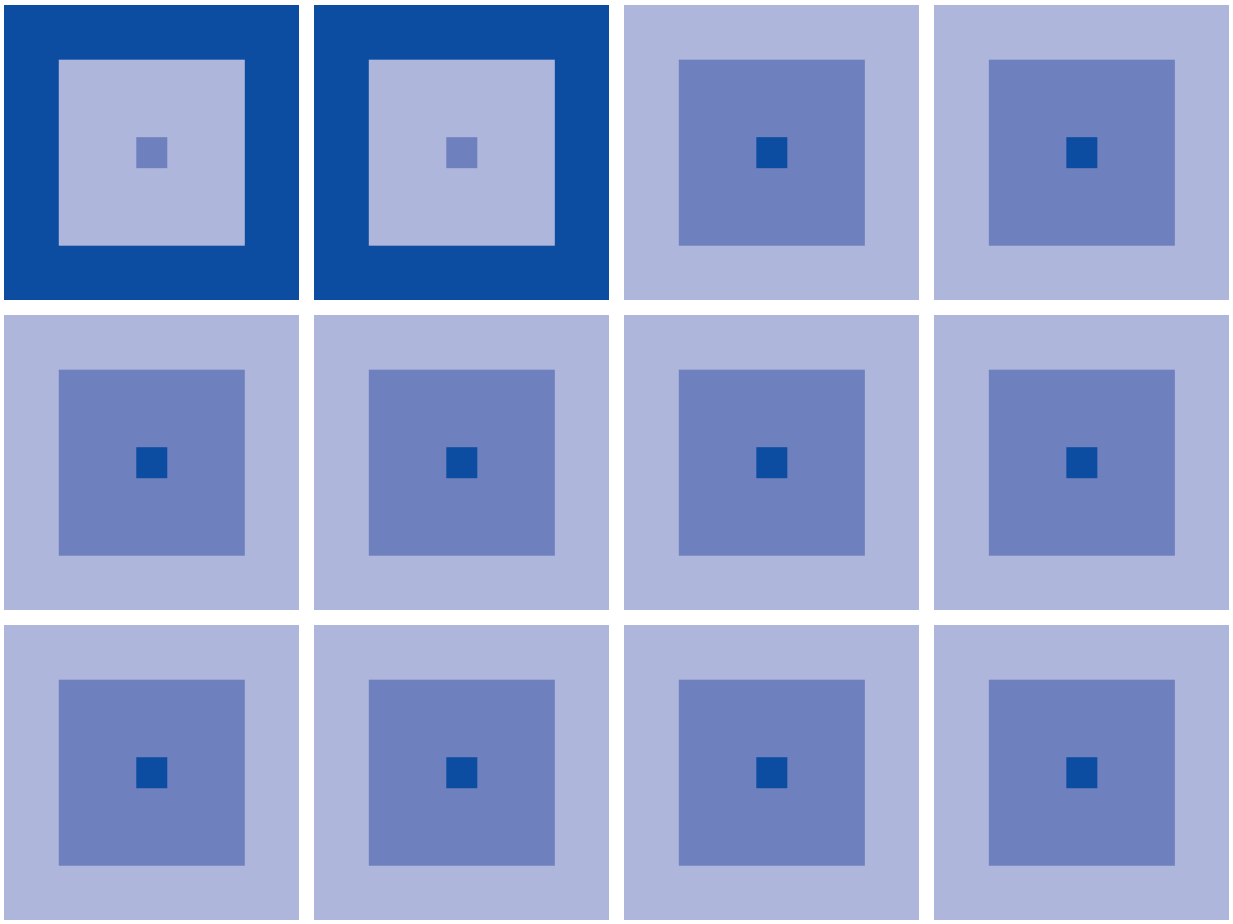
EPSON

CMOS 32-BIT SINGLE CHIP MICROCOMPUTER

S1C33 Family C33 ADV

コアCPUマニュアル



セイコーエプソン株式会社

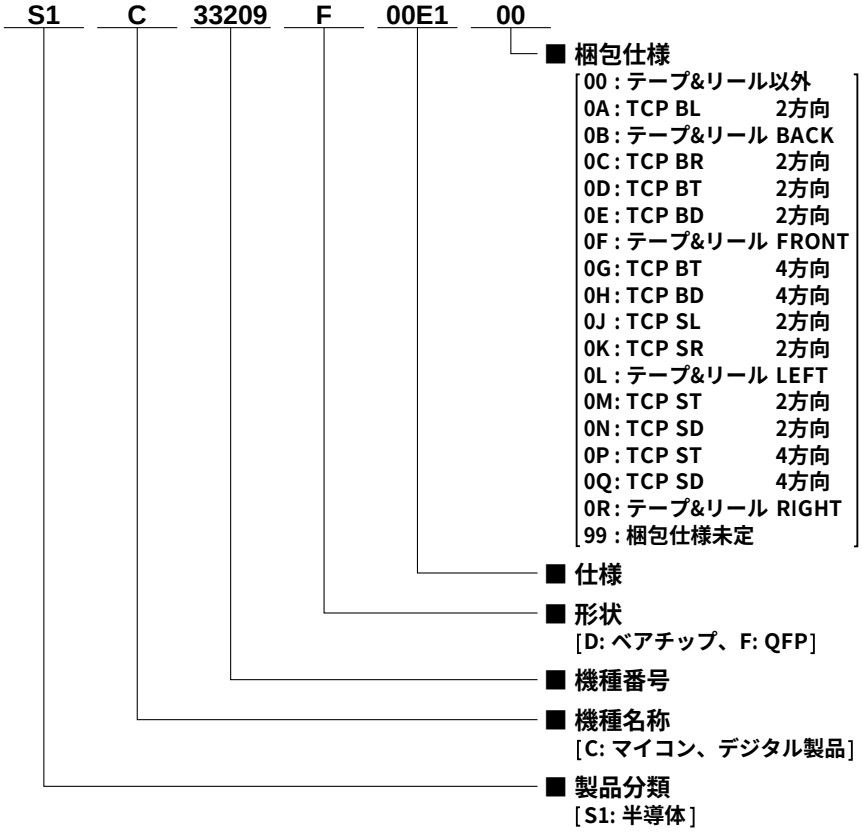
本資料のご使用につきましては、次の点にご留意願います。

1. 本資料の内容については、予告なく変更することがあります。
2. 本資料の一部、または全部を弊社に無断で転載、または、複製など他の目的に使用することは堅くお断りします。
3. 本資料に掲載される応用回路、プログラム、使用方法等はあくまでも参考情報であり、これらに起因する第三者の権利(工業所有権を含む)侵害あるいは損害の発生に対し、弊社は如何なる保証を行うものではありません。また、本資料によって第三者または弊社の工業所有権の実施権の許諾を行うものではありません。
4. 特性表の数値の大小は、数直線上の大小関係で表しています。
5. 本資料に掲載されている製品のうち、「外国為替および外国貿易法」に定める戦略物資に該当するものについては、輸出する場合、同法に基づく輸出許可が必要です。
6. 本資料に掲載されている製品は、一般民生用です。生命維持装置その他、きわめて高い信頼性が要求される用途を前提としていません。よって、弊社は本(当該)製品をこれらの用途に用いた場合の如何なる責任についても負いかねます。

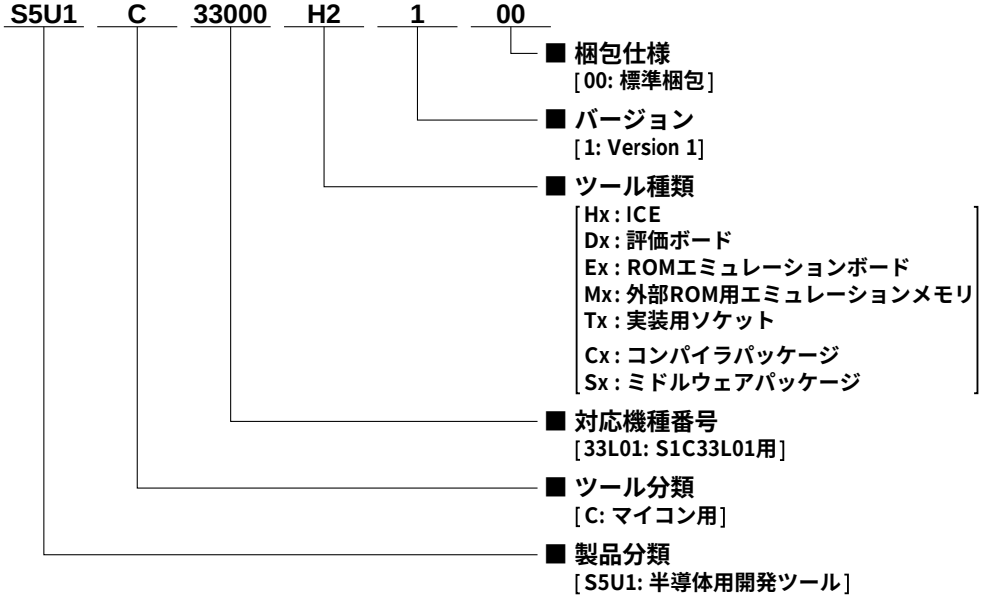
© SEIKO EPSON CORPORATION 2004

製品型番体系

●デバイス



●開発ツール



－ 目 次 －

1 概要

C33 ADV CPU はC33 STD CPUの上位に位置する32ビットRISC CPUで、機能拡張された命令セットを持ち、低消費電力、高速動作を特長としています。C33 ADV CPU は、入力命令キューと5段パイプライン処理を採用し、無駄の少ないCPUアーキテクチャを提供します。S1C33 STD CPUではオプションの乗算器と積和命令を標準でサポートしており、新命令と併せてマルチメディア関連の処理が容易に実現可能となりました。また、MMU(メモリマネージメントユニット)とCCU(キャッシュコントロールユニット)に対応し、より高速でかつ高度な処理を行うことが可能です。
オブジェクトコードはS1C33 STDの上位互換となっていますので、過去の資産を有効に活用することができます。

1.1 特長

プロセッサ形式
- セイコーエプソンオリジナル32ビットRISC CPU
- 32ビット長の内部データ処理
- 32ビット×16ビットの乗算器を内蔵

動作周波数
- DC～66MHz以上(機種により異なります。)

命令セット
- マルチメディア対応の命令セット
- コード長　16ビット固定長
- 命令数　164命令
- 実行サイクル　主要命令は1サイクルで実行
  即値拡張命令を含む2～3命令も1サイクルで実行可能
- 即値拡張命令　即値を32ビットまで拡張

マルチメディア対応機能
- 乗算命令　16×16、32×16、32×32ビットの乗算をサポート
- 積和命令　16×16、32×16、32×32ビットのステップ/連続積和演算をサポート
- ループ命令　範囲指定の繰り返し実行
- リピート命令　1命令の繰り返し実行
- 飽和命令　最小値/最大値への丸め処理
- ポストシフト　ポストシフト付きALU命令実行機能をサポート

レジスタセット
- 32ビット汎用レジスタ
- 32ビットの特殊レジスタ
- 32ビットの積和演算用レジスタ

メモリ空間, 外部バス
- 命令、データ、I/O混在のリニア空間
- 最大4Gバイトのメモリ空間
- リトルエンディアン形式(ソフト切り換えによりビッグエンディアンの設定が可能)

割り込み
- リセット、NMI、128種類の外部割り込みに対応
- 4種類のソフトウェア例外
- 2種類の命令実行例外
- ベクタテーブルからベクタを読み込み処理ルーチンへ直接分岐
- MMU例外に対応

リセット
- コールドリセット(すべてをリセット)
- ホットリセット(バスとポートの状態を保持)

パワーダウンモード
- HALTモード(CPUコアのみ停止)
- SLEEPモード(CPUコアと発振回路を停止)

その他
- MMUのサポート
- CCUのサポート

1.2 C33 ADV追加/変更機能のまとめ

C33 ADVコアCPUでは、C33 STDコアCPU(S1C33000)の機能をもとに以下の追加と変更が行われています。個々の機能の詳細については、後述のそれぞれの説明を参照してください。

1.2.1 命令

C33 ADVコアCPUの命令セットはC33 STDコアCPUの上位互換となっていますが、高性能化のため、以下に示すとおり既存命令の機能拡張と新規命令の追加を行っています。

●機能拡張命令

C33 ADVコアCPUでは以下の命令の機能が拡張されています。詳細は各命令の説明を参照してください。

1. シフト/ローテート命令のシフト量が8ビットから32ビットに拡張

```
shift   %rd,imm5*  0～8ビットシフト → 0～32ビットシフト(shift = srl, sll, sra, sla, rr, rl)
shift   %rd,%rs    0～8ビットシフト → 0～32ビットシフト(shift = srl, sll, sra, sla, rr, rl)
```

   * "shift %rd,imm5"命令は、実際の命令コードを2つ使用していますが、前節の命令数はそれぞれを1個として計算しています。

2. ポストインクリメント命令([%rb]+)がextによるアドレス拡張に対応

```
ext     offset
ld.t    [%rb]+,%rs    ext命令を使用(t = b, h, w)
ext     offset
ld.t    %rd,[%rb]+    ext命令を使用(t = b, ub, h, uh, w)
```

3. 汎用レジスタ - 特殊レジスタ間データ転送命令が、追加された特殊レジスタの指定に対応

```
ld.w    %sd,%rs       %sdで指定可能な特殊レジスタを追加
ld.w    %rd,%ss       %ssで指定可能な特殊レジスタを追加
```

4. スキャン命令の対応ビット数を8ビットから32ビットに拡張

```
scan0   %rd,%rs       スキャンビット数を32ビットに拡張
scan1   %rd,%rs       スキャンビット数を32ビットに拡張
```

●追加命令

C33 ADVコアCPUには以下の命令が追加されています。詳細は各命令の説明を参照してください。

1. ターゲット命令の3オペランド化、フラグ条件による実行/非実行制御、演算命令のポストシフト(最大3ビット)に対応するext命令を追加

```
ext     %rs           3オペランドに拡張
ext     cond          条件付き実行
ext     op,imm2       ポストシフト
ext     %rs,op,imm2   3オペランド拡張 + ポストシフト
```

2. 32ビット×16ビット乗算器の標準実装に対応し、乗算および積和命令を強化
   - 16ビット×16ビット、32ビット×16ビット、32ビット×32ビットをサポート
   - 積和処理にレジスタベースのステップ処理命令を追加
   - 算術演算レジスタの初期化命令を追加

```
mlt.hw  %rd,%rs       乗算、32ビット×16ビット → 48ビット
mac.hw  %rs           積和演算、32ビット×16ビット + 64ビット → 64ビット
mac.w   %rs           積和演算、32ビット×32ビット + 64ビット → 64ビット
mac1.h  %rd,%rs       ステップ積和演算、16ビット×16ビット + 64ビット → 64ビット
mac1.hw %rd,%rs       ステップ積和演算、32ビット×16ビット + 64ビット → 64ビット
mac1.w  %rd,%rs       ステップ積和演算、32ビット×32ビット + 64ビット → 64ビット
macclr                AHRとALRレジスタおよびMOフラグのクリア
```

3. 高速な繰り返し処理を実現するループ、リピート命令を追加
   分岐の必要がなく、高速な連続実行が可能です。

loop	*%rc,%ra*	指定範囲のループ
loop	*%rc,imm4*	指定範囲のループ
loop	*imm4,imm4*	指定範囲のループ
repeat	*%rb*	繰り返し
repeat	*imm4*	繰り返し

4. DP相対アドレスのメモリアクセス命令を追加
   データエリアの先頭アドレスをDPレジスタに設定することで、相対アドレスによる少ない命令数でのアクセスが可能になります。

ld.*t*	*%rd*,[%dp+*imm6*]	DPレジスタ間接ロード(*t* = b, ub, h, uh, w)
ld.*t*	[%dp+*imm6*],*%rs*	DPレジスタ間接ストア(*t* = b, h, w)
add	*%rd*,%dp	加算、DPレジスタがオペランドとして追加

5. 単独レジスタ、特殊レジスタに対応したレジスタ退避/復帰命令を追加

push	*%rs*	単一レジスタの退避
pop	*%rd*	単一レジスタの復帰
pushs	*%ss*	連続する特殊レジスタの退避
pops	*%sd*	連続する特殊レジスタの復帰

6. コプロセッサインタフェースに対応する命令を追加

ld.c	*%rd,imm4*	コプロセッサデータ転送
ld.c	*imm4,%rs*	コプロセッサデータ転送
do.c	*imm6*	コプロセッサ実行
ld.cf		コプロセッサフラグ転送

7. その他、特殊命令を追加

sat.*t*	*%rd*	飽和(*t* = b, ub, h, uh, w, uw)
div.w		符号付き除算、32ビット / 32ビット → 16ビット ... 16ビット
divu.w		符号なし除算、32ビット / 32ビット → 16ビット ... 16ビット
swaph	*%rd,%rs*	ビッグエンディアン - リトルエンディアン変換
psrset	*imm5*	PSRビットのセット
psrclr	*imm5*	PSRビットのクリア
jpr	*%rb*	レジスタ間接無条件相対ジャンプ
retm		MMU例外処理ルーチンからのリターン

注: 既存命令においても、特殊な使い方をしている場合やPSRレジスタの設定によっては完全な互換性を保証できませんので注意してください。

1.2.2 レジスタ

汎用レジスタ(R0～R15)は基本的にC33 STDコアCPUと同一です。
特殊レジスタについては、以下のように大幅な拡張が行われています。

●PC

32ビットすべてが使用されるようになりました。
また、高速なリーフコールを行えるように、読み出しも可能になりました。

●スタックポインタ

スタックポインタとしてSSPとUSPが追加されました。
C33 ADVコアCPUはスーパバイザモード、ユーザモードの2つの動作モードを持ち、それぞれで使用するスタックポインタを分けています。SSPがスーパバイザモード用、USPがユーザモード用です。既存のSPは物理的なレジスタではなくなり、“%sp”のオペランドを持つ命令によるSPへのアクセスは現在のモードに従ってSSPまたはUSPに対して行われます。

●データポインタ

少ない命令数で効率的にメモリをアクセスするために、データポインタDPが追加されました。
データエリアの先頭アドレスをDPに設定し、既存の“SP + オフセット”を指定する命令と同等の“DP + オフセット”という形でアドレスを指定して、メモリデータのロード/ストアを行います。これにより、ext命令の使用量を削減できます。

●トラップテーブルベースレジスタ

トラップテーブルベースレジスタTTBRが追加されました。
C33 STDコアCPUはTTBRが0x48134番地にマップされ、初期値は0xC00000でした。C33 ADVコアCPUではTTBRがCPU内の特殊レジスタとなり、初期値(ブートアドレス)も0x20000000に変更されています。

●シフトアウトレジスタ

シフトアウトレジスタSORが追加されました。
このレジスタには、シフト命令の実行によって指定レジスタからシフトアウトしたビットが入ります。
浮動小数点処理や画像処理など、32ビット以上のシフトを行いたいときに有効です。

●ループ関連レジスタ

ループ、リピート命令に対応するため、ループスタートアドレスレジスタLSA、ループエンドアドレスレジスタLEA、ループカウンタLCOの3つのレジスタが追加されました。
各レジスタはループ、リピート実行するアドレス(範囲)と実行回数を保持します。

●算術演算レジスタ

ALRとAHR自体はC33 STDコアCPUと同一です。
C33 ADVコアCPUではPSRの設定により、ALRとAHRに書き込まれる乗算、除算、積和の演算結果がR4(= ALR)とR5(= AHR)にも書き込まれるように指定できます(R4とR5を介したALRとAHRへのアクセスは不可)。これにより、乗除算や積和演算の結果を直接データ転送命令や算術演算で参照可能となります。

●CPU識別レジスタ

CPU識別レジスタIDIRが追加され、コアのタイプ、バージョンの識別が可能になりました。

●デバッグベースレジスタ

デバッグベースレジスタDBBRが追加されました。このレジスタはデバッグエリアの先頭アドレスを示します。通常は0x60000に固定されています。

●プロセッサステータスレジスタ

プロセッサステータスレジスタPSRには以下のビットが追加されました。

HE	ユーザモードでのhalt、slp命令の使用を許可
RM	リピート命令実行中か否かを表示
LM	ループ命令実行中か否かを表示
PM	連続したプッシュ/ポップ実行中か否かを表示
RC	連続プッシュ/ポップ実行中のレジスタ番号を保持
SW	8ビット、32ビットスキャンの切り換え
OC	論理演算でのVフラグ処理の切り換え
SE	シフト演算でのC、Vフラグ処理の切り換え
LC	ALRのR4へのマップを選択
HC	AHRのR5へのマップを選択
S	飽和命令実行時に飽和が発生したか否かを表示
DE	デバッグ例外の発生状態を表示
ME	MMU例外の発生状態を表示
SV	スーパバイザモード/ユーザモードの選択

1.2.3 アドレス空間、モード、その他

●アドレス空間

32ビットアドレスバスによる4Gバイト空間をサポートします。
HBCUとMMUのサポートにより、CPUがアクセスする空間と実際のメモリなどのデバイスがマップされている空間を、それぞれ「論理アドレス空間」、「物理アドレス空間」という2つの異なる空間として扱えるようになりました。
論理アドレス空間(4GB)はHBCUにより0.5Gバイト単位の8ブロックに分割され、ブロックごとにMMU、ASIDに関する設定が可能です。
MMUを使用するブロックでは、4KBまたは64KB単位に論理アドレスを物理アドレスに変換します。
またASIDを使用して64プロセス×64MBの多重仮想空間も実現可能です。
物理アドレス空間は#CE信号により22エリアに分けられ、メモリなどのデバイスを実際に配置できます。

●スーパバイザモード/ユーザモード

すべてのリソースにアクセスできるスーパバイザモードと、アクセスできるリソースが制限されるユーザモードが設けられました。

ユーザモードの制限事項

1. メモリアクセスの制限
   MMUの設定により、スーパバイザ空間など、特定のページへのユーザモードでのアクセスを禁止できます。
   ASIDを使用する場合、1つのプロセス(ASID)は64KB領域を越えて他のプロセスをアクセスすることはできません。
   ユーザモード時に全メモリ空間でMMUとASIDが強制的に使用されるようにHBCUを設定することも可能です。

2. レジスタアクセスの制限
   一部の特殊レジスタはスーパバイザモードでのみアクセス可能です。

3. 命令実行の制限
   PSRの設定により、ユーザモードからの`slp`および`halt`命令の実行を禁止できます。

4. 割り込みモードの制限
   例外(通常の割り込み、MMU例外、デバッグ例外など)発生時は、すべてスーパバイザモードに移行するため、ユーザモードで直接例外を管理することはできません。

●その他

1. MMU対応
   C33 ADVコアCPUはMMU経由の命令フェッチやデータアクセスで発生するMMU例外に対応しています。また、MMU例外から戻る`retm`命令が追加されました。

2. 割り込み処理
   トラップテーブルベースレジスタTTBRがCPU内の特殊レジスタとなり、コールドリセット時のブートアドレスも0x20000000に変わりました。通常の割り込み、ソフトウェア割り込み、例外処理の内容はC33 STDコアCPUと同様ですが、例外発生時は必ずスーパバイザモードに移行します。
   NMIは、CPU内でNMI例外の多重発生を禁止しています。NMI要求信号のトリガモードもソフトウェアで選択可能になりました。

3. パイプライン
   C33 STDコアCPUは3段パイプライン構造でしたが、C33 ADVコアCPUでは、より高い動作周波数による実行に対応するため5段パイプライン構造(フェッチ、デコード、実行、アクセス、ライトの5段)になっています。
   また、`ext`命令はターゲットとなる通常命令と並列処理を行い、それぞれの拡張命令について、`ext`命令を2個まで0クロック(ターゲット命令のクロック数のみ)で実行可能となっています。

2 レジスタ

C33 ADVコアCPUは、16本の汎用レジスタおよび15本の特殊レジスタを内蔵しています。

特殊レジスタ

	ビット31 … ビット0
#15	PC
#14	SSP
#13	USP
#11	DBBR
#10	IDIR
#9	DP
#8	TTBR
#7	SOR
#6	LEA
#5	LSA
#4	LCO
#3	AHR
#2	ALR
#1	SP
#0	PSR

汎用レジスタ

	ビット31 … ビット0
#15	R15
#14	R14
#13	R13
#12	R12
#11	R11
#10	R10
#9	R9
#8	R8
#7	R7
#6	R6
#5	R5 (AHR)
#4	R4 (ALR)
#3	R3
#2	R2
#1	R1
#0	R0

図2.1 レジスタの構成

表2.1 レジスタのアクセス権

レジスタシンボル	名称	スーパバイザモード	ユーザモード
PC	プログラムカウンタ	R	R
SSP *1	スーパバイザスタックポインタ	R/W	R
USP *1	ユーザスタックポインタ	R/W	R/W
DBBR *1	デバッグベースレジスタ	R	R
IDIR *1	CPU識別レジスタ	R	R
DP *1	データポインタ	R/W	R/W
TTBR *1	トラップテーブルベースレジスタ	R/W	R
SOR *1	シフトアウトレジスタ	R/W	R/W
LEA *1	ループエンドアドレスレジスタ	R/W	R/W
LSA *1	ループスタートアドレスレジスタ	R/W	R/W
LCO *1	ループカウントレジスタ	R/W	R/W
AHR	算術演算上位レジスタ	R/W	R/W
ALR	算術演算下位レジスタ	R/W	R/W
SP *2	スタックポインタ	R/W	R/W
PSR	プロセッサステータスレジスタ	R/W	R/W *3

*1 C33 ADVコアCPUで新規に追加されたレジスタです。

*2 SPレジスタを参照すると、モードによってSSPまたはUSPが参照されます。

*3 PSRの一部のビットは、ユーザモードでの書き込みはできません。

2.1 汎用レジスタ(R0～R15)

シンボル	レジスタ名	サイズ	スーパバイザモード	ユーザモード	初期値
R0~R15	**汎用レジスタ**	**32ビット**	R/W	R/W	**不定**

16本のR0～R15レジスタは、データの演算、データの転送、メモリのアドレッシング等、使用目的が固定されていない32ビット長の汎用レジスタです。これらのレジスタの内容は、すべて32ビットのデータまたはアドレスとして扱われ、8ビット、16ビットデータを扱う命令ではレジスタへのロード時に符号拡張またはゼロ拡張されます。C33 ADVコアCPUでレジスタをアドレス参照に用いる場合は、ひとつのレジスタで32ビットの空間を直接アクセスすることができます。
パワーオンイニシャルリセット時の、汎用レジスタの内容は不定です。

2.2 プログラムカウンタ(PC)

シンボル	レジスタ名	サイズ	スーパバイザモード	ユーザモード	初期値
PC	**プログラムカウンタ**	**32ビット**	R	R	**不定**

プログラムカウンタ(以下、PC)は、実行命令のアドレスを保持する32ビット長のカウンタです。
PCの値は、次に実行されるアドレスを示しています。C33 ADVコアCPUの命令は16ビット固定長のため、PCの下位1ビット(ビット0)は常に0となります。C33 ADVコアCPUではPCをプログラムで参照することが可能です。ただし、データ転送命令で変更することはできません。なお、`ld.w %rd,%pc`命令を実行した場合、指定レジスタには「この`ld`命令のPC値+2」がロードされます。
リセット時には、リセットベクタ(TTBRで示されるベクタテーブルの先頭)に書き込まれているアドレスがPCにロードされ、そのアドレスからプログラムが実行されます。コールドリセット時はTTBRが"0x20000000"に初期設定されますので、0x20000000番地に書き込まれているアドレスがプログラムの実行開始アドレスとなります。

図2.2.1 プログラムカウンタ(PC)

2.3 プロセッサステータスレジスタ(PSR)

シンボル	レジスタ名	サイズ	スーパバイザモード	ユーザモード	初期値
PSR	**プロセッサステータスレジスタ**	**32ビット**	R/W *1	R/W *1	0x00000000

***1 PSRの一部のビットは書き込みができません。ただし、この書き込み制限はスーパバイザモードとユーザモードで異なります。**

プロセッサステータスレジスタ(以下、PSR)は、CPU内部の状態を示す32ビット長のレジスタです。
PSRには命令の実行によって変化した内部ステータス情報が格納されます。これらの内部ステータスは算術演算や分岐命令などで参照され、プログラムを構成する上で重要な情報として用いられます。プログラムによってPSRの変更が可能ですが、図2.3.1に示すとおり一部書き込みのできないビットがあります。
PSRはプログラムの実行に影響を与えるため、割り込みや例外が発生したときには(MMU例外とデバッグ例外を除く)、PSRをスタックに退避して内容を保存します。また、RM(ビット30)、LM(ビット29)、PM(ビット28)、IE(ビット4)の各ビットは0にクリアされます。割り込み処理からは`reti`命令で復帰します。`reti`命令はPCを割り込み/例外が発生した位置に戻すと同時に、PSRの値もスタックからレジスタに戻します。

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	HE	RM	LM	PM	RC[3:0]				–	SW	OC	SE	–	–	LC	HC
初期値	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
スーパバイザモード	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W	R/W	R/W	R	R	R/W	R/W
ユーザモード	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W	R/W	R/W	R	R	R/W	R/W

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	S	DE	ME	SV	IL[3:0]				MO	DS	–	IE	C	V	Z	N
初期値	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
スーパバイザモード	R/W	R	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W
ユーザモード	R/W	R	R	R	R	R	R	R	R/W	R/W	R	R	R/W	R/W	R/W	R/W

図2.3.1 プロセッサステータスレジスタ(PSR)

図中の"–"は未使用です。書き込みは無効で読み出し値は常に0となります。

HE(ビット31): Halt and Sleep Enable

このフラグが1の場合、ユーザモードでもhalt、slp命令が実行可能です。スーパバイザモードではこのフラグの影響は受けません。このフラグはユーザモードで変更することはできません。

RM(ビット30): Repeat Mode Enable

repeat命令が実行されると1となり、このフラグがセットされている間、CPUはrepeat命令に続く命令を連続実行します。repeat命令を実行中に割り込みや例外を受け付けると、CPUはPSRをスタックに退避させた後、このフラグを0にクリアします。ただし、MMU例外およびデバッグ例外ではPSRはスタックには退避されず、このフラグもクリアされません。そのため、アプリケーションの例外処理ルーチンの中でこのフラグを保護する必要があります。

LM(ビット29): Loop Mode Enable

loop命令が実行されると1となり、このフラグがセットされている間、CPUはLSAレジスタ内のアドレスからLEAレジスタ内のアドレスまでの命令をLCOレジスタで指定された回数分繰り返し実行します。loop命令を実行中に割り込みや例外を受け付けると、CPUはPSRをスタックに退避させた後、このフラグを0にクリアします。ただし、MMU例外およびデバッグ例外ではPSRはスタックには退避されず、このフラグもクリアされません。そのため、アプリケーションの例外処理ルーチンの中でこのフラグを保護する必要があります。

PM(ビット28): Push/Pop Mode

pushn、popn、pushs、またはpops命令が実行されると1となり、最後のレジスタが退避または復帰されるまで1を保持します。また、このフラグが1のときにpushn、popn、pushs、またはpops命令が実行されると、退避または復帰動作をRC[3:0](ビット[27:24])で指定されたレジスタから開始し、最後のレジスタまで動作を続けます。pushn、popn、pushs、またはpops命令を実行中に割り込みや例外を受け付けると、CPUはPSRをスタックに退避させた後、このフラグを0にクリアします。ただし、MMU例外およびデバッグ例外ではPSRはスタックには退避されず、このフラグもクリアされません。そのため、アプリケーションの例外処理ルーチンの中でこのフラグを保護する必要があります。

RC[3:0](ビット[27:24]): Register Counter

pushn、popn、pushs、またはpops命令実行中に割り込みまたは例外が発生すると、その時点で退避または復帰中のレジスタ番号が格納されます。PM(ビット28)が1のときにpushn、popn、pushs、またはpops命令が実行されると、ここに格納されたレジスタから、退避または復帰動作が再開されます。

SW(ビット22): Scan Word Enable

このフラグが0のときにscan命令を実行すると8ビットのビットスキャンを行い、1のときは32ビットのビットスキャンを行います。

OC(ビット21): Overflow Clear Enable

このフラグが1のときに論理演算命令が実行されると、PSRのVフラグ(ビット2)をクリアします。

SE(ビット20): Shift with Carry Enable

このフラグが1のとき、シフト関係の命令でシフトアウトしたビットはPSRのCフラグ(ビット3)へ格納されます。また、シフトの結果によってVフラグも変化するようになります。

LC(ビット17): ALR Change Enable

このフラグが1のとき、以下の対象命令によりALRレジスタに書き込まれるデータはR4にも書き込まれます。対象命令はHCフラグ(ビット16)のものと同一です。

```
対象命令: mlt.h  %rd,%rs       mltu.h  %rd,%rs
          mlt.hw %rd,%rs
          mlt.w  %rd,%rs       mltu.w  %rd,%rs
          div0s  %rs           div0u   %rs
          div1   %rs           div2s   %rs          div3s
          div.w  %rs           divu.w  %rs
          mac    %rs           mac.hw  %rs          mac.w  %rs
          mac1.h %rd,%rs       mac1.hw %rd,%rs      mac1.w %rd,%rs
          macclr
```

HC(ビット16): AHR Change Enable

このフラグが1のとき、以下の対象命令によりAHRレジスタに書き込まれるデータはR5にも書き込まれます。対象命令はLCフラグ(ビット17)のものと同一です。

```
対象命令: mlt.h  %rd,%rs       mltu.h  %rd,%rs
          mlt.hw %rd,%rs
          mlt.w  %rd,%rs       mltu.w  %rd,%rs
          div0s  %rs           div0u   %rs
          div1   %rs           div2s   %rs          div3s
          div.w  %rs           divu.w  %rs
          mac    %rs           mac.hw  %rs          mac.w  %rs
          mac1.h %rd,%rs       mac1.hw %rd,%rs      mac1.w %rd,%rs
          macclr
```

S(ビット15): Saturation

飽和命令の実行時に飽和が発生すると1にセットされます。一度セットされたSフラグは、プログラムでクリアしない限り1を保持します。

DE(ビット14): Debug Exception

デバッグ例外が発生すると1になります。DEフラグは`retd`命令の実行によって0にクリアされます。このフラグは読み出しのみ可能です。

ME(ビット13): MMU Exception

MMU例外が発生すると1になります。MEフラグがセットされると以後の例外は禁止されます。MMUは停止状態になり、MMU例外処理中にアクセスされるアドレスはすべて物理アドレスとなります。
MEフラグは`retm`命令の実行によって0にクリアされます。このフラグは読み出しのみ可能です。

SV(ビット12): SuperVisor Mode

割り込みや例外が発生するとCPUはスーパバイザモードとなり、SVフラグが0になります。
スーパバイザモードではすべてのリソースにアクセス可能です。SVフラグが1のとき、CPUはユーザモードとなり、一部レジスタへの書き込みが制限されます。このフラグは、ユーザモードで変更することはできません。ただし、MMU例外処理中はSVフラグの値にかかわらず、常にスーパバイザモードになります。

IL[3:0](ビット[11:8]) Interrupt Level

CPUの割り込みレベルを示します。マスク可能な割り込み要求は、その割り込みレベルがILビットフィールドに設定されたレベルより高い場合にのみ受け付けられます。また、1つの割り込みを受け付けるとILビットフィールドがその割り込みレベルに設定され、それ以降はILビットフィールドを再設定するか、割り込み処理ルーチンを`reti`命令で終了するまで、同じレベルの割り込み要求が再度発生してもマスクされます。このフラグは、ユーザモードで変更することはできません。

MO(ビット7): Mac Overflow

積和演算によるオーバーフローを示します。積和演算実行中に途中結果が符号付き64ビットの有効範囲を超えると1にセットされます。演算は、オーバーフローにかかわらず最後まで実行されますので、演算終了後にMOフラグを読み出して結果が有効かどうかを判断します。MOフラグは一旦セットされるとイニシャルリセットかプログラムによって明示的にリセットされるまで1を保持します。

DS(ビット6): Divide Sign

ステップ除算の実行時に被除数の符号ビットがDSフラグにセットされ、除算の実行を制御します。

注: **`div0s`または`div0u`命令実行直後に`ld.w`命令でPSRを読み出した場合、DSフラグが正しく読み出せないことがあります。DSフラグを正しく読み出すには、`div0s`または`div0u`命令と`ld.w`命令の間に2つ以上の命令を挿入してください。**

IE(ビット4): Interrupt Enable

マスク可能な割り込みを受け付けるか禁止するかを制御します。IEビットが1のとき、CPUはマスク可能な割り込みを許可します。IEビットが0のときはマスク可能な割り込みを禁止します。このフラグは、ユーザモードで変更することはできません。割り込みや例外を受け付けると、CPUはPSRをスタックに退避させた後、このフラグを0にクリアします。ただし、MMU例外およびデバッグ例外ではPSRはスタックには退避されず、このフラグもクリアされません。

C(ビット3): Carry

キャリーまたはボローを示します。加算命令または減算命令において演算結果を符号なし32ビット整数として扱う場合に、命令の実行結果が符号なし32ビット整数の範囲を超えると1にセットされます。
結果が符号なし32ビット整数の範囲内の場合は0にクリアされます。
Cフラグがセットされる条件は以下のとおりです。

(1) 加算命令で、演算結果が符号なし32ビット整数の最大値0xFFFFFFFFよりも大きい値となる加算を実行した場合

(2) 減算命令で、演算結果が符号なし32ビット整数の最小値0x00000000よりも小さい値となる減算を実行した場合

V(ビット2): OVerflow

オーバーフローまたはアンダーフローが発生したことを示します。加算命令または減算命令において演算結果を符号付き32ビット整数として扱う場合に、命令の実行によりオーバーフローまたはアンダーフローが発生すると1にセットされます。加算または減算結果が符号付き32ビット範囲内の場合は0にクリアされます。
Vフラグがセットされる条件は以下のとおりです。

(1) 負の整数と負の整数を加算した場合に、結果の符号ビット(最上位ビット)が0(正)になった場合

(2) 正の整数と正の整数を加算した場合に、結果の符号ビット(最上位ビット)が1(負)になった場合

(3) 正の整数から負の整数を減算した場合に、結果の符号ビット(最上位ビット)が1(負)になった場合

(4) 負の整数から正の整数を減算した場合に、結果の符号ビット(最上位ビット)が0(正)になった場合

Z(ビット1): Zero

結果が0であることを示します。論理演算、算術演算、シフト命令の実行結果がゼロの場合、1にセットされ、ゼロ以外のときは0にクリアされます。

N(ビット0): Negative

符号を示します。論理演算、算術演算、シフト命令の実行結果の最上位ビット(ビット31)がNフラグにコピーされます。ステップ除算の実行時には除数の符号ビットがNフラグにセットされ、除算の実行を制御します。

2.4 スタックポインタ(SP)

シンボル	レジスタ名	サイズ	スーパバイザモード	ユーザモード	初期値
SP	**スタックポインタ**	**32ビット**	R/W (SSP)	R/W (USP)	**不定**
SSP	**スーパバイザスタックポインタ**	**32ビット**	R/W	R	**不定**
USP	**ユーザスタックポインタ**	**32ビット**	R/W	R/W	**不定**

スタックポインタ(以下、SP)は、スタックの先頭アドレスを保持する32ビットレジスタです。
SPにはスーパバイザスタックポインタ(以下、SSP)とユーザスタックポインタ(以下、USP)の2つがあり、命令に"SP"を指定すると、そのときのモードがスーパバイザモードかユーザモードかによって自動的にSSPまたはUSPが参照されます。
CPUをユーザモードにして使用する場合は、SSPを参照することはできません。
スーパバイザモードでは、SSPとUSPのどちらも参照できますが、pushn命令やpopn命令で"SP"というレジスタ記号から間接参照する場合は、常にSSPが参照されます。
スタックはシステムのRAM上に任意に配置可能な領域で、初期設定でSPにスタックの先頭アドレスをセットします。また、SPの下位2ビットは0固定となり書き込みは行えません。したがって、SPで指定できるアドレスはワード境界となります。

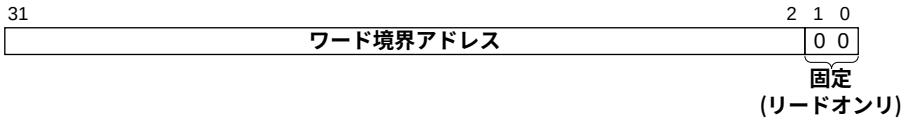


図2.4.1 スタックポインタ(SP)

2.4.1 スタック領域について

スタックとして使用可能な領域サイズは、RAMのサイズと通常のRAMデータが占有する領域サイズによって制限されます。両者が重複しないように注意が必要です。
また、SPはイニシャルリセットにより不定となりますので、初期化ルーチン内の先頭部分でアドレス(スタック最終アドレス+4、下位2ビットは0)を書き込んでください。アドレスの書き込みはロード命令で行えます。スタック設定前に割り込みや例外が発生するとPCやPSRが不定の位置にセーブされ、プログラムの正常な動作が保証できません。このためソフトウェア制御が不可能なNMIは、SPが初期化されるまでハードウェアによってマスクされるようになっています。

2.4.2 push系命令実行時の動作

push系の命令では、まずSPの示すポインタから4を引きSPを低位アドレスに移動します。

SP = SP - 4

次に、命令で指定されたレジスタをSPの示すアドレスにストアします。

rs → [SP]

例: `pushn %r2`

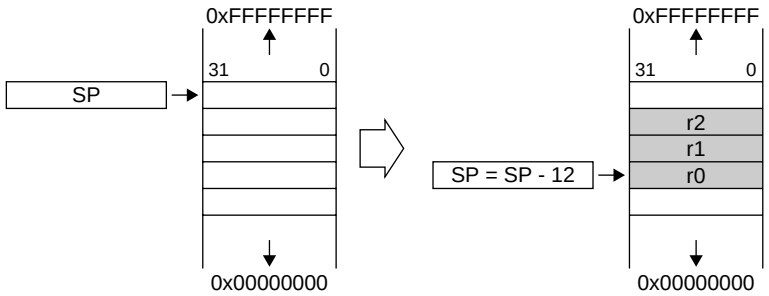


図2.4.2.1 SPとスタック(1)

2.4.3 pop系命令実行時の動作

pop系の命令では、まずSPの示すアドレスからレジスタにデータを復帰します。

[SP] → *rs*

次に、SPの内容に4を加えて高位アドレスへポインタを移動します。

SP = SP + 4

例: `popn  %r2`

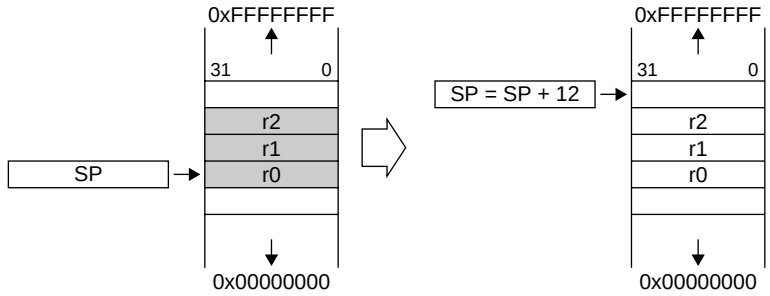


図2.4.3.1 SPとスタック(2)

2.4.4 call命令実行時の動作

サブルーチンコール命令`call`はスタックを1ワード(32ビット)使用します。`call`命令はサブルーチンに分岐する前にPCの内容(リターンアドレス)をスタックにセーブします。セーブされたアドレスはサブルーチンの最後に`ret`命令によってPCに戻され、プログラムは`call`命令の次のアドレスに戻ります。

`call`命令の動作

(1) SP = SP - 4
(2) PC → [SP]

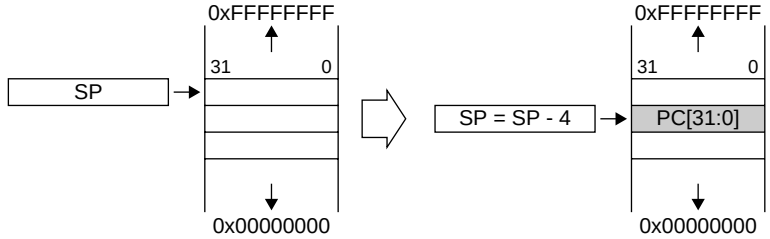


図2.4.4.1 SPとスタック(3)

`ret`命令の動作

(1) [SP] → PC
(2) SP = SP + 4

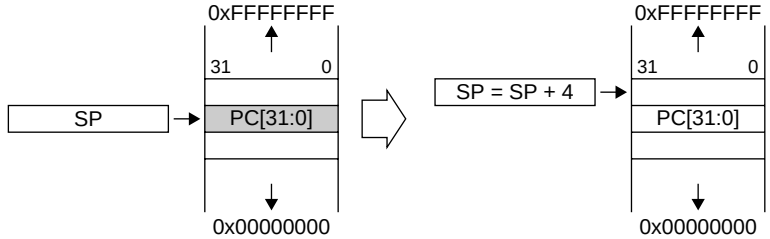


図2.4.4.2 SPとスタック(4)

2.4.5 割り込みまたは例外発生時

割り込み、int命令によるソフトウェア例外等が発生すると、CPUは動作モードをスーパバイザモードにして例外処理に入ります。以降、スタックポインタはSSPが対象となります。
CPUはそれぞれの割り込み処理ルーチンに分岐する前にPCとPSRの内容をSSPの示すスタックにセーブします。これは割り込みや例外によって変更されるこの2つのレジスタの内容を保護するためです。PCとPSRのデータは図2.4.5.1のようにスタックにセーブされます。
処理ルーチンからのリターンにはPCとPSRの内容を復帰するreti命令を使用します。reti命令では通常のpop動作とは異なり、PC、PSRのセーブ内容が図2.4.5.2の順に読み出され、SSPの内容が変更されます。

割り込み発生時の動作

(1) SSP = SSP - 4
(2) PC → [SSP]
(3) SSP = SSP - 4
(4) PSR → [SSP]

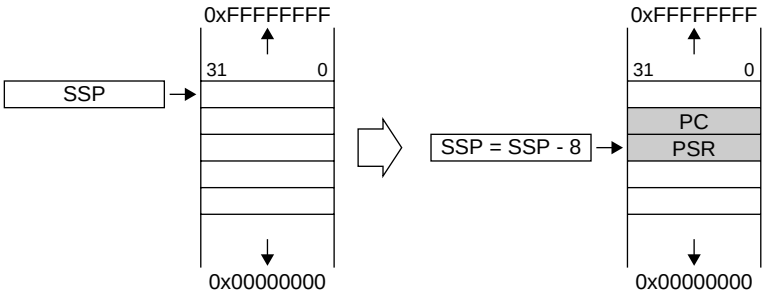


図2.4.5.1 SPとスタック(5)

reti命令実行の動作

(1) [SSP+4] → PC
(2) [SSP] → PSR
(3) SSP = SSP + 8

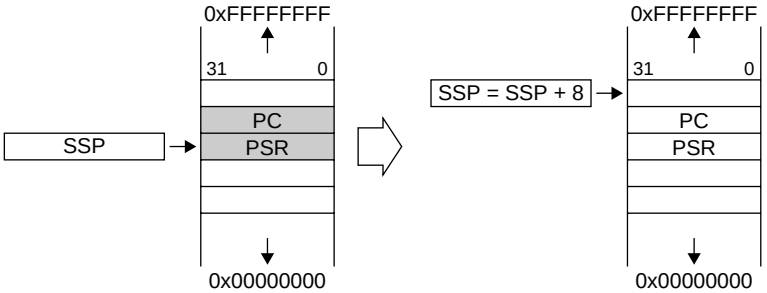


図2.4.5.2 SPとスタック(6)

2.5 データポインタ(DP)

シンボル	レジスタ名	サイズ	スーパバイザモード	ユーザモード	初期値
DP	データポインタ	32ビット	R/W	R/W	不定

データポインタ(以下、DP)は、ロード命令でディスプレースメント付きレジスタ間接アドレッシングを行う場合に、メモリアドレスを格納するレジスタとして使用できる32ビット長のレジスタです。即値拡張命令と組み合わせることで最大「DP + 32ビットの即値」のメモリ空間をサポートします。ただし、DPに対する算術演算はサポートされません。また、DPの下位2ビットは0固定となり書き込みは行えません。したがって、DPで指定できるアドレスはワード境界となります。

```
ld.b   %rd,[%dp+imm6]
ld.ub  %rd,[%dp+imm6]
ld.h   %rd,[%dp+imm6]
ld.uh  %rd,[%dp+imm6]
ld.w   %rd,[%dp+imm6]

ld.b   [%dp+imm6],%rs
ld.h   [%dp+imm6],%rs
ld.w   [%dp+imm6],%rs
```

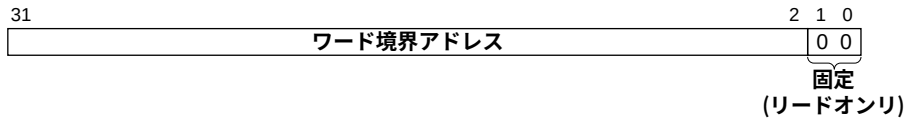


図2.5.1 データポインタ(DP)

2.6 トラップテーブルベースレジスタ(TTBR)

シンボル	レジスタ名	サイズ	スーパバイザモード	ユーザモード	初期値
TTBR	トラップテーブルベースレジスタ	32ビット	R/W	R/W	0x20000000

トラップテーブルベースレジスタ(以下、TTBR)は、割り込みや例外が発生したときに参照するベクタテーブルの先頭アドレスを格納する32ビット長のレジスタです。コールドリセット時には“0x20000000”に初期化され、リセットベクタで示されるアドレスからプログラムの実行を開始します。
TTBRはリード/ライト可能なレジスタで、ソフトウェアによって任意のアドレスに設定することができます。ただし、TTBRのビット9～ビット0は0固定となり書き込みは行えません。したがって、ベクタテーブルを設定できるアドレスは1Kバイト境界ごとになります。また、ユーザモードではTTBRを変更することはできません。

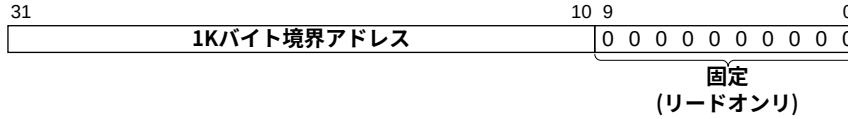


図2.6.1 トラップテーブルベースレジスタ(TTBR)

2.7 シフトアウトレジスタ(SOR)

シンボル	レジスタ名	サイズ	スーパバイザモード	ユーザモード	初期値
SOR	シフトアウトレジスタ	32ビット	R/W	R/W	不定

シフトアウトレジスタ(以下、SOR)は、シフト/ローテート命令の実行時に汎用レジスタからシフトアウトしたビットを格納する32ビット長のレジスタです。最後に実行したシフト/ローテート命令のシフトアウト結果を保持しています。

2.8 ループエンドアドレスレジスタ(LEA)

シンボル	レジスタ名	サイズ	スーパバイザモード	ユーザモード	初期値
LEA	**ループエンドアドレスレジスタ**	32**ビット**	R/W	R/W	**不定**

ループエンドアドレスレジスタ(以下、LEA)は、loop命令によって実行する範囲の最終アドレスを格納する32ビット長のレジスタです。loop命令の実行によってPSRのLMフラグが1になると、それ以降命令フェッチアドレスが常にLEAと比較され、一致すると命令実行後にLSAの示すアドレスへ自動的に分岐します。また、このときLCOがデクリメント(-1)されます。LCOが0になった場合はLSAへの分岐を中止してloop命令を終了します。ここでPSRのLMフラグが0となり、LEAの次の命令の実行に移ります。
LCOが0の状態でloop命令が実行されてもLSAの示すアドレスへは分岐しません。
LEAのビット0は常に0として扱われます。

2.9 ループスタートアドレスレジスタ(LSA)

シンボル	レジスタ名	サイズ	スーパバイザモード	ユーザモード	初期値
LSA	**ループスタートアドレスレジスタ**	32**ビット**	R/W	R/W	**不定**

ループスタートアドレスレジスタ(以下、LSA)は、loop命令によって実行する範囲の先頭アドレスを格納する32ビット長のレジスタです。詳細については、“2.8 ループエンドアドレスレジスタ(LEA)”を参照してください。
また、リピート命令実行時は、LSAの内容がリピート対象アドレスとして使用されます。
LSAのビット0は常に0として扱われます。

2.10 ループカウントレジスタ(LCO)

シンボル	レジスタ名	サイズ	スーパバイザモード	ユーザモード	初期値
LCO	**ループカウントレジスタ**	32**ビット**	R/W	R/W	**不定**

ループカウントレジスタ(以下、LCO)は、loop命令によるLSAとLEA間の実行のループ回数を格納する32ビット長のレジスタです。詳細については、“2.8 ループエンドアドレスレジスタ(LEA)”を参照してください。
また、リピート命令実行時は、LCOの内容がリピート回数として使用されます。

2.11 算術演算レジスタ(ALR, AHR)

シンボル	レジスタ名	サイズ	スーパバイザモード	ユーザモード	初期値
ALR	**算術演算下位レジスタ**	32**ビット**	R/W	R/W	**不定**
AHR	**算術演算上位レジスタ**	32**ビット**	R/W	R/W	**不定**

算術演算下位レジスタ(以下、ALR)と算術演算上位レジスタ(以下、AHR)は、乗除算および積和演算に使用します。それぞれ32ビット長のレジスタで、ロード命令によって汎用レジスタとのデータ転送が行えます。算術演算および積和演算命令はALRに演算結果の下位32ビットを、AHRに演算結果の上位32ビットを格納します。除算命令はALRに商を、AHRに剰余を格納します。イニシャルリセット時、ALRおよびAHRは不定となります。

PSRのLCフラグ(ビット17)、HCフラグ(ビット16)に1をセットすることで、ALRとAHRに書き込まれる乗算、除算、積和の演算結果がR4とR5にも書き込まれるように指定できます。

LC = 1: ALR内の演算結果がR4にも書き込まれる
HC = 1: AHR内の演算結果がR5にも書き込まれる

これにより、乗除算や積和演算の結果を直接データ転送命令や算術演算で参照可能となります。

注: この機能は、乗除算や積和演算の実行終了時に、ALRとAHRにロードされる演算結果をR4とR5にも書き込むものです。LCおよびHCフラグを1に設定しても、R4とR5を介してALRとAHRにアクセスすることはできません。

2.12 CPU識別レジスタ(IDIR)

シンボル	レジスタ名	サイズ	スーパバイザモード	ユーザモード	初期値
IDIR	**CPU識別レジスタ**	**32ビット**	R	R	0x04*XXXXXX*

CPU識別レジスタ(以下、IDIR)は、CPUの種類や改訂コードが格納されている32ビット長のレジスタです。IDIRは読み出し専用で、読み出し値は機種ごとに異なります。

IDIRのビット構成は次のとおりです。

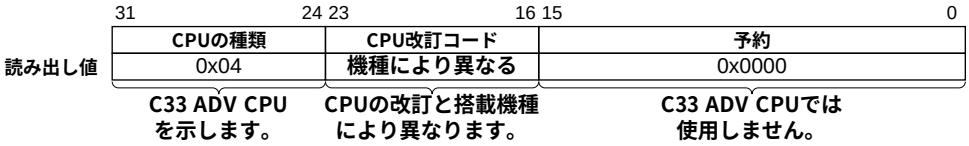


図2.12.1 CPU識別レジスタ(IDIR)

2.13 デバッグベースレジスタ(DBBR)

シンボル	レジスタ名	サイズ	スーパバイザモード	ユーザモード	初期値
DBBR	**デバッグベースレジスタ**	**32ビット**	R	R	0x00060000

デバッグベースレジスタ(以下、DBBR)は、デバッグ用メモリのベースアドレスが格納されている32ビット長のレジスタです。DBBRは読み出し専用のレジスタで、C33 ADV CPUでは0x00060000に固定されています。

2.14 レジスタの表記とレジスタ番号

ここでは、C33 ADVコアCPU命令セットのレジスタ表記とレジスタ番号について説明します。
命令コード中ではレジスタの指定に4ビットのフィールドを使用しており、このフィールドにレジスタ番号が入ります。なお、ニーモニックのレジスタ指定においては、レジスタ名の前に“%”を記述します。

2.14.1 汎用レジスタ

%rs *rs*は演算や転送のソースデータを保持している汎用レジスタを示すメタシンボルです。実際には%r0、%r1、・・・%r15のように記述します。

%rd *rd*はデスティネーション(演算結果が格納される、あるいはデータがロードされる)となる汎用レジスタを示すメタシンボルです。実際には%r0、%r1、・・・%r15のように記述します。

%rb *rb*はアクセスするメモリのベースアドレスを保持している汎用レジスタを示すメタシンボルです。この場合の汎用レジスタはインデックスレジスタとして機能します。
実際の表記は、レジスタ間接アドレッシングを示す[]で囲み、[%r0]、[%r1]、・・・[%r15]のように記述します。レジスタ間接アドレッシングでは、連続したメモリアドレスをアクセスするためのポストインクリメント機能を使用することができます。その場合は[%r0]+のように“+”を後置きします。ポストインクリメントを指定するとメモリアクセスのあとにベースアドレスがアクセスしたサイズに従ってインクリメントされます。
*rb*はcall命令やjp命令の分岐アドレスを格納しているレジスタを示すシンボルとしても使用します。この場合は[]が不要で、%r0、%r1、・・・%r15のように記述します。

命令コード中のレジスタを指定するビットフィールドにはレジスタ番号に対応するコードが入ります。レジスタとレジスタ番号の対応は以下のとおりです。

表2.14.1.1 汎用レジスタ

汎用レジスタ	レジスタ番号	レジスタ表記
R0	0	%r0
R1	1	%r1
R2	2	%r2
R3	3	%r3
R4	4	%r4
R5	5	%r5
R6	6	%r6
R7	7	%r7
R8	8	%r8
R9	9	%r9
R10	10	%r10
R11	11	%r11
R12	12	%r12
R13	13	%r13
R14	14	%r14
R15	15	%r15

2.14.2 特殊レジスタ

%ss *ss*は汎用レジスタに転送するソースデータを保持している特殊レジスタを示すメタシンボルです。特殊レジスタをソースとする命令は次のとおりです。

```
ld.w  %rd,%ss
```

%sd *sd*は汎用レジスタからデータをロードする特殊レジスタを示すメタシンボルです。特殊レジスタを転送先とする命令は次のとおりです。

```
ld.w  %sd,%rs
```

命令コード中のレジスタを指定するビットフィールドにはレジスタ番号に対応するコードが入ります。レジスタとレジスタ番号の対応は以下のとおりです。

表2.14.2.1 特殊レジスタ

特殊レジスタ	レジスタ番号	レジスタ表記
PSR	0	%psr
SP	1	%sp
ALR	2	%alr
AHR	3	%ahr
LCO *	4	%lco
LSA *	5	%lsa
LEA *	6	%lea
SOR *	7	%sor
TTBR *	8	%ttbr
DP *	9	%dp
IDIR *	10	%idir
DBBR *	11	%dbbr
–	(12)	–
USP *	13	%usp
SSP *	14	%ssp
PC	15	%pc

表中の*印は、C33 ADVコアCPUで新規に追加されたレジスタを示します。

3 データ形式

C33 ADVコアCPUは、8ビット長、16ビット長、32ビット長のデータを扱うことができます。
本書では、データのサイズを次のように表します。

8ビット　バイト, Byte, B, b
16ビット　ハーフワード, Halfword, H, h
32ビット　ワード, Word, W, w

データサイズは、メモリと汎用レジスタ間、および汎用レジスタ間のデータ転送(ロード命令)においてのみ選択可能です。
CPUの内部処理はすべて32ビットで行われますので、汎用レジスタへの16ビットデータ転送および8ビットのデータ転送では、レジスタにロードする際に32ビットへの符号拡張またはゼロ拡張が行われます。符号拡張またはゼロ拡張のどちらが行われるかについては、使用するロード命令によって決まります。
汎用レジスタからの16ビットデータ転送または8ビットデータ転送では、転送元レジスタの下位ハーフワードまたは下位1バイトが転送データとなります。

メモリはバイト、ハーフワード、ワード単位にリトルエンディアン形式またはビッグエンディアン形式でアクセスされます。
なお、ハーフワード単位およびワード単位のアクセスは、指定するベースアドレスがそれぞれハーフワード境界(アドレスの最下位ビットが0)、ワード境界(アドレスの下位2ビットが00)であることが必要で、この条件を満たしていないアクセスに対してはアドレス不整例外が発行されます。

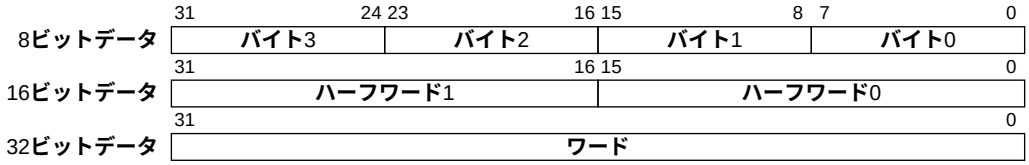


図3.1 リトルエンディアン形式

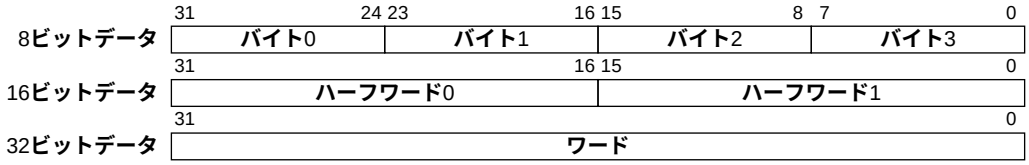


図3.2 ビッグエンディアン形式

データ転送のサイズと種類は以下に示すとおりです。

3.1 符号なし8ビット転送(レジスタ→レジスタ)

例: `ld.ub` *`%rd,%rs`*

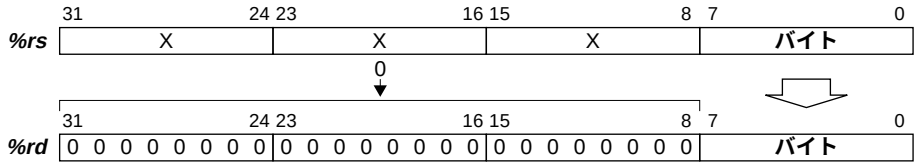


図3.1.1 符号なし8ビット転送(レジスタ→レジスタ)

転送先レジスタのビット31～8はゼロ拡張されます。

3.2 符号付き8ビット転送(レジスタ→レジスタ)

例: ld.b %rd,%rs

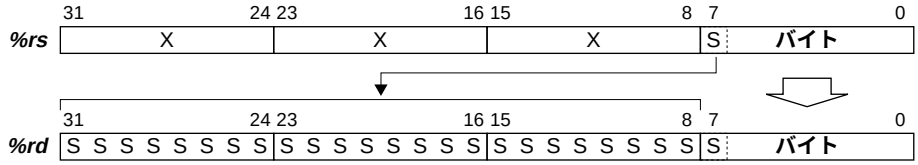


図3.2.1 符号付き8ビット転送(レジスタ→レジスタ)

転送先レジスタのビット31～8は符号拡張されます。

3.3 符号なし8ビット転送(メモリ→レジスタ)

例: ld.ub %rd,[%rb]

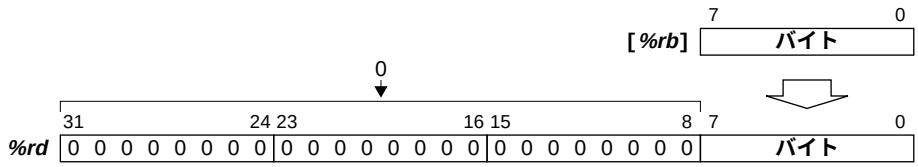


図3.3.1 符号なし8ビット転送(メモリ→レジスタ)

転送先レジスタのビット31～8はゼロ拡張されます。

3.4 符号付き8ビット転送(メモリ→レジスタ)

例: ld.b %rd,[%rb]

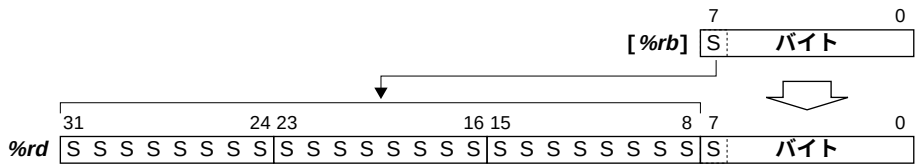


図3.4.1 符号付き8ビット転送(メモリ→レジスタ)

転送先レジスタのビット31～8は符号拡張されます。

3.5 8ビット転送(レジスタ→メモリ)

例: ld.b [%rb],%rs

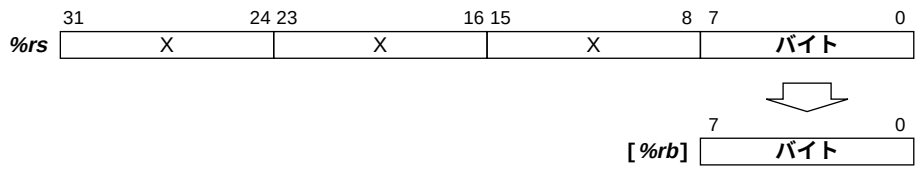


図3.5.1 8ビット転送(レジスタ→メモリ)

3.6 符号なし16ビット転送(レジスタ→レジスタ)

例: ld.uh %rd,%rs

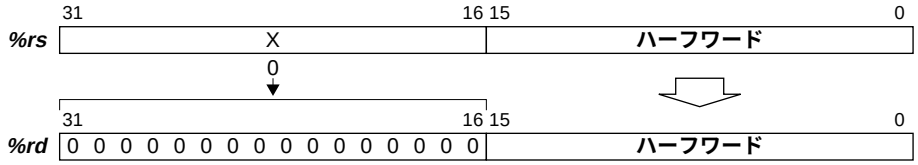


図3.6.1 符号なし16ビット転送(レジスタ→レジスタ)

転送先レジスタのビット31～16はゼロ拡張されます。

3.7 符号付き16ビット転送(レジスタ→レジスタ)

例: ld.h %rd,%rs

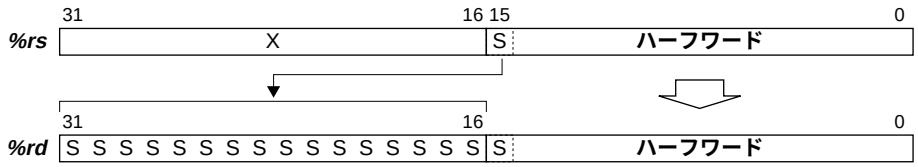


図3.7.1 符号付き16ビット転送(レジスタ→レジスタ)

転送先レジスタのビット31～16は符号拡張されます。

3.8 符号なし16ビット転送(メモリ→レジスタ)

例: ld.uh %rd,[%rb] (リトルエンディアンの場合)

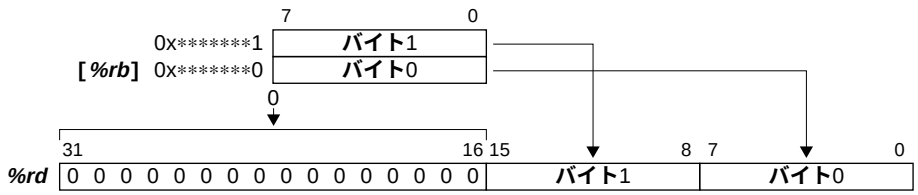


図3.8.1 符号なし16ビット転送(メモリ→レジスタ)

転送先レジスタのビット31～16はゼロ拡張されます。

3.9 符号付き16ビット転送(メモリ→レジスタ)

例: ld.h %rd,[%rb] (リトルエンディアンの場合)

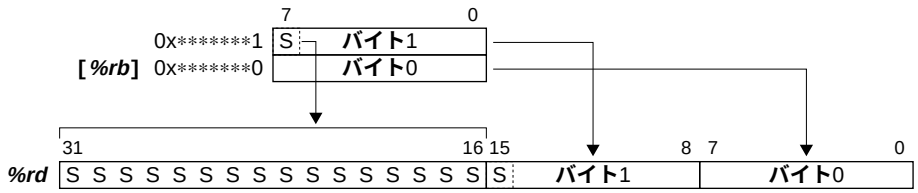


図3.9.1 符号付き16ビット転送(メモリ→レジスタ)

転送先レジスタのビット31～16は符号拡張されます。

3.10 16ビット転送(レジスタ→メモリ)

例: ld.h [%rb],%rs （リトルエンディアンの場合）

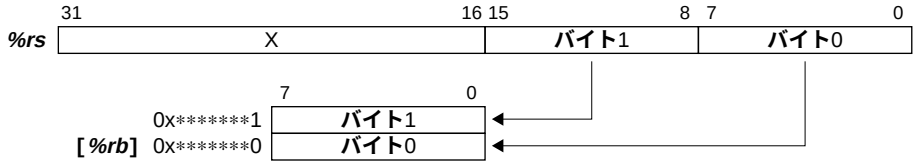


図3.10.1 16ビット転送(レジスタ→メモリ)

3.11 32ビット転送(レジスタ→レジスタ)

例: ld.w %rd,%rs

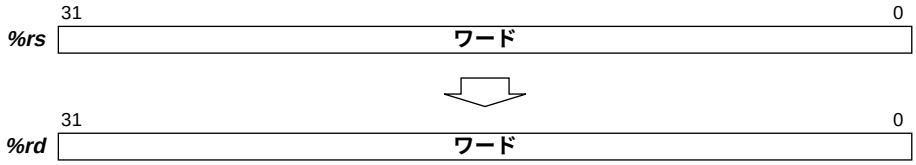


図3.11.1 32ビット転送(レジスタ→レジスタ)

3.12 32ビット転送(メモリ→レジスタ)

例: ld.w %rd,[%rb] （リトルエンディアンの場合）

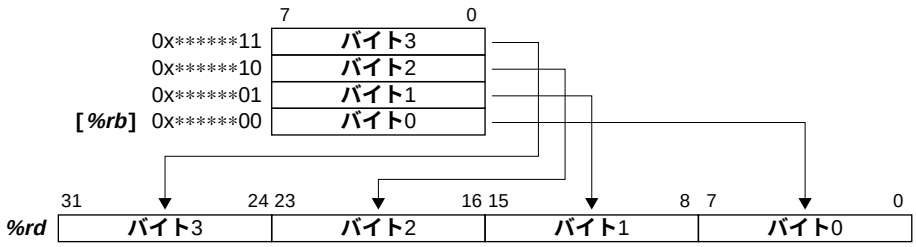


図3.12.1 32ビット転送(メモリ→レジスタ)

3.13 32ビット転送(レジスタ→メモリ)

