

SH マイコン C プログラミング虎の巻

荒川靖弘

1998 年 9 月 3 日

目次

1	はじめに	2
2	まずは基本中の基本	2
2.1	コーディングスタイル	2
2.2	ベクタテーブルは C 言語で書くべきか	2
3	パイプライン基礎講座	2
3.1	パイプラインについて	2
3.2	パイプラインが乱れる原因	3
3.3	パイプラインを乱さないようにするには	4
4	コンパイル時の最適化は必要か	5
4.1	SHC コンパイラの最適化機能	5
4.2	コーリングシーケンス	5
4.3	最適化の例 1	6
4.4	最適化の例 2	9
4.5	最適化の例 3	12
4.6	最適化で気をつけること	16
5	最後に (コーディングテクニッカー覧)	20

1 はじめに

本レポートは、機器組み込み用シングルチップ RISC マイコン日立 SuperH RISC engine ファミリ（以降 SH マイコンと略称します）のプログラミングを行う際のコーディングテクニック等について紹介します。マイコンプログラムに慣れている方には「釈迦に説法」でしょうが、筆者なりに SH マイコン用のプログラムを作成する上での注意点をまとめてみました。

今回使用したコンパイラは (株) 日立マイコンシステム製の以下のコンパイラです。

コンパイラ： SH SERIES C Compiler Ver. 4.1B

アセンブラ： SH SERIES ASSEMBLER Ver. 3.1A

リンカ： H SERIES LINKAGE EDITOR Ver. 5.3B

2 まずは基本中の基本

2.1 コーディングスタイル

今回使用したコンパイラは、最適化についてはかなり強力な機能を持っているようです。しかし、その分ソースコードの構文解釈がデリケートになっています。

最適化については後述しますが、ひとまずコーディングスタイルについては ANSI C の標準的なスタイルを守るようにする必要があります。特にプロトタイピングを行わないソースは最適化機能に大きな支障をきたす場合があります。

算術式や論理式には少し注意してください。現在のバージョンではほとんど直っているようですが、SHC コンパイラにはこのあたりにまだバグが隠れている場合があります。怪しいと思ったら、式を分離するか括弧を付けて優先順位を明示することによって改善することがあります。

2.2 ベクタテーブルは C 言語で書くべきか

実は SHC コンパイラでは C でベクタテーブルを比較的簡単に記述することができます。H シリーズ、SH シリーズのマイコンはプログラムで定義しない場合の「デフォルトの動作」が厳密に決まっていて、ハードウェアマニュアル等にも明記してあるので、割と手拔きのコーディングでも許されてしまいます。

しかしブートストラップルーチンなどはアセンブラでしか書けない部分というものもあるので（特に初期化部分）、C 言語で記述できそうなプログラムでも敢えてアセンブラソースの部分を残しておくというのは意味ある事のように思えます。

3 パイプライン基礎講座

3.1 パイプラインについて

この章では C のコーディングの話からはずれません。しかし SH マイコンの「癖」について割と踏み込んで述べているつもりですので是非読んでみてください。また文献 [1] などでも詳しく紹介されていますので、こちらもぜひ参考にしてください。

SH マイコンは命令の実行に「パイプライン」と呼ばれる機能を使っています。パイプラインとは、簡単にいえば複数の命令を同時に実行することにより 1 命令あたりの実行期間を見かけ上短くする機能です。

通常 1 つの命令を実行する場合、以下の 5 つの状態が発生します。

- 命令フェッチ (IF) : メモリから命令を取り込みます。
- 命令デコード (ID) : 取り込んだ命令を解読します。
- 命令実行 (EX) : 解読結果に従い、データ演算やアドレス計算を行います。
- メモリアクセス (MA) : メモリのデータアクセスを行います。(メモリアクセスを伴う命令のみで発生します。)
- ライトバック (WB) : メモリアクセスした結果(データ)をレジスタに戻します。(メモリアクセスを伴う命令のみで発生します。)

それぞれの状態をここでは「ステージ」と呼びます。各ステージは命令の実行とともに以下のように流れていき、パイプラインを構成します。

	↔	↔	↔	↔	↔	↔	↔	↔	↔	↔	: スロット
命令 1	IF	ID	EX	MA	WB						
命令 2		IF	ID	EX	MA	WB					
命令 3			IF	ID	EX	MA	WB				
命令 4				IF	ID	EX	MA	WB			
命令 5					IF	ID	EX	MA	WB		
命令 6						IF	ID	EX	MA	WB	

ある瞬間では最大 5 つの命令が実行されていることがおわかりでしょうか。MA, WB ステージは命令によつては発生しないこともあります。

ある 1 つのステージが実行される期間を「スロット」と呼びます。命令の各ステージは必ず 1 スロットで実行されます。通常 1 スロット内で 2 つ以上のステージを実行することはありません。ただし、WB ステージは MA の直後に実行されますので、状況によっては MA ステージと WB ステージが同一スロットで実行される場合もあります。

