
効果的プログラム作成手法

効果的プログラム	作成手法	1
1 データ指定.....		5
1.1 局所変数 (データサイズ).....		6
1.2 大域変数 (符号).....		7
1.3 データの構造化.....		8
1.4 局所変数と大域変数.....		10
1.5 ポインタ変数の活用		12
2 関数呼び出し.....		13
2.1 関数のモジュール化.....		14
2.2 関数のインタフェース		15
3 演算方法		17
3.1 ループ内不変式の移動.....		18
3.2 ループ回数の削減		19
4 インライン展開.....		21
4.1 関数のインライン展開		22
4.2 アセンブラ埋め込みのインライン展開		24
5 グローバルベースレジスタ (GBR)の活用		26
6 プリフェッチ命令		28
7 マトリックス演算.....		30

SH シリーズ C コンパイラは最適化を行っています、プログラミングの工夫により一層の性能向上が可能です。

本章では、効果的なプログラム作成のために、ユーザに試みて頂きたい手法について記述します。

プログラムの評価基準には、実行速度が速いこととサイズが小さいことの 2 種類があります。

SH シリーズ C コンパイラでは、最適化を実行速度優先で行うことができます。このときには、コンパイルオプションに " speed" を指定してください。

効果的なプログラムを作成するための原則を以下に示します。

(1) 実行速度向上の原則

実行頻度の高い文、複雑な文で実行速度は決まるので、これらの処理を把握して、重点的に改良してください。

(2) サイズ縮小の原則

プログラムサイズ縮小のためには、類似処理の共通化、複雑な関数の見直しを行ってください。

コンパイラの最適化のため、実行速度が机上で検討したときとは異なる結果になることがあります。様々な手法を駆使し、実際にコンパイラで実行して確認しながら性能追及を進めてください。

本章のアセンブリ言語展開コードは

```
shc  C 言語ファイル  -code = asm code
```

のコマンドラインで取得しています。アセンブリ言語展開コードが CPU 毎に異なる場合のみ、その旨を記しています。今後、コンパイラの改善等により、アセンブリ言語展開コードは変わる可能性があります。

効果的プログラム作成手法の一覧を表 1 に示します。

表 1 効果的プログラム作成手法一覧

項番	項目	ROM 効率	RAM 効率	実行速度	参照
1	局所変数 (データサイズ)		-		1.1
2	大域変数 (符号)		-		1.2
3	乗算のデータサイズ		-		1.3
4	データの構造化		-		1.4
5	局所変数と大域変数		-		1.7
6	ポインタ変数の活用		-	-	1.8
7	関数のモジュール化		-		2.1
8	関数のインタフェース	-			2.3
9	ループ内不変式の移動	-	-		3.1
10	ループ回数の削減	×	-		3.2
11	関数のインライン展開	×	-		4.1
12	アセンブリコードのインライン展開	-	-		4.2
13	グローバルベースレジスタ (GBR) の活用		-		5
14	プリフェッチ命令	-	-		6
15	マトリックス演算		-		7

【注】表中の、× は以下の意味を示します。

...性能向上に効果あり

× ...性能低下の可能性あり

1 データ指定

データに関して考慮すべき事項を表 1.1 に示します。

表 1.1 データ指定における注意事項

項目	注意点	参照
データ型指定子、型修飾子	・データサイズを縮小しようとすると、プログラムサイズが増大する場合があります。データは用途を考えて型宣言してください。 ・符号あり/なしによりプログラムサイズが変わることがあるので、選択時に注意してください。 ・プログラム内で値が不変な初期値化データの場合、const 演算子を付けておくことで使用メモリ量の節約になります。	1.1 ~ 1.2
構造体の定義 / 参照	・頻繁に参照 / 変更するデータは構造体にして、ポインタ変数を用いることによりプログラムサイズを縮小できる場合があります。 ・ビットフィールドを使用すると、データサイズを縮小できます。	1.3
局所変数と大域変数	・局所変数の方が効率がよいので、局所変数として使用できるものは大域変数として宣言しないで必ず局所変数として宣言してください。	1.4
ポインタ型の活用	・配列型を用いたプログラムは、ポインタ型を用いて書き直せないか検討してください。	1.5
内蔵 ROM / RAM の活用	・外部メモリに比べ内蔵メモリへのアクセスは速いので、共通変数は内蔵メモリへ格納するようにしてください。	

1.1 局所変数 (データサイズ)

ポイント

局所変数のサイズは 4 バイトでとると、ROM 効率と実行速度を向上できる場合があります。

説明

日立 SuperH RISC engine ファミリの汎用レジスタは 4 バイトであるため、処理の基本は 4 バイトです。このため、1 バイト/2 バイトの局所変数を用いた演算があると、4 バイトに型変換するコードが付け加わります。1 バイト/2 バイトで十分な変数でも 4 バイトでとっておくとプログラムサイズが小さくなり、実行速度を向上できる場合があります。

