第9章 アセンブリ・プログラミング

アセンブリ言語によるプログラム開発は，昔からDSPを使ったシステムを開発する上で頭の痛い点でした．初期のDSPは低速であったために，プログラム全体を最適化する必要がありました．また，アーキテクチャがプログラマよりではなかったことも，プログラミングを困難にしていました．

幸いなことにADSP-BF533は600MHzや750MHzといった高速で動作するため，はじめからカリカリに最適化しなければならない局面は少なくなっています．性能面の要求が厳しくなければ，C/C++言語で書いてしまえばよいのです．

しかし，ビデオ信号処理を行う場合やソフトウェアIPを作る場合などは，やはりきっちりと最適化したプログラムが要求されることもあります．そういった場合には，アセンブリ・プログラミングは避けて通れません．C/C++言語の最適化機能には限界がありますし，手作業で作るプログラムは，コンパイラが利用できないような特殊命令を活用することもできます．

Blackfinアーキテクチャは，プログラマ自身がアセンブリ言語を使ってアプリケーションを組むときに激しい苦痛を感じなくとも済むよう，随所に工夫が凝らしてあります．パイプラインのインターロックはその一つです．演算結果の依存性によるパイプライン・ハザードが起きて1サイクル待たなければならないようなとき，Blackfinコアは自動的にストール・サイクルを挿入します(**図9-1**)．これによって速度は低下するものの，演算結果は常にプログラムの字面どおりになります．タイミングに

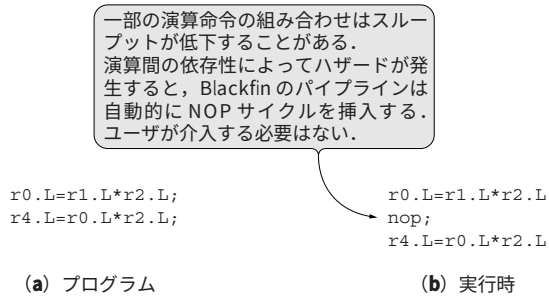


図9-1　インターロック

よる演算結果の間違いが発生しません．この機能のおかげで，プログラマは最適化を始めるまではタイミング問題を忘れてしまってもかまいません．また，アドレス空間が一つにまとめられているため，どの空間を使うべきか頭を悩ます必要もありません．

Blackfinアーキテクチャであれば，それほど苦労せずともアセンブラを使うことができます．この章では，アセンブリ言語を使う上での基本的な事柄について説明します．

9-1 行の構造

● アセンブリ言語の実行文は行単位で記述していく

アセンブリ言語の命令の例については，すでに第3章で紹介しました．Blackfinの命令は，数式風の形をしています[注1]．アセンブリ言語の実行文は，この命令を中心として行単位で記述していきます．行の構造を**図9-2**に示します．

行頭にはラベルを配置します．ラベルは英字か_(アンダースコア)で始まり，英数字，_(アンダースコア)，$(ダラー)，.(ピリオド)を含みます．ラベルは:(コロン)で終わりますが，コロンはラベルの区切りであってラベル自身には含まれません．

VisualDSP++開発環境では，C言語との互換性を維持するために，ラベルは大文字と小文字を区別して取り扱います．つまり，以下の二つのラベルは異なるラベルです．

```
LABEL1:
Label1:
```

関数の終わりには，関数名となるラベルに.endを付け足したラベルを配置します．これによって，リンカは関数の終わりを正しく認識できるようになり，不要コードの削除を正確に行えるようになります．また，第6章で説明した統計的プロファイリングも，この関数の終わりのラベルを利用します．関数の終わりにラベルを置かない場合，アセンブラが警告を出力します．

ラベルに続いてBlackfinの命令を置きます．命令は;(セミコロン)で終わりますが，セミコロン自身は区切り子なので，やはり命令には含まれません．ラベルと異なり，アセンブラは命令の大文字と小文字を区別しません．どちらを使うかはプログラマの好みです．筆者は多くの場合，レジスタ名とニモニックを小文字で書くようにしています．しかし，英文字の“l”と英数字の“1”がわかりにくいため，レジスタ・ハーフを表す部分だけは大文字で書くようにしています．たとえば，次のように書いています．

