EPSON
EXCEED YOUR VISION

S1C17 シリーズ
省メモリプログラミング

セイコーエプソン株式会社

Rev.1.0

評価ボード・キット、開発ツールご使用上の注意事項

1. 本評価ボード・キット、開発ツールは、お客様での技術的評価、動作の確認および開発のみに用いられることを想定し設計されています。それらの技術評価・開発等の目的以外には使用しないで下さい。本品は、完成品に対する設計品質に適合していません。

2. 本評価ボード・キット、開発ツールは、電子エンジニア向けであり、消費者向け製品ではありません。お客様において、適切な使用と安全に配慮願います。弊社は、本品を用いることで発生する損害や火災に対し、いかなる責も負いかねます。通常の使用においても、異常がある場合は使用を中止して下さい。

3. 本評価ボード・キット、開発ツールに用いられる部品は、予告無く変更されることがあります。

本資料のご使用につきましては、次の点にご留意願います。

本資料の内容については、予告無く変更することがあります。

1. 本資料の一部、または全部を弊社に無断で転載、または、複製など他の目的に使用することは堅くお断りいたします。

2. 本資料に掲載される応用回路、プログラム、使用方法等はあくまでも参考情報であり、これらに起因する第三者の知的財産権およびその他の権利侵害あるいは損害の発生に対し、弊社はいかなる保証を行うものではありません。また、本資料によって第三者または弊社の知的財産権およびその他の権利の実施権の許諾を行うものではありません。

3. 特性値の数値の大小は、数直線上の大小関係で表しています。

4. 製品および弊社が提供する技術を輸出等するにあたっては「外国為替および外国貿易法」を遵守し、当該法令の定める手続きが必要です。大量破壊兵器の開発等およびその他の軍事用途に使用する目的をもって製品および弊社が提供する技術を費消、再販売または輸出等しないでください。

5. 本資料に掲載されている製品は、生命維持装置その他、きわめて高い信頼性が要求される用途を前提としていません。よって、弊社は本（当該）製品をこれらの用途に用いた場合のいかなる責任についても負いかねます。

6. 本資料に掲載されている会社名、商品名は、各社の商標または登録商標です。

目　次

1. 概要

本書では、GNU17 により生成されるプログラムがメモリ（ROM および RAM）をどのように使用するか、およびプログラムで使用されるメモリ容量を減らす方法について説明します。

1.1 環境

本書では、プログラムのソースコードが C 言語で作成され、GNU17 によりコンパイルおよびリンクされていることを前提としています。
プログラムに対応する S1C17 命令の例は、GNU17 バージョン 2.2.0 により生成されています。ほとんどの場合、GNU17 バージョン 2.x.x は同じ S1C17 命令を生成しますが、他のバージョンでは異なる命令が生成されることがあります。
使用している GNU17 が実際のプログラムに対して生成する命令は、その都度に確認してください。

本書のプログラム例は、次のメモリマップで実行するように生成されています。

アドレス		領域
0xFF	FFFF	予約
0x03	8000	
0x03	7FFF	ROM （192K バイト）
0x00	8000	
0x00	7FFF	周辺機器の I/O レジスタ （16K バイト）
0x00	4000	
0x00	3FFF	RAM （16K バイト）
0x00	0000	

1.2 GNU17 Cコンパイラにより使用されるメモリ

この項では、C 言語の関数、変数および定数により使用される S1C17 のメモリについて説明します。
GNU17 C コンパイラおよびリンカの詳細については、GNU17 マニュアルの”6.4 コンパイラの出力”を参照してください。

1.2.1 関数

GNU17 C コンパイラは、関数を’.text’セクションに配置します。GNU17 リンカおよびその標準リンカスクリプトは、’.text’セクションを ROM に再配置します。そのため、GNU17 は、通常、C 言語の関数を ROM に配置します。

C ソースコード	S1C17 命令
`extern void external_near_function(void);` `extern void external_far_function(void);`	

`void internal_function(void) {`
`  return;`
`}`
	`0x82ee:ret`

`static void static_function(void) {`
`  return;`
`}`
	`0x82f0:ret`

`void caller_function(void) {`
`  static_function();`
`  internal_function();`
`  external_near_function();`
`  external_far_function();`
`}`
	`0x82f2:call  0x3fe`
	`0x82f4:call  0x3fc`
	`0x82f6:call  0x3`
	`0x82f8:ext  0xf`
	`0x82fa:call  0x282`
	`0x82fc:ret`

S1C17 の 1 命令のサイズは 16 ビットであるため、関数の開始アドレスは 16 ビット境界に割り当てられます。

各関数には、'ret'命令が必要です。つまり、関数のサイズは少なくとも 2 バイトになります。
関数の呼び出し側は、少なくとも'call'命令のために 2 バイトが必要です。プログラムは、'call'命令のオペランドで指定される PC + 2 + 10 ビット符号付き値として計算されるアドレスに分岐します。呼び出される側の関数が 10 ビット領域内にない場合、呼び出し側の関数は、続く'call'命令のオペランドを拡張する'ext'命令のためにさらに 2 バイトを必要とします。
呼び出し側のアドレスと呼び出される側のアドレスを近くに配置することで、呼び出す命令の数を減らすことができます。命令の数を簡単に減らすには、呼び出し側の関数と呼び出される側の関数を C ソースファイル上の近い行に定義します。

1.2.2 グローバル変数

GNU17 C コンパイラは、初期値のないグローバル変数を'.bss'セクションに置きます。
初期値のあるグローバル変数は'.data'セクションに置かれ、それらの初期値は'data_lma'領域に保持されます。
GNU17 リンカは'.bss'および'.data'セクションを RAM に再配置し、'data_lma'領域を ROM に再配置します。そのため、GNU17 は、通常、C 言語のグローバル変数を RAM に配置し、それらの初期値を ROM に保持します。

C ソースコード

```
int global_variable_without_value;
int global_variable_with_value_one = 1;
int global_variable_with_value_zero = 0;
```

GNU17 リンカによりマッピングされるアドレス

```
.bss     0x00000000  0x2
         0x00000000      __START_bss=.
 .bss    0x00000000  0x2 variables.o
         0x00000000      global_variable_without_value
         0x00000002      __END_bss=.

.data    0x00000002  0x4 load address 0x000094fe
         0x00000002      __START_data=.
 .data   0x00000002  0x4 variables.o
         0x00000002      global_variable_with_value_one
         0x00000004      global_variable_with_value_zero
         0x00000006      __END_data=.

         0x000094fe      __START_data_lma=__END_rodata
         0x00009502      __END_data_lma=(__END_rodata+(__END_data-__START_data))
```

GDB コンソール

```
(gdb) p &global_variable_without_value
$1 = (int *) 0x0
(gdb) p &global_variable_with_value_one
$2 = (int *) 0x2
(gdb) p &global_variable_with_value_zero
$3 = (int *) 0x4
(gdb) p (void *)&__START_data_lma
$4 = (void *) 0x94fe
(gdb) p (void *)&__END_data_lma
$5 = (void *) 0x9502
(gdb) x/2h &__START_data_lma
0x94fe:  0x0001   0x0000
```

GNU17 デバッガ（GDB）によりプログラムがターゲットデバイスにロードされると、GDB からデバッグシンボルを使用して、'data_lma'領域の内容を確認できるようになります。

C 言語は明示的に初期化されないグローバル変数の値を 0 としているため、'.bss'セクションは 0 で初期化されます。
0 による初期化を省略することで、'data_lma'領域のサイズを縮小できます。

C ソースコード

```
int global_variable_without_value;
int global_variable_with_value_one = 1;
int global_variable_with_value_zero;
```

GDB コンソール

```
(gdb) p &global_variable_without_value
$1 = (int *) 0x0
(gdb) p &global_variable_with_value_one
$2 = (int *) 0x4
(gdb) p &global_variable_with_value_zero
$3 = (int *) 0x2
(gdb) p (void *)&__START_data_lma
```

```
$4 = (void *) 0x94fe
(gdb) p (void *)&__END_data_lma
$5 = (void *) 0x9500
(gdb) x/1h &__START_data_lma
0x94fe:  0x0001
```

C ソースファイルで同じセクションに属するグローバル変数は、定義された順に従い S1C17 アドレス空間に配置されます。
次の変数は、'.bss'セクションに属し、定義された順に従い配置されます。

C ソースコード

```
char global_a_8bit;
long global_b_32bit;
char global_c_8bit;
short global_d_16bit;
short global_e_16bit;
long global_f_32bit;
```

GDB コンソール

```
(gdb) p &global_a_8bit
$1 = 0x0
(gdb) p &global_b_32bit
$2 = (long int *) 0x4
(gdb) p &global_c_8bit
$3 = 0x8 '¥252' <repeats 200 times>...
(gdb) p &global_d_16bit
$4 = (short int *) 0xa
(gdb) p &global_e_16bit
$5 = (short int *) 0xc
(gdb) p &global_f_32bit
$6 = (long int *) 0x10
```

各変数は、データ型に基づいて独自の境界に割り当てられるため、一部のメモリが未使用になります。

アドレス	アドレス＋0	アドレス＋1	アドレス＋2	アドレス＋3
0x000000	global_a_8bit	未使用	未使用	未使用
0x000004	global_b_32bit			
0x000008	global_c_8bit	未使用	global_d_16bit	
0x00000c	global_e_16bit		未使用	未使用
0x000010	global_f_32bit			

同じデータ型のグローバル変数をまとめて定義することで、未使用のメモリ容量を減らすことができます。

C ソースコード

```
char global_a_8bit;
char global_c_8bit;
short global_d_16bit;
short global_e_16bit;
long global_b_32bit;
long global_f_32bit;
```

GDB コンソール

```
(gdb) p &global_a_8bit
$1 = 0x0
(gdb) p &global_c_8bit
$2 = 0x1 '¥252' <repeats 200 times>...
(gdb) p &global_d_16bit
$3 = (short int *) 0x2
(gdb) p &global_e_16bit
$4 = (short int *) 0x4
(gdb) p &global_b_32bit
$5 = (long int *) 0x8
(gdb) p &global_f_32bit
$6 = (long int *) 0xc
```

アドレス	アドレス＋0	アドレス＋1	アドレス＋2	アドレス＋3
0x000000	global_a_8bit	global_c_8bit	global_d_16bit	
0x000004	global_e_16bit		未使用	未使用
0x000008	global_b_32bit			
0x00000c	global_f_32bit			

変数を大きなデータ型から小さいデータ型の順に定義すると、未使用のメモリ容量をさらに減らすことができます。これは、リンカがその領域に他のオブジェクトを再配置できることがあるためです。

1.2.3 ローカル変数

ほとんどのローカル変数は、自動記憶期間をもつ変数です。GNU17 C コンパイラは、これらの変数を関数のスタックフレームまたは S1C17 レジスタに配置します。
'static'指定子が付くローカル変数は、静的記憶期間をもつ変数です。GNU17 C コンパイラは、これらの初期値のない変数を'.bss'セクションに配置します。

C ソースコード	S1C17 命令

static short static_local_values(**short** a) {
　static short static_local_without_value;
 	
　static_local_without_value = a;
 	
　return static_local_without_value;
}
	0x929a:ld　　[0x2],%r0
	0x929c:ret

GNU17 C コンパイラは、初期値のある'static'ローカル変数を'.data'セクションに配置し、それらの初期値を'data_lma'領域に保持します。

C ソースコード	S1C17 命令

static short static_local_values(**short** a) {
　static short static_local_with_value = 10;
 	
　static_local_with_value += a;
 	
　return static_local_with_value;
}
	0x929a:ld　　%r2,[0x14]
	0x929c:add　　%r0,%r2
	0x929e:ld　　[0x14],%r0
	0x92a0:ret

静的記憶期間をもつローカル変数は'main'関数の前に初期化されるため、この関数で再初期化されることはありません。
0 による初期化を省略することで、'data_lma'領域のサイズを縮小できます。

スタックフレームのローカル変数は、グローバル変数と同様に、定義された順に従い配置されます。つまり、同じデータ型のローカル変数をまとめて定義することで、(RAM から割り当てられる) スタックフレームのサイズを縮小できます。ただし、ローカル変数によっては、GNU17 C コンパイラにより最適化され、スタックフレームに保存されないため、スタックフレームのサイズを把握するには S1C17 命令を確認する必要があります。

C ソースコード	S1C17 命令

short automatic_local_values(**short** a) {
　char local_a_8bit;
　long local_b_32bit;
　char local_c_8bit;
　short local_d_16bit;
　short local_e_16bit;
　long local_f_32bit;
 	
　local_a_8bit = sub_function (a);
　local_b_32bit = sub_function (local_a_8bit);
　local_c_8bit = sub_function(local_b_32bit);
　local_d_16bit = sub_function(local_c_8bit);
　local_e_16bit = sub_function(local_d_16bit);
　local_f_32bit = sub_function(local_e_16bit);
 	
　return local_f_32bit;
}
	0x927a:call　　0x3fe
	0x927c:call.d　0x3fd
	0x927e:ld.b　　%r0,%r0
	0x9280:ld　　%r2,%r0
	0x9282:cv.ls　　%r3,%r2
	0x9284:call.d　0x3f9
	0x9286:ld　　%r0,%r2
	0x9288:call.d　0x3f7
	0x928a:ld.b　　%r0,%r0
	0x928c:call　　0x3f5
	0x928e:call　　0x3f4
	0x9290:ld　　%r2,%r0
	0x9292:cv.ls　　%r3,%r2
	0x9294:ld　　%r0,%r2
	0x9296:ret

1.2.4 定数

グローバル定数および'static'が指定された定数は、静的記憶期間をもちます。
GNU17 C コンパイラは、静的記憶期間をもつ定数を'.rodata'セクションに配置します。GNU17 リンカ

は、’.rodata’セクションを ROM に再配置します。
そのため、GNU17 は、通常、C 言語の定数を ROM に配置します。

C ソースコード

```
const char global_constant_a_8bit = 0x01;
const short global_constant_b_16bit = 0x2345;

short constant_values(void) {
    static const short local_constant_c_16bit = 0x6789;

    return local_constant_c_16bit;
}

const long global_constant_d_32bit = 0xabcdef01;
const short global_constant_e_16bit = 0x2345;
```

GDB コンソール

```
(gdb) p &global_constant_a_8bit
$1 = 0x9568 "."
(gdb) p &global_constant_b_16bit
$2 = (short int *) 0x956a
(gdb) p &global_constant_d_32bit
$3 = (long int *) 0x9570
(gdb) p &global_constant_e_16bit
$4 = (long int *) 0x9574
(gdb) p &__START_rodata
$5 = (<data variable, no debug info> *) 0x9568
(gdb) p &__END_rodata
$6 = (<variable (not text or data), no debug info> *) 0x9576 "."
(gdb) x/14b &__START_rodata
0x9568:  0x01    0x00    0x45    0x23    0x89    0x67    0x00    0x00
0x9570:  0x01    0xef    0xcd    0xab    0x45    0x23
```

静的記憶期間をもつ定数は、変数と同様に、定義された順に従い配置されます。つまり、同じデータ型の定数をまとめて定義することで、’.rodata’セクションのサイズを縮小できます。

C ソースコード

```
const long global_constant_d_32bit = 0xabcdef01;
const short global_constant_b_16bit = 0x2345;
const short global_constant_e_16bit = 0x2345;

short constant_values(void) {
    static const short local_constant_c_16bit = 0x6789;

    return local_constant_c_16bit;
}

const char global_constant_a_8bit = 0x01;
```

GDB コンソール

```
(gdb) p &global_constant_d_32bit
$1 = (long int *) 0x9568
(gdb) p &global_constant_b_16bit
$2 = (short int *) 0x956c
(gdb) p &global_constant_e_16bit
$3 = (short int *) 0x956e
(gdb) p &global_constant_a_8bit
$4 = 0x9572 "."
(gdb) p &__START_rodata
$5 = (<data variable, no debug info> *) 0x9568
(gdb) p &__END_rodata
$6 = (<variable (not text or data), no debug info> *) 0x9574 "."
(gdb) x/12b &__START_rodata
0x9568:  0x01    0xef    0xcd    0xab    0x45    0x23    0x45    0x23
0x9570:  0x89    0x67    0x01    0x00
```

2 つの定数の値が同じであっても、GNU17 はそれぞれに対応したオブジェクトを生成します。そのため、同じ値の定数を 1 つの定数にまとめることで、’.rodata’セクションのサイズを縮小できます。

C ソースコード

```
const long global_constant_d_32bit = 0xabcdef01;
const short global_constant_b_16bit = 0x2345;
#define global_constant_e_16bit      global_constant_b_16bit

short constant_values(void) {
    static const short local_constant_c_16bit = 0x6789;

    return local_constant_c_16bit;
```

```
}

const char global_constant_a_8bit = 0x01;
```

GDB コンソール

```
(gdb) p &__START_rodata
$1 = (<data variable, no debug info> *) 0x9568
(gdb) p &__END_rodata
$2 = (<variable (not text or data), no debug info> *) 0x9574 "."
(gdb) x/12b &__START_rodata
0x9568:  0x01     0xef     0xcd     0xab     0x45     0x23     0x89     0x67
0x9570:  0x01     0x00     0x00     0x00
```

自動記憶期間をもつ定数は、有効になった時点で生成されます。
これは、GNU17 C コンパイラは、これらの定数をスタックフレームに配置し、それらの初期値を'.rodata'セクションに配置するということです。さらに、GNU17 C コンパイラは、初期値を設定するための S1C17 命令を生成します。

C ソースコード

```
short local_constant_values(void) {
    const short constant_array[4] = {
        0x0123, 0x4567, 0x89ab, 0xcdef,
    };

    return sub_function(constant_array);
}
```

S1C17 命令

```
0x8056:sub.a   %sp,0x8
0x8058:ld.a    %r0,%sp
0x805a:ext     0x12b
0x805c:ld.a    %r1,0xc
0x805e:ext     0x2
0x8060:call.d  0x135    <memcpy>
0x8062:ld      %r2,0x8
0x8064:ld.a    %r0,%sp
0x8066:call    0x3f5
0x8068:add.a   %sp,0x8
0x806a:ret
```

