(19)日本国特許庁(JP)　(12)公開特許公報(A)　(11)特許出願公開番号
特開2010-68511
(P2010-68511A)
(43)公開日　平成22年3月25日(2010.3.25)

(51)Int.Cl.		F I			テーマコード（参考）
H03M 7/40	(2006.01)	H03M	7/40		5B060
G06F 12/04	(2006.01)	G06F	12/04	530	5J064

審査請求 未請求　請求項の数 10　ＯＬ　外国語出願　（全 34 頁）

(21)出願番号　特願2009-168779 (P2009-168779)
(22)出願日　平成21年7月17日 (2009.7.17)
(31)優先権主張番号　08252480.2
(32)優先日　平成20年7月21日 (2008.7.21)
(33)優先権主張国　欧州特許庁 (EP)

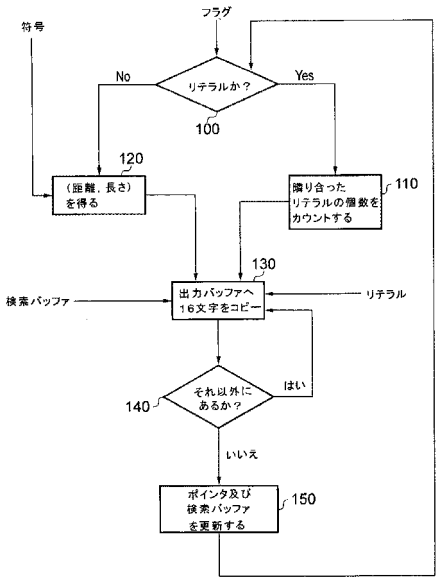
(71)出願人　502070679
ソニー　コンピュータ　エンタテインメント　ヨーロッパ　リミテッド
イギリス国　ダブリュー１エフ　７エルピー　ロンドン　グレート　マールボロ　ストリート　１０
(74)代理人　110000198
特許業務法人湘洋内外特許事務所
(72)発明者　ヒューズ　コリン　ジョナサン
イギリス国　ダブリュー１エフ　７エルピー　ロンドン　グレート　マールボロ　ストリート　１０　ソニー　コンピュータ　エンタテインメント　ヨーロッパ　リミテッド内

Ｆターム(参考)　5B060 DA08
5J064 BA09　BA11　BC01　BD02　BD03

(54)【発明の名称】データ圧縮及びデータ伸張

(57)【要約】　（修正有）
【課題】文字圧縮の改良されたデータ圧縮及び／又は伸張技法を提供する。
【解決手段】前に復号されたデータアイテムの群の参照子の順序付けられたストリーム、復号されるデータアイテムの直接的表現の順序付けられたストリーム、及び各連続的な伸張オペレーションが参照子に作用すべきであるのか、又は直接的表現に作用すべきであるのかを示すフラグの順序付けられたストリームを備える圧縮データに作用するように構成されるデータ伸張装置が開示される。該装置は、出力メモリエリア、伸張オペレーションが直接的表現に作用すべきであることを示す個数ｎの連続したフラグを検出する検出器、及び前に復号されたデータの次の参照される群又は直接的表現の順序付けられたストリームからのｎ個の連続した直接的表現の群のいずれかを出力メモリエリアへコピーするためのデータコピー器を備える。
【選択図】図４

図４



【特許請求の範囲】
【請求項１】
前に復号されたデータアイテムの群の参照子の順序付けられたストリーム、
復号されるデータアイテムの直接的表現の順序付けられたストリーム、及び
各連続的な伸張オペレーションが参照子に作用すべきであるのか、又は直接的表現に作用すべきであるのかを示すフラグの順序付けられたストリーム、
を備える圧縮データに作用するように構成されるデータ伸張装置であって、
該装置は、
出力メモリエリア、
伸張オペレーションが直接的表現に作用すべきであることを示す個数ｎの連続したフラグを検出する検出器、及び
前に復号されたデータの次の参照される群又は前記直接的表現の順序付けられたストリームからのｎ個の連続した直接的表現の群のいずれかを前記出力メモリエリアへコピーするためのデータコピー器、
を備える、装置。
【請求項２】
前記データコピー器は、
前記出力メモリエリアへ既に書き込まれた有効に伸張されたデータの範囲を示す、前記出力メモリエリアに関するポインタを維持し、
前記参照される群のサイズ又は前記個数ｎに関わらず、ｍ個の連続したデータアイテムの群をコピーし、且つ
前記参照される群のサイズ又は前記個数ｎに応じて前記ポインタを設定する、
ように構成される、請求項１に記載の装置。
【請求項３】
前記伸張されるデータは、２つ以上の隣接したデータブロックのシリーズを含み、前記装置は、
前記出力メモリエリアとは別個のデータストアであって、前記シリーズにおける少なくとも最初のブロック以外のブロックに対応する最初に伸張されたデータアイテムの群を記憶するためのデータストアを備え、該データストアは、前記シリーズにおける先行ブロックの伸張が完了した後に、前記記憶されたデータアイテムの群をその元の位置へ再書き込みするように構成される、請求項１に記載の装置。
【請求項４】
前記ブロックは、各ブロックが実質的に同じ量の伸張処理を必要とするようにサイズを構成される、請求項３に記載の装置。
【請求項５】
前記ブロックは、並列に伸張される、請求項４に記載の装置。
【請求項６】
前に復号されたデータアイテムの群の参照子の順序付けられたストリーム、
復号されるデータアイテムの直接的表現の順序付けられたストリーム、及び
各連続的な伸張オペレーションが参照子に作用すべきであるのか、又は直接的表現に作用すべきであるのかを示すフラグの順序付けられたストリーム、
を備える圧縮データに作用するように構成されるデータ伸張方法であって、
該方法は、
伸張オペレーションが直接的表現に作用すべきであることを示す個数ｎの連続したフラグを検出するステップ、及び
前に復号されたデータの次の参照される群又は前記直接的表現の順序付けられたストリームからのｎ個の連続した直接的表現の群のいずれかを出力メモリエリアへコピーするステップ、
を含む、方法。
【請求項７】

入力データアイテムが、前に符号化されたデータアイテムの群の参照子として又は前記入力データアイテムの直接的表現としてのいずれかで符号化されるデータ圧縮方法であって、該方法は、
参照子の順序付けられたストリーム、
直接的表現の順序付けられたストリーム、及び
各連続的な伸張オペレーションが参照子に作用すべきであるのか、又は直接的表現に作用すべきであるのかを示すフラグの順序付けられたストリーム、
の、出力データの別個のストリームとして生成するステップを含む、データ圧縮方法。
【請求項８】
コンピュータによって実行されると、請求項７に記載の方法を該コンピュータに実行させるプログラムコードを含むコンピュータソフトウェアが記憶されているマシン可読媒体。
【請求項９】
入力データアイテムが、前に符号化されたデータアイテムの群の参照子として又は前記入力データアイテムの直接的表現としてのいずれかで符号化されるデータ圧縮装置であって、該装置は、
参照子の順序付けられたストリーム、
直接的表現の順序付けられたストリーム、及び
各連続的な伸張オペレーションが参照子に作用すべきであるのか、又は直接的表現に作用すべきであるのかを示すフラグの順序付けられたストリーム、
の、出力データの別個のストリームとして生成するように構成される、データ圧縮装置。
【請求項１０】
入力データアイテムが、前に符号化されたデータアイテムの群の参照子として又は前記入力データアイテムの直接的表現としてのいずれかで符号化される、データ圧縮プロセスによって生成されるデータ信号であって、該データ信号は、
参照子の順序付けられたストリーム、
直接的表現の順序付けられたストリーム、及び
各連続的な伸張オペレーションが参照子に作用すべきであるのか、又は直接的表現に作用すべきであるのかを示すフラグの順序付けられたストリーム、
の、出力データの別個のストリームとして含む、データ信号。
【発明の詳細な説明】
【技術分野】
【０００１】
本発明は、データ圧縮及びデータ伸張に関する。
【背景技術】
【０００２】
レンペル（Lempel）及びジフ（Ziv）は、いわゆるＬＺ７７アルゴリズム及びＬＺ７８アルゴリズム等の多数の可逆データ圧縮アルゴリズムを生み出した。これらのいわゆるＬＺ７７アルゴリズム及びＬＺ７８アルゴリズムは、ＬＺＷ、ＬＺＳＳ他を含む多くの変形の基礎を成している。これらのＬＺ７７アルゴリズム及びＬＺ７８アルゴリズムは、共に、辞書コーダ（dictionary coder）である。辞書コーダは、符号化器及び復号器の双方をすでに通過した一致データの参照子でデータの一部を置換することによって圧縮を達成する。データの一致するセクションは、距離－長さ対（distance-length pair）と呼ばれる数字の対によって符号化され、効果的には、カウンタ（いくつの文字をコピーするのか）及びポインタ（すでに符号化されたデータ又はすでに復号されたデータにおける最初のこのような文字をどこで見つけるのか）の対によって符号化される。距離－長さ対は、「次の長さ文字のそれぞれが、未圧縮ストリームにおいてその後方にある文字、正確には距離文字に等しい」という記述と等価であると見ることができる。
【０００３】
符号化フェーズ中、（符号化される文字の文字列の）可能な最良の一致するものが探し

出される。すなわち、システムは、利用可能な符号化データ内において最大の長さを与える一致するものを得る。このプロセスは、添付図面の図１ａ～図１ｃに関して模式的に示される。

【０００４】

既に符号化されているデータは、「検索バッファ（search buffer）」１０と呼ばれるメモリのエリアに記憶される。この検索バッファは、特定のアプリケーションにおいて既に符号化されたデータのすべてを保持するのに十分大きなものとすることもできるし、一定の量の最も近時に符号化されたデータのみを保持することもできる。符号化されるデータは、「先読みバッファ（look-ahead buffer）」２０と呼ばれるメモリのエリアに記憶される。先読みバッファ２０の最初の文字（この場合、文字「ｇ」）が、符号化される次の文字である。もちろん、本出願では符号化されるデータを示すのに英数文字が使用され、これらのデータは「文字」として参照されるが、この表記は、解説を明確にするだけのためのものであることが認識されよう。当然、データが英数文字を表すものであろうと、ピクセルを表すものであろうと、オーディオサンプルを表すものであろうと、その他何を表すものであろうと、技術的に重要ではない。同様に、このような各「文字」のビットのサイズも、技術的に重要ではない。