使用例

1 から 10 までの総和を求めます。

改善前ソースコード	改善後ソースコード
<pre>int f(void) { char a = 10; int c = 0; for (; a > 0; a--) c += a; return(c); }</pre>	<pre>int f(void) { long a = 10; int c = 0; for (; a > 0; a--) c += a; return(c); }</pre>
改善前アセンブリ展開コード	改善後アセンブリ展開コード
<pre>_f: MOV #10,R4 MOV #0,R5 L217: EXTS.B R4,R3 ADD R3,R5 ADD #-1,R4 EXTS.B R4,R2 CMP/PL R2 BT L217 RTS MOV R5,R0</pre>	<pre>_f: MOV #10,R4 MOV #0,R5 L217: ADD R4,R5 ADD #-1,R4 CMP/PL R4 BT L217 RTS MOV R5,R0</pre>

改善前後のコードサイズと実行速度

	改善前	改善後
コードサイズ	20byte	16byte
実行速度	35cycle	23cycle

1.2 大域変数 (符号)

ポイント

式中に大域変数の型変換が含まれる場合、整数の型が signed でも unsigned でもよいときには signed で宣言すると、ROM 効率と実行速度を向上できます。

説明

日立 SuperH RISC engine ファミリでは、メモリから MOV 命令で 1 バイト/2 バイトのデータを転送するとき、unsigned のデータでは EXTU 命令を付加します。このため、unsigned 型整数の方が signed 型整数よりも効率が悪くなります。

使用例

変数 b に変数 a の値を代入します。

改善前ソースコード	改善後ソースコード
<pre> unsigned short a; unsigned short b; int c; void f() { c = b + a; } </pre>	<pre> short a; short b; int c; void f() { c = b + a; } </pre>
改善前アセンブリ展開コード	改善後アセンブリ展開コード
<pre> _f: MOV.L L218,R2 MOV.W @R2,R3 MOV.L L218+4,R0 EXTU.W R3,R3 MOV.W @R0,R1 EXTU.W R1,R1 ADD R1,R3 MOV.L L218+8,R1 RTS L218: MOV.L R3,@R1 .DATA.L _b .DATA.L _a .DATA.L _c </pre>	<pre> _f: MOV.L L218,R2 MOV.W @R2,R3 MOV.L L218+4,R0 MOV.W @R0,R1 ADD R1,R3 MOV.L L218+8,R1 RTS MOV.L R3,@R1 L218: .DATA.L _b .DATA.L _a .DATA.L _c </pre>

改善前後のコードサイズと実行速度

	改善前	改善後
コードサイズ	32byte	28byte
実行速度	6cycle	6cycle

1.3 データの構造化

ポイント

関連するデータを構造体で宣言すると、実行速度を向上できる場合があります。

説明

同一関数の中で何度も参照している場合、ベースアドレスがレジスタに割り付けば構造体の方が効率がよくなります。また、引数として渡す場合も効率が向上します。頻繁にアクセスするデータは構造体の先頭に集めると効果的です。

データを構造化すると、データの表現を変更するようなチューニングが容易になります。

使用例

変数 a, b, c に数値を代入します。

改善前ソースコード

```
int a, b, c;
void f()
{
    a = 1;
    b = 2;
    c = 3;
}
```

改善前アセンブリ展開コード

```
_f:
    MOV.L    L218,R2
    MOV      #2,R1
    MOV.L    L218+4,R0
    MOV      #1,R3
    MOV.L    R3,@R2
    MOV      #3,R3
    MOV.L    R1,@R0
    MOV.L    L218+8,R1
    RTS
L218:
    .DATA.L  _a
    .DATA.L  _b
    .DATA.L  _c
```

改善後ソースコード

```
struct s{
    int a;
    int b;
    int c;
} s1;
void f()
{
    register struct s *p=&s1;

    p->a = 1;
    p->b = 2;
    p->c = 3;
}
```

改善後アセンブリ展開コード

```
_f:
    MOV.L    L217,R4
    MOV      #1,R3
    MOV.L    R3,@R4
    MOV      #2,R2
    MOV.L    R2,@(4,R4)
    MOV      #3,R3
    RTS
L217:
    .DATA.L  _s1
```

改善前後のコードサイズと実行速度

	改善前	改善後
コードサイズ	32byte	20byte

実行速度	5cycle	4cycle
------	--------	--------

1.4 局所変数と大域変数

ポイント

一時変数、ループのカウンタ等、局所的に用いる変数は、関数の中で局所変数として宣言すると実行速度を向上できます。

説明

局所変数として使用できるものは、大域変数として宣言しないで必ず局所変数として宣言してください。大域変数は、関数呼び出しやポインタ操作によって値が変化してしまう可能性があるため、大域的最適化の対象にはなりません。

局所変数を使用すると次の利点があります。

- アクセスコストが安い。
- レジスタに割り付けられる可能性がある。
- 最適化の対象になる。

使用例

10 回ループさせます。

改善前ソースコード	改善後ソースコード
<pre>int i; void f() { for (i = 0; i < 10; i++); }</pre>	<pre>void f() { int i; for (i = 0; i < 10; i++); }</pre>
改善前アセンブリ展開コード	改善後アセンブリ展開コード
<pre>_f: MOV.L L218+2,R4 MOV #0,R3 MOV #10,R5 BRA L216 L217: MOV.L @R4,R1 ADD #1,R1 MOV.L R1,@R4 L216: MOV.L @R4,R3 CMP/GE R5,R3 BF L217 RTS NOP L218: .RES.W 1 .DATA.L _i</pre>	<pre>_f: MOV #10,R5 MOV #0,R4 L216: ADD #1,R4 CMP/GE R5,R4 BF L216 RTS NOP</pre>

