お客様各位

カタログ等資料中の旧社名の扱いについて

2010年4月1日を以ってNECエレクトロニクス株式会社及び株式会社ルネサステクノロジが合併し、両社の全ての事業が当社に承継されております。従いまして、本資料中には旧社名での表記が残っておりますが、当社の資料として有効ですので、ご理解の程宜しくお願い申し上げます。

ルネサスエレクトロニクス ホームページ（http://www.renesas.com）

2010年4月1日
ルネサスエレクトロニクス株式会社

【発行】ルネサスエレクトロニクス株式会社（http://www.renesas.com）

【問い合わせ先】http://japan.renesas.com/inquiry

ご注意書き

1. 本資料に記載されている内容は本資料発行時点のものであり、予告なく変更することがあります。当社製品のご購入およびご使用にあたりましては、事前に当社営業窓口で最新の情報をご確認いただきますとともに、当社ホームページなどを通じて公開される情報に常にご注意ください。
2. 本資料に記載された当社製品および技術情報の使用に関連し発生した第三者の特許権、著作権その他の知的財産権の侵害等に関し、当社は、一切その責任を負いません。当社は、本資料に基づき当社または第三者の特許権、著作権その他の知的財産権を何ら許諾するものではありません。
3. 当社製品を改造、改変、複製等しないでください。
4. 本資料に記載された回路、ソフトウェアおよびこれらに関連する情報は、半導体製品の動作例、応用例を説明するものです。お客様の機器の設計において、回路、ソフトウェアおよびこれらに関連する情報を使用する場合には、お客様の責任において行ってください。これらの使用に起因しお客様または第三者に生じた損害に関し、当社は、一切その責任を負いません。
5. 輸出に際しては、「外国為替及び外国貿易法」その他輸出関連法令を遵守し、かかる法令の定めるところにより必要な手続を行ってください。本資料に記載されている当社製品および技術を大量破壊兵器の開発等の目的、軍事利用の目的その他軍事用途の目的で使用しないでください。また、当社製品および技術を国内外の法令および規則により製造・使用・販売を禁止されている機器に使用することができません。
6. 本資料に記載されている情報は、正確を期すため慎重に作成したものですが、誤りがないことを保証するものではありません。万一、本資料に記載されている情報の誤りに起因する損害がお客様に生じた場合においても、当社は、一切その責任を負いません。
7. 当社は、当社製品の品質水準を「標準水準」、「高品質水準」および「特定水準」に分類しております。また、各品質水準は、以下に示す用途に製品が使われることを意図しておりますので、当社製品の品質水準をご確認ください。お客様は、当社の文書による事前の承諾を得ることなく、「特定水準」に分類された用途に当社製品を使用することができません。また、お客様は、当社の文書による事前の承諾を得ることなく、意図されていない用途に当社製品を使用することができません。当社の文書による事前の承諾を得ることなく、「特定水準」に分類された用途または意図されていない用途に当社製品を使用したことによりお客様または第三者に生じた損害等に関し、当社は、一切その責任を負いません。なお、当社製品のデータ・シート、データ・ブック等の資料で特に品質水準の表示がない場合は、標準水準製品であることを表します。

   標準水準： コンピュータ、OA 機器、通信機器、計測機器、AV 機器、家電、工作機械、パーソナル機器、産業用ロボット

   高品質水準： 輸送機器（自動車、電車、船舶等）、交通用信号機器、防災・防犯装置、各種安全装置、生命維持を目的として設計されていない医療機器（厚生労働省定義の管理医療機器に相当）

   特定水準： 航空機器、航空宇宙機器、海底中継機器、原子力制御システム、生命維持のための医療機器（生命維持装置、人体に埋め込み使用するもの、治療行為（患部切り出し等）を行うもの、その他直接人命に影響を与えるもの）（厚生労働省定義の高度管理医療機器に相当）またはシステム等
8. 本資料に記載された当社製品のご使用につき、特に、最大定格、動作電源電圧範囲、放熱特性、実装条件その他諸条件につきましては、当社保証範囲内でご使用ください。当社保証範囲を超えて当社製品をご使用された場合の故障および事故につきましては、当社は、一切その責任を負いません。
9. 当社は、当社製品の品質および信頼性の向上に努めておりますが、半導体製品はある確率で故障が発生したり、使用条件によっては誤動作したりする場合があります。また、当社製品は耐放射線設計については行っておりません。当社製品の故障または誤動作が生じた場合も、人身事故、火災事故、社会的損害などを生じさせないようお客様の責任において冗長設計、延焼対策設計、誤動作防止設計等の安全設計およびエージング処理等、機器またはシステムとしての出荷保証をお願いいたします。特に、マイコンソフトウェアは、単独での検証は困難なため、お客様が製造された最終の機器・システムとしての安全検証をお願いいたします。
10. 当社製品の環境適合性等、詳細につきましては製品個別に必ず当社営業窓口までお問合せください。ご使用に際しては、特定の物質の含有・使用を規制する RoHS 指令等、適用される環境関連法令を十分調査のうえ、かかる法令に適合するようご使用ください。お客様がかかる法令を遵守しないことにより生じた損害に関して、当社は、一切その責任を負いません。
11. 本資料の全部または一部を当社の文書による事前の承諾を得ることなく転載または複製することを固くお断りいたします。
12. 本資料に関する詳細についてのお問い合わせその他お気付きの点等がございましたら当社営業窓口までご照会ください。