例: ld.w [%rb],%rs （リトルエンディアンの場合）

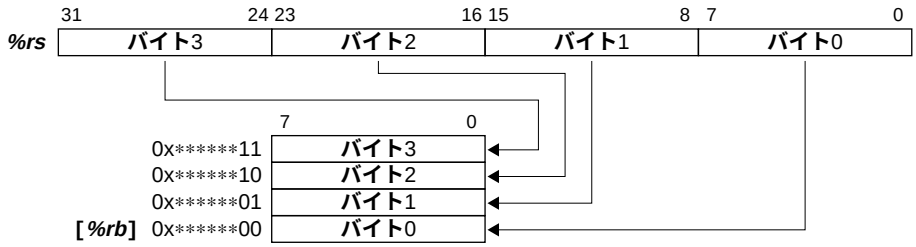


図3.13.1 32ビット転送(レジスタ→メモリ)

4 アドレスマップ

C33 ADVコアCPUは4GBのアドレス空間を持っています。CPUが出力するアドレスは、HBCUとMMUによってあらかじめ設定したアドレスに変換できるため、実際のROMやRAM、I/Oデバイスなどに割り当てられた物理アドレスである必要はありません。つまり、CPUは自由なメモリ構成(仮想空間)を構築可能な4GBのリニアな論理アドレス空間をアクセスできます。CPUが出力した論理アドレスはHBCUがMMUを使用して物理アドレスに変換し、物理アドレス空間を管理しているBBCUに渡されて実際のデバイスがアクセスされます。4GBの物理アドレス空間はBBCUによって23のエリアに分けられ、エリアごとに各種のメモリやI/Oデバイスを配置できるようになっています。

CPUから出力された論理アドレスはHBCU、MMU、BBCUによって以下の5種類のアドレス処理を経て最終的な物理アドレスになります。

1. ブロック処理
2. ASID処理
3. アドレス変換処理
4. ミラー処理
5. エリア処理

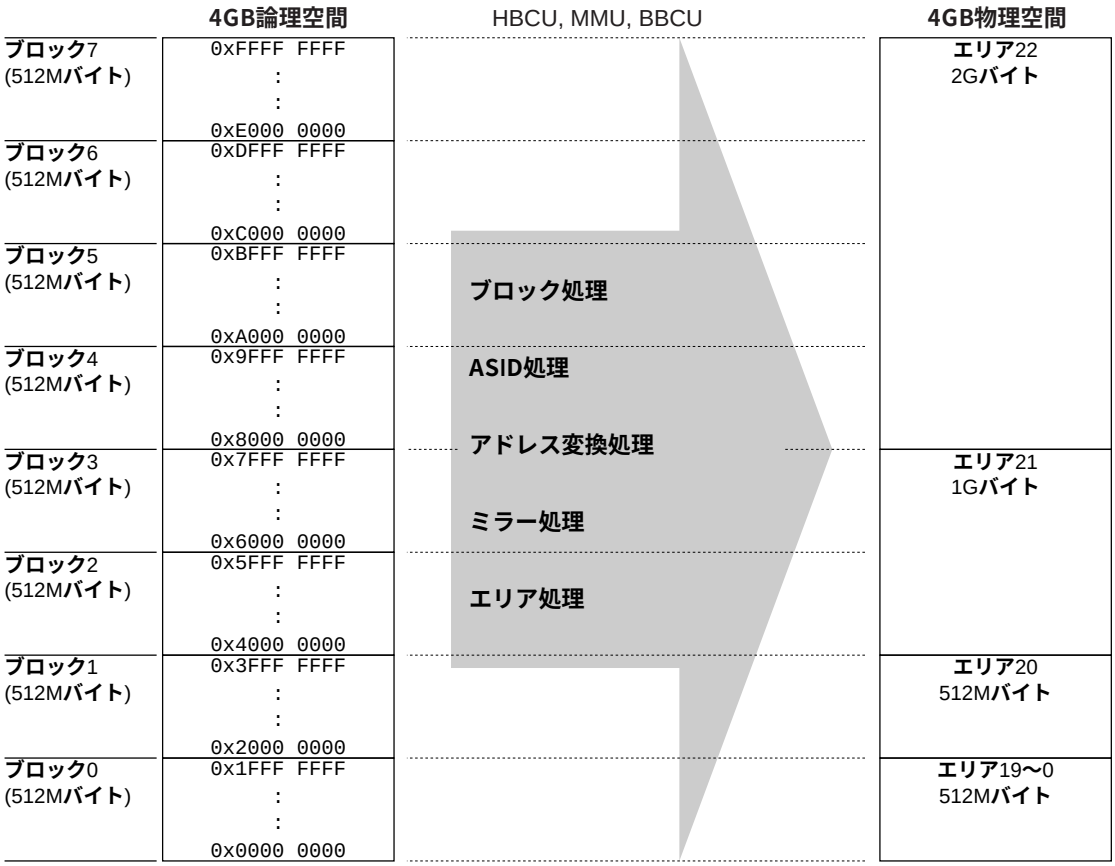


図4.1 論理空間と物理空間の対応

アドレス処理の詳細については、各機種のテクニカルマニュアルを参照してください。

エリア13	0x02FF FFFF 0x0200 0000	**外部メモリ** 16M**バイト**
エリア12	0x01FF FFFF 0x0180 0000	**外部メモリ** 8M**バイト**
エリア11	0x017F FFFF 0x0100 0000	**外部メモリ** 8M**バイト**
エリア10	0x00FF FFFF 0x00C0 0000	**外部メモリ** 4M**バイト**
エリア9	0x00BF FFFF 0x0080 0000	**外部メモリ** 4M**バイト**
エリア8	0x007F FFFF 0x0060 0000	**外部メモリ** 2M**バイト**
エリア7	0x005F FFFF 0x0040 0000	**外部メモリ** 2M**バイト**
エリア6	0x003F FFFF 0x0030 0000	**外部メモリ** 1M**バイト**
エリア5	0x002F FFFF 0x0020 0000	**外部メモリ** 1M**バイト**
エリア4	0x001F FFFF 0x0010 0000	**外部メモリ** 1M**バイト**
エリア3	0x000F FFFF 0x0008 0000	**内部**RAM 512K**バイト**
エリア2	0x0007 FFFF 0x0006 0000	**デバッグ用エリア** 128K**バイト**
エリア1	0x0005 FFFF 0x0002 0000	**内部**I/O 256K**バイト**
エリア0	0x0001 FFFF 0x0000 0000	**内部**RAM 128K**バイト**

エリア22	0xFFFF FFFF 0x8000 0000	**外部メモリ** 2G**バイト**
エリア21	0x7FFF FFFF 0x4000 0000	**外部メモリ** 1G**バイト**
エリア20	0x3FFF FFFF 0x2000 0000	**外部メモリ** 512M**バイト**
エリア19	0x1FFF FFFF 0x1000 0000	**外部メモリ** 256M**バイト**
エリア18	0x0FFF FFFF 0x0C00 0000	**外部メモリ** 64M**バイト**
エリア17	0x0BFF FFFF 0x0800 0000	**外部メモリ** 64M**バイト**
エリア16	0x07FF FFFF 0x0600 0000	**外部メモリ** 32M**バイト**
エリア15	0x05FF FFFF 0x0400 0000	**外部メモリ** 32M**バイト**
エリア14	0x03FF FFFF 0x0300 0000	**外部メモリ** 16M**バイト**

図4.2 物理アドレス空間

5 命令セット

C33 ADVコアCPUの命令セットは、S1C33 STDコアCPUの命令セットをもとに命令の機能を拡張すると共にマルチメディア処理に有効な命令の追加を行っています。オブジェクトコードはS1C33 STDコアCPUの上位互換となっていますので、S1C33シリーズの資産を若干の変更のみでC33アドバンスド機種に移植可能です。命令コードはすべて16ビットの固定長で、パイプライン処理を行うことによって主要な命令を1サイクルで実行します。各命令の詳細については“7 命令の詳細説明”を参照してください。

5.1 S1C33シリーズ互換命令

表5.1.1 S1C33シリーズ互換命令

種　類	ニーモニック		機　能
算術演算	add	*%rd,%rs*	汎用レジスタ間の加算
		%rd,imm6	汎用レジスタと即値の加算
		%sp,*imm10*	SPと即値の加算(即値はゼロ拡張)
	adc	*%rd,%rs*	汎用レジスタ間のキャリー付き加算
	sub	*%rd,%rs*	汎用レジスタ間の減算
		%rd,imm6	汎用レジスタと即値の減算
		%sp,*imm10*	SPと即値の減算(即値はゼロ拡張)
	sbc	*%rd,%rs*	汎用レジスタ間のキャリー付き減算
	cmp	*%rd,%rs*	汎用レジスタ間の算術比較
		%rd,sign6	汎用レジスタと即値の算術比較(即値はゼロ拡張)
	mlt.h	*%rd,%rs*	符号付き整数乗算(16ビット × 16ビット → 32ビット)
	mltu.h	*%rd,%rs*	符号なし整数乗算(16ビット × 16ビット → 32ビット)
	mlt.w	*%rd,%rs*	符号付き整数乗算(32ビット × 32ビット → 64ビット)
	mltu.w	*%rd,%rs*	符号なし整数乗算(32ビット × 32ビット → 64ビット)
	div0s	*%rs*	符号付き整数除算の第1ステップ
	div0u	*%rs*	符号なし整数除算の第1ステップ
	div1	*%rs*	ステップ除算実行
	div2s	*%rs*	符号付き整数除算結果のデータ補正1
	div3s		符号付き整数除算結果のデータ補正2
分岐	jrgt jrgt.d	*sign8*	PC相対条件ジャンプ　分岐条件: !Z & !(N ^ V) ディレイド分岐可
	jrge jrge.d	*sign8*	PC相対条件ジャンプ　分岐条件: !(N ^ V) ディレイド分岐可
	jrlt jrlt.d	*sign8*	PC相対条件ジャンプ　分岐条件: N ^ V ディレイド分岐可
	jrle jrle.d	*sign8*	PC相対条件ジャンプ　分岐条件: Z | N ^ V ディレイド分岐可
	jrugt jrugt.d	*sign8*	PC相対条件ジャンプ　分岐条件: !Z & !C ディレイド分岐可
	jruge jruge.d	*sign8*	PC相対条件ジャンプ　分岐条件: !C ディレイド分岐可
	jrult jrult.d	*sign8*	PC相対条件ジャンプ　分岐条件: C ディレイド分岐可
	jrule jrule.d	*sign8*	PC相対条件ジャンプ　分岐条件: Z | C ディレイド分岐可
	jreq jreq.d	*sign8*	PC相対条件ジャンプ　分岐条件: Z ディレイド分岐可
	jrne jrne.d	*sign8*	PC相対条件ジャンプ　分岐条件: !Z ディレイド分岐可
	jp	*sign8*	PC相対ジャンプ　ディレイド分岐可
	jp.d	*%rb*	絶対ジャンプ　ディレイド分岐可
	call	*sign8*	PC相対サブルーチンコール　ディレイドコール可
	call.d	*%rb*	絶対サブルーチンコール　ディレイドコール可

種　類	ニーモニック		機　能
分岐	ret		サブルーチンからのリターン
	ret.d		ディレイドリターン可
	reti		割り込み、例外処理ルーチンからのリターン
	retd		デバッグ処理ルーチンからのリターン
	int	*imm2*	ソフトウェア例外
	brk		デバッグ例外
データ転送	ld.b	*%rd,%rs*	汎用レジスタ(バイト) → 汎用レジスタ(符号拡張)
		%rd,[*%rb*]	メモリ(バイト) → 汎用レジスタ(符号拡張)
		%rd,[*%rb*]+	ポストインクリメント可
		%rd,[%sp+*imm6*]	スタック(バイト) → 汎用レジスタ(符号拡張)
		[*%rb*]*,%rs*	汎用レジスタ(バイト) → メモリ
		[*%rb*]+*,%rs*	ポストインクリメント可
		[%sp+*imm6*]*,%rs*	汎用レジスタ(バイト) → スタック
	ld.ub	*%rd,%rs*	汎用レジスタ(バイト) → 汎用レジスタ(ゼロ拡張)
		%rd,[*%rb*]	メモリ(バイト) → 汎用レジスタ(ゼロ拡張)
		%rd,[*%rb*]+	ポストインクリメント可
		%rd,[%sp+*imm6*]	スタック(バイト) → 汎用レジスタ(ゼロ拡張)
	ld.h	*%rd,%rs*	汎用レジスタ(ハーフワード) → 汎用レジスタ(符号拡張)
		%rd,[*%rb*]	メモリ(ハーフワード) → 汎用レジスタ(符号拡張)
		%rd,[*%rb*]+	ポストインクリメント可
		%rd,[%sp+*imm6*]	スタック(ハーフワード) → 汎用レジスタ(符号拡張)
		[*%rb*]*,%rs*	汎用レジスタ(ハーフワード) → メモリ
		[*%rb*]+*,%rs*	ポストインクリメント可
		[%sp+*imm6*]*,%rs*	汎用レジスタ(ハーフワード) → スタック
	ld.uh	*%rd,%rs*	汎用レジスタ(ハーフワード) → 汎用レジスタ(ゼロ拡張)
		%rd,[*%rb*]	メモリ(ハーフワード) → 汎用レジスタ(ゼロ拡張)
		%rd,[*%rb*]+	ポストインクリメント可
		%rd,[%sp+*imm6*]	スタック(ハーフワード) → 汎用レジスタ(ゼロ拡張)
	ld.w	*%rd,%rs*	汎用レジスタ(ワード) → 汎用レジスタ
		%rd,sign6	即値 → 汎用レジスタ(符号拡張)
		%rd,[*%rb*]	メモリ(ワード) → 汎用レジスタ
		%rd,[*%rb*]+	ポストインクリメント可
		%rd,[%sp+*imm6*]	スタック(ワード) → 汎用レジスタ
		[*%rb*]*,%rs*	汎用レジスタ(ワード) → メモリ
		[*%rb*]+*,%rs*	ポストインクリメント可
		[%sp+*imm6*]*,%rs*	汎用レジスタ(ワード) → スタック
システム制御	nop		ノーオペレーション
	halt		HALT
	slp		SLEEP
即値拡張	ext	*imm13*	直後の命令のオペランドを拡張
ビット処理	btst	[*%rb*]*,imm3*	メモリデータの指定ビットをテスト
	bclr	[*%rb*]*,imm3*	メモリデータの指定ビットをクリア
	bset	[*%rb*]*,imm3*	メモリデータの指定ビットをセット
	bnot	[*%rb*]*,imm3*	メモリデータの指定ビットを反転
その他	swap	*%rd,%rs*	ワード中のバイト境界でバイト単位のスワップ
	mirror	*%rd,%rs*	ワード中のバイトごとにビット単位でスワップ
	mac	*%rs*	積和演算(16ビット × 16ビット + 64ビット → 64ビット)
	pushn	*%rs*	*%rs*～%r0までの汎用レジスタをスタックにプッシュ
	popn	*%rd*	*%rd*～%r0までの汎用レジスタのデータをスタックからポップ

5.2 機能拡張された命令

表5.2.1 機能拡張された命令

種　　類	ニーモニック		機　　能	拡張機能
論理演算	and	*%rd,%rs*	汎用レジスタ間の論理積	命令実行後、Vフラグをクリアするモードを付加しました。
		%rd,sign6	汎用レジスタと即値の論理積	
	or	*%rd,%rs*	汎用レジスタ間の論理和	
		%rd,sign6	汎用レジスタと即値の論理和	
	xor	*%rd,%rs*	汎用レジスタ間の排他的論理和	
		%rd,sign6	汎用レジスタと即値の排他的論理和	
	not	*%rd,%rs*	汎用レジスタ間の論理反転(1の補数)	
		%rd,sign6	汎用レジスタと即値の論理反転(1の補数)	
シフト&ローテート	srl	*%rd,%rs*	右論理シフト(シフト量: レジスタ指定 0～31)	9～31ビットのシフト/ローテートが可能となりました。
		%rd,imm5	右論理シフト(シフト量: 即値指定 0～31)	
	sll	*%rd,%rs*	左論理シフト(シフト量: レジスタ指定 0～31)	
		%rd,imm5	左論理シフト(シフト量: 即値指定 0～31)	
	sra	*%rd,%rs*	右算術シフト(シフト量: レジスタ指定 0～31)	
		%rd,imm5	右算術シフト(シフト量: 即値指定 0～31)	
	sla	*%rd,%rs*	左算術シフト(シフト量: レジスタ指定 0～31)	
		%rd,imm5	左算術シフト(シフト量: 即値指定 0～31)	
	rr	*%rd,%rs*	右ローテート(シフト量: レジスタ指定 0～31)	
		%rd,imm5	右ローテート(シフト量: 即値指定 0～31)	
	rl	*%rd,%rs*	左ローテート(シフト量: レジスタ指定 0～31)	
		%rd,imm5	左ローテート(シフト量: 即値指定 0～31)	
データ転送	ld.w	*%rd,%ss*	特殊レジスタ(ワード) → 汎用レジスタ	ロード対象の特殊レジスタが増えました。
		%sd,%rs	汎用レジスタ(ワード) → 特殊レジスタ	
その他	scan0	*%rd,%rs*	0のビットをサーチ	スキャン対象が32ビットまで可能となりました。
	scan1	*%rd,%rs*	1のビットをサーチ	

5.3 C33 ADVコアCPUの追加命令

表5.3.1 C33 ADVコアCPUの追加命令

種　類	ニーモニック		機　能
算術演算	add	*%rd*,%dp	DPレジスタの加算
	mlt.hw	*%rd*,*%rs*	符号付き整数乗算(32ビット × 16ビット → 64ビット)
	mac.hw	*%rs*	積和演算(32ビット × 16ビット + 64ビット → 64ビット)
	mac.w	*%rs*	積和演算(32ビット × 32ビット + 64ビット → 64ビット)
	mac1.h	*%rd*,*%rs*	1命令積和演算(16ビット × 16ビット + 64ビット → 64ビット)
	mac1.hw	*%rd*,*%rs*	1命令積和演算(32ビット × 16ビット + 64ビット → 64ビット)
	mac1.w	*%rd*,*%rs*	1命令積和演算(32ビット × 32ビット + 64ビット → 64ビット)
	div.w	*%rs*	符号付き整数除算(32ビット / 32ビット → 16ビット … 16ビット)
	divu.w	*%rs*	符号なし整数除算(32ビット / 32ビット → 16ビット … 16ビット)
分岐	jpr	*%rb*	PC相対ジャンプ
	jpr.d		ディレイド分岐可
	retm		MMU処理ルーチンからのリターン
データ転送	ld.b	*%rd*,[%dp+*imm6*]	メモリ間接(バイト) → 汎用レジスタ(符号拡張)
	ld.ub	*%rd*,[%dp+*imm6*]	メモリ間接(バイト) → 汎用レジスタ(ゼロ拡張)
	ld.h	*%rd*,[%dp+*imm6*]	メモリ間接(ハーフワード) → 汎用レジスタ(符号拡張)
	ld.uh	*%rd*,[%dp+*imm6*]	メモリ間接(ハーフワード) → 汎用レジスタ(ゼロ拡張)
	ld.w	*%rd*,[%dp+*imm6*]	メモリ間接(ワード) → 汎用レジスタ
	ld.b	[%dp+*imm6*],*%rs*	汎用レジスタ(バイト) → メモリ間接(符号拡張)
	ld.h	[%dp+*imm6*],*%rs*	汎用レジスタ(ハーフワード) → メモリ間接(符号拡張)
	ld.w	[%dp+*imm6*],*%rs*	汎用レジスタ(ワード) → メモリ間接
システム制御	psrset	*imm5*	PSRの指定ビットをセット
	psrclr	*imm5*	PSRの指定ビットをクリア
多機能拡張	ext	*%rs*	直後の命令を3オペランド実行
	ext	*cond*	条件実行
	ext	*op*,*imm2*	ポストシフト
	ext	*%rs*,*op*,*imm2*	3オペランド実行 + ポストシフト
コプロセッサ制御	ld.c	*%rd*,*imm4*	コプロセッサからのデータロード
	ld.c	*imm4*,*%rs*	コプロセッサへのデータストア
	do.c	*imm6*	コプロセッサ実行
	ld.cf		コプロセッサからC、V、Z、Nフラグをロード
その他	macclr		ALR、AHRレジスタおよびMOフラグを0クリア
	swaph	*%rd*,*%rs*	ワード中のハーフワード境界でバイト単位のスワップ
	push	*%rs*	汎用レジスタの単独プッシュ
	pop	*%rd*	汎用レジスタの単独ポップ
	pushs	*%ss*	*%ss*～ALRまでの特殊レジスタをスタックにプッシュ
	pops	*%sd*	*%sd*～ALRまでの特殊レジスタのデータをスタックからポップ
	sat.b	*%rd*,*%rs*	汎用レジスタ(バイト)の符号付き飽和処理
	sat.ub	*%rd*,*%rs*	汎用レジスタ(バイト)の符号なし飽和処理
	sat.h	*%rd*,*%rs*	汎用レジスタ(ハーフワード)の符号付き飽和処理
	sat.uh	*%rd*,*%rs*	汎用レジスタ(ハーフワード)の符号なし飽和処理
	sat.w	*%rd*,*%rs*	汎用レジスタ(ワード)の符号付き飽和処理
	sat.uw	*%rd*,*%rs*	汎用レジスタ(ワード)の符号なし飽和処理
	loop	*%rc*,*%ra*	指定範囲(汎用レジスタ)、指定回数(汎用レジスタ)の繰り返し実行
		%rc,*imm4*	指定範囲(即値)、指定回数(汎用レジスタ)の繰り返し実行
		imm4,*imm4*	指定範囲(即値)、指定回数(即値)の繰り返し実行
	repeat	*%rc*	直後の命令の繰り返し実行(汎用レジスタ回数指定)
		imm4	直後の命令の繰り返し実行(即値回数指定)

表中の記号の意味は次のとおりです。

表5.3.2 記号の意味

記　　号	説　　明
%rs	汎用レジスタ(ソース)
%rd	汎用レジスタ(デスティネーション)
%ss	特殊レジスタ(ソース)
%sd	特殊レジスタ(デスティネーション)
[*%rb*]	汎用レジスタ(間接アドレス)
[*%rb*]+	汎用レジスタ(ポストインクリメント付き間接アドレス)
%rc	汎用レジスタ(ループカウント)
%ra	汎用レジスタ(ループアドレス)
%sp	スタックポインタ
%dp	データポインタ
imm2,*imm4*,*imm3*, *imm5*,*imm6*,*imm10*, *imm13*	符号なし即値(数値はビット長) ただし、シフト命令ではシフト量、ビット処理命令においては ビット位置を表します。
sign6,*sign8*	符号付き即値(数値はビット長)

5.4 アドレッシングモード(ext拡張なし)

C33 ADVコアCPUはS1C33シリーズと同様、命令セットは以下に示す6種類のアドレッシングモードを持ちます。CPUは各命令のオペランドによってアドレッシングモードを決定し、データをアクセスします。
(1)即値アドレッシング
(2)レジスタ直接アドレッシング
(3)レジスタ間接アドレッシング
(4)ポストインクリメント付きレジスタ間接アドレッシング
(5)ディスプレースメント付きレジスタ間接アドレッシング
(6)符号付きPC相対アドレッシング

5.4.1 即値アドレッシング

命令コード中に含まれる*immX*(符号なし即値)、*signX*(符号付き即値)で示される即値をソースデータとして使用します。各命令で指定可能な即値サイズは、シンボルの数字(例: *imm4* = 符号なし4ビット、*sign6* = 符号付き6ビット)で示されます。*sign6*などの符号付き即値は最上位ビットが符号となり、このビットが命令実行時にビット31まで拡張されます。

例: `ld.w %r0,0x30`

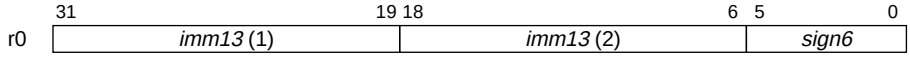
実行前 r0 = 0x*XXXXXXXX*
実行後 r0 = 0xFFFFFFF0

*sign6*は+31～-32(0b011111～0b100000)まで表すことができます。

また、シフト系命令とビット処理命令を除き、`ext`命令によって即値を最大32ビットまで拡張することができます。

例:
```
ext     imm13       (1)
ext     imm13       (2)
ld.w    %r0,sign6
```

実行後のr0



5.4.2 レジスタ直接アドレッシング

指定のレジスタの内容をそのままソースデータとして使用します。また、結果をレジスタにロードする命令のデスティネーションとして指定した場合は、結果がそのレジスタにロードされます。以下のシンボルをオペランドとして持つ命令がこのアドレッシングモードで実行されます。

%rs *rs*は演算や転送のソースデータを保持している汎用レジスタを示すメタシンボルです。実際には`%r0`～`%r15`と記述します。

%rd *rd*はデスティネーションとなる汎用レジスタを示すメタシンボルです。実際には`%r0`～`%r15`と記述します。命令によってはソースデータにもなります。

%ss *ss*は汎用レジスタに転送するソースデータを保持している特殊レジスタを示すメタシンボルです。

%sd *sd*は汎用レジスタからデータをロードする特殊レジスタを示すメタシンボルです。

実際の特殊レジスタ名は次のように記述します。

プロセッサステータスレジスタ	%psr
スタックポインタ	%sp
算術演算下位レジスタ	%alr
算術演算上位レジスタ	%ahr
ループカウントレジスタ	%lco
ループスタートアドレスレジスタ	%lsa
ループエンドアドレスレジスタ	%lea
トラップテーブルベースレジスタ	%ttbr
データポインタ	%dp
シフトアウトレジスタ	%sor
ユーザスタックポインタ	%usp
スーパバイザスタックポインタ	%ssp

レジスタ名はシンボル名やラベル名等と区別するため、必ず“%”を前置きします。

5.4.3 レジスタ間接アドレッシング

アドレスを保持している汎用レジスタを指定して、間接的にメモリをアクセスするモードです。[*%rb*]をオペランドとして持つロード命令にのみ適用されます。実際は汎用レジスタ名を[]で囲み、[%r0]、[%r1]、... [%r15]のように記述します。
CPUは指定レジスタの内容をベースアドレスとして、ロード命令の種類によって決まるデータ形式でデータ転送を行います。
例: メモリ → レジスタ

```
ld.b    %r0,[%r1]
ld.h    %r0,[%r1]
ld.w    %r0,[%r1]
```

レジスタ → メモリ

```
ld.b    [%r1],%r0
ld.h    [%r1],%r0
ld.w    [%r1],%r0
```

この例では、r1の示すアドレスがデータを転送する対象のメモリアドレスになります。

ハーフワード転送、ワード転送の場合、レジスタに設定するベースアドレスはそれぞれハーフワード境界(最下位ビット = 0)、ワード境界(下位2ビット = 0)を指している必要があり、それ以外ではアドレス不整例外が発生します。

5.4.4 ポストインクリメント付きレジスタ間接アドレッシング

レジスタ間接アドレッシングと同様に、汎用レジスタによってアクセスするメモリを間接的に指定します。データ転送が終了すると、指定したレジスタが保持しているベースアドレスを、転送したデータサイズ分インクリメント*します。これにより、最初に1回先頭アドレスを設定するだけで、メモリ上の連続したデータの読み出し/書き込みが行えます。

* インクリメントサイズ

バイト転送(ld.b, ld.ub):	*rb* → *rb* + 1
ハーフワード転送(ld.h, ld.uh):	*rb* → *rb* + 2
ワード転送(ld.w):	*rb* → *rb* + 4

このアドレッシングモードは、レジスタ名を[ ]で囲み“+”を後置きして指定します。
実際には[%r0]+、[%r1]+、... [%r15]+のように記述します。

5.4.5 ディスプレースメント付きレジスタ間接アドレッシング

レジスタの内容に指定の即値(ディスプレースメント)を加えたアドレスから始まるメモリをアクセスするモードです。ext命令を使用しない場合、このアドレッシングモードは[%sp+*imm6*]および[%dp+*imm6*]をオペランドとして持つロード命令にのみ適用されます。

例: `ld.b %r0,[%sp+0x10]`

現在のSPの内容に0x10を加算したアドレスのバイトデータをr0レジスタにロードします。バイトデータ転送の場合、6ビットの即値がそのままディスプレースメントとして加算されます。

`ld.h %r0,[%dp+0x10]`

現在のDPの内容に0x20を加算したアドレスのハーフワードデータをr0レジスタにロードします。ハーフワードデータ転送の場合、ハーフワード境界をアクセスするため、指定した6ビット即値を2倍(最下位ビットは常に0)にした値がディスプレースメントとなります。

`ld.w %r0,[%sp+0x10]`

現在のSPの内容に0x40を加算したアドレスのワードデータをr0レジスタにロードします。ワードデータ転送の場合、ワード境界をアクセスするため、指定した6ビット即値を4倍(下位2ビットは常に0)にした値がディスプレースメントとなります。

ext命令を使用すると、通常のレジスタ間接アドレッシング([*%rb*])がext命令で指定した即値をディスプレースメントとするアドレッシングモードに変わります(詳細は5.5節を参照)。

例:
```
ext     imm13
ld.b    %rd,[%rb]       アクセスするメモリアドレスは"%rb+imm13"となります。
```

5.4.6 符号付きPC相対アドレッシング

符号付き8ビット即値(*sign8*)をオペランドに持つ分岐命令に適用されるアドレッシングモードです。それらの分岐命令を実行すると、現在のPCに*sign8*の2倍の値(ハーフワード境界)を加算したアドレスに分岐します。

例:
```
PC + 0    jrne  0x04        jrneの分岐条件が成立するとPC + 8番地に分岐します。
   :        :               (PC + 0) + 0x04 * 2 → PC + 8
   :        :
PC + 8
```

5.5 ext付きアドレッシングモード

16ビット固定長の命令コードで指定できる即値は、命令によって異なりますが4ビット～8ビットのビットフィールドで指定します。ext命令はこの即値のサイズを拡張するために使用します。
ext命令は、データ転送命令や演算命令と組み合わせて使用し、即値の拡張を行いたい命令の直前に置きます。命令はext *imm13*の形式で記述します。1個のext命令で拡張可能な即値サイズは13ビットで、さらに即値拡張するために2個までext命令を続けて記述することができます。
ext命令が有効となるのは直後に記述された即値拡張が可能な命令に対してのみで、その他の命令に対しては無効です。また、3個以上のext命令が連続的に記述された場合は、最初と最後(即値拡張の対象となる命令の直前のext命令)の2個のみが有効で、その間のext命令は無効となります。

また、C33 ADVコアCPUでは多機能ext命令が追加されています。詳細は後述します。

5.5.1 即値アドレッシングの拡張

●*imm6*の拡張

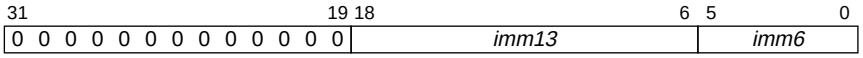
*imm6*を19ビット即値あるいは32ビット即値に拡張します。

19ビット即値への拡張

即値を19ビットに拡張するには、1個のext命令を対象命令の直前に置きます。

例:
```
ext   imm13
add   %rd,imm6
```

拡張された即値



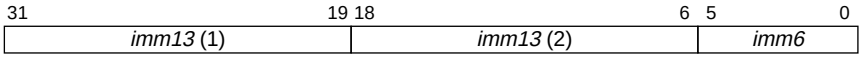
ビット31～19は0で埋まります(ゼロ拡張)。

32ビット即値への拡張

即値を32ビットに拡張するには、2個のext命令を対象命令の直前に置きます。

例:
```
ext   imm13      (1)
ext   imm13      (2)
sub   %rd,imm6
```

拡張された即値



●*sign6*の拡張

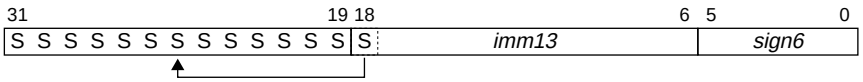
*sign6*を符号拡張された19ビット即値あるいは32ビット即値に拡張します。

19ビット即値への拡張

即値を19ビットに拡張するには、1個のext命令を対象命令の直前に置きます。

例:
```
ext   imm13
ld.w  %rd,sign6
```

拡張された即値



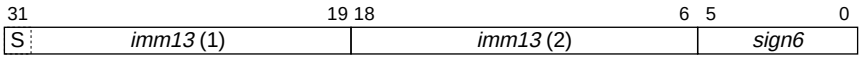
ext命令で拡張された*imm13*の最上位ビット"S"が符号としてビット31～19に拡張され、19ビットの符号付きデータとなります。*sign6*の最上位ビットは符号ではなく、6ビットデータの最上位データとして扱われます。

32ビット即値への拡張

32ビットへ即値を32ビットに拡張するには、2個のext命令を対象命令の直前に置きます。

例: ext *imm13* (1)
ext *imm13* (2)
and *%rd*,*sign6*

拡張された即値



最初のext命令の即値のMSB(ビット12)を符号として、32ビットの符号付きデータに拡張されます。

5.5.2 レジスタ間接アドレッシングの拡張

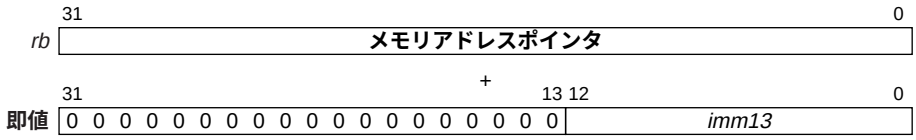
●[*%rb*]にディスプレースメントを付加

[*%rb*]で間接参照されるアドレスに、ext命令で指定される即値を加算したアドレスをアクセスします。

13ビットの即値を加算

*rb*レジスタで指定されるアドレスに*imm13*で指定される13ビットの即値を加えたアドレスをアクセスします。アドレス演算の際、*imm13*は32ビットにゼロ拡張されます。

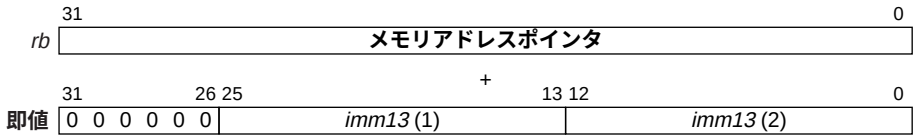
例: ext *imm13*
ld.b *%rd*,[*%rb*]



26ビットの即値を加算

*rb*レジスタで指定されるアドレスに*imm26*で指定される26ビットの即値を加えたアドレスをアクセスします。アドレス演算の際、*imm26*は32ビットにゼロ拡張されます。

例: ext *imm13* (1)
ext *imm13* (2)
ld.uh *%rd*,[*%rb*]



●[%sp+*imm6*]、[%dp+*imm6*]のディスプレースメントを拡張

ディスプレースメント付きレジスタ間接アドレッシング命令の即値(*imm6*)を拡張します。*imm6*の取り扱いがext命令を付加しない単独命令と異なりますので注意してください。
ディスプレースメント付きレジスタ間接アドレッシング命令は、単独使用時には命令の転送データサイズによって境界アドレスを自動的に計算します。
例: `ld.h %rd,[%dp+imm6]`

この例の参照アドレスは、ハーフワード境界の“dp + *imm6* * 2”番地になります。

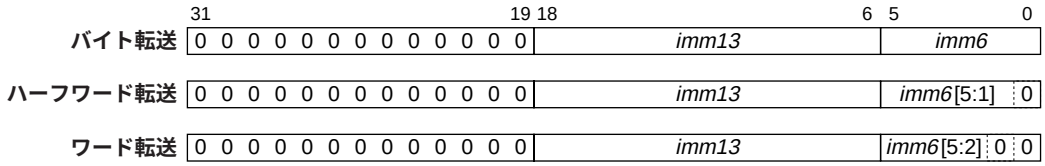
ext命令を付加した場合については、以下の説明を参照してください。

19ビット即値への拡張

即値を19ビットに拡張するには、1個のext命令を対象命令の直前に置きます。19ビットに拡張された即値は、データの転送サイズによって下位ビットが0または00に固定されます。(バイト転送以外のとき)

例:
```
ext   imm13
ld.b  %rd,[%sp+imm6]

ext   imm13
ld.h  [%sp+imm6],%rs
```

拡張された即値



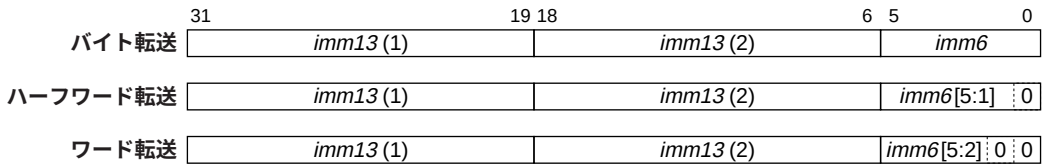
拡張されたデータはspまたはdpと加算され、転送先または転送元アドレスとして扱われます。

32ビット即値への拡張

即値を32ビットに拡張するには、2個のext命令を対象命令の直前に配置します。32ビットに拡張された即値は、データの転送サイズによって下位ビットが0または00に固定されます。(バイト転送以外のとき)

例:
```
ext   imm13      (1)
ext   imm13      (2)
ld.b  %rd,[%sp+imm6]

ext   imm13      (1)
ext   imm13      (2)
ld.h  [%sp+imm6],%rs
```

拡張された即値



拡張されたデータはspまたはdpと加算され、転送先または転送元アドレスとして扱われます。

●レジスタ間演算命令を拡張

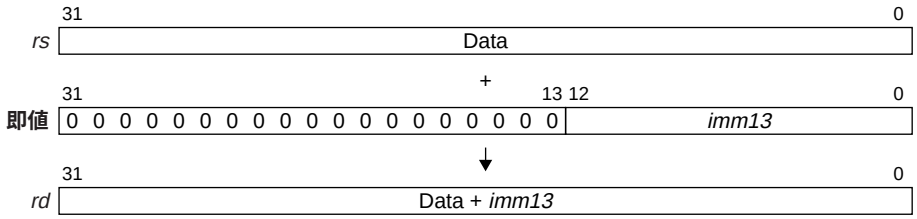
レジスタ間の演算命令をext命令で拡張します。データ転送命令と異なり、この場合は*rs*レジスタの内容とext命令で指定する即値を命令に従って演算し、その結果を*rd*レジスタに格納します。*rd*レジスタの内容は演算には影響を与えません。
加算の場合の拡張例を以下に示します。

rs + *imm13*への拡張

rs + *imm13*に拡張するには、1個のext命令を対象命令の直前に置きます。

例:
```
ext   imm13
add   %rd,%rs
```

拡張がない場合 → *rd* = *rd* + *rs*
1個のext命令で拡張した場合 → *rd* = *rs* + *imm13*

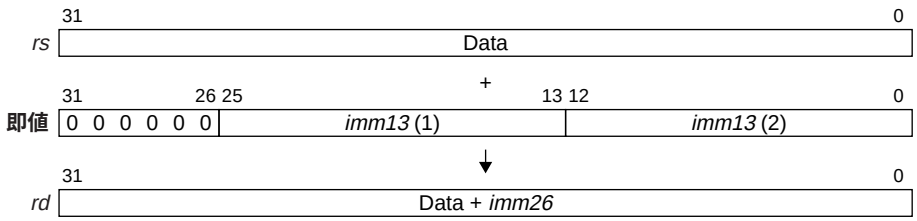


rs + *imm26*への拡張

rs + *imm26*に拡張するには、2個のext命令を対象命令の直前に置きます。

例:
```
ext   imm13      (1)
ext   imm13      (2)
add   %rd,%rs
```

拡張がない場合 → *rd* = *rd* + *rs*
2個のext命令で拡張した場合 → *rd* = *rs* + *imm26*



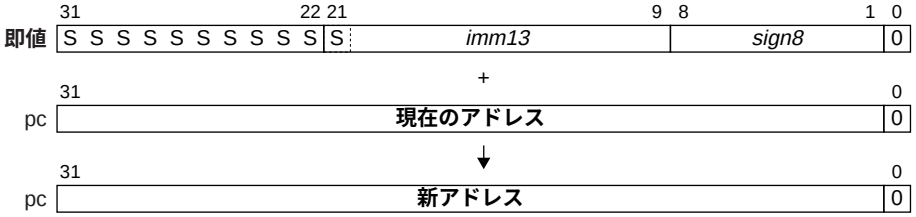
●PC相対分岐命令のディスプレースメントの拡張

PC相対分岐命令の即値*sign8*を、22ビットの符号付き即値または32ビットの符号付き即値に拡張します。*sign8*を2倍にして分岐先アドレスの相対値に変換後、PCと加算して分岐先アドレスを決定します。ext命令はこの分岐相対値を拡張します。

22ビット即値への拡張

*sign8*を22ビットの即値に拡張するには、1個のext命令を対象命令の直前に置きます。

例:
```
ext   imm13
jrgt  sign8
```

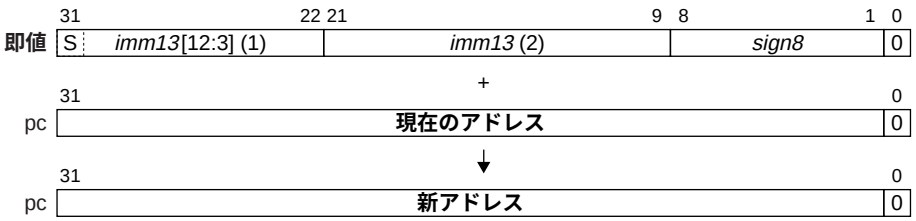


ext命令で拡張された即値の最上位ビット“S”が符号としてビット31～22に拡張され、22ビットの符号付きデータとなります。*sign8*の最上位ビットは符号ではなく、8ビットデータの最上位データとして扱われます。

32ビット即値への拡張

*sign8*を32ビットの即値に拡張するには、2個のext命令を対象命令の直前に置きます。

例:
```
ext   imm13      (1)
ext   imm13      (2)
jrgt  sign8
```



ext命令で拡張された即値の最上位ビット“S”が符号になります。最初のext命令のビット2～0は使用されません。

5.5.3 ポストインクリメント付きレジスタ間接アドレッシング

データ転送用のポストインクリメント付き間接アドレッシング命令にext拡張命令を付加すると、レジスタメモリ間のデータ転送を行った後、間接アドレスを示すレジスタに、ext命令で指定した即値を加算します。即値は符号付きの値として扱われますので、減算を行うことも可能です。
1個のext命令で符号付き13ビットの値を、2個のext命令で符号付き26ビットの値を指定できます。
さらに、多機能ext拡張命令を使用することで、符号付き32ビットの指定ができます。

例1:
```
ext     0x100
ld.w    [%r6]+,%r3      W[r6] → r3, r6 → r6 + 0x100
```

例2:
```
ext     %r1
ld.h    %r5,[%r4]+      r5 → HW[r4], r4 → r4 + r1
```

5.5.4 ext命令の例外処理

ext命令の処理中に発生した例外は、リセットとデバッグブレークについてはすぐに例外処理を開始しますが、これら以外の例外については拡張対象命令を実行するまで処理を開始しません。これは、プリフェッチ時のext命令の圧縮動作を簡略化するためです。
例外処理最後のreti、retm、retd命令では、ext命令(複数ある場合はその先頭、ただしext命令を7つ以上同時に使っている場合には7つ以上前のextには戻れません)のアドレスにリターンするため、例外処理によるext命令の誤動作は起こらないようになっています。

C33 STDコアCPUとの相違点

C33 STDコアCPUでは、ext付きのld.*命令でアドレス不整例外が発生したときの戻りアドレスはld.*命令のアドレスになり、例外処理からの復帰時にext命令が実行されません。
C33 ADVコアCPUでは、この場合の戻りアドレスがext命令(複数ある場合はその先頭)になりますので、復帰時にext命令が正しく実行されます。

5.6 多機能ext命令

多機能ext命令には、以下の機能があります。

ext	***%rs***	オペランド拡張
ext	***%rs*,*op*,*imm2***	即値拡張およびポストシフト
ext	***op*,*imm2***	ポストシフト
ext	***cond***	条件付き命令スキップ

5.6.1 ext %rs

ALU命令、レジスタ間接転送命令を、汎用レジスタを3個使用した3オペランド命令にします。ext *%rs*命令はプリフェッチ機能によってプリデコードされ、ext命令に続くALU命令などの第3のオペランドを設定します。
このext命令は、下記の命令に有効です。

```
ld.b    %rd,[%rb]       ld.b    [%rb],%rs
ld.b    %rd,[%rb]+      ld.b    [%rb]+,%rs
ld.ub   %rd,[%rb]
ld.ub   %rd,[%rb]+
ld.h    %rd,[%rb]       ld.h    [%rb],%rs
ld.h    %rd,[%rb]+      ld.h    [%rb]+,%rs
ld.uh   %rd,[%rb]
ld.uh   %rd,[%rb]+
ld.w    %rd,[%rb]       ld.w    [%rb],%rs
ld.w    %rd,[%rb]+      ld.w    [%rb]+,%rs
and     %rd,%rs         or      %rd,%rs         xor     %rd,%rs
add     %rd,%rs         adc     %rd,%rs         add     %rd,%dp
sub     %rd,%rs         sbc     %rd,%rs
srl     %rd,imm5        sll     %rd,imm5
srl     %rd,%rs         sll     %rd,%rs
sra     %rd,imm5        sla     %rd,imm5
sra     %rd,%rs         sla     %rd,%rs
rr      %rd,imm5        rl      %rd,imm5
rr      %rd,%rs         rl      %rd,%rs
```

ext *%rs*で拡張した場合の命令の動作を、表5.6.1.1に示します。

表5.6.1.1 拡張後の命令機能

対象命令		拡張命令		動　作	
ld.b	*%rd,[%rb1]*	ext	*%rb2*	*rd* ← B[*rb1* + *rb2*]	(符号付き)
ld.b	*[%rb1],%rs*	ext	*%rb2*	B[*rb1* + *rb2*] ← *rs*	(符号付き)
ld.b	*%rd,[%rb1]+*	ext	*%rb2*	*rd* ← B[*rb1*]; *rb1* ← *rb1* + *rb2*	(符号付き)
ld.b	*[%rb1]+,%rs*	ext	*%rb2*	B[*rb1*] ← *rs*; *rb1* ← *rb1* + *rb2*	(符号付き)
lb.ub	*%rd,[%rb1]*	ext	*%rb2*	*rd* ← B[*rb1* + *rb2*]	(符号なし)
lb.ub	*%rd,[%rb1]+*	ext	*%rb2*	*rd* ← B[*rb1*]; *rb1* ← *rb1* + *rb2*	(符号なし)
ld.h	*%rd,[%rb1]*	ext	*%rb2*	*rd* ← H[*rb1* + *rb2*]	(符号付き)
ld.h	*[%rb1],%rs*	ext	*%rb2*	H[*rb1* + *rb2*] ← *rs*	(符号付き)
ld.h	*%rd,[%rb1]+*	ext	*%rb2*	*rd* ← H[*rb1*]; *rb1* ← *rb1* + *rb2*	(符号付き)
ld.h	*[%rb1]+,%rs*	ext	*%rb2*	H[*rb1*] ← *rs*; *rb1* ← *rb1* + *rb2*	(符号付き)
ld.uh	*%rd,[%rb1]*	ext	*%rb2*	*rd* ← H[*rb1* + *rb2*]	(符号なし)
ld.uh	*%rd,[%rb1]+*	ext	*%rb2*	*rd* ← H[*rb1*]; *rb1* ← *rb1* + *rb2*	(符号なし)
ld.w	*%rd,[%rb1]*	ext	*%rb2*	*rd* ← W[*rb1* + *rb2*]	
ld.w	*[%rb1],%rs*	ext	*%rb2*	W[*rb1* + *rb2*] ← *rs*	
ld.w	*%rd,[%rb1]+*	ext	*%rb2*	*rd* ← W[*rb1*]; *rb1* ← *rb1* + *rb2*	
ld.w	*[%rb1]+,%rs*	ext	*%rb2*	W[*rb1*] ← *rs*; *rb1* ← *rb1* + *rb2*	
and	*%rd,%rs1*	ext	*%rs2*	*rd* ← *rs1* & *rs2*	
or	*%rd,%rs1*	ext	*%rs2*	*rd* ←*rs1* | *rs2*	
xor	*%rd,%rs1*	ext	*%rs2*	*rd* ← *rs1* ^ *rs2*	
add	*%rd,%rs1*	ext	*%rs2*	*rd* ← *rs1* + *rs2*	
adc	*%rd,%rs1*	ext	*%rs2*	*rd* ← *rs1* + *rs2* + C	
add	*%rd,%*dp	ext	*%rs*	*rd* ← dp + *rs*	
sub	*%rd,%rs1*	ext	*%rs2*	*rd* ← *rs1* - *rs2*	
sbc	*%rd,%rs1*	ext	*%rs2*	*rd* ← *rs1* - *rs2* - C	
srl	*%rd,imm5*	ext	*%rs*	*rd* ← *rs* >> *imm5*; *rd*[31] ← 0	
srl	*%rd,%rs2*	ext	*%rs1*	*rd* ← *rs1* >> *rs2*; *rd*[31] ← 0	
sll	*%rd,imm5*	ext	*%rs*	*rd* ← *rs* << *imm5*; *rd*[0] ← 0	
sll	*%rd,%rs2*	ext	*%rs1*	*rd* ← *rs1* << *rs2*; *rd*[0] ← 0	
sra	*%rd,imm5*	ext	*%rs*	*rd* ← *rs* >> *imm5*; *rd*[31] ← *rs*[31]	
sra	*%rd,%rs2*	ext	*%rs1*	*rd* ← *rs1* >> *rs2*; *rd*[31] ← *rs1*[31]	
sla	*%rd,imm5*	ext	*%rs*	*rd* ← *rs* << *imm5*; *rd*[0] ← 0	
sla	*%rd,%rs2*	ext	*%rs1*	*rd* ← *rs1* << *rs2*; *rd*[0] ← 0	
rr	*%rd,imm5*	ext	*%rs*	*rd* ← *rs* >> *imm5*; *rd*[31] ← *rs*[0]	
rr	*%rd,%rs2*	ext	*%rs1*	*rd* ← *rs1* >> *rs2*; *rd*[31] ← *rs1*[0]	
rl	*%rd,imm5*	ext	*%rs*	*rd* ← *rs* << *imm5*; *rd*[0] ← *rs*[31]	
rl	*%rd,%rs*	ext	*%rs1*	*rd* ← *rs1* << *rs2*; *rd*[0] ← *rs1*[31]	

5.6.2 ext %rs,op,imm2

%rsによるオペランドの拡張に加え、ポストシフト機能をext命令に続くadd命令またはsub命令に指示します。オペランドにポストシフトが指定されると、add/sub命令の演算結果を右または左にシフトします。本機能は下記の命令にのみに有効で、他の命令で使用した場合は“*op, imm2*”のシフト指定は無視され、ext %rs命令として機能します。

“ext *%rs*,*op*,*imm2*”が有効に機能する命令群

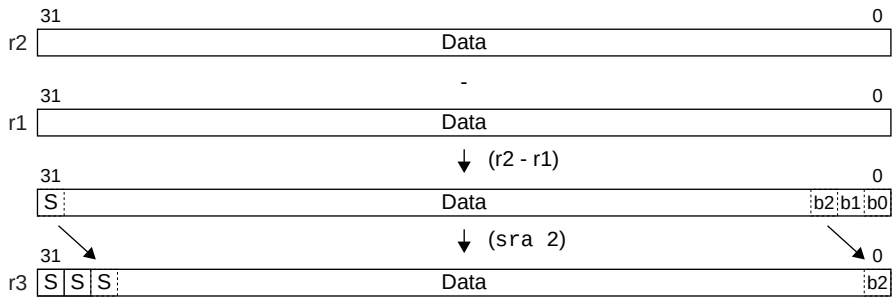
```
add  %rd,%rs      adc  %rd,%rs      add  %rd,%dp
sub  %rd,%rs      sbc  %rd,%rs
```

ポストシフトのシフト動作は、*op*で与えられるシフト命令と*imm2*で与えられるシフト量で決定します。*op*には表5.6.2.1の命令が指定できます。シフト量は最大3ビットです。

表5.6.2.1 ポストシフトオペレーション

op	*imm2*	機　能
sra	0, 1, 2, 3	**右方向算術シフト** >> *imm2*, *rd*[31] ← *rd*[31]
srl	0, 1, 2, 3	**右方向論理シフト** >> *imm2*, *rd*[31] ← 0
sll	0, 1, 2, 3	**左方向論理シフト** << *imm2*, *rd*[0] ← 0

例: ext %r1,sra,2
　sub %r3,%r2　　r3 ← (r2 - r1) >> 2, r3[31:30] = S
　 :　　:



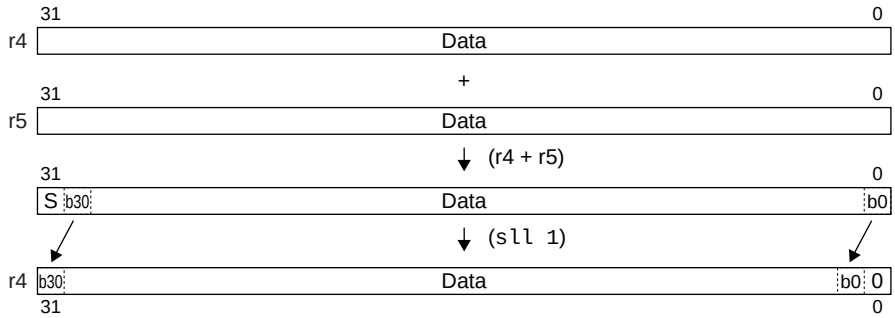
5.6.3 ext op,imm2

この命令は、次に続く命令にポストシフトの実行を指示します。動作は、5.6.2節のext *%rs*,*op*,*imm2*から*%rs*を省いたものになります。本機能は下記の命令にのみに有効です。

“ext *op*,*imm2*”が有効に機能する命令群

```
add  %rd,%rs   add  %rd,imm6   add  %sp,imm10   adc  %rd,%rs   add  %rd,%dp
sub  %rd,%rs   sub  %rd,imm6   sub  %sp,imm10   sbc  %rd,%rs
```

例: ext sll,1
　add %r4,%r5　　r4 ← (r4 + r5) << 1, r4[0] = 0
　 :　　:



5.6.4 ext cond

この命令は、*cond*が示すコンディションコード(PSR[3:0]のC、V、Z、Nフラグ)の状態によって、この命令に続く命令を実行しません。ただし、以下の分岐命令をこの命令の直後に置かないでください。無効となります。

条件実行が無効な命令群

```
jrgt   sign8          jrgt.d  sign8
jrge   sign8          jrge.d  sign8
jrlt   sign8          jrlt.d  sign8
jrle   sign8          jrle.d  sign8
jrugt  sign8          jrugt.d sign8
jruge  sign8          jruge.d sign8
jrult  sign8          jrult.d sign8
jrule  sign8          jrule.d sign8
jreq   sign8          jreq.d  sign8
jrne   sign8          jrne.d  sign8
jp     sign8          jp.d    sign8
jp     %rb            jp.d    %rb
jpr    %rb            jpr.d   %rb
call   sign8          call.d  sign8
call   %rb            call.d  %rb
ret                   ret.d
reti                  retd
retm                  int     imm2
brk                   slp
halt                  loop
repeat
```

ext *cond*命令では、下記の条件を指定可能です。コンディションコードが条件に一致すると命令を実行しません。

表5.6.4.1 条件

命 令	条 件
ext gt	!Z & !(N ^ V)
ext ge	!(N ^ V)
ext lt	N ^ V
ext le	Z | (N ^ V)
ext ugt	!Z & !C
ext uge	!C
ext ult	C
ext ule	Z | C
ext eq	Z
ext ne	!Z

例:
```
cmp   %r2,%r3
ext   eq
ld.w  %r4,%r5       r2 = r3の場合は実行しません。
```

5.6.5 ext命令の組み合わせ

即値拡張ext命令や多機能ext命令を組み合わせて使用することができます。

組み合わせることのできるext命令

1）ext *imm13*　　即値拡張1
（ext *imm13*）　　（即値拡張2）
nextop

2）ext *%rs*　　オペランド拡張
nextop

3）ext *op,imm2*　　ポストシフト
nextop

4）ext *%rs,op,imm2*　　オペランド拡張 + ポストシフト
nextop

5）ext *cond*　　条件付き実行
nextop

6）ext *cond*　　条件付き実行
ext *imm13*　　即値拡張1
（ext *imm13*）　　（即値拡張2）
nextop

7）ext *cond*　　条件付き実行
ext *op,imm2*
nextop

8）ext *cond*　　条件付き実行
ext *op,imm2*　　ポストシフト
ext *imm13*　　即値拡張1
（ext *imm13*）　　（即値拡張2）
nextop

9）ext *cond*　　条件付き実行
ext *%rs,op,imm2*　　オペランド拡張 + ポストシフト
nextop

10）ext *cond*　　条件付き実行
ext *%rs*　　オペランド拡張
nextop

11）ext *op,imm2*　　ポストシフト
ext *imm13*　　即値拡張1
（ext *imm13*）　　（即値拡張2）
nextop

例: cmp %r1,%r2　　r1とr2を比較
ext ne　　r1 != r2のときadd命令を実行しない
ext %r4,sll,3　　オペランド拡張 + ポストシフト
add %r5,%r6　　r5 = (r6 + r4) << 3

注:　上記の例の場合、ext ne命令の条件がコンディションコードと一致した場合、一連のext命令の後にあるｅｘｔ命令以外の命令（この例の場合はａｄｄ命令）が条件実行の対象になります。ｅｘｔ %r4,sll,3命令が対象にならないことに注意してください。

5.7 データ転送命令

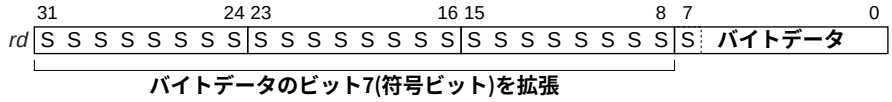
C33 ADVコアCPUの転送命令は、レジスタ～レジスタ間、レジスタ～メモリ間のデータ転送をサポートしています。転送データサイズとデータ拡張形式が命令コードで指定可能です。ニーモニック表記上では次のように分類されます。

ld.b 符号付きバイトデータ転送
ld.ub 符号なしバイトデータ転送
ld.h 符号付きハーフワードデータ転送
ld.uh 符号なしハーフワードデータ転送
ld.w ワードデータ転送

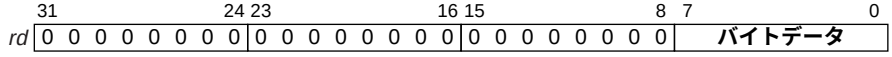
レジスタへの符号付きバイト/ハーフワード転送では、ソースデータが32ビットに符号拡張されます。符号なしバイト/ハーフワード転送では、ソースデータが32ビットにゼロ拡張されます。
レジスタをソースとする転送では、レジスタ内下位側の指定サイズ分が転送データとなります。

転送先が汎用レジスタのとき、転送後のレジスタの内容は次のようになります。

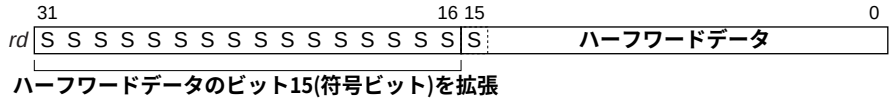
●符号付きバイトデータ転送



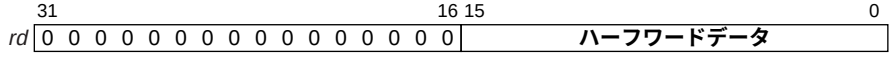
●符号なしバイトデータ転送



●符号付きハーフワードデータ転送



●符号なしハーフワードデータ転送



5.8 論理演算命令

C33 ADVコアCPUでは、4種類の論理演算命令が使用可能です。

and	論理積命令
or	論理和命令
xor	排他的論理和命令
not	否定命令

すべての論理演算は、指定の汎用レジスタ(R0～R15)に対して行われます。
ソースは汎用レジスタの32ビットデータか、符号付き即値データ(6 、19 、32ビット)の2種類です。

C33 STDコアCPUとの相違点

PSRのOCフラグ(ビット21)が1の場合、論理演算命令を実行するとPSRのVフラグ(ビット2)がクリアされます。

5.9 算術演算命令

C33 ADVコアCPU命令セットでは、算術演算用に加減算、比較、乗除算命令をサポートしています(乗除算命令は次節で説明します)。

add 加算命令
adc キャリー付き加算命令
sub 減算命令
sbc ボロー付き減算命令
cmp 比較命令

上記算術演算は、汎用レジスタ間(R0～R15)、汎用レジスタ～即値間で行われます。add命令、sub命令は、さらにSP～即値間の演算にも対応しています。cmp命令を除き、ワードサイズ以外の即値は演算の際にゼロ拡張されます。
cmp命令は2つのオペランドを比較する命令で、比較結果によりPSR内のフラグのみを変更します。基本的には条件ジャンプ命令の条件設定に使用します。ソースにワードサイズ以外の即値を指定した場合は、比較の際に符号拡張されます。

5.10 乗算・除算命令

C33 ADVコアCPUは、標準機能として乗除算に対応しています。

5.10.1 乗算命令

C33 ADVコアCPUの命令セットには5つの乗算命令が含まれています。

mlt.h 16ビット×16 ビット →32ビット(符号付き)
mltu.h 16ビット×16 ビット →32ビット(符号なし)
mlt.w 32ビット×32 ビット →64ビット(符号付き)
mltu.w 32ビット×32 ビット →64ビット(符号なし)
mlt.hw 32ビット×16 ビット →64ビット(64ビットに符号拡張)

乗数と被乗数はそれぞれ指定の汎用レジスタ(R0〜R15)のデータを使用します。16ビット乗算の場合は指定レジスタの下位16ビットが使用されます。符号付き乗算命令は乗数、被乗数のMSBを符号ビットとして扱います。
16ビット×16ビットの演算結果はALRにロードされます。32ビット×32ビットおよび32ビット×16ビットの演算結果は上位32ビットがAHRに、下位32ビットがALR にロードされます。
C33 ADVコアCPUは16ビット×16ビットの乗算を1サイクル、32ビット×32ビットの乗算を2サイクルで実行します。

C33 STDコアCPUとの相違点

- C33 ADVコアCPUには`mlt.hw`命令が追加されました。また、AHRとALRレジスタを一括クリアする`macclr`命令が追加されています。
- PSRのLCフラグ(ビット17)とHCフラグ(ビット16)を1にセットすることで、乗算結果がALR、AHRだけではなく、R4、R5にも書き込まれます。

5.10.2 除算命令

C33 ADVコアCPUは、符号付きおよび符号なしのステップ除算機能を持っています。

符号付きステップ除算に使用する命令: **div0s**, **div1**, **div2s**, **div3s**
符号なしステップ除算に使用する命令: **div0u**, **div1**

さらに、符号付き一括除算命令および符号なし一括除算命令も用意されています。

符号付き一括除算命令: **divs.w**
符号なし一括除算命令: **divu.w**

以下に、ステップ除算の実行手順と各命令の機能を示します。

1. ステップ除算の前処理(`div0s`, `div0u`)

除算を開始する前に、被除数をALR、除数を*rs*レジスタ(汎用レジスタR0〜R15)に用意し、`div0s`または`div0u`を実行します。それぞれの命令は次のような動作を行います。

`div0s`(符号付きステップ除算の前処理)

- AHRにALR(被除数)の符号を拡張
  被除数が正の場合、AHRは0x00000000、負の場合は0xFFFFFFFFに設定されます。
- PSRのDSフラグに被除数の符号ビットをセット
  被除数が正の場合、DSフラグは0にリセット、負の場合は1にセットされます。
- PSRのNフラグに除数(*rs*)の符号ビットをセット
  除数が正の場合、Nフラグは0にリセット、負の場合は1にセットされます。

div0u(符号なしステップ除算の前処理)
- AHRを0x00000000にクリア
- PSRのDSフラグを0にリセット
- PSRのNフラグを0にリセット

※ 被除数が32ビット未満の場合、ALRに格納する際に上詰めにし、div1の実行回数をビット数と合わせることで、より短いサイクルで除算ができます(後述する「一括除算」では、この手法を採ることはできません)。
例: 被除数が0x55の場合
A) ALRに0x55000000を格納し、div1を8回実行
B) ALRに0x00550000を格納し、div1を16回実行
C) ALRに0x00000055を格納し、div1を32回実行
A、B、Cはいずれも同じ計算結果になります。

2. ステップ除算を実行

div1命令を必要ステップ数、実行します。たとえば、32ビット÷32ビットの場合はdiv1命令を32回実行します(上記※の説明のとおり、被除数のビット数次第で回数を調整)。div1命令は符号付き除算、符号なし除算に共通です。
div1命令の1回の実行では、以下の処理が行われます。

(1) {AHR, ALR}の64ビットを左(上位側)に1ビットシフト(ALR[0] = 0)

(2) AHRと*rs*を加算またはAHRから*rs*を減算し、結果によりAHR、ALRを再設定
加減算はAHRの内容にDSフラグを符号ビットとして付加した33ビットと、*rs*レジスタの内容にNフラグを符号ビットとして付加した33ビットデータによって行います。
この処理はPSRのDSフラグおよびNフラグの状態によって以下のように異なります。なお、演算結果の33ビット目の値をtmp[32]として説明します。

DS = 0(被除数: 正)、N = 0(除数: 正)の場合
- tmp = {0, AHR} - {0, *rs*}を実行
- tmp[32] = 1の場合: AHR = tmp[31:0]、ALR[0] = 1として終了
  tmp[32] = 0の場合: AHR、ALRをそのままに終了

DS = 1(被除数: 負)、N = 0(除数: 正)の場合
- tmp = {1, AHR} + {0, *rs*}を実行
- tmp[32] = 0の場合: AHR = tmp[31:0]、ALR[0] = 1として終了
  tmp[32] = 1の場合: AHR、ALRをそのままに終了

DS =0(被除数:正)、N =1(除数:負)の場合
- tmp = {0, AHR} + {1, *rs*}を実行
- tmp[32] = 1の場合: AHR = tmp[31:0]、ALR[0] = 1として終了
  tmp[32] = 0の場合: AHR、ALRをそのままに終了

DS = 1(被除数:負)、N = 1(除数:負)の場合
- tmp = {1, AHR} - {1, *rs*}を実行
- tmp[32] = 0の場合: AHR = tmp[31:0]、ALR[0] = 1として終了
  tmp[32] = 1の場合: AHR、ALRをそのままに終了

符号なし除算の場合は、div1命令を必要回数実行することにより、結果が以下のレジスタから得られます。
AHR　余り
ALR　商

符号付き除算の場合は、次に説明する補正が必要です。

3. 符号付き除算の補正

符号付き除算の場合、div1命令を必要ステップ数実行後、div2s命令とdiv3s命令を続けて実行し、演算結果を補正します。
符号なし除算では、div2s命令とdiv3s命令を実行する必要はありません。なお、実行した場合でもnop命令と同様に機能し、演算結果には影響を与えません。
以下に、div2s命令とdiv3s命令の機能を示します。

div2s(符号付きステップ除算結果の補正1)

被除数が負の数の場合に除算のステップ(div1命令の実行)で演算結果がゼロになると、全ステップ終了後の演算結果が、除数と同じ余り(AHR)、および実際の値よりも絶対値で1 少ない商(ALR)となる可能性があります。div2s命令はこの結果を補正します。
div2s命令の動作は次のとおりです。

DS = 0(被除数: 正)の場合
被除数が正の場合は上記の問題は発生しません。したがって、div2s命令は何も実行せずに終了します(nop命令と同じ)。

DS = 1(被除数: 負)の場合
(1) N = 0(除数: 正)の場合: tmp = AHR + *rs*を実行
N = 1(除数: 負)の場合: tmp = AHR - *rs*を実行
(2) (1)の演算結果により、
tmpがゼロの場合: AHR = tmp[31:0]、ALR = ALR + 1として終了
tmpがゼロ以外の場合: AHR、ALRをそのままに終了

div3s(符号付きステップ除算結果の補正2)

ステップ除算によりALR から得られる商は常に正の数になります。被除数と除数の符号が異なる場合、結果は負でなければなりません。div3s命令はこの場合の符号補正を行います。

DS = N(被除数と除数が同符号)の場合
この場合は上記の問題は発生しません。したがって、div3s命令は何も実行せずに終了します(nop命令と同じ)。

DS ≠ N(被除数と除数の符号が異なる)の場合
ALR(商)の符号を反転します。

div2s命令およびdiv3s命令実行後、符号付き除算の最終結果が以下のレジスタから得られます。
AHR 余り
ALR 商

C33 ADVコアCPUでは、この一連のステップ除算を一括演算する命令を新規に追加しました。
一括除算命令は、符号付きと符号なし除算命令があり、32ビット÷32ビットの演算を1命令で実行します。この一括除算命令で使用する入力レジスタおよび出力レジスタはステップ除算と同様で、被除数をALR、除数を*rs*にロードして実行します。除算の結果は、商がALR、余りがAHRから得られます。

C33 STDコアCPUとの相違点

符号付き一括除算命令と符号なし一括除算命令が追加されました。

div.w	符号付き一括除算命令
divu.w	符号なし一括除算命令

ステップ除算の実行例

(1) 符号付き32ビット÷32ビットの実行
(被除数がR0、除数がR1にロードされている場合)

```
ld.w    %alr,%r0   ; ALRに被除数を設定
div0s   %r1        ; 初期化ステップ
div1    %r1        ; ステップ除算
 :       :
div1    %r1        ; div1命令を32回実行
div2s   %r1        ; 補正命令1
div3s              ; 補正命令2
```

AHRに余り、ALRに商がロードされます。
この例の実行にかかる時間は35サイクルです。
符号付き除算の場合、除算結果の余りの符号は被除数と同じになります。
例: (-8)÷ 5 = - 1 余り-3
8 ÷ (-5) = -1 余り3

(2) 符号なし32ビット÷32ビットの実行
(被除数がR0、除数がR1にロードされている場合)

```
ld.w    %alr,%r0   ; ALRに被除数を設定
div0u   %r1        ; 初期化ステップ
div1    %r1        ; ステップ除算
 :       :
div1    %r1        ; div1命令を32回実行
```

AHRに余り、ALRに商がロードされます。
この例の実行にかかる時間は33サイクルです。

一括除算の実行例

(1) 符号付き32ビット÷32ビットの実行
(被除数がR0、除数がR1にロードされている場合)

```
ld.w    %alr,%r0   ; ALRに被除数を設定
div.w   %r1        ; 符号付き一括除算の実行
```

AHRに余り、ALRに商がロードされます。
この例の実行にかかる時間は35サイクルです。

(2)符号なし32ビット÷32ビットの実行
(被除数がR0、除数がR1にロードされている場合)

```
ld.w    %alr,%r0   ; ALRに被除数を設定
divu.w  %r1        ; 符号なし一括除算の実行
```