改善前後のコードサイズと実行速度

	改善前	改善後
コードサイズ	3 6byte	1 2byte
実行速度	4 8cycle	1 4cycle

1.5 ポインタ変数の活用

ポイント

配列型を用いたプログラムはポインタ型を用いて書き直すと、実行速度を向上できる場合があります。

説明

配列参照 $a[i]$ は、 $a[0]$ のアドレスに i 番目要素のアドレスを加算したコードが生成されます。ポインタ変数を用いれば、変数や演算の数を削減できる場合があります。

使用例

10 (=count) 個の整数の合計を求めます。

改善前ソースコード	改善後ソースコード
<pre>int f(int data[], int count) { int ret = 0, i; for (i = 0; i < count; i++) ret += data[i]; return ret; }</pre>	<pre>int f(int *data, int count) { int ret = 0; for (; count > 0; count--) ret += *data++; return ret; }</pre>
改善前アセンブリ展開コード	改善後アセンブリ展開コード
<pre>_f: MOV #0,R7 MOV R7,R6 CMP/PL R5 BF L219 L220: MOV.L @R4+,R3 ADD #1,R6 ADD R3,R7 CMP/GE R5,R6 BF L220 L219: RTS MOV R7,R0</pre>	<pre>_f: MOV #0,R6 CMP/PL R5 BF L218 L219: ADD #-1,R5 MOV.L @R4+,R3 CMP/PL R5 ADD R3,R6 BT L219 L218: RTS MOV R6,R0</pre>

改善前後のコードサイズと実行速度

	改善前	改善後
コードサイズ	26byte	20byte
実行速度	34cycle	48cycle

2 関数呼び出し

関数呼び出しに関して考慮すべき事項を表 2.1 に示します。

表 2.1 関数呼び出しにおける注意事項

項目	注意点	参照
関数位置	関連の深い関数は 1 ファイルにまとめてください。	2.1
インタフェース	引数が全てレジスタに割り付くように (4 個まで) 引数の数を厳選してください。 引数が多い場合、構造体にしてポインタで渡してください。	2.2
マクロへの置換	関数呼び出しが多数ある場合、マクロにすれば実行速度を向上できます。ただし、マクロにするとプログラムサイズが増大するので、状況により選択してください。	-

2.1 関数のモジュール化

ポイント

関連の深い関数は 1 ファイルにまとめることにより実行速度を向上できます。

説明

異なるファイルにある関数を呼び出す場合、JSR 命令に展開されますが、同一ファイル内の関数呼び出しでは、呼び出し範囲が近いと BSR 命令に展開され、高速かつコンパクトなオブジェクトが生成されます。

また、モジュール化によって、チューンアップ時の修正が容易になります。

使用例

関数 f から関数 g を呼び出します。

改善前ソースコード	改善後ソースコード
<pre>extern g(); int f() { g(); }</pre>	<pre>int g(void) { } int f(void) { g(); }</pre>
改善前アセンブリ展開コード	改善後アセンブリ展開コード
<pre>_f: MOV.L L216+2,R3 JMP @R3 NOP L216: .RES.W 1 .DATA.L _g</pre>	<pre>_g: RTS NOP _f: BRA _g NOP</pre>

改善前後のコードサイズと実行速度

	改善前	改善後
コードサイズ	12byte	4byte
実行速度	5cycle	3cycle

備考

BSR 命令で呼び出せる範囲は ± 4096 バイト (± 2048 命令) です。

ファイルのサイズが大きくなりすぎると BSR を有効に使用できなくなります。

2.2 関数のインタフェース

ポイント

関数の引数を工夫することによりRAM 容量を削減でき、実行速度も向上できます。

(4.12 参照)

説明

引数が全てレジスタに乗るように (4 個まで) 引数の数を厳選してください。引数が多い場合は、構造体にしてポインタで渡してください。引数がレジスタに乗れば、呼び出し、関数の出入り口の処理が簡単になります。また、スタック領域も節約できます。

なお、レジスタは R0～R3 がワークレジスタ、R4～R7 が引数用、R8～R14 が局所変数用です。

使用例

関数 f の引数が引数用レジスタ個数よりも多く 5 個あります。

改善前ソースコード	改善後ソースコード
<pre>int f(int, int, int, int, int); void g() { f(1, 2, 3, 4, 5); }</pre>	<pre>struct b{ int a, b, c, d, e; } b1 = {1, 2, 3, 4, 5}; int f(struct b *p); void g() { f(&b1); }</pre>
改善前アセンブリ展開コード	改善後アセンブリ展開コード
<pre>_g: STS.L PR,@-R15 MOV #5,R3 MOV.L L216+2,R2 MOV #4,R7 MOV.L R3,@-R15 MOV #3,R6 MOV #2,R5 JSR @R2 MOV #1,R4 ADD #4,R15 LDS.L @R15+,PR RTS NOP L216: .RES.W 1 .DATA.L _f</pre>	<pre>_g: MOV.L L217,R4 MOV.L L217+4,R3 JMP @R3 NOP L217: .DATA.L _b1 .DATA.L _f</pre>

