RX ファミリ用 C/C++コンパイラパッケージ CC-RX V3
プログラミング・テクニック編

R20AN0643JJ0101
Rev.1.01
2024.01.16

要旨

コードサイズ・実行速度・ROM データサイズに効果的なプログラミング方法を説明します。

動作確認デバイス

RX ファミリ　CC-RX V3.03

目次

1. はじめに

コードサイズ・実行速度・ROM データサイズに効果的なプログラミング方法として、以下の 3 つの項目に分けてそれぞれ説明します。

- オプション
- 拡張言語
- コーディングテクニック

本アプリケーションノート記載の測定結果、アセンブリ言語展開コードは CC-RX V3.03 を用いて取得しています。別段の記載が無い限り、-isa/-cpu オプションの指定値は -isa=rxv2 です。また、最適化オプションのデフォルト設定は以下のようになります。

サイズ重視の最適化	-size
最適化レベル	-optimize=2
ループ展開	-loop=2
インライン展開	-noinline
定数除算の乗算化	-noconst_div
命令並び替え	-schedule
const 修飾変数の定数伝播	-const_copy
最適化範囲分割	-scope
外部変数アクセス最適化	-nomap
ポインタ指示先の型を考慮した最適化	-alias=noansi

なお、これらのプログラミング方法の効果は前後に存在するプログラムや、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。

2. オプション

CC-RX のオプション指定によるコードサイズ・ROM データサイズ・実行速度への影響を示します。なお、効果の程度はソースプログラムの内容によって異なります。

2.1 コンパイル・オプション

○…改善する　×…劣化する　△…改善することもあれば劣化することもある　―…変化しない

( )…デフォルト設定

オプション	コードサイズ	ROMサイズ	サイクル数	備考
-instalign4	×	―	○	命令キューが 32 ビットの CPU（主に RX200 シリーズ）向けに、分岐先の命令実行向け整合を行います。
-instalign8	×	―	○	命令キューが 64 ビットの CPU（主に RX600 シリーズ）向けに、分岐先の命令実行向け整合を行います。
-nouse_div_inst	×	×	×	除算命令の生成を抑止することで、割り込み関数の応答性を改善します。コードサイズや実行速度は劣化します。
-stack_protector -stack_protector_all	×	―	×	関数の入口・出口にスタック破壊検出コードを生成します。検出コードの分、コードサイズと実行速度は劣化します。
-avoid_cross_boundary_prefetch	×	―	×	ストリング操作命令のデータプリフェッチで 4 バイト境界をまたぐ読み出しを行わないようコードを生成します。コードサイズと実行速度は劣化します。
-optimize	△	△	△	最適化レベルを指定します。デフォルトの指定は-optimize=2 です。
-goptimize	―	―	―	本オプションを指定したファイルは、リンク時にモジュール間最適化の対象になります。 リンク時の最適化については 2.3 リンク・オプション を参照ください。
-speed	×	―	○	
-size	（○）	―	（×）	
-loop	△	―	△	効果はパラメータに依存します。
-inline	△	―	△	効果はパラメータに依存します。
-case=ifthen	×	○	△	case ラベル数が多いほどコードサイズが増大します。
-case=table	○	×	△	case ラベルが多いほど ROM データサイズが増大しますが、サイクル数が常に一定です。
-volatile	×	―	×	

-const_copy	（ ○ ）	－	（ ○ ）	const 修飾型外部変数についても定数伝播を行い、コードサイズと実行速度を改善します。
-noconst_copy	×	－	×	-const_copy を無効にします。
-const_div	×	－	○	
-noconst_div	（ ○ ）	－	（ ○ ）	
-library=intrinsic	○	－	○	
-noscope	○	－	○	レジスタが不足すると性能が劣化する可能性があります。
-schedule	－	－	（ ○ ）	パイプライン処理を考慮した命令並べ替えを行い、実行速度を改善します。
-noschedule	－	－	×	-schedule を無効にします。
-map	○	－	○	
-smap	○	－	○	
-approxdiv	－	－	○	演算の精度と演算の順序が変わる場合があります。
-simple_float_conv	○	－	○	-isa=rxv1 の場合のみ有効です。
-nofpu	×	－	×	
-dpfpu	○	－	○	-ias=rxv3 の場合以外、エラーとなります。 -nofpu が指定されている場合、エラーになります。
-tfu=intrinsic, mathlib	○	－	○	三角関数演算器を利用したコードを生成します。三角関数演算器を利用した場合としない場合では、演算精度が異なります。
-alias=ansi	○	－	○	
-branch_chaining	○	－	×	コードサイズの小さな分岐命令を使用します。 -size 且つ-optimize=2 | max の場合にデフォルトで有効になります。
-nobranch_chaining	×	－	○	-branch_chaining を無効にします。
-ip_optimize	○	△	○	
-merge_files	○	△	○	
-whole_program	○	△	○	
-dbl_size=8	×	×	×	double 型を 8byte としてコンパイルします。
-int_to_short	△	○	△	int 型を short 型に置き換えてコンパイルします。
-auto_enum	△	○	△	
-pack	×	○	×	
-fint_register	△	－	△	高速割り込み関数でのみ使用する汎用レジスタを指定します。割り込み関数の性能が改善しますが、対して通常関数の性能は劣化します。
-branch=16	○	－	－	
-branch=32	×	－	×	

-base	△	―	△	
-nouse_pid_register	×	―	×	PID レジスタを使用せずにコード生成するため、コードサイズと実行速度は劣化します。
-save_acc	△	―	△	割り込み関数に対して、アキュムレータの退避・回復コードを生成します。アキュムレータを用いる命令を生成するようになりますが、割り込み関数の性能は劣化します。
-control_flow_integrity	×	×	×	間接関数呼び出しのチェック機能を生成します。チェック機能の分、コードサイズと実行速度、ROM サイズは劣化します。

2.1.1　-instalign4/-instalign8

分岐先の命令実行向け整合を行うかどうかを指定します。整合を行った場合、CPU の命令キューを効率よく動作させ、プログラムの実行速度を向上させることができますが、コードサイズが増大します。

-instalign4 を指定した場合、4 バイト単位の命令キューを持つ CPU の仕様に合わせて、命令の配置アドレスを整合します。主に RX100 シリーズ（RX110, RX111, RX113, RX130, RX13T グループ）及び RX200 シリーズ（RX210, RX21A, RX220 グループ）向けのオプションです。

-instalign8 を指定した場合、8 バイト単位の命令キューを持つ CPU の仕様に合わせて、命令の配置アドレスを整合します。主に RX600 シリーズ向けのオプションで、CPU として RXv2 以降を搭載した製品で速度向上を図る場合に適しています。

デバイスの命令キューの仕様は、ユーザーズマニュアル ハードウェア編に記載されています。

C ソース

```
long a;

int func1(int num)
{
  return (num + 1);
}

void func2(void)
{
  a += 1;
  a += a;
}

void main(void)
{
  unsigned int i;
  for (i = 0; i < 10; ++i) {
    if (func1(i) < 10) {
      func2();
    }
  }
  a += 1;
}
```

-cpu=rx200(命令キューが 32 ビットの CPU)

	-instalign4	オプション未指定
コードサイズ[バイト]	57	55
サイクル数[サイクル]	278	296

-cpu=rx600(命令キューが 64 ビットの CPU)

	-instalign8	オプション未指定
コードサイズ[バイト]	61	55
サイクル数[サイクル]	269	278

本機能は#pragma でも指定できます。オプションと#pragma の両方が指定された場合は、#pragma の指定を優先します。

例) –instalign4 指定時

```
void func1(void)            /* 4 バイトで整合(オプション指定) */
{
}

#pragma instalign8 func2
void func2(void)            /* 8 バイトで整合 */
{
}

#pragma noinstalign func3
void func3(void)            /* 整合なし */
{
}
```

例) -instalign8 指定時

```
void func1(void)            /* 8 バイトで整合(オプション指定) */
{
}

#pragma instalign4 func2
void func2(void)            /* 4 バイトで整合 */
{
}

#pragma noinstalign func3
void func3(void)            /* 整合なし */
{
}
```

2.1.2　-nouse_div_inst

プログラム中のすべての除算および剰余算を、DIV 命令、DIVU 命令および FDIV 命令を使わないコードを生成します。

DIV、DIVU および FDIV 命令を生成する代わりに、それぞれの命令に相当する処理を行うランタイム関数の呼び出しに置き換えます。このため、割り込み関数の応答性は 1～20 サイクル程度改善しますが、コードサイズや実行速度等のコード効率が劣化します。

C ソース

```
long a, b;
unsigned long c, d;
float e, f;
const float cf= 11.0;

void main(void)
{
  a = a / b;
  c = c / d;

  e = e / f;
  f = e / cf;
}
```

	オプション指定	オプション未指定
コードサイズ[バイト]	85	69
サイクル数[サイクル]	703	59

※オプション指定により ROM サイズも増大する可能性があります

2.1.3　-stack_protector/-stack_protector_all

関数の入口・出口にスタック破壊検出コードを生成します。

検知コードの分、コードサイズと実行速度が劣化します。

C ソース

```
#include <stdio.h>
#include <stdlib.h>

void func(void)
{
  volatile char str[10];
  int i;
  for (i = 0; i <= 9; i++) {
    str[i] = i;
  }
}

void __stack_chk_fail(void)
{
  /* stack is broken! */
  __brk();
}

void main(void)
{
  func();
}
```

	-stack_protector	-stack_protector_all	オプション未指定
コードサイズ[バイト]	44	65	24
サイクル数[サイクル]	75	82	68

本機能は#pragma でも指定できます。オプションと#pragma の両方が指定された場合は、#pragma の指定を優先します。

例) -stack_protector/-stack_protector_all 指定時

```
struct DATA
{
  int a, b, c, d;
};

struct DATA func1(void)    /* スタック破壊検出コードを生成する */
{
  struct DATA data = {0, 1, 2, 3};
  return data;
}

#pragma no_stack_protector (func2)
struct DATA func2(void)    /* スタック破壊検出コードを生成しない */
{
  struct DATA data = {0, 1, 2, 3};
  return data;
}
```

例) -stack_protector/-stack_protector_all 未指定時

```
struct DATA
{
  int a, b, c, d;
};

struct DATA func1(void)    /* スタック破壊検出コードを生成しない */
{
  struct DATA data = {0, 1, 2, 3};
  return data;
}

#pragma stack_protector (func2)
struct DATA func2(void)    /* スタック破壊検出コードを生成する */
{
  struct DATA data = {0, 1, 2, 3};
  return data;
}
```

2.1.4　-avoid_cross_boundary_prefetch

ストリング操作命令のデータプリフェッチで 4 バイト境界をまたぐ読み出しを防止します。

文字列操作用ライブラリ関数 memchr()、strlen()、strcpy()、strncpy()、strcmp()、strncmp()、strcat()、strncat()を-library=intrinsic を指定してコンパイルした際のコードサイズと実行速度が劣化します。

C ソース

```
#include <string.h>

unsigned long len;

void main(void)
{
  char str[] = "abcdefghijklmnopqrstuvwxyz";
  len = strlen(str);
}
```

	オプション指定	オプション未指定
コードサイズ[バイト]	50	43
サイクル数[サイクル]	79	73

2.1.5　-optimize

最適化レベルを指定します。

デフォルト設定は-optimize=2 です。

-size/-speed オプションの選択と、最適化レベルの選択、及び入力ソースの種類(C/C++)によって、以下オプションのデフォルト指定が変化します。

項目	オプション
ループ展開	-loop
インライン展開	-inline / -noinline
定数除算の乗算化	-const_div / -noconst_div
命令並び替え	-schedule / -noschedule
const 修飾変数の定数伝播	-const_copy / -noconst_copy
最適化範囲分割	-scope / -noscope
外部変数アクセス最適化	-map / -smap / -nomap
ポインタ指示先の型を考慮した最適化	-alias=ansi / -alias=noansi

