und die sechste Gleichung **606F**, src_ini! = "11" (Krk_Anfang! = "11") & code_src_done = 1 & (first_pixel_loaded = 1 oder loading_first_pixel = 1) wahr ist, ändert sich der aktuelle Zustand der Zustandsmaschine **600** zu dem WRITE_REP_INI-Zustand **612**. Wenn höherrangige Gleichungen nicht wahr sind und die siebte Gleichung **606G**, src_ini! = "11" & code_src_done = 1 oder (block_00_src = 1 oder block_00_src_d = 1) & (first_pixel_loaded = 0 oder loading_first_pixel = 0) wahr ist, ändert sich der aktuelle Zustand der Zustandsmaschine **600** zu dem WAIT_FOR_FP-Zustand **610**. Wenn höherrangige Gleichungen nicht wahr sind und die achte Gleichung **606H**, src_ini = "11" & code_src_done = 1 & (block_00_src = 1 oder block_00_src_d = 1) & (first_pixel_loaded = 1 oder loading_first_pixel = 1) wahr ist, ändert sich der aktuelle Zustand der Zustandsmaschine **600** zu dem WRITE_REP_INI-Zustand **612**. Wenn höherrangige Gleichungen nicht wahr sind und die neunte Gleichung **606I**, src_ini = 3 & code_src_done = 1 & (src_sub_loaded = 1 (KrK_Folge_geladen = 1) oder ld_src_subsequent = 1 (laden-Krk_Folge = 1)) wahr ist, ändert sich der aktuelle Zustand der Zustandsmaschine **600** zu dem WRITE_SRC_SUB-Zustand **608**. Wenn höherrangige Gleichungen nicht wahr sind, ist die zehnte Gleichung **606J**, „sonst" oder „wahr", immer wahr, und der aktuelle Zustand verbleibt in dem WRITE_SRC_INI-Zustand **606**.

[0049] Der WRITE_SRC_SUB-Zustand **608** umfaßt eine Sequenz priorisierter Gleichungen, die neun Boolesche Gleichungen umfaßt. Wenn die erste Gleichung **608A**, decomp_eoi_b = 1, wahr ist, ändert sich der aktuelle Zustand der Zustandsmaschine **600** zu dem IDLE-Zustand **602**. Wenn die erste Gleichung nicht wahr ist und die zweite Gleichung **608B**, code_src_done = 1 & decomp_eol_b = 1 & ((command_loaded = 1 & next_src_ini! = 0 (Nächstes_Krk_Anfang! = 0)) oder (loading_command = 1 & cmd_ld_src = 1)), wahr ist, ändert sich der aktuelle Zustand der Zustandsmaschine **600** zu dem WRITE_SRC INI-Zustand **606**. Wenn höherrangige Gleichungen nicht wahr sind und die dritte Gleichung **608C**, code_src_done = 1 & decomp_eol_b = 1 & loading_command = 1 & cmd_ld_src = 0 & (first_pixel_loaded = 1 oder loading_first_pixel = 1), wahr ist, ändert sich der aktuelle Zustand der Zustandsmaschine **600** zu dem WRITE_REP_INI-Zustand **612**. Wenn höherrangige Gleichungen nicht wahr sind und die vierte Gleichung **608D**, code_src_done = 1 & decomp_eol_b = 1 & loading_command = 1 & cmd_ld_src = 0, wahr ist, ändert sich der aktuelle Zustand der Zustandsmaschine **600** zu dem WAIT_FOR_FP-Zustand **610**. Wenn höherrangige Gleichungen nicht wahr sind und die fünfte Gleichung **608E**, code_src_done = 1 & decomp_eol_b = 1, wahr ist, ändert sich der aktuelle Zustand der Zustandsmaschine **600** zu dem IDLE-Zustand **602**. Wenn höherrangige Gleichungen

nicht wahr sind und die sechste Gleichung **608F**, src_orig! = 255 & code_src_done = 1 & (first_pixel_loaded = 1 oder loading_first_pixel = 1), wahr ist, ändert sich der aktuelle Zustand der Zustandsmaschine **600** zu dem WRITE_REP_INI-Zustand **612**. Wenn höherrangige Gleichungen nicht wahr sind und die siebte Gleichung **608G**, src_orig! = 255 & code_src_done = 1 oder ((block_00_src = 1 oder block_00_src_d = 1) & (first_pixel_loaded = 0 & loading_first_pixel = 0)), wahr ist, ändert sich der aktuelle Zustand der Zustandsmaschine **600** zu dem WAIT_FOR_FP-Zustand **610**. Wenn höherrangige Gleichungen nicht wahr sind und die achte Gleichung **608H**, src_orig = 255 & code_src_done = 1 & ((block_00_src = 1 oder block_00_src_d = 1) & (first_pixel_loaded = 1 & loading_first_pixel = 1)), wahr ist, ändert sich der aktuelle Zustand der Zustandsmaschine **600** zu dem WRITE_REP_INI-Zustand **612**. Wenn höherrangige Gleichungen nicht wahr sind, ist die neunte Gleichung **608I**, „sonst" oder „wahr", immer wahr, und der aktuelle Zustand verbleibt in dem WRITE_SRC_SUB-Zustand **608**.

[0050] Der WAIT_FOR_FP-Zustand **610** umfaßt eine Sequenz priorisierter Gleichungen, die zwei Boolesche Gleichungen umfaßt. Wenn die erste Gleichung **610A**, first_pixel_loaded = 1 oder loading_first_pixel = 1, wahr ist, ändert sich der aktuelle Zustand der Zustandsmaschine **600** zu dem WRITE_REP_INI-Zustand **612**. Wenn höherrangige Gleichungen und die erste Gleichung nicht wahr sind, ist die zweite Gleichung **610B**, „sonst" oder „wahr", immer wahr, und der aktuelle Zustand verbleibt in dem WAIT_FOR_FP-Zustand **610**.

[0051] Der WRITE_REP_INI-Zustand **612** umfaßt eine Sequenz priorisierter Gleichungen, die vierzehn Boolesche Gleichungen umfaßt. Wenn die erste Gleichung **612A**, decomp_eoi_b = 1, wahr ist, ändert sich der aktuelle Zustand der Zustandsmaschine **600** zu dem IDLE-Zustand **602**. Wenn die erste Gleichung nicht wahr ist und die zweite Gleichung **612B**, rep_orig = 1 & code_rep_done = 1 & ((ld_imm_cmd = 1 (lade_imm_Befehl = 1) und current_byte(4:3)! = "00" (aktuelles_Byte(4:3)! = "00")) oder (ld_next_imm_cmd = 1 & byte1_ahead(4:3)! = "00" (byte1_voraus(4:3)! = "00"))), wahr ist, ändert sich der aktuelle Zustand der Zustandsmaschine **600** zu dem WRITE_SRC_INI-Zustand **606**. Wenn höherrangige Gleichungen nicht wahr sind und die dritte Gleichung **612C**, rep_orig = 1 & code_rep_done = 1 & (ld_imm_cmd = 1 oder ld_next_imm_cmd = 1) & loading_first_pixel = 1 & decomp_eoi_b = 0, wahr ist, verbleibt der aktuelle Zustand der Zustandsmaschine **600** in dem WRITE_REP_INI-Zustand **612**. Wenn höherrangige Gleichungen nicht wahr sind und die vierte Gleichung **612D**, rep_orig = 1 & code_rep_done = 1 & (ld_imm_cmd = 1 oder ld_next_imm_cmd = 1) & loading_first_pixel = 0 & decomp_eoi_b = 0, wahr ist, ändert sich der aktuelle Zustand der Zustandsmaschine **600** zu dem WAIT_FOR_FP-Zustand **610**. Wenn höherrangige Gleichungen nicht wahr sind und die fünfte Gleichung **612E**, ((rep_orig! = 8 & run = 0) oder (rep_orig! = 9 & run = 1)) & code_rep_done = 1 & cmd_ld_src = 1 & loading_command = 1 & decomp_eoi_b = 0, wahr ist, ändert sich der aktuelle Zustand der Zustandsmaschine **600** zu dem WRITE_SRC_INI-Zustand **606**. Wenn höherrangige Gleichungen nicht wahr sind und die sechste Gleichung **612F**, ((rep_orig! = 8 & run = 0) oder (rep_orig! = 9 & run = 1)) & code_rep_done = 1 & cmd_ld_src = 0 & loading_command = 1 & (first_pixel_loaded = 1 oder loading_first_pixel = 1) & decomp_eoi_b = 0, wahr ist, verbleibt der aktuelle Zustand der Zustandsmaschine **600** in dem WRITE_REP_INI-Zustand **612**. Wenn höherrangige Gleichungen nicht wahr sind und die siebte Gleichung **612G**, ((rep_orig! = 8 & run = 0) oder (rep_orig! = 9 & run = 1)) & code_rep_done = 1 & loading_command = 1 & cmd_ld_src = 0 & decomp_eoi_b = 0, wahr ist, ändert sich der aktuelle Zustand der Zustandsmaschine **600** zu dem WAIT_FOR_FP-Zustand **610**. Wenn höherrangige Gleichungen nicht wahr sind und die achte Gleichung **612H**, ((rep_orig! = 8 & run = 0) oder (rep_orig! = 9 & run = 1)) & code_rep_done = 1, wahr ist, ändert sich der aktuelle Zustand der Zustandsmaschine **600** zu dem IDLE-Zustand **602**. Wenn höherrangige Gleichungen nicht wahr sind und die neunte Gleichung **612I**, ((rep_orig = 8 & run = 0) oder (rep_orig = 9 & run = 1)) & code_rep_done = 1 & block_00_rep = 0 & (rep_sub_loaded = 1 oder ld_rep_subsqnt = 1), wahr ist, ändert sich der aktuelle Zustand der Zustandsmaschine **600** zu dem WRITE_REP_SUB-Zustand **614**. Wenn höherrangige Gleichungen nicht wahr sind und die zehnte Gleichung **612J**, ((rep_orig = 8 & run = 0) oder (rep_orig = 9 & run = 1)) & code_rep_done = 1 & block_00_rep = 1 & loading_command = 1 & cmd_ld_src = 1, wahr ist, ändert sich der aktuelle Zustand der Zustandsmaschine **600** zu dem WRITE_SRC_INI-Zustand **606**. Wenn höherrangige Gleichungen nicht wahr sind und die elfte Gleichung **612K**, ((rep_orig = 8 & run = 0) oder (rep_orig = 9 & run = 1)) & code_rep_done = 1 & block_00_rep = 1 & loading_command = 1 & cmd_ld_src = 0 & (first_pixel_loaded = 1 oder loading_first_pixel = 1), wahr ist, verbleibt der aktuelle Zustand der Zustandsmaschine **600** in dem WRITE_REP_INI-Zustand **612**. Wenn höherrangige Gleichungen nicht wahr sind und die zwölfte Gleichung **612L**, ((rep_orig = 8 & run = 0) oder (rep_orig = 9 & run = 1)) & code_rep_done = 1 & block_00_rep = 1 & loading_command = 1 & cmd_ld_src = 0 & (first_pixel_loaded = 0 & loading_first_pixel = 0), wahr ist, ändert sich der aktuelle Zustand der Zustandsmaschine **600** zu dem WAIT_FOR_FP-Zustand **610**. Wenn höherrangige Gleichungen nicht wahr sind und die dreizehnte Gleichung **612M**, ((rep_orig = 8 & run = 0) oder (rep_orig = 9 & run = 1)) & code_rep_done = 1 & block_00_rep = 1, wahr ist, ändert sich der aktuelle Zustand der Zustandsma-

schine **600** zu dem IDLE-Zustand **602**. Wenn höherrangige Gleichungen nicht wahr sind, ist die vierzehnte Gleichung **612N**, „sonst" oder „wahr", immer wahr, und der aktuelle Zustand verbleibt in dem WRITE_REP_INI-Zustand **612**.

[0052] Der WRITE_REP_SUB-Zustand **614** umfaßt eine Sequenz priorisierter Gleichungen, die zehn Boolesche Gleichungen umfaßt. Wenn die erste Gleichung, decomp_eoi_b = 1, wahr ist, ändert sich der aktuelle Zustand der Zustandsmaschine **600** zu dem IDLE-Zustand **602**. Wenn die erste Gleichung **614A** nicht wahr ist und die zweite Gleichung **614B**, rep_orig! = 255 & code_rep_done = 1 & cmd_ld_src = 1, wahr ist, ändert sich der aktuelle Zustand der Zustandsmaschine **600** zu dem WRITE_SRC_INI-Zustand **606**. Wenn höherrangige Gleichungen nicht wahr sind und die dritte Gleichung **614C**, rep_orig! = 255 & code_rep_done = 1 & cmd_ld_src = 0 &% loading_command = 1 & (first_pixel_loaded = 1 oder loading_first_pixel = 1), wahr ist, ändert sich der aktuelle Zustand der Zustandsmaschine **600** zu dem WRITE_REP_INI-Zustand **612**. Wenn höherrangige Gleichungen nicht wahr sind und die vierte Gleichung **614D**, rep_orig! = 255 & code_rep_done = 1 & cmd_ld_src = 0 & loading_command = 1, wahr ist, ändert sich der aktuelle Zustand der Zustandsmaschine **600** zu dem WAIT_FOR_FP-Zustand **610**. Wenn höherrangige Gleichungen nicht wahr sind und die fünfte Gleichung **614E**, rep_orig! = 255 & code_rep_done = 1, wahr ist, ändert sich der aktuelle Zustand der Zustandsmaschine **600** zu dem IDLE-Zustand **602**. Wenn höherrangige Gleichungen nicht wahr sind und die sechste Gleichung **614F**, rep_orig = 255 & code_rep_done & cmd_ld_src = 1 & loading_command = 1, wahr ist, ändert sich der aktuelle Zustand der Zustandsmaschine **600** zu dem WRITE_SRC_INI-Zustand **606**. Wenn höherrangige Gleichungen nicht wahr sind und die siebte Gleichung **614G**, rep_orig = 255 & code_rep_done = 1 & cmd_ld_src = 0 & loading_command = 1 & (first_pixel_loaded = 1 oder loading_first_pixel = 1), wahr ist, ändert sich der aktuelle Zustand der Zustandsmaschine **600** zu dem WRITE_REP_INI-Zustand **612**. Wenn höherrangige Gleichungen nicht wahr sind und die achte Gleichung **614H**, rep_orig = 255 & code_rep_done = 1 & cmd_ld_src = 0 & loading_command = 1 & (first_pixel_loaded = 0 & loading_first_pixel = 0), wahr ist, ändert sich der aktuelle Zustand der Zustandsmaschine **600** zu dem WAIT_FOR_FP-Zustand **610**. Wenn höherrangige Gleichungen nicht wahr sind und die neunte Gleichung **614I**, rep_orig = 255 & code_rep_done = 1 & block_00_rep = 1, wahr ist, ändert sich der aktuelle Zustand der Zustandsmaschine **600** zu dem IDLE-Zustand **602**. Wenn höherrangige Gleichungen nicht wahr sind, ist die zehnte Gleichung **614J**, „sonst" oder „wahr" immer wahr, und der aktuelle Zustand verbleibt in dem _REP_SUB-Zustand **614**.

[0053] Fig. 7 zeigt ein Codierungsformat für komprimierte Rasterpixeldaten RPD. Daten **700** bestehen aus einem Befehlsbyte **702** und optionalen Keimreihenversatzwertfeldern, Ersetzungszählwert-Wertfeldern **706** und Farbdaten. Ersetzungspixeldaten ersetzen die rohen RPD mit einer Ersetzungsdatenkette, die hierin auch als die „Relativwertkette" bezeichnet wird, die aus einem Befehl, Farbdaten besteht; optionale Keimreihenversatzwert- und Ersetzungszählwert-Wertfelder werden nach Bedarf bereitgestellt. Eine Serie von Ersetzungsketten beschreibt ein Raster. Das Befehlsbyte **702** weist vier Segmente auf: (1) CMD-Bit (CMD = command = Befehl) **710**; (2) Pixelquellenbit **712**; (3) Keimreihenzählwert **714**; und (4) Ersetzungszählwert **716**, wobei das CMD-Bit **710** für eine Ersetzungspixellisten-Datengruppe (RPL-Datengruppe, RPL = replacement pixel list) auf Null gesetzt ist und für eine Ersetzungspixellauf-Datengruppe (RPR-Datengruppe, RPR = replacement pixel run) auf eins gesetzt ist.

[0054] Das Pixelquellenbit **712** gibt die Farbe des Komprimierungslaufs an. Bei einer exemplarischen Implementierung impliziert 0 eine neue Farbe (es wird kein Cache-Speichern verwendet); 1 impliziert eine Verwendung einer westlichen Farbe (d.h. vorhergehende Spalte, selbe Reihe); 2 impliziert eine Verwendung einer nordöstlichen Farbe (d.h. darunterliegende Reihe, folgende Spalte); und 3 impliziert eine Verwendung einer Cache-gespeicherten Farbe. In diesem Fall geben RPR-Gruppen-Pixelquellenbits die Farbe für einen gesamten Komprimierungslauf an, wenn lediglich eine Farbe festgelegt ist. Bei einem RPL-Daten-Komprimierungslauf gibt das Pixelquellenbit **712** die Farbe für lediglich das Anfangspixel in dem Lauf an, und verbleibende Pixel sind in dem Datenbytes-Feld **708**, das Rot-, Grün- und Blau-ABS-Bytes (ABS = absolut) **718**, **720**, **722** (d.h. Bytes, die Pixel darstellen, die direkt passiert werden, ohne Codierung oder Mehraufwand) umfaßt, codiert.

[0055] Der Keimreihenzählwert **714** ist die Anzahl von Pixeln, die von der Keimreihe zu kopieren sind; und der Ersetzungszählwert **716** ist die Anzahl von aufeinanderfolgenden Pixeln, die ersetzt werden sollen (bei einem Ausführungsbeispiel ist dies für RPL-Datentransfers eines weniger als die tatsächliche Anzahl (z.B. eine Ersetzung von sechs Pixeln ist mit einem Zählwert von fünf festgelegt); und für RPR-Transfers ist dies zwei weniger als die tatsächliche Anzahl).

[0056] Falls der Keimreihenzählwert **714** des Befehlsbytes größer ist als zwei, werden in dem optionalen Pixelbytes-704-Feld der Ersetzungsdatenkette zusätzliche Versatzwerte gesetzt, die zu dem Gesamtkeimreihenzählwert **714** hinzugefügt werden. Dies geht vonstatten, bis das letzte Keimreihenzählwertbyte durch einen Wert, der geringer ist als zwei-fünf-fünf (255), angegeben wird. Falls der Erset-

zungszählwert **716** in dem Befehlsbyte **710** größer ist als sechs, werden zusätzliche Werte in dem optionalen Ersetzungszählwertbytes-Feld **706** der Ersetzungsdatenkette gesetzt, die zu dem Gesamtersetzungszählwert hinzugefügt werden. Dies geht vonstatten, bis der letzte Ersetzungszählwert-Wert durch einen Wert, der geringer ist als zwei-fünf-fünf (255), angegeben wird.

[0057] Falls das CMD-Bit **710** „0" ist, sind die Ersetzungsdaten eine RPL-codierte Kette. Die Anzahl von Pixeln, die unmittelbar auf einen Befehl und seine optionalen Bytes folgen, ist der Ersetzungszählwert plus 1. Eine Ausnahme liegt vor, wenn eine Pixelquelle keine neue Farbe für das erste Pixel angibt. In diesem Fall ist die Anzahl von Pixeln, die auf den Befehl und seine optionalen Bytes folgen, der Ersetzungszählwert.

[0058] Falls das CMD-Bit „1" ist, sind die Ersetzungsdaten eine RPR-codierte Kette; es können optionale Keimreihenzählwertbytes und Ersetzungszählwertbytes hinzugefügt werden. Falls das Pixelquellenbit eine neue Farbe angibt, wird es in dem Datenbytes-Feld codiert. Andernfalls wird die Farbe des RPR von einer alternativen Quelle erhalten, wie nachfolgend erläutert wird, und es werden keine Datenbytes vorliegen. Genauso wie im Fall der RPL-Daten wird ein codiertes Pixel die absolute Form aufweisen. Die Länge des Laufs in Pixeln ist der Wert des Ersetzungszählwerts **716** plus 2.

[0059] Das Flußdiagramm der Fig. 8 veranschaulicht eine weitere exemplarische Implementierung, bei der ein Verfahren **800** verwendet wird, um Daten zu komprimieren. Die Elemente des Verfahrens können durch ein beliebiges gewünschtes Mittel bewerkstelligt werden, beispielsweise durch die Verwendung einer ASIC oder durch die Ausführung von prozessorlesbaren Anweisungen, die auf einem prozessorlesbaren Medium, beispielsweise einer Platte, einem ROM oder einer anderen Speichervorrichtung, definiert sind. Ferner können Aktionen, die in einem beliebigen Block beschrieben sind, parallel zu Aktionen, die in anderen Blöcken beschrieben sind, durchgeführt werden, können in einer alternativen Reihenfolge stattfinden oder können auf eine Weise, die Aktionen mehr als einem weiteren Block zuweist, verteilt sein.

[0060] Bei Block **802** verschachtelt eine Datenverschachtelungseinrichtung **404** die Farbebenendaten. Die Daten können von einer beliebigen Quelle ankommen, beispielsweise einem Interpretierer, der eine Seitenbeschreibungssprache in Vorrichtung-Bereit-Bits übersetzt.

[0061] Bei Block **804** bestimmt ein Eingangsdateninterpretierer **406** die zu verwendende angemessene Komprimierung, beispielsweise Läufe, Keimreihenkopien und Literale. Dementsprechend werden die Daten in der Alternative an eine Laufbefehlseinheit **426**, eine Keimreihenkopieeinheit **428** oder eine Literalbefehlseinheit **430** gesendet.

[0062] Bei Block **806** wird eine Entropie gemessen, um einen Indikator der Effektivität der Komprimierung zu liefern. Die Messung wird durch ein Zählen von Änderungen zwischen sequentiellen Pixeln durchgeführt und kann durch ein Entropiemodul **408**–**412**, wie es beispielsweise in Fig. 4 zu sehen ist, durchgeführt werden. Die Änderungen in den sequentiellen Pixeln stellen Änderungen des numerischen Wertes einer oder mehrerer Farbebenen, die zwischen zwei benachbarten Pixeln zugeordnet sind, dar. Wenn sich die zwei benachbarten Pixeln zugeordneten Daten in einer oder mehreren Farbebenen geändert haben, wird diese Änderung durch das zugeordnete Entropiemodul **408**–**412** gezählt und verarbeitet.

[0063] Bei Block **808** wird ein Neusortierungspuffer **424** verwendet, um bis zu acht Befehle zu speichern, und er ermöglicht, daß die Keimreihenkopie-, Lauf- und Literaleinheiten **426**, **428**, **430** gleichzeitig arbeiten.

[0064] Bei Block **810** verwendet die Literalbefehlseinheit **430** eine Speichervorrichtung **432**, um ganze 255 Bytes von Literalen (oder eine ähnliche Datenmenge) gleichzeitig zu puffern.

[0065] Bei Block **812** verfolgt der Neusortierungspuffer **424**, welcher Einheit, einschließlich der Laufbefehlseinheit **426**, einer Keimreihenkopieeinheit **428** oder einer Literalbefehlseinheit **430**, ermöglicht werden sollte, Daten in das Komprimierungsausgangsmodul **434** zu schreiben, wodurch die Ausgabe jeder Einheit in der richtigen Reihenfolge sequenziert wird.

[0066] Bei Block **814** gibt ein Komprimierungsausgangsmodul **434** komprimierte Daten aus, die gespeichert oder übertragen werden, bis sie für eine Dekomprimierung benötigt werden.

