お客様各位

資料中の「ラピスセミコンダクタ」等名称の ラピステクノロジー株式会社への変更

2020 年 10 月 1 日をもって、ラピスセミコンダクタ株式会社の LSI 事業部門は、ラピステクノロジー株式会社に分割承継されました。従いまして、本資料中にあります「ラピスセミコンダクタ株式会社」、「ラピスセミ」、「ラピス」といった表記に関しましては、全て「ラピステクノロジー株式会社」に読み替えて適用するものとさせていただきます。なお、会社名、会社商標、ロゴ等以外の製品に関する内容については、変更はありません。以上、ご理解の程よろしくお願いいたします。

2020年10月1日
ラピステクノロジー株式会社

Dear customer

LAPIS Semiconductor Co., Ltd. ("LAPIS Semiconductor"), on the 1st day of October, 2020, implemented the incorporation-type company split (shinsetsu-bunkatsu) in which LAPIS established a new company, LAPIS Technology Co., Ltd. (“LAPIS Technology”) and LAPIS Technology succeeded LAPIS Semiconductor’s LSI business.

Therefore, all references to "LAPIS Semiconductor Co., Ltd.", "LAPIS Semiconductor" and/or "LAPIS" in this document shall be replaced with "LAPIS Technology Co., Ltd."

Furthermore, there are no changes to the documents relating to our products other than the company name, the company trademark, logo, etc.

Thank you for your understanding.

LAPIS Technology Co., Ltd.
October 1, 2020

ROHM GROUP
LAPIS SEMICONDUCTOR

CCU8

コードサイズ圧縮ガイド

第４版

2016年1月15日発行

ROHM SEMICONDUCTOR

ご注意

1) 本資料の記載内容は改良などのため予告なく変更することがあります。

2) ラピスセミコンダクタは常に品質・信頼性の向上に取り組んでおりますが、半導体製品は種々の要因で故障・誤作動する可能性があります。
万が一、本製品が故障・誤作動した場合であっても、その影響により人身事故、火災損害等が起こらないようご使用機器でのディレーティング、冗長設計、延焼防止、バックアップ、フェイルセーフ等の安全確保をお願いします。定格を超えたご使用や使用上の注意書が守られていない場合、いかなる責任もラピスセミコンダクタは負うものではありません。

3) 本資料に記載されております応用回路例やその定数などの情報につきましては、本製品の標準的な動作や使い方を説明するものです。したがいまして、量産設計をされる場合には、外部諸条件を考慮していただきますようお願いいたします。

4) 本資料に記載されております技術情報は、本製品の代表的動作および応用回路例などを示したものであり、それをもって、当該技術情報に関するラピスセミコンダクタまたは第三者の知的財産権その他の権利を許諾するものではありません。したがいまして、上記技術情報の使用に起因して第三者の権利にかかわる紛争が発生した場合、ラピスセミコンダクタはその責任を負うものではありません。

5) 本製品は、一般的な電子機器(AV機器、OA機器、通信機器、家電製品、アミューズメント機器など)および本資料に明示した用途への使用を意図しています。

6) 本資料に掲載されております製品は、耐放射線設計はなされておりません。

7) 本製品を下記のような特に高い信頼性が要求される機器等に使用される際には、ラピスセミコンダクタへ必ずご連絡の上、承諾を得てください。
・輸送機器(車載、船舶、鉄道など)、幹線用通信機器、交通信号機器、防災・防犯装置、安全確保のための装置、医療機器、サーバー、太陽電池、送電システム

8) 本製品を極めて高い信頼性を要求される下記のような機器等には、使用しないでください。
・航空宇宙機器、原子力制御機器、海底中継機器

9) 本資料の記載に従わないために生じたいかなる事故、損害もラピスセミコンダクタはその責任を負うものではありません。

10) 本資料に記載されております情報は、正確を期すため慎重に作成したものですが、万が一、当該情報の誤り・誤植に起因する損害がお客様に生じた場合においても、ラピスセミコンダクタはその責任を負うものではありません。

11) 本製品のご使用に際しては、RoHS 指令など適用される環境関連法令を遵守の上ご使用ください。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、ラピスセミコンダクタは一切の責任を負いません。本製品の RoHS 適合性などの詳細につきましては、セールス・オフィスまでお問合せください。

12) 本製品および本資料に記載の技術を輸出又は国外へ提供する際には、「外国為替及び外国貿易法」、
「米国輸出管理規則」など適用される輸出関連法令を遵守し、それらの定めにしたがって必要な手続を行ってください。

13) 本資料の一部または全部をラピスセミコンダクタの許可なく、転載・複写することを堅くお断りします。

ラピスセミコンダクタ株式会社
〒222-8575 神奈川県横浜市港北区新横浜 2-4-8
http://www.lapis-semi.com

目次

概要

本文書は、U8/U16用CCU8 C コンパイラ（以下CCU8と称す）において、コードサイズを圧縮するための方法をまとめたものです。CCU8が出力するコードサイズを圧縮するには、以下の2つの方法があります。

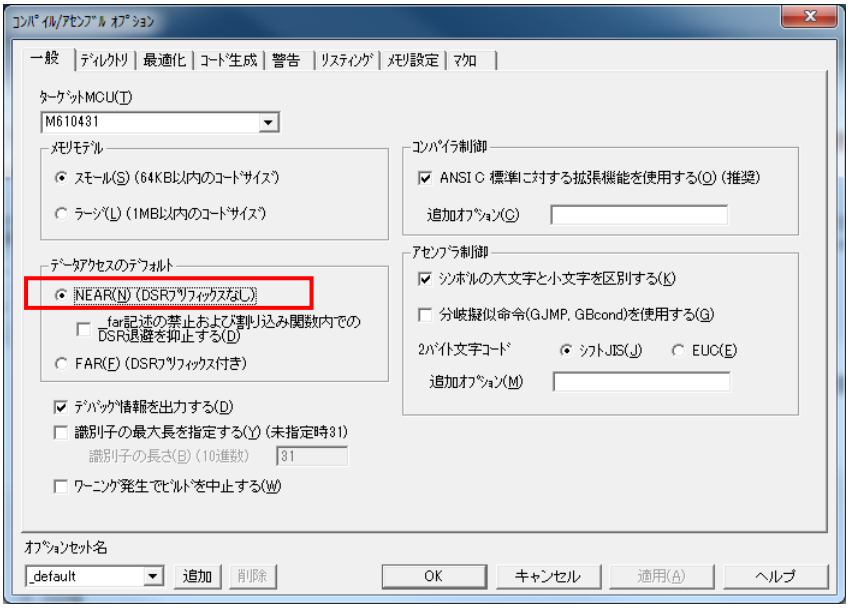
- 出力コードのサイズが最小となるコンパイラのオプションを設定する。
- コンパイラが効率的なコードを出力するように、C言語の記述を変更する

コンパイラのオプション設定と、効率的なコードを出力するための記述について、順に説明します。
本文書では、unsigned char 型とsigned char 型の両方の型を指す意味で、単にchar 型と書きます。特に符号付きの型と符号なしの型を区別する場合には signed char、unsigned char と明記します。同様にint型もunsigned int 型とsigned int 型の両方の型を指し、符号付きの型と符号なしの型を区別する場合には、signed int、unsigned int と明記します。

1. コードサイズを最小にするコンパイラのオプション設定

コードサイズを最小にするためのコンパイラのオプション設定を以下に示します。

[一般]タブ
データアクセスのデフォルトでは「NEAR(N) (DSRプリフィックスなし)」を選択してください。



上記のように設定することで、__far修飾されていない変数はすべてセグメント0に配置され、変数のアクセスに対して効率的な（DSRプリフィックスを使わない）命令が出力されます。

セグメント０に変数が入りきらない場合には、使用頻度の少ない変数を__far修飾してください。

変数がセグメント０に入りきらない場合の対応例

```
int nvar1;
int nvar2;
...
int nvarN;

char __far fvar[32];
```

