User's Manual

CubeSuite+ V1.00.00

統合開発環境

ユーザーズマニュアル　V850 コーディング編

対象デバイス

V850 マイクロコントローラ

本資料に記載の全ての情報は本資料発行時点のものであり、ルネサス エレクトロニクスは、
予告なしに、本資料に記載した製品または仕様を変更することがあります。
ルネサス エレクトロニクスのホームページなどにより公開される最新情報をご確認ください。

ルネサス エレクトロニクス
www.renesas.com

Rev.1.00　2011.04

ご注意書き

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

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

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

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

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

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

このマニュアルの使い方

このマニュアルは，V850 マイクロコントローラ用アプリケーション・システムを開発する際の統合開発環境である CubeSuite+について説明します。

CubeSuite+は，V850 マイクロコントローラの統合開発環境（ソフトウエア開発における，設計，実装，デバッグなどの各開発フェーズに必要なツールをプラットフォームである IDE に統合）です。統合することで，さまざまなツールを使い分ける必要がなく，本製品のみを使用して開発のすべてを行うことができます。

対 象 者　このマニュアルは，CubeSuite+を使用してアプリケーション・システムを開発するユーザを対象としています。

目　　的　このマニュアルは，CubeSuite+の持つソフトウエア機能をユーザに理解していただき，これらのデバイスを使用するシステムのハードウエア，ソフトウエア開発の参照用資料として役立つことを目的としています。

構　　成　このマニュアルは，大きく分けて次の内容で構成しています。

読 み 方　このマニュアルを読むにあたっては，電気，論理回路，マイクロコンピュータに関する一般的知識が必要となります。

凡　　例

データ表記の重み	：左が上位桁，右が下位桁
アクティブ・ロウの表記	：$\overline{\times\times\times}$（端子，信号名称に上線）
注	：本文中につけた注の説明
注意	：気をつけて読んでいただきたい内容
備考	：本文中の補足説明
数の表記	：10 進数　…　××××
	16 進数　…　0x××××

関連資料　関連資料は暫定版の場合がありますが，この資料では「暫定」の表示をしておりません。あらかじめご了承ください。

資　料　名		資料番号	
		和　文	英　文
CubeSuite+ 統合開発環境 ユーザーズ・マニュアル	起動編	R20UT0545J	R20UT0545E
	78K0 設計編	R20UT0546J	R20UT0546E
	78K0R 設計編	R20UT0547J	R20UT0547E
	RL78 設計編	R20UT0548J	R20UT0548E
	V850 設計編	R20UT0549J	R20UT0549E
	R8C 設計編	R20UT0550J	R20UT0550E
	78K0 コーディング編	R20UT0551J	R20UT0551E
	RL78,78K0R コーディング編	R20UT0552J	R20UT0552E
	V850 コーディング編	このマニュアル	R20UT0553E
	コーディング編（CX コンパイラ）	R20UT0554J	R20UT0554E
	R8C コーディング編	R20UT0576J	R20UT0576E
	78K0 ビルド編	R20UT0555J	R20UT0555E
	RL78,78K0R ビルド編	R20UT0556J	R20UT0556E
	V850 ビルド編	R20UT0557J	R20UT0557E
	ビルド編（CX コンパイラ）	R20UT0558J	R20UT0558E
	R8C ビルド編	R20UT0575J	R20UT0575E
	78K0 デバッグ編	R20UT0559J	R20UT0559E
	78K0R デバッグ編	R20UT0560J	R20UT0560E
	RL78 デバッグ編	R20UT0561J	R20UT0561E
	V850 デバッグ編	R20UT0562J	R20UT0562E
	R8C デバッグ編	R20UT0574J	R20UT0574E
	解析編	R20UT0563J	R20UT0563E
	メッセージ編	R20UT0407J	R20UT0407E

注意　**上記関連資料は，予告なしに内容を変更することがあります。設計などには，必ず最新の資料を使用してください。**

〔メ　モ〕

〔メ　モ〕

〔メ　モ〕

目　　次

第 1 章 概 説

この章では，V850 マイクロコントローラ C コンパイラ・パッケージ（CA850）の全体概要について説明します。

1.1 概 要

V850 マイクロコントローラ C コンパイラ・パッケージ（CA850）は，C 言語，またはアセンブリ言語で記述されたプログラムを機械語に変換するプログラムです。

1.2 特 長

V850 マイクロコントローラ C コンパイラ・パッケージは，次の特長を備えています。

（1）ANSI 規格に準拠した言語仕様

C 言語仕様は，ANSI 規格に準拠しています。また，従来の C 言語仕様（K&R 仕様）との両立性も備えています。

（2）高度な最適化

C コンパイラ／アセンブラによるコード・サイズ，および速度優先の最適化を提供しています。

（3）組み込み制御向け機能

アプリケーション・システムの ROM 化作業を容易にするため，ユーティリティを提供しています。

（4）記述性の向上

拡張言語仕様により C 言語プログラミングの記述性を向上させています。

第 2 章 機 能

この章では，CA850 をより効果的に用いるためのプログラミング技法，および拡張機能の利用方法について説明します。

2. 1 変数（C 言語）

この節では，変数（C 言語）について説明します。

2. 1. 1 短い命令長でアクセスできる領域へ配置する

V850 には，命令長が 2 バイトのロード／ストア命令があります。変数をこの命令でアクセスできるセクションに配置することによりコード・サイズを削減することができます。

変数の定義 / 参照時は，#pragma section を使用し，セクション種別に“tidata”を指定します。

```
#pragma section セクション種別 begin
変数宣言／定義
#pragma section セクション種別 end
```

例

```
#pragma section tidata begin
int a = 1;              /*tidata.word セクション配置 */
int b;                  /*tibss.word セクション配置 */
#pragma section tidata end
```

備考 「#pragma section 指令」を参照してください。

2. 1. 2 配置領域を変更する

デフォルトの配置セクションは，次のとおりになります

- 初期値なし変数：.sbss セクション
- 初期値あり変数：.sdata セクション
- const 変数：.const セクション

配置する領域（セクション）を変更するには，#pragma section でセクション種別を指定します。

```
#pragma section セクション種別 begin
変数宣言／定義
#pragma section セクション種別 end
```

セクション種別と生成されるセクションの関係は次の通りです。

セクション種別	初期値あり・なし	デフォルト・セクション名	セクション名変更	ベースレジスタ	アクセス命令
data	あり	.data	可	gp	ld/st 2 命令
	なし	.bss	可	gp	ld/st 2 命令
sdata	あり	.sdata	可	gp	ld/st 1 命令
	なし	.sbss	可	gp	ld/st 1 命令
sedata	あり	.sedata	不可	ep	ld/st 1 命令
	なし	.sebss	不可	ep	ld/st 1 命令
sidata	あり	.sidata	不可	ep	ld/st 1 命令
	なし	.sibss	不可	ep	ld/st 1 命令
tidata.byte	あり	.tidata.byte	不可	ep	sld/sst 1 命令
	なし	.tibss.byte	不可	ep	sld/sst 1 命令
tidata.word	あり	.tidata.word	不可	ep	sld/sst 1 命令
	なし	.tibss.word	不可	ep	sld/sst 1 命令
sconst	あり	.sconst	可	r0	ld/st 1 命令
const	あり	.const	可	r0	ld/st 2 命令

例

```
#pragma section sdata "mysdata" begin
int a = 1;              /*mysdata.sdata セクション配置 */
int b;                  /*mysdata.sbss セクション配置 */
#pragma section sdata "mysdata" end
```

なお，#pragma section 命令を使った変数に別ファイル（参照ファイル）の関数から参照する場合には，参照ファイルにも #pragma section 命令を記述し，該当変数を extern 宣言する必要がります。

例 変数を定義しているファイル

```
#pragma section sconst  begin
const unsigned char table_data[9] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
#pragma section sconst  end
```

例 変数を参照するファイル

```
#pragma section sconst  begin
extern const unsigned char  table_data[];
#pragma section sconst  end
```

備考 「#pragma section 指令」を参照してください。

2. 1. 3 通常時と割り込み時に使用する変数を定義する

通常時の処理と割り込みの処理の両方で使用する変数は，volatile 指定してください。

volatile 修飾子をつけて変数宣言すると，その変数は最適化の対象からはずされ，レジスタに割り付ける最適化などを行わなくなります。volatile 指定された変数に対する操作を行うときは，必ずメモリから値を読み込み，操作後にメモリへ値を書き込むコードになります。また volatile 指定された変数のアクセス幅も変更されません。volatile 指定されていない変数は，最適化によってレジスタに割り付けられ，その変数をメモリからロードするコードが削除されることがあります。また volatile 指定されていない変数に同じ値を代入する場合，冗長な処理と解釈されて最適化によりコードが削除されることもあります。

【volatile 指定した場合のソースと出力コードの例】

“変数 a”，“変数 b”，および “変数 c” を volatile 指定した場合，これらの変数値を必ずメモリから読み込み，操作後にメモリへ書き込むコードが出力されます。たとえば，この間に割り込みが入り，割り込み内で変数値が変更されても，その変更が反映された結果を取得することができます（このような例の場合，割り込みのタイミングによっては，変数の操作区間内を割り込み禁止にするなどの処置が必要となります）。

volatile 指定をすると，メモリの読み込み／書き込み処理が入るため，volatile 指定しなかった場合よりもコード・サイズは大きくなります。

```
volatile int a;
volatile int b;
volatile int c;
void func(void) {
if (a <= 0) {
    b++;
} else {
    c++;
}
b++;
c++;
}
```

```
_func:
    .option volatile
    ld.w $_a, r10
    .option novolatile
    cmp r0, r10
    jgt .L2
    .option volatile
    ld.w $_b, r11
    .option novolatile
    add 1, r11
    .option volatile
    st.w r11, $_b
    .option novolatile
    jbr .L3
.L2:
    .option volatile
    ld.w $_c, r12
    .option novolatile
    add 1, r12
    .option volatile
    st.w r12, $_c
    .option novolatile
.L3:
    .option volatile
    ld.w $_b, r13
    .option novolatile
    add 1, r13
```

	.option volatile st.w r13, $_b .option novolatile .option volatile ld.w $_c, r14 .option novolatile add 1, r14 .option volatile st.w r14, $_c .option novolatile jmp [lp]

2.1.4 ユーザ・ポートを定義する

ユーザ・ポートについては，次の例のように，volatile 指定して，最適化を避けてください。

例 ポート記述の手順例

```
/* 1. ポート・マクロ（型）の定義 */
#define DEFPORTB(addr)  (*((volatile unsigned char *)addr))   /* 8 ビット・ポート */
#define DEFPORTH(addr)  (*((volatile unsigned short *)addr))  /* 16 ビット・ポート */
#define DEFPORTW(addr)  (*((volatile unsigned int *)addr))    /* 32 ビット・ポート */
/* 2. ポートの定義（例：PORT1 0x00100000 8bit）*/
#define PORT1   DEFPORTB(0x00100000)    /* 0x00100000  8 ビット・ポート */
/* 3. ポートの使用 */
{
  PORT1 = 0xFF;   /* PORT1 への書き込み */
  a = PORT1;      /* PORT1 からの読み出し */
}
/* 4.C コンパイラの出力コード */
  :
mov     1048576, r10
#@BEGIN_VOLATILE
st.b    r20, [r10]
#@END_VOLATILE
mov     1048576, r11
#@BEGIN_VOLATILE
ld.b    [r11], r12
#@END_VOLATILE
  :
```

備考 1. 構造体を宣言し，その構造体変数を特定のセクションに割り当て，リンク・ディレクティブで対応するポート・アドレスに割り当てることにより，CA850 の内蔵周辺 I/O レジスタと同様に“X.X”形式のビット・アクセスができます。
ただし，1 ビット／8 バイト・アクセスがある場合，ビット・フィールドとバイトの共用体にする必要があるので，“X.X.X”形式になります。

2. 変数のセクション割り当ては，#pragma section やセクション・ファイルで行ってください。

2. 1. 5 const 定数ポインタを定義する

ポインタについては，“const”の指定場所により，異なる解釈がされます。

なお，.const セクションを .sconst セクションに割り当てるときは #pragma section sconst 指定をしてください。

- const char *p ;

ポインタが示すオブジェクト（*p）を書き換えできないことを示します。

ポインタ自体（p）は書き換え可能です。

したがって，以下のようになり，ポインタ自体は RAM（.sdata/.data）に配置されます。

```
*p = 0;      /* エラー */
p = 0;       /* 正しい */
```

- char *const p ;

ポインタ自体（p）を書き換えできないことを示します。

ポインタが示すオブジェクト（*p）は書き換え可能です。

したがって，以下のようになり，ポインタ自体は ROM（.sconst/.const）に配置されます。

```
*p = 0;      /* 正しい */
p = 0;       /* エラー */
```

- const char *const p ;

ポインタ自体（p），ポインタが示すオブジェクト（*p）を書き換えできないことを示します。

したがって，以下のようになり，ポインタ自体は ROM（.sconst/.const）に配置されます。

```
*p = 0;      /* エラー */
p = 0;       /* エラー */
```

2.2 関　　数

この節では，関数について説明します。

2.2.1 配置領域を変更する

関数のセクション名を変更する場合は，以下のように #pragma text 指令を使用して関数を指定します。

```
#pragma text  [" セクション名 "]  関数名
#pragma text  [" セクション名 "]
```

また，セクション名を変更した text 属性のセクションは，リンク・ディレクティブで入力セクションを作成したときのセクション名を指定してください。

例　C ソース内で「#pragma text "sec1" func1」と記述した場合のリンク・ディレクティブの記述方法（セグメント名：FUNC1）

```
FUNC1:  !LOAD ?RX{
            sec1.text = $PROGBITS ?AX sec1.text;
        };
```

#pragma text 指令で，特定の関数を独自に指定した text 属性のセクションに配置する場合，実際に生成されるセクション名は“指定した文字列＋ .text”となり，このセクション名をリンク・ディレクティブに記述する必要があります。

上記の例であれば“sec1.text セクション”になります。

備考　「#pragma text 指令」を参照してください。

2. 2. 2 離れた関数をコールする

C コンパイラは，関数呼び出しに jarl 命令を使用します。

しかし，jarl 命令は 22 ビット・ディスプレースメントであるため，プログラム配置によってはアドレス解決ができず，リンク時にエラーとなります。

このような場合，C コンパイラの -Xfar_jump オプションで，関数呼び出しを ディスプレースメント幅に依存しない関数呼び出しにすることができます。

far jump を指定した関数の呼び出しに対しては，jarl 命令ではなく jmp 命令が出力されます。

-Xfar_jump オプションで指定するファイルには，1 行に 1 関数を記述していきます。記述する名前は，C 言語関数名の先頭に“_（アンダースコア）”を付けた名前になります。

例

```
_func_led
_func_beep
_func_motor
     :
     :
_func_switch
```

“_ 関数名”のかわりに次のようにを記述すると，すべての関数を far jump 呼び出しの対象にします。

```
{all_function}
```

備考 「far jump 機能」を参照してください。

2. 2. 3 アセンブラ命令の埋め込み

CA850 では，次に示す形式において，C 言語ソース・プログラム中にアセンブラ命令が記述できます。

- asm 宣言

```
__asm( 文字列定数 );   または  _asm( 文字列定数 );
```

- #pragma 指令

```
#pragma asm
    アセンブラ命令
#pragma endasm
```

挿入するアセンブラ命令でレジスタを使用する場合，必要な退避／復帰はプログラム内で行ってください。CA850 では行いません。

例

```
__asm("nop ");
__asm(".str \"string\\0\"");;

#pragma asm
mov r0, r10
st.w r10, $_i
#pragma endasm
```

なお，asm 宣言や #pragma asm ～ #pragma endasm 指令内に書かれたアセンブラ命令は，C 言語による #define 定義されたものがアセンブラ・ソース内にあっても展開されることはありません。

また，asm 宣言や #pragma asm ～ #pragma endasm 指令内のアセンブラ命令中のマクロ疑似命令などは，そのままアセンブラに渡されるため，C コンパイラで -P オプションをつけても展開されません。

備考 「アセンブラ命令の記述」を参照してください。

2. 2. 4 RAM で実行する

リンク時とコピー後で，各セクションと各シンボル（r0，TP，EP，GP）の相対値を壊さなければ，ROM 内に配置されているプログラムを RAM にコピーして，RAM にてプログラムを実行することができます。

コピーされるものと，されないものが存在するので，注意してください。

リセット後に，内蔵 RAM にコピーして，そのプログラムが変更されないのであれば，ROM 化の機能を使用したほうが簡単に実現できます。

romp850 で text セクションをパッキング対象にすることが可能です。

2.3 マイコン機能の使用

この節では，マイコン機能の使用について説明します。

2.3.1 C 言語での周辺 I/O レジスタへアクセスする

C 言語でデバイス内部の周辺 I/O レジスタを読み書きする場合，C ソースにプラグマ指令を追加することにより，周辺 I/O レジスタ名やビット名を用いて読み書きすることが可能となります。

周辺 I/O レジスタ名は，通常の符号なし（unsigned）外部変数のように扱うことができます。ただし，& 演算子で，周辺 I/O レジスタのアドレスを取得することはできません。

```
#pragma ioreg
  レジスタ名 = …
  レジスタ名．ビット番号 = …
  ビット名 = …
```

上記プラグマ指令を記述した以降，周辺 I/O レジスタ名が使用可能となります。

例

```
#pragma ioreg
void main( void ) {
  int i;
  P0 = 1;      /* P0 に 1 を書き込む */
  i = RXB0;    /* RXB0 から読み込み */
}

void func(void) {
  P1 = 0;      /* P1 に 0 を書き込む */
}

void func2(void) {
  P0.1 = 1;    /* P0 のビット 1 を 1 にする */
  P2.3 = 0;    /* P2 のビット 3 を 0 にする */
  PS00 = 1;    /* ビット名 PS00 のビットを 1 にする */
}
```

なお，周辺 I/O レジスタのビット名称は，該当ビット名称が CA850 によって定義されているものに限定されます。

したがって，ビット名称が未定義の場合はエラーになります。

未定義のビットにアクセスする場合は，“レジスタ名．ビット番号”となります。

備考 1. ポート 3 の 4 ビット目をアクセスする場合は，“P3.4”となります。

2. 「周辺 I/O レジスタへのアクセス」を参照してください。

2. 3. 2 C 言語での割り込み処理を記述する

CA850 では，割り込みハンドラの指定を“#pragma interrupt 指令”および“__interrupt 修飾子”（通常の割り込みの場合），または“#pragma interrupt 指令”および“__multi_interrupt 修飾子”（多重割り込みの場合）で行います。

以下に，割り込みハンドラの記述例を示します。

例 ノンマスカブル割り込みの場合

```
#pragma interrupt NMI func1    /* ノンマスカブル割り込み */
__interrupt
void func1(void) {
   :
}
```

例 多重割り込み指定の場合

```
#pragma interrupt INTP0 func2
__multi_interrupt              /* 多重割り込み関数指定 */
void func2(void) {
   :
}
```

備考 「割り込み／例外処理ハンドラ」を参照してください。

2. 3. 3 C 言語での CPU 命令を使用する

アセンブラ命令の一部を “組み込み関数” として C 言語ソースに記述することができます。ただし，“アセンブラ命令そのもの” を記述するのではなく，CA850 で用意した関数の形式で記述します。

次に，関数として記述できる命令を示します。

アセンブラ命令	機能	組み込み関数
di	割り込み制御（di）	`__DI()`
ei	割り込み制御（de）	`__EI()`
nop	nop	`__nop()`
halt	halt	`__halt()`
satadd	飽和加算（satadd）	`long a, b;` `long __satadd(a, b);`
satsub	飽和減算（satsub）	`long a, b;` `long __satsub(a, b);`
bsh	ハーフワード・データのバイト・スワップ（bsh）【V850E】	`long a;` `long __bsh(a);`
bsw	ワード・データのバイト・スワップ（bsw）【V850E】	`long a;` `long __bsw(a);`
hsw	ワード・データのハーフワード・スワップ（hsw）【V850E】	`long a;` `long __hsw(a);`
sxb	バイト・データの符号拡張（sxb）【V850E】	`char a;` `long __sxb(a);`
sxh	ハーフワード・データの符号拡張（sxh）【V850E】	`short a;` `long __sxh(a);`
mul	mul 命令を用いて乗算結果の上位 32 ビットを変数に代入する【V850E】	`long a; long b;` `long __mul32(a, b);`
mulu	mulu 命令を用いて符号なし乗算結果の上位 32 ビットを変数に代入する 【V850E】	`unsigned long a, b;` `unsigned long __mul32u(a, b);`

sasf	論理左シフト付きフラグ条件の設定（sasf）【V850E】	`long a;`
		`unsigned int b;`
		`long __sasf(a, b);`

例

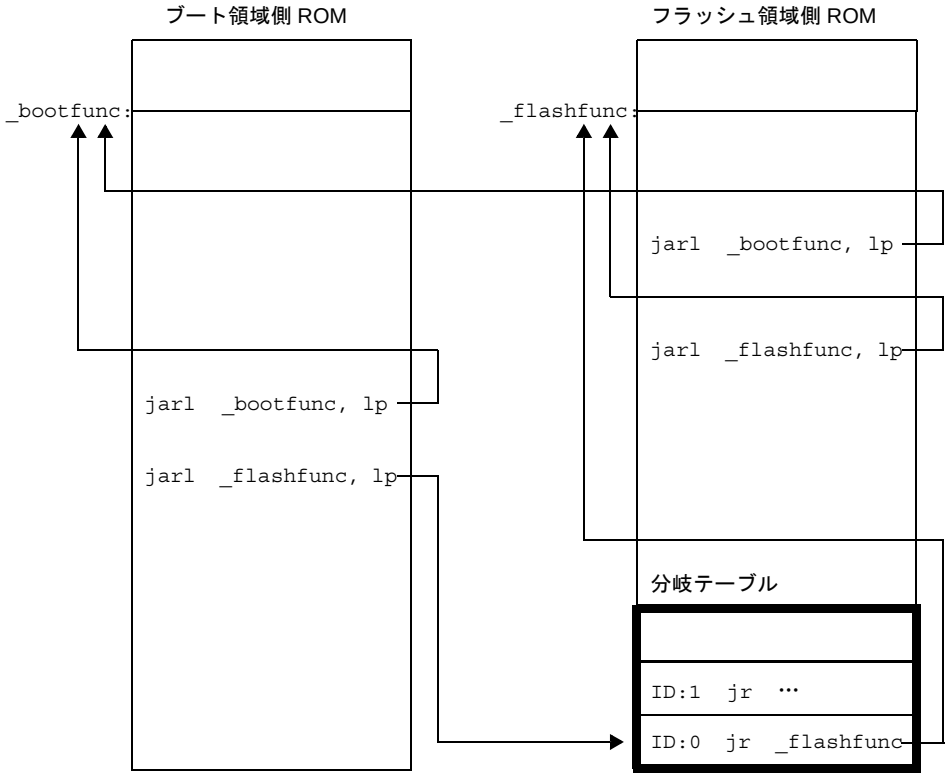
```
long a, b, c;
void func(void) {
        :
    c = __satsub(a, b);      /* a とb の飽和演算結果をc に格納する(c = a - b) */
        :
    __nop();
        :
}
```

備考 「組み込み関数」を参照してください。

2. 3. 4 セルフプログラミングのブート領域を作成する

フラッシュ領域 - ブート領域の変数／関数は，以下の操作により参照することができます。

- フラッシュ領域からは，ブート領域の関数を直接呼び出すことができます。
- ブート領域からフラッシュ領域への関数呼び出しは，分岐テーブルを介して行います。
- フラッシュ領域からは，ブート領域の外部変数を参照できます。
- ブート領域からは，フラッシュ領域の外部変数を参照できません。
- ブート領域のプログラムとフラッシュ領域のプログラムで，同じ外部変数および外部関数を定義できます。
  この場合，定義と同じ領域側にある変数または関数が参照されます。



ブート領域から呼び出すフラッシュ領域の関数を .ext_func 疑似命令で指定します。

```
.ext_func 関数名 , ID 番号
```

例 C 言語プログラム中

```
#pragma asm
.ext_func _func_flash0, 0
.ext_func _func_flash1, 1
.ext_func _func_flash2, 2
#pragma endasm
```

その他にオプション等で設定する必要があります。詳細は「V850 ビルド編」の“フラッシュ再リンク機能”を参照してください。

2.4 変数（アセンブラ）

この節では，変数（アセンブラ）について説明します。

2.4.1 初期値なし変数を定義する

初期値なし変数領域を確保するには，初期値なしセクション中で，.lcomm 疑似命令を使用します。

```
.lcomm ラベル名 , サイズ , 整列条件
```

他のファイルからも参照できるようにするには，そのラベルを .globl 疑似命令で宣言する必要があります。

```
.globl ラベル名 [, サイズ ]
```

例

```
.globl val0             -- val0 を他のファイルから参照できるようにします
.globl val1             -- val1 を他のファイルから参照できるようにします
.globl val2             -- val2 を他のファイルから参照できるようにします
.sbss
.lcomm val0, 4, 4       -- val0 は 4 バイトの領域を確保し整列条件を 4 とします
.lcomm val1, 2, 2       -- val1 は 2 バイトの領域を確保し整列条件を 2 とします
.lcomm val2, 1, 1       -- val2 は 1 バイトの領域を確保し整列条件を 1 とします
```

備考 「.lcomm」,「.globl」を参照してください。

2. 4. 2 初期値あり const 定数を定義する

初期値あり const 定数を定義するには，.const/.sconst セクション中で，次の疑似命令を使用します。

- 1 バイトの値の場合

```
.byte 値 [, 値 , …]
```

- 2 バイトの場合

```
.hword 値 [, 値 , …]
```

- 4 バイトの場合

```
.word 値 [, 値 , …]
```

例 1 ハーフワード分確保し，100 を格納

```
    .const
    .align 4
    .globl _p, 2
_p:
    .hword 100
```

備考 「.byte」，「.hword」，「.word」を参照してください。

2. 4. 3 セクションのアドレスを参照する

.data や .sdata などのセクションの先頭と末尾を指すシンボル（予約シンボル）が用意されています。したがって，アセンブラ・ソースから特定セクションのアドレス値を使用する際には，該当シンボル名を利用します。

先頭シンボル： __s セクション名

末尾シンボル： __e セクション名

たとえば，.sbss セクションの先頭シンボルは __ssbss，末尾シンボルは __esbss という名前になります。

これらのシンボルを使用することにより，セクションの先頭アドレスと末尾アドレスを取得することができますが，C 言語レベルでは，これらのシンボル名を使用して直接参照することはできません。

これらのシンボル値を取得するには，この値を格納する外部変数を作成し，スタート・アップ・モジュールなどのアセンブラ・ソースにて，変数にシンボル値を格納します。

その変数を C ソース上で参照することによって実現することができます。

これは __gp_DATA などのシンボルについても同じです。

たとえば，.data セクションの先頭アドレスと末尾アドレスを取得する方法は，次のようになります。

例 アセンブラ・ソース内

```
.comm   _data_top, 4, 4
.comm   _data_end, 4, 4
.extern __sdata, 4
.extern __edata, 4
mov     #__sdata, r12
st.w    r12, $_data_top
mov     #__edata, r13
st.w    r13, $_data_end
```

例 C ソース内

```
extern  int data_top;   /* data_top を extern 宣言する */
extern  int data_end;   /* data_end を extern 宣言する */
void func1(void){
  int top, end;
  top = data_top;
  end = data_end;
   :
}
```

特定のセクションのみを，C 言語レベルで初期化するような場合，この方法を用いてください。

2.5 スタートアップ・ルーチン

この節では，スタートアップ・ルーチンについて説明します。

2.5.1 スタック領域を確保する

スタック・ポインタ（sp）に値を設定する際には，以下の点に注意する必要があります。

- スタック・フレームは，sp に設定した値から下位方向に生成されます。
- sp は，必ず 4 バイト境界の位置を指すように設定してください。
- コンパイラはスタック相対でメモリを参照する場合，スタック・ポインタが 4 バイト境界の位置を指していることを想定して，コードを生成しています。
  なるべく gp と離れたデータ・セクション（bss 属性セクション）に配置してください。
  gp と近いと，プログラムのデータ領域を破壊するおそれがあります。

例 sp の設定例

```
.set    STACKSIZE, 0x3f0
.bss
.lcomm  __stack, STACKSIZE, 4
mov     #__stack + STACKSIZE, sp
```

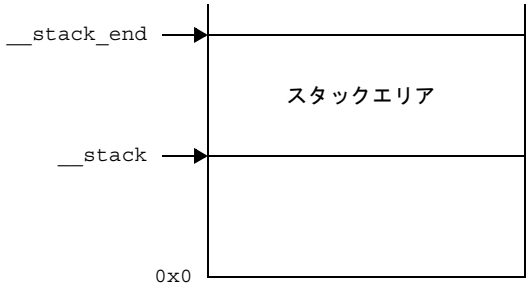
なお，上記の例では，アプリケーションで使用するスタック・フレームのサイズを，0x3f0 バイトと指定して領域を確保します。
“__stack”は，スタック・フレームの最下位（先頭）を指すラベルとなります。

デフォルトのスタート・アップ・モジュールでは，__stack を外部変数定義（.globl 宣言）をしていないため，他ファイルから __stack の参照ができません。
そこで，__stack に対して .globl 宣言を実施すれば，他ファイルからの参照が可能になります。
また，スタック・エリアは，最下位アドレスに __stack というシンボルを定義して，スタック・ポインタに __stack のアドレスとサイズを加算したものを設定しています。
そのため，エンドのアドレスにシンボルはありません。
次のようにすることにより，スタック・エリアのエンドのアドレスの次のアドレスを定義することは可能になります。
スタック・エリアの最後のアドレスではないので注意してください。

```
.set    STACKSIZE, 0x200
.bss
.globl  __stack                 -- 追加
.globl  __stack_end             -- 追加
.lcomm  __stack, STACKSIZE, 4
.lcomm  __stack_end, 0, 0       -- 追加
```

上記の定義により，C ソース上で，_stack，_stack_end のシンボルを参照することが可能です。

マッピングのイメージは，次のようになります。



なお，__stack シンボルは，スタート・アップ・モジュールで，サイズを指定してあるので C ソースでは，次のように配列で定義してください。

スタック・エリアの最後のアドレスではないので注意してください。

```
extern unsigned long  _stack[];
```

備考 アセンブラで定義したラベルを C 言語で使用する場合には，先頭のアンダーバー“_”を 1 つ削除した名前になります。

アセンブリ言語定義： __stack

C 言語での参照： stack

C ソース・プログラムのスタック領域については，スタック見積りツール（stk850）を使うことにより計測することがきでます。

2. 5. 2 スタック領域を確保し配置を指定する

この項では，スタック領域を確保し配置を指定する方法について説明します。

(1) スタック領域の確保

スタートアップ・ルーチンで，セクション名を指定した初期値なし変数のセクションにスタック領域を確保します。

例 sp の設定例

```
.set        STACKSIZE, 0x200
.section    ".stack", bss
.lcomm      __stack, STACKSIZE, 4
```

なお，上記の例では，アプリケーションで使用するスタック・フレームのセクションを .stack，サイズを 0x200 バイトと指定して領域を確保します。

“__stack”は，スタック・フレームの最下位（先頭）を指すラベルとなります。

(2) スタック領域の配置指定

リンクディレクティブ・ファイルで，(1) で作成したセクションの配置を指定します。

例 配置指定例

```
STACK   : !LOAD ?RW V0x3ffee00 {
    .stack  = $NOBITS       ?AW  .stack;
};
```

なお，上記の例では，スタック領域用のセグメントを STACK とし，0x3ffee00 番地に配置しています。

2. 5. 3 RAM を初期化する

この項では，RAM の初期化について説明します。

(1) 初期値なし変数

デフォルトのスタートアップ・ルーチンでは，.sbss セクション，.bss セクションを 0 でクリアする処理が組み込まれています。

上記以外のセクションを使用した場合に，0 クリアしたい場合には，スタートアップ・ルーチンに処理を追加してください。クリアする際には，セクションの先頭と終端を示すシンボルを使用します。

例 .tibss.byte セクションの 0 クリア

```
.extern __stibss.byte, 4        --  .tibss.byte 領域の先頭シンボル
.extern __etibss.byte, 4        --  .tibss.byte 領域の終端シンボル
mov #__stibss.byte, r13
mov #__etibss.byte, r12
cmp r12, r13
jnl .L20
.L21:
st.w r0, [r13]
add 4, r13
cmp r12, r13
jl .L21
.L20:
```

(2) RAM の初期化

ROM 化を行わず，インサーキット・エミュレータにロード・モジュールをダウンロードした場合，data 領域や sdata 領域に置かれる初期値ありデータは，ダウンロードした時点でセットされます。

リンカが出力したロード・モジュールを使用してデバッグするときは，初期値あり変数の RAM 領域を 初期化するような処理は外す必要があります。初期値なし変数の RAM 領域の初期化だけであれば，外す必要はありません。

ROM 化用ロード・モジュールの場合は，コピー関数 _rcopy で初期値ありデータ・コピー等を行う必要があります。

この処理は，スタートアップ・ルーチンではなく，main 関数の初期値あり変数をアクセスする前でも可能ですので，周辺の設定が完了した後に，行ってください。

2. 5. 4 関数／変数アクセスを準備する

関数をアクセスする際には，テキスト・ポインタを，変数をアクセスする際には，グローバル・ポインタ，または，エレメント・ポインタを使用します。

(1) 関数アクセスの準備

アプリケーションのテキスト領域であるプログラム・コードを参照する際に，配置される位置に依存しない参照（PIC：Position Independent Code）を実現するために用意されているポインタが“テキスト・ポインタ（tp）”です。たとえば，プログラム実行中に，コード内のある箇所を参照する必要がある場合，CA850 は tp 相対でアクセスするコードを出力します。

したがって，tp が正しく設定されていることを前提としたコードを出力しているので，スタート・アップ・ルーチン内で tp を正しく設定する必要があります。

テキスト・ポインタの値は，リンク時に決定され，リンク・ディレクティブ・ファイル内に書かれる“シンボル・ディレクティブ”に定義されたシンボルに入っています。たとえば，テキスト・ポインタのシンボル・ディレクティブが次のように記述されていたとします。

```
__tp_TEXT @ %TP_SYMBOL {TEXT};
```

このとき，テキスト・ポインタの値は“TEXT セグメント”の先頭になり，その値は“__tp_TEXT”に入ります。

スタート・アップ・ルーチン内で tp をセットするには，次のように記述してください。

```
.extern __tp_TEXT, 4
mov     #__tp_TEXT, tp
```

(2) 変数アクセスの準備（グローバル・ポインタの設定）

アプリケーション内で定義した外部変数／データはメモリ上に配置されます（.sdata/.sbss セクション）。そのメモリに配置されている変数／データを参照する際，配置位置に依存することのない参照（PID：Position Independent Data）を実現するために用意されているポインタが“グローバル・ポインタ（gp）”です。gp 相対でアクセスするセクションが存在する場合，CA850 は gp 相対でアクセスするコードを出力します。

したがって，gp が正しく設定されていることを前提としたコードを出力しているので，スタート・アップ・ルーチン内で gp を正しく設定する必要があります。

グローバル・ポインタの値は，リンク時に決定され，リンク・ディレクティブ・ファイル内に書かれる“シンボル・ディレクティブ”に定義されたシンボルに入っています。たとえば，グローバル・ポインタのシンボル・ディレクティブが次のように記述されていたとします。

```
__gp_DATA @ %GP_SYMBOL {DATA};
```

また，gp シンボル値は，上記のように“DATA セグメントなどの「データ用セグメント」の先頭を gp シンボル値とする方法”のほかに，“テキスト・シンボルからのオフセットを gp シンボル値とする”方法もあります。

この方法の場合，gp シンボルを「tp に，tp からのオフセット値を加える」ことによって決定できます。つまり，配置に依存しないコードの生成が可能になります。たとえば，“プログラム・コード”と“そのコードが使用するデータ”を同時 RAM 領域にコピーしてから実行させたい場合，コードの先頭（コピー先の先頭アドレス）さえ分かれば，gp の値もすぐ導き出せるというメリットがあります。この場合のシンボル・ディレクティブ記述は次のようになります。

```
__tp_TEXT @ %TP_SYMBOL;
__gp_DATA @ %GP_SYMBOL &__tp_TEXT {DATA};
```

グローバル・ポインタの値は，“__tp_TEXT に __gp_DATA の値を加えた値”となり，加える値（オフセット値）が“__gp_DATA”に入ります。したがって，スタート・アップ・ルーチン内で gp をセットするには，次のように記述します。

```
.extern __tp_TEXT, 4
.extern __gp_DATA, 4
mov     #__tp_TEXT, tp
mov     #__gp_DATA, gp
add     tp, gp
```

これにより，正しいグローバル・ポインタの値が gp に設定されます。

(3) 変数アクセスの準備（エレメント・ポインタの設定）

アプリケーション内で定義した外部変数／データのうち，次に割り当てられているものは，エレメント・ポインタ（ep）からの相対でアクセスされます。

- sedata ／ sebss セクション
- sidata ／ sibss セクション
- tidata ／ tibss セクション
- tidata.byte ／ tibss.byte セクション
- tidata.word ／ tibss.word セクション

これらのセクションが存在する場合，CA850 は ep 相対でアクセスするコードを出力します。

したがって，ep が正しく設定されていることを前提としたコードを出力しているので，スタート・アップ・ルーチン内で ep を正しく設定する必要があります。

エレメント・ポインタの値は，リンク時に決定され，リンク・ディレクティブ・ファイル内に書かれる“シンボル・ディレクティブ”に定義されたシンボルに入っています。たとえば，エレメント・ポインタのシンボル・ディレクティブが次のように記述されていたとします。

```
__ep_DATA @ %EP_SYMBOL;
```

エレメント・ポインタの値は，デフォルトで“SIDATA セグメントの先頭”になり，その値は“__ep_DATA”に入ります。

したがって，スタート・アップ・ルーチン内で ep をセットするには，次のように記述します。

```
.extern __ep_DATA, 4
mov     #__ep_DATA, ep
```

__ep_DATA の絶対アドレス参照を行い，その値を ep に設定します。

2. 5. 5 コードサイズ削減機能を使用する準備する

V850Ex コアを使用している場合で，コードサイズを削減するために，プロローグ／エピローグ・ランタイム・ライブラリを使用する場合（より高度な最適化（実行速度優先）を指定していない場合，または，-Xpro_epi_runtime=on を指定している場合）に，この設定が必要になります。

V850Ex コアで関数のプロローグ／エピローグ・ランタイム・ライブラリを呼び出すとき，CALLT 命令を使用するため，その CALLT 命令に必要な CTBP の値を，関数のプロローグ・エピローグ・ランタイム・ライブラリの関数テーブルの先頭に設定しておく必要があります。

プロローグ／エピローグ・ランタイム・ライブラリを使用する設定になるのは，次の場合です。

- コンパイラ・オプション“-Xpro_epi_runtime=on”を設定している

コンパイラの最適化オプション“-Ot”「以外」を指定していると，自動的に“-Xpro_epi_runtime=on”になります。

関数のプロローグ／エピローグ・ランタイム・ライブラリの関数テーブルの先頭シンボルは，次のとおりです。

- ___PROLOG_TABLE

このシンボルを用いて，次のコードを記述します。

```
mov #___PROLOG_TABLE, r12
ldsr r12, 20
```

CTBP はシステム・レジスタ 20 番なので，ldsr 命令を使用して，値を設定します。

2. 5. 6 スタートアップルーチンを終了する

スタートアップ・ルーチンの処理の最後の処理は，リアルタイム OS を使用するかしないにより違いがあります。

(1) リアルタイム OS を使用しない場合

スタート・アップ・ルーチンで行う必要のある処理がすべて終わったとき，main 関数への分岐命令を実行します。

main 関数への分岐には，次のコードを記述します。

```
jarl _main, lp
```

また，main 関数の実行がすべて終わった後，この分岐命令の 4 バイト先に戻ってくることになります。戻って来ないことが分かっている場合は，次の命令も使用できます。

```
jr _main
```

```
mov #_main, lp
jmp [lp]
```

jmp 命令を使用すると，32 ビット全空間をアクセスすることができます。もし，“jarl _main, lp”を使用する場合は，main 関数実行後に戻ってくるので，戻ってきた後，デッドロックしないように，対策を施しておくと安全です。

(2) リアルタイム OS（RI850V4）を使用する場合

リアルタイム OS を使用したアプリケーションで，スタート・アップ・ルーチンで行う必要のある処理がすべて終わったとき，リアルタイム OS の初期化ルーチンへ分岐します。

```
    .extern __kernel_sit
    .extern __kernel_start
    mov     #__kernel_sit, r6
    mov     #__kernel_start, r11
    jarl    __jump_kernel_start, lp
__boot_error:
    jbr     __boot_error
__jump_kernel_start:
    jmp     [r11]
```

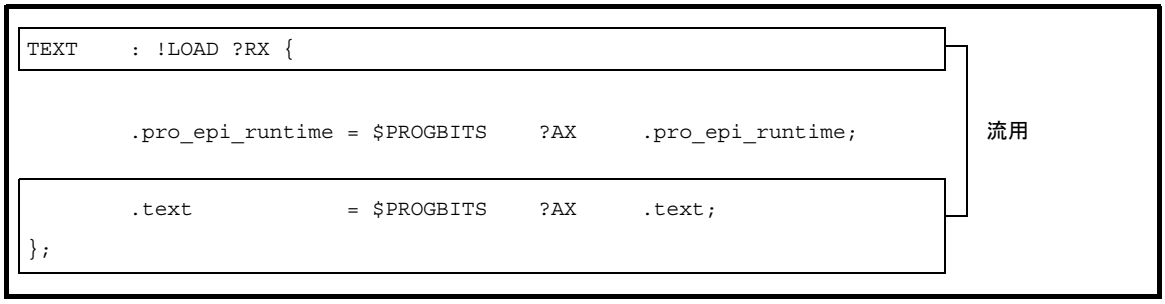
2.6 リンク・ディレクティブ

この節では，リンク・ディレクティブについて説明します。
リンク・ディレクティブ・ファイルは、CubeSuite+ 上で自動生成することも可能です。

備考 リンク・ディレクティブ・ファイルの自動作成方法については，「CubeSuite+ V850 ビルド編」のユーザーズ・マニュアルを参照してください。

2.6.1 関数のセクション配置を追加する

関数のセクション配置をするには，.text セクションの指定部分を流用し，セグメント名，セクション名を変更してください。



例 USRTEXT セグメント，usr.text セクションの配置を指定

```
USRTEXT : !LOAD ?RX {
        usr.text        = $PROGBITS    ?AX     usr.text;
};
```

2. 6. 2 変数のセクション配置を追加する

変数のセクションの配置指定を追加するには，同じ属性のセクション指定部分を流用して，セグメント名，セクション名を変更してください。

セクション属性は，#pragma section で変数にセクションを指定する際にセクション種別を指定しています。

セクション種別	流用するセクション
data	.data/.bss
sdata	.sdata/.sbss
sconst	.sconst
const	.const

例 USRCONST セグメント，usr.const セクションの配置を指定

```
USRCONST        : !LOAD ?R {

                  usr.const       = $PROGBITS     ?A usr.const;
};
```

2. 6. 3 セクション配置を振り分ける

セクション配置を振り分ける方法として，次の 3 通りの方法があります。

(1) セクション名で振り分ける

C ソースやアセンブラソースで配置するセクション名で別々の名前を指定します。

リンクディレクティブ中では，入力セクション名をそれぞれ指定することにより，その名前のセクションが，指定した部分に配置されます。

例

```
TEXT : !LOAD ?RX{
.text = $PROGBITS ?AX .text ;                    ← .text セクションを配置
};
FUNC1 : !LOAD ?RX{
funcsec1.text = $PROGBITS ?AX funcsec1.text ;    ← funcsec.text セクションを配置
};
```

(2) オブジェクトファイル名で振り分ける

リンクディレクティブ中では，入力オブジェクト名をそれぞれ指定することにより，そのオブジェクト中の該当する属性のセクションが，指定した部分に配置されます。

例

```
TEXT1 : !LOAD ?RX {
.text1 = $PROGBITS ?AX { file1.o file2.o }; ← file1.o , file2.o の TEXT 属性のセクションを配置
};
TEXT2 : !LOAD ?RX{
.text2 = $PROGBITS ?AX { file3.o };          ← file3.o の TEXT 属性のセクションを配置
};
```

なお，ライブラリ (.a ファイル) 内のオブジェクト・ファイル名を指定する場合，.a ファイル名を ( ) で囲んで指定します。

例

```
.text2 = $PROGBITS ?AX .text {rcopy.o(c:\micomtools\lib850\r32\libr.a)};
```

(3) セクション属性で振り分ける

入力セクション，入力オブジェクトを指定せずに，属性のみで配置指定します。この指定は，セクション名や，オブジェクト名を指定する部分よりも優先度が低いので，セクション名や，オブジェクト名を指定していない部分全部の配置を指定するときに使用できます。

例

```
TEXT1 : !LOAD ?RX V0x100000{
.text1 = $PROGBITS ?AX{file1.o file2.o}; ← file1.o , file2.o の TEXT 属性のセクションを配置
};
TEXT2 : !LOAD ?RX V0x120000{
.text2 = $PROGBITS ?AX ;                ← file1.o , file2.o 以外のオブジェクト中の TEXT 属性のセクションを配置
};
```

(4) 配置指定の優先度

入力セクションの指定，入力オブジェクトの指定の有無により，優先度があります。リンカは，セクション配置時に優先度が高い指定から配置していきます。

優先度と指定の関係を次に示します（優先度は数値が小さいものが優先度が高くなります）。

優先度	指定	出力
1	入力セクション名 + オブジェクト・ファイル名	指定した入力セクションを，指定したオブジェクトから抽出して出力
2	入力セクション名のみ	指定した入力セクションを，すべてのオブジェクトから抽出して出力
3	オブジェクト・ファイル名のみ	作成する出力セクションと同じ属性を持つセクションを，指定されたオブジェクトから抽出して出力
4	両方とも記述しなかった場合	作成する出力セクションと同じ属性を持つセクションを，すべてのオブジェクトから抽出して出力

2.7 コード・サイズの削減

この節では，コード・サイズの削減について説明します。

2.7.1 コード・サイズの削減（C 言語）

この項では，コード・サイズの削減（C 言語）について説明します。

(1) 変数へのアクセス

外部変数アクセスではロード／ストアに各 4 バイト必要となるため，代入以外の場合にも，外部変数の値をいったんテンポラリ変数に代入してそのテンポラリ変数を使い回すと，メモリ・アクセスがレジスタ・アクセスに変わりコード・サイズが減る可能性があります。

以下の例で s は外部変数であるとします。

変更前

```
if(x != 0){
  if((s & 0x00F00F00) != MASK1){
    return;
  }
  s >>= 12;
  s &= 0xFF;
}else{
  if((s & 0x00FF0000) != MASK2){
    return;
  }
  s >>= 24;
}
```

変更後

```
unsigned int  tmp = s;
if(x != 0){
  if((tmp & 0x00F00F00) != MASK1){
    return;
  }
  tmp >>= 12;
  tmp &= 0xFF;
}else{
  if((tmp & 0x00FF0000) != MASK2){
    return;
  }
  tmp >>= 24;
}
s = tmp;
```

備考 1. 削減量は，個々のケースにより異なります。

2. ソースを変更した結果，出力命令が減少し，高速化もされる場合があります。

3. ソース変更を行う際には，次のような点に注意してください。

- ソース変更によりレジスタの使用状況が変わるため，意図しない箇所において，それまで最適化されずに残っていたレジスタ転送が削除されたり，逆に，最適化が効かなくなって冗長なレジスタ転送が残ったりする可能性があります。
- テンポラリ変数を追加することにより，新たなレジスタ変数用レジスタが使用されるようになり，それに伴い，関数の入口／出口でそのレジスタの退避／復帰が追加されることがあります。

(2) ループ処理のループ回数

以下の例のように，実行回数が少なく，かつ，ループ・ボディが小さい場合，展開した方がサイズが小さくなることがあります。

この場合，実行速度も向上します。

変更前	変更後

for(i = 0; i < 4; i++){
  array[i] = 0;
}
	long *p;
	  :
	p = array;
	*p = 0;
	*(p + 1) = 0;
	*(p + 2) = 0;
	*(p + 3) = 0;

備考 1. 削減量は，個々のケースにより異なります。

2. ソースを変更した結果，出力命令が減少し，高速化もされる場合があります。

3. ソース変更を行う際には，次のような点に注意してください。

- ソース変更によりレジスタの使用状況が変わるため，意図しない箇所において，それまで最適化されずに残っていたレジスタ転送が削除されたり，逆に，最適化が効かなくなって冗長なレジスタ転送が残ったりする可能性があります。
- テンポラリ変数を追加することにより，新たなレジスタ変数用レジスタが使用されるようになり，それに伴い，関数の入口／出口でそのレジスタの退避／復帰が追加されることがあります。

（3）auto 変数の初期化

関数内で auto 変数が初期化されずに使用されている場合，その変数がレジスタに割り付けられずにメモリのまま残るため，コード・サイズが大きくなることがあります。

以下の例では，switch のいずれのケースにも該当しない場合に変数 a が初期化されず，return 文で参照されることになります。

実際には必ずいずれかのケースに該当するとしても，C コンパイラがレジスタ割り付けにおいて，プログラムを解析する際には分からないため，初期化されない場合があるとみなされます。

このような場合に CA850 のレジスタ割り付けでは割り付けられません。

そこで，初期化を追加することにより，レジスタに割り付けられるようになり，コード・サイズが削減されます。

変更前

```
int func(int x) {
  int a;
  switch(x){
    case  0:
      a = VAL0;
      break;
    case  1:
      a = VAL1;
  }
  return(a);
}
```

変更後

```
int func(int x) {
  int a = 0;
  switch(x){
    case  0:
      a = VAL0;
      break;
    case  1:
      a = VAL1;
  }
  return(a);
}
```

備考 1. 削減量は，個々のケースにより異なります。

2. ソースを変更した結果，出力命令が減少し，高速化もされる場合があります。

3. ソース変更を行う際には，次のような点に注意してください。

- ソース変更によりレジスタの使用状況が変わるため，意図しない箇所において，それまで最適化されずに残っていたレジスタ転送が削除されたり，逆に，最適化が効かなくなって冗長なレジスタ転送が残ったりする可能性があります。
- テンポラリ変数を追加することにより，新たなレジスタ変数用レジスタが使用されるようになり，それに伴い，関数の入口／出口でそのレジスタの退避／復帰が追加されることがあります。

（4）switch 文

CA850 は switch 文に対して，case ラベルの個数が 4 個以上でかつラベルの値の下限と上限の差が case の数の 3 倍までであれば，テーブル分岐形式のコードを生成します。

この場合，case の数がおよそ 16 個以下（ただし，この数は switch の式の形式やラベルの値の分布によって異なります）ならば，等価な if-else 文に変更し，比較命令と分岐命令のならびにした方が，コード・サイズが小さくなります。

なお，switch の式が外部変数参照や複雑な式の場合は，いったんテンポラリ変数に値を代入して，if の式ではそのテンポラリを参照するように変更する必要があります。

以下の例では x は auto 変数であるものとします。

変更前

```
switch(x){
  case  VAL0:
    return(RETVAL0);
  case  VAL1:
    return(RETVAL1);
  case  VAL2:
    return(RETVAL2);
  case  VAL3:
    return(RETVAL3);
  case  VAL4:
    return(RETVAL4);
  case  VAL5:
    return(RETVAL5);
}
```

変更後

```
if(x == VAL0)
  return(RETVAL0);
else if(x == VAL1)
  return(RETVAL1);
else if(x == VAL2)
  return(RETVAL2);
else if(x == VAL3)
  return(RETVAL3);
else if(x == VAL4)
  return(RETVAL4);
else if(x == VAL5)
  return(RETVAL5);
```

備考 1. 削減量は，個々のケースにより異なります。

2. ソースを変更した結果，出力命令が減少し，高速化もされる場合があります。

3. ソース変更を行う際には，次のような点に注意してください。

- ソース変更によりレジスタの使用状況が変わるため，意図しない箇所において，それまで最適化されずに残っていたレジスタ転送が削除されたり，逆に，最適化が効かなくなって冗長なレジスタ転送が残ったりする可能性があります。
- テンポラリ変数を追加することにより，新たなレジスタ変数用レジスタが使用されるようになり，それに伴い，関数の入口／出口でそのレジスタの退避／復帰が追加されることがあります。

4. CA850 では，-Xcase オプションにより，switch 文の展開コードを指定することが可能です。

- -Xcase=ifelse

  case 文のならびに沿った if-else 文と同じ形で出力します。

- -Xcase=binary

  バイナリサーチ形式で出力します。

- -Xcase=table

  テーブルジャンプ方式で出力します。

(5) if 文

if-else の組み合わせにおいて，複数のケースで同じ処理を実行する場合，別の条件を用いてその「複数のケース」を 1 つにまとめられるのであればまとめます。

これにより，冗長な部分を削除します。

以下の例で，「x の初期値が 0 で，s および t の値は 0 または 1 のいずれかである」という条件が揃っている場合，次のように変形することが可能です。

変更前	変更後

`if(!s){`
`  if(t){`
`  x = 1;`
`  }`
`}else{`
`  if(!t){`
`  x = 1;`
`  }`
`}`
`if(x){`
`  if((++u) >= v){`
`  u = 0;`
`  }else{`
`  x = 0;`
`  }`
`}`
	`if((s ^ t)){`
	`  if((++u) >= v){`
	`  u = 0;`
	`  x = 1;`
	`  }`
	`}`

備考 1. 削減量は，個々のケースにより異なります。

2. ソースを変更した結果，出力命令が減少し，高速化もされる場合があります。

3. ソース変更を行う際には，次のような点に注意してください。

- ソース変更によりレジスタの使用状況が変わるため，意図しない箇所において，それまで最適化されずに残っていたレジスタ転送が削除されたり，逆に，最適化が効かなくなって冗長なレジスタ転送が残ったりする可能性があります。
- テンポラリ変数を追加することにより，新たなレジスタ変数用レジスタが使用されるようになり，それに伴い，関数の入口／出口でそのレジスタの退避／復帰が追加されることがあります。

代入文の直後で，その代入された値が参照されている場合，参照されている箇所を代入文で置換して 1 つにまとめます。

これにより，余分なレジスタ転送が削除されてコード・サイズが削減される可能性があります。

ただし，多くの場合では C コンパイラの最適化によって冗長なレジスタ転送が削除されているため，コード・サイズは変わりません。

変更前	変更後

`--s;`
`if(s == 0){`
`  :`
`}`
	`if(--s == 0)){`
	`  :`
	`}`

備考 1. 削減量は，個々のケースにより異なります。

2. ソースを変更した結果，出力命令が減少し，高速化もされる場合があります。

3. ソース変更を行う際には，次のような点に注意してください。

- ソース変更によりレジスタの使用状況が変わるため，意図しない箇所において，それまで最適化されずに残っていたレジスタ転送が削除されたり，逆に，最適化が効かなくなって冗長なレジスタ転送が残ったりする可能性があります。
- テンポラリ変数を追加することにより，新たなレジスタ変数用レジスタが使用されるようになり，それに伴い，関数の入口／出口でそのレジスタの退避／復帰が追加されることがあります。

（6）if-else 文

以下の例のように，if-else 文の分岐先のおのおのが同じ変数へ異なる値を代入する文のみを含む場合，一方を if 文の前に移動することにより，else のブロックが削除されて if のブロックから else のブロックへの後へのジャンプ命令が減るため，コード・サイズが削減される可能性があります。

変更前	変更後

`if(x == 10){`
`  s = 1;`
`}else{`
`  s = 0;`
`}`
	`s = 0;`
	`if(x == 10){`
	`  s = 1;`
	`}`

備考 1. 削減量は，個々のケースにより異なります。

2. ソースを変更した結果，出力命令が減少し，高速化もされる場合があります。

3. ソース変更を行う際には，次のような点に注意してください。

- ソース変更によりレジスタの使用状況が変わるため，意図しない箇所において，それまで最適化されずに残っていたレジスタ転送が削除されたり，逆に，最適化が効かなくなって冗長なレジスタ転送が残ったりする可能性があります。
- テンポラリ変数を追加することにより，新たなレジスタ変数用レジスタが使用されるようになり，それに伴い，関数の入口／出口でそのレジスタの退避／復帰が追加されることがあります。

以下の例のように，if-else 文の分岐先のおのおのが return 文のみを含み，その戻り値が分岐条件の結果そのものである場合，分岐条件の式の値を返すようして，if-else 文を削除します。

変更前	変更後

`if(s1 == s2){`
`  return(1);`
`}`
`return(0);`
	`return(s1 == s2);`

備考 1. 削減量は，個々のケースにより異なります。

2. ソースを変更した結果，出力命令が減少し，高速化もされる場合があります。

3. ソース変更を行う際には，次のような点に注意してください。

- ソース変更によりレジスタの使用状況が変わるため，意図しない箇所において，それまで最適化されずに残っていたレジスタ転送が削除されたり，逆に，最適化が効かなくなって冗長なレジスタ転送が残ったりする可能性があります。
- テンポラリ変数を追加することにより，新たなレジスタ変数用レジスタが使用されるようになり，それに伴い，関数の入口／出口でそのレジスタの退避／復帰が追加されることがあります。

分岐後のおのおので，同一関数に対し異なる引き数を用いて呼び出しを行っている場合，関数呼び出しを合流後に移動可能であるならば移動します。

このとき，元々の呼び出しの箇所では，その異なる引き数をテンポラリ変数へ代入し，呼び出しにおいては，そのテンポラリ変数を引き数として用います。

変更前	変更後

if(s){
  :
  func(0, 1, 2);
}else{
  :
  func(0, 1, 3);
}
	int tmp;
	if(s){
	  :
	  tmp = 2;
	}else{
	  :
	  tmp = 3;
	}
	func(0, 1, tmp);

備考 1. 削減量は，個々のケースにより異なります。

2. ソースを変更した結果，出力命令が減少し，高速化もされる場合があります。

3. ソース変更を行う際には，次のような点に注意してください。

- ソース変更によりレジスタの使用状況が変わるため，意図しない箇所において，それまで最適化されずに残っていたレジスタ転送が削除されたり，逆に，最適化が効かなくなって冗長なレジスタ転送が残ったりする可能性があります。
- テンポラリ変数を追加することにより，新たなレジスタ変数用レジスタが使用されるようになり，それに伴い，関数の入口／出口でそのレジスタの退避／復帰が追加されることがあります。

分岐後のおのおのにおいて，同一の代入文や関数呼び出しが存在する場合には，分岐前に移動可能であれば移動します。

その文の評価結果が参照されている場合，いったんテンポラリ変数へ代入し，そのテンポラリを参照するようにします。

以下の例は関数呼び出しの場合です。

変更前	変更後

`if(x >= 0){`
`  if(x > func(0, 1, 2)){`
`  :`
`  }`
`}else{`
`  if(x < -func(0, 1, 2)){`
`  :`
`  }`
`}`
	`long  tmp;`
	`tmp = func(0, 1, 2);`
	`if(x >= 0){`
	`  if(x > tmp){`
	`  :`
	`  }`
	`}else{`
	`  if(x < -tmp){`
	`  :`
	`  }`
	`}`

備考 1. 削減量は，個々のケースにより異なります。

2. ソースを変更した結果，出力命令が減少し，高速化もされる場合があります。

3. ソース変更を行う際には，次のような点に注意してください。

- ソース変更によりレジスタの使用状況が変わるため，意図しない箇所において，それまで最適化されずに残っていたレジスタ転送が削除されたり，逆に，最適化が効かなくなって冗長なレジスタ転送が残ったりする可能性があります。
- テンポラリ変数を追加することにより，新たなレジスタ変数用レジスタが使用されるようになり，それに伴い，関数の入口／出口でそのレジスタの退避／復帰が追加されることがあります。

分岐後のおのおのにおいて，同一の代入文や関数呼び出しが存在する場合，分岐前に移動することが不可能であり，かつ，合流後に移動することは可能であるならば，合流後に移動します。

以下の例は代入文の場合です。

変更前	変更後

`if(tmp & MASK){`
`  :`
`  j++;`
`}else{`
`  :`
`  j++;`
`}`
	`if(tmp & MASK){`
	`  :`
	`}else{`
	`  :`
	`}`
	`j++;`

備考 1. 削減量は，個々のケースにより異なります。

2. ソースを変更した結果，出力命令が減少し，高速化もされる場合があります。

3. ソース変更を行う際には，次のような点に注意してください。

- ソース変更によりレジスタの使用状況が変わるため，意図しない箇所において，それまで最適化されずに残っていたレジスタ転送が削除されたり，逆に，最適化が効かなくなって冗長なレジスタ転送が残ったりする可能性があります。
- テンポラリ変数を追加することにより，新たなレジスタ変数用レジスタが使用されるようになり，それに伴い，関数の入口／出口でそのレジスタの退避／復帰が追加されることがあります。

（7）switch/if-else 文

以下の例のように，switch 文や if-else 文の分岐先のおのおので，同じ外部変数へ異なる値を代入する場合，おのおのの箇所ではいったんテンポラリ変数へ代入し，再び合流したあとにテンポラリ変数から元の外部変数へ代入を行うことにより，コード・サイズが削減される可能性があります。

これは，外部変数はレジスタに割り付けられることが少ないため，外部変数への代入はメモリへのストア命令（4 バイト）となる一方，テンポラリ変数への代入は，多くの場合，レジスタ転送（2 バイト）になるためです。

以下の例で s は外部変数であるとします。

変更前	変更後

switch(x){
  case 0:
  s = 0;
  break;
  case 1:
  s = 0x5555;
  break;
  case 2:
  s = 0xAAAA;
  break;
  case 3:
  s = 0xFFFF;
}
	int tmp;
	  :
	if(x == 0){
	  tmp = 0;
	}else if (x == 1){
	  tmp = 0x5555;
	}else if(x == 2){
	  tmp = 0xAAAA;
	}else if(x == 3) {
	  tmp = 0xFFFF;
	}else{
	  goto label;
	}
	s = tmp;
	label:

備考 1. 削減量は，個々のケースにより異なります。

2. ソースを変更した結果，出力命令が減少し，高速化もされる場合があります。

3. ソース変更を行う際には，次のような点に注意してください。

- ソース変更によりレジスタの使用状況が変わるため，意図しない箇所において，それまで最適化されずに残っていたレジスタ転送が削除されたり，逆に，最適化が効かなくなって冗長なレジスタ転送が残ったりする可能性があります。
- テンポラリ変数を追加することにより，新たなレジスタ変数用レジスタが使用されるようになり，それに伴い，関数の入口／出口でそのレジスタの退避／復帰が追加されることがあります。

（8）for/while 文

CA850 は，for や while のように最初に条件判断の式のあるループに対して，次のように条件判断の式を 2 回生成します。

このようなループの変形は，C コンパイラの最初のフェーズであるフロントエンド（構文解析部）によって行われますが，これはその後の最適化によって最初の条件判断が削除されることが多く，実行速度向上の点ではこのように変形するのが有利であるためです。

ところが，削除されなかった場合，コード・サイズの面では冗長となります。

【for ループ】

変形前	変形後

for(文 1; 式 2; 文 3){
　ループ・ボディ
}
	文 1;
	　if(式 2){
	　　do{
	　　　ループ・ボディ
	　　　文 3;
	　　}while(式 2);
	}

【whlie ループ 】

変形前	変形後

while(式 1){
　ループ・ボディ
}
	if(式 1){
	　do{
	　　ループ・ボディ
	　}while(式 1);
	}

したがって，1 回目の条件判断の式が最適化により削除されない場合には，以下のように goto で構成されるループに変形することで，条件判断の式を 1 個に減らすことができます。

【for ループ】

```
  文 1;
loop_bgn:
  if(! 式 2) goto loop_end;
    ループ・ボディ
    文 3;
    goto  loop_bgn;
loop_end:
```

【whlie ループ 】

```
loop_bgn:
  if(! 式 1) goto loop_end;
    ループ・ボディ
    goto loop_bgn;
loop_end:
```

変更前	変更後

for(i = 0; i < s; ++i){
  array[i] = array[i+1];
}
	i = 0;
	bgn_loop:
	  if(i >= s) goto end_loop;
	  array[i] = array[i+1];
	  ++i;
	  goto bgn_loop;
	end_loop:

備考 1. 削減量は，個々のケースにより異なります。

2. ソースを変更した結果，出力命令が減少し，高速化もされる場合があります。

3. ソース変更を行う際には，次のような点に注意してください。

- ソース変更によりレジスタの使用状況が変わるため，意図しない箇所において，それまで最適化されずに残っていたレジスタ転送が削除されたり，逆に，最適化が効かなくなって冗長なレジスタ転送が残ったりする可能性があります。
- テンポラリ変数を追加することにより，新たなレジスタ変数用レジスタが使用されるようになり，それに伴い，関数の入口／出口でそのレジスタの退避／復帰が追加されることがあります。

(9) 戻り値のない関数

戻り値のない関数は，“void”と宣言します。

2. 7. 2 変数の定義方法で変数領域を削減する

この項では，変数の定義方法で変数領域を削減する方法について説明します。

(1) 変数の符号

V850 マイクロコントローラでは，バイト・データ，ハーフワード・データをメモリからレジスタにロードする場合，最上位ビットの値によりワード長へ符号拡張します。

このため，unsigned char，unsignd short 型データの演算では，上位ビットのマスク・コードが生成される場合があります（データがすでにレジスタ上にある場合の演算では生成されません）。

できるだけワード・データを使用するようにしてください。

また，バイト・データ，ハーフワード・データを使用する場合は，符号付きの型を使用するようにしてください。

備考 1. V850E では，unsigned のロード命令がサポートされています。
そのため，符号拡張は行われず，マスク・コードは生成されません。

2. ワード・データを用いることができず，マスク・コードが生成されてしまうプログラムの場合，マスク・レジスタ機能を利用することによってコード・サイズを削減することができます。

(2) 変数の型

ANSI-C の仕様により，短整数（(unsigned) short，(unsigned) char）型の変数は，演算時に int 型または unsigned int 型に拡張されるため，これらの変数を使用したプログラムに対しては（特にこれらの変数がレジスタに割り付けられた場合には），型変換命令が多く生成されます。

(unsigned) int 型にすればこの型変換が不要となるため，コード・サイズが削減されます。

特に，比較的レジスタに割り付けられやすいスタック変数は，できるだけ (unsigned) int 型を用いましょう。

変更前	変更後

`unsigned char i;`
`  :`
`for(i = 0; i < 4; i++){`
`  array[2 + i] = *(p + i);`
`}`
	`int i;`
	`  :`
	`for(i = 0; i < 4; i++){`
	`  array[2 + i] = *(p + i);`
	`}`

備考 1. 削減量は，個々のケースにより異なります。

2. ソースを変更した結果，出力命令が減少し，高速化もされる場合があります。

3. ソース変更を行う際には，次のような点に注意してください。

- ソース変更によりレジスタの使用状況が変わるため，意図しない箇所において，それまで最適化されずに残っていたレジスタ転送が削除されたり，逆に，最適化が効かなくなって冗長なレジスタ転送が残ったりする可能性があります。
- テンポラリ変数を追加することにより，新たなレジスタ変数用レジスタが使用されるようになり，それに伴い，関数の入口／出口でそのレジスタの退避／復帰が追加されることがあります。この場合，退避／復帰のコード分（8 バイト）だけコード・サイズが増加します。

(3) 自動変数への代入と参照

以下の例のように，スタック変数に値が代入位置と参照位置が離れている場合，その間，レジスタが占有されて他の変数がレジスタに割り付けられる機会が減ることになります。

このような場合には，実際に参照する直前に代入を行うように変更することにより，他の変数のレジスタ割り付けの機会が増えてメモリ・アクセスが減り，コード・サイズが小さくなります。

変更前

```
int i = 0, j = 0, k = 0, m = 0;
/* この間，関数呼び出しあり */
/* これらの変数の使用はなし */
while((k & 0xFF) != 0xFF){
  k = s1;
  if(k & MASK){
    if(m != 1){
      s2 += 2;
      m = 1;
      array[15+i+j] = 0xFF;
      j++;
    }
  }
}
  :
```

変更後

```
int i, j, k, m;
  :
i = 0;
j = 0;
k = 0;
m = 0;
while((k & 0xFF) != 0xFF){
  k = s1;
  if(k & MASK){
    if(m != 1){
      s2 += 2;
      m = 1;
      array[15+i+j] = 0xFF;
      j++;
    }
  }
}
  :
```

備考 1. 削減量は，個々のケースにより異なります。

2. ソースを変更した結果，出力命令が減少し，高速化もされる場合があります。

3. ソース変更を行う際には，次のような点に注意してください。

- ソース変更によりレジスタの使用状況が変わるため，意図しない箇所において，それまで最適化されずに残っていたレジスタ転送が削除されたり，逆に，最適化が効かなくなって冗長なレジスタ転送が残ったりする可能性があります。
- テンポラリ変数を追加することにより，新たなレジスタ変数用レジスタが使用されるようになり，それに伴い，関数の入口／出口でそのレジスタの退避／復帰が追加されることがあります。この場合，退避／復帰のコード分（8 バイト）だけコード・サイズが増加します。

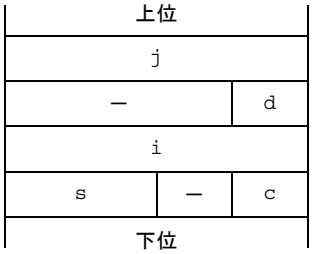
(4) 変数の型と定義順

定義は，データ長の長いものからまとめて行ってください。

V850 マイクロコントローラでは，int 型等のワード・データはワード境界，short 型等のハーフワード・データはハーフワード境界に整列している必要があります。

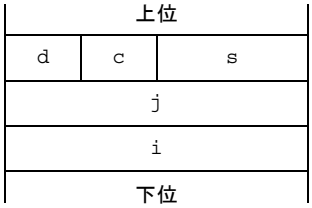
このため，次のようなソースに対しては整列のためのパディング領域が発生します。

```
char  c = 'c';
short s = 0;
int   i = 1;
char  d = 'b';
int   j = 2;
```



このようなパディング領域の発生を防ぐため，変数や構造体メンバの定義では，データ長の長いものからまとめて宣言してください。

```
int   i = 1;
int   j = 2;
short s = 0;
char  c = 'a';
char  d = 'b';
```



2. 8 処理の高速化

この節では，処理の高速化について説明します。

2. 8. 1 記述方法で処理を高速化する

この項では，記述方法で処理を高速化する方法について説明します。

(1) ループ処理のポインタ

以下の例のように，ループの制御を行う変数を帰納変数（誘導変数）と呼びます。

「帰納変数の削除」とは，ループの制御を他の変数を用いるように変更することで帰納変数を削除する最適化です。

この最適化は CA850 でも実装されていますが，適用される条件が限られるため，すべての場合を最適化することはできません。

以下のようにプログラムを修正することにより，この最適化を“手動”で行うことができます。

以下の例では，テンポラリ変数 （ポインタ）p を用いることによって，i を削除します。

変更前	変更後

`int i;`
`for(i = 0; *(table + i) != NULL; ++i){`
`  if((*(table + i) & 0xFF) == x){`
`  return(*(table + i) & 0xFF00);`
`  }`
`}`
	`const unsigned short  *p;`
	`for(p = table; *p != NULL; ++p){`
	`  if((*p & 0xFF) == x){`
	`  return(*p & 0xFF00);`
	`  }`
	`}`

備考 1. 削減量は，個々のケースにより異なります。

2. ソースを変更した結果，出力命令が減少し，高速化もされる場合があります。

3. ソース変更を行う際には，次のような点に注意してください。

- ソース変更によりレジスタの使用状況が変わるため，意図しない箇所において，それまで最適化されずに残っていたレジスタ転送が削除されたり，逆に，最適化が効かなくなって冗長なレジスタ転送が残ったりする可能性があります。
- テンポラリ変数を追加することにより，新たなレジスタ変数用レジスタが使用されるようになり，それに伴い，関数の入口／出口でそのレジスタの退避／復帰が追加されることがあります。この場合，退避／復帰のコード分（8 バイト）だけコード・サイズが増加します。

(2) auto 変数の宣言

auto 変数は，10 個以下（なるべく 6 ～ 7 個くらい）にしてください。

auto 変数はレジスタに割り当てられます。

CA850 では，1 つの関数内で，作業用レジスタとして 10 本，レジスタ変数用レジスタとして 10 本，合計 20 本のレジスタを変数用として使用可能です。(32 レジスタ・モードの場合)

1 つの関数内の処理が，時間のかかる処理であれば，auto 変数を多く使用することを推奨します。

あまり時間のかからない処理であれば，できるだけ作業用レジスタの 10 本のみを使用するようにしてください。

レジスタ変数用レジスタは，退避／復帰のオーバヘッドがかかります。

レジスタ変数を使用するかしないかは，C コンパイラが自動的に判断します。

したがって，auto 変数は 6 ～ 7 個とし，C コンパイラが作業用として使用可能なレジスタを 3 ～ 4 本残してください。

(3) 関数の引数

引き数用のレジスタは，r6 ～ r9 の 4 個です。

引き数が 5 個以上の場合は，5 個め以降はスタック領域を使用します。

したがって，引き数はなるべく 4 個以内にしてください。

どうしても 5 個以上になる場合，構造体のポインタで引き数を渡すようにします。

2.9 コンパイラとアセンブラの相互参照

この節では，コンパイラとアセンブラの相互参照について説明します。

2.9.1 変数を相互参照する

この項では，変数を相互参照する方法について説明します。

(1) C 言語で定義した変数を参照する

C 言語プログラム中で定義した外部変数をアセンブリ言語ルーチン中で参照する場合，extern 宣言します。

アセンブリ言語ルーチン中では，定義した変数の先頭に“_”（アンダースコア）を付けます。

例 C ソース

```
extern void subf ( void );
char c = 0 ;
int i = 0 ;
void main ( void ) {
    subf ( );
}
```

例 アセンブラ・ソース

```
    .globl  _subf
    .extern _c
    .extern _i
    .text
    .align  4
_subf :
    mov     4, r10
    st.b    r10, $_c
    mov     7, r10
    st.w    r10, $_i
    jmp [lp]
```

(2) アセンブリ 言語で定義した変数を参照する

アセンブリ 言語プログラム中で定義した外部変数を C 言語ルーチン中で参照する場合，extern 宣言します。

アセンブリ言語ルーチン中で定義する変数の先頭に“_”（アンダースコア）を付けます。

例 C ソース

```
extern char c ;
extern int i ;
void subf ( void ) {
    c = 'A' ;
    i = 4 ;
}
```

例 アセンブラ・ソース

```
    .globl _c
    .globl _i
    .sbss
    .lcomm _i, 4, 4
    .lcomm _c, 1, 1
```

2. 9. 2 関数を相互参照する

この項では，関数を相互参照する方法について説明します。

(1) C 言語で定義した関数を参照する

アセンブリ言語ルーチンから C 言語により記述された関数を呼び出すときの注意点について説明します。

- スタック・フレーム

「スタック・ポインタ（SP）が，常にスタック・フレームの最下位アドレスを指している」ことを想定したコードを出力します。そのため，アセンブラ関数から C 言語関数へ分岐する前に，スタック領域中の未使用領域の上位アドレスを指すように SP を設定してください。

- 作業用レジスタ

C 言語関数呼び出しの前後において，レジスタ変数用レジスタの値は保持しますが，作業用レジスタの値は保持しません。そのため，保持しなくてはならない値を作業用レジスタに割り当てたままにしないでください。

- アセンブラ関数への戻り先アドレス

「関数の戻り先アドレスは“リンク・ポインタ lp（r31）”に格納される」ことを想定したコードを生成します。C 言語関数へ分岐するとき，lp に関数の戻り先アドレスを格納する必要あります。

(2) アセンブリ 言語で定義した関数を参照する

C 言語により記述された関数からアセンブリ言語ルーチンを呼び出すときの注意点について説明します。

- 識別子

先頭に“_（アンダースコア）”を付けた名前になります。

- スタック・フレーム

「スタック・ポインタ（SP）が，常にスタック・フレームの最下位アドレスを指している」ことを想定したコードを出力します。そのため，C 言語ソースからアセンブラ関数へ分岐後は，アセンブラ関数内では，SP の指すアドレスよりも下位のアドレス領域は自由に使用することができます。逆に上位のアドレス領域の内容を変更した場合，C 言語関数で使用していた領域を破壊することにつながり，以降の動作を保証できませんので注意が必要です。上記を回避するためには，アセンブラ関数の先頭で SP を変更してからスタックを使用してください。

ただし，その際は呼び出しの前後で SP の値が保持されるようにしてください。

- レジスタ変数用レジスタ

アセンブラ関数内でレジスタ変数用レジスタを使用する場合は，アセンブラ関数の呼び出し前後でレジスタ値が保持されるようにしてください（使用前にレジスタ変数用レジスタの値を退避し，使用後は復帰してください）。

- C 言語関数への戻り先アドレス

「関数の戻り先アドレスは“リンク・ポインタ lp（r31）”に格納される」ことを想定したコードを生成します。アセンブラ関数へ分岐するとき，lp に関数の戻り先アドレスが格納されているので，C 言語関数へ戻るときは“jmp [lp]”を実行してください。

第 3 章　コンパイラ言語仕様

この章では，CA850 がサポートする言語仕様について説明します。

3. 1　基本言語仕様

CA850 は，ANSI 規格で規定された言語仕様をサポートしていますが，その中には処理系定義として規定されている項目があります。ここでは，V850 マイクロコントローラの処理系に依存した項目の言語仕様について説明します。

また，厳密な ANSI 準拠処理のオプションを指定した場合と指定しない場合の差分についても説明します。

なお，CA850 で独自に追加されている拡張言語仕様については，「3. 2　拡張言語仕様」を参照してください。

3. 1. 1　処理系依存

この節では，ANSI 規格における処理系依存項目について説明します。

(1) データ型とサイズ

ワード（4 バイト）中のバイト順序は，“下位から上位”です。また，符号付き整数は，2 の補数で表現します。最上位ビットには，符号（正，または 0 の場合 0，負の場合 1）が入ります。

- 1 バイト中のビット数は，8 ビットとします。
- オブジェクト中のバイト数，バイト順序，符号化は，次のように規定します。

表 3―1　データ型とサイズ

データ型	サイズ
char 型	1 バイト
short 型	2 バイト
int，long，float，double 型	4 バイト
ポインタ	unsigned int と同じ

(2) 翻訳段階

ANSI 規格では，翻訳における構文規則間の優先順位を 8 つの翻訳段階（翻訳フェーズ）に規定しています。3 段階目の“前処理字句と空白類文字の並びへの分割”で処理系定義となっている，“改行文字以外の空白類文字の空でない並び”は 1 つに置き換えられずそのまま保持されます。

(3) 診断メッセージ

何らかの構文規則違反，および制約違反を含む翻訳単位に対して，ソース・ファイル名，行番号（特定可能な場合のみ）を含むエラー・メッセージを出力します。なお，エラー・メッセージの書式は“警告”，“致命的エラー”，“その他のエラー”の 3 種類に区別されます。

(4) フリー・スタンディング環境

(a) フリー・スタンディング環境注においては，プログラム開始処理時に呼び出される関数の名前，および型は特に規定しません。したがって，ユーザ・オウン・コーディング，またはターゲット・システムに依存します。

注　オペレーティング・システムの機能を使用せずに C 言語ソース・プログラムを実行する環境のこと。
ANSI 規格では，実行環境にはフリー・スタンディング環境とホスト環境の 2 つが規定されていますが，CA850 では，ホスト環境は現在提供されていません。

(b) フリー・スタンディング環境におけるプログラム終了処理の効果は，特に規定しません。したがって，ユーザ・オウン・コーディング，またはターゲット・システムに依存します。

(5) プログラムの実行

対話型装置の構成については，特に規定しません。
したがって，ユーザ・オウン・コーディング，またはターゲット・システムに依存します。

(6) 文字集合

実行環境文字集合の要素の値は，ASCII コードです。

(7) 多バイト文字

多バイト文字は，文字定数ではサポートしていません。
ただし，コメントと文字列における日本語記述はサポートしています。

(8) 文字表示の意味

拡張表記の値は，次のように規定します。

表 3―2　拡張表記と意味

拡張表記	値（ASCII）	意味
\a	07	アラート（警告音）
\b	08	バックスペース
\f	0C	フォーム・フィード（改ページ）
\n	0A	ニュー・ライン（改行）
\r	0D	キャリッジ・リターン（復帰）
\t	09	水平タブ
\v	0B	垂直タブ

(9) 翻訳限界

次に，翻訳に際しての限界値を示します。

なお，* の付いている値は保証値であり，それ以上の値でも可能な場合もありますが，動作は保証されません。

表 3―3 翻訳限界値

内容	限界値
複文，繰り返し制御構造，および選択制御構造の入れ子のレベル数 （ただし，“case”のラベル数に依存）	127
条件組み込みの入れ子のレベル数	255
1 つの宣言中の 1 つの算術型，構造体型，共用体型，または不完全型を修飾する（任意の組み合わせの）ポインタ，配列，および関数宣言子の数	16
完全な宣言子の中の，かっこで囲まれた宣言子の入れ子のレベル数	255*
完全な式の中の，かっこで囲まれた式の入れ子のレベル数	255*
マクロ名における有効先頭文字数	1023
外部識別子における有効先頭文字数	1022
内部識別子における有効先頭文字数	1023
1 つの翻訳単位内の外部識別子，および 1 つの基本ブロック内で宣言されるブロック有効範囲をもつ識別子の数	4095*
1 つの翻訳単位内で同時に定義されるマクロ識別子の数**注**	2047
1 つの関数定義内の仮引数，および 1 つの関数呼び出し内の実引数の数	255
1 つのマクロ定義内の仮引数の数	127
1 つのマクロ呼び出し内の実引数の数	127
1 つの論理ソース行内の文字数	32768
連結後の 1 つの文字列定数，またはワイド文字列定数内の文字数	32766
インクルード・ファイルに対する入れ子のレベル数	50
1 つの“switch”文に対する“case”ラベルの数 （ネストされている場合，それも含める）	1025
単一構造体，または単一共用体内のメンバ数	1023*
単一列挙型における列挙型定数の数	1023*
単一構造体宣言の並び内の，構造体，または共用体定義の入れ子のレベル数	63*

注 マクロ識別子の上限は，C コンパイラ・オプション（-Xm）で変更することができます。

（10）数量的限界

（a）汎整数型の限界値（limits.h ファイル）

汎整数型（char 型，符号付き／符号なし整数型，および列挙型）で表現できる値の各種限界値を limits.h ファイルに定義しています。

なお，多バイト文字はサポートしていないため，MB_LEN_MAX は該当する限界値を持ちません。そこで，MB_LEN_MAX には 1 として，定義のみ行っています。

また，CA850 の -Xchar=unsigned オプション（単なる char 型の符号を指定）が指定された場合，CHAR_MIN は 0，CHAR_MAX は UCHAR_MAX と同値となります。

次に，limits.h ファイルで定義されている各種限界値を示します。

表 3―4 汎整数型の各種限界値（limits.h ファイル）

名前	値	意味
CHAR_BIT	+8	ビット・フィールドではない最小のオブジェクトのビット数（＝1 バイト）
SCHAR_MIN	-128	signed char 型の最小値
SCHAR_MAX	+127	signed char 型の最大値
UCHAR_MAX	+255	unsigned char 型の最大値
CHAR_MIN	-128	char 型の最小値
CHAR_MAX	+127	char 型の最大値
SHRT_MIN	-32768	short int 型の最小値
SHRT_MAX	+32767	short int 型の最大値
USHRT_MAX	+65535	unsigned short int 型の最大値
INT_MIN	-2147483648	int 型の最小値
INT_MAX	+2147483647	int 型の最大値
UINT_MAX	+4294967295	unsigned int 型の最大値
LONG_MIN	-2147483648	long int 型の最小値
LONG_MAX	+2147483647	long int 型の最大値
ULONG_MAX	+4294967295	unsigned long int 型の最大値

（b）浮動小数点型の各種限界値（float.h ファイル）

浮動小数点型の特性に関する各種限界値を float.h ファイルに定義しています。

次に，float.h ファイルで定義されている各種限界値を示します。

表 3―5 浮動小数点型の各種限界値の定義（float.h ファイル）

名前	値	意味
FLT_ROUNDS	+1	浮動小数点加算に対する丸めのモード V850 マイクロコントローラでは，1 （もっとも近い方向へ丸める）とします。
FLT_RADIX	+2	指数表現の基数（b）
FLT_MANT_DIG	+24	浮動小数点仮数部における FLT_RADIX を底とする数字の数（p）
DBL_MANT_DIG		
LDBL_MANT_DIG		
FLT_DIG	+6	q 桁の 10 進数の浮動小数点数を基数 b の p 桁をもつ浮動小数点数に丸めることができ，再び変更なしに q 桁の 10 進数値に戻すことができるような 10 進数の桁数**注 1** （q）
DBL_DIG		
LDBL_DIG		
FLT_MIN_EXP	-125	FLT_RADIX をその値から 1 引いた値でべき乗したとき，正規化された浮動小数点数となるような最小の負の整数 （e_{min}）
DBL_MIN_EXP		
LDBL_MIN_EXP		
FLT_MIN_10_EXP	-37	10 をその値でべき乗したとき，正規化された浮動小数点数の範囲内になるような最小の負の整数 $\log_{10} b^{e_{min-1}}$
DBL_MIN_10_EXP		
LDBL_MIN_10_EXP		
FLT_MAX_EXP	+128	FLT_RADIX をその値から 1 引いた値でべき乗したとき，表現可能な有限浮動小数点数となるような最大の整数 （e_{max}）
DBL_MAX_EXP		
LDBL_MAX_EXP		
FLT_MAX_10_EXP	+38	表現可能な有限浮動小数点数の最大値 $(1 - b^{-p}) * b^{e_{max}}$
DBL_MAX_10_EXP		
LDBL_MAX_10_EXP		
FLT_MAX	3.40282347E + 38F	表現可能な有限浮動小数点数の最大値 $(1 - b^{-p}) * b^{e_{max}}$
DBL_MAX		
LDBL_MAX		
FLT_EPSILON	1.19209290E - 07F	指定された浮動小数点型で表現できる 1.0 と，1.0 より大きい最も小さい値との差異**注 2** b^{1-p}
DBL_EPSILON		
LDBL_EPSILON		

名前	値	意味
FLT_MIN	1.17549435E - 38F	正規化された正の浮動小数点数の最小値 $b^{e_{min}-1}$
DBL_MIN		
LDBL_MIN		

注 1. DBL_DIG，LDBL_DIG は，ANSI 規格では，10 以上となっていますが，V850 マイクロコントローラでは，double 型も long double 型も 32 ビットであるため 6 となります。

2. DBL_EPSILON と LDBL_EPSILON は，ANSI 規格では，1E - 9 以下となっていますが，V850 マイクロコントローラにおいては 1.19209290E - 07F となります。

（11）識別子

外部名は，最大 1022 文字で一意に識別できるようにしてください。

なお，英字の大文字と小文字は区別されます。

（12）char 型

型指定子（signed，unsigned）の付かない単なる char 型は，符号付き整数として扱います。

ただし，CA850 の -Xchar=unsigned オプションを指定することにより，符号なし整数として扱うこともできます。

ANSI 規格において要求されるソース・プログラムの文字集合に含まれないもの（エスケープ・シーケンス）は，char 型以外を char 型へ代入する場合と同様に，型変換して格納されます。

```
char    c = '\777';     /*c の値は -1 となる */
```

（13）浮動小数点定数

浮動小数点定数は，IEEE754 [注]に準拠しています。

注 IEEE：Institute of Electrical and Electronics Engineers（電気電信学会）の略称です。
また，IEEE754 とは，浮動小数点演算を扱うシステムにおいて，扱うデータ形式や数値範囲などの仕様の統一化を図った標準です。

（14）文字定数

（a）ソース・プログラムの文字集合と実行環境における文字集合は，基本的に両者とも ASCII コードで，同一の値をもつメンバと対応します。
ただし，ソース・プログラムにおける文字列には，日本語文字コードが使用できます（「（8）文字表示の意味」を参照）。

（b）2 つ以上の文字を含む整数文字定数の値は，最後の 1 文字が有効値となります。

（c）基本的な実行環境文字集合で表現されない文字やエスケープ・シーケンスを含む場合，次のようになります。

- 8 進数エスケープ・シーケンス，および 16 進数エスケープ・シーケンスは，その 8 進数表記，および 16 進数表記で示される値となります。

\777	511

- 単純エスケープ・シーケンスは，次のようになります。

\'	'
\"	"
\?	?
\\	\

- \a，\b，\f，\n，\r，\t，\v については，「（8）文字表示の意味」で示されている値と同値になります。

（d）多バイト文字の文字定数はサポートしていません。

（15）文字列

文字列中に日本語が記述できます。

デフォルトの文字コードは，シフト JIS となります。

入力ソース・ファイルの中の文字コードは，CA850 の -Xk オプションで選択できます。

ただし，n，または none を指定すると，文字コードは保証されません。

【オプション指定】

-Xk=[e | euc | n | none | s | sjis]

また，出力オブジェクト・ファイル中の文字コードは，CA850 の -Xkt オプションで変換できます。ただし，n，または none を指定すると，文字コードは変換されません。

【オプション指定】

-Xkt=[e | euc | n | none | s | sjis]

（16）ヘッダ・ファイル名

ヘッダ・ファイル名の 2 つの形式（< >，" "）内の列を，ヘッダ・ファイル，または外部ソース・ファイル名に反映する方法は，「（33）ヘッダ・ファイル取り込み」で規定します。

（17）コメント

コメント中に日本語が記述できます。文字コード，「（15）文字列」の場合と同じです。

（18）符号付き定数と符号なし定数

汎整数型の値がよりサイズの小さい符号付き整数に変換される場合，上位ビットを切り捨てて，ビット列イメージをコピーします。

また，符号なし整数が，対応する符号付き整数に変換される場合，内部表現は変化しません。

（19）浮動小数点と汎整数

汎整数型の値が浮動小数点型に型変換される際，型変換される値が，表現しうる値の範囲内にはあるが正確に表現することができない場合，その結果は，表現しうる最も近い値へ丸められます。

なお，結果がちょうど中央の値である場合には，偶数（仮数の最下位ビットが 0 のもの）に丸められます。

（20）double 型と float 型

CA850 では，double 型は float 型と同じ浮動小数点表現であり，32 ビット・データ（単精度）として扱われます。

（21）ビット単位の演算子における符号付き型

ビット単位の演算子における符号付き型に対する特性は，シフト演算子については，「（27）ビット単位のシフト演算子」の規定に準じます。

また，その他の演算子については，符号なしの値として（ビット・イメージのままで）計算するものとします。

（22）構造体と共用体のメンバ

共用体のメンバの値がそれと異なるメンバに格納される場合，整列条件に従って格納されるため，その共用体のあるメンバへのアクセスは，整列条件に従って行われます（「（6）構造体型」，および「（7）共用体型」を参照）。

ただし，共通の先頭メンバの並びを共有している構造体だけをメンバとして含んでいる共用体の場合，内部表現は同じであるため，どの構造体の共通の先頭メンバを参照しても同じになります。

（23）sizeof 演算子

“sizeof”演算子の結果は，「（1）データ型とサイズ」におけるオブジェクト中のバイトに関する規定に準じます。

なお，構造体と共用体については，パディング領域を含んだバイト数とします。

（24）キャスト演算子

ポインタを汎整数型に変換する場合，要求される変数のサイズは，int 型と同じサイズです。変換結果は，ビット列がそのまま保存されます。

また，任意の整数はポインタに型変換できますが，int 型よりも小さい整数の場合，結果はその型に従って拡張されます。

（25）乗除／剰余演算子

整数同士の除算で割り切れず，オペランドが負の値をもつ場合，“/”演算子の結果は，除数，または被除数のいずれか一方が負の場合は，代数的な商よりも大きい最小の整数となります。

ただし，どちらも負の場合は，代数的な商よりも小さい最大の整数となります。

また，オペランドが負の値をもつ場合，“%”演算子の結果の符号は第 1 オペランドの符号とします。

（26）加減演算子

同一配列の要素を指す 2 つのポインタが減算される場合，結果の型は int 型とし，サイズは 4 バイトとします。

（27）ビット単位のシフト演算子

“E1 >> E2”において，E1 が符号付きの型で負の値をもつ場合，算術シフトを行います。

（28）記憶域クラス指定子

記憶域クラス指定子“register”の宣言は，可能なかぎり高速にアクセスするために行いますが，必ずしも有効であるとはかぎりません。

（29）構造体と共用体指定子

（a）signed，unsigned の付かない単なる int 型ビット・フィールドは，符号付きとして扱い，最上位ビットは符号ビットとして扱います。ただし，CA850 の -Xbitfield オプション（単なる int 型ビット・フィールドの符号を指定）を指定することにより，符号なしとして扱うこともできます。

（b）ビット・フィールドを保持するために，十分な大きさの任意のアドレス付け可能な記憶域単位を割り付けることができますが，十分な領域がなかった場合，合わなかったビット・フィールドはフィールドの型の整列条件に合わせて次の単位に詰め込まれます。

（c）単位内のビット・フィールドの割り付け順序は下位から上位です。

（d）1 つの構造体，または共用体の非ビット・フィールドの各メンバは，次のように境界整列されます。

char，unsignd char 型，およびその配列	バイト境界
short，unsigned short 型，およびその配列	ハーフワード境界
その他（ポインタを含む）	ワード境界

（30）列挙型指定子

列挙型の型は，signed int 型とします。

ただし，-Xenum_type=string オプション指定時は，次のようになります。

char	char として扱う
uchar	unsigned char として扱う
short	short として扱う
ushort	unsigned short として扱う

（31）型修飾子

“volatile”修飾された型をもつデータへのアクセスは，データがマッピングされているアドレス（I/O ポートなど）に依存します。

（32）条件組み込み

（a）条件組み込みで指定される文字定数に対する値と，その他の式中に現れる文字定数の値とは等しくなります。

（b）単一文字の文字定数は，負の値を持たないようにしてください。

（33）ヘッダ・ファイル取り込み

（a）“#include < 文字列 >”という形式の前処理指示

“#include < 文字列 >”という形式の前処理指示は，“文字列”がフルパスのファイル名でない場合[注]，指定されたフォルダ（-I オプション）からヘッダ・ファイルを検索し，次に ca850 が置かれた bin フォルダからの相対パスでの .. \ inc850 フォルダを検索します。

なお，“<”と“>”の区切り記号の間に指定された文字列で一意に識別されるヘッダ・ファイルを探し出すと，そのヘッダ・ファイルの内容全体で置き換えます。

注 “\”と“/”の両者がフォルダの区切りとしてみなされます。

例

```
#include    <header.h>
```

検索順は，次のとおりです。

- -I で指定したフォルダ
- 標準のフォルダ

(b)“#include " 文字列 "”という形式の前処理指示

“#include " 文字列 "”という形式の前処理指示は，“文字列”がフルパスのファイル名でない場合，ソース・ファイルがあるフォルダからヘッダ・ファイルを検索し，次に，指定したフォルダ（-I オプション），最後に ca850 が置かれた bin フォルダからの相対パスでの .. \ inc850 フォルダを検索します。

なお，“ " ”“ " ”の区切り記号の間に指定された文字列で一意に識別されるヘッダ・ファイルを探し出すと，そのヘッダ・ファイルの内容全体で置き換えます。

例

```
#include    "header.h"
```

検索順は，次のとおりです。

- ソース・ファイルがあるフォルダ
- -I で指定したフォルダ
- 標準のフォルダ

(c)“#include 前処理字句列”という形式

“#include 前処理字句列”という形式において，前処理字句列が単一で < 文字列 >，または " 文字列 " の形式に置換されるマクロである場合にのみ，単一のヘッダ・ファイル名の前処理字句として扱われます。

(d)“#include < 文字列 >”という形式の前処理指示

（最終的に）区切られた列とヘッダ・ファイル名との間においては，列中の英文字の長さを判別し，

```
コンパイラ動作環境において有効なファイル名長までが有効
```

となります。ファイルを探すフォルダについては，上記の規定に準じます。

(34) #pragma 指令

CA850 では，次の #pragma 指令が指定できます。

(a) アセンブラ命令の記述

```
#pragma asm
    アセンブラ命令
#pragma endasm
```

C 言語中に，アセンブラ命令を記述することができます。

なお，記述方法についての詳細は「(4) アセンブラ命令の記述」を参照してください。

(b) インライン展開指定

```
#pragma inline  関数名 [, 関数名 , …]
```

インライン展開する関数を指定することができます。
なお，インライン展開についての詳細は「(8) インライン展開」を参照してください。

(c) デバイス種別指定

```
#pragma cpu     デバイス名
```

使用するデバイスの機種依存情報を定義したデバイス・ファイルを参照するように指定します。CA850のデバイス指定オプション（-cpu）と同じ機能です。C 言語ソース内にデバイスを定義したい場合に用います。

(d) データ／プログラムのメモリ割り当て

```
#pragma section セクション種別    [" セクション名 "] [begin | end]
#pragma text    [" セクション名 "] [ 関数名 ]
```

- section

変数を任意のセクションに割り当てます。
なお，配置方法についての詳細は「(1) データのセクション割り当て」を参照してください。

- text

任意の名称のテキスト・セクションに関数を指定できます。
なお，配置指定についての詳細は「(2) 関数のセクション割り当て」を参照してください。

(e) 周辺 I/O レジスタ名有効化指定

```
#pragma ioreg
```

周辺 I/O レジスタ名を用いて，デバイスの持つ周辺 I/O レジスタにアクセスします。周辺 I/O レジスタ名をそのまま用いてプログラミングする場合はこの #pragma 指令を指定する必要があります。

(f) 割り込み／例外ハンドラ指定

```
#pragma interrupt   割り込み要求名    関数名    [ 配置方法 ]
```

割り込み／例外処理ハンドラを C 言語で記述します。
なお，記述方法については「(c) 割り込み／例外ハンドラの記述方法」を参照してください。

（g）割り込み禁止関数指定

```
#pragma block_interrupt 関数名
```

関数全体を割り込み禁止にします。

（h）タスク指定

```
#pragma rtos_task　関数名
```

リアルタイム OS 上で動作するタスクを C 言語で記述します。
なお，記述方法についての詳細は「(a) タスクの記述」を参照してください。

（i）構造体パッキング指定

```
#pragma pack([1248])
```

構造体パッキングを指定します。数値はパッキング値，すなわちメンバのアライメント値を指定します。数値には 1，2，4，8 が指定できます。数値を指定しない場合，デフォルト（8）[注]となります。

注　本バージョンではアライメント値“4”と“8”は同じになります。

（35）あらかじめ定義されたマクロ名

以下に，サポートしているマクロ名を示します。
なお，“__”で終わらないマクロは，従来の C 言語仕様（K&R 仕様）のために提供しているものです。ANSI 規格に厳密な処理を行う場合，前後に“__”のある形式のマクロを利用するようにしてください。

表 3―6　サポートしているマクロ

マクロ名	定義
__LINE__	その時点でのソース行の行番号（10 進数）。
__FILE__	仮定されたソース・ファイルの名前（文字列定数）。
__DATE__	ソース・ファイルの翻訳日付（“Mmm dd yyyy”の形式をもつ文字列定数。ここで，月の名前は ANSI 規格で規定されている asctime 関数で生成されるもの（英字 3 文字の並びで最初の 1 文字のみ大文字）と同じもの。dd の最初の文字は値が 10 より小さい場合空白とします）。
__TIME__	ソース・ファイルの翻訳時間（asctime 関数で生成される時間と同じような“hh : mm : ss”の型式をもつ文字列定数）。
__STDC__	10 進定数 1（-ansi オプション指定時に定義）。[注]
__v800 __v800__	10 進定数 1。

マクロ名	定義
__v850 __v850__	10 進定数 1。
__v850e __v850e__	10 進定数 1（CA850 で，ターゲット・デバイスに V850E*x* を指定した場合に定義）。
__v850e2 __v850e2__	10 進定数 1（CA850 で，ターゲット・デバイスに V850E2 を指定した場合に定義）。
__CA850 __CA850__	10 進定数 1。
__CHAR_SIGNED__	10 進定数 1（-Xchar オプションで，符号つきを指定した場合，および -Xchar オプションを指定しない場合に定義）。
__CHAR_UNSIGNED__	10 進定数 1（-Xchar オプションで，符号なしを指定した場合に定義）。
__DOUBLE_IS_32BITS__	10 進定数 1。
_DOUBLE_IS_32BITS	10 進定数 1。
CPU マクロ	ターゲット CPU を示すマクロで 10 進定数 1。デバイス・ファイル中の「品種指定名」で示される文字列の先頭と末尾に“__”を付けたものが定義されます。
レジスタ・モード・マクロ	ターゲット CPU を示すマクロで 10 進定数 1。 レジスタ・モードと定義されるマクロは次のとおりです。 32 レジスタ・モード ：__reg32__ 26 レジスタ・モード ：__reg26__ 22 レジスタ・モード ：__reg22__

注 -ansi オプション指定時の処理については，「3. 1. 2 ansi オプション」を参照してください。

（36）特別なデータ型の定義

次に，stddef.h ファイルにおける NULL，size_t，ptrdiff_t の定義を示します。

表 3―7 NULL，size_t，ptrdiff_t の定義（stddef.h ファイル）

NULL ／ size_t ／ ptrdiff_t	定義
NULL	((void *) 0)
size_t	unsigned int
ptrdiff_t	int

3. 1. 2 ansi オプション

CA850 で -ansi オプションを指定した場合，ANSI 規格に厳密な処理が行われます。

次に，-ansi オプションを指定した場合と，指定しない場合の処理の違いを示します。

表 3―8 言語仕様に厳密な -ansi オプション指定時の処理

項目	-ansi 指定あり	-ansi 指定なし
トライグラフ系列	トライグラフ系列の置換を行います。	置換しません。
ビット・フィールド	ビット・フィールドに int 型以外の型を指定した場合，エラー[注 1]とします。	警告メッセージを出力し，許可します。
引数のスコープ	関数の引数と同名の自動変数を宣言した場合，二重定義のエラーとします。	警告メッセージを出力し，自動変数を有効とします。
ポインタの代入 1	汎整数型[注 2]変数へのポインタ型の数値の代入は，エラーとします。	警告メッセージを出力し，キャストして代入します。
ポインタの代入 2	異なる型を指すポインタ同士の代入は，エラーとします。	警告メッセージを出力し，許可します。
型変換	左辺値でない配列のポインタへの変換は，エラーとします。	警告メッセージを出力し，許可します。
比較演算子	算術型変数とポインタの比較は，エラーとします。	警告メッセージを出力し，許可します。
条件演算子	第 2 式と第 3 式がともに汎整数型，同じ構造体，同じ共用体，または代入先と同じ型へのポインタ型の数値のいずれでもない場合，エラーとします。	警告メッセージを出力し，キャストして代入します。
# 行番号	エラーとします。	“#line”行番号と同様に扱います。[注 3]
行の途中の # 文字	“#”文字が行の途中で現れた場合，エラーとします。	警告メッセージを出力し，許可します。
_asm	警告メッセージを出力し，関数呼び出しとして扱います。 ただし，__asm は有効とします。	アセンブラ挿入[注 4]として扱います。
__STDC__	値が 1 のマクロとして定義します。	定義しません。
2 進定数	“0b”，または“0B”と，その後ろに続く 1 個以上の“0”，または“1”の数字の並びをエラーとします。	“0b”，または“0B”と，その後ろに続く 1 個以上の“0”，または“1”の数字の並びを 2 進定数とします。

注 1. “E”で始まる通常のエラー。以下同じです。

2. char 型，符号付き／符号なし整数型，および列挙型です。

3. ANSI 規格を参照してください。

4. 「(4) アセンブラ命令の記述」を参照してください。

3. 1. 3　データの内部表現と領域

この項では，CA850 が扱うデータのそれぞれの型における，内部表現と値域について説明します。

（1）整数型

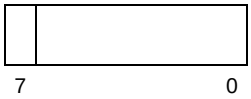
（a）内部表現

領域の左端ビットは，符号付きの型（“unsigned”を伴わずに宣言された型）では，符号ビットとなります。符号付きの型において，値は 2 の補数表現で表されます。

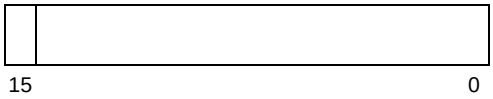
ただし，-Xchar=unsigned が指定された場合，“signed”も“unsigned”も伴わずに宣言された char 型は，符号なし（unsigned）となります。

図 3―1　整数型の内部表現

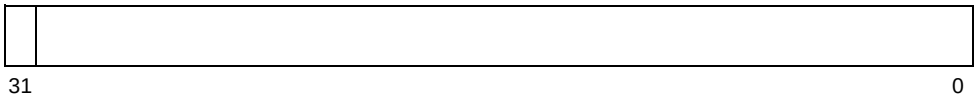
char（unsigned では符号ビットなし）



short（unsigned では符号ビットなし）



int，long（unsigned では符号ビットなし）



（b）値域

表 3―9　整数型の値域

型	値域
char 注	-128 ～ +127
short	-32768 ～ +32767
int	-2147483648 ～ +2147483647
long	-2147483648 ～ +2147483647
unsigned char	0 ～ 255
unsigned short	0 ～ 65535
unsigned int	0 ～ 4294967295
unsigned long	0 ～ 4294967295

注　CA850 で“-Xchar=unsigned”が指定された場合，0 ～ 255 の値域です。

注意　CA850 は 64 ビット長の演算はできません。

(2) 浮動小数点型

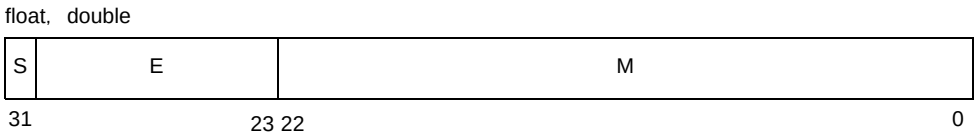
(a) 内部表現

浮動小数点型データの内部表現は，IEEE754 注に準拠しています。領域の左端のビットは，符号ビットとなります。この符号ビットの値が 0 であれば正の値に，1 であれば負の値になります。

また，double 型は float 型と同じ浮動小数点表現であり，32 ビット・データ（単精度）として扱われます。

注 IEEE：Institute of Electrical and Electronics Engineers（電気電信学会）の略称です。
また，IEEE754 とは，浮動小数点演算を扱うシステムにおいて，扱うデータ形式や数値範囲などの仕様の統一化を図った標準です。

図 3―2 浮動小数点型の内部表現



S： 仮想部の符号ビット
E： 指数部（8 ビット）
M： 仮想部（23 ビット）

(b) 値域

表 3―10 浮動小数点型の値域

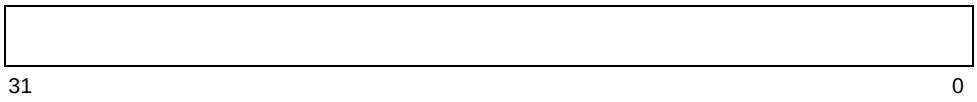
型	値域
float，double	$1.18 \times 10^{-38} \sim 3.40 \times 10^{38}$

(3) ポインタ型

(a) 内部表現

ポインタ型の内部表現は，unsigned int 型の内部表現と同じです。

図 3―3 ポインタ型の内部表現

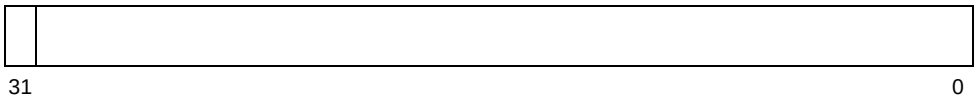


(4) 列挙型

(a) 内部表現

列挙型の内部表現は，singed int 型の内部表現と同じです。領域の左端のビットは，符号ビットとなります。

図 3―4 列挙型の内部表現



-Xenum_type=string オプション指定時には，「(30) 列挙型指定子」を参照してください。

(5) 配列型

(a) 内部表現

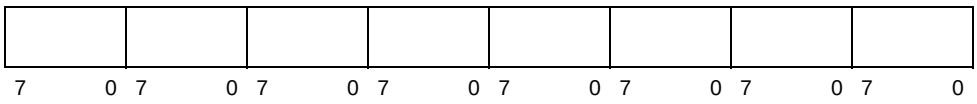
配列型の内部表現は，配列の要素を，その要素の整列条件（alignment）を満たす形で並べたものとなります。

例

```
char    a[8] = {1, 2, 3, 4, 5, 6, 7, 8};
```

上記の例に示した配列に対する内部表現は，次のようになります。

図 3―5 配列型の内部表現



(6) 構造体型

(a) 内部表現

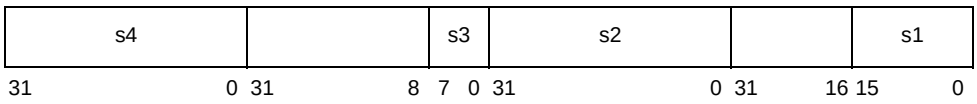
構造体型の内部表現は，構造体の要素をその要素の整列条件を満たす形で並べたものとなります。

例

```
struct{
    short   s1;
    int     s2;
    char    s3;
    long    s4;
}tag;
```

この例に示した構造体に対する内部表現は，次のようになります。

図 3―6 構造体型の内部表現



なお，構造体パッキング機能利用時の内部表現は，「(11) 構造体パッキング」を参照してください。

(7) 共用体型

(a) 内部表現

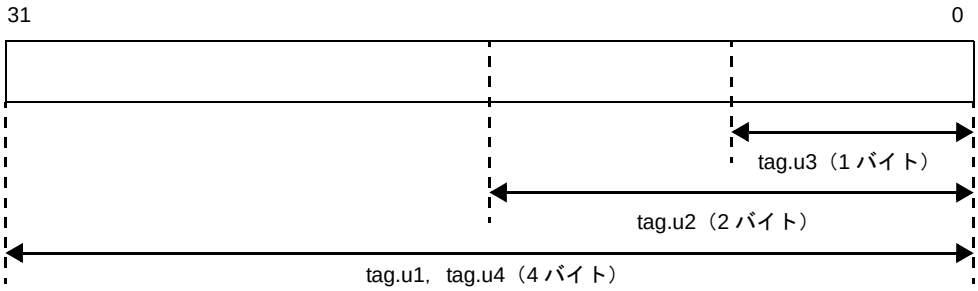
共用体はそのメンバがすべてオフセット 0 から始まり，そのメンバの任意のものを収容するのに十分なサイズを持つ構造体と考えられます。つまり，共用体型の内部表現は，同じアドレスに共用体の要素それぞれが単体で置かれているのと同様です。

例

```
union{
    int     u1;
    short   u2;
    char    u3;
    long    u4;
}tag;
```

この例に示した共用体に対する内部表現は，次のようになります。

図 3―7　共用体型の内部表現



（8）ビット・フィールド

（a）内部表現

ビット・フィールドに対しては，宣言された数のビットを含む領域が取られます。符号付きの型として宣言されたビット・フィールドに対しては，最上位ビットは符号ビットとなります。

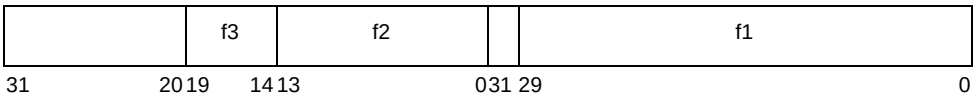
最初に宣言されたビット・フィールドは，ワード領域の最下位ビットから割り当てられます。ビット・フィールドに対し，その前のビット・フィールドに続けて領域を割り当てると，その領域がそのビット・フィールドの宣言において指定された型の整列条件を満たす境界を越えてしまう場合，そのビット・フィールドに対する領域はその整列条件を満たしている境界から割り当てられます。

例

```
struct{
    unsigned int    f1:30;
    int             f2:14;
    unsigned int    f3:6;
}flag;
```

この例に示したビット・フィールドに対する内部表現は，次のようになります

図 3―8　ビット・フィールドの内部表現



なお，ANSI 規格ではビット・フィールドに char 型，および short 型は指定できませんが，CA850 では，ビット・フィールドに char 型，および short 型を指定することができます。

ただし，この場合，警告メッセージが出力され，指定した型の整列条件でパディング[注]されます。

なお，構造体パッキング機能利用時のビットフィールドの内部表現は，「（11）構造体パッキング」を参照してください。

注　CA850 のオプションで，-ansi を指定した場合は，エラーとなります。

（9）整列条件

（a）基本型に対する整列条件

次に，基本型に対する整列条件を示します。

ただし，CA850 の -Xi を指定した場合，配列型はすべてワード境界となります。

表 3―11 基本型に対する整列条件

基本型	整列条件
（unsigned）char とその配列型	バイト境界
（unsigned）short とその配列型	ハーフワード境界
（ポインタを含む）その他の基本型	ワード境界

（b）共用体型に対する整列条件

共用体型に対する整列条件は，最大メンバ・サイズにより，次のようになります。

表 3―12 基本型に対する整列条件

最大メンバ・サイズ	整列条件
2 バイト＜サイズ	ワード境界
サイズ≦ 2 バイト	最大メンバ・サイズ境界

それぞれの場合における例を示します。

例 1.

```
union   tug1{
    unsigned short  i;  /*2 バイト・メンバ */
    unsigned char   c;  /*1 バイト・メンバ */
};  /* 共用体は 2 バイトで整列 */
```

2.

```
union   tug2{
    unsigned int    i;  /*4 バイト・メンバ */
    unsigned char   c;  /*1 バイト・メンバ */
};  /* 共用体は 4 バイトで整列 */
```

(c) 構造体型に対する整列条件

構造体型に対する整列条件は，構造体のサイズ（整列部分を含めない）により，次表のようになります。
ただし，CA850 の -Xi を指定した場合，構造体型はすべてワード境界となります。

表 3―13 構造体型に対する整列条件

構造体サイズ	整列条件
2 バイト＜サイズ	ワード境界
サイズ≦ 2 バイト	サイズとメンバの型により，次のいずれかになります。 - int 型以上の型のメンバが存在する場合 →ワード境界 - 上記以外で，short 型のメンバが存在するか，またはサイズが 2 の場合 →ハーフワード境界 - char 型のメンバのみで，サイズが 1 バイトの場合 →バイト境界

それぞれの場合における例を示します。

例 1.

```
struct  SS{
    int     i;      /*4 バイト・メンバ */
    char    c;      /*1 バイト・メンバ */
};  /* 構造体は 4 バイトで整列 */
```

2.

```
struct  BIT_I{
    int     i1:5;   /*4 バイト・メンバ（サイズは 1 バイト以下）*/
};  /* メンバの型が int のため，構造体は 4 バイトで整列 */
```

3.

```
struct  BIT_C{
    char    c1:5;   /*1 バイト・メンバ */
};  /* 構造体は 1 バイトで整列 */
```

4.

```
struct  BIT_CC{
    char    c1:5;   /*1 バイト・メンバ */
    char    c2:5;   /*1 バイト・メンバ */
};  /* サイズが 2 バイトのため，構造体は 2 バイトで整列 */
```

(d) 関数引数に対する整列条件

関数引数に対する整列条件は，ワード境界となります。

(e) 実行プログラムに対する整列条件

リロケータブルなオブジェクト・ファイルをリンクして実行可能なオブジェクト・ファイルを生成する際の整列条件は，ハーフワード境界となります。

3. 1. 4 汎用レジスタ

以下に，CA850 における汎用レジスタの用い方を示します。
なお，汎用レジスタには，次の機能があります。

(1) ソフトウエア・レジスタ・バンク

作業用レジスタ（r10-r19），およびレジスタ変数用レジスタ（r20-r29）は，CA850 の -reg オプションにより使用本数を抑制できます（「3. 1. 6 ソフトウエア・レジスタ・バンク」を参照）。

(2) マスク・レジスタ機能

r20，および r21 のレジスタは，マスク値の設定のために用いることができます（「3. 1. 7 マスク・レジスタ」を参照）。

表 3―14 汎用レジスタの用い方

レジスタ		使用方法
r0	ゼロ・レジスタ	0 の値として演算時に使用 また，const セクション（ROM 領域に置く読み出し専用セクション）注などに配置されたデータの参照にも使用
r1	アセンブラ予約レジスタ	アセンブラにおける命令展開時に使用
r2（hp）	ハンドラ・スタック・ポインタ	システム予約
r3（sp）	スタック・ポインタ	スタック・フレームの先頭を指すものとして使用
r4（gp）	グローバル・ポインタ	外部変数の参照時に使用
r5（tp）	テキスト・ポインタ	テキスト・セクション（.text セクション）の先頭を指すものとして使用
r6-r9	引数用レジスタ	引数の受け渡しに使用
r10-r19	作業用レジスタ	演算時の作業用の領域として使用（r10 は関数の戻り値の受け渡しにも使用）
r20-r29	レジスタ変数用レジスタ	レジスタ変数用の領域として使用
r30（ep）	エレメント・ポインタ	内蔵 RAM や外部 RAM のセクション注に配置指定された外部変数の参照に使用
r31（lp）	リンク・ポインタ	関数の戻り先アドレスの受け渡しに使用

注 データのセクションへの配置については，「(1) データのセクション割り当て」を参照してください。

3. 1. 5 データの参照方法

次に，CA850 におけるデータの参照方法を示します。

表 3―15 データの参照方法

種類	参照方法
数値定数	イミーディエト
文字列定数	グローバル・ポインタ（gp）からのオフセット エレメント・ポインタ（ep）からのオフセット r0 からのオフセット
自動変数，引数	スタック・ポインタ（sp）からのオフセット
外部変数，関数内静的変数	グローバル・ポインタ（gp）からのオフセット エレメント・ポインタ（ep）からのオフセット r0 からのオフセット
関数のアドレス	テキスト・ポインタ（tp）からのオフセットを用いて実行時に演算

3. 1. 6 ソフトウエア・レジスタ・バンク

CA850 では，ソフトウエアによるレジスタ・バンク機能を実現するため，3 つのレジスタ・モードが提供されています。レジスタ・モードを効率的に指定することにより，割り込み処理時やタスク切り替え時に，一部のレジスタの退避，復帰処理が不要となり，処理速度が高められます。レジスタ・モードの指定は，CA850 のレジスタ・モード指定オプション（-reg）によって行います。この機能は，CA850 が内部で使用するレジスタの本数を段階的に抑制し，次の効果が期待できます。

- 余ったレジスタをアプリケーション・プログラム（アセンブリ言語ソース・プログラム）で自由に使うことができる。
- 退避，復帰で生じるオーバーヘッドが減少する。

注意 CA850 によるレジスタ割り付けの対象となる変数の多いアプリケーション・プログラムでは，レジスタ・モードの指定によって，それまでレジスタに割り付けられていた変数がメモリ・アクセスとなり，その分，処理速度が低下することがあります。

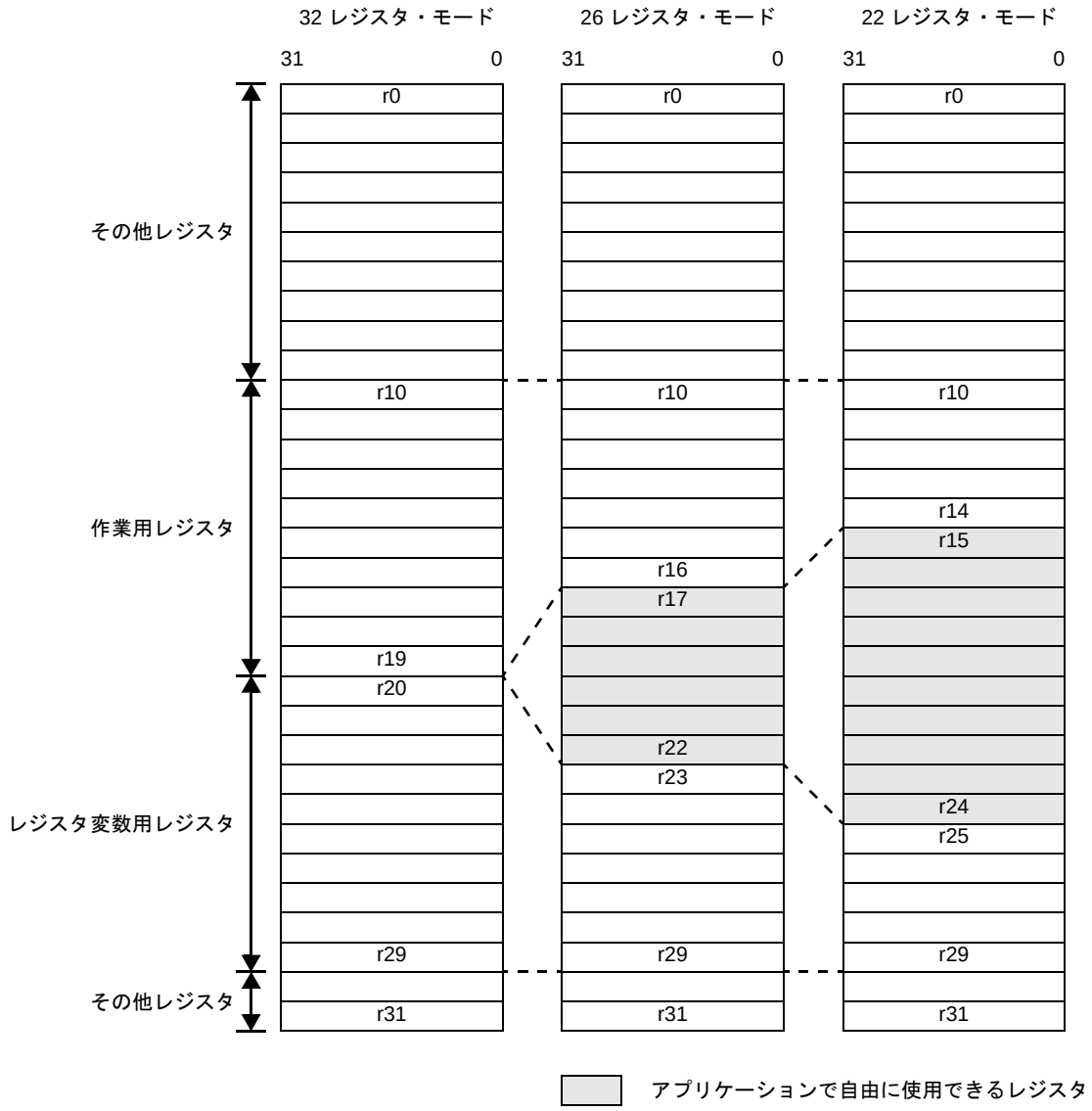
(1) レジスタ・モード

次表，および次図に，CA850 のレジスタ・モードとして提供されている 3 つのモードを示します。

表 3―16 CA850 が提供するレジスタ・モード

レジスタ・モード	作業用レジスタ	レジスタ変数用レジスタ
32 レジスタ・モード（デフォルト）	r10-r19	r20-r29
26 レジスタ・モード	r10-r16	r23-r29
22 レジスタ・モード	r10-r14	r25-r29

図 3―9 レジスタ・モードと使用可能レジスタ



コマンド・ラインにおける指定例

```
> ca850 -cpu 3201 -reg26 file.c ← 26 レジスタ・モードでコンパイル
```

(2) レジスタ・モードとライブラリ

CA850 が提供するライブラリ（「第 6 章 関数仕様」を参照）は，各レジスタ・モードごとに用意されています。ライブラリを検索する標準フォルダは，デフォルトでは，“Install Folder \ lib850 \ r32”，および“Install Folder \ lib850”ですが，CA850 で 22，または 26 レジスタ・モードを指定した場合，“Install Folder \ lib850 \ r32”の代わりに，“Install Folder \ lib850 \ r22”，または“Install Folder \ lib850 \ r26”が，ライブラリに対する標準フォルダとなります。

また，CA850 から ld850 を起動するのではなく，コマンド・ラインで ld850 を直接起動してオブジェクト・ファイルをリンクする場合，ld850 の -reg オプションを指定すると，各レジスタ・モードに合ったライブラリを参照します。

3. 1. 7 マスク・レジスタ

V850 マイクロコントローラでは，バイト・データ，ハーフワード・データをメモリからレジスタにロードする場合，最上位ビットの値によりワード長へ符号拡張します。このため，unsigned char，unsigned short 型データの演算では，上位ビットのマスク・コードが生成される場合があります。

また，演算結果をレジスタ変数へ格納する場合，符号なしバイト・データ，符号なしハーフワード・データでは，上位ビットをクリアするためにマスク・コードが生成されます。どちらの場合も，ワード・データに切り替えれば回避できますが，ワード・データにできず，マスク・コードが生成される場合，マスク・レジスタ機能を用いることにより，コード・サイズの削減ができます。

ただし，マスク・レジスタ機能を利用するか否かの判断には，利用する側で次の点を十分考慮する必要があります。

- マスク・コードが多く出力されるプログラムであるか。
- マスク・レジスタとして使用するため，レジスタ変数用レジスタが 2 本少なくなるが，その影響はないか。

次の例のように，マスク・レジスタ機能では，r20，および r21 をマスク・レジスタとして CA850 が使用します。なお，マスク・レジスタへのマスク値の設定は，プログラムで行う必要があります。

例 マスク・コード生成例

```
unsigned char   UC;
unsigned short  US;
void f(void){
    register unsigned char  ruc;
    register unsigned short rus;
       :
    UC *= UC;
       :
    ruc = UC;
    rus = US;
       :
}
```

【通常の出力コード】	【マスク・レジスタ利用時の出力コード】
ld.b $UC, r11	ld.b $UC, r11
andi 0xff, r11, r11	and r20, r11
mulh r11, r11	mulh r11, r11
st.b r11, $UC	st.b r11, $UC
:	:
ld.b $UC, r29	ld.b $UC, r29
andi 0xff, r29, r29	and r20, r29
ld.h $US, r28	ld.h $US, r28
andi 0xffff, r28, r28	and r21, r28

V850Ex を使用する場合，“符号なしデータの演算”を行う命令が追加されており，CA850 もこの命令を使用するコードを出力します。そのため V850Ex を使用する場合は，マスク・レジスタを使用する設定にしても，さほど効果があがらない場合があります。

(1) マスク値の設定

マスク・レジスタとなる r20，および r21 には，プログラムでマスク値（0xff, 0xffff）を設定する必要があります。CA850 は，マスク値が設定されているものと想定して，マスク用レジスタを使用したマスク・コードを生成します。

例 マスク値の設定例

```
__start:
    mov     #__tp_TEXT, tp
    mov     #__gp_DATA, gp
      :
    mov     0xff, r20    --r20 にマスク値を設定
    mov     0xffff, r21 --r21 にマスク値を設定
      :
    jarl    _main, lp
```

ただし，リアルタイム OS を使用したプログラムの場合，リアルタイム OS の種類によって次のようになります。

(a) RI850V4 を使用する場合

スタート・アップ・モジュールなどで，あらかじめマスク値を設定する必要があります。

(b) リアルタイム OS を使用しない場合

スタート・アップ・モジュールなどで，あらかじめマスク値を設定する必要があります。**注**

注 パッケージ添付のスタート・アップ・モジュール例“crtN. s”（32 レジスタ・モード用）では，マスク値の設定を行っています（「7. 3 スタートアップ・ルーチン」を参照）。

(2) マスク・レジスタ機能の使用方法，および注意事項

マスク・レジスタ機能を使用するための指定，および注意事項は次のとおりです。

(a) C 言語ソース・ファイルを新たにコンパイルする場合

CA850 のマスク・レジスタ機能用オプション（-Xmask_reg）を指定することにより，マスク・レジスタを使用したマスク・コードと，マスク・レジスタ機能を使用していることを示す情報（“.option mask_reg”疑似命令）を含んだアセンブラ命令が出力されます。

(b) リンク時のチェック

CA850 のマスク・レジスタ機能用オプション（-Xmask_reg）を指定して，リンカまで一度に起動した場合，.c ファイルから作成されたことを示すファイル名情報（“.file”疑似命令で指定された情報）を持つオブジェクト・ファイルに対して，リンク時にチェックを行います。このとき，マスク・レジスタ機能を使用しているオブジェクトと使用していないオブジェクトが混在していた場合，エラーとなります。

注 1. アーカイブ・ファイル（.a ファイル）に含まれるオブジェクトはチェックしません。独自に作成した .a ファイルを使用する場合，マスク・レジスタを使用していないことを確認してください。

2. コマンド・ラインから，ld850 を単独で起動する場合，リンク時チェック用オプション（-mc）を指定する必要があります。

(c) 作成したアセンブリ言語ソース・ファイルの場合

はじめからアセンブラ命令で記述したプログラムの場合，マスク・レジスタを破壊していないかチェックしてください。ファイル名情報が“.c”ではないため，リンク時にチェックされません。なお，アセンブラにマスク・レジスタの使用オプション（-m）を指定すれば，アセンブル時の警告で確認できます。

(d) 提供ライブラリの制限

アーカイブ・ファイル中のオブジェクト・ファイルはリンク時チェックされませんが，パッケージ提供のライブラリはマスク・レジスタを破壊することはありません。**注**

注 標準ライブラリ中の bsearch，および qsort では，アプリケーション関数を呼び出すため，マスク・レジスタを破壊する可能性があります。マスク・レジスタ機能使用時には，bsearch，および qsort は使用しないでください（使用しても CA850 ではエラーにはなりません）。

3.1.8 デバイス・ファイル

デバイス・ファイルとは，ターゲット・デバイスの各品種ごと，または各グループごとに 1 つずつ，パッケージとして用意された，機種依存情報を持つバイナリ・ファイルです。コンパイラでは，アプリケーション・システムで使用するターゲット・システムに対応したオブジェクト・コードを生成するために，デバイス・ファイルを参照します。このため，使用するデバイス・ファイルは，デバイス・ファイルの標準フォルダに置くか，デバイス・ファイルの置かれているフォルダをコンパイラのオプションで指定するようにしてください。デバイス・ファイルが見つからないと，コンパイル時にエラーとなります。

(1) デバイス・ファイルの指定方法

C 言語のプログラムで参照するデバイス・ファイルの指定方法には，次の二通りがあります。

(a) コンパイラのオプション（-cpu デバイス名）によるデバイス名指定

例

```
> ca850 -cpu 3201 file.c
```

CubeSuite+ でビルドする場合，デバイス指定することでこのオプションと同等になります。

(b) C 言語ソース・ファイルにおける #pragma 指令（#pragma cpu デバイス名）によるデバイス名指定

例

```
#pragma cpu 3201
```

上記の例では，“3201”がデバイス名（V850ES/SA2）です。“デバイス名”として指定できる文字列は，オプション指定，#pragma 指令ともに共通です。また，大文字，小文字は区別しません。

なお，デバイス名として指定できる文字列については，各デバイスのユーザーズ・マニュアルを参照してください。

注意 1. #pragma 指令でデバイス名を指定する場合，すべてのソース・ファイルにデバイス指定を記述する必要があります。

2. #pragma 指令によるデバイス名の指定は，ソース・ファイルの先頭に記述してください。デバイス名指定の前に記述できる処理は，C 言語の構文に関係のない前処理とコメントに限られます。デバイス名指定を C 言語の構文中に記述した場合，コンパイラは次のエラー・メッセージを出力し，処理を中止します。

```
F2625: illegal placement ' #pragma cpu '
```

例 誤った指定の例

```
#include    <stdio.h>
int i;
#pragma cpu 3201
   :
```

(2) デバイス・ファイル指定時の注意

(a) デバイス名を指定しない場合

#pragma 指令による指定，または -cpu オプションによる指定がなく，-cn オプション，-cnv850e オプション，または -cnv850e2 オプション[注]の指定がない場合，コンパイラは次のエラー・メッセージを出力し，コンパイルを中止します。

```
F2620: unknown cpu type, cannot compile
```

注 -cn オプション，-cnv850e オプション，または -cnv850e2 オプションを指定した場合でも，リンク時にはデバイス・ファイルが必要となります。

(b) オプション指定，#pragma 指令の両方でデバイスを指定した場合

コンパイラは警告メッセージを出力し，オプション指定を優先して処理を続行します。

ただし，複数のオプション指定，または複数の #pragma 指令で，異なるデバイス名を指定した場合，コンパイラは次のエラー・メッセージを出力し，処理を中止します。

```
F2622: duplicated cpu type
```

(c) アセンブラ命令で記述したプログラムの場合

この場合も，リンク可能なオブジェクト・ファイル作成時に，アセンブラのオプション，または .option 疑似命令によるデバイス指定が必要です。

3. 2 拡張言語仕様

この節では，CA850 で拡張されている言語仕様について説明します。

拡張仕様には，データ／テキストのセクション配置指定や，デバイス内蔵の周辺 I/O レジスタのアクセスを C 言語レベルで行う方法，C 言語にアセンブラ命令の記述を挿入する方法，関数ごとにインライン展開を指定する方法，割り込みや例外発生時のハンドラの定義，割り込み禁止指定を C 言語レベルで行う方法，ターゲット環境にリアルタイム OS を使用した場合に有効なリアルタイム OS 用機能，C 言語への命令の組み込みなどがあります。

3. 2. 1 マクロ名

次に，サポートしているマクロ名を示します。

なお，“__”で終わらないマクロは，従来の C 言語仕様（K&R 仕様）のために提供しているものです。ANSI 規格に厳密な処理を行う場合，前後に“__”のある形式のマクロを利用するようにしてください。

表 3―17 サポートしているマクロ

マクロ名	定義
__LINE__	その時点でのソース行の行番号（10 進数）。
__FILE__	仮定されたソース・ファイルの名前（文字列定数）。
__DATE__	ソース・ファイルの翻訳日付（“Mmm dd yyyy”の形式をもつ文字列定数。ここで，月の名前は ANSI 規格で規定されている asctime 関数で生成されるもの（英字 3 文字の並びで最初の 1 文字のみ大文字）と同じもの。dd の最初の文字は値が 10 より小さい場合空白とします）。
__TIME__	ソース・ファイルの翻訳時間（asctime 関数で生成される時間と同じような“hh:mm:ss”の型式をもつ文字列定数）。
__STDC__	10 進定数 1（-ansi オプション指定時に定義）。**注**
__v800 __v800__	10 進定数 1。
__v850 __v850__	10 進定数 1。
__v850e __v850e__	10 進定数 1（CA850 で，ターゲット・デバイスに V850Ex を指定した場合に定義）。
__v850e2 __v850e2__	10 進定数 1（CA850 で，ターゲット・デバイスに V850E2/xxx を指定した場合に定義）。
__CA850 __CA850__	10 進定数 1。
__CHAR_SIGNED__	10 進定数 1（-Xchar オプションで，符号つきを指定した場合，および -Xchar オプションを指定しない場合に定義）。
__CHAR_UNSIGNED__	10 進定数 1（-Xchar オプションで，符号なしを指定した場合に定義）。
__DOUBLE_IS_32BITS__	10 進定数 1。
_DOUBLE_IS_32BITS	10 進定数 1。
CPU マクロ	ターゲット CPU を示すマクロで 10 進定数 1。デバイス・ファイル中の「品種指定名」で示される文字列の先頭と末尾に“__”を付けたものが定義されます。

マクロ名	定義
レジスタ・モード・マクロ	ターゲット CPU を示すマクロで 10 進定数 1。 レジスタ・モードと定義されるマクロは次のとおりです。 32 レジスタ・モード ：__reg32__ 26 レジスタ・モード ：__reg26__ 22 レジスタ・モード ：__reg22__

注 -ansi オプション指定時の処理については，「3. 1. 2 ansi オプション」を参照してください。

3. 2. 2 キーワード

CA850 では，拡張機能を実現するために次の字句をキーワードとして追加しています。これらの字句も ANSI-C のキーワードと同様，ラベルや変数名として使用することはできません。

次に，CA850 で追加されているキーワード一覧を示します。

```
_asm，_bsh，_bsw，data，__DI，__EI，_halt，_hsw，__interrupt，_mul32，_mul32u，__multi_interrupt，_nop，
_sasf，_satadd，_satsub，__set_il，_sxb，_sxh
```

3. 2. 3 #pragma 指令

CA850 では，次の #pragma 指令が指定できます。

(1) アセンブラ命令の記述

C 言語中に，アセンブラ命令を記述することができます。

なお，記述方法についての詳細は「(4) アセンブラ命令の記述」を参照してください。

```
#pragma asm
    アセンブラ命令
#pragma endasm
```

(2) インライン展開指定

インライン展開する関数を指定することができます。

なお，インライン展開についての詳細は「(8) インライン展開」を参照してください。

```
#pragma inline  関数名 [, 関数名 , …]
```

(3) デバイス種別指定

使用するデバイスの機種依存情報を定義したデバイス・ファイルを参照するように指定します。CA850 のデバイス指定オプション（-cpu）と同じ機能です。C 言語ソース内にデバイスを定義したい場合に用います。

```
#pragma cpu      デバイス名
```

(4) データ／プログラムのメモリ割り当て

(a) section

変数を任意のセクションに割り当てます。

なお，配置方法についての詳細は「(1) データのセクション割り当て」を参照してください。

(b) text

任意の名称のテキスト・セクションに関数を指定できます。

なお，配置指定についての詳細は「(2) 関数のセクション割り当て」を参照してください。

```
#pragma section セクション種別　　[" セクション名 "] [begin | end]
#pragma text　　[" セクション名 "] [ 関数名 ]
```

(5) 周辺 I/O レジスタ名有効化指定

周辺 I/O レジスタ名を用いて，デバイスの持つ周辺 I/O レジスタにアクセスします。周辺 I/O レジスタ名をそのまま用いてプログラミングする場合はこの #pragma 指令を指定する必要があります。

```
#pragma ioreg
```

(6) 割り込み／例外ハンドラ指定

割り込み／例外処理ハンドラを C 言語で記述します。

なお，記述方法については「(c) 割り込み／例外ハンドラの記述方法」を参照してください。

```
#pragma interrupt　割り込み要求名　　関数名　　[ 配置方法 ]
```

(7) 割り込み禁止関数指定

関数全体を割り込み禁止にします。

```
#pragma block_interrupt 関数名
```

(8) タスク指定

リアルタイム OS 上で動作するタスクを C 言語で記述します。

なお，記述方法についての詳細は「(a) タスクの記述」を参照してください。

```
#pragma rtos_task　関数名
```

(9) 構造体パッキング指定

構造体パッキングを指定します。数値はパッキング値，すなわちメンバのアライメント値を指定します。数値には 1，2，4，8 が指定できます。数値を指定しない場合，デフォルト 8 [注]となります。

```
#pragma pack([1248])
```

注　本バージョンではアライメント値“4”と“8”は同じになります。

3. 2. 4　拡張仕様の使用方法

この項では，下記の拡張機能の使用方法について説明します。

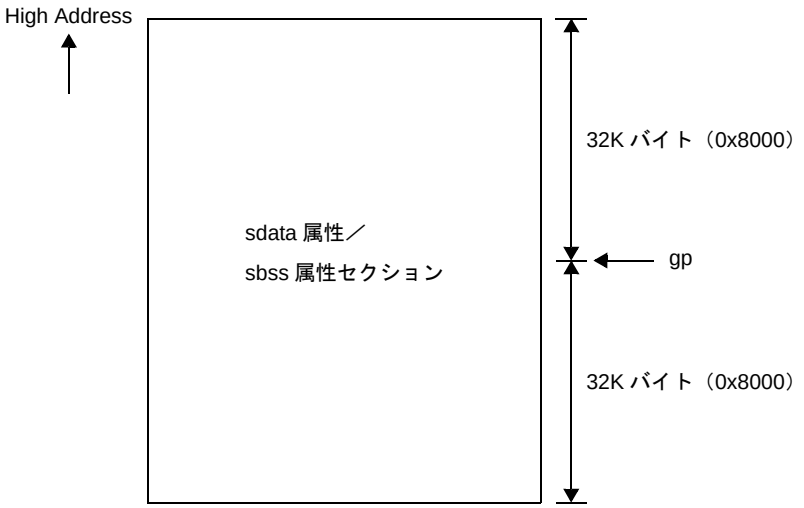
- データのセクション割り当て
- 関数のセクション割り当て
- 周辺 I/O レジスタへのアクセス
- アセンブラ命令の記述
- 割り込みレベルの制御
- 割り込み禁止
- 割り込み／例外処理ハンドラ
- インライン展開
- リアルタイム OS 対応機能
- 組み込み関数
- 構造体パッキング

(1) データのセクション割り当て

CA850 は，C 言語ソース上で外部変数やデータが定義されると，それらをメモリ上に配置します。配置されるメモリ領域は，基本的に“グローバル・ポインタ（gp）”の指すアドレスからのオフセットによって参照できる領域です。つまり，プログラム中で変数やデータにアクセスする場合，デフォルトで gp を使ってアクセスするコードを出力しようとします。

さらに CA850 は，できるだけオブジェクト効率や実行効率を上げるため，gp から 1 命令で参照可能な領域に配置するコードを出力しようとします。しかし，gp から 1 命令で参照可能な範囲は，V850 アーキテクチャ上，gp から± 32K バイト内（合計で 64K バイト内）である必要があります。CA850 では，gp から± 32K バイト内の領域に専用のセクションが設けており，“sdata 属性／ sbss 属性セクション”と呼びます。

図 3―10　sdata 属性／ sbss 属性セクション



しかし，変数の数が多いアプリケーションでは，この範囲内に収まりきらない場合があります。その場合は，それ以外のセクションに割り当てなくてはなりません。CA850 では sdata 属性／ sbss 属性セクション以外に

も，変数やデータを配置するためのさまざまなセクションを用意しています。それぞれに特徴があり，より高速にアクセスしたい変数を配置できるセクションなどもあるので，用途によって使い分けることができます。

次に，sdata 属性／ sbss 属性セクションを含め，CA850 で使用できるセクションを示します。

- sdata 属性／ sbss 属性セクション

gp から 1 命令で参照可能なセクションで，gp から± 32K バイト内に配置される必要があります。初期値ありデータは sdata 属性セクションへ，初期値なしデータは sbss 属性セクションへ配置されます。

CA850 では，まずこのセクションに配置するコードを生成しようとします。

ただし，この属性のセクションに収まりきらないような場合は，エラーになります。

なお，sdata 属性／ sbss 属性セクションへの配置データを少しでも多くする方法として，CA850 のオプション“-G”で，配置されるデータのサイズの上限を指定し，それ以上のサイズのデータは sdata 属性／ sbss 属性セクションに配置しないという指定ができます（オプションの詳細は「V850 ビルド編」を参照）。

なお，プログラム中で sdata 属性／ sbss 属性セクションに配置したい変数を指定する場合は，#pragma section 指令を使用します（詳細は「（a）#pragma section 指令」を参照）。

```
#pragma section sdata   begin
int a = 1;  /*sdata セクション配置 */
int b;      /*sbss セクション配置 */
#pragma section sdata   end
```

- data 属性／ bss 属性セクション

gp から 2 命令で参照可能なセクションです。アドレス生成を行ってからアクセスするため，その分コードが多くなり，実行速度も落ちますが，32 ビット空間内すべてにアクセスが可能です。

したがって，RAM 上であれば，どこにでも配置が可能なセクションです。

なお，C 言語プログラム中で data 属性／ bss 属性セクションに配置したい変数を指定する場合は，#pragma section 指令を使用します（詳細は「（a）#pragma section 指令」を参照）。

```
#pragma section data    begin
int a = 1;  /*data セクション配置 */
int b;      /*bss セクション配置 */
#pragma section data    end
```

- sconst 属性セクション

r0，つまり，0 番地から 1 命令で参照可能なセクションで，0 番地から +32K バイト内に配置される必要があります。基本的に“ROM に固定してもよいデータ”を配置するセクションです。V850 で内蔵 ROM を持つデバイスの場合，0 番地からプラス方向が内蔵 ROM である場合が多く，そこに 1 命令で参照したい，かつ ROM 固定してもよいデータを，sconst 属性セクションとして配置します。sconst 属性セクションに配置するデータは，const 修飾子をつけて宣言された変数／データが対象となります。この属性のセクションに収まりきらないような場合は，const 属性セクションへ配置することになります。

なお，sconst 属性セクションへの配置データを少しでも多くする方法として，CA850 のオプション“-Xsconst”で，配置されるデータのサイズの上限を指定し，それ以上のサイズは sconst 属性セクションに配置しないという指定ができます（オプションの詳細は「V850 ビルド編」を参照）。
なお，プログラム中で sconst 属性セクションに配置したい変数を指定する場合は，#pragma section 指令を使用します。（詳細は「(a) #pragma section 指令」を参照）。

```
#pragma section sconst  begin
const int   a = 1;  /*sconst セクション配置 */
#pragma section sconst  end
```

- const 属性セクション

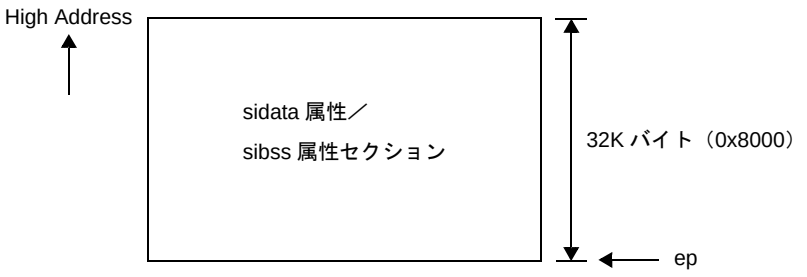
r0，つまり，0 番地から 2 命令で参照可能なセクションです。sconst 属性セクションに入りきらなかった“ROM 固定してもよいデータ”や，V850 の ROM レス品で，外部 ROM にデータを配置したい場合に，const 属性セクションに配置します。const 属性セクションに配置するデータは const 修飾子をつけて宣言された変数／データが対象になります。
また，const 修飾子をつけて宣言された変数，文字列定数は #pragma section 指令で .const セクションに配置する指令がない場合でも const 属性セクションに割り付けられます。アドレス生成を行ってからアクセスするため，その分コードが多くなり，実行速度も落ちますが，32 ビット空間内すべてにアクセスが可能です。したがって，32 ビット空間内であれば，どこにでも配置が可能なセクションです。
なお，プログラム中で const 属性セクションに配置したい変数を指定する場合は，#pragma section 指令を使用します。（詳細は「(a) #pragma section 指令」を参照）。

```
#pragma section const   begin
const int   a = 1;  /*const セクション配置 */
#pragma section const   end
```

- sidata 属性／ sibss 属性セクション

ep（エレメント・ポインタ）から 1 命令で参照可能なセクションで，ep からプラス方向へアクセスするセクションです。つまり，ep からプラス方向 32K バイト内に配置されるセクションです

図 3―11　sidata 属性／ sibss 属性セクション



初期値ありデータは sidata 属性セクションへ，初期値なしデータは sibss 属性セクションへ配置されます。gp から 1 命令でアクセスできる sdata 属性／ sbss 属性セクションに入りきらなくなったが，1 命

令アクセスしたい変数がまだ存在する場合，ep を使って 1 命令でアクセスできる範囲に置くことができます。

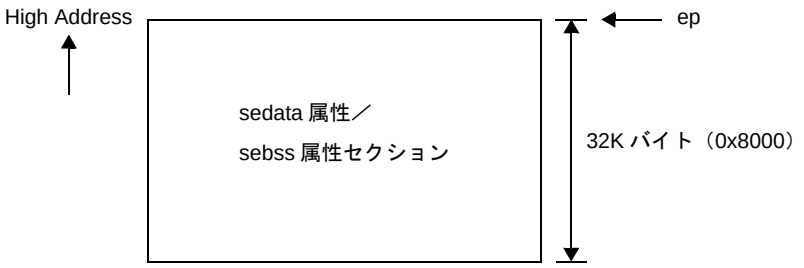
なお，プログラム中で sidata 属性／ sibss 属性セクションに配置したい変数を指定する場合は，#pragma section 指令を使用します（詳細は「(a) #pragma section 指令」を参照）。

```
#pragma section sidata  begin
int a = 1;  /*sidata セクション配置 */
int b;      /*sibss セクション配置 */
#pragma section sidata  end
```

- sedata 属性／ sebss 属性セクション

ep（エレメント・ポインタ）から 1 命令で参照可能なセクションで，ep からマイナス方向へアクセスするセクションです。つまり，ep からマイナス方向 32K バイト内に配置されるセクションです。

図 3―12 sedata 属性／ sibss 属性セクション



初期値ありデータは sedata 属性セクションへ，初期値なしデータは sebss 属性セクションへ配置されます。gp から 1 命令でアクセスできる sdata 属性／ sbss 属性セクションに入りきらなくなったが，1 命令アクセスしたい変数がまだ存在する場合，ep を使って 1 命令でアクセスできる範囲に置くことができます。

なお，プログラム中で sedata 属性／ sebss 属性セクションに配置したい変数を指定する場合は，#pragma section 指令を使用します（詳細は「(a) #pragma section 指令」を参照）。

```
#pragma section sedata  begin
int a = 1;  /*sedata セクション配置 */
int b;      /*sebss セクション配置 */
#pragma section sedata  end
```

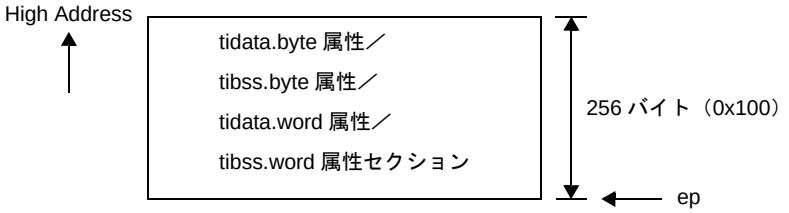
- tidata（tidata.byte，tidata.word）属性／ tibss（tibss.byte，tibss.word）属性セクション

ep（エレメント・ポインタ）から 1 命令で参照可能なセクションで，ep からプラス方向へアクセスするセクションです。sidata 属性／ sibss 属性セクションと違うところは，同じ 1 命令アクセスでも，使用するアセンブル命令が違うことです。

sidata 属性／ sibss 属性セクション，および sedata 属性／ sebss 属性セクションは，格納／参照に 4 バイト長命令の“st ／ ld 命令”を使用しますが，tidata 属性／ tibss 属性セクションは，2 バイト長命令の

“sst／sld 命令”を使用してアクセスします。つまり，sidata 属性／sibss 属性セクション，および sedata 属性／sebss 属性セクションよりもコード効率がよくなります。
ただし，sst／sld 命令が適用できる範囲は小さいので，多くの変数を配置することはできません。

図 3―13　tidata 属性／tibss 属性セクション



初期値ありデータは tidata（tidata.byte，tidata.word）属性セクションへ，初期値なしデータは tibss（tibss.byte，tibss.word）属性セクションへ配置されます。バイト・データを配置する場合は tidata.byte／tibss.byte 属性を，ワード・データを配置する場合は tidata.word／tibss.word 属性を指定しますが，CA850 に自動判別させたい場合は tidata／tibss 属性を指定します。
システムの中でも，より高速にアクセスしたいデータを配置するために使用します。
ただし，配置できる量が少ないため，厳選する必要があります。プログラム中で tidata.byte／tibss.byte 属性，tidata.word／tibss.word 属性セクションに配置したい変数を指定する場合は，#pragma section 指令を使用します（詳細は「(a) #pragma section 指令」を参照）。

```
#pragma section tidata_byte     begin
char            a = 1;  /*tidata.byte セクション配置 */
unsigned char   b;      /*tibss.byte セクション配置 */
#pragma section tidata_byte     end
```

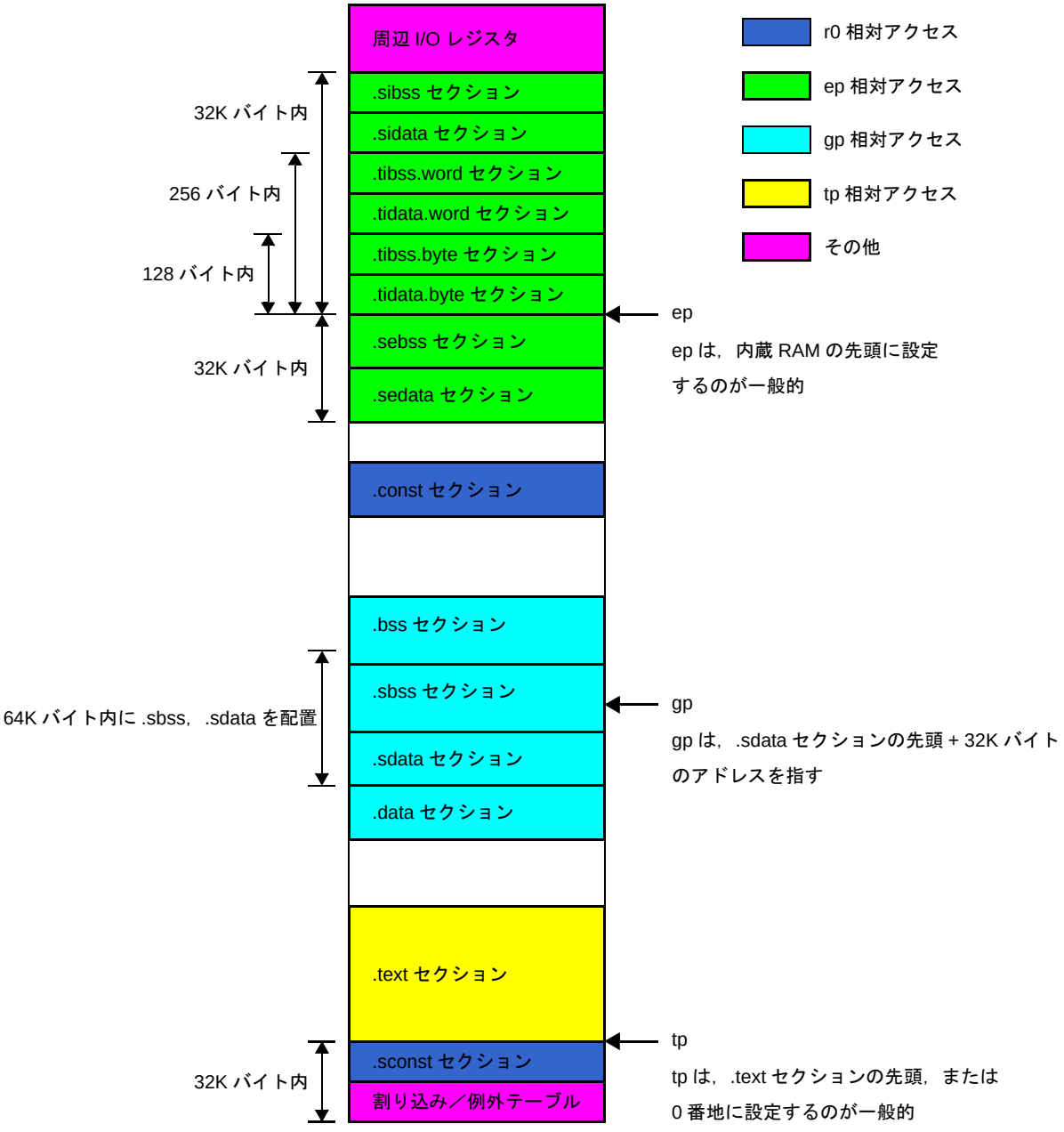
```
#pragma section tidata_word     begin
int     a = 1;          /*tidata.word セクション配置 */
short   b;              /*tibss.word セクション配置 */
#pragma section tidata_word     end
```

```
#pragma section tidata  begin
int     a = 1;          /*tidata.word セクション配置 */
char    b;              /*tibss.byte セクション配置 */
#pragma section tidata  end
```

変数／データの中で，特にシステムの中で参照頻度の高いものを選び，tidata（tidata.byte，tidata.word）属性／tibss（tibss.byte，tibss.word）属性セクションに配置できると，実行速度の面からも効率がよくなります。CA850 には，この参照頻度を調査する“セクション・ファイル・ジェネレータ”があります。調査した頻度情報を“頻度情報ファイル”として出力し，その情報を元にして，自動的に tidata（tidata.byte，tidata.word）属性／tibss（tibss.byte，tibss.word）属性セクションに配置するコードを出

力します。またその頻度情報ファイルをユーザが編集し，優先的に tidata（tidata.byte，tidata.word）属性／ tibss（tibss.byte，tibss.word）属性セクションに割り当てたい変数を選択することもできます。この方法を用いると，ソースに手を加えることなく，これらのセクションに配置することができます。次図に各セクションのメモリ配置イメージの例を示します。

図 3―14　tidata 属性／ tibss 属性セクション



（a）#pragma section 指令

#pragma section 指令を使って，目的のセクションへデータを割り当てる方法を説明します。

- セクション名をデフォルトのまま使用する場合

CA850 で定義されているセクション名をそのまま使用する場合，#pragma section 指令で次のような書式で記述します。

```
#pragma section セクション種別     begin
変数宣言／定義
#pragma section セクション種別     end
```

ここで“セクション種別”に指定できるのは，次のとおりです。

data，sdata，sedata，sidata，tidata，tidata.word，tidata.byte，sconst，const

bss 属性のセクション名をセクション種別に指定することはできません。bss 属性については，宣言／定義された変数・データに“初期値がある場合”は data 属性に，“初期値がない場合”は bss 属性に CA850 が自動的に振り分けます。

```
#pragma section sdata    begin
int a = 1;   /*sdata セクション配置 */
int b;       /*sbss セクション配置 */
#pragma section sdata    end
```

上記の場合，“変数 a”は初期値を持つので，data 属性の“.sdata セクション”へ，“変数 b”は初期値を持たないので，bss 属性の“sbss セクション”へ配置します。

“#pragma section セクション種別 begin”と“#pragma section セクション種別 end”の間には，変数宣言／定義を複数記述することができます。そのセクション種別に配置したい変数を列挙します。

なお，tidata.word と tidata.byte をセクション種別に指定する場合，次のように，間の“.（ピリオド）”の代わりに“_（アンダースコア）”を指定してください。

tidata_word，tidata_byte

- セクション名に独自の名前をつける場合

次の属性を持つセクションに関しては，独自のセクション名を指定し，そこに変数やデータを配置することができます。

data，sdata，sconst，const

この場合，#pragma section 指令で次のような書式で記述します。

```
#pragma section セクション種別     “作成するセクション名”     begin
変数宣言／定義
#pragma section セクション種別     “作成するセクション名”     end
```

ただし，この指定方法で実際に生成されるセクション名は，次のようにユーザが作成するセクション名に“. セクション種別”が付加されたものになります。

```
作成するセクション名 . セクション種別
```

これは，初期値の“ある”／“なし”で，data 属性，bss 属性に分かれるためで，同一セクション名で別属性のセクションができてしまうのを防ぐためです。リンク・ディレクティブ・ファイルでセクション指定するときは，生成されたセクション名を指定してください。リンク・ディレクティブ・ファイルにおける指定例については，「(b) 独自のデータ・セクションのリンク・ディレクティブ指定」を参照してください。ユーザが独自に指定するセクション名と生成されるセクション名の具体例は，次のとおりです。

表 3―18 算術演算命令

ユーザが独自に指定したセクション名	セクション種別	付加される文字列	生成されるセクション名
mydata	data 属性	data ／ .bss	mydata.data ／ mydata.bss
mysdata	sdata 属性	sdata ／ .sbss	mysdata.sdata ／ mysdata.sbss
myconst	const 属性	.const	myconst.const
mysconst	sconst 属性	.sconst	mysconst.sconst

次のように指定した場合，“変数 a”は初期値を持つので“mysdata.sdata セクション”へ，“変数 b”は初期値を持たないので“mysdata.sbss セクション”へ配置されます。

```
#pragma section sdata   "mysdata"       begin
int a = 1;  /*mysdata.sdata セクション配置 */
int b;      /*mysdata.sbss セクション配置 */
#pragma section sdata   "mysdata"       end
```

(b) 独自のデータ・セクションのリンク・ディレクティブ指定

#pragma section 指令で，独自のセクションを作成した場合，そのセクションのリンク・ディレクティブ・ファイルの記述について説明します。

C 言語ソースにて“変数 a”と“変数 b”を次のように指定した場合，“変数 a”は初期値を持つので“mysdata.sdata セクション”へ，“変数 b”は初期値を持たないので“mysdata.sbss セクション”へ配置されます。

```
#pragma section sdata   "mysdata"       begin
int a = 1;  /*mysdata.sdata セクション配置 */
int b;      /*mysdata.sbss セクション配置 */
#pragma section sdata   "mysdata"       end
```

このとき，リンク・ディレクティブ・ファイル内のマッピング・ディレクティブは，次のように記載すると，独自のセクションに配置されます。

```
.data           = $PROGBITS ?AW .data;
.bss            = $NOBITS   ?AW .bss;
mysdata.data    = $PROGBITS ?AW mysdata.data;
mysdata.bss     = $NOBITS   ?AW mysdata.bss;
```

なお，記述順に配置されるので，配置変更したい場合は，記述順を変更してください。また直接アドレス指定することもできます（ただし，セグメントを作成して，その中にマッピング・ディレクティブを記述し，セグメント単位でアドレス指定するのが一般的です）。

ここで注意が必要なのは，mysdata.data の属性が“$PROGBITS ?AW”，mysdata.bss の属性が“$NOBITS ?AW”なので，これらと同じ属性を持つマッピング・ディレクティブにて，入力セクション（上記で，マッピング・ディレクティブの一番右側に書かれる“.data”“.bss”“mysdata.data”“mysdata.bss”のこと）を省略しないでください。

(c) セクション割り当ての注意点

#pragma section 指令，const 修飾子，およびセクション・ファイルによってセクションを割り当てた場合の注意点を次に示します。

- #pragma section 指令は，次のような指定をするとコンパイル時にエラーになります。
  - セクション割り当てがネストしている
  - #pragma section の begin と end がクロスしている
  - #pragma section の begin と end のどちらかしか記述していない

【誤った例“セクションのネスト”】

```
#pragma section data    begin
int     a = 1;
#pragma section sdata   begin
short   b;
char    c = 0x10;
#pragma section sdata   end
int     d;
#pragma section data    end
```

【誤った例“セクションのクロス”】

```
#pragma section data    begin
int     a = 1;
#pragma section sdata   begin
short   b;
char    c = 0x10;
#pragma section data    end
int     d;
#pragma section sdata   end
```

- 自動変数に対してセクション指定を行った場合，その指定は無視されます。セクション指定は外部変数に対する機能です。

- 独自のセクション名を指定する場合，その名前は 256 文字以内にしてください。

- 初期値を設定しない変数宣言は，通常“仮定義”として扱われますが，セクション指定した場合は“定義”として扱われます。初期値を設定しない変数宣言と定義を混在させないようにしてください。

【#pragma section を使用しない変数宣言】	【#pragma section を使用した変数宣言】

`int i;  /* 仮定義 */`
`int i = 10; /* 定義 */`
【エラーになりません】
	`#pragma section sedata  begin`
	`int i;  /* 定義 */`
	`int i = 10; /* 定義 */`
	`#pragma section sedata  end`
	【二重定義エラー】

外部変数を参照するファイルでは，必ず extern 宣言してください。次の場合では，file1.c 側の変数の仮定義で extern がないと，二重定義エラーになります。

【file1.c】	【file2.c】

`#pragma section sedata  begin`
`extern  int  i;`
`#pragma section sedata  end`
	`#pragma section sedata  begin`
	`int  i;`
	`#pragma section sedata  end`

【extern がないと，二重定義エラー】	

- セクション指定した変数を，他のファイルで参照する場合，その変数に対する extern 宣言に対しても，同じセクション種別でセクション指定する必要があります。変数定義時に指定したセクションと異なる種別のセクションを指定した場合はエラーになります。
たとえば，定義側で“#pragma section data begin ～ #pragma section data end”指定して，仮定義側（extern 宣言）で“#pragma section data begin ～ #pragma section data end”指定しなかった場合，仮定義側では sdata に配置されているものとみなされます。つまり，定義側では gp からの 2 命令でアクセスするコードが出力され，仮定義側では gp からの 1 命令でアクセスするコードが出力されることになります。この場合，つじつまが合わなくなるため，リンク時に次のエラー・メッセージが出力されます。

F4163: output section ".data" overflowed or illegal label reference forsymbol "*symbol*" in file "*file*" (value: value, input section: section, offset: offset, type:R_V850_GPHWLO_1). "*symbol*" is allocated in section ".data" (file: *file*).

【正しい例】

【file1.c】	【file2.c】

`#pragma section sedata  begin`
`int i = 1;`
`#pragma section sedata  end`
	`#pragma section sedata  begin`
	`extern  int i;`
	`#pragma section sedata  end`

【誤った例 1】

【file1.c】	【file2.c】

`int i = 1;`	`#pragma section sedata  begin`
	`extern  int i;`
	`#pragma section sedata  end`

file1.c で定義した“変数 i”は sbss セクション，または bss セクションに配置されますが，file2.c では“変数 i”に対して sebss セクションへのアクセス・コードが出力されるため，リンカで次のエラー・メッセージが出力されます。

F4163: output section ".data" overflowed or illegal label reference forsymbol "*symbol*" in file "*file*" (value: value, input section: section, offset: offset, type:R_V850_GPHWLO_1). "*symbol*" is allocated in section ".data" (file: *file*).

【誤った例 2】

【file1.c】	【file2.c】

`#pragma section sedata  begin`
`int i = 1;`
`#pragma section sedata  end`
	`extern  int i;`

file1.c で定義した“変数 i”は sebss セクションに配置されますが，file2.c では“変数 i”に対して sbss セクション，または bss セクションへのアクセス・コードが出力されるため，リンカで次の不整合エラーが出力されます。

F4156: can not find GP-symbol in segment "*DUMMY*" or illegal labelreference for symbol "_i" in file "file2.o" (section: section, offset: offset, type:R_V850_GPHWLO_1). "_i" is allocated in section ".sedata" (file: file1.o).