[0067] Das Flußdiagramm der Fig. 9 veranschaulicht eine weitere exemplarische Implementierung, bei der ein Verfahren **900** verwendet wird, um Daten zu dekomprimieren. Die Elemente des Verfahrens können durch ein beliebiges gewünschtes Mittel bewerkstelligt werden, beispielsweise durch die Verwendung einer ASIC oder durch die Ausführung von prozessorlesbaren Anweisungen, die auf einem prozessorlesbaren Medium, beispielsweise einer Platte, einem ROM oder einer anderen Speichervorrichtung, definiert sind. Ferner können Aktionen, die in einem beliebigen Block beschrieben sind, parallel zu Aktionen, die in anderen Blöcken beschrieben sind, durchgeführt werden, können in einer alternativen Reihenfolge stattfinden oder können auf eine Weise, die Aktionen mehr als einem weiteren Block zuweist, verteilt

sein.

[0068] Bei Block **902** werden einer Mehrzahl von Ladegleichungen Vorgriffsdaten bereitgestellt. Während beispielsweise zwei 4-Bytes-Puffer verwendet werden können, um den Ladegleichungen Vorgriffsdaten bereitzustellen, könnten auch andere Pufferkonfigurationen und -größen verwendet werden.

[0069] Bei Block **904** können die Ladegleichungen asynchron und parallel betrieben werden, um Daten (einschließlich aktueller Pixeldaten und Vorgriffsdaten) zu prüfen, wobei nach Instanzen von Keimreihenkopien, Läufen und Literalen Ausschau gehalten wird.

[0070] Bei Block **906** werden Register auf der Basis der Ladegleichungen geladen. Die Register umfassen Statusindikatoren, Zeiger und andere Daten, was den Decodiererbetrieb erleichtert.

[0071] Bei Block **908** werden die kombinatorischen Gleichungen verwendet, um einen Datenfluß in Zeilenpuffer, einen DMA und eine in Flußrichtung nachgelagerte Vorrichtung (beispielsweise eine Druckmaschine) hinein und aus denselben heraus zu steuern. Manche der kombinatorischen Gleichungen können Boolesche Gleichungen umfassen, wobei die Ladegleichungen und Werte der Register Elemente in den Gleichungen sind.

[0072] Bei Block **910** werden Boolesche Gleichungen aus Ladegleichungen, Registerwerten, Signalen (Übernahmesignalen; Strobes) und/oder kombinatorischen Gleichungen gebildet.

[0073] Bei Block **912** wird ein Betrieb der Dekomprimierungseinrichtung **318** durch eine Zustandsmaschine geregelt. Eine Bewegung zwischen Zuständen in der Zustandsmaschine wird durch eine Priorisierung und Auswertung der Booleschen Gleichungen gesteuert. Eine Priorisierung der Booleschen Gleichungen gibt die korrekte Zustandsänderung an, wenn mehr als eine Zustandsänderung durch eine Auswertung einer Booleschen Gleichung angegeben ist. Wenn zum Beispiel eine Bewegung vom Zustand 4 zu beiden Zuständen 2 und 7 durch die Wahrheit der beteiligten Booleschen Gleichungen angegeben wird, kann eine Priorität einer Booleschen Gleichung angeben, daß sich der Zustand zu dem Zustand 7 ändern sollte.

[0074] Bei Block **914** werden im Lauf des Betriebs der Dekomprimierungseinrichtung **318** bei jedem Taktzyklus Daten, die einem Pixel zugeordnet sind, eingegeben und ausgegeben.

[0075] Eine mögliche Implementierung verschiedener Ausdrücke, Gleichungen, Register, Daten, Variablen, Signale, Befehle und anderer Logik folgt:

Ladegleichungsausdrücke:

– ld_next_cmd_lit: Dieser Ausdruck tritt auf, wenn der aktuelle Befehl kein Lauf ist (der ein Literal angibt). Er sorgt dafür, daß eine Nachbildung im Gange ist (rep_in_progress = 1) (Nachb_im_Gange = 1), daß das aktuelle Byte, das gerade transferiert wird, das letzte Byte ist (letztes Byte = 1) und daß es hinaustransferiert wird (decomp_xfer = 1) (Dekomp_Transfer = 1). Schließlich prüft er, daß das letzte Byte nicht das letzte Byte des Streifens ist (decomp_eoi = 0).
– ld_next_cmd_run: Dieser Ausdruck tritt auf, wenn der aktuelle Befehl ein Lauf ist. Er sorgt dafür, daß eine Nachbildung im Gange ist (rep_in_progress = 1) oder daß eine Keimreihenkopie im Gange ist (src_in_progress = 1) (Krk_im_Gange = 1). Falls uns eine Keimreihenkopie zum Ende einer Zeile führt, kann sie als eigenständiger Befehl angesehen werden. In diesem Fall ist die Keimreihenkopie als ein Lauf codiert, da sie jedoch bis zum Ende der Zeile geht, findet keine tatsächliche Laufersetzung statt, bevor der nächste Befehl geladen ist. Wir prüfen, daß das aktuelle Byte, das gerade transferiert wird, das letzte Byte ist (letztes Byte = 1), daß es hinaustransferiert wird (decomp_xfer = 1) und daß das letzte Byte nicht das letzte Byte des Streifens ist (decomp_eoi = 0).
– ld_imm_cmd: Es gibt einen Fall, bei dem das Literal lediglich eine Länge von 1 Byte aufweist. Wenn wir nicht anhalten wollen, müssen wir in der Lage sein, unmittelbar den nächsten Befehlspuffer zu laden, obwohl für diesen Fall der Ausdruck command_loaded (Befehl_geladen) aufgrund des aktuellen Befehls immer noch wahr ist. Command_loaded wird nicht gelöscht, bis das erste Pixel für den aktuellen Befehl decodiert wurde. Dieser Eckfall gilt für einen 1-Literal-Befehl zu Beginn eines Transfersatzes. Dieser Ausdruck wird ebenfalls verwendet, wenn wir eine einzige Keimreihenkopie am Ende der Zeile aufweisen. Wiederum wird der obige Ausdruck ld_next_cmd_run nicht zum Tragen kommen, da der Ausdruck command_loaded immer noch wahr ist. Für den Fall eines Literals prüft der Ausdruck, daß der ursprüngliche Nachbildungszählwert 1 (rep_orig = 0x01) ist, daß eine Nachbildung im Gange ist und daß die Nachbildung ein Anfangsliteral (Lauf = 0 und rep_mux_ctl = 0) oder ein Folgelauf ist. Wir führen eine Prüfung hinsichtlich des Folgelaufs durch, für den Fall, daß wir auf einen Folgelauf-Zählwert von 1 stoßen. Falls dies geschieht, müssen wir den letzten Laufwert ausgeben und unmittelbar den nächsten Befehl laden. Für den Fall einer Keimreihenkopie führt der Ausdruck eine Prüfung hinsichtlich einer ursprünglichen Keimreihenkopie von 1 (src_orig = 1) durch, prüft, daß eine Keimreihenkopie im Gange ist, daß dies eine Anfangskeimreihenkopie ist

(src_mux_ctl = 0) und daß Ende der Zeile wahr ist (decomp_eol_b = 1). Für jeden beliebigen Fall mit Ausnahme einer Anfangskeimreihenkopie von 1 sind andere Ausdrücke vorhanden, um das Laden des nächsten Befehls zu erleichtern. Für jeglichen Fall mit Ausnahme dessen, daß Ende der Zeile wahr ist, ist die Keimreihenkopie nicht die letzte Aktion des aktuellen Befehls. Für jeden dieser Fälle wird schließlich eine Prüfung durchgeführt, um zu gewährleisten, daß das aktuelle Pixel das letzte Pixel des Streifens ist, daß wir das einzige Pixel hinaustransferieren und daß der nächste Befehl zur Verfügung steht. In diesem Fall prüfen wir hinsichtlich input_available = 1 (verfügbare_Eingabe = 1), was uns sagt, ob current_pointer (aktueller_Zeiger) auf ein gültiges Byte zeigt. In diesem Fall zeigt current_pointer auf den nächsten Befehl, da der aktuelle Befehl definitiv keine Folgebytes aufweist.

– ld_next_imm_cmd: Dieser Ausdruck bezieht sich auf den Fall, bei dem das Literal einer Länge 1 an dem Ende eines langen Literallaufs (aufgrund einer Folgelänge von eins) stattfindet, wobei das nächste Byte zum nächsten Befehl wird. Deshalb halten wir nach byte1_ahead_avail (Byte1_voraus_verfügbar) Ausschau statt nach current_pointer. Wir prüfen ferner, daß die Nachbildungslänge 1 ist und daß wir ein Literal (Lauf = 0) durchführen und daß wir uns in einem Folgenachbildungsmodus (rep_mux_ctl = 1) befinden. Wir stellen sicher, daß das aktuelle Byte nicht das letzte Byte des Streifens ist und daß wir aktuell das letzte Byte transferieren.

– inc_next_ptr_1: Wenn wir entweder durch ld_next_cmd_run oder durch ld_imm_cmd den nächsten Befehl laden, sollten wir den aktuellen Zeiger um 1 Byte inkrementieren. Wir tun dies, weil, wenn ld_next_cmd_run oder ld_imm_cmd wahr ist, der current_pointer auf das nächste Befehlsbyte zeigt. Wenn wir also den nächsten Befehl laden, inkrementieren wir einfach den Zeiger um eins, so daß er nun auf das nächste Byte hinter dem Befehl zeigt.

– ld_next_fb_lit: Wenn wir eine Literalersetzung durchführen und den nächsten Befehl laden, und der nächste Befehl ein erstes Pixel erfordert, aktivieren wir dieses Übernahmesignal. Neben einem Prüfen, daß der aktuelle Befehl ein Literal ist und daß das aktuelle Byte das letzte Byte ist, prüfen wir ferner, daß das byte2_ahead in dem komprimierten Datenstrom in dem Eingangspuffer vorliegt (byte2_ahead_avail = 1). Da wir uns in einer Literalersetzung befinden, zeigt der current_pointer auf das letzte Byte der Ersetzung. Current_pointer plus 1 zeigt auf den nächsten Befehl, und der aktuelle Zeiger plus 2 zeigt auf das erste Pixel (falls eines erforderlich ist). Um zu bestimmen, ob ein erstes Pixel erforderlich ist, stellen wir sicher, daß der nächste Befehl keine Keimreihenkopiezählwerte (byte1_ahead(4) = 0 und byte1_ahead(3) = 0) aufweist und daß die für das erste Pixel in dem Befehl dokumentierte Stelle 0x00 (byte1_ahead(6) = 0 und byte1_ahead(5) = 0) lautet. Ferner stellen wir sicher, daß das erste Pixel nicht schon zuvor geladen wurde (first_pixel_loaded = 0) und daß wir uns nicht dem Ende der Zeile nähern (eol_approach = 1 (eol_Annäherung = 1) oder eol_approach_d = 1).

– ld_next_fb_run: Wenn wir eine Laufnachbildung durchführen und den nächsten Befehl laden, und wenn der nächste Befehl ein erstes Pixel erfordert, aktivieren wir dieses Übernahmesignal. Neben einem Prüfen, daß der aktuelle Befehl ein Lauf ist und daß das aktuelle Byte das letzte Byte ist, prüfen wir ferner, daß das byte1_ahead in dem komprimierten Datenstrom in dem Eingangspuffer vorliegt (byte1_ahead_avail = 1). Da wir uns in einer Laufnachbildung befinden, zeigt der current_pointer auf den nächsten Befehl, und aktueller Zeiger plus 1 zeigt auf das erste Pixel (falls eines erforderlich ist). Um zu bestimmen, ob ein erstes Pixel erforderlich ist, stellen wir sicher, daß der nächste Befehl keine Keimreihenkopiezählwerte (current_byte(4) = 0 und current_byte(3) = 0) aufweist und daß die für das erste Pixel in dem Befehl dokumentierte Stelle 0x00 (current_byte(6) = 0 und current_byte(5) = 0) ist. Ferner stellen wir sicher, daß das erste Pixel nicht schon zuvor geladen wurde (first_pixel_loaded = 0) und daß wir uns nicht dem Ende der Zeile nähern (eol_approach = 1 oder eol_approach_d = 1). Schließlich stellen wir sicher, daß der Befehl nicht schon zuvor geladen wurde (falls dies geschieht, verwenden wir ein Ladeübernahmesignal in dem Keimreihenkopieabschnitt, nachfolgend beschrieben) und daß das aktuelle Pixel nicht das letzte Pixel des Streifens ist.

– ld_imm_fb: Dieses Ladeübernahmesignal gehört zu dem obigen ld_imm_cmd, das statt eines Literals einer Länge 1 oder einer Keimreihenkopie einer Länge 1 vorliegt. Neben einem Sicherstellen, daß wir uns entweder in einem Literal oder einer Keimreihenkopie einer Länge eins befinden (wie ausführlich für das obige ld_imm_cmd beschrieben wurde), stellen wir ferner sicher, daß byte1_ahead_avail wahr ist. Da der current_pointer auf den Befehl zeigt, zeigt current_pointer plus 1 auf das erste Byte. Wir stellen ferner sicher, daß wir das letzte Byte eines vorhergehenden Befehls transferieren, daß wir uns nicht am Streifenende befinden und daß dieser neue Befehl ein erstes Byte benötigt. Schließlich stellen wir sicher, daß nicht mehr als drei Keimreihenkopien durchzuführen sind. Man erinnere sich, daß, wenn mehr als drei Keimreihenkopien durchzuführen sind, das nächste Byte bzw. die nächsten Bytes Folgekeimreihenzählwerte und nicht das erste Pixel ist bzw. sind.

– ld_next_imm_fb: Dieses Ladeübernahmesignal gehört zu dem obigen ld_next_imm_cmd, das

statt eines Literals einer Länge 1 am Ende einer Folgeersetzung vorliegt. Neben einem Sicherstellen, daß wir uns in einer Literalersetzung befinden, stellen wir ferner sicher, daß byte2_ahead_avail wahr ist. Da der current_pointer auf das letzte Pixel zeigt, zeigt current_pointer plus 1 auf den nächsten Befehl und zeigt current_pointer plus 2 auf das erste Pixel. Wir stellen ferner sicher, daß wir das letzte Byte eines vorhergehenden Befehls transferieren, daß wir uns nicht am Streifenende befinden und daß dieser neue Befehl ein erstes Byte benötigt. Schließlich stellen wir sicher, daß nicht mehr als drei Keimreihenkopien durchzuführen sind. Man erinnere sich, daß, wenn mehr als drei Keimreihenkopien durchzuführen sind, das nächste Byte bzw. die nächsten Bytes Folgekeimreihenzählwerte und nicht das erste Pixel ist bzw. sind.

– inc_next_ptr_2: Für die Befehlsladeübernahmesignale ld_next_cmd_lit und ld_next_imm_cmd inkrementieren wir die Zeiger um zwei Positionen, weil der Befehl, der geladen ist, in diesen Fällen dem current_pointer um eine Position vorausgeht. Ferner inkrementieren wir den Zeiger um zwei Positionen, wenn wir erste Byteübernahmesignale ld_next_fb_run oder ld_imm_fb bekommen, da der current_pointer in diesen Fällen auf den neuen Befehl zeigt, was das erste Pixel um eine Byteposition voraus plaziert.

– inc_next_ptr_3: Falls wir die ersten Byteladeübernahmesignale ld_next_fb_lit oder ld_next_imm_fb bekommen, inkrementieren wir die Zeiger um drei Positionen. In diesen Fällen zeigt der current_pointer auf das letzte Pixel des vorhergehenden Befehls, zeigt der current_pointer plus 1 auf den neuen Befehl und zeigt der current_pointer plus 2 auf das erste Pixel. Wenn wir den Befehl und das erste Pixel laden, müssen wir die Zeiger um 3 Positionen vorrücken, um zu dem nächsten ungelesenen Byte zu gelangen.

[0076] In bezug auf all diese Übernahmesignale sollte es klar sein, daß die Befehls- und die Erstes-Pixel-Übernahmesignale in Synchronizität zueinander auftreten sollten. Es kann eine Schaltungsanordnung implementiert sein, um zu gewährleisten, daß sich 4 Pixel, die dem current_pointer vorausgehen, in dem Eingangspuffer befinden, um ein Anhalten zu verhindern. Siehe decomp_ack (Dekomp_Bestätigung) in einem späteren Abschnitt, in dem auf dieses Detail Bezug genommen wird.

[0077] Man beachte ferner, daß inc_next_ptr_3 und inc_next_ptr_2 beide zum gleichen Zeitpunkt stattfinden könnten. Die Strategie dieses Blocks besteht darin, daß, wenn zwei Inkrementübernahmesignale auftreten, das größere dasjenige sein wird, das ausgeführt wird.

– src_sub_case (Krk_Folge_Fall): Dieser Ausdruck definiert, wann wir eine Keimreihenkopie durchführen, die zusätzliche Keimreihenkopiezählwerte erfordert. Dies ist der Fall, wenn wir eine Anfang-Keimreihenkopie durchführen (unter Verwendung des Zählwerts in dem Befehlsbyte) und der Zählwert 0x03 ist oder wenn wir Folgezählwert durchführen (unter Verwendung eines strom-internen Zählwerts) und der Zählwert 0xFF ist.

– ld_src_subsqnt: Dieser Ausdruck wird verwendet, um den nächsten Folgekeimreihenkopiezählwert von dem eingegebenen komprimierten Datenstrom zu laden. Hier stellen wir sicher, daß wir das letzte Byte (src_count = 1 (Krk_Zählwert = 1) und decomp_ack = 1), wie es durch den vorhergehenden Keimreihenkopiezählwert definiert ist, transferieren und daß wir tatsächlich einen weiteren Zählwert (wie er durch src_sub_case = 1 definiert ist) einholen müssen. Wir prüfen ferner, daß wir uns nicht am Ende des Streifens befinden und daß wir den nächsten Folgezählwert noch nicht geladen haben. Da wir den Befehl zuvor geladen haben und eine Keimreihenkopie durchführen (was nicht mehr komprimierte Strom-Bytes erfordert), zeigt der current_pointer auf das nächste Byte in dem komprimierten Datenstrom. Wir gewährleisten ferner, daß es nicht 0x00 (current_byte /= 0x00) ist. Falls es 0x00 wäre, würden wir es überspringen und das erste Pixel laden, falls erforderlich. Falls das nächste Byte 0x00 ist, wird diese Ladegleichung deaktiviert, und es wird ein anderer Ausdruck verwendet, um zu überspringen und zu laden.

– block_00_src: Dieser Ausdruck erwartet den oben erwähnten Fall, bei dem der nächste Keimreihenkopiezählwert tatsächlich ein Endzählwert von 0x00 ist. Um dies zu bestimmen, halten wir nach exakt denselben Bedingungen Ausschau wie für ld_src_subsqnt, wir prüfen jedoch bezüglich current_byte = 0x00. Wenn dieser Ausdruck wahr ist, bedeutet dies, daß wir das nächste Byte überspringen und weitergehen sollten. Er wird in den folgenden Gleichungen verwendet.

– ld_next_cmd_00_src: Dieser Ausdruck wird verwendet, um einen neuen Befehl zu laden, falls die Keimreihenkopie zufällig eine Zeile beendet und der letzte gültige Keimreihenkopiezählwert 0xFF war. In diesem Fall überspringen wir das nächste Byte in dem komprimierten Eingangsdatenstrom und laden das nächste Byte als den nächsten Befehl. Dieser Begriff erwartet, daß block_00_src wahr ist. Falls es wahr ist, wissen wir, daß wir das nächste Byte überspringen. Wir prüfen ferner, ob es als Befehl geladen werden sollte, indem wir eol_approach d betrachten, das eine registrierte Version von eol_approach ist. Eol_approach ist auf wahr gesetzt, wenn wir einen Keimreihenkopiezählwert (Anfang oder Folge) laden, der uns zum Ende der Zeile bringt. Wenn dies wahr ist, wissen wir, daß die Keimreihenkopie eine Zeile

beendet und daß wir einen neuen Befehl laden sollten, statt mit einer Literal- oder Laufersetzung fortzufahren. Wir sind hier nicht mit decomp_xfer qualifiziert, weil das durch block_00_src abgedeckt ist.

– ld_next_fb_00_src: Dieser Ausdruck gehört zu dem letzten Ausdruck. Wenn wir einen neuen Befehl laden und der Keimreihenkopiezählwert geringer als 3 (byte1_ahead 4:3 /= 0x03) ist und eine Stelle des ersten Pixels dieses nächsten Befehls eingereiht (in-line) ist (byte1_ahead 6:5 = '00'), dann aktivieren wir auch diesen Ausdruck. Man beachte, daß dieser Ausdruck mit block_00_src qualifiziert ist, ebenso wie die letzte Gleichung. In diesem Fall halten wir jedoch nach byte2_ahead_avail = 1 Ausschau, da der current_pointer plus 2 auf dieses erste Pixel zeigt (current_pointer zeigt auf 0x00 und current_pointer plus 1 zeigt auf den neuen Befehl). Schließlich stellen wir sicher, daß wir nicht das first_pixel geladen haben (first_pixel_loaded = 0). Wir sind hier nicht mit decomp_xfer qualifiziert, da dies durch block_00_src abgedeckt ist.

– ld_fb_00_src: Dieser Ausdruck lädt das erste Pixel, falls nötig, für den folgenden Lauf oder das folgende Literal, wenn das Ende der Keimreihenkopie erfaßt ist und der letzte src_count 0xFF betrug und das nächste Byte das End-0x00 ist. Wiederum erfassen wir dies unter Verwendung von block_00_src. Wir stellen sicher, daß die aktuelle Keimreihenkopie uns nicht zu dem Ende der Zeile bringt, indem wir eol_approach und eol_approach d betrachten (beide sollten 0 sein). Wir prüfen, daß das Byte an dem current_pointer plus 1 verfügbar ist (byte1_ahead_avail = 1), daß wir das erste Pixel noch nicht geladen haben und daß wir tatsächlich ein erstes Pixel laden müssen (Stelle = 1). Wir sind hier nicht mit decomp_xfer qualifiziert, da dies durch block_00_src abgedeckt ist.

– skip_00_src (überspringe_00_src): Dieser Ausdruck kommt zum Tragen, wenn wir eine Keimreihenkopie haben, die mit einem Anfangszählwert von 0x03 oder einem Folgezählwert von 0xFF beendet wurde, und die Stelle des ersten Pixels nicht 0x00 ist. Da wir kein erstes Pixel benötigen, aktivieren wir diesen Ausdruck, um einfach den current_pointer hinter dem Keimreihenkopieendzählwert 0x00 zu inkrementieren. Wir führen eine Prüfung bezüglich der ersten Bedingung durch, indem wir block_00_ld verwenden, und führen eine Prüfung bezüglich der zweiten Bedingung durch, indem wir eine Stelle verwenden. Unter Verwendung von eol_approach und eol_approach_d stellen wir sicher, daß wir uns nicht am Ende der Zeile befinden.

– ld_next_fb_src: Dieser Ausdruck lädt das erste Pixel, falls erforderlich, für den folgenden Lauf oder das folgende Literal, wenn das Ende der Keimreihe erfaßt ist und die Keimreihenkopie selbst keine zusätzlichen Zählwerte erfordert. Für dieses Übernahmesignal stellen wir unter Verwendung von eol_approach und eol_approach d sicher, daß wir uns nicht am Ende der Zeile befinden. Wir prüfen, daß das Byte, auf das der current_pointer zeigt, gültig ist (input_available = 1), daß der Befehl ein erstes Pixel (Stelle = 00) erfordert und daß wir das erste Pixel nicht geladen haben. Wir prüfen first_pixel_loaded, weil wir sicherstellen wollen, daß, falls wir das erste Pixel als Teil der Befehlsladung geladen haben, wir nicht versuchen, es erneut zu laden. Dies sagt zwingenderweise aus, daß der Anfangskeimreihenkopiezählwert größer war als drei. Da wir an dem letzten Keimreihenkopiezählwert laden möchten, stellen wir sicher, daß wir uns an dem letzten Pixelzählwert befinden. Dazu prüfen wir, ob entweder src_count = 0x01 (src_orig >1) oder daß src_orig = 1 und daß wir keine weiteren Zählwerte erwarten (src_sub_case = 0). Schließlich überprüfen wir, daß wir uns tatsächlich in einer Keimreihenkopie befinden (src_in_progress = 1) und daß wir tatsächlich ein Pixel transferieren (decomp_ack = 1).