【０００５】

符号化器は、検索バッファ１０において「ｇ」のインスタンスを検索する。本例では、２つが見つけられる（図１ａ）。

【０００６】

次に、符号化器は、検索バッファにおいて「ｇ」の各インスタンスに続く位置を調べて、先読みバッファにおける次の文字（文字「ｆ」）が「ｇ」のいずれかのインスタンスの後に続いているか否かを検出する。実際には、次の文字は、双方のインスタンスにおいて後に続いており（図１ｂ）、そこで、検索は、先読みバッファにおける次の後続の文字（「ｓ」）に進む。この文字は、「ｇｆ」の１つのインスタンスの後にしか続いておらず（図１ｃ）、先読みバッファにおける次の後続の文字（「ｈ」）は、検索バッファにおいてそれら３つの文字の後に続いていない。そこで、先読みバッファの最初の３つの文字に関して生成される距離－長さ対は、（１３，３）である。ここで、この文字列は、符号化シーケンス（検索バッファ）において１３文字戻って開始し、３文字の長さを有する。

【０００７】

２つ以上の文字の一致する文字列が検索バッファにおいて見つからない場合には、その文字は、「リテラル（literal）」として符号化される。すなわち、その文字は、出力データストリーム内に単にコピーされるだけである。そこで、圧縮の利益は、距離－長さ対の使用の結果生じ、リテラルは、元の文字と同程度の大きさであるだけでなく、その文字がリテラルであることを示す或る種のフラグも必要となるので、リテラルを引用することは、データ圧縮の目的に反した振る舞いをする。したがって、これらの技法は、有益な圧縮比を達成するために、文字列が繰り返す十分な見込みがあるデータには良く適している。

【０００８】

したがって、一般に、このタイプのプロセスによって生成される圧縮データは、リテラルが点在した距離－長さ対から形成されることになる。

【０００９】

図２ａ～図２ｄは、距離－長さ対（１３，３）の復号を模式的に示している。

【００１０】

図２ａは、距離－長さ対（１３，３）がまさに復号されようとしている時の利用可能な復号データを提供する検索バッファを示している。図２ｂ、図２ｃ、及び図２ｄでは、検索バッファにおいて１３文字戻った位置から開始する３つの文字の文字列が、出力用にコピーされる。

【００１１】

このタイプのデータ圧縮及びデータ伸張は、データの符号化及び復号の成功が、符号化

又は復号された前のすべてのデータに依拠するという点で、本質的に線形プロセスである。

【発明の概要】

【発明が解決しようとする課題】

【００１２】

本発明の目的は、改良されたデータ圧縮及び／又は伸張技法を提供しようとすることである。

【課題を解決するための手段】

【００１３】

本発明は、前に復号されたデータアイテムの群の参照子の順序付けられたストリーム、復号されるデータアイテムの直接的表現の順序付けられたストリーム、及び各連続的な伸張オペレーションが参照子に作用すべきであるのか、又は直接的表現に作用すべきであるのかを示すフラグの順序付けられたストリーム、を備える圧縮データに作用するように構成されるデータ伸張装置であって、該装置は、出力メモリエリア、伸張オペレーションが直接的表現に作用すべきであることを示す個数ｎの連続したフラグを検出する検出器、及び前に復号されたデータの次の参照される群又は直接的表現の順序付けられたストリームからのｎ個の連続した直接的表現の群のいずれかを出力メモリエリアへコピーするためのデータコピー器を備える、装置を提供する。

【００１４】

本発明のさらなる各態様及び各特徴は、添付の特許請求の範囲に規定されている。

【００１５】

本発明の上記の目的、特徴、及び利点、並びに他の目的、特徴、及び利点は、添付図面に関して読まれる例示の実施形態の以下の詳細な説明から明らかであろう。

【図面の簡単な説明】

【００１６】

【図１ａ】これまでに提案されたデータ圧縮技法を模式的に示す。

【図１ｂ】これまでに提案されたデータ圧縮技法を模式的に示す。

【図１ｃ】これまでに提案されたデータ圧縮技法を模式的に示す。

【図２ａ】対応するこれまでに提案されたデータ伸張技法を模式的に示す。

【図２ｂ】対応するこれまでに提案されたデータ伸張技法を模式的に示す。

【図２ｃ】対応するこれまでに提案されたデータ伸張技法を模式的に示す。

【図２ｄ】対応するこれまでに提案されたデータ伸張技法を模式的に示す。

【図３ａ】データの３つのストリームを模式的に示す。

【図３ｂ】データの３つのストリームを模式的に示す。

【図４】図３ｂのデータストリームを使用する伸張技法を示す模式的なフローチャートである。

【図５ａ】図４のフローチャートによるデータ伸張の模式的な加工例（worked example）を示す。

【図５ｂ】図４のフローチャートによるデータ伸張の模式的な加工例を示す。

【図５ｃ】図４のフローチャートによるデータ伸張の模式的な加工例を示す。

【図５ｄ】図４のフローチャートによるデータ伸張の模式的な加工例を示す。

【図６ａ】並列伸張技法を模式的に示す。

【図６ｂ】並列伸張技法を模式的に示す。

【図６ｃ】並列伸張技法を模式的に示す。

【図７】並列圧縮技法を示す模式的なフローチャートである。

【図８】並列伸張技法を模式的に示す。

【図９ａ】各ブロックの最初の１つ又は複数の復号文字に関する並列伸張技法の特徴を模式的に示す。

【図９ｂ】各ブロックの最初の１つ又は複数の復号文字に関する並列伸張技法の特徴を模式的に示す。

【図９ｃ】各ブロックの最初の１つ又は複数の復号文字に関する並列伸張技法の特徴を模式的に示す。
【図１０】並列伸張技法を示す模式的なフローチャートである。
【図１１】データ圧縮及び／又は伸張装置の模式図である。
【発明を実施するための形態】
【００１７】
次に図３ａ及び図３ｂを参照すると、図１及び図２に関して説明された圧縮プロセス等の圧縮プロセスによって出力されたデータ信号が、距離－長さ対（図３ｂで「符号」と呼ばれる）の順序付けられたストリーム５０、リテラルの順序付けられたストリーム６０、及びフラグの順序付けられたストリーム７０の３つのデータのストリームとして（圧縮装置により又は中間プロセスとして）配列されている。フラグは、復号される次のデータアイテムが距離－長さ対であるのか、又はリテラルであるのかを示す。
【００１８】
図３ａは、図１の技法によって生成されたような距離－長さ対及びリテラルが点在した単一のストリームからストリーム５０及び６０を再分割することができた時のストリーム５０及び６０を示している。フラグは、「ｌ」（リテラル）及び「ｃｐ」（符号対、すなわち距離－長さ対）として模式的に示されている。
【００１９】
図３ｂにおいて、ストリームは、ギャップを除去するように連接されている。復号時において、フラグストリームの次のエントリーは、符号化データの次のアイテムを符号ストリームから取り出すのか、又はリテラルストリームから取り出すのかを示す。符号ストリームの次のアイテムを示すポインタ（図示せず）及びリテラルストリームの次のアイテムを示すポインタ（図示せず）が維持される。アイテムが復号用にストリームから取り出されるたびに、関連のあるポインタは更新される（１アイテム分だけ前方に移動される）。
【００２０】
図４は、図３ｂのデータストリームを使用する伸張技法を示す模式的なフローチャートである。
【００２１】
ステップ１００において、フラグストリームにおける次のフラグを検査して、そのフラグがリテラルを示すのか、又は符号対を示すのかを検出する。フラグがリテラルを示す場合には、ステップ１１０において、現在の「リテラル」フラグの後に続く「リテラル」フラグの個数をカウントすることにより、隣り合った（連続した）リテラルの個数が確定される。このカウントは、データ処理システム内の通常の解析的技法を使用して行うことができるが、Ｓｏｎｙ　Ｃｏｍｐｕｔｅｒ　Ｅｎｔｅｒｔａｉｎｍｅｎｔ（ソニーコンピュータエンターテインメント）（登録商標）、Ｔｏｓｈｉｂａ（東芝）（登録商標）、及びＩＢＭ（登録商標）によって開発された「セル（Cell）」マイクロプロセッサを使用する本実施形態（以下の図１１参照）では、このマイクロプロセッサが、同一のビットの列の長さをカウントする単一のコマンドを実際に有する。そこで、ステップ１１０は、このようなセルマイクロプロセッサによって高速且つ容易に実行することができる。
【００２２】
現在のフラグがリテラルを示していない場合には、次の符号対をステップ１２０において検査して、その符号対によって表される距離及び長さの量をリトリーブする。
【００２３】
ステップ１３０において、コピーオペレーションを実行する。このコピーオペレーションでは、１６個の隣り合った文字のブロックが、データ出力バッファ（すなわち、復号データが記憶されるメモリの場所又はエリア）へコピーされる。
（ａ）現在のフラグがリテラルを示していた場合には、次の１６個のリテラルが、リテラルストリームから出力バッファへコピーされる。
（ｂ）現在のフラグが符号対を示していた場合には、検査バッファにおいて距離変数により示される位置の後に続く１６文字が出力バッファへコピーされる。