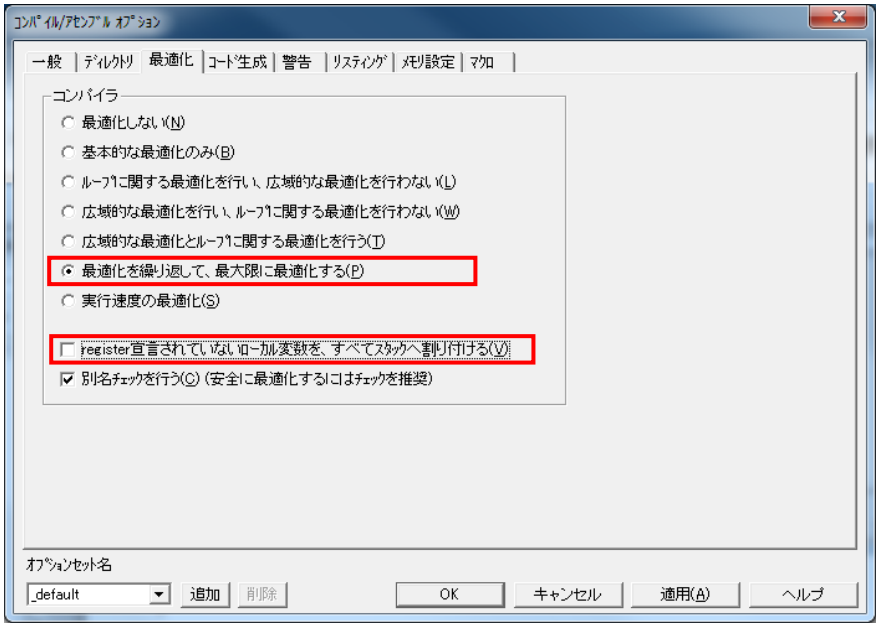
__far 修飾されていない変数はすべてセグメント 0 に配置され、DSR プリフィックスを使わない命令が出力されます

使用頻度の少ない変数を__far 修飾してください

[最適化]タブ

「最適化を繰り返して、最大限に最適化する(P)」を選択してください。

「register宣言されていないローカル変数を、すべてスタックへ割り付ける(V)」のチェックを外してください。



「最適化を繰り返して、最大限に最適化する(P)」を選択した場合、コンパイラのコマンドラインオプションに/Omを指定したことになります。実施される最適化の内容については、『CCU8ユーザーズマニュアル』の「3.2.3.9 最適化オプションのまとめ」を参照してください。

「register宣言されていないローカル変数を、すべてスタックへ割り付ける(V)」のチェックを外した場合の効果を、以下に示します。

ローカル変数が、スタックではなくレジスタに割り付けられた場合、メモリへの書き戻しが不要なため、その分コードサイズを削減できます。

Cソースプログラム	「register宣言されていない…」チェックありの場合	「register宣言されていない…」チェックなしの場合

void func(void)
{
  int i;
  for(i=0;i<10;i++)
  sub(i);
}
	_func :
	…
	;; for(i=0;i<10;i++)
	mov er0, #0
	st er0, -2[fp]
	_$L3 :
	bl _sub
	
	;; for(i=0;i<10;i++)
	l er0, -2[fp]
	add er0, #1
	st er0, -2[fp]
	cmp r0, #0ah
	cmpc r1, #00h
	blts _$L3
	…
		_func :
		…
		;; for(i=0;i<10;i++)
		
		mov er4, #0
		_$L3 :
		mov er0, er4
		bl _sub
		;; for(i=0;i<10;i++)
		add er4, #1
		
		cmp r4, #0ah
		cmpc r5, #00h
		blts _$L3
		…

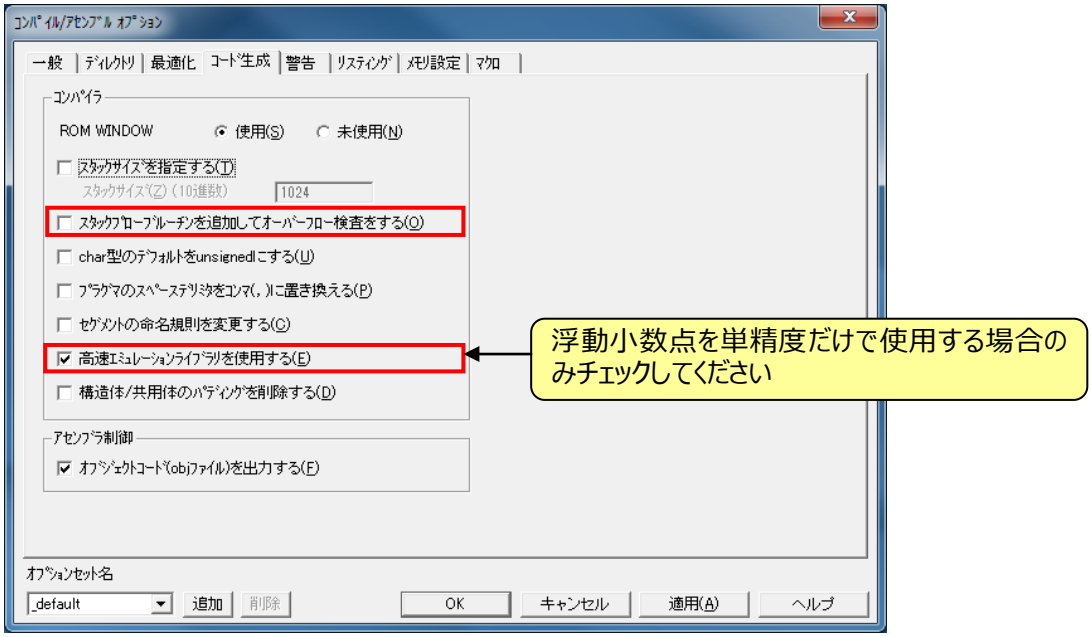
ローカル変数 i に該当

ローカル変数がレジスタに割り付けられた場合、メモリへの書き戻しが不要

[コード生成]タブ

「スタックプローブルーチンを追加してオーバーフロー検査をする」のチェックを外してください。

浮動小数点を単精度だけで使用する場合は「高速エミュレーションライブラリを使用する」をチェックしてください。



「スタックプローブルーチンを追加してオーバーフロー検査をする」のチェックを外すことで、スタックプローブルーチンを呼び出さないようにするため、コードサイズを削減できます。

Cソースプログラム	「スタックプローブルーチン…」チェックありの場合	「スタックプローブルーチン…」チェックなしの場合

void func(void)
{
  volatile int i;
  for(i=0;i<10;i++);
}
 	
スタックプローブルーチンの呼び出しコードが、関数ごとに出力されるため、コードサイズが増加します
	_func :
	  push lr
	  mov er0, #06h
	  bl __chstu8sw
	  push fp
	  mov fp, sp
	  add sp, #-04
	;; for(i=0;i<10;i++);
	  mov er0, #0
	  st er0, -4[fp]
	  bal _$L7
	  …
		_func :
		
		  push fp
		  mov fp, sp
		  add sp, #-04
		;; for(i=0;i<10;i++);
		  mov er0, #0
		  st er0, -4[fp]
		  bal _$L7
		  …

「高速エミュレーションライブラリを使用する」をチェックした場合、単精度のライブラリ（floatu8.lib）がリンクされるため、コードサイズが小さくなります。浮動小数点の乗算を行った場合のコードサイズの違いを以下に示します。

```
float f1, f2;
void test(void)
{
    f1 = f1 * f2;
}
```

「高速エミュレーションライブラリ」を使用しない場合の浮動小数点ライブラリ（乗算）のサイズ：678 byte	「高速エミュレーションライブラリ」を使用した場合の浮動小数点ライブラリ（乗算）のサイズ：518 byte

※高速エミュレーションライブラリ（floatu8.lib）を使用する場合は、float、double型ともに単精度で演算します。
double型で倍精度を必要とする場合は、「高速エミュレーションライブラリを使用する」をOFFにしてください。

2. 効率的なコードを出力するための記述方法

ここでは、CCU8が出力するコードについての基礎知識と、CCU8が冗長なコードを出力する記述に対して改善する記述方法を説明します。またインストラクションの特性を利用して効率的なコードを出力する記述方法について説明します。

以下の項目に関して、冗長なコードの改善方法と効率的なコードの記述方法を説明します。

中項目	小項目		内容
CCU8が出力するコードについての基礎知識	2.1.1	U8/U16の命令の特徴	U8/U16の命令の特徴を簡単に説明します。
	2.1.2	スタックに割り付けられた関数引数およびローカル変数のアクセス	スタックに割り付けられた関数引数およびローカル変数のアクセスについて、相対値による命令サイズの違いを説明します
CCU8が出力する冗長なコードを改善する	2.2.1	char型同士の演算に対する汎整数拡張	char型に対する汎整数拡張を抑制します。
	2.2.2	1ビット長のビットフィールドに対する代入式	1 ビット長のビット変数へ式の結果を代入する場合に、if文を使用してコードサイズを小さくします。
	2.2.3	long変数を用いた演算	複数のlong変数を使用して処理を行う場合に、最適化により出力される冗長なコードを削減します。
	2.2.4	long型変数の比較	long型変数の内容を複数の値と比較する場合で比較値の一部が同じ場合に、比較を分けることでコードサイズを小さくします。
	2.2.5	引数やローカル変数のレジスタへの割り当て	使用頻度の高い引数やローカル変数をレジスタに割り当てることによりスタックへの読み書き処理を削減します。
	2.2.6	同じ変数に対する繰り返しアクセス	同じ変数を繰り返し参照する場合に、レジスタに割り当てられたローカル変数を使用することによりメモリからの読み出し処理を削減します。
	2.2.7	引数として渡されたポインタによるアクセス	引数として渡されたポインタを定数インデックスを用いてアクセスする場合に、BPレジスタを用いたアクセスにすることによりコードサイズを小さくします。
	2.2.8	ビットフィールドの同じ変数に対する複数ビットの書き込み	ビットフィールドの同じ変数に対して4ビット以上のビット操作を行う場合に、AND/OR演算を行うようにすることでコードサイズを小さくします。
U8/U16の命令の特性を利用して効率的なコードを出力する	2.3.1	乗除算、モジュロ算の代用	乗除算、モジュロ算に対し効率的なコードが出力されるようにします。
	2.3.2	0との比較	条件判断をおこなう場合に効率的なコードが出力されるようにします。

