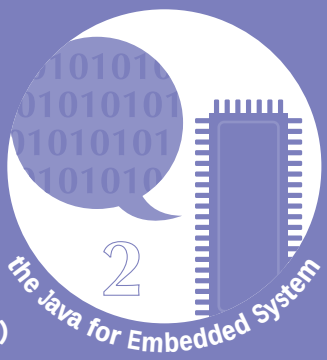
第2章

Javaチップのアーキテクチャ

― picoJava IIコア内蔵マイクロプロセッサの詳細

独古康昭

Java組み込み製品のキー・デバイスとしてJavaチップがある．Javaチップ(Javaプロセッサ)は，Javaバイト・コードをネイティブに実行するCPUである．まもなく実チップが市場に登場する．ここではJavaチップの心臓部となるpicoJavaコアの詳細について解説する．picoJavaコアには二つのバージョンがあるが，その両方をとりあげている．また筆者がスタンダード・セルを使って試作したJavaチップとその評価ボードについても紹介する．(編集部)



Javaプロセッサの歴史と将来

Javaは，米国Sun Microsystems社(以下，たんにSun社とする)のPatrik Naughton，Mike Sheridon，James Goslingらにより，1990年12月にOakという名称でGreenプロジェクトの一部として開発が始まった．そしてその5年後の1995年5月にJavaとして正式に発表された．

Javaのバイト・コードを直接実行できるCPUの開発としては，picoJava Iプロジェクトが1996年2月頃に発表され，1997年3月に最初のリリースが行われた．続いてC/C++の性能を2倍に向上させJava言語性能を約30％向上したpicoJava IIの開発が1997年に開始され，1998年にライセンシにリリースされている．さらに今後，picoJava IIIが開発される予定である．

picoJavaと汎用CPU+Javaの違い

○Javaをネイティブに実行できるCPU

Javaの実行環境はプラットホームに依存しないということが前提となっている．つまりこれは，CPUの種類にも依存しないことを意味している．

Java言語をより効率よく実行できるCPUとしてpicoJavaというコアが発表されたが，これはJavaからすれば，実行環境としてpicoJava CPUという選択肢が一つ増えたにすぎない．

しかし，picoJavaコアの決定的なメリットはJITやスタティック・コンパイラを使用しなくても，Javaをネイティブ速度で動作させることができるということである．

○汎用CPUにおけるプログラムの実行

▶C/C++の場合

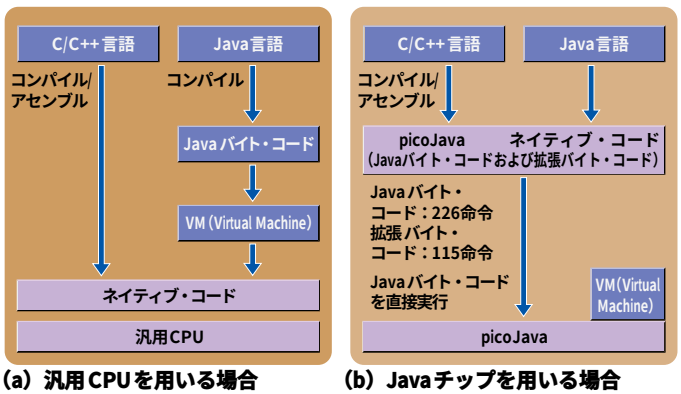
既存の汎用CPUにおいて，C/C++言語で書かれたプログラムを実行する場合を考える．まず，ソース・コードをコンパイルし，ターゲットとなるCPUのネイティブ・コード(機械語)に変換してから実行する(**図1**)．この場合，事前にコンパイルしたプログラムをオブジェクトとしてもっているので，実際のプログラム実行段階ではこのコンパイル時間を気にする必要はない．このC/C++プログラムのネイティブ・コードの実行速度は，当然コンパイラの善し悪しにはよるが，そのCPUに最適化されたコードで最高の性能を出す．

▶Javaの場合

Javaの場合は，コンパイルしてできるのはJavaバイト・コード(アプレット)である．これを実行する前に，ターゲットCPUのネイティブ・コードに翻訳し直す必要がある．具体的には，Javaバイト・コードをJVM(Java Virtual Machine)のインタプリタで逐次変換しながら実行する．このため，Javaの実行時間に対して変換時間は膨大になってしまう．また命令単位で変換するために効率良い最適化もできない．結果として，C/C++などで書かれたプログラムに対して何十倍も遅くなってしまう．

これを補うためにJITやスタティック・コンパイラなどの技術がある．しかし，それなりにメモリやディスクの容量が必要になってしまうことになり，コストとのトレードオフとなってくる．

〔図1〕Java言語とC言語の実行



(a) 汎用CPUを用いる場合　(b) Javaチップを用いる場合

○Javaチップにおけるプログラムの実行

Javaチップを使ってC/C++言語で書かれたプログラムを実行する場合を考える．ソース・プログラムはコンパイラにより，Javaチップのネイティブ・コード，すなわちJavaバイト・コードとC/C++コードに対応する拡張バイト・コードに置き換えられる．これは汎用CPUの場合とまったく同じ手順である．