H/W 条件やステージの競合などによってパイプラインが乱れることもあります。次節でパイプラインが乱れる原因について説明します。

3.2 パイプラインが乱れる原因

パイプラインが乱れる原因としては以下のものが考えられます。

1. ステージの実行にかかるステート数(システムクロックサイクル数)が 1 つのスロット内で同じでない場合、パイプラインの流れは以下ようになります。

命令 1	IF	IF	ID	-	EX	MA	MA	MA	WB	
命令 2			IF	IF	ID	EX	-	-	MA	WB

これは、IF に 2 ステート、MA に 3 ステートかかった場合の様子です。“-” はストールしている状態を表しています。ID や EX, WB ステージは常に 1 ステートで実行されますが、メモリアクセスをともな

う IF や MA ステージの場合、余分に実行時間がかかる場合があります。

2. IF ステージと MA ステージが同一スロットで操作しようとする、同時にメモリアクセスを行うことはできないので、以下のように競合状態が発生します。

命令 1	IF	ID	EX	MA	WB				
命令 2		IF	ID	–	EX	MA	WB		
命令 3			IF	–	ID	–	EX		
命令 4					IF	–	ID	EX	
命令 5							IF	ID	EX

この例では、命令 1 と命令 4、命令 2 と命令 5 が競合しています。この場合競合が発生したスロットはスプリットし、MA ステージが優先的に実行されその他のステージはその後に実行されます。(WB ステージは MA ステージの直後に実行されるので、MA, WB ステージは同一スロットで実行されます。)ただし、命令が内蔵 ROM/RAM または内臓キャッシュに配置されている場合は更に異なる動作をします。これらのメモリに命令が配置されている場合は、1 回の命令フェッチで 2 命令持ってくるできるので、次命令の IF ステージではバスサイクルが発生しないことになります。この場合は下に示すようにスプリットは発生しません。(命令 1 と命令 4 は競合しない)

命令 1	IF	ID	EX	MA	WB				
命令 2		if	ID	EX	MA	WB			
命令 3			IF	ID	–	EX			
命令 4				if	–	ID	EX		
命令 5						IF	ID	EX	

3. メモリロード命令の直後にディスティネーションレジスタを使う命令を実行しようとする以下のようにスプリットが発生します。

命令 1 (MOV.W @R0, R1)	IF	ID	EX	MA	WB		
命令 2 (ADD R1, R2)		IF	ID	–	EX		

これは命令 1 の WB ステージが来る前に命令 2 の EX ステージを実行しようとしたために発生します。

3.3 パイプラインを乱さないようにするには

パイプラインをなるべく乱さないようにするためには、コーディングの際に以下の点に気をつけます。

1. 命令を配置するメモリエリアをなるべく内蔵 ROM/RAM にします。または内臓キャッシュを活用してメモリアクセスの効率を上げます。
2. MA を持つ命令（ロード命令など）をできるだけロングワード境界にのみ配置するようにします。
3. メモリからのロード命令の直後の命令には、ロード命令のディスティネーションレジスタと同じレジスタを使わない命令を配置します。
4. 乗算器を使う命令が連続しないように配置します。乗算器からの結果の取り出しのための MACH, MACL レジスタへのアクセスも乗算器を使う命令に連続しないように配置します。

しかし、実際問題 C 言語レベルでこれらの調整を行うのはほとんど不可能です。そこで、SHC コンパイラは上記の調整を自動的に行うオプションを備えています。

4 コンパイル時の最適化は必要か

4.1 SHC コンパイラの最適化機能

SHC コンパイラの最適化機能はかなり強力です。最適化の主な機能について以下に挙げてみます。

- Auto 変数のレジスタへの自動/最適割り付け
- 演算の強度軽減
- パイプライン最適化
- 定数の畳み込み
- 文字列の共有化
- 共通式/ループ不変式の削除
- 不要文の削除
- テールリカーション最適化

この中でも「パイプライン最適化」は特に強力な機能を持っていて、3.3 章に示したプログラミング技法に基づいてほぼ自動的に「コードの並び替え」を行います。

次節から簡単な C コーディングを例にして、「コードの並び替え」がどのように行われているか見てみることにします。

4.2 コーリングシーケンス

まず実際のコードを見る前に、関数のコーリングシーケンスについて説明しておきます。

レジスタのアサイン 関数実行中の R0 から R15 までの各レジスタは以下のようにアサインされています。

R0～R3： ワーク用

R4～R7： 引数用 (引数が 4 つ以内の場合)