– inc_endsrc_ptr_1: Wir inkrementieren die Zeiger um 1 Position, falls wir lediglich ein erstes Pixel laden (ld_next_fb_src), falls wir lediglich einen zusätzlichen Keimreihenkopiezählwert laden (ld_src_subsqnt) oder falls wir das strominterne 0x00 überspringen müssen (skip_00_src). Ferner inkrementieren wir immer dann, wenn block_00_src wahr ist, um 1, um den Fall abzudecken, bei dem das letzte Byte eines eingegebenen komprimierten Wortes 0x00 beträgt (ein Endzählwert) und das nächste komprimierte Wort noch nicht verfügbar ist. In diesem Fall müssen wir das End-0x00 überspringen, obwohl wir gerade keinen neuen Befehl oder Zählwert laden, damit wir für das nächste gültige Byte bereit sind, wenn das nächste Wort schließlich gültig ist.

– inc_endsrc_ptr_2: Wir inkrementieren die Zeiger um 2 Positionen, wenn wir ein 0x00 überspringen, um den nächsten Befehl zu laden (ld_next_cmd_00_src), oder wenn wir ein 0x00 überspringen müssen, um das erste Pixel für den aktuellen Befehl zu laden (ld_fb_00_src).

– inc_endsrc_ptr_3: Wir inkrementieren die Zeiger um 3 Positionen, wenn wir ein 0x00 überspringen, um das erste Pixel für den nächsten Befehl zu laden (ld_next_fb_00_src).

[0078] In bezug auf all diese Übernahmesignale sollte es klar sein, daß die Befehls- und die Erstes-Pixel-Übernahmesignale in Synchronizität zueinander auftreten sollten, falls ein erstes Pixel benötigt wird. Es kann eine Schaltungsanordnung verwendet werden, um zu gewährleisten, daß zumindest 4 Pixel, die dem current_pointer vorausgehen, in dem Eingangspuffer enthalten sind, um ein Anhalten zu verhindern. Siehe decomp_ack in einem späteren Abschnitt, in dem auf dieses Detail Bezug genommen wird.

[0079] Man beachte ferner, daß inc_endscr_ptr_3 und inc_endscr_ptr_2 beide zum gleichen Zeitpunkt stattfinden könnten. Die Strategie dieses Blocks besteht darin, daß, wenn zwei oder mehr Inkrementübernahmesignale auftreten, das größere dasjenige sein wird, das ausgeführt wird.

– ld_cmd: Dieser Ausdruck ist wahr, wenn wir angehalten haben, während wir auf den nächsten Befehl warten, und neue Daten verfügbar sind. Wenn wir entweder ganz am Anfang beginnen oder Bytes ausgegeben haben und anhalten, während wir auf den nächsten Befehl warten, geht die Steuerzustandsmaschine zu Ruhe (dec_idle) (dekrementieren_Ruhe). Wenn mehr komprimierte Eingangsdaten verfügbar sind, ist input_available wahr. Wenn wir keinen Befehl geladen haben (command_loaded = 0) und wir in Ruhe sind (current_dec_ctl = dec_idle), wird dieser Ausdruck aktiviert.

– ld_first_byte: Dieser Ausdruck ist wahr, wenn wir einen Befehl laden und der Befehl nach einem ersten Pixel ruft und die Anzahl von Keimreihenkopien geringer ist als drei. Der current_pointer zeigt auf den neuen Befehl. Falls current_byte (6:5) = 00, so holen wir ein erstes Pixel aus dem komprimierten Datenstrom. Falls current_byte (4:3) geringer ist als 3, werden weniger als 3 Keimreihenkopien vorliegen. Wir prüfen, daß das Byte nach dem aktuellen Byte verfügbar ist (byte1_ahead_avail = 1). Falls first_pixel_loaded nicht wahr ist, wir uns in Ruhe befinden (wir bei dem obigen Ausdruck) und die Keimreihenkopie des Befehls nicht die Zeile beendet, aktivieren wir dieses Übernahmesignal. Es kann sein, daß der Befehl selbst die Zeile beenden würde, falls uns die Keimreihenkopie zum Ende der Zeile bringt, und falls dies geschieht, möchten wir sicherlich nicht das erste Pixel laden. Wir erfahren, ob uns der Befehl zum Ende der Zeile bringt, indem wir sowohl eol_approach als auch eol_approach d betrachten, die gesetzt werden, falls ein neuer Keimreihenkopiezählwert zum Ende der Reihe läuft.

– inc_idle_ptr_1: Falls wir den Befehl (ld_cmd) laden, inkrementieren wir den Zeiger um 1.

– inc_idle_ptr_2: Falls wir das erste Pixel (ld_first_byte) laden, inkrementieren wir den Zeiger um 2.

[0080] In bezug auf diese Übernahmesignale sollte es klar sein, daß die Befehls- und die Erstes-Pixel-Übernahmesignale in Synchronizität zueinander auftreten sollten, falls ein erstes Pixel benötigt wird. Es liegt eine Schaltungsanordnung vor, um zu gewährleisten, daß sich zumindest 4 Pixel, die dem current_pointer vorausgehen, in dem Eingangspuffer befinden sollten, oder daß andernfalls ein Anhalten eintritt. Siehe decomp_ack in einem späteren Abschnitt, in dem auf dieses Detail Bezug genommen wird.

[0081] Man beachte ferner, daß inc_idle_ptr_3 und inc_idle_ptr_2 beide zum gleichen Zeitpunkt stattfinden könnten. Die Strategie dieses Blocks besteht darin, daß, wenn zwei oder mehr Inkrementübernahmesignale auftreten, das größere dasjenige sein wird, das ausgeführt wird.

– rep_sub_case: Dieser Ausdruck sagt uns, daß wir uns in einer Laufnachbildung oder einer Literalersetzung befinden, wo wir mehr Zählwerte erwarten. Möglicherweise benötigen wir mehr Zählwerte, wenn wir den Anfangszählwert in dem Befehl verwenden und er 7 beträgt oder wenn wir einen Folgezählwert verwenden und er 0xFF beträgt. Das Signalinitial wird verwendet, um uns mitzuteilen, daß wir gerade Zählwerte von dem Befehlszählwert nachbilden (Initial = 1). Wenn also ein Initial gleich 0 ist und das rep_orig gleich 0xFF ist, wissen wir, daß wir gerade einen Folgezählwert von 0xFF nachbilden. Die Anfangsnachbildung weist zwei verschiedene Fälle auf. Der erste tritt ein, wenn wir uns in einem Lauf befinden, da hier rep_orig gleich 9 ist, falls der tatsächliche Wert in dem Befehlsfeld 7 war. Man erinnere sich aus Abschnitt 1.0, daß für einen Lauf der Nachbildungszählwert tatsächlich zwei weniger beträgt als der gewünschte Laufwert, und für Literale ist er einer weniger. Somit prüfen wir bei Literalen, ob rep_orig gleich 8 ist. Wir benützen den rep_orig-Wert statt des Originalwerts aus dem Befehl, da das Befehlsregister ungültig wird, nachdem es gelesen wird, um rep_orig, Stelle, Lauf und src_orig zu laden. Schließlich prüfen wir, daß rep_sub_loaded falsch ist, so daß wir einen Folgewert nicht mehr als einmal laden.

– block_rep_ld: Dieses Übernahmesignal wird verwendet, um die Tatsache zu signalisieren, daß der aktuelle Nachbildungszählwert entweder 8/9 beträgt und wir uns in einer Anfangsnachbildung befinden oder daß der aktuelle Nachbildungszählwert 0xFF beträgt und wir uns in einem Folgezählwert befinden, und das nächste Byte ein 0x00 ist. Spätere Ausdrücke werden dies berücksichtigen, wobei sie dies verwenden, wenn sie bereit sind, entweder einen weiteren Zählwert oder einen neuen Befehl zu laden. Falls rep_sub_case wahr ist, wissen wir, daß wir nach mehr Zählwerte Ausschau halten sollten. Falls das aktuelle Byte verfügbar ist (input_available = 1) und das current_byte (aktuelles_Byte) 0x00 beträgt und wir uns entweder in einem Lauf befinden und decomp_xfer wahr ist oder wir auf einen gültigen Zählwert warten (wait_rep = 1) (warten_Nachbildung = 1), werden wir dieses Übernahmesignal aktivieren. Man beachte, daß in beiden Fällen, in denen wir einen Lauf aufweisen oder in denen wir auf ein gültiges nächstes Byte warten, der current_pointer auf diesen nächsten Zählwert-Wert zeigt, weshalb wir nach input_available Ausschau halten. Alternativ dazu prüfen wir, wenn wir uns in einer Literalnachbil-

dung befinden, bezüglich byte1_ahead_avail = 1 und byte1_ahead = 0x00, um zu bestimmen, ob wir dieses Übernahmesignal aktivieren sollten. Für diesen Fall stellen wir sicher, wie bei dem Lauffall, daß wir Daten transferieren (decomp_ack = 1) und daß wir uns nicht in einem Lauf befinden (Lauf = 0).

– ld_rep_subsqnt: Dieses Übernahmesignal wird verwendet, um einen anderen Nachbildungszählwert zu laden, wenn der aktuelle Zählwert ausgeht (rep_count = 0x01) und wir tatsächlich einen weiteren Zählwert benötigen (rep_sub_case = 1). Wir prüfen, daß der nächste Wert geladen werden sollte (block_rep_ld = 0) und daß wir Daten aus der Dekomprimierungseinrichtung hinaustransferieren (decomp_ack = 1). Wir prüfen, daß Daten entweder für einen Lauf oder falls wir warten, zur Verfügung stehen (input_available = 1), da in beiden Fällen der current_pointer auf das nächste Byte zeigt. Falls wir ein Literal durchführen, prüfen wir, daß byte1_ahead_avail = 1, da der current_pointer auf das letzte Pixel zeigt und der current_pointer plus 1 auf den nächsten Zählwert zeigt.

– inc_repsub_ptr_1: Dieses Übernahmesignal wird verwendet, um die Zeiger um eine Position zu inkrementieren, wenn wir einen Folgenachbildungswert laden und wir einen Lauf durchführen, oder wenn block_00_rep wahr ist. Wenn wir eine Laufnachbildung durchführen, zeigt der aktuelle Zeiger auf das nächste komprimierte Byte, was in diesem Fall der nächste Laufzählwert ist. Wenn wir also den nächsten Laufnachbildungswert laden, inkrementieren wir einfach den Zeiger um 1, um zu dem nächsten Byte zu gelangen. Falls block_00_rep wahr ist und wir einen Lauf durchführen, sollten wir den Zeiger am Ende des aktuellen Transfers, der der letzte Transfer des vorhergehenden Befehls ist, um 1 inkrementieren. Normalerweise würden wir ferner zumindest einen neuen Befehl und möglicherweise ein neues erstes Pixel laden, so daß wir tatsächlich um mehr als eins inkrementieren. Es gibt jedoch den Fall, bei dem sich der nächste Befehl in dem nächsten komprimierten Wort befindet, das noch nicht in die Dekomprimierungseinrichtung geladen wurde. Dies sollte uns veranlassen, zu dem Ruhezustand überzugehen und auf die neuen Daten zu warten. Falls wir den Zeiger nicht um 1 inkrementieren, würden wir auf das End-0x00 zeigen, wenn neue Daten zur Verfügung stünden, was bewirkt, daß das 0x00 als der nächste Befehl geladen wird. Somit bewegen wir den Zeiger um eins voraus, so daß der current_pointer auf das nächste gültige Byte zeigt. Man beachte, daß wir lediglich um eins inkrementieren, da der current_pointer für einen Lauf bereits auf das End-0x00 zeigte.

– block_00_rep: Dieses Übernahmesignal wird verwendet, um ein Flag zu setzen, wenn block_rep_ld wahr ist und wir das letzte Byte des aktuellen Befehls aus der Dekomprimierungseinrichtung hinaustransferieren. Block_rep_ld identifiziert nicht, wann dieser Fall eintritt, es identifiziert lediglich, wann der Fall eintreten könnte. Falls rep_sub_case wahr ist, rep_count = 0x01 und block_rep_ld wahr sind, aktivieren wir dieses Signal, um mitzuteilen, daß wir das letzte Byte hinaustransferieren und das nächste Byte ein End-00 ist, das ignoriert werden sollte.

– ld_next_cmd_00_rep: Falls block_00_rep wahr ist, müssen wir das 00 überspringen und unmittelbar das nächste Byte, das ein Befehlsbyte ist, laden. Falls es ein Lauf ist, zeigt der current_pointer auf das nächste Byte, das in diesem Fall 00 ist. Somit möchten wir das Byte nach dem 00 laden, was bei current_pointer plus 1 vorliegt. Falls wir uns in einem Literal befinden, halten wir danach Ausschau, daß byte2_ahead_avail wahr ist, was bedeutet, daß das Byte bei current_pointer plus 2 gültig ist. Da dies ein Literal ist, zeigt der current_pointer auf das letzte Literal, zeigt der current_pointer plus 1 auf 00 und zeigt der current_pointer plus 2 auf den nächsten Befehl. Sowohl für Lauf als auch für Literal prüfen wir ferner, daß wir Daten transferieren (decomp_xfer = 1).

– ld_next_fp_00_rep:: Falls block_00_rep wahr ist, müssen wir das 00 überspringen und unmittelbar das nächste Byte, das ein Befehlsbyte ist, laden. Es kann sein, daß wir auch ein erstes Pixel laden müssen. Falls block_00_rep wahr ist und wir derzeit einen Lauf durchführen, sehen wir nach, ob byte1_ahead (6:5) 00 ist und ob byte1_ahead (4:3) nicht gleich 3 ist. Falls diese beiden wahr sind, laden wir ein erstes Byte. Da dies ein Lauf ist, zeigt der current_pointer auf das nächste Byte, das in diesem Fall 00 ist. Somit möchten wir das Byte nach dem 00 laden, das sich bei current_pointer plus 1 befindet. Das erste Pixel befindet sich tatsächlich bei current_pointer plus 2, deshalb prüfen wir, daß byte2_ahead_avail wahr ist. Falls wir uns in einem Literal befinden, sehen wir nach, ob byte2_ahead (6:5) 0x00 ist und ob byte2_ahead (4:3) nicht gleich 3 ist. Falls diese beiden wahr sind, laden wir ein erstes Byte. Da dies ein Literal ist, zeigt der current_pointer auf das letzte Literal, zeigt der current_pointer plus 1 auf 00, zeigt der current_pointer plus 2 auf den nächsten Befehl und zeigt current_pointer plus 3 auf das erste Pixel. Somit prüfen wir, ob byte3_ahead_avail wahr ist. Wir halten hier nicht nach decomp_xfer Ausschau, da es in block_00_rep abgedeckt ist.

– inc_repsub_ptr_2: Dieses Übernahmesignal inkrementiert die Zeiger um 2. Wir aktivieren es, falls ld_rep_subsqnt wahr ist und wir ein Literal durchführen, oder wenn ld_next_cmd_00_rep wahr ist und wir einen Lauf durchführen. Wenn wir ein Literal durchführen, erinnere man sich, daß sich der nächste Nachbildungszählwert nicht an

dem current_pointer, sondern an dem current_pointer plus 1 befindet, da current_pointer auf das letzte Literal zeigt. Somit inkrementieren wir um 2. Falls wir einen Lauf durchführen und der letzte Zählwert ein End-00 ist, zeigt der current_pointer auf das 00, und der current_pointer plus 1 zeigt auf den nächsten Befehl. Somit müssen wir nach einem Laden des nächsten Befehls den Zeiger um 2 inkrementieren. Falls block_00_rep wahr ist und wir ein Literal durchführen, sollten wir am Ende des aktuellen Transfers, der der letzte Transfer des vorhergehenden Befehls ist, den Zeiger um 2 inkrementieren. Normalerweise würden wir ferner zumindest einen neuen Befehl und möglicherweise ein neues erstes Pixel laden, so daß wir tatsächlich um mehr als eins inkrementieren. Es gibt jedoch den Fall, bei dem sich der nächste Befehl in dem nächsten komprimierten Wort befindet, das noch nicht in die Dekomprimierungseinrichtung geladen wurde. Dies sollte uns veranlassen, zu dem Ruhezustand überzugehen und auf die neuen Daten zu warten. Falls wir den Zeiger nicht um 2 inkrementieren, würden wir auf das End-0x00 zeigen, wenn neue Daten zur Verfügung stünden, was bewirkt, daß das 0x00 als der nächste Befehl geladen wird. Somit bewegen wir den Zeiger um zwei voraus, so daß der current_pointer auf das nächste gültige Byte zeigt. Man beachte, daß wir um zwei inkrementieren, da der current_pointer für ein Literal auf das letzte Byte, das dem End-0x00 um eins vorausgeht, zeigte.

– inc_repsub_ptr_3: Dieses Übernahmesignal inkrementiert die Zeiger um 3. Wir aktivieren es, wenn wir einen Lauf durchführen und ld_next_fp_00_rep_wahr ist, oder wenn wir ein Literal durchführen und ld_next_cmd_00_rep_wahr ist. Wenn wir einen Lauf durchführen und wir auf ein End-00 stoßen, zeigt current_pointer plus eins auf den nächsten Befehl, was bewirkt, daß current_pointer plus zwei auf das erste Pixel zeigt. Somit müssen wir den Zeiger um 3 inkrementieren. Wenn wir ein Literal durchführen und wir auf ein End-00 stoßen, zeigt der current_pointer auf das letzte Pixel, zeigt der current_pointer plus 1 auf das 0x00 und zeigt der current_pointer plus 2 auf den nächsten Befehl. Somit müssen wir den Zeiger um 3 inkrementieren.

– inc_repsub_ptr4: Dieses Übernahmesignal inkrementiert die Zeiger um 4. Wir aktivieren es, wenn wir ein Literal durchführen und ld_next_fp_00_rep_eintritt. Wenn wir ein Literal durchführen und auf ein End-00 stoßen, zeigt der current_pointer auf das letzte Pixel, zeigt der current_pointer plus 1 auf das 0x00, zeigt der current_pointer plus 2 auf den nächsten Befehl und zeigt der current_pointer plus 3 auf das erste Pixel. Somit müssen wir den Zeiger um 4 inkrementieren.

[0082] In bezug auf diese Übernahmesignale sollte es klar sein, daß die Befehls- und die Erstes-Pixel-Übernahmesignale in Synchronizität zueinander auftreten sollten, falls ein erstes Pixel benötigt wird. Es liegt eine Schaltungsanordnung vor, um zu gewährleisten, daß sich zumindest 4 Pixel, die dem current_pointer vorausgehen, in dem Eingangspuffer befinden sollten, oder daß andernfalls ein Anhalten eintritt. Siehe decomp_ack in einem späteren Abschnitt, in dem auf dieses Detail Bezug genommen wird.

[0083] Man beachte ferner, daß inc_repsub_ptr_2 und inc_repsub_ptr_3 beide zum gleichen Zeitpunkt stattfinden könnten, ebenso wie inc_repsub_ptr_3 und inc_repsub_ptr_4. Die Strategie dieses Blocks besteht darin, daß, wenn zwei oder mehr Inkrementübernahmesignale auftreten, das größere dasjenige sein wird, das ausgeführt wird.

– last_byte: Das Signal last_byte (letztes_Byte) wird in den obigen Gleichungen verwendet, um ein Flag zu setzen, wenn das letzte Datenbyte der Operation ausströmt. Man beachte, daß wir für Literale einer Länge von 1 nicht last_byte signalisieren. Dies geschieht deshalb, damit wir keine Ladegleichung überladen und mehr als eine Ladegleichung aufweisen, die zum Tragen kommt. In diesem Fall haben wir einen expliziten Ladeausdruck (ld_imm_cmd). Das letzte Byte ist ein „ODER" der folgenden Ausdrücke.

– src_of_1_eol: Wenn src_orig_m1, der Ursprungswert des geladenen Keimreihenkopiezählwerts minus 1 gleich 0 ist, eine Keimreihenkopie im Gange ist (src_in_progress = 1) und wir uns am Ende der Zeile befinden (decomp_eol_b = 1), aktivieren wir dieses Übernahmesignal. Man beachte, daß wir nach dem Ende der Zeile Ausschau halten, da, mit der Ausnahme dieses Falles, eine Keimreihenkopie einen Befehl nicht beendet. Statt nach dem Transfer von src_orig = 1 Ausschau zu halten, wird hier src_orig_m1 verwendet, da ein Ausschauhalten nach 0x00 wahrscheinlich leichter ist als ein Ausschauhalten nach 0x01 und daß decomp_xfer wahr ist.

– src_gt_1: Dieses Übernahmesignal wird aktiviert, wenn wir eine Keimreihenkopie durchführen (src_in_progress = 1), der src_count 1 ist, wir nicht mehr Zählwerte erwarten und wir uns am Ende der Zeile befinden. Man beachte, daß wir nach dem Ende der Zeile Ausschau halten, da, mit Ausnahme dieses Falles, eine Keimreihenkopie einen Befehl nicht beendet.

– rep_gt_1: Dieses Übernahmesignal wird aktiviert, wenn wir eine Nachbildung durchführen (rep_in_progress = 1), rep_count_m1 0 ist und wir nicht mehr Zählwerte erwarten.

Kombinatorische Signale

[0084] Die nächsten Ausdrücke werden verwendet,

um Register zu laden und einen Datenfluß in den aus dem Decodierer zu steuern.