改善前後のコードサイズと実行速度

	改善前	改善後
コードサイズ	32byte	16byte
実行速度	10cycle	2cycle

3 演算方法

演算方式に関して考慮すべき事項を表 3.1 に示します。

表 3.1 演算方式における注意事項

項目	注意点	参照
不変式 / 共通式の統合 / 移動	関数内で共通に使用している部分式の一時変数への置換を検討してください。 for 文内で使用する不変式を for 文外に出してください。	3.1
ループ回数の削減	ループ条件が同一または類似しているループ文のマージを検討してください。 ループの展開を試みてください。	3.2
高速なアルゴリズムの利用	配列におけるクイックソートのような計算時間が少なくすむアルゴリズムを検討してください。	

3.1 ループ内不変式の移動

ポイント

ループ内で値が変更されない式は、ループ開始前に計算すると実行速度を向上できます。

説明

ループ内で値が変更されない式をループ開始前に計算すると、毎回の計算が省略でき、実行命令数を低減できます。

使用例

配列 a[]に配列要素 b[5]を代入します。

改善前ソースコード	改善後ソースコード
<pre>void f(void) { int i,j; j = 5; for (i=0; i < 100; i++) a[i] = b[j]; }</pre>	<pre>void f(void) { int i,j,t; j = 5; for (i=0, t=b[j]; i < 100; i++) a[i] = t; }</pre>
改善前アセンブリ展開コード	改善後アセンブリ展開コード
<pre>_f: MOV.L L220,R4 MOV #100,R6 MOV.L L220+4,R7 MOV #0,R5 L219: MOV.L @R7,R3 ADD #1,R5 MOV.L R3,@R4 CMP/GE R6,R5 ADD #4,R4 BF L219 RTS NOP L220: .DATA.L _a .DATA.L H'00000014+_b</pre>	<pre>_f: MOV.L L221,R7 MOV #100,R6 MOV.L L221+4,R4 MOV #0,R5 MOV.L @R7,R7 L220: ADD #1,R5 MOV.L R7,@R4 CMP/GE R6,R5 ADD #4,R4 BF L220 RTS NOP L221: .DATA.L H'00000014+_b .DATA.L _a</pre>

改善前後のコードサイズと実行速度

	改善前	改善後
コードサイズ	40byte	36byte
実行速度	315cycle	275cycle

3.2 ループ回数の削減

ポイント

ループを展開すると 実行速度は大幅に向上できます。

説明

ループの展開は特に内側のループが有効です。ループの展開によりプログラムサイズは増大するので、プログラムサイズを犠牲にしても実行速度を向上させたい場合に適用してください。

使用例

配列 a[]を初期化します。

改善前ソースコード	改善後ソースコード
<pre>void f() { int i; for (i = 0; i < 100; i++) a[i] = 0; }</pre>	<pre>void f() { int i; for (i = 0; i < 100; i+=2) { a[i] = 0; a[i+1] = 0; } }</pre>
改善前アセンブリ展開コード	改善後アセンブリ展開コード
<pre>_f: MOV.L L218+2,R4 MOV #0,R6 MOV R6,R5 MOV #100,R7 L217: MOV.L R6,@R4 ADD #1,R5 ADD #4,R4 CMP/GE R7,R5 BF L217 RTS NOP L218: .RES.W 1 .DATA.L _a</pre>	<pre>_f: MOV.L L219,R5 MOV #0,R4 MOV R4,R6 MOV #100,R7 L218: MOV.L R4,@R5 ADD #2,R6 MOV.L R4,@(4,R5) CMP/GE R7,R6 ADD #8,R5 BF L218 RTS NOP L219: .DATA.L _a</pre>

改善前後のコードサイズと実行速度

	改善前	改善後
コードサイズ	32byte	36byte
実行速度	228cycle	180cycle

4 インライン展開

インライン展開に関して考慮すべき事項を表 4.1 に示します。

表 4.1 インライン展開における注意事項

項目	注意点	参照
関数のインライン展開	頻繁に呼び出される関数はインライン展開を試みてください。 ただし、関数を展開するとプログラムサイズが増大するので実行速度とROM 容量との兼ね合いで選択してください。	4.1
アセンブラ埋め込み インライン展開	・アセンブラコードで記述されたプログラムをC 言語の関数と同じインタフェースで呼び出せます。	4.2

4.1 関数のインライン展開

ポイント

頻繁に呼び出される関数をインライン展開すると実行速度を向上できます。

説明

頻繁に呼び出される関数をインライン展開することにより、実行速度の向上が図れます。特にループ内で呼ばれる関数などを展開すると大きな効果を得られる場合もあります。しかし、インライン展開をした場合、プログラムサイズが増大する傾向にありますので、プログラムサイズを犠牲にしても実行速度を向上させたい場合に適用してください。

使用例

配列 a と配列 b の要素を交換します。

改善前ソースコード

```
int x[10], y[10];
static void g(int *a, int *b, int i)
{
    int temp;

    temp = a[i];
    a[i] = b[i];
    b[i] = temp;
}

void f ()
{
    int i;

    for (i=0;i<10;i++)
        g(x, y, i);
}
```