'static'を定数に指定して記憶期間を自動から静的に変更することで、スタックフレームおよび S1C17 命令のサイズを縮小できます。

C ソースコード

```
short local_constant_values(void) {
    static const short constant_array[4] = {
        0x0123, 0x4567, 0x89ab, 0xcdef,
    };

    return sub_function(constant_array);
}
```

S1C17 命令

```
0x8056:ext     0x12a
0x8058:ld.a    %r0,0x6e
0x805a:call    0x3fb
0x805c:ret
```

1.2.5 スタック領域

RAM の一部は、プログラムのコールスタックを保存するためのスタック領域として使用されます。
S1C17 が'call'命令を実行すると、PC レジスタの値が、SP レジスタで保持されるアドレスに保存されます。PC レジスタのサイズは 24 ビットであるため、SP レジスタの値は 32 ビット境界アドレスを指す必要があります。また、SP レジスタの最下位 2 ビットは 0 に固定されます。
そのため、GNU17 C コンパイラは、SP レジスタが 32 ビット境界を示すように関数のスタックフレームのサイズを調整します。つまり、GNU17 C コンパイラが生成する関数のスタックフレームのサイズは 4 の倍数になるので、1～3 バイトずつ縮小することはできません。

1.3 使用されるメモリ容量の確認方法

1.3.1 使用されるROM

GNU17 リンカは、プログラムによる ROM 領域の使用量をリンクマップ情報（*.map）ファイルに示します。リンクマップ情報ファイルの出力については、GNU17 マニュアルの"9.2.2 出力ファイル"を参照してください。

プログラムによる ROM 領域の使用量は、次のように計算できます。

ROM の最後のシンボルのアドレス－ROM の最初のシンボルのアドレス［バイト］

’ROM の最初のシンボルのアドレス’は 0x008000 でしょう。通常、ここは割り込みベクタテーブルです。
’ROM の最後のシンボルのアドレス’は、’.text’セクション、’.rodata’セクションまたは’data_lma’領域（’.data’セクションの初期値）の最後の最上位アドレスでしょう。

セクション／領域	最後のシンボル
’.text’セクション	__END_text
’.rodata’セクション	__END_rodata
’data_lma’領域	__END_data_lma

1.3.2 使用されるRAM

GNU17 リンカは、プログラムに割り当てられた RAM 領域の量（スタック領域以外）をリンクマップ情報に示します。リンクマップ情報の出力については、GNU17 マニュアルの”9.2.2 出力ファイル”を参照してください。

プログラムに割り当てられた RAM 領域の量は、次のように計算できます。

RAM の最後のシンボルのアドレス－RAM の最初のシンボルのアドレス［バイト］

’RAM の最初のシンボルのアドレス’は 0x000000 です。
’RAM の最後のシンボルのアドレス’は、.bss セクションまたは.data セクションの最後の最上位アドレスでしょう。

セクション	最後のシンボル
’.bss’セクション	__END_bss
’.data’セクション	__END_data

1.3.3 プログラムのスタック領域

GNU17 リンカは、GNU17 IDE で設定された値に従い、SP レジスタの初期値を’__START_stack’シンボルとして出力します。SP レジスタの初期値の設定については、GNU17 マニュアルの”5.7.9 リンカスクリプトの編集”を参照してください。
’__START_stack’を SP レジスタの初期値にするためには、プログラムが SP レジスタに設定しなければなりません。

C ソースコード

```
extern unsigned long __START_stack[];
static void boot(void);

#ifdef __POINTER16
#define VECTOR(p)   (p),0
#else
#define VECTOR(p)   (p)
#endif

void * const vector[] = {
    VECTOR(boot),
};
```

S1C17 命令

```
static void boot(void) {
    asm __volatile__("xld.a %sp, __START_stack");
    asm __volatile__("xjpr main");
}
```

```
0x92a6:ext      0x1f
0x92a8:ld.a     %sp,0x40
0x92aa:jpr      0x5
0x92ac:ret
```

GNU17 リンカによりマッピングされるアドレス

```
        0x00000fc0        __START_stack=0xfc0

.vector  0x00008000   0x10
        0x00008000        __START_vector=.
 vector.o(.rodata)
 .rodata 0x00008000   0x10 vector.o
        0x00008000        vector
        0x00008010        __END_vector=.
```

```
.text    0x00008010 0x155c
         0x00008010         __START_text=.

 vector.o(.text)
 .text   0x000092a6   0x16 vector.o
         0x000092b6         main
```

GDB コンソール

```
(gdb) p &__START_stack
$1 = (<variable (not text or data), no debug info> *) 0xfc0 ".,."
(gdb) p &vector
$2 = (void *(*)[1]) 0x8000
(gdb) p main
$3 = {int ()} 0x92b6 <main>
(gdb) x/1w 0x8000
0x8000 <vector>:  0x000092a6
```

各関数のスタックフレームのサイズは、GNU17 C コンパイラにより生成される S1C17 命令から計算できます。

C ソースコード

```
short using_stack(short a, short b, short c, short d) {
    short buffer[5] = {0, b, c, d, a};
    short f;

    f = sub_function(buffer);

    return f + buffer[0];
}
```

S1C17 命令

```
0x9148:ld.a    -[%sp],%r4
0x914a:sub.a   %sp,0xc
0x914c:ld      %r4,0x0
0x914e:ld      [%sp+0x0],%r4
0x9150:ld      [%sp+0x2],%r1
0x9152:ld      [%sp+0x4],%r2
0x9154:ld      [%sp+0x6],%r3
0x9156:ld      [%sp+0x8],%r0
0x9158:ld.a    %r0,%sp
0x915a:call    0x3f3
0x915c:add.a   %sp,0xc
0x915e:ld.a    %r4,[%sp]+
0x9160:ret
```

各関数の最後では、次の命令により、スタックフレームが除去されます。

- 'add.a %sp'命令が、スタックフレームからローカル変数領域を除去します。
- 一部の'ld.a %rd, [%sp]+'命令が、スタックフレームからレジスタ保存領域を除去します。
- 'ret'命令が、スタックフレームから戻りアドレスを除去し、呼び出し側に戻ります。

関数のスタックフレームのサイズは、次のように計算できます。

（ローカル変数領域のサイズ）＋（レジスタ保存領域のサイズ）＋（戻りアドレス）
＝（'add.a %sp, imm'の'imm'）＋（'ld.a %rd, [%sp]+'の数）×4＋4［バイト］

ある関数におけるスタック領域の使用量は、次のように計算できます。

"その関数の呼び出し側の最大使用スタック領域"＋"その関数のスタックフレーム"

プログラムに対するスタック領域の量は、次のように計算できます。

"S1C17 リセットベクタから呼び出される関数の最大使用スタック領域"
＋"S1C17 割り込みハンドラから呼び出される関数の最大使用スタック領域"

スタック領域のおおよそのサイズは、次の手順で確認できます。

- GNU17 GDB を使用して、プログラムをターゲットデバイスに転送します。
- GDB コンソールで'c17 fw'コマンドを入力し、（GNU17 リンカにより割り当てられた RAM 領域ではない）スタック領域を任意の値で埋めます。
- プログラムを実行し、一時停止します。
- SP レジスタおよびスタック領域を確認します。'c17 fw'コマンドで埋めた値から変更されている領域が、スタックとして使用されている領域です。

2. GNU17 IDE およびツールの設定

この項では、メモリの使用量に影響を与える GNU17 の設定について説明します。
GNU17 IDE およびツールの設定の詳細については、GNU17 マニュアルの”5 GNU17 IDE”を参照してください。

2.1 メモリモデルの選択

メモリモデルは、新しいプロジェクトの作成時に選択できます。また、プロジェクトの［プロパティ］ダイアログボックスで設定できます。
“MIDDLE”モデルで使用されるメモリ容量は、”REGULAR”モデルより小さく、”SMALL”モデルより大きくなります。

- ”REGULAR”モデル（24 ビット空間）：
  アドレス空間は制限されませんが、他のモデルより多くのメモリが必要になります。

- “MIDDLE”モデル（20 ビット空間）：
  プログラムにより使用されるアドレス空間は、20 ビット範囲に制限されます。つまり、プログラムで使用される RAM と ROM およびすべての周辺回路制御レジスタを、0x000000～0x0FFFFF にマッピングしなければなりません。

C ソースコード

```
int middle_model(int * data, int * array, int offset) {
    int r;
    void * buffer[256];

    memcpy(buffer ,data, sizeof(buffer));
    r = sub_function(buffer);

    return r + array[offset];
}
```

S1C17 命令（ミドルモデル）

```
0x8eae:   ld.a    -[%sp],%r4
0x8eb0:   ld.a    -[%sp],%r5
0x907e:   ld.a    -[%sp],%r4
0x9080:   ld.a    -[%sp],%r5
0x9082:   ld.a    -[%sp],%r6
0x9084:   ld.a    -[%sp],%r7
0x9086:   ext     0x8
0x9088:   sub.a   %sp,0x4
0x908a:   ext     0x8
0x908c:   ld.a    [%sp+0x0],%r0
0x908e:   ld.a    %r4,%r1
0x9090:   ld      %r6,%r2
0x9092:   ld.a    %r0,%sp
0x9094:   ext     0x8
0x9096:   ld.a    %r1,[%sp+0x0]
0x9098:   ext     0x8
0x909a:   ld      %r2,0x0
0x909c:   call    0x152    <memcpy>
0x909e:   ld.a    %r0,%sp
0x90a0:   call    0x3ec
0x90a2:   cv.as   %r6,%r6
0x90a4:   add.a   %r6,%r6
0x90a6:   add.a   %r4,%r6
0x90a8:   ld      %r2,[%r4]
0x90aa:   add     %r0,%r2
0x90ac:   ext     0x8
0x90ae:   add.a   %sp,0x4
0x90b0:   ld.a    %r7,[%sp]+
0x90b2:   ld.a    %r6,[%sp]+
0x90b4:   ld.a    %r5,[%sp]+
0x90b6:   ld.a    %r4,[%sp]+
0x90b8:   ret
```

- ”SMALL”モデル（16 ビット空間）：
  C 言語のポインタ型のサイズは 16 ビットです。プログラムにより使用されるアドレス空間は、16 ビットの範囲に制限されます。つまり、プログラムで使用される RAM と ROM およびすべての周辺回路制御レジスタを、0x000000～0x00FFFF にマッピングしなければなりません。
  ポインタ型のサイズは 24 ビット（メモリ上では 32 ビット）から 16 ビットになるので、メモリの使用量は、他のモデルより小さくなります。さらに、16 ビットポインタでは、ポインタ型と整数型との間の操作が簡単になるため、S1C17 命令の数が減ります。

C ソースコード　　S1C17 命令（スモールモデル）

```
int small_model(int * data, int * array, int offset) {
    int r;
    void * buffer[256];

    memcpy(buffer ,data, sizeof(buffer));
    r = sub_function(buffer);

    return r + array[offset];
}
```

```
0x8eae:    ld.a    -[%sp],%r4
0x8eb0:    ld.a    -[%sp],%r5
0x8eb2:    ext     0x4
0x8eb4:    sub.a   %sp,0x0
0x8eb6:    ld      %r3,%r0
0x8eb8:    ld      %r5,%r1
0x8eba:    ld      %r4,%r2
0x8ebc:    ld.a    %r0,%sp
0x8ebe:    ext     0x4
0x8ec0:    ld      %r2,0x0
0x8ec2:    call.d  0x113    <memcpy>
0x8ec4:    ld      %r1,%r3
0x8ec6:    ld.a    %r0,%sp
0x8ec8:    call    0x3f0
0x8eca:    sl      %r4,0x1
0x8ecc:    add     %r4,%r5
0x8ece:    ld      %r2,[%r4]
0x8ed0:    add     %r0,%r2
0x8ed2:    ext     0x4
0x8ed4:    add.a   %sp,0x0
0x8ed6:    ld.a    %r5,[%sp]+
0x8ed8:    ld.a    %r4,[%sp]+
0x8eda:    ret
```

詳細については、GNU17 マニュアルの”5.4.2 新規プロジェクトの作成”、”5.4.11 プロジェクトプロパティ”および”6.3.2 コマンドラインオプション”を参照してください。

2.2 プログラムのセクションの配置

呼び出される側の関数が呼び出し側のアドレスの近くにある場合、GNU17 で生成される S1C17 命令の数は減ります。
GNU17 IDE が C ソースファイルをファイル名のアルファベット順にリンクするため、呼び出し側と呼び出される側が離れて再配置されることがあります。関数を近くのアドレスに配置するには、C ソースファイルを置くためのセクションを作成する必要があります。ただし、このような配置による効果はそれほど大きくはありません。
新しいセクションの作成方法および編集方法については、GNU17 マニュアルの”5.7.9 リンカスクリプトの編集”を参照してください。

3. 基本操作

この項では、C 言語の基本操作に対して GNU17 が出力する S1C17 命令について説明します。
詳細については、GNU17 マニュアルの”6.4 コンパイラの出力”を参照してください。

3.1 整数型の操作

3.1.1 8 ビットおよび 16 ビット整数型の操作

GNU17 C コンパイラは、次のデータ型を S1C17 の 8 ビットまたは 16 ビット整数として扱います。

- ’char’ 8 ビット（’signed char’）
- ’unsigned char’ 8 ビット
- ’short’ 16 ビット（’signed short’）
- ’unsigned short’ 16 ビット
- ’int’ 16 ビット（’singed int’）
- ’unsigned int’ 16 ビット

S1C17 は、これらのデータ型を効率的に操作できます。

3.1.1.1 一般的なメモリおよびレジスタ間でのデータ転送

S1C17 は、’ld’命令を使用して、メモリとレジスタの間で 8 ビットまたは 16 ビット値を転送します。転送元メモリから宛先メモリに値を転送する手順は、転送元から値を読み込む’ld’命令と宛先に値を書き込む’ld’命令の 2 つの S1C17 命令を使用します。

C ソースコード

```
short transmit_pointed_value(short * a, short * b) {
    short c;

    c = *b;
    *a = c;

    return c;
}
```

S1C17 命令

```
0x828e:ld     %r2,[%r1]
0x8290:ld     [%r0],%r2
0x8292:ld     %r0,%r2
0x8294:ret
```

レジスタ値の 8KB 以内の範囲にあるデータにアクセスするための S1C17 命令の数は 2 つです。GNU17 C コンパイラは、これらの命令を使用して、ポインタからのオフセットおよび構造体のメンバにアクセスします。

C ソースコード

```
short transmit_offset_value(short * a, short * b) {
    short c;

    c = *(b + 1);
    *(a + 256) = c;

    return c;
}
```

S1C17 命令

```
0x828e:ext    0x2
0x8290:ld     %r2,[%r1]
0x8292:ext    0x200
0x8294:ld     [%r0],%r2
0x8296:ld     %r0,%r2
0x8298:ret
```

メモリの値を操作するため、S1C17 は値をレジスタに読み込みます。そのため、メモリの値を操作するには、レジスタの値を操作するよりも多くの S1C17 命令が必要になります。

C ソースコード

```
void copy_triple(short * a, short * b) {
    *(a + 1) = *(b + 1);
    *(a + 2) = *(b + 1);
    *(a + 3) = *(b + 1);
}
```

S1C17 命令

```
0x828e:ext    0x2
0x8290:ld     %r2,[%r1]
0x8292:ext    0x2
0x8294:ld     [%r0],%r2
0x8296:ext    0x2
0x8298:ld     %r2,[%r1]
0x829a:ext    0x4
0x829c:ld     [%r0],%r2
```

	0x829e:ext 0x2 0x82a0:ld %r2,[%r1] 0x82a2:ext 0x6 0x82a4:ld [%r0],%r2 0x82a6:ret

メモリからレジスタに読み込まれた値が繰り返し使用される場合、S1C17 命令の数は減ります。

C ソースコード	S1C17 命令

void copy_triple(short * a, short * b) {
short c;
 	
c = *(b + 1);
*(a + 1) = c;
*(a + 2) = c;
*(a + 3) = c;
}
	0x828e:ext 0x2
	0x8290:ld %r2,[%r1]
	0x8292:ext 0x2
	0x8294:ld [%r0],%r2
	0x8296:ext 0x4
	0x8298:ld [%r0],%r2
	0x829a:ext 0x6
	0x829c:ld [%r0],%r2
	0x829e:ret

'ld'命令は、宛先レジスタの値を増加（または減少）できます。C ソースコードが後置増分演算子を使用してポインタにより記述される場合、S1C17 命令の数を減らすことができます。

C ソースコード	S1C17 命令

void copy_triple(short * a, short * b) {
short * p;
short c;
 	
c = *(b + 1);
p = a + 1;
*p++ = c;
*p++ = c;
*p++ = c;
}
	0x828e:ext 0x2
	0x8290:ld %r2,[%r1]
	0x8292:add %r0,0x2
	0x8294:ld [%r0]+,%r2
	0x8296:ld [%r0]+,%r2
	0x8298:ld [%r0],%r2
	0x829a:ret

3.1.1.2 配列要素およびレジスタ間でのデータ転送

配列のインデックスが変数の場合、GNU17 C コンパイラは、インデックス番号を配列（ポインタ）に加えて、配列要素のアドレスを計算します。

C ソースコード	S1C17 命令

short array_access(short * a, short b) {
return a[b];
}
	0x828e:sl %r1,0x1
	0x8290:add %r1,%r0
	0x8292:ld %r0,[%r1]
	0x8294:ret

配列のインデックスが定数値である場合、S1C17 命令および使用されるレジスタの数が減る場合があります。

3.1.1.3 データ操作

S1C17 は 1 命令で、16 ビット整数に関する C 言語のいくつかの操作（反転、負数、加算、乗算、不等式、等式、AND、OR、排他的 OR）を処理できます。これらの操作で他の操作を置き換えることができれば、S1C17 命令の数が減る場合があります。

S1C17 は、16 ビット整数の左シフトおよび右シフトを 1 命令で処理できます。ただし、S1C17 命令のシフト幅は、0～4 および 8 に制限されます。
シフト幅が C ソースコードの変数で指定されている場合、GNU17 C コンパイラは、GNU17 エミュレーションライブラリ（libgcc*.a）で定義されているシフト関数を呼び出す S1C17 命令を出力します。