C ソース

```
int i = 0;
int x[10], y[10];

static void sub(int* a, int* b, int i)
{
  int temp;
  temp = a[i];
  a[i] = b[i];
  b[i] = temp;
}

void main(void)
{
  sub(x, y, i);
}
```

	-optimize=0	-optimize=1	-optimize=2	-optimize=max
コードサイズ[バイト]	63	37	35	24
サイクル数[サイクル]	36	20	14	10

※オプション指定により ROM サイズも増大する可能性があります

2.1.6　-speed/-size

最適化で重視するポイントを設定します。

-speed オプションを指定した場合、実行速度重視の最適化を実施します。

-size オプションを指定した場合、コードサイズ重視の最適化を実施します。(デフォルト)

C ソース

```
long a;

void main(void)
{
  unsigned long i = 0;
  unsigned long j = 0;
  for (i = 0; i < 5; ++i) {
    for (j = 0; j < 5; ++j) {
      a += (i + j);
      a *= (i + j);
    }
  }
}
```

	-speed	-size
コードサイズ[バイト]	66	47
サイクル数[サイクル]	165	245

2.1.7　-loop

ループ展開の最適化を行うかどうかを指定します。

ループ展開を行うとコードサイズは増大しますが、実行速度は向上します。

C ソース

```
long val;

void main(void)
{
  unsigned long i, j, k, l;
  for (i = 1; i < 7; ++i) {
    for (j = 1; j < 6; ++j) {
      for (k = 1; k < 5; ++k) {
        for (l = 1; l < 4; ++l) {
          val += (i + j + k);
          val *= (i + j + k);
        }
      }
    }
    val += (i * 10);
  }
}
```

	-loop=1	-loop=2	-loop=8
コードサイズ[バイト]	154	204	307
サイクル数[サイクル]	4391	3424	3069

※-loop=8 は-optimize=max/-speed 指定時のデフォルト値です

2.1.8　-inline

関数の自動インライン展開を行うかどうかを指定します。

-inline を指定した場合、関数の自動インライン展開を行います。

-noinline を指定した場合、関数の自動インライン展開を行いません。

オプション-inline はパラメータを指定することも可能です。パラメータで、関数サイズが何%増加するまでインライン展開を行うかを指定できます。例えば、inline=100 を指定した場合、コード全体サイズが 100%増加するまで（2 倍まで）インライン展開を行います。また inine=0 を指定した場合はサイズが増えない、あるいは減る場合のみインライン展開を行います。

C ソース

```
long val;
long x[1000];

static void func1(void)
{
  ++val;
}
void func2(int a)
{
  if (a) {
    x[a] = 0;
  }
}

void main(void)
{
  signed int i;
  func2(val);
  for (i = 0; i < 10; ++i) {
    func2(i);
    func1();
    func2(val);
  }
  func2(val);
}
```

	-inline=200	-inline=0	-noinline
コードサイズ[バイト]	93	58	70
サイクル数[サイクル]	153	364	474

尚、インライン展開の機能は#pragma でも指定できます。オプションと#pragma の両方が指定された場合は、#pragma の指定を優先します。

例) -inline 指定時

```
void func1(void)            /* 自動インライン展開を行います(オプション指定) */
{
}

#pragma noinline(func2)
void func2(void)            /* インライン展開されません */
{
}
```

例) -noinline 指定時

```
void func1(void)            /* 自動インライン展開を行いません(オプション指定) */
{
}

#pragma inline(func2)
void func2(void)            /* インライン展開します */
{
}
```

2.1.9　-case

switch 文のコード展開方式を指定します。

-case=ifthen を指定した場合、if_then 方式で展開します。case ラベル数が多いほどコードサイズが増大します。case ラベルによって実行速度が変化します。

-case=table を指定した場合、テーブル方式で展開します。case ラベル数が多いほど ROM データサイズが増大しますが、実行速度が常に一定です。

-case=auto を指定した場合、if_then 方式とテーブル方式のどちらにするかコンパイラが自動的に選択します。(デフォルト設定)

C ソース

```
long val = 10;

void main(void)
{
  switch (val) {
  case 1:
    val += 10;
    break;
  case 2:
    val *= 10;
    break;
  case 3:
    val /= 10;
    break;
  default:
    val -= 10;
    break;
  }
}
```

	-case=ifthen	-case=table
コードサイズ[バイト]	37	43
ROM サイズ[バイト]	4	7
サイクル数[サイクル]	15	13

※上記 C ソース例の場合 -case=auto では if_then 方式が選択されます

2.1.10 -volatile

全ての外部変数を volatile 宣言したものとして扱うかどうか選択します。

-volatile を指定した場合、すべての外部変数を volatile 宣言したものとして扱います。したがって、外部変数のアクセス回数、アクセス順序は C/C++言語ソースファイルで記述した通りになりますが、外部変数に対する最適化が抑止されるため、コードサイズと実行速度は劣化します。

C ソース

```
long val = 0;

void main(void)
{
  val += 1;
  val -= 2;
  val *= 3;
  val /= 4;
}
```

	オプション指定	オプション未指定
コードサイズ[バイト]	33	19
サイクル数[サイクル]	22	12

2.1.11 -const_copy

const 修飾外部変数の定数伝播を行うかどうか選択します。定数伝播を行うと、実行速度が向上します。

-const_copy を指定した場合、const 修飾型外部変数についても定数伝播を行います。(デフォルト設定)

-noconst_copy を指定した場合、const 修飾型外部変数の定数伝播を抑止します。

C ソース

```
const long val = 0;
long result;

void main(void)
{
  result = val + 10;
}
```

	-const_copy	-noconst_copy
コードサイズ[バイト]	10	19
サイクル数[サイクル]	5	8

2.1.12 -const_div/-noconst_div

整数定数による除算や剰余算を、乗算やビット演算(シフトやビット論理積)を用いた命令列に変換するかどうか選択します。変換する場合、コードサイズが増大しますが、実行速度が向上します。

-const_div を指定した場合、ソースファイル中の整数型定数による除算および剰余算を、乗算やビット演算を用いた命令列に変換します。-size オプション指定時に本オプションを指定すると、-noconst_div を指定した場合に比べて実行速度が向上します。

-noconst_div を指定した場合、ソースファイル中の整数型定数による除算および剰余算に、それぞれに対応する除算命令および剰余算命令(2 のべき乗定数値による符号なし整数の除算および剰余算を除く)を用います。-speed オプション指定時に本オプションを指定すると、-const_div を指定した場合に比べてコードサイズを削減できます。

C ソース

```
long a = 0x7FFFFFFF;

void main(void)
{
  a = a / 1000;
}
```

	-const_div (-size)	-const_div (-speed)	-noconst_div (-size/-speed)
コードサイズ[バイト]	26	27	16
サイクル数[サイクル]	13	12	23

2.1.13 -library

一部の標準ライブラリ関数のコード展開方式を指定します。

-library=function を指定した場合、ライブラリ関数をすべて関数呼び出しします。コードサイズと実行速度が劣化します。

-library=intrinsic を指定した場合、abs()、fabsf()およびストリング操作命令が使用できるライブラリ関数を命令展開します。(デフォルトの指定) また、-library=intrinsic とともに-isa=rxv2 が指定されている場合は、sqrtf()または-dbl_size=4 指定時の sqrt()のそれぞれの呼び出しを FSQRT 命令に置き換えます。ただし、本オプションで命令に展開された場合は、変数 errno の内容を変更しません。

C ソース

```
#include <stdlib.h>
int a;

void main(void)
{
  a = abs(a);
}
```

	-library=function	-library=intrinsic
コードサイズ[バイト]	19	13
サイクル数[サイクル]	15	8

2.1.14 -scope/-noscope

コンパイル時の最適化範囲を分割するかどうか選択します。

-noscope を指定した場合、最適化範囲を分割せずにコンパイルします。最適化範囲が広がることによりコンパイル速度は遅くなりますが、一般的にはオブジェクト性能が改善します。ただし、空レジスタの数が不足するとオブジェクト性能が劣化する場合があります。

-scope を指定した場合、サイズの大きい関数について、最適化範囲を複数に分割してコンパイルします。

C ソース

```
long array[40];
long val = 10;
void main(void)
{
  int i;
  for (i = 0; i < 40; ++i) {
    array[i] = val * i;
  }
  for (i = 0; i < 40; ++i) {
    if (array[i] > i) {
      array[i] += val + i;
    }
    else if (array[i] > (i * 2)) {
      array[i] += val + (i * 2);
    }
    else if (array[i] > (i * 3)) {
      array[i] += val + (i * 3);
    }
    else if (array[i] > (i * 4)) {
      array[i] += val + (i * 4);
    }
    else if (array[i] > (i * 5)) {
      array[i] += val + (i * 5);
    }
    else if (array[i] > (i * 6)) {
      array[i] += val + (i * 6);
    }
    else if (array[i] > (i * 7)) {
      array[i] += val + (i * 7);
    }
    else if (array[i] > (i * 8)) {
      array[i] += val + (i * 8);
    }
    else if (array[i] > (i * 9)) {
      array[i] += val + (i * 9);
    }
    else {
      array[i] += val + (i * 10);
    }
  }
}
```

	-scope	-noscope
コードサイズ[バイト]	318	312
サイクル数[サイクル]	1089	1046

※ループ展開(-loop=2)を指定

2.1.15 -schedule/-noschedule

パイプライン処理を考慮した命令並べ替えを行うかどうかを選択します。並べ替えを行う場合、実行速度の向上が期待できます。

-schedule を指定した場合、パイプライン処理を考慮した命令並べ替えを行います。-optimize=2 または-optimize=max オプションを指定した場合は-schedule です。

-noschedule を指定した場合、命令並べ替えを行いません。基本的に C/C++言語ソースファイルで記述した順番で処理を行います。-optimize=1 または-optimize=0 オプションを指定した場合は-noschedule です。

C ソース

```
long a, b;
unsigned long c, d;
float e, f;

void main(void)
{
  a = a + b;
  c = c + d;
  e = e + f;
}
```

	-schedule	-noschedule
コードサイズ[バイト]	58	58
サイクル数[サイクル]	24	25

2.1.16 -map/-smap

外部変数アクセス最適化を行うかどうか選択します。本最適化を行うとコードサイズや実行速度といったコード効率が改善します。

-map を指定した場合、外部変数アクセス最適化を行います。最適化リンケージエディタが生成する外部シンボル割り付け情報を元にベースアドレスを設定し、外部変数もしくは静的変数のアクセスをベースアドレス相対で行うコードを生成します。

-smap を指定した場合、コンパイル対象ファイル内で定義された外部変数もしくは静的変数についてベースアドレスを設定し、アクセスをベースアドレス相対で行うコードを生成します。

C ソース [tp1.c]

```
long a, b, c;
extern long d, e, f;
void main(void)
{
  a = d;
  b = e;
  c = f;
}
```

C ソース [tp2.c]

```
long d = 10;
long e = 10;
long f = 10;
```

	-map	-smap	オプション未指定
コードサイズ[バイト]	18	33	43
サイクル数[サイクル]	13	14	16

2.1.17 -approxdiv

浮動小数点定数除算を、定数の逆数の乗算に変換します。浮動小数点定数除算の実行速度は向上しますが、演算の精度と順序が変わる場合がありますので注意が必要です。

C ソース

```
float a;

void main(void)
{
  a /= 1.1;
}
```

	オプション指定	オプション未指定
コードサイズ[バイト]	18	18
サイクル数[サイクル]	9	23

2.1.18 -simple_float_conv

浮動小数点型の型変換処理の一部を省略します。生成する命令コードの命令セットアーキテクチャが RXv1 の時のみ有効です。

次の場合において、浮動小数点の型変換を行う生成コードが変化します。

a) 32bit 浮動小数点型から符号無し整数型への変換

b) 符号無し整数型から 32bit 浮動小数点型への変換

c) 32bit 浮動小数点型を経由した、整数型から 64bit 浮動小数点型への変換(-optimize=0 のときは無効)