AHRに余り、ALRに商がロードされます。
この例の実行にかかる時間は35サイクルです。

C33 STDコアCPUとの相違点

PSRのLCフラグ(ビット17)とHCフラグ(ビット16)を1にセットすることで、除算結果がALR、AHRだけではなく、R4、R5にも書き込まれます。

5.11 積和演算命令

C33 ADVコアCPUは、以下の演算を指定回数実行する積和演算機能をサポートしています。

mac ***%rs***	16ビット×16ビット+64ビット→64ビット
mac.hw ***%rs***	32ビット×16ビット+64ビット→64ビット
mac.w ***%rs***	32ビット×32ビット+64ビット→64ビット

この機能により、専用のDSPを外部に設けることなく、デジタル信号処理をオンチップで実現します。
積和演算は`mac`命令、`mac.hw`命令または`mac.w`命令によって実行します。

`mac %rs`命令は、
(H[<*rs*+1>]+) × (H[<*rs*+2>]+) + {AHR, ALR} → {AHR, ALR}　　（H: ハーフワード）

`mac.hw %rs`命令は、
(W[<*rs*+1>]+) × (H[<*rs*+2>]+) + {AHR, ALR} → {AHR, ALR}　　（W: ワード）

`mac.w %rs`命令は、
(W[<*rs*+1>]+) × (W[<*rs*+2>]+) + {AHR, ALR} → {AHR, ALR}

の演算を*rs*レジスタで指定される回数分実行します。

*rs*レジスタには、積和演算開始前に積和演算の演算回数を設定しておきます。
*rs*レジスタは演算回数カウンタとして使用され、演算ごとにデクリメントされます。*rs*レジスタが0になると、積和演算命令は終了します。したがって、2^{32} - 1回(4,294,967,295回)までの積和演算が可能です。
ただし、*rs*レジスタに0を設定して積和演算命令を実行しても積和演算は行われず、AHR、ALRも変更されません。*rs*レジスタも0のままデクリメントされません。

<*rs*+1>と<*rs*+2>は*rs*レジスタに続く2つの汎用レジスタです。
例: *rs*にR0レジスタを指定した場合、<*rs*+1> = R1レジスタ、<*rs*+2> = R2レジスタ
　　*rs*にR15レジスタを指定した場合、<*rs*+1> = R0レジスタ、<*rs*+2> = R1レジスタ

H[<*rs*+1>]+、H[<*rs*+2>]+は、上記のレジスタの内容をベースアドレスとして指定されるメモリのハーフワードデータを表します。
積和演算命令は、それらを符号付き16ビットデータとして乗算し、結果を{AHR, ALR}レジスタペアに加算します。“+”は1回の積和演算ごとに、それぞれのベースアドレス(<*rs*+1>と<*rs*+2>レジスタの内容)がインクリメント(ハーフワードの場合は+2、ワード場合は+4)されることを示します。
例: R0 = 16、R1 = 0x100、R2 = 0x120、AHR = ALR = 0に設定し、`mac %r0`を実行
　1) {AHR, ALR} = 0 + H[0x100] × H[0x120]
　2) {AHR, ALR} = {AHR, ALR} + H[0x102] × H[0x122]
　3) {AHR, ALR} = {AHR, ALR} + H[0x104] × H[0x124]
　　:
　16) {AHR, ALR} = {AHR, ALR} + H[0x11E] × H[0x13E]

　演算結果は、AHRを上位32ビット、ALRを下位32ビットとする符号付き64ビットデータとして得られます。レジスタの値は、R0 = 0、R1 = 0x120、R2 = 0x140となります。

W[<*rs*+1>]+は、ワードデータのベースアドレスです。ロードするデータは符号付きの32ビットデータとして扱われ、データをロードするごとに+4されます。

積和演算中のオーバーフロー

積和演算中に演算結果が符号付き64ビットの範囲を越えると、オーバーフローとしてPSRのMOフラグが1にセットされます。この場合でも、*rs*レジスタに設定した回数を終了するまで演算は継続されます。MOフラグは、ソフトウェアによってリセットするまで1を保持します。mac命令の実行終了後にMOフラグを読み出すことで、演算結果が有効かどうかチェックできます。

積和演算中の割り込み

mac命令の実行中は、繰り返しの途中であっても割り込みを受け付けます。
割り込み処理ルーチンに分岐する際、スタックには実行中のmac命令のアドレスがリターンアドレスとしてセーブされます。したがって、割り込み処理ルーチンをreti命令で終了すると、中断していたmac命令の実行を再開します。ただし、その時点の*rs*レジスタの内容が残りのカウント数となりますので、割り込み処理ルーチン中で*rs*レジスタの内容が変更されると、当初設定した回数とは異なる結果となります。同様に、<*rs*+1>、<*rs*+2>レジスタの値が割り込み処理ルーチン中で変化すると、再開したmac命令は正しく実行されません。

C33 STDコアCPUとの相違点

- C33 ADVコアCPUにはmac.hw命令とmac.w命令が追加されました。
  mac.hw %*rs*命令　32ビット×16ビット+64ビット → 64ビット
  mac.w %*rs*命令　32ビット×32ビット+64ビット → 64ビット
- PSRのLCフラグ(ビット17)とHCフラグ(ビット16)を1にセットすることで、積和演算結果がALR、AHRだけではなく、R4、R5にも書き込まれます。

5.12 1積和演算命令

C33 ADVコアCPUは、連続的に積和演算を行う命令(mac命令、mac.hw命令またはmac.w命令)のほかに汎用レジスタにロードされたデータを1回だけ単独に積和演算を行う命令をサポートしています。

mac1.h	***%rd,%rs***	16ビット×16ビット+64ビット→64ビット
mac1.hw	***%rd,%rs***	32ビット×16ビット+64ビット→64ビット
mac1.w	***%rd,%rs***	32ビット×32ビット+64ビット→64ビット

演算結果は、AHRを上位32ビット、ALRを下位32ビットとする符号付き64ビットデータとして得られます。AHRおよびALRは積和演算を行う前にイニシャライズする必要があります。AHR/ALRおよびPSRのMOフラグを0に初期設定するには、macclr命令を実行します。
これらの命令は、任意の回数繰り返すことでmac命令と同様の結果を得ることができますが、他の演算命令と組み合わせることでより複雑な計算を行うことができます。

C33 STDコアCPUとの相違点

- 1積和演算命令はC33 ADVコアCPUからサポートされました。
- PSRのLCフラグ(ビット17)とHCフラグ(ビット16)を1にセットすることで、1積和演算の結果がALR、AHRだけではなく、R4、R5にも書き込まれます。

5.13 シフト/ローテート命令

C33 ADVコアCPUの命令セットは、レジスタデータのシフト、ローテート命令をサポートしています。

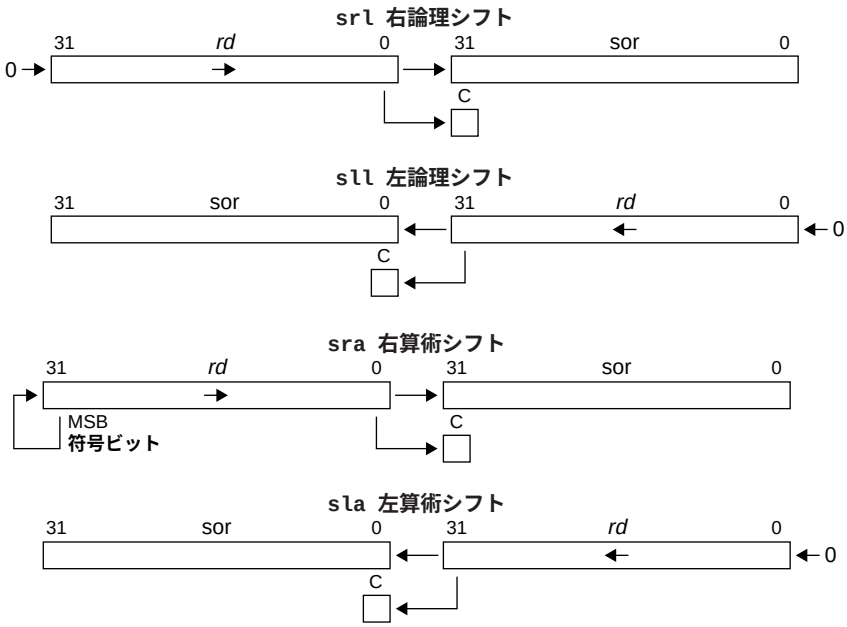
srl 右論理シフト
sll 左論理シフト
sra 右算術シフト
sla 左算術シフト
rr 右ローテート
rl 左ローテート

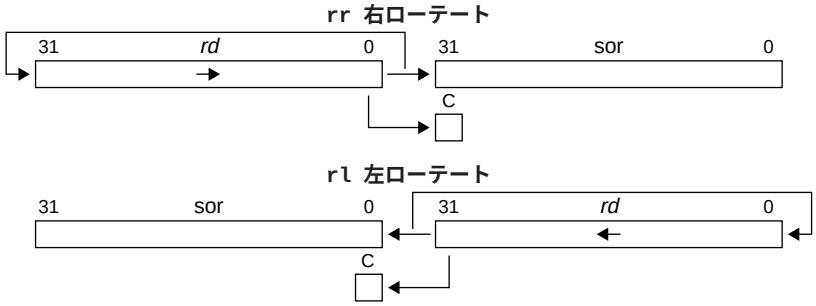
シフト量は、従来の8ビットから32ビットに拡張されています。32ビットシフトがサポートされたことによって命令も追加拡張されました。さらにシフト命令、ローテート命令で汎用レジスタからシフトアウトしたビットは特殊レジスタSORに格納されます。
シフト量の指定は、オペランドの*imm5*または*rs*レジスタで0～31の指定が可能です。
例: `srl %rd,imm5` 0～31ビットの右論理シフト
`srl %rd,%rs` 0～31ビットの右論理シフト

また、PSRのSE(ビット20)を1にセットすると、C(carry)フラグとV(overflow)フラグが変化するようになります。Cフラグには、LSBまたはMSBから最後にシフトアウトしたビットが入ります。Vフラグは、シフト終了時のCフラグとN(negative)フラグによって表5.13.1のように変化します。

表5.13.1 Vフラグの変化

Cフラグ	Nフラグ	Vフラグ
0	0	0
1	1	0
0	1	1
1	0	1





*rs*レジスタまたは*imm5*で指定するシフト量は次のとおりです。

表5.13.2 *imm5*、*rs*で指定するシフト量

imm5/*rs*[5:0]	シフトビット数	*imm5*/*rs*[5:0]	シフトビット数
00000	0	10000	16
00001	1	10001	17
00010	2	10010	18
00011	3	10011	19
00100	4	10100	20
00101	5	10101	21
00110	6	10110	22
00111	7	10111	23
01000	8	11000	24
01001	9	11001	25
01010	10	11010	26
01011	11	11011	27
01100	12	11100	28
01101	13	11101	29
01110	14	11110	30
01111	15	11111	31

*rs*のビット5～31は使用しません。

5.14 ビット操作命令

メモリ上のデータをビット単位で操作するため、以下の4種類の命令が用意されています。これらの命令を使用することで、表示メモリやI/Oマップの制御ビットを直接変更することができます。

btst	**[%*rb*],*imm3***	指定ビットが0ならばZフラグをセット
bclr	**[%*rb*],*imm3***	指定ビットを0にクリア
bset	**[%*rb*],*imm3***	指定ビットを1にセット
bnot	**[%*rb*],*imm3***	指定ビットを反転(1 ↔ 0)

ビット操作は、*rb*(汎用)レジスタで指定されるメモリアドレスに対して行われます。*imm3*はそのアドレスにストアされているバイトデータのビット番号(ビット0～ビット7)を指定します。
これらの命令(btstを除く)により変更される内容は指定のビットのみですが、メモリアクセスはバイト単位のため、指定アドレスは書き換えられます。このため、ビットの書き込み動作により機能が有効となるようなI/O制御ビットが割り付けられているアドレスの操作には注意が必要です。

5.15 プッシュ/ポップ命令

汎用レジスタ、特殊レジスタの内容をスタックに一時退避させるため、また退避させたデータを元のレジスタに復帰させるため、プッシュ命令とポップ命令が用意されています。

プッシュ命令

pushn ***%rs***
push ***%rs***
pushs ***%ss***

pushn命令は、汎用レジスタ*rs*からR0レジスタまでを連続的にスタックにセーブします。
push命令は、*rs*で指定された汎用レジスタを単独にスタックへ退避します。
pushs命令は、*ss*で指定された特殊レジスタをALRまで連続的にスタックに退避します。ただし、*ss*で指定されたレジスタがPSRまたはSPの場合には単独の退避を行います。

ポップ命令

popn ***%rd***
pop ***%rd***
pops ***%sd***

popn命令は、スタックのデータを汎用レジスタR0から*rd*レジスタまで連続的に復帰します。
pop命令は、スタックのデータを*rd*で指定された汎用レジスタへ単独に復帰します。
pops命令は、スタックのデータを特殊レジスタALRから*sd*まで連続的に復帰します。ただし、*sd*で指定されたレジスタがPSRまたはSPの場合には単独の復帰を行います。

プッシュ命令とポップ命令は、同じレジスタ指定のものが対になっている必要があります。これらの命令は、退避/復帰するデータ数に従ってSPを変更します。プッシュ/ポップ命令のほかに、SPをベースアドレスとしたディスプレースメント付きレジスタ間接アドレッシング(`[%sp+imm6]`)のロード命令も用意されていますので、SPを基準にしたレジスタ個別のストア/ロードも行えます。ただし、この場合SPは変更されません。

レジスタには、各レジスタに割り振られたレジスタ番号があります(“2 レジスタ”参照)。汎用レジスタまたは特殊レジスタを連続プッシュする場合、*rs*または*ss*で指定されたレジスタからレジスタ番号の降順にスタックへデータを退避します。連続ポップの場合は、連続プッシュとは逆にR0またはALRから昇順に、指定されたレジスタまでデータを復帰します。また、連続プッシュ、連続ポップ命令実行中は、PSRのPMフラグ(ビット28)が1にセットされます。PMフラグ(ビット28)が1の状態でpushn、pushs、popnまたはpops命令を実行すると、RC[3:0](ビット27～24)に格納されているレジスタ番号のレジスタから連続プッシュ/連続ポップ動作を行います。

pushs命令でUSP、SSP、PCレジスタを指定した場合には、存在しないレジスタである特殊レジスタ番号#12についても、メモリの書き込みおよびSPのデクリメントが一様に行われます。このとき、メモリに書き込まれるデータは不定となります。
pops命令でUSP、SSP、PCレジスタを指定した場合は存在しないレジスタである特殊レジスタ番号#12に対しても、メモリの読み出しおよびSPのインクリメントが一様に行われます。このとき、読み出したデータはどのレジスタにも反映されません。

pushn、pushs、popn、pops命令で連続的にプッシュ、ポップを繰り返してるときにも例外を受け付けることが可能です。これらの命令を実行中に例外を受け付けると、その命令のアドレスをリターンアドレスとしてスタックにセーブします。さらにPSRのRC[3:0]ビットにプッシュまたはポップを実行中のレジスタ番号を退避し、例外処理を行います。このとき、PSRのPMフラグ(ビット28)を0クリアして例外処理ルーチンのベクタアドレスへジャンプします。ただし、MMU例外およびデバッグ例外の場合は、PSRの退避およびPMフラグ(ビット28)の0クリアは行われませんので、それぞれ例外処理ルーチンの中でPSRのセーブを行ってからPSRのPMフラグ(ビット28)を0クリアしてください。

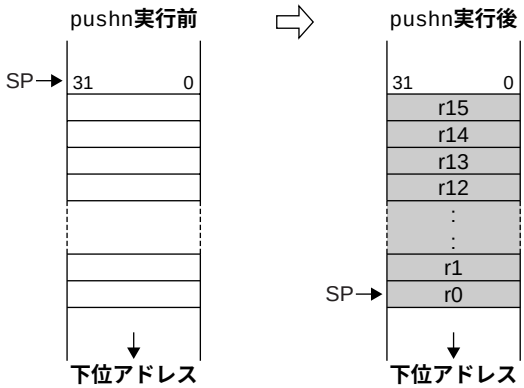
reti、retm、retd命令で例外処理から復帰すると、中断していたpushn、pushs、popnまたはpops命令に復帰しますが、命令を実行するときにPSRのPMフラグ(ビット28)が1になっていると、PSRのRC[3:0]ビットに格納されているレジスタ番号のレジスタからプッシュ/ポップ動作を再開します。命令コードのレジスタフィールドにあるレジスタ番号は参照されません。

pushn、pushs、popn、pops命令の実行の際には、PSRのPMフラグ(ビット28)が0であれば、命令コードに含まれるレジスタフィールドのレジスタ番号が参照され、1であればPSRのRC[3:0]ビットに退避されているレジスタ番号が参照されます。MMU例外、デバッグ例外はPSRをそのまま維持して例外処理に移行しますので、例外処理ルーチンで新たにpushn命令などを使ってレジスタのセーブを行うと、PSRのRC[3:0]ビットが参照され、正しく命令を実行できませんので注意してください。(上記、下線で示した処理を行う必要があります。)

C33 STDコアCPUとの相違点

- C33 ADVコアCPUには汎用レジスタの単独プッシュ/ポップ命令が追加されました。
  `push %ss` `pop %sd`
- C33 ADVコアCPUには特殊レジスタの連続プッシュ/ポップ命令が追加されました。
  `pushs %ss` `pops %sd`

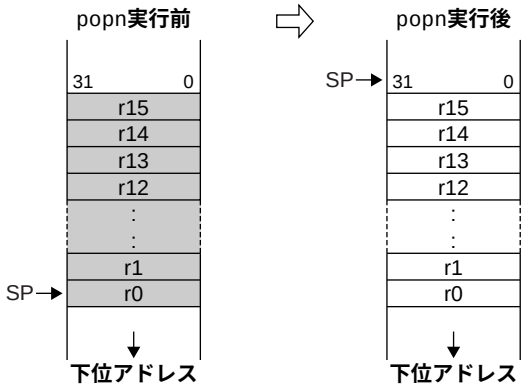
例1: `pushn %r15` 全汎用レジスタのプッシュ
`popn %r15` 全汎用レジスタのポップ



スタックポインタを更新してからレジスタのデータをプッシュします。

SP = SP - 4, *rs* → [SP]

図5.15.1 汎用レジスタの連続プッシュ



レジスタにデータをポップしてからスタックポインタを更新します。

[SP] → *rd*, SP = SP + 4

図5.15.2 汎用レジスタの連続ポップ

例2: pushs %dp　特殊レジスタの連続プッシュ
　　pops %dp　特殊レジスタの連続ポップ

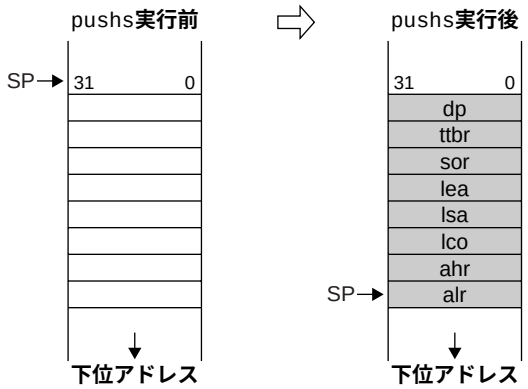


図5.15.3 特殊レジスタの連続プッシュ

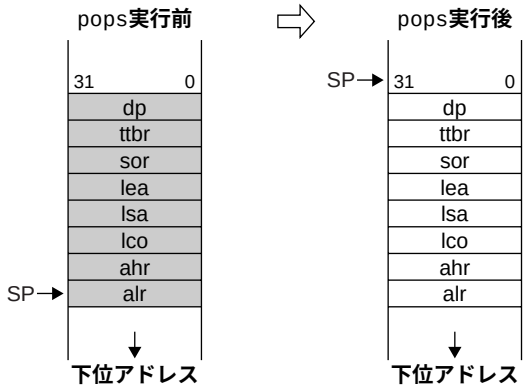


図5.15.4 特殊レジスタの連続ポップ

例3: push *%rs*　任意の汎用レジスタをプッシュ
　　pop *%rs*　任意の汎用レジスタをポップ

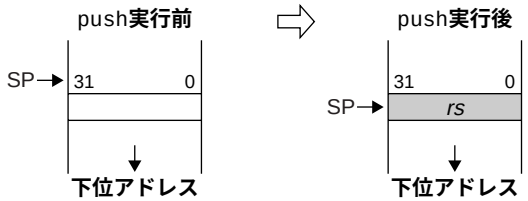


図5.15.5 汎用レジスタの単独プッシュ

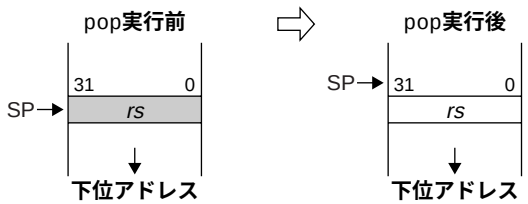


図5.15.6 汎用レジスタの単独ポップ

5.16 分岐命令/ディレイド命令

5.16.1 分岐命令の種類

(1)PC相対ジャンプ命令

PC相対ジャンプ命令には次のものがあります。

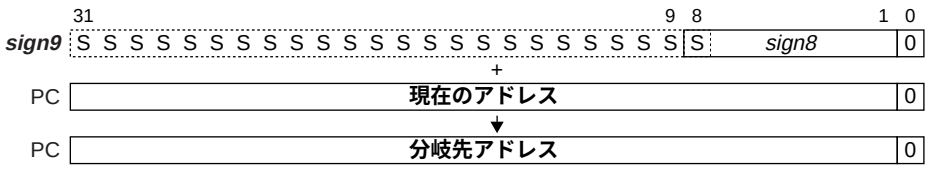
```
jr*  sign8
jp   sign8
jpr  %rb
```

PC相対ジャンプ命令はリロケータブルなプログラミングに対応した分岐命令で、現在のPCが示すアドレス(分岐命令のあるアドレス)にオペランドで指定した符号付きのディスプレースメントを加えたアドレスに分岐します。
*sign8*または*rb*で分岐先までの命令ステップ数を指定します。しかし、C33 ADVコアCPU内部では、命令長が16ビット固定のため、*sign8*または*rb*の値を2倍して16ビット単位のハーフワードアドレスとします。したがって、実際にPCに加算されるディスプレースメントは*sign8*を2倍にした符号付き9ビットとなり(最下位ビットは常に0)、偶数アドレスに分岐します。
指定可能なディスプレースメントはext命令による拡張も可能で、それぞれ次のようになります。

分岐命令単独の場合

```
jp   sign8    ; "jp sign9"として機能します。(sign9 = {sign8, 0})
```

分岐命令単独では、符号付き8ビットのディスプレースメント(*sign8*)が指定可能です。

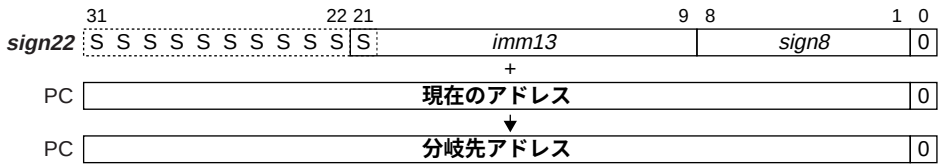


*sign8*は16ビット単位の相対値のため、分岐範囲は(PC - 256)～(PC + 254)です。

ext命令を1個拡張した場合

```
ext  imm13
jp   sign8    ; "jp sign22"として機能します。(sign22 = {imm13, sign8, 0})
```

ext命令で指定した*imm13*を*sign22*の上位13ビットとして拡張します。



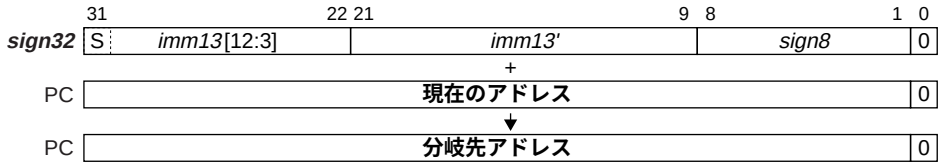
分岐範囲は(PC - 2,097,152)～(PC + 2,097,150)です。

ext命令を2個拡張した場合

```
ext   imm13
ext   imm13'
jp    sign8    ;"jp sign32"として機能します。
```

最初のext命令で指定した*imm13*はビット12～3の10ビットのみが有効(下位3ビットを無視)で、*sign32*は次のように構成されます。

sign32 = {*imm13*[12:3], *imm13'*, *sign8*, 0}



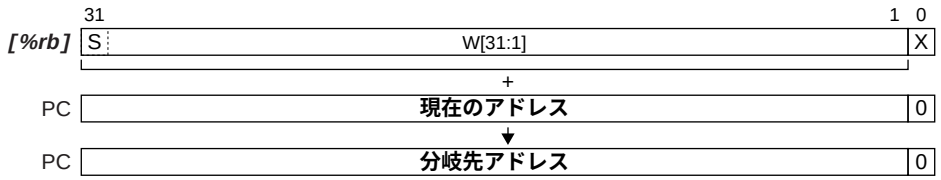
分岐範囲は(PC - 2,147,483,648)～(PC + 2,147,483,646)です。
上記の分岐範囲は論理的な値で、実際は使用するメモリ領域の範囲に制限されます。

jpr分岐の場合

```
jpr   %rb
```

*rb*で符号付き32ビットの相対値を指定可能です。
分岐アドレスは次のように構成されます。

{*rb*[31:1], 0}



*rb*レジスタの最下位ビットは常に0として扱われます。
分岐範囲は(PC - 2,147,483,648)～(PC + 2,147,483,646)です。
上記の分岐範囲は論理的な値で、実際は使用するメモリ領域の範囲に制限されます。

分岐条件

jp命令、jpr命令は常にプログラムが分岐する無条件ジャンプ命令です。
jrで始まる命令は、それぞれフラグの組み合わせによる分岐条件が設定されており、その条件が満たされている場合にのみ指定アドレスに分岐する条件ジャンプ命令です。条件が合っていない場合は分岐しません。条件ジャンプ命令は、基本的にcmp命令による2つの値の比較結果を判定するために使用します。このため、各命令の名称には大小関係を表す文字が使用されています。
条件ジャンプ命令の種類と分岐条件を表5.16.1.1に示します。

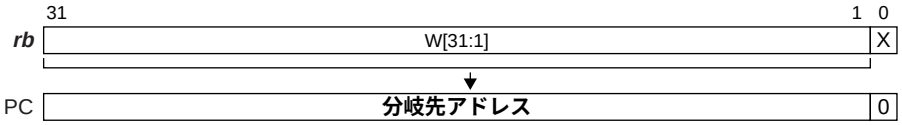
表5.16.1.1 条件ジャンプ命令と分岐条件

命　令		フラグ条件	A:Bの比較	備　考
jrgt	Greater Than	!Z & !(N ^ V)	A > B	符号付きデータ比較用
jrge	Greater or Equal	!(N ^ V)	A ≥ B	
jrlt	Less Than	N ^ V	A < B	
jrle	Less or Equal	Z | (N ^ V)	A ≤ B	
jrugt	Unsigned, Greater Than	!Z & !C	A > B	符号なしデータ比較用
jruge	Unsigned, Greater or Equal	!C	A ≥ B	
jrult	Unsigned, Less Than	C	A < B	
jrule	Unsigned, Less or Equal	Z | C	A ≤ B	
jreq	Equal	Z	A = B	
jrne	Not Equal	!Z	A ≠ B	

A:Bの比較は"cmp A,B"の場合です。

(2)絶対ジャンプ命令

絶対ジャンプ命令`jp %rb`は、指定の汎用レジスタ(*rb*)の内容を絶対アドレスとして無条件に分岐します。*rb*レジスタの内容がPCにロードされると、その最下位ビットは常に0となります。



(3)PC相対コール命令

PC相対コール命令`call sign8`はリロケータブルなプログラミングに対応したサブルーチンコール命令で、現在のPCが示すアドレス(分岐命令のアドレス)にオペランドで指定した符号付きのディスプレースメントを加えたアドレスから始まるサブルーチンへ無条件に分岐します。分岐時には、`call`の次の命令のアドレス(ディレイド分岐時は2つ目の命令のアドレス)をリターンアドレスとしてスタックにセーブします。サブルーチンの最後に`ret`命令を実行するとこのアドレスがPCにロードされ、サブルーチンからリターンします。
なお、命令長が16ビット固定のため、ディスプレースメントの最下位ビットは常に0として扱われ(*sign8*が2倍され)、偶数アドレスに分岐します。
指定可能なディスプレースメントは、PC相対ジャンプ命令と同様に`ext`命令による拡張も可能です。ディスプレースメントの拡張については前述の“(1)PC相対ジャンプ命令”を参照してください。

(4)絶対コール命令

絶対コール命令`call %rb`は、指定の汎用レジスタ(*rb*)の内容を絶対アドレスとして、そのアドレスから始まるサブルーチンを無条件にコールします。*rb*レジスタの内容がPCにロードされると、その最下位ビットは常に0となります。(“(2)絶対ジャンプ命令”参照)

(5)ソフトウェア例外

ソフトウェア例外`int imm2`は、ソフトウェアによって例外を発生させ、指定の例外処理ルーチンを実行するための命令です。4種類の例外処理ルーチンを作成することができ、*imm2*によってそれぞれのベクタ番号を指定します。CPUはソフトウェア例外が発生すると、PSRと`int`の次の命令アドレスをスタックにセーブし、ベクタテーブルから指定のベクタを読み出して例外処理ルーチンを実行します。したがって、例外処理ルーチンからのリターンにはPSRも復帰させる`reti`命令を使用する必要があります。
ソフトウェア例外の詳細については、“6.3 割り込みと例外”を参照してください。

(6)リターン命令

`ret`命令は`call`命令に対応するリターン命令で、スタックにセーブされているリターンアドレスをPCにロードしてサブルーチンを終了します。したがって、`ret`命令実行時のSPの値は、そのサブルーチンの実行開始時の値と同じ(リターンアドレスの位置を示している)でなければなりません。

`reti`命令は例外処理ルーチン用のリターン命令です。例外処理ではリターンアドレスとともにPSRもスタックにセーブされますので、`reti`命令によってPSRの内容を復帰させる必要があります。`reti`命令ではPC、PSRの順にスタックから読み出されます。`ret`命令と同様に、`reti`命令実行時と例外処理ルーチンの実行開始時のSPの値は同じでなければなりません。

(7)デバッグ例外

`brk`命令と`retd`命令はデバッグ例外処理ルーチンの呼び出しとリターンに使用します。基本的にはICEソフトウェア用の命令のため、アプリケーションプログラムでは使用しないでください。
これらの命令の機能については、“6.5 デバッグモード”を参照してください。

C33 STDコアCPUとの相違点

レジスタ間接相対分岐命令が追加されました。

```
jpr  %rb
```

5.16.2 ディレイド分岐命令

C33 ADVコアCPUは、パイプライン処理により命令の実行とフェッチを同時に行います。分岐命令実行時は続く命令がすでにフェッチされているため、分岐前にその命令を実行することによって分岐命令の実行サイクル数を1サイクル削減することができます。これがディレイド分岐機能で、分岐前に実行される命令(分岐命令の次のアドレスの命令)をディレイドスロット命令と呼びます。
ディレイド分岐機能が使用できる命令は以下のとおりで、ニーモニックでは分岐命令の後ろに".d"を付けて指定します。

ディレイド分岐命令

jrgt.d	**jrge.d**	**jrlt.d**	**jrle.d**	**jrugt.d**	**jruge.d**	**jrult.d**
jrule.d	**jreq.d**	**jrne.d**	**call.d**	**jp.d**	**ret.d**	**jpr.d**

ディレイドスロット命令

ディレイドスロット命令は以下の条件をすべて満たしていることが必要です。

- 1サイクル命令
- メモリをアクセスしない
- ext命令による拡張をしない

以下の命令はディレイドスロット命令として使用することができます。

```
ld.b    %rd,%rs          ld.ub   %rd,%rs
ld.h    %rd,%rs          ld.uh   %rd,%rs
ld.w    %rd,%rs          ld.w    %rd,sign6
ld.w    %sd,%rs          ld.w    %rd,%ss
add     %rd,%rs          add     %rd,imm6        add     %sp,imm10
add     %rd,%dp
adc     %rd,%rs
sub     %rd,%rs          sub     %rd,imm6        sub     %sp,imm10
sbc     %rd,%rs
mlt.h   %rd,%rs
mltu.h  %rd,%rs
cmp     %rd,%rs          cmp     %rd,sign6
and     %rd,%rs          and     %rd,sign6
or      %rd,%rs          or      %rd,sign6
xor     %rd,%rs          xor     %rd,sign6
not     %rd,%rs          not     %rd,sign6
srl     %rd,%rs          srl     %rd,imm5
sll     %rd,%rs          sll     %rd,imm5
sra     %rd,%rs          sra     %rd,imm5
sla     %rd,%rs          sla     %rd,imm5
rr      %rd,%rs          rr      %rd,imm5
rl      %rd,%rs          rl      %rd,imm5
scan0   %rd,%rs
scan1   %rd,%rs
swap    %rd,%rs          swaph   %rd,%rs
mirror  %rd,%rs
macclr
sat.b   %rd,%rs          sat.ub  %rd,%rs
sat.h   %rd,%rs          sat.uh  %rd,%rs
sat.w   %rd,%rs          sat.uw  %rd,%rs
psrclr  imm5
psrset  imm5
ld.c    %rd,imm4
ld.c    imm4,%rs
ld.cf
```

注: 上記の条件を満たさない命令は動作が不定となるため、ディレイドスロット命令として使用することは禁止します。

ディレイドスロット命令は、ディレイド分岐命令が条件付きか、あるいは無条件かにかかわらず、また分岐するしないにかかわらず必ず実行されます。
ディレイド分岐でない分岐命令(“.d”の付かないもの)では、分岐命令の次のアドレスの命令は、命令フローが分岐する場合は実行されませんが、条件分岐命令で分岐しなかった場合には、次のアドレスの命令が分岐命令に続く命令として実行されます。
call.d命令でスタックにセーブされるリターンアドレスはディレイドスロット命令の次の命令のアドレスとなり、サブルーチンからのリターン時にディレイド命令は実行されません。
ディレイド分岐命令とディレイドスロット命令の間は、割り込みや例外などはハードウェアによってマスクされ発生しません。

リーフサブルーチンへの応用

ディレイド分岐命令を利用した、リーフサブルーチンの高速コール機能への応用例を次に示します。
例:

```
   jp.d  SUB        ;ディレイド分岐命令でサブルーチンへジャンプ
   ld.w  %r8,%pc    ;ディレイドスロット命令でリターンアドレスを汎用レジスタにロード
   add   %r1,%r2    ;リターンアドレス
     :    :
SUB:
     :    :
   jp    %r8        ;リターン
```

注: ld.w *%rd*,%pc**命令はディレイドスロット命令として使用してください。ディレイド分岐命令の直後以外の場所に記述した場合、*rd*レジスタにロードされるPC値が**ld.w**の次の命令のアドレスを示すとは限りません。**

5.17 スキャン命令

スキャン命令は、指定の汎用レジスタのMSBからビットをスキャンして、最初に見つかった1または0のビットの位置を返します。
C33 ADVコアCPUのスキャン命令は、PSRのSWフラグ(ビット22)によって8ビットモードと32ビットモードに機能が分かれます。

PSR[22](SWフラグ) = 0: 8ビットスキャンモード
PSR[22](SWフラグ) = 1: 32ビットスキャンモード

scan0 %rd,%rs

*rs*レジスタをスキャンして、最初の0のビットの位置(MSBからのオフセット)を*rd*レジスタにロードします。PSRのSWフラグが0のときにはMSBから8ビットだけスキャンして、結果を*rd*レジスタに格納します。0が見つからない場合、8ビットスキャンモードでは“0x00000008”、32ビットスキャンモードのときには“0x00000020”が*rd*レジスタにロードされ、Cフラグがセットされます。

表5.17.1 scan0の機能

*rs*レジスタ MSB	*rd*レジスタ	フラグ C	V	Z	N
0	0x00000000	0	0	1	0
10	0x00000001	0	0	0	0
110	0x00000002	0	0	0	0
1110	0x00000003	0	0	0	0
1111 0	0x00000004	0	0	0	0
1111 10	0x00000005	0	0	0	0
1111 110	0x00000006	0	0	0	0
1111 1110	0x00000007	0	0	0	0
1111 1111 0	0x00000008	1*	0	0	0
1111 1111 10	0x00000009	0	0	0	0
:	:	:	:	:	:
1111 1111 1111 1111 1111 1111 1111 110	0x0000001e	0	0	0	0
1111 1111 1111 1111 1111 1111 1111 1110	0x0000001f	0	0	0	0
1111 1111 1111 1111 1111 1111 1111 1111	0x00000020	1	0	0	0

∗ 8ビットスキャンモードで上位8ビットに0がない場合、Cフラグがセットされます。

scan1 %rd,%rs

*rs*レジスタをスキャンして、最初の1のビットの位置(MSBからのオフセット)を*rd*レジスタにロードします。PSRのSWフラグが0のときにはMSBから8ビットだけスキャンして、結果を*rd*レジスタに格納します。1が見つからない場合、8ビットスキャンモードでは“0x00000008”、32ビットスキャンモードのときには“0x00000020”が*rd*レジスタにロードされ、Cフラグがセットされます。

表5.17.2 scan1の機能

*rs*レジスタ MSB	*rd*レジスタ	フラグ C	V	Z	N
1	0x00000000	0	0	1	0
01	0x00000001	0	0	0	0
001	0x00000002	0	0	0	0
0001	0x00000003	0	0	0	0
0000 1	0x00000004	0	0	0	0
0000 01	0x00000005	0	0	0	0
0000 001	0x00000006	0	0	0	0
0000 0001	0x00000007	0	0	0	0
0000 0000 1	0x00000008	1*	0	0	0
0000 0000 01	0x00000009	0	0	0	0
:	:	:	:	:	:
0000 0000 0000 0000 0000 0000 0000 001	0x0000001e	0	0	0	0
0000 0000 0000 0000 0000 0000 0000 0001	0x0000001f	0	0	0	0
0000 0000 0000 0000 0000 0000 0000 0000	0x00000020	1	0	0	0

∗ 8ビットスキャンモードで上位8ビットに1がない場合、Cフラグがセットされます。

C33 STDコアCPUとの相違点

PSRのSWフラグ(ビット22)によってスキャンモードが選択できるようになりました。
SW = 0: 8ビットスキャンモード
SW = 1: 32ビットスキャンモード

5.18 システム制御命令

以下の3つの命令はシステムを制御するもので、レジスタやメモリには影響を与えません。

nop　PCをインクリメントするのみで、他の動作を行いません。
halt　CPUをHALT モードにします。
slp　CPUをSLEEP モードにします。

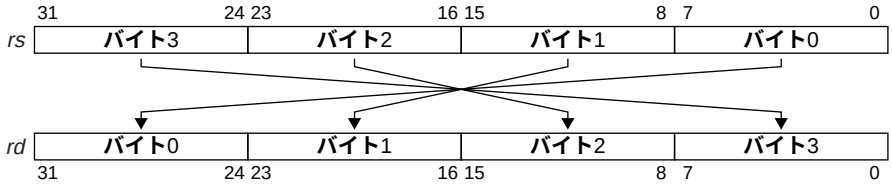
HALT モードとSLEEP モードについては、“6.4 パワーダウンモード”、および各機種のテクニカルマニュアルに記載のクロックマネージメントユニット(CMU)の説明を参照してください。

5.19 スワップ/ミラー命令

スワップ命令とミラー命令は汎用レジスタの内容を下図のように入れ替えます。

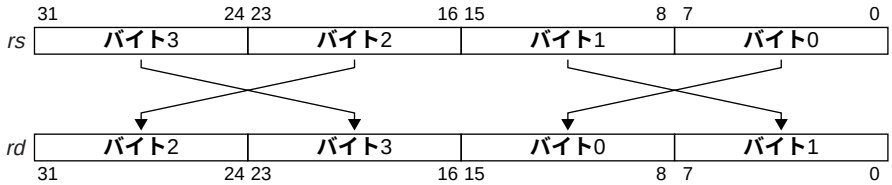
スワップ命令: swap *%rd,%rs*

ワード境界でビッグ/リトルエンディアン変換を行います。



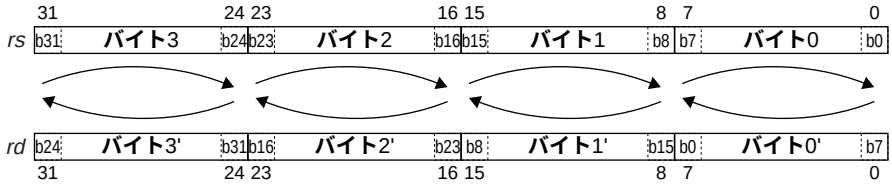
スワップ命令: swaph *%rd,%rs*

ハーフワード境界でビッグ/リトルエンディアン変換を行います。



ミラー命令: mirror *%rd,%rs*

バイト単位で上位/下位のビットの入れ替えを行います。

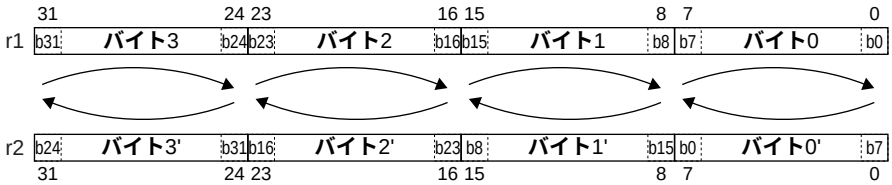


32ビットミラーの応用

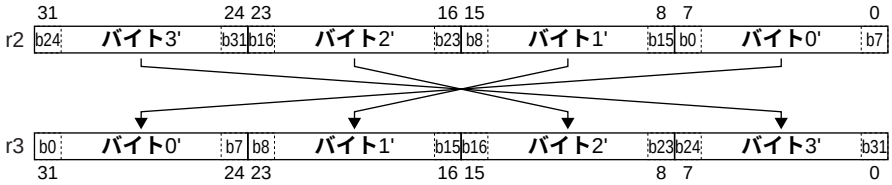
mirror命令とswap命令を組み合わせて32ビットのミラー処理を行うことができます。

例: `mirror  %r2,%r1` (1)
`swap  %r3,%r2` (2)

(1) mirror命令実行



(2) swap命令実行



C33 STDコアCPUとの相違点

swaph命令が追加されました。
`swaph  %rd,%rs`

5.20 飽和命令

C33 ADVコアCPUには、飽和機能付きデータ転送命令が用意されています。データを転送する際に転送元レジスタのデータに従って飽和処理を行い、転送先レジスタにデータをロードします。飽和処理が発生するとPSRのSフラグ(ビット15)が1にセットされます。飽和命令を以下に示します。

sat.b	***%rd,%rs***	符号付き飽和命令(バイト転送)
sat.ub	***%rd,%rs***	符号なし飽和命令(バイト転送)
sat.h	***%rd,%rs***	符号付き飽和命令(ハーフワード転送)
sat.uh	***%rd,%rs***	符号なし飽和命令(ハーフワード転送)
sat.w	***%rd,%rs***	符号付き飽和命令(ワード転送)
sat.uw	***%rd,%rs***	符号なし飽和命令(ワード転送)

符号付き飽和命令

バイトまたはハーフワードのサイズ指定がある場合、転送元レジスタのデータが次の範囲を超えていると飽和処理が発生します。

バイト指定　　+127 ～ -128
ハーフワード指定　+32,767 ～ -32,768

飽和処理が発生すると、転送先レジスタには32ビットに符号拡張された指定のデータサイズの最大値がロードされます。飽和処理が発生しない場合は、転送元レジスタの内容がそのまま転送先レジスタにロードされます。

バイトサイズの符号付き飽和命令(`sat.b`)の場合、以下の処理が行われます。

rs > +127 → +127 (0x0000007F)
rs < –128 → –128 (0xFFFFFF80)

例:

rs		*rd*
`0x00000012`	→	`0x00000012`
`0x12345678`	→	`0x0000007F`
`0xFFFFFFFA`	→	`0xFFFFFFFA`
`0xFFFFFABC`	→	`0xFFFFFF80`

ハーフワードの符号付き飽和命令(`sat.h`)の場合、以下の処理が行われます。

rs > +32,767 → +32,767 (0x00007FFF)
rs < –32,768 → –32,768 (0xFFFF8000)

例:

rs		*rd*
`0x00001234`	→	`0x00001234`
`0x12345678`	→	`0x00007FFF`
`0xFFFFABCD`	→	`0xFFFFABCD`
`0xFFABCDEF`	→	`0xFFFF8000`

飽和命令がワード指定(`sat.w`)の場合は、PSRのNフラグおよびVフラグをチェックして飽和処理を行います。次の条件が満たされたときに飽和処理が発生します。

N = 1かつV = 1のとき
　正の最大値"0x7FFFFFFF" → *rd*

N = 0かつV = 1のとき
　負の最大値"0x80000000" → *rd*

上記以外では飽和処理は発生せず、転送元レジスタの内容がそのまま転送先レジスタに送られます。

符号なし飽和命令

バイトまたはハーフワードのサイズ指定がある場合、転送元レジスタのデータが次の範囲を超えていると飽和処理が発生します。

バイト指定	255
ハーフワード指定	65,535

飽和処理が発生すると、転送先レジスタには指定のデータサイズの最大値がロードされ、ビット31～8は0拡張されます。飽和処理が発生しない場合は、転送元レジスタの内容がそのまま転送先レジスタにロードされます。

バイトサイズの符号なし飽和命令(`sat.ub`)の場合、以下の処理が行われます。

rs > 255 → 255 (0x000000FF)

例:

rs		*rd*
`0x00000012`	→	`0x00000012`
`0x12345678`	→	`0x000000FF`
`0xFFFFFFFA`	→	`0x000000FF`
`0xFFFFFABC`	→	`0x000000FF`

ハーフワードの符号付き飽和命令(`sat.uh`)の場合、以下の処理が行われます。

rs > 65535 → 65535 (0x0000FFFF)

例:

rs		*rd*
`0x00001234`	→	`0x00001234`
`0x12345678`	→	`0x0000FFFF`
`0xFFFFABCD`	→	`0x0000FFFF`
`0xFFABCDEF`	→	`0x0000FFFF`

飽和命令がワード指定(`sat.uw`)の場合は、PSRのCフラグをチェックして飽和処理を行います。次の条件が満たされたときに飽和処理が発生します。

C = 1のとき
0xFFFFFFFF → *rd*

上記以外では飽和処理は発生せず、転送元レジスタの内容がそのまま転送先レジスタに送られます。

いずれの場合も、飽和処理が発生するとPSRのSフラグ(ビット15)が1にセットされます。このフラグはソフトウェアにてクリアされるまで、1を保持します。

表5.20.1 飽和処理発生条件

命　令	条　件	動　作	Sフラグ
sat.b	*rb* > +127	0x0000007F → *rd*	1
	rb < -128	0xFFFFFF80 → *rd*	1
	+127 ≥ *rb* ≥ -128	*rs* → *rd*	–
sat.h	*rb* > +32,767	0x00007FFF → *rd*	1
	rb < -32,768	0xFFFF8000 → *rd*	1
	+32,767 ≥ *rb* ≥ -32,768	*rs* → *rd*	–
sat.w	N = 1 & V = 1	0x7FFFFFFF → *rd*	1
	N = 0 & V = 1	0x80000000 → *rd*	1
	その他	*rs* → *rd*	–
sat.ub	|*rb*| > 255	0x000000FF → *rd*	1
	|*rb*| ≤ 255	*rs* → *rd*	–
sat.uh	|*rb*| > 65,535	0x0000FFFF → *rd*	1
	|*rb*| ≤ 65,535	*rs* → *rd*	–
sat.uw	C = 1	0xFFFFFFFF → *rd*	1
	その他	*rs* → *rd*	–

例1: 演算命令と飽和処理

```
add    %r1,%r2
sat.b  %r1,%r1    ; -128～+127に飽和処理
  :       :
sub    %r4,0x3
sat.uw %r5,%r4    ; 0～65535に飽和処理
```

例2: シフト命令と飽和処理

Vフラグは左シフトでの飽和処理で有効(PSR[20] = 1 → Vフラグ変化)

```
// input  : %r12 shift num, %r13 shift data
// output : %r10 result
SL_SAT:
       loop   %r12,END
       sll    %r13,1
       sat.w  %r13,%r13
END:
       ret.d
       ld.w   %r10,%r13
```

Cフラグは右シフトでの飽和処理で有効(PSR[20] = 1 → Cフラグ変化)

```
// input  : %r12 shift num, %r13 shift data
// output : %r10 result
  SR_SAT:
       loop   %r12,END
       sra    %r13,1
       sat.w  %r13,%r13
  END:
       ret.d
       ld.w   %r10,%r13
```

5.21 リピート命令

5.21.1 設定

リピート動作とは、以下の2つの条件が満たされている場合に、LSAレジスタに設定したアドレスの命令をLCOレジスタに設定された値+1回実行することをいいます(LCOに0が設定されていても、1回は実行されます)。

(1) PSRのRMフラグ(ビット30)がセットされている

(2) LSAレジスタに設定したアドレスと現在のPCが一致

対象命令を実行するたびにLCOレジスタは0になるまで1ずつデクリメントされ、リピート動作が終了するとRMフラグは0にクリアされます。
リピート命令を実行すると、次の命令のアドレスとリピート回数がLSAレジスタとLCOレジスタにそれぞれロードされ、PSRのRMフラグ(ビット30)が1にセットされます。
また、LSA、LCO、RMフラグをそれぞれ個別に設定することによっても、リピート動作に入ることができます(この方法を「予約リピート」と呼びます)。予約リピートを行う場合の注意点は次のとおりです。

(1) RMフラグのセットは最後に行うこと

(2) RMフラグをセットする命令のアドレスとLSAに設定するアドレスとの間に1つ以上の命令があること
(ただし、RMフラグのセット直後に拡張命令が置かれている場合は、拡張分を除いてカウントしてください。)

リピート命令および予約リピートに共通する注意点は、以下のとおりです。

(1) RMフラグをセット後は、LSA、LCOの設定値を変更しないこと

(2) リピート不可命令(後述)をリピート命令の次、またはLSAアドレスに置かないこと

リピート命令が分岐命令より優れている点は、分岐先命令をフェッチし直す必要がないことです。分岐命令の場合、ディレイド命令の実行時を除き、すでにプリフェッチされている命令は分岐条件成立時にすべて破棄されて、分岐先の命令をフェッチし直さなければならないのに対し、リピート動作の場合にはその必要がありません。また、予約リピートを用いることにより、拡張命令を含んだ命令をリピート動作させることができます。
ただし、リピート命令は4クロック命令であり、予約リピートでもLCO、LSAのセットにそれぞれ最短で1クロック、PSRのセットに4クロックかかるため実質6クロック命令となり、リピート回数が極端に少ない場合(3、4回以下程度)には上記利点による高速化を打ち消してしまう可能性があります。

リピート命令のフォーマット

repeat ***%rc***	次に続く命令を、最大4G(0xFFFFFFFF)+1回実行します。
repeat ***imm4***	次に続く命令を、最大15+1回実行します。

例:
```
ld.w   %r1,0x0
   repeat 9              ; リピート命令実行
   ld.w   [%r2]+,%r1     ; 10回繰り返し実行
```

R2のアドレスから10ワード分(40バイト)の領域を0クリアします。

5.21.2 リピート動作のブレーク

リピート動作から途中で抜けるには、brk割り込みを発生させ、その処理の中でRMフラグ(PSRのビット30)をクリアします。

5.21.3 デバッグ例外およびMMU例外でのリピート動作の禁止

デバッグ例外およびMMU例外処理ルーチンではリピート動作を行わないでください。

5.21.4 リピート中の例外処理

リピート命令は、リピート中の例外を受け付けます。例外を受け付けると、リターンアドレスとしてリピート対象命令のアドレスをスタックに退避させた後、PSRもスタックに退避させます。さらにPSRのRMフラグ(ビット30)を0にクリアして例外処理ルーチンのベクタアドレスへジャンプします。ただし、MMU例外およびデバッグ例外の場合は、PSRの退避およびRMフラグの0クリアは行われませんので、それぞれ例外処理ルーチンの中で保護する必要があります。リピートの残り回数は特殊レジスタLCOに残っています。また、リピート命令で使用されるLCOレジスタ、LSAレジスタは例外処理からの復帰時に使用されます。
reti、retmおよびretd命令で例外処理ルーチンから復帰する際、PSRのRMフラグが1で、かつPCとLSAレジスタの値が一致したときに、リピート命令を再開します。LCOの値が残りのリピート回数となり、中断していたリピート対象命令を実行します。
例外処理ルーチンは、特殊レジスタLSA、LCOの値が変更されないように注意が必要です。LSA、LCOの値が変更されると例外処理ルーチンからの復帰後、正しくリピート動作が再開できません。例外処理ルーチンでrepeat命令やloop命令を使用する場合は、LSA、LCOの値はセーブしておく必要があります。ただし、デバッグ例外、MMU例外処理ルーチンではループ/リピート動作を行わないでください。

5.21.5 ループ、リピートの多重使用と割り込み

リピートの最中に発生した割り込み処理内で、さらにループ、リピートを使用したい場合には以下の手順を実行してください。ただし、デバッグ例外およびMMU例外ではループ/リピートを使用しないでください。

(1) LSA、LCOをレジスタもしくはメモリにセーブ

(1') 予約ループ/リピートを使用する場合にはここでRMフラグ(PSRのビット30)をクリア

(2) ループ/リピート命令あるいは予約ループ/リピートを実行

(3) ループ/リピート終了後、元のLSA、LCOをロード

(4) RMフラグをセット

割り込みでループ/リピートを使う場合も基本的には上記と同様です。ただし、MMU例外とデバッグ例外を除き、RMフラグは0クリアされるため、(1')の操作は省略可能です。また、reti命令にはPSRのロードが含まれるためRMフラグのセットも省略可能です。逆に、スタックされたRMフラグを強制的にクリアしても、リピート中の割り込みに対するreti命令は必ずRMフラグをセットして返すため、そのような使用方法は不可能です。

5.21.6 リピート不可命令

以下の命令はリピート対象命令となりません。動作は保証されませんので注意してください。

```
nop
slp
halt
pushn   %rs             pushs   %ss
popn    %rd             pops    %sd
brk
ret                     ret.d
retd                    reti                    retm
int     imm2
ext     imm13           ext     %rs             ext     cond
ext     op,imm2         ext     %rs,op,imm2
mac     %rs             mac.hw  %rs             mac.w   %rs
div.w   %rs             divu.w  %rs
repeat  imm4            repeat  %rb
jrgt    sign8           jrgt.d  sign8
jrge    sign8           jrge.d  sign8
jrlt    sign8           jrlt.d  sign8
jrle    sign8           jrle.d  sign8
jrugt   sign8           jrugt.d sign8
jruge   sign8           jruge.d sign8
jrult   sign8           jrult.d sign8
jrule   sign8           jrule.d sign8
jreq    sign8           jreq.d  sign8
jrne    sign8           jrne.d  sign8
jp      sign8           jp.d    sign8
jp      %rb             jp.d    %rb
jpr     %rb             jpr.d   %rb
call    sign8           call.d  sign8
call    %rb             call.d  %rb
loop    %rc,%ra         loop    %rc,imm4        loop    imm4,imm4
```

5.22 ループ命令

5.22.1 設定

ループ動作とは、以下の3つの条件を満たされている場合に、LSAレジスタに設定したアドレスに無条件分岐を行うことをいいます。

(1) PSRのLMフラグ(ビット29)がセットされている

(2) LEAレジスタに設定したアドレスと現在のPCが一致

(3) LCOレジスタ値が1以上

ループ命令を実行するとループ命令の次のアドレスがLSAレジスタにロードされ、第1オペランドで指定されるループ回数がLCOレジスタにロードされます。また、第2オペランドではループする最終アドレスが指定され、これはLEAレジスタにロードされます。このときループモードを表すPSRのLMフラグ(ビット29)が1になります。使用可能なループ命令は以下のとおりです。

```
loop  %rc,%ra
loop  %rc,imm4
loop  imm4,imm4
```

ループ命令の次のアドレスから順次命令が実行され、PCがループ最終アドレス(LEA)に一致すると実行アドレス(PC)がLSAの示すアドレスへ戻ります。このとき、LCOは-1されます。
この動作はLCOの値が0になるまで繰り返されます。

```
例:         loop  %r1,loop_end
            ld.w  %r2,[%r3]+
            ld.w  [%r4]+,%r2
    loop_end:
              :       :
```

上記例では、R3の示すアドレスからR4の示すアドレスへ、R1の値+1回分ワードデータをコピーします。
第1オペランドに*rc*を使用すると、ループ範囲の実行回数を最大4G+1回まで指定可能です。*imm4*で指定可能な回数は最大15+1回です。なお、ループ回数を0にしてもループ範囲は1回実行されます。
第2オペランドのループ最終アドレスは、*ra*で指定すると絶対アドレスになり、*imm4*で指定するとPC + 2からの相対アドレス(PC + 2 + *imm4* × 2)になります。

また、LEA、LSA、LCOの各レジスタ、およびLMフラグを個別に設定することによっても、ループ動作に入ることができます。(この方法を「予約ループ」と呼びます)。予約ループを行う場合の注意点は次のとおりです。

(1) LMフラグ(PSRのビット29)のセットは最後に行うこと

(2) LMフラグをセットするアドレスとLEAに設定するアドレスとの間が近すぎると正しくループ動作を行えません。以下の条件のどちらか1つを満たしてください。
- LMフラグをセットする命令とLEAに設定するアドレスとの間に分岐を入れること
- LMフラグをセットする命令とLEAに設定するアドレスとの間に5つ以上の命令があること
  (ただし、LMフラグのセット直後に拡張命令が置かれている場合は、拡張分を除いてカウントしてください。)

ループ命令および予約ループに共通する注意点は、以下のとおりです。

(1) LMフラグをセット後は、LCOがゼロになるまでLEA、LSA、LCOの設定値を変更しないこと

(2) ループ不可命令(後述)をLEAアドレス、またはLSA - 2のアドレスに置かないこと

ループが分岐命令より優れている点は、命令のプリフェッチ機能がループ動作条件を監視し、LEAアドレスの命令フェッチ後にフェッチ先アドレスをLSAアドレスに即時に戻すところです。つまり、ループ先の命令をプリフェッチし続けることが可能です。分岐命令の場合、ディレイド命令の実行時を除き、すでにプリフェッチされている命令は分岐条件成立時にすべて破棄されて、分岐先の命令をフェッチし直さなければならないのに対し、ループ動作の場合にはその必要がありません。
ただし、ループ命令は5クロック命令であり、予約ループでもLCO、LEA、LSAのセットにそれぞれ最短で1クロック、PSRのセットに4クロックかかるため実質7クロック命令となり、ループ回数が極端に少ない場合(3、4回以下程度)には上記利点による高速化を打ち消してしまう可能性があります。

なお、ループ命令で繰り返し実行する命令は、2命令以上必要です。1命令を繰り返す場合は、リピート命令を使用してください。

5.22.2 ループ動作のブレーク

ループを中止する場合は、`psrclr`命令を使ってPSRのLMフラグ(ビット29)をクリアしてから、分岐する必要があります(フラグのクリアから分岐命令までの間にループ動作を挟まないでください)。LMフラグをクリアしただけでは、次の命令としてLSAアドレスの命令がプリフェッチされている場合に誤動作してしまうため、分岐命令によりプリフェッチキューをクリアしてください。ただし、一旦分岐した後に再びこのループに戻ってループ命令を続行したい場合はクリアの必要はありません。ループモードは有効のままですので実行アドレス(PC)がLEAに一致して、LCOが0になっていなければループを続行します。

5.22.3 デバッグ例外およびMMU例外でのループ動作の禁止

デバッグ例外およびMMU例外処理ルーチンではループ動作を行わないでください。

5.22.4 ループ中の例外処理

`loop`命令は、ループ中の例外を受け付けます。例外を受け付けると、実行中の命令のアドレスをリターンアドレスとしてスタックに退避させた後、PSRを退避させます。さらにPSRのLMフラグ(ビット29)を0にクリアして例外処理ルーチンのベクタアドレスへジャンプします。ただし、MMU例外およびデバッグ例外の場合は、PSRの退避およびLMフラグの0クリアは行われませんので、それぞれ例外処理ルーチンの中で保護する必要があります。ただし、デバッグ例外、MMU例外処理ルーチンではループ/リピート動作は行わないでください。

5.22.5 ループ、リピートの多重使用と割り込み

ループの中でさらに別のループ、リピートを使用したい場合には以下の手順を実行してください。ただし、デバッグ例外、MMU例外処理ルーチンではループ/リピート動作を行わないでください。

(1) LSA、LEA、LCOをレジスタもしくはメモリにセーブ

(1') 予約ループ/リピートを使用する場合にはここでLMフラグ(PSRのビット29)をクリア

(2) ループ/リピート命令あるいは予約ループ/リピートを実行

(3) ループ/リピート終了後、元のLSA、LEA、LCOをロード

(4) LMフラグをセット

ただし、(4)と元のループのLEAとの関係は、前述の“予約ループを行う場合の注意点(2)”が適用されるため、多重ループ/リピートは`call`等で分岐したサブルーチン先で使用することを推奨します。
割り込みでループ/リピートを使う場合も基本的には上記と同様です。ただし、MMU例外とデバッグ例外を除き、LMフラグは0クリアされるため、(1')の操作は省略可能です。また、`reti`命令にはPSRのロードが含まれるためLMフラグのセットも省略可能です。逆に、スタックされたLMフラグを強制的にクリアしても、リピート中の割り込みに対する`reti`命令は必ずLMフラグをセットして返すため、そのような使用方法は不可能です。

5.22.6 命令使用上の制限

ループ中に記述する命令に関して、以下の制限があります。

● ループ実行中(LMフラグ(PSRのビット29)が1の間)は使用できない命令

```
ret (*1)                ret.d (*1)
reti                    retd (*1)               retm (*1)
repeat  imm4 (*2)       repeat  %rb (*2)
loop    %rc,%ra (*2)    loop    %rc,imm4 (*2)   loop    imm4,imm4 (*2)
```

(*1) 割り込みでループ/リピートを使用する際の注意事項を守れば使用可能
(*2) ループ/リピートを多重使用する際の注意事項を守れば使用可能

● loop命令で使用するレジスタが含まれると使用できない命令

```
pops    %sd
```

● LEAアドレス上では実行できない命令
(それ以外のアドレスの場合、ループ中でも実行可能)

```
brk
int     imm2
ext     %rs             ext     cond            ext     op,imm2
ext     %rs,op,imm2     ext     imm13
jrgt    sign8           jrge    sign8           jrlt    sign8
jrle    sign8           jrugt   sign8           jruge   sign8
jrult   sign8           jrule   sign8           jreq    sign8
jrne    sign8           jp      sign8
jp      %rb
jpr     %rb
call    sign8           call    %rb
slp                     halt
```

● LEAアドレスまたはその1命令前のアドレス上では実行できない命令
(それ以外のアドレスの場合、ループ中でも実行可能)

```
jrgt.d  sign8           jrge.d  sign8           jrlt.d  sign8
jrle.d  sign8           jrugt.d sign8           jruge.d sign8
jrult.d sign8           jrule.d sign8           jreq.d  sign8
jrne.d  sign8
jp.d    sign8           jp.d    %rb
jpr.d   %rb
call.d  sign8           call.d  %rb
```

5.23 その他の命令

フラグ制御命令

C33 ADVコアCPUでは、PSRのフラグを直接操作する命令を追加しました。フラグ制御命令はビット単位のセット/クリアが可能ですので、割り込みの許可/禁止などの制御をそれぞれ1命令で行うことができます。

psrset ***imm5*** *imm5*で指定されたPSRのビットを1にセットします。
psrclr ***imm5*** *imm5*で指定されたPSRのビットを0にクリアします。

6 機能

ここでは、C33 ADVコアCPUのCPU処理状態と動作の概要を説明します。

6.1 CPUの状態遷移

C33 ADVコアCPUの状態遷移を図6.1.1に示します。
C33 ADVコアCPUはスーパバイザモードとユーザモードをサポートし、リセット直後または内部/外部の例外が発生するとスーパバイザモードに移行します。

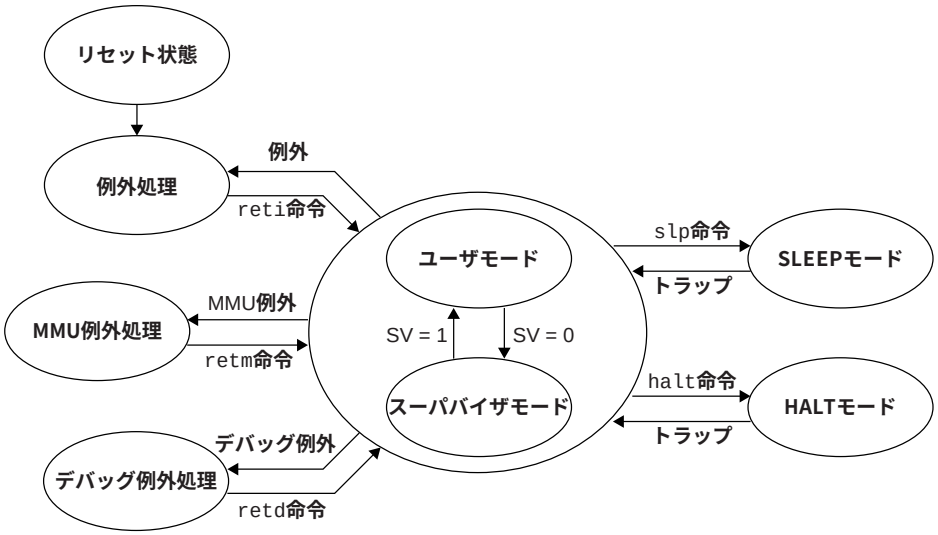


図6.1.1 CPU状態遷移図

6.1.1 リセット状態

リセット信号がアクティブ状態になるとCPUは初期化され、リセット信号がインアクティブになるとCPUはリセットベクタから処理を開始します。

6.1.2 スーパバイザモード

CPUのリセット後、または外部割り込みや例外が発生するとPSRのSVフラグ(ビット12)が0となり、CPUはスーパバイザモードに移行します。スーパバイザモードになるとCPUのすべてのリソースが使用可能となります。スタックはSSPの示す領域に切り換えられ、以後のスタック操作命令(push命令やpop命令など)はスタックポインタとしてSSPを使用します。

6.1.3 ユーザモード

スーパバイザモードの状態でPSRのSVフラグ(ビット12)を1にセットすると、CPUはユーザモードに移行します。また、例外発生時はスーパバイザモードに移行しますが、例外処理を行った後のリターン命令でユーザモードに復帰します。ユーザモードでは、レジスタのアクセスに次の制限があります。

PSR[12]	SVフラグ	変更不可、読み出しのみ可
PSR[11:8]	ILビット	変更不可、読み出しのみ可
PSR[4]	IEフラグ	変更不可、読み出しのみ可
TTBR[31:0]		変更不可、読み出しのみ可
SSP[31:0]		変更不可、読み出しのみ可

メモリアドレス空間の制限については、機種別テクニカルマニュアルのHBCUおよびBBCUの説明を参照してください。

6.1.4 例外処理

ソフトウェア例外など、例外が発生するとPSRのSVフラグ(ビット12)が0にリセットされ、スーパバイザモードとなって例外処理状態に移行します。例外処理発生要因を以下に示します。

(1) 外部割り込み
(2) ソフトウェア例外
(3) アドレス不整例外
(4) ゼロ除算
(5) NMI

6.1.5 MMU例外

MMUが有効な状態のとき、アクセスしようとする論理アドレスがMMUのアドレス変換テーブルに定義されていない場合にMMU例外が発生します。MMU例外が発生した場合には、ソフトウェアでMMUのアドレス変換テーブルを適切な値に書き換える操作が必要です。
詳細については、機種別テクニカルマニュアルのMMUの説明を参照してください。
また、SVフラグ(PSRのビット12)の設定にかかわらず、MMU例外処理ルーチン実行時はスーパバイザモードに固定されます。

6.1.6 デバッグ例外

C33 ADVコアCPUは、アプリケーションの開発効率を向上させるため、デバッグを支援する機能を搭載しています。この機能を使用するためのモードがデバッグモードで、`brk`命令やデバッグ例外によってユーザモードから切り換わるようになっています。通常は、このモードになることはありません。

6.1.7 HALTモード

ソフトウェアで`halt`命令を実行するとCPUはHALTモードに移行します。HALTモードではCPUへのクロック供給が停止し、低消費電力状態になります。HALTモードは、イニシャルリセット、NMI、外部割り込みの発生によって解除できます。

6.1.8 SLEEPモード

ソフトウェアで`slp`命令を実行するとCPUはSLEEPモードに移行します。SLEEPモードではCPUに加えて周辺回路へのクロック供給が停止し、HALTモードよりもさらに低消費電力状態になります。SLEEPモードは、イニシャルリセット、NMI、外部割り込みの発生によって解除できます。

6.2 プログラムの実行

CPUは、イニシャルリセットが解除されるとリセットベクタのアドレスをPCにロードして、そのアドレスから命令の実行を開始します。C33 ADVコアCPUの命令は16ビットの固定長となっていますので、以後CPUはPCの示すアドレスから命令をフェッチするごとにPCに+2ずつ加算し、命令を順次実行します。
分岐命令が実行されるとCPUはPSRのフラグを検査し、分岐条件が成立していれば分岐先のアドレスをPCにロードします。
割り込みや例外が発生すると、CPUはトラップベクタテーブルから割り込み処理ルーチンのアドレスをPCにロードします。
トラップベクタテーブルはリセットベクタを先頭とするトラップベクタのテーブルで、イニシャルリセット時はその先頭アドレスが0x20000000に設定されます。トラップベクタテーブルの先頭アドレスは、CPUのTTBRレジスタで参照することができます。また、ソフトウェアで任意のアドレスを設定することが可能です。この場合、TTBRには1Kバイト境界アドレス(TTBR[9:0] = “00 0000 0000”固定)を設定する必要があります。

6.2.1 命令フェッチと実行

C33 ADVコアCPUは内部で5段のパイプライン処理を行って、データ転送命令や一般の演算命令を1クロックで実行します。
パイプライン処理は、フェッチと実行を同時に行うことで処理時間の高速化を図るもので、5段のパイプラインでは各命令を5つのステージに分けて平行処理します。

基本命令ステージ

命令フェッチ(F)	命令デコード(D)	命令実行(E)	メモリアクセス(A)	レジスタライト(W)

