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＞ SH-2A,SH2A-FPU 編

本ドキュメントでは、SuperH RISC engine C/C++ コンパイラ V.9.01 における、SH-2A / SH2A-FPU 向けコーディングテクニック、C 言語拡張機能 及び コンパイルオプションの使い方を紹介します。

1. SH-2A / SH2A-FPU 概要

SH-2A / SH2A-FPU(以後、SH-2A と略します) は、SH-1、SH-2 とオブジェクトコードレベルで上位互換性を持つ、新しい SH マイコンです。 SH-2A は、

- 新規命令追加による演算処理性能の向上とプログラムサイズの削減
- レジスタバンク採用による割り込み応答時間の短縮
- ジャンプテーブルベースレジスタ(TBR) によるサブルーチンコールの高速化
- 2 ウェイスパースカラによる 2 命令同時実行
- ハーバードアーキテクチャーの導入

という特徴を持っています。本書では、これらの特徴を SuperH RISC engine C/C++ コンパイラで活用する方法を紹介します。

2. SH-2A 新規命令の特徴と活用方法

SH-2A の新規命令の特徴を表 1.1 に示します。

表 1.1 SH-2A 新規命令の特徴

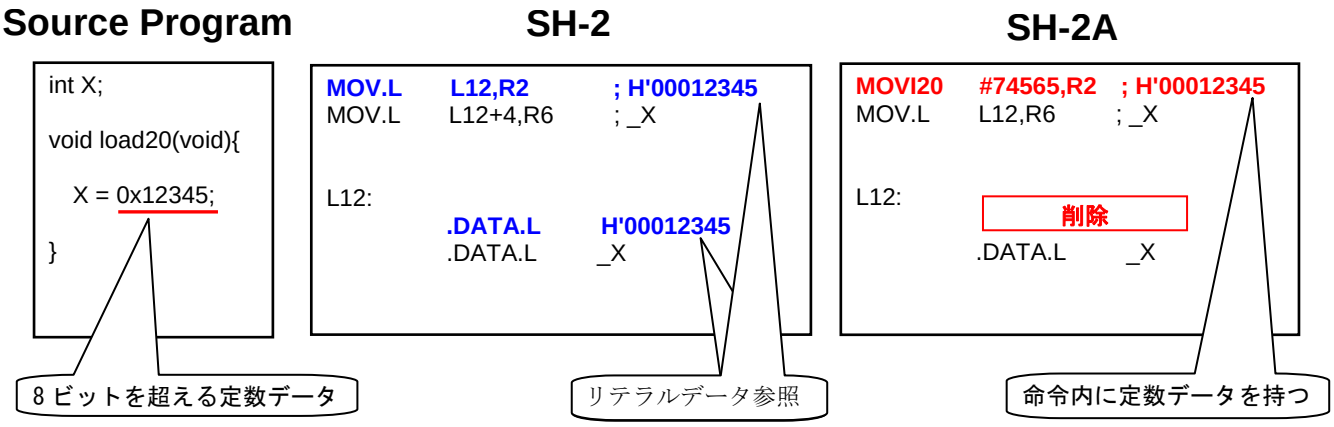
項目	内容	効果	
		スピード	サイズ
20 ビット長即値ロード (32 ビット固定長命令)	命令コード内の 20 ビットイミディエイトデータをレジスタに転送する命令。アドレスロード、定数ロードで効果を発揮。	Up Memory access reduction	Small
12 ビット disp 付レジスタ相対ロード・ストア (32 ビット固定長命令)	12 ビットのディスプレースメントを指定してメモリを参照する命令。構造体のアクセス効率を向上させる。	Up	Small
ロードマルチ/ストアマルチ	複数の連続したレジスタを 1 命令でメモリに退避、回復する命令。レジスタ退避回復処理のコードサイズを削減する。	Even	Small
オートインクリメント/ デクリメント	ポインタの自動インクリメント/デクリメントを行う。配列の連続アクセスなどで活用される。	Up	Small
除算命令	32 ビット÷32 ビットの除算を行う命令。命令で除算が行える為、実行時ルーチンの呼び出しを削除できる。	Up	Small
乗算命令	32 ビット×32 ビットの除算を行い、演算結果の下位 32 ビットを汎用レジスタ Rn に格納する命令。MAC レジスタへのアクセスを削減。	Up	Small
飽和値比較命令	飽和値との比較を行い、飽和値を上回る時は飽和上限値を、下回る場合は飽和下限値を返す命令。オーバーフロー、アンダーフローの判定処理の効率を大幅に向上させる。	Up	Small
ビット操作命令 (32 ビット固定長命令)	メモリ中の 1 ビットに関する各種操作を 1 命令で実行。 (論理演算、操作、取得、反転取得、 反転論理演算)	Up	Small
遅延スロット無し分岐命令	遅延スロットを持たない分岐命令。 不要な NOP 命令を削除することでコードサイズを削減。	Even	Small
バレルシフト命令	任意のビットをシフトする命令。算術シフトと論理シフトの 2 種類の命令がある。	Up	Small