コードサイズや実行速度といったコード効率は改善しますが、変換結果が C/C++言語規格と異なる場合がありますので、ご注意ください。

C ソース

```
unsigned long isrc = 2;
float fsrc = 2.0;
unsigned long idst;
float fdst;

void main(void)
{
  idst = (unsigned long)fsrc;
  fdst = (float)isrc;
}
```

	オプション指定	オプション未指定
コードサイズ[バイト]	36	73
サイクル数[サイクル]	17	23

※RXv1(-isa=rxv1)を指定

2.1.19 -nofpu

FPU 命令を使用したコードを生成するかどうかを選択します。

-nofpu を指定した場合、FPU 命令を使用しないコード生成を行います。

-fpu を指定した場合、FPU 命令を使用したコード生成を行います。

デフォルトの指定は-fpu です。ただし、-cpu=rx200 を指定した場合は-nofpu が指定されます。-cpu=rx200 と-fpu を同時に指定することはできません。

C ソース

```
float a, b;
const float c = 11.0;

void main(void)
{
  a /= b;
  b /= c;
}
```

	-fpu	-nofpu
コードサイズ[バイト]	31	45
サイクル数[サイクル]	40	98

2.1.20 -dpfpu

倍精度浮動小数点処理命令を使用したコード生成を行います。生成する命令コードの命令セットアーキテクチャが RXv3 の時のみ有効です。

C ソース

```
double val;

void main(void)
{
  val /= val;
  val *= val;
}
```

	-dpfpu	-nodpfpu
コードサイズ[バイト]	29	35
サイクル数[サイクル]	45	83

※RXv3(-isa=rxv3)、-dbl_size=8 を指定

2.1.21 -tfu=intrinsic, mathlib

-tfu=intrinsic,mathlib を指定した場合、数学ライブラリ関数の呼び出しを三角関数演算器を利用するコードに置き換えます。

三角関数演算器を利用した演算処理は、リエントラント性がありません。

数学ライブラリ関数の置き換えは関数呼び出しコード自体を置き換えるものであり、ライブラリ内のコードは変わりません。ポインタを使った間接呼び出しがされた場合はライブラリ関数が呼び出されるため、三角関数演算器を利用することができません。

数学ライブラリ関数の呼び出しが三角関数演算器を利用したコードに置き換わった場合、変数 errno の内容は変更されません。

三角関数演算器を利用した場合と利用しない場合では、演算精度が異なります。

三角関数演算器を利用する前にスタートアッププログラム等で組み込み関数__init_tfu() を用いて演算器を初期化してください。初期化を行わなかった場合の動作は保証しません。

三角関数演算器を搭載していないデバイスでは本オプションは指定しないでください。

C ソース

```
float val;

void main(void)
{
  val = sinf(val);
}
```

	-tfu=intrinsic	-tfu=intrinsic, mathlib
コードサイズ[バイト]	19	16
サイクル数[サイクル]	29	22

※__init_tfu()の実行時間は含みません

2.1.22 -alias

ポインタ指示先の型を考慮した最適化を実施するかどうかを選択します。最適化を行うと、コードサイズや実行速度といったコード効率が改善します。但し、ANSI 規格に則った C ソースでない場合、実行結果が期待した値とならない可能性があります。

-alias=ansi を指定した場合、ANSI 規格に基づき、ポインタ指示先の型を考慮した最適化を行います。一般には、-alias=noansi を指定した場合よりもオブジェクト性能が向上しますが、-alias=ansi と -alias=noansi とで実行結果が異なる場合があります。

-alias=noansi を指定した場合、ANSI 規格に基づくポインタ指示先の型を考慮した最適化を行いません。

C ソース

```
long a, b;
short* ps;

void main(void)
{
  a = 1;
  *ps = 2;
  b = a + *ps;
}
```

	-alias=ansi	-alias=noansi
コードサイズ[バイト]	30	36
サイクル数[サイクル]	10	16

2.1.23 -branch_chaining / -nobranch_chaining

コードサイズの小さな分岐命令を使用します。コードサイズの小さな分岐命令を使用するために、最終的な分岐先まで直接分岐するのではなくて、分岐先を、行き先が同じ他の分岐命令にすることがあります。

C ソース

```
int x = 1;
int data[1000];

#pragma inline_asm sub
void sub() {}

void main(void)
{
  int i;
  switch (x) {
    default : for (i = 0; i<32;++i) {data[i] = i;} break;
    case 1 : for (i = (32*1); i<(32*2);++i) {data[i] = i;} break;
    case 2 : for (i = (32*2); i<(32*3);++i) {data[i] = i;} break;
    case 3 : for (i = (32*3); i<(32*4);++i) {data[i] = i;} break;
    case 4 : for (i = (32*4); i<(32*5);++i) {data[i] = i;} break;
    case 5 : for (i = (32*5); i<(32*6);++i) {data[i] = i;} break;
    case 6 : for (i = (32*6); i<(32*7);++i) {data[i] = i;} break;
    case 7 : for (i = (32*7); i<(32*8);++i) {data[i] = i;} break;
    case 8 : for (i = (32*8); i<(32*9);++i) {data[i] = i;} break;
    case 9 : for (i = (32*9); i<(32*10);++i) {data[i] = i;} break;
  }
  sub();
}
```

	-branch_chaining	-nobranch_chaining
コードサイズ[バイト]	1566	1567
サイクル数[サイクル]	53	50

※-loop=32 を指定

2.1.24 -ip_optimize

大域最適化(手続き間別名解析を利用した最適化や引数・戻り値の定数伝播など)を実施するかどうかを選択します。

C ソース

```
long result;

static long func(long x, long y, long z)
{
  return (x - y + z);
}
void main(void)
{
  result = func(3, 4, 5);
}
```

	オプション指定	オプション未指定
コードサイズ[バイト]	21	23
サイクル数[サイクル]	17	19

2.1.25 -merge_files

複数の C ソースファイルをマージしてコンパイルすることにより、1 つのオブジェクトファイルを出力します。

-merge_files は-ip_optimize と同時に指定すると相乗効果を得ることが出来ます。

C ソース [tp1.c]

```
long result;

void main(void)
{
  result = func(3, 4, 5);
}
```

C ソース [tp2.c]

```
#pragma inline (func)
long func(long x, long y, long z)
{
  return (x - y + z);
}
```

	オプション指定	オプション未指定
コードサイズ[バイト]	122	131
サイクル数[サイクル]	5	18

※ROM サイズも改善する場合があります
※サイズはスタートアップを含みます

2.1.26 -whole_program

コンパイル対象ファイルがプログラム全体であることを仮定し、コンパイル対象ファイル全てをマージして大域最適化を実施します。

本オプションを指定した場合、コンパイラは以下の条件を満たすことを前提にコンパイルを行います。条件を満たさない場合の動作は保証しません。

(条件 1) コンパイル対象ファイル内で定義された extern 変数の内容及びアドレスが、コンパイル対象ファイル以外で設定及び参照されることはない。

(条件 2) コンパイル対象ファイル内からコンパイル対象ファイル以外の関数を呼び出したとしても、呼び出された関数からコンパイル対象ファイル内の関数が呼ばれることはない。

C ソース [tp1.c]

```
extern const int c;
int result;

int func(void);

void main(void)
{
  result = c;
  result += func();
}
```

C ソース [tp2.c]

```
const int c = 1;
int x = 10;
int *p;

int func(void)
{
  int i;
  for (i = 0; i < x; ++i) {
    (*p) += c;
  }
  return (*p);
}
```

	オプション指定	オプション未指定
コードサイズ[バイト]	160	171
サイクル数[サイクル]	86	162

※ROM サイズも改善する場合があります
※サイズはスタートアップを含みます

2.1.27 -dbl_size

double 型および long double 型を float 型に変更するかどうかを指定します。

-dbl_size=4 を指定した場合、float 型に変更します。(デフォルト指定)

-dbl_size=8 を指定した場合、float 型に変更しません。

C ソース

```
double a, b;
const double c = 11.0;

void main(void)
{
  a = a / b;
  b = b / c;
}
```

	-dbl_size=4	-dbl_size=8
コードサイズ[バイト]	31	63
ROM サイズ[バイト]	4	8
サイクル数[サイクル]	40	106

2.1.28 -int_to_short

ソースファイル内の int を short に、unsigned int を unsigned short に置換してコンパイルを行います。

int、unsigned int のサイズを 32bit と仮定した移植性のないプログラムでは、実行結果が変化する可能性があります。

C ソース

```
int x;
long y;
const int imm = 1;

void main(void)
{
  x += imm;
  y += x;
}
```

	オプション指定	オプション未指定
コードサイズ[バイト]	26	24
ROM サイズ[バイト]	2	4
サイクル数[サイクル]	13	12

2.1.29 -auto_enum

enum 宣言した列挙型のデータを、列挙値が収まる最小型として処理します。

ROM データサイズが小さくなりますが、enum の値を long 型等に拡張して使用する場合、コードサイズと実行速度が変化する場合もあります。

C ソース

```
enum number {
  one   = 1,
  two   = 2,
  three = 3
};

int x;
enum number num;
const enum number DATA = one;

void main(void)
{
  num += num;
  x += num;
}
```

	オプション指定	オプション未指定
コードサイズ[バイト]	26	24
ROM サイズ[バイト]	1	4
サイクル数[サイクル]	13	12

2.1.30 -pack

構造体メンバ、クラスメンバのアライメント数を指定します。

-pack を指定した場合、構造体、クラスのアライメント数が 1 になり、ROM データサイズが小さくなりますが、アライメント調整が必要になる場合、コードサイズと実行速度が劣化します。

-unpack を指定した場合、構造体、クラスのアライメント数が、メンバの最大のアライメント数と同じになります。(デフォルト設定)

C ソース

```
struct DATA
{
  char c;
  long l;
};

struct DATA data = {1, 2};
long result;

long func(void)
{
  return (data.l);
}

void main(void)
{
  result = func();
}
```

	-pack	-unpack
コードサイズ[バイト]	23	21
ROM サイズ[バイト]	5	8
サイクル数[サイクル]	18	16

本機能は#pragma でも指定できます。オプションと#pragma の両方が指定された場合は、#pragma の指定を優先します。

例) -pack 指定時

```
struct DATA1    /* pack 処理はオプション指定に従う */
{
  char a;       /* バイトオフセット = 0 */
  long b;       /* バイトオフセット = 1 */
  short c;      /* バイトオフセット = 5 */
} data1;        /* 合計サイズ 7 バイト */

#pragma unpack
struct DATA2    /* pack 処理を行わない */
{
  char a;       /* バイトオフセット = 0 */
  long b;       /* バイトオフセット = 4 */
  short c;      /* バイトオフセット = 8 */
} data2;        /* 合計サイズ 12 バイト */

#pragma packoption
struct DATA3    /* pack 処理はオプション指定に従う */
{
  char a;       /* バイトオフセット = 0 */
  long b;       /* バイトオフセット = 1 */
  short c;      /* バイトオフセット = 5 */
} data3;        /* 合計サイズ 7 バイト */
```

例) -unpack 指定時

```
struct DATA1    /* pack 処理はオプション指定に従う */
{
  char a;       /* バイトオフセット = 0 */
  long b;       /* バイトオフセット = 4 */
  short c;      /* バイトオフセット = 8 */
} data1;        /* 合計サイズ 12 バイト */

#pragma pack
struct DATA2    /* pack 処理を行う */
{
  char a;       /* バイトオフセット = 0 */
  long b;       /* バイトオフセット = 1 */
  short c;      /* バイトオフセット = 5 */
} data2;        /* 合計サイズ 7 バイト */

#pragma packoption
struct DATA3    /* pack 処理はオプション指定に従う */
{
  char a;       /* バイトオフセット = 0 */
  long b;       /* バイトオフセット = 4 */
  short c;      /* バイトオフセット = 8 */
} data3;        /* 合計サイズ 12 バイト */
```