R8～R14： Auto 変数用

R15： フレームポインタ

このうちワーク用および引数用のレジスタは、関数の呼び出し前後で値の保証がされないの、関数を呼び出す場合はこれらのレジスタのうち使っているレジスタの値をあらかじめ退避しておかなければなりません。

Auto 変数用のレジスタは、「レジスタ変数」として明示して定義された変数についてこれらのレジスタを割り当てます。

引数と返り値 引数は、上述したように、4 つ以下ならレジスタに割り当てられます。引数が 4 つ以上、あるいは引数が可変の場合はスタックに積まれます。引数がレジスタに入らない場合（浮動小数点値、構造体テーブル）もスタックに積まれます。

返り値は基本的に R0 レジスタにセットされます。返り値がレジスタに入らない場合は返り値の情報はメモリに展開され、R0 レジスタにはそのアドレスがセットされます。

```
int sum(int *tbl, int ct)
{
    register int    sum;

    sum = 0;
    for(; ct>0; ct--, tbl++) {
        sum += (*tbl);
    }
    return(sum);
}
```

図1 sum1.c

4.3 最適化の例 1

図1は、よく見るSUM値計算の関数です。

この一見何のひねりもないソースを最適化なしでアセンブラソースにコンパイルすると、図2のようになります。

図2のソースの内容について細かく説明はしませんが、Cのコードに忠実にアセンブラコードが展開されているのが分かります。また引数をわざわざスタックに積んで、スタックに対し引数のアクセスを行っています。「引数へのポインタへのアクセス」というのもあり得るので、機械的にこうしているだけなのかもしれませんが、この辺はちょっと納得いかない作りではあります。）

図1を最適化オプションをつけてコンパイルしてみましょう。結果を図3に示します。コード量が激減していることに驚かれるかもしれません。今回は以下に示す特徴的な最適化を行っています。

1. 引数の操作をレジスタで直接行っています。
2. レジスタ変数がR6に割り当てられています。レジスタ変数は通常R8からR14に割り当てられているのですが、値を保証しないワーク用のレジスタが余っているため、レジスタのアサインを自動的に変更したものと思われます。
3. 遅延分岐を有効的に利用しています。図2でジャンプ命令の直後にやたらとNOPが挿入されているのに気づかれたでしょうか。遅延分岐命令は高速に動作する半面、直後の命令を同時に実行してしまう副作用があります。(3.3章では述べませんでしたが、命令分岐時のこのような挙動は、SHマイコンに限らずRISCマイコンに一般的に見られる特徴です。) 図3では、NOP命令を挿入することによるコードの冗長性を命令の並び替えで回避しています。

```

        .EXPORT      _sum
        .SECTION     P, CODE, ALIGN=4

_sum:                                ; function: sum
                                      ; frame size=12

        MOV.L        R14, @-R15
        ADD          #-8, R15
        MOV.L        R4, @ (4, R15)
        MOV.L        R5, @R15
        MOV          #0, R14
        BRA          L219
        NOP

L220:
        MOV.L        @ (4, R15), R2
        MOV.L        @R2, R3
        ADD          R3, R14
        MOV.L        @R15, R2
        ADD          #-1, R2
        MOV.L        R2, @R15
        MOV.L        @ (4, R15), R3
        ADD          #4, R3
        MOV.L        R3, @ (4, R15)

L219:
        MOV.L        @R15, R2
        CMP/PL       R2
        BT           L220
        MOV          R14, R0
        ADD          #8, R15
        MOV.L        @R15+, R14
        RTS
        NOP

```

図2 sum1.src (sum1.c のコンパイル結果：最適化なし)