2.1 20 ビット長即値ロード

＜書式＞ MOVI20 #imm20,Rn 、MOVI20S #imm20,Rn

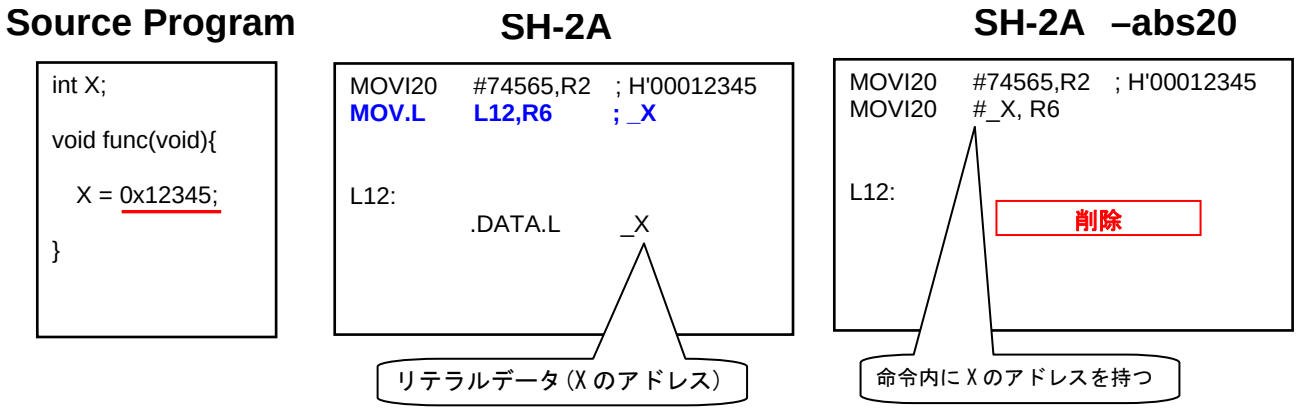
20 ビットイミディエイトデータを転送する命令です。 イミディエイトデータとして、定数や関数のアドレスなどが格納されます。 例えば、従来の SH-2 では 8 ビットを超える定数を扱う時は、リテラルデータを用意する必要があります。 SH-2A では 20 ビットまでのイミディエイトデータを命令に埋め込むことが出来るので、コードサイズの削減とメモリアクセスを削減することができます。尚、本命令は 32 ビット長命令です。

＜例 2-1(1)＞



また、関数・変数のアドレスも 32 ビットであるため、リテラルデータで用意されます。例 2-1(1) では、外部変数 X のアドレスが SH-2,SH-2A のどちらの場合もリテラルデータで用意されています。ここで、関数・変数のアドレスを 20 ビット表現に変える、#pragma abs20 もしくは コンパイルオプション：-abs20 を指定すると、20 ビット即値ロード命令を活用したコードが生成されます。

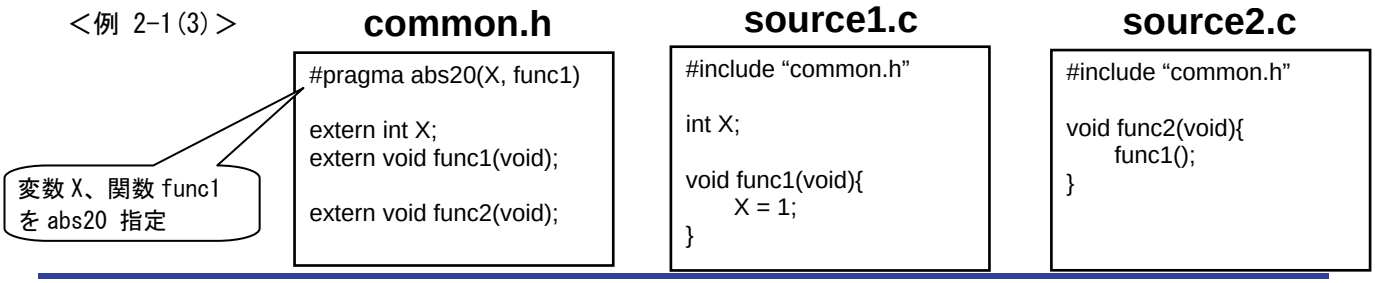
＜例 2-1(2)＞



[#pragma abs20 の指定方法]

変数、関数の宣言/定義の前に、#pragma abs20 <識別子>[,…] を記述すると、指定された変数、関数のアドレス表現が 20 ビット表現となります。#pragma abs20 は当該変数、関数を参照するすべてのソースコードで指定されている必要があります。 共通でインクルードするヘッダファイルに#pragma asb20 を記述することをお勧めします。

＜例 2-1(3)＞



[-abs20 について]

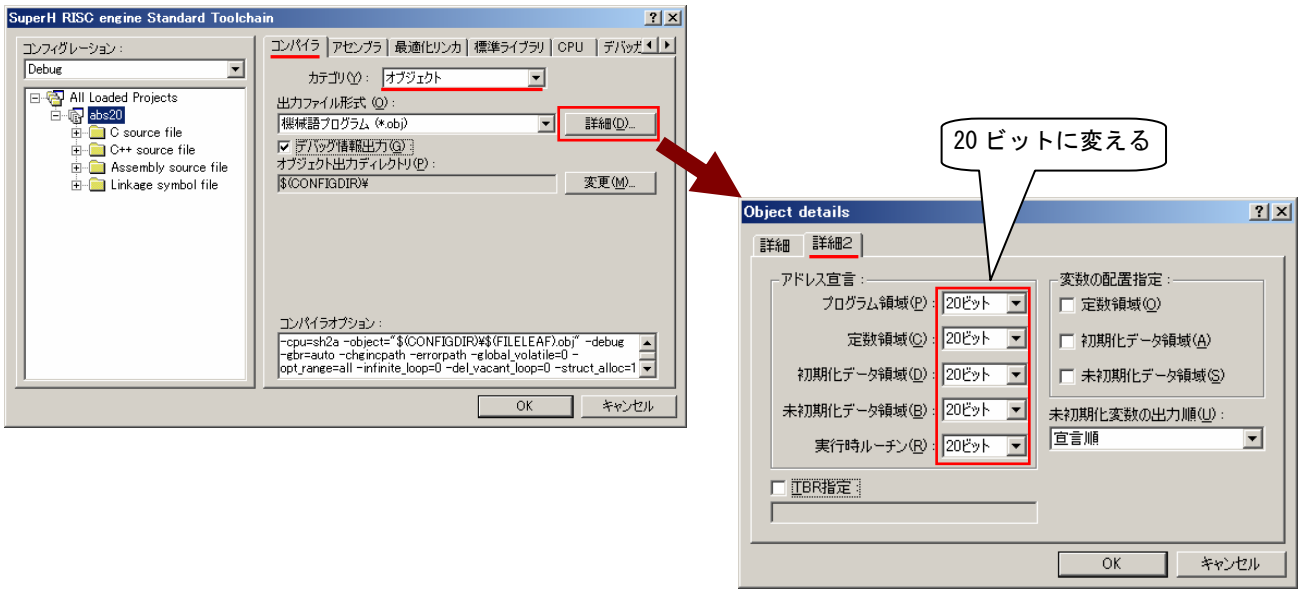
コンパイルオプション”-abs20” を指定すると、当該ソースファイルのすべての関数、変数に対して、アドレス表現を 20 ビット表現に変更することが出来ます。-abs20 オプションには、次のサブコマンドが存在します。