【００２４】
連続したリテラルフラグの個数によって又は長さ変数によって実際に示される文字の個数に関わらず、１６文字のブロックがコピーされる理由は、セルマイクロプロセッサがこの長さのブロックの効率的なコピー用に特に設計されているからである。当然、実際のリテラルの個数又は長さ変数が正確に１６に等しいことは考えにくい。そこで、さまざまな対策を適用してこれに対処しなければならない。
【００２５】
まず、リテラルの個数又は長さ変数が１６よりも大きい状況を考えると、ステップ１４０において、この状況を検出し（「それ以外にあるか？」という質問は、コピーされる文字がそれ以外に残っているか否かを検出する、すなわち、リテラルの元の個数又は元の長さ変数から、既にコピーされた文字の個数を差し引いた残りが、０よりも大きいか否かを検出する）、制御は、ステップ１３０に再び渡り、１６の別のブロックがコピーされる。これは、ステップ１３０によってコピーされたリテラルの個数又は前に復号された文字が、必要とされる個数に達するか又は必要とされる個数を超えるまで続く。
【００２６】
その時点で、制御はステップ１５０に渡る。このステップにおいて、さまざまなポインタを更新する。出力バッファにおける現在の出力位置を示すポインタを、ステップ１３０によって出力バッファ内に書き込まれた最後の有効な文字を示すように更新する（すなわち、このポインタは、出力バッファへ既に書き込まれた有効に伸張されたデータの範囲を示す）。ステップ１３０が、１６（又は１６の倍数）個の文字を、必要な個数に達するか又は必要な個数を超えるようにコピーしたことを思い出すと、最後に有効にコピーされた文字は、ステップ１３０によってコピーされた最後の１６文字内のいずれかにあることになる。
【００２７】
たとえば、ステップ１１０によって検出された連続したリテラルフラグの個数が７であったものと仮定する。ステップ１３０において、現在の出力ポインタの位置から開始して、１６個のリテラルが出力バッファ内にコピーされている。この個数１６（この例では）は、有効なリテラルの個数（７）に関わらず、一定である。しかし、これらの１６個のコピーされたリテラルの最初の７つのみが有効な復号データである。したがって、出力ポインタは、出力データバッファにおいて７文字前方に移動するように更新される。これは、ステップ１３０による次のコピーオペレーションが、前のコピーオペレーションの終端ではなく、前にコピーされたブロックの先頭の後の７文字で開始し、それによって、前にコピーされたが無効な最後の９文字は上書きされることになる。
【００２８】
コピーオペレーションがリテラルに関係していたのか、又は符号対に関係していたのかに応じて、別の２つのポインタの一方をステップ１５０において更新する必要がある。最後のコピーオペレーションがリテラルに関係していた場合には、使用される（リテラルストリームにおける）次のリテラルへのポインタを（この例では、７つの位置だけ）更新する。最後のコピーオペレーションが符号対に関係していた場合には、符号対ストリームにおける次の利用可能な符号対へのポインタを更新する。
【００２９】
最後に、検索バッファのコンテンツを、最も近時の復号データを反映するように更新する。検索バッファの更新は、そのオペレーションサイクルで実行された復号を反映するように、出力バッファにおける新しく追加され且つ有効な文字を検索バッファへコピーすることを伴う。
【００３０】
次に、制御は、次のオペレーションのサイクルのために、ステップ１００に戻る。
【００３１】
特にセルマイクロプロセッサと共にデータの別個のストリームを使用することによって（ただし、セルマイクロプロセッサと共に使用することに限るものではない）、効率的な

復号プロセスが与えられる。ステップ１１０又は１２０は、ステップ１３０と同様に、１つの命令のみによって実行される。そこで、ステップ１００からステップ１５０までのサイクル全体は、比較的少数の命令で完了することができる。

【００３２】

以下に示す一例のセルプロセッサの実施態様は、５つの命令とはならないが、特定のハードウェア実施態様は、わずか５つの命令とすることできる。

【００３３】

```
[
// 必要とされるアラインされていない１６バイトストリームを含む３２バイト（１６バイトにアラインされる）をフェッチする

lq b0to15,base,distance
lq b16to31,base16,distance

//１６バイトのアラインされたストリームを、必要に応じて繰り返し生成する

shufb aligned,b0to15,b16to31,AlignRepeat

// アラインされた１６書き込みブロックは、０～１４個の前に有効なバイトを含むことができる

lq write,dest

// ストリームをアラインされた終端まで挿入する

shufb write,write,aligned,insert0
sq write,dest

// 次の１６バイトのアラインされたブロック内へのオーバーフローを生成する
shufb write,aligned,aligned,overflow
sq write,dest+16

]
```

【００３４】

図５ａ～図５ｄは、図４のフローチャートによるデータ伸張の模式的な加工例を示している。図３ｂに示すデータのストリームは、この加工例への入力として使用される。検索バッファは、ブロック１６０として模式的に示され、ステップ１３０によってコピーされた１６ビット群は、ブロック１７０として模式的に示されている。

【００３５】

図３ｂの最初の３つのフラグはリテラルフラグである。したがって、ステップ１３０は、リテラル「ｓ」から開始して１６個のリテラルを出力用にコピーする（図５ａ）。これらのうちの最初の３つのみが有効であり、そこで、ステップ１５０の結果、「ｓ」、「ｄ」、「ｇ」が検索バッファへ書き込まれ、「次のリテラル」ポインタが、図３ｂのリテラルストリームにおけるリテラル「ｊ」を指すように更新される。

【００３６】

次のフラグは符号対フラグである。図５ｂでは、最初の符号対（７，４）が符号対ストリームから読み出され、１６文字の適切なコピーが、検索バッファから作成されて出力される。これらの文字は、検索バッファからの近時に書き込まれた文字ｓ、ｄ、ｇを含むことに留意されたい。これらの文字のうちの最初の４つは有効であり、検索バッファに再びコピーされる。検索バッファに十分な文字が存在しない場合には、一定の１６文字コピー

は、これらの符号の複製されたバージョンで構成される。この必要性は、距離変数が１６未満である場合に検出され、単純な表として実施される。

すなわち、

距離７：１６文字は｛０，１，２，３，４，５，６，０，１，２，３，４，５，６，０，１｝である。

距離９：１６文字は｛０，１，２，３，４，５，６，７，８，９，０，１，２，３，４，５｝である。

【００３７】

次のフラグも符号対フラグである。次の符号対（１２，３）が読み出され、プロセスが、上述したように図５ｃで実行される。

【００３８】

最後に、次の２つのフラグはリテラルフラグである。「ｊ」から開始する１６個のリテラルのブロックが、図５ｄでコピーされる。これらのリテラルのうちの最初の２つの文字が有効なデータである。

【００３９】

いくつかの実施形態では、検索バッファは、実際には、別個のストアの形態である必要はないことに留意されたい。検索バッファに保持される情報は、出力バッファにも保持され、そこで、検索バッファの機能は、出力バッファに保持された有効に復号されたデータへの更新可能なポインタによって単純に達成することができる。検索バッファを更新するステップ（上述したステップ１５０の一部）は、単にポインタを更新することを伴うだけである。

【００４０】

図６ａ～図６ｃは、並列伸張技法を模式的に示している。これは、セルマイクロプロセッサとの関連で特に有用なさらに別の技法を表している（ただし、セルマイクロプロセッサとの関連に限られるものではない）。

【００４１】

上述したように、符号対及びリテラルを使用するこれまでに提案された技法は、本質的に線形技法である。線形処理は、セルマイクロプロセッサ他等の並列プロセッサによるオペレーションにあまり役に立たない。以下の技法は、並列処理を効率的に使用してデータ圧縮及びデータ伸張に対処することを可能にする。

【００４２】

基本原理は、圧縮されるデータが、複数のデータブロックに分割されるということである。図６ａ～図６ｃでは、４つのこのようなブロック２００が示されているが、この個数は、単に図面を明確にするために選ばれたものである。実際には、使用中のプロセッサの並列処理能力に適合するようにブロックの個数を選ぶことができる。そこで、たとえば、フル仕様のセルプロセッサでは、最大８つまでのブロックが、フル仕様のセルプロセッサ内の８つの利用可能なサブプロセッサを効率的に活用するのに適している。

【００４３】

ブロックは、別個に圧縮及び伸張されるが、１つのブロックの圧縮／伸張が、処理されているブロックだけでなく他のブロックに関連付けられた検索バッファ内の参照子（距離－長さ対による）を使用することを可能にすることができる。

【００４４】

特に伸張時における効率性を改善するために、ブロックは、サイズが可変にされ、ブロックサイズの導出及びデータの圧縮を行うのに、反復プロセスが使用される。この構成（後述）は、たとえば、ビデオゲームの処理中に使用されるビデオ又はテクスチャデータ（等）といった高速且つ効率的に伸張する必要があるデータをハンドリングするのに特に適している。多くの場合、ゲーム記憶媒体（たとえば、Ｂｌｕ－Ｒａｙ（ブルーレイ）（登録商標）ディスク）内にデータを適合させるために圧縮しなければならないが、ビデオフレーム期間内にうまく伸張する必要があるこのようなデータは大量に存在する。このような構成は、圧縮プロセスをプロセッサにより多く集中させることによって、改善された伸

張を提供するシステムを十分に利用することができる。

【００４５】

ブロックサイズは、ブロックがほぼ同じ命令サイクル数を伸張に要するように選択される。伸張時に必要とされるサイクル数は、伸張中に見つけられた一致するブロックのサイズに依存し、また、符号化する必要があるリテラルの個数及び分布にも依存する。一般に、試験的圧縮（trial compression）が、伸張時において必要とされる命令サイクル数を検出する唯一の信頼することができる方法と考えられている。

【００４６】

そこで、図６ａ～図６ｃを参照すると、図６ａは、圧縮されるデータの４つの（当初）等しいサイズのブロック２００を示している。各ブロックは、圧縮プロセスを受け、（プロセスの途中の或る時刻において）既に圧縮された各ブロックの網掛け部分及びまだ圧縮されていない網掛けされていない部分によって模式的に示されている。４つのブロックの圧縮プロセスは、並列に実行することができるが、実際には、これは、圧縮時には必要ではない。

【００４７】

４つのブロックを圧縮する際の相対的な進行度は、各ブロックのコンテンツの性質に依存し、具体的には、圧縮がどれだけ容易であるのかに依存する。図６ｂでは、ブロックの境界が移動された場合に、変更されたブロック２００’の進行度が、はるかに一様なものとなっており、図６ｃのさらなる小さな移動（ブロック２００”）によって、さらにより一様な進行度が与えられることが分かる。

【００４８】

しかし、一様に近づけるべきものは、圧縮時間でもなければ、圧縮データの量でもない。一様にすべきものは、伸張時間、すなわち、より正確には、データを伸張するのに必要とされる命令サイクル数である。これを行うことによって、伸張時の並列プロセスのすべてが同時に開始して終了する。もちろん、正確な一様性が必要とされるわけではなく、合理的にできるだけ可能な一様性を得ることが目的とされているにすぎない。

【００４９】

このポイントを示すために、図７は、ちょうど説明したような並列圧縮技法を示す模式的なフローチャートである。

【００５０】

ステップ３００において、ソースマテリアル（圧縮されるデータ）を、２つ以上の隣接したデータブロックに分割する。最初に、これは、等しいブロックサイズに基づいた分割とすることができるが、ソースマテリアルの統計的解析に基づいたサイズの重み付けを導入することが可能である。

【００５１】

ステップ３１０において、各ブロックを、上述した技法の１つを使用して圧縮符号化する。

【００５２】

ステップ３２０において、各符号化ブロックを復号（伸張）するのに必要とされる処理ステップ（命令サイクル）の個数を検出する。これは、符号化データの統計的解析（たとえば、リテラル又は符号対のような各データアイテムが必要とする命令サイクルがどれだけあるのかの知識に基づいて、リテラルがどれだけあるのか、リテラルの連続した群の長さ、符号対がどれだけあるのか等）により導出することもできるし、実際の試験的伸張により導出することもできる。