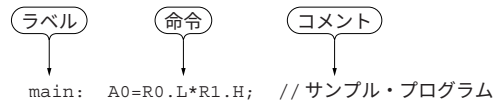


図9-2 行の構造

注1：むしろFORTRAN風といったほうがよいかもしれない．

```
r0.L = r1.H + r2.L (s);
```

命令は;(セミコロン)で区切りながら，1行にいくつでも記述することができます．しかし読みやすさを考えれば，1行1命令にしたほうがよいでしょう．

行末には，//(ダブル・スラッシュ)で始まるコメントを置くことができます．コメントは//(ダブル・スラッシュ)で始まり，改行で終わります．お気づきのように，このコメントはC++と同じです．VisualDSP++のアセンブラはC++コンパイラとプリプロセッサを共用しているため，コメントをはじめとして#defineや#includeといったプリプロセッサ命令を利用できます．このため，/* */形式のコメントも使用できます．

● 注意：コメントは//だけにする

ところでプリプロセッサは1バイト文字にしか対応していないため，日本語のコメントを使う場合には予期せぬ場所でコメントが終了する危険性があります．それを避けるため，コメントは//(ダブル・スラッシュ)だけにすることをおすすめします．さらに2バイト目が¥で終わる文字[注2]を行末で使用すると，知らない間にコメントが次の行に持ち越されてしまうこともあり得ます．そこで，行末を半角文字などで終わらせることをおすすめします．

以上をまとめると，プログラムの各行は次のようになります．

```
_main:                          // ラベルの後ろは空でもOK
          r0 = r1 + r2;
          r3 = r0 >> 2;         // コメントは読み手のためにある．
_main.end:                  // 手続きの終わりを表すラベル．
```

9-2　セクション宣言

アセンブリ・プログラムは，実行文のほかにメモリ領域の宣言部や，大域ラベルの宣言を行う部分をもちます．この節では，それらの補助的な宣言について説明します．

Blackfinプロセッサはメモリ空間を一つしかもたないので，命令もデータも同じ論理空間に配置されます．しかし，ADSP-BF533のL1メモリは，命令専用とデータ専用に分かれています．L1命令メモリの内容をデータとして取り扱うことはできず，同様にL1データ・メモリの内容を命令として実行することもできません．また，データや命令はSDRAMに配置するかL1メモリに配置するかで大幅に性能が変わるので，場合によっては細かい調整が必要になります．

メモリへの配置を調整するために.section命令が用意されています．

注2：たとえば「能」という文字は，2バイト目が1バイト文字の \ と同じ値である．

```
.section セクション名;
```

セクション名は任意の識別子で，これと同じものを第11章で説明するLDF（Linker Description File）に記述します．.section命令より下に現れる変数や命令は，セクション名で表される一つの論理的なメモリ領域に配置されます．この論理的なメモリ領域は，次に.sectionが現れるまで続きます．セクション名をつけた領域が具体的に何番地に配置されるかは，LDFの中で記述します．つまりプログラマは，コードの配置の最終決定をリンクを行うまで気にしなくてもかまいません．

最初のうちはセクション名としてprogramとdata1を使うとよいでしょう．この二つはコンパイラが生成するコードに使用されているため，VisualDSP++に用意されているデフォルトのLDFが対応しています．そのため，手始めに使ってみるときにLDFを編集しなくてもすむのが利点です．名前からわかるようにprogramは命令用の領域で，data1は大域変数用の領域です．

9-3 変数宣言

● .byte命令

変数宣言は，.byte命令を使います．

```
.byte 変数名;
.byte2 変数名;
.byte4 変数名;
```

.byte命令は，1バイトの変数を宣言します．変数名は一つだけではなく複数の変数名をコンマで区切って並べることもできます．同様に.byte2，.byte4命令は2バイト，4バイトの変数を宣言します．ただし，これらの命令は変数のアドレスを適切なワード境界に整列してくれません．たとえば，.byte4命令で宣言された4バイト変数が奇数アドレスに配置されることもあり得ます．こういった配置は問題で，奇数アドレスに配置された4バイト変数をアクセスすると，Blackfinプロセッサは例外イベントを引き起こします．これを防ぐために，.align命令で整列を指定します．