改善前アセンブリ展開コード

```
_g:
        ADD        #-4,R15
        MOV        R6,R7
        SHLL2      R7
        MOV.L      R7,@R15
        ADD        R4,R7
        MOV.L      @R7,R6
        MOV.L      @R15,R4
        ADD        R5,R4
        MOV.L      @R4,R3
        MOV.L      R3,@R7
        MOV.L      R6,@R4
        RTS
        ADD        #4,R15

_f:
        MOV.L      R14,@-R15
        MOV.L      R13,@-R15
        MOV        #0,R14
        MOV.L      R12,@-R15
        MOV        #10,R13
        MOV.L      R11,@-R15
```

改善後ソースコード

```
int x[10], y[10];
#pragma inline (g)
static void g(int *a, int *b, int i)
{
    int temp;

    temp = a[i];
    a[i] = b[i];
    b[i] = temp;
}

void f ()
{
    int i;

    for (i=0;i<10;i++)
        g(x, y, i);
}
```

改善後アセンブリ展開コード

```
_g:
        ADD        #-4,R15
        MOV        R6,R7
        SHLL2      R7
        MOV.L      R7,@R15
        ADD        R4,R7
        MOV.L      @R7,R6
        MOV.L      @R15,R4
        ADD        R5,R4
        MOV.L      @R4,R3
        MOV.L      R3,@R7
        MOV.L      R6,@R4
        RTS
        ADD        #4,R15

_f:
        MOV        #0,R4
        MOV.L      R12,@-R15
        MOV        #10,R12
        MOV.L      R11,@-R15
        MOV.L      R10,@-R15
        MOV.L      L231+2,R10
```

L224:	STS.L	PR,@-R15	L230:	MOV.L	L231+6,R11
	MOV.L	L225+2,R11		MOV	R4,R0
	MOV.L	L225+6,R12		MOV	R11,R1
	MOV	R14,R6		MOV	R10,R6
	MOV	R12,R5		SHLL2	R0
	BSR	_g		MOV	R0,R7
	MOV	R11,R4		ADD	R6,R7
	ADD	#1,R14		MOV.L	@R7,R6
	CMP/GE	R13,R14		MOV	R0,R5
	BF	L224		ADD	R1,R5
	LDS.L	@R15+,PR		ADD	#1,R4
	MOV.L	@R15+,R11		MOV.L	@R5,R3
	MOV.L	@R15+,R12		CMP/GE	R12,R4
	MOV.L	@R15+,R13		MOV.L	R3,@R7
	RTS			MOV.L	R6,@R5
L225:	MOV.L	@R15+,R14		BF	L230
	.RES.W	1	L231:	MOV.L	@R15+,R10
	.DATA.L	_x		MOV.L	@R15+,R11
	.DATA.L	_y		RTS	
				MOV.L	@R15+,R12
				.RES.W	1
				.DATA.L	_x
				.DATA.L	_y

改善前後のコードサイズと実行速度

	改善前	改善後
コードサイズ	108byte	112byte
実行速度	173cycle	89cycle

4.2 アセンブラ埋め込みのインライン展開

ポイント

Cプログラム中にアセンブラコードを記述し、実行速度を向上できます。

説明

性能上、特に実行速度を向上したい場合、アセンブラで記述したいことがあります。 そのような場合、必要な部分だけをアセンブラで記述し、その部分をC言語の関数と同じ要領で呼び出すことができます。

使用例

配列 big の要素の上位バイトと下位バイトを入れ換えて、配列 little に格納します。

改善前ソースコード

```
#define A_MAX 10
typedef unsigned char UChar;
short big[A_MAX], little[A_MAX];
short swap(short p1)
{
    short ret;

    *((UChar *)(&ret)+1) =
        *((UChar *)(&p1));
    *((UChar *)(&ret)) =
        *((UChar *)(&p1)+1);
    return ret;
}

void f (void)
{
    int i;
    short *x, *y;

    x = little;
    y = big;
    for(i=0; i<A_MAX; i++,
        x++, y++){
        *x = swap(*y);
    }
}
```

改善前アセンブリ展開コード

```
_swap:
    ADD     #-8,R15
    MOV     R15,R3
    ADD     #6,R3
    MOV     R15,R2
    MOV.W   R4,@R3
    MOV     R15,R0
    ADD     #6,R0
    MOV     R15,R3
    MOV.B   @R0,R0
    MOV.B   R0,@(1,R2)
    MOV     R15,R2
    ADD     #6,R2
    MOV.B   @(1,R2),R0
    MOV.B   R0,@R3

_f:
    MOV.L   R14,@-R15
    MOV     #0,R14
    MOV.L   R13,@-R15
    MOV.L   R12,@-R15
    MOV.L   R11,@-R15
    MOV     #10,R11
    MOV.L   L226+2,R13
    MOV.L   L226+6,R12
L224:
```

改善後ソースコード

```
#define A_MAX 10
#pragma inline_asm (swap)
typedef unsigned char UChar;
short big[A_MAX], little[A_MAX];
short swap(short p1)
{
    SWAP.B R4,R0
}

void f (void)
{
    int i;
    short *x, *y;

    x = little;
    y = big;
    for(i=0; i<A_MAX; i++,
        x++, y++){
        *x = swap(*y);
    }
}
```