Registersteuersignale

– cmd_ld_src: Dieses Signal sagt uns, daß der nächste Befehl einen Keimreihenkopieinhalt aufweist.
– Dieses Signal wird aktiviert, wenn wir den Befehl geladen haben (command_loaded = 1) und sein zugeordneter Keimreihenkopiezählwert nicht 0 ist (next_src_ini /= 0).
– Falls ld_cmd wahr ist (bei current_dec_ctl von dec_idle auftritt), ld_next_cmd_run wahr ist (was anzeigt, daß ein Lauf gerade endet und wir einen neuen Befehl laden), oder wenn ld_imm_cmd wahr ist (was anzeigt, daß wir eine Keimreihenkopie- oder Literalnachbildung von 1 durchführen), aktivieren wir diesen Befehl, falls current_byte (4:3) nicht 0 ist. Wir verwenden current_byte hier, da in jedem dieser Fälle current_byte auf den neuen Befehl zeigt.
– Falls ld_next_cmd_00_rep wahr ist (was anzeigt, daß wir ein End-00 am Ende eines Laufs oder Literals überspringen) und wir uns in einem Literalersetzungsmodus befinden, aktivieren wir dieses Signal, falls byte2_ahead (4:3) nicht 00 ist. Wir verwenden byte2_ahead für diesen Literalfall, da der current_pointer auf dem letzten Byte auf das letzte Pixel zeigt, der current_pointer plus 1 auf das 00 zeigt und der current_pointer plus 2 auf den neuen Befehl zeigt.
– Falls ld_next_cmd_00_src oder ld_next_imm_cmd wahr sind, aktivieren wir dieses Signal, falls byte1_ahead (4:3) nicht gleich 0 ist. Wir verwenden byte1_ahead, da der current_pointer für eine Keimreihenkopie mit einem End-0x00 auf das 00 zeigt und der current_pointer plus 1 auf den nächsten Befehl zeigt. Für ld_next_imm_cmd zeigt der current_pointer auf den letzten Befehl, was dazu führt, daß der current_pointer plus 1 auf den neuen Befehl zeigt. Somit verwenden wir byte1_ahead für die Prüfung.
– Falls ld_next_cmd_00_rep wahr ist und wir einen Lauf durchführen und byte1_ahead (4:3) nicht gleich 0 ist, aktivieren wir dieses Signal. Wir verwenden hier byte1_ahead, da, falls wir uns in einem Lauf befinden und ein End-0x00 vorliegt, der current_pointer auf das 0x00 zeigt, was bewirkt, daß der current_pointer plus 1 auf den nächsten Befehl zeigt.
– Falls ld_next_cmd_lit wahr ist und wir entweder derzeit einen Folgezählwert transferieren (rep_mux_ctl = 1) oder einen Anfangszählwert (rep_mux_ctl = 0) von mehr als einem Byte (rep_orig_m1 /= 0) transferieren und byte1_ahead (4:3) nicht gleich 0 ist, aktivieren wir dieses Signal. Wir verwenden hier byte1_ahead, da, falls wir eine Literalnachbildung durchführen und wir mehr als ein Byte zu transferieren haben, der current_pointer auf das letzte Pixel der Nachbildung zeigt, was bewirkt, daß der current_pointer plus 1 auf den neuen Befehl zeigt.
– Falls schließlich ld_next_cmd_lit wahr ist und wir einen Anfangszählwert transferieren (rep_mux_ctl = 0) und wir lediglich ein Byte (rep_orig_m1 = 0) und current_byte (4:3) /= 00 transferieren, aktivieren wir dieses Signal. Wir verwenden hier current_byte, da, wenn lediglich ein Byte eines Literals auszusenden ist und wir derzeit dieses erste Byte aussenden, current_pointer auf den nächsten Befehl zeigt. Das liegt daran, daß das first_pixel zuvor mit dem letzten Befehl geladen wurde.
– cmd_ld_no_fp: Dieses Signal ermöglicht es dem Signal loading_first_pixel, zum Tragen zu kommen, auch wenn wir derzeit nicht ein erstes Pixel laden. Wir tun dies deshalb, damit first_pixel_loaded gesetzt wird; dies ermöglicht es uns, die Zustandsmaschine auf einfachere Weise zu steuern.
– Falls wir uns in current_dec_ctl = dec_idle befinden und wir einen Befehl (ld_cmd) laden oder falls ld_next_cmd_run wahr ist oder falls ld_imm_cmd wahr ist und current_byte (6:5) nicht 00 ist, aktivieren wir dieses Signal. Wir verwenden current_byte mit ld_cmd, da, falls wir uns in einem Ruhezustand befinden, current_byte auf das nächste gültige Byte, das der nächste Befehl sein wird, zeigt. Wir müssen die Befehlsbits (6:5) prüfen, um zu sehen, ob eine Stelle 00 ist oder nicht. Falls sie nicht 00 ist, laden wir nicht ein erstes Pixel. ld_next_cmd_run sagt uns, daß wir das letzte Byte eines Laufs aussenden. In diesem Fall zeigt der current_pointer bereits eine Weile auf das nächste gültige Byte, das der nächste Befehl ist. ld_imm_cmd sagt uns, daß ein Literal von einer Länge 1 stattfindet. In diesem Fall haben wir den Befehl und ein first_pixel (falls benötigt) für den letzten Befehl geladen, was bewirkt, daß current_pointer auf das nächste gültige Byte, das der nächste Befehl ist, zeigt.
– Falls ld_next_cmd_00_rep wahr ist und wir ein Literal durchführen (Lauf = 0) und byte2_ahead (6:5) nicht 00 ist, aktivieren wir dieses Signal. Wir verwenden byte2_ahead, da der current_pointer an einem Literal auf das letzte Pixel zeigt, was bewirkt, daß der current_pointer plus 1 auf 00 zeigt und der current_pointer plus 2 auf den neuen Befehl zeigt.
– Falls ld_next_cmd_00_src wahr ist oder ld_next_imm_cmd wahr ist und byte1_ahead (6:5) nicht 00 ist, aktivieren wir dieses Signal. Falls ld_next_cmd_00_src wahr ist, wissen wir, daß current_pointer auf das 0x00-Endbyte zeigt, was bewirkt, daß der current_pointer plus 1 auf den nächsten Befehl zeigt. Falls ld_next_imm_cmd wahr ist, wissen wir, daß current_pointer auf das letzte Literal zeigt, da ld_next_imm_cmd nur dann

stattfindet, wenn sich ein letztes Literal am Ende einer Folge-Literalnachbildung befindet. In diesem Fall zeigt current_pointer plus 1 auf das nächste Byte nach dem letzten Pixel, das der nächste Befehl ist.

– Falls ld_next_cmd_00_rep wahr ist und wir einen Lauf durchführen und byte1_ahead (6:5) wahr ist, aktivieren wir dieses Signal. Im Fall von Läufen inkrementieren die Zeiger, nachdem der Laufbefehl oder Folgezählwerte geladen wurden. Somit zeigt der current_pointer für Läufe auf den nächsten Zählwert, der in diesem Fall der 0x00-Endzählwert ist. Somit zeigt current_pointer plus 1 auf den nächsten Befehl.

– Falls ld_next_cmd_lit wahr ist und wir Folgezählwerte transferieren (rep_mux_ctl = 1) und byte 1_ahead (6:5) nicht 00 ist, aktivieren wir dieses Signal. In diesem Fall führen wir aktuell eine Literalnachbildung durch und wir bilden Folgebytes nach, was bedeutet, daß current_pointer auf die Literalpixel zeigt. Wenn current_pointer auf das letzte Pixel zeigt, ist ld_next_cmd_lit wahr. Zu diesem Zeitpunkt zeigt current_pointer plus 1 auf das Byte nach dem letzten Pixel, das der nächste Befehl ist.

– Falls ld_next_cmd_lit wahr ist und wir eine Anfangsnachbildung durchführen (rep_mux_ctl = 0) und wir mehr als 1 Literal transferieren und byte1_ahead (6:5) nicht 00 ist, aktivieren wir dieses Signal. In diesem Fall führen wir eine Literalersetzung von mehr als 1 Pixel und weniger als 8 Pixeln durch. Dies bedeutet, daß wir keine Folgebytes zu laden haben oder keinen Endzählwert von 0x00 zu überspringen haben. In diesem Fall zeigt current_pointer auf das letzte Pixel (wenn ld_next_cmd_lit wahr ist), was bewirkt, daß current_pointer plus 1 auf den nächsten Befehl zeigt.

– loading_command: Dies ist ein einfaches „ODER" jeder Befehlsladegleichung von oben. Sie lauten ld_next_cmd_lit, ld_next_cmd_run, ld_next_cmd_00_rep, ld_cmd, ld_imm_cmd, ld_next_imm_cmd und ld_next_cmd_00_src.

– loading_first_pixel: Dies ist ein einfaches „ODER" jeder ersten Pixelladegleichung von oben. Sie lauten ld_first_byte, ld_next_fb_run, ld_imm_fb, ld_next_imm_fb, ld_next_fb_lit, ld_next_fb_src, ld_next_fb_00_src, ld_next_fp_00_rep, cmd_ld_no_fp, ld_fp_0_src und skip_00_src.

– inc_ptr_1: Dies ist ein einfaches „ODER" jedes Inkrements um 1 Gleichung von oben. Sie lauten inc_next_ptr_1, inc_endsrc_ptr_1, inc_idle_ptr_1, inc_repsub_ptr_1. Für Literalnachbildungen inkrementieren wir ferner bei jedem Transfer den Zeiger um 1. Dazu halten wir Ausschau, ob dec_rep_count wahr ist (was uns sagt, daß wir transferiert haben), wobei Lauf falsch ist (wir befinden uns im Literalmodus) und das erste Pixel nicht geladen ist (first_pixel_loaded = 0). Wir prüfen, daß das erste Pixel nicht geladen ist, da wir nicht den Zeiger an dem ersten Pixel inkrementieren möchten, wenn wir das erste Pixel bereits geladen haben. Wir erlauben nicht, daß dieses Signal aktiviert wird, falls inc_ptr_2, inc_ptr_3 oder inc_ptr_4 wahr sind.

– inc_ptr_2: Dies ist ein einfaches „ODER" jedes Inkrements um 2 Gleichungen von oben. Sie lauten inc_next_ptr_2, inc_endsrc_ptr_2, inc_idle_ptr_2 und inc_repsub_ptr_2. Wir erlauben nicht, daß dieses Signal zum Tragen kommt, falls inc_ptr_3 oder inc_ptr_4 wahr sind. Inc_ptr_4 befindet sich derzeit nicht in dieser Gleichung, was es eigentlich sollte!!!

– inc_ptr_3: Dies ist ein einfaches „ODER" jedes Inkrements um 3 Gleichungen von oben. Sie lauten inc_repsub_ptr_3, inc_next_ptr_3 und inc_endsrc_ptr_3. Wir erlauben nicht, daß dieses Signal zum Tragen kommt, falls inc_ptr_4 wahr ist.

– inc_ptr_4: Dies ist ein einfaches „ODER" jedes Inkrements um 4 Gleichungen von oben. Die einzige lautet inc_repsub_ptr_4. Dieses Signal liegt hier lediglich der Klarheit halber vor.

– initial: ist aus Bit 9 der current_dec_ctl-Zustandsmaschine decodiert.

– src_in_progress: aus Bit 8 der current_dec_ctl-Zustandsmaschine decodiert.

– rep_in_progress: aus Bit 7 der current_dec_ctl-Zustandsmaschine decodiert.

– src_mux_ctl: aus Bit 6 der current_dec_ctl-Zustandsmaschine decodiert.

– rep_mux_ctl: aus Bit 5 der current_dec_ctl-Zustandsmaschine decodiert.

– wait_src: aus Bit 4 der current_dec_ctl-Zustandsmaschine decodiert.

– wait_rep: aus Bit 3 der current_dec_ctl-Zustandsmaschine decodiert.

– wait_fp: aus Bit 2 der current_dec_ctl-Zustandsmaschine decodiert.

– wait_next_src_00: aus Bit 1 der current_dec_ctl-Zustandsmaschine decodiert.

– byte1_ahead_avail: Es liegen zwei komprimierte Wortpuffer in decode_borg vor. Die vier Zeiger können auf jegliche der acht Stellen zeigen. Falls byte1_ahead_avail wahr ist, sagt es aus, daß entweder der zweite Zeiger (read_buf_p1) auf den Puffer 0 zeigt und der Puffer 0 gültige Daten in demselben aufweist (used_comp_buf0 = 1), oder daß read_buf_p1 auf den Puffer 1 zeigt und der Puffer 1 gültige Daten in demselben aufweist (used_comp_buf1 = 1).

– byte2_ahead_avail: Es liegen zwei komprimierte Wortpuffer in decode_borg vor. Die vier Zeiger können auf jegliche der acht Stellen zeigen. Falls byte1_ahead_avail wahr ist, sagt es aus, daß entweder der zweite Zeiger (read_buf_p2) auf den Puffer 0 zeigt und der Puffer 0 gültige Daten in demselben aufweist (used_comp_buf0 = 1), oder daß read_buf_p2 auf den Puffer 1 zeigt und der

Puffer 1 gültige Daten in demselben aufweist (used_comp_buf1 = 1).
– byte3_ahead_avail: Es liegen zwei komprimierte Wortpuffer in decode_borg vor. Die vier Zeiger können auf jegliche der acht Stellen zeigen. Falls byte1_ahead_avail wahr ist, sagt es aus, daß entweder der zweite Zeiger (read_buf_p3) auf den Puffer 0 zeigt und der Puffer 0 gültige Daten in demselben aufweist (used_comp_buf0 = 1), oder daß read_buf_p3 auf den Puffer 1 zeigt und der Puffer 1 gültige Daten in demselben aufweist (used_comp_buf1 = 1).

Datenflußsteuersignale:

– decomp_eoi_b: Endes des Bildes (eoi) wird verwendet, um in Flußrichtung nachgelagerten Komponenten zu signalisieren, daß das aus der Dekomprimierungseinrichtung kommende aktuelle Pixel das letzte Pixel für einen Streifen oder ein Bild ist. Dieses Flag wird gesetzt, wenn wir uns an dem letzten Pixel einer Zeile befinden (kstate = 1), wir uns auf der letzten Reihe eines Streifens befinden (jstate = 1) und wir Daten transferieren (decomp_xfer = 1).
– decomp_eol_b: Dies ist das interne Ende eines Zeilensignals. Es wird aktiviert, wenn sich der Zeilenzählerkstate am Zählwert von eins befindet. Somit ist dieses Signal nicht mit einem im Gange befindlichen Transfer qualifiziert, so daß es immer dann wahr ist, wenn sich der Zählwert bei eins befindet.
– input_available: Dieses Signal sagt uns, daß current_pointer auf gültige Daten zeigt. Es ist immer dann wahr, wenn einer der Eingangspuffer voll ist. used_comp_buf0 für den Puffer 0 und used_comp_buf1 für den Puffer 1 geben dies an.
– xfer_from_dma: Dieses Signal sagt uns, daß wir ein Wort aus der DMA-Einheit lesen.
– xfer_from_lb: Dieses Signal sagt uns, daß wir ein Wort aus dem Zeilenpuffer lesen.
– comp_buf_full: Dieses Signal sagt uns, daß sowohl Puffer 0 als auch Puffer 1 voll sind.
– dma_rd_req_b: Dies ist die Anforderung nach Daten an die DMA-Einheit. Es ist immer dann wahr, wenn comp_buf_full nicht wahr ist, was einen leeren Puffer angibt.
– lb_rd_req_b: Dies ist die Anforderung von Daten von dem Zeilenpuffer. Sie wird jedesmal aktiviert, wenn wir ein dekomprimiertes Byte hinaustransferieren, da wir uns für jedes Dekomprimierungsbyte selbst auf der Zeile um eine Position nach vorne inkrementieren und somit einen neuen Zeilenpufferwert für die nächste Position benötigen. Wir wissen, daß wir ein dekomprimiertes Byte hinaustransferieren, wenn decomp_ack_b aktiviert ist.
– northeast_pixel: Das nordöstliche Pixel ist das Pixel, das sich an der Position current_pointer plus 1 in der Keimreihe befindet. Dieses Pixel kommt als lb_pixel_p1 in decode_borg hinein.
– decomp_ack: decomp_ack ist der Quittungsaustausch an die in Flußrichtung nachgelagerte Einheit, daß ein dekomprimiertes Pixel verfügbar ist. Da wir in einem Verriegelungsschritt zwischen dem Zeilenpuffer, der Eingabe von dem DMA (für Literale) und der Ausgabe laufen müssen, wird das Tempo für die meisten Transfers durch decomp_ack bestimmt. Falls decomp_ack nicht wahr ist, enden die meisten Dinge in decomp_borg. Decomp_ack wird ferner verwendet, um zu verhindern, daß manche Eckfälle auftreten. Wenn unser Tempo stark durch den eingegebenen komprimierten Datenstrom bestimmt wird (in bezug darauf, wie schnell die Einheit die Daten ausgeben kann, kommen Eingangsdaten sehr langsam herein), können Eckfallprobleme auftreten. Ein Beispiel liegt dort vor, wo die letzte Literalnachbildung eines Streifens eines Nachbildungszählwerts 0xFF das letzte Byte eines Wortes ist. Man nehme ferner an, daß dieser Zählwert vollständig gelöscht wird, bevor das nächste Wort eingebracht wird. Dies bedeutet, daß das Ende des Bildes (decomp_eoi_b) mit dem letzten Byte dieses Zählwerts auftritt. Jedoch zeigt current_pointer nicht auf das letzte Byte des Streifens, das ein End-0x00 ist. Zu dem Zeitpunkt, zu dem das nächste Wort eingebracht wird und wir sehen, daß der Zählwert 0x00 ist, ein Endzählwert für den letzten Transfer, ist es bereits zu spät. Wir transferieren keine Daten mehr hinaus, so daß decomp_eoi_b nicht stattfinden kann und wir die Chance, es zu aktivieren, vollständig verpassen. Um dieses Problem zu umgehen, benötigen wir, daß wir einen vollen Puffer aufweisen (input_available = 1) und daß wir zumindest zwei gültige Bytes aufweisen (byte1_ahead_avail = 1). Dies deckt den Fall ab, bei dem input_available aufgrund lediglich eines vollen Puffers wahr ist, und bei dem wir uns an dem letzten Byte des Puffers befinden, wobei der andere Puffer noch nicht voll ist. Diese Bedingung könnte zu dem verpasten Ende des Bildes führen, wie es oben beschrieben wurde. In diesem Fall deaktivieren wir decomp_ack, bis das nächste Wort verfügbar ist. Wenn decomp_ack nicht wahr ist, sind wir nicht in der Lage, Daten zu transferieren und würden daher das letzte Byte erst transferieren, wenn das nächste Wort verfügbar wäre. Wenn das nächste Wort verfügbar ist, würden wir sehen, daß das nächste Byte ein End-0x00 ist, und würden daher das Ende des Bildes korrekt aktivieren. Decomp_ack besteht auf zwei Teilen:
– Für den Literalteil prüfen wir, daß input_available wahr ist und entweder byte1_ahead_avail wahr ist oder eines von eoi-Pufferflags (eoi0_flag oder eoil_flag) wahr ist, wie oben beschrieben wurde. Falls diese Bedingungen nicht erfüllt sind, können wir immer noch decomp_ack aktivieren, falls dma_hold_eoi wahr ist. Man nehme an, daß das letzte Byte eines Wor-

tes ein Literal eines Bytes ist und das Pixel nicht von der Stelle 00 stammt und daß dieser Befehl geladen ist, decomp_ack jedoch aufgrund einer Ausgabetempobestimmung weggeht. In diesem Fall wird dma_hold_eoi aktiviert, um die Tatsache anzugeben, daß das letzte Byte der Datei gelesen wurde, aber noch nicht ausgegeben wurde. Wenn als nächstes decomp_ack aktiviert wird, setzen wir decomp_eoi auf korrekte Weise an der nächsten Pixelausgabe, was das einzige Literal von dem letzten Befehl ist. Es gibt mehrere andere Signale, die bezüglich des Literalteils geprüft werden. Wir stellen sicher, daß wir uns in einem Literalmodus befinden (Lauf = 0 und rep_in_progress = 1), und wir stellen sicher, daß lb_rd_ack, die Bestätigung an die Zeilenpufferlogik, aktiviert ist. Falls der Zeilenpuffer das Ausgangsbyte nicht speichern kann, halten wir die Dekomprimierungseinrichtung davon ab, das Byte an die in Flußrichtung nachgelagerte Einheit auszugeben, so daß die nachgelagerte Einheit und der Zeilenpuffer weiterhin synchron bleiben.

– Für den Lauf- und Keimreihenteil prüfen wir, daß wir uns entweder in einem Lauf (Lauf = 1 und rep_in_progress = 1) oder in einer Keimreihenkopie (src_in_progress = 1) befinden. Wir prüfen wiederum, daß lb_rd_ack wahr ist, genauso wie wir es für den Literalfall taten. Schließlich stellen wir sicher, daß entweder input_available oder dma_hold_eoi wahr ist. Wenn input_available wahr ist, sagt uns dies, daß wir uns am Ende der aktuellen Keimreihenkopie oder Laufnachbildung in einen anderen Zählwert oder einen anderen Befehl bewegen können. dma_hold_eoi sagt uns, daß die aktuelle Keimreihenkopie oder Nachbildung die letzte Aktion des Bildes ist und daß, wenn der Zählwert ausläuft, wir uns keine Sorgen darüber machen sollten, daß wir kein weiteres Wort mehr haben (input_available = 0), da die aktuelle Aktion die letzte für das Bild erforderliche Aktion ist.

– decomp_xfer: Dieses Signal sagt uns, daß wir derzeit Daten an die nachgelagerte Einheit transferieren. Die nachgelagerte Einheit fordert Daten an, indem sie decomp_req aktiviert, und wir sagen der nachgelagerten Einheit, daß wir verfügbare Daten haben, indem wir decomp_ack aktivieren. Falls beide wahr sind, wird angenommen, daß ein Byte transferiert wird.

– lb_wr_req_b: Dieses Signal ist exakt dasselbe wie decomp_ack (oben beschrieben). Der Grund dafür, daß es dasselbe ist, besteht darin, daß wir die Daten, die zu dem Zeilenpuffer gehen, und die Daten, die zu der nachgelagerten Einheit gehen, synchron halten, und dies ist eine einfache Lösung.

– lb_wr_xfer: Dieses Signal sagt uns, daß wir derzeit Daten an den Zeilenpuffer transferieren. Der Zeilenpuffer sagt, daß er Daten empfangen kann, indem er lb_wr_ack aktiviert. Wir sagen dem Zeilenpuffer, daß wir verfügbare Daten aufweisen, indem wir lb_wr_req aktivieren. Falls beide wahr sind, wird angenommen, daß ein Byte transferiert wird.

– current_byte_wr: Dieses Signal stellt die aktuelle Stelle in einer Zeile dar (kstate).

– code_rep_done: Dieses Signal wird verwendet, um zu bestimmen, wann eine Lauf- oder eine Literalersetzung abgeschlossen ist. Dieser Abschluß könnte entweder für einen Anfangs- oder einen Folgetransfer und für Zählwerte, die gleich oder größer als eins sind, gelten. Wir stellen sicher, daß wir eine Nachbildung durchführen, indem wir prüfen, daß rep_in_progress wahr ist und entweder rep_orig_m1 gleich 0, für Zählwerte von eins, oder rep_count = 0x01, für Zählwerte, die größer sind als eins, gilt. Wie wir später sehen werden, wird rep_count erst mit dem Zählwert aktualisiert, wenn der erste Transfer stattfindet. Somit führen wir eine Prüfung bezüglich des Falls eines Zählwerts von eins separat durch. Schließlich halten wir nach decomp_xfer Ausschau, um sicherzustellen, daß wir tatsächlich das letzte Byte hinaustransferieren.

– code_src_done: Dieses Signal wird verwendet, um zu bestimmen, wann eine Keimreihenkopie abgeschlossen ist. Dieser Abschluß könnte entweder für einen Anfangs- oder einen Folgetransfer und für Zählwerte, die gleich oder größer als eins sind, gelten. Wir stellen sicher, daß wir eine Nachbildung durchführen, indem wir prüfen, daß src_in_progress wahr ist und entweder src_orig_m1 gleich 0, für Zählwerte von eins, oder src_count = 0x01, für Zählwerte, die größer sind als eins, gilt. Wie wir später sehen werden, wird src_count erst mit dem Zählwert aktualisiert, wenn der erste Transfer stattfindet. Somit müssen wir eine Prüfung bezüglich des Falls eines Zählwerts von eins separat durchführen. Schließlich halten wir nach decomp_xfer Ausschau, um sicherzustellen, daß wir tatsächlich das letzte Byte hinaustransferieren.

– run: Dieses Signal sagt uns, daß der aktuelle Befehl entweder ein Lauf oder ein Literal ist. Man erinnere sich, daß eine Keimreihenkopie, die uns zum Ende der Zeile bringt, als ein Lauf codiert ist. Falls read_command, der Zeiger, der uns sagt, welches Befehls-Ping-Pong-Register wir betrachten sollen, nicht aktiviert ist (ist gleich 0), wird ein Lauf dem höchstwertigen Bit des Befehl0-Registers zugewiesen. Falls read_command aktiviert ist (gleich 1 ist), wird ein Lauf dem höchstwertigen Bit des Befehll-Registers zugewiesen.

– location (Stelle): Dieses Signal sagt uns, wo das erste Pixel des Laufs oder Literals erfaßt werden soll. Der Wert der Stelle ist im Abschnitt 1.0 (der Übersicht) beschrieben. Falls read_command, der Zeiger, der uns sagt, welches Befehls-Ping-Pong-Register wir betrachten sollen, nicht aktiviert ist (ist gleich 0), wird ein Lauf den

Bits (6:5) des Befehl0-Registers zugewiesen. Falls read_command aktiviert ist (gleich 1 ist), wird ein Lauf den Bits (6:5) des Befehll-Registers zugewiesen.

– src_ini: Dieses Signal sagt uns, was der Anfangskeimreihenkopiezählwert für den aktuellen Befehl sein sollte. Falls read_command, der Zeiger, der uns sagt, welches Befehls-Ping-Pong-Register wir betrachten sollen, nicht aktiviert ist (ist gleich 0), wird ein Lauf den Bits (4:3) des Befehl0-Registers zugewiesen. Falls read_command aktiviert ist (gleich 1 ist), wird ein Lauf den Bits (4:3) des Befehll-Registers zugewiesen.

– next_src_ini: Dieses Signal greift vor auf den nächsten Befehlspuffer und sagt uns, was der Anfangskeimreihenkopiezählwert für diesen Befehl sein wird. Falls read_command, der Zeiger, der uns sagt, welches Befehls-Ping-Pong-Register wir betrachten sollen, nicht aktiviert ist (ist gleich 0), wird ein Lauf den Bits (4:3) des Befehll-Registers zugewiesen. Falls read_command aktiviert ist (gleich 1 ist), wird ein Lauf den Bits (4:3) des Befehl0-Registers zugewiesen.