● .align命令

.align命令は，続く変数の配置をすべて指定した値に整列します．

```
.align 4;
.byte4 alpha;
.byte beta;

.align 1;
.byte gamma;
```

上の例では変数alphaとbetaが4バイト境界に整列され，変数gammaだけが1バイト境界に整列されます．

● 配列宣言

配列変数も同様に宣言することができます．

```
.align 4;
.byte4 array1[3];
.byte4 array2[3] = {1,2,3};
.byte4 array3[3] = "ファイル名";
```

上の例は，いずれも要素数3の配列を宣言します．ただしarray1は初期化されませんが，array2とarray3は初期化されます．array2の初期化はC言語と同様な静的な式を使うもので，大括弧の中に要素ごとの初期値を区切って並べます．array3の初期化は，ファイルを使います．ファイルはテキスト・ファイルで，初期値を1行に一つずつ並べて記述します．ファイルを使った初期化は，テスト・データを与えるときなどに便利に使えます．

9-4　大域ラベル

アセンブリ言語のソース・ファイルで宣言したラベルや変数を他のソース・ファイルから利用したい場合や，逆にほかのソース・ファイルで宣言したラベルや変数を利用したい場合には，特別な宣言が必要です．

ラベルや変数を外部で利用できるようにするときには，.global命令を利用します．

```
.global _main;
```

この例は，ソース・ファイルで宣言した_mainというラベルをほかのファイルからも利用できるように.global命令を使って公開しています．ラベル名は，コンマで区切って並べることもできます．

逆に外部で宣言されたラベルを読み込むには，.extern命令を使います．

```
.extern _exit;
```

この例は，外部のソース・ファイルで宣言された_exitというラベルをこのファイルで利用できるように読み込んでいます．ラベル名は，コンマで区切って並べることもできます．

9-5　試し斬り

さて，ひととおりアセンブリ・プログラムの構成要素を説明したので，簡単なプログラムを組んでみましょう．

```
.section data1;                          // データ領域に配置
.align 4;                                // 32ビット境界に整列
.byte4 alpha=2,  beta=3, gamma;

.section program;                  // プログラム領域に配置
.global _main;                           // _mainラベルを外部に公開
_main:
    p0.H = hi(alpha);                    // 変数alphaのアドレスを取得
    p0.L = lo(alpha);
    r0 = [p0];                           // alphaの値を取得
    p1.H = hi(beta);                     // 変数betaのアドレスを取得
    p1.L = lo(beta);
    r1 = [p1];                           // betaの値を取得
    r2 = r1 + r0;
    p0.H = hi(gamma);                    // 変数gammaのアドレスを取得
    p0.L = lo(gamma);
    [p0] = r2;                           // alpha + beta をgammnaに格納
    rts;
_main.end:
```

このプログラムは，C言語のmain()関数としてふるまいます．つまり，システムが立ち上がったときに呼ばれるアプリケーション本体です．リセット直後に必要な初期化を行い，実行環境を整える仕事はすべてシステム側のランタイム・ライブラリが行うので，ユーザが気にする必要はありません．プログラムの動作としては変数alpha，betaの値を足し合わせ，和を変数gammaに格納するという簡単なものです．

最初にdata1セクションで32ビット変数を三つ宣言します．このうち，alphaとbetaには初期値を与えます．初期値を与える場合でも，例のようにそれぞれの宣言をコンマで区切って並べることができます．

次のprogramセクションは，プログラム用のセクションです．ここでは，main()関数に相当するラベル，_mainを宣言して，それを外部に公開しています．公開したラベルはシステムのランタイム・コードとリンクされ，アプリケーションの実行開始点となります．なお，C言語同様にメイン・プログラムは関数だと考えられます．アセンブリ言語で関数を書く場合には，関数の終わりに「**関数名**.end」というラベルを置かなければなりません．この例では_main.endというラベルを最後に配置します．