改善後アセンブリ展開コード

```
_swap:
    SWAP.B R4,R0
    .ALIGN 4
    RTS
    NOP

_f:
    MOV.L   R14,@-R15
    MOV     #0,R14
    MOV.L   R13,@-R15
    MOV.L   R12,@-R15
    MOV.L   R11,@-R15
    MOV     #10,R11
    MOV.L   L226+2,R13
    MOV.L   L226+6,R12
L224:
```

	MOV.W	@R15,R0		MOV.W	@R12,R4
	RTS			BRA	L225
	ADD	#8,R15		NOP	
_f:			L226:	.RES.W	1
	MOV.L	R14,@-R15		.DATA.L	_little
	MOV.L	R13,@-R15		.DATA.L	_big
	MOV	#0,R14			
	MOV.L	R12,@-R15	L225:	SWAP.B	R4,R0
	MOV.L	R11,@-R15		.ALIGN	4
	STS.L	PR,@-R15		MOV.W	R0,@R13
	MOV	#10,R11		ADD	#1,R14
	MOV.L	L227+2,R13		ADD	#2,R13
L226:	MOV.L	L227+6,R12		ADD	#2,R12
	BSR	_swap		CMP/GE	R11,R14
	MOV.W	@R12+,R4		BT	L227
	MOV.W	R0,@R13		MOV.L	L228,R2
	ADD	#1,R14		JMP	@R2
	ADD	#2,R13		NOP	
	CMP/GE	R11,R14	L227:		
	BF	L226		MOV.L	@R15+,R11
	LDS.L	@R15+,PR		MOV.L	@R15+,R12
	MOV.L	@R15+,R11		MOV.L	@R15+,R13
	MOV.L	@R15+,R12		RTS	
	MOV.L	@R15+,R13		MOV.L	@R15+,R14
	RTS		L228:	.DATA.L	L224
L227:	MOV.L	@R15+,R14			
	.RES.W	1			
	.DATA.L	_little			
	.DATA.L	_big			

改善前後のコードサイズと実行速度

	改善前	改善後
コードサイズ	116byte	120byte
実行速度	195cycle	107cycle

5 グローバルベースレジスタ(GBR)の活用

ポイント

外部変数を GBR を使ったオフセット参照にすることにより、性能を向上させることができます。

説明

頻繁にアクセスされる外部変数は GBR をベースレジスタとしたオフセット参照にすることにより、コンパクトなオブジェクトが生成されます。また、実行命令数の削減にもつながるので実行速度が向上する場合があります。

使用例

構造体 y の内容を構造体 x に代入します。

改善前ソースコード

```
struct {
    char c1;
    char c2;
    short s1;
    short s2;
    long l1;
    long l2;
} x, y;

void f (void)
{
    x.c1 = y.c1;
    x.c2 = y.c2;
    x.s1 = y.s1;
    x.s2 = y.s2;
    x.l1 = y.l1;
    x.l2 = y.l2;
}
```

改善前アセンブリ展開コード

```
_f:
    MOV.L    L217+2,R5
    MOV.L    L217+6,R4
    MOV.B    @R5,R3
    MOV.B    R3,@R4
    MOV.B    @(1,R5),R0
    MOV.B    R0,@(1,R4)
    MOV.W    @(2,R5),R0
    MOV.W    R0,@(2,R4)
    MOV.W    @(4,R5),R0
    MOV.W    R0,@(4,R4)
    MOV.L    @(8,R5),R3
    MOV.L    R3,@(8,R4)
    MOV.L    @(12,R5),R2
    RTS
    MOV.L    R2,@(12,R4)
L217:
    .RES.W    1
    .DATA.L   _y
    .DATA.L   _x
```

改善後ソースコード

```
#pragma gbr_base(x,y)
struct {
    char c1;
    char c2;
    short s1;
    short s2;
    long l1;
    long l2;
} x, y;

void f (void)
{
    x.c1 = y.c1;
    x.c2 = y.c2;
    x.s1 = y.s1;
    x.s2 = y.s2;
    x.l1 = y.l1;
    x.l2 = y.l2;
}
```

改善後アセンブリ展開コード

```
_f:
    MOV.B    @(_y-(STARTOF
$G0),GBR),R0
    MOV.B    R0,@(_x-(STARTOF
$G0),GBR)
    MOV.B    @(_y-(STARTOF
$G0)+1,GBR),R0
    MOV.B    R0,@(_x-(STARTOF
$G0)+1,GBR)
    MOV.W    @(_y-(STARTOF
$G0)+2,GBR),R0
    MOV.W    R0,@(_x-(STARTOF
$G0)+2,GBR)
    MOV.W    @(_y-(STARTOF
$G0)+4,GBR),R0
    MOV.W    R0,@(_x-(STARTOF
$G0)+4,GBR)
    MOV.L    @(_y-(STARTOF
$G0)+8,GBR),R0
    MOV.L    R0,@(_x-(STARTOF
```



```
$G0)+8,GBR)
MOV.L    @(_y-
(STARTOF$G0)+12,GBR),R0
RTS
MOV.L    R0,@(_x-
(STARTOF$G0)+12,GBR)
```

