2004－EVA－8　(15)
2004／3／19

ＩＢＭのｚシリーズでの文字列操作の命令セット

河野　知行　　株式会社アイ・アイ・エム[1]

コンピュータの利用形態が変化するに連れ、ＣＰＵ性能を向上させるための各種の工夫が施される。ＩＢＭのメインフレームも例外ではなく、常に素子技術や回路構成の見直しが行われている。また新たな命令セットが追加され、新たなアプリケーションの実行性能を確保している。この論文ではＩＢＭのメインフレームで提供されている文字操作の命令セットについて調査し、Ｃ言語環境などでの効果を探る。

IBM zSeries Instruction Set for String Handling

Tomoyuki Kawano, IIM Corporation[2]

In recent years, Web application becomes popular and is important for us to running a business. In order to support these new application, computer venders improve processor speed and provide new functions. IBM's zSeries also provides new instructions to support C language environment. This paper describes functions and performance of those zSeries's new instructions.

１、はじめに

インターネットが整備されコンピュータの利用形態も大きく変わった。またオープン・アーキテクチャの基で開発された数多くのミドルウェアが出現し、業務運用形態も変化した。

多くの企業で基幹システムとして稼動しているメインフレームも例外ではない。今まではオープン・システムとの連携機能を強化していたが、現在ではオープンシステムのミドルウェアをメインフレーム上で動作させることが可能となっている。

ＩＢＭの最新大型コンピュータであるｚシリーズでは、ｚ／ＯＳでＭＶＳやＯＳ／３９０環境で動作していたプログラム資産を継承している。また、ｚ／ＯＳではＪａｖａ実行環境のＷｅｂＳｐｈｅｒｅも稼動する。ｚシリーズでは、このｚ／ＯＳ以外にもＬｉｎｕｘを動作させることもできる。

ｚシリーズでは、ＭＶＳ以来のプログラム資産を継承するために、Ｓｙｓｔｅｍ３７０で採用された命令セットを提供している。また、Ｃ言語環境のプログラムの実行を支援する命令セットなども追加されている。

この論文では、ｚシリーズで利用可能な文字列操作の命令セットについて、その機能と性能を紹介する。今回の調査はハードウェアの仕様を中心としたものであり、Ｃ言語のランタイム・ライブラリなどの実行速度を計測したものではない。

２、文字列の移動と比較操作

文字列操作を行う場合、変数域の大きさは格納されるであろう文字列の最大長に合わせる。格納すべき文字列の長さが変数域の大きさに満たない場合、残りの領域にはパディング文字（埋めこみ文字）が格納される。移動命令（MOVE）や比較命令（COMPARE）では変数域の大きさをバイト長で指定し、変数域全体の操作を行う。

短い文字列を大きな変数域に格納すると、その操作に余分なＣＰＵ時間が必要となる。プログラムの実行時間を短くするには、変数域の大きさを注意深く設定すべきである。

文字列操作を多用するオープン系のプログラムでは、このルールに反し巨大な変数域に短い文字列を格納することが多い。Ｃ言語の場合、文字列の終端に特殊なコード（終端コード：１６進コードの００）を格納し、可変長の文字列操作を行っている。移動命令で、この終端コードを認識すれば無駄なＣＰＵ時間を節約することができる。比較命令では、終端コードを認識するか、パディング文字の格納を行わねば正しい比較結果を得ることができない。

ＩＢＭのメインフレームでは、文字列の終端コードを認識する命令としてＭＶＳＴ（MOVE STRING）やＣＬＳＴ（COMPARE STRING）を提供している。通常の移動や比較を行うＭＶＣ（MOVE CHARACTOR）やＣＬＣ（COMPARE CHARACTOR）では、変数域の開始アドレスと

[1] 東京都文京区本郷２－２７－２０
[2] 2-27-20 Hongo Bunkyo-ku Tokyo 1130033

大きさを指定する。一方、ＭＶＳＴ命令やＣＬＳＴ命令では、変数域の開始アドレスと終端コードを指定する。

ＭＶＳＴ命令は第２変数域に格納された文字列を第１変数域にコピーする。指定された終端コードは、第２変数域に格納された文字列の終端を検出するのに使用される。

ＭＶＳＴ命令が完了すると、第１変数域には第２変数域に格納された文字列と終端コードがコピーされる。また、第１変数域に格納された終端コードのアドレスが返される。その終端コードのアドレスから新たな文字列をコピーすれば文字列を連結したことになる。

ＣＬＳＴ命令は２つの変数域に格納された文字列を比較する。ＭＶＳＴ命令と同様に終端コードも指定する。ＣＬＳＴ命令では第１変数域と第２変数域の文字列の終端コードを意識しながら有効な文字列を比較し、比較結果を条件コード（Condition Code）にセットする。