中項目	小項目		内容
その他	2.4.1	関数形式マクロを使いすぎない	呼び出し回数が多いマクロを関数として定義することでコードサイズを小さくします。
	2.4.2	ビットフィールドを多用しない	ビットフィールドの使用を抑えることでコードサイズを小さくします。
	2.4.3	一度しか呼ばない関数はインライン展開	一度しか呼ばない関数をインライン展開することで、関数呼び出しのオーバーヘッドを減らしてコードサイズを小さくします。
	2.4.4	使用頻度の高い関数をSWI化	使用頻度の高い関数をソフトウェア割り込み関数とすることで、関数呼び出しの命令サイズを短くすることでコードサイズを小さくします。

本文書で提示されているコンパイル結果は、いずれも前述の「1. コードサイズを最小にするコンパイラのオプション設定」でコンパイルした結果です。将来のバージョンや、異なるコンパイルオプションにおいても、同じコードや効果が得られることを保証するものではありません。

2.1. CCU8 が出力するコードについての基礎知識

2.1.1. U8/U16 の命令の特徴

U8/U16の命令は、ロード・ストアアーキテクチャであり、演算を行う場合はメモリからデータをレジスタにロードしてから演算を実行し、演算結果が格納されたレジスタの内容をメモリへストアするのが基本的なコードとなります。
また、U8/U16の命令は8ビットデータおよび16ビットデータの演算を中心とした命令体系となっているため、32ビット以上のデータに対して演算を行う場合、命令数が増加します。
以下に、データのサイズによって、コードサイズがどう変わるかを例に示します。

8ビットの演算の場合	16ビットの演算の場合	32ビットの演算の場合
unsigned char uc1; … uc1 = uc1 + uc2;	unsigned int ui1; … ui1 = ui1 + ui2;	unsigned long ul1; … ul1 = ul1 + ul2;

;---- ロード ------
  l r0, NEAR _uc2
  l r1, NEAR _uc1
;---- 演算 -------
  add r0, r1
;---- ストア ------
  st r0, NEAR _uc1
	;--- ロード --------
	  l er0, NEAR _ui1
	  l er2, NEAR _ui2
	;---- 演算 -------
	  add er0, er2
	;---- ストア ------
	  st er0, NEAR _ui1
		;---- ロード --------
		  l er4, NEAR _ul2
		  l er6, NEAR _ul2+02h
		  l er0, NEAR _ul1
		  l er2, NEAR _ul1+02h
		;---- 演算 ---------
		  add er0, er4
		  addc r2, r6
		  addc r3, r7
		;---- ストア --------
		  st er0, NEAR _ul1
		  st er2, NEAR _ul1+02h

合計：4命令、14バイト	合計：4命令、14バイト	合計：9命令、30バイト

上の例に示すように、8ビット、16ビットの場合は、コードサイズの違いはそれほど出ませんが、32ビットになるとロード/ストア命令が増加し、さらに演算のために多くのレジスタおよび命令を必要とするため、コードサイズが増加します。
極力32ビット以上のデータを使わずに、8ビットおよび16ビットのデータを使って処理することを推奨します。

2.1.2. スタックに割り付けられた関数引数およびローカル変数のアクセス

関数引数やローカル変数がレジスタに割り付けられない場合は、スタックに割り付けられます。
関数引数の渡し方については、『CCU8 プログラミングガイド』の「1.6.4 関数引数の割り当て規則」を参照して下さい。
スタック領域に割り付けられた関数引数やローカル変数をアクセスする場合、CCU8はFP（=ER14）を使った間接アドレッシング命令を使用します。このため、関数の入口・出口でスタックに割り付けられた引数やローカル変数をアクセスするための設定コード（下記の青文字表記の部分）を出力します（__regpushu8xx/__regpopu8xxを呼び出すコードが出力される場合もあります）。
関数引数やローカル変数がスタックに割り付けられない場合は、この設定コードは出力されません。

引数やローカル変数をスタックに割り付けて使用する場合	引数やローカル変数がレジスタのみの場合

void func3(long x, int y)
{
ui1 = y;
}
	void func4(int x, int y)
	{
	ui1 = y;
	}

_func3 :
push fp
mov fp, sp
;; ui1 = y;
l er0, 2[fp]
st er0, NEAR _ui1
;;}
mov sp, fp
pop fp
rt
スタック設定コード
	_func4 :
	;; ui1 = y;
	st er2, NEAR _ui1
	;;}
	rt
	関数引数およびローカル変数をスタックに割り付けない場合はスタック設定コードは出力されません

コードサイズ：16バイト	コードサイズ：6バイト

引数やローカル変数がスタックに割り付けられないように、引数やローカル変数の数を抑えることで、コードサイズの増加を抑えることができます。

ローカル変数や引数がスタックに割り付けられる場合、スタックに割り付けられる位置によってコードサイズが変わってきます。フレームポインタ（FP）によるメモリ間接アドレッシングにおいて、FPからの相対値が-32～+31の場合は2バイトのコードが出力されますが、前記の範囲を超えると4バイトのコードになります。

引数やローカル変数がスタックに割り付けられる場合の例を、以下に示します。

Cソース記述	出力アセンブリコード

void func(long l1, long l2, long l3, long l4, long l5,
char c1, char c2, char c3)
{
volatile char buf[32];
volatile char b = c1;
volatile char c = c2;
 	
test(buf, b, c);
 	
gc1 = buf[1];
gc2 = buf[2];
}
	_func :
	;;{
	push lr
	bl __regpushu8sw
	add sp, #-034
	
	;; volatile char b = c1;
	l r0, 30[fp]
	st r0, -1[fp]
	
	;; volatile char c = c2;
	l r0, 32[fp]
	st r0, -2[fp]
	
	…
	
	;; gc1 = buf[1];
	l r0, -33[fp]
	st r0, NEAR _gc1
	
	;; gc2 = buf[2];
	l r0, -32[fp]
	st r0, NEAR _gc2
	
	;;}
	b __regpopu8sw

引数 c1 に該当。FP からの相対値が 30 なので 2 バイト命令になります

引数 c1 に該当。FP からの相対値が 32 なので 4 バイト命令になります

ローカル変数 buf[1]に該当。FP からの相対値が-33 なので 4 バイト命令になります

ローカル変数 buf[2]に該当。FP からの相対値が-32 なので 2 バイト命令になります

上記のように、引数やローカル変数がスタックに割り付けられる場合、FPからの相対値によってメモリアクセス命令のサイズが変わります。1関数あたりの引数を14バイト以内、ローカル変数を32バイト以内にすることで、コードサイズを抑えることができます。

2.2. CCU8 が出力する冗長なコードを改善する

2.2.1. char 型同士の演算に対する汎整数拡張

CCU8 CコンパイラはANSI-C の汎整数拡張と呼ばれる規則に従ってchar型同士の式の演算結果をint型で出力するため、結果を参照する式のコードサイズが大きくなります。汎整数拡張を抑制するには、演算結果をchar型またはunsigned char型にキャストしてください。

以下に汎整数拡張を抑制する例を示します。
例2-2-1では、左シフトの結果が signed int型として扱われ、16bitでビット積（&）演算が行われています。改善後の記述ではシフト演算の結果をunsigned char型へキャストを行う事で汎整数拡張が抑制され8bitで演算されます。

例 2-2-1）改善前

```
unsigned char uc;
unsigned char uc_flag;

void cast_test1(unsigned char no)
{
    if (uc_flag & (uc << no))
    {
        test1();
    }
}
```

改善後

```
unsigned char uc;
unsigned char uc_flag;

void cast_test1(unsigned char no)
{
    if (uc_flag & (unsigned char)(uc << no))
    {
        test1();
    }
}
```

改善前の出力アセンブリコード（抜粋）

```
    mov  r8,  r0

;;  if (uc_flag & (uc << no))
    l    r0,  NEAR _uc
    mov  r1,  #00h
    mov  r2,  r8
    beq  _$M1
_$M2 :
    sllc r1,  #01h
    sll  r0,  #01h
    add  r2,  #0ffh
    bne  _$M2
_$M1 :
    l    r1,  NEAR _uc_flag
    and  r0,  r1
    beq  _$L1

;;      test1();
    bl   _test1
```