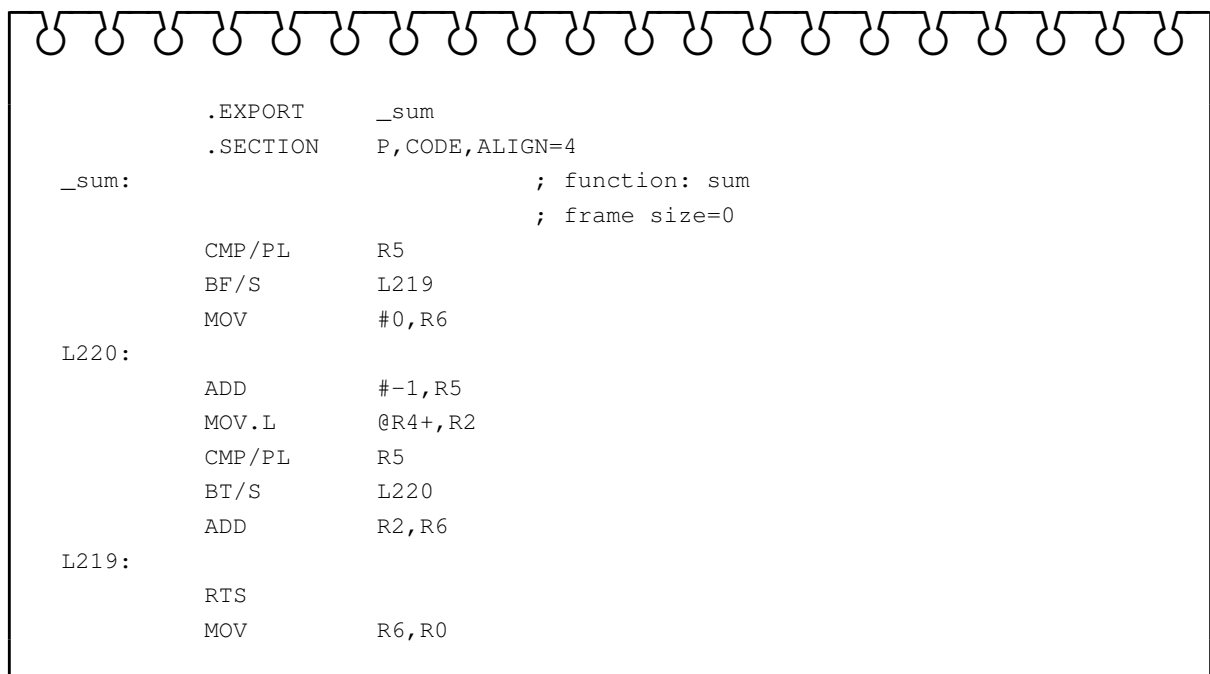


図3 sum1.src (sum1.c のコンパイル結果：最適化あり)


```

int sum(int *tbl, int ct)
{
    register int    *p, i;
    register int    a, b, sum;

    p = tbl;
    sum = 0;
    a = (*p);
    for(i=(ct-1), p++; i>0; i--, p++) {
        b = (*p);
        sum += a;
        a = b;
    }
    sum += a;
    return(sum);
}

```

図4 sum2.c

4.4 最適化の例 2

図1を図4のように少し変形してみます。

図4は「ソフトウェアパイプライン」と呼ばれる技法で、ロード命令の直後にディスティネーションレジスタを使わないように意図的に並べ替えを行っています。また、4.3章で分かったように、最適化を行わない状態では、引数をスタックに展開してしまうので、これらの変数も強引にレジスタ変数として再定義してみました。

このソースに対し最適化を行わずにコンパイルした結果が図5です。そして、最適化を行った場合を図6に示します。

図5は図2に比べてループ内の処理がかなりすっきりしました。その分前後の処理が肥大化しています。アクセスするテーブルのサイズによっては、かなり非効率なコードになりそうです。

図6を見てみます。一見「ソフトウェアパイプライン」というこちらの意図は達成されているようにみえます。しかし図3をよく見てみると実はこのコードでも遅延スロットは発生しないということがわかんと思います。

デバッグ時はあまり最適化をしないものなので、この技法はある程度有効かもしれませんが、コンパイラの性能を考えればこういったトリッキーなコーディングは「大きなお世話」なのかもしれません。

```

        .EXPORT      _sum
        .SECTION     P, CODE, ALIGN=4
_sum:                                ; function: sum
                                    ; frame size=28

        MOV.L        R14, @-R15
        MOV.L        R13, @-R15
        MOV.L        R12, @-R15
        MOV.L        R11, @-R15
        MOV.L        R10, @-R15
        ADD          #-8, R15
        MOV.L        R4, @ (4, R15)
        MOV.L        R5, @R15
        MOV.L        @ (4, R15), R14
        MOV          #0, R10
        MOV.L        @R14, R12
        MOV.L        @R15, R13
        ADD          #-1, R13
        ADD          #4, R14
        BRA          L223
        NOP

L224:
        MOV.L        @R14, R11
        ADD          R12, R10
        MOV          R11, R12
        ADD          #-1, R13
        ADD          #4, R14

L223:
        CMP/PL       R13
        BT           L224
        ADD          R12, R10
        MOV          R10, R0
        BRA          L225
        NOP

L225:
        ADD          #8, R15
        MOV.L        @R15+, R10
        MOV.L        @R15+, R11
        MOV.L        @R15+, R12
        MOV.L        @R15+, R13
        MOV.L        @R15+, R14
        RTS
        NOP

```

図5 sum2.src (sum2.c のコンパイル結果：最適化なし)