改善前後のコードサイズと実行速度

	改善前	改善後
コードサイズ	40byte	26byte
実行速度	13cycle	11cycle

6 プリフェッチ命令

ポイント

配列変数をアクセスするとき、使用に先立ってプリフェッチ命令を実行すると、実行速度の向上が期待できます。

説明

ループで配列を順次アクセスする場合、配列のメンバ参照に先立ちプリフェッチを行うことで、実行速度が向上します。また、ループを展開することで、さらに効果的にプリフェッチが行えます。

なお、プリフェッチ命令は、連続して実行しても速度の向上は期待できませんので、前のプリフェッチ命令が完了するように十分に離して実行してください。

使用例

配列 a と配列 b の各要素の積を配列 c に格納します。

改善前ソースコード

```
int a[1200], b[1200], c[1200];
int f (void)
{
    int i;
    int *pa, *pb, *pc;

    for (pa=a, pb=b, pc=c,
         i=0; i<1200; i+=4){
        *pc++ = *pa++ * *pb++;
        *pc++ = *pa++ * *pb++;
        *pc++ = *pa++ * *pb++;
        *pc++ = *pa++ * *pb++;
    }
}
```

改善前アセンブリ展開コード

```
_f:
        MOV.L    R14,@-R15
        MOV      #0,R7
        MOV.L    L224+2,R6
        STS.L    PR,@-R15
        MOV.W    L224,R14
        MOV.L    L224+6,R4
        MOV.L    L224+10,R5
L223:   MOV.L    @R6+,R1
        MOV.L    L224+14,R2
        JSR      @R2
```

改善後ソースコード

```
#include <umachine.h>
int a[1200], b[1200], c[1200];
int f (void)
{
    int i;
    int *pa, *pb, *pc;

    for (pa=a, pb=b, pc=c,
         i=0; i<1200; i+=4){
#ifdef PREF1
        prefetch(pa+8);
#endif
        *pc++ = *pa++ * *pb++;
        *pc++ = *pa++ * *pb++;
#ifdef PREF2
        prefetch(pb+8);
#endif
        *pc++ = *pa++ * *pb++;
        *pc++ = *pa++ * *pb++;
    }
}
```

改善後アセンブリ展開コード

```
_f:
        MOV.L    R14,@-R15
        MOV      #0,R7
        MOV.L    L224+2,R6
        STS.L    PR,@-R15
        MOV.W    L224,R14
        MOV.L    L224+6,R4
        MOV.L    L224+10,R5
L223:   MOV.L    @R6+,R1
        MOV.L    L224+14,R2
        JSR      @R2
```

	MOV.L @R4+,R0		MOV.L @R4+,R0
	MOV.L R0,@R5		MOV.L R0,@R5
	MOV.L @R6+,R1		MOV.L @R6+,R1
	ADD #4,R5		ADD #4,R5
	MOV.L L224+14,R2		MOV.L L224+14,R2
	JSR @R2		JSR @R2
	MOV.L @R4+,R0		MOV.L @R4+,R0
	MOV.L R0,@R5		MOV.L R0,@R5
	MOV.L @R6+,R1		MOV.L @R6+,R1
	ADD #4,R5		ADD #4,R5
	MOV.L L224+14,R2		MOV.L L224+14,R2
	JSR @R2		JSR @R2
	MOV.L @R4+,R0		MOV.L @R4+,R0
	MOV.L R0,@R5		MOV.L R0,@R5
	MOV.L @R6+,R1		MOV.L @R6+,R1
	ADD #4,R5		ADD #4,R5
	MOV.L L224+14,R2		MOV.L L224+14,R2
	JSR @R2		JSR @R2
	MOV.L @R4+,R0		MOV.L @R4+,R0
	ADD #4,R7		ADD #4,R7
	MOV.L R0,@R5		MOV.L R0,@R5
	CMP/GE R14,R7		CMP/GE R14,R7
	ADD #4,R5		ADD #4,R5
	BF L223		BF L223
	LDS.L @R15+,PR		LDS.L @R15+,PR
	RTS		RTS
	MOV.L @R15+,R14		MOV.L @R15+,R14
L224:		L224:	
	.DATA.W H'04B0		.DATA.W H'04B0
	.DATA.L _a		.DATA.L _a
	.DATA.L _b		.DATA.L _b
	.DATA.L _c		.DATA.L _c
	.DATA.L ___multi		.DATA.L ___multi

改善前後のコードサイズと実行速度

	改善前	改善後 (PREF 1, 2)
コードサイズ	84byte	96byte
実行速度	61650cycle	57930cycle

7 マトリックス演算

ポイント

行列演算の際、組み込み関数を使用すると、実行速度の向上が期待できます。

その際、乗数となる配列はあらかじめ、浮動小数点拡張レジスタに格納しておく必要があります。

説明

4行 4列の配列の積は通常ならば、ループを用いて順次演算を行うため処理が複雑になり実行速度に期待できないが、SH7091 ではマトリックス演算を組み込み関数でサポートしているため、この関数を使用することにより実行速度の大幅な向上が期待できます。