関数全体のサイズ：40 byte

改善後の出力アセンブリコード（抜粋）

```
    mov  r8,  r0

;;  if (uc_flag & (unsigned char)(uc << no))
    l    r0,  NEAR _uc
    mov  r1,  r8
    beq  _$M1
_$M2 :
    sll  r0,  #01h
    add  r1,  #0ffh
    bne  _$M2
_$M1 :
    l    r1,  NEAR _uc_flag
    and  r0,  r1
    beq  _$L1

;;      test1();
    bl   _test1
```

関数全体のサイズ：36 byte

2.2.2. 1 ビット長のビットフィールドに対する代入式

１ビット長のビット変数に対して式の結果を代入する場合は、右辺式をif 文の条件式として記述し、if 文の中ではビット変数に1または0を代入する式を記述してください。

例 2-2-2）改善前

```
struct {
    unsigned char b0 : 1;
    unsigned char b1 : 1;
    unsigned char b2 : 1;

} bitfld;
unsigned char uc_flag1, uc_flag2;

void bit_test2(void)
{
    bitfld.b1 =  (uc_flag1 == 0 ? 0 : 1);

    bitfld.b2 =  (uc_flag2 != 0);
}
```

改善後

```
struct {
    unsigned char b0 : 1;
    unsigned char b1 : 1;
    unsigned char b2 : 1;
} bitfld;
unsigned char uc_flag1, uc_flag2;

void bit_test2(void)
{
    if (uc_flag1 == 0x00)
        bitfld.b1 = 0x00;
    else
        bitfld.b1 = 0x01;

    if (uc_flag2)
        bitfld.b2 = 0x00;
    else
        bitfld.b2 = 0x01;
}
```

改善前の出力アセンブリコード（抜粋）

```
;; bitfld.b1 = uc_flag1 == 0 ? 0 : 1;
    l r0, NEAR _uc_flag1
    bne _$L1
    mov er0, #0
    bal _$L3
_$L1 :
    mov er0, #1
_$L3 :
    rb NEAR _bitfld.1
    and r0, #01h
    beq _$M1
    sb NEAR _bitfld.1
_$M1 :

;; bitfld.b2 = uc_flag2 != 0;
    mov r0, #00h
    l r1, NEAR _uc_flag2
    beq _$M2
    mov r0, #01h
_$M2 :
    rb NEAR _bitfld.2
    tb r0.0
    beq _$M3
    sb NEAR _bitfld.2
_$M3 :

;;}
    rt
```

関数全体のサイズ：48 byte

改善後の出力アセンブリコード（抜粋）

```
;; if (uc_flag1 == 0x00)
    l r0, NEAR _uc_flag1
    bne _$L1

;; bitfld.b1 = 0x00;
    rb NEAR _bitfld.1

;; else
    bal _$L3
_$L1 :

;; bitfld.b1 = 0x01;
    sb NEAR _bitfld.1
_$L3 :

;; if (uc_flag2)
    l r0, NEAR _uc_flag2
    beq _$L4

;; bitfld.b2 = 0x00;
    rb NEAR _bitfld.2

;; else
    rt
_$L4 :

;; bitfld.b2 = 0x01;
    sb NEAR _bitfld.2

;;}
    rt
```

関数全体のサイズ：34 byte

2.2.3. long 型変数を用いた演算

関数内でlong型またはunsigned long型の変数を複数使用した計算を行う場合、レジスタが不足し、それを補うために使用中のレジスタを一時的にスタックに退避するコード（これをスピルコードといいます）が出力され、コードサイズが増加する場合があります。この場合には、long型およびunsigned long型の変数をvolatileで修飾してください。

例 2-2-3）改善前

```
long inst_test1(long a, short s)
{
    long l1, l2;
    long b;

    b = a- 5000;
    l1 = (long)(s) * ((b * 6) >> 4);
    l2 = ((long)(s) * b) >> 12;

    b = l1 + l2;
    return b;
}
```

改善後

```
long inst_test1(long a, short s)
{
    volatile long l1, l2;
    volatile long b;

    b = a- 5000;
    l1 = (long)(s) * ((b * 6) >> 4);
    l2 = ((long)(s) * b) >> 12;

    b = l1 + l2;
    return b;
}
```

改善前の出力コード

```
_inst_test1 :

;;{
    push    lr
    bl  __regpushu8sw
    add sp, #-024
    mov er8,    er0
    mov er10,   er2

;;    return b;
    mov er4,    er0
    mov er6,    er2
    add r4, #078h
    addc    r5, #0ech
    addc    r6, #0ffh
    addc    r7, #0ffh
    push    xr4
    mov er0,    #6
    mov er2,    #0
    push    xr0
    bl  __lmulu8sw
    add sp, #4
    pop xr0

    push    xr4
    mov er4,    er0
    srlc    r4, #04h
    mov er6,    er2
    srlc    r5, #04h
    srlc    r6, #04h
    st  er4,    -12[fp]
    st  er6,    -10[fp]
    pop xr4
    l   er2,    -10[fp]
    sra r3, #04h
    st  er2,    -10[fp]
    l   er0,    14[fp]
    mov r2, r1
    extbw   er2
```

青文字で示しているコードが、レジスタ不足により出力されるスピルコードに該当します。

改善後の出力コード

```
_inst_test1 :

;;{
    push    lr
    bl  __regpushu8sw
    add sp, #-016

;;    b = a- 5000;
    add r0, #078h
    addc    r1, #0ech
    addc    r2, #0ffh
    addc    r3, #0ffh
    st  er0,    -4[fp]
    st  er2,    -2[fp]

;;    l1 = (long)(s) * ((b * 6) >> 4);
    l   er0,    -4[fp]
    l   er2,    -2[fp]
    push    xr0
    mov er0,    #6
    mov er2,    #0
    push    xr0
    bl  __lmulu8sw
    add sp, #4
    pop xr4
    srlc    r4, #04h
    srlc    r5, #04h
    srlc    r6, #04h
    sra r7, #04h
    l   er0,    14[fp]
    mov r2, r1
    extbw   er2
    mov r2, r3
    push    xr0
    push    xr4
    bl  __lmulu8sw
    add sp, #4
    pop xr4
    st  er4,    -8[fp]
```

```
    mov r2, r3
    st  er4,    -20[fp]
    st  er6,    -18[fp]
    push    xr0
    l   er0,    -12[fp]
    l   er2,    -10[fp]
    mov er4,    er0
    mov er6,    er2
    pop xr0
    push    qr0
    bl  __lmulu8sw
    add sp, #4
    st  er0,    -24[fp]
    st  er2,    -22[fp]
    pop xr0
    st  er0,    -16[fp]
    st  er2,    -14[fp]
    l   er2,    -22[fp]
    l   er0,    -24[fp]
    l   er8,    -20[fp]
    mov er4,    er8
    push    xr0
    l   er6,    -18[fp]
    pop xr0
    push    qr0
    bl  __lmulu8sw
    add sp, #4
    pop xr0
    mov r0, r1
    mov r1, r2
    mov r2, r3
    srlc    r0, #04h
    srlc    r1, #04h
    sra r2, #04h
    extbw   er2
    push    xr0
    l   er0,    -16[fp]
    l   er2,    -14[fp]
    mov er4,    er0
    mov er6,    er2
    pop xr0
    add er0,    er4
    addc    r2, r6
    addc    r3, r7

;;}
    b   __regpopu8sw
```

青文字で示しているコードが、レジスタ不足により出力されるスピルコードに該当します。

関数全体のサイズ：166 byte

```
    st  er6,    -6[fp]

;;    l2 = ((long)(s) * b) >> 12;
    push    xr0
    l   er0,    -4[fp]
    l   er2,    -2[fp]
    push    xr0
    bl  __lmulu8sw
    add sp, #4
    pop xr0
    mov r0, r1
    mov r1, r2
    mov r2, r3
    srlc    r0, #04h
    srlc    r1, #04h
    sra r2, #04h
    extbw   er2
    st  er0,    -12[fp]
    st  er2,    -10[fp]

;;    b = l1 + l2;
    l   er0,    -8[fp]
    l   er2,    -6[fp]
    l   er4,    -12[fp]
    l   er6,    -10[fp]
    add er0,    er4
    addc    r2, r6
    addc    r3, r7
    st  er0,    -4[fp]
    st  er2,    -2[fp]

;;    return b;
    l   er0,    -4[fp]
    l   er2,    -2[fp]
    mov er4,    er0
    mov er6,    er2

;;}
    b   __regpopu8sw
```

関数全体のサイズ：136 byte

2.2.4. long 型変数の比較

一つのif文の条件式において、一つのlong型変数の内容を複数の値と比較し、かつ比較する値の上位16ビットまたは下位16ビットが同じ場合、同じ値の部分を比較した後に残りの部分を比較することでコードサイズを小さくできます。
以下の例では、上位16ビットの比較値が0x1122で同じ値のため、上位16ビットの比較をした後に下位16ビットの比較をしています。