しかし，Javaの実行に関しては大きく異なる．picoJava CPUは，Javaバイト・コードを直接実行することができる．汎用CPUを用いる際に必要だったコードの再変換が必要なくなる．

インタプリタ方式のJava実行であれば，同一周波数，同量のメモリを使用している場合，このJava専用CPUのpicoJavaがもっとも性能がよいといえる．

picoJava Ⅰコアのアーキテクチャ

○スタック・アーキテクチャ

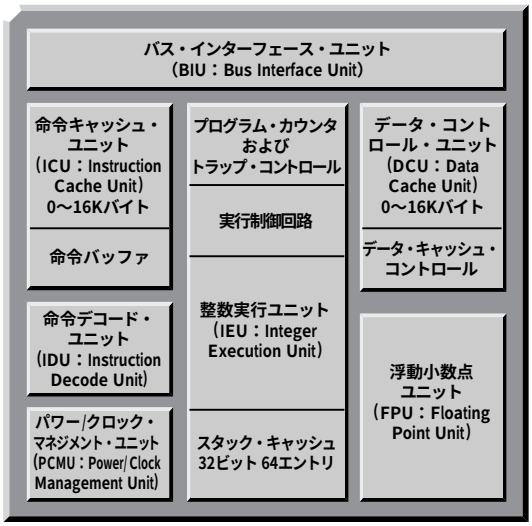
picoJava Ⅰは，**図2**のようなブロックで構成されている．スタック・アーキテクチャのCPUである．

Javaバイト・コードには，RISCマシンのようなレジスタまたはメモリに対するソースとデスティネーション・オペランドをもつ演算命令はない．基本的な命令は，PUSH，POP，LOAD，STOREであり，そのほかに演算命令が存在する．演算命令では，ソースとデスティネーションを指定する必要がない構造となっている．スタック・アーキテクチャと呼ぶ理由は，スタックという言葉のとおり，データを積み重ねて演算を行うので，その相対的な位置の変更と演算だけでデータ処理を行うからである．

たとえば基本的なIADDという命令では，スタックに積み重ねたデータの一番上と次のデータを加算して一番上に演算結果を詰めておくという動作になる．

Javaバイト・コード自体が，このような相対的なアドレスの演算構造になっている．その理由は，Javaがプラットホームに依存しない言語となるために，CPUに固有のアドレス指定方法または特定のレジスタを使用した構造になっていないためである．

おそらくpicoJava Ⅰもこのようなスタック構造になっていると思われる．



〔図2〕picoJava Ⅰコアのブロック図

▶picoJava Ⅰのブロック

全体的な構造としては，(F-D-E-W)であり，フェッチ，デコード，演算，ライトバックの4段パイプライン構造となっている．

▶ICU(命令キャッシュ・ユニット)

命令をフェッチするために，外部メモリから命令を読み込んでバッファとキャッシュに命令を格納する．キャッシュに格納されたアドレスにヒットした場合は，キャッシュから命令をバッファに格納する．命令キャッシュは8バイト・ライン・サイズのダイレクト・マップ・キャッシュである．

キャッシュのメモリ・サイズは0バイト，1Kバイト，2Kバイト，4Kバイト，8 Kバイト，16 Kバイトから選択可能となっていて，命令バッファは12バイトで1サイクル4バイト書き込みと5バイト読み出しが可能となっている．

▶IDU(命令デコード・ユニット)

命令を同時に2命令デコードし，その2命令がフォールディング可能であると判断した場合は，命令フォールディングで分類される1命令に置き換えを行う．この2命令フォールディングは，2命令を並列に実行することと等価の効果が得られる．

▶IEU(整数演算ユニット)

Javaと拡張バイト・コード命令の演算を行う．さらに以下のような内部ブロックがある．

- 32ビットALUとシフタ
- マイクロコードROM
- 内蔵レジスタ
- 乗算器/除算器
- トラップ発生回路
- 依存性チェック/検索回路
- スタック・キャッシュ

命令は以下のように分類されている．

- 算術演算/論理演算/シフト
- 整数乗算/除算
- スタック操作
- 即値のロード
- ローカル値からのロード/ストア
- 変換
- 分岐
- オブジェクト・フィールド操作
- 配列管理
- メソッド呼び出しと復帰
- モニタ・エンタ/エグジット
- トラップ命令

▶SMU(スタック・マネージャ・ユニット)

SMUは，IEUに対して必要なオペランドを貯蔵または供給したり，スタックのオーバフロー/アンダフローの状態を管理するユニットである．

スタック・キャッシュがオーバフロー/アンダフローする前に，ドリブル・マネージャという回路が，スタック・キャッシュの余分な値または足りない値をデータ・キャッシュに対して読み書きする．スタック・キャッシュの値を，つねに使用しているスタックの位置に合わせる機能である．

スタック・キャッシュとして使われている5ポート(3リード/2ライト)RAMのうち1リード/1ライトはつねにこのために使用されている．残りの2リード/1ライトは，フォールディング時にスタック・キャッシュの2箇所から読み出し演