2.1.31 -fint_register

高速割り込み関数（#pragma interrupt で割り込み仕様に高速割り込み指定（fint）のある関数）でのみ使用する汎用レジスタを指定します。高速割り込み関数以外では、指定されたレジスタは使用しません。本オプションで指定した汎用レジスタは、高速割り込み関数内では退避回復なしで使用できるため、高速割り込み関数のさらなる高速化が見込めます。反面、他の関数で使用可能な汎用レジスタが減るため、プログラム全体のレジスタ割り付け効率は低下します。

オプション	高速割り込み専用レジスタ
fint_register=0 (デフォルト設定)	なし
fint_register=1	R13
fint_register=2	R12、R13
fint_register=3	R11、R12、R13
fint_register=4	R10、R11、R12、R13

C ソース [通常関数]

```
long val[40];
long tmp = 10;

void main(void)
{
  int i;
  for (i = 0; i < 40; ++i) {
    if (tmp > i) {
      val[i] = tmp + i;
    }
    else if (tmp > (i * 2)) {
      val[i] = tmp + (i * 2);
    }
    else if (tmp > (i * 3)) {
      val[i] = tmp + (i * 3);
    }
    else if (tmp > (i * 4)) {
      val[i] = tmp + (i * 4);
    }
    else if (tmp > (i * 5)) {
      val[i] = tmp + (i * 5);
    }
    else if (tmp > (i * 6)) {
      val[i] = tmp + (i * 6);
    }
    else if (tmp > (i * 7)) {
      val[i] = tmp + (i * 7);
    }
    else if (tmp > (i * 8)) {
      val[i] = tmp + (i * 8);
    }
    else if (tmp > (i * 9)) {
      val[i] = tmp + (i * 9);
    }
    else {
      val[i] = tmp + (i * 10);
    }
  }
}
```

	fint_register=0	fint_register=1	fint_register=2	fint_register=3	fint_register=4
コードサイズ[バイト]	153	153	153	161	170
サイクル数[サイクル]	1654	1654	1654	1734	1873

C ソース [割り込み関数]

```
volatile int count;

#pragma interrupt int_func(vect=10, fint)
void int_func(void)
{
  count++;
}
```

	fint_register=0	fint_register=1	fint_register=2	fint_register=3	fint_register=4
コードサイズ[バイト]	18	18	14	14	14
サイクル数[サイクル]	12	10	8	8	8

2.1.32 -branch

別セクション、別ファイルに定義した関数への分岐幅を指定します。

-branch=16 を指定した場合、分岐幅が 16bit 以内であるとしてコンパイルします。

-branch=24 を指定した場合、分岐幅が 24bit 以内であるとしてコンパイルします。(デフォルト設定)

-branch=32 を指定した場合、分岐幅を指定しません。

C ソース

```
void sub(void);

void main(void)
{
  sub();
}

#pragma section ResetPRG
void sub(void)
{
}
```

	-branch=16	-branch=24	-branch=32
コードサイズ[バイト]	3	4	8
サイクル数[サイクル]	6	6	7

2.1.33 -base

プログラム全体で、ベースアドレスとして固定で使用する汎用レジスタを指定します。

コンパイル対象とするプログラムによって、コードサイズや実行速度といったコード効率が改善する可能性があります。

C ソース

```
long val[40];
long tmp = 10;

void main(void)
{
  int i;
  for (i = 0; i < 40; ++i) {
    if (tmp > i) {
      val[i] = tmp + i;
    }
    else if (tmp > (i * 2)) {
      val[i] = tmp + (i * 2);
    }
    else if (tmp > (i * 3)) {
      val[i] = tmp + (i * 3);
    }
    else if (tmp > (i * 4)) {
      val[i] = tmp + (i * 4);
    }
    else if (tmp > (i * 5)) {
      val[i] = tmp + (i * 5);
    }
    else if (tmp > (i * 6)) {
      val[i] = tmp + (i * 6);
    }
    else if (tmp > (i * 7)) {
      val[i] = tmp + (i * 7);
    }
    else if (tmp > (i * 8)) {
      val[i] = tmp + (i * 8);
    }
    else if (tmp > (i * 9)) {
      val[i] = tmp + (i * 9);
    }
    else {
      val[i] = tmp + (i * 10);
    }
  }
}
```

	オプション指定 (-base=ram=R8)	オプション未指定
コードサイズ[バイト]	165	166
ROM サイズ[バイト]	4	4
サイクル数[サイクル]	1533	1540

2.1.34 -nouse_pid_register

PID レジスタを使用せずにコード生成を行います。

コードサイズと実行速度が劣化する可能性があります。

C ソース

```
long val[40];
long tmp = 10;

void main(void)
{
  int i;
  for (i = 0; i < 40; ++i) {
    if (tmp > i) {
      val[i] = tmp + i;
    }
    else if (tmp > (i * 2)) {
      val[i] = tmp + (i * 2);
    }
    else if (tmp > (i * 3)) {
      val[i] = tmp + (i * 3);
    }
    else if (tmp > (i * 4)) {
      val[i] = tmp + (i * 4);
    }
    else if (tmp > (i * 5)) {
      val[i] = tmp + (i * 5);
    }
    else if (tmp > (i * 6)) {
      val[i] = tmp + (i * 6);
    }
    else if (tmp > (i * 7)) {
      val[i] = tmp + (i * 7);
    }
    else if (tmp > (i * 8)) {
      val[i] = tmp + (i * 8);
    }
    else if (tmp > (i * 9)) {
      val[i] = tmp + (i * 9);
    }
    else {
      val[i] = tmp + (i * 10);
    }
  }
}
```

	オプション指定	オプション未指定
コードサイズ[バイト]	167	158
サイクル数[サイクル]	1823	1674

※-fint_register=3 で測定

2.1.35 -save_acc

割り込み関数に対して、アキュムレータ（ACC, ACC0, ACC1）の退避・回復コードを生成します。

本オプション指定時は、割り込み時でもアキュムレータの値が保持されるため、C/C++の演算式に対して MACLO 命令などのアキュムレータを用いる命令を生成することがあります。

C ソース [通常関数]

```
short src1[3] = {10, 11, 12};
short src2[3] = {20, 21, 22};
int result;

int func(const short* src1, const short* src2)
{
  return (src1[0] * src2[0]) + (src1[1] * src2[1]) + (src1[2] * src2[2]);
}

void main(void)
{
  result = func(src1, src2);
}
```

	オプション指定	オプション未指定
コードサイズ[バイト]	49	49
サイクル数[サイクル]	25	27

※オプション-speed を指定

C ソース [割り込み関数]

```
#include <machine.h>

long src1 = 10;
long src2 = 20;
long result;

#pragma interrupt func(vect=10)
void func(void)
{
  result = src1 * src2;
}

void main(void)
{
  int_exception(10);
  nop();
}
```

	オプション指定	オプション未指定
コードサイズ[バイト]	62	32
サイクル数[サイクル]	32	18

※割り込み関数のみのサイズ・サイクル数

2.1.36 -control_flow_integrity

間接関数呼び出しに対して、呼び出し先関数のチェックを行います。

チェックのためのコードとデータが追加されるため、コードサイズ、ROM データサイズ、実行速度が劣化します。

C ソース

```
void __control_flow_chk_fail(void) {}

void func1(char a) {}
void func2(long b) {}
void func3(void){}
void (*p1)(char a) = func1;
void (*p2)(long b) = func2;

void func4(void)
{
  func3();
}

void main(void)
{
  p1(1);
  p2(1);
  func4();
}
```

	オプション指定	オプション未指定
コードサイズ[バイト]	192	147
ROM サイズ[バイト]	176	64
サイクル数[サイクル]	82	36

※コードサイズおよび ROM サイズはスタートアップを含みます

2.2　アセンブル・オプション

○…改善する　×…劣化する　△…状況に依存する　—…変化しない

オプション	コード サイズ	ROM サイズ	サイクル 数	備考
-goptimize	—	—	–	本オプションを指定したファイルは、リンク時にモジュール間最適化の対象になります。 リンク時の最適化については 2.3 リンク・オプション を参照ください。

2.3　リンク・オプション

リンカの最適化オプション指定によるコードサイズ・ROM データサイズ・実行速度の影響を示します。コンパイル、アセンブル時に-goptimize を指定したファイルに対して最適化を行います。

-section_forbid で指定したセクションの最適化を抑止することができます。

また、-absolute_forbid で指定したアドレス＋サイズの範囲の最適化を抑止することができます。

〇…改善する　×…劣化する　△…状況に依存する　―…変化しない

オプション	コード サイズ	ROM サイズ	サイクル 数	備考
-optimize=symbol_delete	〇	〇	―	
-optimize=same_code	〇	―	×	
-optimize=short_format	〇	―	―	
-optimize=branch	〇	―	―	

※コマンドラインから起動した場合、デフォルトでは全ての最適化を実行します。

2.3.1　-optimize=symbol_delete

1 度も参照のない変数/関数を削除します。必ずコンパイル時に#pragma entry を指定するか、リンカの-entry で entry シンボルを指定してください。

-symbol_forbid で指定した変数/関数の削除を抑止することができます。

C ソース

```
int value1 = 0;
int value2 = 0;

void func1(void)
{
  value1++;
}

void func2(void)
{
  value2++;
}

void main(void)
{
  func1();
}
```

	オプション指定	オプション未指定
コードサイズ[バイト]	89	135
ROM サイズ[バイト]	60	64

※コードサイズおよび ROM サイズはスタートアップを含みます

2.3.2　-optimize=same_code

複数の同一命令列をサブルーチン化します。

サブルーチン化によりコードサイズは改善しますが、実行速度は劣化します。

-samesize で最適化対象となる最小コードサイズを指定できます。(デフォルトは-samesize=1E)

C ソース

```
volatile int value = 0;

int v1;
int v2;
int v3;
int v4;
int v5;

void sub(void)
{
  value += v1;
  value += v2;
  value += v3;
  value += v4;
  value += v5;
}

void main(void)
{
  value += v1;
  value += v2;
  value += v3;
  value += v4;
  value += v5;

  sub();
}
```

	オプション指定	オプション未指定
コードサイズ[バイト]	188	253
サイクル数[サイクル]	80	68

※コードサイズはスタートアップを含みます

2.3.3　-optimize=short_format

ディスプレースメント/イミディエートのコードサイズを短縮可能な場合、コードサイズがより小さくなる命令に置き換えます。

C ソース

```
volatile int value = 0;

int v1;
int v2;
int v3;
int v4;
int v5;

void main(void)
{
  value += v1;
  value += v2;
  value += v3;
  value += v4;
  value += v5;
}
```

	オプション指定	オプション未指定
コードサイズ[バイト]	162	179

※コードサイズはスタートアップを含みます

2.3.4　-optimize=branch

プログラムの配置情報に基づいて、分岐命令サイズを最適化します。

C ソース

```
extern void sub(void);

void main(void)
{
  sub();
  sub();
  sub();
  sub();
  sub();
}
```

	オプション指定	オプション未指定
コードサイズ[バイト]	135	143

※コードサイズはスタートアップを含みます

3.　拡張言語

拡張言語の中で#pragma 指令によるコードサイズ・ROM データサイズ・実行速度への影響を示します。

3.1　#pragma 指令

〇…改善する　×…劣化する　△…状況に依存する　―…変化しない

#pragma 指令	コードサイズ	ROM サイズ	サイクル数	備考
#pragma interrupt	△	―	△	割り込み仕様(fint など)を変更することで、割り込み関数の性能が改善する場合があります。

3.1.1　#pragma interrupt

割り込み関数となる関数を宣言します。割り込み仕様を変更することで、割り込み関数の速度・サイズが改善する場合があります。

割り込み仕様	形式	指定内容
高速割り込み	fint	高速割り込みに使用する関数を指定します。
割り込み関数レジスタ制限指定	save	割り込み関数内で使用するレジスタの本数を R1～R5、および R14～R15 の 7 本に制限します。これらの退避回復コードを生成しない代わりに割り込み関数内で使用もしないため、劣化する場合があります。
多重割り込み許可指定	enable	割り込み関数の先頭で PSW の I フラグを 1 にし、多重割り込みを許可します。
アキュムレータ保存指定	acc	アキュムレータを退避および回復する命令を生成します。