C ソースコード	S1C17 命令

short shift_variable(short a, short b) {
return a << b;
}
	0x8012:ext 0x2
	0x8014:call 0x8d <__ashlhi3>
	0x8016:ret

シフト幅が定数値で指定されている場合、シフトのための S1C17 命令の数は 1～3 です。

C ソースコード	S1C17 命令

short shift_constant(short a) {
  return a << 7;
}
	0x8018:sl %r0,0x4
	0x801a:sl %r0,0x3
	0x801c:ret

シフトのための S1C17 命令の数は、状況により異なりますが、定数によるシフトの方が S1C17 のサイクルは少なくなります。

S1C17 は、乗算、除算およびモジュラ操作を 1 命令で処理できません。これらの操作を実行するために、GNU17 C コンパイラは、GNU17 エミュレーションライブラリで定義されている関数を呼び出す S1C17 命令を出力します。

C ソースコード	S1C17 命令

short calling_emulation_library(short a) {
  return a / 250;
}
	0x801e:ext 0x1
	0x8020:ld %r1,0x7a
	0x8022:ext 0x2
	0x8024:call 0xed <__divhi3>
	0x8026:ret

これらの操作では、GNU17 エミュレーションライブラリの関数をリンクして呼び出す命令のために、使用されるメモリおよび S1C17 のサイクルが他の操作より増えます。

3.1.1.4 8 ビットと 16 ビット整数間におけるデータ型の変換

S1C17 は、ほとんどの操作を 16 ビット整数として実行します。
S1C17 は、8 ビット整数をレジスタ上で 16 ビット整数として操作し、'ld'命令により 16 ビット整数を 8 ビット整数に変換します。
そのため、C ソースコード上で 8 ビット整数型の変数が計算に使用される場合、GNU17 C コンパイラは、16 ビット整数を 8 ビット整数に変換する S1C17 命令を出力します。

C ソースコード	S1C17 命令

char byte_add_and_convert(char a, short b) {
  return a + b;
}
	0x801e:add %r0,%r1
	0x8020:ld.b %r0,%r0
	0x8022:ret

すべての変数が 16 ビット整数の場合、16 ビット整数を 8 ビット整数に変換する必要はありません。

C ソースコード	S1C17 命令

short short_add_and_no_convert(short a, short b) {
  return a + b;
}
	0x8024:add %r0,%r1
	0x8026:ret

3.1.2 32 ビット整数型の操作

GNU17 C コンパイラは、次のデータ型を S1C17 の 32 ビット整数として扱います。

- 'long' ('signed long')
- 'unsigned long'

32 ビット整数型を操作する S1C17 命令はありません。GNU17 C コンパイラは、2 つの S1C17 レジスタを組み合わせて 32 ビット整数を操作する S1C17 命令を出力します。
GNU17 での 32 ビット整数の操作には、8 ビットおよび 16 ビットの操作より多くのメモリおよび S1C17 レジスタが必要になります。32 ビット操作を 16 ビット操作に置き換えると、プログラムのサイズを縮小できます。

3.1.2.1 一般的なメモリおよびレジスタ間でのデータ転送

GNU17 C コンパイラは、2 つの'ld'命令を出力して、メモリおよびレジスタ間で 32 ビット値を転送しま

す。つまり、32 ビット整数の操作には、2 倍の命令およびレジスタが必要になります。

C ソースコード	S1C17 命令

void transmit_pointed_long(long * a, long * b) {
  *b = *a;
}
	0x82ac:ld %r2,[%r0]
	0x82ae:ext 0x2
	0x82b0:ld %r3,[%r0]
	0x82b2:ld [%r1],%r2
	0x82b4:ext 0x2
	0x82b6:ld [%r1],%r3
	0x82b8:ret

32 ビット型のポインタを 16 ビット整数型のポインタのように扱うことで、GNU17 C コンパイラにより出力される S1C17 命令の数を減らすことができます。

C ソースコード	S1C17 命令

void transmit_pointed_long(long * a, long * b) {
  short * c;
  short * d;
 	
  c = (short *)a;
  d = (short *)b;
  *d++ = *c++;
  *d++ = *c++;
}
	0x82ac:ld %r2,[%r0]+
	0x82ae:ld [%r1]+,%r2
	0x82b0:ld %r2,[%r0]
	0x82b2:ld [%r1],%r2
	0x82b4:ret

3.1.2.2 配列要素およびレジスタ間でのデータ転送

配列のインデックスが 32 ビット変数の場合、GNU17 C コンパイラは、インデックス番号を配列アドレスに加えて、配列要素のアドレスを計算します。

C ソースコード	S1C17 命令（ミドルモデル）

short long_array_access(long i, short * a) {
  return a[i];
}
	0x9124:cv.al %r0,%r1
	0x9126:ld.a %r0,%r0
	0x9128:add.a %r0,%r0
	0x912a:add.a %r2,%r0
	0x912c:ld %r0,[%r2]
	0x912e:ret

“SMALL”モデルプログラムでは、アドレス空間が 16 ビット範囲に制限されるため、配列要素のアドレス計算のために GNU17 C コンパイラが出力する S1C17 命令の数は減ります。

C ソースコード	S1C17 命令（スモールモデル）

short long_array_access(long i, short * a) {
  return a[i];
}
	0x9124:ld %r3,%r0
	0x9128:sl %r3,0x1
	0x912a:add %r3,%r2
	0x912c:ld %r0,[%r3]
	0x912e:ret

配列のインデックスが 16 ビット変数の場合、GNU17 C コンパイラはより少ない S1C17 命令を出力し、S1C17 レジスタを節約します。

C ソースコード	S1C17 命令（スモールモデル）

short short_array_access(short i, short * a) {
  return a[i];
}
	0x9130:sl %r0,0x1
	0x9132:add %r0,%r1
	0x9134:ld %r0,[%r0]
	0x9136:ret

3.1.2.3 データ操作

32 ビット整数を計算するために、GNU17 C コンパイラは、GNU17 エミュレーションライブラリ（libgcc*.a）で定義されている関数を呼び出す S1C17 命令を出力します。同様にして、32 ビット整数を含む比較はより多くの S1C17 命令を必要とします。

C ソースコード	S1C17 命令

```
short compare_long(long a) {
    short b;

    if (a > 0) {
        b = 0;
    } else {
        b = 1;
    }

    return b;
}
```

```
0x8076:cmp     %r1,0x0
0x8078:jrgt.d  0x4
0x807a:cmp     %r1,0x0
0x807c:jrne.d  0x4
0x807e:cmp     %r0,0x0
0x8080:jrule   0x2
0x8082:jpr.d   0x2
0x8084:ld      %r0,0x0
0x8086:ld      %r0,0x1
0x8088:ret
```

値が 16 ビット範囲内にある場合、32 ビット整数型ではなく 16 ビット整数型を使用することで、GNU17 C コンパイラにより出力される S1C17 命令の数を減らすことができます。

C ソースコード

```
short compare_long(long a) {
    short b;

    b = (short)a;
    if (b > 0) {
        b = 0;
    } else {
        b = 1;
    }

    return b;
}
```

S1C17 命令

```
0x8076:cmp     %r0,0x0
0x8078:jrgt.d  0x2
0x807a:ld      %r0,0x0
0x807c:ld      %r0,0x1
0x807e:ret
```

3.1.2.4 32 ビットとその他の整数間におけるデータ型の変換

S1C17 は 32 ビット整数型と他の整数型の値を数命令で変換できます。
32 ビット整数型の値を他のデータ型に変換すれば、GNU17 C コンパイラにより生成される S1C17 命令の数は減ります。

C ソースコード	S1C17 命令

`char convert_long_to_char(long a) {`
`  return (char)a;`
`}`
	`0x82d0:ld.b  %r0,%r0`
	`0x82d2:ret`

`short convert_long_to_short(long a) {`
`  return (short)a;`
`}`
	`0x82d4:ret`

`long convert_char_to_long(char a) {`
`  return a;`
`}`
	`0x82d6:ld.b  %r0,%r0`
	`0x82d8:cv.ls  %r1,%r0`
	`0x82da:ret`

`long convert_short_to_long(short a) {`
`  return a;`
`}`
	`0x82dc:ld  %r0,%r0`
	`0x82de:cv.ls  %r1,%r0`
	`0x82e0:ret`

操作結果に影響が及ばないのであれば、32 ビット整数を 16 ビット整数に変換できるでしょう。

3.1.3 ポインタ型の操作

通常、S1C17 アドレスは、24 ビット整数値（24 ビットポインタ）です。つまり、GNU17 C コンパイラは、C 言語のポインタを S1C17 の 24 ビット整数値として扱います。ただし、”SMALL”モデルが GNU17 IDE で選択されている場合、GNU17 C コンパイラは、ポインタを 16 ビット整数値（16 ビットポインタ）として扱います。
プログラムのすべてのオブジェクトおよび関数が 16 ビット空間（64KB）内にある場合、”SMALL”モデルを選択することをお勧めします。これは、16 ビットポインタでは、必要な S1C17 命令およびレジスタが少ないためです。

この項では、"REGULAR”モデルまたは”MIDDLE”モデルが GNU17 IDE で選択されている場合の 24 ビットポインタの操作について説明します。

3.1.3.1 メモリおよびレジスタ間でのデータ転送

S1C17 は、16 ビット値の場合と同様に、'ld'命令によりメモリおよびレジスタ間で 24 ビット値を転送します。転送元メモリから宛先メモリに値を転送する手順は、転送元から値を読み込む'ld'命令と宛先に値を書き込む'ld'命令の 2 つの S1C17 命令を使用します。

C ソースコード

```
short * copy_pointed_address(short ** a, short ** b) {
    short * c;

    c = *b;
    *a = c;

    return c;
}
```

S1C17 命令

```
0x9162:ld.a    -[%sp],%r4
0x9164:ld.a    %r3,%r0
0x9166:ld.a    %r0,[%r1]
0x9168:ld.a    [%r3],%r0
0x916a:ld.a    %r4,[%sp]+
0x916c:ret
```

3.1.3.2 データ操作

S1C17 は 1 命令で 24 ビットポインタに関する C 言語のいくつかの操作（加算、減算および比較）を処理できます。
配列のインデックスが変数の場合、GNU17 C コンパイラは、24 ビットの加算により配列要素のアドレスを計算します。

C ソースコード

```
char array_access(char * a, short b) {
    return a[b];
}
```

S1C17 命令

```
0x836a:ld.a    %r2,%r0
0x836c:cv.as   %r0,%r1
0x836e:add.a   %r2,%r0
0x8370:ld      %r0,[%r2]
0x8372:ret
```

ポインタの減算は、整数拡張的に 32 ビット整数型として計算されます。これは、GNU17 C コンパイラが 16 ビット整数に結果を保存できないためです。

C ソースコード

```
short pointer_subtraction(short * a, short * b) {
    return a - b;
}
```

S1C17 命令

```
0x9180:ld.a    -[%sp],%r4
0x9182:ld.a    -[%sp],%r5
0x9184:ld.a    -[%sp],%r6
0x9186:ld.a    %r5,%r1
0x9188:ld.a    %r0,%r0
0x918a:cv.la   %r1,%r0
0x918c:ld.a    %r2,%r5
0x918e:cv.la   %r3,%r2
0x9190:sub     %r0,%r2
0x9192:sbc     %r1,%r3
0x9194:call.d  0x196    <__ashrsi3>
0x9196:ld      %r2,0x1
0x9198:ld.a    %r6,[%sp]+
0x919a:ld.a    %r5,[%sp]+
0x919c:ld.a    %r4,[%sp]+
0x919e:ret
```

2 つのポインタが近い場合、ポインタを 16 ビット整数に変換した後で減算することで、S1C17 命令の数を減らすことができます。

C ソースコード

```
short pointer_subtraction(short * a, short * b) {
    return (short)a – (short)b;
}
```

S1C17 命令

```
0x9180:ld.a    -[%sp],%r4
0x9182:sub     %r0,%r1
0x9184:ld.a    %r4,[%sp]+
0x9186:ret
```

3.1.3.3 24 ビットポインタと他の整数間におけるデータ型の変換

S1C17 は 24 ビット整数型と他の整数型の値を数命令で変換できます。
24 ビット整数型の値を他のデータ型に変換しておけば、GNU17 C コンパイラにより生成される S1C17

命令の数が減ることがあります。

3.2 浮動小数点数

浮動小数点数を直接操作する S1C17 命令はありません。GNU17 C コンパイラは、'float'を 4 バイトデータ、'double'を 8 バイトデータとして表し、GNU17 エミュレーションライブラリで定義されている関数を呼び出してこれらを操作することで、浮動小数点数をサポートします。

GNU17 での浮動小数手点の操作では、多くのメモリ、S1C17 レジスタおよびサイクルが必要になります。浮動小数点数の操作を固定小数点数の操作に置き換えることで、プログラムのサイズを縮小できます。

3.3 算術演算

3.3.1 データ型の選択

可能であれば、算術演算の実行には 16 ビット整数型を選択してください。

- 32 ビット整数は 4 バイトのメモリを使用しますが、16 ビット整数が使用するメモリは 2 バイトです。
- 32 ビット整数のほとんどの演算は関数呼び出しとして実行されるため、32 ビット整数によって、S1C17 命令の数が増えます。
- 16 ビット整数が使用するレジスタが 1 つであるのに対して 32 ビット整数では 2 つであるため、32 ビット整数によって、スタックフレームのサイズが大きくなります。
- 32 ビット整数では、メモリとレジスタ間でのデータ転送に、多くの S1C17 命令が必要になります。

8 ビット整数がメモリへのデータ保存に使用される場合、データに使用されるメモリ容量は減ります。

3.3.2 演算の簡素化

GNU17 C コンパイラは、変数と定数の間の乗算を他の演算に置き換えることがあります。

C ソースコード

```
short multiple_249(short a) {
    return a * 249;
}
```

S1C17 命令

```
0x8078:ld     %r2,%r0
0x807a:sl     %r0,0x4
0x807c:sl     %r0,0x1
0x807e:sub    %r0,%r2
0x8080:sl     %r0,0x3
0x8082:add    %r0,%r2
0x8084:ret
```

最適化のために、GNU17 C コンパイラは、乗算、除算およびモジュラについて置き換えを行います。これらの最適化により、GNU17 エミュレーションライブラリで定義されている関数を呼び出す場合よりも多くの S1C17 命令が出力される場合があります。

C ソースコード

```
short multiple_249(short a) {
    extern short __mulhi3(short, short);
    return __mulhi3(a, 249);
}
```

S1C17 命令

```
0x8078:ext    0x1
0x807a:ld     %r1,0x79
0x807c:ext    0x2
0x807e:call   0x2f0
0x8080:ret
```

計算誤差が許容可能であるなら、S1C17 命令の数は減ります。

C ソースコード

```
short multiple_249(short a) {
    return a * 256;
}
```

S1C17 命令

```
0x8078:sl     %r0,0x8
0x807a:ret
```

GNU17 C コンパイラの最適化は、それほど強力ではありません。

C ソースコード

```
short not_optimaized_procedure(short a, short b) {
    short c;

    if (a >= 15) {
        c = (a - 15) << 8;
    } else {
        c = 0x7000 + (a << 8);
    }

    return c;
}
```

S1C17 命令

```
0x8044:cmp     %r0,0xe
0x8046:jrle.d  0x4
0x8048:ld      %r2,0x71
0x804a:add     %r0,%r2
0x804c:jpr.d   0x4
0x804e:sl      %r0,0x8
0x8050:sl      %r0,0x8
0x8052:ext     0xe0
0x8054:add     %r0,0x0
0x8056:ret
```

共通な計算を手順から抽出することで、S1C17 命令の数を減らすことができます。

C ソースコード

```
short not_optimized_procedure(short a, short b) {
    if (a >= 15) {
        a -= 15;
    } else {
        a += 0x70;
    }
    return a << 8;
}
```

S1C17 命令

```
0x8044:cmp     %r0,0xe
0x8046:jrle.d  0x3
0x8048:ld      %r2,0x71
0x804a:jpr.d   0x2
0x804c:add     %r0,%r2
0x804e:add     %r0,0x70
0x8050:sl      %r0,0x8
0x8052:ret
```

3.3.3 計算データの埋め込み

複雑な計算の一部である値を、GNU17 C コンパイラが C ソースコードをコンパイルするときに計算できるのであれば、S1C17 命令の数を減らすことができます。

C ソースコード

```
short multiple_and_add(short a, short b) {
    return a * a * a + b;
}
```

S1C17 命令

```
0x80c4:ld.a    -[%sp],%r4
0x80c6:ld.a    -[%sp],%r5
0x80c8:ld      %r4,%r0
0x80ca:ld      %r5,%r1
0x80cc:ext     0x2
0x80ce:call.d  0x1ba    <__mulhi3>
0x80d0:ld      %r1,%r0
0x80d2:ext     0x2
0x80d4:call.d  0x1b7    <__mulhi3>
0x80d6:ld      %r1,%r4
0x80d8:add     %r0,%r5
0x80da:ld.a    %r5,[%sp]+
0x80dc:ld.a    %r4,[%sp]+
0x80de:ret
```

あらかじめ計算されて定数の表に埋め込まれた値を参照することで計算を実行するように、プログラムを変更します。

C ソースコード

```
short multiple_and_add(short a, short b) {
    static const char calculated_values[] = {
        0 * 0 * 0, 1 * 1 * 1,
        2 * 2 * 2, 3 * 3 * 3,
        4 * 4 * 4, 5 * 5 * 5,
    };

    return calculated_values[a] + b;
}
```

S1C17 命令

```
0x80c4:ext     0x129
0x80c6:ld      %r2,0x68
0x80c8:add     %r0,%r2
0x80ca:ld.b    %r0,[%r0]
0x80cc:add     %r0,%r1
0x80ce:ret
```

ただし、埋め込まれたデータのサイズ以上に、命令数が減らなければなりません。

3.4 制御ステートメント

3.4.1 データ型の選択

可能であれば、比較の実行には 16 ビット整数型を選択してください。S1C17 は、1 命令で 32 ビット整数型を比較できません。

C ソースコード

```
short sum_of_array(long a, int * b) {
    long i;
    short r;

    r = 0;
    for (i = 0; i < a; i++) {
        r += *b++;
    }

    return r;
}
```