注 1. 本資料において使用されている「当社」とは、ルネサスエレクトロニクス株式会社およびルネサスエレクトロニクス株式会社がその総株主の議決権の過半数を直接または間接に保有する会社をいいます。

注 2. 本資料において使用されている「当社製品」とは、注 1 において定義された当社の開発、製造製品をいいます。

SuperH RISC engine C/C++ コンパイラパッケージ

アプリケーションノート：＜コンパイラ活用ガイド＞ 効率の良いプログラミング手法 編

本ドキュメントでは、SuperH RISC engine C/C++ コンパイラ V.9 における効率の良いプログラミング技法を紹介します。

目次

1. はじめに

SuperH RISC engine C/C++コンパイラは最適化を行っていますが、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。
本ドキュメントでは、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を紹介します。
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。
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。

実機での実行速度はコンパイラの生成コード以外にメモリアーキテクチャやキャッシュヒット率、割り込みなどの要因によって変化します。
本ドキュメントで紹介するさまざまな手法は実機で実際に実行し、効果を確認していってください。

本ドキュメントのアセンブリ言語展開コードは

shc△〈C 言語ファイル〉△-code＝asmcode△-cpu=sh2

のコマンドラインで取得しています。ただし、-cpu オプションにより、アセンブリ言語展開コードが SH-1、SH-2、SH-2E、SH-3、および SH-4 で異なる場合があります。なお、アセンブリ言語展開コードは今後のコンパイラ改善などにより変わる可能性があります。
コードサイズ、実行速度は、表 1-1に示すCPU オプションを用いています。他のオプションはデフォルトです。
(一部の技法では特定のオプションを指定しています)

表 1-1CPU オプション一覧

項番	CPU 種別	CPU オプション
1	SH-2	-cpu=sh2
2	SH-2A	-cpu=sh2a
3	SH-3	-cpu=sh3
4	SH-4A	-cpu=sh4a△-fpu=single

本ドキュメントに記載の実行速度は、コンパイラパッケージに付属のシミュレータデバッガを用いて測定しています。SH-2A、SH-3、SH-4A の測定に関しては、キャッシュミスを考慮していません。また、外部メモリへのアクセスサイクル数は１として測定しています。測定結果は参考値です。

効率の良いプログラミング技法の一覧を表 1-2に示します。

表 1-2 効率の良いプログラミング技法一覧

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）	○	-	○	2.1
2	大域変数（符号）	○	-	○	2.2
3	データの構造	○	-	○	2.3
4	データの整合	-	○	-	2.4
5	変数とconst 型	-	○	-	2.5
6	局所変数と大域変数	○	-	○	2.6
7	定数参照	○	-	-	2.7
8	定数除算最適化	×	-	○	2.8
9	構造体宣言のメンバオフセット	○	-	○	2.9
10	ビットフィールドの割り付け	○	-	-	2.10
11	ループ制御変数	×	-	○	2.11
12	関数のモジュール化	○	-	○	3.1
13	関数のインタフェース	-	○	○	3.2
14	ループ回数の削減	×	-	○	4.1
15	テーブルの活用	○	-	○	4.2
16	条件式	○	-	○	4.3
17	分岐	○	-	○	5

[注意] 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

2. データ指定

データに関して考慮すべき点を表 2-1に示します。

表 2-1 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化データの場合、const 演算子を付けておくと使用メモリ量の節約になります。	2.1 2.2 2.5
データの整合	• データ領域に無駄なエリアを生じないように割り付けてください。	2.4
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。	2.3
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。	-

2.1 局所変数（データサイズ）

■ポイント

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

■説明

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 50 までの総和を求めます。

改善前ソースコード

```
int f(void)
{
    char a = 50;
    int c = 0;
    for ( ; a > 0; a-- )
        c += a;
    return(c);
}
```

改善前アセンブリ展開コード

```
_f:
          MOV         #50,R2      ; H'00000032
          MOV         #0,R6       ; H'00000000
L11:
          ADD         R2,R6
          ADD         #-1,R2
          EXTS.B      R2,R2
          CMP/PL      R2
          BT          L11
          RTS
          MOV         R6,R0
```

改善後ソースコード

```
int f(void)
{
    long a = 50;
    int c = 0;
    for ( ; a > 0; a-- )
        c += a;
    return(c);
}
```

改善後アセンブリ展開コード

```
_f:
          MOV         #50,R2      ; H'00000032
          MOV         #0,R6       ; H'00000000
L11:
          ADD         R2,R6
          ADD         #-1,R2
          CMP/PL      R2
          BT          L11
          RTS
          MOV         R6,R0
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度[Cycle]	
	改善前	改善後	改善前	改善後
SH-2	18	16	353	303
SH-2A	16	14	302	252
SH-3	18	16	353	303
SH-4A	18	16	300	268

2.2 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

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、signed 型整数の方が unsigned 型整数よりも効率が良くなります。

なお、SH-2A、SH2A-FPU では、MOV + EXTU 命令の代わりに MOVU 命令を使用することがありますが、MOVU の命令サイズは 32 ビットのため、この場合も signed 型整数の方が unsigned 型整数よりも効率が良くなります。

■使用例

変数 c に変数 a と変数 b の和を代入します。

改善前ソースコード

```
unsigned short a;
unsigned short b;
int c;
void f(void)
{
    c = b + a;
}
```

改善前アセンブリ展開コード