以降、各ステージは次の記号で表します。

命令フェッチ → F (Fetch)
命令デコード → D(Decode)
命令実行 → E(Execute)
メモリアクセス → A(Access)
レジスタライト → W(Write)

パイプライン動作

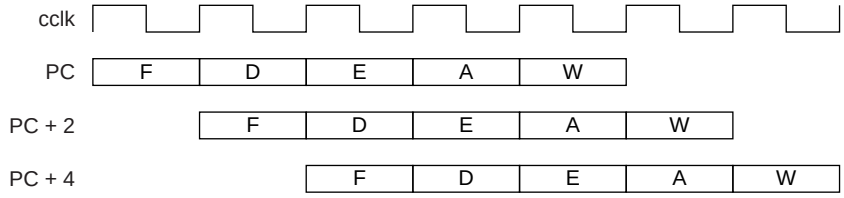


図6.2.1.1 パイプライン動作

注: 上記のパイプライン動作は内部メモリを使用したときのものです。外部メモリや低速外部デバイスの使用時はデバイスによって任意のウェイトサイクルが挿入され、FステージまたはAステージでウェイトします。

6.2.2 実行サイクルとフラグ

C33 ADVコアCPUは、命令(5ステージ/命令)を5段のパイプラインに通して平行処理することにより、1クロックの基本実行サイクルを実現しています。
ただし、レジスタ直接アドレッシングなどレジスタ間でデータの転送を行うような命令は、1クロックで実行されますが、外部メモリや低速の外部周辺回路へのアクセスは、HBCUとBBCUを通過する際のバス調停に費やされるクロックサイクルと、接続されているデバイスの固有ウェイトサイクルが付加されます。
なお、内蔵RAMおよびキャッシュメモリのアクセスは1クロックで完了します。
以下に、内蔵RAMまたはキャッシュメモリにアクセスした場合のクロック数とフラグの変化の一覧を示します。

S1C33 STDコアCPU互換命令

表6.2.2.1 命令実行サイクルクロック数とフラグ変化(S1C33 STDコアCPU互換命令)

分 類	ニーモニック		サイクル	インターロック	フラグ				備 考
					C	V	Z	N	
算術演算	add	*%rd,%rs*	1		↔	↔	↔	↔	
		%rd,imm6	1		↔	↔	↔	↔	
		%sp,*imm10*	1		–	–	–	–	
	adc	*%rd,%rs*	1		↔	↔	↔	↔	
	sub	*%rd,%rs*	1		↔	↔	↔	↔	
		%rd,imm6	1		↔	↔	↔	↔	
		%sp,*imm10*	1		–	–	–	–	
	sbc	*%rd,%rs*	1		↔	↔	↔	↔	
	cmp	*%rd,%rs*	1		↔	↔	↔	↔	
		%rd,sign6	1		↔	↔	↔	↔	
	mlt.h	*%rd,%rs*	1	2 (∗8)	–	–	–	–	
	mltu.h	*%rd,%rs*	1	2 (∗8)	–	–	–	–	
	mlt.w	*%rd,%rs*	2	2 (∗8)	–	–	–	–	
	mltu.w	*%rd,%rs*	2	2 (∗8)	–	–	–	–	
	div0s	*%rs*	1	2 (∗8)	–	–	–	↔	DS**変化**
	div0u	*%rs*	1	2 (∗8)	–	–	–	0	DS = 0
	div1	*%rs*	1	2 (∗8)	–	–	–	–	
	div2s	*%rs*	1	2 (∗8)	–	–	–	–	
	div3s		1	2 (∗8)	–	–	–	–	
分岐	jrgt jrgt.d	*sign8*	1–2 (∗1, ∗7)		–	–	–	–	
	jrge jrge.d	*sign8*	1–2 (∗1, ∗7)		–	–	–	–	
	jrlt jrlt.d	*sign8*	1–2 (∗1, ∗7)		–	–	–	–	
	jrle jrle.d	*sign8*	1–2 (∗1, ∗7)		–	–	–	–	
	jrugt jrugt.d	*sign8*	1–2 (∗1, ∗7)		–	–	–	–	
	jruge jruge.d	*sign8*	1–2 (∗1, ∗7)		–	–	–	–	
	jrult jrult.d	*sign8*	1–2 (∗1, ∗7)		–	–	–	–	
	jrule jrule.d	*sign8*	1–2 (∗1, ∗7)		–	–	–	–	
	jreq jreq.d	*sign8*	1–2 (∗1, ∗7)		–	–	–	–	
	jrne jrne.d	*sign8*	1–2 (∗1, ∗7)		–	–	–	–	
	jp	*sign8*	1–2 (∗7)		–	–	–	–	
	jp.d	*%rb*	2–3 (∗7)		–	–	–	–	
	call	*sign8*	1–2 (∗7)		–	–	–	–	
	call.d	*%rb*	2–3 (∗7)		–	–	–	–	

分　類	ニーモニック		サイクル	インターロック	フラグ				備　考
					C	V	Z	N	
分岐	ret		4–5 (∗7)		–	–	–	–	
	ret.d								
	reti		6		↔	↔	↔	↔	PSR**変化**
	retd		6		–	–	–	–	DE = 0
	int	*imm2*	7		–	–	–	–	SV = 0, IE = 0
	brk		7		–	–	–	–	DE = 1, IE**変化なし**
データ転送	ld.b	*%rd,%rs*	1		–	–	–	–	
		%rd,[*%rb*]	1	1–2 (∗9)	–	–	–	–	
		%rd,[*%rb*]+	1	1–2 (∗9)	–	–	–	–	
		%rd,[%sp+*imm6*]	1	1–2 (∗9)	–	–	–	–	
		[*%rb*]*,%rs*	1		–	–	–	–	
		[*%rb*]+*,%rs*	1		–	–	–	–	
		[%sp+*imm6*]*,%rs*	1		–	–	–	–	
	ld.ub	*%rd,%rs*	1		–	–	–	–	
		%rd,[*%rb*]	1	1–2 (∗9)	–	–	–	–	
		%rd,[*%rb*]+	1	1–2 (∗9)	–	–	–	–	
		%rd,[%sp+*imm6*]	1	1–2 (∗9)	–	–	–	–	
	ld.h	*%rd,%rs*	1		–	–	–	–	
		%rd,[*%rb*]	1	1–2 (∗9)	–	–	–	–	
		%rd,[*%rb*]+	1	1–2 (∗9)	–	–	–	–	
		%rd,[%sp+*imm6*]	1	1–2 (∗9)	–	–	–	–	
		[*%rb*]*,%rs*	1		–	–	–	–	
		[*%rb*]+*,%rs*	1		–	–	–	–	
		[%sp+*imm6*]*,%rs*	1		–	–	–	–	
	ld.uh	*%rd,%rs*	1		–	–	–	–	
		%rd,[*%rb*]	1	1–2 (∗9)	–	–	–	–	
		%rd,[*%rb*]+	1	1–2 (∗9)	–	–	–	–	
		%rd,[%sp+*imm6*]	1	1–2 (∗9)	–	–	–	–	
	ld.w	*%rd,%rs*	1		–	–	–	–	
		%rd,sign6	1		–	–	–	–	
		%rd,[*%rb*]	1	1–2 (∗9)	–	–	–	–	
		%rd,[*%rb*]+	1	1–2 (∗9)	–	–	–	–	
		%rd,[%sp+*imm6*]	1	1–2 (∗9)	–	–	–	–	
		[*%rb*]*,%rs*	1		–	–	–	–	
		[*%rb*]+*,%rs*	1		–	–	–	–	
		[%sp+*imm6*]*,%rs*	1		–	–	–	–	
システム制御	nop		1		–	–	–	–	
	halt		1		–	–	–	–	
	slp		1		–	–	–	–	
即値拡張	ext	*imm13*	0–1 (∗2)		–	–	–	–	
ビット処理	btst	[*%rb*]*,imm3*	3		–	–	↔	–	
	bclr	[*%rb*]*,imm3*	3		–	–	–	–	
	bset	[*%rb*]*,imm3*	3		–	–	–	–	
	bnot	[*%rb*]*,imm3*	3		–	–	–	–	
その他	swap	*%rd,%rs*	1		–	–	–	–	
	mirror	*%rd,%rs*	1		–	–	–	–	
	mac	*%rs*	2 + N × 2	2 (∗8)	–	–	–	–	MO**変化**
	pushn	*%rs*	N		–	–	–	–	PM, RC**変化**
	popn	*%rd*	N	1–2 (∗9)	–	–	–	–	PM, RC**変化**

機能拡張された命令

表6.2.2.2 命令実行サイクルクロック数とフラグ変化(機能拡張された命令)

分　類	ニーモニック		サイクル	インターロック	フラグ				備　考
					C	V	Z	N	
論理演算	and	*%rd,%rs*	1		–	∗3	↔	↔	
		%rd,sign6	1		–	∗3	↔	↔	
	or	*%rd,%rs*	1		–	∗3	↔	↔	
		%rd,sign6	1		–	∗3	↔	↔	
	xor	*%rd,%rs*	1		–	∗3	↔	↔	
		%rd,sign6	1		–	∗3	↔	↔	
	not	*%rd,%rs*	1		–	∗3	↔	↔	
		%rd,sign6	1		–	∗3	↔	↔	
シフト&ローテート	srl	*%rd,%rs*	1		∗4	∗4	↔	↔	
		%rd,imm5	1		∗4	∗4	↔	↔	
	sll	*%rd,%rs*	1		∗4	∗4	↔	↔	
		%rd,imm5	1		∗4	∗4	↔	↔	
	sra	*%rd,%rs*	1		∗4	∗4	↔	↔	
		%rd,imm5	1		∗4	∗4	↔	↔	
	sla	*%rd,%rs*	1		∗4	∗4	↔	↔	
		%rd,imm5	1		∗4	∗4	↔	↔	
	rr	*%rd,%rs*	1		∗4	∗4	↔	↔	
		%rd,imm5	1		∗4	∗4	↔	↔	
	rl	*%rd,%rs*	1		∗4	∗4	↔	↔	
		%rd,imm5	1		∗4	∗4	↔	↔	
データ転送	ld.w	*%rd,%ss*	1		–	–	–	–	
		%sd,%rs	1 (∗10)		–	–	–	–	
その他	scan0	*%rd,%rs*	1		↔	0	↔	0	
	scan1	*%rd,%rs*	1		↔	0	↔	0	

追加命令

表6.2.2.3 命令実行サイクルクロック数とフラグ変化(追加命令)

分　類	ニーモニック		サイクル	インターロック	フラグ				備　考
					C	V	Z	N	
算術演算	add	*%rd,*%dp	1		–	–	–	–	
	mlt.hw	*%rd,%rs*	2	2 (∗8)	–	–	–	–	
	mac.hw	*%rs*	2 + N × 2	2 (∗8)	–	–	–	–	MO変化
	mac.w	*%rs*	3 + N × 2	2 (∗8)	–	–	–	–	MO変化
	mac1.h	*%rd,%rs*	1	2 (∗8)	–	–	–	–	MO変化
	mac1.hw	*%rd,%rs*	2	2 (∗8)	–	–	–	–	MO変化
	mac1.w	*%rd,%rs*	2	2 (∗8)	–	–	–	–	MO変化
	div.w	*%rs*	35	2 (∗8)	–	–	–	↔	DS変化
	divu.w	*%rs*	35	2 (∗8)	–	–	–	0	DS = 0
分岐	jpr jpr.d	*%rb*	3–4 (∗7)		–	–	–	–	
	retm		6		–	–	–	–	ME = 0
データ転送	ld.b	*%rd,*[%dp+*imm6*]	1	1–2 (∗9)	–	–	–	–	
	ld.ub	*%rd,*[%dp+*imm6*]	1	1–2 (∗9)	–	–	–	–	
	ld.h	*%rd,*[%dp+*imm6*]	1	1–2 (∗9)	–	–	–	–	
	ld.uh	*%rd,*[%dp+*imm6*]	1	1–2 (∗9)	–	–	–	–	
	ld.w	*%rd,*[%dp+*imm6*]	1	1–2 (∗9)	–	–	–	–	
	ld.b	[%dp+*imm6*],*%rs*	1		–	–	–	–	
	ld.h	[%dp+*imm6*],*%rs*	1		–	–	–	–	
	ld.w	[%dp+*imm6*],*%rs*	1		–	–	–	–	
システム制御	psrset	*imm5*	4		↔	↔	↔	↔	
	psrclr	*imm5*	4		↔	↔	↔	↔	
多機能拡張	ext	*%rs*	0–1 (∗2)		–	–	–	–	
	ext	*cond*	0–1 (∗2)		–	–	–	–	
	ext	*op,imm2*	0–1 (∗2)		–	–	–	–	
	ext	*%rs,op,imm2*	0–1 (∗2)		–	–	–	–	
コプロセッサ制御	ld.c	*%rd,imm4*	1		–	–	–	–	
	ld.c	*imm4,%rs*	1		–	–	–	–	
	do.c	*imm6*	1		–	–	–	–	
	ld.cf		1		↔	↔	↔	↔	

分 類	ニーモニック		サイクル	インターロック	フラグ C	フラグ V	フラグ Z	フラグ N	備 考
その他	macclr		1		–	–	–	–	MO = 0
	swaph	*%rd,%rs*	1		–	–	–	–	
	push	*%rs*	1		–	–	–	–	
	pop	*%rd*	1	1–2 (∗9)	–	–	–	–	
	pushs	*%ss*	N		–	–	–	–	PM, RC**変化**
	pops	*%sd*	N	1–2 (∗9)	–	–	–	–	PM, RC**変化**
	sat.b	*%rd,%rs*	1		–	–	–	–	S**変化**
	sat.ub	*%rd,%rs*	1		–	–	–	–	S**変化**
	sat.h	*%rd,%rs*	1		–	–	–	–	S**変化**
	sat.uh	*%rd,%rs*	1		–	–	–	–	S**変化**
	sat.w	*%rd,%rs*	1		–	–	–	–	S**変化**
	sat.uw	*%rd,%rs*	1		–	–	–	–	S**変化**
	loop	*%rc,%ra*	5 (∗5)		–	–	–	–	LM**変化**
		%rc,imm4	5 (∗5)		–	–	–	–	LM**変化**
		imm4,imm4	5 (∗5)		–	–	–	–	LM**変化**
	repeat	*%rc*	4 (∗6)		–	–	–	–	RM**変化**
		imm4	4 (∗6)		–	–	–	–	RM**変化**

∗1 分岐条件成立時かつディレイド分岐命令でない場合は2サイクル
∗2 先読みデコード可能な場合は0サイクル
∗3 PSRのOCフラグ(ビット21)が1のときに変化
∗4 PSRのSEフラグ(ビット20)が1のときに変化
∗5 loop命令を実行したときのみ5サイクル、ループ動作は各命令のサイクル
∗6 repeat命令を実行したときのみ4サイクル、リピート動作は各命令のサイクル
∗7 分岐命令でディレイド分岐を行わない場合(命令語の後に".d"がないとき)、分岐中に命令の実行を行わないため1命令分の空白時間ができ、見かけ上+1サイクルになります。
∗8 次の命令でAHR、ALRを参照するときにかかるインターロックサイクル。ただし、次の命令にmac、mac1、mlt、div命令を組み合わせる場合には0サイクル。LCフラグ = 1によりR4、HCフラグ = 1によりR5を参照する場合もインターロックサイクルは2サイクル
∗9 次の命令が、*rd*レジスタを間接アドレスレジスタとして使用する場合のみ2サイクル
インターロックサイクルが2サイクルとなる場合の例

```
ld.w  r1,[r2]          ; r1 ← [r2]
ld.w  r3,[r1]          ; r1を間接アドレスレジスタとして使用
```

∗10 ld.w %psr,*%rs*命令を実行したときのみ4サイクル

備考欄にあるPSRのフラグは、C、V、Z、Nフラグ以外に影響を受けるフラグを示しています。

表中のインターロック欄は、その命令を実行したときに*rd*レジスタが確定するまでのインターロックサイクルです。つまり、直後の命令が*rd*レジスタを参照する場合、*rd*レジスタの値が確定するまで*rd*レジスタにアクセスできないために生じるペナルティクロックサイクルです。したがって、次の命令の実行サイクルは、インターロック欄のサイクル数を加えた値になります。

6.3 割り込みと例外

CPUはプログラム実行中に、外部割り込みや例外が発生すると例外処理状態となります。例外処理状態は、各割り込み/例外要因に対応したユーザの処理ルーチンに分岐するまでのプロセスで、分岐後は再びプログラム実行状態に戻ります。

6.3.1 例外の優先順位

C33 ADVコアCPUがサポートする例外は以下のとおりです。

(1) リセット、CPU内部例外、外部割り込みなどベクタテーブルを参照して例外処理ルーチンへ分岐する例外
(2) MMUを使用する場合に発生するMMUの論理アドレス変換テーブルを制御するためのMMU例外
(3) ユーザデバッグをサポートする、ブレーク処理などのデバッグ例外

これらの例外の優先順位を下表に示します。

表6.3.1.1 例外のベクタアドレスと優先順位

例　外	ベクタアドレス(Hex)	優先順位
リセット	TTBR + 0x00	**高い**
ゼロ除算	TTBR + 0x10	▲
アドレス不整例外	TTBR + 0x18	▲
デバッグ例外	0x00000000 **または** 0x00060000	▲
MMU**例外**	0x00000010	▲
NMI	TTBR + 0x1C	▲
ソフトウェア例外	TTBR + 0x30 〜 TTBR + 0x3C	▲
マスク可能な外部割り込み	TTBR + 0x40 〜 TTBR + 0x3FC	**低い**

同時に例外が発生した場合、より優先順位の高いものから例外処理を行います。
例外が発生すると、CPUはそれ以降の割り込みを禁止し、例外処理に移行します。多重割り込みに対応するには、例外処理ルーチンの中でPSRのIEフラグを1にして例外処理中の例外発生を許可します。基本的には多重割り込みを行う場合でも、IL[3:0]ビットによって同レベル以下の例外の発生は禁止されます。

デバッグ例外およびMMU例外は、ベクタアドレスを格納するベクタが特定のアドレスとなり、ベクタテーブルは参照されません。また、PCの退避にもスタックは使用されず、R0とともに特定の領域に格納されます。
デバッグ例外発生時に参照されるアドレスは、次のとおりです。

表6.3.1.2 デバッグ例外ベクタアドレスとPC/R0退避領域

アドレス	内　容
0x00000000 / 0x00060000	**デバッグ例外処理ベクタ**
0x00000008 / 0x00060008	PC**退避領域**
0x0000000C / 0x0006000C	R0**退避領域**

なお、デバッグ例外処理中は、他の例外および多重のデバッグ例外は受け付けられません。デバッグ例外処理終了後に受け付けられます。

MMU例外発生時に参照されるアドレスは、次のとおりです。

表6.3.1.3 MMU例外ベクタアドレスとPC/R0退避領域

アドレス	内　容
0x00000010	**MMU例外処理ベクタ**
0x00000018	PC**退避領域**
0x0000001C	R0**退避領域**

なお、MMU例外処理中は、他の例外は禁止状態となり受け付けられません。また、MMU例外処理は物理アドレスで行われるため、多重のMMU例外も発生しません。
詳しくは、機種別テクニカルマニュアルのMMUの説明を参照してください。

6.3.2 ベクタテーブル

C33 ADVコアCPUのベクタテーブル

ベクタテーブルを参照するC33 ADVコアCPUの例外を表6.3.2.1に示します。優先順位は割り込みコントローラ(ITC)によって管理されます。

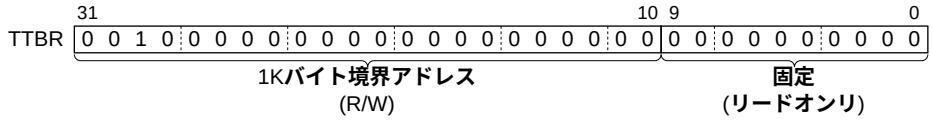
表6.3.2.1 ベクター覧

例　外	ベクタNo.	同期/非同期	分　類	ベクタアドレス	優先順位
リセット	0	**非同期**	**割り込み**	TTBR + 0x00	**高い**
reserved	1~3	–	–	–	↑
ゼロ除算	4	**同期**	**例外**	TTBR + 0x10	↑
reserved	5	–	–	–	↑
アドレス不整例外	6	**同期**	**例外**	TTBR + 0x18	↑
NMI	7	**非同期**	**割り込み**	TTBR + 0x1C	↑
reserved	8~11	–	–	–	↑
ソフトウェア例外0	12	**同期**	**例外**	TTBR + 0x30	↑
ソフトウェア例外1	13	**同期**	**例外**	TTBR + 0x34	↑
ソフトウェア例外2	14	**同期**	**例外**	TTBR + 0x38	↑
ソフトウェア例外3	15	**同期**	**例外**	TTBR + 0x3C	↑
マスク可能な外部割り込み0	16	**非同期**	**割り込み**	TTBR + 0x40	↑
:	:	:	:	:	↓
マスク可能な外部割り込み239	255	**非同期**	**割り込み**	TTBR + 0x3FC	**低い**

同期/非同期は、その例外がプログラムの実行に同期して発生するか、非同期に発生するかを示しています。同期して発生するものを「例外」、非同期に発生するものを「割り込み」として分類しています。割り込み/例外の発生によって行うCPUの処理について、本書では一括して例外処理と記述しています。

ベクタアドレスは、各例外が発生した場合に実行するユーザの例外処理ルーチンへのベクタ(分岐先アドレス)を格納しておくアドレスです。アドレス値を格納しておくため、それぞれワード境界に配置されます。このベクタを格納しておくメモリ領域をベクタテーブルと呼び、ベクタアドレス欄に示した“TTBR”はベクタテーブルのベース(先頭)アドレスを表します。
C33 ADVコアCPUはTTBRを特殊レジスタとして内蔵し、ソフトウェアによって書き込みが可能です。これにより、任意のRAM領域にベクタテーブルを配置することができます。

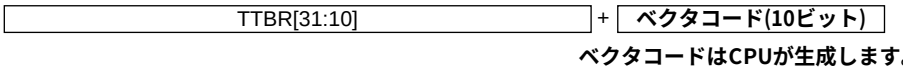
TTBR(Trap Table Base Register)



TTBRは、コールドリセット時に“0x20000000”に初期設定されます。

ベクタテーブル参照アドレス

例外が発生するとTTBRの値と例外要因ごとに設定された10ビットのベクタコードからベクタテーブルが参照されます。TTBRはビット31～10の値が参照されますのでベクタテーブルは1Kバイト境界のRAM領域に割り付けなければなりません。



ベクタコードはCPUが生成します。

6.3.3 例外処理

割り込み、または例外が発生するとCPUは例外処理を開始します。(ここで説明する例外処理は、リセット、MMU例外、デバッグ例外には適用されません。)
以下に、例外処理の動作を示します。

(1) 実行中の命令を中断します。
割り込み、例外は実行中の命令のAステージとWステージ間で、システムクロックの立ち上がりエッジに同期して発生します。

(2) PC、PSRの順にそれぞれの内容をスタック(SSP)に退避させます。

(3) PSRのIE(割り込みイネーブル)ビットをクリアし、それ以降のマスク可能な割り込みを禁止します。発生した例外がマスク可能な割り込みの場合は、PSRのIL(割り込みレベル)を発生した割り込みのレベルに変更します。また、PSRのSVフラグ(ビット12)を0にしてスーパバイザモードに切り換え、RMフラグ(ビット30)、LMフラグ(ビット29)、PMフラグ(ビット28)を0にクリアします。

(4) ベクタテーブルから、発生した例外のベクタを読み出しPCにセットします。これにより、ユーザの例外処理ルーチンに分岐します。

ユーザの例外処理ルーチンでは最後に`reti`命令を実行する必要があります。`reti`命令はPC、PSRの順にスタックからデータを復帰させ、ユーザモードに切り換えて中断していた命令に処理を戻します。

6.3.4 リセット

CPUの#RESET端子にLowパルスを入力することで、CPUがリセットされます。
CPUは#RESETパルスの立ち上がりエッジで動作を開始し、リセット処理を行います。リセット処理ではベクタテーブルの先頭からリセットベクタが読み出され、PCにセットされます。これにより、ユーザの初期化ルーチンに分岐してプログラムの実行を開始します。リセット処理は他のすべての処理に優先します。
C33 ADVコアCPUは、コールドスタートとホットスタートの2種類のリセット方式をサポートしています。

コールドスタート(#RESET端子 = Low, #NMI端子 = High)

#NMI端子をHighにした状態で#RESET端子をLowにしてリセットすると、C33 ADVコアCPUはコールドスタートします。コールドスタートの場合、CPUの他にチップ上の周辺回路もすべて初期化されますので、主にパワーオンリセットに用います。

図6.3.4.1 コールドリセットタイミング

ホットスタート(#RESET端子 = Low, #NMI端子 = Low)

#NMI端子をLowにした状態で#RESET端子をLowにしてリセットすると、C33 ADVコアCPUはホットスタートします。ホットスタートの場合、CPUは初期化されますが、周辺回路の中で外部バスコントロールユニットや入出力ポートなどは初期化されません。外部メモリや外部入出力の状態を保持したままリセットをかけたい場合に用います。

図6.3.4.2 ホットリセットタイミング

リセットにより、PSRの全ビットが0にクリアされます。TTBRはコールドスタートで0x20000000に初期化され、ホットスタートではリセット前の値を保持します。他のレジスタは不定となります。

6.3.5 ゼロ除算例外

除算命令実行時に除数が0であると、このゼロ除算例外が発生します。
この例外が発生するのは、除算の前処理を行う`div0s`、`div0u`の2命令および一括除算命令の`div.w`、`divu.w`の2命令です。除数が0の場合、CPUはこの命令の実行を終了後、例外処理に移行します。
例外処理でスタックにセーブするPC値は、例外が発生した命令の次のアドレスとなります。

6.3.6 アドレス不整例外

メモリやI/O領域をアクセスするロード命令は、命令により転送するデータサイズが決まっています。そのアドレスはデータサイズごとの境界でなければなりません。

命令	転送データサイズ	アドレス
`ld.b/ld.ub`	バイト(8ビット)	バイト境界(全アドレスが対象)
`ld.h/ld.uh`	ハーフワード(16ビット)	ハーフワード境界(アドレスの最下位ビットが0)
`ld.w`	ワード(32ビット)	ワード境界(アドレスの下位2ビットが00)

ロード命令の指定アドレスがこの条件を満たしていない場合、CPUはアドレス不整例外として例外処理に移行します。この場合、ロード命令は実行されません。例外処理でスタックにセーブするPC値は、例外を発生したロード命令の次のアドレスとなります。

積和演算(`mac`)命令は、メモリ上のハーフワードデータまたはワードデータを扱うため、アドレス不整例外が発生する可能性があります。この場合も、スタックにセーブされるリターンアドレスは`mac`命令のアドレスとなり、例外処理ルーチンからのリターン後は残りの回数の積和演算を継続します。

ベースアドレスとしてSP、DPを使用するロード命令においては、アドレスがデータサイズに応じてアライメントされるため、アドレス不整例外は発生しません。

プログラムの分岐を伴う命令(`call %rb`、`jp %rb`)では、PCの最下位ビットが常に0に固定されるため、この例外は発生しません。例外処理のベクタについても同様です。

6.3.7 NMI

CPUのNMI入力がアクティブ(Low)になるとNMIが発生します。NMIが発生すると、CPUは実行中の命令を終了後、例外処理に移行します。例外処理でスタックにセーブするPC値は、実行中の命令のアドレスとなります。
NMI処理中は、新たなNMIを無効として受け付けません(多重NMIの禁止)。NMIが多重に実行されるのを防ぐため、NMI処理ルーチンの実行開始時からNMIはマスクされます。NMIのマスクは`reti`命令の実行によって解除されますが、NMI処理ルーチン内でその他の例外が発生し、その例外処理ルーチンの中で`reti`が実行された場合でもNMIのマスクは解除されます。このような場合、NMI処理ルーチンが正しく実行されない場合があるので、NMI処理ルーチン実行中は他の例外が発生しないようにしてください。
NMIはマスクが不可能な割り込みですが、CPUリセット後(コールドスタート、ホットスタートとも)のSPが不定な期間に発生するとプログラムが暴走するため、`ld.w %sp,%rs`命令によってSPを設定するまではハードウェアによってCPUのNMI入力がマスクされるようになっています。

6.3.8 ソフトウェア例外

ソフトウェア例外は、`int imm2`命令の実行によって発生します。この例外処理でスタックにセーブするPC値は、`int`命令の次のアドレスとなります。`int`命令のオペランド*imm2*は、4種類のソフトウェア例外のベクタアドレスを指定します。CPUはTTBR + 48(ソフトウェア例外0ベクタアドレス)に4 × *imm2*を加算したアドレスからベクタを読み出して処理ルーチンに分岐します。

6.3.9 マスク可能な外部割り込み

C33 ADVコアCPUは128種類までのマスク可能な外部割り込みを受け付けることができます。
マスク可能な割り込みは、PSRのIE(割り込みイネーブル)フラグがセットされている場合にのみCPUが受け付けます。また、PSRのIL(割り込みレベル)フィールドにより受け付け可能な割り込みのレベルが制限されます。ILフィールドの割り込みレベル(0～15)はCPUが受け付け可能な割り込みレベルを示し、その値より大きいレベルの割り込みのみを受け付けます。
IEフラグとILフィールドはソフトウェアで設定可能です。また、例外発生時にはPSRをスタックにセーブ後、IEフラグは0(割り込み禁止)にクリアされ、処理ルーチン内でIEフラグをセットするか、PSRを復帰させる`reti`命令で処理ルーチンを終了させるまで、マスク可能な割り込みを禁止します。ILフィールドも発生した割り込みのレベルに設定されます。
割り込み処理ルーチン内でIEフラグをセットすることによって、現在処理中の割り込みよりも高いレベルの割り込みを受け付ける多重割り込みが容易に実現できます。
CPUがリセットされた場合はPSRが0に初期化されるため、マスク可能な割り込みは禁止され、割り込みレベルは0(1～15の割り込みレベルを許可)に設定されます。

マスク可能な割り込みの発生手順とCPUの例外処理の内容は次のとおりです。

(1) 実行中の命令を中断します。
割り込みは、実行中の命令のAステージとWステージ間で、システムクロックの立ち上がりエッジに同期して受け付けられます。

(2) CPUはPC、PSRの順に各レジスタの内容をスタック(SSP)に退避させます。

(3) PSRのIEフラグをクリアし、発生した割り込みの割り込みレベルをILフィールドにコピーします。さらに、PSRのSVフラグをクリアしてスーパバイザモードに切り換えます。

(4) CPUは割り込みに対応したベクタテーブル内のベクタアドレスからベクタを読み出してPCにセットし、割り込み処理ルーチンに分岐します。

割り込み処理ルーチンでは、処理の最後に`reti`命令を実行する必要があります。`reti`命令は、PC、PSRの順にスタックからデータを復帰させ、ユーザモードに切り換えて中断していた命令に処理を戻します。

6.3.10 MMU例外

C33 ADVコアCPUは、MMUを使用して論理アドレス空間を物理アドレス空間に変換し、メモリ管理を行うことができます。
物理アドレスへの変換は、MMUのレジスタにセットされた情報によって行われ、プログラムがレジスタによって区切られた論理空間を逸脱(ミスヒット)してアクセスを行った場合に、MMU例外が発生します。
ソフトウェアは、MMU例外が発生するとミスヒットしたMMUのレジスタを更新し、新たに論理アドレスを物理アドレスにマッピングします。
MMU例外中はSVフラグ(PSRのビット12)にかかわらず、スーパバイザモードに固定されます。
MMU例外中はハードウェア割り込みおよびNMIを受け付けません。
MMU例外中はループ/リピート動作を行わないでください。

MMU例外の処理手順は次のとおりです。

(1) 実行中の命令を中断します。
MMU例外は、実行中の命令のAステージ最後で発生します。次のシステムクロックの立ち上がりエッジでMMU例外を受け付けます。

(2) CPUは、PC、R0の順にそれぞれの内容を以下のアドレスに格納します。
PC → 0x00000014, R0 → 0x00000018

(3) PSRのMEフラグ(ビット13)を1にセットします。

(4) CPUは、MMU例外ベクタを0x00000010番地からPCにロードし、MMU例外処理ルーチンに分岐します。

例外処理ルーチンでは、MMUのレジスタを適切な値に更新した後、最後に`retm`命令を実行して中断している命令に復帰します。`retm`命令で復帰する際、CPUはR0、PCの順でデータを復帰します。
詳しくは、機種別テクニカルマニュアルのMMUの説明を参照してください。

6.4 パワーダウンモード

C33 ADVコアCPUはHALTモードとSLEEPモードの2種類のパワーダウンモードをサポートしています。
パワーダウンモードは、CPUまたはそれに加えて周辺モジュールの機能を停止させ、消費電力を低減させます。

6.4.1 HALTモード

スーパバイザモード、またはHEフラグ(PSRのビット31)が1のときにCPUが`halt`命令を実行すると、その時点でプログラムの実行を中断しHALTモードに移行します。
HALT モードではCPUの動作が停止します。周辺回路に対してはクロックが供給されますので、周辺回路は動作を継続します。
HALTモードはイニシャルリセットまたはNMIを含む割り込みによって解除され、解除後はそれぞれの例外処理を経てプログラム実行状態に移行します。割り込みによって解除した場合、例外処理によって`halt`の次の命令のアドレスがスタックにセーブされていますので、発生した割り込みの処理ルーチンを`reti`命令で終了すると、`halt`の次の命令の位置にリターンします。

6.4.2 SLEEPモード

スーパバイザモード、またはHEフラグ(PSRのビット31)が1のときにCPUが`slp`命令を実行すると、その時点でプログラムの実行を中断しSLEEPモードに移行します。
SLEEPモードではCPUおよび周辺回路も動作を停止します。このため、HALTモードよりも大幅に消費電流を低減することができます。
SLEEPモードはイニシャルリセットまたはNMIを含む割り込みによって解除され、解除後はそれぞれの例外処理を経てプログラム実行状態に移行します。割り込みによって解除した場合、例外処理によって`slp`の次の命令のアドレスがスタックにセーブされていますので、発生した割り込みの処理ルーチンを`reti`命令で終了すると、`slp`の次の命令の位置にリターンします。なお、SLEEPモードではチップ上の発振回路およびその出力クロックを使用する周辺回路が基本的に停止するため、SLEEPモードの解除は非同期の外部割り込み等により行われます。また、SLEEP解除により発振回路が動作を開始するため、CPUの動作開始には発振安定待ち時間が必要となります。

6.5 デバッグモード

C33 ADVコアCPUには、プログラム開発を支援するデバッグモードが設けられています。
デバッグモードがサポートしている機能は以下のとおりです。

- **命令ブレーク**
  設定した命令のアドレスを実行する前にデバッグ例外を発生します。3カ所のアドレスに命令ブレークを設定できます。
- **データブレーク**
  設定したアドレスに書き込み/読み出しが行われるとデバッグ例外を発生します。
  データブレークの設定が可能なアドレスは1カ所です。
- **シングルステップ**
  各命令ごとにデバッグ例外を発生します。
- **強制ブレーク**
  外部入力信号でデバッグ例外を発生します。
- **エリアブレーク**
  C33 ADVコアCPUで分割されるアドレス空間の指定エリア(独立した#CE信号が出力される)に対しアクセスが行われるとデバッグ例外を発生します。
- **バスブレーク**
  選択したバスのデータが設定値と一致するとデバッグ例外を発生します。
- **バストレース**
  選択したバスの値をトレースします。
- **PCトレース**
  CPUの命令実行状態をトレースします。

デバッグ例外が発生すると、CPUは次の処理を行います。

(1) 実行中の命令を中断します。
  デバッグ例外は、実行中の命令のAステージの最後に発生します。次のシステムクロックの立ち上がりエッジでデバッグ例外を受け付けます。

(2) CPUは、PC、R0の順にそれぞれの内容を以下のアドレスに格納します。
  PC → 0x00060008(または0x00000008)
  R0 → 0x0006000C(または0x0000000C)

(3) CPUは、デバッグ例外ベクタを0x00060000番地(または0x00000000番地)からPCにロードし、デバッグ例外処理ルーチンに分岐します。

例外処理ルーチンでは、処理の最後に`retd`命令を実行して中断している命令に復帰します。`retd`命令で復帰する際、CPUはR0、PCの順でデータを復帰します。
デバッグ例外中はハードウェア割り込みおよびNMIは受け付けられません。
デバッグ例外中はループ/リピート動作を行わないでください。

6.6 コプロセッサインタフェース

C33 ADVコアCPUは、コプロセッサインタフェースを搭載しています。コプロセッサとしてFPU、DSPなど様々なデータプロセッサを接続できるようにコプロセッサ専用の命令を用意し、シンプルなインタフェース(命令用の16ビットバスと入出力それぞれ32ビットのデータバス)で構成されています。

コプロセッサ専用命令

ld.c	***%rd,imm4***	コプロセッサからのデータ転送
ld.c	***imm4,%rs***	コプロセッサへのデータ転送
do.c	***imm6***	コプロセッサ実行
ld.cf		コプロセッサからC、V、Z、Nの各フラグを転送

コプロセッサの具体的なコマンドやステータスは、接続するコプロセッサによって異なりますので、コプロセッサの説明書を参照してください。

7 命令の詳細説明

C33 ADVコアCPUの命令セットは、S1C33 STDコアCPUの上位互換となっています。
次ページより全命令をアルファベット順に解説します。

命令説明中の記号

%rd, rd デスティネーションとして使用する汎用レジスタ(R0～R15)またはその内容です。
%rs, rs ソースとして使用する汎用レジスタ(R0～R15)またはその内容です。
%rb, rb レジスタ間接アドレッシングでアクセスされるベースアドレスを保持している汎用レジスタ(R0～R15)またはその内容です。
%sd, sd デスティネーションとして使用する特殊レジスタまたはその内容です。
%ss, ss ソースとして使用する特殊レジスタまたはその内容です。
%dp, dp データポインタ(DP)またはその内容です。
%sp, sp スタックポインタ(SP)またはその内容です。

コード中のレジスタフィールド(*rd, rs, sd, ss*)には、レジスタ番号が入ります。

汎用レジスタ(*rd, rs*)	R0 = 0b0000, R1 = 0b0001 . . . R15 = 0b1111
特殊レジスタ(*sd, ss*)	PSR = 0b0000, SP = 0b0001, ALR = 0b0010, AHR = 0b0011, LCO = 0b0100, LSA = 0b0101, LEA = 0b0110, SOR = 0b0111, TTBR = 0b1000, DP = 0b1001, IDIR = 0b1010, DBBR = 0b1011, USP = 0b1101, SSP = 0b1110, PC = 0b1111

immX *X*ビット長の符号なし即値です。*X*には即値のビット長を示す数値が入ります。
signX *X*ビット長の符号付き即値です。*X*には即値のビット長を示す数値が入ります。また、最上位ビットは符号ビットとして扱われます。
RM リピートモードイネーブルフラグ
LM ループモードイネーブルフラグ
PM プッシュ/ポップモードフラグ
RC[3:0] レジスタカウンタフィールド
S 飽和フラグ
DE デバッグ例外フラグ
ME MMU例外フラグ
IL[3:0] 割り込みレベルフィールド
MO MACオーバーフローフラグ
DS 被除数符号フラグ
IE 割り込みイネーブル
C キャリーフラグ
V オーバーフローフラグ
Z ゼロフラグ
N ネガティブフラグ
– 命令の実行によって変化しないことを示します。
↔ 命令の実行によってセット(= 1)またはリセット(= 0)されることを示します。
0 命令の実行によってリセット(= 0)されることを示します。

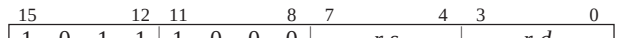
adc *%rd, %rs*

機　能

キャリー付き加算

標準） *rd* ← *rd* + *rs* + C
拡張1) 不可
拡張2) 不可
拡張3) *rd* ← *rs1* + *rs2* + C（"*op, imm2*"使用可）

コード



0xB8__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	↔	↔	↔	↔

モード

Src　レジスタ直接 *%rs* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK

1サイクル

説　明

(1)標準

```
adc  %rd,%rs        ; rd ← rd + rs + C
```

*rs*レジスタの内容とC(キャリー)フラグの内容を*rd*レジスタに加えます。

(2)拡張3

```
ext  %rs2,op,imm2 ; op = sra, srl, sla, imm2 = 0-3
adc  %rd,%rs1     ; rd ← (rs1 + rs2 + C) op imm2
```

*rs1*レジスタの内容にext命令で指定されたレジスタ*rs2*とC(キャリー)フラグを加え、さらに*op*で指示されるシフトを*imm2*のビット数分行い、結果を*rd*レジスタにロードします。*rs1*レジスタ、*rs2*レジスタの内容は変更されません。

(3)ディレイド命令

本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。

(4)ポストシフト

ポストシフト付き拡張命令の直後に記述することによって、本命令の実行結果が最大3ビットシフトされます。シフト動作は、sra, srl, sllの各命令と同じです。ただし、シフトアウトレジスタSORを使用しませんのでSORは変化しません。また、C、V、Z、Nフラグはadc命令の結果のみによって変化し、シフト動作の影響は受けません。

例

(1)
```
adc  %r0,%r1        ; r0 = r0 + r1 + C
```

(2)64ビットデータの加算

データ1 = {r2, r1}, データ2 = {r4, r3}, 加算結果 = {r2, r1}

```
add  %r1,%r3        ; 下位ワードの加算
adc  %r2,%r4        ; 上位ワードの加算
```

(3)
```
ext  %r2,srl,1
adc  %r3,%r1        ; r3 = (r1 + r2 + C) >> 1
```

add *%rd*, %dp

機　能

加算

標準） *rd* ← *rd* + dp
拡張1）*rd* ← dp + *imm13*
拡張2）*rd* ← dp + *imm26*
拡張3）*rd* ← dp + *rs*（“*op, imm2*”使用可）

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	1	0	1	0	1	*rd*			

0x035_

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

Src　レジスタ直接(DP)
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK

1サイクル

説　明

(1)標準

```
add  %rd,%dp       ; rd ← rd + dp
```

DPレジスタの内容を*rd*レジスタに加えます。

(2)拡張1

```
ext  imm13
add  %rd,%dp       ; rd ← dp + imm13
```

DPレジスタの内容に13ビット即値*imm13*をゼロ拡張して加え、結果を*rd*レジスタにロードします。DPレジスタの内容は変更されません。

(3)拡張2

```
ext  imm13         ; = imm26(25:13)
ext  imm13         ; = imm26(12:0)
add  %rd,%dp       ; rd ← dp + imm26
```

DPレジスタの内容に26ビット即値*imm26*をゼロ拡張して加え、結果を*rd*レジスタにロードします。DPレジスタの内容は変更されません。

(4)拡張3

```
ext  %rs,op,imm2   ; op = sra, srl, sla, imm2 = 0–3
add  %rd,%dp       ; rd ← (dp + rs) op imm2
```

DPレジスタの内容にext命令で指定されたレジスタ*rs*を加え、さらに*op*で指示されるシフトを*imm2*のビット数分行い、結果を*rd*レジスタにロードします。DPレジスタ、*rs*レジスタの内容は変更されません。

(5)ディレイド命令

本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。この場合はext命令による拡張は行えません。

(6)ポストシフト

ポストシフト付き拡張命令の直後に記述することによって、本命令の実行結果が最大3ビットシフトされます。シフト動作は、sra, srl, sllの各命令と同じです。ただし、シフトアウトレジスタSORを使用しませんのでSORは変化しません。また、C、V、Z、Nフラグはシフト動作では変化しません。

例

(1)
```
add  %r0,%dp       ; r0 = r0 + dp
ext  0x1
ext  0x1fff
add  %r1,%dp       ; r1 = dp + 0x3fff
```

(2)
```
ext  %r2,srl,1
add  %r3,%dp       ; r3 = (dp + r2) >> 1
```

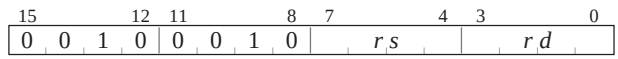
add *%rd, %rs*

機　能

加算

標準） *rd* ← *rd* + *rs*
拡張1) *rd* ← *rs* + *imm13*
拡張2) *rd* ← *rs* + *imm26*
拡張3) *rd* ← *rs1* + *rs2*（“*op, imm2*”使用可）

コード



0x22__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	↔	↔	↔	↔

モード

Src　レジスタ直接 *%rs* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK

1サイクル

説　明

(1)標準

```
add  %rd,%rs       ; rd ← rd + rs
```

*rs*レジスタの内容を*rd*レジスタに加えます。

(2)拡張1

```
ext  imm13
add  %rd,%rs       ; rd ← rs + imm13
```

*rs*レジスタの内容に13ビット即値*imm13*をゼロ拡張して加え、結果を*rd*レジスタにロードします。*rs*レジスタの内容は変更されません。

(3)拡張2

```
ext  imm13         ; = imm26(25:13)
ext  imm13         ; = imm26(12:0)
add  %rd,%rs       ; rd ← rs + imm26
```

*rs*レジスタの内容に26ビット即値*imm26*をゼロ拡張して加え、結果を*rd*レジスタにロードします。*rs*レジスタの内容は変更されません。

(4)拡張3

```
ext  %rs2,op,imm2 ; op = sra, srl, sla, imm2 = 0-3
add  %rd,%rs1     ; rd ← (rs1 + rs2) op imm2
```

*rs1*レジスタの内容にext命令で指定されたレジスタ*rs2*を加え、さらに*op*で指示されるシフトを*imm2*のビット数分行い、結果を*rd*レジスタにロードします。*rs1*レジスタ、*rs2*レジスタの内容は変更されません。

(5)ディレイド命令

本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。この場合はext命令による拡張は行えません。

(6)ポストシフト

ポストシフト付き拡張命令の直後に記述することによって、本命令の実行結果が最大3ビットシフトされます。シフト動作は、sra, srl, sllの各命令と同じです。ただし、シフトアウトレジスタSORを使用しませんのでSORは変化しません。また、C、V、Z、Nフラグはadd命令の結果のみによって変化し、シフト動作の影響は受けません。

例

```
(1)add  %r0,%r0      ; r0 = r0 + r0
(2)ext  0x1
   ext  0x1fff
   add  %r1,%r2      ; r1 = r2 + 0x3fff
(3)ext  %r2,srl,1
   add  %r3,%r1      ; r3 = (r1 + r2) >> 1
```

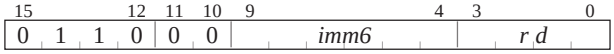
add *%rd, imm6*

機　能

加算

標準） *rd* ← *rd* + *imm6*
拡張1) *rd* ← *rd* + *imm19*
拡張2) *rd* ← *rd* + *imm32*
拡張3) 不可

コード



0x60__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
−	−	−	−	−	−	−	−	−	↔	↔	↔	↔

モード

Src　即値(符号なし)
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK

1サイクル

説　明

(1)標準

```
add  %rd,imm6      ; rd ← rd + imm6
```

6ビット即値*imm6*をゼロ拡張して*rd*レジスタに加えます。

(2)拡張1

```
ext  imm13         ; = imm19(18:6)
add  %rd,imm6      ; rd ← rd + imm19, imm6 = imm19(5:0)
```

19ビット即値*imm19*をゼロ拡張して*rd*レジスタに加えます。

(3)拡張2

```
ext  imm13         ; = imm32(31:19)
ext  imm13         ; = imm32(18:6)
add  %rd,imm6      ; rd ← rd + imm32, imm6 = imm32(5:0)
```

32ビット即値*imm32*を*rd*レジスタに加えます。

(4)ディレイド命令
本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。この場合はext命令による拡張は行えません。

(5)ポストシフト(“ext *op*,*imm2*”のみ)
ポストシフト付き拡張命令の直後に記述することによって、本命令の実行結果が最大3ビットシフトされます。シフト動作は、sra, srl, sllの各命令と同じです。ただし、シフトアウトレジスタSORを使用しませんのでSORは変化しません。また、C、V、Z、Nフラグはadd命令の結果のみによって変化し、シフト動作の影響は受けません。

例

(1)
```
add  %r0,0x3f      ; r0 = r0 + 0x3f
```

(2)
```
ext  0x1fff
ext  0x1fff
add  %r1,0x3f      ; r1 = r1 + 0xffffffff
```

add %sp, *imm10*

機　能　**加算**

標準）sp ← sp + *imm10* × 4
拡張1）不可
拡張2）不可
拡張3）不可

コード

0x80__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
−	−	−	−	−	−	−	−	−	−	−	−	−

モード　Src　即値（符号なし）
Dst　レジスタ直接（SP）

ＣＬＫ　1サイクル

説　明　(1)標準
10ビット即値*imm10*を4倍し、ゼロ拡張してスタックポインタSPに加えます。

(2)ディレイド命令
本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。

(3)ポストシフト（“`ext` *`op`*`,`*`imm2`*”のみ）
ポストシフト付き拡張命令の直後に記述することによって、本命令の実行結果が最大3ビットシフトされます。シフト動作は、`sra`, `srl`, `sll`の各命令と同じです。ただし、シフトアウトレジスタSORを使用しませんのでSORは変化しません。また、C、V、Z、Nフラグはシフト動作では変化しません。

例

```
add  %sp,0x100       ; sp = sp + 0x400
```

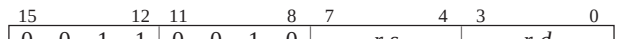
and *%rd, %rs*

機　能

論理積

標準） *rd* ← *rd* & *rs*
拡張1) *rd* ← *rs* & *imm13*
拡張2) *rd* ← *rs* & *imm26*
拡張3) *rd* ← *rs1* & *rs2*

コード



0x32__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	*1	↔	↔

モード

Src　レジスタ直接 *%rs* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK

1サイクル

説　明

(1)標準

```
and  %rd,%rs       ; rd ← rd & rs
```

*rs*レジスタの内容と*rd*レジスタの内容の論理積をとり、結果を*rd*レジスタにロードします。

(2)拡張1

```
ext  imm13
and  %rd,%rs       ; rd ← rs & imm13
```

*rs*レジスタの内容とゼロ拡張した13ビット即値*imm13*の論理積をとり、結果を*rd*レジスタにロードします。*rs*レジスタの内容は変更されません。

(3)拡張2

```
ext  imm13         ; = imm26(25:13)
ext  imm13         ; = imm26(12:0)
and  %rd,%rs       ; rd ← rs & imm26
```

*rs*レジスタの内容とゼロ拡張した26ビット即値*imm26*の論理積をとり、結果を*rd*レジスタにロードします。*rs*レジスタの内容は変更されません。

(4)拡張3

```
ext  %rs2
and  %rd,%rs1      ; rd ← rs1 & rs2
```

*rs1*レジスタの内容とext命令で指定されたレジスタ*rs2*の論理積をとり、結果を*rd*レジスタにロードします。*rs1*レジスタ、*rs2*レジスタの内容は変更されません。

(5)ディレイド命令

本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。この場合はext命令による拡張は行えません。

---

*1 PSRのOCフラグを1にセットして本命令を実行すると、Vフラグが0にクリアされます。他の論理演算系の命令についても同様に機能します。（or命令、xor命令、not命令。命令の機能は各命令の説明を参照してください。）

例

```
(1)and  %r0,%r0       ; r0 = r0 & r0

(2)ext  0x1
   ext  0x1fff
   and  %r1,%r2       ; r1 = r2 & 0x00003fff

(3)ext  %r5
   and  %r3,%r4       ; r3 = r4 & r5
```

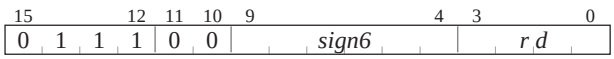
and *%rd, sign6*

機　能

論理積

標準） *rd* ← *rd* & *sign6*
拡張1) *rd* ← *rd* & *sign19*
拡張2) *rd* ← *rd* & *sign32*
拡張3) 不可

コード



0x70__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
−	−	−	−	−	−	−	−	−	−	*1	↔	↔

モード

Src　即値(符号付き)
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK

1サイクル

説　明

(1)標準

```
and  %rd,sign6     ; rd ← rd & sign6
```

*rd*レジスタの内容と符号拡張した6ビット即値*sign6*の論理積をとり、結果を*rd*レジスタにロードします。

(2)拡張1

```
ext  imm13         ; = sign19(18:6)
and  %rd,sign6     ; rd ← rd & sign19, sign6 = sign19(5:0)
```

*rd*レジスタの内容と符号拡張した19ビット即値*sign19*の論理積をとり、結果を*rd*レジスタにロードします。

(3)拡張2

```
ext  imm13         ; = sign32(31:19)
ext  imm13         ; = sign32(18:6)
and  %rd,sign6     ; rd ← rd & sign32, sign6 = sign32(5:0)
```

*rd*レジスタの内容と32ビット即値*sign32*の論理積をとり、結果を*rd*レジスタにロードします。

(4)ディレイド命令
本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。この場合はext命令による拡張は行えません。

---

*1 PSRのOCフラグを1にセットして本命令を実行すると、Vフラグが0にクリアされます。他の論理演算系の命令についても同様に機能します。(or命令、xor命令、not命令。命令の機能は各命令の説明を参照してください。)

例

(1)
```
and  %r0,0x3e      ; r0 = r0 & 0xfffffffe
```

(2)
```
ext  0x7ff
and  %r1,0x3f      ; r1 = r1 & 0x0001ffff
```

bclr [*%rb*], *imm3*

機　能

ビットクリア

標準） B[*rb*](*imm3*) ← 0
拡張1) B[*rb* + *imm13*](*imm3*) ← 0
拡張2) B[*rb* + *imm26*](*imm3*) ← 0
拡張3) 不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	1	1	0	0	*rb*				0	*imm3*		

0xAC__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

Src　即値（符号なし）
Dst　レジスタ直接 *%rb* = %r0～%r15

CLK

3サイクル

説　明

(1)標準

```
bclr  [%rb],imm3  ; B[rb](imm3) ← 0
```

指定メモリのデータビットをクリアします。*rb*レジスタの内容がアクセスされるメモリアドレスとなります。3ビット即値*imm3*はクリアするビット番号（バイトデータのビット7～0の1つ）を指定します。

(2)拡張1

```
ext   imm13
bclr  [%rb],imm3  ; B[rb + imm13](imm3) ← 0
```

ext命令により、アドレッシングモードがディスプレースメント付きレジスタ間接アドレッシングに変わります。これにより、*rb*レジスタの内容に13ビット即値*imm13*を加えたアドレス上の、*imm3*で指定されたデータビットをクリアします。*rb*レジスタの内容は変更されません。

(3)拡張2

```
ext   imm13       ; = imm26(25:13)
ext   imm13       ; = imm26(12:0)
bclr  [%rb],imm3  ; B[rb + imm26](imm3) ← 0
```

アドレッシングモードがディスプレースメント付きレジスタ間接アドレッシングに変わり、*rb*レジスタの内容に26ビット即値*imm26*を加えたアドレス上の、*imm3*で指定されたデータビットをクリアします。*rb*レジスタの内容は変更されません。

例

```
(1)ld.w  %r0,[%sp+0x10]   ;アクセスするメモリアドレスをR0レジスタに設定
   bclr  [%r0],0x0        ;指定アドレス上のデータのビット0をクリア
(2)ext   0x1
   bclr  [%r0],0x7        ;上記アドレスの次のアドレスのデータビット7をクリア
```

bnot [*%rb*], *imm3*

機　能

ビット反転

標準） B[*rb*](*imm3*) ← !B[*rb*](*imm3*)
拡張1) B[*rb* + *imm13*](*imm3*) ← !B[*rb* + *imm13*](*imm3*)
拡張2) B[*rb* + *imm26*](*imm3*) ← !B[*rb* + *imm26*](*imm3*)
拡張3) 不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	1	0	0	*r b*				0	*imm3*		

0xB4__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
−	−	−	−	−	−	−	−	−	−	−	−	−

モード

Src　即値（符号なし）
Dst　レジスタ直接 *%rb* = %r0～%r15

CLK

3サイクル

説　明

(1)標準

```
bnot  [%rb],imm3  ; B[rb](imm3) ← !B[rb](imm3)
```

指定メモリのデータビットを反転(1↔0)します。*rb*レジスタの内容がアクセスされるメモリアドレスとなります。3ビット即値*imm3*は反転するビット番号（バイトデータのビット7～0の1つ）を指定します。

(2)拡張1

```
ext   imm13
bnot  [%rb],imm3  ; B[rb + imm13](imm3) ← !B[rb + imm13](imm3)
```

ext命令により、アドレッシングモードがディスプレースメント付きレジスタ間接アドレッシングに変わります。これにより、*rb*レジスタの内容に13ビット即値*imm13*を加えたアドレス上の、*imm3*で指定されたデータビットを反転します。*rb*レジスタの内容は変更されません。

(3)拡張2

```
ext   imm13       ; = imm26(25:13)
ext   imm13       ; = imm26(12:0)
bnot  [%rb],imm3  ; B[rb + imm26](imm3) ← !B[rb + imm26](imm3)
```

アドレッシングモードがディスプレースメント付きレジスタ間接アドレッシングに変わり、*rb*レジスタの内容に26ビット即値*imm26*を加えたアドレス上の、*imm3*で指定されたデータビットを反転します。*rb*レジスタの内容は変更されません。

例

```
(1)ld.w  %r0,[%sp+0x10]    ;アクセスするメモリアドレスをR0レジスタに設定
   bnot  [%r0],0x0         ;指定アドレス上のデータのビット0を反転
(2)ext   0x1
   bnot  [%r0],0x7         ;上記アドレスの次のアドレスのデータビット7を反転
```

brk

機　能　**デバッグ例外**

標準）W[0x8(or 0x60008)] ← pc + 2, W[0xC(or 0x6000C)] ← r0,
pc ← W[0x0(or 0x60000)]
拡張1）不可
拡張2）不可
拡張3）不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0

0x0400

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	1	–	–	–	–	–	–	–

モード　–

CLK　7サイクル

説　明　デバッグ処理ルーチンを呼び出すソフトウェア例外です。
次の命令のアドレスと汎用レジスタR0をデバッグ用スタックにセーブ後、デバッグ用ベクタアドレス0x0000000（もしくは0x0060000）からデバッグルーチンへのベクタを読み出してPCにロードします。これによりデバッグ処理ルーチンに分岐します。また、CPUはデバッグモードに移行します。
デバッグ処理ルーチンよりのリターンには`retd`命令を使用します。
本命令はICE制御ソフト専用です。一般のプログラムでは使用しません。

例

```
brk             ; デバッグ処理ルーチンを実行
```

bset [*%rb*], *imm3*

機　能

ビットセット

標準） B[*rb*](*imm3*) ← 1
拡張1) B[*rb* + *imm13*](*imm3*) ← 1
拡張2) B[*rb* + *imm26*](*imm3*) ← 1
拡張3) 不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	0	0	*r b*				0	*imm3*		

0xB0__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
−	−	−	−	−	−	−	−	−	−	−	−	−

モード

Src　即値(符号なし)
Dst　レジスタ直接 *%rb* = %r0～%r15

CLK

3サイクル

説　明

(1)標準

```
bset  [%rb],imm3  ; B[rb](imm3) ← 1
```

指定メモリのデータビットをセットします。*rb*レジスタの内容がアクセスされるメモリアドレスとなります。3ビット即値*imm3*はセットするビット番号(バイトデータのビット7～0の1つ)を指定します。

(2)拡張1

```
ext   imm13
bset  [%rb],imm3  ; B[rb + imm13](imm3) ← 1
```

ext命令により、アドレッシングモードがディスプレースメント付きレジスタ間接アドレッシングに変わります。これにより、*rb*レジスタの内容に13ビット即値*imm13*を加えたアドレス上の、*imm3*で指定されたデータビットをセットします。*rb*レジスタの内容は変更されません。

(3)拡張2

```
ext   imm13       ; = imm26(25:13)
ext   imm13       ; = imm26(12:0)
bset  [%rb],imm3  ; B[rb + imm26](imm3) ← 1
```

アドレッシングモードがディスプレースメント付きレジスタ間接アドレッシングに変わり、*rb*レジスタの内容に26ビット即値*imm26*を加えたアドレス上の、*imm3*で指定されたデータビットをセットします。*rb*レジスタの内容は変更されません。

例

```
(1)ld.w  %r0,[%sp+0x10]    ; アクセスするメモリアドレスをR0レジスタに設定
   bset  [%r0],0x0         ; 指定アドレス上のデータのビット0をセット
(2)ext   0x1
   bset  [%r0],0x7         ; 上記アドレスの次のアドレスのデータビット7をセット
```

btst [*%rb*], *imm3*

機　能

ビットテスト

標準） Z flag ← 1 if B[*rb*](*imm3*) = 0 else Z flag ← 0
拡張1) Z flag ← 1 if B[*rb* + *imm13*](*imm3*) = 0 else Z flag ← 0
拡張2) Z flag ← 1 if B[*rb* + *imm26*](*imm3*) = 0 else Z flag ← 0
拡張3) 不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	1	0	0	0	*r*	*b*			0	*imm3*		

0xA8__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	↔	–

モード

Src　即値(符号なし)
Dst　レジスタ直接 *%rb* = %r0～%r15

CLK

3サイクル

説　明

(1)標準

```
btst  [%rb],imm3  ; Z flag ← 1 if B[rb](imm3) = 0
                  ; else Z flag ← 0
```

指定メモリのデータビットをテストし、それが0ならばZ(ゼロ)フラグをセットします。*rb*レジスタの内容がアクセスされるメモリアドレスとなります。3ビット即値*imm3*はテストするビット番号(バイトデータのビット7～0の1つ)を指定します。

(2)拡張1

```
ext   imm13
btst  [%rb],imm3  ; Z flag ← 1 if B[rb + imm13](imm3) = 0
                  ; else Z flag ← 0
```

ext命令により、アドレッシングモードがディスプレースメント付きレジスタ間接アドレッシングに変わります。これにより、*rb*レジスタの内容に13ビット即値*imm13*を加えたアドレス上の、*imm3*で指定されたデータビットをテストします。*rb*レジスタの内容は変更されません。

(3)拡張2

```
ext   imm13       ; = imm26(25:13)
ext   imm13       ; = imm26(12:0)
btst  [%rb],imm3  ; Z flag ← 1 if B[rb + imm26](imm3) = 0
                  ; else Z flag ← 0
```

アドレッシングモードがディスプレースメント付きレジスタ間接アドレッシングに変わり、*rb*レジスタの内容に26ビット即値*imm26*を加えたアドレス上の、*imm3*で指定されたデータビットをテストします。*rb*レジスタの内容は変更されません。

例

```
ld.w  %r0,[%sp+0x10] ; アクセスするメモリアドレスをR0レジスタに設定
btst  [%r0],0x7      ; 指定アドレス上のデータのビット7をテスト
jreq  POSITIVE       ; ビットが0ならばジャンプ
```

call *%rb* / call.d *%rb*

機　能

サブルーチンコール

標準） sp ← sp - 4, W[sp] ← pc + 2, pc ← *rb*
拡張1）不可
拡張2）不可
拡張3）不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	d	0	0	0	0	*r*	*b*		

0x060_, 0x070_

d（ビット8）= 0の場合　`call`　`%rb`
d（ビット8）= 1の場合　`call.d` `%rb`

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

レジスタ直接 `%rb` = `%r0`～`%r15`

CLK

`call`　3サイクル
`call.d`　2サイクル

説　明

(1)標準

```
call  %rb
```

次の命令のアドレスをスタックにセーブ後、*rb*レジスタの内容をPCにロードして、そのアドレスから始まるサブルーチンをコールします。*rb*レジスタの最下位ビットは無視され、常に0として扱われます。サブルーチンで`ret`命令を実行すると、`call`の次の命令にリターンします。

(2)ディレイド分岐（dビット = 1）

```
call.d  %rb
```

`call.d`命令では次の命令がディレイド命令となります。ディレイド命令はサブルーチンへの分岐前に実行されます。このため、スタックにセーブされるリターンアドレスは、ディレイド命令の次の命令のアドレス（PC + 4）となります。`call.d`命令と次のディレイド命令の間はトラップがマスクされ、割り込みや例外は発生しません。

例

```
call  %r0        ; R0レジスタの内容を先頭アドレスとするサブルーチンをコール
```

注　意

`call.d`命令（ディレイド分岐）を使用する場合、次の命令はディレイド命令として使用可能な命令に限られます。それ以外の命令を実行した場合、動作は不定となりますので注意してください。使用可能な命令については、Appendixの命令一覧表を参照してください。

call *sign8* / call.d *sign8*

機　能

サブルーチンコール

標準）sp ← sp - 4, W[sp] ← pc + 2, pc ← pc + *sign8* × 2
拡張1）sp ← sp - 4, W[sp] ← pc + 2, pc ← pc + *sign22*
拡張2）sp ← sp - 4, W[sp] ← pc + 2, pc ← pc + *sign32*
拡張3）不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	d	*sign8*							

0x1C__, 0x1D__

d(ビット8)= 0の場合　`call`　*sign8*
d(ビット8)= 1の場合　`call.d` *sign8*

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

符号付きPC相対

CLK

`call`　2サイクル
`call.d`　1サイクル

説　明

(1)標準

```
call  sign8  ; = "call sign9", sign8 = sign9(8:1), sign9(0) = 0
```

次の命令のアドレスをスタックにセーブ後、PCに符号付き8ビット即値*sign8*を2倍して加算し、そのアドレスから始まるサブルーチンをコールします。*sign8*は16ビット単位のハーフワードアドレスを指定します。サブルーチンで`ret`命令を実行すると、`call`の次の命令にリターンします。*sign8*(×2)による分岐可能範囲はPC - 0x100～PC + 0xFEです。

(2)拡張1

```
ext   imm13  ; = sign22(21:9)
call  sign8  ; = "call sign22", sign8 = sign22(8:1), sign22(0) = 0
```

`ext`命令の13ビット即値*imm13*が符号拡張され、PCに加算するディスプレースメントが符号付き22ビットとなります。*sign22*による分岐可能範囲はPC - 0x200000～PC + 0x1FFFFEです。

(3)拡張2

```
ext   imm13  ; imm13(12:3)= sign32(31:22)
ext   imm13  ; = sign32(21:9)
call  sign8  ; = "call sign32", sign8 = sign32(8:1), sign32(0) = 0
```

`ext`命令の2つの13ビット即値(*imm13* × 2)が符号拡張され、PCに加算するディスプレースメントが符号付き32ビットとなります。これにより、全アドレス空間をカバーします。

(4)ディレイド分岐(dビット = 1)

```
call.d  sign8
```

`call.d`命令では命令コード中のdビットがセットされ、次の命令がディレイド命令となります。ディレイド命令はサブルーチンへの分岐前に実行されます。このため、スタックにセーブされるリターンアドレスは、ディレイド命令の次の命令のアドレス(PC + 4)となります。`call.d`命令と次のディレイド命令の間はトラップがマスクされ、割り込みや例外は発生しません。

例

```
ext   0x1fff
call  0x0       ; PC - 0x200番地から始まるサブルーチンをコール
```

注　意

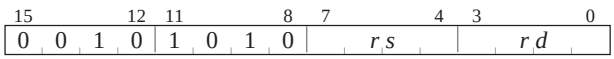
`call.d`命令(ディレイド分岐)を使用する場合、次の命令はディレイド命令として使用可能な命令に限られます。それ以外の命令を実行した場合、動作は不定となりますので注意してください。使用可能な命令については、Appendixの命令一覧表を参照してください。

cmp *%rd, %rs*

機　能　**比較**

標準）*rd - rs*
拡張1）*rs - imm13*
拡張2）*rs - imm26*
拡張3）不可

コード



0x2A__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	↔	↔	↔	↔

モード　Src　レジスタ直接 *%rs* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK　1サイクル

説　明　(1)標準

```
cmp  %rd,%rs       ; rd - rs
```

*rd*レジスタの内容から*rs*レジスタの内容を減算し、結果によりフラグ(C、V、Z、N)をセット/リセットします。*rd*レジスタは変更されません。

(2)拡張1

```
ext  imm13
cmp  %rd,%rs       ; rs - imm13
```

*rs*レジスタの内容から13ビット即値*imm13*を減算し、結果によりフラグ(C、V、Z、N)をセット/リセットします。*rd*および*rs*レジスタは変更されません。

(3)拡張2

```
ext  imm13         ; = imm26(25:13)
ext  imm13         ; = imm26(12:0)
cmp  %rd,%rs       ; rs - imm26
```

*rs*レジスタの内容から26ビット即値*imm26*を減算し、結果によりフラグ(C、V、Z、N)をセット/リセットします。*rd*および*rs*レジスタは変更されません。

(4)ディレイド命令
本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。この場合はext命令による拡張は行えません。

例　(1)`cmp  %r0,%r1 ; r0 - r1`、結果によりフラグを変更

(2)
```
ext  0x1
ext  0x1fff
cmp  %r1,%r2 ; r2 - 0x3fff、結果によりフラグを変更
```

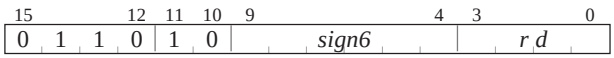
cmp *%rd, sign6*

機　能

比較

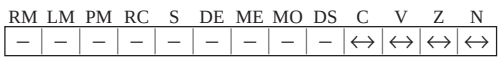
標準） *rd - sign6*
拡張1) *rd - sign19*
拡張2) *rd - sign32*
拡張3) 不可

コード



0x68__

フラグ



モード

Src　即値(符号付き)
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK

1サイクル

説　明

(1)標準

```
cmp  %rd,sign6     ; rd - sign6
```

*rd*レジスタの内容から6ビット即値*sign6*を符号拡張して減算し、結果によりフラグ(C、V、Z、N)をセット/リセットします。*rd*レジスタは変更されません。

(2)拡張1

```
ext  imm13         ; = sign19(18:6)
cmp  %rd,sign6     ; rd - sign19, sign6 = sign19(5:0)
```

*rd*レジスタの内容から19ビット即値*sign19*を符号拡張して減算し、結果によりフラグ(C、V、Z、N)をセット/リセットします。*rd*レジスタは変更されません。

(3)拡張2

```
ext  imm13         ; = sign32(31:19)
ext  imm13         ; = sign32(18:6)
cmp  %rd,sign6     ; rd - sign32, sign6 = sign32(5:0)
```

*rd*レジスタの内容からext命令により拡張した符号付き32ビット即値*sign32*を減算し、結果によりフラグ(C、V、Z、N)をセット/リセットします。*rd*レジスタは変更されません。

(4)ディレイド命令
本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。この場合はext命令による拡張は行えません。