アプリケーション本体で行っていることは単純です．二つの変数をアクセスして，その値の和を変数gammaに格納するだけです．変数は大域変数なので，絶対アドレスを求めてアクセスします．アド

レスは32ビットであり，即値命令を二つ使ってロードします．これは3-3節で説明しました．

関数の末尾には，rts命令を置いて呼び出し元であるシステムのランタイムに戻ります．

このプログラムを5.2節で説明したとおりに登録してEZ-KIT Liteで走らせてみましょう．第5章ではメイン・プログラムをmain.cppに格納してプロジェクトに登録していましたが，今回はファイル名をmain.asmとして保存し，プロジェクトに登録します．ビルドしてロードすると，アセンブリ言語のプログラムであっても_mainでいったん停止します．これは，VisualDSP++が_mainというラベルで停止するようにデフォルトで設定されているためです．

9-6　デュアル演算命令を使う

一度「試し斬り」を済ませてしまえば，あとはどんな命令でも自由に試すことができます．EZ-KIT Liteを使えば，レジスタの内容を見ながらステップ実行もできますから，動作がよくわからないような命令であってもその場で試してみることができます．汎用マイコンの命令と異なり，DSPの命令は動作が直感的でないものがあります．そこで行き詰ったときには，命令セット・リファレンスをにらむばかりではなくEZ-KIT Liteやシミュレータを使って実験することをおすすめします．

● デュアルALUを使った16ビット加算命令の試験プログラム

試しに上で作ったプログラムを，下のように書き換えてみてください．

```
.section program;          // プログラム領域に配置
.global _main;                    // _mainラベルを外部に公開
_main:
    r1.H = 1;
    r1.L = 2;
    r2.H = 3;
    r2.L = 4;
    r0 = r1 +|+ r2;               // デュアル加算
    rts;
_main.end:
```

このプログラムは，デュアルALUを使った16ビット加算命令の試験プログラムです．単純なプログラムですが，EZ-KIT Liteにロードし，VisualDSP++で動作を追いかけると，レジスタがどのように変化するかよくわかります．ステップ実行時には，レジスタをいくつか表示します．表示は，VisualDSP++のメニュー・バーからRegsiter ⇒ Core ⇒ Data Register Fileを選ぶことでデータ・レジスタを表示できるほか，Register ⇒ Core ⇒ Status ⇒ Arithmetic StatusでASTATを表示できます．

ステップ実行中にレジスタの値を手作業で変更することもできます．**図9-3**にそのようすを示します．レジスタ・ウィンドウのレジスタの値を右クリックし，コンテキスト・メニューからEditを選び

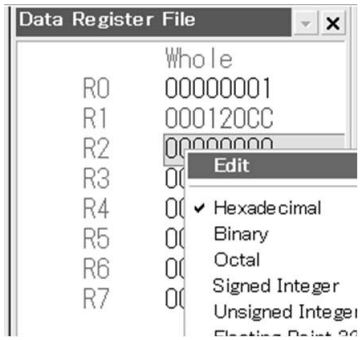


図9-3　レジスタの編集

ます．そうすると，レジスタの値が編集可能な状態になるので，それを編集して終わりです．この編集方法はレジスタの値のほか，メモリ上の変数の変更にも使えます．また，ディスアセンブル・ウィンドウを表示しているときには，インライン・アセンブラ機能を使って直接命令を書き込むこともできます．

9-7　C言語から呼び出す

信号処理をアセンブリ言語で書くとしても，割り込みハンドラやペリフェラルの初期化といった「雑事」までアセンブリ言語で書く必要はありません．よほどの理由がないかぎり，こういった仕事は高級言語で書くべきです．信号処理アプリケーションの演算量はごく狭い範囲に集中しているので，それ以外のところをアセンブリ言語で書いてもほとんどといってよいほど性能には寄与しませんし，何より大幅に時間がかかってしまいます．そういった性能にあまり関係ない部分は，高級言語で手っ取り早く書いたほうが楽です(**図9-4**)．