- #pragma section 指令で，sconst 属性，const 属性の変数を定義する場合，変数に必ず“const 指定”をしてください。また extern 宣言で仮定義している箇所でも，同じく const 指定が必要です。

“#pragma section sconst begin ～ #pragma section sconst end”や“#pragma section const begin ～ #pragma section const end”で指定しても，変数宣言時に const 宣言がなかった場合，sconst セクション／ const セクションに配置されず（#pragma section 指令は無視され），gp 相対セクションである sdata セクションや data セクションに配置されてしまい，意図した配置にならないことになります。

【file1.c】	【file2.c】

`#pragma section sconst begin`
`const int i = 1;`
`#pragma section sconst end`
	`#pragma section sconst begin`
	`int i;`
	`#pragma section sconst end`

file1.c の方は，“変数 i”が sconst セクションに配置するコードが出力されますが，file2.c の方は，“変数 i”に const 指定がないため，#pragma section 指定が無視され，gp 相対の変数として扱われるため，sdata セクションや data セクションに配置されるコードになります。リンクしても sconst に“変数 i”が配置されません。
また，次のように extern 宣言で仮定義している箇所でも const 指定が必要になります。

【file1.c】	【file2.c】

`#pragma section sconst begin`
`const int i = 10;`
`#pragma section sconst end`
	`#pragma section sconst begin`
	`extern const int i;`
	`#pragma section sconst end`

（d）#pragma section 指令の例

次に，#pragma section 指令の例を数点示します。

- 変数 a を tidata.word セクションに，変数 b を tibss.word セクションに配置する

```
#pragma section tidata_word     begin
int     a = 1;      /*tidata.word セクションに配置 */
short   b;          /*tibss.word セクションに配置 */
#pragma section tidata_word     end
```

- 変数 c を tidata.byte セクションに，変数 d を tibss.byte セクションに配置する

```
#pragma section tidata_byte     begin
char    c = 0x10;   /*tidata.byte セクションに配置 */
char    d;          /*tibss.byte セクションに配置 */
#pragma section tidata_byte     end
```

tidata 属性セクションでは，ワード・データ／ハーフワード・データは tidata_word ／ tibss_word セクションに配置され，バイト・データは tidata_byte ／ tibss_byte セクションに配置されます。
ただし C 言語ソースで“char 型の配列”が宣言された場合，それらは“tidata.word セクション”に配置されます。tidata.word セクションは 256 バイトまで使用可能ですが，配列は char 型なので sld.b ／ sst.b 命令を使ったコードが出力されます。
しかし，sld.b ／ sst.b 命令は 128 バイト以上にアクセスすることができません。
したがって，char 型の配列を宣言したとき，その配列自体が 128 バイト以上，または，ep からの相対で 128 バイトを越えた場所に配置されると，リンク時にエラーとなってしまいます。
したがって，char 型の配列を tidata 属性セクションに割り当てる場合は注意が必要です。

- const 指定された変数 e を sconst セクションに，ポインタ p が示す文字列定数データを sconst セクションに配置する

```
#pragma section sconst  begin
const   int     e = 0x10;
const   char    *p = "Hello, World";
#pragma section sconst  end
```

上記の記述で，ポインタ p の示す“Hello World”は sconst セクションへ，ポインタ変数“p”自体は sdata セクション，または data セクションへ配置されます。ポインタ変数とポインタの示す内容の配置場所に関しては，const 指定の方法によって変わってきます。

例 1.

```
const   char    *p = "Hello, World";
```

このように宣言すると，次のようになります。

ポインタ変数“p”	書き換え可能（“p = 0”はコンパイル OK）
文字列定数“Hello World”	書き換え不可能（“*p = 0”はコンパイル NG）

ポインタ変数の指す先を const 属性に配置したい場合は，上記のように記述します。

```
#pragma section sconst  begin
const   char    *p = "Hello, World";
#pragma section sconst  end
```

上記のように定義すると次のようなセクション配置になります。

ポインタ変数“p”	sdata ／ data セクション
文字列定数“Hello World”	sconst セクション

2.

```
char    *const  p;
```

ポインタ変数“p”	書き換え不可能（“p = 0”はコンパイル NG）

ポインタ変数を const 属性に配置したい場合は，上記のように記述します。

```
char    *const  p = "Hello World";
```

上記のように記述すると，ポインタ変数，および文字列定数“Hello World”ともに，const 属性のセクションに配置されます。

```
#pragma section sconst  begin
char    *const  p = "Hello World";
#pragma section sconst  end
```

上記のように定義すると次のようなセクション配置になります。

ポインタ変数“p”	sconst セクション
文字列定数“Hello World”	sconst セクション

3.

```
const   char    *const  p;
```

ポインタ変数“p”	書き換え不可能（“p=0”はコンパイル NG）

ポインタ変数，およびその指す先を const 属性に配置したい場合は，上記のように記述します。どちらも ROM に固定するような場合に使用します。

```
const   char    *const  p = "Hello World";
```

上記のように記述すると，ポインタ変数，および文字列定数“Hello World”ともに，const 属性のセクションに配置されます。

```
#pragma section sconst  begin
const   char    *const  p = "Hello World";
#pragma section sconst  end
```

上記のように定義すると次のようなセクション配置になります。

ポインタ変数“p”	sconst セクション
文字列定数“Hello World”	sconst セクション

なお，const 指定で宣言した変数を sconst セクションに配置する方法として #pragma section 指令のほかに，コンパイラ・オプション“-Xsconst”指定があります。

- #pragma section 指令の extern 宣言する方は，共通に使用するヘッダ・ファイル内で行い，C 言語ソース内にインクルードして使用する

【header.h】
```
#pragma section sidata  begin
extern  int k;
#pragma section sidata  end
```

【file1.c】
```
#include    "header.h"
#pragma section sidata  begin
int k;
#pragma section sidata  end
```

【file2.c】
```
#include    "header.h"
void func1(void){
    k = 0x10;
}
```

上記のように #pragma section 指令の extern 宣言する方は，ヘッダ・ファイルでまとめておくと，間違いが少なくなり，ソースも見やすくなります。

(2) 関数のセクション割り当て

CA850 では，C 言語ソースの関数であるプログラム・コードは，デフォルトで“.text セクション”に配置されます。リンク・ディレクティブ・ファイルで，.text セクションの配置アドレスを決定すると，そこからプログラムを配置します。

しかし，「関数ごとに配置アドレスを変えたい」場合や，「メモリの配置上，配置アドレスを分ける必要がある」場合があります。これに対応するため，CA850 では“#pragma text 指令”を用意しています。#pragma text 指令を使って，text 属性を持つセクションに任意の名前をつけ，リンク・ディレクティブ・ファイルにて配置アドレスを変えます。

(a) #pragma text 指令

#pragma text 指令を使って，text 属性を持つセクションに任意の名前をつけることができます。

#pragma text 指令の使い方には，次の二通りあります。

- #pragma text 指令に，作成するセクションに配置する“関数名”を指定する方法

```
#pragma text     “作成するセクション名”     関数名
```

関数名は，C 言語記述の関数名を記述してください。たとえば "void func1 ( ) { }" という関数であれば "func1" と指定します。また，“作成するセクション名”を省略することもできます。その場合“関数名”で指定した関数を，デフォルトの“.text セクション”に配置します。

- 関数本体（関数定義）の前に #pragma text 指令を記述し，関数名は指定しない方法

```
#pragma text     “作成するセクション名”
```

“作成するセクション名”を省略することもできます。その場合，その直前に指定した“#pragma text”作成するセクション名の指定を解除し，これ以降の関数を，デフォルトの“.text セクション”に配置します。
ただし，この指定方法で実際に生成されるセクション名は，次のようにユーザが指定したセクション名に“.text”が付加されたものになります。

```
セクション名 .text
```

リンク・ディレクティブ・ファイルでセクション指定するときは，生成されたセクション名を指定してください。リンク・ディレクティブ・ファイルにおける指定例については，「（b）独自のデータ・セクションのリンク・ディレクティブ指定」を参照してください。
ユーザが独自に指定するセクション名と生成されるセクション名の具体例は，次のとおりです。

表 3―19 算術演算命令

ユーザが独自に指定したセクション名	セクション種別	付加される文字列	生成されるセクション名
mytext	text 属性	.text	mytext.text

次のように指定した場合，“関数 func1”は“mytext1.text セクション”へ，“関数 func2”は #pragma text 指令がないので，デフォルトで“.text セクション”へ配置されます。

```
#pragma text    "mytext1"       func1
void func1(void){
   :
}
void func2(void){
   :
}
```

また，次のように指定した場合，“関数 func1”，“関数 func2”は“mytext2.text セクション”へ，“関数 func3”は“mytext3.text セクション”へ，“関数 func4”は“#pragma text”で直前の“#pragma text”mytext3”が解除されているので，デフォルトで“.text セクション”へ配置されます。

```
#pragma text    "mytext2"
void func1(void){
   :
}
void func2(void){
   :
}
#pragma text    "mytext3"
void func3(void){
   :
}
#pragma text
void func4(void){
   :
}
```

(b) 独自のテキスト・セクションのリンク・ディレクティブ指定

#pragma text 指令で，独自のテキスト・セクションを作成した場合，そのセクションのリンク・ディレクティブ・ファイルの記述について説明します。

```
#pragma text    "mytext2"
void func1(void){
   :
}
void func2(void){
   :
}
#pragma text    "mytext3"
void func3(void){
   :
}
#pragma text
void func4(void){
   :
}
```

C 言語ソースにて，上記のように #pragma text 指定した場合，“関数 func1”，“関数 func2”は“mytext2.text セクション”へ，“関数 func3”は“mytext3.text セクション”へ，“関数 func4”は“#pragma text”で直前の“#pragma text”mytext3”が解除されているので，デフォルトで“.text セクション”へ配置されます。

```
text = $PROGBITS    ?AX .text;
mytext2 = $PROGBITS ?AX mytext2.text;
mytext3 = $PROGBITS ?AX mytext3.text;
```

なお，記述順に配置されるので，配置変更したい場合は，記述順を変更してください。また，直接アドレス指定することもできます（ただし，セグメントを作成して，その中にマッピング・ディレクティブを記述し，セグメント単位でアドレス指定するのが一般的です）。

ここで注意が必要なのは，mytext2.text ／ mytext3.text の属性が“$PROGBITS ?AX”なので，これらと同じ属性を持つマッピング・ディレクティブにて，入力セクション（上記で，マッピング・ディレクティブの一番右側に書かれる“.text ”“ mytext2.text ”“ mytext3.text”のこと）を省略しないでください。

例 下記のように同じ“$PROGBITS ?AX”属性のマッピング・ディレクティブで，入力セクションが省略されていると，リンカはその属性のセクションをすべて結合して配置するため，独自に作ったセクションへ配置されない結果となります。
したがって，mytext2.text ／ mytext3.text に配置しようとしたプログラムは .text に配置されることになります。

```
.text = $PROGBITS       ?AX;
```

（c）#pragma text 指令の注意点

#pragma text 指令の注意点を次に示します。

- #pragma text 指令は，関数の定義と同一ファイル内で，定義より前に記述してください。関数の定義よりも後ろに記述された場合は，警告メッセージを出力し，#pragma text 指令は無視します。ただし，関数のプロトタイプ宣言の順番には関係ありません。

- #pragma text 指定した関数が“direct 配置指定された割り込みハンドラ”である場合，警告メッセージを出力し，#pragma text 指令は無視します。なお，direct 配置指定についての詳細は「（7）割り込み／例外処理ハンドラ」を参照してください。

- #pragma text 指定した関数は，#pragma inline 指定や，最適化オプションによるインライン展開ができません。インライン展開指定は無視されます。

- セクション名を指定する場合，その名前は 256 文字以内にしてください。

（3）周辺 I/O レジスタへのアクセス

周辺 I/O レジスタとは，各デバイスで内蔵している周辺機能のためのレジスタです。デバイス定義の周辺 I/O レジスタ名を用いることにより，C 言語レベルで，周辺 I/O へアクセスできます。周辺 I/O レジスタ名は，C 言語ソース・プログラム中で，通常の符号なし外部変数（unsigned）のように扱うことができます。ただし，& 演算子で，周辺 I/O レジスタのアドレスを取得することはできません。

なお，指定可能なレジスタ名，属性などについては，各デバイスのユーザーズ・マニュアルを参照してください。

(a) アクセス方法

周辺 I/O レジスタ名を使用できるようにするには，次の #pragma 指令を記述することにより行います。

```
#pragma ioreg
```

“#pragma ioreg”を記述した C 言語ソース内では，それ以降，周辺 I/O レジスタ名を使用することができます。

上記の指令を行わずに，あるいは，適したデバイス名を指定せずに周辺 I/O レジスタ名を使用すると，「変数が未定義である」というエラーになります。

また，指定したレジスタ固有のアクセス属性に反した使用も，エラーになります。

次の場合，例 1 は正しいですが，例 2，例 3 はエラーとなります。

なお，次の例で，P0，P1，P2，RXBO，OVF0 は V850 マイクロコントローラの周辺 I/O レジスタを示します。このように，“レジスタ名”には，デバイス・ファイルで定義されている略号を指定します。

例 1.

```
#pragma ioreg
void func1(void){
    int i;
    P0 = 1;      /*P0 に書き込み */
    i = RXB0;    /*RXB0 から読み込み */
}
void func2(void){
    P1 = 0;      /*P1 に書き込み */
}
```

2.

```
void func(void){
    P1 = 0;      /* 未定義エラー */
}
```

3.

```
#pragma ioreg
void func(void){
    RXB0 = 1;    /*RXB0 の属性が読み込み専用のため，エラー */
}
```

(b) ビット・アクセス

CA850 では，周辺 I/O レジスタに対して，各ビットごとにアクセスすることもできます。“ビット番号”は，32 ビットのレジスタの場合，0 ～ 31 で次のように指定します。

```
レジスタ名．ビット番号 = ...
```

- ビット・アクセスの際の注意
  - ビット・アクセスで 0 と 1 以外の値を代入した場合，その値の 2 進数表現における最下位の値が設定されます（この際，メッセージは出力されません）。

例

```
#pragma ioreg
void func(void){
    P0.1 = 1;    /*P0 のビット 1 を 1 に設定 */
    P2.3 = 0;    /*P2 のビット 3 を 0 に設定 */
}
```

  - 各レジスタの持つフラグのビットにアクセスする場合，それぞれのビット名を用いてアクセスすることができます。ビット名には，デバイス・ファイルで定義されている名前を指定します。

例

```
#pragma ioreg
void func(void){
    OVF0 = 1;    /* ビット名 OVF0 のビットを 1 に設定 */
}
```

（4）アセンブラ命令の記述

CA850 では，次に示す形式において，C 言語ソース・プログラム中にアセンブラ命令が記述できます。

- asm 宣言
- #pragma 指令

挿入するアセンブラ命令でレジスタを使用する場合，必要な退避／復帰はプログラム内で行ってください。CA850 では行いません。

また，アセンブラ命令の記述は，関数の中に挿入することを推奨します。関数の外に記述した場合，次の制限があり，警告が出力されます。

- 関数部分とのコード出力順序が保証されない
- 関数の存在しないファイルでは出力されない

（a）asm 宣言

```
__asm( 文字列定数 );
, または
_asm( 文字列定数 );
```

- _asm 形式は，従来の言語仕様との互換性のために提供されている形式であり，コンパイラで -ansi オプションが指定された場合，_asm 形式に対して警告メッセージを出力し，関数呼び出しとして扱われます。-ansi オプションを指定する場合，__asm 形式を使用してください。

- コンパイラは，asm 宣言が指定された場合，指定された文字列定数[注]の後ろにニューライン（\ n）を付けてアセンブラに渡します。

注 指定された文字列定数は，一般の文字列定数と異なり，ニューライン以外の文字が後ろに続く“\”は後ろに続く文字自身を示します（ニューラインが後ろに続く“\”は誤りとなります）。

例

```
__asm("nop");
__asm (".str    \"string\\0\"");
```

- __asm，または _asm は宣言であり，文として扱われません。このため，次に示す例 1 のように，C 言語の文法上，宣言のみの記述が許されない構文ではエラーとなります。
そこで，例 2 のように“{}”で囲んで複合文としてください。

例 1.

```
if(i == 0)
__asm("mov     r11, r10");  /* 宣言のみのためエラー */
```

2.

```
if(i == 0){
    __asm("mov     r11, r10"); /* 複合文となるため記述可能 */
}
```

（b）#pragma 指令

この #pragma 指令で囲まれた範囲では，そのまま，アセンブラ命令が記述できます。複数のアセンブラ命令を記述する場合に有効です。

```
#pragma asm
アセンブラ命令
#pragma endasm
```

次に示す例 1 の記述は，例 2 の記述と同様の意味となります。

例 1.

```
int i;
void f( ){
#pragma asm
mov     r0, r10
st.w    r10, $_i
  :
#pragma endasm
}
```

2.

```
int i;
void f( ){
    __asm("mov     r0, r10");
    __asm("st.w    r10, $_i");
      :
}
```

“#pragma asm”から“#pragma endasm”までの記述は，そのままアセンブラに渡されます。

したがって，CA850 が内部的にアセンブラ命令を作成し，アセンブラを起動します。

そのため“#pragma asm”宣言後に，アセンブラ命令の疑似命令を使用することも可能です。また，アセンブラ命令で，C 言語ソース・プログラム中のローカル変数は使用できません。ローカル変数は，CA850 で“スタック”，または“レジスタ”に割り当てられるためで，インライン・アセンブラでは使用することはできません。

C 言語ソース・ファイルの #define で定義したシンボルも“#pragma asm”から“#pragma endasm”までの記述では使用できません。ファイル内で #define 定義したマクロをアセンブラ命令で展開したい場合，次の方法で回避してください。

- #pragma asm ～ #pragma endasm 指令内で .macro 命令を使用してマクロ定義する
- C 言語ソースから関数コールでアセンブラ命令を呼ぶ

マクロ定義せず，そのままアセンブラ命令で書くのも 1 つの方法です。

(5) 割り込みレベルの制御

(a) __set_il 関数

CA850 では，V850 マイクロコントローラの割り込みに対して，C 言語ソース上で，次の制御を行うことができます。

- 割り込み優先順位レベルの制御
- マスカブル割り込みの受け付けの許可／禁止（割り込みのマスク）

つまり，“割り込み制御レジスタ”を操作することができます。

“割り込み優先順位レベル”を制御する場合は，“__set_il 関数”を用いて次のように指定します。

```
__set_il(割り込みの優先順位レベル，“割り込み要求名”);
```

指定できる“割り込み要求名”は，デバイス・ファイルで定義されている“マスカブル割り込みの要求名”です。デバイス・ファイルで定義されている要求名を使用するので，この関数を使用する C 言語ソースでは，#pragma ioreg 指令を記述する必要があります。

“割り込みの優先順位レベル”として指定できる値は“1 ～ 8”の整数値です。V850 マイクロコントローラの割り込み優先順位レベルは“0 ～ 7 までの 8 段階”を指定するため，“V850 マイクロコントローラの割り込みの優先順位レベルを 5 にしたい”場合は，この関数で指定する割り込みの優先順位レベルは“6”と指定します。

例

```
__set_il(2, "INTP0");
```

上記の指定の場合，割り込み“INTP0”の優先順位レベルは“1”になります。

次に，“割り込みに対して，マスカブル割り込みの受け付けの許可／禁止”を制御する場合は，次のように指定します。

```
__set_il(マスカブル割り込みの許可／禁止，“割り込み要求名”);
```

“マスカブル割り込みの許可／禁止”に設定できるのは“-1”か“0”です。

表 3―20　マスカブル割り込みの許可／禁止

設定値	動作
-1	マスカブル割り込みの受け付け禁止（割り込みをマスク）
0	マスカブル割り込みの受け付け許可（割り込みのマスクを解除）

例

```
__set_il(-1, "INTP0");
```

上記のように指定すると，割り込み“INTP0”のマスカブル割り込みの受け付けを禁止します（INTP0 をマスクします）。

なお，__set_il 関数では，PSW（プログラム・ステータス・ワード）内の EP フラグ（例外処理中を示すフラグ）の操作は行いません。

（b）__set_il 関数と割り込み制御レジスタ

V850 マイクロコントローラの割り込み制御レジスタの構成は，次のようになっています。

7	6	5	4	3	2	1	0
xxIFn	xxMKn	0	0	0	xxPRn2	xxPRn1	xxPRn0

__set_il 関数を使用した場合は，“優先順位レベル”，または“割り込みマスク・フラグ”のいずれか一方の設定になります。したがって，__set_il 関数では，割り込み要求フラグの設定はできません。

割り込み要求名“INTP000”で，その割り込み制御レジスタ名が“P00IC0”の場合，割り込み優先順位レベルを 6 に設定するために，次のように記述します。

```
__set_il(7, "INTP000");
```

```
【出力コード】
ld.b    P00IC0, r1
andi    0xf8, r1, r1
ori     0x6, r1, r1
st.b    r1, P00IC0
```

つまり，“優先順位レベルの設定”である“下位 3 ビット（xxxPR02-xxxPR00）のみ”を変更するコードが出力されます。

割り込み要求名が“INTP000”で，その割り込み制御レジスタ名が“P00IC0”の場合，このマスカブル割り込みを許可するために，次のように記述します。

```
__set_il(0, "INTP000");
```

```
【出力コード】
clr1    6, P00IC0
```

つまり，“割り込みマスク・フラグのみ”を変更するコードが出力されます。

割り込み制御レジスタに，直接，値を書き込む場合には，“優先順位レベル”，“割り込みマスク・フラグ”，および“割り込み要求フラグ”のすべてに値が設定されます。

例　割り込み制御レジスタ名が“P00IC0”の場合

```
P00IC0 = 0x6;
```

```
【出力コード】
mov     0x6, r29
st.b    r29, P00IC0
```

つまり，次のとおりとなります。

- 優先順位レベルを 6 に設定
- マスカブル割り込みを許可
- 割り込み要求フラグ・クリア設定

(6) 割り込み禁止

CA850 では，C 言語ソースにおいて，マスカブル割り込みを禁止にすることができます。
マスカブル割り込みを禁止する方法には，大きく分けて次の二通りがあります。

- 関数内で部分的に割り込みを禁止する方法
- 関数全体の割り込みを禁止する方法

(a) 関数内で部分的に割り込みを禁止する方法

C 言語で記述した関数内で，部分的に割り込みを禁止する場合，アセンブラ命令の“di 命令”と“ei 命令”を使用することができますが，CA850 では C 言語ソースで割り込み制御を行うことのできる関数を用意しています。

表 3―21 ロード／ストア命令

割り込み制御関数	動作	CA850 の処理
__DI	すべてのマスカブル割り込みの受け付けを禁止します。	di 命令を生成
__EI	すべてのマスカブル割り込みの受け付けを許可します。	ei 命令を生成

例 __DI，__EI 関数の記述方法とその出力コード

【C 言語ソース】

```
void func1(void){
       :
    __DI();
    /* 割り込みを禁止して行いたい処理を記述 */
    __EI();
       :
}
```

【出力コード】

```
_func1:
    -- プロローグ・コード
       :
    di
    -- 割り込みを禁止して行いたい処理
    ei
       :
    -- エピローグ・コード
    jmp      [lp]
```

（b）関数全体の割り込みを禁止する方法

CA850 では，関数全体のマスカブル割り込み割り込みを禁止する“#pragma block_interrupt”指令を用意しています。

次のような書式で記述します。

```
#pragma block_interrupt 関数名
```

関数名は，C 言語記述の関数名を記述してください。たとえば "void func1 ( ) { }" という関数であれば "func1" と指定します。

上記で“関数名”で指定された関数に対し，マスカブル割り込みを禁止します。「（a）関数内で部分的に割り込みを禁止する方法」で説明したように，関数の最初に“__DI ( );”を，最後で“__EI ( );”を記述することもできますが，この場合だと，CA850 が出力する“プロローグ・コード”，“エピローグ・コード”に対してマスカブル割り込みを禁止／許可することができず，関数全体を完全に割り込み禁止にすることができません。

#pragma block_interrupt 指令を用いると，“プロローグ・コード”実行の直前にマスカブル割り込みが禁止され，“エピローグ・コード”実行直後にマスカブル割り込み割り込みが許可されます。そのため関数全体を完全に割り込み禁止にすることができます。

例 #pragma block_interrupt 指令の使用方法と，出力されるコードは次のとおりです。

【C 言語ソース】

```
#pragma block_interrupt func1
void func1(void){
       :
    /* 割り込みを禁止して行いたい処理を記述 */
       :
}
```

【出力コード】

```
_func1:
    di
    -- プロローグ・コード
       :
    -- 割り込みを禁止して行いたい処理
       :
    -- エピローグ・コード
    ei
    jmp     [lp]
```

(c) 関数全体の割り込み禁止時の注意事項

関数全体の割り込みを禁止にした場合の注意事項について，次に示します。

- 同じ関数に対して，割り込みハンドラ指定と #pragma block_interrupt 指定された場合，割り込みハンドラ指定の方が優先され，割り込み禁止の設定は無視されます。

- 割り込み禁止となっている関数内で，次の関数を呼び出した場合，その呼び出しからの復帰時に，割り込み許可状態になるため，注意が必要です。
  - #pragma block_interrupt 指定された関数
  - 関数の先頭で割り込み禁止をし，最後で割り込み許可している関数

- #pragma block_interrupt 指令は，関数の定義と同一ファイル内で，かつ，定義より前に記述してください。
  関数の定義より後ろに記述された場合，コンパイル時にエラーとなります。
  ただし，関数のプロトタイプ宣言順とは関係ありません。

- #pragma block_interrupt 指定された関数は，#pragma inline 指令を指定したり，最適化オプションでインライン展開指定ができません。インライン展開指定は無視されます。

- #pragma block_interrupt 指定しても，PSW（プログラム・ステータス・ワード）内の EP フラグ（例外処理中を示すフラグ）の操作を行うコードは出力されません。

(7) 割り込み／例外処理ハンドラ

CA850 では，C 言語で“割り込み”や“例外”が発生したときに呼ばれる“割り込みハンドラ”，“例外ハンドラ”を記述することができます。ここでは，その記述方法などについて説明します。

(a) 割り込み／例外の発生

V850 マイクロコントローラでは，割り込みや例外が発生すると，その割り込みや例外に対応したハンドラ・アドレスにジャンプします。“割り込み要因”と“ハンドラ・アドレス”は一対一で対応しており，ハンドラ・アドレスの集合を“割り込み／例外テーブル”と呼びます。

たとえば，V850ES/SG2 の場合の割り込み／例外テーブルは次のようになっています（先頭部分のみ掲載）。

表 3―22 割り込み／例外テーブル（V850ES/SG2）

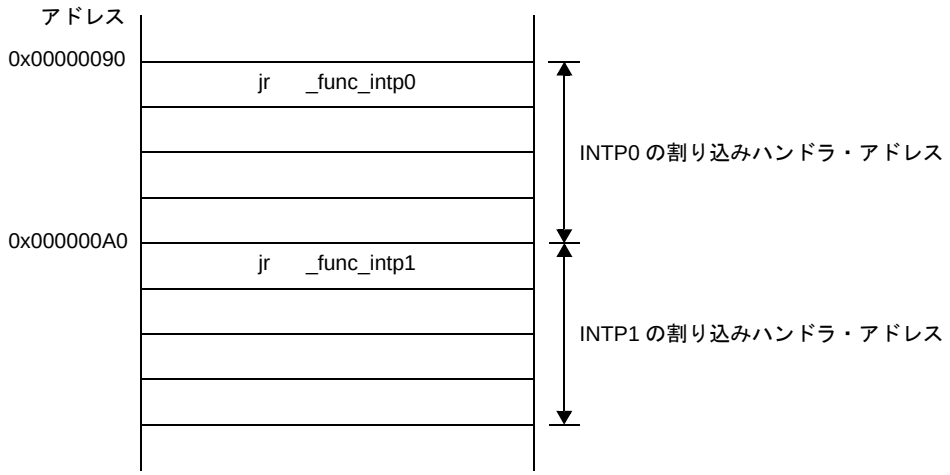
アドレス	割り込み名称	割り込みのトリガ
0x00000000	RESET	RESET 端子入力／内部要因からのリセット
0x00000010	NMI	NMI 端子有効エッジ入力
0x00000020	INTWDT2	WDT2 のオーバーフロー
0x00000040	TRAP0n	TRAP 命令
0x00000050	TRAP1n	TRAP 命令

アドレス	割り込み名称	割り込みのトリガ
0x00000060	LGOP/DBG0	不正命令コード／ DBTRAP 命令
0x00000080	INTLVI	低電圧検出
0x00000090	INTP0	外部割り込み端子入力エッジ検出（INTP0）
0x000000A0	INTP1	外部割り込み端子入力エッジ検出（INTP1）
0x000000B0	INTP2	外部割り込み端子入力エッジ検出（INTP2）
0x000000C0	INTP3	外部割り込み端子入力エッジ検出（INTP3）
:	:	:

なお，ハンドラ・アドレスの並びや，搭載している割り込みは，V850 の品種ごとに異なります。詳細は，使用する各デバイスのユーザーズ・マニュアルを参照してください。

各ハンドラ・アドレスは“16 バイト”の領域を持っており，割り込みが発生すると，その 16 バイトの領域に書かれた命令を実行します。したがって，16 バイトで収まる処理であれば，ハンドラ・アドレス内だけで処理し，収まらなければ，処理の書かれた関数（割り込み／例外ハンドラ）への分岐命令を記述することになります。

図 3―15 ハンドラ・アドレスのイメージ



V850ES/SG2 で INTP0 割り込みが入ると，0x90 番地にジャンプし，そこにあるコードを実行します。上記の例の場合，関数 func_intp0 に分岐するコードが書かれているので。そこへ分岐します。つまり，func_intp0 は INTP0 の割り込みハンドラということになります。

これらはアセンブリ言語ソース・レベルでの話になりますが，CA850 では，C 言語レベルで割り込み／例外処理を記述する場合，この点について留意することなく記述できるようになっています。具体的な記述方法は「（c）割り込み／例外ハンドラの記述方法」で説明します。

(b) 割り込み／例外発生時に行う必要のある処理

関数実行時やタスク実行時に割り込み／例外が入ると，即座に割り込み／例外処理を行う必要があります。そして割り込み／例外処理が終わると，割り込みが入った時点の関数やタスクに戻る必要があります。

注

したがって，割り込み／例外発生時には，そのときのレジスタ情報を保存し，割り込み／例外処理が終わった後は，そのレジスタ情報を復帰する必要があります。

注 リアルタイムOS使用時のタスクの場合，割り込み内でのシステム・コール発行によって，割り込みが入った時点のタスクに戻らないことがあります。詳細は各リアルタイムOSのユーザーズ・マニュアルを参照してください。

通常の関数のプロローグ／エピローグ・コードでは，レジスタ変数用レジスタの退避／復帰を行います。レジスタ変数用レジスタは次のとおりで，必要のあるものに関して退避／復帰を行います。

表3―23 レジスタ変数用レジスタ

レジスタ・モード	レジスタ変数用レジスタ
22レジスタ・モード	r25，r26，r27，r28，r29
26レジスタ・モード	r23，r24，r25，r26，r27，r28，r29
32レジスタ・モード	r20，r21，r22，r23，r24，r25，r26，r27，r28，r29

割り込み／例外ハンドラに移る場合は，レジスタ変数用レジスタのほかに，割り込み／例外ハンドラ用のスタック・フレームとして，次のレジスタの必要のあるものに関して退避します。

表3―24 割り込み／例外ハンドラ用のスタック・フレーム

レジスタ・モード	割り込み／例外発生時に退避／復帰するレジスタ
22レジスタ・モード	r1，r6，r7，r8，r9，r10，r11，r12，r13，r14，r31（lp），CTPC【V850E】，CTPSW【V850E】
26レジスタ・モード	r1，r6，r7，r8，r9，r10，r11，r12，r13，r14，r15，r16，r31（lp），CTPC【V850E】，CTPSW【V850E】
32レジスタ・モード	r1，r6，r7，r8，r9，r10，r11，r12，r13，r14，r15，r16，r17，r18，r19，r31（lp），CTPC【V850E】，CTPSW【V850E】

また，多重割り込み／例外が発生した場合は，レジスタ変数用レジスタのほかに，多重割り込み／例外ハンドラ用のスタック・フレームとして，次のレジスタの必要のあるものに関して退避します。

表3―25 多重割り込み／例外ハンドラ用のスタック・フレーム

レジスタ・モード	多重割り込み／例外発生時に退避／復帰するレジスタ
22レジスタ・モード	r1，r6，r7，r8，r9，r10，r11，r12，r13，r14，r31（lp），EIPC，EIPSW，CTPC【V850E】，CTPSW【V850E】

レジスタ・モード	多重割り込み／例外発生時に退避／復帰するレジスタ
26 レジスタ・モード	r1，r6，r7，r8，r9，r10，r11，r12，r13，r14，r15，r16，r31（lp），EIPC，EIPSW，CTPC【V850E】，CTPSW【V850E】
32 レジスタ・モード	r1，r6，r7，r8，r9，r10，r11，r12，r13，r14，r15，r16，r17，r18，r19，r31（lp），EIPC，EIPSW，CTPC【V850E】，CTPSW【V850E】

上記のレジスタの用途は，次のとおりです。

表 3―26　レジスタの用途

レジスタ	用途
r1	アセンブラ予約レジスタ
r6-r9	引数用レジスタ（関数の引数をセットするためのレジスタ）
r10-r19	作業用レジスタ（CA850 がコード生成時に使用するレジスタ）
r31	リンク・ポインタ
CTPC【V850E】	CALLT 命令実行時のプログラム・カウンタ（PC）
CTPSW【V850E】	CALLT 命令実行時のプログラム・ステータス・ワード（PSW）
EIPC	割り込み／例外処理時のプログラム・カウンタ（PC）
EIPSW	割り込み／例外処理時のプログラム・ステータス・ワード（PSW）

割り込み／例外処理が終わると，退避したレジスタを復帰するコードを出力し，最後に reti 命令を出力します。この命令の発行により，割り込み／例外処理が終了したことを V850 に通知します。

CA850 では，“レジスタの退避／復帰”，および“reti 命令の出力”を「(c) 割り込み／例外ハンドラの記述方法」に従って記述すると自動的に出力します。“レジスタの退避／復帰”については，「表 3―27 割り込みのレジスタの退避／復帰処理」に従って出力します。つまり，ユーザは，この点について留意する必要なく，割り込み／例外ハンドラ本体処理の記述に専念することができます。

表 3―27　割り込みのレジスタの退避／復帰処理

レジスタ名		レジスタ	説明
アセンブラ予約レジスタ		r1	割り込み時には，必ず退避／復帰します。
引数用レジスタ		r6-r9	割り込み要因が TRAP0 ／ TRAP1 の場合には，r6 は，必ず退避／復帰します。 関数コール（ランタイム関数を含む）が存在した場合には，退避／復帰します。 関数コールが存在しない場合には，割り込み関数で使用していれば，退避／復帰します。
作業用レジスタ	22 レジスタ・モード	r10-r14	関数コールが存在した場合には，退避／復帰します。 関数コールが存在しない場合には，割り込み関数で使用していれば退避／復帰します。
	26 レジスタ・モード	r10-r16	
	32 レジスタ・モード	r10-r19	

レジスタ名		レジスタ	説明
レジスタ変数用レジスタ	22 レジスタ・モード	r25-r29	通常の関数と同様に，必要に応じて退避／復帰します。
	26 レジスタ・モード	r23-r29	
	32 レジスタ・モード	r20-r29	
リンク・ポインタ		r31（lp）	関数コール（ランタイム関数を含む）が存在した場合には，退避／復帰します。 関数コールが存在しない場合には，退避／復帰しません。
割り込み関連システム・レジスタ		EIPCE，EIPSW	多重割り込み __multi_interrupt 修飾子を使用した割り込み関数では，必ず退避／復帰します。 __interrupt 修飾子の場合には，退避／復帰しません。
callt 命令関連システム・レジスタ【V850E】		CTPC，CTPSW	V850E/V850ES/V850E2 コアのデバイスを指定してコンパイルしている割り込み関数では，必ず退避／復帰します。

（c）割り込み／例外ハンドラの記述方法

割り込み／例外ハンドラの記述上の形態は，通常の C 言語関数と変わりませんが，C 言語で記述した関数を，CA850 に対して“割り込み／例外ハンドラ”として認識させる必要があります。CA850 では，割り込み／例外ハンドラの指定を“#pragma interrupt 指令”および“__interrupt 修飾子”，または“#pragma interrupt 指令”，および“__multi_interrupt 修飾子”で行います。

- 割り込み／例外ハンドラを指定する場合

#pragma interrupt	割り込み要求名	関数名	配置方法
__interrupt	関数定義，または関数宣言		

- 多重割り込み／例外ハンドラを指定する場合

#pragma interrupt	割り込み要求名	関数名	配置方法
__multi_interrupt	関数定義，または関数宣言		

関数名は，C 言語記述の関数名を記述してください。たとえば，"void func1 ( ) { }" という関数であれば "func1" と指定します。

“多重割り込みハンドラの指定”とは“「多重割り込みされる」ことを許可する関数を指定”ということです。“多重割り込みする関数を指定する”ということではありません。

- 割り込み要求名

デバイス・ファイルに登録されている割り込み要求名を指定できます。デバイス・ファイルに登録されている割り込み要求名は，各デバイスのユーザーズ・マニュアルの“割り込み要求名”に書かれてある文字列になりますので，そちらを参照してください。

なお，ノンマスカブル割り込み（NMI）も，この方法で指定できますが，リセット割り込み（RESET）は指定できません。リセット時の処理に関しては，アセンブラ命令で記述する必要があります。リセット時の処理は，スタート・アップ・ルーチンに記述するのが一般的であるため，詳細は「第 7 章　スタートアップ」を参照してください。

- 関数名

割り込み／例外ハンドラとする関数名を指定します。ここでは“C 言語での関数名”を記述してください。“void func1 (void)”という関数を指定する場合は，関数名に“func1”と指定します。

- 配置方法

ハンドラ・アドレスに関数本体を直接配置するか，それとも割り込み／例外ハンドラ関数への分岐命令だけを配置するかを指定します。直接配置するときだけに“direct”を指定します。直接配置しない場合は，“配置方法”には何も記述しないでください。direct 指定を行うことによって，指定した割り込み要因のハンドラ・アドレスからすべて配置されますが，その結果，以降のハンドラ・アドレスの領域も使用されることになる可能性があるため注意が必要です。
また，direct 指定をする場合は，関数定義より後ろに #pragma interrupt 指令を記述するとコンパイル時にエラーとなります。必ず関数定義より前で行ってください。

次に，#pragma interrupt 指定と __interrupt 修飾子，__multi_interrupt 修飾子の役割について説明します。

- #pragma interrupt 指令

#pragma interrupt で指定された“割り込み要求名”に対応するハンドラ・アドレスに，指定された関数への分岐命令（jr）を配置します。-Xj オプション指定時は，r1 レジスタをスタックに退避する命令と指定された関数への分岐命令（jmp）を配置します。

- __interrupt 修飾子

__interrupt 修飾子のついた関数に，割り込み／例外ハンドラとしてのレジスタの退避／復帰処理を加え，最後に reti 命令を加えます。-Xj オプション指定時は，r1 レジスタに関しては，退避処理はハンドラ・アドレスに出力されているので，関数では復帰処理のみ出力されます。

- __multi_interrupt 修飾子

__multi_interrupt 修飾子のついた関数に，割り込み／例外ハンドラとしてのレジスタの退避／復帰処理を加え，EIPC，EIPSW レジスタの退避／復帰処理を加えます。また，最後に reti 命令を加えます。-Xj オプション指定時は，r1 レジスタに関しては，退避処理はハンドラ・アドレスに出力されているので，関数では復帰処理のみ出力されます。

“#pragma interrupt”と“__interrupt 修飾子，__multi_interrupt 修飾子”を同時に指定する場合は，次のコードが出力され，割り込み／例外処理として完全なハンドラが完成します。

- ハンドラ・アドレスに，指定された割り込み／例外ハンドラへの分岐命令を配置
- 割り込み／例外ハンドラとしてのレジスタの退避／復帰処理（__multi_interrupt 修飾子指定の場合はさらに EIPC，EIPSW の退避／復帰処理）を追加

- 割り込み／例外ハンドラの最後に reti 命令を追加

また，この場合，関数定義と #pragma interrupt 指令を別ファイルに記述することもできます。また，記述順序も問いません。ただし，配置方法で direct 指定した場合は，別ファイルに記述できません。

__interrupt 修飾子，__multi_interrupt 修飾子のみを指定する場合は，次のコードが出力されます。

- 割り込み／例外ハンドラとしてのレジスタの退避／復帰処理（__multi_interrupt 修飾子指定の場合はさらに EIPC，EIPSW の退避／復帰処理）を追加
- 割り込み／例外ハンドラの最後に reti 命令を追加

つまり，割り込み／例外ハンドラとしての起動ができる形の関数になりますが，#pragma interrupt 指令で出力される“ハンドラ・アドレスに割り込み／例外ハンドラへの分岐命令を配置する処理”は行われません。

例 割り込み要求名“INTP0”に対する割り込みハンドラを“void intp0_func (void)”とし，direct 指定せず，多重割り込みを許可しない場合の #pragma interrupt 指定は次のようになります。

```
#pragma interrupt   INTP0   intp0_func
__interrupt
void intp0_func(void){
       :
    割り込み処理本体
       :
}
```

次に，割り込みハンドラとして指定できる“関数の型”について説明します。

- 関数の型

マスカブル割り込み，NMI 割り込みのハンドラに関しては，次のようになります。

void func (void) 型

引数が void 型，戻り値が void 型の関数になります。

ソフトウエア例外処理（トラップ）ハンドラについては，次のようになります。

void func (unsigned int) 型

引数には割り込み要因レジスタ（ECR）の EICC（例外コード）がセットされます。これらの型で指定しなければ，コンパイル時にエラーとなります。ソフトウエア例外処理関数については，次の項目を参照してください。

- ソフトウエア例外処理（トラップ処理）ハンドラ

ソフトウエア例外処理（トラップ処理）を使用する場合，V850 マイクロコントローラの仕様上，エントリ・ポイントは，“TRAP0（0x40 番地）”と“TRAP1（0x50 番地）”の 2 箇所になります。ソフトウエア例外“trap 0x00 ～ trap 0x0f”が入ったときは TRAP0（0x40 番地）へ，“trap 0x10 ～ trap0x1f”が入ったときは，TRAP1（0x50 番地）へ分岐します。その際に，“ソフトウエア例外コード”として TRAP0 のときは“0x40 ～ 0x4f”の値が，TRAP1 のときは“0x50 ～ 0x5f”の値が割り込み要因レジスタ（ECR）にセットされます。

表 3―28　トラップ命令とソフトウエア例外コード

トラップ命令	ソフトウエア例外コード
trap　0x00	0x40
trap　0x01	0x41
trap　0x02	0x42
:	:
trap　0x0a	0x4a
trap　0x0b	0x4b
:	:
trap　0x10	0x50
trap　0x11	0x51
trap　0x12	0x52
:	:
trap　0x1e	0x5e
trap　0x1f	0x5f

TRAP0，TRAP1 に対するソフトウエア例外処理を記述する場合，その関数は，引数を 1 つ持ち，変数の型は“unsigned int 型”になります。その引数には割り込み要因レジスタ（ECR）にセットされた“ソフトウエア例外コード”が入ります。TRAP0 のときは“0x40 ～ 0x4f”の値が，TRAP1 のときは“0x50 ～ 0x5f”の値のいずかになります。ハンドラ内では，これらの値によって場合分けした処理を記述することになります。

```
#pragma interrupt   TRAP0   trap0_func
__interrupt
void trap0_func(unsigned int codenum){
      :
    例外コード別に場合分けし，その処理を記述
      :
}
```

（d）割り込み／例外ハンドラの記述時の注意事項

- __multi_interrupt 修飾子で指定する“多重割り込みハンドラの指定”とは，“「多重割り込みされる」ことを許可する関数を指定”ということです。“多重割り込みする関数を指定する”ということではありません。
- __multi_interrupt で多重割り込みを許可するハンドラとして定義されても，割り込みハンドラ起動時には，割り込みは許可されていません。そのため，必ず割り込みハンドラの中で割り込み許可命令（__EI など）を発行し，最後に割り込み禁止命令（__DI など）を発行してください。最後に割り込み禁止命令を発行しなかった場合，その後のレジスタの復帰部分で割り込みを受け付けてしまう可能性があり，その場合は暴走につながりますので注意が必要です。
- #pragma interrupt 指令で，リセット割り込みは指定できません。

```
#pragma interrupt   RESET   reset_func  /* エラー */
```

上記のように記述をすると，コンパイル時にエラーになります。リセット時の処理に関しては，アセンブラ命令で記述する必要があります。

リセット時の処理は，スタート・アップ・ルーチンに記述するのが一般的であるため，詳細は「第 7 章 スタートアップ」を参照してください。

- 多重割り込みを行うハンドラとして指定する関数には，__multi_interrupt 修飾子を指定して下さい。この場合，EIPC，EIPSW を退避／復帰するコードを出力します。__multi_interrupt 修飾子を指定しない割り込みハンドラは，EIPC，EIPSW を退避／復帰するコードを出力しません。
- #pragma interrupt 指令，__multi_interrupt 修飾子では，多重例外や多重 NMI には対応していません。多重例外や多重 NMI を行う場合は，必要となるシステム・レジスタ（FEPC，FEPSW など）の退避／復帰を行うコードを追加してください。必要となるシステム・レジスタについては，各デバイスのユーザーズ・マニュアルを参照してください。
- リンク・ディレクティブ・ファイルへの割り込みハンドラ・アドレスの追加記述は，ユーザで行う必要はありません。CA850 が内部的に出力します。
- 1 つの割り込み要求名に対し，異なる関数を複数指定することはできません。
- 同じ関数に __interrupt 修飾子，__multi_interrupt 修飾子の両方は指定できません。
- __interrupt 修飾子，__multi_interrupt 修飾子指定なしで関数を定義したあと，__interrupt 修飾子，__multi_interrupt 修飾子指定ありで関数宣言すると，コンパイル時にエラーになります。
- 割り込み／例外ハンドラとして指定された関数はインライン展開できません。#pragma inline 指定しても無視されます。
- 割り込み／例外ハンドラとして指定された関数は割り込み禁止となっているため，#pragma block_interrupt 指定されていても無視されます。
- 割り込み／例外ハンドラとして指定された関数は，通常の関数呼び出しで呼び出すことはできません。ただし，別ファイルから呼び出された場合は，コンパイラでチェックできません。
- 割り込み／例外ハンドラからアセンブラ命令を呼び出し，「表 3―23　レジスタ変数用レジスタ」，および「表 3―24　割り込み／例外ハンドラ用のスタック・フレーム」に示されたレジスタを使用する場合，退避／復帰処理を記述する必要があります。また，SP（r3），GP（r4），TP（r5），EP（r30）を書き換える場合も，退避／復帰処理を記述する必要があります。
- #pragma interrupt 指令，__interrupt 修飾子，__multi_interrupt 修飾子機能では，外部割り込みコントローラに対する処理終了通知（EOI コマンド）は発行していません。必要な場合はユーザで実行してください。
- 多重割り込みの最後は，EIPC，EIPSW の復帰コードが入るので，割り込み禁止にしてください。
- direct 指定をしない場合，ハンドラ・アドレスには“割り込み／例外ハンドラへの分岐命令”が配置されますが，その場合 CA850 はコード効率の面から“jr 命令”を出力しています。ただし，jr 命令で分岐できる範囲には限界があり，jr 命令から± 21 ビット内になります。もし，割り込みハンドラ本体へ jr 命令で分岐できる範囲になかった場合，リンカでエラーになります。その場合は，コンパイル・オプション“-Xj オプション”を指定することにより，jr 命令を jmp 命令に置き換える処置をしてください。

(e) 割り込み／例外ハンドラの記述例

割り込み／例外ハンドラの記述例を次に示します。

ただし，割り込み要求名は，デバイスによって異なりますので，各デバイスのユーザーズ・マニュアルを参照してください。

例 1. ノンマスカブル割り込みの場合

```
#pragma interrupt   NMI     func1   /* ノンマスカブル割り込み */
__interrupt
void func1(void){
   :
}
```

2. トラップの場合

```
#pragma interrupt   TRAP0   func2   /* トラップ 0*/
__interrupt
void func2(unsigned int num){
    switch(num){    /* 例外コード別に場合分け */
       :
    }
}
```

3. #pragma interrupt と __interrupt 修飾子を別ファイルとする場合

【a. c】

```
__interrupt         /*__interrupt 指定 */
void func1(void){
   :
}
```

【b. c】

```
#pragma interrupt   NMI func1   /* 定義より後ろや別ファイルに記述可能 */
```

4. 多重割り込み指定の場合

```
#pragma interrupt   INTP0   func1
__multi_interrupt   /* 多重割り込み関数指定 */
void func1(void){
   :
}
```

(8) インライン展開

CA850 では，関数ごとのインライン展開ができます。ここでは，インライン展開の指定について説明します。

(a) インライン展開とは

インライン展開とは，関数呼び出し部分に関数本体を展開することを言います。これにより，関数呼び出しによるオーバーヘッドが小さくなり，また最適化の可能性が高められることから，実行速度向上を図ることができます。

ただし，インライン展開を行うと，オブジェクト・サイズは増大することになります。

インライン展開したい関数は，#pragma inline で指定します。

```
#pragma inline  関数名 [, 関数名 , ...]
```

関数名は，C 言語記述の関数名を記述してください。たとえば，"void func1 ( ) { }" という関数であれば "func1" と指定します。また，関数名は“,”（カンマ）で区切って複数指定することができます。

```
#pragma inline  func1, func2
void    func1(){...}
void    func2(){...}
void func(void){
    func1();    /* インライン展開対象 */
    func2();    /* インライン展開対象 */
}
```

(b) インライン展開の条件

#pragma inline 指定された関数をインライン展開するためには，最低限次の条件が必要となります。

ただし，“サイズ優先最適化（-Os)”，“実行速度優先最適化（-Ot)”以外の最適化を指定していた場合，CA850 の内部処理の関係により，次の条件を満たしていてもインライン展開されない場合があります。

- インライン展開を“する関数”と“される関数”を同一ファイル内に記述する

  インライン展開を“する関数”と“される関数”，つまり，“関数呼び出し”と“関数定義”は“同一ファイル内”に存在しなければなりません。別の C 言語ソースに書かれてある関数をインライン展開することはできません。この場合，CA850 はエラーも警告メッセージも出力せず，インライン展開指定を無視します。

- #pragma inline を“関数定義より前”に記述する

  pragma inline が，関数定義よりも後ろに記述されていた場合，警告を出力してインライン展開指定を無視します。ただし，関数のプロトタイプ宣言との記述順序は問いません。次に例を示します。

例

【インライン展開指定：有効】	【インライン展開指定：無効】
`#pragma inline func1, func2`	`void func1(); /* プロトタイプ宣言 */`
`void func1(); /* プロトタイプ宣言 */`	`void func2(); /* プロトタイプ宣言 */`
`void func2(); /* プロトタイプ宣言 */`	`void func1(){…} /* 関数定義 */`
`void func1(){…} /* 関数定義 */`	`void func1(){…} /* 関数定義 */`
`void func2(){…} /* 関数定義 */`	`#pragma inline func1, func2`

- インライン展開する関数の“呼び出し”と“定義”の間で，“引数の数”を同じにする

  インライン展開する関数の“呼び出し”と“定義”の間で“引数の数”が違う場合，警告メッセージを出力してインライン展開指定を無視します。

- インライン展開する関数の“呼び出し”と“定義”の間で，“戻り値の型”や“引数の型”を同じにする

  インライン展開する関数の“呼び出し”と“定義”の間で，“戻り値の型”や“引数の型”が異なる場合，警告メッセージを出力してインライン展開指定を無視します。ただし，型変換ができる場合は，次のように変換してインライン展開します。

  - 戻り値の型は“呼び出し側の型”
  - 引数は“関数定義の型”

  ただし，“-ansi オプション”を指定していたときは，型変換を行わずエラーを出力します。

- インライン展開する関数のサイズ，および，使用スタック・サイズは，大きすぎないようにする

  インライン展開する関数のサイズ，および使用スタック・サイズが大きい場合，エラーも警告メッセージも出力せず，インライン展開指定を無視します。ここでいう“サイズ”とは“中間言語”でのサイズを指し，実際のオブジェクトのサイズとは異なります。CA850 では，これらのサイズの上限を変更することができます。

  中間言語における“関数のサイズ”は，次のオプションになります。

  -Wp,-N*num*

  中間言語における“関数の使用スタック・サイズ”は，次のオプションになります。

  -Wp,-G*num*

  また，次のオプションで中間言語における各関数の“サイズ”，“使用スタック・サイズ”を確認することができます。

  -Wp,-l

  このオプションでサイズ指定の目安にすることができます。

- インライン展開する関数の引数は“可変個”にしない

  引数が“可変個”の関数にインライン展開指定した場合，エラーも警告メッセージも出力せず，インライン展開指定を無視します。

- “再帰関数”はインライン展開できない

  自分自身を呼び出す“再帰関数”をインライン展開指定した場合，エラーも警告メッセージも出力せず，インライン展開指定を無視します。ただし，関数呼び出しが複数ネストし，そのネストした中に自分自身を呼び出すコードが存在した場合，インライン展開する場合があります。

- “割り込みハンドラ”はインライン展開できない

  #pragma interrupt，__interrupt，__multi_interrupt 指定で記述された関数は“割り込みハンドラ”として認識されますが，この関数に対してインライン展開指定した場合，警告メッセージを出力して，インライン展開指定を無視します。

- リアルタイム OS の“タスク”はインライン展開できない

  #pragma rtos_task で指定された関数は，リアルタイム OS の“タスク”として認識されますが，この関数に対してインライン展開指定した場合，警告メッセージを出力して，インライン展開指定を無視します。

- #pragma block_interrupt 指定で関数内を割り込み禁止にすると，インライン展開できない

  #pragma block_interrupt で，関数内を割り込み禁止として宣言された関数に対してインライン展開指定した場合，警告メッセージを出力して，インライン展開指定を無視します。

(c) オプションによるインライン展開の制御

“コンパイラによるインライン展開を抑止したい”など，インライン展開を制御したい場合があります。そういった場合，オプションによって制御することができます。制御できる内容とそのオプションは，次のとおりです。

ただし，実行速度優先最適化（-Ot）を指定していた場合は「(d) 実行速度優先最適化とインライン展開」を参照してください。

- 1 回だけ参照される static 関数を，すべてインライン展開したい場合

  このオプションを指定した場合は“最適化指定”や“#pragma inline の有無”に関わらず，1 回だけ参照される static 変数をインライン展開します。

  ただし，“サイズ優先最適化（-Os）”以外の最適化を指定していた場合，CA850 の内部処理の関係により，-Wp,-S オプションを指定してもインライン展開されない場合があります。

-Wp,-S

- すべての関数に対してインライン展開を抑止したい場合

  この場合，-Wp,-S 指定や #pragma inline があっても，インライン展開を抑止します。

```
-Wp,-no_inline
```

(d) 実行速度優先最適化とインライン展開

CA850 の最適化の 1 つである“実行速度優先最適化（-Ot)”をオプション指定した場合，CA850 はインライン展開を最適化手段の 1 つとします。

したがって，実行速度優先最適化（-Ot）が指定されていれば，#pragma inline でインライン展開指定した関数“以外”でも「(b) インライン展開の条件」をクリアしていれば，CA850 が適切な関数を選択し，インライン展開を行います。

しかし，“コンパイラによるインライン展開を抑止したい”など，インライン展開を制御したい場合があります。そういった場合，オプションによって制御することができます。制御できる内容とそのオプションは，次のとおりです。

- 実行速度優先最適化（-Ot）を指定しているが，すべての関数に対してインライン展開を抑止したい場合

  この場合，-Wp,-S 指定や #pragma inline があっても，インライン展開を抑止します。

```
-Wp,-no_inline
```

- 実行速度優先最適化（-Ot）を指定しているが，#pragma inline で指定した関数のみをインライン展開したい場合

  この場合，インライン展開を指定した関数は「(b) インライン展開の条件」をクリアしている必要があります。

```
-Wp,-inline
```

(e) オプション指定によるインライン展開動作の違いの例

#pragma inline 指定とオプション指定による“インライン展開動作の違い”の例は，次のようになります。

“-Os（サイズ優先最適化）指定”（-Ot 以外）

```
#pragma inline  func0
void    func0(){...}    /*#pragma inline 指定により，インライン展開の条件が合致すれば
                          展開 */
void    func1(){...}    /* 展開しない */
void    func2(){...}    /* 展開しない */
```

“-Ot（実行速度優先最適化）指定”

```
#pragma inline  func0
void    func0(){...}    /*-Ot 指定により，インライン展開の条件が合致すれば展開 */
void    func1(){...}    /*-Ot 指定により，インライン展開の条件が合致すれば展開 */
void    func2(){...}    /*-Ot 指定により，インライン展開の条件が合致すれば展開 */
```

“-Ot（実行速度優先最適化）”+“-Wp,-inline（#pragma inline 指定関数のみをインライン展開）指定”

```
#pragma inline  func0
void    func0(){...}    /*#pragma inline 指定により，インライン展開の条件が合致すれば
                          展開 */
void    func1(){...}    /*-Wp,-inline 指定により，#pragma inline 指定がないので展開し
                          ない */
void    func2(){...}    /*-Wp,-inline 指定により，#pragma inline 指定がないので展開し
                          ない */
```

備考 1. CA850 では，#pragma inline によりインライン展開指定された関数は，静的関数として扱いません。静的関数とするには，明示的に static 指定をする必要があります。

2. デバッグの際，インライン展開をした関数に対して C 言語ソース・レベルでブレークポイントを設定することはできません。

(9) リアルタイム OS 対応機能

CA850 は，V850 マイクロコントローラ用リアルタイム OS“RI850V4”を使用したシステムを構築する場合を考慮し，プログラミング記述性向上と，コード削減の機能を備えています。

(a) タスクの記述

リアルタイム OS を使用したアプリケーションは“タスク”を処理単位とします。リアルタイム OS は，そのタスク内で発行された“システム・コール”や“割り込み処理”をきっかけとして，タスクのスケジューリングを行います。タスクを切り替えるとき（コンテキストを切り替えるとき）のレジスタの退避／復帰作業は，リアルタイム OS が行うため，一般の関数としてのプロローグ処理／エピローグ処理とは異なります。

したがって，CA850 が，関数呼び出し時に生成するプロローグ処理／エピローグ処理は，タスクでは実行されないことになります。

記述された関数を“タスク”とする場合，関数呼び出し時のプロローグ処理／エピローグ処理を削除することにより，コード削減がはかれますが，C 言語の記述上，“一般の関数”と“タスク”は区別がつきません。そこで CA850 では，関数を“リアルタイム OS のタスク”と認識させるために，次の #pragma 指令を用意しています。

```
#pragma rtos_task    関数名
```

これにより，“関数名”で指定された関数を，リアルタイム OS のタスクとして認識させることができます。“関数名”は，C 言語記述の関数名を指定します。例として，“void func1 (int inicode) { }”という関数をタスクとする場合は，次のように記述します。

例

```
#pragma rtos_task    func1
```

#pragma rtos_task を指定すると，具体的には次のような効果が得られます。

- 通常の関数で出力される“プロローグ／エピローグ処理”を行いません。具体的には次のようなコードを出力しません。
  - レジスタ変数用レジスタの退避／復帰
  - リンク・ポインタ（lp）の退避／復帰
  - 戻り先へのジャンプ

- システム・コール“ext_tsk”を，定義済みの関数として使用することができます。
  特にアプリケーション内でプロトタイプ宣言しなくても，このシステム・コールを使用することができます。#pragma rtos_task の記述以降であれば，タスク指定した関数以外でも，同様に呼び出すことができます。
  このシステム・コールを呼び出したとき，コード・サイズ削減のために“jr 命令”を使用したコードを出力します。システム・コール“ext_tsk”本体が，jr 命令で分岐可能な範囲にない場合は，リンカ（ld850）でエラーになります。この場合は，次の処置が必要になります。
  - リンク・ディレクティブでメモリ配置を変更する
  - アセンブリ言語ソースで jmp 命令による分岐に切り替える
  - far jump 指定する

また，#pragma rtos_task 指定した場合，次のような注意事項があります。

- 関数と同じように，タスクを呼び出すことはできません。ただし，別のファイルで呼び出された場合はチェックされません。また，関数として呼び出すことができないため，インライン展開することができません。したがって，#pragma rtos_task 指定された関数に，#pragma inline 指定しても，#pragma inline 指定は無視されます。
- “#pragma rtos_task 関数名”を，同じファイル内の関数定義よりも後ろに記述した場合，エラーとなります。また，“#pragma rtos_task 関数名”を記述したあと，そのファイル内に関数定義を記述しない場合は，その関数に対する #pragma 指令は無視されます。
- #pragma rtos_task 指定された関数は，通常の割り込み／例外ハンドラ（「（7）割り込み／例外処理ハンドラ」を参照）として指定することはできません。

なお，リアルタイム OS の機能については，各リアルタイム OS のユーザーズ・マニュアルを参照してください。

（10）組み込み関数

CA850 では，アセンブラ命令の一部を“組み込み関数”として C 言語ソースに記述することができます。ただし，“アセンブラ命令そのもの”を記述するのではなく，CA850 で用意した関数の形式で記述します。これらの関数を使用した場合，出力コードは通常の関数呼び出しを行なわず，対応するアセンブラ 1 命令を出力します。

以下に，関数として記述できる命令を示します。

表 3―29　組み込み関数

アセンブラ命令	機能	組み込み関数
di	割り込み制御（di ／ ei）	__DI(); __EI();
ei		
nop	nop	__nop();
halt	halt	__halt();
satadd	飽和加算（satadd）	long a, b; long __satadd(a, b);
satsub	飽和減算（satsub）	long a, b; long __satsub(a, b);
bsh	ハーフワード・データのバイト・スワップ（bsh）【V850E】	long a; long __bsh(a);
bsw	ワード・データのバイト・スワップ（bsw）【V850E】	long a; long __bsw(a);
hsw	ワード・データのハーフワード・スワップ（hsw）【V850E】	long a; long __hsw(a);
sxb	バイト・データの符号拡張（sxb）【V850E】	char a; long __sxb(a);
sxh	ハーフワード・データの符号拡張（sxh）【V850E】	short a; long __sxh(a);
mul	mul 命令を用いて乗算結果の上位 32 ビットを変数に代入する命令【V850E】	long a, b; long __mul32(a, b);
mulu	mulu 命令を用いて符号なし乗算結果の上位 32 ビットを変数に代入する命令【V850E】	unsigned long a, b; unsigned long __mul32u(a, b);
sasf	論理左シフト付きフラグ条件の設定（sasf）【V850E】	long a; unsigned int b; long __sasf(a, b);

注意 1.　【V850E】マークは，V850Ex コア専用であることを示します。

2.　組み込み関数と同名の関数を定義して使用することはできません。
同名の関数を呼び出そうとしても，コンパイラが用意している組み込み関数処理を優先します。

（a）割り込み制御（di ／ ei）

割り込み制御命令（di ／ ei）の記述例を示します。

例

```
void func(void){
       :
    __DI();
       :   /* 割り込みを禁止して行いたい処理 */
    __EI();
       :
}
```

```
【出力コード】
_func:
    -- プロローグ・コード
       :
    di
       :   -- 割り込みを禁止して行いたい処理
    ei
       :
    -- エピローグ・コード
    jmp     [lp]
```

（b）nop

nop 命令の記述例を示します。

例

```
void func(void){
       :
    __nop();
       :
}
```

```
【出力コード】
_func:
       :
    nop
       :
```

（c）halt

halt 命令の記述例を示します。

例

```
void func(void){
       :
    __halt();
}
```

```
【出力コード】
_func:
       :
    halt
```

（d）飽和加算（satadd）

飽和加算命令（satadd）の記述例を示します。

例

```
void func(void){
    long    a, b, c;
      :
    c = __satadd(a, b); /*a と b の飽和演算，結果を c に格納 */
      :
}
```

```
【出力コード】
_func:
       :
    ld.w    -4 + .A2[sp], r10    -- 変数 a をロード
    ld.w    -8 + .A2[sp], r11    -- 変数 b をロード
    satadd  r11, r10             -- 飽和減算 ( a + b )
    st.w    r10, -12 + .A2[sp]   -- 飽和演算結果を変数 c にストア
      :
```

(e) 飽和減算（satsub）

飽和減算命令（satsub）の記述例を示します。

例

```
void func(void){
    long    a, b, c;
      :
    c = __satsub(a, b); /*a と b の飽和演算，結果を c に格納 (c = a - b)*/
      :
}
```

```
【出力コード】
_func:
      :
    ld.w    -4 + .A2[sp], r10   -- 変数 a をロード
    ld.w    -8 + .A2[sp], r11   -- 変数 b をロード
    satsub  r11, r10            -- 飽和減算（a - b）
    st.w    r10, -12 + .A2[sp]  -- 飽和演算結果を変数 c にストア
      :
```

(f) ハーフワード・データのバイト・スワップ（bsh）【V850E】

ハーフワード・データのバイト・スワップ命令（bsh）の記述例を示します。

例

```
void func(void){
    long    a, b;
      :
    b = __bsh(a);   /*a のハーフワード・データのバイト・スワップ，結果を b に格納 */
      :
}
```

```
【出力コード】
_func:
      :
    ld.w    -4 + .A2[sp], r10   -- 変数 a をロード
    bsh     r10, r10            -- ハーフワード・データのバイト・スワップ
    st.w    r10, -8 + .A2[sp]   -- ハーフワード・データのバイト・スワップ
                                -- 結果を変数 b にストア
      :
```

（g）ワード・データのバイト・スワップ（bsw）【V850E】

ワード・データのバイト・スワップ命令（bsw）の記述例を示します。

例

```
void func(void){
    long    a, b;
      :
    b = __bsw(a);    /*a のワード・データのバイト・スワップ，結果を b に格納 */
      :
}
```

```
【出力コード】
_func:
      :
    ld.w    -8 + .A2[sp], r10    -- 変数 a をロード
    bsw     r10, r10             -- ワード・データのバイト・スワップ
    st.w    r10, -12 + .A2[sp]   -- 変数 b にストア
      :
```

（h）ワード・データのハーフワード・スワップ（hsw）【V850E】

ワード・データのハーフワード・スワップ命令（hsw）の記述例を示します。

例

```
void func(void){
    long    a, b;
      :
    b = __hsw(a);    /*a のワード・データのハーフワード・スワップ，結果を b に格納 */
      :
}
```

```
【出力コード】
_func:
      :
    ld.w    -8 + .A2[sp], r10    -- 変数 a をロード
    hsw     r10, r10             -- ワード・データのハーフワード・スワップ
    st.w    r10, -12 + .A2[sp]   -- 変数 b にストア
      :
```

(i) バイト・データの符号拡張（sxb）【V850E】

バイト・データの符号拡張命令（sxb）の記述例を示します。

例

```
void func(void){
    char    a;
    long    b;
      :
    b = __sxb(a);    /*a のバイト・データの符号拡張，結果を b に格納 */
      :
}
```

```
【出力コード】
_func:
      :
    ld.b    -8 + .A2[sp], r10    -- 変数 a をロード
    sxb     r10, r10             -- バイト・データの符号拡張
    st.w    r10, -12 + .A2[sp]   -- 変数 b にストア
      :
```

(j) ハーフワード・データの符号拡張（sxh）【V850E】

ハーフワード・データの符号拡張命令（sxh）の記述例を示します。

例

```
void func(void){
    short   a;
    long    b;
      :
    b = __sxh(a);    /*a のハーフワード・データの符号拡張，結果を b に格納 */
      :
}
```

```
【出力コード】
_func:
      :
    ld.h    -8 + .A2[sp], r10    -- 変数 a をロード
    sxh     r10                  -- ハーフワード・データの符号拡張
    st.w    r10, -12 + .A2[sp]   -- 変数 b にストア
      :
```

(k) mul 命令を用いて乗算結果の上位 32 ビットを変数に代入する命令【V850E】

mul 命令を用いて乗算結果の上位 32 ビットを変数に代入する命令の記述例を示します。

例

```
void func(void){
    long    a, b, c;
      :
    c = __mul32(a, b);  /*a × b の結果の上位 32 ビットを c に格納 */
      :
}
```

```
【出力コード】
_func:
      :
    ld.w    -4 + .A2[sp], r10   -- 変数 a をロード
    ld.w    -8 + .A2[sp], r11   -- 変数 b をロード
    mul     r11, r10, r12       --a × b
    st.w    r12, -12 + .A2[sp]  -- 変数 c にストア
      :
```

(l) mulu 命令を用いて符号なし乗算結果の上位 32 ビットを変数に代入する命令【V850E】

mulu 命令を用いて符号なし乗算結果の上位 32 ビットを変数に代入する命令の記述例を示します。

例

```
void func(void){
    unsigned long   a, b, c;
      :
    c = __mul32u(a, b); /*a ×b の結果の上位 32 ビットを c に格納 */
      :
}
```

```
【出力コード】
_func:
      :
    ld.w    -4 + .A2[sp], r10   -- 変数 a をロード
    ld.w    -8 + .A2[sp], r11   -- 変数 b をロード
    mulu    r11, r10, r12       --a × b
    st.w    r12, -12 + .A2[sp]  -- 変数 c にストア
      :
```

（m）論理左シフト付きフラグ条件の設定（sasf）【V850E】

例 1 に第 2 引数に条件式を書く場合の論理左シフト付きフラグ条件の設定命令（sasf）の記述例を示します。

例 2 に第 2 引数に変数を書く場合の論理左シフト付きフラグ条件の設定命令（sasf）の記述例を示します。

例 1.　第 2 引数に条件式を書く場合

```
void func(void){
    unsigned long   a, b, c;
      :
    c = __sasf(c, a == b);  /*a == b が真ならば c を左 1 ビット論理シフトして
                              1 を加える，偽ならば c を左 1 ビット論理シフト，
                              結果を c に格納 */
      :
}
```

```
【出力コード】
_func:
      :
    ld.w    -4 + .A2[sp], r10   -- 変数 a をロード
    ld.w    -8 + .A2[sp], r11   -- 変数 b をロード
    cmp     r11, r10            -- 変数 a，b を比較
    ld.w    -12 + .A6[sp], r12  -- 変数 c をロード
    sasf    0x2, r12            --a == b が真ならば c を左 1 ビット論理シフトして
                                --1 を加える，偽ならば c を左 1 ビット論理シフト
    st.w    r12, -12 + .A2[sp]  -- 変数 c にストア
      :
```

2.　第 2 引数に変数を書く場合

```
void func(void){
    unsigned long   a, b;
      :
    b = __sasf(b, a);   /*a が 0 でなければ b を左 1 ビット論理シフトして 1 を加える，
                          a が 0 以外ならば b を左 1 ビット論理シフト，
                          結果を b に格納 */
      :
}
```

```
【出力コード】
_func:
        :
    ld.w    -4 + .A2[sp], r10    -- 変数 a をロード
    cmp     r0, r10              -- 変数 a と 0 を比較
    ld.w    -8 + .A2[sp], r11    -- 変数 b をロード
    sasf    0xa, r11             --a が 0 でなければ b を左 1 ビット論理シフフトして
                                 --1 を加える，a が 0 ならば b を左 1 ビット論理シフト
    st.w    r11, -8 + .A2[sp]    -- 変数 b にストア
      :
```

（11）構造体パッキング

CA850 は，構造体メンバのアライメントを C 言語レベルで指定できます。この機能は，-Xpack オプションと同等ですが，構造体パッキング指令は C 言語ソース内の任意な位置でアライメント値を指定できます。

注意 構造体をパッキングすると，データ領域を小さくできますが，プログラム・サイズは増え，実行速度も低下します。

（a）構造体パッキングの形式

構造体パッキング機能は次の形式で指定します。

```
#pragma pack([1248])
```

#pragma pack は，この指令が出現した時点で，構造体メンバのアライメント値に変更します。この数値をパッキング値と呼び，指定できる数値は，1，2，4，または 8 です。また，パッキング値を指定しない場合，デフォルトのアライメント 8 [注]になります。なお，この指令は出現した時点で有効となるので C 言語ソース内に複数記述することができます。

例

```
#pragma pack(1) /* 構造体メンバを 1 バイトのアライメントで整列 */
struct  TAG{
    char    c;
    int     i;
    short   s;
};
```

注 本バージョンではアライメント値“4”と“8”は同じになります。

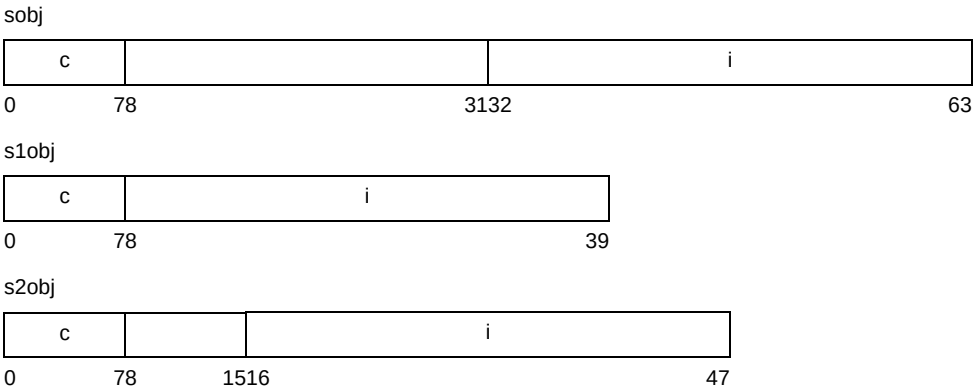
（b）構造体パッキングのルール

構造体のメンバは，構造体のパッキング値とメンバの持つアライメント値の小さい方の値の整列条件を満たす形で並べられます。

たとえば，構造体のパッキング値が 2 のときメンバの形が int 型ならば 2 バイトの整列条件を満たす形で並べられます。

例

```
struct  S{
    char    c;      /*1 バイトの整列条件を満たす */
    int     i;      /*4 バイトの整列条件を満たす */
};
#pragma pack(1)
struct  S1{
    char    c;      /*1 バイトの整列条件を満たす */
    int     i;      /*1 バイトの整列条件を満たす */
};
#pragma pack(2)
struct  S2{
    char    c;      /*1 バイトの整列条件を満たす */
    int     i;      /*2 バイトの整列条件を満たす */
};
struct  S   sobj;   /* サイズ 8 バイト */
struct  S1  s1obj;  /* サイズ 5 バイト */
struct  S2  s2obj;  /* サイズ 6 バイト */
```



(c) 共用体

共用体をパッキングの対象として構造体パッキングと同様に扱います。

例 1.

```
union   U{
    struct  S{
        char    c;
        int     i;
    }sobj;
};
#pragma pack(1)
union   U1{
    struct  S1{
        char    c;
        int     i;
    }s1obj;
};
#pragma pack(2)
union   U2{
    struct  S2{
        char    c;
        int     i;
    }s2obj;
};
union   U   uobj;   /* サイズ 8 バイト */
union   U1  u1obj;  /* サイズ 5 バイト */
union   U2  u2obj;  /* サイズ 6 バイト */
```

2.

```
union   U{
    int i:7;
};
#pragma pack(1)
union   U1{
    int i:7;
};
#pragma pack(2)
union   U2{
    int i:7;
};
union   U   uobj;   /* サイズ 4 バイト */
union   U1  u1obj;  /* サイズ 1 バイト */
union   U2  u2obj;  /* サイズ 2 バイト */
```

(d) ビット・フィールド

ビット・フィールド要素の領域は次のように割り当てます

- 構造体のパッキング値がメンバの型の整列条件値と等しいあるいは大きい場合

  構造体パッキング機能を利用しなかったときと同じように割り当てます。つまり，続けて割り当てるとその領域がメンバの型の整列条件を満たす境界を越えてしまう場合，その整列条件を満たしている領域から割り当てます。

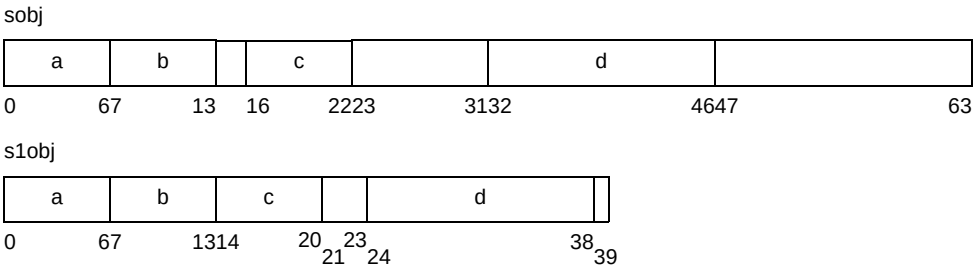
- 構造体のパッキング値が要素の型の整列条件値より小さい場合
  - 続けて割り当てるとその領域を含むバイト数が要素の型よりも大きくなる場合

    構造体のパッキング値の整列条件を満たす形で割り当てます。
  - それ以外の場合

    続けて割り当てます。

例

```
struct  S{
    short   a:7;    /*0 ～ 6 ビット目 */
    short   b:7;    /*7 ～ 13 ビット目 */
    short   c:7;    /*16 ～ 22 ビット目（2 バイト境界に整列）*/
    short   d:15;   /*32 ～ 46 ビット目（2 バイト境界に整列）*/
}sobj;
#pragma pack(1)
struct  S1{
    short   a:7;    /*0 ～ 6 ビット目 */
    short   b:7;    /*7 ～ 13 ビット目 */
    short   c:7;    /*14 ～ 20 ビット目 */
    short   d:15;   /*24 ～ 38 ビット目（バイト境界に整列）*/
}s1obj;
```



(e) 構造体オブジェクトの先頭の整列条件

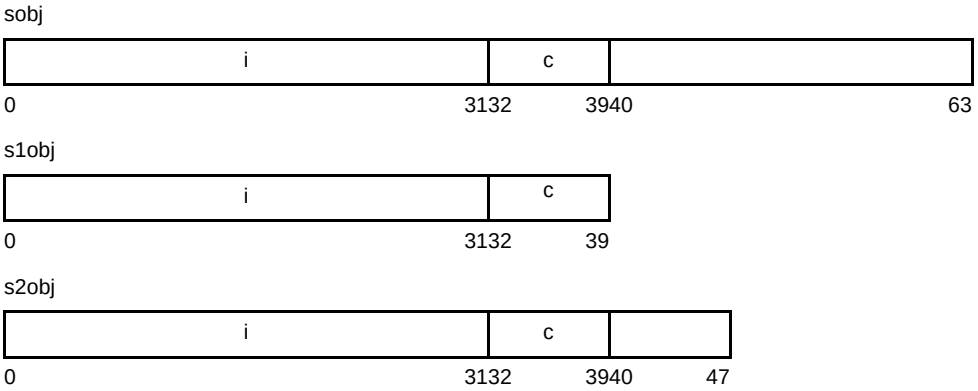
構造体オブジェクトの先頭の整列条件は，構造パッキング機能を利用しなかったときと同じです。

(f) 構造体オブジェクトのサイズ

構造体のサイズが構造体の整列条件の値と構造体のパッキング値の小さい方の値の倍数になるようにパッキングを行います。オブジェクトの先頭の整列条件は構造パッキング機能を利用しなかったときと同じです。

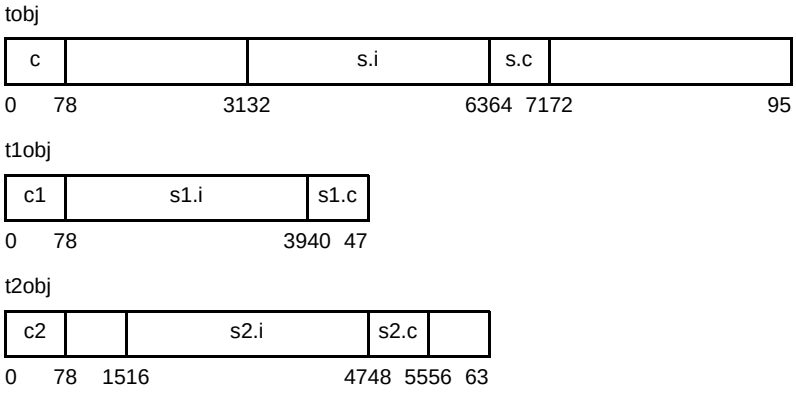
例 1.

```
struct  S{
    int     i;
    char    c;
};
#pragma pack(1)
struct  S1{
    int     i;
    char    c;
};
#pragma pack(2)
struct  S2{
    int     i;
    char    c;
};
struct  S   sobj;   /* サイズ 8 バイト */
struct  S1  s1obj;  /* サイズ 5 バイト */
struct  S2  s2obj;  /* サイズ 6 バイト */
```



2.

```
struct  S{
    int     i;
    char    c;
};
struct  T{
    char    c;
    struct  S   s;
};
#pragma pack(1)
struct  S1{
    int     i;
    char    c;
};
struct  T1{
    char    c;
    struct  S1  s1;
};
#pragma pack(2)
struct  S2{
    int     i;
    char    c;
};
struct  T2{
    char    c;
    struct  S2  s2;
};
struct  T   tobj;   /* サイズ 12 バイト */
struct  T1  t1obj;  /* サイズ  6 バイト */
struct  T2  t2obj;  /* サイズ  8 バイト */
```

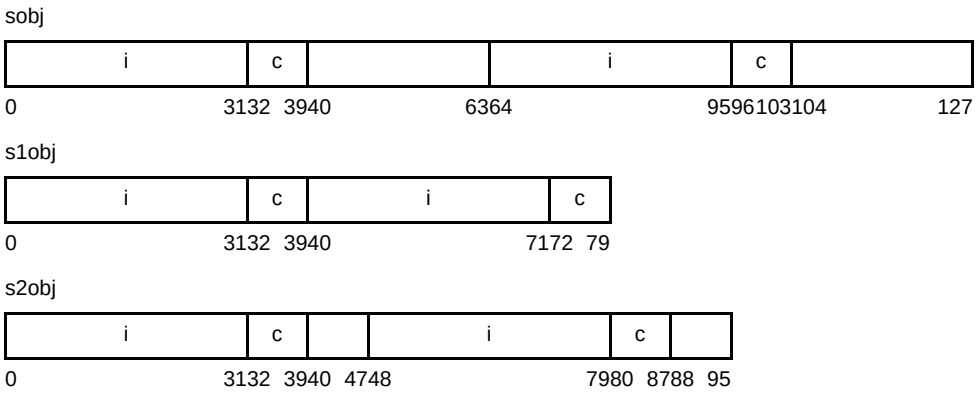


(g) 構造体配列のサイズ

構造体オブジェクトの配列のサイズは要素である構造体オブジェクトのサイズに要素数を乗算した値です。

例

```
struct  S{
    int     i;
    char    c;
};
#pragma pack(1)
struct  S1{
    int     i;
    char    c;
};
#pragma pack(2)
struct  S2{
    int     i;
    char    c;
};
struct  S   sobj[2];    /* サイズ 16 バイト */
struct  S1  s1obj[2];   /* サイズ 10 バイト */
struct  S2  s2obj[2];   /* サイズ 12 バイト */
```

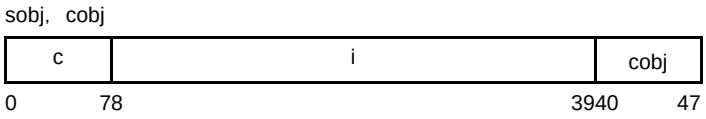


(h) オブジェクト間の領域

たとえば，次のソース・プログラムでは，sobj.c，sobj.i，cobj が隙間なく続いて配置される可能性があります（sobj，cobj の配置順は保証されません）。

例

```
#pragma pack(1)
struct  S{
    char    c;
    int     i;
}sobj;
char    cobj;
```



(i) 構造体パッキング機能の注意点

- -Xpack オプションと #pragma pack 指令の同時指定について

C 言語ソース中に #pragma pack 指令で構造体パッキング指定がある時に -Xpack オプションを指定した場合，最初の #pragma pack 指令が出現するまではオプション指定値がすべての構造体に適用されます。それ以降は #pragma pack 指令の値が適用されます。

ただし，#pragma pack 指令の出現後でも指定がデフォルトになった部分は，オプション指定値が適用されます。

例 （-Xpack=2 を指定した場合）

```
struct  S2{...};     /* オプションでパッキング値 2 を指定している
                        オプション -Xpack=2 が有効：パッキング値 2*/
#pragma pack(1)      /*#pragma 指令でパッキング 1 を指定している
struct  S1{...};        pragma pack(1) が有効：パッキング値 1*/
#pragma pack()       /*#pragma 指令でパッキング値にデフォルトを指定している
struct  S2_2{...};      オプション -Xpack=2 が有効：パッキング値 2*/
```

- 制限事項

V850 マイクロコントローラ，V850Ex 製品ミス・アライン・アクセス禁止の設定の CPU をご使用の場合，次の制限があります。

- 構造体メンバのアドレスでのアクセスが正しく行えません。

次のように構造体メンバのアドレスを取得して，そのアドレスでのアクセスは，デバイスのデータ・アライメントに従い，アドレスをマスクしてアクセスされるため，データの消失や切り捨てが生じます。

例

```
struct  test{
    char    c;      /*offset 0*/
    int     i;      /*offset 1-4*/
}test;
int *ip, i;
void func(void){
    i = *ip;        /* マスクされたアドレスでアクセスされる */
}
void func2(void){
    ip = &(test.i); /* 構造体メンバのアドレス取得 */
}
```

- ビット・フィールドへのアクセスは，そのメンバの型で読み込むためデータがない領域もアクセスします。
  次のようにビット・フィールドの幅がメンバの型以下の場合，メンバの型で読み込むのでオブジェクトの外部にアクセスします。実行上，通常は問題ありませんが，I/O などがマップされていた場合に不正なアクセスとなる場合があります。

例

```
struct  S{
    int x:21;
}sobj;  /*3 バイト */
sobj.x = 1;
```

3. 2. 5　C ソースの修正

拡張機能を使用することにより，効率の良いオブジェクトを生成することができます。しかし，拡張機能はV850 に則したもので，他に利用するためには修正が必要になる場合があります。

ここでは，他の C コンパイラから CA850 への移植と，CA850 から他の C コンパイラへの移植の 2 つの場合について，その方法を説明します。

＜他の C コンパイラから CA850 ＞

- #pragma 注

  他の C コンパイラが #pragma をサポートしている場合は，C ソースを修正する必要があります。修正方法は，その C コンパイラの仕様によって検討します。

- 拡張仕様

  他の C コンパイラがキーワードを追加するなどの仕様の拡張を行っている場合は，修正する必要があります。修正方法はその C コンパイラの仕様によって検討します。

注　ANSI でサポートされている前処理指令の 1 つで，#pragma に続く文字列をコンパイラへの指令として認識させるものです。その指令がコンパイラによってサポートされていなければ，#pragma 指令は無視され，コンパイルが続けられて正常に終了します。

＜ CA850 から他の C コンパイラ＞

- CA850 は，拡張機能としてキーワードの追加を行っているため，他の C コンパイラへ移植するためには，キーワードを削除するか，#ifdef で切り分けなければなりません。

例 1.　キーワードを無効にする

```
#ifndef __CA850__
#define interrupt       /*interrupt 関数を通常の関数にします */
#endif
```

2.　他の型に変更する

```
#ifdef __V850__
#define bit char    /*bit 型変数を char 型変数にします */
#endif
```

3. 3　関数呼び出しインタフェース

この節では，CA850 におけるプログラム呼び出し時の引数などの扱い方について説明します。

3. 3. 1　C 言語関数間の呼び出し

- 通常の関数呼び出し
  → jarl 命令
- 関数を指すポインタを用いた関数呼び出し
  → jmp 命令

C 言語関数から C 言語関数を呼び出す際，4 ワード分の引数を“引数用レジスタ（r6 ～ r9）”に格納し，4 ワードを越えた引数は，呼び出し側の関数のスタック・フレームに格納します。その後，呼び出された関数へ移行（ジャンプ）し，呼び出されるときに格納された“引数用レジスタの値”を，呼び出された関数側で，呼び出された関数側のスタック・フレームに格納します。

スタック・フレームは，関数のプロローグ・コード，つまり，関数が呼び出されてから，関数本体のコードを実行する前に実行されるコード（「図 3―18　スタック・フレームの生成／消滅（引数レジスタ領域がスタックの中央にくる場合）」，「図 3―20　スタック・フレームの生成／消滅（引数レジスタ領域がスタックの先頭にくる場合）」で示す（4）～（7）の処理がプロローグ・コードになります）において，スタック・ポインタ（sp）を，必要サイズ分だけずらすことによって生成されます。また，関数のエピローグ・コード，つまり，関数本体のコードを実行し終わり，呼び出し側の関数に戻るまでに実行されるコード（「図 3―18　スタック・フレームの生成／消滅（引数レジスタ領域がスタックの中央にくる場合）」，「図 3―20　スタック・フレームの生成／消滅（引数レジスタ領域がスタックの先頭にくる場合）」で示す（i）～（iv）の処理がエピローグ・コードになります）において，スタック・ポインタ（sp）を戻すことにより，スタック・フレームは消滅します。

（1）スタック・フレーム／関数呼び出し

スタック・フレームの形状，および関数呼び出し時のスタック・フレームの生成／消滅状態について説明します。

（a）スタック・フレームの形状

CA850 では，スタック・フレームは引数の条件によって，引数レジスタ領域を“スタックの先頭”か“スタックの中央”のどちらかに確保します。引数の条件は次のようになります。

- 引数レジスタ領域が，スタックの先頭に配置される場合

  4 ワード分の引数領域を越えて，連続アクセスする必要がある場合で，次の二通りが該当します。
  - 引数が可変個引数の場合
  - 引数が構造体実体で，その領域が 4 ワード境界をまたぐ場合

- 引数レジスタ領域が，スタックの中央に配置される場合

この場合は上記条件以外の場合になります。

引数レジスタ領域を“スタックの先頭”に持つ場合のスタック・フレームを「図 3―16 スタック・フレーム（引数レジスタ領域がスタック中央になる場合）」，“スタックの中央”に持つ場合のスタック・フレームを「図 3―17 スタック・フレーム（引数レジスタ領域がスタック先頭になる場合）」に示します

図 3―16 スタック・フレーム（引数レジスタ領域がスタック中央になる場合）

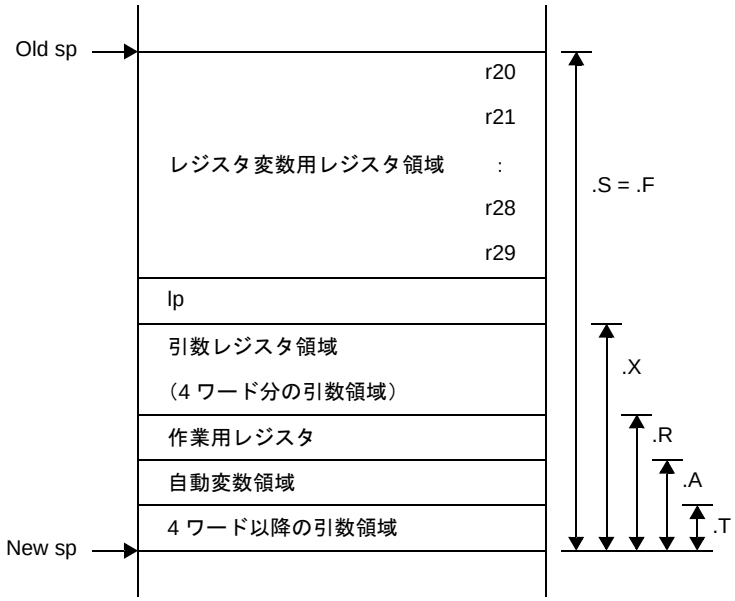
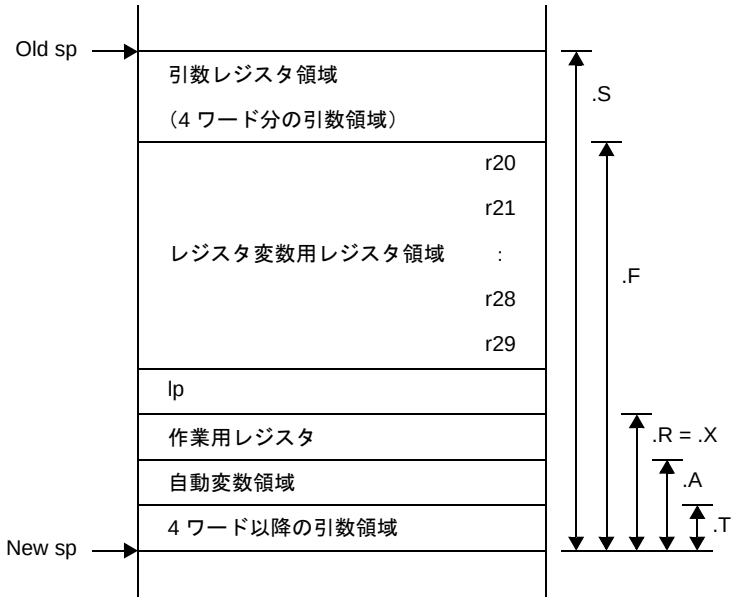


図 3―17 スタック・フレーム（引数レジスタ領域がスタック先頭になる場合）



図中にある“.S，.F，.X，.R，.A，.T”は，コンパイラが内部的に出力する関数用マクロです。
それぞれ，使用用途が決まっており，次表のようになります。

表 3―30　関数用マクロ

マクロ名	意味
.S	スタック・サイズ
.F	スタック・サイズ - 引数レジスタ領域のサイズ（スタックの先頭にある場合）
.X	引数レジスタ領域のサイズ（スタックの中央にくる場合）+ .R
.R	作業用レジスタ領域のサイズ + .A
.A	自動変数領域のサイズ + .T
.T	呼び出す関数の 4 ワード以降の引数領域のサイズ
.P	常に 0（コード生成用のマクロ）**注**

注　.P は常に 0 のため，「図 3―16　スタック・フレーム（引数レジスタ領域がスタック中央になる場合）」，「図 3―17　スタック・フレーム（引数レジスタ領域がスタック先頭になる場合）」には記述してありません。

これらのマクロを使用して，スタック領域にアクセスすることになりますが，具体的なアクセス方法（出力するアクセス・コード）は次表のようになります。

表 3―31　スタック領域のアクセス方法

スタック領域	アクセス方法（ディスプレースメント [sp]）
レジスタ変数用レジスタ領域（lp も含む）	-offset + .F*xx*[sp]
作業用レジスタ領域	-offset + .R*xx*[sp]
自動変数領域	-offset + .A*xx*[sp]
4 ワード以降の引数領域	offset + .P*xx*[sp]
引数レジスタ領域	offset + .F*xx*[sp]
引数レジスタ領域（スタックの中央にくる場合）	offset + .R*xx*[sp]

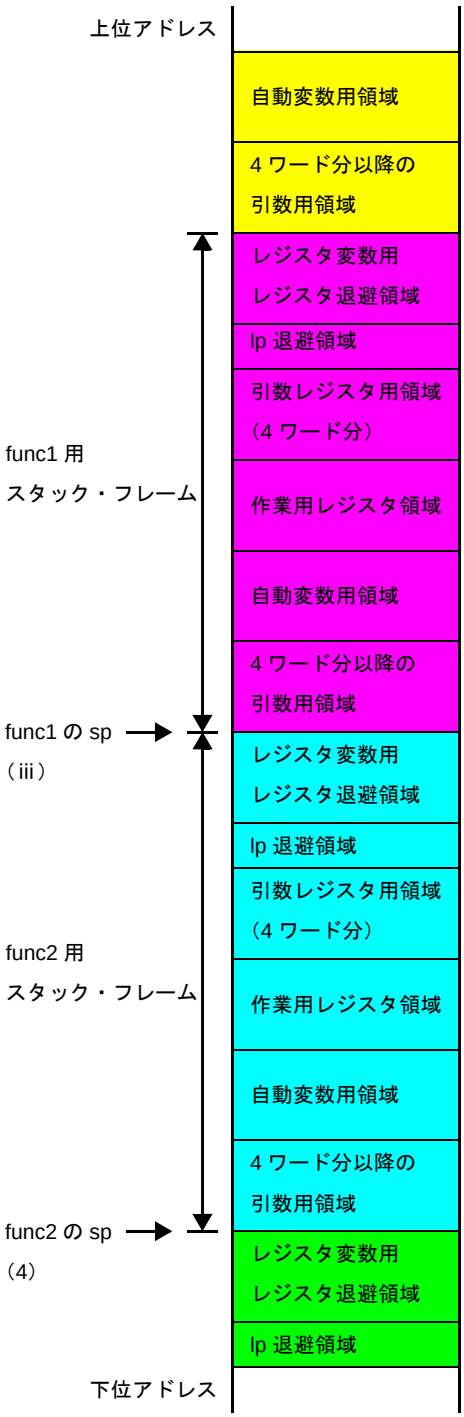
表中で“offset”は正の整数で，各領域中のオフセットを意味します。また，マクロの後に書かれている“*xx*”は正の整数で，関数のフレーム番号を示します。

（b）関数呼び出し時のスタック・フレームの生成／消滅（引数レジスタ領域が“スタックの中央”にくる場合）

“引数レジスタ領域がスタックの中央にくる場合”の，関数呼び出し時のスタック・フレームの生成と消滅について説明します。ほとんどの関数呼び出しは，このケースになります。

以下に，関数 func1 から関数 func2 を呼び出し，その後関数 func1 へ戻るときの，スタック・フレームの生成／消滅の例を示します。

図 3―18　スタック・フレームの生成／消滅（引数レジスタ領域がスタックの中央にくる場合）



【関数 func2 を呼び出すときの関数 func1 側の処理】

（1）引数を引数用レジスタに格納

呼び出す func2 の引数を r6 ～ r9 へ格納

（2）4 ワード分以降の引数をスタックへ格納

r6 ～ r9 の 4 つに入りきらない引数を格納。

（3）jarl 命令で func2 へ分岐

【関数 func1 が呼び出されたときの関数 func2 側の処理】

（4）sp をずらす

func2 が使用するスタックへ移動

（5）lp を退避

func1 の戻りアドレスを格納

（6）レジスタ変数用レジスタを退避

func2 でもレジスタ変数用レジスタを使用するとき，func1 で使用していたレジスタ値を保護する必要があるために退避

（7）引数を“引数レジスタ用領域”に格納

r6 ～ r9 の値を格納

func2 から他関数を呼び出すとき，その時の引数をレジスタ（r6 ～ r9）に格納することになるため，現状の引数値をスタックへ格納しておく

V850Ex の場合，（4）～（6）の処理は prepare 命令で実行できるため，CA850 は prepare 命令を出力します。

【関数 func2 から関数 func1 へ戻るときの関数 func2 側の処理】

（i）レジスタ変数用レジスタの復帰

退避していた“func1 のレジスタ変数用レジスタの値”をレジスタへ戻す

（ii）lp の復帰

func1 の戻りアドレスを復帰

（iii）sp を戻す

func1 が使用するスタックへ移動

（iv）jmp [lp] で戻る

V850Ex の場合，（ i ）～（ iv ）の処理は dispose 命令で実行できるため，CA850 は dispose 命令を出力します。

スタック・フレームに退避されるものと，使用されるスタック・フレームをまとめると次のようになります。

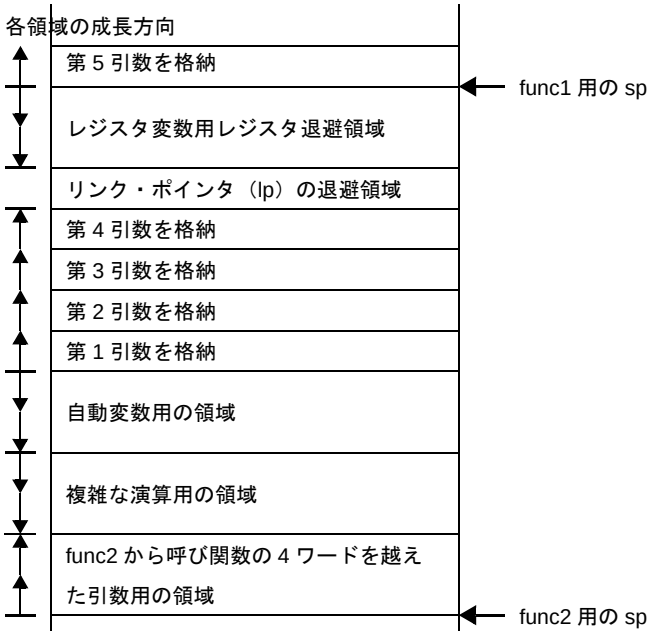
- 呼び出す側 ～ 関数 func1
  - 呼び出す func2 の引数が 4 ワードを越えていた時，越えた分の引数の値

- 呼び出される側 ～ 関数 func2
  - 引数用レジスタに入れられた引数の受け渡し（引数用レジスタに入れるのは，呼び出す側（関数 func1））
  - 呼び出した側（関数 func1）のリンク・ポインタ（lp）（= 関数 func1 の戻りアドレス）の退避
  - “レジスタ変数用レジスタ”の退避
    “レジスタ変数用レジスタ”として割り当てられているのは，次のようになります。
    22 レジスタ・モードのとき：“r25，r26，r27，r28，r29”
    26 レジスタ・モードのとき：“r23，r24，r25，r26，r27，r28，r29”
    32 レジスタ・モードのとき：“r20，r21，r22，r23，r24，r25，r26，r27，r28，r29”
    このうち使用しているものを退避します。
  - “自動変数用”の領域
  - 関数内で非常に複雑な式が用いられた場合，演算のために使用する領域の確保
    この領域を使用する必要が出てきた場合には，自動変数用の領域の下位側に確保されます。

また，関数に戻り値がある場合，その値は r10 に格納されます。

スタック・フレームの各領域の配置と各領域のスタック成長方向のイメージを図にすると，次のようになります（呼び出す関数 func2 の引数が 5 個あるとします）。

図 3―19　スタック・フレームの各領域のスタック成長方向



次に“C 言語関数から C 言語関数を呼び出したソース”と“それをコンパイルしたときのアセンブリ言語ソース”の具体例を示します。

例

```
void func1(void){
    int a, b, c, d, e;
    func2(a, b, c, d, e);
      :
}
int func2(int a, int b, int c, int d, int e){
    register int    i;
      :
    return(i);
}
```

例の関数 func2 呼び出しに対して生成されるアセンブラ命令

【V850】

```
_func1:
    jbr     .L3
.L4:
    ld.w    -8 + .A3[sp], r6
    ld.w    -12 + .A3[sp], r7
    ld.w    -16 + .A3[sp], r8       -- (1)
    ld.w    -20 + .A3[sp], r9
    ld.w    -24 + .A3[sp], r10
    st.w    r10, [sp]               -- (2)
    jarl    _func2, lp              -- (3)
      :
    --func1 に対するエピローグ
    --（ii）から（iv）の処理
.L3:
    --func1 に対するプロローグ
    --(4), (5) の処理
      :
    jbr     .L4
_func2:
    jbr     .L5
.L6:
    st.w    r6, .R2[sp]
    st.w    r7, 4 + .R2[sp]
    st.w    r8, 8 + .R2[sp]         -- (7)
    st.w    r9, 12 + .R2[sp]
    st.w    r29, -4 + .A2[sp]
      :
    jbr     .L2
.L2:
    ld.w    -4 + .A2[sp], r10
    ld.w    -4 + .F2[sp], r29       --（i）
    ld.w    -8 + .F2[sp], lp        --（ii）
    add     .F2, sp                 --（iii）
    jmp[lp]                         --（iv）
.L5:
    add     -.F2, sp                -- (4)
    st.w    lp, -8 + .F2[sp]        -- (5)
    st.w    r29, -4 + .F2[sp]       -- (6)
    jbr     .L6
```

【V850E】

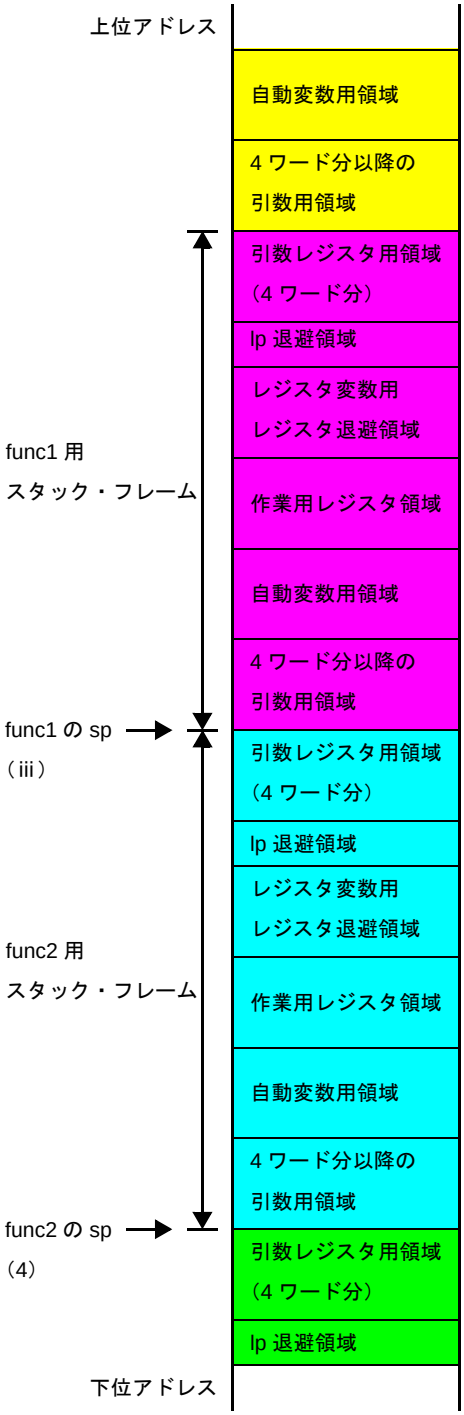
```
_func1:
    jbr     .L3
.L4:
    ld.w    -8 + .A3[sp], r6
    ld.w    -12 + .A3[sp], r7
    ld.w    -16 + .A3[sp], r8       -- (1)
    ld.w    -20 + .A3[sp], r9
    ld.w    -24 + .A3[sp], r10
    st.w    r10, [sp]               -- (2)
    jarl    _func2, lp              -- (3)
      :
    --func1 に対するエピローグ
    -- (ii) から (iv) の処理
.L3:
    --func1 に対するプロローグ
    -- (4), (5) の処理
      :
    jbr     .L4
_func2:
    jbr     .L5
.L6:
    st.w    r6, .R2[sp]
    st.w    r7, 4 + .R2[sp]
    st.w    r8, 8 + .R2[sp]         -- (7)
    st.w    r9, 12 + .R2[sp]
    st.w    r29, -4 + .A2[sp]
      :
    jbr     .L2
.L2:
    ld.w    -4 + .A2[sp], r10
    dispose .X2, 0x3, [lp]
    -- (i), (ii), (iii), (iv)
.L5:
    prepare 0x3, .X2
    -- (4), (5), (6)
    jbr     .L6
```

（c）関数呼び出し時のスタック・フレームの生成／消滅（引数レジスタ領域が“スタックの先頭”にくる場合）

“引数レジスタ領域がスタックの先頭にくる場合”の関数呼び出し時のスタック・フレームの生成と消滅について説明します。

以下に，関数 func1 から関数 func2 を呼び出し，その後関数 func1 へ戻るときの，スタック・フレームの生成／消滅の例を示します。

図 3―20　スタック・フレームの生成／消滅（引数レジスタ領域がスタックの先頭にくる場合）



【関数 func2 を呼び出すときの関数 func1 側の処理】

（1）引数を引数用レジスタに格納

呼び出す func2 の引数を r6 ～ r9 へ格納

（2）4 ワード分以降の引数をスタックへ格納

r6 ～ r9 の 4 つに入りきらない引数を格納。

引数が 5 つ以上の場合は，この処理を行う対象となります。

（3）jarl 命令で func2 へ分岐

【関数 func1 が呼び出されたときの関数 func2 側の処理】

（4）sp をずらす

func2 が使用するスタックへ移動

（5）lp を退避

func1 の戻りアドレスを格納

（6）レジスタ変数用レジスタを退避

func2 でもレジスタ変数用レジスタを使用するとき，func1 で使用していたレジスタ値を保護する必要があるために退避

（7）引数を“引数レジスタ用領域”に格納

r6 ～ r9 の値を格納

func2 から他関数を呼び出すとき，その時の引数をレジスタ（r6 ～ r9）に格納することになるため，現状の引数値をスタックへ格納しておく

V850Ex の場合，（4）～（6）の処理は prepare 命令で実行できるため，CA850 は prepare 命令を出力します。

【関数 func2 から関数 func1 へ戻るときの関数 func2 側の処理】

（i）レジスタ変数用レジスタの復帰

退避していた“func1 のレジスタ変数用レジスタの値”をレジスタへ戻す

（ii）lp の復帰

func1 の戻りアドレスを復帰

（iii）sp を戻す

func1 が使用するスタックへ移動

（iv）jmp [lp] で戻る

V850Ex の場合，（ i ）～（ iv ）の処理は dispose 命令で実行できるため，CA850 は dispose 命令を出力します。

スタック・フレームに退避されるものと，使用されるスタック・フレームをまとめると次のようになります。

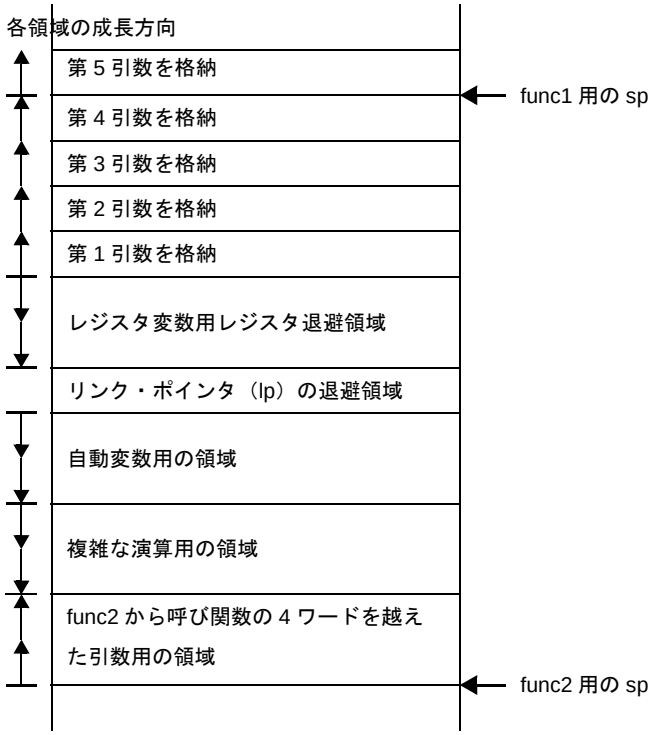
- 呼び出す側 ～ 関数 func1
  - 呼び出す func2 の引数が 4 ワードを越えていた時，越えた分の引数の値

- 呼び出される側 ～ 関数 func2
  - 引数用レジスタに入れられた引数の受け渡し（引数用レジスタに入れるのは，呼び出す側（関数 func1））
  - 呼び出した側（関数 func1）のリンク・ポインタ（lp）（= 関数 func1 の戻りアドレス）の退避
  - “レジスタ変数用レジスタ”の退避
  - “自動変数用”の領域
  - 関数内で非常に複雑な式が用いられた場合，演算のために使用する領域の確保
    この領域を使用する必要が出てきた場合には，自動変数用の領域の下位側に確保されます。

また，関数に戻り値がある場合，その値は r10 に格納されます。

スタック・フレームの各領域の配置と各領域のスタック成長方向のイメージを図にすると，次のようになります（呼び出す関数 func2 の引数が 5 個あるとします）。

図 3―21 スタック・フレームの各領域のスタック成長方向



次に“C 言語関数から C 言語関数を呼び出したソース”と“それをコンパイルしたときのアセンブリ言語ソース”の具体例を示します。

例

```
void func1(void){
    int a, b, c, d, e;
    func2(a, b, c, d, e);
      :
}
int func2(int a, int b, int c, int d, int e){
    register int    i;
      :
    return(i);
}
```

例の関数 func2 呼び出しに対して生成されるアセンブラ命令

【V850】

```
_func1:
    jbr     .L3
.L4:
    ld.w    -8 + .A3[sp], r6
    ld.w    -12 + .A3[sp], r7
    ld.w    -16 + .A3[sp], r8       -- (1)
    ld.w    -20 + .A3[sp], r9
    ld.w    -24 + .A3[sp], r10
    st.w    r10, [sp]               -- (2)
    jarl    _func2, lp              -- (3)
      :
    --func1 に対するエピローグ
    --（ii）から（iv）の処理
.L3:
    --func1 に対するプロローグ
    -- (4), (5) の処理
      :
    jbr     .L4
_func2:
    jbr     .L5
.L6:
    st.w    r6, .F2[sp]
    st.w    r7, 4 + .F2[sp]
    st.w    r8, 8 + .F2[sp]         -- (7)
    st.w    r9, 12 + .F2[sp]
      :
    st.w    r29, -4 + .A2[sp]
    jbr     .L2
.L2:
    ld.w    -4 + .A2[sp], r10
    ld.w    -4 + .F2[sp], r29       --（i）
    ld.w    -8 + .F2[sp], lp        --（ii）
    add     .S2, sp                 --（iii）
    jmp     [lp]                    --（iv）
.L5:
    sub     -.S2, sp                -- (4)
    st.w    lp, -8 + .F2[sp]        -- (5)
    st.w    r29, -4 + .F2[sp]       -- (6)
    jbr     .L6
```

【V850E】

```
_func1:
    jbr     .L3
.L4:
    ld.w    -8 + .A3[sp], r6
    ld.w    -12 + .A3[sp], r7
    ld.w    -16 + .A3[sp], r8       -- (1)
    ld.w    -20 + .A3[sp], r9
    ld.w    -24 + .A3[sp], r10
    st.w    r10, [sp]               -- (2)
    jarl    _func2, lp              -- (3)
      :
    --func1 に対するエピローグ
    --（ii）から（iv）の処理
.L3:
    --func1 に対するプロローグ
    -- (4), (5) の処理
      :
    jbr     .L4
_func2:
    jbr     .L5
.L6:
    st.w    r6, .F2[sp]
    st.w    r7, 4 + .F2[sp]
    st.w    r8, 8 + .F2[sp]         -- (7)
    st.w    r9, 12 + .F2[sp]
      :
    st.w    r29, -4 + .A2[sp]
    jbr     .L2
.L2:
    ld.w    -4 + .A2[sp], r10
    dispose .X2, 0x3
    -- (i), (ii), (iii)
    add     .S2 - .F2, sp           -- (iii)
    jmp     [lp]                    -- (iv)
.L5:
    add     .F2 - .S2, sp           -- (4)
    prepare 0x3, .X2
    -- (4), (5), (6)
    jbr     .L6
```

3. 3. 2 関数のプロローグ／エピローグ

CA850 は，関数のプロローグ／エピローグ処理の一部分を“ランタイム・ライブラリ呼び出し”にすることで，オブジェクト・サイズの削減を行うことができます。これを“プロローグ／エピローグのランタイム化”と呼びます。つまり，関数のプロローグ／エピローグ処理は決まった処理なので，これらを“ランタイム・ライブラリ関数”として CA850 でまとめて用意し，関数呼び出し時や関数戻り時にこれらの関数を呼び出します。

関数のプロローグ／エピローグ部分のアセンブラ命令の例を次に示します。

なお，例中の“(数字)”は「図 3―18 スタック・フレームの生成／消滅（引数レジスタ領域がスタックの中央にくる場合）」の説明に対応しています。

例

```
int func(int a, int b, int c, int d, int e){
    register int    i;
      :
    return(i);
}
```

上記例の関数 f のプロローグ／エピローグ部分のアセンブラ命令

【ランタイムを使用しない場合の出力コード】

```
_func:
    jbr     .L5
.L6:
    st.w    r6, .R2[sp]
    st.w    r7, 4 + .R2[sp]
    st.w    r8, 8 + .R2[sp]          -- (7)
    st.w    r9, 12 + .R2[sp]
      :
    st.w    r29, -4 + .A2[sp]
    jbr     .L2
.L2:
    ld.w    -4 + .A2[sp], r10
    ld.w    -4 + .F2[sp], r29        -- (ⅰ)
    ld.w    -8 + .F2[sp], lp         -- (ⅱ)
    add     .F2, sp                  -- (ⅲ)
    jmp     [lp]                     -- (ⅳ)
.L5:
    add     -.F2, sp                 -- (4)
    st.w    lp, -8 + .F2[sp]         -- (5)
    st.w    r29, -4 + .F2[sp]        -- (6)
    jbr     .L6
```

【ランタイムを使用する場合の出力コード】

```
_func :
    jbr     .L5
.L6:
    st.w    r6, .R2[sp]
    st.w    r7, 4 + .R2[sp]
    st.w    r8, 8 + .R2[sp]          -- (7)
    st.w    r9, 12 + .R2[sp]
      :
    st.w    r29, -4 + .A2[sp]
    jbr     .L2
.L2:
    ld.w    -4 + .A2[sp], r10
    add     .R2, sp                  -- (iii)
    jarl    ___pop2904, lp
    -- (i), (ii), (iii), (iv)
.L5:
    jarl    ___push2904, r10
    -- (4), (5), (6)
    add     -.R2, sp                 -- (4)
    jbr     .L6
```

(1) 関数のプロローグ／エピローグのランタイム化の指定

プロローグ／エピローグのランタイム化は，コンパイル・オプション“-Xpro_epi_runtime=on”を指定することにより行います。逆にランタイム化しない場合には“-Xpro_epi_runtime=off”を指定します。

ただし，最適化オプション“-Ot（実行速度優先最適化）”指定時以外のとき，自動的に関数のプロローグ／エピローグのランタイム化が行われます（自動的にコンパイラ・オプション“-Xpro_epi_runtime=on”が指定されます）。

“-Ot オプション”以外を指定し，かつ，ランタイム化をしたくない場合は“-Xpro_epi_runtime=off オプション”を指定する必要があります。

なお，“-Xpro_epi_runtime オプション”は，ソース・ファイル個別に指定することも可能であるため，“ランタイム化するファイル”と“ランタイム化しないファイル”を混在することができます。

また，“-Xpro_epi_runtime=on オプション”で，関数のプロローグ／エピローグのランタイム化を行う場合，専用のセクション“.pro_epi_runtime セクション”が必要となります。

したがって，リンク・ディレクティブでは，次のように定義する必要があります。

```
.pro_epi_runtime = $PROGBITS    ?AX     .pro_epi_runtime;
```

このセクションには，プロローグ／エピローグ・ランタイム関数のテーブル情報が配置されます。

(2) V850Ex における関数のプロローグ／エピローグのランタイム化

V850Ex を使用している場合，関数のプロローグ／エピローグ・ランタイム関数の呼び出しには，CALLT 命令を使用します。

CALLT 命令は“2 バイト命令”で，関数呼び出しにこの命令を使用することによって，コード・サイズ削減の効果が得られます。CALLT 命令は“CTBP（Callt Base Pointer）レジスタ”に“CALLT 命令の対象となる，関数のテーブル”を指すポインタを設定が必要となります。設定処理がプログラム内になかった場合，リンク時に次のエラー・メッセージが出力されます。

```
F4414: CallTBasePointer(CTBP) is not set. CTBP must be set when compileroption "-Ot" (or "-
Xpro_epi_runtime=off") is not specified.
```

CTBP レジスタへの値設定は，アセンブラ命令で行う必要があるため，スタート・アップ・ルーチンで行うのが適当です。

したがって，次の命令をスタート・アップ・ルーチンに追加してください。

```
mov     #___PROLOG_TABLE, r12    --“PROLOG”の前の“_（アンダースコア）”は 3 つ
ldsr    r12, 20
```

このとき，___PROLOG_TABLE が“関数のプロローグ／エピローグのランタイム関数”の関数テーブルの先頭シンボルとなり，CTBP に設定され，関数テーブル自体は“.pro_epi_runtime セクション”に配置されます。また，上記では“汎用レジスタ r12”を使用していますが，特に r12 である必要はありません。

この CA850 が提供する“関数のプロローグ／エピローグのランタイム化”で CALLT 命令を使用する以外で CALLT 命令を使用する場合，CTBP レジスタを退避／復帰する必要があります。もし，CALLT 命令を他のオブジェクト，たとえばミドルウエアやユーザ作成のライブラリで使用しており，かつ，その中に CTBP レジスタの退避／復帰コードがない，または挿入できない場合，“関数のプロローグ／エピローグのランタイム化”は使用できません。その場合は“-Xpro_epi_runtime=off オプション”を指定して，ランタイム化を抑止してください。

なお，CALLT 命令や CTBP レジスタについての詳細は，各デバイスのユーザーズ・マニュアルを参照してください。

(3) 関数のプロローグ／エピローグのランタイム化の注意事項

関数のプロローグ／エピローグのランタイム化において，次のような注意事項があります。

- 関数のプロローグ／エピローグのランタイム化は，関数呼び出しが入るため，その分，実行速度は低下します。これを避けたい場合は“-Xpro_epi_runtime=off オプション”を指定してください。ファイル単位で“-Xpro_epi_runtime=off オプション”を指定すると効果的です。
- 呼び出される関数が少ないプログラムの場合，プロローグ／エピローグのランタイム化を行っても，コード・サイズの削減ができない場合があります。効果が見込めない場合は“-Xpro_epi_runtime=off ”を指定してください。
- 割り込み関数の場合は，プロローグ／エピローグ・ランタイム化は行いません。割り込み関数から呼び出される関数については，プロローグ／エピローグ・ランタイム化の対象になります。

3. 3. 3 far jump 機能

CA850 は，関数を呼び出す際に，次の“jarl 命令”を使用したコードを出力します。

```
jarl    _func1, lp
```

V850 マイクロコントローラのアーキテクチャ上，jarl 命令の第 1 オペランドとして指定できるのは，符号拡張した 22 ビット値まで（22 ビット・ディスプレースメント）となっています。

したがって，分岐点から± 2M バイト範囲内に分岐先がなかった場合，分岐ができず，リンカで次のエラー・メッセージが出力されます。

```
F4161:symbol " 関数名 "(output section : セクション名 ) is too far from output section " セクション名 ".(value : disp 値, file : main.o, input section : .text, offset: オフセット値 , type : R_V850_PC22).
```

これを解決するには，以下のように配置することですぐに解決できますが，ターゲット・システムにおいては，この範囲内に分岐先を配置できないことがあります。これを解決するのが“far jump 機能”です。

- 分岐先を，分岐点から± 2M バイト範囲内に配置する

far jump 機能を使用すると，関数を呼び出すときに“jmp 命令”を使用したコードを出力します。これにより，V850 が持つ 32 ビット空間すべてに分岐できるようになります。ただし，汎用レジスタを 1 つ使うことになります。far jump 機能を使用した関数呼び出しを“far jump 呼び出し”と呼びます。

（1）far jump 指定の方法

far jump 呼び出しを行う場合“far jump 呼び出しを行う関数”を列挙したファイル（far jump 呼び出し関数一覧ファイル）を用意し，コンパイラ・オプション“-Xfar_jump”を使用します。

```
-Xfar_jump far jump 呼び出し関数一覧ファイル
```

または“=”をつけてもよいです。

```
-Xfar_jump=far jump 呼び出し関数一覧ファイル
```

“far jump 呼び出し関数一覧ファイル”の書式については次節を参照してください。

（2）far jump 呼び出し関数一覧ファイル

far jump 呼び出しの対象とする関数を列挙する“far jump 呼び出し関数一覧ファイル”の書式について説明します。far jump 機能を適用したい関数を，1 行に 1 関数を記述していきます。記述する名前は，C 言語関数名の先頭に“_（アンダースコア）”を付けた名前になります。

【far jump 呼び出し関数一覧ファイルのサンプル】

```
_func_led
_func_beep
_func_motor
  :
_func_switch
```

“_ 関数名”のかわりに次のようにを記述すると，すべての関数を far jump 呼び出しの対象にします。

```
{all_function}
```

{all_function} 指定があると，他に“_ 関数名”があっても，すべての関数が far jump 呼び出しの対象になります。

なお，far jump 機能は，ユーザ関数だけではなく，次のものにも適用できます。

- 標準ライブラリ関数
- ランタイム・ライブラリ関数
- 関数のプロローグ／エピローグ・ランタイム関数
- リアルタイム OS のシステム・コール

“far jump 呼び出し関数一覧ファイル”の注意事項は次のとおりです。

- 使用できる文字は ASCII 文字のみです。
- コメントは挿入できません。
- 1 行に 1 関数のみです。
- 関数名の前後に，空白やタブを挿入できます。
- 1 行に 1023 文字まで記述できます。空白やタブも 1 文字と数えます。
- 関数名は C 言語関数の先頭に“_（アンダースコア）”付で記述してください。
- フラッシュ／外付け ROM の再リンク機能と併用することはできません。

(3) far jump 機能の使用例

far jump 機能の使用例について説明します。

(a) ユーザ関数（標準関数も同様）

【C 言語ソース・ファイル】

```
extern  void    func3(void);
void func(void)
{
    func3();
}
```

【far jump 呼び出し関数一覧ファイル】

```
_func3
```

【通常の呼び出しコード】

```
#@CALL_ARG
jarl    _func3, lp
```

【far jump 呼び出しコード】

```
#@CALL_ARG
    movea   _func3, tp, r10
    movea   .L18, tp, lp
    jmp     [r10]
.L18:
```

（b）ランタイム関数（マクロ呼び出しの場合）

【far jump 呼び出し関数一覧ファイル】

```
___mul
```

【通常の呼び出しコード】

```
.macro  mul     arg1, arg2
    add     -8, sp
    st.w    r6, [sp]
    st.w    r7, 4[sp]
    mov     arg1, r6
    mov     arg2, r7
    jarl    ___mul, lp
    ld.w    4[sp], r7
    mov     r6, arg2
    ld.w    [sp], r6
    add     8, sp
.endm
```

【far jump 呼び出しコード】

```
.macro  mul     arg1, arg2
    .local  macro_ret
    add     -8, sp
    st.w    r6, [sp]
    st.w    r7, 4[sp]
    mov     arg1, r6
    mov     arg2, r7
    movea   macro_ret, tp, r31
    .option nowarning
    movea   #___mul, tp, r1
    jmp     [r1]
    .option warning
macro_ret:
    ld.w    4[sp], r7
    mov     r6, arg2
    ld.w    [sp], r6
    add     8, sp
.endm
```

(c) ランタイム関数の場合（ダイレクト呼び出しの場合）

【far jump 呼び出し関数一覧ファイル】

```
___mul
```

【通常の呼び出しコード】

```
mov     r12, r6
mov     r13, r7
#@CALL_ARG      r6, r7
#@CALL_USE      r6, r7
jarl    ___mul, lp
mov     r6, r13
```

【far jump 呼び出しコード】

```
    mov     r12, r6
    mov     r13, r7
    #@CALL_ARG      r6, r7
    #@CALL_USE      r6, r7
    movea   #___mul, tp, r14
    movea   .L13, tp, lp
    jmp     [r14]
.L13:
    mov     r6, r13
```

ランタイムのマクロ呼び出しとダイレクト呼び出しはコンパイラが最適化の過程でレジスタ効率などを判定し自動的に切り替えます。

(d) リアルタイム OS のシステム・コールの場合

【far jump 呼び出し関数一覧ファイル】

```
_ext_tsk
```

【通常の呼び出しコード】

```
#@B_EPILOGUE
#@BEGIN_NO_OPT
add     .S4, sp
jr      _ext_tsk    --C NR
#@END_NO_OPT
#@E_EPILOGUE
```

【far jump 呼び出しコード】

```
#@B_EPILOGUE
#@BEGIN_NO_OPT
add     .S4, sp
movea   #_ext_tsk, tp, r10
jmp     [r10]       --C NR
#@END_NO_OPT
#@E_EPILOGUE
```

(e) プロローグ／エピローグ・ランタイム関数の場合

【far jump 呼び出し関数一覧ファイル】

```
___pop2900
___push2900
```

【通常の呼び出しコード】

```
#@B_EPILOGUE
    jarl    ___pop2900, lp  --1
#@E_EPILOGUE
.L3:
    jarl    ___push2900, r10
#@E_PROLOGUE
```

【far jump 呼び出しコード】

```
#@B_EPILOGUE
    movea   #___pop2900, tp, r11
    jmp     [r11]           --1
#@E_EPILOGUE
.L3:
    movea   #___push2900, tp, r11
    movea   .L5, tp, r10
    jmp     [r11]
.L5:
#@E_PROLOGUE
```

以下に，far jump 指定可能な，プロローグ／エピローグ・ランタイム関数名を示します。プロローグ／エピローグ・ランタイム関数を指定する場合，一度コンパイル後に出力されたアセンブリ言語ソースで使用されている関数を確認し，指定します。

表 3―32 プロローグ／エピローグ・ランタイム関数

プロローグ／エピローグ・ランタイム関数名					
___pop2000	___pop2001	___pop2002	___pop2003	___pop2004	___pop2040
___pop2100	___pop2101	___pop2102	___pop2103	___pop2104	___pop2140
___pop2200	___pop2201	___pop2202	___pop2203	___pop2204	___pop2240
___pop2300	___pop2301	___pop2302	___pop2303	___pop2304	___pop2340
___pop2400	___pop2401	___pop2402	___pop2403	___pop2404	___pop2440
___pop2500	___pop2501	___pop2502	___pop2503	___pop2504	___pop2540
___pop2600	___pop2601	___pop2602	___pop2603	___pop2604	___pop2640
___pop2700	___pop2701	___pop2702	___pop2703	___pop2704	___pop2740
___pop2800	___pop2801	___pop2802	___pop2803	___pop2804	___pop2840
___pop2900	___pop2901	___pop2902	___pop2903	___pop2904	___pop2940
___poplp00	___poplp01	___poplp02	___poplp03	___poplp04	___poplp40
___push2000	___push2001	___push2002	___push2003	___push2004	___push2040
___push2100	___push2101	___push2102	___push2103	___push2104	___push2140
___push2200	___push2201	___push2202	___push2203	___push2204	___push2240
___push2300	___push2301	___push2302	___push2303	___push2304	___push2340
___push2400	___push2401	___push2402	___push2403	___push2404	___push2440
___push2500	___push2501	___push2502	___push2503	___push2504	___push2540
___push2600	___push2601	___push2602	___push2603	___push2604	___push2640
___push2700	___push2701	___push2702	___push2703	___push2704	___push2740
___push2800	___push2801	___push2802	___push2803	___push2804	___push2840
___push2900	___push2901	___push2902	___push2903	___push2904	___push2940
___pushlp00	___pushlp01	___pushlp02	___pushlp03	___pushlp04	___pushlp40

関数のプロローグ／エピローグ・ランタイム・ライブラリについての詳細は「3. 3. 2 関数のプロローグ／エピローグ」を参照してください。

3. 4　CC78Kx の拡張機能

この節では，CA850 での CC78Kx の拡張機能について説明します。

3. 4. 1　#pragma 指令

CA850 では，CC78Kx と互換の次の #pragma 指令が指定できます。

なお，【78K 互換】マークは，次の意味を示します。

【78K 互換】	-cc78K オプションを指定しないと有効になりません。
	#pragma 以降のキーワードについては，大文字，小文字を区別しません。

(1) デバイス種別指定

【78K 互換】

```
#pragma pc( デバイス名 )
```

使用するデバイスの機種依存情報を定義したデバイス・ファイルを参照するように指定します。CA850 の“#pragma cpu デバイス名”，およびデバイス指定オプション（-cpu）と同じ機能です。

(2) 周辺 I/O レジスタ名有効化指定

【78K 互換】

```
#pragma sfr
```

周辺 I/O レジスタ名を用いて，デバイスの持つ周辺 I/O レジスタにアクセスします。CA850 の“#pragma ioreg”と同じ機能です。

(3) 割り込み禁止指定

【78K 互換】

```
#pragma di
```

関数 DI を組み込み関数 __DI として扱います。

(4) 割り込み許可指定

【78K 互換】

```
#pragma ei
```

関数 EI を組み込み関数 __EI として扱います。

(5) CPU 停止関数指定

【78K 互換】

```
#pragma halt
```

関数 HALT を組み込み関数 __halt として扱います。

(6) ノーオペレーション関数指定

【78K 互換】

```
#pragma nop
```

関数 NOP を組み込み関数 __nop として扱います。

(7) CC78Kx の #pragma 指令

次の指定については，78K 互換ではありません。CA850 の #pragma 指令として扱います。

(a) 割り込み／例外ハンドラ指定

【78K 互換】

```
#pragma interrupt    割り込み要求名    関数名 [ スタック切り替え ] …
#pragma vect         割り込み要求名    関数名 [ スタック切り替え ] …
```

CC78Kx の“#pragma interrupt”／“#pragma vect”を CA850 の“#pragma interrupt 割り込み要求名 関数名 [配置方法]”として扱います。“[スタック切り替え]”以降の記述がある場合，CA850 で認識できない場合には，次のメッセージを出力します。

```
W2150: unexpected character(s) following directive 'directive'
```

(b) セクション指定

【78K 互換】

```
#pragma section …
```

CA850 の“#pragma section セクション種別 [" セクション名 "] [begin|end]”として扱います。CA850 で認識できない場合には，次のメッセージを出力します。

```
W2162: unrecognized pragma directive '#pragma directive', ignored
```

(c) メモリ操作関連指定

【78K 互換】

```
#pragma inline
```

CC78Kx では，memchr，memcmp，memcpy，および memset をインライン展開しますが，CA850 では，指定関数をインライン展開するため，次のメッセージを出力します。

```
W2162: unrecognized pragma directive '#pragma inline', ignored
```

(d) モジュール名指定

【78K 互換】

```
#pragma name　　モジュール名
```

CA850 では，次のメッセージを出力します。

```
W2162: unrecognized pragma directive '#pragma name', ignored
```

(e) データ挿入関数指定

【78K 互換】

```
#pragma opc
```

対応する組み込み関数

```
__OPC
```

CA850 では，次のメッセージを出力し，コンパイルを中止します。

```
W2162: unrecognized pragma directive '#pragma opc', ignored
E2752: cannot call opc function
```

(f) バイトアドレス挿入／生成関数指定

【78K 互換】

```
#pragma addraccess
```

対応する組み込み関数

```
FP_SEG，FP_OFF，MK_FP
```

CA850 では，次のメッセージを出力し，コンパイルを中止します。

W2162: unrecognized pragma directive '#pragma addraccess', ignored
E2752: cannot call addraccess function

（g）レジスタ直接参照関数指定

【78K 互換】

```
#pragma realregister
```

対応する組み込み関数

__absa, __ashra, __clr1cy, __coma, __deca, __geta, __getax, __getcy, __inca, __nega, __not1cy, __rola, __rolca, __rora, __rorca, __set1cy, __seta, __setax, __setcy, __shla, __shra

CA850 では，次のメッセージを出力し，コンパイルを中止します。

W2162: unrecognized pragma directive '#pragma realregister', ignored
E2752: cannot call realregister function

（h）ファームウエア内蔵セルフ書き込みサブルーチン直接呼び出し関数指定

【78K 互換】

```
#pragma hromcall
```

対応する組み込み関数

__FlashAreaBlankCheck, __FlashAreaErase, __FlashAreaIVerify, __FlashAreaPreWrite, __FlashAreaWriteBack, __FlashBlockBlankCheck, __FlashBlockErase, __FlashBlockIVerify, __FlashBlockPreWrite, __FlashBlockWriteBack, __FlashByteRead, __FlashByteWrite, __FlashEnv, __FlashGetInfo, __FlashSetEnv, __FlashWordWrite, __hromcall, __hromcalla, __setsp

CA850 では，次のメッセージを出力し，コンパイルを中止します。

W2162: unrecognized pragma directive '#pragma hromcall', ignored
E2752: cannot call hromcall function

3. 4. 2 アセンブラ制御命令

【78K 互換】

```
#asm
    アセンブラ命令
#endasm
```

CA850 の“#pragma asm”～“#pragma endasm”として扱います。

それぞれに対して次のメッセージを出力します。

W2166: recognized pragma directive '#pragma asm'
W2166: recognized pragma directive '#pragma endasm'

3. 4. 3 割り込み／例外ハンドラの指定方法

割り込み／例外ハンドラの指定は，C 言語ソース・プログラムにおいて，次の #pragma 指令と修飾子で行います。

【78K 互換】

```
#pragma interrupt        割り込み要求名      関数名［配置方法］

__interrupt_brk 関数定義，または関数宣言
```

__interrupt_brk 関数修飾子は，CA850 の __interrupt 関数指定として扱います。

3. 4. 4 サポートしていない拡張機能

CA850 は，サポートしていない CC78Kx の拡張仕様に対してメッセージを出力します。

【78K 互換】

```
__banked1，__banked2，__banked3，__banked4，__banked5，__banked6，__banked7，__banked8，
__banked9，__banked10，__banked11，__banked12，__banked13，__banked14，__banked15，callf，__callf，
callt，__callt，noauto，norec，__pascal，sreg，__sreg，__sreg1，__temp
```

CA850 では，次のメッセージを出力します。

W2761: unrecognized specifier '*specifier*', ignored

3.5　セクション名一覧

以下に，予約されているセクションの名前とそれらのセクション・タイプ，およびセクション属性を示します。

表 3―33　予約セクション

名前[注1]	内容	セクション・タイプ	セクション属性
.bss	.bss セクション	NOBITS	AW
.const	.const セクション	PROGBITS	A
.data	.data セクション	PROGBITS	AW
.ext_info .ext_info_boot	フラッシュ／外付け ROM 再リンク機能用情報セクション	PROGBITS	なし
.ext_table	フラッシュ／外付け ROM 再リンク機能用分岐テーブル・セクション	PROGBITS	AX
.ext_tgsym	フラッシュ／外付け ROM 再リンク機能用情報セクション	PROGBITS	なし
.gptabname	グローバル・ポインタ・テーブル[注2]	GPTAB	なし
.pro_epi_runtime	プロローグ／エピローグ・ランタイム呼び出しセクション	PROGBITS	AX
.regmode	レジスタ・モード情報	REGMODE	なし
.relname	リロケーション情報	REL	なし
.relaname	リロケーション情報	RELA	なし
.sbss	.sbss セクション	NOBITS	AWG
.sconst	.sconst セクション	PROGBITS	A
.sdata	.sdata セクション	PROGBITS	AWG
.sebss	.sebss セクション	NOBITS	AW
.sedata	.sedata セクション	PROGBITS	AW
.shstrtab	セクション名を保持しているストリング・テーブル	STRTAB	なし
.sibss	.sibss セクション	NOBITS	AW
.sidata	.sidata セクション	PROGBITS	AW
.strtab	ストリング・テーブル	STRTAB	なし
.symtab	シンボル・テーブル	SYMTAB	なし
.text	.text セクション	PROGBITS	AX
.tibss	.tibss セクション	NOBITS	AW
.tibss.byte	.tibss.byte セクション	NOBITS	AW
.tibss.word	.tibss.word セクション	NOBITS	AW
.tidata	.tidata セクション	PROGBITS	AW
.tidata.byte	.tidata.byte セクション	PROGBITS	AW
.tidata.word	.tidata.word セクション	PROGBITS	AW
.vdbstrtab	デバッグ情報用シンボル・テーブル	STRTAB	なし

名前[注1]	内容	セクション・タイプ	セクション属性
.vdebug	デバッグ情報	PROGBITS	なし
.version	バージョン情報セクション	PROGBITS	なし
.vline	ライン・ナンバ情報	PROGBITS	なし

注 1. gptabname，.relname，および .relaname の name の部分は，それぞれそのセクションに対応するセクションの名前を示します。

2. リンカにおける -A オプションの処理において用いられる情報です。

第 4 章　アセンブラ言語仕様

この章では，CA850 アセンブラ（as850）がサポートするアセンブリ言語仕様について説明します。

4. 1　ソースの記述方法

この節では，ソースの記述方法，式と演算子などについて説明します。

4. 1. 1　記述方法

アセンブリ言語文は，“ラベル”，“ニモニック”，“オペランド”，および“コメント”から構成されます。

```
[ ラベル ]:        [ ニモニック ]      [ オペランド ], [ オペランド ]     --[ コメント ]
```

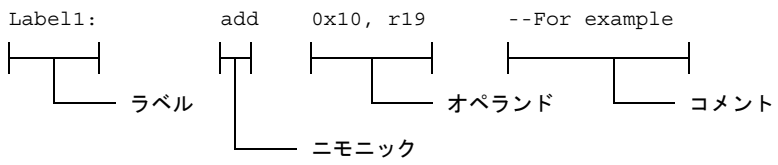
以下の箇所には，1 つ以上の空白があってもなくても問題ありません。

- ラベルとコロンの間
- コロンとニモニックの間
- 2 つ目以降のオペランドの前
- コメントの始まり“--”の前

以下の箇所には，1 つ以上の空白が必要です。

- ニモニックとオペランドの間

図 4―1　アセンブリ言語文の構成



アセンブリ言語文は，基本的に 1 行に 1 文を記述します。文の最後は改行（リターン）します。なお，“;（セミコロン）”を使用すると，1 行に複数のアセンブリ言語文を記述することができます。

(1) 文字セット

as850 がサポートするソース・プログラムで，使用できる文字は次のとおりです。

表 4―1 文字セットとその用途

文字	用途
英小文字（a-z）	ニモニック，識別子，および定数の構成
英大文字（A-Z）	識別子，および定数の構成
_（アンダースコア）	識別子の構成
.（ピリオド）	識別子，および定数の構成
数字	識別子，および定数の構成
:（コロン）	ラベルの終わり
,（カンマ）	オペランドの区切り
-（ハイフン）	負符号，減算演算子，およびコメントの開始
#	ラベルの絶対アドレス参照，およびコメントの開始
;（セミコロン）	文の終わり
'（シングルクォーテーション）	文字定数の開始と終了
"（ダブルクォーテーション）	文字列定数の開始と終了
$	ラベルの gp オフセット参照
[ ]	ベース・レジスタの指定
+	加算演算子
*	乗算演算子
/	除算演算子
%	ラベルのセクション内オフセット参照（命令展開なし），および剰余演算子
<<	左シフト演算子
>>	右シフト演算子
!	ラベルの絶対アドレス参照（命令展開なし），および否定演算子
&	論理積演算子
|	論理和演算子
^	排他的論理和演算子
( )	演算順序の指定

(2) ラベル

ラベルとは，プログラムの任意の行に記述できる，いわゆる“名札”です。ラベルは，条件分岐するときや，サブルーチンへ分岐するとき，その分岐先の名前として使用できます。

たとえば，分岐命令の 1 つである“jr 命令”を使うときは，次のように記述します。

```
jr      Label1
```

この命令の実行によって Label1 のある箇所へ分岐します。つまり，Label1 という名前にラベルを記述するときは，次のように記述します。

```
Label1:
```

ラベルは，複数行に渡って違うラベルを定義することができます。

```
Label1:
Label2:
```

しかし，1 行に複数のラベルを指定することはできません。

```
Label1: Label2: --1 行に複数のラベルを指定することはできません。
```

なお，ラベル名とコロンの間は，1 つ以上の空白があってもなくても問題ありません。

ラベルは，使用する前に“定義／宣言”をする必要があります。

(a) ラベルの定義

ラベルの定義方法には 2 つあります。

- 文の先頭で名前の後に“:”を続けることにより，ローカルなラベルとして定義

```
label1:
```

ローカルなラベル定義方法としては，この方法が一般的で，以後“通常のラベル定義”と呼びます。

- .lcomm 疑似命令により，ローカルなラベルとして定義

```
.lcomm  label1, 0x100, 4
```

記の意味は「4 バイトでアラインされたアドレスから，サイズ“0x100 バイト”分を確保し，その領域の先頭ラベルを“label1”とする」という意味です。

（b）ラベルの宣言

ラベルの宣言には 4 つあります。

- .comm 疑似命令により，未定義外部ラベルとして宣言

```
.comm   label1, 4, 4
```

上記の意味は「サイズ“4 バイト”で，整列条件 4 バイトの未定義外部ラベル“label1”を宣言する」という意味です。

- .extern 疑似命令により，外部ラベルとして宣言（指定したファイル内に定義を持たないラベル）

```
.extern label1
```

- .globl 疑似命令により，外部ラベルとして宣言（指定したファイル内に定義を持つラベル）

```
.globl  label1
```

- ファイル内に定義を持たせないことにより，外部ラベルとして宣言

```
mov     label1, r10
```

上記で，label1 の定義が同一ファイル内にない場合，label1 は外部ラベルとみなされます。

（c）ラベルとして使用できる文字

ラベルとして使用できるのは「（1）文字セット」で示したうち，次の文字になります。

- 英小文字
- 英大文字
- _（アンダースコア）
- .（ピリオド）
- 数字

ただし，ラベルの先頭に数字は使用できません。数字で始まるラベルを指定した場合，次のメッセージが出力され，アセンブルが中止されます。

また，予約語をラベルとして使用することもできません。

```
E3249: illegal syntax
```

注意 “_（アンダースコア）”で始まるラベルは，コンパイラにより出力されたラベル名と一致する可能性があり，予期せぬ動作を招く可能性があるので注意してください。なお“.（ピリオド）”で始まるラベルが，今後予約語となる可能性がありますので，なるべく使用しないでください。

(d) ラベルの最大文字数と最大個数

ラベルの最大文字数は 1037 文字です。1038 文字以上のラベルが指定された場合，次のメッセージが出力され，アセンブルが中止されます。

なお，定義可能なラベルの最大個数は，利用可能なメモリ領域の大きさに依存します。

E3260: token too long

(e) sbss ／ bss 属性 セクションにおける通常のラベル定義

sbss ／ bss 属性セクションにおいて，通常のラベル定義を行った場合，次のメッセージが出力され，アセンブルが中止されます。

なお，このエラーが出力された場合は，.lcomm 疑似命令を使用して定義してください。

E3246: illegal section

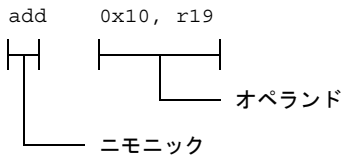
(3) ニモニックとオペランド

ニモニックとは，各命令（V850 マシン・コード）に対し，それぞれ割り当てられた英文字列です。マシン・コードは，そのままの状態では人間には理解しづらいので，各マシン・コードに対してつけられた名前が“ニモニック”です。つまり，命令そのものを意味します。ニモニックはその操作内容が容易に推定できるように，英語を基本とした“言葉に近い表記”になっています。

たとえば，“add”というニモニックは“加算”を意味し，“mul”であれば“乗算”を意味します。オペランドとは，各命令操作に操作される対象を指します。ニモニックが“add”（加算）だった場合，オペランドはその演算の対象となるものです。ニモニックの次（右）にオペランドを記述します。

なお，ニモニックとオペランドの間は，1 つ以上の空白が必要です。

図 4―2 ニモニックとオペランド



アセンブリ言語命令は“ニモニック”と“オペランド”の組み合わせで成り立っています。オペランドの数はニモニックによって異なります。

V850 に搭載されているアセンブリ言語命令の一覧とその仕様については，「4. 5. 4 命令セット」を参照してください。

(4) コメント

アセンブリ言語プログラム内にコメントを記入することができます。as850 では，次の記述以降をその行の終わりまでをコメントとして扱います。

```
--
```

```
#
```

ただし，“#”は文の先頭[注]にあった場合のみ，その行の終わりまでをコメントとして扱います。
なお，コメント内では“EUC”または“シフト JIS コード”の日本語を記述することができます。

```
# comment
add     0x10, r19    --comment 1
sub     r18, r19     --comment 2
```

注 行頭の空白は，文に含まれません。“#”の前に空白が含まれる場合も，その行の終わりまでをコメントとして扱います。

(5) 定数

as850 では“数値定数”，“文字定数”，および“文字列定数”を定数として扱うことができます。

(a) 数値定数

数値定数には“整定数”と“浮動小数点数”があります。

- 整定数

整定数，つまり，整数の定数は，32 ビット幅をもつ定数です。負の数は 2 の補数で表現されます。32 ビットで表現することのできる値を越える整数値が指定された場合，as850 はその整数値の下位 32 ビットの値を用いて処理を続行します（メッセージ等は出力しません）。

【2 進定数】

2 進定数は，“0b”，または“0B”と，その後ろに続く 1 個以上の“0”，または“1”の数字の並びで構成されます。

例

```
0b00010110111101010111111010010111
```

【8 進定数】

8 進定数は，“0”と，その後ろに続く 1 個以上の“0”から“7”までの数字の並びで構成されます。

例

```
02675277227
```

【10 進定数】

10 進定数は，“0”以外の数字で始まる 1 個以上の数字の並びで構成されます。

例

```
385187479
```

【16 進定数】

16 進定数は，“0x”，または“0X”と，その後ろに続く 1 個以上の“0”から“9”までの数字，“a”から“f”までの英小文字，または“A”から“F”までの英大文字の並びで構成されます。

例

```
0x16f57e97
```

- 浮動小数点数

浮動小数点数は 32 ビット幅を持ちます。浮動小数点数は，次に示す要素で構成されます。

（ｉ） 仮数部の符号（“+”は省略可）
（ii） 仮数値
（iii） 指数部を示す“e”，または“E”
（iv） 指数部の符号（“+”は省略可）
（ｖ） 指数値

指数値，および仮数値は 10 進定数で指定します。ただし，指数表現を用いない場合，（iii），（iv），および（ｖ）は用いられません。

例

```
123.4
-100.
10e-2
-100.2E+5
```

なお，仮数値の先頭に“0f”または“0F”を置くことにより，浮動小数点数であることを示すこともできます。たとえば，as850 では“10”は整定数であるとみなされますが，“0f10”は浮動小数点数であるとみなされます。また，“060”のように“0”で始まり，小数点を持たない数字列を指定することはできません（“0”のみは指定できます）。

（b）文字列定数

文字定数は，1 つの文字をシングル・クォート“’”で囲むことにより構成され，囲まれた文字の値を示します。**注**

以下に示したエスケープ・シーケンスを“’”と“’”の間に指定した場合，as850 では，1 つの文字を表すものとして扱われます。

例

```
'a'
'\0'
'\012'
'\x0a'
```

注　文字定数を指定した場合，as850 では，その文字定数の値を持つ整定数を指定したものとみなして扱われます。

表 4―2　エスケープ・シーケンスの値と意味

エスケープ・シーケンス	値	意味
\0	0x00	null 文字
\a	0x07	アラート
\b	0x08	バックスペース
\f	0x0c	フォーム・フィード
\n	0x0a	改行
\r	0x0d	キャリッジ・リターン
\t	0x09	水平タブ
\v	0x0b	垂直タブ
\\	0x5c	バックスラッシュ
\'	0x27	シングル・クォート
\"	0x22	ダブル・クォート
\?	０ｘ３ｆ	疑問符
\ddd	0 ～ 0377	3 桁までの 8 進数（0 < d < 7）**注**
\xhh	0 ～ 0xff	2 桁までの 16 進数 （0 < h < 9，a < h < f，または A < h < F）

注“\377”を越える指定の場合，そのエスケープ・シーケンスの値は，下位 1 バイト分のみの値となります。0377 より大きい値にはなりません。たとえば“\777”の値は 0377 です。

（6）シンボル

シンボルとは，値（整数値）を持った名前のことをいい，ユーザが定義します。シンボルを定義するときには，“.set 疑似命令”を使用します。

```
.set    sym1, 0x10   --sym1 は 0x10 という値を持つシンボル
mov     sym1, r10    --sym1 の値（0x10）をレジスタに格納
```

as850 では，ファイルの先頭から最初の .set 疑似命令までの間に出現したシンボル参照を“その時点において未定義なシンボル参照”とし，定義済みのシンボル参照とは区別して扱われます（「4. 1. 2　式」の「（1）絶対値式」を参照）。

（a）シンボルとして使用できる文字

シンボルとして使用できるのは「（1）文字セット」で示したうち，次の文字になります。

- 英小文字
- 英大文字
- _（アンダースコア）
- .（ピリオド）
- 数字

ただし，シンボルの先頭に数字は使用できません。数字で始まるシンボルを指定した場合，次のメッセージが出力され，アセンブルが中止されます。

また，予約語をラベルとして使用することもできません。

```
E3260: token too long
```

注意 “_（アンダースコア）”で始まるシンボルは，コンパイラにより出力されたラベル名と一致する可能性があり，予期せぬ動作を招く可能性があるので注意してください。なお“.（ピリオド）”で始まるシンボルが，今後予約語となる可能性がありますので，なるべく使用しないでください。

（b）シンボルの最大文字数と最大個数

シンボルの最大文字数は 1037 文字です。1038 文字以上のシンボルが指定された場合，次のメッセージが出力され，アセンブルが中止されます。

```
E3260: token too long
```

定義可能なシンボルの最大個数は，利用可能なメモリ領域の大きさに依存します。

(7) アセンブリ言語文の例

アセンブリ言語プログラムの簡単な例は，次のようになります。

```
# sample program
    .extern __tp_TEXT, 4
    .extern __gp_DATA, 4
    .extern _main
    .section    "RESET", text       --Reset Handler address
    jr      __boot                  --Jump to __boot
    .text                           --Text section
    .align  4                       --Code alignment
    .globl  __boot                  --Alignment
__boot:
    mov     #__tp_TEXT, tp          --Set tp
    mov     #__gp_DATA, gp          --Set gp
    .extern __ssbss, 4
    .extern __esbss, 4
    # start of bss initialize
    mov     #__ssbss, r13
    mov     #__esbss, r13
    cmp     r12, r13
    jnl     sbss_init_end
sbss_init_loop:
    st.w    r0, 0[r13]
    add     4, r13
    cmp     r12, r13
    jl      sbss_init_loop
sbss_init_end:
    # end of bss initialize
    jarl    _main, lp               --Call main function
    .data
    .align 4
data_area:
    .word   0x00                    --data1
    .hword  0x01                    --data2
    .byte   0xff;   .byte   0xfe    --data3, data4
```

4. 1. 2　式

式は“定数”，“シンボル”，“ラベルの参照”，“演算子”，および“かっこ”を要素とし，これらの要素で構成される値を示します。as850 では，絶対値式と相対値式に分けて扱います。

（1）絶対値式

定数値を示す式を“絶対値式”と呼びます。絶対値式は，命令においてオペランドを指定する場合，または疑似命令において値／サイズ／整列条件／フィリング値／ビット幅を指定する場合に用いることができます。通常，絶対値式は，定数，またはシンボルによって構成されます。as850 では，次に示した形式が絶対値式として扱われます。ただし，ビット幅の指定を持たない .byte ／ .hword ／ .shword【V850E】／ .word 疑似命令，および .frame 疑似命令以外の疑似命令に対しては，“定数式”形式以外の絶対値式は指定できません（ビット幅の指定を持たない .byte ／ .hword ／ .shword【V850E】／ .word 疑似命令に対しては，値の指定において次に示したすべての形式の絶対値式が指定でき，.frame 疑似命令に対してはサイズの指定において“定数式”形式以外に“シンボル”形式の絶対値式が指定できます）。

（a）定数式

定義済みのシンボル参照を指定した場合，as850 では，そのシンボルに対して定義した値の定数が指定されたものとして扱われます。したがって，定数式は，定義済みのシンボル参照を，その構成要素として持つことができます。

例

```
.set    sym1, 0x100 -- シンボル sym1 を定義
mov     sym1, r10   -- 定義済みの sym1 は定数式として扱う
```

（b）シンボル

シンボルに関する式には，次のものがあります（“±”は“+”か“-”のどちらかになります）。

- シンボル
- シンボル±定数式
- シンボル - シンボル
- シンボル - シンボル±定数式

ここで言う“シンボル”とは，その時点において未定義なシンボル参照を指します。定義済みのシンボル参照を指定した場合は，as850 はそのシンボルに対して定義した値の“定数”が指定されたものとして扱います。

例

```
add     SYM1 + 0x100, r11   -- この時点で SYM1 は未定義シンボル
.set    SYM1, 0x10          --SYM1 を定義
```

（c）ラベル参照

ラベル参照に関する式には，次のものがあります（“±”は“+”か“-”のどちらかになります）。

- ラベル参照 - ラベル参照
- ラベル参照 - ラベル参照±定数式

ラベル参照に関する式の例は，次のようになります。

例

```
mov     $label1 - $label2, r11
```

上記の例のような“2 つのラベル参照”は，次のように参照にする必要があります。

- 指定したファイル内で，同じセクションに定義をもつ
- 同じ参照方法（$label は $label 同士，#label は #label 同士など）
- 指定したファイル内に定義を持たないラベル参照が指定された場合は，次のメッセージが出力され，アセンブルが中止されます。

```
E3209: illegal expression (labels must be defined)
```

同じセクションに定義を持たない 2 つのラベル参照が指定された場合は，次のメッセージが出力され，アセンブルが中止されます。

```
E3208: illegal expression (labels in different sections)
```

同じ参照方法によらない 2 つのラベル参照が指定された場合は，次のメッセージが出力され，アセンブルが中止されます。

```
E3207: illegal expression (labels have different reference types)
```

ただし，現在のアセンブラの構成上，“ラベル参照に関する式”の中の“- ラベル参照”側のラベル参照に，指定されたファイル内に定義を持たないラベルの絶対アドレス参照が指定された場合，他方のラベル参照の参照方法と同じ参照方法が用いられたものとみなして扱われます。なお，分岐命令に対して，この形式の絶対値式は指定できません。指定した場合は，次のメッセージが出力され，アセンブルが中止されます。

```
E3221: illegal operand (label-label)
```

(2) 相対値式

特定のアドレスからのオフセット値[注1]を示す式を“相対値式”と呼びます。相対値式は，命令においてオペランドを指定する場合，またはビット幅の指定を持たない .byte ／ .hword ／ .word 疑似命令において値を指定する場合に用いることができます。通常，相対値式は，ラベル参照によって構成されます。as850 では，次に示した形式[注2]が相対値式として扱われます。

例 1. このアドレスは CA850 に含まれるリンカ（ld850）におけるリンク時に定められます。このため，このオフセットの値もリンク時に定められます。

2. （絶対値式に対しても同じことが言えますが）as850 では，“- シンボル + ラベルの参照”の形式の式を“ラベルの参照 - シンボル”の形式の式とみなすことはできますが，“ラベルの参照 -（+ シンボル）”の形式の式を“ラベルの参照 - シンボル”の形式の式とみなすことはできません。このため，かっこ“( )”は定数式においてのみ用いるようにしてください。

(a) ラベル参照

ラベル参照に関する式には，次のものがあります（“±”は“+”か“-”のどちらかになります）。

- ラベル参照
- ラベル参照±定数式
- ラベル参照 - シンボル
- ラベル参照 - シンボル±定数式

ラベル参照に関する式の例は次のようになります。

例

```
add     #labe11 + 0x10, r10
add     #label2 - SIZE, r10
.set    SIZE, 0x10
```

4.1.3 演 算 子

演算子は，式における演算の指示に用いることができます。

(1) 演算子の種類

演算子は，“算術演算子”，“シフト演算子”，“ビット論理演算子”，および“比較演算子”の4つに分類できます。なお，“-”は，単項演算子としても二項演算子としても用いることができます。

表 4―3 演算子

種類	演算子
算術演算子	+ - * / %
シフト演算子	<< >>
ビット論理演算子	! | & ^
比較演算子	== < <= != > >= && ||

(2) 演算子の優先順位

次表に，演算子の優先順位を示します。なお，優先順位の等しい演算子が隣り合っている場合，一方の側がかっこで囲まれている場合には，そのかっこで囲まれた方が先に実行されますが，どちらもかっこで囲まれていない場合，またはどちらもかっこで囲まれている場合には，左側にあるものが先に実行されます。

ただし，かっこは定数式においてのみ用いるようにしてください（「4.1.2 式」を参照）。

表 4―4 演算子の優先順位

優先順位	演算子
高い	- ! （単項演算子）
↑	* / << >> %
	& | ^
	+ -
↓	== < <= != > >=
低い	&& ||

4. 1. 4　算術演算子

(1) +

第 1 オペランドと第 2 オペランドの和を求めます。

(2) -

第 1 オペランドと第 2 オペランドの差を求めます。

なお，単項演算子の場合はオペランドの 2 の補数を求めます。

(3) ***

第 1 オペランドと第 2 オペランドの積を求めます。

(4) /

第 1 オペランドと第 2 オペランドの商を求めます。

(5) %

第 1 オペランドを第 2 オペランドで割った余りを求めます。

4. 1. 5　シフト演算子

(1) <<

第 1 オペランドを第 2 オペランドで指定したビット数だけ左シフトします。なお，右側（LSB[注]）には，シフトしたビット数分だけ 0 を入れます。

注　Least Significant Bit（最も下位の桁に対応するビット）の略です。

例

`0x12345678 << 4`	`0x23456780`

(2) >>

第 1 オペランドを，第 2 オペランドで指定したビット数だけ右シフトします。なお，左側（MSB[注]）には，第 1 オペランドが正（MSB が 0）の場合は，シフトしたビット数分だけ 0 を，第 1 オペランドが負（MSB が 1）の場合は，シフトしたビット数分だけ 1 を入れます。

注　Most Significant Bit（最も上位の桁に対応するビット）の略です。

例

`0x12345678 >> 4`	`0x01234567`
`0x87654321 >> 4`	`0xF8765432`

4. 1. 6　ビット論理演算子

(1) !