【００５３】

ステップ３３０において、伸張処理所要量（decompression processing requirements）がブロック間で均一に分散されているか否かについての検出を行う。ここで、当業者は、「均一」は、「正確に均一」又は場合によっては「（たとえば）２％以内までで均一」を意味することができることを認識するであろう。

【００５４】

分散が（定義されように）均一である場合には、プロセスは終了し、（ステップ３１０において既に圧縮されたような）ブロックは、その個数の伸張プロセッサを使用する並列伸張用に適切に圧縮されている。一方、分散が（定義されたように）均一でない場合には、ブロック境界を、ステップ３４０において移動する。伸張するのにより多くの処理を必要としたブロックはより小さくされ、逆もまだ同様に行われる。ブロックサイズの変更は、伸張処理所要量の差に比例したものとすることができる。そこで、たとえば、４ブロックシステムでは、処理所要量（１００に正規化）が以下の通りである場合には、

【００５５】

【表１】

ブロックＡ	ブロックＢ	ブロックＣ	ブロックＤ
１１０	１００	９０	１００

【００５６】

ブロックＡのサイズは、１／１１だけ削減され、ブロックＣのサイズは、同じ量だけ増加される。これは、ブロックＢのブロックサイズを同じに保ちながら、ブロックＢの先頭境界及び終端境界の双方が移動することをおそらく意味する。（この例では）境界のこの変更のみがデータコンテンツに影響を与えることができ、したがって、ブロックＢを伸張するための処理所要量に影響を与えることができるので、サイズが変更されたブロックだけでなく、４つのすべてのブロックをステップ３１０において再び処理することが適切である。

【００５７】

伸張時には、各ブロックは並列に伸張される。たとえブロックサイズ（及び各ブロックの圧縮データのサイズ）がおそらく異なっていようとも、意図するものは、ブロック伸張プロセスがすべて同時に（又はほぼ同時に）開始して終了することである。図８は、ほぼ半分過ぎた時点でのこのような並列伸張技法を模式的に示している。各ブロックの伸張の開始点は、インジケータデータ（図示せず）によって示される。各圧縮ブロック２１０は、伸張されて、元のブロック２００に戻されており、これまでの進行度は、既に伸張されたデータを表す網掛け部分によって示されている。

【００５８】

図９ａ～図９ｃは、各ブロックの最初の１つ又は複数の復号文字に関する並列伸張技法の特徴を模式的に示している。詳細には、図４の伸張技法が使用される場合、ステップ１３０によって書き出された１６文字のブロックのオーバーラップを伴った問題が存在する可能性がある。

【００５９】

この問題は、ブロックが、それよりも大きなソースデータセットの隣接したセクションを表すことから生じる可能性がある。それらのブロックは、伸張されるとき、メモリの隣接したエリアに記憶される可能性がある。これは、或るブロックの伸張の終端における最後に書き込まれた１６文字群が、次のブロックの先頭における既に伸張された有効なデータに上書きされる可能性があることを意味し得る。

【００６０】

図９ａは、２つ出力ブロック２２０、２３０のみを模式的に示している。伸張プロセスは、（描かれた）ブロックの左端から右に向かって進行する。ブロック２２０の終端部分及びブロック２３０の先頭部分のみが示されている。

【００６１】

最初の１６文字群が、ステップ１３０によってブロック２３０内に書き込まれる。これらの文字のいくつかは有効でない可能性があるが、少なくとも正に最初の文字が有効であることは事実である。ブロック２３０の伸張は、出力ポインタ（上記ステップ１５０の説明を参照）が前方に移動して、ブロック２３０の１６番目の文字を超えるまで続く。その時点で、ブロック２３０最初の１６文字の全体（網掛け群２３２によって図９ａに示される）が有効であることは事実である。その１６文字群は、出力バッファとは別個のストア

２３４へコピーされる。
【００６２】
伸張は、その後、上述したように進行する。ブロック２２０の伸張の終了に向かうにつれて（実際には、伸張はすべて、伸張プロセスのすべての終了に向かうにつれて、同時に終了する）、１６文字の最終ブロックが、ステップ１３０によって書き込まれる。これは、図９ｂに網掛けエリア２３６として模式的に示されている。データ２３６は、２つのブロックの境界にまたがっていることが分かる。ブロック２２０内に含まれる群２３６の文字は、もちろん、有効なデータである。境界を越えて含まれる文字、すなわち、ブロック２３０内の文字は、無効な文字であり、ステップ１３０のコピーオペレーションのアーティファクトである。しかし、これらは、ブロック２３０の先頭部分が伸張された時に、前に書き込まれた有効なデータに上書きされる。
【００６３】
解決法は、先行ブロック２２０の伸張が完了した後に、ブロック２３０の最初の１６個の文字内に、記憶されたデータ２３２を再書き込みすることである。
【００６４】
先行ブロック２２０の最後に書き込まれた１６文字群の少なくとも１つの文字は、その先行ブロック２２０内に含まれなければならないので、先行ブロックの伸張によって破損したあらゆるデータを元に戻すことを確実にするには、厳密には、各ブロックの最初の１５文字のみを保存することが必要であることが認識されよう。しかし、本システムは、１６文字データワードを効率的にハンドリングするように設計されているので、１６文字群２３２が保存される。
【００６５】
図１０は、上述した並列伸張技法を示す模式的なフローチャートである。このプロセスは、ソースデータが２つ以上の隣接したブロック内に圧縮されているものと仮定するが、図７に関して説明した並列オペレーションの効率性改善は、図１０の方法が有用となるための要件ではない。
【００６６】
ステップ３００において、各ブロックの伸張を開始する。１６個の有効な文字を伸張すると（たとえば、出力ポインタが１６番目の文字を過ぎることによって示される）、ステップ３１０において、最初の１６個の伸張文字群を保存する。すなわち、別個のメモリエリアに記憶する。
【００６７】
ステップ３２０において、伸張は、各ブロックの終端に続く。
【００６８】
最後に、ステップ３３０において、各ブロックの先頭からの保存された１６文字を、それらの元の位置へ再書き込みする。
【００６９】
明らかに、この対策は、必ずしも、特定のソースマテリアルから導出された隣接したブロックのセットの最初のブロックについて行う必要はない。しかし、各ブロックについて同じ処理を使用すること、すなわち、最初のブロックについてもこれらのステップを実行することは並列システムにおいてより簡単なものとなり得る。
【００７０】
最後に、図１１は、データ圧縮及び／又は伸張装置の模式図である。
【００７１】
この装置は、図を明確にするために簡単化した形態で示されている。当業者は、他のルーチンコンポーネントが必要とされる場合があることを認識するであろう。
【００７２】
この装置は、データメモリ４００（たとえば、圧縮データ又は伸張データを保持する検索バッファ及び出力バッファを記憶する）、プログラムメモリ４１０、中央処理装置（ＣＰＵ）４２０（セルマイクロプロセッサ等）、並びに入力／出力ロジック４３０を備える

。これらはすべて、バス４４０を介して相互接続されている。

【００７３】

この装置は、ＣＰＵ４２０がプログラムメモリ４１０に記憶されているプログラムコードを実行することによって、上述したオペレーションを実行する。このようなプログラムコードは、ディスク媒体４５０等の記憶媒体から得ることもできるし、場合によってはインターネット若しくは別のネットワーク４７０を介したサーバ４６０へのデータ接続等の伝送媒体によって得ることもできるし、他の既知のソフトウェア配布手段若しくはコンピュータプログラム製品によって得ることもできる。

【００７４】

もちろん、代替的に、図１１の機能は、注文ハードウェアによって達成することもできるし、ＡＳＩＣ（特定用途向け集積回路）又はＦＰＧＡ（フィールドプログラマブルゲートアレイ）等のセミプログラマブルハードウェアによって達成することもできることが認識されよう。

【００７５】

本明細書では、添付図面に関して、本発明の例示の実施形態を詳細に説明してきたが、本発明は、それらの正確な実施形態に限定されるものではないこと、並びに添付の特許請求の範囲によって規定されるような本発明の範囲及び精神から逸脱せずに、当業者は、さまざまな変更及び修正を実施形態において行うことができることが理解されるべきである。