C ソース

```
long count;
long l1, l2;

#pragma interrupt int_func
void int_func(void)
{
  count = l1 * l2;
}
```

このC ソースに対して、割り込み仕様を指定しない場合と、fint/save/enable/acc のそれぞれを指定した場合を比較すると次の表の通りになります。

	未指定	fint	save	enable	acc
コードサイズ[バイト]	31	31	31	33	57
サイクル数[サイクル]	18	15	18	19	32

高速割り込み(fint)は、-fint_register で指定したレジスタを退避、回復せずに割り込み関数で使用できます。このため、割り込み関数の性能は改善しますが、割り込み関数以外で使用可能なレジスタが減るため、プログラム全体の性能が劣化する場合があります。

C ソース

```
volatile int count;

#pragma interrupt int_func(vect=10, fint)
void int_func(void)
{
  count++;
}
```

	-fint_register 指定無し	-fint_register=2
コードサイズ[バイト]	18	14
サイクル数[サイクル]	12	8

4.　コーディングテクニック

本章では、ユーザプログラムのコーディング方法によりコードサイズ・ROM データサイズ・実行速度を改善する方法を説明します。

○…改善する　×…劣化する　△…状況に依存する　—…変化しない

項目	コードサイズ	ROMサイズ	サイクル数	備考
データの構造	○	―	○	
変数と const 型	―	○	―	
局所変数と大域変数	○	○	○	
ビットフィールドの割り付け	○	○	○	
ループ制御変数	×	―	○	
関数のインタフェース	○	―	○	
ループ回数の削減	×	―	○	
テーブルの活用	○	×	○	
分岐	―	―	○	
インライン展開	△	―	○	
switch 文の代わりに if～else 文を使う	△	―	×	
複数箇所にある外部変数アクセスをテンポラリ変数により 1 箇所にまとめる	○	―	○	
条件分岐先の同一文は分岐前に移動する	○	―	○	
複雑な if 文を論理的に等しいものに置き換える	○	―	○	
short, char 型の変数は int 型にする	○	―	○	
switch 文の共通な case の処理をまとめる	○	―	○	
for ループを do while ループに置き換える	○	—	○	
2 のべき乗の除算はシフト演算に置き換える	○	―	○	
2 ビット以上のビットフィールドは char 型に変更する	○	×	○	
定数参照の際には絶対値の小さい値を割り当てる	○	―	―	

4.1　データの構造体化

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。

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、コードサイズや実行速度といったコード効率の改善が期待できます。また、引数として渡す場合もコード効率が改善します。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。

データを構造体化すると、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。

<table>
<tr><th>変更前</th><th>変更後</th></tr>
<tr><td>C ソース

<pre>int a, b, c;

void func(void)
{
  a = 1;
  b = 2;
  c = 3;
}</pre></td><td>C ソース

<pre>struct s
{
  int a;
  int b;
  int c;
} s1;

void func(void)
{
  struct s *p = &s1;
  p->a = 1;
  p->b = 2;
  p->c = 3;
}</pre></td></tr>
<tr><td>アセンブリ展開コード

<pre>_func:
   .STACK_func=4
   MOV.L #_a, R14
   MOV.L #00000001H, [R14]
   MOV.L #_b, R14
   MOV.L #00000002H, [R14]
   MOV.L #_c, R14
   MOV.L #00000003H, [R14]
   RTS</pre></td><td>アセンブリ展開コード

<pre>_func:
   .STACK_func=4
   MOV.L #_s1, R1
   MOV.L #00000001H, [R1]
   MOV.L #00000002H, 04H[R1]
   MOV.L #00000003H, 08H[R1]
   RTS</pre></td></tr>
<tr><td>コードサイズ:28 バイト
サイクル数:9 サイクル</td><td>コードサイズ:16 バイト
サイクル数:7 サイクル</td></tr>
</table>

4.2　変数と const 型

値を変更しない変数は、const 修飾で宣言してください。

大域変数を宣言と同時に初期化するプログラムを書くと、初期値は ROM に、大域変数は RAM にそれぞれ配置されます。大域変数の初期化は、プログラムの開始時に ROM から RAM に初期値を転送することにより実現されています。初期値のある変数を const 修飾しておくと、変数を書き換えるおそれがなくなるため、コンパイラは RAM を確保しません。その結果、RAM を節約することができ、ROM から RAM への転送処理も省略することができます。

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。

変更前	変更後
C ソース `char a[] =` `  {1, 2, 3, 4, 5};`	C ソース `const char a[] =` `  {1, 2, 3, 4, 5};`
ROM サイズ:5 バイト RAM サイズ:5 バイト	ROM サイズ:5 バイト RAM サイズ:0 バイト

4.3　局所変数と大域変数

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が向上します。

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、最適化が掛りにくくなります。

変更前	変更後
C ソース `int tmp;` `void func(int* a, int* b)` `{` `  tmp = *a;` `  *a = *b;` `  *b = tmp;` `}`	C ソース `void func(int* a, int* b)` `{` `  int tmp;` `  tmp = *a;` `  *a = *b;` `  *b = tmp;` `}`
アセンブリ展開コード `_func:` `  .STACK_func=4` `  MOV.L #_tmp, R14` `  MOV.L [R1], [R14]` `  MOV.L [R2], [R1]` `  MOV.L [R14], [R2]` `  RTS`	アセンブリ展開コード `_func:` `  .STACK_func=4` `  MOV.L [R1], R14` `  MOV.L [R2], [R1]` `  MOV.L R14, [R2]` `  RTS`
コードサイズ:13 バイト サイクル数:13 サイクル	コードサイズ:7 バイト サイクル数:8 サイクル

4.4　構造体宣言のメンバオフセット

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るとサイズが改善する場合があります。

変更前	変更後
C ソース struct str { long l1[8]; char c1; }; struct str str1; char x; void func(void) { x = str1.c1; }	C ソース struct str { char c1; long l1[8]; }; struct str str1; char x; void func(void) { x = str1.c1; }
アセンブリ展開コード _func: .STACK_func=4 MOV.L #_x, R14 MOV.L #_str1, R15 MOV.B 20H[R15], [R14] RTS	アセンブリ展開コード _func: .STACK_func=4 MOV.L #_x, R14 MOV.L #_str1, R15 MOV.B [R15], [R14] RTS
コードサイズ:16 バイト サイクル数:8 サイクル	コードサイズ:15 バイト サイクル数:8 サイクル

構造体を定義する際には、境界調整数を意識してメンバの宣言をおこなってください。構造体の境界調整数は構造体内の最も大きな境界調整値に合わせられ、構造体のサイズは境界調整数の倍数となります。その為、構造体の末尾が構造体自身の境界調整数と合わない場合に、次の境界調整を保証するために生成される、未使用領域もサイズに含めてしまいます。

変更前	変更後
C ソース /* 最大メンバが long 型の為、境界調整数は 4 */ struct str { char c1; /* 1byte + 境界調整分 3byte */ long l1; /* 4byte */ char c2; /* 1byte */ char c3; /* 1byte */ char c4; /* 1byte + 境界調整分 1byte */ } str1;	C ソース /* 最大メンバが long 型の為、境界調整数は 4 */ struct str { char c1; /* 1byte */ char c2; /* 1byte */ char c3; /* 1byte */ char c4; /* 1byte */ long l1; /* 4byte */ } str1;
アセンブリ展開コード .SECTION B,DATA,ALIGN=4 _str1: .blkl 3	アセンブリ展開コード .SECTION B,DATA,ALIGN=4 _str1: .blkl 2

4.5　ビットフィールドの割り付け

ビットフィールドで、連続して値を設定されるものは、同じ構造体内に割り付けるようにしてください。

異なる構造体内にあるビットフィールドのメンバを設定するためには、構造体ごとに分けてアクセス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、このアクセスを一度で済ませることができます。

以下は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です。

変更前	変更後
C ソース `struct str` `{` `  int flag1:1;` `} b1, b2, b3;` `void func(void)` `{` `  b1.flag1 = 1;` `  b2.flag1 = 1;` `  b3.flag1 = 1;` `}`	C ソース `struct str` `{` `  int flag1:1;` `  int flag2:1;` `  int flag3:1;` `} a1;` `void func(void)` `{` `  a1.flag1 = 1;` `  a1.flag2 = 1;` `  a1.flag3 = 1;` `}`
アセンブリ展開コード `_func:` `  .STACK_func=4` `  MOV.L #_b1, R14` `  BSET #00H, [R14].B` `  MOV.L #_b2, R14` `  BSET #00H, [R14].B` `  MOV.L #_b3, R14` `  BSET #00H, [R14].B` `  RTS`	アセンブリ展開コード `_func:` `  .STACK_func=4` `  MOV.L #_a1, R14` `  BSET #00H, [R14].B` `  BSET #01H, [R14].B` `  BSET #02H, [R14].B` `  RTS`
コードサイズ:25 バイト サイクル数:13 サイクル	コードサイズ:13 バイト サイクル数:13 サイクル

4.6　ループ制御変数

ループ制御変数は符号付き 4 バイト整数型(signed int/signed long)で宣言すると、ループ内で各種最適化が適用され易くなり、コードサイズの削減、実行速度の向上が期待できます。

<table>
<tr><th>変更前</th><th>変更後</th></tr>
<tr><td>C ソース

<pre>
unsigned long n = 50;
signed short array[100];

void func(void)
{
  signed short i;
  for (i = 0; i < n; i++) {
    array[i+5] = 0;
  }
}
</pre></td><td>C ソース

<pre>
unsigned long n = 50;
signed short array[100];

void func(void)
{
  signed long i;
  for (i = 0; i < n; i++) {
    array[i+5] = 0;
  }
}
</pre></td></tr>
<tr><td>アセンブリ展開コード

<pre>
_func:
    .STACK_func=4
    MOV.L #00000000H, R15
    MOV.L #_n, R14
    MOV.L [R14], R14
    MOV.L #_array, R5
L11: ; bb7
    MOV.W R15, R15
    CMP R14, R15
    BGEU L13
L12: ; bb
    SHLL #01H, R15, R1
    ADD R5, R1
    ADD #01H, R15
    MOV.W #0000H, 0AH[R1]
    BRA L11
L13: ; return
    RTS
</pre></td><td>アセンブリ展開コード

<pre>
_func:
    .STACK_func=4
    MOV.L #00000000H, R5
    MOV.L #_n, R15
    MOV.L [R15], R15
    MOV.L #_array, R14
    ADD #0AH, R14
    ADD #01H, R15
L11: ; bb6
    SUB #01H, R15
    BEQ L13
L12: ; bb
    MOV.W R5, [R14+]
    BRA L11
L13: ; return
    RTS
</pre></td></tr>
<tr><td>コードサイズ:35 バイト
サイクル数:511 サイクル</td><td>コードサイズ:29 バイト
サイクル数:312 サイクル</td></tr>
</table>

4.7　関数のインタフェース

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。

引数がすべてレジスタに割りつくように（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らない場合があります。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。

関数のインタフェース仕様は、コンパイラユーザーズマニュアルに記載されています。

<table>
<tr><th>変更前</th><th>変更後</th></tr>
<tr><td>C ソース

<pre>
void func(char a, char b, char c,
         char d, char e, char f,
         char g, char h) {}

void call_func(void)
{
  func(1,2,3,4,5,6,7,8);
}
</pre></td><td>C ソース

<pre>
struct str
{
  char a;
  char b;
  char c;
  char d;
  char e;
  char f;
  char g;
  char h;
};

void func(struct str* str_arg) {}

void call_func(void)
{
  struct str arg =
    {1,2,3,4,5,6,7,8};
  func(&arg);
}
</pre></td></tr>
<tr><td>アセンブリ展開コード