例 2-2-4）改善前

```
void test1(unsigned long ldat)
{
    if ((ldat != 0x11223344) &&
        (ldat != 0x11223456) &&
        (ldat != 0x11224455))
    {
        func();
    }
}
```

改善後

```
void test1(unsigned long ldat)
{
    unsigned short tmph = (ldat>>16)&0xffff;
    unsigned short tmpl = ldat&0xffff;
    if (tmph != 0x1122)
    {
        if ((tmpl != 0x3344) &&
            (tmpl != 0x3456) &&
            (tmpl != 0x4455))
        {
            func();
        }
    }
}
```

改善前の出力アセンブリコード（抜粋）

```
;;      (ldat != 0x11224455))
    cmp r0, #044h
    cmpc    r1, #033h
    cmpc    r2, #022h
    cmpc    r3, #011h
    beq _$L1
    cmp r0, #056h
    cmpc    r1, #034h
    cmpc    r2, #022h
    cmpc    r3, #011h
    beq _$L1
    cmp r0, #055h
    cmpc    r1, #044h
    cmpc    r2, #022h
    cmpc    r3, #011h
    beq _$L1

;;      func();
    bl  _func

;;  }
_$L1 :
```

関数全体のサイズ：38 byte

改善後の出力アセンブリコード（抜粋）

```
;;  if (tmph != 0x1122)
    cmp r2, #022h
    cmpc    r3, #011h
    beq _$L14

;;          (tmpl != 0x4455))
    cmp r0, #044h
    cmpc    r1, #033h
    beq _$L14
    cmp r0, #056h
    cmpc    r1, #034h
    beq _$L14
    cmp r0, #055h
    cmpc    r1, #044h
    beq _$L14

;;          func();
    bl  _func

;;      }
_$L14 :
```

関数全体のサイズ：32 byte

2.2.5. 引数やローカル変数のレジスタへの割り当て

使用頻度の高い引数やローカル変数をレジスタに割り当てることにより、スタックに対する読み書きが削減できるためコードサイズを小さくすることができます。
引数は自動でレジスタに割り当てられますが、ローカル変数をレジスタに割り当てるには、IDEU8の[コンパイル/アセンブルオプション]ダイアログの[最適化]タブにて、「register宣言されていないローカル変数を、すべてスタックへ割り当てる」をOFFに設定します。

１関数につきレジスタに割り当てられる引数は最大4バイトで、ローカル変数は引数と合わせて最大１０バイトとなるため、使用頻度の高い引数およびローカル変数は最初に定義してください。
引数やローカル変数の割り当て規則については『CCU8 プログラミングガイド』(SQ003023E004)の「1.6 アセンブリ言語との結合」を参照して下さい。

例2-2-5-1)

```
void test(int N, char A)
{
    int l5, l1, l2, l3, l4;

    for (l5 = 0; l5 < 100; l5++) {
        l2 = N * l1;
    }
    (中略)
}
```

使用頻度の高い引数 N とローカル変数 l5 をレジスタに割り当てるため先に定義する。

また使用頻度の高い引数がスタックに割り当てられている場合でも、一旦レジスタに割り当てられたローカル変数にコピーして処理を行うことでスタックに対する読み書きが削減できます。

例2-2-5-2では、引数s1、s2はレジスタに割り当てられ、引数nはスタックに割り当てられます。
改善前はレジスタに割り当てられなかった引数nをそのまま使用し、改善後では引数nをローカル変数cntにコピーした後、そのローカル変数を用いて処理を行っています。
改善前では引数に対する演算に対しメモリに対するロード・ストア命令を必要としますが、改善後ではローカル変数cntはレジスタに割り当てられているため、演算に対しロード・ストア命令を必要とせずコードサイズを小さくできます。

例2-2-5-2) 改善前

```
char *copy(char *s1, char *s2, int n)
{
    char *s;

    for (s = s1; 0 < n && *s2 != '¥0'; --n)
        *s++ = *s2++;
    for (; 0 < n; --n)
        *s++ = '¥0';
    return s1;
}
```

改善後

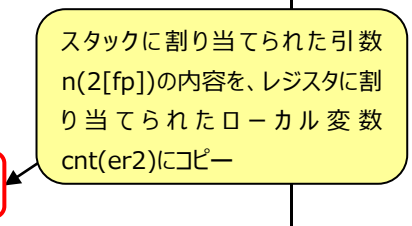
```
char *copy(char *s1, char *s2, int n)
{
    char *s;
    int cnt = n;

    for (s = s1; 0 < cnt && *s2 != '¥0'; --cnt)
        *s++ = *s2++;
    for (; 0 < cnt; --cnt)
        *s++ = '¥0';
    return s1;
}
```

改善前の出力アセンブリコード（抜粋）	改善後の出力アセンブリコード（抜粋）

_copy　:
;;{
（中略）
 	
;;　for (s = s1; 0 < n && *s2 != '¥0'; --n)
　mov bp, er0
 	
;;　　*s++ = *s2++;
_$L12 :
 	
;;　for (s = s1; 0 < n && *s2 != '¥0'; --n)
　mov er0,　#0
　l　er2,　2[fp]
　cmp er0,　er2
　bges　_$L19
　l　r0, [er10]
　beq _$L19
 	
;;　　*s++ = *s2++;
　st　r0, [bp]
　add er10,　#1
　add bp, #1
 	
;;　for (s = s1; 0 < n && *s2 != '¥0'; --n)
　mov er0,　er2
　add er0,　#-1
　st　er0,　2[fp]
 	
;;　　*s++ = *s2++;
　bal _$L12
 	
;;　for (; 0 < n; --n)
_$L15 :
（中略）
 	
;;　for (; 0 < n; --n)
　mov er0,　er2
　add er0,　#-1
　st　er0,　2[fp]
_$L19 :
 	
;;　for (; 0 < n; --n)
　mov er0,　#0
　l　er2,　2[fp]
　cmp er0,　er2
　blts　_$L15
（以下省略）
	_copy　:
	;;{
	（中略）
	
	;;　int cnt = n;
	　l　er0,　2[fp]
	　mov er2,　er0
	
	;;　for (s = s1; 0 < cnt && *s2 != '¥0'; --cnt)
	　mov bp, er8
	
	;;　　*s++ = *s2++;
	_$L12 :
	
	;;　for (s = s1; 0 < cnt && *s2 != '¥0'; --cnt)
	　mov er0,　#0
	　cmp er0,　er2
	　bges　_$L19
	　l　r0, [er10]
	　beq _$L19
	
	;;　　*s++ = *s2++;
	　st　r0, [bp]
	　add er10,　#1
	　add bp, #1
	
	;;　for (s = s1; 0 < cnt && *s2 != '¥0'; --cnt)
	　add er2,　#-1
	
	;;　　*s++ = *s2++;
	　bal _$L12
	
	;;　for (; 0 < cnt; --cnt)
	_$L15 :
	（中略）
	
	;;　for (; 0 < cnt; --cnt)
	　add er2,　#-1
	_$L19 :
	
	;;　for (; 0 < cnt; --cnt)
	　mov er0,　#0
	　cmp er0,　er2
	　blts　_$L15
	
	（以下省略）

関数全体のサイズ：72 byte	関数全体のサイズ：62 byte



2.2.6. 同じ変数に対する繰り返しアクセス

同じ変数に対して繰り返しアクセスする場合、その変数の内容を一旦ローカル変数に代入し、そのローカル変数を使用することによりコードサイズを小さくすることができます。

以下の例2-2-6-1では、配列変数modes[status]を繰り返し参照する記述をしています。CCU8はif文、else if文の行に対し、その都度配列変数modes[status]の値をメモリから読み込むコードを生成するためコードサイズが大きくなります。
改善後の記述のように配列変数modes[status]の内容を一旦ローカル変数に代入し、そのローカル変数を参照することによりメモリから読み込むコードが生成されなくなるためコードサイズが小さくなります。
ただし、プログラムによっては、記述を変更したことによりコードサイズが大きくなる場合があります。そのため、必ず変更後のコードサイズが大きくなっていないことを確認してください。コードサイズが大きくなっている場合は、元に戻してください。

例2-2-6-1） 改善前

```
unsigned char   modes[16];

void reuse_test1(unsigned int status)
{
    if (modes[status] == 0x01) {
        proc1();
    } else if (modes[status] == 0x02) {
        proc2();
    } else if (modes[status] == 0x03) {
        proc3();
    } else if (modes[status] == 0x04) {
        proc4();
    }
}
```

改善後

```
unsigned char   modes[16];

void reuse_test1(unsigned int status)
{
    unsigned char mode = modes[status];

    if (mode == 0x01) {
        proc1();
    } else if (mode == 0x02) {
        proc2();
    } else if (mode == 0x03) {
        proc3();
    } else if (mode == 0x04) {
        proc4();
    }
}
```