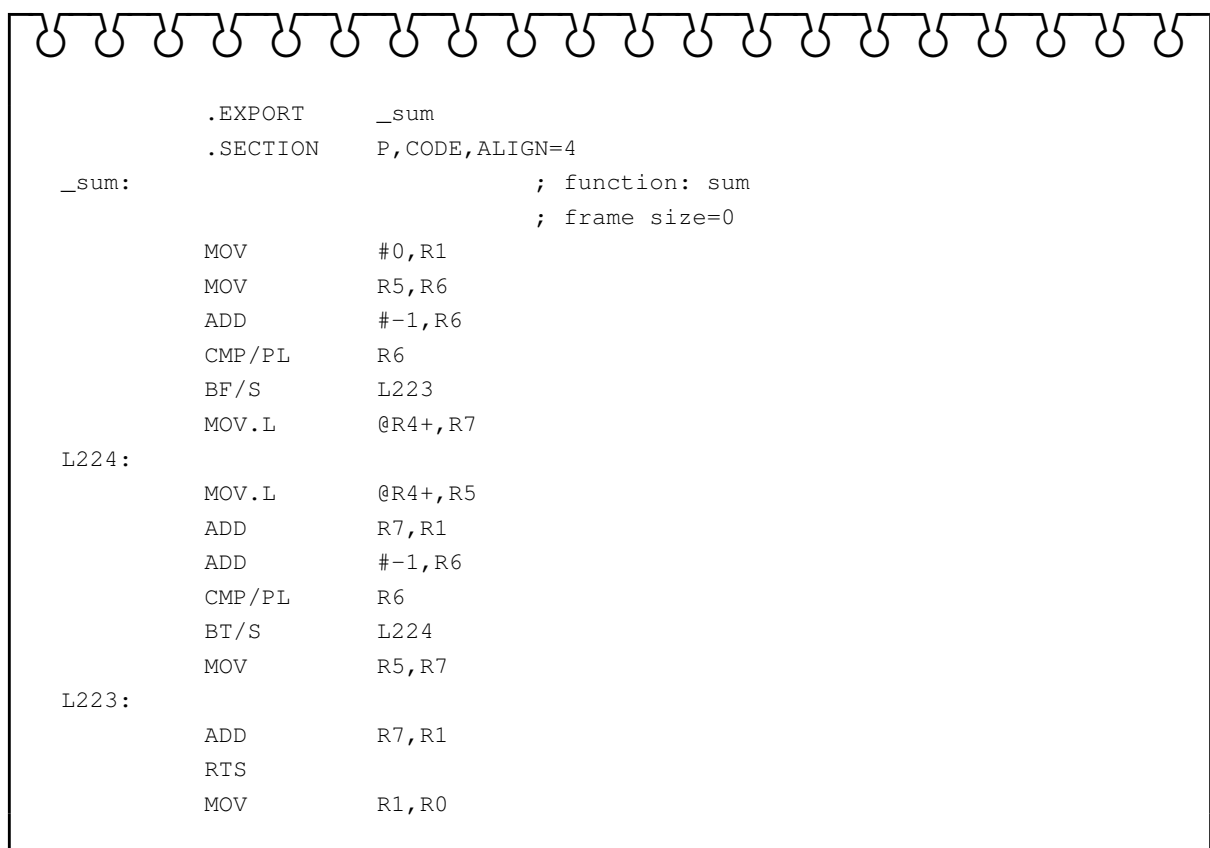


図 6 sum2.src (sum2.c のコンパイル結果：最適化あり)

```

int sum(int *tbl, int ct)
{
    int sum, i;

    sum = 0;
    for(i=ct>>2; i>0; i--, tbl+=4) {
        sum += tbl[0];
        sum += tbl[1];
        sum += tbl[2];
        sum += tbl[3];
    }
    for(i=ct&3; i>0; i--, tbl++) {
        sum += (*tbl);
    }
    return(sum);
}

