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5回　古くて新しいアセンブリ言語

本連載では，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．今回は簡易的にアセンブリ言語を記述するための言語体系であるリニア・アセンブリを紹介する．これを用いることで，アセンブリ言語によるコーディングが行いやすくなる．
（編集部）

冴木 元

● アセンブリ言語による実装のイメージ

皆さんは，アセンブリ言語による実装と聞くとどのようなイメージを持たれるのでしょう．「アセンブリ言語によるコーディングはお手のものだ．かつて○○の開発をしたときは…」と自慢話を始めるベテラン・エンジニアもいれば，「アセンブリ言語ってコードがバグったときにデバッガに出てくる変な命令のこと？今どき人間が書くことなんてあるの？」という初心者もいるでしょう．

本連載の今回と次回では，アセンブリ言語を用いたDSPアルゴリズムの最適化について解説します．冒頭のベテラン・エンジニアのような方は，今回用いるC64x+というDSPのアーキテクチャの複雑さに注意してください．C64x+は最適化に適したアーキテクチャを持つ反面，アセンブリ言語での記述が困難です．一昔前のプログラマが当たり前のようにアセンブリ言語を書いていた時代とはコード記述の難易度が違います．そして，C64x+用のコンパイラは非常に優秀であるため，コンパイラの生成コードより速いコードを書くことは容易ではありません．

ここまでの話を聞いて，アセンブリ言語の初心者はもうやる気をなくしてしまったかもしれません．というのは，C64x+の統合開発環境であるCode Composer Studio（CCS）にはコンパイラが付いているので，アセンブリ言語を書かなくても開発できるからです．しかし，C64x+の開発環境にはリニア・アセンブリと呼ばれる簡易的にアセンブリ言語を記述するための言語体系が用意されています．実際の最適化手順では，C言語をこのリニア・アセンブリで記述し直し，その後でハンド・アセンブリ，すなわちCPUの命令をプログラマがすべて書く工程に移る方が便利です．今回は，このリニア・アセンブリについて解説します．

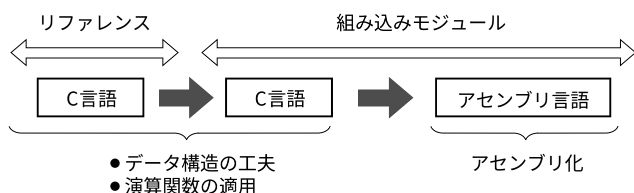

図1　最適化工程の位置づけ

工程の全体像をもう一度確認する．前回までに，データ構造の工夫と演算関数の活用方法を解説した．今回からはアセンブリ言語を駆使した最適化について解説する．

● C64x+ のアーキテクチャ

アセンブリ言語による個々の関数の最適化に先立って，設計段階における作り込みが不可欠となります．それが，データ構造の工夫（第3回，2008年4月号，pp.165-169）と演算関数の適用（第4回，同5月号，pp.182-188）であることは既に解説しました（**図1**）．

アセンブリ言語による最適化では，用いるDSPのアーキテクチャの理解が不可欠です．このプロセッサ・アーキテクチャの特徴は，コンパイラの最適化に適したアーキテクチャであるという点です．C64x+はA，Bの二つのデータパスが用意されています．それぞれ四つずつ（L，S，M，D）の機能ユニットが用意されています．全体で8個の機能ユニットがあるため，最大で8並列の命令を同時に実行できます．

そして，ロード・ストア命令はD，乗算命令はMというように各ユニットに命令が割り当てられています．また，各データパスには32ビットの汎用レジスタが32個ずつあり，全体で64個の汎用レジスタが用意されています．汎用レジスタを豊富に持ち，機能ユニットの制約を踏まえて一定の手順で命令を並べていきます．すると，8並列という特徴を生かしたソフトウェア・パイプラインを実装できるため，オプティマイザによって自動的に最適化を図る上で便利な構造となっています．

コンパイラが生成コードを最適化する際は，以上のよう

KEYWORD —— DSP，Speex，アセンブリ言語，リニア・アセンブリ，ハンド・アセンブリ，SPLOOP

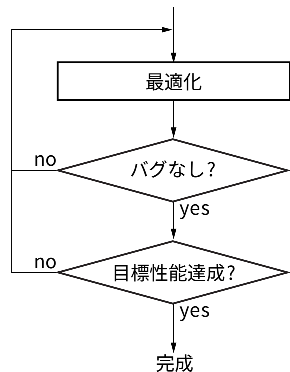

図2 最適化の進め方

最適化を行ってもバグを生まない仕組みがまず必要．従って，最適化作業は試験項目が確立していることが前提となる．少しずつ改造して処理量の削減効果を実測し，目標値に達したら完成となる．