＜書式＞ -abs20 = { Program | Const | Data | Bss | Run | All }

Program ： プログラム領域を対象にする
Const ： 定数領域を対象にする
data ： 初期化データ領域を対象にする
bss ： 未初期化データ領域を対象にする
run ： 実行時ルーチンを対象にする
all ： すべての領域を対象にする

[-abs20 を High-performance Embedded Workshop（以下、HEW と略します）で指定する方法]

ツールチェインダイアログより、-abs20 の指定が行えます。



[#pragma abs20, -abs20 使用上の注意点]

#pragma abs20, -abs20 が指定された関数・変数は、次のアドレス空間の範囲に配置されている必要があります。
範囲に収まっていない場合は、プログラムが正常に動作しませんのでご注意ください。

表. 1-1 20 ビットアドレス表現の範囲

アドレス範囲	
下限	上限
0x00000000	0x0007FFFF
0xFFF80000	0xFFFFFFFF

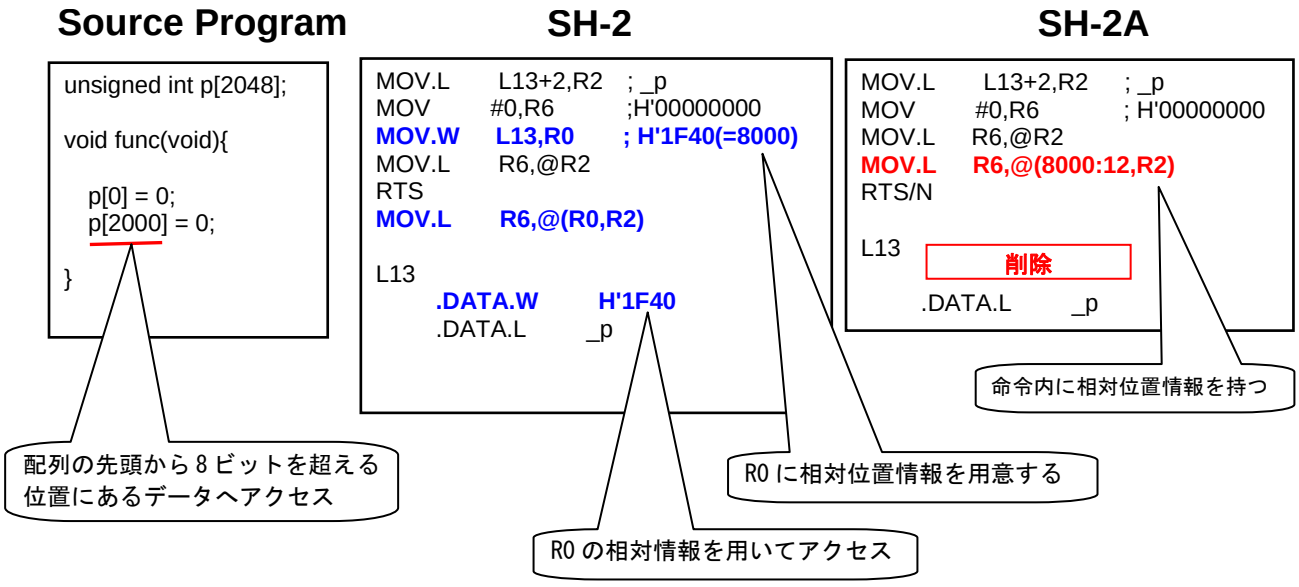
尚、-abs20 と#pragma abs16|abs20|abs28|abs32 が同時に指定された場合は#pragma 指定が有効になります。

2.2 12 ビット disp 付レジスタ相対ロード・ストア

＜書式＞ MOV.B Rm,@(disp 12,Rn) 、MOV.W Rm,@(disp 12,Rn) 、MOV.L Rm,@(disp 12,Rn)
MOV.B @(disp 12,Rn),Rm 、MOV.W @(disp 12,Rn),Rm 、MOV.L @(disp 12,Rn),Rm

12 ビット disp を持つ転送命令です。 配列、構造体、スタック内のデータアクセスなどで活用されます。SH-2 では 8 ビット disp 付レジスタ相対アクセス命令しか持っていませんので、先頭から 8 ビットを超える位置にある配列データなどにアクセスには、(多くの場合) オフセット値をリテラルデータで用意するコードが生成されます。SH-2A では、命令内に 12 ビットまでのオフセット値を埋め込むことが可能ですので、SH-2 よりも広い範囲のデータアクセスを効率よく行えます。尚、本命令は 32 ビット長命令です。

＜例 2-2＞



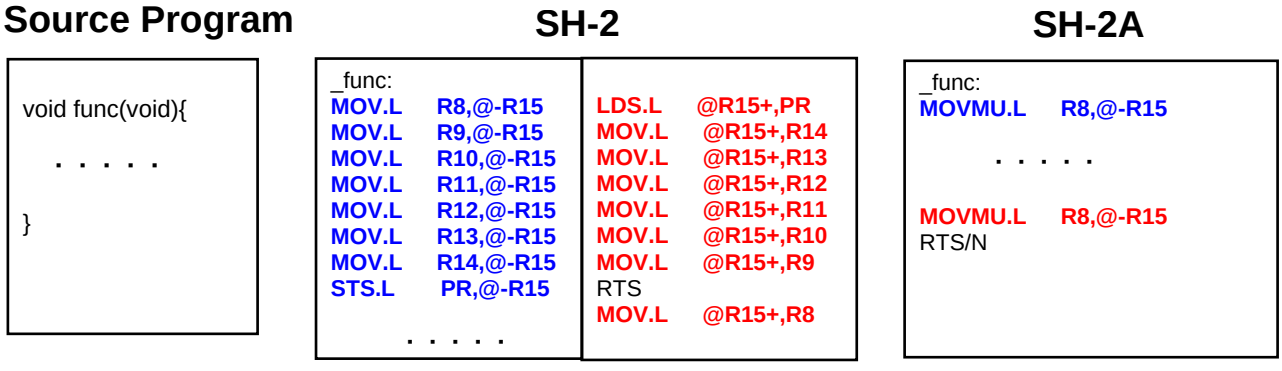
本命令を活用するため、構造体、配列を使用する時は 12 ビット disp が活用できるサイズ(配列であれば要素数 4095 以下)に収めるよう心掛けてください。尚、本命令を使用するための特別な#pragma 指定などは不要です。