<pre>
_call_func:
    .STACK_call_func=8
    SUB #04H, R0
    MOV.L #00000004H, R4
    MOV.B #05H, [R0]
    MOV.L #00000003H, R3
    MOV.L #00000002H, R2
    MOV.B #08H, 03H[R0]
    MOV.L #00000001H, R1
    MOV.B #07H, 02H[R0]
    MOV.B #06H, 01H[R0]
    BSR _func
    ADD #04H, R0
    RTS
</pre></td><td>アセンブリ展開コード

<pre>
_call_func:
    .STACK_call_func=12
    SUB #08H, R0
    MOV.L R0, R1
    MOV.L #04030201H, [R0]
    MOV.L #08070605H, 04H[R0]
    BSR _func
    RTSD #08H
</pre></td></tr>
<tr><td>コードサイズ:28 バイト
サイクル数:22 サイクル</td><td>コードサイズ:22 バイト
サイクル数:15 サイクル</td></tr>
</table>

4.8　ループ回数の削減

ループを展開すると、実行速度は大幅に向上します。

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。

<table>
<tr><th>変更前</th><th>変更後</th></tr>
<tr><td>C ソース

<pre>
int a[100];

void func(void)
{
  int i;
  for (i = 0; i < 100; i++) {
    a[i] = 0;
  }
}
</pre></td><td>C ソース

<pre>
int a[100];

void func(void)
{
  int i;
  for (i = 0; i < 100; i += 2) {
    a[i] = 0;
    a[i+1] = 0;
  }
}
</pre></td></tr>
<tr><td>アセンブリ展開コード

<pre>
_func:
    .STACK_func=4
    MOV.L #00000064H, R15
    MOV.L #_a, R14
    MOV.L #00000000H, R5
L11: ; bb
    SUB #01H, R15
    MOV.L R5, [R14+]
    BNE L11
L12: ; return
    RTS
</pre></td><td>アセンブリ展開コード

<pre>
_func:
    .STACK_func=4
    MOV.L #00000032H, R14
    MOV.L #_a, R1
L11: ; bb
    MOV.L #00000000H, [R1]
    MOV.L #00000000H, 04H[R1]
    ADD #08H, R1
    SUB #01H, R14
    BNE L11
L12: ; return
    RTS
</pre></td></tr>
<tr><td>コードサイズ:19 バイト
サイクル数:504 サイクル</td><td>コードサイズ:22 バイト
サイクル数:402 サイクル</td></tr>
</table>

なお、-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。ループ内の処理を全て展開した場合は、ループの判定式も削除されます。例えばループ回数が 8 回の場合は、-loop=8 を指定するとループの判定式も削除されます。

4.9　テーブルの活用

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。

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。

以下の例では、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。

<table>
<tr><th>変更前</th><th>変更後</th></tr>
<tr><td>C ソース

<pre>
char func(int i)
{
  char ch;
  switch (i){
  case 0:
    ch = 'a'; break;
  case 1:
    ch = 'x'; break;
  case 2:
    ch = 'b'; break;
  default:
    ch = 0; break;
  }
  return (ch);
}
</pre></td><td>C ソース

<pre>
const char chbuf[] =
  {'a', 'x', 'b'};

char func(int i)
{
  if ((unsigned int)i < 3) {
    return (chbuf[i]);
  }
  return (0);
}
</pre></td></tr>
<tr><td>アセンブリ展開コード

<pre>
_func:
    .STACK_func=4
    CMP #00H, R1
    BEQ L14
L11: ; entry
    CMP #01H, R1
    BEQ L15
L12: ; entry
    CMP #02H, R1
    BEQ L16
L13: ; switch_clause_bb5
    MOV.L #00000000H, R1
    RTS
L14: ; switch_break_bb
    MOV.L #00000061H, R1
    RTS
L15: ; switch_clause_bb3
    MOV.L #00000078H, R1
    RTS
L16: ; switch_clause_bb4
    MOV.L #00000062H, R1
    RTS
</pre></td><td>アセンブリ展開コード

<pre>
_func:
    .STACK_func=4
    CMP #02H, R1
    BGTU L12
L11: ; if_then_bb
    MOV.L #_chbuf, R14
    MOVU.B [R14,R1], R1
    RTS
L12: ; bb8
    MOV.L #00000000H, R1
    RTS
</pre></td></tr>
<tr><td>コードサイズ：27 バイト
サイクル数:9 サイクル</td><td>コードサイズ：17 バイト
サイクル数:7 サイクル</td></tr>
</table>

4.10 分岐

分岐するケースの位置を変更することで実行速度が向上します。else if 文のように上から順に比較をする場合、場合分けが増えると末端のケースの実行速度は低下します。頻繁に分岐するケースは先頭近くに配置してください。

<table>
<tr><th>変更前</th><th>変更後</th></tr>
<tr><td>C ソース

<pre>
int func(int a)
{
  if (a == 1) {
    a = 2;
  }
  else if (a == 2) {
    a = 4;
  }
  else if (a == 3) {
    a = 8;
  }
  else {
    a = 0;
  }
  return (a);
}
</pre></td><td>C ソース

<pre>
int func(int a)
{
  if (a == 3) {
    a = 8;
  }
  else if (a == 2) {
    a = 4;
  }
  else if (a == 1) {
    a = 2;
  }
  else {
    a = 0;
  }
  return (a);
}
</pre></td></tr>
<tr><td>アセンブリ展開コード

<pre>
_func:
    .STACK_func=4
    CMP #01H, R1
    STZ #02H, R1
    BEQ L12
L11: ; if_else_bb
    CMP #02H, R1
    STZ #04H, R1
L12: ; if_else_bb
    BEQ L14
L13: ; if_else_bb9
    CMP #03H, R1
    SCEQ.L R1
    SHLL #03H, R1
L14: ; if_break_bb17
    RTS
</pre></td><td>アセンブリ展開コード

<pre>
_func:
    .STACK_func=4
    CMP #03H, R1
    STZ #08H, R1
    BEQ L12
L11: ; if_else_bb
    CMP #02H, R1
    STZ #04H, R1
L12: ; if_else_bb
    BEQ L14
L13: ; if_else_bb9
    CMP #01H, R1
    SCEQ.L R1
    SHLL #01H, R1
L14: ; if_break_bb17
    RTS
</pre></td></tr>
<tr><td>サイクル数：12 サイクル
(a=3 の場合)</td><td>サイクル数：9 サイクル
(a=3 の場合)</td></tr>
</table>

4.11 インライン展開

頻繁に呼び出される関数をインライン展開すると実行速度を向上できます。#pragma inline でインライン展開する関数を指定することができます。一方、インライン展開をした場合、一般的にはプログラムサイズが増大する傾向にあります。

インライン展開する関数が他のソースファイルから参照されていない場合、static 関数にしておくと関数本体のコードが削除されコードサイズが小さくなる可能性があります。

変更前	変更後
C ソース `int x[10], y[10];` `static void sub(int* a, int* b,` `  int i)` `{` `  int temp;` `  temp = a[i];` `  a[i] = b[i];` `  b[i] = temp;` `}` `void func(void)` `{` `  int i;` `  for (i = 0; i < 10; i++) {` `  sub(x, y, i);` `  }` `}`	C ソース `int x[10], y[10];` `#pragma inline (sub)` `static void sub(int* a, int* b,` `  int i)` `{` `  int temp;` `  temp = a[i];` `  a[i] = b[i];` `  b[i] = temp;` `}` `void func(void)` `{` `  int i;` `  for (i = 0; i < 10; i++) {` `  sub(x, y, i);` `  }` `}`

アセンブリ展開コード	アセンブリ展開コード
`` __$sub: .STACK__$sub=4 MOV.L [R3,R2], R14 MOV.L [R3,R1], R15 MOV.L R14, [R3,R1] MOV.L R15, [R3,R2] RTS _func: .STACK_func=16 PUSHM R6-R8 MOV.L #_y, R7 MOV.L #_x, R8 MOV.L #00000000H, R6 L12: ; bb MOV.L R8, R1 MOV.L R7, R2 MOV.L R6, R3 BSR __$sub ADD #01H, R6 CMP #0AH, R6 BNE L12 L13: ; return RTSD #0CH, R6-R8 ```	``` _func: .STACK_func=4 MOV.L #0000000AH, R5 MOV.L #_y, R14 MOV.L #_x, R15 L11: ; bb MOV.L [R14], R4 SUB #01H, R5 MOV.L [R15], R3 MOV.L R4, [R15+] MOV.L R3, [R14+] BNE L11 L12: ; return RTS ```
コードサイズ：47 バイト サイクル数：189 サイクル	コードサイズ：29 バイト サイクル数：84 サイクル

4.12 switch 文の代わりに if～else 文を使う

switch 文の分岐コードはコードサイズが増大する要因になるため、if~else 文に置き換えることでコードサイズを小さくできることがあります。また、頻繁に分岐するケースは先頭近くに配置することで実行速度を向上させることができます。(4.10 参照)

変更前	変更後
C ソース `int func(int x)` `{` `  switch(x) {` `  case 0:` `  sub0();` `  break;` `  case 1:` `  sub1();` `  break;` `  case 2:` `  sub2();` `  break;` `  case 3:` `  sub3();` `  break;` `  }` `  return (0);` `}`	C ソース `int func(int x)` `{` `  if (x == 0) {` `  sub0();` `  }` `  else if (x == 1) {` `  sub1();` `  }` `  else if (x == 2) {` `  sub2();` `  }` `  else if (x == 3) {` `  sub3();` `  }` `  return (0);` `}`

アセンブリ展開コード	アセンブリ展開コード
_func: 　.STACK_func=4 　CMP #00H, R1 　BEQ L19 L15:　; entry 　CMP #01H, R1 　BEQ L20 L16:　; entry 　CMP #02H, R1 　BEQ L21 L17:　; entry 　CMP #03H, R1 　BEQ L22 L18:　; switch_break_bb 　MOV.L #00000000H, R1 　RTS L19:　; switch_clause_bb 　BSR _sub0 　BRA L18 L20:　; switch_clause_bb2 　BSR _sub1 　BRA L18 L21:　; switch_clause_bb3 　BSR _sub2 　BRA L18 L22:　; switch_clause_bb4 　BSR _sub3 　BRA L18	_func: 　.STACK_func=4 　CMP #00H, R1 　BNE L16 L15:　; if_then_bb 　BSR _sub0 　BRA L22 L16:　; if_else_bb 　CMP #01H, R1 　BNE L18 L17:　; if_then_bb8 　BSR _sub1 　BRA L22 L18:　; if_else_bb9 　CMP #02H, R1 　BNE L20 L19:　; if_then_bb14 　BSR _sub2 　BRA L22 L20:　; if_else_bb15 　CMP #03H, R1 　BNE L22 L21:　; if_then_bb20 　BSR _sub3 L22:　; if_break_bb23 　MOV.L #00000000H, R1 　RTS
コードサイズ：39 バイト サイクル数：25 サイクル	コードサイズ：32 バイト サイクル数：23 サイクル

4.13 複数箇所にある外部変数アクセスをテンポラリ変数により 1 箇所にまとめる

テンポラリ変数へのアクセスは外部変数へのアクセスと比べてレジスタ転送コードになる傾向が強いため、テンポラリ変数を活用して外部変数へのアクセス箇所を減らすことでコードサイズが小さくなる可能性があります。

変更前

C ソース

```
extern int s;

int func(int x)
{
  switch (x) {
  case 0:
    s = 0;
    break;
  case 1000:
    s = 0x5555;
    break;
  case 2000:
    s = 0xAAAA;
    break;
  case 3000:
    s = 0xFFFF;
  }
  return (0);
}
```

変更後

C ソース

```
extern int s;

int func(int x)
{
  int tmp;
  if (x == 0) {
   tmp = 0;
  }
  else if (x == 1000) {
    tmp = 0x5555;
  }
  else if (x == 2000) {
    tmp = 0xAAAA;
  }
  else if (x == 3000) {
    tmp = 0xFFFF;
  }
  else {
    goto label;
  }
  s = tmp;
label:
  return (0);
}
```