– next_location: Dieses Signal sagt uns, wo das erste Pixel des nächsten Befehls erfaßt werden soll. Der Wert der Stelle ist im Abschnitt 1.0 (der Übersicht) beschrieben. Falls read_command, der Zeiger, der uns sagt, welches Befehls-Ping-Pong-Register wir betrachten sollen, nicht aktiviert ist (ist gleich 0), wird ein Lauf den Bits (6:5) des Befehll-Registers zugewiesen. Falls read_command aktiviert ist (gleich 1 ist), wird ein Lauf den Bits (6:5) des Befehl0-Registers zugewiesen.

– dec_rep_count: Dieses Signal sagt uns, wann unsere Bytezeiger während eines Literal- oder Lauftransfers inkrementiert werden sollen. Die Zeiger werden jedesmal inkrementiert, wenn ein Transfer stattfindet und wir uns in einer Nachbildung befinden. Der Name wird von der Tatsache abgeleitet, daß wir, während wir den Bytezeiger inkrementieren, den Nachbildungszähler dekrementieren.

– src_rep_count: Dieses Signal sagt uns, wann unsere Bytezeiger während eines Keimreihenkopietransfers inkrementiert werden sollen. Die Zeiger werden jedesmal inkrementiert, wenn ein Transfer stattfindet und wir uns in einer Keimreihenkopie befinden. Der Name wird von der Tatsache abgeleitet, daß wir, während wir den Bytezeiger inkrementieren, den Keimreihenkopiezähler dekrementieren.

– eol_approach: Dieses Signal ist wahr, wenn der nächste Keimreihenzählwert gleich der Anzahl von Bytes ist, die auf einer Zeile übrig sind, und wenn wir uns in einem Keimreihenkopiemodus befinden. Dieses Signal wird durch eol_approach_d registriert, bis das Ende der Zeile erreicht ist. Wir wissen, ob wir uns in einer Keimreihenkopie befinden, wenn src_in_progress wahr ist. Wir prüfen, daß entweder command_loaded wahr ist oder daß src_sub_loaded wahr ist. Beide diese Übernahmesignale sind lediglich wahr, bis der erste Transfer von denselben stattfindet. Während der Zeit, während der eines dieser Übernahmesignale wahr ist, prüfen wir, ob src_orig gleich kstate, der verbleibenden Anzahl von Bytes in der Zeile, ist. Falls sie gleich sind und die anderen Übernahmesignale wahr sind, wird dieses Signal aktiviert.

Register:

[0085] Es gibt viele Register in decode_borg, die verwendet werden, um den Fortschritt der Dekomprimierung zu verfolgen, entweder durch ein Halten von benötigten Werten, Zählwerten oder Steuerflags. Da diese Register durch die Prozesse, in denen sie sich befinden, ziemlich sauber herauskristallisiert werden, wird der Prozeßname unten als Abschnittstitel verwendet.

Eingaberegister:

– current_ptr: Dieses Registyer ist ein Zwei-Bit-Wert, der uns sagt, welches Byte in einem der Befehlspuffer sich am vorderen Ende der Schlange befindet; d.h. welches Byte als nächstes verwendet werden soll. In vielen der obigen Beschreibungen wurde dieses Signal (der Klarheit halber) als current_pointer bezeichnet.

– current_ptr_p1: Dieses Register ist ein Zwei-Bit-Wert, der uns sagt, welches Byte an zweiter Stelle steht, hinter dem Byte, auf das durch current_ptr gezeigt wird.

– current_ptr_p2: Dieses Register ist ein Zwei-Bit-Wert, der uns sagt, welches Byte an dritter Stelle steht, hinter dem Byte, auf das durch current_ptr_p1 gezeigt wird.

– current_ptr_p3: Dieses Register ist ein Zwei-Bit-Wert, der uns sagt, welches Byte an vierter Stelle steht, hinter dem Byte, auf das durch current_ptr_p2 gezeigt wird.

– load_comp_buf: Dieses Einbitregister zeigt auf den nächsten Puffer, der mit den Eingangsdaten geladen werden sollte (komprimierter Strom). Dieses Bit schaltet jedesmal um, wenn ein Transfer von dem DMA stattfindet (xfer_from_dma = 1).

– read_comp_buf: Dieses Einzelbitregister zeigt auf den Puffer, der verwendet werden sollte, um aus demselben Daten herauszuziehen. Der Puffer schaltet um, wenn eine von zwei Bedingungen vorliegt. Die erste liegt vor, wenn ein Ende eines Bildes angetroffen wird. Da alle Bilder an wortausgerichteten Grenzen starten, springen wir sofort zu dem nächsten Puffer, wenn ein eoi auftritt, so daß wir für den nächsten Strom von Eingangsdaten für den nächsten Streifen oder das nächste Bild richtig positioniert sind. Es gibt jedoch einen

Fall, bei dem dies nicht auftreten sollte. Falls das letzte Byte des komprimierten Stroms sich an der Stelle 0 befindet und es ein End-0x00 ist, sollten wir read_comp_buf nicht umschalten, auch wenn sich das letzte Byte nicht in dem Puffer befindet, auf den durch read_comp_buf gezeigt wird. Dies liegt daran, daß eoi aktiviert wurde, als das letzte Byte des Puffers, auf den durch read_comp_buf gezeigt wurde, gelesen wurde, d.h. eoi wird aktiviert, bevor wir das tatsächliche letzte Byte betrachten. In diesem Fall schalten wir read_comp_buf nicht um, da sich der nächste Befehl in demselben Puffer befinden wird, auf den read_comp_buf zeigt. Der zuletzt verwendete Puffer (der das End-0x00 enthält, das niemals tatsächlich gelesen wird) ist der Puffer, auf den nicht durch read_comp_buf gezeigt wird. Die zweite Bedingung liegt auf der Basis der aktuellen Stelle des aktuellen und des inkrementierten Übernahmesignals, das auftritt, vor. Falls current_ptr auf das Byte vier des aktuellen Befehlspuffers zeigt und jegliche der Inkrementübernahmesignale auftreten (inc_ptr_1, inc_ptr_2, inc_ptr_3 oder inc_ptr_4), schaltet read_comp_buf um. Desgleichen gilt, daß, falls current_ptr auf Byte 3 zeigt und inc_ptr_2 oder inc_ptr_3 oder inc_ptr_4 vorliegt, read_comp_buf umschaltet. Ähnliche Ausdrücke liegen für current_ptr vor, der ebenfalls auf eines der ersten beiden Bytes zeigt.

– comp_buf0: Dies ist ein 32-Bit-Register, das ein Wort von Daten eines komprimierten Stroms hält. Es ist geladen, falls ein Transfer von dem DMA stattfindet (xfer_from_dma = 1) und der Komprimierte-Daten-Pufferzeiger auf Puffer 0 zeigt (load_comp_buf = 0).

– comp_buf1: Dies ist ein 32-Bit-Register, das ein Wort von Daten eines komprimierten Stroms hält. Es ist geladen, falls ein Transfer von dem DMA stattfindet (xfer_from_dma = 1) und der Komprimierte-Daten-Pufferzeiger auf Puffer 0 zeigt (load_comp_buf = 0).

– used_comp_buf0: Dieses Flag sagt uns, daß der erste eingegebene Komprimierte-Daten-Puffer gültige Daten in demselben zur Verwendung durch die Dekomprimierungseinrichtung aufweist. Es wird gesetzt, wenn wir den ersten Puffer laden (load_comp_buf = 0) und wenn wir Daten von dem DMA transferieren (xfer_from_dma = 1). Entweder ein Ende eines Bildes oder ein Datentransfer, der den Puffer leert, löscht das Flag. Wenn wir aus dem ersten Puffer lesen (read_comp_buf = 0) und auf das letzte Byte des Puffers zeigen (current_ptr = 11) und wir einen beliebigen Betrag (1, 2, 3 oder 4) inkrementieren, brauchen wir alle Daten in dem Puffer 0 auf und sollten ihn somit als nicht gebraucht (leer) markieren. Wenn wir auf das dritte Byte in dem Puffer zeigen (current_ptr = 10) und wenn wir um mehr als eins inkrementieren, sollten wir den Puffer als nicht gebraucht markieren. Desgleichen gilt, daß, wenn wir auf ein Byte zeigen und wir mehr als zwei inkrementieren, wir den Puffer als leer markieren sollten. Eine ähnliche Logik gilt für current_ptr = 00. Ein Löschen dieses Flags auf der Basis eines Endes eines Bildes beinhaltet einen Eckfall. Falls sich das letzte Byte des komprimierten Stroms an der Stelle 0 befindet und ein End-0x00 ist, löschen wir used_comp_buf0, obwohl wir tatsächlich niemals daraus lesen werden. Dies liegt daran, daß eoi aktiviert wurde, als das letzte Byte des anderen Puffers gelesen wurde, d.h. eoi aktiviert wird, bevor wir das tatsächliche letzte Byte betrachten. In diesem Fall löschen wir trotzdem das gebrauchte Flag, da der nächste Befehl in dem nächsten Puffer wortausgerichtet sein wird (vorzugsweise sind alle komprimierten Datenströme zu Beginn wortausgerichtet).

– used_comp_buf1: Dieses Flag sagt uns, daß der zweite eingegebene Komprimierte-Daten-Puffer in demselben gültige Daten zur Verwendung durch die Dekomprimierungseinrichtung aufweist. Die Logik für dieses Flag ist genau dieselbe wie die des Flags für used_comp_buf0, mit der Ausnahme, daß wir read_comp_buf1 und load_comp_buf1 betrachten.

– read_buf_p1: Dieses Flag sagt uns, auf welchen der beiden Komprimiertes-Wort-Puffer der current_ptr_p1-Zeiger zeigt. Dieses Flag schaltet immer dann um, wenn ein Inkrementwert auftritt, der größer ist als die Position des current_ptr_p1-Zeigers. Falls current_ptr_p1 beispielsweise auf 10 zeigt und der Zeigerinkrementwert größer ist als 2, sollten wir dieses Flag umschalten, um anzuzeigen, daß wir nun auf 00 (in dem nächsten komprimierten Wort) zeigen. Wir schalten dieses Flag ferner um, wenn ein Ende eines Bildes vorliegt und derselbe Eckfall, der oben für read_comp_buf beschrieben wurde, vorliegt.

– read_buf_p2: Dieses Flag sagt uns, auf welchen der beiden Komprimiertes-Wort-Puffer der current_ptr_p1-Zeiger zeigt. Dieses Flag schaltet immer dann um, wenn ein Inkrementwert auftritt, der größer ist als die Position des current_ptr_p1-Zeigers. Falls current_ptr_p1 beispielsweise auf 10 zeigt und der Zeigerinkrementwert größer ist als 2, sollten wir dieses Flag umschalten, um anzuzeigen, daß wir nun auf 00 (in dem nächsten komprimierten Wort) zeigen. Wir schalten dieses Flag ferner um, wenn ein Ende eines Bildes vorliegt und derselbe Eckfall, der oben für read_comp_buf beschrieben wurde, vorliegt.

– read_buf_p3: Dieses Flag sagt uns, auf welchen der beiden Komprimiertes-Wort-Puffer der current_ptr_p1-Zeiger zeigt. Dieses Flag schaltet immer dann um, wenn ein Inkrementwert auftritt, der größer ist als die Position des current_ptr_p1-Zeigers. Falls current_ptr_p1 beispielsweise auf 10 zeigt und der Zeigerinkrementwert größer ist als 2, sollten wir dieses Flag umschalten, um anzuzeigen, daß wir nun auf 00 (in

dem nächsten komprimierten Wort) zeigen. Wir schalten dieses Flag ferner um, wenn ein Ende eines Bildes vorliegt und derselbe Eckfall, der oben für read_comp_buf beschrieben wurde, vorliegt.
– eoi0_flag: Dieses Flag wird gesetzt, falls comp_buf0 das letzte Byte des Streifens enthält. Dies wird erfaßt, falls ein DMA-Transfer stattfindet (xfer_from_dma = 1), der DMA signalisiert, daß der aktuelle Transfer der letzte ist (dma_eoi = 1) und wir den Puffer 0 laden (load_comp_buf = 0). Dieses Flag wird gelöscht, wenn decomp_eoi_b stattfindet und das Bit gesetzt ist. Decomp_eoi_b ist das Flag von Delta zu den nachgelagerten Einheiten, das das Ende des Bildes signalisiert. Es fällt mit dem letzten Byte des Streifens/Bildes zusammen.
– eoil_flag: Dieses Flag wird gesetzt, falls comp_buf1 das letzte Byte des Streifens enthält. Dies wird erfaßt, falls ein DMA-Transfer stattfindet (xfer_from_dma = 1), der DMA signalisiert, daß der aktuelle Transfer der letzte ist (dma_eoi = 1) und wir den Puffer 0 laden (load_comp_buf = 1). Dieses Flag wird gelöscht, wenn decomp_eoi_b stattfindet und das Bit gesetzt ist. Decomp_eoi_b ist das Flag von Delta zu den nachgelagerten Einheiten, das das Ende des Bildes signalisiert. Es fällt mit dem letzten Byte des Streifens/Bildes zusammen.
– last_byte_eoi: Dies ist ein Zwei-Bit-Vektor, der das DMA-Signal last_byte_out in ein Signal übersetzt, das verwendet werden kann, um zu bestimmen, wie viele Bytes ausgesendet werden sollen. last_byte_eoi ist einfach die beiden mindestwertigen Bits des programmierten Bytezählwerts. Wir müssen jedoch wissen, welches Byte des letzten Wortpuffers das letzte Byte des komprimierten Datenstroms ist, und dies ist nicht dasselbe, wie die Byteausrichtung der Eingangsdatengröße zu kennen. Die Abbildung ist nachfolgend gezeigt

last_byte_out	last_byte_eoi
00	11
01	10
10	01
11	00

– dma_hold_eoi: Dieses Flag wird gesetzt, wenn eine Keimreihenkopie oder ein Lauf vorliegt, wo wir viele Zyklen, bevor das letzte Byte tatsächlich hinaustransferiert wird, das letzte Byte lesen. Es wird ferner gesetzt, wenn sich das letzte Byte in einem Literalbefehl befindet, der kein explizites (Stelle = 00) erstes Byte aufweist. Dieses Flag wird gelöscht, falls es gesetzt ist und decomp_eoi_b auftritt. Es wird gesetzt, wenn das eoi-Flag des Puffers, aus dem wir lesen, gesetzt ist, und die Datenmenge, die wir lesen, die letzten Daten sein werden. Falls beispielsweise eoi0_flag gesetzt ist, current_ptr_p3 gleich der last_byte_eoi-Position ist, read_buf_p3 ebenfalls auf Puffer 0 zeigt, wissen wir, daß current_ptr_p3 auf das letzte Byte zeigt. Falls inc_ptr_4 auftritt, wissen wir, daß wir gerade alle Daten in dem Streifen/Bild gelesen haben und daher das dma_hold_eoi-Flag setzen sollten.
– decomp_leoi_error_b: Dieses Flag wird gesetzt, wenn wir eine Keimreihenkopie durchführen und der src_count nicht 0 ist, oder wenn wir eine Nachbildung durchführen und der rep_count nicht 0 ist und wir decomp_eoi_b aktivieren.

Diverse Register:

– west_pixel: Das westliche Pixel ist das Pixel zur Linken des aktuellen Pixels, das gerade ausgegeben wird, wenn es von einer Perspektive einer physischen Seite betrachtet wird. Das westliche Pixel wird bei jedem Transfer aus einem anderen dekomprimierten Byte hinaus (decomp_xfer = 1) mit dem aktuellen Pixel aktualisiert.
– cache_pixel: Dieses Pixel stellt das letzte verwendete Nicht-Keimreihenkopiepixel dar. Es wird mit dem aktuellen Pixel aktualisiert, falls decomp_xfer wahr ist und wir keine Keimreihenkopie durchführen (src_in_progress = 0).
– first_pixel: Dieses Pixel stellt das erste Pixel der nächsten Lauf- oder Literalersetzung dar. Der zum Laden dieses Bytes verwendete Zeiger hängt davon ab, welches Ladeübernahmesignal aktiviert ist. Falls ld_next_fb_src aktiviert ist, wissen wir, daß wir auf das erste Pixel des Literals oder Laufs zeigen, da ld_next_fb_src aktiviert ist, wenn wir eine Keimreihenkopie beenden und uns für eine Nachbildung bereit machen. Für dieses Übernahmesignal speichern wir also einfach das Byte, auf das durch current_ptr gezeigt wird, current_byte, in first_pixel.
– Falls ld_first_byte aktiviert ist, was vorliegt, wenn wir uns im Ruhezustand befinden und auf den nächsten Befehl und das mögliche erste Pixel warten, laden wir byte1_ahead in first_pixel. Wir verwenden byte1_ahead, da current_ptr auf den nächsten Befehl zeigt, was bewirkt, daß current_ptr_p1 auf das erste Byte zeigt. Falls ld_next_fb_run aktiviert ist, laden wir auch byte1_ahead. ld_next_fb_run sagt uns, daß wir gerade einen Lauf beenden und einen neuen Befehl und ein erstes Byte laden. Während eines Laufs zeigt current_ptr auf das Byte hinter dem letzten Zählwert, und falls dieses Byte ein neuer Befehl ist, sollte current_ptr_p1 auf das erste Byte zeigen. Wir verwenden ferner byte1_ahead für first_pixel, falls ld_imm_fb wahr ist. Dieses Übernahmesignal sagt uns, daß wir gerade einen Befehl und ein erstes Pixel laden und daß current_ptr auf den nächsten Befehl zeigt, da ld_imm_fb nur wahr ist, wenn ein Literal einer

Länge eins angetroffen wird. In diesem Fall zeigt current_ptr auf eines hinter dem letzten Pixel, das in diesem Fall der nächste Befehl ist. Schließlich verwenden wir byte1_ahead, falls ld_fb_00_src aktiviert ist. In diesem Fall beenden wir eine Keimreihenkopie, diese Keimreihenkopie weist jedoch ein End-0x00 auf. Dies bedeutet, daß current_ptr auf das 0x00 zeigt, was bewirkt, daß current_ptr_p1 auf das erste Byte der folgenden Nachbildung zeigt.

– Wir verwenden byte2_ahead, falls ld_next_fb_lit aktiviert ist. Dieses Übernahmesignal ist aktiviert, falls wir ein Literal beenden und current_ptr auf das letzte Pixel zeigt. Dies bewirkt, daß current_ptr_p1 auf den nächsten Befehl zeigt und daß current_ptr_p2 auf das nächste erste Byte zeigt. Ferner verwenden wir byte2_ahead, falls ld_next_imm_fb aktiviert ist. Wenn dieses Übernahmesignal aktiviert ist, beenden wir eine lange Literalersetzung, wo der abschließende Ersetzungszählwert eins war. current_ptr zeigt auf das letzte Literalbyte, current_ptr_p1 zeigt auf den nächsten Befehl und current_ptr_p2 zeigt auf das nächste erste Byte. Die Aktivierung von ld_next_fb_00_src sagt uns, daß wir gerade eine Keimreihenkopie durchführen, die eine Zeile beendet, und daß wir das einem neuen Befehl zugeordnete nächste erste Byte laden. In diesem Fall zeigt current_ptr auf das End-0x00, zeigt current_ptr_p1 auf den nächsten Befehl und zeigt current_ptr_p2 auf das nächste erste Byte.

– Falls ld_next_fp_00_rep aktiviert ist, wissen wir, daß wir ein neues erstes Byte für einen neuen Befehl laden, da wir eine Nachbildung beendeten, die in einem End-0x00 endete. Falls die Nachbildung ein Literal ist, zeigt current_ptr auf das letzte Pixel, zeigt current_ptr_p1 auf das 0x00, zeigt current_ptr_p2 auf den nächsten Befehl und zeigt current_ptr_p3 auf das nächste erste Pixel. Dies bedeutet, daß wir für diesen Fall byte3_ahead verwenden, um ein erstes Pixel damit zu laden. Falls die Nachbildung ein Lauf ist, zeigt current_ptr auf 0x00, zeigt current_ptr_p1 auf den nächsten Befehl und zeigt current_ptr_p2 auf das nächste erste Pixel. In diesem Fall laden wir first_pixel mit byte2_ahead.

– first_pixel_loaded: Dieses Flag sagt uns, daß wir ein erstes Pixel geladen haben und zu einer Nachbildung schreiten können. Dieses Flag ist gesetzt, falls loading_first_command als wahr gesetzt ist. Es wird an dem ersten Byte der Nachbildung gelöscht (first_pixel_loaded, rep_in_progress und decomp_xfer allesamt wahr), und aufgrund eines unmittelbaren Ladeübernahmesignals laden wir nicht schon ein neues erstes Pixel (loading_first_pixel falsch).

– rep_ini_fp: Dieses Flag verfolgt das Laden jedes Befehls, so daß wir wissen können, wann das erste Anfangsnachbildungsbyte ausgesendet werden soll. Dies ermöglicht es uns, das korrekte erste Pixel auf der Basis der angeforderten Stelle zu senden. Dieses Flag ist gesetzt, wenn ein Befehl geladen wird (loading_command = 1). Es wird gelöscht, sobald das erste Pixel gesendet ist. Dies wird dadurch angegeben, daß sich die Zustandsmaschine in dem write_rep_ini-Zustand befindet und daß decomp_xfer aktiviert ist. Falls code_rep_done wahr ist, löschen wir dieses Flag nicht, weil es bedeutet, daß das nächste Pixel aus einer Nachbildung ein weiteres erstes Pixel sein wird.

Src-Register (Keimreihenkopieregister)

– current_dec_ctl: Die Zustandsmaschine führt eine Flipflop-Schaltung durch.

– cmd0: Dies ist das erste von zwei Registern, die Befehlsbytes enthalten. Wir verwenden hier zwei Register, damit wir vorausarbeiten können, wobei wir den nächsten Befehl laden, während wir immer noch an dem aktuellen Befehl arbeiten. Für jeden der unten abgedeckten Ladefälle stellen wir sicher, daß wir aktuell Puffer 0 laden (load_command = 0), bevor wir tatsächlich den cmd0-Puffer laden. Die zum Laden dieses Registers verwendete Logik ähnelt stark der Logik, die zum Laden eines ersten Pixels verwendet wird.

– Falls ld_cmd wahr ist, befinden wir uns im Ruhezustand, während wir auf gültige Daten warten. Current_ptr zeigt auf das erste gültige Byte, das der nächste Befehl sein wird. Somit verwenden wir current_byte, um cmd0 damit zu laden. Ferner verwenden wir current_byte, falls ld_next_cmd_run wahr ist. Hier führen wir aktuell einen Lauf durch, was bewirkt, daß das erste Pixel auf den nächsten Befehl hinter dem letzten Laufzählwert zeigt. Dieses Byte wird der nächte Befehl sein. Falls ld_imm_cmd wahr ist, verwenden wir auch current_byte. ld_imm_cmd beschreibt den Fall, bei dem wir einen Literalbefehl eines Bytes aufweisen. Falls das geschieht, wissen wir, daß wir den Befehl und das erste (und einzige) Pixel luden, bevor der Befehl ausgeführt wurde. Somit zeigt current_ptr auf den nächsten Befehl. Falls wir den nächsten Befehl laden, der aktuelle Befehl ein Literal ist (ld_next_cmd_lit = 1), wir Initialbytes nachbilden (rep_mux_ctl = 0) und lediglich ein Byte nachzubilden ist (rep_org_m1 = 0), verwenden wir current_byte, um cmd0 zu laden. Dies ist auf die Tatsache zurückzuführen, daß current_ptr bereits hinter dem letzten Pixel des Literals liegt, da lediglich ein Pixel vorlag und es mit dem letzten Befehl geladen wurde.