```
_f:
          MOV.L       L11,R1
          MOV.L       L11+4,R2
          MOV.W       @R1,R5
          EXTU.W      R5,R4
          MOV.L       L11+8,R5
          MOV.W       @R5,R7
          EXTU.W      R7,R7
          ADD         R7,R4
          RTS
          MOV.L       R4,@R2
L11:
          .DATA.L     _b
          .DATA.L     _c
          .DATA.L     _a
```

改善後ソースコード

```
short a;
short b;
int c;
void f(void)
{
    c = b + a;
}
```

改善後アセンブリ展開コード

```
_f:
          MOV.L       L11,R1
          MOV.L       L11+4,R4
          MOV.W       @R1,R5
          MOV.W       @R4,R7
          MOV.L       L11+8,R2
          ADD         R7,R5
          RTS
          MOV.L       R5,@R2
L11:
          .DATA.L     _b
          .DATA.L     _a
          .DATA.L     _c
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度[Cycle]	
	改善前	改善後	改善前	改善後
SH-2	32	28	15	11
SH-2A	32	28	8	8
SH-3	32	28	15	11
SH-4A	32	28	16	10

2.3 データの構造

■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

関連するデータを同一関数の中で何度も参照している場合、構造体を用いると相対アクセスを利用したコードが生成され易くなり、効率向上が期待できます。また、引数として渡す場合も効率が向上します。相対アクセスにはアクセス範囲に制限があるため、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(void)
{
    a = 1;
    b = 2;
    c = 3;
}
```

改善後ソースコード

```
struct s{
    int a;
    int b;
    int c;
} s1;

void f(void)
{
    register struct s *p=&s1;

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

改善前アセンブリ展開コード

```
_f:
          MOV.L       L11,R7      ; _a
          MOV         #1,R1       ; H'00000001
          MOV.L       R1,@R7      ; a
          MOV.L       L11+4,R1    ; _b
          MOV.L       L11+8,R2    ; _c
          MOV         #2,R4       ; H'00000002
          MOV         #3,R5       ; H'00000003
          MOV.L       R4,@R1      ; b
          RTS
          MOV.L       R5,@R2      ; c
L11:
          .DATA.L     _a
          .DATA.L     _b
          .DATA.L     _c
```

改善後アセンブリ展開コード

```
_f:
          MOV.L       L11,R2      ; _s1
          MOV         #1,R1       ; H'00000001
          MOV         #2,R4       ; H'00000002
          MOV         #3,R5       ; H'00000003
          MOV.L       R1,@R2      ; (p)->a
          MOV.L       R4,@(4,R2)  ; (p)->b
          RTS
          MOV.L       R5,@(8,R2)  ; (p)->c
L11:
          .DATA.L     _s1
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度[Cycle]	
	改善前	改善後	改善前	改善後
SH-2	32	20	12	9
SH-2A	32	20	9	6
SH-3	32	20	14	10
SH-4A	32	20	10	8

2.4 データの整合

■ポイント

データの宣言順序を交換することにより、RAM 容量を削減できる場合があります。

■説明

大きさの異なる型の変数を宣言する場合は、同じ大きさの型の変数をまとめて宣言してください。これにより、データの整合によるデータ領域の空きが最小になります。

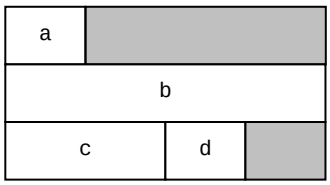
■使用例

全部で 8 バイトのデータを配置します。

改善前ソースコード

```
char a;
int b;
short c;
char d;
```

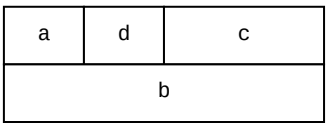
改善前データ配置



改善後ソースコード

```
char a;
char d;
short c;
int b;
```

改善後データ配置



2.5 変数と const 型

■ポイント

値を変更しない変数は、const 型で宣言してください。

■説明

初期化データは、通常、起動時に ROM エリアから RAM エリアに転送して、RAM エリアを使って処理を行います。このため、プログラム内で値が不変な初期化データの場合、確保した RAM エリアが無駄になります。初期化データに const 演算子を付けておくと、起動時の RAM エリアへの転送が抑止され、使用メモリ量の節約になります。

また、初期値は変更しない、というルールでプログラムを作成すると、ROM 化が容易になります。

■使用例

5 個の初期化データを設定します。

改善前ソースコード	改善後ソースコード

`char a[] =`
`  {1, 2, 3, 4, 5};`
 	
初期値をROMからRAMへ転送して処理を行います
	`const char a[] =`
	`  {1, 2, 3, 4, 5};`
	
	ROM 上の初期値を使用して処理を行います

2.6 局所変数と大域変数

■ポイント

一時変数、ループのカウンタなど、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、最適化の効率が悪くなります。

局所変数を使用すると次の利点があります。

a. アクセスコストが安い。
b. レジスタに割り付けられる可能性がある。
c. 最適化の効率が良い

■使用例

一時変数に大域変数を使った場合(改善前) と 局所変数を使った場合(改善後)。

改善前ソースコード

```
int tmp;

void f(int* a, int* b)
{
    tmp = *a;
    *a = *b;
    *b = tmp;
}
```

改善前アセンブリ展開コード

```
_f:
          MOV.L       @R4,R1      ; *(a)
          MOV.L       L11,R6      ; _tmp
          MOV.L       R1,@R6      ; tmp
          MOV.L       @R5,R7      ; *(b)
          MOV.L       R7,@R4      ; *(a)
          MOV.L       @R6,R2      ; tmp
          RTS
          MOV.L       R2,@R5      ; *(b)
L11:
          .DATA.L     _tmp
```

改善後ソースコード