```

図7 sum3.c

4.5 最適化の例 3

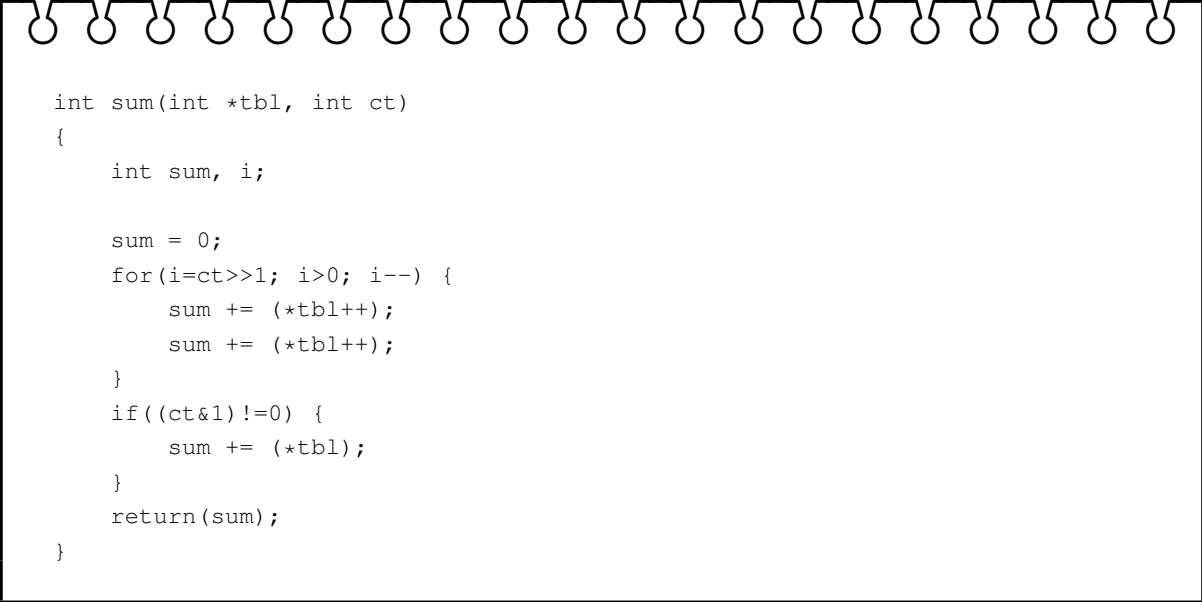
図1を違う形で変形してみます。この図7のコードは一見冗長にみえますが、RISC マイコンは分岐処理があると実行効率が低下してしまうため、ループ回数を減らして処理全体を高速化できるようにしています。

図7を最適化してコンパイルしたのが図8です。図3に比べるとコード量は増加していますが、ループ回数が少ないため、テーブルが充分大きければ全体として高速な動作が期待できます。

ところで、図7のコードは、さらに図9のように変形させることができます。このコードをコンパイルしたものが図10です。本当はループ回数を1/4にしたかった（「4でアンロールする」といいます）のですが、スロットのスプリットがひんばんに発生するため実際にはあまり効率が良くないようです。（どういうコードになるかは是非試してみてください。これも何だか納得がいかないような気がしますが、きっと何か意味があるのでしょう。）図7とどちらがいいかは微妙なところです。要素数によって使い分けるのもいいかもしれません。

	.EXPORT	_sum	
	.SECTION	P, CODE, ALIGN=4	
_sum:			; function: sum
			; frame size=0
	MOV	R5, R7	
	SHAR	R7	
	SHAR	R7	
	CMP/PL	R7	
	BF/S	L221	
	MOV	#0, R6	
L222:			
	MOV.L	@R4, R2	
	ADD	#-1, R7	
	MOV.L	@(4, R4), R3	
	ADD	R2, R6	
	MOV.L	@(8, R4), R2	
	CMP/PL	R7	
	ADD	R3, R6	
	ADD	R2, R6	
	MOV.L	@(12, R4), R3	
	ADD	R3, R6	
	BT/S	L222	
	ADD	#16, R4	
L221:			
	MOV	#3, R7	
	AND	R5, R7	
	CMP/PL	R7	
	BF	L223	
L224:			
	MOV.L	@R4+, R2	
	ADD	#-1, R7	
	CMP/PL	R7	
	BT/S	L224	
	ADD	R2, R6	
L223:			
	RTS		
	MOV	R6, R0	

図 8 sum3.src (sum3.c のコンパイル結果：最適化あり)



```
int sum(int *tbl, int ct)
{
    int sum, i;

    sum = 0;
    for(i=ct>>1; i>0; i--) {
        sum += (*tbl++);
        sum += (*tbl++);
    }
    if((ct&1)!=0) {
        sum += (*tbl);
    }
    return(sum);
}
```