例

(1)
```
cmp  %r0,0x3f      ;r0 - 0x3f、結果によりフラグを変更
```

(2)
```
ext  0x1fff
ext  0x1fff
cmp  %r1,0x3f      ;r1 - 0xffffffff、結果によりフラグを変更
```

div0s *%rs*

機　能

符号付き除算第1ステップ
標準）除算実行用の初期化処理
拡張1）不可
拡張2）不可
拡張3）不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	0	1	1	*rs*				0	0	0	0

0x8B_0

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	↔	–	–	–	↔

モード　レジスタ直接 *%rs* = %r0～%r15

CLK　1サイクル

説　明

符号付き除算を行う場合は、ALRに被除数、*rs*レジスタに除数を設定し、最初にdiv0s命令を実行します。div0s命令は以下の初期化処理を行います。
1）ALRの被除数を{AHR, ALR}の64ビットに符号拡張します。
2）PSRのDSフラグに、被除数の符号ビット（ALRの最上位ビット）をセットします。
3）PSRのNフラグに、除数の符号ビット（*rs*レジスタの最上位ビット）をセットします。

このため、ALRと*rs*レジスタに用意しておく被除数と除数は32ビットに符号拡張されていることが必要です。
div0s命令実行後はdiv1命令による除算を実行します。符号付き除算では、その後にdiv2s命令およびdiv3s命令による補正が必要です。

例

符号付き除算例（32ビット÷32ビット）
R0レジスタに被除数、R1レジスタに除数が設定されている場合

```
ld.w   %alr,%r0 ; ALRに被除数を設定
div0s  %r1      ; 符号付き除算の初期化ステップ
div1   %r1      ; div1を32回実行
 :       :
div1   %r1
div2s  %r1      ; 補正命令1
div3s           ; 補正命令2
```

上記命令実行後、商がALRから、余りがAHRから得られます。

注　意

（1）*rs*レジスタに0を設定してdiv0s命令を実行すると、ゼロ除算例外が発生します。
被除数、除数とも演算可能なデータサイズは32ビットまでです。

（2）div0s命令実行直後にld.w命令でPSRを読み出した場合、DSフラグが正しく読み出せないことがあります。DSフラグを正しく読み出すには、div0s命令とld.w命令の間に2つ以上の命令を挿入してください。

div0u *%rs*

機　能

符号なし除算第1ステップ
標準）除算実行用の初期化処理
拡張1）不可
拡張2）不可
拡張3）不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	1	1	1	*rs*				0	0	0	0

0x8F_0

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	0	–	–	–	0

モード

レジスタ直接 *%rs* = %r0～%r15

CLK

1サイクル

説　明

符号なし除算を行う場合は、ALRに被除数、*rs*レジスタに除数を設定し、最初にdiv0u命令を実行します。div0u命令は以下の初期化処理を行います。
1）AHRをオール0にクリアします。
2）PSRのDSフラグをリセット(0)します。
3）PSRのNフラグをリセット(0)します。

div0u命令実行後はdiv1命令による除算を実行します。符号なし除算では、div1命令によって除算結果が得られ、その後の補正は不要です。

例

符号なし除算例(32ビット÷32ビット)
R0レジスタに被除数、R1レジスタに除数が設定されている場合

```
ld.w   %alr,%r0 ; ALRに被除数を設定
div0u  %r1      ; 符号なし除算の初期化ステップ
div1   %r1      ; div1を32回実行
 :       :
div1   %r1
```

上記命令実行後、商がALRから、余りがAHRから得られます。

注　意

(1) *rs*レジスタに0を設定してdiv0u命令を実行すると、ゼロ除算例外が発生します。
被除数、除数とも演算可能なデータサイズは32ビットまでです。

(2) div0u命令実行直後にld.w命令でPSRを読み出した場合、DSフラグが正しく読み出せないことがあります。DSフラグを正しく読み出すには、div0u命令とld.w命令の間に2つ以上の命令を挿入してください。

div1 *%rs*

機　能

除算

標準）ステップ除算実行
拡張1）不可
拡張2）不可
拡張3）不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	0	1	1	*rs*				0	0	0	0

0x93_0

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

レジスタ直接 *%rs* = %r0～%r15

CLK

1サイクル

説　明

div1は符号付き除算、符号なし除算に共通なステップ除算実行命令です。本命令は、div0s命令（符号付き除算）またはdiv0u（符号なし除算）の初期化処理を実行後、被除数のデータサイズに従った回数を実行する必要があります。たとえば32ビット÷32ビットの場合は32回、16ビット÷16ビットの場合は16回実行します。
div1命令は1ステップで以下の処理を行います。

1）{AHR, ALR}の64ビットを左（上位側）に1ビットシフト（ALR(0) = 0）

2）AHRと*rs*を加算またはAHRから*rs*を減算し、結果によりAHR、ALRを再設定
加減算はAHRの内容にDSフラグを符号ビットとして付加した33ビットと、*rs*レジスタの内容にNフラグを符号ビットとして付加した33ビットデータによって行います。
この処理はPSRのDSフラグおよびNフラグによって以下のように異なります。なお、演算結果の33ビット目の値をtmp(32)として説明します。

DS = 0（被除数: 正）、N = 0（除数: 正）の場合
　2-1）tmp = {0, AHR} - {0, *rs*}を実行
　2-2）tmp(32) = 0の場合: AHR = tmp(31:0)、ALR(0) = 1として終了
　　tmp(32) = 1の場合: AHR、ALRをそのままに終了

DS = 1（被除数: 負）、N = 0（除数: 正）の場合
　2-1）tmp = {1, AHR} + {0, *rs*}を実行
　2-2）tmp(32) = 1の場合: AHR = tmp(31:0)、ALR(0) = 1として終了
　　tmp(32) = 0の場合: AHR、ALRをそのままに終了

DS = 0（被除数: 正）、N = 1（除数: 負）の場合
　2-1）tmp = {0, AHR} + {1, *rs*}を実行
　2-2）tmp(32) = 0の場合: AHR = tmp(31:0)、ALR(0) = 1として終了
　　tmp(32) = 1の場合: AHR、ALRをそのままに終了

DS = 1（被除数: 負）、N = 1（除数: 負）の場合
　2-1）tmp = {1, AHR} - {1, *rs*}を実行
　2-2）tmp(32) = 1の場合: AHR = tmp(31:0)、ALR(0) = 1として終了
　　tmp(32) = 0の場合: AHR、ALRをそのままに終了

符号なし除算の場合は、div1命令を必要数実行することにより結果が次のレジスタから得られます。
　符号なし除算結果: ALR = 商　AHR = 余り

符号付き除算では、この後にdiv2s命令およびdiv3s命令による補正が必要です。

例

符号なし除算例(32ビット÷32ビット)
R0レジスタに被除数、R1レジスタに除数が設定されている場合

```
ld.w   %alr,%r0 ; ALRに被除数を設定
div0u  %r1      ; 符号なし除算の初期化ステップ
div1   %r1      ; div1を32回実行
 :      :
div1   %r1
```

上記命令実行後、商がALRから、余りがAHRから得られます。

符号付き除算例(32ビット÷32ビット)
R0レジスタに被除数、R1レジスタに除数が設定されている場合

```
ld.w   %alr,%r0 ; ALRに被除数を設定
div0s  %r1      ; 符号付き除算の初期化ステップ
div1   %r1      ; div1を32回実行
 :      :
div1   %r1
div2s  %r1      ; 補正命令1
div3s           ; 補正命令2
```

上記命令実行後、商がALRから、余りがAHRから得られます。

div2s *%rs*

機　能

符号付き除算結果補正1

標準）符号付き除算実行結果の補正処理
拡張1）不可
拡張2）不可
拡張3）不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	1	1	1	*rs*				0	0	0	0

0x97_0

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

レジスタ直接 *%rs* = %r0～%r15

ＣＬＫ

1サイクル

説　明

div2s命令は符号付き除算結果の補正を行います。符号なし除算の場合はdiv2s命令を実行する必要はありません。

被除数が負の数の場合に除算のステップ(div1命令の実行)で演算結果がゼロになると、全ステップ終了後の演算結果が、除数と同じ余り(AHR)、および実際の値よりも絶対値で1少ない商(ALR)となる可能性があります。div2s命令はこの結果を補正します。
div2s命令の動作は次のとおりです。

DS = 0(被除数: 正)の場合
　被除数が正の場合は上記の問題は発生しません。したがって、div2s命令は何も実行せずに終了します(nop命令と同じ)。

DS = 1(被除数: 負)の場合
1）N = 0(除数: 正)の場合、tmp = AHR + *rs*を実行
　N = 1(除数: 負)の場合、tmp = AHR - *rs*を実行
2）1)の演算結果により、
　tmpがゼロの場合:　　AHR = tmp(31:0)、ALR = ALR + 1として終了
　tmpがゼロ以外の場合: AHR、ALRをそのままに終了

例

符号付き除算例(32ビット÷32ビット)
R0レジスタに被除数、R1レジスタに除数が設定されている場合

```
ld.w   %alr,%r0 ; ALRに被除数を設定
div0s  %r1      ; 符号付き除算の初期化ステップ
div1   %r1      ; div1を32回実行
 :       :
div1   %r1
div2s  %r1      ; 補正命令1
div3s           ; 補正命令2
```

上記命令実行後、商がALRから、余りがAHRから得られます。

div3s

機　能　**符号付き除算結果補正2**

標準）符号付き除算実行結果の補正処理
拡張1）不可
拡張2）不可
拡張3）不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	1	1	*rs*				0	0	0	0

0x9B_0

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード　–

CLK　1サイクル

説　明　`div3s`命令は符号付き除算結果の補正を行います。符号なし除算の場合は`div3s`命令を実行する必要はありません。

ステップ除算によりALRから得られる商は常に正の数になります。被除数と除数の符号が異なる場合、結果は負でなければなりません。`div3s`命令はこの場合の符号補正を行います。`div3s`命令の動作は次のとおりです。

DS = N（被除数と除数が同符号）の場合
　この場合は上記の問題は発生しません。したがって、`div3s`命令は何も実行せずに終了します（`nop`命令と同じ）。

DS = ! N（被除数と除数の符号が異なる）の場合
　ALR（商）の符号を反転します。

`div2s`命令および`div3s`命令実行後、符号付き除算の最終結果が次のレジスタから得られます。
　ALR = 商　AHR = 余り

例　符号付き除算例（32ビット÷32ビット）
R0レジスタに被除数、R1レジスタに除数が設定されている場合

```
ld.w   %alr,%r0 ; ALRに被除数を設定
div0s  %r1      ; 符号付き除算の初期化ステップ
div1   %r1      ; div1を32回実行
 :       :
div1   %r1
div2s  %r1      ; 補正命令1
div3s           ; 補正命令2
```

上記命令実行後、商がALRから、余りがAHRから得られます。

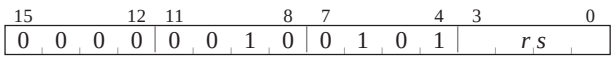
div.w *%rs*

機　能

符号付き32ビット除算

標準） ALR ← ALR / *rs*（商）, AHR ← ALR / *rs*（剰余）
拡張1）不可
拡張2）不可
拡張3）不可

コード



0x025_

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	↔	–	–	–	↔

モード

レジスタ直接 *%rs* = %r0～%r15

ＣＬＫ

35サイクル

説　明

ALRレジスタを被除数、*rs*レジスタを除数として符号付き32ビット除算を行います。
結果は、ALRに商、AHRに剰余が入ります。

例

r0 = -20、r1 = 6の場合

```
ld.w   %alr,%r0      ; ALR ← -20
div.w  %r1           ; ALR ← -3, AHR ← -2
```

注　意

LCフラグ（PSRのビット17）が1の場合、ALRレジスタに書き込まれるデータは、ALR以外にR4レジスタにも書き込まれます。HCフラグ（PSRのビット16）が1のときは、AHRレジスタに書き込まれるデータは、AHR以外にR5レジスタにも書き込まれます。

divu.w *%rs*

機　能

符号なし32ビット除算

標準） ALR ← ALR / *rs*（商）, AHR ← ALR / *rs*（剰余）
拡張1）不可
拡張2）不可
拡張3）不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	0	0	0	1	*r*	*s*		

0x021_

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	0	–	–	–	0

モード

レジスタ直接 *%rs* = %r0～%r15

CLK

35サイクル

説　明

ALRレジスタを被除数、*rs*レジスタを除数として符号なし32ビット除算を行います。
結果は、ALRに商、AHRに剰余が入ります。

例

r0 = 20、r1 = 6の場合

```
ld.w   %alr,%r0      ; ALR ← 20
divu.w %r1           ; ALR ← 3, AHR ← 2
```

注　意

LCフラグ（PSRのビット17）が1の場合、ALRレジスタに書き込まれるデータは、ALR以外にR4レジスタにも書き込まれます。HCフラグ（PSRのビット16）が1のときは、AHRレジスタに書き込まれるデータは、AHR以外にR5レジスタにも書き込まれます。

do.c *imm6*

機　能　**コプロセッサ実行**

標準） W[CA(*imm6*)]
拡張1）不可
拡張2）不可
拡張3）不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	0	*imm6*					

0xBF0_

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード　即値（符号なし）

ＣＬＫ　1サイクル

説　明　コプロセッサに対し、*imm6*で指定されるコマンドを発行します。
*imm6*はコプロセッサ専用のアドレスバスに出力されます。

例

```
do.c  0x1a        ; コプロセッサでコマンド1Aを実行
```

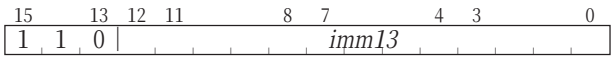
ext *imm13*

機　能

即値拡張

標準）次の命令の即値/オペランドを拡張
拡張1）不可
拡張2）不可
拡張3）不可

コード



0xC0__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

即値（符号なし）

CLK

0または1サイクル（プリデコード時は0サイクル）

説　明

直後の命令の即値あるいはオペランドを拡張します。
即値拡張の場合、ext命令で指定した即値が上位側、ターゲット命令（拡張対象命令）が持つ即値が下位側となります。

ext命令は連続2個まで使用可能です。その場合は、最初のext命令で指定した即値が最も上位側となります。3個以上連続して記述した場合は、最初のext命令とターゲット命令の直前にあるext命令の2個が有効となり、途中のext命令は無効となります。
ext命令による拡張内容および使用例については各命令の説明を参照してください。

ext命令に対する例外（リセット、デバッグブレークを除く）はハードウェアによりマスクされ、拡張対象命令実行時に例外処理が受け付けられます。ただし、この場合の例外処理からのリターンアドレスはext命令の先頭になります。

例

```
ext  0x1000     ; 有効
ext  0x1        ; 無効
ext  0x1fff     ; 有効
add  %r1,0x3f   ; r1 = r1 + 0x8007ffff
```

注　意

ext命令に続けてメモリとレジスタ間のロード命令を実行する場合、そのロード命令の実行前にアドレス不整例外が発生する可能性があります（ext命令で指定した即値をディスプレースメントとする指定アドレスが、転送データサイズのアドレス境界を指していない場合）。ここでアドレス不整例外が発生し、それによって実行されたトラップ処理ルーチンを単純にreti命令で終了させると、そのトラップ処理ルーチンからは例外を発生したロード命令のアドレスに戻り、直前のext命令が無効となります。したがって、リターンアドレスの操作等が必要になりますので注意してください。

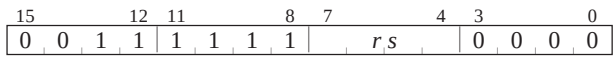
ext *%rs*

機　能

オペランド拡張

標準）3オペランドへの拡張
拡張1）不可
拡張2）不可
拡張3）不可

コード



0x3F_0

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード　レジスタ直接 *%rs* = %r0～%r15

CLK　0または1サイクル（プリデコード時は0サイクル）

説　明　直後の命令のオペランドに*rs*で指定されるレジスタを追加し、3オペランド命令として機能させます。

拡張可能な命令

ALU命令	アドレッシングモード
add, sub, adc, sbc, and, or, xor	*%rd*,*%rs*

シフト命令	アドレッシングモード
srl, sll, sra, sla, rr, rl	*%rd*,*imm5*
srl, sll, sra, sla, rr, rl	*%rd*,*%rs*

ロード命令	アドレッシングモード
ld.b, ld.ub, ld.h, ld.uh, ld.w	*%rd*,[*%rb*]
ld.b, ld.ub, ld.h, ld.uh, ld.w	[*%rb*],*%rs*
ld.b, ld.ub, ld.h, ld.uh, ld.w	*%rd*,[*%rb*]+
ld.b, ld.ub, ld.h, ld.uh, ld.w	[*%rb*]+,*%rs*

ext命令に対する例外（リセット、デバッグブレークを除く）はハードウェアによりマスクされ、拡張対象命令実行時に例外処理が受け付けられます。ただし、この場合の例外処理からのリターンアドレスはext命令の先頭になります。

例

```
(1)ext   %r3
   add   %r1,%r2      ; r1 ← r2 + r3

(2)ext   %r2
   sla   %r3,5        ; r3(31:5) ← r2 << 5, r3(4:0) ← 0000

(3)ext   %r3
   ld.h  %r5,[%r6]    ; r5 ← [r6 + r3]

(4)ext   %r4
   ld.w  [%r7]+,%r8   ; [r7] ← r8, r7 ← r7 + r4
```

ext *%rs*, *op*, *imm2*

機　能

ポストシフト付きオペランド拡張

標準）オペランドを拡張、add/sub等演算命令のポストシフト
拡張1）不可
拡張2）不可
拡張3）不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	1	1	1	1	1	1	*rs*				*op*		*imm2*		0x3F__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

Src　レジスタ直接 *%rs* = %r0～%r15
op　シフトモード（sra、srl、sll）
imm2　シフト量

CLK

0または1サイクル（プリデコード時は0サイクル）

説　明

直後の命令のオペランドに*rs*で指定されるレジスタを追加し、3オペランド命令として機能させます。また、*op*および*imm2*を指定することで、下記演算命令の実行結果を最大3ビットシフトします。シフトの方向、種類は*op*によりsra、srl、sllを指定できます。シフト動作は、sra、srl、sllの各命令と同じです。ただし、シフトアウトレジスタSORを使用しませんのでSORは変化しません。また、C、V、Z、Nフラグは拡張対象命令の演算結果によってのみ変化し、シフト動作の影響は受けません。

ext命令に対する例外（リセット、デバッグブレークを除く）はハードウェアによりマスクされ、拡張対象命令実行時に例外処理が受け付けられます。ただし、この場合の例外処理からのリターンアドレスはext命令の先頭になります。

***op*, *imm2*が有効に機能する命令**

```
sub  %rd,%rs    sbc  %rd,%rs
add  %rd,%rs    adc  %rd,%rs    add  %rd,%dp
```

例

```
(1)ext  %r3,sll,2
   add  %r1,%r2       ; r1(31:2) = (r2 + r3) << 2, r1(1:0) = 00

(2)ext  %r1,srl,3
   ext  0x05
   sub  %r2,%r3       ; %r2(28:0) = (r3 – 5) >> 3, r2(31:29) = 000
```

ext *op, imm2*

機　能

ポストシフト付き即値拡張

標準）add/adc/sub演算命令のポストシフト
拡張1）不可
拡張2）不可
拡張3）不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	1	1	0	0	0	0	*op*		*imm2*	

0x3B0_

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

op　　シフトモード（sra、srl、sll）
imm2　シフト量

CLK

0または1サイクル（プリデコード時は0サイクル）

説　明

*op*および*imm2*を指定することで、下記演算命令の実行結果を最大3ビットシフトします。シフトの方向、種類は*op*によりsra、srl、sllを指定できます。シフト動作は、sra、srl、sllの各命令と同じです。ただし、シフトアウトレジスタSORを使用しませんのでSORは変化しません。また、C、V、Z、Nフラグは拡張対象命令の演算結果によってのみ変化し、シフト動作の影響は受けません。

ext命令に対する例外（リセット、デバッグブレークを除く）はハードウェアによりマスクされ、拡張対象命令実行時に例外処理が受け付けられます。ただし、この場合の例外処理からのリターンアドレスはext命令の先頭になります。

***op,imm2*が有効に機能する命令**

```
sub  %rd,%rs    sub  %rd,imm6   sub  %sp,imm10  sub  %rd,%rs
add  %rd,%rs    add  %rd,imm6   add  %sp,imm10  adc  %rd,%rs
add  %rd,%dp
```

例

```
(1)ext  sll,2
   add  %r1,%r2       ; r1(31:2)← (r1 + r2) << 2, r1(1:0)← 00

(2)ext  sra,1
   ext  0x0123
   add  %r3,%r4       ; r3(30:0)← (r4 + 0x0123) >> 1, r3(31)← r3(31)
```

ext *cond*

機　能

条件付き実行

標準）条件成立時、次の命令を実行しない
拡張1）不可
拡張2）不可
拡張3）不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	1	1	*cond*				0	0	0	0

0x3B_0

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

cond　psr[3:0]

CLK

0または1サイクル（プリデコード時は0サイクル）

説　明

PSRのステータスフラグの条件が*cond*に一致した場合、直後に記述された命令を実行しません。
*cond*とフラグの条件は下記のとおりです。

cond	フラグ条件
gt	!Z & !(N ^ V)
ge	!(N ^ V)
lt	N ^ V
le	Z | (N ^ V)
ugt	!Z & !C
uge	!C
ult	C
ule	Z | C
eq	Z
ne	!Z

ext命令に対する例外（リセット、デバッグブレークを除く）はハードウェアによりマスクされ、拡張対象命令実行時に例外処理が受け付けられます。ただし、この場合の例外処理からのリターンアドレスはext命令の先頭になります。

例

```
cmp  %r2,%r3
ext  ult        ; PSR(C) = 1の場合、次の命令を実行しない
add  %r1,%r2    ; r1 ← r1 + r2
```

注　意

この命令の直後に下記の分岐命令を置かないでください。無効な命令となります。

```
jrgt   sign8    jrgt.d  sign8   jrge   sign8    jrge.d  sign8
jrlt   sign8    jrlt.d  sign8   jrle   sign8    jrle.d  sign8
jrugt  sign8    jrugt.d sign8   jruge  sign8    jruge.d sign8
jrult  sign8    jrult.d sign8   jrule  sign8    jrule.d sign8
jreq   sign8    jreq.d  sign8   jrne   sign8    jrne.d  sign8
jp     sign8    jp.d    sign8   jp     %rb      jp.d    %rb
jpr    %rb      jpr.d   %rb
call   sign8    call    %rb     call.d sign8    call.d  %rb
ret             ret.d           reti             retd
retm
int    imm2     brk
slp              halt
loop             repeat
```

halt

機　能　**HALT**

標準）CPUをHALTモードに設定
拡張1）不可
拡張2）不可
拡張3）不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0

0x0080

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード　–

ＣＬＫ　1サイクル

説　明　CPUをHALTモードにします。
これにより、CPUは動作を停止し、消費電流を抑えることができます。
チップ上の周辺回路はHALTモードでも動作します。
HALTモードは割り込みによって解除され、その割り込み処理ルーチンを実行後、halt命令の次の命令位置に戻ります。

例

```
halt        ; CPUをHALTモードに設定
```

int *imm2*

機　能

ソフトウェア例外

標準）ssp ← ssp - 4, W[ssp] ← pc + 2, ssp ← ssp - 4, W[ssp] ← psr,
pc ← ソフトウェア例外ベクタ
拡張1）不可
拡張2）不可
拡張3）不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	0	0	1	0	0	1	0	0	0	0	0	*imm2*		0x048_

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
0	0	–	–	–	–	–	–	–	–	–	–	–

モード　即値（符号なし）

CLK　7サイクル

説　明

ソフトウェア例外を発生させます。
次の命令のアドレスとPSRをスタックにセーブ後、トラップテーブルからソフトウェア例外ベクタを読み出してPCにロードします。これによりソフトウェア例外処理ルーチンに分岐します。
S1C33 ADVは4種類のソフトウェア例外に対応しており、オペランドの2ビット即値*imm2*によってその番号（0～3）を指定します。この番号によりトラップテーブル内のベクタアドレスが決定します。

	imm2	ベクタアドレス
ソフトウェア例外0:	0	Base + 48
ソフトウェア例外1:	1	Base + 52
ソフトウェア例外2:	2	Base + 56
ソフトウェア例外3:	3	Base + 60

BaseはTTBRレジスタに設定されているトラップテーブルの先頭アドレスです（デフォルト: 0x20000000）。
処理ルーチンよりのリターンには`reti`命令を使用します。

例

```
int  2     ; ソフトウェア例外2の処理ルーチンを実行
```

注　意

`int`命令実行時は、CPUの動作モードにかかわらず、常にSSPがスタックポインタとして使用されます。

jp *%rb* / jp.d *%rb*

機　能

無条件ジャンプ

標準）pc ← *rb*

拡張1）不可

拡張2）不可

拡張3）不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	d	1	0	0	0	*r*	*b*		

0x068_, 0x078_

d（ビット8）= 0の場合　`jp`　`%rb`

d（ビット8）= 1の場合　`jp.d` `%rb`

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
−	−	−	−	−	−	−	−	−	−	−	−	−

モード

レジスタ直接 `%rb` = `%r0`～`%r15`

CLK

`jp`　　3サイクル

`jp.d`　　2サイクル

説　明

(1)標準

`jp`　`%rb`

*rb*レジスタの内容をPCにロードして、そのアドレスに分岐します。*rb*レジスタの最下位ビットは無視され、常に0として扱われます。

(2)ディレイド分岐（dビット = 1）

`jp.d`　`%rb`

`jp.d`命令では次の命令がディレイド命令となります。ディレイド命令は分岐前に実行されます。`jp.d`命令と次のディレイド命令の間はトラップがマスクされ、割り込みや例外は発生しません。

例

```
jp  %r0          ; R0レジスタが示すアドレスにジャンプ
```

注　意

`jp.d`命令（ディレイド分岐）を使用する場合、次の命令はディレイド命令として使用可能な命令に限られます。それ以外の命令を実行した場合、動作は不定となりますので注意してください。使用可能な命令については、Appendixの命令一覧表を参照してください。

jp *sign8* / jp.d *sign8*

機　能

無条件PC相対ジャンプ

標準）pc ← pc + *sign8* × 2
拡張1）pc ← pc + *sign22*
拡張2）pc ← pc + *sign32*
拡張3）不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	d	*sign8*							

0x1E__, 0x1F__

d(ビット8)= 0の場合　`jp  sign8`
d(ビット8)= 1の場合　`jp.d sign8`

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

符号付きPC相対

CLK

`jp`　　2サイクル
`jp.d`　　1サイクル

説　明

(1)標準

```
jp  sign8    ; = "jp sign9", sign8 = sign9(8:1), sign9(0)=0
```

符号付き8ビット即値*sign8*を2倍してPCに加算し、そのアドレスに分岐します。*sign8*は16ビット単位のハーフワードアドレスを指定します。
sign8(×2)による分岐可能範囲はPC - 0x100～PC + 0xFEです。

(2)拡張1

```
ext  imm13   ; = sign22(21:9)
jp   sign8   ; = "jp sign22", sign8 = sign22(8:1), sign22(0)=0
```

`ext`命令の13ビット即値*imm13*が符号拡張され、PCに加算するディスプレースメントが符号付き22ビットとなります。
*sign22*による分岐可能範囲はPC - 0x200000～PC + 0x1FFFFEです。

(3)拡張2

```
ext  imm13   ; imm13(12:3)= sign32(31:22)
ext  imm13   ; = sign32(21:9)
jp   sign8   ; = "jp sign32", sign8 = sign32(8:1), sign32(0)=0
```

`ext`命令の2つの13ビット即値(*imm13*×2)が符号拡張され、PCに加算するディスプレースメントが符号付き32ビットとなります。これにより、全アドレス空間をカバーします。最初の*imm13*の下位3ビットは無視されますので注意が必要です。

(4)ディレイド分岐(dビット = 1)

```
jp.d  sign8
```

`jp.d`命令では次の命令がディレイド命令となります。ディレイド命令は分岐前に実行されます。`jp.d`命令と次のディレイド命令の間はトラップがマスクされ、割り込みや例外は発生しません。

例

```
ext  0x8
ext  0x0
jp   0x80 ; PC + 0x400100番地へジャンプ
```

注　意

`jp.d`命令(ディレイド分岐)を使用する場合、次の命令はディレイド命令として使用可能な命令に限られます。それ以外の命令を実行した場合、動作は不定となりますので注意してください。使用可能な命令については、Appendixの命令一覧表を参照してください。

jpr *%rb* / jpr.d *%rb*

機　能

無条件PC相対ジャンプ

標準）pc ← pc + *rb*
拡張1）不可
拡張2）不可
拡張3）不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	d	1	1	0	0	*rb*			

0x02C_, 0x03C_

d（ビット8）= 0の場合　`jpr`　*%rb*
d（ビット8）= 1の場合　`jpr.d`　*%rb*

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

レジスタ直接 *%rb* = %r0～%r15

CLK

`jpr`　　4サイクル
`jpr.d`　3サイクル

説　明

(1)標準

`jpr`　*%rb*

*rb*レジスタの内容をPCに加算して、そのアドレスに分岐します。

(2)ディレイド分岐(dビット = 1)

`jpr.d`　*%rb*

`jpr.d`命令では次の命令がディレイド命令となります。ディレイド命令は分岐前に実行されます。`jpr.d`命令と次のディレイド命令の間はトラップがマスクされ、割り込みや例外は発生しません。

例

```
jpr  %r0        ; PC ← PC + R0
```

注　意

`jpr.d`命令(ディレイド分岐)を使用する場合、次の命令はディレイド命令として使用可能な命令に限られます。それ以外の命令を実行した場合、動作は不定となりますので注意してください。使用可能な命令については、Appendixの命令一覧表を参照してください。

jreq *sign8* / jreq.d *sign8*

機　能

条件付きPC相対ジャンプ

標準）pc ← pc + *sign8* × 2 if Z is true
拡張1）pc ← pc + *sign22* if Z is true
拡張2）pc ← pc + *sign32* if Z is true
拡張3）不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	d	*sign8*							

0x18__, 0x19__

d(ビット8)= 0の場合　`jreq`　*sign8*
d(ビット8)= 1の場合　`jreq.d` *sign8*

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

符号付きPC相対

CLK

`jreq`　　1サイクル(分岐しない場合)、2サイクル(分岐する場合)
`jreq.d`　　1サイクル

説　明

(1)標準

```
jreq  sign8  ; = "jreq sign9", sign8 = sign9(8:1), sign9(0)=0
```

次の条件が成立している場合、符号付き8ビット即値*sign8*を2倍してPCに加算し、そのアドレスに分岐します。条件が不成立の場合は、分岐しません。

- Zフラグ＝1(例: `cmp A,B`の実行結果が"A＝B")

*sign8*は16ビット単位のハーフワードアドレスを指定します。*sign8*(×2)による分岐可能範囲はPC - 0x100～PC + 0xFEです。

(2)拡張1

```
ext   imm13  ; = sign22(21:9)
jreq  sign8  ; = "jreq sign22", sign8 = sign22(8:1), sign22(0)=0
```

`ext`命令の13ビット即値*imm13*が符号拡張され、PCに加算するディスプレースメントが符号付き22ビットとなります。*sign22*による分岐可能範囲はPC - 0x200000～PC + 0x1FFFFEです。

(3)拡張2

```
ext   imm13  ; imm13(12:3)= sign32(31:22)
ext   imm13  ; = sign32(21:9)
jreq  sign8  ; = "jreq sign32", sign8 = sign32(8:1), sign32(0)=0
```

`ext`命令の2つの13ビット即値(*imm13*×2)が符号拡張され、PCに加算するディスプレースメントが符号付き32ビットとなります。これにより、全アドレス空間をカバーします。最初の*imm13*の下位3ビットは無視されますので注意が必要です。

(4)ディレイド分岐(dビット = 1)

```
jreq.d  sign8
```

`jreq.d`命令では次の命令がディレイド命令となります。ディレイド命令は分岐前に実行されます。`jreq.d`命令と次のディレイド命令の間はトラップがマスクされ、割り込みや例外は発生しません。

例

```
cmp   %r0,%r1
jreq  0x2       ; r0 = r1ならば次の命令をスキップ
```

注　意

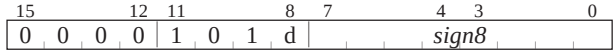
`jreq.d`命令(ディレイド分岐)を使用する場合、次の命令はディレイド命令として使用可能な命令に限られます。それ以外の命令を実行した場合、動作は不定となりますので注意してください。使用可能な命令については、Appendixの命令一覧表を参照してください。

jrge *sign8* / jrge.d *sign8*

機　能　**条件付きPC相対ジャンプ(符号付き演算結果判定)**

標準) pc ← pc + *sign8*×2 if !(N^V) is true
拡張1) pc ← pc + *sign22* if !(N^V) is true
拡張2) pc ← pc + *sign32* if !(N^V) is true
拡張3) 不可

コード



0x0A__, 0x0B__

d(ビット8)= 0の場合　`jrge　 sign8`
d(ビット8)= 1の場合　`jrge.d sign8`

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード　符号付きPC相対

CLK

`jrge`　　1サイクル(分岐しない場合)、2サイクル(分岐する場合)
`jrge.d`　　1サイクル

説　明

(1)標準

```
jrge  sign8  ; = "jrge sign9", sign8 = sign9(8:1), sign9(0)=0
```

次の条件が成立している場合、符号付き8ビット即値*sign8*を2倍してPCに加算し、そのアドレスに分岐します。条件が不成立の場合は、分岐しません。

- Nフラグ＝Vフラグ(例: `cmp A,B`の実行結果が"A≧B")

*sign8*は16ビット単位のハーフワードアドレスを指定します。*sign8*(×2)による分岐可能範囲はPC - 0x100～PC + 0xFEです。

(2)拡張1

```
ext   imm13  ; = sign22(21:9)
jrge  sign8  ; = "jrge sign22", sign8 = sign22(8:1), sign22(0)=0
```

`ext`命令の13ビット即値*imm13*が符号拡張され、PCに加算するディスプレースメントが符号付き22ビットとなります。*sign22*による分岐可能範囲はPC - 0x200000～PC + 0x1FFFFEです。

(3)拡張2

```
ext   imm13  ; imm13(12:3)= sign32(31:22)
ext   imm13  ; = sign32(21:9)
jrge  sign8  ; = "jrge sign32", sign8 = sign32(8:1), sign32(0)=0
```

`ext`命令の2つの13ビット即値(*imm13*×2)が符号拡張され、PCに加算するディスプレースメントが符号付き32ビットとなります。これにより、全アドレス空間をカバーします。最初の*imm13*の下位3ビットは無視されますので注意が必要です。

(4)ディレイド分岐(dビット = 1)

```
jrge.d  sign8
```

`jrge.d`命令では次の命令がディレイド命令となります。ディレイド命令は分岐前に実行されます。`jrge.d`命令と次のディレイド命令の間はトラップがマスクされ、割り込みや例外は発生しません。

例

```
cmp   %r0,%r1   ; r0、r1に符号付きデータがロードされている場合
jrge  0x2       ; r0≧r1ならば次の命令をスキップ
```

注　意　`jrge.d`命令(ディレイド分岐)を使用する場合、次の命令はディレイド命令として使用可能な命令に限られます。それ以外の命令を実行した場合、動作は不定となりますので注意してください。使用可能な命令については、Appendixの命令一覧表を参照してください。

jrgt *sign8* / jrgt.d *sign8*

機　能

条件付きPC相対ジャンプ(符号付き演算結果判定)

標準) pc ← pc + *sign8* × 2 if !Z&!(N^V) is true
拡張1) pc ← pc + *sign22* if !Z&!(N^V) is true
拡張2) pc ← pc + *sign32* if !Z&!(N^V) is true
拡張3) 不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	d	*sign8*							

0x08__, 0x09__

d(ビット8)= 0の場合　`jrgt`　*sign8*
d(ビット8)= 1の場合　`jrgt.d` *sign8*

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

符号付きPC相対

CLK

`jrgt`　　1サイクル(分岐しない場合)、2サイクル(分岐する場合)
`jrgt.d`　　1サイクル

説　明

(1)標準

```
jrgt  sign8  ; = "jrgt sign9", sign8 = sign9(8:1), sign9(0)=0
```

次の条件が成立している場合、符号付き8ビット即値*sign8*を2倍してPCに加算し、そのアドレスに分岐します。条件が不成立の場合は、分岐しません。

- Zフラグ＝0 かつ Nフラグ＝Vフラグ(例: `cmp A,B`の実行結果が“A＞B”)

*sign8*は16ビット単位のハーフワードアドレスを指定します。*sign8*(×2)による分岐可能範囲はPC - 0x100～PC + 0xFEです。

(2)拡張1

```
ext   imm13  ; = sign22(21:9)
jrgt  sign8  ; = "jrgt sign22", sign8 = sign22(8:1), sign22(0)=0
```

`ext`命令の13ビット即値*imm13*が符号拡張され、PCに加算するディスプレースメントが符号付き22ビットとなります。*sign22*による分岐可能範囲はPC - 0x200000～PC + 0x1FFFFEです。

(3)拡張2

```
ext   imm13  ; imm13(12:3)= sign32(31:22)
ext   imm13  ; = sign32(21:9)
jrgt  sign8  ; = "jrgt sign32", sign8 = sign32(8:1), sign32(0)=0
```

`ext`命令の2つの13ビット即値(*imm13*×2)が符号拡張され、PCに加算するディスプレースメントが符号付き32ビットとなります。これにより、全アドレス空間をカバーします。最初の*imm13*の下位3ビットは無視されますので注意が必要です。

(4)ディレイド分岐(dビット = 1)

```
jrgt.d  sign8
```

`jrgt.d`命令では次の命令がディレイド命令となります。ディレイド命令は分岐前に実行されます。`jrgt.d`命令と次のディレイド命令の間はトラップがマスクされ、割り込みや例外は発生しません。

例

```
cmp   %r0,%r1   ; r0、r1に符号付きデータがロードされている場合
jrgt  0x2       ; r0＞r1ならば次の命令をスキップ
```

注　意

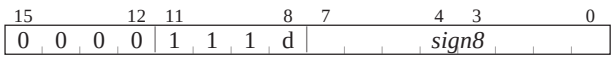
`jrgt.d`命令(ディレイド分岐)を使用する場合、次の命令はディレイド命令として使用可能な命令に限られます。それ以外の命令を実行した場合、動作は不定となりますので注意してください。使用可能な命令については、Appendixの命令一覧表を参照してください。

jrle *sign8* / jrle.d *sign8*

機　能　**条件付きPC相対ジャンプ(符号付き演算結果判定)**

標準) pc ← pc + *sign8*×2 if Z | (N^V) is true
拡張1) pc ← pc + *sign22* if Z | (N^V) is true
拡張2) pc ← pc + *sign32* if Z | (N^V) is true
拡張3) 不可

コード



0x0E__, 0x0F__

d(ビット8)= 0の場合　`jrle`　*sign8*
d(ビット8)= 1の場合　`jrle.d` *sign8*

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード　符号付きPC相対

CLK

`jrle`	1サイクル(分岐しない場合)、2サイクル(分岐する場合)
`jrle.d`	1サイクル

説　明

(1)標準

```
jrle  sign8  ; = "jrle sign9", sign8 = sign9(8:1), sign9(0)=0
```

次の条件が成立している場合、符号付き8ビット即値*sign8*を2倍してPCに加算し、そのアドレスに分岐します。条件が不成立の場合は、分岐しません。

- Zフラグ＝1 または Nフラグ≠Vフラグ(例: `cmp A,B`の実行結果が"A≦B")

*sign8*は16ビット単位のハーフワードアドレスを指定します。*sign8*(×2)による分岐可能範囲はPC - 0x100～PC + 0xFEです。

(2)拡張1

```
ext   imm13  ; = sign22(21:9)
jrle  sign8  ; = "jrle sign22", sign8 = sign22(8:1), sign22(0)=0
```

`ext`命令の13ビット即値*imm13*が符号拡張され、PCに加算するディスプレースメントが符号付き22ビットとなります。*sign22*による分岐可能範囲はPC - 0x200000～PC + 0x1FFFFEです。

(3)拡張2

```
ext   imm13  ; imm13(12:3)= sign32(31:22)
ext   imm13  ; = sign32(21:9)
jrle  sign8  ; = "jrle sign32", sign8 = sign32(8:1), sign32(0)=0
```

`ext`命令の2つの13ビット即値(*imm13*×2)が符号拡張され、PCに加算するディスプレースメントが符号付き32ビットとなります。これにより、全アドレス空間をカバーします。最初の*imm13*の下位3ビットは無視されますので注意が必要です。

(4)ディレイド分岐(dビット = 1)

```
jrle.d  sign8
```

`jrle.d`命令では次の命令がディレイド命令となります。ディレイド命令は分岐前に実行されます。`jrle.d`命令と次のディレイド命令の間はトラップがマスクされ、割り込みや例外は発生しません。

例

```
cmp   %r0,%r1   ; r0、r1に符号付きデータがロードされている場合
jrle  0x2       ; r0≦r1ならば次の命令をスキップ
```

注　意　`jrle.d`命令(ディレイド分岐)を使用する場合、次の命令はディレイド命令として使用可能な命令に限られます。それ以外の命令を実行した場合、動作は不定となりますので注意してください。使用可能な命令については、Appendixの命令一覧表を参照してください。

jrlt *sign8* / jrlt.d *sign8*

機　能

条件付きPC相対ジャンプ(符号付き演算結果判定)

標準) pc ← pc + *sign8* × 2 if N^V is true
拡張1) pc ← pc + *sign22* if N^V is true
拡張2) pc ← pc + *sign32* if N^V is true
拡張3) 不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	d	*sign8*							

0x0C__, 0x0D__

d(ビット8)= 0の場合　`jrlt`　*sign8*
d(ビット8)= 1の場合　`jrlt.d` *sign8*

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

符号付きPC相対

CLK

`jrlt`　　1サイクル(分岐しない場合)、2サイクル(分岐する場合)
`jrlt.d`　　1サイクル

説　明

(1)標準

```
jrlt  sign8  ; = "jrlt sign9", sign8 = sign9(8:1), sign9(0)=0
```

次の条件が成立している場合、符号付き8ビット即値*sign8*を2倍してPCに加算し、そのアドレスに分岐します。条件が不成立の場合は、分岐しません。

- Nフラグ≠Vフラグ(例: `cmp A,B`の実行結果が"A＜B")

*sign8*は16ビット単位のハーフワードアドレスを指定します。*sign8*(×2)による分岐可能範囲はPC - 0x100～PC + 0xFEです。

(2)拡張1

```
ext   imm13  ; = sign22(21:9)
jrlt  sign8  ; = "jrlt sign22", sign8 = sign22(8:1), sign22(0)=0
```

`ext`命令の13ビット即値*imm13*が符号拡張され、PCに加算するディスプレースメントが符号付き22ビットとなります。*sign22*による分岐可能範囲はPC - 0x200000～PC + 0x1FFFFEです。

(3)拡張2

```
ext   imm13  ; imm13(12:3)= sign32(31:22)
ext   imm13  ; = sign32(21:9)
jrlt  sign8  ; = "jrlt sign32", sign8 = sign32(8:1), sign32(0)=0
```

`ext`命令の2つの13ビット即値(*imm13*×2)が符号拡張され、PCに加算するディスプレースメントが符号付き32ビットとなります。これにより、全アドレス空間をカバーします。最初の*imm13*の下位3ビットは無視されますので注意が必要です。

(4)ディレイド分岐(dビット = 1)

```
jrlt.d  sign8
```

`jrlt.d`命令では次の命令がディレイド命令となります。ディレイド命令は分岐前に実行されます。`jrlt.d`命令と次のディレイド命令の間はトラップがマスクされ、割り込みや例外は発生しません。

例

```
cmp   %r0,%r1   ; r0、r1に符号付きデータがロードされている場合
jrlt  0x2       ; r0＜r1ならば次の命令をスキップ
```

注　意

`jrlt.d`命令(ディレイド分岐)を使用する場合、次の命令はディレイド命令として使用可能な命令に限られます。それ以外の命令を実行した場合、動作は不定となりますので注意してください。使用可能な命令については、Appendixの命令一覧表を参照してください。

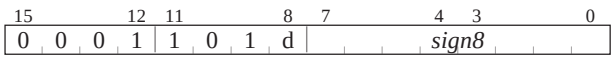
jrne *sign8* / jrne.d *sign8*

機　能

条件付きPC相対ジャンプ

標準） pc ← pc + *sign8*×2 if !Z is true
拡張1) pc ← pc + *sign22* if !Z is true
拡張2) pc ← pc + *sign32* if !Z is true
拡張3) 不可

コード



0x1A__, 0x1B__

d(ビット8)= 0の場合　`jrne  sign8`
d(ビット8)= 1の場合　`jrne.d sign8`

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

符号付きPC相対

CLK

`jrne`　　1サイクル(分岐しない場合)、2サイクル(分岐する場合)
`jrne.d`　　1サイクル

説　明

(1)標準

```
jrne  sign8  ; = "jrne sign9", sign8 = sign9(8:1), sign9(0)=0
```

次の条件が成立している場合、符号付き8ビット即値*sign8*を2倍してPCに加算し、そのアドレスに分岐します。条件が不成立の場合は、分岐しません。

- Zフラグ＝0(例: `cmp A,B`の実行結果が“A≠B”)

*sign8*は16ビット単位のハーフワードアドレスを指定します。*sign8*(×2)による分岐可能範囲はPC - 0x100～PC + 0xFEです。

(2)拡張1

```
ext   imm13  ; = sign22(21:9)
jrne  sign8  ; = "jrne sign22", sign8 = sign22(8:1), sign22(0)=0
```

`ext`命令の13ビット即値*imm13*が符号拡張され、PCに加算するディスプレースメントが符号付き22ビットとなります。*sign22*による分岐可能範囲はPC - 0x200000～PC + 0x1FFFFEです。

(3)拡張2

```
ext   imm13  ; imm13(12:3)= sign32(31:22)
ext   imm13  ; = sign32(21:9)
jrne  sign8  ; = "jrne sign32", sign8 = sign32(8:1), sign32(0)=0
```

`ext`命令の2つの13ビット即値(*imm13*×2)が符号拡張され、PCに加算するディスプレースメントが符号付き32ビットとなります。これにより、全アドレス空間をカバーします。最初の*imm13*の下位3ビットは無視されますので注意が必要です。

(4)ディレイド分岐(dビット = 1)

```
jrne.d  sign8
```

`jrne.d`命令では次の命令がディレイド命令となります。ディレイド命令は分岐前に実行されます。`jrne.d`命令と次のディレイド命令の間はトラップがマスクされ、割り込みや例外は発生しません。

例

```
cmp   %r0,%r1
jrne  0x2       ; r0≠r1ならば次の命令をスキップ
```

注　意

`jrne.d`命令(ディレイド分岐)を使用する場合、次の命令はディレイド命令として使用可能な命令に限られます。それ以外の命令を実行した場合、動作は不定となりますので注意してください。使用可能な命令については、Appendixの命令一覧表を参照してください。

jruge *sign8* / jruge.d *sign8*

機　能

条件付きPC相対ジャンプ(符号なし演算結果判定)

標準) pc ← pc + *sign8* × 2 if !C is true
拡張1) pc ← pc + *sign22* if !C is true
拡張2) pc ← pc + *sign32* if !C is true
拡張3) 不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	1	d	*sign8*							

0x12__, 0x13__

d(ビット8)= 0の場合　`jruge`　*sign8*
d(ビット8)= 1の場合　`jruge.d` *sign8*

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

符号付きPC相対

CLK

`jruge`　　1サイクル(分岐しない場合)、2サイクル(分岐する場合)
`jruge.d`　1サイクル

説　明

(1)標準

```
jruge  sign8 ; = "jruge sign9", sign8 = sign9(8:1), sign9(0)=0
```

次の条件が成立している場合、符号付き8ビット即値*sign8*を2倍してPCに加算し、そのアドレスに分岐します。条件が不成立の場合は、分岐しません。

- Cフラグ＝0(例: `cmp A,B`の実行結果が"A≧B")

*sign8*は16ビット単位のハーフワードアドレスを指定します。*sign8*(×2)による分岐可能範囲はPC - 0x100～PC + 0xFEです。

(2)拡張1

```
ext    imm13 ; = sign22(21:9)
jruge  sign8 ; = "jruge sign22", sign8 = sign22(8:1), sign22(0)=0
```

`ext`命令の13ビット即値*imm13*が符号拡張され、PCに加算するディスプレースメントが符号付き22ビットとなります。*sign22*による分岐可能範囲はPC - 0x200000～PC + 0x1FFFFEです。

(3)拡張2

```
ext    imm13 ; imm13(12:3)= sign32(31:22)
ext    imm13 ; = sign32(21:9)
jruge  sign8 ; = "jruge sign32", sign8 = sign32(8:1), sign32(0)=0
```

`ext`命令の2つの13ビット即値(*imm13*×2)が符号拡張され、PCに加算するディスプレースメントが符号付き32ビットとなります。これにより、全アドレス空間をカバーします。最初の*imm13*の下位3ビットは無視されますので注意が必要です。

(4)ディレイド分岐(dビット = 1)

```
jruge.d  sign8
```

`jruge.d`命令では次の命令がディレイド命令となります。ディレイド命令は分岐前に実行されます。`jruge.d`命令と次のディレイド命令の間はトラップがマスクされ、割り込みや例外は発生しません。

例

```
cmp    %r0,%r1  ; r0、r1に符号なしデータがロードされている場合
jruge  0x2      ; r0≧r1ならば次の命令をスキップ
```

注　意

`jruge.d`命令(ディレイド分岐)を使用する場合、次の命令はディレイド命令として使用可能な命令に限られます。それ以外の命令を実行した場合、動作は不定となりますので注意してください。使用可能な命令については、Appendixの命令一覧表を参照してください。

jrugt *sign8* / jrugt.d *sign8*

機　能

条件付きPC相対ジャンプ(符号なし演算結果判定)

標準) pc ← pc + *sign8*×2 if !Z&!C is true
拡張1) pc ← pc + *sign22* if !Z&!C is true
拡張2) pc ← pc + *sign32* if !Z&!C is true
拡張3) 不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	0	d	*sign8*							

0x10__, 0x11__

d(ビット8)= 0の場合　`jrugt`　*sign8*
d(ビット8)= 1の場合　`jrugt.d` *sign8*

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
−	−	−	−	−	−	−	−	−	−	−	−	−

モード

符号付きPC相対

CLK

`jrugt`　　1サイクル(分岐しない場合)、2サイクル(分岐する場合)
`jrugt.d`　1サイクル

説　明

(1)標準

`jrugt` *sign8* `; = "jrugt` *sign9*`",` *sign8* `=` *sign9*`(8:1),` *sign9*`(0)=0`

次の条件が成立している場合、符号付き8ビット即値*sign8*を2倍してPCに加算し、そのアドレスに分岐します。条件が不成立の場合は、分岐しません。

- Zフラグ＝0 かつ Cフラグ＝0(例: `cmp A,B`の実行結果が“A＞B”)

*sign8*は16ビット単位のハーフワードアドレスを指定します。*sign8*(×2)による分岐可能範囲はPC - 0x100～PC + 0xFEです。

(2)拡張1

```
ext    imm13 ; = sign22(21:9)
jrugt  sign8 ; = "jrugt sign22", sign8 = sign22(8:1), sign22(0)=0
```

`ext`命令の13ビット即値*imm13*が符号拡張され、PCに加算するディスプレースメントが符号付き22ビットとなります。*sign22*による分岐可能範囲はPC - 0x200000～PC + 0x1FFFFEです。

(3)拡張2

```
ext    imm13 ; imm13(12:3)= sign32(31:22)
ext    imm13 ; = sign32(21:9)
jrugt  sign8 ; = "jrugt sign32", sign8 = sign32(8:1), sign32(0)=0
```

`ext`命令の2つの13ビット即値(*imm13*×2)が符号拡張され、PCに加算するディスプレースメントが符号付き32ビットとなります。これにより、全アドレス空間をカバーします。最初の*imm13*の下位3ビットは無視されますので注意が必要です。

(4)ディレイド分岐(dビット = 1)

```
jrugt.d  sign8
```

`jrugt.d`命令では次の命令がディレイド命令となります。ディレイド命令は分岐前に実行されます。`jrugt.d`命令と次のディレイド命令の間はトラップがマスクされ、割り込みや例外は発生しません。

例

```
cmp    %r0,%r1  ; r0、r1に符号なしデータがロードされている場合
jrugt  0x2      ; r0＞r1ならば次の命令をスキップ
```

注　意

`jrugt.d`命令(ディレイド分岐)を使用する場合、次の命令はディレイド命令として使用可能な命令に限られます。それ以外の命令を実行した場合、動作は不定となりますので注意してください。使用可能な命令については、Appendixの命令一覧表を参照してください。

jrule *sign8* / jrule.d *sign8*

機　能

条件付きPC相対ジャンプ(符号なし演算結果判定)

標準) pc ← pc + *sign8* × 2 if Z | C is true
拡張1) pc ← pc + *sign22* if Z | C is true
拡張2) pc ← pc + *sign32* if Z | C is true
拡張3) 不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	1	d	*sign8*							

0x16__, 0x17__

d(ビット8)= 0の場合　`jrule`　*sign8*
d(ビット8)= 1の場合　`jrule.d` *sign8*

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

符号付きPC相対

CLK

`jrule`　　1サイクル(分岐しない場合)、2サイクル(分岐する場合)
`jrule.d`　1サイクル

説　明

(1)標準

```
jrule  sign8 ; = "jrule sign9", sign8 = sign9(8:1), sign9(0)=0
```

次の条件が成立している場合、符号付き8ビット即値*sign8*を2倍してPCに加算し、そのアドレスに分岐します。条件が不成立の場合は、分岐しません。

- Zフラグ＝1 または Cフラグ＝1(例: `cmp A,B`の実行結果が"A≦B")

*sign8*は16ビット単位のハーフワードアドレスを指定します。*sign8*(×2)による分岐可能範囲はPC - 0x100～PC + 0xFEです。

(2)拡張1

```
ext    imm13 ; = sign22(21:9)
jrule  sign8 ; = "jrule sign22", sign8 = sign22(8:1), sign22(0)=0
```

`ext`命令の13ビット即値*imm13*が符号拡張され、PCに加算するディスプレースメントが符号付き22ビットとなります。*sign22*による分岐可能範囲はPC - 0x200000～PC + 0x1FFFFEです。

(3)拡張2

```
ext    imm13 ; imm13(12:3)= sign32(31:22)
ext    imm13 ; = sign32(21:9)
jrule  sign8 ; = "jrule sign32", sign8 = sign32(8:1), sign32(0)=0
```

`ext`命令の2つの13ビット即値(*imm13*×2)が符号拡張され、PCに加算するディスプレースメントが符号付き32ビットとなります。これにより、全アドレス空間をカバーします。最初の*imm13*の下位3ビットは無視されますので注意が必要です。

(4)ディレイド分岐(dビット = 1)

```
jrule.d  sign8
```

`jrule.d`命令では次の命令がディレイド命令となります。ディレイド命令は分岐前に実行されます。`jrule.d`命令と次のディレイド命令の間はトラップがマスクされ、割り込みや例外は発生しません。

例

```
cmp    %r0,%r1  ; r0、r1に符号なしデータがロードされている場合
jrule  0x2      ; r0≦r1ならば次の命令をスキップ
```

注　意

`jrule.d`命令(ディレイド分岐)を使用する場合、次の命令はディレイド命令として使用可能な命令に限られます。それ以外の命令を実行した場合、動作は不定となりますので注意してください。使用可能な命令については、Appendixの命令一覧表を参照してください。

jrult *sign8* / jrult.d *sign8*

機　能　**条件付きPC相対ジャンプ(符号なし演算結果判定)**

標準) pc ← pc + *sign8*×2 if C is true
拡張1) pc ← pc + *sign22* if C is true
拡張2) pc ← pc + *sign32* if C is true
拡張3) 不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	0	d	*sign8*							

0x14__, 0x15__

d(ビット8)= 0の場合　`jrult`　*sign8*
d(ビット8)= 1の場合　`jrult.d` *sign8*

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード　符号付きPC相対

CLK

`jrult`　　1サイクル(分岐しない場合)、2サイクル(分岐する場合)
`jrult.d`　1サイクル

説　明

(1)標準

```
jrult  sign8 ; = "jrult sign9", sign8 = sign9(8:1), sign9(0)=0
```

次の条件が成立している場合、符号付き8ビット即値*sign8*を2倍してPCに加算し、そのアドレスに分岐します。条件が不成立の場合は、分岐しません。

- Cフラグ＝1(例: `cmp A,B`の実行結果が“A＜B”)

*sign8*は16ビット単位のハーフワードアドレスを指定します。*sign8*(×2)による分岐可能範囲はPC - 0x100～PC + 0xFEです。

(2)拡張1

```
ext    imm13 ; = sign22(21:9)
jrult  sign8 ; = "jrult sign22", sign8 = sign22(8:1), sign22(0)=0
```

`ext`命令の13ビット即値*imm13*が符号拡張され、PCに加算するディスプレースメントが符号付き22ビットとなります。*sign22*による分岐可能範囲はPC - 0x200000～PC + 0x1FFFFEです。

(3)拡張2

```
ext    imm13 ; imm13(12:3)= sign32(31:22)
ext    imm13 ; = sign32(21:9)
jrult  sign8 ; = "jrult sign32", sign8 = sign32(8:1), sign32(0)=0
```

`ext`命令の2つの13ビット即値(*imm13*×2)が符号拡張され、PCに加算するディスプレースメントが符号付き32ビットとなります。これにより、全アドレス空間をカバーします。最初の*imm13*の下位3ビットは無視されますので注意が必要です。

(4)ディレイド分岐(dビット = 1)

```
jrult.d  sign8
```

`jrult.d`命令では次の命令がディレイド命令となります。ディレイド命令は分岐前に実行されます。`jrult.d`命令と次のディレイド命令の間はトラップがマスクされ、割り込みや例外は発生しません。

例

```
cmp    %r0,%r1  ; r0、r1に符号なしデータがロードされている場合
jrult  0x2      ; r0＜r1ならば次の命令をスキップ
```

注　意　`jrult.d`命令(ディレイド分岐)を使用する場合、次の命令はディレイド命令として使用可能な命令に限られます。それ以外の命令を実行した場合、動作は不定となりますので注意してください。使用可能な命令については、Appendixの命令一覧表を参照してください。

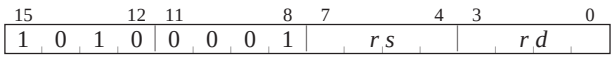
ld.b *%rd, %rs*

機　能

符号付きバイトデータ転送

標準） *rd*(7:0) ← *rs*(7:0), *rd*(31:8) ← *rs*(7)
拡張1) 不可
拡張2) 不可
拡張3) 不可

コード



0xA1__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

Src　レジスタ直接 *%rs* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK

1サイクル

説　明

(1)標準
*rs*レジスタの下位8ビット(バイトデータ)を32ビットに符号拡張して*rd*レジスタに転送します。

(2)ディレイド命令
本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。

例

```
ld.b  %r0,%r1   ; r0 ← r1(7:0)を符号拡張
```

ld.b *%rd*, [*%rb*]

機　能

符号付きバイトデータ転送

標準） *rd*(7:0) ← B[*rb*], *rd*(31:8) ← B[*rb*](7)
拡張1) *rd*(7:0) ← B[*rb* + *imm13*], *rd*(31:8) ← B[*rb* + *imm13*](7)
拡張2) *rd*(7:0) ← B[*rb* + *imm26*], *rd*(31:8) ← B[*rb* + *imm26*](7)
拡張3) *rd*(7:0) ← B[*rb1* + *rb2*], *rd*(31:8) ← B[*rb1* + *rb2*](7)

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	0	0	0	*rb*				*rd*			

0x20__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

Src　レジスタ間接 *%rb* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK

1サイクル

説　明

(1)標準

```
ld.b  %rd,[%rb]   ; memory address = rb
```

指定メモリのバイトデータを32ビットに符号拡張して*rd*レジスタに転送します。*rb*レジスタの内容がアクセスされるメモリアドレスとなります。

(2)拡張1

```
ext   imm13
ld.b  %rd,[%rb]   ; memory address = rb + imm13
```

ext命令により、アドレッシングモードがディスプレースメント付きレジスタ間接アドレッシングに変わります。これにより、*rb*レジスタの内容に13ビット即値*imm13*を加えたアドレスのバイトデータを*rd*レジスタに転送します。*rb*レジスタの内容は変更されません。

(3)拡張2

```
ext   imm13       ; = imm26(25:13)
ext   imm13       ; = imm26(12:0)
ld.b  %rd,[%rb]   ; memory address = rb + imm26
```

アドレッシングモードがディスプレースメント付きレジスタ間接アドレッシングに変わり、*rb*レジスタの内容に26ビット即値*imm26*を加えたアドレスのバイトデータを*rd*レジスタに転送します。*rb*レジスタの内容は変更されません。

(4)拡張3

```
ext   %rb2
ld.b  %rd,[%rb1]  ; memory address = rb1 + rb2
```

3オペランドのレジスタ間接アドレッシングに変わり、*rb1*レジスタの内容に*rb2*レジスタを加えたアドレスのバイトデータを*rd*レジスタに転送します。*rb1*レジスタおよび*rb2*レジスタの内容は変更されません。

ld.b *%rd*, [*%rb*]+

機　能

符号付きバイトデータ転送

標準） *rd*(7:0) ← B[*rb*], *rd*(31:8) ← B[*rb*](7), *rb* ← *rb* + 1
拡張1) *rd*(7:0) ← B[*rb*], *rd*(31:8) ← B[*rb*](7), *rb* ← *rb* + *sign13*
拡張2) *rd*(7:0) ← B[*rb*], *rd*(31:8) ← B[*rb*](7), *rb* ← *rb* + *sign26*
拡張3) *rd*(7:0) ← B[*rb1*], *rd*(31:8) ← B[*rb1*](7), *rb1* ← *rb1* + *rb2*

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	0	0	1	*rb*				*rd*			

0x21__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

Src　ポストインクリメント付きレジスタ間接 *%rb* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK

1サイクル

説　明

(1)標準

```
ld.b  %rd,[%rb]+  ; memory address = rb
```

指定メモリのバイトデータを32ビットに符号拡張して*rd*レジスタに転送します。*rb*レジスタの内容がアクセスされるメモリアドレスとなります。データ転送後、*rb*レジスタ内のアドレスをインクリメント(+1)します。

(2)拡張1

```
ext   imm13       ; = sign13
ld.b  %rd,[%rb]+  ; memory address = rb
```

指定メモリのバイトデータを32ビットに符号拡張して*rd*レジスタに転送します。*rb*レジスタの内容がアクセスされるメモリアドレスとなります。データ転送後、*rb*レジスタ内のアドレスに*sign13*を加え*rb*に格納します。*sign13*の範囲は、-4,096～+4,095です。

(3)拡張2

```
ext   imm13       ; = sign26(25:13)
ext   imm13       ; = sign26(12:0)
ld.b  %rd,[%rb]+  ; memory address = rb
```

指定メモリのバイトデータを32ビットに符号拡張して*rd*レジスタに転送します。*rb*レジスタの内容がアクセスされるメモリアドレスとなります。データ転送後、*rb*レジスタ内のアドレスに*sign26*を加え*rb*に格納します。*sign26*の範囲は、-33,554,432～+33,554,431です。

(4)拡張3

```
ext   %rb2
ld.b  %rd,[%rb1]+ ; memory address = rb1, rb1 ← rb1 + rb2
```

指定メモリのバイトデータを32ビットに符号拡張して*rd*レジスタに転送します。*rb1*レジスタの内容がアクセスされるメモリアドレスとなります。データ転送後、*rb1*レジスタ内のアドレスに*rb2*レジスタを加え*rb1*に格納します。*rb2*の範囲は、-2,147,483,648～+2,147,483,647です。

ld.b *%rd*, [%dp + *imm6*]

機　能　**符号付きバイトデータ転送**

標準） *rd*(7:0) ← B[dp + *imm6*], *rd*(31:8) ← B[dp + *imm6*](7)
拡張1) *rd*(7:0) ← B[dp + *imm19*], *rd*(31:8) ← B[dp + *imm19*](7)
拡張2) *rd*(7:0) ← B[dp + *imm32*], *rd*(31:8) ← B[dp + *imm32*](7)
拡張3)不可

コード

15	14	13	12	11	10	9 … 4	3 … 0
1	1	1	0	0	0	*imm6*	*rd*

0xE0__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード　Src　ディスプレースメント付きレジスタ間接
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK　1サイクル

説　明　(1)標準

```
ld.b  %rd,[%dp + imm6] ; memory address = dp + imm6
```

指定メモリのバイトデータを32ビットに符号拡張して*rd*レジスタに転送します。現在のDPの内容に6ビット即値*imm6*をディスプレースメントとして加算した値がアクセスされるメモリアドレスとなります。

(2)拡張1

```
ext   imm13             ; = imm19(18:6)
ld.b  %rd,[%dp + imm6]  ; memory address = dp + imm19,
                        ; imm6 ← imm19(5:0)
```

ext命令により、ディスプレースメントが19ビットに拡張されます。これにより、DPの内容に19ビット即値*imm19*を加えたアドレスのバイトデータを*rd*レジスタに転送します。

(3)拡張2

```
ext   imm13             ; = imm32(31:19)
ext   imm13             ; = imm32(18:6)
ld.b  %rd,[%dp + imm6]  ; memory address = dp + imm32,
                        ; imm6 ← imm32(5:0)
```

2つのext命令により、ディスプレースメントが32ビットに拡張されます。これにより、DPの内容に32ビット即値*imm32*を加えたアドレスのバイトデータを*rd*レジスタに転送します。

例

```
ext   0x1
ld.b  %r0,[%dp + 0x1]     ; r0 ← [dp + 0x41]を符号拡張
```

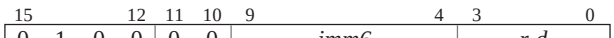
ld.b *%rd*, [%sp + *imm6*]

機　能

符号付きバイトデータ転送

標準） *rd*(7:0) ← B[sp + *imm6*], *rd*(31:8) ← B[sp + *imm6*](7)
拡張1) *rd*(7:0) ← B[sp + *imm19*], *rd*(31:8) ← B[sp + *imm19*](7)
拡張2) *rd*(7:0) ← B[sp + *imm32*], *rd*(31:8) ← B[sp + *imm32*](7)
拡張3) 不可

コード



0x40__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

Src　ディスプレースメント付きレジスタ間接
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK

1サイクル

説　明

(1)標準

```
ld.b  %rd,[%sp + imm6] ; memory address = sp + imm6
```

指定メモリのバイトデータを32ビットに符号拡張して*rd*レジスタに転送します。現在のSPの内容に6ビット即値*imm6*をディスプレースメントとして加算した値がアクセスされるメモリアドレスとなります。

(2)拡張1

```
ext   imm13             ; = imm19(18:6)
ld.b  %rd,[%sp + imm6]  ; memory address = sp + imm19,
                        ; imm6 ← imm19(5:0)
```

ext命令により、ディスプレースメントが19ビットに拡張されます。これにより、SPの内容に19ビット即値*imm19*を加えたアドレスのバイトデータを*rd*レジスタに転送します。

(3)拡張2

```
ext   imm13             ; = imm32(31:19)
ext   imm13             ; = imm32(18:6)
ld.b  %rd,[%sp + imm6]  ; memory address = sp + imm32,
                        ; imm6 ← imm32(5:0)
```

2つのext命令により、ディスプレースメントが32ビットに拡張されます。これにより、SPの内容に32ビット即値*imm32*を加えたアドレスのバイトデータを*rd*レジスタに転送します。

例

```
ext   0x1
ld.b  %r0,[%sp + 0x1]     ; r0 ← [sp + 0x41]を符号拡張
```

ld.b [*%rb*], *%rs*

機　能

符号付きバイトデータ転送

標準） B[*rb*] ← *rs*(7:0)
拡張1) B[*rb* + *imm13*] ← *rs*(7:0)
拡張2) B[*rb* + *imm26*] ← *rs*(7:0)
拡張3) B[*rb1* + *rb2*] ← *rs*(7:0)

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	0	0	*rb*				*rs*			

0x34__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

Src　レジスタ直接 *%rs* = %r0～%r15
Dst　レジスタ間接 *%rb* = %r0～%r15

CLK

1サイクル

説　明

(1)標準

```
ld.b  [%rb],%rs   ; memory address = rb
```

*rs*レジスタの下位8ビットを指定のメモリに転送します。*rb*レジスタの内容がアクセスされるメモリアドレスとなります。

(2)拡張1

```
ext   imm13
ld.b  [%rb],%rs   ; memory address = rb + imm13
```

ext命令により、アドレッシングモードがディスプレースメント付きレジスタ間接アドレッシングに変わります。*rs*レジスタの下位8ビットを、*rb*レジスタの内容に13ビット即値*imm13*を加えたアドレスに転送します。*rb*レジスタの内容は変更されません。

(3)拡張2

```
ext   imm13       ; = imm26(25:13)
ext   imm13       ; = imm26(12:0)
ld.b  [%rb],%rs   ; memory address = rb + imm26
```

アドレッシングモードがディスプレースメント付きレジスタ間接アドレッシングに変わり、*rs*レジスタの下位8ビットを、*rb*レジスタの内容に26ビット即値*imm26*を加えたアドレスに転送します。*rb*レジスタの内容は変更されません。

(4)拡張3

```
ext   %rb2
ld.b  [%rb1],%rs  ; memory address = rb1 + rb2
```

3オペランドのレジスタ間接アドレッシングに変わります。*rs*レジスタの下位8ビットを、*rb1*レジスタの内容に*rb2*レジスタを加えたアドレスに転送します。*rb1*レジスタおよび*rb2*レジスタの内容は変更されません。

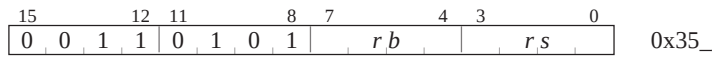
ld.b [*%rb*]+, *%rs*

機　能

符号付きバイトデータ転送

標準） B[*rb*] ← *rs*(7:0), *rb* ← *rb* + 1
拡張1) B[*rb*] ← *rs*(7:0), *rb* ← *rb* + *sign13*
拡張2) B[*rb*] ← *rs*(7:0), *rb* ← *rb* + *sign26*
拡張3) B[*rb1*] ← *rs*(7:0), *rb1* ← *rb1* + *rb2*

コード



0x35__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

Src　レジスタ直接 *%rs* = %r0～%r15
Dst　ポストインクリメント付きレジスタ間接 *%rb* = %r0～%r15

CLK

1サイクル

説　明

(1)標準

```
ld.b  [%rb]+,%rs  ; memory address = rb, rb ← rb + 1
```

*rs*レジスタの下位8ビットを指定のメモリに転送します。*rb*レジスタの内容がアクセスされるメモリアドレスとなります。データ転送後、*rb*レジスタ内のアドレスをインクリメント(+1)します。