３、文字列の検索操作

Ｃ言語では文字列に含まれるサブストリングの検索が頻繁に行われる。strstr 関数（ＶＢ言語の instr 関数に相当）を使用した文字列操作である。この処理を高速化するには、文字列に含まれる特定文字の位置を簡単に検索する機能が必要となる。

この文字検索を高速に行うには、ＴＲＴ（TRANSLATE AND TEST）を使用するのが一般的である。このＴＲＴは構文解析をサポートするために準備された命令であり、文字列に含まれる複数種の文字検索を行う。例えば、演算子（=+-*/）や括弧（{[()]}）などの文字の内、最初に現れた文字とその種類を知るために使用する。

ＴＲＴ命令では全ての１バイト文字に対応する変換テーブル（２５６バイト）を使用し、検索対象文字列の一文字一文字を検査する。変換テーブルに指定された制御情報がゼロの文字は検索対象外であり、ゼロ以外の文字が検索対象である。

ＴＲＴ命令は文字列検索に充分な機能を提供するが、文字列を構成する文字ごとに制御情報を生成する処理が必要となる。一方、サブストリングの検索ではサブストリングの開始文字だけを検索できれば良い。複数の文字を検索対象とはしないため、命令実行のオーバーヘッドの原因となる検索テーブルは不要である。このような高速検索処理を可能にするため、新たにＳＲＳＴ（SEARCH STRING）が準備された。

ＳＲＳＴ命令では、検索対象の変数域の開始アドレスと終端アドレス、それに検索対象文字コードを指定する。指定された文字列に検索文字を検出すると、条件コードをゼロにすると同時に、その文字を検出したアドレスが返される。

もう一つ、文字列検索のために準備された命令がある。ＣＵＳＥ（COMPARE UNTIL SUBSTRING EQUAL）である。この命令では２つの文字列を比較し、指定されたサブストリング（連続した文字列）に一致する文字位置を求める。例えば、ＡＢＣＤＥＦとＺＱＣＤＥＨは、第３文字目から３桁がＣＤＥで一致する。この２つの変数域をＣＵＳＥ命令で「３桁のサブストリングを探せ」と比較すると、このＣＤＥを見つけ、第３文字のアドレスを返す。残念ながら、このＣＵＳＥ命令の活用例を直ぐに思い浮かべることはできない。

４、文字列の変換処理

文字コードの変換を行う場合、ＴＲ（TRANSLATE）を使用する。この命令では変換対象の文字列が格納された変数域と変換テーブルを指定する。変換テーブルは２５６バイト長であり、１バイトの文字コード（０から２５５）に対応した変換後の文字コードを持つ。

ＴＲ命令は変数域から文字コードを１文字（１バイト）ずつ取り出し、変換テーブルを使用した文字コード変換を行い、元の文字位置に格納する。半角の小文字（a）を同じ半角の大文字（A）に変換するような際には非常に便利な命令である。しかし、一回の実行で変換可能な文字列は最大２５６バイトに制限されている。

２５６バイト以上の文字列を一気に変換するためにＴＲＥ（TRANSLATE EXTENDED）が準備されている。この命令はＴＲ命令と同じ機能を提供すると共に、文字列の終端コードの指定も可能としている。ＴＲ命令やＴＲＥ命令では文字列を１バイトずつの文字コードに分割し、文字コードの変換を行う。この方式を採用する限り漢字コードのような２バイト文字をサポートすることができない。

半角文字（１バイト文字）を全角文字（２バイト文字）に変換するなどのため、次の４種の命令が準備されている。

TRANSLATE ONE TO ONE (TROO)
TRANSLATE ONE TO TWO (TROT)
TRANSLATE TWO TO ONE (TRTO)
TRANSLATE TWO TO TWO (TRTT)

命令名称の後部の「ONE TO TWO」は１バイト文字を２バイト文字に、「TWO TO TWO」は２バイト文字を２バイト文字に変換することを意味する。文字コードの変換を正しく行うために、必要とする変換テーブルの

大きさは命令により異なる。その大きさは次の通りである。

命令	テーブルサイズ（バイト）
ＴROO	２５６
ＴROT	５１２
ＴRTO	６４K
ＴRTT	１２８K

表１　変換テーブルの大きさ

ＴROO命令は１バイト文字を１バイト文字に変換する命令であり、ＴＲＥ命令と等価である。何故２つの等価な命令が提供されているのか。ＩBMの説明によれば「ＴＲＥ命令はＴＲ命令の機能拡張版であり、ＴROOは他のＴＲｘｘ命令と一貫した機能を提供するもの」としている。具体的な違いは、変換終了の条件判定にある。