改善前の出力アセンブリコード（抜粋）	改善後の出力アセンブリコード（抜粋）

_reuse_test1　　:
;;{
　　push　　lr
　　push　　er8
　　mov er8,　　er0
 	
;;　if (modes[status] == 0x01) {
　　l　　r0, NEAR _modes[er0]
　　cmp r0, #01h
　　bne _$L1
 	
;;　　proc1();
　　bl　_proc1
 	
;;　} else if (modes[status] == 0x02) {
　　bal _$L10
_$L1 :
　　l　　r0, NEAR _modes[er8]
　　cmp r0, #02h
　　bne _$L4
 	
;;　　proc2();
　　bl　_proc2
 	
　　(中略)
;;　}
_$L10 :
 	
;;}
　　pop er8
　　pop pc
	_reuse_test1　　:
	;;{
	　　push　　lr
	
	;;　unsigned char mode = modes[status];
	　　l　　r2, NEAR _modes[er0]
	
	;;　if (mode == 0x01) {
	　　cmp r2, #01h
	　　bne _$L1
	
	;;　　res = proc1();
	　　bl　_proc1
	
	;;　} else if (mode == 0x02) {
	　　bal _$L10
	_$L1 :
	　　cmp r2, #02h
	　　bne _$L4
	
	;;　　res = proc2();
	　　bl　_proc2
	
	　　(中略)
	;;　}
	_$L10 :
	
	;;}
	　　pop pc

関数全体のサイズ：56 byte	関数全体のサイズ：46 byte

以下の例2-2-6-2では、構造体配列の要素tdat[idx1].m2[idx2]を繰り返しアクセスする記述をしています。CCU8は配列要素をアクセスするためのアドレス算出コードをその都度生成するため、コードサイズが大きくなります。
改善後の記述では、構造体配列のメンバtdat[idx1].m2[idx2]のアドレスを一旦ローカル変数のポインタに代入し、そのローカル変数のポインタを使用してメンバをアクセスするように変更しています。この変更により、アドレス算出のコードが最初の一回だけになるため、コードサイズが小さくなります。

例2-2-6-2） 改善前

```
typedef unsigned char UC;

typedef struct st1 {
    UC a;
    UC b;
    UC c;
    UC d;
} ST1;

typedef struct st2 {
    UC  m1[4];
    ST1 m2[4];
} ST2;

ST2 tdat[4];

void reuse_test2(UC mode, UC idx1, UC idx2)
{

    if(mode == 1)
    {
        tdat[idx1].m2[idx2].a = 0x01;
        tdat[idx1].m2[idx2].b = 0x02;
        tdat[idx1].m2[idx2].c = 0x03;
    }
    else
    {
        tdat[idx1].m2[idx2].a = 0x1f;
        tdat[idx1].m2[idx2].b = 0x2f;
        tdat[idx1].m2[idx2].c = 0x3f;
    }
}
```

改善後

```
typedef unsigned char UC;

typedef struct st1 {
    UC a;
    UC b;
    UC c;
    UC d;
} ST1;

typedef struct st2 {
    UC  m1[4];
    ST1 m2[4];
} ST2;

ST2 tdat[4];

void reuse_test2(UC mode, UC idx1, UC idx2)
{
    ST1 *p = &tdat[idx1].m2[idx2];
    if(mode == 1)
    {
        p->a = 0x01;
        p->b = 0x02;
        p->c = 0x03;
    }
    else
    {
        p->a = 0x1f;
        p->b = 0x2f;
        p->c = 0x3f;
    }
}
```

改善前の出力アセンブリコード(抜粋)

```
_reuse_test2 :
;;{
    push    lr
    bl  __regpushu8sw
    mov r9, r1
    mov r10,    r2
```

改善後の出力アセンブリコード(抜粋)

```
_reuse_test2 :
;;{
    push    lr
    push    xr8
    mov r10,    r2
    mov r8, r0

;;    ST1 *p = &tdat[idx1].m2[idx2];
    mov r0, r1
    mov r1, #00h
    mov er2,    #20
    bl  __imulu8sw
    add r0, #BYTE1 OFFSET _tdat
    addc    r1, #BYTE2 OFFSET _tdat
    mov r2, r10
```

```
;;    if(mode == 1)
    cmp r0, #01h
    bne _$L1

;;        tdat[idx1].m2[idx2].a = 0x01;
    mov r0, r1
    mov r1, #00h
    mov er2,    #20
    bl  __imulu8sw
    mov r2, r10
    mov r3, #00h
    sllc    r3, #02h
    sll r2, #02h
    mov bp, er0
    add bp, er2
    mov r0, #01h
    st  r0, NEAR _tdat+04h[bp]

;;        tdat[idx1].m2[idx2].b = 0x02;
    mov r0, r9
    mov r1, #00h
    mov er2,    #20
    bl  __imulu8sw
    mov r2, r10
    mov r3, #00h
    sllc    r3, #02h
    sll r2, #02h
    mov bp, er0
    add bp, er2
    mov r0, #02h
    st  r0, NEAR _tdat+05h[bp]
    (中略)

;;    else
    bal _$L3
_$L1 :
    (中略)
;;    }
_$L3 :
    st  r0, NEAR _tdat+06h[bp]

;;}
    b   __regpopu8sw
```

関数全体のサイズ：184 byte

```
    mov r3, #00h
    sllc    r3, #02h
    sll r2, #02h
    add er2,    er0
    add er2,    #4

;;    if(mode == 1)
    cmp r8, #01h
    bne _$L5

;;        p->a = 0x01;
    mov r0, #01h
    st  r0, [er2]

;;        p->b = 0x02;
    mov r0, #02h
    st  r0, 01h[er2]

    (中略)

;;    else
    bal _$L7
_$L5 :
    (中略)
;;    }
_$L7 :
    st  r0, 02h[er2]

;;}
    pop xr8
    pop pc
```

関数全体のサイズ：72 byte

2.2.7. 引数として渡されたポインタによるアクセス

引数として渡されたポインタを、関数内にてparg[2]のように定数インデックスを用いてアクセスする場合、disp16[ERn]を用いたメモリアクセス命令（4バイト）が使われます。

ポインタをローカル変数として宣言し、そのローカル変数にポインタ引数の内容をコピーしてアクセスするように変更するとdisp6[BP]を用いたメモリアクセス命令（2バイト）が使われる場合があります。このような場合に、コードサイズを小さくすることができます。ただし、必ずしもローカル変数のポインタがBPレジスタに割り当てられるとは限らないため、コードサイズが大きくなる場合があります。そのため、必ず変更後のコードサイズが大きくなっていないことを確認してください。コードサイズが大きくなっている場合は、元に戻してください。

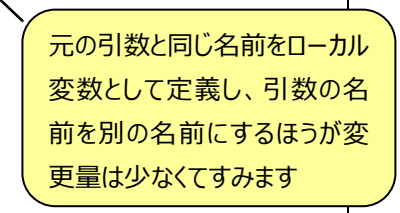
例2-2-7） 改善前

```
void test1(char *parg, int mode)
{

    if (mode == 1){
        parg[1] = 1;
        parg[2] = 5;
    } else {
        parg[3] = 10;
        parg[4] = 30;
    }
}
```

改善後

```
void test1(char *pargX, int mode)
{
    char *parg = pargX;
    if (mode == 1){
        parg[1] = 1;
        parg[2] = 5;
    } else {
        parg[3] = 10;
        parg[4] = 30;
    }
}
```



改善前の出力アセンブリコード（抜粋）	改善後の出力アセンブリコード（抜粋）

_test1 :
push er8
mov er8, er0
 	
;; if (mode == 1){
cmp r2, #01h
cmpc r3, #00h
bne _$L1
;; parg[1] = 1;
mov r0, #01h
st r0, 01h[er8]
;; parg[2] = 5;
mov r0, #05h
st r0, 02h[er8]
;; } else {
bal _$L3
_$L1 :
（中略）
;; }
_$L3 :
pop er8
rt
	_test1 :
	push bp
	
	;; char *parg = pargX;
	mov bp, er0
	;; if (mode == 1){
	cmp r2, #01h
	cmpc r3, #00h
	bne _$L5
	;; parg[1] = 1;
	mov r0, #01h
	st r0, 1[bp]
	;; parg[2] = 5;
	mov r0, #05h
	st r0, 2[bp]
	;; } else {
	pop bp
	rt
	_$L5 :
	（中略）
	;; }
	_$L7 :
	pop bp
	rt

関数全体のサイズ：40 byte	関数全体のサイズ：34 byte

2.2.8. ビットフィールドの同じ変数に対する複数ビットの書き込み

ビットフィールドの同じ変数に対して、複数のビット操作を行う場合、1ビットごとに0/1を書き込むよりも、AND/OR演算を行うようにするほうがコードサイズを小さくすることができます。
効果が現れる条件は、ビットフィールドの型、書き込むビット位置、および書き込む値によって、以下のように変わります。