```
void f(int* a, int* b)
{
    int tmp;

    tmp = *a;
    *a = *b;
    *b = tmp;
}
```

改善後アセンブリ展開コード

```
_f:
          MOV.L       @R4,R6      ; *(a)
          MOV.L       @R5,R2      ; *(b)
          MOV.L       R2,@R4      ; *(a)
          RTS
          MOV.L       R6,@R5      ; *(b)
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度[Cycle]	
	改善前	改善後	改善前	改善後
SH-2	20	10	12	7
SH-2A	20	10	10	6
SH-3	20	10	15	7
SH-4A	20	10	11	7

2.7 定数参照

■ポイント

定数値は、1 バイトで表現できるようにしておくとコードサイズを縮小できます。

■説明

2 バイトまたは 4 バイトの定数値を用いると、定数値がリテラルデータとしてメモリ上に確保され、そのデータを MOV 命令でレジスタに読み込むコードが生成されます。これに対し、1 バイトの定数値を使用すると、MOV 命令の中に定数データを埋め込むことができます。これにより、リテラルデータ読み込みのメモリアクセスが削減され、リテラルデータの分のコードサイズを削減することができます。

なお、SH-2A、SH2A-FPU では 20 ビット長までの定数値をコード内に埋め込むことが可能です。

const_load=inline オプションもしくは、speed オプションを指定することで、すべての 2 バイト定数および一部の 4 バイト定数を 1 バイト定数値から演算で求める命令へと展開することができます。この場合、コードサイズは大きくなりますが、メモリアクセスが削減できるため、実行速度の向上が期待できます。

■使用例

改善前ソースコード

```
#define   CODE  (567)

int data;
void f(void)
{
    data= CODE;
}
```

改善前アセンブリ展開コード

```
_f:
          MOV.L       L11+4,R6    ; _data
          MOV.W       L11,R2      ; H'0237
          RTS
          MOV.L       R2,@R6      ; data
L11:
          .DATA.W     H'0237
          .RES.W      1
          .DATA.L     _data
```

改善後ソースコード

```
#define   CODE  (123)

int data;
void f(void)
{
    data  = CODE;
}
```

改善後アセンブリ展開コード

```
_f:
          MOV.L       L11,R6      ; _data
          MOV         #123,R2     ; H'0000007B
          RTS
          MOV.L       R2,@R6      ; data
L11:
          .DATA.L     _data
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度[Cycle]	
	改善前	改善後	改善前	改善後
SH-2	14	12	5	5
SH-2A	14	12	4	4
SH-3	14	12	5	5
SH-4A	14	12	6	5

2.8 定数除算最適化

■ポイント

定数による除算は、除算以外の演算に展開する最適化を実施しています。したがって、できるだけ定数の除算を使用してください。

■説明

通常、変数同士の除算には、除算の実行時ルーチン、または、除算命令(SH-2A, SH2A-FPU)が使用されます。定数による除算に対しては、その逆数の近似値を乗算し、その結果を微調整する、という最適化を実施しています。これにより、除算の実行時ルーチン呼び出しもしくは除算命令に対して大幅に実行速度を改善することができます。

■使用例

以下の改善例では、除数を定数にすることにより、除算ルーチンを呼び出さずに直接 3 による商を求める命令列が生成されます。他の定数による除算に対しても同様のコードが生成されます。

改善前ソースコード

```
int x;
int z=3;
void f (int y){
    x=y/z;
}
```

改善前アセンブリ展開コード

```
_f:
          STS.L       PR,@-R15
          MOV.L       L11,R5      ; _z
          MOV.L       L11+4,R2    ; __divls
          MOV.L       @R5,R0      ; z
          MOV.L       L11+8,R6    ; _x
          JSR         @R2
          MOV         R4,R1
          LDS.L       @R15+,PR
          RTS
          MOV.L       R0,@R6      ; x
L11:
          .DATA.L     _z
          .DATA.L     __divls
          .DATA.L     _x
```

改善後ソースコード

```
int x;
void f (int y){
    x=y/3;
}
```

改善後アセンブリ展開コード

```
_f:
          STS.L       MACL,@-R15
          STS.L       MACH,@-R15
          MOV.L       L11,R1      ; H'55555556
          MOV.L       L11+4,R5    ; _x
          DMULS.L     R4,R1
          STS         MACH,R6
          MOV         R6,R0
          ROTL        R0
          AND         #1,R0
          ADD         R0,R6
          MOV.L       R6,@R5      ; x
          LDS.L       @R15+,MACH
          RTS
          LDS.L       @R15+,MACL
L11:
          .DATA.L     H'55555556
          .DATA.L     _x
```

【注】 この最適化は、スピードを大幅に改善しますが、展開したコードが大きくなる場合があるため、サイズ最適化のときは適用されません。

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度[Cycle]	
	改善前	改善後	改善前	改善後
SH-2	32	36	74	22
SH-2A	20	36	42	16
SH-3	32	36	76	24
SH-4A	32	36	77	19

[注] y=10000 の場合

2.9 構造体宣言のメンバオフセット

■ポイント

構造体の中でよく使用するメンバは、先頭に宣言するようにすればサイズ、スピードとも向上します。

■説明

構造体メンバは、構造体アドレスにオフセットを加算してアクセスします。オフセットが小さいほうがサイズ、スピードとも有利なので、よく使用するメンバを先頭に宣言するようにしてください。

もっとも効果的なのは、char, unsigned char 型で先頭から 16 byte 未満, short, unsigned short 型で先頭から 32 byte 未満, int, unsigned, long, unsigned long 型で先頭から 64 byte 未満です。

■使用例

以下の例は、構造体のオフセットによってコードが変わる例を示します。

改善前ソースコード