S1C17 命令

```
0x812e:ld.a    -[%sp],%r4
0x8130:ld.a    -[%sp],%r5
0x8132:ld.a    -[%sp],%r6
0x8134:ld      %r6,0x0
0x8136:sub     %r4,%r4
0x8138:sub     %r5,%r5
0x813a:cmp     %r1,%r6
0x813c:jrgt.d  0x4
0x813e:ld      %r3,%r2
0x8140:jrne.d  0xc
0x8142:cmp     %r0,%r4
0x8144:jrule   0xa
0x8146:ld      %r2,[%r3]+
0x8148:add     %r4,0x1
0x814a:adc     %r5,0x0
0x814c:cmp     %r1,%r5
0x814e:jrgt.d  0x7b
0x8150:add     %r6,%r2
0x8152:cmp     %r1,%r5
0x8154:jrne.d  0x2
0x8156:cmp     %r0,%r4
0x8158:jrugt   0x76
0x815a:ld      %r0,%r6
0x815c:ld.a    %r6,[%sp]+
0x815e:ld.a    %r5,[%sp]+
0x8160:ld.a    %r4,[%sp]+
0x8162:ret
```

32 ビット整数間の比較を 16 ビット整数に置き換えると、S1C17 命令の数を減らすことができます。

C ソースコード

```
short sum_of_array(long a, int * b) {
    short i;
    short r;

    r = 0;
    for (i = 0; i < (short)a; i++) {
        r += *b++;
    }

    return r;
}
```

S1C17 命令

```
0x812e:    ld.a    -[%sp],%r4
0x8130:    ld.a    -[%sp],%r5
0x8132:    ld      %r5,%r1
0x8134:    ld      %r4,%r0
0x8136:    ld      %r0,%r2
0x8138:    ld      %r1,0x0
0x813a:    cmp     %r1,%r4
0x813c:    jrge.d  0x6
0x813e:    ld      %r3,%r1
0x8140:    ld      %r2,[%r0]+
0x8142:    add     %r3,0x1
0x8144:    cmp     %r3,%r4
0x8146:    jrlt.d  0x7c
0x8148:    add     %r1,%r2
0x814a:    ld      %r0,%r1
0x814c:    ld.a    %r5,[%sp]+
0x814e:    ld.a    %r4,[%sp]+
0x8150:    ret
```

3.4.2 比較の簡素化

変数の比較に必要な S1C17 命令およびレジスタの数は、定数と変数の間の比較より多くなります。たとえば、継続条件式に 0 を使用することで、命令の数を減らすことができます。

C ソースコード

```
short sum_of_array(short a, int * b) {
    short r;

    for (r = 0; a > 0; a--) {
```

S1C17 命令

```
0x812e:ld      %r3,0x0
0x8130:cmp     %r0,%r3
0x8132:jrle    0x6
0x8134:ld      %r2,[%r1]+
```

```
        r += *b++;
    }

    return r;
}
```

```
0x8136:add    %r3,%r2
0x8138:ld     %r2,0x7f
0x813a:add    %r0,%r2
0x813c:cmp    %r0,0x0
0x813e:jrgt   0x7a
0x8140:ld     %r0,%r3
0x8142:ret
```

3.4.3 条件の変更

条件が常に真であれば、より少ない命令で実行されるように条件を変更できます。たとえば、カウンタが 0 ではないなら、最初の 0 との比較を省略することで、S1C17 命令の数を減らすことができます。

C ソースコード

```
short sum_of_array(short a, int * b) {
    short s;

    s = 0;
    do {
        s += *b++;
    } while (--a > 0);

    return s;
}
```

S1C17 命令

```
0x8262:ld     %r3,0x0
0x8264:ld     %r2,[%r1]+
0x8266:add    %r3,%r2
0x8268:ld     %r2,0x7f
0x826a:add    %r0,%r2
0x826c:cmp    %r0,0x0
0x826e:jrgt   0x7a
0x8270:ld     %r0,%r3
0x8272:ret
```

次の例では、継続条件でカウンタを使用しないことで、S1C17 命令の数を減らしています。

C ソースコード

```
short sum_of_array (int * b) {
    short a;
    short s;

    s = 0;
    do {
        a += *b++;
        s += a;
    } while (a != 0);

    return s;
}
```

S1C17 命令

```
0x8274:ld     %r3,0x0
0x8276:ld     %r2,[%r0]+
0x8278:cmp    %r2,0x0
0x827a:jrne.d 0x7d
0x827c:add    %r3,%r2
0x827e:ld     %r0,%r3
0x8280:ret
```

3.4.4 ループの変換

似たような複数のループを統合することで、S1C17 命令の数を減らすことができます。

C ソースコード

```
short multi_loops(short * a, short * b) {
    short i;
    short r;

    r = 0;
    for (i = 10; i > 0; i--) {
        r += sub_function(*a++);
    }
    for (i = 10; i > 0; i--) {
        r += sub_function(*b++);
    }

    return r;
}
```

S1C17 命令

```
0x8284:ld.a   -[%sp],%r4
0x8286:ld.a   -[%sp],%r5
0x8288:ld.a   -[%sp],%r6
0x828a:ld.a   -[%sp],%r7
0x828c:ld     %r5,%r0
0x828e:ld     %r7,%r1
0x8290:ld     %r6,0x0
0x8292:ld     %r4,0xa
0x8294:ld     %r0,[%r5]+
0x8296:call   0x3f5
0x8298:ld     %r2,0x7f
0x829a:add    %r4,%r2
0x829c:cmp    %r4,0x0
0x829e:jrgt.d 0x7a
0x82a0:add    %r6,%r0
0x82a2:ld     %r4,0xa
0x82a4:ld     %r0,[%r7]+
0x82a6:call   0x3ed
0x82a8:ld     %r2,0x7f
0x82aa:add    %r4,%r2
0x82ac:cmp    %r4,0x0
0x82ae:jrgt.d 0x7a
0x82b0:add    %r6,%r0
```

	0x82b2:ld %r0,%r6 0x82b4:ld.a %r7,[%sp]+ 0x82b6:ld.a %r6,[%sp]+ 0x82b8:ld.a %r5,[%sp]+ 0x82ba:ld.a %r4,[%sp]+ 0x82bc:ret

たとえば、2 つの’for’ループを 1 つの’for’ループにまとめることで、S1C17 命令の数を減らすことができます。

C ソースコード	S1C17 命令

short multi_loops(short * a, short * b) {
  short i;
  short r;
 	
  r = 0;
  for (i = 10; i > 0; i--) {
  r += sub_function(*a++);
  r += sub_function(*b++);
  }
 	
  return r;
}
	0x82be:ld.a -[%sp],%r4
	0x82c0:ld.a -[%sp],%r5
	0x82c2:ld.a -[%sp],%r6
	0x82c4:ld.a -[%sp],%r7
	0x82c6:ld %r7,%r0
	0x82c8:ld %r6,%r1
	0x82ca:ld %r4,0x0
	0x82cc:ld %r5,0xa
	0x82ce:ld %r0,[%r7]+
	0x82d0:call 0x3d8
	0x82d2:add %r4,%r0
	0x82d4:ld %r0,[%r6]+
	0x82d6:call 0x3d5
	0x82d8:ld %r2,0x7f
	0x82da:add %r5,%r2
	0x82dc:cmp %r5,0x0
	0x82de:jrgt.d 0x77
	0x82e0:add %r4,%r0
	0x82e2:ld %r0,%r4
	0x82e4:ld.a %r7,[%sp]+
	0x82e6:ld.a %r6,[%sp]+
	0x82e8:ld.a %r5,[%sp]+
	0x82ea:ld.a %r4,[%sp]+
	0x82ec:ret

3.5 I/Oレジスタへのアクセス

3.5.1 データ型の選択

I/O レジスタにアクセスするにはハードウェア仕様で指定されたビット幅を用いてください。ただし、S1C17 に埋め込まれるほとんどの I/O レジスタは、8 ビット幅および 16 ビット幅の両方にアクセスできます。

ビットフィールドが 16 ビット整数型として定義されている場合でも、ビット幅が 8 以下のフィールドは、GNU17 C コンパイラにより、8 ビット整数型としてアクセスされます。

C ソースコード	S1C17 命令

typedef union {
  unsigned short word;
  unsigned char byte[2];
  struct {
  unsigned short a: 1;
  unsigned short b: 7;
  unsigned short c: 3;
  unsigned short d: 5;
  } bit;
} short_fields;
 	
#define device16 (*(volatile short_fields *)0x4000)	

unsigned short get_device_a(void) {
  return device16.bit.a;
}
	0x8fc0:ext 0x80
	0x8fc2:ld.b %r0,[0x0]
	0x8fc4:and %r0,0x1
	0x8fc6:ret

I/O レジスタが、連続する 4 バイトに配置される場合、この I/O レジスタを 32 ビット整数としてアクセスできます。

ただし、S1C17 は 32 ビット整数の読み込みおよび書き込みを 2 つの'ld'命令として実行するので、S1C17 命令の数は減りません。

C ソースコード	S1C17 命令

typedef union {
  unsigned long dword;
  void * pointer;
  struct {
  short_fields low;
  short_fields high;
  } word;
} long_fields;
 	
#define device32 (*(volatile long_fields *)0x4000)	

void set_long_device(unsigned long value) {
  device32.dword = value;
}
	0x8fc8:ext 0x80
	0x8fca:ld [0x0],%r0
	0x8fcc:ext 0x80
	0x8fce:ld [0x2],%r1
	0x8fd0:ret

void set_low_high(unsigned int l, unsigned int h) {
  device32.word.low.word = l;
  device32.word.high.word = h;
}
	0x8fd2:ext 0x80
	0x8fd4:ld [0x0],%r0
	0x8fd6:ext 0x80
	0x8fd8:ld [0x2],%r1
	0x8fda:ret

"SMALL"モデルが選択されず、ポインタが 24 ビット整数値の場合、連続する 4 バイトに配置された I/O レジスタは、24 ビット値としてアクセスできます。この場合、これらの I/O レジスタは、32 ビット幅でアクセスできる必要があります。

C ソースコード	S1C17 命令

void set_24bit_pointer(void * pointer) {
  device32.pointer = pointer;
}
	0x91be:ext 0x80
	0x91c0:ld.a [0x0],%r0
	0x91c2:ret

この C ソースコードから GNU17 C コンパイラにより生成される S1C17 命令は、"SMALL"モデルでは期待したように動作しないので注意してください。

3.5.2 I/Oレジスタへの値の設定

メモリ上に保持される値からいくつかのビットを変更するために、GNU17 C コンパイラは、次のステップを生成します。

- メモリの値を S1C17 レジスタに読み込む。
- 読み込まれた値のいくつかのビットを変更する。
- 変更した値をメモリに書き込む。

I/O レジスタがビットフィールドとして定義される場合、GNU17 C コンパイラは、I/O レジスタを書き込むために同様の命令を生成します。

C ソースコード	S1C17 命令

void initialize_device_field(void) {
  device16.bit.a = 0;
  device16.bit.b = 1;
  device16.bit.c = 7;
  device16.bit.d = 2;
}
	0x91aa:ld.a -[%sp],%r4
	0x91ac:ext 0x80
	0x91ae:ld.a %r3,0x0
	0x91b0:ld.b %r2,[%r3]
	0x91b2:and %r2,0x7e
	0x91b4:ld.b [%r3],%r2
	0x91b6:ld.b %r2,[%r3]
	0x91b8:and %r2,0x1
	0x91ba:or %r2,0x2
	0x91bc:ld.b [%r3],%r2
	0x91be:ext 0x80
	0x91c0:ld.b %r2,[0x1]
	0x91c2:or %r2,0x7
	0x91c4:ext 0x80
	0x91c6:ld.b [0x1],%r2
	0x91c8:ext 0x80

	0x91ca:ld.b %r2,[0x1] 0x91cc:and %r2,0x7 0x91ce:or %r2,0x10 0x91d0:ext 0x80 0x91d2:ld.b [0x1],%r2 0x91d4:ld.a %r4,[%sp]+ 0x91d6:ret

ただし、すべてのビットを同時に上書きできるのであれば、GNU17 C コンパイラにより生成される S1C17 命令は減ります。

C ソースコード	S1C17 命令

void initialize_device_field(void) {
short_fields value;
 	
value.bit.a = 0;
value.bit.b = 1;
value.bit.c = 7;
value.bit.d = 2;
 	
device16.word = value.word;
}
	0x91aa:ext 0x2e
	0x91ac:ld %r2,0x2
	0x91ae:ext 0x80
	0x91b0:ld [0x0],%r2
	0x91b2:ret

次の場合、I/O レジスタのすべてのビットを同時に書き込むことができます。

- 16 ビット（または 8 ビット）I/O レジスタ中に存在する有効なビットフィールドは 1 つだけである。
- 初期化などのために I/O レジスタのすべてのビットを更新する。
- 変更するビットフィールド以外のビットの値が事前にわかっている。

3.5.3 I/Oレジスタからの値の取得

いくつかのビットフィールドが 16 ビット（または 8 ビット）I/O レジスタに含まれる場合、GNU17 C コンパイラは、各ビットフィールドを読み込むために、複数の'ld'命令を生成します。

C ソースコード	S1C17 命令

short check_device_flag(void) {
if ((device16.bit.a == 1) && (device16.bit.b == 3)) {
return 1;
}
return 0;
}
	0x91e2:ld.a -[%sp],%r4
	0x91e4:ext 0x80
	0x91e6:ld.a %r3,0x0
	0x91e8:ld.b %r2,[%r3]
	0x91ea:ld %r0,%r2
	0x91ec:and %r0,0x1
	0x91ee:cmp %r0,0x1
	0x91f0:jrne 0x5
	0x91f2:ld.b %r2,[%r3]
	0x91f4:ld.ub %r2,%r2
	0x91f6:sr %r2,0x1
	0x91f8:cmp %r2,0x3
	0x91fa:jreq 0x1
	0x91fc:ld %r0,0x0
	0x91fe:ld.a %r4,[%sp]+
	0x9200:ret

同じ I/O レジスタ中のいくつかのビットフィールドが同時に参照される場合、いくつかのビットをまとめて操作することで、GNU17 C コンパイラにより生成される S1C17 命令を減らすことができます。

C ソースコード	S1C17 命令

short check_device_flag(void) {
unsigned char flag;
 	
flag = device16.byte[0];
if (flag== ((3 << 1) | (1 << 0))) {
return 1;
}
return 0;
}
	0x91e2:ext 0x80
	0x91e4:ld.b %r2,[0x0]
	0x91e6:ld.ub %r2,%r2
	0x91e8:cmp %r2,0x7
	0x91ea:jreq.d 0x2
	0x91ec:ld %r0,0x1
	0x91ee:ld %r0,0x0
	0x91f0:ret

4. 関数設計

この項では、C 関数の引数および戻り値を選択して、S1C17 命令の数を減らす方法について説明します。詳細については、GNU17 マニュアルの”6.4.3 レジスタ使用方法”および”6.4.4 関数呼び出し”を参照してください。

4.1 引数

4.1.1 引数の数

GNU17 C コンパイラの呼び出し規則では、呼び出し側は、最初の 4 つの引数を、S1C17 レジスタの R0～R3 で呼び出される側の関数に渡し、他の引数をスタック領域で渡します。8 ビット、16 ビットおよび 24 ビット（レギュラおよびラージモデルのポインタ）の各引数は、S1C17 レジスタを 1 つ使用します。つまり、スタック領域を使用しない限り引数の最大数は 4 です。

C ソースコード	S1C17 命令

int argument_4_callee(int a, **int** b, **int** c, **int** d) {
　return a + b + c + d;
}
	0x8010:add　%r0,%r1
	0x8012:add　%r0,%r2
	0x8014:add　%r0,%r3
	0x8016:ret

int argument_4_caller(int a[]) {
　return argument_4_callee(a[0], a[1], a[2], a[3]);
}
	0x8018:ld　%r3,%r0
	0x801a:ld　%r0,[%r0]
	0x801c:ext　0x2
	0x801e:ld　%r1,[%r3]
	0x8020:ext　0x4
	0x8022:ld　%r2,[%r3]
	0x8024:ext　0x6
	0x8026:ld　%r3,[%r3]
	0x8028:call　0x3f3
	0x802a:ret

5 番目の引数はスタック領域に渡されるため、呼び出し側および呼び出される側の関数で必要なスタックフレームは大きくなり、スタックに置かれた値を取り出すための S1C17 命令が増えます。

C ソースコード	S1C17 命令

int argument_5_callee(int a, **int** b, **int** c, **int** d, **int** e) {
　return a + b + c + d + e;
}
	0x802c:add　%r0,%r1
	0x802e:add　%r0,%r2
	0x8030:add　%r0,%r3
	0x8032:ld　%r2,[%sp+0x4]
	0x8034:add　%r0,%r2
	0x8036:ret

int argument_5_caller(int a[]) {
　return argument_5_callee(a[0], a[1], a[2], a[3], a[4]);
}
	0x8038:sub.a　%sp,0x4
	0x803a:ld　%r3,%r0
	0x803c:ext　0x8
	0x803e:ld　%r2,[%r0]
	0x8040:ld　[%sp+0x0],%r2
	0x8042:ld　%r0,[%r0]
	0x8044:ext　0x2
	0x8046:ld　%r1,[%r3]
	0x8048:ext　0x4
	0x804a:ld　%r2,[%r3]
	0x804c:ext　0x6
	0x804e:ld　%r3,[%r3]
	0x8050:call　0x3ed
	0x8052:add.a　%sp,0x4
	0x8054:ret

4.1.2 引数の順序

GNU17 C コンパイラは、R0 レジスタを最初の引数に、R1 レジスタを 2 番目の引数に割り当て、これ以降も同様に割り当てを行います。R0 および R1 レジスタは、返された値の格納にも使用されます。ある関数が引数を別の関数に渡す場合、順序が一致しない引数の S1C17 レジスタを取り換えるため、必要な S1C17 命令は増えます。