2.3 ロードマルチ / ストアマルチ

＜書式＞ MOVMUL.L Rm,@-R15 、MOVMUL.L @R15+,Rn
MOVMU.L Rm,@-R15 、MOVMU.L @R15+,Rn

複数レジスタのロード、ストアを 1 命令で実行する命令です。 命令実行ステート数は 1 レジスタずつ指定した場合と変わりありませんが、1 命令で指定できますので命令サイズを削減することが出来ます。関数呼び出し時のレジスタ退避回復などで活用されます。本命例はコンパイラが自動的に活用します。特にコーディング上の注意点などはありません。

＜例 2-3＞



2.4 オートインクリメント / デクリメント

＜書式＞ ① MOV.B R0,@Rn+ 、MOV.W R0,@Rn+ 、MOV.L R0,@Rn+
② MOV.B @-Rn,R0 、MOV.W @-Rn,R0 、MOV.L @-Rn,R0

① R0 レジスタを Rn レジスタが示すアドレスへ転送し、Rn レジスタの値を.B のときは１、.W のときは２、.L のときは 4 インクリメントする。
② Rm レジスタが示すアドレスのデータを R0 レジスタへ転送し、Rm レジスタの値を.B のときは１、.W のときは２、.L のときは 4 デクリメントする。

配列要素に連続してアクセスするときなどに活用されます。本命例はコンパイラが自動的に活用します。特にコーディング上の注意点などはありません。

＜例 2-4＞

Source Program

```
int array[20];

void gunc(int s[20]){
   int I;

   for(i=0, i<20, i++){
      array[i] = s[i];
   }
}
```

SH-2

```
_func:
   MOV     #20,R5        ; H'00000014
   MOV.L   L17,R6        ; _aray

L15:
   MOV.L   @R4+,R2
   DT      R5
   MOV.L   R2,@R6
   BF/S    L15
   ADD     #4,R6
   RTS
   NOP

L17
   .DATA.L       _aray
```

SH-2A -abs20

```
_func:
   MOV      #20,R5     ; H'00000014
   MOVI20   #_aray,R6

L14:
   MOV.L    @R4+,R0
   DT       R5
   BF/S     L14
   MOV.L    R0,@R6+
   RTS/N
```

2.5 除算命令

＜書式＞ DIVS R0,Rn ＜実行ステート数：36＞ 、DIVU R0,Rn ＜実行ステート数：34＞

Rn レジスタの 32 ビットの内容(被除数)を RO の内容(除数)で除算する命令です。SH-2 で 32 ビットの除算を行うときは、1 ビット除算命令を複数回呼び出して演算する必要があります。コンパイラはこの 1 ビット除算命令の数回呼び出しによる除算処理を実行時ルーチン(ランタイムライブラリ)で提供しています。SH-2A では 32 ビット除算命令がサポートされていますので、実行時ルーチンではなく、命令で除算処理を行います。これにより、実行速度の向上とプログラムサイズの削減が可能です。

＜例 2-5(1)＞

Source Program

```
int A;

void func(int d){

   A   =   A / d;

}
```

SH-2

```
_func:
   STS.L     PR,@-R15
   MOV.L     L18,R6   ; _A
   MOV.L     L18+4,R2   ; __divls
   MOV.L     @R6,R1
   JSR       @R2
   MOV       R4,R0
   LDS.L     @R15+,PR
   RTS
   MOV.L     R0,@R6

L18
   .DATA.L       _A
   .DATA.L       __divls
```

実行時ルーチンの呼び出し

SH-2A -abs20

```
_func:
   MOVI20    #_A,R6
   MOV.L     @R6,R2
   MOV       R4,R0
   DIVS      R0,R2
   RTS
   MOV.L     R2,@R6
```

［ 整数型定数による除算 ］

除数が整数型定数の場合、除算を乗算として処理する最適化が実施可能です。SH-2A では除算命令がサポートされていますが、除算命令の実行ステート数は 34 もしくは 36 ステートであるため、乗算で処理した方が高速に処理できます。ただし、命令数は多くなるため、プログラムサイズは増加します。除数を逆数の乗算として処理する最適化は、コンパイルオプション”DIvision=cpu= Inline | Runtime” で指定できます。また、スピード・サイズ最適化の指定で、デフォルトの設定が異なります。スピード優先、サイズ&スピード を指定している場合は、デフォルトで本最適化が適用されます。サイズ優先を指定している場合は、デフォルトでは本最適化は実施されません(表.2-5)。 尚、SHC コンパイラ V.9.00 では、SH-2A に対して定数除算を乗算として処理する最適化を使用できません。常に除算命令を用いた演算が行われます。

表.2-5 定数除算の演算方式

CPU	SH-2	SH-2A (SHC V.9.00)	SH-2A (SHC V.9.01)
サイズ優先(-size)	Runtime	Runtime	Runtime
サイズ&スピード	Inline	Runtime	Inline
スピード優先(-speed)	Inline	Runtime	Inline

＜SH-2＞
Inline ：定数除算を乗算に変換し演算します。変数除算は実行時ルーチンを呼び出します。
Runtime ：シフト演算で行えない定数除算、変数除算は実行時ルーチンを呼び出します。
＜SH-2A＞
Inline ：定数除算を乗算に変換し演算します。変数除算は除算命令を用います。
Runtime ：シフト演算で行えない定数除算、変数除算は除算命令を用います。

＜例 2-5(2)＞

Source Program

```
unsigned int A;

void func(void){

   A   =   A / 10;

}
```

SH-2A -size -abs20

```
_func:
      MOVI20   #_A,R5
      MOV.L    @R5,R6
      MOV      #10,R0      ;H'0000000A
      DIVU     R0,R6
      RTS
      MOV.L    R6,@R5
```