ビットフィールドの型	書き込む値のビット位置	書き込む値	効果が現れる条件	変更例
char型	ビット0～7	0のみ	3ビット以上の書き込み	`Bitvar &= ~0x25;`
		1のみ		`Bitvar |= 0x25;`

	ビット0～7	0と1の混在	4ビット以上の書き込み	`tmp = Bitvar;`
				`tmp &= ~0x25;`
				`tmp |= 0x42;`
				`Bitvar = tmp;`

int型	ビット0～7、または ビット8～15のどちらか	0のみ	3ビット以上の書き込み	`Bitvar &= ~0x25;`
		1のみ		`Bitvar |= 0x2500;`

		0と1の混在	4ビット以上の書き込み	`tmp = Bitvar;`
				`tmp &= ~0x25;`
				`tmp |= 0x4200;`
				`Bitvar = tmp;`

	ビット0～7とビット8～15の両方	0のみ	4ビット以上の書き込み	`Bitvar &= ~0x205;`
		1のみ		`Bitvar |= 0x205;`

		0と1の混在	5ビット以上の書き込み	`tmp = Bitvar;`
				`tmp &= ~0x205;`
				`tmp |= 0x4020;`
				`Bitvar = tmp;`

以下の例では、char型のビットフィールド変数に対して4箇所のビットに対して0と1を書き込んでいます。このような記述に対し、AND演算とOR演算の組み合わせで各ビットへの書き込みを行うように変更することでコードサイズを小さくできます。

例８－１） 改善前

```
typedef union {
    struct {
        unsigned char b0:1;
        unsigned char b1:1;
        unsigned char b2:1;
        unsigned char b3:1;
        unsigned char b4:1;
        unsigned char b5:1;
        unsigned char b6:1;
        unsigned char b7:1;
    }bit;
    unsigned char byte;
}BF;

volatile BF data;
void test1(void)
{
    data.bit.b0 = 0;
    data.bit.b2 = 1;
    data.bit.b3 = 0;
    data.bit.b5 = 1;
}
```

改善後

```
typedef union {
    struct {
        unsigned char b0:1;
        unsigned char b1:1;
        unsigned char b2:1;
        unsigned char b3:1;
        unsigned char b4:1;
        unsigned char b5:1;
        unsigned char b6:1;
        unsigned char b7:1;
    }bit;
    unsigned char byte;
}BF;

volatile BF data;
void test1(void)
{
    unsigned char tmp = data.byte;
    tmp &= ~0x09; // Clear bit 0 & 3
    tmp |= 0x24;  // Set bit 2 & 5
    data.byte = tmp;
}
```

改善前の出力アセンブリコード（抜粋）	改善後の出力アセンブリコード（抜粋）

_test1 :
 	
;;{
 	
;; data.bit.b0 = 0;
rb NEAR _data.0
 	
;; data.bit.b2 = 1;
sb NEAR _data.2
 	
;; data.bit.b3 = 0;
rb NEAR _data.3
 	
;; data.bit.b5 = 1;
sb NEAR _data.5
 	
;;}
rt
	_test2 :
	
	;;{
	
	;; unsigned char tmp = data.byte;
	l r0, NEAR _data
	
	;; tmp &= ~0x09; // Clear bit 0 & 3
	and r0, #0f6h
	
	;; tmp |= 0x24; // Set bit 2 & 5
	or r0, #024h
	
	;; data.byte = tmp;
	st r0, NEAR _data
	
	;;}
	rt

関数全体のサイズ：18 byte	関数全体のサイズ：14 byte

2.3. U8/U16 の命令の特性を利用して効率的なコードを出力する

2.3.1. 乗除算、モジュロ算の代用

符号付き変数に対する2 の乗数による乗算・除算は、右シフトや左シフトに置き換えることでコードサイズを縮小することができます。（除算については、変数の内容が正値の場合に限られます）。

符号なし変数に対する2 の乗数による乗算・除算は、自動で右シフトや左シフトに置き換えられます。

また、符号なし変数に対する2 の乗数によるモジュロ算はビット単位のAND 演算子(&)に置き換え可能です。(符号付き変数に対するモジュロ算は結果が異なるためビット単位のAND 演算子に置き換え出来ません。)

例 2-3-1）改善前

```
long arith_test2(long a)
{
    return a/16;
}
```

改善後

```
long arith_test2(long a)
{
    return a>>4;
}
```

改善前の出力アセンブリコード（抜粋）	改善後の出力アセンブリコード（抜粋）

;; return a/16;
push xr0
mov er0, #16
mov er2, #0
push xr0
bl __ldivu8lw
add sp, #4
pop xr0
	;; return a>>4;
	srlc r0, #04h
	srlc r1, #04h
	srlc r2, #04h
	sra r3, #04h

関数全体のサイズ：20 byte	関数全体のサイズ：10 byte

2.3.2. 0 との比較

比較を行う場合、0 と比較する記述を行うことによりコードサイズを小さくすることができます。

U8/U16 コアでは、ロード命令によってZ フラグが更新されます。この特性を利用して、0 以外の値と比較するよりも0 と比較したほうが効率的なコードが出力されます。

例 2-3-2）改善前

```
#define FALSE (0)
#define TRUE  (1)
int b;

void inst_test1(void)
{
    if(b == TRUE) test1();
}
```

改善後

```
#define FALSE (0)
#define TRUE  (1)
int b;

void inst_test1(void)
{
    if(b != FALSE) test1();
}
```

改善前の出力アセンブリコード（抜粋）	改善後の出力アセンブリコード（抜粋）

;; if(b == TRUE) test1();
l er0, NEAR _b
cmp r0, #01h
cmpc r1, #00h
bne _$L1
bl _test1
_$L1 :
	;; if(b != FALSE) test1();
	l er0, NEAR _b
	beq _$L1
	bl _test1
	_$L1 :

関数全体のサイズ：18 byte	関数全体のサイズ：14 byte

2.4. その他

2.4.1. 関数形式マクロを使いすぎない

短い処理を関数形式マクロにして使用するのは一般的ですが、関数形式のマクロを使いすぎるとコードサイズが異常に大きくなりすぎてしまう場合があります。
呼び出し回数が 2 回以上のマクロは、関数として定義することを推奨します。

例2-4-1） 改善前

```
typedef unsigned long U32;
unsigned char buf[32];

#define rd4(p)¥
    (((U32)*p)|¥
    (((U32)(*(p+1)))<<8)|¥
    (((U32)(*(p+2)))<<16)|¥
    (((U32)(*(p+3)))<<24))


U32 l1, l2;
void funcA()
{
    l1 = rd4(&buf[1]);
    l2 = rd4(&buf[5]);
}
```

改善後

```
typedef unsigned long U32;
unsigned char buf[32];

U32 rd4X(unsigned char *p)
{
    return ((U32)*p)|
      (((U32)(*(p+1)))<<8)|
      (((U32)(*(p+2)))<<16)|
      (((U32)(*(p+3)))<<24);
}

U32 l1, l2;
void funcB()
{
    l1 = rd4X(&buf[1]);
    l2 = rd4X(&buf[5]);
}
```

改善前の出力アセンブリコード（抜粋）

```
_funcA  :

;;{
    push    xr4

;;    l1 = rd4(&buf[1]);
    l   r0, NEAR _buf+02h
    mov er2,    #0
    mov r3, r2
    mov r1, r0
    mov r0, #00h
    l   r4, NEAR _buf+01h
    mov r5, #00h
    mov er6,    #0
    or  r0, r4
    or  r2, r6
    or  r3, r7
    l   r4, NEAR _buf+03h
    mov er6,    er4
    mov er4,    #0
    or  r4, r0
    or  r5, r1
    or  r6, r2
    or  r7, r3
    l   r0, NEAR _buf+04h
    mov r3, r0
    mov r2, #00h
    mov er0,    #0
    or  r0, r4
```

改善後の出力アセンブリコード（抜粋）

```
_rd4X   :

;;{
    push    xr4
    push    er8
    mov er8,    er0

;;      (((U32)(*(p+3)))<<24);
    l   r0, 01h[er0]
    mov er2,    #0
    mov r3, r2
    mov r1, r0
    mov r0, #00h
    l   r4, [er8]
    mov r5, #00h
    mov er6,    #0
    or  r0, r4
    or  r2, r6
    or  r3, r7
    l   r4, 02h[er8]
    mov er6,    er4
    mov er4,    #0
    or  r4, r0
    or  r5, r1
    or  r6, r2
    or  r7, r3
    l   r0, 03h[er8]
    mov r3, r0
    mov r2, #00h
```

```
    or   r1, r5
    or   r2, r6
    or   r3, r7
    st   er0,     NEAR _l1
    st   er2,     NEAR _l1+02h

;;    l2 = rd4(&buf[5]);
    l    r0, NEAR _buf+06h
    mov er2,     #0
    mov r3, r2
    mov r1, r0
    mov r0, #00h
    l    r4, NEAR _buf+05h
    mov r5, #00h
    mov er6,     #0
    or   r0, r4
    or   r2, r6
    or   r3, r7
    l    r4, NEAR _buf+07h
    mov er6,     er4
    mov er4,     #0
    or   r4, r0
    or   r5, r1
    or   r6, r2
    or   r7, r3
    l    r0, NEAR _buf+08h
    mov r3, r0
    mov r2, #00h
    mov er0,     #0
    or   r0, r4
    or   r1, r5
    or   r2, r6
    or   r3, r7
    st   er0,     NEAR _l2
    st   er2,     NEAR _l2+02h

;;}
    pop xr4
    rt
```