(2)拡張1

```
ext   imm13       ; = sign13
ld.b  [%rb]+,%rs  ; memory address = rb, rb ← rb + sign13
```

*rs*レジスタの下位8ビットを指定のメモリに転送します。*rb*レジスタの内容がアクセスされるメモリアドレスとなります。データ転送後、*rb*レジスタ内のアドレスに*sign13*を加え、*rb*に格納します。*sign13*の範囲は、-4,096～+4,095です。

(3)拡張2

```
ext   imm13       ; = sign26(25:13)
ext   imm13       ; = sign26(12:0)
ld.b  [%rb]+,%rs  ; memory address = rb, rb ← rb + sign26
```

*rs*レジスタの下位8ビットを指定のメモリに転送します。*rb*レジスタの内容がアクセスされるメモリアドレスとなります。データ転送後、*rb*レジスタ内のアドレスに*sign26*を加え、*rb*に格納します。*sign26*の範囲は、-33,554,432～+33,554,431です。

(4)拡張3

```
ext   %rb2
ld.b  [%rb1]+,%rs ; memory address = rb1, rb1 = rb1 + rb2
```

3オペランドのレジスタ間接アドレッシングに変わります。*rs*レジスタの下位8ビットを*rb1*レジスタが示すアドレスに転送します。データ転送後、*rb1*レジスタの内容に*rb2*レジスタの内容を加え、*rb1*レジスタに格納します。*rb2*の範囲は、-2,147,483,648～+2,147,483,647です。

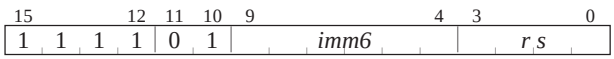
ld.b [%dp + *imm6*], *%rs*

機　能

符号付きバイトデータ転送

標準） B[dp + *imm6*] ← *rs*(7:0)
拡張1) B[dp + *imm19*] ← *rs*(7:0)
拡張2) B[dp + *imm32*] ← *rs*(7:0)
拡張3) 不可

コード



0xF4__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

Src　レジスタ直接 *%rs* = %r0～%r15
Dst　ディスプレースメント付きレジスタ間接

CLK

1サイクル

説　明

(1)標準

```
ld.b  [%dp + imm6],%rs ; memory address = dp + imm6
```

*rs*レジスタの下位8ビットを指定メモリに転送します。現在のDPの内容に6ビット即値*imm6*をディスプレースメントとして加算した値がアクセスされるメモリアドレスとなります。

(2)拡張1

```
ext   imm13             ; = imm19(18:6)
ld.b  [%dp + imm6],%rs  ; memory address = dp + imm19,
                        ; imm6 = imm19(5:0)
```

ext命令により、ディスプレースメントが19ビットに拡張されます。これにより、*rs*レジスタの下位8 ビットを、DPの内容に19ビット即値*imm19*を加えたアドレスに転送します。

(3)拡張2

```
ext   imm13             ; = imm32(31:19)
ext   imm13             ; = imm32(18:6)
ld.b  [%dp + imm6],%rs  ; memory address = dp + imm32,
                        ; imm6 = imm32(5:0)
```

2つのext命令により、ディスプレースメントが32ビットに拡張されます。これにより、*rs*レジスタの下位8ビットを、DPの内容に32ビット即値*imm32*を加えたアドレスに転送します。

例

```
ext   0x1
ld.b  [%dp + 0x1],%r0     ; B[dp + 0x41] ← r0の下位8ビット
```

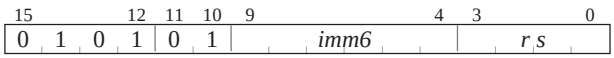
ld.b [%sp + *imm6*], *%rs*

機　能

符号付きバイトデータ転送

標準） B[sp + *imm6*] ← *rs*(7:0)
拡張1) B[sp + *imm19*] ← *rs*(7:0)
拡張2) B[sp + *imm32*] ← *rs*(7:0)
拡張3) 不可

コード



0x54__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

Src　レジスタ直接 *%rs* = %r0～%r15
Dst　ディスプレースメント付きレジスタ間接

CLK

1サイクル

説　明

(1)標準

```
ld.b  [%sp + imm6],%rs  ; memory address = sp + imm6
```

*rs*レジスタの下位8ビットを指定メモリに転送します。現在のSPの内容に6ビット即値*imm6*をディスプレースメントとして加算した値がアクセスされるメモリアドレスとなります。

(2)拡張1

```
ext   imm13             ; = imm19(18:6)
ld.b  [%sp + imm6],%rs  ; memory address = sp + imm19,
                        ; imm6 = imm19(5:0)
```

ext命令により、ディスプレースメントが19ビットに拡張されます。これにより、*rs*レジスタの下位8 ビットを、SPの内容に19ビット即値*imm19*を加えたアドレスに転送します。

(3)拡張2

```
ext   imm13             ; = imm32(31:19)
ext   imm13             ; = imm32(18:6)
ld.b  [%sp + imm6],%rs  ; memory address = sp + imm32,
                        ; imm6 = imm32(5:0)
```

2つのext命令により、ディスプレースメントが32ビットに拡張されます。これにより、*rs*レジスタの下位8ビットを、SPの内容に32ビット即値*imm32*を加えたアドレスに転送します。

例

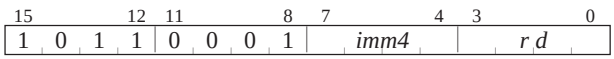
```
ext   0x1
ld.b  [%sp + 0x1],%r0     ; B[sp + 0x41] ← r0の下位8ビット
```

ld.c *%rd, imm4*

機　能　**コプロセッサからのデータ転送**

標準）*rd*(7:0) ← W[CA(*imm4*)]
拡張1）不可
拡張2）不可
拡張3）不可

コード



0xB1__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード　Src　即値（符号なし）
Dst　レジスタ直接 *%rd* = %r0～%r15

ＣＬＫ　1サイクル

説　明　(1)標準
*imm4*で指定されるコプロセッサのレジスタの内容を、汎用レジスタ*rd*に転送します。
*imm4*はコプロセッサ専用のアドレスバスに出力されます。

(2)ディレイド命令
本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。

例

```
ld.c  %r1,0x3   ; r1 ← コプロセッサ reg3
```

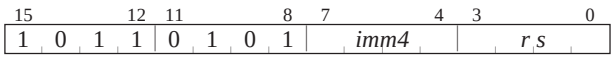
ld.c *imm4*, *%rs*

機　能

コプロセッサへのデータ転送

標準） W[CA(*imm4*)] ← *rs*(7:0)
拡張1) 不可
拡張2) 不可
拡張3) 不可

コード



0xB5__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

Src　レジスタ直接 *%rs* = %r0～%r15
Dst　即値(符号なし)

CLK

1サイクル

説　明

(1)標準
*imm4*で指定されるコプロセッサのレジスタに、汎用レジスタ*rs*の内容を転送します。
*imm4*はコプロセッサ専用のアドレスバスに出力されます。

(2)ディレイド命令
本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。

例

```
ld.c  0x5,%r2   ; コプロセッサ reg5 ← r2
```

ld.cf

機　能　**コプロセッサからC、V、Z、Nフラグを転送**

標準）PSR(3:0) ← コプロセッサフラグ
拡張1）不可
拡張2）不可
拡張3）不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	1	1	0	1	0	0	0	0

0x01D0

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	↔	↔	↔	↔

モード　–

ＣＬＫ　1サイクル

説　明　(1)標準
　コプロセッサからC、V、ZおよびNフラグをPSR(3:0)に転送します。

(2)ディレイド命令
　本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。

例

```
ld.cf       ; コプロセッサフラグをPSRにコピー
```

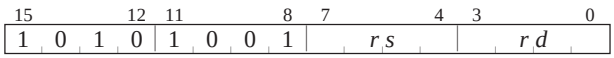
ld.h *%rd, %rs*

機　能

符号付きハーフワードデータ転送

標準） *rd*(15:0) ← *rs*(15:0), *rd*(31:16) ← *rs*(15)
拡張1) 不可
拡張2) 不可
拡張3) 不可

コード



0xA9__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

Src　レジスタ直接 *%rs* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK

1サイクル

説　明

(1)標準
*rs*レジスタの下位16ビット（ハーフワードデータ）を32ビットに符号拡張して*rd*レジスタに転送します。

(2)ディレイド命令
本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。

例

```
ld.h  %r0,%r1   ; r0 ← r1(15:0)を符号拡張
```

ld.h *%rd*, [*%rb*]

機　能

符号付きハーフワードデータ転送

標準） *rd*(15:0) ← H[*rb*], *rd*(31:16) ← H[*rb*](15)
拡張1) *rd*(15:0) ← H[*rb* + *imm13*], *rd*(31:16) ← H[*rb* + *imm13*](15)
拡張2) *rd*(15:0) ← H[*rb* + *imm26*], *rd*(31:16) ← H[*rb* + *imm26*](15)
拡張3) *rd*(15:0) ← H[*rb1* + *rb2*], *rd*(31:16) ← H[*rb1* + *rb2*](15)

コード

15			12	11			8	7			4	3			0
0	0	1	0	1	0	0	0	*rb*				*rd*			

0x28__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

Src　レジスタ間接 *%rb* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK

1サイクル

説　明

(1)標準

```
ld.h  %rd,[%rb]   ; memory address = rb
```

指定メモリのハーフワードデータを32ビットに符号拡張して*rd*レジスタに転送します。*rb*レジスタの内容がアクセスされるメモリアドレスとなります。

(2)拡張1

```
ext   imm13
ld.h  %rd,[%rb]   ; memory address = rb + imm13
```

ext命令により、アドレッシングモードがディスプレースメント付きレジスタ間接アドレッシングに変わります。これにより、*rb*レジスタの内容に13ビット即値*imm13*を加えたアドレスのハーフワードデータを*rd*レジスタに転送します。*rb*レジスタの内容は変更されません。

(3)拡張2

```
ext   imm13       ; = imm26(25:13)
ext   imm13       ; = imm26(12:0)
ld.h  %rd,[%rb]   ; memory address = rb + imm26
```

アドレッシングモードがディスプレースメント付きレジスタ間接アドレッシングに変わり、*rb*レジスタの内容に26ビット即値*imm26*を加えたアドレスのハーフワードデータを*rd*レジスタに転送します。*rb*レジスタの内容は変更されません。

(4)拡張3

```
ext   %rb2
ld.h  %rd,[%rb1]  ; memory address = rb1 + rb2
```

3オペランドのレジスタ間接アドレッシングに変わり、*rb1*レジスタの内容に*rb2*レジスタを加えたアドレスのハーフワードデータを*rd*レジスタに転送します。*rb1*レジスタおよび*rb2*レジスタの内容は変更されません。

注　意

*rb*レジスタおよびディスプレースメントで指定されるメモリアドレスは、ハーフワード境界(最下位ビット = 0)を示していることが必要です。奇数アドレスが指定されると、アドレス不整例外が発生します。

ld.h *%rd*, [*%rb*]+

機　能

符号付きハーフワードデータ転送

標準） *rd*(15:0) ← H[*rb*], *rd*(31:16) ← H[*rb*](15), *rb* ← *rb* + 2
拡張1) *rd*(15:0) ← H[*rb*], *rd*(31:16) ← H[*rb*](15), *rb* ← *rb* + *sign13*
拡張2) *rd*(15:0) ← H[*rb*], *rd*(31:16) ← H[*rb*](15), *rb* ← *rb* + *sign26*
拡張3) *rd*(15:0) ← H[*rb1*], *rd*(31:16) ← H[*rb1*](15), *rb1* ← *rb1* + *rb2*

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	0	0	1	*rb*				*rd*			

0x29__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

Src　ポストインクリメント付きレジスタ間接 *%rb* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK

1サイクル

説　明

(1)標準

```
ld.h  %rd,[%rb]+  ; memory address = rb
```

指定メモリのハーフワードデータを32ビットに符号拡張して*rd*レジスタに転送します。*rb*レジスタの内容がアクセスされるメモリアドレスとなります。データ転送後、*rb*レジスタ内のアドレスをインクリメント(+2)します。

(2)拡張1

```
ext   imm13       ; = sign13
ld.h  %rd,[%rb]+  ; memory address = rb
```

指定メモリのハーフワードデータを32ビットに符号拡張して*rd*レジスタに転送します。*rb*レジスタの内容がアクセスされるメモリアドレスとなります。データ転送後、*rb*レジスタ内のアドレスに*sign13*を加え*rb*に格納します。*sign13*の範囲は、-4,096～+4,094です。

(3)拡張2

```
ext   imm13       ; = sign26(25:13)
ext   imm13       ; = sign26(12:0)
ld.h  %rd,[%rb]+  ; memory address = rb
```

指定メモリのハーフワードデータを32ビットに符号拡張して*rd*レジスタに転送します。*rb*レジスタの内容がアクセスされるメモリアドレスとなります。データ転送後、*rb*レジスタ内のアドレスに*sign26*を加え*rb*に格納します。*sign26*の範囲は、-33,554,432～+33,554,430です。

(4)拡張3

```
ext   %rb2
ld.h  %rd,[%rb1]+ ; memory address = rb1, rb1 ← rb1 + rb2
```

指定メモリのハーフワードデータを32ビットに符号拡張して*rd*レジスタに転送します。*rb1*レジスタの内容がアクセスされるメモリアドレスとなります。データ転送後、*rb1*レジスタ内のアドレスに*rb2*レジスタを加え*rb1*に格納します。*rb2*の範囲は、-2,147,483,648～+2,147,483,646です。

注　意

(1)*rb*レジスタおよびディスプレースメントで指定されるメモリアドレスは、ハーフワード境界(最下位ビット = 0)を示していることが必要です。奇数アドレスが指定されると、アドレス不整例外が発生します。

(2)*rd*と*rb*に同一のレジスタを指定すると、*rd*レジスタにはデータ転送後のインクリメントされたアドレスがロードされます。

ld.h *%rd*, [%dp + *imm6*]

機　能

符号付きハーフワードデータ転送

標準） *rd*(15:0) ← H[dp + *imm6* × 2], *rd*(31:16) ← H[dp + *imm6* × 2](15)
拡張1) *rd*(15:0) ← H[dp + *imm19*], *rd*(31:16) ← H[dp + *imm19*](15)
拡張2) *rd*(15:0) ← H[dp + *imm32*], *rd*(31:16) ← H[dp + *imm32*](15)
拡張3)不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	*imm6*						*rd*			

0xE8__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

Src　ディスプレースメント付きレジスタ間接
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK

1サイクル

説　明

(1)標準

```
ld.h  %rd,[%dp + imm6] ; memory address = dp + imm6 × 2
```

指定メモリのハーフワードデータを32ビットに符号拡張して*rd*レジスタに転送します。現在のDPの内容に6ビット即値*imm6* ×2をディスプレースメントとして加算した値がアクセスされるメモリアドレスとなります。ディスプレースメントの最下位ビットは常に0となります。

(2)拡張1

```
ext    imm13            ; =imm19(18:6)
ld.h   %rd,[%dp + imm6] ; memory address = dp + imm19,
                        ; imm6 = imm19(5:0)
```

ext命令により、ディスプレースメントが19ビットに拡張されます。これにより、DPの内容に19ビット即値*imm19*を加えたアドレスのハーフワードデータを*rd*レジスタに転送します。*imm6*はハーフワード境界(最下位ビット = 0)を指定してください。

(3)拡張2

```
ext    imm13            ; = imm32(31:19)
ext    imm13            ; = imm32(18:6)
ld.h   %rd,[%dp + imm6] ; memory address = dp + imm32,
                        ; imm6 = imm32(5:0)
```

2つのext命令により、ディスプレースメントが32ビットに拡張されます。これにより、DPの内容に32ビット即値*imm32*を加えたアドレスのハーフワードデータを*rd*レジスタに転送します。*imm6*はハーフワード境界(最下位ビット = 0)を指定してください。

例

```
ext   0x1
ld.h  %r0,[%dp + 0x2]     ; r0 ← [dp + 0x42]を符号拡張
```

ld.h *%rd*, [%sp + *imm6*]

機　能

符号付きハーフワードデータ転送

標準） *rd*(15:0) ← H[sp + *imm6* × 2], *rd*(31:16) ← H[sp + *imm6* × 2](15)
拡張1) *rd*(15:0) ← H[sp + *imm19*], *rd*(31:16) ← H[sp + *imm19*](15)
拡張2) *rd*(15:0) ← H[sp + *imm32*], *rd*(31:16) ← H[sp + *imm32*](15)
拡張3)不可

コード

15			12	11	10	9			4	3			0
0	1	0	0	1	0	*imm6*				*rd*			

0x48__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

Src　ディスプレースメント付きレジスタ間接
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK

1サイクル

説　明

(1)標準

```
ld.h  %rd,[%sp + imm6] ; memory address = sp + imm6 × 2
```

指定メモリのハーフワードデータを32ビットに符号拡張して*rd*レジスタに転送します。現在のSPの内容に6ビット即値*imm6* ×2をディスプレースメントとして加算した値がアクセスされるメモリアドレスとなります。ディスプレースメントの最下位ビットは常に0となります。

(2)拡張1

```
ext    imm13            ; =imm19(18:6)
ld.h   %rd,[%sp + imm6] ; memory address = sp + imm19,
                        ; imm6 = imm19(5:0)
```

ext命令により、ディスプレースメントが19ビットに拡張されます。これにより、SPの内容に19ビット即値*imm19*を加えたアドレスのハーフワードデータを*rd*レジスタに転送します。*imm6*はハーフワード境界(最下位ビット = 0)を指定してください。

(3)拡張2

```
ext    imm13            ; = imm32(31:19)
ext    imm13            ; = imm32(18:6)
ld.h   %rd,[%sp + imm6] ; memory address = sp + imm32,
                        ; imm6 = imm32(5:0)
```

2つのext命令により、ディスプレースメントが32ビットに拡張されます。これにより、SPの内容に32ビット即値*imm32*を加えたアドレスのハーフワードデータを*rd*レジスタに転送します。*imm6*はハーフワード境界(最下位ビット = 0)を指定してください。

例

```
ext   0x1
ld.h  %r0,[%sp + 0x2]     ; r0 ← [sp + 0x42]を符号拡張
```

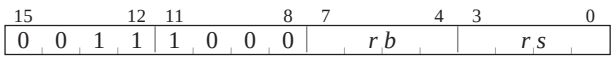
ld.h [*%rb*], *%rs*

機　能

符号付きハーフワードデータ転送

標準） H[*rb*] ← *rs*(15:0)
拡張1) H[*rb* + *imm13*] ← *rs*(15:0)
拡張2) H[*rb* + *imm26*] ← *rs*(15:0)
拡張3) H[*rb1* + *rb2*] ← *rs*(15:0)

コード



0x38__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
−	−	−	−	−	−	−	−	−	−	−	−	−

モード

Src　レジスタ直接 *%rs* = %r0～%r15
Dst　レジスタ間接 *%rb* = %r0～%r15

CLK

1サイクル

説　明

(1)標準

```
ld.h  [%rb],%rs   ; memory address = rb
```

*rs*レジスタの下位16ビットを指定のメモリに転送します。*rb*レジスタの内容がアクセスされるメモリアドレスとなります。

(2)拡張1

```
ext   imm13
ld.h  [%rb],%rs   ; memory address = rb + imm13
```

ext命令により、アドレッシングモードがディスプレースメント付きレジスタ間接アドレッシングに変わります。*rs*レジスタの下位16ビットを、*rb*レジスタの内容に13ビット即値*imm13*を加えたアドレスに転送します。*rb*レジスタの内容は変更されません。

(3)拡張2

```
ext   imm13       ; = imm26(25:13)
ext   imm13       ; = imm26(12:0)
ld.h  [%rb],%rs   ; memory address = rb + imm26
```

アドレッシングモードがディスプレースメント付きレジスタ間接アドレッシングに変わり、*rs*レジスタの下位16ビットを、*rb*レジスタの内容に26ビット即値*imm26*を加えたアドレスに転送します。*rb*レジスタの内容は変更されません。

(4)拡張3

```
ext   %rb2
ld.h  [%rb1],%rs  ; memory address = rb1 + rb2
```

3オペランドのレジスタ間接アドレッシングに変わります。*rs*レジスタの下位16ビットを、*rb1*レジスタの内容に*rb2*レジスタを加えたアドレスに転送します。*rb1*レジスタおよび*rb2*レジスタの内容は変更されません。

注　意

*rb*レジスタおよびディスプレースメントで指定されるメモリアドレスは、ハーフワード境界(最下位ビット = 0)を示していることが必要です。奇数アドレスが指定されると、アドレス不整例外が発生します。

ld.h [*%rb*]+, *%rs*

機　能

符号付きハーフワードデータ転送

標準） H[*rb*] ← *rs*(15:0), *rb* ← *rb* + 2
拡張1) H[*rb*] ← *rs*(15:0), *rb* ← *rb* + *sign13*
拡張2) H[*rb*] ← *rs*(15:0), *rb* ← *rb* + *sign26*
拡張3) H[*rb1*] ← *rs*(15:0), *rb1* ← *rb1* + *rb2*

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	1	*rb*				*rs*			

0x39__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

Src　レジスタ直接 *%rs* = %r0～%r15
Dst　ポストインクリメント付きレジスタ間接 *%rb* = %r0～%r15

CLK

1サイクル

説　明

(1)標準

```
ld.h  [%rb]+,%rs  ; memory address = rb, rb ← rb + 2
```

*rs*レジスタの下位16ビットを指定のメモリに転送します。*rb*レジスタの内容がアクセスされるメモリアドレスとなります。データ転送後、*rb*レジスタ内のアドレスをインクリメント(+2)します。

(2)拡張1

```
ext   imm13       ; = sign13
ld.h  [%rb]+,%rs  ; memory address = rb, rb ← rb + sign13
```

*rs*レジスタの下位16ビットを指定のメモリに転送します。*rb*レジスタの内容がアクセスされるメモリアドレスとなります。データ転送後、*rb*レジスタ内のアドレスに*sign13*を加え、*rb*に格納します。*sign13*の範囲は、-4,096～+4,094です。

(3)拡張2

```
ext   imm13       ; = sign26(25:13)
ext   imm13       ; = sign26(12:0)
ld.h  [%rb]+,%rs  ; memory address = rb, rb ← rb + sign26
```

*rs*レジスタの下位16ビットを指定のメモリに転送します。*rb*レジスタの内容がアクセスされるメモリアドレスとなります。データ転送後、*rb*レジスタ内のアドレスに*sign26*を加え、*rb*に格納します。*sign26*の範囲は、-33,554,432～+33,554,430です。

(4)拡張3

```
ext   %rb2
ld.h  [%rb1]+,%rs ; memory address = rb1, rb1 = rb1 + rb2
```

3オペランドのレジスタ間接アドレッシングに変わります。*rs*レジスタの下位16ビットを*rb1*レジスタが示すアドレスに転送します。データ転送後、*rb1*レジスタの内容に*rb2*レジスタの内容を加え、*rb1*レジスタに格納します。*rb2*の範囲は、-2,147,483,648～+2,147,483,646です。

注　意

*rb*レジスタおよびディスプレースメントで指定されるメモリアドレスは、ハーフワード境界(最下位ビット = 0)を示していることが必要です。奇数アドレスが指定されると、アドレス不整例外が発生します。

ld.h [%dp + *imm6*], *%rs*

機　能

符号付きハーフワードデータ転送

標準） H[dp + *imm6* × 2] ← *rs*(15:0)
拡張1) H[dp + *imm19*] ← *rs*(15:0)
拡張2) H[dp + *imm32*] ← *rs*(15:0)
拡張3) 不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	*imm6*						*rs*			

0xF8__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

Src　レジスタ直接 *%rs* = %r0～%r15
Dst　ディスプレースメント付きレジスタ間接

CLK

1サイクル

説　明

(1)標準

```
ld.h  [%dp + imm6],%rs ; memory address = dp + imm6×2
```

*rs*レジスタの下位16ビットを指定メモリに転送します。現在のDPの内容に6ビット即値*imm6*×2をディスプレースメントとして加算した値がアクセスされるメモリアドレスとなります。ディスプレースメントの最下位ビットは常に0となります。

(2)拡張1

```
ext   imm13             ; = imm19(18:6)
ld.h  [%dp + imm6],%rs  ; memory address = dp + imm19,
                        ; imm6 = imm19(5:0)
```

ext命令により、ディスプレースメントが19ビットに拡張されます。これにより、*rs*レジスタの下位16ビットを、DPの内容に19ビット即値*imm19*を加えたアドレスに転送します。*imm6*はハーフワード境界(最下位ビット = 0)を指定してください。

(3)拡張2

```
ext   imm13             ; = imm32(31:19)
ext   imm13             ; = imm32(18:6)
ld.h  [%dp + imm6],%rs  ; memory address = dp + imm32,
                        ; imm6 = imm32(5:0)
```

2つのext命令により、ディスプレースメントが32ビットに拡張されます。これにより、*rs*レジスタの下位16ビットを、DPの内容に32ビット即値*imm32*を加えたアドレスに転送します。*imm6*はハーフワード境界(最下位ビット = 0)を指定してください。

例

```
ext   0x1
ld.h  [%dp + 0x2],%r0     ; H[dp + 0x42] ← r0の下位16ビット
```

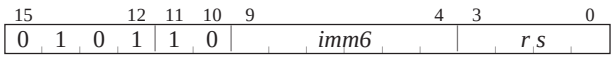
ld.h [%sp + *imm6*], *%rs*

機　能

符号付きハーフワードデータ転送

標準） H[sp + *imm6* × 2] ← *rs*(15:0)
拡張1) H[sp + *imm19*] ← *rs*(15:0)
拡張2) H[sp + *imm32*] ← *rs*(15:0)
拡張3) 不可

コード



0x58__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

Src　レジスタ直接 *%rs* = %r0～%r15
Dst　ディスプレースメント付きレジスタ間接

CLK

1サイクル

説　明

(1)標準

```
ld.h  [%sp + imm6],%rs  ; memory address = sp + imm6×2
```

*rs*レジスタの下位16ビットを指定メモリに転送します。現在のSPの内容に6ビット即値*imm6*×2をディスプレースメントとして加算した値がアクセスされるメモリアドレスとなります。ディスプレースメントの最下位ビットは常に0となります。

(2)拡張1

```
ext   imm13             ; = imm19(18:6)
ld.h  [%sp + imm6],%rs  ; memory address = sp + imm19,
                        ; imm6 = imm19(5:0)
```

ext命令により、ディスプレースメントが19ビットに拡張されます。これにより、*rs*レジスタの下位16ビットを、SPの内容に19ビット即値*imm19*を加えたアドレスに転送します。*imm6*はハーフワード境界(最下位ビット = 0)を指定してください。

(3)拡張2

```
ext   imm13             ; = imm32(31:19)
ext   imm13             ; = imm32(18:6)
ld.h  [%sp + imm6],%rs  ; memory address = sp + imm32,
                        ; imm6 = imm32(5:0)
```

2つのext命令により、ディスプレースメントが32ビットに拡張されます。これにより、*rs*レジスタの下位16ビットを、SPの内容に32ビット即値*imm32*を加えたアドレスに転送します。*imm6*はハーフワード境界(最下位ビット = 0)を指定してください。

例

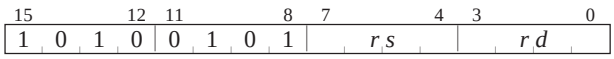
```
ext   0x1
ld.h  [%sp + 0x2],%r0     ; H[sp + 0x42] ← r0の下位16ビット
```

ld.ub *%rd, %rs*

機　能　**符号なしバイトデータ転送**

標準）　*rd*(7:0) ← *rs*(7:0), *rd*(31:8) ← 0
拡張1）不可
拡張2）不可
拡張3）不可

コード



0xA5__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
−	−	−	−	−	−	−	−	−	−	−	−	−

モード　Src　レジスタ直接 *%rs* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK　1サイクル

説　明　(1)標準
*rs*レジスタの下位8ビット（バイトデータ）を32ビットにゼロ拡張して*rd*レジスタに転送します。

(2)ディレイド命令
本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。

例

```
ld.ub  %r0,%r1  ; r0 ← r1(7:0)をゼロ拡張
```

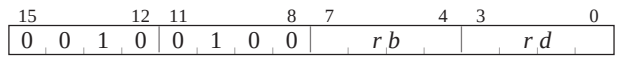
ld.ub *%rd*, [*%rb*]

機　能

符号なしバイトデータ転送

標準） *rd*(7:0) ← B[*rb*], *rd*(31:8) ← 0
拡張1) *rd*(7:0) ← B[*rb* + *imm13*], *rd*(31:8) ← 0
拡張2) *rd*(7:0) ← B[*rb* + *imm26*], *rd*(31:8) ← 0
拡張3) *rd*(7:0) ← B[*rb1* + *rb2*], *rd*(31:8) ← 0

コード



0x24__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

Src　レジスタ間接 *%rb* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK

1サイクル

説　明

(1)標準

```
ld.ub  %rd,[%rb]  ; memory address = rb
```

指定メモリのバイトデータを32ビットにゼロ拡張して*rd*レジスタに転送します。*rb*レジスタの内容がアクセスされるメモリアドレスとなります。

(2)拡張1

```
ext    imm13
ld.ub  %rd,[%rb]  ; memory address = rb + imm13
```

ext命令により、アドレッシングモードがディスプレースメント付きレジスタ間接アドレッシングに変わります。これにより、*rb*レジスタの内容に13ビット即値*imm13*を加えたアドレスのバイトデータを*rd*レジスタに転送します。*rb*レジスタの内容は変更されません。

(3)拡張2

```
ext    imm13      ; = imm26(25:13)
ext    imm13      ; = imm26(12:0)
ld.ub  %rd,[%rb]  ; memory address = rb + imm26
```

アドレッシングモードがディスプレースメント付きレジスタ間接アドレッシングに変わり、*rb*レジスタの内容に26ビット即値*imm26*を加えたアドレスのバイトデータを*rd*レジスタに転送します。*rb*レジスタの内容は変更されません。

(4)拡張3

```
ext    %rb2
ld.ub  %rd,[%rb1] ; memory address = rb1 + rb2
```

3オペランドのレジスタ間接アドレッシングに変わり、*rb1*レジスタの内容に*rb2*レジスタを加えたアドレスのバイトデータを*rd*レジスタに転送します。*rb1*レジスタおよび*rb2*レジスタの内容は変更されません。

ld.ub *%rd*, [*%rb*]+

機　能

符号なしバイトデータ転送

標準） *rd*(7:0) ← B[*rb*], *rd*(31:8) ← 0, *rb* ← *rb* + 1
拡張1) *rd*(7:0) ← B[*rb*], *rd*(31:8) ← 0, *rb* ← *rb* + *sign13*
拡張2) *rd*(7:0) ← B[*rb*], *rd*(31:8) ← 0, *rb* ← *rb* + *sign26*
拡張3) *rd*(7:0) ← B[*rb1*], *rd*(31:8) ← 0, *rb1* ← *rb1* + *rb2*

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	*rb*				*rd*			

0x25__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

Src　ポストインクリメント付きレジスタ間接 *%rb* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK

1サイクル

説　明

(1)標準

```
ld.ub  %rd,[%rb]+ ; memory address = rb
```

指定メモリのバイトデータを32ビットにゼロ拡張して*rd*レジスタに転送します。*rb*レジスタの内容がアクセスされるメモリアドレスとなります。データ転送後、*rb*レジスタ内のアドレスをインクリメント(+1)します。

(2)拡張1

```
ext    imm13      ; = sign13
ld.ub  %rd,[%rb]+ ; memory address = rb
```

指定メモリのバイトデータを32ビットにゼロ拡張して*rd*レジスタに転送します。*rb*レジスタの内容がアクセスされるメモリアドレスとなります。データ転送後、*rb*レジスタ内のアドレスに*sign13*を加え*rb*に格納します。*sign13*の範囲は、-4,096～+4,095です。

(3)拡張2

```
ext    imm13      ; = sign26(25:13)
ext    imm13      ; = sign26(12:0)
ld.ub  %rd,[%rb]+ ; memory address = rb
```

指定メモリのバイトデータを32ビットにゼロ拡張して*rd*レジスタに転送します。*rb*レジスタの内容がアクセスされるメモリアドレスとなります。データ転送後、*rb*レジスタ内のアドレスに*sign26*を加え*rb*に格納します。*sign26*の範囲は、-33,554,432～+33,554,431です。

(4)拡張3

```
ext    %rb2
ld.ub  %rd,[%rb1]+; memory address = rb1, rb1 ← rb1 + rb2
```

指定メモリのバイトデータを32ビットにゼロ拡張して*rd*レジスタに転送します。*rb1*レジスタの内容がアクセスされるメモリアドレスとなります。データ転送後、*rb1*レジスタ内のアドレスに*rb2*レジスタを加え*rb1*に格納します。*rb2*の範囲は、-2,147,483,648～+2,147,483,647です。

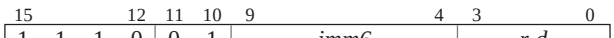
ld.ub *%rd*, [%dp + *imm6*]

機　能

符号なしバイトデータ転送

標準） *rd*(7:0) ← B[dp + *imm6*], *rd*(31:8) ← 0
拡張1) *rd*(7:0) ← B[dp + *imm19*], *rd*(31:8) ← 0
拡張2) *rd*(7:0) ← B[dp + *imm32*], *rd*(31:8) ← 0
拡張3)不可

コード



0xE4__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

Src　ディスプレースメント付きレジスタ間接
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK

1サイクル

説　明

(1)標準

```
ld.ub  %rd,[%dp + imm6] ; memory address = dp + imm6
```

指定メモリのバイトデータを32ビットにゼロ拡張して*rd*レジスタに転送します。現在のDPの内容に6ビット即値*imm6*をディスプレースメントとして加算した値がアクセスされるメモリアドレスとなります。

(2)拡張1

```
ext    imm13            ; = imm19(18:6)
ld.ub  %rd,[%dp + imm6] ; memory address = dp + imm19,
                        ; imm6 ← imm19(5:0)
```

ext命令により、ディスプレースメントが19ビットに拡張されます。これにより、DPの内容に19ビット即値*imm19*を加えたアドレスのバイトデータを*rd*レジスタに転送します。

(3)拡張2

```
ext    imm13            ; = imm32(31:19)
ext    imm13            ; = imm32(18:6)
ld.ub  %rd,[%dp + imm6] ; memory address = dp + imm32,
                        ; imm6 ← imm32(5:0)
```

2つのext命令により、ディスプレースメントが32ビットに拡張されます。これにより、DPの内容に32ビット即値*imm32*を加えたアドレスのバイトデータを*rd*レジスタに転送します。

例

```
ext    0x1
ld.ub  %r0,[%dp + 0x1]    ; r0 ← [dp + 0x41]をゼロ拡張
```

ld.ub *%rd*, [%sp + *imm6*]

機　能　**符号なしバイトデータ転送**

標準） *rd*(7:0) ← B[sp + *imm6*], *rd*(31:8) ← 0
拡張1) *rd*(7:0) ← B[sp + *imm19*], *rd*(31:8) ← 0
拡張2) *rd*(7:0) ← B[sp + *imm32*], *rd*(31:8) ← 0
拡張3)不可

コード

15	14	13	12	11	10	9 … 4	3 … 0
0	1	0	0	0	1	*imm6*	*rd*

0x44__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード　Src　ディスプレースメント付きレジスタ間接
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK　1サイクル

説　明　(1)標準

```
ld.ub  %rd,[%sp + imm6]; memory address = sp + imm6
```

指定メモリのバイトデータを32ビットにゼロ拡張して*rd*レジスタに転送します。現在のSPの内容に6ビット即値*imm6*をディスプレースメントとして加算した値がアクセスされるメモリアドレスとなります。

(2)拡張1

```
ext    imm13            ; = imm19(18:6)
ld.ub  %rd,[%sp + imm6]; memory address = sp + imm19,
                        ; imm6 ← imm19(5:0)
```

ext命令により、ディスプレースメントが19ビットに拡張されます。これにより、SPの内容に19ビット即値*imm19*を加えたアドレスのバイトデータを*rd*レジスタに転送します。

(3)拡張2

```
ext    imm13            ; = imm32(31:19)
ext    imm13            ; = imm32(18:6)
ld.ub  %rd,[%sp + imm6]; memory address = sp + imm32,
                        ; imm6 ← imm32(5:0)
```

2つのext命令により、ディスプレースメントが32ビットに拡張されます。これにより、SPの内容に32ビット即値*imm32*を加えたアドレスのバイトデータを*rd*レジスタに転送します。

例

```
ext    0x1
ld.ub  %r0,[%sp + 0x1]    ; r0 ← [sp + 0x41]をゼロ拡張
```

ld.uh *%rd, %rs*

機　能

符号なしハーフワードデータ転送

標準） *rd*(15:0) ← *rs*(15:0), *rd*(31:16) ← 0
拡張1) 不可
拡張2) 不可
拡張3) 不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	1	1	0	1	*rs*				*rd*			

0xAD__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

Src　レジスタ直接 *%rs* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK

1サイクル

説　明

(1)標準
*rs*レジスタの下位16ビット（ハーフワードデータ）を32ビットにゼロ拡張して*rd*レジスタに転送します。

(2)ディレイド命令
本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。

例

```
ld.uh  %r0,%r1  ; r0 ← r1(15:0)をゼロ拡張
```

ld.uh *%rd*, [*%rb*]

機　能　**符号なしハーフワードデータ転送**

標準）*rd*(15:0) ← H[*rb*], *rd*(31:16) ← 0
拡張1）*rd*(15:0) ← H[*rb* + *imm13*], *rd*(31:16) ← 0
拡張2）*rd*(15:0) ← H[*rb* + *imm26*], *rd*(31:16) ← 0
拡張3）*rd*(15:0) ← H[*rb1* + *rb2*], *rd*(31:16) ← 0

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	1	0	0	*rb*				*rd*			

0x2C__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード　Src　レジスタ間接 *%rb* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK　1サイクル

説　明　(1)標準

```
ld.uh  %rd,[%rb]  ; memory address = rb
```

指定メモリのハーフワードデータを32ビットにゼロ拡張して*rd*レジスタに転送します。*rb*レジスタの内容がアクセスされるメモリアドレスとなります。

(2)拡張1

```
ext    imm13
ld.uh  %rd,[%rb]  ; memory address = rb + imm13
```

ext命令により、アドレッシングモードがディスプレースメント付きレジスタ間接アドレッシングに変わります。これにより、*rb*レジスタの内容に13ビット即値*imm13*を加えたアドレスのハーフワードデータを*rd*レジスタに転送します。*rb*レジスタの内容は変更されません。

(3)拡張2

```
ext    imm13      ; = imm26(25:13)
ext    imm13      ; = imm26(12:0)
ld.uh  %rd,[%rb]  ; memory address = rb + imm26
```

アドレッシングモードがディスプレースメント付きレジスタ間接アドレッシングに変わり、*rb*レジスタの内容に26ビット即値*imm26*を加えたアドレスのハーフワードデータを*rd*レジスタに転送します。*rb*レジスタの内容は変更されません。

(4)拡張3

```
ext    %rb2
ld.uh  %rd,[%rb1] ; memory address = rb1 + rb2
```

3オペランドのレジスタ間接アドレッシングに変わり、*rb1*レジスタの内容に*rb2*レジスタを加えたアドレスのハーフワードデータを*rd*レジスタに転送します。*rb1*レジスタおよび*rb2*レジスタの内容は変更されません。

注　意　*rb*レジスタおよびディスプレースメントで指定されるメモリアドレスは、ハーフワード境界(最下位ビット = 0)を示していることが必要です。奇数アドレスが指定されると、アドレス不整例外が発生します。

ld.uh *%rd*, [*%rb*]+

機　能

符号なしハーフワードデータ転送

標準） *rd*(15:0) ← H[*rb*], *rd*(31:16) ← 0, *rb* ← *rb* + 2
拡張1) *rd*(15:0) ← H[*rb*], *rd*(31:16) ← 0, *rb* ← *rb* + *sign13*
拡張2) *rd*(15:0) ← H[*rb*], *rd*(31:16) ← 0, *rb* ← *rb* + *sign26*
拡張3) *rd*(15:0) ← H[*rb1*], *rd*(31:16) ← 0, *rb1* ← *rb1* + *rb2*

コード

15			12	11			8	7			4	3			0
0	0	1	0	1	1	0	1	*rb*				*rd*			

0x2D__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

Src　ポストインクリメント付きレジスタ間接 *%rb* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK

1サイクル

説　明

(1)標準

```
ld.uh  %rd,[%rb]+ ; memory address = rb
```

指定メモリのハーフワードデータを32ビットにゼロ拡張して*rd*レジスタに転送します。*rb*レジスタの内容がアクセスされるメモリアドレスとなります。データ転送後、*rb*レジスタ内のアドレスをインクリメント(+2)します。

(2)拡張1

```
ext    imm13      ; = sign13
ld.uh  %rd,[%rb]+ ; memory address = rb
```

指定メモリのハーフワードデータを32ビットにゼロ拡張して*rd*レジスタに転送します。*rb*レジスタの内容がアクセスされるメモリアドレスとなります。データ転送後、*rb*レジスタ内のアドレスに*sign13*を加え*rb*に格納します。*sign13*の範囲は、-4,096～+4,094です。

(3)拡張2

```
ext    imm13      ; = sign26(25:13)
ext    imm13      ; = sign26(12:0)
ld.uh  %rd,[%rb]+ ; memory address = rb
```

指定メモリのハーフワードデータを32ビットにゼロ拡張して*rd*レジスタに転送します。*rb*レジスタの内容がアクセスされるメモリアドレスとなります。データ転送後、*rb*レジスタ内のアドレスに*sign26*を加え*rb*に格納します。*sign26*の範囲は、-33,554,432～+33,554,430です。

(4)拡張3

```
ext    %rb2
ld.uh  %rd,[%rb1]+; memory address = rb1, rb1 ← rb1 + rb2
```

指定メモリのハーフワードデータを32ビットにゼロ拡張して*rd*レジスタに転送します。*rb1*レジスタの内容がアクセスされるメモリアドレスとなります。データ転送後、*rb1*レジスタ内のアドレスに*rb2*レジスタを加え*rb1*に格納します。*rb2*の範囲は、-2,147,483,648～+2,147,483,646です。

注　意

(1)*rb*レジスタおよびディスプレースメントで指定されるメモリアドレスは、ハーフワード境界(最下位ビット = 0)を示していることが必要です。奇数アドレスが指定されると、アドレス不整例外が発生します。

(2)*rd*と*rb*に同一のレジスタを指定すると、*rd*レジスタにはデータ転送後のインクリメントされたアドレスがロードされます。

ld.uh *%rd*, [%dp + *imm6*]

機　能

符号なしハーフワードデータ転送

標準） *rd*(15:0) ← H[dp + *imm6* × 2], *rd*(31:16) ← 0
拡張1) *rd*(15:0) ← H[dp + *imm19*], *rd*(31:16) ← 0
拡張2) *rd*(15:0) ← H[dp + *imm32*], *rd*(31:16) ← 0
拡張3)不可

コード

15	14	13	12	11	10	9 … 4	3 … 0
1	1	1	0	1	1	*imm6*	*rd*

0xEC__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

Src　ディスプレースメント付きレジスタ間接
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK

1サイクル

説　明

(1)標準

```
ld.uh  %rd,[%dp + imm6]; memory address = dp + imm6 × 2
```

指定メモリのハーフワードデータを32ビットにゼロ拡張して*rd*レジスタに転送します。現在のDPの内容に6ビット即値*imm6* ×2をディスプレースメントとして加算した値がアクセスされるメモリアドレスとなります。ディスプレースメントの最下位ビットは常に0となります。

(2)拡張1

```
ext    imm13            ; =imm19(18:6)
ld.uh  %rd,[%dp + imm6]; memory address = dp + imm19,
                        ; imm6 = imm19(5:0)
```

ext命令により、ディスプレースメントが19ビットに拡張されます。これにより、DPの内容に19ビット即値*imm19*を加えたアドレスのハーフワードデータを*rd*レジスタに転送します。*imm6*はハーフワード境界(最下位ビット = 0)を指定してください。

(3)拡張2

```
ext    imm13            ; = imm32(31:19)
ext    imm13            ; = imm32(18:6)
ld.uh  %rd,[%dp + imm6]; memory address = dp + imm32,
                        ; imm6 = imm32(5:0)
```

2つのext命令により、ディスプレースメントが32ビットに拡張されます。これにより、DPの内容に32ビット即値*imm32*を加えたアドレスのハーフワードデータを*rd*レジスタに転送します。*imm6*はハーフワード境界(最下位ビット = 0)を指定してください。

例

```
ext    0x1
ld.uh  %r0,[%dp + 0x2]    ; r0 ← [dp + 0x42]をゼロ拡張
```

ld.uh *%rd*, [%sp + *imm6*]

機　能

符号なしハーフワードデータ転送

標準） *rd*(15:0) ← H[sp + *imm6* × 2], *rd*(31:16) ← 0
拡張1) *rd*(15:0) ← H[sp + *imm19*], *rd*(31:16) ← 0
拡張2) *rd*(15:0) ← H[sp + *imm32*], *rd*(31:16) ← 0
拡張3)不可

コード

15			12	11	10	9			4	3			0
0	1	0	0	1	1	*imm6*				*r d*			

0x4C__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

Src　ディスプレースメント付きレジスタ間接
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK

1サイクル

説　明

(1)標準

```
ld.uh  %rd,[%sp + imm6] ; memory address = sp + imm6 × 2
```

指定メモリのハーフワードデータを32ビットにゼロ拡張して*rd*レジスタに転送します。現在のSPの内容に6ビット即値*imm6* ×2をディスプレースメントとして加算した値がアクセスされるメモリアドレスとなります。ディスプレースメントの最下位ビットは常に0となります。

(2)拡張1

```
ext    imm13            ; =imm19(18:6)
ld.uh  %rd,[%sp + imm6] ; memory address = sp + imm19,
                        ; imm6 = imm19(5:0)
```

ext命令により、ディスプレースメントが19ビットに拡張されます。これにより、SPの内容に19ビット即値*imm19*を加えたアドレスのハーフワードデータを*rd*レジスタに転送します。*imm6*はハーフワード境界(最下位ビット = 0)を指定してください。

(3)拡張2

```
ext    imm13            ; = imm32(31:19)
ext    imm13            ; = imm32(18:6)
ld.uh  %rd,[%sp + imm6] ; memory address = sp + imm32,
                        ; imm6 = imm32(5:0)
```

2つのext命令により、ディスプレースメントが32ビットに拡張されます。これにより、SPの内容に32ビット即値*imm32*を加えたアドレスのハーフワードデータを*rd*レジスタに転送します。*imm6*はハーフワード境界(最下位ビット = 0)を指定してください。

例

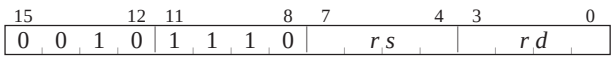
```
ext    0x1
ld.uh  %r0,[%sp + 0x2]    ; r0 ← [sp + 0x42]をゼロ拡張
```

ld.w *%rd, %rs*

機　能　**ワードデータ転送**

標準）*rd* ← *rs*
拡張1）不可
拡張2）不可
拡張3）不可

コード



0x2E__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
−	−	−	−	−	−	−	−	−	−	−	−	−

モード　Src　レジスタ直接 *%rs* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK　1サイクル

説　明　(1)標準
*rs*レジスタの内容(ワードデータ)を*rd*レジスタに転送します。

(2)ディレイド命令
本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。

例

```
ld.w  %r0,%r1   ; r0 ← r1
```

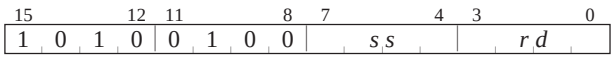
ld.w *%rd, %ss*

機　能

ワードデータ転送

標準）*rd* ← *ss*
拡張1）不可
拡張2）不可
拡張3）不可

コード



0xA4__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

Src　レジスタ直接 *%ss* = %psr, %sp, %alr, %ahr, %lco, %lsa, %lea, %sor, %ttbr, %dp, %idir, %dbbr, %usp, %ssp, %pc
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK

1サイクル

説　明

(1)標準
特殊レジスタの内容(ワードデータ)を*rd*レジスタに転送します。

(2)ディレイド命令
本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。

例

```
ld.w  %r0,%psr  ; r0 ← psr
```

注　意

(1)%spは動作モードによってSSPまたはUSPが参照されます。
ユーザモード　→ USP
スーパバイザモード → SSP

(2)ld.w *%rd*,%pc命令を実行すると、レジスタに読み込まれる値はこのld.w命令のPC値+2となります。この命令はディレイドスロット命令として使用してください。ディレイド分岐命令の直後以外の場所に記述した場合、*rd*レジスタにロードされるPC値がld.wの次の命令のアドレスを示すとは限りません。

ld.w *%rd*, [*%rb*]

機　能　**ワードデータ転送**

標準）*rd* ← W[*rb*]
拡張1）*rd* ← W[*rb* + *imm13*]
拡張2）*rd* ← W[*rb* + *imm26*]
拡張3）*rd* ← W[*rb1* + *rb2*]

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	0	0	0	*rb*				*rd*			

0x30__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード　Src　レジスタ間接 *%rb* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK　1サイクル

説　明　(1)標準

```
ld.w  %rd,[%rb]   ; memory address = rb
```

指定メモリのワードデータを*rd*レジスタに転送します。*rb*レジスタの内容がアクセスされるメモリアドレスとなります。

(2)拡張1

```
ext   imm13
ld.w  %rd,[%rb]   ; memory address = rb + imm13
```

ext命令により、アドレッシングモードがディスプレースメント付きレジスタ間接アドレッシングに変わります。これにより、*rb*レジスタの内容に13ビット即値*imm13*を加えたアドレスのワードデータを*rd*レジスタに転送します。*rb*レジスタの内容は変更されません。

(3)拡張2

```
ext   imm13       ; = imm26(25:13)
ext   imm13       ; = imm26(12:0)
ld.w  %rd,[%rb]   ; memory address = rb + imm26
```

アドレッシングモードがディスプレースメント付きレジスタ間接アドレッシングに変わり、*rb*レジスタの内容に26ビット即値*imm26*を加えたアドレスのワードデータを*rd*レジスタに転送します。*rb*レジスタの内容は変更されません。

(4)拡張3

```
ext   %rb2
ld.w  %rd,[%rb1]  ; memory address = rb1 + rb2
```

3オペランドのレジスタ間接アドレッシングに変わり、*rb1*レジスタの内容に*rb2*レジスタを加えたアドレスのワードデータを*rd*レジスタに転送します。*rb1*レジスタおよび*rb2*レジスタの内容は変更されません。

注　意　*rb*レジスタおよびディスプレースメントで指定されるメモリアドレスは、ワード境界(下位2ビット = 0)を示していることが必要です。それ以外のアドレスが指定されると、アドレス不整例外が発生します。

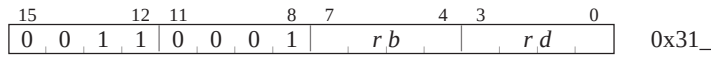
ld.w *%rd*, [*%rb*]+

機　能

ワードデータ転送

標準） *rd* ← W[*rb*], *rb* ← *rb* + 4
拡張1) *rd* ← W[*rb*], *rb* ← *rb* + *sign13*
拡張2) *rd* ← W[*rb*], *rb* ← *rb* + *sign26*
拡張3) *rd* ← W[*rb1*], *rb1* ← *rb1* + *rb2*

コード



0x31__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

Src　ポストインクリメント付きレジスタ間接 *%rb* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK

1サイクル

説　明

(1)標準

```
ld.w  %rd,[%rb]+  ; memory address = rb
```

指定メモリのワードデータを*rd*レジスタに転送します。*rb*レジスタの内容がアクセスされるメモリアドレスとなります。データ転送後、*rb*レジスタ内のアドレスをインクリメント(+4)します。

(2)拡張1

```
ext   imm13       ; = sign13
ld.w  %rd,[%rb]+  ; memory address = rb
```

指定メモリのワードデータを*rd*レジスタに転送します。*rb*レジスタの内容がアクセスされるメモリアドレスとなります。データ転送後、*rb*レジスタ内のアドレスに*sign13*を加え*rb*に格納します。*sign13*の範囲は、-4,096～+4,092です。

(3)拡張2

```
ext   imm13       ; = sign26(25:13)
ext   imm13       ; = sign26(12:0)
ld.w  %rd,[%rb]+  ; memory address = rb
```

指定メモリのワードデータを*rd*レジスタに転送します。*rb*レジスタの内容がアクセスされるメモリアドレスとなります。データ転送後、*rb*レジスタ内のアドレスに*sign26*を加え*rb*に格納します。*sign26*の範囲は、-33,554,432～+33,554,428です。

(4)拡張3

```
ext   %rb2
ld.w  %rd,[%rb1]+ ; memory address = rb1, rb1 ← rb1 + rb2
```

指定メモリのワードデータを*rd*レジスタに転送します。*rb1*レジスタの内容がアクセスされるメモリアドレスとなります。データ転送後、*rb1*レジスタ内のアドレスに*rb2*レジスタを加え*rb1*に格納します。*rb2*の範囲は、-2,147,483,648～+2,147,483,644です。

注　意

(1)*rb*レジスタおよびディスプレースメントで指定されるメモリアドレスは、ワード境界(下位2ビット = 0)を示していることが必要です。それ以外のアドレスが指定されると、アドレス不整例外が発生します。

(2)*rd*と*rb*に同一のレジスタを指定すると、*rd*レジスタにはデータ転送後のインクリメントされたアドレスがロードされます。

ld.w *%rd*, [%dp + *imm6*]

機　能

ワードデータ転送

標準） *rd* ← W[dp + *imm6* × 4]
拡張1) *rd* ← W[dp + *imm19*]
拡張2) *rd* ← W[dp + *imm32*]
拡張3)不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	0	*imm6*						*rd*				0xF0__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

Src　ディスプレースメント付きレジスタ間接
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK

1サイクル

説　明

(1)標準

```
ld.w  %rd,[%dp + imm6] ; memory address = dp + imm6 × 4
```

指定メモリのワードデータを*rd*レジスタに転送します。現在のDPの内容に6ビット即値*imm6* ×4をディスプレースメントとして加算した値がアクセスされるメモリアドレスとなります。ディスプレースメントの下位2ビットは常に0となります。

(2)拡張1

```
ext    imm13            ; =imm19(18:6)
ld.w   %rd,[%dp + imm6] ; memory address = dp + imm19,
                        ; imm6 = imm19(5:0)
```

ext命令により、ディスプレースメントが19ビットに拡張されます。これにより、DPの内容に19ビット即値*imm19*を加えたアドレスのワードデータを*rd*レジスタに転送します。*imm6*はワード境界(下位2ビット = 0)を指定してください。

(3)拡張2

```
ext    imm13            ; = imm32(31:19)
ext    imm13            ; = imm32(18:6)
ld.w   %rd,[%dp + imm6] ; memory address = dp + imm32,
                        ; imm6 = imm32(5:0)
```

2つのext命令により、ディスプレースメントが32ビットに拡張されます。これにより、DPの内容に32ビット即値*imm32*を加えたアドレスのワードデータを*rd*レジスタに転送します。*imm6*はワード境界(下位2ビット = 0)を指定してください。

ld.w *%rd*, [%sp + *imm6*]

機　能

ワードデータ転送

標準） *rd* ← W[sp + *imm6* × 4]
拡張1) *rd* ← W[sp + *imm19*]
拡張2) *rd* ← W[sp + *imm32*]
拡張3)不可

コード

15			12	11	10	9			4	3			0
0	1	0	1	0	0	*imm6*				*r d*			

0x50__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

Src　ディスプレースメント付きレジスタ間接
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK

1サイクル

説　明

(1)標準

```
ld.w  %rd,[%sp + imm6] ; memory address = sp + imm6 × 4
```

指定メモリのワードデータを*rd*レジスタに転送します。現在のSPの内容に6ビット即値*imm6* ×4をディスプレースメントとして加算した値がアクセスされるメモリアドレスとなります。ディスプレースメントの下位2ビットは常に0となります。

(2)拡張1

```
ext    imm13            ; =imm19(18:6)
ld.w   %rd,[%sp + imm6] ; memory address = sp + imm19,
                        ; imm6 = imm19(5:0)
```

ext命令により、ディスプレースメントが19ビットに拡張されます。これにより、SPの内容に19ビット即値*imm19*を加えたアドレスのワードデータを*rd*レジスタに転送します。*imm6*はワード境界(下位2ビット = 0)を指定してください。

(3)拡張2

```
ext    imm13            ; = imm32(31:19)
ext    imm13            ; = imm32(18:6)
ld.w   %rd,[%sp + imm6] ; memory address = sp + imm32,
                        ; imm6 = imm32(5:0)
```

2つのext命令により、ディスプレースメントが32ビットに拡張されます。これにより、SPの内容に32ビット即値*imm32*を加えたアドレスのワードデータを*rd*レジスタに転送します。*imm6*はワード境界(下位2ビット = 0)を指定してください。

ld.w *%rd, sign6*

機　能

ワードデータ転送

標準） *rd*(5:0) ← *sign6*(5:0), *rd*(31:6) ← *sign6*(5)
拡張1) *rd*(18:0) ← *sign19*(18:0), *rd*(31:19) ← *sign19*(18)
拡張2) *rd* ← *sign32*
拡張3) 不可

コード

15			12	11	10	9				4	3			0
0	1	1	0	1	1	*sign6*					*rd*			

0x6C__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
−	−	−	−	−	−	−	−	−	−	−	−	−

モード

Src　即値(符号付き)
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK

1サイクル

説　明

(1)標準

```
ld.w  %rd,sign6   ; rd ← sign6 (sign-extended)
```

6ビット即値データ*sign6*を32ビットに符号拡張して*rd*レジスタにロードします。

(2)拡張1

```
ext   imm13       ; = sign19(18:6)
ld.w  %rd,sign6   ; rd ← sign19 (sign-extended),
                  ; sign6 = sign19(5:0)
```

ext命令で拡張した19ビット即値データ*sign19*を32ビットに符号拡張して*rd*レジスタにロードします。

(3)拡張2

```
ext   imm13       ; = sign32(31:19)
ext   imm13       ; = sign32(18:6)
ld.w  %rd,sign6   ; rd ← sign32, sign6 = sign32(5:0)
```

ext命令で拡張した32ビット即値データ*sign32*を*rd*レジスタにロードします。

(4)ディレイド命令

本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。ただし、その場合はext命令による拡張はできません。

例

```
ld.w  %r0,0x3f  ; r0 ← 0xffffffff
```

ld.w *%sd, %rs*

機　能

ワードデータ転送

標準）*sd* ← *rs*
拡張1）不可
拡張2）不可
拡張3）不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	0	0	*rs*				*sd*			

0xA0__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

*sd*がPSRの場合、*rs*の内容がコピーされます。

モード

Src　レジスタ直接 *%rs* = %r0～%r15
Dst　レジスタ直接 *%sd* = %psr, %sp, %alr, %ahr, %lco, %lsa, %lea, %sor, %ttbr, %dp, %idir, %dbbr, %usp, %ssp, %pc

CLK

1サイクル（ld.w %psr,*%rs*命令を実行するときのみ4サイクル）

説　明

(1)標準
*rs*レジスタの内容（ワードデータ）を特殊レジスタに転送します。

(2)ディレイド命令
本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。

例

```
ld.w  %sp,%r0   ; sp ← r0
```

注　意

%spは動作モードによってSSPまたはUSPが参照されます。
ユーザモード　　→ USP
スーパバイザモード → SSP

ユーザモード時はSSPを変更することはできません。

ld.w [*%rb*], *%rs*

機　能

ワードデータ転送

標準） W[*rb*] ← *rs*
拡張1）W[*rb* + *imm13*] ← *rs*
拡張2）W[*rb* + *imm26*] ← *rs*
拡張3）W[*rb1* + *rb2*] ← *rs*

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	1	0	0	*rb*				*rs*			

0x3C__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
−	−	−	−	−	−	−	−	−	−	−	−	−

モード

Src　レジスタ直接 *%rs* = %r0～%r15
Dst　レジスタ間接 *%rb* = %r0～%r15

CLK

1サイクル

説　明

(1)標準

```
ld.w  [%rb],%rs   ; memory address = rb
```

*rs*レジスタの内容(ワードデータ)を指定のメモリに転送します。*rb*レジスタの内容がアクセスされるメモリアドレスとなります。

(2)拡張1

```
ext   imm13
ld.w  [%rb],%rs   ; memory address = rb + imm13
```

ext命令により、アドレッシングモードがディスプレースメント付きレジスタ間接アドレッシングに変わります。*rs*レジスタの内容を、*rb*レジスタの内容に13ビット即値*imm13*を加えたアドレスに転送します。*rb*レジスタの内容は変更されません。

(3)拡張2

```
ext   imm13       ; = imm26(25:13)
ext   imm13       ; = imm26(12:0)
ld.w  [%rb],%rs   ; memory address = rb + imm26
```

アドレッシングモードがディスプレースメント付きレジスタ間接アドレッシングに変わり、*rs*レジスタの内容を、*rb*レジスタの内容に26ビット即値*imm26*を加えたアドレスに転送します。*rb*レジスタの内容は変更されません。

(4)拡張3

```
ext   %rb2
ld.w  [%rb1],%rs  ; memory address = rb1 + rb2
```

3オペランドのレジスタ間接アドレッシングに変わります。*rs*レジスタの内容を、*rb1*レジスタの内容に*rb2*レジスタを加えたアドレスに転送します。*rb1*レジスタおよび*rb2*レジスタの内容は変更されません。

注　意

*rb*レジスタおよびディスプレースメントで指定されるメモリアドレスは、ワード境界(下位2ビット = 0)を示していることが必要です。それ以外のアドレスが指定されると、アドレス不整例外が発生します。

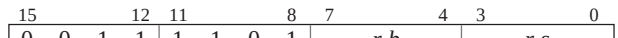
ld.w [*%rb*]+, *%rs*

機　能

ワードデータ転送

標準） W[*rb*] ← *rs*, *rb* ← *rb* + 4
拡張1) W[*rb*] ← *rs*, *rb* ← *rb* + *sign13*
拡張2) W[*rb*] ← *rs*, *rb* ← *rb* + *sign26*
拡張3) W[*rb1*] ← *rs*, *rb1* ← *rb1* + *rb2*

コード



0x3D__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

Src　レジスタ直接 *%rs* = %r0～%r15
Dst　ポストインクリメント付きレジスタ間接 *%rb* = %r0～%r15

CLK

1サイクル

説　明

(1)標準

```
ld.w  [%rb]+,%rs  ; memory address = rb, rb ← rb + 4
```

*rs*レジスタの内容(ワードデータ)を指定のメモリに転送します。*rb*レジスタの内容がアクセスされるメモリアドレスとなります。データ転送後、*rb*レジスタ内のアドレスをインクリメント(+4)します。

(2)拡張1

```
ext   imm13       ; = sign13
ld.w  [%rb]+,%rs  ; memory address = rb, rb ← rb + sign13
```

*rs*レジスタの内容を指定のメモリに転送します。*rb*レジスタの内容がアクセスされるメモリアドレスとなります。データ転送後、*rb*レジスタ内のアドレスに*sign13*を加え、*rb*に格納します。*sign13*の範囲は、-4,096～+4,092です。

(3)拡張2

```
ext   imm13       ; = sign26(25:13)
ext   imm13       ; = sign26(12:0)
ld.w  [%rb]+,%rs  ; memory address = rb, rb ← rb + sign26
```

*rs*レジスタの内容を指定のメモリに転送します。*rb*レジスタの内容がアクセスされるメモリアドレスとなります。データ転送後、*rb*レジスタ内のアドレスに*sign26*を加え、*rb*に格納します。*sign26*の範囲は、-33,554,432～+33,554,428です。

(4)拡張3

```
ext   %rb2
ld.w  [%rb1]+,%rs ; memory address = rb1, rb1 = rb1 + rb2
```

3オペランドのレジスタ間接アドレッシングに変わります。*rs*レジスタの内容を*rb1*レジスタが示すアドレスに転送します。データ転送後、*rb1*レジスタの内容に*rb2*レジスタの内容を加え、*rb1*レジスタに格納します。*rb2*の範囲は、-2,147,483,648～+2,147,483,644です。

注　意

*rb*レジスタおよびディスプレースメントで指定されるメモリアドレスは、ワード境界(下位2ビット = 0)を示していることが必要です。それ以外のアドレスが指定されると、アドレス不整例外が発生します。

ld.w [%dp + *imm6*], *%rs*

機　能

ワードデータ転送

標準） W[dp + *imm6*×4] ← *rs*
拡張1）W[dp + *imm19*] ← *rs*
拡張2）W[dp + *imm32*] ← *rs*
拡張3）不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	*imm6*						*rs*			

0xFC__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

Src　レジスタ直接 *%rs* = %r0～%r15
Dst　ディスプレースメント付きレジスタ間接

CLK

1サイクル

説　明

(1)標準

```
ld.w  [%dp + imm6],%rs ; memory address = dp + imm6×4
```

*rs*レジスタの内容(ワードデータ)を指定メモリに転送します。現在のDPの内容に6ビット即値*imm6*×4をディスプレースメントとして加算した値がアクセスされるメモリアドレスとなります。ディスプレースメントの下位2ビットは常に0となります。

(2)拡張1

```
ext   imm13             ; = imm19(18:6)
ld.w  [%dp + imm6],%rs  ; memory address = dp + imm19,
                        ; imm6 = imm19(5:0)
```

ext命令により、ディスプレースメントが19ビットに拡張されます。これにより、*rs*レジスタの内容を、DPの内容に19ビット即値*imm19*を加えたアドレスに転送します。*imm6*はワード境界(下位2ビット = 0)を指定してください。

(3)拡張2

```
ext   imm13             ; = imm32(31:19)
ext   imm13             ; = imm32(18:6)
ld.w  [%dp + imm6],%rs  ; memory address = dp + imm32,
                        ; imm6 = imm32(5:0)
```

2つのext命令により、ディスプレースメントが32ビットに拡張されます。これにより、*rs*レジスタの内容を、DPの内容に32ビット即値*imm32*を加えたアドレスに転送します。*imm6*はワード境界(下位2ビット = 0)を指定してください。

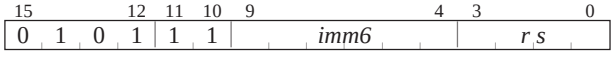
ld.w [%sp + *imm6*], *%rs*

機　能

ワードデータ転送

標準） W[sp + *imm6*×4] ← *rs*
拡張1) W[sp + *imm19*] ← *rs*
拡張2) W[sp + *imm32*] ← *rs*
拡張3) 不可

コード



0x5C__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

Src　レジスタ直接 *%rs* = %r0～%r15
Dst　ディスプレースメント付きレジスタ間接

CLK

1サイクル

説　明

(1)標準

```
ld.w  [%sp + imm6],%rs  ; memory address = sp + imm6×4
```

*rs*レジスタの内容(ワードデータ)を指定メモリに転送します。現在のSPの内容に6ビット即値*imm6*×4をディスプレースメントとして加算した値がアクセスされるメモリアドレスとなります。ディスプレースメントの下位2ビットは常に0となります。

(2)拡張1

```
ext   imm13             ; = imm19(18:6)
ld.w  [%sp + imm6],%rs  ; memory address = sp + imm19,
                        ; imm6 = imm19(5:0)
```

ext命令により、ディスプレースメントが19ビットに拡張されます。これにより、*rs*レジスタの内容を、SPの内容に19ビット即値*imm19*を加えたアドレスに転送します。*imm6*はワード境界(下位2ビット = 0)を指定してください。

(3)拡張2

```
ext   imm13             ; = imm32(31:19)
ext   imm13             ; = imm32(18:6)
ld.w  [%sp + imm6],%rs  ; memory address = sp + imm32,
                        ; imm6 = imm32(5:0)
```

2つのext命令により、ディスプレースメントが32ビットに拡張されます。これにより、*rs*レジスタの内容を、SPの内容に32ビット即値*imm32*を加えたアドレスに転送します。*imm6*はワード境界(下位2ビット = 0)を指定してください。

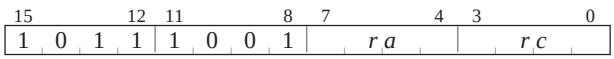
loop *%rc, %ra*

機　能

ループ実行

標準） pc + 2～*ra*を*rc* + 1回実行
拡張1）不可
拡張2）不可
拡張3）不可

コード



0xB9__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	↔	–	–	–	–	–	–	–	–	–	–	–

モード

Addr　レジスタ直接 *%ra* = %r0～%r15
Count　レジスタ直接 *%rc* = %r0～%r15

CLK

5サイクル

説　明

loop命令の次の命令から*ra*レジスタで指定される絶対アドレスまでの命令を、*rc*レジスタの値+1回実行することを指示します。loop命令が実行されるとLMフラグ（PSRのビット29）が1になりループ実行中であることを示します。
繰り返す命令の先頭アドレスがLSAレジスタに格納され、最終アドレスがLEAレジスタに格納されます。また、繰り返し回数はLCOレジスタに格納されます。
実行アドレスがLEAと一致するとLCOはデクリメント（-1）され、LSAをPCにロードしてループ範囲の先頭アドレスにジャンプします。
LCOの値が0になるとループ実行を終了し、LMフラグ（PSRのビット29）を0にリセットしてLEAアドレス以降の命令を通常に実行します。
ループ実行を中止して通常実行に戻るには、プログラムでLMフラグ（PSRのビット29）を0にする必要があります。
ループ実行回数*rc*は、1の指定で1回LEAからLSAに戻ります。つまりLSAからLEAの命令を2回実行します。

例

r0 = 2、r1 = endの場合

```
      loop  %r0,%r1        ; loop start
      ld.w  %r2,[%r3]+     ; copy data
end:  ld.w  [%r4]+,%r2     ;   [%r3] to [%r4]
```

3ワードのデータをコピーします。

注　意

ループ実行中にも割り込みを受け付けますので、割り込み処理ルーチンでloop命令を実行する際は、LSA、LEA、LCOをスタックなどに退避して、データを保護してください。
デバッグ例外、MMU例外処理ルーチンの中では、この命令を使用しないでください。

loop *%rc, imm4*

機　能

ループ実行

標準）pc + 2～pc + 2 + *imm4* × 2を*rc* + 1回実行
拡張1）不可
拡張2）不可
拡張3）不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	0	1	0	*imm4*				*rc*			

0xBA__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	↔	–	–	–	–	–	–	–	–	–	–	–