C ソースコード	S1C17 命令

int argument_disordering(int a, **int** b, **int** c, **int** d) {
  return argument_4_callee(d, c, b, a);
}
	0x8056:ld.a -[%sp],%r4
	0x8058:ld.a -[%sp],%r5
	0x805a:ld %r5,%r0
	0x805c:ld %r4,%r1
	0x805e:ld %r0,%r3
	0x8060:ld %r1,%r2
	0x8062:ld %r2,%r4
	0x8064:call.d 0x3d5
	0x8066:ld %r3,%r5
	0x8068:ld.a %r5,[%sp]+
	0x806a:ld.a %r4,[%sp]+
	0x806c:ret

引数の順序が、呼び出し側と呼び出される側の関数で一致する場合、GNU17 C コンパイラで生成される S1C17 命令は減ります。

C ソースコード	S1C17 命令

int argument_ordering(int a, **int** b, **int** c, **int** d) {
  return argument_4_callee(a, b, c, d);
}
	0x806e: call 0x3d0
	0x8070: ret

4.1.3 パラメータセット

関数の各呼び出しで、GNU17 C コンパイラは、値を引数に設定する S1C17 命令を生成します。引数の数および引数の順序が複数の関数間で一致する場合でも、引数の S1C17 レジスタは再び設定されます。これは、呼び出し規則に従うと、呼び出される側の関数が R0～R3 レジスタの値を変更できるためです。

C ソースコード	S1C17 命令

int argument_4_callee(int a, **int** b, **int** c, **int** d) {
  return a + b + c + d;
}
	0x8010:add %r0,%r1
	0x8012:add %r0,%r2
	0x8014:add %r0,%r3
	0x8016:ret

int argument_with_parameters(int a, **int** b, **int** c) {
  int sum;
 	
  sum = argument_4_callee(a, b, c, 0);
  return argument_4_callee(a, b, c, sum);
}
	0x8074:ld.a -[%sp],%r4
	0x8076:ld.a -[%sp],%r5
	0x8078:ld.a -[%sp],%r6
	0x807a:ld %r4,%r0
	0x807c:ld %r5,%r1
	0x807e:ld %r3,0x0
	0x8080:call.d 0x3c7
	0x8082:ld %r6,%r2
	0x8084:ld %r3,%r0
	0x8086:ld %r0,%r4
	0x8088:ld %r1,%r5
	0x808a:call.d 0x3c2
	0x808c:ld %r2,%r6
	0x808e:ld.a %r6,[%sp]+
	0x8090:ld.a %r5,[%sp]+
	0x8092:ld.a %r4,[%sp]+
	0x8094:ret

関数呼び出しの負荷は、引数の数に従い大きくなります。

呼び出し側がパラメータセットを生成し、呼び出される側の関数にそのアドレスを渡す場合、S1C17 命令の数は減ります。パラメータセットを初期化して、そこからパラメータを取り出すための S1C17 命令が増えますが、関数を呼び出すための命令の数が少なくなります。

C ソースコード	S1C17 命令

int argument_set_callee(int a, **int** set[]) {
  return a + set[0] + set[1] + set[2];
}
	0x8096:ld %r2,[%r1]
	0x8098:add %r0,%r2
	0x809a:ext 0x2
	0x809c:ld %r2,[%r1]
	0x809e:add %r0,%r2
	0x80a0:ext 0x4
	0x80a2:ld %r2,[%r1]
	0x80a4:add %r0,%r2
	0x80a6:ret

```
int argument_with_set(int a, int b, int c) {
    int set[3] = {
        a, b, c,
    };
    int sum;

    sum = argument_set_callee(0, set);
    return argument_set_callee(sum, set);
}
```

```
0x80a8:sub.a   %sp,0x8
0x80aa:ld      [%sp+0x0],%r0
0x80ac:ld      [%sp+0x2],%r1
0x80ae:ld      [%sp+0x4],%r2
0x80b0:ld      %r0,0x0
0x80b2:ld.a    %r1,%sp
0x80b4:call    0x3f0
0x80b6:ld.a    %r1,%sp
0x80b8:call    0x3ee
0x80ba:add.a   %sp,0x8
0x80bc:ret
```

4.1.4 32 ビットおよびその他の大きな値

いずれかのパラメータが 32 ビット値の場合、スタック領域を使用していないときの引数の最大数は 4 未満です。これは、S1C17 レジスタのペアが、32 ビット（long または float）引数の格納に使用されるためです。

- 最初の引数が 32 ビット値の場合、R0 レジスタは下位 16 ビットで、R1 レジスタは上位 16 ビットです。
- 2 番目（または 3 番目）の引数が 32 ビット値の場合、R2 レジスタは下位 16 ビットで、R3 レジスタは上位 16 ビットです。

C ソースコード	S1C17 命令

`long argument_value32_callee(long a, long b) {`
`  return (long)((short)a + (short)b);`
`}`
	`0x80be:add  %r2,%r0`
	`0x80c0:ld  %r0,%r2`
	`0x80c2:cv.ls  %r1,%r0`
	`0x80c4:ret`

`long argument_value32_caller(void) {`
`  return argument_value32_callee(1L, 2L);`
`}`
	`0x80c6:ld  %r0,0x1`
	`0x80c8:sub  %r1,%r1`
	`0x80ca:ld  %r2,0x2`
	`0x80cc:sub  %r3,%r3`
	`0x80ce:call  0x3f7`
	`0x80d0:ret`

16 ビット値を渡すか、値自体ではなく 32 ビット値のアドレスを渡すことをお勧めします。

C ソースコード	S1C17 命令

`long argument_value16_callee(short a, short b) {`
`  return a + b;`
`}`
	`0x80d2:add  %r0,%r1`
	`0x80d4:ld  %r0,%r0`
	`0x80d6:cv.ls  %r1,%r0`
	`0x80d8:ret`

`long argument_value16_caller(void) {`
`  return argument_value16_callee(1, 2);`
`}`
	`0x80da:ld  %r0,0x1`
	`0x80dc:call.d  0x3fa`
	`0x80de:ld  %r1,0x2`
	`0x80e0:ret`

構造体のサイズが 64 ビット以下で、そのメンバを 4 つのレジスタに格納できる場合、GNU17 C コンパイラは、R0～R3 のレジスタを構造体に割り当て、値渡しで関数を呼び出します。

C ソースコード	S1C17 命令

`struct member4 {`
`  int r0;`
`  int r1;`
`  int r2;`
`  int r3;`
`};`	

`int member4_value_caller(void) {`
`  struct member4 m4 = {`
`  0, 1, 2, 3,`
`  };`
`  int r;`
 	
`  r = value_callee1(m4);`
`  return r + value_callee2(m4);`
`}`
	`0x80fa:ld.a  -[%sp],%r4`
	`0x80fc:sub.a  %sp,0x8`
	`0x80fe:ld.a  %r0,%sp`
	`0x8100:ext  0x103`
	`0x8102:ld  %r1,0x16`
	`0x8104:call.d  0x40  <memcpy>`
	`0x8106:ld  %r2,0x8`
	`0x8108:ld  %r0,[%sp+0x0]`
	`0x810a:ld  %r1,[%sp+0x2]`
	`0x810c:ld  %r2,[%sp+0x4]`

	0x810e:ld %r3,[%sp+0x6] 0x8110:call 0x3e8 0x8112:ld %r4,%r0 0x8114:ld %r0,[%sp+0x0] 0x8116:ld %r1,[%sp+0x2] 0x8118:ld %r2,[%sp+0x4] 0x811a:ld %r3,[%sp+0x6] 0x811c:call 0x3e8 0x811e:add %r4,%r0 0x8120:ld %r0,%r4 0x8122:add.a %sp,0x8 0x8124:ld.a %r4,[%sp]+ 0x8126:ret

通常、構造体が参照渡しにより、呼び出される側に渡される場合、S1C17 命令の数は減ります。

C ソースコード	S1C17 命令

int member4_reference_caller(void) {
static const struct member4 m4 = {
0, 1, 2, 3,
};
int r;
 	
r = reference_callee1(&m4);
return r + reference_callee2(&m4);
}
	0x8158:ld.a -[%sp],%r4
	0x815a:ext 0x103
	0x815c:ld %r0,0x1e
	0x815e:call 0x3e4
	0x8160:ld %r4,%r0
	0x8162:ext 0x103
	0x8164:ld %r0,0x1e
	0x8166:call 0x3ec
	0x8168:add %r4,%r0
	0x816a:ld %r0,%r4
	0x816c:ld.a %r4,[%sp]+
	0x816e:ret

64 ビット以上の構造体は、GNU17 C コンパイラでは常にスタック領域に格納されるため、参照渡しで渡される必要があります。
64 ビット（long long または double）型の引数も常にスタック領域に格納されます。これらについても、値渡しではなく、参照渡しをお勧めします。

4.2 戻り値

呼び出し規則に従い、GNU17 C コンパイラは、呼び出される側が R0 レジスタに値を返す S1C17 命令を生成します。

C ソースコード	S1C17 命令

int return_callee(void) {
return 0;
}
	0x8176:ld %r0,0x0
	0x8178:ret

int return_caller(int a) {
return a + return_callee();
}
	0x817a:call 0x3fd
	0x817c:add %r0,0x1
	0x817e:ret

4.2.1 大きな値

’long’などの 32 ビット値を返す場合、2 つの S1C17 レジスタ（R0 レジスタおよび R1 レジスタ）が使用されます。また、これより大きな値は常にスタック領域に返されます。

C ソースコード	S1C17 命令

struct big {
short a;
short b;
short c;
short d;
};	

struct big return_big_callee(void) {
struct big r;
 	
r.a = 1;
r.b = 2;
 	
return r;
	0x8180:ld.a -[%sp],%r4
	0x8182:sub.a %sp,0x8
	0x8184:ld %r4,%r0
	0x8186:ld %r2,0x1
	0x8188:ld [%sp+0x0],%r2
	0x818a:ld %r2,0x2
	0x818c:ld [%sp+0x2],%r2

}	0x818e:ld.a %r1,%sp 0x8190:call.d 0x1d <memcpy> 0x8192:ld %r2,0x8 0x8194:ld %r0,%r4 0x8196:add.a %sp,0x8 0x8198:ld.a %r4,[%sp]+ 0x819a:ret

long **return_long_callee**(**void**) {
    return 0L;
}
	0x819c:sub %r0,%r0
	0x819e:sub %r1,%r1
	0x81a0:ret

int **return_big_caller**(**void**) {
    struct big r;
 	
    r = return_big_callee();
    return r.a + r.b + (**short**)return_long_callee();
}
	0x81a2:ld.a -[%sp],%r4
	0x81a4:sub.a %sp,0x8
	0x81a6:ld.a %r0,%sp
	0x81a8:call 0x3eb
	0x81aa:ld %r4,[%sp+0x0]
	0x81ac:ld %r2,[%sp+0x2]
	0x81ae:call.d 0x3f6
	0x81b0:add %r4,%r2
	0x81b2:add %r4,%r0
	0x81b4:ld %r0,%r4
	0x81b6:add.a %sp,0x8
	0x81b8:ld.a %r4,[%sp]+
	0x81ba:ret

大きなオブジェクトは常に参照渡しで返されるため、呼び出される側の関数は、戻り値を格納するアドレスを引数として受け取ります。

C ソースコード	S1C17 命令

struct big * **return_pointer_callee**(**struct** big * r) {
    r->a = 1;
    r->b = 2;
 	
    return r;
}
	0x81bc:ld %r3,0x1
	0x81be:ld [%r0],%r3
	0x81c0:ld %r3,0x2
	0x81c2:ext 0x2
	0x81c4:ld [%r0],%r3
	0x81c6:ret

int **return_pointer_caller**(**void**) {
    struct big r;
 	
    (**void**)return_pointer_callee(&r);
    return r.a + r.b + (**short**)return_long_callee();
}
	0x81c8:ld.a -[%sp],%r4
	0x81ca:sub.a %sp,0x8
	0x81cc:ld.a %r0,%sp
	0x81ce:call 0x3f6
	0x81d0:ld %r4,[%sp+0x0]
	0x81d2:ld %r2,[%sp+0x2]
	0x81d4:call.d 0x3e3
	0x81d6:add %r4,%r2
	0x81d8:add %r4,%r0
	0x81da:ld %r0,%r4
	0x81dc:add.a %sp,0x8
	0x81de:ld.a %r4,[%sp]+
	0x81e0:ret

4.2.2 2 つの値

関数が返すオブジェクトは 1 つだけで、大きなオブジェクトは常にスタック領域に返されるため、複数のパラメータを返す場合、参照呼び出しが使用されます。

C ソースコード	S1C17 命令

int **return_error_callee**(**int** * out) {
    *out = 0;
    return 0;
}
	0x81e2: ld %r2,%r0
	0x81e4: ld %r0,0x0
	0x81e6: ld [%r2],%r0
	0x81e8: ret

int **return_error_caller**(**void**) {
    int r;
    int error;
 	
    error = return_error_callee(&r);
 	
    return error;
}
	0x81ea: sub.a %sp,0x4
	0x81ec: ld.a %r0,%sp
	0x81ee: call 0x3f9
	0x81f0: add.a %sp,0x4
	0x81f2: ret

構造体のサイズが 32 ビット以下で、そのメンバ数が 2 の場合、GNU17 C コンパイラは、R0～R1 のレジスタを割り当て、値渡しで構造体を返すことができます。

C ソースコード	S1C17 命令

union value32 {
  long v;
  struct value32_m {
  short e;
  short r;
  } m;
};	

union value32 return_value32_callee(void) {
  union value32 v;
 	
  v.m.e = 0;
  v.m.r = 0;
 	
  return v;
}
	0x81f4:sub %r0,%r0
	0x81f6:sub %r1,%r1
	0x81f8:ld %r1,%r0
	0x81fa:ret

int return_value32_caller(void) {
  union value32 v;
 	
  v = return_value32_callee();
 	
  return v.m.e;
}
	0x81fc:call 0x3fb
	0x81fe:ret

各値のビット幅が 16 未満の場合、16 ビット値により 2 つのパラメータを返すことができます。

C ソースコード	S1C17 命令

union value16 {
  short v;
  struct value16_m {
  char e;
  char r;
  } m;
};	

union value16 return_value16_callee(void) {
  union value16 v;
 	
  v.m.e = 0;
  v.m.r = 0;
 	
  return v;
}
	0x8200:ld %r0,0x0
	0x8202:ret

int return_value16_caller(void) {
  union value16 v;
 	
  v = return_value16_callee();
 	
  return v.m.e;
}
	0x8204:call 0x3fd
	0x8206:ld.b %r0,%r0
	0x8208:ret

4.2.3 次の関数のパラメータ

R0 レジスタは、戻り値および最初の引数の格納に使用されます。最初の引数として使用した変数が関数の戻り値により更新され、次の関数に最初の引数として渡される場合、GNU17 C コンパイラは S1C17 命令を最適化し、R0 レジスタを変数に割り当てます。

C ソースコード	S1C17 命令

int return_next_caller(int a) {
  int r = a;
 	
  r = return_next_callee1(r);
  r = return_next_callee2(r);
  r = return_next_callee3(r);
 	
  return r;
}
	0x8210:call 0x3fc
	0x8212:call 0x3fc
	0x8214:call 0x3fc
	0x8216:ret

4.3 複数の関数への分割

さまざまな処理が 1 つの関数で実装される場合、その関数の変数の数が増えます。スタックフレームは、関数の入口で作成され、実行中に拡張されることはなく、関数の出口で削除されます。そのため、変数の数が増えると、スタックフレームのサイズが大きくなります。

C ソースコード	S1C17 命令

int function_too_big_stack(void) {
  unsigned char a[16];
  unsigned char b[16];
  int r;
 	
  r = sub_function_a(a);
  r += sub_function_b(b);
 	
  return r;
}
	0x8178:ld.a -[%sp],%r4
	0x817a:sub.a %sp,0x20
	0x817c:ld.a %r0,%sp
	0x817e:call 0x3f8
	0x8180:ld %r4,%r0
	0x8182:ld.a %r0,%sp
	0x8184:call.d 0x3f7
	0x8186:add %r0,0x10
	0x8188:add %r4,%r0
	0x818a:ld %r0,%r4
	0x818c:add.a %sp,0x20
	0x818e:ld.a %r4,[%sp]+
	0x8190:ret

大きな関数を、スタックフレームのサイズがほぼ同じであるような複数の小さい関数に分割することで、プログラムで必要なスタック領域の容量を減らすことができます。

C ソースコード	S1C17 命令

int part_function_a(void) {
  unsigned char a[16];
 	
  return sub_function_a(a);
}
	0x8192:sub.a %sp,0x10
	0x8194:ld.a %r0,%sp
	0x8196:call 0x3ec
	0x8198:add.a %sp,0x10
	0x819a:ret

int part_function_b(void) {
  unsigned char b[16];
 	
  return sub_function_b(b);
}
	0x819c:sub.a %sp,0x10
	0x819e:ld.a %r0,%sp
	0x81a0:call 0x3e9
	0x81a2:add.a %sp,0x10
	0x81a4:ret

int function_divided_stack(void) {
  int r;
 	
  r = part_function_a();
  r += part_function_b();
 	
  return r;
}
	0x81a6:ld.a -[%sp],%r4
	0x81a8:call 0x3f4
	0x81aa:call.d 0x3f8
	0x81ac:ld %r4,%r0
	0x81ae:add %r4,%r0
	0x81b0:ld %r0,%r4
	0x81b2:ld.a %r4,[%sp]+
	0x81b4:ret

5. データ構造体設計

この項では、データ構造体およびデータそのものを定義することにより、データストレージを減らす方法について説明します。
詳細については、GNU17 マニュアルの”6.4.2 データ表現”を参照してください。

5.1 定数

5.1.1 記憶期間

GNU17 C コンパイラは、静的記憶期間をもつ定数を’.rodata’セクションに配置し、通常、リンカは、’.rodata’セクションを ROM に再配置します。
自動記憶期間をもつ定数は、RAM のスタックフレームに配置されます。これらの初期値を格納するためには、ROM の’.rodata’セクションが必要です。これらのタイプの定数は、初期値をもつ自動変数とおおよそ同様にして、通常、メモリを浪費します。静的記憶期間をもつよう設定するには、ローカル定数に’static’を指定する必要があります。