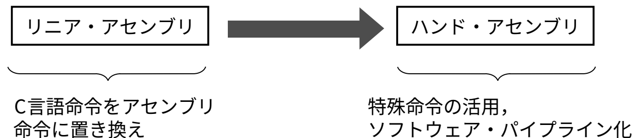

図3 リニア・アセンブリの活用

C64x+は，昔のシンプルなDSPと違ってアセンブリ・コードを手書きするのはかなり難しい．そこで，命令コードのみを記述すると，機能ユニットの割り当てや並列化をアセンブラが実行してくれるリニア・アセンブリを活用する．

なアーキテクチャがフルに活用されます．従って，C言語だけに頼った開発でもかなりの効率化を図れます．アセンブリ・コードを人間が記述せず，コンパイル時にオプティマイザによって行われる最適化に頼っても，かなりの性能が得られます．

実際の最適化作業の解説に入る前に，最適化工程の全体像について説明します(**図2**)．最適化を進めるに当たっての大前提は，最適化を施してもモジュールが仕様どおりに正しく動くことです．せっかく最適化作業を行ってモジュールの処理量を減らしたのにバグが混入してしまったのでは元も子もありません．従って，最適化を進める前に，以下の二つの項目を実施する必要があります．

1）テスト項目が確立されていることを確認する

DSPの組み込みアルゴリズムを実装する上でどのような項目をテストするのが望ましいかは，ページ数を費やして議論する価値のあるテーマですが，ここでは詳細な説明を省略します．最適化によってバグを混入しないために，テスト項目の確立が前提となることを忘れないでください．

2）最適化を少しずつ進めて削減効果を測定する

最適化作業は，仕様を満たすモジュールが通常のコーディングによりいったん完成してから行います．つまり新規開発ではなく改造の作業です．また，通常のコーディングと違って最適化コーディングは内容が非常に複雑です．

最適化を少しずつ進めるのではなく，一気に予定の全コーディングを終了してバグを混入させてしまい，さらに処理量削減の効果が大してなかったら元も子もありません．

● 簡易的にアセンブリ言語を記述する言語体系

冒頭でも述べたように，C64x+は最適化に適したアーキテクチャである反面，そのアセンブリ言語の記述は複雑です．例えば，C64x+はパイプラインが8並列で組まれているため，同時に最大8個の命令を並列実行できます．しかし，一部の命令は実行が1サイクルでは済まず，遅延が発生します．例えば，乗算は1サイクル，Loadは4サイクル，分岐は5サイクルの遅延があります．従って，ソフトウェア・パイプラインを組んで命令の並列実行を図る場合は，このことを考慮したコードを記述します．また，命令は同じ機能ユニットを一度に一つしか使えないので，命令がどの機能ユニットを使っているかを意識する必要があります．

うまくいけば大幅な最適化が望めても，工数との兼ね合いやバグを見逃したときのリスクなどを考えると，フル・アセンブリのコードを書くことをためらうかもしれません．

そこで，CCSを用いた開発環境では，リニア・アセンブリという言語が用意されています(**図3**)．このリニア・アセンブリを用いてコードを記述する場合，命令をプログラマ自身が並列化する必要はありません．アセンブリ・オプティマイザがコードの最適化を図り，適切な並列化を行います．上述した遅延の生じる命令を用いる場合，実行したい命令の順に記述すると，アセンブリ・オプティマイザが最適なコードに変換します．また，命令が使用する機能ユニットを記述しなくても，アセンブリ・オプティマイザが適切なユニットに割り当てます．レジスタに対して，A16やB4といったレジスタの名前をプログラマが記述する代わりに，「レジスタ変数」というものを宣言することで，アセンブリ・オプティマイザがこのレジスタ変数に適切なレジスタを割り当てます．これらについては，コード記述の解説のところで再度説明します．

昔のDSPは構造が非常にシンプルであったため，C64x+のような複雑なアーキテクチャと命令群を理解しないとコードが記述できないということはありませんでした．昔からDSPのソフトウェアを開発していて腕に覚えのある開発者であっても，C64x+を用いた開発に初めて臨む場合