图9 sum3b.c

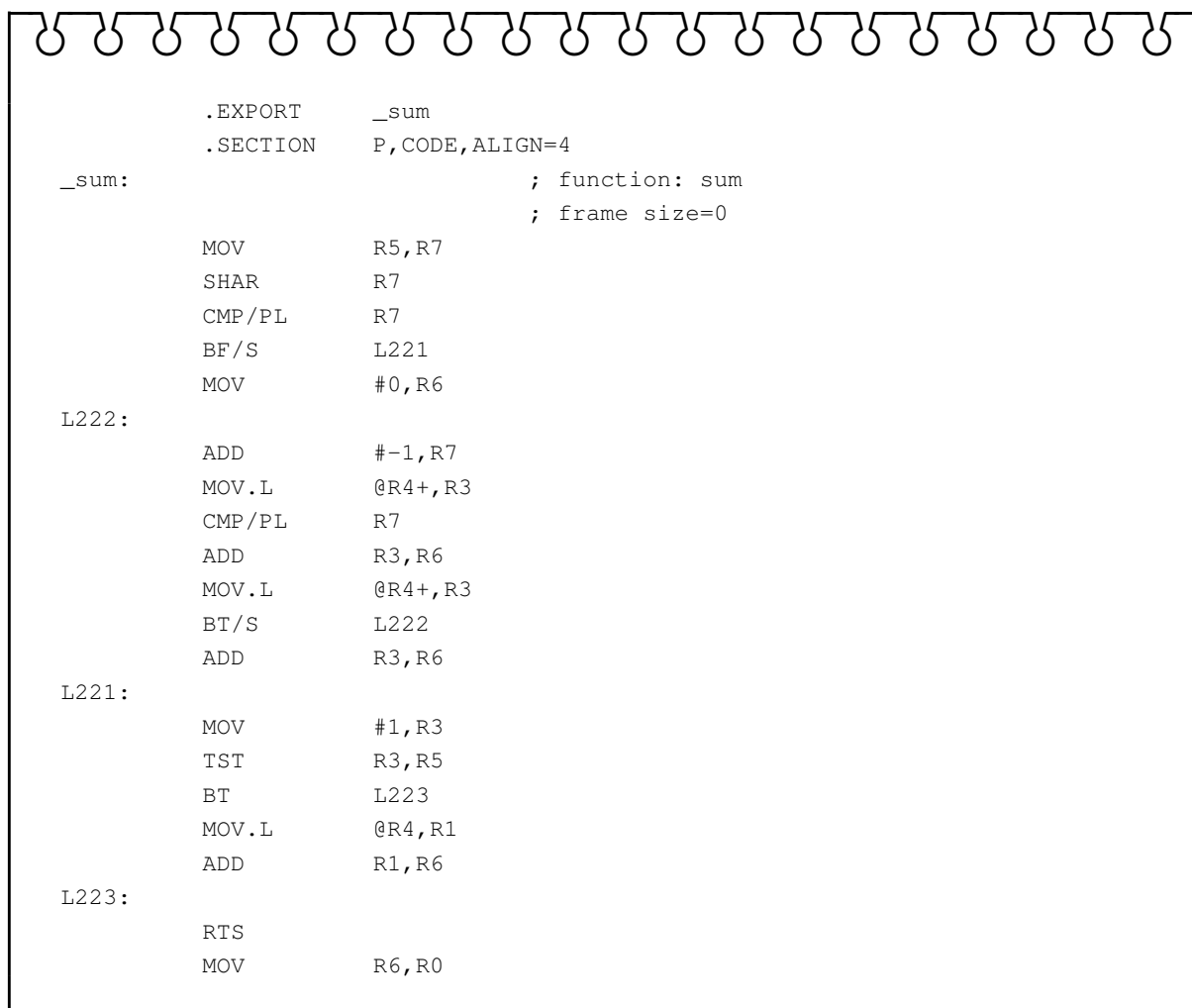


図 10 sum3b.src (sum3b.c のコンパイル結果 : 最適化あり)

4.6 最適化で気をつけること

最適化は非常に便利で強力な機能ですが、いいかげんなコーディングに対しては予想外のコードを吐き出してしまふ副作用も内包しています。

以下に、最適化を行う際に注意すべき点についていくつか紹介します。

ロード/ストア命令について この章で紹介したいいくつかのソースを見てもらおうと気づくかもしれませんが、Cソースで記述しているロード/ストア命令については基本的にコードの並べ替えを行わないようです。単なるメモリアクセスならまだしも、ペリフェラルの設定等の部分でポートアクセスなどを勝手に並べ替えられては困ることがおきますので、この辺の処理は概ね妥当といえます。なお、ポートやレジスタへのアクセスについては、`volatile` 型宣言を使用することをお勧めします。例えば、汎用ポート A を以下のように宣言します。

```
#define PADRH ((volatile u_short *)0xffff8380)
```

ループ処理について SHC コンパイラに限らないのですが、ループは最適化の標的になりやすい処理です。例えば、以下の関数マクロを作成したとします。

```
#define DELAY(x) {int i; for(i=0; i<x; i++);} /* Delay */
```

単純な時間稼ぎのためのこのようなループ処理は皆さんもよく使われるのではにのでしょうか。しかし、このループ処理は完全にこの系内で閉じていて他のコードに影響を及ぼさないため、最適化オプションによっては、この処理自体が削除される可能性があります。

これを防ぐためには、上記の例では以下のように宣言しておきます。

```
#define DELAY(x) {volatile int i; for(i=0; i<x; i++);} /* Delay */
```