オペランドのビットごとの論理否定を求めます。

例

!0x12345678	0xEDCBA987

(2) |

第 1 オペランドと第 2 オペランドの論理和を求めます。

例

0x1234 | 0x5678	0x567C

(3) &

第 1 オペランドと第 2 オペランドの論理積を求めます。

例

0x1234 & 0x5678	0x1230

(4) ^

第 1 オペランドと第 2 オペランドの排他的論理和を求めます。

例

0x1234 ^ 0x5678	0x444C

4. 1. 7　比較演算子

(1) ==

第 1 オペランドと第 2 オペランドを比較し，等しい場合は 1 を返し，等しくない場合は 0 を返します。

例

1 == 1	1
1 == 0	0

(2) <

第 1 オペランドと第 2 オペランドを比較し，第 1 オペランドが第 2 オペランドより小さい場合は 1 を返し，第 1 オペランドが第 2 オペランドより大きい，または等しい場合は 0 を返します。

例

1 < 10	1
10 < 1	0

(3) <=

第 1 オペランドと第 2 オペランドを比較し，第 1 オペランドが第 2 オペランドより小さいか等しい場合は 1 を返し，第 1 オペランドが第 2 オペランドより大きい場合は 0 を返します。

例

1 <= 2	1
1 <= 1	1
1 <= 0	0

(4) !=

第 1 オペランドと第 2 オペランドを比較し，等しくない場合は 1 を返し，等しい場合は 0 を返します。

例

1 != 0	1
1 != 1	0

(5) >

第 1 オペランドと第 2 オペランドを比較し，第 1 オペランドが第 2 オペランドより大きい場合は 1 を返し，第 1 オペランドが第 2 オペランドより小さい，または等しい場合は 0 を返します。

例

1 > 0	1
1 > 2	0

(6) >=

第 1 オペランドと第 2 オペランドを比較し，第 1 オペランドが第 2 オペランドより大きい，または等しい場合は 1 を返し，第 1 オペランドが第 2 オペランドより小さい場合は 0 を返します。

例

1 >= 0	1
1 >= 1	1
1 >= 2	0

(7) &&

第 1 オペランドの論理値と第 2 オペランドの論理値の論理積を求めます。

例

1 != 3 && 1 <= 3	1
1 == 1 && 1 != 1	0
1 != 1 && 3 <= 1	0

(8) ||

第 1 オペランドの論理値と第 2 オペランドの論理値の論理和を求めます。

例

1 != 3 || 1 <= 3	1
1 == 1 || 1 != 1	1
1 != 1 || 3 <= 1	0

4. 1. 8 演算の制限

as850 の演算規則を，次に示します。

ただし，その時点において未定義なシンボルの参照，またはラベルの参照を含む式に関しては「4. 1. 2 式」に示した規則を優先します。

(1) 単項演算

単項演算子のオペランドに，絶対値式以外のものは指定できません。また，単項演算子 ! のオペランドに，浮動小数点数値を扱う式は指定できません。

(2) 二項演算

二項演算子のオペランドに指定できる整数値式の組み合わせを，以下に示します。なお，表内では，演算子とオペランドの組み合わせによって構成される式について，次に示す記号を使用しています。

abs	絶対値式
rel	“指定したファイル内に定義を持つラベルを参照する”相対値式
ext	“指定したファイル内に定義を持たないラベルを参照する”相対値式
×	as850 では，その演算子とオペランドの組み合わせが許されない

ただし，浮動小数点数値に関しては浮動小数点数値同士の演算でなければならず，浮動小数点数値と相対値式を同一式内に混在させることはできません。

表 4―5 二項演算における演算規則

オペランド		演算子											
		+			-			*，/			その他		
第 2		abs	rel	ext	abs	rel	ext	abs	rel	ext	abs	rel	ext
第 1	abs	abs	rel	ext	abs	×	×	abs	×	×	abs	×	×
	rel	rel	×	×	rel	abs **注**	×	×	×	×	×	×	×
	ext	ext	×	×	ext	×	×	×	×	×	×	×	×

注 この部分の詳細に関しては，「4. 1. 2 式」を参照してください。

4. 1. 9　絶対式の定義

定数値を示す式を“絶対値式”と呼びます。絶対値式は，命令においてオペランドを指定する場合，または疑似命令において値／サイズ／整列条件／フィリング値／ビット幅を指定する場合に用いることができます。

通常，絶対値式は，定数，またはシンボルによって構成されます。

as850 では，次に示した形式が絶対値式として扱われます。ただし，ビット幅の指定を持たない .byte ／ .hword ／ .shword【V850E】／ .word 疑似命令，および .frame 疑似命令以外の疑似命令に対しては，“定数式”形式以外の絶対値式は指定できません（ビット幅の指定を持たない .byte ／ .hword ／ .shword【V850E】／ .word 疑似命令に対しては，値の指定において次に示したすべての形式の絶対値式が指定でき，.frame 疑似命令に対してはサイズの指定において“定数式”形式以外に“シンボル”形式の絶対値式が指定できます）。

（1）定数式

例

```
.set    sym1, 0x100 -- シンボル sym1 を定義
mov     sym1, r10   -- 定義済みの sym1 は定数式として扱う
```

定義済みのシンボル参照を指定した場合，as850 では，そのシンボルに対して定義した値の定数が指定されたものとして扱われます。したがって，定数式は，定義済みのシンボル参照を，その構成要素として持つことができます。

（2）シンボル

シンボルに関する式には，次のものがあります（“±”は“＋”か“－”のどちらかになります）。

- シンボル
- シンボル±定数式
- シンボル－シンボル
- シンボル－シンボル±定数式

ここで言う“シンボル”とは，その時点において未定義なシンボル参照を指します。定義済みのシンボル参照を指定した場合は，as850 はそのシンボルに対して定義した値の“定数”が指定されたものとして扱います。

例

```
add     SYM1 + 0x100, r11   -- この時点で SYM1 は未定義シンボル
mov     sym1, r10           -- 定義済みの sym1 は定数式として扱う
```

(3) ラベル参照

ラベル参照に関する式には，次のものがあります（“±”は“＋”か“－”のどちらかになります）。

- ラベル参照－ラベル参照
- ラベル参照－ラベル参照±定数式

ラベル参照に関する式の例は，次のようになります。

例

```
mov     $label1 - $label2, r11
```

この例のような“2 つのラベル参照”は，次のように参照にする必要があります。

- 指定したファイル内で，同じセクションに定義をもつ
- 同じ参照方法（$label は $label 同士，#label は #label 同士など）
- 指定したファイル内に定義を持たないラベル参照が指定された場合は，次のメッセージが出力され，アセンブルが中止されます。

```
E3209: illegal expression (labels must be defined)
```

同じセクションに定義を持たない 2 つのラベル参照が指定された場合は，次のメッセージが出力され，アセンブルが中止されます

```
E3208: illegal expression (labels in different sections)
```

同じ参照方法によらない 2 つのラベル参照が指定された場合は，次のメッセージが出力され，アセンブルが中止されます。

```
E3207: illegal expression (labels have different reference types)
```

ただし，現在のアセンブラの構成上，“ラベル参照に関する式”の中の“－ラベル参照”側のラベル参照に，指定されたファイル内に定義を持たないラベルの絶対アドレス参照が指定された場合，他方のラベル参照の参照方法と同じ参照方法が用いられたものとみなして扱われます。なお，分岐命令に対して，この形式の絶対値式は指定できません。指定した場合は，次のメッセージが出力され，アセンブルが中止されます。

```
E3221: illegal operand (label-label)
```

4. 1. 10 識 別 子

識別子とは，シンボル，ラベル，マクロ名などに使用する名前です。識別子として使用できるのは「(1) 文字セット」で示した文字のうち，次の文字になります。

- 英小文字
- 英大文字
- _（アンダースコア）
- .（ピリオド）
- 数字

ただし，名前の先頭に数字は使用できません。また，“_（アンダースコア）”で始まる識別子は，コンパイラにより出力されたラベル名と一致する可能性があり，予期せぬ動作を招く可能性があるので注意してください。なお，“.（ピリオド）”については，予約語となる可能性があるため，使用を避けてください。

4. 1. 11 オペランドの特性

as850 では，命令，および疑似命令に対するオペランドとして，レジスタ，定数，シンボル，ラベル参照，および演算子を指定できます。

(1) レジスタ

as850 において指定できるレジスタを次に示します[注]。

注 PSW，およびシステム・レジスタは，ldsr ／ stsr 命令において，番号で指定します。
なお，as850 では，PC をオペランドに指定する方法はありません。
r0 と zero（ゼロ・レジスタ），r2 と hp（ハンドラ・スタック・ポインタ），r3 と sp（スタック・ポインタ），r4 と gp（グローバル・ポインタ），r5 と tp（テキスト・ポインタ），r30 と ep（エレメント・ポインタ），r31 と lp（リンク・ポインタ）は同じレジスタを示します。

```
r0，zero，r1，r2，hp，r3，sp，r4，gp，r5，tp，r6，r7，r8，r9，r10，r11，r12，r13，r14，r15，r16，r17，
r18，r19，r20，r21，r22，r23，r24，r25，r26，r27，r28，r29，r30，ep，r31，lp
```

(a) r0

r0 は，常に 0 の値を持つレジスタです。したがって，デスティネーション・レジスタとして指定した場合にも，結果の代入は行われません。なお，r0 をデスティネーション・レジスタとして指定した場合，次のメッセージが出力され[注]，アセンブルが続行されます。

注 このメッセージの出力は，as850 の起動時に警告メッセージ抑止オプション（-w）を指定することにより抑止できます。

```
mov     0x10, r0
 ↓
W3013: register r0 used as destination register
```

- ターゲット・デバイスに V850Ex を使用する場合，次の命令で r0 をデスティネーション・レジスタとして指定すると，警告メッセージではなくエラー・メッセージが出力されます。
  - dispose，divh 命令の形式 (1)，および (2)
  - ld.bu，ld.hu，mov 命令の形式 (2)
  - movea，movhi，mulh，mulhi，satadd，satsub，satsubi，satsubr，sld.bu，sld.hu

```
divh    r10, r0
 ↓
E3240: illegal operand (can not use r0 as destination in V850E mode）
```

- 次の命令で，ターゲット・デバイスに V850Ex を使用する場合，次の命令で r0 をソース・レジスタとして指定すると，警告メッセージではなくエラー・メッセージが出力されます。
  - divh 命令の形式 (1)
  - switch

```
divh    r0, r10
 ↓
E3239: illegal operand (can not use r0 as source in V850E mode）
```

（b）r1

アセンブラ予約レジスタ（r1）は，as850 において，命令展開を行う際のテンポラリ・レジスタとして用いられるレジスタです。なお，r1 をソース・レジスタ，またはデスティネーション・レジスタとして指定した場合，次のメッセージが出力され**注**，アセンブルが続行されます。

注 このメッセージの出力は，as850 の起動時に警告メッセージ抑止オプション（-w）を指定することにより抑止できます。

```
mov     0x10, r1
 ↓
W3013: register r1 used as destination register
```

```
mov     r1, r10
 ↓
W3013: register r1 used as source register
```

（2）定数

as850 では，命令，および疑似命令のオペランド指定で使用可能な絶対値式，または相対値式の構成要素として，整定数，および文字定数を用いることができます。また，ld ／ st 命令，およびビット操作命令のオペランド指定には，各デバイス・ファイルで定義されている「周辺 I/O レジスタ名」も指定できます。これにより，ポート・アドレスに対する入出力を行うことができます。また，.float 疑似命令のオペランド指定には浮動小数点定数を，.str 疑似命令のオペランド指定には文字列定数を用いることができます。

(3) シンボル

as850 では，命令，および疑似命令のオペランド指定で使用可能な絶対値式，または相対値式の構成要素として，シンボルを用いることができます。

(4) ラベル参照

as850 では，次に示した命令／疑似命令のオペランド指定で，使用可能な相対値式の構成要素として，ラベル参照を用いることができます。

- メモリ参照命令（ロード／ストア命令，およびビット操作命令）
- 演算命令（算術演算命令，飽和演算命令，および論理演算命令）
- 分岐命令
- 領域確保疑似命令（ただし，.word ／ .hword ／ .byte 疑似命令のみ）

as850 では，ラベル参照は参照方法の違い，およびそのラベル参照を用いている命令／疑似命令の違いにより次に示すように異なる意味を持ちます。

表 4―6 ラベル参照

参照方法	用いている命令	意味
#label	メモリ参照命令，演算命令，jmp 命令	ラベル label 定義の存在する位置の絶対アドレス（アドレス 0 からのオフセット**注 1**）。 32 ビットのアドレスを持ち，V850Ex 以外では，必ず 2 命令に展開。
	領域確保疑似命令（.word ／ .hword ／ .byte）	ラベル label 定義の存在する位置の絶対アドレス（アドレス 0 からのオフセット**注 1**）。 ただし，32 ビットのアドレスを，確保した領域の大きさに準じてマスクした値。
label	メモリ参照命令，演算命令	ラベル label 定義の存在する位置のセクション内オフセット（ラベル label 定義の存在するセクションの先頭アドレスからのオフセット**注 2**）。 32 ビットのオフセットを持ち，必ず 2 命令に展開。 ただし，tp シンボルの生成において対象となっているセグメントに割り当てられたセクションに対しては，tp シンボルからのオフセット参照。
	jmp 命令を除く分岐命令	ラベル label 定義の存在する位置の PC オフセット（ラベル label 参照を用いている命令の先頭アドレスからのオフセット**注 2**）。
	領域確保疑似命令（.word ／ .hword ／ .byte）	ラベル label 定義の存在する位置のセクション内オフセット（ラベル label 定義の存在するセクションの先頭アドレスからのオフセット**注 2**）。 ただし，32 ビットのオフセットを，確保した領域の大きさに準じてマスクした値。
$label	メモリ参照命令，演算命令	ラベル label 定義の存在する位置 gp オフセット（グローバル・ポインタの指すアドレスからのオフセット**注 3**）。

参照方法	用いている命令	意味
!label	メモリ参照命令，演算命令	ラベル label 定義の存在する位置の絶対アドレス（アドレス 0 からのオフセット[注 1]）。 16 ビットのアドレスを持ち，16 ビット・ディスプレースメント，またはイミーディエトをもつ命令に指定した場合，命令展開は行われません。 その他の命令に指定した場合，適切な 1 命令に展開されます。 ラベル label の定義したアドレスが，16 ビットで表現できる範囲でない場合，リンク時にエラーになります。
	領域確保疑似命令（.word ／ .hword ／ .byte）	ラベル label 定義の存在する位置の絶対アドレス（アドレス 0 からのオフセット[注 1]）。 ただし，32 ビットのアドレスを，確保した領域の大きさに準じてマスクした値。
%label	メモリ参照命令，演算命令	ラベル label 定義の存在する位置のセクション内オフセット（ラベル label 定義の存在するセクションの先頭アドレスからのオフセット[注 2]）。 16 ビットのオフセットを持ち，16 ビット・ディスプレースメント，またはイミーディエトをもつ命令に指定した場合，命令展開は行われません。 その他の命令に指定した場合，適切な 1 命令に展開されます。 ラベル label の定義したアドレスが，16 ビットで表現できる範囲でない場合，リンク時にエラーになります。 または，ラベル label 定義の存在する位置の ep オフセット（エレメント・ポインタの示すアドレスからのオフセット）。
	領域確保疑似命令（.word ／ .hword ／ .byte）	ラベル label 定義の存在する位置のセクション内オフセット（ラベル label 定義の存在するセクションの先頭アドレスからのオフセット[注 2]）。 ただし，32 ビットのオフセットを，確保した領域の大きさに準じてマスクした値。

注 1. リンク後のオブジェクト・ファイルにおけるアドレス 0 からのオフセットです。

2. リンク後のオブジェクト・ファイルにおいて，ラベル label の定義の存在するセクションが割り当てられたセクション（出力セクション）の先頭アドレスからのオフセットです。

3. 上記出力セクションが割り当てられたセグメントに対する，テキスト・ポインタ・シンボルの値＋グローバル・ポインタ・シンボルの値の指すアドレスからのオフセットです。

次に，メモリ参照命令，演算命令，分岐命令，および領域確保疑似命令におけるラベル参照の意味を示します。

表 4―7 メモリ参照命令

参照方法	意味
#label [reg]	ラベル label の絶対アドレスがディスプレースメントとして扱われます。 32 ビットの値を持ち，必ず 2 命令に展開されます。#label[r0] とすることにより絶対アドレスによる参照を指定できます。 [reg] の部分が省略でき，省略した場合，as850 では，[r0] が指定されたものとみなされます。
label [reg]	ラベル label のセクション内オフセットがディスプレースメントして扱われます。32 ビットの値を持ち，必ず 2 命令に展開されます。reg に対象とするセクションの先頭アドレスを指すレジスタを指定し，label[reg] とすることにより一般的なレジスタ相対の参照が指定できます。 ただし，tp シンボルの生成において対象となっているセグメントに割り当てられたセクションに対しては，tp シンボルからのオフセットをディスプレースメントとして扱います。
$label [reg]	ラベル label の gp オフセットがディスプレースメントとして扱われます。ラベル label の定義されたセクションにより，32，または 16 ビットの値を持ち，命令展開のパターンが変化します。16 ビットの値を持つ命令展開が行われた場合，ラベル label の定義したアドレスより算出されたオフセットが 16 ビットで表現できる範囲でない場合，リンク時にエラーになります。$label[gp] とすることにより gp レジスタ相対の参照（gp オフセット参照と呼ぶ）が指定できます。[reg] の部分が省略でき，省略した場合，as850 では，[gp] が指定されたものとみなされます。
!label [reg]	ラベル label の絶対アドレスがディスプレースメントとして扱われます。16 ビットの値を持ち，命令展開は行われません。ラベル label に定義したアドレスが 16 ビットで表現できない場合，リンク時にエラーになります。!lable[r0] とすることにより絶対アドレスによる参照が指定できます。 [reg] の部分が省略でき，省略した場合は [r0] が指定されたものとみなされます。 ただし，#label[reg] 参照とは異なり，命令展開は行われません。
%label [reg]	ラベル label のセクション内オフセットがディスプレースメントとして扱われます。ラベル label が ep シンボルとなっているセクションに配置された場合には，ep シンボルからのオフセットをディスプレースメントとして扱われます。16 ビット，または命令によってはそれ以下の値を持ち，その範囲で表現できる値でない場合，リンク時にエラーになります。 [reg] の部分が省略でき，省略した場合，as850 では，[ep] が指定されたものとみなされます。

表 4―8 演算命令

参照方法	意味
#label	ラベル label の絶対アドレスがイミーディエトとして扱われます。 32 ビットの値を持ち，必ず 2 命令に展開されます。
label	ラベル label のセクション内オフセットがイミーディエトとして扱われます。 32 ビットの値を持ち，必ず 2 命令に展開されます。 ただし，tp シンボルの生成において対象となっているセグメントに割り当てられたセクションに対しては，tp シンボルからのオフセットがイミーディエトとして扱われます。
$label	ラベル label の gp オフセットがイミーディエトとして扱われます。 ラベル label の定義されたセクションにより，32，または 16 ビットの値を持ち，命令のパターンが変化します。16 ビットの値を持つ展開をされた場合，ラベル label の定義したアドレスより算出されたオフセットが 16 ビットで表現できる範囲でない場合，リンク時にエラーになります。
!label	ラベル label の絶対アドレスがイミーディエトとして扱われます。 16 ビットの値を持ち，イミーディエトとして 16 ビットの値を指定できるアーキテクチャの演算命令[注]に指定した場合，命令展開は行われません。add，mov，および mulh 命令に指定した場合，適切な 1 命令に展開されます。それ以外の命令に指定することはできません。16 ビットで表現できる範囲でない場合，リンク時にエラーになります。
%label	ラベル label のセクション内オフセットがイミーディエトとして扱われます。 ラベル label が ep シンボルの対象となっているセクションに配置された場合には，ep シンボルからのオフセットをディスプレースメントとして扱われます。 16 ビットの値を持ち，イミーディエトとして 16 ビットの値を指定できるアーキテクチャの演算命令[注]に指定した場合，命令展開は行われません。 ただし，label 参照とは異なり，命令展開は行われません。また，この参照方法は，イミーディエトとして 16 ビットの値を指定できるアーキテクチャの演算命令と，add，mov，および mulh 命令のみで指定できます。add，mov，および mulh 命令に指定した場合，適切な 1 命令に展開します。それ以外の命令に指定することはできません。16 ビットで表現できる範囲でない場合，リンク時にエラーになります。

注 イミーディエトとして 16 ビットの値を指定できる命令は，addi，andi，movea，mulhi，ori，satsubi，および xori です。

表 4―9 分岐命令

参照方法	意味
#label	jmp 命令において，ラベル label の絶対アドレスが飛び先アドレスとして扱われます。 32 ビットの値を持ち，3 命令に展開されますが，V850E の場合は 2 命令に展開されます。
label	jmp 命令以外の分岐命令において，ラベル label の PC オフセットがディスプレースメントとして扱われます。 22 ビットの値を持ち，表現できない範囲である場合，リンク時にエラーになります。

表 4―10 領域確保議事命令

参照方法	意味
#label !label	.word ／ .hword ／ .byte 疑似命令において，ラベル label の絶対アドレスを値として扱われます。 32 ビットの値を持ちますが，各疑似命令のビット幅に応じてマスクされます。
label %label	.word ／ .hword ／ .byte 疑似命令において，ラベル label の定義されたセクション内オフセットを値として扱われます。 32 ビットの値を持ちますが，各疑似命令のビット幅に応じてマスクされます。
$label	.word ／ .hword ／ .byte 疑似命令において，ラベル label の gp オフセットを値として扱われます。 32 ビットの値を持ちますが，各疑似命令のビット幅に応じてマスクされます。

（5）ep オフセット参照

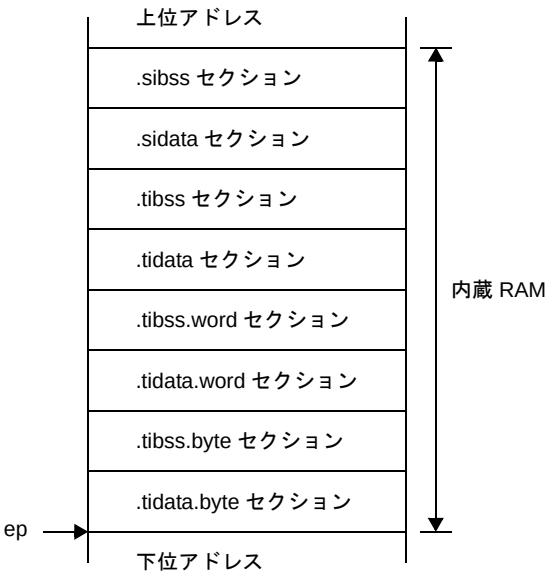
CA850 では，明示的に内蔵 RAM に置かれるデータについては，基本的に，次のことが想定されています。

エレメント・ポインタ（ep）の指すアドレスからのオフセットによって参照する

なお，内蔵 RAM に置かれるデータは，次の 2 つに分けられます。

- .tidata ／ .tibss ／ .tidata.byte ／ .tibss.byte ／ .tidata.word ／ .tibss.word セクション
  コード・サイズが小さいメモリ参照命令（sld ／ sst）で参照するデータ
- .sidata ／ .sibss セクション
  コード・サイズが大きいメモリ参照命令（ld ／ st）で参照するデータ

図 4―3 内蔵 RAM のメモリ配置イメージ



(a) データの割り当て

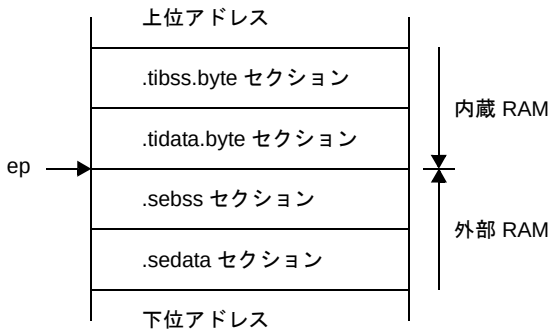
内蔵 RAM に置くセクションへのデータの割り当ては，次の方法で行います。

- C 言語を用いてプログラムを作成する場合
  - “#pragma section”指令により，セクション種別“tidata”，“tidata.byte”，“tidata.word”，または“sidata”を指定してデータを割り当てます。
  - セクション・ファイルで，セクション種別“tidata”，“tidata.byte”，“tidata.word”，または“sidata”を指定してデータを割り当てます。ca850 のオプションにより，そのセクション・ファイルをコンパイル時に入力します。

- アセンブリ言語を用いてプログラムを作成する場合

  セクション定義疑似命令により，セクション種別 .tidata.byte，.tibss.byte，.tidata.word，.tibss.word，.sidata，または .sibss のセクションへデータを割り当てます。なお，上記と同様の方法により，.sedata，または .sebss セクションへデータを割り当てることで，外部 RAM に置かれる一定範囲のデータに対しても，ep オフセット参照を行うことができます。

図 4―4 外部 RAM（.sedata，／.sebss セクション）のメモリ配置イメージ



（b）データの参照

「（a）データの割り当て」のデータの割り付けに従い，as850 では，次のように機械語命令列が生成されます。

- .tidata，.tibss，.tidata.byte，.tibss.byte，.tidata.word，.tibss.word，.sidata，.sibss，.sedata，または .sebss セクションに割り当てるデータの %label による参照に対しては，ep オフセット参照を行う機械語命令が生成
- 上記以外のセクションに割り当てるデータの %label による参照に対しては，セクション内オフセット参照を行う機械語命令列が生成

例

```
    .sidata
sidata: .hword  0xfff0
    .data
data:   .hword  0xfff0
    .text
    ld.h    %sidata, r20    --(1)
    ld.h    %data, r20      --(2)
```

as850 では，%label による参照に対して，(1) の場合，定義したデータが .sidata セクションに配置されているため ep オフセット参照とみなされ，(2) の場合，セクション内オフセット参照とみなされて，機械語の命令列が生成されます。なお，as850 では，データが配置されたセクションが正しいものとして処理が行われます。このため，データの配置に誤りがある場合でも，検出できません。

例

```
.text
ld.h    %label[ep], r20
```

label を .sidata セクションに配置し，ep オフセット参照を行う命令を記述したが，配置誤りにより .data セクションに配置された場合，ベース・レジスタである ep シンボル値 + label の .data セクション内オフセット値にあるデータがロードされます。

例

```
.text
ld.h    %label1[r10], r20   --  (1)
.option ep_label
ld.h    %label2[ep], r21    --  (2)
.option no_ep_label
ld.h    %label3[r10], r22   --  (3)
```

(1)：
定義したデータが配置されたセクションによって，ep オフセット参照，またはセクション内オフセット参照となります（デフォルト）。
(2)：
.option ep_label 疑似命令によって指定された範囲内であるので，定義したデータが配置されたセクションにかかわらず，ep オフセット参照となります。
(3)：
.option no_ep_label 疑似命令によって指定された範囲内であるので，(1) の場合と同じ扱いとなります。

(6) gp オフセット参照

CA850 では，(.sedata，／ .sebss セクション以外の）外部 RAM に置かれるデータについては，基本的に次のことが想定されています。

グローバル・ポインタ（gp）の指すアドレスからのオフセットによって参照する

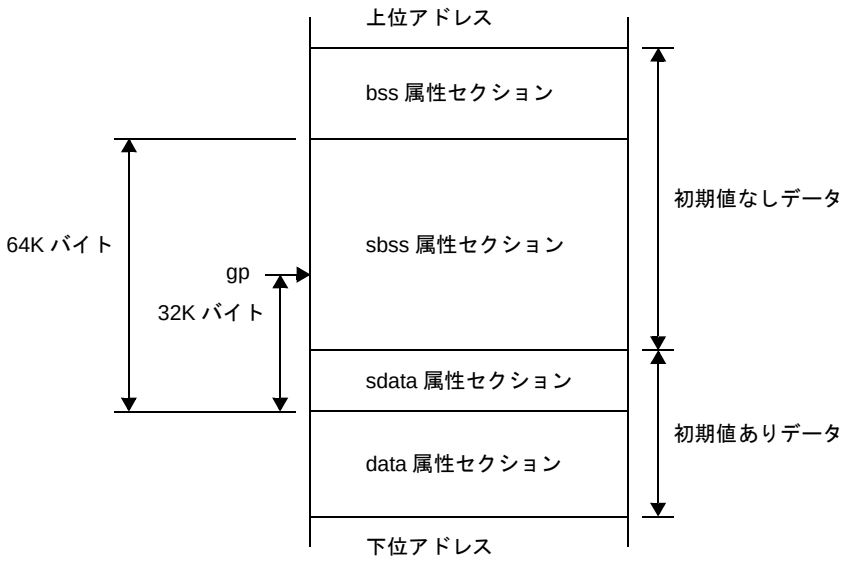
また，C 言語における“#pragma section”指令や，C コンパイラに入力するセクション・ファイル，アセンブリ言語におけるセクション定義疑似命令による，内蔵 ROM ／内蔵 RAM などの r0 相対のメモリ割り当てを行わない場合，すべてのデータが gp オフセット参照となります。

(a) データの割り当て

V850 マイクロコントローラの機械語命令におけるメモリ参照命令（ld ／ st）は，ディスプレースメントに 16 ビットのイミーディエトしかとれません。このため，CA850 ではデータを以下の 2 つに分け，前者のデータが sdata 属性セクション，または sbss 属性セクションに，後者のデータが data 属性セクション，または bss 属性セクションに割り当てられます。なお，初期値ありデータは sdata ／ data 属性セクションに，初期値なしデータは sbss ／ bss 属性セクションに割り当てられます。CA850 では，デフォルトで，data ／ sdata ／ sbss ／ bss 属性セクションの順に下位アドレスから割り当てられます。また，グローバル・ポインタ（gp）は sdata 属性セクションの先頭アドレスに 32 K バイトを加算したアドレスを指すよう，スタート・アップ・モジュールなどにおいて設定することを想定しています。

- グローバル・ポインタ（gp）と 16 ビットのディスプレースメントを用いて参照することのできるメモリ範囲に割り当てるデータ
- グローバル・ポインタ（gp）と（複数の命令によって構成される）32 ビットのディスプレースメントを用いて参照することのできるメモリ範囲に割り当てるデータ

図 4―5　gp オフセット参照セクションのメモリ配置イメージ



備考　sdata 属性セクションと sbss 属性セクションをあわせて 64 K バイトです。gp は sdata 属性セクションの先頭から 32 K バイトの位置です。

したがって，sdata ／ sbss 属性セクションのデータを参照する場合は，1 命令で行えますが，data ／ bss 属性セクションのデータを参照する場合は，複数の命令が必要となります。つまり，より多くのデータを sdata ／ sbss 属性セクションに割り当てたほうが，生成された機械語命令の実行効率，およびオブジェクト効率は向上します。しかし，16 ビットのディスプレースメントで参照できるメモリ範囲の大きさは限られています。

そこで，すべてのデータを sdata ／ sbss 属性セクションに割り当てられない場合，どのデータを sdata ／ sbss 属性セクションに割り当てるかを定めなければなりません。

CA850 では，“できるだけ多くのデータを sdata ／ sbss 属性セクションに割り当てる”という方針がとられています。デフォルトの処理では，すべてのデータが sdata ／ sbss 属性セクションに割り当てられますが，割り当てられるデータを選別する場合，次の方法により行います。

- -G*num* オプションを指定する場合

  C コンパイラ（ca850），またはアセンブラ（as850）起動時に -G*num* オプションを指定することにより，sdata ／ sbss 属性セクションへ *num* バイト以下のデータが割り当てられます。

- プログラムでデータを割り当てるセクションを指定する場合

  参照頻度の高いデータを，明示的に sdata ／ sbss 属性セクションへ割り当てます。アセンブリ言語の場合，セクション定義疑似命令で，C 言語の場合，#pragma section 指令で割り当てられます。

- セクション・ファイルで指定する場合

  C 言語の場合，セクション・ファイルでセクション種別“sdata”を指定してデータを割り当てます。ca850 のオプションにより，そのセクション・ファイルをコンパイル時に入力します。

(b) データの参照

「(a) データの割り当て」のデータの割り付けに従い，as850 では，次のように機械語命令列が生成されます。

- sdata ／ sbss 属性セクションに割り当てるデータの gp オフセット参照に対しては，16 ビットのディスプレースメントを用いた参照を行う機械語命令が生成
- data ／ bss 属性セクションに割り当てるデータの gp オフセット参照に対しては，（複数の機械語命令によって構成される）32 ビットのディスプレースメントを用いた参照を行う機械語命令列が生成

例

```
    .data
data:   .word   0xfff00010  --(1)
    .text
    ld.w    $data[gp], r20  --(2)
```

as850 では，(1) で定義したデータを gp オフセット参照している (2) の ld.w 命令に対して，次の命令列に等価な機械語の命令列が生成されます。

```
movhi   hi1($data), gp, r1
ld.w    lo($data)[r1], r20
```

なお，as850 では，1 ファイルずつ処理が行われます。このため，指定したファイル内に定義を持つデータに対しては，そのデータがどの属性セクションに割り当てられるデータであるかを判断することができますが，指定したファイル内の定義を持たないデータに対しては判定できません。そこで，as850 では，“前述の割り当ての方針（一定サイズ以下のデータを sdata ／ sbss 属性セクションに割り当てる）が守られている”ことを想定し，起動時に -G*num* オプションが指定された場合，次のように機械語命令が生成されます**注**。

注 .option 疑似命令のオプションで，data，または sdata が指定されたデータに対しては，サイズに関係なく .data セクション，.sdata セクションへ割り当てられるものとして扱われます。

- 指定したファイル内に定義を持たない，num バイト以下のデータの gp オフセット参照に対しては，16 ビットのディスプレースメントを用いた参照を行う機械語命令が生成されます。
- 指定したファイル内に定義を持たない，num バイトより大きいデータの gp オフセット参照に対しては，（複数の機械語命令によって構成される）32 ビットのディスプレースメントを用いた参照を行う機械語命令列が生成されます。

しかし，この判定を行うには，指定したファイル内に定義を持たない gp オフセット参照されているデータのサイズが，判定できなければなりません。そこで，アセンブリ言語を用いてプログラムを作成する場合，指定したファイル内に定義を持たないデータ（実際には指定したファイル内に定義を持たず gp オフセット参照されているラベル）に対し，.extern 疑似命令を用いて，サイズを指定してください。

例

```
.extern data, 4          --(1)
.text
ld.w    $data[gp], r20   --(2)
```

as850 では，起動時に -G2 を指定した場合，(1) で宣言したデータを gp オフセット参照している (2) の ld.w 命令に対しては，次の命令列に等価な機械語の命令列が生成されます。

```
movhi   hi1($data), gp, r1
ld.w    lo($data)[r1], r20
```

また，C 言語を用いてプログラム作成する場合，CA850 に含まれる C コンパイラ（ca850）が，指定したファイル内に定義を持たないデータ（実際には指定したファイル内に定義を持たず gp オフセット参照されているラベル）に対して，自動的に .extern 疑似命令を生成し，サイズを指定するコードを出力します。

【 まとめ 】

as850 におけるデータの gp オフセット参照（具体的には，ラベルの gp オフセット参照を持つ相対値式をディスプレースメントとするメモリ参照命令）の扱いをまとめると，次のようになります。

- そのデータが，指定したファイル内に定義を持つ場合
  - sdata ／ sbss 属性セクションに割り付けるデータである場合[注]
    16 ビットのディスプレースメントを用いた参照を行う機械語命令が生成されます。
  - sdata ／ sbss 属性セクションに割り付けるデータでない場合
    32 ビットのディスプレースメントを用いた参照を行う機械語命令列が生成されます。

注 “ラベル±定数式”の形式の相対値式で，定数式の値が 16 ビットの範囲を越える場合は，as850 では，32 ビットのディスプレースメントを用いた参照を行う機械語命令列が生成されます。

- そのデータが，指定したファイル内に定義を持たない場合
  - 起動時に -G*num* オプションを指定した場合
    データ（gp オフセット参照されているラベル）に対し，.comm ／ .extern ／ .globl ／ .lcomm ／ .size 疑似命令により，
    【 0 以外かつ num バイト以下を指定した場合 】
    sdata ／ sbss 属性セクションに割り当てるデータであるとみなされ，16 ビットのディスプレースメントを用いた参照を行う機械語命令が生成されます。
    【 上記以外の場合 】
    sdata ／ sbss 属性セクションに割り当てるデータでないとみなされ，32 ビットのディスプレースメントを用いた参照を行う機械語命令が生成されます。

- 起動時に -G*num* オプションを指定しなかった場合

  sdata ／ sbss 属性セクションに割り当てるデータであるとみなされ，16 ビットのディスプレースメントを用いた参照を行う機械語命令が生成されます。

（7）hi ( ) ／ lo ( ) ／ hi1 ( ) について

（a）32 ビットの定数値をレジスタに入れる場合

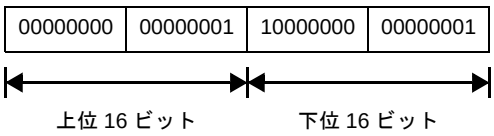
V850 マイクロコントローラの V850 コアでは，1 命令で 32 ビットの定数値をレジスタに格納する機械語命令を持ちません。このため，as850 では，32 ビットの定数値をレジスタに格納する場合，命令展開を行い movhi 命令と movea 命令を用いて，32 ビットの定数値を上位 16 ビットと下位 16 ビットに分けてレジスタに格納する命令列が生成されます。

例

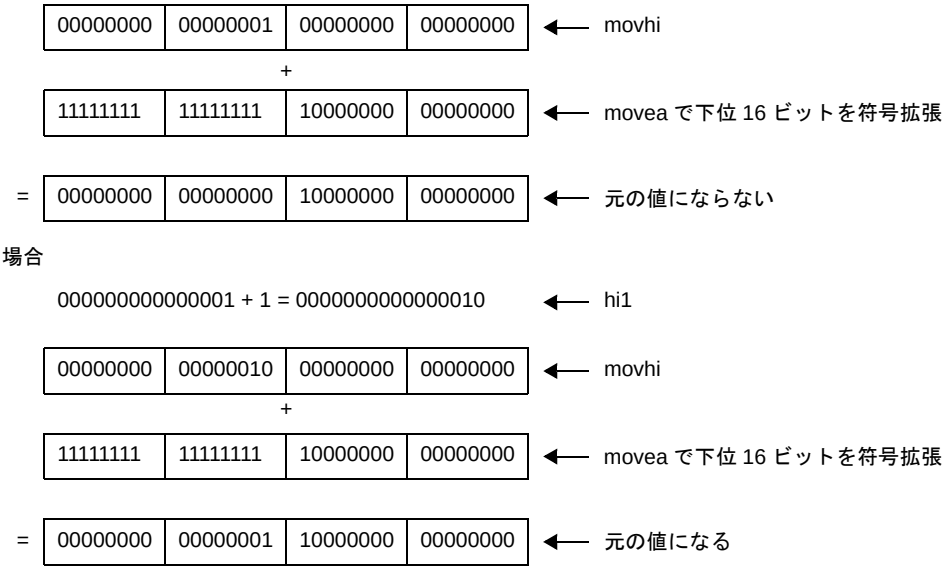
`mov 0x18000, r11`	`movhi hi1(0x18000), r0, r1` `movea lo(0x18000), r1, r11`

この際，下位 16 ビットの格納に用いられる機械語の movea 命令は，指定された 16 ビットの値を符号拡張し 32 ビットの値として扱われます。この符号拡張された部分を補正するため as850 では，movhi 命令を用いて上位 16 ビットをレジスタに格納する際，ただ単に上位 16 ビットをレジスタに格納するのではなく，以下の値をレジスタに格納します。

上位 16 ビット + 下位 16 ビットの最上位ビット（ビット番号 15 のビット）



補正しない場合



補正する場合

（b）32 ビットのディスプレースメントを用いてメモリを参照する場合

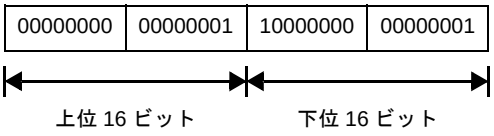
V850 マイクロコントローラの機械語のメモリ参照命令（ロード／ストア命令，およびビット操作命令）は，ディスプレースメントに 16 ビットのイミーディエトしかとれません。このため，as850 では，32 ビットのディスプレースメントを用いてメモリを参照する場合，命令展開が行われ，movhi 命令とメモリ参照命令が用いられて，32 ビットのディスプレースメントの上位 16 ビットと，下位 16 ビットから，32 ビットのディスプレースメントが構成されて参照を行う命令列が生成されます。

例

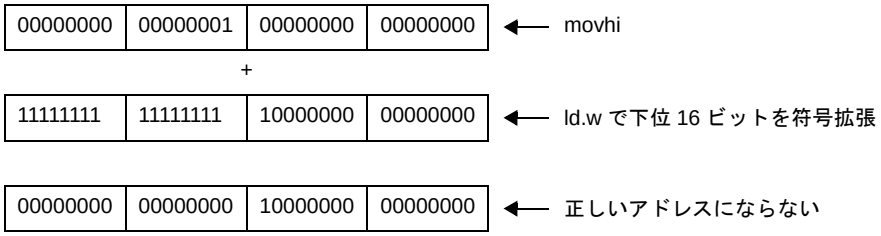
`ld.w  0x18000[r11], r12`	`movhi  hi1(0x18000), r11, r1` `ld.w  lo(0x18000)[r1], r12`

この際，下位 16 ビットをディスプレースメントとして用いる機械語のメモリ参照命令は，指定された 16 ビットのディスプレースメントを符号拡張し，32 ビットの値として扱います。この符号拡張された部分を補正するために，as850 では，movhi 命令を用いて上位 16 ビットのディスプレースメントを構成する際，ただ単に上位 16 ビットのディスプレースメントを構成するのではなく，次ののディスプレースメントを構成します。

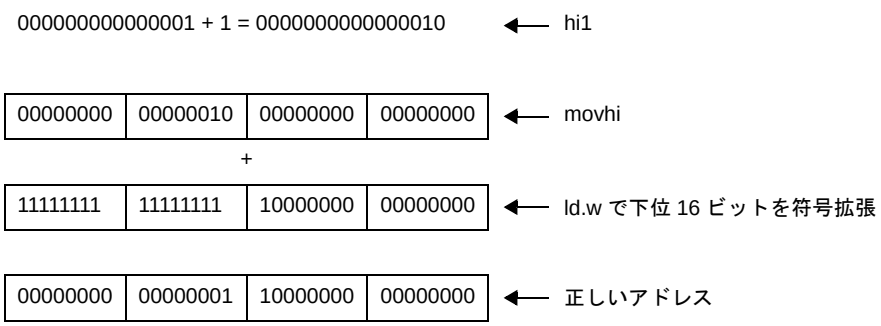
上位 16 ビット + 下位 16 ビットの最下位ビット（ビット番号 15 のビット）



補正しない場合



補正する場合



(c) hi ( ) ／ lo ( ) ／ hi1 ( )

次表のように as850 では，hi ( )，lo ( )，および hi1 ( ) を用いることにより，32 ビットの値の上位 16 ビット，32 ビットの値の下位 16 ビット，および 32 ビットの値の上位 16 ビット + ビット番号 15 のビット値を指定できます**注**。

注　アセンブラ内部では解決できない場合，この情報はリロケーション情報に反映されリンカ（ld850）において解決されます。

表 4―11　領域確保議事命令

hi ( ) ／ lo ( ) ／ hi1 ( )	意味
hi (*value*)	value の上位 16 ビット
lo (*value*)	value の下位 16 ビット
hi1 (*value*)	*value* の上位 16 ビット + *value* のビット番号 15 のビット値

例

```
    .data
L1:
      :
    .text
    movhi   hi($L1), r0, r10  --L1 の gp オフセットの値の上位 16 ビットを r10 の上位
                              --16 ビットに格納し，下位 16 ビットに 0 を格納する。
    movea   lo($L1), r0, r10  --L1 の gp オフセットの値の下位 16 ビットを符号拡張し，
                              --r10 に格納する。
      :
    movhi   hi1($L1), r0, r1  --L1 の gp オフセットの値を r10 に格納する。
    movea   lo($L1), r1, r10
```

4. 2　疑似命令

この節では，CA850 アセンブラ（as850）がサポートする疑似命令について説明します。

4. 2. 1　概　　要

疑似命令とは，アセンブラが機械語命令を生成するために必要な前処理を行うものです。アセンブラに対し，セクション定義やファイル入力などを指示します。また，条件による出力コードの加工やマクロ置換などの指示もできます。

4. 2. 2 セクション定義疑似命令

as850 では，セクション定義疑似命令を用いることにより，ソース・プログラム（アセンブリ言語）に対して生成するコードを指定したセクション[注]に割り当てることができます。次に，この項において説明するセクション定義疑似命令を示します。

注 CA850 では，機械語命令やデータをセクションと呼ばれる単位に割り当てて扱います。

表 4―12 セクション定義疑似命令

疑似命令	意味
.tidata	.tidata セクションへの割り当て
.tidata.byte	.tidata.byte セクションへの割り当て
.tidata.word	.tidata.word セクションへの割り当て
.tibss	.tibss セクションへの割り当て
.tibss.byte	.tibss.byte セクションへの割り当て
.tibss.word	.tibss.word セクションへの割り当て
.data	.data セクションへの割り当て
.bss	.bss セクションへの割り当て
.sdata	.sdata セクションへの割り当て
.sbss	.sbss セクションへの割り当て
.sedata	.sedata セクションへの割り当て
.sebss	.sebss セクションへの割り当て
.sidata	.sidata セクションへの割り当て
.sibss	.sibss セクションへの割り当て
.sconst	.sconst セクションへの割り当て
.const	.const セクションへの割り当て
.text	.text セクションへの割り当て
.vdbstrtab	.vdbstrtab セクションへの割り当て
.vdebug	.vdebug セクションへの割り当て
.vline	.vline セクションへの割り当て
.section	指定した種類のセクションへの割り当て
.previous	現在のセクション定義疑似命令を指定しているセクション定義疑似命令の前のセクション定義疑似命令の（再）指定

なお，アセンブリ言語ソース・プログラム中にセクション定義疑似命令が 1 つもない場合，そのプログラムで生成されるセクションは，.text セクションとなります。

.tidata

.tidata セクションへ割り当てます。

[指定形式]

.tidata

[機能]

次のセクション定義疑似命令まで，または次のセクション定義疑似命令が現れなかった場合は，そのアセンブリ言語ソース・ファイルの終わりまでの間に記述したアセンブリ言語ソース・プログラムに対して生成するコードを，.tidata セクション[注]に割り当てます。

注 セクション名 .tidata，セクション・タイプ PROGBITS，およびセクション属性 AW を持つ予約セクションです。

[詳細説明]

.tidata セクションは，V850 マイクロコントローラの内蔵 RAM に置かれ，sld ／ sst 命令を用いて ep 相対でアクセスされることを前提としています。as850，および ld850 は，.tidata.byte，.tibss.byte，.tidata.word，.tibss.word のいずれのセクションも使用されていない場合，ep の示すアドレスに .tidata を配置します。.tidata.byte，.tibss.byte，.tidata.word，.tibss.word のいずれかのセクションが使用されている場合，“ep の示すアドレス + 使用された .tidata.byte ／ .tibss.byte ／ .tidata.word ／ .tibss.word セクションのサイズ”のアドレスに，.tidata を配置します。

sld ／ sst 命令では，データのサイズによってアクセスする領域の範囲が異なるため，より有効に sld ／ sst 命令を用いるためには，バイト・データは .tidata.byte ／ .tibss.byte セクションに，ハーフワード以上のデータは .tidata.word ／ .tibss.word セクションに置くことを推奨します。ただし，内蔵 RAM に置くデータが少なく，アクセス領域を細かく考慮する必要がない場合，本疑似命令でデータを .tidata セクションに置いて，データをサイズで分ける手間を省くこともできます。

[使用例]

次のセクション定義疑似命令まで .tidata セクションとなる。

```
    .tidata
    .align  4
    .globl  _p, 4
_p:
    .word   10
```

.tidata.byte

.tidata.byte セクションへ割り当てます。

[指定形式]

.tidata.byte

[機能]

次のセクション定義疑似命令まで，または次のセクション定義疑似命令が現れなかった場合は，そのアセンブリ言語ソース・ファイルの終わりまでの間に記述したアセンブリ言語ソース・プログラムに対して生成するコードを，.tidata.byte セクション[注]に割り当てます。

注 セクション名 .tidata.byte，セクション・タイプ PROGBITS，およびセクション属性 AW を持つ予約セクションです。

[詳細説明]

.tidata.byte セクションは，V850 マイクロコントローラの内蔵 RAM に置かれ，sld ／ sst 命令を用いて ep 相対でアクセスされることを前提としています。sld ／ sst 命令では，以下の領域範囲をアクセスできます。

- アクセスするデータがバイト・データの場合：128 バイト以内
- ハーフワード以上のデータの場合： 256 バイト以内

as850，および ld850 は，sld ／ sst 命令がアクセスできる領域を有効に使用するため，データのサイズによって，セクションを，.tidata.byte ／ .tibss.byte と .tidata.word ／ .tibss.word に分け，.tidata.byte を ep が示すアドレスへ配置するようにしています。したがって，内蔵 RAM に置く初期値ありバイト・データは，本疑似命令により .tidata.byte セクションに置くことを推奨します。[注]

注 初期値ありバイト・データを .tidata.word セクションに置いてアクセスすること自体はできます。

[使用例]

次のセクション定義疑似命令まで .tidata.byte セクションとなる。

```
    .tidata.byte
    .globl  _p, 1
_p:
    .byte   1
```

.tidata.word

.tidata.word セクションへ割り当てます。

[指定形式]

.tidata.word

[機能]

次のセクション定義疑似命令まで，または次のセクション定義疑似命令が現れなかった場合は，そのアセンブリ言語ソース・ファイルの終わりまでの間に記述したアセンブリ言語ソース・プログラムに対して生成するコードを，.tidata.word セクション注に割り当てます。

注 セクション名 .tidata.word，セクション・タイプ PROGBITS，およびセクション属性 AW を持つ予約セクションです。

[詳細説明]

.tidata.word セクションは，V850 マイクロコントローラの内蔵 RAM に置かれ，sld ／ sst 命令を用いて ep 相対でアクセスされることを前提としています。sld ／ sst 命令では，以下の領域範囲をアクセスできます。

- アクセスするデータがバイト・データの場合：128 バイト以内
- ハーフワード以上のデータの場合： 256 バイト以内

as850，および ld850 は，sld ／ sst 命令がアクセスできる領域を有効に使用するため，データのサイズによって，セクションを，.tidata.byte ／ .tibss.byte と .tidata.word ／ .tibss.word に分け，.tidata.word を，“ep の示すアドレス + .tidata.byte ／ .tibss.byte セクションのサイズ”のアドレスへ配置するようにしています。したがって，内蔵 RAM に置くハーフワード以上の初期値ありデータは，本疑似命令により .tidata.word セクションに置くようにしてください。

[使用例]

次のセクション定義疑似命令まで .tidata.word セクションとなる。

```
    .tidata.word
    .align  4
    .globl  _p, 4
_p:
    .word   100000
```

.tibss

.tibss セクションへ割り当てます。

[指定形式]

.tibss

[機能]

次のセクション定義疑似命令まで，または次のセクション定義疑似命令が現れなかった場合は，そのアセンブリ言語ソース・ファイルの終わりまでの間に記述したアセンブリ言語ソース・プログラムに対して生成するコードを，.tibss セクション[注]に割り当てます。

注 セクション名 .tibss，セクション・タイプ NOBITS，およびセクション属性 AW を持つ予約セクションです。

[詳細説明]

.tidbss セクションは，V850 マイクロコントローラの内蔵 RAM に置かれ，sld ／ sst 命令を用いて ep 相対でアクセスされることを前提とした初期値なしデータです。as850，および ld850 は，.tidata.byte，.tibss.byte，.tidata.word，.tibss.word，.tidata のいずれのセクションも使用されていない場合，ep の指すアドレスに .tibss を配置します。.tidata.byte，.tibss.byte，.tidata.word，.tibss.word，.tidata のいずれかのセクションが使用されている場合，“ep の示すアドレス + 使用された .tidata.byte ／ .tibss.byte ／ .tidata.word ／ .tibss.word ／ .tidata のセクションのサイズ”のアドレスに，.tibss を配置します。

sld ／ sst 命令では，データのサイズによってアクセスする領域の範囲が異なるため，より有効に sld ／ sst 命令を用いるためには，バイト・データは .tidata.byte ／ .tibss.byte セクションに，ハーフワード以上のデータは .tidata.word ／ .tibss.word セクションに置くことを推奨します。ただし，内蔵 RAM に置くデータが少なく，アクセス領域を細かく考慮する必要がない場合，初期値なしデータに関しては本疑似命令でデータを .tibss セクションに置いて，データをサイズで分ける手間を省くこともできます。

[使用例]

次のセクション定義疑似命令まで .tibss セクションとなる。

```
.tibss
.globl  _1, 4
.lcomm  _1, 4, 4
```

.tibss.byte

.tibss.byte セクションへ割り当てます。

[指定形式]

.tibss.byte

[機能]

次のセクション定義疑似命令まで，または次のセクション定義疑似命令が現れなかった場合は，そのアセンブリ言語ソース・ファイルの終わりまでの間に記述したアセンブリ言語ソース・プログラムに対して生成するコードを，.tibss.byte セクション注に割り当てます。

注 セクション名 .tibss.byte，セクション・タイプ NOBITS，およびセクション属性 AW を持つ予約セクションです。

[詳細説明]

.tibss.byte セクションは，V850 マイクロコントローラの内蔵 RAM に置かれ，sld ／ sst 命令を用いて ep 相対でアクセスされることを前提としています。sld ／ sst 命令では，以下の領域範囲をアクセスできます。

- アクセスするデータがバイト・データの場合：128 バイト以内
- ハーフワード以上のデータの場合： 256 バイト以内

as850，および ld850 は，sld ／ sst 命令がアクセスできる領域を有効に使用するため，データのサイズによって，セクションを，.tidata.byte ／ .tibss.byte と .tidata.word ／ .tibss.word に分け，.tibss.byte を，“ep の示すアドレスに + 使用された .tidata.byte セクションのサイズ”のアドレスへ配置するようにしています。したがって，内蔵 RAM に置く初期値なしバイト・データは，本疑似命令により，.tibss.byte セクションに置くことを推奨します。注

注 バイト・データを .tibss.word セクションに置いてアクセスすること自体はできます。

[使用例]

次のセクション定義疑似命令まで .tibss.byte セクションとなる。

```
.tibss.byte
.globl  _1, 1
.lcomm  _1, 1, 1
```

.tibss.word

.tibss.word セクションへ割り当てます。

[指定形式]

.tibss.word

[機能]

次のセクション定義疑似命令まで，または次のセクション定義疑似命令が現れなかった場合は，そのアセンブリ言語ソース・ファイルの終わりまでの間に記述したアセンブリ言語ソース・プログラムに対して生成するコードを，.tibss.word セクション[注]に割り当てます。

注 セクション名 .tibss.word，セクション・タイプ NOBITS，およびセクション属性 AW を持つ予約セクションです。

[詳細説明]

.tibss.word セクションは，V850 マイクロコントローラの内蔵 RAM に置かれ，sld ／ sst 命令を用いて ep 相対でアクセスされることを前提としています。sld ／ sst 命令では，以下の領域範囲をアクセスできます。

- アクセスするデータがバイト・データの場合：128 バイト以内
- ハーフワード以上のデータの場合： 256 バイト以内

as850，および ld850 は，sld ／ sst 命令がアクセスできる領域を有効に使用するため，データのサイズによって，セクションを，.tidata.byte ／ .tibss.byte と .tidata.word ／ .tibss.word に分け，.tibss.word を，“ep の示すアドレス + 使用された .tidata.byte ／ .tibss.byte ／ .tidata.word セクションのサイズ”のアドレスへ配置するようにしています。したがって，内蔵 RAM に置くハーフワード以上の初期値なしデータは，本疑似命令により .tibss.word セクションに置くようにしてください。

[使用例]

次のセクション定義疑似命令まで .tibss.word セクションとなる。

```
.tibss.word
.globl  _1, 4
.lcomm  _1, 4, 4
```

.data

.data セクションへ割り当てます。

[指定形式]

.data

[機能]

次のセクション定義疑似命令まで，または次のセクション定義疑似命令が現れなかった場合は，そのアセンブリ言語ソース・ファイルの終わりまでの間に記述したアセンブリ言語ソース・プログラムに対して生成するコードを，.data セクション注に割り当てます。

注 セクション名 .data，セクション・タイプ PROGBITS，およびセクション属性 AW を持つ予約セクションです。

[詳細説明]

.data セクションは，初期値を持ち，gp と 2 命令によって構成される，32 ビットのディスプレースメントを用いて参照されるメモリ範囲に配置されるセクションです。

[使用例]

次のセクション定義疑似命令まで .data セクションとなる。

```
    .data
    .align  4
    .globl  _p, 4
_p:
    .word   10
```

.bss

.bss セクションへ割り当てます。

[指定形式]

.bss

[機能]

次のセクション定義疑似命令まで，または次のセクション定義疑似命令が現れなかった場合は，そのアセンブリ言語ソース・ファイルの終わりまでの間に記述したアセンブリ言語ソース・プログラムに対して生成するコードを，.bss セクション[注]に割り当てます。

注 セクション名 .bss，セクション・タイプ NOBITS，およびセクション属性 AW を持つ予約セクションです。

[詳細説明]

.bss セクションは，初期値を持たず，gp と 2 命令によって構成される，32 ビットのディスプレースメントを用いて参照されるメモリ範囲に配置されるセクションです。

[使用例]

次のセクション定義疑似命令まで .bss セクションとなる

```
.bss
.lcomm  _stack, 0x100, 4
```

.sdata

.sdata セクションへ割り当てます。

[指定形式]

.sdata

[機能]

次のセクション定義疑似命令まで，または次のセクション定義疑似命令が現れなかった場合は，そのアセンブリ言語ソース・ファイルの終わりまでの間に記述したアセンブリ言語ソース・プログラムに対して生成するコードを，.sdata セクション[注]に割り当てます。

注 セクション名 .sdata，セクション・タイプ PROGBITS，およびセクション属性 AWG を持つ予約セクションです。

[詳細説明]

.sdata セクションは，初期値を持ち，gp と 16 ビットのディスプレースメントを用いて 1 命令で参照されるメモリ範囲（.sbss セクションとあわせて最大 64 K バイト）に配置されるセクションです。

[使用例]

次のセクション定義疑似命令まで .sdata セクションとなる。

```
    .sdata
    .align  4
    .globl  _p, 4
_p:
    .word   10
```

.sbss

.sbss セクションへ割り当てます。

[指定形式]

.sbss

[機能]

次のセクション定義疑似命令まで，または次のセクション定義疑似命令が現れなかった場合は，そのアセンブリ言語ソース・ファイルの終わりまでの間に記述したアセンブリ言語ソース・プログラムに対して生成するコードを，.sbss セクション注に割り当てます。

注 セクション名 .sbss，セクション・タイプ NOBITS，およびセクション属性 AWG を持つ予約セクションです。

[詳細説明]

.sbss セクションは，初期値を持たず，gp と 16 ビットのディスプレースメントを用いて 1 命令で参照されるメモリ範囲（.sdata セクションとあわせて最大 64 K バイト）に配置されるセクションです。

[使用例]

次のセクション定義疑似命令まで .sbss セクションとなる。

```
.sbss
.globl  _1, 4
.lcomm  _1, 4, 4
```

.sedata

.sedata セクションへ割り当てます。

［指定形式］

.sedata

［機能］

次のセクション定義疑似命令まで，または次のセクション定義疑似命令が現れなかった場合は，そのアセンブリ言語ソース・ファイルの終わりまでの間に記述したアセンブリ言語ソース・プログラムに対して生成するコードを，.sedata セクション注に割り当てます。

注　セクション名 .sedata，セクション・タイプ PROGBITS，およびセクション属性 AW を持つ予約セクションです。

［詳細説明］

.sedata セクションは，初期値を持ち，ep と 16 ビットのディスプレースメントを用いて 1 命令で参照されるメモリ範囲（ep からマイナス方向に最大 32 K バイト）のうち，.sebss セクションのサイズ分だけ下位のアドレスに配置されるセクションです。

［使用例］

次のセクション定義疑似命令まで .sedata セクションとなる。

```
    .sedata
    .align  4
    .globl  _p, 4
_p:
    .word   10
```

.sebss

.sebss セクションへ割り当てます。

［指定形式］

.sebss

［機能］

次のセクション定義疑似命令まで，または次のセクション定義疑似命令が現れなかった場合は，そのアセンブリ言語ソース・ファイルの終わりまでの間に記述したアセンブリ言語ソース・プログラムに対して生成するコードを，.sebss セクション注に割り当てます。

注 セクション名 .sebss，セクション・タイプ NOBITS，およびセクション属性 AW を持つ予約セクションです。

［詳細説明］

.sebss セクションは，初期値を持たず，ep と 16 ビットのディスプレースメントを用いて 1 命令で参照されるメモリ範囲（ep からマイナス方向に最大 32 K バイト）のうち，.sedata セクションのサイズ分だけ上位のアドレスに配置されるセクションです。

［使用例］

次のセクション定義疑似命令まで .sebss セクションとなる。

```
.sebss
.globl  _1, 4
.lcomm  _1, 4, 4
```

.sidata

.sidata セクションへ割り当てます。

[指定形式]

.sidata

[機能]

次のセクション定義疑似命令まで，または次のセクション定義疑似命令が現れなかった場合は，そのアセンブリ言語ソース・ファイルの終わりまでの間に記述したアセンブリ言語ソース・プログラムに対して生成するコードを，.sidata セクション[注]に割り当てます。

注 セクション名 .sidata，セクション・タイプ PROGBITS，およびセクション属性 AW を持つ予約セクションです。

[詳細説明]

.sidata セクションは，ep と 16 ビットのディスプレースメントを用いて 1 命令で参照されるメモリ範囲（ep からプラス方向に最大 32 K バイト）のうち，使用された .tidata.byte，.tibss.byte，.tidata.word，.tibss.word，.tidata，.tibss セクションのサイズ分だけ上位のアドレスに配置されるセクションです。

[使用例]

次のセクション定義疑似命令まで .sidata セクションとなる。

```
    .sidata
    .align  4
    .globl  _p, 4
_p:
    .word   10
```

.sibss

.sibss セクションへ割り当てます。

[指定形式]

.sibss

[機能]

次のセクション定義疑似命令まで，または次のセクション定義疑似命令が現れなかった場合は，そのアセンブリ言語ソース・ファイルの終わりまでの間に記述したアセンブリ言語ソース・プログラムに対して生成するコードを，.sibss セクション注に割り当てます。

注 セクション名 .sibss，セクション・タイプ NOBITS，およびセクション属性 AW を持つ予約セクションです。

[詳細説明]

.sibss セクションは，初期値を持たず，ep と 16 ビットのディスプレースメントを用いて 1 命令で参照されるメモリ範囲（ep からプラス方向に最大 32 K バイト）のうち，使用された .tidata.byte，.tibss.byte，.tidata.word，.tibss.word，.tidata，.tibss，.sidata セクションのサイズ分だけ上位のアドレスに配置されるセクションです。

[使用例]

次のセクション定義疑似命令まで .sibss セクションとなる。

```
.sibss
.globl  _1, 4
.lcomm  _1, 4, 4
```

.sconst

.sconst セクションへ割り当てます。

[指定形式]

.sconst

[機能]

次のセクション定義疑似命令まで，または次のセクション定義疑似命令が現れなかった場合は，そのアセンブリ言語ソース・ファイルの終わりまでの間に記述したアセンブリ言語ソース・プログラムに対して生成するコードを，.sconst セクション[注]に割り当てます。

注 セクション名 .sconst，セクション・タイプ PROGBITS，およびセクション属性 A を持つ予約セクションです。

[詳細説明]

.sconst セクションは，定数データ用（読み出し専用）のセクションで，r0 と 16 ビットのディスプレースメントを用いて 1 命令で参照されるメモリ範囲（r0 からプラス方向に最大 32 K バイト）に配置されるセクションです。

[使用例]

次のセクション定義疑似命令まで .sconst セクションとなる。

```
    .sconst
    .align  4
    .globl  _p, 4
_p:
    .word   10
```

.const

.const セクションへ割り当てます。

[指定形式]

.const

[機能]

次のセクション定義疑似命令まで，または次のセクション定義疑似命令が現れなかった場合は，そのアセンブリ言語ソース・ファイルの終わりまでの間に記述したアセンブリ言語ソース・プログラムに対して生成するコードを，.const セクション注に割り当てます。

注 セクション名 .const，セクション・タイプ PROGBITS，およびセクション属性 A を持つ予約セクションです。

[詳細説明]

.const セクションは，定数データ用（読み出し専用）のセクションで，r0 と 2 命令によって構成される，32 ビットのディスプレースメントを用いて参照されるメモリ範囲に配置されるセクションです。

[使用例]

次のセクション定義疑似命令まで .const セクションとなる。

```
    .const
    .align  4
    .globl  _p, 4
_p:
    .word   10
```

.text

.text セクションへ割り当てます。

[指定形式]

.text

[機能]

次のセクション定義疑似命令まで，または次のセクション定義疑似命令が現れなかった場合は，そのアセンブリ言語ソース・ファイルの終わりまでの間に記述したアセンブリ言語ソース・プログラムに対して生成するコードを，.text セクション[注1]に割り当てます。[注2]

注1. セクション名 .text，セクション・タイプ PROGBITS，およびセクション属性 AX を持つ予約セクションです。

2. as850 では，1 つのアセンブリ言語ソース・ファイル中のアセンブリ言語ソース・プログラムの前には .text が 2 回指定されているものとみなされます（たとえば，セクション定義疑似命令を指定する前に“.word 1”を指定した場合，それは .text セクションに割り当てられます）。ただし，.text セクションを明示的に指定せずかつデフォルトで指定された .text セクションに対しラベルの定義，命令，ロケーション・カウンタ制御疑似命令，領域確保疑似命令のいずれも指定しなかった場合，as850 では，.text セクションが生成されません。

[使用例]

次のセクション定義疑似命令まで .text セクションとなる。

```
    .text
    .align  4
    .globl  _start
_start:
    mov     #_tp_TEXT, tp
```

.vdbstrtab

.vdbstrtab セクションへ割り当てます。

［指定形式］

.vdbstrtab

［機能］

次のセクション定義疑似命令まで，または次のセクション定義疑似命令が現れなかった場合は，そのアセンブリ言語ソース・ファイルの終わりまでの間に記述したアセンブリ言語ソース・プログラムに対して生成するコードを，.vdbstrtab セクション注に割り当てます。

注　セクション名 .vdbstrtab とセクション・タイプ STRTAB を持つ予約セクションです。

.vdebug

.vdebug セクションへ割り当てます。

［指定形式］

.vdebug

［機能］

次のセクション定義疑似命令まで，または次のセクション定義疑似命令が現れなかった場合は，そのアセンブリ言語ソース・ファイルの終わりまでの間に記述したアセンブリ言語ソース・プログラムに対して生成するコードを，.vdebug セクション注に割り当てます。

注　セクション名 .vdebug とセクション・タイプ PROGBITS を持つ予約セクションです。

.vline

.vline セクションへ割り当てます。

［指定形式］

.vline

［機能］

次のセクション定義疑似命令まで，または次のセクション定義疑似命令が現れなかった場合は，そのアセンブリ言語ソース・ファイルの終わりまでの間に記述したアセンブリ言語ソース・プログラムに対して生成するコードを，.vline セクション注に割り当てます。

注　セクション名 .vline とセクション・タイプ PROGBITS を持つ予約セクションです。

.section

指定した種類のセクションへ割り当てます。

［指定形式］

.section " セクション名 "[, セクションの種類]

［機能］

次のセクション定義疑似命令まで，または次のセクション定義疑似命令が現れなかった場合は，そのアセンブリ言語ソース・ファイルの終わりまでの間に記述したアセンブリ言語ソース・プログラムに対して生成するコードを，第1オペランドに指定したセクション名で，第2オペランドに指定した種類のセクションに割り付けます。

セクションの種類には，次に示す7種類があります。注

注 セクションの種類の指定には大文字を用いることもできます（たとえば，text のかわりに TEXT を用いることもできます）。

表 4―13 セクションの種類

種類	意味
data	data 属性セクション セクション・タイプ PROGBITS とセクション属性 AW を持つセクション
bss	bss 属性セクション セクション・タイプ NOBITS とセクション属性 AW を持つセクション
sdata	sdata 属性セクション セクション・タイプ PROGBITS とセクション属性 AWG を持つセクション
sbss	sbss 属性セクション セクション・タイプ NOBITS とセクション属性 AWG を持つセクション
const	const 属性セクション セクション・タイプ PROGBITS とセクション属性 A を持つセクション
text	text 属性セクション セクション・タイプ PROGBITS とセクション属性 AX を持つセクション
comment	comment 属性セクション セクション・タイプ PROGBITS を持ちセクション属性を持たないセクション

［使用例］

sec という名前で data 属性セクションを定義する。

```
    .section    "sec", data
    .align  4
    .globl  _p, 4
_p:
    .word   10
```

［注意事項］

- CA850 において，セクション名 .text, .data, .bss, .sdata, .sbss, .sconst, .const, .sidata, .sibss, .sedata, .sebss, .tidata, .tibss, .tidata.byte, .tibss.byte, .tidata.word, .tibss.word, .pro_epi_runtime, .version は予約されており，これらの予約セクションとセクションの種類の対応は次のようになっています。

表 4―14 セクションの種類

種類	予約セクション名
data	.tidata，.tidata.byte，.tidata.word，.data，.sedata，.sidata
bss	.tibss，.tibss.byte，.tibss.word，.bss，.sebss，.sibss
sdata	.sdata
sbss	.sbss
const	.sconst，.const
text	.text，.pro_epi_runtime
comment	.version

したがって，これらのセクション名を第 1 オペランドに指定した場合，第 2 オペランドは省略するか，それぞれの予約セクションに対して定められているセクションの種類を指定しなければなりません。なお，定められているセクションの種類と異なる種類を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
F3504: illegal section kind
```

- 上記の予約セクション名以外の名前を第 1 オペランドに指定し，第 2 オペランドを省略した場合，セクションの種類として text が指定されたものとみなされます。
- ある名前を持つセクションに対して複数の異なるセクションの種類を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
F3504: illegal section kind
```

- 第１オペランドにデバイス・ファイルで定義されている割り込み要求名を指定した場合，リンカが，そのセクションを該当するハンドラ・アドレスに自動割り付けします。したがって，割り込み要求名を指定したセクションに対しては，リンカで配置アドレスを指定することはできません。また，割り込みハンドラでないセクションに対し，割り込み要求名を指定しないようにしてください。

［割り込み要求名使用例］

リセット時に __start へジャンプするセクションを定義する

```
.section    "RESET", text
jr      _start
```

.previous

現在のセクション定義疑似命令を指定しているセクション定義疑似命令の前のセクション定義疑似命令の（再）指定をします。

［指定形式］

.previous

［機能］

現在のセクション定義疑似命令を指定しているセクション定義疑似命令の前のセクション定義疑似命令を（再）指定します。

たとえば，.data 疑似命令，.text 疑似命令，.previous 疑似命令の順に指定した場合，.previous 疑似命令の指定は .data 疑似命令の指定と等価になります。

［使用例］

.previous は .data と等価となる。

```
    .data
    .align  4
    .globl  _p, 4
_p:
    .word   10
    .text
lab:
    jbr     LL
    .previous
```

4. 2. 3　シンボル制御疑似命令

as850 では，シンボル制御疑似命令を用いることにより，シンボル・テーブル・エントリの生成，シンボルの定義，およびラベルの示すデータ・サイズの指定ができます。次に，この項において説明するシンボル制御疑似命令を示します。

表 4―15　シンボル制御疑似命令

疑似命令	意味
.set	シンボルの定義
.size	ラベルの示すデータ・サイズの指定
.frame	シンボル・テーブル・エントリの生成（FUNC タイプ）
.file	シンボル・テーブル・エントリの生成（FILE タイプ）
.ext_func	フラッシュ・テーブル・エントリの生成
.ext_ent_size	フラッシュ・テーブル・エントリのサイズ指定

なお，シンボル制御疑似命令において指定するサイズ（バイト数）は，2^{31} 未満にしてください。2^{31} 以上の値を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E3247: illegal size value
```

.set

シンボルの定義をします。

[指定形式]

.set シンボル名 , 値

[機能]

第 1 オペランドに指定したシンボル名と，第 2 オペランドに指定した値（整数値）を持つシンボルを定義します。1 つのアセンブリ言語ソース・ファイルにおいて，あるシンボルに対して複数回 .set 疑似命令を指定した場合，そのシンボルの参照は，その参照が現れた位置に従って，次の値を持ちます。

- ファイルの先頭からそのシンボルに対する最初の .set 疑似命令までの間に現れた場合
  そのシンボルに対する最後の .set 疑似命令で指定した値
- ある .set 疑似命令から次の .set 疑似命令までの間，または次の .set 疑似命令が現れなかった場合には，そのアセンブリ言語ソース・ファイルの終わりまでの間に現れた場合
  その .set 疑似命令で指定した値

[使用例]

シンボル sym1 の値を 0x10 として定義する。

```
.set    sym1, 0x10
```

[注意事項]

- 値の指定にラベル参照や，その時点において未定義なシンボル参照を用いることはできません。
  値の指定にラベル参照や，その時点において未定義なシンボル参照を用いた場合，次のメッセージが出力され，アセンブルが終了されます。

```
E3203: illegal expression (string)
```

- ラベル名，.macro 疑似命令で定義済みのマクロ名，およびマクロの仮パラメータと同名のシンボルを指定した場合，次のメッセージが出力され，アセンブルが終了されます。

```
E3212: symbol already define as string
```

.size

ラベルの示すデータ・サイズの指定をします。

[指定形式]

.size ラベル名 , サイズ

[機能]

第 2 オペランドに指定したサイズを，第 1 オペランドに指定したラベルの示すデータのサイズとして指定します。**注**

注 すでにサイズが設定されていた場合，その値は上書きされます。

[使用例]

label1 のサイズは 15 とする。

```
.size   label1, 15
```

[注意事項]

CA850 に含まれるリンカにおける -A オプションを用いる場合，sdata 属性セクションに割り当てるデータ（実際には gp オフセット参照されているラベル）の定義に対しては，本疑似命令，または .globl 疑似命令を用いてサイズを設定してください。**注**

注 上記の方法で設定しなかった場合，リンカにおける -A オプションにおいて正しい情報が得られません。

.frame

シンボル・テーブル・エントリを生成（FUNC タイプ）します。

［指定形式］

.frame　ラベル名 , サイズ

［機能］

オブジェクト・ファイル生成時，第 1 オペランドに指定したラベルに対するシンボル・テーブル・エントリの生成において，第 2 オペランドに指定したサイズとタイプ FUNC を持つシンボル・テーブル・エントリを生成します。**注**

注　C 言語レベルでのデバッグのために使用する疑似命令です。アセンブラ・デバッグ機能向けに記述する場合，サイズには 0 を指定してください。

.file

シンボル・テーブル・エントリを生成（FILE タイプ）します。

［指定形式］

.file　" ファイル名 "

［機能］

オブジェクト・ファイルの生成時，オペランドに指定したファイル名と，タイプ FILE を持つシンボル・テーブル・エントリ注を生成します。なお，入力したソース・ファイル中に本疑似命令が存在しない場合，「.file“入力ファイル名”」を指定したものとみなし，入力ファイル名とタイプ FILE を持つシンボル・テーブル・エントリを生成します。

注　バインディング・クラスは LOCAL とします。

.ext_func

フラッシュ・テーブル・エントリを生成します。

[指定形式]

.ext_func　　ラベル名 , ID 値

[機能]

オブジェクト・ファイルの生成時，オペランドに指定したラベル名と，ID 値を持つフラッシュ・テーブル・エントリを生成します。フラッシュ／外 ROM 再リンク機能を使用する際に指定します。

[詳細説明]

書き換え／取り換え不可能領域（以降，ブート領域）から，可能領域（以降，フラッシュ領域）へ分岐する場合，本疑似命令を指定することにより，フラッシュ領域側の指定したアドレスに分岐テーブルが作成され，テーブルを介しての二段分岐が行われます。

[注意事項]

- 本疑似命令は，ブート領域側の該当する分岐命令のあるソース・ファイル，およびフラッシュ領域側の該当するラベル定義のあるソース・ファイルに記述する必要があります。
- 同じラベル名で異なる ID 値を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

E3253: symbol "*identifier*" already defined as another id

- 同じ ID 値で異なるラベル名を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

E3252: id already defined as symbol "*identifier*"

- 上記不整合を防止するため，該当するラベル名すべてを 1 ファイルにまとめて記述し，.include 疑似命令を用いて，ブート／フラッシュ両側のソース・ファイルにインクルードすることを推奨します。
- ID 値は正数を指定してください。また，確保される分岐テーブルのサイズは ID 値の最大に依存します。ID 値は詰めて使用することを推奨します。

.ext_ent_size

フラッシュ・テーブル・エントリのサイズ指定をします。

[指定形式]

.ext_ent_size サイズ

[機能]

オブジェクト・ファイルの生成時，フラッシュ・テーブル・エントリ・サイズにオペランドに指定した値を設定します。フラッシュ／外部 ROM 再リンク機能を使用する際に指定します。

[詳細説明]

書き換え／取り換え不可能領域（以降，ブート領域）から，可能領域（以降，フラッシュ領域）へ分岐する場合，本疑似命令を指定することにより，フラッシュ領域側の指定したアドレスに分岐テーブルが作成され，テーブルを介しての二段分岐が行われます。このテーブルのエントリ・サイズはデフォルトで 4 バイトであり，jr 命令が生成され，分岐命令から 22 ビットの範囲に分岐可能です。このテーブルの分岐命令から 22 ビットの範囲を越えるアドレスに分岐する必要がある場合，本疑似命令でエントリ・サイズに V850 コアの場合には 10 を V850Ex コアの場合には 8 を指定することにより，32 ビットアドレス空間全体へ分岐可能です。

[注意事項]

- 本疑似命令は，ブート領域側の該当する分岐命令のあるソース・ファイル，およびフラッシュ領域側の該当するラベル定義のあるソース・ファイルに記述する必要があります。
- 本疑似命令にて指定するサイズは，ブート領域／フラッシュ領域を含め，全体で唯一の値となります。
- 異なるサイズを指定した場合，次のメッセージを出力し，アセンブルを続行します。
  なお，複数のリロケータブル・オブジェクト・ファイル間で異なるサイズが指定された場合，リンク時にエラーとなります。

```
W3021: .ext_ent_size already specified, ignored.
```

- 上記不整合を防止するため，該当するラベル名すべてを 1 ファイルにまとめて記述し，.include 疑似命令を用いて，ブート／フラッシュ両側のソース・ファイルにインクルードすることを推奨します。
- サイズは，4（デフォルト），8【V850E】，10【V850】のいずれかを指定してください。
- なお，共通オブジェクト（-cn オプション指定）作成時には，V850 と V850Ex の両方で動作する必要があるため，8【V850E】の指定はできません。

4. 2. 4　ロケーション・カウンタ制御疑似命令

as850 では，ロケーション・カウンタ制御疑似命令を用いることにより，ロケーション・カウンタ[注]の値を整列したり進めたりできます。次に，この項において説明するロケーション・カウンタ制御疑似命令を示します。

表 4―16　ロケーション・カウンタ制御疑似命令

疑似命令	意味
.align	ロケーション・カウンタの値の整列
.org	ロケーション・カウンタの値を進める

なお，sbss ／ bss 属性セクションにおいてロケーション・カウンタ制御疑似命令を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E3246: illegal section
```

注　セクションごとに存在し，そのファイル内の対応するセクションに対する最初のセクション定義疑似命令が現れたときに 0 に初期化されます。

.align

ロケーション・カウンタの値を整列します。

[指定形式]

.align 整列条件 [, フィリング値]

[機能]

前に指定されたセクション定義疑似命令によって指定される現在のセクションに対するロケーション・カウンタ値を，第 1 オペランドで指定した整列条件で整列します。なお，ロケーション・カウンタ値を整列したことによりホールが生じた場合，生じたホールを第 2 オペランドで指定したフィリング値，またはデフォルト値の 0 で埋めます。

たとえば，現在のロケーション・カウンタ値が 3 で .align 4 を指定した場合，ロケーション・カウンタ値を整列条件 4（ワード境界）で整列して 4 とし，生じた 1 バイト分のホールをデフォルト値の 0 で埋めます。

[使用例]

16 バイト整列する。

```
.align  16
```

[注意事項]

- 整列条件は 2 以上 2^{31} 未満の偶数にしてください。それ以外のものを指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E3200: illegal alignment value
```

- フィリング値を指定する場合，1 バイト分の値を指定します。1 バイト分以上の値を指定した場合，下位 1 バイト分が用いられます。
- sdata 属性セクションにおいて 4 より大きな整列条件を指定して本疑似命令を用いた場合，sdata ／ sbss 属性セクションに割り当てるデータのサイズを指定するための目安を表示する（ld850 の -A オプション）際に，正しい情報が得られなくなります。
- 本疑似命令は，そのセクションに対する指定したファイル内でのロケーション・カウンタ値を整列するだけであり，絶対アドレス**注 1** を整列するものでも，セクション内オフセット**注 2** を整列するものでもありません。

注 1. リンク後のオブジェクト・ファイルにおけるアドレス 0 からのオフセットです。

2. リンク後のオブジェクト・ファイルにおいてそのセクションが割り当てられたセクション（出力セクション）の先頭アドレスからのオフセットです。

.org

ロケーション・カウンタの値を進めます。

[指定形式]

.org 値

[機能]

前に指定されたセクション定義疑似命令によって指定される現在のセクションに対するロケーション・カウンタ値を，オペランドで指定した値（2^{31} 未満）まで進めます。なお，ロケーション・カウンタ値を進めることによりホールが生じた場合，生じたホールを 0 で埋めます

[使用例]

ロケーション・カウンタ値を 16 バイトに進める。

```
.org    16
```

[注意事項]

- 現在のロケーション・カウンタ値より小さな値を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E3244: illegal origin value value
```

- sdata 属性セクションにおいて本疑似命令を用いた場合，sdata ／ sbss 属性セクションに割り当てるデータのサイズを指定するための目安を表示する（ld850 の -A オプション）際に，正しい情報が得られなくなります。
- 本疑似命令は，そのセクションに対する指定したファイル内でのロケーション・カウンタ値を進めるだけであり，絶対アドレス**注 1** を指定するものでも，セクション内オフセット**注 2** を指定するものでもありません。

注 1. リンク後のオブジェクト・ファイルにおけるアドレス 0 からのオフセットです。

2. リンク後のオブジェクト・ファイルにおいてそのセクションが割り当てられたセクション（出力セクション）の先頭アドレスからのオフセットです。

4. 2. 5 領域確保疑似命令

as850 では，領域確保疑似命令を用いることにより，領域の確保，および確保した領域の初期化ができます。次に，この項において説明する領域確保疑似命令を示します。

表 4―17 領域確保疑似命令

疑似命令	意味
.byte	1 バイトごとの領域の確保
.hword	1 ハーフワードごとの領域の確保
.shword	1 ハーフワードごとの領域の確保【V850E】
.word	1 ワードごとの領域の確保
.float	浮動小数点数値の設定
.space	サイズ分の領域の確保
.str	文字列分の領域の確保
.lcomm	領域を確保したラベルの定義

なお，sbss ／ bss 属性セクションにおいて .lcomm 疑似命令以外の領域確保疑似命令を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E3246: illegal section
```

また，領域確保疑似命令において指定するサイズ（バイト数），および整列条件の値は，2^{31} 未満にしてください。2^{31} 以上の値を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E3247: illegal size value
，または
E3200: illegal alignment value
```

.byte

1バイトごとの領域を確保します。

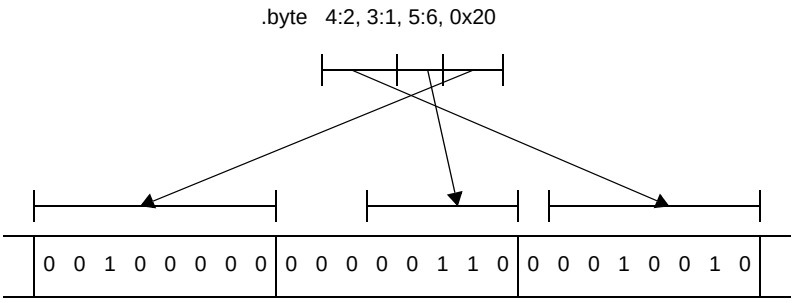
[指定形式]

- .byte 値 [, 値 , …]
- .byte ビット幅 : 値 [, ビット幅 : 値 , …]

[機能]

- .byte 疑似命令の最初の形式は，各オペランドに対して1バイト分の領域を確保し，確保した領域に指定した値の下位1バイトの値を格納します。
- .byte 疑似命令の2番目の形式は，指定したビット幅の領域を確保し，確保した領域に指定した値を格納します。
  - ビット幅は0から8の範囲で指定します。
  - バイト幅を越える場合，バイト幅でマスクします。
  - 最初に指定したビット幅を持つ値は，バイト領域の最下位のビットから割り付けます。その前のビット幅を持つ値を割り付けた領域に続けて領域を割り付けると，その領域がバイト境界を越えてしまう場合，そのビット幅を持つ値は，バイト境界から割り付けます（下図を参照）。
  - ホールが生じた場合，生じたホールを0で埋めます。

図 4―6 ビット幅を指定した割り付けの例



- 1つの .byte 疑似命令に，上記の2種類の指定を混在させることもできます（上図を参照）。

[使用例]

1バイト分確保し，1を格納する。

```
    .tidata.byte
    .align  4
    .globl  _p, 4
_p:
    .byte   1
```

.hword

1 ハーフワードごとの領域を確保します。

[指定形式]

- .hword 値 [, 値 , …]
- .hword ビット幅 : 値 [, ビット幅 : 値 , …]

[機能]

- .hword 疑似命令の最初の形式は，各オペランドに対して 1 ハーフワード（2 バイト）分の領域を確保し，確保した領域に指定した値の下位 1 ハーフワードの値を格納します。
- .hword 疑似命令の 2 番目の形式は，指定したビット幅の領域を確保し，確保した領域に指定した値を格納します。
  - ビット幅は 0 から 16 の範囲で指定します。
  - 値がハーフワード幅を越える場合，ハーフワード幅でマスクします。
  - 最初に宣言したビット幅を持つ値は，ハーフワード領域の最下位のビットから割り付けます。その前のビット幅を持つ値を割り付けた領域に続けて領域を割り付けるとその領域がハーフワード境界を越えてしまう場合，そのビット幅を持つ値は，ハーフワード境界から割り付けます。
  - ホールが生じた場合，生じたホールを 0 で埋めます。

- 1 つの .hword 疑似命令に，上記の 2 種類の指定を混在させることもできます。

[使用例]

1 ハーフワード分確保し，100 を格納する。

```
    .tidata
    .align  4
    .globl  _p, 4
_p:
    .hword  100
```

.shword

1 ハーフワードごとの領域を確保します。【V850E】

［指定形式］

- .shword 値 [, 値 , …]
- .shword ビット幅 : 値 [, ビット幅 : 値 , …]

［機能］

- shword 疑似命令の最初の形式は，各オペランドに対して 1 ハーフワード分の領域を確保し，確保した領域に指定した値を右に 1 ビット・シフトして格納します。
- .shword 疑似命令の 2 番目の形式は，指定したビット幅の領域を確保し，確保した領域に指定した値を右に 1 ビット・シフトして格納します。
  - ビット幅は 0 から 16 の範囲で指定します。
  - 値がハーフワード幅を越える場合，ハーフワード幅でマスクします。
  - 最初に宣言したビット幅を持つ値は，ハーフワード領域の最下位のビットから割り付けます。その前のビット幅を持つ値を割り付けた領域に続けて領域を割り付けるとその領域がハーフワード境界を越えてしまう場合，そのビット幅を持つ値は，ハーフワード境界から割り付けます。
  - ホールが生じた場合，生じたホールを 0 で埋めます。

- 1 つの .shword 疑似命令に，上記の 2 種類の指定を混在させることもできます。
- この疑似命令は，switch 命令用のテーブル作成に適しています。

［使用例］

1 ハーフワード分確保し，10 を右 1 ビット・シフトして格納する。

```
    .sdata
    .align  4
    .globl  _p, 4
_p:
    .shword 10
```

.word

1 ワードごとの領域を確保します。

[指定形式]

- .word 値 [, 値 , …]
- .word ビット幅 : 値 [, ビット幅 : 値 , …]

[機能]

- .word 疑似命令の最初の形式は，各オペランドに対して 1 ワード分の領域を確保し，確保した領域に指定した値を格納します。
- .word 疑似命令の 2 番目の形式は，指定したビット幅の領域を確保し，確保した領域に指定した値を格納します。
  - ビット幅は 0 から 32 の範囲で指定します。
  - 幅がワード幅を越える場合，ワード幅でマスクします。
  - 最初に宣言したビット幅を持つ値は，ワード領域の最下位のビットから割り付けます。その前のビット幅を持つ値を割り付けた領域に続けて領域を割り付けるとその領域がワード境界を越えてしまう場合，そのビット幅を持つ値は，ワード境界から割り付けます。
  - ホールが生じた場合，生じたホールを 0 で埋めます。

- 1 つの .word 疑似命令に，上記の 2 種類の指定を混在させることもできます。

[使用例]

1 ワード分確保し，0xa で埋める。

```
    .sidata
    .align  4
    .globl  _p, 4
_p:
    .word   0xa
```

.float

浮動小数点数値を設定します。

[指定形式]

.float 値 [, 値 , …]

[機能]

各オペランドに対して 1 ワード分の領域を確保し，確保した領域に指定した浮動小数点数値を格納します。注

注 整定数を指定した場合，1 ワード分の領域を確保し，確保した領域に指定した整定数の値を格納します。

[使用例]

1 ワード分確保し，1.2345 を格納する。

```
    .sidata
    .align  4
    .globl  _p, 4
_p:
    .float  1.2345
```

.space

サイズ分の領域を確保します。

[指定形式]

.space サイズ [, フィリング値]

[機能]

- 第 1 オペランドで指定したサイズ分の領域を確保し，確保した領域を第 2 オペランドで指定したフィリング値（デフォルト値は 0）で埋めます。
- フィリング値を指定する場合，1 バイト分の値を指定します。
- 1 バイト分以上の値を指定した場合，下位 1 バイト分を用います。

[使用例]

4 バイトを 0 で埋める。

```
    .sidata
    .globl  _p, 4
_p:
    .space  4
```

.str

文字列分の領域を確保します。

［指定形式］

.str " 文字列定数 "[, " 文字列定数 ", …]

［機能］

各オペランドに対して，指定された文字列分の領域を確保し，確保した領域に指定した文字列を格納します。注

注 C 言語の場合と異なり，文字列の最後にデフォルトで "\0" を入れることはありません。

［使用例］

文字列 "hello" 分の領域を確保し，格納する。

```
.str    "hello"
```

.lcomm

領域を確保したラベルを定義します。

[指定形式]

.lcomm ラベル名 , サイズ , 整列条件

[機能]

前に指定されたセクション定義疑似命令によって指定される現在のセクションに対するロケーション・カウンタ値を第 3 オペランドで指定した整列条件で整列し，第 2 オペランドで指定したサイズの領域を確保し，その先頭のアドレスに対して第 1 オペランドで指定したラベル名を持つローカルなラベル[注]を定義します。

注 ローカル・シンボル（LOCAL のバインディング・クラスを持つシンボル）です。

[使用例]

_stack ラベルのサイズは 0x100 とし，4 バイト整列する。

```
.bss
.lcomm  _stack, 0x100, 4
```

[注意事項]

- 前に指定されたセクション定義疑似命令によって指定されている現在のセクションは，sbss ／ bss 属性セクションでなければなりません。これら以外のセクションにおいて本疑似命令を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E3246: illegal section
```

- sbss 属性セクションにおいて 4 より大きな整列条件を指定して本疑似命令を用いた場合，sdata ／ sbss 属性セクションに割り当てるデータのサイズを指定するための目安を表示する（ld850 の -A オプション）際に，正しい情報が得られなくなります。

4. 2. 6　プログラム・リンケージ疑似命令

as850 では，プログラム・リンケージ疑似命令を用いることにより，指定したサイズ，および整列条件を持つ未定義外部ラベル[注 1] や外部ラベル[注 2] を宣言することができます。次に，この項において説明するプログラム・リンケージ疑似命令を示します。

表 4―18　プログラム・リンケージ疑似命令

疑似命令	意味
.globl	外部ラベルの宣言
.extern	外部ラベルの宣言
.comm	未定義外部ラベルの宣言

なお，プログラム・リンケージ疑似命令において指定するサイズ（バイト数），および整列条件は，2^{31} 未満にしてください。2^{31} 以上の値を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E3247: illegal size value
，または
E3200: illegal alignment value
```

注 1. 未定義外部シンボル（GLOBAL のバインディング・クラスと GPCOMMON，または COMMON のセクション・ヘッダ・テーブル・インデクスを持つシンボル）です。

2. 外部シンボル（GLOBAL のバインディング・クラスを持つシンボル）です。

.globl

外部ラベルを宣言します。

[指定形式]

.globl ラベル名 [, サイズ]

[機能]

第 1 オペランドで指定したラベル名と同名のラベルを外部ラベル[注]として宣言します。なお，第 2 オペランドを指定した場合，指定した値をそのラベルの示すデータのサイズとして指定します。本疑似命令と .extern 疑似命令は外部ラベルを宣言するという機能において変わりませんが，指定したファイル内に定義を持つラベルを外部ラベルとして宣言する場合，本疑似命令を用い，指定したファイル内に定義を持たないラベルを外部ラベルとして宣言する場合，.extern 疑似命令を用いるようにしてください。

注 外部シンボル（GLOBAL のバインディング・クラスを持つシンボル）です。

[使用例]

外部ラベル _func を宣言する（_func はファイル内に定義を持つ）。

```
.globl  _func
```

[注意事項]

- この宣言により，第 1 オペランドで指定したラベル名と同名のラベルを定義した場合，そのラベルは他のアセンブリ言語ソース・ファイルから参照できるようになります。
- sdata ／ sbss 属性セクションに割り当てるデータのサイズを指定するための目安を表示する（ld850 の -A オプション）場合，sdata 属性セクションに割り当てるデータ（実際には gp オフセット参照されているラベル）の定義に対しては，本疑似命令，または .size 疑似命令を用いてそのサイズを設定してください。[注]

注 上記の方法で設定しなかった場合，正しい情報が得られません。

.extern

外部ラベルを宣言します。

[指定形式]

.extern ラベル名 [, サイズ]

[機能]

第 1 オペランドで指定したラベル名と同名のラベルを，外部ラベル[注]として宣言します。なお，第 2 オペランドを指定した場合，指定した値をそのラベルの示すデータのサイズとして指定します。本疑似命令と .globl 疑似命令は外部ラベルを宣言するという機能において変わりませんが，指定したファイル内に定義を持たないラベルを外部ラベルとして宣言する場合，本疑似命令を用い，指定したファイル内に定義を持つラベルを外部ラベルとして宣言する場合，.globl 疑似命令を用いるようにしてください。

注 外部シンボル（GLOBAL のバインディング・クラスを持つシンボル）です。

[使用例]

外部ラベル _main を宣言する（_main はファイル内に定義を持たない）。

```
.extern _main
```

[注意事項]

- as850 では，ラベルは指定したファイル内に定義を持たないことにより，デフォルトで外部ラベルとして宣言されます。このため，第 1 オペランドで指定したラベル名と同名のラベルが，指定したファイル内に定義を持たない場合，本疑似命令は，実質上，そのラベルの示すデータのサイズを指定するだけの意味になります。
- as850 では，指定したファイル内に定義を持たないデータの gp オフセット参照に対し，「16 ビットのディスプレースメントを用いた参照を行う機械語命令を生成するか」，「（複数の機械語命令によって構成される）32 ビットのディスプレースメントを用いた参照を行う機械語命令列を生成するか」ということを，そのデータのサイズに基づいて判定するため，指定したファイル内に定義を持たず gp オフセット参照されているラベルに対しては，本疑似命令を用いてそのサイズを指定してください。

.comm

未定義外部ラベルを宣言します。

[指定形式]

.comm ラベル名 , サイズ , 整列条件

[機能]

第 1 オペランドで指定したラベル名，第 2 オペランドで指定したサイズ，および第 3 オペランドで指定した整列条件を持つ未定義外部ラベルを宣言します。

CA850 に含まれるリンカ（ld850）は，未定義外部シンボル（GLOBAL のバインディング・クラスと GPCOMMON，または COMMON のセクション・ヘッダ・テーブル・インデクスを持つシンボル）に対する定義が存在しなかった場合，GPCOMMON のセクション・ヘッダ・テーブル・インデクスを持つ未定義外部シンボルに対しては .sbss セクションに，COMMON のセクション・ヘッダ・テーブル・インデクスを持つ未定義外部シンボルに対しては .bss セクションに，指定された整列条件で整列され指定されたサイズを持つ領域を割り付けます（異なるサイズを持つ未定義外部シンボルが複数存在した場合，ld850 は大きい方のサイズを用います）。定義が存在した場合，その定義の方を優先します。

- 起動時に -G*num* オプションを指定した場合
  - 指定したサイズが 1 以上 *num* バイト以下の場合
    オブジェクト・ファイル生成時のそのラベルに対するシンボル・テーブル・エントリの生成において，GPCOMMON のセクション・ヘッダ・テーブル・インデクスを持つシンボル・テーブル・エントリを生成します。
  - 指定したサイズが 0 であるか *num* バイトより大きい場合
    オブジェクト・ファイル生成時のそのラベルに対するシンボル・テーブル・エントリの生成において，COMMON セクション・ヘッダ・テーブル・インデクスを持つシンボル・テーブル・エントリを生成します。

- 起動時に -G*num* オプションを指定しなかった場合
  オブジェクト・ファイル生成時のそのラベルに対するシンボル・テーブル・エントリの生成において，GPCOMMON のセクション・ヘッダ・テーブル・インデクスを持つシンボル・テーブル・エントリを生成します。

[使用例]

サイズ 4 の未定義外部ラベルを整列条件 4 で宣言する。

```
.sbss
.comm   _p, 4, 4
```

［注意事項］

- 第 1 オペランドで指定したラベル名と同名のラベルが，本疑似命令と同じファイル内において通常のラベル定義により定義されている場合

  - そのラベルが本疑似命令により GPCOMMON のシンボル・テーブル・エントリ・インデクスを持つよう宣言され data 属性セクションにおいて通常のラベル定義により定義されている場合，または本疑似命令により COMMON のシンボル・テーブル・エントリ・インデクスを持つように宣言され sdata 属性セクションにおいて通常のラベル定義により定義されている場合

```
    .comm   lab1, 4, 4  --“-G”なしでアセンブルすると GPCOMMON
      :
    .data
lab1:                   --.data セクションで通常のラベル定義
```

次のメッセージが出力され，アセンブルが中止されます。

```
E3213: label identifier redefined
```

  - 上記以外の場合

通常のラベル定義により定義されたラベルが外部ラベルとみなされ，本疑似命令の指定が無視されます。オブジェクト・ファイル生成時のそのラベルに対するシンボル・テーブル・エントリの生成において，GLOBAL のバインディング・クラスを持つシンボル・テーブル・エントリが生成されます。

```
    .comm   lab1, 4, 4  --“-G”なしでアセンブルすると GPCOMMON
      :
    .sdata
lab1 :                  --.sdata セクションで通常のラベル定義
```

- 第 1 オペランドで指定したラベル名と同名のラベルが，本疑似命令と同じファイル内において .lcomm 疑似命令により定義されている場合

  - .lcomm 疑似命令で指定されたサイズ，または整列条件と，本疑似命令において指定されたサイズ，または整列条件が異なる場合

```
.comm   lab1, 4, 4
  :
.sbss
.lcomm  lab1, 4, 2      -- 整列条件が異なる
```

次のメッセージが出力され，アセンブルが中止されます。

```
E3213: label identifier redefined
```

- そのラベルが本疑似命令により GPCOMMON のセクション・ヘッダ・テーブル・インデクスを持つよう宣言され bss 属性セクションにおいて .lcomm 疑似命令により定義されている場合，または本疑似命令により COMMON のセクション・ヘッダ・テーブル・インデクスを持つよう宣言され sbss 属性セクションにおいて .lcomm 疑似命令により定義されている場合

```
.comm   lab1, 4, 4      --“-G”なしでアセンブルすると GPCOMMON
  :
.bss
.lcomm  lab1, 4, 4      --.bss セクションで定義
```

次のメッセージが出力され，アセンブルが中止されます。

```
E3213: label identifier redefined
```

- 上記以外の場合

.lcomm により定義されたラベルが外部ラベルとみなされ，本疑似命令の指定が無視されます。オブジェクト・ファイル生成時のそのラベルに対するシンボル・テーブル・エントリの生成において，GLOBAL のバインディング・クラスを持つシンボル・テーブル・エントリが生成されます。

```
.comm   lab1, 4, 4      --“-G”なしでアセンブルすると GPCOMMON
  :
.sbss
.lcomm  lab1, 4, 4      --.sbss セクションで定義
```

- 第 1 オペランドで指定したラベル名と同名のラベルが，本疑似命令と同じファイル内において本疑似命令により（再）定義されている場合

- サイズ，または境界条件において異なっている場合

```
.comm   lab1, 4, 4
  :
.comm   lab1, 2, 4      -- サイズが異なる
```

次のメッセージが出力され，アセンブルが中止されます。

```
E3213: label identifier redefined
```

- サイズ，および境界条件が同じ場合

.comm 疑似命令が 1 回だけ指定されたものと見なされます。

4. 2. 7　アセンブラ制御疑似命令

as850 では，アセンブラ制御疑似命令を用いることにより，アセンブラが行う処理を制御できます。次に，この項において説明するアセンブラ制御疑似命令を示します。

表 4―19　アセンブラ制御疑似命令

疑似命令	意味
.option	オプションによるアセンブラ制御

.option

オプションによるアセンブラ制御を行います。

［指定形式］

.option オプション

［機能］

オペランドに指定したオプションに従ってアセンブラを制御します。指定することのできるオプションを，次に示します。**注**

注　オプションの指定には大文字を用いることもできます（たとえば，nomacro のかわりに NOMACRO を用いることもできます）。

- asm

　本疑似命令以降の構文エラーに対し，c オプションの指定を解除します。

- az_info_j

　本疑似命令の直後の命令のアドレスが，AZ850 用のアドレス情報セクション（セクション名：az_info_j）に出力されます。このオプションは，関数呼び出しを行っている命令のアドレスの情報収集を指定するオプションです。

- az_info_r

　本疑似命令の直後の命令のアドレスが，AZ850 用のアドレス情報セクション（セクション名：az_info_r）に出力されます。このオプションは，関数からの復帰を行っている命令のアドレスの情報収集を指定するオプションです。

- az_info_ri

　本疑似命令の直後の命令のアドレスが，AZ850 用のアドレス情報セクション（セクション名：az_info_ri）に出力されます。このオプションは，割り込み関数からの復帰を行っている命令のアドレスの情報収集を指定するオプションです。

- c *linenum* ["*filename*"]

　本疑似命令以降の構文エラーに対し，エラー・メッセージの行数（*linenum*），ファイル名（*filename*）を，指定されたものに置き換えて出力します。アセンブリ言語ソース・ファイル内の二度目以降の本疑似命令の "*filename*" は省略可能です。省略した場合，直前に本疑似命令に指定したファイル名が指定されたものとして処理をします。この場合，直前の本疑似命令との間の asm オプションの有無は問いません。
　アセンブリ言語ソース・ファイル内の一度目の "*filename*" を省略した場合，次のメッセージを出力し，アセンブルを中止します。

E3249: illegal syntax

- callt

コンパイラ予約の疑似命令です。

注意 コンパイラが出力したアセンブリ言語ソース・ファイル中に存在する場合は，削除しないでください。削除した場合，プロローグ／エピローグ・ランタイムのリンクを確認する機能が動作しません。

- cpu *devicename*

devicename で指定されたターゲット・デバイスのデバイス・ファイルを読み込みます。デバイス・ファイルを読み込むためのデバイス名指定は，as850 起動時に，-cpu オプションによっても指定できます。-cpu オプション，-cn*xxx* オプションによる指定も本疑似命令による指定もない場合，次のエラー・メッセージが出力され，処理が中止されます。

```
F3522: unknown cpu type
```

-cpu オプション，疑似命令の両方で指定した場合，警告メッセージが出力され，オプション指定を優先します。ただし，オプション指定，または疑似命令で，複数の異なるデバイス名を指定した場合，次のエラー・メッセージが出力され，処理が中止されます。

```
F3523: duplicated cpu type
```

例 使用するデバイスとして V850ES/SA2 を指定する。

```
.option cpu     3201
```

なお，使用するデバイス・ファイルは，“ as850 のあるディレクトリ\ .. \ .. \ .. \ dev”ディレクトリ（C コンパイラ・パッケージのインストール・ディレクトリ\ .. \ .. \ dev）に置くか，またはデバイス・ファイルのあるディレクトリを，as850 の“-F オプション”で指定するようにしてください。
devicename で指定するデバイス・ファイルのファイル名に空白文字が含まれる場合，次のエラー・メッセージが出力され，処理が中止されます。

```
E3250: illegal syntax string
```

- data *extern_symbol*

シンボル名 *extern_symbol* の外部データが，ca850 や as850 の -G オプションで指定されたサイズ値に関わらず，data 属性，または bss 属性セクションに割り当てられているとみなされ，そのデータを参照する命令に対して命令展開が行われます。この形式は，# pragma section やセクション・ファイルで“data”を指定した変数を，アセンブリ言語ソース・ファイルで外部参照する場合に利用します。

例 _d はオプションに関係なく .data セクションとなり，参照時に命令展開される。

```
.option data    _d
.text
mov     $_d, r11
```

- ep_label

それ以降の命令に対し，%label によるラベル参照がすべて ep オフセット参照として扱われます。

- macro

それ以降の命令に対し，nomacro オプションの指定が解除されます。

- mask_reg

as850 が生成するリロケータブルなオブジェクト・ファイル中に，マスク・レジスタ機能を使用していることを示す情報が埋めこまれます。このオプションは，たとえば，マスク・レジスタ機能をサポートしていない旧バージョンの C コンパイラ（CA850 Ver.1.00）が出力したアセンブリ言語ソース・ファイルを，マスク・レジスタ機能指定として利用する場合に有効です。このオプションを指定することにより，マスク・レジスタ機能を使用しているとみなされ，マスク・レジスタ指定でコンパイルして作成したオブジェクトのリンク時にエラーとなりません。

注意 マスク・レジスタ機能では，マスク・レジスタとして r20，および r21 を C コンパイラが使用します。これらのレジスタに設定したマスク値を，アセンブリ言語ソース・プログラムにより変更しないようにしてください。

- new_fcall

as850 が生成するリロケータブルなオブジェクト・ファイル中に，新呼び出し仕様であることを示す情報が埋めこまれます。このオプションは，たとえば，呼び出し仕様の異なる旧版の C コンパイラ（CA850 Ver.1.xx）が出力したアセンブリ言語ソース・ファイルを，現版の C コンパイラにより作成されたオブジェクトとともに利用する場合に有効です。このオプションを指定することにより，新呼び出し仕様を満たしているとみなされ，C コンパイラのデフォルトの新呼び出し仕様で作成されたオブジェクトとのリンク時に，エラーとなりません。

- no_ep_label

それ以降の命令に対し，ep_label オプションの指定が解除されます。

- nomacro

それ以降の setf*cnd*，sasf*cnd* 【V850E】，cmov*cnd*【V850E】，adf*cnd* 【V850E2】，sbf*cnd*【V850E2】，j*cnd*，jmp，jarl，jr 命令を除く命令に対し，命令展開が行われません。

- nooptimize

それ以降の命令に対し，命令並べ換え最適化が行われません。

- novolatile

それ以降の命令に対し，nooptimize ／ volatile オプションの指定が解除されます。

- nowarning

それ以降の命令に対し，警告メッセージが出力されません。

- optimize

novolatile オプションと同様です。

- reg_mode *tnum pnum*

as850 が生成するリロケータブルなオブジェクト・ファイル中に，レジスタ・モード情報セクションが埋め込まれます。レジスタ・モード情報セクションとは，コンパイラが使用する作業用レジスタとレジスタ変数用レジスタの本数情報を保持するものです。この命令により，作業用レジスタとレジスタ変数用レジスタの本数が，*tnum*，*pnum* 本として設定されます。なお，22 レジスタ・モードを使用する場合，*tnum*，*pnum* は 5 本ずつとなり，26 レジスタ・モードを使用する場合，*tnum*，*pnum* は 7 本ずつとなります。

例 レジスタ・モード 22 を用いる。

```
.option reg_mode    5   5
```

- sdata *extern_symbol*

シンボル名 extern_symbol の外部データが，ca850 や as850 の -G オプションで指定されたサイズ値に関わらず，sdata 属性，または sbss 属性セクションに割り当てられているとみなされ，そのデータを参照する命令に対して命令展開が行われません。この形式は，# pragma section やセクション・ファイルで“sdata”が指定された変数を，アセンブリ言語ソース・ファイルで外部参照する場合に利用します。

例 _d はオプションに関係なく .sdata セクションとなり，参照時に命令展開されない。

```
.option sdata   _d
.text
mov     $_d, r11
```

- volatile

nooptimize オプションと同様です。

- warning

それ以降の命令に対し，警告メッセージが出力されます。

4. 2. 8　ファイル入力制御疑似命令

as850 では，ファイル入力制御疑似命令を用いることにより，アセンブリ言語ソース・ファイル，またはバイナリ・ファイルを，指定した位置に取り込むことができます。次に，この項において説明するファイル入力制御疑似命令を示します。

表 4―20　ファイル入力制御疑似命令

疑似命令	意味
.include	アセンブリ言語ソース・ファイルの入力
.binclude	バイナリ・ファイルの入力

.include

アセンブリ言語ソース・ファイルの入力をします。

[指定形式]

.include " ファイル名 "

[機能]

オペランドに指定したファイルの内容を，本疑似命令の置かれている位置に置かれているものとして扱います。指定したファイルは，本疑似命令を含むソース・ファイルの置かれているディレクトリで検索されます。また，ファイル名には，ソース・ファイルの置かれているディレクトリからの相対パスによる記述もできます。アセンブラのオプション（-I）でディレクトリを指定した場合，指定したディレクトリが先に検索されます。ソース・ファイルの置かれているディレクトリにファイルが存在しない場合，C 言語ソース・ファイルの置かれているディレクトリ（.file 疑似命令より導出），カレント・ディレクトリを検索します。

[使用例]

aa.s ファイルをインクルードする。

```
.include    "aa.s"
```

[注意事項]

- 指定するファイル名は，" で囲んでください。
- 存在しないファイルを指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
F3503: can not open file file
```

- .include 文が 9 回以上ネストして用いられた場合，次のメッセージが出力され，アセンブルが中止されます。

```
F3517: include nest over
```

.binclude

バイナリ・ファイルの入力をします。

[指定形式]

.binclude " ファイル名 "

[機能]

オペランドに指定したバイナリ・ファイルの内容を，本疑似命令の置かれている位置に置かれたソース・プログラムのアセンブル結果であるとみなして扱います。指定したファイルは，本疑似命令を含むソース・ファイルの置かれているディレクトリで検索されます。また，ファイル名には，ソース・ファイルの置かれているディレクトリからの相対パスによる記述も可能です。アセンブラのオプション（-I）でディレクトリを指定した場合，指定したディレクトリが先に検索されます。ソース・ファイルの置かれているディレクトリにファイルが存在しない場合，C 言語ソース・ファイルの置かれているディレクトリ（.file 疑似命令より導出），カレント・ディレクトリを検索します。

[使用例]

aa.bin ファイルをインクルードする。

```
.binclude       "aa.bin"
```

[注意事項]

- 本疑似命令は，バイナリ・ファイルの内容全体を扱います。リロケータブル・ファイルを指定した場合には，ELF フォーマットで構成されたファイル全体を扱います。.text セクション等の内容のみを扱うということではありません。
- 指定するファイル名は，" " で囲んでください。
- 存在しないファイルを指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
F3503: can not open file file
```

4. 2. 9　繰り返しアセンブル疑似命令

as850 では，繰り返しアセンブル疑似命令とそれに対応する .endm 疑似命令とで囲まれた文の並び（ブロック）を，繰り返しアセンブル疑似命令の置かれている位置で，繰り返してアセンブルができます。次に，この項において説明する繰り返しアセンブル疑似命令を示します。

表 4―21　繰り返しアセンブル疑似命令

疑似命令	意味
.repeat	回数の指定による繰り返し
.irepeat	パラメータの指定による繰り返し

.repeat

指定された回数だけ繰り返します。

［指定形式］

.repeat 絶対値式

［機能］

本疑似命令と本疑似命令に対応する .endm 疑似命令で囲まれている文の並び（ブロック）を，第 1 オペランドで指定した絶対値式で与えられた回数だけ繰り返してアセンブルします。

［使用例］

次のように展開されます。

【展開前】

```
.repeat 2
    nop
.endm
```

【展開後】

```
nop
nop
```

［注意事項］

- .repeat と .endm は対応させてください。対応する .endm が存在しない場合，次のメッセージが出力され，アセンブルが中止されます。

```
F3513: unexpected EOF in .repeat/.irepeat
```

- 値は，32 ビットの符号付き整数として評価されます。
- 文の並び（ブロック）がない場合，何も行いません。
- 式の評価結果が負になった場合，次のメッセージが出力され，アセンブルが中止されます。

```
E3225: illegal operand (must be evaluated positive or zero)
```

.irepeat

指定されたパラメータ分だけ繰り返します。

[指定形式]

.irepeat 仮パラメータ 実パラメータ [, 実パラメータ , …]

[機能]

本疑似命令と，本疑似命令に対応する .endm 疑似命令で囲まれる文の並び（ブロック）を，そのブロック内に現れた第 1 オペランドで指定した仮パラメータを第 2 オペランド以降に指定した実パラメータに置き換え，繰り返しアセンブルします。第 2 オペランド以降に指定した実パラメータがすべて置き換えに用いられた場合，繰り返しを中止します。

[使用例]

次のように展開されます。

【展開前】

```
.irepeat    x   a, b, c, d
    .word   x
.endm
```

【展開後】

```
.word   a
.word   b
.word   c
.word   d
```

[注意事項]

- .irepeat と .endm は対応させてください。対応する .endm が存在しない場合，次のメッセージが出力され，アセンブルが中止されます。

```
F3513: unexpected EOF in .repeat/.irepeat
```

- 33 個以上の実パラメータを指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
F3514: paramater table overflowt
```

- 仮パラメータと実パラメータに同じパラメータ名を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
F3238: illegal operand (.irepeat parameter)
```

- 仮パラメータと実パラメータにラベルや他の疑似命令で定義済みのパラメータ名を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
F3238: illegal operand (.irepeat parameter)
```

4. 2. 10　条件アセンブル疑似命令

as850 では，条件アセンブル疑似命令を用いることにより，条件式の評価結果に従って，アセンブルを行う範囲が制御できます。次に，この項において説明する条件アセンブル疑似命令を示します。

表 4―22　条件アセンブル疑似命令

疑似命令	意味
.if	絶対値式による制御（真のときアセンブル）
.ifn	絶対値式による制御（偽のときアセンブル）
.ifdef	シンボルによる制御（定義されているときアセンブル）
.ifndef	シンボルによる制御（定義されていないときアセンブル）
.else	絶対値式／シンボルによる制御
.elseif	絶対値式による制御（真のときアセンブル）
.elseifn	絶対値式による制御（偽のときアセンブル）
.endif	制御範囲の終わり

なお，条件アセンブル疑似命令が 17 回以上ネストして用いられた場合，次のメッセージが出力され，アセンブルが中止されます。

```
F3512: .if, .ifn, etc. too deeply nested
```

.if

絶対値式による制御（真のときアセンブル）をします。

［指定形式］

.if 絶対値式

［機能］

- オペランドで指定した絶対値式が真（≠ 0）に評価された場合

（a）本疑似命令と本疑似命令に対応する .else 疑似命令，.elseif 疑似命令，または .elseifn 疑似命令が存在する場合は，本疑似命令とその疑似命令とで囲まれるブロックをアセンブルします。

（b）それらの疑似命令が存在しない場合は，本疑似命令と本疑似命令に対応する .endif 疑似命令とで囲まれるブロックをアセンブルします。

- 偽（= 0）に評価された場合

本疑似命令に対応する .else 疑似命令，.elseif 疑似命令，.elseifn 疑似命令，または .endif 疑似命令までスキップします。

［使用例］

次のように展開されます。

【展開前】

```
.if     10
    .word   10
.endif
.if     10 < 20
    .word   20
.endif
.set    expr, 30
.if     expr
    .word   expr
.endif
```

【展開後】

```
.word   10
.word   20
.word   30
```

［注意事項］

- オペランドに未定義なシンボルを指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E3202: illegal expression
```

- 対応する疑似命令が存在しない場合，次のメッセージが出力され，アセンブルが中止されます。

```
F3511: .endif unmatched
```

.ifn

絶対値式による制御（偽のときアセンブル）をします。

［指定形式］

.ifn　絶対値式

［機能］

- オペランドで指定した絶対値式が真（≠ 0）に評価された場合

　本疑似命令に対応する .else 疑似命令，.elseif 疑似命令，.elseifn 疑似命令，または .endif 疑似命令までスキップします。

- 偽（= 0）に評価された場合

（a）本疑似命令と本疑似命令に対応する .else 疑似命令，.elseif 疑似命令，または .elseifn 疑似命令が存在する場合は，本疑似命令とその疑似命令とで囲まれるブロックをアセンブルします。

（b）それらの疑似命令が存在しない場合は，本疑似命令と本疑似命令に対応する .endif 疑似命令とで囲まれるブロックをアセンブルします。

［使用例］

次のように展開されます。

【展開前】

```
.ifn    0
    .word   10
.endif
.ifn    10 > 20
    .word   20
.endif
.set    expr, 0
.ifn    expr
    .word   expr
.endif
```

【展開後】

```
.word   10
.word   20
.word   0
```

［注意事項］

- 対応する疑似命令が存在しない場合，次のメッセージが出力され，アセンブルが中止されます。

```
F3511: .endif unmatched
```

.ifdef

シンボルによる制御（定義されているときアセンブル）をします。

［指定形式］

.ifdef 名前

［機能］

- オオペランドで指定した名前が定義されている場合

（a）本疑似命令と本疑似命令に対応する .else 疑似命令，.elseif 疑似命令，または .elseifn 疑似命令が存在する場合は，本疑似命令とその疑似命令とで囲まれるブロックをアセンブルします。

（b）それらの疑似命令が存在しない場合は，本疑似命令とその疑似命令に対応する .endif 疑似命令とで囲まれるブロックをアセンブルします。

- 指定した名前が定義されていない場合

本疑似命令に対応する .else 疑似命令，.elseif 疑似命令，.elseifn 疑似命令，または .endif 疑似命令までスキップします。

［使用例］

次のように展開されます。

【展開前】

```
define_symbol:
    .ifdef  define_symbol
        .word   10
    .endif
    .ifdef  undef_symbol
        .word   20
    .else
        .ifdef  define_symbol
            .str    "x"
        .endif
    .endif
    .set    expr, 20
    .ifdef  expr
        .word   expr
    .endif
```

【展開後】

```
.word   10
.str    "x"
.word   20
```

［注意事項］

- 名前には，シンボル，ラベル，マクロ名を指定できますが，予約語は指定できません。予約語を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E3220: illegal operand (identifier is reserved word)
```

- 対応する疑似命令が存在しない場合，次のメッセージが出力され，アセンブルが中止されます。

```
F3511: .endif unmatched
```

- .local 疑似命令によりアセンブラが生成したローカル・シンボル名は未定義として扱われます。

.ifndef

シンボルによる制御（定義されていないときアセンブル）をします。

［指定形式］

.ifndef 名前

［機能］

- オペランドで指定した名前が定義されている場合

本疑似命令に対応する .else 疑似命令，.elseif 疑似命令，.elseifn 疑似命令，または .endif 疑似命令までスキップします。

- 指定した名前が定義されていない場合

（a）本疑似命令と本疑似命令に対応する .else 疑似命令，.elseif 疑似命令，または .elseifn 疑似命令が存在する場合は，本疑似命令とその疑似命令とで囲まれるブロックをアセンブルします。

（b）それらの疑似命令が存在しない場合は，本疑似命令と本疑似命令に対応する .endif 疑似命令とで囲まれるブロックをアセンブルします。

［使用例］

次のように展開されます。

【展開前】

```
define_symbol:
    .ifndef define_symbol
        .word   10
    .else
        .str    "a"
    .endif
    .ifndef undef_symbol
        .word   20
    .else
        .ifndef define_symbol
            .str    "x"
        .endif
    .endif
    .set    expr, 20
    .ifndef expr
        .word   expr
    .endif
```

【展開後】

```
.str    "a"
.word   20
```

［注意事項］

- 名前には，シンボル，ラベル，マクロ名を指定できますが，予約語は指定できません。予約語を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E3220: illegal operand (identifier is reserved word)
```

- 対応する疑似命令が存在しない場合，次のメッセージが出力され，アセンブルが中止されます。

```
F3511: .endif unmatched
```

- .local 疑似命令によりアセンブラが生成したローカル・シンボル名は未定義として扱われます。

.else

絶対値式／シンボルによる制御をします。

［指定形式］

.else

［機能］

.if 疑似命令，.elseif 疑似命令，または .ifdef 疑似命令において絶対値式が偽（= 0）に評価された場合，あるいは本疑似命令に対応する .ifn 疑似命令，.elseifn 疑似命令，または .ifndef 疑似命令において絶対値式が真（≠ 0）に評価された場合，本疑似命令と本疑似命令に対応する .endif 疑似命令とで囲まれる文の並び（ブロック）をアセンブルします。

［使用例］

次のように展開されます。

【展開前】

```
.if     0
    .word   10
.else
    .str    "a"
.endif
.if     10 > 20
    .word   20
.else
    .str    "b"
.endif
.set    expr, 0
.if     expr
    .word   expr
.else
    .str    "c"
.endif
```

【展開後】

```
.str    "a"
.str    "b"
.str    "c"
```

［注意事項］

- 本疑似命令に対応する .if 疑似命令，.ifn 疑似命令，.elseif 疑似命令，.elseifn 疑似命令，.ifdef 疑似命令，.ifndef 疑似命令が存在しない場合は，次のメッセージが出力され，アセンブルが中止されます。

```
F3510: .else unexpected
```

.elseif

絶対値式による制御（真のときアセンブル）をします。

［指定形式］

.elseif 絶対値式

［機能］

- オペランドで指定した絶対値式が真（≠ 0）に評価された場合

（a）本疑似命令と本疑似命令に対応する .else 疑似命令，.elseif 疑似命令，または .elseifn 疑似命令が存在する場合は，本疑似命令とその疑似命令とで囲まれるブロックをアセンブルします。

（b）それらの疑似命令が存在しない場合は，本疑似命令とその疑似命令に対応する .endif 疑似命令とで囲まれるブロックをアセンブルします。

- 偽（= 0）に評価された場合

本疑似命令に対応する .else 疑似命令，.elseif 疑似命令，.elseifn 疑似命令，または .endif 疑似命令までスキップします。

［使用例］

次のように展開されます。

【展開前】

```
.if     0
    .word   10
.elseif 10
    .str    "a"
.endif
.if     10 > 20
    .word   20
.elseif 10 == 20
    .str    "b"
.endif
.set    expr, 0
.if     expr
    .word   expr
.elseifn    expr - 10
    .str    "c"
.endif
```

【展開後】

```
.str    "a"
```

［注意事項］

- 対応する疑似命令が存在しない場合，次のメッセージが出力され，アセンブルが中止されます。

```
F3511: .endif unmatched
```

.elseifn

絶対値式による制御（偽のときアセンブル）をします。

［指定形式］

.elseifn　絶対値式

［機能］

- オペランドで指定した絶対値式が真（≠ 0）に評価された場合

本疑似命令に対応する .else 疑似命令，.elseif 疑似命令，.elseifn 疑似命令，または .endif 疑似命令までスキップします。

- 偽（= 0）に評価された場合

（a）本疑似命令と本疑似命令に対応する .else 疑似命令，.elseif 疑似命令，または .elseifn 疑似命令が存在する場合は，本疑似命令とその疑似命令とで囲まれるブロックをアセンブルします。

（b）それらの疑似命令が存在しない場合は，本疑似命令とその疑似命令に対応する .endif 疑似命令とで囲まれるブロックをアセンブルします。

［使用例］

次のように展開されます。

【展開前】

```
.if     0
    .word   10
.elseifn    10
    .str    "a"
.endif
.if     10 > 20
    .word   20
.elseifn    10 >= 20
    .str    "b"
.endif
.set    expr, 0
.if     expr
    .word   expr
.elseif expr - 10
    .str    "c"
.endif
```

【展開後】

```
.str    "b"
.str    "c"
```

［注意事項］

- 対応する疑似命令が存在しない場合，次のメッセージが出力され，アセンブルが中止されます。

```
F3511: .endif unmatched
```

.endif

制御範囲の終わりを示します。

［指定形式］

.endif

［機能］

条件アセンブル疑似命令による制御の範囲の終わりを示します

［注意事項］

- 本疑似命令に対応する .if 疑似命令，.ifn 疑似命令，.elseif 疑似命令，.elseifn 疑似命令，.ifdef 疑似命令，.ifndef 疑似命令が存在しない場合，次のメッセージが出力され，アセンブルが中止されます。

```
F3510: .endif unexpected
```

4. 2. 11　スキップ疑似命令

as850 では，スキップ疑似命令を用いることにより，繰り返しアセンブル疑似命令において残りの繰り返しをスキップすることができます。次に，この項において説明するスキップ疑似命令を示します。

表 4―23　スキップ疑似命令

疑似命令	意味
.exitm	1 つ外側にスキップ
.exitma	一番外側にスキップ

.exitm

1 つ外側にスキップします。

［指定形式］

.exitm

［機能］

本疑似命令は，本疑似命令を囲んでいる最も内側の繰り返しアセンブル疑似命令の繰り返しアセンブルをスキップします。

［使用例］

次のように展開されます。

【展開前】

```
.repeat 2
    .set    expr, 1
    .word   10
    .repeat 10
        .if     expr < 5
            .byte   expr
            .set    expr, expr + 1
        .else
            .ifdef  undefine_symbol
                .byte   expr
                .set    expr, expr + 1
            .else
                .exitm
            .endif
        .endif
    .endm
    .hword  20
    .hword  30
.endm
.word   expr
```

【展開後】

```
.word   10
.byte   1
.byte   2
.byte   3
.byte   4
.hword  20
.hword  30
.word   10
.byte   1
.byte   2
.byte   3
.byte   4
.hword  20
.hword  30
.word   5
```

［注意事項］

- 本疑似命令が繰り返しアセンブル疑似命令に囲まれていない場合，次のメッセージが出力され，アセンブルが中止されます。

```
F3513: unexpected EOF in .repeat/.irepeat
```

.exitma

一番外側にスキップします。

［指定形式］

.exitma

［機能］

本疑似命令は，本疑似命令を囲んでいる最も外側の繰り返しアセンブル疑似命令の繰り返しをスキップします。

［使用例］

次のように展開されます。

【展開前】

```
.repeat 2
    .set    expr, 1
    .word   10
    .repeat 10
        .if     expr < 5
            .byte   expr
            .set    expr, expr + 1
        .else
            .ifdef  undefine_symbol
                .byte   expr
                .set    expr, expr + 1
            .else
                .exitma
            .endif
        .endif
    .endm
    .hword  20
    .hword  30
.endm
.word   expr
```

【展開後】

```
.word   10
.byte   1
.byte   2
.byte   3
.byte   4
.word   5
```

［注意事項］

- 本疑似命令が繰り返しアセンブル疑似命令に囲まれていない場合，次のメッセージが出力され，アセンブルが中止されます。

```
F3515: .exitma not in .repeat/.irepeat
```

4. 2. 12　マクロ疑似命令

as850 では，マクロ疑似命令を用いることにより，任意の文の並びを指定したマクロ名に対応するマクロ本体として定義できます。また，ソース・プログラム内でこのマクロ名を参照することにより，その位置にこのマクロ名に対応する文の並びを記述したものとして扱うことができます。次に，この項において説明するマクロ疑似命令を示します。

表 4―24　マクロ疑似命令

疑似命令	意味
.macro	マクロ定義の始まり
.endm	繰り返しの区間の終わり，またはマクロ定義の終わり
.local	ローカル・シンボルの定義

.macro

マクロ定義の始まりを示します。

［指定形式］

.macro マクロ名 [仮パラメータ , …]

［機能］

本疑似命令と .endm 疑似命令とで囲まれた文の並びを，第 1 オペランドで指定したマクロ名に対するマクロ本体として定義します。このマクロ名が参照された場合，その位置にこのマクロ名に対応するマクロ本体が記述されたものとして扱います。

［使用例］

次のように展開されます。

【展開前】

```
.macro  PUSH    REG
    add     -4, sp
    st.w    REG, 0x0[sp]
.endm
.macro  POP     REG
    ld.w    0x0[sp], REG
    add     0x4, sp
.endm
    PUSH    r10
    mov     10, r10
    add     r10, r20
    POP     r10
```

【展開後】

```
add     -4, sp
st.w    r10, 0x0[sp]
mov     10, r10
add     r10, r20
ld.w    0x0[sp], r10
add     0x4, sp
```

［注意事項］

- 本疑似命令に対応する .endm 疑似命令が存在しない場合，次のメッセージが出力され，アセンブルが中止されます。

```
F3513: unexpected EOF in .macro
```

- あるマクロ名が再定義された場合，それ以降のそのマクロ呼び出しに対しては，再定義されたマクロ本体がそのマクロ名に対応するマクロ本体となります。
- 33 個以上の仮パラメータを指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
F3514: paramater table overflow
```

- マクロ本体内で参照されていない余分な仮パラメータは無視されます。この場合，as850 では，メッセージが出力されないので注意が必要です。
- マクロ呼び出しにおいて実パラメータが足りない場合，次のメッセージが出力され，アセンブルが中止されます。

```
F3519: argument mismatch
```

- マクロ本体内で，未定義なマクロの呼び出しが行われた場合，次のメッセージが出力され，アセンブルが中止されます。

```
E3249: illegal syntax
```

- マクロ本体内で，現在定義中のマクロの呼び出しが行われた場合，次のメッセージが出力され，アセンブルが中止されます。

```
F3518: unreasonable macro_call nesting
```

- 仮パラメータにラベルや疑似命令で定義済みのパラメータを指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E3212: symbol already defined as string
```

- マクロ呼び出しにおいて実パラメータに指定可能なものは，ラベル名，シンボル名，数値，レジスタ，および命令ニモニックのみです。
  ラベル式（LABEL-1），参照方法指定ラベル（#LABEL），またはベース・レジスタ指定（[gp]）などを指定した場合，指定された実パラメータに依存したメッセージが出力され，アセンブルが中止されます。
- マクロ本体内には，文の並びが指定可能です。オペランドなど，文の一部を指定することはできません。オペランドにマクロ呼び出しが存在する場合，マクロ名の未定義なラベル参照として扱われるか，次のメッセージが出力され，アセンブルが中止されます。

```
E3249: illegal syntax
```

.endm

繰り返しの区間の終わり，またはマクロ定義の終わりを示します。

［指定形式］

.endm

［機能］

繰り返しの区間は終わり，またはマクロ本体の終わりを示します。

［注意事項］

- 本疑似命令に対応する .repeat 疑似命令，.irepeat 疑似命令，.macro 疑似命令が存在しない場合，次のメッセージが出力され，アセンブルが中止されます。

```
F3510: .endm unexpected
```

.local

ローカル・シンボルの定義をします。

[指定形式]

.local ローカル・シンボル [, ローカル・シンボル , …]

[機能]

指定した文字列を特有の識別子として置き換えられるローカル・シンボルとして宣言します。

[使用例]

次のように展開されます。

【展開前】

```
.macro  m1      x
    .local  a, b
    a:      .word   a
    b:      .word   x
.endm
m1  10
m1  20
```

【展開後】

```
.??0000:    .word   .??0000
.??0001:    .word   10
.??0002:    .word   .??0002
.??0003:    .word   20
```

[注意事項]

- 本疑似命令の仮パラメータに 33 個以上のローカル・シンボルを指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
F3514: paramater table overflow
```

- アセンブラによって生成されるローカル・シンボル名は，.??0000 から .??FFFF までの範囲で生成されます。
- アセンブラによって生成されるローカル・シンボル名は，条件アセンブル疑似命令において未定義として扱われます。

4.3 マクロ

この節では，マクロ機能の使い方について説明します。
プログラムの中で一連の命令群を何回も記述する場合に使用すると，便利な機能です。

4.3.1 概　　要

ソースの中で一連の命令群を何回も記述する場合，マクロ機能を使用すると便利です。

マクロ機能とは，.macro，.endm 疑似命令により，マクロ・ボディとして定義された一連の命令群をマクロ参照している箇所に展開することです。

マクロは，ソースの記述性を向上させるために使用するもので，サブルーチンとは異なります。

マクロとサブルーチンには，それぞれ次のような特徴があります。それぞれ目的に応じて有効に使用してください。

- サブルーチン

  プログラム中で何回も必要となる処理を 1 つのサブルーチンとして記述します。サブルーチンは，アセンブラにより一度だけ機械語に変換されます。

  サブルーチンの参照には，サブルーチン・コール命令（一般にはその前後に引数設定の命令が必要）を記述するだけで済みます。したがって，サブルーチンを活用することにより，プログラムのメモリを効率よく使用することができます。

  プログラム中の一連のまとまった処理をサブルーチン化することにより，プログラムの構造化を図ることができます（プログラムを構造化することにより，プログラム全体の構造が分かりやすくなり，プログラムの設計が容易になります）。

- マクロ

  マクロの基本的な機能は，命令群の置き換えです。

  .macro，.endm 疑似命令によりマクロ・ボディとして定義された一連の命令群が，マクロ参照時にその場所に展開されます。アセンブラは，マクロ参照を検出するとマクロ・ボディを展開し，マクロ・ボディの仮パラメータを参照時の実パラメータに置き換えながら，命令群を機械語に変換します。

  マクロは，パラメータを記述することができます。

  たとえば，処理手順は同じであるがオペランドに記述するデータだけが異なる命令群がある場合，そのデータに仮パラメータを割り当ててマクロを定義します。マクロ参照時には，マクロ名と実パラメータを記述することにより，記述の一部分だけが異なる種々の命令群に対処することができます。

サブルーチン化の手法が，メモリ・サイズの削減やプログラムの構造化を図るために用いられるのに対し，マクロは，コーディングの効率を向上させるために用いられます。

4. 3. 2 マクロの利用

マクロとは，一連の決まった手順パターンを登録し，それを利用して記述するものです。マクロはユーザが定義します。マクロの定義方法は次のように，マクロ本体を“.macro”と“.endm”で囲む形になります。

```
.macro  PUSH    REG     -- 次の 2 つの文がマクロ本体
add     -4, sp
st.w    REG, 0x0[sp]
.endm
```

上記を定義したあと，次のように記述した場合，「r19 をスタックに格納する」というコードに置き換えられます。

```
PUSH    r19
```

したがって，次のようなコードに展開されます。

```
add     -4, sp
st.w    r19, 0x0[sp]
```

4. 3. 3 マクロ内のシンボル

マクロ内で定義するシンボルには，グローバル・シンボルとローカル・シンボルの 2 種類があります。

- グローバル・シンボル

  ソース内のすべてのステートメントから参照することができます。

  したがって，そのシンボルを定義しているマクロを 2 回以上参照し，一連のステートメントが展開されると，シンボルは二重定義エラーとなります。

  .local 疑似命令で定義されていないシンボルは，グローバル・シンボルです。

- ローカル・シンボル

  ローカル・シンボルは，.local 疑似命令で定義します（「4. 2. 12 マクロ疑似命令」を参照してください）。

  ローカル・シンボルは，.local 疑似命令で宣言されたマクロ内でのみ参照することができます。

  マクロ外からローカル・シンボルを参照することはできません。

使用例を以下に示します。

```
.macro  m1      x
    .local  a, b
    a:      .word   a
    b:      .word   x
.endm
m1  10
m1  20
```

4. 3. 4 マクロ・オペレータ

この項では，マクロ本体内において長さ 0 の区切り子として用いることができるチルダ記号“~”，マクロ呼び出しにおいてシンボル値を引数として指定する場合に用いるダラー記号“$”について説明します。

(1) チルダ記号

as850 では，マクロ本体内に現れたチルダ記号“~”は長さ 0 の区切り子として扱われます。ただし，文字列定数やコメントの中に現れた場合，区切り子としては扱われず，通常のチルダ記号“~”として扱われます。

例 1.

```
.macro  abc     x
    abc~x:  mov     r10, r20
            sub def~x, r20
.endm
abc STU
```

```
【展開結果】
abcSTU  :   mov     r10, r20
            sub     defSTU, r20
```

2.

```
.macro  abc     x, xy
    a_~xy:  mov     r10, r20
    a_~x~y: mov     r20, r10
.endm
abc stu, STU
```

```
【展開結果】
a_STU:      mov     r10, r20
a_stuy:     mov     r20, r10
```

3.

```
.macro  abc     x, xy
    ~ab:    mov r10, r20
.endm
abc stu, STU
```

```
【展開結果】
ab:     mov     r10, r20
```

(2) ダラー記号

as850 では，マクロ呼び出しにおける実引数としてダラー記号“$”を前に置いたシンボルを指定した場合，そのシンボルの値が実引数として指定されたものみなされます。ただし，ダラー記号 $ に続けてシンボル以外の識別名，または未定義シンボル名を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
F3520: $ must be followed by defined symbol
```

例

```
.macro  mac1    x
    mov     x, r10
.endm
.macro  mac2
    .set    value, 10
    mac1    value
    mac1    $value
.endm
mac2
```

```
【展開結果】
.set    value, 10
mov     value, r10
mov     10, r10
```

4.4 予約語

as850 には予約語が存在します。予約語をシンボル，ラベル，セクション名に使用することはできません。予約語を指定した場合は，次のメッセージが出力され，アセンブルが中止されます。

```
E3245: identifier is reserved word
```

予約語は次のとおりです。

- 命令（add，sub，mov など）
- 疑似命令（.section，.lcomm，.globl など）
- hi，lo，hi1（hi ( )，lo ( )，hi1 ( ) として使用しているため）
- レジスタ名

4.5 インストラクション

この節では，V850 マイクロコントローラ製品の持つ各種命令機能を説明します。

4.5.1 メモリ空間

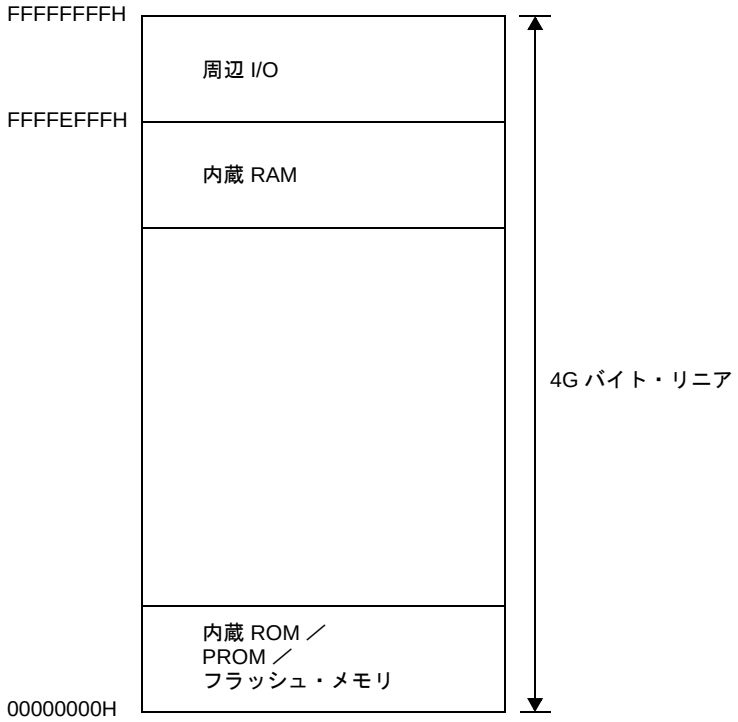
V850 マイクロコントローラは，32 ビット・アーキテクチャであり，オペランド・アドレシングにおいては，最大 4G バイトのリニア・アドレス空間（データ空間）をサポートしています。

一方，命令アドレスのアドレスにおいては，最大 16M バイトのリニア・アドレス空間（プログラム空間）をサポートしています。

以下に，V850 マイクロコントローラのメモリ・マップを示します。

ただし，内蔵 ROM，内蔵 RAM などの容量については，製品ごとにことなるため，詳細は，各製品のユーザーズ・マニュアルを参照してください。

図 4―7 V850 マイクロコントローラのメモリ・マップ



4.5.2 レジスタ

レジスタは，一般のプログラム用として使用するプログラム・レジスタと，実行環境の制御用として使用するシステム・レジスタの 2 種類に大別することができます。レジスタは，どれも 32 ビット幅を持っています。

システム・レジスタは，アーキテクチャにより異なります。詳細は「(2) システム・レジスタ」を参照してください。

図 4―8　プログラム・レジスタ

31 0

r0：ゼロ・レジスタ
r1：アセンブラ予約レジスタ
r2
r3：スタック・ポインタ（SP）
r4：グローバル・ポインタ（GP）
r5：テキスト・ポインタ（TP）
r6
r7
r8
r9
r10
r11
r12
r13
r14
r15
r16
r17
r18
r19
r20
r21
r22
r23
r24
r25
r26
r27
r28
r29
r30：エレメント・ポインタ（EP）
r31：リンク・ポインタ（LP）

PC：プログラム・カウンタ

図 4―9 システム・レジスタ

31 0

EIPC：割り込み時状態退避レジスタ
EIPSW：割り込み時状態退避レジスタ
FEPC：NMI 時状態退避レジスタ
FEPSW：NMI 時状態退避レジスタ
ECR：割り込み要因レジスタ
PSW：プログラム・ステータス・ワード
CTPC：CALLT 実行時状態退避レジスタ
CTPSW：CALLT 実行時状態退避レジスタ
DBPC：例外／デバッグ・トラップ時状態退避レジスタ
DBPSW：例外／デバッグ・トラップ時状態退避レジスタ
CTBP：CALLT ベース・ポインタ
DIR：デバッグ・インタフェース・レジスタ
BPC0：ブレークポイント制御レジスタ
BPC1：ブレークポイント制御レジスタ
BPC2：ブレークポイント制御レジスタ
BPC3：ブレークポイント制御レジスタ
ASID：プログラム ID レジスタ
BPAV0：ブレークポイント・アドレス設定レジスタ
BPAV1：ブレークポイント・アドレス設定レジスタ
BPAV2：ブレークポイント・アドレス設定レジスタ
BPAV3：ブレークポイント・アドレス設定レジスタ
BPAM0：ブレークポイント・アドレス・マスク・レジスタ
BPAM1：ブレークポイント・アドレス・マスク・レジスタ
BPAM2：ブレークポイント・アドレス・マスク・レジスタ
BPAM3：ブレークポイント・アドレス・マスク・レジスタ
BPDV0：ブレークポイント・データ設定レジスタ
BPDV1：ブレークポイント・データ設定レジスタ
BPDV2：ブレークポイント・データ設定レジスタ
BPDV3：ブレークポイント・データ設定レジスタ
BPDM0：ブレークポイント・データ・マスク・レジスタ
BPDM1：ブレークポイント・データ・マスク・レジスタ
BPDM2：ブレークポイント・データ・マスク・レジスタ
BPDM3：ブレークポイント・データ・マスク・レジスタ

(1) プログラム・レジスタ

プログラム・レジスタには，汎用レジスタ（r0-r31）とプログラム・カウンタ（PC）があります。

表 4―25 プログラム・レジスタ

名称	用途	動作
r0	ゼロ・レジスタ	常に，0 を保持
r1	アセンブラ予約レジスタ	アドレスを生成する際のワーキング・レジスタ
r2	アドレス／データ変数用レジスタ（使用するリアルタイム OS が r2 を使用していない場合）	
r3	スタック・ポインタ	関数コール時，スタック・フレームを生成する際に使用
r4	グローバル・ポインタ	データ領域のグローバル変数をアクセスする際に使用
r5	テキスト・ポインタ	テキスト領域（プログラム・コードを配置する領域）の先頭を示すレジスタとして使用
r6-r29	アドレス／データ変数用レジスタ	
r30	エレメント・ポインタ	メモリ・アクセス時，アドレスを生成する際のベース・ポインタとして使用
r31	リンク・ポインタ	コンパイラが関数コールをする際に使用
PC	プログラム・カウンタ	プログラム実行中の命令アドレスを保持

(a) 汎用レジスタ：r0-r31

汎用レジスタとして，r0-r31 の 32 本が用意されています。これらのレジスタは，どれもアドレス変数用，またはデータ変数用として利用できます。

ただし，r0-r5，r30-r31 を使用する際には，以下の注意が必要です。

- r0，r30

命令により暗黙的に使用されます。

r0 は常に 0 を保持しているレジスタであり，0 を使用する演算やオフセット 0 のアドレッシングで使用されます。

r30 は SLD 命令，または SST 命令により，メモリをアクセスする際のベース・ポインタとして使用されます。

- r1，r3-r5，r31

アセンブラ，および C コンパイラにより暗黙的に使用されます。

これらのレジスタを使用する際には，レジスタの内容を破壊しないように退避してから使用し，使用後には元の状態に戻す必要があります。

- r2

リアルタイム OS が使用する場合があります。

リアルタイム OS が r2 を使用していない場合は，アドレス変数用，またはデータ変数用として利用できます。

（b）プログラム・カウンタ：PC

プログラム実行中の命令アドレスを保持しています。

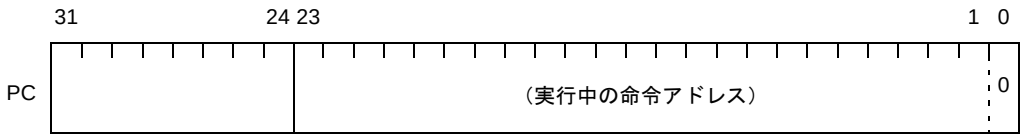
なお，PC の各ビットの意味は，CPU の種類（V850，V850ES，V850E1，V850E2）により異なります。

- V850

ビット 23-0 が有効で，ビット 31-24 は将来の機能拡張のために予約されています（0 に固定）。

ビット 23 からビット 24 へのキャリーが発生しても無視します。また，ビット 0 は 0 に固定されており，奇数番地への分岐はできません。

図 4―10 プログラム・カウンタ【V850】

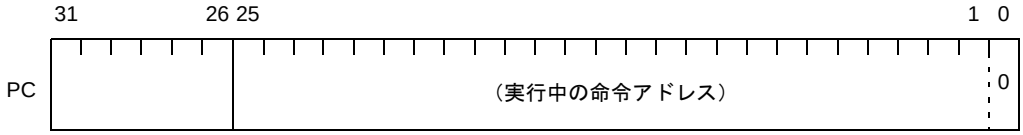


- V850ES，V850E1

ビット 25-0 が有効で，ビット 31-26 は将来の機能拡張のために予約されています（0 に固定）。

ビット 25 からビット 26 へのキャリーが発生しても無視します。また，ビット 0 は 0 に固定されており，奇数番地への分岐はできません。

図 4―11 プログラム・カウンタ【V850ES，V850E1】

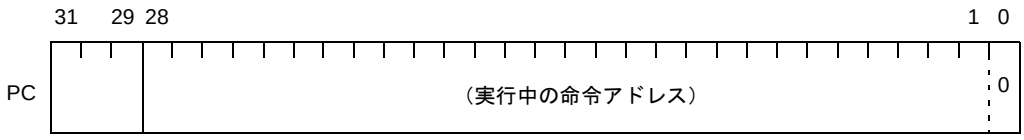


- V850E2

ビット 28-0 が有効で，ビット 31-29 は将来の機能拡張のために予約されています（0 に固定）。

ビット 28 からビット 29 へのキャリーが発生しも無視します。また，ビット 0 は 0 に固定されており，奇数番地への分岐はできません。

図 4―12 プログラム・カウンタ【V850E2】



(2) システム・レジスタ

システム・レジスタは，CPU の状態制御，割り込み情報保持などを行います。

システム・レジスタへのリード／ライトは，システム・レジスタのロード／ストア命令（LDSR，STSR 命令）により，次に示すレジスタ番号を指定することで行います。

表 4―26 システム・レジスタ番号

レジスタ番号	レジスタ名	オペランド指定の可否	
		LDSR 命令	STSR 命令
0	割り込み時状態退避レジスタ EIPC	○	○
1	割り込み時状態退避レジスタ EIPSW	○	○
2	NMI 時状態退避レジスタ FEPC	○	○
3	NMI 時状態退避レジスタ FEPSW	○	○
4	割り込み要因レジスタ ECR	―	○
5	プログラム・ステータス・ワード PSW	○	○
6-15	予約番号	―	―
16	【V850ES，V850E1，V850E2】 CALLT 実行時状態退避レジスタ CTPC	○	○
17	【V850ES，V850E1，V850E2】 CALLT 実行時状態退避レジスタ CTPSW	○	○
18	【V850ES，V850E1，V850E2】 例外／デバッグ・トラップ時状態退避レジスタ DBPC	○	○注 1
19	【V850ES，V850E1，V850E2】 例外／デバッグ・トラップ時状態退避レジスタ DBPSW	○	○注 1
20	【V850ES，V850E1，V850E2】 CALLT ベース・ポインタ CTBP	○	○
21	【V850ES，V850E1，V850E2】 デバッグ・インタフェース・レジスタ DIR	○注 1	○
22	【V850E1，V850E2】 ブレークポイント制御レジスタ BPCn 注 2	○注 1	○注 1
23	【V850E1，V850E2】 プログラム ID レジスタ ASID	○	○
24	【V850E1，V850E2】 ブレークポイント・アドレス設定レジスタ BPAVn 注 2	○注 1	○注 1
25	【V850E1，V850E2】 ブレークポイント・アドレス・マスク・レジスタ BPAMn 注 2	○注 1	○注 1
26	【V850E1，V850E2】 ブレークポイント・データ設定レジスタ BPDVn 注 2	○注 1	○注 1
27	【V850E1，V850E2】 ブレークポイント・データ・マスク・レジスタ BPDMn 注 2	○注 1	○注 1

レジスタ番号	レジスタ名	オペランド指定の可否	
		LDSR 命令	STSR 命令
28-31	【V850E1，V850E2】 予約番号	－	－

注 1. V850E1 のタイプ A，B のデバッグ・モード時に限りアクセス可能です。その他の製品ではアクセス禁止です。

2. 実際にアクセスされるレジスタは，DIR.CS フラグによって設定されます。

備考 n = 0-3

－：アクセス禁止

○：アクセス可能

注意 LDSR 命令により EIPC，FEPC，CTPC，または DBPC のビット 0 をセット（1）したあと，割り込み処理を行い，RETI 命令で復帰する際，ビット 0 は無視されます（PC のビット 0 が 0 固定のため）。EIPC，FEPC，CTPC を設定する場合は，偶数値（ビット 0 = 0）を設定してください。

（a）割り込み時状態退避レジスタ：EIPC，EIPSW【V850，V850ES，V850E，V850E2】

割り込み時状態退避レジスタには，EIPC と EIPSW があります。

これらのレジスタには，ソフトウエア例外やマスカブル割り込みが発生した際，PC の内容が EIPC に，PSW の内容が EIPSW に退避されます（ノンマスカブル割り込み（NMI）やラン・タイム・エラーが発生した際には，NMI 時状態退避レジスタに退避されます）。

- EIPC

一部の命令を除き，ソフトウエア例外やマスカブル割り込みが発生した際に実行していた命令の次の命令のアドレスが退避されます。

- EIPSW

ソフトウエア例外やマスカブル割り込みが発生した際の PSW の内容が退避されます。

割り込み時状態退避レジスタは 1 組しかないため，多重割り込みを許可する場合はプログラムによってこれらのレジスタの内容を退避する必要があります。

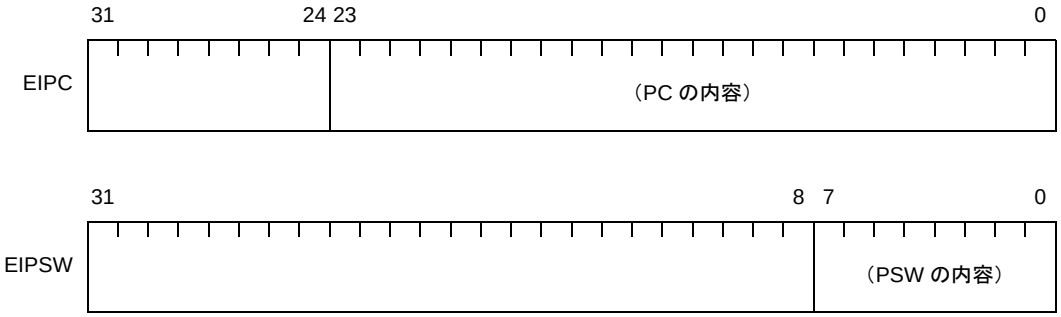
なお，EIPC，EIPSW の各ビットの意味は，CPU の種類（V850，V850ES，V850E1，V850E2）により異なります。

- V850

EIPC は，ビット 23-0 が有効で，ビット 31-24 は将来の機能拡張のために予約されています（0 に固定）。

EIPSW は，ビット 7-0 が有効で，ビット 31-8 は将来の機能拡張のために予約されています（0 に固定）。

図 4―13 割り込み時状態退避レジスタ【V850】

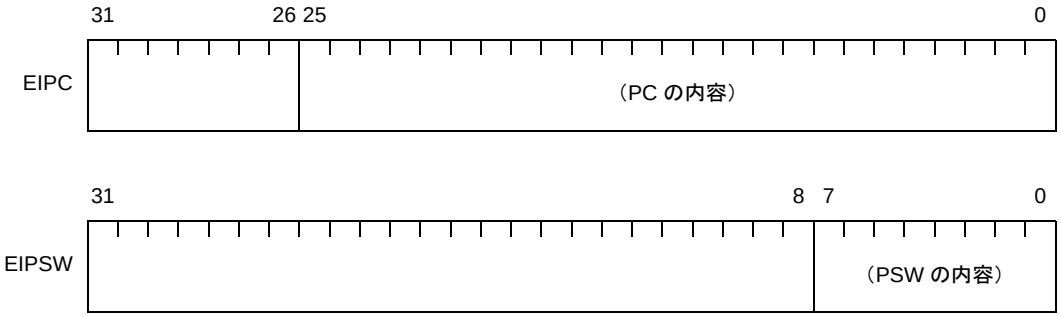


- V850ES

EIPC は，ビット 25-0 が有効で，ビット 31-26 は将来の機能拡張のために予約されています（0 に固定）。

EIPSW は，ビット 7-0 が有効で，ビット 31-8 は将来の機能拡張のために予約されています（0 に固定）。

図 4―14 割り込み時状態退避レジスタ【V850ES】



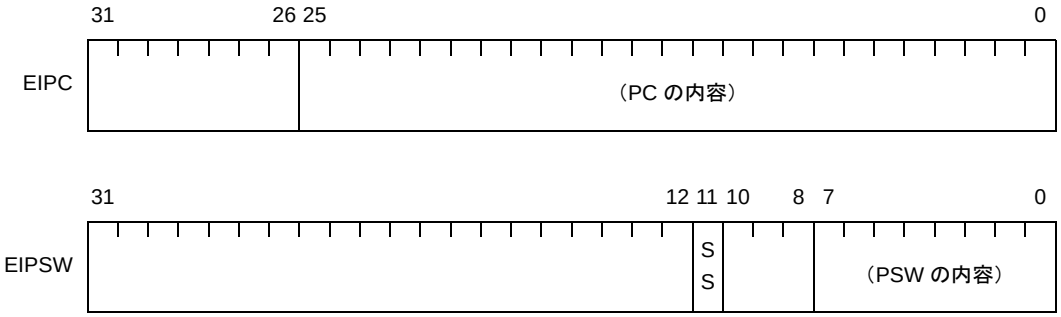
- V850E1

EIPC は，ビット 25-0 が有効で，ビット 31-26 は将来の機能拡張のために予約されています（0 に固定）。

EIPSW は，ビット 11，7-0 が有効で，ビット 31-12，10-8 は将来の機能拡張のために予約されています（0 に固定）。

なお，EIPSW のビット 11 には，PSW の SS フラグが退避されます。

図 4―15 割り込み時状態退避レジスタ【V850E1】



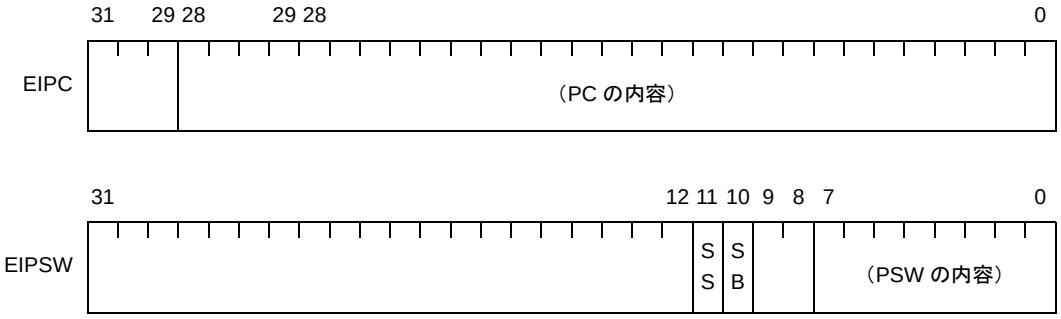
- V850E2

EIPC は，ビット 28-0 が有効で，ビット 31-29 は将来の機能拡張のために予約されています（0 に固定）。

EIPSW は，ビット 11-10，7-0 が有効で，ビット 31-12，9-8 は将来の機能拡張のために予約されています（0 に固定）。

なお，EIPSW のビット 11 には，PSW の SS フラグが，EIPSW のビット 10 には，PSW の SB フラグが退避されます。

図 4―16 割り込み時状態退避レジスタ【V850E2】



（b）NMI 時状態退避レジスタ：FEPC，FEPSW【V850，V850ES，V850E1，V850E2】

NMI 時状態退避レジスタには，FEPC と FEPSW があります。

これらのレジスタには，ノンマスカブル割り込み（NMI）やラン・タイム・エラーが発生した際，PC の内容が FEPC に，PSW の内容が FEPSW に退避されます（ソフトウエア例外やマスカブル割り込みが発生した際には，割り込み時状態退避レジスタに退避されます）。

- FEPC

一部の命令を除き，ノンマスカブル割り込みやラン・タイム・エラーが発生した際に実行していた命令の次の命令のアドレスが退避されます。

- FEPSW

ノンマスカブル割り込みやラン・タイム・エラーが発生した際の PSW の内容が退避されます。

NMI 時状態退避レジスタは 1 組しかないため，多重割り込みを許可する場合はプログラムによってこれらのレジスタの内容を退避する必要があります。

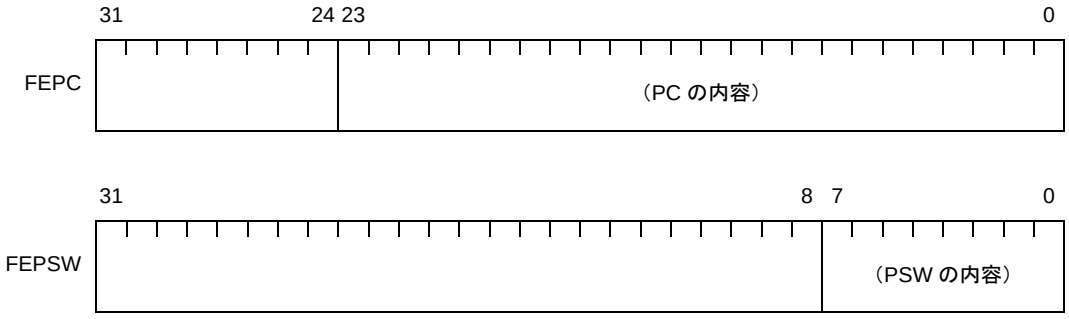
なお，FEPC，PEPSW の各ビットの意味は，CPU の種類（V850，V850ES，V850E1，V850E2）により異なります。

- V850

FEPC は，ビット 23-0 が有効で，ビット 31-24 は将来の機能拡張のために予約されています（0 に固定）。

FEPSW は，ビット 7-0 が有効で，ビット 31-8 は将来の機能拡張のために予約されています（0 に固定）。

図 4―17 NMI 時状態状態退避レジスタ【V850】

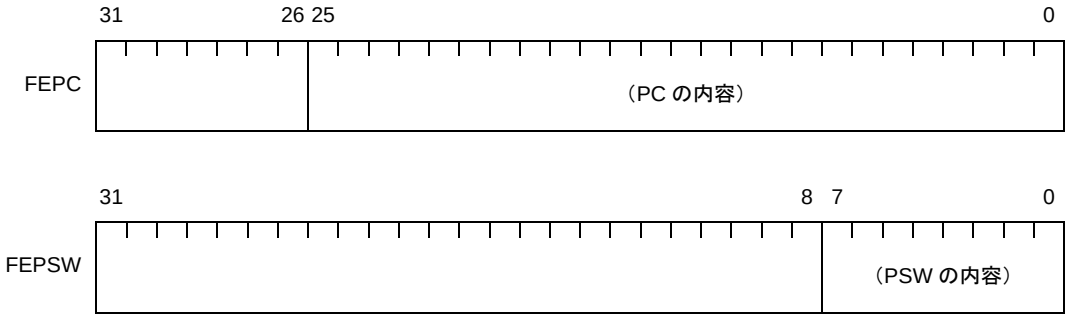


- V850ES

FEPC は，ビット 25-0 が有効で，ビット 31-26 は将来の機能拡張のために予約されています（0 に固定）。

FEPSW は，ビット 7-0 が有効で，ビット 31-8 は将来の機能拡張のために予約されています（0 に固定）。

図 4―18 NMI 時状態状態退避レジスタ【V850ES】



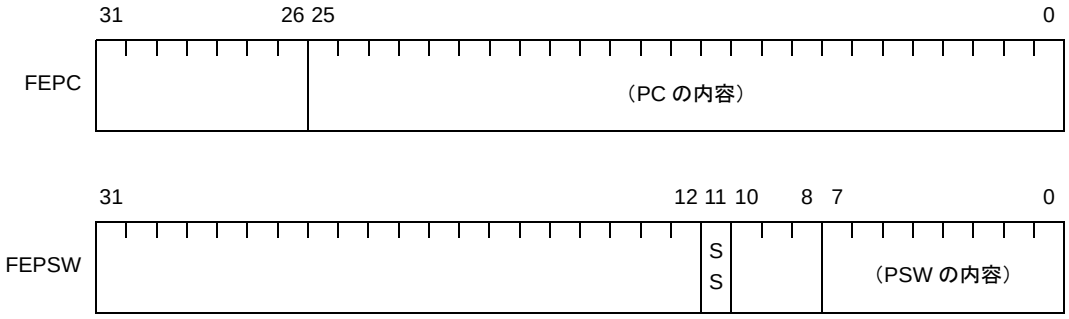
- V850E1

FEPC は，ビット 25-0 が有効で，ビット 31-26 は将来の機能拡張のために予約されています（0 に固定）。

FEPSW は，ビット 11，7-0 が有効で，ビット 31-12，10-8 は将来の機能拡張のために予約されています（0 に固定）。

なお，FEPSW のビット 11 には，PSW の SS フラグが退避されます。

図 4―19 NMI 時状態退避レジスタ【V850E1】



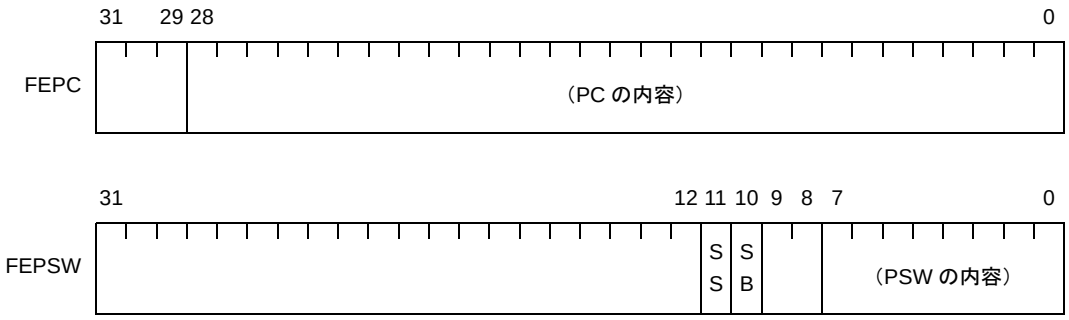
- V850E2

FEPC は，ビット 28-0 が有効で，ビット 31-29 は将来の機能拡張のために予約されています（0 に固定）。

FEPSW は，ビット 11-10，7-0 が有効で，ビット 31-12，9-8 は将来の機能拡張のために予約されています（0 に固定）。

なお，FEPSW のビット 11 には，PSW の SS フラグが，FEPSW のビット 10 には，PSW の SB フラグが退避されます。

図 4―20 NMI 時状態退避レジスタ【V850E2】



（c）割り込み要因レジスタ：ECR【V850，V850ES，V850E1，V850E2】

ソフトウエア例外，マスカブル割り込み，ノンマスカブル割り込みが発生した際，その要因（割り込み要因ごとにコード化された値）が保持されます。

なお，このレジスタは読み出し専用のため，LDSR 命令を使用してこのレジスタに値を書き込むことはできません。

図 4―21 割り込み要因レジスタ【V850，V850ES，V850E1，V850E2】

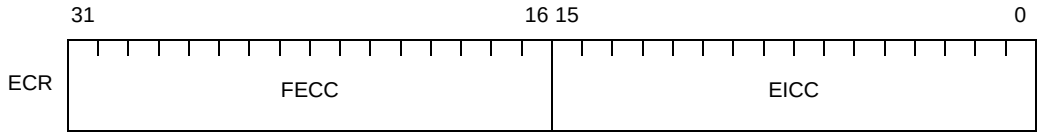


表 4―27 割り込み要因レジスタ【V850，V850ES，V850E1，V850E2】

ビット位置	フラグ名	意味
31-16	FECC	ノンマスカブル割り込み（NMI）の例外コード
15-0	EICC	ソフトウエア例外／マスカブル割り込みの例外コード

（d）プログラム・ステータス・ワード：PSW【V850，V850ES，V850E1，V850E2】

プログラムの状態（命令実行の結果）や CPU の状態を示すフラグの集合です。

LDSR 命令を使用してこのレジスタの各ビットの内容を変更した場合には，LDSR 命令実行終了直後から変更内容が有効となります。ただし，ID フラグをセット（1）した場合は，LDSR 命令実行中から割り込み要求の受け付けを禁止します。

なお，PSW の各ビットの意味は，CPU の種類（V850，V850ES，V850E1，V850E2）により異なります。

- V850，V850ES

ビット 7-0 が有効で，ビット 31-8 は将来の機能拡張のために予約されています（0 に固定）。

図 4―22 プログラム・ステータス・ワード【V850，V850ES】

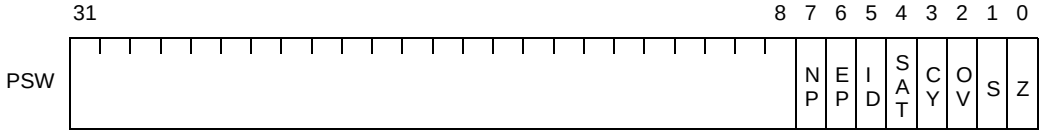


表 4―28 プログラム・ステータス・ワード【V850，V850ES】

ビット位置	フラグ名	意味
7	NP	ノンマスカブル割り込み（NMI）処理中がどうかとを示します。NMI 要求が受け付けられるとセット（1）され，多重割り込みを禁止します。 0：NMI 処理中でない。 1：NMI 処理中である。
6	EP	ソフトウエア例外処理中がどうかを示します。ソフトウエア例外が発生するとセット（1）されます。 なお，このビットがセットされてもマスカブル割り込み要求は受け付けます。 0：ソフトウエア例外処理中でない。 1：ソフトウエア例外処理中である。
5	ID	マスカブル割り込み要求を受け付ける状態かどうかを示します。 0：割り込み許可（EI） 1：割り込み禁止（DI）
4	SAT 注	飽和演算命令の演算結果がオーバフローし，演算結果が飽和していることを示します。累積フラグのため，飽和演算命令で演算結果が飽和すると，セット（1）され，以降の命令の演算結果が飽和しなくてもクリア（0）されません。クリア（0）する場合は，LDSR 命令を使用して PSW にデータをロードします。なお，一般の算術演算命令ではセット（1）もクリア（0）も行われません。 0：飽和していない。 1：飽和している。
3	CY	演算結果に，キャリー，またはボローが発生したかどうかを示します。 0：キャリー，またはボローは発生していない。 1：キャリー，またはボローが発生した。
2	OV 注	演算中にオーバフローが発生したかどうかを示します。 0：オーバフローは発生していない。 1：オーバフローが発生した。
1	S 注	演算結果が負かどうかを示します。 0：演算結果は正または 0 であった。 1：演算結果は負であった。
0	Z	演算結果が 0 かどうかを示します。 0：演算結果は 0 でなかった。 1：演算結果は 0 であった。

注　飽和演算時の OV フラグと S フラグの内容で飽和処理した演算結果が決まります。また，飽和演算時に OV フラグがセット（1）された場合に限り，SAT フラグがセット（1）されます。

演算結果の状態	フラグの状態			飽和処理をした演算結果
	SAT	OV	S	
正の最大値を越えた	1	1	0	7FFFFFFFH
負の最大値を越えた	1	1	1	80000000H
正（最大値を越えない）	演算前の値を保持	0	0	演算結果そのもの
負（最大値を越えない）	演算前の値を保持	0	1	演算結果そのもの

- V850E1

ビット 11，7-0 が有効で，ビット 31-12，10-8 は将来の機能拡張のために予約されています（0 に固定）。

図 4―23　プログラム・ステータス・ワード【V850E1】

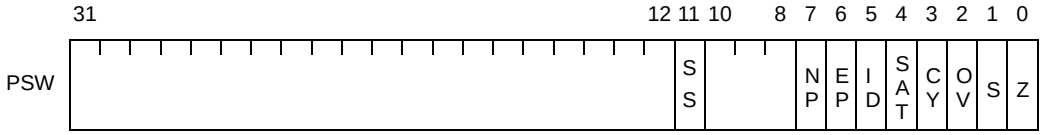


表 4―29　プログラム・ステータス・ワード【V850E1】

ビット位置	フラグ名	意味
11	SS 注 1	このフラグをセット（1）すると，シングルステップ実行で動作します（各命令の実行ごとにデバッグ・トラップを発生）。 このフラグは，割り込み処理ルーチンに分岐する際にクリア（0）されます。 また，DIR の SE フラグが 0 の場合は，このフラグはセットされません（0 に固定）。
7	NP	ノンマスカブル割り込み（NMI）処理中であることを示します。NMI 要求が受け付けられるとセット（1）され，多重割り込みを禁止します。 0：NMI 処理中でない。 1：NMI 処理中である。
6	EP	ソフトウエア例外処理中であることを示します。ソフトウエア例外の発生でセット（1）されます。 なお，このビットがセットされてもマスカブル割り込み要求は受け付けます。 0：ソフトウエア例外処理中でない。 1：ソフトウエア例外処理中である。

ビット位置	フラグ名	意味
5	ID	マスカブル割り込み要求を受け付ける状態かどうかを示します。 0：割り込み許可（EI） 1：割り込み禁止（DI）
4	SAT 注2	飽和演算命令の演算結果がオーバフローし，演算結果が飽和していることを示します。累積フラグのため，飽和演算命令で演算結果が飽和すると，セット（1）され，以降の命令の演算結果が飽和しなくてもクリア（0）されません。クリア（0）する場合は，LDSR 命令を使用して PSW にデータをロードします。なお，一般の算術演算命令ではセット（1）もクリア（0）も行われません。 0：飽和していない。 1：飽和している。
3	CY	演算結果に，キャリー，またはボローが発生したかどうかを示します。 0：キャリー，またはボローは発生していない。 1：キャリー，またはボローが発生した。
2	OV 注2	演算中にオーバフローが発生したかどうかを示します。 0：オーバフローは発生していない。 1：オーバフローが発生した。
1	S 注2	演算結果が負かどうかを示します。 0：演算結果は正または 0 であった。 1：演算結果は負であった。
0	Z	演算結果が 0 かどうかを示します。 0：演算結果は 0 でなかった。 1：演算結果は 0 であった。

注 1. V850E1 のタイプ A，B に限りアクセス可能です。その他の製品ではアクセス禁止です。

2. 飽和演算時の OV フラグと S フラグの内容で飽和処理した演算結果が決まります。また，飽和演算時に OV フラグがセット（1）された場合に限り，SAT フラグがセット（1）されます。

演算結果の状態	フラグの状態			飽和処理をした演算結果
	SAT	OV	S	
正の最大値を越えた	1	1	0	7FFFFFFFH
負の最大値を越えた	1	1	1	80000000H
正（最大値を越えない）	演算前の値を保持	0	0	演算結果そのもの
負（最大値を越えない）	演算前の値を保持	0	1	演算結果そのもの

- V850E2

ビット 11-10，7-0 が有効で，ビット 31-12，9-8 は将来の機能拡張のために予約されています（0 に固定）。

図 4―24 プログラム・ステータス・ワード【V850E2】

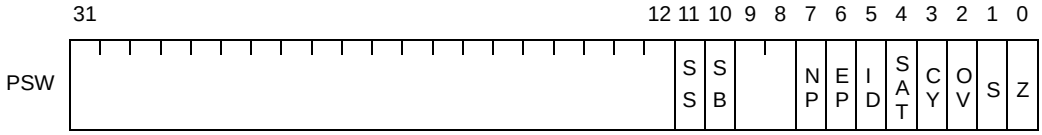


表 4―30 プログラム・ステータス・ワード【V850E2】

ビット位置	フラグ名	意味
11	SS	このフラグをセット（1）すると，シングルステップ実行で動作します（各命令の実行ごとにデバッグ・トラップを発生）。 ただし，割り込み処理ルーチンに分岐する際は，SB フラグの内容が転送されます。このため，SB フラグがクリア（0）されていると，割り込み処理ルーチンでのシングルステップ動作は行いません。 また，DIR の SSE フラグ = 0 の場合は，このフラグはセットされません（0 に固定）。
10	SB	割り込み処理ルーチンに分岐する際に，このフラグの内容（初期値：0）が SS フラグに転送されます。 したがって，このフラグをセット（1）しておくことにより，割り込み処理ルーチンでのシングルステップ動作が可能となります。
7	NP	ノンマスカブル割り込み（NMI）処理中であることを示します。NMI 要求が受け付けられるとセット（1）され，多重割り込みを禁止します。 0：NMI 処理中でない。 1：NMI 処理中である。
6	EP	ソフトウエア例外処理中であることを示します。ソフトウエア例外の発生でセット（1）されます。 なお，このビットがセットされてもマスカブル割り込み要求は受け付けます。 0：ソフトウエア例外処理中でない。 1：ソフトウエア例外処理中である。
5	ID	マスカブル割り込み要求を受け付ける状態かどうかを示します。 0：割り込み許可（EI） 1：割り込み禁止（DI）

ビット位置	フラグ名	意味
4	SAT 注	飽和演算命令の演算結果がオーバフローし，演算結果が飽和していることを示します。累積フラグのため，飽和演算命令で演算結果が飽和すると，セット（1）され，以降の命令の演算結果が飽和しなくてもクリア（0）されません。クリア（0）する場合は，LDSR 命令を使用して PSW にデータをロードします。なお，一般の算術演算命令ではセット（1）もクリア（0）も行われません。 0：飽和していない。 1：飽和している。
3	CY	演算結果に，キャリー，またはボローが発生したかどうかを示します。 0：キャリー，またはボローは発生していない。 1：キャリー，またはボローが発生した。
2	OV 注	演算中にオーバフローが発生したかどうかを示します。 0：オーバフローは発生していない。 1：オーバフローが発生した。
1	S 注	演算結果が負かどうかを示します。 0：演算結果は正または 0 であった。 1：演算結果は負であった。
0	Z	演算結果が 0 かどうかを示します。 0：演算結果は 0 でなかった。 1：演算結果は 0 であった。

注 飽和演算時の OV フラグと S フラグの内容で飽和処理した演算結果が決まります。また，飽和演算時に OV フラグがセット（1）された場合に限り，SAT フラグがセット（1）されます。

演算結果の状態	フラグの状態			飽和処理をした演算結果
	SAT	OV	S	
正の最大値を越えた	1	1	0	7FFFFFFFH
負の最大値を越えた	1	1	1	80000000H
正（最大値を越えない）	演算前の値を保持	0	0	演算結果そのもの
負（最大値を越えない）	演算前の値を保持	0	1	演算結果そのもの

（e）CALLT 実行時状態退避レジスタ：CTPC，CTPSW【V850ES，V850E1，V850E2】

CALLT 実行時状態退避レジスタには，CTPC と CTPSW があります。

これらのレジスタには，CALLT 命令が実行された際，PC の内容が CTPC に，PSW の内容が CTPSW に退避されます。

- CTPC

CALLT 命令の次の命令のアドレスが退避されます。

- CTPSW

CALLT 命令が実行された際の PSW の内容が退避されます。

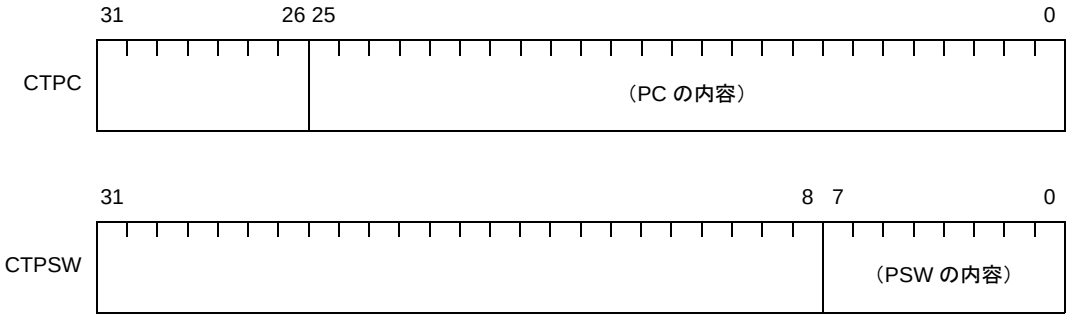
なお，CTPC，CTPSW の各ビットの意味は，CPU の種類（V850ES，V850E1，V850E2）により異なります。

- V850ES

CTPC は，ビット 25-0 が有効で，ビット 31-26 は将来の機能拡張のために予約されています（0 に固定）。

CTPSW は，ビット 7-0 が有効で，ビット 31-8 は将来の機能拡張のために予約されています（0 に固定）。

図 4―25 CALLT 実行時状態退避レジスタ【V850ES】



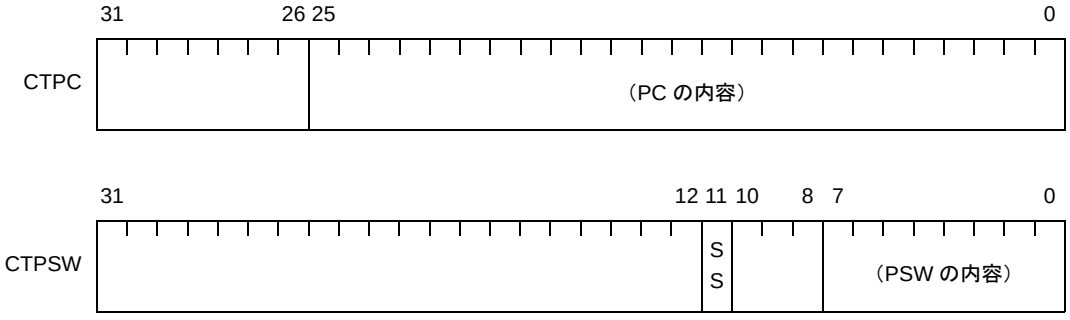
- V850E1

CTPC は，ビット 25-0 が有効で，ビット 31-26 は将来の機能拡張のために予約されています（0 に固定）。

CTPSW は，ビット 11，7-0 が有効で，ビット 31-12，10-8 は将来の機能拡張のために予約されています（0 に固定）。

なお，CTPSW のビット 11 には，PSW の SS フラグが退避されます。

図 4―26 CALLT 実行時状態退避レジスタ【V850E1】



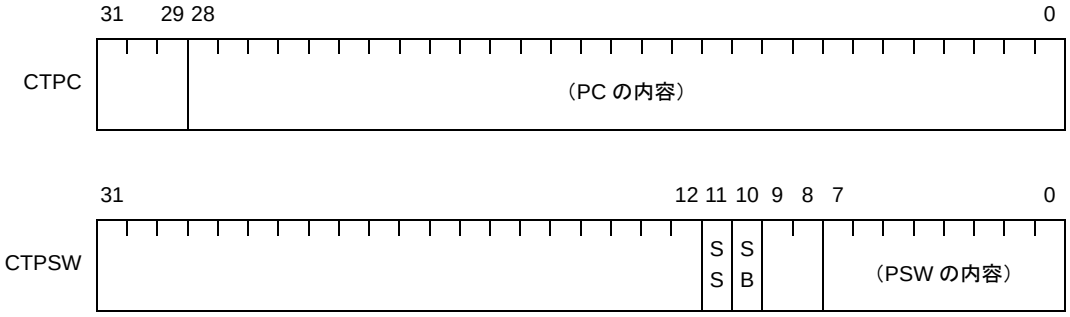
- V850E2

CTPC は，ビット 28-0 が有効で，ビット 31-29 は将来の機能拡張のために予約されています（0 に固定）。

CTPSW は，ビット 11-10，7-0 が有効で，ビット 31-12，9-8 は将来の機能拡張のために予約されています（0 に固定）。

なお，CTPSW のビット 11 には，PSW の SS フラグが，CTPSW のビット 10 には，PSW の SB フラグが退避されます。

図 4―27 CALLT 実行時状態退避レジスタ【V850E2】



(f) 例外／デバッグ・トラップ時状態退避レジスタ：DBPC，DBPSW【V850ES，V850E1，V850E2】

例外／デバッグ・トラップ時状態退避レジスタには，DBPC と DBPSW があります。

これらのレジスタには，例外トラップ，デバッグ・トラップ，デバッグ・ブレークが発生した際，またはシングルステップ動作が実行された際，PC の内容が DBPC に，PSW の内容が DBPSW に退避されます。

このレジスタは，ユーザ・モード時（DIR.DM フラグ = 0）は不定値となります。

- DBPC

DBPC には，以下に示した内容が退避されます。

表 4―31 DBPC に退避される内容

退避要因	DBPC に退避される内容
例外トラップの発生	例外トラップの発生要因となった命令の次の命令のアドレス
デバッグ・トラップ注 1 の発生	デバッグ・トラップの発生要因となった命令の次の命令のアドレス
デバッグ・ブレークの発生注 2 - 実行系トラップ - ミス・アライン・アクセス例外 - アライメント・エラー例外	ブレークの発生要因となった命令のアドレス
デバッグ・ブレークの発生注 2 - アクセス系トラップ	ブレークの発生要因となった命令の次の命令のアドレス

退避要因	DBPC に退避される内容
シングルステップ動作の実行注 2	次に実行される命令（デバッグ・モニタ・ルーチンからの復帰時に実行される命令）のアドレス

注 1. V850E1 のタイプ C では“デバッグ・・トラップ”をサポートしていません。

2. V850ES では，“デバッグ・ブレークの発生”，“シングルステップ動作の実行”をサポートしていません。

- DBPSW

DBPSW には，例外トラップ，デバッグ・トラップ，デバッグ・ブレークが発生した際，またはシングルステップ動作が実行された際の PSW の内容が退避されます。

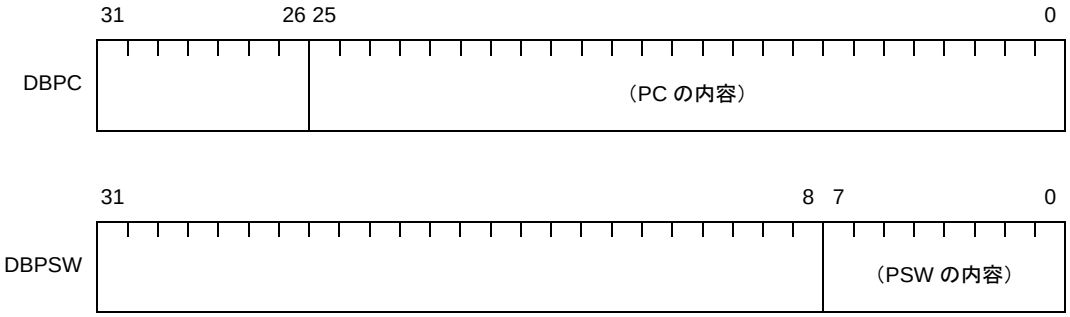
なお，DBPC，DBPSW の各ビットの意味は，CPU の種類（V850ES，V850E1，V850E2）により異なります。

- V850ES

DBPC は，ビット 25-0 が有効で，ビット 31-26 は将来の機能拡張のために予約されています（0 に固定）。

DBPSW は，ビット 7-0 が有効で，ビット 31-8 は将来の機能拡張のために予約されています（0 に固定）。

図 4―28 例外／デバッグ・トラップ時状態退避レジスタ【V850ES】



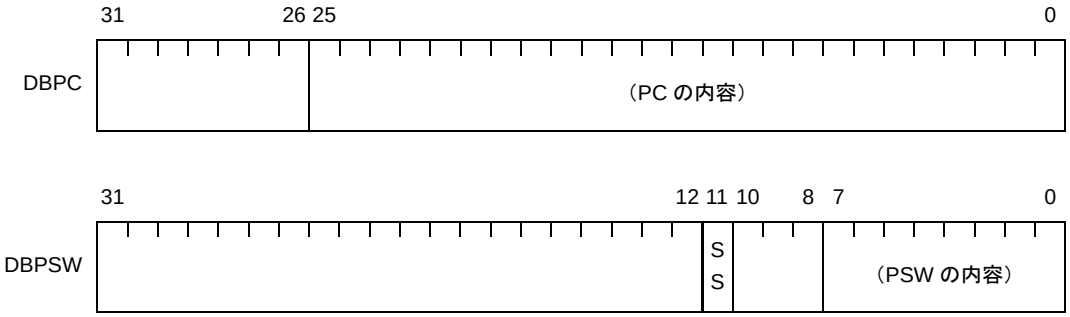
- V850E1

DBPC は，ビット 25-0 が有効で，ビット 31-26 は将来の機能拡張のために予約されています（0 に固定）。

DBPSW は，ビット 11，7-0 が有効で，ビット 31-12，10-8 は将来の機能拡張のために予約されています（0 に固定）。

なお，DBPSW のビット 11 には，PSW の SS フラグが退避されます。

図 4―29 例外／デバッグ・トラップ時状態退避レジスタ【V850E1】



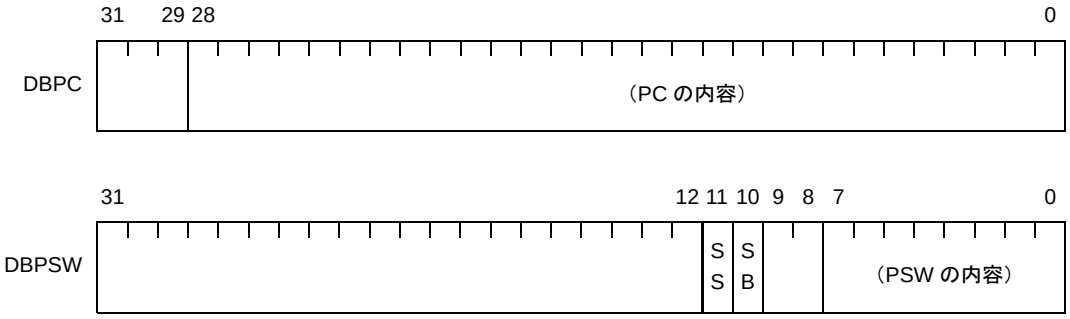
- V850E2

DBPC は，ビット 28-0 が有効で，ビット 31-26 は将来の機能拡張のために予約されています（0 に固定）。

DBPSW は，ビット 11-10，7-0 が有効で，ビット 31-12，9-8 は将来の機能拡張のために予約されています（0 に固定）。

なお，DBPSW のビット 11 には，PSW の SS フラグが，DBPSW のビット 10 には，PSW の SB フラグが退避されます。

図 4―30 例外／デバッグ・トラップ時状態退避レジスタ【V850E2】



（g）CALLT ベース・ポインタ：CTBP【V850ES，V850E1，V850E2】

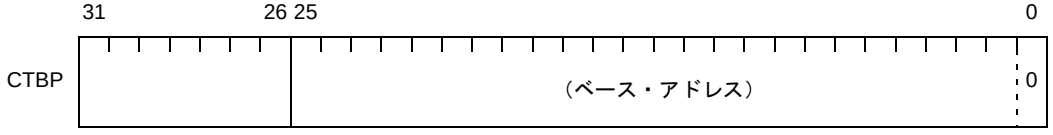
テーブル・アドレスの指定，ターゲット・アドレスの生成に使用されます（ビット 0 は 0 に固定）。

なお，CTBP の各ビットの意味は，CPU の種類（V850ES，V850E1，V850E2）により異なります。

- V850ES，V850E1

ビット 25-0 が有効で，ビット 31-26 は将来の機能拡張のために予約されています（0 に固定）。

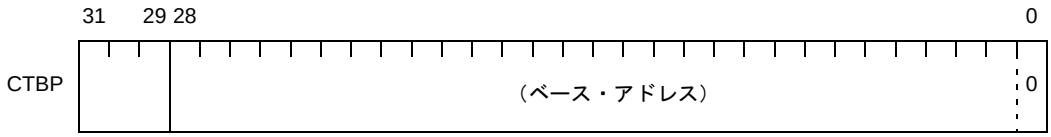
図 4―31 CALLT ベース・ポインタ【V850ES，V850E1】



- V850E2

ビット 28-0 が有効で，ビット 31-29 は将来の機能拡張のために予約されています（0 に固定）。

図 4―32 CALLT ベース・ポインタ【V850E2】



（h）デバッグ・インタフェース・レジスタ：DIR【V850ES，V850E1，V850E2】

デバッグ機能の制御や状態を示します。

なお，DIR の各ビットの意味は，CPU の種類（V850ES，V850E1，V850E2）により異なります。

- V850ES

ビット 0 が有効で，ビット 31-1 は将来の機能拡張のために予約されています（0 に固定）。

STSR 命令を使用してこのレジスタの内容を汎用レジスタに退避することにより，このレジスタの内容を読み込むことができます。なお，このレジスタへの書き込みはできません。

図 4―33 デバッグ・インタフェース・レジスタ【V850ES】

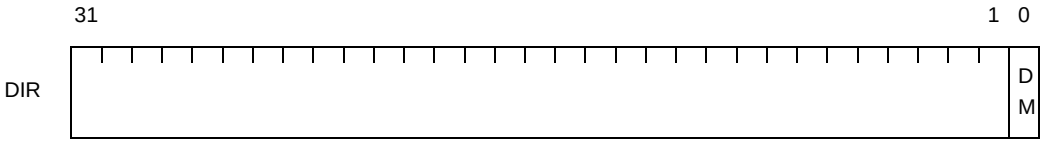


表 4―32 デバッグ・インタフェース・レジスタ【V850ES】

ビット位置	フラグ名	意味
0	DM	例外トラップ時，および DBTRAP 命令時にセット（1）され， DBRET 命令によりクリア（0）されます。 0: 通常モード 1：デバッグ・モード

- V850E1

ビット 14-8，6-0 が有効で，ビット 31-15，7 は将来の機能拡張のために予約されています（0 に固定）。

LDSR 命令を使用してこのレジスタの各ビットの内容を変更した場合には，LDSR 命令実行終了直後から変更内容が有効となります。

このレジスタへの書き込み（ビット 3，1 を除く）は，デバッグ・モード時（DM フラグ = 1）に限られます。

このレジスタの読み込みは常時可能ですが，ユーザ・モード時（DM フラグ = 0）にはビット 14-8，6-4，2-1 が不定値となります。

注意 V850E1 のタイプ A，B に限りアクセス可能です。その他の製品ではアクセス禁止です。

図 4―34 デバッグ・インタフェース・レジスタ【V850E1】

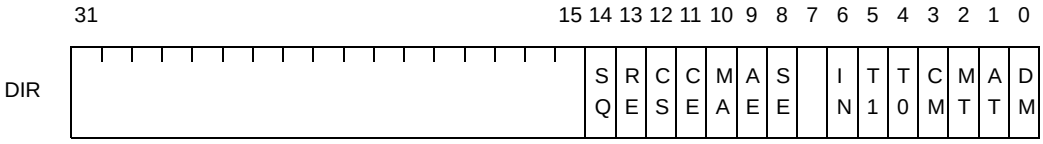


表 4―33 デバッグ・インタフェース・レジスタ【V850E1】

ビット位置	フラグ名	意味
14	SQ 注 1，2	シーケンシャル・ブレーク・モード（チャネル 0 → 1 の順にブレークが発生した場合にブレーク）を設定します。 0：通常ブレーク・モード 1：シーケンシャル・ブレーク・モード
13	RE 注 1，2	レインジ・ブレーク・モード（チャネル 0，1 に同時にブレークが発生した場合にブレーク）を設定します。 0：通常ブレーク・モード 1：レインジ・ブレーク・モード
12	CS 注 2	ブレーク・レジスタ・バンクを設定します。 0：バンク 0 レジスタ（チャネル 0 制御レジスタ）を選択 1：バンク 1 レジスタ（チャネル 1 制御レジスタ）を選択
11	CE	COMBO 割り込みの許可／禁止を設定します。 0：COMBO 割り込み禁止 1：COMBO 割り込み許可
10	MA	ミス・アライン・アクセス例外の検出の許可／禁止を設定します。 0：ミス・アライン・アクセス例外の検出を禁止 1：ミス・アライン・アクセス例外の検出を許可
9	AE	アライメント・エラー例外の検出の許可／禁止を設定します。 0：アライメント・エラー例外の検出を禁止 1：アライメント・エラー例外の検出を許可