5.1.2 初期値をもつ変数

初期値をもつ変数には、RAM と ROM の両方が必要です。可能な場合、初期値をもつ変数ではなく、定数を使用することをお勧めします。

初期値をもつ複雑な変数（たとえば、配列型または構造型の変数）の一部を定数として定義できる場合、そのサイズを縮小できます。

C ソースコード

```
struct all_member_is_variable {
        int current_value;
        int initial_value;
        char message[16];
} all_variable[4] = {
        {0, 0, "1st Message"},
        {1, 1, "2nd Message"},
        {2, 2, "3rd Message"},
        {3, 3, "4th Message"},
};

const struct constant_part {
        int initial_value;
        char message[16];
} part_constant[4] = {
        {0, "1st Message"},
        {1, "2nd Message"},
        {2, "3rd Message"},
        {3, "4th Message"},
};

struct variable_part {
        int current_value;
        const struct constant_part * constants;
} part_variable[4] = {
        {0, part_constant + 0},
        {1, part_constant + 1},
        {2, part_constant + 2},
        {3, part_constant + 3},
};
```

GDB コンソール

```
(gdb) p sizeof(struct all_member_is_variable) * 4
$1 = 80
(gdb) p sizeof(struct constant_part) * 4
$2 = 72
(gdb) p sizeof(struct variable_part) * 4
$3 = 16
```

変数部分は、定数部分を示すために 1～4 バイト増加します。そのため、定数部分は 4 バイトを超えなければなりません。

定数部分を参照するために、S1C17 命令の数は増えます。

C ソースコード	S1C17 命令

char * message1(struct all_member_is_variable * v) {
  return v->message;
}
	0x81bc:add %r0,0x4
	0x81be:ret

const char * message2(struct variable_part * v) {
  return v->constants->message;
}
	0x81ce:ext 0x2
	0x81d0:ld %r0,[%r0]
	0x81d2:add %r0,0x2
	0x81d4:ret

ソースコード中で変数部分から定数部分を参照する個所が少ない場合、追加になる命令の影響はそれほどありません。

C ソースコード	S1C17 命令

char * message_of_index1(int i) {
  return all_variable[i].message;
}
	0x81c0:ld %r2,%r0
	0x81c2:sl %r0,0x2
	0x81c4:add %r0,%r2
	0x81c6:sl %r0,0x2
	0x81c8:ld %r3,0x4
	0x81ca:add %r0,%r3
	0x81cc:ret

const char * message_of_index2(int i) {
  return part_constant[i]. message;
}
	0x81d6:ld %r2,%r0
	0x81d8:sl %r0,0x3
	0x81da:add %r0,%r2
	0x81dc:sl %r0,0x1
	0x81de:ext 0x105
	0x81e0:ld %r3,0x3a
	0x81e2:add %r0,%r3
	0x81e4:ret

5.1.3 小さい整数値

S1C17 データ転送命令は、即値をとることができます。即値のサイズは通常 7 ビットで、24 ビットまで拡張できます。

C ソースコード	S1C17 命令

int imm0(void) {
  int r = 0;
 	
  return r;
}
	0x81be:ld %r0,0x0
	0x81c0:ret

void * imm0x876543 (void) {
  void * r = (void *)0x876543;
 	
  return r;
}
	0x81c2:ext 0x8
	0x81c4:ext 0xeca
	0x81c6:ld.a %r0,0x43
	0x81c8:ret

static const int internal_value = 0x1234;	

int const_internal_value(void) {
  return internal_value;
}
	0x81d0:ext 0x24
	0x81d2:ld %r0,0x34
	0x81d4:ret

小さい整数定数を定義する代わりに、命令に埋め込まれる即値を使用することで、使用される ROM 領域容量を減らすことができます。

C ソースコード	S1C17 命令
extern const int external_value;	

int const_external_value(void) {
  return external_value;
}
	0x81ca:ext 0x106
	0x81cc:ld %r0,[0x50]
	0x81ce:ret

GDB コンソール: (gdb) p sizeof(external_value) $1 = 2	

次に、即値に適した値を示します。GNU17 C コンパイラは、これらの値を S1C17 命令に埋め込みます。

- I/O レジスタのアドレス

- I/O レジスタ値と比較するビットパターン
- I/O レジスタ値に書き込むビットパターン
- テーブルの要素の数（配列のサイズ）
- 繰り返しの最大数
- カウンタおよびその他の一時変数の初期値
- ステータスコード

値の幅が 24 ビット未満の場合、整数定数を定義するよりも、即値を埋め込むことをお勧めします。ただし、即値が埋め込まれるプログラムの操作は、動的には変更できません。

5.2 構造体メンバの整列

5.2.1 メンバの順序

構造体のメンバは、宣言された順にメモリに配置されます。

C ソースコード

```
struct disordered_structure {
    char a_8bit;
    long b_32bit;
    char c_8bit;
    short d_16bit;
    short e_16bit;
    long f_32bit;
};

struct disordered_structure dv;
```

GDB コンソール

```
(gdb) p &(dv.a_8bit)
$1 = 0x0
(gdb) p &(dv.b_32bit)
$2 = (long int *) 0x4
(gdb) p &(dv.c_8bit)
$3 = 0x8 '¥252' <repeats 200 times>...
(gdb) p &(dv.d_16bit)
$4 = (short int *) 0xa
(gdb) p &(dv.e_16bit)
$5 = (short int *) 0xc
(gdb) p &(dv.f_32bit)
$6 = (long int *) 0x10
```

構造体の各メンバは、そのデータ型に基づいて独自の境界に割り当てられるため、メンバの順序により、一部のメモリが未使用になります。

アドレス	アドレス＋0	アドレス＋1	アドレス＋2	アドレス＋3
0x000000	dv.a_8bit	未使用	未使用	未使用
0x000004	dv.b_32bit			
0x000008	dv.c_8bit	未使用	dv.d_16bit	
0x00000c	dv.e_16bit		未使用	未使用
0x000010	dv.f_32bit			

配置方法は、同じセクションに属する変数または定数の順序と同じです。構造体がどのようなセクションに置かれる場合でも、これらのメンバは同じ方法で配置されます。
同じデータ型のメンバをまとめて定義することで、構造体の未使用メモリ量を減らすことができます。

C ソースコード

```
struct ordered_structure {
    char a_8bit;
    char c_8bit;
    short d_16bit;
    short e_16bit;
    long b_32bit;
    long f_32bit;
};

struct ordered_structure ov;
```

GDB コン

```
(gdb) p &(ov.a_8bit)
```

ソール

```
$7 = 0x14 '¥252' <repeats 200 times>...
(gdb) p &(ov.c_8bit)
$8 = 0x15 '¥252' <repeats 200 times>...
(gdb) p &(ov.d_16bit)
$9 = (short int *) 0x16
(gdb) p &(ov.e_16bit)
$10 = (short int *) 0x18
(gdb) p &(ov.b_32bit)
$11 = (long int *) 0x1c
(gdb) p &(ov.f_32bit)
$12 = (long int *) 0x20
```

アドレス	アドレス＋0	アドレス＋1	アドレス＋2	アドレス＋3
0x000014	ov.a_8bit	ov.c_8bit	ov.d_16bit	
0x000018	ov.e_16bit		未使用	未使用
0x00001c	ov.b_32bit			
0x000020	ov.f_32bit			

メンバを大きなデータ型から小さいデータ型の順に定義すると、未使用のメモリ量をさらに減らすことができます。これは、リンカが他のオブジェクトをそこへ再配置できることがあるためです。

5.2.2 複数の構造体への分割

整列のため、いくつかの名前のないパディングが構造体内に挿入されることがあります。パディングされた構造体が配列の要素である場合、未使用メモリ量は要素の数だけ増えます。

C ソースコード

```
struct ordered_structure {
    char a_8bit;
    char c_8bit;
    short d_16bit;
    short e_16bit;
    long b_32bit;
    long f_32bit;
};

struct ordered_structure array_all[16];
```

GDB コンソール

```
(gdb) p sizeof(array_all)
$1 = 256
```

パディングされた構造体を分割すると、パディングのない新しい構造体を宣言できます。新しい構造体の配列によるメモリ使用量は、元の構造体の配列で使用される量より小さくなります。

C ソースコード

```
struct ordered_structure_a {
    long b_32bit;
    long f_32bit;
};

struct ordered_structure_b {
    short d_16bit;
    short e_16bit;
    char a_8bit;
    char c_8bit;
};

struct ordered_structure_a array_a[16];
struct ordered_structure_b array_b[16];
```

GDB コンソール

```
(gdb) p sizeof(array_a)
$2 = 128
(gdb) p sizeof(array_b)
$3 = 96
(gdb) p &(array_a[0].b_32bit)
$4 = (long int *) 0x124
(gdb) p &(array_a[0].f_32bit)
$5 = (long int *) 0x128
(gdb) p &(array_a[1].b_32bit)
$6 = (long int *) 0x12c
(gdb) p &(array_a[1].f_32bit)
$7 = (long int *) 0x130
```

```
(gdb) p &(array_b[0].d_16bit)
$8 = (short int *) 0x1a4
(gdb) p &(array_b[0].e_16bit)
$9 = (short int *) 0x1a6
(gdb) p &(array_b[0].a_8bit)
$10 = 0x1a8 '¥252' <repeats 200 times>...
(gdb) p &(array_b[0].c_8bit)
$11 = 0x1a9 '¥252' <repeats 200 times>...
(gdb) p &(array_b[1].d_16bit)
$12 = (short int *) 0x1aa
(gdb) p &(array_b[1].e_16bit)
$13 = (short int *) 0x1ac
(gdb) p &(array_b[1].a_8bit)
$14 = 0x1ae '¥252' <repeats 200 times>...
(gdb) p &(array_b[1].c_8bit)
$15 = 0x1af '¥252' <repeats 200 times>...
```

アドレス	アドレス＋0	アドレス＋1	アドレス＋2	アドレス＋3
0x000124	array_a[0].b_32bit			
0x000128	array_a[0].f_32bit			
0x00012c	array_a[1].b_32bit			
0x000130	array_a[1].f_32bit			

アドレス	アドレス＋0	アドレス＋1	アドレス＋2	アドレス＋3
0x0001a4	array_b[0].d_16bit		array_b[0].e_16bit	
0x0001a8	array_b[0].a_8bit	array_b[0].c_8bit	array_b[1].d_16bit	
0x0001ac	array_b[1].e_16bit		array_b[1].a_8bit	array_b[1].c_8bit

ただし、複数の構造体を操作するために、S1C17 命令の数が増える場合があります。

C ソースコード	S1C17 命令

int member_sum_array_all(int index) {
  int r;
 	
  r = (**int**)array_all[index].b_32bit;
  r += array_all[index].d_16bit;
 	
  return r;
}
	0x81d2:sl %r0,0x4
	0x81d4:ld %r2,0x24
	0x81d6:add %r0,%r2
	0x81d8:ext 0x8
	0x81da:ld %r2,[%r0]
	0x81dc:ext 0x2
	0x81de:ld %r3,[%r0]
	0x81e0:add %r2,%r3
	0x81e2:ld %r0,%r2
	0x81e4:ret

int member_sum_partial_array(int index) {
  int r;
 	
  r = (**int**)array_a [index].b_32bit;
  r += array_b [index].d_16bit;
 	
  return r;
}
	0x81e6:ld %r2,%r0
	0x81e8:sl %r2,0x3
	0x81ea:ext 0x2
	0x81ec:ld %r3,0x24
	0x81ee:add %r2,%r3
	0x81f0:ld %r3,[%r2]
	0x81f2:ld %r2,%r0
	0x81f4:sl %r2,0x1
	0x81f6:add %r2,%r0
	0x81f8:sl %r2,0x1
	0x81fa:ext 0x3
	0x81fc:ld %r1,0x24
	0x81fe:add %r2,%r1
	0x8200:ld %r2,[%r2]
	0x8202:add %r3,%r2
	0x8204:ld %r0,%r3
	0x8206:ret

5.2.3 最初のメンバ

構造体の各メンバは、通常、構造体のアドレス（最下位アドレス）からのオフセットを介してアクセスされます。S1C17 レジスタが構造体のアドレスを保持する場合、S1C17 レジスタとメモリの構造体のメンバ同士でデータを転送するには、2 つの命令が必要です。

C ソースコード	S1C17 命令
short member_read_e(struct ordered_structure_b * p) {	0x820c:ext 0x2

`    return p->e_16bit;` `}`	`0x820e:ld      %r0,[%r0]` `0x8210:ret`

構造体のアドレスはその最初のメンバを指すため、レジスタと最初のメンバ間でデータを転送するための命令の数は 1 つです。

C ソースコード	S1C17 命令

`short member_read_d(struct ordered_structure_b * p) {`
`    return p->d_16bit;`
`}`
	`0x8208:ld      %r0,[%r0]`
	`0x820a:ret`

最も頻繁に参照されるメンバを構造体の最初のメンバに配置することで、S1C17 命令の数を減らすことができます。

5.3 静的データ定義

計算時間を短縮するために、C コンパイラにより生成された次の計算の結果が、定数として保持されます。

- 計算が複雑で実行に時間がかかる
- 計算が頻繁に実行される

計算が複雑な場合、通常、使用されるメモリ量は増えます。

C ソースコード	S1C17 命令

`short calc(short n) {`
`    short p;`
`    short r;`
 	
`    if (n < 5) {`
`        p = n * n;`
`        r = (((7 * 6 - p) * p - 7 * 6 * 5 * 4) * p * n);`
`        r = ((n * 7 * 6 * 5 * 4 * 3 * 2 * 1) + r) / (7 * 3 * 5 * 3);`
`} else {`
`        r = 0;`
`    }`
 	
`    return r;`
`}`
	`0x8170:ld.a    -[%sp],%r4`
	`0x8172:ld.a    -[%sp],%r5`
	`0x8174:cmp     %r0,0x4`
	`0x8176:jrgt.d  0x1c`
	`0x8178:ld      %r5,%r0`
	`0x817a:call.d  0x10b    <__mulhi3>`
	`0x817c:ld      %r1,%r0`
	`0x817e:ld      %r4,%r0`
	`0x8180:ld      %r0,0x2a`
	`0x8182:sub     %r0,%r4`
	`0x8184:call.d  0x106    <__mulhi3>`
	`0x8186:ld      %r1,%r4`
	`0x8188:ext     0x1f9`
	`0x818a:ld      %r1,0x38`
	`0x818c:add     %r0,%r1`
	`0x818e:call.d  0x101    <__mulhi3>`
	`0x8190:ld      %r1,%r4`
	`0x8192:call.d  0xff     <__mulhi3>`
	`0x8194:ld      %r1,%r5`
	`0x8196:ld      %r3,%r5`
	`0x8198:sl      %r3,0x2`
	`0x819a:add     %r3,%r5`
	`0x819c:ld      %r2,%r3`
	`0x819e:sl      %r2,0x4`
	`0x81a0:sl      %r2,0x2`
	`0x81a2:sub     %r2,%r3`
	`0x81a4:sl      %r2,0x4`
	`0x81a6:ext     0x2`
	`0x81a8:ld      %r1,0x3b`
	`0x81aa:call.d  0xce     <__divhi3>`
	`0x81ac:add     %r0,%r2`
	`0x81ae:jpr     0x1`
	`0x81b0:ld      %r0,0x0`
	`0x81b2:ld.a    %r5,[%sp]+`
	`0x81b4:ld.a    %r4,[%sp]+`
	`0x81b6:ret`

そのため、計算結果のサイズは、複雑な計算を実行する関数のサイズよりも小さくなります。

C ソースコード	S1C17 命令
`#define C1(n)    (((7*6-n*n)*n*n-7*6*5*4)*n*n*n)` `#define C2(n)    ((n*7*6*5*4*3*2*1)+C1(n))/(7*3*5*3)`	

```
static const short calc_result[5] = {
    C2(0), C2(1), C2(2), C2(3), C2(4),
};
```

```
0x8326:0x0000
0x8328:0x000d
0x832a:0x000e
0x832c:0x0001
0x832e:0xffea
```

```
short calc(short n) {
    return (n < 5) ? calc_result[n]: 0;
}
```

```
0x8170:cmp     %r0,0x4
0x8172:jrgt.d  0x7
0x8174:ld      %r2,%r0
0x8176:sl      %r2,0x1
0x8178:ext     0x106
0x817a:ld      %r3,0x26
0x817c:add     %r2,%r3
0x817e:ld      %r0,[%r2]
0x8180:jpr     0x1
0x8182:ld      %r0,0x0
0x8184:ret
```

浮動小数点および 64 ビット整数（”long long”）を含む計算では、簡単な場合であっても、GNU17 エミュレーションライブラリの関数が呼び出されます。エミュレーションライブラリのリンクされた関数により、メモリ使用量は増えます。

C ソースコード

```
float convert_to_float(int n) {
    return (float)n;
}
```

GNU17 リンカによりマッピングされるアドレス

```
convert_to_float.o(.text)
 .text   0x00008186    0x convert_to_float.o
         0x00008186       convert_to_float

libgcc.a(.text)
 .text   0x00008308   0x54 libgcc.a(_ashlsi3.o)
         0x00008308        __ashlsi3
 .text   0x0000835c   0x50 libgcc.a(lshrdi3.o)
         0x0000835c        __lshrsi3
 .text   0x000083ac   0x74 libgcc.a(addsf3.o)
         0x000083ac        __floatsisf
.text    0x00008420   0x7a libgcc.a(_c17_emulib_private_func_scan16.o)
         0x00008420        _c17_emulib_private_func_scan16
         0x0000844a        _c17_emulib_private_func_scan32
         0x00008468        _c17_emulib_private_func_scan64
```

これらの関数を呼び出す代わりに、計算された値をテーブルに格納することで、メモリの使用量を減らすことができます。

C ソースコード

```
static const float int_float[16] = {
    0.0, 1.0, 2.0, 3.0,
    4.0, 5.0, 6.0, 7.0,
    8.0, 9.0, 10.0, 11.0,
    12.0, 13.0, 14.0, 15.0,
};
```