【図１Ａ】

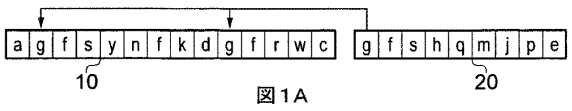


図1A

【図１Ｂ】

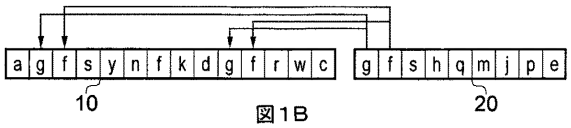


図1B

【図１Ｃ】

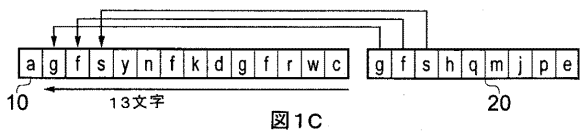


図1C

【図２Ａ】

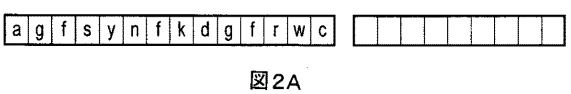


図2A

【図２Ｂ】

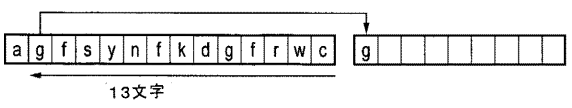


図2B

【図２Ｃ】

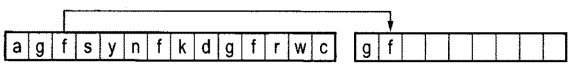


図2C

【図２Ｄ】

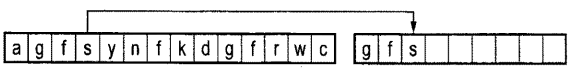


図2D

【図３Ａ】

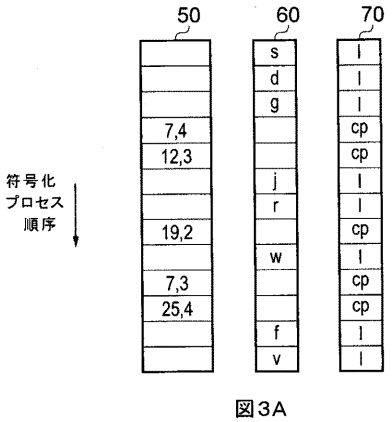


図3A

【図３Ｂ】

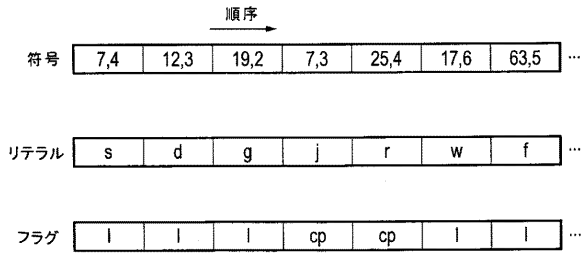
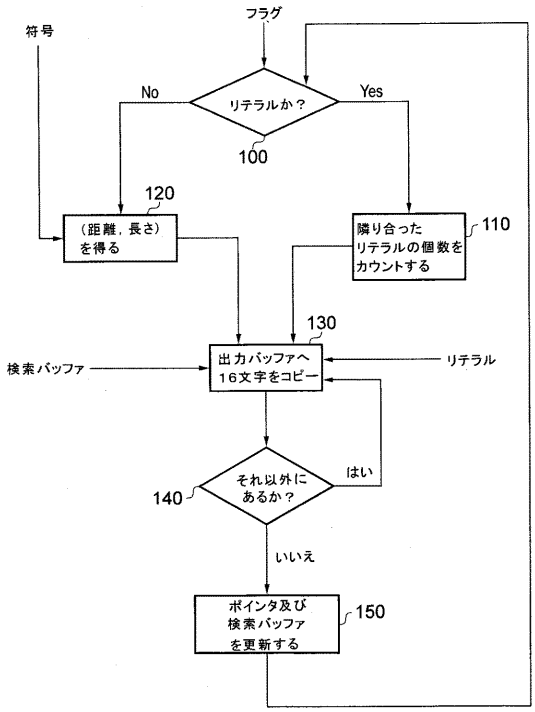


図3B

【図４】

図4



【図５Ａ】

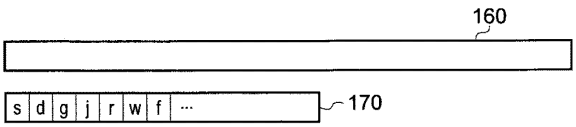


図5A

【図５Ｂ】

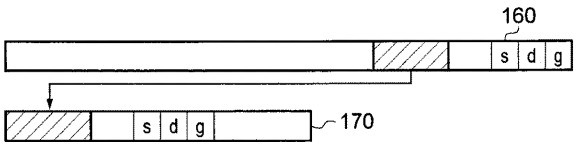


図5B

【図５Ｃ】

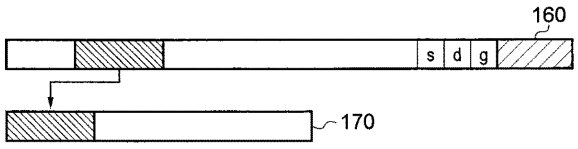


図5C

【図５Ｄ】

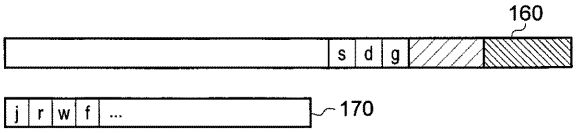


図5D

【図６Ａ】

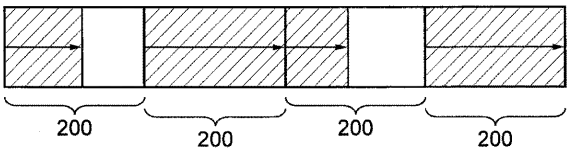


図6A

【図６Ｂ】

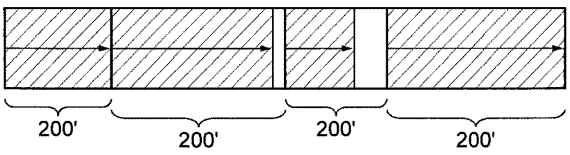


図6B

【図６Ｃ】

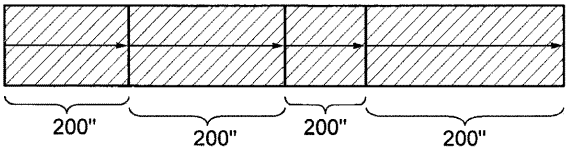
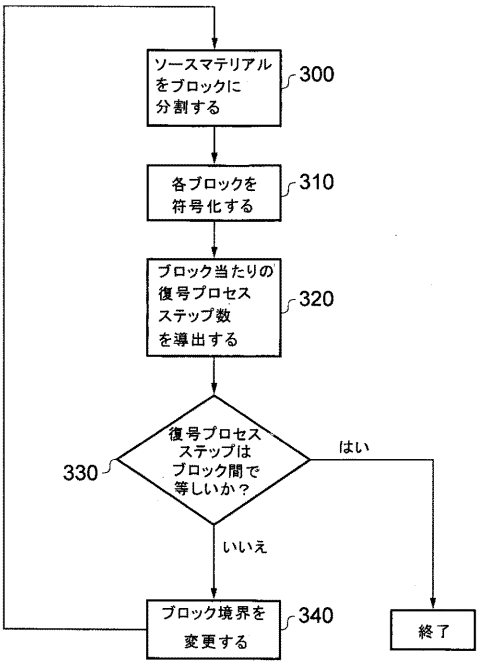


図6C

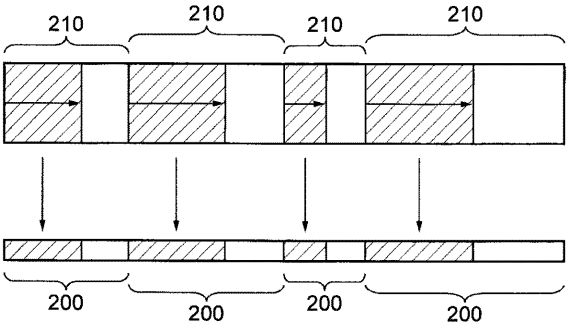
【図７】

図7



【図８】

図8



【図９Ａ】

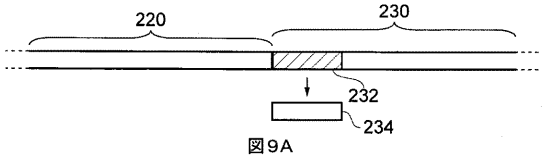


図9A

【図９Ｂ】

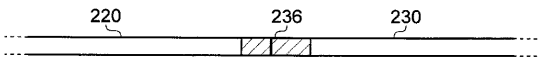


図9B

【図９Ｃ】

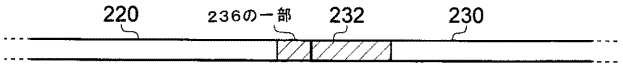
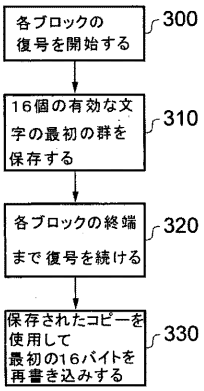


図9C

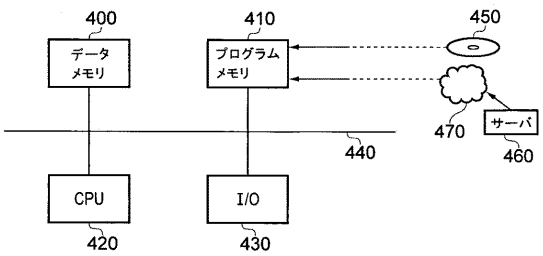
【図１０】

図10



【図１１】

図11



【外国語明細書】

DATA COMPRESSION AND DECOMPRESSION

BACKGROUND OF THE INVENTION

Field of the Invention

This invention relates to data compression and decompression.

Description of the Prior Art

Lempel and Ziv generated a number of lossless data compression algorithms such as the so-called LZ77 and LZ78 algorithms, which in turn form the basis for many variants including LZW, LZSS and others. They are both dictionary coders which achieve compression by replacing portions of the data with references to matching data that has already passed through both encoder and decoder. A matching section of data is encoded by a pair of numbers called a distance-length pair, effectively a counter (how many characters to copy) and a pointer (where to find the first such character in the already-encoded or already-decoded data). The distance-length pair can be seen as equivalent to the statement "each of the next ***length*** characters is equal to the character exactly ***distance*** characters behind it in the uncompressed stream."

During the encoding phase, the best possible match (of a string of characters to be encoded) is sought, which is to say that the system obtains the match within the available encoded data which gives the greatest length. This process will be illustrated schematically with reference to Figures 1a to 1c of the accompanying drawings.

Data which has already been encoded is stored in an area of memory called a “search buffer” 10. The search buffer may be big enough to hold all of the already-encoded data in a particular application, or may just hold a certain amount of most-recently-encoded data. Data to be encoded is stored in an area of memory called a “look-ahead buffer” 20. The first character in the look-ahead buffer 20 (in this case, the character “g”) is the next character to be encoded. It will of course be appreciated that alphanumeric characters are used in the present application to illustrate data to be encoded, and the data are referred to as “characters”, but this notation is just for clarity of the explanation. Naturally, it is not technically important whether the data represents alphanumeric characters, pixels, audio samples or whatever. Similarly, the size in bits of each such “character” is not technically important.

The encoder searches for instances of “g” in the search buffer 10. In the present example, two are found (Figure 1a).

The encoder then tests the position following each instance of "g" in the search buffer to detect whether the next character in the look-ahead buffer (the character "f") follows either instance of "g". In fact it does in both instances (Figure 1b) , so the search proceeds to the next following character in the look-ahead buffer ("s"). This follows only one instance of "gf" (Figure 1c), and the next-following character in the look-ahead buffer ("h") does not follow those three characters in the search buffer. So, the distance-length pair generated in respect of the first three characters of the look-ahead buffer is (13,3), where the string starts 13 characters back in the encoded sequence (the search buffer) and has a length of 3 characters.

If no matching strings of two or more characters are found in the search buffer, the character is encoded as a "literal", which is to say it is simply copied into the output data stream. So, the benefits of compression result from the use of distance-length pairs; the quoting of literals acts against the aims of data compression, because not only is a literal as large as the original character, there is also a need for some sort of a flag to indicate that it is a literal. Therefore, these techniques are well suited to data in which there is a good chance of repetition of character strings, in order to achieve a useful compression ratio.