アセンブリ展開コード	アセンブリ展開コード
_func: .STACK_func=4 CMP #00H, R1 MOV.L #_s, R14 BEQ L15 L11: ; entry CMP #03E8H, R1 BEQ L16 L12: ; entry CMP #07D0H, R1 BEQ L17 L13: ; entry CMP #0BB8H, R1 BEQ L18 L14: ; switch_break_bb MOV.L #00000000H, R1 RTS L15: ; switch_clause_bb MOV.L #00000000H, [R14] BRA L14 L16: ; switch_clause_bb2 MOV.L #00005555H, [R14] BRA L14 L17: ; switch_clause_bb3 MOV.L #0000AAAAH, [R14] BRA L14 L18: ; switch_clause_bb4 MOV.L #0000FFFFH, [R14] BRA L14	_func: .STACK_func=4 CMP #00H, R1 MOV.L #00000000H, R14 BEQ L15 L11: ; if_else_bb CMP #03E8H, R1 MOV.L #00005555H, R14 BEQ L15 L12: ; if_else_bb11 CMP #07D0H, R1 MOV.L #0000AAAAH, R14 BEQ L15 L13: ; if_else_bb17 CMP #0BB8H, R1 MOV.L #0000FFFFH, R14 BEQ L15 L14: ; label MOV.L #00000000H, R1 RTS L15: ; if_break_bb26 MOV.L #_s, R15 MOV.L R14, [R15] BRA L14
コードサイズ：56 バイト	コードサイズ：50 バイト

4.14 条件分岐先の同一文は分岐前に移動する

条件分岐の各々の分岐先に同一の文がある場合、条件分岐の前に移動して 1 箇所にまとめてください。

変更前	変更後
C ソース `int s;` `int func(int a, int b, int c)` `{` `  return (a + b + c);` `}` `int call_func(int x)` `{` `  if (x >= 0) {` `  if (x > func(0, 1, 2)) {` `  s++;` `  }` `  }` `  else {` `  if (x < -func(0, 1, 2)) {` `  s--;` `  }` `  }` `  return (0);` `}`	C ソース `int s;` `int func(int a, int b, int c)` `{` `  return (a + b + c);` `}` `int call_func(int x)` `{` `  int tmp = func(0, 1, 2);` `  if (x >= 0) {` `  if (x > tmp) {` `  s++;` `  }` `  }` `  else {` `  if (x < -tmp) {` `  s--;` `  }` `  }` `  return 0;` `}`

アセンブリ展開コード	アセンブリ展開コード
_call_func: .STACK_call_func=12 PUSHM R6-R7 ADD #00H, R1, R6 BN L15 L12: ; if_then_bb MOV.L #00000002H, R3 MOV.L #00000001H, R2 MOV.L #00000000H, R1 BSR _func CMP R6, R1 MOV.L #_s, R7 BGE L16 L13: ; if_then_bb9 MOV.L [R7], R14 ADD #01H, R14 L14: ; if_then_bb9 MOV.L R14, [R7] L15: ; if_break_bb22 MOV.L #00000000H, R1 RTSD #08H, R6-R7 L16: ; if_else_bb MOV.L #00000002H, R3 MOV.L #00000001H, R2 MOV.L #00000000H, R1 BSR _func NEG R1 CMP R1, R6 BGE L15 L17: ; if_then_bb18 MOV.L [R7], R14 SUB #01H, R14 BRA L14	_call_func: .STACK_call_func=8 PUSH.L R6 MOV.L R1, R6 MOV.L #00000002H, R3 MOV.L #00000001H, R2 MOV.L #00000000H, R1 BSR _func CMP #00H, R6 BN L15 L12: ; if_then_bb CMP R6, R1 MOV.L #_s, R14 BGE L16 L13: ; if_then_bb12 MOV.L [R14], R15 ADD #01H, R15 L14: ; if_then_bb12 MOV.L R15, [R14] L15: ; if_break_bb25 MOV.L #00000000H, R1 RTSD #04H, R6-R6 L16: ; if_else_bb NEG R1 CMP R1, R6 BGE L15 L17: ; if_then_bb21 MOV.L [R14], R15 SUB #01H, R15 BRA L14
コードサイズ：58 バイト サイクル数：43 サイクル	コードサイズ：50 バイト サイクル数：31 サイクル

4.15 複雑な if 文を論理的に等しいものに置き換える

if 文の条件式が複雑である場合、等しい意味の簡単な式に置き換えてください。

<table>
<tr><th>変更前</th><th>変更後</th></tr>
<tr><td>C ソース

<pre>
int x;

int func(int s, int t)
{
  s &= 1;
  t &= 1;
  if (!s) {
    if (t) {
      x = 1;
    }
  }
  else {
    if (!t) {
      x = 1;
    }
  }
  return (0);
}
</pre></td><td>C ソース

<pre>
int x;

int func(int s, int t)
{
  s &= 1;
  t &= 1;
  if (! (s ^ t)) {
    x = 1;
  }
  return (0);
}
</pre></td></tr>
<tr><td>アセンブリ展開コード

<pre>
_func:
   .STACK_func=4
   AND #01H, R2
   BTST #00H, R1
   MOV.L #_x, R14
   BNE L12
L11: ; if_then_bb
   CMP #00H, R2
   BNE L13
   BRA L14
L12: ; if_else_bb
   CMP #00H, R2
   BNE L14
L13: ; if_then_bb28
   MOV.L #00000001H, [R14]
L14: ; if_break_bb30
   MOV.L #00000000H, R1
   RTS
</pre></td><td>アセンブリ展開コード

<pre>
_func:
   .STACK_func=4
   XOR R1, R2
   BTST #00H, R2
   BNE L12
L11: ; if_then_bb
   MOV.L #_x, R14
   MOV.L #00000001H, [R14]
L12: ; if_break_bb
   MOV.L #00000000H, R1
   RTS
</pre></td></tr>
<tr><td>コードサイズ：24 バイト
サイクル数：12 サイクル</td><td>コードサイズ：18 バイト
サイクル数：8 サイクル</td></tr>
</table>

4.16 short, char 型の変数は int 型にする

CC-RX は ANSI-C の仕様により、short、char 型の演算を演算前に int 型に型変換してから演算するコードを生成します。また int 型の値を short、char 型の変数に代入するときにも型変換が発生します。あらかじめ int 型で変数定義しておくことで余分な型変換処理を削減できます。

注意：int 型に変更することで、変数や演算結果がとり得る値の範囲が変わります。型を変更する場合はプログラムの動作に影響が無いように注意してください。

変更前	変更後
C ソース `char func(char a, char b, char c)` `{` `  char t = a + b;` `  return (t >> c);` `}`	C ソース `int func(int a, int b, int c)` `{` `  int t = a + b;` `  return (t >> c);` `}`
アセンブリ展開コード `_func:` `  .STACK_func=4` `  ADD R1, R2` `  MOVU.B R2, R14` `  SHLR R3, R14` `  MOVU.B R14, R1` `  RTS`	アセンブリ展開コード `_func:` `  .STACK_func=4` `  ADD R1, R2` `  MOV.L R2, R1` `  SHAR R3, R1` `  RTS`
コードサイズ：10 バイト サイクル数：7 サイクル	コードサイズ：8 バイト サイクル数：6 サイクル

4.17 switch 文の共通な case の処理をまとめる

複数の case ラベルの分岐先の処理が同一である場合、case ラベルを移動して処理を 1 箇所にまとめてください。

変更前	変更後
C ソース `int x;` `void func(void)` `{` `  switch(x) {` `  case 0:` `  dummy1();` `  break;` `  case 1:` `  dummy1();` `  break;` `  case 2:` `  dummy1();` `  break;` `  case 3:` `  dummy2();` `  break;` `  case 4:` `  dummy2();` `  break;` `  default:` `  break;` `  }` `}`	C ソース `int x;` `void func(void)` `{` `  switch(x) {` `  case 0:` `  case 1:` `  case 2:` `  dummy1();` `  break;` `  case 3:` `  case 4:` `  dummy2();` `  break;` `  default:` `  break;` `  }` `}`

アセンブリ展開コード	アセンブリ展開コード
_func: 　.STACK_func=4 　MOV.L #_x, R14 　MOV.L [R14], R14 　CMP #00H, R14 　BEQ L15 L13:　; entry 　CMP #01H, R14 　BEQ L15 L14:　; entry 　CMP #02H, R14 L15:　; entry 　BEQ L20 L16:　; entry 　CMP #03H, R14 　BEQ L18 L17:　; entry 　CMP #04H, R14 L18:　; entry 　BEQ L21 L19:　; return 　RTS L20:　; switch_clause_bb2 　BRA _dummy1 L21:　; switch_clause_bb4 　BRA _dummy2	_func: 　.STACK_func=4 　MOV.L #_x, R14 　MOV.L [R14], R14 　CMP #03H, R14 　BLTU L15 L13:　; entry 　SUB #03H, R14 　CMP #02H, R14 　BLTU L16 L14:　; return 　RTS L15:　; switch_clause_bb 　BRA _dummy1 L16:　; switch_clause_bb1 　BRA _dummy2
コードサイズ：28 バイト サイクル数：20 サイクル	コードサイズ：23 バイト サイクル数：15 サイクル

4.18 for ループを do while ループに置き換える

1 回以上ループを実行することが分かっている for 文の場合、do while 文に置き換えることでコードサイズが小さくなる可能性があります。また、条件式を等価比較に置き換えることでもコードサイズが小さくなる可能性があります。

<table>
<tr><th>変更前</th><th>変更後</th></tr>
<tr><td>C ソース

<pre>
int array[10][10];

void func(int nsize, int msize)
{
  int i;
  int *p;
  int s;
  p = &array[0][0];
  s = nsize * msize;
  for (i = 0; i < s; i++) {
    *p++ = 0;
  }
}
</pre></td><td>C ソース

<pre>
int array[10][10];

void func(int nsize, int msize)
{
  int i;
  int *p;
  int s;
  p = &array[0][0];
  s = nsize * msize;
  i = 0;
  do {
    *p++ = 0;
    i++;
  } while (i != s);
}
</pre></td></tr>
<tr><td>アセンブリ展開コード

<pre>
_func:
   .STACK_func=4
   MUL R1, R2
   MOV.L #_array, R15
   MOV.L #00000000H, R14
   MOV.L #00000000H, R5
L11: ; bb13
   CMP R2, R14
   BGE L13
L12: ; bb
   MOV.L R5, [R15+]
   ADD #01H, R14
   BRA L11
L13: ; return
   RTS
</pre></td><td>アセンブリ展開コード

<pre>
_func:
   .STACK_func=4
   MUL R1, R2
   MOV.L #_array, R15
   MOV.L #00000000H, R14
   MOV.L #00000000H, R5
L11: ; bb
   ADD #01H, R14
   CMP R2, R14
   MOV.L R5, [R15+]
   BLT L11
L12: ; return
   RTS
</pre></td></tr>
<tr><td>コードサイズ：24 バイト
サイクル数：458 サイクル</td><td>コードサイズ：22 バイト
サイクル数：389 サイクル</td></tr>
</table>

4.19 2 のべき乗の除算はシフト演算に置き換える

除算の除数が 2 のべき乗であり、被除数が正の値と分かっている場合は、除算をシフト演算に置き換えてください。

変更前	変更後
C ソース `int s;` `void func(void)` `{` `  s = s / 2;` `}`	C ソース `int s;` `void func(void)` `{` `  s = s >> 1;` `}`
アセンブリ展開コード `_func:` `  .STACK_func=4` `  MOV.L #_s, R14` `  MOV.L [R14], R15` `  DIV #02H, R15` `  MOV.L R15, [R14]` `  RTS`	アセンブリ展開コード `_func:` `  .STACK_func=4` `  MOV.L #_s, R14` `  MOV.L [R14], R15` `  SHAR #01H, R15` `  MOV.L R15, [R14]` `  RTS`
コードサイズ：15 バイト サイクル数：10 サイクル	コードサイズ：13 バイト サイクル数：8 サイクル