ＴＲＥ命令もＴROO命令も変換文字列の終端コードを指定するが、ＴＲＥ命令では変換前の文字コードが比較対象である。一方、ＴＲｘｘ命令では変換後の文字コードが比較対象となっている。

変換前後の文字コードが１バイト文字や２バイト文字であれば前述の命令で処理できる。しかし、文字セットによっては文字コードによりバイト数が変化するものがある。ＵＴＦ－８（Unicode Text Format）である。

ＵＴＦ－８はＪａｖａなどで使用される Unicode をテキストデータとして入出力する際に使用されるコード体系である。Unicode のデータを効率的に入出力するため、Unicode を１バイトから４バイトの文字コードに変換する。また、入出力制御文字として使用されている１６進コード（ＦＥやＦＦ）を生成しないようにしている。

Unicode をＵＴＦ－８に、もしくはＵＴＦ－８を Unicode に変換するロジックは煩雑であるため、この文字変換をサポートする命令が準備されている。ＣＵＵＴＦ（CONVERT UNICODE TO UTF-8）とＣＵＴＦＵ（CONVERT UTF-8 TO UNICODE）である。

２バイト文字（Unicode など）の移動・比較操作を行うための命令としてＭＶＣＬＵ（MOVE LONG UNICODE）とＣＬＣＬＵ（COMPARE LOGICAL LONG UNICODE）が準備されている。

ＭＶＣＬＵ命令では、第３変数域に格納された文字列を第１変数域にコピーする。もし、第１変数域が長ければ、残りのメモリ域には２バイトのパディング文字が格納される。

ＣＬＣＬＵ命令では、２つの変数域に格納された２バイト文字の文字列を比較する。この命令では文字列の終端コードは指定できず、有効な文字列をバイト長で指定する。一方の変数域が短い場合、指定した２バイトのパディング文字を使用して両変数域の長さを整えた後、比較操作を行う。

５、一つの命令での連続ＣＰＵ使用

巨大な文字列（例えば全体が数メガバイト）を移動・比較・検索する場合、キャッシュ（ＨＳＢ：High Speed Buffer）ミスが頻発するであろう。機能は単純であるが、ＣＰＵに取っては効率が悪い命令である。

昔からメインフレームで使用されていたＭＶＣ命令やＣＬＣ命令などでは、操作できる文字列の最大長は２５６バイトに制限されている。これらの命令では命令実行中にハードウェア割り込みが発生しても、その実行を中断することはない。必要であれば命令が完了した時点で、割り込み処理が行われる。

ＭＶＣやＣＬＣ命令の拡張版として提供されるＭＶＣＬ（MOVE CHARACTER LONG）やＣＬＣＬ（COMPARE LOGICAL CHARACTER LONG）では、操作可能な文字列の最大長は１６メガバイトにまで拡張された。これら命令では命令実行中にハードウェア割り込みが発生した場合、ＣＰＵは即命令実行を中断する。命令が中断された時の経過情報は汎用レジスタ（General Purpose Register）に退避される。また次に実行する命令アドレスを格納するＰＳＷ（Program Status Word）は、実行を中断した命令を再実行するように調整される。

ハードウェア割り込みの処理を行った後、このプログラムの実行が再開されると、自動的に実行が中断された命令の継続処理が行われる。この操作はハードウェアとＯＳが連携して行うものである。アプリケーションプログラムは、命令が中断されたか否かを知ることはできない。

ＭＶＳＴやＣＬＳＴなどの命令では、命令実行中にハードウェア割り込みを検出すると強制的に条件コード３で命令の実行を中断する。その時の経過情報は汎用レジスタに退避される。ハードウェア割り込み処理を行った後、プログラムの実行を再開される。その際に実行されるのは実行が中断された命令の次の命令である。

アプリケーションプログラムでＭＶＳＴ命令などを使用する際、このような命令実行の中断処理を意識したコーディングを行う必要がある。具体的には、これらの命令が条件コード３で完了した場合は、命令実行が中断されたと判断し、ブランチバック（もう一度、中断命令に戻る）する処理を追加する。例えば、次の

ようなコーディングが必要となる。

```
MVST            <- 移動命令
BC    1,*-4     <- ブランチバック
```

何故、このような処理を必要とするのか。理由は定かでないが、歓迎すべきことではないように思われる。

６、移動命令の性能

今まで説明した命令の実行性能を見てみよう。最初に試験環境の説明する。使用したＣＰＵはＩＢＭ２０６４－１０３（ｚ９００）である。このＣＰＵはＰＲ／ＳＭ（Processor Resource / Storage Manager）環境で動作している。