– Falls ld_next_cmd_lit wahr ist, wir jedoch weder Initialbytes nachbilden noch lediglich ein Byte nachzubilden haben, laden wir cmd0 mit byte1_ahead. In diesem Fall zeigt current_ptr auf das letzte Literal, und current_ptr_p1 zeigt auf das nächste Byte, das der nächste Befehl ist, hinter dem letzten Literal. Falls ld_next_cmd_00_rep

wahr ist und Lauf wahr ist, verwenden wir byte1_ahead, um cmd0 damit zu laden. Hier zeigt current_ptr auf das End-0x00, und current_ptr_p1 zeigt auf das nächste Byte, das ein Befehl ist. Falls ld_next_cmd_00_src wahr ist, verwenden wir byte1_ahead, da current_ptr auf das End-0x00 zeigen wird und current_ptr_p1 auf das nächste Byte, das der nächste Befehl ist, zeigen wird. Schließlich verwenden wir byte1_ahead, falls ld_next_imm_cmd aktiviert ist. Dieses Übernahmesignal ist aktiviert, falls wir einen Folge-Literalzählwert von eins nachbilden und unmittelbar den nächsten Befehl laden. In diesem Fall zeigt current_ptr auf das letzte Literal, was bewirkt, daß current_ptr_p1 auf das nächste Byte, das der nächste Befehl ist, hinter dem letzten Literal zeigt.

– Falls ld_next_cmd_00_rep wahr ist und wir uns in einer Literalersetzung befinden, verwenden wir byte2_ahead. In diesem Fall zeigt current_ptr auf das letzte Pixel, zeigt current_ptr_p1 auf das End-0x00 und zeigt current_ptr_p2 auf den nächsten Befehl.

– command_loaded: Dieses Flag sagt uns, daß wir einen gültigen Befehl laden lassen. Dieses Bit ist gesetzt, falls wir aktuell einen neuen Befehl laden (loading_command aktiviert). Dieses Bit wird gelöscht, falls es gesetzt ist und decomp_xfer auftritt, was das erste Byte signalisiert, das für diesen aktuellen Befehl dekomprimiert wird. Jedoch schleusen wir dies mit loading_command, was nicht aktiviert sein sollte. Auf diese Weise stellen wir sicher, daß wir uns nicht in einem Eckfall befinden, in dem wir antiparallele Befehle haben, und während wir den nächsten Befehl laden, bewirkt der aktuelle Befehl, daß ein Byte durch die Dekomprimierungseinrichtung ausgegeben wird. In diesem Fall halten wir command_loaded wahr, obwohl wir gerade das erste (und in diesem Fall einzige) Byte aussenden, da wir zur selben Zeit einen neuen Befehl laden.

– src_sub0: Dies ist das erste von zwei Ping-Pong-Registern, die die Menge an Zählwerten verfolgen, die durch den Folgezählwert benötigt werden. Falls ld_src_subsqnt wahr ist, was angibt, daß gerade ein Folgezählwert geladen wird, und load_src_sub 0 ist, was angibt, daß wir gerade src_sub0 laden sollten, ist dieses Register damit geladen, was sich in current_byte befindet.

– src_sub_loaded: Dieses Flag sagt uns, daß wir einen Folgezählwert geladen haben. Es wird auf wahr gesetzt, wenn ld_src_subsqnt auftritt, und wird gelöscht, sobald wir das erste Keimreihenkopiebyte ausgesendet haben.

– src_count: Dieser Zähler verfolgt, wie viele Bytes wir aus Keimreihendaten ausgesendet haben. Dieser Zähler wird mit dem Originalkeimreihenkopiezählwert (minus 1) geladen, nachdem das erste Keimreihenkopiebyte in die Flußrichtung gesendet wurde. Wir laden diesen Zähler nicht mit dem ursprünglichen Wert, da der ursprüngliche Wert mit Daten transferiert werden könnte oder auch nicht, und diese Einheit mittels eines Datentransfers mit sich selbst Schritt hält. Somit wird src_count erst geladen, wenn die erste Keimreihenkopie des nächsten Zählwerts vorliegt. Andernfalls dekrementieren wir diesen Zählwert bei jedem Transfer. Dieses Dekrementieren wird durch dec_src_count signalisiert.

– block_00_src_d: Dieses Flag ist die registrierte Version von block_00_src, das auftritt, wenn wir einen Anfangszählwert von 0x03 antreffen, dem keine weiteren Zählwerte folgen, oder wenn wir einen Folgezählwert von 0xff antreffen, dem keine weiteren Zählwerte folgen. Wir halten dies aktiviert, bis eine Nachbildung beginnt oder wir das Ende der Zeile erreichen.

– eol_approach_d: Dieses Flag ist eine registrierte Version von eol_approach, das uns sagt, daß uns eine Keimreihenkopie ganz bis zum Ende der Zeile bringen wird. Dies verhindert, daß wir versuchen, Nachbildungsinformationen zu laden, und ermöglicht uns statt dessen, den nächsten Befehl zu laden.

mux_ctl:

– src_orig: Dieses Register enthält den aktuellen Keimreihenkopiezählwert. Wenn wir uns in einem Folgekeimreihenkopiemodus befinden, enthält dieses Signal src_sub0, falls read_src_sub_0 ist, ansonsten enthält es src_sub1. Falls wir uns in einem Anfangskeimreihenkopiemodus befinden, enthält dieses Signal die Bits (6:5) des cmd0-Registers, falls read_command 0 ist, ansonsten enthält es die Bits (6:5) des cmd1-Registers.

– rep_orig: Dieses Register enthält den aktuellen Nachbildungskopiezählwert. Falls wir uns in einem Folgenachbildungsmodus befinden, enthält dieses Signal rep_sub0, falls read_rep_sub 0 ist, ansonsten enthält es rep_sub1. Falls wir uns in einem Anfangsnachbildungsmodus befinden, enthält dieses Signal eine modifizierte Version der Bits (2:0) des cmd0-Registers, falls read_command 0 ist, ansonsten enthält es eine modifizierte Version der Bits (6:5) des cmd1-Registers. Die Modifikation stammte von der Tatsache, daß der Nachbildungszählwert in dem Befehl eines weniger ist als der tatsächliche Zählwert für Literalersetzungen und zwei weniger ist als der tatsächliche Zählwert für Laufnachbildungen. Das Signal cmd0_p1 ist die Bits (2:0) des cmd0-Registers, wobei eins zu denselben hinzuaddiert ist. Cmd0_p2 ist die Bits (2:0) des cmd0-Registers, wobei 2 zu denselben hinzuaddiert ist.

Mux-Daten (Multiplex-Daten):

– current_byte: Dieses Signal enthält das nächste Byte, das durch die Dekomprimierungseinrichtungslogik verwendet werden soll. Falls

read_comp_buf nicht aktiviert ist, liest diese aus dem ersten Komprimierungspuffer, comp_buf0, ansonsten liest es aus comp_buf1. Welches Byte es liest, wird durch den Zeiger current_ptr diktiert.
– byte1_ahead: Genau dasselbe wie das current_byte, mit der Ausnahme, daß es auf der Basis dessen, wohin current_ptr_p1 zeigt, Daten liest und daß es den Puffer liest, auf den durch read_buf_p1 gezeigt ist.
– byte2_ahead: Genau dasselbe wie das current_byte, mit der Ausnahme, daß es auf der Basis dessen, wohin current_ptr_p2 zeigt, Daten liest und daß es den Puffer liest, auf den durch read_buf_p2 gezeigt ist.
– byte3_ahead: Genau dasselbe wie das current_byte, mit der Ausnahme, daß es auf der Basis dessen, wohin current_ptr_p3 zeigt, Daten liest und daß es den Puffer liest, auf den durch read_buf_p3 gezeigt ist.

Rep-Register (Ersetzungsregister)

– rep_sub0: Dieses Register ist das erste von zwei Ping-Pong-Registern, die verwendet werden, um den Zählwert für Folgenachbildungen zu speichern. Es wird mit current_byte geladen, wenn ld_rep_subsqnt aktiviert ist, Lauf aktiviert ist und load_rep_sub null ist, was anzeigt, daß rep_sub0 das Register ist, das als nächstes geladen werden sollte. Man beachte, daß für einen Lauf current_ptr auf das nächste Byte zeigt, das in diesem Fall der nächste Folgezählwert ist. Falls ld_rep_subsqnt aktiviert ist und wir uns nicht in einem Lauf befinden, laden wir von byte1_ahead. In diesem Fall verwendeten wir byte1_ahead, da current_ptr auf das letzte Literalpixel zeigt, was bewirkt, daß current_ptr_p1 auf den nächsten Zählwert zeigt.
– rep_sub_loaded: Dieses Flag gibt an, daß ein Folgenachbildungszählwert geladen wurde. Es wird dadurch gesetzt, daß ld_rep_subsqnt aktiviert wird, und wird gelöscht, wenn der erste Folgezählwert ausgesendet wird.
– rep_count: Dieses Register enthält den Nachbildungszählwertbetrag. Es wird mit dem rep_orig-Wert minus eins geladen, da es erst geladen wird, wenn der erste Transfer stattfindet. Wie bei src_count wird dieses Register erst geladen, wenn ein Datentransfer stattfindet, so daß wir ohne weiteres eine Synchronisation mit dem Rest der Logik aufrechterhalten können. Falls dec_rep_count stattfindet, wird rep_count um eins dekrementiert.
– block_00_rep_d: Dies ist eine gepufferte Version von block_00_rep, die auftritt, wenn wir einen Anfangszählwert von 7 (Anfang) oder 255 (Folge), gefolgt von keinen weiteren Zählwerten, antreffen. Wir halten dies aktiviert, bis eine Nachbildung beginnt oder wir das Ende der Zeile erreichen.
– run_pixel_d: Dies ist eine registrierte Version des run_pixel, das das erste Pixel eines Laufs ist. Wenn first_pixel_loaded aktiviert ist, laden wir run_pixel_d mit run_pixel. First_pixel ändert sich, nachdem das first_pixel_loaded-Flag deaktiviert ist, so daß run_pixel_d das ist, was wir dem dekomprimierten Byte zu allen anderen Zeitpunkten mit Ausnahme des ersten Pixels eines Laufs zuweisen.

Zustandsmaschine:

[0086] Im Gegensatz zu dem Prozeßnamen enthält dieser Prozeß die nächsten ptr-Zuweisungen für die current_ptr-Zeiger.
– next_ptr: Wenn eoi auftritt, möchten wir, daß current_ptr auf die Stelle 00 zeigt, und so weisen wir next_ptr dem first_byte zu, das 00 ist. Falls inc_ptr_3 auftritt, weisen wir next_ptr dem fourth_byte zu, da wir möchten, daß current_ptr auf 11 zeigt. Falls inc_ptr_2 auftritt, weisen wir next_ptr dem third_byte zu, da wir wollen, daß current_ptr auf 10 zeigt. Schließlich, falls inc_ptr_1 auftritt, weisen wir next_ptr dem second_byte zu, da wir wollen, daß current_ptr auf 01 zeigt.
– next_ptr_p1: Hier wird dieselbe bei next_ptr verwendete Logik verwendet, mit der Ausnahme, daß wir bei eoi next_ptr_p1 dem second_byte zuweisen wollen, da current_ptr_p1 zu Beginn dem current_ptr um eins vorausgehen sollte.
– next_ptr_p2: Hier wird dieselbe bei next_ptr verwendete Logik verwendet, mit der Ausnahme, daß wir bei eoi next_ptr_p2 dem third_byte zuweisen wollen, da current_ptr_p1 zu Beginn dem current_ptr um zwei vorausgehen sollte.
– next_ptr_p3: Hier wird dieselbe bei next_ptr verwendete Logik verwendet, mit der Ausnahme, daß wir bei eoi next_ptr_p3 dem fourth_byte zuweisen wollen, da current_ptr_p1 zu Beginn dem current_ptr um drei vorausgehen sollte.

Abgehende Daten:

– decomp_byte_b: Dies ist der dekomprimierte Datenbus zu der nachgelagerten Einheit. Auf der Basis dessen, in welchem Zustand wir uns aktuell befinden, wählen wir aus, was auszugeben ist.
– Wenn wir uns in dem Anfangskeimreihenkopiezustand befinden (write_src_ini = 1) und wir uns nicht auf der ersten Reihe befinden, weisen wir lb_pixel_decomp_byte_b zu. lb_pixel ist das Datenbyte von dem Zeilenpuffer, das das Keimreihenbyte direkt über unserer aktuellen Zeilenposition darstellt. Falls wir uns auf der ersten Reihe befinden, weisen wir first_row_byte decomp_byte_b zu. first_row_byte ist ein konfigurierbares Register, das uns sagt, was das Erste-Reihe-Byte sein sollte. Dies ermöglicht es uns, eine bessere Komprimierung zu erlangen, da wir vorab einstellen können, was der Hintergrund sein

könnte.
– Falls wir uns in dem Folgekeimreihenkopiezustand befinden (write_src_sub = 1), tun wir genau dasselbe wie für den Fall write_src_ini.
– Falls wir uns in dem Anfangsnachbildungszustand befinden (write_rep_ini = 1), müssen wir mehr Flags prüfen. Falls rep_ini_fp wahr ist, wissen wir, daß wir das erste Pixel, das gemäß der Stelle ausgewählt wird, ausgeben sollten. Falls die Stelle 00 ist, wird es first_pixel sein, falls die Stelle 01 ist, wird es west_pixel sein; falls die Stelle 10 ist, wird es northeast_pixel sein; falls die Stelle 11 ist, wird es cache_pixel sein. Falls rep_ini_fp nicht wahr ist und wir uns nicht in einem Halbtonnachbildungsmodus befinden und wir uns in einem Lauf befinden, wird das ausgegebene Byte run_pixel sein. Wenn wir uns einem Halbtonnachbildungsmodus befinden und Lauf wahr ist, senden wir ht_pixel aus. Falls keine der obigen Bedingungen wahr ist, befinden wir uns in einem Literalmodus. In diesem Fall senden wir current_byte an decomp_byte_b aus.
– Falls wir uns in einem write_rep_sub-Zustand befinden (write_rep_sub = 1) und wir uns nicht in einem Halbtonnachbildungsmodus befinden und wir uns in einem Lauf befinden, ist das Ausgabebyte run_pixel. Falls wir uns in einem Halbtonnachbildungsmodus befinden und Lauf wahr ist, werden wir ht_pixel aussenden. Falls keine der obigen Bedingungen wahr ist, befinden wir uns in einem Literalmodus. In diesem Fall senden wir current_byte an decomp_byte_b aus.
– run_pixel: Dem run_pixel wird auf der Basis der Stelle west_pixel, northeast_pixel, cache_pixel oder first_pixel zugewiesen, falls first_pixel_loaded wahr ist. Nachdem das erste Pixel ausgesendet wurde, wird first_pixel_loaded nicht wahr sein. Nachdem first_pixel_loaded deaktiviert ist, wird dem run_pixel das run_pixel_d zugewiesen.

[0087] Obwohl die Offenbarung in einer Sprache beschrieben wurde, die für strukturelle Merkmale und/oder methodologische Schritte spezifisch ist, versteht es sich von selbst, daß die beigefügten Patentansprüche nicht auf die beschriebenen spezifischen Merkmale oder Schritte beschränkt sind. Vielmehr sind die spezifischen Merkmale und Schritte beispielhafte Formen einer Implementierung dieser Offenbarung. Während beispielsweise rot, grün und blau als Basis für ein Farbdrucken verwendet wurden, versteht es sich, daß ersatzweise auch magenta, zyan und gelb oder andere Farben verwendet werden könnten. Während eine beträchtliche Menge an spezifischen Informationen offenbart wurde, um eine aktivierte Version zu beschreiben, könnten Änderungen an diesen Informationen vorgenommen werden, wobei trotzdem die hierin offenbarten Lehren befolgt werden.

[0088] Während ferner ein oder mehrere Verfahren mittels Flußdiagrammen und mittels eines den Blöcken zugeordneten Textes offenbart wurden, versteht es sich, daß die Blöcke nicht unbedingt in der Reihenfolge, in der sie dargestellt sind, durchgeführt werden müssen und daß eine alternative Reihenfolge zu ähnlichen Vorteilen führen kann.

Patentansprüche

1. Anwendungsspezifische integrierte Schaltung (ASIC) (**202**), die folgende Merkmale aufweist:
eine Direktspeicherzugriffs-(DMA-)Entscheidungseinheit (**304**), um Farbebenendaten syntaktisch zu analysieren, wobei die Daten jeder Farbebene an einen jeder Farbebene zugeordneten DMA (**306**–**310**) gesendet werden;
eine Komprimierungseinrichtung (**312**), um Farbebenendaten von dem jeder Farbebene zugeordneten DMA zu empfangen und um Farbdaten in Läufen, Keimreihenkopien und Literalen zu komprimieren, wodurch komprimierte Daten gebildet werden;
eine Dekomprimierungseinrichtung (**318**), um die komprimierten Daten zu dekomprimieren, wobei die Dekomprimierungseinrichtung folgende Merkmale aufweist:
einen Dekomprimierungseinrichtungskern (**504**), um eine Datendekomprimierung zu leiten; und
eine Zustandsmaschine (**600**), die in dem Dekomprimierungseinrichtungskern (**504**) arbeitet, um Läufe, Keimreihenkopien und Literale zu erkennen und zu dekomprimieren.

2. Anwendungsspezifische integrierte Schaltung (ASIC) (**202**) gemäß Anspruch 1, bei der die Komprimierungseinrichtung (**312**) zusätzlich folgendes Merkmal aufweist:
eine Datenverschachtelungseinrichtung (**404**), um Farbebenendaten in einem Format, das RGBR-, GBRG- und BRGB-Sequenzen aufweist, zu verschachteln.

3. Anwendungsspezifische integrierte Schaltung (ASIC) (**202**) gemäß Anspruch 1 oder 2, bei der die Komprimierungseinrichtung (**312**) folgendes Merkmal aufweist:
eine Komprimierungssteuerung (**422**), um zu bestimmen, ob die Farbebenendaten als Lauf, Keimreihenkopie oder Literal komprimiert werden sollten.

4. Anwendungsspezifische integrierte Schaltung (ASIC) (**202**) gemäß Anspruch 3, bei der die Komprimierungseinrichtung (**312**) ferner folgende Merkmale aufweist:
eine Laufbefehlseinheit (**426**), um Läufe in den Farbebenendaten zu komprimieren;
eine Keimreihenkopieeinheit (**428**), um Keimreihenkopien in den Farbebenendaten zu komprimieren; und
eine Literaleinheit (**430**) zum Verarbeiten von Litera-

len in den Farbebenendaten.

5. Anwendungsspezifische integrierte Schaltung (ASIC) (**202**) gemäß einem der Ansprüche 1 bis 4, bei der die Dekomprimierungseinrichtung (**318**) ferner folgende Merkmale aufweist:
einen Vorgriffspuffer (**506**), um komprimierte Pixeldaten zur Verwendung bei einem Laden von Booleschen Gleichungen innerhalb einer Sequenz von priorisierten Gleichungen in der Zustandsmaschine (**600**) zu enthalten; und
wobei die Dekomprimierungseinrichtung (**318**) konfiguriert ist, um Daten bei einem Pixel pro Taktzyklus der ASIC (**202**) zu dekomprimieren.

6. Anwendungsspezifische integrierte Schaltung (ASIC) (**202**) gemäß einem der Ansprüche 1 bis 5, bei der die Komprimierungseinrichtung (**312**) folgendes Merkmal aufweist:
ein Entropiemodul (**408**, **410**, **412**), das in jeder Farbebene als Indikator einer Komprimierungseffektivität instanziert ist und das Änderungen zwischen sequentiellen Pixeln zählt, um eine Entropie auf einer Zeilenbasis zu messen, und das mehrere Zeilen miteinander mittelt, um den Indikator zu verbreitern.

7. Anwendungsspezifische integrierte Schaltung (ASIC) (**202**), die folgende Merkmale aufweist:
eine Dekomprimierungseinrichtung (**318**), um komprimierte Daten zum Zweck einer Übertragung an eine Druckmaschine (**206**) zu dekomprimieren, wobei die Dekomprimierungseinrichtung folgende Merkmale aufweist:
einen Vorgriffspuffer (**506**), um komprimierte Daten zu enthalten;
eine Zustandsmaschine (**600**), um die komprimierten Daten aus dem Vorgriffspuffer (**506**) in Stellen innerhalb einer Sequenz von priorisierten Gleichungen, die jedem einer Mehrzahl von Zuständen zugeordnet sind, zu laden, wobei jede Gleichung einer Bewegung von einem ersten Zustand (**602**–**614**) zu einem zweiten Zustand (**602**–**614**) zugeordnet ist und wobei Gleichungen innerhalb der Sequenz priorisierter Gleichungen für eine sequentielle Auswertung konfiguriert sind; und
Zustände (**602**–**614**), die in der Mehrzahl von Zuständen enthalten sind, um Daten, die Lauf-, Keimreihenkopie- und Literal-Pixeln zugeordnet sind, zu erkennen.

8. Anwendungsspezifische integrierte Schaltung (ASIC) (**202**) gemäß Anspruch 7, die zusätzlich folgendes Merkmal aufweist:
eine Komprimierungseinrichtung (**312**), um Farbebenendaten, die jeder Farbebene zugeordnet sind, zu empfangen und um Farbdaten in Läufen, Keimreihenkopien und Literalen zu komprimieren.

9. Anwendungsspezifische integrierte Schaltung (ASIC) (**202**) gemäß Anspruch 8, bei der die Komprimierungseinrichtung (**312**) folgendes Merkmal aufweist:
ein Entropiemodul (**408**, **410**, **412**), um eine Komprimierungseffektivität durch ein Zählen von Änderungen zwischen sequentiellen Pixeln zu messen.

10. Anwendungsspezifische integrierte Schaltung (ASIC) (**202**) gemäß Anspruch 8 oder 9, bei der die Komprimierungseinrichtung (**312**) folgende Merkmale aufweist:
eine Datenverschachtelungseinrichtung (**404**), um Farbebenendaten in einem Format, das RGBR-, GBRG- und BRGB-Sequenzen aufweist, zu verschachteln; und
eine Komprimierungssteuerung (**422**), um zu bestimmen, ob Daten, die jedem Pixel der Farbebenendaten zugeordnet sind, als Lauf, Keimreihenkopie oder als Literal komprimiert werden sollten.

11. Anwendungsspezifische integrierte Schaltung (ASIC) (**202**) gemäß Anspruch 10, bei der die Komprimierungseinrichtung (**312**) zusätzlich folgende Merkmale aufweist:
eine Laufbefehlseinheit (**426**), um Läufe zu komprimieren;
eine Keimreihenkopieeinheit (**428**), um Keimreihenkopien zu komprimieren; und
eine Literaleinheit (**430**), um Literale zu verarbeiten.

12. Anwendungsspezifische integrierte Schaltung (ASIC) (**202**) gemäß einem der Ansprüche 7 bis 11, bei der die Zustandsmaschine (**600**) folgende Merkmale aufweist:
einen Ruhezustand (**602**), um auf einen neuen Befehl zu warten;
einen Decodiere-Befehl-Zustand (**604**), um einen aktuellen Befehl zu betrachten;
einen Schreibe-Keimreihenkopie-Anfangszustand (**606**), um eine Anzahl von spezifizierten Anfangskeimreihenkopiepixeln auszuschreiben;
einen Schreibe-Keimreihenkopie-Folgezustand (**608**), um Folgekeimreihenkopiepixel, die in Keimreihenkopiezählwertbytes, die auf einen Befehl folgen, spezifiziert sind, auszuschreiben;
einen Warte-Auf-Erstes-Pixel-Zustand (**610**), um darauf zu warten, das ein erstes Pixel eines Laufs oder eines Literals verfügbar ist;
einen Schreibe-Ersetzung-Anfangszustand (**612**), um Literal- oder Laufpixel auszuschreiben; und
einen Schreibe-Ersetzung-Folgezustand (**614**), um Folge-Literal- oder Laufpixel auszuschreiben.

13. Verfahren zum Komprimieren von Daten, das folgende Schritte aufweist:
Verschachteln von Farbebenendaten;
Bestimmen eines geeigneten Komprimierungstyps aus Läufen, Keimreihenkopien und Literalen;
Puffern von Befehlen, um zu unterstützen, daß ein Laufmodul, ein Keimreihenkopiemodul und ein Literalmodul gleichzeitig betrieben werden; und

Puffern von Literalen, um es dem Literalmodul zu ermöglichen, Literale zur Ausgabe als Stapel zu verarbeiten.