```
struct S{
    int a[100];
    int x;
};
int f(struct S *p){
    return p->x;
}
```

改善前アセンブリ展開コード

```
_f:
        MOV       #100,R0    ; H'00000064
        SHLL2     R0
        RTS
        MOV.L     @(R0,R4),R0; (p)->x
```

改善後ソースコード

```
struct S{
    int x;
    int a[100];
};
int f(struct S *p){
    return p->x;
}
```

改善後アセンブリ展開コード

```
_f:
        RTS
        MOV.L     @R4,R0     ; (p)->x
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度[Cycle]	
	改善前	改善後	改善前	改善後
SH-2	8	4	5	3
SH-2A	6	4	5	5
SH-3	8	4	5	3
SH-4A	8	4	6	5

2.10 ビットフィールドの割り付け

■ポイント

ビットフィールドで、同じ式で関連して参照されるものは、同じ構造体内に割り付けるようにしてください。

■説明

異なるビットフィールドのメンバを参照するためには、そのたびにビットフィールドを含むデータをロードしなければなりません。関連するビットフィールドを同じ構造体内にまとめて割り付けることによって、このロードを一度で済ますことができます。

■使用例

同じ構造体に関連するビットフィールドを割り付けることによってサイズが改善する例を示します。

改善前ソースコード

```
struct bits{
    unsigned int b0: 1;
} f1, f2;
int f(void){
    if (f1.b0 && f2.b0) return 1;
    else return 0;
}
```

改善前アセンブリ展開コード

```
_f:
          MOV.L       L15,R6      ; _f1
          MOV.B       @R6,R0      ; (part of)f1
          TST         #128,R0
          BT          L12
          MOV.L       L15+4,R6    ; _f2
          MOV.B       @R6,R0      ; (part of)f2
          TST         #128,R0
          BF          L13
L12:
          RTS
          MOV         #0,R0       ; H'00000000
L13:
          RTS
          MOV         #1,R0       ; H'00000001
L15:
          .DATA.L     _f1
          .DATA.L     _f2
```

改善後ソースコード

```
struct bits{
    unsigned int b0: 1;
    unsigned int b1: 1;
} f1;
int f(void){
    if (f1.b0 && f1.b1) return 1;
    else return 0;
}
```

改善後アセンブリ展開コード

```
_f:
          MOV.L       L11,R1      ; _f1
          MOV         #-64,R2     ; H'FFFFFFC0
          MOV.B       @R1,R0      ; (part of)f1
          EXTU.B      R2,R2
          AND         #192,R0
          CMP/EQ      R2,R0
          RTS
          MOVT        R0
L11:
          .DATA.L     _f1
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度[Cycle]	
	改善前	改善後	改善前	改善後
SH-2	32	20	11	9
SH-2A	32	24	12	12
SH-3	32	20	11	9
SH-4A	32	20	11	11

2.11 ループ制御変数

■ポイント

ループ制御変数を符号付き4バイト整数型(signed int/signed long)に変更すると、ループ展開最適化が適用され易くなり、実行速度の向上が期待できます。

■説明

speed または loop オプションを指定していても、ループ制御変数の型が以下の型の場合は、ループ展開最適化は適用されません。

- unsigned char
- unsigned short
- unsigned / signed long long

ループ制御変数がこれらの整数型以外の場合はループ展開最適化の対象となりますが、signed char、signed short、unsigned int、unsigned long に比べ、signed int/signed long の方がループ展開最適化の適用がされ易いです。ループ展開最適化を活用したい場合はループ制御変数を符号付き 4 バイト整数型としてください。

■使用例

改善前ソースコード

```
int ub;
char a[16];

void f2() {
    unsigned char i;

    for(i=0;i<ub;i++) {
        a[i]=0;
    }
}
```

改善前アセンブリ展開コード

```
<loopオプション指定時>
_f2:
          MOV.L       L14+2,R2    ; _ub
          MOV         #0,R6       ; H'00000000
          MOV.L       @R2,R5      ; ub
          BRA         L11
          MOV         R6,R4
L12:
          MOV.L       L14+6,R2    ; _a
          EXTU.B      R6,R0
          MOV.B       R4,@(R0,R2); a[]
          ADD         #1,R0
          MOV         R0,R6
L11:
          EXTU.B      R6,R2
          CMP/GE      R5,R2
          BF          L12
          RTS
          NOP
L14:
          .RES.W      1
          .DATA.L     _ub
          .DATA.L     _a
```

改善後ソースコード

```
int ub;
char a[16];

void f2() {
    int i;

    for(i=0;i<ub;i++) {
        a[i]=0;
    }
}
```

改善後アセンブリ展開コード

```
<loopオプション指定時>
_f2:
          MOV.L       L21+2,R2    ; _ub
          MOV.L       @R2,R4      ; ub
          MOV         R4,R5
          ADD         #-1,R5
          CMP/GT      R5,R4
          BF/S        L12
          MOV         #0,R6       ; H'00000000
          MOV.L       L21+6,R7    ; _a
          MOV         #0,R1       ; H'00000000
          BRA         L13
          MOV         R7,R2
L14:
          MOV         R1,R0
          MOV.B       R1,@R2      ; a[]
          MOV.B       R0,@(1,R2) ; a[]
          ADD         #2,R2
          ADD         #2,R6
L13:
          CMP/GE      R5,R6
          BF          L14
          CMP/GE      R4,R6
          BT          L17
          MOV         R6,R0
          RTS
          MOV.B       R1,@(R0,R7); a[]
L12:
          MOV.L       L21+6,R2    ; _a
          MOV         #0,R1       ; H'00000000
L19:
          CMP/GE      R4,R6
          BT          L17
          MOV.B       R1,@R2      ; a[]
          ADD         #1,R2
          BRA         L19
          ADD         #1,R6
L17:
```