4.20 2 ビット以上のビットフィールドは char 型に変更する

ビットフィールドのメンバのサイズが 2 ビット以上の場合は、ビットフィールドは使用せずに char 型に変更してください。ただし、ROM データサイズは増加します。

<table>
<tr><th>変更前</th><th>変更後</th></tr>
<tr><td>C ソース

<pre>
struct {
  unsigned char b0:1;
  unsigned char b1:2;
} dw;

unsigned char dummy;

int func(void)
{
  if (dw.b1) {
    dummy++;
  }
  return (0);
}
</pre></td><td>C ソース

<pre>
struct {
  unsigned char b0:1;
  unsigned char b1;
} db;

unsigned char dummy;

int func(void)
{
  if (db.b1) {
    dummy++;
  }
  return (0);
}
</pre></td></tr>
<tr><td>アセンブリ展開コード

<pre>
_func:
    .STACK_func=4
    MOV.L #00000006H, R15
    MOV.L #_dw, R14
    TST [R14].UB, R15
    MOV.L #00000000H, R1
    BNE L12
L11: ; if_break_bb
    RTS
L12: ; if_then_bb
    MOV.L #_dummy, R14
    MOVU.B [R14], R15
    ADD #01H, R15
    MOV.B R15, [R14]
    RTS
</pre></td><td>アセンブリ展開コード

<pre>
_func:
    .STACK_func=4
    MOV.L #_db, R1
    MOVU.B 01H[R1], R1
    CMP #00H, R1
    MOV.L #00000000H, R1
    BNE L12
L11: ; if_break_bb
    RTS
L12: ; if_then_bb
    MOV.L #_dummy, R14
    MOVU.B [R14], R15
    ADD #01H, R15
    MOV.B R15, [R14]
    RTS
</pre></td></tr>
<tr><td>コードサイズ：29 バイト
ROM サイズ：1 バイト
サイクル数：10 サイクル</td><td>コードサイズ：28 バイト
ROM サイズ：2 バイト
サイクル数：9 サイクル</td></tr>
</table>

4.21 定数参照の際には絶対値の小さい値を割り当てる

定数参照は絶対値が小さい値の方がコードサイズは小さくなる可能性があります。ID の割り当てなどで定数値を使用する際には、小さい値を割り当てるようにしてください。

変更前	変更後
C ソース `#define ID_1  (1000)` `int id;` `void func(void)` `{` `  id = ID_1;` `}`	C ソース `#define ID_1  (1)` `int id;` `void func(void)` `{` `  id = ID_1;` `}`
アセンブリ展開コード `_func:` `  .STACK_func=4` `  MOV.L #_id, R14` `  MOV.L #000003E8H, [R14]` `  RTS`	アセンブリ展開コード `_func:` `  .STACK_func=4` `  MOV.L #_id, R14` `  MOV.L #00000001H, [R14]` `  RTS`
コードサイズ：11 バイト	コードサイズ：10 バイト

改訂記録<revision history>

Rev.	発行日	改訂内容	
		ページ	ポイント
1.00	2021.8.31	4	新規発行
1.01	2024.1.16	7	具体的な対象デバイスを追記

製品ご使用上の注意事項

ここでは、マイコン製品全体に適用する「使用上の注意事項」について説明します。個別の使用上の注意事項については、本ドキュメントおよびテクニカルアップデートを参照してください。

1. 静電気対策
   CMOS 製品の取り扱いの際は静電気防止を心がけてください。CMOS 製品は強い静電気によってゲート絶縁破壊を生じることがあります。運搬や保存の際には、当社が出荷梱包に使用している導電性のトレーやマガジンケース、導電性の緩衝材、金属ケースなどを利用し、組み立て工程にはアースを施してください。プラスチック板上に放置したり、端子を触ったりしないでください。また、CMOS 製品を実装したボードについても同様の扱いをしてください。
2. 電源投入時の処置
   電源投入時は、製品の状態は不定です。電源投入時には、LSI の内部回路の状態は不確定であり、レジスタの設定や各端子の状態は不定です。外部リセット端子でリセットする製品の場合、電源投入からリセットが有効になるまでの期間、端子の状態は保証できません。同様に、内蔵パワーオンリセット機能を使用してリセットする製品の場合、電源投入からリセットのかかる一定電圧に達するまでの期間、端子の状態は保証できません。
3. 電源オフ時における入力信号
   当該製品の電源がオフ状態のときに、入力信号や入出力プルアップ電源を入れないでください。入力信号や入出力プルアップ電源からの電流注入により、誤動作を引き起こしたり、異常電流が流れ内部素子を劣化させたりする場合があります。資料中に「電源オフ時における入力信号」についての記載のある製品は、その内容を守ってください。
4. 未使用端子の処理
   未使用端子は、「未使用端子の処理」に従って処理してください。CMOS 製品の入力端子のインピーダンスは、一般に、ハイインピーダンスとなっています。未使用端子を開放状態で動作させると、誘導現象により、LSI 周辺のノイズが印加され、LSI 内部で貫通電流が流れたり、入力信号と認識されて誤動作を起こす恐れがあります。
5. クロックについて
   リセット時は、クロックが安定した後、リセットを解除してください。プログラム実行中のクロック切り替え時は、切り替え先クロックが安定した後に切り替えてください。リセット時、外部発振子（または外部発振回路）を用いたクロックで動作を開始するシステムでは、クロックが十分安定した後、リセットを解除してください。また、プログラムの途中で外部発振子（または外部発振回路）を用いたクロックに切り替える場合は、切り替え先のクロックが十分安定してから切り替えてください。
6. 入力端子の印加波形
   入力ノイズや反射波による波形歪みは誤動作の原因になりますので注意してください。CMOS 製品の入力がノイズなどに起因して、V_{IL}（Max.）から V_{IH}（Min.）までの領域にとどまるような場合は、誤動作を引き起こす恐れがあります。入力レベルが固定の場合はもちろん、V_{IL}（Max.）から V_{IH}（Min.）までの領域を通過する遷移期間中にチャタリングノイズなどが入らないように使用してください。
7. リザーブアドレス（予約領域）のアクセス禁止
   リザーブアドレス（予約領域）のアクセスを禁止します。アドレス領域には、将来の拡張機能用に割り付けられている リザーブアドレス（予約領域）があります。これらのアドレスをアクセスしたときの動作については、保証できませんので、アクセスしないようにしてください。
8. 製品間の相違について
   型名の異なる製品に変更する場合は、製品型名ごとにシステム評価試験を実施してください。同じグループのマイコンでも型名が違うと、フラッシュメモリ、レイアウトパターンの相違などにより、電気的特性の範囲で、特性値、動作マージン、ノイズ耐量、ノイズ幅射量などが異なる場合があります。型名が違う製品に変更する場合は、個々の製品ごとにシステム評価試験を実施してください。

ご注意書き

1. 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。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、お客様の責任において、お客様の機器・システムを設計ください。これらの使用に起因して生じた損害（お客様または第三者いずれに生じた損害も含みます。以下同じです。）に関し、当社は、一切その責任を負いません。
2. 当社製品または本資料に記載された製品データ、図、表、プログラム、アルゴリズム、応用回路例等の情報の使用に起因して発生した第三者の特許権、著作権その他の知的財産権に対する侵害またはこれらに関する紛争について、当社は、何らの保証を行うものではなく、また責任を負うものではありません。
3. 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。
4. 当社製品を組み込んだ製品の輸出入、製造、販売、利用、配布その他の行為を行うにあたり、第三者保有の技術の利用に関するライセンスが必要となる場合、当該ライセンス取得の判断および取得はお客様の責任において行ってください。
5. 当社製品を、全部または一部を問わず、改造、改変、複製、リバースエンジニアリング、その他、不適切に使用しないでください。かかる改造、改変、複製、リバースエンジニアリング等により生じた損害に関し、当社は、一切その責任を負いません。
6. 当社は、当社製品の品質水準を「標準水準」および「高品質水準」に分類しており、各品質水準は、以下に示す用途に製品が使用されることを意図しております。
   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、産業用ロボット等
   高品質水準：輸送機器（自動車、電車、船舶等）、交通制御（信号）、大規模通信機器、金融端末基幹システム、各種安全制御装置等
   当社製品は、データシート等により高信頼性、Harsh environment 向け製品と定義しているものを除き、直接生命・身体に危害を及ぼす可能性のある機器・システム（生命維持装置、人体に埋め込み使用するもの等）、もしくは多大な物的損害を発生させるおそれのある機器・システム（宇宙機器と、海底中継器、原子力制御システム、航空機制御システム、プラント基幹システム、軍事機器等）に使用されることを意図しておらず、これらの用途に使用することは想定していません。たとえ、当社が想定していない用途に当社製品を使用したことにより損害が生じても、当社は一切その責任を負いません。
7. あらゆる半導体製品は、外部攻撃からの安全性を 100%保証されているわけではありません。当社ハードウェア／ソフトウェア製品にはセキュリティ対策が組み込まれているものもありますが、これによって、当社は、セキュリティ脆弱性または侵害（当社製品または当社製品が使用されているシステムに対する不正アクセス・不正使用を含みますが、これに限りません。）から生じる責任を負うものではありません。当社は、当社製品または当社製品が使用されたあらゆるシステムが、不正な改変、攻撃、ウイルス、干渉、ハッキング、データの破壊または窃盗その他の不正な侵入行為（「脆弱性問題」といいます。）によって影響を受けないことを保証しません。当社は、脆弱性問題に起因しまたはこれに関連して生じた損害について、一切責任を負いません。また、法令において認められる限りにおいて、本資料および当社ハードウェア／ソフトウェア製品について、商品性および特定目的との合致に関する保証ならびに第三者の権利を侵害しないことの保証を含め、明示または黙示のいかなる保証も行いません。
8. 当社製品をご使用の際は、最新の製品情報（データシート、ユーザーズマニュアル、アプリケーションノート、信頼性ハンドブックに記載の「半導体デバイスの使用上の一般的な注意事項」等）をご確認の上、当社が指定する最大定格、動作電源電圧範囲、放熱特性、実装条件その他指定条件の範囲内でご使用ください。指定条件の範囲を超えて当社製品をご使用された場合の故障、誤動作の不具合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いますが、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、データシート等において高信頼性、Harsh environment 向け製品と定義しているものを除き、耐放射線設計を行っておりません。仮に当社製品の故障または誤動作が生じた場合であっても、人身事故、火災事故その他社会的損害等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、お客様の機器・システムとしての出荷保証を行ってください。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、お客様の機器・システムとしての安全検証をお客様の責任で行ってください。
10. 当社製品の環境適合性等の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。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. 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・システムに使用することはできません。当社製品および技術を輸出、販売または移転等する場合は、「外国為替及び外国貿易法」その他日本国および適用される外国の輸出管理関連法規を遵守し、それらの定めるところに従い必要な手続きを行ってください。
12. お客様が当社製品を第三者に転売等される場合には、事前に当該第三者に対して、本ご注意書き記載の諸条件を通知する責任を負うものといたします。
13. 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禁じます。
14. 本資料に記載されている内容または当社製品についてご不明な点がございましたら、当社の営業担当者までお問合せください。

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直接的、間接的に支配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。

(Rev.5.0-1　2020.10)

本社所在地

〒135-0061 東京都江東区豊洲 3-2-24（豊洲フォレシア）

www.renesas.com

お問合せ窓口

弊社の製品や技術、ドキュメントの最新情報、最寄の営業お問合せ窓口に関する情報などは、弊社ウェブサイトをご覧ください。

www.renesas.com/contact/

商標について

ルネサスおよびルネサスロゴはルネサス エレクトロニクス株式会社の商標です。すべての商標および登録商標は、それぞれの所有者に帰属します。