関数全体のサイズ：142 byte

```
    mov er0,     #0
    or   r0, r4
    or   r1, r5
    or   r2, r6
    or   r3, r7

;;}
    pop er8
    pop xr4
    rt



_funcB  :

;;{
    push    lr

;;    l1 = rd4X(&buf[1]);
    mov r0, #BYTE1 OFFSET (_buf+01h)
    mov r1, #BYTE2 OFFSET (_buf+01h)
    bl   _rd4X
    st   er0,     NEAR _l1
    st   er2,     NEAR _l1+02h

;;    l2 = rd4X(&buf[5]);
    mov r0, #BYTE1 OFFSET (_buf+05h)
    mov r1, #BYTE2 OFFSET (_buf+05h)
    bl   _rd4X
    st   er0,     NEAR _l2
    st   er2,     NEAR _l2+02h

;;}
    pop pc
```

関数全体のサイズ：106 byte

2.4.2. ビットフィールドを多用しない

ビットフィールドは、データの圧縮をするための有効な手段ですが、2 ビット以上のビットフィールドのメンバをアクセスする場合には、コードサイズが大きくなります。

例2-4-2） 改善前

```
typedef unsigned char U8;
typedef struct st1{
    U8 data1:3;
    U8 data2:3;
    U8 flag1:1;
    U8 flag2:1;
}ST1;

ST1 stvar;
void funcA(U8 a)
{
    stvar.data2 = a;
}
```

改善後

```
U8 ucvar;
void funcB(U8 a)
{
    ucvar = a;
}
```

改善前の出力アセンブリコード（抜粋）	改善後の出力アセンブリコード（抜粋）
_funcA : ;;{ ;; stvar.data2 = a; sll r0, #03h l r1, NEAR _stvar xor r0, r1 and r0, #038h xor r1, r0 st r1, NEAR _stvar ;;} rt	_funcB : ;;{ ;; gvar2 = a; st r0, NEAR _ucvar ;;} rt
関数全体のサイズ：18 byte	関数全体のサイズ：6 byte

上記の改善前の例では、ビット 3～5 のデータだけを更新し、他のビットの内容を保持する必要があるため、処理が複雑になります。一方、改善後の例では右辺、左辺ともに同じ型でそのまま上書きできるため、単純な処理で済みます。

2.4.3. 一度しか呼ばない関数はインライン展開

一度しか呼ばない小さな関数は、インライン展開することで関数呼び出しのオーバーヘッドを削減できます。

例2-4-3） 改善前

```
typedef unsigned char UC;
UC ucdata[32];
UC uc;


UC get_data(UC i)
{
    return  ucdata[i];
}

void funcA()
{
    uc = get_data(3);
}
```

改善後

```
typedef unsigned char UC;
UC ucdata[32];
UC uc;
#pragma inline get_data

UC get_data(UC i)
{
    return  ucdata[i];
}

void funcA()
{
    uc = get_data(3);
}
```

改善前の出力アセンブリコード（抜粋）

```
_get_data   :
;;  return  ucdata[i];
    mov r1, #00h
    l   r0, NEAR _ucdata[er0]
    rt

_funcA  :
    push    lr
;;  uc = get_data(3);
    mov r0, #03h
    bl  _get_data
    st  r0, NEAR _uc
    pop pc
```

関数呼び出しのオーバヘッド

2つの関数（get_dataとfuncA）合計のサイズ：22 byte

改善後の出力アセンブリコード（抜粋）

関数呼び出しのオーバーヘッドがなくなり、さらにインライン展開された関数本体（get_data）が削除されるため、コードサイズが小さくなります。

```
_funcA :
;;  uc = get_data(3);
    l   r0, NEAR _ucdata+03h
    st  r0, NEAR _uc
    rt
```

関数全体のサイズ：10 byte

2.4.4. 使用頻度の高い関数を SWI 化

使用頻度の高い小さな関数は、ソフトウェア割り込み（SWI）関数とすることで、関数呼び出しのコードサイズを小さくできます。ただし、ソフトウェア割込み関数とすることに対し、以下の注意点があります。

・割り込み関数としての位置づけからレジスタ退避・復帰のコードが、通常の関数よりも増加する場合があります。
・SWIの分岐先テーブル（ベクタコード：2バイト）が生成されるため、その分のコードが増加します。
・呼び出しおよび復帰のためのサイクル数が増加し、通常の関数呼び出しよりも遅くなります。
・割り込み関数のため、明示的に割り込みを許可にしない限り、割り込み禁止となります。

以上より、SWI関数の対象とするものは、サブルーチン呼び出しのない小さな関数で（実行時間の短い関数）で、呼び出し回数が10回以上の関数を目安にしてください。

例2-4-4） 改善前

```
typedef unsigned char UC;
UC uc;

UC isnumber(UC a)
{
    if((a>=0x30)&&(a<=0x39))
        return  1;
    else  return  0;
}
void funcA()
{
    if(isnumber(uc))
        func();
}
```

改善後

```
typedef unsigned char UC;
UC uc;
#pragma swi  isnumber 0x80 1
UC isnumber(UC a)
{
    if((a>=0x30)&&(a<=0x39))
        return  1;
    else  return  0;
}
void funcA()
{
    if(isnumber(uc))
        func();
}
```

改善前の出力アセンブリコード（抜粋）

```
_isnumber    :
;;{
;;    if((a >= 0x30) && (a <= 0x39))
    cmp r0, #030h
    blt _$L68
    cmp r0, #039h
    bgt _$L68
;;        return  1;
    mov r0, #01h
;;}

    rt
;;    else
_$L68 :
;;        return  0;
    mov r0, #00h
    rt


_funcA :
;;{
    push    lr
```

改善後の出力アセンブリコード（抜粋）

```
_isnumber    :
;;{
;;    if((a >= 0x30) && (a <= 0x39))
    cmp r0, #030h
    blt _$L80
    cmp r0, #039h
    bgt _$L80
;;        return  1;
    mov r0, #01h
;;}
_$L79 :
    rti
;;    else
_$L80 :
;;        return  0;
    mov r0, #00h
    rti


_funcA    :
;;{
    push    lr
```

注意：ソフトウェア割り込み関数にすることにより、関数入口・出口でのレジスタ退避・復帰コードが増加する場合があります。

```
;;     if(isnumber(uc))
       |   r0, NEAR _uc
       bl   _isnumber
       cmp r0, #00h
       beq _$L77
;;         func();
       bl  _func
_$L77 :
;;}
           pop pc
```

関数呼び出しのコードは４バイト

```
;;     if(isnumber(uc))
       |   r0, NEAR _uc
       swi  _vct80$isnumber
       cmp r0, #00h
       beq _$L89
;;         func();
       bl  _func
_$L89 :
;;}
           pop pc

       cseg #00h at 080h
_vct80$isnumber :
       dw  _isnumber
```

関数呼び出しのコードは ２バイト。この部分のコード削減が図れます

注意：ソフトウェア割り込み関数にすると、分岐先テーブル（ベクタコード：２バイト）が別途出力されます。

改版履歴

版数	発行日	改版内容
1	2011.6.1	初版
2	2011.6.22	項目の追加 1.6 高速エミュレーションライブラリ（floatu8.lib）の利用
3	2014.9.11	小項目のタイトルを変更 1.1. キャストによる汎整数拡張の抑制 →1.1. char 型同士の演算に対する汎整数拡張 1.3. long 変数の取り扱い →1.3. long 変数を用いた複雑な演算 1.5. 同じ変数を繰り返し参照する記述 →1.6. 同じ変数に対する繰り返しアクセス 項目を追加 1.4. long 型変数の比較 1.6. 同じ変数に対する繰り返しアクセス →例を追加 1.7. 引数として渡されたポインタによるアクセス 1.8. ビットフィールドの同じ変数に対する複数ビットの書き込み
4	2016.1.15	表題を『CCU8 コードサイズ圧縮ガイド』に変更 以下の項目を追加 「1.コードサイズを最小にするオプション設定」 「2.1. CCU8 が出力するコードについての基礎知識」 「2.4.その他」の内容を下記に変更、追加 2.4.1 関数形式マクロを使いすぎない 2.4.2 ビットフィールドを多用しない 2.4.3 一度しか呼ばない関数はインライン展開 2.4.4 使用頻度の高い関数を SWI 化