S1C17 命令

```
0x833c:0x00000000  0x3f800000
0x8344:0x40000000  0x40400000
0x834c:0x40800000  0x40a00000
0x8354:0x40c00000  0x40e00000
0x835c:0x41000000  0x41100000
0x8364:0x41200000  0x41300000
0x836c:0x41400000  0x41500000
0x8374:0x41600000  0x41700000
```

```
float convert_to_float(int n) {
    union {
        float f;
        long l;
    } r;

    if (n < 16) {
        r.f = int_float[n];
} else {
        r.l = 0x7f900000;          /* NaN */
    }

    return r.f;
}
```

```
0x8170:cmp     %r0,0xf
0x8172:jrgt.d  0x9
0x8174:ld      %r2,%r0
0x8176:sl      %r2,0x2
0x8178:ext     0x106
0x817a:ld      %r3,0x3c
0x817c:add     %r2,%r3
0x817e:ld      %r0,[%r2]
0x8180:ext     0x2
0x8182:ld      %r1,[%r2]
0x8184:jpr     0x3
0x8186:sub     %r0,%r0
0x8188:ext     0xff
0x818a:ld      %r1,0x10
0x818c:ret
```

5.4 動的データ生成

5.4.1 要求時の計算

プログラムは、実行中に動的に値を計算できます。また、プログラムは、コンパイラにより計算された値を定数として保持できます。

C ソースコード

```
const unsigned long calculated_exponent2[64] = {
    0 * 0, 1 * 1, 2 * 2, 3 * 3,
    4 * 4, 5 * 5, 6 * 6, 7 * 7,
    8 * 8, 9 * 9, 10 * 10, 11 * 11,
    12 * 12, 13 * 13, 14 * 14, 15 * 15,
    …
};

unsigned long exponent2(unsigned long n) {
    return (n < 32) ? calculated_exponent2[n]:ULONG_MAX;
}
```

GNU17 リンカによりマッピングされるアドレス

```
exponent2.o(.text)
 .text   0x000081d4   0x24 exponent2.o
         0x000081d4        exponent2

 exponent2.o(.rodata)
 .rodata 0x000083a0  0x100 exponent2.o
         0x000083a0        calculated_exponent2
```

静的に格納されたデータが大きすぎる場合、必要に応じてデータを生成することで、メモリの使用量を減らすことができます。

C ソースコード

```
unsigned long exponent2(unsigned long n) {
    return n * n;
}
```

GNU17 リンカによりマッピングされるアドレス

```
dynamic.o(.text)
 .text   0x000081d4    0x8 dynamic.o
         0x000081d4        exponent2

 libgcc.a(.text)
 .text   0x00008326   0x64 libgcc.a(_mulhi3.o)
         0x00008326        __mulhi3
 .text   0x0000838a   0x40 libgcc.a(_mulsi3.o)
         0x0000838a        __mulsi3
```

5.4.2 圧縮と伸長

通常、データの表現は冗長です。大きな冗長データを圧縮することで、メモリの使用量を減らすことができます。

C ソースコード

```
extern const struct constant_part *part_constant;

struct variable_part2 {
    int current_value[4];
    char index[4];
} part_variable2 = {
    {0, 1, 2, 3},
    {0, 1, 2, 3},
};
```

GDB コンソール

```
(gdb) p sizeof(struct constant_part) * 4
$1 = 72
(gdb) p sizeof(struct variable_part) * 4
$2 = 16
(gdb) p sizeof(struct variable_part2)
$3 = 12
```

圧縮および伸長のプロセスが簡単な場合、メモリの使用量は小さくなります。

C ソースコード	S1C17 命令
`const char * message_of_index3(int i) {`	`0x81f8:ext  0x4`

```
    int index = part_variable2.constant_index[i];
    return part_constant[index].message;
}
```

```
0x81fa:ld      %r2,0xc
0x81fc:add     %r0,%r2
0x81fe:ld.b    %r2,[%r0]
0x8200:ld      %r0,%r2
0x8202:sl      %r0,0x3
0x8204:add     %r0,%r2
0x8206:sl      %r0,0x1
0x8208:ext     0x109
0x820a:ld      %r2,[0x3a]
0x820c:add     %r0,%r2
0x820e:add     %r0,0x2
0x8210:ret
```

メモリの使用量が増え、GNU17 の”MIDDLE”モデルがプロジェクトに選択される場合、S1C17 ポインタ型のサイズは 32 ビットになります。
ターゲット RAM のサイズが 64KB 未満で、変数が常に RAM 上のアドレスを示す場合、変数の実データは 16 ビット以下になります。この場合、上位 16 ビットを切り捨てることで、32 ビットアドレスを 16 ビットデータに圧縮できます。16 ビットデータを伸長するには、この逆の処理を行います。

5.5 メモリ共有

5.5.1 ヒープ領域

GNU17 ANSI ライブラリ’libc.a’は、ヒープ関数’malloc’、’calloc’、’realloc’および’free’をサポートします。GNU17 のヒープ関数の使用方法については、GNU17 マニュアルの”7.3.3 グローバル変数の宣言と初期化”を参照してください。
小さなヒープ領域はすぐに使い切られるため、スモールメモリプログラムでヒープ領域を使用する効果はあまり大きくありません。

5.5.2 スタック領域

同じ関数から呼び出される関数は、それぞれのスタックフレームを同じアドレス（スタックポインタの値）から割り当てます。これらの関数は、同じスタック領域を共有します。これらの関数の’auto’変数（自動記憶期間をもつ変数）も異なる時間に同じアドレスを共有します。
そのため、関数で使用される変数に自動記憶期間を指定することで、RAM 領域の使用量を減らすことができます。関数で記憶域クラス指定子’static’なしで宣言される変数は、自動記憶期間をもつ変数です。

C ソースコード

```
void * stack_share_callee1(int n) {
    char test[16];
    void * r;

    memset(test, n, sizeof(test));
    r = (void *)test;

    return r;
}

void * stack_share_callee2(int n) {
    char test[16];
    void * r;

    memset(test, n, sizeof(test));
    r = (void *)test;

    return r;
}

void * stack_share_callee3(int n) {
    char test[16];
    void * r;

    memset(test, n, sizeof(test));
    r = (void *)test;

    return r;
```

```
}

void * stack_share_caller() {
    void * stack;
    void * r;

    r = stack = stack_share_callee1(1);
    stack = stack_share_callee2(2);
    if (stack < r) {
        r = stack;
    }
    stack = stack_share_callee3(3);
    if (stack < r) {
        r = stack;
    }

    return r;
}
```

GNU17 リンカによりマッピングされるアドレス

```
stack_share.o(.text)
 .text   0x0000833c 0x52 stack_share.o
        0x0000833c      stack_share_callee1
        0x0000834c      stack_share_callee2
        0x0000835c      stack_share_callee3
        0x0000836c      stack_share_caller
```

GDB コンソール

```
(gdb) set $pc = stack_share_caller
(gdb) set $sp = 0x400
(gdb) next 4
(gdb) p r
$1 = (void *) 0x3e8
(gdb) p $sp - (int)r
$2 = 20
(gdb) x/20xb r
0x3e8:   0x03     0x03     0x03     0x03     0x03     0x03     0x03     0x03
0x3f0:   0x03     0x03     0x03     0x03     0x03     0x03     0x03     0x03
0x3f8:   0x82     0x83     0x00     0x00
```

これが効果を発揮するには、ほとんどの関数が同じスタック領域を使用することが必要です。そのため、一部の関数が大きなスタックフレームを必要とする場合、スタック領域が十分に共有されていないので、その効果は小さくなります。

C ソースコード

```
void * stack_share_callee2(int n) {
    char test[32];      //'test'のサイズが変化
    void * r;

    memset(test, n, sizeof(test));
    r = (void *)test;

    return r;
}
```

GDB コンソール

```
(gdb) set $pc = stack_share_caller
(gdb) set $sp = 0x400
(gdb) next 7
(gdb) p r
$1 = (void *) 0x3d8
(gdb) p $sp - (int)r
$2 = 36
(gdb) x/36xb r
0x3d8:   0x02     0x02     0x02     0x02     0x02     0x02     0x02     0x02
0x3e0:   0x02     0x02     0x02     0x02     0x66     0x83     0x00     0x00
0x3e8:   0x03     0x03     0x03     0x03     0x03     0x03     0x03     0x03
0x3f0:   0x03     0x03     0x03     0x03     0x03     0x03     0x03     0x03
0x3f8:   0x82     0x83     0x00     0x00
```

5.5.3 Union

同時には使用されない複数のメンバを含む union 型を定義することで、1 つのメモリ領域を複数の変数として使用できます。

C ソースコード

```
short mode_a_variables[8];
short mode_b_variables[16];
```

GNU17 リンカによりマッピングされるアドレス

```
mode_variables.o(.bss)
 .bss    0x00000204 0x30 mode_variables.o
         0x00000204      mode_a_variables
         0x00000214      mode_b_variables
```

プログラムの状態に従い、グローバル変数の一部または構造体のいくつかのメンバが使用される場合、union 型を定義してください。

C ソースコード

```
struct global_parameter {
    short current_mode;
    union mode_variable {
        short a_variables[8];
        short b_variables[16];
    } mode;
} global_parameter;
```

GNU17 リンカによりマッピングされるアドレス

```
mode_variables.o(.bss)
 .bss    0x00000204 0x22 mode_variables.o
         0x00000204      global_parameter
```

6. 手順やデータの結合

この項では、共通する手順に対して関数を利用することで S1C17 命令の数を減らす方法について、および共通する値を共有することでデータストレージを減らす方法について説明します。

6.1 標準ライブラリ関数の使用

GNU17 は、ANSI C 標準ライブラリ'libc.a'を提供します。このライブラリは、Epson により作成されており、GPL の対象ではありません。ライブラリ関数の詳細については、GNU17 マニュアルの"7.3 ANSI ライブラリ"を参照してください。

'string.h'で宣言される標準文字関数（'memcmp'、'memcpy'など）は、メモリ中のストリームデータの操作に役立ちます。GNU17 の文字関数のソースコードは、S1C17 命令の数を減らすために、S1C17 アセンブラにより書かれています。

C ソースコード	S1C17 命令

short clear_and_call (**void**) {
　short work[16];
　int i;
 	
　for (i = 0; i < 16; i++) {
　　work[i] = 0;
　}
 	
　return output_some_value(work);
}
	0x8192:sub.a　%sp,0x20
	0x8194:ld　%r0,0x0
	0x8196:ld　%r2,%r0
	0x8198:sl　%r2,0x1
	0x819a:ld.a　%r3,%sp
	0x819c:add　%r2,%r3
	0x819e:ld　%r3,0x0
	0x81a0:add　%r0,0x1
	0x81a2:cmp　%r0,0xf
	0x81a4:jrle.d　0x78
	0x81a6:ld　[%r2],%r3
	0x81a8:ld.a　%r0,%sp
	0x81aa:call　0x3f1
	0x81ac:add.a　%sp,0x20
	0x81ae:ret

C ソースコード	S1C17 命令

short clear_and_call (**void**) {
　short work[16];
 	
　memset(work, 0, **sizeof**(work));
 	
　return output_some_value(work);
}
	0x8192:sub.a　%sp,0x20
	0x8194:ld.a　%r0,%sp
	0x8196:ld　%r1,0x0
	0x8198:call.d　0x115　<memset>
	0x819a:ld　%r2,0x20
	0x819c:ld.a　%r0,%sp
	0x819e:call　0x3f7
	0x81a0:add.a　%sp,0x20
	0x81a2:ret

GNU17 ライブラリのすべてのソースコードは、GNU17 の'utility/lib_src'フォルダに含まれています。

GNU17 C コンパイラは、配列および構造体をコピーする C ソースコードから、'memcpy'関数を呼び出す S1C17 命令を生成します。

C ソースコード	S1C17 命令

struct copy_struct {
　short buffer[16];
};	

short copy_and_call(**struct** copy_struct * value) {
　struct copy_struct work;
 	
　work = *value;
 	
　return output_some_value(work.buffer);
}
	0x81b0:sub.a　%sp,0x20
	0x81b2:ld　%r1,%r0
	0x81b4:ld.a　%r0,%sp
	0x81b6:call.d　0x10d　<memcpy>
	0x81b8:ld　%r2,0x20
	0x81ba:ld.a　%r0,%sp
	0x81bc:call　0x3e8
	0x81be:add.a　%sp,0x20
	0x81c0:ret

'memcpy'関数が GNU17 C コンパイラによりすでに呼び出されている場合、新たな C ソースコードが'memcpy'を呼び出しても、メモリの使用量は増えません。

6.2 同じ手順に対する新しい関数の定義

同じ手順に対して新しい関数を定義すると、メモリの使用量を減らすことができます。

C ソースコード	S1C17 命令

short partly_same_procedure_1(**int** n) {
  int i;
  short buffer[8];
 	
  if (8 < n) {
  n = 8;
  }
  for (i = 0; i < n; i++) {
  buffer[i] = i;
  }
 	
  return similar_function_1(buffer, n);
}
	0x81dc:sub.a %sp,0x10
	0x81de:ld %r2,0x8
	0x81e0:cmp %r0,%r2
	0x81e2:jrle.d 0x2
	0x81e4:ld %r1,%r0
	0x81e6:ld %r1,%r2
	0x81e8:ld %r3,0x0
	0x81ea:cmp %r3,%r1
	0x81ec:jrge.d 0x9
	0x81ee:ld %r2,%r3
	0x81f0:sl %r2,0x1
	0x81f2:ld.a %r0,%sp
	0x81f4:add %r2,%r0
	0x81f6:ld [%r2],%r3
	0x81f8:add %r3,0x1
	0x81fa:cmp %r3,%r1
	0x81fc:jrlt.d 0x79
	0x81fe:ld %r2,%r3
	0x8200:ld.a %r0,%sp
	0x8202:call 0x3e8
	0x8204:add.a %sp,0x10
	0x8206:ret

short partly_same_procedure_2(**int** n) {
  int i;
  short buffer[4];
 	
  if (4 < n) {
  n = 4;
  }
  for (i = 0; i < n; i++) {
  buffer[i] = i;
  }
 	
  return similar_function_2(buffer, n);
}
	0x8208:sub.a %sp,0x8
	0x820a:cmp %r0,0x4
	0x820c:jrle.d 0x2
	0x820e:ld %r1,%r0
	0x8210:ld %r1,0x4
	0x8212:ld %r3,0x0
	0x8214:cmp %r3,%r1
	0x8216:jrge.d 0x9
	0x8218:ld %r2,%r3
	0x821a:sl %r2,0x1
	0x821c:ld.a %r0,%sp
	0x821e:add %r2,%r0
	0x8220:ld [%r2],%r3
	0x8222:add %r3,0x1
	0x8224:cmp %r3,%r1
	0x8226:jrlt.d 0x79
	0x8228:ld %r2,%r3
	0x822a:ld.a %r0,%sp
	0x822c:call 0x3d5
	0x822e:add.a %sp,0x8
	0x8230:ret

ただし、1.2.1 項”関数”で説明されているように、GNU17 C コンパイラは、新しい関数を呼び出し、開始、終了する手順を追加します。

C ソースコード	S1C17 命令

short partly_same_procedure_1(**int** n) {
  short buffer[8];
 	
  if (8 < n) {
  n = 8;
  }
  same_procedure(buffer, n);
 	
  return similar_function_1(buffer, n);
}
	0x81dc:ld.a -[%sp],%r4
	0x81de:sub.a %sp,0x10
	0x81e0:ld %r2,0x8
	0x81e2:cmp %r0,%r2
	0x81e4:jrle.d 0x2
	0x81e6:ld %r4,%r0
	0x81e8:ld %r4,%r2
	0x81ea:ld.a %r0,%sp
	0x81ec:call.d 0x16
	0x81ee:ld %r1,%r4
	0x81f0:ld.a %r0,%sp
	0x81f2:call.d 0x3f0
	0x81f4:ld %r1,%r4
	0x81f6:add.a %sp,0x10
	0x81f8:ld.a %r4,[%sp]+
	0x81fa:ret

short partly_same_procedure_2(**int** n) { **short** buffer[4];	0x81fc:ld.a -[%sp],%r4 0x81fe:sub.a %sp,0x8

`    if (4 < n) {`
`        n = 4;`
`    }`
`    same_procedure(buffer, n);`
 	
`    return similar_function_2(buffer, n);`
`}`
	`0x8200:cmp    %r0,0x4`
	`0x8202:jrle.d  0x2`
	`0x8204:ld     %r4,%r0`
	`0x8206:ld     %r4,0x4`
	`0x8208:ld.a   %r0,%sp`
	`0x820a:call.d  0x7`
	`0x820c:ld     %r1,%r4`
	`0x820e:ld.a   %r0,%sp`
	`0x8210:call.d  0x3e3`
	`0x8212:ld     %r1,%r4`
	`0x8214:add.a   %sp,0x8`
	`0x8216:ld.a   %r4,[%sp]+`
	`0x8218:ret`

`void same_procedure(short * buffer, int n) {`
`    int i;`
 	
`    for (i = 0; i < n; i++) {`
`        buffer[i] = i;`
`    }`
`}`
	`0x821a:ld.a   -[%sp],%r4`
	`0x821c:ld     %r3,0x0`
	`0x821e:cmp    %r3,%r1`
	`0x8220:jrge.d  0x8`
	`0x8222:ld     %r2,%r3`
	`0x8224:sl     %r2,0x1`
	`0x8226:add    %r2,%r0`
	`0x8228:ld     [%r2],%r3`
	`0x822a:add    %r3,0x1`
	`0x822c:cmp    %r3,%r1`
	`0x822e:jrlt.d  0x7a`
	`0x8230:ld     %r2,%r3`
	`0x8232:ld.a   %r4,[%sp]+`
	`0x8234:ret`

プログラムのサイズを実際に削減するには、次の関係が成立する必要があります。

A * N > A + B + C * N
- A：その手順における S1C17 命令の数
- N：手順が実行される場所の数
- B：新しい関数中に追加される S1C17 命令の数
- C：引数を準備し、関数を呼び出すために追加される S1C17 命令の数