モード

Addr　符号なし即値 *imm4*
Count　レジスタ直接 *%rc* = %r0～%r15

CLK

5サイクル

説　明

loop命令の次の命令から符号なし即値*imm4*の値を2倍してPC+2を加えたアドレスまでの命令を、*rc*レジスタの値+1回実行することを指示します。loop命令が実行されるとLMフラグ（PSRのビット29）が1になりループ実行中であることを示します。
繰り返す命令の先頭アドレスがLSAレジスタに格納され、最終アドレスがLEAレジスタに格納されます。また、繰り返し回数はLCOレジスタに格納されます。
実行アドレスがLEAと一致するとLCOはデクリメント（-1）され、LSAをPCにロードしてループ範囲の先頭アドレスにジャンプします。
LCOの値が0になるとループ実行を終了し、LMフラグ（PSRのビット29）を0にリセットしてLEAアドレス以降の命令を通常に実行します。
ループ実行を中止して通常実行に戻るには、プログラムでLMフラグ（PSRのビット29）を0にする必要があります。
ループ実行回数*rc*は、1の指定で1回LEAからLSAに戻ります。つまりLSAからLEAの命令を2回実行します。

例

r1 = 3の場合

```
loop  %r1,1          ; loop start
ld.w  %r2,[%r3]+     ; copy data
ld.w  [%r4]+,%r2     ;   [%r3] to [%r4]
```

4ワードのデータをコピーします。

注　意

ループ実行中にも割り込みを受け付けますので、割り込み処理ルーチンでloop命令を実行する際は、LSA、LEA、LCOをスタックなどに退避して、データを保護してください。
デバッグ例外、MMU例外処理ルーチンの中では、この命令を使用しないでください。

loop *imm4(count)*, *imm4(addr)*

機　能

ループ実行

標準） pc + 2～pc + 2 + *imm4(addr)* × 2を*imm4(count)* + 1回実行
拡張1）不可
拡張2）不可
拡張3）不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	0	1	1	*imm4(addr)*				*imm4(count)*			

0xBB__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	↔	–	–	–	–	–	–	–	–	–	–	–

モード

Addr　符号なし即値 *imm4*
Count　符号なし即値 *imm4*

CLK

5サイクル

説　明

loop命令の次の命令から符号なし即値*imm4(addr)*の値を2倍してPC+2を加えたアドレスまでの命令を、符号なし即値*imm4(count)*の値+1回実行することを指示します。loop命令が実行されるとLMフラグ（PSRのビット29）が1になりループ実行中であることを示します。
繰り返す命令の先頭アドレスがLSAレジスタに格納され、最終アドレスがLEAレジスタに格納されます。また、繰り返し回数はLCOレジスタに格納されます。
実行アドレスがLEAと一致するとLCOはデクリメント（-1）され、LSAをPCにロードしてループ範囲の先頭アドレスにジャンプします。
LCOの値が0になるとループ実行を終了し、LMフラグ（PSRのビット29）を0にリセットしてLEAアドレス以降の命令を通常に実行します。
ループ実行を中止して通常実行に戻るには、プログラムでLMフラグ（PSRのビット29）を0にする必要があります。
ループ実行回数*rc*は、1の指定で1回LEAからLSAに戻ります。つまりLSAからLEAの命令を2回実行します。

例

```
loop  7,1            ; loop start
ld.w  %r2,[%r3]+     ; copy data
ld.w  [%r4]+,%r2     ;   [%r3] to [%r4]
```

8ワードのデータをコピーします。

注　意

ループ実行中にも割り込みを受け付けますので、割り込み処理ルーチンでloop命令を実行する際は、LSA、LEA、LCOをスタックなどに退避して、データを保護してください。
デバッグ例外、MMU例外処理ルーチンの中では、この命令を使用しないでください。

mac *%rs*

機　能　**積和演算**

標準）“{ahr, alr} ← {ahr, alr} + H[<*rs*+1>] × H[<*rs*+2>], <*rs*+1> ← <*rs*+1> + 2,
<*rs*+2> ← <*rs*+2> + 2” × *rs*回
拡張1）不可
拡張2）不可
拡張3）不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	1	0	*rs*				0	0	0	0

0xB2_0

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	↔	–	–	–	–	–

モード　レジスタ直接 *%rs* = %r0～%r15

CLK　2 + N × 2サイクル

説　明　mac *%rs*命令は“{AHR, ALR} ← {AHR, ALR} + H[<*rs*+1>]+ × H[<*rs*+2>]+”(64ビット+16ビット×16ビット)を*rs*レジスタで指定される回数分実行します。
*rs*レジスタは回数カウンタとして使用され、演算ごとにデクリメントされます。*rs*レジスタが0になると、mac命令は終了します。したがって、2^{32}-1回までの繰り返しが可能です。*rs*レジスタに0を設定してmac命令を実行しても、積和演算は行われず、AHR、ALRレジスタも変更されません。*rs*レジスタも0のままデクリメントされません。
<*rs*+1>と<*rs*+2>は*rs*レジスタに続く2つの汎用レジスタです。
例: *rs*にR0レジスタを指定　<*rs*+1> = R1レジスタ、<*rs*+2> = R2レジスタ
*rs*にR15レジスタを指定　<*rs*+1> = R0レジスタ、<*rs*+2> = R1レジスタ

積和演算は、これらのレジスタをベースアドレスとして指定されるメモリのハーフワードを符号付き16ビットデータとして演算します。ベースアドレスは1回の演算ごとにポストインクリメント(+2)されます。
演算結果は、AHRを上位32ビット、ALRを下位32ビットとする符号付き64ビットデータとして得られます。
積和演算中に演算結果が符号付き64ビットの範囲を越えると、オーバーフローとしてMOフラグ(PSRのビット7)が1にセットされます。この場合でも、*rs*レジスタに設定した回数を終了するまで演算は継続されます。MOフラグは、ソフトウェアによってリセットするまで1を保持します。mac命令の実行終了後にMOフラグを読み出すことで、演算結果が有効かどうかチェックできます。
mac命令の実行中は、繰り返しの途中であっても割り込みを受け付けます。割り込み処理ルーチンに分岐する際、スタックには実行中のmac命令のアドレスがリターンアドレスとしてセーブされます。したがって、割り込み処理ルーチンをreti命令で終了すると、中断していたmac命令の実行を再開します。ただし、その時点の*rs*レジスタの内容が残りのカウント数となりますので、割り込み処理ルーチン中で*rs*レジスタの内容が変更されると、当初設定した回数とは異なる結果となります。同様に、<*rs*+1>、<*rs*+2>レジスタの値が割り込み処理ルーチン中で変化すると、再開したmac命令は正しく実行されません。

例
```
mac   %r1   ; “{ahr, alr}← {ahr,alr} + H[r2]+ × H[r3]+”をr1回繰り返し実行
```

注　意　(1)<*rs*+1>および<*rs*+2>レジスタで指定されるメモリアドレスは、ハーフワード境界(最下位ビット = 0)を示していることが必要です。奇数アドレスが指定されると、アドレス不整例外が発生します。

(2)LCフラグ(PSRのビット17)が1の場合、ALRレジスタに書き込まれるデータは、ALR以外にR4レジスタにも書き込まれます。HCフラグ(PSRのビット16)が1のときは、AHRレジスタに書き込まれるデータは、AHR以外にR5レジスタにも書き込まれます。

mac.hw *%rs*

機　能

積和演算

標準）“{ahr, alr} ← {ahr, alr} + W[<*rs*+1>] × H[<*rs*+2>], <*rs*+1> ← <*rs*+1> + 4,
<*rs*+2> ← <*rs*+2> + 2” × *rs*回

拡張1）不可
拡張2）不可
拡張3）不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	0	1	0	1	*rs*			

0x015_

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	↔	–	–	–	–	–

モード

レジスタ直接 *%rs* = %r0～%r15

CLK

2 + N × 2サイクル

説　明

mac.hw *%rs*命令は“{AHR, ALR} ← {AHR, ALR} + W[<*rs*+1>]+ × H[<*rs*+2>]+”(64ビット+32ビット×16ビット)を*rs*レジスタで指定される回数分実行します。
*rs*レジスタは回数カウンタとして使用され、演算ごとにデクリメントされます。*rs*レジスタが0になると、mac.hw命令は終了します。したがって、2^{32}-1回までの繰り返しが可能です。*rs*レジスタに0を設定してmac.hw命令を実行しても、積和演算は行われず、AHR、ALRレジスタも変更されません。*rs*レジスタも0のままデクリメントされません。
<*rs*+1>と<*rs*+2>は*rs*レジスタに続く2つの汎用レジスタです。
例: *rs*にR0レジスタを指定　<*rs*+1> = R1レジスタ、<*rs*+2> = R2レジスタ
　　*rs*にR15レジスタを指定　<*rs*+1> = R0レジスタ、<*rs*+2> = R1レジスタ

積和演算は、これらのレジスタをベースアドレスとして指定されるメモリのデータを符号付きデータとして演算します。ベースアドレスは1回の演算ごとにポストインクリメント(それぞれ+4、+2)されます。
演算結果は、AHRを上位32ビット、ALRを下位32ビットとする符号付き64ビットデータとして得られます。
積和演算中に演算結果が符号付き64ビットの範囲を越えると、オーバーフローとしてMOフラグ(PSRのビット7)が1にセットされます。この場合でも、*rs*レジスタに設定した回数を終了するまで演算は継続されます。MOフラグは、ソフトウェアによってリセットするまで1を保持します。mac.hw命令の実行終了後にMOフラグを読み出すことで、演算結果が有効かどうかチェックできます。
mac.hw命令の実行中は、繰り返しの途中であっても割り込みを受け付けます。割り込み処理ルーチンに分岐する際、スタックには実行中のmac.hw命令のアドレスがリターンアドレスとしてセーブされます。したがって、割り込み処理ルーチンをreti命令で終了すると、中断していたmac.hw命令の実行を再開します。ただし、その時点の*rs*レジスタの内容が残りのカウント数となりますので、割り込み処理ルーチン中で*rs*レジスタの内容が変更されると、当初設定した回数とは異なる結果となります。同様に、<*rs*+1>、<*rs*+2>レジスタの値が割り込み処理ルーチン中で変化すると、再開したmac.hw命令は正しく実行されません。

例

```
mac.hw  %r1     ; “{ahr, alr}← {ahr,alr} + W[r2]+ × H[r3]+”をr1回繰り返し実行
```

注　意

(1)<*rs*+1>および<*rs*+2>レジスタで指定されるメモリアドレスは、それぞれワード境界(下位2ビット = 0)、ハーフワード境界(最下位ビット = 0)を示していることが必要です。それ以外のアドレスが指定されると、アドレス不整例外が発生します。

(2)LCフラグ(PSRのビット17)が1の場合、ALRレジスタに書き込まれるデータは、ALR以外にR4レジスタにも書き込まれます。HCフラグ(PSRのビット16)が1のときは、AHRレジスタに書き込まれるデータは、AHR以外にR5レジスタにも書き込まれます。

mac.w *%rs*

機　能

積和演算

標準）“{ahr, alr} ← {ahr, alr} + W[<*rs*+1>] × W[<*rs*+2>], <*rs*+1> ← <*rs*+1> + 4,
<*rs*+2> ← <*rs*+2> + 4” × *rs*回

拡張1）不可

拡張2）不可

拡張3）不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	0	0	0	1	*r*	*s*		

0x011_

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	↔	–	–	–	–	–

モード

レジスタ直接 *%rs* = %r0～%r15

CLK

3 + N × 2サイクル

説　明

mac.w *%rs*命令は“{AHR, ALR} ← {AHR, ALR} + W[<*rs*+1>]+ × W[<*rs*+2>]+”（64ビット+32ビット×32ビット）を*rs*レジスタで指定される回数分実行します。
*rs*レジスタは回数カウンタとして使用され、演算ごとにデクリメントされます。*rs*レジスタが0になると、mac.w命令は終了します。したがって、2^{32}-1回までの繰り返しが可能です。
*rs*レジスタに0を設定してmac.w命令を実行しても、積和演算は行われず、AHR、ALRレジスタも変更されません。*rs*レジスタも0のままデクリメントされません。
<*rs*+1>と<*rs*+2>は*rs*レジスタに続く2つの汎用レジスタです。
例: *rs*にR0レジスタを指定　<*rs*+1> = R1レジスタ、<*rs*+2> = R2レジスタ
　　*rs*にR15レジスタを指定　<*rs*+1> = R0レジスタ、<*rs*+2> = R1レジスタ

積和演算は、これらのレジスタをベースアドレスとして指定されるメモリのワードデータを符号付き32ビットデータとして演算します。ベースアドレスは1回の演算ごとにポストインクリメント（+4）されます。
演算結果は、AHRを上位32ビット、ALRを下位32ビットとする符号付き64ビットデータとして得られます。
積和演算中に演算結果が符号付き64ビットの範囲を越えると、オーバーフローとしてMOフラグ（PSRのビット7）が1にセットされます。この場合でも、*rs*レジスタに設定した回数を終了するまで演算は継続されます。MOフラグは、ソフトウェアによってリセットするまで1を保持します。mac.w命令の実行終了後にMOフラグを読み出すことで、演算結果が有効かどうかチェックできます。
mac.w命令の実行中は、繰り返しの途中であっても割り込みを受け付けます。割り込み処理ルーチンに分岐する際、スタックには実行中のmac.w命令のアドレスがリターンアドレスとしてセーブされます。したがって、割り込み処理ルーチンをreti命令で終了すると、中断していたmac.w命令の実行を再開します。ただし、その時点の*rs*レジスタの内容が残りのカウント数となりますので、割り込み処理ルーチン中で*rs*レジスタの内容が変更されると、当初設定した回数とは異なる結果となります。同様に、<*rs*+1>、<*rs*+2>レジスタの値が割り込み処理ルーチン中で変化すると、再開したmac.w命令は正しく実行されません。

例

```
mac.w  %r1 ;
```

“{ahr, alr}← {ahr,alr} + W[r2]+ × W[r3]+”をr1回繰り返し実行

注　意

（1）<*rs*+1>および<*rs*+2>レジスタで指定されるメモリアドレスは、ワード境界（下位2ビット = 0）を示していることが必要です。 それ以外のアドレスが指定されると、アドレス不整例外が発生します。

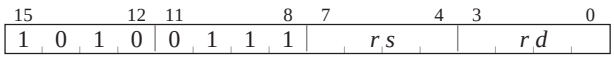
（2）LCフラグ（PSRのビット17）が1の場合、ALRレジスタに書き込まれるデータは、ALR以外にR4レジスタにも書き込まれます。HCフラグ（PSRのビット16）が1のときは、AHRレジスタに書き込まれるデータは、AHR以外にR5レジスタにも書き込まれます。

mac1.h *%rd, %rs*

機　能　**積和演算**

標準）{ahr, alr} ← {ahr, alr} + *rd*(15:0) × *rs*(15:0)
拡張1）不可
拡張2）不可
拡張3）不可

コード



0xA7__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	↔	–	–	–	–	–

モード　Src　レジスタ直接 *%rs* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK　1サイクル

説　明　*rd*レジスタの下位16ビットと*rs*レジスタの下位16ビットを乗算し、64ビットのAHRとALRレジスタペアに加算します。
演算結果が64ビットのAHRとALRレジスタペアをオーバーフローするとMOフラグ(PSRのビット7)が1にセットされます。MOフラグはソフトウェアでクリアされるまで1を保持します。

例

```
mac1.h  %r1,%r2      ; {ahr,alr} ← r1[15:0] × r2[15:0] + {ahr,alr}
```

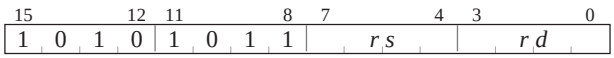
注　意　LCフラグ(PSRのビット17)が1の場合、ALRレジスタに書き込まれるデータは、ALR以外にR4レジスタにも書き込まれます。HCフラグ(PSRのビット16)が1のときは、AHRレジスタに書き込まれるデータは、AHR以外にR5レジスタにも書き込まれます。

mac1.hw *%rd*, *%rs*

機　能　**積和演算**

標準）{ahr, alr} ← {ahr, alr} + *rd*(31:0) × *rs*(15:0)
拡張1）不可
拡張2）不可
拡張3）不可

コード



0xAB__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	↔	–	–	–	–	–

モード　Src　レジスタ直接 *%rs* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK　2サイクル

説　明　*rd*レジスタの32ビットと*rs*レジスタの下位16ビットを乗算し、64ビットのAHRとALRレジスタペアに加算します。
演算結果が64ビットのAHRとALRレジスタペアをオーバーフローするとMOフラグ(PSRのビット7)が1にセットされます。MOフラグはソフトウェアでクリアされるまで1を保持します。

例

```
mac1.hw  %r1,%r2     ; {ahr,alr} ← r1[31:0] × r2[15:0] + {ahr,alr}
```

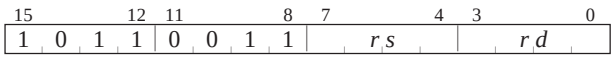
注　意　LCフラグ(PSRのビット17)が1の場合、ALRレジスタに書き込まれるデータは、ALR以外にR4レジスタにも書き込まれます。HCフラグ(PSRのビット16)が1のときは、AHRレジスタに書き込まれるデータは、AHR以外にR5レジスタにも書き込まれます。

mac1.w *%rd, %rs*

機　能　**積和演算**

標準）{ahr, alr} ← {ahr, alr} + *rd* × *rs*
拡張1）不可
拡張2）不可
拡張3）不可

コード



0xB3__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	↔	–	–	–	–	–

モード　Src　レジスタ直接 *%rs* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK　2サイクル

説　明　*rd*レジスタと*rs*レジスタを乗算し、64ビットのAHRとALRレジスタペアに加算します。
演算結果が64ビットのAHRとALRレジスタペアをオーバーフローするとMOフラグ（PSRのビット7）が1にセットされます。MOフラグはソフトウェアでクリアされるまで1を保持します。

例

```
mac1.w  %r1,%r2       ; {ahr,alr} ← r1 × r2 + {ahr,alr}
```

注　意　LCフラグ（PSRのビット17）が1の場合、ALRレジスタに書き込まれるデータは、ALR以外にR4レジスタにも書き込まれます。HCフラグ（PSRのビット16）が1のときは、AHRレジスタに書き込まれるデータは、AHR以外にR5レジスタにも書き込まれます。

macclr

機　能

AHR、ALRのクリア

標準）{ahr, alr} ← 0, MO ← 0
拡張1）不可
拡張2）不可
拡張3）不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0

0x0190

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	0	–	–	–	–	–

モード

–

CLK

1サイクル

説　明

(1)標準

AHRレジスタ、ALRレジスタおよびMOフラグ(PSRのビット7)をクリアします。
`macclr`命令は、`mac1`命令もしくは`mac`命令の実行前にAHR、ALR、MOフラグをクリアするために使用します。通常のフォワーディングとは異なる機構で動作するため、`macclr`命令後2命令以内にAHR、ALR、PSRをソースレジスタとする命令(`ld %rd,%ss`命令または`pushs %ss`命令)を記述した場合、正しく実行されない場合があります。

LCフラグ(PSRのビット17)が1のときは、ALRレジスタのほかにR4レジスタもクリアされます。HCフラグ(PSRのビット16)が1のときは、AHRレジスタのほかにR5レジスタもクリアされます。

(2)ディレイド命令

本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。

mirror *%rd, %rs*

機　能　**ミラー**

標準）*rd*(31:24) ← *rs*(24:31), *rd*(23:16) ← *rs*(16:23), *rd*(15:8) ← r*rs*(8:15), *rd*(7:0) ← *rs*(0:7)
拡張1）不可
拡張2）不可
拡張3）不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	1	1	0	*rs*				*rd*			

0x96__

フラグ

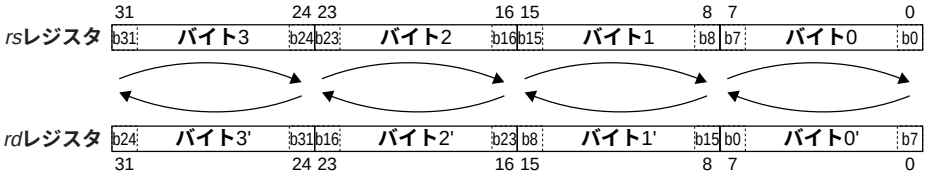
RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード　Src　レジスタ直接 *%rs* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK　1サイクル

説　明　(1)標準

*rs*レジスタのバイトデータごとにビットの上位と下位を入れ替え、結果を*rd*レジスタにロードします。



(2)ディレイド命令

本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。

例　r1が0x88442211の場合

```
mirror  %r0,%r1      ; r0 ← 0x11224488
```

32ビットデータのミラー（r1が0x44332211の場合）

```
swap    %r1,%r1      ; r1 ← 0x11223344
mirror  %r1,%r1      ; r1 ← 0x8844CC22
```

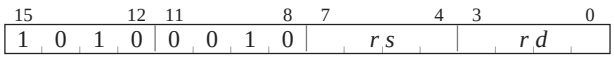
mlt.h *%rd*, *%rs*

機　能

符号付き16ビット × 16ビット乗算

標準） alr ← *rd*(15:0) × *rs*(15:0)
拡張1) 不可
拡張2) 不可
拡張3) 不可

コード



0xA2__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

Src　レジスタ直接 *%rs* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK

1サイクル

説　明

(1)標準
*rd*レジスタの下位16ビットと*rs*レジスタの下位16ビットを符号付きで乗算し、32ビットの演算結果をALRにロードします。

(2)ディレイド命令
本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。

例

```
mlt.h  %r0,%r1        ; alr ← r0(15:0) × r1(15:0)  符号付き乗算
```

注　意

LCフラグ(PSRのビット17)が1の場合、ALRレジスタに書き込まれるデータは、ALR以外にR4レジスタにも書き込まれます。

mlt.hw *%rd*, *%rs*

機　能　**符号付き32ビット × 16ビット乗算**

標準）{ahr, alr} ← *rd*(31:0) × *rs*(15:0)
拡張1）不可
拡張2）不可
拡張3）不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	1	1	*rs*				*rd*			

0xA3__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
−	−	−	−	−	−	−	−	−	−	−	−	−

モード　Src　レジスタ直接 *%rs* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK　2サイクル

説　明　*rd*レジスタの32ビットと*rs*レジスタの下位16ビットを乗算し、48ビットの演算結果を64ビットに符号拡張してAHRとALRレジスタペアにロードします。

例

```
mlt.hw  %r1,%r2      ; {ahr,alr} ← r1(31:0) × r2(15:0)  符号付き乗算
```

注　意　LCフラグ（PSRのビット17）が1の場合、ALRレジスタに書き込まれるデータは、ALR以外にR4レジスタにも書き込まれます。HCフラグ（PSRのビット16）が1のときは、AHRレジスタに書き込まれるデータは、AHR以外にR5レジスタにも書き込まれます。

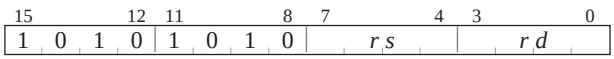
mlt.w *%rd*, *%rs*

機　能

符号付き32ビット × 32ビット乗算

標準）{ahr, alr} ← *rd* × *rs*
拡張1）不可
拡張2）不可
拡張3）不可

コード



0xAA__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

Src　レジスタ直接 *%rs* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK

2サイクル

説　明

*rd*レジスタと*rs*レジスタの内容を符号付きで乗算し、64ビットの演算結果をAHRとALRレジスタペアにロードします。

例

```
mlt.w  %r0,%r1        ; {ahr,alr} ← r0 × r1  符号付き乗算
```

注　意

LCフラグ(PSRのビット17)が1の場合、ALRレジスタに書き込まれるデータは、ALR以外にR4レジスタにも書き込まれます。HCフラグ(PSRのビット16)が1のときは、AHRレジスタに書き込まれるデータは、AHR以外にR5レジスタにも書き込まれます。

mltu.h *%rd, %rs*

機　能　**符号なし16ビット × 16ビット乗算**

標準）alr ← *rd*(15:0) × *rs*(15:0)
拡張1）不可
拡張2）不可
拡張3）不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	1	0	*rs*				*rd*			

0xA6__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード　Src　レジスタ直接 *%rs* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK　1サイクル

説　明　(1)標準
*rd*レジスタの下位16ビットと*rs*レジスタの下位16ビットを符号なしで乗算し、32ビットの演算結果をALRにロードします。

(2)ディレイド命令
本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。

例
```
mltu.h  %r0,%r1       ; alr ← r0(15:0) × r1(15:0)  符号なし乗算
```

注　意　LCフラグ(PSRのビット17)が1の場合、ALRレジスタに書き込まれるデータは、ALR以外にR4レジスタにも書き込まれます。

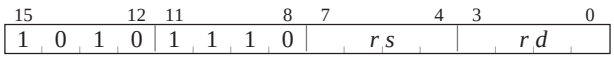
mltu.w *%rd, %rs*

機　能

符号なし32ビット × 32ビット乗算

標準）{ahr, alr} ← *rd* × *rs*
拡張1）不可
拡張2）不可
拡張3）不可

コード



0xAE__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

Src　レジスタ直接 *%rs* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK

2サイクル

説　明

*rd*レジスタと*rs*レジスタの内容を符号なしで乗算し、64ビットの演算結果をAHRとALRレジスタペアにロードします。

例

```
mltu.w  %r0,%r1       ; {ahr,alr} ← r0 × r1  符号なし乗算
```

注　意

LCフラグ(PSRのビット17)が1の場合、ALRレジスタに書き込まれるデータは、ALR以外にR4レジスタにも書き込まれます。HCフラグ(PSRのビット16)が1のときは、AHRレジスタに書き込まれるデータは、AHR以外にR5レジスタにも書き込まれます。

nop

機　能　**No Operation**
標準）なし
拡張1）不可
拡張2）不可
拡張3）不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

0x0000

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード　–

ＣＬＫ　1サイクル

説　明　何の動作もせずに1サイクルの時間を費やします。PCはインクリメント(+2)されます。

例
```
nop
nop          ; 2サイクルのウェイト
```

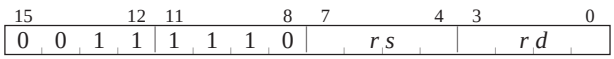
not *%rd, %rs*

機　能

論理否定

標準） *rd* ← !*rs*
拡張1) 不可
拡張2) 不可
拡張3) 不可

コード



0x3E__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	*1	↔	↔

モード

Src　レジスタ直接 *%rs* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK

1サイクル

説　明

(1)標準
*rs*レジスタの全ビットを反転し、*rd*レジスタにロードします。

(2)ディレイド命令
本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。

*1 PSRのOCフラグを1にセットして本命令を実行すると、Vフラグが0にクリアされます。他の論理演算系の命令についても同様に機能します。（and命令、or命令、xor命令。命令の機能は各命令の説明を参照してください。）

例

r1レジスタ = 0x55555555の場合

```
not  %r0,%r1    ; r0 = 0xAAAAAAAA
```

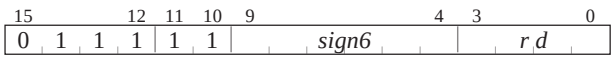
not *%rd, sign6*

機　能

論理否定

標準） *rd* ← !*sign6*
拡張1) *rd* ← !*sign19*
拡張2) *rd* ← !*sign32*
拡張3) 不可

コード



0x7C__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
−	−	−	−	−	−	−	−	−	−	*1	↔	↔

モード

Src　即値(符号付き)
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK

1サイクル

説　明

(1)標準

```
not  %rd,sign6     ; rd ← !sign6
```

符号拡張した6ビット即値*sign6*の全ビットを反転し、結果を*rd*レジスタにロードします。

(2)拡張1

```
ext  imm13         ; = sign19(18:6)
not  %rd,sign6     ; rd ← !sign19, sign6 = sign19(5:0)
```

符号拡張した19ビット即値*sign19*の全ビットを反転し、結果を*rd*レジスタにロードします。

(3)拡張2

```
ext  imm13         ; = sign32(31:19)
ext  imm13         ; = sign32(18:6)
not  %rd,sign6     ; rd ← !sign32, sign6 = sign32(5:0)
```

32ビット即値*sign32*の全ビットを反転し、結果を*rd*レジスタにロードします。

(4)ディレイド命令

本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。この場合はext命令による拡張は行えません。

*1 PSRのOCフラグを1にセットして本命令を実行すると、Vフラグが0にクリアされます。他の論理演算系の命令についても同様に機能します。(and命令、or命令、xor命令。命令の機能は各命令の説明を参照してください。)

例

(1)
```
not  %r0,0x1f      ; r0 = 0xffffffe0
```

(2)
```
ext  0x7ff
not  %r1,0x3f      ; r1 = 0xfffe0000
```

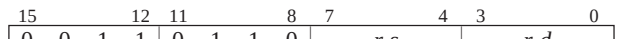
or *%rd, %rs*

機　能

論理和

標準） *rd* ← *rd* | *rs*
拡張1) *rd* ← *rs* | *imm13*
拡張2) *rd* ← *rs* | *imm26*
拡張3) *rd* ← *rs1* | *rs2*

コード



0x36__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	*1	↔	↔

モード

Src　レジスタ直接 *%rs* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK

1サイクル

説　明

(1)標準

```
or   %rd,%rs       ; rd ← rd | rs
```

*rs*レジスタの内容と*rd*レジスタの内容の論理和をとり、結果を*rd*レジスタにロードします。

(2)拡張1

```
ext  imm13
or   %rd,%rs       ; rd ← rs | imm13
```

*rs*レジスタの内容とゼロ拡張した13ビット即値*imm13*の論理和をとり、結果を*rd*レジスタにロードします。*rs*レジスタの内容は変更されません。

(3)拡張2

```
ext  imm13         ; = imm26(25:13)
ext  imm13         ; = imm26(12:0)
or   %rd,%rs       ; rd ← rs | imm26
```

*rs*レジスタの内容とゼロ拡張した26ビット即値*imm26*の論理和をとり、結果を*rd*レジスタにロードします。*rs*レジスタの内容は変更されません。

(4)拡張3

```
ext  %rs2
or   %rd,%rs1      ; rd ← rs1 | rs2
```

*rs1*レジスタの内容とext命令で指定されたレジスタ*rs2*の論理和をとり、結果を*rd*レジスタにロードします。*rs1*レジスタおよび*rs2*レジスタの内容は変更されません。

(5)ディレイド命令

本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。この場合はext命令による拡張は行えません。

---

*1 PSRのOCフラグを1にセットして本命令を実行すると、Vフラグが0にクリアされます。他の論理演算系の命令についても同様に機能します。（and命令、xor命令、not命令。命令の機能は各命令の説明を参照してください。）

例

```
(1)or   %r0,%r0       ; r0 = r0 | r0
(2)ext  0x1
   ext  0x1fff
   or   %r1,%r2       ; r1 = r2 | 0x00003fff
(3)ext  %r5
   or   %r3,%r4       ; r3 = r4 | r5
```

or *%rd, sign6*

機　能

論理和

標準） *rd* ← *rd* | *sign6*
拡張1) *rd* ← *rd* | *sign19*
拡張2) *rd* ← *rd* | *sign32*
拡張3) 不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	1	*sign6*						*rd*			

0x74__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
−	−	−	−	−	−	−	−	−	−	*1	↔	↔

モード

Src　即値(符号付き)
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK

1サイクル

説　明

(1)標準

```
or   %rd,sign6     ; rd ← rd | sign6
```

*rd*レジスタの内容と符号拡張した6ビット即値*sign6*の論理和をとり、結果を*rd*レジスタにロードします。

(2)拡張1

```
ext  imm13         ; = sign19(18:6)
or   %rd,sign6     ; rd ← rd | sign19, sign6 = sign19(5:0)
```

*rd*レジスタの内容と符号拡張した19ビット即値*sign19*の論理和をとり、結果を*rd*レジスタにロードします。

(3)拡張2

```
ext  imm13         ; = sign32(31:19)
ext  imm13         ; = sign32(18:6)
or   %rd,sign6     ; rd ← rd | sign32, sign6 = sign32(5:0)
```

*rd*レジスタの内容と32ビット即値*sign32*の論理和をとり、結果を*rd*レジスタにロードします。

(4)ディレイド命令
本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。この場合はext命令による拡張は行えません。

*1 PSRのOCフラグを1にセットして本命令を実行すると、Vフラグが0にクリアされます。他の論理演算系の命令についても同様に機能します。(and命令、xor命令、not命令。命令の機能は各命令の説明を参照してください。)

例

(1)
```
or   %r0,0x3e      ; r0 = r0 | 0xfffffffe
```

(2)
```
ext  0x7ff
or   %r1,0x3f      ; r1 = r1 | 0x0001ffff
```

pop *%rd*

機　能

ポップ

標準） *rd* ← W[sp], sp ← sp + 4
拡張1）不可
拡張2）不可
拡張3）不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	1	0	1	*r*	*d*		

0x005_

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

レジスタ直接 *%rd* = %r0～%r15

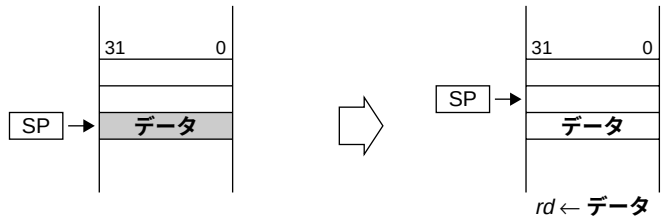
CLK

1サイクル

説　明

push命令でスタックに退避させた汎用レジスタのデータを復帰させます。
pop命令は、現在のSPが示すアドレスのワードデータを*rd*レジスタに復帰し、SPを1ワード（4バイト）分インクリメントします。

pop *%rd*実行時のスタックの動作



例

```
pop  %r3  ; r3 ← W[sp], sp ← sp + 4
```

popn *%rd*

機　能

ポップ

標準） “*rN* ← W[sp], sp ← sp + 4”を*rN* = r0～*rd*まで繰り返し実行
拡張1）不可
拡張2）不可
拡張3）不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	0	1	0	0	*rd*			

0x024_

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	↔	↔	–	–	–	–	–	–	–	–	–

モード

レジスタ直接 *%rd* = %r0～%r15

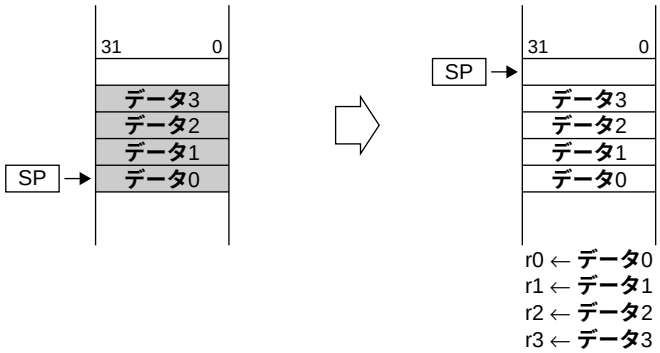
CLK

Nサイクル（N: 復帰するレジスタ数）

説　明

pushn命令でスタックに退避させた汎用レジスタのデータを各レジスタに復帰させます。現在のSPが示すアドレスのワードデータをr0レジスタに復帰し、SPを1ワード（4バイト）分インクリメントします。この動作を*rd*レジスタに一致するまで繰り返します。*rd*は対応するpushn命令で指定したレジスタと同一である必要があります。

popn *%rd*実行時のスタックの動作（*%rd* = %r3の場合）



例

```
popn   %r3  ;  r0, r1, r2, r3を復帰
```

注　意

popn命令を実行する際には、PMフラグ（PSRのビット28）が0であれば、命令語に含まれるレジスタフィールドのレジスタ番号が参照され、1であればPSRのRC[3:0]に格納されているレジスタ番号が参照されます。連続ポップ動作はこの参照されたレジスタに一致するまで行われます。

pops *%sd*

機　能

ポップ

標準） *sd* = psrまたはspの場合: *sd* ← W[sp], sp ← sp + 4
sd = alr～pcの場合: "*sN* ← W[sp], sp ← sp + 4"を*sN* = alr～*sd*まで繰り返し実行

拡張1) 不可
拡張2) 不可
拡張3) 不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	1	0	1	*s*	*s*	*s*	*d*

0x00D_

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	↔	↔	–	–	–	–	–	–	–	–	–

モード

レジスタ直接 *%sd* = `%psr`, `%sp`, `%alr`, `%ahr`, `%lco`, `%lsa`, `%lea`, `%sor`, `%ttbr`, `%dp`, `%idir`, `%dbbr`, `%usp`, `%ssp`, `%pc`

CLK

Nサイクル(N: 復帰するレジスタ数)

説　明

pushs命令でスタックに退避させた特殊レジスタのデータを各レジスタに復帰させます。

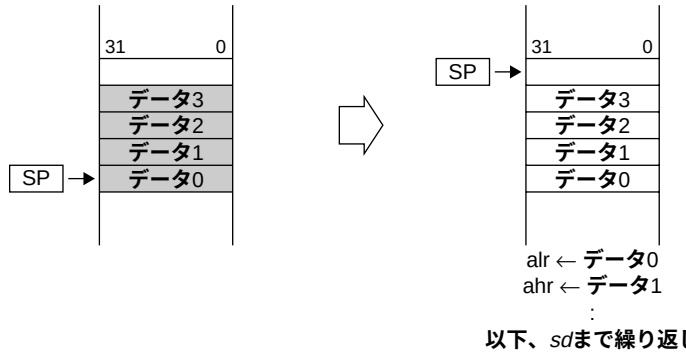
(1) *sd*レジスタがPSRまたはSPの場合

PSRレジスタまたはSPレジスタのみへの単独復帰を行います。
現在のSPが示すアドレスのワードデータを*sd*レジスタに復帰し、SPを1ワード(4バイト)分インクリメントします。

(2) *sd*レジスタがALR、AHR、LCO、LSA、LEA、SOR、DP、IDIR、DBBR、USP、SSPまたはPCの場合

現在のSPが示すアドレスのワードデータをALRレジスタに復帰し、SPを1ワード(4バイト)分インクリメントします。次に、存在しないレジスタも含めてレジスタ番号順に復帰対象となる特殊レジスタを変更します。この動作を*sd*レジスタに一致するまで繰り返します。*sd*は対応するpushs命令で指定したレジスタと同一である必要があります。

pops *%sd*実行時のスタックの動作(*%sd*が%alr～%pcの場合)



例

```
(1) pops  %sp     ; spの単独復帰
(2) pops  %lea    ; alr, ahr, lco, lsa, leaの順に復帰
```

注　意

(1)*sd*レジスタにIDIR、DBBR、USP、SSP、PCを指定した場合には、IDIR、DBBR、USP、SSPおよび存在しないレジスタである特殊レジスタ番号#12に対しても、メモリの読み出しおよびSPのインクリメントが一様に行われます。このとき、#12用に読み出したデータはどのレジスタにも反映されません。

(2)SP、USP、SSPレジスタに対してポップする場合、メモリの読み出しおよびSPのインクリメントが行われますが、このとき読み出した退避データがレジスタに書き戻されることはありません。

(3)PCレジスタに対してポップする場合、SPはインクリメントされますが、PCレジスタは変更されません。

(4)pops命令を実行する際には、PMフラグ(PSRのビット28)が0であれば、命令語に含まれるレジスタフィールドのレジスタ番号が参照され、1であればPSRのRC[3:0]に格納されているレジスタ番号が参照されます。連続ポップ動作はこの参照されたレジスタに一致するまで行われます。

psrclr *imm5*

機　能

PSRビットクリア

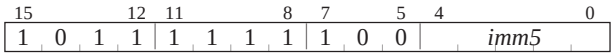
標準） psr ← psr & !*imm5*

拡張1) 不可

拡張2) 不可

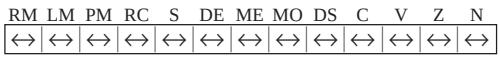
拡張3) 不可

コード



0xBF8_

フラグ



モード

即値

ＣＬＫ

4サイクル

説　明

(1)標準

*imm5*で指定されるPSRのビットを0にクリアします。

*imm5*の値はビット番号を表し、0→ビット0、 1→ビット1、 2→ビット2、... 30→ビット30、 31→ビット31のように指定されます。

(2)ディレイド命令

本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。

例

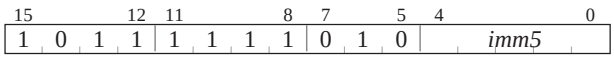
```
psrclr  2 ; V ← 0 （Vフラグをクリア）
```

psrset *imm5*

機　能　**PSRビットセット**

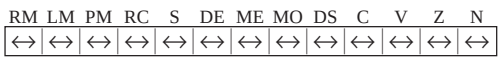
標準） psr ← psr | *imm5*
拡張1）不可
拡張2）不可
拡張3）不可

コード



0xBF4_

フラグ



モード　即値

ＣＬＫ　4サイクル

説　明　(1)標準

*imm5*で指定されるPSRのビットを1にセットします。
*imm5*の値はビット番号を表し、0→ビット0、 1→ビット1、 2→ビット2、... 30→ビット30、 31→ビット31のように指定されます。

(2)ディレイド命令

本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。

例　`psrset  12 ; SV ← 1` （SVフラグをセット）

push *%rs*

機　能

プッシュ
標準） sp ← sp - 4, W[sp] ← *rs*
拡張1) 不可
拡張2) 不可
拡張3) 不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	1	*rs*			

0x001_

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

レジスタ直接 *%rs* = %r0～%r15

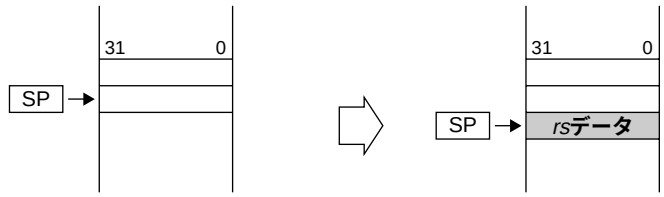
CLK

1サイクル

説　明

汎用レジスタのデータをスタックに退避させます。
push命令は、最初に現在のSPを1ワード(4バイト)分デクリメントし、*rs*レジスタの内容をそのアドレスに退避させます。

push *%rs*実行時のスタックの動作



例

```
push  %r3 ; sp ← sp - 4, W[sp] ← r3
```

pushn *%rs*

機　能　**プッシュ**

標準） “sp ← sp - 4, W[sp] ← *rN*”を*rN* = *rs*～r0まで繰り返し実行
拡張1）不可
拡張2）不可
拡張3）不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	0	0	0	0	*r*	*s*		

0x020_

フラグ

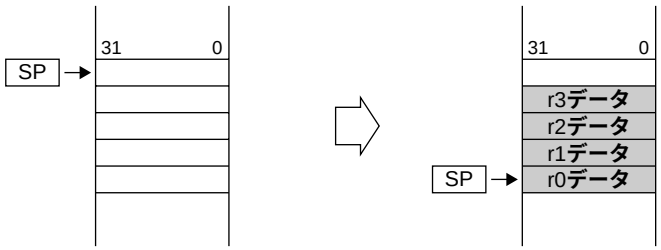
RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	↔	↔	–	–	–	–	–	–	–	–	–

モード　レジスタ直接 *%rs* = %r0～%r15

CLK　Nサイクル（N: 退避するレジスタ数）

説　明　汎用レジスタのデータをスタックに退避させます。
pushn命令は、最初に現在のSPを1ワード（4バイト）分デクリメントし、*rs*レジスタの内容をそのアドレスに退避させます。この動作をr0レジスタまで連続して行います。

pushn *%rs*実行時のスタックの動作（*%rs* = %r3の場合）



例　pushn　%r3 ;　r3, r2, r1, r0を退避

注　意　pushn命令を実行する際には、PMフラグ（PSRのビット28）が0であれば、命令語に含まれるレジスタフィールドのレジスタ番号が参照され、1であればPSRのRC[3:0]に格納されているレジスタ番号が参照されます。連続プッシュ動作はこの参照されたレジスタから行われます。

pushs *%ss*

機　能

プッシュ

標準） *ss* = psrまたはspの場合: sp ← sp - 4, W[sp] ← *ss*
ss = alr～pcの場合: “sp ← sp - 4, W[sp] ← *sN*”を*sN* = *ss*～alrまで繰り返し実行

拡張1) 不可
拡張2) 不可
拡張3) 不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	0	0	1		*ss*		

0x009_

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	↔	↔	–	–	–	–	–	–	–	–	–

モード

レジスタ直接 *%ss* = `%psr`, `%sp`, `%alr`, `%ahr`, `%lco`, `%lsa`, `%lea`, `%sor`, `%ttbr`, `%dp`, `%idir`, `%dbbr`, `%usp`, `%ssp`, `%pc`

CLK

Nサイクル(N: 退避するレジスタ数)

説　明

特殊レジスタのデータをスタックに退避させます。

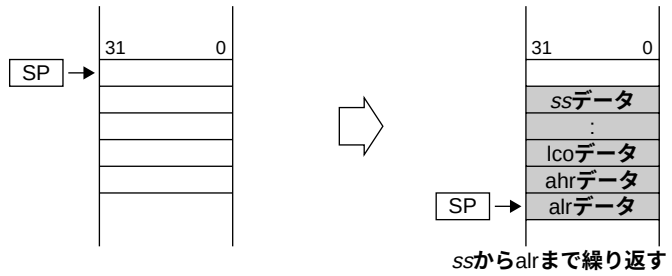
(1) *ss*レジスタがPSRまたはSPの場合

PSRレジスタまたはSPレジスタの単独退避を行います。
現在のSPを1ワード(4バイト)分デクリメントし、*ss*レジスタの内容をそのアドレスに退避させます。

(2) *ss*レジスタがALR、AHR、LCO、LSA、LEA、SOR、DP、IDIR、DBBR、USP、SSPまたはPCの場合

現在のSPを1ワード(4バイト)分デクリメントし、*ss*レジスタの内容をそのアドレスに退避させます。この動作を、存在しないレジスタも含めてレジスタ番号降順にALRレジスタまで繰り返します。

pushs *%ss*実行時のスタックの動作(*%ss* = %alr～%pcの場合)



例

(1) `pushs %sp ; sp`の単独退避

(2) `pushs %lea ; lea, lsa, lco, ahr, alr`の順に退避

注　意

(1) *ss*レジスタにIDIR、DBBR、USP、SSP、PCを指定した場合、存在しないレジスタである特殊レジスタ番号#12に対しても、メモリの書き込みおよびSPのデクリメントが一様に行われます。このとき、メモリに書き込まれるデータは不定となります。

(2) 本命令に対応する`pops` *%sd*命令は、SP、SSP、USPレジスタの退避データをメモリから読み込みますが、レジスタには書き戻しません。これらのレジスタに対してはSPのインクリメントだけが行われることになります。

(3) `pushs`命令を実行する際には、PMフラグ(PSRのビット28)が0であれば、命令語に含まれるレジスタフィールドのレジスタ番号が参照され、1であればPSRのRC[3:0]に格納されているレジスタ番号が参照されます。連続プッシュ動作はこの参照されたレジスタから行われます。

repeat *%rc*

機　能

リピート実行

標準） pc + 2の命令を*rc* + 1回実行
拡張1）不可
拡張2）不可
拡張3）不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	0	0	0	1	0	1	0	0	1	*rc*				0x029_

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
↔	–	–	–	–	–	–	–	–	–	–	–	–

モード　レジスタ直接 *%rc* = %r0～%r15

CLK　4サイクル

説　明

repeat命令の次の命令を*rc*レジスタの値+1回実行することを指示します。repeat命令が実行されるとRM（PSRのビット30）が1になり、リピート実行中であることを示します。
繰り返し回数はLCOレジスタに格納され、リピート対象命令のアドレス（repeat命令の次の命令）がLSAレジスタに格納されます。次のリピート対象命令を実行してPCとLSAレジスタの値が一致するとPCは固定されます。LCOはリピート対象命令の実行ごとにデクリメント（-1）され、LCOの値が0になるまで同じ命令が繰り返し実行されます。LCOが0になるとRM（PSRのビット30）が0となってリピート実行は終了します。
リピート回数*rc*は、1の指定でrepeat命令の次の命令を2回実行します。

例

r0 = 99の場合

```
repeat %r0              ; リピート開始
ld.w   [%r1]+,%r2       ; データのフィル
```

r1で指定されるアドレスから100ワード分の領域に、r2のデータを書き込みます。

注　意

リピート実行中にも割り込みを受け付けますので、割り込み処理ルーチンでrepeat命令を実行する際は、LSA、LEA、LCOをスタックなどに退避して、データを保護してください。
デバッグ例外、MMU例外処理ルーチンの中では、この命令を使用しないでください。

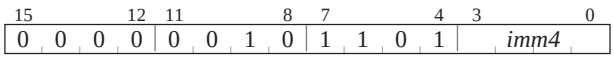
repeat *imm4*

機　能

リピート実行

標準）pc + 2の命令を*imm4* + 1回実行
拡張1）不可
拡張2）不可
拡張3）不可

コード



0x02D_

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
↔	–	–	–	–	–	–	–	–	–	–	–	–

モード　符号なし即値

CLK　4サイクル

説　明

repeat命令の次の命令を*imm4*の値+1回実行することを指示します。repeat命令が実行されるとRM(PSRのビット30)が1になり、リピート実行中であることを示します。
繰り返し回数はLCOレジスタに格納され、リピート対象命令のアドレス(repeat命令の次の命令)がLSAレジスタに格納されます。次のリピート対象命令を実行してPCとLSAレジスタの値が一致するとPCは固定されます。LCOはリピート対象命令の実行ごとにデクリメント(-1)され、LCOの値が0になるまで同じ命令が繰り返し実行されます。LCOが0になるとRM(PSRのビット30)が0となってリピート実行は終了します。
リピート回数*imm4*は、1の指定でrepeat命令の次の命令を2回実行します。

例

```
repeat 7              ; リピート開始
ld.w   [%r1]+,%r2     ; データのフィル
```

r1で指定されるアドレスから8ワード分の領域に、r2のデータを書き込みます。

注　意

リピート実行中にも割り込みを受け付けますので、割り込み処理ルーチンでrepeat命令を実行する際は、LSA、LEA、LCOをスタックなどに退避して、データを保護してください。
デバッグ例外、MMU例外処理ルーチンの中では、この命令を使用しないでください。

ret / ret.d

機　能

サブルーチンからのリターン

標準） pc ← W[sp], sp ← sp + 4
拡張1）不可
拡張2）不可
拡張3）不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	d	0	1	0	0	0	0	0	0

0x0640, 0x0740

d（ビット8）= 0の場合　ret
d（ビット8）= 1の場合　ret.d

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

–

CLK

ret　　5サイクル
ret.d　4サイクル

説　明

(1)標準

```
ret
```

call命令実行時にスタックに待避させたPC値（リターンアドレス）をPCに戻して、サブルーチンから呼び出し元のルーチンにリターンします。SPは1ワード分インクリメントされます。
サブルーチン内でスタック操作を行った場合は、ret命令実行前にSPがリターンアドレスを示すように戻しておく必要があります。

(2)ディレイド分岐（dビット = 1）

```
ret.d
```

ret.d命令では次の命令がディレイド命令となります。ディレイド命令は分岐前に実行されます。ret.d命令と次のディレイド命令の間はトラップがマスクされ、割り込みや例外は発生しません。

例

```
ret.d
add   %r0,%r1   ; リターン前に実行
```

注　意

ret.d命令（ディレイド分岐）を使用する場合、次の命令はディレイド命令として使用可能な命令に限られます。それ以外の命令を実行した場合、動作は不定となりますので注意してください。使用可能な命令については、Appendixの命令一覧表を参照してください。

retd

機　能

デバッグ処理ルーチンからのリターン

標準） r0 ← W[0xC (or 0x6000C)], pc ← W[0x8 (or 0x60008)]
拡張1）不可
拡張2）不可
拡張3）不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0

0x0440

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	0	–	–	–	–	–	–	–

モード

–

CLK

6サイクル

説　明

brk命令実行時にデバッグ用スタックに待避させたR0とPCの内容をそれぞれに戻して、デバッグ処理ルーチン（デバッグモード）からリターンします。
本命令はICE制御ソフト専用です。一般のプログラムでは使用しません。

例

```
retd        ; デバッグモードからのリターン
```

reti

機　能　**トラップ処理ルーチンからのリターン**

標準） pc ← W[ssp + 4], psr ← W[ssp], ssp ← ssp + 8
拡張1）不可
拡張2）不可
拡張3）不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	1	1	0	0	0	0	0	0

0x04C0

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
↔	↔	↔	↔	↔	↔	↔	↔	↔	↔	↔	↔	↔

モード　–

ＣＬＫ　6サイクル

説　明　例外や割り込み発生時にスタックに退避させたPCとPSRの内容をそれぞれに戻して、トラップ処理ルーチンからリターンします。SSPは2ワード分インクリメントされます。

例

```
reti        ; トラップ処理ルーチンからリターン
```

注　意　(1) reti命令実行時は、CPUの動作モードにかかわらず、常にSSPがスタックポインタとして使用されます。

(2) ITC(割り込みコントローラ)の割り込み要因フラグをリセット後(ld命令によるリセット)、CPUへの割り込み要求がネゲートされるまでに機種によっては数クロック必要になる場合があります。このため、割り込み要因フラグをリセットした命令(ld命令)の直後にreti命令が実行されるような場合、割り込み処理から正しく復帰できないことがあります。これを避けるため、reti命令より十分に前の時点で割り込み要因フラグをリセットしておくか、割り込み要因フラグのリセット後に要因フラグレジスタを読み出してからreti命令を実行してください。

retm

機　能

MMU例外処理ルーチンからのリターン

標準） r0 ← W[0x1C], pc ← W[0x18]
拡張1）不可
拡張2）不可
拡張3）不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	1	0	0	0	0	0	0

0x06C0

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	0	–	–	–	–	–	–

モード

–

CLK

6サイクル

説　明

MMU例外発生時にMMU例外用メモリ空間に退避させたR0、PCの内容をそれぞれ復帰して、MMU例外処理ルーチンからリターンします。

例

```
retm        ; MMU例外処理ルーチンからリターン
```

rl *%rd, %rs*

機　能

左方向ローテート

標準）*rd*の内容を*rs*の指定ビット（0～31）分、左にローテート
LSB ← MSB

拡張1）不可

拡張2）不可

拡張3）*rd* ← *rs1* << *rs2*

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	0	1	1	1	0	1	*rs*				*rd*				0x9D__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	↔	↔	↔	↔

C、Vフラグは SEフラグ（PSRのビット20）が1のときに変化

モード

Src　レジスタ直接 *%rs* = %r0～%r15

Dst　レジスタ直接 *%rd* = %r0～%r15

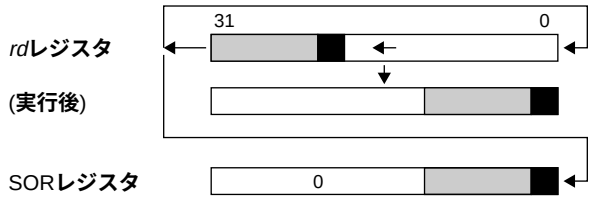
ＣＬＫ

1サイクル

説　明

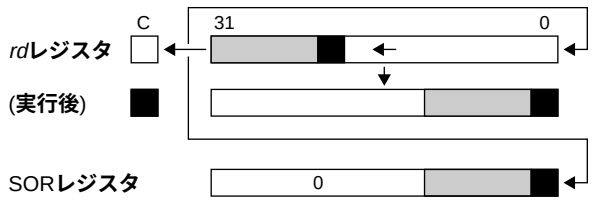
(1)標準モード（PSRのSEフラグが0の場合）

*rd*レジスタのビットを図のように回転させます。ビットのシフト量は*rs*レジスタの下位5ビットによって0～31まで指定できます。*rd*レジスタの最下位ビットには最上位ビットが入ります。*rd*レジスタからシフトアウトしたビットは、SORレジスタから読み出すことができます。



(2)アドバンスドモード（PSRのSEフラグが1の場合）

標準モードにCフラグを含めた動作になります。*rd*レジスタの最上位ビットからシフトアウトしたビットはPSRのCフラグと*rd*レジスタの最下位ビットに入ります。



Vフラグはシフト完了時のCフラグとZフラグの状態によって変化します。

C & N | !C & !N　→ V = 0

C ^ N　　→ V = 1

(3)拡張3

```
ext  %rs1
rl   %rd,%rs2
```

*rs1*レジスタのビットが*rs2*で指定されるビット数分左に回転します。本命令はビットを回転するソースレジスタが`ext %rs1`命令で与えられることを除き、標準またはアドバンスドモードと同じ動作となります。
*rd*レジスタの最下位ビットには、最上位ビットが入ります。
シフトアウトしたビットは、SORレジスタから読み出すことができます。

(4)ディレイド命令

本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。

rl *%rd, imm5*

機　能

左方向ローテート

標準） *rd*の内容を*imm5*の指定ビット(0～31)分、左にローテート
LSB ← MSB

拡張1) 不可

拡張2) 不可

拡張3) *rd* ← *rs* << *imm5*

コード

imm5(4) = 0の場合: 0～15ビットの左方向ローテート

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	0	1	1	1	0	0	*imm5*(3:0)				*r d*				0x9C__

imm5(4) = 1の場合: 16～31ビットの左方向ローテート

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	1	1	0	1	1	1	*imm5*(3:0)				*r d*				0x37__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	↔	↔	↔	↔

C、Vフラグは SEフラグ(PSRのビット20)が1のときに変化

モード

Src　即値(符号なし)

Dst　レジスタ直接 *%rd* = %r0～%r15

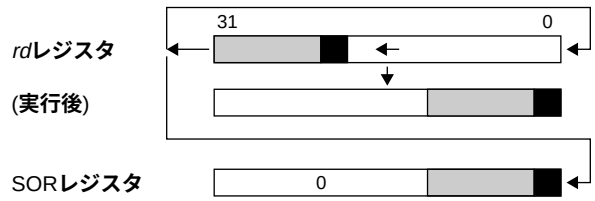
CLK

1サイクル

説　明

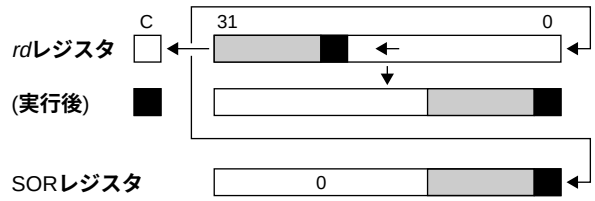
(1)標準モード(PSRのSEフラグが0の場合)

*rd*レジスタのビットを図のように回転させます。ビットのシフト量は5ビット即値*imm5*によって0～31まで指定できます。*rd*レジスタの最下位ビットには最上位ビットが入ります。*rd*レジスタからシフトアウトしたビットは、SORレジスタから読み出すことができます。



(2)アドバンスドモード(PSRのSEフラグが1の場合)

標準モードにCフラグを含めた動作になります。*rd*レジスタの最上位ビットからシフトアウトしたビットはPSRのCフラグと*rd*レジスタの最下位ビットに入ります。



Vフラグはシフト完了時のCフラグとZフラグの状態によって変化します。

C & N | !C & !N　→ V = 0

C ^ N　　→ V = 1

(3)拡張3

```
ext  %rs
rl   %rd,imm5
```

*rs*レジスタのビットが*imm5*で指定されるビット数分左に回転します。本命令はビットを回転するソースレジスタが`ext %rs`命令で与えられることを除き、標準またはアドバンスドモードと同じ動作となります。
*rd*レジスタの最下位ビットには、最上位ビットが入ります。
シフトアウトしたビットは、SORレジスタから読み出すことができます。

(4)ディレイド命令

本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。

rr %rd, %rs

機　能

右方向ローテート

標準）*rd*の内容を*rs*の指定ビット(0～31)分、右にローテート
MSB ← LSB
拡張1) 不可
拡張2) 不可
拡張3) *rd* ← *rs1* >> *rs2*

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	0	1	*rs*				*rd*			

0x99__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	↔	↔	↔	↔

C、Vフラグは SEフラグ(PSRのビット20)が1のときに変化

モード

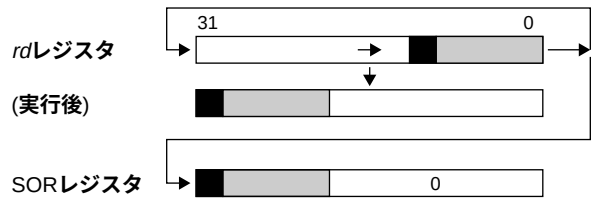
Src　レジスタ直接 *%rs* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

ＣＬＫ

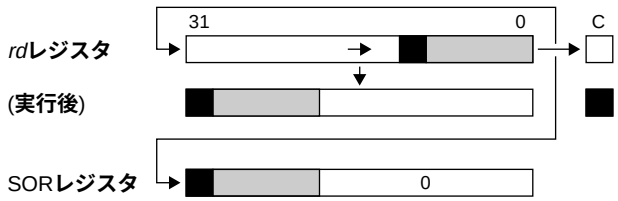
1サイクル

説　明

(1)標準モード(PSRのSEフラグが0の場合)
*rd*レジスタのビットを図のように回転させます。ビットのシフト量は*rs*レジスタの下位5ビットによって0～31まで指定できます。*rd*レジスタの最上位ビットには最下位ビットが入ります。*rd*レジスタからシフトアウトしたビットは、SORレジスタから読み出すことができます。



(2)アドバンスドモード(PSRのSEフラグが1の場合)
標準モードにCフラグを含めた動作になります。*rd*レジスタの最下位ビットからシフトアウトしたビットはPSRのCフラグと*rd*レジスタの最上位ビットに入ります。



Vフラグはシフト完了時のCフラグとZフラグの状態によって変化します。
C & N | !C & !N　→ V = 0
C ^ N　　→ V = 1

(3)拡張3

```
ext  %rs1
rr   %rd,%rs2
```

*rs1*レジスタのビットが*rs2*で指定されるビット数分右に回転します。本命令はビットを回転するソースレジスタが`ext %rs1`命令で与えられることを除き、標準またはアドバンスドモードと同じ動作となります。
*rd*レジスタの最上位ビットには、最下位ビットが入ります。
シフトアウトしたビットは、SORレジスタから読み出すことができます。

(4)ディレイド命令

本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。

rr *%rd, imm5*

機　能

右方向ローテート

標準） *rd*の内容を*imm5*の指定ビット(0～31)分、右にローテート
MSB ← LSB

拡張1) 不可

拡張2) 不可

拡張3) *rd* ← *rs* >> *imm5*

コード

imm5(4) = 0の場合: 0～15ビットの右方向ローテート

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	0	1	1	0	0	0	*imm5*(3:0)				*r d*				0x98__

imm5(4) = 1の場合: 16～31ビットの右方向ローテート

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	1	1	0	0	1	1	*imm5*(3:0)				*r d*				0x33__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	↔	↔	↔	↔

C、Vフラグは SEフラグ(PSRのビット20)が1のときに変化

モード

Src　即値(符号なし)

Dst　レジスタ直接 *%rd* = %r0～%r15

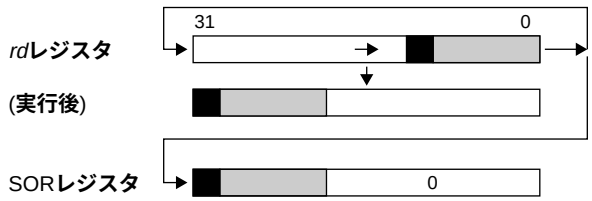
ＣＬＫ

1サイクル

説　明

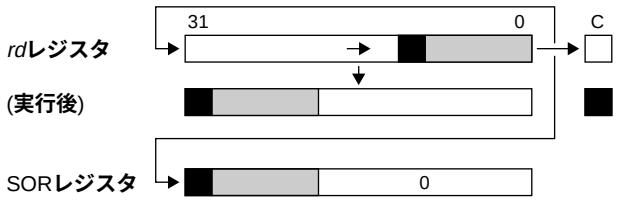
(1)標準モード(PSRのSEフラグが0の場合)

*rd*レジスタのビットを図のように回転させます。ビットのシフト量は5ビット即値*imm5*によって0～31まで指定できます。*rd*レジスタの最上位ビットには最下位ビットが入ります。*rd*レジスタからシフトアウトしたビットは、SORレジスタから読み出すことができます。



(2)アドバンスドモード(PSRのSEフラグが1の場合)

標準モードにCフラグを含めた動作になります。*rd*レジスタの最下位ビットからシフトアウトしたビットはPSRのCフラグと*rd*レジスタの最上位ビットに入ります。



Vフラグはシフト完了時のCフラグとZフラグの状態によって変化します。

C & N | !C & !N　→ V = 0

C ^ N　→ V = 1

(3)拡張3

```
ext  %rs
rr   %rd,imm5
```

*rs*レジスタのビットが*imm5*で指定されるビット数分右に回転します。本命令はビットを回転するソースレジスタが`ext %rs`命令で与えられることを除き、標準またはアドバンスドモードと同じ動作となります。
*rd*レジスタの最上位ビットには、最下位ビットが入ります。
シフトアウトしたビットは、SORレジスタから読み出すことができます。

(4)ディレイド命令

本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。

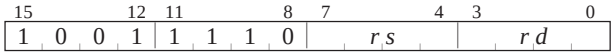
sat.b *%rd, %rs*

機　能　**符号付き飽和処理(8ビット)**

標準）*rd* ← *rs* if -128 ≤ *rs* ≤ +127;
rd ← 0xFFFFFF80 if *rs* < -128;
rd ← 0x0000007F if *rs* > +127

拡張1）不可
拡張2）不可
拡張3）不可

コード



0x9E__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	↔	–	–	–	–	–	–	–	–

モード　Src　レジスタ直接 *%rs* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

ＣＬＫ　1サイクル

説　明　(1)標準

符号付き8ビット飽和処理を行います。
*rs*レジスタの内容をテストして*rd*に結果を格納します。

*rs*の条件	処理	Sフラグ
-128 ≤ *rs* ≤ +127	*rs* → *rd*	–
rs < -128	0xFFFFFF80 → *rd*	1
rs > +127	0x0000007F → *rd*	1

(2)ディレイド命令

本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。

例　r1 = 0x555の場合

```
sat.b  %r2,%r1     ; 0x0000007F → r2
```

r1 = 0xFEDBA987の場合

```
sat.b  %r2,%r1     ; 0xFFFFFF80 → r2
```

sat.h *%rd, %rs*

機　能

符号付き飽和処理(16ビット)

標準) *rd* ← *rs* if -32,768 ≤ *rs* ≤ +32,767;
rd ← 0xFFFF8000 if *rs* < -32,768;
rd ← 0x00007FFF if *rs* > +32,767

拡張1) 不可
拡張2) 不可
拡張3) 不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	1	1	0	1	1	0	*r*	*s*			*r*	*d*			0xB6__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	↔	–	–	–	–	–	–	–	–

モード

Src　レジスタ直接 *%rs* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK

1サイクル

説　明

(1)標準
符号付き16ビット飽和処理を行います。
*rs*レジスタの内容をテストして*rd*に結果を格納します。

*rs*の条件	処理	Sフラグ
-32,768 ≤ *rs* ≤ +32,767	*rs* → *rd*	–
rs < -32,768	0xFFFF8000 → *rd*	1
rs > +32,767	0x00007FFF → *rd*	1

(2)ディレイド命令
本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。

例

r1 = 0x555の場合

```
sat.h  %r2,%r1     ; 0x00000555 → r2
```

r1 = 0xFEDBA987の場合

```
sat.h  %r2,%r1     ; 0xFFFF8000 → r2
```

sat.ub *%rd, %rs*

機　能　**符号なし飽和処理(8ビット)**

標準) *rd* ← *rs* if *rs* ≤ 255;
rd ← 0x000000FF if *rs* > 255
拡張1) 不可
拡張2) 不可
拡張3) 不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	1	1	1	*rs*				*rd*			

0x9F__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	↔	–	–	–	–	–	–	–	–

モード　Src　レジスタ直接 *%rs* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK　1サイクル

説　明　(1)標準
符号なし8ビット飽和処理を行います。
*rs*レジスタの内容をテストして*rd*に結果を格納します。

*rs*の条件	処理	Sフラグ
rs ≤ 255	*rs* → *rd*	–
rs > 255	0x000000FF → *rd*	1

(2)ディレイド命令
本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。

例　r1 = 0x555の場合

```
sat.ub  %r2,%r1   ; 0x000000FF → r2
```

r1 = 0xFEDBA987の場合

```
sat.ub  %r2,%r1   ; 0x000000FF → r2
```

sat.uh *%rd, %rs*

機　能

符号なし飽和処理(16ビット)

標準) *rd* ← *rs* if *rs* ≤ 65,535;
rd ← 0x0000FFFF if *rs* > 65,535
拡張1) 不可
拡張2) 不可
拡張3) 不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	1	1	0	1	1	1	*rs*				*rd*				0xB7__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	↔	–	–	–	–	–	–	–	–

モード

Src　レジスタ直接 *%rs* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK

1サイクル

説　明

(1)標準
符号なし16ビット飽和処理を行います。
*rs*レジスタの内容をテストして*rd*に結果を格納します。

*rs*の条件	処理	Sフラグ
rs ≤ 65,535	*rs* → *rd*	–
rs > 65,535	0x0000FFFF → *rd*	1

(2)ディレイド命令
本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。

例

r1 = 0x555の場合

```
sat.uh  %r2,%r1   ; 0x00000555 → r2
```

r1 = 0xFEDBA987の場合

```
sat.uh  %r2,%r1   ; 0x0000FFFF → r2
```

sat.uw *%rd, %rs*

機　能　**符号なし飽和処理(32ビット)**

標準）*rd* ← *rs* if C = 0;
　　rd ← 0xFFFFFFFF if C = 1
拡張1) 不可
拡張2) 不可
拡張3) 不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	1	1	1	1	1	0	*rs*				*rd*				0xBE__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	↔	–	–	–	–	–	–	–	–

モード　Src　レジスタ直接 *%rs* = %r0～%r15
　　Dst　レジスタ直接 *%rd* = %r0～%r15

CLK　1サイクル

説　明　(1)標準
　　符号なし32ビット飽和処理を行います。
　　*rs*レジスタの内容をテストして*rd*に結果を格納します。

フラグ条件	処理	Sフラグ
C = 0	*rs* → *rd*	–
C = 1	0xFFFFFFFF → *rd*	1

(2)ディレイド命令
　　本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。

例　r1 = 0x555、C = 0の場合

```
sat.uw  %r2,%r1   ; 0x00000555 → r2
```

0xFEDBA987、C = 1の場合

```
sat.uw  %r2,%r1   ; 0xFFFFFFFF → r2
```

sat.w *%rd, %rs*

機　能

符号付き飽和処理(32ビット)

標準) *rd* ← *rs* if V = 0;
rd ← 0x80000000 if V = 1 & N = 0;
rd ← 0xFFFFFFFF if V = 1 & N = 1