SH-2A -speed -abs20 -macsave=0

```
_func:
   MOVI20    #_A,R6
   MOV.L     L12+4,R1   ; H'CCCCCCCD
   MOV.L     @R5,R6     ; A
   DMULU.L   R6,R1
   STS       MACH,R2
   MOV.L     R2,@R6
   SHLR2     R2
   SHLR      R2
   RTS
   MOV.L     R2,@R5     ; A

L12:
   .DATA.L   H'CCCCCCCD
```

2.6 乗算命令

＜書式＞ MULR R0,Rn

汎用レジスタ R0 の内容と Rn を 32 ビットで乗算し、結果の下位側 32 ビットを汎用レジスタ Rn に格納する命令です。SH-2 からサポートされている乗算命令(MUL.L Rm,Rn) では、演算結果を MACL レジスタに格納する為、MACL から汎用レジスタに結果をコピーするコードが必要です。MULR 命令を活用すると、このコピー処理が不要となり、速度とコードサイズの両方で性能向上が期待できます。本命例はコンパイラが自動的に活用します。特にコーディング上の注意点などはありません。

＜例 2-6＞

Source Program

```
int A;

void func(int m){

   A   =   A / m;

}
```

SH-2

```
_func:
   STS.L      MACL,@-R15
   MOV.L      L18,R6   ; A
   MOV.L      @R6,R1
   MUL.L      R1,R4
   STS        MACL,R5
   MOV        R5,@R6
   RTS
   LDS.L      @R15+,MACL

L18
   .DATA.L       _A
```

SH-2A -abs20

```
_func:
   MOVI20     #_A,R5
   MOV.L      @R5,R6
   MOV        R6,R0
   MULR       R0,R6
   RTS
   MOV.L      R6,@R5
```

2.7 飽和値比較命令

＜書式＞ CLIPS.B Rn 、CLIPS.W Rn 、CLIPU.B Rn 、CLIPU.W Rn

飽和を判定する命令です。汎用レジスタ Rn が飽和上限値を上回る場合は飽和上限値を、飽和下限値を下回るときは飽和下限値を Rn に格納し、CS ビットを 1 にセットします。本命令を活用することにより、飽和判定の処理が大幅に高速化できます。 本命令を活用するには、組み込み関数を使用してください。

組み込み関数	説明	下限値	上限値
long clipsb(long data)	データが-128～127 の範囲内の場合はその値を、 範囲外の場合は上限値もしくは下限値を返します。	-128	127
long clipsw(long data)	データが-32768～32767 の範囲内の場合はその値を、 範囲外の場合は上限値もしくは下限値を返します。	-32768	32767
unsigned long clipub (unsigned long data)	データが 0～255 の範囲内の場合はその値を、 範囲外の場合は上限値もしくは下限値を返します。	0	255
unsigned long clipuw (unsigned long data)	データが 0～65535 の範囲内の場合はその値を、 範囲外の場合は上限値もしくは下限値を返します。	0	65535

＜例 2-7＞

SH-2

＜Cソースコード＞

```
unsigned long result,x,y;

void func(void){

    result = x * y;
    if( result >255)
        result = 255;
}
```

＜生成コード＞

```
   STS.L       MACL,@-R15
   MOV.L       L23+44,R1
   MOV         #1,R6
   MOV.L       @R1,R2
   MOV.L       L23+48,R1
   MOV.L       L23+52,R5
   MOV.L       @R1,R4
   SHLL8       R6
   MUL.L       R2,R4
   STS         MACL,R7
   CMP/HI      R6,R7
   BF/S        L19
   MOV.L       R7,@R5
   MOV         #-1,R2
   EXTU.B      R2,R2
   MOV.L       R2,@R5
L19:
   RTS
   LDS.L       @R15+,MACL
```

SH-2A (組み込み関数使用)

＜Cソースコード＞

```
#include <machine.h>
unsigned long result,x,y;

void func(void){

    result = clipub( x * y );
}
```

＜生成コード＞

```
   MOVI20      #_x,R1
   MOVI20      #_y,R4
   MOV.L       @R1,R7
   MOV.L       @R4,R0
   MOV.L       L19+16,R2   ; _result
   MULR        R0,R7
   CLIPU.B     R7
   RTS
   MOV.L       R7,@R2
```

2.8 ビット操作命令

＜書式＞

ビット論理積	: BAND.B #imm3,@(disp12,Rn)	＜実行ステート数：3＞
ビットノット論理積	: BANDNOT.B #imm3,@(disp12,Rn)	＜実行ステート数：3＞
ビットクリア	: BCLR.B #imm3,@(disp12,Rn)	＜実行ステート数：3＞
(16 ビット命令)	: BCLR #imm3,Rn	＜実行ステート数：1＞
ビットロード	: BLD.B #imm3,@(disp12,Rn)	＜実行ステート数：3＞
(16 ビット命令)	: BLD #imm3,Rn	＜実行ステート数：1＞
ビットノットロード	: BLDNOT.B #imm3,@(disp12,Rn)	＜実行ステート数：3＞
ビット論理和	: BOR.B #imm3,@(disp12,Rn)	＜実行ステート数：3＞
ビットノット論理和	: BORNOT.B #imm3,@(disp12,Rn)	＜実行ステート数：3＞
ビットセット	: BSET.B #imm3,@(disp12,Rn)	＜実行ステート数：3＞
(16 ビット命令)	: BSET #imm3,Rn	＜実行ステート数：1＞
ビットストア	: BST.B #imm3,@(disp12,Rn)	＜実行ステート数：3＞
(16 ビット命令)	: BST #imm3,Rn	＜実行ステート数：1＞
ビット排他的論理和	: XBOR.B #imm3,@(disp12,Rn)	＜実行ステート数：3＞

ビット操作演算を 1 命令で実行する命令です。(disp12+Rn) が示すアドレスのメモリ中の imm3 で指定された 1 ビットに対して、それぞれの演算を行います。これらの命令は 32 ビット命令です(一部 16 ビット命令あり)。