In general, therefore, the compressed data generated by this type of process will be formed of distance-length pairs interspersed with literals.

Figures 2a to 2d schematically illustrate the decoding of a distance-length pair (13,3).

Figure 2a illustrates a search buffer providing the decoded data available as the distance-length pair (13,3) is about to be decoded. In figures 2b, 2c and 2d, a string of three characters starting from the position 13 characters back in the search buffer is copied across for output.

Data compression and decompression of this type are essentially linear processes, in that the successful encoding and decoding of the data relies on all previous data having been encoded or decoded.

An object of the invention is to seek to provide an improved data compression and/or decompression technique.

Summary of the Invention

This invention provides data decompression apparatus arranged to act on compressed data comprising:

an ordered stream of references to groups of previously decoded data items;

an ordered stream of direct representations of data items to be decoded; and

an ordered stream of flags indicating whether each successive decompression operation should act on a reference or a direct representation;

the apparatus comprising:

an output memory area;

a detector to detect the number ***n*** of consecutive flags indicating that a decompression operation should act on a direct representation; and

a data copier for copying to the output memory area either a next referenced group of previously decoded data or a group of ***n*** consecutive direct representations from the ordered stream of direct representations.

Further respective aspects and features of the invention are defined in the appended claims.

Brief Description of the Drawings

The above and other objects, features and advantages of the invention will be apparent from the following detailed description of illustrative embodiments which is to be read in connection with the accompanying drawings, in which: Figures 1a to 1c schematically illustrate a previously proposed data compression technique;

Figures 2a to 2d schematically illustrate a corresponding previously proposed data decompression technique;

Figures 3a and 3b schematically illustrate three streams of data;

Figure 4 is a schematic flow chart showing a decompression technique using the data streams of Figure 3b;

Figures 5a to 5d show a schematic worked example of data decompression in accordance with the flowchart of Figure 4;

Figures 6a to 6c schematically illustrate a parallel compression technique;

Figure 7 is a schematic flow chart showing the parallel compression technique;

Figure 8 schematically illustrates a parallel decompression technique;

Figures 9a to 9c schematically illustrate a feature of the parallel decompression technique relating to the first one or more decoded characters of each block;

Figure 10 is a schematic flow chart showing the parallel decompression technique; and

Figure 11 is a schematic diagram of a data compression and/or decompression apparatus.

Description of the Preferred Embodiments

Referring now to Figures 3a and 3b, the data signal output by a compression process such as that described with reference to Figures 1 and 2 has been arranged (by the compression apparatus or as an intermediate process) as three streams of data: an ordered stream 50 of distance-length pairs (called "codes" in Figure 3b); an ordered stream 60 of literals; and an ordered stream 70 of flags. The flags indicate whether the next data item to be decoded is a distance-length pair or a literal.

Figure 3a shows the streams 50 and 60 as they might be subdivided from an interspersed single stream of distance-length pair and literals as generated by the technique of Figure 1. The flags are schematically shows as "l" (literal) and "cp" (code pair, or distance-length pair.

In Figure 3b the streams have been concatenated so as to remove gaps. At decoding, the next entry in the flag stream indicates whether to take the next item of encoded data from the code stream or the literal stream. Pointers (not shown) are maintained to indicate the next item in the code stream and the next item in the literal stream. The relevant pointer is updated (moved forward by one item) each time an item is taken from a stream for decoding.

Figure 4 is a schematic flow chart showing a decompression technique using the data streams of Figure 3b.

At a step 100, the next flag in the flag stream is examined to detect whether it indicates a literal or a code pair. If it indicates a literal, then at a step 110 the number of adjacent (consecutive) literals is established by counting the number of "literal" flags following the current one. This count can be done using normal analytical techniques within a data processing system, but in the present embodiment (see Figure 11 below) using a "Cell" microprocessor developed by Sony Computer Entertainment ®, Toshiba ® and IBM ®, the microprocessor actually has a single command which counts the length of a string of identical bits. So, the step 110 can be carried out quickly and easily by such a Cell microprocessor.

If the current flag does not indicate a literal, then the next code pair is examined at a step 120 to retrieve the distance and length quantities represented by that code pair.

At a step 130 a copy operation is carried out, in which a block of 16 adjacent characters is copied to a data output buffer (i.e. a place or area in memory where decoded data is stored):

(a) In the case that the current flag indicated a literal, the next 16 literals are copied from the literal stream to the output buffer.

(b) In the case that the current flag indicated a code pair, the 16 characters following the position in the search buffer indicated by the distance variable are copied to the output buffer.

The reason that a block of 16 characters is copied across, regardless of the number of characters actually indicated by the number of consecutive literal flags or by the length variable, is that the Cell microprocessor is particularly designed for efficient copying of blocks of this length. Naturally, it is quite unlikely that the actual number of literals or the length variable is exactly equal to 16. So, various measures have to be applied to deal with this.

Considering first the situation where the number of literals or the length variable is greater than 16, at a step 140 this situation is detected (the question "Any more?" detects whether there remain any more characters to be copied, i.e. whether the original number of literals or length variable, less the number of characters already copied, is greater than zero) and control passed back to the step 130 to copy another block of 16. This continues until the number of literals or previously decoded characters copied by the step 130 reaches or exceeds the required number.

At that point, control passes to a step 150. At this step, various pointers are updated. A pointer indicating a current output position in the output buffer is updated to show the last valid character written into the output buffer by the step 130 (i.e. it indicates the extent of validly decompressed data already written to the output buffer). Recalling that the step 130 copied 16 (or multiples of 16) characters so as to reach or exceed the required number, the last validly copied character will be somewhere within the last 16 characters copied by the step 130.

For example, assume that the number of consecutive literal flags detected by the step 110 was 7. At the step 130, 16 literals were copied into the output buffer, starting at the current output pointer position. The number 16 (in this example) is a constant, irrespective of the number (7) of valid literals. But only the first 7 of these 16 copied literals are valid decoded data. Accordingly, the output pointer is updated so as to move seven characters forward in the output data buffer. This means that the next copy operation by the step 130 will start (in the output buffer) not at the end of the previous copy operation but seven characters after the start of the previously copied block, thereby overwriting the previously copied but invalid last 9 characters.

One of another two pointers need to be updated at the step 150, depending on whether the copy operation concerned literals or a code pair. If the last copy operation

concerned literals, then a pointer to the next literal (in the literal stream) to be used is updated (in the example, by seven positions). If the last copy operation concerned a code pair, then a pointer to the next available code pair in the code pair stream is updated.

Finally, the contents of the search buffer are updated to reflect the most recently decoded data. The updating of the search buffer involves copying the newly added and valid characters in the output buffer to the search buffer, to reflect the decoding performed in that operation cycle.

Control then returns to the step 100 for the next cycle of operation.

The use of the separate streams of data, particularly (though not exclusively) with a Cell microprocessor, gives an efficient decoding process. The steps 110 or 120 are carried out by just one instruction, as is the step 130. So the whole cycle from the step 100 to the step 150 can be completed in a relatively small number of instructions.

An example Cell processor implementation shown below wouldn't be five instructions, but a particular hardware implementation could be as little as 5 instructions.

[

```
// Fetch 32 bytes ( aligned to 16 bytes ) that contain required unaligned 16 byte stream

lq b0to15,base,distance
lq b16to31,base16,distance

// Generate 16 byte aligned stream, with repeats as needed

shufb aligned,b0to15,b16to31,AlignRepeat

// Aligned 16 write block may contain 0-14 previously valid bytes

lq write,dest

// Insert stream up to aligned end

shufb write,write,aligned,insert0
sq write,dest
```

```
// Generate overflow into next 16 byte aligned block
shufb write,aligned,aligned,overflow
sq write,dest+16
```

]

Figures 5a to 5d show a schematic worked example of data decompression in accordance with the flowchart of Figure 4. The streams of data shown in Figure 3b are used as inputs to this worked example. A search buffer is shown schematically as the block 160 and a sixteen bit group copied by the step 130 is shown schematically as a block 170.

The first three flags in Figure 3b are literal flags. Therefore, the step 130 copies 16 literals for output, starting from the literal "s" (Figure 5a). Only the first three of these are valid, and so the step 150 results in "s", "d", "g" being written to the search buffer, and the "next literal" pointer being updated to point to the literal "j" in the literal stream of Figure 3b.

The next flag is a code pair flag. In Figure 5b, the first code pair (7,4) is read from the code pair stream, and an appropriate copy of 16 characters is made from the search buffer to output. Note that these characters include the recently-written characters s, d, g from the search buffer. The first 4 of these characters are valid and are copied back to the search buffer. If there are not enough characters in the search buffer the constant 16 character copy is made up of duplicated versions of these codes. The need for this is detected if the distance variable is less than 16 and is implemented as a simple table.

ie:

Distance 7 : 16 characters are { 0,1,2,3, 4,5,6,0, 1,2,3,4, 5,6,0,1 }

Distance 9: 16 characters are { 0,1,2,3, 4,5,6,7, 8,9,0,1, 2,3,4,5 }

The next flag is also a code pair flag. The next code pair (12,3) is read and the process carried out in Figure 5c as above.

Finally, the next two flags are literal flags. A block of 16 literals starting from "j" is copied in Figure 5d, of which the first two characters are valid data.

Note that in some embodiments the search buffer is not in fact needed in the form of a separate store. The information held in the search buffer is also held in the output buffer, so the function of the search buffer could be achieved simply by means of updatable pointers to the validly decoded data held in the output buffer. The step of updating the search buffer (part of the step 150 described above) would simply involve updating the pointers.

Figures 6a to 6c schematically illustrate a parallel decompression technique. This represents a further technique that is particularly (though not exclusively) useful in connection with the Cell microprocessor.

As described above, previously proposed techniques using code pairs and literals are essentially linear techniques. Linear processing does not lend itself well to operation by parallel processors such as the Cell microprocessor and others. The following technique allows parallel processing to be used efficiently to deal with data compression and decompression.

The basic principle is that the data to be compressed are divided into plural blocks of data. In Figures 6a to 6c, four such blocks 200 are shown, but this number has been selected simply for clarity of the drawing. In practice, the number of blocks could be chosen so as to suit the parallel processing capabilities of the processor in use. So, for example, with the full specification Cell processor, up to eight blocks would be appropriate so as to make efficient use of the eight available sub-processors within the full specification Cell processor.