拡張1) 不可
拡張2) 不可
拡張3) 不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	0	1	*rs*				*rd*			

0xBD__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	↔	–	–	–	–	–	–	–	–

モード

Src　レジスタ直接 *%rs* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK

1サイクル

説　明

(1)標準
符号付き32ビット飽和処理を行います。
*rs*レジスタの内容をテストして*rd*に結果を格納します。

フラグ条件	処理	Sフラグ
V = 0	*rs* → *rd*	–
V = 1 & N = 0	0x80000000 → *rd*	1
V = 1 & N = 1	0xFFFFFFFF → *rd*	1

(2)ディレイド命令
本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。

例

V = 1、N = 1の場合

```
sat.w  %r2,%r1     ; 0x7FFFFFFF → r2
```

V = 1、N = 0の場合

```
sat.w  %r2,%r1     ; 0x80000000 → r2
```

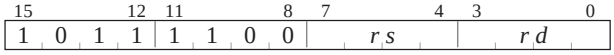
sbc *%rd, %rs*

機　能

ボロー付き減算

標準） *rd* ← *rd* - *rs* - C
拡張1）不可
拡張2）不可
拡張3） *rd* ← *rs1* - *rs2* - C（"*op, imm2*"使用可）

コード



0xBC__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	↔	↔	↔	↔

モード

Src　レジスタ直接 *%rs* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK

1サイクル

説　明

(1)標準

```
sbc  %rd,%rs       ; rd ← rd - rs - C
```

*rd*レジスタから*rs*レジスタの内容とC(キャリー)フラグの内容を減算します。

(2)拡張3

```
ext  %rs2,op,imm2 ; op = sra, srl, sla, imm2 = 0-3
sbc  %rd,%rs1     ; rd ← (rs1 - rs2 - C) op imm2
```

*rs1*レジスタからext命令で指定されたレジスタ*rs2*とC(キャリー)フラグの内容を減算し、さらに*op*で指示されるシフトを*imm2*のビット数分行い、結果を*rd*レジスタにロードします。*rs1*レジスタ、*rs2*レジスタの内容は変更されません。

(3)ディレイド命令

本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。

(4)ポストシフト

ポストシフト付き拡張命令の直後に記述することによって、本命令の実行結果が最大3ビットシフトされます。シフト動作は、sra, srl, sllの各命令と同じです。ただし、シフトアウトレジスタSORを使用しませんのでSORは変化しません。また、C、V、Z、Nフラグはsbc命令の結果のみによって変化し、シフト動作の影響は受けません。

例

(1)
```
sbc  %r0,%r1       ; r0 = r0 - r1 - C
```

(2)64ビットデータの減算

データ1 = {r2, r1}, データ2 = {r4, r3}, 減算結果 = {r2, r1}

```
sub  %r1,%r3       ; 下位ワードの減算
sbc  %r2,%r4       ; 上位ワードの減算
```

(3)
```
ext  %r2,srl,1
sbc  %r3,%r1       ; r3 = (r1 - r2 - C) >> 1
```

scan0 *%rd, %rs*

機　能

0のビットスキャン

標準）*rd* ← *rs*(31:0)の0のビットオフセット
拡張1）不可
拡張2）不可
拡張3）不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	0	1	0	*rs*				*rd*			

0x8A__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	↔	0	↔	0

モード

Src　レジスタ直接 *%rs* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK

1サイクル

説　明

(1)標準モード(PSRのSWフラグが0の場合)
SWフラグ(PSRのビット22)を0にしてscan0命令を実行すると、S1C33 STDコアCPUと同様の処理を行います。*rs*レジスタの最上位バイト(ビット31～24)をスキャンし、0のビットが見つかると、その位置(MSBからのオフセット)を*rd*レジスタにロードします。MSBが0の場合、*rd*レジスタには0がロードされ、Zフラグがセットされます。*rs*レジスタの最上位バイト中に0が見つからなかった場合、*rd*レジスタには0x00000008がロードされ、Cフラグがセットされます。

*rs*レジスタの上位8ビット (2進数)	*rd*レジスタ (16進数)	フラグ			
		C	V	Z	N
0xxx xxxx	0x00000000	0	0	1	0
10xx xxxx	0x00000001	0	0	0	0
110x xxxx	0x00000002	0	0	0	0
1110 xxxx	0x00000003	0	0	0	0
1111 0xxx	0x00000004	0	0	0	0
1111 10xx	0x00000005	0	0	0	0
1111 110x	0x00000006	0	0	0	0
1111 1110	0x00000007	0	0	0	0
1111 1111	0x00000008	1	0	0	0

(2)アドバンスドモード(PSRのSWフラグが1の場合)
SWフラグ(PSRのビット22)を1にしてscan0命令を実行すると、32ビットscan0命令となります。*rs*レジスタの(ビット31～0)をスキャンし、0のビットが見つかると、その位置(MSBからのオフセット)を*rd*レジスタにロードします。MSBが0の場合、*rd*レジスタには0がロードされ、Zフラグがセットされます。*rs*レジスタに0が見つからなかった場合、*rd*レジスタには0x00000020がロードされ、Cフラグがセットされます。

*rs*レジスタ (2進数)	*rd*レジスタ (16進数)	フラグ			
		C	V	Z	N
0xxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx	0x00000000	0	0	1	0
10xx xxxx xxxx xxxx xxxx xxxx xxxx xxxx	0x00000001	0	0	0	0
110x xxxx xxxx xxxx xxxx xxxx xxxx xxxx	0x00000002	0	0	0	0
1110 xxxx xxxx xxxx xxxx xxxx xxxx xxxx	0x00000003	0	0	0	0
1111 0xxx xxxx xxxx xxxx xxxx xxxx xxxx	0x00000004	0	0	0	0
1111 10xx xxxx xxxx xxxx xxxx xxxx xxxx	0x00000005	0	0	0	0
1111 110x xxxx xxxx xxxx xxxx xxxx xxxx	0x00000006	0	0	0	0
1111 1110 xxxx xxxx xxxx xxxx xxxx xxxx	0x00000007	0	0	0	0
1111 1111 0xxx xxxx xxxx xxxx xxxx xxxx	0x00000008	0	0	0	0
1111 1111 10xx xxxx xxxx xxxx xxxx xxxx	0x00000009	0	0	0	0
:	:	:	:	:	:
1111 1111 1111 1111 1111 1111 1111 110x	0x0000001e	0	0	0	0
1111 1111 1111 1111 1111 1111 1111 1110	0x0000001f	0	0	0	0
1111 1111 1111 1111 1111 1111 1111 1111	0x00000020	1	0	0	0

(3)ディレイド命令

本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。

例

```
psrset  22          ; psr(22) ← 1
scan0   %r1,%r0     ; r0(31:0)の0スキャン
```

scan1 *%rd, %rs*

機　能

1のビットスキャン

標準）*rd* ← *rs*(31:0)の1のビットオフセット
拡張1）不可
拡張2）不可
拡張3）不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	1	1	0	*rs*				*rd*			

0x8E__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	↔	0	↔	0

モード

Src　レジスタ直接 *%rs* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK

1サイクル

説　明

(1)標準モード(PSRのSWフラグが0の場合)
SWフラグ(PSRのビット22)を0にしてscan1命令を実行すると、S1C33 STDコアCPUと同様の処理を行います。*rs*レジスタの最上位バイト(ビット31～24)をスキャンし、1のビットが見つかると、その位置(MSBからのオフセット)を*rd*レジスタにロードします。MSBが1の場合、*rd*レジスタには0がロードされ、Zフラグがセットされます。*rs*レジスタの最上位バイト中に1が見つからなかった場合、*rd*レジスタには0x00000008がロードされ、Cフラグがセットされます。

*rs*レジスタの上位8ビット (2進数)	*rd*レジスタ (16進数)	フラグ			
		C	V	Z	N
1xxx xxxx	0x00000000	0	0	1	0
01xx xxxx	0x00000001	0	0	0	0
001x xxxx	0x00000002	0	0	0	0
0001 xxxx	0x00000003	0	0	0	0
0000 1xxx	0x00000004	0	0	0	0
0000 01xx	0x00000005	0	0	0	0
0000 001x	0x00000006	0	0	0	0
0000 0001	0x00000007	0	0	0	0
0000 0000	0x00000008	1	0	0	0

(2)アドバンスドモード(PSRのSWフラグが1の場合)
SWフラグ(PSRのビット22)を1にしてscan1命令を実行すると、32ビットscan1命令となります。*rs*レジスタの(ビット31～0)をスキャンし、1のビットが見つかると、その位置(MSBからのオフセット)を*rd*レジスタにロードします。MSBが1の場合、*rd*レジスタには0がロードされ、Zフラグがセットされます。*rs*レジスタに1が見つからなかった場合、*rd*レジスタには0x00000020がロードされ、Cフラグがセットされます。

*rs*レジスタ (2進数)	*rd*レジスタ (16進数)	フラグ			
		C	V	Z	N
1xxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx	0x00000000	0	0	1	0
01xx xxxx xxxx xxxx xxxx xxxx xxxx xxxx	0x00000001	0	0	0	0
001x xxxx xxxx xxxx xxxx xxxx xxxx xxxx	0x00000002	0	0	0	0
0001 xxxx xxxx xxxx xxxx xxxx xxxx xxxx	0x00000003	0	0	0	0
0000 1xxx xxxx xxxx xxxx xxxx xxxx xxxx	0x00000004	0	0	0	0
0000 01xx xxxx xxxx xxxx xxxx xxxx xxxx	0x00000005	0	0	0	0
0000 001x xxxx xxxx xxxx xxxx xxxx xxxx	0x00000006	0	0	0	0
0000 0001 xxxx xxxx xxxx xxxx xxxx xxxx	0x00000007	0	0	0	0
0000 0000 1xxx xxxx xxxx xxxx xxxx xxxx	0x00000008	0	0	0	0
0000 0000 01xx xxxx xxxx xxxx xxxx xxxx	0x00000009	0	0	0	0
:	:	:	:	:	:
0000 0000 0000 0000 0000 0000 0000 001x	0x0000001e	0	0	0	0
0000 0000 0000 0000 0000 0000 0000 0001	0x0000001f	0	0	0	0
0000 0000 0000 0000 0000 0000 0000 0000	0x00000020	1	0	0	0

(3)ディレイド命令

本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。

例

```
psrset  22          ; psr(22) ← 1
scan1   %r1,%r0     ; r0(31:0)の1スキャン
```

sla *%rd, %rs*

機　能

左方向算術シフト

標準）*rd*の内容を*rs*の指定ビット(0～31)分、左にシフト

LSB ← 0

拡張1) 不可

拡張2) 不可

拡張3) *rd* ← *rs1* << *rs2*

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	1	0	1	*rs*				*rd*			

0x95__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	↔	↔	↔	↔

C、Vフラグは SEフラグ(PSRのビット20)が1のときに変化

モード

Src　レジスタ直接 *%rs* = %r0～%r15

Dst　レジスタ直接 *%rd* = %r0～%r15

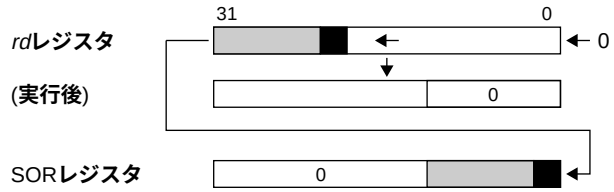
CLK

1サイクル

説　明

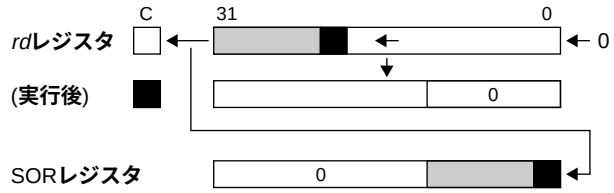
(1)標準モード(PSRのSEフラグが0の場合)

*rd*レジスタのビットを図のようにシフトさせます。ビットのシフト量は*rs*レジスタの下位5ビットによって0～31まで指定できます。*rd*レジスタの最下位ビットには0が入ります。*rd*レジスタからシフトアウトしたビットは、SORレジスタから読み出すことができます。



(2)アドバンスドモード(PSRのSEフラグが1の場合)

シフト動作は(1)の標準モードと同じですが、*rd*レジスタの最上位ビットからシフトアウトしたビットがPSRのCフラグに入ります。



Vフラグはシフト完了時のCフラグとZフラグの状態によって変化します。

C & N | !C & !N　→ V = 0

C ^ N　　→ V = 1

(3)拡張3

```
ext  %rs1
sla  %rd,%rs2
```

*rs1*レジスタのビットを*rs2*で指定されるビット数分左にシフトします。本命令はビットをシフトするソースレジスタが`ext %rs1`命令で与えられることを除き、標準またはアドバンスドモードと同じ動作となります。
*rd*レジスタの最下位ビットには0が入ります。
シフトアウトしたビットは、SORレジスタから読み出すことができます。

(4)ディレイド命令

本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。

sla *%rd, imm5*

機　能

左方向算術シフト

標準） *rd*の内容を*imm5*の指定ビット(0～31)分、左にシフト
LSB ← 0
拡張1) 不可
拡張2) 不可
拡張3) *rd* ← *rs* << *imm5*

コード

imm5(4) = 0の場合: 0～15ビットの左方向算術シフト

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	0	1	0	1	0	0	*imm5*(3:0)				*r d*				0x94__

imm5(4) = 1の場合: 16～31ビットの左方向算術シフト

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	1	0	1	1	1	1	*imm5*(3:0)				*r d*				0x2F__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	↔	↔	↔	↔

C、Vフラグは SEフラグ(PSRのビット20)が1のときに変化

モード

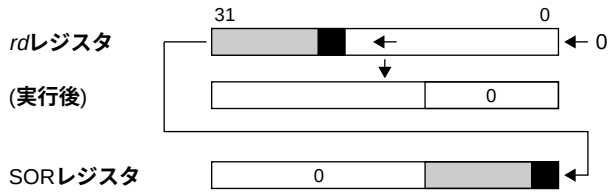
Src　即値(符号なし)
Dst　レジスタ直接 *%rd* = %r0～%r15

ＣＬＫ

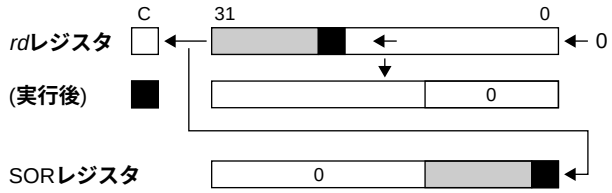
1サイクル

説　明

(1)標準モード(PSRのSEフラグが0の場合)
*rd*レジスタのビットを図のようにシフトさせます。ビットのシフト量は5ビット即値*imm5*によって0～31まで指定できます。*rd*レジスタの最下位ビットには0が入ります。*rd*レジスタからシフトアウトしたビットは、SORレジスタから読み出すことができます。



(2)アドバンスドモード(PSRのSEフラグが1の場合)
シフト動作は(1)の標準モードと同じですが、*rd*レジスタの最上位ビットからシフトアウトしたビットがPSRのCフラグに入ります。



Vフラグはシフト完了時のCフラグとZフラグの状態によって変化します。
C & N | !C & !N → V = 0
C ^ N → V = 1

(3)拡張3

```
ext  %rs
sla  %rd,imm5
```

*rs*レジスタのビットを*imm5*で指定されるビット数分左にシフトします。本命令はビットをシフトするソースレジスタが`ext %rs`命令で与えられることを除き、標準またはアドバンスドモードと同じ動作となります。
*rd*レジスタの最下位ビットには0が入ります。
シフトアウトしたビットは、SORレジスタから読み出すことができます。

(4)ディレイド命令

本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。

sll *%rd, %rs*

機　能

左方向論理シフト

標準） *rd*の内容を*rs*の指定ビット(0～31)分、左にシフト

LSB ← 0

拡張1) 不可

拡張2) 不可

拡張3) *rd* ← *rs1* << *rs2*

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	1	0	1	*rs*				*rd*			

0x8D__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	↔	↔	↔	↔

C、Vフラグは SEフラグ(PSRのビット20)が1のときに変化

モード

Src　レジスタ直接 *%rs* = %r0～%r15

Dst　レジスタ直接 *%rd* = %r0～%r15

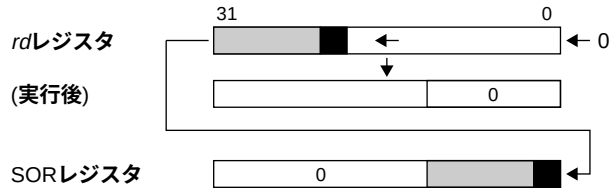
CLK

1サイクル

説　明

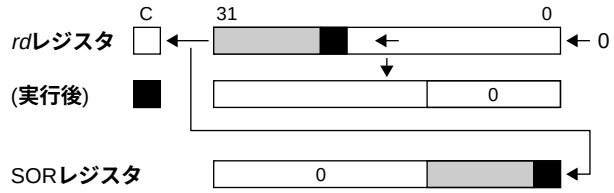
(1)標準モード(PSRのSEフラグが0の場合)

*rd*レジスタのビットを図のようにシフトさせます。ビットのシフト量は*rs*レジスタの下位5ビットによって0～31まで指定できます。*rd*レジスタの最下位ビットには0が入ります。*rd*レジスタからシフトアウトしたビットは、SORレジスタから読み出すことができます。



(2)アドバンスドモード(PSRのSEフラグが1の場合)

シフト動作は(1)の標準モードと同じですが、*rd*レジスタの最上位ビットからシフトアウトしたビットがPSRのCフラグに入ります。



Vフラグはシフト完了時のCフラグとZフラグの状態によって変化します。

C & N | !C & !N　→ V = 0

C ^ N　　→ V = 1

(3)拡張3

```
ext  %rs1
sll  %rd,%rs2
```

*rs1*レジスタのビットを*rs2*で指定されるビット数分左にシフトします。本命令はビットをシフトするソースレジスタが`ext %rs1`命令で与えられることを除き、標準またはアドバンスドモードと同じ動作となります。
*rd*レジスタの最下位ビットには0が入ります。
シフトアウトしたビットは、SORレジスタから読み出すことができます。

(4)ディレイド命令

本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。

sll *%rd, imm5*

機　能

左方向論理シフト

標準） *rd*の内容を*imm5*の指定ビット(0～31)分、左にシフト
LSB ← 0
拡張1) 不可
拡張2) 不可
拡張3) *rd* ← *rs* << *imm5*

コード

imm5(4) = 0の場合: 0～15ビットの左方向論理シフト

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	0	0	1	1	0	0	*imm5*(3:0)				*r d*				0x8C__

imm5(4) = 1の場合: 16～31ビットの左方向論理シフト

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	1	0	0	1	1	1	*imm5*(3:0)				*r d*				0x27__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	↔	↔	↔	↔

C、Vフラグは SEフラグ(PSRのビット20)が1のときに変化

モード

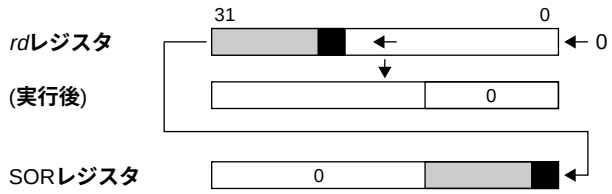
Src　即値(符号なし)
Dst　レジスタ直接 *%rd* = %r0～%r15

ＣＬＫ

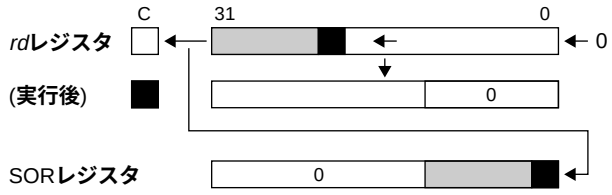
1サイクル

説　明

(1)標準モード(PSRのSEフラグが0の場合)
*rd*レジスタのビットを図のようにシフトさせます。ビットのシフト量は5ビット即値*imm5*によって0～31まで指定できます。*rd*レジスタの最下位ビットには0が入ります。*rd*レジスタからシフトアウトしたビットは、SORレジスタから読み出すことができます。



(2)アドバンスドモード(PSRのSEフラグが1の場合)
シフト動作は(1)の標準モードと同じですが、*rd*レジスタの最上位ビットからシフトアウトしたビットがPSRのCフラグに入ります。



Vフラグはシフト完了時のCフラグとZフラグの状態によって変化します。
C & N | !C & !N → V = 0
C ^ N → V = 1

(3)拡張3

```
ext  %rs
sll  %rd,imm5
```

*rs*レジスタのビットを*imm5*で指定されるビット数分左にシフトします。本命令はビットをシフトするソースレジスタが`ext %rs`命令で与えられることを除き、標準またはアドバンスドモードと同じ動作となります。
*rd*レジスタの最下位ビットには0が入ります。
シフトアウトしたビットは、SORレジスタから読み出すことができます。

(4)ディレイド命令

本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。

slp

機　能　**SLEEP**

標準）CPUをSLEEPモードに設定
拡張1）不可
拡張2）不可
拡張3）不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0

0x0040

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード　–

CLK　1サイクル

説　明　CPUをSLEEPモードにします。
これにより、CPUおよびチップ上の周辺回路は動作を停止し、消費電流を大幅に抑えることができます。
SLEEPモードは割り込みによって解除され、その割り込み処理ルーチンを実行後、slp命令の次の命令位置に戻ります。
slp命令実行後、NMIなどの外部SLEEP解除要因の発生までCPUをSLEEPモードにしておくためには、CMUの制御ビット（0x48368のビット0）を0に設定する必要があります（CMUレジスタを変更する場合は、0x4836Eのビット0～7に“0x96”を書き込んでレジスタの書き込み保護を解除してください）。
この制御ビットを1にすると、slp命令実行からある時間が経過後にCPUは自動的にSLEEPモードから起床します。詳細については、機種別テクニカルマニュアルのCMUの説明を参照してください。

例

```
slp             ; CPUをSLEEPモードに設定
```

sra *%rd, %rs*

機　能

右方向算術シフト

標準） *rd*の内容を*rs*の指定ビット(0～31)分、右にシフト
MSB ← MSB
拡張1) 不可
拡張2) 不可
拡張3) *rd* ← *rs1* >> *rs2*

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	0	1	0	0	0	1	*rs*				*rd*				0x91__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	↔	↔	↔	↔

C、Vフラグは SEフラグ(PSRのビット20)が1のときに変化

モード

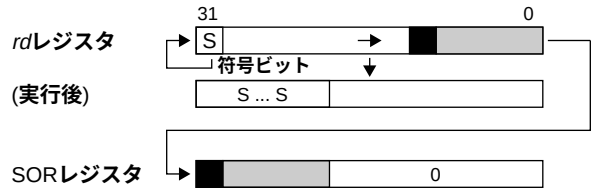
Src　レジスタ直接 *%rs* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK

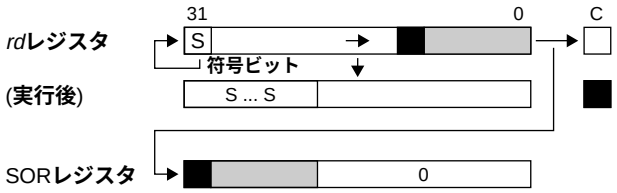
1サイクル

説　明

(1)標準モード(PSRのSEフラグが0の場合)
*rd*レジスタのビットを図のようにシフトさせます。ビットのシフト量は*rs*レジスタの下位5ビットによって0～31まで指定できます。*rd*レジスタの最上位ビットには符号ビットがコピーされます。*rd*レジスタからシフトアウトしたビットは、SORレジスタから読み出すことができます。



(2)アドバンスドモード(PSRのSEフラグが1の場合)
シフト動作は(1)の標準モードと同じですが、*rd*レジスタの最下位ビットからシフトアウトしたビットがPSRのCフラグに入ります。



Vフラグはシフト完了時のCフラグとZフラグの状態によって変化します。
C & N | !C & !N　→ V = 0
C ^ N　　→ V = 1

(3)拡張3

```
ext  %rs1
sra  %rd,%rs2
```

*rs1*レジスタのビットを*rs2*で指定されるビット数分右にシフトします。本命令はビットをシフトするソースレジスタが`ext %rs1`命令で与えられることを除き、標準またはアドバンスドモードと同じ動作となります。
*rd*レジスタの最上位ビットには符号ビットがコピーされます。
シフトアウトしたビットは、SORレジスタから読み出すことができます。

(4)ディレイド命令

本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。

sra *%rd, imm5*

機　能

右方向算術シフト

標準） *rd*の内容を*imm5*の指定ビット(0～31)分、右にシフト
MSB ← MSB

拡張1) 不可

拡張2) 不可

拡張3) *rd* ← *rs* >> *imm5*

コード

imm5(4) = 0の場合: 0～15ビットの右方向算術シフト

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	0	1	0	0	0	0	*imm5*(3:0)				*r d*				0x90__

imm5(4) = 1の場合: 16～31ビットの右方向算術シフト

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	1	0	1	0	1	1	*imm5*(3:0)				*r d*				0x2B__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	↔	↔	↔	↔

C、VフラグはSEフラグ(PSRのビット20)が1のときに変化

モード

Src　即値(符号なし)

Dst　レジスタ直接 *%rd* = %r0～%r15

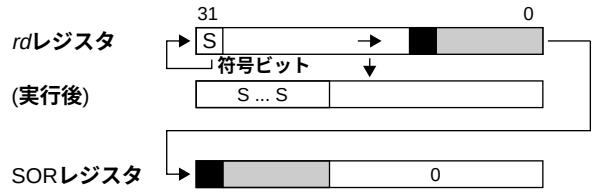
ＣＬＫ

1サイクル

説　明

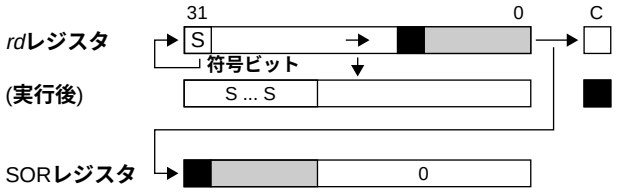
(1)標準モード(PSRのSEフラグが0の場合)

*rd*レジスタのビットを図のようにシフトさせます。ビットのシフト量は5ビット即値*imm5*によって0～31まで指定できます。*rd*レジスタの最上位ビットには符号ビットがコピーされます。*rd*レジスタからシフトアウトしたビットは、SORレジスタから読み出すことができます。



(2)アドバンスドモード(PSRのSEフラグが1の場合)

シフト動作は(1)の標準モードと同じですが、*rd*レジスタの最下位ビットからシフトアウトしたビットがPSRのCフラグに入ります。



Vフラグはシフト完了時のCフラグとZフラグの状態によって変化します。

C & N | !C & !N　→ V = 0

C ^ N　→ V = 1

(3)拡張3

```
ext  %rs
sra  %rd,imm5
```

*rs*レジスタのビットを*imm5*で指定されるビット数分右にシフトします。本命令はビットをシフトするソースレジスタが`ext %rs`命令で与えられることを除き、標準またはアドバンスドモードと同じ動作となります。
*rd*レジスタの最上位ビットには符号ビットがコピーされます。
シフトアウトしたビットは、SORレジスタから読み出すことができます。

(4)ディレイド命令

本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。

srl *%rd, %rs*

機　能

右方向論理シフト

標準） *rd*の内容を*rs*の指定ビット（0～31）分、右にシフト
MSB ← 0
拡張1）不可
拡張2）不可
拡張3）*rd* ← *rs1* >> *rs2*

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	0	0	1	*rs*				*rd*			

0x89__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	↔	↔	↔	↔

C、Vフラグは SEフラグ（PSRのビット20）が1のときに変化

モード

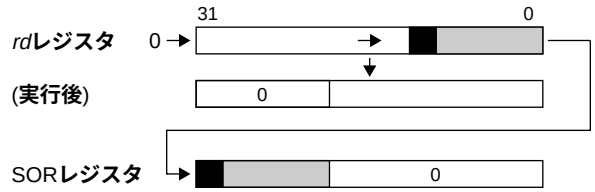
Src　レジスタ直接 *%rs* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

ＣＬＫ

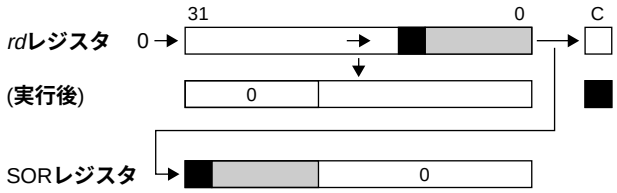
1サイクル

説　明

(1)標準モード（PSRのSEフラグが0の場合）
*rd*レジスタのビットを図のようにシフトさせます。ビットのシフト量は*rs*レジスタの下位5ビットによって0～31まで指定できます。*rd*レジスタの最上位ビットには0が入ります。*rd*レジスタからシフトアウトしたビットは、SORレジスタから読み出すことができます。



(2)アドバンスドモード（PSRのSEフラグが1の場合）
シフト動作は(1)の標準モードと同じですが、*rd*レジスタの最下位ビットからシフトアウトしたビットがPSRのCフラグに入ります。



Vフラグはシフト完了時のCフラグとZフラグの状態によって変化します。
C & N | !C & !N　→ V = 0
C ^ N　　→ V = 1

(3)拡張3

```
ext  %rs1
srl  %rd,%rs2
```

*rs1*レジスタのビットを*rs2*で指定されるビット数分右にシフトします。本命令はビットをシフトするソースレジスタが`ext %rs1`命令で与えられることを除き、標準またはアドバンスドモードと同じ動作となります。
*rd*レジスタの最上位ビットには0が入ります。
シフトアウトしたビットは、SORレジスタから読み出すことができます。

(4)ディレイド命令

本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。

srl *%rd, imm5*

機　能

右方向論理シフト

標準） *rd*の内容を*imm5*の指定ビット(0～31)分、右にシフト
MSB ← 0
拡張1) 不可
拡張2) 不可
拡張3) *rd* ← *rs* >> *imm5*

コード

imm5(4) = 0の場合: 0～15ビットの右方向論理シフト

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	0	0	1	0	0	0	*imm5*(3:0)				*r d*				0x88__

imm5(4) = 1の場合: 16～31ビットの右方向論理シフト

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	1	0	0	0	1	1	*imm5*(3:0)				*r d*				0x23__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	↔	↔	↔	↔

C、Vフラグは SEフラグ(PSRのビット20)が1のときに変化

モード

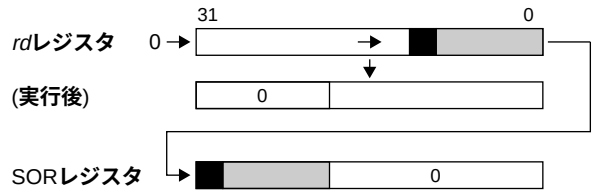
Src　即値(符号なし)
Dst　レジスタ直接 *%rd* = %r0～%r15

ＣＬＫ

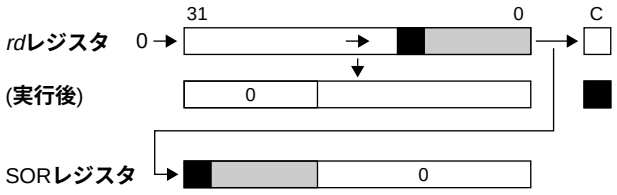
1サイクル

説　明

(1)標準モード(PSRのSEフラグが0の場合)
*rd*レジスタのビットを図のようにシフトさせます。ビットのシフト量は5ビット即値*imm5*によって0～31まで指定できます。*rd*レジスタの最上位ビットには0が入ります。*rd*レジスタからシフトアウトしたビットは、SORレジスタから読み出すことができます。



(2)アドバンスドモード(PSRのSEフラグが1の場合)
シフト動作は(1)の標準モードと同じですが、*rd*レジスタの最下位ビットからシフトアウトしたビットがPSRのCフラグに入ります。



Vフラグはシフト完了時のCフラグとZフラグの状態によって変化します。
C & N | !C & !N　→ V = 0
C ^ N　→ V = 1

(3)拡張3

```
ext  %rs
srl  %rd,imm5
```

*rs*レジスタのビットを*imm5*で指定されるビット数分右にシフトします。本命令はビットをシフトするソースレジスタが`ext %rs`命令で与えられることを除き、標準またはアドバンスドモードと同じ動作となります。
*rd*レジスタの最上位ビットには0が入ります。
シフトアウトしたビットは、SORレジスタから読み出すことができます。

(4)ディレイド命令

本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。

sub *%rd, %rs*

機　能　**減算**

標準）*rd* ← *rd - rs*
拡張1）*rd* ← *rs - imm13*
拡張2）*rd* ← *rs - imm26*
拡張3）*rd* ← *rs1 - rs2*（“*op, imm2*”使用可）

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	1	0	*rs*				*rd*			

0x26__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	↔	↔	↔	↔

モード　Src　レジスタ直接 *%rs* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK　1サイクル

説　明　(1)標準

```
sub  %rd,%rs       ; rd ← rd - rs
```

*rd*レジスタから*rs*レジスタの内容を減算します。

(2)拡張1

```
ext  imm13
sub  %rd,%rs       ; rd ← rs - imm13
```

*rs*レジスタの内容から13ビット即値*imm13*を減算し、結果を*rd*レジスタにロードします。*rs*レジスタの内容は変更されません。

(3)拡張2

```
ext  imm13         ; = imm26(25:13)
ext  imm13         ; = imm26(12:0)
sub  %rd,%rs       ; rd ← rs - imm26
```

*rs*レジスタの内容から26ビット即値*imm26*を減算し、結果を*rd*レジスタにロードします。*rs*レジスタの内容は変更されません。

(4)拡張3

```
ext  %rs2,op,imm2 ; op = sra, srl, sla, imm2 = 0–3
sub  %rd,%rs1     ; rd ← (rs1 - rs2) op imm2
```

*rs1*レジスタの内容からext命令で指定されたレジスタ*rs2*の内容を減算し、さらに*op*で指示されるシフトを*imm2*のビット数分行って結果を*rd*レジスタにロードします。*rs1*レジスタ、*rs2*レジスタの内容は変更されません。

(5)ディレイド命令

本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。この場合はext命令による拡張は行えません。

(6)ポストシフト

ポストシフト付き拡張命令の直後に記述することによって、本命令の実行結果が最大3ビットシフトされます。シフト動作は、sra, srl, sllの各命令と同じです。ただし、シフトアウトレジスタSORを使用しませんのでSORは変化しません。また、C、V、Z、Nフラグはsub命令の結果のみによって変化し、シフト動作の影響は受けません。

例

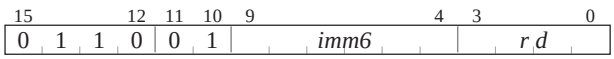
```
(1)sub  %r0,%r0      ; r0 = r0 - r0

(2)ext  0x1
   ext  0x1fff
   sub  %r1,%r2      ; r1 = r2 - 0x3fff

(3)ext  %r2,srl,1
   sub  %r3,%r1      ; r3 = (r1 - r2) >> 1
```

sub *%rd, imm6*

機　能　**減算**

標準）*rd* ← *rd* - *imm6*
拡張1）*rd* ← *rd* - *imm19*
拡張2）*rd* ← *rd* - *imm32*
拡張3）不可

コード



0x64__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
−	−	−	−	−	−	−	−	−	↔	↔	↔	↔

モード　Src　即値（符号なし）
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK　1サイクル

説　明　(1)標準

```
sub  %rd,imm6      ; rd ← rd - imm6
```

*rd*レジスタから6ビット即値*imm6*を減算します。

(2)拡張1

```
ext  imm13         ; = imm19(18:6)
sub  %rd,imm6      ; rd ← rd - imm19, imm6 = imm19(5:0)
```

*rd*レジスタからext命令により拡張した19ビット即値*imm19*を減算します。

(3)拡張2

```
ext  imm13         ; = imm32(31:19)
ext  imm13         ; = imm32(18:6)
sub  %rd,imm6      ; rd ← rd - imm32, imm6 = imm32(5:0)
```

*rd*レジスタからext命令により拡張した32ビット即値*imm32*を減算します。

(4)ディレイド命令
本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。この場合はext命令による拡張は行えません。

(5)ポストシフト（“ext *op,imm2*”のみ）
ポストシフト付き拡張命令の直後に記述することによって、本命令の実行結果が最大3ビットシフトされます。シフト動作は、sra, srl, sllの各命令と同じです。ただし、シフトアウトレジスタSORを使用しませんのでSORは変化しません。また、C、V、Z、Nフラグはsub命令の結果のみによって変化し、シフト動作の影響は受けません。

例　(1)

```
sub  %r0,0x3f      ; r0 = r0 - 0x3f
```

(2)

```
ext  0x1fff
ext  0x1fff
sub  %r1,0x3f      ; r1 = r1 - 0xffffffff
```

sub %sp, *imm10*

機　能

減算

標準）sp ← sp - *imm10*×4
拡張1）不可
拡張2）不可
拡張3）不可

コード

0x84__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
−	−	−	−	−	−	−	−	−	−	−	−	−

モード

Src　即値（符号なし）
Dst　レジスタ直接（SP）

CLK

1サイクル

説　明

(1)標準
10ビット即値*imm10*を4倍し、スタックポインタSPから減算します。

(2)ディレイド命令
本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。

(3)ポストシフト（“`ext` *op*,*imm2*”のみ）
ポストシフト付き拡張命令の直後に記述することによって、本命令の実行結果が最大3ビットシフトされます。シフト動作は、`sra`, `srl`, `sll`の各命令と同じです。ただし、シフトアウトレジスタSORを使用しませんのでSORは変化しません。また、C、V、Z、NフラグはSub命令の結果のみによって変化し、シフト動作の影響は受けません。

例

```
sub  %sp,0x100       ; sp = sp - 0x400
```

swap *%rd, %rs*

機　能　**スワップ**

標準）*rd*(31:24) ← *rs*(7:0), *rd*(23:16) ← *rs*(15:8), *rd*(15:8) ← *rs*(23:16), *rd*(7:0) ← *rs*(31:24)
拡張1）不可
拡張2）不可
拡張3）不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	0	1	0	*rs*				*rd*			

0x92__

フラグ

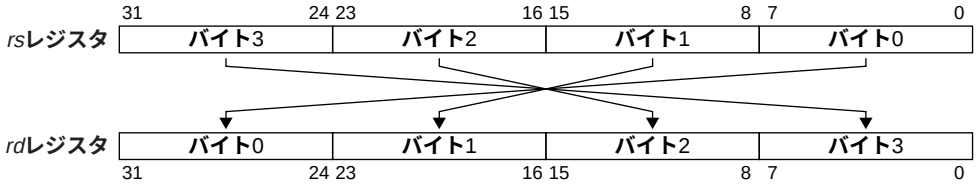
RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード　Src　レジスタ直接 *%rs* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK　1サイクル

説　明　(1)標準

*rs*レジスタのデータをバイト単位で上位と下位を入れ替え、結果を*rd*レジスタにロードします。



(2)ディレイド命令

本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。

例　r1 = 0x87654321の場合

```
swap  %r0,%r1   ; r0 ← 0x21436587
```

swaph *%rd, %rs*

機　能

スワップ

標準） *rd*(31:24) ← *rs*(23:16), *rd*(23:16) ← *rs*(31:24), *rd*(15:8) ← *rs*(7:0), *rd*(7:0) ← *rs*(15:8)
拡張1) 不可
拡張2) 不可
拡張3) 不可

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	1	0	*rs*				*rd*			

0x9A__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	–	–	–

モード

Src　レジスタ直接 *%rs* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

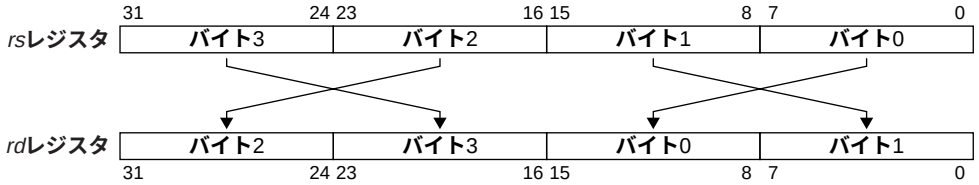
CLK

1サイクル

説　明

(1)標準

*rs*レジスタのハーフワードごとにバイトデータを入れ替え、ビッグ/リトルエンディアン変換を行います。



(2)ディレイド命令

本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。

例

r1 = 0x12345678の場合

```
swaph  %r2,%r1        ; 0x34127856 → r2
```

xor *%rd, %rs*

機　能

排他的論理和

標準） *rd* ← *rd* ^ *rs*
拡張1) *rd* ← *rs* ^ *imm13*
拡張2) *rd* ← *rs* ^ *imm26*
拡張3) *rd* ← *rs1* ^ *rs2*

コード

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	1	0	*rs*				*rd*			

0x3A__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
–	–	–	–	–	–	–	–	–	–	*1	↔	↔

モード

Src　レジスタ直接 *%rs* = %r0～%r15
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK

1サイクル

説　明

(1)標準

```
xor   %rd,%rs      ; rd ← rd ^ rs
```

*rs*レジスタの内容と*rd*レジスタの内容の排他的論理和をとり、結果を*rd*レジスタにロードします。

(2)拡張1

```
ext   imm13
xor   %rd,%rs      ; rd ← rs ^ imm13
```

*rs*レジスタの内容とゼロ拡張した13ビット即値*imm13*の排他的論理和をとり、結果を*rd*レジスタにロードします。*rs*レジスタの内容は変更されません。

(3)拡張2

```
ext   imm13        ; = imm26(25:13)
ext   imm13        ; = imm26(12:0)
xor   %rd,%rs      ; rd ← rs ^ imm26
```

*rs*レジスタの内容とゼロ拡張した26ビット即値*imm26*の排他的論理和をとり、結果を*rd*レジスタにロードします。*rs*レジスタの内容は変更されません。

(4)拡張3

```
ext   %rs2
xor   %rd,%rs1     ; rd ← rs1 ^ rs2
```

*rs1*レジスタの内容とext命令で指定されたレジスタ*rs2*の排他的論理和をとり、結果を*rd*レジスタにロードします。*rs1*レジスタ、*rs2*レジスタの内容は変更されません。

(5)ディレイド命令
本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。この場合はext命令による拡張は行えません。

---

*1 PSRのOCフラグを1にセットして本命令を実行すると、Vフラグが0にクリアされます。他の論理演算系の命令についても同様に機能します。（and命令、or命令、not命令。命令の機能は各命令の説明を参照してください。）

例

(1)
```
xor   %r0,%r0      ; r0 = r0 ^ r0
```

(2)
```
ext   0x1
ext   0x1fff
xor   %r1,%r2      ; r1 = r2 ^ 0x00003fff
```

(3)
```
ext   %r5
xor   %r3,%r4      ; r3 = r4 ^ r5
```

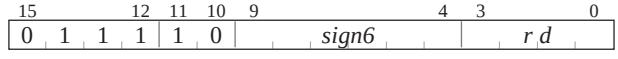
xor *%rd, sign6*

機　能

排他的論理和

標準）*rd* ← *rd* ^ *sign6*
拡張1）*rd* ← *rd* ^ *sign19*
拡張2）*rd* ← *rd* ^ *sign32*
拡張3）不可

コード



0x78__

フラグ

RM	LM	PM	RC	S	DE	ME	MO	DS	C	V	Z	N
−	−	−	−	−	−	−	−	−	−	*1	↔	↔

モード

Src　即値（符号付き）
Dst　レジスタ直接 *%rd* = %r0～%r15

CLK

1サイクル

説　明

(1)標準

```
xor  %rd,sign6     ; rd ← rd ^ sign6
```

*rd*レジスタの内容と符号拡張した6ビット即値*sign6*の排他的論理和をとり、結果を*rd*レジスタにロードします。

(2)拡張1

```
ext  imm13         ; = sign19(18:6)
xor  %rd,sign6     ; rd ← rd ^ sign19, sign6 = sign19(5:0)
```

*rd*レジスタの内容と符号拡張した19ビット即値*sign19*の排他的論理和をとり、結果を*rd*レジスタにロードします。

(3)拡張2

```
ext  imm13         ; = sign32(31:19)
ext  imm13         ; = sign32(18:6)
xor  %rd,sign6     ; rd ← rd ^ sign32, sign6 = sign32(5:0)
```

*rd*レジスタの内容と32ビット即値*sign32*の排他的論理和をとり、結果を*rd*レジスタにロードします。

(4)ディレイド命令

本命令は、dビット付きの分岐命令の直後に記述することによってディレイド命令として実行されます。この場合はext命令による拡張は行えません。

---

*1 PSRのOCフラグを1にセットして本命令を実行すると、Vフラグが0にクリアされます。他の論理演算系の命令についても同様に機能します。（and命令、or命令、not命令。命令の機能は各命令の説明を参照してください。）

例

```
(1)xor   %r0,0x3e     ; r0 = r0 ^ 0xfffffffe

(2)ext   0x7ff
   xor   %r1,0x3f     ; r1 = r1 ^ 0x0001ffff
```

APPENDIX 命令コード一覧表(コード順)

クラス0(1)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	ニーモニック	サイクル	拡張	ディレイド	ループ	リピート
クラス			op1				d	op2		0	0	*imm2,rd,rs,rb*									
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	nop	1	×	×	○	×
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	slp	1	×	×	○	×
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	halt	1	×	×	○	×
0	0	0	0	0	0	1	0	0	0	0	0	*rs*				pushn *%rs*	N	×	×	○	×
0	0	0	0	0	0	1	0	0	1	0	0	*rd*				popn *%rd*	N	×	×	○	×
0	0	0	0	0	0	1	0	1	1	0	0	*rb*				**jpr *%rb***	4	×	×	∗1	×
0	0	0	0	0	0	1	1	1	1	0	0	*rb*				**jpr.d *%rb***	3	×	×	∗2	×
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	brk	7	×	×	∗1	×
0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	retd	6	×	×	×	×
0	0	0	0	0	1	0	0	1	0	0	0	0	0	*imm2*		int *imm2*	7	×	×	∗1	×
0	0	0	0	0	1	0	0	1	1	0	0	0	0	0	0	reti	6	×	×	×	×
0	0	0	0	0	1	1	0	0	0	0	0	*rb*				call *%rb*	3	×	×	∗1	×
0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	ret	5	×	×	×	×
0	0	0	0	0	1	1	0	1	0	0	0	*rb*				jp *%rb*	3	×	×	∗1	×
0	0	0	0	0	1	1	0	1	1	0	0	0	0	0	0	**retm**	6	×	×	×	×
0	0	0	0	0	1	1	1	0	0	0	0	*rb*				call.d *%rb*	2	×	×	∗2	×
0	0	0	0	0	1	1	1	0	1	0	0	0	0	0	0	ret.d	4	×	×	×	×
0	0	0	0	0	1	1	1	1	0	0	0	*rb*				jp.d *%rb*	2	×	×	∗2	×

クラス0(2)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	ニーモニック	サイクル	拡張	ディレイド	ループ	リピート
クラス			op1				op2			0	1	*imm4,r,s*									
0	0	0	0	0	0	0	0	0	0	0	1	*rs*				**push *%rs***	1	×	×	○	○
0	0	0	0	0	0	0	0	0	1	0	1	*rd*				**pop *%rd***	1	×	×	○	○
0	0	0	0	0	0	0	0	1	0	0	1	*ss*				**pushs *%ss***	N	×	×	○	×
0	0	0	0	0	0	0	0	1	1	0	1	*sd*				**pops *%sd***	N	×	×	∗3	×
0	0	0	0	0	0	0	1	0	0	0	1	*rs*				**mac.w *%rs***	3 + N × 2	×	×	○	×
0	0	0	0	0	0	0	1	0	1	0	1	*rs*				**mac.hw *%rs***	2 + N × 2	×	×	○	×
0	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	**macclr**	1	×	○	○	○
0	0	0	0	0	0	0	1	1	1	0	1	0	0	0	0	**ld.cf**	1	×	○	○	○
0	0	0	0	0	0	1	0	0	0	0	1	*rs*				**divu.w *%rs***	35	×	×	○	×
0	0	0	0	0	0	1	0	0	1	0	1	*rs*				**div.w *%rs***	35	×	×	○	×
0	0	0	0	0	0	1	0	1	0	0	1	*rc*				**repeat *%rc***	4	×	×	×	×
0	0	0	0	0	0	1	0	1	1	0	1	*imm4*				**repeat *imm4***	4	×	×	×	×
0	0	0	0	0	0	1	1	0	1	0	1	*rd*				**add *%rd*,%dp**	1	○	○	○	○

クラス0(3)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	ニーモニック	サイクル	拡張	ディレイド	ループ	リピート
クラス			op1				d	*sign8*													
0	0	0	0	1	0	0	0	*sign8*								jrgt *sign8*	1–2	○	×	∗1	×
0	0	0	0	1	0	0	1	*sign8*								jrgt.d *sign8*	1	○	×	∗2	×
0	0	0	0	1	0	1	0	*sign8*								jrge *sign8*	1–2	○	×	∗1	×
0	0	0	0	1	0	1	1	*sign8*								jrge.d *sign8*	1	○	×	∗2	×
0	0	0	0	1	1	0	0	*sign8*								jrlt *sign8*	1–2	○	×	∗1	×
0	0	0	0	1	1	0	1	*sign8*								jrlt.d *sign8*	1	○	×	∗2	×
0	0	0	0	1	1	1	0	*sign8*								jrle *sign8*	1–2	○	×	∗1	×
0	0	0	0	1	1	1	1	*sign8*								jrle.d *sign8*	1	○	×	∗2	×
0	0	0	1	0	0	0	0	*sign8*								jrugt *sign8*	1–2	○	×	∗1	×
0	0	0	1	0	0	0	1	*sign8*								jrugt.d *sign8*	1	○	×	∗2	×
0	0	0	1	0	0	1	0	*sign8*								jruge *sign8*	1–2	○	×	∗1	×
0	0	0	1	0	0	1	1	*sign8*								jruge.d *sign8*	1	○	×	∗2	×
0	0	0	1	0	1	0	0	*sign8*								jrult *sign8*	1–2	○	×	∗1	×
0	0	0	1	0	1	0	1	*sign8*								jrult.d *sign8*	1	○	×	∗2	×
0	0	0	1	0	1	1	0	*sign8*								jrule *sign8*	1–2	○	×	∗1	×
0	0	0	1	0	1	1	1	*sign8*								jrule.d *sign8*	1	○	×	∗2	×
0	0	0	1	1	0	0	0	*sign8*								jreq *sign8*	1–2	○	×	∗1	×
0	0	0	1	1	0	0	1	*sign8*								jreq.d *sign8*	1	○	×	∗2	×
0	0	0	1	1	0	1	0	*sign8*								jrne *sign8*	1–2	○	×	∗1	×
0	0	0	1	1	0	1	1	*sign8*								jrne.d *sign8*	1	○	×	∗2	×
0	0	0	1	1	1	0	0	*sign8*								call *sign8*	2	○	×	∗1	×
0	0	0	1	1	1	0	1	*sign8*								call.d *sign8*	1	○	×	∗2	×
0	0	0	1	1	1	1	0	*sign8*								jp *sign8*	2	○	×	∗1	×
0	0	0	1	1	1	1	1	*sign8*								jp.d *sign8*	1	○	×	∗2	×

クラス1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	ニーモニック		サイクル	拡張	ディレイド	ループ	リピート
クラス			op1			op2		*imm5,rb,rs*				*imm2,rs,rd*										
0	0	1	0	0	0	0	0	*rb*				*rd*				`ld.b`	*`%rd,[%rb]`*	1	○	×	○	○
0	0	1	0	0	0	0	1	*rb*				*rd*				`ld.b`	*`%rd,[%rb]+`*	1	○	×	○	○
0	0	1	0	0	0	1	0	*rs*				*rd*				`add`	*`%rd,%rs`*	1	○	○	○	○
0	0	1	0	0	0	1	1	*imm5*(3:0)				*rd*				**`srl`**	***`%rd,imm5`***	1	△	○	○	○
0	0	1	0	0	1	0	0	*rb*				*rd*				`ld.ub`	*`%rd,[%rb]`*	1	○	×	○	○
0	0	1	0	0	1	0	1	*rb*				*rd*				`ld.ub`	*`%rd,[%rb]+`*	1	○	×	○	○
0	0	1	0	0	1	1	0	*rs*				*rd*				`sub`	*`%rd,%rs`*	1	○	○	○	○
0	0	1	0	0	1	1	1	*imm5*(3:0)				*rd*				**`sll`**	***`%rd,imm5`***	1	△	○	○	○
0	0	1	0	1	0	0	0	*rb*				*rd*				`ld.h`	*`%rd,[%rb]`*	1	○	×	○	○
0	0	1	0	1	0	0	1	*rb*				*rd*				`ld.h`	*`%rd,[%rb]+`*	1	○	×	○	○
0	0	1	0	1	0	1	0	*rs*				*rd*				`cmp`	*`%rd,%rs`*	1	○	○	○	○
0	0	1	0	1	0	1	1	*imm5*(3:0)				*rd*				**`sra`**	***`%rd,imm5`***	1	△	○	○	○
0	0	1	0	1	1	0	0	*rb*				*rd*				`ld.uh`	*`%rd,[%rb]`*	1	○	×	○	○
0	0	1	0	1	1	0	1	*rb*				*rd*				`ld.uh`	*`%rd,[%rb]+`*	1	○	×	○	○
0	0	1	0	1	1	1	0	*rs*				*rd*				`ld.w`	*`%rd,%rs`*	1	×	○	○	○
0	0	1	0	1	1	1	1	*imm5*(3:0)				*rd*				**`sla`**	***`%rd,imm5`***	1	△	○	○	○
0	0	1	1	0	0	0	0	*rb*				*rd*				`ld.w`	*`%rd,[%rb]`*	1	○	×	○	○
0	0	1	1	0	0	0	1	*rb*				*rd*				`ld.w`	*`%rd,[%rb]+`*	1	○	×	○	○
0	0	1	1	0	0	1	0	*rs*				*rd*				`and`	*`%rd,%rs`*	1	○	○	○	○
0	0	1	1	0	0	1	1	*imm5*(3:0)				*rd*				**`rr`**	***`%rd,imm5`***	1	△	○	○	○
0	0	1	1	0	1	0	0	*rb*				*rs*				`ld.b`	*`[%rb],%rs`*	1	○	×	○	○
0	0	1	1	0	1	0	1	*rb*				*rs*				`ld.b`	*`[%rb]+,%rs`*	1	○	×	○	○
0	0	1	1	0	1	1	0	*rs*				*rd*				`or`	*`%rd,%rs`*	1	○	○	○	○
0	0	1	1	0	1	1	1	*imm5*(3:0)				*rd*				**`rl`**	***`%rd,imm5`***	1	△	○	○	○
0	0	1	1	1	0	0	0	*rb*				*rs*				`ld.h`	*`[%rb],%rs`*	1	○	×	○	○
0	0	1	1	1	0	0	1	*rb*				*rs*				`ld.h`	*`[%rb]+,%rs`*	1	○	×	○	○
0	0	1	1	1	0	1	0	*rs*				*rd*				`xor`	*`%rd,%rs`*	1	○	○	○	○
0	0	1	1	1	0	1	1	0	0	0	0	0	1	*imm2*		**`ext`**	**`sra,`*`imm2`**	0–1	×	×	∗1	×
0	0	1	1	1	0	1	1	0	0	0	0	1	0	*imm2*		**`ext`**	**`srl,`*`imm2`**	0–1	×	×	∗1	×
0	0	1	1	1	0	1	1	0	0	0	0	1	1	*imm2*		**`ext`**	**`sll,`*`imm2`**	0–1	×	×	∗1	×
0	0	1	1	1	0	1	1	0	1	0	0	0	0	0	0	**`ext`**	**`gt`**	0–1	×	×	∗1	×
0	0	1	1	1	0	1	1	0	1	0	1	0	0	0	0	**`ext`**	**`ge`**	0–1	×	×	∗1	×
0	0	1	1	1	0	1	1	0	1	1	0	0	0	0	0	**`ext`**	**`lt`**	0–1	×	×	∗1	×
0	0	1	1	1	0	1	1	0	1	1	1	0	0	0	0	**`ext`**	**`le`**	0–1	×	×	∗1	×
0	0	1	1	1	0	1	1	1	0	0	0	0	0	0	0	**`ext`**	**`ugt`**	0–1	×	×	∗1	×
0	0	1	1	1	0	1	1	1	0	0	1	0	0	0	0	**`ext`**	**`uge`**	0–1	×	×	∗1	×
0	0	1	1	1	0	1	1	1	0	1	0	0	0	0	0	**`ext`**	**`ult`**	0–1	×	×	∗1	×
0	0	1	1	1	0	1	1	1	0	1	1	0	0	0	0	**`ext`**	**`ule`**	0–1	×	×	∗1	×
0	0	1	1	1	0	1	1	1	1	0	0	0	0	0	0	**`ext`**	**`eq`**	0–1	×	×	∗1	×
0	0	1	1	1	0	1	1	1	1	0	1	0	0	0	0	**`ext`**	**`ne`**	0–1	×	×	∗1	×
0	0	1	1	1	1	0	0	*rb*				*rs*				`ld.w`	*`[%rb],%rs`*	1	○	×	○	○
0	0	1	1	1	1	0	1	*rb*				*rs*				`ld.w`	*`[%rb]+,%rs`*	1	○	×	○	○
0	0	1	1	1	1	1	0	*rs*				*rd*				`not`	*`%rd,%rs`*	1	×	○	○	○
0	0	1	1	1	1	1	1	*rs*				0	0	0	0	**`ext`**	***`%rs`***	0–1	×	×	∗1	×
0	0	1	1	1	1	1	1	*rs*				0	1	*imm2*		**`ext`**	***`%rs,`*`sra,`*`imm2`**	0–1	×	×	∗1	×
0	0	1	1	1	1	1	1	*rs*				1	0	*imm2*		**`ext`**	***`%rs,`*`srl,`*`imm2`**	0–1	×	×	∗1	×
0	0	1	1	1	1	1	1	*rs*				1	1	*imm2*		**`ext`**	***`%rs,`*`sll,`*`imm2`**	0–1	×	×	∗1	×

クラス2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	ニーモニック		サイクル	拡張	ディレイド	ループ	リピート
クラス			op1			*imm6*						*rs,rd*										
0	1	0	0	0	0	*imm6*						*rd*				`ld.b`	*`%rd,[%sp+imm6]`*	1	○	×	○	○
0	1	0	0	0	1	*imm6*						*rd*				`ld.ub`	*`%rd,[%sp+imm6]`*	1	○	×	○	○
0	1	0	0	1	0	*imm6*						*rd*				`ld.h`	*`%rd,[%sp+imm6]`*	1	○	×	○	○
0	1	0	0	1	1	*imm6*						*rd*				`ld.uh`	*`%rd,[%sp+imm6]`*	1	○	×	○	○
0	1	0	1	0	0	*imm6*						*rd*				`ld.w`	*`%rd,[%sp+imm6]`*	1	○	×	○	○
0	1	0	1	0	1	*imm6*						*rs*				`ld.b`	*`[%sp+imm6],%rs`*	1	○	×	○	○
0	1	0	1	1	0	*imm6*						*rs*				`ld.h`	*`[%sp+imm6],%rs`*	1	○	×	○	○
0	1	0	1	1	1	*imm6*						*rs*				`ld.w`	*`[%sp+imm6],%rs`*	1	○	×	○	○

クラス3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	ニーモニック		サイクル	拡張	ディレイド	ループ	リピート
クラス			op1			*imm6,sign6*						*rd*										
0	1	1	0	0	0	*imm6*						*rd*				add	*%rd,imm6*	1	○	○	○	○
0	1	1	0	0	1	*imm6*						*rd*				sub	*%rd,imm6*	1	○	○	○	○
0	1	1	0	1	0	*sign6*						*rd*				cmp	*%rd,sign6*	1	○	○	○	○
0	1	1	0	1	1	*sign6*						*rd*				ld.w	*%rd,sign6*	1	○	○	○	○
0	1	1	1	0	0	*sign6*						*rd*				and	*%rd,sign6*	1	○	○	○	○
0	1	1	1	0	1	*sign6*						*rd*				or	*%rd,sign6*	1	○	○	○	○
0	1	1	1	1	0	*sign6*						*rd*				xor	*%rd,sign6*	1	○	○	○	○
0	1	1	1	1	1	*sign6*						*rd*				not	*%rd,sign6*	1	○	○	○	○

クラス4(1)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	ニーモニック		サイクル	拡張	ディレイド	ループ	リピート
クラス			op1			*imm10*																
1	0	0	0	0	0	*imm10*										add	%sp,*imm10*	1	×	○	○	○
1	0	0	0	0	1	*imm10*										sub	%sp,*imm10*	1	×	○	○	○

クラス4(2)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	ニーモニック		サイクル	拡張	ディレイド	ループ	リピート
クラス			op1			op2		*imm5,rs*				0,*rd*										
1	0	0	0	1	0	0	0	*imm5*(3:0)				*rd*				**srl**	***%rd,imm5***	1	△	○	○	○
1	0	0	0	1	0	0	1	*rs*				*rd*				**srl**	***%rd,%rs***	1	△	○	○	○
1	0	0	0	1	0	1	0	*rs*				*rd*				**scan0**	***%rd,%rs***	1	×	○	○	○
1	0	0	0	1	0	1	1	*rs*				0	0	0	0	div0s	*%rs*	1	×	×	○	○
1	0	0	0	1	1	0	0	*imm5*(3:0)				*rd*				**sll**	***%rd,imm5***	1	△	○	○	○
1	0	0	0	1	1	0	1	*rs*				*rd*				**sll**	***%rd,%rs***	1	△	○	○	○
1	0	0	0	1	1	1	0	*rs*				*rd*				**scan1**	***%rd,%rs***	1	×	○	○	○
1	0	0	0	1	1	1	1	*rs*				0	0	0	0	div0u	*%rs*	1	×	×	○	○
1	0	0	1	0	0	0	0	*imm5*(3:0)				*rd*				**sra**	***%rd,imm5***	1	△	○	○	○
1	0	0	1	0	0	0	1	*rs*				*rd*				**sra**	***%rd,%rs***	1	△	○	○	○
1	0	0	1	0	0	1	0	*rs*				*rd*				swap	*%rd,%rs*	1	×	○	○	○
1	0	0	1	0	0	1	1	*rs*				0	0	0	0	div1	*%rs*	1	×	×	○	○
1	0	0	1	0	1	0	0	*imm5*(3:0)				*rd*				**sla**	***%rd,imm5***	1	△	○	○	○
1	0	0	1	0	1	0	1	*rs*				*rd*				**sla**	***%rd,%rs***	1	△	○	○	○
1	0	0	1	0	1	1	0	*rs*				*rd*				mirror	*%rd,%rs*	1	×	○	○	○
1	0	0	1	0	1	1	1	*rs*				0	0	0	0	div2s	*%rs*	1	×	×	○	○
1	0	0	1	1	0	0	0	*imm5*(3:0)				*rd*				**rr**	***%rd,imm5***	1	△	○	○	○
1	0	0	1	1	0	0	1	*rs*				*rd*				**rr**	***%rd,%rs***	1	△	○	○	○
1	0	0	1	1	0	1	0	*rs*				*rd*				**swaph**	***%rd,%rs***	1	×	○	○	○
1	0	0	1	1	0	1	1	*rs*				0	0	0	0	div3s		1	×	×	○	○
1	0	0	1	1	1	0	0	*imm5*(3:0)				*rd*				**rl**	***%rd,imm5***	1	△	○	○	○
1	0	0	1	1	1	0	1	*rs*				*rd*				**rl**	***%rd,%rs***	1	△	○	○	○
1	0	0	1	1	1	1	0	*rs*				*rd*				**sat.b**	***%rd,%rs***	1	×	○	○	○
1	0	0	1	1	1	1	1	*rs*				*rd*				**sat.ub**	***%rd,%rs***	1	×	○	○	○

クラス5(1)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	ニーモニック		サイクル	拡張	ディレイド	ループ	リピート
クラス			op1			op2		*imm4,r,s*				0,*imm3,r,s*										
1	0	1	0	0	0	0	0	*rs*				*sd*				**ld.w**	***%sd,%rs***	1	×	○	○	○
1	0	1	0	0	0	0	1	*rs*				*rd*				ld.b	*%rd,%rs*	1	×	○	○	○
1	0	1	0	0	0	1	0	*rs*				*rd*				mlt.h	*%rd,%rs*	1	×	○	○	○
1	0	1	0	0	0	1	1	*rs*				*rd*				**mlt.hw**	***%rd,%rs***	2	×	×	○	○
1	0	1	0	0	1	0	0	*ss*				*rd*				**ld.w**	***%rd,%ss***	1	×	○	○	○
1	0	1	0	0	1	0	1	*rs*				*rd*				ld.ub	*%rd,%rs*	1	×	○	○	○
1	0	1	0	0	1	1	0	*rs*				*rd*				mltu.h	*%rd,%rs*	1	×	○	○	○
1	0	1	0	0	1	1	1	*rs*				*rd*				**mac1.h**	***%rd,%rs***	1	×	×	○	○
1	0	1	0	1	0	0	0	*rb*				0	*imm3*			btst	*[%rb],imm3*	3	○	×	○	○
1	0	1	0	1	0	0	1	*rs*				*rd*				ld.h	*%rd,%rs*	1	×	○	○	○
1	0	1	0	1	0	1	0	*rs*				*rd*				mlt.w	*%rd,%rs*	2	×	×	○	○
1	0	1	0	1	0	1	1	*rs*				*rd*				**mac1.hw**	***%rd,%rs***	2	×	×	○	○
1	0	1	0	1	1	0	0	*rb*				0	*imm3*			bclr	*[%rb],imm3*	3	○	×	○	○
1	0	1	0	1	1	0	1	*rs*				*rd*				ld.uh	*%rd,%rs*	1	×	○	○	○
1	0	1	0	1	1	1	0	*rs*				*rd*				mltu.w	*%rd,%rs*	2	×	×	○	○
1	0	1	1	0	0	0	0	*rb*				0	*imm3*			bset	*[%rb],imm3*	3	○	×	○	○
1	0	1	1	0	0	0	1	*imm4*				*rd*				**ld.c**	***%rd,imm4***	1	×	○	○	○
1	0	1	1	0	0	1	0	*rs*				0	0	0	0	mac	*%rs*	2 + N × 2	×	×	○	×
1	0	1	1	0	0	1	1	*rs*				*rd*				**mac1.w**	***%rd,%rs***	2	×	×	○	○
1	0	1	1	0	1	0	0	*rb*				0	*imm3*			bnot	*[%rb],imm3*	3	○	×	○	○
1	0	1	1	0	1	0	1	*imm4*				*rs*				**ld.c**	***imm4,%rs***	1	×	○	○	○
1	0	1	1	0	1	1	0	*rs*				*rd*				**sat.h**	***%rd,%rs***	1	×	○	○	○
1	0	1	1	0	1	1	1	*rs*				*rd*				**sat.uh**	***%rd,%rs***	1	×	○	○	○
1	0	1	1	1	0	0	0	*rs*				*rd*				**adc**	***%rd,%rs***	1	△	○	○	○
1	0	1	1	1	0	0	1	*ra*				*rc*				**loop**	***%rc,%ra***	5	×	×	×	×
1	0	1	1	1	0	1	0	*imm4*				*rc*				**loop**	***%rc,imm4***	5	×	×	×	×
1	0	1	1	1	0	1	1	*imm4(addr)*				*imm4(count)*				**loop**	***imm4,imm4***	5	×	×	×	×
1	0	1	1	1	1	0	0	*rs*				*rd*				**sbc**	***%rd,%rs***	1	△	○	○	○
1	0	1	1	1	1	0	1	*rs*				*rd*				**sat.w**	***%rd,%rs***	1	×	○	○	○
1	0	1	1	1	1	1	0	*rs*				*rd*				**sat.uw**	***%rd,%rs***	1	×	○	○	○

クラス5(2)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	ニーモニック		サイクル	拡張	ディレイド	ループ	リピート
クラス			op1			op2		op3		*imm5,imm6*												
1	0	1	1	1	1	1	1	0	0	*imm6*						**do.c**	***imm6***	1	×	×	○	○
1	0	1	1	1	1	1	1	0	1	0	*imm5*					**psrset**	***imm5***	4	×	○	○	○
1	0	1	1	1	1	1	1	1	0	0	*imm5*					**psrclr**	***imm5***	4	×	○	○	○

クラス6

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	ニーモニック		サイクル	拡張	ディレイド	ループ	リピート
クラス			*imm13*																			
1	1	0	*imm13*													**ext**	***imm13***	0–1	×	×	*1	×

クラス7

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	ニーモニック		サイクル	拡張	ディレイド	ループ	リピート
クラス			op1			*imm6*						*rs,rd*										
1	1	1	0	0	0	*imm6*						*rd*				**ld.b**	***%rd,[%dp+imm6]***	1	○	×	○	○
1	1	1	0	0	1	*imm6*						*rd*				**ld.ub**	***%rd,[%dp+imm6]***	1	○	×	○	○
1	1	1	0	1	0	*imm6*						*rd*				**ld.h**	***%rd,[%dp+imm6]***	1	○	×	○	○
1	1	1	0	1	1	*imm6*						*rd*				**ld.uh**	***%rd,[%dp+imm6]***	1	○	×	○	○
1	1	1	1	0	0	*imm6*						*rd*				**ld.w**	***%rd,[%dp+imm6]***	1	○	×	○	○
1	1	1	1	0	1	*imm6*						*rs*				**ld.b**	***[%dp+imm6],%rs***	1	○	×	○	○
1	1	1	1	1	0	*imm6*						*rs*				**ld.h**	***[%dp+imm6],%rs***	1	○	×	○	○
1	1	1	1	1	1	*imm6*						*rs*				**ld.w**	***[%dp+imm6],%rs***	1	○	×	○	○

Inst C33 ADVで機能が拡張された命令

Inst C33 ADVで追加された命令

△ 拡張3(3オペランド拡張)のみ有効

*1 ループエンドアドレス(LEA)となる位置に配置することはできません。

*2 ループエンドアドレス(LEA)とLEA - 2となる位置に配置することはできません。

*3 loop命令で使用しているレジスタ(LSA、LEA、LCO)のポップはできません。

セイコーエプソン株式会社
電子デバイス営業本部

〈ED東日本営業部〉
東京　〒191-8501　東京都日野市日野421-8
TEL（042）587-5313（直通）　FAX（042）587-5116

仙台　〒980-0013　宮城県仙台市青葉区花京院1-1-20　花京院スクエア19F
TEL（022）263-7975（代表）　FAX（022）263-7990

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

名古屋　〒461-0005　名古屋市東区東桜1-10-24　栄大野ビル4F
TEL（052）953-8031（代表）　FAX（052）953-8041

インターネットによる電子デバイスのご紹介　http://www.epsondevice.com/domcfg.nsf

ドキュメントコード：405162600

2004年10月作成 ⓁB