組み込み技術者のためのオープンソースによるDSPアルゴリズム開発手法

第６回（最終回）ソフトウェア・パイプラインの活用

この連載では，ディジタル信号処理プロセッサ（DSP）の上で動作するソフトウェアのアルゴリズムを開発する方法を解説している．今回はハンド・アセンブリやソフトウェア・パイプラインの現状などについて説明する．（編集部）

冴木 元

● 時には機械のように

アセンブリ言語によるコーディングについて，「昔と今で一番違う点はどんなところか」と尋ねられたら，皆さんはどう答えますか? 筆者ならば，「昔と違って現在のアセンブリ・コーディングはシステマティックなノウハウが必要である」と答えます．その背景に，まず「CPUのアーキテクチャが飛躍的に複雑化したこと」が挙げられます．また，「CPUの設計はコンパイラの存在が前提となっていること」も無視できないと思います．

要するに，現在のプログラマは，アセンブリ言語によるコーディングを行う場合，コンパイラの最適化機能と競争して勝たなければなりません．コンパイラの生成コードの方が効率が良いのであれば，わざわざアセンブリ言語を人間が書く必要はありません．コンパイラは，一定の規則に従って最適化を施します．人間がコードを書く際も，「機械的に」コーディングしていくノウハウを確立することが欠かせないのです．今回紹介するソフトウェア・パイプラインというテクニックも，その例外ではありません[注1]．

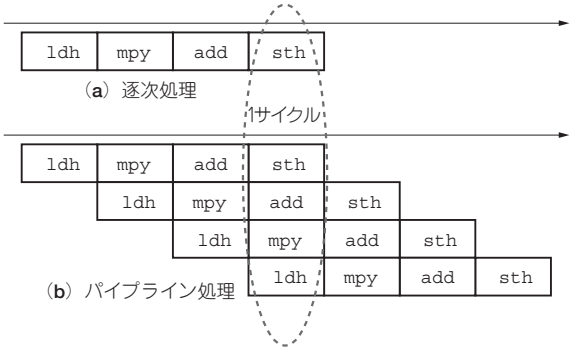


図1　パイプライン処理のイメージ

普通に逐次的に書くと1サイクルにつき1命令しか実行できない．1サイクルに複数命令を効率良くこなすようにプログラムを記述するやり方がソフトウェア・パイプライン．

● 1サイクルに複数命令を効率良く実行

ソフトウェア・パイプラインとは，どういったものでしょうか．一言で言えば，数段階にわたる処理を並行して同時に行うことです．**図1**（**a**）に挙げたように，逐次処理を行っていれば，1サイクルに1命令しか実行できません．しかし，命令を並列実行できるCPUの機能をフルに活用することで，1サイクルに複数の命令を効率良く実行できます．これがソフトウェア・パイプラインです〔**図1**（**b**）〕．

なおソフトウェア・パイプラインは，ループする処理にのみ効果があるテクニックです．ループ処理というのは処理が繰り返されるため，処理量を削減すれば，ループの回数だけ削減効果があります．例えば100回のループであれば，ループの中を1サイクル削っただけで，ループ全体では100サイクルの削減効果があります．信号処理のアルゴリズムにおいて，最適化の要となるフィルタ処理などはループ処理で実行されるので，ソフトウェア・パイプラインはこれらを最適化するために欠かせないテクニックです．

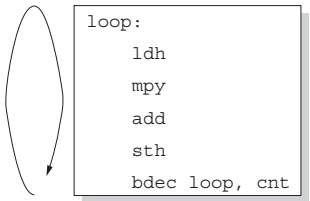
それではソフトウェア・パイプラインの具体的な説明に移ります．以下の1）～3）の順番に説明していきます．

1）遅延を無視したパイプラインの説明
2）遅延の説明
3）ソフトウェア・パイプラインの実際

注1：本稿で記した各命令の詳細な仕様は，TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide（spru732c.pdf）を参照．<CCS_INSTALL_DIR>¥docsに収められている．

▶ 図2
パイプラインなしのループ

パイプライン処理を行わない積和演算のループ．並列化を考えなければこのような処理になると思うが，最適化の余地は残る．どのようにすれば高速化できるか．ちなみにbdecとは，カウンタが負にならない間ジャンプする分岐命令．



```
loop:
    ldh
    mpy
    add
    sth
    bdec loop, cnt
```