```
                RTS
                NOP
L21:
                .RES.W      1
                .DATA.L     _ub
                .DATA.L     _a
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度[Cycle]	
	改善前	改善後	改善前	改善後
SH-2	38	74	204	104
SH-2A	36	72	155	77
SH-3	38	74	204	120
SH-4A	38	74	142	91

[注] ub=16 の場合

3. 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 3-1に示します。

表 3-1 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。	3.1
インタフェース	• 引数がすべ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。	3.2
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。	-

3.1 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。

さらに、同一ファイル内の関数呼び出しではインライン展開が可能となります。speed オプションもしくは inline オプションを指定すると自動インライン展開が行われ、高速なオブジェクトが生成されることもあります（プログラムサイズは増大する傾向にあります）。

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

```
#include <machine.h>
extern g(void);

int f(void)
{
    g();
    nop();
}
```

改善前アセンブリ展開コード

```
_f:
          STS.L       PR,@-R15
          MOV.L       L11,R2      ; _g
          JSR         @R2
          NOP
          NOP
          LDS.L       @R15+,PR
          RTS
          NOP
L11:
          .DATA.L     _g
```

改善後ソースコード

```
#include <machine.h>
int g(void)
{
}

int f(void)
{
     g();
     nop();
}
```

改善後アセンブリ展開コード

```
_g:
          RTS
          NOP
_f:
          STS.L       PR,@-R15
          BSR         _g
          NOP
          NOP
          LDS.L       @R15+,PR
          RTS
          NOP
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度[Cycle]	
	改善前	改善後	改善前	改善後
SH-2	20	14	15	13
SH-2A	16	12	15	12
SH-3	20	14	16	14
SH-4A	20	14	16	15

■補足

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。
このような場合、頻繁に呼び合う関数を BSR 命令で呼び出せる位置に置くことをお勧めします。

3.2 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。
コンパイラマニュアル 9.3.2 関数呼び出しのインタフェースを参照してください。

■説明

引数がすべ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。もし、構造体のポインタではなく、構造体そのものを受け渡すとレジスタに乗りません。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。

SH-2E では浮動小数点レジスタで単精度浮動小数点数を扱います。FR0～FR3 がワークレジスタ、FR4～FR11 が引数用、FR12～FR14 が局所変数用です。

また、SH2A-FPU、SH-4、SH-4A では、単精度/倍精度浮動小数点数を浮動小数点レジスタで扱えます。倍精度浮動小数点数を扱う場合は、DR4～DR10（４個）が引数用のレジスタとなります。

■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

```
int f(int, int, int, int, int);

void g(void)
{
    f(1, 2, 3, 4, 5);
}
```

改善前アセンブリ展開コード

```
_g:
          STS.L       PR,@-R15
          MOV         #5,R1       ; H'00000005
          MOV.L       R1,@-R15
          MOV.L       L11+2,R2    ; _f
          MOV         #4,R7       ; H'00000004
          MOV         #3,R6       ; H'00000003
          MOV         #2,R5       ; H'00000002
          JSR         @R2
          MOV         #1,R4       ; H'00000001
          ADD         #4,R15
          LDS.L       @R15+,PR
          RTS
          NOP
L11:
          .RES.W      1
          .DATA.L     _f
```

改善後ソースコード

```
struct b{
    int a, b, c, d, e;
} b1 = {1, 2, 3, 4, 5};

int f(struct b *p);

void g(void)
{
    f(&b1);
}
```

改善後アセンブリ展開コード

```
_g:
          MOV.L       L11,R4      ; _b1
          MOV.L       L11+4,R2    ; _f
          JMP         @R2
          NOP
L11:
          .DATA.L     _b1
          .DATA.L     _f
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度[Cycle]	
	改善前	改善後	改善前	改善後
SH-2	30	16	20	9
SH-2A	28	16	19	9
SH-3	30	16	22	9
SH-4A	30	16	20	12

4. 演算方法

演算方式に関して考慮すべき事項を表 4-1に示します。

表 4-1 演算方式における注意事項

項目	注意点	参照
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。	4.1
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てすむアルゴリズムを検討してください。	-
テーブルの活用	■ switch 文の各 case の処理がほぼ同じ場合は、テーブルを使用できないか検討してください。 ■ あらかじめ演算した結果をテーブルに代入しておき、演算結果が必要になった際、テーブルの値を参照することで実行速度を向上させる手法があります。ただし、この手法は、ROM 容量の増大になるので、必要実行速度と余裕 ROM 要領との兼ね合いで選択してください。	4.2
条件式	定数との比較は 0 で行うと効率の良いコードが生成されます。	4.3

4.1 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

```
extern int a[100];
void f(void)
{
    int i;
    for ( i = 0; i < 100; i++)
        a[i] = 0;
}
```