アッセンブラでプログラムを作成し、目的の命令を１，０４８，５７６回繰り返し、そのプログラムのＣＰＵ時間を実測した。この繰り返し回数に大きな意味はない。プログラムの開始・終了に必要なＣＰＵ時間よりも、試験対象のロジックで充分大きなＣＰＵ時間を消費させれば良い。

複数ＯＳを単一プロセッサ上で動作させるＰＲ／ＳＭ環境のプログラム実行であるため、常に一定のＣＰＵ時間が報告されることはない。このためプログラムを複数回実行し、その平均ＣＰＵ時間を求めた。また今回は単一命令の実行時間の測定が目的ではなく異なった命令の実行時間比較であることから、ＰＲ／ＳＭ環境であっても問題はないと判断した。

最初に移動命令（MOVE）の実行時間を考察する。比較対象に選んだのはＭＶＣＬ命令とＭＶＳＴ命令である。使用する変数域の大きさは３２キロバイトである。

ケース１として、ＭＶＣＬ命令で３２キロバイトのデータを主記憶内で移動する時間を実測した。１，０４８，５７６回の繰り返し実行で，そのＣＰＵ時間は７．６１秒であった。この際、２つ（入力側と出力側）の変数域の開始アドレスは、共にダブルワード境界に整えられている。これはＣＰＵに取っては、都合の良いものである。

通常、主記憶のアクセスはダブルワード単位で行われる。両方の変数域の開始アドレスがダブルワード境界に整えられていれば、ＣＰＵは移動操作をダブルワード単位で行うことができるからである。

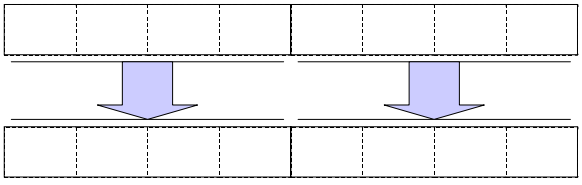

図２　ケース１での移動操作

ケース２では、このダブルワード境界のメリットを除外したＭＶＣＬ命令の実行を試みた。入力側の開始アドレスはダブルワード境界であるが、出力側の開始アドレスをダブルワード境界から１バイトずらした。その際のＣＰＵ時間は９．５２秒であった。ケース１に比べ１．９１秒長くなった。

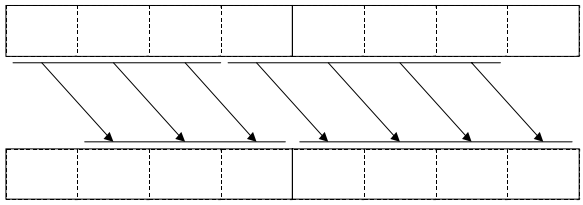

図３　ケース２での移動操作

ケース３では、ＭＶＳＴ命令での移動操作を行う。この場合、３２キロバイトの領域に終端コード以外の文字列を挿入し、３２キロバイト域の最終文字として終端コード（１６進コードの００）を格納した。２つ（入力側と出力側）の変数域の開始アドレスは、共にダブルワード境界に整えられている。この条件でＭＶＳＴ命令を実行すると、ＣＰＵ時間は５９．７３秒であった。

ＭＶＳＴ命令は巨大な変数域に短い文字列が格納されている場合の処理を目的としたものである。そのＭＶＳＴ命令で最大長の文字列が格納されている場合の処理を比較するのは適切でないかも知れない。しかし、ＭＶＳＴ命令では文字列の終端コードを判定する必要があるため、一つ一つの文字を検査している。そのオーバーヘッドを知るには、適切な試験であると信ずる。

ケース３とケース１のＣＰＵ時間の比率は７．８５（=59.73/7.61）である。このことから３２キロバイトのＭＶＣＬ命令と４キロバイトのＭＶＳＴ命令は同程度のＣＰＵ時間と考えられる。ＭＶＳＴ命令で４キロバイトの文字列移動を行った場合の実測ＣＰＵ時間は７．４秒であり、この換算は正しかった。

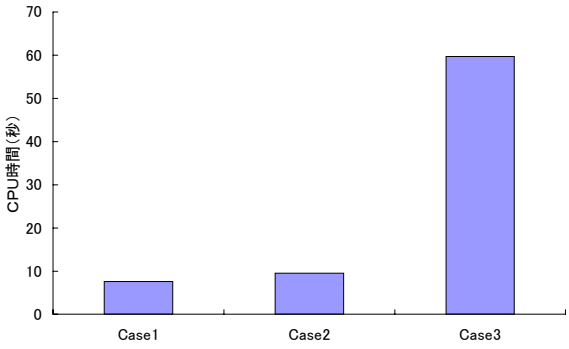

図４　移動命令の実行時間

７、比較命令の性能

比較命令でも移動命令と同様の試験を行った。ＭＶＣＬ命令の替わりにＣＬＣＬ命令を、ＭＶＳＴ命令の替わりにＣＬＳＴ命令を使用したプログラムである。何れのプログラムも３２キロバイトの同じ値を比較する。