ビット位置	フラグ名	意味
8	SE	PSW の SS フラグへの書き込みの許可／禁止を設定します。 0：SS フラグへの書き込みを禁止（SS フラグを 0 に固定） 1：SS フラグへの書き込みを許可
6	IN 注 3	デバッグ機能のリセットにより，セット（1）されます。 リセット後はセット（1）されるため，必ずクリア（0）してください。このビットがセット（1）されていると，SQ，RE，CS フラグへの書き込みができません。また，T1，T0 フラグが動作しません。
5	T1 注 3，4	チャネル 1 のブレークが発生するとセット（1）されます。 0 を設定するとクリア（0）されます。注 4
4	T0 注 3，4	チャネル 0 のブレークが発生するとセット（1）されます。 0 を設定するとクリア（0）されます。注 4
3	CM 注 5	COMBO 割り込みルーチン，またデバッグ・モニタ・ルーチン 2 に移行するとセット（1）されます。 このビットへの書き込みはできません。
2	MT 注 3	ミス・アライン・アクセス例外が検出されるとセット（1）されます。 0 を設定するとクリア（0）されます。注 6
1	AT 注 3	アライメント・エラー例外が検出されるとセット（1）されます。 0 を設定するとクリア（0）されます。注 6
0	DM 注 5	デバッグ・モードに移行するとセット（1）され，ユーザ・モードに移行するとクリア（0）されます。 このビットへの書き込みはできません。

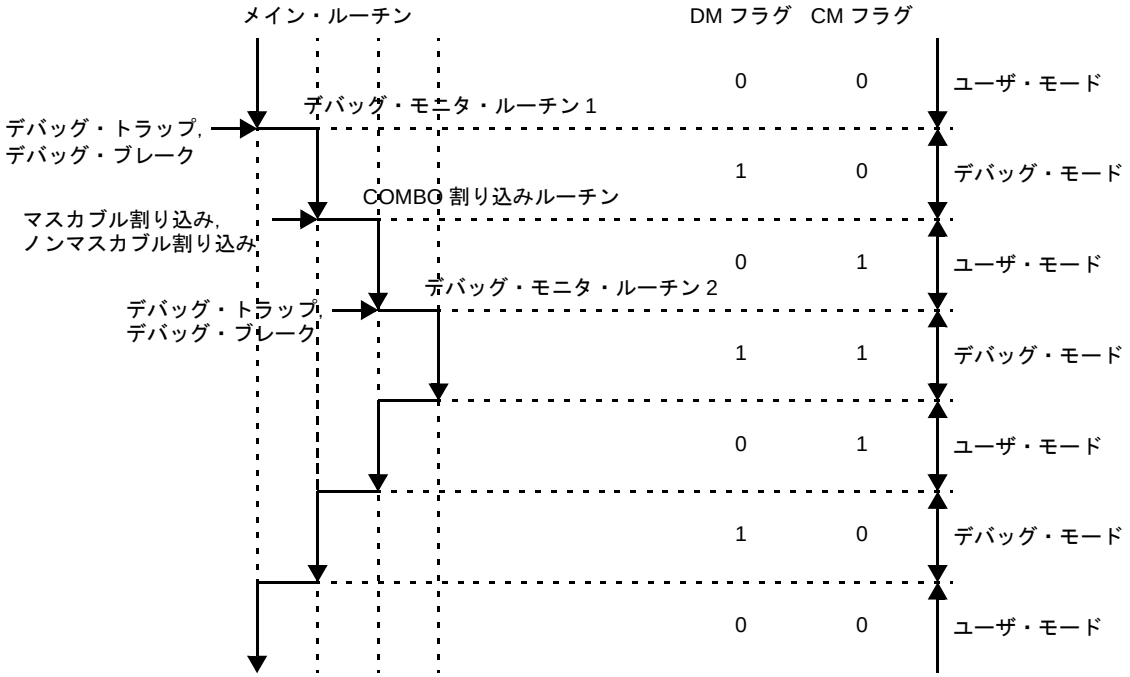
注 1. SQ，RE フラグは，常にどちらか一方をセット（1），または両フラグともクリア（0）された状態としてください。両フラグともセット（1）された場合の状態では動作が保証されません。

2. IN フラグがセット（1）されていると，SQ，RE，CS フラグへの書き込みはできません。また，IN フラグをセット（1）すると各フラグは自動的にクリア（0）されます。

3. IN，T1，T0，MT，AT フラグはセット（1）されると自動的にクリア（0）されません（LDSR 命令を用いてクリア（0）します）。

4. IN フラグがセット（1）されていると，T1，T0 フラグは動作しません（ブレークが発生してもセット（1）されません）。また，IN フラグをセット（1）すると自動的にクリア（0）されます。

5. DM，CM フラグは，次のように変化します。



6. T1，T0，MT，AT フラグは，ユーザ・プログラムによる任意のセット（1）はできません。

- V850E2

ビット 30-28，22-20，16，14-12，10-4，2-0 が有効で，ビット 31，27-23，19-17，15，11，3 は将来の機能拡張のために予約されています（0 に固定）。

LDSR 命令を使用してこのレジスタの各ビットの内容を変更した場合には，LDSR 命令実行終了直後から変更内容が有効となります。

このレジスタへの書き込みは，デバッグ・モード時には書き込み禁止のビット 31，27-23，19-17，15，および，リード・オンリーのビット 3，0 を除いたビットに対して，ユーザ・モード時にはビット 22 に対して可能となります。

このレジスタの読み込みは常時可能ですが，ユーザ・モード時（DM フラグ = 0）にはビット 22 以外が不定値となります。

図 4―35　デバッグ・インタフェース・レジスタ【V850E2】

	31	30	29	28	27	23	22	21	20	19	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DIR		SQ1	RE1	CS1			CSL	BT3	BT2			STT		SQ0	RE0	CS0		MAE	AEE	SSE	EXT	INI	BT1	BT0		MT	AT	DM

表 4―34　デバッグ・インタフェース・レジスタ【V850E2】

ビット位置	フラグ名	意味
30	SQ1 注1	チャネル 2，3 に対するシーケンシャル・ブレーク・モード（チャネル 2→3 の順にブレークが発生した場合いブレーク）を設定します。 0：通常ブレーク・モード 1：シーケンシャル・ブレーク・モード
29	RE1 注1	チャネル 2，3 に対するレインジ・ブレーク・モード（チャネル 2，3 に同時にブレークが発生した場合だけブレーク）を設定します。 0：通常ブレーク・モード 1：レインジ・ブレーク・モード
28	CS1 注1	チャネル 2，3 の制御レジスタ（BPCn，BPAVn，BPAMn，BPDVn，BPDMn）への設定を許可します。 0：チャネル 2 の制御レジスタ（BPC2，BPxx2）への設定を許可 1：チャネル 3 の制御レジスタ（BPC3，BPxx3）への設定を許可
22	CSL	各チャネルの制御レジスタへの設定を許可します。 0：チャネル 0，1 1：チャネル 2，3
21	BT3 注2	チャネル 3 のブレークが発生するとセット（1）されます。
20	BT2 注2	チャネル 2 のブレークが発生するとセット（1）されます。
16	STT	デバッグ・トラップ実行時にセット（1）されます。 このビットは，セット（1）されると自動的にクリア（0）されません。LDSR 命令によってのみクリア（0）できます。
14	SQ0 注3	チャネル 0，1 に対するシーケンシャル・ブレーク・モード（チャネル 0→1 の順にブレークが発生した場合にブレーク）を設定します。 0：通常ブレーク・モード 1：シーケンシャル・ブレーク・モード
13	RE0 注3	チャネル 0，1 に対するレインジ・ブレーク・モード（チャネル 0，1 に同時にブレークが発生した場合だけブレーク）を設定します。 0：通常ブレーク・モード 1：レインジ・ブレーク・モード
12	CS0 注3	チャネル 0，1 の制御レジスタ（BPCｎ，BPAVｎ，BPAMｎ，BPDVｎ，BPDMｎ）への設定を許可します。 0：チャネル 0 の制御レジスタ（BPC0，BPxx0）への設定を許可 1：チャネル 1 の制御レジスタ（BPC1，BPxx1）への設定を許可

ビット位置	フラグ名	意味
10	MAE	ミス・アライン・アクセス例外の検出を禁止／許可を設定します。 0：ミス・アライン・アクセス例外の検出を禁止 1：ミス・アライン・アクセス例外の検出を許可
9	AEE	アライメント・エラー例外の検出を禁止／許可を設定します。 0：アライメント・エラー例外の検出を禁止 1：アライメント・エラー例外の検出を許可
8	SSE	PSW の SS フラグへの書き込みを禁止／許可を設定します。 0：SS フラグへの書き込みを禁止 1：SS フラグへの書き込みを許可
7	EXT	拡張デバッグ機能の有効／無効を設定します。 0: 無効 1：有効
6	INI **注 4**	デバッグ機能のリセットにより，セット（1）されます。 リセット後はセット（1）されるため，必ずクリア（0）してください。このビットがセット（1）されていると，SQｎ，REｎ，CSｎ フラグへの書き込みができません。また，ビット 3-0 が動作しません。
5	BT1 **注 5**	チャネル 1 のブレークが発生するとセット（1）されます。
4	BT0 **注 5**	チャネル 0 のブレークが発生するとセット（1）されます。
2	MT 注 4	ミス・アライン・アクセス例外が検出されるとセット（1）されます。
1	AT **注 4**	アライメント・エラー例外が検出されるとセット（1）されます。
0	DM **注 6**	デバッグ・モードに移行するとセット（1）されます。

注 1. INI フラグがセット（1）されていると，SQ1，RE1，CS1 フラグへの書き込みはできません。また，INI フラグをセット（1）すると，SQ1，RE1，CS1 フラグは自動的にクリア（0）されます。

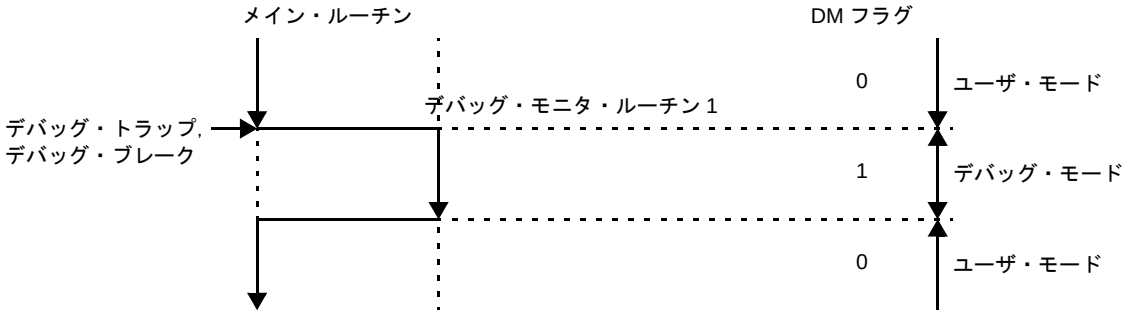
2. BT2，BT3 フラグは，INI フラグがセット（1）されていると動作しません（ブレークが発生してもせっと（1）されません）。また，セット（1）されると，LDSR 命令によって 0 を設定するか，INI フラグをセット（1）するまでクリア（0）されません。

3. INI がセット（1）されていると，SQ0，RE0，CS0 フラグへの書き込みはできません。また，INI フラグをセット（1）すると，SQ0，RE0，CS0 フラグは自動的にクリア（0）されます。

4. INI，MT，AT フラグは，セット（1）されると自動的にクリア（0）されません。LDSR 命令によってのみクリア（0）できます。

5. BT0，BT1 フラグは，INI フラグがセット（1）されていると，動作しません（ブレークが発生してもセット（1）されません）。また，セット（1）されると，LDSR 命令によって 0 を設定するか，INI フラグをセット（1）するまでクリア（0）されません。

6. DM フラグは，次のように変化します。



(i) ブレークポイント制御レジスタ：BPCn【V850E1，V850E2】

デバッグ機能の制御や状態を示します。

なお，BPC n の各ビットの意味は，CPU の種類（V850E1，V850E2）により異なります。

- V850E1

V850E1 のブレークポイント制御レジスタには，BPC0 と BPC1 があり，DIR.CS フラグの設定により，どちらか一方のレジスタが有効になります。

BPCn は，ビット 23-15，11-7，4-0 が有効で，ビット 31-24，14-12，6-5 は将来の機能拡張のために予約されています（0 に固定）。

LDSR 命令を使用してこのレジスタの各ビットの内容を変更した場合には，LDSR 命令実行終了直後から変更内容が有効となります（FE フラグをセット（1）した場合，有効となるタイミングが遅れますが，DBRET 命令を実行したあとには，必ず設定が反映されます）。

このレジスタへの書き込みは，デバッグ・モード時（DIR.DM フラグ = 1）に限られます。

このレジスタの読み込みは常時可能ですが，ユーザ・モード時（DIR.DM フラグ = 0）にはビット 0 が 0，ビット 23-15，11-7，4-1 が不定値となります。

注意　V850E1 のタイプ A，B に限りアクセス可能です。その他の製品ではアクセス禁止です。

図 4―36　ブレークポイント制御レジスタ【V850E1】

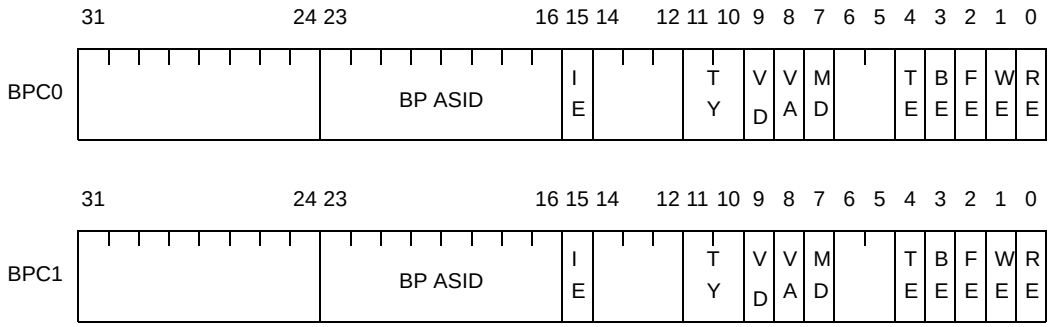


表 4―35 ブレークポイント制御レジスタ【V850E1】

ビット位置	フラグ名	意味
23-16	BP ASID	ブレークを発生させるプログラム ID を設定します（IE フラグ = 1 の場合のみ有効）。
15	IE	BP ASID フラグと ASID に設定されているプログラム ID の比較を設定します。 - 0：比較しない - 1：比較する
11-10	TY	ブレークを検出するアクセスの種類を設定します。 なお，実行系トラップの場合は，このレジスタの内容は無視されます。 0，0：すべてのデータ・タイプによるアクセス 0，1: バイト・アクセス（ビット操作を含む） 1，0: ハーフワード・アクセス 1，1：ワード・アクセス
9	VD	データ・コンパレータの一致条件を設定します。 0：一致でブレーク 1：不一致でブレーク
8	VA	アドレス・コンパレータの一致条件を設定します。 0：一致でブレーク 1：不一致でブレーク
7	MD	データ・コンパレータの動作を設定します。 0：データが条件に一致したときにブレーク 1：VD フラグや BPDVx，BPDMx の設定の設定に関係なくデータの一致判定（データ・コンパレータは無視）
4	TE 注 1	トリガ出力の許可／禁止を設定します。 0：トリガ出力禁止 1：トリガ出力許可（チャネル 0，1 のブレーク発生時に対応トリガを出力）
3	BE 注 1	チャネル 0，1 のブレークを CPU へ通知する／しないを設定します。 0：通知しない 1：通知する（ブレークする）
2	FE	命令実行アドレス一致によるブレーク／トリガの許可／禁止を設定します。 0：ブレーク／トリガ禁止 1：ブレーク／トリガ許可 注 2
1	WE	データ・ライト時のブレーク／トリガの許可／禁止を設定します。 0：ブレーク／トリガ禁止 1：ブレーク／トリガ許可 注 3

ビット位置	フラグ名	意味
0	RE	データ・リード時のブレーク／トリガの許可／禁止を設定します。 0：ブレーク／トリガ禁止 1：ブレーク／トリガ許可**注 3**

注 1. TE，BE フラグは V850E1 のタイプ B のみ設定可能です。その他の製品では，“0”に固定されます（BE フラグは 0 に固定されますが，CPU へのブレークの通知は行います）。

2. FE フラグをセット（1）した場合は，必ず WE，RE フラグをクリア（0）してください。

3. WE フラグ，または RE フラグをセット（1）した場合は，必ず FE フラグをクリア（0）してください。

- V850E2

V850E2 のブレークポイント制御レジスタには，BPC0，BPC1，BPC2，BPC3 があり，DIR.CSL，CS1，CS0 フラグの設定により，有効となるレジスタが選択されます。

BPCn は，ビット 26-15，11-7，4-0 が有効で，ビット 31-27，14-12，6-5 は将来の機能拡張のために予約されています（0 に固定）。

LDSR 命令を使用してこのレジスタの各ビットの内容を変更した場合には，LDSR 命令実行終了直後から変更内容が有効となります（FE フラグをセット（1）した場合，有効となるタイミングが遅れますが，DBRET 命令を実行したあとには，必ず設定が反映されます）。

ビット 31-27，14-12，6-5 は，必ずクリア（0）してください。いずれかのビットをセット（1）した場合の動作は保証しません。

FB2-FB0 フラグへの書き込みは，クリア（0）のみ可能です。これらのビットの値を更新する場合は，すべてのビットをクリア（0）してください。いずれかのビットをセット（1）した場合の動作は保証しません。

図 4―37 ブレークポイント制御レジスタ【V850E2】

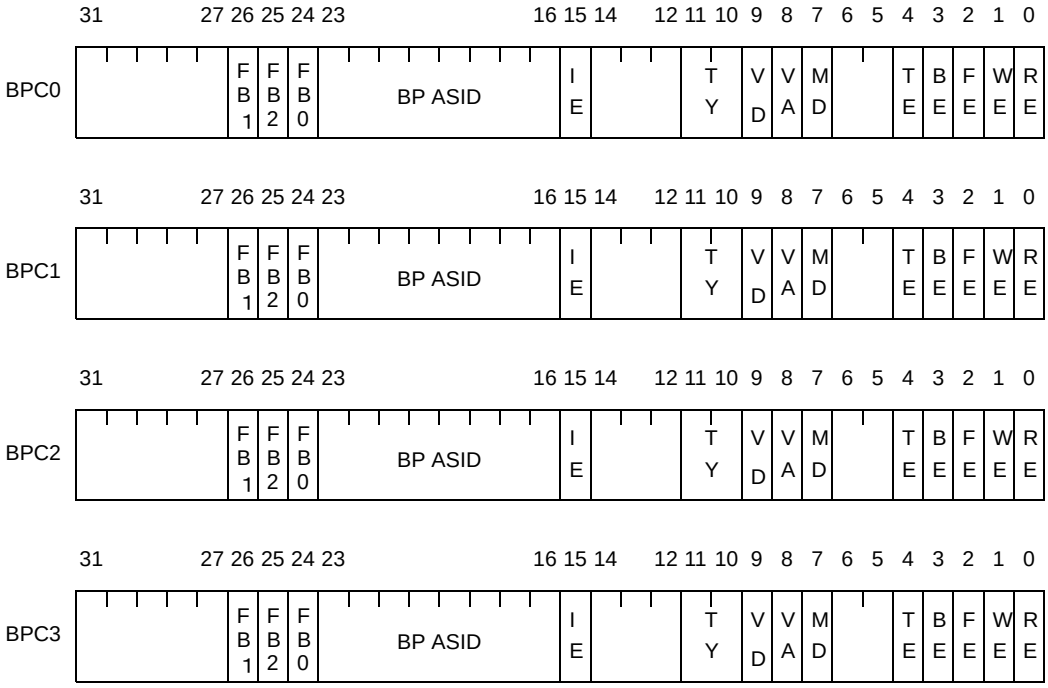


表 4―36 ブレークポイント制御レジスタ【V850E2】

ビット位置	フラグ名	意味
26-24	FBn	命令フェッチ・イベントによって発生したブレークの生類を示します。 0，0，0：ブレーク対象命令の実行中断によるブレーク 0，1，0：ブレーク対象命令とその 1 つ前の命令の実行中断によるブレーク 1，0，0：ブレーク対象命令とその 1 つ前と 2 つ前の命令の実行中断によるブレーク 0，0，1：ブレーク対象命令の実行完了によるブレーク 上記以外：将来の機能拡張のために予約
23-16	BP ASID	ブレークを発生させるプログラム ID を設定します（IE フラグ = 1 の場合のみ有効）。
15	IE	BP ASID フラグと ASID に設定されているプログラム ID の比較を設定します。 0：比較しない 1：比較する
11-10	TY	ブレークを検出するアクセスの種類を設定します。 なお，実行系トラップの場合は，このレジスタの内容は無視されます。 0，0：すべてのデータ・タイプによるアクセス 0，1: バイト・アクセス（ビット操作を含む） 1，0: ハーフワード・アクセス 1，1：ワード・アクセス

ビット位置	フラグ名	意味
9	VD	データ・コンパレータの一致条件を設定します。 0：一致でブレーク 1：不一致でブレーク
8	VA	アドレス・コンパレータの一致条件を設定します。 0：一致でブレーク 1：不一致でブレーク
7	MD	データ・コンパレータの動作を設定します。 0：データが条件に一致したときにブレーク 1：VD フラグや BPDVx，BPDMx の設定の設定に関係なくデータの一致判定（データ・コンパレータは無視）
4	TE	チャネル 3 のイベント発生時のトリガ出力の許可／禁止を設定します。 0：トリガ出力禁止 1：トリガ出力許可（対応トリガを出力）
3	BE	チャネル 0-3 のイベント発生時にブレークを CPU へ通知する／しないを設定します。 0：通知しない 1：通知する（ブレークする）
2	FE	命令フェッチ時のイベント動作を設定します。 0：イベント・マスク 1：イベント発生**注 1**
1	WE	データ・ライト時のイベント動作を設定します。 0：イベント・マスク 1：イベント発生**注 2**
0	RE	データ・リード時のイベント動作を設定します。 0：イベント・マスク 1：イベント発生**注 2**

注 1. FE フラグをセット（1）した場合は，必ず，WE，RE フラグをクリア（0）してください。

2. WE フラグ，または RE フラグをセット（1）した場合は，必ず FE フラグをクリア（0）してください。

(j) プログラム ID レジスタ：ASID【V850E1，V850E2】

現在実行中のプログラム ID を設定します。

ASID は，ビット 7-0 が有効で，ビット 31-8 は将来の機能拡張のために予約されています（0 に固定）。

プログラム ID は，同じアドレス領域の RAM に異なるプログラムをダウンロードして実行する際に，特定のプログラムの実行時のみデバッグ・モードへの移行を行う必要がある場合などにしようします。

BPCn.IE フラグをセット（1）している場合は，BPCn.BP ASID フラグと ASID に設定したプログラムが一致しないと。ブレーク条件が一致してもデバッグ・モードには移行しません。

注意 V850E1 のタイプ A，B，および，V850E2 に限りアクセス可能です。その他の製品ではアクセス禁止です。

図 4―38 プログラム ID レジスタ【V850E1，V850E2】

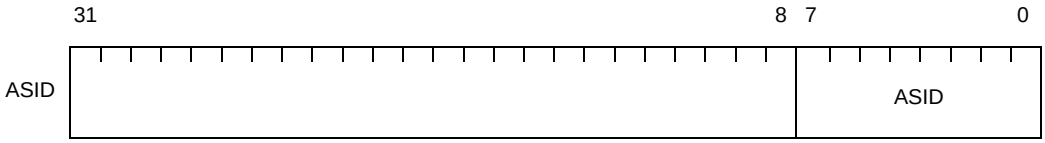


表 4―37 プログラム ID レジスタ【V850E1，V850E2】

ビット位置	フラグ名	意味
7-0	ASID	現在実行中のプログラム ID

（k）ブレークポイント・アドレス設定レジスタ：BPAVn【V850E1，V850E2】

アドレス・コンパレータで使用するブレークポイント・アドレスを設定します。

なお，BPAVｎ の各ビットの意味は，CPU の種類（V850E1，V850E2）により異なります。

- V850E1

V850E1 のブレークポイント・アドレス設定レジスタには，BPAV0 と BPAV1 があり，DIR.CS フラグの設定により，どちらか一方のレジスタが有効になります。

BPAVn は，ビット 27-0 が有効で，ビット 31-28 は将来の機能拡張のために予約されています（0 に固定）。

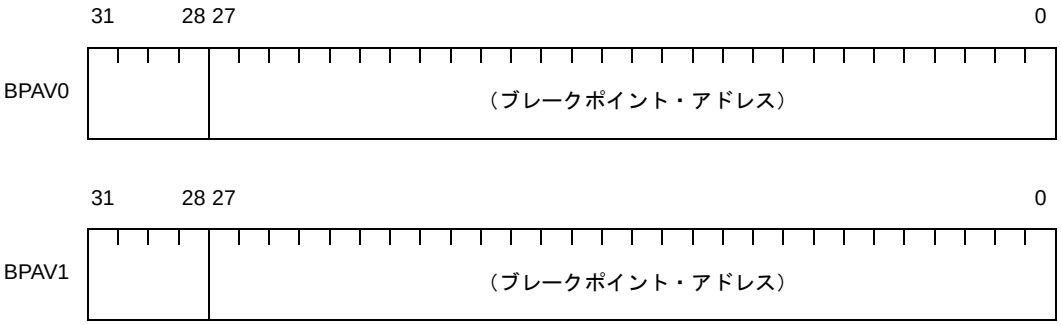
このレジスタへの書き込み／読み込みは，デバッグ・モード時（DIR.DM フラグ = 1）に限られます。

このレジスタの読み込みは常時可能ですが，ユーザ・モード時（DIR.DM フラグ = 0）には不定値となります。

なお，使用しない場合は，必ず各ビットをセット（1）してください。

注意 V850E1 のタイプ A，B に限りアクセス可能です。その他の製品ではアクセス禁止です。

図 4―39 ブレークポイント・アドレス設定レジスタ【V850E1】



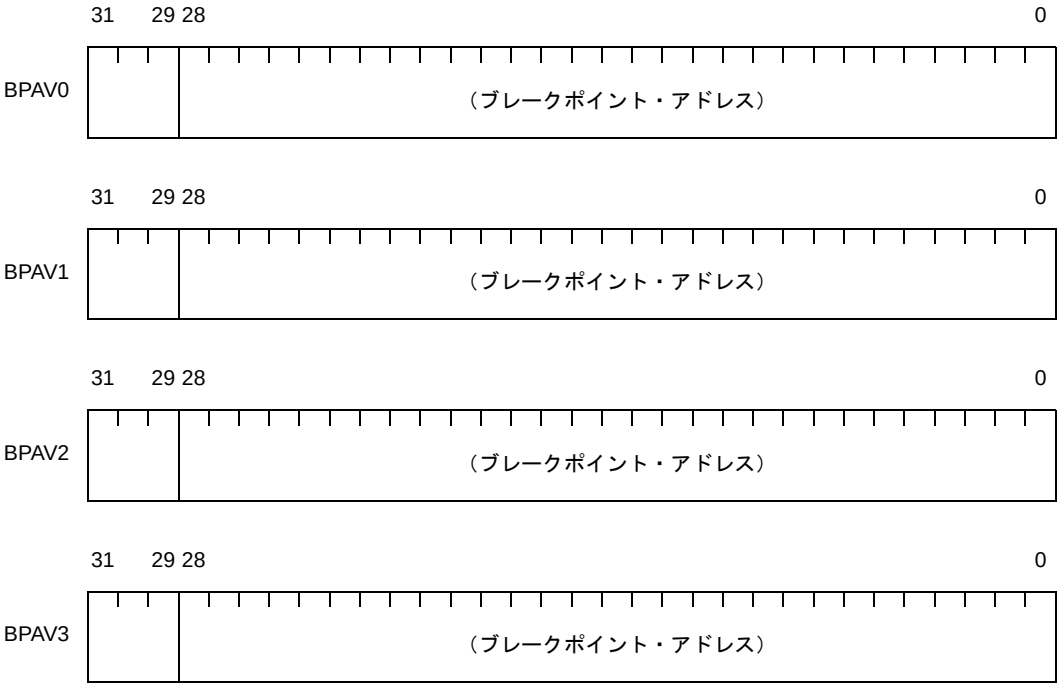
- V850E2

V850E2 のブレークポイント・アドレス設定レジスタには，BPAV0，BPAV1，BPAV2，BPAV3 があり，DIR.CSL，CS1，CS0 フラグの設定により，有効となるレジスタが選択されます。

BPAVn は，ビット 28-0 が有効で，ビット 31-29 は将来の機能拡張のために予約されています（0 に固定）。

なお，使用しない場合は，必ず各ビットをセット（1）してください。

図 4―40 ブレークポイント・アドレス設定レジスタ【V850E2】



(I) ブレークポイント・アドレス・マスク・レジスタ：BPAMn【V850E1，V850E2】

アドレスを比較する際のビット・マスク（1 でマスク）を設定します。

なお，BPAM n の各ビットの意味は，CPU の種類（V850E1，V850E2）により異なります。

- V850E1

V850E1 のブレークポイント・アドレス・マスク・レジスタには，BPAM0 と BPAM1 があり，DIR.CS フラグの設定により，どちらか一方のレジスタが有効になります。

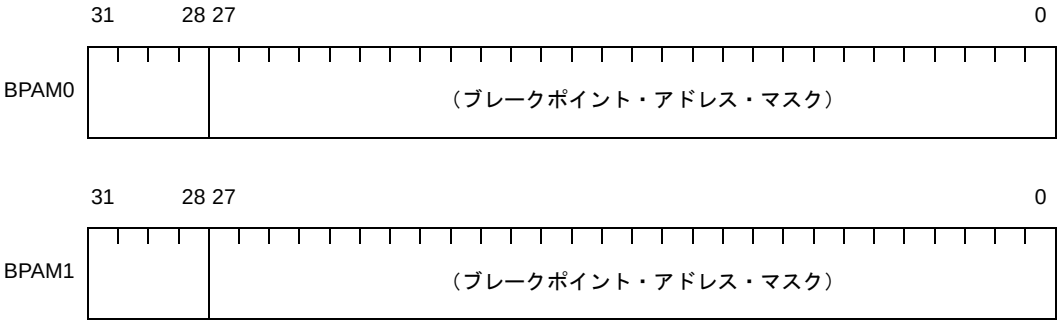
BPAMn は，ビット 27-0 が有効で，ビット 31-28 は将来の機能拡張のために予約されています（0 に固定）。

このレジスタへの書き込み／読み込みは，デバッグ・モード時（DIR.DM フラグ = 1）に限られます。

このレジスタの読み込みは常時可能ですが，ユーザ・モード時（DIR.DM フラグ = 0）には不定値となります。

なお，使用しない場合は，必ず各ビットをセット（1）してください。

図 4―41 ブレークポイント・アドレス・マスク・レジスタ【V850E1】



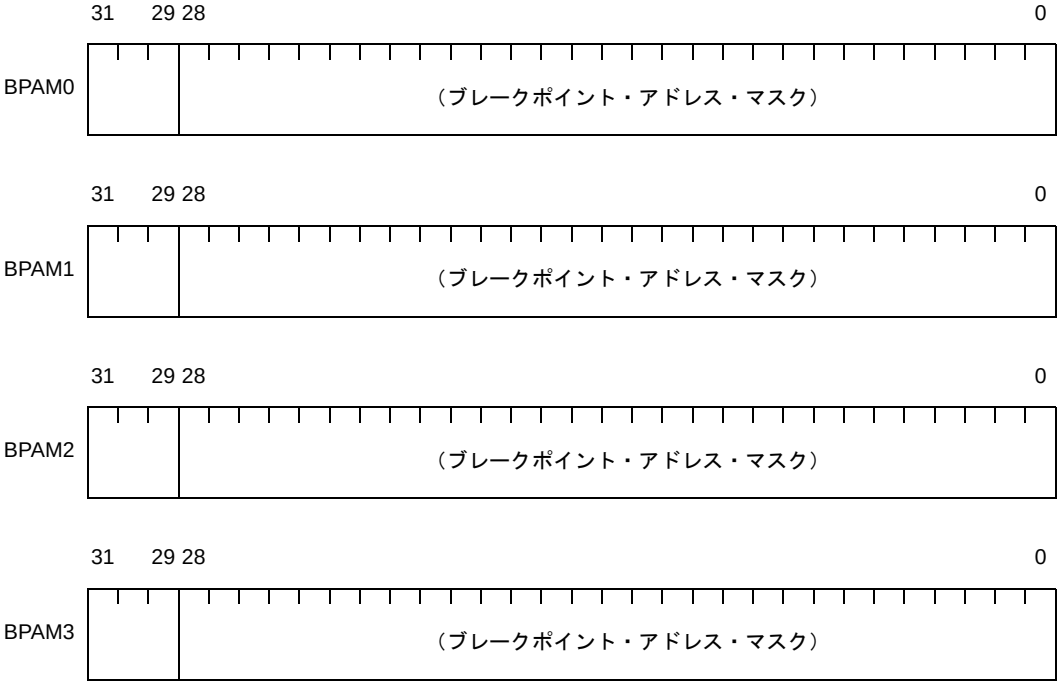
- V850E2

V850E2 のブレークポイント・アドレス・マスク・レジスタには，BPAM0，BPAM1，BPAM2，BPAM3 があり，DIR.CSL，CS1，CS0 フラグの設定により，有効となるレジスタが選択されます。

BPAMn は，ビット 28-0 が有効で，ビット 31-29 は将来の機能拡張のために予約されています（0 に固定）。

なお，使用しない場合は，必ず各ビットをセット（1）してください。

図 4―42 ブレークポイント・アドレス・マスク・レジスタ【V850E2】



（m）ブレークポイント・データ設定レジスタ：BPDVn【V850E1，V850E2】

データ・コンパレータで使用するブレークポイント・データを設定します。

なお，BPDVｎ の各ビットの意味は，CPU の種類（V850E1，V850E2）により異なります。

- V850E1

V850E1 のブレークポイント・データ設定レジスタには，BPDV0 と BPDV1 があり，DIR.CS フラグの設定により，どちらか一方のレジスタが有効になります。

このレジスタへの書き込み／読み込みは，デバッグ・モード時（DIR.DM フラグ = 1）に限られます。

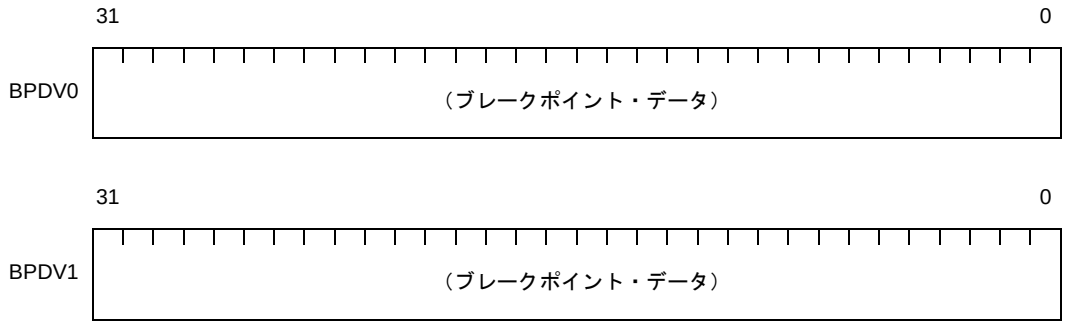
このレジスタの読み込みは常時可能ですが，ユーザ・モード時（DIR.DM フラグ = 0）には不定値となります。

なお，使用しない場合は，必ず各ビットをセット（1）してください。

注意 V850E1 のタイプ A，B に限りアクセス可能です。その他の製品ではアクセス禁止です。

備考 16 ビット命令の命令コードを設定する場合は，LSB 詰めで設定してください。32 ビット命令の命令コードを設定する場合は，リトル・エンディアン形式で設定してください。

図 4―43 ブレークポイント・データ設定レジスタ【V850E1】



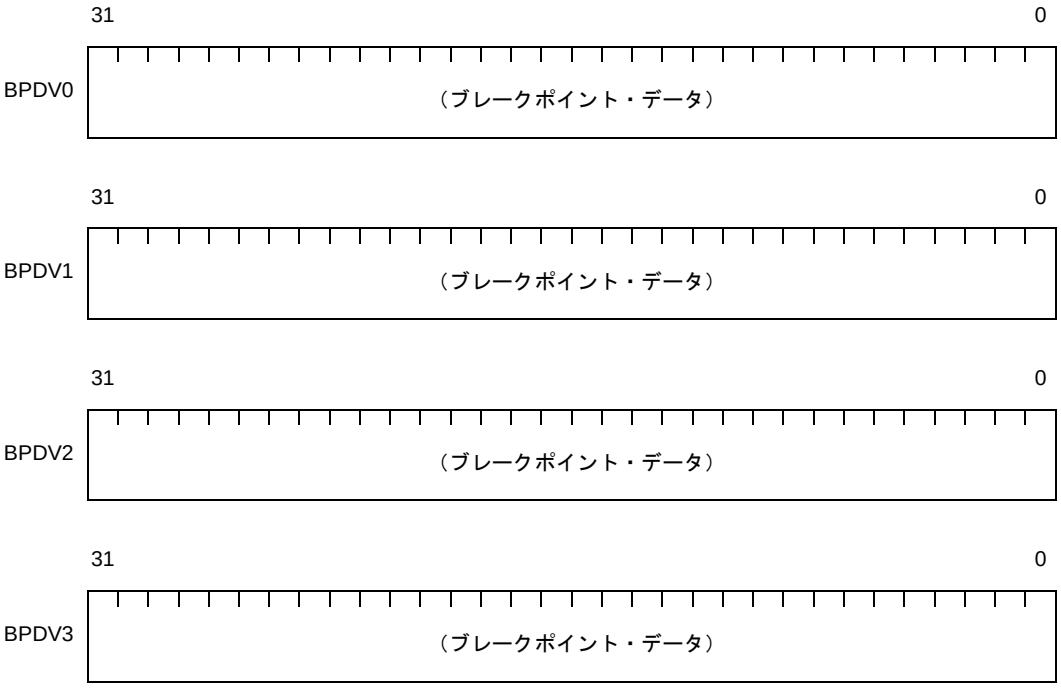
- V850E2

V850E2 のブレークポイント・データ設定レジスタには，BPDV0，BPDV1，BPDV2，BPDV3 があり，DIR.CSL，CS1，CS0 フラグの設定により，有効となるレジスタが選択されます。

なお，使用しない場合は，必ず各ビットをセット（1）してください。

備考 16 ビット命令の命令コードを設定する場合は，LSB 詰めで設定してください。32 ビット命令の命令コードを設定する場合は，リトル・エンディアン形式で設定してください。

図 4―44　ブレークポイント・データ設定レジスタ【V850E2】



（n）ブレークポイント・データ・マスク・レジスタ：BPDMn【V850E1，V850E2】

データを比較する際のビット・マスク（1 でマスク）を設定します。

なお，BPDMｎ の各ビットの意味は，CPU の種類（V850E1，V850E2）により異なります。

- V850E1

V850E1 のブレークポイント・データ・マスク・レジスタには，BPDM0 と BPDM1 があり，DIR.CS フラグの設定により，どちらか一方のレジスタが有効になります。

このレジスタへの書き込み／読み込みは，デバッグ・モード時（DIR.DM フラグ = 1）に限られます。

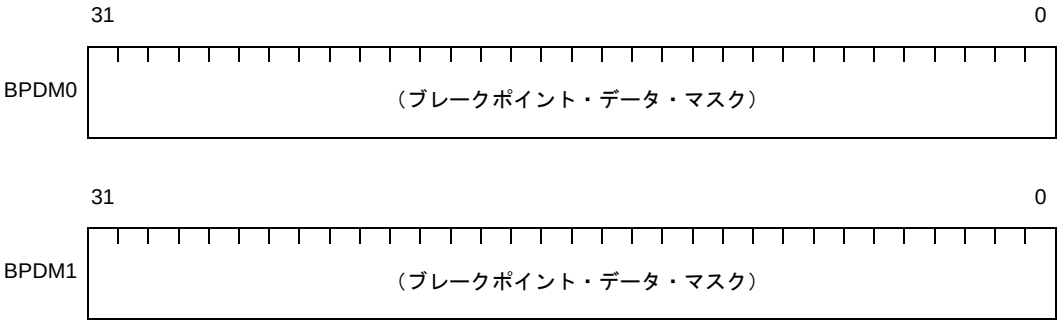
このレジスタの読み込みは常時可能ですが，ユーザ・モード時（DIR.DM フラグ = 0）には不定値となります。

なお，使用しない場合は，必ず各ビットをセット（1）してください。

また，ブレークを検出するデータ・アクセスの種類をバイト・アクセスに設定している場合（BPCn.TY フラグ = 0，1）はビット 31-8 を，ハーフワード・アクセスに設定している場合（BPCn.TY フラグ = 1，0）はビット 31-16 をセット（1）してください。

注意　V850E1 のタイプ A，B に限りアクセス可能です。その他の製品ではアクセス禁止です。

図 4―45 ブレークポイント・データ・マスク・レジスタ【V850E1】

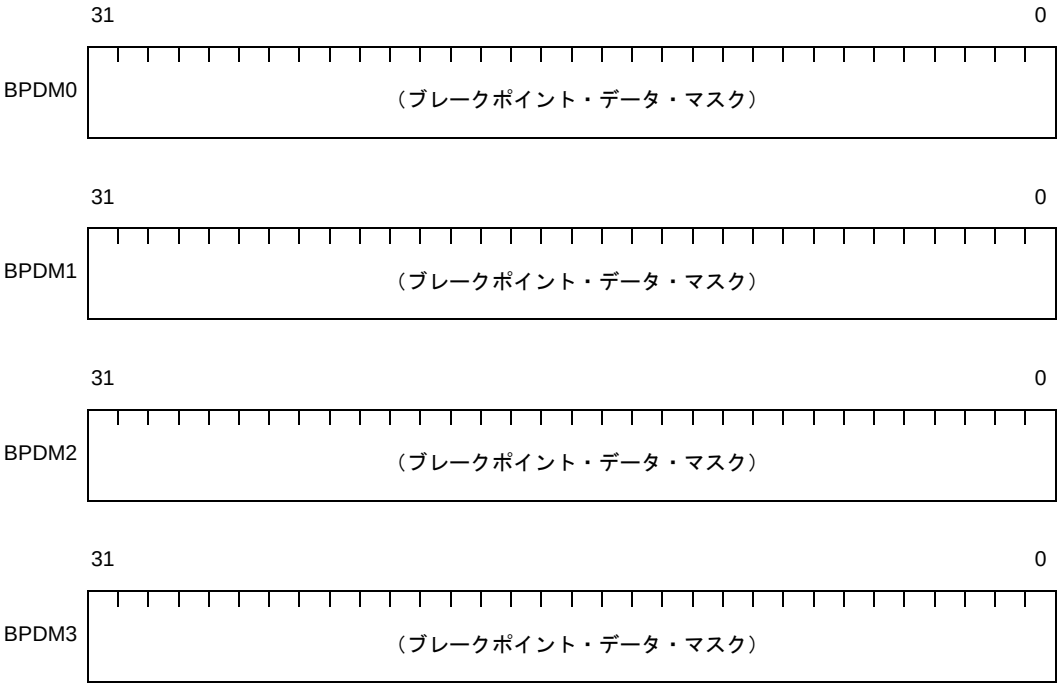


- V850E2

V850E2 のブレークポイント・データ・マスク・レジスタには，BPDM0，BPDM1，BPDM2，BPDM3 があり，DIR.CSL，CS1，CS0 フラグの設定により，有効となるレジスタが選択されます

なお，使用しない場合は，必ず各ビットをセット（1）してください。

図 4―46 ブレークポイント・データ・マスク・レジスタ【V850E2】



4. 5. 3　アドレッシング

アドレス生成には，分岐を伴う命令が使用する命令アドレス，データをアクセスする命令が使用するオペランド・アドレスの 2 種類があります。

(1) 命令アドレス

命令アドレスは，プログラム・カウンタ（PC）の内容によって決定され，実行した命令のバイト数に応じて自動的にインクリメント（+2）されます。また，分岐命令を実行する際には，次に示すアドレッシングにより，分岐先アドレスを PC にセットします。

(a) レラティブ・アドレッシング（PC 相対）

プログラム・カウンタ（PC）に，命令コードの符号付き 9，22，または 32 ビット・データ（ディスプレースメント：disp）を加算します。このとき，ディスプレースメントは，2 の補数データとして扱われ，それぞれ，ビット 8，21，31 が符号ビット（S）となります。

JR disp22 命令，JARL disp22, reg2 命令，JR disp32 命令，JARL disp32, reg1 命令，Bcnd disp9 命令が本アドレッシングの対象となります。

図 4―47　レラティブ・アドレッシング（JR disp22 ／ JARL disp22, reg2）【V850 マイクロコントローラ】

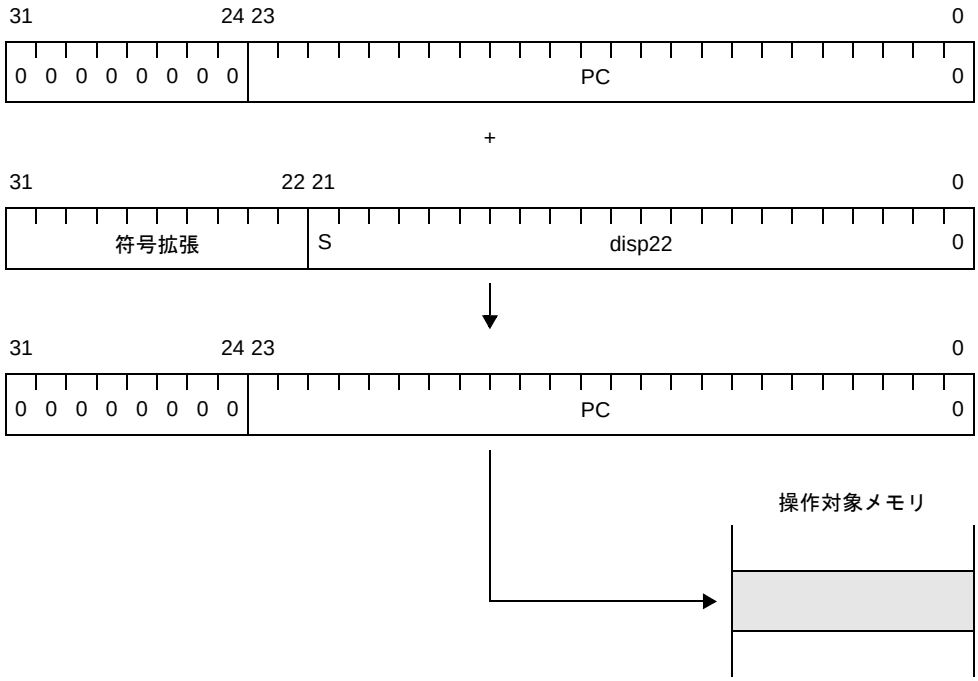


図 4―48 レラティブ・アドレッシング（JR disp22 ／ JARL disp22, reg2）【V850ES，V850E1】

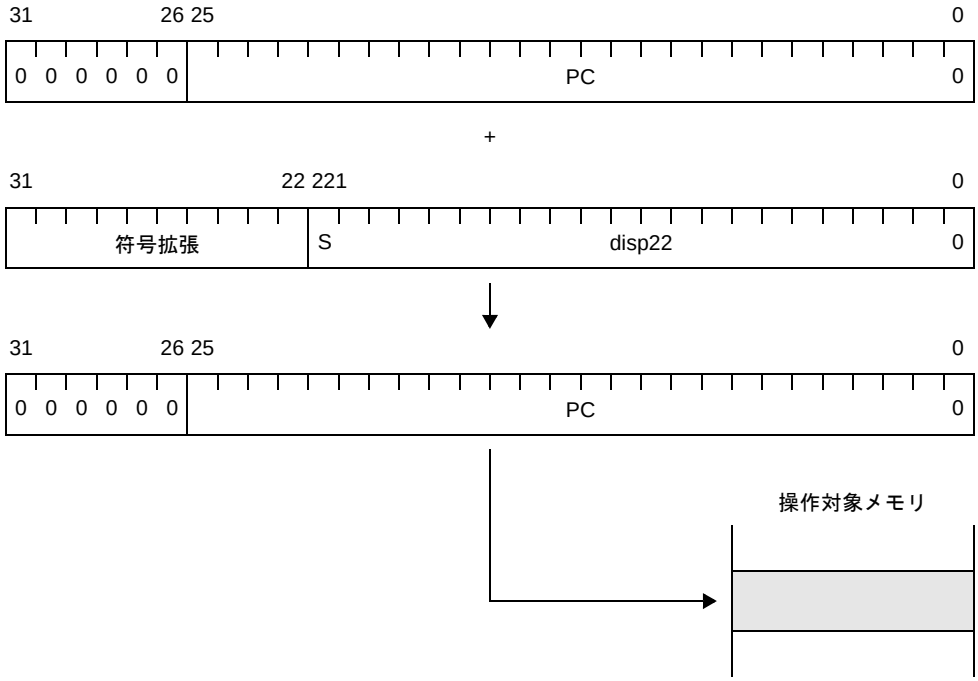


図 4―49 レラティブ・アドレッシング（JR disp22 ／ JARL disp22, reg2）【V850E2】

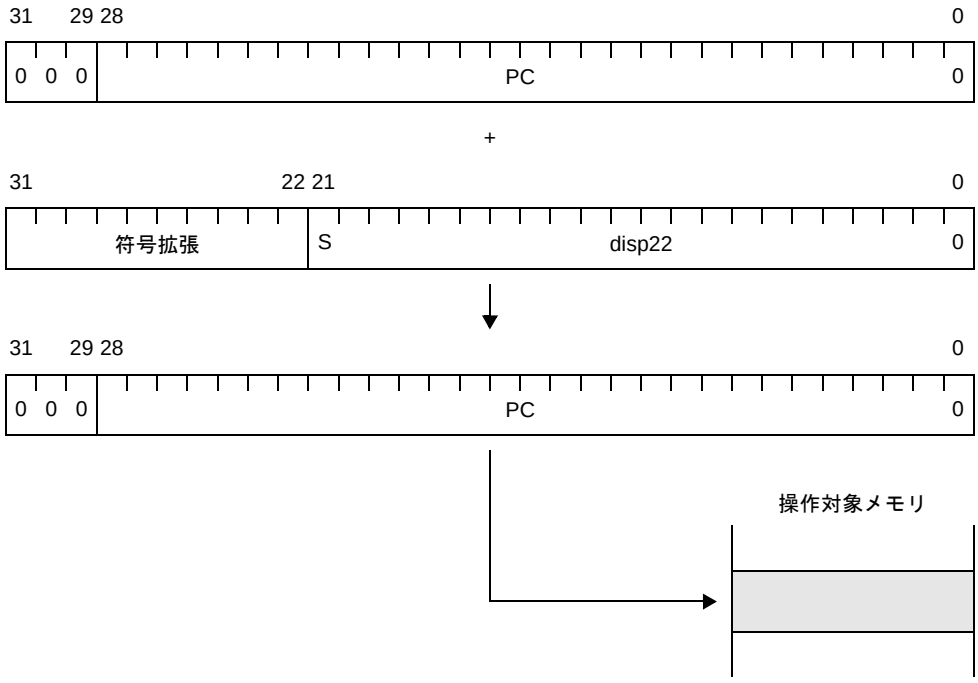


図 4―50 レラティブ・アドレッシング（JR disp32 ／ JARL disp32, reg2）【V850E2】

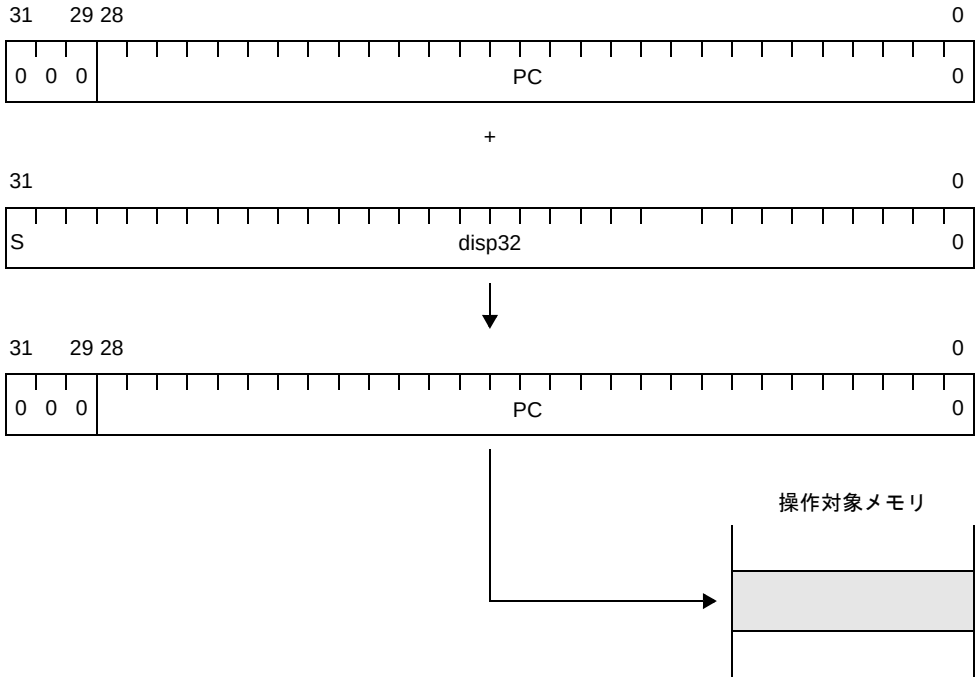


図 4―51 レラティブ・アドレッシング（Bcnd disp9）【V850 マイクロコントローラ】

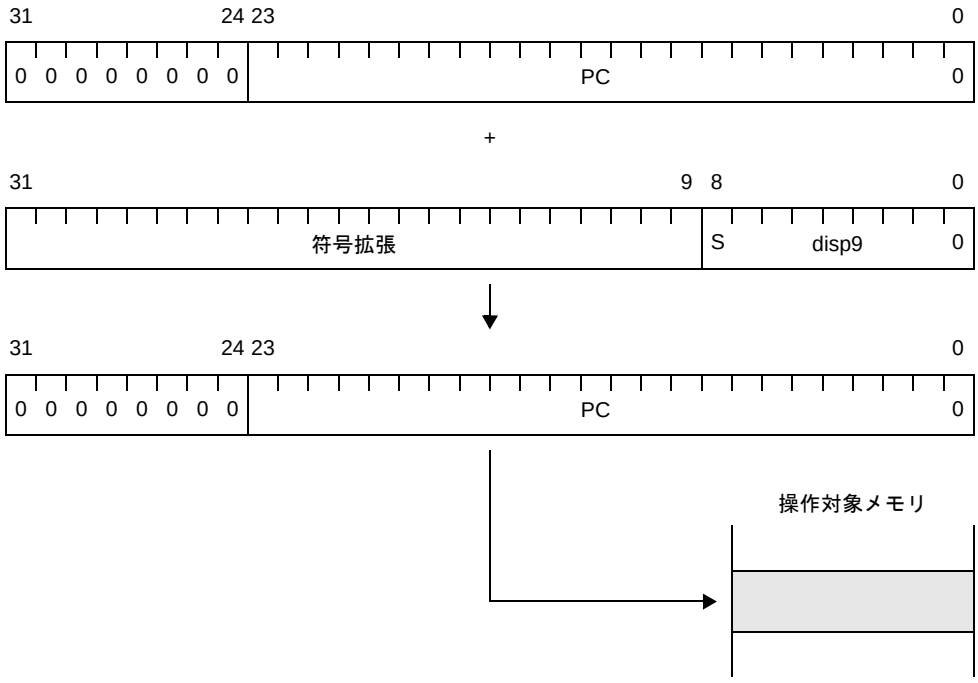


図 4―52 レラティブ・アドレッシング（Bcnd disp9）【V850ES，V850E1】

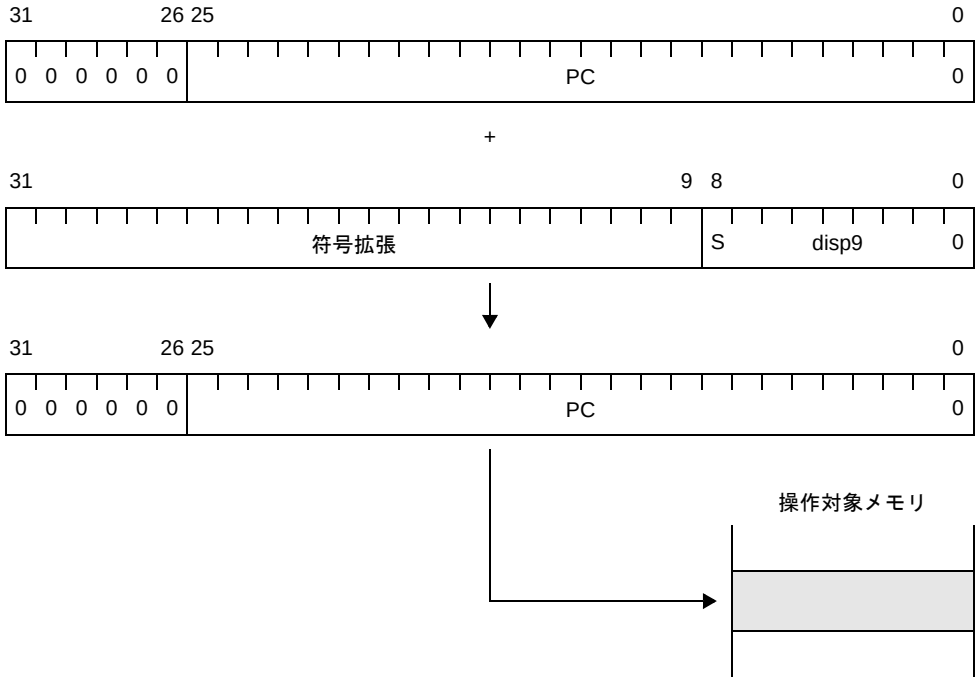
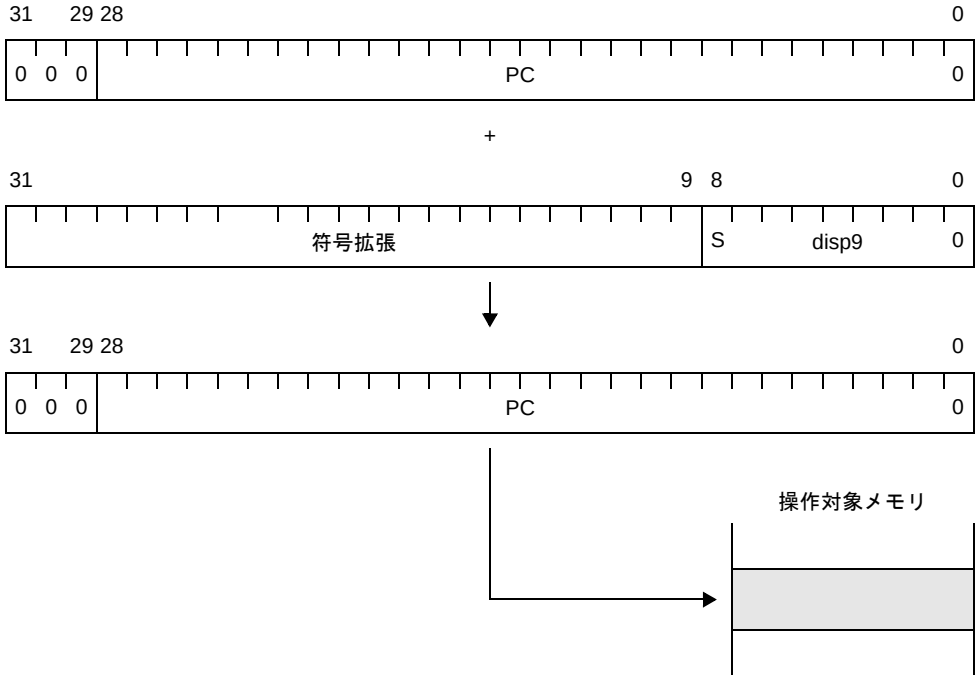


図 4―53 レラティブ・アドレッシング（Bcnd disp9）【V850E2】



（b）レジスタ・アドレッシング（レジスタ間接）

命令によって指定される汎用レジスタ（reg1）の内容をプログラム・カウンタ（PC）に転送します。

JMP [reg1] 命令が本アドレッシングの対象となります。

図 4―54 レジスタ・アドレッシング（JMP [reg1]）【V850 マイクロコントローラ】

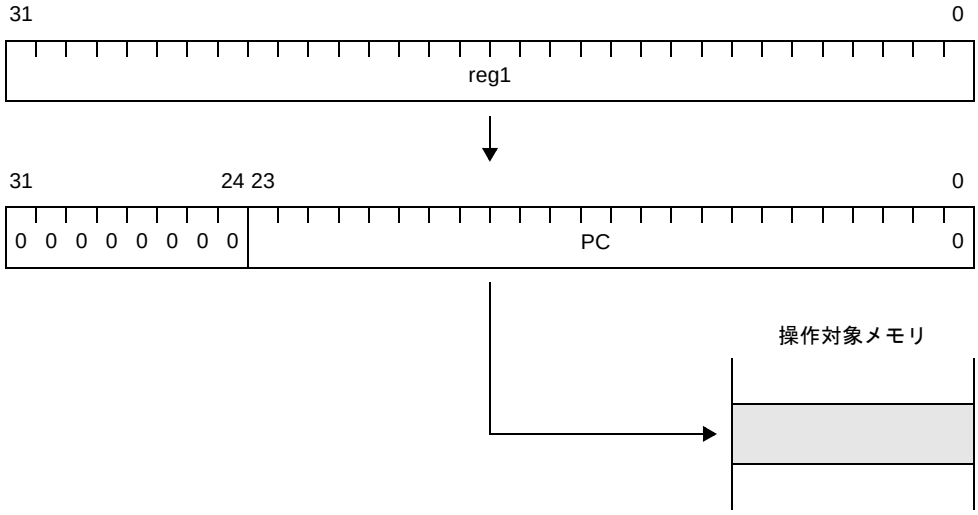


図 4―55 レジスタ・アドレッシング（JMP [reg1]）【V850ES，V850E1】

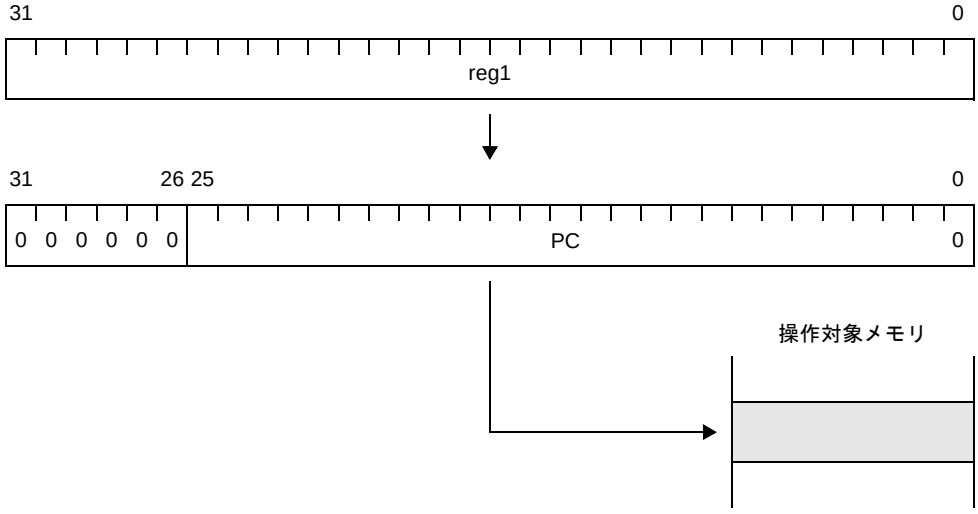
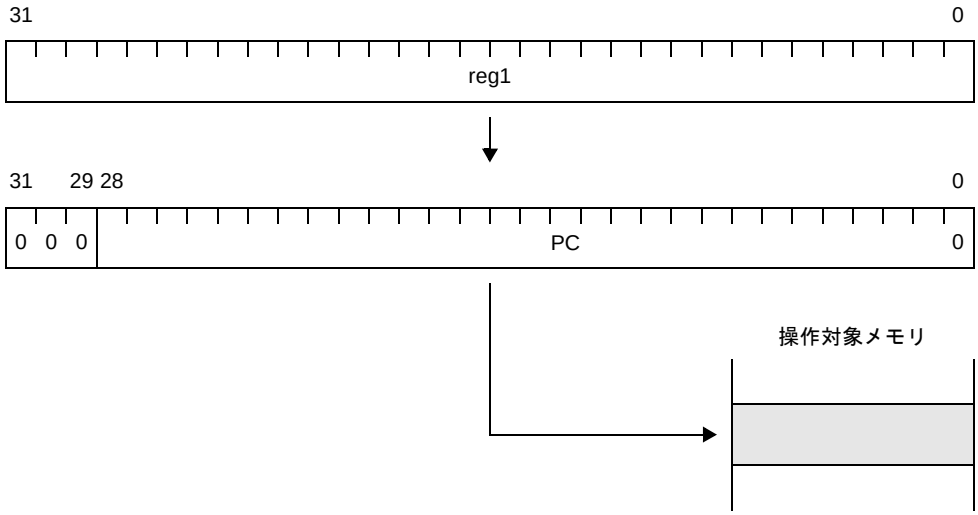


図 4―56 レジスタ・アドレッシング（JMP [reg1]）【V850E2】

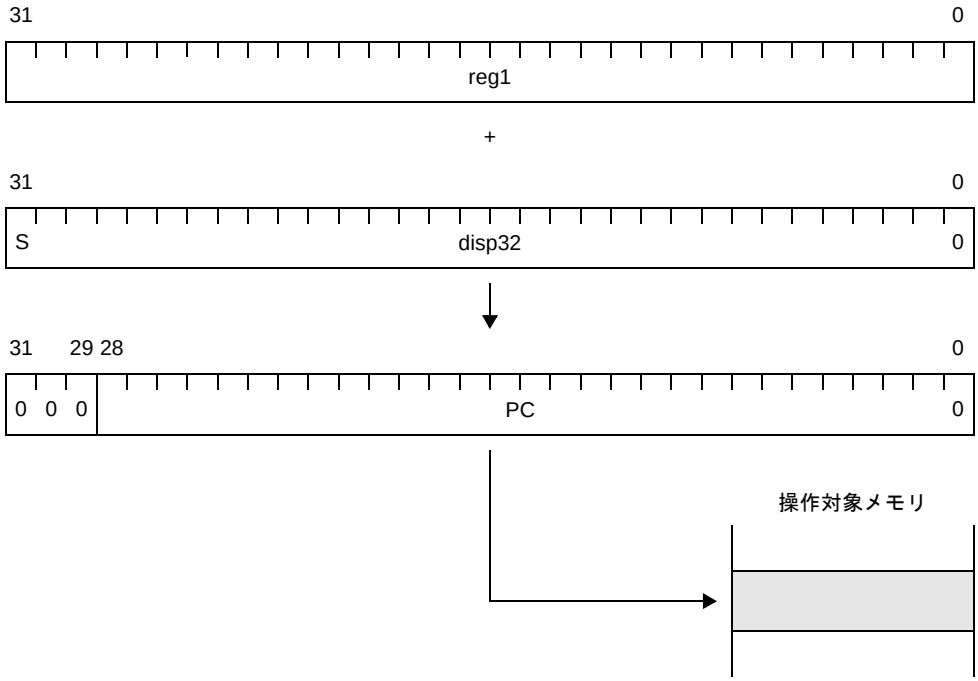


（c）ペースト・アドレッシング

命令によって指定される汎用レジスタ（reg1）に，32 ビット・データ（ディスプレースメント：disp）を加算した内容をプログラム・カウンタ（PC）に転送します。

JMP disp32[reg1] 命令が本アドレッシングの対象となります。

図 4―57 レジスタ・アドレッシング（JMP disp32[reg1]）【V850E2】



(2) オペランド・アドレス

命令を実行する際に対象となるレジスタやメモリなどをアクセスするために，次に示す方法があります。

(a) レジスタ・アドレッシング

汎用レジスタ指定フィールドにより指定される汎用レジスタ，またはシステム・レジスタをオペランドとしてアクセスするアドレッシングです。

オペランドに reg1，reg2,，reg3，または regID を含む命令が本アドレッシングの対象となります。

(b) イミーディエト・アドレッシング

命令コード中に操作対象となる 5 ビット・データ，16 ビット・データを持つアドレッシングです。

オペランドに imm5，imm16，vector，または cccc を含む命令が本アドレッシングの対象となります。

- vector

トラップ・ベクタ（00H-1FH）を指定する 5 ビット・イミーディエトであり，TRAP 命令で使用されるオペランドです。

- cccc

条件コード指定用の 4 ビット・データであり，CMOV，SASF，SETF 命令などで使用されるオペランドです。0 の 1 ビットを上位に付加し，5 ビット・イミーディエトとして命令コード中に割り当てられます。

(c) ペースト・アドレッシング

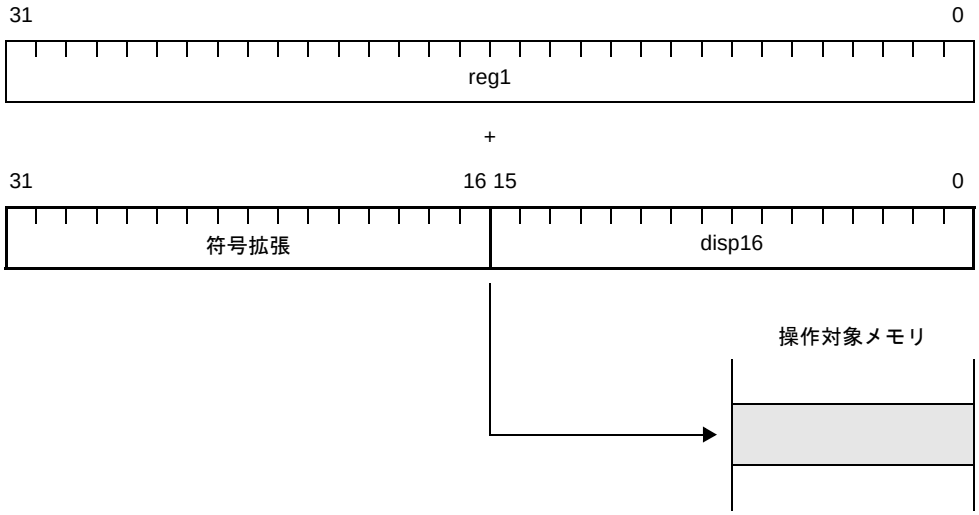
ペースト・アドレッシングには，次に示す 2 種類があります。

- タイプ 1

命令コード中のアドレッシング指定フィールドで指定される汎用レジスタ（reg1）の内容と 16 ビット・ディスプレースメント（disp16）の和がオペランド・アドレスとなって，操作対象となるメモリへのアクセスを行うアドレッシングです。

オペランドに disp16[reg1] を含む命令が本アドレッシングの対象となります。

図 4―58 ペースト・アドレッシング（タイプ 1）【V850 マイクロコントローラ，V850ES，V850E1，V850E2】

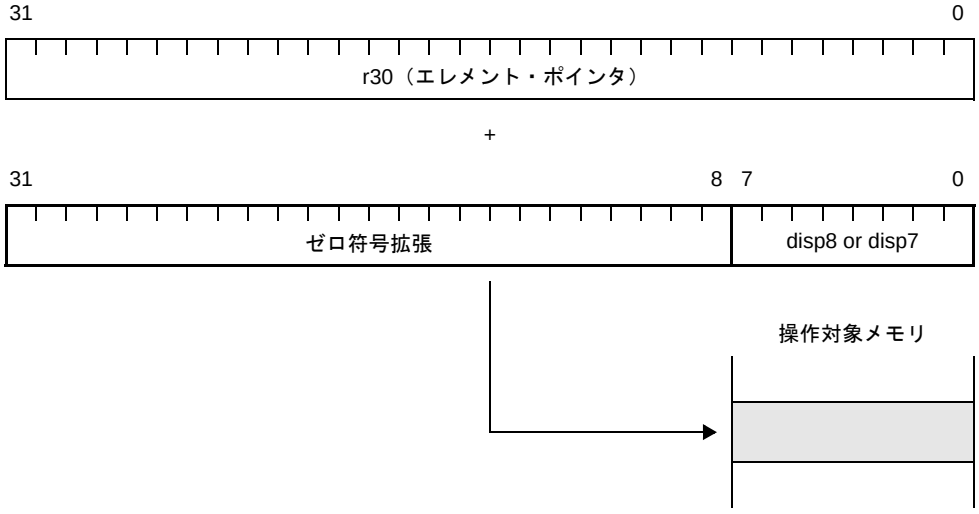


- タイプ 2

エレメント・ポインタ（r30）の内容と，7，または 8 ビット・ディスプレースメント・データ（disp7，disp8）の和がオペランド・アドレスとなって，操作対象となるメモリへのアクセスを行うアドレッシングです。

SLD 命令，SST 命令が本アドレッシングの対象となります。

図 4―59 ペースト・アドレッシング（タイプ 2）【V850 マイクロコントローラ，V850ES，V850E1，V850E2】



備考 バイト・アクセス = disp7

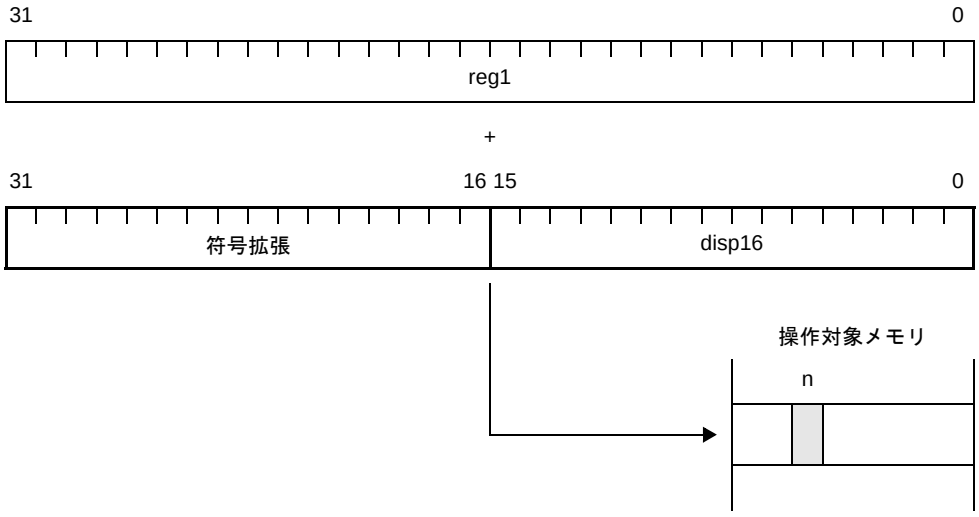
ハーフワード・アクセス，またはワード・アクセス = disp8

(d) ビット・アドレッシング

汎用レジスタ（reg1）の内容とワード長まで符号拡張した 16 ビット・ディスプレースメント（disp16）の和をオペランド・アドレスとして，操作対象となるメモリ空間の 1 バイト中の 1 ビット（3 ビット・データ「bit#3」で指定）をアクセスするアドレッシングです。

ビット操作命令が本アドレッシングの対象となります。

図 4―60 ビット・アドレッシング【V850 マイクロコントローラ，V850ES，V850E1，V850E2】



備考 n：3 ビット・データ（bit#3）で指定されるビット位置（n = 0-7）

4. 5. 4　命令セット

この項では，CA850 アセンブラ（as850）がサポートする命令セットについて説明します。

（1）記号の説明

次表に，以降で用いる記号の意味を示します。

表 4―38　記号の意味

記号	意味
CMD	命令
CMDi	命令（andi，ori，または xori）
reg，reg1，reg2，reg3，reg4	レジスタ
r0	ゼロ・レジスタ
r1	アセンブラ予約レジスタ
gp	グローバル・ポインタ（r4）
ep	エレメント・ポインタ（r30）
[reg]	ベース・レジスタ
disp	ディスプレースメント（アドレスからの偏位） 特に記述のない場合 32 ビット幅を持ちます。
imm	イミーディエト（即値） 特に記述のない場合 32 ビット幅を持ちます。
bit#3	ビット・ナンバ指定用 3 ビット・データ
#label	ラベルの絶対アドレス参照
label	ラベルのセクション内オフセット参照，または PC オフセット参照 ただし，tp シンボルの生成において対象となっているセグメントに割り当てられたセクションに対しては，セクション内オフセットの代わりに tp シンボルからのオフセット参照
$label	ラベルの gp オフセット参照
!label	ラベルの絶対アドレス参照（命令展開なし）
%label	ラベルのセクション内オフセット参照（命令展開なし）
hi(*value*)	*value* の上位 16 ビット
lo(*value*)	*value* の下位 16 ビット
hi1(*value*)	*value* の上位 16 ビット + value のビット番号 15 のビット値[注]
addr	アドレス
PC	プログラム・カウンタ
PSW	プログラム・ステータス・ワード
regID	システム・レジスタ番号（0 ～ 31）
vector	トラップ・ベクタ（0 ～ 31）
BITIO	周辺 I/O レジスタ（1 ビット操作専用）

注 LSB（Least Significant Bit）はビット番号 0 です。

(2) オペランド

以下に，as850 におけるオペランドの記述形式について説明します。as850 では，命令，および疑似命令に対するオペランドとして，レジスタ，定数，シンボル，ラベル参照，および定数，シンボル，ラベル参照，演算子を指定できます。

(a) レジスタ

as850 において指定できるレジスタを次に示します。**注**

注 PSW，およびシステム・レジスタは，ldsr ／ stsr 命令において，番号で指定します。なお，as850 では，PC をオペランドに指定する方法はありません。

```
r0，zero，r1，r2，hp，r3，sp，r4，gp，r5，tp，r6，r7，r8，r9，r10，r11，r12，r13，r14，r15，r16，
r17，r18，r19，r20，r21，r22，r23，r24，r25，r26，r27，r28，r29，r30，ep，r31，lp
```

r0 と zero（ゼロ・レジスタ），r2 と hp（ハンドラ・スタック・ポインタ），r3 と sp（スタック・ポインタ），r4 と gp（グローバル・ポインタ），r5 と tp（テキスト・ポインタ），r30 と ep（エレメント・ポインタ），r31 と lp（リンク・ポインタ）は同じレジスタを示します。

(b) r0

r0 は，常に 0 の値を持つレジスタです。したがって，デスティネーション・レジスタとして指定した場合にも，結果の代入は行われません。なお，r0 をデスティネーション・レジスタとして指定した場合，次のメッセージが出力され**注**，アセンブルが続行されます。

注 このメッセージの出力は，as850 の起動時に警告メッセージ抑止オプション（-w）を指定することにより抑止できます。

```
mov     0x10, r0
  ↓
W3013: register r0 used as destination register
```

- ターゲット・デバイスに V850Ex を使用する場合，次の命令で r0 をデスティネーション・レジスタとして指定すると，警告メッセージではなくエラー・メッセージが出力されます。
  dispose，divh 命令の形式（1），および（2），ld.bu，ld.hu，mov 命令の形式（2），movea，movhi，mulh，mulhi，satadd，satsub，satsubi，satsubr，sld.bu，sld.hu

```
divh    r10, r0
  ↓
E3240: illegal operand (can not use r0 as destination in V850E mode)
```

- 次の命令で，ターゲット・デバイスに V850Ex を使用する場合，次の命令で r0 をソース・レジスタとして指定すると，警告メッセージではなくエラー・メッセージが出力されます。

divh 命令の形式（1），switch

```
divh r0, r10
  ↓
E3239: illegal operand (can not use r0 as source in V850E mode）
```

（c）r1

アセンブラ予約レジスタ（r1）は，as850 において，命令展開を行う際のテンポラリ・レジスタとして用いられるレジスタです。なお，r1 をソース・レジスタ，またはデスティネーション・レジスタとして指定した場合，次のメッセージが出力され**注**，アセンブルが続行されます。

注 このメッセージの出力は，as850 の起動時に警告メッセージ抑止オプション（-w）を指定することにより抑止できます。

```
mov     0x10, r1
  ↓
W3013: register r1 used as destination register
```

```
mov     r1, r10
  ↓
W3013: register r1 used as source register
```

（d）定数

as850 では，命令，および疑似命令のオペランド指定で使用可能な絶対値式，または相対値式の構成要素として，整定数，および文字定数を用いることができます。また，ld ／ st 命令，およびビット操作命令のオペランド指定には，各デバイス・ファイルで定義されている「周辺 I/O レジスタ名」も指定できます。これにより，ポート・アドレスに対する入出力を行うことができます。また，.float 疑似命令のオペランド指定には浮動小数点定数を，.str 疑似命令のオペランド指定には文字列定数を用いることができます。

（e）シンボル

as850 では，命令，および疑似命令のオペランド指定で使用可能な絶対値式，または相対値式の構成要素として，シンボルを用いることができます。

（f）ラベル参照

as850 では，次に示した命令／疑似命令のオペランド指定で，使用可能な相対値式の構成要素として，ラベル参照を用いることができます。

- メモリ参照命令（ロード／ストア命令，およびビット操作命令）
- 演算命令（算術演算命令，飽和演算命令，および論理演算命令）
- 分岐命令
- 領域確保疑似命令（ただし，.word ／ .hword ／ .byte 疑似命令のみ）

as850 では，ラベル参照は参照方法の違い，およびそのラベル参照を用いている命令／疑似命令の違いにより次に示すように異なる意味を持ちます。

表 4―39　ラベル参照

参照方法	用いている命令	意味
#label	メモリ参照命令，演算命令，jmp 命令	ラベル label 定義の存在する位置の絶対アドレス（アドレス 0 からのオフセット**注 1**）。 32 ビットのアドレスを持ち，V850Ex 以外では，必ず 2 命令に展開。
	領域確保疑似命令（.word ／ .hword ／ .byte）	ラベル label 定義の存在する位置の絶対アドレス（アドレス 0 からのオフセット**注 1**）。 ただし，32 ビットのアドレスを，確保した領域の大きさに準じてマスクした値
label	メモリ参照命令，演算命令	ラベル label 定義の存在する位置のセクション内オフセット（ラベル label 定義の存在するセクションの先頭アドレスからのオフセット**注 2**）。 32 ビットのオフセットを持ち，必ず 2 命令に展開。 ただし，tp シンボルの生成において対象となっているセグメントに割り当てられたセクションに対しては，tp シンボルからのオフセット参照。
	jmp 命令を除く分岐命令	ラベル label 定義の存在する位置の PC オフセット（ラベル label 参照を用いている命令の先頭アドレスからのオフセット**注 2**）。
	領域確保疑似命令（.word ／ .hword ／ .byte）	ラベル label 定義の存在する位置のセクション内オフセット（ラベル label 定義の存在するセクションの先頭アドレスからのオフセット**注 2**）。 ただし，32 ビットのオフセットを，確保した領域の大きさに準じてマスクした値。
$label	メモリ参照命令，演算命令	ラベル label 定義の存在する位置 gp オフセット（グローバル・ポインタの指すアドレスからのオフセット**注 3**）。

参照方法	用いている命令	意味
!label	メモリ参照命令，演算命令	ラベル label 定義の存在する位置の絶対アドレス（アドレス 0 からのオフセット**注 1**）。 16 ビットのアドレスを持ち，16 ビット・ディスプレースメント，またはイミーディエトをもつ命令に指定した場合，命令展開は行われません。 その他の命令に指定した場合，適切な 1 命令に展開されます。 ラベル label の定義したアドレスが，16 ビットで表現できる範囲でない場合，リンク時にエラーになります。
	領域確保疑似命令（.word ／ .hword ／ .byte）	ラベル label 定義の存在する位置の絶対アドレス（アドレス 0 からのオフセット**注 1**）。 ただし，32 ビットのアドレスを，確保した領域の大きさに準じてマスクした値。
%label	メモリ参照命令，演算命令	ラベル label 定義の存在する位置のセクション内オフセット（ラベル label 定義の存在するセクションの先頭アドレスからのオフセット**注 2**）。 16 ビットのオフセットを持ち，16 ビット・ディスプレースメント，またはイミーディエトをもつ命令に指定した場合，命令展開は行われません。 その他の命令に指定した場合，適切な 1 命令に展開されます。 ラベル label の定義したアドレスが，16 ビットで表現できる範囲でない場合，リンク時にエラーになります。 または，ラベル label 定義の存在する位置の ep オフセット（エレメント・ポインタの示すアドレスからのオフセット）。
	領域確保疑似命令（.word ／ .hword ／ .byte）	ラベル label 定義の存在する位置のセクション内オフセット（ラベル label 定義の存在するセクションの先頭アドレスからのオフセット**注 2**）。 ただし，32 ビットのオフセットを，確保した領域の大きさに準じてマスクした値。

注 1. リンク後のオブジェクト・ファイルにおけるアドレス 0 からのオフセットです。

2. リンク後のオブジェクト・ファイルにおいて，ラベル label の定義の存在するセクションが割り当てられたセクション（出力セクション）の先頭アドレスからのオフセットです。

3. 上記出力セクションが割り当てられたセグメントに対する，テキスト・ポインタ・シンボルの値 + グローバル・ポインタ・シンボルの値の指すアドレスからのオフセットです。

次に，メモリ参照命令，演算命令，分岐命令，および領域確保疑似命令におけるラベル参照の意味を示します。

表 4―40 メモリ参照命令

参照方法	意味
#label[reg]	ラベル label の絶対アドレスがディスプレースメントとして扱われます。 32 ビットの値を持ち，必ず 2 命令に展開されます。 #label[r0] とすることにより絶対アドレスによる参照を指定できます。 [reg] の部分が省略でき，省略した場合，as850 では，[r0] が指定されたものとみなされます。
label[reg]	ラベル label のセクション内オフセットがディスプレースメントして扱われます。32 ビットの値を持ち，必ず 2 命令に展開されます。reg に対象とするセクションの先頭アドレスを指すレジスタを指定し，label[reg] とすることにより一般的なレジスタ相対の参照が指定できます。 ただし，tp シンボルの生成において対象となっているセグメントに割り当てられたセクションに対しては，tp シンボルからのオフセットをディスプレースメントとして扱います。
$label[reg]	ラベル label の gp オフセットがディスプレースメントとして扱われます。ラベル label の定義されたセクションにより，32，または 16 ビットの値を持ち，命令展開のパターンが変化します**注**。16 ビットの値を持つ命令展開が行われた場合，ラベル label の定義したアドレスより算出されたオフセットが 16 ビットで表現できる範囲でない場合，リンク時にエラーになります。$label[gp] とすることにより gp レジスタ相対の参照（gp オフセット参照と呼ぶ）が指定できます。[reg] の部分が省略でき，省略した場合，as850 では，[gp] が指定されたものとみなされます。
!label[reg]	ラベル label の絶対アドレスがディスプレースメントとして扱われます。16 ビットの値を持ち，命令展開は行われません。ラベル label に定義したアドレスが 16 ビットで表現できない場合，リンク時にエラーになります。!label[r0] とすることにより絶対アドレスによる参照が指定できます。 [reg] の部分が省略でき，省略した場合は [r0] が指定されたものとみなされます。 ただし，#label[reg] 参照とは異なり，命令展開は行われません。

参照方法	意味
%label[reg]	ラベル label のセクション内オフセットがディスプレースメントとして扱われます。ラベル label が ep シンボルとなっているセクションに配置された場合には，ep シンボルからのオフセットをディスプレースメントとして扱われます。16 ビット，または命令によってはそれ以下の値を持ち，その範囲で表現できる値でない場合，リンク時にエラーになります。[reg] の部分が省略でき，省略した場合，as850 では，[ep] が指定されたものとみなされます。

注 「(h) gp オフセット参照」を参照してください。

表 4―41 演算命令

参照方法	意味
#label	ラベル label の絶対アドレスがイミーディエトとして扱われます。 32 ビットの値を持ち，必ず 2 命令に展開されます。
label	ラベル label のセクション内オフセットがイミーディエトとして扱われます。 32 ビットの値を持ち，必ず 2 命令に展開されます。ただし，tp シンボルの生成において対象となっているセグメントに割り当てられたセクションに対しては，tp シンボルからのオフセットがイミーディエトとして扱われます。
$label	ラベル label の gp オフセットがイミーディエトとして扱われます。 ラベル label の定義されたセクションにより，32，または 16 ビットの値を持ち，命令のパターンが変化します**注 1**。16 ビットの値を持つ展開をされた場合，ラベル label の定義したアドレスより算出されたオフセットが 16 ビットで表現できる範囲でない場合，リンク時にエラーになります。
!label	ラベル label の絶対アドレスがイミーディエトとして扱われます。 16 ビットの値を持ち，イミーディエトとして 16 ビットの値を指定できるアーキテクチャの演算命令**注 2** に指定した場合，命令展開は行われません。 add，mov，および mulh 命令に指定した場合，適切な 1 命令に展開されます。それ以外の命令に指定することはできません。16 ビットで表現できる範囲でない場合，リンク時にエラーになります。

参照方法	意味
%label	ラベル label のセクション内オフセットがイミーディエトとして扱われます。 ラベル label が ep シンボルの対象となっているセクションに配置された場合には，ep シンボルからのオフセットをディスプレースメントとして扱われます。 16 ビットの値を持ち，イミーディエトとして 16 ビットの値を指定できるアーキテクチャの演算命令 **注 2** に指定した場合，命令展開は行われません。 ただし，label 参照とは異なり，命令展開は行われません。また，この参照方法は，イミーディエトとして 16 ビットの値を指定できるアーキテクチャの演算命令と，add，mov，および mulh 命令のみで指定できます。add，mov，および mulh 命令に指定した場合，適切な 1 命令に展開します。それ以外の命令に指定することはできません。16 ビットで表現できる範囲でない場合，リンク時にエラーになります

注 1. 「(h) gp オフセット参照」を参照してください。

2. イミーディエトとして 16 ビットの値を指定できる命令は，addi，andi，movea，mulhi，ori，satsubi，および xori です。

表 4―42 分岐命令

参照方法	意味
#label	jmp 命令において，ラベル label の絶対アドレスが飛び先アドレスとして扱われます。 32 ビットの値を持ち，必ず 3 命令に展開されます。
label	jmp 命令以外の分岐命令において，ラベル label の PC オフセットがディスプレースメントとして扱われます。 22 ビットの値を持ち，表現できない範囲である場合，リンク時にエラーになります。

表 4―43 領域確保疑似命令

参照方法	意味
#label !label	.word ／ .hword ／ .byte 疑似命令において，ラベル label の絶対アドレスを値として扱われます。 32 ビットの値を持ちますが，各疑似命令のビット幅に応じてマスクされます。

参照方法	意味
label %label	word ／ .hword ／ .byte 疑似命令において，ラベル label の定義されたセクション内オフセットを値として扱われます。 32 ビットの値を持ちますが，各疑似命令のビット幅に応じてマスクされます。
$label	.word ／ .hword ／ .byte 疑似命令において，ラベル label の gp オフセットを値として扱われます。 32 ビットの値を持ちますが，各疑似命令のビット幅に応じてマスクされます。

（g）ep オフセット参照

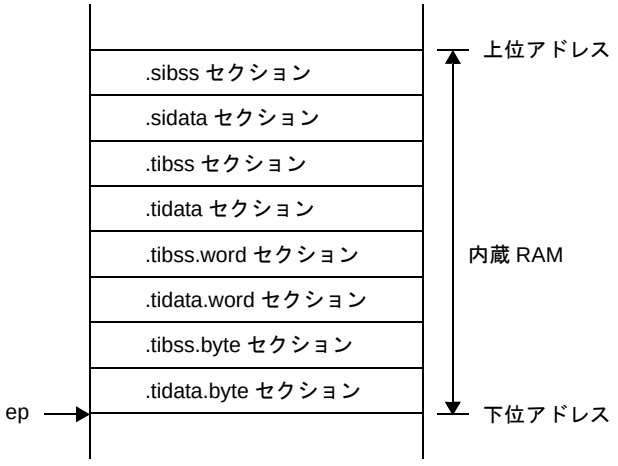
ここでは，ep オフセット参照について説明します。CA850 では，明示的に内蔵 RAM に置かれるデータについては，基本的に，次のことが想定されています。

エレメント・ポインタ（ep）の指すアドレスからのオフセットによって参照する

なお，内蔵 RAM に置かれるデータは，次の 2 つに分けられます。

- .tidata ／ .tibss ／ .tidata.byte ／ .tibss.byte ／ .tidata.word ／ .tibss.word セクション（コード・サイズが小さいメモリ参照命令（sld ／ sst）で参照するデータ）
- .sidata ／ .sibss セクション（コード・サイズが大きいメモリ参照命令（ld ／ st）で参照するデータ）

図 4―61　内蔵 RAM のメモリ配置イメージ



- データの割り当て

内蔵 RAM に置くセクションへのデータの割り当ては，次の方法で行います。

- C 言語を用いてプログラムを作成する場合

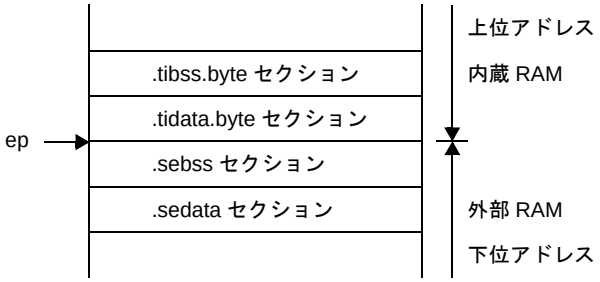
“#pragma section”指令により，セクション種別“tidata”，“tidata.byte”，“tidata.word”，または“sidata”を指定してデータを割り当てます。

セクション・ファイルで，セクション種別“tidata”，“tidata.byte”，“tidata.word”，または“sidata”を指定してデータを割り当てます。ca850 のオプションにより，そのセクション・ファイルをコンパイル時に入力します。

- アセンブリ言語を用いてプログラムを作成する場合

セクション定義疑似命令により，セクション種別 .tidata.byte，.tibss.byte，.tidata.word，.tibss.word，.sidata，または .sibss のセクションへデータを割り当てます。なお，上記と同様の方法により，.sedata，または .sebss セクションへデータを割り当てることで，外部 RAM に置かれる一定範囲のデータに対しても，ep オフセット参照を行うことができます。

図 4―62　外部 RAM（.sedata ／ .sebss セクション）のメモリ配置イメージ



- データの参照

「データの割り当て」に従い，as850 では，次のように機械語命令列が生成されます。

- .tidata，.tibss，.tidata.byte，.tibss.byte，.tidata.word，.tibss.word，.sidata，.sibss，.sedata，または .sebss セクションに割り当てるデータの %label による参照に対しては，ep オフセット参照を行う機械語命令が生成
- 上記以外のセクションに割り当てるデータの %label による参照に対しては，セクション内オフセット参照を行う機械語命令列が生成

例

```
    .sidata
sidata: .hword  0xfff0
    .data
data:   .hword  0xfff0
    .text
    ld.h    %sidata, r20    -- (1)
    ld.h    %data, r20      -- (2)
```

as850 では，%label による参照に対して，（1）の場合，定義したデータが .sidata セクションに配置されているため ep オフセット参照とみなされ，（2）の場合，セクション内オフセット参照とみなされて，機械語の命令列が生成されます。なお，as850 では，データが配置されたセクションが正しいものとして処理が行われます。このため，データの配置に誤りがある場合でも，検出できません。

例

```
.text
ld.h    %label[ep], r20
```

label を .sidata セクションに配置し，ep オフセット参照を行う命令を記述したが，配置誤りにより .data セクションに配置された場合，ベース・レジスタである ep シンボル値 + label の .data セクション内オフセット値にあるデータがロードされます。

例

```
.text
ld.h    %label1[r10], r20   --  (1)
.option ep_label
ld.h    %label2[ep], r21    --  (2)
.option no_ep_label
ld.h    %label3[r10], r22   --  (3)
```

（1）：

定義したデータが配置されたセクションによって，ep オフセット参照，またはセクション内オフセット参照となります（デフォルト）。

（2）：

.option ep_label 疑似命令によって指定された範囲内であるので，定義したデータが配置されたセクションにかかわらず，ep オフセット参照となります。

（3）：

.option no_ep_label 疑似命令によって指定された範囲内であるので，(1) の場合と同じ扱いとなります。

（h）gp オフセット参照

ここでは，gp オフセット参照について説明します。CA850 では，（.sedata ／ .sebss セクション以外の）外部 RAM に置かれるデータについては，基本的に次のことが想定されています。

グローバル・ポインタ（gp）の指すアドレスからのオフセットによって参照する

また，C 言語における“#pragma section”指令や，C コンパイラに入力するセクション・ファイル，アセンブリ言語におけるセクション定義疑似命令による，内蔵 ROM ／内蔵 RAM などの r0 相対のメモリ割り当てを行わない場合，すべてのデータが gp オフセット参照となります。

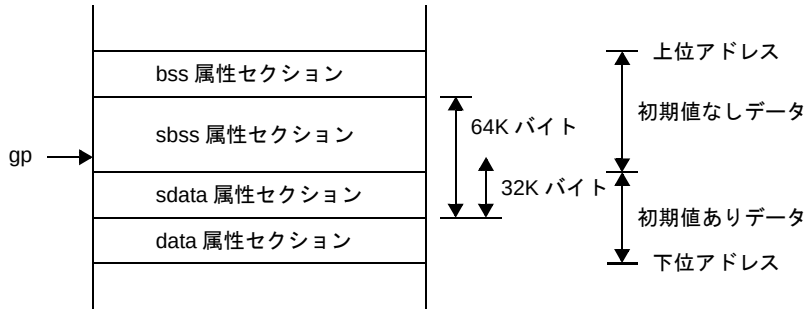
- データの割り当て

V850 マイクロコントローラの機械語命令におけるメモリ参照命令（ld ／ st）は，ディスプレースメントに 16 ビットのイミーディエトしかとれません。このため，CA850 ではデータを以下に示した 2 つに分け，前者のデータが sdata 属性セクション，または sbss 属性セクションに，後者のデータが data 属性セクション，または bss 属性セクションに割り当てられます。なお，初期値ありデータは sdata ／ data 属性セクションに，初期値なしデータは sbss ／ bss 属性セクションに割り当てられます。

CA850 では，デフォルトで，data ／ sdata ／ sbss ／ bss 属性セクションの順に下位アドレスから割り当てられます。また，グローバル・ポインタ（gp）は sdata 属性セクションの先頭アドレスに 32 K バイトを加算したアドレスを指すよう，スタート・アップ・モジュールなどにおいて設定することを想定しています。

- グローバル・ポインタ（gp）と 16 ビットのディスプレースメントを用いて参照することのできるメモリ範囲に割り当てるデータ
- グローバル・ポインタ（gp）と（複数の命令によって構成される）32 ビットのディスプレースメントを用いて参照することのできるメモリ範囲に割り当てるデータ

図 4―63　gp オフセット参照セクションのメモリ配置イメージ



備考　sdata 属性セクションと sbss 属性セクションをあわせて 64 K バイトです。gp は sdata 属性セクションの先頭から 32 K バイトの位置です。

したがって，sdata ／ sbss 属性セクションのデータを参照する場合は，1 命令で行えますが，data ／ bss 属性セクションのデータを参照する場合は，複数の命令が必要となります。つまり，より多くのデータを sdata ／ sbss 属性セクションに割り当てたほうが，生成された機械語命令の実行効率，およびオブジェクト効率は向上します。しかし，16 ビットのディスプレースメントで参照できるメモリ範囲の大きさは限られています。

そこで，すべてのデータを sdata ／ sbss 属性セクションに割り当てられない場合，どのデータを sdata ／ sbss 属性セクションに割り当てるかを定めなければなりません。

CA850 では，“できるだけ多くのデータを sdata ／ sbss 属性セクションに割り当てる”という方針がとられています。デフォルトの処理では，すべてのデータが sdata ／ sbss 属性セクションに割り当てられますが，割り当てられるデータを選別する場合，次の方法により行います。

- -G*num* オプションを指定する場合

  C コンパイラ（ca850），またはアセンブラ（as850）起動時に -G*num* オプションを指定することにより，sdata ／ sbss 属性セクションへ *num* バイト以下のデータが割り当てられます。

- プログラムでデータを割り当てるセクションを指定する場合

  参照頻度の高いデータを，明示的に sdata ／ sbss 属性セクションへ割り当てます。アセンブリ言語の場合，セクション定義疑似命令で，C 言語の場合，#pragma section 指令で割り当てられます。

- セクション・ファイルで指定する場合

  C 言語の場合，セクション・ファイルでセクション種別“sdata”を指定してデータを割り当てます。ca850 のオプションにより，そのセクション・ファイルをコンパイル時に入力します。

- データの参照

前述のデータの割り付けに従い，as850 では，次のようになります。

- sdata ／ sbss 属性セクションに割り当てるデータの gp オフセット参照に対しては，16 ビットのディスプレースメントを用いた参照を行う機械語命令が生成されます。
- data ／ bss 属性セクションに割り当てるデータの gp オフセット参照に対しては，（複数の機械語命令によって構成される）32 ビットのディスプレースメントを用いた参照を行う機械語命令列が生成されます。

例

```
    .data
data:   .word   0xfff00010  -- (1)
    .text
    ld.w    $data[gp], r20  -- (2)
```

as850 では，（1）で定義したデータを gp オフセット参照している（2）の ld.w 命令に対して，次の命令列に等価な機械語の命令列が生成されます。**注**

```
movhi   hi1($data), gp, r1
ld.w    lo($data)[r1], r20
```

注 hi1 ／ lo に関しては，「(i) hi ／ lo ／ hi1 について」を参照してください。

なお，as850 では，1 ファイルずつ処理が行われます。このため，指定したファイル内に定義を持つデータに対しては，そのデータがどの属性セクションに割り当てられるデータであるかを判断することができますが，指定したファイル内の定義を持たないデータに対しては判定できません。そこで，as850 では，“前述の割り当ての方針（一定サイズ以下のデータを sdata ／ sbss 属性セクションに割り当てる）が守られている”ことを想定し，起動時に -G*num* オプションが指定された場合，次のように機械語命令が生成されます。**注**

注 .option 疑似命令のオプションで，data，または sdata が指定されたデータに対しては，サイズに関係なく .data セクション，.sdata セクションへ割り当てられるものとして扱われます。

- 指定したファイル内に定義を持たない，*num* バイト以下のデータの gp オフセット参照に対しては，16 ビットのディスプレースメントを用いた参照を行う機械語命令が生成されます。
- 指定したファイル内に定義を持たない，*num* バイトより大きいデータの gp オフセット参照に対しては，（複数の機械語命令によって構成される）32 ビットのディスプレースメントを用いた参照を行う機械語命令列が生成生成されます。

しかし，この判定を行うには，指定したファイル内に定義を持たない gp オフセット参照されているデータのサイズが，判定できなければなりません。そこで，アセンブリ言語を用いてプログラムを作成する場合，指定したファイル内に定義を持たないデータ（実際には指定したファイル内に定義を持たず gp オフセット参照されているラベル）に対し，.extern 疑似命令を用いて，サイズを指定してください。

```
.extern data, 4         -- (1)
.text
ld.w    $data[gp], r20  -- (2)
```

as850 では，起動時に -G2 を指定した場合，（1）で宣言したデータを gp オフセット参照している（2）の ld.w 命令に対しては，次の命令列に等価な機械語の命令列が生成されます。**注**

```
movhi   hi1($data), gp, r1
ld.w    lo($data)[r1], r20
```

注 hi1 ／ lo に関しては，「(i) hi ／ lo ／ hi1 について」を参照してください。

また，C 言語を用いてプログラム作成する場合，CA850 に含まれる C コンパイラ（ca850）が，指定したファイル内に定義を持たないデータ（実際には指定したファイル内に定義を持たず gp オフセット参照されているラベル）に対して，自動的に .extern 疑似命令を生成し，サイズを指定するコードを出力します。

備考 as850 におけるデータの gp オフセット参照（具体的には，ラベルの gp オフセット参照を持つ相対値式をディスプレースメントとするメモリ参照命令）の扱いをまとめると，次のようになります。

- そのデータが，指定したファイル内に定義を持つ場合
  - sdata ／ sbss 属性セクションに割り付けるデータである場合**注**
    16 ビットのディスプレースメントを用いた参照を行う機械語命令が生成されます。
  - sdata ／ sbss 属性セクションに割り付けるデータでない場合
    32 ビットのディスプレースメントを用いた参照を行う機械語命令列が生成されます。

注 “ラベル±定数式”の形式の相対値式で，定数式の値が 16 ビットの範囲を越える場合は，as850 では，32 ビットのディスプレースメントを用いた参照を行う機械語命令列が生成されます。

- そのデータが，指定したファイル内に定義を持たない場合
  - 起動時に -G*num* オプションを指定した場合
    データ（gp オフセット参照されているラベル）に対し，.comm ／ .extern ／ .globl ／ .lcomm ／ .size 疑似命令により，
    【0 以外かつ *num* バイト以下を指定した場合】
    sdata ／ sbss 属性セクションに割り当てるデータであるとみなされ，16 ビットのディスプレースメントを用いた参照を行う機械語命令が生成されます。
    【上記以外の場合】
    sdata ／ sbss 属性セクションに割り当てるデータでないとみなされ，32 ビットのディスプレースメントを用いた参照を行う機械語命令が生成されます

  - 起動時に -G*num* オプションを指定しなかった場合
    sdata ／ sbss 属性セクションに割り当てるデータであるとみなされ，16 ビットのディスプレースメントを用いた参照を行う機械語命令が生成されます。

（i）hi ／ lo ／ hi1 について

- 32 ビットの定数値をレジスタに入れる場合

V850 マイクロコントローラの V850 コアでは，1 命令で 32 ビットの定数値をレジスタに格納する機械語命令を持ちません。このため，as850 では，32 ビットの定数値をレジスタに格納する場合，命令展開を行い movhi 命令と movea 命令を用いて，32 ビットの定数値を上位 16 ビットと下位 16 ビットに分けてレジスタに格納する命令列が生成されます。

例

`mov 0x18000, r11`	`movhi hi1(0x18000), r0, r1` `movea lo(0x18000), r1, r11`

この際，下位 16 ビットの格納に用いられる機械語の movea 命令は，指定された 16 ビットの値を符号拡張し 32 ビットの値として扱われます。この符号拡張された部分を補正するため as850 では，movhi 命令を用いて上位 16 ビットをレジスタに格納する際，ただ単に上位 16 ビットをレジスタに格納するのではなく，以下の値をレジスタに格納します。

上位 16 ビット + 下位 16 ビットの最上位ビット（ビット番号 15 のビット）

- 32 ビットのディスプレースメントを用いてメモリを参照する場合

V850 マイクロコントローラの機械語のメモリ参照命令（ロード／ストア命令，およびビット操作命令）は，ディスプレースメントに 16 ビットのイミーディエトしかとれません。このため，as850 では，32 ビットのディスプレースメントを用いてメモリを参照する場合，命令展開が行われ，movhi 命令とメモリ参照命令が用いられて，32 ビットのディスプレースメントの上位 16 ビットと，下位 16 ビットから，32 ビットのディスプレースメントが構成されて参照を行う命令列が生成されます。

例

`ld.w 0x18000[r11], r12`	`movhi hi1(0x18000), r11, r1` `ld.w lo(0x18000)[r1], r12`

この際，下位 16 ビットをディスプレースメントとして用いる機械語のメモリ参照命令は，指定された 16 ビットのディスプレースメントを符号拡張し，32 ビットの値として扱います。この符号拡張された部分を補正するために，as850 では，movhi 命令を用いて上位 16 ビットのディスプレースメントを構成する際，ただ単に上位 16 ビットのディスプレースメントを構成するのではなく，次のディスプレースメントを構成します。

上位 16 ビット + 下位 16 ビットの最上位ビット（ビット番号 15 のビット）

- hi ／ lo ／ hi1

次表のように as850 では，hi，lo，および hi1 を用いることにより，32 ビットの値の上位 16 ビット，32 ビットの値の下位 16 ビット，および 32 ビットの値の上位 16 ビット + ビット番号 15 のビット値を指定できます。**注**

注 アセンブラ内部では解決できない場合，この情報はリロケーション情報に反映されリンカ（ld850）において解決されます。

表 4―44 領域確保疑似命令

hi ／ lo ／ hi1	意味
hi (*value*)	*value* の上位 16 ビット
lo (*value*)	*value* の下位 16 ビット
hi1 (*value*)	*value* の上位 16 ビット + *value* のビット番号 15 のビット値

例

```
    .data
L1:
      :
    .text
    movhi   hi($L1), r0, r10    --L1 の gp オフセットの値の上位 16 ビットを r10 の
                                -- 上位 16 ビットに格納し，下位 16 ビットに 0 を格納
    movea   lo($L1), r0, r10    --L1 の gp オフセットの値の下位 16 ビットを符号拡張
                                -- し，r10 に格納
      :
    movhi   hi1($L1), r0, r1    --L1 の gp オフセットの値を r10 に格納
    movea   lo($L1), r1, r10
```

4. 5. 5　命令の説明

as850 がサポートするアセンブリ言語命令について，次の形式で説明します。

なお，as850 が生成する機械語命令についての詳細は，各デバイスのユーザーズ・マニュアルを参照してください。

命令

命令の意味を日本語と英語で示します。

[指定形式]

命令の形式を示します。

[機能]

命令の機能を示します。

[詳細説明]

命令の動作の仕方を示します。

[フラグ]

命令実行によるフラグ（PSW）の動きを示します。

ただし，ビット操作命令（set1，clr1，not1）では，実行前のフラグの値を示しています。

なお，表中の“—”は，フラグの値が変化しないことを示します。

[注意事項]

命令における注意事項を示します。

4. 5. 6　ロード／ストア命令

この項では，ロード／ストア命令について説明します。次に，この項において説明する命令を示します。

表 4―45　ロード／ストア命令

命令		意味
ld	ld.b	バイト・データのロード
	ld.h	ハーフワード・データのロード
	ld.w	ワード・データのロード
	ld.bu	符号なしバイト・データのロード【V850E】
	ld.hu	符号なしハーフワード・データのロード【V850E】
sld	sld.b	バイト・データのロード（ショート・フォーマット）
	sld.h	ハーフワード・データのロード（ショート・フォーマット）
	sld.w	ワード・データのロード（ショート・フォーマット）
	sld.bu	符号なしバイト・データのロード（ショート・フォーマット）【V850E】
	sld.hu	符号なしハーフワード・データのロード（ショート・フォーマット）【V850E】
st	st.b	バイト・データのストア
	st.h	ハーフワード・データのストア
	st.w	ワード・データのストア
sst	sst.b	バイト・データのストア（ショート・フォーマット）
	sst.h	ハーフワード・データのストア（ショート・フォーマット）
	sst.w	ワード・データのストア（ショート・フォーマット）

ld

データのロードを行います。（Load）

［指定形式］

- ld.b disp[reg1], reg2
- ld.h disp[reg1], reg2
- ld.w disp[reg1], reg2
- ld.bu disp[reg1], reg2 【V850E】
- ld.hu disp[reg1], reg2 【V850E】

disp に指定可能なものを以下に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式
- 上記のものに hi，lo，または hi1 を適用したもの

［機能］

ld.b，ld.h，ld.w，ld.bu，ld.hu 命令は，第 1 オペランドに指定したアドレスから 1 バイト分，1 ハーフワード分，および 1 ワード分のデータを取り込み，第 2 オペランドに指定したレジスタにロードします。

［詳細説明］

- disp に次のものを指定した場合，as850 では，機械語命令の ld 命令[注]が 1 つ生成されます。なお，例中の“ld”は ld.b，ld.h，ld.w，ld.bu，ld.hu のいずれかになります。

（a）-32768 ～ +32767 の範囲の絶対値式

ld disp16[reg1], reg2	ld disp16[reg1], reg2

（b）sdata ／ sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

ld $label[reg1], reg2	ld $label[reg1], reg2

（c）!label，または %label を持つ相対値式

ld !label[reg1], reg2	ld !label[reg1], reg2
ld %label[reg1], reg2	ld %label[reg1], reg2

（d）hi ( )，lo ( )，または hi1 ( ) を適用したもの

`ld disp16[reg1], reg2`	`ld disp16[reg1], reg2`

注 機械語命令の ld 命令は，ディスプレースメントに -32768 ～ +32767（0xffff8000 ～ 0x7fff）の範囲のイミーディエトをとります。

- disp に次のものを指定した場合，as850 では，命令展開が行われ，複数個の機械語命令が生成されます。

（a）-32768 ～ +32767 の範囲を越える絶対値式

`ld disp[reg1], reg2`	`movhi hi1(disp), reg1, r1` `ld lo(disp)[r1], reg2`

（b）#label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

`ld #label[reg1], reg2`	`movhi hi1(#label), reg1, r1` `ld lo(#label)[r1], reg2`
`ld label[reg1], reg2`	`movhi hi1(label), reg1, r1` `ld lo(label)[r1], reg2`
`ld $label[reg1], reg2`	`movhi hi1($label), reg1, r1` `ld lo($label)[r1], reg2`

- disp を省略した場合，as850 では，0 を指定したものとみなされます。
- disp に #label を持つ相対値式，または #label を持つ相対値式に hi ( )，lo ( )，または hi1 ( ) を適用したものを指定した場合，その後ろの [reg1] の部分が省略できます。ただし，省略した場合，as850 では，[r0] が指定されたものとみなされます。
- disp に $label を持つ相対値式，または $label を持つ相対値式に hi ( )，lo ( )，または hi1 ( ) を適用したものを指定した場合，その後ろの [reg1] の部分が省略できます。ただし，省略した場合 as850 では，[gp] が指定されたものとみなされます。
- disp にデバイス・ファイルで定義されている周辺 I/O レジスタ名を指定した場合，その後ろの [reg1] の部分が省略できます。ただし，省略した場合，as850 は [r0] が指定されたものとみなされます。

［フラグ］

CY	－
OV	－
S	－
Z	－
SAT	－

［注意事項］

- ld.b，および ld.h は，1 バイト分，および 1 ハーフワード分のデータを符号拡張し，1 ワード分のデータとしてレジスタにロードされます。
- ld.bu，および ld.hu は，1 バイト分，および 1 ハーフワード分のデータをゼロ拡張し，1 ワード分のデータとしてレジスタにロードされます。
- ld.h，ld.w，ld.hu の disp に 2 の倍数でない値を指定した場合，as850 では，disp に対して，2 でアライメントしたコードが生成され，次のいずれかのメッセージが出力されます。

W3010: illegal displacement in ld instruction.
W4659: relocated value (*value*) of relocation entry (symbol: *symbol*, file: *file*, section: *section*, offset: *offset*, type: *relocation type*) for load/store command become odd value.

- ld.bu，および ld.hu 命令に対し，第 2 オペランドに r0 を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

E3240: illegal operand (can not use r0 as destination in V850E mode)

sld

データのロードを行います。（Short format Load）

［指定形式］

- sld.b disp7[ep], reg2
- sld.h disp8[ep], reg2
- sld.w disp8[ep], reg2
- sld.bu disp4[ep], reg2 【V850E】
- sld.hu disp5[ep], reg2 【V850E】

disp4 ／ 5 ／ 7 ／ 8 に指定できるものを次に示します。

- sld.b では 7 ビット，sld.h，および sld.w では 8 ビット，sld.bu では 4 ビット，sld.hu では 5 ビット幅までの値を持つ絶対値式
- 相対値式

［機能］

sld.b，sld.h，sld.w，sld.bu，sld.hu 命令は，第 1 オペランドに指定したディスプレースメントとレジスタ ep の内容を加算して得たアドレスから，1 バイト分，1 ハーフワード分，および 1 ワード分のデータを取り込み，第 2 オペランドに指定したレジスタにロードします。

［詳細説明］

as850 では，機械語命令の sld 命令が 1 つ生成されます。ベース・レジスタの指定“[ep]”は省略できます。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

［注意事項］

- sld.b，および sld.h は，1 バイト分，および 1 ハーフワード分のデータを符号拡張し，1 ワードのデータとしてレジスタに格納されます。
- sld.bu，および sld.hu は，1 バイト分，および 1 ハーフワード分のデータをゼロ拡張し，1 ワードのデータとしてレジスタに格納されます。

- sld.h ／ sld.hu の disp8 ／ disp5 に 2 の倍数でない値を指定した場合，および sld.w の disp8 に 4 の倍数でない値を指定した場合，as850 では，disp8 ／ disp5 に対して，それぞれ 2 の倍数，4 の倍数にアライメントしたコードが生成され，次のいずれかのメッセージが出力されます。

W3010: illegal displacement in sld instruction.
W4659: relocated value (*value*) of relocation entry (symbol: *symbol*, file: *file*, section: *section*, offset: *offset*, type: *relocation type*) for load/store command become odd value.

- sld.b の disp7 に 127 を越える値を指定した場合，sld.h，sld.w の disp8 に 255 を越える値を指定した場合，sld.bu の disp4 に 16 を越える値を指定した場合，および sld.hu の disp5 に 32 を越える値を指定した場合，次のメッセージが出力され，disp7，disp8，disp4，disp5 をそれぞれ 0x7f，0xff，0xf，0x1f でマスクしたコードが生成されます。

W3011: illegal operand (range error in immediate）

- sld.bu，および sld.hu の第 2 オペランド（reg2）に r0 を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

E3240: illegal operand (can not use r0 as destination in V850E mode)

st

データのストアを行います。（Store）

［指定形式］

- st.b reg2, disp[reg1]
- st.h reg2, disp[reg1]
- st.w reg2, disp[reg1]

disp に指定できるものを，次に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式
- 上記のものに hi ( )，lo ( )，または hi1 ( ) を適用したもの

［機能］

st.b，st.h，st.w 命令は，第 1 オペランドに指定したレジスタの下位 1 バイト分，下位 1 ハーフワード分，および 1 ワード分のデータを取り込み，第 2 オペランドに指定したアドレスにストアします。

［詳細説明］

- disp に次のものを指定した場合，as850 では，機械語命令の st 命令[注]が 1 つ生成されます。なお，例中の“st”は st.b，st.h のいずれかになります。

（a）-32768 ～ +32767 の範囲の絶対値式

st reg2, disp16[reg1]	st reg2, disp16[reg1]

（b）sdata ／ sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

st reg2, $label[reg1]	st reg2, $label[reg1]

（c）!label，または %label を持つ相対値式

st reg2, !label[reg1]	st reg2, !label[reg1]
st reg2, %label[reg1]	st reg2, %label[reg1]

（d）hi ( )，lo ( )，または hi1 ( ) を適用したもの

st reg2, disp16[reg1]	st reg2, disp16[reg1]

注 機械語命令の st 命令は，ディスプレースメントに -32768 ～ +32767（0xffff8000 ～ 0x7fff）の範囲のイミーディエトをとります。

- disp に次のものを指定した場合，as850 では，命令展開が行われ，複数個の機械語命令が生成されます。

（a）-32768 ～ +32767 の範囲を越える絶対値式

st reg2, disp[reg1], reg2	movhi hi1(disp), reg1, r1 st reg2, lo(disp)[r1], reg2

（b）#label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

st reg2, #label[reg1]	movhi hi1(#label), reg1, r1 st reg2, lo(#label)[r1]
st reg2, label[reg1]	movhi hi1(label), reg1, r1 st reg2, lo(label)[r1]
st reg2, $label[reg1]	movhi hi1($label), reg1, r1 st reg2, lo($label)[r1]

- disp を省略した場合，as850 では，0 を指定したものとみなされます。
- disp に #label を持つ相対値式，または #label を持つ相対値式に hi ( )，lo ( )，または hi1 ( ) を適用したものを指定した場合，その後ろの [reg1] の部分が省略できます。ただし，省略した場合，as850 では，[r0] が指定されたものとみなされます。
- disp に $label を持つ相対値式，または $label を持つ相対値式に hi ( )，lo ( )，または hi1 ( ) を適用したものを指定した場合，その後ろの [reg1] の部分が省略できます。ただし，省略した場合，as850 では，[gp] が指定されたものとみなされます。
- disp にデバイス・ファイルで定義されている周辺 I/O レジスタ名を指定した場合，その後ろの [reg1] の部分が省略できます。ただし，省略した場合，as850 では，[r0] が指定されたものとみなされます。

［フラグ］

CY	－
OV	－
S	－
Z	－
SAT	－

［注意事項］

- st.h，および st.w の disp に 2 の倍数でない値を指定した場合，as850 では，disp に対して 2 でアライメントしたコードが生成され，次のいずれかのメッセージが出力されます。

W3010: illegal displacement in st instruction.
W4659: relocated value (*value*) of relocation entry (symbol: *symbol*, file: *file*, section: *section*, offset: *offset*, type: *relocation type*) for load/store command become odd value.

sst

データのストアを行います。（Short format Store）

［指定形式］

- sst.b reg2, disp7[ep]
- sst.h reg2, disp8[ep]
- sst.w reg2, disp8[ep]

disp7 ／ 8 に指定できるものを次に示します。

- sst.b では 7 ビット，sst.h，および sst.w では 8 ビット幅までの値を持つ絶対値式
- 相対値式

［機能］

sst.b，sst.h，sst.w 命令は，第 1 オペランドに指定したレジスタの下位 1 バイト分，下位 1 ハーフワード分，および 1 ワード分のデータを，第 2 オペランドに指定したディスプレースメントとしてレジスタ ep の内容を加算して得たアドレスに格納します。

［詳細説明］

as850 では，機械語命令の sst 命令が 1 つ生成されます。ベース・レジスタの指定“[ep]”は省略できます。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

［注意事項］

- sst.h の disp8 に 2 の倍数でない値を指定した場合，および sst.w の disp8 に 4 の倍数でない値を指定した場合，as850 では，disp8 に対して，それぞれ 2 の倍数，4 の倍数にアライメントしたコードが生成され，次のいずれかのメッセージが出力されます。

W3010: illegal displacement in sst instruction.
W4659: relocated value (*value*) of relocation entry (symbol: *symbol*, file: *file*, section: *section*, offset: *offset*, type: *relocation type*) for load/store command become odd value.

- sst.b の disp7 に 127 を越える値を指定した場合，および sst.h，sst.w の disp8 に 255 を越える値を指定した場合，次のメッセージが出力され，disp7，disp8 をそれぞれ 0x7f，0xff でマスクしたコードが生成されます。

W3011: illegal operand (range error in immediate)

4. 5. 7 算術演算命令

この項では，算術演算命令について説明します。次に，この項において説明する命令を示します。

表 4―46 算術演算命令

命令	意味
add	加算
addi	加算（イミーディエト）
adf	条件付き加算【V850E2】
sub	減算
subr	逆減算
sbf	条件付き減算【V850E2】
mulh	符号付き乗算（ハーフワード）
mulhi	符号付き乗算（ハーフワード・イミーディエト）
mul	符号付き乗算（ワード）【V850E】
mac	符号付きワード・データの加算付き乗算【V850E2】
mulu	符号なし乗算【V850E】
macu	符号なしワード・データの加算付き乗算【V850E2】
divh	符号付き除算（ハーフワード）
div	符号付き除算（ワード）【V850E】
divhu	符号なし除算（ハーフワード）【V850E】
divu	符号なし除算（ワード）【V850E】
cmp	比較
mov	データの転送
movea	実行アドレスの転送
movhi	上位ハーフワードの転送
mov32	32 ビット・データの転送【V850E】
cmov	フラグ条件付きデータの転送【V850E】
setf	フラグ条件の設定
sasf	論理左シフト付きフラグ条件の設定【V850E】

add

加算を行います。（Add）

［指定形式］

- add reg1, reg2
- add imm, reg2

imm に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式

［機能］

- “add reg1, reg2”の形式

  第 1 オペランドに指定したレジスタ値と，第 2 オペランドに指定したレジスタ値を加算し，結果を第 2 オペランドに指定したレジスタに格納します。

- “add imm, reg2”の形式

  第 1 オペランドに指定した絶対値式，または相対値式の値と第 2 オペランドに指定したレジスタ値を加算し，結果を第 2 オペランドに指定したレジスタに格納します。

［詳細説明］

- “add reg1, reg2”の形式の命令に対し，as850 では，機械語命令の add 命令が 1 つ生成されます。
- “add imm, reg2”の形式で imm に次のものを指定した場合，as850 では，機械語命令の add 命令[注]が 1 つ生成されます。

（a）-16 ～ +15 の範囲の絶対値式

`add imm5, reg`	`add imm5, reg`

注 機械語命令の add 命令は，第 1 オペランドにレジスタ，または -16 ～ +15（0xfffffff0 ～ 0xf）の範囲のイミーディエトをとります。

- “add imm, reg2”の形式で imm に次のものを指定した場合，as850 では，命令展開が行われ，1 つ，または複数個の機械語命令が生成されます。

（a）-16 ～ +15 の範囲を越え，-32768 ～ +32767 の範囲の絶対値式

`add imm16, reg`	`addi imm16, reg, reg`

（b）-32768 ～ +32767 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

add imm, reg	movhi hi(imm), r0, r1 add r1, reg

上記以外の場合

add imm, reg	movhi hi1(imm), r0, r1
	movea lo(imm), r1, r1
	add r1, reg

（c）-32768 ～ +32767 の範囲を越える絶対値式【V850E】

imm の値の下位 16 ビットがすべて 0 の場合

add imm, reg	movhi hi(imm), r0, r1 add r1, reg

上記以外の場合

add imm, reg	mov imm, r1 add r1, reg

（d）!label，または %label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

add !label, reg	addi !label, reg, reg
add %label, reg	addi %label, reg, reg
add $label, reg	addi $label, reg, reg

（e）#label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

add #label, reg	movhi hi1(#label), r0, r1
	movea lo(#label), r1, r1
	add r1, reg

add label, reg	movhi hi1(label), r0, r1
	movea lo(label), r1, r1
	add r1, reg

add $label, reg	movhi hi1($label), r0, r1
	movea lo($label), r1, r1
	add r1, reg

（f）#label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式【V850E】

`add #label, reg`	`mov #label, r1` `add r1, reg`
`add label, reg`	`mov label, r1` `add r1, reg`
`add $label, reg`	`mov $label, r1` `add r1, reg`

［フラグ］

CY	MSB（Most Significant Bit）からのキャリーを生じた場合 1，そうでない場合 0
OV	Integer-Overflow を生じた場合 1，そうでない場合 0
S	結果が負になった場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	−

addi

加算を行います。（Add Immediate）

［指定形式］

- addi imm, reg1, reg2

imm に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式
- 上記のものに hi ( )，lo ( )，または hi1 ( ) を適用したもの

［機能］

第 1 オペランドに指定した絶対値式，相対値式，あるいは，hi ( )，lo ( )，または hi1 ( ) を適用した値と第 2 オペランドに指定したレジスタ値を加算し，結果を第 3 オペランドに指定したレジスタに格納します。

［詳細説明］

- imm に次のものを指定した場合，as850 では，機械語命令の addi 命令[注]が 1 つ生成されます。

（a）-32768 ～ +32767 の範囲の絶対値式

addi imm16, reg1, reg2	addi imm16, reg1, reg2

（b）sdata ／ sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

addi $label, reg1, reg2	addi $label, reg1, reg2

（c）!label，または %label を持つ相対値式

addi !label, reg1, reg2	addi !label, reg1, reg2
addi %label, reg1, reg2	addi %label, reg1, reg2

（d）hi ( )，lo ( )，または hi1 ( ) を適用したもの

addi imm16, reg1, reg2	addi imm16, reg1, reg2

注 機械語命令の addi 命令は，第 1 オペランドに -32768 ～ +32767（0xffff8000 ～ 0x7fff）の範囲のイミーディエトをとります。

- imm に次のものを指定した場合，as850 では，命令展開が行われ，複数個の機械語命令が生成されます。

（a）-32768 ～ +32767 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

addi imm, reg1, reg2	movhi hi(imm), r0, reg2 add reg1, reg2

imm の値の下位 16 ビットがすべて 0，ただし reg2 が r0 の場合

addi imm, reg1, r0	movhi hi(imm), r0, r1 add reg1, r1

上記以外の場合

addi imm, reg1, reg2	movhi hi1(imm), r0, r1
	movea lo(imm), r1, reg2
	add reg1, reg2

上記以外，ただし reg2 が r0 の場合

addi imm, reg1, r0	movhi hi1(imm), r0, r1
	movea lo(imm), r1, r1
	add reg1, r1

（b）-32768 ～ +32767 の範囲を越える絶対値式【V850E】

imm の値の下位 16 ビットがすべて 0 の場合

addi imm, reg1, reg2	movhi hi(imm), r0, reg2 add reg1, reg2

imm の値の下位 16 ビットがすべて 0，ただし reg2 が r0 の場合

addi imm, reg1, r0	movhi hi(imm), r0, r1 add reg1, r1

上記以外の場合

addi imm, reg1, reg2	mov imm, reg2 add reg1, reg2

上記以外，ただし reg2 が r0 の場合

addi imm, reg1, r0	mov imm, r1 add reg1, r1

（c）label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

reg2 が r0 の場合

addi #label, reg1, r0	movhi hi1(#label), r0, r1
	movea lo(#label), r1, r1
	add reg1, reg2

addi label, reg1, r0	movhi hi1(label), r0, r1
	movea lo(label), r1, r1
	add reg1, r1

addi $label, reg1 r0	movhi hi1($label), r0, r1
	movea lo($label), r1, r1
	add reg1, r1

上記以外の場合

addi #label, reg1, reg2	movhi hi1(#label), r0, r1
	movea lo(#label), r1, reg2
	add reg1, reg2

addi label, reg1, reg2	movhi hi1(label), r0, r1
	movea lo(label), r1, reg2
	add reg1, reg2

addi $label, reg1 reg2	movhi hi1($label), r0, r1
	movea lo($label), r1, reg2
	add reg1, reg2

（d）#label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式【V850E】

reg2 が r0 の場合

addi #label, reg1, r0	mov #label, r1 addi reg1, r1
addi label, reg1, r0	mov label, r1 add reg1, r1
addi $label, reg1, r0	mov $label, r1 add reg1, r1

上記以外の場合

addi　#label, reg1, reg2	mov　#label, reg2 addi　reg1, reg2
addi　label, reg1, reg2	mov　label, reg2 add　reg1, reg2
addi　$label, reg1, reg2	mov　$label, reg2 add　reg1, reg2

［フラグ］

CY	MSB（Most Significant Bit）からのキャリーを生じた場合 1，そうでない場合 0
OV	Integer-Overflow を生じた場合 1，そうでない場合 0
S	結果が負になった場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	—

adf

条件付き加算を行います。（Add on Condition Flag）【V850E2】

［指定形式］

- adf　imm4, reg1, reg2, reg3
- adf*cnd*　reg1, reg2, reg3

imm4 に指定できるものを次に示します。

- 4 ビット幅までの値を持つ絶対値式（0xd は指定できません）

［機能］

-“adf imm4, reg1, reg2, reg3”の形式

　第 1 オペランドに指定した絶対値式の下位 4 ビットの値で示されるフラグ状態（「表 4―47　adfcnd 命令」を参照）と現在のフラグ状態を比較します。

　値が一致した場合には，第 3 オペランドに指定したレジスタのワード・データに，第 2 オペランドに指定したレジスタのワード・データを加算し，さらに 1 を加算します。結果を第 4 オペランドに指定したレジスタに格納します。

　値が一致しなかった場合には，第 3 オペランドに指定したレジスタのワード・データに，第 2 オペランドに指定したレジスタのワード・データを加算します。その結果を第 4 オペランドに指定したレジスタに格納します。

-“adf*cnd* reg1, reg2, reg3”の形式

　cnd 部分の文字列で示されるフラグ状態と現在のフラグの状態を比較します。

　値が一致した場合に第 2 オペランドに指定したレジスタのワード・データに，第 1 オペランドに指定したレジスタのワード・データを加算し，さらに 1 を加算します。その結果を第 3 オペランドに指定したレジスタに格納します。

　値が一致しなかった場合には，第 2 オペランドに指定したレジスタのワード・データに，第 1 オペランドに指定したレジスタのワード・データを加算します。その結果を第 3 オペランドに指定したレジスタに格納します。

［詳細説明］

- adf 命令に対し，as850 では，機械語命令の adf 命令が 1 つ生成されます。
- adf*cnd* 命令に対し，as850 では，対応する adf 命令が生成され（「表 4―47　adfcnd 命令」を参照），“adf imm4, reg1, reg2, reg3”の形式に展開されます。

表 4―47　adf*cnd* 命令

命令	フラグ状態	フラグ状態の意味	命令展開
adfgt	((S xor OV) or Z) = 0	Greater than（signed）	adf 0xf

命令	フラグ状態	フラグ状態の意味	命令展開
adfge	（S xor OV）= 0	Greater than or equal（signed）	adf 0xe
adflt	（S xor OV）= 1	Less than（signed）	adf 0x6
adfle	((S xor OV）or Z）= 1	Less than or equal（signed）	adf 0x7
adfh	（CY or Z）= 0	Higher（Greater than）	adf 0xb
adfnl	CY = 0	Not lower（Greater than or equal）	adf 0x9
adfl	CY = 1	Lower（Less than）	adf 0x1
adfnh	（CY or Z）= 1	Not higher（Less than or equal）	adf 0x3
adfe	Z = 1	Equal	adf 0x2
adfne	Z = 0	Not equal	adf 0xa
adfv	OV = 1	Overflow	adf 0x0
adfnv	OV = 0	No overflow	adf 0x8
adfn	S = 1	Negative	adf 0x4
adfp	S = 0	Positive	adf 0xc
adfc	CY = 1	Carry	adf 0x1
adfnc	CY = 0	No carry	adf 0x9
adfz	Z = 1	Zero	adf 0x2
adfnz	Z = 0	Not zero	adf 0xa
adft	always 1	Always 1	adf 0x5

[フラグ]

CY	MSB からのキャリーがあれば 1，そうでない場合 0
OV	オーバーフローが起こった場合 1，そうでない場合 0
S	演算結果が負の場合 1，そうでない場合 0
Z	演算結果が 0 の場合 1，そうでない場合 0
SAT	—

[注意事項]

- adf 命令の imm4 に 4 ビットの範囲を越える絶対値式を指定した場合，次のメッセージが出力され，指定された値の下位 4 ビットが用いられてアセンブルが続行されます。

```
W3011: illegal operand (range error in immediate).
```

- adf 命令の imm4 に 0xd を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E3261: illegal condition code.
```

sub

減算を行います。（Subtract）

［指定形式］

- sub reg1, reg2
- sub imm, reg2

imm に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式

［機能］

-“sub reg1, reg2”の形式

第 2 オペランドに指定したレジスタ値から，第 1 オペランドに指定したレジスタ値を減算し，結果を第 2 オペランドに指定したレジスタに格納します。

-“sub imm, reg2”の形式

第 2 オペランドに指定したレジスタ値から，第 1 オペランドに指定した絶対値式，または相対値式の値を減算し，結果を第 2 オペランドに指定したレジスタに格納します。

［詳細説明］

-“sub reg1, reg2”の形式の命令に対し，as850 では，機械語命令の sub 命令が 1 つ生成されます。

-“sub imm, reg2”の形式で imm に次のものを指定した場合，as850 では，命令展開が行われ，1 つ，または複数個の機械語命令が生成されます。注

（a）0

`sub 0, reg`	`sub r0, reg`

（b）-16 ～ +15 の範囲の絶対値式（0 以外）

`sub imm5, reg`	`mov imm5, r1` `sub r1, reg`

（c）-16 ～ +15 の範囲を越え，-32768 ～ +32767 の範囲の絶対値式

`sub imm16, reg`	`movea imm16, r0, r1` `sub r1, reg`

（d）imm に -32768 ～ +32767 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

sub imm, reg	movhi hi(imm), r0, r1 sub r1, reg

上記以外の場合

sub imm, reg	movhi hi1(imm), r0, r1
	movea lo(imm), r1, r1
	sub r1, reg

（e）imm に -32768 ～ +32767 の範囲を越える絶対値式【V850E】

imm の値の下位 16 ビットがすべて 0 の場合

sub imm, reg	movhi hi(imm), r0, r1 sub r1, reg

上記以外の場合

sub imm, reg	mov imm, r1 sub r1, reg

（f）sdata ／ sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

sub $label, reg	movea $label, r0, r1 sub r1, reg

（g）#label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

sub #label, reg	movhi hi1(#label), r0, r1
	movea lo(#label), r1, r1
	sub r1, reg

sub label, reg	movhi hi1(label), r0, r1
	movea lo(label), r1, r1
	sub r1, reg

sub $label, reg	movhi hi1($label), r0, r1
	movea lo($label), r1, r1
	sub r1, reg

（h）#label，または label を持つ相対値式，および sdata／sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式【V850E】

`sub #label, reg`	`mov #label, r1` `sub r1, reg`
`sub label, reg`	`mov label, r1` `sub r1, reg`
`sub $label, reg`	`mov $label, r1` `sub r1, reg`

注 機械語命令の sub 命令は，オペランドにイミーディエトをとりません。

［フラグ］

CY	MSB（Most Significant Bit）へのボローが生じた場合 1，そうでない場合 0
OV	Integer-Overflow を生じた場合 1，そうでない場合 0
S	結果が負になった場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	−

subr

逆減算を行います。（Subtract Reverse）

［指定形式］

- subr reg1, reg2
- subr imm, reg2

imm に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式

［機能］

- “subr reg1, reg2”の形式

  第 1 オペランドに指定したレジスタ値から，第 2 オペランドに指定したレジスタ値を減算し，結果を第 2 オペランドに指定したレジスタに格納します。

- “subr imm, reg2”の形式

  第 1 オペランドに指定した絶対値式，または相対値式の値から，第 2 オペランドに指定したレジスタ値を減算し，結果を第 2 オペランドに指定したレジスタに格納します。

［詳細説明］

- “subr reg1, reg2”の形式の命令に対し，as850 では，機械語命令の subr 命令が 1 つ生成されます。
- “subr imm, reg2”の形式で imm に次のものを指定した場合，as850 では，命令展開が行われ，1 つ，または複数個の機械語命令が生成されます。**注**

（a）0

`subr 0, reg`	`subr r0, reg`

（b）-16 ～ +15 の範囲の絶対値式（0 以外）

`subr imm5, reg`	`mov imm5, r1` `subr r1, reg`

（c）-16 ～ +15 の範囲を越え，-32768 ～ +32767 の範囲の絶対値式

`subr imm16, reg`	`movea imm16, r0, r1` `subr r1, reg`

（d）imm に -32768 ～ +32767 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

`subr imm, reg`	`movhi hi(imm), r0, r1` `subr r1, reg`

上記以外の場合

`subr imm, reg`	`movhi hi1(imm), r0, r1`
	`movea lo(imm), r1, r1`
	`subr r1, reg`

（e）imm に -32768 ～ +32767 の範囲を越える絶対値式【V850E】

imm の値の下位 16 ビットがすべて 0 の場合

`subr imm, reg`	`movhi hi(imm), r0, r1` `subr r1, reg`

上記以外の場合

`subr imm, reg`	`mov imm, r1` `subr r1, reg`

（f）sdata ／ sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

`subr $label, reg`	`movea $label, r0, r1` `subr r1, reg`

（g）#label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

`subr #label, reg`	`movhi hi1(#label), r0, r1`
	`movea lo(#label), r1, r1`
	`subr r1, reg`

`subr label, reg`	`movhi hi1(label), r0, r1`
	`movea lo(label), r1, r1`
	`subr r1, reg`

`subr $label, reg`	`movhi hi1($label), r0, r1`
	`movea lo($label), r1, r1`
	`subr r1, reg`

（h）#label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式【V850E】

`subr #label, reg`	`mov #label, r1` `subr r1, reg`
`subr label, reg`	`mov label, r1` `subr r1, reg`
`subr $label, reg`	`mov $label, r1` `subr r1, reg`

注 機械語命令の subr 命令は，オペランドにイミーディエトをとりません。

［フラグ］

CY	MSB（Most Significant Bit）へのボローが生じた場合 1，そうでない場合 0
OV	Integer-Overflow を生じた場合 1，そうでない場合 0
S	結果が負になった場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	−

sbf

条件付き減算を行います。（Subtract on Condition Flag）【V850E2】

［指定形式］

- sbf imm4, reg1, reg2, reg3
- sbf*cnd* reg1, reg2, reg3

imm4 に指定できるものを次に示します。

- 4 ビット幅までの値を持つ絶対値式（0xd は指定できません）

［機能］

- “sbf imm4, reg1, reg2, reg3”の形式

  第 1 オペランドに指定した絶対値式の下位 4 ビットの値で示されるフラグ状態（「表 4―48 sbfcnd 命令」を参照）と現在のフラグ状態を比較します。

  値が一致した場合には，第 3 オペランドに指定したレジスタのワード・データから，第 2 オペランドに指定したレジスタのワード・データを減算し , さらに 1 を減算します。その結果を第 4 オペランドに指定したレジスタに格納します。

  値が一致しなかった場合には，第 3 オペランドに指定したレジスタのワード・データから，第 2 オペランドに指定したレジスタのワード・データを減算します。その結果を第 4 オペランドに指定したレジスタに格納します。

- “sbf*cnd* reg1, reg2, reg3”の形式

  cnd 部分の文字列で示されるフラグ状態と現在のフラグの状態を比較します。

  値が一致した場合には，第 2 オペランドに指定したレジスタのワード・データから，第 1 オペランドに指定したレジスタのワード・データを減算し，さらに 1 を減算します。その結果を第 3 オペランドに指定したレジスタに格納します。

  値が一致しなかった場合には，第 2 オペランドに指定したレジスタのワード・データから，第 1 オペランドに指定したレジスタのワード・データを減算します。その結果を第 3 オペランドに指定したレジスタに格納します。

［詳細説明］

- sbf 命令に対し，as850 では，機械語命令の sbf 命令が 1 つ生成されます。
- sbf*cnd* 命令に対し，as850 では，対応する sbf 命令が生成され（「表 4―48 sbfcnd 命令」を参照），“subr reg1, reg2”の形式に展開されます。

表 4―48 sbf*cnd* 命令

命令	フラグ状態	フラグ状態の意味	命令展開
sbfgt	((S xor OV) or Z) = 0	Greater than（signed）	sbf 0xf

命令	フラグ状態	フラグ状態の意味	命令展開
sbfge	（S xor OV）= 0	Greater than or equal（signed）	sbf 0xe
sbflt	（S xor OV）= 1	Less than（signed）	sbf 0x6
sbfle	((S xor OV）or Z）= 1	Less than or equal（signed）	sbf 0x7
sbfh	（CY or Z）= 0	Higher（Greater than）	sbf 0xb
sbfnl	CY = 0	Not lower（Greater than or equal）	sbf 0x9
sbfl	CY = 1	Lower（Less than）	sbf 0x1
sbfnh	（CY or Z）= 1	Not higher（Less than or equal）	sbf 0x3
sbfe	Z = 1	Equal	sbf 0x2
sbfne	Z = 0	Not equal	sbf 0xa
sbfv	OV = 1	Overflow	sbf 0x0
sbfnv	OV = 0	No overflow	sbf 0x8
sbfn	S = 1	Negative	sbf 0x4
sbfp	S = 0	Positive	sbf 0xc
sbfc	CY = 1	Carry	sbf 0x1
sbfnc	CY = 0	No carry	sbf 0x9
sbfz	Z = 1	Zero	sbf 0x2
sbfnz	Z = 0	Not zero	sbf 0xa
sbft	always 1	Always 1	sbf 0x5

[フラグ]

CY	MSB からのボローがあれば 1，そうでない場合 0
OV	オーバーフローが起こった場合 1，そうでない場合 0
S	演算結果が負の場合 1，そうでない場合 0
Z	演算結果が 0 の場合 1，そうでない場合 0
SAT	—

[注意事項]

- sbf 命令の imm4 に 4 ビットの範囲を越える絶対値式を指定した場合，次のメッセージが出力され，指定された値の下位 4 ビットが用いられてアセンブルが続行されます。

```
W3011: illegal operand (range error in immediate).
```

- sbf 命令の imm4 に 0xd を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E3261: illegal condition code.
```

mulh

符号付き乗算（ハーフワード）を行います。（Multiply Half-word）

［指定形式］

- mulh reg1, reg2
- mulh imm, reg2

imm に指定可能なものを以下に示します。

- 16 ビット幅までの値を持つ絶対値式[注]
- 相対値式

注 as850 では，16 ビットを越えるかどうかチェックは行われません。生成された機械語命令 mulh 命令では，下位 16 ビットを用いて演算が行われます。

［機能］

- “mulh reg1, reg2”の形式

  第 1 オペランドに指定したレジスタの下位ハーフワード・データの値と，第 2 オペランドに指定したレジスタの下位ハーフワード・データの値を，符号付きの値として乗算し，結果を第 2 オペランドに指定したレジスタに格納します。
- “mulh imm, reg2”の形式

  第 1 オペランドに指定した絶対値式，または相対値式の下位ハーフワード・データの値と，第 2 オペランドに指定したレジスタの下位ハーフワード・データの値を，符号付きの値として乗算し，結果を第 2 オペランドに指定したレジスタに格納します。

［詳細説明］

- “mulh reg1, reg2”の形式の命令に対し，as850 では，機械語命令の mulh 命令が 1 つ生成されます。
- “mulh imm, reg2”の形式で imm に次のものを指定した場合，as850 では，機械語命令の add 命令[注]が 1 つ生成されます。

（a）-16 ～ +15 の範囲の絶対値式

`mulh imm5, reg`	`mulh imm5, reg`

注 機械語命令の add 命令は，第 1 オペランドにレジスタ，または -16 ～ +15（0xfffffff0 ～ 0xf）の範囲のイミーディエトをとります。

- “mulh imm, reg2”の形式で imm に次のものを指定した場合，as850 では，命令展開が行われ，1 つ，または複数個の機械語命令が生成されます。

（a）-16 ～ +15 の範囲を越える絶対値式

mulh imm16, reg	mulhi imm16, reg, reg

（b）-32768 ～ +32767 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

mulh imm, reg	movhi hi(imm), r0, r1 mulh r1, reg

上記以外の場合

mulh imm, reg	movhi hi1(imm), r0, r1
	movea lo(imm), r1, r1
	mulh r1, reg

（c）-32768 ～ +32767 の範囲を越える絶対値式【V850E】

imm の値の下位 16 ビットがすべて 0 の場合

mulh imm, reg	movhi hi(imm), r0, r1 mulh r1, reg

上記以外の場合

mulh imm, reg	mov imm, r1 mulh r1, reg

（d）!label，または %label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

mulh !label, reg	mulhi !label, reg, reg
mulh %label, reg	mulhi %label, reg, reg
mulh $label, reg	mulhi $label, reg, reg

（e）#label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

`mulh #label, reg`	`movhi hi1(#label), r0, r1`
	`movea lo(#label), r1, r1`
	`mulh r1, reg`

`mulh label, reg`	`movhi hi1(label), r0, r1`
	`movea lo(label), r1, r1`
	`mulh r1, reg`

`mulh $label, reg`	`movhi hi1($label), r0, r1`
	`movea lo($label), r1, r1`
	`mulh r1, reg`

（f）#label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式【V850E】

`mulh #label, reg`	`mov #label, r1` `mulh r1, reg`
`mulh label, reg`	`mov label, r1` `mulh r1, reg`
`mulh $label, reg`	`mov $label, r1` `mulh r1, reg`

［フラグ］

CY	—
OV	—
S	—
Z	—
SAT	—

［注意事項］

- ターゲット・デバイスが V850Ex の場合，第 2 オペランドに r0 を指定すると，次のメッセージが出力され，アセンブルが中止されます。

```
E3240: illegal operand (can not use r0 as destination in V850E mode）
```

- ターゲット・デバイスが V850Ex 以外の場合，第 2 オペランドに r0 を指定すると，次のメッセージが出力され，アセンブルが続行されます。

```
W3013: register r0 used as destination register
```

mulhi

符号付き乗算（ハーフワード）を行います。（Multiply Half-word）

［指定形式］

- mulhi imm, reg1, reg2

imm に指定可能なものを以下に示します。

- 16 ビット幅までの値を持つ絶対値式注
- 相対値式
- 上記のものに hi ( )，lo ( )，または hi1 ( ) を適用したもの

注 as850 では，16 ビットを越えるかどうかチェックは行われません。生成された機械語命令 mulhi 命令では，下位 16 ビットを用いて演算が行われます。

［機能］

第 1 オペランドに指定した絶対値式，相対値式，あるいは，hi ( )，lo ( )，または hi1 ( ) を適用した値と，第 2 オペランドに指定したレジスタ値を符号付きの値として乗算し，結果を第 3 オペランドに指定したレジスタに格納します。

［詳細説明］

- imm に次のものを指定した場合，as850 では，機械語命令の mulhi 命令注が 1 つ生成されます。

（a）-32768 ～ +32767 の範囲の絶対値式

`mulhi imm16, reg1, reg2`	`mulhi imm16, reg1, reg2`

（b）sdata ／ sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

`mulhi $label, reg1, reg2`	`mulhi $label, reg1, reg2`

（c）!label，または %label を持つ相対値式

`mulhi !label, reg1, reg2`	`mulhi !label, reg1, reg2`
`mulhi %label, reg1, reg2`	`mulhi %label, reg1, reg2`

（d）hi ( )，lo ( )，または hi1 ( ) を適用したもの

`mulhi imm16, reg1, reg2`	`mulhi imm16, reg1, reg2`

注 機械語命令の mulhi 命令は，第 1 オペランドに -32768 ～ +32767（0xffff8000 ～ 0x7fff）の範囲のイミーディエトをとります。

- imm に次のものを指定した場合，as850 では，命令展開が行われ，複数個の機械語命令が生成されます。

（a）-32768 ～ +32767 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

`mulhi imm, reg1, reg2`	`movhi hi(imm), r0, reg2` `mulh reg1, reg2`

imm の値の下位 16 ビットがすべて 0，ただし reg2 が r0 の場合

`mulhi imm, reg1, r0`	`movhi hi(imm), r0, r1` `mulh reg1, r1`

上記以外の場合

`mulhi imm, reg1, reg2`	`movhi hi1(imm), r0, r1`
	`movea lo(imm), r1, reg2`
	`mulh reg1, reg2`

上記以外，ただし reg2 が r0 の場合

`mulhi imm, reg1, reg2`	`movhi hi1(imm), r0, r1`
	`movea lo(imm), r1, r1`
	`mulh reg1,r1`

（b）-32768 ～ +32767 の範囲を越える絶対値式【V850E】

imm の値の下位 16 ビットがすべて 0 の場合

`mulhi imm, reg1, reg2`	`movhi hi(imm), r0, reg2` `mulh reg1, reg2`

上記以外の場合

`mulhi imm, reg1, reg2`	`mov imm, reg2` `mulh reg1, reg2`

（c）#label，または label を持つ相対値式，および sdata／sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

reg2 が r0 の場合

`mulhi #label, reg1, r0`	`movhi hi1(#label), r0, r1`
	`movea lo(#label), r1, r1`
	`mulh reg1, r1`

`mulhi label, reg1, r0`	`movhi hi1(label), r0, r1`
	`movea lo(label), r1, r1`
	`mulh reg1, r1`

`mulhi $label, reg1 r0`	`movhi hi1($label), r0, r1`
	`movea lo($label), r1, r1`
	`mulh reg1, r1`

上記以外の場合

`mulhi #label, reg1, reg2`	`movhi hi1(#label), r0, r1`
	`movea lo(#label), r1, reg2`
	`mulh reg1, reg2`

`mulhi label, reg1, reg2`	`movhi hi1(label), r0, r1`
	`movea lo(label), r1, reg2`
	`mulh reg1, reg2`

`mulhi $label, reg1 reg2`	`movhi hi1($label), r0, r1`
	`movea lo($label), r1, reg2`
	`mulh reg1, reg2`

（d）#label，または label を持つ相対値式，および sdata／sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式【V850E】

`mulhi #label, reg1, reg2`	`mov #label, reg2` `mulhi reg1, reg2`
`mulhi label, reg1, reg2`	`mov label, reg2` `mulh reg1, reg2`
`mulhi $label, reg1, reg2`	`mov $label, reg2` `mulh reg1, reg2`

［フラグ］

CY	－
OV	－
S	－
Z	－
SAT	－

［注意事項］

- ターゲット・デバイスが V850Ex の場合，第 2 オペランドに r0 を指定すると，次のメッセージが出力され，アセンブルが中止されます。

```
E3240: illegal operand (can not use r0 as destination in V850E mode）
```

- ターゲット・デバイスが V850Ex 以外の場合，第 2 オペランドに r0 を指定すると，次のメッセージが出力され，アセンブルが続行されます。

```
W3013: register r0 used as destination register
```

mul

符号付き乗算（ワード）を行います。（Multiply Word）【V850E】

［指定形式］

- mul reg1, reg2, reg3
- mul imm, reg2, reg3

imm に指定可能なものを以下に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式

［機能］

- “mul reg1, reg2, reg3”の形式

  第 1 オペランドに指定したレジスタ値と，第 2 オペランドに指定したレジスタ値を，符号付きの値として乗算し，結果の下位 32 ビットを第 2 オペランドに指定したレジスタに，上位 32 ビットを第 3 オペランドに指定したレジスタに格納します。第 2 オペランドと第 3 オペランドが同じレジスタの場合，レジスタには乗算結果の上位 32 ビットを格納します。

- “mul imm, reg2, reg3”の形式

  第 1 オペランドに指定した絶対値式，または相対値式の値と，第 2 オペランドに指定したレジスタの値を，符号付きの値として乗算し，結果の下位 32 ビットを第 2 オペランドに指定したレジスタに，上位 32 ビットを第 3 オペランドに指定したレジスタに格納します。第 2 オペランドと第 3 オペランドが同じレジスタの場合，レジスタには乗算結果の上位 32 ビットを格納します。

［詳細説明］

- “mul reg1, reg2, reg3”の形式の命令に対し，as850 では，機械語命令の mul 命令が 1 つ生成されます。
- “mul imm, reg2, reg3”の形式で imm に次のものを指定した場合，as850 では，命令展開が行われ，1 つ，または複数個の機械語命令が生成されます。

（a）0

`mul 0, reg2, reg3`	`mul r0, reg2, reg3`

（b）0 以外の -256 ～ +255 の範囲の絶対値式

`mul imm9, reg2, reg3`	`mul imm9, reg2, reg3`

（c）-256 ～ +255 の範囲を越え，-32768 ～ +32767 の範囲の絶対値式

mul imm16, reg2, reg3	movea imm16, r0, r1 mul r1, reg2, reg3

（d）-32768 ～ +32767 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

mul imm, reg2, reg3	movhi hi(imm), r0, r1 mul r1, reg2, reg3

上記以外の場合

mul imm, reg2, reg3	mov imm, r1 mul r1, reg2, reg3

（e）sdata ／ sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

mul $label, reg2, reg3	movea $label, r0, r1 mul r1, reg2, reg3

（f）#label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

mul #label, reg2, reg3	mov #label, r1 mul r1, reg2, reg3
mul label, reg2, reg3	mov label, r1 mul r1, reg2, reg3
mul $label, reg2, reg3	mov $label, r1 mul r1, reg2, reg3

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

［注意事項］

- “mul reg1, reg2, reg3”の形式の命令に対して“reg1 と reg3 は同一のレジスタである”，“reg2 は reg1, reg3 と異なるレジスタである”，“reg1，reg3 が r0，または r1 ではない”の条件をすべて満たす場合，as850 は命令展開を行い，複数個の機械語命令を生成します。

```
mov     reg1, r1
mul     r1, reg2, reg3
```

- “mul reg1, reg2, reg3”の形式の命令に対し，“reg1, reg3 が r1 である”，“reg2 は reg1, reg3 と異なるレジスタである”の条件をすべて満たす場合，as850 は次のメッセージを出力し，アセンブルを中止します。

```
W3013: register r1 used as source register
W3013: register r1 used as destination register
E3259: can not use r1 as destination in mul/mulu
```

- “mul imm, reg2, reg3”の形式の命令に対し，“reg2 と reg3 が r1 のレジスタである”の条件をて満たす場合，as850 は次のメッセージを出力し，アセンブルを中止します。

```
W3013: register r1 used as source register
W3013: register r1 used as destination register
E3259: can not use r1 as destination in mul/mulu
```

- 警告メッセージ抑止オプション -wr1- を指定した場合，次のメッセージを出力し，アセンブルを中止します。

```
E3259: can not use r1 as destination in mul/mulu
```

mac

符号付きワード・データの加算付き乗算を行います。（Multiply Word and Add）【V850E2】

［指定形式］

- mac reg1, reg2, reg3, reg4

［機能］

汎用レジスタ reg2 のワード・データに，汎用レジスタ reg1 のワード・データを乗算した結果（64 ビット・データ）と，汎用レジスタ reg3 を下位 32 ビットとして，汎用レジスタ reg3+1（たとえば，reg3 が r6 の場合，「reg3+1」は r7 となります）を上位 32 ビットとして結合した 64 ビット・データを加算し，その結果（64 ビット・データ）の上位 32 ビットを汎用レジスタ reg4+1 に，下位 32 ビットを汎用レジスタ reg4 に格納します。

汎用レジスタ reg1，reg2 の内容を 32 ビットの符号付き整数として扱います。

汎用レジスタ reg1，reg2，reg3，reg3+1 は影響を受けません。

［詳細説明］

as850 では，機械語命令の mac 命令が 1 つ生成されます。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

［注意事項］

- reg3，または reg4 に指定できる汎用レジスタは，偶数番号の付いたレジスタ（r0，r2，r4，…，r30）だけです。奇数番号の付いたレジスタ（r1，r3，…，r31）を指定した場合は，次のメッセージが出力され，偶数番号の付いたレジスタ（r0，r2，r4，…，r30）を指定したとして，アセンブルが続行されます。

```
W3026: illegal register number, aligned odd register(rXX) to be even register(rYY).
```

mulu

符号なし乗算（ワード）を行います。（Multiply Word Unsigned）【V850E】

［指定形式］

- mulu reg1, reg2, reg3
- mulu imm, reg2, reg3

imm に指定可能なものを以下に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式

［機能］

- “mulu reg1, reg2, reg3”の形式

  第 1 オペランドに指定したレジスタ値と，第 2 オペランドに指定したレジスタ値を，符号なしの値として乗算し，結果の下位 32 ビットを第 2 オペランドに指定したレジスタに，上位 32 ビットを第 3 オペランドに指定したレジスタに格納します。第 2 オペランドと第 3 オペランドが同じレジスタの場合，レジスタには結果の上位 32 ビットを格納します。

- “mulu imm, reg2, reg3”の形式

  第 1 オペランドに指定した絶対値式，または相対値式の値と，第 2 オペランドに指定したレジスタの値を，符号なしの値として乗算し，結果の下位 32 ビットを第 2 オペランドに指定したレジスタに，上位 32 ビットを第 3 オペランドに指定したレジスタに格納します。第 2 オペランドと第 3 オペランドが同じレジスタの場合，レジスタには結果の上位 32 ビットを格納します。

［詳細説明］

- “mulu reg1, reg2, reg3”の形式の命令に対し，as850 では，機械語命令の mulu 命令が 1 つ生成されます。
- “mulu imm, reg2, reg3”の形式で imm に次のものを指定した場合，as850 では，命令展開が行われ，1 つ，または複数個の機械語命令が生成されます。

（a）0

`mulu 0, reg2, reg3`	`mulu r0, reg2, reg3`

（b）0 以外の 0 ～ 511 の範囲

`mulu imm9, reg2, reg3`	`mulu imm9, reg2, reg3`

（c）0 ～ 511 の範囲を越え，0 ～ 65535 の範囲の絶対値式

`mulu imm16, reg2, reg3`	`movea imm16, r0, r1` `mulu r1, reg2, reg3`

（d）0 ～ 65535 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

`mulu imm, reg2, reg3`	`movhi hi(imm), r0, r1` `mulu r1, reg2, reg3`

上記以外の場合

`mulu imm, reg2, reg3`	`mov imm, r1` `mulu r1, reg2, reg3`

（e）sdata ／ sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

`mulu $label, reg2, reg3`	`movea $label, r0, r1` `mulu r1, reg2, reg3`

（f）#label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

`mulu #label, reg2, reg3`	`mov #label, r1` `mulu r1, reg2, reg3`
`mulu label, reg2, reg3`	`mov label, r1` `mulu r1, reg2, reg3`
`mulu $label, reg2, reg3`	`mov $label, r1` `mulu r1, reg2, reg3`

［フラグ］

CY	—
OV	—
S	—
Z	—
SAT	—

［注意事項］

- “mulu reg1, reg2, reg3”の形式の命令に対して“reg1 と reg3 は同一のレジスタである”，“reg2 は reg1, reg3 と異なるレジスタである”，“reg1，reg3 が r0，または r1 ではない”の条件をすべて満たす場合，as850 は命令展開を行い，複数個の機械語命令を生成します。

```
mov     reg1, r1
mulu    r1, reg2, reg3
```

- “mulu reg1, reg2, reg3”の形式の命令に対し，“reg1 と reg3 は同一のレジスタである”，“reg2 は reg1, reg3 と異なるレジスタである”，“reg1, reg3 が r1 である”の条件をすべて満たす場合，as850 は次のメッセージを出力し，アセンブルを中止します。

W3013: register r1 used as source register
W3013: register r1 used as destination register
E3259: can not use r1 as destination in mul/mulu

- “mulu imm, reg2, reg3”の形式の命令に対し，“reg2 と reg3 が同一のレジスタである”，“reg3 が r1 である”の条件をすべて満たす場合，as850 は次のメッセージを出力し，アセンブルを中止します。

W3013: register r1 used as destination register
E3259: can not use r1 as destination in mul/mulu

- 警告メッセージ抑止オプション -wr1- を指定した場合，次のメッセージを出力し，アセンブルを中止します。

E3259: can not use r1 as destination in mul/mulu

macu

符号なしワード・データの加算付き乗算を行います。（Multiply Word Unsigned and Add）【V850E2】

［指定形式］

- macu reg1, reg2, reg3, reg4

［機能］

汎用レジスタ reg2 のワード・データに，汎用レジスタ reg1 のワード・データを乗算した結果（64 ビット・データ）と，汎用レジスタ reg3 を下位 32 ビットとして，汎用レジスタ reg3+1（たとえば，reg3 が r6 の場合，「reg3+1」は r7 となります）を上位 32 ビットとして結合した 64 ビット・データを加算し，その結果（64 ビット・データ）の上位 32 ビットを汎用レジスタ reg4+1 に，下位 32 ビットを汎用レジスタ reg4 に格納します。

汎用レジスタ reg1，reg2 の内容を 32 ビットの符号なし整数として扱います。

汎用レジスタ reg1，reg2，reg3，reg3+1 は影響を受けません。

［詳細説明］

as850 では，機械語命令の macu 命令が 1 つ生成されます。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

［注意事項］

- reg3，または reg4 に指定できる汎用レジスタは，偶数番号の付いたレジスタ（r0，r2，r4，…，r30）だけです。奇数番号の付いたレジスタ（r1，r3，…，r31）を指定した場合は，次のメッセージが出力され，偶数番号の付いたレジスタ（r0，r2，r4，…，r30）を指定したとして，アセンブルが続行されます。

```
W3026: illegal register number, aligned odd register(rXX) to be even register(rYY).
```

divh

符号付き除算（ハーフワード）を行います。（Divide Half-word）

［指定形式］

- divh reg1, reg2
- divh imm, reg2
- divh reg1, reg2, reg3 【V850E】
- divh imm, reg2, reg3 【V850E】

imm に指定可能なものを以下に示します。

- 16 ビット幅までの値を持つ絶対値式注
- 相対値式

注 as850 では，16 ビットを越えるかどうかのチェックは行われません。生成された機械語命令では，下位 16 ビットを用いて演算が行われます。

［機能］

- “divh reg1, reg2”の形式

  第 2 オペランドに指定したレジスタ値を，第 1 オペランドに指定したレジスタの下位ハーフワード・データの値で符号付きの値として除算し，商を第 2 オペランドに指定したレジスタに格納します。

- “divh imm, reg2”の形式

  第 2 オペランドに指定したレジスタ値を，第 1 オペランドに指定した絶対値式，または相対値式の下位ハーフワード・データの値で符号付きの値として除算し，商を第 2 オペランドに指定したレジスタに格納します。

- “divh reg1, reg2, reg3”の形式

  第 2 オペランドに指定したレジスタ値を，第 1 オペランドに指定したレジスタの下位ハーフワード・データの値で符号付きの値として除算し，商を第 2 オペランドに指定したレジスタに，剰余を第 3 オペランドに指定したレジスタに格納します。第 2 オペランドと第 3 オペランドが同じレジスタの場合，レジスタには剰余を格納します。

- “divh imm, reg2, reg3”の形式

  第 2 オペランドに指定したレジスタ値を，第 1 オペランドに指定した絶対値式，または相対値式の下位ハーフワード・データの値で符号付きの値として除算し，商を第 2 オペランドに指定したレジスタに，剰余を第 3 オペランドに指定したレジスタに格納します。第 2 オペランドと第 3 オペランドが同じレジスタの場合，レジスタには剰余を格納します。

［詳細説明］

- “divh reg1, reg2”，および“divh reg1, reg2, reg3”の形式の命令に対し，as850 では，機械語命令の divh 命令が 1 つ生成されます。
- “divh imm, reg2, reg3”の形式で imm に次のものを指定した場合，as850 では，命令展開が行われ，1 つ，または複数個の機械語命令が生成されます。**注**

（a）0

`divh  0, reg`	`divh  r0, reg`

（b）-16 ～ +15 の範囲の絶対値式（0 以外）

`divh  imm5, reg`	`mov  imm5, r1` `divh  r1, reg`

（c）-16 ～ +15 の範囲を越え，-32768 ～ +32767 の範囲の絶対値式

`divh  imm16, reg`	`movea  imm16, r0, r1` `divh  r1, reg`

（d）imm に -32768 ～ +32767 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

`divh  imm, reg`	`movhi  hi(imm), r0, r1` `divh  r1, reg`

上記以外の場合

`divh  imm, reg`	`movhi  hi1(imm), r0, r1`
	`movea  lo(imm), r1, r1`
	`divh  r1, reg`

（e）imm に -32768 ～ +32767 の範囲を越える絶対値式【V850E】

imm の値の下位 16 ビットがすべて 0 の場合

`divh  imm, reg`	`movhi  hi(imm), r0, r1` `divh  r1, reg`

上記以外の場合

`divh  imm, reg`	`mov  imm, r1` `divh  r1, reg`

（f）sdata ／ sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

divh $label, reg	movea $label, r0, r1 divh r1, reg

（g）#label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

divh #label, reg	movhi hi1(#label), r0, r1
	movea lo(#label), r1, r1
	divh r1, reg

divh label, reg	movhi hi1(label), r0, r1
	movea lo(label), r1, r1
	divh r1, reg

divh $label, reg	movhi hi1($label), r0, r1
	movea lo($label), r1, r1
	divh r1, reg

（h）#label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式【V850E】

divh #label, reg	mov #label, r1 divh r1, reg
divh label, reg	mov label, r1 divh r1, reg
divh $label, reg	mov $label, r1 divh r1, reg

注 機械語命令の div 命令は，オペランドにイミーディエトをとりません。

- “divh imm, reg2, reg3”の形式で imm に次のものを指定した場合，as850 では，命令展開が行われ，複数個の機械語命令が生成されます。【V850E】

（a）0

divh 0, reg2, reg3	divh r0, reg2, reg3

（b）0 以外の -16 ～ +15 の範囲の絶対値式

divh imm5, reg2, reg3	mov imm5, r1 divh r1, reg2, reg3

（c）-16 ～ +15 の範囲を越え，-32768 ～ +32767 の範囲の絶対値式

divh imm16, reg2, reg3	movea imm16, r0, r1 divh r1, reg2, reg3

（d）-32768 ～ +32767 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

divh imm, reg2, reg3	movhi hi(imm), r0, r1 divh r1, reg2, reg3

上記以外の場合

divh imm, reg2, reg3	mov imm, r1 divh r1, reg2, reg3

（e）sdata ／ sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

divh $label, reg2, reg3	movea $label, r0, r1 divh r1, reg2, reg3

（f）#label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義に持たないラベルの $label を持つ相対値式

divh #label, reg2, reg3	mov #label, r1 divh r1, reg2, reg3
divh label, reg2, reg3	mov label, r1 divh r1, reg2, reg3
divh $label, reg2, reg3	mov $label, r1 divh r1, reg2, reg3

［フラグ］

CY	－
OV	Integer-Overflow を生じた場合 1，そうでない場合 0
S	結果が負になった場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	－

［注意事項］

- ターゲット・デバイスが V850Ex の場合“divh reg1, reg2”の第 1 オペランド（reg1）に r0 を指定すると，次のメッセージが出力され，アセンブルが中止されます。

```
E3239: illegal operand (can not use r0 as source in V850E mode）
```

- ターゲット・デバイスが V850Ex 以外の場合“divh reg1, reg2”の第 1 オペランド（reg1）に r0 を指定すると，次のメッセージが出力され，アセンブルが続行されます。

```
W3013: register r0 used as source register
```

- ターゲット・デバイスが V850Ex の場合，“divh reg1, reg2”，および“divh imm, reg2, reg3”の第 2 オペランド（reg2）に r0 を指定すると，次のメッセージが出力され，アセンブルが中止されます。

```
E3240: illegal operand (can not use r0 as destination in V850E mode）
```

- ターゲット・デバイスが V850Ex 以外の場合，“divh reg1, reg2”，および“divh imm, reg2, reg3”の第 2 オペランド（reg2）に r0 を指定すると，次のメッセージが出力され，アセンブルが続行されます。

```
W3013: register r0 used as destination register
```

div

符号付き除算（ワード）を行います。（Divide Word）【V850E】

［指定形式］

- div reg1, reg2, reg3
- div imm, reg2, reg3

imm に指定可能なものを以下に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式

［機能］

- “div reg1, reg2, reg3”の形式

  第 2 オペランドに指定したレジスタ値を，第 1 オペランドに指定したレジスタ値で符号付きの値として除算し，商を第 2 オペランドに指定したレジスタに，剰余を第 3 オペランドに指定したレジスタに，それぞれ格納します。第 2 オペランドと第 3 オペランドが同じレジスタの場合，レジスタには剰余を格納します。

- “div imm, reg2, reg3”の形式

  第 2 オペランドに指定したレジスタ値を，第 1 オペランドに指定した絶対値式，または相対値式の値で符号付きの値として除算し，商を第 2 オペランドに指定したレジスタに，剰余を第 3 オペランドに指定したレジスタに格納します。第 2 オペランドと第 3 オペランドが同じレジスタの場合，レジスタには剰余を格納します。

［詳細説明］

- “div reg1, reg2, reg3”の形式の命令に対し，as850 では，機械語命令の div 命令が 1 つ生成されます。
- “div imm, reg2, reg3”の形式で imm に次のものを指定した場合，as850 では，命令展開が行われ，複数個の機械語命令が生成されます。**注**

（a）0

`div 0, reg2, reg3`	`div r0, reg2, reg3`

（b）0 以外の -16 ～ +15 の範囲の絶対値式

`div imm5, reg2, reg3`	`mov imm5, r1` `div r1, reg2, reg3`

（c）-16 ～ +15 の範囲を越え，-32768 ～ +32767 の範囲の絶対値式

`div imm16, reg2, reg3`	`movea imm16, r0, r1` `div r1, reg2, reg3`

（d）-32768 ～ +32767 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

`div imm, reg2, reg3`	`movhi hi(imm), r0, r1` `div r1, reg2, reg3`

上記以外の場合

`div imm, reg2, reg3`	`mov imm, r1` `div r1, reg2, reg3`

（e）sdata ／ sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

`div $label, reg2, reg3`	`movea $label, r0, r1` `div r1, reg2, reg3`

（f）#label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義に持たないラベルの $label を持つ相対値式

`div #label, reg2, reg3`	`mov #label, r1` `div r1, reg2, reg3`
`div label, reg2, reg3`	`mov label, r1` `div r1, reg2, reg3`
`div $label, reg2, reg3`	`mov $label, r1` `div r1, reg2, reg3`

注 機械語命令の div 命令は，オペランドにイミーディエトをとりません。

［フラグ］

CY	—
OV	Integer-Overflow を生じた場合 1，そうでない場合 0
S	結果が負になった場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	—

divhu

符号なし除算（ハーフワード）を行います。（Divide Half-word Unsigned）【V850E】

［指定形式］

- divhu reg1, reg2, reg3
- divhu imm, reg2, reg3

imm に指定可能なものを以下に示します。

- 16 ビット幅までの値を持つ絶対値式注
- 相対値式

注 as850 では，16 ビットを越えるかどうかのチェックは行われません。生成された機械語命令では，下位 16 ビットを用いて演算が行われます。

［機能］

- “divhu reg1, reg2, reg3”の形式

  第 2 オペランドに指定したレジスタ値を，第 1 オペランドに指定したレジスタの下位ハーフワード・データの値で符号なしの値として除算し，商を第 2 オペランドに指定したレジスタに，剰余を第 3 オペランドに指定したレジスタに格納します。第 2 オペランドと第 3 オペランドが同じレジスタの場合，レジスタには剰余を格納します。