改善前アセンブリ展開コード

```
_f:
          MOV         #100,R6     ; H'00000064
          MOV.L       L13+2,R2    ; _a
          MOV         #0,R5       ; H'00000000
L11:
          DT          R6
          MOV.L       R5,@R2      ; a[]
          BF/S        L11
          ADD         #4,R2
          RTS
          NOP
L13:
          .RES.W      1
          .DATA.L     _a
```

改善後ソースコード

```
extern int a[100];
void f(void)
{
    int i;

    for ( i = 0; i < 100; i+=2)
    {
        a[i] = 0;
        a[i+1] = 0;
    }
}
```

改善後アセンブリ展開コード

```
_f:
          MOV         #50,R6      ; H'00000032
          MOV.L       L13,R2      ; _a
          MOV         #0,R5       ; H'00000000
L11:
          DT          R6
          MOV.L       R5,@R2      ; a[]
          MOV.L       R5,@(4,R2)  ; a[]
          BF/S        L11
          ADD         #8,R2
          RTS
          NOP
L13:
          .DATA.L     _a
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度[Cycle]	
	改善前	改善後	改善前	改善後
SH-2	22	24	506	356
SH-2A	20	22	403	253
SH-3	22	24	606	505
SH-4A	22	24	539	268

■補足

loop オプションを指定すると、ループ展開最適化が行われます。使用例の改善前ソースコードに loop オプションを指定しコンパイルすると、改善後ソースコードのアセンブリ展開コードと同じアセンブリ展開コードが出力されます。

改善前ソースコード（loopオプション指定あり）

```
void f(void)
{
    int i;
    for ( i = 0; i < 100; i++)
        a[i] = 0;
}
```

改善後ソースコード

```
extern int a[100];
void f(void)
{
    int i;

    for ( i = 0; i < 100; i+=2)
    {
        a[i] = 0;
        a[i+1] = 0;
    }
}
```

改善前アセンブリ展開コード

```
<-loop>
_f:
          MOV         #50,R6      ; H'00000032
          MOV.L       L13,R2      ; _a
          MOV         #0,R5       ; H'00000000
L11:
          DT          R6
          MOV.L       R5,@R2      ; a[]
          MOV.L       R5,@(4,R2) ; a[]
          BF/S        L11
          ADD         #8,R2
          RTS
          NOP
L13:
          .DATA.L     _a
```

改善後アセンブリ展開コード

```
_f:
          MOV         #50,R6      ; H'00000032
          MOV.L       L13,R2      ; _a
          MOV         #0,R5       ; H'00000000
L11:
          DT          R6
          MOV.L       R5,@R2      ; a[]
          MOV.L       R5,@(4,R2) ; a[]
          BF/S        L11
          ADD         #8,R2
          RTS
          NOP
L13:
          .DATA.L     _a
```

4.2 テーブルの活用

■ポイント

switch 文による分岐の代わりにテーブルを用いることで実行速度を向上できます。

■説明

switch 文の各 case の処理がほぼ同じ場合は、テーブルを使用できないか検討してください。

■使用例

変数 i の値により変数 ch に代入する文字定数を変えます。

改善前ソースコード

```
char f (int i)
{
    char ch;

    switch (i)
    {
    case 0:
        ch = 'a'; break;
    case 1:
        ch = 'x'; break;
    case 2:
        ch = 'b'; break;
    }
    return (ch);
}
```

改善前アセンブリ展開コード

```
_f:
          TST         R4,R4
          BT          L17
          MOV         R4,R0
          CMP/EQ      #1,R0
          BT          L19
          CMP/EQ      #2,R0
          BT          L20
          BRA         L21
          NOP
L17:
          BRA         L21
          MOV         #97,R2      ; H'00000061
L19:
          BRA         L21
          MOV         #120,R2     ; H'00000078
L20:
          MOV         #98,R2      ; H'00000062
L21:
          RTS
          MOV         R2,R0
```

改善後ソースコード

```
char chbuf[] = { 'a', 'x', 'b' };

char f(int i)
{
    return (chbuf[i]);
}
```

改善後アセンブリ展開コード

```
_f:
          MOV.L       L11,R6      ; _chbuf
          MOV         R4,R0
          RTS
          MOV.B       @(R0,R6),R0; chbuf[]
L11:
          .DATA.L     _chbuf
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度[Cycle]	
	改善前	改善後	改善前	改善後
SH-2	32	12	13	5
SH-2A	30	12	11	7
SH-3	32	12	13	5
SH-4A	32	12	18	5

[注] i=2 の場合

4.3 条件式

■ポイント

定数との比較は 0 で行うと効率の良いコードが生成されます。

■説明

0 との比較をする場合、定数値をロードする命令が生成されないので 0 以外との比較をする場合に比べ短いコードが生成されます。ループや if 文などの条件式は 0 との比較になるように設定してください。

■使用例

引数の値が 1 以上かどうかによりリターン値を変えます。

改善前ソースコード

```
int f (int x)
{
    if ( x >= 1 )
        return 1;
    else
        return 0;
}
```

改善前アセンブリ展開コード

```
_f:
          MOV         #1,R2       ; H'00000001
          CMP/GE      R2,R4
          RTS
          MOVT        R0
```

改善後ソースコード

```
int f (int x)
{
    if ( x > 0 )
        return 1;
    else
        return 0;
}
```