The blocks are compressed and decompressed separately, although it can be made possible for the compression / decompression of one block to use references (by distance-length pairs) into search buffers associated with other blocks as well as with the block being processed.

For improved efficiency, particularly at decompression, the blocks are made variable in size, and an iterative process is used to derive the block size and to compress the data. This arrangement (which will be described below) is particularly suitable for handling data which needs to be decompressed quickly and efficiently, for example video or texture data (or the like) used during the processing of a video game. Often there are large quantities of such data which have to be compressed in order to fit them into a game storage medium (e.g. a Blu-Ray ® disc) but which are required to be decompressed well within a video frame period. Such an arrangement can make good use of a system which provides improved decompression by means of a more processor-intensive compression process.

The block sizes are selected so that the blocks take about the same number of instruction cycles to decompress. The number of cycles needed at decompression depends on the sizes of matching blocks found during decompression and also the number and distribution of literals which need to be encoded. Generally, a trial compression is considered the only reliable method of detecting the number of instruction cycles needed at decompression.

So, referring to Figures 6a to 6c, in Figure 6a shows four (initially) equally sized blocks 200 of data to be compressed. Each is subjected to a compression process, illustrated schematically by a shaded portion of each block having been compressed (at a time part way through the process) and an unshaded portion not having yet been compressed. The compression processes for the four blocks can be run in parallel though in fact this is not necessary at the time of compression.

The relative progress in compressing the four blocks depends on the nature of the content of each block, and specifically, how easily it compresses. In Figure 6b, it can be seen that if the block boundaries are moved, the progress through the revised blocks 200' is much more uniform, and a small further movement in Figure 6c (blocks 200'') again gives a more uniform progress still.

But it is not the compression time, or the quantity of compressed data, that should be made nearer to being uniform. It is the decompression time or, more precisely, the number of instruction cycles needed to decompress the data, that should be made uniform. By doing this, all of the parallel processes at decompression will start and finish at the same time. Of course, exact uniformity is not required; just an aim to get as uniform as is reasonably possible.

To illustrate this point, Figure 7 is a schematic flow chart showing a parallel compression technique as just described.

At a step 300, source material (data to be compressed) is divided into two or more contiguous blocks of data. Initially, this can be a division based on equal block sizes, though it is possible to introduce a weighting of the sizes based on a statistical analysis of the source material.

At a step 310, each block is compression-encoded using one of the techniques described above.

At a step 320, the number of processing steps (instruction cycles) needed to decode (decompress) each encoded block is detected. This can be derived by a statistical analysis of the encoded data (e.g. how many literals, lengths of consecutive groups of literals, how many code pairs etc, based on a knowledge of how many instruction cycles each such data item requires) or by an actual trial decompression.

At a step 330 a detection is made as to whether the decompression processing requirements are evenly distributed amongst the blocks. Here, the skilled person will appreciate that "evenly" can mean "exactly evenly" or perhaps "evenly to within (say) 2%".

If the distribution is even (as defined) then the process ends; the blocks (as already compressed at the step 310) have been compressed appropriately for a parallel decompression using that number of decompression processors. If, however, the distribution is uneven (as defined) then the block boundaries are moved at a step 340. A block which required more processing to decompress is made smaller, and vice versa. The changes in block size can be in proportion to the differences in decompression processing requirements. So, for example, in a four-block system, if the processing requirements (normalised to 100) are as follows:

Block A	Block B	Block C	Block D
110	100	90	100

then the size of block A is reduced by $1/11^{th}$ and the size of block C is increased by the same amount. This may well mean that both the start and end boundaries of block B move, while keeping the block size of block B the same. As (in this example) this change in boundaries alone can affect the data content and therefore the processing requirements for decompressing block B, it is appropriate to process all four blocks again at the step 310 rather than just the blocks which have changed in size.

At decompression, each block is decompressed in parallel. Even though the block sizes (and the size of the compressed data for each block) may well be different, the intention is that the block decompression processes all start and finish at (or nearly at) the same time. Figure 8 schematically illustrates such a parallel decompression technique, about half way through. The start point for decompression of each block is indicated by indicator data (not shown). Each compressed block 210 is being decompressed back to an original block 200, with progress so far being illustrated by a shaded portion representing already-decompressed data.

Figures 9a to 9c schematically illustrate a feature of the parallel decompression technique relating to the first one or more decoded characters of each block. In particular, if the decompression technique of Figure 4 is used, there can be a problem involving the overlap of the 16 character blocks written out by the step 130.

The problem could arise because the blocks represent contiguous sections of a larger set of source data. When they are decompressed, they will likely be stored in a contiguous area of memory. This could mean that the last-written 16 character group at the end of decompression of one block could overwrite valid data already decompressed at the start of the next block.

Figure 9a schematically illustrates just two output blocks, 220, 230. The decompression process proceeds from the left hand end of the block (as drawn) towards the right. Only the end portion of the block 220 and the start portion of the block 230 are shown.

A first 16 character group is written by the step 130 into the block 230. While some of the characters may not be valid, it is a fact that at least the very first character is valid. The decompression of the block 230 continues until the output pointer (see the description of the step 150 above) has moved forwards beyond the sixteenth character in the block 230. At that time, it is a fact that the whole of the first 16 characters of the block 230 (shown in Figure 9a by a shaded group 232) is valid. That group of 16 characters is copied to a store 234 which is separate from the output buffer.

Decompression then proceeds as described above. Towards the end of the decompression of the block 220 (in fact, as they all finish together, towards the end of all of the decompression processes), a final block of 16 characters is written by the step 130. This is shown schematically as a shaded area 236 in Figure 9b. It can be seen that the data 236 spans the boundary of the two blocks. Those characters of the group 236 falling within the block 220 are of course valid data. Those falling over the boundary, i.e. into the block 230, are invalid data – an artefact of the copy operation of the step 130. But these overwrite valid data previously written when the start portion of the block 230 was decompressed.

The solution is to rewrite the stored data 232 into the first 16 characters of the block 230 after the decompression of the preceding block 220 has completed.

It will be appreciated that because at least one character of the last-written 16 character group of the preceding block 220 must fall within that preceding block 220, it is strictly only necessary to preserve the first 15 characters of each block to be sure of reinstating any data corrupted by the decompression of the preceding block. But since the present system is designed to handle 16 character data words efficiently, a 16 character group 232 is preserved.

Figure 10 is a schematic flow chart showing the parallel decompression technique described above. This process assumes that source data was compressed into two or more contiguous blocks, though the efficiency improvements for parallel operation described with reference to Figure 7 are not a requirement for the method of Figure 10 to be useful.

At a step 300, the decompression of each block is started. Once 16 valid characters have been decompressed (as indicated, for example, by the output pointer going past the 16^{th} character), at a step 310 a group of the first 16 decompressed characters is saved, i.e. stored in a separate memory area.

At a step 320, decompression continues to the end of each block.

Finally, at a step 330, the saved 16 characters from the start of each block are rewritten to their original positions.

Clearly, this measure does not necessarily need to be taken for the first block in the set of contiguous blocks derived from a particular source material. But it can be more straightforward in a parallel system to use the same processing for each block, i.e. to carry out these steps even for the first block.

Finally, Figure 11 is a schematic diagram of a data compression and/or decompression apparatus.

The apparatus is shown in a simplified form for clarity of the diagram. The skilled person will appreciate that other routine components may be needed.

The apparatus comprises a data memory 400 (which stores, for example, the search buffer and the output buffer holding compressed or decompressed data), a program memory 410, a central processing unit (CPU) 420 (such as a Cell microprocessor) and input/output logic 430, all interconnected via a bus 440.

The apparatus carries out the operations described above by the CPU 420 executing program code stored in the program memory 410. Such program code could be obtained from a storage medium such as a disc medium 450, or by a transmission medium such as a data connection to a server 460, possibly via the internet or another network 470, or by other known software distribution means or computer program product.

Alternatively, of course, it will be appreciated that the functionality of Figure 11 could be achieved by bespoke hardware or by semi-programmable hardware such as an ASIC (application specific integrated circuit) or FPGA (field programmable gate array).

Although illustrative embodiments of the invention have been described in detail herein with reference to the accompanying drawings, it is to be understood that the invention is not limited to those precise embodiments, and that various changes and modifications can be effected therein by one skilled in the art without departing from the scope and spirit of the invention as defined by the appended claims.

I CLAIM:

1. Data decompression apparatus arranged to act on compressed data comprising:

an ordered stream of references to groups of previously decoded data items;

an ordered stream of direct representations of data items to be decoded; and

an ordered stream of flags indicating whether each successive decompression operation should act on a reference or a direct representation;

the apparatus comprising:

an output memory area;

a detector to detect the number ***n*** of consecutive flags indicating that a decompression operation should act on a direct representation; and

a data copier for copying to the output memory area either a next referenced group of previously decoded data or a group of ***n*** consecutive direct representations from the ordered stream of direct representations.

2. Apparatus according to claim 1, in which the data copier is arranged:

to maintain a pointer with respect to the output memory area indicating the extent of validly decompressed data already written to the output memory area;

to copy a group of ***m*** consecutive data items irrespective of the size of the referenced group or the number ***n***; and

to set the pointer in dependence on the size of the referenced group or the number ***n***.

3. Apparatus according to claim 1, in which the data to be decompressed comprises a series of two or more contiguous blocks of data, the apparatus comprising:

a data store, separate from the output memory area, for storing a first-decompressed group of data items corresponding to at least blocks other than the first block in the series; the data store being arranged to rewrite the stored group of data items to its original position after decompression of the preceding block in the series has completed.

4. Apparatus according to claim 3, in which the blocks are arranged in size so that each block requires substantially the same amount of decompression processing.

5. Apparatus according to claim 4, in which the blocks are decompressed in parallel.

6. A data decompression method arranged to act on compressed data comprising:

an ordered stream of references to groups of previously decoded data items;

an ordered stream of direct representations of data items to be decoded; and

an ordered stream of flags indicating whether each successive decompression operation should act on a reference or a direct representation;

the method comprising the steps of:

detecting the number ***n*** of consecutive flags indicating that a decompression operation should act on a direct representation; and

copying to an output memory area either a next referenced group of previously decoded data or a group of ***n*** consecutive direct representations from the ordered stream of direct representations.