ここでは、B および C は引数の数により異なるため、次の関係を前提とします。

B = b1 * NA + b2
- b1 * NA：　関数を開始するための S1C17 命令の数
- b2：　関数を終了するための S1C17 命令の数
- NA：　関数の引数の数

C = c1 * NA + c2
- c1 * NA：　引数を準備するための S1C17 命令の数
- c2：　関数を呼び出すための S1C17 命令の数

GNU17 において、b1、b2、c1 および c2 は小さな値です。どれも 3 命令と仮定するなら、プログラムのサイズを実際に削減するには、次の関係が成立する必要があります。

A > 3 * (NA + 1) * (N + 1) / (N - 1)
- N <= 1：　プログラムのサイズを削減することはできません。
- N == 2：　A > 9 * (NA + 1)
- N == 3：　A > 6 * (NA + 1)
- N == 4：　A > 5 * (NA + 1)
- N は十分大きい：　A > 3 * (NA + 1)

手順に対する S1C17 命令の数が少ない場合、その手順はマクロ関数として定義するべきです。

6.3 類似する機能の結合

手順が似ているものの同じではない場合、変更なしに新しい関数を定義することはできません。共通部分のサイズが相違部分より大きい場合、手順が以前より複雑になっても、プログラム全体のサイズが小さくなることがあります。このような場合に適用できる設計方法はいくつかあります。

6.3.1 共通する関数の定義

共通部分を実行するための新しい関数を定義します。新しい関数を呼び出すように、既存の手順を変更します。

C ソースコード

```
char p0_get(void) {
	static const char convert[8] = {
		0, 0, 0, 1, 0, 1, 1, 1,
	};
	int bit, byte, sample, length;

	t16.ctl = 0x0103;
	length = 8;
	byte = 0;

	do {
		bit = 0;
		sample = 3;
		do {
			while (t16.intf == 0);
			t16.intf = 0x0001;
			bit <<= 1;
			bit |= p0.dat & 1;
		} while (--sample > 0);

		byte <<= 1;
		byte |= convert[bit];
	} while (--length > 0);

	t16.ctl = 0x0000;
			t16.intf = 0x0001;

	return (char)byte;
}
```

C ソースコード

```
char p1_get(void) {
	static const char convert[8] = {
		0, 0, 0, 1, 0, 1, 1, 1,
	};
	int bit, byte, sample, length;

	t16.ctl = 0x0103;
	length = 8;
	byte = 0;

	do {
		bit = 0;
		sample = 3;
		do {
			while (t16.intf == 0);
			t16.intf = 0x0001;
			bit <<= 1;
			bit |= p1.dat & 1;
		} while (--sample > 0);

		byte <<= 1;
		byte |= convert[bit];
	} while (--length > 0);

	t16.ctl = 0x0000;
			t16.intf = 0x0001;

	return (char)byte;
}
```

相違部分の前後の部分が、新しい関数として定義されます。

C ソースコード

```
void t16_wait(void) {
    while (t16.intf == 0);
            t16.intf = 0x0001;
}
```

C ソースコード

```
void t16_stop(void) {
    t16.ctl = 0x0000;
            t16.intf = 0x0001;
}
```

C ソースコード

```
int convert_bit(int byte, int bit) {
    static const char convert[8] = {
        0, 0, 0, 1, 0, 1, 1, 1,
    };
    byte <<= 1;
    byte |= convert[bit];

    return byte;
}
```

C ソースコード

```
char p0_get(void) {
    int bit, byte, sample, length;

    t16.ctl = 0x0103;
    length = 8;
    byte = 0;

    do {
        bit = 0;
        sample = 3;
        do {
            t16_wait();
            bit <<= 1;
            bit |= p0.dat & 1;
        } while (--sample > 0);
        byte = convert_bit(byte, bit);
    } while (--length > 0);

    t16_stop();

    return (char)byte;
}
```

```
char p1_get(void) {
    int bit, byte, sample, length;

    t16.ctl = 0x0103;
    length = 8;
    byte = 0;

    do {
        bit = 0;
        sample = 3;
        do {
            t16_wait();
            bit <<= 1;
            bit |= p1.dat & 1;
        } while (--sample > 0);
        byte = convert_bit(byte, bit);
    } while (--length > 0);

    t16_stop();

    return (char)byte;
}
```

この例では、関数の数は 3 つ増えます。

6.3.2 相違部分を選択する新しいパラメータの定義

条件分岐により相違部分を実行する新しい関数を定義します。関数のパラメータは、どの部分を実行するのかを指定します。新しい関数を呼び出すように、類似した手順を変更します。

C ソースコード

```
char port_get(int channel) {
    static const char convert[8] = {
        0, 0, 0, 1, 0, 1, 1, 1,
    };
    int bit, byte, sample, length;

    t16.ctl = 0x0103;
    length = 8;
    byte = 0;

    do {
        bit = 0;
        sample = 3;
        do {
            while (t16.intf == 0);
            t16.intf = 0x0001;
            bit <<= 1;
            switch (channel) {
            case 1:
            bit |= p1.dat & 1;
                break;
            case 0:
            default:
            bit |= p0.dat & 1;
                break;
            }
        } while (--sample > 0);

        byte <<= 1;
        byte |= convert[bit];
    } while (--length > 0);

    t16.ctl = 0x0000;
            t16.intf = 0x0001;

    return (char)byte;
}
```

C ソースコード

```
char p0_get(void) {
    return port_get(0);
}
```

C ソースコード

```
char p1_get(void) {
    return port_get(1);
}
```

この例では、関数の数は 1 つ増えます。

6.3.3 異なる関数の定義

各相違部分を実行する新しい関数と、それとは別に共通部分を実行する新しい関数を定義します。共通部分の関数のパラメータで実行するべき相違部分を指定します。共通部分の関数を呼び出すように、類似した手順を変更します。

C ソースコード

```
char port_get(int (*get_dat)(void)) {
    static const char convert[8] = {
        0, 0, 0, 1, 0, 1, 1, 1,
    };
    int bit, byte, sample, length;

    t16.ctl = 0x0103;
    length = 8;
    byte = 0;

    do {
        bit = 0;
        sample = 3;
        do {
            while (t16.intf == 0);
            t16.intf = 0x0001;
            bit <<= 1;
            bit |= get_dat();
        } while (--sample > 0);
```

C ソースコード

```
        byte <<= 1;
        byte |= convert[bit];
    } while (--length > 0);

    t16.ctl = 0x0000;
        t16.intf = 0x0001;

    return (char)byte;
}
```

C ソースコード

```
int p0_dat(void) {
    return p0.dat & 1;
}
```

C ソースコード

```
int p1_dat(void) {
    return p1.dat & 1;
}
```

C ソースコード

```
char p0_get(void) {
    return port_get(p0_dat);
}
```

C ソースコード

```
char p1_get(void) {
    return port_get(p1_dat);
}
```

この例では、関数の数は 3 つ増えます。
相違部分の関数を呼び出す場所が少ないため、この方法では、他の方法と比較して、メモリの使用量は少なくなりません。

6.4 同じ値の共有

変数だけでなく定数であっても、値が一致するか否かに関わらず別のオブジェクトとなり、他のアドレスに配置されます。

C ソースコード

```
const int zero = 0;
const int initial = 0;
const char test_message[] = "TEST:";
const char work_message[] = "TEST:";
```

GNU17 リンカによりマッピングされるアドレス

```
samevalues.o(.rodata)
 .rodata 0x00008704 0x10 samevalues.o
        0x00008704      zero
        0x00008706      initial
        0x00008708      test_message
        0x0000870e      work_message
```

GDB コンソール

```
(gdb) p zero
$1 = 0
(gdb) p initial
$2 = 0
(gdb) p test_message
$3 = "TEST:"
(gdb) p work_message
$4 = "TEST:"
```

複数の定数が、同じ C ソースコードにあって、同じ文字列リテラルへのポインタである場合、これらは同じアドレスを指します。これらの定数が異なる C ソースコードにある場合、異なるアドレスを指します。

C ソースコード

```
const char * const test_message = "TEST:";
const char * const work_message = "TEST:";
```

C ソースコード

```
const char * const external_message = "TEST:";
```

GNU17 リンカによりマッピングされるアドレス

```
externvalues.o(.rodata)
 .rodata 0x000086b8  0x8 externvalues.o
        0x000086be      external_message
 samevalues.o(.rodata)
 .rodata 0x0000870c  0xa samevalues.o
        0x00008712      test_message
        0x00008714      work_message
```

GDB コンソール

```
(gdb) p test_message
$1 = 0x870c "TEST:"
```

```
(gdb) p work_message
$2 = 0x870c "TEST:"
(gdb) p external_message
$3 = 0x86b8 "TEST:"
```

文字列リテラルが C ヘッダファイルでマクロとして定義される場合、このマクロを使用する各 C ソースコード中で同じ値の文字列リテラルが生成されます。
定数を特定の C ソースコード中で生成し、そのアドレスを他のソースコードが参照する場合、メモリの使用量を減らすことができます。

C ソースコード

```
const char * error_message(short code) {
    static const struct error_message_t {
        short code;
        const char * string;
    } error_messages[] = {
        {0x0001, "Failed."},
        {0x0000, "Succeeded."},
        {-1, "Unknown Error."},
    };
    const struct error_message_t * message;

    message = error_messages;
    while (message->code != code) {
        if (message->code == -1) {
            break;
        }
        message++;
    }

    return message->string;
}
```

7. 未使用な手順およびデータの削除

この項では、未使用の関数、変数および定数を検出し、プログラムの未使用部分を削除する方法について説明します。

7.1 Cソースコードの分析

7.1.1 GNU17 Cコンパイラ

'-Wall'が GNU17 C コンパイラのコマンドラインオプションに指定されている場合、コンパイラは、静的記憶期間をもつ未使用の関数および変数、また自動記憶期間をもつ未使用の変数に対して警告を出力します。
GNU17 C コンパイラのコマンドラインオプションの詳細については、GNU17 マニュアルの"6.3.2 コマンドラインオプション"を参照してください。

GNU17 IDE は、通常、C コンパイラに'-Wall'を指定して C ソースコードからオブジェクトを生成します。そのため、GNU17 IDE でプログラムをビルドする場合、コンパイラが未使用の関数および変数に対して警告を出力します。
警告は、GNU17 IDE のコンソールビューおよび問題ビューに表示されます。

C ソースコード
```
static void unused_internal_function(void) {
    return;
}
```
IDE のコンソールビュー
```
unused_function.c:8:warning:`unused_internal_function' defined but not used
```

警告された関数および変数も、ビルドされたプログラムのメモリ中に存在します。警告された関数および変数を削除することで、メモリの使用量を減らすことができます。
警告されたオブジェクトを削除し、プログラムを再びビルドした後に、C コンパイラがあらためて他の関数および変数が未使用であると警告する場合があります。新しく警告された未使用オブジェクトも同様に削除してください。

ただし、GNU17 C コンパイラは、静的記憶期間をもつ未使用の定数については警告しません。

7.1.2 GNU17 IDEの機能

GNU17 C コンパイラは、外部結合として定義される未使用の関数および変数については警告しません。記憶クラス指定子をもたない関数は外部結合です。

C ソースコード
```
void unused_external_function(void) {
    return;
}
```

外部結合である未使用の関数および変数も、ビルドされたプログラムのメモリ中に存在します。
GNU17 IDE で、[未使用関数] 検出を有効にして、プロジェクトをビルドする場合、そのプロジェクトに未使用関数が存在すると、Splint が警告を出力します。警告は、GNU17 IDE の問題ビューに表示されます。
検出を有効にする方法の詳細については、GNU17 マニュアルの"5.7.11 未使用関数の検出"を参照してください。

Splint は、C ソースコードの解析に失敗すると'parse error'と報告して、プロジェクトの解析を実行しないことがあります。この場合、失敗する部分を Splint による解析の対象外とするために、'S_SPLINT_S' マクロを C ソースコードに挿入します。

C ソースコード
```
static void boot(void) {
#ifndef S_SPLINT_S
    asm __volatile__("xld.a %ps, __START_stack ");
    asm __volatile__("xjpr main");
```

```
#endif
}
```

警告された関数を削除することで、メモリの使用量を減らすことができます。警告された関数を削除し、プログラムを再びビルドした後に、Splint があらためて他の関数が未使用であると警告する場合があります。新しく警告された未使用関数も同様に削除してください。

7.1.3 Eclipse CDT インデクサー

（GNU17 IDE のベースとなった）Eclipse CDT には、C ソースコードのインデクサーが含まれています。GNU17 IDE メニューの［ウィンドウ］>［プロパティ］から、いくつかのインデクサーを選択できます。正確に分析するために、［プロパティ］ダイアログの［C/C++］>［インデクサー］セクションから［フル C/C++ インデクサー］を選択します。
次の手順に従い、インデクサーでプロジェクトを分析します。

- プロジェクトをビルドする。
- C/C++プロジェクトビューでコンテキストメニューを開く。
- コンテキストメニューの［インデックス］>［すべてのファイルを更新］を選択する。
- C ソースコード上で分析する名前を選択する。
- Ctrl キー＋Alt キー＋H キーを押して、呼び出し階層を開く。

結果は、呼び出し階層ビューに出力されます。呼び出し側が存在しない場合、選択された関数、変数または定数は C ソースコード中で使用されていません。ただし、S1C17 アセンブラソースコード（インラインアセンブラを含む）から使用される場合があります。

関数の呼び出し側を遡ると、S1C17 プログラムでは割り込みベクタテーブルにたどり着きます。テーブルがアセンブラにより記述される場合、割り込みベクタテーブルに登録された関数までたどり着きます。関数のアドレスが変数および定数に設定された場合、その変数および定数を参照しているものを確認してください。

未使用定数は、この方法で検出できます。

7.2 オブジェクトファイルの分析

’—cref’が GNU17 リンカのコマンドラインオプションで指定されていると、リンカは、クロスリファレンステーブルを出力します。
コマンドラインオプションをリンカに追加する方法の詳細については、GNU17 マニュアルの”5.7.5 リンカオプションの設定”を参照してください。

プロジェクトがビルドされるとき、リンカは、次に示すクロスリファレンステーブルをマップ（*.map）ファイルの最後に出力します。

GNU17 リンカによるクロスリファレンステーブル

```
Cross Reference Table

Symbol                                  File
__START_stack                           vector.o
__START_bss                             vector.o
__START_data                            vector.o
__START_data_lma                        vector.o
__END_bss                               vector.o
__END_data                              vector.o
main                                    main.o
                                        vector.o
memcpy                                  libc.a(memcpy.o)
                                        vector.o
memset                                  libc.a(memset.o)
                                        vector.o
vector                                  vector.o
```

テーブルの左側（’Symbol’列）はグローバルシンボルで、右側（’File’列）はシンボルを参照するオブジェクトファイルです。グローバルシンボルは外部結合です。

複数のオブジェクトファイルが参照するシンボルは、使用されているシンボルです。参照するオブジェクトファイルが 1 つであるシンボルは、未使用シンボルの可能性があります。参照するオブジェクトファイルが 1 つであるシンボルを、外部結合にするべきではありません。'static'を指定して、このシンボルを内部結合に変更してください。シンボルが実際に使用されていないなら、その後に GNU17 C コンパイラが警告を出力します。

7.3 環境に応じた機能の削除

7.3.1 デバッグおよびテスト用の機能

製品に組み込まれるプログラムを、デバッグおよびテスト用の機能を含むプロジェクトからビルドする場合、GNU17 IDE は、組み込まない機能を除いて、ソースコードをコンパイルしなければなりません。このため、マクロを定義することにより組み込まない機能を除外するように C ソースコードを実装します。次の例では、デバッグの必要がない状態では'NDEBUG'マクロを定義します。

C ソースコード

```
#ifdef NDEBUG
#define ASSERT(c)
#else
#define ASSERT(c)     ((c) ?TRUE:assert_output(__FILE__, __LINE__, #c), FALSE)
#endif
```

C ソースコード

```
#ifdef NDEBUG
#define PACKET_LOG(b, n)
#else
extern void packet_log_add(const char * packet, int length);
#define PACKET_LOG(b, l)          packet_log_add((b), (l))
#endif
```

C ソースコード

```
int packet_get(char * packet) {
      int length;
      int error;

      length = sio_read(packet, 16);
      ASSERT(length <= 16);

      if (length <= 0) {
            error = 0;
} else {
            PACKET_LOG(packet, length);
            if (0 == checksum(packet, length)) {
                  error = length;
} else {
                  error = ERROR_PACKET_CHECKSUM;
            }
      }

      return error;
}
```

マクロ定義の状態は、GNU17 C コンパイラのコマンドラインオプションで変更されます。詳細については、GNU17 マニュアルの"5.7.3 コンパイラオプションの設定"の［シンボル］項を参照してください。

7.3.2 S1C17 向けではない機能

PC 向けに開発された C ソースコードを GNU17 C コンパイラでコンパイルする場合、S1C17 で使用しない機能を除外する必要があります。
このため、GNU17 の定義済みマクロに基づいて、PC 用の機能を除外するように C ソースコードを実装します。

C ソースコード

```
#ifdef __c17
//GNU17用
extern int spi0_read(char * buffer, int length);
#define sio_read(b, l)   spi0_read((b), (l));
#else
//PC用
#include <stdio.h>
extern FILE * sio_in;
```

```
#define sio_read(b, l)   fread((b), 1, (l), sio_in)
#endif
```

’__c17’は、GNU17 C コンパイラの定義済みマクロで、GNU17 C コンパイラが C ソースコードをコンパイルするとき常に定義されます。

改訂履歴表

付－1

Rev. No.	日付	ページ	種別	改訂内容（旧内容を含む） および改訂理由
Rev 1.0	2013/12/19	全ページ	新規	新規制定

セイコーエプソン株式会社
マイクロデバイス事業部　IC 営業部

東京　〒191-8501　東京都日野市日野 421-8
TEL（042）587-5313（直通）　FAX（042）587-5116

大阪　〒541-0059　大阪市中央区博労町 3-5-1　エプソン大阪ビル 15F
TEL（06）6120-6000（代表）　FAX（06）6120-6100

ドキュメントコード：412655800
2013 年 12 月 作成