ケース１で１３．２３秒、ケース２で１５．６０秒、ケース３で７６．３１秒であった。傾向は移動命令の場合と同じで、終端コード検索機能を持ったＣＬＳＴ命令のＣＰＵ時間が最も長い。また比較命令では２つの変数の比較処理が行われるため、移動命令よりも多くのＣＰＵ時間を必要としている。

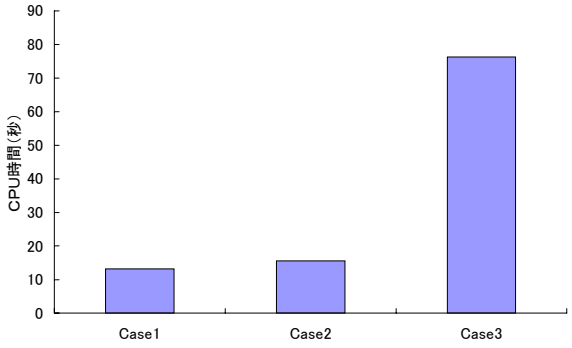

図５　比較命令の実行時間

８、検索命令の性能

文字列の検索処理を行うのは、ＳＲＳＴ命令である。この命令の性能を知るために、文字を１文字ずつ比較するＣＬＩ（COMPARE LOGICAL IMMEDIATE）での文字検索、ＴＲＴ命令での文字検索、それにＳＲＳＴ命令と３つの検索プログラムを準備した。何れのプログラムにおいても、３２キロバイトの文字列を検索し、３２キロバイトの最終文字が検索対象の文字である。

ＣＬＩ命令での検索（ケース１）が２２５．１４秒、ＴＲＴ命令による検索（ケース２）が４０．３３秒、ＳＲＳＴ命令による検索（ケース３）が６２．６３秒のＣＰＵ時間を必要とした。何故かケース２のＴＲＴ命令が一番早い。ＳＲＳＴ命令の性能改善が望まれる。

もう一つ、ＣＵＳＥ命令による検索も行った。ＣＵＳＥ命令ではサブストリングを検索するが、一方の変数域全体に同一文字を埋め尽くし、１文字のサブストリング検索を行うとＳＲＳＴ命令と同じ機能となる。それをケース４としてＣＰＵ時間を実測したところ４５１．３７秒となった。やはりＣＵＳＥ命令は、何か違った方法で活用すべきか。

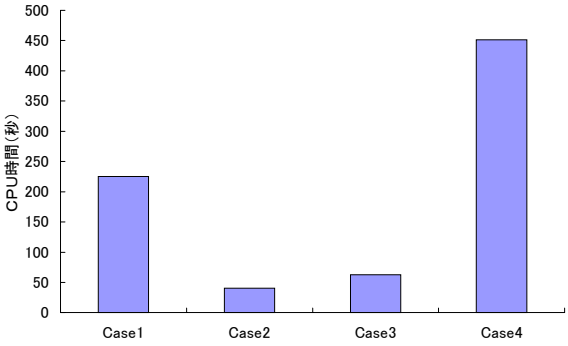

図６　検索命令の実行時間

９、チェックサムの取得

メインフレームでは入出力操作を行う場合、専用の入出力制御装置がデータの整合性を確認する。このため、プログラムでデータの整合性を確認するチェックサムを考慮する必要はなかった。オープン・アーキテクチャでは、プログラムでチェックサムを取り扱う必要がある。例えばＴＣＰ／ＩＰ通信である。

チェックサムの計算は簡単なようではあるが、演算時のキャリー操作が異なる。チェックサムを作成するために文字データの加算を行う際、最上位ビットからのキャリーは最下位ビットに循環（加算）させる必要がある。このようなロジックを作るとなると、その実行に相当なＣＰＵ時間を覚悟しなければならない。

ＩＢＭはチェックサムを算出する命令としてＣＫＳＭ（CHECK SUM）を準備している。この命令ではチェックサムを求めるべき文字列が格納された変数域を指定すれば、チェックサムを求めることができる。ＣＫＳＭ命令では３２ビットのチェックサムを求めるが、ＴＣＰ／ＩＰなどで使用する１６ビットのチェックサムに簡単に変換できる。