- “divhu imm, reg2, reg3”の形式

  第 2 オペランドに指定したレジスタ値を，第 1 オペランドに指定した絶対値式，または相対値式の下位ハーフワード・データの値で符号なしの値として除算し，商を第 2 オペランドに指定したレジスタに，剰余を第 3 オペランドに指定したレジスタに格納します。第 2 オペランドと第 3 オペランドが同じレジスタの場合，レジスタには剰余を格納します。

［詳細説明］

- “divhu reg1, reg2, reg3”の形式の命令に対し，as850 では，機械語命令の divhu 命令が 1 つ生成されます。
- “divhu imm, reg2, reg3”の形式で imm に次のものを指定した場合，as850 では，命令展開が行われ，複数個の機械語命令が生成されます。注

（a）0

`divhu 0, reg2, reg3`	`divhu r0, reg2, reg3`

（b）0 以外の 0 ～ 31 の範囲の絶対値式

`divhu imm5, reg2, reg3`	`mov imm5, r1` `divhu r1, reg2, reg3`

（c）0 ～ 31 の範囲を越え，0 ～ 65535 の範囲の絶対値式

`divhu imm16, reg2, reg3`	`movea imm16, r0, r1` `divhu r1, reg2, reg3`

（d）0 ～ 65535 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

`divhu imm, reg2, reg3`	`movhi hi(imm), r0, r1` `divhu r1, reg2, reg3`

上記以外の場合

`divhu imm, reg2, reg3`	`mov imm, r1` `divhu r1, reg2, reg3`

（e）sdata ／ sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

`divhu $label, reg2, reg3`	`movea $label, r0, r1` `divhu r1, reg2, reg3`

（f）#label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義に持たないラベルの $label を持つ相対値式

`divhu #label, reg2, reg3`	`mov #label, r1` `divhu r1, reg2, reg3`
`divhu label, reg2, reg3`	`mov label, r1` `divhu r1, reg2, reg3`
`divhu $label, reg2, reg3`	`mov $label, r1` `divhu r1, reg2, reg3`

注 機械語命令の divhu 命令は，オペランドにイミーディエトをとりません。

［フラグ］

CY	–
OV	Integer-Overflow を生じた場合 1，そうでない場合 0
S	演算結果のワード・データの MSB が 1 の場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	–

divu

符号なし除算（ワード）を行います。（Divide Word Unsigned）【V850E】

［指定形式］

- divu reg1, reg2, reg3
- divu imm, reg2, reg3

imm に指定可能なものを以下に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式

［機能］

- “divu reg1, reg2, reg3”の形式

  第 2 オペランドに指定したレジスタ値を，第 1 オペランドに指定したレジスタ値で符号なしの値として除算し，商を第 2 オペランドに指定したレジスタに，剰余を第 3 オペランドに指定したレジスタに格納します。第 2 オペランドと第 3 オペランドが同じレジスタの場合，レジスタには剰余を格納します。

- “divu imm, reg2, reg3”の形式

  第 2 オペランドに指定したレジスタ値を，第 1 オペランドに指定した絶対値式，または相対値式の値で符号なしの値として除算し，商を第 2 オペランドに指定したレジスタに，剰余を第 3 オペランドに指定したレジスタに格納します。第 2 オペランドと第 3 オペランドが同じレジスタの場合，レジスタには剰余を格納します。

［詳細説明］

- “divu reg1, reg2, reg3”の形式の命令に対し，as850 では，機械語命令の divu 命令が 1 つ生成されます。
- “divu imm, reg2, reg3”の形式で imm に次のものを指定した場合，as850 では，命令展開が行われ，複数個の機械語命令が生成されます。**注**

（a）0

`divu 0, reg2, reg3`	`divu r0, reg2, reg3`

（b）0 以外の -16 ～＋ 15 の範囲の絶対値式

`divu imm5, reg2, reg3`	`mov imm5, r1` `divu r1, reg2, reg3`

（c）0 ～ 31 の範囲を越え，-32,768 ～ +32,767 の範囲の絶対値式

`divu imm16, reg2, reg3`	`movea imm16, r0, r1` `divu r1, reg2, reg3`

（d）0 ～ 65535 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

`divu imm, reg2, reg3`	`movhi hi(imm), r0, r1` `divu r1, reg2, reg3`

上記以外の場合

`divu imm, reg2, reg3`	`mov imm, r1` `divu r1, reg2, reg3`

（e）sdata ／ sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

`divu $label, reg2, reg3`	`movea $label, r0, r1` `divu r1, reg2, reg3`

（f）#label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義に持たないラベルの $label を持つ相対値式

`divu #label, reg2, reg3`	`mov #label, r1` `divu r1, reg2, reg3`
`divu label, reg2, reg3`	`mov label, r1` `divu r1, reg2, reg3`
`divu $label, reg2, reg3`	`mov $label, r1` `divu r1, reg2, reg3`

注 機械語命令の divu 命令は，オペランドにイミーディエトをとりません。

［フラグ］

CY	—
OV	Integer-Overflow を生じた場合 1，そうでない場合 0
S	演算結果のワード・データの MSB が 1 の場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	—

cmp

比較を行います。（Compare）

［指定形式］

- cmp reg1, reg2
- cmp imm, reg2

imm に指定可能なものを以下に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式

［機能］

- “cmp reg1, reg2”の形式

  第 1 オペランドに指定したレジスタ値と第 2 オペランドに指定したレジスタ値を比較し，結果をフラグに示します。なお，比較は，第 2 オペランドに指定したレジスタ値から第 1 オペランドに指定したレジスタ値を減算することにより行われます。

- “cmp imm, reg2”の形式

  第 1 オペランドに指定した絶対値式，または相対値式の値と第 2 オペランドに指定したレジスタ値を比較し，結果をフラグに示します。なお，比較は，第 2 オペランドに指定したレジスタ値から第 1 オペランドに指定した値を減算することにより行われます。

［詳細説明］

- “cmp reg1, reg2”の形式の命令に対し，as850 では，機械語命令の cmp 命令が 1 つ生成されます。
- “cmp imm, reg2”の形式の形式で imm に次のものを指定した場合，as850 では，機械語命令の cmp 命令[注]が 1 つ生成されます。

（a）-16 ～ +15 の範囲の絶対値式

`cmp imm5, reg`	`cmp imm5, reg`

注 機械語命令の cmp 命令は，第 1 オペランドにレジスタ，または -16 ～ +15（0xfffffff0 ～ 0xf）の範囲のイミーディエトをとります。

- “cmp imm, reg2”の形式の形式で imm に次のものを指定した場合，as850 では，機械語命令の cmp 命令が 1 つ生成されます。

（a）-16 ～ +15 の範囲を越え，-32768 ～ +32767 の範囲の絶対値式

cmp imm16, reg	movea imm16, r0, r1 cmp r1, reg

（b）-32768 ～ +32767 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

cmp imm, reg	movhi hi(imm), r0, r1 cmp r1, reg

上記以外の場合

cmp imm, reg	movhi hi1(imm), r0, r1
	movea lo(imm), r1, r1
	cmp r1, reg

（c）-32768 ～ +32767 の範囲を越える絶対値式【V850E】

imm の値の下位 16 ビットがすべて 0 の場合

cmp imm, reg	movhi hi(imm), r0, r1 cmp r1, reg

上記以外の場合

cmp imm, reg	mov imm, r1 cmp r1, reg

（d）sdata ／ sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

cmp $label, reg	movea $label, r0, r1 cmp r1, reg

（e）#label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

`cmp #label, reg`	`movhi hi1(#label), r0, r1`
	`movea lo(#label), r1, r1`
	`cmp r1, reg`

`cmp label, reg`	`movhi hi1(label), r0, r1`
	`movea lo(label), r1, r1`
	`cmp r1, reg`

`cmp $label, reg`	`movhi hi1($label), r0, r1`
	`movea lo($label), r1, r1`
	`cmp r1, reg`

（f）#label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式【V850E】

`cmp #label, reg`	`mov #label, r1` `cmp r1, reg`
`cmp label, reg`	`mov label, r1` `cmp r1, reg`
`cmp $label, reg`	`mov $label, r1` `cmp r1, reg`

［フラグ］

CY	MSB（Most Significant Bit）へのボローが生じた場合 1，そうでない場合 0
OV	Integer-Overflow を生じた場合 1，そうでない場合 0
S	結果が負になった場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	—

mov

データの転送を行います。（Move）

［指定形式］

- mov reg1, reg2
- mov imm, reg2

imm に指定可能なものを以下に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式

［機能］

- “mov reg1, reg2”の形式

  第 1 オペランドに指定したレジスタ値を，第 2 オペランドに指定したレジスタに格納します。

- “mov imm, reg2”の形式

  第 1 オペランドに指定した絶対値式，または相対値式の値を，第 2 オペランドに指定したレジスタに格納します。

［詳細説明］

- “mov reg1, reg2”の形式の命令に対し，as850 では，機械語命令の mov 命令が 1 つ生成されます。
- “mov imm, reg2”の形式で imm に次のものを指定した場合，as850 では，機械語命令の mov 命令[注]が 1 つ生成されます。

（a）-16 ～ +15 の範囲の絶対値式

`mov imm5, reg`	`mov imm5, reg`

注 機械語命令の mov 命令は，V850 の場合 16 ビット長形式のみであり，V850Ex の場合 48 ビット長形式がサポートされています。したがって，機械語命令の mov 命令は，V850 の場合，第 1 オペランドにレジスタ，または -16 ～ +15（0xfffffff0 ～ 0xf）の範囲のイミーディエトをとります。V850Ex の場合，これらに加えて，-2147483648 ～ -2147483647（0x80000000 ～ 0x7fffffff）の範囲のイミーディエトをとります。

- “mov imm, reg2”の形式で imm に次のものを指定した場合，as850 では，命令展開が行われ，1 つ，または複数個の機械語命令が生成されます。

（a）-16 ～ +15 の範囲を越え -32768 ～ +32767 の範囲の絶対値式

mov imm16, reg	movea imm16, r0, reg

（b）-32768 ～ +32767 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

mov imm, reg	movhi hi(imm), r0, reg

上記以外の場合

mov imm, reg	movhi hi1(imm), r0, r1 movea lo(imm), r1, reg

（c）-32768 ～ +32767 の範囲を越える絶対値式【V850E】

imm の値の下位 16 ビットがすべて 0 の場合

mov imm, reg	movhi hi(imm), r0, reg

上記以外の場合[注]

mov imm, reg	mov imm, reg

注 16 ビット長の mov 命令を，48 ビット長の mov 命令に置き換えます。

（d）!label，または %label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

mov !label, reg	movea !label, r0, reg
mov %label, reg	movea %label, r0, reg
mov $label, reg	movea $label, r0, reg

（e）#label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

mov #label, reg	movhi hi1(#label), r0, r1 movea lo(#label), r1, reg
mov label, reg	movhi hi1(label), r0, r1 movea lo(label), r1, reg
mov $label, reg	movhi hi1($label), r0, r1 movea lo($label), r1, reg

（f）#label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式[注]【V850E】

mov #label, reg	mov #label, reg
mov label, reg	mov label, reg
mov $label, reg	mov $label, reg

注 16 ビット長の mov 命令を，48 ビット長の mov 命令に置き換えます。

［フラグ］

CY	—
OV	—
S	—
Z	—
SAT	—

［注意事項］

- “mov reg1, reg2”の第 1 オペランドに r0 を指定し，かつ，第 2 オペランドに r0 を指定すると，アセンブルした結果は nop の命令コードになります。
- ターゲット・デバイスが V850Ex の場合，“mov imm, reg2”の命令に対し，第 1 オペランドに -16 ～ +15 の範囲の絶対値式を指定し，かつ第 2 オペランドに r0 を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

E3240: illegal operand (can not use r0 as destination in V850E mode)

- ターゲット・デバイスが V850Ex の場合，“mov imm, reg2”の命令に対し，第 1 オペランドに -32768 ～ +32767 の範囲を越える絶対値式，#label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式を指定し，疑似命令 .option nomacro の指定により命令展開を抑止している場合，次のメッセージが出力され，アセンブルが中止されます。
  このような場合は，mov32 命令を使用してください。

E3249: illegal syntax

movea

実効アドレスの転送を行います。（Move Effective Address）

［指定形式］

- movea imm, reg1, reg2

imm に指定可能なものを以下に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式
- 上記のものに hi ( )，lo ( )，または hi1 ( ) を適用したもの

［機能］

第 1 オペランドに指定した絶対値式，相対値式，あるいは，hi ( )，lo ( )，または hi1 ( ) を適用した値を，第 2 オペランドに指定したレジスタ値と加算し，結果を第 3 オペランドに指定したレジスタに格納します。

［詳細説明］

- imm に次のものを指定した場合，as850 では，機械語命令の movea 命令[注]が 1 つ生成されます。
- reg1 に r0 を指定した場合，as850 では，mov imm, reg2 を指定したものとして扱います。

（a）-32768 ～ +32767 の範囲の絶対値式

movea imm16, reg1, reg2	movea imm16, reg1, reg2

（b）sdata ／ sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

movea $label, reg1, reg2	movea $label, reg1, reg2

（c）!label，または %label を持つ相対値式

movea !label, reg1, reg2	movea !label, reg1, reg2
movea %label, reg1, reg2	movea %label, reg1, reg2

（d）hi ( )，lo ( )，または hi1 ( ) を適用したもの

movea imm16, reg1, reg2	movea imm16, reg1, reg2

注 機械語命令の movea 命令は，第 1 オペランドに -32768 ～ +32767（0xffff8000 ～ 0x7fff）の範囲のイミーディエトをとります。

- imm に次のものを指定した場合，as850 では，命令展開が行われ，1 つ，または複数個の機械語命令が生成されます。

（a）-32768 ～ +32767 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

movea imm, reg1, reg2	movhi hi(imm), reg1, reg2

上記以外の場合

movea imm, reg1, reg2	movhi hi1(imm), reg1, r1 movea lo(imm), r1, reg2

（b）#label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

movea #label, reg1, reg2	movhi hi1(#label), reg1, r1 movea lo(#label), r1, reg2
movea label, reg1, reg2	movhi hi1(label), reg1, r1 movea lo(label), r1, reg2
movea $label, reg1, reg2	movhi hi1($label), reg1, r1 movea lo($label), r1, reg2

［フラグ］

CY	—
OV	—
S	—
Z	—
SAT	—

［注意事項］

- ターゲット・デバイスが V850Ex の場合，第 3 オペランドに r0 を指定すると，次のメッセージが出力され，アセンブルが中止されます。

```
E3240: illegal operand (can not use r0 as destination in V850E mode）
```

- ターゲット・デバイスが V850Ex 以外の場合，第 3 オペランドに r0 を指定すると，次のメッセージが出力され，アセンブルが続行されます。

```
W3013: register r0 used as destination register
```

movhi

上位ハーフワードの転送を行います。（Move High half-word）

［指定形式］

- movhi imm16, reg1, reg2

imm16 に指定可能なものを以下に示します。

- 16 ビット幅までの値を持つ絶対値式
- 相対値式
- 上記のものに hi ( )，lo ( )，または hi1 ( ) を適用したもの

［機能］

第 1 オペランドに指定した値を上位 16 ビットの値，0 を下位 16 ビットの値とするワード・データと，第 2 オペランドに指定したレジスタ値を加算し，結果を第 3 オペランドに指定したレジスタに格納します。

［詳細説明］

as850 では，機械語命令の movhi 命令が 1 つ生成されます。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

［注意事項］

- imm16 に 0 ～ 65535 の範囲を越える絶対値式を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E3231: illegal operand (range error in immediate）
```

- ターゲット・デバイスが V850Ex の場合，第 3 オペランドに r0 を指定すると，次のメッセージが出力され，アセンブルが中止されます。

```
E3240: illegal operand (can not use r0 as destination in V850E mode）
```

- ターゲット・デバイスが V850Ex 以外の場合，第 3 オペランドに r0 を指定すると，次のメッセージが出力され，アセンブルが続行されます。

```
W3013: register r0 used as destination register
```

mov32

32 ビット・データの転送を行います。（32 bit Move）【V850E】

［指定形式］

- mov32　imm, reg2

imm に指定可能なものを以下に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式

［機能］

第 1 オペランドに指定した絶対値式，または相対値式の値を，第 2 オペランドに指定したレジスタに格納します。

［詳細説明］

as850 では，機械語命令の 48 ビット長 mov 命令が 1 つ生成されます。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

cmov

フラグ条件付きデータの転送を行います。（Conditional Move）【V850E】

［指定形式］

- cmov imm4, reg1, reg2, reg3
- cmov imm4, imm, reg2, reg3
- cmov*cnd* reg1, ret2, reg3
- cmov*cnd* imm, reg2, reg3

imm4 に指定可能なものを以下に示します。

- 4 ビット幅までの値を持つ絶対値式**注**

注 機械語命令の cmov 命令は，第 1 オペランドに 0 ～ 15（0x0 ～ 0xf）の範囲のイミーディエトをとります。

imm に指定可能なものを以下に示します。

- 32 ビット幅までの値を持つ絶対値式

［機能］

- “cmov imm4, reg1, reg2, reg3”の形式

  第 1 オペランドに指定した定数式の値の下位 4 ビットの値で示されるフラグ状態と，現在のフラグ状態を比較し，値が一致した場合は，第 2 オペランドに指定したレジスタ値を，一致しなかった場合は第 3 オペランドに指定したレジスタ値を，第 4 オペランドに指定したレジスタに格納します。
- “cmov imm4, imm, reg2, reg3”の形式

  第 1 オペランドに指定した定数式の値の下位 4 ビットの値で示されるフラグ状態と，現在のフラグ状態を比較し，値が一致した場合は，第 2 オペランドに指定した絶対値式の値を，一致しなかった場合は第 3 オペランドに指定したレジスタ値を，第 4 オペランドに指定したレジスタに格納します。
- “cmov*cnd* reg1, ret2, reg3”の形式

  cnd 部分の文字列で示されるフラグ状態と現在のフラグの状態を比較し，値が一致した場合は，第 1 オペランドに指定したレジスタ値を，一致しなかった場合は第 2 オペランドに指定したレジスタ値を，第 3 オペランドに指定したレジスタに格納します。
- “cmov*cnd* imm, reg2, reg3”の形式

  cnd 部分の文字列で示されるフラグ状態と現在のフラグの状態を比較し，値が一致した場合は，第 1 オペランドに指定した絶対値式の値を，一致しなかった場合は第 2 オペランドに指定したレジスタ値を，第 3 オペランドに指定したレジスタに格納します。

表 4―49　cmov*cnd* 命令

命令	フラグ状態	フラグ状態の意味	命令展開
cmovgt	((S xor OV) or Z) = 0	Greater than（signed）	cmov 0xf
cmovge	(S xor OV) = 0	Greater than or equal（signed）	cmov 0xe
cmovlt	(S xor OV) = 1	Less than（signed）	cmov 0x6
cmovle	((S xor OV) or Z) = 1	Less than or equal（signed）	cmov 0x7
cmovh	(CY or Z) = 0	Higher（Greater than）	cmov 0xb
cmovnl	CY = 0	Not lower（Greater than or equal）	cmov 0x9
cmovl	CY = 1	Lower（Less than）	cmov 0x1
cmovnh	(CY or Z) = 1	Not higher（Less than or equal）	cmov 0x3
cmove	Z = 1	Equal	cmov 0x2
cmovne	Z = 0	Not equal	cmov 0xa
cmovv	OV = 1	Overflow	cmov 0x0
cmovnv	OV = 0	No overflow	cmov 0x8
cmovn	S = 1	Negative	cmov 0x4
cmovp	S = 0	Positive	cmov 0xc
cmovc	CY = 1	Carry	cmov 0x1
cmovnc	CY = 0	No carry	cmov 0x9
cmovz	Z = 1	Zero	cmov 0x2
cmovnz	Z = 0	Not zero	cmov 0xa
cmovt	always 1	Always 1	cmov 0x5
cmovsa	SAT = 1	Saturated	cmov 0xd

［詳細説明］

- “cmov imm4, reg1, reg2, reg3”の形式の命令に対し，as850 では，機械語命令の cmov 命令[注]が 1 つ生成されます。

注　機械語命令の cmov 命令は，第 2 オペランドにレジスタ，または -16 ～ +15（0xfffffff0 ～ 0xf）の範囲のイミーディエトをとります。

- “cmov imm4, imm, reg2, reg3”の形式で imm に次のものを指定した場合，as850 では，機械語命令の cmov 命令が 1 つ生成されます。

（a）-16 ～ +15 の範囲の絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

cmov　imm4, imm5, reg2, reg3	cmov　imm4, imm5, reg2, reg3

- “cmov imm4, imm, reg2, reg3”の形式で imm に次のものを指定した場合，as850 では，命令展開が行われ，複数個の機械語命令が生成されます。

（a）-16 ～ +15 の範囲を越え，-32768 ～ +32767 の範囲の絶対値式

`cmov imm4, imm16, reg2, reg3`	`movea imm16, r0, r1` `cmov imm4, r1, reg2, reg3`

（b）-32768 ～ +32767 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

`cmov imm4, imm, reg2, reg3`	`movhi hi(imm), r0, r1` `cmov imm4, r1, reg2, reg3`

上記以外の場合

`cmov imm4, imm, reg2, reg3`	`mov imm, r1` `cmov imm4, r1, reg2, reg3`

（c）#label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

`cmov imm4, #label, reg2, reg3`	`mov #label, r1` `cmov imm4, r1, reg2, reg3`
`cmov imm4, label, reg2, reg3`	`mov label, r1` `cmov imm4, r1, reg2, reg3`
`cmov imm4, $label, reg2, reg3`	`mov $label, r1` `cmov imm4, r1, reg2, reg3`

（d）!label，または %label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

`cmov imm4, !label, reg2, reg3`	`movea !label, r0, r1` `cmov imm4, r1, reg2, reg3`
`cmov imm4, %label, reg2, reg3`	`movea %label, r0, r1` `cmov imm4, r1, reg2, reg3`
`cmov imm4, $label, reg2, reg3`	`movea $label, r0, r1` `cmov imm4, r1, reg2, reg3`

- “cmov*cnd* reg1, ret2, reg3”の形式の命令に対し，as850 では，対応する cmov 命令が生成され（「表 4―49 cmovcnd 命令」を参照），“cmov imm4, reg1, reg2, reg3”の形式に展開されます。

- “cmov*cnd* imm, reg2, reg3”の形式で imm に次のものを指定した場合，as850 では，対応する cmov 命令が生成され（「表 4―49 cmovcnd 命令」を参照），“cmov imm4, imm, reg2, reg3”の形式に展開されます。

（a）-16 ～ +15 の範囲の絶対値式

- “cmov*cnd* imm, reg2, reg3”の形式で imm に次のものを指定した場合，as850 では，命令展開が行われ，複数個の機械語命令が生成されます。

（a）-16 ～ +15 の範囲を越え，-32768 ～ +32767 の範囲の絶対値式

cmov*cnd* imm16, reg2, reg3	movea imm16, r0, r1 cmov*cnd* r1, reg2, reg3

（b）-32768 ～ +32767 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

cmov*cnd* imm, reg2, reg3	movhi hi(imm), r0, r1 cmov*cnd* r1, reg2, reg3

上記以外の場合

cmov*cnd* imm, reg2, reg3	mov imm, r1 cmov*cnd* r1, reg2, reg3

（c）#label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

cmov*cnd* #label, reg2, reg3	mov #label, r1 cmov*cnd* r1, reg2, reg3
cmov*cnd* label, reg2, reg3	mov label, r1 cmov*cnd* r1, reg2, reg3
cmov*cnd* $label, reg2, reg3	mov $label, r1 cmov*cnd* r1, reg2, reg3

（d）!label，または %label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

cmov*cnd* !label, reg2, reg3	movea !label, r0, r1 cmov*cnd* r1, reg2, reg3
cmov*cnd* imm4, %label, reg2, reg3	movea %label, r0, r1 cmov*cnd* r1, reg2, reg3
cmov*cnd* imm4, $label, reg2, reg3	movea $label, r0, r1 cmov*cnd* r1, reg2, reg3

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

［注意事項］

- cmov 命令の imm4 に 4 ビットの範囲を越える定数式を指定した場合，次のメッセージが出力されます。
  なお，4 ビット幅を越える場合は，0xf でマスクした値を用いてアセンブルが続行されます。

```
W3011: illegal operand (range error in immediate）
```

- cmov 命令の imm4 に定数式以外（未定義シンボルやラベルの参照）を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E3249: illegal syntax
```

setf

フラグ条件の設定を行います。（Set Flag Condition）

［指定形式］

- setf　imm4, reg
- setf*cnd* reg

imm4 に指定可能なものを以下に示します。

- 4 ビット幅までの値を持つ絶対値式

［機能］

-“setf imm4, reg”の形式

　第 1 オペランドに指定した絶対値式の値の下位 4 ビットの値で示されるフラグ状態と，現在のフラグ状態を比較し，値が一致した場合は 1 を，一致しなかった場合は 0 を第 2 オペランドに指定したレジスタに格納します。

-“setf*cnd* reg”の形式

　cnd 部分の文字列で示されるフラグ状態と現在のフラグの状態を比較し，一致する場合は 1，一致しなかった場合は 0 を第 2 オペランドに指定したレジスタに格納します。

［詳細説明］

- setf 命令に対し，as850 では，機械語命令の setf 命令が 1 つ生成されます。
- setf*cnd* 命令に対し，as850 では，対応する sesf 命令が生成され（「表 4―50　setfcnd 命令」を参照），“setf imm4, reg”の形式に展開されます。

表 4―50　setf*cnd* 命令

命令	フラグ状態	フラグ状態の意味	命令展開
setfgt	((S xor OV) or Z) = 0	Greater than（signed）	setf 0xf
setfge	(S xor OV) = 0	Greater than or equal（signed）	setf 0xe
setflt	(S xor OV) = 1	Less than（signed）	setf 0x6
setfle	((S xor OV) or Z) = 1	Less than or equal（signed）	setf 0x7
setfh	(CY or Z) = 0	Higher（Greater than）	setf 0xb
setfnl	CY = 0	Not lower（Greater than or equal）	setf 0x9
setfl	CY = 1	Lower（Less than）	setf 0x1
setfnh	(CY or Z) = 1	Not higher（Less than or equal）	setf 0x3
setfe	Z = 1	Equal	setf 0x2
setfne	Z = 0	Not equal	setf 0xa

命令	フラグ状態	フラグ状態の意味	命令展開
setfv	OV = 1	Overflow	setf 0x0
setfnv	OV = 0	No overflow	setf 0x8
setfn	S = 1	Negative	setf 0x4
setfp	S = 0	Positive	setf 0xc
setfc	CY = 1	Carry	setf 0x1
setfnc	CY = 0	No carry	setf 0x9
setfz	Z = 1	Zero	setf 0x2
setfnz	Z = 0	Not zero	setf 0xa
setft	always 1	Always 1	setf 0x5
setfsa	SAT = 1	Saturated	setf 0xd

[フラグ]

CY	–
OV	–
S	–
Z	–
SAT	–

[注意事項]

- setf 命令の imm4 に 4 ビットの範囲を越える絶対値式を指定した場合，次のメッセージが出力され，指定された値の下位 4 ビットが用いられてアセンブルが続行されます。

W3011: illegal operand (range error in immediate).

sasf

論理左シフト付きフラグ条件の設定を行います。（Shift And Set Flag Condition）【V850E】

［指定形式］

- sasf　imm4, reg
- sasf*cnd* reg

imm4 に指定可能なものを以下に示します。

- 4 ビット幅までの値を持つ絶対値式

［機能］

- “sasf imm4, reg”の形式

　第 1 オペランドに指定した絶対値式の値の下位 4 ビットの値で示されるフラグ状態（「表 4―51　sasfcnd 命令」を参照）と，現在のフラグ状態を比較します。値が一致した場合は，第 2 オペランドで指定したレジスタの内容を左 1 ビット論理シフトした値と 1 とを論理和して，第 2 オペランドに指定したレジスタに格納し，一致しなかった場合は，第 2 オペランドで指定したレジスタの内容を左 1 ビット論理シフトして第 2 オペランドで指定したレジスタに格納します。

- “sasf*cnd* reg”の形式

　cnd 部分の文字列で示されるフラグ状態と現在のフラグの状態を比較します。値が一致した場合は，第 2 オペランドで指定したレジスタの内容を左 1 ビット論理シフトした値と 1 とを論理和して第 2 オペランドで指定したレジスタに格納し，一致しなかった場合は，第 2 オペランドで指定したレジスタの内容を左 1 ビット論理シフトして第 2 オペランドで指定したレジスタに格納します。

［詳細説明］

- sasf 命令に対し，as850 では，機械語命令の sasf 命令が 1 つ生成されます。
- sasf*cnd* 命令に対し，as850 では，対応する sasf 命令が生成され（「表 4―51　sasfcnd 命令」を参照），“sasf imm4, reg”の形式に展開されます。

表 4―51　sasf*cnd* 命令

命令	フラグ状態	フラグ状態の意味	命令展開
sasfgt	((S xor OV) or Z) = 0	Greater than（signed）	sasf 0xf
sasfge	(S xor OV) = 0	Greater than or equal（signed）	sasf 0xe
sasflt	(S xor OV) = 1	Less than（signed）	sasf 0x6
sasfle	((S xor OV) or Z) = 1	Less than or equal（signed）	sasf 0x7
sasfh	(CY or Z) = 0	Higher（Greater than）	sasf 0xb
sasfnl	CY = 0	Not lower（Greater than or equal）	sasf 0x9
sasfl	CY = 1	Lower（Less than）	sasf 0x1

命令	フラグ状態	フラグ状態の意味	命令展開
sasfnh	（CY or Z）= 1	Not higher（Less than or equal）	sasf 0x3
sasfe	Z = 1	Equal	sasf 0x2
sasfne	Z = 0	Not equal	sasf 0xa
sasfv	OV = 1	Overflow	sasf 0x0
sasfnv	OV = 0	No overflow	sasf 0x8
sasfn	S = 1	Negative	sasf 0x4
sasfp	S = 0	Positive	sasf 0xc
sasfc	CY = 1	Carry	sasf 0x1
sasfnc	CY = 0	No carry	sasf 0x9
sasfz	Z = 1	Zero	sasf 0x2
sasfnz	Z = 0	Not zero	sasf 0xa
sasft	always 1	Always 1	sasf 0x5
sasfsa	SAT = 1	Saturated	sasf 0xd

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

［注意事項］

- sasf 命令の imm4 に 4 ビットの範囲を越える絶対値式を指定した場合，次のメッセージが出力され，指定された値の下位 4 ビットが用いられてアセンブルが続行されます。

W3011: illegal operand (range error in immediate).

4. 5. 8　飽和演算命令

この項では，飽和演算命令について説明します。次に，この項において説明する命令を示します。

表 4―52　飽和演算命令

命令	意味
satadd	飽和加算
satsub	飽和減算
satsubi	飽和減算（イミーディエト）
satsubr	飽和逆減算

satadd

飽和加算を行います。（Saturated Add）

［指定形式］

- satadd reg1, reg2
- satadd imm, reg2
- satadd reg1, reg2, reg3 【V850E2】

imm に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式

［機能］

- “satadd reg1, reg2”の形式

  第 1 オペランドに指定したレジスタ値と，第 2 オペランドに指定したレジスタ値を加算し，結果を第 2 オペランドに指定したレジスタに格納します。ただし，結果が正の最大値 0x7fffffff を越えた場合は 0x7fffffff を，負の最大値 0x80000000 を越えた場合は 0x80000000 を第 2 オペランドに指定したレジスタに格納し，SAT フラグに 1 を設定します。

- “satadd imm, reg2”の形式

  第 1 オペランドに指定した絶対値式，または相対値式の値と，第 2 オペランドに指定したレジスタの値を加算し，結果を第 2 オペランドに指定したレジスタに格納します。ただし，結果が正の最大値 0x7fffffff を越えた場合は 0x7fffffff を，負の最大値 0x80000000 を越えた場合は 0x80000000 を第 2 オペランドに指定したレジスタに格納し，SAT フラグに 1 を設定します。

- “satadd reg1, reg2, reg3”の形式

  第 1 オペランドに指定したレジスタ値と，第 2 オペランドに指定したレジスタ値を加算し，結果を第 3 オペランドに指定したレジスタに格納します。ただし，結果が正の最大値 0x7fffffff を越えた場合は 0x7fffffff を，負の最大値 0x80000000 を越えた場合は 0x80000000 を第 3 オペランドに指定したレジスタに格納し，SAT フラグに 1 を設定します。

［詳細説明］

- “satadd reg1, reg2”，および“satadd reg1, reg2, reg3 ”の形式の命令に対し，as850 では，機械語命令の satadd 命令が 1 つ生成されます。

- “satadd imm, reg2”の形式で imm に次のものを指定した場合，as850 では，機械語命令の satadd 命令[注]が 1 つ生成されます。

（a）-16 ～ +15 の範囲の絶対値式

satadd imm5, reg	satadd imm5, reg

注 機械語命令の satadd 命令は，第 1 オペランドにレジスタ，または -16 ～ +15（0xfffffff0 ～ 0xf）の範囲のイミーディエトをとります。

- “satadd imm, reg2”の形式で imm に次のものを指定した場合，as850 では，命令展開が行われ，複数個の機械語命令が生成されます。

（a）-16 ～ +15 の範囲を越え，-32768 ～ +32767 の範囲の絶対値式

satadd imm16, reg	movea imm16, r0, r1 satadd r1, reg

（b）-32768 ～ +32767 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

satadd imm, reg	movhi hi(imm), r0, r1 satadd r1, reg

上記以外の場合

satadd imm, reg	movhi hi1(imm), r0, r1
	movea lo(imm), r1, r1
	satadd r1, reg

（c）-32768 ～ +32767 の範囲を越える絶対値式【V850E】

imm の値の下位 16 ビットがすべて 0 の場合

satadd imm, reg	movhi hi(imm), r0, r1 satadd r1, reg

上記以外の場合

satadd imm, reg	mov imm, r1 satadd r1, reg

（d）sdata ／ sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

satadd $label, reg	movea $label, r0, r1 satadd r1, reg

（e）#label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

satadd #label, reg	movhi hi1(#label), r0, r1
	movea lo(#label), r1, r1
	satadd r1, reg

satadd label, reg	movhi hi1(label), r0, r1
	movea lo(label), r1, r1
	satadd r1, reg

satadd $label, reg	movhi hi1($label), r0, r1
	movea lo($label), r1, r1
	satadd r1, reg

（f）#label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式【V850E】

satadd #label, reg	mov #label, r1 satadd r1, reg
satadd label, reg	mov label, r1 satadd r1, reg
satadd $label, reg	mov $label, r1 satadd r1, reg

［フラグ］

CY	MSB（Most Significant Bit）からのキャリーを生じた場合 1，そうでない場合 0
OV	Integer-Overflow を生じた場合 1，そうでない場合 0
S	結果が負になった場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	OV = 1 になった場合 1，そうでない場合 －

［注意事項］

- “satadd reg1, reg2”，および“satadd imm, reg2”の形式の命令に対し，ターゲット・デバイスが V850Ex の場合，第 2 オペランドに r0 を指定すると，次のメッセージが出力され，アセンブルが中止されます。

```
E3240: illegal operand (can not use r0 as destination in V850E mode）
```

- “satadd reg1, reg2”，および“satadd imm, reg2”の形式の命令に対し，ターゲット・デバイスが V850Ex 以外の場合，第 2 オペランドに r0 を指定すると，次のメッセージが出力され，アセンブルが続行されます。

```
W3013: register r0 used as destination register
```

satsub

飽和減算を行います。（Saturated Subtract）

［指定形式］

- satsub reg1, reg2
- satsub imm, reg2
- satsub reg1, reg2, reg3 【V850E2】

imm に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式

［機能］

- “satsub reg1, reg2”の形式

  第 2 オペランドに指定したレジスタ値から，第 1 オペランドに指定したレジスタ値を減算し，結果を第 2 オペランドに指定したレジスタに格納します。ただし，結果が正の最大値 0x7fffffff を越えた場合は 0x7fffffff を，負の最大値 0x80000000 を越えた場合は 0x80000000 を第 2 オペランドに指定したレジスタに格納し，SAT フラグに 1 を設定します。

- “satsub imm, reg2”の形式

  第 2 オペランドに指定したレジスタ値から，第 1 オペランドに指定した絶対値式，または相対値式の値を減算し，結果を第 2 オペランドに指定したレジスタに格納します。ただし，結果が正の最大値 0x7fffffff を越えた場合は 0x7fffffff を，負の最大値 0x80000000 を越えた場合は 0x80000000 を第 2 オペランドに指定したレジスタに格納し，SAT フラグに 1 を設定します。

- “satsub reg1, reg2, reg3”の形式

  第 2 オペランドに指定したレジスタ値から，第 1 オペランドに指定したレジスタ値を減算し，結果を第 3 オペランドに指定したレジスタに格納します。ただし，結果が正の最大値 0x7fffffff を越えた場合は 0x7fffffff を，負の最大値 0x80000000 を越えた場合は 0x80000000 を第 3 オペランドに指定したレジスタに格納し，SAT フラグに 1 を設定します。

［詳細説明］

- “satsub reg1, reg2”，および“satsub reg1, reg2, reg3”の形式の命令に対し，as850 では，機械語命令の satsub 命令が 1 つ生成されます。

- “satsub imm, reg2”の形式で imm に次のもの，as850 では，命令展開が行われ，1 つ，または複数個の機械語命令が生成されます。注

（a）0

satsub 0, reg	satsub r0, reg

（b）-32768 ～ +32767 の範囲の絶対値式

satsub imm16, reg	satsubi imm16, reg, reg

（c）-32768 ～ +32767 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

satsub imm, reg	movhi hi(imm), r0, r1 satsub r1, reg

上記以外の場合

satsub imm, reg	movhi hi1(imm), r0, r1
	movea lo(imm), r1, r1
	satsub r1, reg

（d）-32768 ～ +32767 の範囲を越える絶対値式【V850E】

imm の値の下位 16 ビットがすべて 0 の場合

satsub imm, reg	movhi hi(imm), r0, r1 satsub r1, reg

上記以外の場合

satsub imm, reg	mov imm, r1 satsub r1, reg

（e）data ／ sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

satsub $label, reg	satsubi $label, reg, reg

（f）#label，または label を持つ相対値式，および sdata／sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

satsub #label, reg	movhi hi1(#label), r0, r1
	movea lo(#label), r1, r1
	satsub r1, reg

satsub label, reg	movhi hi1(label), r0, r1
	movea lo(label), r1, r1
	satsub r1, reg

satsub $label, reg	movhi hi1($label), r0, r1
	movea lo($label), r1, r1
	satsub r1, reg

（g）#label，または label を持つ相対値式，および sdata／sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式【V850E】

satsub #label, reg	mov #label, r1 satsub r1, reg
satsub label, reg	mov label, r1 satsub r1, reg
satsub $label, reg	mov $label, r1 satsub r1, reg

注 機械語命令の satsub 命令は，オペランドにイミーディエトをとりません。

［フラグ］

CY	MSB（Most Significant Bit）へのボローが生じた場合 1，そうでない場合 0
OV	Integer-Overflow を生じた場合 1，そうでない場合 0
S	結果が負になった場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	OV = 1 になった場合 1，そうでない場合 －

［注意事項］

- “satsub reg1, reg2”，および“satsub imm, reg2”の形式の命令に対し，ターゲット・デバイスが V850Ex の場合，第 2 オペランドに r0 を指定すると，次のメッセージが出力され，アセンブルが中止されます。

```
E3240: illegal operand (can not use r0 as destination in V850E mode）
```

- “satsub reg1, reg2”，および“satsub imm, reg2”の形式の命令に対し，ターゲット・デバイスが V850Ex 以外の場合，第 2 オペランドに r0 を指定すると，次のメッセージが出力され，アセンブルが続行されます。

```
W3013: register r0 used as destination register
```

satsubi

飽和減算を行います。（Saturated Subtract Immediate）

［指定形式］

- satsubi imm, reg1, reg2

imm に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式
- 上記のものに hi ( )，lo ( )，または hi1 ( ) を適用したもの

［機能］

第 1 オペランドに指定した絶対値式，相対値式，あるいは，hi ( )，lo ( )，または hi1 ( ) を適用したものの値を，第 2 オペランドに指定したレジスタ値から減算し，結果を第 3 オペランドに指定したレジスタに格納します。ただし，結果が正の最大値 0x7fffffff を越えた場合は 0x7fffffff を，負の最大値 0x80000000 を越えた場合は 0x80000000 を第 3 オペランドに指定したレジスタに格納し，SAT フラグに 1 を設定します。

［詳細説明］

- imm に次のものを指定した場合，as850 では，機械語命令の satsubi 命令[注]が 1 つ生成されます。

（a）-32768 ～ +32767 の範囲の絶対値式

satsubi imm16, reg1, reg2	satsubi imm16, reg1, reg2

（b）sdata ／ sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

satsubi $label, reg1, reg2	satsubi $label, reg1, reg2

（c）!label，または %label を持つ相対値式

satsubi !label, reg1, reg2	satsubi !label, reg1, reg2
satsubi %label, reg1, reg2	satsubi %label, reg1, reg2

（d）hi ( )，lo ( )，または hi1 ( ) を適用したもの

satsubi imm16, reg1, reg2	satsubi imm16, reg1, reg2

注 機械語命令の satsubi 命令は，第 1 オペランドに -32768 ～ +32767（0xffff8000 ～ 0x7fff）の範囲のイミーディエトをとります。

- imm に次のものを指定した場合，as850 では命令展開が行われ，1 つ，または複数個の機械語命令が生成されます。

（a）-32768 ～ +32767 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

`satsubi imm, reg1, reg2`	`movhi hi(imm), r0, reg2` `satsubr reg1, reg2`

imm の値の下位 16 ビットがすべて 0，ただし reg2 が r0 の場合

`satsubi imm, reg1, r0`	`movhi hi(imm), r0, r1` `satsubr reg1, r1`

上記以外の場合

`satsubi imm, reg1, reg2`	`movhi hi1(imm), r0, r1`
	`movea lo(imm), r1, reg2`
	`satsubr reg1, reg2`

上記以外，ただし reg2 が r0 の場合

`satsubi imm, reg1, reg2`	`movhi hi1(imm), r0, r1`
	`movea lo(imm), r1, r1`
	`satsubr reg1, r1`

（b）-32768 ～ +32767 の範囲を越える絶対値式【V850E】

imm の値の下位 16 ビットがすべて 0 の場合

`satsubi imm, reg1, reg2`	`movhi hi(imm), r0, reg2` `satsubr reg1, reg2`

上記以外の場合

`satsubi imm, reg1, reg2`	`mov imm, reg2` `satsubr reg1, reg2`

（c）#label，または label を持つ相対値式，および sdata／sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

reg2 が r0 の場合

satsubi #label, reg1, r0	movhi hi1(#label), r0, r1
	movea lo(#label), r1, r1
	satsubr reg1, r1

satsubi label, reg1, r0	movhi hi1(label), r0, r1
	movea lo(label), r1, r1
	satsubr reg1, r1

satsubi $label, reg1, r0	movhi hi1($label), r0, r1
	movea lo($label), r1, r1
	satsubr reg1, r1

上記以外の場合

satsubi #label, reg1, reg2	movhi hi1(#label), r0, r1
	movea lo(#label), r1, reg2
	satsubr reg1, reg2

satsubi label, reg1, reg2	movhi hi1(label), r0, r1
	movea lo(label), r1, reg2
	satsubr reg1, reg2

satsubi $label, reg1, reg2	movhi hi1($label), r0, r1
	movea lo($label), r1, reg2
	satsubr reg1, reg2

（d）#label，または label を持つ相対値式，および sdata／sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式【V850E】

satsubi #label, reg1, reg2	movhi #label, reg2 satsubr reg1, reg2
satsubi label, reg1, reg2	mov label, reg2 satsubr reg1, reg2
satsubi $label, reg1, reg2	mov $label, reg2 satsubr reg1, reg2

［フラグ］

CY	MSB（Most Significant Bit）へのボローが生じた場合 1，そうでない場合 0
OV	Integer-Overflow を生じた場合 1，そうでない場合 0
S	結果が負になった場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	OV = 1 になった場合 1，そうでない場合 －

［注意事項］

- ターゲット・デバイスが V850Ex の場合，第 2 オペランドに r0 を指定すると，次のメッセージが出力され，アセンブルが中止されます。

```
E3240: illegal operand (can not use r0 as destination in V850E mode）
```

- ターゲット・デバイスが V850Ex 以外の場合，第 2 オペランドに r0 を指定すると，次のメッセージが出力され，アセンブルが続行されます。

```
W3013: register r0 used as destination register
```

satsubr

飽和逆減算を行います。（Saturated Subtract Reverse）

［指定形式］

- satsubr reg1, reg2
- satsubr imm, reg2

imm に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式

［機能］

- “satsubr reg1, reg2”の形式

  第 1 オペランドに指定したレジスタ値から，第 2 オペランドに指定したレジスタ値を減算し，結果を第 2 オペランドに指定したレジスタに格納します。ただし，結果が正の最大値 0x7fffffff を越えた場合は 0x7fffffff を，負の最大値 0x80000000 を越えた場合は 0x80000000 を第 2 オペランドに指定したレジスタに格納し，SAT フラグに 1 を設定します。

- “satsubr imm, reg2”の形式

  第 1 オペランドに指定した絶対値式，または相対値式の値から，第 2 オペランドに指定したレジスタの値を減算し，結果を第 2 オペランドに指定したレジスタに格納します。ただし，結果が正の最大値 0x7fffffff を越えた場合は 0x7fffffff を，負の最大値 0x80000000 を越えた場合は 0x80000000 を第 2 オペランドに指定したレジスタに格納し，SAT フラグに 1 を設定します。

［詳細説明］

- “satsubr reg1, reg2”の形式の命令に対し，as850 では，機械語命令の satsubr 命令が 1 つ生成されます。
- “satsubr imm, reg2”の形式で imm に次のものを指定した場合，as850 では，命令展開が行われ，1 つ，または複数個の機械語命令が生成されます。**注**

（a）0

`satsubr 0, reg`	`satsubr r0, reg`

（b）-16 ～ +15 の範囲の絶対値式（0 以外）

`satsubr imm5, reg`	`mov imm5, r1` `satsubr r1, reg`

（c）-16 ～ +15 の範囲を越え，-32768 ～ +32767 の範囲の絶対値式

satsubr imm16, reg	movea imm16, r0, r1 satsubr r1, reg

（d）imm に -32768 ～ +32767 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

satsubr imm, reg	movhi hi(imm), r0, r1 satsubr r1, reg

上記以外の場合

satsubr imm, reg	movhi hi1(imm), r0, r1
	movea lo(imm), r1, r1
	satsubr r1, reg

（e）imm に -32768 ～ +32767 の範囲を越える絶対値式【V850E】

imm の値の下位 16 ビットがすべて 0 の場合

satsubr imm, reg	movhi hi(imm), r0, r1 satsubr r1, reg

上記以外の場合

satsubr imm, reg	mov imm, r1 satsubr r1, reg

（f）sdata ／ sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

satsubr $label, reg	movea $label, r0, r1 satsubr r1, reg

（g）#label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

satsubr #label, reg	movhi hi1(#label), r0, r1
	movea lo(#label), r1, r1
	satsubr r1, reg

satsubr label, reg	movhi hi1(label), r0, r1
	movea lo(label), r1, r1
	satsubr r1, reg

satsubr $label, reg	movhi hi1($label), r0, r1
	movea lo($label), r1, r1
	satsubr r1, reg

（h）#label，または label を持つ相対値式，および sdata／sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式【V850E】

satsubr #label, reg	mov #label, r1 satsubr r1, reg
satsubr label, reg	mov label, r1 satsubr r1, reg
satsubr $label, reg	mov $label, r1 satsubr r1, reg

注 機械語命令の satsubr 命令は，オペランドにイミーディエトをとりません。

［フラグ］

CY	MSB（Most Significant Bit）へのボローが生じた場合 1，そうでない場合 0
OV	Integer-Overflow を生じた場合 1，そうでない場合 0
S	結果が負になった場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	OV = 1 になった場合 1，そうでない場合 －

［注意事項］

- ターゲット・デバイスが V850Ex の場合，第 2 オペランドに r0 を指定すると，次のメッセージが出力され，アセンブルが中止されます。

```
E3240: illegal operand (can not use r0 as destination in V850E mode）
```

- ターゲット・デバイスが V850Ex 以外の場合，第 2 オペランドに r0 を指定すると，次のメッセージが出力され，アセンブルが続行されます。

```
W3013: register r0 used as destination register
```

4. 5. 9 論理演算命令

この項では，論理演算命令について説明します。次に，この項において説明する命令を示します。

表 4―53 論理演算命令

命令	意味
or	論理和
ori	論理和（イミディエト）
xor	排他的論理和
xori	排他的論理和（イミーディエト）
and	論理積
andi	論理積（イミーディエト）
not	論理否定（1 の補数をとる）
shr	論理右シフト
sar	算術右シフト
shl	論理左シフト
sxb	バイト・データの符号拡張【V850E】
sxh	ハーフワード・データの符号拡張【V850E】
zxb	バイト・データのゼロ拡張【V850E】
zxh	ハーフワード・データのゼロ拡張【V850E】
bsh	ハーフワード・データのバイト・スワップ【V850E】
bsw	ワード・データのバイト・スワップ【V850E】
hsh	ハーフワード・データのハーフワード・スワップ【V850E2】
hsw	ワード・データのハーフワード・スワップ【V850E】
tst	テスト
sch0l	MSB 側からのビット（0）検索【V850E2】
sch0r	LSB 側からのビット（0）検索【V850E2】
sch1l	MSB 側からのビット（1）検索【V850E2】
sch1r	LSB 側からのビット（1）検索【V850E2】

or

論理和を行います。（Or）

［指定形式］

- or reg1, reg2
- or imm, reg2

imm に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式

［機能］

- “or reg1, reg2”の形式

  第 1 オペランドに指定したレジスタ値と，第 2 オペランドに指定したレジスタ値の論理和をとり，結果を第 2 オペランドに指定したレジスタに格納します。

- “or imm, reg2”の形式

  第 1 オペランドに指定した絶対値式，または相対値式の値と，第 2 オペランドに指定したレジスタ値の論理和をとり，結果を第 2 オペランドに指定したレジスタに格納します。

［詳細説明］

- “or reg1, reg2”の形式の命令に対し，as850 では，機械語命令の or 命令が 1 つ生成されます。
- “or imm, reg2”の形式で imm に次のものを指定した場合，as850 では，命令展開が行われ，1 つ，または複数個の機械語命令が生成されます。**注**

（a）0

`or 0, reg`	`or r0, reg`

（b）1 ～ 65535 の範囲の絶対値式

`or imm5, reg`	`ori imm16, reg, reg`

（c）-16 ～ -1 の範囲の絶対値式

`or imm16, reg`	`mov imm5, r1` `or r1, reg`

（d）-32768 ～ -17 の範囲の絶対値式

or imm16, reg	movea imm16, r0, r1 or r1, reg

（e）上記の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

or imm, reg	movhi hi(imm), r0, r1 or r1, reg

上記以外の場合

or imm, reg	movhi hi1(imm), r0, r1
	movea lo(imm), r1, r1
	or r1, reg

（f）上記の範囲を越える絶対値式【V850E】

imm の値の下位 16 ビットがすべて 0 の場合

or imm, reg	movhi hi(imm), r0, r1 or r1, reg

上記以外の場合

or imm, reg	mov imm, r1 or r1, reg

（g）sdata ／ sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

or $label, reg	movea $label, r0, r1 or r1, reg

（h）#label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

`or #label, reg`	`movhi hi1(#label), r0, r1`
	`movea lo(#label), r1, r1`
	`or r1, reg`

`or label, reg`	`movhi hi1(label), r0, r1`
	`movea lo(label), r1, r1`
	`or r1, reg`

`or $label, reg`	`movhi hi1($label), r0, r1`
	`movea lo($label), r1, r1`
	`or r1, reg`

（i）#label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式【V850E】

`or #label, reg`	`mov #label, r1` `or r1, reg`
`or label, reg`	`mov label, r1` `or r1, reg`
`or $label, reg`	`mov $label, r1` `or r1, reg`

注 機械語命令の or 命令は，オペランドにイミーディエトをとりません。

［フラグ］

CY	－
OV	0
S	演算結果のワード・データの MSB が 1 の場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	－

ori

論理和を行います。（Or Immediate）

［指定形式］

- ori imm, reg1, reg2

imm に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式
- 上記のものに hi ( )，lo ( )，または hi1 ( ) を適用したもの

［機能］

第 1 オペランドに指定した絶対値式，相対値式，あるいは，hi ( )，lo ( )，または hi1 ( ) を適用した値と，第 2 オペランドに指定したレジスタ値の論理和をとり，結果を第 3 オペランドに指定したレジスタに格納します。

［詳細説明］

- imm に次のものを指定した場合，as850 では，機械語命令の ori 命令[注]が 1 つ生成されます。

（a）0 ～ 65535 の範囲の絶対値式

`ori imm16, reg1, reg2`	`ori imm16, reg1, reg2`

（b）!label，または %label を持つ相対値式

`ori !label, reg1, reg2`	`ori !label, reg1, reg2`
`ori %label, reg1, reg2`	`ori %label, reg1, reg2`

（c）hi ( )，lo ( )，または hi1 ( ) を適用したもの

`ori imm16, reg1, reg2`	`ori imm16, reg1, reg2`

注 機械語命令の ori 命令は，第 1 オペランドに 0 ～ 65535（0 ～ 0xffff）のイミーディエトをとります。

- imm に次のものを指定した場合，as850 では，命令展開が行われ，1 つ，または複数個の機械語命令が生成されます。

（a）-16 ～ -1 の範囲の絶対値式

ori imm5, reg1, reg2	mov imm5, reg2 or reg1, reg2

（b）-32768 ～ -17 の範囲の絶対値式

reg2 が r0 の場合

ori imm16, reg1, r0	movea imm16, r0, r1 or reg1, r1

上記以外の場合

ori imm16, reg1, reg2	movea imm16, r0, reg2 or reg1, reg2

（c）上記の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

ori imm, reg1, reg2	movhi hi(imm), r0, reg2 or reg1, reg2

imm の値の下位 16 ビットがすべて 0，ただし reg2 が r0 の場合

ori imm, reg1, r0	movhi hi(imm), r0, r1 or reg1, r1

上記以外の場合

ori imm, reg1, reg2	movhi hi1(imm), r0, r1
	movea lo(imm), r1, reg2
	or reg1, reg2

上記以外，ただし reg2 が r0 の場合

ori imm, reg1, r0	movhi hi1(imm), r0, r1
	movea lo(imm), r1, r1
	or reg1, r1

（d）上記の範囲を越える絶対値式【V850E】

imm の値の下位 16 ビットがすべて 0 の場合

ori imm, reg1, reg2	movhi hi(imm), r0, reg2 or reg1, reg2

imm の値の下位 16 ビットがすべて 0，ただし reg2 が r0 の場合

ori imm, reg1, r0	movhi hi(imm), r0, r1 or reg1, r1

上記以外の場合

ori imm, reg1, reg2	mov imm, reg2 or reg1, reg2

上記以外，ただし reg2 が r0 の場合

ori imm, reg1, r0	mov imm, r1 or reg1, r1

（e）sdata ／ sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

reg2 が r0 の場合

ori $label, reg1, r0	movea $label, r0, r1 or reg1, r1

上記以外の場合

ori $label, reg1, reg2	movea $label, r0, reg2 or reg1, reg2

（f）#label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

reg2 が r0 の場合

ori #label, reg1, r0	movhi hi1(#label), r0, r1
	movea lo(#label), r1, r1
	or reg1, r1

ori label, reg1, r0	movhi hi1(label), r0, r1
	movea lo(label), r1, r1
	or reg1, r1

ori $label, reg1, r0	movhi hi1($label), r0, r1
	movea lo($label), r1, r1
	or reg1, r1

上記以外の場合

ori #label, reg1, reg2	movhi hi1(#label), r0, r1
	movea lo(#label), r1, reg2
	or reg1, reg2

ori label, reg1, reg2	movhi hi1(label), r0, r1
	movea lo(label), r1, reg2
	or reg1, reg2

ori $label, reg1, reg2	movhi hi1($label), r0, r1
	movea lo($label), r1, reg2
	or reg1, reg2

（g）#label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式【V850E】

reg2 が r0 の場合

ori #label, reg1, r0	mov #label, r1 or reg1, r1
ori label, reg1, r0	mov label, r1 or reg1, r1
ori $label, reg1, r0	mov $label, r1 or reg1, r1

上記以外の場合

ori　　#label, reg1, reg2	mov　　#label, reg2 or　　reg1, reg2
ori　　label, reg1, reg2	mov　　label, reg2 or　　reg1, reg2
ori　　$label, reg1, reg2	mov　　$label, reg2 or　　reg1, reg2

［フラグ］

CY	－
OV	0
S	演算結果のワード・データの MSB が 1 の場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	－

xor

排他的論理和を行います。（Exclusive Or）

［指定形式］

- xor reg1, reg2
- xor imm, reg2

imm に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式

［機能］

- “xor reg1, reg2”の形式

  第 1 オペランドに指定したレジスタ値と，第 2 オペランドに指定したレジスタ値の排他的論理和をとり，結果を第 2 オペランドに指定したレジスタに格納します。

- “xor imm, reg2”の形式

  第 1 オペランドに指定した絶対値式，または相対値式の値と，第 2 オペランドに指定したレジスタ値の排他的論理和をとり，結果を第 2 オペランドに指定したレジスタに格納します。

［詳細説明］

- “xor reg1, reg2”の形式の命令に対し，as850 では，機械語命令の xor 命令が 1 つ生成されます。
- “xor imm, reg2”の形式で imm に次のものを指定した場合，as850 では，命令展開が行われ，1 つ，または複数個の機械語命令が生成されます。**注**

（a）0

`xor 0, reg`	`xor r0, reg`

（b）1 ～ 65535 の範囲の絶対値式

`xor imm16, reg`	`xori imm16, reg, reg`

（c）-16 ～ -1 の範囲の絶対値式

`xor imm5, reg`	`mov imm5, r1` `xor r1, reg`

（d）-32768 ～ -17 の範囲の絶対値式

xor imm16, reg	movea imm16, r0, r1 xor r1, reg

（e）上記の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

xor imm, reg	movhi hi(imm), r0, r1 xor r1, reg

上記以外の場合

xor imm, reg	movhi hi1(imm), r0, r1
	movea lo(imm), r1, r1
	xor r1, reg

（f）上記の範囲を越える絶対値式【V850E】

imm の値の下位 16 ビットがすべて 0 の場合

xor imm, reg	movhi hi(imm), r0, r1 xor r1, reg

上記以外の場合

xor imm, reg	mov imm, r1 xor r1, reg

（g）sdata ／ sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

xor $label, reg	movea $label, r0, r1 xor r1, reg

（h）label，または label を持つ相対値式，および sdata／sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

`xor #label, reg`	`movhi hi1(#label), r0, r1`
	`movea lo(#label), r1, r1`
	`xor r1, reg`

`xor label, reg`	`movhi hi1(label), r0, r1`
	`movea lo(label), r1, r1`
	`xor r1, reg`

`xor $label, reg`	`movhi hi1($label), r0, r1`
	`movea lo($label), r1, r1`
	`xor r1, reg`

（i）#label，または label を持つ相対値式，および sdata／sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式【V850E】

`xor #label, reg`	`mov #label, r1` `xor r1, reg`
`xor label, reg`	`mov label, r1` `xor r1, reg`
`xor $label, reg`	`mov $label, r1` `xor r1, reg`

注 機械語命令の xor 命令は，オペランドにイミーディエトをとりません。

［フラグ］

CY	－
OV	0
S	演算結果のワード・データの MSB が 1 の場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	－

xori

排他的論理和を行います。（Exclusive Or Immediate）

［指定形式］

- xori imm, reg1, reg2

imm に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式
- 上記のものに hi ( )，lo ( )，または hi1 ( ) を適用したもの

［機能］

第 1 オペランドに指定した絶対値式，相対値式，あるいは，hi ( )，lo ( )，または hi1 ( ) を適用した値と，第 2 オペランドに指定したレジスタ値の排他的論理和をとり，結果を第 3 オペランドに指定したレジスタに格納します。

［詳細説明］

- imm に次のものを指定した場合，as850 では，機械語命令の xori 命令[注]が 1 つ生成されます。

（a）0 ～ 65535 の範囲の絶対値式

`xori imm16, reg1, reg2`	`xori imm16, reg1, reg2`

（b）!label，または %label を持つ相対値式

`xori !label, reg1, reg2`	`xori !label, reg1, reg2`
`xori %label, reg1, reg2`	`xori %label, reg1, reg2`

（c）hi ( )，lo ( )，または hi1 ( ) を適用したもの

`xori imm16, reg1, reg2`	`xori imm16, reg1, reg2`

注 機械語命令の xori 命令は，第 1 オペランドに 0 ～ 65535（0 ～ 0xffff）のイミーディエトをとります。

- imm に次のものを指定した場合，as850 では，命令展開が行われ，1 つ，または複数個の機械語命令が生成されます。

（a）-16 ～ -1 の範囲の絶対値式

xori imm5, reg1, reg2	mov imm5, reg2 xor reg1, reg2

（b）-32768 ～ -17 の範囲の絶対値式

reg2 が r0 の場合

xori imm16, reg1, r0	movea imm16, r0, r1 xor reg1, r1

上記以外の場合

xori imm16, reg1, reg2	movea imm16, r0, reg2 xor reg1, reg2

（c）上記の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

xori imm, reg1, reg2	movhi hi(imm), r0, reg2 xor reg1, reg2

imm の値の下位 16 ビットがすべて 0，ただし reg2 が r0 の場合

xori imm, reg1, r0	movhi hi(imm), r0, r1 xor reg1, r1

上記以外の場合

xori imm, reg1, reg2	movhi hi1(imm), r0, r1
	movea lo(imm), r1, reg2
	xor reg1, reg2

上記以外，ただし reg2 が r0 の場合

xori imm, reg1, r0	movhi hi1(imm), r0, r1
	movea lo(imm), r1, r1
	xor reg1, r1

（d）上記の範囲を越える絶対値式【V850E】

imm の値の下位 16 ビットがすべて 0 の場合

xori imm, reg1, reg2	movhi hi(imm), r0, reg2 xor reg1, reg2

imm の値の下位 16 ビットがすべて 0，ただし reg2 が r0 の場合

xori imm, reg1, r0	movhi hi(imm), r0, r1 xor reg1, r1

上記以外の場合

xori imm, reg1, reg2	mov imm, reg2 xor reg1, reg2

上記以外，ただし reg2 が r0 の場合

xori imm, reg1, r0	mov imm, r1 xor reg1, r1

（e）sdata ／ sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

reg2 が r0 の場合

xori $label, reg1, r0	movea $label, r0, r1 xor reg1, r1

上記以外の場合

xori $label, reg1, reg2	movea $label, r0, reg2 xor reg1, reg2

（f）#label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

reg2 が r0 の場合

xori #label, reg1, r0	movhi hi1(#label), r0, r1
	movea lo(#label), r1, r1
	xor reg1, r1

xori label, reg1, r0	movhi hi1(label), r0, r1
	movea lo(label), r1, r1
	xor reg1, r1

xori $label, reg1, r0	movhi hi1($label), r0, r1
	movea lo($label), r1, r1
	xor reg1, r1

上記以外の場合

xori #label, reg1, reg2	movhi hi1(#label), r0, r1
	movea lo(#label), r1, reg2
	xor reg1, reg2

xori label, reg1, reg2	movhi hi1(label), r0, r1
	movea lo(label), r1, reg2
	xor reg1, reg2

xori $label, reg1, reg2	movhi hi1($label), r0, r1
	movea lo($label), r1, reg2
	xor reg1, reg2

（g）#label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式【V850E】

reg2 が r0 の場合

xori #label, reg1, r0	mov #label, r1 xor reg1, r1
xori label, reg1, r0	mov label, r1 xor reg1, r1
xori $label, reg1, r0	mov $label, r1 xor reg1, r1

上記以外の場合

xori　#label, reg1, reg2	mov　#label, reg2 xor　reg1, reg2
xori　label, reg1, reg2	mov　label, reg2 xor　reg1, reg2
xori　$label, reg1, reg2	mov　$label, reg2 xor　reg1, reg2

［フラグ］

CY	－
OV	0
S	演算結果のワード・データの MSB が 1 の場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	－

and

論理積を行います。（And）

［指定形式］

- and reg1, reg2
- and imm, reg2

imm に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式

［機能］

- “and reg1, reg2”の形式

  第 1 オペランドに指定したレジスタ値と，第 2 オペランドに指定したレジスタ値の論理積をとり，結果を第 2 オペランドに指定したレジスタに格納します。

- “and imm, reg2”の形式

  第 1 オペランドに指定した絶対値式，または相対値式の値と，第 2 オペランドに指定したレジスタ値の論理積をとり，結果を第 2 オペランドに指定したレジスタに格納します。

［詳細説明］

- “and reg1, reg2”の形式の命令に対し，as850 では，機械語命令の and 命令が 1 つ生成されます。
- “and imm, reg2”の形式で imm に次のものを指定した場合，as850 では，命令展開が行われ，1 つ，または複数個の機械語命令が生成されます。**注**

（a）0

`and 0, reg`	`and r0, reg`

（b）1 ～ 65535 の範囲の絶対値式

`and imm16, reg`	`andi imm16, reg, reg`

（c）-16 ～ -1 の範囲の絶対値式

`and imm5, reg`	`mov imm5, r1` `and r1, reg`

（d）-32768 ～ -17 の範囲の絶対値式

and imm16, reg	movea imm16, r0, r1 and r1, reg

（e）上記の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

and imm, reg	movhi hi(imm), r0, r1 and r1, reg

上記以外の場合

and imm, reg	movhi hi1(imm), r0, r1
	movea lo(imm), r1, r1
	and r1, reg

（f）上記の範囲を越える絶対値式【V850E】

imm の値の下位 16 ビットがすべて 0 の場合

and imm, reg	movhi hi(imm), r0, r1 and r1, reg

上記以外の場合

and imm, reg	mov imm, r1 and r1, reg

（g）sdata ／ sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

and $label, reg	movea $label, r0, r1 and r1, reg

（h）#label，または label を持つ相対値式，および sdata／sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

`and #label, reg`	`movhi hi1(#label), r0, r1`
	`movea lo(#label), r1, r1`
	`and r1, reg`

`and label, reg`	`movhi hi1(label), r0, r1`
	`movea lo(label), r1, r1`
	`and r1, reg`

`and $label, reg`	`movhi hi1($label), r0, r1`
	`movea lo($label), r1, r1`
	`and r1, reg`

（i）#label，または label を持つ相対値式，および sdata／sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式【V850E】

`and #label, reg`	`mov #label, r1` `and r1, reg`
`and label, reg`	`mov label, r1` `and r1, reg`
`and $label, reg`	`mov $label, r1` `and r1, reg`

注 機械語命令の and 命令は，オペランドにイミーディエトをとりません。

［フラグ］

CY	－
OV	0
S	演算結果のワード・データの MSB が 1 の場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	－

andi

論理積を行います。（And Immediate）

［指定形式］

- andi imm, reg1, reg2

imm に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式
- 上記のものに hi ( )，lo ( )，または hi1 ( ) を適用したもの

［機能］

第 1 オペランドに指定した絶対値式，相対値式，あるいは，hi ( )，lo ( )，または hi1 ( ) を適用した値と，第 2 オペランドに指定したレジスタ値の論理積をとり，結果を第 3 オペランドに指定したレジスタに格納します。

［詳細説明］

- imm に次のものを指定した場合，as850 では，機械語命令の andi 命令[注]が 1 つ生成されます。

（a）0 ～ 65535 の範囲の絶対値式

andi imm16, reg1, reg2	andi imm16, reg1, reg2

（b）!label，または %label を持つ相対値式

andi !label, reg1, reg2	andi !label, reg1, reg2
andi %label, reg1, reg2	andi %label, reg1, reg2

（c）hi ( )，lo ( )，または hi1 ( ) を適用したもの

andi imm16, reg1, reg2	andi imm16, reg1, reg2

注 機械語命令の andi 命令は，第 1 オペランドに 0 ～ 65535（0 ～ 0xffff）のイミーディエトをとります。

- imm に次のものを指定した場合，as850 では，命令展開が行われ，1 つ，または複数個の機械語命令が生成されます。

（a）-16 ～ -1 の範囲の絶対値式

andi imm5, reg1, reg2	mov imm5, reg2 and reg1, reg2

（b）-32768 ～ -17 の範囲の絶対値式

reg2 が r0 の場合

andi imm16, reg1, r0	movea imm16, r0, r1 and reg1, r1

上記以外の場合

andi imm16, reg1, reg2	movea imm16, r0, reg2 and reg1, reg2

（c）上記の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

andi imm, reg1, reg2	movhi hi(imm), r0, reg2 and reg1, reg2

imm の値の下位 16 ビットがすべて 0，ただし reg2 が r0 の場合

andi imm, reg1, r0	movhi hi(imm), r0, r1 and reg1, r1

上記以外の場合

andi imm, reg1, reg2	movhi hi1(imm), r0, r1
	movea lo(imm), r1, reg2
	and reg1, reg2

上記以外，ただし reg2 が r0 の場合

andi imm, reg1, r0	movhi hi1(imm), r0, r1
	movea lo(imm), r1, r1
	and reg1, r1

（d）上記の範囲を越える絶対値式【V850E】

imm の値の下位 16 ビットがすべて 0 の場合

`andi imm, reg1, reg2`	`movhi hi(imm), r0, reg2` `and reg1, reg2`

imm の値の下位 16 ビットがすべて 0，ただし reg2 が r0 の場合

`andi imm, reg1, r0`	`movhi hi(imm), r0, r1` `and reg1, r1`

上記以外の場合

`andi imm, reg1, reg2`	`mov imm, reg2` `and reg1, reg2`

上記以外，ただし reg2 が r0 の場合

`andi imm, reg1, reg2`	`mov imm, r1` `and reg1, r1`

（e）sdata ／ sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

reg2 が r0 の場合

`andi $label, reg1, r0`	`movea $label, r0, r1` `and reg1, r1`

上記以外の場合

`andi $label, reg1, reg2`	`movea $label, r0, reg2` `and reg1, reg2`

（f）#label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

reg2 が r0 の場合

andi #label, reg1, r0	movhi hi1(#label), r0, r1
	movea lo(#label), r1, r1
	and reg1, r1

andi label, reg1, r0	movhi hi1(label), r0, r1
	movea lo(label), r1, r1
	and reg1, r1

andi $label, reg1, r0	movhi hi1($label), r0, r1
	movea lo($label), r1, r1
	and reg1, r1

上記以外の場合

andi #label, reg1, reg2	movhi hi1(#label), r0, r1
	movea lo(#label), r1, reg2
	and reg1, reg2

andi label, reg1, reg2	movhi hi1(label), r0, r1
	movea lo(label), r1, reg2
	and reg1, reg2

andi $label, reg1, reg2	movhi hi1($label), r0, r1
	movea lo($label), r1, reg2
	and reg1, reg2

（g）#label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式【V850E】

reg2 が r0 の場合

andi #label, reg1, r0	mov #label, r1 and reg1, r1
andi label, reg1, r0	mov label, r1 and reg1, r1
andi $label, reg1, r0	mov $label, r1 and reg1, r1

上記以外の場合

andi #label, reg1, reg2	mov #label, reg2 and reg1, reg2
andi label, reg1, reg2	mov label, reg2 and reg1, reg2
andi $label, reg1, reg2	mov $label, reg2 and reg1, reg2

［フラグ］

CY	－
OV	0
S	演算結果のワード・データの MSB が 1 の場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	－

not

論理否定（1 の補数をとる）を行います。（Not）

［指定形式］

- not reg1, reg2
- not imm, reg2

imm に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式

［機能］

- “not reg1, reg2”の形式

  第 1 オペランドに指定したレジスタ値の論理否定（1 の補数）をとり，結果を第 2 オペランドに指定したレジスタに格納します。

- “not imm, reg2”の形式

  第 1 オペランドに指定した絶対値式，または相対値式の値の論理否定（1 の補数）をとり，結果を第 2 オペランドに指定したレジスタに格納します。

［詳細説明］

- “not reg1, reg2”の形式の命令に対し，as850 では，機械語命令の not 命令が 1 つ生成されます。
- “not imm, reg2”の形式で imm に次のものを指定した場合，as850 では，命令展開が行われ，1 つ，または複数個の機械語命令が生成されます。**注**

（a）0

`not 0, reg`	`not r0, reg`

（b）-16 ～ +15 の範囲の絶対値式（0 以外）

`not imm5, reg`	`mov imm5, r1` `not r1, reg`

（c）-16 ～ +15 の範囲を越え，-32768 ～ +32767 の範囲の絶対値式

`not imm16, reg`	`movea imm16, r0, r1` `not r1, reg`

（d）imm に -32768 ～ +32767 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

not imm, reg	movhi hi(imm), r0, r1 not r1, reg

上記以外の場合

not imm, reg	movhi hi1(imm), r0, r1
	movea lo(imm), r1, r1
	not r1, reg

（e）imm に -32768 ～ +32767 の範囲を越える絶対値式【V850E】

imm の値の下位 16 ビットがすべて 0 の場合

not imm, reg	movhi hi(imm), r0, r1 not r1, reg

上記以外の場合

not imm, reg	mov imm, r1 not r1, reg

（f）sdata ／ sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

not $label, reg	movea $label, r0, r1 not r1, reg

（g）#label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

not #label, reg	movhi hi1(#label), r0, r1
	movea lo(#label), r1, r1
	not r1, reg

not label, reg	movhi hi1(label), r0, r1
	movea lo(label), r1, r1
	not r1, reg

not $label, reg	movhi hi1($label), r0, r1
	movea lo($label), r1, r1
	not r1, reg

（h）#label，または label を持つ相対値式，および sdata／sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式【V850E】

`not  #label, reg`	`mov  #label, r1` `not  r1, reg`
`not  label, reg`	`mov  label, r1` `not  r1, reg`
`not  $label, reg`	`mov  $label, r1` `not  r1, reg`

注　機械語命令の not 命令は，オペランドにイミーディエトをとりません。

［フラグ］

CY	－
OV	0
S	演算結果のワード・データの MSB が 1 の場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	－

shr

論理右シフトを行います。（Shift Logical Right）

［指定形式］

- shr reg1, reg2
- shr imm5, reg2
- shr reg1, reg2, reg3 【V850E2】

imm5 に指定できるものを次に示します。

- 5 ビット幅までの値を持つ絶対値式

［機能］

-“shr reg1, reg2”の形式

第 1 オペランドに指定したレジスタ値の下位 5 ビットで示されるビット数分，第 2 オペランドに指定したレジスタ値を右に論理シフトし，結果を第 2 オペランドに指定したレジスタに格納します。

-“shr imm5, reg2”の形式

第 1 オペランドに指定した絶対値式の値で示されるビット数分，第 2 オペランドに指定したレジスタ値を右に論理シフトし，結果を第 2 オペランドに指定したレジスタに格納します。

-“shr reg1, reg2, reg3”の形式

第 1 オペランドに指定したレジスタ値の下位 5 ビットで示されるビット数分，第 2 オペランドに指定したレジスタ値を右に論理シフトし，結果を第 3 オペランドに指定したレジスタに格納します。

［詳細説明］

as850 では，機械語命令の shr 命令が 1 つ生成されます。

［フラグ］

CY	最後にシフト・アウトしたビットの値が 1 の場合 1，そうでない場合 0 （指定したビット数が 0 の場合 0）
OV	0
S	結果が負になった場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	―

［注意事項］

- “shr imm5, reg2”の形式で imm5 に 0 ～ 31 の範囲を越える絶対値式を指定した場合，次のメッセージが出力され，指定した値の下位 5 ビット注が用いられてアセンブルが続行されます。

```
W3011: illegal operand (range error in immediate).
```

注　機械語命令の shr 命令は，第 1 オペランドに 0 ～ 31（0x0 ～ 0x1f）のイミーディエトをとります。

sar

算術右シフトを行います。（Shift Arithmetic Right）

［指定形式］

- sar reg1, reg2
- sar imm5, reg2
- sar reg1, reg2, reg3 【V850E2】

imm5 に指定できるものを次に示します。

- 5 ビット幅までの値を持つ絶対値式

［機能］

-“sar reg1, reg2”の形式

第 1 オペランドに指定したレジスタの値の下位 5 ビットで示されるビット数分，第 2 オペランドに指定したレジスタ値を右に算術シフトし，結果を第 2 オペランドに指定したレジスタに格納します。

-“sar imm5, reg2”の形式

第 1 オペランドに指定した絶対値式の値で示されるビット数分，第 2 オペランドに指定したレジスタ値を右に算術シフトし，結果を第 2 オペランドに指定したレジスタに格納します。

-“sar reg1, reg2, reg3”の形式

第 1 オペランドに指定したレジスタ値の下位 5 ビットで示されるビット数分，第 2 オペランドに指定したレジスタ値を右に算術シフトし，結果を第 3 オペランドに指定したレジスタに格納します。

［詳細説明］

as850 では，機械語命令の sar 命令が 1 つ生成されます。

［フラグ］

CY	最後にシフト・アウトしたビットの値が 1 の場合 1，そうでない場合 0 （指定したビット数が 0 の場合 0）
OV	0
S	結果が負になった場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	−

［注意事項］

- “sar imm5, reg2”の形式で imm5 に 0 ～ 31 の範囲を越える絶対値式を指定した場合，次のメッセージが出力され，指定した値の下位 5 ビット注が用いられてアセンブルが続行されます。

```
W3011: illegal operand (range error in immediate).
```

注　機械語命令の sar 命令は，第 1 オペランドに 0 ～ 31（0x0 ～ 0x1f）のイミーディエトをとります。

shl

論理左シフトを行います。（Shift Logical Left）

［指定形式］

- shl reg1, reg2
- shl imm5, reg2
- shl reg1, reg2, reg3 【V850E2】

imm5 に指定できるものを次に示します。

- 5 ビット幅までの値を持つ絶対値式

［機能］

-“shl reg1, reg2”の形式

第 1 オペランドに指定したレジスタ値の下位 5 ビットで示されるビット数分，第 2 オペランドに指定したレジスタ値を左に論理シフトし，結果を第 2 オペランドに指定したレジスタに格納します。

-“shl imm5, reg2”の形式

第 1 オペランドに指定した絶対値式の値で示されるビット数分，第 2 オペランドに指定したレジスタ値を左に論理シフトし，結果を第 2 オペランドに指定したレジスタに格納します。

-“shl reg1, reg2, reg3”の形式

第 1 オペランドに指定したレジスタ値の下位 5 ビットで示されるビット数分，第 2 オペランドに指定したレジスタ値を左に論理シフトし，結果を第 3 オペランドに指定したレジスタに格納します。

［詳細説明］

as850 では，機械語命令の shl 命令が 1 つ生成されます。

［フラグ］

CY	最後にシフト・アウトしたビットの値が 1 の場合 1，そうでない場合 0 （指定したビット数が 0 の場合 0）
OV	0
S	結果が負になった場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	−

［注意事項］

- “shl imm5, reg2”の形式で imm5 に 0 ～ 31 の範囲を越える絶対値式を指定した場合，次のメッセージが出力され，指定した値の下位 5 ビット[注]が用いられてアセンブルが続行されます。

```
W3011: illegal operand (range error in immediate).
```

注　機械語命令の shl 命令は，第 1 オペランドに 0 ～ 31（0x0 ～ 0x1f）のイミーディエトをとります。

sxb

バイト・データの符号拡張を行います。（Sign Extend Byte）【V850E】

［指定形式］

- sxb　reg

［機能］

第 1 オペランドに指定したレジスタの下位 1 バイトのデータを，ワード長に符号拡張します。

［詳細説明］

as850 では，機械語命令の sxb 命令が 1 つ生成されます。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

sxh

ハーフワード・データの符号拡張を行います。（Sign Extend Half-word）【V850E】

［指定形式］

- sxh reg

［機能］

第 1 オペランドに指定したレジスタの下位 2 バイトのデータを，ワード長に符号拡張します。

［詳細説明］

as850 では，機械語命令の sxh 命令が 1 つ生成されます。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

zxb

バイト・データのゼロ拡張を行います。（Zero Extend Byte）【V850E】

［指定形式］

- zxb　reg

［機能］

第 1 オペランドに指定したレジスタの下位 1 バイトのデータを，ワード長にゼロ拡張します。

［詳細説明］

as850 では，機械語命令の zxb 命令が 1 つ生成されます。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

zxh

ハーフワード・データのゼロ拡張を行います。（Zero Extend Half-word）【V850E】

［指定形式］

- zxh　reg

［機能］

第 1 オペランドに指定したレジスタの下位ハーフワードのデータを，ワード長にゼロ拡張します。

［詳細説明］

as850 では，機械語命令の zxh 命令が 1 つ生成されます。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

bsh

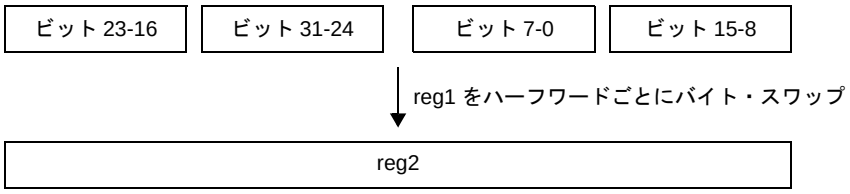
ハーフワード・データのバイト・スワップを行います。（Byte Swap Half-word）【V850E】

［指定形式］

- bsh reg1, reg2

［機能］

第 1 オペランドに指定したレジスタ値を，ハーフワード単位でバイト・スワップして，第 2 オペランドに指定したレジスタに格納します。



［詳細説明］

as850 では，機械語命令の bsh 命令が 1 つ生成されます。

［フラグ］

CY	レジスタの下位ハーフワード中の 1 つ以上のバイトが 0 の場合 1，そうでない場合 0
OV	0
S	演算結果のワード・データの MSB が 1 の場合 1，そうでない場合 0
Z	演算結果の下位ハーフ・ワード・データが 0 になった場合 1，そうでない場合 0
SAT	–

bsw

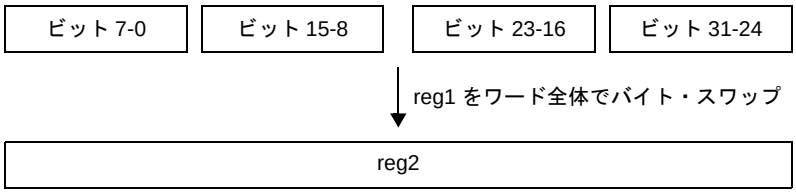
ワード・データのバイト・スワップを行います。（Byte Swap Word）【V850E】

［指定形式］

- bsw reg1, reg2

［機能］

第 1 オペランドに指定したレジスタ値を，バイト・スワップして，第 2 オペランドに指定したレジスタに格納します。



［詳細説明］

as850 では，機械語命令の bsw 命令が 1 つ生成されます。

［フラグ］

CY	レジスタのワード中の 1 つ以上のバイトが 0 の場合 1，そうでない場合 0
OV	0
S	演算結果のワード・データの MSB が 1 の場合 1，そうでない場合 0
Z	演算結果のワード・データが 0 になった場合 1，そうでない場合 0
SAT	—

hsh

ハーフワード・データのハーフワード・スワップを行います。（Half-word Swap Half-word）【V850E2】

［指定形式］

- hsh reg2, reg3

［機能］

第 1 オペランドに指定したレジスタ値を第 2 オペランドに指定したレジスタに格納し，フラグの判定結果を PSW レジスタに格納します。

［詳細説明］

as850 では，機械語命令の hsh 命令が 1 つ生成されます。

［フラグ］

CY	演算結果の下位ハーフワードが 0 の場合 1，そうでない場合 0
OV	0
S	演算結果のワード・データの MSB が 1 の場合 1，そうでない場合 0
Z	演算結果の下位ハーフワードが 0 の場合 1，そうでない場合 0
SAT	−

hsw

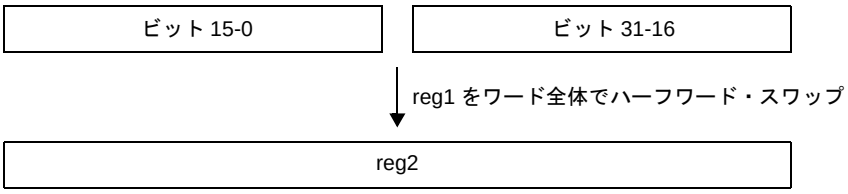
ワード・データのハーフワード・スワップを行います。（Half-word Swap Word）【V850E】

［指定形式］

- hsw reg1, reg2

［機能］

第 1 オペランドに指定したレジスタ値を，ハーフワード・スワップして，第 2 オペランドに指定したレジスタに格納します。



［詳細説明］

as850 では，機械語命令の hsw 命令が 1 つ生成されます。

［フラグ］

CY	演算結果のワード・データ中に 0 のハーフワードが 1 つ以上含まれる場合 1，そうでない場合 0
OV	0
S	演算結果のワード・データの MSB が 1 の場合 1，そうでない場合 0
Z	演算結果のワード・データが 0 になった場合 1，そうでない場合 0
SAT	−

tst

テストを行います。（Test）

［指定形式］

- tst reg1, reg2
- tst imm, reg2

imm に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式

［機能］

- “tst reg1, reg2”の形式

  第 2 オペランドに指定したレジスタ値と，第 1 オペランドに指定したレジスタ値の論理積をとり，結果は格納せず，フラグのみを設定します。

- “tst imm, reg2”の形式

  第 2 オペランドに指定したレジスタ値と，第 1 オペランドに指定した絶対値式，または相対値式の値の論理積をとり，結果は格納せず，フラグのみを設定します。

［詳細説明］

- “tst reg1, reg2”の形式の命令に対し，as850 では，機械語命令の tst 命令が 1 つ生成されます。
- “tst imm, reg2”の形式で imm に次のものを指定した場合，as850 では，命令展開が行われ，1 つ，または複数個の機械語命令が生成されます。

（a）0

`tst 0, reg`	`tst r0, reg`

（b）-16 ～ +15 の範囲の絶対値式（0 以外）

`tst imm5, reg`	`mov imm5, r1` `tst r1, reg`

（c）-16 ～ +15 の範囲を越え，-32768 ～ +32767 の範囲の絶対値式

`tst imm16, reg`	`movea imm16, r0, r1` `tst r1, reg`

（d）imm に -32768 ～ +32767 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

tst imm, reg	movhi hi(imm), r0, r1 tst r1, reg

上記以外の場合

tst imm, reg	movhi hi1(imm), r0, r1
	movea lo(imm), r1, r1
	tst r1, reg

（e）imm に -32768 ～ +32767 の範囲を越える絶対値式【V850E】

imm の値の下位 16 ビットがすべて 0 の場合

tst imm, reg	movhi hi(imm), r0, r1 tst r1, reg

上記以外の場合

tst imm, reg	mov imm, r1 tst r1, reg

（f）sdata ／ sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

tst $label, reg	movea $label, r0, r1 tst r1, reg

（g）#label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

tst #label, reg	movhi hi1(#label), r0, r1
	movea lo(#label), r1, r1
	tst r1, reg

tst label, reg	movhi hi1(label), r0, r1
	movea lo(label), r1, r1
	tst r1, reg

tst $label, reg	movhi hi1($label), r0, r1
	movea lo($label), r1, r1
	tst r1, reg

（h）#label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式【V850E】

`tst #label, reg`	`mov #label, r1` `tst r1, reg`
`tst label, reg`	`mov label, r1` `tst r1, reg`
`tst $label, reg`	`mov $label, r1` `tst r1, reg`

［フラグ］

CY	—
OV	0
S	演算結果のワード・データの MSB が 1 の場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	—

sch0l

MSB 側からのビット（0）検索を行います。（Search zero from left）【V850E2】

［指定形式］

- sch0l reg2, reg3

［機能］

第 1 オペランドに指定したレジスタのワード・データを左側（MSB 側）から検索し，最初にビット（0）が見つかった位置を第 2 オペランドに指定したレジスタに 16 進数で格納します。（たとえば，第 1 オペランドに指定したレジスタのビット 31 が 0 の場合は，第 2 オペランドに指定したレジスタに 01H を格納します）。

ビット（0）が見つからなかった場合は，第 2 オペランドに指定したレジスタに 0 を書き込み，同時に Z フラグをセット（1）します。最後にビット（0）が見つかった場合は，CY フラグをセット（1）します。

［詳細説明］

as850 では，機械語命令の sch0l 命令が 1 つ生成されます。

［フラグ］

CY	最後にビット（0）が見つかった場合 1，そうでない場合 0
OV	0
S	0
Z	ビット（0）が見つからなかった場合 1，そうでない場合 0
SAT	—

sch0r

LSB 側からのビット（0）検索を行います。（Search zero from right）【V850E2】

［指定形式］

- sch0r reg2, reg3

［機能］

第 1 オペランドに指定したレジスタのワード・データを右側（LSB 側）から検索し，最初にビット（0）が見つかった位置を第 2 オペランドに指定したレジスタに 16 進数で格納します（たとえば，第 1 オペランドに指定したレジスタのビット 0 が 0 の場合は，第 2 オペランドに指定したレジスタ に 01H を格納します）。

ビット（0）が見つからなかった場合は，第 2 オペランドに指定したレジスタに 0 を書き込み，同時に Z フラグをセット（1）します。最後にビット（0）が見つかった場合は，CY フラグをセット（1）します。

［詳細説明］

as850 では，機械語命令の sch0r 命令が 1 つ生成されます。

［フラグ］

CY	最後にビット（0）が見つかった場合 1，そうでない場合 0
OV	0
S	0
Z	ビット（0）が見つからなかった場合 1，そうでない場合 0
SAT	—

sch1l

MSB 側からのビット（1）検索を行います。（Search one from left）【V850E2】

［指定形式］

- sch1l reg2, reg3

［機能］

第 1 オペランドに指定したレジスタのワード・データを左側（MSB 側）から検索し，最初にビット（1）が見つかった位置を第 2 オペランドに指定したレジスタに 16 進数で書き込みます（たとえば，第 1 オペランドに指定したレジスタのビット 31 が 1 の場合は，第 2 オペランドに指定したレジスタに 01H を格納します）。

ビット（1）が見つからなかった場合は，第 2 オペランドに指定したレジスタに 0 を書き込み，同時に Z フラグをセット（1）します。最後にビット（1）が見つかった場合は，CY フラグをセット（1）します。

［詳細説明］

as850 では，機械語命令の sch1l 命令が 1 つ生成されます。

［フラグ］

CY	最後にビット（1）が見つかった場合 1，そうでない場合 0
OV	0
S	0
Z	ビット（1）が見つからなかった場合 1，そうでない場合 0
SAT	—

sch1r

LSB 側からのビット（1）検索を行います。（Search zero from right）【V850E2】

［指定形式］

- sch1r reg2, reg3

［機能］

第 1 オペランドに指定したレジスタのワード・データを右側（LSB 側）から検索し，最初にビット（1）が見つかった位置を第 2 オペランドに指定したレジスタに 16 進数で格納します（たとえば，第 1 オペランドに指定したレジスタのビット 0 が 1 の場合は，第 2 オペランドに指定したレジスタに 01H を格納します）。

ビット（1）が見つからなかった場合は，第 2 オペランドに指定したレジスタに 0 を書き込み，同時に Z フラグをセット（1）します。最後にビット（1）が見つかった場合は，CY フラグをセット（1）します。

［詳細説明］

as850 では，機械語命令の sch1r 命令が 1 つ生成されます。

［フラグ］

CY	最後にビット（1）が見つかった場合 1，そうでない場合 0
OV	0
S	0
Z	ビット（1）が見つからなかった場合 1，そうでない場合 0
SAT	—

4. 5. 10　分岐命令

この項では，分岐命令について説明します。次に，この項において説明する命令を示します。

表 4―54　分岐命令

命令	意味
jmp	無条件分岐
jmp32	無条件分岐【V850E2】
jr	無条件分岐（PC 相対）
jr22	無条件分岐（PC 相対）【V850E2】
jr32	無条件分岐（PC 相対）【V850E2】
jcnd	条件分岐
jarl	ジャンプ・アンド・レジスタ・リンク
jarl22	ジャンプ・アンド・レジスタ・リンク【V850E2】
jarl32	ジャンプ・アンド・レジスタ・リンク【V850E2】

jmp

無条件分岐を行います。（Jump）

［指定形式］

- jmp [reg]
- jmp disp32[reg] 【V850E2】
- jmp addr

addr に指定できるものを次に示します。

- ラベルの絶対アドレス参照を持つ相対値式

disp32 に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式

［機能］

- “jmp [reg]”の形式

  オペランドに指定したレジスタ値が示すアドレスに制御を移します。
- “jmp disp32[reg]”の形式

  オペランドに指定したディスプレースメントとレジスタの内容を加算して得たアドレスに制御を移します。
- “jmp addr”の形式

  オペランドに指定した相対値式の値が示すアドレスに制御を移します。

［詳細説明］

- “jmp [reg]”の形式の命令に対し，as850 では，機械語命令の jmp 命令が 1 つ生成されます。
- “jmp disp32[reg]”の形式の命令に対し，as850 では，機械語命令の jmp 命令（6 バイト長命令）が 1 つ生成されます。
- “jmp addr”の形式の命令に対し，V850/V850E1 動作時には，as850 では，命令展開が行われ，複数個の機械語命令が生成されます。

【V850】

`jmp  #label`	`movhi  hi1(#label), r0, r1`
	`movea  lo(#label), r1, r1`
	`jmp  [r1]`

【V850E】

`jmp  #label`	`mov  #label, r1` `jmp  [r1]`

- “jmp addr”の形式の命令に対し，V850E2 動作時には，as850 では，機械語命令の jmp 命令（6 バイト長命令）が 1 つ生成されます。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

［注意事項］

- “jmp addr”の形式において，addr にラベルの絶対アドレス参照を持つ相対値式以外のものを指定した場合，次のメッセージが出力され，アセンブルが中止されます。

E3224: illegal operand (label reference for jmp must be #label)

jmp32

無条件分岐を行います。（Jump）【V850E2】

［指定形式］

- jmp32 disp32[reg]
- jmp32 addr

addr に指定できるものを次に示します。

- ラベルの絶対アドレス参照を持つ相対値式

disp32 に指定できるものを次に示します。

- 22 ビット幅までの値を持つ絶対値式

［機能］

- “jmp32 disp32[reg]”の形式

  オペランドに指定したディスプレースメントとレジスタの内容を加算して得たアドレスに制御を移します。

- “jmp32 addr”の形式

  オペランドに指定した相対値式の値が示すアドレスに制御を移します。

［詳細説明］

as850 では，機械語命令の jmp 命令（6 バイト長命令）が 1 つ生成されます。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

［注意事項］

- “jmp32 addr”の形式において，addr にラベルの絶対アドレス参照を持つ相対値式以外のものを指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E3224: illegal operand (label reference for jmp must be #label)
```

jr

無条件分岐（PC 相対）を行います。（Jump Relative）

［指定形式］

- jr disp22
- jr disp32 【V850E2】

disp22 に指定できるものを次に示します。

- 22 ビット幅までの値を持つ絶対値式
- ラベルの PC オフセット参照を持つ相対値式

disp32 に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- ラベルの PC オフセット参照を持つ相対値式

［機能］

- “jr disp22”の形式

  オペランドに指定した絶対値式，または相対値式の値と，現在のプログラム・カウンタ（PC）の値を加算したアドレスに制御を移します。

- “jr disp32”の形式

  オペランドに指定した絶対値式，または相対値式の値と，現在のプログラム・カウンタ（PC）の値を加算したアドレスに制御を移します。

［詳細説明］

- “jr disp22”の形式の命令に対し，disp22 に次のものを指定した場合，as850 で機械語命令の jr 命令[注]が 1 つ生成されます。

  （a）-2097152 ～ +2097151 の範囲の絶対値式

  （b）本命令と同じファイル内の同じセクションに定義を持つラベルの PC オフセット参照を持ち，-2097152 ～ +2097151 の範囲の相対値式

  （c）本命令と同じファイル内に定義を持たないか同じセクションに定義を持たないラベルの PC オフセット参照を持つ相対値式

注 機械語命令の jr は，ディスプレースメントに -2097152 ～ +2097151（0xfe00000 ～ 0x1fffff）の範囲のイミーディエトをとります。

- “jr disp32”の形式の命令に対し，as850 では，機械語命令の jr 命令（6 バイト長命令）が 1 つ生成されます。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

［注意事項］

- disp22 に，-2097152 ～ +2097151 の範囲を越える絶対値式，または本命令と同じファイル内の同じセクションに定義を持つラベルの PC オフセット参照を持ち -2097152 ～ +2097151 の範囲を越える相対値式を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E3230: illegal operand (range error in displacement)
```

- disp22 に，奇数値を持つ絶対値式，または本命令と同じファイル内の同じセクションに定義を持つラベルの PC オフセット参照を持ち奇数値を持つ相対値式を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E3226: illegal operand (must be even displacement)
```

- アセンブラオプション -Xfar_jump を指定しない場合に，disp32 に，-2097152 ～ +2097151 の範囲を越える絶対値式，または本命令と同じファイル内の同じセクションに定義を持つラベルの PC オフセット参照を持ち -2097152 ～ +2097151 の範囲を越える相対値式を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E3230: illegal operand (range error in displacement)
```

jr22

無条件分岐（PC 相対）を行います。（Jump Relative）【V850E2】

［指定形式］

- jr22 disp22

disp22 に指定できるものを次に示します。

- 22 ビット幅までの値を持つ絶対値式
- ラベルの PC オフセット参照を持つ相対値式

［機能］

オペランドに指定した絶対値式，または相対値式の値と，現在のプログラム・カウンタ（PC）の値を加算したアドレスに制御を移します。

［詳細説明］

- disp22 に次のものを指定した場合，as850 では，機械語命令の jr 命令[注]が 1 つ生成されます。

（a）-2097152 ～ +2097151 の範囲の絶対値式

（b）本命令と同じファイル内の同じセクションに定義を持つラベルの PC オフセット参照を持ち，-2097152 ～ +2097151 の範囲の相対値式

（c）本命令と同じファイル内に定義を持たないか同じセクションに定義を持たないラベルの PC オフセット参照を持つ相対値式

注 機械語命令の jr は，ディスプレースメントに -2097152 ～ +2097151（0xfe00000 ～ 0x1fffff）の範囲のイミーディエトをとります。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

［注意事項］

- disp22 に，-2097152 ～ +2097151 の範囲を越える絶対値式，または本命令と同じファイル内の同じセクションに定義を持つラベルの PC オフセット参照を持ち -2097152 ～ +2097151 の範囲を越える相対値式を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E3230: illegal operand (range error in displacement)
```

- disp22 に，奇数値を持つ絶対値式，または本命令と同じファイル内の同じセクションに定義を持つラベルの PC オフセット参照を持ち奇数値を持つ相対値式を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E3226: illegal operand (must be even displacement)
```

jr32

無条件分岐（PC 相対）を行います。（Jump Relative）【V850E2】

［指定形式］

- jr32 disp32

disp32 に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- ラベルの PC オフセット参照を持つ相対値式

［機能］

オペランドに指定した絶対値式，または相対値式の値と，現在のプログラム・カウンタ（PC）の値を加算したアドレスに制御を移します。

［詳細説明］

as850 では，機械語命令の jr 命令（6 バイト長命令）が 1 つ生成されます。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

［注意事項］

- disp32 に，奇数値を持つ絶対値式，または本命令と同じファイル内の同じセクションに定義を持つラベルの PC オフセット参照を持ち奇数値を持つ相対値式を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E3226: illegal operand (must be even displacement)
```

jcnd

条件分岐を行います。（Jump on Condition）

［指定形式］

- j*cnd* disp22

disp22 に指定できるものを次に示します。

- 22 ビット幅までの値を持つ絶対値式
- ラベルの PC オフセット参照を持つ相対値式

［機能］

cnd 部分の文字列で示されるフラグ状態（「表 4―55 jcnd 命令」を参照）と，現在のフラグ状態を比較し，一致した場合は，オペランドに指定した絶対値式，または相対値式の値と現在のプログラム・カウンタ（PC）の値を加算したアドレスに制御を移します。**注**

注 機械語命令の jr は，ディスプレースメントに -2097152 ～ +2097151（0xfe00000 ～ 0x1fffff）の範囲のイミーディエトをとります。

表 4―55 j*cnd* 命令

命令	フラグ状態	フラグ状態の意味
jgt	((S xor OV) or Z) = 0	Greater than（signed）
jge	(S xor OV) = 0	Greater than or equal（signed）
jlt	(S xor OV) = 1	Less than（signed）
jle	((S xor OV) or Z) = 1	Less than or equal（signed）
jh	(CY or Z) = 0	Higher（Greater than）
jnl	CY = 0	Not lower（Greater than or equal）
jl	CY = 1	Lower（Less than）
jnh	(CY or Z) = 1	Not higher（Less than or equal）
je	Z = 1	Equal
jne	Z = 0	Not equal
jv	OV = 1	Overflow
jnv	OV = 0	No overflow
jn	S = 1	Negative
jp	S = 0	Positive
jc	CY = 1	Carry
jnc	CY = 0	No carry
jz	Z = 1	Zero

命令	フラグ状態	フラグ状態の意味
jnz	Z = 0	Not zero
jbr	―	Always（無条件）
jsa	SAT = 1	Saturated

［詳細説明］

- disp22 に次のものを指定した場合，as850 では，機械語命令の b*cnd* 命令[注]が 1 つ生成されます。

（a）-256 ～ +255 の範囲の絶対値式

（b）本命令と同じファイル内の同じセクションに定義を持つラベルの PC オフセット参照を持ち，-256 ～ +255 の範囲の相対値式

j*cnd*　disp9	b*cnd*　disp9

注　機械語命令の b*cnd* は，ディスプレースメントに -256 ～ +255（0xffffff00 ～ 0xff）の範囲のイミーディエットをとります。

- disp22 に次のものを指定した場合，as850 では，命令展開が行われ，複数の機械語命令が生成されます。

（a）-256 ～ +255 の範囲を越え -2097150 ～ +2097153 の範囲[注 1] の絶対値式

（b）本命令と同じファイル内の同じセクションに定義を持つラベルの PC オフセット参照を持ち，-256 ～ +255 の範囲を越え -2097150 ～ +2097153 の範囲の相対値式

（c）本命令と同じファイル内に定義を持たないか同じセクションに定義を持たないラベルの PC オフセット参照を持つ相対値式

jbr　disp22	jr　disp22
jsa　disp22	bsa　Label1 br　Label2 Label1 : jr　disp22 - 4 Label2 :
j*cnd*　disp22	bn*cnd*　Label[注 2] jr　disp22 - 2 Label :

注 1.　-2097150 ～ +2097153 の範囲は，jbr，および jsa 命令以外の場合の範囲で，jbr 命令の場合は -2097152 ～ +2097151，jsa 命令の場合は -2097148 ～ +2097155 となります。

2. bn*cnd* は，たとえば，bz に対する bnz.bgt に対する ble というように，逆の条件で分岐を行う命令を示しています。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

［注意事項］

- disp22 に，-2097150 ～ +2097153 の範囲を越える絶対値式，または本命令と同じファイル内の同じセクションに定義を持つラベルの PC オフセット参照を持ち -2097150 ～ +2097153 の範囲を越える相対値式を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E3230: illegal operand (range error in displacement）
```

- disp22 に，奇数値を持つ絶対値式，または本命令と同じファイル内の同じセクションに定義を持つラベルの PC オフセット参照を持ち奇数値を持つ相対値式を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E3226: illegal operand (must be even displacement）
```

- disp22 に，本命令と同じファイル内の同じセクションに定義を持つラベルの PC オフセット参照を持つ相対値式を指定した場合，as850 は，その相対値式の値に基づいて命令展開を行うか否かを判定しますが，この相対値式の値自体，一般に命令展開の影響を受けて変動します。as850 は，この変動を考慮した処理を行うための機能を組み込んでいますが，本命令と PC オフセット参照されているラベルとの間に .align 疑似命令または .org 疑似命令を持つような場合，次のメッセージが出力され，アセンブルが中止されます。この場合，可能であれば，間にある .align 疑似命令または .org 疑似命令を外して試してみてください。

```
F3507: overflow error(9bit)
```

jarl

ジャンプ・アンド・レジスタ・リンクを行います。（Jump and Register Link）

［指定形式］

- jarl disp22, reg2
- jarl disp32, reg1 【V850E2】

disp22 に指定できるものを次に示します。

- 22 ビット幅までの値を持つ絶対値式
- ラベルの PC オフセット参照を持つ相対値式

disp32 に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- ラベルの PC オフセット参照を持つ相対値式

［機能］

- “jarl disp22, reg2”の形式

  第 1 オペランドに指定した絶対値式，または相対値式の値と，現在のプログラム・カウンタ（PC）値を加算したアドレスに制御を移します。なお，戻りアドレスは，第 2 オペランドに指定したレジスタに格納されます。

- “jarl disp32, reg1”の形式

  第 1 オペランドに指定した絶対値式，または相対値式の値と，現在のプログラム・カウンタ（PC）値を加算したアドレスに制御を移します。なお，戻りアドレスは，第 2 オペランドに指定したレジスタに格納されます。

［詳細説明］

- “jarl disp22, reg2”の形式の命令に対し，disp22 に次のものを指定した場合，as850 では，機械語命令の jarl 命令[注]が 1 つ生成されます。

  （a）-2097152 ～ +2097151 の範囲の絶対値

  （b）本命令と同じファイル内の同じセクションに定義を持つラベルの PC オフセット参照を持ち，-2097152 ～ +2097151 の範囲の相対値式

  （c）本命令と同じファイル内に定義を持たないか，同じセクションに定義を持たないラベルの PC オフセット参照を持つ相対値式

注 機械語命令の jarl は，ディスプレースメントに -2097152 ～ +2097151（0xfe00000 ～ 0x1fffff）の範囲のイミーディエトをとります。

- “jarl disp32, reg1”の形式の命令に対し，as850 では，機械語命令の jarl 命令（6 バイト長命令）が 1 つ生成されます。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

［注意事項］

- disp22 に，-2097152 ～ +2097151 の範囲を越える絶対値式，または本命令と同じファイル内の同じセクションに定義を持つラベルの PC オフセット参照を持ち -2097152 ～ +2097151 の範囲を越える相対値式を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E3230: illegal operand (range error in displacement）
```

- disp22 ／ disp32 に，奇数値を持つ絶対値式，または本命令と同じファイル内の同じセクションに定義を持つラベルの PC オフセット参照を持ち奇数値を持つ相対値式を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E3226: illegal operand (must be even displacement）
```

- アセンブラオプション -Xfar_jump を指定しない場合に，disp32 に，-2097152 ～ +2097151 の範囲を越える絶対値式，または本命令と同じファイル内の同じセクションに定義を持つラベルの PC オフセット参照を持ち -2097152 ～ +2097151 の範囲を越える相対値式を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E3230: illegal operand (range error in displacement)
```

jarl22

ジャンプ・アンド・レジスタ・リンクを行います。（Jump and Register Link）【V850E2】

［指定形式］

- jarl22 disp22, reg1

disp22 に指定できるものを次に示します。

- 22 ビット幅までの値を持つ絶対値式
- ラベルの PC オフセット参照を持つ相対値式

［機能］

第 1 オペランドに指定した絶対値式，または相対値式の値と，現在のプログラム・カウンタ（PC）値を加算したアドレスに制御を移します。なお，戻りアドレスは，第 2 オペランドに指定したレジスタに格納されます。

［詳細説明］

- disp22 に次のものを指定した場合，as850 では，機械語命令の jarl 命令[注]が 1 つ生成されます。

（a）-2097152 ～ +2097151 の範囲の絶対値

（b）本命令と同じファイル内の同じセクションに定義を持つラベルの PC オフセット参照を持ち，-2097152 ～ +2097151 の範囲の相対値式

（c）本命令と同じファイル内に定義を持たないか，同じセクションに定義を持たないラベルの PC オフセット参照を持つ相対値式

注 機械語命令の jarl は，ディスプレースメントに -2097152 ～ +2097151（0xfe00000 ～ 0x1fffff）の範囲のイミーディエトをとります。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

［注意事項］

- disp22 に，-2097152 ～ +2097151 の範囲を越える絶対値式，または本命令と同じファイル内の同じセクションに定義を持つラベルの PC オフセット参照を持ち -2097152 ～ +2097151 の範囲を越える相対値式を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E3230: illegal operand (range error in displacement)
```

- disp22 に，奇数値を持つ絶対値式，または本命令と同じファイル内の同じセクションに定義を持つラベルの PC オフセット参照を持ち奇数値を持つ相対値式を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E3226: illegal operand (must be even displacement)
```

jarl32

ジャンプ・アンド・レジスタ・リンクを行います。（Jump and Register Link）【V850E2】

［指定形式］

- jarl32 disp32, reg1

disp32 に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- ラベルの PC オフセット参照を持つ相対値式

［機能］

第 1 オペランドに指定した絶対値式，または相対値式の値と，現在のプログラム・カウンタ（PC）値を加算したアドレスに制御を移します。なお，戻りアドレスは，第 2 オペランドに指定したレジスタに格納されます。

［詳細説明］

as850 では，機械語命令の jarl 命令（6 バイト長命令）が 1 つ生成されます。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

［注意事項］

- disp32 に，奇数値を持つ絶対値式，または本命令と同じファイル内の同じセクションに定義を持つラベルの PC オフセット参照を持ち奇数値を持つ相対値式を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E3226: illegal operand (must be even displacement)
```

4. 5. 11　ビット操作命令

この項では，ビット操作命令について説明します。次に，この項において説明する命令を示します。

表 4―56　ビット操作命令

命令	意味
set1	ビット・セット
clr1	ビット・クリア
not1	ビット・ノット
tst1	ビット・テスト

set1

ビット・セットを行います。（Set Bit）

［指定形式］

- set1 bit#3, disp[reg1]
- set1 reg2, [reg1] 【V850E】
- set1 BITIO

disp に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式
- 上記のものに hi ( )，lo ( )，または hi1 ( ) を適用したもの

注意 “set1 reg2, [reg1]”の形式では disp は指定できません。

［機能］

-“set1 bit#3, disp[reg1]”の形式

第 2 オペランドで指定したアドレスが示すデータの，第 1 オペランドに指定したビットをセットします。指定したビット以外は影響を受けません。

-“set1 reg2, [reg1]”の形式

第 2 オペランドのレジスタ値で指定したアドレスが示すデータの，第 1 オペランドで指定したレジスタ値の下位 3 ビットが示すビットをセットします。指定したビット以外は影響を受けません。

-“set1 BITIO”の形式

第 1 オペランドで指定したアドレスが示すデータにおいて，周辺 I/O レジスタのビット名（デバイス・ファイルで定義されている予約語のみ）で指定したビットをセットします。

［詳細説明］

- disp に次のものを指定した場合，as850 では，機械語命令の set1 命令[注]が 1 つ生成されます。

（a）-32768 ～ +32767 の範囲の絶対値式

`set1 #bit3, disp16[reg1]`	`set1 #bit3, disp16[reg1]`

（b）sdata ／ sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

`set1 #bit3, $label[reg1]`	`set1 #bit3, $label[reg1]`

（c）!label，または %label を持つ相対値式

`set1 #bit3, !label[reg1]`	`set1 #bit3, !label[reg1]`
`set1 #bit3, %label[reg1]`	`set1 #bit3, %label[reg1]`

（d）hi ( )，lo ( )，または hi1 ( ) を適用したもの

`set1 #bit3, disp16[reg1]`	`set1 #bit3, disp16[reg1]`

注 機械語命令の set1 命令は，ディスプレースメントに -32768 ～ +32767（0xffff8000 ～ 0x7fff）の範囲のイミーディエトをとります。

- disp に次のものを指定した場合，as850 では，命令展開が行われ，複数個の機械語命令が生成されます。

（a）-32768 ～ +32767 の範囲を越える絶対値式

`set1 #bit3, disp[reg1]`	`movhi hi1(disp), reg1, r1` `set1 #bit3, lo(disp)[r1]`

（b）#label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

`set1 #bit3, #label[reg1]`	`movhi hi1(#label), reg1, r1` `set1 #bit3, lo(#label)[r1]`
`set1 #bit3, label[reg1]`	`movhi hi1(label), reg1, r1` `set1 #bit3, lo(label)[r1]`
`set1 #bit3, $label[reg1]`	`movhi hi1($label), reg1, r1` `set1 #bit3, lo($label)[r1]`

- disp を省略した場合，as850 では，0 が指定されたものとみなされます。
- disp に #label を持つ相対値式，または #label を持つ相対値式に hi ( )，lo ( )，または hi1 ( ) を適用したものを指定した場合，その後ろの [reg1] の部分が省略できます。ただし，省略した場合，as850 では，[r0] が指定されたものとみなされます。
- disp に $label を持つ相対値式，または $label を持つ相対値式に hi ( )，lo ( )，または hi1 ( ) を適用したものを指定した場合，その後ろの [reg1] の部分が省略できます。ただし，省略した場合，as850 では，[gp] が指定されたものとみなされます。
- disp にデバイス・ファイルで定義されている周辺 I/O レジスタ名を指定した場合，その後ろの [reg1] の部分が省略できます。ただし，省略した場合，as850 では，[r0] が指定されたものとみなされます。

［フラグ］

CY	－
OV	－
S	－
Z	指定したビットが 0 の場合 1，1 の場合 0 **注**
SAT	－

注　Z フラグの値は，この命令実行前の該当ビットの値を示しています。この命令実行後を示すものではありません。

clr1

ビット・クリアを行います。（Clear Bit）

［指定形式］

- clr1 bit#3, disp[reg1]
- clr1 reg2, [reg1] 【V850E】
- clr1 BITIO

disp に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式
- 上記のものに hi ( )，lo ( )，または hi1 ( ) を適用したもの

注意 “clr1 reg2, [reg1]”の形式では disp は指定できません。

［機能］

- “clr1 bit#3, disp[reg1]”の形式

  第 2 オペランドで指定したアドレスが示すデータの，第 1 オペランドに指定したビットをクリアします。指定したビット以外は影響を受けません。

- “clr1 reg2, [reg1]”の形式

  第 2 オペランドのレジスタ値で指定したアドレスが示すデータの，第 1 オペランドで指定したレジスタ値の下位 3 ビットが示すビットをクリアします。指定したビット以外は影響を受けません。

- “clr1 BITIO”の形式

  第 1 オペランドで指定したアドレスが示すデータにおいて，周辺 I/O レジスタのビット名（デバイス・ファイルで定義されている予約語のみ）で指定したビットをクリアします。

［詳細説明］

- disp に次のものを指定した場合，as850 では，機械語命令の clr1 命令[注]が 1 つ生成されます。

（a）-32768 ～ +32767 の範囲の絶対値式

`clr1 #bit3, disp16[reg1]`	`clr1 #bit3, disp16[reg1]`

（b）sdata ／ sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

`clr1 #bit3, $label[reg1]`	`clr1 #bit3, $label[reg1]`

（c）!label，または %label を持つ相対値式

clr1 #bit3, !label[reg1]	clr1 #bit3, !label[reg1]
clr1 #bit3, %label[reg1]	clr1 #bit3, %label[reg1]

（d）hi ( )，lo ( )，または hi1 ( ) を適用したもの

clr1 #bit3, disp16[reg1]	clr1 #bit3, disp16[reg1]

注 機械語命令の clr1 命令は，ディスプレースメントに -32768 ～ +32767（0xffff8000 ～ 0x7fff）の範囲のイミーディエトをとります。

- disp に次のものを指定した場合，as850 では，命令展開が行われ，複数個の機械語命令が生成されます。

（a）-32768 ～ +32767 の範囲を越える絶対値式

clr1 #bit3, disp[reg1]	movhi hi1(disp), reg1, r1 clr1 #bit3, lo(disp)[r1]

（b）#label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

clr1 #bit3, #label[reg1]	movhi hi1(#label), reg1, r1 clr1 #bit3, lo(#label)[r1]
clr1 #bit3, label[reg1]	movhi hi1(label), reg1, r1 clr1 #bit3, lo(label)[r1]
clr1 #bit3, $label[reg1]	movhi hi1($label), reg1, r1 clr1 #bit3, lo($label)[r1]

- disp を省略した場合，as850 では，0 が指定されたものとみなされます。
- disp に #label を持つ相対値式，または #label を持つ相対値式に hi ( )，lo ( )，または hi1 ( ) を適用したものを指定した場合，その後ろの [reg1] の部分を省略できます。ただし，省略した場合，as850 では，[r0] が指定されたものとみなされます。
- disp に $label を持つ相対値式，または $label を持つ相対値式に hi ( )，lo ( )，または hi1 ( ) を適用したものを指定した場合，その後ろの [reg1] の部分が省略できます。ただし，省略した場合，as850 では，[gp] が指定されたものとみなされます。
- disp にデバイス・ファイルで定義されている周辺 I/O レジスタ名を指定した場合，その後ろの [reg1] の部分が省略できます。ただし，省略した場合，as850 では，[r0] が指定されたものとみなされます。

［フラグ］

CY	－
OV	－
S	－
Z	指定したビットが 0 の場合 1，1 の場合 0 **注**
SAT	－

注　Z フラグの値は，この命令実行前の該当ビットの値を示しています。この命令実行後を示すものではありません。

not1

ビット・ノットを行います。（Not Bit）

［指定形式］

- not1 bit#3, disp[reg1]
- not1 reg2, [reg1] 【V850E】
- not1 BITIO

disp に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式
- 上記のものに hi ( )，lo ( )，または hi1 ( ) を適用したもの

注意 “not1 reg2, [reg1]”の形式では disp は指定できません。

［機能］

- “not1 bit#3, disp[reg1]”の形式

  第 2 オペランドで指定したアドレスが示すデータの，第 1 オペランドに指定したビットを反転（0→1，1→0）します。指定したビット以外は影響を受けません。

- “not1 reg2, [reg1]”の形式

  第 2 オペランドのレジスタ値で指定したアドレスが示すデータの，第 1 オペランドで指定したレジスタ値の下位 3 ビットが示すビットを反転（0→1，1→0）します。指定したビット以外は影響を受けません。

- “not1 BITIO”の形式

  第 1 オペランドで指定したアドレスが示すデータにおいて，周辺 I/O レジスタのビット名（デバイス・ファイルで定義されている予約語のみ）で指定したビットを反転（0→1，1→0）します。

［詳細説明］

- disp に次のものを指定した場合，as850 では，機械語命令の not1 命令[注]が 1 つ生成されます。

（a）-32768 ～ +32767 の範囲の絶対値式

`not1 #bit3, disp16[reg1]`	`not1 #bit3, disp16[reg1]`

（b）sdata ／ sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

`not1 #bit3, $label[reg1]`	`not1 #bit3, $label[reg1]`

（c）!label，または %label を持つ相対値式

not1 #bit3, !label[reg1]	not1 #bit3, !label[reg1]
not1 #bit3, %label[reg1]	not1 #bit3, %label[reg1]

（d）hi ( )，lo ( )，または hi1 ( ) を適用したもの

not1 #bit3, disp16[reg1]	not1 #bit3, disp16[reg1]

注 機械語命令の not1 命令は，ディスプレースメントに -32768 ～ +32767（0xffff8000 ～ 0x7fff）の範囲のイミーディエトをとります。

- disp に次のものを指定した場合，as850 では，命令展開が行われ，複数個の機械語命令が生成されます。

（a）-32768 ～ +32767 の範囲を越える絶対値式

not1 #bit3, disp[reg1]	movhi hi1(disp), reg1, r1 not1 #bit3, lo(disp)[r1]

（b）#label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

not1 #bit3, #label[reg1]	movhi hi1(#label), reg1, r1 not1 #bit3, lo(#label)[r1]
not1 #bit3, label[reg1]	movhi hi1(label), reg1, r1 not1 #bit3, lo(label)[r1]
not1 #bit3, $label[reg1]	movhi hi1($label), reg1, r1 not1 #bit3, lo($label)[r1]

- disp を省略した場合，as850 では，0 が指定されたものとみなされます。
- disp に #label を持つ相対値式，または #label を持つ相対値式に hi ( )，lo ( )，または hi1 ( ) を適用したものを指定した場合，その後ろの [reg1] の部分が省略できます。ただし，省略した場合，as850 では，[r0] が指定されたものとみなされます。
- disp に $label を持つ相対値式，または $label を持つ相対値式に hi ( )，lo ( )，または hi1 ( ) を適用したものを指定した場合，その後ろの [reg1] の部分が省略できます。ただし，省略した場合，as850 では，[gp] が指定されたものとみなされます。
- disp にデバイス・ファイルで定義されている周辺 I/O レジスタ名を指定した場合，その後ろの [reg1] の部分が省略できます。ただし，省略した場合，as850 では，[r0] が指定されたものとみなされます。

［フラグ］

CY	–
OV	–
S	–
Z	指定したビットが 0 の場合 1，1 の場合 0 **注**
SAT	–

注　Z フラグの値は，この命令実行前の該当ビットの値を示しています。この命令実行後を示すものではありません。

tst1

ビット・テストを行います。（Test Bit）

［指定形式］

- tst1 bit#3, disp[reg1]
- tst1 reg2, [reg1] 【V850E】
- tst1 BITIO

disp に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式
- 上記のものに hi ( )，lo ( )，または hi1 ( ) を適用したもの

注意 “tst1 bit#3, disp[reg1]”の形式では disp は指定できません。

［機能］

-“tst1 bit#3, disp[reg1]”の形式

第 2 オペランドで指定したアドレスが示すデータの，第 1 オペランドに指定したビットの値に従い，フラグのみを設定します。第 2 オペランドの値，および指定したビットは変更されません。

-“tst1 reg2, [reg1]”の形式

第 2 オペランドで指定したアドレスが示すデータの，第 1 オペランドで指定したレジスタ値の下位 3 ビットが示すビットの値に従い，フラグのみを設定します。第 2 オペランドの値，および指定したビットは変更されません。

-“tst1 BITIO”の形式

第 1 オペランドで指定したアドレスが示すデータにおいて，周辺 I/O レジスタのビット名（デバイス・ファイルで定義されている予約語のみ）で指定したビットの値に従い，フラグのみを設定します。周辺 I/O レジスタのビットの値は変更されません。

［詳細説明］

- disp に次のものを指定した場合，as850 では，機械語命令の tst1 命令[注]が 1 つ生成されます。

（a）-32768 ～ +32767 の範囲の絶対値式

`tst1 #bit3, disp16[reg1]`	`tst1 #bit3, disp16[reg1]`

（b）sdata ／ sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

`tst1 #bit3, $label[reg1]`	`tst1 #bit3, $label[reg1]`

（c）!label，または %label を持つ相対値式

tst1 #bit3, !label[reg1]	tst1 #bit3, !label[reg1]
tst1 #bit3, %label[reg1]	tst1 #bit3, %label[reg1]

（d）hi ( )，lo ( )，または hi1 ( ) を適用したもの

tst1 #bit3, disp16[reg1]	tst1 #bit3, disp16[reg1]

注 機械語命令の tst1 命令は，ディスプレースメントに -32768 ～ +32767（0xffff8000 ～ 0x7fff）の範囲のイミーディエトをとります。

- disp に次のものを指定した場合，as850 では，命令展開が行われ，複数個の機械語命令が生成されます。

（a）-32768 ～ +32767 の範囲を越える絶対値式

tst1 #bit3, disp[reg1]	movhi hi1(disp), reg1, r1 tst1 #bit3, lo(disp)[r1]

（b）#label，または label を持つ相対値式，および sdata ／ sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

tst1 #bit3, #label[reg1]	movhi hi1(#label), reg1, r1 tst1 #bit3, lo(#label)[r1]
tst1 #bit3, label[reg1]	movhi hi1(label), reg1, r1 tst1 #bit3, lo(label)[r1]
tst1 #bit3, $label[reg1]	movhi hi1($label), reg1, r1 tst1 #bit3, lo($label)[r1]

- disp を省略した場合，as850 では，0 が指定されたものとみなされます。
- disp に #label を持つ相対値式，または #label を持つ相対値式に hi ( )，lo ( )，または hi1 ( ) を適用したものを指定した場合，その後ろの [reg1] の部分が省略できます。ただし，省略した場合，as850 では，[r0] が指定されたものとみなされます。
- disp に $label を持つ相対値式，または $label を持つ相対値式に hi ( )，lo ( )，または hi1 ( ) を適用したものを指定した場合，その後ろの [reg1] の部分が省略できます。ただし，省略した場合，as850 では，[gp] が指定されたものとみなされます。
- disp にデバイス・ファイルで定義されている周辺 I/O レジスタ名を指定した場合，その後ろの [reg1] の部分が省略できます。ただし，省略した場合，as850 では，[r0] が指定されたものとみなされます。

［フラグ］

CY	–
OV	–
S	–
Z	指定したビットが 0 の場合 1，1 の場合 0
SAT	–

4. 5. 12　スタック操作命令

この項では，スタック操作命令について説明します。次に，この項において説明する命令を示します。

表 4―57　スタック操作命令

命令	意味
push	スタック領域へのプッシュ（単一レジスタ）
pushm	スタック領域へのプッシュ（複数レジスタ）
pop	スタック領域からのポップ（単一レジスタ）
popm	スタック領域からのポップ（複数レジスタ）

push

スタック領域へのプッシュを行います。（Push）

［指定形式］

push reg

［機能］

オペランドに指定したレジスタ値を，スタック領域にプッシュします。

［詳細説明］

- push 命令に対し，as850 では，命令展開が行われ，複数個の機械語命令が生成されます。

`push reg`	`add -4, sp` `st.w reg, [sp]`

［フラグ］

CY	MSB（Most Significant Bit）からのキャリーを生じた場合 1，そうでない場合 0
OV	Integer-Overflow を生じた場合 1，そうでない場合 0
S	結果が負になった場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	−

注意 命令展開が行われ，add 命令により設定されます。

pushm

スタック領域へのプッシュ（複数レジスタ）を行います。（Push Multiple）

［指定形式］

pushm reg1, reg2, …, regN

［機能］

オペランドに指定したレジスタ値を，スタック領域へプッシュします。なお，オペランドに指定可能なレジスタ数は，最大 32 個です。

［詳細説明］

- pushm 命令に対し，as850 では，命令展開が行われ，複数個の機械語命令が生成されます。

レジスタが 4 個以下の場合

pushm reg1, reg2, …, regN	add -4 * N, sp st.w regN, 4 * (N - 1)[sp] : st.w reg2, 4 * 1[sp] st.w reg1, 4 * 0[sp]

レジスタが 5 個以上の場合

pushm reg1, reg2, …, regN	addi -4 * N, sp, sp st.w regN, 4 * (N - 1)[sp] : st.w reg2, 4 * 1[sp] st.w reg1, 4 * 0[sp]

［フラグ］

CY	MSB（Most Significant Bit）からのキャリーを生じた場合 1，そうでない場合 0
OV	Integer-Overflow を生じた場合 1，そうでない場合 0
S	結果が負になった場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	—

注意 命令展開が行われ，add ／ addi 命令により設定されます。

pop

スタック領域からのポップを行います。（Pop）

［指定形式］

pop reg

［機能］

オペランドに指定したレジスタ値を，スタック領域からポップします。

［詳細説明］

- pop 命令に対し，as850 では，命令展開が行われ，複数個の機械語命令が生成されます。

`pop reg`	`ld.w [sp], reg` `add 4, sp`

［フラグ］

CY	MSB（Most Significant Bit）からのキャリーを生じた場合 1，そうでない場合 0
OV	Integer-Overflow を生じた場合 1，そうでない場合 0
S	結果が負になった場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	−

注意 命令展開が行われ，add 命令により設定されます。

popm

スタック領域からのポップ（複数レジスタ）を行います。（Pop Multiple）

［指定形式］

popm　reg1, reg2, …, regN

［機能］

オペランドに指定したレジスタ値を，指定した順にスタック領域からポップします。なお，オペランドに指定可能なレジスタ数は，最大 32 個です。

［詳細説明］

- popm 命令に対し，as850 では，命令展開が行われ，複数個の機械語命令が生成されます。

レジスタが 3 個以下の場合

popm　reg1, …, regN	ld.w　4 * 0[sp], reg1 : ld.w　4 * (N - 1)[sp], regN add　4 * N, sp

レジスタが 4 個以上の場合

popm　reg1, reg2, …, regN	ld.w　4 * 0[sp], reg1 ld.w　4 * 1[sp], reg2 : ld.w　4 * (N - 1)[sp], regN addi　4 * N, sp, sp

［フラグ］

CY	MSB（Most Significant Bit）からのキャリーを生じた場合 1，そうでない場合 0
OV	Integer-Overflow を生じた場合 1，そうでない場合 0
S	結果が負になった場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	−

注意　命令展開が行われ，add ／ addi 命令により設定されます。

4. 5. 13　特殊命令

この項では，特殊命令について説明します。次に，この項において説明する命令を示します。

表 4―58　特殊命令

命令	意味
ldsr	システム・レジスタへのロード
stsr	システム・レジスタの内容のストア
di	マスカブル割り込みの禁止
ei	マスカブル割り込みの許可
reti	トラップ，または割り込みルーチンからの復帰
halt	プロセッサの停止
trap	ソフトウエア・トラップ
nop	ノー・オペレーション
switch	テーブル参照分岐【V850E】
callt	テーブル参照コール【V850E】
ctret	callt からの復帰【V850E】
dbtrap	デバッグ・トラップ【V850E】
dbret	デバッグ・トラップからの復帰【V850E】
prepare	スタック・フレームの生成（関数の前処理）【V850E】
dispose	スタック・フレームの削除（関数の後処理）【V850E】

ldsr

システム・レジスタへのロードを行います。（Load System Register）

［指定形式］

- ldsr reg, regID

regID に指定できるものを次に示します。

- 5 ビット幅までの値を持つ絶対値式

［機能］

第 1 オペランドに指定したレジスタ値を，第 2 オペランドに指定したシステム・レジスタ番号で示されるシステム・レジスタ注に格納します。

注 システム・レジスタに関しては，各デバイスのユーザーズ・マニュアル，および次表を参照してください。

表 4―59 システム・レジスタ番号（ldsr）

番号	システム・レジスタ	
0	割り込み時状態退避レジスタ	EIPC
1	割り込み時状態退避レジスタ	EIPSW
2	NMI 時状態退避レジスタ	FEPC
3	NMI 時状態退避レジスタ	FEPSW
4	割り込み要因レジスタ注	ECR
5	プログラム・ステータス・ワード	PSW
6-31	予約	―

注 割り込み要因レジスタは，オペランド指定できません。アクセス禁止です。

表 4―60 システム・レジスタ番号【V850E/MA1】（ldsr）

番号	システム・レジスタ	
0	割り込み時状態退避レジスタ	EIPC
1	割り込み時状態退避レジスタ	EIPSW
2	NMI 時状態退避レジスタ	FEPC
3	NMI 時状態退避レジスタ	FEPSW
4	割り込み要因レジスタ注	ECR
5	プログラム・ステータス・ワード	PSW
6-15	予約	―

番号	システム・レジスタ	
16	テーブル参照コール時状態退避レジスタ	CTPC
17	テーブル参照コール時状態退避レジスタ	CTPSW
18	不正命令例外時状態退避レジスタ	DBPC
19	不正命令例外時状態退避レジスタ	DBPSW
20	CALLT 用ベース・ポインタ	CTBP
21-31	予約	−

注 割り込み要因レジスタは，オペランド指定できません。アクセス禁止です。

表 4−61 システム・レジスタ番号【V850E/ME2】（ldsr）

番号	システム・レジスタ	
0	割り込み時状態退避レジスタ	EIPC
1	割り込み時状態退避レジスタ	EIPSW
2	NMI 時状態退避レジスタ	FEPC
3	NMI 時状態退避レジスタ	FEPSW
4	割り込み要因レジスタ**注 1**	ECR
5	プログラム・ステータス・ワード	PSW
6-15	予約	−
16	テーブル参照コール時状態退避レジスタ	CTPC
17	テーブル参照コール時状態退避レジスタ	CTPSW
18	不正命令例外時状態退避レジスタ	DBPC
19	不正命令例外時状態退避レジスタ	DBPSW
20	CALLT 用ベース・ポインタ	CTBP
21	デバッグ・インタフェース・レジスタ**注 2**	DIR
22	ブレークポイント制御レジスタ 0，1**注 2，注 3**	BPC0，BPC1
23	プログラム ID レジスタ	ASID
24	ブレークポイント・アドレス設定レジスタ 0，1**注 2，注 3**	BPAV0，BPAV1
25	ブレークポイント・アドレス・マスク・レジスタ**注 2，注 3**	BPAM0，BPAM1
26	ブレークポイント・データ設定レジスタ 0，1**注 2，注 3**	BPDV0，BPDV1
27	ブレークポイント・データ・マスク・レジスタ 0，1**注 2，注 3**	BPDM0，BPDM1
28-31	予約	−

注 1. 割り込み要因レジスタは，オペランド指定できません。アクセス禁止です。

2. デバッグ・モード時だけアクセス可能です。

3. 実際にアクセスされるレジスタは，DIR レジスタの CS ビットによって指定されます。

[フラグ]

CY	–
OV	–
S	–
Z	–
SAT	–

注意 システム・レジスタにプログラム・ステータス・ワード（PSW）を指定した場合，各フラグには reg に対応したビットを設定します。

[注意事項]

- ldsr 命令により EIPC か FEPC，DBPC，または CTPC のビット 0 をセット“1”したあと，reti 命令，ctret 命令，または dbret 命令で復帰する際に，ビット 0 の値は無視されます（PC のビット 0 が 0 固定のため）。EIPC，FEPC，DBPC，CTPC に値を設定する場合は，偶数値（ビット 0 = 0）を設定してください。
- regID に 0 ～ 31 の範囲を越える絶対値式を指定した場合，次のメッセージが出力され，指定した値を下位 5 ビット[注]を用いてアセンブルが続行されます。

```
W3011: illegal operand (range error in immediate）
```

注 機械語命令の ldsr 命令は，第 2 オペランドに 0 ～ 31（0x0 ～ 0x1f）の範囲のイミーディエトをとります。

- regID に予約レジスタ番号やアクセス禁止のレジスタ番号（ECR など）を指定した場合，またはデバッグ・モード時だけアクセス可能のレジスタ番号を指定した場合，次のメッセージが出力され，そのままアセンブルが続行されます。

```
W3018: illegal regID for ldsr
```

stsr

システム・レジスタの内容のストアを行います。（Store System Register）

［指定形式］

- stsr regID, reg

regID に指定できるものを次に示します。

- 5 ビット幅までの値を持つ絶対値式

［機能］

第 1 オペランドに指定したシステム・レジスタ番号で示されるシステム・レジスタ注の値を，第 2 オペランドに指定したレジスタに格納します。

注 システム・レジスタに関しては，各デバイスのユーザーズ・マニュアル，および次表を参照してください。

表 4―62 システム・レジスタ番号（stsr）

番号	システム・レジスタ	
0	割り込み時状態退避レジスタ	EIPC
1	割り込み時状態退避レジスタ	EIPSW
2	NMI 時状態退避レジスタ	FEPC
3	NMI 時状態退避レジスタ	FEPSW
4	割り込み要因レジスタ	ECR
5	プログラム・ステータス・ワード	PSW
6-31	予約	―

表 4―63 システム・レジスタ番号【V850E/MA1】（stsr）

番号	システム・レジスタ	
0	割り込み時状態退避レジスタ	EIPC
1	割り込み時状態退避レジスタ	EIPSW
2	NMI 時状態退避レジスタ	FEPC
3	NMI 時状態退避レジスタ	FEPSW
4	割り込み要因レジスタ	ECR
5	プログラム・ステータス・ワード	PSW
6-15	予約	―
16	テーブル参照コール時状態退避レジスタ	CTPC
17	テーブル参照コール時状態退避レジスタ	CTPSW

番号	システム・レジスタ	
18	不正命令例外時状態退避レジスタ	DBPC
19	不正命令例外時状態退避レジスタ	DBPSW
20	CALLT 用ベース・ポインタ	CTBP
21-31	予約	－

表 4―64　システム・レジスタ番号【V850E/ME2】（stsr）

番号	システム・レジスタ	
0	割り込み時状態退避レジスタ	EIPC
1	割り込み時状態退避レジスタ	EIPSW
2	NMI 時状態退避レジスタ	FEPC
3	NMI 時状態退避レジスタ	FEPSW
4	割り込み要因レジスタ	ECR
5	プログラム・ステータス・ワード	PSW
6-15	予約	－
16	テーブル参照コール時状態退避レジスタ	CTPC
17	テーブル参照コール時状態退避レジスタ	CTPSW
18	不正命令例外時状態退避レジスタ	DBPC
19	不正命令例外時状態退避レジスタ	DBPSW
20	CALLT 用ベース・ポインタ	CTBP
21	デバッグ・インタフェース・レジスタ	DIR
22	ブレークポイント制御レジスタ 0，1	BPC0，BPC1
23	プログラム ID レジスタ	ASID
24	ブレークポイント・アドレス設定レジスタ 0，1	BPAV0，BPAV1
25	ブレークポイント・アドレス・マスク・レジスタ	BPAM0，BPAM1
26	ブレークポイント・データ設定レジスタ 0，1	BPDV0，BPDV1
27	ブレークポイント・データ・マスク・レジスタ 0，1	BPDM0，BPDM1
28-31	予約	－

注 1.　デバッグ・モード時だけアクセス可能です。

2.　実際にアクセスされるレジスタは，DIR レジスタの CS ビットによって指定されます。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

［注意事項］

- regID に 0 ～ 31 の範囲を越える絶対値式を指定した場合，次のメッセージが出力され，指定した値の下位 5 ビット[注]を用いてアセンブルが続行されます。

```
W3011: illegal operand (range error in immediate）
```

注　機械語命令の stsr 命令は，第 1 オペランドに 0 ～ 31（0x0 ～ 0x1f）の範囲のイミーディエトをとります。

- regID に予約レジスタ番号を指定した場合，またはデバッグ・モード時だけアクセス可能のレジスタ番号を指定した場合，次のメッセージが出力され，そのままアセンブルが続行されます。

```
W3018: illegal regID for ldsr
```

di

マスカブル割り込みの禁止を行います。（Disable Interrupt）

［指定形式］

di

［機能］

PSW 中の ID ビットに 1 を設定し，本命令実行中からマスカブル割り込みの受け付けを禁止します。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–
ID	1

ei

マスカブル割り込みの許可を行います。（Enable Interrupt）

［指定形式］

ei

［機能］

PSW 中の ID ビットに 0 を設定し，次の命令よりマスカブル割り込みの受け付けを許可します。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–
ID	0

reti

トラップ，または割り込みルーチンからの復帰を行います。（Return from Trap or Interrupt）

［指定形式］

reti

［機能］

トラップ，または割り込みルーチンから復帰します。**注**

注 機能の詳細に関しては，各デバイスのユーザーズ・マニュアルを参照してください。

［フラグ］

CY	取り出した値
OV	取り出した値
S	取り出した値
Z	取り出した値
SAT	取り出した値

halt

停止します。（Halt）

［指定形式］

halt

［機能］

プロセッサを停止し，HALT 状態に遷移します。なお，HALT 状態からの処理再開は，マスカブル割り込み，NMI，リセットにより行われます。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

trap

ソフトウエア・トラップを発生させます。（Trap）

［指定形式］

- trap vector

vector に指定できるものを次に示します。

- 5 ビット幅までの値を持つ絶対値式

［機能］

ソフトウエア・トラップを発生させます。注

注 機能の詳細に関しては，各デバイスのユーザーズ・マニュアルを参照してください。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

［注意事項］

- vector に 0 ～ 31 の範囲を越える絶対値式を指定した場合，次のメッセージが出力され，指定した値の下位 5 ビット注を用いてアセンブルが続行されます。

```
W3011: illegal operand (range error in immediate）
```

注 機械語命令の trap 命令は，オペランドに 0 ～ 31（0x0 ～ 0x1f）の範囲のイミーディエトをとります。

nop

ノー・オペレーションです。（No Operation）

［指定形式］

nop

［機能］

何も行いません。命令シーケンス内に領域を確保したり，命令実行に遅延を挿入したりするために用いることができます。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

switch

テーブル参照分岐を行います。（Jump With Table Look Up）【V850E】

［指定形式］

switch reg

［機能］

次の順に処理を行います。

（1）オペランドで指定した値を 1 ビット論理左シフトした値とテーブルの先頭アドレス（switch 命令の次のアドレス）とを加算し，テーブル・エントリ・アドレスを生成します。

（2）生成したテーブル・エントリ・アドレスから符号付きハーフワード・データをロードします。

（3）ロードした値を 1 ビット論理左シフトし，ワード長に符号拡張したあと，テーブルの先頭アドレスを加算し，アドレスを生成します。

（4）生成したアドレスへ分岐します。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

［注意事項］

- reg に r0 を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E3240: illegal operand (can not use r0 as source in V850E mode)
```

callt

テーブル参照コールを行います。（Call With Table Look Up）【V850E】

［指定形式］

- callt imm6

imm6 に指定できるものを次に示します。

- 6 ビット幅までの値を持つ絶対値式

［機能］

次の順に処理を行います。**注**

（1）復帰 PC と PSW の値を CTPC と CTPSW に退避します。

（2）オペランドで指定された値を 1 ビット左シフトして CTBP（CALLT Base Pointer）からのオフセット値とし，CTBP 値と加算してテーブル・エントリ・アドレスを生成します。

（3）生成したテーブル・エントリ・アドレスから符号なしハーフワード・データをロードします。

（4）ロードした値と CTBP 値を加算してアドレスを生成します。

（5）生成したアドレスへ分岐します。

注 システム・レジスタについては，各デバイスのユーザーズ・マニュアルを参照してください。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

ctret

callt から復帰します。（Return From Callt）【V850E】

［指定形式］

ctret

［機能］

callt による分岐から復帰します。次の順に処理を行います。**注**

（1）復帰 PC と PSW を，CTPC と CTPSW から取り出します。

（2）取り出した値を PC と PSW に設定し，制御を移します。

注　システム・レジスタについては，各デバイスのユーザーズ・マニュアルを参照してください。

［フラグ］

CY	取り出した値
OV	取り出した値
S	取り出した値
Z	取り出した値
SAT	取り出した値

dbtrap

デバッグ・トラップを起こします。（Debug Trap）【V850E】

［指定形式］

dbtrap

［機能］

デバッグ・トラップを起こします。注

注 機能の詳細に関しては，各デバイスのユーザーズ・マニュアルを参照してください。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

dbret

デバッグ・トラップから復帰します。（Return From Debug Trap）【V850E】

［指定形式］

dbret

［機能］

デバッグ・トラップから復帰します。**注**

注　機能の詳細に関しては，各デバイスのユーザーズ・マニュアルを参照してください。

［フラグ］

CY	取り出した値
OV	取り出した値
S	取り出した値
Z	取り出した値
SAT	取り出した値

prepare

スタック・フレームの生成（関数の前処理）を行います。（Function Prepare）【V850E】

［指定形式］

- prepare list, imm1
- prepare list, imm1, imm2
- prepare list, imm1, sp

imm1 ／ imm2 に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式

list は，prepare 命令でプッシュ可能な 12 本のレジスタを指定するものです。list に指定できるものを次に示します。

- レジスタ

  プッシュの対象となるレジスタ（r20 ～ r31）をカンマで区切って指定します。

- 12 ビット幅までの値を持つ定数式

  12 ビットと 12 本のレジスタとの対応は次のとおりです。

ビット 11 ビット 0

r30	r24	r25	r26	r27	r20	r21	r22	r23	r28	r29	r31

次の 2 つの指定は等価です。

`prepare r26, r29, r31, 0x10`	`prepare 0x103, 0x10`

［機能］

prepare 命令は，関数の前処理をする命令です。

- “prepare list, imm1”の形式

（a）第 1 オペランドで指定したレジスタを 1 つプッシュし，スタック・ポインタ（sp）から 4 を減算します。

（b）第 1 オペランドで指定したレジスタをすべてプッシュし終わるまで（a）を繰り返します。

（c）第 2 オペランドで指定した絶対値式の値を sp から減算[注]し，sp をレジスタ退避領域に設定します。

- “prepare list, imm1, imm2”の形式

（a）第 1 オペランドで指定したレジスタを 1 つプッシュし，sp から 4 を減算します。

（b）第 1 オペランドで指定したレジスタをすべてプッシュし終わるまで（a）を繰り返します。

（c）第 2 オペランドで指定した絶対値式の値を sp から減算[注]し，sp をレジスタ退避領域に設定します。

（d）第 3 オペランドで指定した絶対値式の値を ep に設定します。

- “prepare list, imm1, sp”の形式

（a）第 1 オペランドで指定したレジスタを 1 つプッシュし，sp から 4 を減算します。

（b）第 1 オペランドで指定したレジスタをすべてプッシュし終わるまで（a）を繰り返します。

（c）第 2 オペランドで指定した絶対値式の値を sp から減算[注]し，sp をレジスタ退避領域に設定します。

（d）第 3 オペランドで指定した sp の値を ep に設定します。

注 機械語命令で実際に sp に加算される値は，imm1 を左へ 2 ビット・シフトした値となります。このためアセンブラは，指定した imm1 をあらかじめ右へ 2 ビット・シフトしてコードに反映します。

[詳細説明]

- imm1 に次のものを指定した場合，as850 では，機械語命令の prepare 命令が 1 つ生成されます。

（a）0 ～ 127 の範囲の絶対値式

`prepare list, imm1`	`prepare list, imm1`
`prepare list, imm1, imm2`	`prepare list, imm1, imm2`
`prepare list, imm1, sp`	`prepare list, imm1, sp`

- list に定数式以外[注]を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E3249: illegal syntax
```

注 未定義シンボルやラベルの参照です。

- imm1 に次のものを指定した場合，as850 では，命令展開が行われ，複数個の機械語命令が生成されます。

（a）0 ～ 127 の範囲を越え，0 ～ 32767 の範囲の絶対値式

`prepare list, imm1`	`prepare list, 0` `movea -imm1, sp, sp`
`prepare list, imm1, imm2`	`prepare list, 0, imm2` `movea -imm1, sp, sp`
`prepare list, imm1, sp`	`prepare list, 0, sp` `movea -imm1, sp, sp`

（b）0 ～ 32767 の範囲を越える絶対値式

`prepare list, imm1`	`prepare list, 0`
	`mov imm1, r1`
	`sub r1, sp`

`prepare list, imm1, imm2`	`prepare list, 0, imm2`
	`mov imm1, r1`
	`sub r1, sp`

`prepare list, imm1, sp`	`prepare list, 0, sp`
	`mov imm1, r1`
	`sub r1, sp`

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

注意　命令展開により sub 命令が生じた場合，フラグ値は変化する可能性があります。

［注意事項］

- sp で指定された下位 2 ビットのアドレスは，ミス・アライン・アクセスがイネーブルであっても 0 にマスクされます。そのため sp の値は 4 バイト・アライメントした値を設定してください。

dispose

スタック・フレームの削除（関数の後処理）を行います。（Function Dispose）【V850E】

［指定形式］

- dispose imm, list
- dispose imm, list, [reg]

imm に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式

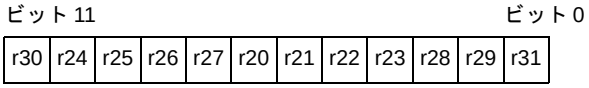
list は，dispose 命令でポップ可能な 12 本のレジスタを指定するものです。list に指定できるものを，次に示します。

- レジスタ

  プッシュの対象となるレジスタ（r20 ～ r31）をカンマで区切って指定します。

- 12 ビット幅までの値を持つ定数式

  12 ビットと 12 本のレジスタとの対応は次のとおりです。



次の 2 つの指定は等価です。

`dispose 0x10, r26, r29, r31`	`dispose 0x10, 0x103`

［機能］

dispose 命令は，関数の後処理をする命令です。

- “dispose imm, list”の形式

（a）第 1 オペランドで指定した絶対値式の値をスタック・ポインタ（sp）に加算[注]し，sp をレジスタ退避領域に設定します。

（b）第 2 オペランドで指定したレジスタを 1 つポップし，sp に 4 を加算します。

（c）第 2 オペランドで指定したレジスタをすべてポップし終わるまで（b）を繰り返します。

注　機械語命令で実際に sp に加算される値は，imm を左へ 2 ビット・シフトした値となります。したがって，アセンブラは，指定した imm をあらかじめ右へ 2 ビット・シフトしてコードに反映します。

- “dispose imm, list, [reg]”の形式

（a）第 1 オペランドで指定した絶対値式の値をスタック・ポインタ（sp）に加算[注]し，sp をレジスタ退避領域に設定します。

（b）第 2 オペランドで指定したレジスタを 1 つポップし，sp に 4 を加算します。

（c）第 2 オペランドで指定したレジスタをすべてポップし終わるまで（b）を繰り返します。

（d）第 3 オペランドで指定したレジスタ値をプログラム・カウンタ（PC）に設定します。

注 未定義シンボルやラベルの参照です。

[詳細説明]

- imm に次のものを指定した場合，as850 では，機械語命令の dispose 命令を 1 つ生成します。

（a）0 ～ 127 の範囲の絶対値式

`dispose imm, list`	`dispose imm, list`
`dispose imm, list, [reg]`	`dispose imm, list, [reg]`

- list に定数式以外を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E3249: illegal syntax
```

- imm に次のものを指定した場合，as850 では，命令展開が行われ，複数個の機械語命令が生成されます。

（a）0 ～ 127 の範囲を越え，0 ～ 32767 の範囲の絶対値式

`dispose imm, list`	`mov imm, r1`
	`add r1, sp`
	`dispose 0, list`

`dispose imm, list, [reg]`	`movea imm, sp, sp` `dispose 0, list, [reg]`

（b）0 ～ 32767 の範囲を越える絶対値式

`dispose imm, list`	`mov imm, r1`
	`add r1, sp`
	`dispose 0, list`

dispose imm, list, [reg]	mov imm, r1
	add r1, sp
	dispose 0, list, [reg]

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

注意 命令展開により add 命令が生じた場合，フラグ値は変化する可能性があります。

［注意事項］

- sp で指定された下位 2 ビットのアドレスは，ミス・アライン・アクセスがイネーブルであっても 0 にマスクされます。そのため sp の値は 4 バイト・アライメントした値を設定してください。
- 形式“dispose imm, list, [reg]”の [reg] に r0 を指定すると，次のメッセージが出力され，アセンブルが中止されます。

E3240: illegal operand (can not use r0 as destination in V850E mode)

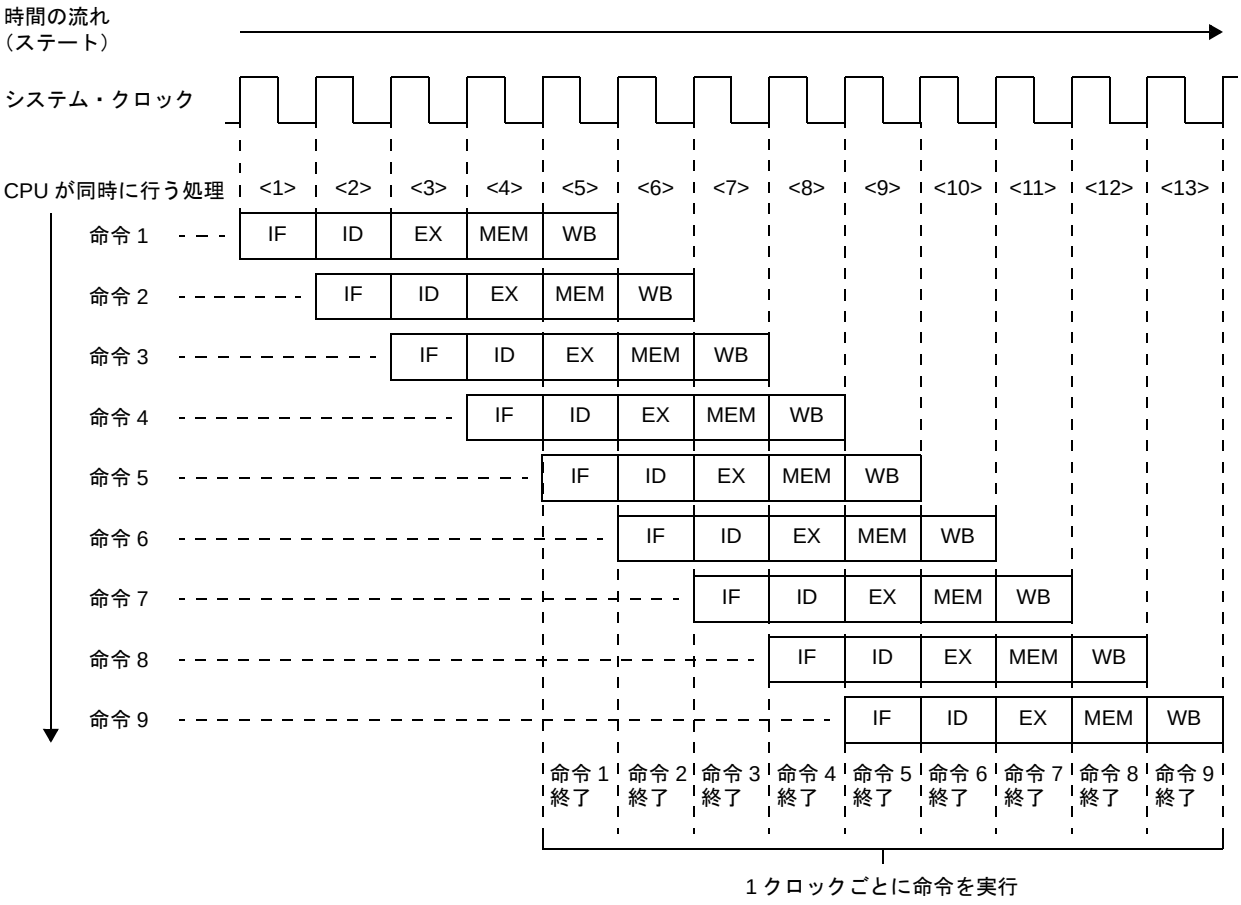
4. 5. 14　パイプライン（V850）

V850 は，RISC アーキテクチャをベースとした 5 段パイプライン制御により，ほとんどの命令を 1 クロックで実行します。命令実行手順は，通常，IF（インストラクション・フェッチ）から WB（ライトバック）までの 5 ステージで構成されています。

IF（インストラクション・フェッチ）	命令をフェッチし，フェッチ・ポインタをインクリメント
ID（インストラクション・デコード）	命令をデコードし，イミーディエト・データの作成，レジスタの読み出しを行う
EX（実行）	デコードした命令を実行
MEM（メモリ・アクセス）	対象アドレスのメモリをアクセス
WB（ライトバック）	レジスタに実行結果を書き込む

各ステージの実行時間は，命令の種類，アクセス対象となるメモリの種類などによって異なります。
パイプラインの動作例として，標準的な命令を 9 個続けて実行した際の CPU の処理を下図に示します。

図 4―64　標準的な命令を 9 個続けて実行する例



<1> ～ <13> は CPU のステートを示します。各ステートでは，命令 n の WB（ライトバック），命令 n + 1 の MEM（メモリ・アクセス），命令 n + 2 の EX（実行），命令 n + 3 の ID（インストラクション・デコード），命令 n + 4 の IF（インストラクション・フェッチ）が同時に行われます。標準的な命令では，IF ステージから WB ステージまで 5 クロックかかります。しかし，同時に 5 命令を処理できるため，標準的な命令では平均 1 クロックごとに実行可能です。

(1) パイプラインの乱れ

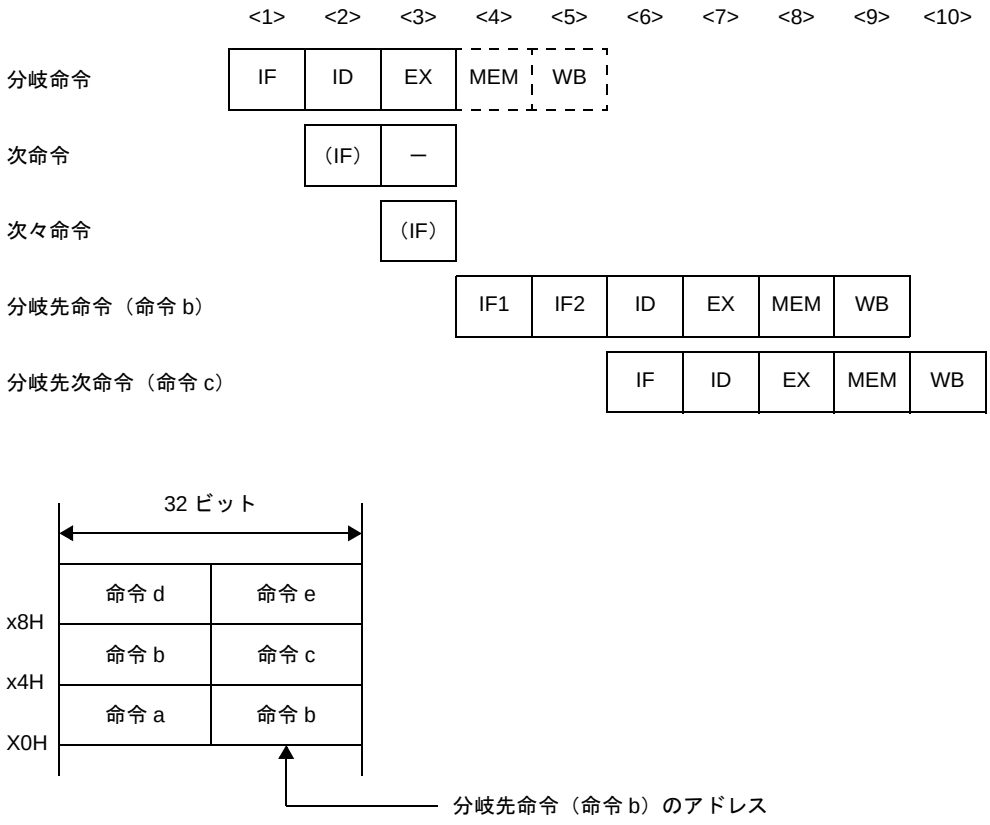
パイプラインは IF（インストラクション・フェッチ）から WB（ライトバック）までの 5 ステージで構成され，基本的にはそれぞれのステージは 1 クロックで処理されますが，場合によってはパイプラインが乱れて，実行クロック数が増加する場合があります。以下に，パイプラインを乱す主な要因を示します。

(a) アライン・ハザード

分岐先命令のアドレスがワード・アラインではなく（A1 = 1，A0 = 0），かつ 4 バイト長命令の場合，命令をワード単位に揃えるために IF を 2 回続ける必要があります。これをアライン・ハザードと呼びます。

たとえば，命令 a から命令 e までがアドレス X0H から配置されており，命令 b は 4 バイト長命令で，その他の命令は 2 バイト長命令であるとします。この場合，命令 b は X2H に配置され（A1 = 1，A0 = 0），ワード・アライン（A1 = 0，A0 = 0）となっていません。したがって，この命令 b が分岐先命令となる場合，アライン・ハザードが発生します。アライン・ハザードが発生した場合の分岐命令の実行クロック数は，4 となります。

図 4―65　アライン・ハザードの例



備考 （IF）：無効となる命令フェッチ

ー：待ち合わせのために挿入されるアイドル

IF1：アライン・ハザード時に発生する 1 回目の命令フェッチです。2 バイト長のフェッチで，命令 b の下位アドレス側の 2 バイトがフェッチされます。

IF2：アライン・ハザード時に発生する 2 回目の命令フェッチです。通常の 4 バイト長のフェッチで，命令 b の上位アドレス側の 2 バイトと命令 c（2 バイト長）がフェッチされます。

アライン・ハザードは，次のような対処によって回避が可能で，命令実行速度の向上が図れます。

- 分岐先命令に 2 バイト長命令を使用する
- 分岐先命令にワード境界（A1 = 0，A0 = 0）に配置した 4 バイト長命令を使用する

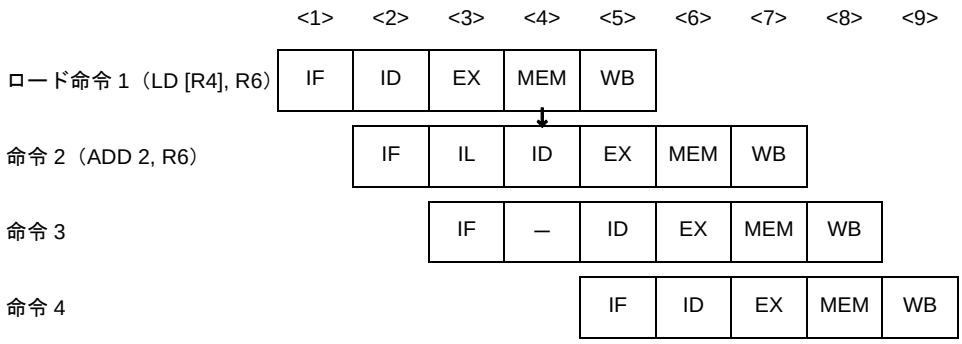
（b）ロード命令実行結果の参照

ロード命令（LD，SLD）では，MEM ステージで読み出されたデータの格納が WB ステージで行われます。したがって，ロード命令の直後の命令で同一のレジスタの内容を使用する場合，ロード命令がレジスタの使用を終えるまで，直後の命令はレジスタの使用を遅らせる必要があります。これをハザードと呼びます。V850 マイクロコントローラは，このハザードを CPU で自動的に対処するインタロック機能を持っており，次命令の ID ステージを遅らせます。

また，V850 マイクロコントローラは，MEM ステージで読み出したデータを次命令の ID ステージで使用できるように，ショート・パスを持っています。このショート・パスによって，ロード命令によって MEM ステージでデータを読み出すことと，このデータを次命令の ID ステージで使用することを，同一タイミングで行うことができます。

以上のことにより，結果を直後の命令で使用する場合，ロード命令の実行クロック数は，2 になります。

図 4―66 ロード命令実行結果の参照例



備考 IL：インタロック機能によりデータの待ち合わせのために挿入されるアイドル

ー：待ち合わせのために挿入されるアイドル

↓：ショート・パス

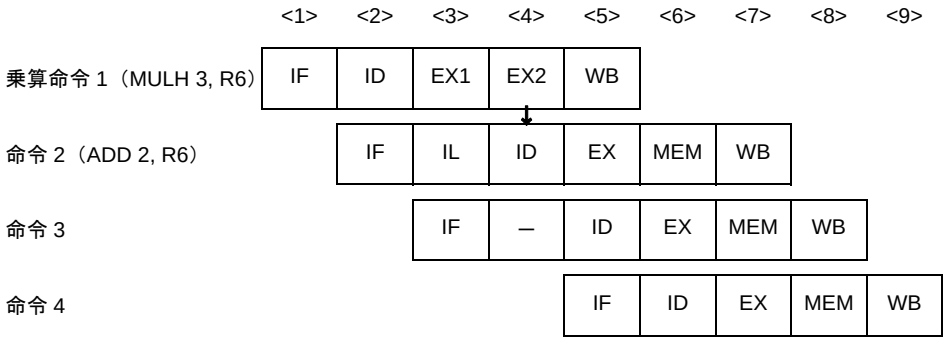
上図のように，ロード命令の直後にその結果を使用する命令を配置すると，インタロック機能によるデータの待ち合わせ時間が発生し，実行速度が低下します。ロード命令の結果を使用する命令は，ロード命令の 2 命令以後に配置することにより，実行速度の低下を防げます。

(c) 乗算命令実行結果の参照

乗算命令（MULH，MULHI）では，乗算結果のレジスタへの格納が WB ステージで行われます。したがって，乗算命令の直後の命令で同一レジスタの内容を使用する場合，乗算命令がレジスタの使用を終えるまで，直後の命令はレジスタの使用を遅らせる必要があります（ハザードの発生）。

V850 マイクロコントローラではインタロック機能により直後の命令の ID ステージを遅らせます。また，ショート・パスにより，乗算命令の EX2 ステージと，この演算結果を直後の命令の ID ステージで使用することが，同一タイミングで行えます。

図 4―67 乗算命令実行結果の参照例



備考 IL：インタロック機能によりデータの待ち合わせのために挿入されるアイドル
―：待ち合わせのために挿入されるアイドル
↓：ショート・パス

上図のように，乗算命令の直後にその結果を使用する命令を配置すると，インタロック機能によるデータの待ち合わせ時間が発生し，実行速度が低下します。乗算命令の結果を使用する命令は，乗算命令の 2 命令以後に配置することにより，実行速度の低下を防げます。

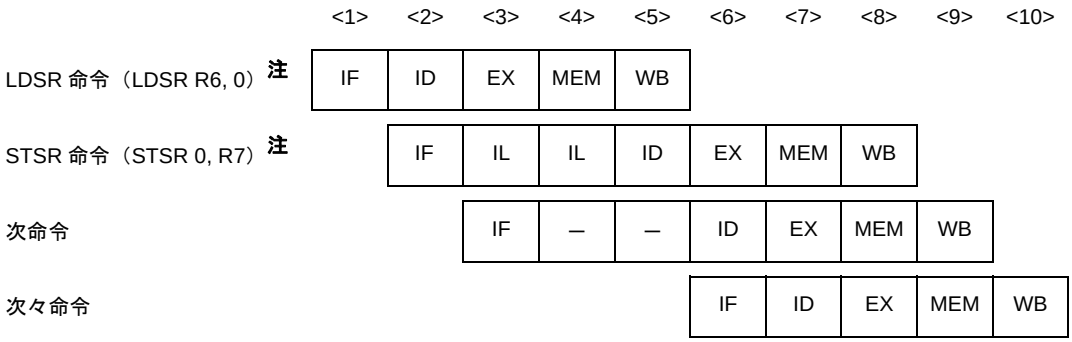
(d) EIPC，FEPC を対象とする LDSR 命令実行結果の参照

LDSR 命令によって，システム・レジスタの EIPC，FEPC のデータ設定を行い，直後に STSR 命令で同一システム・レジスタの参照を行う場合，LDSR 命令のシステム・レジスタ設定が終わるまで，直後の STSR 命令はシステム・レジスタの使用が遅れます（ハザードの発生）。

V850 マイクロコントローラでは，インタロック機能により，直後の STSR 命令の ID ステージを遅らせます。

以上のことより，EIPC，FEPC の LDSR 命令実行結果を直後の STSR 命令で参照する場合，LDSR 命令の実行クロック数は，3 になります。

図 4―68　EIPC，FEPC を対象とする LDSR 命令実行結果の参照例



注　LDSR，STSR 命令で使用しているシステム・レジスタ番号の 0 は EIPC を表します。

備考　IL：インタロック機能によりデータの待ち合わせのために挿入されるアイドル
　　―：待ち合わせのために挿入されるアイドル

上図のように，EIPC，または FEPC をオペランドとする LDSR 命令の直後に，STSR 命令によってその結果を参照すると，インタロック機能によるデータの待ち合わせ時間が発生し，実行速度が低下します。LDSR 命令の結果を参照する STSR 命令は，LDSR 命令の 3 命令以後に配置することにより，実行速度の低下を防げます。

(e) プログラム作成時の注意点

プログラムを作成する場合，次のことに注意するとパイプラインが乱れず，命令実行速度が向上します。

- ロード命令（LD，SLD）の結果を使用する命令は，ロード命令の 2 命令以後に配置する。
- 乗算命令（MULH，MULHI）の結果を使用する命令は，乗算命令の 2 命令以後に配置する。
- LDSR 命令による EIPC，または FEPC への設定結果を STSR 命令により読み出す場合は，LDSR 命令の 3 命令以後に STSR 命令を配置する。
- 分岐先の最初の命令は，2 バイト長命令か，またはワード境界に配置された 4 バイト長命令を使用する。

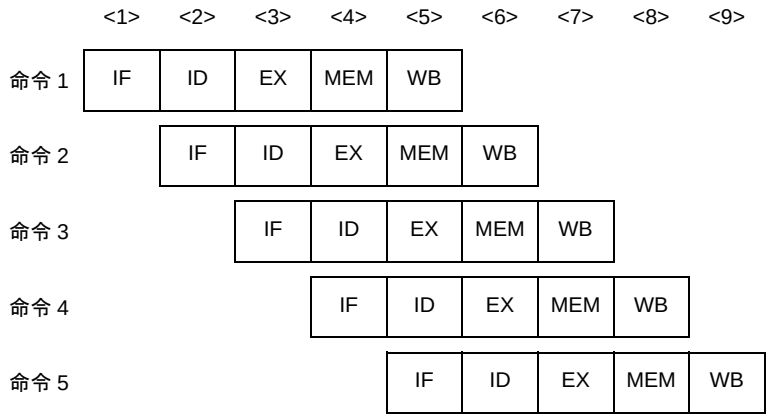
(2) パイプラインに関する補足事項

(a) ハーバード・アーキテクチャ

V850 マイクロコントローラでは，ハーバード・アーキテクチャを採用しており，内蔵 ROM からの命令フェッチ用のパスと，内蔵 RAM へのメモリ・アクセス用のパスが独立して動作します。これにより，IF ステージと MEM ステージのバス・アービトレーションの競合が発生せず，パイプラインがスムーズに流れます。

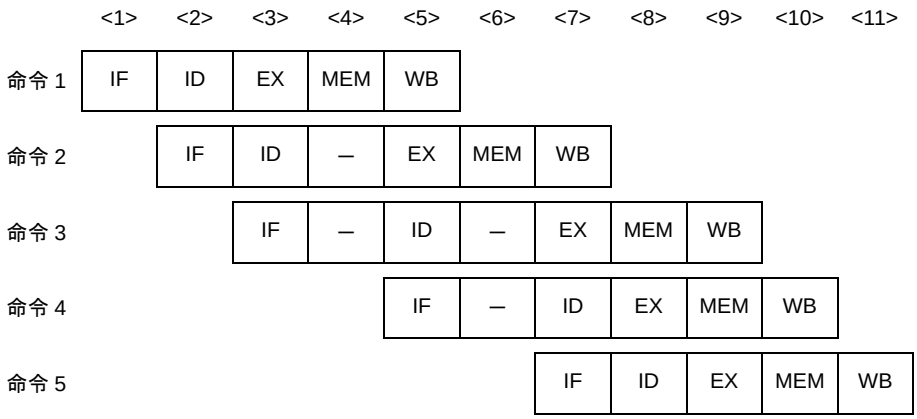
- V850 マイクロコントローラ（ハーバード・アーキテクチャ）の場合

命令 1 の MEM と命令 4 の IF，および命令 2 の MEM と命令 5 の IF が同時に実行でき，パイプラインが乱れません。



- 非ハーバード・アーキテクチャの場合

命令 1 の MEM と命令 4 の IF，および命令 2 の MEM と命令 5 の IF が競合するため，バスの待ち合わせが発生し，パイプラインが乱れ実行速度が低下します。



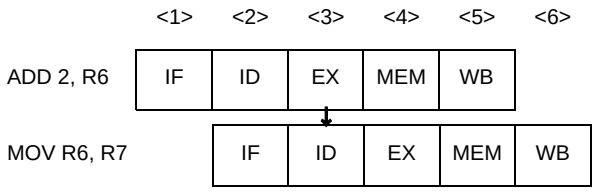
備考 —：待ち合わせのために挿入されるアイドル

（b）ショート・パス

V850 マイクロコントローラはショート・パスを内蔵しているため，前命令の WB（ライトバック）が終了する前に，後続の命令がその結果を使用できます。

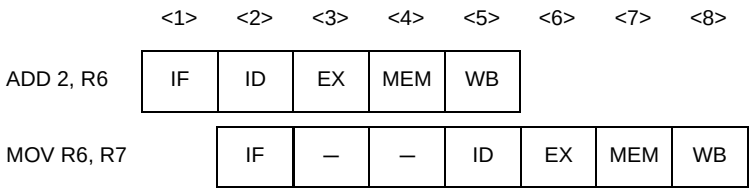
例 1. 算術演算命令，論理演算命令の実行結果を直後の命令で使用する場合：V850 マイクロコントローラ（ショート・パス内蔵）の場合

前命令の WB を待たず実行結果が出た時点（EX ステージ）で，直後の命令の ID を処理できます。



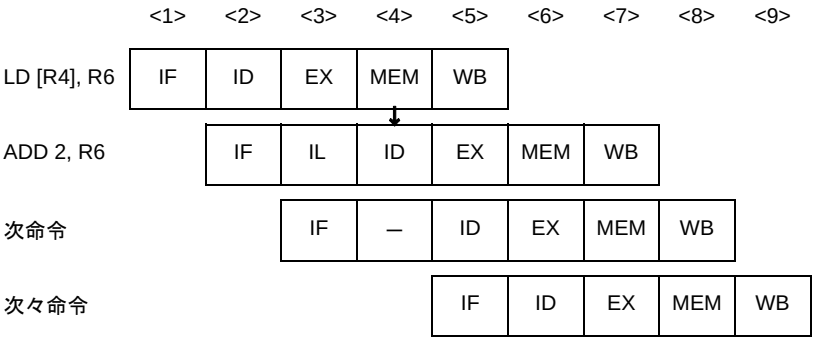
2. 算術演算命令，論理演算命令の実行結果を直後の命令で使用する場合：ショート・パスがない場合

前命令の WB まで，直後の ID が遅れます。



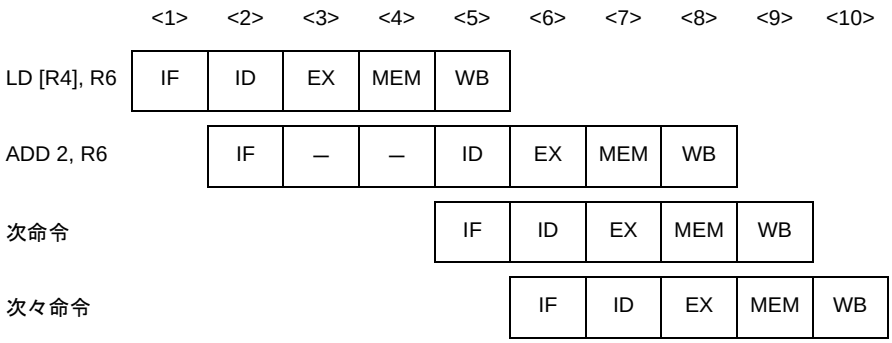
3. ロード命令によりメモリから読み出したデータを，直後の命令で使用する場合：V850 マイクロコントローラ（ショート・パス内蔵）の場合