SH-2 では、ビット操作を行う時は、次の 3 つの命令を実行します。

1．対象となるアドレス領域のメモリ値をリードする
2．AND 命令 もしくは OR 命令で読み出したメモリ値をビット操作した値を求める
3．演算により求めたメモリ値を対象となるアドレス領域にライトする

この 3 つ命令を実行している途中で割り込みが発生し、割り込み処理の中でビット操作の対象となるアドレス領域の値を変更した場合は、このビット操作処理の結果は不正となってしまいます。そのため、割り込み関数の中で書き換えられる可能性があるアドレス領域をビット操作する場合は、割り込みを禁止してから処理する必要があります。しかし、SH-2A で追加されたビット操作命令を活用すると、1 命令でビット操作を行うことが出来る為、割り込みを禁止する必要がなくなります。これにより、処理速度と割り込み応答性を向上させることが出来ます。

本命例は最適化によって生成されます。その為、最適化を実施しない場合(optimize=0)はビット操作命令が使用されませんので、ご注意ください。また、ビット操作命令はアクセス幅が 8 ビットです。そのため、volatile 宣言されている変数ではアクセス幅が signed / unsigned char 型 以外の場合はビット操作命令が使用されません。この点もご注意ください。

＜例 2-8＞

Source Program

```
typedef union {
   unsigned char BYTE;
      struct{
         unsigned char        B0:1;
         unsigned char        B1:1;
         unsigned char        B2:1;
         unsigned char        B3:1;
         unsigned short       W4:1;
         unsigned short       W5:1;
         unsigned long        L6:1;
         unsigned long        L7:1;
      }BIT;
}REG;
```

```
volatile REG Reg;
void func1(void){
      Reg.BIT.B2 = 1;
}
void func2(void){
      Reg.BIT.W4 = 1;
}
REG Reg2;
void func3(void){
      Reg2.BIT.W4 = 1;
}
```

SH-2

```
_func1:
  MOV.L       L13+4,R6      ;Reg
  MOV.B       @R6,R0
  OR          #32,R0
  RTS
  MOV.B       R0,@R6

_func2:
  MOV.L       L13+4,R6      ;Reg
  MOV.W       L13,R5        ; H'8000
  MOV.W       @(2,R6),R0
  OR          R5,R0
  RTS
  MOV.W       R0,@(2,R6)

_func3:
  MOV.L       L13+8,R6      ;Reg2
  MOV.B       @(2,R6),R0
  OR          #128,R0
  RTS
  MOV.B       R0,@(2,R6)

L13:
            .DATA.W     H'8000
            .RES.W      1
            .DATA.L     _Reg
            .DATA.L     _Reg2
```

SH-2A -abs20

```
_func1:
  MOVI20      #_Reg,R2
  BSET.B      #5,@(0,R2)
  RTS/N

_func2:
  MOVI20      #_Reg,R6
  MOV.W       @(2,R6),R0
  MOVI20      #32768,R5   ; H'00008000
  OR          R5,R0
  RTS
  MOV.W       R0,@(2,R6)

_func3:
  MOVI20      #_Reg2,R2
  BSET.B      #7,@(2,R2)
  RTS/N
```

ビット操作命令

volatile 宣言されている為
アクセス幅が保証され、
ビット操作命令は
使用されません。

volatile 宣言が無く
アクセス幅の保証が不要
の為、ビット操作命令が
使用されます。

2.9 遅延スロットなし分岐命令

＜書式＞ RTS/N 、RTV/N Rm

遅延スロットが無い、サブルーチンプロシージャからの復帰命令です。 SH-2 では、遅延スロットがあるサブルーチンプロシージャからの復帰命令しかないため、遅延スロットに入れるべき処理が無い時は NOP を遅延スロットに入れていました。RTS/N を用いると、NOP 命令を削除できる為、コードサイズを削減できます。また、SHC コンパイラでは関数の戻り値は R0 レジスタで戻すルールとなっていますが、RTV/N 命令を用いると R0 レジスタへのデータの移動をサブルーチンプロシージャからの復帰命令に含めることが出来る為、コードサイズを削減できます。本命例はコンパイラが自動的に活用します。特にコーディング上の注意点などはありません。

＜例 2-9＞

Source Program

```
int func(int a){

   return a;

}
```

SH-2

```
_func:
   RTS
   MOV.L        R4,R0
```

SH-2A -abs20

```
_func:
   RTV/N        R4
```

2.10 バレルシフト命令

＜書式＞ SHAD Rm,Rn 、SHLD Rm,Rn

汎用レジスタ Rn の内容を算術的(SHAD)、論理的(SHLD)にシフトします。Rm がシフト方向とシフトするビット数を表します。SH-2 では、バレルシフト命令がないため、シフト数が未定の場合は実行時ルーチンを、シフト数が定数の場合は、固定数シフト命令を組み合わせてシフト演算を行います。バレルシフト命令を用いると、実行時ルーチンの呼び出しや固定数シフト命令の複数回呼び出しが行われなくなり、サイズ、速度共に効率向上が期待できます。本命例はコンパイラが自動的に活用します。特にコーディング上の注意点などはありません。

＜例 2-10(1)＞

Source Program

```
int A;

int func(int s){

   A = A << s;

}
```

SH-2

```
_func:
   STS.L        PR,@-R15
   MOV.L        L24,R5      ; _A
   MOV.L        L24+4,R2    ; __sftl
   MOV.L        @R5,R0
   JSR          @R2
   MOV          R4,R1
   LDS.L        @R15+,PR
   RTS
   MOV.L        R0,@R5

L24
   .DATA.L      _A
   .DATA.L      __sftl
```

実行時ルーチン呼び出し

SH-2A -abs20

```
_func:
   MOVI20       #_A,R6
   MOV.L        @R6,R2
   SHAD         R4,R2
   RTS
   MOV.L        R2,@R6
```

バレルシフト命令

＜例 2-10(2)＞

Source Program

```
int A;

int func(int s){

   A = A << 14;

}
```

SH-2

```
_func:
   MOV.L       L25,R5   ; _A
   MOV.L       @R5,R6
   SHLL8       R6
   SHLL2       R6
   SHLL2       R6
   SHLL2       R6
   RTS
   MOV.L       R6,@R5

L25
   .DATA.L     _A
```