１０、ソート支援操作

文字操作命令にＣＦＣ（COMPARE AND FORM CODEWORD）がある。機能を調査しても理解するのが難しい。ＩＢＭが提供する使用例を見ると、ＵＰＴ（UPDATE TREE）と組み合わせて「ソートを高速化する」とのこと。

メインフレームの業務運用では、ソートは特別な処

理であり専用のプログラムを使用する。一方、オープン系のプログラムではメモリ内ソートを行うことが多い。このことからソートをサポートする命令群の整備が望まれるところである。

最初に、この２つの命令の概要を説明しておこう。ソート機能を提供するのはＵＰＴ命令である。ツリーと呼ばれるメモリ域を使用し、ソートを高速に実行する。ツリーはトーナメント手法をソートに適用したものである。但し、ＵＰＴ命令で取り扱うツリー方式では、ソートのキー情報は４バイト幅（厳密には３１ビット）に制限されている。

キー長が４バイトに制限されると、通常のソート処理で使用するのは難しい。そこで３２キロバイトまでのキー長であれば、それを４バイト情報に変換する機能を提供するＣＦＣ命令が準備された。単純に長いキー情報を４バイト情報に変換はできないため、基準値からの差分を４バイトの情報として算出するようにしている。

ＣＦＣ命令やＵＰＴ命令は、実行時のアドレッシングモードで取り扱うデータのビット幅やバイト長が変わる。ここでは２４ビットもしくは３１ビット・アドレッシングモードでの説明を行う。６４ビット・アドレッシングモードでの仕様は異なるので注意を要する。また、ここではＣＦＣやＵＰＴ命令機能の全ては紹介していないことをお断りしておく。

１１、ＵＰＴ命令

前述したようにＵＰＴ命令ではツリーを活用したソート機能を提供する。先ず、ＵＰＴ命令で使用するツリーについて紹介しよう。

ツリーは情報を格納するノードと、それらの親子関係を定義する。ノードには８バイトの記憶域が割り当てられる。図７に示した１から１５までの箱はノードを、矢印はノードの親子関係を示す。最上位のノード（ノード１）は特別にルーツノード（Root Node）と呼ぶ。親子関係はノード番号で決定されるものであり固定である。ノード番号はノード開始位置で決定され、ノード配列の順序に一致する。子ノードを持たない末端のノードを、特別に終端ノード（Terminal Node）と呼ぶ。

ノード群は連続した領域（ツリー域）に格納され、命令実行時にはその開始アドレスを指定する。ツリー域の先頭の８バイトはダミーノードと呼ばれ、使用されることはない。次の８バイトがノード１、その次の８バイトがノード２となる。

図７を参照願いたい。ノード１５の親ノードは７であり、ノード８の親ノードは４である。ここには規則性がある。親ノード番号は子ノード番号の半分（小数点以下切り捨て）となる。

前述したようにノードには８バイトの記憶域が提供される。この８バイトの領域には、上位４バイトにキー情報、下位４バイトに任意情報が格納される。通常、任意情報にはノードに対応するレコードの開始アドレスである。

ツリー領域を初期化する際、各ノードには有効な情報が格納されていない（空）状態を定義する必要がある。ＵＰＴ命令ではノード記憶域のキー情報がマイナス（最上位ビットが１）である場合、ノードは空であると判定する。

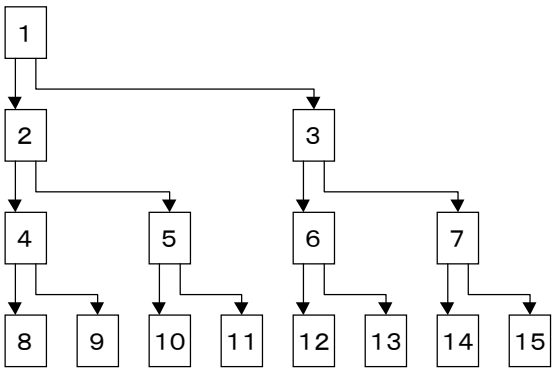

図７　ツリー構造のノード

ＵＰＴ命令を実行する際には、操作対象のノード番号とノード情報を指定する。ノード情報はトーナメントに新たに参加する情報であり、上位４バイトがキー情報、下位４バイトが任意情報である。

ＵＰＴ命令の実行が開始されると、指定されたノードの親ノードが検索される。例えば、ノード１５が指定されたとすると、その親ノード（ノード７、３、１）が検索対象となる。ノード１０番であればノード５、２、１が検索対象となる。