前命令の WB を待たず実行結果が出た時点（MEM ステージ）で，直後の命令の ID を処理できます。



4. ロード命令によりメモリから読み出したデータを，直後の命令で使用する場合：ショート・パスがない場合

前命令の WB まで，直後の ID が遅れます。



(3) 各命令実行時のパイプラインの流れ

以下に，各命令実行時のパイプラインの流れについて説明します。

命令フェッチ（IF ステージ）は内蔵 ROM/PROM を，メモリ・アクセス（MEM ステージ）は内蔵 RAM を対象にしています。この場合，IF ステージ，MEM ステージは 1 クロックで処理されます。それ以外の場合は，所定のアクセス時間と，場合によってはバスの待ち合わせ時間がかかります。アクセス時間は次のとおりです。

表 4―65 アクセス時間（クロック数）

ステージ	内蔵 ROM/PROM（32 ビット）	内蔵 RAM（32 ビット）	内蔵周辺 I/O（8/16 ビット）	外部メモリ（16 ビット）
命令フェッチ（IF）	1	3	不可	3 + n
メモリ・アクセス（MEM）	3	1	3 + n	3 + n

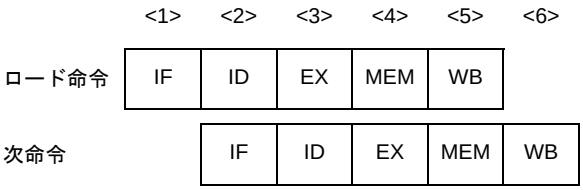
備考 n：ウエイト数

ロード命令

[対象の命令]

LD，SLD

[パイプライン]



[詳細説明]

パイプラインは IF，ID，EX，MEM，WB の 5 ステージです。

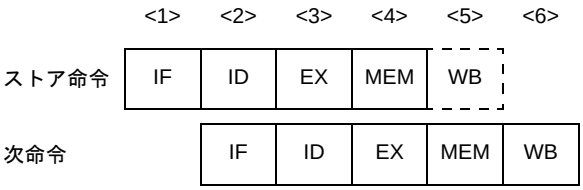
ロード命令の直後に，実行結果を使用する命令を配置すると，データの待ち合わせ時間が発生します。

ストア命令

[対象の命令]

SST，ST

[パイプライン]



[詳細説明]

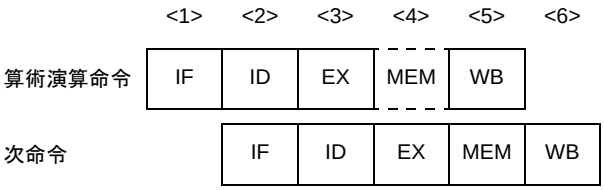
パイプラインは IF，ID，EX，MEM，WB の 5 ステージですが，レジスタへのデータの書き込みがないので WB ステージでは何も行いません。

算術演算命令（乗算命令／除算命令を除く）

[対象の命令]

ADD，ADDI，CMP，MOV，MOVEA，MOVHI，SETF，SUB，SUBR

[パイプライン]



[詳細説明]

パイプラインは IF，ID，EX，MEM，WB の 5 ステージですが，メモリへのアクセスがないので MEM ステージでは何も行いません。

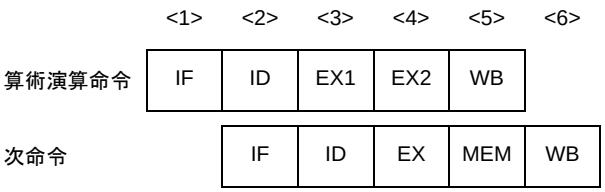
算術演算命令（乗算命令）

[対象の命令]

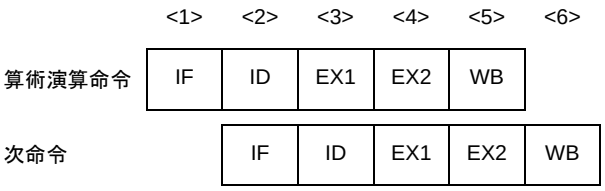
MULH，MULHI

[パイプライン]

（1）次命令が乗算命令以外の場合



（2）次命令が乗算命令の場合



[詳細説明]

パイプラインは IF，ID，EX1，EX2，WB の 5 ステージです。

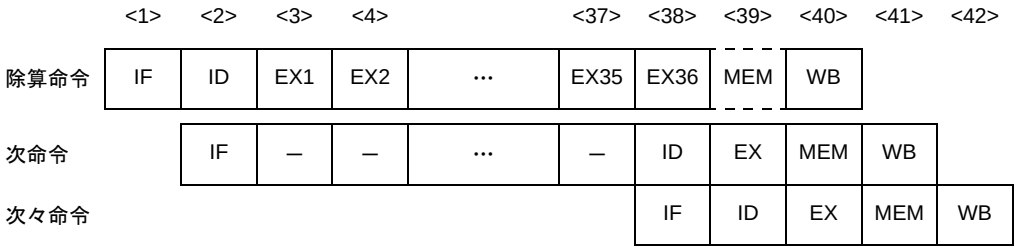
MEM ステージはありません。EX ステージには 2 クロックかかりますが，EX1 と EX2 は独立して動作できます。したがって，乗算命令を繰り返しても命令実行クロック数は，1 となります。ただし，乗算命令の直後に実行結果を使用する命令を配置すると，データの待ち合わせ時間が発生します。

算術演算命令（除算命令）

[対象の命令]

DIVH

[パイプライン]



備考 ー：待ち合わせのために挿入されるアイドル

[詳細説明]

パイプラインは IF，ID，EX1-EX36，MEM，WB の 40 ステージですが，メモリへのアクセスがないので MEM ステージでは何も行いません。

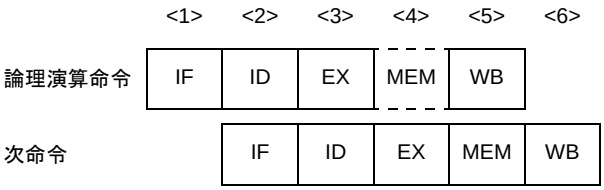
EX ステージには 36 クロックかかります。

論理演算命令

[対象の命令]

AND，ANDI，NOT，OR，ORI，SAR，SHL，SHR，TST，XOR，XORI

[パイプライン]



[詳細説明]

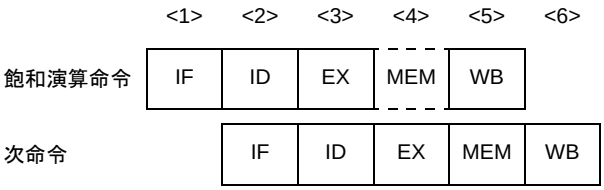
パイプラインは IF，ID，EX，MEM，WB の 5 ステージですが，メモリへのアクセスがないので MEM ステージでは何も行いません。

飽和演算命令

[対象の命令]

SATADD，SATSUB，SATSUBI，SATSUBR

[パイプライン]



[詳細説明]

パイプラインは IF，ID，EX，MEM，WB の 5 ステージですが，メモリへのアクセスがないので MEM ステージでは何も行いません。

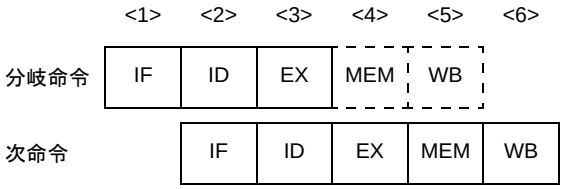
分岐命令（条件分岐命令：BR 命令を除く）

[対象の命令]

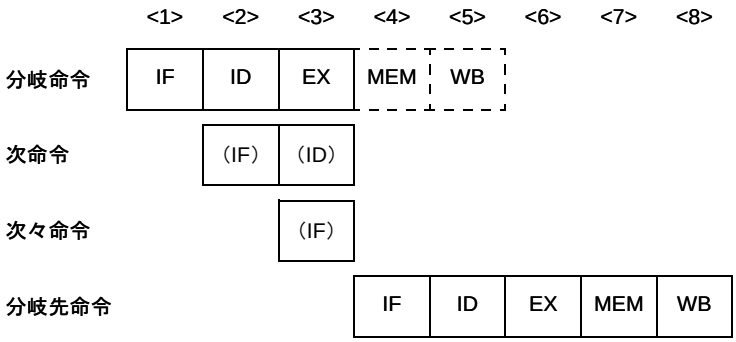
Bcnd 命令

[パイプライン]

(1) 条件が成立しない場合



(2) 条件が成立した場合



備考　(IF)：無効となる命令フェッチ
(ID)：無効となる命令デコード

[詳細説明]

パイプラインは IF，ID，EX，MEM，WB の 5 ステージですが，メモリへのアクセス，レジスタへのデータ書き込みがないので MEM ステージ，WB ステージでは何も行いません。

(1) 条件が成立しない場合

分岐命令の命令実行クロック数は，1 となります。

(2) 条件が成立した場合

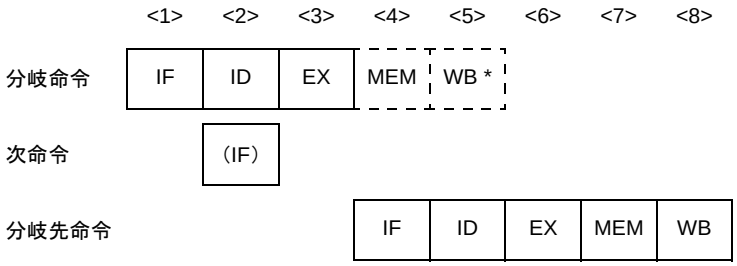
分岐命令の命令実行クロック数は，3 となります。分岐命令の次命令の IF，次々命令の IF は無効となります。

分岐命令（BR 命令，無条件分岐命令）

[対象の命令]

BR，JARL，JMP，JR

[パイプライン]



備考 （IF）：無効となる命令フェッチ

WB * : JMP 命令，JR 命令，BR 命令の場合は何も行われませんが，JARL 命令の場合は復帰 PC の書き込みが行われます。

[詳細説明]

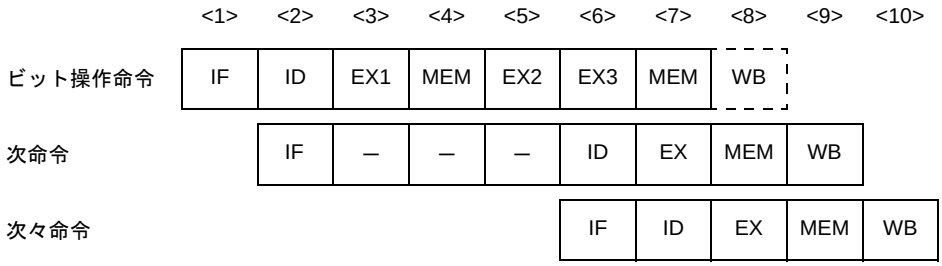
パイプラインは IF，ID，EX，MEM，WB の 5 ステージですが，メモリへのアクセス，レジスタへのデータ書き込みがないので MEM ステージ，WB ステージでは何も行いません。ただし，JARL 命令の場合には WB ステージにおいて復帰 PC の書き込みが行われます。また，分岐命令の次命令の IF は無効となります。

ビット操作命令（CLR1，NOT1，SET1 命令）

［対象の命令］

CLR1，NOT1，SET1

［パイプライン］



備考 ー：待ち合わせのために挿入されるアイドル

［詳細説明］

パイプラインは IF，ID，EX1，MEM，EX2，EX3，MEM，WB の 8 ステージですが，レジスタへのデータ書き込みがないので WB ステージでは何も行いません。

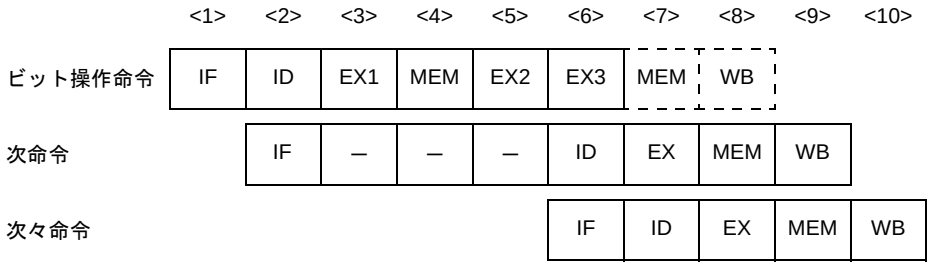
この命令では，メモリ・アクセスがリード・モディファイ・ライトとなり，EX ステージには 3 クロック，MEM ステージには 2 クロックかかります。

ビット操作命令（TST1 命令）

［対象の命令］

TST1

［パイプライン］



備考　ー：待ち合わせのために挿入されるアイドル

［詳細説明］

パイプラインは IF，ID，EX1，MEM，EX2，EX3，MEM，WB の 8 ステージですが，2 回目のメモリへのアクセス，レジスタへのデータ書き込みがないので 2 回目の MEM ステージ，WB ステージでは何も行いません。

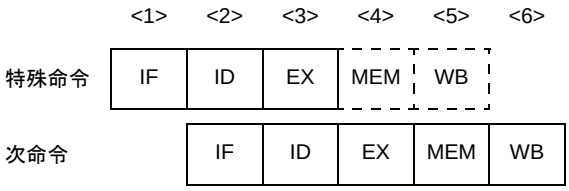
この命令では，メモリ・アクセスがリード・モディファイ・ライトとなり，EX ステージには 3 クロック，MEM ステージには 2 クロックかかります。

特殊命令（DI，EI 命令）

[対象の命令]

DI，EI

[パイプライン]



[詳細説明]

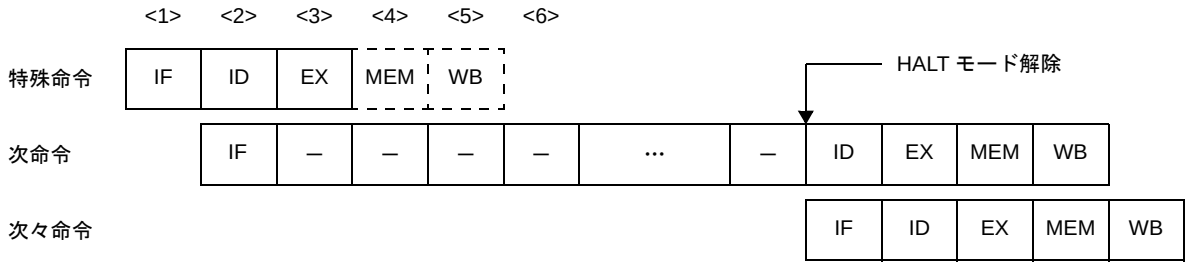
パイプラインは IF，ID，EX，MEM，WB の 5 ステージですが，メモリへのアクセス，レジスタへのデータ書き込みがないので MEM ステージ，WB ステージでは何も行いません。

特殊命令（HALT 命令）

[対象の命令]

HALT

[パイプライン]



備考 ー：待ち合わせのために挿入されるアイドル

[詳細説明]

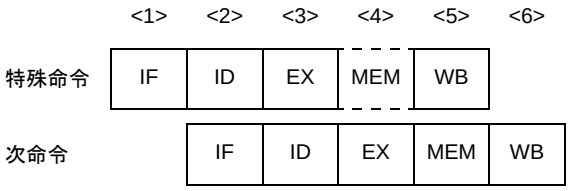
パイプラインは IF，ID，EX，MEM，WB の 5 ステージですが，メモリへのアクセス，レジスタへのデータ書き込みがないので MEM ステージ，WB ステージでは何も行いません。また，次命令では，HALT モードが解除されるまで ID ステージが遅れます。

特殊命令（LDSR，STSR 命令）

［対象の命令］

LDSR，STSR

［パイプライン］



［詳細説明］

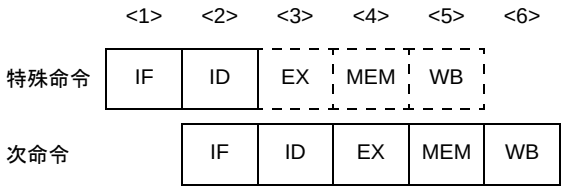
パイプラインは IF，ID，EX，MEM，WB の 5 ステージですが，メモリへのアクセスがないので MEM ステージでは何も行いません。また，システム・レジスタの EIPC，FEPC を設定する LDSR 命令の直後に，同一レジスタを使用する STSR 命令を配置すると，データの待ち合わせ時間が発生します。

特殊命令（NOP 命令）

[対象の命令]

NOP

[パイプライン]



[詳細説明]

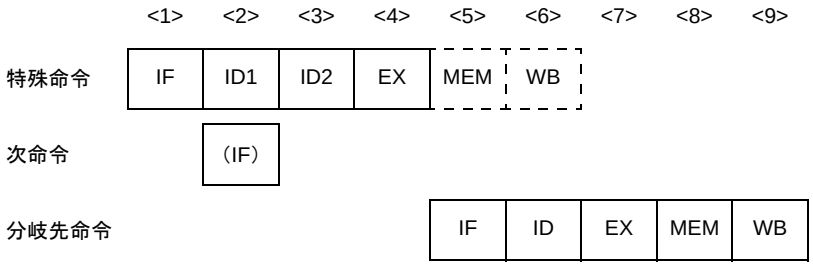
パイプラインは IF，ID，EX，MEM，WB の 5 ステージですが，演算，メモリへのアクセス，レジスタへのデータ書き込みがないので EX ステージ，MEM ステージ，WB ステージでは何も行いません。

特殊命令（RETI 命令）

[対象の命令]

RETI

[パイプライン]



備考 （IF）：無効となる命令フェッチ
ID1：レジスタ選択
ID2：EIPC/FEPC 読み込み

[詳細説明]

パイプラインは IF，ID1，ID2，EX，MEM，WB の 6 ステージですが，メモリへのアクセス，レジスタへのデータ書き込みがないので MEM ステージ，WB ステージでは何も行いません。

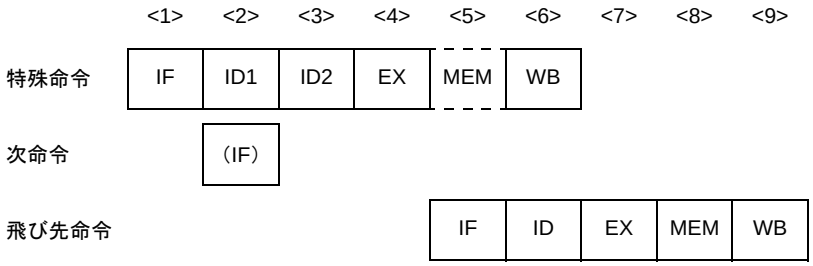
ID ステージには 2 クロックかかります。また，次命令の IF，次々命令の IF は無効となります。

特殊命令（TRAP 命令）

［対象の命令］

TRAP

［パイプライン］



備考 （IF）：無効となる命令フェッチ
ID1：TRAP コードの検出
ID2：アドレス生成

［詳細説明］

パイプラインは IF，ID1，ID2，EX，MEM，WB の 6 ステージですが，メモリへのアクセスがないので MEM ステージでは何も行いません。

ID ステージには 2 クロックかかります。また，次命令の IF，次々命令の IF は無効となります。

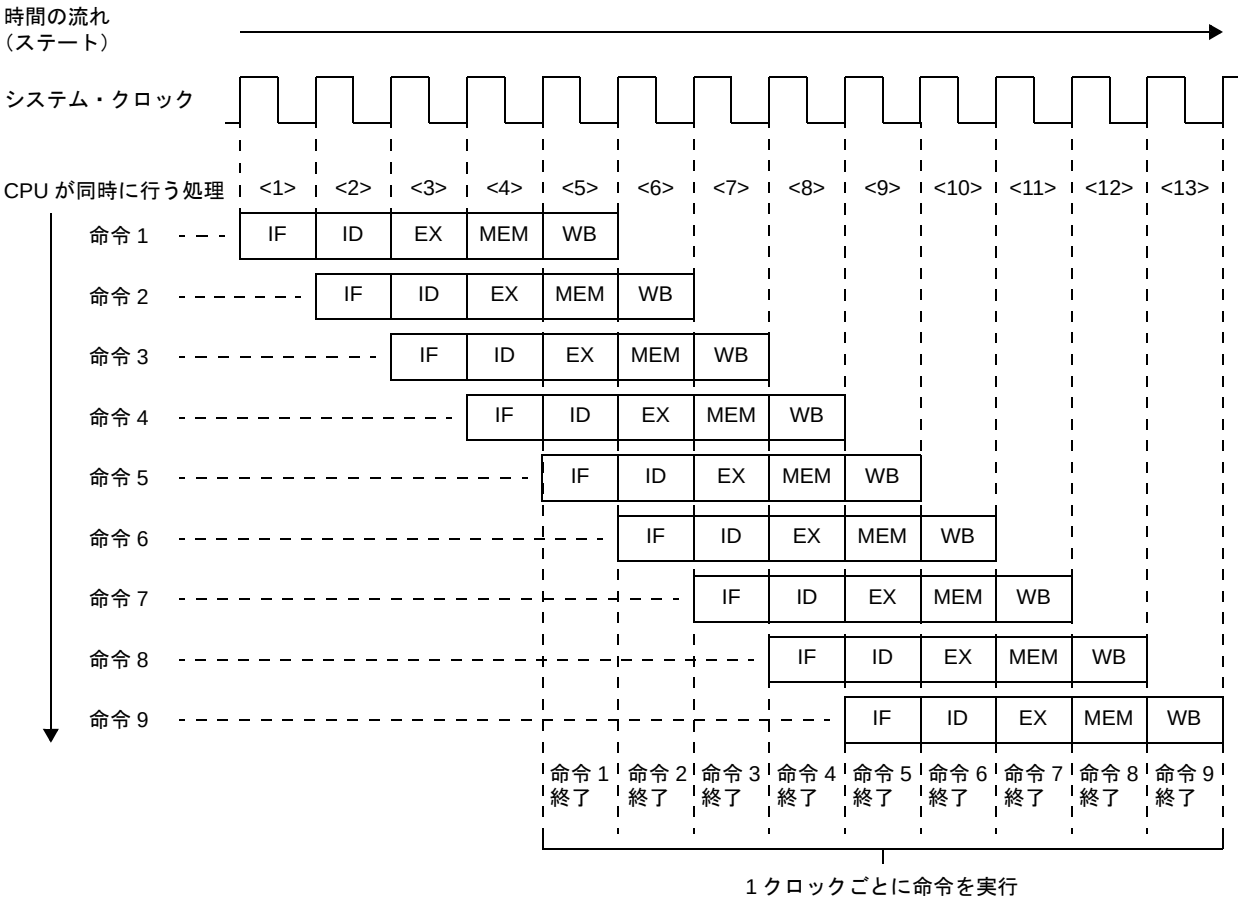
4. 5. 15　パイプライン（V850ES）

V850ES は，RISC アーキテクチャをベースとした 5 段パイプライン制御により，ほとんどの命令を 1 クロックで実行します。命令実行手順は，通常，IF（インストラクション・フェッチ）から WB（ライトバック）までの 5 ステージで構成されています。

IF（インストラクション・フェッチ）	命令をフェッチし，フェッチ・ポインタをインクリメント
ID（インストラクション・デコード）	命令をデコードし，イミーディエト・データの作成，レジスタの読み出しを行う
EX（実行）	デコードした命令を実行
MEM（メモリ・アクセス）	対象アドレスのメモリをアクセス
WB（ライトバック）	レジスタに実行結果を書き込む

各ステージの実行時間は，命令の種類，アクセス対象となるメモリの種類などによって異なります。
パイプラインの動作例として，標準的な命令を 9 個続けて実行した際の CPU の処理を下図に示します。

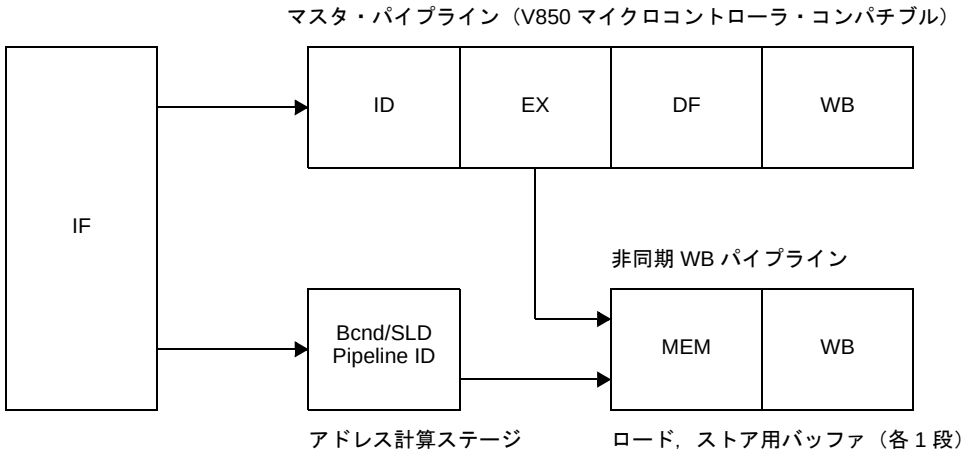
図 4―69　標準的な命令を 9 個続けて実行する例



<1> ～ <13> は CPU のステートを示します。各ステートでは，命令 n の WB（ライトバック），命令 n + 1 の MEM（メモリ・アクセス），命令 n + 2 の EX（実行），命令 n + 3 の ID（インストラクション・デコード），命令 n + 4 の IF（インストラクション・フェッチ）が同時に行われます。標準的な命令では，IF ステージから WB ステージまで 5 クロックかかります。しかし，同時に 5 命令を処理できるため，標準的な命令では平均 1 クロックごとに実行可能です。

V850ES では，パイプラインの最適化を行うことにより，CPI（Cycle per instruction）を従来の V850 マイクロコントローラよりも向上させています。以下に，V850ES のパイプライン構成を示します。

図 4―70　パイプライン構成（V850ES）



備考　DF（データ・フェッチ）：WB ステージに実行データを転送

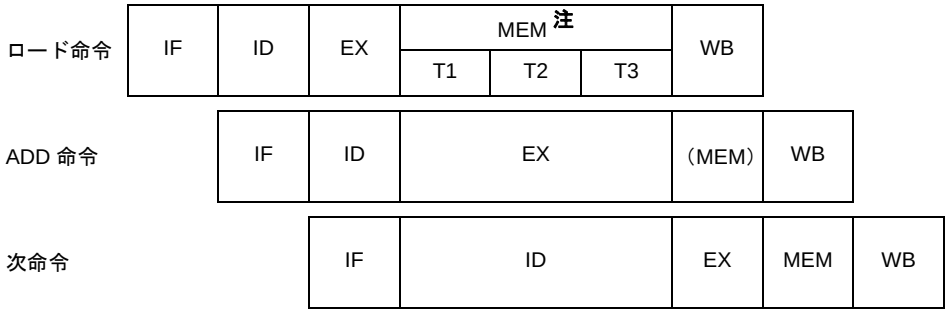
(1) ノンブロッキング・ロード／ストア

外部メモリ・アクセス時にパイプラインが停止することなく，効率的な処理が可能です。

例として，外部メモリに対するロード命令実行後に ADD 命令を実行する場合の V850 マイクロコントローラと V850ES のパイプライン動作の比較を以下に示します。

(a) V850 マイクロコントローラの場合

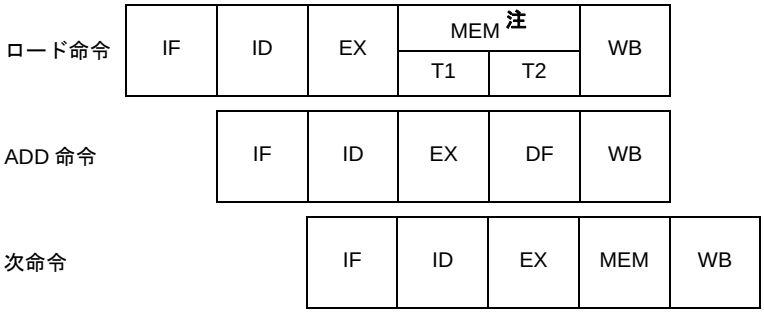
ADD 命令の EX ステージは，本来 1 クロックで実行されます。しかし，直前のロード命令の MEM ステージ実行中，ADD 命令の EX ステージに待ち時間が発生します。これは，パイプライン上の 5 つの命令が，同一内部クロック間に同じステージを実行できないためです。この影響により，ADD 命令の次命令の ID ステージにも待ち時間が発生します。



注 外部メモリに対する基本バス・サイクルは，3 クロックです。

（b）V850ES の場合

マスタ・パイプラインのほかに，MEM ステージを必要とする命令用に非同期 WB パイプラインを持っています。このため，ロード命令の MEM ステージは，非同期 WB パイプラインで処理されます。下図の例では，ADD 命令はマスタ・パイプラインで処理されるため，パイプラインの待ち時間が発生することなく，効率的に命令を実行できます。



注 外部メモリに対する基本バス・サイクルは，2 クロックです。

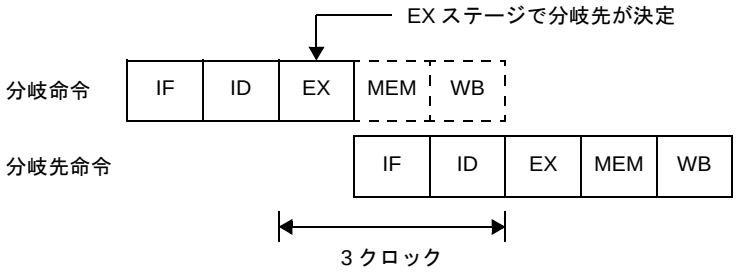
（2）2 クロック分岐

分岐命令実行時は，ID ステージで分岐先が決定します。

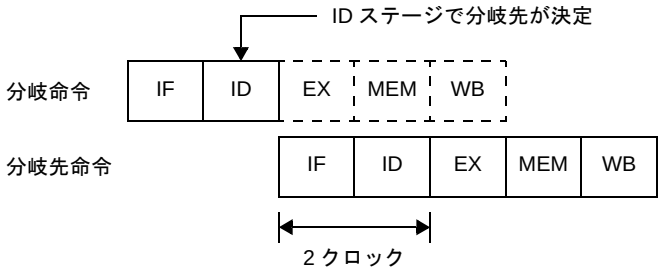
V850 マイクロコントローラの場合，分岐命令実行時の分岐先は，EX ステージ実行後に決定されていましたが，V850ES では，分岐／ SLD 命令用に追加したアドレス計算ステージにより，ID ステージで分岐先が決定します。このため，V850ES は，V850 マイクロコントローラと比較して，1 クロック早く分岐先の命令をフェッチすることができます。

V850 マイクロコントローラと V850ES の分岐命令でのパイプライン動作の比較を以下に示します。

（a）V850 マイクロコントローラの場合

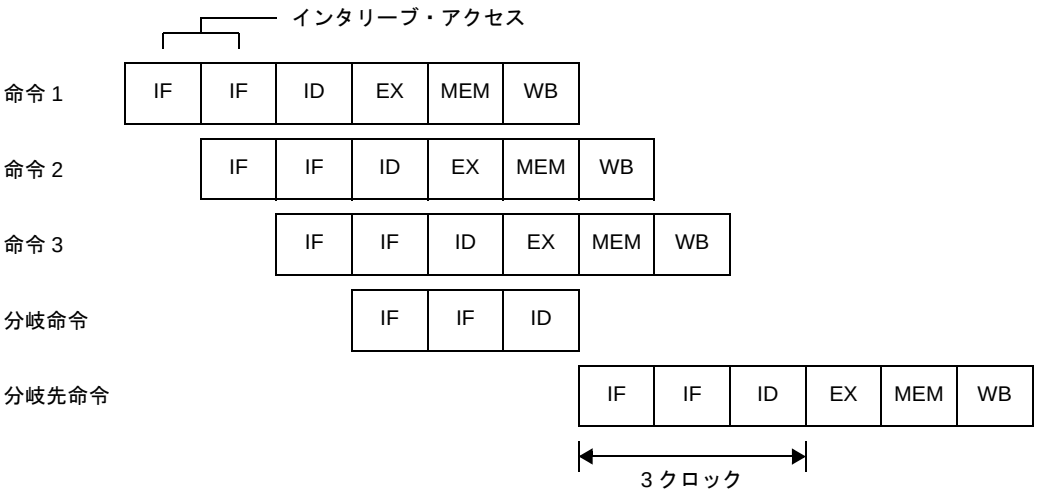


（b）V850ES の場合



備考　V850ES タイプ B の製品は，内蔵フラッシュ・メモリ，または内蔵マスク ROM に対し，インターリーブ・アクセスを行います。このため，割り込み発生直後の命令フェッチや，分岐命令実行後の命令フェッチに 2 クロックかかります。したがって，分岐先命令の ID ステージ実行までに 3 クロックかかります。

例



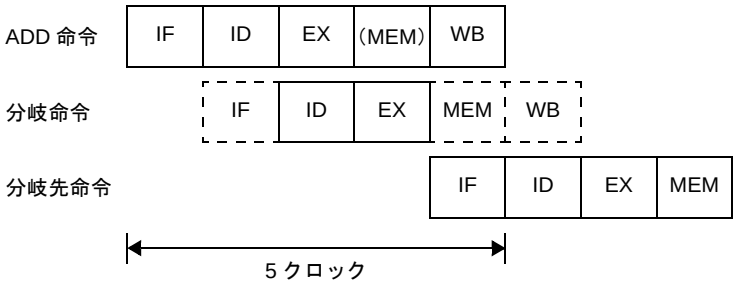
(3) 効率的なパイプライン処理

V850ES は，マスタ・パイプライン上の ID ステージのほかに，分岐／ SLD 命令用の ID ステージを持っているため，効率的なパイプライン処理が行えます。

例として，ADD 命令の IF ステージで，次分岐命令をフェッチした場合のパイプライン動作例を以下に示します（専用バスに直結された ROM に対する命令フェッチは，32 ビット単位で行われます。以下の ADD 命令，分岐命令はどちらも 16 ビット・フォーマット命令です）。

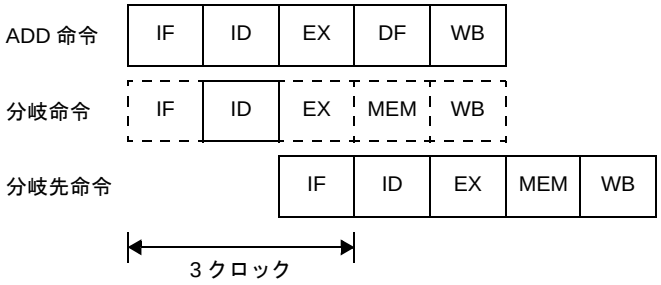
(a) V850 マイクロコントローラの場合

ADD 命令の IF ステージで次分岐命令の命令コードまでフェッチしますが，ADD 命令の ID ステージと分岐命令の ID ステージを同一クロック中に実行できません。そのため，分岐命令のフェッチから分岐先命令のフェッチまで，5 クロックかかります。



(b) V850ES の場合

マスタ・パイプライン上の ID ステージのほかに，分岐／ SLD 命令用の ID ステージを持っているため，同一クロック中に並行して ADD 命令の ID ステージと分岐命令の ID ステージが実行できます。このため，分岐命令のフェッチ開始から分岐先の命令フェッチ完了まで，3 クロックで実行できます。



備考 SLD 命令と Bcnd 命令については，ほかの 16 ビット・フォーマットの命令と同時実行される場合があるため，注意が必要です。たとえば，SLD 命令と NOP 命令が同時に実行された場合，NOP 命令によるディレイ・タイムの発生が行われない可能性があります。

(4) パイプラインの乱れ

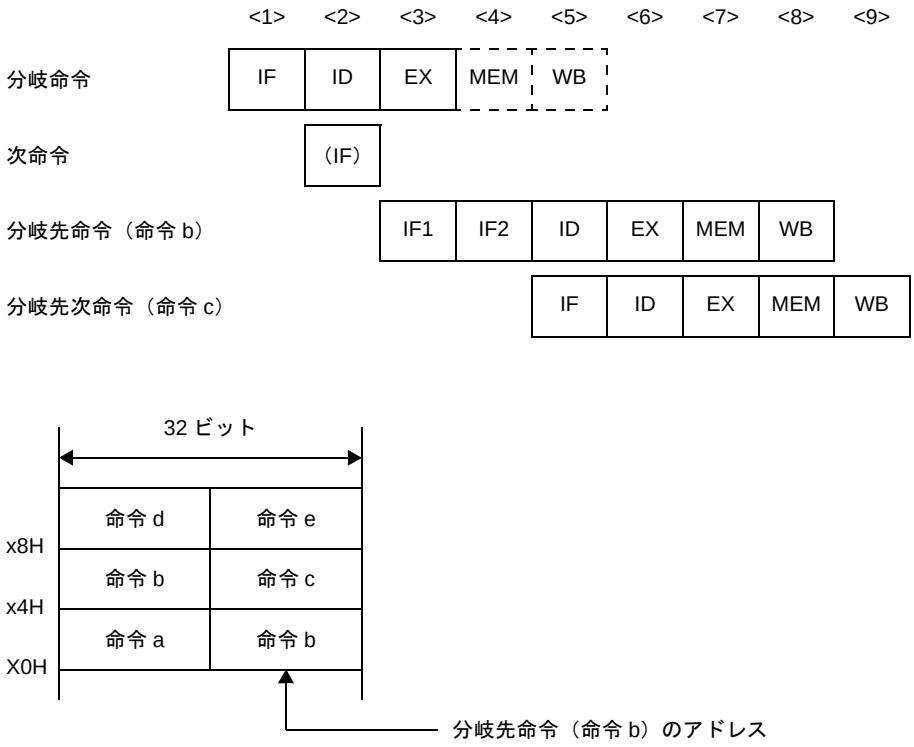
パイプラインは IF（インストラクション・フェッチ）から WB（ライトバック）までの 5 ステージで構成され，基本的にはそれぞれのステージは 1 クロックで処理されますが，場合によってはパイプラインが乱れて，実行クロック数が増加する場合があります。以下に，パイプラインを乱す主な要因を示します。

(a) アライン・ハザード

分岐先命令のアドレスがワード・アラインではなく（A1 = 1，A0 = 0），かつ 4 バイト長命令の場合，命令をワード単位に揃えるために IF を 2 回続ける必要があります。これをアライン・ハザードと呼びます。

たとえば，命令 a から命令 e までがアドレス X0H から配置されており，命令 b は 4 バイト長命令で，その他の命令は 2 バイト長命令であるとします。この場合，命令 b は X2H に配置され（A1 = 1，A0 = 0），ワード・アライン（A1 = 0，A0 = 0）となっていません。したがって，この命令 b が分岐先命令となる場合，アライン・ハザードが発生します。アライン・ハザードが発生した場合の分岐命令の実行クロック数は，4 となります。

図 4―71　アライン・ハザードの例



備考　(IF)：無効となる命令フェッチ

IF1：アライン・ハザード時に発生する 1 回目の命令フェッチです。2 バイト長のフェッチで，命令 b の下位アドレス側の 2 バイトがフェッチされます。

IF2：アライン・ハザード時に発生する 2 回目の命令フェッチです。通常の 4 バイト長のフェッチで，命令 b の上位アドレス側の 2 バイトと命令 c（2 バイト長）がフェッチされます。

アライン・ハザードは，次のような対処によって回避が可能で，命令実行速度の向上が図れます。

- 分岐先命令に 2 バイト長命令を使用する
- 分岐先命令にワード境界（A1 = 0，A0 = 0）に配置した 4 バイト長命令を使用する

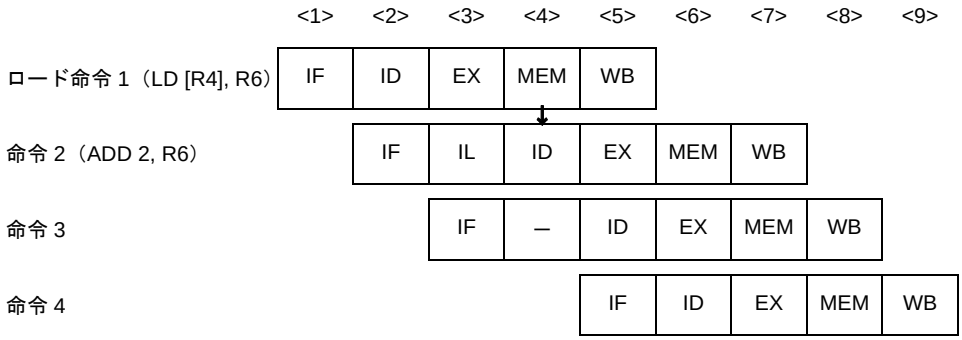
（b）ロード命令実行結果の参照

ロード命令（LD，SLD）では，MEM ステージで読み出されたデータの格納が WB ステージで行われます。したがって，ロード命令の直後の命令で同一のレジスタの内容を使用する場合，ロード命令がレジスタの使用を終えるまで，直後の命令はレジスタの使用を遅らせる必要があります。これをハザードと呼びます。V850ES は，このハザードを CPU で自動的に対処するインタロック機能を持っており，次命令の ID ステージを遅らせます。

また，V850ES は，MEM ステージで読み出したデータを次命令の ID ステージで使用できるように，ショート・パスを持っています。このショート・パスによって，ロード命令によって MEM ステージでデータを読み出すことと，このデータを次命令の ID ステージで使用することを，同一タイミングで行うことができます。

以上のことにより，結果を直後の命令で使用する場合，ロード命令の実行クロック数は，2 になります。

図 4―72　ロード命令実行結果の参照例



備考　IL：インタロック機能によりデータの待ち合わせのために挿入されるアイドル
―：待ち合わせのために挿入されるアイドル
↓：ショート・パス

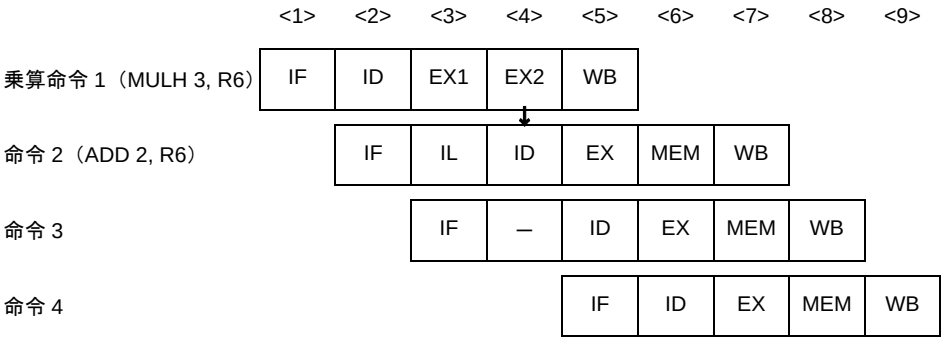
上図のように，ロード命令の直後にその結果を使用する命令を配置すると，インタロック機能によるデータの待ち合わせ時間が発生し，実行速度が低下します。ロード命令の結果を使用する命令は，ロード命令の 2 命令以後に配置することにより，実行速度の低下を防げます。

（c）乗算命令実行結果の参照

乗算命令では，乗算結果のレジスタへの格納が WB ステージで行われます。したがって，乗算命令の直後の命令で同一レジスタの内容を使用する場合，乗算命令がレジスタの使用を終えるまで，直後の命令はレジスタの使用を遅らせる必要があります（ハザードの発生）。

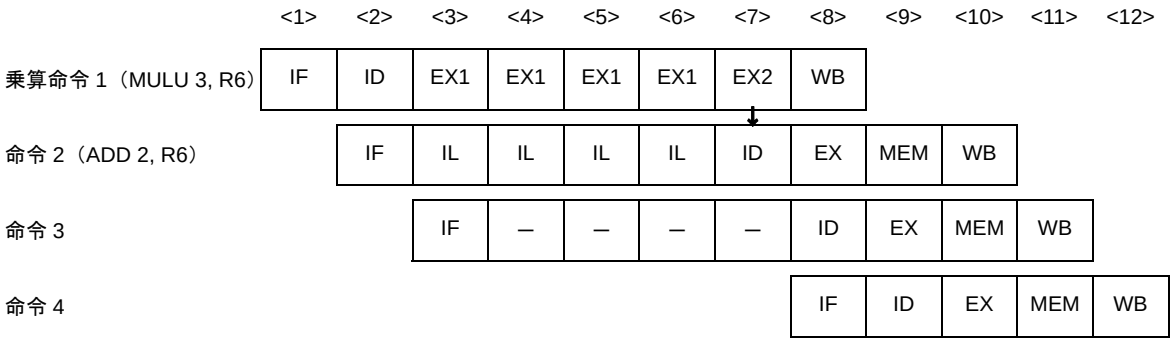
V850ES では，インタロック機能により直後の命令の ID ステージを遅らせます。また，ショート・パスにより，乗算命令の EX2 ステージと，この演算結果を直後の命令の ID ステージで使用することが，同一タイミングで行えます。

図 4―73　乗算命令実行結果の参照例（ハーフワード乗算命令の場合）



備考　IL：インタロック機能によりデータの待ち合わせのために挿入されるアイドル
　　―：待ち合わせのために挿入されるアイドル
　　↓：ショート・パス

図 4―74　乗算命令実行結果の参照例（ワード乗算命令の場合）



備考　IL：インタロック機能によりデータの待ち合わせのために挿入されるアイドル
　　―：待ち合わせのために挿入されるアイドル
　　↓：ショート・パス

上図のように，乗算命令の直後にその結果を使用する命令を配置すると，インタロック機能によるデータの待ち合わせ時間が発生し，実行速度が低下します。乗算命令の結果を使用する命令は，乗算命令の 2 命令以後に配置することにより，実行速度の低下を防げます。ただし，ワード・データ乗算命令（MUL，MULU）の場合，乗算命令の結果を使用する命令は，乗算命令の 5 命令以後に配置しなければ，1-4 の IL ステージが挿入されます。

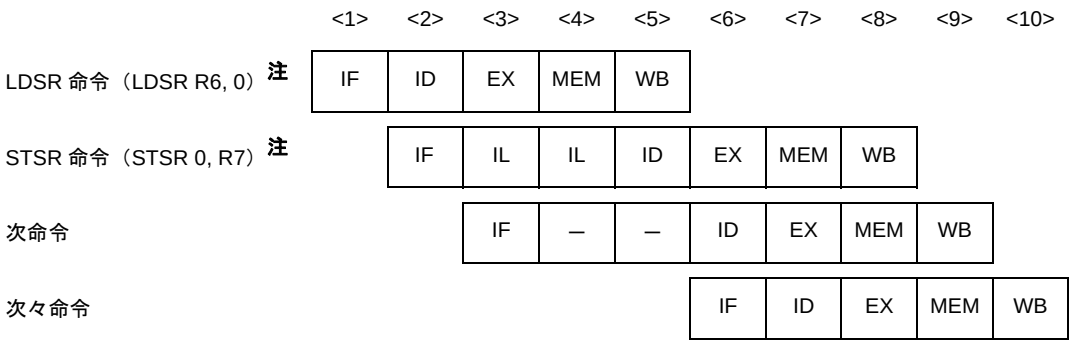
（d）EIPC，FEPC を対象とする LDSR 命令実行結果の参照

LDSR 命令によって，システム・レジスタの EIPC，FEPC のデータ設定を行い，直後に STSR 命令で同一システム・レジスタの参照を行う場合，LDSR 命令のシステム・レジスタ設定が終わるまで，直後の STSR 命令はシステム・レジスタの使用が遅れます（ハザードの発生）。

V850ES では，インタロック機能により，直後の STSR 命令の ID ステージを遅らせます。

以上のことより，EIPC，FEPC の LDSR 命令実行結果を直後の STSR 命令で参照する場合，LDSR 命令の実行クロック数は，3 になります。

図 4―75　EIPC，FEPC を対象とする LDSR 命令実行結果の参照例



注　LDSR，STSR 命令で使用しているシステム・レジスタ番号の 0 は EIPC を表します。

備考　IL：インタロック機能によりデータの待ち合わせのために挿入されるアイドル
　　―：待ち合わせのために挿入されるアイドル

上図のように，EIPC，または FEPC をオペランドとする LDSR 命令の直後に，STSR 命令によってその結果を参照すると，インタロック機能によるデータの待ち合わせ時間が発生し，実行速度が低下します。LDSR 命令の結果を参照する STSR 命令は，LDSR 命令の 3 命令以後に配置することにより，実行速度の低下を防げます。

（e）プログラム作成時の注意点

プログラムを作成する場合，次のことに注意するとパイプラインが乱れず，命令実行速度が向上します。

- ロード命令（LD，SLD）の結果を使用する命令は，ロード命令の 2 命令以後に配置する。
- 乗算命令（MULH，MULHI）の結果を使用する命令は，乗算命令の 2 命令以後に配置する。
- LDSR 命令による EIPC，または FEPC への設定結果を STSR 命令により読み出す場合は，LDSR 命令の 3 命令以後に STSR 命令を配置する。
- 分岐先の最初の命令は，2 バイト長命令か，またはワード境界に配置された 4 バイト長命令を使用する。

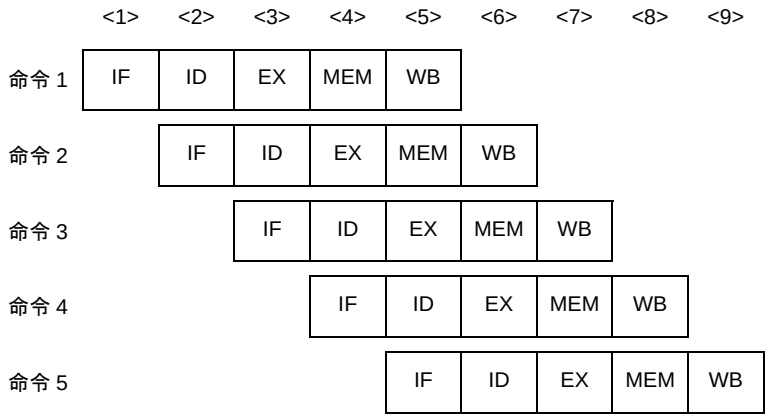
(5) パイプラインに関する補足事項

(a) ハーバード・アーキテクチャ

V850ES では，ハーバード・アーキテクチャを採用しており，内蔵 ROM からの命令フェッチ用のパスと，内蔵 RAM へのメモリ・アクセス用のパスが独立して動作します。これにより，IF ステージと MEM ステージのバス・アービトレーションの競合が発生せず，パイプラインがスムーズに流れます。

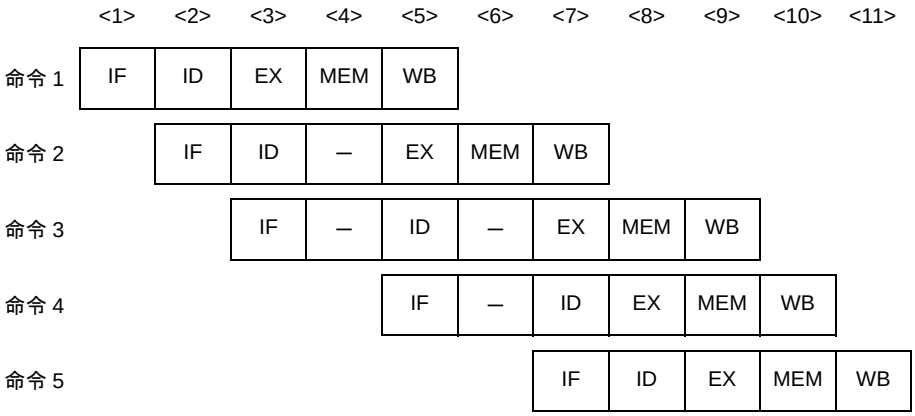
- V850ES（ハーバード・アーキテクチャ）の場合

命令 1 の MEM と命令 4 の IF，および命令 2 の MEM と命令 5 の IF が同時に実行でき，パイプラインが乱れません。



- 非ハーバード・アーキテクチャの場合

命令 1 の MEM と命令 4 の IF，および命令 2 の MEM と命令 5 の IF が競合するため，バスの待ち合わせが発生し，パイプラインが乱れ実行速度が低下します。



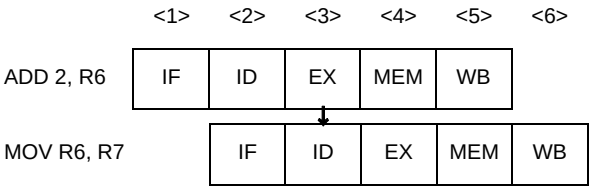
備考　ー：待ち合わせのために挿入されるアイドル

（b）ショート・パス

V850ES はショート・パスを内蔵しているため，前命令の WB（ライトバック）が終了する前に，後続の命令がその結果を使用できます。

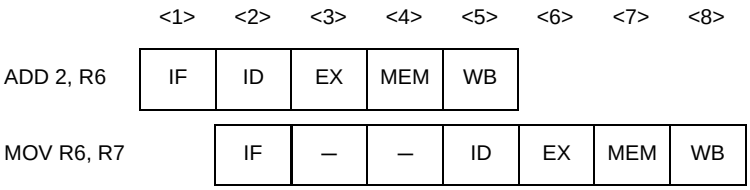
例 1. 算術演算命令，論理演算命令の実行結果を直後の命令で使用する場合：V850ES（ショート・パス内蔵）の場合

前命令の WB を待たず実行結果が出た時点（EX ステージ）で，直後の命令の ID を処理できます。



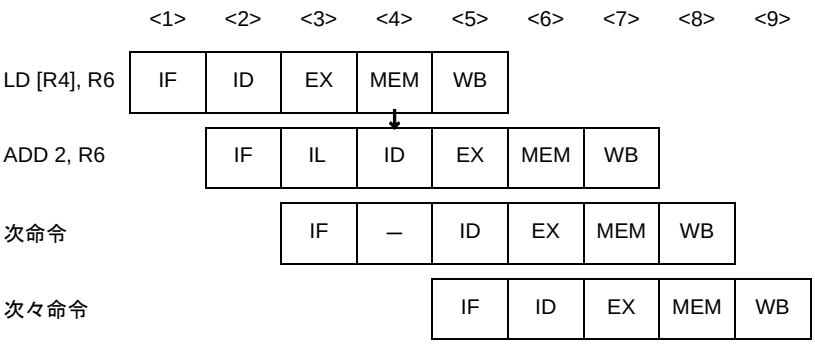
2. 算術演算命令，論理演算命令の実行結果を直後の命令で使用する場合：ショート・パスがない場合

前命令の WB まで，直後の ID が遅れます。



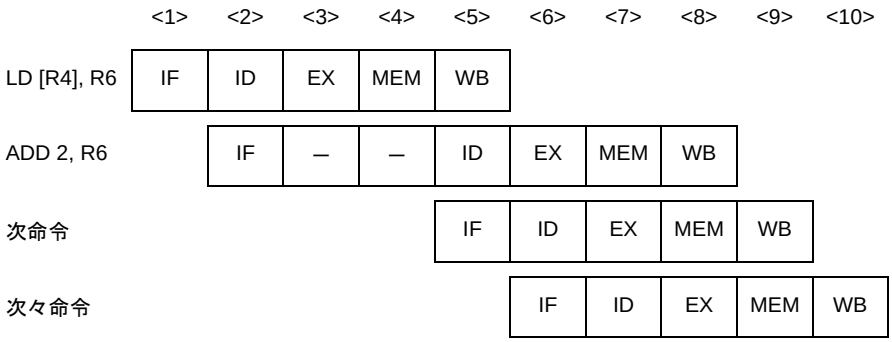
3. ロード命令によりメモリから読み出したデータを，直後の命令で使用する場合：V850ES（ショート・パス内蔵）の場合

前命令の WB を待たず実行結果が出た時点（MEM ステージ）で，直後の命令の ID を処理できます。



4. ロード命令によりメモリから読み出したデータを，直後の命令で使用する場合：ショート・パスがない場合

前命令の WB まで，直後の ID が遅れます。



（6）各命令実行時のパイプラインの流れ

以下に，各命令実行時のパイプラインの流れについて説明します。

パイプライン処理のため，メモリや I/O のライト・サイクルが発生する時点で，CPU は後続の命令をすでに実行しています。そのため，I/O 操作や割り込み要求マスクの反映は，次命令発行（ID ステージ）に対して遅れて実行されます。

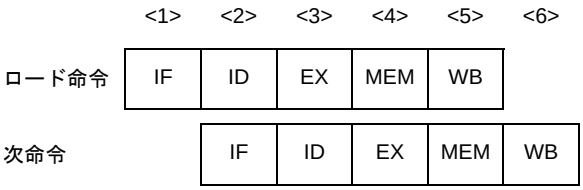
内蔵 INTC へのアクセスを CPU が検出（ID ステージ）して割り込み要求のマスク処理を行うため，割り込みマスク操作を行う場合，直後の命令からマスカブル割り込みの受け付けを禁止します。

ロード命令（LD 命令）

[対象の命令]

LD.B，LD.BU，LD.H，LD.HU，LD.W

[パイプライン]



[詳細説明]

パイプラインは IF，ID，EX，MEM，WB の 5 ステージです。
LD 命令の直後に，実行結果を使用する命令を配置すると，データの待ち合わせ時間が発生します。

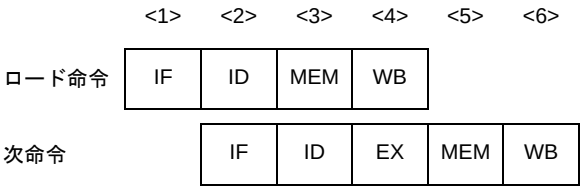
備考　ノンブロッキング制御のため，MEM ステージの間にバス・サイクルが完了している保証はありません。ただし，周辺 I/O 領域へのアクセスはブロッキング制御となるため，MEM ステージでバス・サイクルの完了を待ち合わせます。

ロード命令（SLD 命令）

[対象の命令]

SLD.B，SLD.BU，SLD.H，SLD.HU，SLD.W

[パイプライン]



[詳細説明]

パイプラインは IF，ID，MEM，WB の 4 ステージです。

SLD 命令の直後に，実行結果を使用する命令を配置すると，データの待ち合わせ時間が発生します。

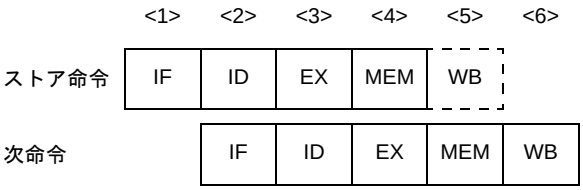
備考 ノンブロッキング制御のため，MEM ステージの間にバス・サイクルが完了している保証はありません。ただし，周辺 I/O 領域へのアクセスはブロッキング制御となるため，MEM ステージでバス・サイクルの完了を待ち合わせます。

ストア命令

[対象の命令]

SST.B，SST.H，SST.W，ST.B，ST.H，ST.W

[パイプライン]



[詳細説明]

パイプラインは IF，ID，EX，MEM，WB の 5 ステージですが，レジスタへのデータの書き込みがないので WB ステージでは何も行いません。

備考 ノンブロッキング制御のため，MEM ステージの間にバス・サイクルが完了している保証はありません。ただし，周辺 I/O 領域へのアクセスはブロッキング制御となるため，MEM ステージでバス・サイクルの完了を待ち合わせます。

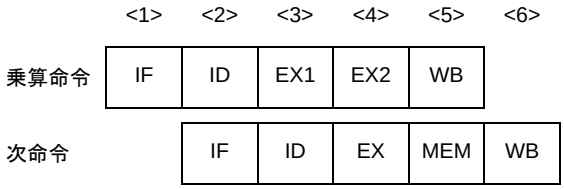
乗算命令（ハーフワード・データ乗算命令）

[対象の命令]

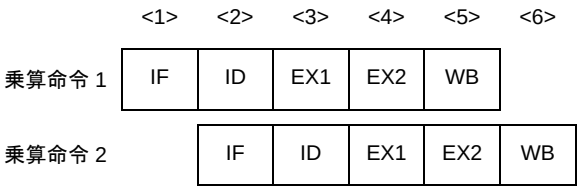
MULH，MULHI

[パイプライン]

（1）次命令が乗算命令以外の場合



（2）次命令が乗算命令の場合



[詳細説明]

パイプラインは IF，ID，EX1，EX2，WB の 5 ステージです。

EX ステージは乗算器実行のため 2 クロックかかりますが，EX1 と EX2（通常の EX ステージとは異なります）は独立して動作できます。したがって，乗算命令を繰り返しても命令実行クロック数は，1 となります。ただし，乗算命令の直後に実行結果を使用する命令を配置すると，データの待ち合わせ時間が発生します。

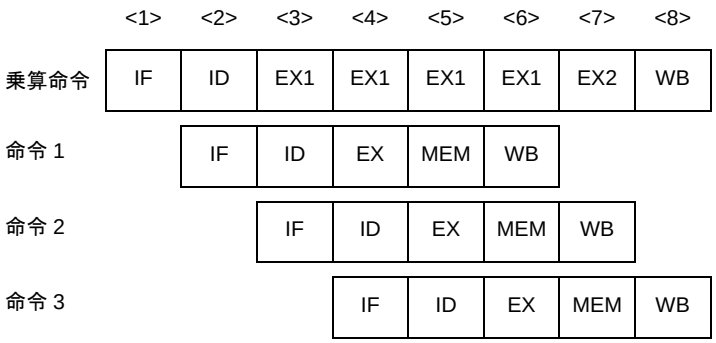
乗算命令（ワード・データ乗算命令）

[対象の命令]

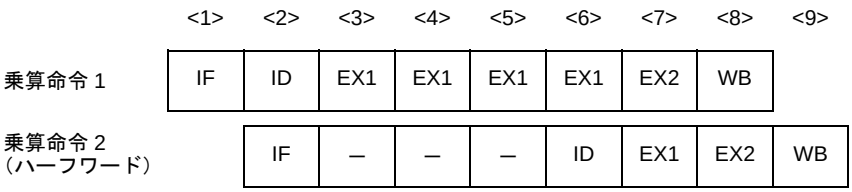
MUL，MULU

[パイプライン]

(1) 続く 3 命令が乗算命令以外の場合

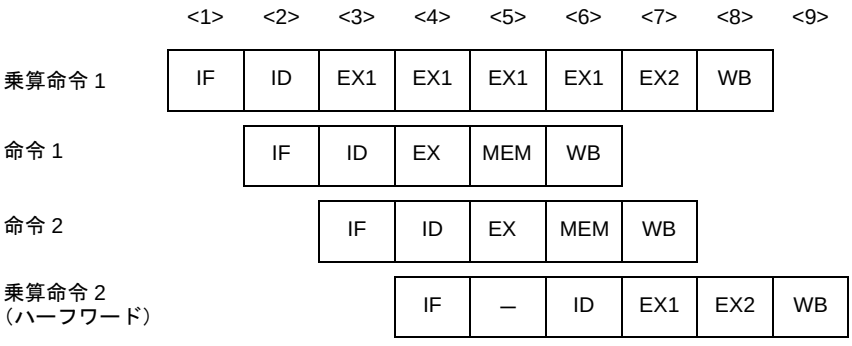


(2) 次命令が乗算命令の場合



備考 ー：待ち合わせのために挿入されるアイドル

(3) 続く 3 命令目が乗算命令の場合



備考 ー：待ち合わせのために挿入されるアイドル

［詳細説明］

パイプラインは IF，ID，EX1（4 ステージ），EX2，WB の 8 ステージです。

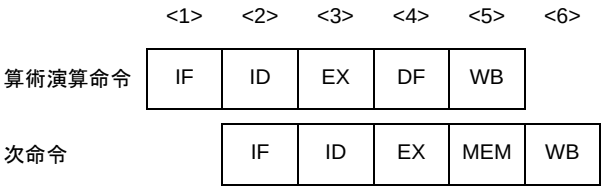
EX ステージは乗算器実行のため 5 クロックかかりますが，EX1 と EX2（通常の EX ステージとは異なる）は独立して動作できます。したがって，乗算命令を繰り返しても命令実行クロック数は，4 となります。ただし，乗算命令の直後に実行結果を使用する命令を配置すると，データの待ち合わせ時間が発生します。

算術演算命令（除算命令／ワード転送命令を除く）

[対象の命令]

ADD，ADDI，CMOV，CMP，MOV，MOVEA，MOVHI，SASF，SETF，SUB，SUBR

[パイプライン]



[詳細説明]

パイプラインは IF，ID，EX，DF，WB の 5 ステージです。

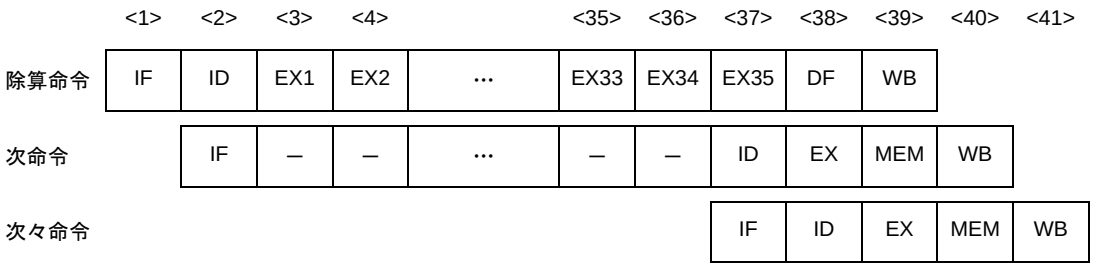
算術演算命令（除算命令）

[対象の命令]

DIV，DIVH，DIVHU，DIVU

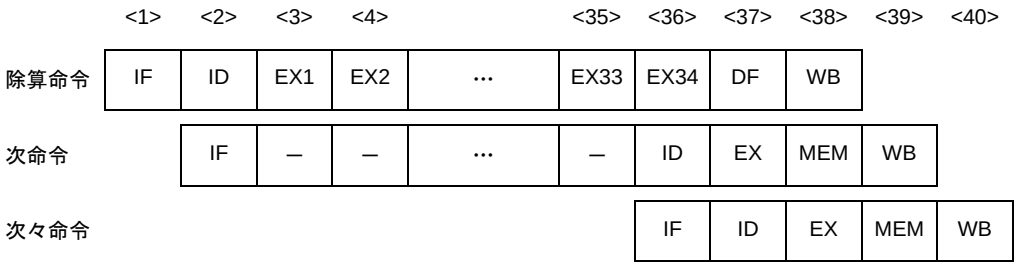
[パイプライン]

（1）DIV，DIVH 命令の場合



備考　ー：待ち合わせのために挿入されるアイドル

（2）DIVHU，DIVU 命令の場合



備考　ー：待ち合わせのために挿入されるアイドル

[詳細説明]

パイプラインは，DIV，DIVH 命令の場合，IF，ID，EX1-EX35（通常の EX ステージ），DF，WB の 39 ステージ，DIVHU，DIVU 命令の場合，IF，ID，EX1-EX34（通常の EX ステージ），DF，WB の 38 ステージです。

[備考]

除算命令実行中に割り込みが発生すると実行を中止し，戻り番地をこの命令の先頭アドレスとして割り込みを処理します。割り込み処理完了後，この命令を再実行します。この場合，汎用レジスタ reg1 と汎用レジスタ reg2 は，この命令実行前の値を保持します。

算術演算命令（ワード転送命令）

[対象の命令]

MOV imm32

[パイプライン]

備考　ー：待ち合わせのために挿入されるアイドル

[詳細説明]

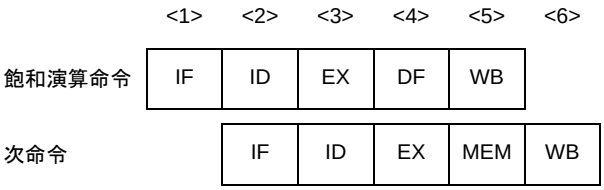
パイプラインは IF，ID，EX1，EX2（通常の EX ステージ），DF，WB の 6 ステージです。

飽和演算命令

[対象の命令]

SATADD，SATSUB，SATSUBI，SATSUBR

[パイプライン]



[詳細説明]

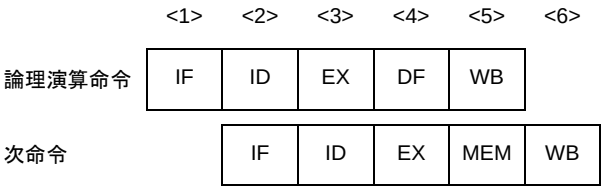
パイプラインは IF，ID，EX，DF，WB の 5 ステージです。

論理演算命令

[対象の命令]

AND，ANDI，BSH，BSW，HSW，NOT，OR，ORI，SAR，SHL，SHR，SXB，SXH，TST，XOR，XORI，ZXB，ZXH

[パイプライン]



[詳細説明]

パイプラインは IF，ID，EX，DF，WB の 5 ステージです。

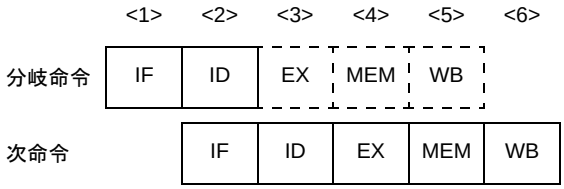
分岐命令（条件分岐命令：BR 命令を除く）

[対象の命令]

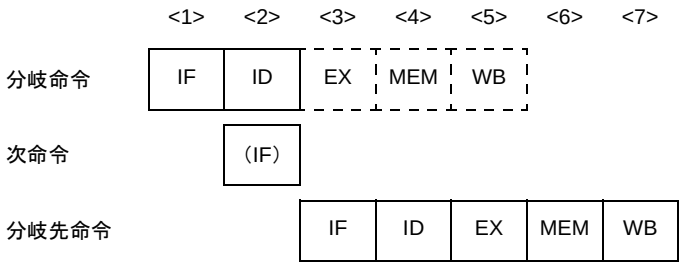
Bcnd 命令

[パイプライン]

(1) 条件が成立しない場合



(2) 条件が成立した場合



備考 (IF)：無効となる命令フェッチ

[詳細説明]

パイプラインは IF，ID，EX，MEM，WB の 5 ステージですが，ID ステージで分岐先が決定するため EX ステージ，MEM ステージ，WB ステージでは何も行いません。

(1) 条件が成立しない場合

分岐命令の命令実行クロック数は，1 となります。

(2) 条件が成立した場合

分岐命令の命令実行クロック数は，2 となります。分岐命令の次命令の IF は無効となります。

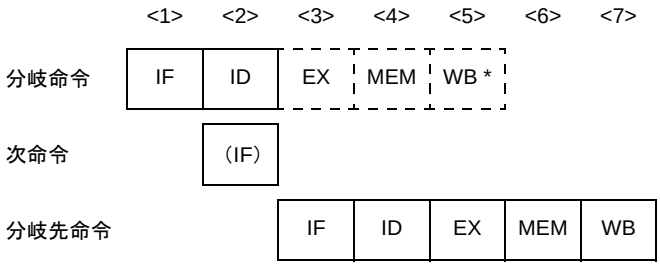
ただし，直前に PSW の内容を書き換える命令があると，フラグ・ハザード発生のために命令実行クロック数は，3 となります。

分岐命令（BR 命令，無条件分岐命令：JMP 命令を除く）

[対象の命令]

BR，JARL，JR

[パイプライン]



備考　（IF）：無効となる命令フェッチ

WB *：BR，JR 命令の場合は何も行われませんが，JARL 命令の場合は復帰 PC の書き込みが行われます。

[詳細説明]

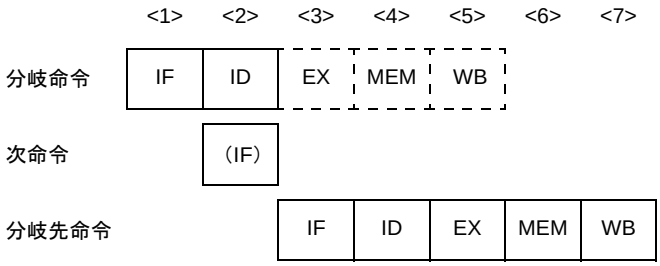
パイプラインは IF，ID，EX，MEM，WB の 5 ステージですが，ID ステージで分岐先が決定するため EX ステージ，MEM ステージ，WB ステージでは何も行いません。ただし，JARL 命令の場合には WB ステージにおいて復帰 PC の書き込みが行われます。また，分岐命令の次命令の IF は無効となります。

分岐命令（JMP 命令）

[対象の命令]

JMP

[パイプライン]



備考　（IF）：無効となる命令フェッチ

[詳細説明]

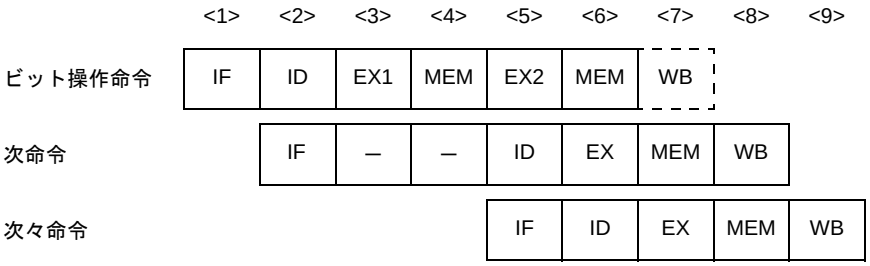
パイプラインは IF，ID，EX，MEM，WB の 5 ステージですが，ID ステージで分岐先が決定するため EX ステージ，MEM ステージ，WB ステージでは何も行いません。

ビット操作命令（CLR1，NOT1，SET1 命令）

［対象の命令］

CLR1，NOT1，SET1

［パイプライン］



備考　―：待ち合わせのために挿入されるアイドル

［詳細説明］

パイプラインは IF，ID，EX1，MEM，EX2（通常のステージ），MEM，WB の 7 ステージですが，レジスタへのデータ書き込みがないので WB ステージでは何も行いません。

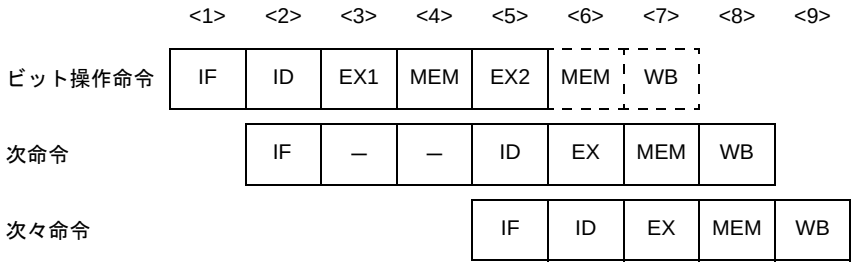
この命令では，メモリ・アクセスがリード・モディファイ・ライトとなり，EX ステージに計 2 クロック，MEM ステージに計 2 サイクルかかります。

ビット操作命令（TST1 命令）

[対象の命令]

TST1

[パイプライン]



備考 ー：待ち合わせのために挿入されるアイドル

[詳細説明]

パイプラインは IF，ID，EX1，MEM，EX2（通常のステージ），MEM，WB の 7 ステージですが，2 回目のメモリへのアクセス，レジスタへのデータ書き込みがないので 2 回目の MEM ステージ，WB ステージでは何も行いません。

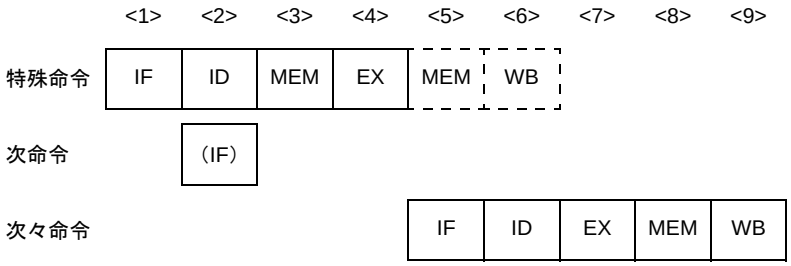
この命令では，計 2 クロックかかります。

特殊命令（CALLT 命令）

［対象の命令］

CALLT

［パイプライン］



備考　（IF）：無効となる命令フェッチ

［詳細説明］

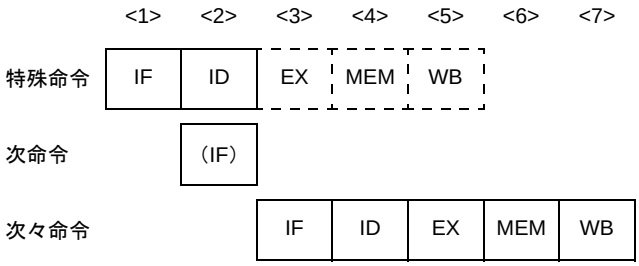
パイプラインは IF，ID，MEM，EX，MEM，WB の 6 ステージでが，2 回目のメモリへのアクセス，レジスタへのデータ書き込みがないので 2 回目の MEM ステージ，WB ステージでは何も行いません。

特殊命令（CTRET 命令）

［対象の命令］

CTRET

［パイプライン］



備考　（IF）：無効となる命令フェッチ

［詳細説明］

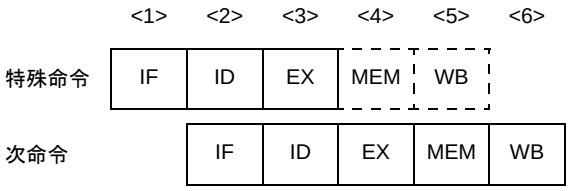
パイプラインは IF，ID，EX，MEM，WB の 5 ステージですが，ID ステージで分岐先が決定するため EX ステージ，MEM ステージ，WB ステージでは何も行いません。

特殊命令（DI，EI 命令）

[対象の命令]

DI，EI

[パイプライン]

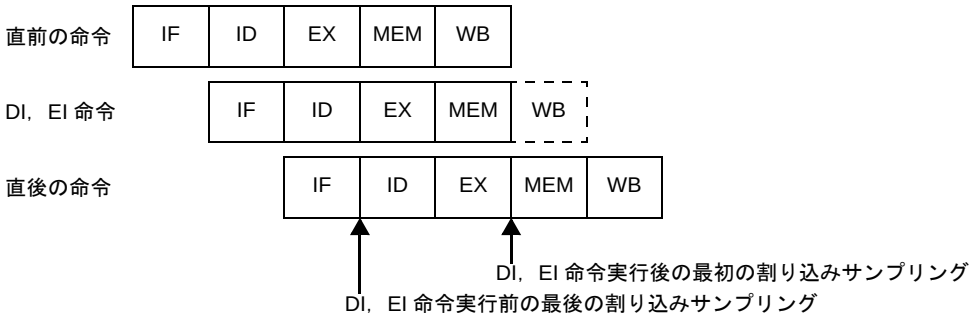


[詳細説明]

パイプラインは IF，ID，EX，MEM，WB の 5 ステージですが，メモリへのアクセス，レジスタへのデータ書き込みがないので MEM ステージ，WB ステージでは何も行いません。

[備考]

DI，EI 命令は，いずれも割り込み要求非サンプル命令です。これらの命令実行時における割り込みサンプリング・タイミングは，次のようになります。



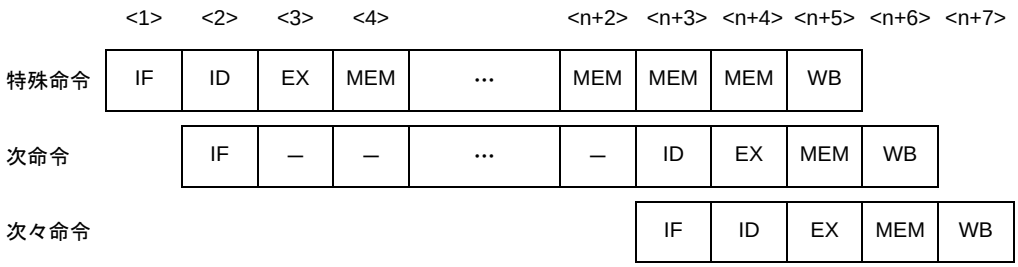
特殊命令（DISPOSE 命令）

[対象の命令]

DISPOSE

[パイプライン]

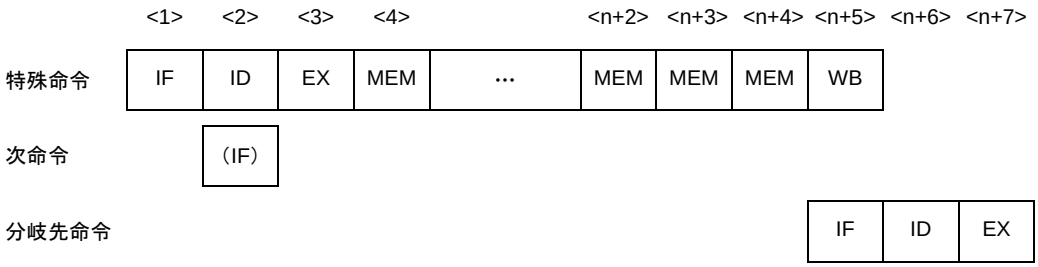
（1）分岐しない場合



備考 ー：待ち合わせのために挿入されるアイドル

n：レジスタ・リスト（list12）で指定されるレジスタの数

（2）分岐する場合



備考 （IF）：無効となる命令フェッチ

ー：待ち合わせのために挿入されるアイドル

n：レジスタ・リスト（list12）で指定されるレジスタの数

[詳細説明]

パイプラインは IF，ID，EX，n + 1 回の MEM，WB の n + 5 ステージ（n：レジスタ・リスト・ナンバ）です。

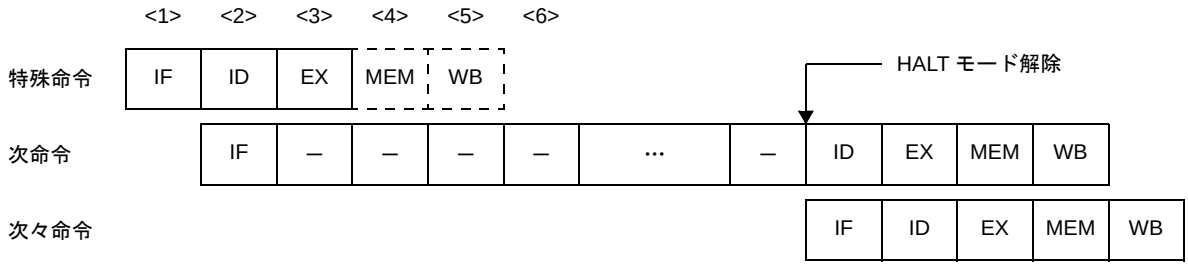
MEM ステージは，n + 1 サイクル必要です。

特殊命令（HALT 命令）

[対象の命令]

HALT

[パイプライン]



備考 ー：待ち合わせのために挿入されるアイドル

[詳細説明]

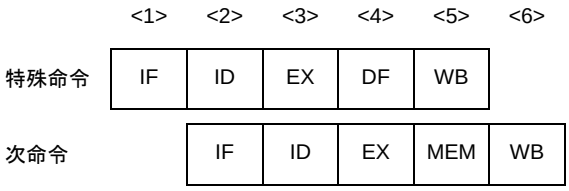
パイプラインは IF，ID，EX，MEM，WB の 5 ステージですが，メモリへのアクセス，レジスタへのデータ書き込みがないので MEM ステージ，WB ステージでは何も行いません。また，次命令では，HALT モードが解除されるまで ID ステージが遅れます。

特殊命令（LDSR，STSR 命令）

［対象の命令］

LDSR，STSR

［パイプライン］



［詳細説明］

パイプラインは IF，ID，EX，DF，WB の 5 ステージです。

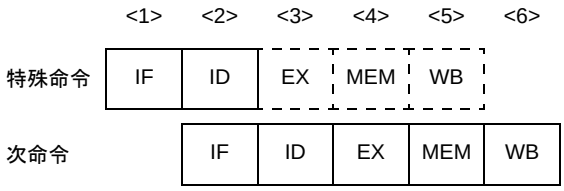
システム・レジスタの EIPC，FEPC を設定する LDSR 命令の直後に，同一レジスタを使用する STSR 命令を配置すると，データの待ち合わせ時間が発生します。

特殊命令（NOP 命令）

[対象の命令]

NOP

[パイプライン]



[詳細説明]

パイプラインは IF，ID，EX，MEM，WB の 5 ステージですが，演算，メモリへのアクセス，レジスタへのデータ書き込みがないので EX ステージ，MEM ステージ，WB ステージでは何も行いません。

[注意]

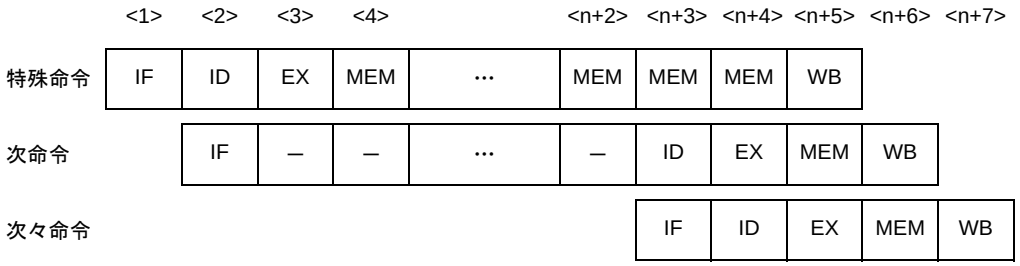
SLD 命令と Bcnd 命令については，ほかの 16 ビット・フォーマットの命令と同時実行される場合があるため注意が必要です。たとえば，SLD 命令と NOP 命令が同時に実行された場合，NOP 命令によるディレイ・タイムの発生が行われない可能性があります。

特殊命令（PREPARE 命令）

[対象の命令]

PREPARE

[パイプライン]



備考　ー：待ち合わせのために挿入されるアイドル
n：レジスタ・リスト（list12）で指定されるレジスタの数

[詳細説明]

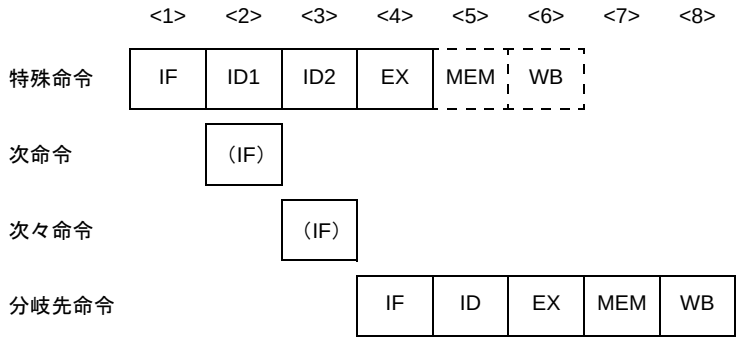
パイプラインは IF，ID，EX，n + 1 回の MEM，WB の n + 5 ステージ（n：レジスタ・リスト・ナンバ）です。
MEM ステージは，n + 1 サイクル必要です。

特殊命令（RETI 命令）

[対象の命令]

RETI

[パイプライン]



備考 （IF）：無効となる命令フェッチ
ID1：レジスタ選択
ID2：EIPC/FEPC 読み込み

[詳細説明]

パイプラインは IF，ID1，ID2，EX，MEM，WB の 6 ステージですが，メモリへのアクセス，レジスタへのデータ書き込みがないので MEM ステージ，WB ステージでは何も行いません。

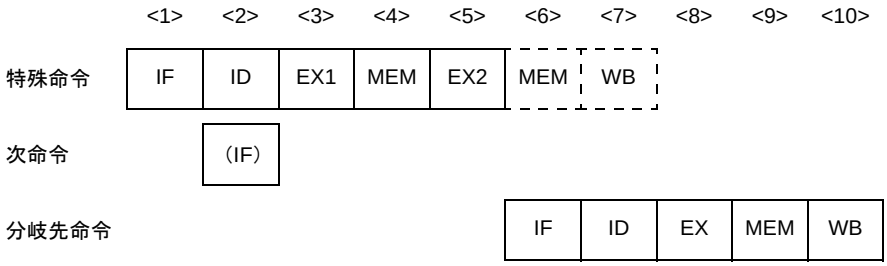
ID ステージには 2 クロックかかります。また，次命令の IF，次々命令の IF は無効となります。

特殊命令（SWITCH 命令）

［対象の命令］

SWITCH

［パイプライン］



備考　（IF）：無効となる命令フェッチ

［詳細説明］

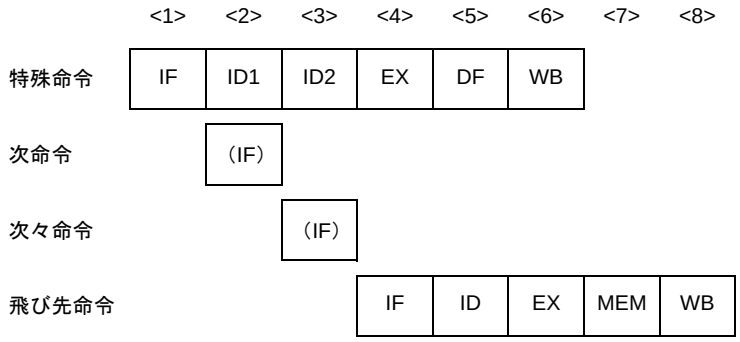
パイプラインは IF，ID，EX1（通常の EX ステージ），MEM，EX2，MEM，WB の 7 ステージですが，2 回目のメモリへのアクセス，レジスタへのデータ書き込みがないので 2 回目の MEM ステージ，WB ステージでは何も行いません。

特殊命令（TRAP 命令）

[対象の命令]

TRAP

[パイプライン]



備考 （IF）：無効となる命令フェッチ
ID1：例外コード（004nH，005nH）検出（n = 0-FH）
ID2：アドレス生成

[詳細説明]

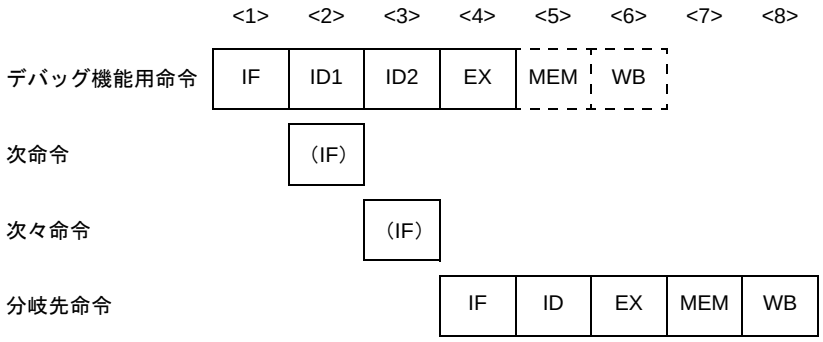
パイプラインは IF，ID1，ID2，EX，DF，WB の 6 ステージです。
ID ステージには 2 クロックかかります。また，次命令の IF，次々命令の IF は無効となります。

デバッグ機能用命令（DBRET 命令）

[対象の命令]

DBRET

[パイプライン]



備考 （IF）：無効となる命令フェッチ
ID1：レジスタ選択
ID2：DBPC 読み込み

[詳細説明]

パイプラインは IF，ID1，ID2，EX，MEM，WB の 6 ステージですが，メモリへのアクセス，レジスタへのデータ書き込みがないので MEM ステージ，WB ステージでは何も行いません。

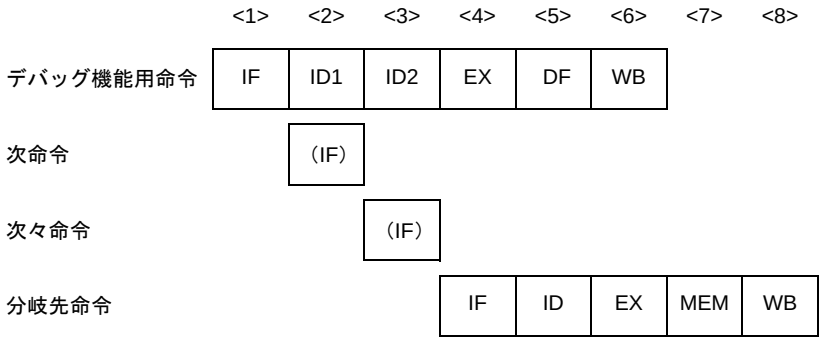
ID ステージには 2 クロックかかります。また，次命令の IF，次々命令の IF は無効となります。

デバッグ機能用命令（DBTRAP 命令）

[対象の命令]

DBTRAP

[パイプライン]



備考 （IF）：無効となる命令フェッチ
ID1：例外コード（0060H）検出
ID2：アドレス生成

[詳細説明]

パイプラインは IF，ID1，ID2，EX，DF，WB の 6 ステージです。
ID ステージには 2 クロックかかります。また，次命令の IF，次々命令の IF は無効となります。

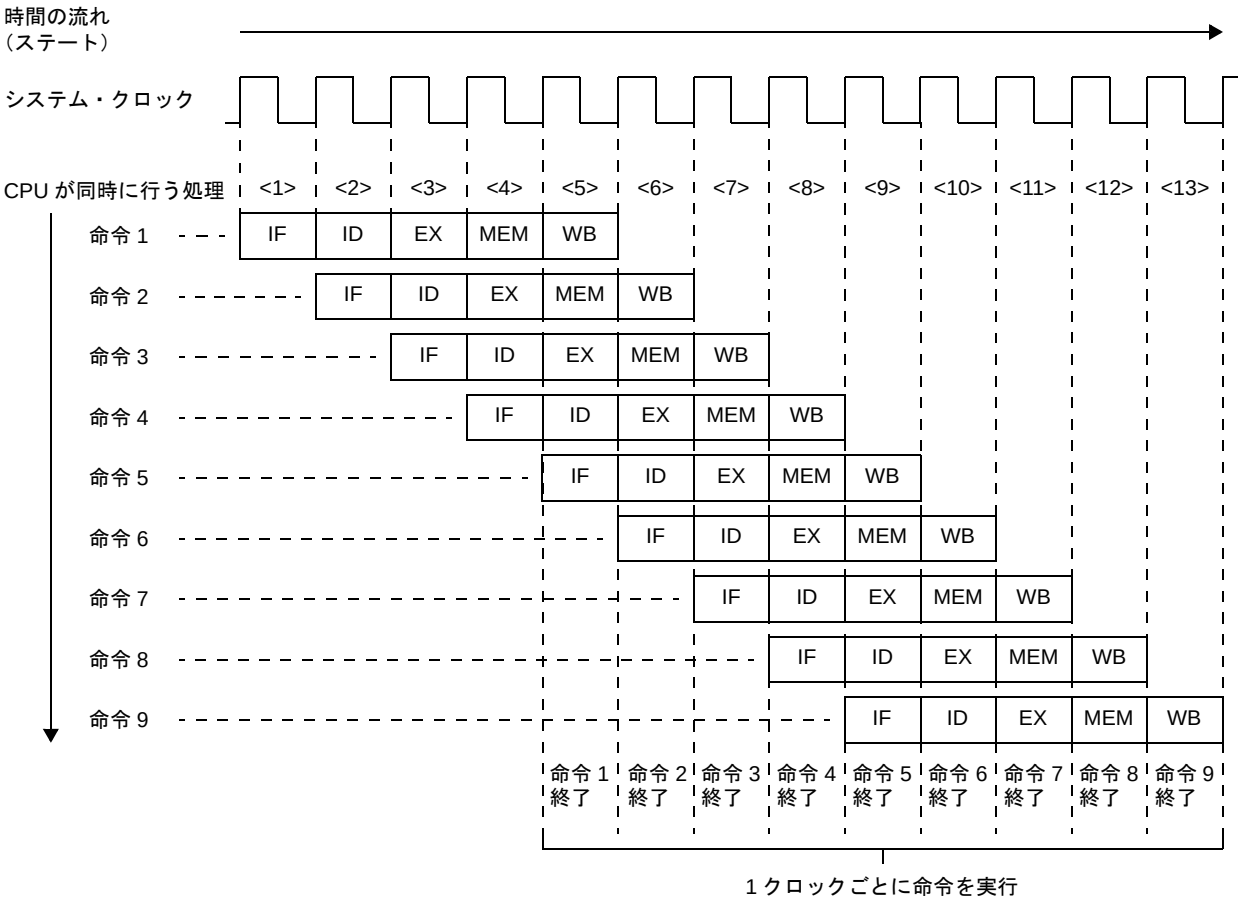
4. 5. 16　パイプライン（V850E1）

V850E1 は，RISC アーキテクチャをベースとした 5 段パイプライン制御により，ほとんどの命令を 1 クロックで実行します。命令実行手順は，通常，IF（インストラクション・フェッチ）から WB（ライトバック）までの 5 ステージで構成されています。

IF（インストラクション・フェッチ）	命令をフェッチし，フェッチ・ポインタをインクリメント
ID（インストラクション・デコード）	命令をデコードし，イミーディエト・データの作成，レジスタの読み出しを行う
EX（実行）	デコードした命令を実行
MEM（メモリ・アクセス）	対象アドレスのメモリをアクセス
WB（ライトバック）	レジスタに実行結果を書き込む

各ステージの実行時間は，命令の種類，アクセス対象となるメモリの種類などによって異なります。
パイプラインの動作例として，標準的な命令を 9 個続けて実行した際の CPU の処理を下図に示します。

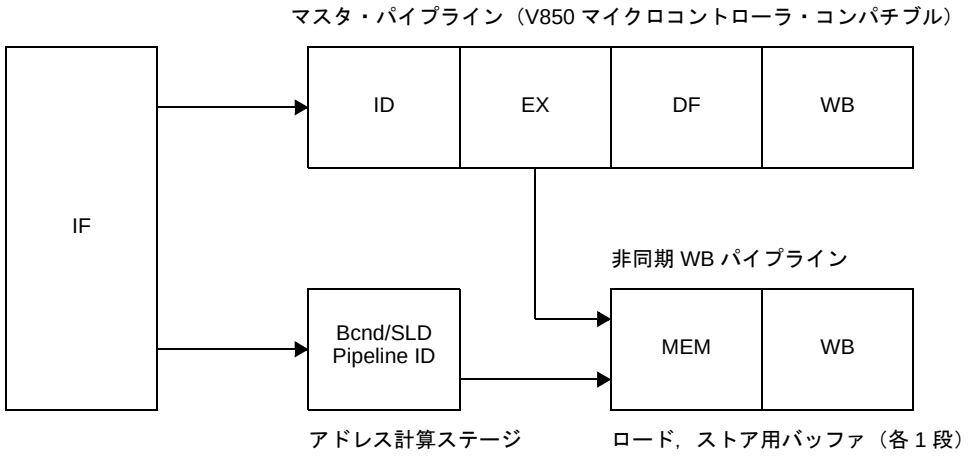
図 4―76　標準的な命令を 9 個続けて実行する例



<1> ～ <13> は CPU のステートを示します。各ステートでは，命令 n の WB（ライトバック），命令 n + 1 の MEM（メモリ・アクセス），命令 n + 2 の EX（実行），命令 n + 3 の ID（インストラクション・デコード），命令 n + 4 の IF（インストラクション・フェッチ）が同時に行われます。標準的な命令では，IF ステージから WB ステージまで 5 クロックかかります。しかし，同時に 5 命令を処理できるため，標準的な命令では平均 1 クロックごとに実行可能です。

V850E1 では，パイプラインの最適化を行うことにより，CPI（Cycle per instruction）を従来の V850 マイクロコントローラよりも向上させています。以下に，V850E1 のパイプライン構成を示します。

図 4―77　パイプライン構成（V850E1）



備考　DF（データ・フェッチ）：WB ステージに実行データを転送

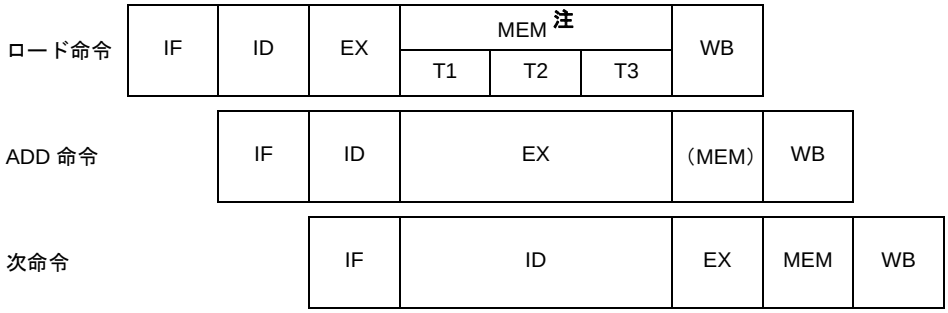
(1) ノンブロッキング・ロード／ストア

外部メモリ・アクセス時にパイプラインが停止することなく，効率的な処理が可能です。

例として，外部メモリに対するロード命令実行後に ADD 命令を実行する場合の V850 マイクロコントローラと V850E1 のパイプライン動作の比較を以下に示します。

(a) V850 マイクロコントローラの場合

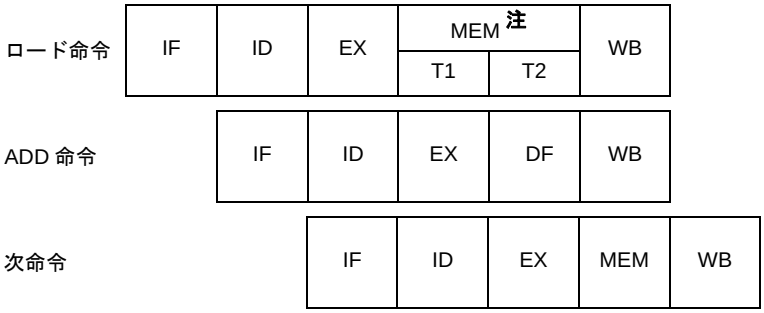
ADD 命令の EX ステージは，本来 1 クロックで実行されます。しかし，直前のロード命令の MEM ステージ実行中，ADD 命令の EX ステージに待ち時間が発生します。これは，パイプライン上の 5 つの命令が，同一内部クロック間に同じステージを実行できないためです。この影響により，ADD 命令の次命令の ID ステージにも待ち時間が発生します。



注 外部メモリに対する基本バス・サイクルは，3 クロックです。

（b）V850E1 の場合

マスタ・パイプラインのほかに，MEM ステージを必要とする命令用に非同期 WB パイプラインを持っています。このため，ロード命令の MEM ステージは，非同期 WB パイプラインで処理されます。下図の例では，ADD 命令はマスタ・パイプラインで処理されるため，パイプラインの待ち時間が発生することなく，効率的に命令を実行できます。



注 外部メモリに対する基本バス・サイクルは，2 クロックです。

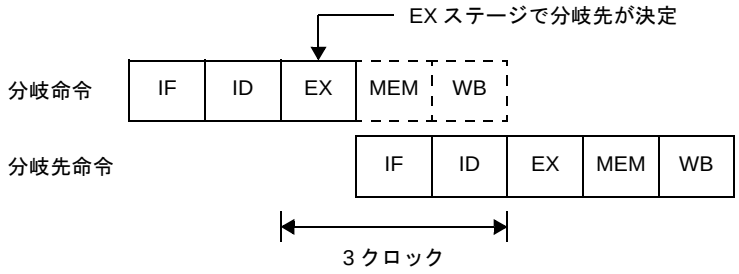
（2）2 クロック分岐

分岐命令実行時は，ID ステージで分岐先が決定します。

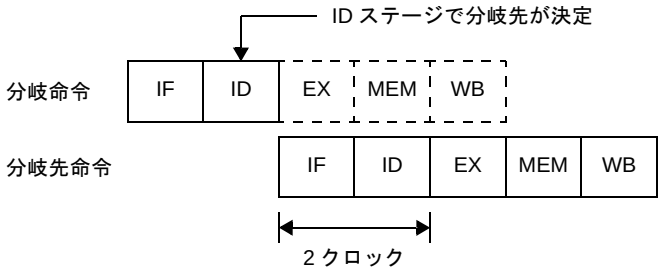
V850 マイクロコントローラの場合，分岐命令実行時の分岐先は，EX ステージ実行後に決定されていましたが，V850E1 では，分岐／ SLD 命令用に追加したアドレス計算ステージにより，ID ステージで分岐先が決定します。このため，V850E1 は，V850 マイクロコントローラと比較して，1 クロック早く分岐先の命令をフェッチすることができます。

V850 マイクロコントローラと V850E1 の分岐命令でのパイプライン動作の比較を以下に示します。

（a）V850 マイクロコントローラの場合

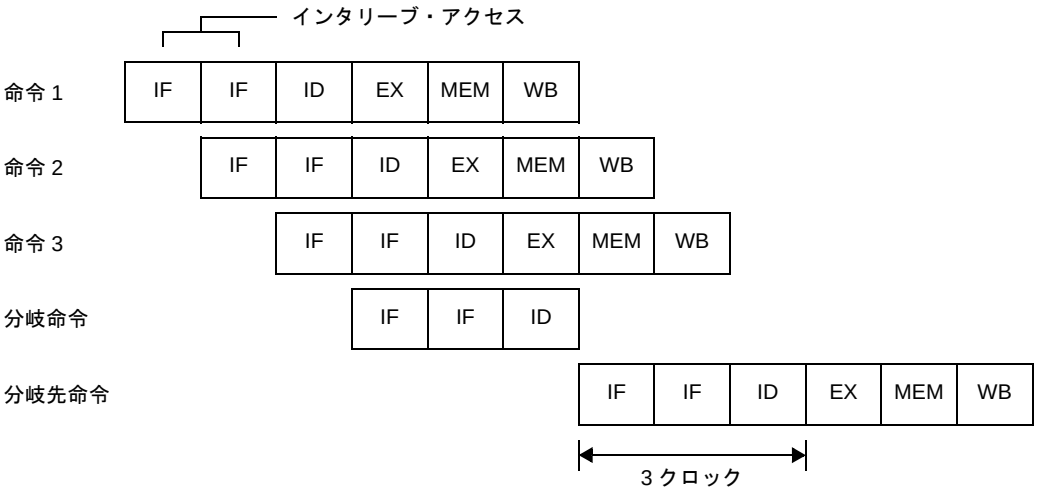


（b）V850E1 の場合



備考 V850E1 タイプ D，V850E1 タイプ E の製品は，内蔵フラッシュ・メモリ，または内蔵マスク ROM に対し，インターリーブ・アクセスを行います。このため，割り込み発生直後の命令フェッチや，分岐命令実行後の命令フェッチに 2 クロック（V850E1 タイプ E の場合は，3 クロック）かかります。したがって，分岐先命令の ID ステージ実行までに 3 クロック（V850E1 タイプ E のの場合は，4 クロック）かかります。

例



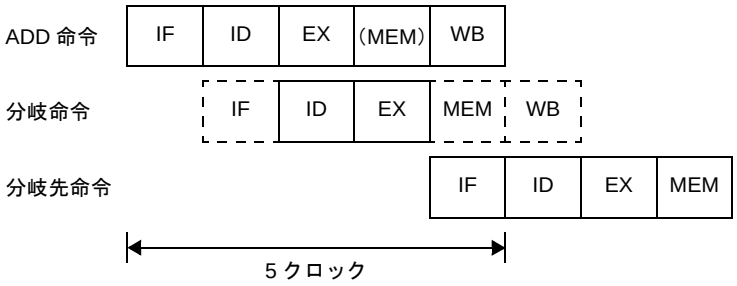
(3) 効率的なパイプライン処理

V850E1 は，マスタ・パイプライン上の ID ステージのほかに，分岐／ SLD 命令用の ID ステージを持っているため，効率的なパイプライン処理が行えます。

例として，ADD 命令の IF ステージで，次分岐命令をフェッチした場合のパイプライン動作例を以下に示します（専用バスに直結された ROM に対する命令フェッチは，32 ビット単位で行われます。以下の ADD 命令，分岐命令はどちらも 16 ビット・フォーマット命令です）。

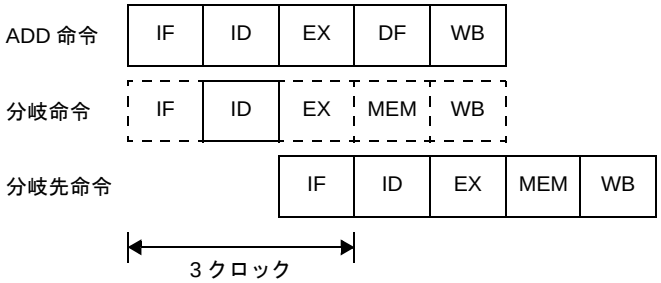
(a) V850 マイクロコントローラの場合

ADD 命令の IF ステージで次分岐命令の命令コードまでフェッチしますが，ADD 命令の ID ステージと分岐命令の ID ステージを同一クロック中に実行できません。そのため，分岐命令のフェッチから分岐先命令のフェッチまで，5 クロックかかります。



(b) V850E1 の場合

マスタ・パイプライン上の ID ステージのほかに，分岐／ SLD 命令用の ID ステージを持っているため，同一クロック中に並行して ADD 命令の ID ステージと分岐命令の ID ステージが実行できます。このため，分岐命令のフェッチ開始から分岐先の命令フェッチ完了まで，3 クロックで実行できます。



備考 SLD 命令と Bcnd 命令については，ほかの 16 ビット・フォーマットの命令と同時実行される場合があるため，注意が必要です。たとえば，SLD 命令と NOP 命令が同時に実行された場合，NOP 命令によるディレイ・タイムの発生が行われない可能性があります。

(4) パイプラインの乱れ

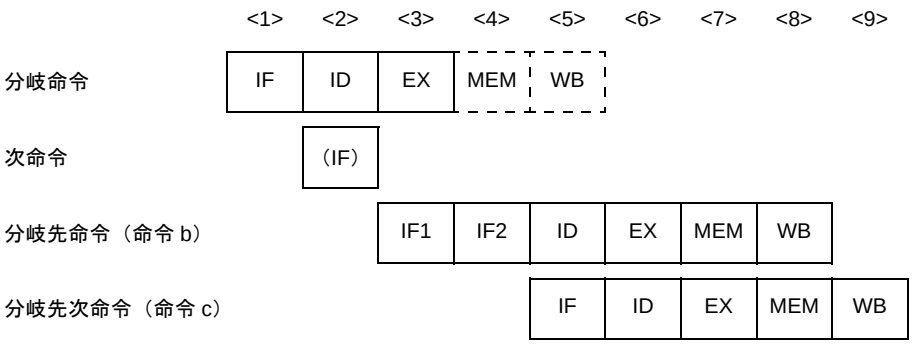
パイプラインは IF（インストラクション・フェッチ）から WB（ライトバック）までの 5 ステージで構成され，基本的にはそれぞれのステージは 1 クロックで処理されますが，場合によってはパイプラインが乱れて，実行クロック数が増加する場合があります。以下に，パイプラインを乱す主な要因を示します。

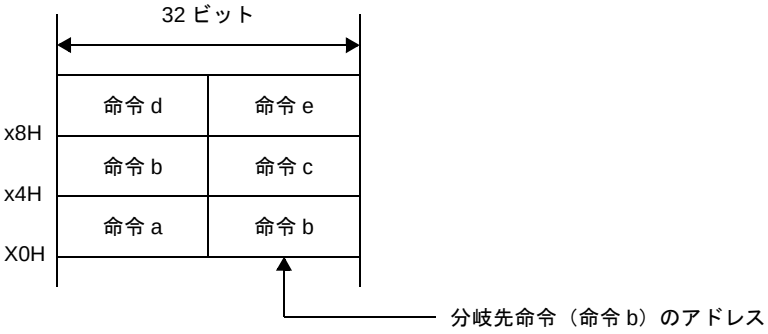
(a) アライン・ハザード

分岐先命令のアドレスがワード・アラインではなく（A1 = 1，A0 = 0），かつ 4 バイト長命令の場合，命令をワード単位に揃えるために IF を 2 回続ける必要があります。これをアライン・ハザードと呼びます。

たとえば，命令 a から命令 e までがアドレス X0H から配置されており，命令 b は 4 バイト長命令で，その他の命令は 2 バイト長命令であるとします。この場合，命令 b は X2H に配置され（A1 = 1，A0 = 0），ワード・アライン（A1 = 0，A0 = 0）となっていません。したがって，この命令 b が分岐先命令となる場合，アライン・ハザードが発生します。アライン・ハザードが発生した場合の分岐命令の実行クロック数は，4 となります。

図 4―78　アライン・ハザードの例





備考　（IF）：無効となる命令フェッチ

IF1：アライン・ハザード時に発生する 1 回目の命令フェッチです。2 バイト長のフェッチで，命令 b の下位アドレス側の 2 バイトがフェッチされます。

IF2：アライン・ハザード時に発生する 2 回目の命令フェッチです。通常の 4 バイト長のフェッチで，命令 b の上位アドレス側の 2 バイトと命令 c（2 バイト長）がフェッチされます。

アライン・ハザードは，次のような対処によって回避が可能で，命令実行速度の向上が図れます。

- 分岐先命令に 2 バイト長命令を使用する
- 分岐先命令にワード境界（A1 = 0，A0 = 0）に配置した 4 バイト長命令を使用する

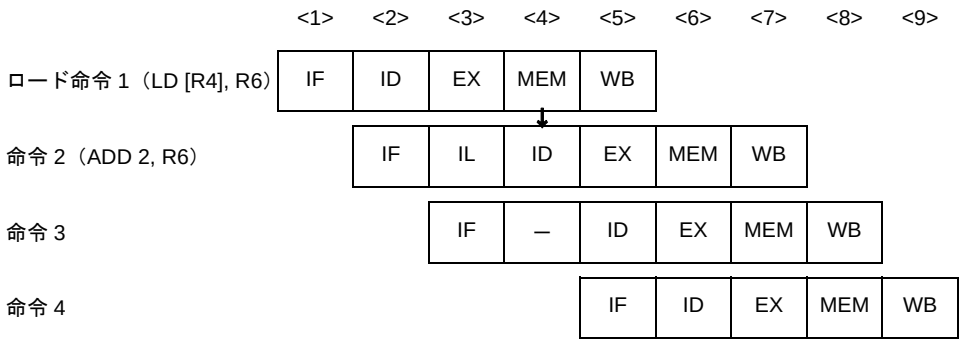
(b) ロード命令実行結果の参照

ロード命令（LD，SLD）では，MEM ステージで読み出されたデータの格納が WB ステージで行われます。したがって，ロード命令の直後の命令で同一のレジスタの内容を使用する場合，ロード命令がレジスタの使用を終えるまで，直後の命令はレジスタの使用を遅らせる必要があります。これをハザードと呼びます。V850E1 は，このハザードを CPU で自動的に対処するインタロック機能を持っており，次命令の ID ステージを遅らせます。

また，V850E1 は，MEM ステージで読み出したデータを次命令の ID ステージで使用できるように，ショート・パスを持っています。このショート・パスによって，ロード命令によって MEM ステージでデータを読み出すことと，このデータを次命令の ID ステージで使用することを，同一タイミングで行うことができます。

以上のことにより，結果を直後の命令で使用する場合，ロード命令の実行クロック数は，2 になります。

図 4―79 ロード命令実行結果の参照例



備考 IL：インタロック機能によりデータの待ち合わせのために挿入されるアイドル
―：待ち合わせのために挿入されるアイドル
↓：ショート・パス

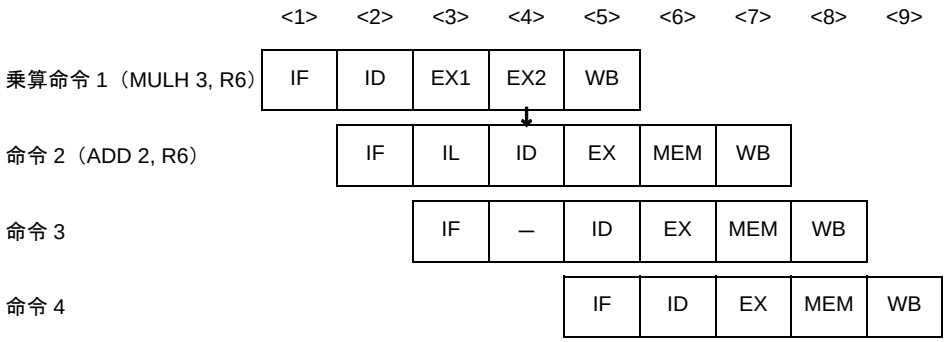
上図のように，ロード命令の直後にその結果を使用する命令を配置すると，インタロック機能によるデータの待ち合わせ時間が発生し，実行速度が低下します。ロード命令の結果を使用する命令は，ロード命令の 2 命令以後に配置することにより，実行速度の低下を防げます。

(c) 乗算命令実行結果の参照

乗算命令では，乗算結果のレジスタへの格納が WB ステージで行われます。したがって，乗算命令の直後の命令で同一レジスタの内容を使用する場合，乗算命令がレジスタの使用を終えるまで，直後の命令はレジスタの使用を遅らせる必要があります（ハザードの発生）。

V850E1 では，インタロック機能により直後の命令の ID ステージを遅らせます。また，ショート・パスにより，乗算命令の EX2 ステージと，この演算結果を直後の命令の ID ステージで使用することが，同一タイミングで行えます。

図 4―80　乗算命令実行結果の参照例



備考　IL：インタロック機能によりデータの待ち合わせのために挿入されるアイドル
　　―：待ち合わせのために挿入されるアイドル
　　↓：ショート・パス

上図のように，乗算命令の直後にその結果を使用する命令を配置すると，インタロック機能によるデータの待ち合わせ時間が発生し，実行速度が低下します。乗算命令の結果を使用する命令は，乗算命令の 2 命令以後に配置することにより，実行速度の低下を防げます。

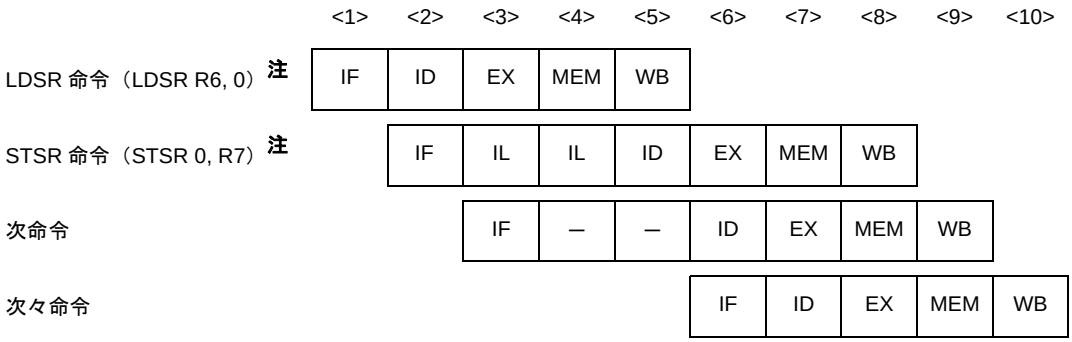
（d）EIPC，FEPC を対象とする LDSR 命令実行結果の参照

LDSR 命令によって，システム・レジスタの EIPC，FEPC のデータ設定を行い，直後に STSR 命令で同一システム・レジスタの参照を行う場合，LDSR 命令のシステム・レジスタ設定が終わるまで，直後の STSR 命令はシステム・レジスタの使用が遅れます（ハザードの発生）。

V850E1 では，インタロック機能により，直後の STSR 命令の ID ステージを遅らせます。

以上のことより，EIPC，FEPC の LDSR 命令実行結果を直後の STSR 命令で参照する場合，LDSR 命令の実行クロック数は，3 になります。

図 4―81　EIPC，FEPC を対象とする LDSR 命令実行結果の参照例



注　LDSR，STSR 命令で使用しているシステム・レジスタ番号の 0 は EIPC を表します。

備考　IL：インタロック機能によりデータの待ち合わせのために挿入されるアイドル
　　―：待ち合わせのために挿入されるアイドル

上図のように，EIPC，または FEPC をオペランドとする LDSR 命令の直後に，STSR 命令によってその結果を参照すると，インタロック機能によるデータの待ち合わせ時間が発生し，実行速度が低下します。LDSR 命令の結果を参照する STSR 命令は，LDSR 命令の 3 命令以後に配置することにより，実行速度の低下を防げます。

(e) プログラム作成時の注意点

プログラムを作成する場合，次のことに注意するとパイプラインが乱れず，命令実行速度が向上します。

- ロード命令（LD，SLD）の結果を使用する命令は，ロード命令の 2 命令以後に配置する。
- 乗算命令の結果を使用する命令は，乗算命令の 2 命令以後に配置する。
- LDSR 命令による EIPC，または FEPC への設定結果を STSR 命令により読み出す場合は，LDSR 命令の 3 命令以後に STSR 命令を配置する。
- 分岐先の最初の命令は，2 バイト長命令か，またはワード境界に配置された 4 バイト長命令を使用する。

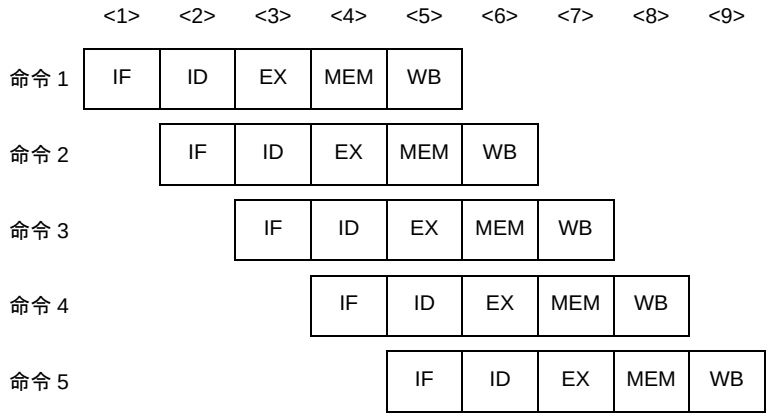
(5) パイプラインに関する補足事項

(a) ハーバード・アーキテクチャ

V850E1 では，ハーバード・アーキテクチャを採用しており，内蔵 ROM からの命令フェッチ用のパスと，内蔵 RAM へのメモリ・アクセス用のパスが独立して動作します。これにより，IF ステージと MEM ステージのバス・アービトレーションの競合が発生せず，パイプラインがスムーズに流れます。

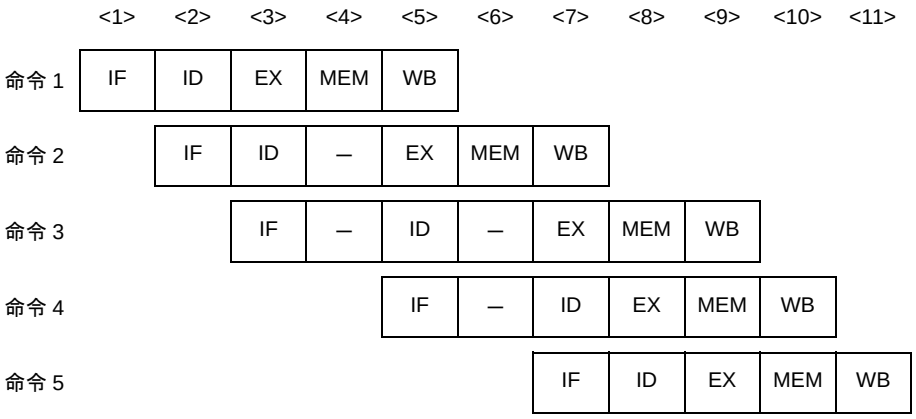
- V850E1（ハーバード・アーキテクチャ）の場合

命令 1 の MEM と命令 4 の IF，および命令 2 の MEM と命令 5 の IF が同時に実行でき，パイプラインが乱れません。



- 非ハーバード・アーキテクチャの場合

命令 1 の MEM と命令 4 の IF，および命令 2 の MEM と命令 5 の IF が競合するため，バスの待ち合わせが発生し，パイプラインが乱れ実行速度が低下します。



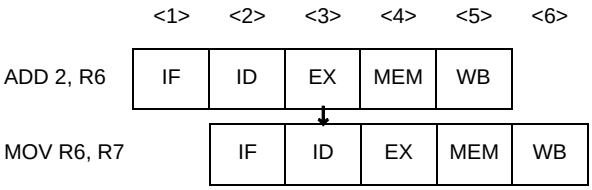
備考 —：待ち合わせのために挿入されるアイドル

（b）ショート・パス

V850E1 はショート・パスを内蔵しているため，前命令の WB（ライトバック）が終了する前に，後続の命令がその結果を使用できます。

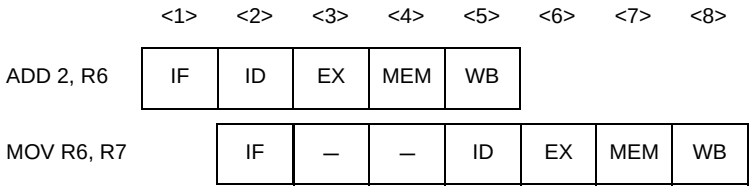
例 1. 算術演算命令，論理演算命令の実行結果を直後の命令で使用する場合：V850E1（ショート・パス内蔵）の場合

前命令の WB を待たず実行結果が出た時点（EX ステージ）で，直後の命令の ID を処理できます。



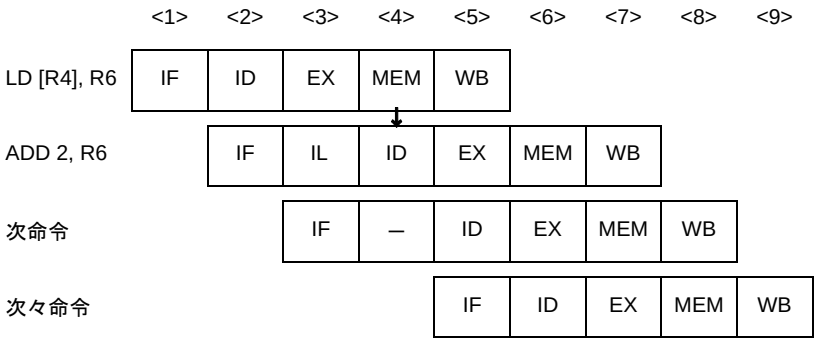
2. 算術演算命令，論理演算命令の実行結果を直後の命令で使用する場合：ショート・パスがない場合

前命令の WB まで，直後の ID が遅れます。



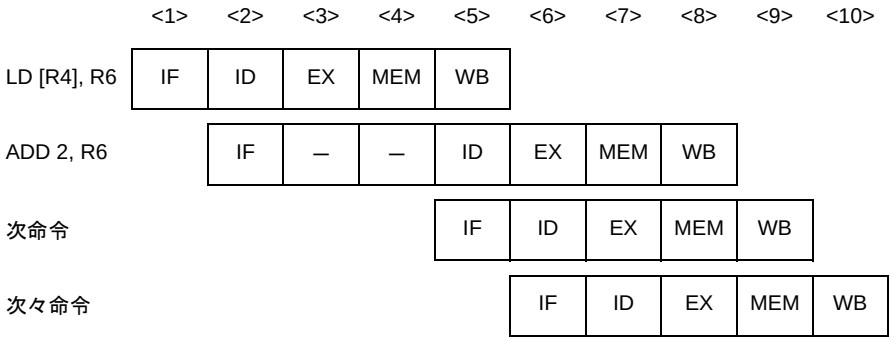
3. ロード命令によりメモリから読み出したデータを，直後の命令で使用する場合：V850E1（ショート・パス内蔵）の場合

前命令の WB を待たず実行結果が出た時点（MEM ステージ）で，直後の命令の ID を処理できます。



4. ロード命令によりメモリから読み出したデータを，直後の命令で使用する場合：ショート・パスがない場合

前命令の WB まで，直後の ID が遅れます。



（6）各命令実行時のパイプラインの流れ

以下に，各命令実行時のパイプラインの流れについて説明します。

パイプライン処理のため，メモリや I/O のライト・サイクルが発生する時点で，CPU は後続の命令をすでに実行しています。そのため，I/O 操作や割り込み要求マスクの反映は，次命令発行（ID ステージ）に対して遅れて実行されます。

（a）タイプ A，B，C の場合

NPB に専用の割り込みコントローラ（INTC）が接続されている場合は，INTC へのアクセスを CPU が検出して割り込み要求のマスク処理を行うため，直後の命令からマスカブル割り込みの受け付けを禁止します。

（b）タイプ D，E，F の場合

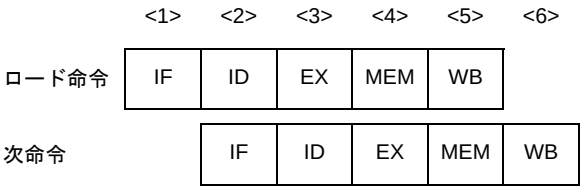
内蔵 INTC へのアクセスを CPU が検出（ID ステージ）して割り込み要求のマスク処理を行うため，割り込みマスク操作を行う場合，直後の命令からマスカブル割り込みの受け付けを禁止します。

ロード命令（LD 命令）

[対象の命令]

LD.B，LD.BU，LD.H，LD.HU，LD.W

[パイプライン]



[詳細説明]

パイプラインは IF，ID，EX，MEM，WB の 5 ステージです。

LD 命令の直後に，実行結果を使用する命令を配置すると，データの待ち合わせ時間が発生します。

備考 ノンブロッキング制御のため，MEM ステージの間にバス・サイクルが完了している保証はありません。ただし，周辺 I/O 領域へのアクセスはブロッキング制御となるため，MEM ステージでバス・サイクルの完了を待ち合わせます。

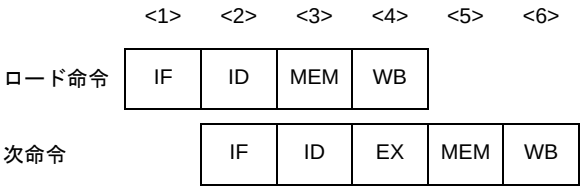
タイプ A，B，C の場合，プログラマブル周辺 I/O 領域へのアクセスは，ノンブロッキング制御です。

ロード命令（SLD 命令）

[対象の命令]

SLD.B，SLD.BU，SLD.H，SLD.HU，SLD.W

[パイプライン]



[詳細説明]

パイプラインは IF，ID，MEM，WB の 4 ステージです。

SLD 命令の直後に，実行結果を使用する命令を配置すると，データの待ち合わせ時間が発生します。

備考 ノンブロッキング制御のため，MEM ステージの間にバス・サイクルが完了している保証はありません。ただし，周辺 I/O 領域へのアクセスはブロッキング制御となるため，MEM ステージでバス・サイクルの完了を待ち合わせます。

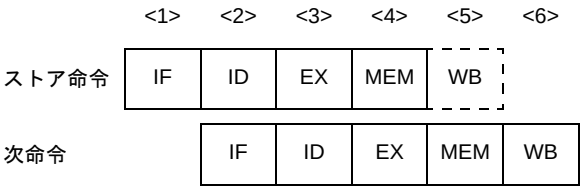
タイプ A，B，C の場合，プログラマブル周辺 I/O 領域へのアクセスは，ノンブロッキング制御です。

ストア命令

[対象の命令]

SST.B，SST.H，SST.W，ST.B，ST.H，ST.W

[パイプライン]



[詳細説明]

パイプラインは IF，ID，EX，MEM，WB の 5 ステージですが，レジスタへのデータの書き込みがないので WB ステージでは何も行いません。

備考 ノンブロッキング制御のため，MEM ステージの間にバス・サイクルが完了している保証はありません。ただし，周辺 I/O 領域へのアクセスはブロッキング制御となるため，MEM ステージでバス・サイクルの完了を待ち合わせます。
タイプ A，B，C の場合，プログラマブル周辺 I/O 領域へのアクセスは，ノンブロッキング制御です。

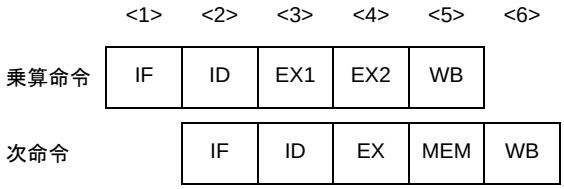
乗算命令

[対象の命令]

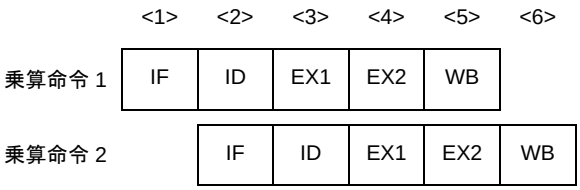
MUL，MULH，MULHI，MULU

[パイプライン]

(1) 次命令が乗算命令以外の場合



(2) 次命令が乗算命令の場合



[詳細説明]

パイプラインは IF，ID，EX1，EX2，WB の 5 ステージです。

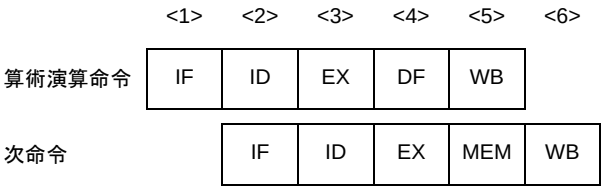
EX ステージは乗算器実行のため 2 クロックかかりますが，EX1 と EX2（通常の EX ステージとは異なります）は独立して動作できます。したがって，乗算命令を繰り返しても命令実行クロック数は，1 となります。ただし，乗算命令の直後に実行結果を使用する命令を配置すると，データの待ち合わせ時間が発生します。

算術演算命令（除算命令／ワード転送命令を除く）

[対象の命令]

ADD，ADDI，CMOV，CMP，MOV，MOVEA，MOVHI，SASF，SETF，SUB，SUBR

[パイプライン]



[詳細説明]

パイプラインは IF，ID，EX，DF，WB の 5 ステージです。

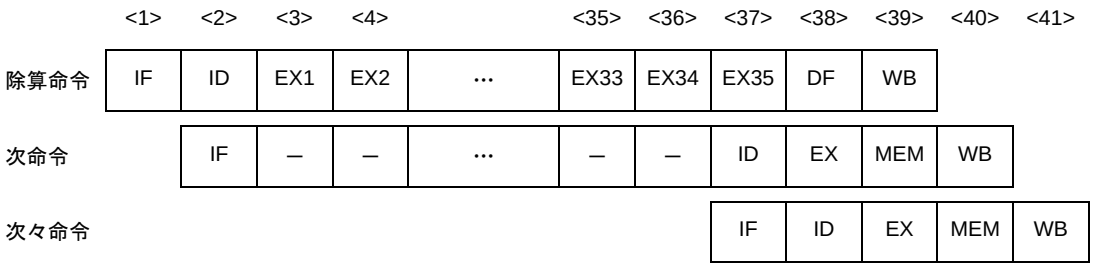
算術演算命令（除算命令）

[対象の命令]

DIV，DIVH，DIVHU，DIVU

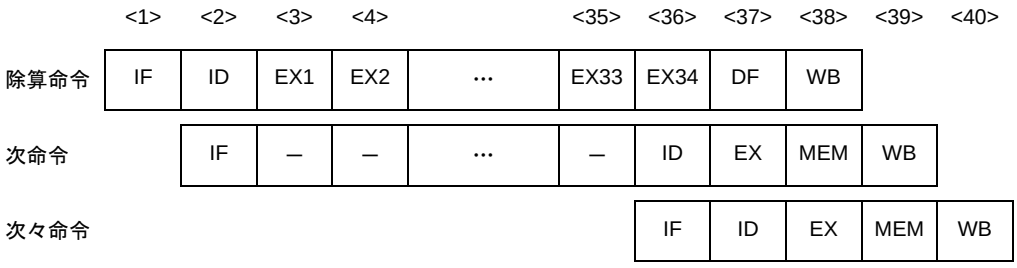
[パイプライン]

（1）DIV，DIVH 命令の場合



備考　ー：待ち合わせのために挿入されるアイドル

（2）DIVHU，DIVU 命令の場合



備考　ー：待ち合わせのために挿入されるアイドル

[詳細説明]

パイプラインは，DIV，DIVH 命令の場合，IF，ID，EX1-EX35（通常の EX ステージ），DF，WB の 39 ステージ，DIVHU，DIVU 命令の場合，ID，ID，EX1-EX34（通常の EX ステージ），DF，WB の 38 ステージです。

[備考]

除算命令実行中に割り込みが発生すると実行を中止し，戻り番地をこの命令の先頭アドレスとして割り込みを処理します。割り込み処理完了後，この命令を再実行します。この場合，汎用レジスタ reg1 と汎用レジスタ reg2 は，この命令実行前の値を保持します。

算術演算命令（ワード転送命令）

[対象の命令]

MOV imm32

[パイプライン]

備考　ー：待ち合わせのために挿入されるアイドル

[詳細説明]

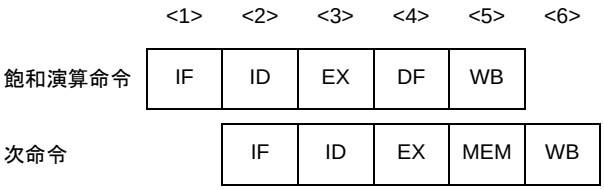
パイプラインは IF，ID，EX1，EX2（通常の EX ステージ），DF，WB の 6 ステージです。

飽和演算命令

[対象の命令]

SATADD，SATSUB，SATSUBI，SATSUBR

[パイプライン]



[詳細説明]

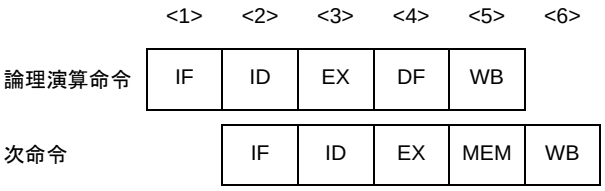
パイプラインは IF，ID，EX，DF，WB の 5 ステージです。

論理演算命令

[対象の命令]

AND，ANDI，BSH，BSW，HSW，NOT，OR，ORI，SAR，SHL，SHR，SXB，SXH，TST，XOR，XORI，ZXB，ZXH

[パイプライン]



[詳細説明]

パイプラインは IF，ID，EX，DF，WB の 5 ステージです。

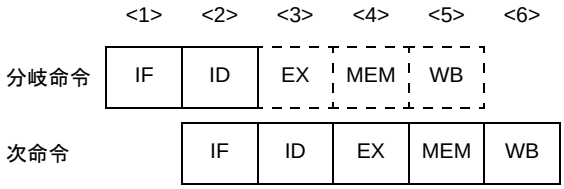
分岐命令（条件分岐命令：BR 命令を除く）

[対象の命令]

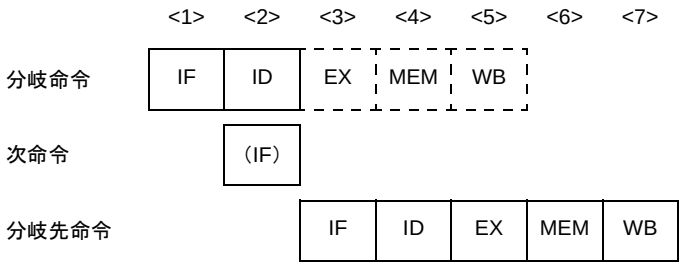
Bcnd 命令

[パイプライン]

(1) 条件が成立しない場合



(2) 条件が成立した場合



備考　(IF)：無効となる命令フェッチ

[詳細説明]

パイプラインは IF，ID，EX，MEM，WB の 5 ステージですが，ID ステージで分岐先が決定するため EX ステージ，MEM ステージ，WB ステージでは何も行いません。

(1) 条件が成立しない場合

分岐命令の命令実行クロック数は，1 となります。

(2) 条件が成立した場合

分岐命令の命令実行クロック数は，2 となります。分岐命令の次命令の IF は無効となります。

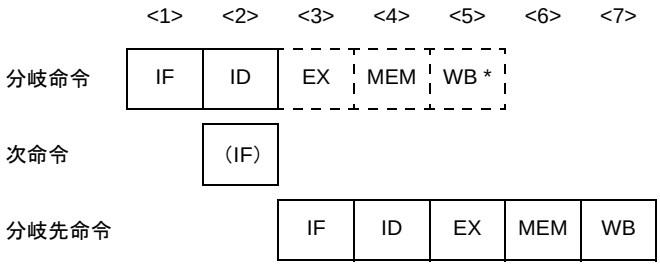
ただし，直前に PSW の内容を書き換える命令があると，フラグ・ハザード発生のために命令実行クロック数は，3 となります。

分岐命令（BR 命令，無条件分岐命令：JMP 命令を除く）

[対象の命令]

BR，JARL，JR

[パイプライン]



備考　（IF）：無効となる命令フェッチ

WB *：BR，JR 命令の場合は何も行われませんが，JARL 命令の場合は復帰 PC の書き込みが行われます。

[詳細説明]

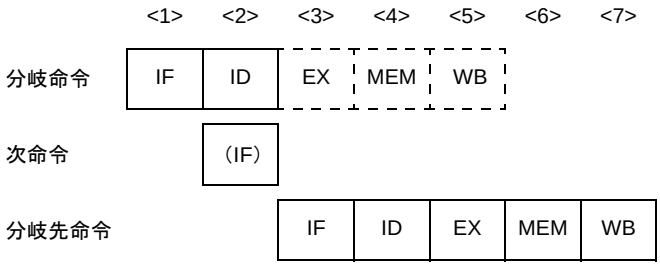
パイプラインは IF，ID，EX，MEM，WB の 5 ステージですが，ID ステージで分岐先が決定するため EX ステージ，MEM ステージ，WB ステージでは何も行いません。ただし，JARL 命令の場合には WB ステージにおいて復帰 PC の書き込みが行われます。また，分岐命令の次命令の IF は無効となります。

分岐命令（JMP 命令）

[対象の命令]

JMP

[パイプライン]



備考　（IF）：無効となる命令フェッチ

[詳細説明]

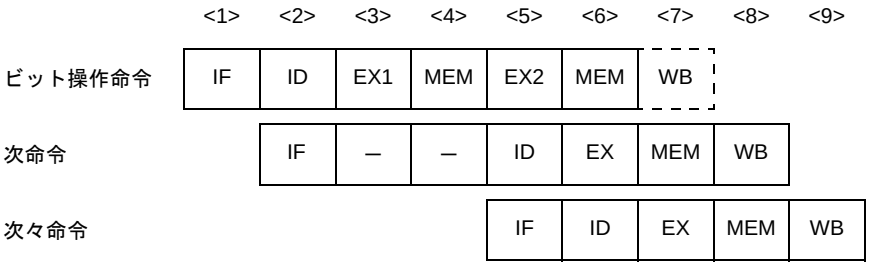
パイプラインは IF，ID，EX，MEM，WB の 5 ステージですが，ID ステージで分岐先が決定するため EX ステージ，MEM ステージ，WB ステージでは何も行いません。

ビット操作命令（CLR1，NOT1，SET1 命令）

［対象の命令］

CLR1，NOT1，SET1

［パイプライン］



備考 ―：待ち合わせのために挿入されるアイドル

［詳細説明］

パイプラインは IF，ID，EX1，MEM，EX2（通常のステージ），MEM，WB の 7 ステージですが，レジスタへのデータ書き込みがないので WB ステージでは何も行いません。

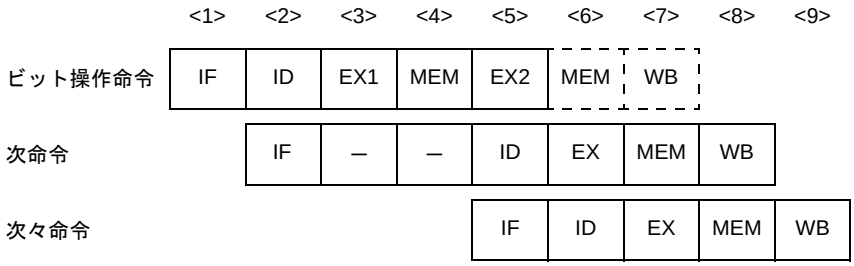
この命令では，メモリ・アクセスがリード・モディファイ・ライトとなり，EX ステージに計 2 クロック，MEM ステージに計 2 サイクルかかります。

ビット操作命令（TST1 命令）

［対象の命令］

TST1

［パイプライン］



備考　ー：待ち合わせのために挿入されるアイドル

［詳細説明］

パイプラインは IF，ID，EX1，MEM，EX2（通常のステージ），MEM，WB の 7 ステージですが，2 回目のメモリ・アクセス，レジスタへのデータ書き込みがないので 2 回目の MEM ステージ，WB ステージでは何も行いません。

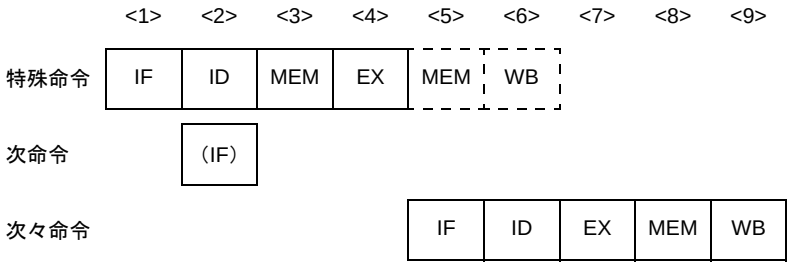
この命令では，計 2 クロックかかります。

特殊命令（CALLT 命令）

[対象の命令]

CALLT

[パイプライン]



備考 （IF）：無効となる命令フェッチ

[詳細説明]

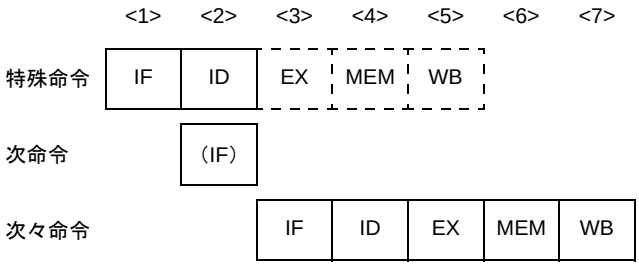
パイプラインは IF，ID，MEM，EX，MEM，WB の 6 ステージですが，2 回目のメモリへのアクセス，レジスタへのデータ書き込みがないので 2 回目の MEM ステージ，WB ステージでは何も行いません。

特殊命令（CTRET 命令）

［対象の命令］

CTRET

［パイプライン］



備考　（IF）：無効となる命令フェッチ

［詳細説明］

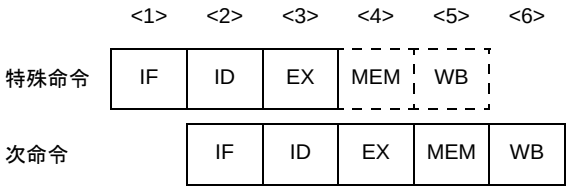
パイプラインは IF，ID，EX，MEM，WB の 5 ステージですが，ID ステージで分岐先が決定するため EX ステージ，MEM ステージ，WB ステージでは何も行いません。

特殊命令（DI，EI 命令）

[対象の命令]

DI，EI

[パイプライン]

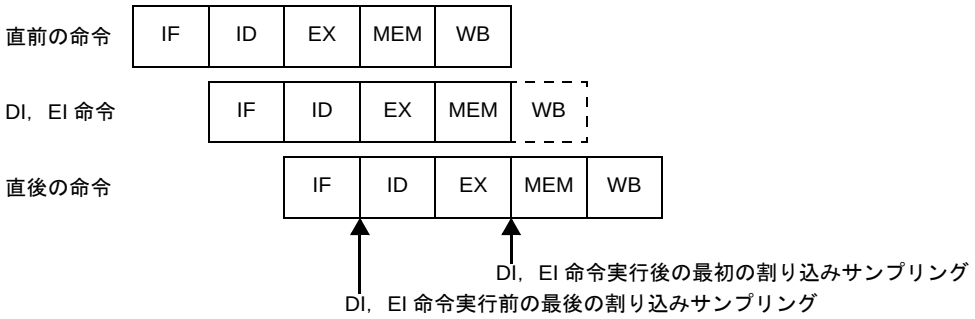


[詳細説明]

パイプラインは IF，ID，EX，MEM，WB の 5 ステージですが，メモリへのアクセス，レジスタへのデータ書き込みがないので MEM ステージ，WB ステージでは何も行いません。

[備考]

DI，EI 命令は，いずれも割り込み要求非サンプル命令です。これらの命令実行時における割り込みサンプリング・タイミングは，次のようになります。



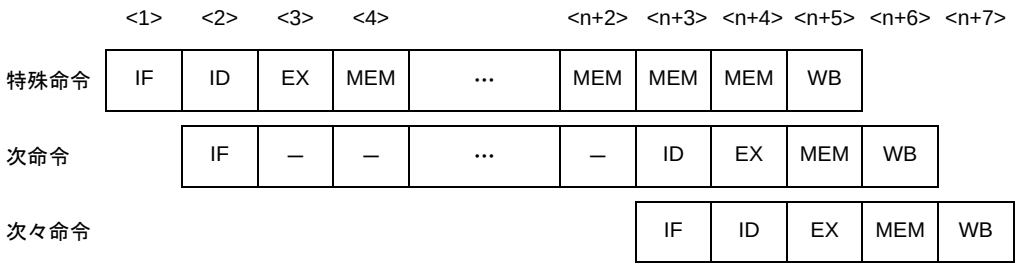
特殊命令（DISPOSE 命令）

[対象の命令]

DISPOSE

[パイプライン]

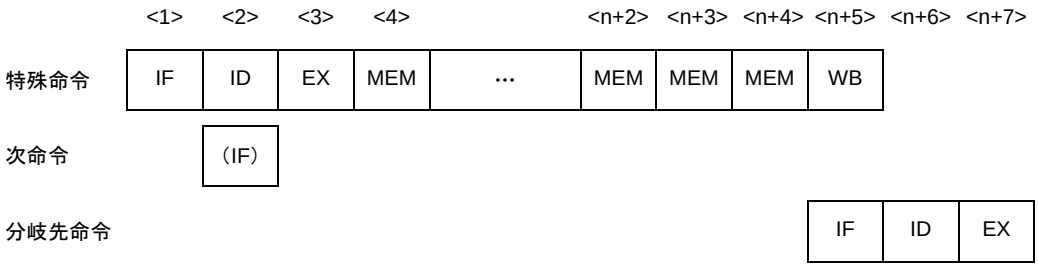
（1）分岐しない場合



備考 ー：待ち合わせのために挿入されるアイドル

n：レジスタ・リスト（list12）で指定されるレジスタの数

（2）分岐する場合



備考 （IF）：無効となる命令フェッチ

ー：待ち合わせのために挿入されるアイドル

n：レジスタ・リスト（list12）で指定されるレジスタの数

[詳細説明]

パイプラインは IF，ID，EX，n + 1 回の MEM，WB の n + 5 ステージ（n：レジスタ・リスト・ナンバ）です。

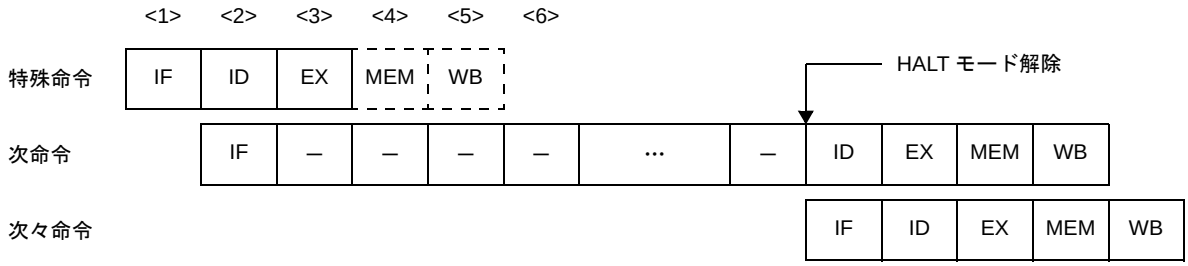
MEM ステージは，n + 1 サイクル必要です。

特殊命令（HALT 命令）

[対象の命令]

HALT

[パイプライン]



備考 ー：待ち合わせのために挿入されるアイドル

[詳細説明]

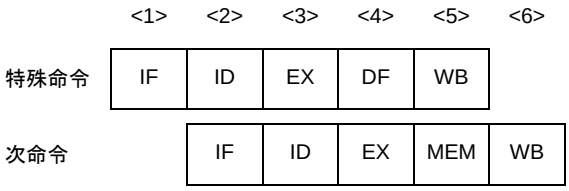
パイプラインは IF，ID，EX，MEM，WB の 5 ステージですが，メモリへのアクセス，レジスタへのデータ書き込みがないので MEM ステージ，WB ステージでは何も行いません。また，次命令では，HALT モードが解除されるまで ID ステージが遅れます。

特殊命令（LDSR，STSR 命令）

［対象の命令］

LDSR，STSR

［パイプライン］



［詳細説明］

パイプラインは IF，ID，EX，DF，WB の 5 ステージです。

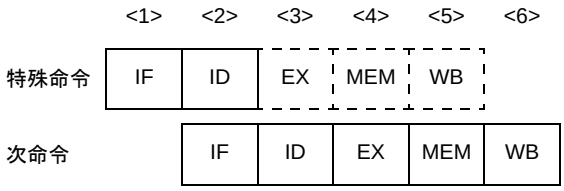
システム・レジスタの EIPC，FEPC を設定する LDSR 命令の直後に，同一レジスタを使用する STSR 命令を配置すると，データの待ち合わせ時間が発生します。

特殊命令（NOP 命令）

[対象の命令]

NOP

[パイプライン]



[詳細説明]

パイプラインは IF，ID，EX，MEM，WB の 5 ステージですが，演算，メモリへのアクセス，レジスタへのデータ書き込みがないので EX ステージ，MEM ステージ，WB ステージでは何も行いません。

[注意]

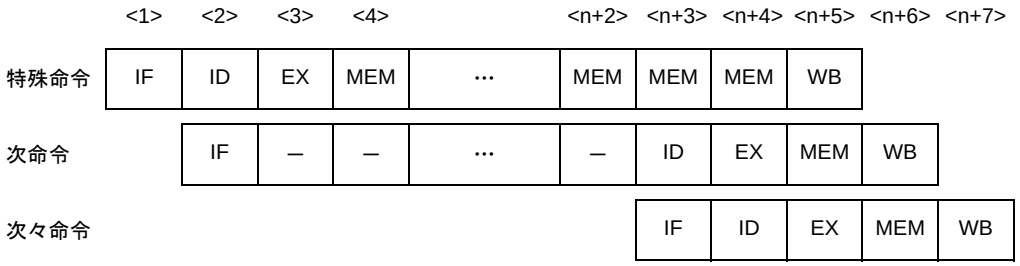
SLD 命令と Bcnd 命令については，ほかの 16 ビット・フォーマットの命令と同時実行される場合があるため注意が必要です。たとえば，SLD 命令と NOP 命令が同時に実行された場合，NOP 命令によるディレイ・タイムの発生が行われない可能性があります。

特殊命令（PREPARE 命令）

[対象の命令]

PREPARE

[パイプライン]



備考　ー：待ち合わせのために挿入されるアイドル
n：レジスタ・リスト（list12）で指定されるレジスタの数

[詳細説明]

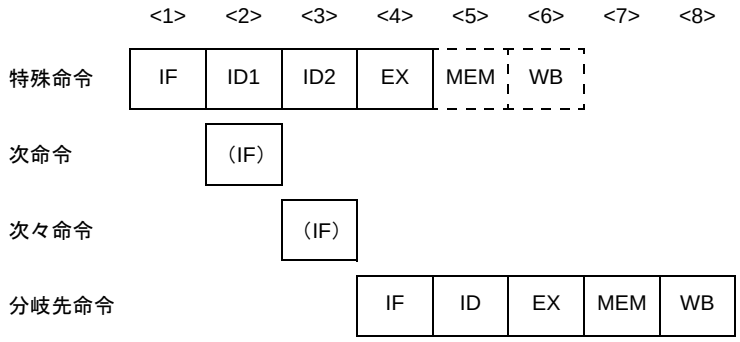
パイプラインは IF，ID，EX，n + 1 回の MEM，WB の n + 5 ステージ（n：レジスタ・リスト・ナンバ）です。
MEM ステージは，n + 1 サイクル必要です。

特殊命令（RETI 命令）

[対象の命令]

RETI

[パイプライン]



備考 （IF）：無効となる命令フェッチ

ID1：レジスタ選択

ID2：EIPC/FEPC 読み込み

[詳細説明]

パイプラインは IF，ID1，ID2，EX，MEM，WB の 6 ステージですが，メモリへのアクセス，レジスタへのデータ書き込みがないので MEM ステージ，WB ステージでは何も行いません。

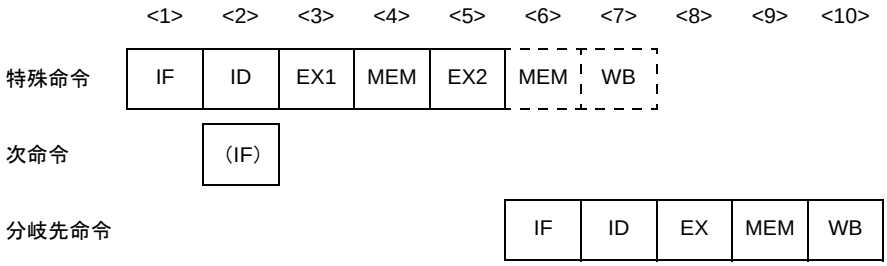
ID ステージには 2 クロックかかります。また，次命令の IF，次々命令の IF は無効となります。

特殊命令（SWITCH 命令）

［対象の命令］

SWITCH

［パイプライン］



備考　（IF）：無効となる命令フェッチ

［詳細説明］

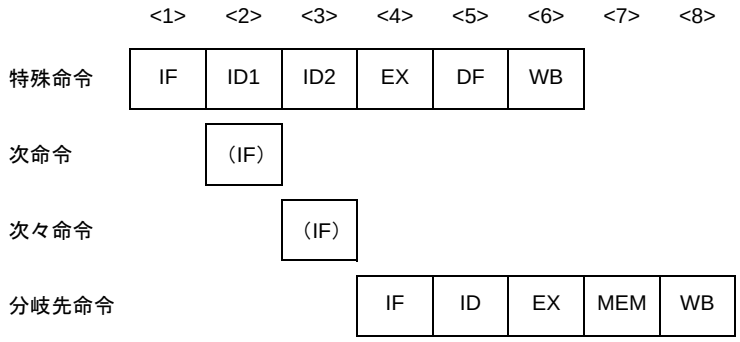
パイプラインは IF，ID，EX1（通常の EX ステージ），MEM，EX2，MEM，WB の 7 ステージですが，2 回目のメモリへのアクセス，レジスタへのデータ書き込みがないので 2 回目の MEM ステージ，WB ステージでは何も行いません。

特殊命令（TRAP 命令）

[対象の命令]

TRAP

[パイプライン]



備考 （IF）：無効となる命令フェッチ

ID1：例外コード（004nH，005nH）検出（n = 0-FH）

ID2：アドレス生成

[詳細説明]

パイプラインは IF，ID1，ID2，EX，DF，WB の 6 ステージです。

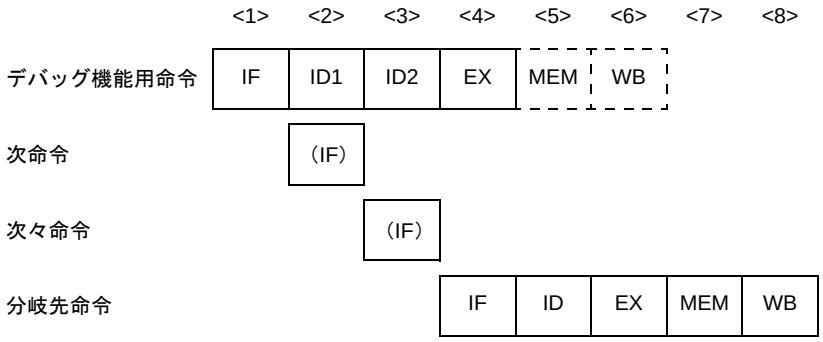
ID ステージには 2 クロックかかります。また，次命令の IF，次々命令の IF は無効となります。

デバッグ機能用命令（DBRET 命令）

[対象の命令]

DBRET

[パイプライン]



備考 （IF）：無効となる命令フェッチ
ID1：レジスタ選択
ID2：DBPC 読み込み

[詳細説明]

パイプラインは IF，ID1，ID2，EX，MEM，WB の 6 ステージですが，メモリへのアクセス，レジスタへのデータ書き込みがないので MEM ステージ，WB ステージでは何も行いません。

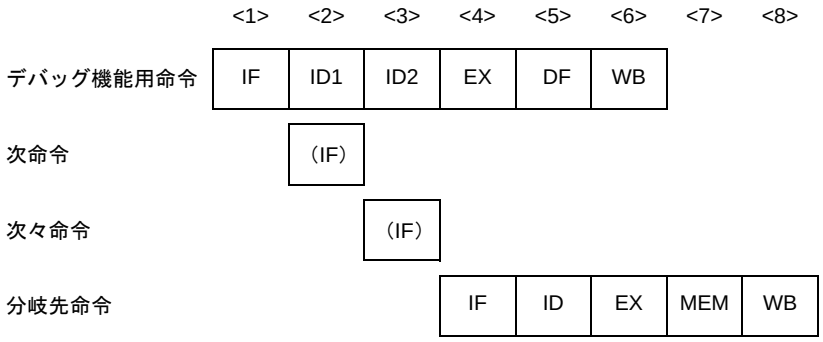
ID ステージには 2 クロックかかります。また，次命令の IF，次々命令の IF は無効となります。

デバッグ機能用命令（DBTRAP 命令）

[対象の命令]

DBTRAP

[パイプライン]



備考 （IF）：無効となる命令フェッチ
ID1：例外コード（0060H）検出
ID2：アドレス生成

[詳細説明]

パイプラインは IF，ID1，ID2，EX，DF，WB の 6 ステージです。
ID ステージには 2 クロックかかります。また，次命令の IF，次々命令の IF は無効となります。

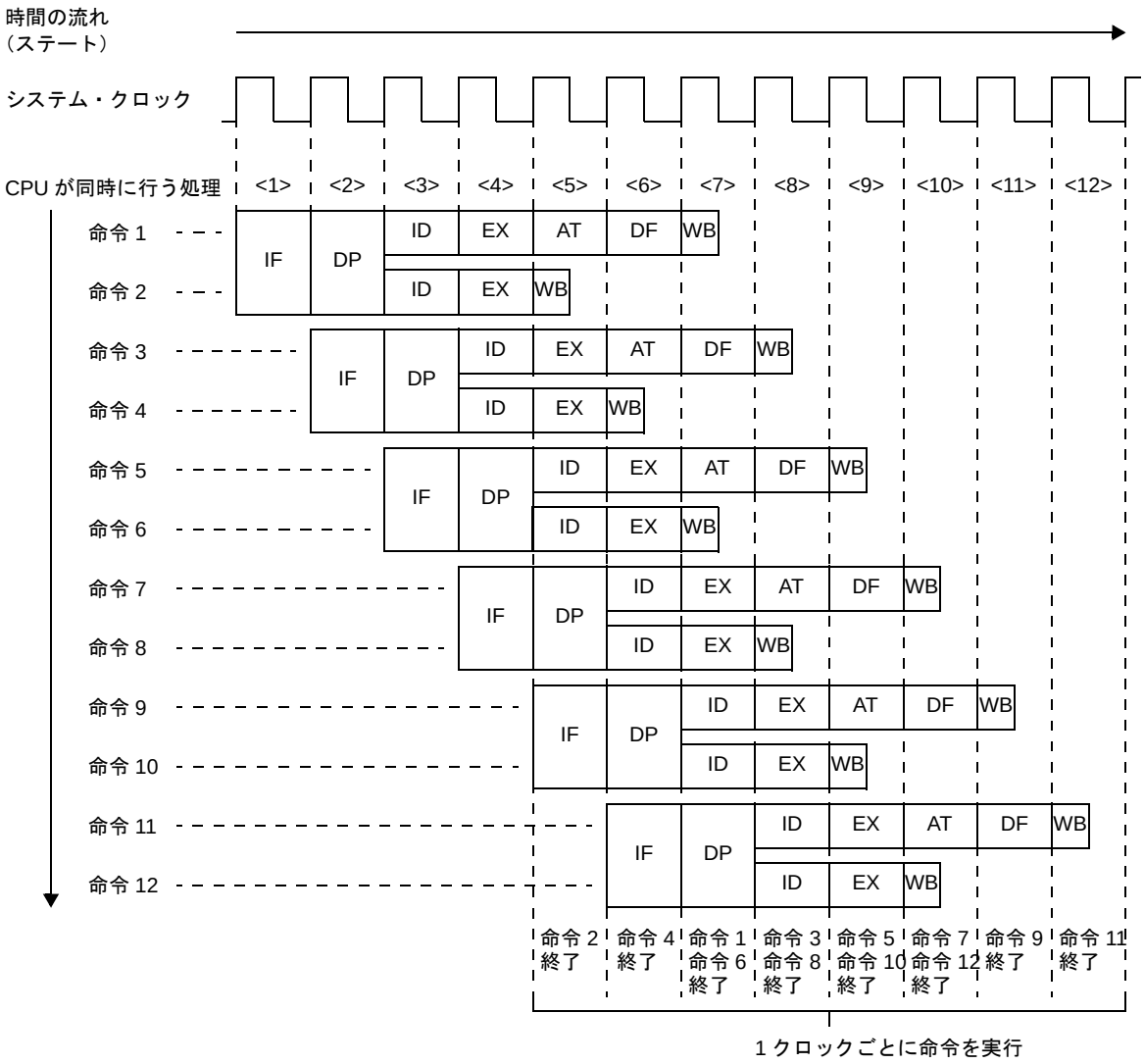
4. 5. 17　パイプライン（V850E2）

V850E2 は，RISC アーキテクチャをベースとした 7 段パイプライン制御により，ほとんどの命令を 1 クロックで実行します。命令実行手順は，通常，IF（インストラクション・フェッチ）から WB（ライトバック）までの 7 ステージで構成されています。

IF（インストラクション・フェッチ）	命令をフェッチし，フェッチ・ポインタをインクリメント
DP（ディスパッチ）	命令の種類，依存関係を調べ，対応するパイプラインに発行
ID（インストラクション・デコード）	命令をデコードし，イミーディエト・データの作成，レジスタの読み出しを行う
EX（実行）	デコードした命令を実行
AT（アドレス転送）	対応メモリにアドレスを転送
DF（データ・フェッチ）	対象メモリからデータを読み出す
WB（ライトバック）	レジスタに実行結果を書き込む

各ステージの実行時間は，命令の種類，アクセス対象となるメモリの種類などによって異なります。
パイプラインの動作例として，標準的な命令を 12 個続けて実行した際の CPU の処理を下図に示します。

図 4―82 標準的な命令を 12 個続けて実行する例



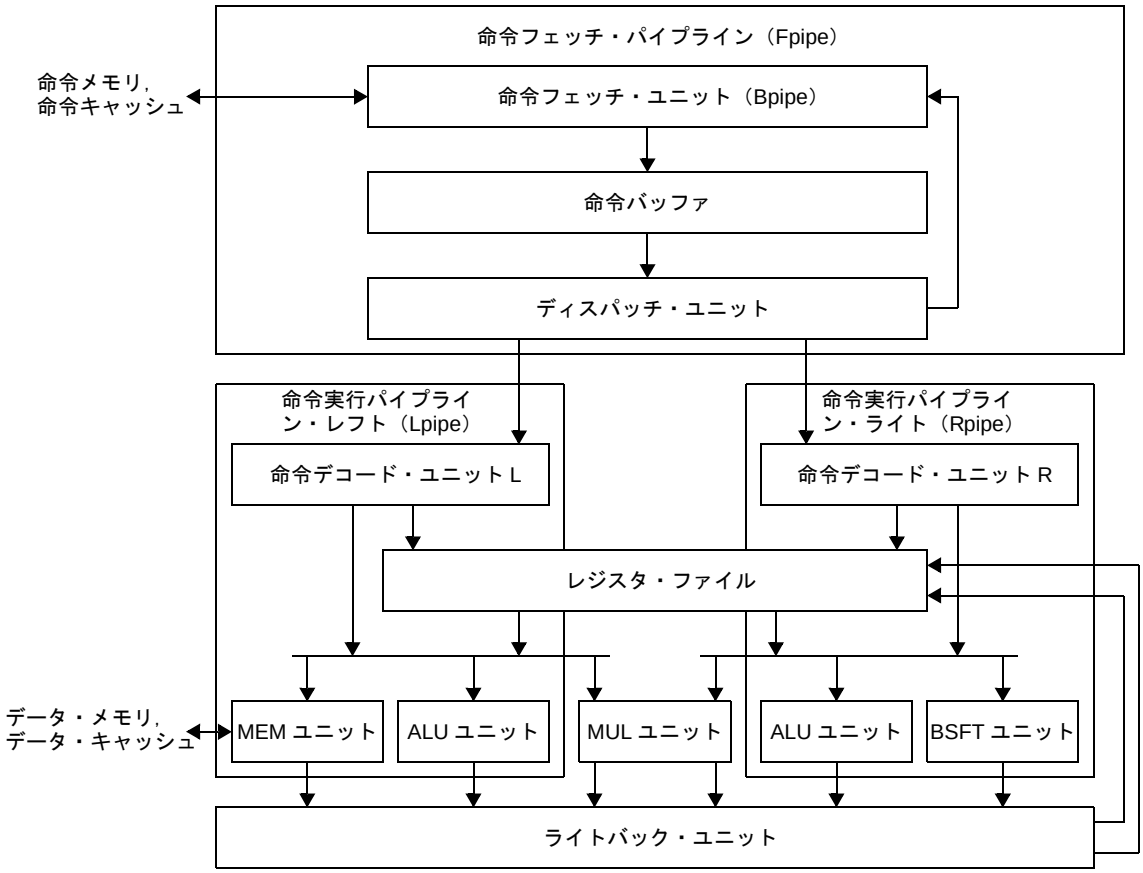
<1> ～ <12> は CPU のステートを示します。標準的な命令では，1 クロックに 2 個の命令の実行（EX）が並列に行えます。

V850E2 は，次に示す独立した 3 個のパイプラインで構成されます。

- Fpipe（命令フェッチ・パイプライン）
- Lpipe（命令実行パイプライン・レフト）
- Rpipe（命令実行パイプライン・ライト）

V850E2 は，命令の依存関係を検出し，最大で 2 個の命令を同時に発行可能な構成になっています。以下に，V850E2 のパイプライン構成を示します。

図 4―83 パイプライン構成（V850E2）



- 命令フェッチ・パイプライン（Fpipe）

  次に示す 3 つのユニットで構成されています。

  - 命令フェッチ・ユニット（Bpipe）

    128 ビットのフェッチ・バス（iLB）から最大 8 命令（1 命令が 16 ビットの場合）を 1 サイクルでフェッチします。

  - ディスパッチ・ユニット

    128 ビット× 2 段の命令キューを内蔵しており，このキューで命令の依存関係を検出し，最大で 2 つの命令を効率よく命令実行パイプラインに発行します。

  - 命令バッファ

    命令フェッチ・ユニット（Bpipe）によってフェッチされた命令を格納します。

- 命令実行パイプライン・レフト（Lpipe）

  次に示す 3 つのユニットで構成されています。

  - 命令デコード・ユニット L

    ディスパッチ・ユニットから発行された命令をデコードします。

  - ALU ユニット

    整数演算，論理演算を行う命令を実行します。

  - MEM ユニット

    ロード命令，ストア命令を含むメモリ・アクセスを行う命令を実行します。

- 命令実行パイプライン・ライト（Rpipe）

  次に示す 3 つのユニットで構成されています。

  - 命令デコード・ユニット R

    ディスパッチ・ユニットから発行された命令をデコードします。

  - ALU ユニット

    整数演算，論理演算を行う命令を実行します。

  - BSFT ユニット

    データ操作を行う命令を実行します。

- MUL ユニット

  整数乗算を行う命令を実行します。

- ライトバック・ユニット

  レジスタ・ファイルにライトバックする制御をします。

（1）各命令実行時のパイプラインの流れ

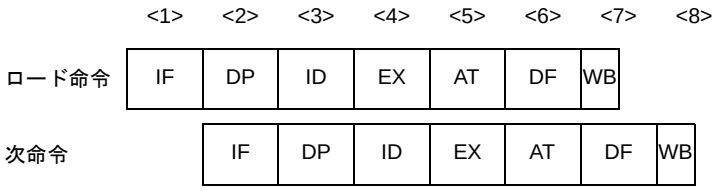
以下に，各命令実行時のパイプラインの流れについて説明します。

ロード命令

[対象の命令]

LD.B，LD.BU，LD.H，LD.HU，LD.W，SLD.B，SLD.BU，SLD.H，SLD.HU，SLD.W

[パイプライン]



[詳細説明]

パイプラインは IF，DP，ID，EX，AT，DF，WB の 7 ステージです。

上図では，Lpipe でロード命令が実行され，Lpipe に次命令が発行された場合の動作を示しています。Rpipe は，ロード命令と依存関係がないかぎり，独立に処理を実行します。ロード命令の直後に，実行結果を使用する命令を配置すると，データの待ち合わせ時間が発生します。

各命令は，ほかの命令との並列発行が可能です。

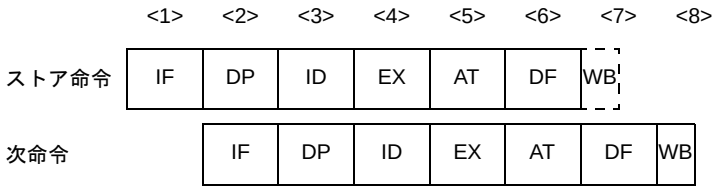
備考 ロード命令は，命令実行パイプライン・レフト（Lpipe）の MEM ユニットで実行されます。

ストア命令

[対象の命令]

SST.B，SST.H，SST.W，ST.B，ST.H，ST.W

[パイプライン]



[詳細説明]