SH-2A -abs20

```
_func:
   MOVI20      #_A,R5
   MOV.L       @R5,R2
   MOV         #14,R6   ; H'0000000E
   SHAD        R6,R2
   RTS
   MOV.L       R2,@R5
```

3. SH-2A 新規アーキテクチャーの特徴と活用方法

SH-2A の新規アーキテクチャーであるレジスタバンクと TBR の活用方法を紹介します。

3.1 レジスタバンク

SH-2A には、割り込み処理に伴うレジスタの退避、回復を高速に行う為のレジスタバンクを内蔵しています。レジスタバンクの構成を図 3-1 に示します。

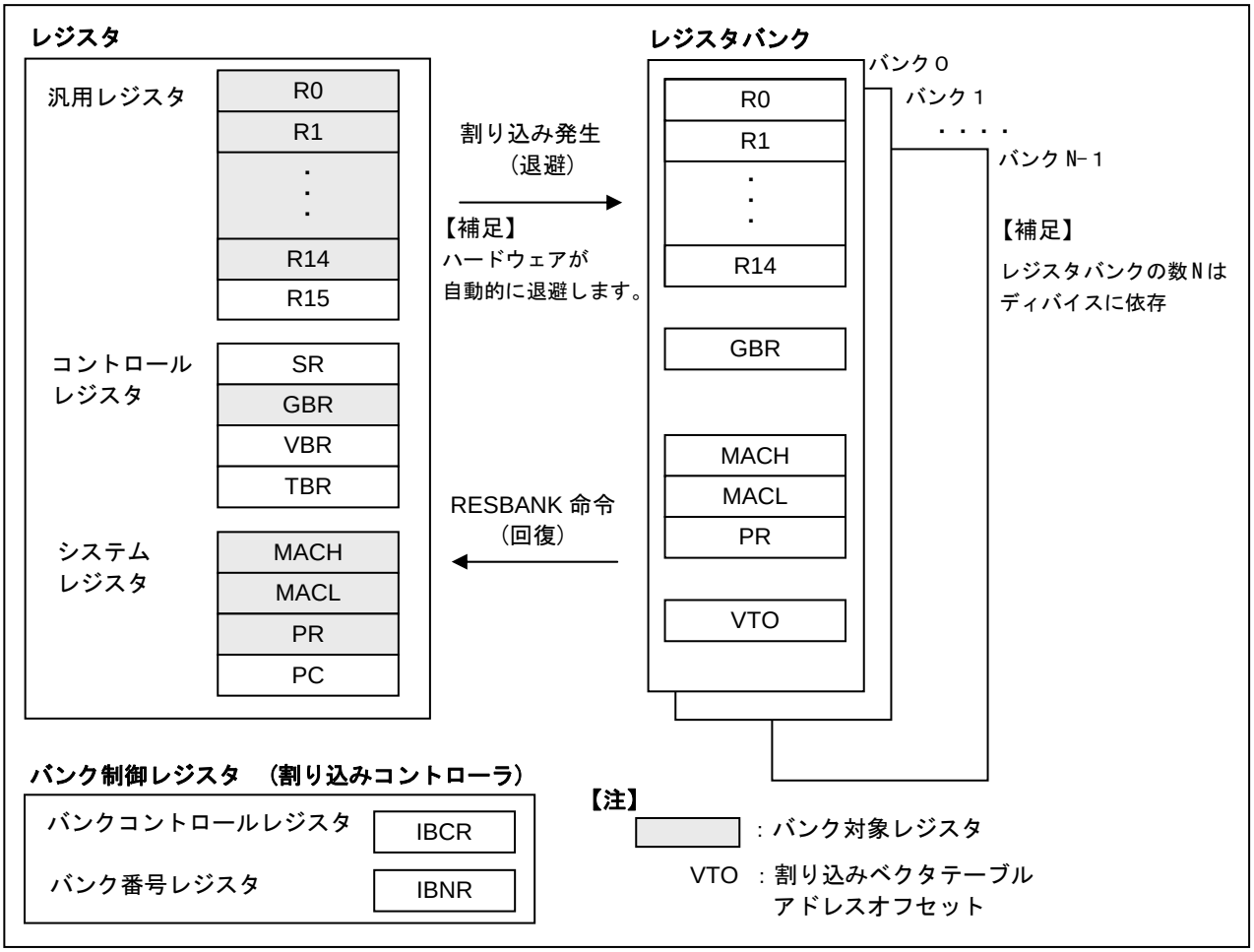


図 3-1 レジスタバンクの構成と概要

割り込みが発生した場合、割り込み処理で使用されるレジスタは値を退避・回復する必要があります。SH-2 ではスタックへ必要なレジスタの退避・回復を行います。この退避・回復処理はソフトウェアによって行う必要があります。C 言語を使用している場合は、割り込み関数(#pragma interrupt 指定された関数)の先頭と末尾でレジスタの退避・回復を行うコードが生成されます。SH-2A ではレジスタの退避・回復をハードウェアで高速に行う”レジスタバンク”という機構が備わっています。このレジスタバンクを使用することで、割り込み応答速度を大幅に向上させることができます。

レジスタバンクを活用するには、バンク制御レジスタの設定と割り込み関数指定の両方にて、レジスタバンクに対応する必要があります。

バンク制御レジスタの設定では、バンク制御レジスタのバンク番号レジスタ(IBNR) 及び バンクコントロールレジスタ(IBCR)をレジスタバンクが有効となるように設定する必要があります。具体的な設定値は、ハードウェアマニュアルの「レジスタバンク」をご参照ください。

割り込み関数の指定では、割り込み関数指定#pragma interrupt に対して、レジスタバンク指定”resbank” を行う必要があります。”resbank”が指定された割り込み関数は、関数の先頭でスタックへレジスタを退避するコードを生成せず、RTE 命令で戻る前に resbank 命令を発行するコードを生成します。HEW が自動生成する SH-2A 向けのサンプルスタートアップルーチンでは、割り込み関数に対して”resbank”指定が行われていません。そのため、レジスタバンクを使用しない(SH-2 と同様の)割り込み関数が生成されます。SH-2A のレジスタバンク機能を活用したい場合は、”resbank”指定を追記し、バンク制御レジスタの設定を行ってください。#pragma interrupt の詳細な仕様は、コンパイラユーザーズマニュアル 「10.3.1 #pragma 拡張子」 をご参照ください。