14. Verfahren gemäß Anspruch 13, das zusätzlich folgende Schritte aufweist:
Sequenzieren einer Ausgabe des Keimreihenkopiemoduls und des Literalmoduls, um eine korrekte Pixelreihenfolge aufrechtzuerhalten; und
Ausgeben komprimierter Daten zur Speicherung bis zu einer Dekomprimierung.

15. Verfahren gemäß Anspruch 13 oder 14, das zusätzlich folgenden Schritt aufweist:
Messen einer Entropie, um eine Komprimierungseffektivität zu bestimmen, wobei die Messung durch ein Zählen von Änderungen zwischen sequentiellen Pixeln durchgeführt wird.

16. Anwendungsspezifische integrierte Schaltung (ASIC) (**202**), die eine Logik für folgende Schritte aufweist:
Verschachteln von Farbebenendaten;
Bestimmen eines geeigneten Komprimierungstyps aus Läufen, Keimreihenkopien und Literalen;
Puffern von Befehlen, um zu unterstützen, daß ein Laufmodul, ein Keimreihenkopiemodul und ein Literalmodul gleichzeitig betrieben werden; und
Puffern von Literalen, um es dem Literalmodul zu ermöglichen, Literale zur Ausgabe als Stapel zu verarbeiten.

17. Drucker (**106**), der folgende Merkmale aufweist:
eine Druckmaschineneinrichtung (**206**) zum Erzeugen einer Druckausgabe;
eine Steuerungseinrichtung (**204**) zum Steuern eines Betriebs der Druckmaschine (**206**);
eine Einrichtung zum Verschachteln von Farbebenendaten;
eine Einrichtung zum Bestimmen eines geeigneten Komprimierungstyps aus Läufen, Keimreihenkopien und Literalen;
eine Einrichtung zum Puffern von Befehlen, um zu unterstützen, daß ein Laufmodul, ein Keimreihenkopiemodul und ein Literalmodul gleichzeitig betrieben werden; und
eine Einrichtung zum Puffern von Literalen, um es dem Literalmodul zu ermöglichen, Literale zur Ausgabe als Stapel zu verarbeiten.

18. Drucker (**106**) gemäß Anspruch 17, der ferner eine Einrichtung zum Zuordnen der zu komprimierenden verschachtelten Daten zu Läufen, Keimreihenkopien und Literalen aufweist.

19. Drucker (**106**) gemäß Anspruch 18, der zusätzlich folgendes Merkmal aufweist:
eine Einrichtung zum Überwachen des Laufmoduls, des Keimreihenkopiemoduls und des Literalmoduls, um ein Ausgabesequenzieren zu erleichtern.

20. Drucker (**106**) gemäß Anspruch 18 oder 19, der zusätzlich folgendes Merkmal aufweist:
eine Einrichtung zum Puffern von Literalen, um es dem Literalmodul zu ermöglichen, Literale zur Ausgabe als Stapel zu verarbeiten.

21. Drucker (**106**) gemäß einem der Ansprüche 18 bis 20, der zusätzlich folgendes Merkmal aufweist:
eine Einrichtung zum Messen einer Entropie, um eine Komprimierungseffektivität zu bestimmen, wobei das Messen durch ein Zählen von Änderungen numerischer Farbebenendaten zwischen sequentiellen Pixeln durchgeführt wird.

22. Verfahren (**900**) zum Dekomprimieren von Daten, das folgende Schritte aufweist:
Puffern (**902**) noch nicht dekomprimierter Daten in einem Vorgriffspuffer (**506**);
Laden (**906**) von Booleschen Ausdrücken parallel zu Ladegleichungen, Registern und Signalen, die teilweise auf dem Vorgriffspuffer (**506**) basieren;
Auswerten (**912**) einer Sequenz priorisierter Gleichungen, die die Booleschen Ausdrücke umfassen, wobei die Sequenz einem aktuellen Zustand (**602**–**614**) in einer Zustandsmaschine (**600**) zugeordnet ist; und
Lokalisieren und Dekomprimieren (**914**), durch einen Betrieb von Zuständen in der Zustandsmaschine (**600**), von Instanzen von Läufen, Keimreihenkopien und Literalen.

23. Verfahren (**600**) gemäß Anspruch 22, das zusätzlich folgenden Schritt aufweist:
Durchführen der Auswertung in der Zustandsmaschine (**600**), wobei eine Bewegung zwischen Zuständen (**602**–**614**) auf einer Identifizierung eines ersten wahren Ausdrucks innerhalb der Sequenz priorisierter Gleichungen, die einem aktuellen Pixel zugeordnet sind, basiert.

24. Verfahren (**900**) gemäß Anspruch 22 oder 23, das zusätzlich folgende Schritte aufweist:
Auslösen einer Bewegung zwischen Zuständen (**602**–**614**) durch Finden einer wahren Gleichung innerhalb der Sequenz priorisierter Gleichungen; und
Bewegen zu einem der wahren Gleichung zugeordneten Zustand (**602**–**614**).

25. Verfahren (**900**) gemäß einem der Ansprüche 22 bis 24, bei dem ein Betrieb der Zustandsmaschine (**600**) folgende Schritte aufweist:
Ruhen in einem Ruhezustand (**602**), um auf einen neuen Befehl zu warten;
Decodieren eines aktuellen Befehls in einem Decodiere-Befehl-Zustand (**604**);
Ausschreiben einer Anzahl von spezifizierten Anfangskeimreihenkopiepixeln in einem Schreibe-Keimreihenkopie-Anfangszustand (**606**);

Ausschreiben von Folgekeimreihenkopiepixeln, die in Keimreihenkopiezählwertbytes, die einem Befehl folgen, spezifiziert sind, in einem Schreibe-Keimreihenkopie-Folgezustand (**608**);
Warten darauf, daß ein erstes Pixel eines Laufs oder eines Literals verfügbar ist, innerhalb eines Warte-Auf-Erstes-Pixel-Zustands (**610**);
Ausschreiben von Literal- oder Laufpixeln in einem Schreibe-Ersetzung-Anfangszustand (**612**);
Ausschreiben von Folge-Literal- oder Laufpixeln in einem Schreibe-Ersetzung-Folgezustand (**614**); und
Priorisieren von Gleichungen, die einer Bewegung von einem ersten Zustand zu einem zweiten Zustand in der Zustandsmaschine (**600**) zugeordnet sind.

26. Verfahren (**900**) gemäß einem der Ansprüche 22 bis 25, das zusätzlich folgenden Schritt aufweist:
Ausgeben dekomprimierter Daten, die pro Taktzyklus einem Pixel zugeordnet sind.

27. Anwendungsspezifische integrierte Schaltung (ASIC) (**202**), die eine Logik für folgende Schritte aufweist:
Ruhen in einem Ruhezustand (**602**), um auf einen neuen Befehl zu warten;
Decodieren eines aktuellen Befehls in einem Decodiere-Befehl-Zustand (**604**);
Ausschreiben einer Anzahl von spezifizierten Anfangskeimreihenkopiepixeln in einem Schreibe-Keimreihenkopie-Anfangszustand (**606**);
Ausschreiben von Folgekeimreihenkopiepixeln, die in Keimreihenkopiezählwertbytes, die einem Befehl folgen, spezifiziert sind, in einem Schreibe-Keimreihenkopie-Folgezustand (**608**);
Warten darauf, daß ein erstes Pixel eines Laufs oder eines Literals verfügbar ist, innerhalb eines Warte-Auf-Erstes-Pixel-Zustands (**610**);
Ausschreiben von Literal- oder Laufpixeln in einem Schreibe-Ersetzung-Anfangszustand (**612**);
Ausschreiben von Folge-Literal- oder Laufpixeln in einem Schreibe-Ersetzung-Folgezustand (**614**); und
Priorisieren von Gleichungen, die einer Bewegung von einem ersten Zustand zu einem zweiten Zustand in der Zustandsmaschine (**600**) zugeordnet sind.

28. Drucker (**106**), der folgende Merkmale aufweist:
eine Druckmaschineneinrichtung (**206**) zum Erzeugen einer Druckausgabe;
eine Steuerungseinrichtung (**204**) zum Steuern eines Betriebs der Druckmaschineneinrichtung (**206**);
eine Einrichtung, um die Steuerungseinrichtung (**204**) ruhen zu lassen, um auf einen neuen Befehl zu warten;
eine Einrichtung zum Decodieren eines aktuellen Befehls in einem Decodiere-Befehl-Zustand (**604**);
eine Einrichtung zum Ausschreiben einer von Anzahl spezifizierten Anfangskeimreihenkopiepixeln in einem Schreibe-Keimreihenkopie-Anfangszustand (**606**);
eine Einrichtung zum Ausschreiben von Folgekeimreihenkopiepixeln, die in Keimreihenkopiezählwertbytes, die einem Befehl folgen, spezifiziert sind, in einem Schreibe-Keimreihenkopie-Folgezustand (**608**);
eine Einrichtung zum Warten darauf, daß ein erstes Pixel eines Laufs oder eines Literals verfügbar ist, innerhalb eines Warte-Auf-Erstes-Pixel-Zustands (**610**);
eine Einrichtung zum Ausschreiben von Literal- oder Laufpixeln in einem Schreibe-Ersetzung-Anfangszustand (**612**);
eine Einrichtung zum Ausschreiben von Folge-Literal- oder Laufpixeln in einem Schreibe-Ersetzung-Folgezustand (**614**); und
eine Einrichtung zum Priorisieren von Gleichungen, die einer Bewegung von einem ersten Zustand zu einem zweiten Zustand in der Zustandsmaschine (**600**) zugeordnet sind.

29. Drucker (**106**), der folgende Merkmale aufweist:
eine Druckmaschineneinrichtung (**206**) zum Erzeugen einer Druckausgabe;
eine Steuerungseinrichtung (**204**) zum Steuern eines Betriebs der Druckmaschineneinrichtung (**206**);
eine Einrichtung zum Puffern noch nicht dekomprimierter Daten in einem Vorgriffspuffer (**506**);
eine Einrichtung zum Laden einer Sequenz priorisierter Gleichungen für einen aktuellen Zustand in einer Zustandsmaschine (**600**), teilweise unter Verwendung von Vorgriffsdaten;
eine Einrichtung zum Untersuchen (**318**) eines aktuellen Pixels auf Instanzen von Läufen, Keimreihenkopien und Literalen; und
eine Einrichtung zum Decodieren (**504**) der Instanzen von Läufen, Keimreihenkopien und Literalen für eine Übertragung an die Druckmaschine (**206**).

30. Drucker (**106**) gemäß Anspruch 29, der zusätzlich folgendes Merkmal aufweist:
eine Einrichtung zum Durchführen des parallelen Ladens von Gleichungen innerhalb der Sequenz priorisierter Gleichungen, wobei Ladegleichungen, Register und Signale teilweise auf dem Vorgriffspuffer (**506**) basieren.

31. Drucker (**106**) gemäß Anspruch 29 oder 30, der zusätzlich folgendes Merkmal aufweist:
eine Einrichtung zum Ausgeben dekomprimierter Daten, die pro Taktzyklus einem Pixel zugeordnet sind.

32. Drucker (**106**) gemäß einem der Ansprüche 29 bis 31, bei dem die Zustandsmaschine (**600**) zusätzlich folgende Merkmale aufweist:
eine Einrichtung zum Instanzieren eines Ruhezustands (**602**), um auf einen neuen Befehl zu warten;
eine Einrichtung zum Instanzieren eines Decodiere-Befehl-Zustands (**604**), um einen aktuellen Befehl zu betrachten;
eine Einrichtung zum Instanzieren eines Schrei-

be-Keimreihenkopie-Anfangszustands (**606**), um eine Anzahl von spezifizierten Anfangskeimreihenkopiepixeln auszuschreiben;
eine Einrichtung zum Instanzieren eines Schreibe-Keimreihenkopie-Folgezustands (**608**), um Folgekeimreihenkopiepixel, die in Keimreihenkopiezählwertbytes, die auf einen Befehl folgen, spezifiziert sind, auszuschreiben;
eine Einrichtung zum Instanzieren eines Warte-Auf-Erstes-Pixel-Zustands (**610**), um darauf zu warten, das ein erstes Pixel eines Laufs oder eines Literals verfügbar ist;
eine Einrichtung zum Instanzieren eines Schreibe-Ersetzung-Anfangszustands (**612**), um Literal- oder Laufpixel auszuschreiben; und
eine Einrichtung zum Instanzieren eines Schreibe-Ersetzung-Folgezustands (**614**), um Folge-Literal- oder Laufpixel auszuschreiben; und
eine Sequenz priorisierter Gleichungen, die jedem Zustand zugeordnet sind, wobei jede Gleichung innerhalb jeder Sequenz priorisierter Gleichungen einem Zustand zugeordnet ist.

Es folgen 8 Blatt Zeichnungen

Anhängende Zeichnungen

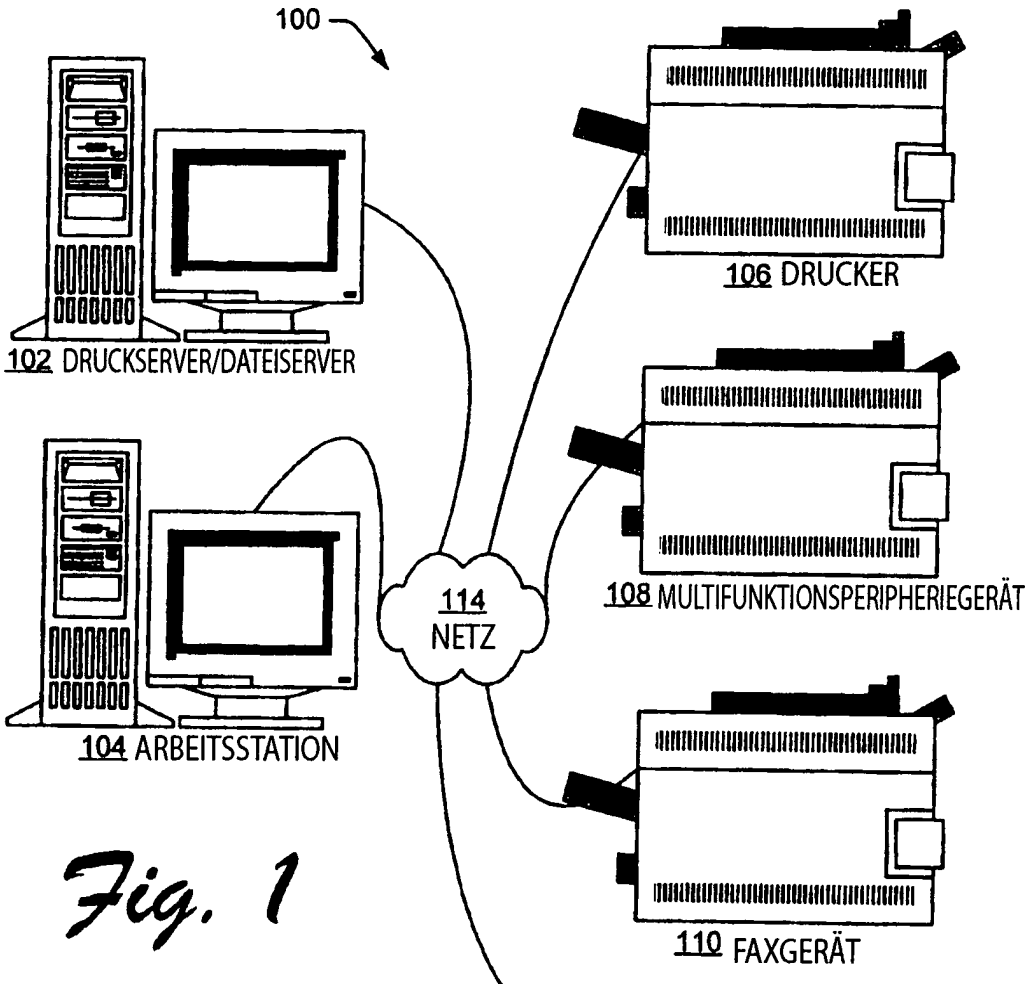


Fig. 1

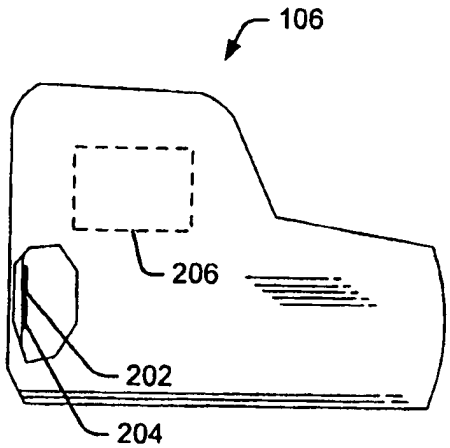


Fig. 2

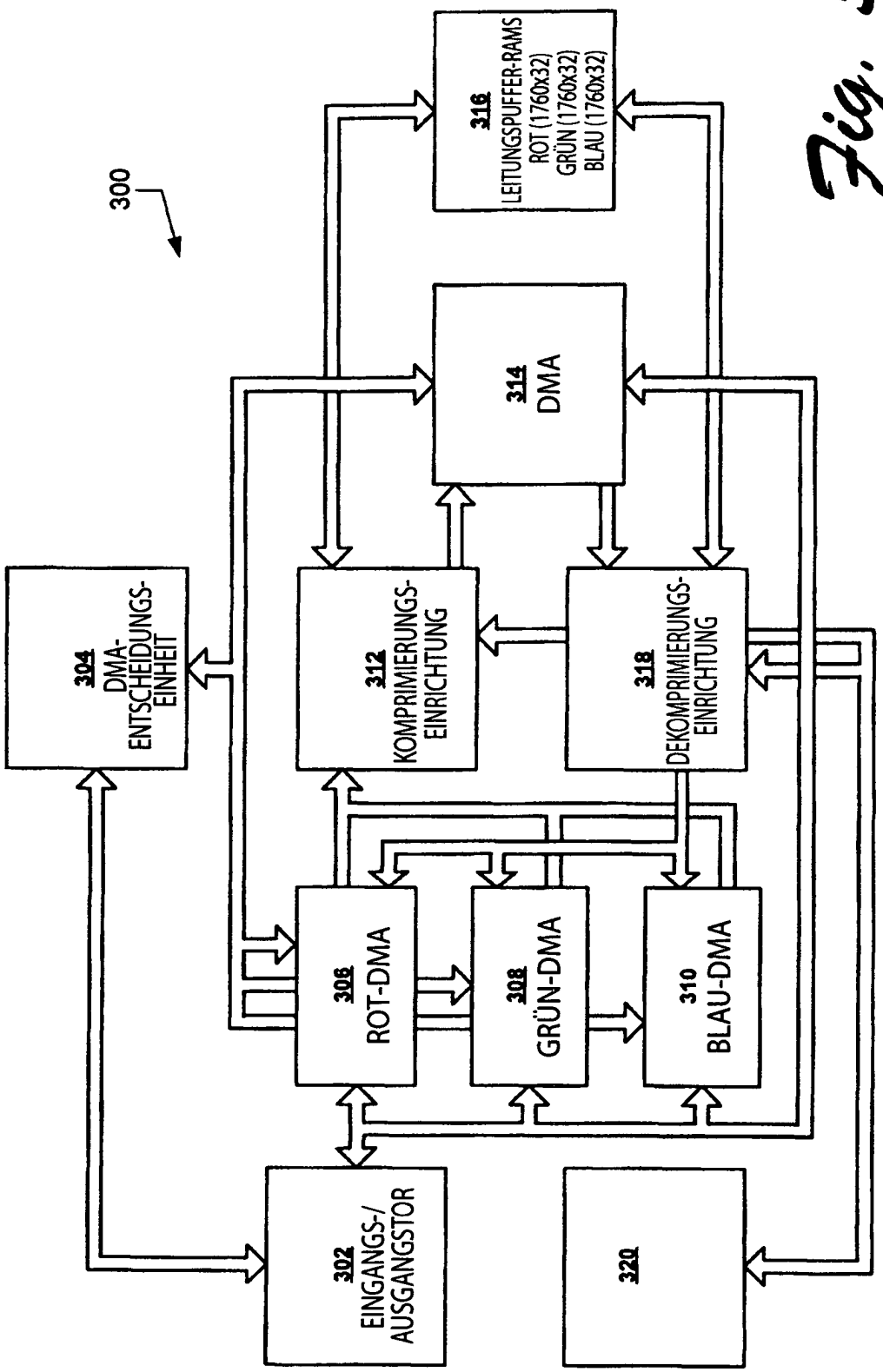

300
316
LEITUNGSPUFFER-RAMS
ROT (1760x32)
GRÜN (1760x32)
BLAU (1760x32)
314
DMA
304
DMA-
ENTSCHEIDUNGS-
EINHEIT
312
KOMPRIMIERUNGS-
EINRICHTUNG
318
DEKOMPRIMIERUNGS-
EINRICHTUNG
306
ROT-DMA
308
GRÜN-DMA
310
BLAU-DMA
302
EINGANGS-/
AUSGANGSTOR
320

Fig. 3

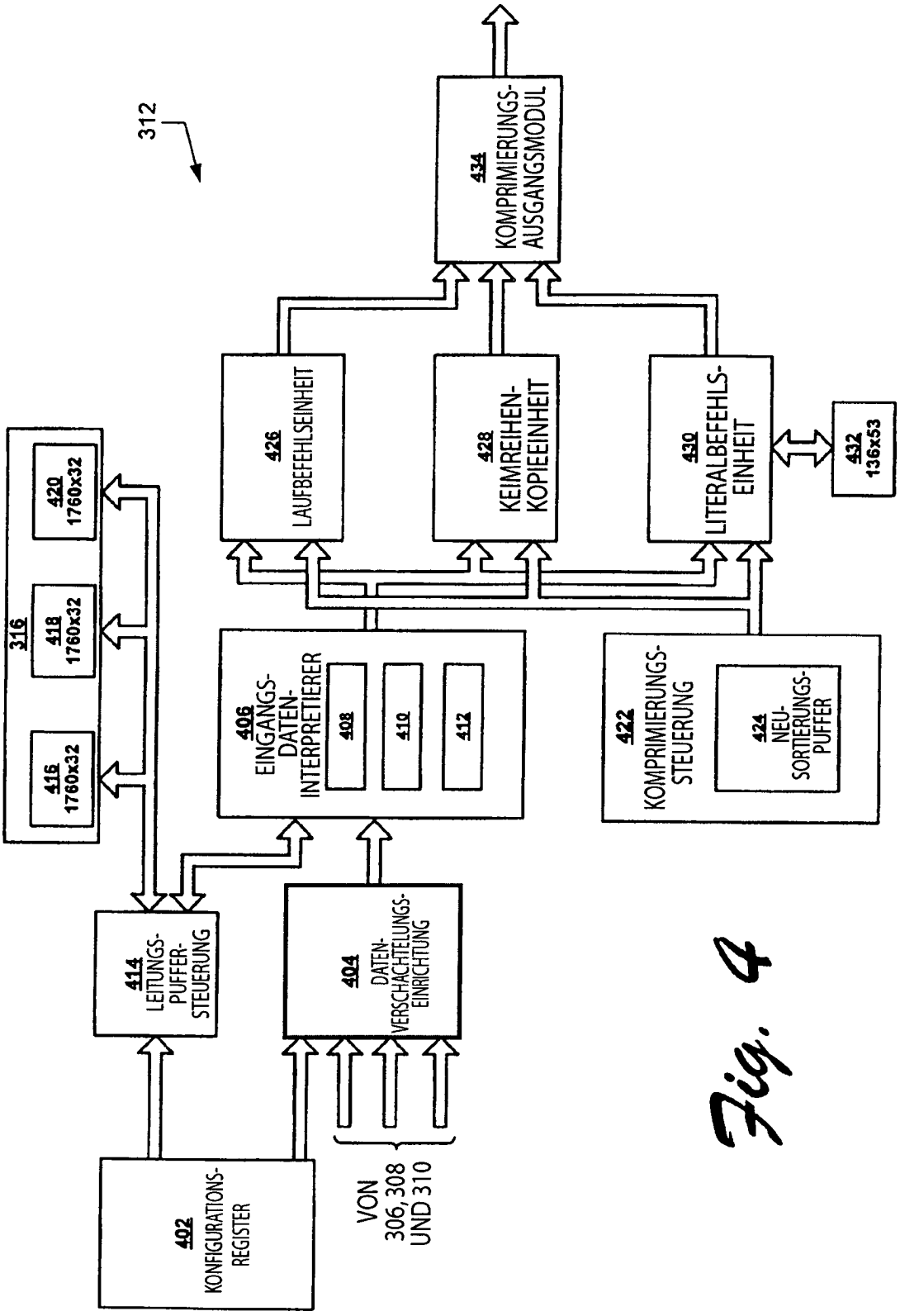

312
434 KOMPRIMIERUNGS-AUSGANGSMODUL
426 LAUFBEFEHLSEINHEIT
428 KEIMREIHEN-KOPIEEINHEIT
430 LITERALBEFEHLS-EINHEIT
432 136x53
316
420 1760x32
418 1760x32
416 1760x32
406 EINGANGS-DATEN-INTERPRETIERER
408
410
412
422 KOMPRIMIERUNGS-STEUERUNG
424 NEU-SORTIERUNGS-PUFFER
414 LEITUNGS-PUFFER-STEUERUNG
404 DATEN-VERSCHACHTELUNGS-EINRICHTUNG
402 KONFIGURATIONS-REGISTER
VON 306, 308 UND 310