パイプラインは IF，DP，ID，EX，AT，DF，WB の 7 ステージですが，レジスタへのデータの書き込みがないので WB ステージでは何も行いません。

上図では，Lpipe でストア命令が実行され，Lpipe に次命令が発行された場合の動作を示しています。Rpipe は，ストア命令と依存関係がないかぎり，独立に処理を実行します。

各命令は，ほかの命令との並列発行が可能です。

備考　ストア命令は，命令実行パイプライン・レフト（Lpipe）の MEM ユニットで実行されます。

乗算命令

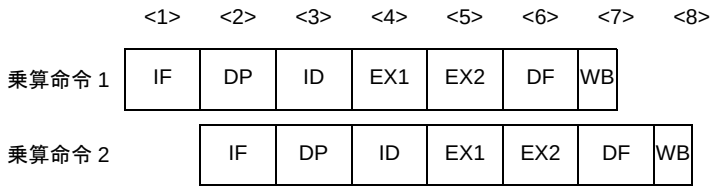
[対象の命令]

MUL，MULH，MULHI，MULU

[パイプライン]

(1) 次命令が乗算命令（，または加算付き付き乗算命令）以外の場合

(2) 次命令が乗算命令（，または加算付き乗算命令）の場合



[詳細説明]

パイプラインは IF，DP，ID，EX1，EX2，DF，WB の 7 ステージです。

EX ステージは 2 クロックかかりますが，EX1 と EX2 は独立して動作できます。したがって，乗算命令（，または加算付き乗算命令）を繰り返しても命令実行クロック数は，1 となります。

上図では，Lpipe で乗算命令が実行され，Lpipe に次命令が発行された場合の動作を示しています。ただし，乗算命令の直後に実行結果を使用する命令を配置すると，データの待ち合わせ時間が発生します。

各命令は，ほかの命令との並列発行が可能です。

備考 乗算命令は，命令実行パイプライン・レフト（Lpipe）の MUL ユニットで実行されます。

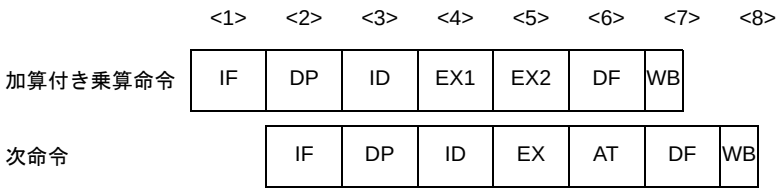
加算付き乗算命令

[対象の命令]

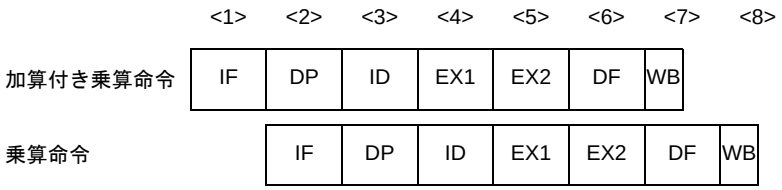
MAC，MACU

[パイプライン]

（1）次命令が乗算命令（，または加算付き付き乗算命令）以外の場合



（2）次命令が乗算命令（，または加算付き乗算命令）の場合



[詳細説明]

パイプラインは IF，DP，ID，EX1，EX2，DF，WB の 7 ステージです。

EX ステージは 2 クロックかかりますが，EX1 と EX2 は独立して動作できます。したがって，加算付き乗算命令（，または乗算命令）を繰り返しても命令実行クロック数は，1 となります。

上図では，Lpipe で加算付き乗算命令が実行され，Lpipe に次命令が発行された場合の動作を示しています。Rpipe は，加算付き乗算命令と依存関係がないかぎり，独立に処理を実行します。ただし，加算付き乗算命令の直後に実行結果を使用する命令を配置すると，データの待ち合わせ時間が発生します。

各命令は，単独で発行されます。

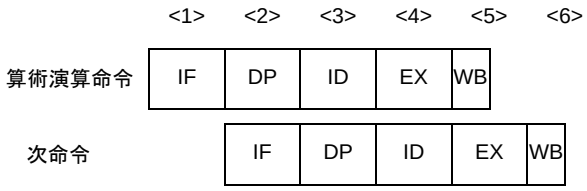
備考　加算付き乗算命令は，命令実行パイプライン・レフト（Lpipe）の MUL ユニットで実行されます。

算術演算命令

[対象の命令]

ADD，ADDI，CMP，MOV，MOVEA，MOVHI，SUB，SUBR

[パイプライン]



[詳細説明]

パイプラインは IF，DP，ID，EX，WB の 5 ステージです。

上図では，Rpipe で算術演算命令が実行され，Rpipe に次命令が発行された場合の動作を示しています。Lpipe は，算術演算命令と依存関係がないかぎり，独立に処理を実行します。

MOV imm32, reg1 命令を除く各命令は，ほかの命令との並列発行が可能です（MOV imm32, reg1 命令は，単独で発行されます）。

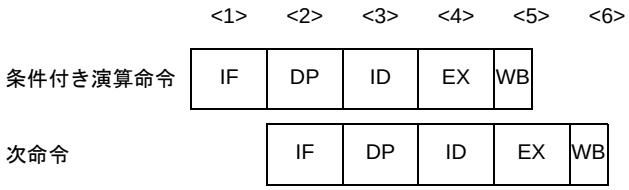
備考 算術演算命令は，命令実行パイプライン・レフト（Lpipe），または命令実行パイプライン・ライト（Rpipe）の ALU ユニットで実行されます。

条件付き演算命令

[対象の命令]

ADF，SBF

[パイプライン]



[詳細説明]

パイプラインは IF，DP，ID，EX，WB の 5 ステージです。

上図では，Rpipe で条件付演算命令が実行され，Rpipe に次命令が発行された場合の動作を示しています。Lpipe は，条件付演算命令と依存関係がないかぎり，独立に処理を実行します。

各命令は，単独で発行されます。

備考 条件付き演算命令は，命令実行パイプライン・レフト（Lpipe），または命令実行パイプライン・ライト（Rpipe）の ALU ユニットで実行されます。

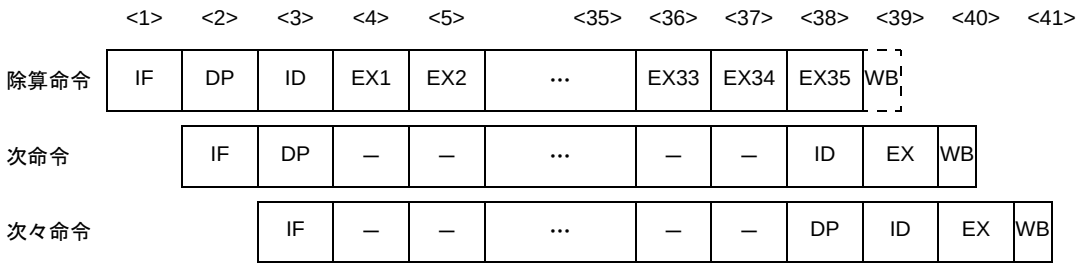
除算命令

[対象の命令]

DIV，DIVH，DIVHU，DIVU

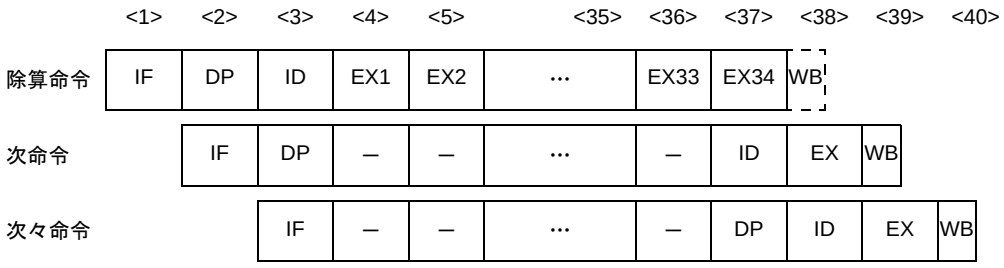
[パイプライン]

（1）DIV，DIVH 命令の場合



備考　ー：待ち合わせのために挿入されるアイドル

（2）DIVHU，DIVU 命令の場合



備考　ー：待ち合わせのために挿入されるアイドル

[詳細説明]

パイプラインは，DIV，DIVH 命令の場合，IF，DP，ID，EX1-EX35，WB の 39 ステージ，DIVHU，DIVU 命令の場合，IF，DP，ID，EX1-EX34，WB の 38 ステージです。

上図では，Rpipe で除算命令が実行され，Rpipe に次命令が発行された場合の動作を示しています。

ただし，除算命令が ID ステージで命令をデコードしている期間と EX ステージで命令を実行している期間は，ディスパッチ・ユニットは Rpipe に命令を発行しません。

各命令は，単独で発行されます。

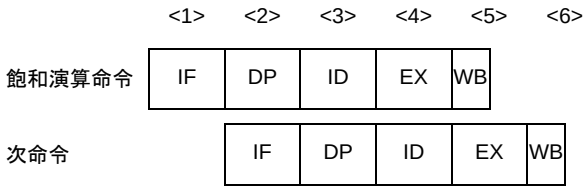
備考　乗算命令は，命令実行パイプライン・ライト（Rpipe）の ALU ユニットで実行されます。

飽和演算命令

[対象の命令]

SATADD，SATSUB，SATSUBI，SATSUBR

[パイプライン]



[詳細説明]

パイプラインは IF，DP，ID，EX，WB の 5 ステージです。

上図では，Rpipe で飽和演算命令が実行され，Rpipe に次命令が発行された場合の動作を示しています。Lpipe は，飽和演算命令と依存関係がないかぎり，独立に処理を実行します。

SATADD reg1, reg2, reg3 命令と SATSUB reg1, reg2, reg3 命令を除く各命令は，ほかの命令との並列発行が可能です（SATADD reg1, reg2, reg3 命令と SATSUB reg1, reg2, reg3 命令は，単独で発行されます）。

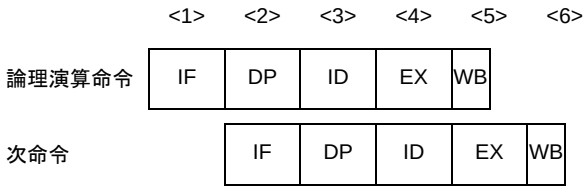
備考 飽和演算命令は，命令実行パイプライン・レフト（Lpipe），または命令実行パイプライン・ライト（Rpipe）の ALU ユニットで実行されます。

論理演算命令

[対象の命令]

AND，ANDI，NOT，OR，ORI，TST，XOR，XORI

[パイプライン]



[詳細説明]

パイプラインは IF，DP，ID，EX，WB の 5 ステージです。

上図では，Rpipe で論理演算命令が実行され，Rpipe に次命令が発行された場合の動作を示しています。Lpipe は，論理演算命令と依存関係がないかぎり，独立に処理を実行します。

各命令は，ほかの命令との並列発行が可能です。

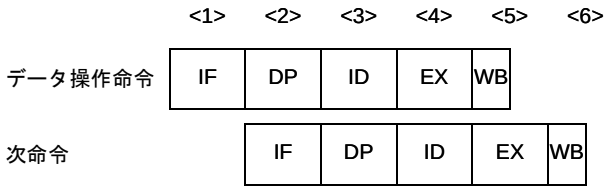
備考 論理演算命令は，命令実行パイプライン・レフト（Lpipe），または命令実行パイプライン・ライト（Rpipe）の ALU ユニットで実行されます。

データ操作命令

[対象の命令]

BSH，BSW，CMOV，HSH，HSW，SAR，SASF，SETF，SHL，SHT，SXB，SXH，ZXB，ZXH

[パイプライン]



[詳細説明]

パイプラインは IF，DP，ID，EX，WB の 5 ステージです。

上図では，Rpipe でデータ操作命令が実行され，Rpipe に次命令が発行された場合の動作を示しています。Lpipe は，データ操作命令と依存関係がないかぎり，独立に処理を実行します。

各命令は，ほかの命令との並列発行が可能です。

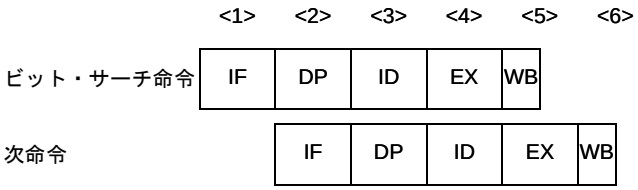
備考 データ操作命令は，命令実行パイプライン・ライト（Rpipe）の BSFT ユニットで実行されます。

ビット・サーチ命令

[対象の命令]

SCH0L，SCH0R，SCH1L，SCH1R

[パイプライン]



[詳細説明]

パイプラインは IF，DP，ID，EX，WB の 5 ステージです。

上図では，Rpipe でビット・サーチ命令が実行され，Rpipe に次命令が発行された場合の動作を示しています。Lpipe は，ビット・サーチ命令と依存関係がないかぎり，独立に処理を実行します。

各命令は，ほかの命令との並列発行が可能です。

備考 ビット・サーチ命令は，命令実行パイプライン・ライト（Rpipe）の BSFT ユニットで実行されます。

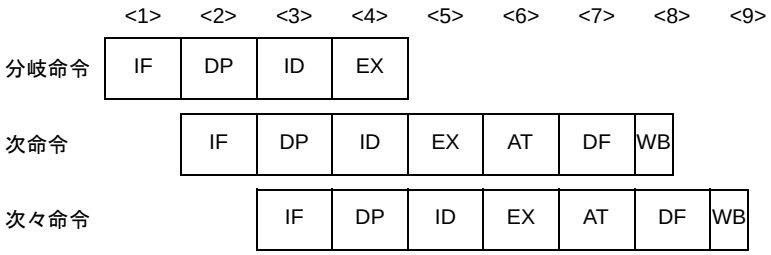
分岐命令（条件分岐命令：BR 命令を除く）

[対象の命令]

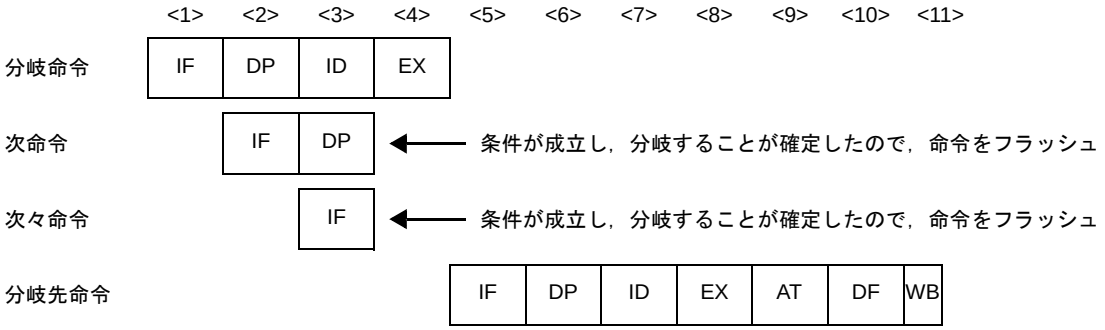
Bcnd 命令

[パイプライン]

(1) 条件が成立しない場合



(2) 条件が成立した場合



[詳細説明]

上図では，Bpipe で Bcnd 命令が実行され，Lpipe ですべての命令が実行された場合の動作を示しています。
各命令は，ほかの命令との並列発行が可能です。
また，実行クロック数は，以下の通りです。

分岐命令	実行クロック数
条件が成立しない場合	1
条件が成立した場合	4 **注**

注　命令バッファにターゲットの命令が存在していた場合は 3（4 - 1 = 3）
直前に PSW を書き換える命令がある場合は 6

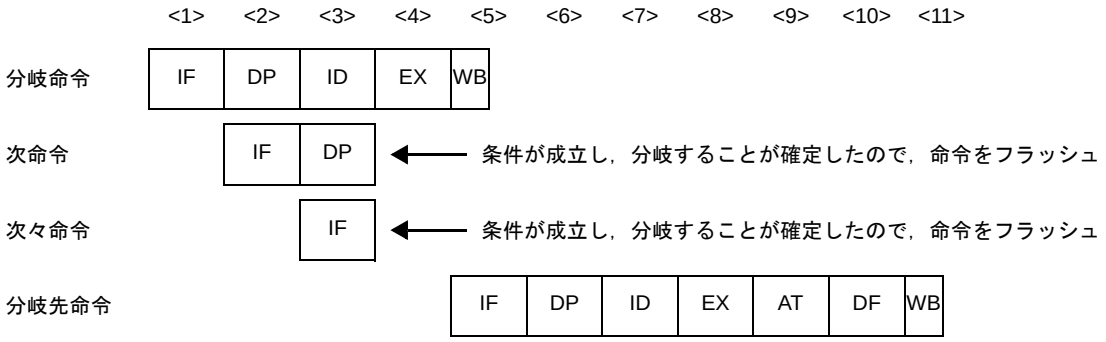
備考　分岐命令（条件分岐命令：BR 命令を除く）は，命令フェッチ・ユニット（Bpipe）で実行されます。

分岐命令（BR 命令，無条件分岐命令：JMP 命令を除く）

[対象の命令]

BR，JARL，JR

[パイプライン]



[詳細説明]

上図では，Bpipe で分岐命令が実行され，Lpipe ですべての命令が実行された場合の動作を示しています。

JARL disp32, reg1 命令と JR disp32 命令以外の各命令は，ほかの命令との並列発行が可能です。

また，実行クロック数は，4（命令バッファにターゲットの命令が存在していた場合は 3（4 - 1 = 3））です。

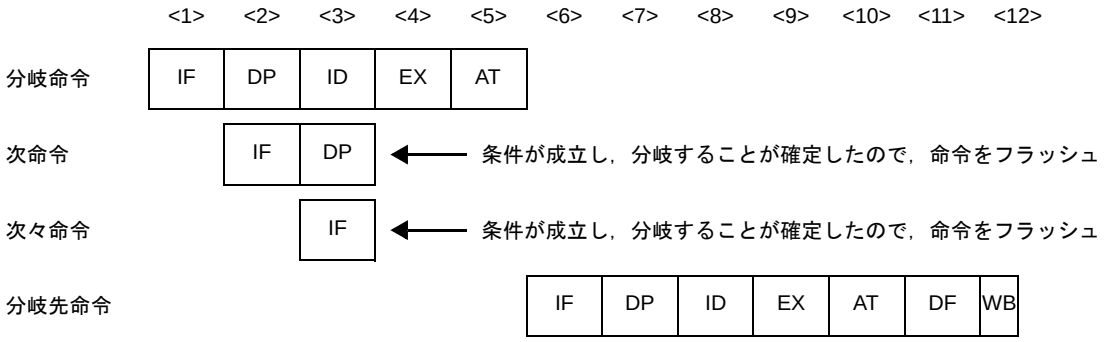
備考 分岐命令（BR 命令，無条件分岐命令：JMP 命令を除く）は，命令フェッチ・ユニット（Bpipe）で実行されます。

分岐命令（JMP 命令）

[対象の命令]

JMP

[パイプライン]



[詳細説明]

上図では，Bpipe で JMP 命令が発行され，Lpipe ですべての命令が実行された場合の動作を示しています。

JMP [reg1] 命令は，ほかの命令との並列発行が可能です（JMP disp32[reg1] 命令は不可）。

また，実行クロック数は，5 です（命令バッファにターゲットの命令が存在していた場合は 4（5 - 1 = 4））。

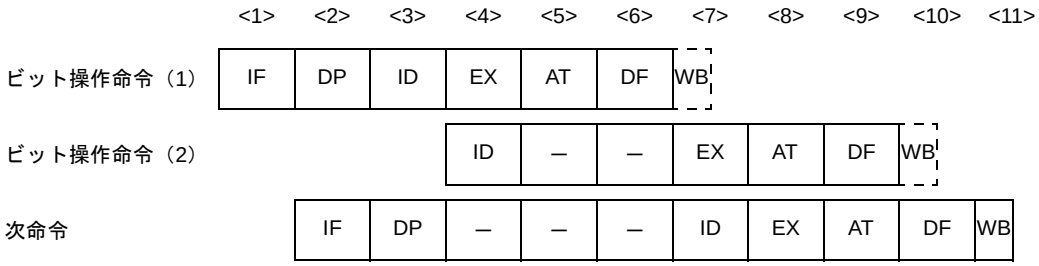
備考 分岐命令（JMP 命令）は，命令フェッチ・ユニット（Bpipe）で実行されます。

ビット操作命令（CLR1，NOT1，SET1 命令）

［対象の命令］

CLR1，NOT1，SET1

［パイプライン］



備考　―：待ち合わせのために挿入されるアイドル

［詳細説明］

ID ステージで 2 つの命令に分割され，最初にロード命令を，次にビット操作命令を含むストア命令を実行しますが，レジスタへのデータ書き込みがないので WB ステージでは何も行いません。

上図では，Lpipe でビット操作命令が実行され，Lpipe に次命令が発行された場合の動作を示しています。Rpipe は，ビット操作命令と依存関係がないかぎり，独立に処理を実行します。ID ステージで命令をデコードしている期間はディスパッチ・ユニットは Lpipe に命令を発行しません。

各命令は，単独で発行されます。

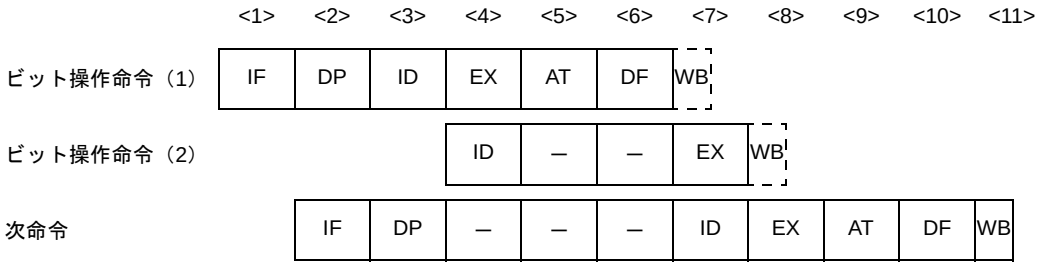
備考　ビット操作命令（CLR1，NOT1，SET1 命令）は，命令実行パイプライン・レフト（Lpipe）の ALU ユニットで実行されます。

ビット操作命令（TST1 命令）

[対象の命令]

TST1

[パイプライン]



備考 ー：待ち合わせのために挿入されるアイドル

[詳細説明]

ID ステージで 2 つの命令に分割され，最初にロード命令を，次にビット操作命令を実行しますが，レジスタへのデータ書き込みがないので WB ステージでは何も行いません。

上図では，Lpipe で TST1 命令が実行され，Lpipe に次命令が発行された場合の動作を示しています。Rpipe は，ビット操作命令と依存関係がないかぎり，独立に処理を実行します。ID ステージで命令をデコードしている期間はディスパッチ・ユニットは Lpipe には命令を発行しません。

各命令は，単独で発行されます。

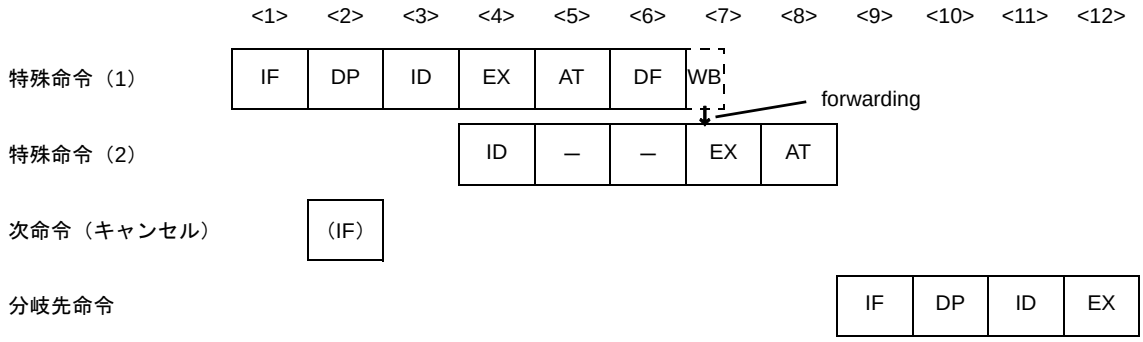
備考 ビット操作命令（TST1 命令）は，命令実行パイプライン・レフト（Lpipe）の ALU ユニットで実行されます。

特殊命令（CALLT 命令）

[対象の命令]

CALLT

[パイプライン]



備考　ー：待ち合わせのために挿入されるアイドル
（IF）：無効となる命令フェッチ

[詳細説明]

ID ステージで 2 つの命令に分割され，最初にロード命令を，次に CTBP 相対の分岐命令を実行しますが，レジスタへのデータ書き込みがないので WB ステージでは何も行いません。

上図では，Lpipe で CALLT 命令が実行され，分岐先から命令をフェッチして実行するまでの動作を示しています。Rpipe は，CALLT 命令と依存関係がないかぎり，独立に処理を実行します。

この命令は，単独で発行されます。

また，実行クロック数は，8 です。

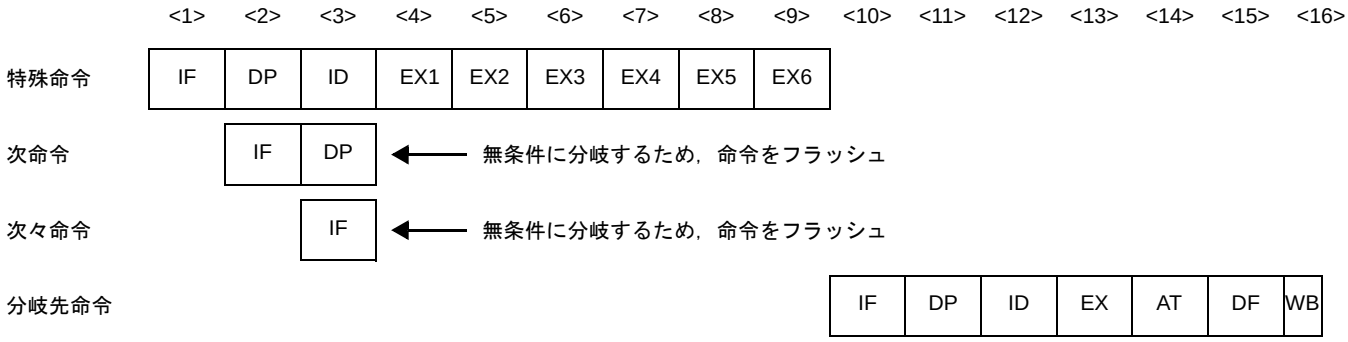
備考　特殊命令（CALLT 命令）は，命令実行パイプライン・レフト（Lpipe）の ALU ユニットで実行されます。

特殊命令（CTRET，TRAP 命令）

［対象の命令］

CTRET，TRAP

［パイプライン］



［詳細説明］

上図では，Bpipe で CTRET，TRAP 命令が実行され，Lpipe ですべての命令が発行された場合の動作を示しています。

各命令は，単独で発行されます。

また，実行クロック数は，9 です。

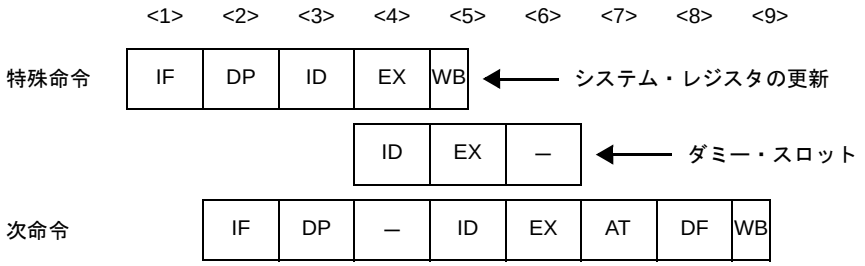
備考　特殊命令（CRET，TRAP 命令）は，命令フェッチ・ユニット（Bpipe）で実行されます。

特殊命令（DI，EI，LDSR 命令）

[対象の命令]

DI，EI，LDSR

[パイプライン]



備考 ー：待ち合わせのために挿入されるアイドル

[詳細説明]

パイプラインは IF，DP，ID，EX，WB の 5 ステージです。

上図では，Rpipe で DI，EI，LDSR 命令が実行され，Rpipe ですべての命令が実行された場合の動作を示しています。

各命令は，単独で発行されます。

備考 特殊命令（DI，EI，LDSR 命令）は，命令実行パイプライン・ライト（Rpipe）の ALU ユニットで実行されます。

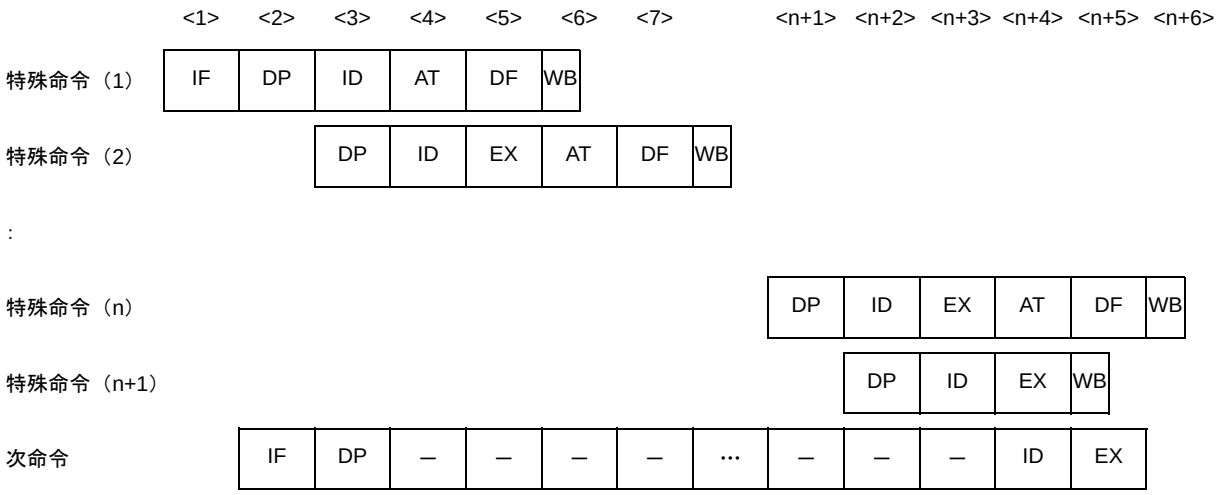
特殊命令（DISPOSE 命令）

［対象の命令］

DISPOSE

［パイプライン］

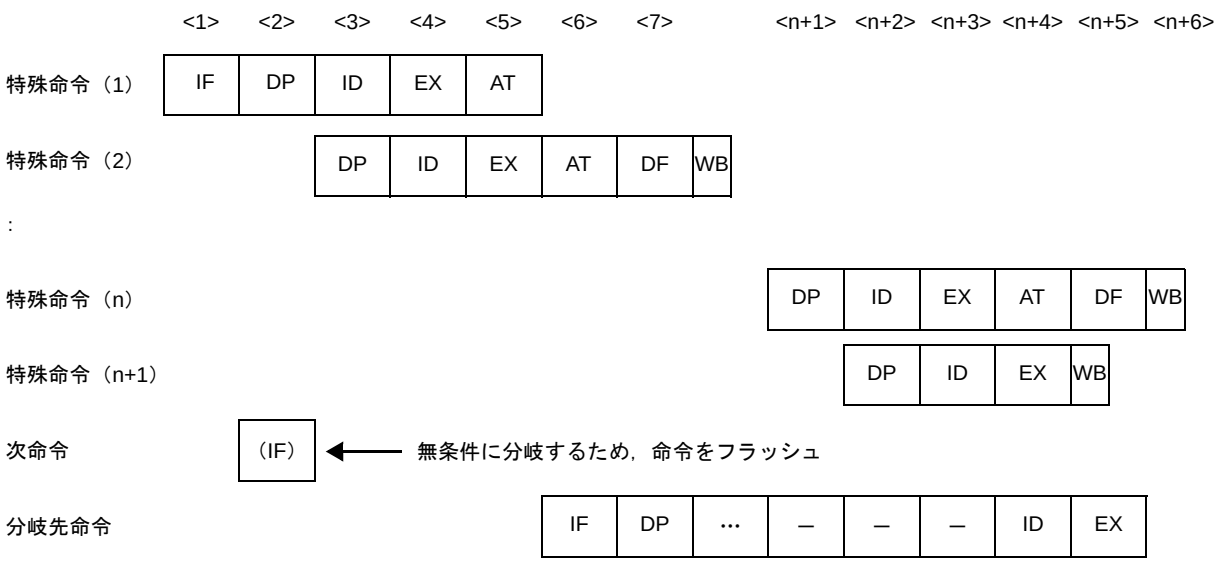
（1）分岐しない場合



備考　ー：待ち合わせのために挿入されるアイドル

n：レジスタ・リスト（list12）で指定されるレジスタの数

（2）分岐する場合



備考　（IF）：無効となる命令フェッチ

ー：待ち合わせのために挿入されるアイドル

n：レジスタ・リスト（list12）で指定されるレジスタの数

［詳細説明］

DP ステージで n + 1 個の命令に分割され，最初に n 個のロード命令を，最後にスタック・ポインタ SP への書き込み命令を実行します。

上図では，Lpipe で DISPOSE 命令が実行され，Lpipe に次命令が発行された場合の動作を示しています。Rpipe は，DISPOSE 命令と依存関係がないかぎり，独立に処理を実行します。

DP ステージで命令をデコードしている期間は，ディスパッチ・ユニットは Lpipe には命令を発行しません。

この命令は，単独で発行されます。

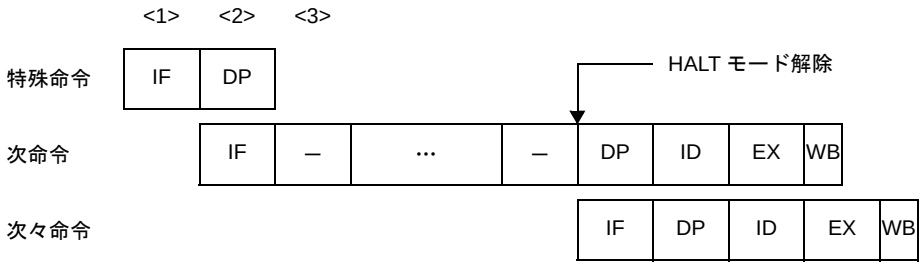
備考　特殊命令（DISPOSE 命令）は，命令実行パイプライン・ライト（Rpipe）の ALU ユニットで実行されます。

特殊命令（HALT 命令）

[対象の命令]

HALT

[パイプライン]



備考 ー：待ち合わせのために挿入されるアイドル

[詳細説明]

DP ステージで HALT 命令が検出されると，HALT 命令が解除されるまで，ID ステージへ命令の発行を停止します。したがって，次命令では HALT 命令が解除されるまで ID ステージが遅れます。

上図では，Rpipe で HALT 命令が実行され，Rpipe に次命令が発行された場合の動作を示しています。

この命令は，単独で発行されます。

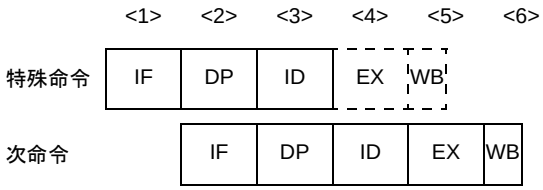
備考 特殊命令（HALT 命令）は，命令フェッチ・パイプライン（Fpipe）のディスパッチ・ユニットで実行されます。

特殊命令（NOP 命令）

[対象の命令]

NOP

[パイプライン]



[詳細説明]

パイプラインは IF，DP，ID，EX，WB の 5 ステージですが，演算，レジスタへのデータ書き込みがないので EX ステージ，WB ステージでは何も行いません。

上図では，Rpipe で NOP 命令が実行され，Rpipe に次命令が発行された場合の動作を示しています。Lpipe は，NOP 命令と依存関係にないかぎり，独立に処理を実行します。

この命令は，ほかの命令との並列発行が可能です。

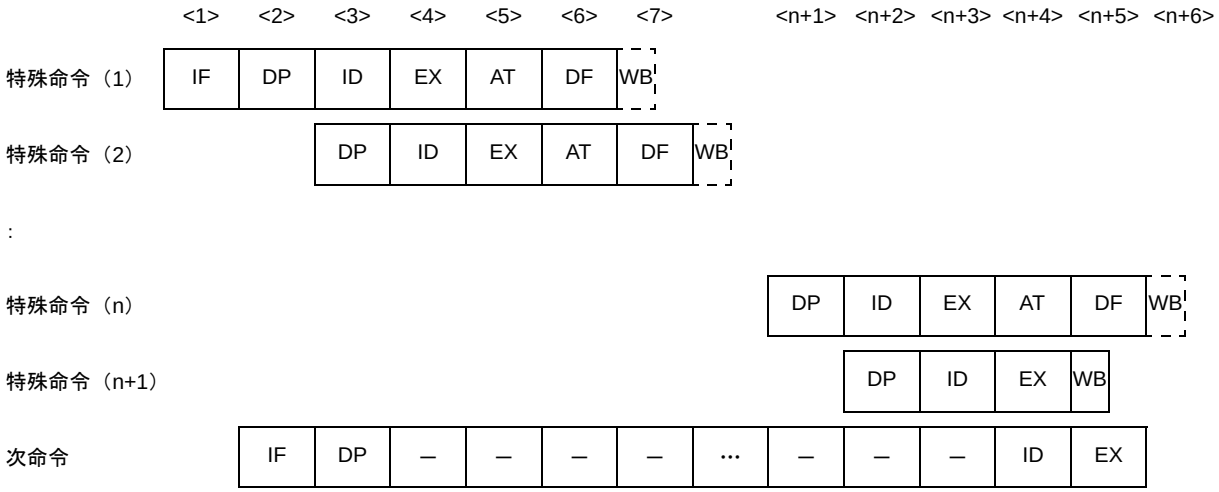
備考 特殊命令（NOP 命令）は，命令実行パイプライン・レフト（Lpipe），または命令実行パイプライン・ライト（Rpipe）の ALU ユニットで実行されます。

特殊命令（PREPARE 命令）

［対象の命令］

PREPARE

［パイプライン］



備考　ー：待ち合わせのために挿入されるアイドル
n：レジスタ・リスト（list12）で指定されるレジスタの数

［詳細説明］

DP ステージで n + 1 個の命令に分割され，最初に n 個のストア命令を，最後にスタック・ポインタ SP への書き込み命令を実行しますが，ストア命令ではレジスタへのデータ書き込みがないので WB ステージでは何も行いません。

上図では，Lpipe で PREPARE 命令が実行され，Lpipe に次命令が発行された場合の動作を示しています。Rpipe は，PREPARE 命令と依存関係がないかぎり，独立に処理を実行します。

DP ステージで命令をデコードしている期間は，ディスパッチ・ユニットは Lpipe には命令を発行しません。

この命令は，単独で発行されます。

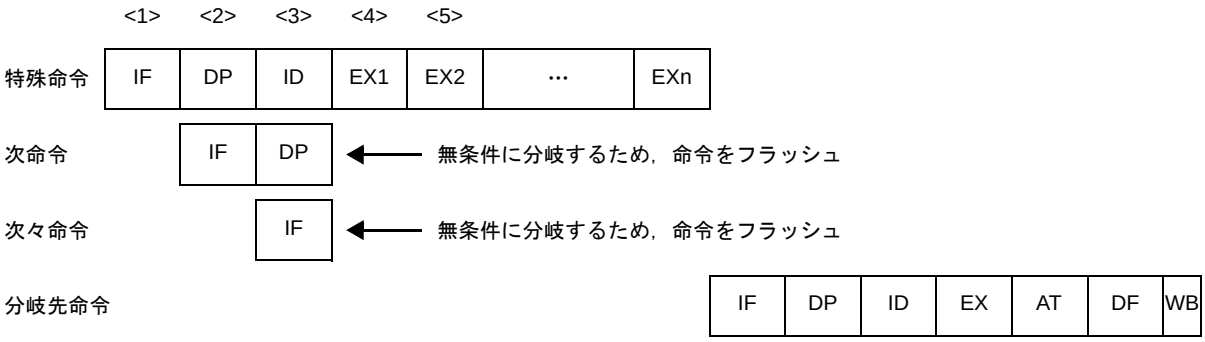
備考　特殊命令（PREPARE 命令）は，命令実行パイプライン・レフト（Lpipe）の ALU ユニットで実行されます。

特殊命令（RETI 命令）

[対象の命令]

RETI

[パイプライン]



[詳細説明]

上図では，Bpipe で RETI 命令が発行され，Lpipe ですべての命令が発行された場合の動作を示しています。
この命令は，単独で発行されます。
また，実行クロック数は，システムによって異なります（割り込みコントローラの動作仕様に依存）。

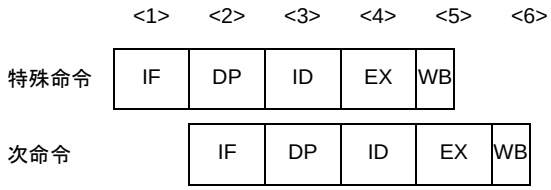
備考 特殊命令（RETI 命令）は，命令フェッチ・ユニット（Bpipe）で実行されます。

特殊命令（STSR 命令）

[対象の命令]

STSR

[パイプライン]



[詳細説明]

パイプラインは IF，DP，ID，EX，WB の 5 ステージです。

上図では，Rpipe で STSR 命令が実行され，Rpipe に次命令が発行された場合の動作を示しています。Lpipe は，STSR 命令と依存関係にないかぎり，独立に処理を実行します。

この命令は，単独で発行されます。

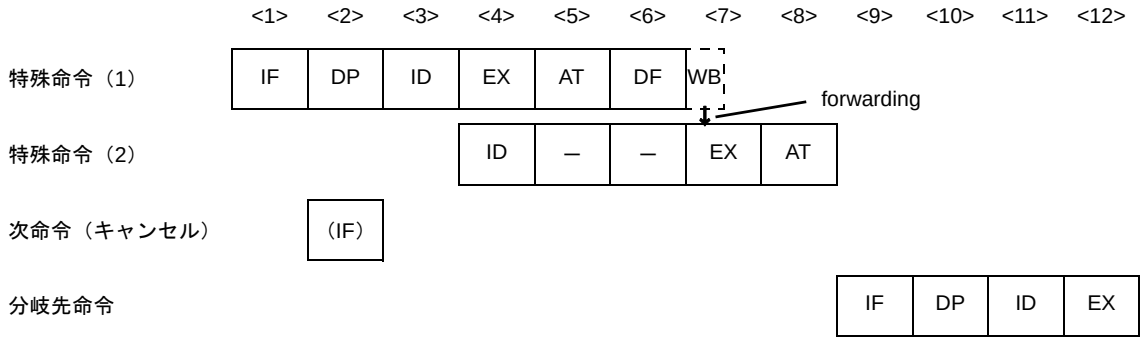
備考　特殊命令（STSR 命令）は，命令実行パイプライン・ライト（Rpipe）の ALU ユニットで実行されます。

特殊命令（SWITCH 命令）

[対象の命令]

SWITCH

[パイプライン]



備考 ー：待ち合わせのために挿入されるアイドル
（IF）：無効となる命令フェッチ

[詳細説明]

ID ステージで 2 つの命令に分割され，最初にロード命令を，次に PC 相対の分岐命令を実行しますが，レジスタへのデータ書き込みがないので WB ステージでは何も行いません。

上図では，Lpipe で SWITCH 命令が実行され，Lpipe に次命令が発行された場合の動作を示しています。Rpipe は，SWITCH 命令と依存関係がないかぎり，独立に処理を実行します。

この命令は，単独で発行されます。

また，実行クロック数は，8 です。

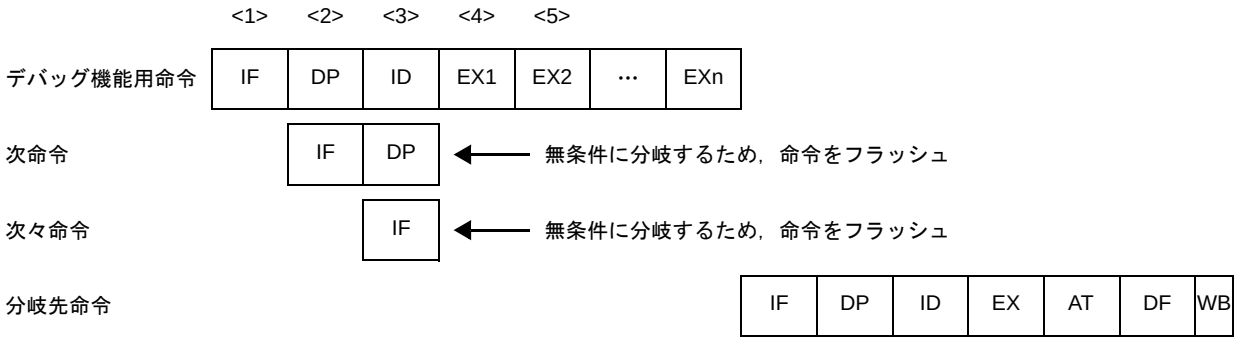
備考 特殊命令（SWITCH 命令）は，命令実行パイプライン・レフト（Lpipe）の ALU ユニットで実行されます。

デバッグ機能用命令（DBRET，DBTRAP 命令）

［対象の命令］

DBRET，DBTRAP

［パイプライン］



［詳細説明］

上図では，Bpipe で DBRET，DBTRAP 命令が実行され，Lpipe ですべての命令が発行された場合の動作を示しています。

各命令は，単独で発行されます。

また，各命令は，CPU で保留されているすべての命令の処理が完了するまで分岐命令の実行を行いません。

備考 デバッグ機能用命令（DBRET，DBTRAP 命令）は，命令フェッチ・ユニット（Bpipe）で実行されます。

第 5 章　リンク・ディレクティブ仕様

この章では，リンク・ディレクティブに必要となる項目や，リンク・ディレクティブ・ファイルの記述方法について説明します。

組み込み系のアプリケーションでは，プログラム・コードをある番地から配置したり，分割して配置するなど，メモリ配置に気を配る必要があります。

期待どおりのメモリ配置を実現するには，プログラム・コードやデータの配置情報を，リンカに指示する必要があります。この指示の情報を「リンク・ディレクティブ」と呼び，リンク・ディレクティブを記述するファイルを「リンク・ディレクティブ・ファイル」と呼びます。

リンカは，このリンク・ディレティブ・ファイルに従ってメモリ配置を決定し，ロード・モジュールを作成します。

5. 1　コーディング方法

ここではリンク・ディレクティブ・ファイルの書式について，次の項目ごとに説明します。

- セグメント・ディレクティブ
- マッピング・ディレクティブ
- シンボル・ディレクティブ

リンク・ディレクティブの書式の概要は，次に示すとおりで，これらをエディタなどにより，テキスト形式で記述します。

```
セグメント・ディレクティブ1 {
    マッピング・ディレクティブ;
};
セグメント・ディレクティブ2 {
    マッピング・ディレクティブ;
};
セグメント・ディレクティブ3 {
    マッピング・ディレクティブ;
};
セグメント・ディレクティブ4 {
    マッピング・ディレクティブ;
};
tp シンボル・ディレクティブ;
gp シンボル・ディレクティブ;
ep シンボル・ディレクティブ;
```

備考　セグメント・ディレクティブは，アドレスの若い順に記述することを推奨します。

5.1.1 使用できる文字

リンク・ディレクティブ・ファイルで使用できる文字は次のとおりです。

- 数字（0 ～ 9）
- 英大文字（A ～ Z）
- 英小文字（a ～ z）
- アンダスコア（_）
- ドット（.）
- スラッシュ（/）
- バック・スラッシュ，円マーク（\，￥）
- コロン（:）（ファイル名のみ使用できます）
- S-JIS 文字（ファイル名のみ使用できます）
- 半角カナ文字（ファイル名のみ使用できます）
- #（コメント用）

リンク・ディレクティブ・ファイル中の“#”は，コメントの始まりを示します。“#”が現れた位置からその行の行末まではコメントとして扱われます。

5.1.2 ファイル名

リンク・ディレクティブ・ファイルにつける「ファイル名」には，ファイル名として指定できる文字を使用していれば，特に制限はありません。ただし，拡張子が必要です。推奨は“dir”です。CubeSuite+ を使用する際は，必ず“dir”または“dr”としてください。また，文字数があまり長くなると，リンク時に扱える全体の文字数（OS依存）を越えてしまい，リンクできなくなってしまうことがあるので注意が必要です。

リンク時には，コマンド・ラインやメイク・ファイルを使用する際は，“-D”オプションを使用してリンク・ディレクティブ・ファイルを指定します。

5.1.3 セグメント・ディレクティブ

ここでは，次の項目でセグメント・ディレクティブの書式について説明します。

- 指定項目
- セグメント・ディレクティブの指定例

(1) 指定項目

セグメント・ディレクティブで指定する項目は，次のとおりです。

表 5―1 セグメント・ディレクティブで指定できる項目

項目	指定形式	意味	省略
セグメント名	セグメント名	作成するセグメントの名前	不可
セグメント・タイプ	!LOAD	メモリにロードされるタイプ（固定）	不可

項目	指定形式	意味	省略
セグメント属性	?[R][W][X]	作成するセグメントの属性が「読み出し可能（R）」「書き込み可能（W）」「実行可能（X）」であるかを指定（複数指定可能）	不可
アドレス	V アドレス	作成するセグメントの先頭アドレス	可
最大メモリ・サイズ	L 最大メモリ・サイズ	作成するセグメントが占有するメモリの上限	可
ホール・サイズ	H ホール・サイズ	セグメントの後ろに作成するホールのサイズ（次のセグメントとの間に入れる余白）	可
フィリング値	F フィリング値	ホール領域を埋めるのに用いる値	可
整列条件	A 整列条件	メモリ割り付けにおける整列条件（アライメント）	可

セグメント・ディレクティブの具体的な書式は次とおりです。

```
セグメント名 : !セグメント・タイプ ?セグメント属性 Vアドレス Lメモリ・サイズ Hホール・サイズ Fフィリング値 A整列条件 {
        :
    （マッピング・ディレクティブ）
        :
};
```

それぞれの項目は，空白で区切って記述します。また終わりには，必ず“;”を付けます。

なお，「V アドレス」「L 最大メモリ・サイズ」「H ホール・サイズ」「F フィリング値」「A 整列条件」は省略できます。省略した場合はデフォルトの値が使われます。デフォルトの値は次のとおりです。

表 5―2 省略可能項目のデフォルト値（セグメント・ディレクティブ）

項目	意味
アドレス	先頭のセグメントである場合は 0x0 番地，以降のセグメントは，前のセグメントの終端に続いた値
最大メモリ・サイズ	0x100000（バイト），セグメントを配置するメモリサイズが 1M を越えるデバイスを品種指定した場合は，セグメントを配置するメモリのサイズ
ホール・サイズ	0x0（バイト）
フィリング値	0x0000
整列条件	0x8（バイト）

備考 セグメント・ディレクティブは，アドレスの若い順に記述することを推奨します。

(a) セグメント名

作成するセグメントの名前を指定します。

セグメント作成においては，このセグメント名の指定は省略できません。

セグメント名として，任意の長さの文字列を指定できます。ただし，次表にある予約セクションを割り当てるセグメントは，セグメント名が固定されています。それ以外のセグメント名は使用できません。

表 5―3　セグメント名が固定されている予約セクション名

セクション名	セグメント名
.sidata .sibss .tidata .tibss .tidata.byte .tibss.byte .tidata.word .tibss.word	SIDATA
.sedata .sebss	SEDATA
.sconst	SCONST

備考　.sconst は変更することはできますが，一部エラー・チェックが行われません。

(b) セグメント・タイプ

作成するセグメントのタイプを指定します。

セグメント作成においては，このセグメント・タイプの指定を省略できません。

ここで指定できるタイプは，現在のところ「メモリにロードされるセグメント・タイプ」で「LOAD」のみです。これ以外の値を指定すると，リンカはエラーを出力します。なお「LOAD」は小文字でも可です。

セグメント・タイプの指定は“!”を先頭とし，“!”の直後に空白を置かないでください。

(c) セグメント属性

作成するセグメントの属性を指定します。

セグメント作成においては，このセグメント属性の指定は省略できません。

指定できるセグメント属性とその意味は次のとおりです。

なお，セグメント属性は，そのセグメントに属するマッピング・ディレクティブの属性に依存します。指定するときはマッピング・ディレクティブで指定するセクションの属性を考慮して設定する必要があります。

表 5―4　セグメント属性とその意味

セグメント属性	意味
R	読み出し可能なセグメント
W	書き込み可能なセグメント
X	実行可能なセグメント

セグメント属性は，複数個同時に指定することができ，R，W，X の順番は任意で，空白を置かずに続けて記述します。また，セグメント属性の指定は，“?”を先頭とし，“?”の直後に空白を置かないでください。

備考 1 つのセグメント・ディレクティブにおいて，セグメント属性の指定が複数回行われた場合，リンカはエラーを出力しリンクを中止します。

例

```
SEG:    !LOAD   ?RX ?RW {};
```

(d) アドレス

作成するセグメントの開始アドレスを指定します。

セグメント作成においては，このアドレスの指定は省略できます。省略した場合，先頭のセグメントである場合は 0x0 番地，以降のセグメントは，前のセグメントの終端に続いた（アライメントを考慮した）値が開始アドレスとなります。

アドレス指定は，使用するマイコンのメモリ配置を考慮して指定する必要があります。

たとえば，V850 コアの場合，0x0 番地はリセット割り込み処理（リセット割り込みハンドラ）となっています。リセット割り込み処理を行う場合，他のセグメントが 0x0 番地に割り付かないようにアドレスを設定する必要があります。

また，V850 コアも種類ごとに搭載しているメモリの大きさが違っているので，内蔵 ROM ／ RAM の開始・終了アドレスも違っています。そのため使用するマイコンに応じて，各セグメントの配置アドレスを考慮する必要があります。各 CPU のメモリ情報などに関しては，該当 CPU のユーザーズ・マニュアルを参照してください。

アドレスの値は偶数で指定してください。奇数を指定すると，リンカは メッセージを出力し，指定されたアドレスに 1 を加えたアドレスが指定されたものとみなしてリンクを続行します。

アドレスの指定は“V”（大文字・小文字どちらでも可）を先頭とし，“V”の直後に空白を置かないでください。アドレスとして指定できる数値は 10 進数，16 進数のどちらかで，16 進数で指定する場合は数値の前に“0x”を記述してください。なお，アドレスの指定に式を用いることはできません。

(e) 最大メモリ・サイズ

作成するセグメントのメモリ・サイズの最大値を指定します。

これはセグメントが意図したサイズを越えないようにするための指定です。ですから，実際のサイズが最大メモリ・サイズとして指定した値よりも小さい場合は，次にくるセグメントは，そのセグメントの直後になります。

セグメント作成においては，この最大メモリ・サイズの指定は省略できます。省略した場合，0x100000（バイト），または，セグメントを配置するメモリ・サイズが 1M を越えるデバイスを品種指定した場合は，セグメントを配置するメモリのサイズがデフォルトの値として用いられます。

作成されたセグメントが最大メモリ・サイズで指定された値を越えた場合，リンカはエラーを出力し，リンクを中止します。

最大メモリ・サイズの指定は“L”（大文字・小文字どちらでも可）を先頭とし，“L”の直後に空白を置かないでください。なお，最大メモリ・サイズの指定に式を用いることはできません。

（f）ホール・サイズ

作成するセグメントのホールのサイズを指定します。

セグメントのホールとは，セグメントとセグメントの間の余白を意味します。ホール・サイズの指定を行った場合，そのセグメントの終わりにホールが作成されます。

セグメント作成においては，このホール・サイズの指定は省略できます。省略した場合，0x0（バイト）がデフォルトの値として用いられます（ホールは作成しません）。

ホール・サイズの指定は“H”（大文字・小文字どちらでも可）を先頭とし，“H”の直後に空白を置かないでください。

なお，ホール・サイズの指定に式を用いることはできません。

（g）フィリング値

セグメントの割り付けにおいて生じるホール，および“H”指定で明示的に作成したホールに対し，ホールの領域を埋めるのに用いる値（フィリング値）を指定します。

フィリング値の指定を用いる場合は，“-B”オプションを指定し，2 パス・モードでリンクを行ってください。デフォルトの 1 パス・モードでフィリング値の指定が用いられた場合，リンカはメッセージを出力し，この指定を無視してリンクを続行します。

セグメント作成においては，このフィリング値の指定は省略できます。省略した場合，0x0000 がデフォルトの値として用いられます（0 で埋めます）。ただし，リンカのオプションで“-f”（フィリング値指定オプション）が指定された場合，リンカは メッセージを出力し，この指定を無視してリンクを続行します。

フィリング値の指定は“F”（大文字・小文字どちらでも可）を先頭とし，“F”の直後に空白を置かないでください。フィリング値は 2 バイト 4 桁の 16 進数で指定してください。4 桁に満たない場合は，満たない桁分の 0 が指定されたものとします。また，ホールのサイズが 2 バイトに満たない場合は必要な桁数分だけ，指定されたフィリング値の下位から取り出します。なお，フィリング値の指定に式を用いることはできません。

（h）整列条件

作成するセグメントのメモリ割り付けにおいて，セグメントの整列条件（アライメント値）を指定します。

セグメント作成においては，この整列条件の指定は省略できます。省略した場合，0x8（バイト）がデフォルトの値として用いられます（8 バイトでアラインされます）。

整列条件の指定は“A”（大文字・小文字どちらでも可）を先頭とし，“A”の直後に空白を置かないでください。整列条件の値は偶数で指定してください。奇数を指定すると，リンカは メッセージを出力し，指定された値に 1 を加えた値が指定されたものとみなしてリンクを続行します。なお，整列条件の指定に式を用いることはできません。

(2) セグメント・ディレクティブの指定例

次のようなセグメントを作成する場合を例とします。

表 5―5　セグメント例

項目	値
セグメント名	PROG1
セグメント・タイプ	読み出し可能，実行可能
配置アドレス	0x1000 番地
最大メモリ・サイズ	0x200000（バイト）
ホール・サイズ	0x20（バイト）
フィリング値	0xffff
整列条件	0x16（バイト）

この場合，セグメント・ディレクティブの記述は次のようになります。

```
PROG1:  !LOAD   ?RX V0x1000 L0x200000   H0x20   F0xffff A0x16 {
        :
     （マッピング・ディレクティブ）
        :
};
```

備考　基本的に，セグメント・ディレクティブは，配置するアドレス順に記述しなくても問題ありません。ただし，.sedata セクション／.sebss セクションをもつ セグメント（デフォルトでは "SEDATA セグメント " ）に関しては，配置アドレスを省略したときにかぎり，この規則の例外となります。
CA850 では，SEDATA セグメントは 内蔵 RAM 手前の領域を ep 相対の 1 命令 参照するものとして定義しており，配置アドレスを省略すると，デバイス・ファイルに 定義された内蔵 RAM の先頭アドレスから 0x8000 を引いたアドレスを指定したものとして扱います。

たとえば，リンク・ディレクティブ・ファイル中に次の記述があったとします。

```
SIDATA: !LOAD   ?RW V0xffb000 {
    .tidata.byte    = $PROGBITS ?AW     .tidata.byte;
    .tibss.byte     = $NOBITS   ?AW     .tibss.byte;
    .tidata.word    = $PROGBITS ?AW     .tidata.word;
    .tibss.word     = $NOBITS   ?AW     .tibss.word;
    .sidata         = $PROGBITS ?AW     .sidata;
    .sibss          = $NOBITS   ?AW     .sibss;
};
SEDATA: !LOAD   ?RW {
    .sedata         = $PROGBITS ?AW     .sedata;
    .sebss          = $NOBITS   ?AW     .sebss;
};
DATA:   !LOAD   ?RW {
    .data           = $PROGBITS ?AW     .data;
    .sdata          = $PROGBITS ?AWG    .sdata;
    .sbss           = $NOBITS   ?AWG    .sbss;
    .bss            = $NOBITS   ?AW     .bss;
};
```

SEDATA のアドレスが省略されており，デバイス・ファイル情報から，この先頭アドレスは 0xff2000（= 0xffb000-0x8000）と判断します。SIDATA は 0xffb000 番地に配置定義されているため，CA850 は内部的に SEDATA を SIDATA の前に移動してリンクをします。

また，その後に定義されている DATA セグメントもアドレスが省略されているため，SEDATA セグメントの直後に配置されることになります。

5. 1. 4 マッピング・ディレクティブ

ここでは，次の項目でマッピング・ディレクティブの書式について説明します。

- 指定項目
- マッピング・ディレクティブの指定例

(1) 指定項目

マッピング・ディレクティブで指定する項目は，次のとおりです。

表 5―6 マッピング・ディレクティブで指定できる項目

項目	指定形式	意味	省略
出力セクション名	出力セクション名	ロード・モジュールに出力されるセクションの名前	不可
セクション・タイプ	$PROGBITS $NOBITS	作成するセクションのタイプ	不可

項目	指定形式	意味	省略
セクション属性	?[A][W][X][G]	作成するセクションの属性が「メモリを占有（A）」「書き込み可能（W）」「実行可能（X）」「gp と 16 ビット・ディスプレースメントで参照可（G）」であるかを指定（複数指定可能）	不可
アドレス	V アドレス	作成するセクションの先頭アドレス	可
ホール・サイズ	H ホール・サイズ	セクションの後ろに作成するホールのサイズ（次のセクションとの間に入れる余白）	可
整列条件	A 整列条件	メモリ割り付けにおける整列条件（アライメント）	可
入力セクション名	入力セクション名	出力セクションに割り付ける入力セクションの名前	可
オブジェクト・ファイル名	{ ファイル名 ファイル名 …}	入力セクションとして抽出したいセクションを含むオブジェクト・ファイル（複数指定可，スペースで区切る）	可

マッピング・ディレクティブの具体的な書式は次とおりです。

```
出力セクション名 =
    $ セクション・タイプ
    ? セクション属性
    V アドレス
    H ホール・サイズ
    A 整列条件
    入力セクション名
    { オブジェクト・ファイル名  オブジェクト・ファイル名 };
```

それぞれの項目は，空白で区切って記述します。また，終わりには，必ず“;”を付けます。

「V アドレス」「H ホール・サイズ」「A 整列条件」「入力セクション名」「オブジェクト・ファイル名」は省略できます。省略した場合はデフォルトの値が使われたり，決まった規則が用いられます。デフォルトの値，規則は次のとおりです。

表 5―7　マッピング・ディレクティブの指定項目で省略可能な値のデフォルト値／規則

項目	意味
アドレス	セグメント・ディレクティブで指定されたアドレスに従う 複数セクションがあり，先頭のセクションでなければ，前のセクションの終端に続く値 先頭のセクションならば，セグメントの先頭に続く値
ホール・サイズ	0x0（バイト）
整列条件	.tidata.byte ／ . tibss.byte セクション：0x1（バイト） 上記セクション以外：0x4（バイト）

項目	意味
入力セクション	作成する出力セクションと同じ属性を持つセクションを，すべてのオブジェクトから抽出 オブジェクト・ファイル名が指定されていれば，そのオブジェクトから抽出
オブジェクト・ファイル名	作成する出力セクションと同じ属性を持つセクションを，すべてのオブジェクトから抽出 入力セクションが指定されていれば，作成する出力セクションと同じ属性を持つすべてのオブジェクトから抽出

次にそれぞれの指定項目について詳しく説明します。

(a) 出力セクション名

ロード・モジュールに出力されるセクションの名前を指定します。セクション作成においては，この出力セクション名の指定は省略できません。

出力セクション名として，任意の長さの文字列を指定できます。

ただし，次表にあるセクションは，出力セクション名と入力セクション名の対応が固定されているので，別の名前のセクション名は指定できません。

表 5―8　セグメント名が固定されている予約セクション名

入力セクション名	出力セクション名
.tidata セクション	.tidata
.tibss セクション	.tibss
.tidata.byte セクション	.tidata.byte
.tibss.byte セクション	.tibss.byte
.tidata.word セクション	.tidata.word
.tibss.word セクション	.tibss.word
.sidata セクション	.sidata
.sibss セクション	.sibss
.sedata セクション	.sedata
.sebss セクション	.sebss
.pro_epi_runtime セクション	.pro_epi_runtime

備考　.sconst は変更することはできますが，一部エラー・チェックが行われません。

また，同一セグメント・ディレクティブ内で，複数のマッピング・ディレクティブを記述することができますが，異なるセグメント・ディレクティブであっても，同じ出力セクション名を複数個指定することはできません。同じ出力セクション名が複数個指定された場合，リンカはエラーを出力し，リンクを中止します。

(b) セクション・タイプ

出力セクションのタイプを指定します。

セクション作成においては，この出力セクションのタイプ指定は省略できません。

指定できるセクション・タイプとその意味は次のとおりです。

表 5―9　セクション・タイプとその意味

セクション・タイプ	意味
PROGBITS	オブジェクト・ファイル内に実際の値を持っているセクション → テキスト，初期値ありデータ（変数）
NOBITS	オブジェクト・ファイル内に実際の値を持っていないセクション → 初期値なしデータ（変数）

セクション・タイプの指定は，“$”を先頭とし，“$”の直後に空白を置かないでください。

また，“$”のみが指定された場合，リンカはエラーを出力し，リンクを中止します。

(c) セクション属性

作成するセクションの属性を指定します。

セクション作成においては，このセクション属性の指定は省略できません。

指定できるセクション属性とその意味は次のとおりです。

表 5―10　セクション属性とその意味

セクション属性	意味
A	メモリを占有するセクション（すべてのセクションが該当）
W	書き込み可能なセクション（RAM 上に配置するセクション）
X	実行可能なセクション（主にテキスト・セクション）
G	グローバル・ポインタ（gp）と 16 ビットのディスプレースメントを用いて参照することのできるメモリ範囲内に割り付けるセクション（.sdata, .sbss セクション）

セクション属性は，複数個同時に指定でき，A，W，X，G の順番は任意で，空白を置かずに続けて記述します。また，セクション属性の指定は“?”を先頭とし，“?”の直後に空白を置かないでください。

(d) アドレス

作成するセクションの開始アドレスを指定します。

セクション作成においては，このアドレスの指定は省略できます。省略した場合，セグメント・ディレクティブで指定されたアドレスに従います。複数セクションがあり，先頭のセクションでなければ，前のセクションの終端に続く値になります。

通常セクションのアドレス指定は，セグメントごとにまとめて行いますが，特定のセクションを特定のアドレスに割り付けたい場合に用いることができます。

.tidata.byte，.tibss.byte 以外のアドレスの値は偶数で指定してください。奇数を指定すると，リンカはメッセージを出力し，指定されたアドレスに 1 を加えたアドレスが指定されたものとみなしてリンクを続行します。

アドレスの指定は“V”（大文字・小文字どちらでも可）を先頭とし，“V”の直後に空白を置かないでください。アドレスとして指定できる数値は 10 進数，16 進数のどちらかで，16 進数で指定する場合は数値の前に“0x”を記述してください。なお，アドレスの指定に式を用いることはできません。

（e）ホール・サイズ

作成するセクションのホールのサイズを指定します。

セクションのホールとは，セクションとセクションの間の余白を意味します。ホール・サイズの指定を行った場合，そのセクションの終わりにホールが作成されます。

セクション作成においては，このホール・サイズの指定は省略できます。省略した場合，0x0（バイト）がデフォルトの値として用いられます（ホールは作成しません）。

ホール・サイズの指定は“H”（大文字・小文字どちらでも可）を先頭とし，“H”の直後に空白を置かないでください。なお，ホール・サイズの指定に式を用いることはできません。

（f）整列条件

作成するセクションのメモリ割り付けにおいて，セクションの整列条件（アライメント値）を指定します。

セクション作成においては，この整列条件の指定は省略できます。省略した場合はデフォルトの値が使用されますが，次に示すようにセクションの種類によってその値が違います。

表 5―11 セクションの種類とその整列条件のデフォルト値

セクション名	整列条件
.tidata.byte ／ .tibss.byte セクション	0x1（バイト）
上記以外のセクション	0x4（バイト）

整列条件の指定は“A”（大文字・小文字どちらでも可）を先頭とし，“A”の直後に空白を置かないでください。

整列条件の値として指定できるものは，.tidata.byte，.tibss.byte セクションの場合は偶数か奇数，その他のセクションにおいては偶数のみです。.tidata.byte，.tibss.byte 以外のセクションで奇数を指定すると，リンカは メッセージを出力し，指定された値に 1 を加えた値が指定されたものとみなしてリンクを続行します。なお，整列条件の指定に式を用いることはできません。

（g）入力セクション名

作成する出力セクションの基となる，入力セクション情報を指定します。

セクション作成においては，この入力セクション名の指定，オブジェクト・ファイル名の指定は省略できます。省略した場合，次のような記述の組み合わせにより，出力セクションに出力される情報が変化します。

表 5―12　入力セクションとオブジェクト・ファイルの組み合わせ別の出力

記述パターン		出力
(1)	入力セクション名＋オブジェクト・ファイル名	指定した入力セクションを，指定したオブジェクトから抽出して出力
(2)	入力セクション名のみ	指定した入力セクションを，すべてのオブジェクトから抽出して出力
(3)	オブジェクト・ファイル名のみ	作成する出力セクションと同じ属性を持つセクションを，指定されたオブジェクトから抽出して出力
(4)	両方とも記述しなかった場合	作成する出力セクションと同じ属性を持つセクションを，すべてのオブジェクトから抽出して出力

具体的な例を示すと次のようになります。

表 5―13　入力セクションとオブジェクト・ファイルの組み合わせの具体例

記述例	出力

SEG1: !LOAD ?RX {
  sec1 = $PROGBITS ?AX usrsec1 {file1.o};
}
	file1.o にある，usrsec1 セクションを抽出し，sec1 セクションとして出力

SEG1: !LOAD ?RX {
  sec1 = $PROGBITS ?AX usrsec1;
}
	すべてのオブジェクトから，usrsec1 セクションを抽出し，sec1 セクションとして出力

SEG1: !LOAD ?RX {
  sec1 = $PROGBITS ?AX {file1.o file2.o};
}
	file1.o と file2.o にある，$PROGBITS タイプで，属性 A，X のセクションを抽出し，sec1 セクションとして出力

SEG1: !LOAD ?RX {
  sec1 = $PROGBITS ?AX;
}
	すべてのオブジェクトから，$PROGBITS タイプで，属性 A，X のセクションを抽出し，sec1 セクションとして出力

なお，セクションの配置時に候補が複数ある場合では，「表 5―12　入力セクションとオブジェクト・ファイルの組み合わせ別の出力」内の [記述パターン] 項目で示した番号を優先順位として（同順位の場合は低位アドレス優先），セクションを配置します。

入力セクション名は，アプリケーションで設定したセクション名を指定してください。特にアプリケーション中で設定しなかった場合は，デフォルトのセクション名として定義されていますので，そのセクション名を指定してください。

なお，「(a) 出力セクション名」で説明したように，出力セクション名と入力セクション名の対応が固定されているものがあります。これに該当するものは，別の名前のセクション名を指定することはできません。

(h) オブジェクト・ファイル名

オブジェクト・ファイル名の指定は，マッピング・ディレクティブの最後に記述し，ファイル名を“{”と“}”で囲んでください。また複数指定するときは「空白」で区切ります（ファイル名に「空白」が含まれている場合は，ファイル名を“”で囲みます）。

オブジェクト・ファイルを複数指定した場合，指定した順で，下位アドレスから上位アドレスの方向に割り付けられます。ただし，リンカ起動時に指定する「リンクするオブジェクト」の記述順が，リンク・ディレクティブ・ファイルにおける順番と異なる場合，引数で指定したファイル名の順番が優先されます。

```
リンク・ディレクティブ
sec = $PROGBITS ?AX {file1.o file2.o file3.o}

リンカ起動
ld850 file3.o file1.o file2.o
  → file3.o，file1.o，file2.o の順番で，下位から割り付く
```

マッピング・ディレクティブにおいて，オブジェクト・ファイル名を指定する場合，その属性のセクションを含んでいるファイル名はすべて記述してください。

たとえば，file1.o，file2.o，file3.o，file4.o の 4 つのオブジェクトが存在し，これらすべてに text 属性のセクションが含まれていたとします。このとき

```
TEXT1:  !LOAD   ?RX {
    .text1  = $PROGBITS ?AX {file1.o file2.o};
};
TEXT2:  !LOAD   ?RX {
    .text2  = $PROGBITS ?AX {file3.o};
};
```

というリンク・ディレクティブを記述し，file4.o の text 属性の配置場所を特定しなかった場合，file4.o 内の text 属性を，適当な text 属性のセクションを探して配置します。したがって，意図したとおりのマッピングにならない可能性がありますので注意が必要です（他のどのセクションにも配置されない場合，リンカはメッセージを出力します）。

また，異なるディレクトリに配置した同名ファイルを指定したい場合には，リンク・マップに表示されたパス付きファイル名で次のように指定することができます。

```
textsec1    = $PROGBITS ?AX {c:\work\dir1\file1.o};
textsec2    = $PROGBITS ?AX {c:\work\dir2\file1.o};
textsec3    = $PROGBITS ?AX {file1.o};
```

上記の場合，指定したディレクトリに存在する file1.o は，それぞれ textsec1 ／ textsec2 に配置され，それ以外は textsec3 に配置されます。なお，この際のパスの指定方法は，リンク・マップに表示された形式のみであるため，記述の際に注意が必要です。

さらに，ライブラリなど，アーカイブ・ファイル中のオブジェクトを，入力オブジェクト名に指定することもできます。たとえば libusr.a というアーカイブ・ファイル中にある lib1.o というオブジェクトを usrlib セクションに出力したい場合，次のように記述します。

```
usrlib  = $PROGBITS ?AX {lib1.o(a:\usrlib\libusr.a)};
```

指定したライブラリ内のオブジェクトすべてを配置指定したい場合は，次のように記述します。

```
usrlib  = $PROGBITS ?AX {libusr.a};
```

上記の場合，libusr.a 内のオブジェクトは usrlib セクションに配置されます。

(i) 同じ指定の場合

複数のセグメントに対して，同じセクション・タイプ／セクション属性／入力セクション名（省略可）／入力ファイル名（入力可）が指定された場合で，それに対応するセクションが存在した場合には，低位アドレスに配置されたセグメントに対し割り付けが行われます。

```
TEXT1:  !LOAD   ?RX V0x1000 {
    .text1  = $PROGBITS ?AX .text   {file1.o file2.o};
};
TEXT2:  !LOAD   ?RX V0x2000 {
    .text2  = $PROGBITS ?AX .text   {file1.o file2.o};
};
```

上記の場合，セクション・タイプ／セクション属性／入力セクション名／入力ファイル名が TEXT1 と TEXT2 で同じであるため，低位アドレスに配置されている TEXT1 に対して割り付けします。

(2) マッピング・ディレクティブの指定例

次のような出力セクションを作りたい場合を例とし，2 種類のセクションを作るとします。

表 5―14　マッピング・ディレクティブの指定例

項目	値 － 1	値 － 2
出力セクション名	.text	textsec1
セクション・タイプ	テキスト	テキスト
セクション属性	読み出し可能，実行可能	読み出し可能，実行可能
ホール・サイズ	0x10（バイト）	0x20（バイト）
フィリング値	0xffff	0xffff

項目	値 － 1	値 － 2
整列条件	0x10（バイト）	0x10（バイト）
入力セクション名	.text	usrsec1
オブジェクト・ファイル名	main.o	－

この場合，マッピング・ディレクティブの記述は次のようになります。

```
.text       = $PROGBITS ?AX H0x10   F0xffff A0x10   .text   {main.o};
textsec1    = $PROGBITS ?AX H0x20   F0xffff A0x10   usrsec1;
```

5. 1. 5　シンボル・ディレクティブ

ここでは，次の項目でシンボル・ディレクティブの書式について説明します。

- 指定項目
- シンボル・ディレクティブの指定例

（1）指定項目

シンボル・ディレクティブで指定する項目は，次のとおりです。

- tp シンボル

表 5―15　tp シンボル作成で指定できる項目

項目	指定形式	意味	省略
シンボル名	シンボル名	作成する tp シンボルの名前	不可
シンボル種別	%TP_SYMBOL	作成するシンボル種別（固定）	不可
アドレス	V アドレス	作成する tp シンボルのアドレス	可
整列条件	A 整列条件	シンボル値の整列条件（アライメント）	可
セグメント名	{ セグメント名 セグメント名 …}	作成する tp の参照対象としたいセグメント名（複数指定可，空白で区切る）	可

具体的な書式は次のようになります。

```
シンボル名 @%TP_SYMBOL　V アドレス　A 整列条件　{ セグメント名 セグメント名 };
```

それぞれの項目は，空白で区切って記述します。また終わりには，必ず“;”を付けます。
「V アドレス」「A 整列条件」「セグメント名」は省略できます。省略した場合はデフォルトの値が使われます。デフォルトの値は次のとおりです。

表 5―16　tp シンボルのデフォルト値

項目	意味
アドレス	セグメント名が指定されていた場合，そのセグメント内で最下位に割り付けられた text 属性のセクションの先頭アドレス セグメント名が指定されていない場合，ロード・モジュールに存在する text 属性セグメント内で，最下位に割り付けられた text 属性のセクションの先頭アドレス
整列条件	0x4（バイト）
セグメント名	オブジェクトに存在するすべての text 属性セグメントを対象とします。

- gp シンボル

表 5―17　gp シンボル作成で指定できる項目

項目	指定形式	意味	省略
シンボル名	シンボル名	作成する gp シンボルの名前	不可
シンボル種別	%GP_SYMBOL	作成するシンボル種別（固定）	不可
ベース・シンボル名	& ベース・シンボル名	gp シンボルを tp シンボルからのオフセット値として指定する際の tp シンボル名	可
アドレス	V アドレス	作成する gp シンボルのアドレス	可
整列条件	A 整列条件	シンボル値の整列条件（アライメント）	可
セグメント名	{ セグメント名 セグメント名 …}	作成する gp の参照対象としたいセグメント名（複数指定可，空白で区切る）	可

具体的な書式は次のようになります。

```
シンボル名 @%GP_SYMBOL　& ベース・シンボル名　V アドレス　A 整列条件　{ セグメント名　セグメント名 };
```

それぞれの項目は，空白で区切って記述します。また終わりには，必ず“;”を付けます。
「& ベース・シンボル名」「V アドレス」「A 整列条件」「セグメント名」は省略できます。省略した場合はデフォルトの値が使われます。デフォルトの値は次のとおりです。

表 5―18　gp シンボルのデフォルト値

項目	意味
ベース・シンボル名	tp シンボルからのオフセットではなく，gp シンボル値として決定されるアドレス

項目	意味
アドレス	リンカが下記事項から gp シンボル値を決定します。 - sdata 属性／ sbss 属性／ data 属性／ bss 属性のセクションの有無 - ベース・シンボル指定の有無
整列条件	0x4（バイト）
セグメント名	オブジェクトに存在するすべての sdata ／ data ／ sbss ／ bss 属性セクションを含むセグメントを対象とします。

- ep シンボル

表 5―19　ep シンボル作成で指定できる項目

項目	指定形式	意味	省略
シンボル名	シンボル名	作成する ep シンボルの名前	不可
シンボル種別	%EP_SYMBOL	作成するシンボル種別（固定）	不可
アドレス	V アドレス	作成する ep シンボルのアドレス	可
整列条件	A 整列条件	シンボル値の整列条件（アライメント）	可

具体的な書式は次のようになります。

```
シンボル名 @%EP_SYMBOL　V アドレス　A 整列条件 ;
```

それぞれの項目は，空白で区切って記述します。また終わりには，必ず“;”を付けます。
「V アドレス」「A 整列条件」は省略できます。省略した場合はデフォルトの値が使われます。デフォルトの値は次のとおりです。

表 5―20　ep シンボルのデフォルト値

項目	意味
アドレス	リンカが下記事項から ep シンボル値を決定します。 - SIDATA セグメントの有無 - デバイス・ファイル中の内蔵 RAM 領域の定義の有無
整列条件	0x4（バイト）

次にそれぞれの指定項目について詳しく説明します。

（a）シンボル名【 該当シンボル：tp，gp，ep 】

生成するシンボルの名前を指定します。シンボル作成においては，このシンボル名の指定は省略できません。

シンボル名として，任意の長さの文字列を指定できます。

（b）シンボル種別【 該当シンボル：tp，gp，ep 】

tp シンボルを生成するか，gp シンボルを生成するか，ep シンボルを生成するかを指定します。シンボル作成においては，このシンボル種別の指定は省略できません。

ここで指定できる種別は「tp シンボル」「gp シンボル」「ep シンボル」のどれかで，それぞれ「TP_SYMBOL」「GP_SYMBOL」「EP_SYMBOL」です。これ以外の値を指定すると，リンカはエラーを出力します。

シンボル種別の指定は“%”を先頭とし，“%”の直後に空白を置かないでください。

（c）ベース・シンボル名【 該当シンボル：gp 】

gp シンボルの生成において，gp シンボル値を定める際に用いる tp シンボルを指定します。ベース・シンボル名を指定すると，tp シンボル値からのオフセット値が gp シンボル値となります。

gp シンボル作成においては，このベース・シンボル名の指定は省略できます。

ベース・シンボルの指定は“&”を先頭とし，“&”の直後に空白を置かないでください。“&”の後にベースとしたい tp シンボル名を記述します。

（d）アドレス【 該当シンボル：tp，gp，ep 】

tp シンボル値，gp シンボル値，つまり，アドレスを指定します。

シンボル作成においては，このアドレスの指定は省略できます。アドレスを省略した場合は，次のように決定されます。

表 5―21　tp シンボル，gp シンボル，ep シンボルのアドレス指定

シンボル値	決定規則
tp シンボル	- セグメント名が指定されていた場合 そのセグメント内で最下位に割り付けられた text 属性のセクションの先頭アドレス - セグメント名が指定されていない場合 ロード・モジュールに存在する text 属性セグメント内で，最下位に割り付けられた text 属性のセクションの先頭アドレス
gp シンボル	リンカが下記事項を調べ，gp シンボル値を決定します。 - sdata 属性／ sbss 属性／ data 属性／ bss 属性のセクションの有無 - ベース・シンボル指定の有無
ep シンボル	リンカが下記事項から ep シンボル値を決定します。 - SIDATA セグメントの有無 - デバイス・ファイル中の内蔵 RAM 領域の定義の有無

アドレスの指定は“V”（大文字・小文字どちらでも可）を先頭とし，“V”の直後に空白を置かないでください。

(e) 整列条件【 該当シンボル：tp，gp，ep 】

作成する tp シンボル値，gp シンボル値，ep シンボル値の設定における整列条件（アライメント値）を指定します。

シンボルの作成においては，この整列条件の指定は省略できます。省略した場合はデフォルトの値が使用されます。デフォルト値は 0x4（バイト）です。

整列条件の指定は“A”（大文字・小文字どちらでも可）を先頭とし，“A”の直後に空白を置かないでください。整列条件の値は偶数で指定してください。奇数を指定すると，リンカは メッセージを出力し，指定された値に 1 を加えた値が指定されたものとみなしてリンクを続行します。なお，整列条件の指定に式を用いることはできません。

(f) セグメント名【 該当シンボル：tp，gp 】

作成する tp シンボル値，gp シンボル値の参照対象とするセグメント名を指定します。

つまり，作成する tp シンボル，gp シンボルで参照したいセグメントを指定します。参照対象のセグメントは複数指定することができます。

シンボルの作成においては，このセグメント名の指定は省略できます。省略した場合はデフォルトとして，次のように指定されたものとみなされます。

表 5―22　tp シンボル，gp シンボルの参照対象セグメント名

シンボル値	決定規則
tp シンボル	オブジェクトに存在するすべての text 属性セグメントを対象とします。
gp シンボル	オブジェクトに存在するすべての sdata / data / sbss / bss 属性セクションを含むセグメントを対象とします。

なお，gp シンボル参照の対象とするセグメント名には，gp 相対参照が前提となっているセグメント名を指定してください。

たとえば，ep 相対参照が前提となっている .sedata や .sebss セクションを含むセグメントは指定しないでください。

セグメント名の指定は，シンボル・ディレクティブの最後に記述し，セグメント名“{”と“}”で囲んでください。また複数指定するときは「空白」で区切ります。

(2) シンボル・ディレクティブの指定例

次のようなシンボルを作成する場合とします。

表 5―23　シンボル・ディレクティブの指定例

シンボル	指摘項目	指定値
tp シンボル	シンボル名	__tp_TEXT
	参照対象セグメント名	TEXT1

シンボル	指摘項目	指定値
gp シンボル	シンボル名	__gp_DATA
	オフセット指定シンボル	__tp_TEXT
	参照対象セグメント名	DATA1，DATA2
ep シンボル	シンボル名	__ep_DATA
	アドレス	0xffffd000

この場合，シンボル・ディレクティブの記述は次のようになります。

```
__tp_TEXT@%TP_SYMBOL    {TEXT1};
__gp_DATA@%GP_SYMBOL    &__tp_TEXT  {DATA1 DATA2};
__ep_DATA@%EP_SYMBOL    V0xFFFFD000;
```

なお，シンボル・ディレクティブの指定がされていなかった場合，シンボルは作成されないので注意が必要です。

5.2 予約語

リンク・ディレティブ・ファイルには予約語が存在します。予約語を指定された用途以外に使用することはできません。

予約語は次のとおりです。

- セグメント名（SIDATA，SEDATA，SCONST）
- セグメント・タイプ（LOAD）
- 出力セクション名（.tidata，.tibss など）
- セクション・タイプ（PROGBITS，NOBITS）
- シンボル種別（TP_SYMBOL，GP_SYMBOL，EP_SYMBOL）

第 6 章　関数仕様

この章では，CA850 が提供するライブラリ関数について説明します。

6.1　提供ライブラリ

CA850 で提供しているライブラリは，次のとおりです。

表 6―1　提供ライブラリ

提供ライブラリ	ライブラリ名	概要
標準ライブラリ	libc.a	可変個引数関数 文字列関数 メモリ管理関数 文字変換関数 文字分類関数 標準入出力関数 標準ユーティリティ関数 非局所分岐関数 ランタイム・ライブラリ 関数のプロローグ／エピローグ・ランタイム・ライブラリ
数学ライブラリ	libm.a	数学関数
ROM 化用ライブラリ	libr.a	コピー関数

標準ライブラリや数学ライブラリを，アプリケーション内で使用するときは，関連するヘッダ・ファイルをインクルードして，ライブラリ関数を使用します。

また，リンカ・オプション（-l）で，これらのライブラリを参照するようにします。

ただし，可変個引数関数，文字変換関数，文字分類関数だけを使用している場合，ライブラリの参照を行う必要はありません。

CubeSuite+ を使用する場合，これらのライブラリはデフォルトで参照する設定になっています。

また，数学ライブラリは，内部で標準ライブラリを参照しているため，数学ライブラリを使用する場合は，標準ライブラリも必要となります。

ランタイム・ライブラリは，標準ライブラリの一部ですが，浮動小数点演算や整数演算（32 ビット整数乗除算や剰余算）を行うときに，CA850 が自動的に呼び出すルーチンです。また，関数のエピローグ／プロローグ・ランタイム・ライブラリも，標準ライブラリの一部ですが，関数のプロローグ／エピローグ処理で CA850 が自動的に呼び出すルーチンです。

したがって，“ランタイム・ライブラリ”，および“関数のエピローグ／プロローグ・ランタイム・ライブラリ”の 2 つは，他のライブラリ関数とは異なり，C 言語ソースやアセンブリ言語ソースで記述する関数ではありません。

また，32 レジスタ・モードを使用し，さらにマスク・レジスタ機能を使用する場合は，標準ライブラリとして“マスク・レジスタ用フォルダ（*Install Folder* \ lib850 \ r32msk）”に格納されているものを使用します。

リンカは，次の場合は，自動的に上記のフォルダにある標準ライブラリを参照します。

- 32 レジスタ・モードを指定
- コンパイル・オプション“-Xmask_reg”でマスク・レジスタ機能を使用している

ROM 化用ライブラリは，コンパイラ・オプション“-Xr”を指定すると，リンカが参照するライブラリです。パッキングされたデータをコピーする関数（_rcopy，_rcopy1，_rcopy2，_rcopy4）が格納されています。

以降に各ライブラリについて記します。
なお，表中の各要素は，以下の意味を持ちます。

関数／マクロ名	関数／マクロの名前です。
概要	関数／マクロの機能概要です。
#include	この関数／マクロを使用するうえで，C 言語ソースでインクルードする必要あるヘッダ・ファイルです。#include 指令でインクルードしてください。 例外発生時の errno を使用する場合は“errno.h”もインクルードする必要があります。
ANSI	この関数が ANSI 規格で規定されている関数かどうかの区別です。 規定されていれば“○”，規定されていなければ“×”がついています。
sdata	この関数／マクロが，メモリ領域“sdata 領域”を使用するかどうかの区別です。 つまり，関数が初期値を持つデータを RAM に配置しているかどうかを区別します。セクション名は“.sdata”でなくてはならないため，ユーザ・アプリケーションでこの領域を使用していなくても，“.sdata セクション”を生成してください。 .sdata セクションを使用する場合“○”，使用しない場合“×”がついています。“○”の場合，初期値ありデータを持つ必要があるため，初期値をプログラム実行前に RAM にコピーする必要があります。つまり，“コピー関数”を用いて ROM 化処理を行う必要があります。
sbss	この関数／マクロが，メモリ領域“sbss 領域”を使用するかどうかの区別です。 つまり，関数がテンポラリとして RAM を使用するかどうかを区別します。セクション名は“.sbss”でなくてはならないため，ユーザ・アプリケーションでこの領域を使用していなくても，“.sbss セクション”を生成してください。 .sbss セクションを使用する場合“○”，使用しない場合“×”がついています。.sbss セクションは，初期値を持たないデータが配置されるので，“.sdata 使用”のときと違って ROM 化処理を行う必要はありません。
Reent	“リエントラント性”を持つかどうかの区別です。 リエントラント性がある場合“○”，ない場合“×”がついています。 “リエントラント”とは“再入可能”という意味です。リエントラント性を持つ関数は，その関数実行中に，他のプロセスでもその関数実行しようとした場合でも，正しく実行できます。たとえば，リアルタイム OS を用いたアプリケーションで，あるタスクがこの関数を実行している最中に，割り込みなどがトリガになって他のタスクへディスパッチし，そこでもこの関数を実行しても正しく実行されます。関数が RAM をテンポラリとして使う必要のある関数は，リエントラント性を持たないことがあります。

6. 1. 1 標準ライブラリ

標準ライブラリに収録されている関数を示します。収録ライブラリは“libc.a”です。

(1) 可変個引数関数

表 6―2 可変個引数関数

関数／マクロ名	#include	ANSI	sdata	sbss	Reent
va_start	stdarg.h	○	×	×	–
va_end	stdarg.h	○	×	×	–
va_arg	stdarg.h	○	×	×	–

(2) 文字列関数

表 6―3 文字列関数

関数／マクロ名	#include	ANSI	sdata	sbss	Reent
index	string.h	×	×	×	○
strpbrk	string.h	○	×	×	○
rindex	string.h	×	×	×	○
strrchr	string.h	○	×	×	○
strchr	string.h	○	×	×	○
strstr	string.h	○	×	×	○
strspn	string.h	○	×	×	○
strcspn	string.h	○	×	×	○
strcmp	string.h	○	×	×	○
strncmp	string.h	○	×	×	○
strcpy	string.h	○	×	×	○
strncpy	string.h	○	×	×	○
strcat	string.h	○	×	×	○
strncat	string.h	○	×	×	○
strtok	string.h	○	×	○	×
strlen	string.h	○	×	×	○
strerror	string.h	○	○	×	×

(3) メモリ管理関数

表 6―4 メモリ管理関数

関数／マクロ名	#include	ANSI	sdata	sbss	Reent
memchr	string.h	○	×	×	○
memcmp	string.h	○	×	×	○
bcmp	string.h	×	×	×	○
memcpy	string.h	○	×	×	○
bcopy	string.h	×	×	×	○
memmove	string.h	○	×	×	○
memset	string.h	○	×	×	○

(4) 文字変換関数

表 6―5 文字変換関数

関数／マクロ名	#include	ANSI	sdata	sbss	Reent
toupper	ctype.h	○	○	×	○
_toupper	ctype.h	×	×	×	○
tolower	ctype.h	○	○	×	○
_tolower	ctype.h	×	×	×	○
toascii	ctype.h	×	×	×	○

(5) 文字分類関数

表 6―6 文字分類関数

関数／マクロ名	#include	ANSI	sdata	sbss	Reent
isalnum	ctype.h	○	○	×	○
isalpha	ctype.h	○	○	×	○
isascii	ctype.h	×	×	×	○
isupper	ctype.h	○	○	×	○
islower	ctype.h	○	○	×	○
isdigit	ctype.h	○	○	×	○
isxdigit	ctype.h	○	○	×	○
iscntrl	ctype.h	○	○	×	○
ispunct	ctype.h	○	○	×	○
isspace	ctype.h	○	○	×	○
isprint	ctype.h	○	○	×	○

関数／マクロ名	#include	ANSI	sdata	sbss	Reent
isgraph	ctype.h	○	○	×	○

(6) 標準入出力関数

表 6―7 標準入出力関数

関数／マクロ名	#include	ANSI	sdata	sbss	Reent
fread	stdio.h	○	○	×	×
getc	stdio.h	○	○	×	×
fgetc	stdio.h	○	○	×	×
fgets	stdio.h	○	○	×	×
fwrite	stdio.h	○	○	×	×
putc	stdio.h	○	○	×	×
fputc	stdio.h	○	○	×	×
fputs	stdio.h	○	○	×	×
getchar	stdio.h	○	○	×	×
gets	stdio.h	○	○	×	×
putchar	stdio.h	○	○	×	×
puts	stdio.h	○	○	×	×
sprintf	stdio.h	○	○	○	×
fprintf	stdio.h	○	○	○	×
vsprintf	stdio.h	○	○	○	×
printf	stdio.h	○	○	○	×
vfprintf	stdio.h	○	○	○	×
vprintf	stdio.h	○	○	○	×
sscanf	stdio.h	○	○	○	×
fscanf	stdio.h	○	○	○	×
scanf	stdio.h	○	○	○	×
ungetc	stdio.h	○	○	×	×
rewind	stdio.h	○	○	×	×
perror	stdio.h	○	○	×	×注

注 stderr に関してリエントラント性は，持ちません。

(7) 標準ユーティリティ関数

表 6―8 標準ユーティリティ関数

関数／マクロ名	#include	ANSI	sdata	sbss	Reent
abs	stdlib.h	○	×	×	○
labs	stdlib.h	○	×	×	○
bsearch	stdlib.h	○	×	×	○
qsort	stdlib.h	○	×	×	○
div	stdlib.h	○	×	×	○
ldiv	stdlib.h	○	×	×	○
itoa	stdlib.h	×	×	×	○
ltoa	stdlib.h	×	×	×	○
ultoa	stdlib.h	×	×	×	○
ecvtf	stdlib.h	×	○	○	×
fcvtf	stdlib.h	×	○	○	×
gcvtf	stdlib.h	×	○	○	×
atoi	stdlib.h	○	○	×	×注 2
atol	stdlib.h	○	○	×	×注 2
strtol	stdlib.h	○	○	○	×注 2
strtoul	stdlib.h	○	○	○	×注 2
atoff	stdlib.h	○	○	×	×注 2
strtodf	stdlib.h	○	○	○	×注 2
calloc	stdlib.h	○	○	○	×注 1
malloc	stdlib.h	○	○	○	×注 1
realloc	stdlib.h	○	○	○	×注 1
free	stdlib.h	○	○	○	×注 1
rand	stdlib.h	○	○	×	×
srand	stdlib.h	○	○	×	×

注 1. 再帰呼び出しはできます。

2. 例外発生時に errno が更新されたときは，リエントラント性は持ちません。

備考 例外発生時の errno を使用する場合は，errno.h をインクルードする必要があります

(8) 非局所分岐関数

表 6―9 非局所分岐関数

関数／マクロ名	#include	ANSI	sdata	sbss	Reent
longjmp	setjmp.h	○	×	×	○
setjmp	setjmp.h	○	×	×	○

備考 例外発生時の errno を使用する場合は“errno.h”を，“汎整数型の各種限界値”をマクロ名で使用する場合は“limits.h”を，浮動小数点型の各種限界値を使用する場合は“float.h”をインクルードする必要があります。

(9) ランタイム・ライブラリ

ランタイム・ライブラリは，C 言語ソース・プログラムで浮動小数点演算や整数演算（32 ビット整数乗除算や乗余算）を行うときに，CA850 が自動的に呼び出す関数です。ランタイム・ライブラリは，“関数のエピローグ／プロローグ・ランタイム・ライブラリ”と同様に，C 言語ソースやアセンブリ言語ソースで記述する関数ではありません。

表 6―10 ランタイム・ライブラリ

関数／マクロ名	sdata	sbss	Reent
__mul	×	×	○
__mulu	×	×	○
__div	×	×	○
__divu	×	×	○
__mod	×	×	○
__modu	×	×	○
__addf.s	○	×	△注
__subf.s	○	×	△注
__mulf.s	○	×	△注
__divf.s	○	×	△注
__cvt.ws	×	×	○
__trnc.sw	×	×	○
__cmpf.s	○	×	△注

注 △：例外発生時の errno の更新と matherr の呼び出しに関してのみリエントラント性なし

備考 例外発生時の errno を使用する場合は“errno.h”を，“汎整数型の各種限界値”をマクロ名で使用する場合は“limits.h”を，浮動小数点型の各種限界値を使用する場合は“float.h”をインクルードする必要があります。

（10）関数のプロローグ／エピローグ・ランタイム・ライブラリ

関数のエピローグ／プロローグ・ランタイム・ライブラリは，関数のプロローグ／エピローグ処理で CA850 が自動的に呼び出すルーチンです。関数のエピローグ／プロローグ・ランタイム・ライブラリは，“ランタイム・ライブラリ”と同様に，C 言語ソースやアセンブリ言語ソースで記述する関数ではありません。

V850Ex コアの場合，関数のプロローグ／エピローグ・ランタイム・ライブラリの呼び出しに CALLT 命令を使用します。これらの関数を CALLT 命令によるテーブル呼び出しにすることにより，コード効率を上げます。

関数のプロローグ／エピローグ・ランタイム・ライブラリ呼び出しを有効にするためには，次のようになります。

- 最適化オプション“-Ot（実行速度優先最適化）”指定時以外のとき
- コンパイラ・オプション“-Xpro_epi_runtime=on”を指定したとき

表 6―11 関数のプロローグ／エピローグ・ランタイム・ライブラリ

関数／マクロ名	概要
___push2000, ___push2001, ___push2002, ___push2003, ___push2004, ___push2040, ___push2100, ___push2101, ___push2102, ___push2103, ___push2104, ___push2140, ___push2200, ___push2201, ___push2202, ___push2203, ___push2204, ___push2240, ___push2300, ___push2301, ___push2302, ___push2303, ___push2304, ___push2340, ___push2400, ___push2401, ___push2402, ___push2403, ___push2404, ___push2440, ___push2500, ___push2501, ___push2502, ___push2503, ___push2504, ___push2540, ___push2600, ___push2601, ___push2602, ___push2603, ___push2604, ___push2640, ___push2700, ___push2701, ___push2702, ___push2703, ___push2704, ___push2740, ___push2800, ___push2801, ___push2802, ___push2803, ___push2804, ___push2840, ___push2900, ___push2901, ___push2902, ___push2903, ___push2904, ___push2940, ___pushlp00, ___pushlp01, ___pushlp02, ___pushlp03, ___pushlp04, ___pushlp40	関数のプロローグ処理

関数／マクロ名	概要
___Epush250, ___Epush251, ___Epush252, ___Epush253, ___Epush254, ___Epush260, ___Epush261, ___Epush262, ___Epush263, ___Epush264, ___Epush270, ___Epush271, ___Epush272, ___Epush273, ___Epush274, ___Epush280, ___Epush281, ___Epush282, ___Epush283, ___Epush284, ___Epush290, ___Epush291, ___Epush292, ___Epush293, ___Epush294, ___Epushlp0, ___Epushlp1, ___Epushlp2, ___Epushlp3, ___Epushlp4	関数のプロローグ処理【V850E】
___pop2000, ___pop2001, ___pop2002, ___pop2003, ___pop2004, ___pop2040, ___pop2100, ___pop2101, ___pop2102, ___pop2103, ___pop2104, ___pop2140, ___pop2200, ___pop2201, ___pop2202, ___pop2203, ___pop2204, ___pop2240, ___pop2300, ___pop2301, ___pop2302, ___pop2303, ___pop2304, ___pop2340, ___pop2400, ___pop2401, ___pop2402, ___pop2403, ___pop2404, ___pop2440, ___pop2500, ___pop2501, ___pop2502, ___pop2503, ___pop2504, ___pop2540, ___pop2600, ___pop2601, ___pop2602, ___pop2603, ___pop2604, ___pop2640, ___pop2700, ___pop2701, ___pop2702, ___pop2703, ___pop2704, ___pop2740, ___pop2800, ___pop2801, ___pop2802, ___pop2803, ___pop2804, ___pop2840, ___pop2900, ___pop2901, ___pop2902, ___pop2903, ___pop2904, ___pop2940, ___poplp00, ___poplp01, ___poplp02, ___poplp03, ___poplp04, ___poplp40	関数のエピローグ処理
___Epop250, ___Epop251, ___Epop252, ___Epop253, ___Epop254, ___Epop260, ___Epop261, ___Epop262, ___Epop263, ___Epop264, ___Epop270, ___Epop271, ___Epop272, ___Epop273, ___Epop274, ___Epop280, ___Epop281, ___Epop282, ___Epop283, ___Epop284, ___Epop290, ___Epop291, ___Epop292, ___Epop293, ___Epop294, ___Epoplp0, ___Epoplp1, ___Epoplp2, ___Epoplp3, ___Epoplp4	関数のエピローグ処理【V850E】

6. 1. 2 数学ライブラリ

数学ライブラリに収録されている関数を示します。収録ライブラリは“libm.a”です。

（1）数学関数

表 6―12 数学関数

関数／マクロ名	#include	ANSI	sdata	sbss	Reent
j0f	math.h	×	○	○	△注
j1f	math.h	×	○	○	△注
jnf	math.h	×	○	○	△注
y0f	math.h	×	○	○	△注
y1f	math.h	×	○	○	△注
ynf	math.h	×	○	○	△注
erff	math.h	×	○	○	△注
erfcf	math.h	×	○	○	△注
expf	math.h	○	○	○	△注
logf	math.h	○	○	○	△注
log2f	math.h	○	○	○	△注
log10f	math.h	○	○	○	△注
powf	math.h	○	○	○	△注
sqrtf	math.h	○	○	○	△注
cbrtf	math.h	×	×	×	○
ceilf	math.h	○	×	×	○
fabsf	math.h	○	×	×	○
floorf	math.h	○	×	×	○
fmodf	math.h	○	○	○	△注
frexpf	math.h	○	○	○	△注
ldexpf	math.h	○	○	○	△注
modff	math.h	○	×	×	○
gammaf	math.h	×	○	○	△注
hypotf	math.h	×	○	○	△注
matherr	math.h	×	×	×	○
cosf	math.h	○	○	○	△注
sinf	math.h	○	○	○	△注
tanf	math.h	○	○	○	△注
acosf	math.h	○	○	○	△注
asinf	math.h	○	○	○	△注
atanf	math.h	○	○	○	△注

関数／マクロ名	#include	ANSI	sdata	sbss	Reent
atan2f	math.h	○	○	○	△注
coshf	math.h	○	○	○	△注
sinhf	math.h	○	○	○	△注
tanhf	math.h	○	○	○	△注
acoshf	math.h	×	○	○	△注
asinhf	math.h	×	○	○	△注
atanhf	math.h	×	○	○	△注

注 △：例外発生時の errno の更新と matherr の呼び出しに関してのみリエントラント性なし

備考 例外発生時の errno を使用する場合は“errno.h”を，“汎整数型の各種限界値”をマクロ名で使用する場合は“limits.h”を，浮動小数点型の各種限界値を使用する場合は“float.h”をインクルードする必要があります。

6. 1. 3 ROM 化用ライブラリ

ROM 化用ライブラリに収録されている関数を示します。収録ライブラリは“libr.a”です。

初期値ありデータや，プログラム・コードを RAM にコピーするためのルーチンです。

- ROM 化用の関数自体は sdata 領域，sbss 領域は使用しません。sdata 領域にデータを書き込む動作は行います。
- ROM 化用の関数は，通常メイン・プログラムを実行する前に一度だけ呼び出します。そのため，リエントラント性に関しては考慮しません。
- インサーキット・エミュレータ（ICE）にロード・モジュールをダウンロードした場合，data 領域や sdata 領域に置かれる初期値ありデータは，ダウンロードした時点でセットされます。
  したがって，コピー関数を呼び出さなくても，デバッグが可能になります。しかし，ROM 化用ロード・モジュールを作成して実機で動作させる場合，コピー関数で初期値ありデータ・コピー等を行わないと，初期値が設定されずに期待した動作をしません。「インサーキット・エミュレータで動作したが，実機でうまく動作しない」という場合，その原因のほとんどは，このコピー関数で初期値を設定していないためです。また，初期化時に，RAM をゼロクリアする処理を行っているような場合は，そのルーチンよりも後にコピー関数を呼び出してください。初期値もゼロクリアされてしまうためです。

（1）コピー関数

表 6―13 コピー関数

関数／マクロ名	概要
_rcopy	パッキング・データを 1 バイトずつ RAM へコピーする（_rcopy1 と同じ）
_rcopy1	パッキング・データを 1 バイトずつ RAM へコピーする（_rcopy と同じ）
_rcopy2	パッキング・データを 2 バイトずつ RAM へコピーする
_rcopy4	パッキング・データを 4 バイトずつ RAM へコピーする

備考 _rcopy と _rcopy1 はどちらもまったく同じ動作をします。前版との互換性のために用意しています。内蔵命令 RAM を持っている V850（V850E/ME2 など）で，プログラム・コードを内蔵命令 RAM にコピーする場合，ハードウエアの仕様上，4 バイト単位でコピーする必要があります。その場合，_rcopy4 を使ってコピーします。ハードウエア的な制限がなければ，どの関数を使っても問題ありませんが，2 バイト，4 バイト単位でコピーする場合は，コピーの必要がある領域を越える（パッキングされた領域のサイズが 4 の倍数でなかった場合，パッキング領域外の領域も同時にコピーされる）可能性があるため，注意が必要です。

6.2 ヘッダ・ファイル

CA850 でライブラリを使用するときに必要となるヘッダ・ファイルの一覧は次のとおりです。
なお，各ファイルにはマクロ定義，関数宣言が記述されています。

表 6―14 ヘッダ・ファイル

ファイル名	概要
ctype.h	文字の変換，分類のためのヘッダ・ファイル
errno.h	エラー条件の報告のためのヘッダ・ファイル
float.h	浮動小数点表現，浮動小数点演算のためのヘッダ・ファイル
limits.h	整数の数量的限界のためのヘッダ・ファイル
math.h	数学計算のためのヘッダ・ファイル
setjmp.h	非局所分岐のためのヘッダ・ファイル
stdarg.h	可変個の引数を持つ関数をサポートするためのヘッダ・ファイル
stddef.h	共通の定義のためのヘッダ・ファイル
stdio.h	標準入出力のためのヘッダ・ファイル
stdlib.h	標準ユーティリティのためのヘッダ・ファイル
string.h	メモリ操作，文字列操作のためのヘッダ・ファイル

6.3 リエントラント性

“リエントラント”とは“再入可能”という意味です。リエントラント性を持つ関数は，その関数実行中に，他のプロセスでもその関数実行しようとした場合でも，正しく実行できます。たとえば，リアルタイム OS を用いたアプリケーションで，あるタスクがこの関数を実行している最中に，割り込みなどがトリガになって他のタスクへディスパッチし，そこでもこの関数を実行しても正しく実行されます。関数が RAM をテンポラリとして使う必要のある関数は，リエントラント性を持たないことがあります。

各関数のリエントラント性については，「表 6―2 可変個引数関数」から「表 6―12 数学関数」を参照してください。

6.4　ライブラリ関数

この節では，ライブラリ関数について説明します。

6.4.1　可変個引数関数

可変個引数関数は，標準ライブラリ libc.a に収録されています。

なお，可変個引き数関数として，以下のものがあります。

表 6―15　可変個引数関数

関数／マクロ名	概要
va_start	引数リスト走査用変数の初期化
va_end	引数リスト走査の終了
va_arg	引数リスト走査用変数の移動

va_start

引数リスト走査用変数の初期化を行います。

[所属]

標準ライブラリ libc.a

[指定形式]

#include <stdarg.h>

void va_start(va_list *ap*, *last-named-argument*);

[詳細説明]

変数 *ap* を，可変個引数リストの先頭（*last-named-argument* の次の引数）を指すように初期化します。

なお，可変個の引数を持つ関数 *func* を，移植性を持つ形で定義するには，次に示した形式を用います。

```
#include    <stdarg.h>
void func(arg-declarations, ...){
    va_list ap;
    type    argN;
    va_start(ap, last-named-argument);
    argN = va_arg(ap, type);
    va_end(ap);
}
```

備考 *arg-declarations* は，引数リストで，最後に *last-named-argument* が宣言されているものとします。後ろに続く“, ...”は可変個引数リストを示します。va_list は，引数リストの走査に用いられる変数（この例の場合 *ap*）の型です。

[使用例]

```
#include    <stdarg.h>
void abc(int first, int second, ...){
    va_list ap;
    int     i;
    char    c, *fmt;
    va_start(ap, second);
    i = va_arg(ap, int);
    c = va_arg(ap, int);    /*char 型は int 型に変換 */
    fmt = va_arg(ap, char *);
    va_end(ap);
}
```

va_end

引数リスト走査の終了を示します。

［所属］

標準ライブラリ libc.a

［指定形式］

```
#include <stdarg.h>
void    va_end(va_list ap);
```

［詳細説明］

リストの走査の終了を示します。va_arg を va_start と本関数とで囲むことにより，リストの走査を繰り返すことができます。

va_arg

引数リスト走査用変数の移動を行います。

［所属］

標準ライブラリ libc.a

［指定形式］

```
#include <stdarg.h>
type    va_arg(va_list ap, type);
```

［詳細説明］

変数 *ap* の指している引数を返し，次の引数を指すように変数 *ap* を進めます。*type* には，引数が関数に渡される際に変換される型を指定します。コンパイラでは，singed char 型，および short 型の引数に対しては int 型を指定し，unsinged char 型，および unsigned short 型の引数に対しては unsigned int 型を指定してください。引数ごとに異なる型を指定することができますが，“どの型の引数が渡されてきているか”は，呼び出された側と呼び出し側の関数との間の取り決めによって規定されるようにしてください。

また，“実際に引数がいくつ渡されてきているか”に関しても，呼び出された側と呼び出し側の関数との間の取り決めによって規定されるようにしてください。

6. 4. 2 文字列関数

文字列関数は，標準ライブラリ libc.a に収録されています。

なお，文字列関数として，以下のものがあります。

表 6―16 文字列関数

関数 / マクロ名	概要
index	文字列検索（最初の位置）
strpbrk	文字列検索（最初の位置）
rindex	文字列検索（最後の位置）
strrchr	文字列検索（最後の位置）
strchr	文字列検索（指定文字の最初の位置）
strstr	文字列検索（指定文字列の最初の位置）
strspn	文字列検索（指定文字を含む最大の長さ）
strcspn	文字列検索（指定文字を含まない最大の長さ）
strcmp	文字列比較
strncmp	文字列比較（文字数指定）
strcpy	文字列コピー
strncpy	文字列コピー（文字数指定）
strcat	文字列連結
strncat	文字列連結（文字数指定）
strtok	トークン分割
strlen	文字列の長さ
strerror	エラー番号の文字列変換

index

文字列検索（最初の位置）を行います。

［所属］

標準ライブラリ libc.a

［指定形式］

```
#include <string.h>
char    *index(const char *s, int c);
```

［戻り値］

見つかった文字を指すポインタを返します。*c* がこの文字列中に現れなかった場合は，null ポインタを返します。

［詳細説明］

char 型に変換された *c* と同じ文字が，*s* の指す文字列中に最初に現れる位置を求めます。終端を示す null 文字（\0）は，この文字列の一部であるとみなされます。

strpbrk

文字列検索（最初の位置）を行います。

［所属］

標準ライブラリ libc.a

［指定形式］

```
#include <string.h>
char    *strpbrk(const char *s1, const char *s2);
```

［戻り値］

この文字を指すポインタを返します。*s2* からの文字がいずれも *s1* の中に現れなかつた場合は，null ポインタを返します。

［詳細説明］

s2 の指す文字列中のいずれかの文字（null 文字（\0）は除く）が *s1* の指す文字列中に最初に現れた位置を求めます。

rindex

文字列検索（最後の位置）を行います。

［所属］

標準ライブラリ libc.a

［指定形式］

```
#include <string.h>
char    *rindex(const char *s, int c);
```

［戻り値］

見つかった *c* を指すポインタを返します。*c* がこの文字列中に現れなかった場合，null ポインタを返します。

［詳細説明］

char 型に変換された *c* が *s* の指す文字列中に最後に現れた位置を求めます。終端を示す null 文字（\0）は，この文字列の一部であるとみなされます。

strrchr

文字列検索（最後の位置）を行います。

［所属］

標準ライブラリ libc.a

［指定形式］

```
#include <string.h>
char    *strrchr(const char *s, int c);
```

［戻り値］

見つかった *c* を指すポインタを返します。*c* がこの文字列中に現れなかった場合，null ポインタを返します。

［詳細説明］

char 型に変換された *c* が *s* の指す文字列中に最後に現れた位置を求めます。終端を示す null 文字（\0）は，この文字列の一部であるとみなされます。

strchr

文字列検索（指定文字の最初の位置）を行います。

［所属］

標準ライブラリ libc.a

［指定形式］

```
#include <string.h>
char    *strchr(const char *s, int c);
```

［戻り値］

見つかった文字を指すポインタを返します。*c* がこの文字列中に現れなかった場合は，null ポインタを返します。

［詳細説明］

char 型に変換された *c* と同じ文字が，*s* の指す文字列中に最初に現れる位置を求めます。終端を示す null 文字（\0）は，この文字列の一部であるとみなされます。

strstr

文字列検索（指定文字列の最初の位置）を行います。

［所属］

標準ライブラリ libc.a

［指定形式］

```
#include <string.h>
char    *strstr(const char *s1, const char *s2);
```

［戻り値］

見つかった文字列を指すポインタを返します。文字列 *s2* が見つからなかった場合，null ポインタを返します。*s2* が長さゼロの文字列を指している場合，*s1* を返します。

［詳細説明］

s1 の指す文字列の中で，*s2* の指す文字列と最初に一致する部分（null 文字（\0）は除く）の位置を求めます。

strspn

文字列検索（指定文字を含む最大の長さ）を行います。

［所属］

標準ライブラリ libc.a

［指定形式］

```
#include <string.h>
size_t  strspn(const char *s1, const char *s2);
```

［戻り値］

先頭部分の長さを返します。

［詳細説明］

s1 の指す文字列の中で，*s2* の指す文字列中にある文字（null 文字（\0）は除く）のみで構成されている先頭部分の長さを求めます。

strcspn

文字列検索（指定文字を含まない最大の長さ）を行います。

［所属］

標準ライブラリ libc.a

［指定形式］

```
#include <string.h>
size_t  strcspn(const char *s1, const char *s2);
```

［戻り値］

先頭部分の長さを返します。

［詳細説明］

s1 の指す文字列の中で，*s2* の指す文字列（終わりの null 文字（\0）は除く）の中にない文字のみで構成されている，先頭部分の長さを求めます。

strcmp

文字列比較を行います。

［所属］

標準ライブラリ libc.a

［指定形式］

#include <string.h>

int strcmp(const char *s1, const char *s2);

［戻り値］

s1 の指す文字列が *s2* の指す文字列と比べて大きい，等しい，または小さいかによって，0 より大きい，0 に等しい，0 より小さい整数を返します。

［詳細説明］

s1 の指す文字列と *s2* の指す文字列とを比較します。

strncmp

文字列比較（文字数指定）を行います。

［所属］

標準ライブラリ libc.a

［指定形式］

#include <string.h>

int strncmp(const char *s1, const char *s2, size_t *length*);

［戻り値］

s1 の指す配列が *s2* の指す配列より大きい，等しい，または小さいかによって，0 より大きい，0 に等しい，0 より小さい整数を返します。

［詳細説明］

s1 の指す配列の文字と *s2* の指す配列の文字を最大で *length* 文字比較します。

strcpy

文字列コピーを行います。

［所属］

標準ライブラリ libc.a

［指定形式］

#include <string.h>

char *strcpy(char **dst*, const char **src*);

［戻り値］

dst の値を返します。

［詳細説明］

src の指す文字列を *dst* の指す配列にコピーします。

［使用例］

```
#include    <string.h>
void func(char *str, const char *src){
    strcpy(str, src);    /*src の指す文字列を str の指す配列にコピー */
       :
}
```

strncpy

文字列コピー（文字数指定）を行います。

［所属］

標準ライブラリ libc.a

［指定形式］

```
#include <string.h>
char    *strncpy(char *dst, const char *src, size_t length);
```

［戻り値］

dst の値を返します。

［詳細説明］

src の指す配列から *dst* の指す配列に最大で *length* 文字（null 文字（\0）を含む）コピーします。*src* の指す配列が *length* 文字より短い文字列の場合，全部で *length* 文字分書き込まれるまで，*dst* の指す配列内のコピーに null 文字（\0）が付加されます。

strcat

文字列連結を行います。

［所属］

標準ライブラリ libc.a

［指定形式］

```
#include <string.h>
char    *strcat(char *dst, const char *src);
```

［戻り値］

dst の値を返します。

［詳細説明］

src の指す文字列のコピーを，null 文字（\0）を含めて，*dst* の指す文字列の末尾に連結します。*src* の最初の文字は *dst* の終わりの null 文字（\0）を上書きします。

strncat

文字列連結（文字数指定）を行います。

［所属］

標準ライブラリ libc.a

［指定形式］

#include <string.h>

char　*strncat(char *dst, const char **src*, size_t *length*);

［戻り値］

dst の値を返します。

［詳細説明］

src の指す文字列の先頭から，最大で *length* 文字（*src* の null 文字（\0）を含む）を *dst* の指す文字列の末尾に連結します。*src* の最初の文字は *dst* の終わりの null 文字（\0）を上書きします。この結果には，終端を示す null 文字（\0）が常に付加されます。

［注意事項］

null 文字（\0）は常に付加されるので，コピーが *length* によって制限される場合，*dst* に付加される文字の個数は *length* + 1 になることに注意してください。

strtok

トークン分割を行います。

［所属］

標準ライブラリ libc.a

［指定形式］

```
#include <string.h>
char    *strtok(char *s, const char *delimiters);
```

［戻り値］

トークンへのポインタを返します。トークンが存在しない場合は，null ポインタを返します。

［詳細説明］

s の指す文字列を，*delimiters* の指す文字列中の文字で区切ることによって，トークンの列に分割します。これは最初に呼び出されると，最初の引数として *s* を持ち，その後は null ポインタを最初の引数とする呼び出しが続きます。*delimiters* の指す区切り文字列は，呼び出しごとに異なっていてもかまいません。最初の呼び出しでは，*delimiters* の指す区切り文字列中に含まれない最初の文字を求めて *s* の指す文字列中をサーチします。そのような文字が見つからなかった場合，null ポインタを返します。そのような文字が見つかった場合，その文字が最初のトークンの始まりとなります。その後，そのときの区切り文字列に含まれる文字を求めてそこからサーチを行います。

そのような文字が見つからなかった場合，そのときのトークンは *s* の指す文字列の終わりまで拡張され，あとに続くサーチは null ポインタを返します。そのような文字が見つかった場合，その文字はトークンの終端を示す null 文字（\0）で上書きされます。本関数は，あとに続く文字を指すポインタをセーブします。null ポインタを最初の引数の値としている場合，リエントラントでないコードになります。これは，最後の区切り文字のアドレスをアプリケーション・プログラムの中で保存しておき，これを使って，*s* を空でない引数として渡すようにすることで回避できます。

strlen

文字列の長さを求めます。

［所属］

標準ライブラリ libc.a

［指定形式］

```
#include <string.h>
size_t  strlen(const char *s);
```

［戻り値］

終端を示す null 文字（\0）の前に存在する文字の数を返します。

［詳細説明］

s の指す文字列の長さを求めます。

strerror

エラー番号の文字列変換を行います。

［所属］

標準ライブラリ libc.a

［指定形式］

```
#include <string.h>
char    *strerror(int errnum);
```

［戻り値］

変換した文字列へのポインタを返します。

［詳細説明］

処理系定義の対応関係に従って，エラー番号 *errnum* を文字列に変換します。*errnum* の値は，通常，グローバル変数 errno がコピーされたものです。指されている配列は，アプリケーション・プログラム側で変更しないでください。

6. 4. 3 メモリ管理関数

メモリ管理関数は，標準ライブラリ libc.a に収録されています。

なお，メモリ管理関数として，以下のものがあります。

表 6―17 メモリ管理関数

関数 / マクロ名	概要
memchr	メモリ検索
memcmp	メモリ比較
bcmp	メモリ比較（memcmp の char 引数版）
memcpy	メモリ・コピー
bcopy	メモリ・コピー（memcpy の char 引数版）
memmove	メモリ移動
memset	メモリ・セット

memchr

メモリ検索を行います。

［所属］

標準ライブラリ libc.a

［指定形式］

#include <string.h>

void *memchr(const void *s, int c, size_t length);

［戻り値］

c が見つかった場合はこの文字を指すポインタを返し，*c* が見つからなかった場合は null ポインタを返します。

［詳細説明］

s の指す領域の最初の *length* 個の文字の中に（char 型に変換された）文字 *c* が最初に現れた位置を求めます。

memcmp

メモリ比較を行います。

[所属]

標準ライブラリ libc.a

[指定形式]

#include <string.h>

int memcmp(const void *s1, const void *s2, size_t n);

[戻り値]

s1 の指すオブジェクトが *s2* の指すオブジェクトより大きい，等しい，または小さいかによって，*0* より大きい，*0* に等しい，または *0* より小さい整数を返します。

[詳細説明]

s1 の指すオブジェクトの最初の *n* 文字を *s2* の指すオブジェクトと比較します。

[使用例]

```
#include    <string.h>
int func(const void *s1, const void *s2){
    int i;
    i = memcmp(s1, s2, 5);  /*s1 の指す文字列の最初の 5 文字を s2 の指す文字列の最初の 5 文字と比較 */
    return(i);
}
```

bcmp

メモリ比較（memcmp の char 引数版）を行います。

［所属］

標準ライブラリ libc.a

［指定形式］

#include <string.h>

int bcmp(const char *s1, const char *s2, size_t n);

［戻り値］

s1 の指すオブジェクトが *s2* の指すオブジェクトより大きい，等しい，または小さいかによって，*0* より大きい，*0* に等しい，または *0* より小さい整数を返します。

［詳細説明］

s1 の指すオブジェクトの最初の *n* 文字を *s2* の指すオブジェクトと比較します。

memcpy

メモリ・コピーを行います。

［所属］

標準ライブラリ libc.a

［指定形式］

#include <string.h>

void *memcpy(void **out*, const void **in*, size_t *n*);

［戻り値］

out の値を返します。コピー元とコピー先の領域が重なっている場合，その動作は不定です。

［詳細説明］

n バイト分を *in* の指すオブジェクトから *out* の指すオブジェクトへコピーします。

bcopy

メモリ・コピー（memcpy の char 引数版）を行います。

［所属］

標準ライブラリ libc.a

［指定形式］

```
#include <string.h>
char*    bcopy(const char *in, char *out, size_t n);
```

［戻り値］

out の値を返します。コピー元とコピー先の領域が重なっている場合，その動作は不定です。

［詳細説明］

n バイト分を *in* の指すオブジェクトから *out* の指すオブジェクトへコピーします。

memmove

メモリ移動を行います。

[所属]

標準ライブラリ libc.a

[指定形式]

```
#include <string.h>
void    *memmove(void *dst, void *src, size_t length);
```

[戻り値]

コピー先の *dst* の値を返します。

[詳細説明]

length 個の文字を，*src* の指すメモリ領域から *dst* の指すメモリ領域へ移動します。コピー元とコピー先の 2 つの領域が重なり合っていても，文字を *dst* の指すメモリ領域に正しくコピーします。

memset

メモリ・セットを行います。

［所属］

標準ライブラリ libc.a

［指定形式］

```
#include <string.h>
void    *memset(const void *s, int c, size_t length);
```

［戻り値］

s の値を返します。

［詳細説明］

s の指すオブジェクトの最初の *length* 文字に unsigned char 型に変換した *c* の値をコピーします。

6. 4. 4 文字変換関数

文字変換関数は，標準ライブラリ libc.a に収録されています。

なお，文字変換関数として，以下のものがあります。

表 6―18 文字変換関数

関数 / マクロ名	概要
toupper	英小文字から英大文字変換（引数が英大文字以外のときそのまま）
_toupper	英小文字から英大文字変換（引数が英小文字のときだけ正しく変換）
tolower	英大文字から英小文字変換（引数が英大文字以外のときそのまま）
_tolower	英大文字から英小文字変換（引数が英大文字のときだけ正しく変換）
toascii	整数から ASCII 文字変換

toupper

英小文字から英大文字変換（引数が英大文字以外のときそのまま）を行います。

［所属］

標準ライブラリ libc.a

［指定形式］

#include <ctype.h>

int toupper(int *c*);

［戻り値］

c に対して islower が真となる場合，それに対応して isupper が真となる文字を返します。そうでない場合，*c* を返します。

［詳細説明］

小文字の英字を対応する大文字の英字に変換し，他の文字はすべてそのままにするマクロです。

c が EOF ～ 255 の範囲の整数の場合にのみ定義されています。“#undef toupper”を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンが使用できます。

［使用例］

```
#include    <ctype.h>
int chc = 'a';
int ret = func(chc);
int func(int c){
    int i;
    i = toupper(c); /*c の英小文字 'a' を英大文字 'A' に変換 */
    return(i);
}
```

_toupper

英小文字から英大文字変換（引数が英小文字のときだけ正しく変換）を行います。

[所属]

標準ライブラリ libc.a

[指定形式]

#include <ctype.h>

int _toupper(int *c*);

[戻り値]

c に対して islower が真となる場合，それに対応して isupper が真となる文字を返します。そうでない場合，*c* を返します。

なお，*c* に不正な値が指定された場合，その動作は不定です。

[詳細説明]

引数が小文字の英字の場合に toupper と同じ動作をするマクロです。

引数のチェックは行わないため，引数が小文字の英字である場合にのみ正しい変換を行い，それ以外のものである場合，その動作は不定となります。“#undef _toupper”を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンを使用できます。

tolower

英大文字から英小文字変換（引数が英大文字以外のときそのまま）を行います。

［所属］

標準ライブラリ libc.a

［指定形式］

#include <ctype.h>

int tolower(int *c*);

［戻り値］

c に対して isupper が真となる場合，それに対応して islower が真となる文字を返します。そうでない場合，*c* を返します。

［詳細説明］

大文字の英字を対応する小文字の英字に変換し，他の文字はすべてそのままにするマクロです。

c が EOF ～ 255 の範囲の整数の場合にのみ定義されています。“#undef tolower”を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンが使用できます。

_tolower

英大文字から英小文字変換（引数が英大文字のときだけ正しく変換）を行います。

[所属]

標準ライブラリ libc.a

[指定形式]

#include <ctype.h>

int _tolower(int *c*);

[戻り値]

c に対して isupper が真となる場合，それに対応して islower が真となる文字を返します。そうでない場合，*c* を返します。

なお，*c* に不正な値が指定された場合，その動作は不定です。

[詳細説明]

引数が大文字の英字の場合に tolower と同じ動作をするマクロです。

引数のチェックは行わないため，引数が大文字の英字である場合にのみ正しい変換を行い，それ以外のものである場合，その動作は不定となります。“#undef _tolower”を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンを使用できます。

toascii

整数から ASCII 文字変換を行います。

［所属］

標準ライブラリ libc.a

［指定形式］

#include <ctype.h>

int toascii(int *c*);

［戻り値］

0 から 127 までの範囲の整数を返します。

［詳細説明］

引数の 8 ビット目以上のビットを 0 にすることで，整数を ASCII 文字（0 ～ 127）に強制変換するマクロです。

“#undef toascii”を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンを使用できます。

6. 4. 5 文字分類関数

文字分類関数は，標準ライブラリ libc.a に収録されています。

なお，文字分類関数として，以下のものがあります。

表 6―19 文字分類関数

関数 / マクロ名	概要
isalnum	ASCII 英字，または数字であるかを判定
isalpha	ASCII 英字であるかを判定
isascii	ASCII コードであるかを判定
isupper	英大文字であるかを判定
islower	英小文字であるかを判定
isdigit	10 進数であるかを判定
isxdigit	16 進数であるかを判定
iscntrl	制御文字であるかを判定
ispunct	区切り文字であるかを判定
isspace	スペース／タブ／復帰／改行／垂直タブ／改ページであるかを判定
isprint	表示文字であるかを判定
isgraph	スペース以外の表示文字であるかを判定

isalnum

ASCII 英字，または数字であるかを判定します。

［所属］

標準ライブラリ libc.a

［指定形式］

#include <ctype.h>

int isalnum(int *c*);

［戻り値］

引数 *c* の値がそれぞれの記述に合致した場合（真）に 0 以外を返します。結果が偽であった場合は 0 を返します。

［詳細説明］

ASCII 英字，または数字であるかどうか調べるマクロです。すべての整数値に対して定義されています。“#undef isalnum”を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンが使用できます。

isalpha

ASCII 英字であるかを判定します。

［所属］

標準ライブラリ libc.a

［指定形式］

#include <ctype.h>

int isalpha(int *c*);

［戻り値］

c の値がそれぞれの記述に合致した場合（真）に 0 以外を返します。結果が偽であった場合は 0 を返します。

［詳細説明］

ASCII 英字であるかどうか調べるマクロです。*c* が isascii で真になるか，または *c* が EOF の場合にのみ，定義されています。“#undef isalpha”を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンが使用できます。

isascii

ASCII コードであるかを判定します。

［所属］

標準ライブラリ libc.a

［指定形式］

#include <ctype.h>

int isascii(int *c*);

［戻り値］

c の値がそれぞれの記述に合致した場合（真）に 0 以外を返します。結果が偽であった場合は 0 を返します。

［詳細説明］

ASCII コード（0x00 ～ 0x7F）であるかどうかを調べるマクロです。すべての整数に対して定義されています。“#undef isascii”を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンが使用できます。

isupper

英大文字であるかを判定します。

［所属］

標準ライブラリ libc.a

［指定形式］

#include <ctype.h>

int isupper(int *c*);

［戻り値］

c の値がそれぞれの記述に合致した場合（真）に 0 以外を返します。結果が偽であった場合は 0 を返します。

［詳細説明］

大文字の英字（A ～ Z）であるかどうかを調べるマクロです。*c* が isascii で真になるか，または *c* が EOF の場合にのみ，定義されています。“#undef isupper”を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンが使用できます。

islower

英小文字であるかを判定します。

［所属］

標準ライブラリ libc.a

［指定形式］

#include <ctype.h>

int islower(int *c*);

［戻り値］

c の値がそれぞれの記述に合致した場合（真）に 0 以外を返します。結果が偽であった場合は 0 を返します。

［詳細説明］

小文字の英字（a ～ z）であるかどうかを調べるマクロです。*c* が isascii で真になるか，または *c* が EOF の場合にのみ，定義されています。“#undef islower”を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンが使用できます。

isdigit

10 進数であるかを判定します。

［所属］

標準ライブラリ libc.a

［指定形式］

#include <ctype.h>

int isdigit(int *c*);

［戻り値］

c の値がそれぞれの記述に合致した場合（真）に 0 以外を返します。結果が偽であった場合は 0 を返します。

［詳細説明］

10 進数の数字であるかどうか調べるマクロです。*c* が isascii で真になるか，または *c* が EOF の場合にのみ，定義されています。“#undef isdigit”を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンが使用できます。

isxdigit

16 進数であるかを判定します。

［所属］

標準ライブラリ libc.a

［指定形式］

#include <ctype.h>

int isxdigit(int *c*);

［戻り値］

c の値がそれぞれの記述に合致した場合（真）に 0 以外を返します。結果が偽であった場合は 0 を返します。

［詳細説明］

16 進数の数字（0 ～ 9，a ～ f，または A ～ F）であるかどうかを調べるマクロです。*c* が isascii で真になるか，または *c* が EOF の場合にのみ，定義されています。“#undef isxdigit”を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンが使用できます。

iscntrl

制御文字であるかを判定します。

［所属］

標準ライブラリ libc.a

［指定形式］

#include <ctype.h>

int iscntrl(int *c*);

［戻り値］

c の値がそれぞれの記述に合致した場合（真）に 0 以外を返します。結果が偽であった場合は 0 を返します。

［詳細説明］

制御文字（0x00 ～ 0x1F，または 0x7F）であるかどうかを調べるマクロです。*c* が isascii で真になるか，または *c* が EOF の場合にのみ，定義されています。“#undef iscntrl”を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンが使用できます。

ispunct

区切り文字であるかを判定します。

［所属］

標準ライブラリ libc.a

［指定形式］

#include <ctype.h>

int ispunct(int *c*);

［戻り値］

c の値がそれぞれの記述に合致した場合（真）に 0 以外を返します。結果が偽であった場合は 0 を返します。

［詳細説明］

印字可能な区切り文字“isgraph (c) && ! isalnum (c)”であるかどうかを調べるマクロです。*c* が isascii で真になるか，または *c* が EOF の場合にのみ，定義されています。“#undef ispunct”を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンが使用できます。

isspace

スペース／タブ／復帰／改行／垂直タブ／改ページであるかを判定します。

［所属］

標準ライブラリ libc.a

［指定形式］

#include <ctype.h>

int isspace(int *c*);

［戻り値］

c の値がそれぞれの記述に合致した場合（真）に 0 以外を返します。結果が偽であった場合は 0 を返します。

［詳細説明］

スペース，タブ，復帰，改行，垂直タブ，改ページ（0x09 ～ 0x0D，または 0x20）であるかどうかを調べるマクロです。*c* が isascii で真になるか，または *c* が EOF の場合にのみ，定義されています。“#undef isspace”を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンが使用できます。

isprint

表示文字であるかを判定します。

［所属］

標準ライブラリ libc.a

［指定形式］

#include <ctype.h>

int isprint(int *c*);

［戻り値］

c の値がそれぞれの記述に合致した場合（真）に 0 以外を返します。結果が偽であった場合は 0 を返します。

［詳細説明］

表示文字（0x20 ～ 0x7E）であるかどうかを調べるマクロです。*c* が isascii で真になるか，または *c* が EOF の場合にのみ，定義されています。“#undef isprint”を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンが使用できます。

isgraph

スペース以外の表示文字であるかを判定します。

［所属］

標準ライブラリ libc.a

［指定形式］

#include <ctype.h>

int isgraph(int *c*);

［戻り値］

c の値がそれぞれの記述に合致した場合（真）に 0 以外を返します。結果が偽であった場合は 0 を返します。

［詳細説明］

スペース（0x20）以外の表示文字[注]（0x20 ～ 0x7E）であるかどうかを調べるマクロです。*c* が isascii で真になるか，または *c* が EOF の場合にのみ，定義されています。“#undef isgraph”を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンが使用できます。

注 printing character のことです。

6. 4. 6 標準入出力関数

標準入出力関数は，標準ライブラリ libc.a に収録されています。

なお，標準入出力関数として，以下のものがあります。

表 6―20 標準入出力関数

関数 / マクロ名	概要
fread	ストリームからの読み込み
getc	ストリームからの文字読み込み（fgetc と同じ）
fgetc	ストリームからの文字読み込み（getc と同じ）
fgets	ストリームからの一行読み込み
fwrite	ストリームへの書き込み
putc	ストリームへの文字書き込み
fputc	ストリームへの文字書き込み
fputs	ストリームへの文字列出力
getchar	標準入力からの一文字読み込み
gets	標準入力からの文字列読み込み
putchar	標準出力ストリームへの文字書き込み
puts	標準出力ストリームへの文字列出力
sprintf	書式付き出力
fprintf	フォーマット指定したテキストをストリームへ出力
vsprintf	フォーマット指定したテキストを文字列へ書き込み
printf	フォーマット指定したテキストを標準出力ストリームへ出力
vfprintf	フォーマット指定したテキストをストリームへ書き込み
vprintf	フォーマット指定したテキストを標準出力ストリームへ書き込み
sscanf	書式付き入力
fscanf	ストリームからのデータ読み込みと解釈
scanf	標準入力ストリームからのテキストの読み込みと解釈
ungetc	入力ストリームへの文字押し戻し
rewind	ファイル位置指示子のリセット
perror	エラー処理

fread

ストリームからの読み込みを行います。

備考 ルネサス エレクトロニクス製の統合デバッガ，システム・シミュレータでは，サポートされていません。

[所属]

標準ライブラリ libc.a

[指定形式]

#include <stdio.h>

size_t fread(void *_ptr_, size_t _size_, size_t _nmemb_, FILE *_stream_);

[戻り値]

入力した要素数 _nmemb_ を返します。

エラー・リターンはありません。

[詳細説明]

stream が指す入力ストリームから，_size_ の大きさの要素を _nmemb_ 個入力し，_ptr_ へ格納します。_stream_ に指定できるのは，標準入出力の stdin だけです。

[使用例]

```
#include    <stdio.h>
void func(void){
    struct{
        int     c;
        double  d;
    }buf[10];
    fread(buf, sizeof(buf[0]), sizeof(buf) / sizeof(buf [0]), stdin);
}
```

getc

ストリームからの文字読み込みを行います。（fgetc と同じ）

備考 ルネサス エレクトロニクス製の統合デバッガ，システム・シミュレータでは，サポートされていません。

［所属］

標準ライブラリ libc.a

［指定形式］

#include <stdio.h>

int getc(FILE *_stream_);

［戻り値］

入力文字を返します。

エラー・リターンはありません。

［詳細説明］

stream が指す入力ストリームから，1 文字を入力します。*stream* に指定できるのは，標準入出力の stdin だけです。

fgetc

ストリームからの文字読み込みを行います。（getc と同じ）

備考 ルネサス エレクトロニクス製の統合デバッガ，システム・シミュレータでは，サポートされていません。

［所属］

標準ライブラリ libc.a

［指定形式］

#include <stdio.h>

int fgetc(FILE *_stream_);

［戻り値］

入力文字を返します。

エラー・リターンはありません。

［詳細説明］

stream が指す入力ストリームから，1 文字を入力します。_stream_ に指定できるのは，標準入出力の stdin だけです。

［使用例］

```
#include    <stdio.h>

int func(void){
    int c;
    c = fgetc(stdin);
    return(c);
}
```

fgets

ストリームからの一行読み込みを行います。

備考 ルネサス エレクトロニクス製の統合デバッガ，システム・シミュレータでは，サポートされていません。

[所属]

標準ライブラリ libc.a

[指定形式]

```
#include <stdio.h>
char    *fgets(char *s, int n, FILE *stream);
```

[戻り値]

s を返します。
エラー・リターンはありません。

[詳細説明]

stream が指す入力ストリームから，最大 *n* - 1 文字を入力し，*s* へ格納します。文字の入力は，改行文字の検出によっても終了します。この場合，改行文字も *s* へ格納されます。最後に文字列の終結 null 文字が *s* へ格納されます。*stream* に指定できるのは，標準入出力の stdin だけです。

fwrite

ストリームへの書き込みを行います。

備考 ルネサス エレクトロニクス製の統合デバッガ，システム・シミュレータでは，サポートされていません。

[所属]

標準ライブラリ libc.a

[指定形式]

#include <stdio.h>

size_t fwrite(const void *_ptr_, size_t _size_, size_t _nmemb_, FILE *_stream_);

[戻り値]

出力した要素数 _nmemb_ を返します。

エラー・リターンはありません。

[詳細説明]

stream が指す出力ストリームへ，_ptr_ が指す配列から，_size_ の大きさの要素を _nmemb_ 個出力します。_stream_ に指定できるのは，標準入出力の stdout と stderr だけです。

putc

ストリームへの文字書き込みを行います。

備考 ルネサス エレクトロニクス製の統合デバッガ，システム・シミュレータでは，サポートされていません。

[所属]

標準ライブラリ libc.a

[指定形式]

#include <stdio.h>

int putc(int *c*, FILE **stream*);

[戻り値]

文字 *c* を返します。

エラー・リターンはありません。

[詳細説明]

stream が指す出力ストリームへ，文字 *c* を出力します。*stream* に指定できるのは，標準入出力の stdout と stderr だけです。

fputc

ストリームへの文字書き込みを行います。

備考 ルネサス エレクトロニクス製の統合デバッガ，システム・シミュレータでは，サポートされていません。

[所属]

標準ライブラリ libc.a

[指定形式]

#include <stdio.h>

int fputc(int *c*, FILE **stream*);

[戻り値]

文字 *c* を返します。

エラー・リターンはありません。

[詳細説明]

stream が指す出力ストリームへ，文字 *c* を出力します。*stream* に指定できるのは，標準入出力の stdout と stderr だけです。

[使用例]

```
#include    <stdio.h>
void func(void){
    fputc('a', stdout);
}
```

fputs

ストリームへの文字列出力を行います。

備考 ルネサス エレクトロニクス製の統合デバッガ，システム・シミュレータでは，サポートされていません。

［所属］

標準ライブラリ libc.a

［指定形式］

```
#include <stdio.h>
int fputs(const char *s, FILE *stream);
```

［戻り値］

0 を返します。
エラー・リターンはありません。

［詳細説明］

stream が指す出力ストリームへ，文字列 *s* を出力します。文字列の終端 null 文字は出力しません。*stream* に指定できるのは，標準入出力の stdout と stderr だけです。

getchar

標準入力からの一文字読み込みを行います。

備考 ルネサス エレクトロニクス製の統合デバッガ，システム・シミュレータでは，サポートされていません。

[所属]

標準ライブラリ libc.a

[指定形式]

```
#include <stdio.h>
int getchar(void);
```

[戻り値]

入力文字を返します。
エラー・リターンはありません。

[詳細説明]

標準入出力の stdin から，1 文字を入力します。

gets

標準入力からの文字列読み込みを行います。

備考 ルネサス エレクトロニクス製の統合デバッガ，システム・シミュレータでは，サポートされていません。

[所属]

標準ライブラリ libc.a

[指定形式]

```
#include <stdio.h>
char    *gets(char *s);
```

[戻り値]

s を返します。
エラー・リターンはありません。

[詳細説明]

標準入出力の stdin から，改行文字を検出するまで文字を入力し，*s* へ格納します。入力した改行文字は捨て，最後に，文字列の終結 null 文字が *s* へ格納されます。

putchar

標準出力ストリームへの文字書き込みを行います。

備考 ルネサス エレクトロニクス製の統合デバッガ，システム・シミュレータでは，サポートされていません。

[所属]

標準ライブラリ libc.a

[指定形式]

```
#include <stdio.h>
int putchar(int c);
```

[戻り値]

文字 *c* を返します。
エラー・リターンはありません。

[詳細説明]

標準入出力の stdout へ，文字 *c* を出力します。

puts

標準出力ストリームへの文字列出力を行います。

備考 ルネサス エレクトロニクス製の統合デバッガ，システム・シミュレータでは，サポートされていません。

[所属]

標準ライブラリ libc.a

[指定形式]

```
#include <stdio.h>
int puts(const char *s);
```

[戻り値]

0 を返します。
エラー・リターンはありません。

[詳細説明]

標準入出力の stdout へ，文字列 s を出力します。文字列の終端 null 文字は出力せず，代わりに改行文字を出力します。

sprintf

書式付き出力を行います。

[所属]

標準ライブラリ libc.a

[指定形式]

#include <stdio.h>

int sprintf(char *s, const char *format[, arg, ...]);

[戻り値]

出力された文字（null 文字（\0）は除きます）の数を返します。

エラー・リターンはありません。

[詳細説明]

それぞれの *arg* に *format* の指す文字列で指定された書式を適用し，それにより出力された書式付きデータを *s* の指す配列に書き出します。

書式に対して引数が十分にない場合，動作は不定です。書式文字列の終わりに到達するとリターンします。書式で必要としている以上に引数がある場合，余分の引数を無視します。また，*s* の領域が引数の 1 つと重なっていると動作は不定になります。

format は，“後ろに続く引数がどのような出力に変換されるか”を指定しています。書き込まれた文字の最後には null 文字（\0）が付加されます（null 文字（\0）は返り値におけるカウントの対象とはなりません）。

format は，次に示す 2 種類のディレクティブにより構成されます。

通常文字	変換されずにそのまま出力にコピーされるものです（“%”以外）。
変換指示	0 個以上の引数を取り込み，指示を与えるものです。

各変換指示は，文字“%”で始まります（出力中に“%”を入れたい場合は，書式文字列の中では“%%”とします）。“%”の後ろは，次のようになります。

%[フラグ][フィールド長][精度][サイズ][型指定文字]

それぞれの変換指示について，次に説明します。

(1) フラグ

任意の順に置かれた，変換指示の意味を修飾する 0 個以上のフラグです。フラグ文字とその意味を次に示します。

-	変換された結果をフィールド中に左詰めにし，右側は空白で満たされます（このフラグが指定されない場合，変換された結果は右詰めにされます）。
+	符号付きの変換の結果を常に + 符号，または - 符号で始めます（このフラグが指定されない場合，変換された結果は，負の値が変換された場合にのみ符号で始められます）。
スペース	符号付きの変換の最初の文字が符号でない場合，または符号付きの変換が文字を生じない場合，その結果の前にスペース（“ ”）を付けます。スペース・フラグと + フラグとが両方現れる場合，スペース・フラグは無視されます。
#	結果を“別の形式[注 1]”に変換します。o 変換に対しては，その変換結果の最初の数字が 0 になるようにその精度を増やします。x，または X 変換に対しては，0 以外の変換結果の先頭に 0x，または 0X を付加します。e，f，g，E，G 変換に対しては，その変換結果に小数点以下の数字が存在しない場合であっても，小数点“.”を付加します[注 2]。g，G 変換に対しては，変換結果から後ろに続く 0 が削除されないようにします。これら以外の変換に対しては，その動作は不定となります。
0	d，e，f，g，i，o，u，x，E，G，X 変換に対し，フィールド長を埋めるために，符号，または基底の指示に続いて 0 を付加します。 0 フラグと - フラグの両方が指定された場合，0 フラグは無視されます。d，i，o，u，x，X 変換については，精度を指定している場合，ゼロ（0）フラグを無視します。 0 はフラグとして解釈され，フィールド幅の始まりとは解釈されないことに注意してください。 これら以外の変換に対してはその動作は不定となります。

注 1. alternate format のことです。

2. 通常，小数点は，その後ろに数字が続く場合にのみ現れます。

(2) フィールド長

オプショナルな最小フィールド長です。変換された値がこのフィールド長より小さい場合，左側にスペースが詰められます（前述の左詰めフラグが与えられた場合は右側にスペースが詰められます）。このフィールド長は“*”，または 10 進整数の形を取ります。“*”で指定した場合，int 型の引数をフィールド長として使用します。負のフィールド長は，サポートしていません。負のフィールド長を指定しようとすると，正のフィールド長の前にマイナス（-）フラグが付いたものと解釈されます。

(3) 精度[注]

これに与えられる値は，d，i，o，u，x，X 変換に対しては現れる数字の個数の最小値であり，e, f, E 変換に対しては“.”の後ろに現れる数字の個数であり，g，G 変換に対しては最大有効桁数です。精度は，“*”，または 10 進整数が後ろに続く“.”の形式を取ります。“*”を指定した場合，int 型の引数を精度として使用します。負の精度を指定した場合，精度を省略したものとみなされます。“.”のみが指定された場合，精度は 0 とされます。精度がこれら以外の変換指示とともに現れた場合，動作は不定となります。

注 precision のことです。

(4) サイズ

対応する引数のデータ型を解釈するためのデフォルトの方法を変更する，任意選択のサイズ文字 h，l，および L です。

h を指定した場合，後ろに続く d，i，o，n，u，x，X の型指定を強制的に short，または unsigned short に適用します。

l を指定した場合，後ろに続く d，i，o，u，x，X の型指定を強制的に long，または unsigned long に適用します。l はさらに，後ろに続く n の型指定を強制的に long へのポインタに適用します。h，または l といっしょにこれと別の型指定文字を使用した場合，その動作は不定です。

L を指定した場合，後ろに続く e，E，f，g，G の型指定を強制的に long double に適用します。L といっしょにこれ以外の型指定文字を使用した場合，その動作は不定です。

(5) 型指定文字

適用される変換の型を指定する文字です。

変換の型を指定する文字とその意味を次に示します。

%	文字“%”を出力します。引数は変換されません。変換指示は“%%”となります。
c	int 型の引数を unsigned char 型に変換し，変換結果の文字を出力します。
d	int 型の引数を符号付きの 10 進数に変換します。
e，E	double 型の引数を，小数点の前に（引数が 0 でない場合 0 でない）1 つの文字を持ち，小数点以下の数字の個数は精度に等しい [-]d.dddde ± dd の形式に変換します。E 変換指示は，指数部が“e”ではなく“E”で始まる数字を生成します。
f	double 型の引数を [-]dddd.dddd の形式の 10 進表記に変換します。
g，G	精度には仮数部の数字の個数を指定するものとし，double 型の引数を e（G 変換指示の場合 E），または f の形式に変換します。変換結果の末尾の 0 は結果の小数点部から除かれます。小数点は，後ろに数字が続く場合にのみ現れます。
i	d の変換と同じ変換をします。
n	同じオブジェクト内で出力された文字の個数を格納します。int 型へのポインタを引数とします。
p	処理系定義書式でポインタを出力します。CA850 では，ポインタを unsigned long として扱っています（lu の指定と同じです）。
o，u，x，X	unsigned int 型の引数を dddd の形式の 8 進表記（o），符号なしの 10 進表記（u），符号なしの 16 進表記（x，または X）に変換します。x 変換に対しては文字 abcdef が用いられ X 変換に対しては文字 ABCDEF が用いられます。
s	引数は文字型の配列を指すポインタでなければなりません。この配列からの文字を，終端を示す null 文字（\0）の前まで（null 文字（\0）自身は含まずに）出力します。精度が指定された場合，それ以上の個数の文字は出力されません。精度が指定されなかった，または精度がこの配列の大きさ以上の値であった場合，この配列は null 文字（\0）を含むようにしてください。

［使用例］

```
#include    <stdio.h>
void func(int val){
    char    s[20];
    sprintf(s, "%-10.5lx\n", val);  /*val の値に対し，左詰め，フィールド長 10，精度 5，サイズ long，
                                        16 進表記を指定し，改行文字を付加して s の指す配列へ出力 */
}
```

fprintf

フォーマット指定したテキストをストリームへ出力します。

備考 ルネサス エレクトロニクス製の統合デバッガ，システム・シミュレータでは，サポートされていません。

[所属]

標準ライブラリ libc.a

[指定形式]

#include <stdio.h>

int fprintf(FILE *_stream_, const char *_format_[, _arg_, ...]);

[戻り値]

出力された文字数を返します。

[詳細説明]

それぞれの _arg_ に _format_ の指す文字列で指定された書式を適用し，それにより出力された書式付きデータを _stream_ に出力します。_stream_ に指定できるのは，標準入出力の stdout と stderr だけです。_format_ の記述方法は sprintf と同様です。sprintf と違って，最後に null 文字（\0）は出力されません。

[注意事項]

stream に stdout（標準出力）, stderr（標準エラー）を指定します。ストリームの入出力先は I/O アドレスなど 1 メモリ・アドレスを割り当てます。デバッガとの連携でこれらのストリームを使用するには，stdio.h ファイルで定義されている，ストリーム構造体の初期値設定が必要です。関数を呼び出す前に，初期値設定を行ってください。

【stdio.h におけるストリーム構造体の定義】

```
typedef struct{
    int         mode;   /*with error descriptions*/
    unsigend    handle;
    int         unget_c;
}FILE;
typedef int     fpos_t;
#pragma section sdata   begin
extern  FILE    __struct_stdin;
extern  FILE    __struct_stdout;
extern  FILE    __struct_stderr;
#pragma section sdata   end
```

```
#define stdin(&__struct_stdin)
#define stdout(&__struct_stdout)
#define stderr(&__struct_stderr)
```

構造体の第一メンバ mode は，入出力状態を示します。ACCSD_OUT/ACCSD_IN として内部定義されています。第三メンバ unget_c は，押し戻し文字（stdin のみ）を示し，-1 として内部定義されています。

-1 の場合，押し戻し文字“なし”を表します。第二メンバ handle は，入出力 I/O アドレスを示します。handle には，使用するデバッガで決められている値を設定してください。

【入出力 I/O アドレス設定例】

```
__struct_stdout.handle = 0xfffff000;
__struct_stderr.handle = 0x00fff000;
__struct_stdin.handle = 0xfffff002;
#pragma section sdata   begin
extern  FILE    __struct_stdout;
extern  FILE    __struct_stderr;
#pragma section sdata   end
#define stdin(&__struct_stdin)
#define stdout(&__struct_stdout)
#define stderr(&__struct_stderr)
```

［使用例］

```
#include    <stdio.h>
void func(int val){
    fprintf(stdout, "%-10.5x\n", val);
}
/* 汎用のエラー報告ルーチンにおける使用例 */
void error(char *function_name, char *format, ...){
    va_list arg;
    va_start(arg, format);
    fprintf(stderr, "ERROR in %s:", function_name); /* エラーが発生した関数名を出力 */
    vfprintf(stderr, format, arg);                  /* 残りのメッセージを出力 */
    va_end(arg);
}
```

vsprintf

フォーマット指定したテキストを文字列へ書き込みます。

［所属］

標準ライブラリ libc.a

［指定形式］

#include <stdio.h>

int vsprintf(char *s, const char *f*ormat*, va_list *arg*);

［戻り値］

出力された文字（null 文字（\0）は除きます）の数を返します。

エラー・リターンはありません。

［詳細説明］

arg の指す引数列に *format* の指す文字列で指定された書式を適用し，それにより出力された書式付きデータを *s* が指す配列に出力します。本関数は，可変個数実引数並びを *arg* で置き換えた sprintf と等価です。本関数の呼び出しの前に，va_start で *arg* を初期化しておく必要があります。

printf

フォーマット指定したテキストを標準出力ストリームへ出力します。

備考 ルネサス エレクトロニクス製の統合デバッガ，システム・シミュレータでは，サポートされていません。

[所属]

標準ライブラリ libc.a

[指定形式]

```
#include <stdio.h>
int printf(const char *format[, arg, ...]);
```

[戻り値]

出力された文字数を返します。

[詳細説明]

それぞれの *arg* に *format* の指す文字列で指定された書式を適用し，それにより出力された書式付きデータを標準入出力の stdout に出力します。*format* の記述方法は sprintf と同様です。sprintf と異なり，最後に null 文字（\0）は出力されません。

vfprintf

フォーマット指定したテキストをストリームへ書き込みます。

備考 ルネサス エレクトロニクス製の統合デバッガ，システム・シミュレータでは，サポートされていません。

[所属]

標準ライブラリ libc.a

[指定形式]

#include <stdio.h>

int vfprintf(FILE *_stream_, const char *_format_, va_list _arg_);

[戻り値]

出力された文字数を返します。

[詳細説明]

arg の指す引数列に _format_ の指す文字列で指定された書式を適用し，それにより出力された書式付きデータを _stream_ に出力します。_stream_ に指定できるのは，標準入出力の stdout と stderr だけです。_format_ の記述方法は sprintf と同様です。本関数は，可変個数実引数並びを _arg_ で置き換えた fprintf と等価です。本関数の呼び出しの前に，va_start で _arg_ を初期化しておく必要があります。

[注意事項]

stream に stdin（標準入力）, stdout（標準エラー）を指定します。ストリームの入出力先は I/O アドレスなど 1 メモリ・アドレスを割り当てます。デバッガとの連携でこれらのストリームを使用するには，stdio.h ファイルで定義されている，ストリーム構造体の初期値設定が必要です。関数を呼び出す前に，初期値設定を行ってください。

【stdio.h におけるストリーム構造体の定義】

```
typedef struct{
    int         mode;   /*with error descriptions*/
    unsigend    handle;
    int         unget_c;
}FILE;
typedef int     fpos_t;
#pragma section sdata   begin
extern  FILE    __struct_stdin;
extern  FILE    __struct_stdout;
```

```
extern  FILE    __struct_stderr;
#pragma section sdata   end
#define stdin(&__struct_stdin)
#define stdout(&__struct_stdout)
#define stderr(&__struct_stderr)
```

構造体の第一メンバ mode は，入出力状態を示します。ACCSD_OUT/ACCSD_IN として内部定義されています。第三メンバ unget_c は，押し戻し文字（stdin のみ）を示し，-1 として内部定義されています。

-1 の場合，押し戻し文字“なし”を表します。第二メンバ handle は，入出力 I/O アドレスを示します。handle には，使用するデバッガで決められている値を設定してください。

【入出力 I/O アドレス設定例】

```
__struct_stdout.handle = 0xfffff000;
__struct_stderr.handle = 0x00fff000;
__struct_stdin.handle = 0xfffff002;
#pragma section sdata   begin
extern  FILE    __struct_stdout;
extern  FILE    __struct_stderr;
#pragma section sdata   end
#define stdin(&__struct_stdin)
#define stdout(&__struct_stdout)
#define stderr(&__struct_stderr)
```

［使用例］

```
#include    <stdio.h>
void func(int val){
    fprintf(stdout, "%-10.5x\n", val);
}
/* 汎用のエラー報告ルーチンにおける使用例 */
void error(char *function_name, char *format, ...){
    va_list arg;
    va_start(arg, format);
    fprintf(stderr, "ERROR in %s:", function_name); /* エラーが発生した関数名を出力 */
    vfprintf(stderr, format, arg);                  /* 残りのメッセージを出力 */
    va_end(arg);
}
```

vprintf

フォーマット指定したテキストを標準出力ストリームへ書き込みます。

備考 ルネサス エレクトロニクス製の統合デバッガ，システム・シミュレータでは，サポートされていません。

[所属]

標準ライブラリ libc.a

[指定形式]

```
#include <stdio.h>
int vprintf(const char *format, va_list arg);
```

[戻り値]

出力された文字数を返します。

[詳細説明]

arg の指す引数列に *format* の指す文字列で指定された書式を適用し，それにより出力された書式付きデータを標準入出力の stdout に出力します。*format* の記述方法は sprintf と同様です。本関数は，可変個数実引数並びを *arg* で置き換えた printf と等価です。本関数の呼び出しの前に，va_start で *arg* を初期化しておく必要があります。

sscanf

書式付き入力を行います。

[所属]

標準ライブラリ libc.a

[指定形式]

#include <stdio.h>

int sscanf(const char *s, const char *format[, arg, ...]);

[戻り値]

走査，変換，格納が正常に実行できた入力フィールドの個数を返します。返却値には，格納されなかった走査済みフィールドは含まれません。ファイルの終わりで読み込もうとした場合，返却値は EOF です。フィールドが格納されなかった場合は，返却値は 0 です。

[詳細説明]

format の指す文字列で指定された書式に従い，その後ろに続く引数 *arg* を，変換された入力を格納するオブジェクトを指すポインタとして扱い，*s* の指す配列から変換する入力を読み込みます。

format には，認識されうる入力列，および“代入のためにどのように変換を行うか”ということを指定します。*format* に対し十分な引数が存在しない場合，その動作は不定となります。引数が残っているのに *format* が使い果たされた場合，残された引数は無視されます。

format は，次に示す 3 種類のディレクティブにより構成されます。

1 個以上の空白類	スペース（ ），タブ（\t），改行（\n）です。 本関数を実行して，文字列内に空白文字が見つかった場合，次の空白でない文字まで連続するすべての空白文字を読み込みます（格納はしません）。
通常の文字	“%”以外のすべての ASCII 文字です。 本関数を実行して，文字列内に通常の文字が見つかった場合，それを読み込みますが，格納はしません。変換指示により，本関数は，入力フィールドから文字列を読み込み，特定の型の値に変換し，引数で指定した位置に格納します。変換指示で明示されて一致しているのでなければ，後ろに続く空白は読み込まれません。
変換指示	0 個以上の引数を取り込み，変換の指示を与えます。

各変換指示は“%”で始まります。“%”の後ろは，次のようになります。

%[代入抑制文字][フィールド長][サイズ][型指定文字]

それぞれの変換指示について，次に説明します。

(1) 代入抑制文字

入力フィールドの解釈，および代入を抑制する“*”です。

(2) フィールド長

最大フィールド長を規定する 0 以外の 10 進整数です。入力フィールドを変換する前に読み込まれる最大文字数を指定します。入力フィールドがこのフィールド長より小さい場合，本関数はフィールド内のすべての文字を読み込み，次のフィールドとその変換指示へ進みます。また，フィールド長分を読み込む前に，空白文字，または変換不能文字が見つかった場合，その文字までの文字群を読み込み，変換し，格納します。その後，本関数は次の変換指示へ進みます。

(3) サイズ

対応する引数のデータ型を解釈するデフォルトの方法を変更する，任意選択のサイズ文字 h, l，および L です。

h を指定した場合，後ろに続く d，i，n，o，u，x の型指定を強制的に short int 型に変換し，short 型で格納します。c，e，f，n，p，s，D，I，O，U，X では，何もしません。

l を指定した場合，後ろに続く d，i，n，o，u，x の型指定を強制的に long int 型に変換し，long 型で格納します。e，f，g では，強制的に double 型に変換し，double 型で格納します。c，n，p，s，D，I，O，U，X では，何もしません。

L を指定した場合，後ろに続く d，i，o，u，x の型指定を強制的に long double 型に変換し，long double 型で格納します。他の型指定では，何もしません。

これら以外の場合，その動作は不定です。

(4) 型指定文字

適用される変換の型を指定する文字です。変換の型を指定する文字とその意味を次に示します。

%	文字“%”にマッチします。変換も代入も行われません。変換指示は“%%”となります。
c	1 文字を走査します。対応する引数は“char *arg”にしてください。
d	10 進整数を対応する引数に読み込みます。対応する引数は“int *arg”にしてください。
e，f，g	浮動小数点数を対応する引数に読み込みます。対応する引数は“float *arg”にしてください。
i	10 進，8 進，または 16 進整数を対応する引数に読み込みます。対応する引数は“int *arg”にしてください。
n	対応する引数に読み込んだ文字の個数を格納します。対応する引数は“int *arg”にしてください。
o	8 進整数を対応する引数に読み込みます。対応する引数は“int *arg”にしてください。
p	走査したポインタを格納します。これは処理系定義です。 CA850 では，%p を %U とまったく同じように処理しています。対応する引数は“void **arg”にしてください。
s	与えられた配列の中に文字列を読み込みます。対応する引数は“char arg[ ]”にしてください。
u	符号なし 10 進整数を対応する引数に読み込みます。対応する引数は“unsigned int *arg”にしてください。

x，X	16 進整数を対応する引数に読み込みます。対応する引数は“int *arg”にしてください。
D	10 進整数を対応する引数に読み込みます。対応する引数は“long *arg”にしてください。
E，F，G	浮動小数点数を対応する引数に読み込みます。対応する引数は“double *arg”にしてください。
I	10 進，8 進，または 16 進整数を対応する引数に読み込みます。対応する引数は“long *arg”にしてください。
O	8 進整数を対応する引数に読み込みます。対応する引数は“long *arg”にしてください。
U	符号なし 10 進整数を対応する引数に読み込みます。対応する引数は“unsigned long *arg”にしてください。
[ ]	空でない文字列を引数 arg で始まるメモリの中へ読み込みます。この領域には，文字列と，自動的に付加される，文字列の終わりを示す null 文字（\0）とを受け入れられる大きさが必要です。対応する引数は“char *arg”にしてください。 [ ] で囲まれた文字パターンを，型指定文字 s の代わりに使用することができます。文字パターンは，本関数の入力フィールドを構成する文字の検索セットを定義する文字集合です。[ ] 内の最初の文字が“^”の場合，検索セットは反転され，[ ] 内の文字以外のすべての ASCII 文字が含まれます。また，ショートカットとして使用できる範囲指定機能もあります。たとえば，%[0-9] は，すべての 10 進数字と一致します。この集合内では，“-”は最初，または最後の文字にはできません。“-”の前の文字は，その後ろの文字よりも辞書式順序で小さくなるようにしてください。 - %[abcd] a，b，c，d のみを含む文字列と一致します。 - %[^abcd] a，b，c，d 以外の任意の文字を含む文字列と一致します。 - %[A-DW-Z] A，B，C，D，W，X，Y，Z を含む文字列と一致します。 - %[z-a] z，-，a と一致します（範囲指定とはみなされません）。

浮動小数点数 (型指定文字 e，f，g，E，F，G) の場合，次の一般形式に対応させてください。

[+ | -] ddddd [.] ddd [E | e [+ | -] ddd]

ただし，上記の一般形式のうち [ ] で囲まれた部分は任意選択であり，ddd は 10 進数字を表します。

[注意事項]

- 通常のフィールド終了文字に到達する前に，特定フィールドの走査を停止したり完全に終了したりする可能性があります。
- 次の状況では，その時点でのフィールドの走査，格納を停止し，次の入力フィールドに移動します。
  - 代入抑制文字（*）が書式指定の中で“%”の後ろに現れており，その時点の入力フィールドは走査されているが格納はされていない。
  - フィールド長（正の 10 進整数）指定文字を読み込んだ。
  - 読み込む次の文字がその変換指示では変換できない（たとえば，指示が 10 進のときに Z を読み込む場合）。
  - 入力フィールド内の次の文字が検索セット内に現れていない（，または反転検索セット内に現れている）。

以上の理由からその時点の入力フィールドの走査を停止すると，次の文字が未読であるとみなされ，次の入力フィールドの最初の文字，またはその入力のあとの読み込み操作の最初の文字として使用されます。

- 本関数は，次の状況では終了します。
  - 入力フィールド内の次の文字が変換する文字列内の対応する通常文字と一致していない。
  - 入力フィールド内の次の文字が EOF である。
  - 変換する文字列が終了した。
- 変換する文字列に変換指示の一部ではない文字の並びが含まれている場合，この同じ文字の並びは入力の中に現れないようにしてください。本関数は一致する文字を走査しますが，格納はしません。不一致があった場合，一致していない最初の文字は読み取られていなかったかのように入力の中に残っています。

［使用例］

```
#include    <stdio.h>
void func(void){
    int         i, n;
    float       x;
    const char  *s;
    char        name[10];
    s = "23 11.1e-1 NAME";
    n = sscanf(s,"%d%f%s", &i, &x, name);    /*i に 23，x に 1.110000，name に“NAME”を格納，
                                                戻り値 n は 3*/
}
```

fscanf

ストリームからのデータ読み込みと解釈を行います。

備考 ルネサス エレクトロニクス製の統合デバッガ，システム・シミュレータでは，サポートされていません。

[所属]

標準ライブラリ libc.a

[指定形式]

```
#include <stdio.h>
int fscanf(FILE *stream, const char *format[, arg, ...]);
```

[戻り値]

走査，変換，格納が正常に実行できた入力フィールドの個数を返します。返却値には，格納されなかった走査済みフィールドは含まれません。ファイルの終わりで読み込もうとした場合，返却値は EOF です。フィールドが格納されなかった場合は，返却値は 0 です。

[詳細説明]

format の指す文字列で指定された書式に従い，その後ろに続く引数 *arg* を，変換された入力を格納するオブジェクトとして扱い，*stream* から変換する入力を読み込みます。*stream* に指定できるのは，標準入出力の stdin だけです。*format* の記述方法は sscanf と同様です。

scanf

標準出力ストリームからのテキストの読み込みと解釈を行います。

備考 ルネサス エレクトロニクス製の統合デバッガ，システム・シミュレータでは，サポートされていません。

[所属]

標準ライブラリ libc.a

[指定形式]

#include <stdio.h>

int scanf(const char **format*[, *arg*, ...]);

[戻り値]

走査，変換，格納が正常に実行できた入力フィールドの個数を返します。返却値には，格納されなかった走査済みフィールドは含まれません。ファイルの終わりで読み込もうとした場合，返却値は EOF です。フィールドが格納されなかった場合は，返却値は 0 です。

[詳細説明]

format の指す文字列で指定された書式に従い，その後ろに続く引数 *arg* を，変換された入力を格納するオブジェクトとして扱い，標準入出力の stdin から変換する入力を読み込みます。*format* の記述方法は sscanf と同様です。

[使用例]

```
#include    <stdio.h>
void func(void){
    int     i, n;
    double  x;
    char    name[10];
    n = scanf("%d%lf%s", &i, &x, name); /*“23 11.1e-1 NAME”の形式の stdin から入力を
                                        書式化入力 */
}
```

ungetc

入力ストリームへの文字押し戻しを行います。

備考 ルネサス エレクトロニクス製の統合デバッガ，システム・シミュレータでは，サポートされていません。

[所属]

標準ライブラリ libc.a

[指定形式]

```
#include <stdio.h>
int ungetc(int c, FILE *stream);
```

[戻り値]

文字 *c* を返します。
エラー・リターンはありません。

[詳細説明]

文字 *c* を *stream* が指す入力ストリームへ押し戻します。ただし，*c* が EOF の場合，押し戻しは行われません。

押し戻された文字 *c* は，次の文字入力の際，最初の文字として入力されることになります。本関数によって，押し戻すことができるのは 1 文字だけです。本関数を続けて実行した場合，効果があるのは最後だけです。*stream* に指定できるのは，標準入出力の stdin だけです。

rewind

ファイル位置指示子のリセットを行います。

備考 ルネサス エレクトロニクス製の統合デバッガ，システム・シミュレータでは，サポートされていません。

［所属］

標準ライブラリ libc.a

［指定形式］

```
#include <stdio.h>
void    rewind(FILE *stream);
```

［詳細説明］

stream が指す入力ストリームのエラー表示子をクリアし，ファイル位置表示子をファイルの先頭に位置付けます。

ただし，*stream* に指定できるのは，標準入出力の stdin だけです。そのため，本関数は ungetc による押し戻し文字を破棄する効果だけを持ちます。

perror

エラー処理を行います。

[所属]

標準ライブラリ libc.a

[指定形式]

#include <stdio.h>

void perror(const char *s);

[詳細説明]

グローバル変数 errno に対応するエラー・メッセージを stderr へ出力します。

出力されるメッセージは，次のようになります。

s が NULL でない場合	`fprintf(stderr, "%s:%s\n", s, s_fix);`
s が NULL の場合	`fprintf(stderr, "%s\n", s_fix);`

s_fix は，次のようになります。

errno が EDOM の場合	"EDOM error"
errno が ERANGE の場合	"ERANGE error"
errno が 0 の場合	"no error"
その他の場合	"error *xxx*"（*xxx* は abs (errno) % 1000）

[使用例]

```
#include    <stdio.h>
void func(double x){
    double  d;
    errno = 0;
    d = exp(x);
    if(errno)
        perror("func1");     /*exp で演算例外が発生した場合，perror を呼び出す */
}
```

6. 4. 7 標準ユーティリティ関数

標準ユーティリティ関数は，標準ライブラリ libc.a に収録されています。

なお，標準ユーティリティ関数として，以下のものがあります。

表 6―21 標準ユーティリティ関数

関数 / マクロ名	概要
abs	絶対値（int 型）を出力
labs	絶対値（long 型）を出力
bsearch	バイナリ検索
qsort	整列
div	除算（int 型）
ldiv	除算（long 型）
itoa	整数（int 型）を文字列に変換
ltoa	整数（long 型）を文字列に変換
ultoa	整数（unsigned long 型）を文字列に変換
ecvtf	浮動小数点値を数字文字列へ変換（総文字数指定）
fcvtf	浮動小数点値を数字文字列へ変換（小数点数字数指定）
gcvtf	浮動小数点値を数字文字列へ変換（書式指定）
atoi	文字列を整数（int 型）へ変換
atol	文字列を整数（long 型）へ変換
strtol	文字列を整数（long 型）へ変換し，最終文字列へのポインタを格納
strtoul	文字列を整数（unsigned long 型）へ変換し，最終文字列へのポインタを格納
atoff	文字列を浮動小数点数への変換
strtodf	文字列を浮動小数点数への変換（最終文字列へのポインタ格納）
calloc	メモリ割り当て（ゼロ初期化付き）
malloc	メモリ割り当て（ゼロ初期化なし）
realloc	メモリの再割り当て
free	メモリ開放
rand	疑似乱数列生成
srand	疑似乱数列の種類を設定

abs

絶対値（int 型）を出力します。

［所属］

標準ライブラリ libc.a

［指定形式］

#include <stdlib.h>

int abs(int *j*);

［戻り値］

j の絶対値 (*j* の大きさ)，|*j*| を返します。

［詳細説明］

j の絶対値（*j* の大きさ)，|*j*| を求めます。つまり，*j* が負の数の場合，結果は *j* の反転であり，負でない場合，*j* となります。

［使用例］

```
#include    <stdlib.h>
void func(int l){
    int val;
    val = -15;
    l = abs(val);    /*val の値の絶対値 15 を l に返す */
}
```

labs

絶対値（long 型）を出力します。

［所属］

標準ライブラリ libc.a

［指定形式］

```
#include <stdlib.h>
long    labs(long j);
```

［戻り値］

j の絶対値 (*j* の大きさ)，| *j* | を返します。

［詳細説明］

j の絶対値（*j* の大きさ)，| *j* | を求めます。つまり，*j* が負の数の場合，結果は *j* の反転であり，負でない場合，*j* となります。abs と同じですが，int 型の値の代わりに long 型を使用し，戻り値も long 型です。

bsearch

バイナリ検索を行います。

[所属]

標準ライブラリ libc.a

[指定形式]