改善後アセンブリ展開コード

```
_f:
          CMP/PL      R4
          RTS
          MOVT        R0
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度[Cycle]	
	改善前	改善後	改善前	改善後
SH-2	8	6	5	4
SH-2A	8	6	6	5
SH-3	8	6	5	4
SH-4A	8	6	6	5

5. 分岐

分岐に関して考慮すべき事項を以下に示します。

- 同じ判定はまとめてください。
- switch 文、else if 文が長い場合、早く処理したいケースや頻繁に分岐するケースを先頭近くに置いてください。
- switch 文、else if 文が長い場合、段階を分けて判定することにより実行速度を向上できます。

■ポイント

case の数が 5～6 個までの switch 文は if 文にすると実行速度を向上できます。

■説明

case の数が少ない switch 文は if 文に置換してください。
switch 文は case 値のテーブルを引く前に変数の値の範囲をチェックするので、オーバヘッドがあります。
一方、if 文は何度も比較するので、場合分けが増えると効率が低下します。

switch 文のコード展開方式は case オプションで指定できます。case=ifthen を指定した場合、switch 文を if_then 方式で展開します。case=table を指定した場合、switch 文をテーブル方式で展開します。本オプションを省略した場合は、いずれかの展開方式をコンパイラが自動的に選択します。

■使用例

引数の値が 1 以上かどうかによりリターン値を変えます。

改善前ソースコード

```
int x(int a)
{
    switch (a)
    {
    case 1:
        a = 2; break;
    case 10:
        a = 4; break;
    default:
        a = 0; break;
    }
    return (a);
}
```

改善後ソースコード

```
int x (int a)
{
    if (a==1)
        a = 2;
    else if (a==10)
        a = 4;
    else
        a = 0;
    return (a);
}
```

改善前アセンブリ展開コード

```
_x:
          MOV         R4,R0
          CMP/EQ      #1,R0
          BT          L16
          CMP/EQ      #10,R0
          BT          L17
          BRA         L18
          NOP
L16:
          BRA         L19
          MOV         #2,R2       ; H'00000002
L17:
          BRA         L19
          MOV         #4,R2       ; H'00000004
L18:
          MOV         #0,R2       ; H'00000000
L19:
          RTS
          MOV         R2,R0
```

改善後アセンブリ展開コード

```
_x:
          MOV         R4,R0
          CMP/EQ      #1,R0
          BF          L12
          BRA         L13
          MOV         #2,R4       ; H'00000002
L12:
          CMP/EQ      #10,R0
          BF/S        L13
          MOV         #0,R4       ; H'00000000
          MOV         #4,R4       ; H'00000004
L13:
          RTS
          MOV         R4,R0
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度[Cycle]	
	改善前	改善後	改善前	改善後
SH-2	28	22	11	9
SH-2A	22	20	8	5
SH-3	28	22	11	9
SH-4A	28	22	20	10

[注] a=1 のとき

ホームページとサポート窓口<website and support,ws>

ルネサステクノロジホームページ

http://japan.renesas.com/

お問合せ先

http://japan.renesas.com/inquiry

csc@renesas.com

改訂記録<revision history,rh>

Rev.	発行日	改訂内容	
		ページ	ポイント
1.00	2007.6.1	—	初版発行

安全設計に関するお願い

1. 弊社は品質、信頼性の向上に努めておりますが、半導体製品は故障が発生したり、誤動作する場合があります。弊社の半導体製品の故障又は誤動作によって結果として、人身事故、火災事故、社会的損害などを生じさせないような安全性を考慮した冗長設計、延焼対策設計、誤動作防止設計などの安全設計に十分ご留意ください。

本資料ご利用に際しての留意事項

1. 本資料は、お客様が用途に応じた適切なルネサス テクノロジ製品をご購入いただくための参考資料であり、本資料中に記載の技術情報についてルネサス テクノロジが所有する知的財産権その他の権利の実施、使用を許諾するものではありません。
2. 本資料に記載の製品データ、図、表、プログラム、アルゴリズムその他応用回路例の使用に起因する損害、第三者所有の権利に対する侵害に関し、ルネサス テクノロジは責任を負いません。
3. 本資料に記載の製品データ、図、表、プログラム、アルゴリズムその他全ての情報は本資料発行時点のものであり、ルネサス テクノロジは、予告なしに、本資料に記載した製品または仕様を変更することがあります。ルネサス テクノロジ半導体製品のご購入に当たりましては、事前にルネサス テクノロジ、ルネサス販売または特約店へ最新の情報をご確認頂きますとともに、ルネサス テクノロジホームページ(http://www.renesas.com)などを通じて公開される情報に常にご注意ください。
4. 本資料に記載した情報は、正確を期すため、慎重に制作したものですが万一本資料の記述誤りに起因する損害がお客様に生じた場合には、ルネサス テクノロジはその責任を負いません。
5. 本資料に記載の製品データ、図、表に示す技術的な内容、プログラム及びアルゴリズムを流用する場合は、技術内容、プログラム、アルゴリズム単位で評価するだけでなく、システム全体で十分に評価し、お客様の責任において適用可否を判断してください。ルネサス テクノロジは、適用可否に対する責任は負いません。
6. 本資料に記載された製品は、人命にかかわるような状況の下で使用される機器あるいはシステムに用いられることを目的として設計、製造されたものではありません。本資料に記載の製品を運輸、移動体用、医療用、航空宇宙用、原子力制御用、海底中継用機器あるいはシステムなど、特殊用途へのご利用をご検討の際には、ルネサス テクノロジ、ルネサス販売または特約店へご照会ください。
7. 本資料の転載、複製については、文書によるルネサス テクノロジの事前の承諾が必要です。
8. 本資料に関し詳細についてのお問い合わせ、その他お気付きの点がございましたらルネサス テクノロジ、ルネサス販売または特約店までご照会ください。