検索対象となった親ノードの記憶域の内容と指定されたノード情報が順次比較される。途中の親ノードでノード記憶域に大きなキー情報を見つけると、その記憶域の内容と指定されたノード情報が置換される。その後、親ノードの検索は継続される。ルートノードまでの検索が終了すると、その検索ルートで最大のキー情報を持っていたノード情報を取り出すことができる。

ＵＰＴ命令の実行には、２つの例外処理が準備されている。親ノードを検索している際にノード記憶域のキー情報がマイナス（そのノードが空の状態）であった場合、例外処理を行えるように命令実行を中断する。その場合、空のノードの記憶域には、それまでの検索で抽出されたノード情報が格納される。

検索処理中に、それまでの検索で抽出されたノード情報のキー情報と、検索対象のノード記憶域のキー情報が一致した場合にも命令実行を中断する。これは、ＵＰＴ命令をＣＦＣ命令と組み合わせて使用する際に必要とする処理である。

１２、ツリー操作とソート

実際のツリー操作とソート処理の実行過程を見てみよう。ここでは８から１５までの数値を持つ８つのレコードを降順にソートすることを考える。この場合、８つの終端ノードを持つツリーを使用する。また、説明を簡単にするため、データの値とそのデータが格納されるノード番号は一致しているとする。

図８に、このツリー域が初期化された状態を示す。アンダーライン「_」が表示されているノードは、空であることを示す。この状態でノード８から１４までを指定したＵＰＴ命令を合計７回実行すると、ツリー域の状態は図９の様に変化する。ＵＰＴ命令を実行する際、指定したノード番号の内容は空にする操作を行うものとする。

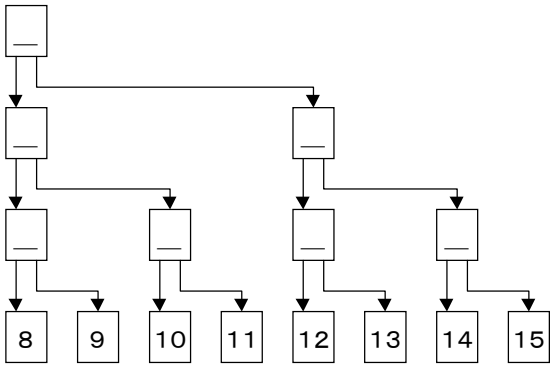

図８　ツリーの初期状態

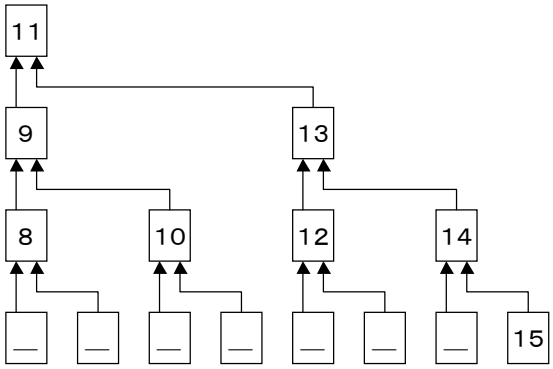

図９　終端ノードの更新

ＵＰＴ命令では、親ノードが空であれば指定されたノード情報を親ノードの記憶域にコピーして命令実行を完了する。このためノード８から１４までのＵＰＴ命令の操作では、それらのノードの内容が親ノードの記憶域にシフトされるだけである。

次にノード１５を指定した８回目のＵＰＴ命令を実行すると、親ノード（ノード７、２、１）の検索が行われる。それらのノードの値は１４、１３、１１であり、ノード１５の値（１５）よりも小さい。このためＵＰＴ命令はトーナメントでの最高位の値（勝者）として１５を返す。この操作の様子を、図１０に示す。

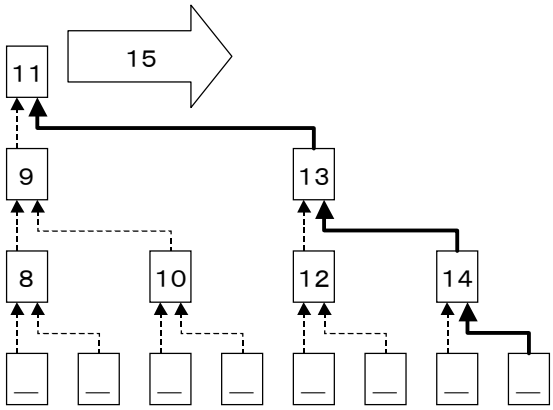

図１０　勝者の決定

ＵＰＴ命令を使用してソートを実現するためには、１つのルールがある。誰か勝者が出た場合、次の検索処理は勝者と対戦した敗者を優先すると言うものである。つまり、図１０の操作が終わった後は、ノード１５の検索ルートで有効なデータを持ったノードを対象にＵＰＴ命令を実行する。この例に従えば、次にＵＰＴ命令の対象となるのは１４の値を持つノード７である。