#include <stdlib.h>

void* bsearch(const void *key, const void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void*));

[戻り値]

key と一致する配列の要素へのポインタを返します。一致する要素が複数ある場合，結果はその中で最初に見つかった要素を指します。*key* と一致する要素が見つからなかった場合，null ポインタを返します。

[詳細説明]

バイナリ検索法により，*base* から始まる配列の中で，*key* と一致する要素を検索します。*nmemb* は，配列の要素数です。*size* は，各要素のサイズです。配列は，*compar*（最後の引数）が指す比較関数に関し昇順で整列するようにしてください。*compar* が指す比較関数は，2 つの引数を持つように定義してください。結果は，1 番目の引数が 2 番目の引数よりも小さい場合は負，2 つの引数が一致する場合はゼロ，1 番目の引数が 2 番目の引数よりも大きい場合は正の整数を返すようにしてください。

[使用例]

```
#include    <stdlib.h>
#include    <string.h>
int compar(char **x, char **y);
void func(void){
    static  char    *base[] = {"a", "b", "c", "d", "e", "f"};
    char            *key = "c";     /* 検索キーは“c”*/
    char            **ret;
                                    /*ret に“c”へのポインタを格納 */
    ret = (char **) bsearch((char *) &key, (char *) base, 6, sizeof(char *), compar);
}
int compar(char **x, char **y){
    return(strcmp(*x, *y));          /* 引数を比較して正，ゼロ，または負の整数を返す */
}
```

qsort

整列します。

［所属］

標準ライブラリ libc.a

［指定形式］

#include <stdlib.h>

void　qsort(void *_base_, size_t _nmemb_, size_t _size_, int (*_compar_)(const void*, const void *));

［詳細説明］

base の指す配列を _compar_ が指す比較関数に関し昇順に整列します。_nmemb_ は配列の要素数，_size_ は各要素のサイズです。_compar_ が指す比較関数は bsearch と同様です。

div

除算（int 型）を行います。

［所属］

標準ライブラリ libc.a

［指定形式］

#include <stdlib.h>

div_t div(int *n*, int *d*);

［戻り値］

除算の結果を格納した構造体を返します。

［詳細説明］

int 型の値を除算する場合に使用します。

分子 *n* を分母 *d* で割ったその商と剰余を算出し，その 2 つの整数を次に示す構造体 div_t のメンバとして格納します。

```
typedef struct{
    int quot;
    int rem;
}div_t;
```

quot は商で，rem は剰余です。*d* がゼロでない場合，“*r* = div (*n*, *d*);”であれば，*n* は“*r*.rem + *d* * *r*.quot”に等しい値です。

d がゼロの場合，結果の quot メンバは，符号が *n* と同じで，大きさが表現可能な最大の大きさとなります。また，rem メンバは 0 です。

［使用例］

```
#include    <stdlib.h>
void func(void){
    div_t   r;
    r = div(110, 3);    /*r.quot には 36，r.rem には 2 を格納 */
}
```

ldiv

除算（long 型）を行います。

［所属］

標準ライブラリ libc.a

［指定形式］

#include <stdlib.h>

ldiv_t ldiv(long *n*, long *d*);

［戻り値］

除算の結果を格納した構造体を返します。

［詳細説明］

long 型の値を除算する場合に使用します。

分子 *n* を分母 *d* で割ったその商と剰余を算出し，その 2 つの整数を次に示す構造体 ldiv_t のメンバとして格納します。

```
typedef struct{
    long    quot;
    long    rem;
}ldiv_t;
```

quot は商で，rem は剰余です。*d* がゼロでない場合，“*r* = ldiv (*n*, *d*);”であれば，*n* は“*r*.rem + *d* * *r*.quot”に等しい値です。

d がゼロの場合，結果の quot メンバは，符号が *n* と同じで，大きさが表現可能な最大の大きさとなります。また，rem メンバは 0 です。

itoa

整数（int 型）を文字列に変換します。

［所属］

標準ライブラリ libc.a

［指定形式］

#include <stdlib.h>

char *itoa(int *value*, char **string*, int *radix*);

［戻り値］

string を返します。

［詳細説明］

int 型数値 *value* を *radix* 進数の文字列に変換して，*string* の示す配列に格納します。文字列の終わりには終端を示す null 文字（\0）が常に付加されます。*radix* に指定できるのは，2 から 36 までの数値です。*radix* が 10 の場合，*value* は符号付き数値として扱われ，*value* < 0 の場合文字列の先頭に’-’文字が付きます。その他の場合，*value* は符号なし数値として扱われます。*radix* > 10 の場合，10 から 35 に英小文字の a から z が当てられます。

［使用例］

```
#include    <stdlib.h>
void func(void){
    char    buf[128];
    itoa(12345, buf, 16);    /*12345 を 16 進数文字列に変換 */
}
```

ltoa

整数（long 型）を文字列に変換します。

［所属］

標準ライブラリ libc.a

［指定形式］

#include <stdlib.h>

char *ltoa(long int *value*, char **string*, int *radix*);

［戻り値］

string を返します。

［詳細説明］

long int 型数値 *value* を *radix* 進数の文字列に変換して，*string* の示す配列に格納します。*value* の型を除き，itoa と同じです。

ultoa

整数（unsigned long 型）を文字列に変換します。

［所属］

標準ライブラリ libc.a

［指定形式］

#include <stdlib.h>

char *ultoa(unsigned long int *value*, char **string*, int *radix*);

［戻り値］

string を返します。

［詳細説明］

unsigned long int 型数値 *value* を *radix* 進数の文字列に変換して，*string* の示す配列に格納します。*value* の型を除き，itoa と同じです。

ecvtf

浮動小数点値を数字文字列へ変換（総文字数指定）します。

［所属］

標準ライブラリ libc.a

［指定形式］

#include <stdlib.h>

char *ecvtf(float *val*, int *chars*, int **decpt*, int **sgn*);

［戻り値］

val の文字列表現を含む新しい文字列を指すポインタを返します。

［詳細説明］

float 型数値 *val* を数字で表した（null 文字（\0）で終端する）文字列を生成します。2 番目の引数 *chars* には，書き込む総文字数を指定します（数字のみを書き込むので変換された文字列の中の有効数字の数でもあります）。常に，*val* の整数部の桁がすべて含まれます。

［使用例］

```
#include    <stdlib.h>
void func(void){
    float   val;
    int     dec, sgn;
    val = 111.11;
    ecvtf(val, 12, &dec, &sgn); /*val の値 111.11 を 12 文字の文字列へ変換
                                    dec には小数点の左側の桁数 3，sgn には符号（正のため 0）を記録 */
}
```

fcvtf

浮動小数点値を数字文字列へ変換（小数点数字数指定）します。

［所属］

標準ライブラリ libc.a

［指定形式］

#include <stdlib.h>

char *fcvtf(float *val*, int *decimals*, int **decpt*, int **sgn*);

［戻り値］

val の文字列表現を含む新しい文字列を指すポインタを返します。

［詳細説明］

2 番目の引数の解釈以外は ecvtf と同じです。2 番目の引数 *decimals* には，小数点後に書き込む文字の数を指定します。ecvtf と本関数は出力文字列の中に数字だけを書き込むので，小数点の位置を **decpt* に，数値の符号を **sgn* に記録しておきます。数をフォーマットしたあと，**decpt* には小数点の左側の桁数が入ります。**sgn* には，数値が正の場合は 0 が，負の場合は 1 が入ります。

gcvtf

浮動小数点値を数字文字列へ変換（書式指定）します。

［所属］

標準ライブラリ libc.a

［指定形式］

```
#include <stdlib.h>
char    *gcvtf(float val, int prec, char *buf);
```

［戻り値］

val の書式付き文字列表現へのポインタ（引数 *buf* と同じです）を返します。

［詳細説明］

数値を文字列に書式変換し，バッファ *buf* に格納します。本関数は，sprintf の書式“%.*prec*”（負数だけに符号が付く）と同じ規則を使用し，有効桁数（*prec* で指定）に応じて，指数形式か，または通常の小数形式を選択します。

atoi

文字列を整数（int 型）へ変換します。

［所属］

標準ライブラリ libc.a

［指定形式］

#include <stdlib.h>

int atoi(const char *_str_);

［戻り値］

部分文字列が変換できた場合，変換された値を返します。変換できなかった場合，0 を返します。

［詳細説明］

str の指す文字列の最初の部分を int 型の表現に変換します。本関数は，“(int) strtol (*str*, NULL, 10)”と同じです。

atol

文字列を整数（long 型）へ変換します。

［所属］

標準ライブラリ libc.a

［指定形式］

```
#include <stdlib.h>
long    atol(const char *str);
```

［戻り値］

部分文字列が変換できた場合，変換された値を返します。変換できなかった場合，0 を返します。

［詳細説明］

str の指す文字列の最初の部分を long int 型の表現に変換します。本関数は，“strtol (*str*, NULL, 10)”と同じです。

strtol

文字列を整数（long 型）へ変換し，最終文字列へのポインタを格納します。

［所属］

標準ライブラリ libc.a

［指定形式］

```
#include <stdlib.h>
long    strtol(const char *str, char **ptr, int base);
```

［戻り値］

部分文字列が変換できた場合，変換された値を返します。変換できなかった場合，0 を返します。

オーバーフローが生じる（変換された値の大きさが大きすぎる）場合，LONG_MAX，または LONG_MIN を返し，マクロ ERANGE をグローバル変数 errno にセットします。

［詳細説明］

str の指す文字列の最初の部分を long 型の表現に変換します。まず，入力文字を次の 3 つの部分，“最初の空白類”，“*base* の値により定められる基数において表現され，整数にする対象となる列”，“（null 文字（\0）を含む）最後の 1 個以上の認識されない文字列”に分割します。その後，対象となる列を整数へ変換し，その結果を返します。

（1）引数 *base* は，0，または 2 ～ 36 を指定します。

（a）*base* が 0 の場合

対象となる文字列の予期される形式は，オプショナルな + 符号，または - 符号，16 進数であることを示す“0x”を前に持つ整数の形式となります。

（b）*base* の値が 2 ～ 36 の場合

対象となる文字列の予期される形式は，オプショナルな + 符号，または - 符号を前に持ち，*base* によって基数が指定された整数を表す文字列，または数字列となります。“a”（，または“A”）から“z”（，または“Z”）までの英字は 10 から 35 までの値を示すものとみなされます。与えられた値が *base* よりも小さい英字しか使用できません。

（c）*base* の値が 16 の場合

“0x”が文字と数字の列の前（符号が存在する場合は符号の後ろ）に置かれます（省略可能）。

(2) 対象となる列は，空白類以外の最初の文字で始まり，予期される形式を持つ入力文字列の先頭部分の最長の部分列として定義されます。

(a) 入力の文字列が空である場合やすべて空白類で構成されている場合，または空白類でない最初の文字が符号でも許容されうる文字でも数字でもない場合，対象となる列は空となります。

(b) 対象となる列が予期される形式を持ち，かつ，*base* の値が 0 の場合，入力文字列から基数を判断します。0x が先行する文字列は，16 進数数値とみなされ，先行 0 が付いていて x が付いていない文字列は 8 進数としてみなされます。他の文字列はすべて 10 進数としてみなされます。

(c) *base* が 2 から 36 までの間の値の場合，上述のように，これを変換用基数として使用します。

(d) 対象となる列が - 符号で始まる場合，変換結果の値の符号は反転されます。

(3) 最初の文字列を指すポインタ

(a) *ptr* が null ポインタでない場合，*ptr* の指すオブジェクトの中に格納されます。

(b) 対象となる列が空である場合，または予期された形式を持たない場合，変換は行われません。*str* の値は，*ptr* が null ポインタでない場合，*ptr* の指すオブジェクトに格納されます。

備考 本関数は，リエントラントではありません。

[使用例]

```
#include    <stdlib.h>
void func(long ret){
    char    *p;
    ret = strtol("10", &p, 0);      /*ret に 10 を返す */
    ret = strtol("0x10", &p, 0);    /*ret に 16 を返す */
    ret = strtol("10x", &p, 2);     /*ret に 2 を返し，p の領域には‘x’へのポインタを格納 */
    ret = strtol("2ax3", &p, 16);   /*ret に 42 を返し，p の領域には‘x’へのポインタを格納 */
      :
}
```

strtoul

文字列を整数（unsigned long 型）へ変換し，最終文字列へのポインタを格納します。

［所属］

標準ライブラリ libc.a

［指定形式］

#include <stdlib.h>

unsigned long strtoul(const char **str*, char ***ptr*, int *base*);

［戻り値］

部分文字列が変換できた場合，変換された値を返します。変換できなかった場合，0 を返します。

オーバーフローが生じる場合，ULONG_MAX を返し，マクロ ERANGE をグローバル変数 errno にセットします。

［詳細説明］

返却値の型が unsigned long 型になること以外，strtol と同じです。

atoff

文字列を浮動小数点数への変換を行います。

［所属］

標準ライブラリ libc.a

［指定形式］

```
#include <stdlib.h>
float   atoff(const char *str);
```

［戻り値］

部分文字列が変換できた場合，その値を返します。変換できなかった場合，0 を返します。オーバーフローが生じる（値が表現可能な値の範囲にない）場合，HUGE_VAL，または -HUGE_VAL を返し，ERANGE をグローバル変数 errno にセットします。アンダフローが生じる場合，0 を返し，マクロ ERANGE をグローバル変数 errno にセットします。

［詳細説明］

str の指す文字列の最初の部分を float 型の表現に変換します。本関数は，“strtodf (*str*, NULL)”と同じです。

strtodf

文字列を浮動小数点数への変換（最終文字列へのポインタ格納）を行います。

[所属]

標準ライブラリ libc.a

[指定形式]

```
#include <stdlib.h>
float   strtodf(const char *str, char **ptr);
```

[戻り値]

部分文字列が変換できた場合，その値を返します。変換できなかった場合，0 を返します。オーバーフローが生じる（値が表現可能な値の範囲にない）場合，HUGE_VAL，または -HUGE_VAL を返し，ERANGE をグローバル変数 errno にセットします。アンダフローが生じる場合，0 を返し，マクロ ERANGE をグローバル変数 errno にセットします。

[詳細説明]

str の指す文字列の最初の部分を float 型の表現に変換します。変換される部分文字列は，次の形式の，空白でない通常の文字から始まる，*str* の最長先頭部分文字列です。

[+ | -] digits [.] [digits] [(e | E) [+ | -] digits]

str が空か，または空白文字だけから成り立っている場合，および最初の通常文字が“+，“-”，“.”，または数字以外の場合，部分文字列には文字が含まれていません。部分文字列が空の場合，変換は行われず，str の値が *ptr* の指す領域に格納されます。空でない場合，部分文字列は変換され，最終文字列（少なくとも str の終端を示す null 文字（\0）を含む）へのポインタが *ptr* の指す領域に格納されます。

備考 本関数は，リエントラントではありません。

［使用例］

```
#include    <stdlib.h>
#include    <stdio.h>
void func(float ret){
    char    *p, *str, s[30];
    str = "+5.32a4e";
    ret = strtodf(str, &p);          /*ret には 5.320000 を返し，p の領域には‘a’へのポインタを
                                       格納 */
    sprintf(s, "%lf\t%c", ret, *p); /*s の指す配列に“5.320000 a”を格納 */
}
```

calloc

メモリ割り当て（ゼロ初期化付き）を行います。

[所属]

標準ライブラリ libc.a

[指定形式]

#include <stdlib.h>

void *calloc(size_t *nmemb*, size_t *size*);

[戻り値]

領域の割り付けに成功した場合，その領域へのポインタを返します。割り付けができなかった場合，null ポインタを返します。

[詳細説明]

大きさが *size* の，要素数 *nmemb* 個の配列領域を割り付けます。割り付けられた領域は 0 で初期化されます。

[注意事項]

記憶域管理の関数は，ヒープ・メモリ領域から必要に応じて自動的にメモリ領域を確保します。

また，コンパイラは自動でこの領域を確保しないため，本関数を利用する場合は，ヒープ・メモリ領域を確保する必要があります。領域の確保は，アプリケーションの最初で行ってください。

【ヒープ・メモリ設定例】

```
#define SIZEOF_HEAP 0x1000
int __sysheap[SIZEOF_HEAP >> 2];
size_t  __sizeof_sysheap = SIZEOF_HEAP;
```

備考 1. 変数“__sysheap”（アンダースコア‘_’は 2 つ）のシンボル“___sysheap”（アンダースコア‘_’は 3 つ）は，ヒープ・メモリの先頭アドレスを指します。この値は，4 の倍数にしてください。

2. 変数“__sizeof_sysheap”（最初のアンダースコア‘_’は 2 つ）に，必要なヒープ・メモリのサイズ（バイト）を設定してください。アセンブラ命令で記述する場合，シンボル“___sizeof_sysheap”（最初のアンダースコア‘_’は 3 つ）に設定してください。

［使用例］

```
#include    <stdlib.h>
typedef struct {
    double  d[3];
    int     i[2];
} s_data;
int func(void){
    sdata   *buf;
    if((buf = calloc(40, sizeof(s_data))) == NULL)  /*s_data40 個のための領域を割り付け */
        return(1);
    /*  処理を記述  */
    free(buf);                                      /* 領域を開放 */
    return(0);
}
```

malloc

メモリ割り当て（ゼロ初期化なし）を行います。

[所属]

標準ライブラリ libc.a

[指定形式]

#include <stdlib.h>

void *malloc(size_t *size*);

[戻り値]

領域の割り付けに成功した場合，その領域へのポインタを返します。割り付けができなかった場合，null ポインタを返します。

[詳細説明]

大きさ *size* の領域を割り付けます。領域は初期化されません。

[注意事項]

記憶域管理の関数は，ヒープ・メモリ領域から必要に応じて自動的にメモリ領域を確保します。

また，コンパイラは自動でこの領域を確保しないため，本関数を利用する場合は，ヒープ・メモリ領域を確保する必要があります。領域の確保は，アプリケーションの最初で行ってください。

【ヒープ・メモリ設定例】

```
#define SIZEOF_HEAP 0x1000
int __sysheap[SIZEOF_HEAP >> 2];
size_t  __sizeof_sysheap = SIZEOF_HEAP;
```

備考 1. 変数“__sysheap”（アンダースコア‘_’は 2 つ）のシンボル“___sysheap”（アンダースコア‘_’は 3 つ）は，ヒープ・メモリの先頭アドレスを指します。この値は，4 の倍数にしてください。

2. 変数“__sizeof_sysheap”（最初のアンダースコア‘_’は 2 つ）に，必要なヒープ・メモリのサイズ（バイト）を設定してください。アセンブラ命令で記述する場合，シンボル“___sizeof_sysheap”（最初のアンダースコア‘_’は 3 つ）に設定してください。

realloc

メモリの再割り当てを行います。

[所属]

標準ライブラリ libc.a

[指定形式]

```
#include <stdlib.h>
void    *realloc(void *ptr, size_t size);
```

[戻り値]

領域の割り付けに成功した場合，その領域へのポインタを返します。割り付けができなかった場合，null ポインタを返します。

[詳細説明]

ptr が指す領域の大きさを，*size* の大きさに変更します。以前の大きさと，*size* の小さい方までの領域の内容は変わりません。領域を拡張する場合の，以前の大きさ以降の領域内容は初期化されません。*ptr* が null ポインタのときは，“malloc (*size*)”と同じ動作をします。それ以外の場合，*ptr* には，calloc，malloc，および本関数で獲得した領域を指定しなければなりません。

[注意事項]

記憶域管理の関数は，ヒープ・メモリ領域から必要に応じて自動的にメモリ領域を確保します。

また，コンパイラは自動でこの領域を確保しないため，本関数を利用する場合は，ヒープ・メモリ領域を確保する必要があります。領域の確保は，アプリケーションの最初で行ってください。

【ヒープ・メモリ設定例】

```
#define SIZEOF_HEAP 0x1000
int __sysheap[SIZEOF_HEAP >> 2];
size_t  __sizeof_sysheap = SIZEOF_HEAP;
```

備考 1. 変数“__sysheap”（アンダースコア‘_’は 2 つ）のシンボル“___sysheap”（アンダースコア‘_’は 3 つ）は，ヒープ・メモリの先頭アドレスを指します。この値は，4 の倍数にしてください。

2. 変数“__sizeof_sysheap”（最初のアンダースコア‘_’は 2 つ）に，必要なヒープ・メモリのサイズ（バイト）を設定してください。アセンブラ命令で記述する場合，シンボル“___sizeof_sysheap”（最初のアンダースコア‘_’は 3 つ）に設定してください。

free

メモリ開放を行います。

[所属]

標準ライブラリ libc.a

[指定形式]

#include <stdlib.h>

void free(void *ptr);

[詳細説明]

ptr が指す領域を開放し，その後の割り付けに使用できるようにします。*ptr* には，calloc，malloc，および realloc で獲得した領域を指定しなければなりません。

[使用例]

```
#include    <stdlib.h>
typedef struct {
    double  d[3];
    int     i[2];
} s_data;
int func(void){
    sdata   *buf;
s
    if((buf = calloc(40, sizeof(s_data))) == NULL)  /*s_data40 個のための領域を割り付け */
        return(1);
    /*  処理を記述  */
    free(buf);                                      /* 領域を開放 */
    return(0);
}
```

rand

疑似乱数列生成を行います。

［所属］

標準ライブラリ libc.a

［指定形式］

#include <stdlib.h>

int rand(void);

［戻り値］

乱数を返します。

［詳細説明］

0 以上 RAND_MAX 以下の乱数を返します。

［使用例］

```
#include    <stdlib.h>
void func(void){
    if(rand() & 0xf) < 4)
        func1();    /*25% の確率で func1 を実行 */
}
```

srand

疑似乱数列の種類を設定します。

［所属］

標準ライブラリ libc.a

［指定形式］

#include <stdlib.h>

void　srand(unsigned int *seed*);

［詳細説明］

後続する rand の呼び出しで使用する新しい疑似乱数列の種として，*seed* を与えます。本関数を同じ *seed* の値で呼んだ場合，rand により得られる乱数は，同じ値が同じ順番で現れることになります。本関数を実行せずに rand を実行した場合，最初に“srand (1)”を実行した場合と同じ結果になります。

6. 4. 8 非局所分岐関数

非局所分岐関数は，標準ライブラリ libc.a に収録されています。

なお，非局所分岐関数として，以下のものがあります。

表 6―22 非局所分岐関数

関数 / マクロ名	概要
longjmp	非局所分岐
setjmp	非局所分岐の分岐先をセット

longjmp

非局所分岐を行います。

[所属]

標準ライブラリ libc.a

[指定形式]

#include <setjmp.h>

void longjmp(jmp_buf *env*, int *val*);

[戻り値]

第二引数 *val* を返します。ただし，*val* が 0 の場合，1 を返します。

[詳細説明]

setjmp で保存された *env* を使い，setjmp 直後へ非局所分岐します。*val* は，setjmp の返却値となります。

[使用例]

```
#include    <setjmp.h>
#define ERR_XXX1    1
jmp_buf jmp_env;
void func(void){
    for(;;){
        switch(setjmp(jmp_env)){
            case    ERR_XXX1:   /*error XXX1 の終結処理 */
                break;
            case    0:          /* 非局所分岐ではない */
            default:
                    break;
        }
    }
}
void func1(void){
    longjmp(jmp_env, ERR_XXX1); /*error XXX1 の発生により非局所分岐 */
}
```

setjmp

非局所分岐の分岐先をセットします。

［所属］

標準ライブラリ libc.a

［指定形式］

#include <setjmp.h>

int setjmp(jmp_buf *env*);

［戻り値］

0 を返します。

［詳細説明］

非局所分岐のための戻り先を *env* にセットします。*env* には，本関数が実行された時点の環境が保存されます。

6. 4. 9 数学関数

数学関数は，数学ライブラリ libm.a に収録されています。

なお，数学関数として，以下のものがあります。

表 6―23 数学関数

関数 / マクロ名	概要
j0f	第一種ベッセル関数（0 次）
j1f	第一種ベッセル関数（1 次）
jnf	第一種ベッセル関数（*n* 次）
y0f	第二種ベッセル関数（0 次）
y1f	第二種ベッセル関数（1 次）
ynf	第二種ベッセル関数（*n* 次）
erff	誤差関数（近似値）
erfcf	誤差関数（相補確率）
expf	指数関数
logf	対数関数（自然対数）
log2f	対数関数（底 = 2）
log10f	対数関数（底 = 10）
powf	べき乗関数
sqrtf	平方根関数
cbrtf	立方根関数
ceilf	ceiling 関数
fabsf	絶対値関数
floorf	floor 関数
fmodf	剰余関数
frexpf	浮動小数点数を仮数部とべき乗に分割
ldexpf	浮動小数点数をべき乗に変換
modff	浮動小数点数を整数部と小数部に分割
gammaf	対数ガンマ関数
hypotf	ユークリッド距離関数
matherr	エラー処理関数
cosf	余弦
sinf	正弦
tanf	正接
acosf	逆余弦
asinf	逆正弦
atanf	逆正接
atan2f	逆正接（y / x）

関数 / マクロ名	概要
coshf	双曲線余弦
sinhf	双曲線正弦
tanhf	双曲線正接
acoshf	逆双曲線余弦
asinhf	逆双曲線正弦
atanhf	逆双曲線正接

j0f

第一種ベッセル関数（0 次）です。

［所属］

数学ライブラリ libm.a

［指定形式］

```
#include <math.h>
float   j0f(float x);
```

［戻り値］

0 次の第一種ベッセル関数値を返します。

［詳細説明］

0 次の第一種ベッセル関数値を求めます。

j1f

第一種ベッセル関数（1 次）です。

［所属］

数学ライブラリ libm.a

［指定形式］

#include <math.h>

float j1f(float *x*);

［戻り値］

1 次の第一種ベッセル関数値を求めます。

［詳細説明］

1 次の第一種ベッセル関数値を返します。

［使用例］

```
#include    <math.h>
float func(void){
    float   ret, x;
    ret = j1f(x);    /*x の値に対して，1 次の第一種ベッセル関数値を求め，ret に返す */
      :
    return(ret);
}
```

jnf

第一種ベッセル関数（*n* 次）です。

［所属］

数学ライブラリ libm.a

［指定形式］

```
#include <math.h>
float   jnf(int n, float x);
```

［戻り値］

n 次の第一種ベッセル関数値を返します。

［詳細説明］

n 次の第一種ベッセル関数値を求めます。

y0f

第二種ベッセル関数（0 次）です。

［所属］

数学ライブラリ libm.a

［指定形式］

```
#include <math.h>
float   y0f(float x);
```

［戻り値］

0 次の第二種ベッセル関数値を返します。

［詳細説明］

0 次の第二種ベッセル関数値を求めます。

y1f

第二種ベッセル関数（1 次）です。

［所属］

数学ライブラリ libm.a

［指定形式］

#include <math.h>

float y1f(float *x*);

［戻り値］

1 次の第二種ベッセル関数値を返します。

［詳細説明］

1 次の第二種ベッセル関数値を求めます。

ynf

第二種ベッセル関数（*n* 次）です。

［所属］

数学ライブラリ libm.a

［指定形式］

#include <math.h>

float ynf(int *n*, float *x*);

［戻り値］

n 次の第二種ベッセル関数値を返します。

［詳細説明］

n 次の第二種ベッセル関数値を求めます。

erff

誤差関数（近似値）です。

［所属］

数学ライブラリ libm.a

［指定形式］

#include <math.h>

float erff(float *x*);

［戻り値］

“誤差関数”の近似値（0 と 1 の間の数値）を返します。

［詳細説明］

観測値が平均の *x* 標準偏差の範囲になる確率を推定する“誤差関数”の近似値（0 と 1 の間の数値）を求めます。

［使用例］

```
#include    <math.h>
float func(void){
    float   ret, x;
    ret = erff(x);   /*x の値に対して，誤差関数の近似値を求め，ret に返す */
       :
    return(ret);
}
```

erfcf

誤差関数（相補確率）です。

［所属］

数学ライブラリ libm.a

［指定形式］

```
#include <math.h>
float   erfcf(float x);
```

［戻り値］

相補確率を返します。

［詳細説明］

“1.0 - erff (*x*)”をして相補確率を求めます。これは，値の大きな *x* について“erff (*x*)”が呼び出された場合，その結果を 1.0 から引かれると精度が損なわれるために用意されています。

expf

指数関数です。

[所属]

数学ライブラリ libm.a

[指定形式]

```
#include <math.h>
float   expf(float x);
```

[戻り値]

e の *x* 乗を返します。

アンダフローが生じた場合（*x* が結果を表せない大きさの負の数の場合），非正規化数を返し，グローバル変数 errno にマクロ ERANGE をセットします。オーバーフローが生じた場合（*x* が大きすぎる数の場合），HUGE_VAL（表現可能な最大の double 型数値）を返し，グローバル変数 errno にマクロ ERANGE をセットします。

[詳細説明]

e の *x* 乗を求めます（e は自然対数の底で，約 2.71828 です）。

備考 matherr を使用して，本関数のエラー処理を変更できます。

logf

対数関数（自然対数）です。

［所属］

数学ライブラリ libm.a

［指定形式］

#include <math.h>

float　logf(float *x*);

［戻り値］

x の自然対数を返します。

x が負の場合非数を返し，グローバル変数 errno にマクロ EDOM をセットします。*x* が 0 の場合，- ∞（0xff800000）を返し，グローバル変数 errno にマクロ ERANGE をセットします。

［詳細説明］

x の自然対数，つまり，底を e としてその対数を求めます。

備考　matherr を使用して，本関数のエラー処理を変更できます。

log2f

対数関数（底 = 2）です。

［所属］

数学ライブラリ libm.a

［指定形式］

#include <math.h>

float　log2f(float *x*);

［戻り値］

2 を底とする *x* の対数を返します。

x が負の場合非数を返し，グローバル変数 errno にマクロ EDOM をセットします。*x* が 0 の場合，- ∞を返し，グローバル変数 errno にマクロ ERANGE をセットします。

［詳細説明］

2 を底とする *x* の対数を求めます。これは“log (*x*) / log (2)”によって実現されています。

備考　matherr を使用して，本関数のエラー処理を変更できます。

log10f

対数関数（底 = 10）です。

［所属］

数学ライブラリ libm.a

［指定形式］

#include <math.h>

float　log10f(float *x*);

［戻り値］

10 を底とする *x* の対数を返します。

x が負の場合非数を返し，グローバル変数 errno にマクロ EDOM をセットします。x が 0 の場合，- ∞を返し，グローバル変数 errno にマクロ ERANGE をセットします。

［詳細説明］

10 を底とする *x* の対数を求めます。これは，“log (*x*) / log (10)”によって実現されています。

備考　matherr を使用して，本関数のエラー処理を変更できます。

powf

べき乗関数です。

[所属]

数学ライブラリ libm.a

[指定形式]

#include <math.h>

float powf(float *x*, float *y*);

[戻り値]

x の *y* 乗を返します。

x < 0 かつ *y* が奇整数の場合にのみ負の解を返します。*x* < 0 で *y* が非整数の場合，または *x* = *y* = 0 の場合，非数を返し，グローバル変数 errno にマクロ EDOM をセットします。*x* = 0 かつ *y* < 0 の場合，またはオーバーフローが発生した場合，± HUGE_VAL を返し，errno にマクロ ERANGE をセットします。解が 0 へ向かって消滅した場合，± 0 を返し，errno に ERANGE をセットします。解が非正規化数の場合，errno に ERANGE をセットします。

[詳細説明]

x の *y* 乗を求めます。

備考 matherr を使用して，本関数のエラー処理を変更できます。

[使用例]

```
#include    <math.h>
float func(void){
    float   ret, x, y;
    ret = powf(x, y);    /*x を y 乗した値を ret に返す */
      :
    return(ret);
}
```

sqrtf

平方根関数です。

［所属］

数学ライブラリ libm.a

［指定形式］

```
#include <math.h>
float   sqrtf(float x);
```

［戻り値］

x の正の平方根を返します。

x が実数で負の場合，非数を返し，グローバル変数 errno にマクロ EDOM をセットします。

［詳細説明］

x の正の平方根を求めます。

備考 matherr を使用して，本関数のエラー処理を変更できます。

cbrtf

立方根関数です。

［所属］

数学ライブラリ libm.a

［指定形式］

```
#include <math.h>
float   cbrtf(float x);
```

［戻り値］

x の立方根を返します。

［詳細説明］

x の立方根を求めます。

備考　matherr を使用して，本関数のエラー処理を変更できます。

ceilf

ceiling 関数です。

［所属］

数学ライブラリ libm.a

［指定形式］

```
#include <math.h>
float   ceilf(float x);
```

［戻り値］

x 以上の最小の整数値を返します。

［詳細説明］

x 以上の最小の整数値を求めます。

備考　matherr を使用して，本関数のエラー処理を変更できます。

fabsf

絶対値関数です。

［所属］

数学ライブラリ libm.a

［指定形式］

#include <math.h>

float　fabsf(float *x*);

［戻り値］

x の絶対値（大きさ）を返します。

［詳細説明］

x のビット表現を直接操作して，*x* の絶対値（大きさ）を求めます。

備考　matherr を使用して，本関数のエラー処理を変更できます。

floorf

floor 関数です。

［所属］

数学ライブラリ libm.a

［指定形式］

```
#include <math.h>
float   floorf(float x);
```

［戻り値］

x 以下の最大の整数値を返します。

［詳細説明］

x 以下の最大の整数値を求めます。

備考　matherr を使用して，本関数のエラー処理を変更できます。

fmodf

剣余関数です。

[所属]

数学ライブラリ libm.a

[指定形式]

#include <math.h>

float fmodf(float *x*, float *y*);

[戻り値]

x を *y* で割った剰余である浮動小数点数値を返します。

“fmodf (*x*, 0)”は，*x* を返します。

[詳細説明]

x を *y* で割った剰余である浮動小数点数値を求めます。つまり，*y* が 0 でない場合に，その結果の符号が *x* と同じ符号で大きさが *y* よりも小さい最大整数 i に対し，値“*x* - i * *y*”を求めます。

備考 matherr を使用して，本関数のエラー処理を変更できます。

[使用例]

```
#include    <math.h>
void func(void){
    float   ret, x, y;
    ret = fmodf(x, y);  /*x を y で割った剰余を ret に返す */
      :
}
```

frexpf

浮動小数点数を仮数部とべき乗に分割します。

[所属]

数学ライブラリ libm.a

[指定形式]

#include <math.h>

float frexpf(float *val*, int **exp*);

[戻り値]

仮数部 m を返します。

val が 0 の場合，**exp* に 0 をセットし，0 を返します。

[詳細説明]

float 型の *val* を仮数部 m と 2 の p 乗で表します。結果の仮数部 m は，*val* が 0 でないかぎり，0.5 <= | x | < 1.0 となります。p は **exp* に格納されます。m，および p は，*val* = m * 2^p となるように計算されます。

[使用例]

```
#include    <math.h>
void func(void){
    float   ret, x;
    int     exp;
    x = 5.28;
    ret = frexpf(x, &exp);  /* 結果の仮数部 0.66 が ret に返り，exp には 3 を格納 */
      :
}
```

ldexpf

浮動小数点数をべき乗に変換します。

［所属］

数学ライブラリ libm.a

［指定形式］

#include <math.h>

float ldexpf(float *val*, int *exp*);

［戻り値］

val * 2 ^{exp} で求めた値を返します。

アンダフロー，またはオーバーフローが生じた場合，グローバル変数 errno にマクロ ERANGE がセットされます。アンダフローの場合，非正規化数を返します。オーバーフローの場合，HUGE_VAL と同じ符号∞（+ ∞ = 0x7f800000，- ∞ = 0xff800000）を返します。

［詳細説明］

val * 2 ^{exp} を求めます。

備考 matherr を使用して，本関数のエラー処理を変更できます。

modff

浮動小数点数を整数部と小数部に分割します。

［所属］

数学ライブラリ libm.a

［指定形式］

#include <math.h>

float modff(float *val*, float **ipart*);

［戻り値］

小数部を返します。結果の符号は *val* の符号と同じです。

［詳細説明］

float 型の *val* を整数部と小数部とに分割し，整数部を **ipart* に格納します。丸めは行いません。整数部と小数部の和は，正確に *val* と一致するように保証されています。たとえば，“*realpart* = modff (*val*, *&intpart*)”であるとき，“*realpart* + *intpart*”は *val* と一致します。

gammaf

対数ガンマ関数です。

[所属]

数学ライブラリ libm.a

[指定形式]

#include <math.h>

float gammaf(float *x*);

[戻り値]

x のガンマ関数の自然対数を返します。

x が 0 のとき，またはオーバーフローが生じた場合，HUGE_VAL を返し，グローバル変数 errno にマクロ ERANGE をセットします。

[詳細説明]

“ln (Γ (*x*))”，つまり，*x* のガンマ関数の自然対数を求めます。ガンマ関数“expf (gammaf (*x*))”は階乗の一般化であり，Γ (N) ≡ N * Γ (N - 1) という関係式を持っています。したがって，ガンマ関数自体の結果は非常に早く大きくなります。そのため，本関数は，表現可能な結果の有効範囲を拡大するために，単なる“Γ (*x*)”ではなく“ln (Γ (*x*))”として定義されています。

備考 matherr を使用して，本関数のエラー処理を変更できます。

[使用例]

```
#include    <math.h>
float func(float x){
    float   ret;
    ret = gammaf(x);     /*x のガンマ関数の自然対数を ret に返す */
      :
    return(ret);
}
```

hypotf

ユークリッド距離関数です。

[所属]

数学ライブラリ libm.a

[指定形式]

#include <math.h>

float hypotf(float *x*, float *y*);

[戻り値]

原点（0, 0）とカーテシアン座標（*x*, *y*）で表される点との間のユークリッド距離“$\sqrt{(x^2+y^2)}$”を返します。

オーバーフローが生じた場合，HUGE_VAL を返し，グローバル変数 errno にはマクロ ERANGE をセットします。

[詳細説明]

原点（0, 0）とカーテシアン座標（*x*, *y*）で表される点との間のユークリッド距離“$\sqrt{(x^2+y^2)}$”を求めます。

備考 matherr を使用して，本関数のエラー処理を変更できます。

[使用例]

```
#include    <math.h>
void func(float x){
    float   ret, y;
    ret = hypotf(x, y); /* (0, 0) 座標と (x, y) 座標の間のユークリッド距離を ret に返す */
}
```

matherr

エラー処理関数です。

[所属]

数学ライブラリ libm.a

[指定形式]

#include <math.h>

int matherr(struct exception *e);

[戻り値]

e->retval の値を変更することにより，カスタマイズした本関数からの呼び出し関数の結果を変更できます。これは，呼び出し側の関数にも伝播します。本関数は，エラーを解決した場合，0 以外の値を返し，エラーを解決できなかった場合，0 を返します。本関数が 0 を返す場合，呼び出し側でグローバル変数 errno に適切な値をセットします。

[詳細説明]

数学ライブラリ関数内でエラーが発生した場合に呼ばれる関数です。

したがって，matherr という名前の関数をユーザ・サブルーチンで用意することにより，エラー処理をカスタマイズできます。カスタマイズする matherr は，エラーの解決に失敗した場合に 0 を返し，エラーを解決した場合に 0 以外の値を返すようにする必要があります。matherr が 0 以外の値を返した場合，グローバル変数 errno の値は変更されません。

エラー処理のカスタマイズは，構造体 exception へのポインタ *e で渡された情報を利用して行うことができます。構造体 exception は“math.h”中で次のように定義されています。

```
#if !defined(__cplusplus)
#define __exception exception
#endif
struct  exception{
    int     type;
    char    *name;
    double  arg1, arg2, retval;
};
```

各メンバの意味は，次のとおりです。

type	発生した数学関数エラーのタイプです。 マクロ・エンコーディング・エラーのタイプも“math.h”の中で定義されています。
name	エラーが発生した数学ライブラリ関数の名前を保持し，空文字で終わっている文字列を指すポインタです。
arg1，arg2	エラーの原因となった引数です。
retval	呼び出し関数が返すエラー・リターン値です。

発生する可能性のある数学ライブラリ関数エラーのタイプは，次のとおりです。

DOMAIN	引数が関数の定義域の範囲にない 例 logf (-1);
OVERFLOW	オーバーフロー 例 expf (1000);
UNDERFLOW	アンダフロー，非正規化数の解 解 < 1.1755e -38 かつ非 0 の数で，精度が通常の数値より落ちた状態
Z_DIVISION	ゼロ除算

備考 演算例外発生時の matherr の呼び出しと，標準関数でのグローバル変数 errno の更新は，リエントラント性を持ちません。

［使用例］

```
#include    <math.h>
#include    <stdio.h>
void func(void){
    float   ret;
    ret = logf(-0.1);    /*ret に 3 を返す */
}
int matherr(struct exception *e){
    char    s[30];
    switch(e->type){
        case    DOMAIN:
            sprintf(s, "%s DOMAIN error %e\n", e->name, e->arg1);
            e->retval = 3;  /* エラー・リターン値を 3 に変更 */
            break;
        default:
            sprintf(s, "%s other error %e\n", e->name, e->arg1);
    }
    return(1);
}
```

cosf

余弦を求めます。

［所属］

数学ライブラリ libm.a

［指定形式］

```
#include <math.h>
float   cosf(float x);
```

［戻り値］

x の余弦を返します。

［詳細説明］

x の余弦を求めます。角度はラジアン単位で指定します。

備考　matherr を使用して，本関数のエラー処理を変更できます。

sinf

正弦を求めます。

［所属］

数学ライブラリ libm.a

［指定形式］

```
#include <math.h>
float   sinf(float x);
```

［戻り値］

x の正弦を返します。

［詳細説明］

x の正弦を求めます。角度はラジアン単位で指定します。

備考 matherr を使用して，本関数のエラー処理を変更できます。

tanf

正接を求めます。

［所属］

数学ライブラリ libm.a

［指定形式］

```
#include <math.h>
float   tanf(float x);
```

［戻り値］

x の正接を返します。

［詳細説明］

x の正接を求めます。角度はラジアン単位で指定します。

備考　matherr を使用して，本関数のエラー処理を変更できます。

acosf

逆余弦を求めます。

［所属］

数学ライブラリ libm.a

［指定形式］

```
#include <math.h>
float   acosf(float x);
```

［戻り値］

x の逆余弦（アークコサイン）を返します。返す値はラジアン単位で，0 から π までの範囲です。

x が -1 と 1 の間にない場合，非数を返します。また，グローバル変数 errno にマクロ EDOM をセットます。

［詳細説明］

x の逆余弦（アークコサイン）を求めます。*x* は，$-1 \leqq x \leqq 1$ で指定します。

備考 matherr を使用して，本関数のエラー処理を変更できます。

asinf

逆正弦を求めます。

［所属］

数学ライブラリ libm.a

［指定形式］

```
#include <math.h>
float   asinf(float x);
```

［戻り値］

x の逆正弦（アークサイン）を返します。返す値はラジアン単位で，-π / 2 から π / 2 までの範囲です。

x が -1 と 1 の間にない場合，非数を返します。また，グローバル変数 errno にマクロ EDOM をセットします。

［詳細説明］

x の逆正弦（アークサイン）を求めます。*x* は，-1 ≦ *x* ≦ 1 で指定します。

備考 matherr を使用して，本関数のエラー処理を変更できます。

atanf

逆正接を求めます。

［所属］

数学ライブラリ libm.a

［指定形式］

#include <math.h>

float atanf(float *x*);

［戻り値］

x の逆正接（アークタンジェント）を返します。返す値はラジアン単位で，-π / 2 から π / 2 の範囲です。

［詳細説明］

x の逆正接（アークタンジェント）を求めます。

備考 matherr を使用して，本関数のエラー処理を変更できます。

［使用例］

```
#include    <math.h>
float func(float x){
    float   ret;
    ret = atanf(x); /*x の逆正接の値を ret に返す */
      :
    return(ret);
}
```

atan2f

逆正接（y / x）を求めます。

［所属］

数学ライブラリ libm.a

［指定形式］

#include <math.h>

float　atan2f(float *y*, float *x*);

［戻り値］

y / *x* の逆正接（アークタンジェント）を返します。返す値はラジアン単位で，-π から π までの範囲です。

x と *y* がともに 0.0 の場合，非数を返し，グローバル変数 errno にマクロ EDOM をセットします。解が 0 へ向かって消滅した場合，± 0 を返し，グローバル変数 errno にマクロ ERANGE をセットします。解が非正規化数の場合，グローバル変数 errno に ERANGE をセットします。

［詳細説明］

y / *x* の逆正接（アークタンジェント）を求めます。本関数は，π / 2，または -π / 2 付近（*x* が 0 に近い場合）の角度の場合も正しい結果を求めます。

備考　matherr を使用して，本関数のエラー処理を変更できます。

coshf

双曲線余弦を求めます。

［所属］

数学ライブラリ libm.a

［指定形式］

```
#include <math.h>
float   coshf(float x);
```

［戻り値］

x の双曲線余弦を返します。

オーバーフローが生じた場合，HUGE_VAL を返し，グローバル変数 errno にはマクロ ERANGE をセットします。

［詳細説明］

x の双曲線余弦を求めます。角度はラジアン単位で指定します。定義式は次のとおりです。

$$(e^{x} + e^{-x}) / 2$$

備考 matherr を使用して，本関数のエラー処理を変更できます。

sinhf

双曲線正弦を求めます。

［所属］

数学ライブラリ libm.a

［指定形式］

#include <math.h>

float sinhf(float *x*);

［戻り値］

x の双曲線正弦を返します。

オーバーフローが生じた場合，HUGE_VAL を返し，グローバル変数 errno にはマクロ ERANGE をセットします。

［詳細説明］

x の双曲線正弦を求めます。角度はラジアン単位で指定します。定義式は次のとおりです。

$$(e^{x} - e^{-x}) / 2$$

備考 matherr を使用して，本関数のエラー処理を変更できます。

tanhf

双曲線正接を求めます。

［所属］

数学ライブラリ libm.a

［指定形式］

#include <math.h>

float tanhf(float *x*);

［戻り値］

x の双曲線正接を返します。

［詳細説明］

x の双曲線正接を求めます。角度はラジアン単位で指定します。定義式は次のとおりです。

sinh (*x*) / cosh (*x*)

備考 matherr を使用して，本関数のエラー処理を変更できます。

acoshf

逆双曲線余弦を求めます。

[所属]

数学ライブラリ libm.a

[指定形式]

#include <math.h>

float acoshf(float *x*);

[戻り値]

x（*x* は 1 以上の数値）の逆双曲線余弦を返します。

x が 1 より小さい場合，非数を返します。また，グローバル変数 errno にはマクロ EDOM をセットします。

[詳細説明]

x（*x* は 1 以上の数値）の逆双曲線余弦を求めます。定義式は次のとおりです。

$$\ln\left(x + \sqrt{\left(x^2 - 1\right)}\right)$$

備考 matherr を使用して，本関数のエラー処理を変更できます。

[使用例]

```
#include    <math.h>
float func(float x) {
    float   ret;
    ret = acoshf(x);    /*x の逆双曲線余弦の値を ret に返す */
      :
    return(ret);
}
```

asinhf

逆双曲線正弦を求めます。

［所属］

数学ライブラリ libm.a

［指定形式］

```
#include <math.h>
float   asinhf(float x);
```

［戻り値］

x の逆双曲線正弦を返します。

［詳細説明］

x の逆双曲線正弦を求めます。定義式は次のとおりです。

$$\mathrm{sign}(x) * \ln(|x| + \sqrt{(1 + x^2)})$$

備考 matherr を使用して，本関数のエラー処理を変更できます。

atanhf

逆双曲線正接を求めます。

［所属］

数学ライブラリ libm.a

［指定形式］

#include <math.h>

float　atanhf(float *x*);

［戻り値］

x の逆双曲線正接を返します。

x の絶対値が 1 以上の場合，非数を返し，グローバル変数 errno にはマクロ EDOM をセットします。

［詳細説明］

x の逆双曲線正接を求めます。

備考　matherr を使用して，本関数のエラー処理を変更できます。

6. 4. 10 コピー関数

コピー関数は，ROM 化用ライブラリ libr.a に収録されています。

なお，コピー関数として，以下のものがあります。

表 6―24 コピー関数

関数 / マクロ名	概要
_rcopy	パッキング・データを 1 バイトずつ RAM へコピーする（_rcopy1 と同じ）
_rcopy1	パッキング・データを 1 バイトずつ RAM へコピーする（_rcopy と同じ）
_rcopy2	パッキング・データを 2 バイトずつ RAM へコピーする
_rcopy4	パッキング・データを 4 バイトずつ RAM へコピーする

備考 コピー関数についての詳細は，「8. 4 コピー関数」を参照してください。

6.5 ランタイム・ライブラリ

ここでは，ランタイム・ライブラリについて説明します。V850 マイクロコントローラ のアーキテクチャでは，32 ビット・データの乗算，除算や浮動小数点演算などの命令を持ちません。そこで CA850 では，ANSI 規格の言語仕様を満たすため，32 ビット・データの整数の乗算，除算，剰余算，およびすべての浮動小数点演算を，libc.a ファイルに含まれるランタイム・ライブラリを呼び出して行います。

また，V850 マイクロコントローラ用の新規のアセンブリ言語ソースを作成する際に，ランタイム・ライブラリを呼び出すこともできます。ただし，V850E では 32 ビット・データの乗算命令があるため，CA850 は 32 ビット・データの乗算，除算，剰余算についてランタイム・ライブラリを使用しません。浮動小数点演算のランタイム・ライブラリは使用します。

ランタイム・ライブラリは，CA850 がコンパイル時に自動的に使用するルーチンです。標準ライブラリとともに，libc.a ファイルに含まれています。ヘッダ・ファイルのインクルードは必要ありません。

なお，ランタイム・ライブラリをアプリケーション・プログラムで使用する場合，実行可能なオブジェクト・ファイル作成時に，ld850 で libc.a を参照する必要があります

図 6―1 ランタイム・ライブラリ使用イメージ

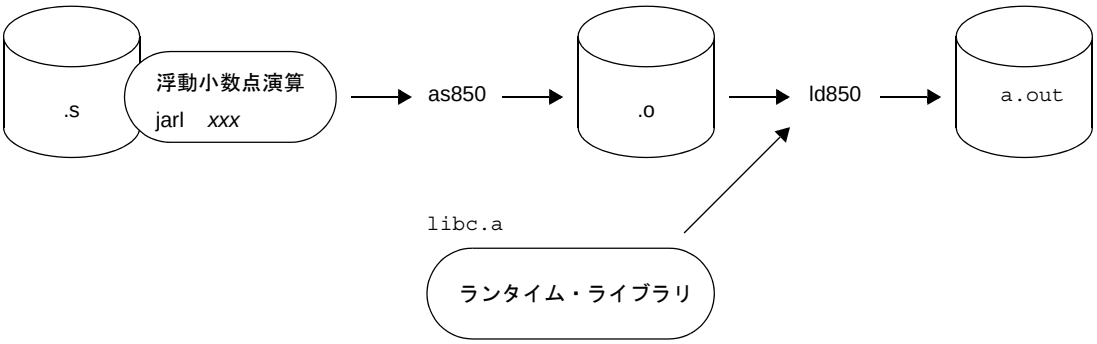


表 6―25 ランタイム・ライブラリ

分類	関数名	概要
ADDF.S	__addf.s	単精度浮動小数点の加算
CMPF.S	__cmpf.s	単精度浮動小数点の比較，およびフラグの変更
CVT.WS	__cvt.ws	整数から単精度浮動小数点への変換
DIV	__div	符号付き 32 ビット整数の除算
	__divu	符号なし 32 ビット整数の除算
DIVF.S	__divf.s	単精度浮動小数点の除算
MOD	__mod	符号付き 32 ビット整数の剰余算
	__modu	符号なし 32 ビット整数の剰余算
MUL	__mul	符号付き 32 ビット整数の乗算
	__mulu	符号なし 32 ビット整数の乗算

分類	関数名	概要
MULF.S	__mulf.s	単精度浮動小数点の乗算
SUBF.S	__subf.s	単精度浮動小数点の減算
TRNC.SW	__trnc.sw	単精度浮動小数点数から整数への変換

備考 1. ランタイム・ライブラリは，本来，コード生成部（cgen850）が使用するものであり，単体で使用することを前提としていません。したがって，アセンブリ言語ソースで使用する場合，ランタイム・ライブラリを呼び出すための前処理が必要です。

2. ランタイム・ライブラリは，C 言語ソース・プログラムでは使用できません。

3. CA850 のデフォルト処理では，16 ビット・データ以下の整数に対しては，ランタイム・ライブラリのうち，乗算の ___mul ／ ___mulu，および除算の ___div ／ ___divu は利用されず，それぞれ，mulh，divh 命令が利用されます。コンパイラで -Xe オプションを指定した場合，16 ビット・データ以下の整数に対してもランタイム・ライブラリが利用されます。
この場合，ランタイム・ライブラリを利用すると，ANSI 規格に厳密に従った乗除算処理を行いますが，mulh，divh 命令で行うよりも実行速度が遅くなります。

6.6 ライブラリ消費スタック一覧

この節では，ライブラリに含まれている各種関数のスタック消費量について説明します。

6.6.1 標準ライブラリ

以下に，標準ライブラリに含まれている各種関数のスタック消費量（単位：バイト）を示します。

（1）可変個引数関数

表 6―26 可変個引数関数

関数／マクロ名	32 レジスタ・モード		26 レジスタ・モード	22 レジスタ・モード
	マスク・レジスタ機能使用時	マスク・レジスタ機能未使用時		
va_start	0	0	0	0
va_end	0	0	0	0
va_arg	0	0	0	0

(2) 文字列関数

表 6―27 文字列関数

関数／マクロ名	32 レジスタ・モード		26 レジスタ・モード	22 レジスタ・モード
	マスク・レジスタ機能使用時	マスク・レジスタ機能未使用時		
memchr	0	0	0	0
memcmp	0	0	0	0
bcmp	0	0	0	0
memcpy	0	0	0	0
bcopy	0	0	0	0
memmove	0	0	0	0
memset	0	0	0	0

(3) メモリ管理関数

表 6―28 メモリ管理関数

関数／マクロ名	32 レジスタ・モード		26 レジスタ・モード	22 レジスタ・モード
	マスク・レジスタ機能使用時	マスク・レジスタ機能未使用時		
index	0	0	0	0
strpbrk	0	0	0	0
rindex	0	0	0	0
strrchr	0	0	0	0
strchr	0	0	0	0
strstr	0	0	0	0
strspn	0	0	0	0
strcspn	0	0	0	0
strcmp	0	0	0	0
strncmp	0	0	0	0
strcpy	0	0	0	0
strncpy	0	0	0	0
strcat	0	0	0	0
strncat	0	0	0	0
strtok	0	0	0	0
strlen	0	0	0	0
strerror	24	24	28	28

(4) 文字変換関数

表 6―29 文字変換関数

関数／マクロ名	32 レジスタ・モード		26 レジスタ・モード	22 レジスタ・モード
	マスク・レジスタ機能使用時	マスク・レジスタ機能未使用時		
toupper	0	0	0	0
_toupper	0	0	0	0
tolower	0	0	0	0
_tolower	0	0	0	0
toascii	0	0	0	0

(5) 文字分類関数

表 6―30 文字分類関数

関数／マクロ名	32 レジスタ・モード		26 レジスタ・モード	22 レジスタ・モード
	マスク・レジスタ機能使用時	マスク・レジスタ機能未使用時		
isalnum	0	0	0	0
isalpha	0	0	0	0
isascii	0	0	0	0
isupper	0	0	0	0
islower	0	0	0	0
isdigit	0	0	0	0
isxdigit	0	0	0	0
iscntrl	0	0	0	0
ispunct	0	0	0	0
isspace	0	0	0	0
isprint	0	0	0	0
isgraph	0	0	0	0

（6）標準入出力関数

表 6―31 標準入出力関数

関数／マクロ名	32 レジスタ・モード		26 レジスタ・モード	22 レジスタ・モード
	マスク・レジスタ機能使用時	マスク・レジスタ機能未使用時		
fread	28	28	28	40
getc	0	0	0	0
fgetc	0	0	0	0
fgets	0	0	0	0
fwrite	28	28	28	28
putc	0	0	0	0
fputc	0	0	0	0
fputs	0	0	0	0
getchar	0	0	0	0
gets	0	0	0	0
putchar	0	0	0	0
puts	0	0	0	0
sprintf	208	208	224	220
fprintf	200	200	216	212
vsprintf	192	192	208	204
printf	200	200	216	212
vfprintf	180	180	196	192
vprintf	184	184	200	200
sscanf	192	196	192	188
fscanf	184	188	184	180
scanf	184	188	184	180
ungetc	0	0	0	0
rewind	0	0	0	0
perror	212	212	228	224

(7) 標準ユーティリティ関数

表 6―32 標準ユーティリティ関数

関数／マクロ名	32 レジスタ・モード		26 レジスタ・モード	22 レジスタ・モード
	マスク・レジスタ機能使用時	マスク・レジスタ機能未使用時		
abs	0	0	0	0
labs	0	0	0	0
bsearch	32	32	32	40
qsort	76	76	96	100
div	32	32	36	44
ldiv	32	32	36	44
itoa	32	32	40	48
ltoa	32	32	40	48
ultoa	32	32	36	44
ecvtf	96	96	96	108
fcvtf	96	96	96	108
gcvtf	164	156	172	172
atoi	64	64	76	72
atol	64	64	76	72
strtol	64	64	80	76
strtoul	64	64	80	76
atoff	104	112	100	100
strtodf	104	112	100	100
calloc	24	24	24	28
malloc	4	4	4	4
realloc	12	12	16	16
free	8	8	8	12
rand	16	16	16	16
srand	0	0	0	0

(8) 非局所分岐関数

表 6―33 非局所分岐関数

関数／マクロ名	32 レジスタ・モード		26 レジスタ・モード	22 レジスタ・モード
	マスク・レジスタ機能使用時	マスク・レジスタ機能未使用時		
longjmp	0	0	0	0
setjmp	0	0	0	0

(9) ランタイム・ライブラリ

表 6―34 ランタイム・ライブラリ

関数／マクロ名	32 レジスタ・モード		26 レジスタ・モード	22 レジスタ・モード
	マスク・レジスタ機能使用時	マスク・レジスタ機能未使用時		
__mul	12	12	12	12
__mulu	12	12	12	12
__div	20	20	20	20
__divu	16	16	16	16
__mod	20	20	20	20
__modu	16	16	16	16
__addf.s	72	72	60	52
__subf.s	72	72	60	52
__mulf.s	72	72	60	52
__divf.s	72	72	60	52
__cvt.ws	12	12	12	12
__trnc.sw	0	0	0	0
__cmpf.s	72	72	60	52

(10) 関数のプロローグ／エピローグ・ランタイム・ライブラリ

表 6―35 関数のプロローグ／エピローグ・ランタイム・ライブラリ

関数／マクロ名	32 レジスタ・モード		26 レジスタ・モード	22 レジスタ・モード
	マスク・レジスタ機能使用時	マスク・レジスタ機能未使用時		
___push2000	0	0	0	0

関数／マクロ名	32 レジスタ・モード		26 レジスタ・モード	22 レジスタ・モード
	マスク・レジスタ機能使用時	マスク・レジスタ機能未使用時		
___push2001	0	0	0	0
___push2002	0	0	0	0
___push2003	0	0	0	0
___push2004	0	0	0	0
___push2040	0	0	0	0
___push2100	0	0	0	0
___push2101	0	0	0	0
___push2102	0	0	0	0
___push2103	0	0	0	0
___push2104	0	0	0	0
___push2140	0	0	0	0
___push2200	0	0	0	0
___push2201	0	0	0	0
___push2202	0	0	0	0
___push2203	0	0	0	0
___push2204	0	0	0	0
___push2240	0	0	0	0
___push2300	0	0	0	0
___push2301	0	0	0	0
___push2302	0	0	0	0
___push2303	0	0	0	0
___push2304	0	0	0	0
___push2340	0	0	0	0
___push2400	0	0	0	0
___push2401	0	0	0	0
___push2402	0	0	0	0
___push2403	0	0	0	0
___push2404	0	0	0	0
___push2440	0	0	0	0
___push2500	0	0	0	0
___push2501	0	0	0	0
___push2502	0	0	0	0
___push2503	0	0	0	0
____push2504	0	0	0	0
___push2540	0	0	0	0

関数／マクロ名	32 レジスタ・モード		26 レジスタ・モード	22 レジスタ・モード
	マスク・レジスタ機能使用時	マスク・レジスタ機能未使用時		
___push2600	0	0	0	0
___push2601	0	0	0	0
___push2602	0	0	0	0
___push2603	0	0	0	0
___push2604	0	0	0	0
___push2640	0	0	0	0
___push2700	0	0	0	0
___push2701	0	0	0	0
___push2702	0	0	0	0
___push2703	0	0	0	0
___push2704	0	0	0	0
___push2740	0	0	0	0
___push2800	0	0	0	0
___push2801	0	0	0	0
___push2802	0	0	0	0
___push2803	0	0	0	0
___push2804	0	0	0	0
___push2840	0	0	0	0
___push2900	0	0	0	0
___push2901	0	0	0	0
___push2902	0	0	0	0
___push2903	0	0	0	0
___push2904	0	0	0	0
___push2940	0	0	0	0
___pushlp00	0	0	0	0
___pushlp01	0	0	0	0
___pushlp02	0	0	0	0
___pushlp03	0	0	0	0
___pushlp04	0	0	0	0
___pushlp40	0	0	0	0
___Epush250	0	0	0	0
___Epush251	0	0	0	0
___Epush252	0	0	0	0
___Epush253	0	0	0	0
___Epush254	0	0	0	0

関数／マクロ名	32 レジスタ・モード		26 レジスタ・モード	22 レジスタ・モード
	マスク・レジスタ機能使用時	マスク・レジスタ機能未使用時		
___Epush260	0	0	0	0
___Epush261	0	0	0	0
___Epush262	0	0	0	0
___Epush263	0	0	0	0
___Epush264	0	0	0	0
___Epush270	0	0	0	0
___Epush271	0	0	0	0
___Epush272	0	0	0	0
___Epush273	0	0	0	0
___Epush274	0	0	0	0
___Epush280	0	0	0	0
___Epush281	0	0	0	0
___Epush282	0	0	0	0
___Epush283	0	0	0	0
___Epush284	0	0	0	0
___Epush290	0	0	0	0
___Epush291	0	0	0	0
___Epush292	0	0	0	0
___Epush293	0	0	0	0
___Epush294	0	0	0	0
___Epushlp0	0	0	0	0
___Epushlp1	0	0	0	0
___Epushlp2	0	0	0	0
___Epushlp3	0	0	0	0
___Epushlp4	0	0	0	0
___pop2000	0	0	0	0
___pop2001	0	0	0	0
___pop2002	0	0	0	0
___pop2003	0	0	0	0
___pop2004	0	0	0	0
___pop2040	0	0	0	0
___pop2100	0	0	0	0
___pop2101	0	0	0	0
___pop2102	0	0	0	0
___pop2103	0	0	0	0

関数／マクロ名	32 レジスタ・モード		26 レジスタ・モード	22 レジスタ・モード
	マスク・レジスタ機能使用時	マスク・レジスタ機能未使用時		
___pop2104	0	0	0	0
___pop2140	0	0	0	0
___pop2200	0	0	0	0
___pop2201	0	0	0	0
___pop2202	0	0	0	0
___pop2203	0	0	0	0
___pop2204	0	0	0	0
___pop2240	0	0	0	0
___pop2300	0	0	0	0
___pop2301	0	0	0	0
___pop2302	0	0	0	0
___pop2303	0	0	0	0
___pop2304	0	0	0	0
___pop2340	0	0	0	0
___pop2400	0	0	0	0
___pop2401	0	0	0	0
___pop2402	0	0	0	0
___pop2403	0	0	0	0
___pop2404	0	0	0	0
___pop2440	0	0	0	0
___pop2500	0	0	0	0
___pop2501	0	0	0	0
___pop2502	0	0	0	0
___pop2503	0	0	0	0
___pop2504	0	0	0	0
___pop2540	0	0	0	0
___pop2600	0	0	0	0
___pop2601	0	0	0	0
___pop2602	0	0	0	0
___pop2603	0	0	0	0
___pop2604	0	0	0	0
___pop2640	0	0	0	0
___pop2700	0	0	0	0
___pop2701	0	0	0	0
___pop2702	0	0	0	0

関数／マクロ名	32 レジスタ・モード		26 レジスタ・モード	22 レジスタ・モード
	マスク・レジスタ機能使用時	マスク・レジスタ機能未使用時		
___pop2703	0	0	0	0
___pop2704	0	0	0	0
___pop2740	0	0	0	0
___pop2800	0	0	0	0
___pop2801	0	0	0	0
___pop2802	0	0	0	0
___pop2803	0	0	0	0
___pop2804	0	0	0	0
___pop2840	0	0	0	0
___pop2900	0	0	0	0
___pop2901	0	0	0	0
___pop2902	0	0	0	0
___pop2903	0	0	0	0
___pop2904	0	0	0	0
___pop2940	0	0	0	0
___poplp00	0	0	0	0
___poplp01	0	0	0	0
___poplp02	0	0	0	0
___poplp03	0	0	0	0
___poplp04	0	0	0	0
___poplp40	0	0	0	0
___Epop250	0	0	0	0
___Epop251	0	0	0	0
___Epop252	0	0	0	0
___Epop253	0	0	0	0
___Epop254	0	0	0	0
___Epop260	0	0	0	0
___Epop261	0	0	0	0
___Epop262	0	0	0	0
___Epop263	0	0	0	o
___Epop264	0	0	0	0
___Epop270	0	0	0	0
___Epop271	0	0	0	0
___Epop272	0	0	0	0
___Epop273	0	0	0	0

関数／マクロ名	32 レジスタ・モード		26 レジスタ・モード	22 レジスタ・モード
	マスク・レジスタ機能使用時	マスク・レジスタ機能未使用時		
___Epop274	0	0	0	0
___Epop280	0	0	0	0
___Epop281	0	0	0	0
___Epop282	0	0	0	0
___Epop283	0	0	0	0
___Epop284	0	0	0	0
___Epop290	0	0	0	0
___Epop291	0	0	0	0
___Epop292	0	0	0	0
___Epop293	0	0	0	0
___Epop294	0	0	0	0
___Epoplp0	0	0	0	0
___Epoplp1	0	0	0	0
___Epoplp2	0	0	0	0
___Epoplp3	0	0	0	0
___Epoplp4	0	0	0	0

6. 6. 2 数学ライブラリ

以下に，数学ライブラリに含まれている各種関数のスタック消費量（単位：バイト）を示します。

（1）数学関数

表 6―36 数学関数

関数／マクロ名	32 レジスタ・モード		26 レジスタ・モード	22 レジスタ・モード
	マスク・レジスタ機能使用時	マスク・レジスタ機能未使用時		
j0f	32	32	44	52
j1f	32	32	44	52
jnf	52	52	64	72
y0f	44	44	56	64
y1f	44	44	56	64
ynf	64	64	76	84
erff	32	32	44	52
erfcf	32	32	44	52
expf	28	28	28	28
logf	28	28	28	32
log2f	28	28	28	32
log10f	28	28	28	32
powf	28	28	32	40
sqrtf	28	28	28	28
cbrtf	28	28	28	32
ceilf	0	0	0	0
fabsf	0	0	0	0
floorf	0	0	0	0
fmodf	28	28	28	28
frexpf	28	28	28	28
ldexpf	28	28	28	28
modff	0	0	0	0
gammaf	28	28	32	40
hypotf	28	28	28	36
matherr	0	0	0	0
cosf	28	28	28	28
sinf	28	28	28	28
tanf	28	28	32	40
acosf	28	28	28	36

関数／マクロ名	32 レジスタ・モード		26 レジスタ・モード	22 レジスタ・モード
	マスク・レジスタ機能使用時	マスク・レジスタ機能未使用時		
asinf	28	28	28	36
atanf	28	28	28	36
atan2f	28	28	32	40
coshf	28	28	28	28
sinhf	28	28	28	28
tanhf	28	28	28	36
acoshf	28	28	28	32
asinhf	28	28	28	32
atanhf	28	28	28	32

6. 6. 3 ROM 化用ライブラリ

以下に，ROM 化用ライブラリに含まれている各種関数のスタック消費量（単位：バイト）を示します。

(1) コピー関数

表 6―37 コピー関数

関数／マクロ名	32 レジスタ・モード		26 レジスタ・モード	22 レジスタ・モード
	マスク・レジスタ機能使用時	マスク・レジスタ機能未使用時		
_rcopy	0	0	8	20
_rcopy1	0	0	8	20
_rcopy2	0	0	8	16
_rcopy4	0	0	8	16

第 7 章　スタートアップ

この章では，スタート・アップ・ルーチンについて説明します。

7.1　機能概要

C 言語によるプログラムを実行させるには，システムへ組み込むための ROM 化処理，ユーザ・プログラム（main 関数）の起動などを行うプログラムが必要となります。このプログラムのことをスタート・アップ・ルーチンと呼びます。

ユーザが作成したプログラムを実行させるには，そのプログラムに応じたスタート・アップ・ルーチンを作成しなければなりません。CubeSuite+ では，プログラム実行前に必要な処理を含むスタート・アップ・ルーチンのオブジェクト・ファイルと，ユーザがシステムに合わせて変更できるようにスタート・アップ・ルーチンのソース・ファイルを提供しています。

7.2　ファイルの構成

CubeSuite+ が提供しているスタート・アップ・ルーチンは，以下のとおりです。

表 7―1　スタート・アップ・ルーチンのサンプル

格納場所	ファイル名	内容
Install Folder\lib850\r22	crtN.s	22 レジスタ・モード用 V850 コア用スタート・アップ・ルーチンのサンプル
	crtE.s	22 レジスタ・モード用 V850Ex コア用スタート・アップ・ルーチンのサンプル
Install Folder\lib850\r26	crtN.s	26 レジスタ・モード用 V850 コア用スタート・アップ・ルーチンのサンプル
	crtE.s	26 レジスタ・モード用 V850Ex コア用スタート・アップ・ルーチンのサンプル
Install Folder\lib850\r32	crtN.s	32 レジスタ・モード用 V850 コア用スタート・アップ・ルーチンのサンプル
	crtE.s	32 レジスタ・モード用 V850Ex コア用スタート・アップ・ルーチンのサンプル

また，スタート・アップ・ルーチンをプロジェクトに追加しなかった場合，スタート・アップ・ルーチン（オブジェクト）を自動的にリンクしません。

スタートアップ・ルーチンを新たに作成する場合には，上記のサンプルをコピーして，プロジェクトに追加後，編集してください。

それぞれ，サンプルのスタート・アップ・ルーチン“crtN.s”，および“crtE.s”をコンパイル（アセンブル）したファイルです。

また，これらのオブジェクトは，アセンブラ・オプション“-cn”，“-cnv850e”，および“-cnv850e2”を指定してアセンブルしており，V850 マイクロコントローラで共通に使用できるようにしたオブジェクトになっています。

7.3　スタートアップ・ルーチン

スタート・アップ・ルーチンとは，V850 をリセットしたあと，main 関数を実行する前に，実行するルーチンを言います。基本的にはシステムをリセットしたあとの初期化を行います。具体的には，次のことを行います。

- リセットが入ったときの RESET ハンドラの設定
- スタート・アップ・ルーチンのレジスタ・モード設定
- スタック領域の確保とスタック・ポインタの設定
- main 関数の引数領域の確保
- テキスト・ポインタ（tp）の設定
- グローバル・ポインタ（gp）の設定
- エレメント・ポインタ（ep）の設定
- マスク・レジスタ（r20，r21）へマスク値を設定
- main 関数実行前に行う必要のある周辺 I/O レジスタの初期化
- main 関数実行前に行う必要のあるユーザ・ターゲットの初期化
- sbss 領域のゼロクリア
- bss 領域のゼロクリア
- sebss 領域のゼロクリア
- tibss.byte 領域のゼロクリア
- tibss.word 領域のゼロクリア
- sibss 領域のゼロクリア
- 関数のプロローグ・エピローグ・ランタイム・ライブラリ用の CTBP 値の設定【V850E】
- プログラマブル周辺 I/O レジスタ値の設定【V850E】
- r6 と r7 を main 関数の引数に設定
- main 関数へ分岐する（リアルタイム OS を使用していない場合）
- リアルタイム OS の初期化ルーチンへ分岐する（リアルタイム OS を使用している場合）

もちろん，システムによっては必要のない処理もありますので，それらに関しては省略できます。

また，これ以外にもユーザで行っておきたい処理があった場合は記述しておきます。

なお，これらの処理は，基本的にアセンブラ命令で記述する必要があります。

7.3.1　リセットが入ったときの RESET ハンドラの設定

リセット（リセット割り込み）が入ったときの処理を記述します。V850 では，リセットが入ると 0x0 番地のハンドラ・アドレスに分岐します。そこで，0x0 番地にスタート・アップ・ルーチンの先頭へ分岐する命令を配置します。リセット割り込みは C 言語上で #pragma interrupt 指定による記述ができないので，アセンブラ命令で記述します。記述は次のようになります。

```
    .section    "RESET", text
    jr      __start
__start:
```

ハンドラ・アドレスへの配置には，.section 疑似命令を使用します。上記のように記述することによって RESET のハンドラ・アドレスに“jr __start”という命令が配置されます。

また，jr 命令で届かない場所，つまり，0x0 番地から± 2M バイト内に“__start”がなかった場合は，次のように jmp 命令を使用します。

```
    .section    "RESET", text
    mov     #__start, lp
    jmp     [lp]
__start:
```

この場合，レジスタを 1 つ使用します。上記の例では lp（r31）レジスタを使用していますが，この時点で破壊してもよい汎用レジスタがあれば使用可能です。リセット時点では，関数からの戻りアドレスが入る lp（r31）レジスタを使用することはないので，lp（r31）レジスタの使用が安全です。

なお，これらの .section 疑似命令を記述は，特にスタート・アップ・ルーチン内ではなくても問題ありません。

また，例ではスタート・アップ・ルーチンのシンボルを“__start”としていますが，これも別の名前で問題ありません。

7.3.2　スタート・アップ・ルーチンのレジスタ・モード設定

アセンブラ命令で記述するスタート・アップ・ルーチンに，レジスタ・モードの設定する記述をします。

ただし，この設定をする必要があるのは，システム全体で 22 レジスタ・モード，26 レジスタ・モードを指定している場合です。32 レジスタ・モードを指定している場合は記述する必要がありません。

【22 レジスタ・モード時】

```
.option reg_mode    5 5
```

【26 レジスタ・モード時】

```
.option reg_mode    7 7
```

この設定をしていなかった場合，リンク時に次の警告メッセージが出力されます。

```
W4608: input files have different register modes. use -rc option for more information.
```

7. 3. 3　スタック領域の確保とスタック・ポインタの設定

システムで使用するスタック領域を確保し，その領域の先頭にスタック・ポインタ（SP=r3）を設定します。ただし，リアルタイム OS を使用している場合，ここで指定するスタックは，リアルタイム OS の初期化ルーチンに分岐するまでに使用するスタックとなります。

したがって，ほとんど使用しない，またはまったく使用しないことが多いので，ここで多く確保してしまうとRAM 領域が無駄になります。リアルタイム OS の初期化ルーチンに分岐するまでにスタックを使用しているかどうかを確認してください。特に割り込みには注意が必要ですが，スタート・アップ・ルーチン内は割り込み禁止で実行することが通例です。

スタック領域の確保の方法は次のようになります。

```
.set    STACKSIZE, 0x200
.bss
.lcomm  __stack, STACKSIZE, 4
mov     #__stack + STACKSIZE, sp
```

上記は，確保するスタック・サイズは 0x200 バイトで，.bss 領域に確保する例です。スタックの内容は，初期値を持たないので“bss 属性の領域”に配置します。もちろん sbss 領域に配置することも可能ですが，sbss 領域には gp 相対 1 命令でアクセス領域のために，配置できるサイズに限界があります。他の変数等を配置した方がよいこともあるので，スタック・サイズが大きくなる場合は，bss 領域に配置することを推奨しています。

確保するスタック・サイズを変更するときは，.set 命令に書かれている数値を変更します。また，CA850 は，スタック・ポインタ（sp）相対でメモリを参照する場合，sp が 4 バイト境界に位置していることを前提としたコードを出力しています。そのため，スタック・ポインタは必ず“4 バイト境界”に配置するようにしてください。必要ならば疑似命令“.align 4”を使用してください。

スタックはシステムの動作に大きく関わってきます。スタックが不足すると，確保した領域を越えて破壊するため，システムの暴走につながります。確保すべきスタック・サイズは，CA850 にパッケージされている stk850 などを用いて，関数で使用するスタック・サイズを見積もり，十分なサイズを確保してください。

7. 3. 4　main 関数の引数領域の確保

ANSI C 仕様では，main 関数の形式は，仮引数を持たない“int main (void) { ... }”として定義されるか，2 つの仮引数をもつ関数“int main (int argc, char *argv[ ]) { ... }”として定義されます。

ここで 2 つの仮引数を持つ関数の場合，argc は非負の値であり，仮引数の総計を示します。argv は引数文字列へのポインタの配列を示します。argv[argc] は NULL（空ポインタ）で，argc が 1 以上ならば argv[0] ～ argv[argc - 1] は文字列へのポインタになります。

この argc と argv の領域をスタート・アップ・ルーチン内で確保します。確保の方法は次のとおりです。

```
    .data
    .size   __argc, 4
    .align  4
__argc:
    .word   0
    .size   __argv, 4
__argv:
    .word   #.L16
.L16:
    .byte   0
    .byte   0
    .byte   0
    .byte   0
```

この領域は初期値定義しておくため，“data 属性領域”に配置します。

“int main (void) { ... }”の形で main 関数を定義する場合は，上記の領域は不要です。

削除することにより，上記の分 RAM 領域を削減することができます。

なお，実際に main 関数の引数（r6 と r7）へ設定する処理は main 関数の直前で行います。r6 と r7 をスタート・アップ・ルーチン内で使用しないのであれば，上記のプログラム直後に行っても問題ありません。設定する処理は「7. 3. 19　r6 と r7 を main 関数の引数に設定」を参照してください。

7. 3. 5　テキスト・ポインタ（tp）の設定

アプリケーションのテキスト領域であるプログラム・コードを参照する際に，配置される位置に依存しない参照（PIC：Position Independent Code）を実現するために用意されているポインタが“テキスト・ポインタ（tp）”です。たとえば，プログラム実行中に，コード内のある箇所を参照する必要がある場合，CA850 は tp 相対でアクセスするコードを出力します。

したがって，tp が正しく設定されていることを前提としたコードを出力しているので，スタート・アップ・ルーチン内で tp を正しく設定する必要があります。

テキスト・ポインタの値は，リンク時に決定され，リンク・ディレクティブ・ファイル内に書かれる“シンボル・ディレクティブ”に定義されたシンボルに入っています。たとえば，テキスト・ポインタのシンボル・ディレクティブが次のように記述されていたとします。

```
__tp_TEXT @ %TP_SYMBOL {TEXT};
```

このとき，テキスト・ポインタの値は“TEXT セグメント”の先頭になり，その値は“__tp_TEXT”に入ります。

スタート・アップ・ルーチン内で tp をセットするには，次のように記述してください。

```
.extern __tp_TEXT, 4
mov     #__tp_TEXT, tp
```

7. 3. 6　グローバル・ポインタ（gp）の設定

アプリケーション内で定義した外部変数／データはメモリ上に配置されます。そのメモリに配置されている変数／データを参照する際，配置位置に依存することのない参照（PID：Position Independent Data）を実現するために用意されているポインタが“グローバル・ポインタ（gp）”です。gp 相対でアクセスするセクションが存在する場合，CA850 は gp 相対でアクセスするコードを出力します。

したがって，gp が正しく設定されていることを前提としたコードを出力しているので，スタート・アップ・ルーチン内で gp を正しく設定する必要があります。

グローバル・ポインタの値は，リンク時に決定され，リンク・ディレクティブ・ファイル内に書かれる“シンボル・ディレクティブ”に定義されたシンボルに入っています。たとえば，グローバル・ポインタのシンボル・ディレクティブが次のように記述されていたとします。

```
__gp_DATA @ %GP_SYMBOL {DATA};
```

また，gp シンボル値は，上記のように“DATA セグメントなどの「データ用セグメント」の先頭を gp シンボル値とする方法”のほかに，“テキスト・シンボルからのオフセットを gp シンボル値とする”方法もあります。

この方法の場合，gp シンボルを「tp に，tp からのオフセット値を加える」ことによって決定できます。つまり，配置に依存しないコードの生成が可能になります。たとえば，“プログラム・コード”と“そのコードが使用するデータ”を同時 RAM 領域にコピーしてから実行させたい場合，コードの先頭（コピー先の先頭アドレス）さえ分かれば，gp の値もすぐ導き出せるというメリットがあります。この場合のシンボル・ディレクティブ記述は次のようになります。

```
__tp_TEXT @ %TP_SYMBOL {TEXT};
__gp_DATA @ %GP_SYMBOL &__tp_TEXT {DATA};
```

グローバル・ポインタの値は，“__tp_TEXT に __gp_DATA の値を加えた値”となり，加える値（オフセット値）が“__gp_DATA”に入ります。したがって，スタート・アップ・ルーチン内で gp をセットするには，次のように記述します。

```
.extern __tp_TEXT, 4
.extern __gp_DATA, 4
mov     #__tp_TEXT, tp
mov     #__gp_DATA, gp
add     tp, gp
```

これにより，正しいグローバル・ポインタの値が gp に設定されます。

7.3.7　エレメント・ポインタ（ep）の設定

アプリケーション内で定義した外部変数／データのうち，次に割り当てられているものは，エレメント・ポインタ（ep）からの相対でアクセスされます。

- sedata ／ sebss セクション
- sidata ／ sibss セクション
- tidata.byte ／ tibss.byte セクション
- tidata.word ／ tibss.word セクション

これらのセクションが存在する場合，CA850 は ep 相対でアクセスするコードを出力します。

したがって，ep が正しく設定されていることを前提としたコードを出力しているので，スタート・アップ・ルーチン内で ep を正しく設定する必要があります。

エレメント・ポインタの値は，リンク時に決定され，リンク・ディレクティブ・ファイル内に書かれる“シンボル・ディレクティブ”に定義されたシンボルに入っています。たとえば，エレメント・ポインタのシンボル・ディレクティブが次のように記述されていたとします。

```
__ep_DATA @ %EP_SYMBOL;
```

エレメント・ポインタの値は，デフォルトで“SIDATA セグメントの先頭”になり，その値は“__ep_DATA”に入ります。

したがって，スタート・アップ・ルーチン内で ep をセットするには，次のように記述します。

```
.extern __ep_DATA, 4
mov     #__ep_DATA, ep
```

__ep_DATA の絶対アドレス参照を行い，その値を ep に設定します。

7.3.8　マスク・レジスタ（r20，r21）へマスク値を設定

マスク・レジスタを使用する場合，スタート・アップ・ルーチン内で設定します。マスク・レジスタは“r20”と“r21”で，次の値を設定します。

- r20 は 8 ビットのマスク値“0xff”
- r21 は 16 ビットのマスク値“0xffff”

設定方法は次のようになります。

```
.option nowarning
mov     0xff, r20
mov     0xffff, r21
.option warning
```

“.option nowarning”と“.option warning”は，アセンブル時の警告メッセージの出力を抑止するための疑似命令です。アセンブラ・オプションで“-m オプション”（マスク・オプションの使用）を設定していると，r20 と r21 にマスク値が設定されたコードを出力します。そのため，ユーザが故意に r20 と r21 に対して値を代入しようとした場合，次の警告メッセージが出力されます。

```
W3013: mask register r20 or r21 used as destination register.
```

なお，マスク・レジスタについての詳細は「3. 1. 7 マスク・レジスタ」を参照してください。

7. 3. 9 main 関数実行前に行う必要のある周辺 I/O レジスタの初期化

スタート・アップ・ルーチン内で外部 RAM の初期化などを行う場合，まず周辺 I/O に対して外部メモリ設定等を行わないと，メモリ領域のアクセスができず，初期化ができません。その他，スタート・アップ・ルーチンを実行するうえで，設定しなくてはならない周辺 I/O レジスタの初期化を行います。

なお，レジスタ設定は，アセンブラ命令でそのまま記述してもよいですし，いったんスタート・アップ・ルーチンから C 言語関数へ分岐し，その C 言語関数内で行ってもよいです。C 言語で行うと，周辺 I/O への読み出しや代入を分かりやすく記述できます。たとえば C 言語関数“void reset(void)”を作成して，スタート・アップ・ルーチンから呼び出すときは，スタート・アップ・ルーチン内に次の命令を記述します。

```
jarl    _reset, lp
```

アセンブラ命令の記述と C 言語記述の違いの例を示します。たとえば，P0（ポート 0）に“1”を代入する命令を，アセンブリ言語ソース（r10 を使用）と C 言語ソースで記述すると次のようになります。

【アセンブリ言語ソース 】

```
mov     1, r10
st.b    r10, P0
```

【C 言語ソース】

```
#pragma ioreg
P0 = 1;
```

外部メモリ設定等は，デバイスによって異なりますので，使用する各デバイスのユーザーズ・マニュアルを参照してください。

また，クロック発生機能を使用し，V850 内蔵の各ユニットに供給される “内部システム・クロック”を発生させる必要がありますが，このとき，PLL （Phase locked loop）シンセサイザによって，クロックを逓倍して 使用します。したがって，使用する周波数を正しく設定しなければ，想定している動作速度に誤差を生じます。

PLL のデフォルト値は，たいてい逓倍値が小さく，動作周波数が低くなっています。スタート・アップ・ルーチンにおいても例外ではなく，「7. 3. 11 sbss 領域のゼロクリア」以降で説明する “メモリ領域のクリア”を，動作周波数が低いまま実行すると，実行完了までにたいへん時間がかかってしまいます。したがって，PLL の設定に 関しては，スタート・アップ・ルーチンの初期の方で行うことを推奨します。

```
--V850ES/SG2 において 5MHz を 4 逓倍（20MHz）にする設定
mov     0x80, r10
st.b    r10, PRCMD
st.b    r10, PCC    --fcpu = fxx
nop
nop
nop
nop
nop
set1    0, PLLCTL   --PLLON = 1
```

他“システム・ウエイト・コントロール・レジスタ（VSWC)”やコマンド・レジスタ（PRCMD)，必要であれば“ウォッチ・ドック・タイマ（WDT)”などの設定も必要になりますので，使用する各デバイスのユーザーズ・マニュアルを参考に正しく設定してください。

7. 3. 10　main 関数実行前に行う必要のあるユーザ・ターゲットの初期化

スタート・アップ・ルーチン内でユーザ・ターゲットに初期化が必要なものがある場合は，その初期化処理を記述しておきます。

処理はアセンブリ言語ソースで記述してもよいですし，いったんスタート・アップ・ルーチンから C 言語関数へ分岐し，その C 言語関数内で行ってもよいです。

7. 3. 11　sbss 領域のゼロクリア

初期値を持たない領域である“bss 属性”領域の 1 つである“sbss 領域”の初期化を行います。

V850 リセット後のメモリ内容は不定なので，sbss 領域をゼロクリアすることを推奨します。

なお，sbss セクションを作成していない場合や，ゼロクリアの必要がない場合は，この処理を行う必要はありません。

sbss 領域のクリアを行うときは，CA850 で予約されているシンボル“__ssbss”と“__esbss”を使用します。それぞれのシンボルの意味は次のとおりです。

表 7―2　sbss 領域のシンボル

シンボル名	意味
__ssbss	sbss 領域の先頭シンボル
__esbss	sbss 領域の最後尾シンボル

これらのシンボルの値（アドレス）は，リンク時に決定されます。このシンボルを使用して，sbss 領域をゼロクリアするプログラムは次のとおりです。

```
    .extern __ssbss, 4
    .extern __esbss, 4
    mov     #__ssbss, r13
    mov     #__esbss, r12
    cmp     r12, r13
    jnl     .L11
.L12:
    st.w    r0, [r13]
    add     4, r13
    cmp     r12, r13
    jl      .L12
.L11:
```

sbss 領域を 4 バイトずつゼロクリアしていきます。

7. 3. 12　bss 領域のゼロクリア

初期値を持たない領域である“bss 属性”領域の 1 つである“bss 領域”の初期化を行います。

V850 リセット後のメモリ内容は不定なので，bss 領域をゼロクリアすることを推奨します。

なお，bss セクションを作成していない場合や，ゼロクリアの必要がない場合は，この処理を行う必要はありません。

bss 領域のクリアを行うときは，CA850 で予約されているシンボル“__sbss”と“__ebss”を使用します。それぞれのシンボルの意味は次のとおりです。

表 7―3　bss 領域のシンボル

シンボル名	意味
__sbss	bss 領域の先頭シンボル
__ebss	bss 領域の最後尾シンボル

これらのシンボルの値（アドレス）は，リンク時に決定されます。このシンボルを使用して，bss 領域をゼロクリアするプログラムは次のとおりです（bss 領域を 4 バイトずつゼロクリアしていきます）。

```
    .extern __sbss, 4
    .extern __ebss, 4
    mov     #__sbss, r13
    mov     #__ebss, r12
    cmp     r12, r13
    jnl     .L14
.L15:
    st.w    r0, [r13]
    add     4, r13
    cmp     r12, r13
    jl      .L15
.L14:
```

7. 3. 13　sebss 領域のゼロクリア

初期値を持たない領域である“bss 属性”領域の 1 つである“sebss 領域”の初期化を行います。

V850 リセット後のメモリ内容は不定なので，sebss 領域をゼロクリアすることを推奨します。

なお，sebss セクションを作成していない場合や，ゼロクリアの必要がない場合は，この処理を行う必要はありません。

sebss 領域のクリアを行うときは，CA850 で予約されているシンボル“__ssebss”と“__esebss”を使用します。それぞれのシンボルの意味は次のとおりです。

表 7―4　sebss 領域のシンボル

シンボル名	意味
__ssebss	sebss 領域の先頭シンボル
__esebss	sebss 領域の最後尾シンボル

これらのシンボルの値（アドレス）は，リンク時に決定されます。このシンボルを使用して，sebss 領域をゼロクリアするプログラムは次のとおりです（sebss 領域を 4 バイトずつゼロクリアしていきます）。

```
    .extern __ssebss, 4
    .extern __esebss, 4
    mov     #__ssebss, r13
    mov     #__esebss, r12
    cmp     r12, r13
    jnl     .L17
.L18:
    st.w    r0, [r13]
    add     4, r13
    cmp     r12, r13
    jl      .L18
.L17:
```

7. 3. 14　tibss.byte 領域のゼロクリア

初期値を持たない領域である“bss 属性”領域の 1 つである“tibss.byte 領域”の初期化を行います。

V850 リセット後のメモリ内容は不定なので，tibss.byte 領域をゼロクリアすることを推奨します。

なお，tibss.byte セクションを作成していない場合や，ゼロクリアの必要がない場合は，この処理を行う必要はありません。

tibss.byte 領域のクリアを行うときは，CA850 で予約されているシンボル“__stibss.byte”と“__etibss.byte”を使用します。それぞれのシンボルの意味は次のとおりです。

表 7―5　tibss.byte 領域のシンボル

シンボル名	意味
__stibss.byte	tibss.byte 領域の先頭シンボル
__etibss.byte	tibss.byte 領域の最後尾シンボル

これらのシンボルの値（アドレス）は，リンク時に決定されます。このシンボルを使用して，tibss.byte 領域をゼロクリアするプログラムは次のとおりです（tibss.byte 領域を 4 バイトずつゼロクリアしていきます）。

```
    .extern __stibss.byte, 4
    .extern __etibss.byte, 4
    mov     #__stibss.byte, r13
    mov     #__etibss.byte, r12
    cmp     r12, r13
    jnl     .L20
.L21:
    st.w    r0, [r13]
    add     4, r13
    cmp     r12, r13
    jl      .L21
.L20:
```

7. 3. 15　tibss.word 領域のゼロクリア

初期値を持たない領域である“bss 属性”領域の 1 つである“tibss.word 領域”の初期化を行います。

V850 リセット後のメモリ内容は不定なので，tibss.word 領域をゼロクリアすることを推奨します。

なお，tibss.word セクションを作成していない場合や，ゼロクリアの必要がない場合は，この処理を行う必要はありません。

tibss.word 領域のクリアを行うときは，CA850 で予約されているシンボル“__stibss.word”と“__etibss.word”を使用します。それぞれのシンボルの意味は次のとおりです。

表 7―6　tibss.word 領域のシンボル

シンボル名	意味
__stibss.word	tibss.word 領域の先頭シンボル

シンボル名	意味
__etibss.word	tibss.word 領域の最後尾シンボル

これらのシンボルの値（アドレス）は，リンク時に決定されます。このシンボルを使用して，tibss.word 領域をゼロクリアするプログラムは次のとおりです（tibss.word 領域を 4 バイトずつゼロクリアしていきます）。

```
    .extern __stibss.word, 4
    .extern __etibss.word, 4
    mov     #__stibss.word, r13
    mov     #__etibss.word, r12
    cmp     r12, r13
    jnl     .L23
.L24:
    st.w    r0, [r13]
    add     4, r13
    cmp     r12, r13
    jl      .L24
.L23:
```

7. 3. 16　sibss 領域のゼロクリア

初期値を持たない領域である“bss 属性”領域の 1 つである“sibss 領域”の初期化を行います。

V850 リセット後のメモリ内容は不定なので，sibss 領域をゼロクリアすることを推奨します。

なお，sibss セクションを作成していない場合や，ゼロクリアの必要がない場合は，この処理を行う必要はありません。

sibss 領域のクリアを行うときは，CA850 で予約されているシンボル“__ssibss”と“__esibss”を使用します。それぞれのシンボルの意味は次のとおりです。

表 7―7　sibss 領域のシンボル

シンボル名	意味
__ssibss	sibss 領域の先頭シンボル
__esibss	sibss 領域の最後尾シンボル

これらのシンボルの値（アドレス）は，リンク時に決定されます。このシンボルを使用して，sibss 領域をゼロクリアするプログラムは次のとおりです（sibss 領域を 4 バイトずつゼロクリアしていきます）。

```
    .extern __ssibss, 4
    .extern __esibss, 4
    mov     #__ssibss, r13
    mov     #__esibss, r12
    cmp     r12, r13
    jnl     .L26
.L25:
    st.w    r0, [r13]
    add     4, r13
    cmp     r12, r13
    jl      .L25
.L26:
```

7. 3. 17　関数のプロローグ・エピローグ・ランタイム・ライブラリ用の CTBP 値の設定【V850E】

V850Ex コアを使用している場合で，プロローグ／エピローグ・ランタイム・ライブラリを使用する場合にこの設定が必要になります。

V850Ex コアで関数のプロローグ／エピローグ・ランタイム・ライブラリを呼び出すとき，CALLT 命令を使用するため，その CALLT 命令に必要な CTBP の値を，関数のプロローグ・エピローグ・ランタイム・ライブラリの関数テーブルの先頭に設定しておく必要があります。

プロローグ／エピローグ・ランタイム・ライブラリを使用する設定になるのは，次の場合です。

- コンパイラ・オプション“-Xpro_epi_runtime=on”を設定している

コンパイラの最適化オプション“-Ot”以外を指定していると，自動的に“-Xpro_epi_runtime=on”になります。

関数のプロローグ／エピローグ・ランタイム・ライブラリの関数テーブルの先頭シンボルは，次のとおりです。

- ___PROLOG_TABLE

このシンボルを用いて，次のコードを記述します。

```
mov     #___PROLOG_TABLE, r12
ldsr    r12, 20
```

CTBP はシステム・レジスタ 20 番なので，ldsr 命令を使用して，値を設定します。

7. 3. 18 プログラマブル周辺 I/O レジスタ値の設定【V850E】

プログラマブル周辺 I/O レジスタを搭載している V850 マイクロコントローラを使用していて，かつ，プログラマブル周辺 I/O レジスタを使用する場合に，BPC を設定する必要があります。

たとえば，V850E/IA1 の場合，次のようになっています。

図 7―1 BPC レジスタ

15 0

PA15	0	PA13	PA12	PA11	PA10	PA9	PA8	PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0

表 7―8 BPC レジスタ

ビット位置	ビット名	意味
15	PA15	プログラマブル周辺 I/O エリアの使用許可／不許可を設定 0： プログラマブル周辺 I/O 領域の使用を不許可 1： プログラマブル周辺 I/O 領域の使用を許可
13-0	PA13-PA0	プログラマブル周辺 I/O エリアのアドレスを設定

プログラマブル周辺 I/O レジスタを使用する場合，コンパイラ・オプション“-Xbpc”で，プログラマブル周辺 I/O レジスタの値を設定する必要があります。これによって CA850 は，プログラマブル周辺 I/O レジスタへアクセスするコードを出力します。ただし，このオプションで BPC に値がセットされるわけではありません。

BPC に値をセットするには，スタート・アップ・ルーチン等で BPC レジスタに値を書き込む処理が必要となります。

V850E/IA1 の場合は，PA15 に 1 を立て，PA13 ～ PA0 にプログラマブル周辺 I/O エリアのアドレスを設定することになります。たとえば，プログラマブル周辺 I/O エリアのアドレスを 0x1234 としたい場合は，BPC レジスタへの設定は次のようになります。

```
mov     0x9234, r13
st.h    r13, BPC
```

PA15 に 1 を立てる必要があるため，0x1234 と 0x8000 の論理和（OR）を取った値を BPC に設定します。

CA850 のコンパイル・オプション“-Xbpc”に設定する値は 0x1234 で，BPC に設定する値は 0x9234 であるため，矛盾が生じないように注意する必要があります。

プログラマブル周辺 I/O レジスタについての詳細は，各デバイスのユーザーズ・マニュアルを参照してください。

7. 3. 19　r6 と r7 を main 関数の引数に設定

main 関数を，2 つの仮引数をもつ関数“int main (int argc, char *argv[ ]) { /* ... */ }”というように，2 つの仮引数をもつ関数として定義した場合，main 関数へ分岐する前に，引数（r6 と r7）へ値を設定する処理が必要になります。領域の確保は「7. 3. 4　main 関数の引数領域の確保」を参照してください。

なお，リアルタイム OS を使用したアプリケーションでは，main 関数は作成しないため，この処理は必要ありません。

r6 と r7 へ値を設定する処理は次のようになります。

```
ld.w    $__argc, r6
movea   $__argv, gp, r7
```

main 関数の引数の領域を .data セクションに配置したので，gp 相対のアクセス・コードを記述します。

7. 3. 20　main 関数へ分岐する（リアルタイム OS を使用していない場合）

スタート・アップ・ルーチンで行う必要のある処理がすべて終わったとき，main 関数への分岐命令を実行します。

ただし，リアルタイム OS を使用したアプリケーションの場合は，main 関数は作成しないため，この処理は必要ありません。代わりにリアルタイム OS の初期化ルーチンへ分岐する命令が必要になります。この詳細については「7. 3. 21　リアルタイム OS の初期化ルーチンへ分岐する（リアルタイム OS を使用している場合）」を参照してください。

main 関数への分岐には，次のコードを記述します。

```
jarl    _main, lp
```

また，main 関数の実行がすべて終わった後，この分岐命令の次の 4 バイトに戻ってくることになります。

戻って来ないことが分かっている場合は，次の命令も使用できます。

```
jr      _main
```

```
mov     #_main, lp
jmp     [lp]
```

jmp 命令を使用すると，32 ビット全空間をアクセスすることができます。

もし，“jarl _main, lp”を使用する場合は，main 関数実行後に戻ってくるので，戻ってきた後，デッドロックしないように，対策を施しておくと安全です。

7. 3. 21　リアルタイム OS の初期化ルーチンへ分岐する（リアルタイム OS を使用している場合）

リアルタイム OS を使用したアプリケーションで，スタート・アップ・ルーチンで行う必要のある処理がすべて終わったとき，初期化ルーチンへ分岐します。リアルタイム OS を使用していないアプリケーションの場合，main 関数へ分岐することになりますので「7. 3. 20　main 関数へ分岐する（リアルタイム OS を使用していない場合）」を参照してください。

【RI850V4 の場合】

```
    .extern __kernel_sit
    .extern __kernel_start
    mov     #__kernel_sit, r6
    mov     #__kernel_start, r11
    jarl    __jump_kernel_start, lp
__boot_error:
    jbr     __boot_error
__jump_kernel_start:
    jmp     [r11]
```

詳細については，リアルタイム OS のユーザーズ・マニュアルを参照してください。

7.4　コーディング例

スタート・アップ・ルーチンの例を，次に示します。

表 7―9　スタート・アップ・ルーチンの例

```
#------------------------------------------------------------------
# CA850 予約シンボルの外部ラベル宣言 1（tp，gp，ep 用）
#------------------------------------------------------------------
    .extern __tp_TEXT, 4
    .extern __gp_DATA, 4
    .extern __ep_DATA, 4
#------------------------------------------------------------------
# CA850 予約シンボルの外部ラベル宣言 2（bss 属性セクション初期化用）
# 使用していないセクションがある場合は削除。
# まだ使用するセクションが決まっていないような場合は，すべて書いておいてセクションの追加・削除によってスター
# ト・アップ・ルーチンのアセンブル・エラーを出さないようにしておくとよい
#------------------------------------------------------------------
    .extern __ssbss, 4
    .extern __esbss, 4
    .extern __sbss, 4
    .extern __ebss, 4
    .extern __ssebss, 4
    .extern __esebss, 4
    .extern __stibss.byte, 4
    .extern __etibss.byte, 4
    .extern __stibss.word, 4
    .extern __etibss.word, 4
    .extern __ssibss, 4
    .extern __esibss, 4
#------------------------------------------------------------------
# CA850 予約シンボルの外部ラベル宣言
# V850Ex で関数のプロローグ・エピローグ・ランタイム・ライブラリを使用するとき，関数テーブルの先頭アドレスを
# 外部ラベル宣言しておく
#------------------------------------------------------------------
    .extern ___PROLOG_TABLE
#------------------------------------------------------------------
# main 関数の外部ラベル宣言
#------------------------------------------------------------------
    .extern _main
#------------------------------------------------------------------
# main 関数の引数の領域（void main(void) 型の場合は必要なし）
#------------------------------------------------------------------
    .data
    .size   __argc, 4
    .align  4
```

```
__argc:
    .word   0
    .size   __argv, 4
__argv:
    .word   #.L16
.L16:
    .byte   0
    .byte   0
    .byte   0
    .byte   0
#------------------------------------------------------------------
# 以下はセクション生成のための“ダミーデータ”
# このダミーは，後で出てくる bss 属性のセクションをゼロクリアするためのもの
#
# その先頭シンボルと最後尾シンボルは，リンク時に該当セクションにデータが存在したときに生成される。しかし，ま
# だ使用するセクションが決まっていないような場合は，リンク・ディレクティブ・ファイルを書き換えてセクションの
# 追加・削除するたびに，スタート・アップ・ルーチンのアセンブル・エラーが出てしまう。これを避けるために，ダミ
# ー・データをセクションに配置することで，セクションの先頭シンボルと最後尾シンボルをとりあえず生成しておく
# bss セクションは，スタック生成コードでデータが割り当てられており，ここにダミーを作る必要がないため，記述し
# ていない。
#
# 使用するセクションが決まったときは，このダミー部分を削除し，また，ゼロクリア・ルーチンも必要なものだけ残し
# て削除すると，無駄がなくなり，コード効率が上がる
#------------------------------------------------------------------
    .sbss
    .lcomm  __sbss_dummy, 0, 0
    .sebss
    .lcomm  __sebss_dummy, 0, 0
    .tibss.byte
    .lcomm  __tibss_byte, 0, 0
    .tibss.word
    .lcomm  __tibss_word, 0, 0
    .sibss
    .lcomm  __sibss_dummy, 0, 0
#------------------------------------------------------------------
# スタック確保
# bss 領域に 0x200 バイト確保
#------------------------------------------------------------------
    .set    STACKSIZE, 0x200
    .bss
    .lcomm __stack, STACKSIZE, 4
#------------------------------------------------------------------
# リセット・ハンドラ
# リセット・ハンドラに配置する命令を記述
#------------------------------------------------------------------
    .section    "RESET", text
```

```
    jr      __start
#------------------------------------------------------------------------
# スタート・アップ・ルーチン本体
#------------------------------------------------------------------------
    .text
    .align  4
    .globl  __start
    .globl  __exit
    .globl  __startend
__start:
#------------------------------------------------------------------------
# __gp_DATA は tp からの相対値とすることをシンボル・ディレクティブで設定していることを想定
# そのため gp は tp に __gp_DATA の値を加算する
#------------------------------------------------------------------------
    mov     #__tp_TEXT, tp
    mov     #__gp_DATA, gp
    add     tp, gp
    mov     #__stack + STACKSIZE, sp
    mov     #__ep_DATA, ep
#------------------------------------------------------------------------
# マスク・レジスタの設定
# マスク・レジスタを使用しない場合はコード削減のため削除する
# 削除しなくても，プログラム中で書きつぶすので，動作上は問題ない
#------------------------------------------------------------------------
    .option nowarning
    mov     0xff, r20
    mov     0xffff, r21
    .option warning
.L11:
#------------------------------------------------------------------------
# sbss セクションのゼロクリア
# sbss セクションを使用していないときは，コード削減のため削除する
#------------------------------------------------------------------------
    .extern __ssbss, 4
    .extern __esbss, 4
    mov     #__ssbss, r13
    mov     #__esbss, r12
    cmp     r12, r13
    jnl     .L11
.L12:
    st.w    r0, [r13]
    add     4, r13
    cmp     r12, r13
    jl      .L12
#------------------------------------------------------------------------
```

```
# bss セクションのゼロクリア
# bss セクションを使用していないときは，コード削減のため削除する
#----------------------------------------------------------------------
    .extern __sbss, 4
    .extern __ebss, 4
    mov     #__sbss, r13
    mov     #__ebss, r12
    cmp     r12, r13
    jnl     .L14
.L15:
    st.w    r0, [r13]
    add     4, r13
    cmp     r12, r13
    jl      .L15
.L14:
#----------------------------------------------------------------------
# sebss セクションのゼロクリア
# sebss セクションを使用していないときは，コード削減のため削除する
#----------------------------------------------------------------------
    .extern __ssebss, 4
    .extern __esebss, 4
    mov     #__ssebss, r13
    mov     #__esebss, r12
    cmp     r12, r13
    jnl     .L17
.L18:
    st.w    r0, [r13]
    add     4, r13
    cmp     r12, r13
    jl      .L18
.L17:
#----------------------------------------------------------------------
# tibss.byte セクションのゼロクリア
# tibss.byte セクションを使用していないときは，コード削減のため削除する
#----------------------------------------------------------------------
    .extern __stibss.byte, 4
    .extern __etibss.byte, 4
    mov     #__stibss.byte, r13
    mov     #__etibss.byte, r12
    cmp     r12, r13
    jnl     .L20
.L21:
    st.w    r0, [r13]
    add     4, r13
    cmp     r12, r13
```

```
    jl      .L21
.L20:
#-------------------------------------------------------------------
# tibss.word セクションのゼロクリア
# tibss.word セクションを使用していないときは，コード削減のため削除する
#-------------------------------------------------------------------
    .extern __stibss.word, 4
    .extern __etibss.word, 4
    mov     #__stibss.word, r13
    mov     #__etibss.word, r12
    cmp     r12, r13
    jnl     .L23
.L24:
    st.w    r0, [r13]
    add     4, r13
    cmp     r12, r13
    jl      .L24
.L23:
#-------------------------------------------------------------------
# sibss セクションのゼロクリア
# sibss セクションを使用していないときは，コード削減のため削除する
#-------------------------------------------------------------------
    .extern __ssibss, 4
    .extern __esibss, 4
    mov     #__ssibss, r13
    mov     #__esibss, r12
    cmp     r12, r13
    jnl     .L26
.L25:
    st.w    r0, [r13]
    add     4, r13
    cmp     r12, r13
    jl      .L25
.L26:
#-------------------------------------------------------------------
# 関数のプロローグ・エピローグ・ランタイム・ライブラリの設定
# ライブラリ関数テーブルの先頭アドレスを CTBP（システムレジスタ 20 番）にセット
# V850Ex 以外は，この記述は削除
#-------------------------------------------------------------------
    mov     #___PROLOG_TABLE, r12
    ldsr    r12, 20
#-------------------------------------------------------------------
# プログラマブル周辺 I/O レジスタの設定
# プログラマブル周辺 I/O レジスタを持たない V850 の場合はこの記述は削除
# 下記は BPC レジスタの値（設定アドレス）が 0x1234 の例
```

```
# 0x1234（アドレス） と 0x8000（プログラマブル周辺 I/O 使用）の論理和を BPC に設定する
#----------------------------------------------------------------------
    mov     0x9234, r13
    st.h    r13, BPC
#----------------------------------------------------------------------
# main 関数の引数を r6，r7 に設定
#----------------------------------------------------------------------
    ld.w    $__argc, r6
    movea   $__argv, gp, r7
#----------------------------------------------------------------------
# main 関数へ分岐
#----------------------------------------------------------------------
    jarl    _main, lp
#----------------------------------------------------------------------
# main 関数が戻ってきた後の処理
#----------------------------------------------------------------------
__exit:
    halt
__startend:
```

第 8 章　ROM 化

この章では，ROM 化プロセッサ（romp850）の概要，ROM 化の手順，操作方法などを説明します。

8.1　概　　要

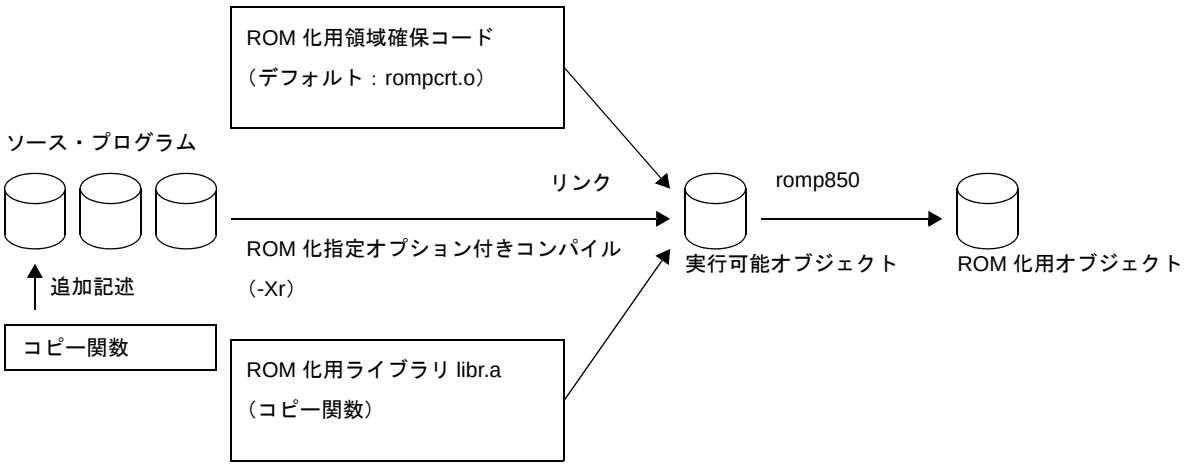
プログラム中で，グローバルに変数を宣言すると，初期値を持つ変数ならば data 属性のセクションへ，初期値を持たない変数ならば bss 属性のセクションというように，RAM 上のセクションに配置されます。特に初期値を持つ変数ならば，その初期値自体が RAM 上に配置されます。その他，アプリケーションの高速化のために，プログラム・コードを内蔵 RAM 領域へ配置する場合もあります。

組み込みシステムの場合，デバッグ時にインサーキット・エミュレータなどを使用する場合，実行可能なモジュールを配置イメージのままダウンロードして実行できます。しかし実際にプログラムをターゲット・システムの ROM 領域に書き込んで実行する場合，data 属性のセクションにある初期値情報や，RAM 領域に配置するプログラム・コードを，実行前に RAM 上に展開されていなければなりません。つまり RAM に展開するデータを ROM 上に持たせておき，それをアプリケーション実行前に ROM から RAM へコピーする作業が必要になります。

romp850 は，data 属性セクションの変数の初期値情報や，RAM 上に配置するプログラムを，1 つのセクションにパッキングするツールです。このセクションを ROM 上に配置し，CA850 で用意されているコピー関数を呼び出すことによって，初期値情報やプログラムを容易に RAM 上へ展開することができます。

ROM 化用オブジェクトを作成する流れの概要は，次図のようになります。

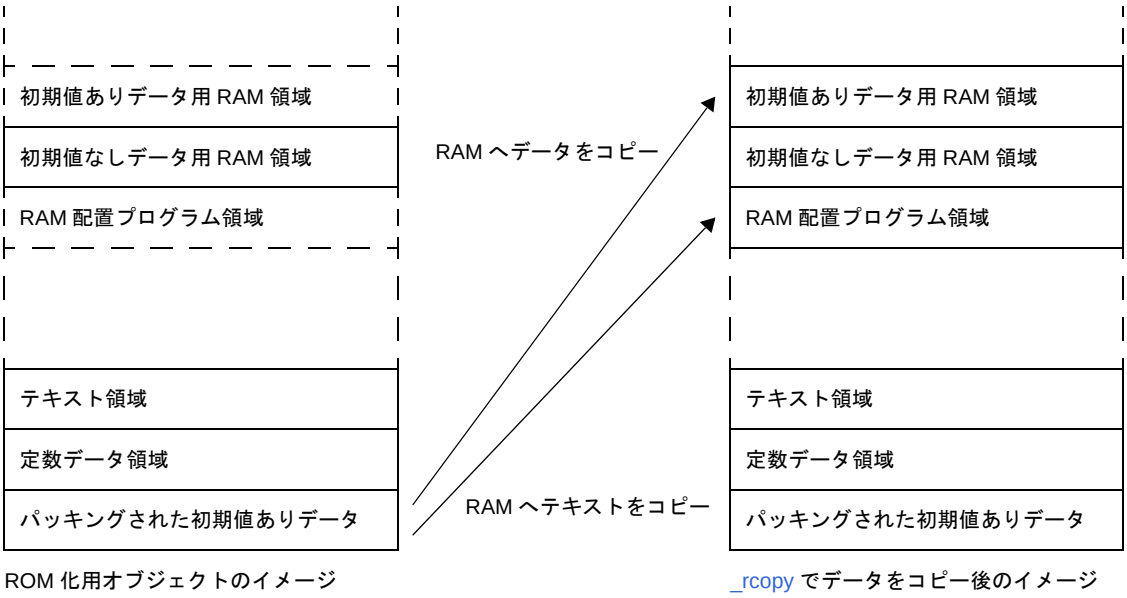
図 8―1　ROM 化用オブジェクトの作成



「表 8―1　コピー関数」のように ROM 化用オブジェクトを作成すると，コピー関数 _rcopy を実行することによって，RAM に配置するデータを，パッキングされた ROM からコピーします。

イメージは次のようになります。

図 8―2 コピー関数呼び出し前後のイメージ



ここで，ROM 化用オブジェクトに必要となるセクション名，およびそのセクションの先頭アドレス（ラベル名）は，デフォルトでは次のようになっています。

- パッキングしたセクション名 → rompsec セクション
- rompsec セクションの先頭アドレス（ラベル名）→ __S_romp

そして rompsec セクションから，RAM 領域へコピーする関数は次のとおりです。

- コピー関数 → _rcopy，_rcopy1，_rcopy2，_rcopy4

この関数は Install Folder \ lib850 \ r** にあるライブラリ“libr.a”に格納されています。

__S_romp は Install Folder \ lib850 \ r** にある“rompcrt.o”で定義されているラベルです（このソース・ファイルは rompcrt.s）。rompcrt.o をそのまま使用することにより，romp850 によって自動的に .text 属性の直後（4 バイトでアラインしたところ）に，rompsec セクションを作成します。そして __S_romp が rompsec セクションの先頭アドレスを指すラベルになります。

このように自動的に rompsec セクションを作成する方法のほかに，rompcrt.s に相当するプログラムを独自に作成して配置することもできます。

実際に ROM 化するには，この ROM 化用オブジェクトを作成してから，ヘキサ・ファイルに変換し，ROM 上に書き込むことになります。

なお，パッキングの必要なデータがアプリケーションに存在しなかった場合は，この ROM 化用オブジェクトを生成する必要はありません。ld850 で作成したオブジェクトを，そのままヘキサ・ファイルに変換してください。

また，romp850 は，リロケーション解決したオブジェクト・ファイルにシンボル情報，デバッグ情報が含まれる場合，それらを削除することなく ROM 化用のオブジェクト・ファイルを生成します。そのため，ROM 化後のオブジェクト・ファイルでもデバッガによるソース・デバッグができます。

8. 2 rompsec セクション

この節では，rompsec セクションについて説明します。

8. 2. 1 パッキングするセクションの種類

rompsec セクションとしてパッキングする対象となるものは，デフォルトでは「書き込み可能な属性を持つセクションに割り当てられたデータ」です。その他，オプション（-t オプション）を指定することによって「text 属性，const 属性を持つ任意のセクション」をパッキングすることも可能です。

具体的には次に示すようになります。

- 予約セクション（.data，.sdata，.sedata，.sidata，.tidata，.tidata.byte，.tidata.word）
- アセンブラ・プログラムにおいて .section 疑似命令により sdata 属性，または data 属性を指定し，任意の名前で生成したセクション，および内蔵命令 RAM に配置するセクション（V850E2 コアを持つデバイス指定時は対象となりません）

ただし，ユーザが指定する「text 属性，const 属性を持つ任意のセクション」をパッキングせず，さらに上記のセクションが実行可能モジュールに存在しない場合は，ROM 化オブジェクトを生成する必要はありません。

予約セクション（.data，.sdata，.sedata，.sidata，.tidata，.tidata.byte，.tidata.word）が存在するかしないかは，リンク・マップ・ファイルを参照してください。

なお，romp850 によって作成されたオブジェクト・ファイルをダンプ・コマンド（dump850）で参照することにより，.data セクションや .sdata セクションなどの代わりに rompsec セクションが作成されていることを確認できます。

8. 2. 2 rompsec セクションのサイズ

rompsec セクションとして確保されるサイズについて説明します。

ROM 化用モジュールを作成するときは，rompsec セクションのサイズと，使用している CPU の内蔵 ROM 領域，ターゲット・システムの ROM 領域のアドレスやサイズに注意します。rompsec セクションが，他のセクションとオーバラップしないようにリンク・ディレクティブ・ファイルを記述してください。

次に rompsec セクションのサイズを求める計算式を示します。

8 + 16 * (sdata ／ data 属性セクションの数) + sdata ／ data 属性セクションのサイズ + パディング・サイズ **注**

たとえば，.sdata，.data セクションが存在し，それぞれのサイズが 1002 バイト，1000 バイトで，それぞれのセクションの整列条件が 4 バイトの場合，rompsec セクションのサイズは次のようになります。

8 + 16 * 2 + 1002 + 1000 + 2 = 2044（単位：バイト）

注 ROM 化対象のセクションの整列条件により 1 セクションあたり 0 ～ 3 バイトになります。

8. 2. 3 rompsec セクションとリンク・ディレクティブ

ROM 化時には，.text セクションの直後に rompsec セクションが追加されます。よって，.text セクションを ROM の最後に配置することで，ROM の終端までの rompsec セクションを配置できます。

図 8―3 ROM 化処理を考慮したリンク・ディレクティブ

```
# 内蔵 ROM に SCONST ／ CONST ／ TEXT を配置
SCONST  : !LOAD ?R {
    .sconst = $PROGBITS ?A .sconst;
};
CONST   : !LOAD ?R {
    .const = $PROGBITS ?A .const;
};
# 内蔵 ROM の最後に .text を配置
TEXT    : !LOAD ?RX {
    .pro_epi_runtime = $PROGBITS ?AX .pro_epi_runtime;
    .text = $PROGBITS ?AX .text;
};
# 外部 RAM に DATA を配置
DATA    : !LOAD ?RX V0x100000 {
    .data = $PROGBITS ?AW;
    .sdata = $PROGBITS ?AWG;
    .sbss = $NOBIT ?AWG;
    .bss = $NOBIT ?AW;
};
# 内蔵 RAM に SIDATA を配置
SIDATA  : !LOAD ?RX V0xffe000 {
    .sidata = $PROGBITS ?AW .sidata;
    .sibss = $NOBIT ?AWG .sibss;
};
__tp_TEXT@%TP_SYMBOL;
__gp_DATA@%GP_SYMBOL &__tp_TEXT{DATA};
__ep_DATA@%EP_SYMBOL;
```

rompsec セクションが内蔵 ROM 領域を越えた場合には，次のメッセージを出力して処理を中止します。

```
F8425: rompsec section overflowed highest address of target machine.
```

-rom_less オプションを指定することで，内蔵 ROM 領域を無視することができます。

また，-Ximem_overflow=warning オプションを指定することで，エラー・メッセージを警告メッセージにすることができます。

rompsec セクションを外部 ROM 領域の終端に配置する場合には，これらのチェックは行われません。メモリマップ情報を参照して，ROM に収まっているかの判断を行ってください。

なお，ROM の途中に rompsec セクションを配置する必要がある場合には，次のように rompsec セクションのサイズと配置アドレスから，rompsec セクションの配置される領域を認識した上で，rompsec セクション直後のセグメントに対して，適切なアドレス指定をしてください。

図 8―4 ROM 化処理を考慮したリンク・ディレクティブ（サイズ考慮）

```
# 内蔵 ROM に SCONST ／ CONST ／ TEXT を配置
SCONST   : !LOAD ?R {
    .sconst = $PROGBITS ?A .sconst;
};
# 内蔵 ROM の途中に .text を配置
TEXT     : !LOAD ?RX {
    .pro_epi_runtime = $PROGBITS ?AX .pro_epi_runtime;
    .text = $PROGBITS ?AX .text;
};
#TEXT と CONST の間に rompsec
# 内蔵 ROM の最後に rompsec のサイズを考慮したアドレス指定を行って CONST を配置
CONST    : !LOAD ?R Vx3f800 {
    .const = $PROGBITS ?A .const;
};
# 外部 RAM に DATA を配置
DATA     : !LOAD ?RX V0x100000 {
    .data = $PROGBITS ?AW;
    .sdata = $PROGBITS ?AWG;
    .sbss = $NOBIT ?AWG;
    .bss = $NOBIT ?AW;
};
# 内蔵 RAM に SIDATA を配置
SIDATA   : !LOAD ?RX V0xffe000 {
    .sidata = $PROGBITS ?AW .sidata;
    .sibss = $NOBIT ?AWG .sibss;
};
__tp_TEXT@%TP_SYMBOL;
__gp_DATA@%GP_SYMBOL &__tp_TEXT{DATA};
__ep_DATA@%EP_SYMBOL;
```

8. 3 ROM 化用オブジェクトの作成

この節では，ROM 化用オブジェクトの作成手順について説明します。

8. 3. 1 作成手順（デフォルト）

ここでは，デフォルトで用意されている ROM 化用領域確保コード（rompcrt.o）を使用した方法を示します。

(1) コピー関数の呼び出し

コピー関数はスタート・アップ・ルーチン内や main 関数の先頭など，なるべく始めの方で起動するようにします。コピー関数には _rcopy，_rcopy1，_rcopy2，_rcopy4 があり，それぞれ転送サイズに違いがあります（_rcopy と _rcopy1 は同じ）。

次図に，main 関数の先頭でコピー関数 _rcopy を呼び出す場合の例を示します。

図 8―5 コピー関数の使用例 1

```
#define ALL_COPY    (-1)

int _rcopy(unsigned long *, long);
extern  unsigned long   _S_romp;

void main(void){
    int ret;
    ret = _rcopy(&_S_romp, ALL_COPY);
      :
}
```

(2) .text セクションの直後に rompsec セクションを追加

.text セクションを ROM の最後に配置することで，ROM の終端までの rompsec セクションを配置できます。

(3)“ROM 化用オブジェクトの生成”の指定

以下のいずれかの操作を行うことにより，__S_romp というラベルが，オブジェクト内の .text セクションの終端を越える最初のアドレスを指すコードが生成されます。

- コマンド・ラインからの場合

  コンパイル・オプション“-Xr”を追加指定します。

- CubeSuite+ からの場合

  プロパティ パネルの［ROM 化プロセス・オプション］タブをオープンし，［出力ファイル］カテゴリの［ROM 化用オブジェクト・ファイルを出力する］プロパティで［はい (-Xr -lr)］を選択します。

(4) ROM 化プロセス・オプションの指定

- CubeSuite+ からの場合

  プロパティ パネルの［ROM 化プロセス・オプション］タブをオープンし，［入力ファイル］カテゴリの［標準の ROM 化用領域確保コード・ファイルを使用する］プロパティで［はい］（デフォルト）を選択します。

(5) コンパイル，リンク

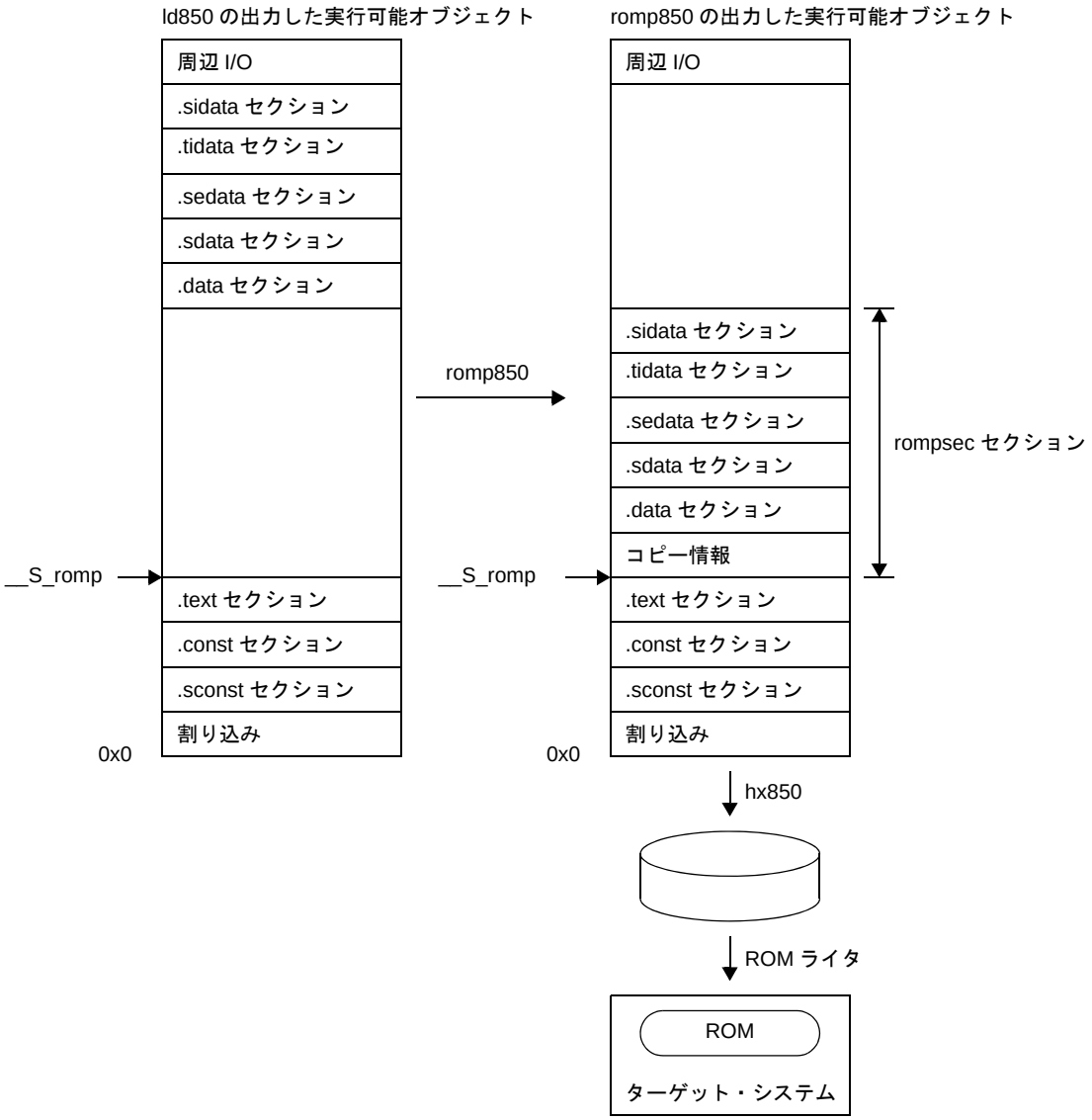
ca850 対して「ROM 化用オブジェクトの生成」を指示することにより，ROM 化用領域確保コードである“rompcrt.o”（Install Folder \ lib850 \ r** 内に存在）と，コピー関数 _rcopy が格納されている“libr.a”が自動的にリンクされます。この際，リンクする順番が関係します。“rompcrt.o”は，TEXT 属性群の最後にリンクする必要がありますので，コマンド・ラインから起動している場合，-l オプションでリンク指定するライブラリ群よりも後にリンクしてください。

(6) ROM 化プロセッサ（romp850）の起動

(5) で完成した実行可能モジュールから，romp850 を使用して ROM 化用モジュールを生成します。

なお，コマンド・ラインから起動している場合は，ca850 から ld850 の起動まで行って実行可能モジュールを生成したあと，romp850 を起動して ROM 化用オブジェクトを生成します。マップのイメージを図にすると，次のようになります。

図 8―6 ROM 化のイメージ 1



8. 3. 2 作成手順（カスタマイズ）

ここでは，ROM 化用領域確保コードにあたる“rompcrt.o”を独自に作成し，rompcrt セクションの先頭アドレスや配置場所を自分で決定する方法を示します。

（1）デフォルトの ROM 化用領域確保コード“rompcrt.s”に相当するコードの記述

ここではファイル名を“rompack.s”，ROM 化用領域の先頭を指すシンボル名を“__rompack”とします。また，このとき，このシンボルの存在するセクションを“rompack セクション”とします。この場合，rompack.s は，次のようなコードになります。

図 8―7 rompack.s の例

```
    .file       "rompack.s"
    .section    ".rompack", text
    .align      4
    .globl      __rompack, 4
__rompack:
```

（2）コピー関数の呼び出し

コピー関数はスタート・アップ・ルーチン内や main 関数の先頭など，なるべく始めの方で起動するようにします。コピー関数には _rcopy，_rcopy1，_rcopy2，_rcopy4 があり，それぞれ転送サイズに違いがあります（_rcopy と _rcopy1 は同じ）。

次図に，main 関数の先頭でコピー関数 _rcopy を呼び出す場合の例を示します。

図 8―8 コピー関数の使用例 2

```
#define ALL_COPY    (-1)

int _rcopy(unsigned long *, long);
extern  unsigned long   _rompack;

void main(void){
    int ret;
    ret = _rcopy(&_rompack, ALL_COPY);
      :
}
```

（3）rompack セクションの定義

これと同時にアドレスを指定すると，rompack セクションの配置場所を任意に決定することもできます。

例として rompack セクションを含むセグメントを ROMPACK とし，このセグメントを 0x3000 番地に配置するとした場合，リンク・ディレクティブは次のようになります。

図 8―9 リンク・ディレクティブの指定例

```
TEXT    : !LOAD ?RX V0x1000 {
    .text = $PROGBITS ?AX .text;
};
ROMPACK : !LOAD ?RX V0x3000 {
    .rompack = $PROGBITS ?AX .rompack;
};
```

このとき，ROMPACK セグメントの配置アドレスが，前後のセグメントと重ならないように，rompack セクションのサイズを，「8. 2. 2 rompsec セクションのサイズ」にしたがって見積もり，リンク・ディレクティブ・ファイルに反映します。

(4)“ROM 化用オブジェクトの生成”の指定

以下のいずれかの操作を行うことにより，ラベル“rompack”が rompsec と同じアドレスを指すコードを生成します。

- コマンド・ラインからの場合

  コンパイル・オプション“-Xr”を追加指定します。

- CubeSuite+ からの場合

  プロパティ パネルの［ROM 化プロセス・オプション］タブをオープンし，［出力ファイル］カテゴリの［ROM 化用オブジェクト・ファイルを出力する］プロパティで［はい (-Xr -lr)］を選択します。

(5)コンパイラ共通オプション，ROM 化プロセッサ・オプションの指定

以下のいずれかの操作を行い，オプションの指定を行います。

- コマンド・ラインからの場合

  ROM 化プロセッサのオプションで，ROM 化用領域確保コードのエントリ・シンボルを指定する“-b オプション”で“__rompack”を指定します。

- CubeSuite+ からの場合

  プロパティ パネルの［ROM 化プロセス・オプション］タブをオープンし，［入力ファイル］カテゴリの［標準の ROM 化用領域確保コード・ファイルを使用する］プロパティで［いいえ］を選択し，［ROM 化用領域確保コードのファイル名］プロパティに“rompack.s”，または“rompack.o”を追加します。

  ［その他］カテゴリの［エントリ・ラベル］プロパティに rompack セクションの先頭ラベルである“__rompack”を指定します。

(6)コンパイル，リンク

ca850 に対して「ROM 化用オブジェクトの生成」を指示することにより，コピー関数 _rcopy が格納されている“libr.a”が自動的にリンクされます。

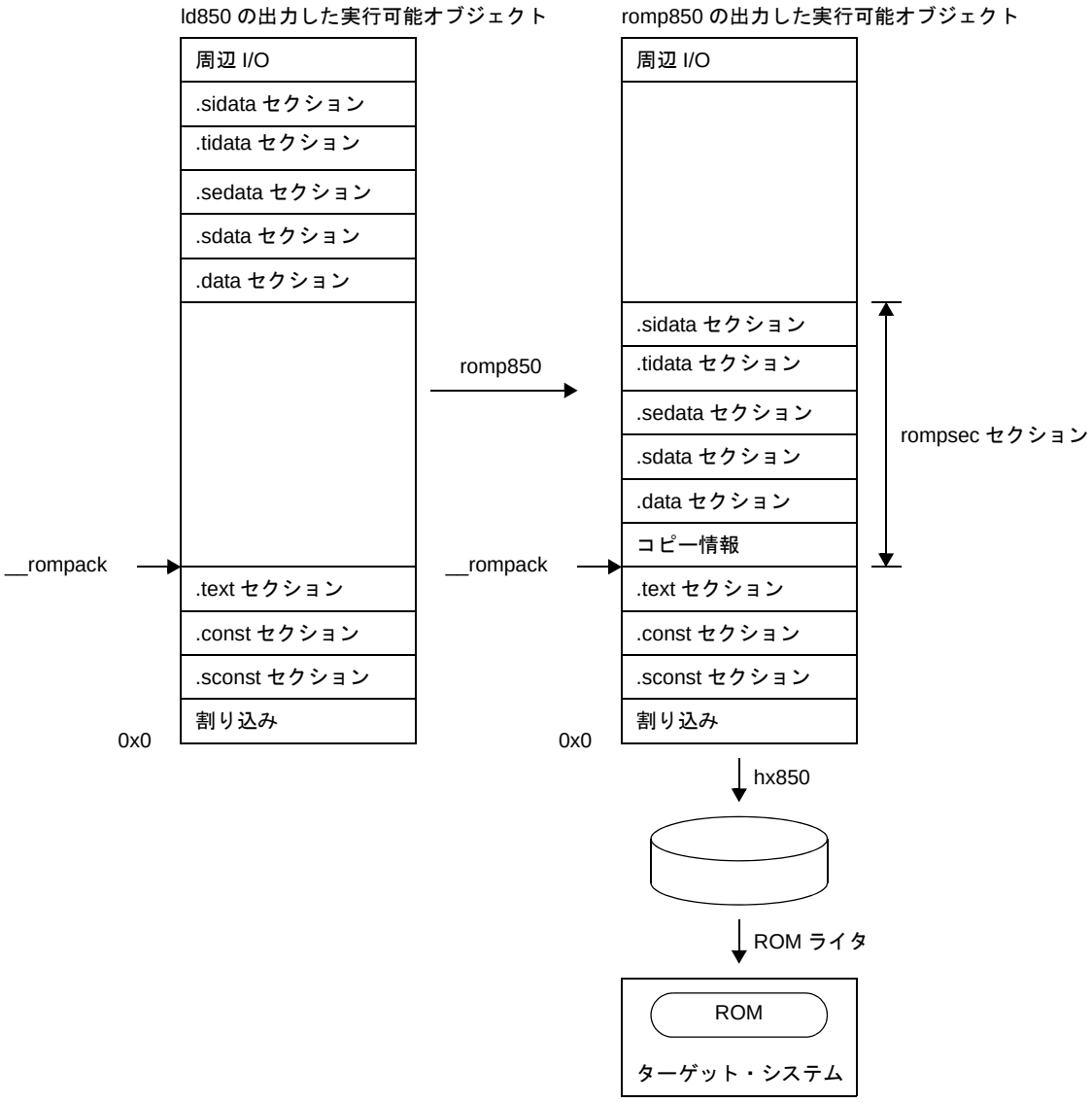
(7)ROM 化プロセッサ（romp850）の起動

(6) で完成した実行可能モジュールから，romp850 を使用して ROM 化用モジュールを作成します。

コマンド・ラインから起動している場合は，ca850 から ld850 の起動まで行って実行可能モジュールを作成し，romp850 を起動して ROM 化用オブジェクトを作成します。

マップのイメージを次図に示します。

図 8―10 ROM 化のイメージ 1



8.4 コピー関数

ここでは，ROM 化するプログラムに必要なコピー関数について説明します。

表 8―1 コピー関数

関数名	機能
_rcopy	パッキング・データを 1 バイトずつ RAM へコピーする（_rcopy1 と同じ）
_rcopy1	パッキング・データを 1 バイトずつ RAM へコピーする（_rcopy と同じ）
_rcopy2	パッキング・データを 2 バイトずつ RAM へコピーする
_rcopy4	パッキング・データを 4 バイトずつ RAM へコピーする

転送先の RAM の仕様に応じて，1 バイト転送，2 バイト転送，4 バイト転送を使い分けてください。

_rcopy

初期値データ／RAM テキスト注のコピー（1 バイト）を行います。

注 RAM に配置する初期値ありデータ・セクション，および内蔵 RAM 用テキスト・セクションです。

[所属]

ROM 化用ライブラリ libr.a

[指定形式]

int _rcopy(&*label*, *number*);
unsigned long *label*;
long *number*;

[戻り値]

0	正常終了（正しくコピーされた場合）
-1	異常終了（正しくコピーされなかった場合）

[詳細説明]

label の示すアドレス以降に存在する rompsec セクション内の情報を元に，コピーしたいセクション番号 *number* の初期値データ，または RAM に配置するテキストを，RAM 領域に 1 バイトずつコピーします。*number* に -1 を指定した場合，rompsec セクション内のすべてのセクションをコピーします。セクション番号 *number* は，1 から始まる正数です。

デフォルトは，セクションが入力ファイル中に出現した順番に割り当てられます。romp850 のオプション“-p オプション”，“-t オプション”で rompsec セクションに配置するセクションを指定した場合は，指定した順番に割り当てられます。

ただし，CubeSuite+ において“ROM 化セクションファイル”を作成すると，#define による“番号”と“ラベル”の対応付けされた C 言語ソース・ヘッダファイルが生成され，ラベル名によって，*number* を指定することができます。

［注意事項］

- *label* の示すアドレスが rompsec セクションの先頭でなかった場合はコピーを行いません。
- 本関数は romp850 で生成された情報にしたがってコピーを行います。本関数実行時にコピー先のアドレスにオフセットを加えるような処理はできません。
- コピーを行うとオーバライトが生じる場合，コピーを行いません。
- 本関数の第一引数 *label* には，絶対値を持つグローバルなラベル，または絶対アドレスを指定してください。これら以外のものを指定した場合，その結果は保証されません。
- 本関数は _rcopy1 と同じ機能です。旧版からの互換性のために本関数を用意してあります。

_rcopy1

初期値データ／RAM テキスト注のコピー（1 バイト）を行います。

注 RAM に配置する初期値ありデータ・セクション，および内蔵 RAM 用テキスト・セクションです。

[所属]

ROM 化用ライブラリ libr.a

[指定形式]

int _rcopy1(&*label*, *number*);
unsigned long *label*;
long *number*;

[戻り値]

0	正常終了（正しくコピーされた場合）
-1	異常終了（正しくコピーされなかった場合）

[詳細説明]

label の示すアドレス以降に存在する rompsec セクション内の情報を元に，コピーしたいセクション番号 *number* の初期値データ，または RAM に配置するテキストを，RAM 領域に 1 バイトずつコピーします。*number* に -1 を指定した場合，rompsec セクション内のすべてのセクションをコピーします。セクション番号 *number* は，1 から始まる正数です。

デフォルトは，セクションが入力ファイル中に出現した順番に割り当てられます。romp850 のオプション“-p オプション”，“-t オプション”で rompsec セクションに配置するセクションを指定した場合は，指定した順番に割り当てられます。

ただし，CubeSuite+ において“ROM 化セクションファイル”を作成すると，#define による“番号”と“ラベル”の対応付けされた C 言語ソース・ヘッダファイルが生成され，ラベル名によって，*number* を指定することができます。

［注意事項］

- *label* の示すアドレスが rompsec セクションの先頭でなかった場合はコピーを行いません。
- 本関数は romp850 で生成された情報にしたがってコピーを行います。本関数実行時にコピー先のアドレスにオフセットを加えるような処理はできません。
- コピーを行うとオーバライトが生じる場合，コピーを行いません。
- 本関数の第一引数 *label* には，絶対値を持つグローバルなラベル，または絶対アドレスを指定してください。これら以外のものを指定した場合，その結果は保証されません。
- 本関数は _rcopy と同じ機能です。旧版からの互換性のために _rcopy を用意してあります。

_rcopy2

初期値データ／RAM テキスト[注]のコピー（2 バイト）を行います。

注 RAM に配置する初期値ありデータ・セクション，および内蔵 RAM 用テキスト・セクションです。

[所属]

ROM 化用ライブラリ libr.a

[指定形式]

int _rcopy2(&*label*, *number*);
unsigned long *label*;
long *number*;

[戻り値]

0	正常終了（正しくコピーされた場合）
-1	異常終了（正しくコピーされなかった場合）

[詳細説明]

label の示すアドレス以降に存在する rompsec セクション内の情報を元に，コピーしたいセクション番号 *number* の初期値データ，または RAM に配置するテキストを，RAM 領域に 2 バイトずつコピーします。*number* に -1 を指定した場合，rompsec セクション内のすべてのセクションをコピーします。セクション番号 *number* は，1 から始まる正数です。

デフォルトは，セクションが入力ファイル中に出現した順番に割り当てられます。romp850 のオプション“-p オプション”，“-t オプション”で rompsec セクションに配置するセクションを指定した場合は，指定した順番に割り当てられます。

ただし，CubeSuite+ において“ROM 化セクションファイル”を作成すると，#define による“番号”と“ラベル”の対応付けされた C 言語ソース・ヘッダファイルが生成され，ラベル名によって，*number* を指定することができます。

[注意事項]

- *label* の示すアドレスが rompsec セクションの先頭でなかった場合はコピーを行いません。
- 本関数は romp850 で生成された情報にしたがってコピーを行います。本関数実行時にコピー先のアドレスにオフセットを加えるような処理はできません。
- コピーを行うとオーバライトが生じる場合，コピーを行いません。
- 本関数の第一引数 *label* には，絶対値を持つグローバルなラベル，または絶対アドレスを指定してください。これら以外のものを指定した場合，その結果は保証されません。

_rcopy4

初期値データ／RAM テキスト[注]のコピー（4 バイト）を行います。

注 RAM に配置する初期値ありデータ・セクション，および内蔵 RAM 用テキスト・セクションです。

[所属]

ROM 化用ライブラリ libr.a

[指定形式]

int _rcopy4(&*label*, *number*);
unsigned long *label*;
long *number*;

[戻り値]

0	正常終了（正しくコピーされた場合）
-1	異常終了（正しくコピーされなかった場合）

[詳細説明]

label の示すアドレス以降に存在する rompsec セクション内の情報を元に，コピーしたいセクション番号 *number* の初期値データ，または RAM に配置するテキストを，RAM 領域に 4 バイトずつコピーします。*number* に -1 を指定した場合，rompsec セクション内のすべてのセクションをコピーします。セクション番号 *number* は，1 から始まる正数です。

デフォルトは，セクションが入力ファイル中に出現した順番に割り当てられます。romp850 のオプション“-p オプション”，“-t オプション”で rompsec セクションに配置するセクションを指定した場合は，指定した順番に割り当てられます。

[注意事項]

- *label* の示すアドレスが rompsec セクションの先頭でなかった場合はコピーを行いません。
- 本関数は romp850 で生成された情報にしたがってコピーを行います。本関数実行時にコピー先のアドレスにオフセットを加えるような処理はできません。
- コピーを行うとオーバライトが生じる場合，コピーを行いません。
- 本関数の第一引数 *label* には，絶対値を持つグローバルなラベル，または絶対アドレスを指定してください。これら以外のものを指定した場合，その結果は保証されません。

第 9 章　コンパイラとアセンブラの相互参照

この章では，CA850 におけるプログラム呼び出し時の引数などの扱い方について説明します。

9.1　引数，自動変数のアクセス方法

(1) アセンブラ関数への引数

CA850 は 4 ワード分の引数を“引数用レジスタ（r6 ～ r9）”に格納し，それを越えた分の引数は，呼び出し側のスタック・フレームに格納します。アセンブラ関数内で引数値を使用するときは，それぞれに格納された値を参照してください。

ただし，構造体を返すアセンブラ関数の場合，3 ワード分の引数を“引数用レジスタ（r7 ～ r9）”に格納し，それを越えた分の引数は，呼び出し側のスタック・フレームに格納します。そして，r6 レジスタには戻り値を格納するアドレスを格納しますので，引数格納場所に注意が必要です。

なお，引数値は C 言語関数において，引数に指定された値そのもので，この値をアセンブラ関数内で変更しても，C 言語関数の動作に影響を及ぼすことはありません。

(2) C 言語関数への引数

CA850 は 4 ワード分の引数を“引数用レジスタ（r6 ～ r9）”に格納し，それを越えた分の引数は，呼び出し側のスタック・フレームに格納します。4 ワード分を越えた引数は，SP の指すアドレスから，上位方向に向かって格納してください。

ただし，構造体を返す C 言語関数の場合，3 ワード分の引数を“引数用レジスタ（r7 ～ r9）”に格納し，それを越えた分の引数は，呼び出し側のスタック・フレームに格納します。そして，r6 レジスタには戻り値を格納するアドレスを格納します。

9.2　返り値の格納方法

(1) アセンブラ関数からの戻り値

CA850 は「関数の戻り値は“レジスタ r10”に格納される」ことを想定したコードを生成します。アセンブラ関数からの戻り値は r10 に格納するようにしてください。

なお，構造体を返す関数の場合は“呼び出し側の関数のスタック・フレーム内”に戻り値である構造体が格納されます。

(2) C 言語関数からの戻り値

CA850 は「関数の戻り値は“レジスタ r10”に格納される」ことを想定したコードを生成します。C 言語関数からの戻り値を使用する場合は，r10 レジスタを参照してください。

なお，構造体を返す関数の場合は，呼び出し側の戻り値用の領域に値が格納され，その領域のアドレスを引数として渡す形のコードを出力します。呼び出し側であらかじめ戻り値用の領域を確保しておく必要があります。

9.3　C 言語からアセンブリ言語ルーチンの呼び出し

C 言語関数からアセンブラ関数を呼び出すときの注意点について説明します。

(1) 識別子について

CA850 では C 言語ソース内で外部名，たとえば，関数や外部変数が記述された場合，それらの名前をアセンブラへ出力すると，先頭に“_（アンダースコア）”を付けた名前になります。

表 9―1　識別子について

C	アセンブラ
func1 ( )	_func1

アセンブラ命令で関数や外部変数を定義するときは，識別子の先頭に“_”をつけ，C 言語関数から参照するときは“_”を取った形で行ってください。

(2) スタック・フレームに関して

CA850 は「スタック・ポインタ（SP）が，常にスタック・フレームの最下位アドレスを指している」ことを想定したコードを出力します。そのため，C 言語ソースからアセンブラ関数へ分岐後は，アセンブラ関数内では，SP の指すアドレスよりも下位のアドレス領域は自由に使用することができます。逆に上位のアドレス領域の内容を変更した場合，C 言語関数で使用していた領域を破壊することにつながり，以降の動作を保証できませんので注意が必要です。上記を回避するためには，アセンブラ関数の先頭で SP を変更してからスタックを使用してください。

ただし，その際は呼び出しの前後で SP の値が保持されるようにしてください。

また，アセンブラ関数内でレジスタ変数用レジスタを使用する場合は，アセンブラ関数の呼び出し前後でレジスタ値が保持されるようにしてください（使用前にレジスタ変数用レジスタの値を退避し，使用後は復帰してください）。

レジスタ変数用レジスタは，レジスタ・モードにより異なります。

表 9―2　レジスタ変数用レジスタ

レジスタ・モード	レジスタ変数用レジスタ
22 レジスタ・モード	r25，r26，r27，r28，r29
26 レジスタ・モード	r23，r24，r25，r26，r27，r28，r29
32 レジスタ・モード	r20，r21，r22，r23，r24，r25，r26，r27，r28，r29

(3) C 言語関数への戻り先アドレス

CA850 は「関数の戻り先アドレスは“リンク・ポインタ lp（r31）”に格納される」ことを想定したコードを生成します。アセンブラ関数へ分岐するとき，lp に関数の戻り先アドレスが格納されているので，C 言語関数へ戻るときは“jmp [lp]”を実行してください。

9.4　アセンブリ言語から C 言語ルーチンの呼び出し

アセンブラ関数から C 言語関数を呼び出すときの注意点について説明します。

(1) スタック・フレームについて

CA850 は「スタック・ポインタ（SP）が，常にスタック・フレームの最下位アドレスを指している」ことを想定したコードを出力します。そのため，アセンブラ関数から C 言語関数へ分岐する前に，スタック領域中の未使用領域の上位アドレスを指すように SP を設定してください。これは下位アドレスの方向にスタック・フレームが取られるためです。

(2) 作業用レジスタ

CA850 は C 言語関数呼び出しの前後において，レジスタ変数用レジスタの値は保持しますが，作業用レジスタの値は保持しません。そのため，保持しなくてはならない値を作業用レジスタに割り当てたままにしないでください。

レジスタ変数用レジスタ，作業用レジスタは，レジスタ・モードにより異なります。

表 9―3　レジスタ変数用レジスタ

レジスタ・モード	レジスタ変数用レジスタ
22 レジスタ・モード	r25，r26，r27，r28，r29
26 レジスタ・モード	r23，r24，r25，r26，r27，r28，r29
32 レジスタ・モード	r20，r21，r22，r23，r24，r25，r26，r27，r28，r29

表 9―4　作業用レジスタ

レジスタ・モード	作業用レジスタ
22 レジスタ・モード	r10，r11，r12，r13，r14
26 レジスタ・モード	r10，r11，r12，r13，r14，r15，r16
32 レジスタ・モード	r10，r11，r12，r13，r14，r15，r16，r17，r18，r19

(3) アセンブラ関数への戻り先アドレス

CA850 は「関数の戻り先アドレスは“リンク・ポインタ lp（r31）”に格納される」ことを想定したコードを生成します。C 言語関数へ分岐するとき，lp に関数の戻り先アドレスを格納する必要あります。

一般的には jarl 命令によって，C 言語関数へ分岐します。

9.5　他言語で定義された変数の参照

アセンブリ言語で定義した変数を C 言語側で参照する方法を以下に示します。

【C 言語のプログラム例】

```
extern  char    c;
extern  int     i;
void subf(){
    c = 'A';
    i = 4;
}
```

CA850 アセンブラでは，次のように行います。

```
    .globl  _i
    .globl  _c
    .sdata
_i:
    .word   0x0
_c:
    .byte   0x0
```

第 10 章　注意事項

この章では，CA850 を用いる際に注意すべき点について説明します。

10. 1　フォルダ／パスの区切り

“ \ ”と“/”の両方が区切りとみなされます。

10. 2　オプションの指定順序

CA850 は，コマンド・ラインにおけるコマンド起動時に指定したオプションの指定順序について，次の制限があります。

-W オプションで特定のモジュールに渡される引数と，ドライバによって認識されたオプションの引数とで，モジュール起動時に実際に渡される順序は保証されません。**注**

注　CA850 から ld850 を起動する場合，ld850 へは -W オプションで指定しなくても，デフォルトで -lm -lc が渡されます。また，CA850 から ld850 を起動する場合，デフォルトでスタート・アップ・モジュール crtN.o ／ crtE.o が，ld850 へ渡されます。

例

```
> ca850 -cpu 3201 file.o -Wl,-D,dfile.dir
```

ld850 起動時には次のように渡されます。

```
ld850 Install Folder\lib850\r32\crtN.o -o a.out file.o -lm -lc -D dfile.dir
```

ただし，ld850 は Install Folder \ bin に置かれているものとします。

注意　ld850 を直接起動する場合，ライブラリの指定は，数学ライブラリが標準ライブラリを参照するため，“-lc ”は“-lm”の後ろに指定してください。

10. 3　関数宣言／定義における K&R 形式との混在

関数の宣言と定義において，K&R 形式と ANSI 規格形式が混在している場合，K&R 形式における引数拡張処理の結果，CA850 によるコンパイル時にエラーとなる場合があります。

たとえば，次の例では，関数宣言を ANSI 規格で行っていますが，関数定義は K&R 形式で行っているため，引数の型に不整合が生じ，CA850 は“関数の再宣言”エラーを出力します。

【エラーとなる例】

```
void    func(int a, int b, float c);
/*ANSI 規格形式で宣言 */
/* 第 3 引数は float 型として宣言 */
   :
void func(a, b, c)
int     a, b;
float   c;
{
    /*K&R 形式で定義 */
    /* 第 3 引数は，K&R のデフォルトの拡張のため，double 型 */
       :
}
```

この例の場合，関数宣言で“void func ( );”として K&R 形式に統一する，または関数定義で“void func (int a, int b, float c);”として ANSI 規格形式に統一することにより，正常にコンパイルができます。

ただし，CA850 では，ANSI 規格の形式で統一することを推奨しています。

10. 4 ポジション・インディペンデントではないコード出力

CA850 は，基本的に，位置に依存しない（ポジション・インディペンデント）コードを出力します。ただし，「自動変数以外のポインタ型の変数に対する，数値以外の初期値による初期化指示」に対しては，次の例に示すようなコードを出力します。

例

【C 言語の記述】

```
char    *ptr = "test\n";
```

【出力コード】

```
    .size   LL20, 6
LL20:
    .str    "test\n\0"
    .align  4
    .globl  _ptr, 4
_ptr:
    .word   #LL20   -- ラベルの絶対アドレス参照
```

CA850 は，-Xd オプションを指定した場合，自動変数以外のポインタ変数に対する，数値以外の初期値による初期化の指示が現れると，次に示す警告メッセージを出力し，コンパイルを続行します。

```
W2231: Initialization of non-auto pointer using non-number initializer is not position independent.
```

10.5 型の構成における派生型修飾の回数

CA850 は，型の構成において 17 回以上の派生型修飾[注]が行われた場合，次のエラー・メッセージを出力し，コンパイルを続行します。

```
E2260: compiler limit : complicated type modifiers [16]
```

ただし，エラー発生数によってはコンパイルを中止することがあります。

注 宣言子の中に含まれる *（ポインタ），[ ]（配列），および関数宣言子のことを意味します。

10.6 識別子の長さと有効文字数

CA850 は，外部識別子で 1023 文字，内部識別子で 1024 文字以上の長さの識別子が記述された場合，次のエラー・メッセージを出力し，コンパイルを続行します。

```
E2117: compiler limit:too long identifier 'symbol' [1022 / 1023]
```

ただし，エラー発生数によってはコンパイルを中止することがあります。

識別子名における有効な文字は，外部識別子の場合，先頭から 1022 文字まで，内部識別子の場合，先頭から 1023 文字までです。

10.7 ブロックのネスティングの回数

CA850 は，“{”と“}”の組（ブロック）が 128 回以上ネストした形で用いられた場合，次のメッセージを出力します。

```
F2020: compiler limit : scope level too deep [127]
```

10.8 switch 文中の case ラベルの個数

CA850 は，1 つの switch 文中に 1026 個以上の case ラベルが記述された場合，次のエラー・メッセージを出力し，コンパイルを中止します。

```
F2410: compiler limit : too many case labels [1025]
```

ただし，switch 文のネストの数によっては，case ラベルの数が 1025 個に満たなくても，上記のメッセージを出力し，コンパイルを中止することがあります。

10. 9　定数式の演算による浮動小数点演算例外

CA850 は，定数式の演算において浮動小数点演算例外が発生した場合，次のエラー・メッセージを出力し，コンパイルを続行します。

```
E2519: exception has occurred at compile time.
```

ただし，エラー発生数によってはコンパイルを中止することがあります。

なお，exception には例外の種類によって inexact，underflow，overflow，division-by-0，または others のいずれかが出力されます。

10. 10　巨大な／大量のファイルのマージ

CA850 では，最適化レベルに応じて中間言語ファイルのマージを行います。このとき，マージを行うプリオプティマイザ（popt850）はコンパイル処理の高速化のために，メモリ上で処理します。そのため，巨大な中間言語ファイル，または大量の中間言語ファイルをマージする場合，メモリ不足から次のエラー・メッセージを出力し，異常終了することがあります。

```
F7009: out of memory
```

この場合，プリオプティマイザがメモリ消費を抑える処理を行うオプション（-Wp, -D）を指定して，コマンド・ラインで再コンパイルしてください。

10. 11　巨大なファイルの最適化

CA850 では，オブジェクト・サイズ優先最適化，または実行速度優先最適化の場合，広域最適化部（opt850）内部で，データ・フローを関数単位で解析し，広域的な最適化を行います。この最適化はメモリを多く必要とするため，巨大な関数を含むソース・ファイルを最適化する場合，メモリ不足から次のエラー・メッセージを出力し，異常終了することがあります。

```
F5104: out of memory
```

特に，実行速度優先最適化の場合，関数のインライン展開が行われた結果として，関数の大きさが巨大になっている可能性があります。この場合，最適化レベルを低くして，再コンパイルしてください。

10. 12　オプション指定によるライブラリ・ファイル検索

CA850 では，オプション（-L, -l）によるライブラリ・ファイルの検索[注]の結果，指定されたライブラリ・ファイルが存在しなくてもメッセージを表示しません。ただし，ライブラリ・ファイル名をコマンド・ライン，およびコマンド・ファイルで直接指定した場合はフォルダ，メッセージを表示します

注　-L オプションを指定しない場合，標準のフォルダ（*Install Folder* \ lib850，およびその下の各レジスタ・モード・フォルダ）で検索します。

例

```
> ca850 -cpu 3201 a.c usr.a
```

```
F4002: can not open input file"usr.a".
```

10. 13 volatile 修飾子

volatile 修飾子をつけて変数宣言すると，その変数は最適化の対象からはずされ，レジスタに割り付ける最適化などを行わなくなります。volatile 指定された変数に対する操作を行うときは，必ずメモリから値を読み込み，操作後にメモリへ値を書き込むコードになります。また volatile 指定された変数のアクセス幅も変更されません。

volatile 指定されていない変数は，最適化によってレジスタに割り付けられ，その変数をメモリからロードするコードが削除されることがあります。また volatile 指定されていない変数に同じ値を代入する場合，冗長な命令と解釈されて最適化により命令が削除されることもあります。特に周辺 I/O レジスタへアクセスする変数や，割り込み処理で値が変更される変数，また外部から値が変更される変数に対しては，volatile 指定する必要があります。ただし，CA850 では，# pragma ioreg 指令を使って周辺 I/O レジスタにアクセスする場合，内部的に volatile 指定されているコードが出力されるので，改めて volatile 修飾子をつけて宣言する必要ありません。

volatile 指定すべきところで指定されていなかった場合，次の現象が起こることがあります。

- 正しい計算結果が得られない
- ループ内で変数を使っていた場合，ループから抜け出せない

ただし，volatile 指定した変数を使用する際，ある区間でその変数の値が外部から変更されないことが自明な場合，volatile 指定されていない変数に，その値を代入してその変数を参照することにより，その変数が最適化され，実行速度が向上する可能性があります。

【volatile 指定しなかった場合のソースと出力コードの例】

“変数 a ”，“変数 b”，および “変数 c” を volatile 指定しなかった場合，これらの変数がレジスタに割り付けられ，最適化されます。たとえば，この間に割り込みが入り，割り込み内で変数値を変更しても，値が反映されないことになります。

```
int a;
int b;
int c;
void func(void){
    if(a <= 0){
        b++;
    }else{
        c++;
    }
    b++;
    c++;
}
```

```
_func:
    #@B_PROLOGUE
    #@E_PROLOGUE
    ld.w    $_a, r12
    cmp     r0, r12
    jgt     .L2
    ld.w    $_b, r11
    ld.w    $_c, r10
    add     1, r11
    jbr     .L3
.L2:
    ld.w    $_c, r10
    ld.w    $_b, r11
    add     1, r10
.L3:
    addi    1, r11, r13
    st.w    r13, $_b
    addi    1, r10, r14
    st.w    r14, $_c
    #@B_EPILOGUE
    jmp     [lp]
    #@E_EPILOGUE
```

【volatile 指定した場合のソースと出力コードの例】

“変数 a”，“変数 b”，および “変数 c” を volatile 指定した場合，これらの変数値を必ずメモリから読み込み，操作後にメモリへ書き込むコードが出力されます。たとえば，この間に割り込みが入り，割り込み内で変数値が変更されても，その変更が反映された結果を取得することができます（このような例の場合，割り込みのタイミングによっては，変数の操作区間内を割り込み禁止にするなどの処置が必要となります）。

volatile 指定をすると，メモリの読み込み／書き込み処理が入るため，volatile 指定しなかった場合よりもコード・サイズは大きくなります。

```
volatile int    a;
volatile int    b;
volatile int    c;
void func(void){
    if(a <= 0){
        b++;
    }else{
        c++;
    }
    b++;
    c++;
}
```

```
func:
    #@B_PROLOGUE
    #@E_PROLOGUE
    .option volatile
    ld.w    $_a, r10
    .option novolatile
    cmp     r0, r10
    jgt     .L2
    .option volatile
    ld.w    $_b, r11
    .option novolatile
    add     1, r11
    .option volatile
    st.w    r11, $_b
    .option novolatile
    jbr     .L3
.L2:
    .option volatile
    ld.w    $_c, r12
    .option novolatile
    add     1, r12
    .option volatile
    st.w    r12, $_c
    .option novolatile
.L3:
    .option volatile
    ld.w    $_b, r13
    .option novolatile
    add     1, r13
    .option volatile
    st.w    r13, $_b
    .option novolatile
    .option volatile
    ld.w    $_c, r14
    .option novolatile
    add     1, r14
    .option volatile
    st.w    r14, $_c
    .option novolatile
    #@B_EPILOGUE
    jmp     [lp]
    #@E_EPILOGUE
```

10. 14 関数宣言での余計なかっこ

関数宣言で，かっこ“( )”が余計に記述されている場合，ANSI-C では次のように規定されていますが，CA850 では，エラーとなります。

例

```
typedef int Int;
void    f1((Int));
```

【ANSI-C での規定】

仮引数宣言内では，かっこで囲まれた型定義名は，単一の仮引数をもつ関数を指定する抽象宣言子と解釈する。宣言子の識別子を囲む冗長なかっこは解釈しない。

したがって，上記の例では，ANSI-C では次のように解釈されます。

```
void    f(int (*)(int));
```

かっこを余計に記述してしまった場合には，次のように不必要なかっこを削除してください。

例

```
typedef int Int;
void    f1(Int);
```

付録 A　エディタ

この付録では，テキスト・ファイル／ソース・ファイルの表示／編集を行うエディタ パネルについて説明します。

エディタ パネル

テキスト・ファイル／ソース・ファイルの表示／編集を行います。

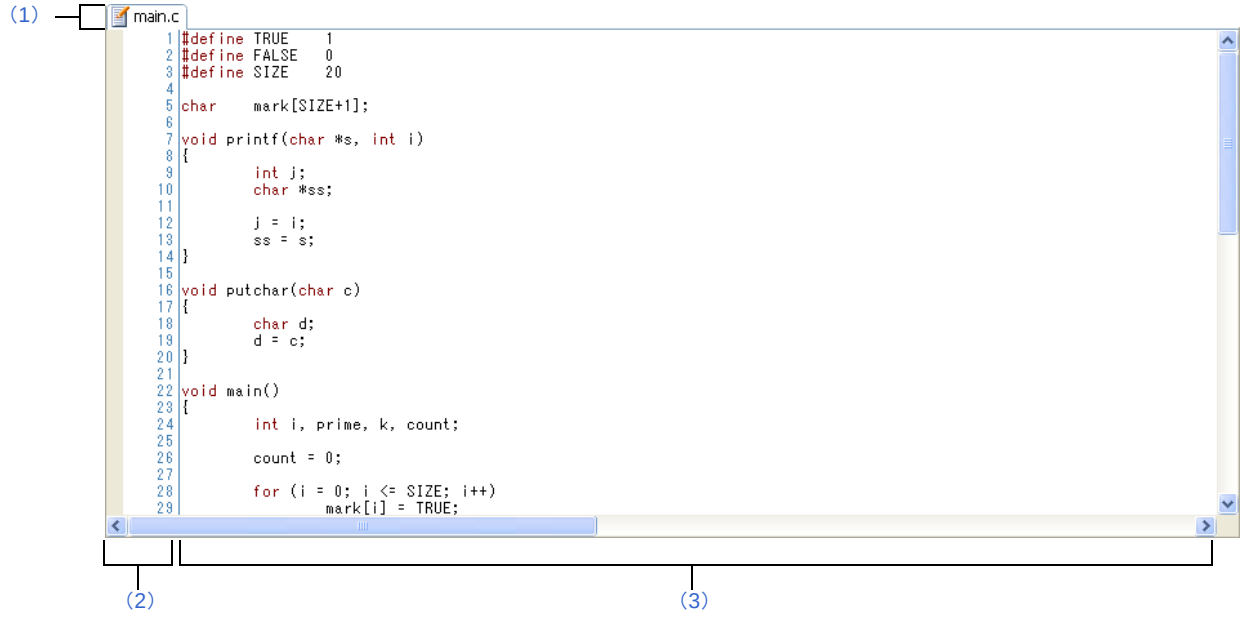
自動的にファイルのエンコード（Shift_JIS/EUC-JP/UTF-8）と改行コードを判別してオープンし，保存の際は元のエンコードと改行コードで保存します。

ただし，ファイルの保存設定 ダイアログでエンコードと改行コードを指定した場合は，それに従って保存します。

本パネルは複数オープンすることができます（最大個数：100 個）。

備考　ソース・ファイルをオープンする際，ダウンロードしているロード・モジュールの更新日時よりオープンするソース・ファイルの更新日時が新しい場合，メッセージを表示します。

図 A―1　エディタ パネル



ここでは，次の項目について説明します。

- ［オープン方法］
- ［各エリアの説明］
- ［［ファイル］メニュー（エディタ パネル専用部分）］
- ［［編集］メニュー（エディタ パネル専用部分）］
- ［コンテキスト・メニュー］

［オープン方法］

- プロジェクト・ツリー パネルにおいて，ファイルをダブルクリック
- プロジェクト・ツリー パネルにおいて，ソース・ファイルを選択したのち，コンテキスト・メニューの［開く］を選択
- プロジェクト・ツリー パネルにおいて，ファイルを選択したのち，コンテキスト・メニューの［内部エディタで開く ...］を選択
- プロジェクト・ツリー パネルにおいて，コンテキスト・メニューの［追加］→［新しいファイルを追加］を選択したのち，テキスト・ファイル／ソース・ファイルを作成

［各エリアの説明］

（1）タイトルバー

オープンしているテキスト・ファイル／ソース・ファイルのファイル名を表示します。

なお，ファイル名の末尾に表示するマークの意味は次のとおりです。

マーク	意味
*	編集中のテキスト・ファイルの内容を編集している場合に表示します。
(編集不可)	書き込み禁止状態のテキスト・ファイルをオープンしている場合に表示します。
ID 番号	同一のテキスト・ファイルを複数オープンしている場合に表示します。

（2）行番号エリア

表示しているテキスト・ファイル／ソース・ファイルの行番号を表示します。

（3）文字列エリア

テキスト・ファイル／ソース・ファイルの文字列の表示／編集を行います。

本エリアは，次の機能を備えています。

（a）文字列の編集

キーボードより，IME などの日本語入力システムを使用した文字列を入力することができます。

また，編集機能を充実させるための様々なショートカットキーを使用することができます。

（b）ファイルの監視機能

ソース・ファイルを管理するために，次の監視機能を備えています。

- CubeSuite+ 以外によって，現在表示しているファイルの内容を変更していた場合，ファイルを保存するかどうかのメッセージを表示し，どちらかを選択することができます。

備考　オプション ダイアログの設定により，次の項目をカスタマイズすることができます。

- 表示フォント
- タブ幅
- コントロール・キャラクタ（空白記号を含む制御コード）の表示／非表示／色分け

- 予約語／コメントの色分け

[[ファイル]メニュー（エディタ パネル専用部分）]

エディタ パネル専用の［ファイル］メニューは次のとおりです（その他の項目は共通です）。

*ファイル名*を閉じる	現在編集しているエディタ パネルをクローズします。 なお，パネルの内容を保存していない場合は，確認メッセージを表示します。
*ファイル名*を保存	現在編集しているエディタ パネルの内容を上書き保存します。 なお，ファイルを一度も保存していない，またはファイルが書き込み禁止の場合は，［名前を付けて *ファイル名*を保存 ...］の選択と同等の動作となります。
*ファイル名*の保存設定 ...	エディタ パネルで編集中のファイルのエンコードと改行コードを設定するために，ファイルの保存設定 ダイアログをオープンします。
名前を付けて *ファイル名*を保存 ...	現在編集しているエディタ パネルの内容を新規保存するために，名前を付けて保存 ダイアログをオープンします。
ページ設定 ...	Windows の印刷用ページ設定 を行うダイアログをオープンします。
印刷 ...	現在編集しているエディタ パネルの内容を印刷するために，Windows の印刷用 ダイアログをオープンします。

[[編集]メニュー（エディタ パネル専用部分）]

エディタ パネル専用の［編集］メニューは次のとおりです（その他の項目はすべて無効となります）。

元に戻す	エディタ パネル上で前回行った操作をキャンセルし，文字とキャレット位置を元に戻します（最大 100 回まで）。
やり直し	エディタ パネル上で前回行った［元に戻す］の操作をキャンセルし，文字とキャレット位置を元に戻します。
切り取り	選択範囲の文字列を切り取り，クリップ・ボードにコピーします。
コピー	選択範囲の文字列をクリップ・ボードにコピーします。
貼り付け	クリップ・ボードにコピーしている文字列をキャレット位置に，挿入モードの場合は挿入し，上書きモードの場合は上書きします。 ただし，クリップ・ボードの内容を文字列として認識できない場合は無効となります。
削除	キャレット位置の文字を 1 文字削除します。 ただし，範囲選択している場合は，選択しているの文字列を削除します。
すべて選択	現在編集中のテキストの先頭から最終までを選択状態にします。
検索 ...	検索・置換 ダイアログを［クイック検索］タブが選択状態でオープンします。 範囲選択している場合は，選択している範囲内での検索モードとなります。
置換 ...	検索・置換 ダイアログを［クイック置換］タブが選択状態でオープンします。 範囲選択している場合は，選択している範囲内での置換モードとなります。
移動 ...	指定した行へキャレットを移動するため，指定位置へ移動 ダイアログをオープンします。

［コンテキスト・メニュー］

【文字列エリア／行番号エリア】

関数へジャンプ	選択している範囲の文字列，またはキャレット位置の単語を関数名と判断して，その関数へジャンプします。 ただし，シンボル情報が存在するファイルをダウンロード対象として指定している場合のみ機能します。 なお，本メニューは，プロジェクトがアクティブ・プロジェクトの場合，およびライブラリ用のプロジェクト以外の場合のみ有効です。
ジャンプ前の位置へ戻る	キャレット位置がジャンプする前の位置へ戻ります。
ジャンプ先の位置へ進む	［ジャンプ前の位置へ戻る］を実行する前の位置へ進みます。
タグ・ジャンプ	キャレット行のメッセージに対応するエディタ（ファイル，行，桁）へジャンプします。
切り取り	選択している範囲の文字列を切り取ってクリップ・ボードに移動します。
コピー	選択している範囲の文字列をクリップ・ボードにコピーします。
貼り付け	クリップ・ボードの内容をキャレット位置に挿入します。
新しく開く	現在フォーカスのあるエディタ パネルと同じ内容で，新規にエディタ パネルをオープンします（新規にオープンしたエディタ パネルのタイトルバー上には，ファイル名とともに ID 番号を表示します）。 なお，エディタ パネルは最大 100 個までオープンすることができます。

付録B 索 引

【T】

改訂記録

Rev.	発行日	改訂内容	
		ページ	ポイント
1.00	2011.04.01	―	初版発行

CubeSuite+ V1.00.00 ユーザーズマニュアル
V850 コーディング編

発行年月日　2011 年 4 月 1 日　Rev.1.00
発行　ルネサス エレクトロニクス株式会社
〒211-8668 神奈川県川崎市中原区下沼部 1753

ルネサス エレクトロニクス株式会社

■営業お問合せ窓口

http://www.renesas.com

※営業お問合せ窓口の住所・電話番号は変更になることがあります。最新情報につきましては、弊社ホームページをご覧ください。

ルネサス エレクトロニクス販売株式会社　〒100-0004　千代田区大手町2-6-2（日本ビル）　(03)5201-5307

■技術的なお問合せおよび資料のご請求は下記へどうぞ。
総合お問合せ窓口：http://japan.renesas.com/inquiry

CubeSuite+ V1.00.00

ルネサス エレクトロニクス株式会社

R20UT0553JJ0100