Fig. 4

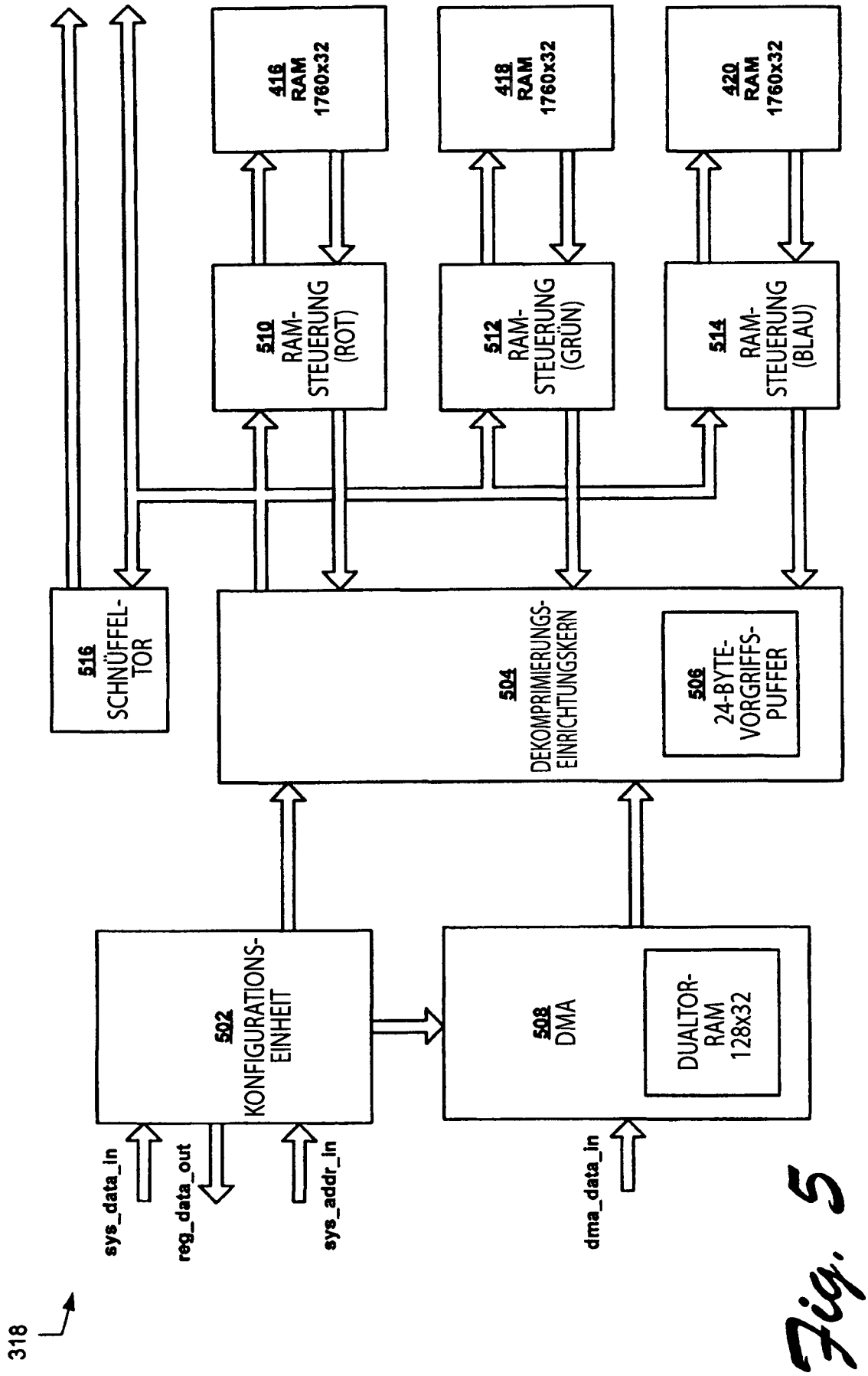

318
sys_data_in
reg_data_out
sys_addr_in
dma_data_in
502 KONFIGURATIONS-EINHEIT
508 DMA
DUALTOR-RAM 128x32
504 DEKOMPRIMIERUNGS-EINRICHTUNGSKERN
506 24-BYTE-VORGRIFFS-PUFFER
516 SCHNÜFFEL-TOR
510 RAM-STEUERUNG (ROT)
512 RAM-STEUERUNG (GRÜN)
514 RAM-STEUERUNG (BLAU)
416 RAM 1760x32
418 RAM 1760x32
420 RAM 1760x32

Fig. 5

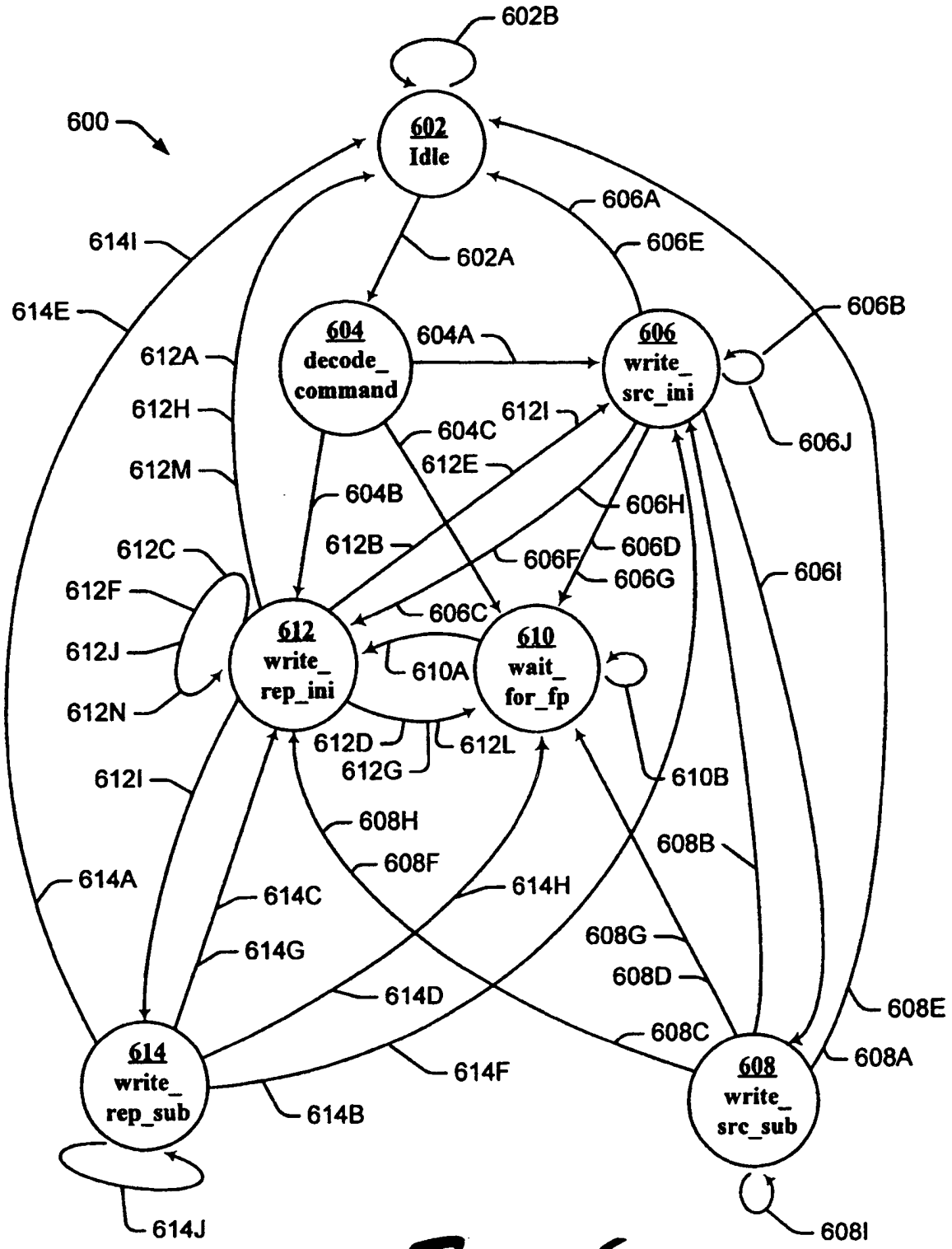

600
602
Idle
602B
602A
604
decode_
command
604A
606
write_
src_ini
606A
606B
606E
606J
610
wait_
for_fp
610A
610B
612
write_
rep_ini
614
write_
rep_sub
608
write_
src_sub
608A
608B
608C
608D
608E
608G
608H
608F
608I
612A
612B
612C
612D
612E
612F
612G
612H
612I
612J
612L
612M
612N
614A
614B
614C
614D
614E
614F
614G
614H
614I
614J
604B
604C
606C
606D
606F
606G
606H
606I

Fig. 6

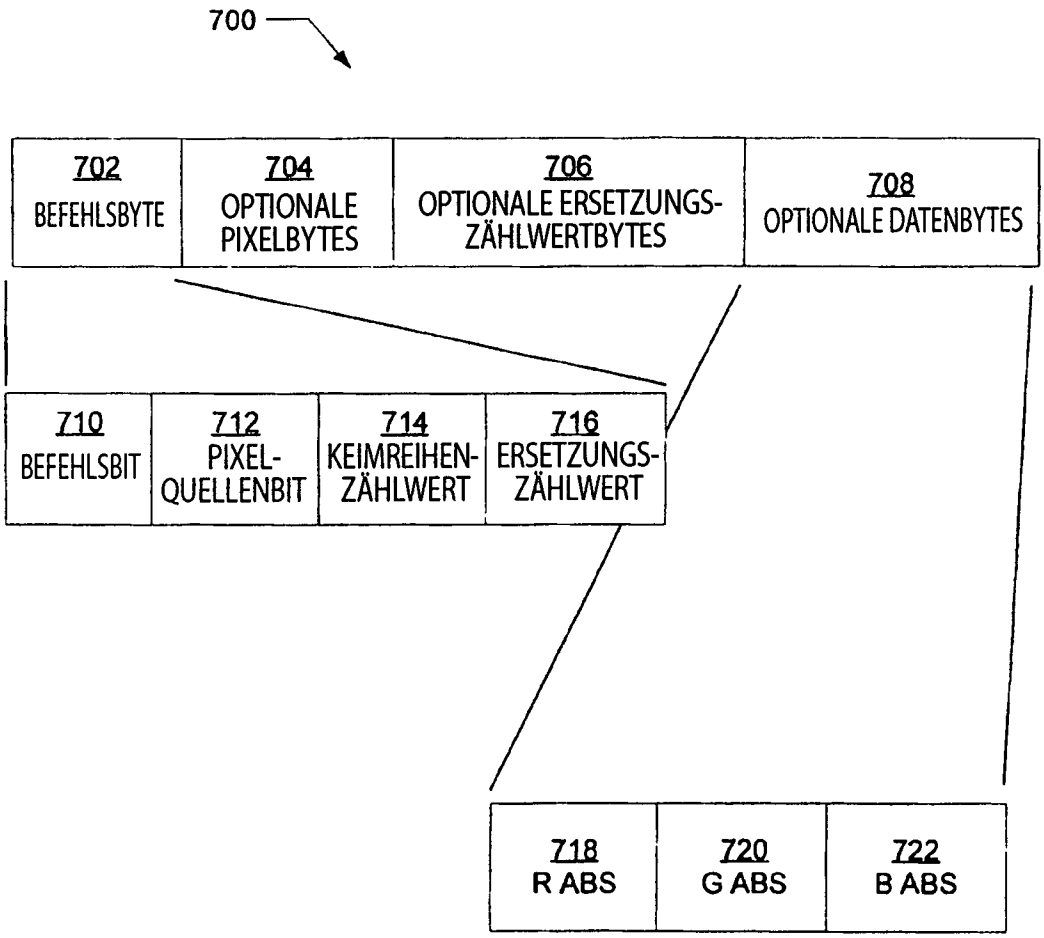

700
702
BEFEHLSBYTE
704
OPTIONALE PIXELBYTES
706
OPTIONALE ERSETZUNGS-ZÄHLWERTBYTES
708
OPTIONALE DATENBYTES
710
BEFEHLSBIT
712
PIXEL-QUELLENBIT
714
KEIMREIHEN-ZÄHLWERT
716
ERSETZUNGS-ZÄHLWERT
718
R ABS
720
G ABS
722
B ABS

Fig. 7

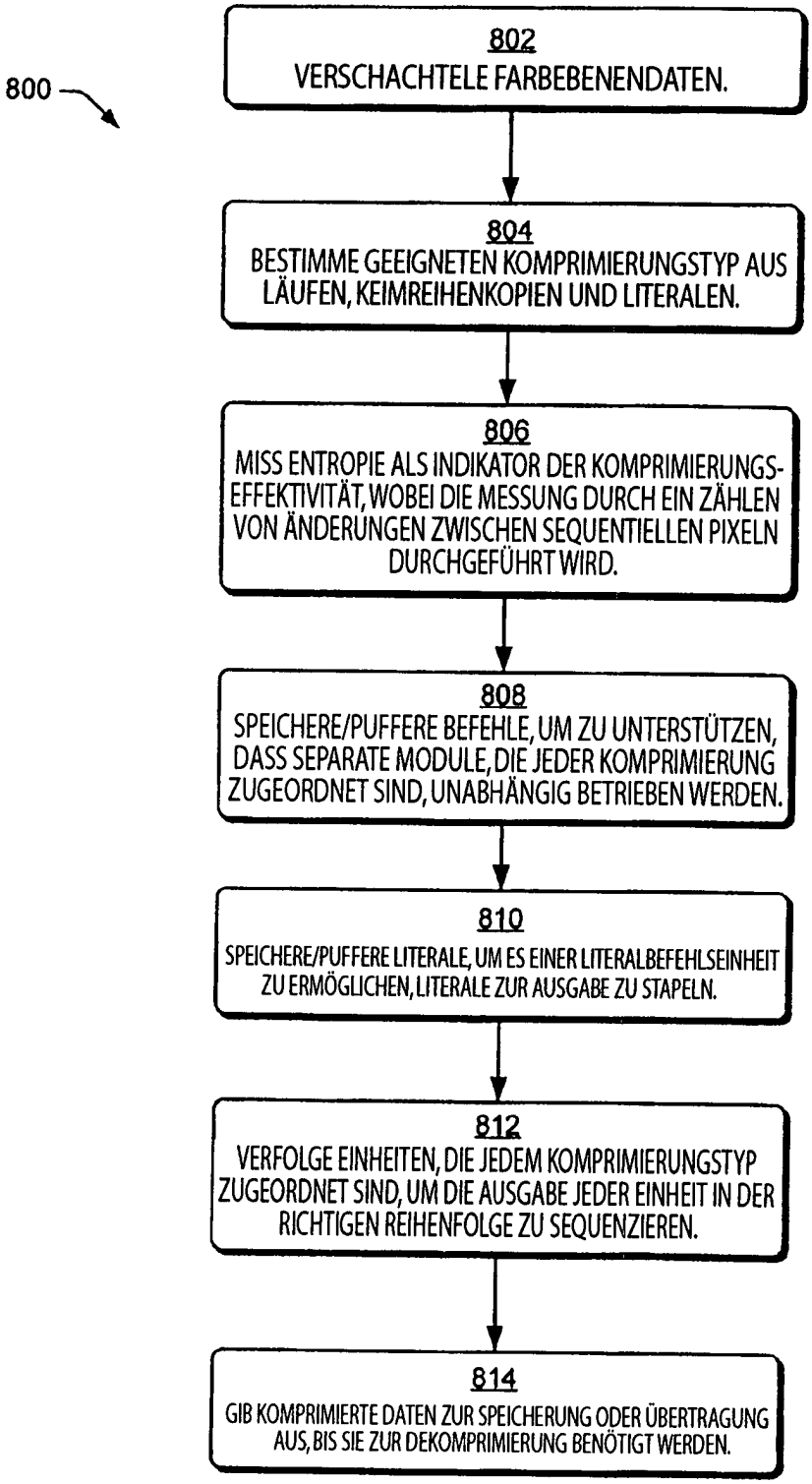

800
802
VERSCHACHTELE FARBEBENENDATEN.
804
BESTIMME GEEIGNETEN KOMPRIMIERUNGSTYP AUS LÄUFEN, KEIMREIHENKOPIEN UND LITERALEN.
806
MISS ENTROPIE ALS INDIKATOR DER KOMPRIMIERUNGS-EFFEKTIVITÄT, WOBEI DIE MESSUNG DURCH EIN ZÄHLEN VON ÄNDERUNGEN ZWISCHEN SEQUENTIELLEN PIXELN DURCHGEFÜHRT WIRD.
808
SPEICHERE/PUFFERE BEFEHLE, UM ZU UNTERSTÜTZEN, DASS SEPARATE MODULE, DIE JEDER KOMPRIMIERUNG ZUGEORDNET SIND, UNABHÄNGIG BETRIEBEN WERDEN.
810
SPEICHERE/PUFFERE LITERALE, UM ES EINER LITERALBEFEHLSEINHEIT ZU ERMÖGLICHEN, LITERALE ZUR AUSGABE ZU STAPELN.
812
VERFOLGE EINHEITEN, DIE JEDEM KOMPRIMIERUNGSTYP ZUGEORDNET SIND, UM DIE AUSGABE JEDER EINHEIT IN DER RICHTIGEN REIHENFOLGE ZU SEQUENZIEREN.
814
GIB KOMPRIMIERTE DATEN ZUR SPEICHERUNG ODER ÜBERTRAGUNG AUS, BIS SIE ZUR DEKOMPRIMIERUNG BENÖTIGT WERDEN.

Fig. 8

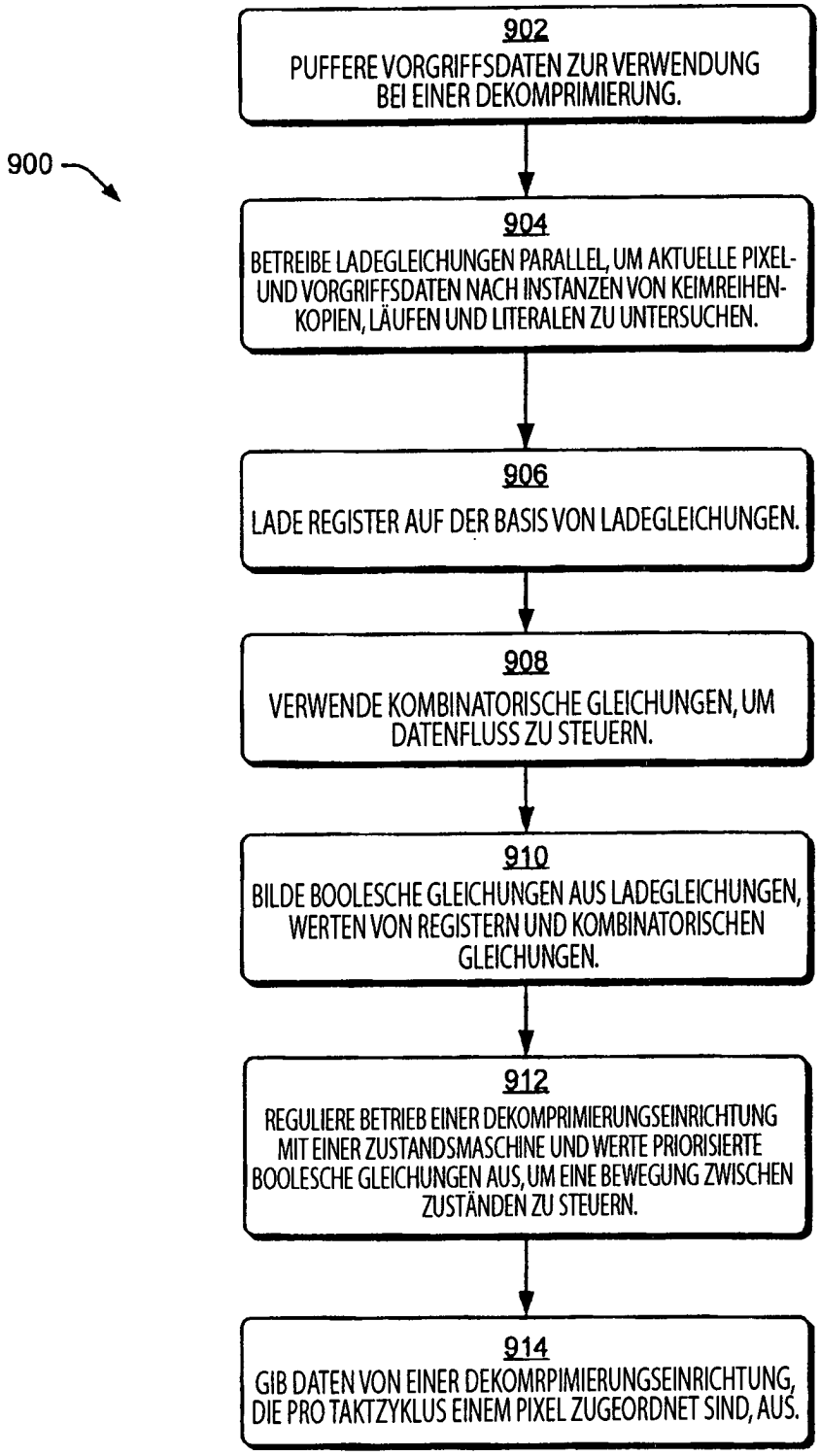

900
902
PUFFERE VORGRIFFSDATEN ZUR VERWENDUNG BEI EINER DEKOMPRIMIERUNG.
904
BETREIBE LADEGLEICHUNGEN PARALLEL, UM AKTUELLE PIXEL- UND VORGRIFFSDATEN NACH INSTANZEN VON KEIMREIHEN-KOPIEN, LÄUFEN UND LITERALEN ZU UNTERSUCHEN.
906
LADE REGISTER AUF DER BASIS VON LADEGLEICHUNGEN.
908
VERWENDE KOMBINATORISCHE GLEICHUNGEN, UM DATENFLUSS ZU STEUERN.
910
BILDE BOOLESCHE GLEICHUNGEN AUS LADEGLEICHUNGEN, WERTEN VON REGISTERN UND KOMBINATORISCHEN GLEICHUNGEN.
912
REGULIERE BETRIEB EINER DEKOMPRIMIERUNGSEINRICHTUNG MIT EINER ZUSTANDSMASCHINE UND WERTE PRIORISIERTE BOOLESCHE GLEICHUNGEN AUS, UM EINE BEWEGUNG ZWISCHEN ZUSTÄNDEN ZU STEUERN.
914
GIB DATEN VON EINER DEKOMRPIMIERUNGSEINRICHTUNG, DIE PRO TAKTZYKLUS EINEM PIXEL ZUGEORDNET SIND, AUS.

Fig. 9