● コードの再利用性を飛躍的に向上させる方法

アセンブリ言語で信号処理プログラムを書くときに高級言語で呼ぶことができるようにしておくと，コードの再利用性が飛躍的に向上します．アセンブリ言語で書いたルーチンは変数の受け渡し方法に柔軟性がありすぎるため，自分で書いたものであっても，よく文書を読み直さないと使い方がわからないといったことがあります．たとえ今その必要がなくても，おもな機能モジュールはC言語から呼べるようにしておけば，後々あわてずにすむでしょう．

アセンブリ言語でC言語から呼び出せるルーチンを書くには，名前の付け方や引き数の受け渡し方法とレジスタの使い方を把握しておく必要があります．つまり，

- 変数名，関数名の規則
- スタック・フレームの確立と廃棄
- 引き数の受け取り方

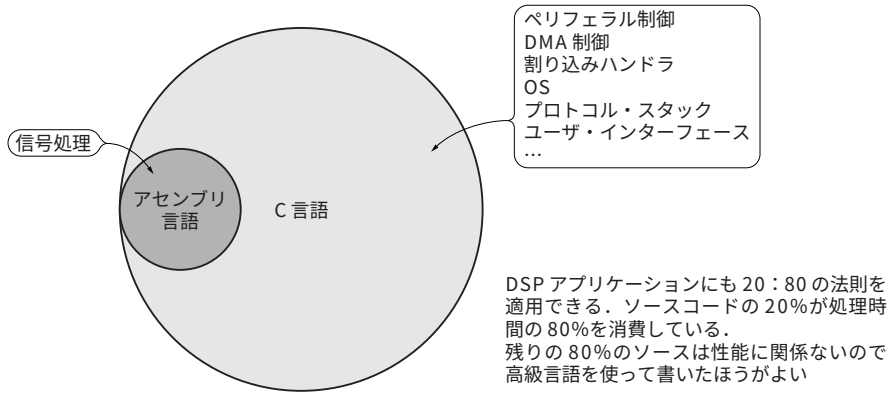


図9-4　雑用は高級言語で

- 戻り値の返し方
- 使ってはいけないレジスタ
- 元に戻さなければならないレジスタ

を，あらかじめ知っておかなければなりません．これらの情報はVisualDSP++のコンパイラ・マニュアルに詳細に書いてありますが，本書でも概要を説明しておきます．

● 変数名，関数名の規則

C言語で使用する変数名や関数名は，アセンブリ言語とリンクする際に異なる名前に変換されます．たとえば次のプログラムを見てみましょう．

```
short a;          // 大域変数 _a
int main(void)    // 関数 _main
{
}
```

大域変数aと関数mainが宣言されています．この両者は，アセンブリ言語とリンクするときに頭に“_”(アンダースコア)をつけた名前に変換されます．つまり，_aおよび_mainという名前になります．逆にいえば，C言語から利用できる変数や関数をアセンブリ言語で記述する場合，それらの名前は“_”で始まらなければなりません．

C++言語の場合は，事情が異なります．C++言語は関数の多重定義を許すため，関数名に引き数の型を組み合わせて別の名前に変換する「ネーム・マングリング」と呼ばれる処理がコンパイラによって行われます．この変換をアセンブリ言語を使うプログラマが追うのはたいへんなので，C++言語には宣言によってネーム・マングリングを抑止する機能が付いています．

```
extern "C" {
    void func1( int a, int b );
    void func2( int a, int b );
}
```

extern "C"によって，コンパイラは指定された関数がC言語のライブラリとリンクされると判断します．したがって，アセンブリ・プログラム側ではC言語から呼び出されることだけを考えて開発をすればよいということになります．

上のプロトタイプ宣言はC++言語用なので，Cコンパイラにかけるとエラーが発生します．そこで，次のように条件コンパイルを行うことで，アセンブリ関数のプロトタイプ宣言をC/C++言語両対応にすることができます．

```
#ifdef __cplusplus
extern "C" {
#endif
    void func1( int a, int b );
    void func2( int a, int b );
#ifdef __cplusplus
}
#endif
```