KEYWORD ── DSP，C64x+，コンパイラ，コーディング，アセンブリ言語，ソフトウェア・パイプライン，bw_lpc，spkernel，SPLOOP

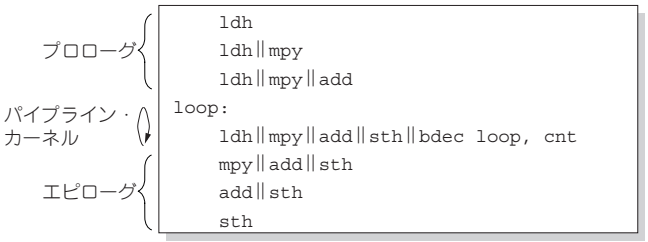


図3　パイプライン化したループ

最初のロードの結果は次のmpyによって処理されるが，並列化して次のロード命令が実行される．カーネル部分を繰りかえした後，エピローグを経てループを抜ける．ループ回数が望み通りになるように注意する．

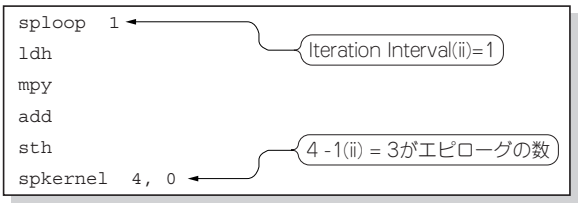


図4　SPLOOP命令

C64x+の命令であるsploopを用いると，パイプラインの記述は簡単になる．sploopの引き数は，繰り返しをまとめて行うマルチサイクル・ループ以外は1を指定する．spkernelの引き数はエピローグ処理の長さを指定する．

なお説明の便宜上，1)～2)については実際には実行できない疑似コードを用います．

● パイプラインの説明 ―― まずは遅延を無視

遅延を無視した状態で説明します．**図2**のような並列化されていない演算を考えます．これをパイプライン処理するとどのようになるのでしょう．もう一度，**図1**を眺めてください．まず，ロード→乗算→加算→ストアという各演算の前後の依存関係を保つことが必要です．そして，別のループの互いに依存しない演算は，並列に実行させます．つまり**図1**(**b**)のように演算を並べればよいのです．これらを満たすようなコードを書くと**図3**のようになります．

図3を見て分かるように，パイプライン処理には，「プロローグ」，「カーネル」，「エピローグ」と呼ばれる三つの部分があります．**図1**(**b**)でいうと，四つの演算が並列化されている，破線で囲った部分がカーネル部分にあたります．その前後に，段違いに命令を配置した部分が「プロローグ」と「エピローグ」に当たります．これらの命令を望みどおりのループ回数になるようにコードを記述すれば，ソフトウェア・パイプラインのできあがりです．

C64x+は，Software Pipelined Loop(SPLOOP) Bufferと呼ぶ機構を備えており，パイプライン化された命令のスケジューリングの記述が楽にできます．SPLOOPを用いたソフトウェア・パイプラインの記述を**図4**に示します．「段違い」に命令を記述していく従来のソフトウェア・パイプラインの記述方法は，コードの規模が大きくなったときに見にくくなるという欠点がありました．しかし，SPLOOPを用いたコード記述は**図4**のようにシンプルになります．

sploopとspkernelで挟まれた部分がソフトウェア・パイプラインによる演算部分です．**図3**の記述がどのように**図4**に移し変えられたのか，両者を見比べてください．

SPLOOPの引き数はiteration intervalと呼ばれます[注2]．ループの反復回数なので，繰り返しをまとめて行うようなマルチサイクル・ループ以外は1を指定します．従って，**図1**と**図3**は引き数が1です．マルチサイクル・ループについては，この後bw_lpcで実際のコーディングを紹介するときにもう一度説明します．

spkernelの引き数はエピローグ処理の長さを指定します．第1引き数の4からiteration intervalの1を引くと，エピローグの長さである3になっている点に注目してください．

● 遅延があるとうまく動かない

次に遅延について説明します．ロード命令を受けて乗算をするつもりだとしたら，**図5**(**a**)のコードは正しく動くでしょうか．C64x+のロード命令は遅延があるので，正解は「正しく動かない」です．**図5**(**b**)に，正しく動かないメカニズムを説明しています．ロード命令には4サイクルの遅延があります．つまり，**図5**(**b**)の⑤のところまでCPUのプログラム・カウンタが到着しないと，1行目のロード命令は結果が反映されません．**図5**(**a**)はロード命令の直後に乗算命令が置かれているので，2行目の乗算命令に1行目のロード命令の結果は反映されません．

図5(**c**)のように，4サイクル分のnopを入れてしまえば，命令の前後関係は保たれます．ただ，このやり方は効率が悪すぎます．「CPUが何もしないnop命令の代わりに，互いに依存しないほかの命令を並列化して用いれば，もっと効率の良いコードが書けるのでは？」，こう考える人も多いと思います．そのためのテクニックがソフトウェア・パイプラインです．

注2：DSP Libraryは前回紹介したように，http://focus.ti.com/docs/toolsw/folders/print/sprc265.htmlからダウンロード可能．