ノード７を指定した９回目のＵＰＴ命令を実行すると、ツリー域の状態は図１１のようになる。この実行で次の勝者は１４であることが判る。この方法でＵＰＴ命令を繰り返せば、合計１９回のＵＰＴ命令の実行でソートが完了することになる。

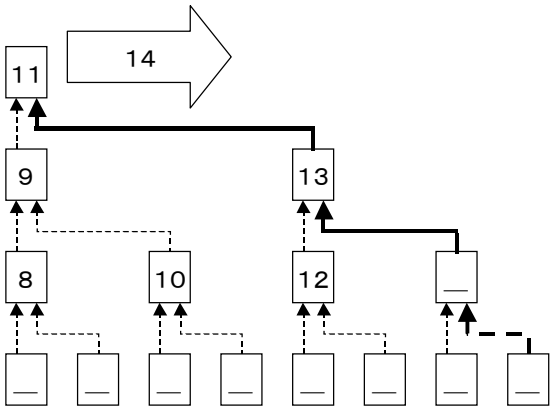

図１１　次の勝者の決定

１３、ＣＦＣ命令

前述したようにＣＦＣ命令では、３２キロバイトまでのキー情報を４バイトのＵＰＴ命令のキー情報（コードワード）に変換する機能を提供する。ＣＦＣ命令を実行する際には、通常の比較命令と同様に２つの変数域を指定する。その文字列（勿論バイナリ数値も可）比較を行い、その比較結果を条件コードとして通知すると同時にコードワードも作成する。

コードワードは、上位２バイトがオフセット、下位２バイトが文字列を示す情報である。指定された文字列が一致しない場合、不一致文字を検出した文字位置をオフセットとして、また不一致文字列の大きい側の２バイトを文字列としてコードワードにセットする。

指定された文字列が一致する場合には、例外処理として特別なコードワードが生成される。そのコードワードとは最上位ビットが１で、最上位ビット以外には第２変数域の開始アドレスが格納されている。つまりコードワードの値がマイナスであれば、２つの文字列が同じであったことを示す。

ＡＢＣＤＥＦとＡＢＸＹＥＦの２つの文字列を比較したとすると、３文字目から２文字がＣＤとＸＹで異なっている。この場合、コードワードとしてオフセットが４、文字列としてＸＹが格納される。実際にはコードワードを生成する際、文字列ＸＹの２の補数値が格納される。

何故、文字列の２の補数値をコードワードとして生成するのか。この解は、ＣＦＣ命令はソート支援のキー情報生成機能を提供するものであると言うことにある。ソートの場合、キー情報には範囲があろう。キーが取り得る最小値と最大値である。

最小値と実際のキー値を比較すれば、小さなキー値を持つキー情報の上位桁の多くは最小値に一致することになる。つまり生成されるコードワードのオフセットの値は大きくなる。また、最小値をバイナリのゼロと仮定すると、不一致文字列の２の補数を取ると、コードワードの値が大きければ大きいほど、比較対象のキー値はその最小値に近いことになる。

ＵＰＴ命令が降順のソート機能を提供するのは、ＣＦＣ命令で生成するコードワードが、キー値の最小値に近いほど大きくなるからである。また、ＵＰＴ命令でノードが空（ノード記憶域の最上位ビットが１）の状態を特別に管理しているのは、コードワードがマイナス（両者が一致）の場合の特殊処理を可能にするためでもある。

コードワードは、長いキー情報を４バイトの制御情報にまとめる、スマートな考え方である。但し１つ問題もある。最小値を元にして２つの文字列（例えばＡとＢ）に対して、それぞれのコードワードを生成したとしよう。それらのコードワードが一致するからと言って、両者（ＡとＢ）の文字列は一致するとは言えるであろうか。

前述したように、コードワードは不一致が検出されるまでの文字数と、不一致個所の文字列を示しているに過ぎない。同じコードワードであっても、コードワードで示されたオフセット以降の文字列が一致するかは定かでない。同じコードワードを見つけた際には、２つの変数（ＡとＢ）を指定したＣＦＣ命令で比較を行うと同時にコードワードを生成する必要がある。

ＵＰＴ命令でノードのキー情報（ＣＦＣ命令のコードワード）が一致した場合、命令実行を中断する。これはＣＦＣ命令で両者を比較した際のコードワードを新たに生成できるようにするためである。

参　考　文　献

1) z/Architecture Principles of Operation (SA22-7832-01), IBM Corporation