＜例 3-1＞

SH-2A resbank指定なし -abs20

＜Cソースコード＞
```
#pragma interrupt (func)
void func(void);

unsigned int A;

void func(void){

        A = 0;

}
```

```
_func:
  MOV.L   R2,@-R15
  MOV.L   R3,@-R15
  MOV     #0,R2       ; H'00000000
  MOVI20  #_A,R3
  MOV.L   R2,@R3
  MOV.L   @R15+,R3
  MOV.L   @R15+,R2
  RTE
  NOP
```

スタックへ退避・回復

SH-2A resbank 指定あり -abs20

＜Cソースコード＞
```
#pragma interrupt (func2(resbank))
void func2(void);

unsigned int A;

void func(void){

        A = 0;

}
```

resbank 指定

```
_func:
  削除
  MOV     #0,R2       ; H'00000000
  MOVI20  #_A,R3
  MOV.L   R2,@R3
  削除
  RESBANK
  RTE
  NOP
```

スタックへの
退避・回復 削除

3.2 ジャンプテーブルベースレジスタ(TBR)

SH-2A のコントロールレジスタにはジャンプテーブルベースレジスタ(TBR)が追加されています。TBR はメモリに配置された関数テーブル(=TBR 関数テーブル) を指すレジスタであり、テーブル参照サブルーチンコール命令 JSR/N @@(disp8,TBR)（= TBR レジスタ相対呼び出し）で使用されます。TBR レジスタ相対呼び出しを用いることで、関数アドレスのレジスタへの読み込みが不要となり、コードサイズの削減と速度向上が期待できます。また、20 ビット長即値ロード(MOVI20)を用いた関数呼び出しでは、関数の配置アドレス範囲に制限がありますが、TBR を活用したサブルーチンコールではアドレス範囲に制限がありません。命令長もテーブル参照サブルーチンコールは 16 ビット命令、20 ビット長即値ロードは 32 ビット命令であるため、TBR レジスタ相対呼び出しを活用した関数呼び出しの方がコードサイズの面で有利となります。

TBR を活用した関数呼び出しを行うには、#pragma tbr (関数名) 指定 もしくは TBR オプション を使用します。#pragma tbr (関数名) 指定が行われた関数は、関数呼び出しが TBR レジスタ相対呼び出しとなり、関数定義がある場合はセクション：”$TBR” に関数のアドレスが出力されます。 この$TBR セクションが TBR 関数テーブルです。1 つの TBR

関数テーブルには 255 個の関数が指定できます。もし、255 個を超える関数を扱う必要があるときには、#pragma tbr (関数名(sn=セクション名)) を使用します。#pragma tbr (関数名(sn=セクション名)) で指定された関数は、セクション：”$TBR セクション名” に関数のアドレスが出力される為、複数の TBR 関数テーブルを用意することが出来ます。TBR 関数テーブルが異なる関数を呼び出す時は TBR の再設定が必要となります。TBR 関数テーブルはリンカのセクション指定で ROM に配置してください。 TBR オプションを用いた場合は、ファイル内のすべての関数が TBR レジスタ相対呼び出しとなります。 #pragma tbr の詳細な仕様は、コンパイラユーザーズマニュアル 「10.3.1 #pragma 拡張子」 を TBR オプションの詳細は「2.3.4 C/C++コンパイラ操作方法 オブジェクトオプション」をご参照ください。

＜例 3-2＞

Source Program

```
＜関数定義部＞
#pragma tbr (test)
#pragma tbr (test2 (sn=_2))

unsigned int A;

void test(void);
void test2(void);

void test(void){

      A = 0;
}

void test2(void){

      A = 0;
}
```

例として、test と test1 を別の TBR 関数テーブルに配置する

set_tbr を使う時は必ず include して下さい

```
＜関数呼び出し部＞
#include <machine.h>

#pragma tbr (test)
#pragma tbr (test2 (sn=_2))

void test(void);
void test2(void);

void call(void)
{
        set_tbr(__sectop("$TBR"));
        test();
        set_tbr(__sectop("$TBR_2"));
        test2();
}
```

呼び出し側にも定義時と同じ#pragma tbr 指定が必要

TBR レジスタ相対呼び出しの前に、TBR に TBR 関数テーブルの先頭アドレスを設定します

TBR 関数テーブルが異なる関数を呼び出す為、TBR の再設定が必要

SH-2A -abs20

```
＜関数定義部＞
        .EXPORT     _A
        .EXPORT     _test
        .EXPORT     _test2
        .EXPORT     $_test
        .EXPORT     $_test2
_test:
        MOV         #0,R2
        MOVI20      #_A,R3
        MOV.L       R2,@R3
        RTS/N

_test2:
        MOV         #0,R2
        MOVI20      #_A,R3
        MOV.L       R2,@R3
        RTS/N
_A:
        .RES.L      1

        .SECTION    $TBR,DATA,ALIGN=4
$_test:
        .DATA.L     _test

        .SECTION    $TBR_2,DATA,ALIGN=4
$_test2:
        .DATA.L     _test2
```

test のアドレスが $TBR セクションに出力

test2 のアドレスが $TBR_2 セクションに出力

```
＜関数呼び出し部＞
_call:
        STS.L       PR,@-R15

        MOV.L       L12+2,R2    ; STARTOF $TBR
        LDC         R2,TBR

        JSR/N       @@($_test-(STARTOF $TBR),TBR)

        MOV.L       L12+6,R2    ; STARTOF $TBR_2
        LDC         R2,TBR

        JSR/N       @@($_test2-(STARTOF $TBR_2),TBR)

        LDS.L       @R15+,PR
        RTS/N
L12:

        .RES.W      1
        .DATA.L     STARTOF $TBR
        .DATA.L     STARTOF $TBR_2
```

TBR の設定

TBR レジスタ相対呼び出し

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。