使用例

配列 data と配列 tbl の積を配列 ret に格納します。

改善前ソースコード	改善後ソースコード
<pre>void mtrx4mul1 (float data[4][4], float tbl[4][4], float ret[4][4]) { int i,j,k; for(i=0;i<4;i++) { for(j=0;j<4;j++) { for(k=0;k<4;k++) { ret[i][j]+= data[i][k]*tbl[k][j]; } } } }</pre>	<pre>#include <machine.h> void _mtrx4mul (float data[4][4], float tbl[4][4],float ret[4][4]) { ld_ext(data);/*拡張レジスタに格納*/ mtrx4mul(tbl,ret);/*演算部分*/ }</pre>
改善前アセンブリ展開コード	改善後アセンブリ展開コード
<pre>_mtrx4mul1: MOV.L R14,@-R15 MOV.L R13,@-R15 MOV #4,R14 MOV.L R12,@-R15 MOV.L R11,@-R15 MOV.L R10,@-R15 MOV.L R9,@-R15 MOV.L R8,@-R15 MOV #0,R8 ADD #-4,R15 MOV.L R8,@R15 L255: MOV.L @R15,R11 SHLL2 R11 SHLL2 R11 MOV R8,R9 L256: MOV R9,R12</pre>	<pre>_mtrx4mul: ADD #-12,R15 MOV.L R4,@(8,R15) MOV.L R5,@(4,R15) MOV.L R6,@R15 MOV.L @(8,R15),R2 FRCHG FMOV.S @R2+,FR0 FMOV.S @R2+,FR1 FMOV.S @R2+,FR2 FMOV.S @R2+,FR3 FMOV.S @R2+,FR4 FMOV.S @R2+,FR5 FMOV.S @R2+,FR6 FMOV.S @R2+,FR7 FMOV.S @R2+,FR8 FMOV.S @R2+,FR9 FMOV.S @R2+,FR10 FMOV.S @R2+,FR11</pre>

L257:	MOV	#0,R7	FMOV.S	@R2+,FR12
	SHLL2	R12	FMOV.S	@R2+,FR13
	MOV	#0,R13	FMOV.S	@R2+,FR14
	MOV	R8,R10	FMOV.S	@R2+,FR15
	ADD	R5,R7	FRCHG	
			MOV.L	@(4,R15),R3
	MOV	R11,R3	MOV.L	@R15,R1
	ADD	R4,R3	FMOV.S	@R3+,FR0
	ADD	R13,R3	FMOV.S	@R3+,FR1
	MOV	R11,R0	FMOV.S	@R3+,FR2
	ADD	R6,R0	FMOV.S	@R3+,FR3
	ADD	R12,R0	FTRV	XMTRX,FV0
	MOV.L	R0,@-R15	ADD	#16,R1
	FMOV.S	@R3,FR3	FMOV.S	FR3,@-R1
	STS	FPSCR,R3	FMOV.S	FR2,@-R1
	MOV.L	L259+10,R1	FMOV.S	FR1,@-R1
	OR	R1,R3	FMOV.S	FR0,@-R1
	MOV.L	@R15+,R2	FMOV.S	@R3+,FR0
	MOV	R12,R0	FMOV.S	@R3+,FR1
	LDS	R3,FPSCR	FMOV.S	@R3+,FR2
	FMOV.S	@R2,FR2	FMOV.S	@R3+,FR3
	FMOV.S	@(R0,R7),FR0	FTRV	XMTRX,FV0
	ADD	#16,R7	ADD	#32,R1
	ADD	#1,R10	FMOV.S	FR3,@-R1
	FMAC	FR0,FR3,FR2	FMOV.S	FR2,@-R1
	FMOV.S	FR2,@R2	FMOV.S	FR1,@-R1
	CMP/GE	R14,R10	FMOV.S	FR0,@-R1
	BF/S	L257	FMOV.S	@R3+,FR0
	ADD	#4,R13	FMOV.S	@R3+,FR1
	ADD	#1,R9	FMOV.S	@R3+,FR2
	CMP/GE	R14,R9	FMOV.S	@R3+,FR3
	BF	L256	FTRV	XMTRX,FV0
	MOV.L	@R15,R3	ADD	#32,R1
	ADD	#1,R3	FMOV.S	FR3,@-R1
	CMP/GE	R14,R3	FMOV.S	FR2,@-R1
	BF/S	L255	FMOV.S	FR1,@-R1
	MOV.L	R3,@R15	FMOV.S	FR0,@-R1
	ADD	#4,R15	FMOV.S	@R3+,FR0
	MOV.L	@R15+,R8	FMOV.S	@R3+,FR1
	MOV.L	@R15+,R9	FMOV.S	@R3+,FR2
	MOV.L	@R15+,R10	FMOV.S	@R3+,FR3
	MOV.L	@R15+,R11	FTRV	XMTRX,FV0
	MOV.L	@R15+,R12	ADD	#32,R1
	MOV.L	@R15+,R13	FMOV.S	FR3,@-R1
	RTS		FMOV.S	FR2,@-R1
	MOV.L	@R15+,R14	FMOV.S	FR1,@-R1
			FMOV.S	FR0,@-R1
			ADD	#12,R15
			RTS	
			NOP	

改善前後のコードサイズと実行速度

	改善前	改善後
コードサイズ	36byte	12byte
実行速度	951cycle	88cycle