7. A data compression method in which input data items are encoded either as references to groups of previously encoded data items or as direct representations of the input data items, the method comprising the step of generating as separate streams of output data:

an ordered stream of references;

an ordered stream of direct representations; and

an ordered stream of flags indicating whether each successive decompression operation should act on a reference or a direct representation.

8. A machine-readable medium on which is stored computer software comprising program code which, when executed by a computer, causes the computer to carry out a method according to claim 7.

9. Data compression apparatus in which input data items are encoded either as references to groups of previously encoded data items or as direct representations of the input data items, the apparatus being arranged to generate as separate streams of output data:

an ordered stream of references;

an ordered stream of direct representations; and

an ordered stream of flags indicating whether each successive decompression operation should act on a reference or a direct representation.

10. A data signal generated by a data compression process in which input data items are encoded either as references to groups of previously encoded data items or as direct representations of the input data items, the data signal comprising as separate streams of output data:

an ordered stream of references;

an ordered stream of direct representations; and

an ordered stream of flags indicating whether each successive decompression operation should act on a reference or a direct representation.

ABSTRACT OF THE DISCLOSURE

Data decompression apparatus is disclosed which is arranged to act on compres data comprising: an ordered stream of references to groups of previously decoded data ite an ordered stream of direct representations of data items to be decoded; and an orde stream of flags indicating whether each successive decompression operation should act o reference or a direct representation. The apparatus comprises an output memory area detector to detect the number ***n*** of consecutive flags indicating that a decompress operation should act on a direct representation; and a data copier for copying to the out memory area either a next referenced group of previously decoded data or a group o consecutive direct representations from the ordered stream of direct representations.

【図１Ａ】

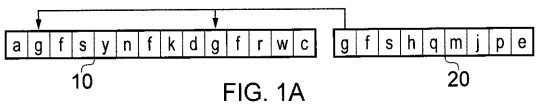


FIG. 1A

【図１Ｂ】

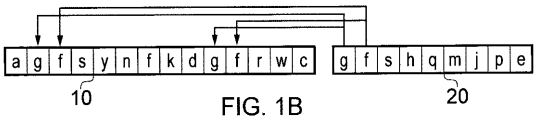


FIG. 1B

【図１Ｃ】

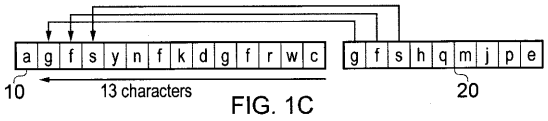


FIG. 1C

【図２Ａ】

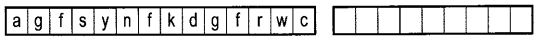


FIG. 2A

【図２Ｂ】

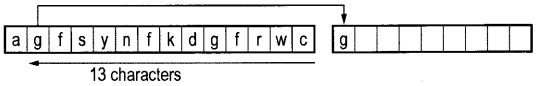


FIG. 2B

【図２Ｃ】

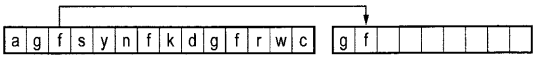


FIG. 2C

【図２Ｄ】

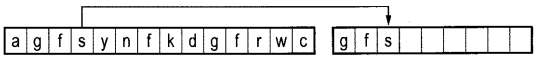


FIG. 2D

【図３Ａ】

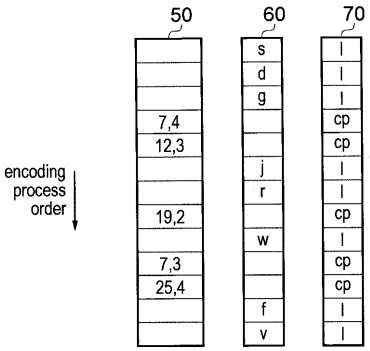


FIG. 3A

【図３Ｂ】

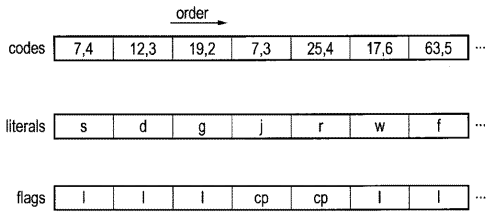


FIG. 3B

【図４】

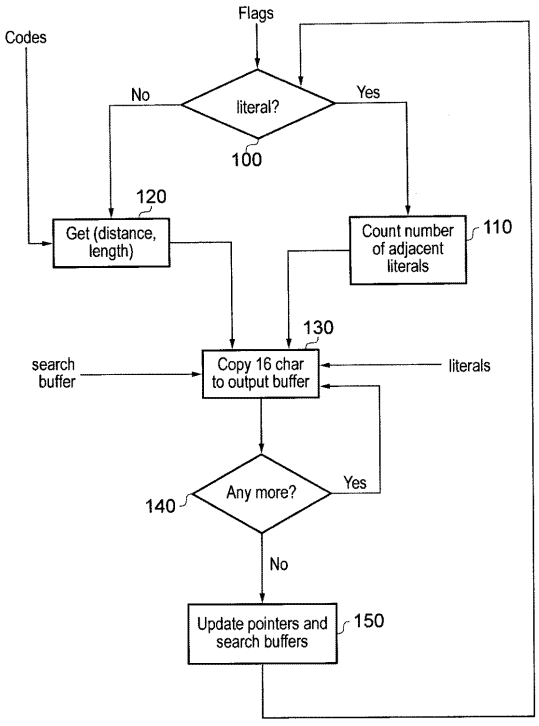


FIG. 4

【図５Ａ】

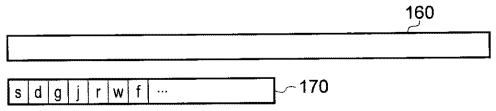


FIG. 5A

【図５Ｂ】

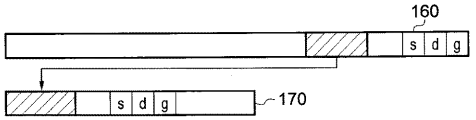


FIG. 5B

【図５Ｃ】

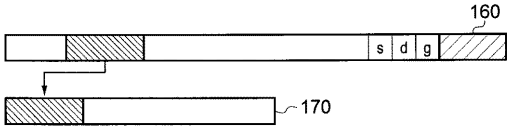


FIG. 5C

【図５Ｄ】

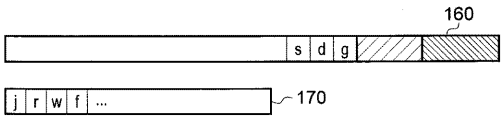


FIG. 5D

【図６Ａ】

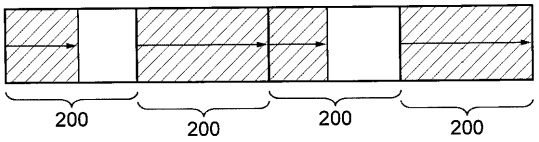


FIG. 6A

【図６Ｂ】

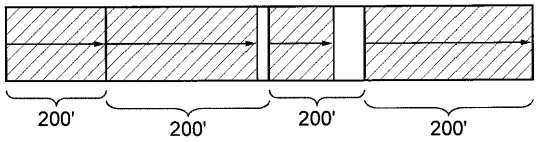


FIG. 6B

【図６Ｃ】

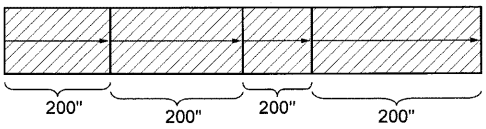


FIG. 6C

【図７】

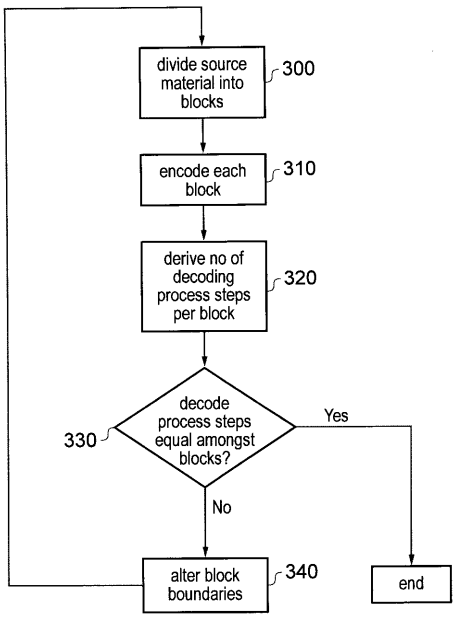


FIG. 7

【図８】

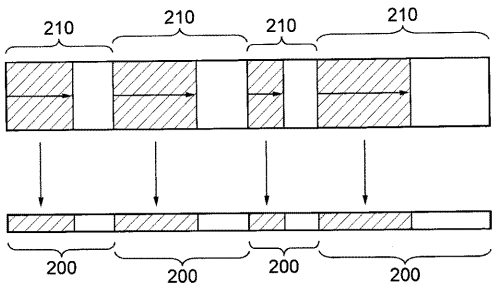


FIG. 8

【図９Ａ】

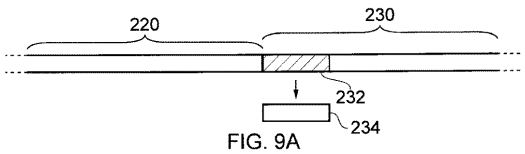


FIG. 9A

【図９Ｂ】

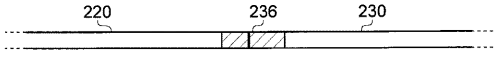


FIG. 9B

【図９Ｃ】

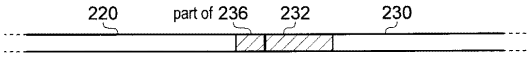


FIG. 9C

【図１０】

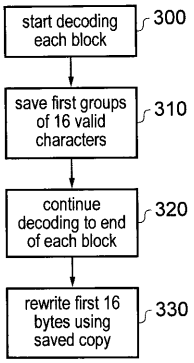


FIG. 10

【図１１】

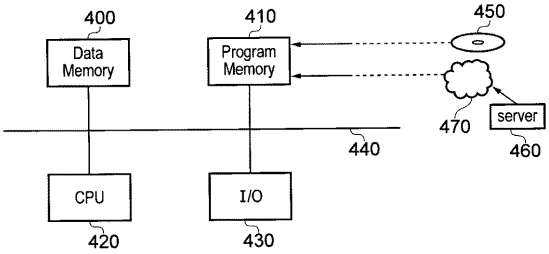


FIG. 11