`volatile` 型宣言は、先程も出てきましたが、意図的に最適化の対象から逃れられる効果をもっています。多用は禁物です。

関数呼出しについて 呼び出す関数が同一ファイルの場合、BSR インストラクションを使った高速な関数呼出しにできる場合があります。(図 11, 図 12 参照) また処理の最後に関数を呼び出した場合、「テールリカーション」最適化により関数戻り時のオーバーヘッドを抑えることができます。(図 13, 図 14 参照)


```
void    func(int *sum, int *tbl, int ct)
{
    register int    s;

    s = 0;
    for(; ct>0; ct--, tbl++) {
        s += (*tbl);
    }
    (*sum) = s;
    return;
}
int sum(int *tbl, int ct)
{
    int sum;

    func(&sum, tbl, ct);
    return(sum);
}
```

图 11 call1.c

```

        .EXPORT      _func
        .EXPORT      _sum
        .SECTION     P, CODE, ALIGN=4
_func:
                                ; function: func
                                ; frame size=0

        CMP/PL        R6
        BF/S          L225
        MOV           #0, R7
L226:
        ADD           #-1, R6
        MOV.L         @R5+, R2
        CMP/PL        R6
        BT/S          L226
        ADD           R2, R7
L225:
        RTS
        MOV.L         R7, @R4
_sum:
                                ; function: sum
                                ; frame size=16

        STS.L         PR, @-R15
        MOV           R5, R6
        ADD           #-12, R15
        MOV.L         R4, @ (4, R15)
        MOV.L         R5, @ (8, R15)
        MOV.L         @ (4, R15), R5
        BSR           _func
        MOV           R15, R4
        MOV.L         @R15, R0
        ADD           #12, R15
        LDS.L         @R15+, PR
        RTS
        NOP

```

図 12 call11.src (call11.c のコンパイル結果 : 最適化あり)

```

int func(int *tbl, int ct)
{
    register int    s;

    s = 0;
    for(; ct>0; ct--, tbl++) {
        s += (*tbl);
    }
    return(s);
}
int sum(int *tbl, int ct)
{
    return(func(tbl, ct));
}

```

図 13 call2.c

```

        .EXPORT      _func
        .EXPORT      _sum
        .SECTION     P, CODE, ALIGN=4
_func:                                     ; function: func
                                           ; frame size=0

        CMP/PL       R5
        BF/S         L223
        MOV          #0, R6
L224:
        ADD          #-1, R5
        MOV.L        @R4+, R2
        CMP/PL       R5
        BT/S         L224
        ADD          R2, R6
L223:
        RTS
        MOV          R6, R0
_sum:                                     ; function: sum
                                           ; frame size=0

        BRA          _func
        NOP

```

図 14 call2.src (call2.c のコンパイル結果：最適化あり)

5 最後に（コーディングテクニック一覧）

今まで述べてきたことを以下にまとめてみました。

1. プロトタイピングは必ず行います。
2. 算術式および論理式に注意します。冗長な記述で不具合等を回避できる場合があります。
3. コンパイル時の最適化処理により、パイプラインの乱れを最小限に抑えることができます。
4. トリッキーなコーディングはかえって処理効率を下げる可能性があります。
5. `volatile` 型宣言をうまく使って最適化処理をコントロールします。
6. 関数の引数は 4 つ以下になるようにします。また返り値も 32bit サイズ以下になるように工夫します。
7. 頻繁に呼び出す関数を 1 つのファイルにまとめることにより処理効率を上げることができます。

また、上記以外にもコーディング上のテクニックがあります。以下にいくつか紹介します。（文献 [2] でも詳しく紹介されています。こちらもぜひ参考にしてみてください。）

1. 乗算を行う際、データサイズを 2 byte 以下にしておくと処理速度が向上します。これは特にターゲットが SH-1 の場合に有効です。
2. イミディエイト値を 1 byte 以下にすると、データがコード内に埋めこまれるため処理効率が向上します。
3. `switch` 文は一般的に効率が悪いので、`if` 文や（分岐が少ない場合）関数テーブル（処理が似ている場合）で置き換えられないかどうか検討すべきです。
4. 数値比較をする場合、0 で比較するようにすると処理効率が向上します。

今回は SH マイコン特有と思われるコーディングテクニックについて紹介しました。現在マイコンプログラムでも C や C++ 言語が使えるようになり開発効率は著しく向上しましたが、本当に効率的なコードにするためには小手先のテクニックが必要となります。皆さんが実際に SH マイコンでシステムを組む際に、このレポートが少しでもお役に立てれば幸いです。

参考文献

- [1] 株式会社 日立製作所 半導体事業部. 『SH1/SH2 プログラミングマニュアル』, 株式会社 日立マイコンシステム技術情報センタ, Aug 1996.
- [2] 株式会社 日立製作所 半導体事業部. 『SuperH RISC engine ファミリ C コンパイラ編』, 株式会社日立マイコンシステム 技術ドキュメントセンタ, Jan 1996.