● スタック・フレーム

スタック・フレームはC/C++をはじめとする高級言語でよく用いられるデータ構造で，スタック上の自動変数を小さなオーバヘッドで実装できるのが特徴です．アセンブリ言語では自動変数を使うことが少ないので，必ずしもスタック・フレームが必須というわけではありません．しかし，空でもスタック・フレームを作っておけば引き数の受け取りが容易になるため，積極的に利用することをおすすめします．

スタック・フレームの確立は簡単です．関数の先頭で，次の一文を実行するだけです．

```
link frame_size;                    //フレームを確保
```

この命令は，次の動作を続けて行います(**図9-5**)．

- RETSレジスタの値をスタックにプッシュする．
- FPレジスタの値をスタックにプッシュする．
- FPレジスタにSPレジスタの値をロードする．
- SPレジスタの値からframe_sizeを引く．

frame_sizeは関数内で自由に使える領域で，一時変数として使用します．関数内ではFPが古いFP

のアドレスを指し示しており，そこを基点として引き数にもフレーム内部の変数にも容易にアクセスできます．

スタック・フレームの廃棄はもっと簡単で，パラメータは必要ありません．

```
unlink;                                    //フレームを廃棄
rts;                                       //サブルーチンから戻る
```

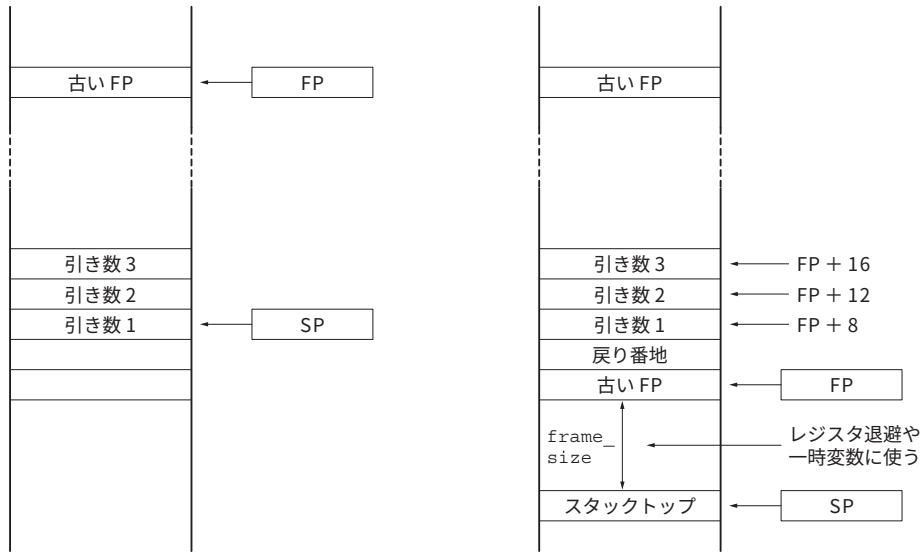
一般には，スタック・フレームの廃棄と関数からの戻りは同時に行うので，上のように熟語として覚えておくといいでしょう．

● 引き数渡し

さて次は，引き数の受け取りです．VisualDSP++のコンパイラはほかのCコンパイラと同様に，関数の引き数を引き数リストの後ろから順にスタックに積みます．この結果，呼ばれた側から見ると，関数の第1引き数がFPにいちばん近いところに陣取ることになります(**図9-5**)．ただしこれには例外があります．

- 引き数の最初の3ワードは，レジスタによって渡される．
- 変数の数によらず，また，引き数がレジスタから渡される場合でも，スタック上には引き数領域として最低3ワード(12バイト)が確保される．

これらのようすを，**図9-6**と**図9-7**に示します．レジスタによる引き数渡しはオーバヘッド削減を目的としたものです．引き数のプッシュと取り出しの手間を省ける分，性能の向上を見込めます．



(**a**) `link frame_size;` 実行前　　(**b**) `link frame_size;` 実行後

図9-5　link命令